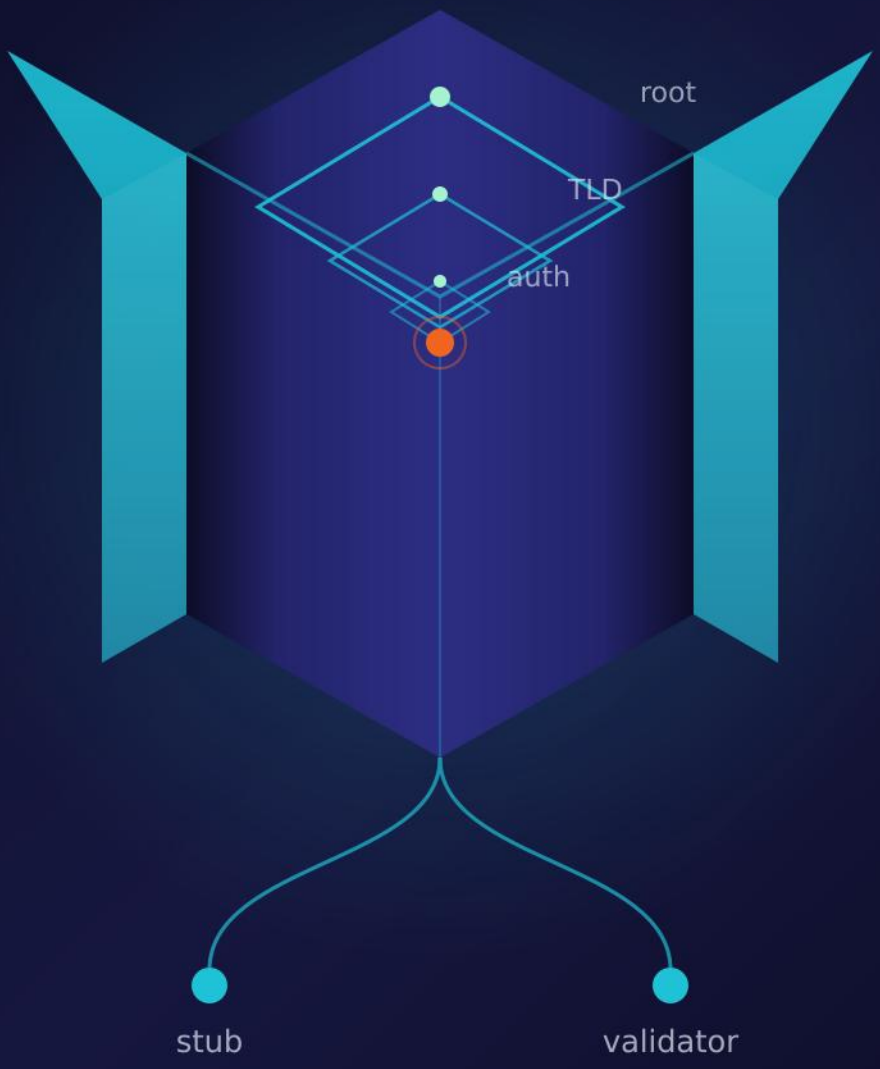


DNSSEC · DoT · DoH · DoQ · QMIN

DNS的加密和隐私防护

从如何加固 Unbound 说起

DNS Encryption and Privacy · 从加固 Unbound 说起

@LeisureLinux

CC BY-SA 4.0

DNS 的加密和隐私防护——从如何加固 Unbound 说起

DNS Encryption and Privacy –Hardening Unbound from the Ground Up

B 站频道，微信公众号：@LeisureLinux

2026 年 11 月·v0.7.3

图书介绍

0.1 这本书讲什么

每一次联网都从一句提问开始：这个名字在哪里。DNS 把这句提问送出去，再把答案带回来，然后才有网页、支付、推送和视频。绝大多数人对它的了解止于「填一个递归解析器的地址」；而这本书关心的是地址填完之后发生的事——谁替你从根开始迭代、应答有没有在路上被改过、链路上有谁看得见完整查询名、缓存把旧答案留了多久、以及当加密端点不在自己手里时，本机那些策略还剩多少仍然生效。

书名点明了两条主线：**加密**解决「路上谁能看见」，**隐私防护**解决「这些看见能不能被收拢成一份关于你的档案」；而 Unbound 是贯穿全书的落点——一台边界清楚、只做递归验证、可以按角色拆成多个实例的解析器。从加固它说起，是因为所有抽象的判断最终都要落到一份能跑起来、能验证、坏了能退回来的配置上。

0.2 全书结构：五篇分别讲什么

第一篇认识 DNS 与 Unbound（第 1–3 章）先把地基铺平：Unbound 为什么被设计成「不做权威服务器」，存根、缓存、转发、递归四种角色分别承担什么，以及 RD / RA / AD / CD 这些报文位到底在宣告谁的责任。这一篇不贴长配置，只建立后面所有章节共用的语言。

第二篇按角色配置 Unbound（第 4–7 章）是动手的起点：十行起步的缓存解析器、单上游且 fail-closed 的转发器、带本地根区镜像的递归解析器（含 NSD 根镜像与按 uid 豁免出向 53），以及多实例下的端口规划、过滤表内存与 dnstap 观测。每个配置块都紧跟「为什么这么设」「一条可复制的验证命令」「失败模式与回滚点」三件事。

第三篇加固（第 8–14 章）是全书最重的一篇。用 NIST SP 800-81r3 的框架画出信任边界，再逐层展开：最小暴露面与访问控制、DNSSEC 验证的三种状态、DoT 服务端、DoH 与 Web 服务共 443 的前端组合、查询名最小化与防降级、DDR 压制、以及加密通道选路与 nftables 劫持的边界。这里反复钉死一条区分：**DNSSEC 保证数据完整性与来源认证，DoT / DoH / DoQ 保证传输加密与链路隐私，两者不能互相替代**。

第四篇 RFC 与协议理论（第 15–18 章）回到条文，解释参数为什么是现在这个样子：基础 RFC 族、DNSSEC 族、加密与隐私族、以及 IPv6 / RA / RDNSS 在不同实现里的处理差异。理论为主，不重复贴配置。

第五篇工程实践（第 19–37 章）收束到现场：IPv6 的 RA / RDNSS / ULA 与前缀轮换、Kubernetes 里的 DNS 选型与 `ndots`、anycast 集群与故障域隔离、DHCP 交接面、隧道与扫描这类攻击面、蜜罐与被动观测、以及一套六层的排查工具箱。第 37 章单独讨论 HTTPDNS——一种不属于 DNS 协议体系、却真实改变了 App 解析路径的形态，以及想用 Unbound 自建时该怎么搭。

0.3 为什么要讲安全加固

DNS 是 1983 年定下骨架的协议，默认明文、默认信任链路。一台配错的递归解析器会被拿去做放大攻击；一次上游瞬时 NXDOMAIN 会按负缓存规则被放大成小时级故障；一个 `qname-minimisation-strict: yes` 就能撞上违反 RFC 8020 的 CDN 区，让整站静态资源解析失败。这些都不是「参数没背熟」，而是**加固没做在正确的那一层**：缓存、验证、过滤、加密、故障域，每一层的失效方式都不同，排查时若在错误的层刷缓存、找 AD 位，就会白花一整天。

转发器上不配信任锚、多 `forward-addr` 是竞速不是主备、上游超时不许静默降级回明文 53、本机解析器进程的出向 53 必须按 uid 豁免——每一条都来自实际踩过的坑，也都给出回滚点。

0.4 为什么必须讲隐私防护

过去想知道一个人的行踪，得在他家门口装一台摄像头，或者用 GPS 跟踪他的行动轨迹。现在不需要了：**只要把他访问过的目标网站按时间排起来，就足以拼出他的作息、健康状况、财务往来、家庭关系与政治倾向**。而 DNS 查询正是这份清单的原始形态——它发生在任何连接建立之前，包含完整的目标名字，且默认不加密。

更关键的是，看见这份清单的往往不止一方：运营商的 LocalDNS、链路上任何一个被动观察者、被托付加密端点的服务运营者、以及自己门口那份没有设定保留期的 dnstap 日志。第 8 章的威胁模型把「谁看得见什么」逐条列出来；第 13、17 章讲查询名最小化、加密传输与 DDR 压制各自能挡住哪一段；第 12 章则提醒，把 DoH 端点放到别人的机器上，等于把本地策略一起交了出去。

隐私防护不是锦上添花，而是这本书从「能跑」走向「可交付」的分界线。加密只保护一段链路，不保证数据来源可信；DNSSEC 只保证数据没被改过，一个字节都不隐藏。两者叠加，再加上把解析器握在自己手里，才谈得上「既不被偷看、也不被篡改」。这也是书名把「加密」和「隐私防护」并列的原因。

0.5 适合哪些读者

- **有 Linux 基础的入门者 / 家庭网关与自托管玩家**：想让全屋设备的解析干净、可过滤、可观测，且不想把查询交给第三方。从第 4 章十行起步，按篇顺序读即可。
- **运维与 SRE**：要在一台机器上跑多个解析器实例、按域名或信任域分流、处理上游抖动与缓存放大。重点看第 5–7、26、28、31、36 章。
- **安全工程师**：要画清 DNS 的信任边界、对齐 NIST 部署基线、判断加密传输与 DNSSEC 各自的覆盖面。重点看第 8–14、29、34、35 章。
- **Kubernetes / 云原生运维**：要理解 CoreDNS 与 Unbound 的分工、`ndots`、NodeLocal DNSCache 与出口验证。看第 20 章，再回第 3 章补角色划分。
- **移动端与 App 架构从业者**：关心 LocalDNS 劫持、精准调度、私有解析 API 与 SDK 改造代价。看第 32、37 章。
- **想读 RFC 但不知从哪下手的读者**：第四篇按族给出条文逻辑，可当索引用。

需要的基础：会看 `dig` 输出、能改 systemd 单元、知道 IPv4 / IPv6 子网是怎么回事。**不需要读过 RFC**，也不需要事先用过 Unbound。

版权信息

项	内容
书名	DNS 的加密和隐私防护——从如何加固 Unbound 说起
副标题	*DNS Encryption and Privacy — Hardening Unbound from the Ground Up*
作者	@LeisureLinux
发布	2026 年 11 月，v0.7.3
正文许可	CC BY-SA 4.0
构建脚本许可	MIT
出版形态	开源中文电子书，PDF / ePub / 独立 HTML 由 Pandoc 从 Markdown 自动构建

正文以 CC BY-SA 4.0 发布。可以自由复制、修改、再发布，包括商业用途，前提是署名并以相同许可共享。仓库里的构建脚本沿用 MIT 模板。

引用第三方材料时只使用其事实与结论并注明出处，不复制受版权保护的正文。 Unbound 官方文档为 CC BY 3.0，NIST 出版物属美国政府公有领域作品，RFC 文本与厂商公开文档各自保留自己的许可——它们不因被本书引用而改变。各章引用的厂商产品文档、学术论文与运维 BCP，均在章末《参考资料》列出可核验的原始 URL。

商标与产品名：BIND、Unbound、NSD、PowerDNS、Knot Resolver、CoreDNS、dnsmasq、Kubernetes、NIST、以及书中出现的云厂商与移动平台名称，均为各自权利人的商标或产品名。本书为独立技术写作，与上述项目无隶属、无背书关系；封面图形是依据官方配色调性重新演绎的原创示意图，不是官方标识。

责任范围：书中的配置、命令与架构建议按「可复现、可验证、有回滚点」给出，但不构成针对特定网络环境的安全承诺或合规意见。在生产环境变更前请先在实验环境验证，并保留配置备份。

本书不设纸质书号，不通过商业渠道发行。最新版与构建产物见项目仓库与在线阅读站点；发现错漏欢迎通过仓库 Issue 反馈。

作者简介

LeisureLinux，B 站频道与微信公众号同名。长期做企业 IT 基础设施与基础架构，从 Unix 主机、数据库与数据中心运维，一路做到跨国团队的 IT 负责人与上市公司 CIO；近年的重心转向开源自建、家庭网关、DNS 与网络链的自主可控。

与这本书直接相关的一段经历，是在大规模企业环境里长期负责「名字与地址」这一层：

- 负责过跨地域数据中心的 7×24 基础架构运营，涉及高可用集群、容灾、容量规划与事件响应，也管过把数据中心整体迁移到另一地区的项目。那段时间最深的体会是：**故障往往不在某一台服务器，而在解析器、路由、缓存与人的排查顺序交界处**。
- 主导过大型企业办公网络与园区网络的基础架构，包含广域网线路、负载均衡、SSL VPN、企业无线、以及 DDI（DNS / DHCP / IP 地址管理）平台的选型与替换。DDI 这类系统平时没人注意，一旦解析不对，全公司都上不了网。
- 长期维护 Unix / Linux、虚拟化与私有云环境（Solaris / AIX / Linux、VMware、Proxmox / KVM），也做过基于 WireGuard 的企业远程接入、多云备份与边缘网关。
- 写过面向 IT 服务的自动化与观测平台：资产自动发现、CMDB、监控与报表，以及把事件流程、升级路径和 SLA 固化下来的运营体系。

技术写作之外，也做内容运营：B 站与微信公众号 *LeisureLinux* 主要聊 Linux、开源自建、家庭网络与 DNS。

目录

图书介绍 **iii**
0.1 这本书讲什么 . . . iii
0.2 全书结构：五篇分别讲什么 . . . iii
0.3 为什么要讲安全加固 . . . iii
0.4 为什么必须讲隐私防护 . . . iv
0.5 适合哪些读者 . . . iv

版权信息 **v**

作者简介 **vii**

前言 **1**
为什么写这本书 . . . 1
这本书不做什么 . . . 1
一个必须先说清的区分 . . . 1
谁应该读 . . . 2
本书结构 . . . 2
关于示例地址 . . . 2
许可 . . . 2

第一篇　认识 DNS 与 Unbound **3**

第一章　Unbound 的历史演进 **5**
1.1 两个名字：BIND 是什么，Unbound 又是什么 . . . 5
1.2 起点：解析器单一化带来的系统性风险 . . . 5
1.3 BIND 的体量、安全史与「可审计」的含义 . . . 6
1.4 从 Java 原型到 C 实现 . . . 7
1.5 里程碑：每个特性回答了哪个问题 . . . 9
1.6 本章要点 . . . 13
1.7 参考资料 . . . 14

第二章　DNS 基础与主流实现横评 **15**
2.1 命名空间、委派与区域 . . . 15
2.2 递归与迭代：谁在替谁跑腿 . . . 16
2.3 资源记录、缓存与 TTL . . . 17
2.4 CNAME 链与解析过程中的”重来一次” . . . 18
2.5 主流实现横评 . . . 19
2.6 重点对照：BIND 与 dnsmasq . . . 20
2.7 本章要点 . . . 21
2.8 参考资料 . . . 21

第三章　存根、缓存、转发、递归：四种角色 **23**
3.1 四种角色的职责边界 . . . 23
3.2 报文里的四个标志位：RD / RA / AD / CD . . . 25
3.3 一条串联的数据流图（文字描述） . . . 26
3.4 一台机器上跑多个实例的动机 . . . 27
3.5 为第二篇铺路：读配置前先问四个问题 . . . 28

3.6 本章要点 . . . 29
3.7 参考资料 . . . 29

第二篇 按角色配置 Unbound **31**

第四章 缓存解析器：最小可用配置 **33**
4.1 十行起步 . . . 33
4.2 把监听面从回环扩到 LAN . . . 34
4.3 降权、chroot 与信息隐藏 . . . 34
4.4 EDNS 缓冲区：1232，不要 1472 . . . 35
4.5 线程与缓存尺寸 . . . 36
4.6 serve-expired 全家：先刷新，超时才回过期缓存 . . . 36
4.7 完整闭环：checkconf →启动→ dig → stats . . . 37
4.8 从”能跑”到”可交付”的核对表 . . . 38
4.9 本章要点 . . . 38
4.10 参考资料 . . . 39

第五章 转发器：单上游、加密上游、fail-closed **41**
5.1 最小转发器：唯一上游 . . . 41
5.2 多个 forward-addr 是并发竞速，不是主备 . . . 42
5.3 TCP 承载的上游：forward-tcp-upstream . . . 42
5.4 加密上游：forward-tls-upstream 与 tls-cert-bundle . . . 43
5.5 do-not-query-localhost: no 的适用条件 . . . 44
5.6 转发器上不要配 auto-trust-anchor-file . . . 44
5.7 巡检脚本生成配置的幂等写法 . . . 45
5.8 转发器的验收清单 . . . 46
5.9 本章要点 . . . 46
5.10 参考资料 . . . 46

第六章 递归解析器与本地根区镜像 **49**
6.1 auth-zone 的四个开关 . . . 49
6.2 用 NSD 做本地根镜像 . . . 50
6.3 直连 AXFR 的实测形态 . . . 52
6.4 root-hints 与 auth-zone 的区别 . . . 53
6.5 zonefile 持久化：重启别把根区弄丢 . . . 53
6.6 出向 53 必须按 uid 豁免，否则自己吃掉自己 . . . 54
6.7 验收与镜像失效演练 . . . 55
6.8 本章要点 . . . 56
6.9 参考资料 . . . 56

第七章 多实例、缓存与可观测性 **57**
7.1 同一主机多实例的端口 / 接口规划 . . . 57
7.2 按域名分流到不同实例 . . . 59
7.3 过滤表（local-zone 大表）的内存代价与 AppArmor 路径限制 . . . 60
7.4 DDR 压制与 always_nodata . . . 60
7.5 dnstap 全量查询流与 30 天保留期 . . . 61
7.6 unbound-control 开启与不开的取舍 . . . 62
7.7 日志字段与 SERVFAIL 排查 . . . 63
7.8 缓存命中率与延迟的基线测量方法 . . . 63
7.9 本章要点 . . . 64
7.10 参考资料 . . . 65

第三篇 加固 **67**

第八章 威胁模型与 NIST SP 800-81r3 基线 **69**
8.1 为什么先谈可见性 . . . 69
8.2 NIST SP 800-81r3 提供的框架 . . . 70
8.3 可见性矩阵：谁看得见什么 . . . 72

8.4 三类控制的纵深关系 . . . 72
8.5 一条不能含糊的分界线 . . . 73
8.6 控制目标→本书章节 . . . 74
8.7 把基线读成问题，而不是清单 . . . 74
8.8 本章要点 . . . 75
8.9 参考资料 . . . 75

第九章 最小暴露面与访问控制 77
9.1 监听地址逐条列出，不要 `0.0.0.0` . . . 77
9.2 `access-control`：默认拒绝、最长前缀匹配、与顺序无关 . . . 77
9.3 不对公网开放递归：放大倍数从哪来 . . . 78
9.4 降权、chroot 与 AppArmor 路径白名单 . . . 78
9.5 `private-address`：防 DNS rebinding . . . 79
9.6 `local-zone` 与 [RFC 6761] 特殊用途域 . . . 80
9.7 限流与丢弃异常查询 . . . 80
9.8 日志最小化与留存策略 . . . 81
9.9 防火墙只放行必要端口：nftables 基础写法 . . . 82
9.10 完整可复制的加固基线 . . . 82
9.11 逐条验证命令 . . . 83
9.12 出向收敛：把 DNS 隧道数据外泄堵在边界上 . . . 84
9.13 回滚点 . . . 84
9.14 本章要点 . . . 85
9.15 参考资料 . . . 85

第十章 DNSSEC 验证：信任锚、放行与拒绝 87
10.1 信任锚不是一段静态配置 . . . 87
10.2 用 `unbound-anchor` 初始化根锚 . . . 87
10.3 完整验证器配置 . . . 88
10.4 验证器的三态 . . . 89
10.5 验证器行为验收表 . . . 90
10.6 `harden-dnssec-stripped`：有锚时主动关掉就是制造漏洞 . . . 91
10.7 三个名字像、作用却完全不同的 `harden-*` . . . 91
10.8 信任锚失效 / 根 KSK 轮换的观测与恢复 . . . 93
10.9 转发器不要配信任锚 . . . 94
10.10 加固项速查 . . . 95
10.11 KeyTrap 与 NSEC3：验证器自身的拒绝服务 . . . 95
10.12 本章要点 . . . 96
10.13 参考资料 . . . 96

第十一章 DoT：端口 853 的服务端与客户端 97
11.1 服务端的最小骨架：`@853` 才是开关 . . . 97
11.2 证书的两条路线：Let’s Encrypt 与自签 + 客户端 pin . . . 98
11.3 客户端：单上游、TLS 认证名、fail-closed . . . 100
11.4 验证命令：`openssl s_client` 与 `kdig` . . . 101
11.5 与 nginx / HAProxy 共端口的取舍 . . . 102
11.6 连接复用、TCP fastopen 与 MTU / EDNS 1232 . . . 103
11.7 一张验收清单 . . . 104
11.8 本章要点 . . . 104
11.9 参考资料 . . . 105

第十二章 DoH：与 Web 服务共存的加密解析 107
12.1 为什么 Unbound 不是 DoH 前端 . . . 107
12.2 dnsdist：`addDOHLocal` + `newServer(pool)` 的分流 . . . 108
12.3 nginx 反代：`grpc_pass` / `proxy_pass` 与 h2c 的坑 . . . 109
12.4 `/dns-query` 的 `content-type` 与端点语义 . . . 111
12.5 iOS / Android 描述文件：`ServerURL` 与认证限制 . . . 111
12.6 浏览器 canary 域：让内置 DoH 主动关闭 . . . 112

12.7 日志留存方是谁：DoH 并不等于「无人记录」 . . . 113
12.8 验收清单 . . . 114
12.9 本章要点 . . . 114
12.10 参考资料 . . . 114

第十三章 查询隐私：QMIN、防降级与 DDR 压制 117
13.1 QMIN 的动机与边界 . . . 117
13.2 `yes` 与 `yes+strict` 的区别 . . . 118
13.3 真实兼容坑：违反 RFC 8020 的 CNAME 区 . . . 118
13.4 fail-closed：禁止静默明文降级 . . . 119
13.5 DDR 压制与浏览器 canary . . . 121
13.6 ECS 默认不发 . . . 122
13.7 dnstap 是自己的查询流副本 . . . 122
13.8 一张验收清单 . . . 123
13.9 本章要点 . . . 123
13.10 参考资料 . . . 124

第十四章 加密通道选路与 fail-closed 工程 125
14.1 整体链路：从客户端:53 到远端递归验证器 . . . 125
14.2 nft 规则：劫持:53 与 uid 豁免 . . . 126
14.3 `nft -f` 重建表会清空集合 . . . 128
14.4 跨境链路实测：为什么加密通道优先 TCP . . . 128
14.5 真实形态的坑：WG 回包被 `output-tproxy` 吃掉 . . . 130
14.6 一张验收清单 . . . 131
14.7 本章要点 . . . 131
14.8 参考资料 . . . 132

第四篇 RFC 与协议理论 133

第十五章 DNS 基础 RFC 族 135
15.1 命名空间与报文：[RFC 1034] / [RFC 1035] . . . 135
15.2 操作与规范的澄清：[RFC 1912] / [RFC 2181] . . . 136
15.3 负缓存：[RFC 2308] . . . 137
15.4 EDNS0：[RFC 2671] → [RFC 6891] . . . 137
15.5 未知 RR 类型：[RFC 3597] . . . 138
15.6 大小写：[RFC 4343] . . . 139
15.7 特殊用途域名：[RFC 6761] . . . 139
15.8 NXDOMAIN cut：[RFC 8020] . . . 140
15.9 术语统一：[RFC 8499] . . . 141
15.10 过期缓存应答：[RFC 8767] . . . 141
15.11 文档地址段：[RFC 5737] . . . 142
15.12 条文之间的因果链 . . . 142
15.13 本章要点 . . . 143
15.14 参考资料 . . . 143

第十六章 DNSSEC RFC 族 145
16.1 DNSSECbis：[RFC 4033] / [RFC 4034] / [RFC 4035] . . . 145
16.2 验证器为什么必须 fail-closed . . . 146
16.3 被剥离签名为什么等于降级 . . . 147
16.4 信任锚滚动：[RFC 5011] . . . 148
16.5 澄清与实现说明：[RFC 6840] . . . 149
16.6 算法信号与算法要求：[RFC 6975] / [RFC 8624] . . . 149
16.7 签名区的否认存在：[RFC 7129] / [RFC 9276] . . . 150
16.8 运维实践：[RFC 6781] / [RFC 7583] . . . 151
16.9 DLV 的历史与退役：[RFC 4431] / [RFC 5074] . . . 151
16.10 根 KSK 轮换（2018） . . . 152
16.11 条文之间的因果链 . . . 152
16.12 一个设计哲学的代价：算法复杂度攻击 . . . 153

16.13 本章要点 . . . 153
16.14 参考资料 . . . 153

第十七章 加密与隐私 RFC 族 155
17.1 查询名最小化的两版规范：[RFC 7816] → [RFC 9156] . . . 155
17.2 DoT：[RFC 7858] . . . 156
17.3 DoH：[RFC 8484] . . . 157
17.4 DoQ：[RFC 9250] . . . 157
17.5 使用配置：[RFC 8310] . . . 158
17.6 两跳分离：[RFC 9230] . . . 159
17.7 发现与压制：[RFC 9462] . . . 160
17.8 运营者视角：[RFC 8932] . . . 160
17.9 客户端子网：[RFC 7871] . . . 161
17.10 整体隐私考量：[RFC 9076] . . . 162
17.11 两个收尾判断 . . . 162
17.12 本章要点 . . . 163
17.13 参考资料 . . . 163

第十八章 IPv6 / RA / RDNSS RFC 族与实现差异 165
18.1 底盘：[RFC 4291] 地址架构与 [RFC 8200] 1280 字节最小 MTU . . . 165
18.2 [RFC 4861] NDP 与 RA：报文为什么必须是 hop limit 255 . . . 166
18.3 [RFC 4862] SLAAC：为什么 PIO lifetime=0 缩不掉一个长寿命地址 . . . 167
18.4 [RFC 6106] → [RFC 8106]：RDNSS / DNSSL 与 lifetime=0 . . . 168
18.5 [RFC 4191]：RIO 与默认路由器优先级 . . . 169
18.6 [RFC 4941] 临时地址与它的后续修订 . . . 169
18.7 [RFC 6724] 源地址选择：规则编号的迁移与两条关键偏好 . . . 170
18.8 [RFC 4193] ULA：稳定的「内部编号」，不是可路由公网地址 . . . 171
18.9 PIO 的后续修订：编号核对与 [RFC 8978] / [RFC 9096] . . . 171
18.10 横向对比：同一组 RFC 在六种实现里的行为差异 . . . 172
18.11 [RFC 6724] 的另一面：`getaddrinfo()` 的返回顺序 . . . 174
18.12 本章要点 . . . 175
18.13 参考资料 . . . 175

第五篇 工程实践 177

第十九章 IPv6 部署：RA、RDNSS、ULA 与前缀轮换 179
19.1 「全屋无污染 DNS」的真正机制 . . . 179
19.2 一份合格 RA 里的六项内容 . . . 179
19.3 ULA 做 DNS 地址，GUA 做上网地址 . . . 181
19.4 前缀轮换 × 临时地址寿命 . . . 182
19.5 光猫只当管道用：一次数据面缺陷的教训 . . . 183
19.6 可复制的 sysctl 与 RA 引导脚本骨架 . . . 183
19.7 验证命令与验收清单 . . . 184
19.8 回滚点汇总 . . . 185
19.9 本章要点 . . . 185
19.10 参考资料 . . . 185

第二十章 Kubernetes 中的 DNS 选型 187
20.1 先划清一条边界：服务发现≠递归解析 . . . 187
20.2 从 kube-dns 到 CoreDNS . . . 187
20.3 四种 dnsPolicy 与 resolv.conf 的形态 . . . 188
20.4 ndots:5 的查询放大，以及 search 域与绝对域名 . . . 189
20.5 NodeLocal DNSCache：大规模集群几乎必需 . . . 190
20.6 一份可复制的 Corefile . . . 190
20.7 三种拓扑与取舍 . . . 193
20.8 验收清单 . . . 194
20.9 本章要点 . . . 195
20.10 参考资料 . . . 195

第二十一章 综合实验：一台网关 + 一台云主机 197
21.1 拓扑总览：把前四篇拼成一条链 . . . 197
21.2 客户端 DNS 指向：引导层只做一件事 . . . 198
21.3 分流规则：策略 DNS 只留一个出口 . . . 199
21.4 各实例监听表 . . . 199
21.5 失败注入与观测点 . . . 200
21.6 验收命令清单 . . . 201
21.7 回滚点汇总与实验纪律 . . . 202
21.8 本章要点 . . . 203
21.9 参考资料 . . . 203

第二十二章 故障案例集 205
22.1 案例索引与排查纪律 . . . 205
22.2 案例①单上游变竞速 . . . 205
22.3 案例②静默明文降级 . . . 206
22.4 案例③ strict QMIN 撞违反 RFC 8020 的 CDN 区 . . . 207
22.5 案例④转发器上配 trust anchor 导致 priming 失败 . . . 207
22.6 案例⑤负缓存放大 . . . 208
22.7 案例⑥本机解析器出向 53 被自己 hijack . . . 209
22.8 案例⑦ WG 回包被 output-tproxy 打标 . . . 210
22.9 案例⑧临时地址寿命 × 前缀轮换 . . . 210
22.10 案例⑨ /48 与 /64 粒度 . . . 211
22.11 案例⑩跨境 UDP QoS . . . 211
22.12 跨案例的三条通用纪律 . . . 212
22.13 本章要点 . . . 212
22.14 参考资料 . . . 212

第二十三章 WPAD 与 PAC：自动代理发现的实践 215
23.1 两层：找 URL 与决定怎么走 . . . 215
23.2 标准状态：一份过期的工作组草案 . . . 216
23.3 草案 §4.4：五种发现机制与 MUST / SHOULD . . . 216
23.4 DNS devolution：会走出组织边界的那一跳 . . . 217
23.5 现代化尝试：draft-joshco-wpadng-02 . . . 217
23.6 PAC 层：`FindProxyForURL()` 与四个常用函数 . . . 218
23.7 落地：把 wpad.dat 放到一个可控的 URL 上 . . . 219
23.8 校验、开关与排障 . . . 221
23.9 为什么不推荐它 . . . 222
23.10 本章要点 . . . 222
23.11 参考资料 . . . 223

第二十四章 Adhole：DNS 层过滤表的工程实践 225
24.1 能力边界：DNS 层只能表达”名字” . . . 225
24.2 六类上游格式逐条拆 . . . 225
24.3 可复制的转换骨架 . . . 227
24.4 输出三份表与 domain-state.tsv . . . 229
24.5 分类靠解析 A 记录，不靠域名后缀猜 . . . 229
24.6 `local-zone` 与 `forward-zone` 的优先级（本章核心） . . . 230
24.7 原子生成：临时文件 + 校验 + `install` . . . 232
24.8 大表的内存代价、加载时间与 AppArmor . . . 233
24.9 定时同步到另一台解析器 . . . 233
24.10 误伤排查、白名单回滚点与保留期 . . . 234
24.11 与第三篇的三处呼应 . . . 234
24.12 本章要点 . . . 235
24.13 参考资料 . . . 235

第二十五章 内网 DDNS 与微软 DNS 对照 237
25.1 Unbound 不接受动态更新 . . . 237

25.2 微软 DNS：真正实现了 [RFC 2136] 的那套模型 . . . 238
25.3 三条绕行路径 . . . 240
25.4 本章要点 . . . 243
25.5 参考资料 . . . 243

第二十六章 企业级大规模部署：anycast 与集群 245
26.1 前提与边界 . . . 245
26.2 官方手册里的 anycast 事实 . . . 245
26.3 NIST SP 800-81r3 的部署指导 . . . 247
26.4 工程骨架 . . . 248
26.5 验收清单 . . . 250
26.6 失败模式与回滚点 . . . 251
26.7 本章要点 . . . 251
26.8 参考资料 . . . 251

第二十七章 单播 DNS 与多播 DNS：从 mDNS 到 DNS-SD 253
27.1 两条互不相交的解析通道 . . . 253
27.2 mDNS：只在本链路里成立的名字 . . . 254
27.3 DNS-SD：三件套，以及它并不依赖 mDNS . . . 255
27.4 四维对照表 . . . 256
27.5 一段可复制的教学配置：把分流钉死 . . . 257
27.6 边界与排障：企业里最常见的三种「时通时不通」 . . . 259
27.7 本章要点 . . . 260
27.8 参考资料 . . . 261

第二十八章 DNS 故障域隔离：从一次全球性中断说起 263
28.1 先把归因摆正：不是「同域名」 . . . 263
28.2 证据分级表：四类，不许混 . . . 264
28.3 六层隔离框架 . . . 265
28.4 映射到前 27 章 . . . 267
28.5 故障注入与验收清单 . . . 268
28.6 本章要点 . . . 270
28.7 参考资料 . . . 270

第二十九章 DNS 作为攻击面：隧道、扫描与资产暴露 271
29.1 DNS 隧道：把数据编码进查询名 . . . 271
29.2 高速扫描与开放解析器：不提供可被滥用的能力 . . . 274
29.3 互联网暴露面排查：只查自己有授权的名字 . . . 275
29.4 悬空 CNAME 与子域接管 . . . 276
29.5 历史漏洞与 Unbound 的加固模式 . . . 277
29.6 攻击→观测信号→加固动作→本书章节 . . . 278
29.7 与后续章节的分工 . . . 279
29.8 本章要点 . . . 279
29.9 参考资料 . . . 279

第三十章 DNS 里的信任数据：邮件、SSH 与自动化 281
30.1 一张大表：谁在读这些记录 . . . 281
30.2 完整配置片段 . . . 284
30.3 共同结论：四条物理性质 . . . 287
30.4 自动化写记录的凭据与故障域 . . . 288
30.5 本章要点 . . . 288
30.6 参考资料 . . . 288

第三十一章 可观测性栈：dnstap、Grafana、eBPF 与告警 291
31.1 先划边界：采集点已有，缺的是栈 . . . 291
31.2 第一层：应用内统计与 SERVFAIL 日志 . . . 292
31.3 第二层：dnstap 查询流的集成与隐私边界 . . . 293
31.4 第三层：内核与网络，不改 Unbound . . . 295

31.5 集成骨架 . . . 296
31.6 告警规则示例与误报治理 . . . 297
31.7 演练：三条链路都要看见 . . . 299
31.8 与其他章的交叉引用 . . . 300
31.9 本章要点 . . . 300
31.10 参考资料 . . . 300

第三十二章 代理与过渡场景里的 DNS 303
32.1 Fake-IP：把域名解析推迟到连接建立时 . . . 303
32.2 DNS64 / NAT64 / 464XLAT：纯 IPv6 网络访问 IPv4 目标 . . . 305
32.3 域名还原：当 DNS 已经不给出名字时 . . . 307
32.4 Fake-IP 与 DNS64 的本质区别 . . . 308
32.5 SSH / SOCKS 通道里，DNS 到底归谁解析 . . . 309
32.6 本章要点 . . . 309
32.7 参考资料 . . . 309

第三十三章 DHCP 与 DNS 的交接面 311
33.1 一条配置要穿过四层 . . . 311
33.2 一张表：option 编号→ DNS 行为→症状→验证 . . . 311
33.3 RDNSS 与 DHCPv6 的优先级战争 . . . 314
33.4 租约时序陷阱 . . . 314
33.5 option 15 / 119 的污染 . . . 315
33.6 option 81 与动态更新注入的攻防 . . . 316
33.7 rogue DHCP 与 DHCPv6 假 DNS . . . 316
33.8 教学配置片段（全部文档地址段） . . . 317
33.9 本章要点 . . . 318
33.10 参考资料 . . . 319

第三十四章 红队视角的 DNS 侦察 321
34.1 授权纪律与本章边界 . . . 321
34.2 侦察链总览 . . . 322
34.3 被动收集：不留查询到目标 . . . 322
34.4 名字枚举：从静默到高噪声 . . . 323
34.5 区结构探测：委派、传送与签名链 . . . 323
34.6 解析器探测：把解析器本身当资产 . . . 324
34.7 信任记录枚举：公开可读的供应链地图 . . . 325
34.8 内网侧授权场景：广播与交接面 . . . 325
34.9 从信号到告警：喂给第 31 章与第 29 章 . . . 326
34.10 授权测试纪律的完整闭环 . . . 327
34.11 阶段 / 工具形态 / 观测信号 / 防守动作 / 本书章节 . . . 327
34.12 本章要点 . . . 328
34.13 参考资料 . . . 328

第三十五章 DNS 蜜罐与被动观测 329
35.1 三种形态：把 DNS 变成观测面 . . . 329
35.2 蜜罐究竟能观测到什么 . . . 330
35.3 不能做什么 . . . 331
35.4 与本书其他章的闭环 . . . 331
35.5 工程骨架（教学用） . . . 333
35.6 指标与判读 . . . 335
35.7 演练：三条链路都要有信号 . . . 336
35.8 伦理与授权边界 . . . 337
35.9 与其他章的交叉引用 . . . 337
35.10 本章要点 . . . 337
35.11 参考资料 . . . 338

第三十六章 DNS 故障排查工具箱 339
36.1 第一层：谁在解析 . . . 339

36.2 第二层：解析器怎么答 . . . 340
36.3 第三层：抓到什么包 . . . 341
36.4 第四层：内核动过没 . . . 343
36.5 第五层：应用怎么连 . . . 343
36.6 第六层：归属与边界 . . . 344
36.7 哪条命令会骗你 . . . 345
36.8 工具实装表 . . . 345
36.9 症状→第一条命令→判读→下一层→本书章节 . . . 346
36.10 本章要点 . . . 347
36.11 参考资料 . . . 348

第三十七章 HTTPDNS：为什么引入、如何用 Unbound 自建、大厂后端如何架构 349
37.1 先把它和标准 DNS 分开 . . . 349
37.2 为什么要引入 HTTPDNS . . . 350
37.3 用 Unbound 自建：三层缺一不可 . . . 351
37.4 大厂后端是怎么架构的 . . . 355
37.5 它不可能覆盖 App 的全部解析 . . . 356
37.6 本章要点 . . . 357
37.7 参考资料 . . . 357

附录 A RFC 清单 359

附录 B 缩略语 371

版本信息 375
本次版本修订记录 (v0.7.2) . . . 375
历史版本 . . . 375

前言

为什么写这本书

DNS 是 1983 年定下骨架的协议。四十多年后，几乎所有联网动作仍要先问它一句”这个名字在哪里”。但绝大多数人对它的了解停留在”填一个公共递归解析器的地址”。

这本书不打算教你怎么填 DNS 地址。它要回答的是另一组问题：

- 一台机器上的解析器，凭什么相信拿到的答案？
- 从客户端到权威服务器，这条链上有谁看得见你在查什么？
- 把 DNS 加密之后，隐私就真的到手了吗？
- 为什么明明配置”看起来都对”，网站却打不开？

我们用过几乎所有形态：递归、转发、缓存、验证、加密前端、策略网关、IPv6 前缀轮换、无线路由通告、容器集群里的服务发现。也因此在每一层都踩过坑。有些坑的代价是一整天：为了在光猫里配一个 IPv6 DNS，把工作模式一改，ICMP 通、TCP 握手通、数据阶段静默丢包，此后所有”网站打不开”都被误判成别的问题。

这些坑有一个共同点：它们都不在 DNS 教材里，而在解析器、内核、nftables 和路由通告的交界处。所以这本书的结构不是按功能罗列参数，而是按”角色→配置→加固→协议→工程”层层推进，并且每一处配置都配一句”怎么验证”和一句”坏了怎么退回来”。

这本书不做什么

- **不是官方文档的中文版**。NLnet Labs 的 Unbound 官方文档写得很好，且以 CC BY 3.0 开放。本书引用它、链接它，但不复述它。
- **不是任何英文书的翻译**。市面上专门写 Unbound 的书极少；Jan-Piet Mens 的 *Alternative DNS Servers*（UIT Cambridge, 2008/2009）是第一本记录 Unbound 的书，作者 2010 年放出了无 DRM 的 PDF。本书把它当作史料与横评参考，不节录、不翻译其正文。
- **不是产品宣传**。BIND、dnsmasq、CoreDNS、NSD、PowerDNS、Knot Resolver 各有各的主场。书里会明确写 Unbound 不适合做什么。
- **不是合规清单的复读**。第三篇会用 NIST SP 800-81r3 的框架建立威胁模型，但落点始终是可运行的配置与可观测的验证。

一个必须先说清的区分

全书反复用到这条分界，先在这里定死：

	解决什么	不解决什么
DNSSEC	数据完整性与来源认证：答案有没有被改过、是不是真的来自该区	不加密传输，不隐藏查询内容
DoT / DoH / DoQ	传输加密与链路隐私：旁人看不到查询内容	不验证答案真伪，不保证上游可信

两者叠加才构成”既不被偷看、也不被篡改”，但**任何一层都不能替代另一层**。混淆这两者，是很多”我已经上了加密 DNS 所以安全了”的根源。

同样要区分的是**谁在做验证**：存根解析器（操作系统里的那一小段）通常不验证，它把 RD 位置 1 丢给上游，然后看上游回的 AD 位。真正做验证的是递归解析器。把验证器放在链路的哪一层，决定了整条链的安全边界在哪里。

谁应该读

- **家庭网关 / 自托管玩家**：想让全屋设备的 DNS 干净、可过滤、可观测，且不想把查询交给第三方。
- **运维 / SRE**：要在一台机器上跑多个解析器实例，按域名或信任域分流，并处理上游抖动。
- **安全工程师**：要能画出 DNS 的信任边界，说清”谁看得见什么”，并对齐 NIST 的部署指南。
- **Kubernetes 运维**：要理解 CoreDNS 与 Unbound 的分工，处理 `ndots`、NodeLocal DNSCache 与集群出口验证。
- **网络爱好者**：想从 RFC 层面理解为什么某个参数是这样，而不是”教程里都这么写”。

需要的基础：会看 `dig` 输出、能改 systemd 单元、知道 IPv4/IPv6 子网是怎么回事。不需要读过 RFC。

本书结构

第一篇认识 DNS 与 Unbound（第 1–3 章）从 Unbound 的诞生讲起，把 DNS 基础与主流实现横评一遍，严格区分存根、缓存、转发、递归四种角色。第一篇不贴长配置。

第二篇按角色配置 Unbound（第 4–7 章）按角色分别给出缓存解析器、转发器、递归解析器 + 本地根镜像、多实例与可观测性的完整配置。每个配置块后面紧跟”为什么”、“怎么验证”、“失败模式与回滚点”。

第三篇加固（第 8–14 章）用 NIST SP 800-81r3 建立威胁模型，然后逐层展开：最小暴露面、DNSSEC 验证、DoT、DoH、查询隐私（QMIN / 防降级 / DDR 压制）、加密通道选路与 fail-closed 工程。这是全书最重的一篇。

第四篇 RFC 与协议理论（第 15–18 章）这一篇**以理论为主**，不重复贴配置：DNS 基础 RFC 族、DNSSEC RFC 族、加密与隐私 RFC 族、IPv6 / RA / RDNSS RFC 族，以及同一组 RFC 在不同实现里的处理差异。

第五篇工程实践（第 19–37 章）IPv6 部署（RA、RDNSS、ULA、前缀轮换与临时地址寿命）、Kubernetes 中的 DNS 选型、一套完整的教学拓扑、以及十个真实故障案例的复盘。实战与理论在这里收束。

关于示例地址

本书所有 IP、域名、主机名、token 都是**教学用示例**，IPv4 取自 RFC 5737 文档段（`192.0.2.0/24`、`198.51.100.0/24`、`203.0.113.0/24`），IPv6 取自 RFC 3849 的 `2001:db8::/32`，域名用 `example.net` / `example.org`。任何与真实环境的相似都属巧合。

书中的实测数据（延迟、吞吐、计数器、失败率）来自一台家庭网关加一台境外云主机的实验环境，写进书里是为了说明**机制**与**量级**，不是性能承诺。你的链路、运营商与内核版本不同，数字必然不同。请以书里的验证命令在你自己的环境里重新测一遍。

许可

正文以 **CC BY-SA 4.0** 发布，构建脚本沿用 MIT 模板；完整许可与第三方材料声明见「版权信息」页。

构建产物（PDF / ePub / 独立 HTML）由 GitHub Actions 从 Markdown 自动构建；`scripts/unbound-audit.sh` 可检查一份 `unbound.conf` 是否违反第三篇的加固基线。

第一篇　认识 DNS 与 Unbound



第一章

Unbound 的历史演进

Unbound 不是某家厂商为了补齐产品线而做的 DNS 组件，它是带着一个明确问题出生的：当互联网上绝大多数递归解析器都跑着同一套代码时，一个解析器自身的漏洞就不再是”某台机器的故障”，而是全局性的单点风险。理解这条设计动机，比记住版本号更重要——它解释了为什么一个只做递归验证的实现会刻意把边界收得那么窄。

1.1 两个名字：BIND 是什么，Unbound 又是什么

先把名字说清楚，后面十几章才不会含混。

BIND = Berkeley Internet Name Domain。1980 年代初诞生于加州大学伯克利分校的计算机系统研究组（CSRG），由 DARPA 资助，Douglas Terry、Mark Painter、David Riggle、Songnian Zhou 四名研究生完成，随 4.3BSD 发行。它的主进程叫 `named`（name daemon），`dig`、`nsupdate` 与 resolver 库都出自同一套软件包。ISC 的《The History of BIND》与 BIND 9 官方文档都这么写。

Unbound 的名字与 BIND 没有词源关系。2008 年 5 月 20 日的官方发布新闻稿给出的定位是「a secure, high-performance **alternative to** BIND」——是**作为 BIND 的替代方案**，不是从 BIND 这个词拆出来的。官方历史页只说它是一个「开发独立开源 DNS 软件与文档」的项目，**没有公布过命名故事**。因此「unbound 取『不受束缚』之意、不被权威与递归双重职责绑住」是常见解读，这里按**推断**标注，不作为官方词源引用。

后来 Unbound 在 FreeBSD、OpenBSD 里取代 BIND 成为 base system 默认解析器，那是**结果**，也不是命名原因。把这两件事分开，才看得清第 1 章后面那条主线：Unbound 的取舍不是「比 BIND 多几个功能」，而是**主动把职责收窄**。

1.2 起点：解析器单一化带来的系统性风险

2000 年代中期，需要自己搭一台递归解析器的运维人员，可选项实际上不多。BIND 9 几乎等同于”DNS 服务器”这个词的默认实现：它既能做权威服务器，也能做递归解析器，还能做转发、区域传送、动态更新、视图（view）、访问控制列表，以及后来加入的 DNSSEC 全套能力。功能齐全是优点，但代价是代码体量、配置语法的规模，以及一个不那么直观的事实：

- 一台只想要”帮我查域名并验证签名”的机器，也必须承载一整套权威服务器的代码路径；
- 配置面越大，运维误配的概率越高，审计时需要读完的代码也越多；
- 更关键的是，如果绝大多数解析器共享同一个实现，那么该实现的一个远程漏洞会同时影响整个互联网，这就是所谓的**解析器单一化（resolver monoculture）**风险。

Unbound 项目自身的早期需求文档把这一点写成了明确的工程目标，原文列出的目标里包括”code diversity in the DNS resolver monoculture”（在递归解析器的单一化格局中提供代码多样性）、“drop-in replacement for BIND apart from config”（除配置语法外可作为 BIND 的替代）、以及”smallest as possible component that does the job”（尽可能小的、刚好够用的组件）。同一份文档的 non-goals 只有两条，值得原样记住：**不做权威服务器、不要过多特性**。

这两条约束决定了后面十几年的所有取舍。Unbound 的”小”不是功能少，而是**边界清晰**：递归、缓存、DNSSEC 验证是它的职责，权威应答、DHCP、服务发现、Web 界面不是。第二篇第 4–7 章会看到，正是这个清晰的边界让”按角色拆成多个实例”成为一种可维护的部署方式，而不是被迫的权宜之计。

1.2.1　为什么「可审计」和「安全」是同一件事

需求文档里把设计拆成三个可替换的模块：验证器（validator）、迭代器（iterator）、缓存（cache），网络与查询管理代码只负责把它们串起来。这个划分不是软件工程洁癖，它有直接的安全含义：

- **迭代器**只负责向层级中的权威服务器发问，它不信任应答里的附加内容；
- **验证器**只负责检查 DNSSEC 签名链，它不参与发包；
- **缓存**只存放被前两者认可的数据。

于是”一个伪造的应答能不能进缓存”这个问题，可以被收敛到一条明确的路径上审查，而不是散落在几万行代码的各个分支里。Unbound 后来把清理应答、丢弃不该进缓存的附加数据的组件叫 **scrubber**（清理器）。2008 年缓存投毒事件爆发时，官方历史文档给出的结论正是：由于设计时已经采用了最大程度的查询端口与查询 ID 随机化、并且清理器不会把不信任的数据写入缓存，Unbound 不需要为这次漏洞紧急打补丁来补齐抗 forged-reply 能力。

这不是说 Unbound 天然免疫一切攻击。同一份文档明确写着：对 Kaminsky 所描述的那类缓存投毒，**只有 DNSSEC 是完整的解决方案**，端口随机化、0x20 大小写编码等手段只是提高伪造难度。把这个边界讲清楚很重要：DNSSEC 提供的是**数据完整性与来源认证**，而后面章节提到的 DoT、DoH、DoQ 提供的是**传输加密与隐私**，两者不互相替代。历史章节里最容易犯的错，就是把「抗投毒」和「加密传输」当成同一件事来叙述。

1.3　BIND 的体量、安全史与「可审计」的含义

谈「为什么需要一个只做递归验证、小而可审计的实现」，绕不开 BIND。BIND 是 DNS 协议最早的参考实现，从 BIND 4、BIND 8 到后来重写的 BIND 9（重写年份见 ISC 官方资料），长期承担双重角色：既是互联网上多数权威服务器（authoritative server）的运行环境，也是多数递归解析器的运行环境。ISC 的公开公告与 CVE 记录里，BIND 9 多年间出现过远程拒绝服务、缓存投毒、断言失败、动态更新与区域传送路径上的问题；2008 年的补丁公告由多家厂商同时发布，说明这不是某一家的配置失误，而是**同一类递归逻辑在多个实现里同时存在的缺陷**。

这里要避免一个不严谨的推论：「CVE 多 = 质量差」。更准确的说法是三条：

- 部署面越广、代码路径越多，被研究者审视得越充分，公开漏洞数量自然更多；
- 权威 + 递归 + 动态更新 + 视图（view）+ 区域传送都在一个进程里，任何一条路径的缺陷都可能影响另一条路径上的数据；
- 对运维而言，**能被完整读完并审计的代码量**，直接决定了「这次升级到底改了什么」能不能自己判断，而不是只能相信发布公告。

第三条才是 Unbound 需求文档里 `code diversity` 与 `smallest as possible component` 的真实指向。代码多样性不是为了证明谁更安全，而是为了让一次实现层面的漏洞不至于同时打穿所有解析器；小而专注的实现，则是让运维在出事时能沿着一条明确路径把因果链走完。2008 年那次事件里，Unbound 因清理器不把不信任的附加数据写入缓存、且端口熵从设计之日就取满，得以在不紧急改架构的前提下应对——这是**边界清晰带来的红利**，不是「代码少所以没 bug」。

BIND 至今仍是权威服务器与复杂企业场景里成熟、可靠、文档完备的选择。这里要判断的是**当需求被限定为「只做递归验证」时，全能实现的额外代码路径是否还值得承担**。

1.3.1　2008 年的外部冲击：一次全行业压力测试

2008 年，Dan Kaminsky 披露了一类影响广泛的 DNS 缓存投毒漏洞（CVE-2008-1447；CERT/CC 编号为 VU#800113，见本章参考资料 [1][2]）。攻击者不需要猜中某个特定域名的应答，而是可以通过反复触发对同一父域下随机子名的查询，制造大量”在途查询”，从而在较短时间里撞中 16 位查询 ID 与源端口的组合，把伪造的权威应答塞进缓存。伪造数据一旦带上很长的 TTL，会把后续所有访问导向攻击者指定的地址。

1.3.2　投毒是怎么实现的（CVE-2008-1447 图解）

下图把一次成功的伪造应答拆成六个步骤。[1][2] 描述的要点是：出向查询的 **16 位 TXID** 与**源端口**共同构成匹配空间；当端口固定或可预测时，攻击者只需撞 65536 种 ID；而攻击者自己能制造大量「在途查询」，等于同时买了很多张彩票。

```
① 攻击者想污染  example.org  的 NS / A 记录
② 攻击者向目标递归器反复查询随机子名：
     < 随机 1>.example.org  A?   → 递归器发往 example.org 权威（TXID=t1, sport=p）
     < 随机 2>.example.org  A?   → 递归器发往 example.org 权威（TXID=t2, sport=p）
     ...                          （同时有 N 个查询「在途」）
③ 递归器等待权威应答；此时攻击者以 example.org 权威的身份
   批量伪造应答，每个伪造包猜一个 TXID：
```

```
    [伪造应答]  QNAME=< 随机 k>.example.org  TXID=guess
                ADDITIONAL:  example.org.  NS  ns.attacker.example.
                             ns.attacker.example.  A  203.0.113.53
④ 只要有一次（guess，sport）命中某个在途查询，且比真应答先到，
   递归器就把它当作合法应答接受；其余 N-1 个随机子名只是「制造彩票」。
⑤ 关键在 ADDITIONAL 段：夹带的 NS + glue 把整个 example.org 的
   后续解析引到攻击者控制的地址（bailiwick 检查不严时尤其有效）。
⑥ 伪造记录带长 TTL → 缓存里留很久；之后所有访问 example.org 的客户端
   都被导向 203.0.113.53，且链路上每一层缓存都要清理。

缓解（不是根除）：源端口随机化把空间从 2^16 扩到约 2^32，
0x20 大小写随机化再加熵；根除要靠 DNSSEC 签名验证 [3]。
```

①为什么这么设：这张图只画机理，不提供复现命令。三条对应关系要记住——「随机子名」不是目标而是**制造在途查询**的手段，「夹带 glue」才是把单次命中放大成整域劫持的原因，「长 TTL」决定故障持续时间。

②怎么验证：`dig +short example.org A` 看应答里有没有意料之外的 NS / A；用 dnstap 或抓包观察出向查询的源端口是否随机、同一名字是否反复出现高熵随机子标签。

③失败模式 / 回滚点：若已污染，清缓存不会自动修复——权威侧改回正确记录后，链路上每个缓存实例都要刷，且要检查有没有别的区域也被同一个 glue 带走。回滚点是换成带 DNSSEC 验证的递归器，并确认 `harden-glue` 一类边界开启。

把攻击面拆开看，会更清楚为什么「代码体量」是个安全指标：

- 查询 ID 只有 16 位，若出向源端口固定或可预测，攻击者需要撞中的空间只剩下 65536；
- 攻击者可以自己发起对目标域下大量随机子名的查询，制造许多在途请求，从而同时尝试多组伪造应答；
- 一旦有一次命中，伪造应答里可以夹带指向目标域的 NS 与胶水记录（glue record），把整个域的后续解析都引到攻击者控制的地址；
- 伪造记录通常带很长的 TTL，清掉它不是重启进程就一定能解决，链路上每一层缓存都要处理。

最后两条在本书后面的故障案例里会再次出现：一是**域外夹带的 glue 是经典缓存投毒面**，二是**负缓存（negative caching）与长 TTL 会放大一次瞬时故障**。它们不是配置细节，而是 DNS 数据模型本身的性质。

对本书读者而言，这件事的意义不在于复现攻击，而在于它把两条结论钉死了：

1. 递归解析器的**熵来源**（查询 ID、源端口、0x20 大小写）是安全边界的一部分，不是性能细节；
2. 只要签名验证链没有铺开，缓解措施就只是提高成本，不是消除风险。

Unbound 的 1.0 恰好在这段时间内发布。它在 1.0 中实现了基于 ARC4 随机例程的查询 ID 与端口随机化，并从”未分配端口池”中随机选取出向端口；1.0.2 的补丁公告里又追加了一批此前为避免被逆向推导而暂缓公开的对抗措施。也就是说，Unbound 的”默认安全”不是后来营销出来的定位，而是它在第一次公开亮相时就带着的设计前提。

1.4 从 Java 原型到 C 实现

1.4.1 2004：架构与资助

按通行记载，Unbound 的设计工作始于 2004 年，由 **Jakob Schlyter（Kirei）**与 **Roy Arends（Nominet）**提出架构，项目由 **VeriSign** 与 **EP.NET** 资助，早期的 Java 原型由 **David Blacka、Matt Larson** 等人实现。原型采用 BSD 许可，目标是做一台”full-service, iterative-mode, validating DNS resolver”——完整的、迭代模式的、带验证的解析器。

Nominet 与 VeriSign 的参与不是偶然：前者运营国家级顶级域，后者运营根与顶级域基础设施，双方都对”递归侧不能被单一实现垄断”有直接利益。关于早期参与者名单，不同史料的表述略有出入——官方需求文档把项目的起始归为 Bill Manning、David Blacka、Matt Larson，并记载 Java 原型获得了 Geoff Sisson 与 Roy Arends（Nominet）的贡献；Java 原型页面则写明项目由 VeriSign 与南加州大学信息科学研究所（USC/ISI）共同资助。**年份与参与者在不同来源间不完全一致时，请见官方手册的历史章节与 NLnet Labs 发布说明**，本书不为具体人名与年份做超出来源的断言。

1.4.2 原型到底验证了什么

Java 原型不是”先写个能跑的版本再说”。原型页面明确说明：Java 被选中的理由是 **dnsjava** 库质量好、且主要开发者熟悉它，而最终实现语言从一开始就定为 C。原型的目的是**验证一种设计**：把解析器拆成一条模块链，既可以组装成完整的递归解析器，也可以只取出其中几段，组装成一个”带 DNSSEC 验证的存根解析器”。缓存等少数关键组件还允许替换实现。

这段历史直接解释了 Unbound 后来两个长期特征：

- `libunbound` 能把解析能力嵌进应用程序，而不是必须起一个守护进程——这是”模块链可裁剪”的遗产；
- 配置文件里 `module-config` 这一行的存在感很强——模块次序（例如验证器、迭代器、缓存类模块的先后）是设计层面的概念，不是后来加的补丁。

原型本身是事件驱动、异步的服务器：每个收到的报文被转成一个事件，定时器与内部生成的查询也是事件。这个模型在 C 版本里被保留下来，也是 Unbound 能用少量线程承载大量在途查询的原因。

1.4.3 2006 年底：NLnet Labs 加入并用 C 重写

大约 2006 年，项目产生了”做一个可部署的、完整的 C 实现”的想法，**NLnet Labs** 承担了这次实现。官方需求文档的原文是”Around 2006 the idea came to create a full-fledged C implementation ready for deployed use. NLnet Labs volunteered to write this implementation.”

NLnet Labs 当时已有 **NSD**（一个只做权威服务器的实现）的开发经验，把”只做一件事、把它做干净”的工程习惯带到了 Unbound：C 实现延续了 BSD 许可、模块划分与”不做权威服务器”的边界，同时也复用了不少底层库与构建经验。**Wouter Wijngaards** 是这一阶段及之后长期的主要开发者。

值得注意的是方向上的对称：NSD 把”权威”这一半从 BIND 式全能服务器里切出来，Unbound 把”递归 + 验证”这一半切出来。两者都不试图覆盖对方。**Unbound 从来不是”半个 BIND”，而是一个边界先定好、再填功能的实现**。

1.4.4 2008-05-20：1.0 公开发布

2008 年 5 月 20 日，Unbound 1.0 随一份联合新闻稿公开发布，参与方为 NLnet Labs、VeriSign、Nominet 与 Kirei，采用 **BSD 许可**。官方历史页的表述是”It was released in May 2008 with this press release”，并说明 C 实现基于一份 Java 原型。具体的发布日期与联合发布方名单，以官方手册与 NLnet Labs 发布说明为准；不同二手资料在”5 月”内部的具体日子上偶有差异，本书采用 5 月 20 日这一通行记载，不据此派生其他日期。

1.0 已经具备：迭代解析、缓存、DNSSEC 验证、端口与查询 ID 随机化、清理器、存根区域（stub zone）支持。换句话说，1.0 不是「能跑起来的骨架」，而是**带着完整安全模型的第一个版本**。官方历史文档里还保留了 1.0.2 的补丁公告、2008 年缓存投毒说明、超时与服务器选择算法说明、信任锚说明等技术文档，这些是研究早期设计取舍的一手材料。

1.4.5 许可与治理：BSD 与「谁在维护」

从 Java 原型到 C 实现，许可始终是 **BSD**（BSD 3-Clause 一类宽松许可）。这件事对运维的实际影响比想象中大：

- 发行版可以无阻碍地打包，嵌入式与路由器固件可以集成，也允许被其他项目以库形式链接（`libunbound`）；
- 没有 copyleft 的传染性要求，商业设备与内部工具可以按需修改；
- 反过来，BSD 也意味着**没有人为你的部署兜底**：安全公告、升级节奏、默认值变更都要自己跟。

治理结构上，NLnet Labs 承担主要开发与文档维护，开发讨论与用户支持通过公开的邮件列表与代码仓库进行。对本书读者更实用的一条经验是：**发行版仓库里的 Unbound 版本常常明显滞后于上游**，而 `unbound -V` 与 `unbound-control status` 才是判断本机实际能力的依据。判断某个参数是否可用，先确认版本，再查该版本的手册。

「BSD 许可」和「BSD 操作系统默认带 Unbound」是两回事。后者是可核验的具体事件——FreeBSD **10.0-RELEASE**（2014-01-20）的发布公告在 highlights 里同时写了两句：「Unbound has been imported to the base system as the local caching DNS resolver.」与「BIND has been removed from the base system.」[4]。也就是说 FreeBSD 用 Unbound 作为**本地缓存解析器**替换了基本系统里的 BIND，但这是**该版本 FreeBSD 的选择**，不是所有 BSD 系（OpenBSD / NetBSD / DragonFly）都如此，也不是「BSD 许可导致默认使用」。OpenBSD 同期主推的是自己的 `unwind` / `resolvd` 一类组件，按各自发行说明核对。

1.4.6 一条需求，三种读法

把「只做递归验证」这句口号落到运维语言里，可以拆成三个可验证的判断，后面各章的配置选择都会回到它们：

判断	具体含义	违反了会怎样
只做递归	不承载权威区域、不做区域传送、不做动态更新	权威侧的配置复杂度与故障半径被带进递归侧
默认验证	信任锚就位时按签名链判断应答，坏签名返回 SERVFAIL	无锚或 permissive 模式下，坏签名应答也会被放行

判断	具体含义	违反了会怎样
小而可审计	处理链由少数几个模块组成、行为可由日志与统计复现	出事时只能靠重启和猜，无法定位是哪一层缓存

第三条在本书后续章节里最容易被低估。运维真正需要的不是「代码行数少」这个事实，而是**故障时可复现的因果链**：一次 SERVFAIL 究竟是上游超时、验证失败、负缓存命中、还是过期缓存应答的刷新超时，必须能从日志与统计里读出来。第 1 章讲历史，第 2 章讲角色，第三篇讲验证命令，都是围绕这条能力展开的。

1.5 里程碑：每个特性回答了哪个问题

下面的梳理按「问题→回应」组织，而不是按版本号流水账。版本号用于定位，不用于记忆；具体引入版本请以官方手册对应章节与 NLnet Labs 发布说明为准，尤其是 Linux 发行版打包版本往往滞后于上游版本。

先给一张速查表。表中「版本」一列只标官方手册明确写出的引入/变更版本，其余一律留作「见官方手册」，避免把大致时期写成确定事实。

时期 / 版本	变化	回答的问题
2004 前后	Java 原型，BSD 许可，模块链设计	迭代解析器与可分离的验证器能否这样拆
约 2006	NLnet Labs 加入，转向 C 实现	原型如何变成可部署的生产实现
2008-05	1.0 发布	一个默认带随机化、清理器与 DNSSEC 验证的递归解析器
1.1	DLV 支持、统计改进	根未签名时信任锚从哪来
1.6.0	`serve-expired`	上游不可用时能否继续应答、提高命中率
1.10.0 / 1.14.0	RPZ 初版 / 完整触发器与动作	过滤策略能否跨实现、由外部源直接加载
1.11.0 / 1.23.0	引入 [RFC 8767] 行为 / 成为默认	过期缓存应答如何与标准对齐
1.12.0	下游 DoH	只有 HTTPS 能力的客户端如何接入加密解析
1.22.0 / 1.24.0	下游 DoQ / 可用标准 OpenSSL	QUIC 承载 DNS 与依赖库收敛
见官方手册	DoT 相关能力、libunbound、pythonmod、QMIN、cachedb	传输加密、嵌入式调用、模块扩展、元数据与共享缓存
见官方手册	`edns-buffer-size` 默认 1232	分片导致 TCP 回退的默认值修正

1.5.1 读版本说明的正确姿势

在逐条看特性之前，先约定一条方法，避免把「大致时期」写成「确定事实」：

- 官方手册里标注为 `New in version X.Y.Z` / `Changed in version X.Y.Z` 的，才写成确定版本；
- 其他只在任务描述、邮件列表或新闻稿里出现过的时期，一律写成「见官方手册与 NLnet Labs 发布说明」；
- 发行版打包版本与上游版本不是一回事，能力判断以本机 `unbound -V` 为准；
- 默认是「当前默认」而非「历代默认」，例如 `edns-buffer-size` 的 1232 来自 DNS Flag Day 2020 推荐，早期默认值不同。

速查表里几行故意留白：宁可去查一手来源，也不要一个看似精确、实际无法核验的版本号。

1.5.2 DLV：根尚未签名时的过渡方案

DNSSEC 的验证链需要一个信任锚 (trust anchor)。在根区尚未签署、或信任链上端不完整的年代，**DLV (DNSSEC Lookaside Validation, [RFC 4431])** 提供了一个”旁路信任仓库”：把本来应该从根逐级验证的锚，改为从一个 DLV 仓库查询。Unbound 1.1 加入了 DLV 支持（官方历史页在介绍 1.1 时明确提到 “Unbound 1.1 has DLV support and improved statistics”）。

DLV 的历史意义是**过渡性**的：它让验证器在根签名空窗期仍然有活可干。随着根区完成签名、[RFC 5011] 定义的信任锚自动更新机制成为常态，DLV 退出主流配置。今天再提它，是为了说明”信任锚从哪来”这个问题曾经有过别的答案——第三篇第 10 章讲信任锚时，会看到为什么转发器上配自动信任锚是行不通的。

1.5.3 信任锚的三种状态

DLV 之所以存在，是因为「验证」这件事有一个容易被忽略的前提：**验证器必须先有一个可信的起点**。这个起点叫信任锚（trust anchor），通常就是根区的密钥。围绕它，解析器实际会呈现三种状态，第三篇会详细展开，此处先建立词汇：

状态	行为	运维可见的现象
无信任锚	透传应答，不验证签名	坏签名域照样返回答案，看起来一切正常
permissive 模式	记录日志但放行	日志里有验证失败，客户端仍拿到应答
真验证	坏签名→ SERVFAIL；好签名→应答带 AD 标志	坏签名域整体失败，好签名域可凭 AD 位确认

注意第三种状态里的 AD 位：它是**解析器向客户端声明「我验证过了」**的标志，不是数据本身自带的属性。历史章节只需记住：DLV 与 [RFC 5011] 都是在解决「起点从哪来、怎么更新」，而不是在解决「传输路上有没有被偷看」。

1.5.4 DNSSEC 默认验证：从可选到基线

早期 DNSSEC 验证常被视为”要显式打开的高级特性”。Unbound 的演进方向相反：**验证成为默认假设**，运维要操心的是”锚是否就位” “验证失败要不要放行”，而不是”要不要启用验证”。与之配套的是三种可观测的状态：无信任锚时透传不验证坏签名、permissive 模式记录日志但放行、真正验证时坏签名直接返回 SERVFAIL 且好签名带 AD 标志。

这个转变把安全责任从”运维记得勾选”移到了”实现默认如此”。默认开着不等于真的在验证，锚缺失或被显式降级时，解析器看起来一切正常。

1.5.5 libunbound 与 pythonmod：把能力交给应用程序

libunbound 让应用程序以库的形式调用解析能力（同步与异步接口、线程上下文、DNSSEC 结果检查），不必自己起守护进程，也不必依赖系统存根解析器的行为。**pythonmod**（官方手册称 Pyunbound / Python 模块）则允许用 Python 编写 Unbound 的模块，插入到处理链里做定制逻辑。

这两项是”模块链可裁剪”设计的直接兑现。它们的取舍也很清楚：库接口把信任边界移到了进程内，适合做专门的探测、校验或嵌入工具；Python 模块带来灵活性，但也把一段自定义代码放进了解析热路径，性能与故障半径都要自己承担。第一篇只交代它们为何存在，配置与调用放到后续实战章节。

1.5.6 模块链与配置的对应关系

`libunbound` 与 pythonmod 不是两个孤立特性，它们是同一套模块链设计的两种出口。官方手册的配置参考里，`module-config` 一行决定处理链上有哪些模块、按什么顺序排列；例如子网缓存模块要求排在迭代器之前，cachedb 模块则给出 `validator cachedb iterator` 这样的次序示例。

对运维的含义是：**功能不是开关，而是链上的位置**。插错位置会表现为「参数明明写了却不生效」，而不是报错。这个现象在第二篇第 4–7 章按角色配置时会反复出现，本篇只把它作为历史遗产记录下来：2004 年的原型为了「能组装成不同形态的解析器」而设计了模块链，二十年后它依然决定了配置文件的写法。

1.5.7 加密传输：DoT、DoH、DoQ

DoT（DNS-over-TLS，[RFC 7858]）用专用端口 853 把 DNS 报文直接装在 TLS 里；**DoH（DNS-over-HTTPS，[RFC 8484]）**在 DNS 报文与 TLS 之间加了一层 HTTP（Unbound 的下游 DoH 要求 HTTP/2，靠 ALPN 协商，以满足流水线与乱序处理的性能需要）；**DoQ（DNS-over-QUIC，[RFC 9250]）**走 QUIC，用加密的多路复用承载 DNS。

ALPN 是什么：ALPN 是 **Application-Layer Protocol Negotiation**（应用层协议协商），定义在 [RFC 7301]。它是 TLS 握手里的一个扩展：客户端在 ClientHello 里列出自己支持的应用协议标识（如 `h2`、`http/1.1`），服务端在 ServerHello 里挑一个，双方还没传应用数据就已约定好上层协议。DoH 依赖 HTTP/2，而 HTTP/2 在 TLS 上正是靠 ALPN 里的 `h2` 标识协商出来的——本书第 12 章 nginx 回源必须走 h2c / `grpc_pass` 才能接上 dnsdist 2.x，根子就在这里。注意 ALPN 只协商「TLS 之上跑什么协议」，与 DNSSEC、DoT 的 853 端口、DoQ 的 ALPN 标识 `doq` 都属不同层的概念，不要混为一谈。[5]

时间线上有两个容易被混淆的点，值得写清：

- 官方手册的 DoH 章节记载，Unbound **早在 2011 年就能处理 TLS 加密的 DNS 报文**，远早于 IETF DPRIVE 工作组启动 DoT 规范工作；而作为正式配置能力的下游 DoT/DoH 支持是后来的事。把”代码里出现过 TLS 处理”和”作为标准特性发布”混为一谈，是常见的年份错误来源。
- 下游 DoH 在官方手册中标注为 “New in version 1.12.0”，下游 DoQ 标注为 “New in version 1.22.0”（且早期需要带 QUIC 支持的 OpenSSL 分支，1.24.0 起可用 OpenSSL 3.5.0 及以上）。任务描述里提到的”1.7 前后”可视为 DoT 相关能力趋于可用的大致时期，**确切版本以官方手册对应章节为准**。

三者解决的问题是**传输加密与隐私**（链路上的观察者、客户端对解析器的身份认证），不解决”应答数据是否被伪造”——那是 DNSSEC 的职责。DoT 与 DoH 的服务端/客户端配置在第三篇第 11、12 章，协议层对比在第四篇第 17 章。

1.5.8 三种加密传输的角色边界

把三者并列时，最容易写错的是「谁在服务端、谁在客户端」：

协议	承载	端口	在本章语境里的定位
DoT	DNS 报文直接装在 TLS 里	853	专用端口，链路边界清晰，便于策略管控
DoH	DNS 报文经 HTTP/2 再经 TLS	443	与 Web 流量共存，穿透性好但边界模糊
DoQ	DNS 报文经 QUIC	UDP 853 类专用端口	加密 + 多路复用，依赖库较新

三者都只覆盖**客户端↔解析器**或**解析器↔上游**这一段链路，都不改变「解析器仍需自己做迭代、缓存与验证」这一事实。第 1 章交代它们出现的顺序与动机，第三篇第 11、12 章分别落地 DoT 与 DoH 的服务端/客户端，第四篇第 17 章做协议层对比。

1.5.9 过滤能力：RPZ 与本地策略

Unbound 一直有 `local-zone` / `local-data`（对指定名字直接给本地应答）与 `respip`（改写含指定地址的应答）这两类本地手段，但它们都是 Unbound 专有语法。**RPZ（Response Policy Zones）**的意义在于用一种跨实现通用、且可由外部源通过区域传送直接加载的格式表达过滤策略，运维不必为不同厂商的解析器各维护一份策略、也不必手工转换格式。

官方手册标注 RPZ 于 1.10.0 首次引入（仅 QNAME 与 Response IP Address 触发器），1.14.0 补齐完整的触发器与动作集合。它的取舍和过滤本身一样：策略越集中，误拦时的排查面越大；而本地策略直答与递归验证之间的优先级，也会直接影响故障现象。

1.5.10 过滤与验证的优先级

过滤策略一旦介入，就会和「递归验证」这个默认职责产生张力：本地直答的名字没有签名链可验，改写过的应答也不再是权威服务器原样给出的数据。因此 RPZ、`local-zone` / `local-data`、`respip` 这类能力在工程上必须明确两件事：

1. 这条策略作用于**哪个信任域**（是只对内网客户端生效，还是对所有经过本实例的查询生效）；
2. 策略命中时，客户端**能不能分辨**这是本地策略而非权威数据。

第二篇会用「一台机器上跑多个实例」来隔离不同过滤表，第 3 章给出动机。历史章节只需指出：RPZ 解决的是策略表达与分发的标准化问题，它不替代验证，也不改变验证器的位置。

1.5.11 过期缓存应答：serve-stale 与 [RFC 8767]

缓存的 TTL 是权威服务器与缓存解析器之间的契约：过期就不该再用。**过期缓存应答（serve-stale）**是对这份契约的有条件例外：[RFC 8767] 规定，当 TTL 到期后数据无法从权威侧刷新时，可以把它当作未过期来使用，目的是在权威或链路不可用时提高韧性。

Unbound 的 `serve-expired` 早于该 RFC 存在，最初目的是提高缓存命中率；官方手册记载 1.6.0 引入、1.11.0 引入 [RFC 8767] 行为、1.23.0 起 RFC 行为成为新的默认。这里有一个在实战中反复被误读的细节：`serve-expired-client-timeout` 的单位是**毫秒**（默认 1800，即 1.8 秒），语义是”先尝试刷新，超过这个时间才回过期缓存”，不是 1800 秒。第三篇第 10 章与第五篇第 22 章会把它与负缓存放大故障的现象放在一起讲。

1.5.12　可用性与正确性的交换

过期缓存应答是把「正确性」换「可用性」：TTL 到期意味着权威服务器不再授权你使用这份数据，[RFC 8767] 给的例外是「刷不到就用旧的」。这个交换有一个常被忽略的放大效应——**负缓存（negative caching）也会放大故障**。上游一次瞬时 NXDOMAIN 会按 SOA minimum 被缓存（常见 3600 秒），权威侧修好之后客户端仍在失败；若链路上有多个实例，每一层的缓存都要刷，只刷最靠近客户端那一层没用。

因此读 [RFC 8767] 时，要把它和 [RFC 2308] 的负缓存一起读：一个决定「旧数据能不能用」，一个决定「不存在这件事能被记多久」。第五篇第 22 章的故障案例会把这两种现象放在同一条排查链里。

1.5.13　查询名最小化：从 [RFC 7816] 到 [RFC 9156]

查询名最小化（QNAME minimisation）的想法很朴素：向顶级域服务器询问时，没必要把完整的 `www.example.org` 都发过去，只需要问到定位下一级委派所必需的标签数。它减少的是发给上级的 QNAME 标签数量，从而限制元数据聚合的范围。[RFC 7816] 是早期的实验性描述，[RFC 9156] 是后续修订。

必须与加密传输划清界限：**查询名最小化不是加密**，它不抗全路径嗅探；如果根这一跳本来就走本地根区镜像、不出本机，收益主要落在 TLD 层。另一个方向上的教训是严格模式的兼容性：某些大型 CDN 的 CNAME 区违反 [RFC 8020]（父名返回 NXDOMAIN、子名却存在），`qname-minimisation-strict: yes` 会在中间层拿到 NXDOMAIN 就直接收工，导致整站静态资源解析失败。第三篇第 13 章讲参数与兼容坑，第四篇第 17 章讲 [RFC 7816] → [RFC 9156] 的演进，本篇只交代它在历史里的位置。

1.5.14　隐私特性之间的分工

把查询名最小化、加密传输、DNSSEC 三者的边界画成一张表，比反复用文字强调更不容易混淆：

机制	防的是谁	不防的是谁	相关 RFC
DNSSEC 验证	伪造/篡改应答的中介	链路上看得到查询内容的观察者	[RFC 4033] 等
查询名最小化	上级服务器对完整查询名的元数据聚合	全路径嗅探者、本机解析器本身	[RFC 7816] / [RFC 9156]
DoT / DoH / DoQ	客户端与解析器之间的链路观察者	解析器向权威侧发出的迭代查询	[RFC 7858] / [RFC 8484] / [RFC 9250]

最后一列尤其重要：**加密传输保护的是一段链路，不是整条解析路径**。一台本地递归解析器即使对客户端提供 DoH，向上游权威服务器发出的迭代查询依然可能走明文 53。这正是「为什么需要一个自己做迭代与验证的本地实现」在隐私语境下的答案——只有把递归放在自己可控的机器上，才谈得上整条链路的策略。

1.5.15　EDNS 1232：被 MTU 教育出来的一次默认值变更

EDNS(0) 全称 **Extension Mechanisms for DNS (Version 0)**，定义在 [RFC 6891]：它在不改动 DNS 报文头的前提下，用一条伪资源记录 **OPT** 承载扩展能力——通告「我能重组多大的 UDP 应答」、携带 ECS / DNS Cookie 等选项、以及 DO 位（DNSSEC OK）。原始 DNS 的 UDP 报文被限制在 512 字节，带 DNSSEC 签名的应答放不下，EDNS(0) 就是为此而来。

早期运维习惯把 `edns-buffer-size` 调大到 1472 甚至更高，以减少 TCP 回退。但当链路上存在 MTU 1280（IPv6 最小 MTU、部分隧道）或 PPPoE 1492 时，大报文会分片；分片一旦被丢弃，表现不是”报文变小”，而是**等超时再重试 TCP**，延迟陡增。

Unbound 当前官方默认值为 **1232**，来源是 **DNS Flag Day 2020** 的推荐 [6]；`max-udp-size` 同样默认 1232，但两者方向不同：`edns-buffer-size` 是**出向**通告给对端的重组缓冲大小，`max-udp-size` 只管**入向**应答的最大 UDP 尺寸。把入向调小却期望出向不分片，是这类故障里最常见的误判。这个数字看起来只是个默认值，实际上是”协议设计向真实链路条件让步”的一个标本。

DNS Flag Day 是什么：它不是学术会议，而是由 DNS 软件厂商、权威运营商与递归运营商共同发起的**协同变更日**——约定在同一天一起改掉某个长期互操作问题，让「谁先改谁吃亏」的僵局被打破。2019 年那次针对权威服务器对 EDNS 的兼容性，2020 年这（2020-10-01）针对 **IP 分片**带来的运维与安全问题，官网给出的核心建议就是把默认 UDP 缓冲收敛到 1232，避免大应答在 MTU 1280 的链路上分片 [6]。所以 1232 不是某个 RFC 的强制值，而是行业协同的默认基线；链路条件确实更好时，运营商仍可显式调大。

`edns-buffer-size` 与 `max-udp-size` 的默认值同为 1232，但方向相反，是实战中最容易调反的一对：

参数	作用方向	管的是什么
`edns-buffer-size`	出向	向权威服务器通告「我能重组多大的 UDP 应答」
`max-udp-size`	入向	本实例返回给客户端的 UDP 应答最大尺寸

把入向调小、却期望出向不分片，不会有任何效果；反过来把出向调到 1472，在 MTU 1280 的隧道或 PPPoE 1492 链路上会产生分片，分片被丢后表现是**等超时再重试 TCP**，而不是「报文自动变小」。1232 是默认基线，不是可随意调大的性能旋钮。

1.5.16 cachedb / Redis 与 ECS 取舍

cachedb 的意义：Unbound 默认把缓存放在进程内存里，重启即丢、多个实例各存一份。cachedb 模块把缓存挪到外部后端，解决两件运维事——**多实例共享一份缓存**（冷启动不再各自预热）与**重启后缓存仍在**（持久）。代价同样明确：Redis 后端是同步通信，后端变慢会直接拖住查询处理；缓存里的数据还该不该用，模块不替你判断，仍由 TTL 与管理员负责。它和 ECS 是不同维度：cachedb 管「缓存放哪儿」，ECS 管「缓存键里要不要加客户端网段」。

EDNS 与 HTTPDNS 有关系吗：没有直接关系，只在「都跟 DNS 有关」这点上同名。EDNS(0) 是 **DNS 报文内部的扩展机制**（OPT 伪记录，[RFC 6891]），作用对象是标准 DNS 查询；HTTPDNS 是**应用层的私有 HTTP API**（第 37 章），根本不发 DNS 报文，自然也用不上 EDNS OPT。两者唯一的交集是「都想解决标准递归链路的某个不足」，但一个在协议内扩展、一个绕开协议另起炉灶。不要把 HTTPDNS 返回的 `client_ip` 与 EDNS Client Subnet（[RFC 7871]）混为一谈：ECS 是把网段塞进 DNS 报文给权威侧，HTTPDNS 是在 TCP 连接里直接看到客户端出口 IP。

ECS 则是 Unbound 明确”取舍”而非”拥抱”的领域。它的动机是让权威服务器按客户端所在网段返回更近的 CDN 节点，但代价是把客户端地址信息向上游扩散，同时把缓存从”按查询名”变成”按查询名 + 网段”，命中率与隐私都受损。官方手册的子网缓存模块说明因此把发送范围做成显式白名单（`send-client-subnet` / `client-subnet-zone`），并提醒启用后缓存与性能会受影响。

把这两项和前面的查询名最小化、加密传输放在一起看，能看到一条贯穿十几年的主线：**Unbound 对”多泄露一点信息换一点便利”的功能，默认倾向是关掉或收窄，而不是默认打开**。这不是性能判断，是隐私与安全基线上的选择。

1.5.17 一条贯穿十几年的默认立场

把 ECS、cachedb、查询名最小化、加密传输放在一起看，会发现一条一致的设计倾向：**凡是「多泄露一点信息、或多引入一个外部依赖，换一点便利」的功能，默认都关着或收得很窄**。ECS 需要显式白名单，cachedb 需要显式编译并启用模块，查询名最小化的严格模式默认不咬人，加密传输需要自己配证书与端口。

这不是保守，而是把「信任域」的判断留给部署者：ECS 与 cachedb 需要显式启用，加密传输需要自己配证书与端口，查询名最小化的严格模式要按自身兼容性评估后再决定是否打开，默认值与适用条件均见官方手册。不同上游、不同过滤表、不同信任域本就应该由不同实例承担，而不是在一个实例里用一堆开关互相覆盖——四种角色的串联图见第 3 章。

1.6 本章要点

- Unbound 的核心动机是打破递归解析器的**单一化风险**：需求文档把”代码多样性”“除配置外可替代 BIND”“尽可能小的够用组件”列为目标，并把”不做权威服务器”“不要过多特性”列为明确的非目标。
- 2004 年由 Jakob Schlyter（Kirei）与 Roy Arends（Nominet）提出架构、VeriSign 与 EP.NET 资助 Java 原型（David Blacka、Matt Larson 等）；约 2006 年 NLnet Labs 加入并承担 C 实现（借鉴 NSD 经验，Wouter Wijngaards 主开发）；2008-05-20 随 1.0 公开发布，BSD 许可。早期参与者与年份在不同史料中表述略有出入，以官方手册历史章节与 NLnet Labs 发布说明为准。
- 2008 年 Kaminsky 缓存投毒事件把”查询熵是安全边界”和”只有 DNSSEC 能彻底解决伪造”钉死；Unbound 1.0 因端口/查询 ID 随机化与清理器设计，无需为该漏洞紧急补打抗伪造补丁。
- 里程碑按问题归类：DLV（根未签名时的过渡锚）、DNSSEC 默认验证、libunbound/pythonmod（模块链可裁剪）、DoT/DoH/DoQ（传输加密，ALPN 见 [RFC 7301]）、RPZ（跨实现过滤策略）、serve-stale（[RFC 8767] 可用性）、QMIN（[RFC 7816] → [RFC 9156] 元数据）、EDNS(0) 1232（DNS Flag Day 2020, [RFC 6891]）、cachedb/Redis 与 ECS（共享缓存与隐私取舍）。

- DNSSEC 是**数据完整性/来源认证**，DoT/DoH/DoQ 是**传输加密/隐私**，查询名最小化不是加密；三类机制互不替代，这是全书一致口径。
- FreeBSD 10.0-RELEASE（2014-01-20）把 Unbound 引入基本系统作本地缓存解析器并移除 BIND [4]；这是该发行版的事件，不是「BSD 许可→默认 Unbound」的推论。EDNS(0) 属标准 DNS 报文内扩展，与 HTTPDNS 私有 API 无直接关系。

1.7 参考资料

本章引用标记：[1] CERT/CC VU#800113；[2] CVE-2008-1447；[3] NLnet Labs 关于 2008 缓存投毒的说明；[4] FreeBSD 10.0-RELEASE 发布公告；[5] [RFC 7301] ALPN；[6] DNS Flag Day 2020 官网。正文里的 [1][2] 等即指向下列条目。

- [1] CERT Coordination Center, *VU#800113 — Multiple DNS implementations vulnerable to cache poisoning*（2008-07；含 Kaminsky 披露、TXID/源端口熵与厂商声明）：https://www.kb.cert.org/vuls/id/800113
- [2] MITRE / NVD, *CVE-2008-1447*（DNS 缓存投毒）：https://www.cve.org/CVERecord?id=CVE-2008-1447
- [3] NLnet Labs, Unbound 官方手册·2008 年 DNS 缓存投毒漏洞说明: https://unbound.docs.nlnetlabs.nl/en/latest/reference/history/dns-cache-poisoning-vulnerability.html
- [4] FreeBSD Release Engineering Team, *FreeBSD 10.0-RELEASE Announcement*（2014-01-20, highlights 中 Unbound 引入 / BIND 移除）：https://www.freebsd.org/releases/10.0R/announce.html
- [5] RFC Editor, [RFC 7301] *Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension*: https://www.rfc-editor.org/rfc/rfc7301
- [6] DNS Flag Day, *DNS flag day 2020*（2020-10-01, IP 分片与 1232 默认缓冲）：https://www.dnsflagday.net/2020/
- RFC Editor, [RFC 6891] *Extension Mechanisms for DNS (EDNS(0))*: https://www.rfc-editor.org/rfc/rfc6891
- RFC Editor, [RFC 5452] *Measures for Making DNS More Resilient against Forged Answers*: https://www.rfc-editor.org/rfc/rfc5452
- NLnet Labs, Unbound 官方手册·历史章节：https://unbound.docs.nlnetlabs.nl/en/latest/reference/history/index.html
- NLnet Labs,Unbound 官方手册·早期需求文档(目标与非目标、项目起源):https://unbound.docs.nlnetlabs.nl/en/latest/reference/history/requirements.html
- NLnet Labs, Unbound 官方手册·Java 原型说明：https://unbound.docs.nlnetlabs.nl/en/latest/reference/history/prototype-resolver.html
- NLnet Labs, Unbound 官方手册·2008 年 DNS 缓存投毒漏洞说明：https://unbound.docs.nlnetlabs.nl/en/latest/reference/history/dns-cache-poisoning-vulnerability.html
- NLnet Labs, Unbound 官方手册·1.0.2 补丁公告：https://unbound.docs.nlnetlabs.nl/en/latest/reference/history/patch-announce102.html
- NLnet Labs, Unbound 官方手册·`unbound.conf(5)` 配置参考：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html
- RFC Editor, [RFC 8767] *Serving Stale Data to Improve DNS Resiliency*: https://www.rfc-editor.org/rfc/rfc8767.txt

第二章

DNS 基础与主流实现横评

一台递归解析器上出现的”解析失败”，可能来自命名空间里的任何一层：本地存根解析器没发出去、缓存里有一条过期的 NXDOMAIN、委派链上某一级给了不该给的 glue、CNAME 链断了、EDNS 通告的缓冲大小导致分片被丢。不把 DNS 的数据模型和解析全过程先讲透，后面所有的加固配置都会变成”照抄参数但不知道在防谁”。

本章前半部分是通用入门，与具体产品无关；后半部分是横评，按角色和取舍对照，重点放在用户最容易混淆的 BIND 与 dnsmasq 上。全章不贴长配置块，配置属于第二篇起的内容。

2.1 命名空间、委派与区域

2.1.1 一棵倒过来的树

DNS 的命名空间是一棵树，根（root）写作 .，往下依次是顶级域（TLD，如 `org`）、二级域（如 `example.org`）、以及更下层的名字。完整域名（FQDN）从最具体的标签一路写到根，例如 `www.example.org.`，末尾那个点表示根；日常书写常省略最后一个点，但**在配置文件里省略与不省略语义不同**，这是运维踩坑的高频来源之一。

每个标签最长 63 字节，完整域名总长不超过 253 字节（wire 格式下的常见表述）。这里先交代 **wire format**：它指 DNS 报文在**线路上传输时的二进制格式**，由 [RFC 1035] 定义——固定 12 字节报文头（ID / 标志 / 各段计数）+ question / answer / authority / additional 四段，名字用「长度字节 + 标签」的编码，类型与类别用数字而非文本。「同一份 DNS wire format 装在 UDP/53、TCP/53、DoT、DoH 里」指的就是这一串字节；DoH 只是换了个外壳，HTTPDNS（第 37 章）则连这串字节都没有。名字不区分大小写，但 wire 报文会保留查询名的大小写——这一性质后来被 0x20 编码用作额外的熵来源，第 1 章提到过。

树上的每个节点可以拥有一组**资源记录（resource record, RR）**。相同名字、相同类型、相同类别的记录组成一个 **RRset（resource record set，资源记录集）**，RRset 是缓存与 DNSSEC 签名的基本单位，不是单条记录。理解这一点很重要：TTL 作用于 RRset，签名也覆盖整个 RRset；“一条 A 记录过期了”严格说应是”这个 A RRset 的 TTL 到了”。

RRset 举例（这是最常见的一处误解）：针对 `www.example.org.` 查到一条 A 记录和一条 AAAA 记录，它们**不是同一个 RRset**——类型不同（A / AAAA）、名字类别相同，但 RRset 按 `(owner, type, class)` 划分，所以是两个 RRset：A RRset 与 AAAA RRset，各自有各自的 TTL 与 RRSIG。同理 MX 记录是**另一个** RRset；一个 MX RRset 里可以有多条 MX 记录（不同优先级），它们共享一个 TTL、被**一条** RRSIG 整体签名。`www.example.org.` 若有两条 A 记录（例如 `192.0.2.10` 与 `192.0.2.11`），这两条才属于**同一个** A RRset。缓存里刷 `www.example.org A` 只会淘汰 A RRset，不会动 AAAA / MX。

2.1.2 区域与委派

区域（zone）是命名空间里由同一管理者负责、并被作为一个单元维护的连续部分。区域与域（domain）不是同义词：一个域可以包含多个区域，因为其中某些子树被委派给了别人。

委派（delegation）发生在父区域里：父区域用 NS 记录声明”这个子树由这几台权威服务器（authoritative server）负责”。例如 `org` 区域里为 `example.org` 写入 NS 记录，就把 `example.org` 及其以下的名字交了出去。负责该子树的服务器称为该区域的权威服务器，它给出的应答在报文里带 **AA（Authoritative Answer）**标志。

委派带来两个必须记住的推论：

- 父区域**不负责**子区域里的具体记录，它只知道”去问谁”；因此父区域的数据变了，子区域的记录不会自动跟着变；
- 权威服务器只对**自己被委派的区域**有权威。一个应答里夹带了不属于自己区域的记录，就是典型的越界数据，解析器必须能识别并丢掉。第 1 章说的清理器（scrubber）与 `harden-glue` 处理的正是这一类：域外夹带的 glue 是经典缓存投毒面，且**它与 DNSSEC 无关**，任何模式都该开。

2.1.3 胶水记录：委派的自举问题

如果 `example.org` 的 NS 记录写的是 `ns1.example.org`，那么要查 `ns1.example.org` 的地址，又得先问 `example.org` 的权威服务器——形成循环。解决办法是：**父区域在委派应答的附加段（additional section）里，一并给出这些 NS 名字的 A / AAAA 记录**，这些记录就叫**胶水记录（glue record）**。

胶水记录有两条性质，直接决定了它的风险：

1. 它**不是权威应答**。父区域给的是”顺带提供的地址提示”，不是对该名字的权威数据；
2. 它位于附加段，历史上很多实现会不加分辨地把它写进缓存。

第 2 条就是缓存投毒的经典入口：伪造的应答里夹带一条指向攻击者地址的 glue，一旦被缓存，后续对该域的所有查询都会被导向攻击者。Unbound 的处理方式是只放行特定条件下的 glue（委派中出现、类型为 A / AAAA、名字确实出现在 authority 段的 NS 记录里），并且把它标记为”来自附加段”，不会作为对客户端的直接应答返回；对这个名字的直接查询会另走一遍解析，用来自 answer 段的权威数据覆盖它。

这段机制的实践含义是：**别把”有 glue 就能解析”当成理所当然**，也别把”开了 DNSSEC 就不用管 glue”当成结论。两者处理的是不同层面的问题。

2.2 递归与迭代：谁在替谁跑腿

2.2.1 两种查询模式

递归查询（recursive query）的意思是：客户端把整个解析任务交给服务器，服务器要么给出最终答案，要么给出明确的失败，中间过程客户端不参与。报文里用 **RD（Recursion Desired）**位表示”我希望你替我递归”；服务器用 **RA（Recursion Available）**位回答”我提供递归服务”。

迭代查询（iterative query）的意思是：被问的服务器不替你继续跑，只给出它知道的最好的答案——可能是最终记录，也可能是”我不知道，但你可以去问这几台”（referral，指引应答）。递归解析器向根、TLD、各级权威服务器发出的，就是这种不带 RD 的迭代查询：每得到一次 referral，就换一台服务器再问，直到拿到答案或明确的不存在证明。

于是”递归解析器”这个名字其实描述的是**它对客户端的承诺**，而不是它对外的工作方式：对客户端它做递归，对上级它做迭代。第 3 章会把它拆成四种角色。

2.2.2 一次 `www.example.org` 的追踪

假设本机存根解析器把查询交给一台本地递归解析器，缓存为空：

1. **存根→递归解析器**：`www.example.org. A?`，带 RD。存根不打算自己问根，它要一个最终答案。
2. **递归解析器→根服务器**：`www.example.org. A?`，不带 RD。根不解析具体主机，返回对 `org` 的 referral：`org` 的 NS 列表与对应 glue。
3. **递归解析器→ `org` 的权威服务器**：同样问 `www.example.org. A?`，不带 RD。`org` 返回对 `example.org` 的 referral。
4. **递归解析器→ `example.org` 的权威服务器**：`www.example.org. A?`，不带 RD。这里可能出现三种结果：
   - 直接给出 `www.example.org` 的 A / AAAA RRset，带 AA；
   - 给出一条 CNAME，指向另一个名字，于是从第 4 步重新开始（CNAME 链见后文）；
   - 给出 NXDOMAIN 或 NODATA，表示名字不存在或该类型不存在（负缓存见后文）。
5. **递归解析器→存根**：最终答案，带 RA；若做了 DNSSEC 验证且签名链通过，还会带 **AD（Authenticated Data）**位。

注意第 2、3 步：递归解析器每次都把**同一个完整查询名**发给上级，这正是查询名最小化（QNAME minimisation）要改掉的行为。它只减少发给上级的标签数，不加密报文；相关 RFC 的演进在第四篇第 17 章，本篇不展开。

2.2.3 追踪示例里的可观测点

真要排查时，`dig +trace` 能模拟这条迭代链，但它与真实递归解析器的行为有差别：`+trace` 是从**本机**逐级发问，不会经过本地缓存，也不会体现解析器自己的端口随机化、超时算法、验证与缓存策略。因此它适合看”委派链在哪一级断了”，不适

合判断”我的解析器为什么返回 SERVFAIL”。

一条实用的分工是:

想看什么	看哪一层	常见误判
委派链是否完整、哪一级 NS 配错	逐级迭代追踪	把 referral 当成失败
本地解析器是否命中缓存、是否验证	解析器统计与应答标志位	只看 TTL 就以为数据新鲜
客户端实际拿到什么	从客户端发查询、看 RA/AD/状态	在服务器上测完就以为客户端一样

第三篇会给出可复制的验证命令；此处只需建立”链路上每一层都可能单独出问题”的直觉。第 5 章的故障案例里，同一个 NXDOMAIN 在链路上被层层缓存、层层刷不干净，就是这个直觉的实战版本。

2.3 资源记录、缓存与 TTL

2.3.1 常见记录类型

类型	作用	与本书加固的关系
A / AAAA	名字→ IPv4 / IPv6 地址	最常见的应答，也是过滤与改写策略的作用对象
NS	委派：该区域由哪些权威服务器负责	referral 的核心，glue 依附于它
SOA	区域的起始授权记录，含序列号、刷新/重试/过期 timers、minimum TTL	负缓存时长、区域传送与主备同步的依据
CNAME	名字的规范名别名	形成链，链上任一环失败整条失败
PTR	反向解析	日志与反查，不是本章重点
MX / TXT / SRV / SVCB / HTTPS	邮件路由、文本、服务发现等	现代客户端常查，`ndots` 放大时会一并出现

SVCB 是什么：SVCB 全称 **Service Binding**（服务绑定），定义在 [RFC 9460]。它是一条通用资源记录，用来在**发起连接之前**就告诉客户端「这个服务可以从哪些替代端点访问、各自带什么参数」——例如优先用 HTTP/3、端口是多少、可用的 IP hint、以及 TLS 相关配置。它还支持 `AliasMode`，能在区域顶点（apex）做别名，这是 CNAME 在 apex 上做不到的。**HTTPS 记录**是 SVCB 专用于 HTTP 的变体（[RFC 9460] 一并定义），浏览器查它来决定用不用 HTTP/2、HTTP/3、ECH。本书第 13 章的 DDR 压制要挡的 `_dns.resolver.arpa. SVCB`，正是借 SVCB 来「发现加密解析器」：客户端查一条 SVCB 就能拿到 DoT / DoH / DoQ 端点，所以本地解析器用 `always_nodata` 让它查不到。第 30 章讨论信任数据时会再回到 SVCB / HTTPS 的字段。| DNSKEY / RRSIG / NSEC / NSEC3 / DS | DNSSEC 的密钥、签名、否认存在与委派签名 | 完整性/来源认证，与加密传输不是一回事 | | OPT | 并非真实记录，是 EDNS(0) 的伪记录 | 通告 UDP 缓冲大小、携带选项（如 ECS）|

要特别区分 **DS** 与 **NS**：NS 把子树交给谁，DS 把父区域对子区域密钥的承诺写进父区域，两者共同构成 DNSSEC 的信任链。第 3 章讲 AD/CD 位时会再用到这个区分。

2.3.2 缓存与 TTL

每条 RRset 带一个 **TTL（time to live）**，单位是秒，表示”这份数据可以被缓存多久”。递归解析器命中缓存时，返回给客户端的 TTL 是**剩余** TTL，不是原始值——所以连续查同一个名字，看到的 TTL 会递减，这是正常现象，不是数据变了。

TTL 的本质是权威服务器与缓存解析器之间的契约：到期后就不该再用。[RFC 8767] 定义的**过期缓存应答（serve-stale）**是对这份契约的有条件例外——刷不到新数据时可以暂用旧的，以提高韧性。Unbound 的 `serve-expired` 早于该 RFC 存在，官方手册标注 1.6.0 引入、1.11.0 引入 RFC 行为、1.23.0 起 RFC 行为成为默认。

这里有一个在实测中反复被读错的细节：`serve-expired-client-timeout` 的单位是**毫秒**（默认 1800，即 1.8 秒），语义是”先尝试刷新，超过这个时间才回过期缓存”，不是 1800 秒。把它当成 1800 秒，会得出完全相反的超时判断。

2.3.3　缓存的层级性

一次客户端查询背后往往不止一层缓存：浏览器或应用有自己的缓存，操作系统存根解析器可能有缓存，本地递归解析器有缓存，前面还可能串着转发器或节点级缓存。于是会有两个反直觉的后果：

- **缓存命中率不等于”权威侧健康”**：上游已经全断，本地仍可能靠缓存正常应答几十分钟；
- **故障修复不等于”客户端恢复”**：某一层缓存里存着旧的负面结果，修好权威后那一层仍会继续返回失败，必须逐层刷。

第 7 章讲多实例与可观测性时会把”链路上每个实例都要刷缓存”写成检查清单；本篇只建立模型：**缓存是分层的，任何一层都可能是你排查时漏掉的那一层**。

2.3.4　负缓存：不存在的东西也会被记住

[RFC 2308] 规定了**负缓存（negative caching）**：NXDOMAIN（名字不存在）与 NODATA（名字存在但没有该类型）也可以被缓存，时长取 SOA 记录里的 minimum 字段与 SOA 自身 TTL 中较小的一个，常见值是 3600 秒。

它的价值是挡住对不存在名字的重复查询；代价是**故障放大**。上游瞬时抖动返回一次 NXDOMAIN，权威侧恢复后客户端仍会失败整整一个负缓存周期。更麻烦的是，如果链路上串了多个实例，只刷最靠近客户端那一层没用——每一层都可能各自缓存了一份 NXDOMAIN。

负缓存与过期缓存应答于是构成一组对照：前者是”把失败记久一点”，后者是”把旧数据再用一会儿”。两者都改变 TTL 的字面语义，方向却相反。第五篇第 22 章的故障案例会把它们放在同一条排查链里。

2.3.5　EDNS(0) 与 1232

原始 DNS 的 UDP 报文被限制在 512 字节，放不下带 DNSSEC 签名的应答。**EDNS(0)** 全称 **Extension Mechanisms for DNS (Version 0)**（[RFC 6891]），用一条伪记录 OPT 扩展了报文：客户端在 OPT 里通告”我能重组多大的 UDP 应答”，还可以携带各种选项（ECS 就是其中之一）。

通告值不是越大越好。Unbound 的 `edns-buffer-size` 当前官方默认 **1232**，来自 DNS Flag Day 2020 的推荐；把出向值调大到 1472，在 MTU 1280（IPv6 最小 MTU、部分隧道）或 PPPoE 1492 链路上会分片，分片被丢后表现是**等超时再重试 TCP**，而不是自动变小报文。同一份手册里 `max-udp-size` 也默认 1232，但它只管**入向**应答；把入向调小却指望出向不分片，是常见误判。

参数	方向	管什么
`edns-buffer-size`	出向	向权威服务器通告的重组缓冲大小
`max-udp-size`	入向	本实例返回客户端的 UDP 应答最大尺寸

EDNS 选项里还有一类与隐私直接相关的：ECS（[RFC 7871]）把客户端网段信息带给权威服务器以换取更近的 CDN 节点，代价是泄露客户端位置信息、并把缓存键从”查询名”变成”查询名 + 网段”。Unbound 把它做成显式白名单式的能力。

2.4　CNAME 链与解析过程中的”重来一次”

CNAME 表示”这个名的规范名是另一个名”。解析器遇到 CNAME，必须**用新名字重新走一遍解析**，可能再次遇到 CNAME，形成链。链上任何一环失败，整个查询失败；链太长或出现环，会触发实现自己的上限与错误处理。

三个 practical 后果：

1. **CNAME 不能与同名的其他数据共存**（除少数 DNSSEC 相关记录外）。给 `www` 同时写 A 和 CNAME 是配置错误，不是”两条都生效”。
2. **CNAME 的目标可能在另一个区域、另一条委派链上**。所以一次”简单的 A 查询”可能触发两三轮独立的迭代。CDN 场景里尤其常见：站点域名 CNAME 到 CDN 提供的名字，而那个名字又在另一套权威体系下。
3. **查询名最小化的严格模式会咬人**。某些大型 CDN 的 CNAME 区违反 [RFC 8020]（父名返回 NXDOMAIN、子名却存在），`qname-minimisation-strict: yes` 会在中间层拿到 NXDOMAIN 就直接收工，不再回退完整查询名→整站 CSS/JS 全部解析失败。改回 `strict: no` 才恢复，**并且链路上每一层缓存都要刷**，否则前一层照样返回缓存的 NXDOMAIN。参数与兼容坑在第三篇第 13 章，这里只解释为什么”一条 CNAME”能引发整站故障。

CNAME 链也解释了为什么”解析全过程”不能只画一次直线：它是一条可能分叉、可能回头、可能在任意一层命中缓存的图。第 3 章把这张图按角色切成四段，就是为了定位”到底哪一段出了问题”。

2.5 主流实现横评

2.5.1 先定角色，再看产品

比较 DNS 软件时最常见的错误，是拿不同角色的产品互比”谁更快、谁更安全”。一台权威服务器和一台存根解析器没有可比性。先把五种角色列清楚：

角色	做什么	对谁提供递归
权威	只对自己被委派的区域给出权威应答	不提供
递归（验证）	自己从根开始迭代，缓存并验证	提供，且通常是最终信任边界
转发	把查询整体交给指定上游，自己不做完整迭代	视配置，可能只是代理
缓存	主要靠缓存应答，减少上游查询	视配置
存根	把查询交给别处，自己不迭代	不提供

一个产品可以覆盖多个角色（BIND 9 几乎全覆盖），也可以只做其中一个（NSD 只做权威）。**“全能”不是优点也不是缺点，它是取舍**：功能集中省机器，配置面与故障半径同时变大。

2.5.2 总对照表

下表中的”强 / 中 / 弱”是相对定位，用于建立选型直觉，不是基准测试结论；具体能力以各项目官方文档的当前版本为准。

实现	主要角色	DNSSEC 验证	加密传输	过滤能力	配置复杂度	资源占用	典型场景
BIND 9	权威 + 递归 + 转发 + 缓存	完整（并支持权威侧签名）	支持 DoT 等能力，见官方手册	view / RPZ 等，能力强	高	中–高	企业级权威与递归一体、复杂策略
dnsmasq	转发 + 缓存 + DHCP	可作为验证器但通常弱/可选	依赖上游，自身不主攻	基于域名与地址的基础过滤	低	极低	家庭网关、嵌入式、小型网络
CoreDNS	权威（插件）+ 转发 + 服务发现	依赖插件	依赖插件	依赖插件	中（插件链）	低–中	Kubernetes 集群内服务发现
NSD	权威	权威侧签名，不做递归验证	不涉及递归侧	不涉及	中	低	只做权威、追求精简
PowerDNS Recursor	递归 + 缓存（+ 转发）	支持	支持，见官方手册	脚本与策略较强	中	中	运营商/大型递归、需要脚本化策略
PowerDNS Authoritative	权威	权威侧	不涉及递归侧	不涉及	中	中	多后端权威、数据库驱动
Knot Resolver	递归 + 缓存 + 转发	支持	支持	模块化策略	中–高	低–中	模块化、偏好 Lua 策略的递归
systemd-resolved	存根 + 缓存 + 转发	有限，依系统策略	支持部分加密上游	弱	低（但与其他解析器共存易冲突）	极低	桌面/主机本地存根
Unbound	递归（验证）+ 缓存 + 转发 + 可作库	默认验证，信任锚就位时真验证	下游 DoT/DoH/DoQ、上游加密	本地策略 + RPZ	中	低	自建递归验证、按角色拆实例

读表时请注意两点：一是”支持”不等于”默认开启”，例如 DNSSEC 验证在有无信任锚时行为完全不同（第 1 章的三

种状态）；二是”加密传输”列只说明能力覆盖，**DNSSEC = 数据完整性/来源认证，DoT/DoH/DoQ = 传输加密/隐私**，两者不能互相替代，也不能用其中一项的勾选去推断另一项已生效。

2.5.3　按维度展开的取舍

配置复杂度。 BIND 9 的配置文件有 view、ACL、zone、options、日志通道等大量结构，一项策略常常要同时改几处；dnsmasq 一个文件加几个命令行参数就能跑，但要把”哪个客户端用哪套过滤表”表达清楚就吃力；Unbound 介于两者之间，配置以 `server:` / `forward-zone:` / `local-zone:` / 模块声明为主，边界清楚。CoreDNS 用插件链表达，灵活但插件顺序本身就是语义。

DNSSEC 验证。需要区分”能做验证”和”默认真的在验证”。BIND 9、Unbound、Knot Resolver、PowerDNS Recursor 都具备完整验证能力；dnsmasq 与 systemd-resolved 在这条链上更依赖上游或系统策略，因而在本书的威胁模型里通常不被当作最终验证边界。NSD 与 PowerDNS Authoritative 属于权威侧，不参与递归验证。第三篇第 10 章会把信任锚、放行与拒绝展开成可验证的状态。

加密传输。这里也要分两段看：客户端到解析器的一段（下游 DoT/DoH/DoQ），以及解析器到上游的一段（加密上游）。浏览器可能自己启用 DoH 而绕过本地解析器，这是第五篇讨论 DDR 压制时的核心问题；而一台本地递归解析器向权威服务器发出的迭代查询，通常仍是明文 53。把”我开了 DoH”当成”整条链路都加密了”是典型误判。

过滤能力。 dnsmasq 的过滤面向”家用够用”：按域名屏蔽、按地址改写，表达力有限且可观测性弱；BIND 的 RPZ 与 view 能承载复杂策略；Unbound 有 `local-zone` / `local-data` / `respip` 与 RPZ（官方手册标注 1.10.0 初版、1.14.0 补齐完整触发器与动作）。过滤越强，越要明确它作用于哪个信任域，否则”误拦”与”绕过”会同时出现。

可观测性与资源占用。 dnsmasq 与 systemd-resolved 占用极低，但日志与统计粒度也有限，排查”为什么这个域 SERVFAIL”时常常要换工具；BIND 与 Unbound 都有统计与查询日志能力，代价是多占一点内存与运维注意力。需要全量查询日志时，还要明确保留期——dnstap 是自己的查询流副本，无期限保留会在自家门口把”抗元数据聚合”破功。

2.6　重点对照：BIND 与 dnsmasq

用户最容易在这两者之间做错选择，因为它们常被放在同一台”网关”上讨论，但实际面向的场景完全不同。

2.6.1　BIND 9：全能的代价

强在哪。权威与递归一体、view 能按来源给出不同视图、ACL 与日志通道细、RPZ 与 DNSSEC 工具链完整、文档与社区积累最厚。需要一台服务器同时承担对外权威、对内递归、按来源分流、并做复杂策略时，BIND 9 依然是成熟可靠的选择。

代价在哪。配置面大意味着**误配概率高**：一个 view 的匹配顺序、一条 `allow-recursion` 的范围、一次 `recursion yes` 与转发器的组合，都可能把递归服务暴露给不该暴露的来源。历史 CVE 数量多，既有部署广、被审视充分的原因，也有权威 + 递归 + 动态更新 + 区域传送共处一个进程、代码路径相互牵连的原因（第 1 章讨论过这条）。对只想要”帮我查域名并验证签名”的场景，这些额外路径都是不需要承担的面。

典型失败模式。不是”跑不起来”，而是”跑起来了但语义不对”：递归被意外开放、view 命中了预期外的客户端、转发与递归混用导致验证链断裂。这类问题不会在启动时报错，只能靠从客户端侧实际发查询去验证。

2.6.2　dnsmasq：小型网络的一体化方案

强在哪。极低的资源占用、DHCP + DNS + TFTP 一体、配置短、在路由器固件与嵌入式设备里几乎无处不在。对”一台家庭网关要给局域网发地址、顺便把域名查询转给上游”这个需求，它是最省事的答案。

代价在哪。它面向的是**转发与缓存**，不是自主迭代与验证。DNSSEC 验证与加密传输能力有限且常依赖上游，过滤表达力有限，日志与统计粒度弱；当出现”某个域解析失败”，很难从 dnsmasq 自身判断是上游失败、自己缓存了 NXDOMAIN、还是过滤规则命中。可观测性弱在平时不是问题，在排查时是主要瓶颈。

典型失败模式。上游切换或过滤表更新后，客户端仍在拿到旧结果；因为缓存与转发逻辑混在一起，定位要跨工具。另一个常见冲突是与本机其他存根解析器（如 systemd-resolved）同时占用 53 端口，表现为”配置没改但行为变了”。

2.6.3　三者对照：Unbound 的位置

维度	BIND 9	dnsmasq	Unbound
定位	权威 + 递归全能	小型网络 DHCP/DNS 一体	只做递归验证，模块化
默认安全	依赖配置收敛暴露面	依赖上游，验证与可观测性弱	默认随机化、清理器、DNSSEC 验证
配置面	大	极小	中，边界清楚
可观测性	强	弱	统计 + 日志 + dnstap，可按需
适合	复杂企业/权威场景	嵌入式、家用网关	自建递归、按角色拆实例
不适合	只想要一台最小验证器	要自主迭代、要可审计验证链	要做权威服务器、要做集群服务发现

这不是产品宣传，是角色取舍。三句话概括：

- 需要对外权威、view、复杂策略→ BIND 9（或 NSD / PowerDNS Authoritative 专职做权威）；
- 需要 DHCP 一体、资源极省、验证交给上游→ dnsmasq；
- 需要自己从根开始迭代、默认验证、边界可审计、按信任域拆实例→ Unbound。

三者也可以共存：dnsmasq 继续管 DHCP，把 DNS 查询转给本机 Unbound；Unbound 自己做迭代与验证。**不要用一个产品覆盖所有角色，再靠一堆开关互相覆盖语义**。

2.7 本章要点

- DNS 命名空间是倒树，区域是管理单元，委派靠父区域的 NS 记录；胶水记录（glue record）解决委派自举，但它是附加段的非权威数据，域外夹带 glue 是经典缓存投毒面，与 DNSSEC 无关。
- 递归是解析器对客户端的承诺（RD/RA），迭代是解析器对上级的工作方式；一次 `www.example.org` 解析要经过根 → TLD →各级权威的多轮 referral，`dig +trace` 只能看委派链，不能代表本地解析器的真实行为。
- 缓存作用于 RRset，返回的是剩余 TTL；负缓存（negative caching，[RFC 2308]）会记住 NXDOMAIN/NODATA 并放大故障，过期缓存应答（serve-stale，[RFC 8767]）则是刷不到新数据时的有条件例外。
- EDNS(0) 用 OPT 伪记录扩展 UDP 报文，`edns-buffer-size`（出向）与 `max-udp-size`（入向）默认值均为 1232，方向不可调反；CNAME 链会让一次查询重走解析，且与查询名最小化的严格模式存在兼容坑。
- BIND 9 全能但配置面大、历史 CVE 多；dnsmasq 面向小型网络/嵌入式、DHCP+DNS 一体但验证与可观测性弱；Unbound 只做递归验证、默认安全、模块化。三者是角色取舍，不是优劣排名。
- DNSSEC = 数据完整性/来源认证，DoT/DoH/DoQ = 传输加密/隐私，二者不能互相替代。

2.8 参考资料

- RFC Editor, [RFC 1034] *Domain Names –Concepts and Facilities*: https://www.rfc-editor.org/rfc/rfc1034.txt
- RFC Editor, [RFC 1035] *Domain Names –Implementation and Specification*（wire format 与报文头定义）: https://www.rfc-editor.org/rfc/rfc1035.txt
- RFC Editor, [RFC 9460] *Service Binding and Parameter Specification via the DNS (SVCB and HTTPS Resource Records)*: https://www.rfc-editor.org/rfc/rfc9460
- RFC Editor, [RFC 2308] *Negative Caching of DNS Queries (DNS NCACHE)*: https://www.rfc-editor.org/rfc/rfc2308.txt
- RFC Editor, [RFC 6891] *Extension Mechanisms for DNS (EDNS(0))*: https://www.rfc-editor.org/rfc/rfc6891.txt
- RFC Editor, [RFC 8767] *Serving Stale Data to Improve DNS Resiliency*: https://www.rfc-editor.org/rfc/rfc8767.txt
- NLnet Labs, Unbound 官方手册·`unbound.conf(5)` 配置参考: https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html
- ISC, BIND 9 项目页: https://www.isc.org/bind/
- Simon Kelley, dnsmasq 文档: https://thekelleys.org.uk/dnsmasq/doc.html

第三章

存根、缓存、转发、递归：四种角色

“DNS 服务器”这个说法太粗，掩盖了链路上的分工。一次查询可能依次经过存根解析器（stub resolver）、缓存解析器（caching resolver）、转发器（forwarder）、递归解析器（recursive resolver），每一段对报文做了什么、谁在做迭代、谁在缓存、谁在验证，都不一样。故障排查时若把这四段混成一段，最典型的后果是：在错误的那一层刷缓存、在错误的那一层找 AD 位、或在错误的那一层期待 DNSSEC 验证发生。

本章把四种角色严格拆开，给出报文位的语义、一条串联的数据流图（文字描述）、以及”一台机器上跑多个实例”的动机。

3.1 四种角色的职责边界

3.1.1 存根解析器（stub resolver）

存根解析器是应用程序与 DNS 之间的第一跳，通常不是一个独立进程，而是操作系统解析库（如 glibc 的 resolver）或一个本地轻量守护进程（如 systemd-resolved）。它的职责只有三条：

1. 按本机配置（`/etc/resolv.conf`、NetworkManager、systemd 的链路配置等）选出一个上游解析器；
2. 组装查询报文、置位 **RD**，发出去；
3. 等待最终应答，把结果交给应用。

它**不做迭代**：不会自己去问根服务器；**不承担验证**：即使应答里带了 AD 位，多数存根也只把它当普通数据处理；很多存根**不做缓存**，或只做极短的进程内缓存。它信任的是”被配置指向的那台服务器”。

这个”信任”是全书安全模型的起点。如果应用自行启用加密传输（DoT / DoH / DoQ）而绕过本机存根，本机上配置的所有过滤、验证与日志策略都不会作用于这条查询——第五篇讨论 DDR 压制与浏览器 canary 域时，处理的正是这个绕过问题。

3.1.2 缓存解析器（caching resolver）

缓存解析器的核心职责是**用缓存换取延迟与上游负载**：命中即答，未命中时要么自己解析、要么向别处取，然后把结果按 TTL 存下来。

“缓存”是一个**能力**，不是一个排他的角色：递归解析器必然带缓存，转发器通常也带缓存，存根解析器可能带一小块缓存。因此本节的”缓存解析器”指的是**以缓存为主要职责、不承担完整迭代与验证的那一层**，例如节点级缓存或专用的 cache 实例。

两个反直觉现象，按角色看：

- 上游已经全断，缓存层仍可能正常应答一段时间，于是”客户端没报错”不等于”上游健康”；
- 上游瞬时 NXDOMAIN 会被按 SOA minimum **负缓存（negative caching）**，权威侧修好后这一层仍继续失败，且链路上每一层缓存都要各自刷。

3.1.3 转发器（forwarder）

转发器把收到的查询**整体交给指定的上游解析器**，自己不从根开始迭代。它像一个有明确出口的代理：客户端问什么，它就把什么转给上游（或按策略改写），拿到应答后返回给客户端，通常也会缓存。

转发器的关键性质有三条：

- **不自主迭代**：一切依赖上游。上游的验证状态、过滤策略、日志口径会原样传导到下游；
- **与验证器二选一**：DNSSEC 信任锚（trust anchor）需要自己向根 priming DNSKEY 才能建立与更新；若把 `forward-zone "."` 把一切都交给上游，这一步无法完成，日志会持续出现 `failed to prime trust anchor`，并间歇性返回全量 SERVFAIL。转发器上不要配自动信任锚；
- **单上游不是冗余**：给转发器配第二个 `forward-addr` 时，Unbound 的行为是”并发竞速取最快”，而不是主备。于是查询会被送到不该去的解析器，过滤表失效。转发器只留一个上游，切换交给外部巡检脚本。

第三、四两条合起来还有一条容易漏掉的推论：**上游超时后若允许回退到明文 53，整条查询流会静默降级**。要 `forward-first: no`、fail-closed，两波都失败就返回 SERVFAIL，而不是”为了可用性偷偷走明文”。加密传输（DoT/DoH/DoQ）保护的是链路隐私，DNSSEC 保证的是数据来源与完整性，两者不互相替代；静默降级恰好把前者抹掉。

3.1.4　递归（验证）解析器（recursive / validating resolver）

递归解析器对客户端承诺”我给你最终答案”（RD/RA），对上级则自己从根开始做**迭代**：根→ TLD →各级权威服务器，遇到 referral 就换一台再问，遇到 CNAME 就用新名字重走一遍，直到拿到应答或明确的不存在证明。

“验证”是它区别于普通递归解析器的地方：信任锚就位时，按 DNSSEC 签名链检查应答，坏签名返回 SERVFAIL，好签名在应答里置 **AD** 位。第 1 章列出的三种状态在这里仍然适用：

状态	行为	运维可见现象
无信任锚	透传不验证	坏签名域也返回答案，看起来一切正常
permissive 模式	记录日志但放行	日志里有验证失败，客户端仍拿到应答
真验证	坏签名→ SERVFAIL，好签名带 AD	坏签名域整体失败，好签名域可凭 AD 确认

信任锚（trust anchor）与 priming：信任锚是验证器手里那份「我预先相信的公钥」——通常是根区的 KSK（Key Signing Key），以 DS / DNSKEY 形式写进 `auto-trust-anchor-file` 指定的文件。**priming**（引导查询）指验证器启动时按 [RFC 5011] **主动向根权威服务器发一次 DNSKEY 查询**，把根公钥取回来、核对并写入该文件，之后按滚动周期自动维护。它是「自己长出信任链」的第一步，不是读一份静态配置就完事：这也解释了为什么转发器（一切交给上游）无法完成 priming——它根本没有机会与根直接对话。第 5、10 章会分别给出反面配置与可观测的报错，第 16 章讲 [RFC 5011] 的滚动逻辑。

严格说，“递归”和”验证”是两个可分离的能力，但本书语境里它们由同一台机器承担，这也是 Unbound 的原始定位：“只做递归验证、小而可审计”（第 1 章）。**只有自己掌握迭代与验证，才谈得上把整条链路的策略握在手里**：一台只做转发的机器，其安全上限就是上游的安全上限。

3.1.5　一张职责对照表

维度	存根解析器	缓存解析器	转发器	递归（验证）解析器
是否做迭代	否	否	否	**是**，从根开始
是否缓存	通常否/极短	**是**，主要职责	通常有	是
是否验证 DNSSEC	否，信任上游	否	否，继承上游	**是**，信任锚就位时
对客户端置 RA	不置（它是客户端侧）	置	置	置
对上级发 RD	置（发给上游）	视取数方式	置（发给指定上游）	**不置**（迭代查询）
是否置 AD	不置	转发上游的 AD 或自己不置	转发上游的 AD	验证通过才置
故障半径	本机应用	一层缓存的旧数据	完全依赖上游	迭代链 + 自己的缓存
信任边界	被配置指向的服务器	上游数据	上游解析器	**自己**，信任锚

表里最容易被看错的两格是”对上级发 RD”与”是否置 AD”。前者把”递归”和”迭代”的区别落到报文上；后者决定了**AD 位只能由真正做了验证的那一层产生**，下游各层最多是把它透传下去。

3.2 报文里的四个标志位：RD / RA / AD / CD

四个位都在 DNS 报文头部，各自回答一个不同的问题。把它们按”谁设置、谁消费”来记，比按字母顺序记牢得多。

3.2.1 RD：Recursion Desired

谁设置：客户端（通常是存根解析器）在**请求**里设置。**含义**：“我希望你替我把这次解析做完，不要只给我一个 referral。”**谁消费**：收到请求的服务器。若它提供递归且策略允许，就自己跑完整个流程；若它是权威服务器或不提供递归，就忽略 RD，只给出自己知道的最好答案（通常是 referral）。

一个常见误判：把 RD=1 当成”这台服务器一定会递归”。RD 只是**请求**，服务器有没有能力、愿不愿意，由 RA 和访问控制共同决定。

3.2.2 RA：Recursion Available

谁设置：服务器在**应答**里设置。**含义**：“我提供递归服务。”它宣告的是**能力**，不是”这次我真的递归了”——命中缓存直接返回时，RA 同样会置位。

RA 与 RD 的组合可以快速判断一段链路的性质：

请求 RD	应答 RA	典型含义
1	1	客户端把任务交给了递归/转发/缓存层
0	1	对端有递归能力，但本次没被请求使用（如探测、迭代查询到达递归口）
0	0	权威服务器或关闭递归的服务器，给的是 referral 或权威应答
1	0	请求了递归但被拒绝（无权限、未开启递归），需查访问控制

最后一行在加固排查里最有用：客户端 RD=1 却拿到 RA=0，通常不是”DNS 坏了”，而是**访问控制把递归关掉或排除在外**了。第三篇第 9 章的最小暴露面会围绕这一行展开。

3.2.3 AD：Authenticated Data

谁设置：**真正做了 DNSSEC 验证**且验证通过的那台服务器，在应答里设置。**含义**：“这份数据我按签名链验过了。”它是解析器对客户端的声明，不是数据自带的属性。

两条必须记住的边界：

- **AD 可以透传**。转发器与缓存层可能把上游给的 AD 位原样交给客户端，于是”客户端看到 AD=1”并不证明**紧邻它的那一层**做了验证。要确认验证发生在哪一层，得看那一层自己的统计与日志，不能只看位。

如果 AD 没透传，或者标志位被篡改，会发生什么：分三种情况。

情况	客户端看到	实际后果
中间层把 AD 清 0（未透传）	AD=0	数据本身没变，安全验证过的应答不会因此变成未验证；只是**下游丢了「已验证」这个信号**。依赖 AD 才敢用数据的验证型存根会退回自己验签或视为不可信，属于「少了一个提示」，不是漏洞。
中间层把 AD 置 1（伪造/误置）	AD=1	这是危险的：客户端以为有人验过，其实可能没有。若客户端自己不做验证，就会把一份未签名/坏签名的应答当可信数据用，DNSSEC 的保护在这一跳被架空。

情况	客户端看到	实际后果
链路上有人改标志位（on-path 篡改）	视改法	UDP/53 明文下改一个 bit 很容易；这正是 DoT / DoH / DoQ 要保护的「链路完整性」。但加密只保证**没被路上改**，不保证**端点真验过**——端点自己伪造 AD，加密也照样送到。

AD 只能由真正做了验证的那一层产生，下游最多透传。判断依据是「那一层自己的信任锚、统计与日志」，不是报文里的一个 bit。

3.2.4　CD: Checking Disabled

谁设置：客户端在**请求**里设置。**含义**："请不要因为验证失败而拒绝这份数据，把原始数据给我。"它常用于诊断：当应用怀疑是验证器把某个域判成 SERVFAIL 时，置 CD 绕过验证，看权威侧到底返回了什么。

CD 是**请求方的意愿**，不是服务器的开关。实现可以忽略它、也可以按策略只放行给受信任的来源；把它开放给所有客户端，等于给外部提供了一条绕过验证的通道，是明显的降级面。与 CD 相邻的还有 **DO（DNSSEC OK）**标志（在 EDNS OPT 里），表示"请把 DNSSEC 相关记录一起给我"，它同样不等于"请替我验证"。

3.2.5　四位速查与一句话总结

位	方向	设置者	一句话
RD	请求	客户端	请你替我递归
RA	应答	服务器	我提供递归能力
AD	应答	验证通过的解析器	这份数据我验过了
CD	请求	客户端	别因验证失败就拒绝

一句话总结：**RD/RA 描述"谁跑腿"，AD/CD 描述"验证这件事做没做、要不要做"**。前一组是角色问题，后一组是信任问题；把两组混起来，就会得出"开了加密传输所以 AD 一定可信"这类错误结论。

顺带补两个不属于这四位、但常一起出现的标志：**AA**（Authoritative Answer）由权威服务器设置，表示"这是权威数据"，与 RA 互斥于典型场景；**TC**（TrunCated）表示 UDP 应答被截断，客户端应改用 TCP 重试——第 2 章讲的 EDNS 缓冲与分片问题，最终就会以 TC 或超时重试的形态表现出来。

3.3　一条串联的数据流图（文字描述）

```
[应用 / 浏览器]
      |
      | ① 查询 www.example.org. A?  RD=1
      v
[存根解析器 stub]  本机 127.0.0.1:53（或 systemd-resolved 监听口）
      |  不做迭代、不验证；只按本机配置选上游
      | ② 同一查询 RD=1，转发给 192.0.2.1:53
      v
[缓存解析器 caching]  192.0.2.1:53
      |  命中 → 直接返回（TTL 递减、可能带上游传来的 AD）
      |  未命中 → ③ 继续向上，RD=1
      v
[转发器 forwarder]  127.0.0.1:5357（单上游、fail-closed）
      |  不自主迭代；把查询整体交给唯一上游
      |  ④ 加密上游（DoT/DoH），RD=1
      v
[递归（验证）解析器 recursive/validating]  2001:db8::53
      |  对客户端：RD 进来，RA +（验证通过时）AD 出去
      |  对上级：⑤⑥⑦ 迭代查询，RD=0，不置 AD
      |
```

```
      |  ⑤ → 根服务器：referral 到 org
      |  ⑥ → org 权威服务器：referral 到 example.org
      |  ⑦ → example.org 权威服务器：最终应答或 CNAME
      |
      |  验证：按 DNSKEY/DS/RRSIG 链检查；坏签名 → SERVFAIL
      v
[应答回流]  ⑦ → 转发器 → 缓存解析器 → 存根 → 应用
              每层都可能缓存，每层都可能改写 TTL
```

把这条图读成五条判据，排查时就能快速定位”该看哪一层”：

1. **①与②之间**：如果应用自己启用加密传输（DoT / DoH / DoQ），这段根本不存在，本机所有策略被绕过。
2. **②与③之间**：缓存层命中就返回，所以”客户端正常”不证明上游正常。
3. **④这一段**：单上游、fail-closed、不允许静默回退明文；第二篇第 5 章全部约束都落在这里。
4. **⑤⑥⑦这一段**：只有递归解析器发的是 **RD=0 的迭代查询**，也只有它会去 priming 信任锚、做验证。转发器上配自动信任锚之所以失败，就是因为这一段不存在。
5. **回流路径**：每一层都可能缓存，包括负缓存。修好权威后要**逐层刷**，只刷一层不够。

3.3.1 图里没有画出来的东西

四点常被忽略、但会直接改变上面判据的适用性：

- **应用侧缓存**。浏览器与运行时通常有自己的 DNS 缓存，TTL 未必遵守 DNS 的 TTL；于是”刷了解析器缓存仍不生效”时，还要考虑应用进程。
- **CNAME 链**。遇到 CNAME 要用新名字重走解析，链上可能跨到另一套权威体系；图里的⑤⑥⑦只是**一轮**，不是全部。
- **多实例**。同一台机器上多个实例时，图的某些段会重复出现，且不同段的过滤表与上游不同（多实例的动机见后文）。
- **分层加密**。加密传输只覆盖某一段链路（如④），而递归解析器向权威服务器发出的⑤⑥⑦通常仍是明文 53。“我对客户端开了 DoH”不等于”整条链路都加密了”。

3.3.2 用四个位读这条链

把 3.2 节的表套到图上，会得到一组很实用的诊断结论：

观察点	期望	异常时说明什么
客户端发出 RD	1	0 说明客户端被当成迭代方，或查询来自另一个解析器
缓存层应答 RA	1	0 说明递归能力被关，或访问控制拒绝
递归解析器向根发 RD	0	1 说明它其实在转发，不是自己迭代
最终应答 AD	验证通过时 1	0 不一定错：无信任锚、permissive 模式、或该域未签名都会是 0
诊断请求 CD	仅受信任来源 1	任意客户端都能置 CD 且被放行，说明绕过验证的通道开着

第四行尤其要小心：**AD=0 不是”验证器坏了”的证据**。未签名域、无信任锚、permissive 模式都会得到 AD=0；判断验证是否真的在跑，必须结合信任锚状态与该层的验证统计，不能只盯一个位。

3.4 一台机器上跑多个实例的动机

第二篇会按角色分别配置缓存解析器、转发器、递归解析器与本地根区镜像。这里先回答一个前置问题：**为什么不把四种角色塞进一个实例，用几个开关区分？**

3.4.1 不同上游

一台网关上可能同时存在几类出口需求：内网客户端走”过滤 + 加密上游”的链路，监控探针需要”不过滤、可复现”的直连递归，某些内部域名要交给内网权威服务器解析。这些需求的**上层解析器不同**。

若在一个实例里混配，会撞上第 1 章与本篇反复提到的那条实测结论：给转发器配第二个 `forward-addr`，Unbound 是**并发竞速取最快**而非主备，查询会被送到不该去的解析器，过滤表失效。把不同上游拆到不同实例、每个转发器只留一个上游、切换交给外部巡检脚本，语义才确定。

3.4.2　不同过滤表

不同信任域的客户端需要不同策略：儿童设备与办公终端的屏蔽列表不同，访客网络不应享受内网的分流规则，运维探针则应当看到未经改写的原始应答。RPZ、`local-zone` / `local-data`、`respip` 这些策略一旦混在一个实例里，就要靠”按来源匹配”的复杂规则互相覆盖，误拦与绕过会同时出现，且很难说清某次应答到底命中了哪条策略。

拆成实例后，**每个实例的过滤表就是它的身份**：`192.0.2.1:53` 是过滤实例，`127.0.0.1:5357` 是无过滤的转发实例，`2001:db8::53` 是自主递归验证实例。策略边界与进程边界重合，排查时只需确定”客户端问的是哪个实例”。

3.4.3　不同信任域

信任域的差异不只是过滤表，还包括**验证状态与信任锚**：

- 转发器不应配自动信任锚（3.1.3 节）；
- 递归验证器需要自己 priming 与更新信任锚，坏签名直接 SERVFAIL；
- 某些内部域名故意不做验证（内部权威未签名），只应存在于特定实例上。

把”验证”和”转发”放进同一个实例，等于让 `forward-zone "."` 与信任锚互相打架，日志持续报 `failed to prime trust anchor`，间歇性全量 SERVFAIL。转发器与验证器二**选**一，是实例拆分最硬的一条理由。

3.4.4　可观测性与故障隔离

多实例还有两个工程上的收益：

- **统计口径干净**。缓存命中率、验证失败数、上游超时数可以按实例分别读，不会把”过滤实例的屏蔽”混进”递归实例的验证失败”。
- **故障半径可控**。刷新缓存、改过滤表、重启某个实例，只影响对应信任域。反过来，若所有角色挤在一个实例里，一次刷缓存就是一次全量影响。

代价是**多个监听口、多份缓存、多份日志、多个回滚点**。`dnstap` 全量记录每个实例的查询流时，必须明确保留期，否则”抗元数据聚合”会在自己门口破功。

3.4.5　拆分的边界条件

不是所有场景都需要拆。判断依据可以简化成三个问题：

1. 是否存在**两个以上互不信任的客户端集合**?
2. 是否存在**两种以上不同的上游或出口策略**?
3. 是否需要**明确区分”谁验证了、谁只是转发”**?

三个问题里有两个答”是”，拆实例就比堆开关更可维护；三个都答”否”，一台递归验证器加一个本地存根通常就够了。这条判断也是第二篇各章的组织逻辑：第 4 章最小缓存解析器、第 5 章单上游加密转发器、第 6 章自主递归与本地根区镜像、第 7 章把它们串起来并做可观测。

3.5　为第二篇铺路：读配置前先问四个问题

拿到任何一段 DNS 配置（无论 Unbound、BIND 还是 dnsmasq），先问四个问题，再谈参数：

1. **这一段在图上属于哪一层?** 存根、缓存、转发、递归?
2. **它向谁发 RD，向谁回 RA?** 这决定了它到底是”跑腿的”还是”被跑腿的”。
3. **这一层的 AD 是自己产生的，还是透传上游的?** 这决定了验证边界画在哪。
4. **这一层的缓存会存什么、存多久?** 包括正应答、负缓存、过期缓存应答与 CNAME 中间结果。

四个问题答完，第二篇各章的每个参数都能对上位置：`forward-first: no` 落在第 4 段链路的 fail-closed 上，信任锚落在递归验证器上，`serve-expired-client-timeout` 落在缓存层，`access-control` 落在”RD=1 / RA=0”那一行的判断上。

反过来，如果跳过这四问直接抄配置，最常出现的三种失败是：在转发器上配信任锚、给转发器加第二个上游以为在做冗余、以及以为客户端开了加密传输就等于整条解析链都被验证。**这三种都不是参数写错，而是角色认错。**

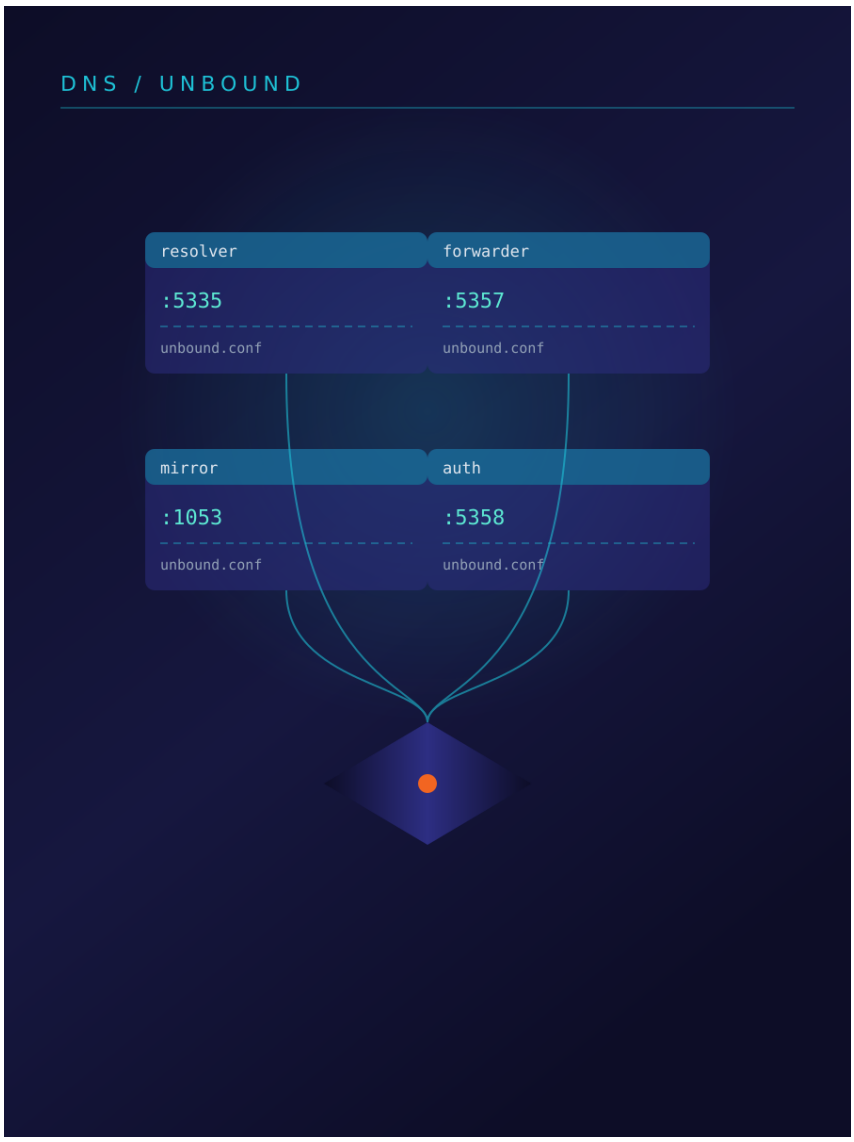
3.6 本章要点

- 存根解析器只选上游、置 RD、等最终应答，不做迭代与验证；缓存解析器以缓存为主要职责；转发器把查询整体交给唯一上游、不自主迭代；递归（验证）解析器自己从根迭代并在信任锚就位时做 DNSSEC 验证。
- RD 是客户端的”请你替我递归”，RA 是服务器的”我提供递归能力”；AD 由真正验证通过的解析器设置、下游只能透传；CD 是客户端”别因验证失败就拒绝”的意愿，开放给任意来源就是降级面。
- 递归是对客户端的承诺、迭代是对上级的工作方式：只有递归解析器发出 RD=0 的迭代查询，也只有它会 priming 信任锚；转发器上配自动信任锚会持续 `failed to prime trust anchor`。
- 数据流图里每一层都可能缓存，负缓存会放大故障，修好权威后必须逐层刷；加密传输只覆盖某一段链路，不等于整条链都被验证或加密。
- 一台机器上跑多个实例的动机是不同上游、不同过滤表、不同信任域，以及统计口径干净与故障隔离；代价是多份缓存、多份日志与明确保留期。
- 读任何 DNS 配置先问四件事：属于哪一层、向谁发 RD / 回 RA、AD 自产还是透传、缓存存什么存多久。第二篇第 4–7 章按此展开。

3.7 参考资料

- RFC Editor, [RFC 1034] *Domain Names –Concepts and Facilities*: https://www.rfc-editor.org/rfc/rfc1034.txt
- RFC Editor, [RFC 1035] *Domain Names –Implementation and Specification*: https://www.rfc-editor.org/rfc/rfc1035.txt
- RFC Editor, [RFC 4033] *DNS Security Introduction and Requirements*: https://www.rfc-editor.org/rfc/rfc4033.txt
- RFC Editor, [RFC 4035] *Protocol Modifications for the DNS Security Extensions*（AD / CD 位语义）: https://www.rfc-editor.org/rfc/rfc4035.txt
- RFC Editor, [RFC 6840] *Clarifications and Implementation Notes for DNSSEC*: https://www.rfc-editor.org/rfc/rfc6840.txt
- NLnet Labs, Unbound 官方手册·`unbound.conf(5)` 配置参考: https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html
- NLnet Labs, Unbound 官方手册·入门配置: https://unbound.docs.nlnetlabs.nl/en/latest/getting-started/configuration.html
- RFC Editor, [RFC 2308] *Negative Caching of DNS Queries (DNS NCACHE)*: https://www.rfc-editor.org/rfc/rfc2308.txt
- systemd, `systemd-resolved.service` 手册: https://www.freedesktop.org/software/systemd/man/systemd-resolved.service.html

第二篇　按角色配置 Unbound



第四章

缓存解析器：最小可用配置

一台家庭网关 + 一台境外云主机的实验环境里，第一件要跑起来的东西不是加密传输，也不是过滤表，而是一台能被 `dig` 问通、能被 `unbound-control stats` 看见的缓存解析器（caching resolver）。

这一章只处理”缓存解析器”这一个角色：它自己做递归、自己维护缓存、对下游表现为一台可用的递归解析器。转发器（forwarder）在第 5 章，本地根区镜像在第 6 章，DNSSEC 验证在第三篇第 10 章，QNAME 最小化（QNAME minimisation，QMIN）的参数在第三篇第 13 章，都不在这里展开。

4.1 十行起步

先给一台什么都还没配的机器写出最小可用配置。下面这十行是完整文件，不是片段（注释行不计入十行）：

```
# /etc/unbound/unbound.conf.d/10-minimal.conf
server:
  interface: 127.0.0.1
  interface: ::1
  access-control: 127.0.0.0/8 allow
  access-control: ::1/128 allow
  do-ip4: yes
  do-ip6: yes
  do-udp: yes
  do-tcp: yes
  verbosity: 1
```

为什么这么设：`interface` 决定监听哪些地址，官方手册说明”若一条都不给，默认只监听 localhost”；这里显式写出来，是为了后面把它换成 LAN 地址时不会误以为默认就是全网监听。`access-control` 的默认动作是 `refuse`（无匹配时拒绝），且按最长前缀匹配、与语句顺序无关，所以必须显式放行回环，否则连本机 `dig` 都拿不到答案。`do-ip4` / `do-ip6` 同时管”是否应答”与”是否向互联网发出”两个方向；`do-ip6: no` 关掉的是 IPv6 传输，不影响 DNS 报文里照常携带 AAAA 记录内容。`do-udp` / `do-tcp` 都留 `yes`：UDP 是常态，TCP 是 EDNS 报文过大、区域传送与重试时的退路，关掉 TCP 会在大应答场景里直接失败。

怎么验证：

```
unbound-checkconf /etc/unbound/unbound.conf.d/10-minimal.conf && echo OK
```

失败模式 / 回滚点：`unbound-checkconf` 只做语法与路径检查，不会告诉你 `access-control` 是否放行、端口是否被占用。若它报错，改回上一版文件即可，配置未被加载，不影响正在运行的实例。若它通过了但 `dig` 拿到 `REFUSED`，问题在 `access-control` 而不是语法，先把源文件地址 `dig -b` 指定成 `127.0.0.1` 再测。

启动与第一次问答：

```
systemctl enable --now unbound
dig @127.0.0.1 example.org A +short
```

为什么这么设：先用回环地址验证，是为了把”Unbound 本身是否工作”和”防火墙 / 接口地址是否正确”两件事分开。任何一步失败都先回到这一条最小命令，不要在十几个参数里乱猜。

怎么验证：dig 输出里 status: NOERROR 且 flags: 带 ra（Recursion Available），说明这台实例确实以递归解析器身份应答。若 flags 里没有 ra，说明问到的不是本实例。

失败模式 / 回滚点：systemctl start 失败时看 journalctl -u unbound -n 50，最常见是 53 端口被 systemd-resolved 或其他实例占住。最小配置没有动 port，回滚就是 systemctl stop unbound，无状态可丢失（缓存为空）。

4.2　把监听面从回环扩到 LAN

```
# /etc/unbound/unbound.conf.d/20-listen.conf
server:
  interface: 192.0.2.1
  interface: 2001:db8::1
  interface: 127.0.0.1
  interface: ::1
  access-control: 192.0.2.0/24 allow
  access-control: 2001:db8::/64 allow
  access-control: 127.0.0.0/8 allow
  access-control: ::1/128 allow
  do-ip4: yes
  do-ip6: yes
  do-udp: yes
  do-tcp: yes
```

为什么这么设：interface 可以重复给多条；@port 后缀（如 192.0.2.1@5353）用于同一主机多实例，第 7 章专门讲。注意官方手册明确写：**interface 的变更在 kill -HUP reload 时不生效，只有 restart 才生效**。所以本章凡是改 interface，都必须走 systemctl restart，reload 会留下”配置已改但不监听”的假象。access-control 只写允许网段、依赖默认 refuse 兜底，比”先 allow 全网再 deny 几条”安全；动作还有 deny（静默丢包）、refuse（回 REFUSED）、allow_setrd、allow_snoop、allow_cookie、deny_non_local、refuse_non_local，最小配置只用 allow。

do-ip4 / do-ip6 在这里保持 yes 是刻意选择：一台双栈网关若关掉 do-ip6，Unbound 就不会向互联网的 IPv6 权威服务器发查询，只剩 IPv4 出向；反过来，如果链路上 IPv6 只有 ULA、没有可用出向，就该关掉 do-ip6 而保留 AAAA 内容的解析能力，避免每根查询都先等一次 IPv6 超时。这两件事要分开判断，不要因为”客户端没有 IPv6”就顺手关掉 AAAA。

怎么验证：

```
ss -lunp | awk '$5 ~ /:53$/ {print}'
dig @192.0.2.1 example.org A +short
dig @2001:db8::1 example.org AAAA +short
```

失败模式 / 回滚点：ss 里看不到新地址，先怀疑是 reload 没生效（上面那条限制），再怀疑地址尚未配到网卡上。回滚：删掉新增 interface 行并 systemctl restart unbound，回环仍可用。若 dig 从 LAN 侧拿到 REFUSED，用 dig -b 192.0.2.1 指定源地址复核 access-control 的最长前缀匹配。

4.3　降权、chroot 与信息隐藏

```
# /etc/unbound/unbound.conf.d/30-hardening.conf
server:
  username: unbound
  chroot: ""
  hide-identity: yes
  hide-version: yes
  identity: ""
  version: ""
```

为什么这么设：username 的默认值就是 unbound，语义是”绑定端口之后降权”；显式写出来是为了让审计的人一眼看到降权意图，也为了配合第 6 章按 uid 做 nftables 豁免——那里需要知道进程到底跑在哪个 uid 下。username: "" 表示不降权，只用于排障，不要留在生产配置里。

chroot 需要谨慎：一旦启用，配置文件里所有相对路径都在 chroot 内解释，include 相对路径、pidfile、logfile、密钥文件、根区 zonefile 都要重新对齐；AppArmor / SELinux 的路径白名单也要跟着改。这里给 chroot: "" 是明确选

择”暂不启用”；写了 chroot 却没把 tls-cert-bundle、auto-trust-anchor-file 等文件的路径搬进去，是启动即失败的经典原因。

hide-identity: yes 让 id.server / hostname.bind 这类 CHAOS 类查询回 REFUSED，hide-version: yes 对 version.server / version.bind 同样处理。两者默认都是 no，identity: "" 时会返回主机名，version: "" 时会返回真实版本串。这是减少暴露面，不是访问控制；真正挡住外部查询的是 access-control 与 interface。

怎么验证：

```
dig @127.0.0.1 version.server TXT CHAOS +short
dig @127.0.0.1 id.server TXT CHAOS +short
ps -o user=,pid=,comm= -C unbound
```

失败模式 / 回滚点：前两条应为空或 REFUSED；若返回了版本串，说明 hide-version 没被加载（检查 include 顺序、是否 reload 了错的实例）。ps 应显示 unbound 用户；若仍是 root，检查是否 username: "" 被后面的 include 覆盖。回滚：把 chroot 恢复为空串并 restart，降权与隐藏两项可以独立保留。

4.4 EDNS 缓冲区：1232，不要 1472

```
# /etc/unbound/unbound.conf.d/40-edns.conf
server:
  edns-buffer-size: 1232
  max-udp-size: 1232
```

为什么这么设：这两个参数方向不同，最容易记反。

参数	管的方向	官方默认值	作用
edns-buffer-size	出向（发给对等方的 UDP 报文里通告的 EDNS 重组缓冲区）	1232	决定上级权威服务器最多能回多大的 UDP 应答
max-udp-size	入向（本实例回给下游客户端的最大 UDP 应答）	1232	限制自己的应答大小，不适用 TCP

1232 是 DNS Flag Day 2020 的推荐值，Unbound 官方手册把它同时作为两个选项的默认值。它的来历是 IPv6 最小 MTU 1280：1280 减去 40 字节 IPv6 首部、8 字节 UDP 首部，得到 1232。于是在 MTU 1280 的链路上（WireGuard、部分 IPv6 隧道）一个 1232 字节的 DNS 应答恰好是不分片的单包。若改成 1472，在 MTU 1280 上必然分片；在 PPPoE 的 1492 MTU 上同样会踩到分片边界（1472 是按以太网 1500 减 28 算出来的，PPPoE 少了 8 字节）。IP 分片在不少家宽 / 云链路上被丢弃，结果不是立刻报错，而是”UDP 等超时→重试 TCP”，表现为偶发的解析慢、DNSSEC 验证超时、大应答域名随机 SERVFAIL。

关键区分：max-udp-size 只管入向。上游被你的大缓冲区撑爆的场景，要靠 edns-buffer-size 收敛，光调 max-udp-size 没用。另外官方手册提醒 edns-buffer-size 不要设得高于 msg-buffer-size（该值默认 65552），设成 512 虽然能绕过最严苛的 PMTU 问题，但会制造过量 TCP 回退，属于过度矫正。

PMTU 是什么：PMTU 全称 **Path MTU**（路径最大传输单元），指一条端到端路径上**所有链路里最小的那个 MTU**——不是本机网卡的 MTU，而是整条路（含隧道、PPPoE、运营商封装）的瓶颈值。IPv6 规定最小 MTU 为 1280，隧道再叠加一层封装只会更小；PPPoE 常见 1492。**PMTU 问题**就是「发出的包大于路径能承载的大小，又没被正确发现」：IPv4 可能由路由器分片，IPv6 则由源端分片或依赖 PMTUD（Path MTU Discovery）；一旦中间设备丢弃分片或 ICMPv6 Packet Too Big 被防火墙挡掉，表现为**大包静默消失、等超时再重试**。DNS 里这正是 1472 出事的机理：应答 UDP 包超过 PMTU →分片被丢→客户端等超时→重试 TCP。EDNS(0)（**Extension Mechanisms for DNS (Version 0)**，[RFC 6891]）的 1232 就是按「IPv6 最小 MTU 1280 减 40 字节 IPv6 头、8 字节 UDP 头」算出的保守值。

怎么验证：

```
dig @127.0.0.1 example.org DNSKEY +dnssec +bufsize=1232 | grep -E 'status|EDNS|UDPsize'
dig @127.0.0.1 +bufsize=4096 example.org A +short
```

第一条看 EDNS: version: 0, flags:; udp: 1232 一类输出；第二条验证客户端即便请求 4096，本实例也不会回一个超过 max-udp-size 的 UDP 包。更直接的链路级证据是 tcpdump -ni any 'udp port 53 and (ip[6:2] & 0x1fff) != 0' 抓分片，正常应无输出。

失败模式 / 回滚点：把 edns-buffer-size 改成 1472 后若出现”部分域名偶发 2 秒延迟后才有答案”，先怀疑分片被丢，而不是怀疑上游。回滚就是改回 1232 并 reload（这两个参数不需要改 interface，reload 生效）。注意 dig +bufsize= 是测试工具侧的覆盖，不要拿它当配置值来源。

4.5　线程与缓存尺寸

```
# /etc/unbound/unbound.conf.d/50-cache.conf
server:
  num-threads: 2
  msg-cache-size: 64m
  rrset-cache-size: 128m
  prefetch: yes
  prefetch-key: yes
  so-rcvbuf: 4m
```

为什么这么设：

num-threads 默认 1（无线程），官方性能调优页建议设成 CPU 核数。家庭网关 / 小机房通常 2–4 核足够；盲目设成 32 只会让 msg-cache-slabs 等自动按 2 的幂调整、锁争用与内存开销一起上升。配合 so-reuseport: yes 在支持的内核上能改善多核 UDP 分发，用 unbound-control status 可看它是否真的激活。

msg-cache-size 与 rrset-cache-size 默认都是 4m。官方性能页给出的经验值是 **rrset 约为 msg 的两倍**（示例为 msg-cache-size: 50m / rrset-cache-size: 100m），因为 rrset 缓存存放 RRset 等底层对象、被多条消息共享。这里取 64m / 128m 遵循同一比例，适合一台兼做过滤与网关的机器；官方同时提醒 malloc 开销会让实际内存占用达到配置值的 2 到 2.5 倍，所以别把两项之和直接当成内存预算。缓存尺寸不是越大越好：它决定命中率上限，也决定重启后要重新预热多久。

prefetch: yes（默认 no）让消息缓存中 TTL 剩余不足 10% 的条目在命中时触发预取，热门条目不会过期；代价是约 10% 的额外流量与负载。prefetch-key: yes（默认 no）在验证过程中遇到 DS 记录就提前取 DNSKEY，降低首包延迟、多花一点 CPU。这两项都属于”用一点出向流量换稳定延迟”，在没有可用上游的链路上别开。

so-rcvbuf 默认 0（用系统值），官方建议忙服务器试 4m：它设置 53 端口入向 UDP 的 SO_RCVBUF，用来吸收短时突发、避免丢包。Linux 上会受 net.core.rmem_max 限制，Unbound 需要 root 权限才能绕过；另一个方向 so-sndbuf 默认 4m，管出向应答的发送缓冲。丢包是否真的发生，看 netstat -su 的接收错误计数，不要凭猜测调这个数。

怎么验证：

```
unbound-control stats_noreset | grep -E 'total.num.cachehits|total.num.cachemiss|num.prefetch|mem.cache'
nproc
sysctl net.core.rmem_max
```

stats_noreset 与 stats 的区别是”只看不清零”，做基线测量时用它，避免自己把计数读没了（第 7 章展开）。若 so-rcvbuf: 4m 未生效，sysctl net.core.rmem_max 的值就是线索。

失败模式 / 回滚点：num-threads 与缓存尺寸都需要 restart 才能完全生效；reload 后看到数字没变不要急着改第二遍。缓存开太大导致内存吃紧时，优先降 msg-cache-size，保持 1:2 比例回滚。配置里若把缓存设成 0，prefetch / prefetch-key 都失去意义，官方手册对此有明确说明。

4.6　serve-expired 全家：先刷新，超时才回过期缓存

下文把”用过期缓存应答”叫**过期缓存应答（serve-stale）**，配置项按原样写 serve-expired*。这一组参数在 Unbound 里有两种模式，理解它们靠 serve-expired-client-timeout：

```
# /etc/unbound/unbound.conf.d/60-serve-expired.conf
server:
  serve-expired: yes
  serve-expired-ttl: 86400
  serve-expired-ttl-reset: yes
  serve-expired-reply-ttl: 30
  serve-expired-client-timeout: 1800
```

为什么这么设：

- serve-expired: yes（默认 no）是总开关，允许用超过 TTL 的旧记录应答。
- serve-expired-ttl: 86400（单位**秒**，默认 86400）限制记录过期后还能被服务多久，0 表示不设上限。官方手册援引 [RFC 8767] 的建议区间是 86400（1 天）到 259200（3 天）。
- serve-expired-ttl-reset: yes（默认 no）在刷新失败后把过期记录的 TTL 重置为 serve-expired-ttl，让”一直有人问”的记录能持续被应答。
- serve-expired-reply-ttl: 30（单位**秒**，默认 30）是回给客户端的 TTL，且被原始 TTL 封顶；官方手册在启用 client-timeout 时明确建议取 30，依据 [RFC 8767]。给 30 秒的含义是”客户端可以短暂复用，但别长期缓存这个降级答案”。
- **serve-expired-client-timeout: 1800 单位是毫秒**，即 **1.8 秒**，不是 1800 秒，也不是 1.8 分钟。语义是：先正常尝试解析，超过 1800 毫秒还没结果，才暂停解析、回过期缓存，之后再继续后台刷新。这是 [RFC 8767] 模式的开关（值大于 0 即启用），默认 1800，Unbound 1.23.0 起 RFC 模式成为默认行为。
- 把它设成 0 会切到 Unbound 早期的 serve-expired 模式：命中过期记录就**立即**回给客户端，同时后台继续刷新。命中率更高，但客户端更容易拿到较旧的数据；官方性能页建议这种模式下务必配 prefetch: yes。

本书选 1800（RFC 模式）：把过期缓存当”上游故障时的最后一道兜底”，而不是平时的加速手段。还要记住它与负缓存（negative caching）的相互作用：上游一次瞬时 NXDOMAIN 会被按 SOA minimum 缓存（常见 3600 秒），这不是 serve-expired 能救的；修好权威后客户端仍失败，需要刷缓存（第 7 章）。

怎么验证：

```
unbound-control stats_noreset | grep -E 'num.expired|num.queries_timed_out|num.recursivereplies'
dig @127.0.0.1 example.org A +short
```

num.expired 是”用过期缓存应答了多少次”的计数。做故障演练时，把上游链路临时断开，再问一个刚过 TTL 的名字，应看到 num.expired 增长且 dig 在约 1.8 秒后返回答案（带 30 秒 TTL）；恢复链路后该计数应停止增长。

失败模式 / 回滚点：把 1800 误读成秒、写成 1800 以为给了半小时缓冲，实际是 1.8 秒；反过来若写成 1800000 想表达 1.8 秒也会让客户端等半小时。回滚：serve-expired: no 并 reload 即可关闭整组行为，缓存本身不受影响。另一个坑是”上游慢但没全断”时，1.8 秒到点就回旧数据，业务上若要求强一致性，应改用 serve-expired-client-timeout: 0 之外的方式——直接关闭 serve-expired，不要靠调大超时硬扛。

4.7 完整闭环：checkconf →启动→ dig → stats

把前面各段合起来，形成一次可复现的上线闭环。下面这段脚本假设配置放在 unbound.conf.d/、由主配置 include 进去；每台机器上路径可能不同，按实际发行版调整。

```
#!/bin/sh
# 上线闭环：语法检查 → 启动 → 问答验证 → 统计基线
set -eu
CONF=/etc/unbound/unbound.conf

unbound-checkconf "$CONF" || { echo " 配置语法不通过，保持原实例不动" >&2; exit 1; }

systemctl restart unbound
systemctl is-active --quiet unbound

# 1) 基本递归能力
dig @127.0.0.1 example.org A +short >/dev/null
dig @127.0.0.1 example.org A | grep -q 'flags:.* ra' || { echo " 未以递归解析器身份应答" >&2; exit 1; }

# 2) TCP 退路可用
dig @127.0.0.1 example.org A +tcp +short >/dev/null

# 3) 信息隐藏生效
test -z "$(dig @127.0.0.1 version.server TXT CHAOS +short)" || echo " 注意：version 未被隐藏"

# 4) 统计基线（不清零）
unbound-control stats_noreset | grep -E 'total.num.queries|total.num.cachehits|total.num.cachemiss|num.expired'
```

为什么这么设：顺序不能颠倒。unbound-checkconf 在最前面，保证语法错误时根本不碰运行中的实例；restart 而不是 reload，因为 4.2 节改了 interface；第 2 步专门测 +tcp，因为很多 EDNS / 大应答故障只在 TCP 退路上暴露；第

4 步用 stats_noreset 而不是 stats，因为 stats 默认会把计数清零（受 statistics-cumulative 控制），会在”验证”这一步把刚建立的基线抹掉。

要启用 unbound-control，主配置里需要 remote-control: 段的 control-enable: yes 与由 unbound-control-setup 生成的自签证书；不启用时这一步会失败，但前 3 步仍然有效。开启与不开的取舍、以及”不开时刷缓存只能重启实例”的代价在第 7 章展开。

怎么验证：

```
unbound-control status
unbound-control stats | grep -E 'total.num.cachehits|total.num.cachemiss'
```

status 应返回 alias is up；stats 里 cachehits 随重复查询增长、cachemiss 只在首次解析 / 缓存过期时增长，说明缓存真的在工作。第一次跑完再连问三次同一个名字，命中数的增量应约等于 3。

失败模式 / 回滚点：unbound-control 连不上时先查 control-interface（默认只听回环）、证书文件权限与 username 降权后能否读取密钥，不要首先怀疑网络。整段脚本的回滚点是”保留上一版 unbound.conf.d/ 目录的副本”：所有变更都是纯配置文件，systemctl restart unbound 会重建一个空缓存，不会损坏数据。真正的状态只有缓存，丢弃它是可接受的；如果链路上还有第二层缓存实例，别忘了那一层也要单独确认（第 7 章）。

4.8　从”能跑”到”可交付”的核对表

项	期望	一条检查命令
语法	无错误	unbound-checkconf
监听	回环 + 指定 LAN 地址	ss -lunp 'sport = :53'
访问控制	允许网段 NOERROR，其余 REFUSED	dig -b 192.0.2.1 @192.0.2.1 example.org A
降权	进程 uid 为 unbound	ps -o user= -C unbound
信息隐藏	CHAOS 查询无版本串	dig @127.0.0.1 version.server TXT CHAOS +short
EDNS	出向 / 入向均 1232	dig @127.0.0.1 +bufsize=4096 example.org A
TCP 退路	+tcp 有答案	dig @127.0.0.1 example.org A +tcp +short
缓存	重复查询命中数增长	unbound-control stats_noreset | grep cachehits
过期兜底	断链时 num.expired 增长	unbound-control stats_noreset | grep expired

这张表不是”最佳实践清单”，而是把本章每个配置块后面的验证命令收拢成一次交付前的自查。任何一行不过，回到对应小节看”失败模式 / 回滚点”，不要同时改多个参数。

一个容易忽略的边界：unbound-checkconf 检查的是单文件语法与文件路径，不检查 access-control 语义、不检查 nftables 是否把 53 劫持到了别处、也不检查上游是否可达。第二篇后续章节会依次补上这三类问题——第 5 章解决”上游是谁”，第 6 章解决”根那一跳从哪来”，第 7 章解决”多实例下日志与缓存怎么看”。

4.9　本章要点

- 最小可用配置只需 server: 段的 interface、access-control、do-ip4 / do-ip6、do-udp / do-tcp；access-control 无匹配时默认 refuse，interface 改动只在 restart 生效、reload 不生效。
- edns-buffer-size 管出向通告、max-udp-size 管入向应答，两者默认均为 1232（DNS Flag Day 2020）；1472 在 MTU 1280 与 PPPoE 1492 上分片，分片被丢会导致等超时重试 TCP。
- msg-cache-size 与 rrset-cache-size 默认各 4m，建议按 1:2 放大并计入 2–2.5 倍 malloc 开销；num-threads 取核数，prefetch 用约 10% 额外流量换热门条目不过期，so-rcvbuf: 4m 吸收突发丢包。
- serve-expired-client-timeout: 1800 单位是**毫秒**（1.8 秒），语义是”先刷新、超时才回过期缓存”；设 0 才切成”命中即回旧数据”的早期模式，serve-expired-ttl / reply-ttl 是秒。

- 完整闭环为 `unbound-checkconf` → `systemctl restart` → `dig`（含 `+tcp` 与 CHAOS 检查）→ `unbound-control stats_noreset`；统计基线用 `stats_noreset`，避免 `stats` 把计数清零。
- `username: unbound` 在绑定端口后降权；`chroot` 一旦启用，所有相对路径与 AppArmor 白名单一并要重排，未准备好就明确写 `chroot: ""`。

4.10 参考资料

- Unbound 官方手册，`unbound.conf(5)` 配置项与默认值：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html
- Unbound 官方手册，`unbound-checkconf(8)`：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound-checkconf.html
- Unbound 官方手册,`unbound-control(8)` 命令与统计计数器:https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound-control.html
- Unbound 官方手册，Serving Stale Data（`serve-expired` 与 RFC 8767 两种模式）：https://unbound.docs.nlnetlabs.nl/en/latest/topics/core/serve-stale.html
- Unbound 官方手册,Performance Tuning(线程、缓存比例、`so-reuseport`):https://unbound.docs.nlnetlabs.nl/en/latest/topics/core/performance.html
- DNS Flag Day 2020，EDNS 缓冲区尺寸建议：https://dnsflagday.net/2020/
- RFC 8767, *Serving Stale Data to Improve DNS Resiliency*: https://www.rfc-editor.org/rfc/rfc8767

第五章

转发器：单上游、加密上游、fail-closed

缓存解析器（第 4 章）自己走完整递归；转发器（forwarder）不递归，它把查询整个交给另一台递归解析器，自己只做缓存、过滤与访问控制。这个角色在家宽网关、Kubernetes 节点级缓存、以及”出口只有一条加密通道”的场景里非常常见，也是最容易配错的一种角色。

5.1 最小转发器：唯一上游

```
# /etc/unbound/unbound.conf.d/10-forward.conf
server:
  interface: 127.0.0.1@5357
  interface: 192.0.2.1@5357
  access-control: 127.0.0.0/8 allow
  access-control: 192.0.2.0/24 allow
  do-ip4: yes
  do-ip6: no
  do-udp: yes
  do-tcp: yes
  username: unbound
  chroot: ""
  hide-identity: yes
  hide-version: yes
  edns-buffer-size: 1232
  max-udp-size: 1232

forward-zone:
  name: "."
  forward-addr: 192.0.2.53
  forward-first: no
```

为什么这么设：`forward-zone` 的 `name: "."` 表示”所有查询都交给这个区”，官方手册明确写：`name: "."` 加一条 `forward-addr` 会把所有查询（除可从缓存直接回答的）转发给那台服务器。文档同时强调，列表里的 `forward-host` / `forward-addr` 指向的必须是**能继续递归的递归解析器**，不是权威服务器；Unbound 在这个区里不做递归，只逐个追 CNAME 链。

`forward-first: no`（默认 `no`）是 fail-closed 的第一道闸：它表示转发查询遇到 SERVFAIL 时**不要**回退到”较不具体的区”或普通递归解析。官方手册对 `yes` 的警告是”任何解析失败都会允许逃出这个区”——一旦逃出去，查询就会自己向互联网递归，绕开既定上游，也绕开上游那侧的过滤与审计策略。

`do-ip6: no` 是因为这台实验网关没有可用的 IPv6 出向；按第 4 章的说明，它只关 IPv6 传输，不影响应答里照常返回 AAAA 记录内容。端口 `@5357` 是给第 7 章多实例规划预留的语义，不与真实主机名或公网 IP 同段出现。

怎么验证：

```
unbound-checkconf /etc/unbound/unbound.conf
dig @127.0.0.1 -p 5357 example.org A +short
unbound-control -c /etc/unbound/unbound.conf list_forwards
```

`list_forwards` 应只列出一条 `forward-addr`，这是”单上游”最可靠的核对方式——比翻配置文件可靠，因为 include 拼接后的结果才是生效值。

失败模式 / 回滚点：若 `dig` 返回 SERVFAIL 而不是超时，先查上游是否真的会递归（拿 `dig @192.0.2.53 example.org A` 直连测一次）。回滚：删掉 `forward-zone` 段并 restart，实例退回第 4 章的普通缓存解析器，缓存会清空但无持久状态损坏。

5.2　多个 forward-addr 是并发竞速，不是主备

这是本章最要紧的一条实测结论，也是”唯一上游”要求的由来。

```
# 反面示例：不要这样写
forward-zone:
  name: "."
  forward-addr: 192.0.2.53
  forward-addr: 198.51.100.53
  forward-first: no
```

为什么不能这么设：很多人把第二条 `forward-addr` 理解成”备胎”，以为第一条挂了才用第二条。Unbound 不是这个语义。多条 `forward-addr` 构成的是一组候选上级，解析器会在它们之间做服务器选择 / 并发探测，谁先答、谁答得快就用谁——也就是**并发竞速取最快**，不是主备切换。官方手册对 `auth-zone` 的 `primary` 用的是”都试一遍”（tried if one fails），对 `forward-addr` 则没有”主 / 备”或优先级的表述；实际行为是流量会分摊到所有候选。

后果不是”多一份冗余”，而是**过滤表失效**：假设 `192.0.2.53` 是你精心配置、带过滤与审计的上游，`198.51.100.53` 只是顺手加的公共解析器，那么一部分查询会稳定地走后者。客户端看到的过滤结果变成随机——同一个域名，这次被挡、下次放行。更糟的是这种故障很难归因：日志里两条上游都没报错，`forward-first: no` 也确实拦住了递归回退，问题只是”查询根本没送到该去的地方”。

还有一层比过滤表更容易被忽略的代价：**并发竞速意味着同一个 QNAME 会同时送给多家上游**。即便链路都加了密、每家单看都合规，「谁看得见这份查询」也从一家变成了多家，且每家都拿到一份独立的完整查询名与时序。上游数量越多，这份元数据的副本越多；用「波次」也一样——第一波两家并行就已经把查询扩散出去了，第二波再补两家只是把范围进一步扩大。这是**上游隐私与往返延迟之间的直接交换**：并行换低延迟，代价是信任面变宽；串行换单一家可见，代价是失败时的等待。本书的取向是后者——只留一条 `forward-addr`，把可用性与切换交给外部巡检脚本。

静默明文降级是同一类问题的另一面：若允许上游超时后退回明文 53 或退回自身递归，整条查询流就在无人察觉的情况下裸奔。所以这里的策略是：**转发器只留一条 `forward-addr`，上游的可用性与切换交给外部巡检脚本**（5.6 节），fail-closed 由 `forward-first: no` 与 `forward-tls-upstream` 共同保证，两波都失败就明确回 SERVFAIL，让故障可见。

怎么验证：

```
unbound-control list_forwards | grep forward-addr | wc -l
tcpdump -ni any 'udp port 53 and host 198.51.100.53' -c 5
```

第一条应为 1；第二条在持续查询时不应抓到任何发往”备用”地址的报文。若抓到了，说明某处 include 又追加了一条上游。

失败模式 / 回滚点：已经上了双上游的生产实例，修复就是删掉多余行并 restart（缓存清空是预期代价）。不要试图用”给第二条设个更差的超时”来模拟主备——语义不支持。`forward-first: yes` 不是补救，它会打开另一条逃生通道。

5.3　TCP 承载的上游：forward-tcp-upstream

```
# /etc/unbound/unbound.conf.d/20-forward-tcp.conf
forward-zone:
  name: "."
  forward-addr: 192.0.2.53
  forward-tcp-upstream: yes
  forward-first: no
```

为什么这么设：`forward-tcp-upstream: yes`（默认 `no`）让这个区的上游查询**只用 TCP**，不管全局 `tcp-upstream` 是什么。它的直接收益是避开 UDP 分片与放大限制：第 4 章讲过 1232 是出向 EDNS 通告值，但某些链路对 UDP 大包或跨境 UDP 有 QoS，TCP 承载更稳。实测形态是同一条拥挤链路上 UDP/QUIC 下载 1MB 只到约 300 B/s，而 TCP 承载的通道能到约 660 KB/s；DNS 查询虽然小，超时重试的行为完全同源。

代价是每条上游查询都要建连（Unbound 会复用连接，但仍有握手与状态开销），延迟通常高于 UDP。所以它是”链路质量差时的选择”，不是默认值。注意它只管**上游方向**；下游客户端仍可用 UDP 问本机，由 `do-udp` 决定。

怎么验证：

```
dig @127.0.0.1 -p 5357 example.org A +short
tcpdump -ni any 'tcp port 53 and host 192.0.2.53' -c 5
```

应只看到 TCP/53 出向报文，没有 UDP/53 到该上游。

失败模式 / 回滚点：上游若只监听 UDP（少见但存在），会全量 SERVFAIL。回滚：改为 `no` 或删掉该行并 reload（`forward-zone` 选项可 reload 生效，不必改 `interface`）。若 `unbound-control` 未开启，改完记得 `systemctl restart` 才保险。

5.4 加密上游：forward-tls-upstream 与 tls-cert-bundle

`forward-tcp-upstream` 解决”用 TCP 承载”，但不加密。**加密传输（DoT / DoH / DoQ）与 DNSSEC 是两件事**：前者提供传输加密 / 隐私，后者提供数据完整性 / 来源认证，任何配置都不得用一个替代另一个。本节的 DoT 只是上游方向的客户端用法；服务端侧 DoT（853 端口监听）在第三篇第 11 章，DoH 在第 12 章，协议族对比在第四篇第 17 章。

```
# /etc/unbound/unbound.conf.d/30-forward-dot.conf
server:
  tls-cert-bundle: /etc/ssl/certs/ca-certificates.crt
  edns-buffer-size: 1232
  max-udp-size: 1232

forward-zone:
  name: "."
  forward-addr: 192.0.2.53@853#dns.example.org
  forward-tls-upstream: yes
  forward-first: no
```

为什么这么设：`forward-tls-upstream: yes`（默认 `no`，旧别名 `forward-ssl-upstream`）让上游查询走 TLS。官方手册的硬性要求：**启用它就必须配 `tls-cert-bundle`（或 `tls-win-cert`），否则连接无法被认证**。`tls-cert-bundle` 指向 CA 证书包，官方手册说明它在启动阶段、降权与 chroot 之前读取——所以第 4 章的 `chroot` 与 `username` 组合不会让它读不到，但若改成自己写的 chroot 路径，仍要确认这个文件在启动时的可见性。

`forward-addr` 里的写法有三段，顺序固定：`IP@ 端口 # 认证名`。`@853` 指定非默认端口，`#dns.example.org` 指定 TLS 证书要匹配的名字。官方手册明确：只有 `@` 时用默认 53 端口，只有 `#` 时用 `tls-port` 配置的端口；`@` 必须在 `#` 前面。**如果省略 `#` 和认证名，任何名字都被接受**——这等于把证书校验退化成”有证书就行”，中间人拿一张任意可信 CA 签的证书即可顶替上游。所以认证名不是可选项。

`forward-first: no` 在这里是加密链路的 fail-closed 关键：TLS 握手失败或上游 SERVFAIL 时，绝不退回明文 53 或自行递归，直接 SERVFAIL。少了这一行，上游一抖动，查询流就静默明文降级，而客户端完全无感。

怎么验证：

```
unbound-checkconf /etc/unbound/unbound.conf
dig @127.0.0.1 -p 5357 example.org A +short
tcpdump -ni any 'tcp port 853 and host 192.0.2.53' -c 5
openssl s_client -connect 192.0.2.53:853 -servername dns.example.org -CAfile /etc/ssl/certs/ca-certificates.crt </dev/null 2>&1 |
```

`openssl s_client` 的 `Verify return code: 0 (ok)` 证明 CA 包与认证名配置一致；`tcpdump` 应看到 853/TCP 而不是 53/UDP。高 verbosity 下 Unbound 会把对端 TLS 证书打进日志，可用于核对指纹。

失败模式 / 回滚点：`tls-cert-bundle` 路径写错或 `chroot` 后不可见→启动失败或全量 SERVFAIL；证书名不匹配→握手失败，日志出现 TLS 认证错误。回滚：保留 `forward-tls-upstream: yes` 但先不要把实例切到生产，用 `forward-addr: 192.0.2.53@53` 的明文配置做对照实验，验证完再切回。任何情况下都不要靠删掉 `forward-first: no` 来”让它通”。

5.5　do-not-query-localhost: no 的适用条件

```
# /etc/unbound/unbound.conf.d/40-local-upstream.conf
server:
  do-not-query-localhost: no

forward-zone:
  name: "."
  forward-addr: 127.0.0.1@5358
  forward-first: no
```

为什么这么设：`do-not-query-localhost` 默认 `yes`，作用是把 `::1` 与 `127.0.0.1/8` 加进 `do-not-query-address` 列表，即”绝不向本机回环地址发查询”。这是防环的默认保护：递归解析器若把根 / 上级指向自己，会形成自问自答的死循环。

只有在**确实要把本机另一个实例当上游**时才改成 `no`。典型场景就是第 7 章的多实例链：一个实例专做过滤 / 缓存（`@5357`），另一个实例专做加密出口或本地根区镜像（`@5358`），前者把 `127.0.0.1@5358` 作为唯一 `forward-addr`。此时回环就是设计中的下一跳，不是环路。

严格适用条件，三条都要满足：

1. 上游是**另一台实例 / 另一个端口**，且它自己有独立出口（递归、加密上游或本地根区），否则整条链没有终点。
2. 两个实例的端口、pidfile、控制接口、缓存目录互不重叠（第 7 章）。
3. 上游实例不会再把查询转回下游实例；`forward-first: no` 能保证失败时不逃逸，但不能替你检查拓扑成环。

`do-not-query-localhost: no` 放开的是回环，不是安全边界。若 `forward-addr` 写的是 `192.0.2.1` 这类本机非回环地址，改这个选项毫无作用，要用 `do-not-query-address` 相关配置去排查。

怎么验证：

```
dig @127.0.0.1 -p 5357 example.org A +short
ss -lunp | awk '$5 ~ /:5358$/ {print}'
```

两条都要成立：下游能答、且 5358 确实是另一个进程在听。

失败模式 / 回滚点：若只有下游实例在跑，全量 SERVFAIL 且日志有“no query to localhost”一类提示。回滚：恢复默认 `yes`，把 `forward-addr` 指向真正可达的非回环上游。这是本章最容易在”改了配置文件但没重启第二个实例”时踩到的坑。

5.6　转发器上不要配 auto-trust-anchor-file

```
# 反面示例：转发器上不要这样写
server:
  auto-trust-anchor-file: /var/lib/unbound/root.key
forward-zone:
  name: "."
  forward-addr: 192.0.2.53
  forward-first: no
```

为什么不能这么设：信任锚（trust anchor）不是一段静态配置，Unbound 需要**自己向根发起 priming、取回根区 DNSKEY** 并按 [RFC 5011] 做滚动维护。而 `forward-zone: name "."` 把包括根 DNSKEY 在内的所有查询都交给了上游转发器：本机解析器根本没有机会与根权威服务器直接对话，priming 永远无法完成。

实测形态是日志里持续出现 `failed to prime trust anchor`，并伴随**间歇性全量 SERVFAIL**。之所以”间歇性”而不是”全挂”，是因为部分答案能从缓存或上游直接给出，一旦某个查询路径需要验证器介入，就会撞上没建立好的信任链。这个故障特别容易被误判成上游不稳定，于是运维去调上游超时、加第二条 `forward-addr`——正好又踩中 5.2 节的竞速坑，问题越修越散。

结论是角色二选一：转发器与验证器（validator）不要在同一实例上混。要么做纯转发器（本章），信任锚与 DNSSEC 验证交给上游那台递归解析器；要么撤掉 `forward-zone`、让本机自己递归并配好信任锚（第三篇第 10 章）。第 6 章的本地根区镜像是第三种形态，但它仍需要本机有真实出向。

顺带区分三个概念：无信任锚 = 透传不验证（坏签名域也返回答案）；`val-permissive-mode: yes` = 记录日志但放行；`val-permissive-mode: no` = 真验证（坏签名回 SERVFAIL，好签名带 AD 标志）。这些是验证器的状态，不是转发器的选项。

怎么验证:

```
journalctl -u unbound --since '10 min ago' | grep -c 'failed to prime trust anchor'
unbound-checkconf /etc/unbound/unbound.conf && grep -rn 'auto-trust-anchor-file' /etc/unbound/unbound.conf.d/
```

第一条应为 0 且不再增长；第二条应无输出（或确认该文件属于另一个实例的配置）。

失败模式 / 回滚点：删掉 auto-trust-anchor-file 行并 restart，日志里的 priming 报错应立即停止。若业务要求本机出 AD 标志，正确做法不是把它加回来，而是改成递归解析器角色或换用本地根区镜像。

5.7 巡检脚本生成配置的幂等写法

上游切换交给外部脚本，就必须保证脚本**幂等**。下面这个形态是本章最值得抄走的一段：

```
#!/bin/sh
# /usr/local/sbin/unbound-upstream-sync.sh
# 由 cron / systemd timer 每分钟调用；配置没变就不 install、不 restart
set -eu

STAGING=$(mktemp /run/unbound-upstream.XXXXXX)
DEST=/etc/unbound/unbound.conf.d/10-forward.conf
trap 'rm -f "$STAGING"' EXIT

# 1) 探测唯一上游，失败就保持现状（fail-closed，不自动降级）
UPSTREAM=192.0.2.53
if ! dig @"$UPSTREAM" example.org A +time=1 +tries=1 +short >/dev/null 2>&1; then
    logger -t unbound-sync 'upstream probe failed, keep current config'
    exit 0
fi

# 2) 先写临时文件，再比较
cat >"$STAGING" <<EOF
# generated by unbound-upstream-sync.sh; teaching addresses only
forward-zone:
  name: "."
  forward-addr: ${UPSTREAM}
  forward-tls-upstream: no
  forward-first: no
EOF

if cmp -s "$STAGING" "$DEST"; then
    exit 0                      # 内容一致：不 install、不 restart
fi

unbound-checkconf "$STAGING" >/dev/null 2>&1 || {
    logger -t unbound-sync 'generated config invalid, keep current config'
    exit 1
}
install -m 0644 "$STAGING" "$DEST"
systemctl restart unbound
logger -t unbound-sync "upstream config installed: ${UPSTREAM}"
```

为什么这么设: cmp -s 是整段的轴心。巡检任务常被设成每分钟跑一次, 如果每次都无条件 install + systemctl restart, 那台转发器的缓存就会被**每分钟清空一次**：命中率归零、所有热门域名重新回源、延迟曲线周期性锯齿，而配置内容其实一个字都没变。cmp -s 在内容相同时直接退出，把”探测”与”变更”解耦。

其余几处也是必要的: 先写 mktemp 临时文件再 install, 避免脚本中途失败留下半截配置; unbound-checkconf 在 install 之前跑，保证语法错误时磁盘上的生效文件没被动过；探测失败时 exit 0 且只记日志，是刻意的 fail-closed——不因为一次抖动就把上游换成别处，也不退回明文；trap 清理临时文件。

注意这段脚本只演示单上游明文形态。切到 5.4 节的 DoT 上游时,把 forward-addr 换成 192.0.2.53@853#dns.example.org、forward-tls-upstream: yes 即可；**认证名与端口都参与 cmp -s 的比较**，所以切换上游会正常触发一次 restart，而”每分钟重复写同一段”不会。

怎么验证：

```
/usr/local/sbin/unbound-upstream-sync.sh
stat -c '%y %n' /etc/unbound/unbound.conf.d/10-forward.conf
sleep 65
stat -c '%y %n' /etc/unbound/unbound.conf.d/10-forward.conf
systemctl show unbound -p NRestarts
```

两次 `stat` 的 mtime 应完全相同，`NRestarts` 不增长；再手工改一个字符并跑脚本，应看到 mtime 更新且 restart 计数加一。这两条一起才算验证了幂等。

失败模式 / 回滚点：最常见的错是把 `unbound-checkconf "$STAGING"` 写成检查 `$DEST`——那样检查的是旧文件，新错误配置照样被 install。另一个错是 `restart` 写成 `reload` 且同时改了 `interface`（第 4 章：interface 变更只有 restart 生效）。回滚：脚本生成的只有一个 `10-forward.conf`，用备份覆盖并 restart 即可；`logger` 记录给出了每次 install 的时间点，便于定位。

5.8　转发器的验收清单

检查项	期望	命令 / 依据
上游数量	恰好 1 条	`unbound-control list_forwards`
回退	关闭	配置里 `forward-first: no`
传输	按设计只用 TCP / 853	`tcpdump` 抓对应端口，不应出现明文 53 出向
证书认证名	已指定 `#name`	`forward-addr` 里 `#` 后非空
CA 包	已配置且可读	`tls-cert-bundle` + `openssl s_client`
信任锚	转发器上不存在	`grep -rn auto-trust-anchor-file`
回环豁免	仅在本机多实例链上放开	`do-not-query-localhost: no`
幂等	配置不变则不重启	`cmp -s` + `stat` mtime + `NRestarts`
故障可见	上游断时明确 SERVFAIL	断链演练，不是静默降级

最后一行是这一章的收敛点：fail-closed 的价值不在于"永远不失败"，而在于**失败时看得见**。两波都失败就回 SERVFAIL，让监控和人有东西可查；静默明文降级、静默绕过过滤表、每分钟静默清缓存，才是真正难查的三类故障。

5.9　本章要点

- `forward-zone: name "."` 加唯一 `forward-addr` 才是单上游；多条 `forward-addr` 是**并发竞速取最快**而非主备，会把查询送到既定上游之外，令过滤表失效。
- `forward-first: no` 是 fail-closed 核心，禁止 SERVFAIL 时逃出该区自行递归；`forward-first: yes` 会打开静默明文降级的通道。
- `forward-tcp-upstream: yes` 让上游只用 TCP 承载（跨境 UDP 被 QoS 时更稳）；`forward-tls-upstream: yes` 提供传输加密，必须同时配 `tls-cert-bundle` 与 `# 认证名`，否则证书不被认证或任何名字都被接受。
- **加密传输（DoT / DoH / DoQ）与 DNSSEC 不互相替代**：前者是传输加密 / 隐私，后者是数据完整性 / 来源认证。
- 转发器上不要配 `auto-trust-anchor-file`：信任锚需自行向根 priming DNSKEY，全量转发时无法完成，日志持续 `failed to prime trust anchor` 并伴随间歇性全量 SERVFAIL；转发器与验证器二选一。
- `do-not-query-localhost: no` 只在本机另一实例作为设计中的下一跳时才用，且需保证端口、pidfile、控制接口互不重叠且拓扑不成环；巡检脚本用 `mktemp` + `cmp -s` + `unbound-checkconf` + `install` 保证幂等，配置没变就不 install、不 restart，否则每分钟都会清空缓存。

5.10　参考资料

- Unbound 官方手册，`unbound.conf(5)` 的 Forward Zone Options、Authority Zone Options 与 `tls-cert-bundle`：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html

- Unbound 官方手册,`unbound-control(8)` 的 `list_forwards` 与统计命令:https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound-control.html
- Unbound 官方手册，DNS-over-TLS / DNS-over-HTTPS 主题页：https://unbound.docs.nlnetlabs.nl/en/latest/topics/privacy/dns-over-https.html
- NLnet Labs，Unbound 项目页与文档入口：https://nlnetlabs.nl/projects/unbound/about/
- RFC 7858, *Specification for DNS over Transport Layer Security (TLS)*: https://www.rfc-editor.org/rfc/rfc7858
- RFC 5011, *Automated Updates of DNS Security (DNSSEC) Trust Anchors*: https://www.rfc-editor.org/rfc/rfc5011
- RFC 8767, *Serving Stale Data to Improve DNS Resiliency*（转发器故障时的兜底语义）: https://www.rfc-editor.org/rfc/rfc8767
- RFC 9230, *Oblivious DNS over HTTPS*（把查询内容与客户端 IP 拆到不同参与方）: https://www.rfc-editor.org/rfc/rfc9230

第六章

递归解析器与本地根区镜像

第 5 章的转发器把所有查询交给上游，本机不做递归。本章走相反方向：让 Unbound 自己做完整递归，并把根区（root zone）复制到本机，做成”本地根区镜像”（hyperlocal root）。这样做的收益是根这一跳不出本机——查询 `www.example.org` 时，根 NS 与胶水记录（glue record）直接从内存里读，不必向根权威服务器发 UDP 包；在跨境链路抖动、根服务器被 QoS、以及需要减少出向元数据的场景里很实用。

代价是这台实例必须有**真实可用的出向 53**：初次 AXFR 拉根区、后续 SOA 探测 / IXFR 更新都要用。这正是本章最容易翻车的地方，6.6 节会单独讲。

AXFR / IXFR 是什么：两者都是 DNS 的**区域传送**协议，定义在 [RFC 1035] 与 [RFC 1995]。**AXFR** 全称 **Authoritative (Full) Zone Transfer**（全量区域传送）：把整个区的所有记录一次性传过来，首次建镜像、或序列号差距太大时用。**IXFR** 全称 **Incremental Zone Transfer**（增量区域传送）：只传「自某个序列号以来改了哪些记录」，靠 SOA 序列号比对触发，日常更新靠它，流量远小于 AXFR。注意这是**权威服务器到权威服务器/递归器的传送协议**，不是普通递归查询；Unbound 自己不提供 AXFR / IXFR 服务端（第 25 章写过它的”非常有限的权威服务”），所以本章要请 NSD 出面。

实验环境仍是一台家庭网关 + 一台境外云主机。**根服务器地址一律写成示意地址**（例如 `198.51.100.53`），实际部署时应换成权威机构公开的根服务器地址。

6.1 auth-zone 的四个开关

`auth-zone` 是 Unbound 的”权威区”机制：把一份区数据装进本机，按两个不同阶段参与应答。配根区镜像就是配一个 `name: "."` 的 `auth-zone`。

```
# /etc/unbound/unbound.conf.d/60-auth-zone.conf
auth-zone:
  name: "."
  primary: 198.51.100.53
  fallback-enabled: yes
  for-downstream: no
  for-upstream: yes
  zonefile: /var/lib/unbound/root.zone
```

为什么这么设：四个开关各管一件事，组合起来才叫 hyperlocal。

选项	默认	语义
`name: "."`	必填	区名；可配多个 `auth-zone`，按”最接近查询名”的区生效
`primary`	无	用 AXFR / IXFR 从这里下载区副本；可给多条，一条失败就试下一条
`for-downstream`	`yes`	在 local-zone 之后、缓存之前处理，Unbound 对下游**像权威服务器一样**应答（CNAME 原样返回、不再解析）

选项	默认	语义
`for-upstream`	`yes`	在缓存查询之后、真正出网递归之前处理，递归过程从这里取本地副本
`fallback-enabled`	`no`	本区查询失败（含 DNSSEC 验证失败）时，退回互联网正常递归
`zonefile`	无	区数据落盘路径；不配则只用内存

官方手册对两个阶段的说明值得逐字理解：`for-downstream: yes` 时处于”local-zone 之后、缓存之前”，本实例对区内名字表现为权威服务器，CNAME 记录**原样**回给客户端；`for-upstream: yes` 时处于”缓存查询之后、出网抓取之前”，提供一份权威服务器的本地副本用来加速递归。若两者都开（默认），客户端查根区会得到权威式应答，而内部递归需要根区数据时读本地副本。

本章选 `for-downstream: no` + `for-upstream: yes`，正是官方手册点名的 hyperlocal 组合：客户端与内部查询**都走完整递归流程**，只是根那一跳从本地副本取，CNAME 会被彻底解析而不是原样丢回。好处是下游拿到的是正常递归答案，不会误以为这台机器是根的权威服务器；同时若配了 DNSSEC 验证，本机会先验证区内容再入缓存。

`fallback-enabled: yes` 是可用性兜底：区数据过期、验证失败或 AXFR 全挂时，退回正常的互联网递归而不是全量 SERVFAIL。官方手册明确，区数据到达 SOA expiry 后会过期，过期时查询返回 SERVFAIL，此时若 fallback 开启就改走上游递归。取舍是：开启后镜像失效时仍可解析（但失去”根不出本机”的收益），关闭后故障立刻可见。教学环境建议先开，观察稳定后再按策略收紧。

`primary` 可以写多条，也可以追加 `@ 端口`或 `# 认证名`。**认证名**是 `#` 后面那个字符串：启用 AXFR over TLS（XFR-over-TLS，[RFC 9103]）时，它指定**要拿去核对上游 TLS 证书的名字**——客户端会检查证书 SAN / CN 是否与这个名字一致，不是随便填的标签。不写 `#` 时按官方手册「不校验特定主体」，只靠 CA 信任链；这与第 11 章 `forward-addr: IP@853#认证名`里的 `#` 是同一套语义（`@` 在前、`#` 在后）。纯明文 AXFR over TCP/53 不需要 `#`。

官方手册还提醒**不要把 `primary` 写成需要靠这份区数据本身才能解析的主机名**，否则形成循环依赖，用纯 IP 最稳。还有一条容易踩：把 `primary` 指向另一个 Unbound 实例是无效的，因为 Unbound 不提供 AXFR / IXFR 服务；这也是 6.2 节要引入 NSD 的原因。

怎么验证：

```
unbound-checkconf /etc/unbound/unbound.conf
unbound-control list_auth_zones
dig @127.0.0.1 . NS +norecurse +short | head
```

失败模式 / 回滚点：`unbound-checkconf` 只验语法，不会验 AXFR 是否真能拉到数据。若 `dig . NS +norecurse` 返回的是空答案或 SERVFAIL，而日志里有区过期 / 传输失败信息，说明镜像没建立起来，`fallback-enabled: yes` 此时正在替你兜底。回滚：删掉整个 `auth-zone` 段并 restart，实例退回普通递归解析器（用内置根提示）。

6.2　用 NSD 做本地根镜像

6.2.1　NSD 是什么，为什么它是 Unbound 的兄弟项目

NSD（Name Server Daemon）是 NLnet Labs 的另一个开源 DNS 实现，只做**权威服务器**这一半：持有区文件、对区内名字给出权威应答（AA 位）、支持 AXFR / IXFR 区域传送与 TSIG，但**不做递归、不做 DNSSEC 验证、不缓存客户端查询**。它与 Unbound 出自同一个组织、共享一部分代码与工程习惯（BSD 许可、按角色收窄边界），正好互为镜像：

	NSD	Unbound
角色	权威服务器（authoritative）	递归验证器（recursive / validating）
应答带	AA（权威）	RA + 验证通过时 AD
区数据	自己持有 zonefile，可提供 AXFR	通过 `auth-zone` 拉一份副本，或 `local-zone` 少量静态数据
递归	不做	核心职责
动态更新 [RFC 2136]	**不支持**（第 25 章）	不支持

Unbound 的需求文档把「不做权威服务器」列为明确的非目标；本机根镜像需要一台能**对外提供 AXFR** 的权威服务器，Unbound 自己干不了，于是 NSD 出场。这不是「再加一个产品」，而是把书名里那条角色边界补齐。

6.2.2 nsd.conf 逐项

Unbound 自己不能对外提供 AXFR，所以本机根镜像的常见形态是：**NSD 持有一份根区 zonefile 并允许本机 AXFR，Unbound 通过 auth-zone 从 NSD 拉取**。这一步把 primary 从”远端根服务器”换成”本机 NSD”，也让 6.6 节的出向 53 豁免问题更直观。

```
# /etc/nsd/nsd.conf.d/root-mirror.conf （NSD 配置，非 unbound.conf）
server:
  ip-address: 127.0.0.1@5355
  ip-address: ::1@5355
  do-ip4: yes
  do-ip6: yes
  username: nsd
  zonesdir: /var/lib/nsd
  logfile: /var/log/nsd.log
  verbosity: 1

zone:
  name: "."
  zonefile: root.zone
  provide-xfr: 127.0.0.1 NOKEY
  provide-xfr: ::1 NOKEY
  request-xfr: AXFR 198.51.100.53 NOKEY
```

①**为什么这么设**（逐项，不是整段笼统解释）：

选项	作用	配错的后果
server.ip-address: 127.0.0.1@5355	只听回环、非默认端口	听 0.0.0.0 会把一台权威服务器暴露到全网；端口要与 Unbound 的 primary 一致
do-ip4 / do-ip6	是否启用对应地址族	关掉 v6 却给 ::1 地址会启动失败
username: nsd	降权运行	必须与实际 uid 一致，6.6 节的 nft meta skuid 豁免按它写
zonesdir / zonefile	区文件目录与相对文件名	zonefile 用相对名时以 zonesdir 为基准；路径需对 nsd 用户可读
zone.name: "."	声明持有**根区**	写成别的区名，AXFR 拉来的数据就对不上
provide-xfr: 127.0.0.1 NOKEY	允许**本机 Unbound** 来拉 AXFR	漏掉这一行是最常见的故障：Unbound 侧 auth-zone 一直建不起来，dig . AXFR 返回 Transfer failed
request-xfr: AXFR < 上游 > NOKEY	NSD 作为**从服务器**主动从真实根服务器拉区	只配 provide-xfr 不配它，本机区文件永远是空的/旧的
NOKEY	本例不用 TSIG	上游若要求签名认证，改用 key: 段定义密钥并在 request-xfr 里引用；密钥一律 <TOKEN>

NOKEY 表示本例不用 TSIG，若上游要求签名认证则按 NSD 手册配置密钥，密钥一律写 <TOKEN> / ***，不要出现真实凭据。这里 < 上游根服务器 > 是示意地址，实际部署换成公开根服务器地址。

还有一条容易忽略：**allow-query**。不写它时 NSD 默认允许任何来源查询；本例只听回环所以影响有限，但若将来把 NSD 放到独立主机上，应显式写 allow-query: <Unbound 的地址 > NOKEY，别让一台权威服务器顺带变成对全网开放的查询点。TSIG 密钥用 key: 段定义（algorithm / secret），再由 provide-xfr / request-xfr / allow-notify 按名字引用。

注意 NSD 与 Unbound 的 uid 不同：这里分别是 nsd 与 unbound。6.6 节的 nftables 豁免必须**两个 uid 都覆盖**。

Unbound 侧把 primary 指向本机 NSD：

```
auth-zone:
  name: "."
  primary: 127.0.0.1@5355
  fallback-enabled: yes
  for-downstream: no
  for-upstream: yes
  zonefile: /var/lib/unbound/root.zone
```

为什么这么设：primary 是回环地址，Unbound 侧还需要 do-not-query-localhost: no（适用条件：本机另一实例是设计中的下一跳，且端口不冲突）。这里 5355 是 NSD、Unbound 监听 53 / 5354，pidfile 与缓存目录各自独立，拓扑不成环：NSD 只做权威，不向 Unbound 发查询。

②**怎么验证**（NSD 配置用 nsd-checkconf 校验，不要用 unbound-checkconf）：

```
nsd-checkconf /etc/nsd/nsd.conf
dig @127.0.0.1 -p 5355 . SOA +norecurse +short
dig @127.0.0.1 -p 5355 . AXFR +norecurse | head -20
unbound-control stats_noreset | grep -E 'num.queries|num.cachemiss'
```

第二条应回一条根区 SOA（应答带 aa）；第三条应看到 NS / A / AAAA 胶水记录成批输出——这是 AXFR 真的可用的直接证据。若 AXFR 被拒，按这个顺序查：① nsd.conf 里有没有 provide-xfr；②它列的 IP 与 NSD 实际监听地址是否一致；③ Unbound 侧 primary 写的是 @ 端口还是 # 认证名（第 6.1 节已交代，# 是 TLS 认证名不是端口）；④出向 TCP/53 是否被 nftables 吃掉（6.6 节）。

③**失败模式 / 回滚点**：最典型的是「SOA 查得到、AXFR 拿不到」——dig . SOA 正常但 dig . AXFR 返回 Transfer failed，八成是 provide-xfr 缺失或 ACL 不对；另一种是区文件一直不更新，stat 看 zonesdir 下 root.zone 的 mtime 没前进，查 request-xfr 与出向 TCP。回滚：撤掉 Unbound 侧 auth-zone 段并 restart，fallback-enabled: yes 会让解析退回内置根提示，业务不中断；NSD 可以单独停，不影响客户端。

6.2.3　与 Unbound 的其它搭配

除根镜像外，NSD 在本书里还出现两次：第 25 章讨论内网 DDNS 时，它作为「能持有权威区、但不接受 [RFC 2136] 动态更新」的实现，说明为什么「把 Unbound 的 DDNS 配一下」是句没有对应配置的话；第 35 章的假权威 / 蜜罐里，NSD 用来给诱饵域提供受控应答。三处用法是同一个角色：**权威数据由 NSD 持有，Unbound 只负责查到结果**。如果内网要做 split-horizon（内外网不同视图），正确做法是 NSD 侧配 allow-query 与不同 zonefile，或 Unbound 侧用 local-zone / stub-zone 分开，而不是指望一台 Unbound 同时当权威和递归。

6.3　直连 AXFR 的实测形态

不引入 NSD 也可以：auth-zone 的 primary 直接写根服务器地址，Unbound 自己发起 AXFR / IXFR。

```
# /etc/unbound/unbound.conf.d/60-auth-zone-direct.conf ——示意地址
server:
  do-not-query-localhost: no
  edns-buffer-size: 1232
  max-udp-size: 1232

auth-zone:
  name: "."
  primary: 198.51.100.53
  primary: 203.0.113.53
  fallback-enabled: yes
  for-downstream: no
  for-upstream: yes
  zonefile: /var/lib/unbound/root.zone
```

为什么这么设：官方手册说 auth-zone 可以从 zonefile 读，也可以通过 AXFR / IXFR 保持更新；更新机制按 SOA 计时器工作，先发 **UDP SOA 查询**探测序列号是否变化，变了才做 IXFR / AXFR，更新完成后重写 zonefile。两条 primary 在这里是”一条失败就试下一条”（与第 5 章 forward-addr 的并发竞速语义不同，不要混淆）。

edns-buffer-size: 1232 仍然必要：SOA 探测小，但 AXFR 走 TCP，大应答场景下若之前被人为调成 1472，分片被丢会先卡在超时重试（第 4 章）。do-not-query-localhost: no 在直连形态下并非必需，写出来是为了和 6.2 节的本机

NSD 形态共用一份配置模板时少踩坑——实际不需要时应删掉，保持默认 yes 更安全。

直连 AXFR 的实测形态，按时间顺序大致是这样：

1. 启动时 zonefile 不存在或为空→ Unbound 尝试从 primary 取区数据。
2. 先向 primary 发 UDP/53 的 SOA 查询，比对序列号。
3. 序列号有变化→建立 TCP/53 做 IXFR（首次通常直接 AXFR）。
4. 收到完整区数据→写入 zonefile，后续按 SOA refresh 周期重复第 2 步。
5. 若连续失败直到 SOA expiry →区过期，查询 SERVFAIL；fallback-enabled: yes 时改走互联网递归。

怎么验证：

```
tcpdump -ni any 'host 198.51.100.53 and (tcp port 53 or udp port 53)' -c 20
ls -l /var/lib/unbound/root.zone
dig @198.51.100.53 . AXFR +norecurse | wc -l
```

抓包里应同时看到 UDP SOA 探测与 TCP 传送；手工 AXFR 的行数能给”完整区有多大”一个基线。若只看到 UDP 没有 TCP，通常是序列号没变（正常）或 TCP/53 被拦（故障）。

失败模式 / 回滚点：最典型的失败是 **TCP/53 出向被防火墙 / nftables 吃掉**——UDP SOA 探测能通，AXFR 却永远建不起来，表现为 root.zone 一直是空文件或旧时间戳。这正是下一节的主题。回滚：fallback-enabled: yes 保证解析不断，撤掉 auth-zone 段即可彻底回到内置根提示。

6.4 root-hints 与 auth-zone 的区别

这两个都和”根”有关，但完全不是一回事，混用会产生很难解释的行为。

项	root-hints	auth-zone: name "."
内容	一份根 NS 名字与地址的**提示列表**（zone file 格式，仅 NS / A / AAAA）	一份**完整根区副本**（NS、胶水、DNSKEY、RRSIG、SOA 等）
默认值	""，用内置根提示	不存在，需显式配置
作用	告诉递归器”从哪几台根服务器开始 priming”	让递归器**不出网就能拿到根区数据**
更新方式	手工换文件；官方手册提醒内置值可能随根服务器变更而过时	按 SOA 计时器 UDP 探测 + IXFR / AXFR 自动更新，重写 zonefile
对外表现	普通递归解析器	for-downstream: yes 时像根的权威服务器；no 时仍是普通递归
依赖	需要出向 53 做 priming 与后续迭代查询	需要出向 53 做 SOA 探测与区域传送（初次尤其）

一句话记：root-hints 是”通讯录”，auth-zone 是”把整本通讯录连同内容复印回家”。前者只解决起点问题，后者才让根这一跳落在本地。

怎么验证：

```
unbound-checkconf -o root-hints
unbound-control lookup example.org | head -20
```

第一项为空说明用内置根提示；第二项能看出递归时选中的名字服务器集合。若两者都配了，以 auth-zone 的本地副本优先（for-upstream: yes 时它在出网前就被查），root-hints 只在本地副本未命中 / fallback 时才起作用。

失败模式 / 回滚点：把 root-hints 指向一台不可达的根服务器，会让 priming 一直超时；把 auth-zone 的 zonefile 当成 root-hints 文件用（后者只该含 NS 与地址）会导致解析异常。回滚时一次只撤一个，别同时改两处。

6.5 zonefile 持久化：重启别把根区弄丢

```
auth-zone:
  name: "."
  primary: 198.51.100.53
```

```
  fallback-enabled: yes
  for-downstream: no
  for-upstream: yes
  zonefile: /var/lib/unbound/root.zone
```

为什么这么设：官方手册对 `zonefile` 的说明很直白——不给就没有 zonefile；文件不存在或为空时，Unbound 会尝试从 `primary` 重新取区数据，更新完成后**重写该文件**。也就是说：

- 不配 `zonefile`：区数据只在内存里，每次重启都要重新做一遍完整 AXFR。若重启时出向 53 恰好不通，区就建立不起来，只能靠 `fallback-enabled` 兜底。
- 配了 `zonefile`：重启后先从磁盘载入已有副本，立刻可用，再按 SOA 计时器做增量更新。

因此 `zonefile` 不只是”顺手存个档”，它是**冷启动可用性**的一部分。三个工程约束：

1. 路径必须对 `username: unbound` 降权后的进程可写；同时若启用了 `chroot`（第 4 章），路径在 chroot 内解释，要一并搬进去。
2. AppArmor / SELinux 需要放行该路径的读写；过滤表大表与 AppArmor 路径白名单的冲突同样适用于根区 zonefile。
3. 目录要持久（`/var/lib/unbound/`），不要放在 `/run` 或 tmpfs 上——否则”配了 zonefile”和”没配”在重启后是同一回事。

官方手册还有个相关提醒：区过期（SOA expiry 到达）时查询会 SERVFAIL，且此后**即便主服务器发来更旧的序列号也会被接受**，因为本地已经没有可用副本了；此时 `fallback-enabled` 才会切到互联网递归。这意味着 zonefile 的时间戳与 SOA 序列号是排障时最先要看的两个值。

怎么验证：

```
stat -c '%y %s %U' /var/lib/unbound/root.zone
dig @198.51.100.53 . SOA +norecurse +short | awk '{print $3}'
grep -m1 -A3 '^\.' /var/lib/unbound/root.zone | grep -i serial
sudo -u unbound test -w /var/lib/unbound/root.zone && echo writable
```

失败模式 / 回滚点：文件属主是 root 且权限 0644 → Unbound 能读不能写，AXFR 更新后重写失败，日志报错但解析暂时照旧，直到重启暴露问题。`sudo -u unbound test -w` 是这条最便宜的检查。回滚：把 zonefile 移到正确目录并修正属主，或临时撤掉 `zonefile` 行靠 `primary` 每次重拉（接受冷启动代价）。

6.6　出向 53 必须按 uid 豁免，否则自己吃掉自己

这是本章的核心故障机理。

策略网关常把 LAN / WG 侧发往 53 的查询劫持进透明代理（`hijack-dns`），让所有客户端的 DNS 强制走本地解析器。规则一般挂在 `output-tproxy` 一类链上。问题在于：**本机解析器进程自己发出的出向 53 也会被这条规则匹配**。Unbound 要向根服务器发 SOA 探测 / AXFR，NSD 要向根服务器发 `request-xfr`，这些包同样从本机 `output` 出去，同样目的端口 53，于是被送回 tproxy，落到别的解析器或代理上——甚至形成环。

结果不是”慢一点”，而是**根区镜像永远不更新**：`root.zone` 时间戳停在初次拉取那一刻，SOA 探测没有真实应答，IXFR 永不触发；NSD 的 `request-xfr` 一直失败，AXFR 建不起来。若同时 `fallback-enabled: no`，客户端最终会看到间歇性全量 SERVFAIL，很容易被误判成”上游根服务器不稳”。

正确做法是在 `output-tproxy` **链首**按 uid 显式 accept，把两个解析器进程的出向查询放过去：

```
#!/usr/sbin/nft -f
# /etc/nftables.d/10-dns-exempt.conf ——教学用示意规则
table inet filter {
  chain output-tproxy {
    # 本机解析器自己发出的 53 查询必须豁免，否则被自己的 hijack-dns 吃掉
    meta skuid unbound udp dport 53 accept
    meta skuid unbound tcp dport 53 accept
    meta skuid nsd     udp dport 53 accept
    meta skuid nsd     tcp dport 53 accept

    # 下面才是原本对 LAN / WG 客户端的劫持规则（示意）
    # ip saddr 192.0.2.0/24 udp dport 53 meta mark set 0x2332
    # ...
```

```
  }
}
```

为什么这么设：豁免必须覆盖 **UDP 与 TCP 两种、unbound 与 nsd 两个 uid**。只放 UDP 会导致 SOA 探测通、AXFR 的 TCP 建连失败，正好对应 6.3 节那个”只看到 UDP 没有 TCP”的实测形态；只放 `unbound` 会让 NSD 的 `request-xfr` 被吃掉，本机 NSD 形态（6.2 节）下根区照样不更新。

顺序也关键：豁免规则要放在链首。同类问题上还有个已被验证的坑——`meta skuid 0` 匹配不到内核发出的某些包，必须按实际特征（例如 `udp sport`）显式 accept，否则 WireGuard 自己的回包也会被 `meta mark set 0x2332` 打标送进代理，握手响应到不了对端、隧道起不来。这两件事同源：**本机 / 内核发出的包不能被面向客户端的劫持规则捕获**。

`meta skuid` 匹配的是降权后的实际 uid，所以第 4 章的 `username: unbound` 与 NSD 的 `username: nsd` 必须和这里的规则一致。若你改成 `username: ""`（不降权），这些豁免全部失效——这也是不建议在生产里留空 username 的原因之一。

怎么验证：

```
nft list chain inet filter output-tproxy
ps -o user=,pid=,comm= -C unbound -C nsd
tcpdump -ni any 'host 198.51.100.53 and (tcp port 53 or udp port 53)' -c 20
stat -c '%y' /var/lib/unbound/root.zone
```

`nft list` 里四条豁免要在劫持规则之前；`ps` 确认 uid 与规则一致；`tcpdump` 应看到真实的出向 53 报文抵达示意根服务器地址（而非被重定向到本机代理端口）；`stat` 的时间戳应在 SOA refresh 周期之后前进。四条一起成立才算镜像真的在更新。

失败模式 / 回滚点：改完 nftables 仍不更新，先看规则命中计数（`nft list ruleset -a` 配合 `nft list chain ... -a`），确认豁免规则有没有被匹配；再看是不是 `primary` 用了文档示意地址而实际根本不可达。回滚：`nft flush` 前保留一份可回滚的规则文件，或临时 `nft delete rule` 撤掉新增豁免观察差异；解析业务本身有 `fallback-enabled: yes` 兜底，不会立刻全断。

6.7 验收与镜像失效演练

把本章各段收成一次可复现的验收。下面的顺序故意从”数据是否存在”查到”链路是否真的出去”，避免一上来就怀疑网络：

```
#!/bin/sh
# 本地根区镜像验收 ——教学用示意地址
set -eu

echo '== 1. 配置语法 =='
unbound-checkconf /etc/unbound/unbound.conf
nsd-checkconf /etc/nsd/nsd.conf

echo '== 2. zonefile 落盘与属主 =='
stat -c '%y %s %U' /var/lib/unbound/root.zone
sudo -u unbound test -w /var/lib/unbound/root.zone && echo 'writable by unbound'

echo '== 3. 本机 AXFR 可用（NSD 形态） =='
dig @127.0.0.1 -p 5355 . AXFR +norecurse | wc -l

echo '== 4. uid 与豁免规则一致 =='
ps -o user=,comm= -C unbound -C nsd
nft list chain inet filter output-tproxy | grep -E 'skuid|accept' | head

echo '== 5. 递归答案正常、根这一跳本地化 =='
dig @127.0.0.1 example.org A +short
unbound-control stats_noreset | grep -E 'total.num.cachemiss|num.recursivereplies'

echo '== 6. 镜像失效演练 =='
# 临时断掉示意根服务器的出向（按实际环境替换），观察 fallback 是否接管
# nft add rule inet filter output ip daddr 198.51.100.53 drop
# dig @127.0.0.1 example.org A +short   # 应仍有答案
# nft flush chain inet filter output    # 恢复
```

为什么这么设：第 6 步是刻意保留的注释式演练命令，不是自动执行——断链规则会影响同一台机器上其他业务，必须由人按环境替换地址并确认回滚路径。演练的预期是：`fallback-enabled: yes` 时解析继续可用（但根这一跳重新出网），`fallback-enabled: no` 时会出现 SERVFAIL，让镜像失效可见。两种都是正确行为，取决于你选的可用性策略。

怎么验证：演练前后各跑一次 `dig` 与 `unbound-control stats_noreset`，对比 `num.cachemiss` 与 `num.recursivereplies`；恢复后 `stat` 的 zonefile 时间戳应在下一个 SOA refresh 周期后前进。

失败模式 / 回滚点：`nft flush chain` 会把整条链清空，若链里还有别的规则必须逐条删除而不是 flush。所有变更的回滚点都是配置文件 + 一份 nftables 备份；根区 zonefile 本身是可重新拉取的派生数据，删掉不会损坏系统。

镜像仍要求本机有出向 53，因此它**不是**第 5 章转发器的替代品，也不是无网环境下的完整方案。本地根镜像只让根那一跳不出口，QMIN 的收益落在 TLD 层。

6.8 本章要点

- `auth-zone: name "."` 的四个开关：`primary` 用 AXFR / IXFR 拉副本，`for-upstream: yes` 让递归从本地副本取根数据，`for-downstream: no` 保持普通递归语义（CNAME 被彻底解析），`fallback-enabled: yes` 在区失效时退回互联网递归。
- `for-downstream: yes` / `for-upstream: yes` 是两个不同阶段：前者在 local-zone 之后、缓存之前，对下游像权威服务器；后者在缓存之后、出网之前；官方手册把 `for-downstream: no` + `for-upstream: yes` 称为 hyperlocal 组合。
- Unbound 不提供 AXFR / IXFR，所以本机根镜像常用 NSD 持有 zonefile 并对回环 `provide-xfr`，再由 NSD 的 `request-xfr` 向真实根服务器拉区；`primary` 也可直连根服务器，实测形态是”UDP SOA 探测序列号→ TCP IXFR/AXFR →重写 zonefile”。
- `root-hints` 只是根 NS 与地址的提示列表（默认 `""` 用内置值），`auth-zone` 是完整根区副本；前者解决起点，后者才让根这一跳不出本机。
- `zonefile` 必须对 `unbound` uid 可写、位于持久目录、且在 chroot / AppArmor 白名单内；不配则每次重启都要重做完整 AXFR，冷启动时出向不通就没副本可用。
- 本机解析器进程（`unbound` uid、`nsd` uid）的出向 53（**`UDP 与 TCP 两种`**）必须在 `nft output-tproxy` 链首按 uid 豁免，否则被自己的 `hijack-dns` 吃掉，表现为根区镜像永远不更新、AXFR 失败；排查这类故障先看 nftables 规则。

6.9 参考资料

- Unbound 官方手册，`unbound.conf(5)` 的 Authority Zone Options（`auth-zone`、`primary`、`fallback-enabled`、`for-upstream`、`for-downstream`、`zonefile`）：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html
- Unbound 官方手册，`unbound.conf(5)` 的 `root-hints` 与 server options：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html
- RFC Editor，[RFC 9103] *DNS Zone Transfer over TLS*：https://www.rfc-editor.org/rfc/rfc9103
- NLnet Labs，NSD 官方配置手册（`nsd.conf(5)` 的 `zone` / `provide-xfr` / `request-xfr` / `allow-query` / `key`）：https://nsd.docs.nlnetlabs.nl/en/latest/manpages/nsd.conf.html
- NLnet Labs，NSD 项目页（权威服务器定位）：https://nlnetlabs.nl/projects/nsd/about/
- Unbound 官方手册，`unbound-control(8)` 的 `lookup` / `flush_zone` 等命令：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound-control.html
- IANA，根区与根服务器信息入口：https://www.iana.org/domains/root/servers
- RFC Editor，[RFC 1995] *Incremental Zone Transfer in DNS*（IXFR）：https://www.rfc-editor.org/rfc/rfc1995
- RFC Editor，[RFC 9103] *DNS Zone Transfer over TLS*：https://www.rfc-editor.org/rfc/rfc9103
- RFC Editor，[RFC 1034]，*Domain names - concepts and facilities*：https://www.rfc-editor.org/rfc/rfc1034
- RFC 5936，*DNS Zone Transfer Protocol (AXFR)*：https://www.rfc-editor.org/rfc/rfc5936
- RFC 1995，*Incremental Zone Transfer in DNS*（IXFR）：https://www.rfc-editor.org/rfc/rfc1995

第七章

多实例、缓存与可观测性

一台机器上只跑一个 Unbound 实例时，第 4–6 章的配置都还直观：改配置、restart、看 dig。一旦把角色拆开——一个实例做过滤与客户端入口，一个做递归 / 本地根区镜像，一个做加密出口——就会冒出三类新问题：端口与接口怎么划、过滤表大表吃掉多少内存、以及”到底哪一层缓存里有旧数据”。

7.1 同一主机多实例的端口 / 接口规划

多实例最容易出的低级错是”配置文件都对了，但两个进程抢同一个东西”。先把独占资源列全：

资源	冲突表现	规划原则
interface / port	第二个实例启动报地址已占用	每个实例一组独立 IP@ 端口
pidfile	后启动的覆盖前者，systemctl 管错进程	每实例独立路径，如 /run/unbound-filter.pid
remote-control 的 control-interface / 端口	unbound-control 连到别的实例	每实例独立控制端口，或只给一个实例开
证书 / 密钥文件	控制连接认证失败	每实例一套 server-key-file 等四件套
chroot / 工作目录	相对路径互相覆盖	不共用；本章统一 chroot: ""
zonefile / 缓存落盘路径	根区副本互相踩	每实例独立目录
uid 豁免规则（nft meta skuid）	第 6 章的根区镜像不更新	所有实例的 uid 都要覆盖

一个可复制的三实例规划。注意 interface 用 @port 后缀，这是 Unbound 原生支持的写法：

为什么这么设（三段配置合起来看）：三实例分别承担入口过滤、递归 / 本地根区镜像、加密出口，因此端口段、pidfile、zonefile 目录与控制端口必须互不重叠；A 把 B 当作唯一 forward-addr，B 有真实出向 53，C 走 853。这三点共同决定了下面三段配置里的每一个端口与路径。

```
# 实例 A：客户端入口 + 过滤表，/etc/unbound/filter.conf
server:
  interface: 192.0.2.1@53
  interface: 127.0.0.1@5301
  access-control: 192.0.2.0/24 allow
  access-control: 127.0.0.0/8 allow
  username: unbound
  chroot: ""
  pidfile: /run/unbound-filter.pid
  num-threads: 2
  msg-cache-size: 64m
  rrset-cache-size: 128m
  do-not-query-localhost: no
  edns-buffer-size: 1232
  max-udp-size: 1232
```

```
  log-servfail: yes

forward-zone:
  name: "."
  forward-addr: 127.0.0.1@5302
  forward-first: no

remote-control:
  control-enable: yes
  control-interface: 127.0.0.1@8953

# 实例 B：递归 + 本地根区镜像，/etc/unbound/recursor.conf ——教学用示意地址
server:
  interface: 127.0.0.1@5302
  access-control: 127.0.0.0/8 allow
  username: unbound
  chroot: ""
  pidfile: /run/unbound-recursor.pid
  num-threads: 2
  msg-cache-size: 64m
  rrset-cache-size: 128m
  edns-buffer-size: 1232
  max-udp-size: 1232
  log-servfail: yes

auth-zone:
  name: "."
  primary: 198.51.100.53
  fallback-enabled: yes
  for-downstream: no
  for-upstream: yes
  zonefile: /var/lib/unbound-recursor/root.zone

remote-control:
  control-enable: yes
  control-interface: 127.0.0.1@8954

# 实例 C：加密出口（示意），/etc/unbound/egress.conf
server:
  interface: 127.0.0.1@5303
  access-control: 127.0.0.0/8 allow
  username: unbound
  chroot: ""
  pidfile: /run/unbound-egress.pid
  tls-cert-bundle: /etc/ssl/certs/ca-certificates.crt
  log-servfail: yes

forward-zone:
  name: "."
  forward-addr: 192.0.2.53@853#dns.example.org
  forward-tls-upstream: yes
  forward-first: no

remote-control:
  control-enable: yes
  control-interface: 127.0.0.1@8955
```

为什么这么设：三个实例分别承担”入口过滤 / 递归 / 加密出口”，端口段 5301–5303 与控制端口 8953–8955 全不重叠；`pidfile`、`zonefile` 目录、控制端口各自独立。A 用 `do-not-query-localhost: no` 把 B 当作唯一 `forward-addr`，条件与第 5 章一致：下一跳是本机另一实例、端口不冲突、拓扑不成环（B 做递归 / 本地根镜像，不会转发回 A）。

B 是第 6 章的形态，因此它**需要真实出向 53** 做 SOA 探测与 AXFR；C 走 853 的加密传输（DoT），与 DNSSEC 的数据完整性 / 来源认证不是一回事（第三篇第 11 章展开服务端 DoT）。

三个实例都以 username: unbound 运行，所以第 6 章的 nft output-tproxy 豁免规则里 meta skuid unbound udp/tcp dport 53 accept 对它们同时生效；若你把某个实例改成别的 uid，别忘了补规则。

怎么验证：

```
for f in filter recursor egress; do unbound-checkconf /etc/unbound/${f}.conf || exit 1; done
ss -lunp | awk '$5 ~ /:(53|5301|5302|5303|8953|8954|8955)$/ {print}'
unbound-control -c /etc/unbound/filter.conf status
unbound-control -c /etc/unbound/recursor.conf status
unbound-control -c /etc/unbound/egress.conf status
```

unbound-control -c 指定配置文件是关键：它会读取该文件里的 remote-control: 段去连对应控制端口，不指定就会连到默认实例，出现”命令执行成功但改的不是我想改的实例”。

失败模式 / 回滚点：ss 里缺某个端口→先检查 interface 是否只 reload 没 restart（第 4 章：interface 变更只有 restart 生效）；控制端口连不上→查 control-interface 与证书四件套是否各实例独立。回滚：每实例的配置、pidfile、zonefile 都是独立文件，逐个 systemctl stop 即可，不会互相牵连。

7.2 按域名分流到不同实例

“按域名分流”有两种完全不同的实现，先分清再动手：

1. **在单个实例内部按区（zone）分流**：用 forward-zone: name: "example.org" / stub-zone 把特定后缀交给指定上级，其他查询走默认路径。这是 Unbound 原生的能力，不需要多进程。
2. **让不同客户端落到不同实例**：靠 interface 地址 / 端口区分，由上游设备（RA/RDNSS、DHCP、策略路由）决定客户端问哪一个。第五篇第 19–21 章讲 RA / RDNSS 与网关工程，这里只给 DNS 侧的接口。

第 1 种的写法，把内部域名交给内网权威服务器、其余交给唯一上游：

```
# /etc/unbound/filter.conf 的追加片段
# 需与 7.1 节的 server: 段合并使用：那里已有 do-not-query-localhost: no
server:
  do-not-query-localhost: no

forward-zone:
  name: "corp.example.net."
  forward-addr: 192.0.2.53
  forward-first: no

forward-zone:
  name: "."
  forward-addr: 127.0.0.1@5302
  forward-first: no
```

为什么这么设：多个 forward-zone 段可以共存，官方手册说”按最接近查询名的区生效”，所以 corp.example.net. 比 . 更具体、优先命中。两条都写 forward-first: no，保证任一路径失败时不会逃出去自行递归。注意每个区仍只给一条 forward-addr：第 5 章的”并发竞速取最快”结论对每一个 forward-zone 都成立，多写一条就会绕过既定上游。

第 2 种是让不同实例监听不同地址，客户端按需选：

客户端类别	询问地址	承担实例
普通 LAN 客户端	192.0.2.1@53	A（入口 + 过滤）
需要完整递归的排障主机	127.0.0.1@5302	B（递归 + 根镜像）
只走加密出口的进程	127.0.0.1@5303	C（DoT 出口）
IPv6 / ULA 客户端	fd00::1@53	A（同一实例另一 interface）

怎么验证：

```
dig @127.0.0.1 -p 5302 host.corp.example.net A +short
dig @192.0.2.1 host.corp.example.net A +short
dig @127.0.0.1 -p 5303 example.org A +short
unbound-control -c /etc/unbound/filter.conf list_forwards
```

失败模式 / 回滚点：最常见的错是”配了具体区却忘了默认区”，导致不匹配的名字无路可走；或者反过来，把具体区写成了 .，等于没分流。回滚：删掉具体 forward-zone 段并 restart，所有查询回落到默认区。还要注意 uid 豁免——B 实例的分流若把出向 53 指向了自己，会形成自问自答的环。

7.3　过滤表（local-zone 大表）的内存代价与 AppArmor 路径限制

```
# /etc/unbound/filter.conf 的追加片段
server:
  # 过滤表通常通过 include 引入，路径必须在 AppArmor 白名单内
  include: /etc/unbound/filter.d/*.conf

  private-address: 192.0.2.0/24
  private-address: 10.0.0.0/8
  private-address: fd00::/8
  private-address: 2001:db8::/32
```

为什么这么设：local-zone 大表是过滤的主流做法——每个要挡的域名一条 local-zone: "name." always_nxdomain 或 refuse。官方手册允许用 include 拆分配置文件，但这里有两个工程约束。

内存代价：local-zone 条目在启动时全部载入内存，不是按需查磁盘。表越大，常驻内存越高；叠加 msg-cache-size / rrset-cache-size（按 1:2 放大、实际还有 2–2.5 倍 malloc 开销）与每个实例的独立开销，三实例机器上很容易把内存吃满。用 unbound-control stats_noreset 里的 mem.* 计数器观察，别只盯着 RSS；如果过滤表从十万条涨到百万条，先做一次内存基线再上线。RPZ 是另一种过滤机制（官方手册单列一节），大表场景下可以评估是否比逐条 local-zone 更省，但那是另一套配置。

AppArmor 路径限制：include 的相对路径在 chroot / 工作目录内解释，而 AppArmor / SELinux 只允许进程读白名单里的路径。把过滤表放到 /etc/unbound/filter.d/ 之外的新目录（例如 /srv/blocklist/）而不更新 profile，现象是启动失败或表里根本没加载——unbound-checkconf 可能照样通过，因为检查的是语法而不是 LSM 放行。第 6 章的根区 zonefile 落盘也受同一条限制：路径要同时对 unbound uid 可写、在 chroot 内可见、在 AppArmor 白名单内。

private-address 与过滤表是两件事：它把指定网段从**公网域名的应答**里剔除，防 DNS Rebinding；官方手册说明它移除 A / AAAA / SVCB / HTTPS 里匹配的资源记录（含 SVCB 的 ipv4hint / ipv6hint 参数），且启用 DNSSEC 验证时可能把含私有地址的应答标为 bogus。这里把内网段与文档地址段都列上，仅作教学示例。**副作用要提前知道**：若你自己用 192.0.2.x 作为内网服务的权威应答，也会被剔除，需要用 local-zone / local-data 或 private-domain 例外处理。

怎么验证：

```
unbound-control -c /etc/unbound/filter.conf list_local_zones | wc -l
unbound-control -c /etc/unbound/filter.conf stats_noreset | grep -E 'mem\.|total.num.queries'
aa-status 2>/dev/null | grep -i unbound || true
dig @192.0.2.1 blocked.example.net A +short
```

失败模式 / 回滚点：list_local_zones | wc -l 远小于预期条数，先怀疑 include 通配符没匹配到、或 AppArmor 拒绝读取；journalctl -k / audit.log 里的 DENIED 是证据。回滚：把过滤表放回白名单内的既有目录，或临时注释掉 include 行 restart（过滤失效是预期代价，要同步告知使用方）。

7.4　DDR 压制与 always_nodata

```
# /etc/unbound/filter.conf 的追加片段
server:
  local-zone: "_dns.resolver.arpa." always_nodata
  local-zone: "use-application-dns.net." always_nxdomain
  local-zone: "blocked.example.net." always_nxdomain
  local-zone: "ads.example.org." always_nodata
```

为什么这么设：DDR（Discovery of Designated Resolvers）允许客户端通过 _dns.resolver.arpa. 自动发现采用加密传输（DoT / DoH / DoQ）的服务器，然后绕过本地解析器直连外部 DoH / DoT。给这个特区配 always_nodata（返回 NODATA，且忽略 local-data）就是压制自动发现，让客户端继续用本地实例。官方手册对 always_nodata 的定义是”像 static，但忽略 local data 并返回 NODATA”；always_nxdomain 同理返回 NXDOMAIN。

浏览器侧还有 canary 域：Firefox 用 use-application-dns.net 判断该不该启用自带 DoH，回 NXDOMAIN 即可关闭；Chrome 靠系统解析器是否匹配已知 DoH 提供商列表来判断，不需要按域名封。**按 IP 封 443 是 denylist 而不是证明**，会误伤且随时失效，别把它当主要手段。

注意这里 always_nodata / always_nxdomain 的语义与 static 不同：它们忽略该区下的 local-data，因此”整区一律这样答”的场景才用；需要给个别名字留白名单时，应改用 transparent 或配合具体 local-data。

怎么验证：

```
dig @192.0.2.1 _dns.resolver.arpa. SVCB +short
dig @192.0.2.1 use-application-dns.net A +short
dig @192.0.2.1 blocked.example.net A
```

第一条应为空且 status: NOERROR（NODATA），第二条 status: NXDOMAIN，第三条同样 NXDOMAIN。若第一条返回了 SVCB 记录，说明 local-zone 没生效或客户端问的是另一个实例。

失败模式 / 回滚点：过度压制会挡掉合法的内部 SVCB / HTTPS 记录使用（例如自己部署的服务发现），排查时先确认是不是这条 always_nodata 吞掉了。回滚：把对应 local-zone 行改为 nodefault 或删掉并 reload。

7.5 dnstap 全量查询流与 30 天保留期

dnstap 是结构化二进制日志，**记录的是自己的查询流副本**：客户端问了什么、本实例向谁问了什么、回了什么。它对抗”元数据在别处被聚合”很有用，但代价是把同样的元数据集中到自己门口。开启全量日志的同时必须写清保留期并配套轮转与到期删除。

```
# /etc/unbound/filter.conf 的追加片段
dnstap:
  dnstap-enable: yes
  dnstap-socket-path: /run/dnstap.sock
  dnstap-bidirectional: yes
  dnstap-log-client-query-messages: yes
  dnstap-log-client-response-messages: yes
  dnstap-log-resolver-query-messages: yes
  dnstap-log-resolver-response-messages: yes
  dnstap-log-forwarder-query-messages: yes
  dnstap-log-forwarder-response-messages: yes
```

为什么这么设：官方手册说明 dnstap 需编译时带 --enable-dnstap，启用后（带线程编译时会）另起一个线程把日志写到目标；dnstap-socket-path 指向接收端监听的 unix socket，dnstap-ip 为空时就用 socket。上面六项 dnstap-log-*-messages 分别覆盖”客户端查询 / 应答、递归器查询 / 应答、转发器查询 / 应答”，六项全开才是全量。多实例机器上要注意：A 是转发器角色（会产 forwarder 类消息），B 是递归角色（会产 resolver 类消息），C 也是转发器，三份日志合起来才是完整链路视图。

dnstap-tls 默认 yes，只在用 dnstap-ip 走 TCP 时才涉及；unix socket 形态下用不到。接收端（例如 dnstap 收集器）把帧流写盘，落盘路径同样要过 AppArmor 白名单。

保留期的落地写法（示意，接收端落盘后由 logrotate / timer 执行）：

```
#!/bin/sh
# /usr/local/sbin/dnstap-retention.sh ——保留 30 天，教学用示例
set -eu
DIR=/var/log/dnstap
find "$DIR" -type f -name '*.dnstap' -mtime +30 -delete
find "$DIR" -type f -name '*.dnstap' -mtime +1 -exec gzip -9 {} +
```

为什么这么设：-mtime +30 -delete 明确实现 30 天保留，+1 起压缩省空间。没有这一步，“抗元数据聚合”在自己门口破功：一年后磁盘里躺着完整的全屋查询历史，风险比上游聚合更集中。保留期要写进运维文档并定期审计，不要只放在脚本里。

怎么验证：

```
unbound-checkconf /etc/unbound/filter.conf
ls -l /run/dnstap.sock
find /var/log/dnstap -type f -name '*.dnstap*' -printf '%TY-%Tm-%Td %p\n' | sort | head
find /var/log/dnstap -type f -name '*.dnstap*' -mtime +31 | wc -l
```

最后一条应为 0（若有残留说明清理任务没跑）。接收端是否真的收到帧，用收集器自带的计数或 dnstap-* 工具查看，具体命令随实现不同，不要凭 ls 有文件就断定内容完整。

失败模式 / 回滚点：dnstap-enable: yes 但 socket 不存在→连接失败，日志里持续报错，解析本身通常不受影响；编译没带 dnstap 时选项可能被忽略。回滚：dnstap-enable: no 并 restart，同时确认保留期脚本仍在跑（历史文件不会因为关掉采集就自动消失）。

7.6　unbound-control 开启与不开的取舍

remote-control 默认 control-enable: no。开启需要 unbound-control-setup 生成自签证书四件套（server-key-file、server-cert-file、control-key-file、control-cert-file），客户端与服务端都读同一段配置。

维度	开启 unbound-control	不开
刷缓存	flush / flush_zone / flush_type 精确清除	**只能重启实例**清空整份缓存
统计	stats / stats_noreset 随时取	只能靠 statistics-interval 打日志或 dnstap
在线改配置	set_option、list_forwards、list_local_zones 等	只能改文件 + restart / reload
暴露面	多一个控制端口 + 证书文件权限管理	无控制端口
多实例	需每实例独立控制端口与证书	无此问题，但也无法定位刷哪一层

不开时的代价要特别算清：负缓存（negative caching）会放大故障——上游一次瞬时 NXDOMAIN 按 SOA minimum 缓存（常见 3600 秒），修好权威服务器后客户端仍在失败。unbound-control 关着时只能重启实例，而**链路上每个实例都要刷**：7.1 节里 A（入口过滤）、B（递归 / 根镜像）、C（加密出口）各自有独立缓存，只重启 A 的话，B 里那条缓存的 NXDOMAIN 照样返回给 A。同一类问题还有 qname-minimisation-strict: yes 的兼容坑：某大型 CDN 的 CNAME 区违反 [RFC 8020]（父名 NXDOMAIN、子名却存在），strict 模式下中间层拿到 NXDOMAIN 就不回退完整域名，导致整站 CSS/JS 全挂；改成 strict: no 后**链路上每一层缓存都要刷**，否则前一层照样返回缓存的 NXDOMAIN。参数本身在第三篇第 13 章，这里只处理”缓存怎么清”。

因此本书的建议是：**多实例部署必须开 unbound-control**，但把 control-interface 严格限制在回环、证书文件属主与权限收紧、每实例独立端口。单实例且完全不需要运行时观测的极简部署才考虑不开，并接受”排障靠重启”的代价。

```
# 精确刷缓存：先定位是哪一层，再决定刷哪个实例
unbound-control -c /etc/unbound/recursor.conf flush_zone example.org
unbound-control -c /etc/unbound/filter.conf  flush_zone example.org
unbound-control -c /etc/unbound/egress.conf  flush_zone example.org

# 只清坏条目 / 只看不清零的统计
unbound-control -c /etc/unbound/recursor.conf flush_bogus
unbound-control -c /etc/unbound/recursor.conf stats_noreset | grep -E 'total.num.cachehits|total.num.cachemiss'
```

为什么这么设：flush_zone name 清除该名字及以下的所有缓存信息，比整实例重启温和得多；flush_bogus 清掉 DNSSEC 相关的坏条目与迭代器失败记录。三条命令逐个实例执行，就是把”链路上每个实例都要刷”落成可执行动作。统计一律用 stats_noreset——官方手册说明 stats 默认会重置计数（受 statistics-cumulative 控制），做基线时把自己读没的数据抹掉很冤。

怎么验证：

```
unbound-control -c /etc/unbound/recursor.conf cache_lookup example.org | head
unbound-control -c /etc/unbound/recursor.conf lookup example.org | head
dig @192.0.2.1 example.org A +short
```

cache_lookup 刷前后应看到条目消失 / 重建；lookup 显示”将会用哪些名字服务器查这个名字”，可用于确认根镜像是否生效。

失败模式 / 回滚点：控制证书权限不对（降权后 unbound 用户读不到）→连不上，journalctl 里有 TLS / 文件权限错误。回滚：control-enable: no 并 restart，回到”只能重启刷缓存”的模式；此时务必把链路上所有实例列成清单，避免漏刷。

7.7 日志字段与 SERVFAIL 排查

```
# 各实例 server: 段的公共片段
server:
  verbosity: 1
  log-servfail: yes
  statistics-interval: 0
  extended-statistics: yes
  log-time-ascii: yes
```

为什么这么设：log-servfail: yes 专门打印”查询为什么回了 SERVFAIL”，官方手册说它独立于 verbosity 的调试日志、体积小、且以 error 级别输出——这是排障时性价比最高的一行，不必把 verbosity 拉到 4 把磁盘灌满。extended-statistics: yes 让统计里多出查询类型等维度；statistics-interval: 0 关掉周期性日志打点，改由 unbound-control 按需取（官方监控页也是这个建议）。log-time-ascii 让时间戳可读，便于把 dnstap、journald、nftables 计数三处时间对齐。

怎么验证：改完 systemctl restart 后查一次 unbound-control -c < 对应配置 > stats_noreset | grep -E 'num.queries|num.answer.bogus|recursion.time'，确认扩展统计字段已出现；unbound-control -c < 对应配置 > get_option log-servfail 应回 yes；journalctl -u unbound-filter -n 20 应能看到可读的 ASCII 时间戳。

失败模式 / 回滚点：四行是各实例 server: 段的公共片段，不要单独作为一个配置文件加载；若某个实例的 include 顺序把它覆盖掉，排查时先比对生效值（unbound-control -c < 配置 > get_option log-servfail）。回滚：删掉追加的 extended-statistics / log-time-ascii 两行，保留 log-servfail: yes 即可。

一条 SERVFAIL 的排查顺序，按”离客户端由近到远”：

1. **哪个实例回的**：dig 的 SERVER: 行 + unbound-control -c < 对应配置 > stats_noreset 看 num.queries / num.SERVFAIL 类计数。多实例时最容易在这一步认错对象。
2. **是不是访问控制**：access-control 无匹配默认 refuse（回 REFUSED，不是 SERVFAIL）；REFUSED 与 SERVFAIL 要分清。
3. **看 log-servfail 的具体原因**：常见是上游超时、DNSSEC 验证失败（bogus）、根区 / 转发区无法建立。第 5 章的 failed to prime trust anchor、第 6 章的区过期都在这里出现。
4. **是不是缓存里的旧负面结果**：unbound-control cache_lookup <name> 看有没有缓存的 NXDOMAIN / bogus；有就按 7.6 节逐层刷。
5. **是不是出向链路被自己劫持**：第 6 章的结论——先看 nft list ruleset，确认 meta skuid unbound udp/tcp dport 53 accept 在劫持规则之前。SOA 探测通但 AXFR 建不起来，就是 TCP/53 没豁免。
6. **是不是 EDNS / MTU**：edns-buffer-size 被改成 1472 导致分片被丢，表现为”超时后重试 TCP”而非立刻报错（第 4 章）。

怎么验证：

```
journalctl -u unbound-filter --since '10 min ago' | grep -iE 'servfail|error|prime' | tail -30
unbound-control -c /etc/unbound/recursor.conf stats_noreset | grep -E 'num.queries_timed_out|num.recursivereplies|num.expired'
dig @192.0.2.1 example.org A | grep -E 'status:|SERVER:'
```

失败模式 / 回滚点：只把 verbosity 调高而不开 log-servfail，会得到大量无关调试行却仍看不清 SERVFAIL 原因。回滚：verbosity: 1 + 保留 log-servfail: yes。num.queries_timed_out 持续增长而 num.recursivereplies 不涨，基本可以断定是出向链路或上游，不是本机缓存。

7.8 缓存命中率与延迟的基线测量方法

没有基线就谈不上”调优有效”。下面给一套可复制的测量法，全部用 stats_noreset 与 dig，不依赖外部监控栈。

```
#!/bin/sh
# /usr/local/sbin/unbound-baseline.sh
# 用法: unbound-baseline.sh < 实例配置 > < 监听地址 > [端口]
set -eu
CONF=${1:?实例配置}
ADDR=${2:?监听地址}
PORT=${3:-53}
CTRL="unbound-control -c $CONF"

# 1) 清空计数、冷缓存起点
```

```
$CTRL flush_stats >/dev/null
$CTRL flush_zone . >/dev/null 2>&1 || true

# 2) 固定样本集：冷查一次 + 热查三次
NAMESET="example.org example.net example.com"
for n in $NAMESET; do dig @${ADDR} -p ${PORT} "$n" A +short >/dev/null; done
$CTRL stats_noreset | grep -E 'total.num.cachehits|total.num.cachemiss' > /tmp/ub-cold.$$
for i in 1 2 3; do
  for n in $NAMESET; do dig @${ADDR} -p ${PORT} "$n" A +short >/dev/null; done
done
$CTRL stats_noreset | grep -E 'total.num.cachehits|total.num.cachemiss' > /tmp/ub-hot.$$

# 3) 延迟样本：dig 输出的 "Query time"
for n in $NAMESET; do
  dig @${ADDR} -p ${PORT} "$n" A | grep 'Query time'
done

# 4) 递归耗时分布（仅统计需要递归的查询，缓存命中不计入）
$CTRL stats_noreset | grep -E 'recursion.time.avg|recursion.time.median|num.prefetch|num.expired|num.queries_timed_out'

echo "--- 冷 ---"; cat /tmp/ub-cold.$$
echo "--- 热 ---"; cat /tmp/ub-hot.$$
rm -f /tmp/ub-cold.$$ /tmp/ub-hot.$$
```

为什么这么设：

- stats_noreset 全程使用，避免自己把计数读没；flush_stats 只在起点调用一次，用来定义测量窗口。
- flush_zone . 是可选的冷缓存起点（会清整份缓存），生产环境慎用；若不想清空，就改成”记录起始计数、用差值计算”，不要影响线上业务。
- 命中率按窗口算：hit / (hit + miss)。冷查三个新名字应几乎全是 miss，随后三轮同名字查询应几乎全是 hit，这个对比能直接证明缓存确实在工作。
- dig 的 Query time: N msec 是**单次、客户端侧**的观测值，样本少、抖动大，只用来看量级；真正稳定的递归延迟看 threadX.recursion.time.avg 与 recursion.time.median。官方手册说明 avg 只统计需要递归处理的查询（缓存命中的不计入），且因个别不响应服务器的离群值，avg 常大于 median，median 由直方图插值得出、代表 50% 用户查询的上限。
- num.prefetch（预取次数）、num.expired（过期缓存应答次数）、num.queries_timed_out 一起看，能区分”命中率高是因为缓存健康”还是”因为一直在回过期数据”。

基线要落在纸上：改动前后各跑一次，记录命中率、median 递归延迟、num.expired、num.queries_timed_out 四项。第 4 章的 num-threads / 缓存尺寸 / prefetch、第 5 章巡检脚本的 cmp -s、第 6 章的 uid 豁免，效果都应该能在这四项上看到——cmp -s 缺失时最典型的信号就是命中率周期性掉到接近 0（每分钟被 restart 清一次缓存）。

怎么验证：

```
unbound-baseline.sh /etc/unbound/filter.conf 192.0.2.1 53
unbound-control -c /etc/unbound/filter.conf stats_noreset | grep -E 'total.num.cachehits|total.num.cachemiss'
```

热查阶段的 cachehits 增量应约等于重复查询次数；若命中数不涨，先怀疑脚本里 dig 的目标地址 / 端口与 unbound-control -c 指向的不是同一个实例。

失败模式 / 回滚点：flush_zone . 在多实例链路上只清了一个实例的缓存，其他层仍返回旧答案——这正是 7.6 节的结论，测量时若发现冷查也全命中，先逐层确认。回滚：脚本只做读操作与一次可选的 flush，不改任何配置；删掉 flush_zone . 行即可让测量完全无副作用。

7.9　本章要点

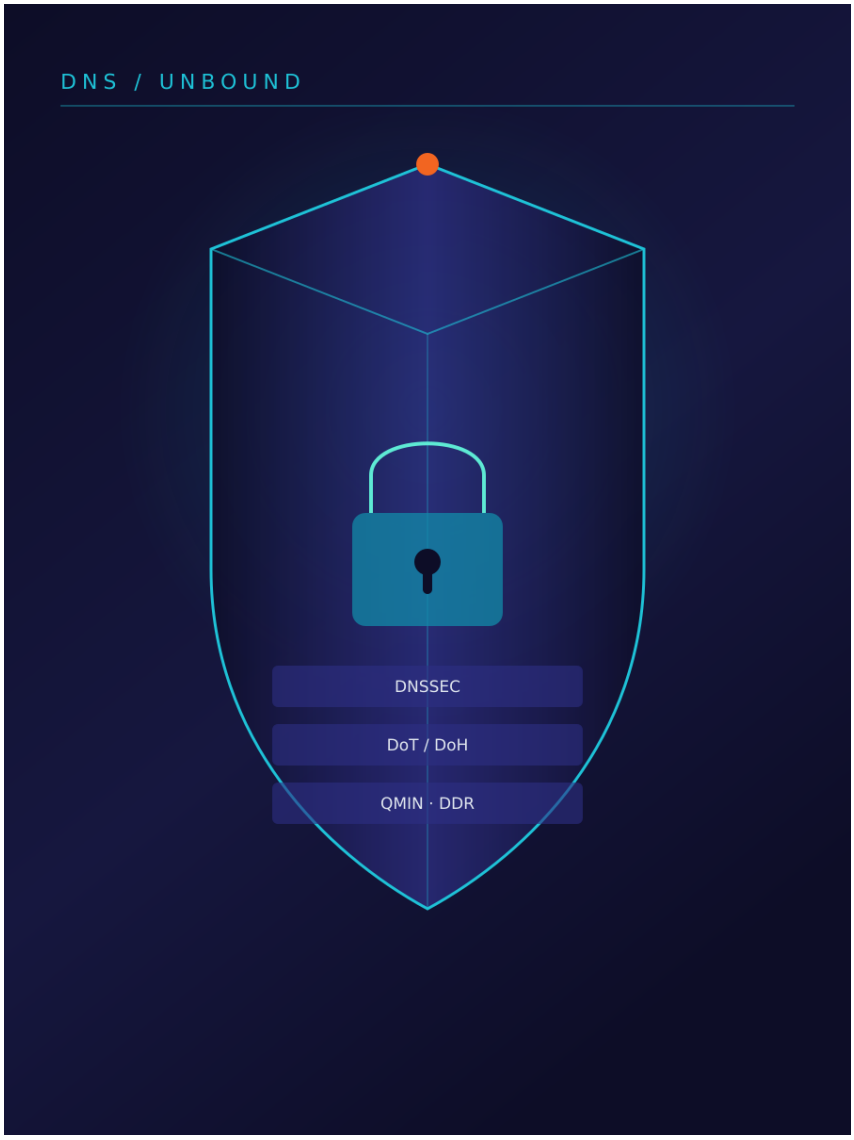
- 同一主机多实例要独立规划 interface@port、pidfile、remote-control 控制端口与证书、zonefile 目录；interface 变更只有 restart 生效，unbound-control -c < 配置 > 才能连到指定实例。
- 按域名分流可用多个 forward-zone（最具体区优先，每区仍只留一条 forward-addr），或靠不同 interface 地址让客户端落到不同实例；后者由 RA / DHCP / 策略路由决定，DNS 侧只负责监听面。
- local-zone 大表全量常驻内存，需与缓存尺寸、malloc 开销一起做基线；include 路径与根区 zonefile 都要同时满足 chroot 可见、unbound uid 可读写、AppArmor 白名单三项。

- DDR 压制用 `local-zone: "_dns.resolver.arpa." always_nodata`，Firefox canary 域 `use-application-dns.net` 回 NXDOMAIN；按 IP 封 443 是 denylist 不是证明。`private-address` 剔除公网应答里的私有地址防 DNS Rebinding，会误伤内网权威应答。dnstap 六类 `dnstap-log-*-messages` 全开才是全量查询流副本；保留期与到期删除必须显式配置，否则”抗元数据聚合”在自己门口破功。
- 不开 `unbound-control` 时刷缓存只能重启实例，且链路上每个实例都要刷（负缓存 NXDOMAIN、QMIN strict 兼容坑都属此类）；多实例部署建议开启但把控制接口限制在回环。
- `log-servfail: yes` 是排障首选，SERVFAIL 按”哪个实例→访问控制→日志原因→缓存旧条目→ nftables 出向豁免→ EDNS/MTU”顺序查；基线用 `stats_noreset` 记录改动前后的命中数、`recursion.time.median` 与 `num.expired`。

7.10 参考资料

- Unbound 官方手册, `unbound.conf(5)`（`interface`、`local-zone`、`private-address`、`include`、`remote-control`、`dnstap` 各节）：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html
- Unbound 官方手册，`unbound-control(8)` 命令与统计计数器（`stats_noreset`、`flush_zone`、`flush_bogus`、`lookup`、`cache_lookup`、`list_local_zones`）：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound-control.html
- Unbound 官方手册，Monitoring and Reporting（`extended-statistics`、`statistics-interval`、按需统计）：https://unbound.docs.nlnetlabs.nl/en/latest/topics/core/monitoring.html
- Unbound 官方手册,Performance Tuning(缓存内存比例、线程与 slabs):https://unbound.docs.nlnetlabs.nl/en/latest/topics/core/performance.html
- Unbound 官方手册，Tags and Views / RPZ 过滤主题页：https://unbound.docs.nlnetlabs.nl/en/latest/topics/filtering/rpz.html
- RFC 9462，*Discovery of Designated Resolvers (DDR)*：https://www.rfc-editor.org/rfc/rfc9462
- RFC 8020, *NXDOMAIN: There Really Is Nothing Underneath*（QMIN strict 兼容问题的背景）：https://www.rfc-editor.org/rfc/rfc8020

第三篇　加固



第八章

威胁模型与 NIST SP 800-81r3 基线

一次 DNS 查询从存根解析器出发，到权威服务器取回应答，中途会经过若干台你并不完全控制的机器。加固的起点不是「打开哪个开关」，而是先回答一个更朴素的问题：**这条查询，谁看得见？看见了什么？又能凭看到的做什么？**

NIST SP 800-81r3《Secure Domain Name System (DNS) Deployment Guide》给出的是一套部署基线。本章借它的骨架来组织威胁模型，但不把它读成一张打勾的合规清单。清单问的是「控制项齐没齐」，威胁模型问的是「在这条具体链路上，可见性落到了谁手里」。同一个控制，在「本地递归」和「把查询转交给上游」两种拓扑里，保护的可能是完全不同的人。落点始终是「**谁看得见什么**」。

8.1 为什么先谈可见性

DNS 报文里有两个东西格外值钱：一是**查询名**（你要访问谁），二是**客户端标识**（谁在问、什么时候问、从哪个地址问）。这两样合起来就是一份几乎不需要破解的浏览画像。明文 53 端口的年代，这份画像对链路上的每一跳都是敞开的自助餐；递归解析器自己因为要缓存、要过滤，也必然看得见全部查询名。

于是「加固」可以换一种说法：**把可见性从你不信任的位置，收缩到你信任的位置，并且让收缩后的那个位置无法滥用这份可见性**。三类控制——防护型 DNS、加密传输、解析验证——各自关掉的是不同观察者的一扇窗，谁也替不了谁。后面会反复回到这句话。

这一章只判断「该由哪一层负责」。各控制的参数、命令、验证与回滚分别落在第 9–14 章，RFC 的条文逻辑与协议演进在第四篇第 16、17 章，故障案例在第 22 章。

8.1.1 可见性不是一个开关

「看得见」不是有或没有，而是至少三种粒度叠在一起。把它们拆开，后面的矩阵才读得懂：

粒度	具体内容	谁最在意	哪类控制能收敛
内容可见	完整 QNAME、应答记录、EDNS 参数	画像者、过滤点	加密传输（链路上）、查询名最小化（上级）
关联可见	客户端地址、时序、查询频率、会话连续性	画像者、审计方	转发与集中端点（把多客户端收敛成一个地址）
留存可见	落盘后的日志、dnstap、备份与检索能力	日志留存方	保留期与访问授权（没有任何加密能替你做）

三者的关键差别：加密传输能把**内容可见**从链路上抹掉，却基本不动**关联可见**（端点 IP、报文时序、连接时长仍然可读），更完全不碰**留存可见**。于是「链路上已经抓不到 QNAME」并不等于「查询行为不再可追踪」——时序与流量形态仍能拼出画像。这也是为什么后面矩阵要按列拆，而不是只留一列「隐私」。

反过来，**内容可见**又恰恰是防护型 DNS 的工作前提。过滤点必须读到 QNAME 才能判断该不该拦；如果为了「隐私」把查询名也从过滤点收走，过滤就一起没了。可见性治理从来不是「越少越好」，而是「把该留的那一格留对地方」。

8.2　NIST SP 800-81r3 提供的框架

NIST SP 800-81r3《Secure Domain Name System (DNS) Deployment Guide》（第三版 / Revision 3，作者 Scott Rose、Cricket Liu、Ross Gibson）是面向「部署一套可信 DNS」的指南。它的价值在于把「安全 DNS」拆成可以逐层追问的结构：先列**资产**，再划**信任边界**，再枚举**攻击者能力**，最后才把结论落到具体部件上的**控制目标**。

按本章口径，只取这套结构，不取条目编号：

- **控制目标一律用名称表述**：「缩减暴露面」「区域数据完整性」「传输机密性」「查询元数据最小化」「上游可信」「失败关闭」等等；
- 本章**不引用任何具体控制项编号**，也不做审计项对表；编号、版本年份、修订记录与署名对应关系一律**见官方手册**（以 NIST 官方出版物页面为准），拿不准的地方不填数；
- 指南同时覆盖权威侧与解析侧两侧部署；这里只看**递归解析器 / 转发器**这一侧。

这不是把基线复述一遍。基线是「应当有什么」，威胁模型是「少了它，谁会多看见什么」。下面按资产、边界、能力三步走。

8.2.1　资产：要保护的四类对象

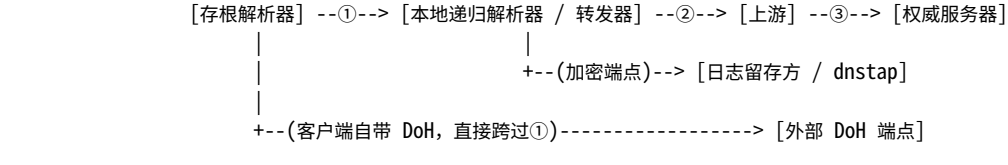
资产	它是什么	最怕什么	主要归哪一层管
区域数据（zone data）	权威侧的应答内容，以及递归侧缓存下来的副本	被改、被伪造、被塞进假记录	解析验证 / DNSSEC
解析器（resolver）本身	Unbound 进程、配置、缓存、监听地址	被绕过、被劫持、被当成开放解析器	最小暴露面与访问控制
查询流（query flow）	查询名、时序、客户端地址、EDNS 参数	被旁路复制、被聚合画像	加密传输 / 元数据最小化
日志（logs）	dnstap、查询日志、审计留存	无限期留存、越权读取	保留期与访问授权

四类资产里最容易看漏的是**日志**。查询流是「在路上」的资产，日志是「落在自己门口」的同一份资产的副本：把链路上的观察者请出去之后，如果自己在本地全量留存、又不设保留期，等于把散落的可见性收拢成一个更方便检索的集中观察者——「抗元数据聚合」在自己门口破功（集中案例见第 22 章）。资产清单里把它单列，正是为了后面把「日志留存方本身」也当成攻击者能力来建模。

区域数据还要注意「副本」二字。递归解析器并不持有权威数据，它持有一份带 TTL 的副本；缓存投毒打的不是权威服务器，正是这份副本。因此「数据完整性」在本书语境里，几乎总是指**验证器能不能识别手里这份副本被改过**，而不是「权威服务器有没有被攻破」。

8.2.2　信任边界：数据换手的三个界面

信任边界不是网线，是「这一段我认、下一段我不认」的那道线。查询每跨过一道边界，可见性的持有人就换一次。



① 存根 -> 递归 -> 权威：每跳看到的 QNAME 粒度与客户端信息不同
② 本地 -> 上游：把查询「托付」给谁，就等于把可见性交给谁
③ 加密端点 -> 日志留存方：链路上看不见了，端点与日志仍看得见

这张图怎么读：三条编号线是**数据换手处**，不是物理链路；竖着分出去的两条是**可见性的分流处**。①被绕过、或②被悄悄改了对象、或③被忽略，都会让「我以为只有我看得见」变成别的人也看得见。

三个界面各自对应一种失守方式：

- **存根→递归→权威**：存根解析器（stub resolver）通常只发一个递归查询、把全部判断交出去；递归解析器（recursive resolver）为了完成递归，会代客户端向权威服务器（authoritative server）逐层追问。查询名最小化（QNAME minimisation）能减少发给上级的标签数，但它**不是加密**：只抗元数据聚合，不抗全路径嗅探（实测口径，参数与兼容坑在第 13 章，RFC 演进在第 17 章）。

- **本地→上游**：这是最容易被「配置一个字段就悄悄改掉」的边界。给转发器（forwarder）多配一个 `forward-addr`，Unbound 是并发竞速取最快应答，而不是主备，查询会被送到你并不打算托付的解析器，本地过滤表随之失效；上游超时若允许回退明文 53，整条查询流又退回裸奔（`forward-first: no` 与 fail-closed 的落地在第 5、14 章）。
- **加密端点→日志留存方**：DoT / DoH / DoQ 把链路加密后，端点本身成为新的可见性集中点。加密**不消灭端点可见性**，它只是把观察者从「链路上任意一跳」收敛成「你选定的那个端点 + 它的日志系统」。所以「日志留存方」必须被当成边界内的一等角色来建模，而不是运维附属品。

```
客户端          192.0.2.10                ——实验环境内的存根解析器
本地解析器      192.0.2.1 / 2001:db8::1   ——自己控制的那一个端点
加密上游        198.51.100.1:853          —— resolver.example.net
权威服务器      203.0.113.53              —— dns.example.org
日志副本        本地 dnstap socket，保留期 30 天
```

为什么用文档段：地址本身就是公开示例，不会把真实拓扑带进这本 CC 许可的公开电子书。

怎么验证（一条可复制的命令）：`dig dns.example.org @192.0.2.1`。它只证明「解析通不通」，**证明不了链路上谁看得见**；可见性得靠抓包或 dnstap 去观察，二者不是一回事——这正是本章反复强调的区分。

失败模式 / 回滚点：若把示例地址误当生产地址写进配置，转发目标不可达，两波都失败即 SERVFAIL，回到 `forward-first: no` 的 fail-closed 语义（第 14 章）；此时应回滚到单上游、单一加密端点，而不是再加一个「备用」上游去竞速。

明文时，`192.0.2.10 → 192.0.2.1` 这一段上的被动观察者能读到完整 QNAME 与客户端地址；改走 `198.51.100.1:853` 后，同一观察者只剩端点 IP、端口与报文时序。可见性没有消失，只是换了持有人——这正是下一节矩阵要摊开讲的事。

8.2.3 攻击者能力清单

把「攻击者」拆成能力而不是身份，才对得上前面那三道边界。下面七项是本书后续所有加固动作的假想敌；它们并不互斥，真实故障往往是两三项叠加。

能力	站在哪道边界	能看见 / 能做到什么	对应被突破的资产
被动嗅探（passive sniffing）	存根→递归、本地→上游的链路上	读 QNAME、时序、客户端地址，长期聚合画像	查询流
链路注入（on-path injection）	同一链路，但可写	抢答、改应答、伪造源地址	区域数据副本、查询流
缓存投毒（cache poisoning）	递归解析器的缓存	塞入伪造记录，污染后续所有命中	区域数据副本
权威劫持（authoritative hijack）	递归→权威	换掉应答来源，返回「看起来正常」的假数据	区域数据
上游被控	本地→上游	你主动托付出去的那一端反过来骗你	查询流、区域数据副本
客户端绕过	存根这一侧	应用自带 DoH / 自动发现加密 DNS，跨过本地解析器	解析器、查询流、策略
日志留存方本身	加密端点→日志留存方	端点内部全量留存、越权查询历史	日志

三点值得单独拎出来：

1. 「**上游被控**」**最难自查**，因为查询是你自己主动送过去的。转发器与验证器在这里二选一：把根区交给上游的转发器上再配信任锚（trust anchor），锚需要自己向根 priming DNSKEY，而 `forward-zone "."` 又把所有查询都交了出去，结果是日志持续报失败、间歇性全量 SERVFAIL（实测结论，配置取舍见第 10 章）。信任托付得越彻底，越不能假装自己还在验证。
2. 「**客户端绕过**」**会让防护型 DNS 整体失效**，而不是削弱。策略过滤的前提是「过滤点看得见查询名」；浏览器一旦直连外部 DoH 端点，本地解析器连查询都收不到。第 13 章的 DDR 压制与浏览器侧 canary 域（回 NXDOMAIN 即表示「别启用自带 DoH」，具体域名见官方手册）处理的正是这件事，比按 IP 封 443 更准——封 IP 是 denylist，不是证明。
3. 「**日志留存方本身**」**是加密之后才凸显的攻击者**。加密把链路观察者清掉，却没有给日志系统加任何锁；若全量 dnstap 不设保留期，攻击者不需要碰链路，只要能读日志就够了。可见性总量没变，只是搬了家。

还有一项能力容易被归错类：域外夹带的胶水记录（glue record）是经典缓存投毒面，它与 DNSSEC 签名无关，任何验证模式下都该单独挡（`harden-glue` 的开启在第 9、10 章）。把它算进「缓存投毒」这一行，而不是「权威劫持」，才不会误以为开了验证就顺带解决了它。

两项能力叠加时，最常出现两种误判：

- 「**被动嗅探 ＋ 客户端绕过**」：本地抓包看不到明文查询，便以为过滤仍在工作。实际上查询根本没经过本地解析器，是走加密通道直连外部端点；链路安静不是因为控制生效，而是因为流量换了路。判断依据应当是本地解析器有没有收到查询，而不是链路上有没有 53 端口的报文。
- 「**上游被控 ＋ 静默明文降级**」：两项分别由不同章节处理，但叠加后最难发现——上游本来就是主动托付的对象，超时后再退回明文 53，等于把被控上游的可见性又向沿途每一跳开放了一遍。第 14 章把失败路径单独拎出来做 fail-closed 工程，原因就在这里。

所以能力清单不是七个互不相干的格子。建模时至少要问一句「哪两项会一起发生」，否则容易把「某一项已被挡住」误当成「这类风险已消失」。

8.3　可见性矩阵：谁看得见什么

把三道边界和七项能力压到一张表里，就是本章要留下的那张图。表中「是 / 部分 / 否」指**默认明文 53、且没有额外控制时**的可见性，不是「永远如此」；后面三类控制会逐格把「是」改成「否」。

观察者 / 位置	完整 QNAME	关联客户端	读应答	改应答	长期留存
同链路被动嗅探者	是	是	是	否	可自行留存
同链路注入者（可写）	是	是	是	是（抢答）	可自行留存
本地递归解析器（自己）	是	是	是	技术上可以	取决于日志策略
上游转发目标	是	部分（只见本端地址）	是	技术上可以	取决于对方
权威服务器	部分（受 QMIN 收敛）	部分	是	可伪造	取决于对方
日志留存方（dnstap / 审计）	是	是	是	否	是（若不设保留期）
外部 DoH 端点（客户端绕过时）	是	是	是	技术上可以	取决于对方

「客户端绕过」的特殊性不在多了最后一行，而在「**本地递归解析器**」**那一行会整行变空**：本地解析器连查询都收不到，过滤表、缓存、审计一起空转。可见性没有被消灭，只是从第 3 行搬到了第 7 行，搬到了你不控制的那一端。这是「防护型 DNS 与隐私控制的张力」在矩阵上的样子。

再按列看，哪一列由哪一层负责，一目了然：

- 「**完整 QNAME**」**这一列**：加密传输把第 1、2 行清掉，查询名最小化把第 5 行收敛成「部分」；两者都**不动**第 3、6、7 行。端点自己、日志系统、外部端点依然看得见全部内容。
- 「**改应答**」**这一列**：只有解析验证（DNSSEC）能把「是」变成「改了会被发现」。加密传输只挡得住第 2 行的链路注入，挡不住第 4、5、7 行的端点本身作恶。
- 「**长期留存**」**这一列**：**没有任何加密或验证控制能清掉第 6 行**，只有保留期与访问授权能。这一格最容易被「我们已经全链路加密了」的错觉盖过去。

所以「加固完成」不是「表上全是否」——第 3 行那份可见性是防护型 DNS 得以工作的前提，必须保留，只是要锁在你能审计、能设保留期的那个端点上。目标是把可见性**收敛并关进笼子**，不是让它归零。这一句，就是本章与合规清单最根本的分歧。

8.4　三类控制的纵深关系

把基线里的控制按「关掉哪一扇窗」分成三层，比按产品功能分更不容易漏项。三者是**叠加**关系，不是**任选**关系。

8.4.1 防护型 DNS：先决定「让不让查」

防护型 DNS（protective DNS）指在解析路径上按策略过滤：拦截已知恶意域名、拒绝不该服务的来源、限制谁能用这台解析器、给应答打上本地策略。它回答的问题是「**这个查询该不该被回答**」。

关键在于：它**需要看得见查询名**才能工作。过滤点必须坐在矩阵第 3 行那个「本地递归解析器」上，且必须真的收到查询。于是它和隐私控制天然有张力——隐私要把查询名从链路上收走，过滤却要在某处把它摊开。二者不矛盾的唯一前提是：**摊开查询名的那一处，正是你信任并控制的那一处**。客户端一旦绕过，过滤表不是变弱，是整行空转。

对应章节：最小暴露面、监听地址收敛、访问控制（第 9 章）；DDR 压制与绕过检测（第 13 章）。

8.4.2 加密传输：把链路上的观察者请出去

加密传输（DoT / DoH / DoQ）做的是**传输加密与链路隐私**：把查询从明文 53 移进加密通道，顺带用证书认证「我连的是不是我要连的那个端点」。它回答的问题是「**链路上谁能看见、谁能抢答**」。

它清掉的是矩阵第 1、2 行（被动嗅探、链路注入），以及「本地→上游」这一跳的沿途观察者。它**不清掉**第 3、4、6、7 行：端点自己、上游自己、日志系统、外部端点仍然看得见全部内容。所以「上了 DoH 就隐私了」是错的——更准确的说法是「把观察者名单从『任意一跳』缩短为『我选定的端点及其日志方』」。

还有一个容易反噬的边界：加密通道如果允许失败后回退明文 53，前面清掉的第 1、2 行会原样回来，而且往往静默发生。`forward-first: no` 与 fail-closed 的取舍就是把这条退路堵死（第 5、14 章）。审计时先问失败路径，再问正常路径，原因就在这里。

8.4.3 解析验证：确认「答案是不是原样的」

解析验证（DNSSEC validation）由验证器（validator）从信任锚（trust anchor）出发逐级验签，做的是**数据完整性与来源认证**：应答有没有被改过、是不是这个区域真正签发的。它回答的问题是「**这份数据我能不能信**」。

它针对的是矩阵里「改应答」那一列：缓存投毒、权威劫持、上游被控，以及端点本身返回的假数据。这些恰恰是加密传输**管不到**的——加密只保证「路上没人动」，不保证「出发时就是对的」。反过来，DNSSEC 也一个字节都不隐藏：签过名的应答仍是明文可读的完整 QNAME，链路上照样能画像。

验证还有状态之分：没有信任锚就是透传不验证（坏签名域照样返回答案），permissive 只记录日志仍放行，真验证才会把坏签名变成 SERVFAIL、给好签名打 AD 标志；「有锚却主动允许降级」这类设置会自己制造漏洞。三种状态与相关开关在第 10 章。另外，`harden-below-nxdomain` 只对 DNSSEC-secure 的 NXDOMAIN 生效，没有验证器时它是空转——别把「开关已打开」误判成「控制已生效」（见官方手册对应 NOTE，配置在第 10 章）。

8.4.4 三者如何叠加而不互相替代

控制层	回答的问题	关掉的观察者 / 能力	明确管不到的
防护型 DNS	该不该回答	客户端绕过、未授权来源	链路可见性、应答真伪
加密传输	链路上谁看得见	被动嗅探、链路注入	端点作恶、日志留存、数据伪造
解析验证	数据能不能信	缓存投毒、权威劫持、上游被控	元数据画像、日志留存

「纵深」的意思在这张表里很清楚：三行的**最后一列互不重叠**。少任何一层，都对应着一类攻击者能力完全没人接管——不是「防护变薄」，是「那一格是空的」。而「不互相替代」也在这里落地：拿加密传输去挡缓存投毒，或拿 DNSSEC 去挡被动嗅探，都是把控制放错了列。

8.5 一条不能含糊的分界线

全书只认这一组定义，任何章节都不许混用：

> **DNSSEC = 数据完整性 + 来源认证**（应答没被改、且确实由该区域签发）。**DoT / DoH / DoQ = 传输加密 + 链路隐私**（路上不可读、不可抢答、端点可认证）。**二者不互相替代**：前者不隐藏一个字节，后者不校验一个签名。

两条常见的错认，正好对应两种坏结果：

- 「**我们已经上了 DoH，DNSSEC 可以省了**」：链路确实干净了，但上游被控、权威劫持、缓存投毒这三项全都还在。加密通道会把一份伪造应答完好无损、加密地送到客户端手里——「路上没人动」反而让伪造更难被察觉。
- 「**我们开了验证，就不用管传输了**」：应答确实可信了，但完整 QNAME、客户端地址、时序在每一跳明文可读，查询流与日志两列一个「是」都没少。签名不提供保密性。

判断某个新机制属于哪一类，只需问一句：**它改的是矩阵里「改应答」那一列，还是「完整 QNAME / 关联客户端」那一列？**改前者的是验证族，改后者的是加密族，两列都改的目前没有。查询名最小化常被误当成加密，就是因为它只碰「权威服务器那一行的 QNAME 粒度」，既不给链路上锁，也不给应答签名（协议条文见第四篇第 17 章）。

一句话收束：加密传输决定**谁看不见**，解析验证决定**看见了也改不了**，防护型 DNS 决定**该不该让它被看见**。三句话里没有一句能由另外两句推出来。

8.6　控制目标→本书章节

下表把基线里的**控制目标名称**（不带编号）对到第三篇后续各章。左列是「要达成什么」，中间两列刻意保留威胁模型口径——你要能说清这个目标**把可见性从谁手里拿走、又挡住哪项能力**，否则就只是在抄清单。具体控制项编号、审计映射与版本对应关系**见官方手册**。

控制目标（名称，无编号）	收敛的可见性	主要挡住的能力	本书章节	该章标题
缩减暴露面与访问授权	谁能向解析器提问	未授权使用、被当开放解析器	第 9 章	最小暴露面与访问控制
区域数据完整性与来源认证	谁能改应答而不被发现	缓存投毒、权威劫持、上游被控	第 10 章	DNSSEC 验证：信任锚、放行与拒绝
传输机密性与服务端点认证（DoT）	存根→递归、本地→上游链路	被动嗅探、链路注入	第 11 章	DoT：端口 853 的服务端与客户端
加密查询的承载与共存（DoH）	与 Web 共端口时的链路	被动嗅探、按端口封锁	第 12 章	DoH：与 Web 服务共存的加密解析
查询元数据最小化与防绕过	上级看到的标签数、客户端是否绕过	元数据聚合、客户端绕过	第 13 章	查询隐私：QMIN、防降级与 DDR 压制
上游可信与失败关闭	本地→上游的托付对象、失败后的退路	上游被控、静默明文降级	第 14 章	加密通道选路与 fail-closed 工程

三点读表提示：

1. **第 9、13 章看起来「不像安全章」，其实管的是可见性的入口**：一个决定谁能进来问，一个决定客户端会不会干脆不进来。少了它们，第 10–12 章的控制连流量都碰不到。
2. **第 10 章单独占住「数据完整性」这一列**，不与第 11、12 章合并，正是那条分界线的编排结果：验证与加密在本书里永远分章，参数也互不借用。
3. **第 14 章是唯一处理「失败之后」的一章**。威胁模型里最阴的往往是降级路径——控制都在，但超时之后悄悄退回明文 53，等于把整张矩阵复原。单上游、fail-closed、两波失败即 SERVFAIL，说的都是同一件事：不给可见性留一条偷偷搬家的退路。

表中没有「合规通过」这一列。一个目标是否达成，判断标准不是「配置项写没写」，而是回到矩阵问：这一格现在的持有人，是不是我信任、能审计、且留不久的那一个。

8.7　把基线读成问题，而不是清单

合规清单的读法是「逐项确认已配置」，威胁模型的读法是「对每一跳追问四个问题」。合上本章前，不妨拿自己那台解析器过一遍：

1. **这条查询现在落在矩阵的第几行？**也就是：完整 QNAME 此刻在谁手里。
2. **我信任那个持有人吗？**不信任却把查询托付过去（多一个 `forward-addr`、外部 DoH、上游递归），就是「上游被控」的入口。
3. **它能不能改应答？**能改的话，是 DNSSEC 在拦，还是我误以为加密传输会拦。

4. **失败之后会退到哪里?** 退到明文 53、退到第二个竞速上游、退到客户端自带解析器，都意味着可见性搬了家而不是消失。

再补两问，专门针对最容易被漏掉的两格：**日志留存方是谁、保留多久**（第 6 行），以及**客户端有没有办法绕开我**（第 3 行会不会整行变空）。这两格没有任何加密或验证控制会替你关掉，只能靠保留期、访问授权与绕过检测。

跳转	QNAME 此刻在谁手里	我信任它吗	它能改应答吗	失败后退到哪里
`192.0.2.10` → `192.0.2.1`	存根与本地解析器	是（自己控制）	技术上可以，靠本地审计约束	客户端自带 DoH（须压制）
`192.0.2.1` → `198.51.100.1:853`	加密上游端点	取决于托付关系	可以，靠解析验证兜底	明文 53（须堵死）
递归→ `203.0.113.53`	权威服务器	不一定	可以，靠签名识别	换上游（须单上游）
本地→ dnstap	日志留存方	是，但需限权	否	无限期留存（须设保留期）

四行里没有一行是「没人看得见」。第一行是主动保留的可见性，第二、三行靠验证与单上游约束，第四行靠保留期约束。差别只在「持有人是谁、能留多久」——这就是把清单读成问题之后，剩下的全部工作。

于是整章可以压成一句：**加固不是把「谁看得见什么」改成「没人看得见」，而是改成「只有我信任、且我能审计的那一个端点看得见，并且它看见了也改不了、留不久、别人绕不过去」**。第 9–14 章各把这句话里的一个分句落到配置与验证命令上。

8.8 本章要点

- 本章借 NIST SP 800-81r3 的「资产→信任边界→攻击者能力→控制目标」骨架建模，只读控制目标名称、不引用控制项编号；版本年份、编号与审计映射见官方手册。
- 四类资产（区域数据、解析器、查询流、日志）里最易漏的是日志：它是查询流在自己门口的副本，加密之后会变成新的集中观察者。
- 三道信任边界（存根→递归→权威、本地→上游、加密端点→日志留存方）各对应一种失守方式；可见性每跨一道边界就换一次持有人。
- 防护型 DNS、加密传输、解析验证是叠加而非任选：分别回答「该不该回答」「链路上谁看得见」「数据能不能信」，三者管不到的那一格互不重叠。
- DNSSEC = 数据完整性与来源认证，DoT / DoH / DoQ = 传输加密与链路隐私，二者不互相替代；加密不校验签名，签名不隐藏字节。
- 落点是「谁看得见什么」：目标不是让可见性归零，而是把它收敛到可信任、可审计、有保留期、且客户端绕不过的那一个端点。

8.9 参考资料

- NIST SP 800-81r3《Secure Domain Name System (DNS) Deployment Guide》官方出版物页面：https://csrc.nist.gov/pubs/sp/800/81/r3/final
- NIST CSRC 出版物总目（版本年份、修订记录与署名以此为准）：https://csrc.nist.gov/publications
- RFC 4033, *DNS Security Introduction and Requirements*: https://www.rfc-editor.org/rfc/rfc4033
- RFC 7858, *Specification for DNS over Transport Layer Security (TLS)*: https://www.rfc-editor.org/rfc/rfc7858
- RFC 8484, *DNS Queries over HTTPS (DoH)*: https://www.rfc-editor.org/rfc/rfc8484
- RFC 9250, *DNS over Dedicated QUIC Connections*: https://www.rfc-editor.org/rfc/rfc9250
- RFC 5737, *IPv4 Address Blocks Reserved for Documentation*: https://www.rfc-editor.org/rfc/rfc5737
- NLnet Labs Unbound 官方文档（各控制的参数与验证以此为准）：https://unbound.docs.nlnetlabs.nl/en/latest/

第九章

最小暴露面与访问控制

`interface: 0.0.0.0` 配一行 `access-control: 0.0.0.0/0 allow`，能让全网立刻问通，也能让这台机器几小时内变成别人的反射放大器。前者是「能用」，后者是「被用」。本章处理两者之间那道边界：谁可以问、从哪个地址问、能问出多大的应答、问完留下多少痕迹。

DNSSEC 验证在第 10 章，加密传输（DoT / DoH）在第 11、12 章，QMIN 与 DDR 压制在第 13 章，nftables 的 `:53` 劫持与按 uid 豁免在第 14 章。

9.1 监听地址逐条列出，不要 `0.0.0.0`

`interface` 决定在哪些本地地址上收包。`0.0.0.0` 与 `::` 的含义是「本机所有地址，包括以后新加的地址」：今天多一个网卡、明天多一条隧道，解析器就自动多一张嘴。

```
# /etc/unbound/unbound.conf.d/10-exposure.conf
server:
  interface: 127.0.0.1
  interface: ::1
  interface: 192.0.2.1
  interface: 2001:db8::1
  port: 53
  do-ip4: yes
  do-ip6: yes
  do-udp: yes
  do-tcp: yes
```

为什么这么设：`interface` 可重复给多条，官方手册说明「一条都不给时默认只监听 localhost」。回环留给本机诊断与 `unbound-control`，另两个地址是面向客户端的网卡地址，不写通配。`do-tcp: yes` 必须保留：UDP 是常态，TCP 是大应答与重试的退路，为「少开一个面」关掉 TCP 只会把故障推到超时。`@port` 后缀用于同机多实例（第 7 章）。

怎么验证：`ss -lunp 'sport = :53' ; ss -lntp 'sport = :53'` ——只应看到上述四个地址，不应出现 `0.0.0.0:53` 或 `[::]:53`。

失败模式 / 回滚点：官方手册明确写了 `interface` 的变更在 `kill -HUP` reload 时**不生效**、只有 restart 生效，所以「改了配置但 `ss` 没变」先怀疑 reload。地址还没配到网卡上就 restart 会绑定失败。回滚：保留上一版副本，删掉新增行并 `systemctl restart unbound`；回环仍可用，缓存重建是可接受代价。

9.2 `access-control`：默认拒绝、最长前缀匹配、与顺序无关

`interface` 管「收不收这个包」，`access-control` 管「收到之后答不答」，两者不是一回事。

```
# /etc/unbound/unbound.conf.d/20-access.conf
server:
  access-control: 127.0.0.0/8 allow
  access-control: ::1/128 allow
  access-control: 192.0.2.0/24 allow
```

```
  access-control: 192.0.2.9/32 deny
  access-control: 2001:db8::/64 allow
  access-control: 0.0.0.0/0 refuse
  access-control: ::/0 refuse
```

为什么这么设：官方手册原文是「使用最具体的网段匹配，若都不匹配则 refuse，因此语句顺序无关紧要」。三条要一起记：① **默认拒绝**，不写 `0.0.0.0/0 refuse` 也生效；② **最长前缀匹配**，`192.0.2.9/32` 比 `192.0.2.0/24` 更具体，因此客户端网段里单独这一台被静默丢弃，哪怕它被写在 allow 段之后也一样；③ **顺序无关**，allow 写在最后不会被前面的 deny 覆盖，这与从 iptables 带来的直觉相反。两条兜底 `refuse` 只是把意图写给人看。`refuse` 回 REFUSED、`deny` 静默丢包；手册提醒 DNS 协议不擅长处理按策略丢弃，丢包会引发重试，故兜底用 `refuse`。`allow` 只给递归查询，非递归查询仅能取本地 `local-data`，动态缓存不会被顺手挖走。

怎么验证：`dig -b 192.0.2.1 @192.0.2.1 example.org A +short ; dig @192.0.2.1 example.org A +norecurse`——前者有答案且带 `ra`，后者应 REFUSED；从 `203.0.113.9` 这类未放行源问也应 REFUSED。用 `-b` 指定源地址很关键，否则测到的是回环放行规则。

失败模式 / 回滚点：客户端拿到 REFUSED 而端口正常，问题一定在 `access-control`；最隐蔽的错是把 allow 段写成 `/32` 却以为写的是 `/24`。回滚：删掉误加的 `deny` 行后 `unbound-control reload`（`access-control` 支持 reload，`interface` 不支持）。

9.3　不对公网开放递归：放大倍数从哪来

一台对任意源地址都提供递归的解析器叫**开放解析器（open resolver）**。它被别人当免费 DNS 用，也被当反射器打第三方。

机理要讲清「倍数」的来历。DNS over UDP 没有握手、源地址可伪造：攻击者把查询包的源 IP 写成受害者地址发往开放解析器。查询包很小——一个 QNAME 加固定字段，量级几十字节；而解析器要做完整递归并回一个带多条记录、可能还含 authority / additional 段的应答，量级几百到上千字节。**放大倍数就是这两个字节数相除**：60 字节查询换 1200 字节应答，约 20 倍。这不是实测结论，只是把两个字节数相除的示意；真实倍数取决于查询名、记录类型与应答是否被截断，具体数值见官方手册与 NIST SP 800-81r3。

环节	控制	本章位置
谁能触发递归	`interface` + `access-control` 默认拒绝	9.1 / 9.2
单个应答能有多大	`minimal-responses`、`max-udp-size: 1232`	9.7
单位时间能触发多少	`ratelimit` / `ip-ratelimit`	9.7
包能不能进内核	防火墙只放行客户端网段	9.9

`max-udp-size` 管**入向应答上限**，`edns-buffer-size` 管**出向通告**，两者默认 1232（DNS Flag Day 2020 建议）；参数细节在第 4 章，这里只取「压低放大倍数」这层含义。防火墙是最后一道：即使某天误配成 `0.0.0.0/0 allow`，包过滤仍挡得住公网源。

怎么验证：在 `203.0.113.9` 上执行 `dig @192.0.2.1 example.net A +time=2 +tries=1`，期望 `status: REFUSED` 或超时，绝不能是 `NOERROR` 带答案。这是整章最重要的一条断言。

失败模式 / 回滚点：若返回答案，说明有更宽的 allow 段或测到了另一个实例。回滚：撤掉最宽的 allow 段、reload，再逐段放回；不要靠「关掉防火墙」排障，那会把暴露面直接打开。相关 BCP 编号与年份见官方手册，不在这里编造。

9.4　降权、chroot 与 AppArmor 路径白名单

`username` 在绑定 53 端口后降权，`chroot` 把文件视图限制到一棵子树，AppArmor 再从 LSM 层限制可读路径。三层叠加，最容易踩的是路径。

```
# /etc/unbound/unbound.conf.d/30-privilege.conf
server:
  username: unbound
  chroot: ""
  directory: "/etc/unbound"
  hide-identity: yes
  hide-version: yes
```

```
  identity: ""
  version: ""
```

为什么这么设：username 默认就是 unbound，显式写出是给审计与第 14 章按 uid 做 nftables 豁免留下依据；username: "" 只用于排障。chroot: "" 是**明确选择暂不启用**——启用后所有相对路径都在 chroot 内解释，include、pidfile、logfile、密钥与根区 zonefile 要整体重排，还得准备 dev/urandom 与 dev/log 绑定挂载，AppArmor 白名单也要跟着改；半开半关是启动即失败的经典原因。hide-identity / hide-version 默认均为 no：identity: "" 会回主机名、version: "" 会回真实版本串，打开后 id.server / hostname.bind / version.server / version.bind 这些 CHAOS 查询回 REFUSED。这是**减少信息暴露，不是访问控制**。

怎么验证：dig @127.0.0.1 version.server TXT CHAOS +short ; ps -o user=,pid=,comm= -C unbound ——前者为空或 REFUSED，后者显示 unbound 而非 root。

失败模式 / 回滚点：返回版本串通常是后面的 include 把 hide-version 覆盖回 no，或 reload 了错的实例。回滚：chroot 恢复空串并 restart，降权与信息隐藏可独立保留。

```
# /etc/apparmor.d/local/usr.sbin.unbound ——路径白名单示意，按本机 profile 合并
  /etc/unbound/               r,
  /etc/unbound/**             r,
  /run/unbound/**             rw,
  # 注意：没有 /var/lib/unbound/** 的授权
```

为什么这么设：unbound-checkconf 只查语法与文件路径，不查 LSM 是否放行。把过滤表、include 片段或运行时数据挪到 /var/lib/unbound/ 下而 profile 没授权，现象是启动失败、或 checkconf 通过但配置根本没加载；日志里只有 DENIED，不会有一句「路径不允许」。第 7 章过滤表大表与第 6 章根区 zonefile 落的是同一个坑。

怎么验证：aa-status 2>/dev/null | grep -i unbound ; journalctl -k --since '10 min ago' | grep -i DENIED | tail -5

失败模式 / 回滚点：出现 DENIED 就把文件放回 /etc/unbound/ 既有目录，或按实际路径补 r，后 apparmor_parser -r 重载。回滚：注释掉新增 include 行并 restart，代价是那部分配置不生效，要同步告知使用方。SELinux 机制相同、工具不同，见官方手册对应章节。

9.5 private-address: 防 DNS rebinding

DNS rebinding 让浏览器先解析到攻击者控制的公网地址，再把同一名字的应答改成你的内网地址；浏览器同源策略按域名判断，页面脚本就成了进入内网的代理。

```
# /etc/unbound/unbound.conf.d/40-private.conf
server:
  private-address: 10.0.0.0/8
  private-address: 172.16.0.0/12
  private-address: 192.168.0.0/16
  private-address: 169.254.0.0/16
  private-address: fd00::/8
  private-address: fe80::/10
  private-address: ::ffff:0:0/96
  private-domain: "home.example.net."
```

为什么这么设：官方手册说明该选项把指定网段从**公网域名的应答**里剔除，覆盖 A / AAAA / SVCB / HTTPS，连 SVCB / HTTPS 里的 ipv4hint / ipv6hint 参数也检查；默认不启用任何网段。::ffff:0:0/96 堵住 IPv4-mapped IPv6 绕过过滤的写法，private-domain 给确实要返回内网地址的域名开白名单（自己配置的 local-data 默认允许）。手册提示把 127.0.0.0/8 加进去会误伤以 127.* 作应答码的垃圾邮件黑名单；启用 DNSSEC 验证时含私有地址的应答可能被标为 bogus（第 10 章），需用 private-domain 或本地数据例外处理。

怎么验证：unbound-control get_option private-address ; unbound-control list_local_zones | head ——确认运行中的实例确实加载了这些网段，而不是只写在文件里。

失败模式 / 回滚点：最典型的误判是「内网服务突然解析不到地址」，因为自己的权威应答也返回 192.0.2.x，被一并剔除。回滚：给该域名加 private-domain，或逐条注释网段后 reload；不要整段关掉。

9.6　local-zone 与 [RFC 6761] 特殊用途域

local-zone 让解析器在本地给出权威式应答，查询根本不出网；既省流量，也防内网信息泄漏到公网权威服务器。

```
# /etc/unbound/unbound.conf.d/50-special-zones.conf
server:
  local-zone: "test." always_nxdomain
  local-zone: "invalid." always_nxdomain
  local-zone: "example." always_nxdomain
  local-zone: "onion." always_nxdomain
```

为什么这么设：[RFC 6761] 把 test.、invalid.、localhost.、example. 列为特殊用途域名，本就不该去公网查。官方手册列出的默认区包括 localhost.、回环反向区、home.arpa.、resolver.arpa.、service.arpa.、onion.、test.、invalid. 及 AS112 反向区；onion. 依据 [RFC 7686]，home.arpa. 依据 [RFC 8375]，具体清单以所用版本的官方手册为准。这里用 always_nxdomain 覆盖默认 static：官方定义是「像 static，但忽略 local data 并返回 NXDOMAIN」，好处是日后有人往该区塞 local-data 也绝不出网，代价是丢掉默认 SOA / NS 内容，依赖它的客户端应改回 static。要连应答都不给就用 always_deny。刻意**不写** example.net. / example.org.：验收命令要用这两个名字做真实解析，写进去等于挡住自己的测试；home.arpa. 与 resolver.arpa. 留给默认区，后者的 DDR 自动发现压制方法在第 13 章。

怎么验证：dig @127.0.0.1 whatever.test. A ; dig @127.0.0.1 whatever.invalid. A ; dig @127.0.0.1 example.org A +short ——前两条 NXDOMAIN 且不产生出向递归，第三条仍有正常答案，证明没有误伤。

失败模式 / 回滚点：过度压制会挡掉合法内部命名（例如真用了 *.test 做测试域）。回滚：把对应行改成 nodefault 或删掉再 reload，过滤失效是预期代价。

9.7　限流与丢弃异常查询

先纠正高频拼写错误：官方选项名是 **ratelimit**（无连字符）与 **ip-ratelimit**，写成 rate-limit 会直接报 unknown option。两者管的不是同一件事：

选项	统计维度	超限动作	默认
ratelimit	发给某目标域的递归查询 qps	回 SERVFAIL；**缓存命中不受限**	0（关闭）
ip-ratelimit	每客户端 IP 接受的查询 qps	**完全丢弃**，连 SERVFAIL 都没有	0（关闭）
ip-ratelimit-cookie	带有效 DNS Cookie 的客户端	同上，手册建议约为前者十倍	0（关闭）
unwanted-reply-threshold	每线程累计「不该来的应答」数	告警并清空 rrset / message 缓存	0（关闭）

```
# /etc/unbound/unbound.conf.d/60-ratelimit.conf ——数值需按自己的 qps 基线调
server:
  ratelimit: 1000
  ratelimit-factor: 10
  ip-ratelimit: 50
  ip-ratelimit-cookie: 500
  unwanted-reply-threshold: 10000000
  do-not-query-localhost: yes
  tcp-connection-limit: 192.0.2.0/24 50
  tcp-connection-limit: 2001:db8::/64 50
  deny-any: yes
  harden-short-bufsize: yes
  harden-large-queries: yes
  minimal-responses: yes
  max-udp-size: 1232
  edns-buffer-size: 1232
```

为什么这么设：ratelimit 挡随机子域洪泛（自身不被压垮、也不把流量灌向受害权威域），ratelimit-factor: 10 表示超限后仍放行约 1/10 真实查询以便缓存恢复；手册明确写了配置的转发器不受此限，且这两组选项标为 **experimental**，上面数字只是起点，须按 stats_noreset 基线调，不要照抄当结论。ip-ratelimit 在查缓存**之前**生效，手册说明用途

正是缓解放大攻击；带有效 DNS Cookie 的客户端会绕过它，需限制时用 `ip-ratelimit-cookie`。`unwanted-reply-threshold` 是投毒防线：收到大量并非本实例发出的应答就清缓存（手册建议量级为千万），代价是误触发时命中率掉一截。`do-not-query-localhost: yes` 默认开启，把回环与零网段排除在出向目标外，避免不必要的 unwanted reply 顶高计数。

后半段是「丢弃异常查询」：`deny-any: yes` 对 ANY 查询给空应答，不再触发可能很长的上游应答；`harden-short-bufsize: yes` 忽略异常小的 EDNS 缓冲区声明（默认已是 yes）；`harden-large-queries: yes` 忽略超大查询（默认 **no**，合法的大 TSIG / EDNS 载荷确实存在，开启前确认自己没有这类客户端）；`minimal-responses: yes` 不塞非必需的 authority / additional 段，直接压低 9.3 的放大倍数；两个缓冲区保持 1232，不在本章重复第 4 章的 MTU 分析。`tcp-connection-limit` 限制单网段并发 TCP 连接，同样是实验选项。

怎么验证：`unbound-control stats_noreset | grep -E 'num.queries|num.queries_cookie|num.queries_discard_timeout|num.cacheh` ——先记空闲基线，再短时并发打一批不同名字，看丢弃类计数是否增长、命中率是否不合比例下降。

失败模式 / 回滚点：`ip-ratelimit` 设得低于正常突发量，表现是「偶尔整批超时且日志里什么都没有」——超限是静默丢包，最易误判成网络抖动。回滚：几个值改回 0 并 `unbound-control reload`（手册把 `ratelimit` / `ip-ratelimit` 列为可 reload 项），确认业务恢复后逐项加回；`unwanted-reply-threshold` 触发时缓存被清空是设计行为，不是数据损坏。

9.8 日志最小化与留存策略

查询日志是查询流在自己门口的副本：客户端地址、QNAME、类型与时序都在里面。加密传输把链路上的观察者请出去之后，日志就成了新的集中观察者。原则是**默认不记、排障时临时开、开了必须有到期删除**。

```
# /etc/unbound/unbound.conf.d/70-logging.conf
server:
  use-syslog: yes
  logfile: ""
  verbosity: 1
  log-queries: no
  log-replies: no
  log-local-actions: no
  log-servfail: yes
  statistics-interval: 0
```

为什么这么设：手册说明 `log-queries` / `log-replies` 每查询打一行、会**显著降低**服务速度，默认 no；`verbosity: 1` 只给运行级信息，2 以上开始出现逐查询细节；`statistics-interval: 0` 关闭周期统计打印。唯一保留的例外是 `log-servfail: yes`：手册说它单独打印 SERVFAIL 原因，量小、级别是 error，排障价值远高于画像风险。`logfile: ""` 与 `use-syslog: yes` 显式化，避免日志落两份（手册说明两者互相覆盖）。全量结构化日志用 dnstap 时是另一套配置（第 7、13 章），保留期与到期删除要一并配好。

怎么验证：`unbound-control get_option log-queries ; journalctl -u unbound --since '5 min ago' | grep -c 'query:' || true` ——前者为 no、后者为 0；随后制造一次 SERVFAIL，应能看到原因行。

失败模式 / 回滚点：临时用 `set_option verbosity 3` / `set_option log-queries yes` 排障后忘记关，是最常见的破功方式，且要 reload 才回到配置文件的值。回滚：`set_option log-queries no` 立即生效；已落盘的历史日志不会因关掉采集而消失，必须靠下面的策略。

```
# /etc/systemd/journald.conf.d/30-dns-retention.conf —— 30 天留存
[Journal]
Storage=persistent
SystemMaxUse=512M
MaxRetentionSec=30day
```

为什么这么设：`MaxRetentionSec` 把「30 天」变成系统执行而非口头承诺，`SystemMaxUse` 给磁盘设上限，避免日志写满根分区把解析器一起拖挂。若发行版用 rsyslog 落文件，等价做法见官方手册的 logrotate 配置（`daily` / `rotate 30` / `compress`），二者选一。

怎么验证：`grep -R '^MaxRetentionSec\|^SystemMaxUse' /etc/systemd/journald.conf.d/ ; journalctl --disk-usage`

失败模式 / 回滚点：改完不 `systemctl restart systemd-journald` 就不生效；`SystemMaxUse` 过小会让日志在保留期到期前被清掉。回滚：删掉 drop-in 并重启 journald，同时确认备份里没有无限期保存的查询日志副本。

9.9　防火墙只放行必要端口：nftables 基础写法

access-control 是应用层的门，防火墙是内核层的门；两道都要有，一次误配才不至于直接变成公网暴露。

```
#!/usr/sbin/nft -f
# /etc/nftables.conf
flush ruleset
table inet filter {
  chain input {
    type filter hook input priority filter; policy drop;
    ct state established,related accept
    ct state invalid drop
    iif lo accept
    # 管理口：只从管理网段进 SSH（本机唯一额外放行的端口）
    ip saddr 198.51.100.0/24 tcp dport 22 ct state new accept
    ip saddr 192.0.2.0/24 udp dport 53 ct state new accept   # DNS，只从客户端网段进
    ip saddr 192.0.2.0/24 tcp dport 53 ct state new accept
    ip6 saddr 2001:db8::/64 udp dport 53 ct state new accept
    ip6 saddr 2001:db8::/64 tcp dport 53 ct state new accept
    icmpv6 type { nd-neighbor-solicit, nd-neighbor-advert, nd-router-advert, packet-too-big } accept
    icmp type { destination-unreachable, time-exceeded } accept
  }
  chain output {
    type filter hook output priority filter; policy accept;  # 递归出向 53 需留路
  }
}
```

为什么这么设：policy drop 默认拒绝，配 ct state established,related accept 让应答回来，ct state new 限定只有新建连接需显式放行。53 同时开 UDP 与 TCP，对应前面的 do-udp / do-tcp。IPv6 ND 与 packet-too-big 必须留：全丢掉表现为「v4 正常、v6 时通时断」，常被误判成上游问题。出向暂用 accept；收紧出向时记得本机解析器自己的递归查询也要留路，若网关还有 :53 劫持规则，本机 Unbound 的出向 53 必须按 uid 豁免（第 14 章）。

怎么验证：nft -c -f /etc/nftables.conf && nft list ruleset | head -30 ——-c 只查语法、不加载；加载后从 203.0.113.9 测 dig @192.0.2.1 example.net A +time=2 +tries=1 应超时，从 192.0.2.0/24 内应有答案。

失败模式 / 回滚点：在唯一一条远程 SSH 会话里直接 nft -f 一个 policy drop 的文件，可能把自己锁在外面——-c 查语法，不查「你会不会被锁」。操作前先 nft list ruleset > /etc/nftables.conf.bak，最好有带外控制台。回滚：nft -f /etc/nftables.conf.bak；nft -f 重建表会清掉运行时 nft add element 加的集合元素与计数器（第 14 章），所以备份必须是完整 ruleset 而非几行命令。

9.10　完整可复制的加固基线

把前九节合成一**个文件**。假设客户端在 192.0.2.0/24 与 2001:db8::/64，管理主机在 198.51.100.0/24，本机监听 192.0.2.1 / 2001:db8::1，走普通 53 端口、不做加密传输、不做 DNSSEC 验证。

```
# /etc/unbound/unbound.conf.d/90-minimal-exposure.conf
# 完整基线
server:
  # 9.1 监听面 / 9.2 默认拒绝 + 最长前缀匹配
  interface: 127.0.0.1
  interface: ::1
  interface: 192.0.2.1
  interface: 2001:db8::1
  port: 53
  do-ip4: yes
  do-ip6: yes
  do-udp: yes
  do-tcp: yes
  access-control: 127.0.0.0/8 allow
  access-control: ::1/128 allow
  access-control: 192.0.2.0/24 allow
  access-control: 192.0.2.9/32 deny
```

```
  access-control: 2001:db8::/64 allow
  access-control: 0.0.0.0/0 refuse
  access-control: ::/0 refuse
  # 9.4 降权与信息隐藏 / 9.5 防 DNS rebinding
  username: unbound
  chroot: ""
  directory: "/etc/unbound"
  hide-identity: yes
  hide-version: yes
  identity: ""
  version: ""
  private-address: 10.0.0.0/8
  private-address: 172.16.0.0/12
  private-address: 192.168.0.0/16
  private-address: 169.254.0.0/16
  private-address: fd00::/8
  private-address: fe80::/10
  private-address: ::ffff:0:0/96
  private-domain: "home.example.net."
  # 9.6 特殊用途域不出网 / 9.7 限流与异常查询
  local-zone: "test." always_nxdomain
  local-zone: "invalid." always_nxdomain
  local-zone: "example." always_nxdomain
  local-zone: "onion." always_nxdomain
  ratelimit: 1000
  ratelimit-factor: 10
  ip-ratelimit: 50
  ip-ratelimit-cookie: 500
  unwanted-reply-threshold: 10000000
  do-not-query-localhost: yes
  tcp-connection-limit: 192.0.2.0/24 50
  tcp-connection-limit: 2001:db8::/64 50
  deny-any: yes
  harden-short-bufsize: yes
  harden-large-queries: yes
  minimal-responses: yes
  max-udp-size: 1232
  edns-buffer-size: 1232
  # 9.8 日志最小化
  use-syslog: yes
  logfile: ""
  verbosity: 1
  log-queries: no
  log-replies: no
  log-local-actions: no
  log-servfail: yes
  statistics-interval: 0
```

为什么这么设：两个刻意的缺席：没有 `auto-trust-anchor-file` / `module-config`——DNSSEC 验证在第 10 章；没有 `tls-service-key` / `tls-service-pem`、也没有 853 或 443 监听——加密传输是第 11、12 章。提前塞进来会引入给不出验证命令的状态。

怎么验证：`unbound-checkconf /etc/unbound/unbound.conf.d/90-minimal-exposure.conf && echo CONFIG_OK`

失败模式 / 回滚点：`unbound-checkconf` 只验语法与路径，不验 `access-control` 语义、不验 AppArmor 放行、不验防火墙丢包；通过不等于基线生效。上线顺序固定为「先备份旧 `unbound.conf.d/` 与 ruleset → checkconf → `systemctl restart unbound`（改了 `interface`，reload 不生效）→跑 9.11」，回滚见 9.12。

9.11 逐条验证命令

按序号跑，任何一条不过就停在原地，不要同时改多个参数。

```
# 0）备份（必须先做）；1）语法
cp -a /etc/unbound/unbound.conf.d /root/unbound.conf.d.bak.$(date +%F)
nft list ruleset > /etc/nftables.conf.bak
unbound-checkconf /etc/unbound/unbound.conf.d/90-minimal-exposure.conf
# 2）监听面：不应出现 0.0.0.0 或 [::]
ss -lunp 'sport = :53' ; ss -lntp 'sport = :53'
# 3）放行源可递归（带 ra）；5）非递归查询不能挖缓存
dig -b 192.0.2.1 @192.0.2.1 example.org A +short
dig @192.0.2.1 example.org A +norecurse
# 4）公网源不可递归（在 203.0.113.9 上跑，期望 REFUSED 或超时）
dig @192.0.2.1 example.net A +time=2 +tries=1
# 6）信息隐藏 + 降权；7）特殊用途域本地终结且不误伤测试域名
dig @127.0.0.1 version.server TXT CHAOS +short ; ps -o user=,comm= -C unbound
dig @127.0.0.1 whatever.test. A ; dig @127.0.0.1 example.org A +short
nft -c -f /etc/nftables.conf ; nft list chain inet filter input -a  # 8）防火墙语法与计数
# 9）日志最小化与留存
unbound-control get_option log-queries ; grep -R '^MaxRetentionSec' /etc/systemd/journald.conf.d/
unbound-control stats_noreset | grep -E 'total.num.queries|num.cachehits|num.queries_discard_timeout'  # 10）统计基线，不清零
```

为什么这么设：顺序是「配置→内核→应用→行为」，先排除语法与端口，再问递归，最后看统计；反过来（先看统计）容易把 REFUSED 当成缓存未命中。第 4 条必须从**未放行的源地址**执行，在本机跑会命中回环 allow，得到假阳性。

怎么验证：第 3 条有答案且带 ra，第 4 条 REFUSED / 超时，第 6 条 CHAOS 为空且进程用户是 unbound，第 7 条前空后有，第 9 条 log-queries 为 no 且看得到 30 天配置，即视为基线生效。

失败模式 / 回滚点：unbound-control 连不上先查 control-interface 与证书权限，不要据此判定实例挂了；nft -c 通过但第 4 条仍返回答案，说明最宽的那条 allow 还在。逐条期望值就是验收表，失败时回到对应小节的回滚点。

9.12　出向收敛：把 DNS 隧道数据外泄堵在边界上

DNS 隧道是加固中最常见的通病之一：防火墙放行了 UDP/53，却没管**谁能发、发给谁**。攻击者在内网主机上把凭据或配置编码进子域标签（base64chunk.exfil.example.net），查询一路经过内部解析器到达其控制的权威服务器；应答用 TXT / NULL / CNAME 回传，就形成一条双向 C2 与外泄信道。吞吐很低也够用，而 DGA 轮换域名让「封几个坏域名」必然失效。检测与排查见第 29 章。

```
# 教学骨架：客户端只能问内部解析器，内部解析器只能问受控上游
# 示例地址取自 RFC 5737 文档段
nft add rule inet filter output ip daddr != 192.0.2.53 udp dport 53 counter drop
nft add rule inet filter output ip daddr != 192.0.2.53 tcp dport 53 counter drop
```

为什么这么设：把「谁能发 53」收敛到一台自己可控的解析器，外泄信道的最后一跳就消失；同时保留内部解析器向受控上游的例外。若上游用 DoT，边界上还要放行 853 给指定地址，并在解析器侧写死 forward-addr。

怎么验证：从一台客户端 dig @198.51.100.9 example.org 应超时；dig @192.0.2.53 example.org 应正常；nft counter 应随被丢弃的直连查询增长。再用 dnstap 看是否还有高熵长标签流向非常规域。

失败模式 / 回滚点：最容易误伤的是容器/kubelet 自带的 DNS、以及 IoT 设备硬编码的公共解析器。先只 counter 观察一周再 drop；回滚就是删掉这两条规则。**注意**：这一层只防「客户端绕过内部解析器」，不防内部解析器自己被滥用——那要靠第 29 章的速率与唯一 FQDN 检测。

9.13　回滚点

变更	生效方式	回滚动作	回滚代价
interface / port	**仅 restart**	恢复 unbound.conf.d.bak.*，systemctl restart unbound	缓存重建，可接受
access-control、ratelimit、harden-*	reload 可生效	撤掉误配的行，unbound-control reload	无状态损失

变更	生效方式	回滚动作	回滚代价
`private-address` / `local-zone`	reload 可生效	逐条加 `private-domain` 或删区行	该域保护暂时失效
AppArmor profile	`apparmor_parser -r`	文件移回 `/etc/unbound/**` 或补规则	部分配置不加载
nftables ruleset	`nft -f`	`nft -f /etc/nftables.conf.bak`	运行时集合 / 计数被清空
journald 留存	重启 journald	删 drop-in 并重启	历史日志不会自动消失

统一顺序：**先恢复配置，再 restart，最后重跑 9.11**。有一类故障不该靠回滚解决——若第 4 条（公网源不可递归）失败，回滚到「更宽松的旧配置」只会把暴露面重新打开；正确方向是收紧到 REFUSED 为止。另有一条跨层约束：`unbound-checkconf` 从不告诉你防火墙与 AppArmor 的判决，所以「配置全对但不生效」时按 `nft list ruleset` → `journalctl -k | grep DENIED` → `access-control` 的顺序查，而不是先怀疑上游。

9.14 本章要点

- `interface` 只列必要地址，绝不写 `0.0.0.0` / `::`；它的变更 reload 不生效、只有 restart 生效。`access-control` 无匹配即 `refuse`，按**最长前缀**匹配且**与语句顺序无关**，兜底写 `refuse` 比 `deny` 更协议友好。
- 开放解析器能放大攻击，是因为 UDP 源地址可伪造、而「几十字节的小查询」能换回「上千字节的大应答」，放大倍数就是这两个字节数之比；默认拒绝递归、`minimal-responses` / 1232 压应答、`ratelimit` / `ip-ratelimit` 压速率、防火墙兜底，四道缺一不可。
- `username` 降权、`hide-identity` / `hide-version` 隐藏 CHAOS 信息、`chroot: ""` 明确暂不启用，三者互相独立；AppArmor 只允许读 `/etc/unbound/**` 时，把文件放到 `/var/lib/unbound/` 会被 DENIED，而 `unbound-checkconf` 照样通过。
- `private-address` 剔除公网应答里的内网地址（含 SVCB / HTTPS 的 hint）防 DNS rebinding，用 `private-domain` 开白名单；`local-zone` 让 [RFC 6761] 特殊用途域本地终结，别把本书验收用的 `example.net` / `example.org` 也写进去。
- 官方选项名是 `ratelimit` / `ip-ratelimit`（无连字符）且标为 experimental：前者按目标域回 SERVFAIL、缓存命中不受限，后者按源 IP 静默丢弃；`unwanted-reply-threshold` 触发时清缓存是设计行为。日志默认 `verbosity: 1` + `log-queries: no`，只留 `log-servfail`，且必须有 30 天到期删除。
- 基线刻意不含信任锚与 TLS 监听：DNSSEC 在第 10 章、DoT / DoH 在第 11、12 章。所有变更都要「先备份 → checkconf → restart →逐条验证」，且防火墙与 AppArmor 的判决不在 `unbound-checkconf` 检查范围内。

9.15 参考资料

- Unbound 官方手册，`unbound.conf(5)`（`interface`、`access-control`、`private-address`、`local-zone`、`ratelimit`、`ip-ratelimit`、`unwanted-reply-threshold` 与默认值）：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html
- Unbound 官方手册，`unbound-checkconf(8)`（检查范围与用法）：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound-checkconf.html
- Unbound 官方手册，`unbound-control(8)`（`get_option` / `set_option` / `stats_noreset` / `list_local_zones`）：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound-control.html
- NLnet Labs，Unbound 项目文档入口（版本相关的默认区清单与 AppArmor 说明）：https://nlnetlabs.nl/documentation/unbound/
- NIST SP 800-81r3《Secure Domain Name System (DNS) Deployment Guide》官方出版物页面（开放解析器与部署基线）：https://csrc.nist.gov/pubs/sp/800/81/r3/final
- RFC 6761，*Special-Use Domain Names*：https://www.rfc-editor.org/rfc/rfc6761
- RFC 5737，*IPv4 Address Blocks Reserved for Documentation*：https://www.rfc-editor.org/rfc/rfc5737
- DNS Flag Day 2020，EDNS 缓冲区尺寸建议（1232 的由来）：https://dnsflagday.net/2020/

第十章

DNSSEC 验证：信任锚、放行与拒绝

一台递归解析器把 DNSSEC 打开，最容易产生的错觉是「配置里写了 `validator`，所以我已经安全了」。实际生效与否取决于三件事：`module-config` 里有没有 `validator`、信任锚（trust anchor）文件在不在且是否可写、以及验证失败时策略是**放行**还是**拒绝**。三者任一缺位，应答就可能被原样透传给客户端，而配置文件看起来毫无问题。

10.1 信任锚不是一段静态配置

验证器判断一个应答是 Secure 还是 Bogus，需要从一个它无条件信任的公钥出发逐级验签。这个起点就是信任锚。对绝大多数部署来说，它只有一个：根区的 KSK，以一条 DS 或 DNSKEY 记录的形式存在。

关键不在于「有没有这条记录」，而在于**它是不是一个会随时间演化的状态**。根 KSK 会轮换，密钥会撤销，[RFC 5011] 定义的就是验证器如何在无人值守的情况下跟踪这套变化。因此 Unbound 把信任锚分成两类配置：

配置项	语义	是否自动滚动	适用场景
`trust-anchor-file`	静态可信密钥文件（DS / DNSKEY，zone file 格式）	否	固定的内部锚、实验室里手动维护的锚
`trust-anchor: "<RR>"`	直接在配置里写一条 DS / DNSKEY	否	单个测试锚
`auto-trust-anchor-file`	由 [RFC 5011] 探针跟踪并**回写**的锚文件	是	根锚，生产环境默认选择
`trusted-keys-file`	BIND 9 风格 `trusted-keys { ... };` 格式	否	从 BIND 迁移时的兼容入口

`auto-trust-anchor-file` 的「auto」落在两处：一是 Unbound 定期（官方手册写的是每月数次）向锚所在区发 [RFC 5011] 探针，观察新密钥并更新状态；二是**它会写文件**——文件变化时要创建临时文件再原子替换。这带来三条容易被忽略的运行条件：① Unbound 运行用户（通常 `unbound`）对**锚文件本身**有写权限；②对**锚文件所在目录**也有写权限（回写要创建临时文件）；③若启用 `chroot`，路径必须落在 chroot 内且目录同样可写（第 4 章讲过 chroot 会重写所有相对路径的解释基准）。只要有一条不满足，实例仍能启动、仍能解析，但锚再也不会更新——在根 KSK 轮换到来之前完全不可见，是典型的「平时全绿、某天全挂」。

10.2 用 `unbound-anchor` 初始化根锚

`unbound-anchor(8)` 是官方提供的根信任锚引导工具，用来在 Unbound 启动前创建或更新锚文件。它带有内置的初始根锚与根更新证书；文件不存在或为空时写入内置锚，随后尝试通过 DNS 做 [RFC 5011] 跟踪，跟踪失败时再按 [RFC 7958] 从 IANA 获取 `root-anchors.xml` 并用证书验签后更新。网络不通或证书过期时它**不改动原文件**，让解析器照常能起来——这个「宁可保持旧状态也不盲写」的行为是刻意设计。

推荐的初始化顺序（以 Debian/Ubuntu 常见路径为例，其他发行版路径见官方手册或打包说明）：

```
# 1. 建目录并给 Unbound 运行用户可写权限（文件 + 目录都要）
install -d -o unbound -g unbound -m 0755 /var/lib/unbound

# 2. 以 Unbound 运行用户身份生成 / 更新根锚
sudo -u unbound unbound-anchor -a /var/lib/unbound/root.key

# 3. 查看锚内容：应能看到状态标记（VALID / VALIDATING 等）
cat /var/lib/unbound/root.key

# 4. 确认属主与权限
stat -c '%U:%G %a %n' /var/lib/unbound /var/lib/unbound/root.key
```

为什么这么设：`sudo -u unbound` 不是形式主义——回写由 Unbound 进程自己完成，锚文件属主是 `root` 时只能在内存里跟踪滚动，重启就丢回旧锚。退出码也值得注意：官方手册写明**用证书更新成功或写入内置锚时退出 1**，无需更新、通过 [RFC 5011] 更新或出错时退出 0；不要写成 `unbound-anchor ... && systemctl start unbound`，第一次初始化反而会挡住启动。

怎么验证：`unbound-anchor -l | head -20` 打印内置锚与证书（不改动文件），用于核对初始锚来源；再跑 `sudo -u unbound test -w /var/lib/unbound/root.key && echo "anchor writable"` 与 `sudo -u unbound test -w /var/lib/unbound && echo "anchor dir writable"`，两条必须都输出，只测文件不测目录是常见漏项。

失败模式 / 回滚点：若出网受限拿不到 `root-anchors.xml`，工具通常静默退出并保留内置锚，用 `cat` 确认文件非空即可，不要误以为失败。目录可写而文件是 `root:root 0644` 时改属主后重启。回滚：停实例、移走 `root.key`、重跑 `unbound-anchor` 写回内置锚；这会让 [RFC 5011] 状态从零开始、**滚动观察期需重新累积**，不要在轮换窗口内轻易做。

10.3　完整验证器配置

下面是本章的基线配置。它假定这是一台**自己做完整递归**的解析器（第 4、6 章的角色），不是转发器。

```
# /etc/unbound/unbound.conf.d/10-dnssec.conf
server:
    interface: 127.0.0.1@5353
    interface: 192.0.2.1@5353
    access-control: 127.0.0.0/8 allow
    access-control: 192.0.2.0/24 allow
    username: unbound
    chroot: ""
    do-ip4: yes
    do-ip6: no
    do-udp: yes
    do-tcp: yes
    edns-buffer-size: 1232
    max-udp-size: 1232

    # 模块链：validator 必须在 iterator 之前
    module-config: "validator iterator"

    # 根信任锚：由 RFC 5011 探针跟踪，路径必须可写且在 chroot 内
    auto-trust-anchor-file: "/var/lib/unbound/root.key"

    # 真验证（默认 no）：Bogus → SERVFAIL，Secure → 置 AD 位
    val-permissive-mode: no

    # 验证失败留痕：1 = 每个失败查询一行；2 = 附原因与来源服务器
    val-log-level: 1
    val-log-squelch: no

    # 有锚时要求 DNSSEC 数据存在，缺数据即判 Bogus（默认 yes）
    harden-dnssec-stripped: yes

    # 只依据 DNSSEC-secure 的 NXDOMAIN 做子树剪枝
    harden-below-nxdomain: yes
```

```
    # 与 DNSSEC 无关：只信任权威范围内的 glue
    harden-glue: yes

    # 附加段清理：只删除 secure 应答中签名不正确的附加数据
    val-clean-additional: yes

    # 对客户端 CD 位不放松：不接受「请别替我验证」的请求
    ignore-cd-flag: yes
```

为什么这么设：module-config: "validator iterator" 是验证器的开关本身，官方手册明确写「只写 iterator 就是非验证服务器」，且**顺序有意义**，绝大多数模块要排在链首。auto-trust-anchor-file 提供起点，没有它，即使 validator 在链里也退化成透传。val-permissive-mode: no 是拒绝策略；val-log-level: 1 让每次 Bogus 留一行日志，否则验证失败只表现为客户端看到 SERVFAIL，排查没有抓手。edns-buffer-size: 1232 沿用第 4 章结论：DNSSEC 应答比普通应答大，出向设 1472 会在 MTU 1280（WireGuard / IPv6 最小 MTU）与 PPPoE 1492 上分片，分片被丢后表现为「验证偶发超时」「大应答域名随机 SERVFAIL」；max-udp-size 只管入向，出向要靠 edns-buffer-size。其余 harden-* 与 val-clean-additional、ignore-cd-flag 的边界在 10.6 与 10.7 逐一拆开，这里先按基线全开。

怎么验证：

```
unbound-checkconf /etc/unbound/unbound.conf
for o in module-config val-permissive-mode harden-dnssec-stripped; do unbound-control -c /etc/unbound/unbound.conf get_option "$o"
```

第一条应无错误；循环依次输出 validator iterator、no、yes。get_option 读的是**生效值**（含 include 拼接与 set_option 运行时修改），比翻配置文件可靠；某些构建不带它，此时以 stats_noreset 与配置文件为准，不要凭「配了就生效」下结论。

失败模式 / 回滚点：最常见的启动失败是 chroot 开着但锚路径没搬进 chroot、或文件属主是 root；日志里出现锚文件读写失败，同时 module-config 仍有 validator——进入「有验证器、无锚」的透传状态，比完全不验证更迷惑。回滚：恢复 chroot: "" 与可写路径、改正属主后 restart（不是 reload，模块链与 interface 变更都需 restart）。

10.4 验证器的三态

同一份 validator 模块链，在不同配置下有三种截然不同的对外行为。这是本章最需要记牢的一张表。

状态	判定条件	坏签名域的应答	好签名域的应答	日志	是否提供保护
透传不验证	无信任锚（或 module-config 只写 iterator）	返回答案，无 AD	返回答案，无 AD	无验证失败记录	否
permissive	有锚，val-permissive-mode: yes	返回答案（Bogus 被标成 Indeterminate），不置 AD	返回答案并置 AD	按 val-log-level 记录	否，仅观测
真验证	有锚，val-permissive-mode: no	**SERVFAIL**	返回答案并置 **AD**	按 val-log-level 记录	是

三态里最危险的是第一种，因为它「什么错都不报」：客户端拿到答案、监控看到成功率 100%、日志干净，而伪造数据照样进缓存。第二种是官方手册明确写出的折中——安全检查照做，但 Bogus 结果**不向客户端扣留**，仍收到坏数据，只是记成 Indeterminate 并写日志；适合上线前的灰度观测，绝不能当最终状态。**AD 位只由真正完成验证且结果 Secure 的那台服务器设置**（第 3 章）：DO 位只表示「请把 DNSSEC 相关记录一起给我」，CD 位只表示「请别替我验证」，两者都不是服务器的开关。ignore-cd-flag: yes 让本机不接受客户端的「请别验证」，把策略留在自己手里；开放给所有客户端等于提供一条绕过验证的通道。

10.4.1 三态之间的切换方法

切换不需要改文件，可以用 unbound-control set_option 临时调整（**不写回配置文件、不清缓存**），适合验收时快速对照：

```
# 切到 permissive（仅用于观测，验收完必须切回）
unbound-control -c /etc/unbound/unbound.conf set_option val-permissive-mode: yes
```

```
unbound-control -c /etc/unbound/unbound.conf get_option val-permissive-mode

# 切回真验证
unbound-control -c /etc/unbound/unbound.conf set_option val-permissive-mode: no
unbound-control -c /etc/unbound/unbound.conf get_option val-permissive-mode
```

为什么这么设：`set_option` 能在「同一份缓存、同一台实例、同一时刻」下对比两种策略，排除缓存命中的干扰；但官方手册提醒并非所有选项都支持运行时修改，且新值不写回配置文件，临时切换后必须 `restart` 让盘上值重新生效。

怎么验证：`get_option` 读当前生效值；`unbound-control stats_noreset | grep num.rrset.bogus` 的计数随坏签名查询增长，说明验证器确实在做检查——与最终放行/拒绝是两件事，permissive 模式下它同样会涨。

失败模式 / 回滚点：忘记切回 `no` 是最大风险，务必把 `set_option val-permissive-mode: no` 作为验收脚本固定收尾并在巡检里用 `get_option` 断言。回滚：`systemctl restart unbound`。

10.5　验证器行为验收表

四类样本：**好签名**＝正常签署且信任链完整（Secure）；**坏签名**＝故意破坏签名或密钥不匹配（Bogus）；**无签名（Insecure）**＝区未签署且父区有可验证的「未签名」证明，正常返回；**被剥离签名**＝本应签名，应答里的 DNSSEC 相关记录被路径上的中间设备或攻击者删掉。教学环境不要去公网找坏签名域，用 `example.net` / `example.org` 下的四个子域作**本地测试域名占位符**，在实验用的权威服务器（第 21 章的 `cloud-01`，示例地址 `192.0.2.53`）上分别配成四类；也可用 `domain-insecure` 与本地 `local-zone` 模拟无签名分支。`@127.0.0.1 -p 5353` 是本章验证器实例：

```
# 统一前置：确认实例处于「真验证」态
unbound-control -c /etc/unbound/unbound.conf get_option module-config
unbound-control -c /etc/unbound/unbound.conf get_option val-permissive-mode
```

类别	测试域名（占位符）	dig 命令	真验证下的预期	permissive 下的预期	无锚下的预期
好签名	`signed.example.net`	`dig @127.0.0.1 -p 5353 signed.example.net A +dnssec`	status: NOERROR，flags 含 **ad**，返回 A	同左（AD 照置）	NOERROR，**无 ad**，返回 A
坏签名	`bogus.example.net`	`dig @127.0.0.1 -p 5353 bogus.example.net A +dnssec`	**status: SERVFAIL**，无答案	NOERROR 返回坏数据，**无 ad**，日志记 Bogus	NOERROR 返回坏数据，无 ad，日志无记录
无签名	`unsigned.example.org`	`dig @127.0.0.1 -p 5353 unsigned.example.org A +dnssec`	status: NOERROR，返回 A，**无 ad**	同左	同左（三者一致）
被剥离签名	`stripped.example.net`（本应签名）	`dig @127.0.0.1 -p 5353 stripped.example.net A +dnssec`	**status: SERVFAIL**（harden-dnssec-stripped: yes 判 Bogus）	NOERROR 返回答案，无 ad，日志记异常	NOERROR 返回答案，无 ad，日志无记录

读表四个要点：① **无签名一类三态一致**——Insecure 不等于 Bogus，把「没有 ad」直接当故障是误判。② **只有坏签名与被剥离签名能区分三态**：好签名能区分「有没有做验证」（看 AD），区分不了「permissive 还是真验证」；四行缺一不可。③ `+dnssec` 只请求对端把 DNSSEC 相关记录一并给出，不是「请替我验证」；客户端不置 DO，验证器仍会自己向权威侧取所需记录。④ **AD 位要在本机这一跳看**：若前面有第 12 章的 DoH 前端或第 14 章的多级链，中间设备可能搬运或重写标志位，验收命令必须打在真正做验证的那台实例上。

配套的日志与计数器核对：

```
# 每次坏签名 / 剥离签名查询后，日志应新增一行
journalctl -u unbound --since '5 min ago' | grep -i 'validation failure\|bogus' | tail -20
```

```
# 验证器确实执行了检查：计数应随测试增长
unbound-control -c /etc/unbound/unbound.conf stats_noreset | grep -E 'num.valops|num.rrset.bogus|num.query.type.A'

# 清掉测试产生的缓存，避免下一轮验收命中旧结果
unbound-control -c /etc/unbound/unbound.conf flush_zone example.net
unbound-control -c /etc/unbound/unbound.conf flush_zone example.org
```

为什么这么设：val-log-level: 1 保证「每个失败的查询一行」，日志条数即可核对验收是否触发了验证路径；num.valops 与 num.rrset.bogus 把「检查是否执行」与「策略放行/拒绝」拆成两个独立观测量。flush_zone 是必要收尾：第 5 章讲过负缓存会放大故障，Bogus / SERVFAIL 同样可能被缓存，不刷会让下一轮测到上一轮的结论。

怎么验证：按「无锚→ permissive →真验证」跑三遍（无锚态用一个不配 auto-trust-anchor-file 的独立实例、端口另开，不要在生产实例上删锚），逐格对照预期列。

失败模式 / 回滚点：好签名域始终不出现 ad，按顺序查 module-config、锚文件非空可读、查询是否命中了转发器。坏签名域返回答案，第一嫌疑是 set_option 改成 yes 忘了切回，第二是 harden-dnssec-stripped: no。回滚：恢复 10.3 基线并 restart，用 get_option 逐项断言。

10.6 harden-dnssec-stripped：有锚时主动关掉就是制造漏洞

官方手册对 harden-dnssec-stripped 的定义是：**要求有信任锚的区必须带回 DNSSEC 数据，数据缺失就把该区判为 Bogus**。默认值 yes。若设为 no，当收不到 DNSSEC 数据、或 DNSKEY 数据验证不过时，该区会被判成 Insecure——手册的原话是「表现得就像没有信任锚」，并直接警告这会带来**禁用该区安全性的降级攻击风险**。

这条结论要连着本章标题读：在有锚的实例上把 harden-dnssec-stripped: no 写上，不是「略微放松」，而是**主动制造一个剥离签名降级漏洞**。路径上的中间设备（或攻击者）只要把 RRSIG / DNSKEY 从应答里删掉，验证器就从「拒绝」滑到「按未签名放行」，且客户端拿到答案、不报 SERVFAIL。验收表第四行会因此从 SERVFAIL 变成 NOERROR，这正是降级成功的证据。

```
# 反面示例：有信任锚时不要这样写
server:
    auto-trust-anchor-file: "/var/lib/unbound/root.key"
    module-config: "validator iterator"
    val-permissive-mode: no
    harden-dnssec-stripped: no
```

为什么不能这么设：手册给这个选项留的合法理由只有一种——你确实身处会剥掉 DNSSEC 数据的侵入式防火墙之后，或某区在「签名 / 未签名 / 错误签名」间频繁抖动。这是**已知环境下的妥协**，代价是把「缺证据」当成「已证明未签名」（条文区别见第 16 章）；妥协要写进变更记录、限定作用域、设恢复期限，不能作为默认值留着。

怎么验证：unbound-control -c /etc/unbound/unbound.conf get_option harden-dnssec-stripped 读生效值，必须是 yes；grep -rn 'harden-dnssec-stripped' /etc/unbound/unbound.conf.d/ 排查 include 覆盖——后加载的文件可能把前面写的 yes 盖掉，光看主配置文件会漏。

失败模式 / 回滚点：中间设备剥数据导致大面积 SERVFAIL 时，正确动作是**修链路**（换出向、改 MTU / EDNS 载荷、或另设一台能完整取到 DNSSEC 数据的验证器），而不是关开关。确需临时妥协时用 domain-insecure: " 区名" 限定单个区，影响面比全局关小得多；恢复后立即撤销并 restart。

10.7 三个名字像、作用却完全不同的 harden-*

harden-dnssec-stripped、harden-below-nxdomain、harden-glue 常一起出现在模板里，容易被当成「一组 DNSSEC 加固」；实际上三者分属不同层面，只有一个与 DNSSEC 直接相关。

10.7.1 harden-below-nxdomain：官方 NOTE 限定了生效范围

选项来自 [RFC 8020] 的 NXDOMAIN cut：某名字已知 NXDOMAIN，其下不必再问。但 DNSSEC 对空非终端（empty nonterminal）要求 NOERROR，而很老的权威软件会对空非终端错误返回 NXDOMAIN（反向解析里常见），故需收窄范围。官方手册在这里写了明确的 **NOTE：只有 DNSSEC-secure 的 NXDOMAIN 才会被采用，NSEC3 with opt-out 不足够**——老软件本来就没有 DNSSEC，限定在「已验证的 NXDOMAIN」就自动排除了它们。默认值 yes。

运维上最重要的推论：**它在没有验证器的部署里是空转**。无信任锚时验证器不产生 Secure 判定，剪枝没有依据；此时 `get_option harden-below-nxdomain` 仍返回 `yes`，但机制并未生效。把「参数存在」误判成「机制生效」是这一节要堵的坑；第 13 章从 QMIN 角度提过同一边界，这里补上验证器侧原因。

```
# 核对三件事，缺一不可
unbound-control -c /etc/unbound/unbound.conf get_option module-config
unbound-control -c /etc/unbound/unbound.conf get_option harden-below-nxdomain
test -s /var/lib/unbound/root.key && echo "anchor file non-empty" || echo "NO ANCHOR: nxdomain cut inert"
```

为什么这么设：三条命令分别对应「有验证器」「开关为 yes」「有可用锚」；只有前两条而第三条失败，剪枝照样不触发，这正是「无验证器时空转」的可观测证据。

怎么验证：`dig @127.0.0.1 -p 5353 nonexistent.example.org A` 返回 `status: NXDOMAIN` 只说明名字不存在，**不说明剪枝生效**。要确认剪枝，需对比同一父名下第二次查询是否不再产生出向报文（`tcpdump -ni any 'udp port 53'`），或查 `stats_noreset` 里与 aggressive 使用相关的计数（具体计数名见官方手册）。

失败模式 / 回滚点：把它当成「QMIN 的强力版」是常见误读——前者管「已知 NXDOMAIN 之下不必再问」，后者管「向每一级发多少标签」，二者独立、互不为前置（第 13 章）。老式反向解析区若因空非终端被误剪，可临时设 `no` 观察，代价仅是少量额外出向查询。

10.7.2　`harden-glue`：与 DNSSEC 无关，任何模式都该开

官方手册的定义很简短：**只信任位于服务器权威范围之内的胶水记录（glue record）**，默认 `yes`。它针对应答附加段里夹带的、不属于该权威服务器所辖区域的 A/AAAA 记录——经典的缓存投毒面：伪造的域外 glue 一旦被收下并用于后续迭代，查询就被导向攻击者指定的地址。

这条与 DNSSEC **没有依赖关系**：域外 glue 是**非权威的附加数据**，本来就不在签名覆盖范围内；即使全域部署 DNSSEC、`val-permissive-mode: no`、所有 `harden-*` 全开，若 `harden-glue: no`，投毒路径依然存在。反过来，即使完全不配信任锚、做纯透传缓存或纯转发器，清理器（scrubber）也该把域外 glue 丢掉。结论：**`harden-glue: yes` 在任何模式下都该开**，它是缓存解析器的基本卫生，不是 DNSSEC 的子选项（第 2 章讲委派自举时已提过）。

```
unbound-control -c /etc/unbound/unbound.conf get_option harden-glue
```

为什么这么设：它是默认值，最实际的验证是防「被覆盖」——从旧模板拷来的配置可能显式写 `no`。

怎么验证：`get_option` 返回 `yes`；也可用 `unbound-control lookup` 查缓存里是否出现不该有的域外地址记录。第 14 章的链路图同样把 `harden-glue: yes` 放在转发器与递归验证器两侧，属跨角色一致项。

失败模式 / 回滚点：依赖「域外 glue 直接可用」的畸形委派会变慢或解析不出来；手册里的 `harden-unverified-glue`（默认 `no`）会尝试解析域外 glue 并在失败时回退原值，是这类场景的微调。优先修权威侧委派，不要靠关 `harden-glue` 兜底。

10.7.3　`harden-referral-path`：默认关闭的实验选项

官方手册把它定义为「对引用路径做加固：为基础设施数据发起额外查询，在配了信任锚且相关区已签名时验证这些应答」，强制对引用路径上遇到的 NS 集合与名字服务器地址做 DNSSEC 验证。**默认是关闭的**，手册给出的理由有三条：给权威服务器增加负担、不是 RFC 标准、额外查询负载可能带来性能问题，并明确标为 **experimental option**；若启用，建议同时调大 `target-fetch-policy` 后面的数字以增加检查深度。

它和本章其他开关的差别值得写清楚：

选项	默认值	与 DNSSEC 的关系	成本	本章建议
`harden-dnssec-stripped`	`yes`	直接相关：决定缺 DNSSEC 数据时判 Bogus 还是 Insecure	无额外查询	保持 `yes`，关掉即降级漏洞
`harden-below-nxdomain`	`yes`	依赖 Secure NXDOMAIN，无验证器时空转	减少查询	保持 `yes`，但别误判其生效范围
`harden-glue`	`yes`	**无关**：处理非权威附加数据	少量解析	任何模式都开

选项	默认值	与 DNSSEC 的关系	成本	本章建议
harden-referral-path	no	部分相关：对引用路径基础设施数据做验证	**显著增加出向查询**	按基线保持 no，仅在受控实验里评估

怎么验证：第一条读生效值（期望 no），第二条观察额外查询负载，开启后可用同一条命令前后对比。

```
unbound-control -c /etc/unbound/unbound.conf get_option harden-referral-path
unbound-control -c /etc/unbound/unbound.conf stats_noreset | grep -E 'num.query|num.valops'
```

失败模式 / 回滚点：家庭网关或跨境链路上打开它，表现为解析变慢与出向查询数上升——引用路径每一级都多出一批基础设施查询，这不是 DNSSEC 验证本身慢，是选项的额外负载。回滚：设回 no 并 restart；第 14 章链路本来就窄，更不该默认打开实验选项。

10.7.4 val-clean-additional：只清 secure 应答里的坏附加数据

官方手册的定义：指示验证器**从 secure 应答的附加段中删除签名不正确的数据**；对 Insecure / Bogus / Indeterminate / 未检查的应答不产生影响。默认 no，用途是保护依赖这台验证器做认证的客户端免受附加段坏数据影响。它容易被误读成「又一条 DNSSEC 加固」，实际边界很窄：只作用于**已判定 Secure 的消息**，且只清附加段中签名不正确的部分；不替代 harden-glue（管非权威 glue、与签名无关），也不改变 Bogus 的 SERVFAIL 策略。

get_option val-clean-additional 应返回 yes；构造一个附加段含坏数据的 secure 应答做对比，或观察下游客户端是否仍能通过独立查询拿到所需记录。

为什么这么设：本章基线把它设为 yes，与 harden-glue：yes 形成「非权威附加数据」与「已签名消息中的坏附加数据」两道互补清理。若下游有老客户端依赖附加段里某些未正确签名的记录，可能出现「主答案正常、附加信息变少」的观感，这是符合手册语义的预期行为，不是丢包。

失败模式 / 回滚点：设回 no 并 restart 即可，不影响信任链与 AD 位判定；它属于可选增强，不进 fail-closed 的关键路径。

10.8 信任锚失效 / 根 KSK 轮换的观测与恢复

锚一旦失效，症状不是「某个域解析不了」，而是**大面积、看似无规律的 SERVFAIL**——因为根是所有签名链的公共起点。轮换窗口内若本地锚没跟上新 KSK，验证器会把整个签名 DNS 判成 Bogus。第 16 章会讲 2018 年根 KSK 轮换的条文与坐标（KeyTag 等以 IANA 的 root-anchors.xml 与官方手册为准，本章不复述编号与日期），这里只给运维动作。

10.8.1 先分清三类「全挂」

现象	可能原因	快速判别
大面积 SERVFAIL，日志有 validation failure / bogus	锚过期、锚被轮换、系统时钟漂移	date -R 对时；get_option 核对锚路径；unbound-anchor -l 对照内置锚
大面积 SERVFAIL，日志有 failed to prime trust anchor	实例被配成了转发器却带了锚	grep -rn forward-zone + 第 5 章
部分域 SERVFAIL、大应答域名更明显	EDNS 分片 / MTU，不是锚问题	dig +bufsize=1232 对比；第 4 章

第二行是角色错配、第三行是传输问题，都不是锚失效；混在一起排查最常见的后果是去调上游超时、加第二条 forward-addr（第 5 章的竞速坑），问题越修越散。系统时钟要单独强调：RRSIG 生效/过期时间与 [RFC 5011] 观察期都按墙钟计算，网关断电后时钟回到出厂时间是「间歇性全量 SERVFAIL」最易忽略的成因；验收脚本应含 timedatectl status 或 date -R 断言，NTP 要在 Unbound 启动**之前**同步完。

10.8.2 观测命令

```
# 1. 时钟与锚文件状态
date -R
```

```
timedatectl status 2>/dev/null | grep -i 'synchronized\|system clock' || true
stat -c '%U:%G %a %y %n' /var/lib/unbound /var/lib/unbound/root.key
cat /var/lib/unbound/root.key

# 2. 验证器是否还在做检查、Bogus 是否在涨
unbound-control -c /etc/unbound/unbound.conf stats_noreset | grep -E 'num.valops|num.rrset.bogus'

# 3. 日志里的验证失败是不是集中爆发
journalctl -u unbound --since '1 hour ago' | grep -i 'validation failure\|bogus\|trust anchor' | tail -40

# 4. 用内置锚对照当前锚（unbound-anchor 不带 -v 时几乎不输出）
unbound-anchor -l | head -20
```

为什么这么设：stat 的 mtime 是判断「有没有成功回写过锚」最直接的证据——长期不变说明 [RFC 5011] 探针可能没在写（权限问题、长期离线、或实例从未真正跑过验证器）。num.valops 增长而 num.rrset.bogus 暴涨，说明验证器在工作但起点不对；两个都不动则更可能是模块链或角色问题。

10.8.3　恢复顺序

```
# 1. 先修时钟（NTP 未同步时不要动锚）
systemctl restart systemd-timesyncd    # 或用站点既有的 NTP 客户端
timedatectl status | grep -i synchronized

# 2. 修权限（文件 + 目录，属主为 Unbound 运行用户）
chown unbound:unbound /var/lib/unbound /var/lib/unbound/root.key
chmod 0644 /var/lib/unbound/root.key
sudo -u unbound test -w /var/lib/unbound/root.key && echo "anchor writable"

# 3. 让 unbound-anchor 引导 / 更新锚（退出 1 表示发生了更新，不是失败）
sudo -u unbound unbound-anchor -a /var/lib/unbound/root.key
cat /var/lib/unbound/root.key

# 4. 重启并清掉事故期间缓存的 Bogus 结果
systemctl restart unbound
unbound-control -c /etc/unbound/unbound.conf flush_bogus
unbound-control -c /etc/unbound/unbound.conf flush_negative

# 5. 复跑 10.5 验收表的好签名 / 坏签名两行
dig @127.0.0.1 -p 5353 signed.example.net A +dnssec | grep -E 'status:|flags:'
dig @127.0.0.1 -p 5353 bogus.example.net A +dnssec | grep -E 'status:|flags:'
```

为什么这么设：顺序不能乱，**先修时钟再动锚**——时钟不对时，新锚的观察期与签名有效期判断都不可靠。unbound-anchor 在 [RFC 5011] 跟踪失败时退回 [RFC 7958] 证书验签下载，网络不通则保持原文件不动，故可安全重跑。flush_bogus / flush_negative 清理事故期间缓存的坏结果与负缓存；第 5 章讲过负缓存按 SOA minimum 放大故障（常见 3600s），且链路上**每一个实例都要刷**，否则上游修好了下游仍返回缓存的失败结果。

怎么验证：第 5 步第一行应 status: NOERROR 且 flags 含 ad，第二行应为 status: SERVFAIL。若好签名域仍不带 AD，回 10.3 用 get_option 核对 module-config、锚路径与 val-permissive-mode；若日志仍有 failed to prime trust anchor，说明这台是转发器，应转第 5 章而不是继续修锚。

失败模式 / 回滚点：最糟的是「为了先恢复业务」直接把 val-permissive-mode 改成 yes 或把 harden-dnssec-stripped 改成 no——所有签名域瞬间恢复，但保护同时被抹掉，日志里多出大量 Indeterminate 记录，事后难与真正的锚恢复区分。业务确需临时可用性时，应把解析切到锚正常的实例或按第 6 章本地根区镜像形态调整，而不是在本机降验证等级；恢复后撤销全部临时改动并 restart。另一个隐蔽坑：长期关机或锚文件只读的实例会**错过 [RFC 5011] 观察期**（手册写探针每月数次、机器需经常在线），只在用时开机的实验网关不能假设锚会自动跟上，应在每次启用前跑一次 unbound-anchor 并把锚纳入配置管理（第 21 章会再用到）。

10.9　转发器不要配信任锚

第 5 章已给出反面配置与日志证据：信任锚需自己向根 priming DNSKEY，而 forward-zone: name "." 把根 DNSKEY 查询也交给上游，priming 无法完成，日志持续 failed to prime trust anchor 并伴随间歇性全量 SERVFAIL——**转**

发器与验证器二选一，这里只做交叉引用。本章所有配置都假定实例自己走完整递归（第 4、6 章）；若拓扑是第 12 章的 DoH 前端 + 后端验证器或第 14 章的多级链路，确认 `dig` 验收打在**真正做验证的那一台**上，前端只搬运 DNS 报文，不能替后端验证。

10.10 加固项速查

检查项	期望	命令 / 依据
模块链	`validator iterator`	`unbound-control get_option module-config`
根锚文件	存在、非空、运行用户可写	`test -s` + `sudo -u unbound test -w`
锚所在目录	运行用户可写	`sudo -u unbound test -w /var/lib/unbound`
拒绝策略	`val-permissive-mode: no`	`get_option val-permissive-mode`
缺 DNSSEC 数据	判 Bogus	`get_option harden-dnssec-stripped` → `yes`
剥签名降级面	已堵	10.5 验收表第四行 SERVFAIL
域外 glue	已清理（与 DNSSEC 无关）	`get_option harden-glue` → `yes`
NXDOMAIN 剪枝	开，且理解只对 secure 生效	`get_option harden-below-nxdomain` + 锚就位
引用路径加固	保持默认 `no`	`get_option harden-referral-path`
附加段清理	`val-clean-additional: yes`	`get_option val-clean-additional`
客户端 CD 位	不放松验证	`get_option ignore-cd-flag` → `yes`
验证失败可观测	日志有留痕	`get_option val-log-level` $\geqslant 1$ + `journalctl`
时钟	NTP 已同步	`timedatectl status` / `date -R`
EDNS 出向	1232，避免 DNSSEC 大应答分片	`get_option edns-buffer-size`
角色	非转发器	`grep -rn 'forward-zone'`、`grep -rn 'auto-trust-anchor-file'`

本章收敛到一点：**验证器的安全性不由「开关数量」决定，由「是否 fail-closed」决定**。信任锚就位、坏签名返回 SERVFAIL、好签名带 AD、缺 DNSSEC 数据判 Bogus、域外 glue 一律清理——这五条同时成立，才算真正把「放行与拒绝」握在自己手里。

10.11 KeyTrap 与 NSEC3：验证器自身的拒绝服务

2024 年 2 月披露的两个协议级漏洞，正好打在「验证器必须尽力验证」这个设计上：**CVE-2023-50387（KeyTrap）**用同 key-tag 的大量 DNSKEY 与 RRSIG 组合，迫使验证器做平方级签名尝试；**CVE-2023-50868（NSEC3）**用高迭代计数与多 NSEC3 RRSET 迫使大量哈希计算。单个精心构造的应答就能占满一个线程，所有线程都被占住即整体拒绝服务。Unbound $\leqslant$ 1.19.0 受影响，**1.19.1** 修复。

Unbound 的修法不是「关掉 DNSSEC」，而是给验证工作量设上限并让出 CPU：

内置限额（1.19.1 起）	值
构建信任链时允许的 DNSKEY key 碰撞	4
每个 RRSET 的验证尝试	8
每次应答的验证尝试（超过则挂起）	8
挂起前允许的 NSEC3 哈希计算	8
单次查询的挂起总上限（超过则报错）	16

挂起（suspension）意味着验证器先去做别的工作、再按退避计时恢复，其它请求不会被一条恶意应答锁死；已完成的 RRSET 仍会进缓存。**关键运维结论**：遭受攻击时**不要关闭 DNSSEC 验证**来「缓解」——那等于主动降级，把缓存投毒的门重新打开；`serve-expired`（[RFC 8767]）若要兜底，应在独立线程返回过期数据，不要和验证抢同一个线程。

怎么验证：`unbound -V` 确认版本≥ 1.19.1；观察 CPU 是否长期单线程 100%、SERVFAIL 率与 bogus 计数；升级前后用同一批正常域名做基线对比。**回滚点**：升级失败时回退到上一版二进制，但那等于重新暴露在这两个 CVE 下，应视为临时状态。协议设计的根因（[RFC 4035] + [RFC 6840] 要求尝试所有 key/signature 组合）在第 16 章展开。

10.12 本章要点

- 信任锚是**会随时间演化的状态**，不是静态配置：`auto-trust-anchor-file` 按 [RFC 5011] 跟踪滚动并会回写文件，因此锚文件与所在目录都必须对 Unbound 运行用户可写，启用 `chroot` 时路径要落在 chroot 内；`unbound-anchor -a` 负责初始化，退出码 1 表示发生了更新而非失败。
- 验证器三态必须逐格验收：**无锚 = 透传不验证**（坏签名域也返回答案、日志干净）、`val-permissive-mode: yes` = **记日志但放行**（Bogus 标成 Indeterminate，好签名仍置 AD）、`val-permissive-mode: no` = **真验证**（坏签名 SERVFAIL、好签名带 AD）。无签名（Insecure）一类三态一致，不能用来区分策略。
- **有锚时设 `harden-dnssec-stripped: no` 等于主动制造剥离签名降级漏洞**：官方手册明确写它会让缺 DNSSEC 数据的区「表现得就像没有信任锚」，并警告降级攻击风险；中间设备删掉签名记录即可让验证器从拒绝滑向放行。
- 三个名字相近的开关分属不同层面：`harden-below-nxdomain` 依官方 man page 的 NOTE **只对 DNSSEC-secure 的 NXDOMAIN 生效**（NSEC3 opt-out 不够），无验证器时是空转，别把参数存在误判成机制生效；`harden-glue` **与 DNSSEC 无关**，域外夹带 glue 是缓存投毒面，任何模式都该开；`harden-referral-path` 是默认关闭的实验选项，会显著增加出向查询。`val-clean-additional` 只清 secure 应答附加段里签名不正确的数据。
- 信任锚失效 / 根 KSK 轮换表现为**大面积 SERVFAIL 且日志有 validation failure**，先分清「锚 / 时钟」「转发器角色错配」「EDNS 分片」三类成因；恢复顺序是修时钟→修权限→ `unbound-anchor` → restart → `flush_bogus` / `flush_negative`（链路上每个实例都要刷）→复跑验收表，切勿用 permissive 或关 `harden-dnssec-stripped` 来「先恢复业务」。
- 转发器上不要配信任锚（第 5 章已论证，角色二选一）；DNSSEC 的 RFC 族与设计原理在第 16 章，本章只讲配置与验证。加密传输（DoT / DoH / DoQ）保护传输隐私，DNSSEC 保护数据完整性与来源认证，二者不互相替代。

10.13 参考资料

- Unbound 官方手册，`unbound.conf(5)` 的 `auto-trust-anchor-file`、`module-config`、`val-permissive-mode`、`harden-dnssec-stripped`、`harden-below-nxdomain`、`harden-glue`、`harden-referral-path`、`val-clean-additional`、`ignore-cd-flag`: https:// unbound.docs.nlnetlabs.nl/ en/ latest/ manpages/ unbound.conf.html
- Unbound 官方手册，`unbound-anchor(8)` 的根信任锚引导、退出码与 [RFC 7958] 回退路径：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound-anchor.html
- Unbound 官方手册,`unbound-control(8)` 的 `get_option`、`set_option`、`stats_noreset`、`flush_zone`、`flush_bogus`、`flush_negative`: https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound-control.html
- Unbound 官方手册,Getting Started 配置示例中的 DNSSEC 信任锚段:https://unbound.docs.nlnetlabs.nl/en/latest/getting-started/configuration.html
- Unbound 官方手册，Home Resolver 用例（含 `auto-trust-anchor-file` 的完整配置形态）：https://unbound.docs.nlnetlabs.nl/en/latest/use-cases/home-resolver.html
- RFC Editor, [RFC 5011] *Automated Updates of DNS Security (DNSSEC) Trust Anchors*: https://www.rfc-editor.org/rfc/rfc5011.txt
- RFC Editor, [RFC 6840] *Clarifications and Implementation Notes for DNS Security*: https://www.rfc-editor.org/rfc/rfc6840.txt
- IANA，根区信任锚数据（`root-anchors.xml`，含 KeyTag 与有效期）：https://data.iana.org/root-anchors/root-anchors.xml

第十一章

DoT：端口 853 的服务端与客户端

端口 853 是 DoT（DNS-over-TLS）的专用端口：DNS 报文按 [RFC 1034] 的线格式原样封装在 TLS 记录里，前面不加 HTTP 层，也不改报文结构。它与 DNSSEC 完全是两件事——DNSSEC 提供数据完整性与来源认证，DoT 提供**这一段链路**的传输加密与隐私，两者不能互相替代。这里补齐另一侧：让 Unbound 在 853 上作为服务端接受加密查询，并给出一套完整的客户端侧配置。

11.1 服务端的最小骨架：@853 才是开关

Unbound 对 DoT 的启用方式很容易被误读：`tls-service-key` / `tls-service-pem` 只是**把证书装上去**，真正决定「哪个监听端口跑 TLS」的是 `interface` 行上的 `@853` 后缀与 `tls-port`。官方手册的说明是：`tls-port`（默认 853）上只对那些以 `@` 端口形式配置的 interface 提供 TLS 服务，而且这些端口**不再提供普通 DNS TCP 服务**。

```
# /etc/unbound/unbound.conf.d/10-dot-server.conf
# 教学用示例地址，服务端为 cloud-01 上的 Unbound
server:
    # 公网侧：仅 853，且标为 TLS 端口
    interface: 0.0.0.0@853
    interface: ::0@853
    tls-port: 853

    # 本机明文入口：给本机存根解析器与本地诊断用，不暴露到公网
    interface: 127.0.0.1@5353
    interface: ::1@5353

    tls-service-key: "/etc/unbound/tls/server.key"
    tls-service-pem: "/etc/unbound/tls/server-chain.pem"
    tls-cert-bundle: "/etc/ssl/certs/ca-certificates.crt"

    username: "unbound"
    chroot: ""
    do-udp: yes
    do-tcp: yes
    access-control: 127.0.0.0/8 allow
    access-control: 192.0.2.0/24 allow
    access-control: 2001:db8::/32 allow
    access-control: 0.0.0.0/0 refuse
    access-control: ::0/0 refuse

    # 出向 EDNS 缓冲，DNS Flag Day 2020 的保守值
    edns-buffer-size: 1232
    max-udp-size: 1232

    qname-minimisation: yes
    qname-minimisation-strict: no
```

```
    harden-glue: yes
```

为什么这么设：0.0.0.0@853 与 ::0@853 把两个地址族都挂在 TLS 端口上；127.0.0.1@5353 与 ::1@5353 是普通 DNS 端口，方便本机用 dig -p 5353 做不带 TLS 的对照诊断。tls-service-pem 必须是**完整证书链**（叶子证书 + 中间 CA，按此顺序拼接），否则某些客户端会因链不完整而失败。tls-cert-bundle 在这里不是给客户端看的下游证书，而是 Unbound **向外出向连接**（如加密上游、auth-zone 拉取）时用来认证对端的 CA 包，两项用途不要混。access-control 默认无匹配即 refuse，这里显式写出允许段与拒绝段，避免日后加接口时漏配。

怎么验证：

```
unbound-checkconf /etc/unbound/unbound.conf.d/10-dot-server.conf
ss -lntp | grep -E ':853|:5353'
```

ss 应显示 853 上只有 TCP 监听、5353 上同时有 UDP/TCP。unbound-checkconf 只校验语法与文件路径，不校验证书是否可被客户端信任，后者要靠 11.4 节的 openssl s_client。

失败模式 / 回滚点：最常见的错是只写了 interface: 0.0.0.0 加 tls-service-key，却没写 @853——结果 853 根本没被监听，或 53 上被当成了 TLS 端口而普通 TCP DNS 失效。第二个坑是证书文件放在 root 权限目录或 chroot 之外，此时换证必须 **restart**，unbound-control reload 读不到新文件（官方手册明确写了这点）；回滚点就是保留上一份 server-chain.pem 与 server.key 的备份，restart 前先做 unbound-checkconf。

11.1.1　证书文件与权限

```
install -d -m 0750 -o root -g unbound /etc/unbound/tls
# 实际部署时把私钥与完整链写入该目录，例如：
#   install -m 0640 -o root -g unbound server.key /etc/unbound/tls/server.key
#   cat leaf.pem intermediate.pem > /etc/unbound/tls/server-chain.pem
#   install -m 0644 -o root -g root /etc/unbound/tls/server-chain.pem /etc/unbound/tls/server-chain.pem
openssl x509 -in /etc/unbound/tls/server-chain.pem -noout -subject -dates
openssl pkey -in /etc/unbound/tls/server.key -check -noout
```

为什么这么设：私钥只让 root 与 unbound 组可读；证书链是公开信息，0644 即可。Unbound 在绑定端口后降权到 username: unbound，所以私钥的组权限必须包含 unbound，否则会以低权限身份读取失败。官方手册也提示，ACME 类客户端常按「服务以 root 启动」来设权限，需要主动检查。

怎么验证：

```
namei -l /etc/unbound/tls/server.key
unbound-checkconf /etc/unbound/unbound.conf.d/10-dot-server.conf
```

失败模式 / 回滚点：namei -l 任何一级目录缺执行位都会导致降权后读不到；日志里通常是 TLS 初始化失败而不是「证书过期」。回滚：恢复 0640/0750 权限后 restart，不要通过关掉 username 来「绕过」权限问题——那等于把整个递归解析器跑在 root 下。

11.2　证书的两条路线：Let’s Encrypt 与自签 + 客户端 pin

DoT 的认证名来自客户端配置里的 # 名字（见 11.3），服务端证书必须覆盖这个名字。两条路线各有边界。

11.2.1　公共 CA（以 Let’s Encrypt 为例）

```
# 教学示例：用 DNS-01 挑战为 resolver.example.net 签发证书
certbot certonly --dns-rfc2136 \
  --dns-rfc2136-credentials /etc/letsencrypt/rfc2136.ini \
  -d resolver.example.net --non-interactive --agree-tos -m admin@example.org
cat /etc/letsencrypt/live/resolver.example.net/fullchain.pem \
  > /etc/unbound/tls/server-chain.pem
cat /etc/letsencrypt/live/resolver.example.net/privkey.pem \
  > /etc/unbound/tls/server.key
chown root:unbound /etc/unbound/tls/server.key && chmod 0640 /etc/unbound/tls/server.key
systemctl restart unbound
```

为什么这么设：公共 CA 的优势是客户端只要带系统 CA 包就能认证，不需要额外分发材料；代价是必须有一个可解析、可被 CA 验证的域名，并且续期要能触发 restart。用 DNS-01 而不是 HTTP-01，是因为这台机器上 80/443 可能根本不开放（共端口取舍见后文）。`fullchain.pem` 已经是叶子 + 中间 CA，直接可用，不要再手工拼一次。

怎么验证：

```
openssl s_client -connect 192.0.2.1:853 -servername resolver.example.net \
  -showcerts -CAfile /etc/ssl/certs/ca-certificates.crt </dev/null 2>/dev/null \
  | grep -E 'Verify return code|subject=|issuer='
```

期望 `Verify return code: 0 (ok)`。

失败模式 / 回滚点：certbot 的续期钩子如果只做 `reload`，而私钥路径在 chroot 之外，新证书不会被加载，表现为续期后某一天突然全量 TLS 握手失败（旧证书过期的那一刻）。钩子必须 `systemctl restart unbound`，并在钩子开头跑一次 `unbound-checkconf`。回滚：保留 `server-chain.pem.bak` / `server.key.bak`，restart 前校验。

11.2.2 自签 CA + 客户端 pin

自己签一张只用于这台解析器的证书，把 CA 公钥（或叶子证书的 SPKI 哈希）固化到客户端，可以避免「公共 CA 误签发」这一类信任面，也省掉域名与续期流程。

```
# 教学示例：生成自签 CA 与服务端证书（仅在实验环境内使用）
openssl req -x509 -newkey rsa:2048 -sha256 -days 3650 -nodes \
  -keyout /etc/unbound/tls/lab-ca.key \
  -out /etc/unbound/tls/lab-ca.pem \
  -subj "/CN=Lab DoT CA" -addext "basicConstraints=critical,CA:TRUE"

openssl req -newkey rsa:2048 -nodes \
  -keyout /etc/unbound/tls/server.key \
  -out /etc/unbound/tls/server.csr \
  -subj "/CN=resolver.example.net"
openssl x509 -req -in /etc/unbound/tls/server.csr \
  -CA /etc/unbound/tls/lab-ca.pem -CAkey /etc/unbound/tls/lab-ca.key \
  -CAcreateserial -days 825 -sha256 -out /etc/unbound/tls/leaf.pem \
  -extfile <(printf 'subjectAltName=DNS:resolver.example.net\nextendedKeyUsage=serverAuth\n')

cat /etc/unbound/tls/leaf.pem /etc/unbound/tls/lab-ca.pem \
  > /etc/unbound/tls/server-chain.pem
chmod 0640 /etc/unbound/tls/server.key && chown root:unbound /etc/unbound/tls/server.key

# 生成客户端要 pin 的 SPKI SHA-256 (Base64)
openssl x509 -in /etc/unbound/tls/leaf.pem -pubkey -noout \
  | openssl pkey -pubin -outform DER \
  | openssl dgst -sha256 -binary | openssl base64 -A
```

为什么这么设：`subjectAltName` 必须写 DNS 名，现代 TLS 栈不再看 CN；`extendedKeyUsage=serverAuth` 防止这张证书被拿去做别的用途。生成 SPKI 哈希而不是证书指纹，是因为**换证书时可以复用同一把私钥**，pin 不失效；如果 pin 的是整张证书指纹，每次续期都要重发所有客户端配置。

怎么验证：

```
openssl verify -CAfile /etc/unbound/tls/lab-ca.pem /etc/unbound/tls/leaf.pem
openssl s_client -connect 192.0.2.1:853 -servername resolver.example.net \
  -CAfile /etc/unbound/tls/lab-ca.pem </dev/null 2>/dev/null \
  | grep 'Verify return code'
```

失败模式 / 回滚点：自签路线的典型故障是客户端拿系统 CA 包去验证自签证书，必然失败——此时不是「DoT 不通」，而是「认证材料不对」。回滚：客户端侧改回带公共 CA 的上游，或把 `lab-ca.pem` 正确分发为 `tls-cert-bundle`。pin 值一旦泄露不敏感（它是公钥哈希），但**换私钥必须同步换 pin**，这是运维上最容易漏的一步。

11.3　客户端：单上游、TLS 认证名、fail-closed

下面是一份完整可复制的客户端配置。客户端是网关上的 Unbound 转发器，上游是云主机上的 DoT 服务端（用公共解析器地址 192.0.2.53 或 1.1.1.1 作教学占位均可，重点是 @853# 名字的形态）。

```
# /etc/unbound/unbound.conf.d/10-dot-client.conf
# 教学用示例地址：上游为 192.0.2.53@853（可替换为公开 DoT 服务）
server:
    interface: 127.0.0.1@53
    interface: ::1@53
    access-control: 127.0.0.0/8 allow
    access-control: 192.0.2.0/24 allow
    access-control: 0.0.0.0/0 refuse

    # 认证上游证书所用的 CA 包；自签场景换成 lab-ca.pem
    tls-cert-bundle: "/etc/ssl/certs/ca-certificates.crt"

    edns-buffer-size: 1232
    max-udp-size: 1232
    qname-minimisation: yes
    qname-minimisation-strict: no
    harden-glue: yes

    # fail-closed：禁止 SERVFAIL 时逃出本区自行递归 / 走明文
    # 转发器上不要配 auto-trust-anchor-file（见第 5 章）
    do-not-query-localhost: yes

forward-zone:
    name: "."
    # @853 指定端口，# 认证名 指定证书要匹配的名字
    forward-addr: 192.0.2.53@853#resolver.example.net
    forward-tls-upstream: yes
    forward-first: no
```

若上游用公开 DoT 服务，forward-addr 的形态同样是 IP@853# 认证名（例如 1.1.1.1@853#cloudflare-dns.com）；这里统一用 192.0.2.53@853#resolver.example.net 作为教学占位，避免把章内示例绑到某家具体厂商的名字上。认证名必须与该服务的证书 SAN 一致，不是随便填的标签。

为什么这么设：@853 让客户端连 TLS 端口而不是 53；#resolver.example.net 是**认证名**，官方手册说得很清楚：不写 # 时「任何名字都被接受」，证书仍需匹配 tls-cert-bundle 里的某个 CA——这等于把认证从「验证特定主体」退化成「任意可信 CA 签发的证书都行」，中间人只要能拿到任一可信 CA 的证书就能冒充。因此 # 后必须非空且与实际证书 SAN 一致（组合顺序是 @ 在前、# 在后）。forward-first: no 是第 5 章的 fail-closed 结论沿用：上游 SERVFAIL 或超时，两波都失败就给客户端 SERVFAIL，绝不静默退回明文 53。forward-addr 只写一条——多条是并发竞速取最快，不是主备，会把查询送到既定上游之外。

怎么验证：

```
unbound-checkconf /etc/unbound/unbound.conf.d/10-dot-client.conf
dig @127.0.0.1 example.org A +short
tcpdump -ni any 'tcp port 853 or udp port 53 or tcp port 53' -c 20
```

出向抓包里应只有 tcp port 853 的 TLS 流量，**不应出现**向 192.0.2.53:53 的明文查询。再断掉上游 853，重复 dig，应看到 SERVFAIL 而不是「慢一点但还是返回答案」。

失败模式 / 回滚点：三个高频错误：①漏写 tls-cert-bundle →连接无法认证，日志里是 TLS 相关错误而非超时；②漏写 # 认证名→表面上能通，但认证强度已下降，这是**静默的**；③误把 forward-first 设成 yes 或加了第二条 forward-addr →静默明文降级或过滤表失效。回滚：恢复 10-dot-client.conf.bak 并 restart；断链演练后记得检查链路各层负缓存（第 13 章）。

11.3.1　自签场景的客户端写法

```
# /etc/unbound/unbound.conf.d/10-dot-client-pinned.conf
server:
```

```
    interface: 127.0.0.1@53
    tls-cert-bundle: "/etc/unbound/tls/lab-ca.pem"
forward-zone:
    name: "."
    forward-addr: 192.0.2.53@853#resolver.example.net
    forward-tls-upstream: yes
    forward-first: no
```

为什么这么设：tls-cert-bundle 换成自签 CA 的 PEM，认证链闭合在我们自己手里；# 后的名字仍要与自签证书的 SAN 一致，Unbound 不会因为是自签 CA 就跳过主机名检查。换个角度说，「pin」在这里是通过**缩窄 CA 信任集**实现的，而不是靠某个 pin 参数（Unbound 的 forward-addr 没有 SPKI pin 选项；SPKI pin 用在 kdig 这类诊断客户端上，见 11.4）。

怎么验证：

```
unbound-checkconf /etc/unbound/unbound.conf.d/10-dot-client-pinned.conf
dig @127.0.0.1 example.org A +short
```

失败模式 / 回滚点：把 lab-ca.pem 分发错版本（比如分发了私钥或旧 CA）会导致所有客户端一起 TLS 失败，且各客户端表现一致，容易被误判为「上游挂了」。回滚：客户端 tls-cert-bundle 指回系统 CA 包，同时上游临时换回公共 CA 证书。

11.4 验证命令：openssl s_client 与 kdig

服务端的 TLS 层与 DNS 层要分开验证。只看 dig 成功，无法证明「证书被正确认证」。

```
# ① 证书链 / 认证名 / SNI（把 192.0.2.1 换成实际服务端教学地址）
openssl s_client -connect 192.0.2.1:853 \
  -servername resolver.example.net \
  -CAfile /etc/ssl/certs/ca-certificates.crt \
  -showcerts -tlsextdebug </dev/null 2>/dev/null \
  | grep -E 'Verify return code|subject=|issuer=|ALPN|server name'

# ② 用 kdig 走 DoT 实际发一次查询（严格认证）
kdig -d @192.0.2.1 +tls-ca +tls-hostname=resolver.example.net \
     +tls-sni=resolver.example.net example.org A

# ③ 用 kdig 走 SPKI pin（自签或不想依赖 CA 时）
kdig -d @192.0.2.1 +tls-pin=<BASE64_SPKI_SHA256> \
     +tls-hostname=resolver.example.net example.org A

# ④ 一次调用里复用同一条 TLS 连接，验证连接复用
kdig @192.0.2.1 +tls-ca +tls-hostname=resolver.example.net +keepopen \
     example.org A example.net AAAA

# ⑤ 不带认证名的机会式（Opportunistic）模式，仅作连通性对照
kdig -d @192.0.2.1 +tls example.org A
```

为什么这么设：openssl s_client 只验证 TLS 握手与证书链，Verify return code: 0 (ok) 加上 subject= 中的 SAN 匹配，才算服务端这一侧装对了；kdig +tls-ca +tls-hostname 则同时验证「DNS 报文能装进 TLS 并拿回答案」，是端到端的检查。+tls-pin 用 Base64 编码的 SPKI SHA-256 做带外密钥 pin（Out-of-Band key-pinned privacy profile），对应 [RFC 7858] 的认证模型；+tls-sni 显式设置 SNI。+keepopen 让同一次 kdig 调用里的多个查询共用一条连接，用来观察连接复用是否生效。第⑤条的 +tls 是机会式模式，不做认证，**只用于判断端口通不通**，绝不能当作验收通过。

怎么验证（验收判据）：

```
kdig @192.0.2.1 +tls-ca +tls-hostname=resolver.example.net example.org A \
  | grep -E 'status: NOERROR|ANSWER SECTION|TLS'
unbound-control stats_noreset | grep -E 'num.query.tcpout|num.query.tls' || true
```

status: NOERROR 且有 ANSWER SECTION 才算通过；若第①步 Verify return code 非 0，不要往下查 DNS，先回去修证书链。

失败模式 / 回滚点：`+tls-ca` 未指定文件时默认用系统证书库，自签场景必须 `-CAfile` 指向 `lab-ca.pem`（`kdig` 用 `+tls-ca=/path` 形式）。另一个坑是 `+tls-hostname` 与 `-servername` 写错成 IP——证书里通常没有 IP 的 SAN，会报主机名不匹配。回滚：先用第⑤条确认 853 可连，再逐项加回认证参数，二分定位。

11.5　与 nginx / HAProxy 共端口的取舍

853 是专用端口，链路边界清晰：防火墙放行 853/TCP 即可，抓包能一眼区分「加密传输（DoT / DoH / DoQ）」与「普通 Web」。但专用端口也意味着在只放行 443 的网络里会被直接丢掉。想让 443 同时承载 Web 与 DoT，有两条路。

方案	做法	优点	代价
独占 853	Unbound 直接 `0.0.0.0@853`	配置最简、审计最清楚、无需额外组件	只放行 443 的网络里不通
nginx `stream` 按 SNI 分流	nginx 监听 443，SNI 为 DNS 名时转给本机 853，其余转 Web	只占一个公网端口	多一层进程、TLS 终止位置变化、日志归属变复杂
HAProxy TCP 模式	按 SNI / ALPN 或源地址分流，后端指向 853	与健康检查、限速、多后端集成方便	同上，且要正确处理 PROXY protocol
改用 DoH	443 上跑 HTTPS，见第 12 章	穿透性最好，与 Web 完全同形	引入 HTTP 层、端点仍属第三方、日志归属要重写

nginx 的 `stream` 模块可以按 TLS SNI 分流，但这里要非常小心：**分流是在 nginx 上看 ClientHello 的 SNI，TLS 本身仍由后端 Unbound 终止**（nginx 只做 TCP 转发，不解密）。因此客户端证书认证的对象依然是 Unbound，nginx 不需要、也不应该持有私钥。

```
# /etc/nginx/stream.d/dot-multiplex.conf ——教学示例，仅示意分流形态
stream {
    map $ssl_preread_server_name $backend {
        resolver.example.net  127.0.0.1:853;
        default               127.0.0.1:8443;
    }
    server {
        listen 443;
        ssl_preread on;
        proxy_pass $backend;
        proxy_connect_timeout 5s;
    }
}
```

为什么这么设：`ssl_preread` 只读 ClientHello 里的 SNI，不解密流量，所以后端 Unbound 仍独立完成 TLS 握手与证书认证；`default` 指向本地 Web 的 8443，避免把普通 HTTPS 误送进 853。

怎么验证：

```
nginx -t
openssl s_client -connect 192.0.2.1:443 -servername resolver.example.net \
  -CAfile /etc/ssl/certs/ca-certificates.crt </dev/null 2>/dev/null \
  | grep -E 'Verify return code|subject='
kdig -d @192.0.2.1 -p 443 +tls-ca +tls-hostname=resolver.example.net example.org A
```

失败模式 / 回滚点：DoT 客户端**不一定发送 SNI**（协议并不强制），一旦客户端省略 SNI，`$ssl_preread_server_name` 为空，查询会被送到 Web 后端，表现为「偶尔 TLS 握手成功但 DNS 无响应」。这类间歇故障极难查。回滚：退回独占 853，或在 443 上改用 DoH（第 12 章），DoH 的 HTTP 层天然带 `:authority`，不依赖 SNI 分流。

11.5.1　SNI 与隐私的边界

SNI 是明文的：即使 DNS 报文被 TLS 加密，ClientHello 里的 `servername` 仍会向链路上的观察者暴露「客户端在连哪台解析器」。因此 SNI 不能作为隐私增强手段，它只是**分流与虚拟主机**的机制。想减少这类元数据，可考虑：客户端不配 SNI 而靠 IP + 认证名认证（但会破坏上面的分流方案）、或使用 ECH 一类机制（超出本章范围）。

更实际的边界是：**SNI 暴露的是「你连了谁」，不是「你查了什么」**。查询内容在 TLS 内，QNAME 不外泄给链路观察者；但查询量、时序、报文长度仍是可观测的元数据，[RFC 7830] 的 EDNS(0) padding 与 [RFC 9230] 的相关机制可用于对齐长度（协议层对比见第四篇第 17 章）。

11.6 连接复用、TCP fastopen 与 MTU / EDNS 1232

11.6.1 连接复用是 DoT 的性能前提

TLS 握手 + TCP 三次握手的成本远高于一次 UDP 查询。如果每条查询都新建连接，DoT 的延迟会退化到不可接受。因此 DoT 依赖**长连接 + 多查询复用**，相关参数在官方手册里是一组 TCP 选项：

```
# /etc/unbound/unbound.conf.d/20-tcp-tuning.conf
server:
    tcp-idle-timeout: 30000         # 毫秒，空闲连接保持 30 秒
    tcp-reuse-timeout: 60000        # 毫秒，出向持久化连接保持 60 秒
    max-reuse-tcp-queries: 200      # 单条连接最多承载 200 次查询
    edns-tcp-keepalive: yes
    edns-tcp-keepalive-timeout: 120000
    incoming-num-tcp: 50
    outgoing-num-tcp: 50
    so-reuseport: yes
    stream-wait-size: 4m
    http-nodelay: yes
```

为什么这么设：`tcp-idle-timeout` 管入向（客户端连到我们 853）、`tcp-reuse-timeout` 管出向（我们连权威服务器），两者常被混为一谈；`max-reuse-tcp-queries` 防止单条连接无限占用工作线程，达到上限后重建。`edns-tcp-keepalive` 启用后，`edns-tcp-keepalive-timeout` 会覆盖 `tcp-idle-timeout`，由服务端把建议的空闲时长告诉客户端，避免「客户端以为连接还在、服务端已经关了」的半开状态。`stream-wait-size` 限制 TCP/TLS 流上排队应答的总内存，是 DoT/DoH 高并发下的内存护栏。

怎么验证：

```
unbound-checkconf /etc/unbound/unbound.conf.d/20-tcp-tuning.conf
ss -tnp 'sport = :853' | head -20
kdig @192.0.2.1 +tls-ca +tls-hostname=resolver.example.net +keepopen \
     example.org A example.net AAAA example.com TXT
unbound-control stats_noreset | grep -E 'num.query.tcp|num.query.tcpout'
```

`ss` 里同一条连接应承载多次查询；`+keepopen` 的三次查询在抓包里应只出现一次 TLS 握手。

失败模式 / 回滚点：把 `tcp-idle-timeout` 设得极短（例如 200ms）会让移动端频繁重握手、耗电且延迟抖动；设得极长则连接表被空闲客户端占满，官方手册说明在空闲缓冲低于阈值时会**自动逐级缩短**这个值，所以「参数没生效」不一定是配置错。回滚：恢复默认值（30s / 60s / 200 次），观察 `incoming-num-tcp` 是否触顶。

11.6.2 TCP fastopen 的适用范围

TCP Fast Open（TFO）允许在 SYN 里带数据，省掉一次 RTT。它对「短连接、每次重连」收益明显，对本章的长连接复用场景收益有限，且依赖内核与中段网络不丢弃带数据的 SYN。客户端诊断可用 `kdig +fastopen` 观察，服务端侧是否启用要看系统与 Unbound 编译/平台支持，不能假定「配了就一定生效」。

```
# 客户端侧诊断：显式要求 TFO，并用 -d 看协商细节
kdig -d @192.0.2.1 +tls-ca +tls-hostname=resolver.example.net +fastopen \
     example.org A
sysctl net.ipv4.tcp_fastopen
```

为什么这么设：给出可观测的命令而不是「建议开启」，是因为在跨境链路上带数据的 SYN 被中间设备丢弃的概率不低（UDP/QUIC 被 QoS 的实测对比见第 14 章）。在已经复用长连接的前提下，TFO 省下的一次 RTT 通常不如把连接保持住来得实在。

怎么验证：`sysctl` 返回非 0 仅说明内核侧允许；真正是否用了 TFO 要靠 `kdig -d` 的输出与抓包中的 TCP Fast Open 选项判断。

失败模式 / 回滚点：TFO 引发的连接异常往往表现为「偶发首次查询超时、重试后正常」，很容易被误判成上游抖动。回滚：`+fastopen` 去掉即可，Unbound 侧不必改动。

11.6.3　MTU 与 EDNS 1232

`edns-buffer-size` 与 `max-udp-size` 的默认值都是 **1232**（DNS Flag Day 2020 建议值）。这个数字不是随手取的：IPv6 最小 MTU 是 1280，扣掉 40 字节 IPv6 头与 8 字节 UDP 头，剩 1232；这样即便链路上 MTU 只有 1280（例如 WireGuard 隧道、PPPoE 1492 上的封装叠加）也不会 IP 分片。历史上常见的 `edns-buffer-size: 1472` 在以太网 1500 下看似安全，一旦套上隧道就分片，而分片在很多网络里被直接丢弃，表现为「等超时后重试 TCP」。

```
# /etc/unbound/unbound.conf.d/30-edns.conf
server:
    edns-buffer-size: 1232    # 出向：通告给对端的 EDNS 重组缓冲
    max-udp-size: 1232        # 入向：允许返回给客户端的最大 UDP 应答
```

为什么这么设：两个参数方向不同，`max-udp-size` **只管入向应答**，出向上游看到的是 `edns-buffer-size` 通告的值。DoT 走 TCP/TLS，本身没有 UDP 分片问题，但**服务端自己向权威服务器发出的迭代查询仍有 UDP 段**，所以这两个参数在 DoT 服务端上照样要设对；否则客户端会看到「DoT 连上了，但某些大应答超时后变慢」。

怎么验证：

```
dig @127.0.0.1 -p 5353 +bufsize=4096 example.org A | grep -E 'EDNS|MSG SIZE'
tcpdump -ni any 'ip[6:2] & 0x1fff != 0 or ip6[6] = 44' -c 10   # 观察分片
ip link show wg0 | grep mtu || true
```

`dig` 输出里的 `udp: 1232` 证明入向被限制住了；抓分片只是为了确认链路上没有碎片。

失败模式 / 回滚点：把 `edns-buffer-size` 降到 512 能绕过最严苛的 MTU，但会制造大量 TC=1 与 TCP 重试，官方手册称其为「极端值」。回滚：改回 1232，并检查隧道 MTU（`ip link show`）而不是只调 DNS 参数。

11.7　一张验收清单

检查项	期望	命令
853 监听	仅 TCP，无 UDP	`ss -lntp 'sport = :853'`
证书链	`Verify return code: 0`	`openssl s_client -connect 192.0.2.1:853 ...`
认证名	SAN 与 `# 名字`一致	`openssl x509 -in server-chain.pem -noout -text | grep -A1 'Subject Alternative'`
端到端 DNS	NOERROR + ANSWER	`kdig @192.0.2.1 +tls-ca +tls-hostname=... example.org A`
单上游	恰好一条 `forward-addr`	`unbound-control list_forwards`
无明文出向	抓包无 `dst port 53`	`tcpdump -ni any 'dst port 53' -c 20`
fail-closed	断 853 后 SERVFAIL	断链演练
EDNS	出向 / 入向均 1232	`dig +bufsize=4096 ... | grep udp:`
连接复用	一次握手多次查询	`kdig +keepopen` + 抓包

DoT 保护的是**客户端↔解析器这一段链路**。服务端自己向权威服务器发出的迭代查询通常仍是明文 53，除非把递归放到自己可控的机器上（第 6 章的本地根区镜像、第 14 章的加密隧道）。「开了 853」不等于「整条解析路径都加密了」，也不等于「应答数据可信」——后者是 DNSSEC 的职责。

11.8　本章要点

- 启用服务端 DoT 需要三件事同时到位：`tls-service-key` + `tls-service-pem`（完整链）、`tls-cert-bundle`（出向认证用 CA）、以及 `interface: 0.0.0.0@853` / `tls-port: 853`；`@853` 的端口不再提供普通 DNS TCP。证书文件在 chroot 外或仅 root 可读时，换证必须 **restart** 而非 reload。
- 公共 CA（Let’s Encrypt）省去客户端分发，但要域名与续期钩子；自签 CA 把信任集缩到自己手里，客户端 `tls-cert-bundle` 必须指向该 CA，SPKI pin 适合诊断客户端而非 Unbound 的 `forward-addr`。

- 客户端 `forward-addr: 192.0.2.53@853#resolver.example.net` 里 `@` 指定端口、`#` 指定认证名；**不写 `#` 时任何名字都被接受**，是静默的认证退化。`forward-tls-upstream: yes` 与 `forward-first: no` 一起构成 fail-closed，且只留一条上游。
- 验证要分两层：`openssl s_client` 验 TLS 链与 SNI，`kdig +tls-ca +tls-hostname` 验端到端 DNS；`+tls` 机会式模式只能用来判断端口连通性，不作验收。
- nginx `stream` + `ssl_preread` 可按 SNI 让 443 与 DoT 共存，但 DoT 客户端不强制发 SNI，省略 SNI 会被分流到 Web 后端；SNI 本身明文，暴露「连了谁」而非「查了什么」。
- DoT 性能依赖长连接复用（`tcp-idle-timeout` 管入向、`tcp-reuse-timeout` 管出向、`max-reuse-tcp-queries` 限单连接次数）；`edns-buffer-size` / `max-udp-size` 默认 1232 是 IPv6 最小 MTU 1280 的推导结果，1472 在隧道上分片会被丢。

11.9 参考资料

- Unbound 官方手册, `unbound.conf(5)`: `tls-service-key`、`tls-service-pem`、`tls-cert-bundle`、`forward-addr` 的 `#` 认证名与 TCP 选项: https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html
- Unbound 官方手册，DNS-over-HTTPS / TLS 相关主题页（下游加密服务与 HTTP/2 要求）：https://unbound.docs.nlnetlabs.nl/en/latest/topics/privacy/dns-over-https.html
- Unbound 官方手册,Performance Tuning(`so-reuseport`、线程与缓冲区):https://unbound.docs.nlnetlabs.nl/en/latest/topics/core/performance.html
- NLnet Labs，Unbound 项目文档入口：https://nlnetlabs.nl/documentation/unbound/
- RFC 7858, *Specification for DNS over Transport Layer Security (TLS)*: https://www.rfc-editor.org/rfc/rfc7858
- RFC 8310，*Usage Profiles for DNS over TLS and DNS over DTLS*（机会式与带外 pin 认证模型）：https://www.rfc-editor.org/rfc/rfc8310
- RFC 7766，*DNS Transport over TCP - Implementation Requirements*（连接复用与流水线）：https://www.rfc-editor.org/rfc/rfc7766
- RFC 7830，*The EDNS(0) Padding Option*: https://www.rfc-editor.org/rfc/rfc7830
- OpenSSL 官方手册，`openssl-s_client(1)`：https://www.openssl.org/docs/man3.0/man1/openssl-s_client.html
- Knot DNS 官方手册，`kdig(1)`（`+tls-ca`、`+tls-pin`、`+fastopen`、`+keepopen`）：https://www.knot-dns.cz/docs/3.3/html/man_kdig.html

第十二章

DoH：与 Web 服务共存的加密解析

DoH（DNS-over-HTTPS，[RFC 8484]）把 DNS 报文塞进 HTTP 请求体，再经 TLS 送出，因此能跑在标准 443 端口上，与 Web 流量同形。这一形态带来两个必须一开始就讲清的事实：**Unbound 本身不是给公网做 DoH 前端的合适组件**，以及 **DoH 端点仍属第三方，日志留存方是谁要写进设计文档**。本章按「前端→后端→客户端→绕过面」的顺序落地配置。

12.1 为什么 Unbound 不是 DoH 前端

Unbound 从 1.12.0 起提供下游 DoH 能力，但它要求编译时带 `libnghttp2`，**只支持 HTTP/2**，且实现的是固定的 `/dns-query` 端点（`http-endpoint` 可改名），媒体类型只有 `application/dns-message`。官方手册返回的状态码是协议层的：404（路径不匹配）、413/414（payload 过大）、415（media type 不对）、400（畸形请求）。这套能力用于「本机或内网里让 HTTPS 客户端能接入」是合适的，放到公网边缘就缺了几样东西：

公网前端要做的事	Unbound 下游 DoH	dnsdist	nginx
同一 443 上按路径分流 Web / DoH	只能独占标记为 `https-port` 的端口	需要反代配合	原生 `location`
自定义 Header 校验（Bearer / token）	不支持	可用 HTTP 规则	原生 `map` / `auth_request`
按路径、按 SNI、按客户端做限速与灰度	无	规则链完整	限速模块完整
HTTP/1.1 回源兼容	只支持 HTTP/2	2.x 起 nghttp2 只支持 h2	回源可 h2c / grpc
访问日志归属与脱敏	只有 DNS 层日志	有 HTTP 状态码计数	有 access log，可裁字段
与已有 Web 证书、ACME 续期复用	需单独管理证书	可共用	天然共用

结论不是「Unbound 不能做 DoH」，而是**分工**：Unbound 做后端的递归 / 缓存 / DNSSEC 验证；dnsdist 做 DNS 感知的 DoH 前端与分流；nginx 做与既有 Web 服务共端口的反代与认证。三者组合时，公网只暴露 443，Unbound 只监听回环。

```
# /etc/unbound/unbound.conf.d/10-backend.conf
# 教学示例：Unbound 只做后端，绝不直接监听公网 443
server:
    interface: 127.0.0.1@5353
    interface: ::1@5353
    access-control: 127.0.0.0/8 allow
    access-control: ::1 allow
    access-control: 0.0.0.0/0 refuse
    edns-buffer-size: 1232
    max-udp-size: 1232
    qname-minimisation: yes
    qname-minimisation-strict: no
```

```
    harden-glue: yes
    module-config: "validator iterator"
    auto-trust-anchor-file: "/var/lib/unbound/root.key"
```

为什么这么设：后端只听回环，公网侧的 TLS 终止、路径校验、认证、限速全部由前端承担；后端没有公网攻击面，也不需要证书。`module-config` 里保留 `validator`，是因为 DNSSEC 的三种状态（无信任锚 / permissive / 真验证）由后端决定，前端只搬运 DNS 报文，不能替它验证（第 10 章）。

怎么验证：

```
unbound-checkconf /etc/unbound/unbound.conf.d/10-backend.conf
dig @127.0.0.1 -p 5353 example.org A +dnssec +short
ss -lntp | grep 5353
```

失败模式 / 回滚点：如果把后端也配上 `interface: 0.0.0.0@443` + `tls-service-key`，就等于绕过了前端的所有认证与日志策略。回滚：`ss -lntp` 确认 443 只被 nginx/dnsdist 占用，注释掉多余的 `interface` 行 restart（`interface` 变更 reload 不生效）。

12.2　dnsdist：`addDOHLocal` + `newServer(pool)` 的分流

dnsdist 是 DNS 感知的负载均衡器，适合放在 Unbound 前面：它终止 DoH 的 HTTP/2，把 DNS 报文解出来，按规则选后端池，再以普通 DNS（或 DoH / DoT）发给后端。下面的例子给出「按 QNAME 后缀分流到不同池」的完整形态。

```
-- /etc/dnsdist/dnsdist.conf ——教学示例，地址均为 RFC 5737 文档段
setLocal("127.0.0.1:5300")                    -- 控制台/明文 DNS 入口（可选）

-- 后端池：default 走本机递归验证器；internal 走内网权威/转发器
newServer({address="127.0.0.1:5353", pool="default", name="unbound-recursive"})
newServer({address="198.51.100.53:53", pool="internal", name="lab-forwarder"})

-- DoH 前端：公网 443，路径带 token 前缀（媒体类型与路径校验见后文）
addDOHLocal("192.0.2.10:443",
  "/etc/dnsdist/tls/fullchain.pem",
  "/etc/dnsdist/tls/privkey.pem",
  {"/dns-query/<TOKEN>"},
  {reusePort=true, keepIncomingHeaders=true})

-- 按 QNAME 后缀选池：内部域及其子域进 internal，其余进 default
addAction(QNameSuffixRule("internal.example.org."), PoolAction("internal"))
addAction(AllRule(), PoolAction("default"))

-- 控制面与观测
controlSocket("127.0.0.1:5199")
setKey("<TOKEN>")
addAction("use-application-dns.net.", RCodeAction(DNSRCode.NXDOMAIN))
```

为什么这么设：`addDOHLocal` 的第四参数是接受查询的 URL 路径列表，这里把 `<TOKEN>` 放进路径，是因为 iOS 的 `DNSSettings` 只能配 `ServerURL`、不能加自定义请求头（12.5 节）；`keepIncomingHeaders=true` 让 `HTTPHeaderRule()` 与 `DNSQuestion:getHTTPHeaders()` 可用，便于审计 Header。`newServer` 的 `pool` 把后端分组，`PoolAction` 按 QNAME 把查询投进对应池——这就是「分流」在 DoH 前端的形态，与第 14 章的 nft 策略分流是不同层次。`QNameSuffixRule` 匹配后缀及其子域（`QNameRule` 只精确匹配一个名字），`AllRule()` 做兜底，`RCodeAction(DNSRCode.NXDOMAIN)` 在本地把浏览器 canary 域压成 NXDOMAIN（12.6 节）。

注意 `PoolAction` 的 `stop` 默认为 `true`：命中第一条规则后不再处理后续规则，所以能用作兜底的 `AllRule()` 必须放在最后。未指定 `pool` 的后端会落在默认池（空字符串 `''`）；上面把所有后端都显式命名，避免「规则没命中就落到空池」的歧义。实际部署时应按所用 dnsdist 版本的 `setPoolServerPolicy` / `PoolAction` 语义核对，见官方手册对应章节。

怎么验证：

```
dnsdist --check-config
curl -sS --http2 -H 'content-type: application/dns-message' \
  --data-binary @query.bin "https://dns.example.org/dns-query/<TOKEN>" -o reply.bin -w '%{http_code}\n'
```

```
dnsdist -c 127.0.0.1:5199 -k '<TOKEN>' -e 'showDOHResponseCodes()'
dnsdist -c 127.0.0.1:5199 -k '<TOKEN>' -e 'showServers()'
```

showDOHResponseCodes() 若出现大量 404，先查路径与 addDOHLocal 的 urls 是否完全一致；出现 415 则是 content-type 没设对。query.bin 可以用 kdig +https 生成，或先用一个已知的 DNS 线格式报文。

失败模式 / 回滚点：dnsdist 2.x 起用 nghttp2 实现入向 DoH，**只支持 HTTP/2，不支持 HTTP/1.1**（官方手册明确写了）。如果前面再加一层 nginx 且回源用 HTTP/1.1，会直接握手失败；回源写法见后文。另一个坑是证书/私钥权限：dnsdist 通常以非 root 运行，私钥必须对它可读，否则 addDOHLocal 启动时报错。回滚：把 addDOHLocal 注掉、保留 addLocal 明文入口，先恢复 DNS 可用性，再逐项加回。

12.2.1 生成一次可复用的 DoH 请求体

```
# 用 kdig 直接发 DoH，省去手工构造线格式报文
kdig -d @dns.example.org +https=/dns-query/<TOKEN> \
     +tls-ca +tls-hostname=dns.example.org example.org A

# 或明确 content-type 用 curl POST 一个二进制 DNS 报文
printf '\xab\xcd\x01\x00\x00\x01\x00\x00\x00\x00\x00\x00\x07example\x03org\x00\x00\x01\x00\x01' \
  > /tmp/query.bin
curl -sS --http2-prior-knowledge -X POST \
  -H 'content-type: application/dns-message' \
  -H 'accept: application/dns-message' \
  --data-binary @/tmp/query.bin \
  "https://dns.example.org/dns-query/<TOKEN>" -o /tmp/reply.bin -w '%{http_code}\n'
```

为什么这么设：[RFC 8484] 规定 DoH 的请求与应答媒体类型都是 application/dns-message，GET 用 base64url 的 dns 参数、POST 用请求体承载原始 DNS 线格式。用 curl 手工发一次，可以把「HTTP 层问题」与「DNS 层问题」分开：400/404/415 是 HTTP 层，200 但应答里 SERVFAIL 才是 DNS 层。

怎么验证：%{http_code} 应为 200；kdig 输出里应有 status: NOERROR 与 ANSWER SECTION。

失败模式 / 回滚点：curl 不带 --http2 时默认用 HTTP/1.1，遇到只支持 h2 的前端会失败；这不是服务端挂了。回滚：加 --http2 或改用 kdig +https。

12.3 nginx 反代：grpc_pass / proxy_pass 与 h2c 的坑

已有 nginx 在 443 上跑 Web 时，最省事的是给 DoH 加一个 location，把 /dns-query 反代给本机 dnsdist。这里要处理两个独立的坑。

12.3.1 坑一：nginx 回源不是 HTTP/2

nginx 的 proxy_pass 到上游默认走 **HTTP/1.1**（proxy_http_version 1.1 是默认值），而 dnsdist 2.x 的入向 DoH 只支持 h2。官方手册给出的绕行方案是用 grpc_pass：gRPC 本身就要求 HTTP/2，nginx 的 grpc_pass 会向上游发起 h2（或 h2c）连接，从而满足 dnsdist 的要求。

```
# /etc/nginx/conf.d/doh.conf ——教学示例
server {
    listen 443 ssl http2;
    server_name dns.example.org;

    ssl_certificate     /etc/nginx/tls/fullchain.pem;
    ssl_certificate_key /etc/nginx/tls/privkey.pem;

    # 路径里带 token：iOS 描述文件只能配 ServerURL，不能加自定义头
    location ~ ^/dns-query/(?<doh_token>[A-Za-z0-9_-]+)$ {
        if ($doh_token != 'example-token-0000') { return 403; }

        # 用 grpc_pass 让回源走 HTTP/2（dnsdist 2.x 不收 HTTP/1.1）
        grpc_pass grpcs://127.0.0.1:8443;
        grpc_set_header Host $host;
        grpc_set_header X-Real-IP $remote_addr;
```

```
        grpc_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        access_log /var/log/nginx/doh.access.log combined;
    }

    # Bearer 校验形态（客户端能加自定义头时用）
    location /dns-query {
        if ($http_authorization != 'Bearer <TOKEN>') { return 403; }
        proxy_http_version 1.1;
        proxy_set_header content-type 'application/dns-message';
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_pass http://127.0.0.1:8053/dns-query;
    }

    location / {
        return 404;
    }
}
```

为什么这么设：两条 `location` 演示两种认证形态。第一条把 token 放在**路径**里，用 nginx 的正则捕获并比对，适用于「客户端只能配 URL」的场景；第二条用 `Authorization: Bearer <TOKEN>`，适用于浏览器扩展、`curl`、自写客户端等能加头的场景。`grpc_pass grpcs://` 对应 dnsdist 侧仍带 TLS 的 DoH 监听；若回源是明文 h2c，则写 `grpc_pass grpc://127.0.0.1:8053`（dnsdist 的 `addDOHLocal("127.0.0.1:8053", nil, nil, ...)` 不提供证书即为明文 DoH）。

要注意 `grpc_pass` 是给 gRPC 用的：它把请求按 HTTP/2 送到上游，但**不会替你做 DoH 的媒体类型转换**。因此这个方案只适用于上游能够接受该形态的 DoH 实现；若上游是 Unbound 自身的下游 DoH（同样只支持 h2），也应先用一个最小请求验证，而不是假定「换成 grpc_pass 就通了」。

怎么验证：

```
nginx -t
curl -sS --http2 -o /dev/null -w '%{http_code}\n' -H 'content-type: application/dns-message' \
  --data-binary @/tmp/query.bin 'https://dns.example.org/dns-query/example-token-0000'
curl -sS --http2 -o /dev/null -w '%{http_code}\n' \
  -H 'authorization: Bearer <TOKEN>' -H 'content-type: application/dns-message' \
  --data-binary @/tmp/query.bin 'https://dns.example.org/dns-query'
```

第一条应 200，第二条在 token 正确的前提下也应 200；把 token 改错，两条都应 403。

失败模式 / 回滚点：最典型的错是把 `proxy_pass` 直接指向 dnsdist 的 DoH 端口，nginx 以 HTTP/1.1 回源→dnsdist 直接拒绝，日志里既没有 DNS 查询也没有 200，只有连接层错误。第二个错是 `grpc_pass` 与 `proxy_pass` 写在同一 `location` 里，nginx 配置检查会报错。回滚：先用 `curl --http2-prior-knowledge http://127.0.0.1:8053/dns-query` 直连 dnsdist 确认回源可用，再接上 nginx。

12.3.2　坑二：h2c 与「明文 HTTP/2」

h2c（HTTP/2 over cleartext TCP）不使用 TLS，靠 HTTP/1.1 Upgrade 或先验知识（prior knowledge）建立 h2 连接。它在「nginx →本机 dnsdist」这一段很常见，因为公网侧 TLS 已在 nginx 终止，回源再加密一次意义不大。但有两个必须写明的边界：

1. `proxy_http_version 1.1` 不会升级成 h2；`proxy_pass` 无法直接回源 h2c。要么用 `grpc_pass grpc://`（h2 prior knowledge），要么让 dnsdist 侧接受 HTTP/1.1（2.x 已不支持）。
2. h2c 是**明文**，因此只在回环或严格隔离的内网使用；跨主机回源要走 TLS 或另加隧道。客户端到 nginx 的那段仍是 HTTPS，不受影响。

```
# 回源为明文 h2c 时的最小形态（与 12.3.1 二选一）
location /dns-query {
    if ($http_authorization != 'Bearer <TOKEN>') { return 403; }
    grpc_pass grpc://127.0.0.1:8053;
}
```

为什么这么设：`grpc://` 明确不加密，`grpcs://` 明确加密，两者不能混写；dnsdist 侧对应 `addDOHLocal("127.0.0.1:8053", nil, nil, "/dns-query")` 这种不传证书的形态。

怎么验证：

```
nginx -t && nginx -s reload
curl -sS --http2-prior-knowledge -o /dev/null -w '%{http_code}\n' \
  -H 'content-type: application/dns-message' --data-binary @/tmp/query.bin \
  -H 'authorization: Bearer <TOKEN>' http://127.0.0.1:8053/dns-query
```

直连回源的 200 证明 h2c 通路成立；再经 443 测一次，区分「回源问题」与「公网侧问题」。

失败模式 / 回滚点：把 grpc:// 写成 grpcs://（或反过来）会表现为 TLS 握手失败或明文被当成 TLS，日志里通常是 upstream 连接错误。回滚：统一前后端的加密形态，并保留 nginx -t 通过前的配置备份。

12.4 /dns-query 的 content-type 与端点语义

[RFC 8484] 把 DoH 的请求/应答媒体类型定为 application/dns-message，路径惯例是 /dns-query。Unbound 下游 DoH 的默认 http-endpoint 也是 /dns-query，dnsdist 自 1.5.0 起默认同样如此。几个容易踩的细节：

- 请求没有 content-type 头时，Unbound 会**当作** application/dns-message 处理；但只要带了任何其他值就返回 415。因此「漏设头」和「设错头」表现不同，排查时不能混为一谈。
- GET 用 ?dns=<base64url>，URL 过长会 414；POST 用请求体，过长会 413。移动端偶发失败时，先看是不是走了 GET 且查询被 EDNS 选项撑大。
- 路径不匹配一律 404；把 token 放路径里时，addDOHLocal 的 urls、nginx 的 location、描述文件里的 ServerURL 三处必须**完全一致**。
- DoH 端点仍属第三方：即使查询内容加密，**TLS 终止方（nginx / dnsdist 所在主机）能看到完整 HTTP 层信息**——路径、Header、客户端 IP、时序、查询量。DNS 层则由后端 Unbound 看到完整 QNAME。两处的日志留存方都要写清楚。

```
# 一次最小验收：分别触发 200 / 415 / 404
curl -sS --http2 -o /dev/null -w 'ok=%{http_code}\n' \
  -H 'content-type: application/dns-message' --data-binary @/tmp/query.bin \
  'https://dns.example.org/dns-query/example-token-0000'
curl -sS --http2 -o /dev/null -w 'wrong-type=%{http_code}\n' \
  -H 'content-type: application/json' --data-binary @/tmp/query.bin \
  'https://dns.example.org/dns-query/example-token-0000'
curl -sS --http2 -o /dev/null -w 'wrong-path=%{http_code}\n' \
  -H 'content-type: application/dns-message' --data-binary @/tmp/query.bin \
  'https://dns.example.org/other'
```

为什么这么设：这三条把端点行为固定成可回归的断言。以后换证书、改路径、加一层反代，重跑这三条就能定位到 HTTP 层的哪一段变了。

怎么验证：期望依次 ok=200、wrong-type=415（或前端自定义 403/415，取决于哪一层先拦）、wrong-path=404。

失败模式 / 回滚点：如果 415 变成 200，说明中间有组件把 content-type 改写了；如果 404 变成 200，说明 location 匹配范围过宽，token 校验被绕过。后者是安全问题，必须立刻收紧正则。

12.5 iOS / Android 描述文件：ServerURL 与认证限制

移动端配 DoH 有两种常见途径：MDM 下发的 com.apple.dnsSettings.managed 描述文件（iOS/iPadOS 14+），以及 Android 的「私人 DNS / Private DNS」设置。二者的能力边界不同，直接决定了 token 该放哪。

12.5.1 iOS：DNSSettings 只能配 ServerURL

Apple 的 DNSSettings 描述文件里，DoH 由 ServerURL 指定「URI 模板」（[RFC 8484] 语义），并要求 https:// scheme；系统用 URL 里的主机名或地址去验证服务器证书。同一个字典里还有 ServerName（DoT 的主机名，按 [RFC 7858] 校验证书）、ServerAddresses、SupplementalMatchDomains、PayloadCertificateUUID（用身份证书向解析器认证客户端）等键。关键限制是：**描述文件里没有「自定义 HTTP 请求头」这一项**。因此 Authorization: Bearer <TOKEN> 在 iOS 上配不出来，唯一能由我们控制、又能随请求带上的字符串就是 URL 本身。

```
<!-- 教学示例：MDM 下发的 com.apple.dnsSettings.managed 片段 -->
<dict>
  <key>PayloadType</key><string>com.apple.dnsSettings.managed</string>
```

```
  <key>PayloadIdentifier</key><string>example.org.doh.lab</string>
  <key>PayloadUUID</key><string>00000000-0000-0000-0000-000000000001</string>
  <key>PayloadVersion</key><integer>1</integer>
  <key>DNSSettings</key>
  <dict>
    <key>DNSProtocol</key><string>HTTPS</string>
    <key>ServerURL</key>
    <string>https://dns.example.org/dns-query/example-token-0000</string>
    <key>SupplementalMatchDomains</key>
    <array/>
  </dict>
</dict>
```

为什么这么设：ServerURL 里带 token 前缀路径，等价于把「认证」折进 URL；SupplementalMatchDomains 留空数组表示所有域都走该解析器。注意 Apple 文档注明：从 MDM 安装时设置只作用于受管 Wi-Fi 网络，手工安装时才同时作用于蜂窝网络——这会影响「全屋覆盖」的验收结论。此外该 payload 在新版本系统中已被标记为 deprecated（改用声明式管理配置），落地时要按目标系统版本核对官方手册对应章节，不要照抄旧键名。

怎么验证：

```
# 服务端侧看请求路径是否带上了 token、有没有 403
grep 'dns-query' /var/log/nginx/doh.access.log | tail -20
# 客户端侧：装好描述文件后访问一个测试域，确认解析走的是自建端点
kdig +https=/dns-query/example-token-0000 @dns.example.org example.org A
```

失败模式 / 回滚点：token 放路径意味着它会进 nginx access log、CDN 日志、以及任何做 URL 记录的中间设备——**这不是秘密，只是凭据**，必须按凭据管理：定期轮换、日志里脱敏、不要复用同一 token 给所有设备。另一个坑是 URL 里的 {?dns} 模板语法与自定义路径混写，系统实际请求路径与 addDOHLocal 的 urls 不一致→ 404。回滚：先用不带 token 的 /dns-query 确认连通，再加路径段。

12.5.2　Android：Private DNS 与 DoH 的边界

Android 的「私人 DNS」在用户层面配置的是一个 **DoT 主机名**（系统按 [RFC 7858] 用它做证书校验），而不是任意 DoH URL；较新版本的解析模块支持 DoH 的能力由系统实现与版本决定，配置面并不等同于 iOS 的 ServerURL。因此在本书的口径下：**要给 Android 一个可控的加密端点，优先用第 11 章的 DoT 853，而不是假定描述文件能配 DoH URL**。

```
# 教学示例：Android Private DNS 里填的是 DoT 主机名，不是 URL
#   resolver.example.net
# 服务端验证该名字能被系统解析器解析、且 853 证书 SAN 覆盖它
openssl s_client -connect 192.0.2.1:853 -servername resolver.example.net \
  -CAfile /etc/ssl/certs/ca-certificates.crt </dev/null 2>/dev/null | grep 'Verify return code'
```

为什么这么设：把两种机制分开写，避免把「iOS 描述文件的 ServerURL」误套到 Android 上。DoT 主机名形态还带来一个好处：它不依赖 HTTP 层的 token 路径，认证完全由证书与主机名校验完成。

怎么验证：Verify return code: 0 (ok)；再在设备上确认「私人 DNS」状态为已验证而非「无法连接」。

失败模式 / 回滚点：若填的是带 https:// 的 URL，设备会把它当主机名做 DNS 查询与证书校验，必然失败。回滚：改回纯主机名，并确保该名字在系统当前 DNS 下可解析（自签 CA 场景需要先装 CA）。

12.6　浏览器 canary 域：让内置 DoH 主动关闭

自建 DoH 端点最容易被绕过的方式，不是链路被破解，而是**浏览器根本没用你的解析器**。Firefox 与 Chrome 各自有「是否启用内置 DoH」的判断机制，理解它们的差异比按 IP 封 443 有效得多。

12.6.1　Firefox：use-application-dns.net 回 NXDOMAIN

Firefox 在启用内置 DoH 前会查询 canary 域 use-application-dns.net.。如果本地解析器对该名字返回 NXDOMAIN（或该查询被阻断导致无应答），Firefox 就不启用内置 DoH，改用系统解析器——于是流量回到我们可控的链路上。

```
# /etc/unbound/unbound.conf.d/40-canary.conf
server:
    local-zone: "use-application-dns.net." always_nxdomain
    local-zone: "_dns.resolver.arpa." always_nodata
```

为什么这么设：always_nxdomain 忽略该区下的 local-data，保证整区一律 NXDOMAIN，不会因为某条遗留白名单而漏过；_dns.resolver.arpa. 的 always_nodata 是 DDR 压制（第 13 章展开），两者配合覆盖「浏览器自动发现加密解析器」这条绕过路径。

怎么验证：

```
dig @127.0.0.1 -p 5353 use-application-dns.net A +short
dig @127.0.0.1 -p 5353 use-application-dns.net A | grep 'status:'
dig @127.0.0.1 -p 5353 _dns.resolver.arpa. SVCB +short
```

期望 status：NXDOMAIN，SVCB 查询返回 NODATA（无记录、无错误）。

失败模式 / 回滚点：always_nxdomain 与 static 语义不同——前者忽略 local-data，若日后需要给个别名字留白名单要改用 transparent。另一个坑是只在本机 Unbound 配了 canary，但客户端的 DNS 被 nft 劫持到了别的实例（第 14 章），那台实例没配→ Firefox 照样启用内置 DoH。回滚：确认链路上**每一层**解析器都返回 NXDOMAIN。

12.6.2 Chrome：靠系统解析器匹配已知提供商列表

Chrome 的内置 DoH 升级逻辑是：当系统解析器属于已知的 DoH 提供商列表时，才把该解析器升级为对应的 DoH 端点。指向私有解析器（例如 192.0.2.1、fd00::1 或 192.0.2.1）时，名字不在列表里，Chrome **不会自行升级**，会继续用系统解析器。因此不需要按域名封，也不需要额外 canary 域。

```
# 验证「系统解析器是谁」，而不是去封 443
cat /etc/resolv.conf
dig @127.0.0.1 -p 5353 whoami.example.org A +short
# 观察有没有绕过本地解析器直连外部 DoH 的流量
tcpdump -ni any 'tcp port 443 and (dst net 192.0.2.0/24 or dst net 198.51.100.0/24)' -c 20
```

为什么这么设：这是「白名单匹配」而不是「黑名单封禁」，所以**指向私有解析器天然不升级**。反过来，如果哪天把系统解析器改成了某家公共解析器，Chrome 就可能升级——那时封 IP 也拦不住，因为端点会换 IP、换域名。

怎么验证：结合 12.6.1 的 canary 断言与这里的 tcpdump，确认没有「客户端直接向外部 443 发 DoH」的连接；企业环境还可用浏览器策略显式禁用内置 DoH，见官方策略文档。

失败模式 / 回滚点：**按 IP 封 443 是 denylist 不是证明**。一方面会误伤同 IP 上的正常 Web 服务，另一方面端点换地址就失效，而且它无法告诉你「客户端到底有没有绕过」——只能证明「这几个 IP 不通」。真正要测的是「能不能绕过」：用一台客户端把系统 DNS 改成公共解析器，看浏览器是否升级到内置 DoH，以及本地 canary 是否仍生效。回滚：恢复系统解析器指向自建端点，并复查 canary 域。

12.7 日志留存方是谁：DoH 并不等于「无人记录」

把这一节单独列出来，是因为「加密」最容易被误读成「不留痕」。一次 DoH 查询会在至少三处留下可关联的记录：

位置	能看到什么	谁负责留存	建议
nginx / dnsdist（TLS 终止方）	客户端 IP、User-Agent、URL 路径（含 token）、Header、时序、状态码、报文长度	前端运维	access log 裁掉 token 字段，明确保留期
Unbound 后端（DNS 层）	完整 QNAME / QTYPE、应答、缓存状态；若开 dnstap 则是查询流的完整副本	解析器运维	dnstap 保留期写进配置（例如 30 天），见第 13 章
上游 / 权威服务器	解析器发出的迭代查询（通常仍是明文 53），看到的是**解析器 IP** 而非客户端 IP	第三方	明确「DoH 端点仍属第三方」这一事实
客户端设备 / MDM	描述文件、证书、设备标识	设备管理员	轮换 token、按设备分发

关键结论：**DoH 端点仍属第三方**。哪怕客户端到端点这一段是加密的，端点运营方依然拥有完整的查询元数据；客户端把查询从「本地解析器」搬到「自建 DoH 端点」，本质是把信任从网关搬到了那台前端主机。若前端是同一台家庭网关，信任边界没变；若前端在云主机上，就要把「云主机上谁能读日志」写进威胁模型（第 8 章）。

```
# 审计：确认日志里没有把 token 原样落盘
grep -c 'example-token-0000' /var/log/nginx/doh.access.log || true
# 保留期复核（示例：30 天）
find /var/log/nginx -name 'doh.access.log*' -mtime +30 -print
```

为什么这么设：这两条是自查而不是配置。第一条检查凭据是否泄漏进日志，第二条检查保留期是否真的执行——「写了 30 天」和「logrotate 真的按 30 天删」是两件事。

怎么验证：`grep -c` 应为 0（若前端把 token 放路径，需要在 `log_format` 里用 `map` 把该段替换成 `REDACTED`）；`find` 在稳定运行的系统上不应列出 30 天前的归档。

失败模式 / 回滚点：最常见的失败是「前端日志脱敏做了，dnstap 全量查询流却无限期留着」——抗元数据聚合在自己门口破功。回滚：为 dnstap 单独设保留期与清理任务；日志策略变更要同步通知所有能读日志的人。

12.8　验收清单

检查项	期望	命令
公网端口	只有 443，Unbound 只听回环	`ss -lntp`
媒体类型	200 / 415 / 404 三态正确	12.4 节三条 `curl`
h2 回源	`grpc_pass` 或 h2c，非 HTTP/1.1	`nginx -t` + 直连回源 `curl --http2-prior-knowledge`
token	路径或 Bearer 二选一，校验生效	改错 token 应 403
iOS `ServerURL`	路径与 `addDOHLocal` urls 一致	描述文件 + nginx access log
Android	填 DoT 主机名而非 URL	`openssl s_client -servername`
canary	NXDOMAIN	`dig use-application-dns.net A`
DDR	`_dns.resolver.arpa.` NODATA	`dig _dns.resolver.arpa. SVCB`
日志	token 脱敏、保留期执行	`grep -c` + `find -mtime`

12.9　本章要点

- Unbound 有下游 DoH（1.12.0 起，需 `libnghttp2`、只支持 HTTP/2、只认 `application/dns-message`），但不适合单独做公网前端：**用 dnsdist 做 DNS 感知的 DoH 前端与池分流，用 nginx 做与 Web 共 443 的反代与认证，Unbound 只监听回环做递归验证**。
- `addDOHLocal(地址, 证书, 私钥, {路径}, {选项})` 定义 DoH 监听；`newServer({pool=...})` 把后端分组，`PoolAction` 按 QNAME 分流。dnsdist 2.x 用 nghttp2 实现入向 DoH，**不支持 HTTP/1.1**。
- nginx `proxy_pass` 默认 HTTP/1.1 回源会打不通 dnsdist 2.x；官方手册给出的绕行是 `grpc_pass`（`grpcs://` 加密 / `grpc://` 明文 h2c），且 `grpc_pass` 不负责媒体类型转换，需自行验证端点行为。
- `/dns-query` 的请求与应答媒体类型都是 `application/dns-message`；漏设该头时 Unbound 会当作默认处理，设成别的值返回 415；路径不匹配 404，GET 过长 414、POST 过长 413。token 放路径时三处路径必须完全一致。
- iOS 的 `DNSSettings` 描述文件只有 `ServerURL`（以及 DoT 用的 `ServerName` 等），**不能加自定义请求头**，所以 Bearer 校验用不上，token 必须折进 URL；Android「私人 DNS」填的是 DoT 主机名而非 DoH URL，优先按第 11 章的 853 方案落地。URL 里的 token 是凭据不是秘密，要轮换、要在日志里脱敏。
- Firefox 查 `use-application-dns.net` 回 NXDOMAIN 即关闭内置 DoH；Chrome 靠系统解析器匹配已知提供商列表，指向私有解析器时不升级。**按 IP 封 443 是 denylist 不是证明**，验收要能测出「客户端能否绕过」。
- DoH 端点仍属第三方：TLS 终止方看到 HTTP 元数据，后端看到完整 QNAME，dnstap 是自己的查询流副本。三处日志的留存方与保留期都要写明，否则「抗元数据聚合」在自己门口破功。

12.10　参考资料

- RFC 8484, *DNS Queries over HTTPS (DoH)*: https://www.rfc-editor.org/rfc/rfc8484
- RFC 7858, *Specification for DNS over Transport Layer Security (TLS)*: https://www.rfc-editor.org/rfc/rfc7858

- RFC 9462, *Discovery of Designated Resolvers (DDR)*: https://www.rfc-editor.org/rfc/rfc9462
- Unbound 官方手册, DNS-over-HTTPS 章节（HTTP/2、`http-endpoint`、状态码、`libnghttp2`）: https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html
- Unbound 官方手册, `unbound.conf(5)` 的 `local-zone` 类型（`always_nxdomain` / `always_nodata`）: https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html
- dnsdist 官方手册, `addDOHLocal` / `newServer` 配置参考: https://www.dnsdist.org/reference/config.html
- dnsdist 官方手册, DNS-over-HTTPS 指南（URL 路径、HTTP/1 支持、nginx `grpc_pass` 绕行、outgoing DoH）: https://www.dnsdist.org/guides/dns-over-https.html
- nginx 官方手册,`ngx_http_grpc_module`:https://nginx.org/en/docs/http/ngx_http_grpc_module.html
- nginx 官方手册,`ngx_http_proxy_module`:https://nginx.org/en/docs/http/ngx_http_proxy_module.html
- Mozilla Support, Canary domain `use-application-dns.net`: https://support.mozilla.org/en-US/kb/canary-domain-use-application-dnsnet
- Mozilla Support, 企业网络禁用 DoH 的配置说明: https://support.mozilla.org/en-US/kb/configuring-networks-disable-dns-over-https
- Apple Developer, `DNSSettings` MDM payload (`ServerURL` / `ServerName`): https://developer.apple.com/documentation/devicemanagement/dnssettings

第十三章

查询隐私：QMIN、防降级与 DDR 压制

一台递归解析器向权威侧发出的迭代查询，默认会把完整查询名（QNAME）逐级送出去。查询名最小化（QNAME minimisation，QMIN）要减少的正是这份元数据；防降级要保证加密链路不会被静默剥掉；DDR 压制要保证客户端不会绕过本地解析器直连外部端点。这三条是**同一件事的三个方向**：谁看到了查询、链路有没有被剥掉、查询有没有走别的路。

本章只处理第三篇的「怎么做」：参数、命令、验证与失败模式。QMIN 的协议演进（[RFC 7816] → [RFC 9156]）、DoT/DoH/DoQ 的 RFC 对比放在第四篇第 17 章。

13.1 QMIN 的动机与边界

13.1.1 它抗的是「元数据聚合」，不是「全路径嗅探」

未开 QMIN 时，解析器查 `assets.cdn.example.org A`，向根、TLD、以及该域的权威服务器都会带上完整 QNAME。这些服务器各自只负责命名空间的一部分，本不需要知道完整名字，但完整名字让它们能长期积累「谁在查什么」的元数据。QMIN 的做法是：向每一级只发**推进解析所必需的最少标签数**，并在可能时把 QTYPE 改成 `A`，只在需要最终答案的那一跳才发完整 QNAME 与原始 QTYPE。

```
# /etc/unbound/unbound.conf.d/10-qmin.conf
server:
    qname-minimisation: yes
    qname-minimisation-strict: no
```

为什么这么设：`qname-minimisation: yes` 是官方手册的默认值，语义是**尽力而为（best effort）**——上游若返回 NOERROR 以外的 RCODE，Unbound 会回退成发送完整 QNAME 与原始 QTYPE；`qname-minimisation-strict: no` 保留这个回退（默认值也是 `no`）。这一对组合是「隐私收益」与「兼容性」的平衡点。

怎么验证：

```
unbound-control get_option qname-minimisation
unbound-control get_option qname-minimisation-strict
tcpdump -ni any 'udp port 53 and dst net 192.0.2.0/24' -c 30 -A | grep -i 'example\|cdn' || true
dig @127.0.0.1 -p 5353 assets.cdn.example.org A +short
```

失败模式 / 回滚点：`get_option` 在某些构建里可能不可用，此时以 `unbound-control stats_noreset` 与配置文件为准，不要凭「配了就生效」下结论。抓包里若仍看到完整 QNAME 出现在 TLD 那一跳，先确认查询是不是命中了缓存（命中缓存不出向）。回滚：把 `strict` 明确写成 `no`，别只删掉一行——默认值随版本可能变化。

13.1.2 三条必须写明的边界

1. **QMIN 不是加密**。它只减少发给上级的 QNAME 标签数，报文仍在链路上以明文（或解密后的形式）存在。能观察完整路径的人依旧能看到查询；QMIN 削减的是「每一级权威服务器各自能聚合到多少元数据」。
2. **`harden-below-nxdomain` 只对 DNSSEC-secure 的 NXDOMAIN 生效**。官方手册的 NOTE 明确写了这一点：只有 DNSSEC 验证过的 NXDOMAIN 才会用于剪枝（且 NSEC3 opt-out 不够）。无信任锚、无签名区里的 NXDOMAIN 不触发，所以它在「没有验证器」的部署里是空转，别误判成「已生效」。

3. **本地根区镜像时，根这一跳本来就不出口**。第 6 章的本地根镜像让根区的 SOA / NS / glue 全在本机，因此 QMIN 在根层没有可出口的元数据可减；收益主要落在 **TLD 层**（以及其后的委派层）。这一点决定了「要不要为 QMIN 单独改架构」——通常不需要，它是默认开启的渐进收益。

```
# 边界②的可观测证据：无信任锚时，NXDOMAIN 剪枝不会触发
unbound-control get_option module-config
dig @127.0.0.1 -p 5353 nonexistent.example.org A | grep 'status:'
unbound-control stats_noreset | grep -E 'num.query.aggressive|num.val' || true
```

为什么这么设：把「参数存在」与「机制生效」分开验证。`module-config` 里没有 `validator` 时，DNSSEC-secure 的 NXDOMAIN 无从谈起，剪枝自然不触发。

怎么验证：`status: NXDOMAIN` 只说明名字不存在，不说明剪枝生效；要看 `module-config` 是否含 `validator`、`auto-trust-anchor-file` 是否就位（第 10 章）。

失败模式 / 回滚点：把 `harden-below-nxdomain: yes` 当成「QMIN 的强力版」是常见误读——它管的是「已知 NXDOMAIN 之下不必再问」，与「向每一级发多少标签」是两件事。回滚：二者独立配置，互不为前置。

13.2　`yes` 与 `yes+strict` 的区别

官方手册对 `qname-minimisation-strict` 的定义是：严格模式下**不回退**到向可能有问题的权威服务器发送完整 QNAME，并附了一句很直白的警告——「启用后很多域将无法解析，只在你知道自己在做什么时使用」。

行为	`qname-minimisation: yes`（strict: no）	`qname-minimisation: yes` + `strict: yes`
中间层返回 NXDOMAIN	回退：重新发完整 QNAME 与原始 QTYPE	直接接受 NXDOMAIN，不再回退
中间层返回 REFUSED / SERVFAIL 等非 NOERROR	回退	不回退
兼容性	好，能绕过违反 [RFC 8020] 的区	差，某些域整站解析失败
隐私强度	略有回退，但仍减少大部分标签	最强，无完整 QNAME 外泄
适用	通用递归 / 转发器	受控实验、明确知道上游全部合规

判据可以简化成一句话：**strict 问中间层拿到 NXDOMAIN 就不回退完整域名**，于是「中间层说没有」被当成了「最终答案」。这在严格符合 [RFC 8020]（*NXDOMAIN: There Really Is Nothing Underneath*）的命名空间里是对的：父名 NXDOMAIN 意味着其下所有名字都不存在。问题在于现实里存在不合规的区。

```
# 实验对照：仅用于受控环境，不要直接上生产
server:
    qname-minimisation: yes
    qname-minimisation-strict: yes
```

为什么这么设：只在能承受「部分域解析失败」的实验里开，用来复现下一节的故障；生产环境保持 `strict: no`。

怎么验证：

```
unbound-checkconf /etc/unbound/unbound.conf.d/10-qmin.conf
unbound-control get_option qname-minimisation-strict
dig @127.0.0.1 -p 5353 www.example.org A +short
```

失败模式 / 回滚点：`strict: yes` 的失败不是「某个域查不到」，而是「整站资源解析失败」（见 13.3），且伴随负缓存放大。改配置必须 **restart** 并逐层刷缓存，回滚点就是 `strict: no` 的完整配置备份。

13.3　真实兼容坑：违反 RFC 8020 的 CNAME 区

这是本章最值得记住的一条实测结论。某大型 CDN 的 CNAME 区违反了 [RFC 8020]：对中间层名字返回 **NXDOMAIN**，但其子名（例如带主机名前缀的 CNAME）**确实存在**。按规范这是矛盾的——父名 NXDOMAIN 意味着下面什么都没有。

QMIN 严格模式下的解析过程会这样走：

① 客户端查 assets.cdn.example.org A
② 解析器按 QMIN 只向中间层问 cdn.example.org（QTYPE=A）
③ 该 CDN 的权威服务器对 cdn.example.org 回 NXDOMAIN
④ strict: yes → 不回退完整 QNAME，直接把 NXDOMAIN 当作最终答案
⑤ 客户端收到 NXDOMAIN；页面 HTML 能开（主域已缓存），但 CSS/JS/图片域名全部解析失败

现象很有迷惑性：「网站打开了但样式全丢」「只有静态资源挂」「换个网络就好了」。改回 qname-minimisation-strict: no 之后，第④步会回退成发完整 QNAME，解析恢复。

```
# /etc/unbound/unbound.conf.d/10-qmin.conf ——修复后的配置
server:
    qname-minimisation: yes
    qname-minimisation-strict: no
```

为什么这么设：保留 QMIN 的元数据收益，只放弃「不回退」这一条最激进的约束。这是「隐私 vs 可用性」里最划算的一刀：绝大部分查询仍以最少标签发出，只有遇到非 NOERROR 的那一小部分才回退。

怎么验证：

```
systemctl restart unbound
dig @127.0.0.1 -p 5353 assets.cdn.example.org A +short
dig @127.0.0.1 -p 5353 cdn.example.org A | grep 'status:'
unbound-control stats_noreset | grep -E 'num.query|num.cache' | head -10
```

失败模式 / 回滚点：**改完配置不等于恢复**。原因有两条：

1. **负缓存（negative caching）会放大故障**。第③步的 NXDOMAIN 会按该区 SOA 的 minimum TTL 被缓存（常见 3600 秒）。即使上游修好了、或我们改了 strict，缓存里那条 NXDOMAIN 照样被返回。
2. **链路上每一层缓存都要刷**。第 7 章的多实例链、第 12 章的 dnsdist / nginx 后端、系统存根解析器缓存、浏览器内部缓存，任何一层留着旧的 NXDOMAIN，客户端就仍然失败。只重启最靠近客户端的那一层，是最常见的「改了没用」。

```
# 逐层刷缓存：顺序按「离客户端由近到远」逐台执行，每台都要做
unbound-control -c /etc/unbound/filter.conf flush_zone cdn.example.org
unbound-control -c /etc/unbound/recursive.conf flush_zone cdn.example.org
unbound-control -c /etc/unbound/exit.conf flush_zone cdn.example.org
# unbound-control 关着时只能重启实例；并同步清掉浏览器/系统存根缓存
resolvectl flush-caches 2>/dev/null || true
unbound-control stats_noreset | grep -E 'num.cache|num.query' | head -10
```

为什么这么设：flush_zone 精确刷掉该区及其下的缓存条目，比 flush_all / 整机重启代价小；但它是**每个实例各自的操作**，所以命令按实例配置文件逐个列出。第 7 章已说明链路 A/B/C 各有独立缓存。

怎么验证：dig 恢复 ANSWER SECTION；抓包确认向中间层重新发出了查询（而不是直接命中缓存返回 NXDOMAIN）。

失败模式 / 回滚点：若刷完仍失败，先检查是不是 strict: yes 还留在某个实例的 include 片段里（grep -rn qname-minimisation-strict /etc/unbound/），再检查上游权威服务器是否真的修好了。回滚：整条链回滚到 strict: no 的配置快照，并保留一次逐层刷缓存的记录，便于复盘。

13.4 fail-closed：禁止静默明文降级

隐私机制再完整，只要有一条「失败时偷偷走明文」的路径，前面全部作废。这一节把第 5 章的结论收拢成可断言的工程要求。

13.4.1 两波都失败就 SERVFAIL

「两波」指加密上游的两轮尝试（含重试）都失败：TLS 握手失败、超时、或对端 SERVFAIL。此时正确行为是给客户端 **SERVFAIL**，让监控与人看见故障；错误行为是回退到明文 53 自行递归，查询流全程裸奔，而且客户端完全无感。

```
# /etc/unbound/unbound.conf.d/20-fail-closed.conf
server:
    # 禁止 SERVFAIL 时逃出本区自行递归 / 走明文
    forward-first: no
```

```
    tcp-upstream: yes
    edns-buffer-size: 1232
    max-udp-size: 1232

forward-zone:
    name: "."
    forward-addr: 192.0.2.53@853#dns.example.org
    forward-tls-upstream: yes
    forward-first: no
```

为什么这么设：`forward-first: no` 在 `forward-zone` 里再写一次，是为了让这个约束贴着上游定义，避免日后在 `server:` 段被别的文件覆盖；`forward-tls-upstream: yes` 与 `# 认证名`保证加密且认证（第 11 章）；`tcp-upstream: yes` 让出向只用 TCP 承载，配合第 14 章跨境链路的实测结论。

怎么验证：

```
# 断链演练：临时阻断到上游 853，观察客户端应答与抓包
nft add rule inet filter output tcp dport 853 drop
dig @127.0.0.1 -p 5353 example.org A | grep -E 'status:|SERVFAIL'
tcpdump -ni any 'tcp port 853 or udp port 53 or tcp port 53' -c 20
nft flush chain inet filter output   # 演练结束立刻恢复
```

期望 `status: SERVFAIL`，且抓包里**没有**向任何地址的 53 端口明文查询。

失败模式 / 回滚点：演练时 `nft` 规则一定要记得恢复（用 `nft list ruleset > /etc/nftables.conf.bak` 先备份）。若出现「慢一点但还是返回答案」，几乎一定是 `forward-first: yes` 或存在第二条 `forward-addr`。

13.4.2　单上游断言

转发器里给 `forward-zone` 配第二个 `forward-addr`，Unbound 是**并发竞速取最快**，不是主备。这会把查询送到既定上游之外，令过滤表、加密策略、日志归属全部失效。单上游不是「冗余不足」，而是「策略唯一」的断言：上游切换交给外部巡检脚本（第 5 章）。

```
unbound-control list_forwards
grep -rn 'forward-addr' /etc/unbound/unbound.conf.d/ | grep -v '^.*#'
```

为什么这么设：用可执行的断言代替「配置里看一眼」。`list_forwards` 显示运行态实际生效的转发目标，`grep` 检查所有 include 片段里有没有藏着的第二条。

怎么验证：`list_forwards` 中每个 zone 只有一条地址；`grep` 输出里 `forward-addr` 行数与 zone 数一致。

失败模式 / 回滚点：配置文件被某个包管理器的默认片段追加了一条上游，是最隐蔽的来源。回滚：把上游定义集中到单一文件，并在巡检脚本里加一条「条数不等于 1 就告警」。

13.4.3　fallback 定义了但无人引用，也算安全

一个常被忽略的细节：配置里可以存在一段 `forward-zone` 或某个备用池，只要**没有任何规则把查询引到它**，它就不会被使用。第 12 章 dnsdist 的 `newServer({pool="..."})` 定义了池但不写 `PoolAction` 引用，同样不会生效。这类「死配置」在审计时是安全的，但有两个前提：

```
# 确认没有规则引用某个备用池 / 备用转发区
dnsdist -c 127.0.0.1:5199 -k '<TOKEN>' -e 'showRules()' 2>/dev/null || true
unbound-control list_forwards
unbound-control list_local_zones | grep -E 'fallback|backup' || true
```

为什么这么设：把「定义了」与「被引用」分开验证。审计结论要写成「该 fallback 无引用路径」，而不是「没有 fallback」——后者会被下次配置变更推翻。

怎么验证：`showRules()` / `list_forwards` 里看不到指向备用目标的规则；同时用断链演练（13.4.1）确认故障时确实没有流量改道。

失败模式 / 回滚点：某天有人加了一条 `PoolAction` 或 `forward-first: yes`，死配置瞬间变成**静默降级通道**。回滚：把这类 fallback 定义放进单独的注释文件并定期复查引用关系；发现被引用就按 13.4.1 重做断链演练。

13.5 DDR 压制与浏览器 canary

客户端绕过本地解析器的路径有两条：DDR（Discovery of Designated Resolvers，[RFC 9462]）自动发现加密解析器，以及浏览器自带的 DoH。两条都要在本地解析器上压制，而不是靠封 IP。

DDR 不是 IPv6 独有的：它由 [RFC 9462] 定义，触发条件是「客户端只知道一个**解析器 IP 地址**」——不论这个地址是 IPv4 还是 IPv6。RFC §4 明确写了：若指定解析器通过 IPv4 可达，就应提供一条 A 记录指向指定解析器的 IPv4 地址；若通过 IPv6 可达，就提供 AAAA 记录；并提醒「只连得上某一个地址族的客户端，可能用不上另一族发现的端点」。也就是说 DHCP（[RFC 2132]）下发的 IPv4 DNS 与 IPv6 RA/RDNSS（[RFC 8415]）下发的 IPv6 DNS 都能触发 DDR，只是发现出的加密端点要按地址族给对记录。本书第 19 章讲 RDNSS 时提到它，是因为那里正好在配 IPv6 解析器地址，**不是**因为 DDR 只能跑在 v6 上。

```
# /etc/unbound/unbound.conf.d/30-ddr-canary.conf
server:
    local-zone: "_dns.resolver.arpa." always_nodata
    local-zone: "use-application-dns.net." always_nxdomain
```

为什么这么设：`_dns.resolver.arpa.` 是 DDR 的特区，客户端通过查询它的 SVCB / HTTPS 记录来发现 DoT / DoH / DoQ 端点。**SVCB**（Service Binding，[RFC 9460]）是「服务绑定」记录：在发起连接前就告诉客户端「这个服务有哪些替代端点、各自带什么参数」（端口、协议、IP hint、TLS 配置等），HTTPS 记录是它专用于 HTTP 的变体；DDR 借它把加密解析器端点一次性交出去（第 2 章有 SVCB 的完整概念）。`always_nodata` 返回 NODATA（名字存在但没有所问类型的记录），且**忽略该区下的 local-data**，等于告诉客户端「这里没有可发现的加密解析器」，客户端继续用本地解析器。`always_nxdomain` 对 Firefox 的 canary 域生效：回 NXDOMAIN 让 Firefox 关闭内置 DoH（第 12 章）。两者语义不同，不能互换——`always_nxdomain` 会让「名字不存在」，`always_nodata` 让「名字存在但无此类型」。

怎么验证：

```
dig @127.0.0.1 -p 5353 _dns.resolver.arpa. SVCB +short
dig @127.0.0.1 -p 5353 _dns.resolver.arpa. SVCB | grep 'status:'
dig @127.0.0.1 -p 5353 use-application-dns.net A | grep 'status:'
unbound-control list_local_zones | grep -E '_dns.resolver|use-application'
```

期望 SVCB 查询 `status: NOERROR`、ANSWER SECTION 为空（NODATA）；canary 域 `status: NXDOMAIN`。

canary 域是什么意思：canary 原意是「矿井里的金丝雀」——先于矿工察觉毒气。在软件里指**一个专门用来探测某项功能是否被启用的探测点**。Firefox 的 `use-application-dns.net` 就是这样的探测域：浏览器在启用内置 DoH 之前先查它，若返回 NXDOMAIN（或不是预期应答），就判定「当前网络不希望我自动开 DoH」并退回系统解析器；返回正常地址则允许升级。它给企业/家庭网络留了一个**不用封 IP、不用改浏览器配置**就能声明策略的开关。本书用 `always_nxdomain` 让本地解析器对该域回 NXDOMAIN，正是在模拟这个声明；Chrome 不走这个机制（靠已知 DoH 提供商列表匹配，见下节），所以 canary 只对 Firefox 系生效。[1]

失败模式 / 回滚点：如果链路上还有另一层解析器（nft 劫持目标、dnsdist 后端、系统存根），那一层没配这两条→客户端照样发现端点或启用内置 DoH。**要逐层验证**。另一个坑是 `always_nodata` 与 `static` 的区别：前者忽略 `local-data`，需要给个别名字留白名单时应改用 `transparent` 或配合具体 `local-data`。回滚：注释掉两条 `local-zone` 并 restart，客户端立即恢复默认发现行为。

13.5.1 Chrome 与「按 IP 封 443 是 denylist 不是证明」

Chrome 的内置 DoH 升级依赖**系统解析器是否匹配已知 DoH 提供商列表**。指向私有解析器（如 `192.0.2.1`、`fd00::1`、`192.0.2.1`）时不在列表里，Chrome 不会自行升级，会继续用系统解析器——因此不需要 canary 域，也不需要封域名。

按 IP 封 443 的问题是三重的：会误伤同一 IP 上的正常 Web 服务；端点换地址就失效；最重要的是它**不能证明客户端没有绕过**，只能证明「这几个地址不通」。真正的验收是「能不能测出绕过」：把一台客户端的系统 DNS 临时改成公共解析器，看浏览器是否升级到内置 DoH、本地 canary 是否仍把它压回来，并用抓包确认有没有直连外部 443 的 DoH 流量。

```
# 验收要测「绕过」，不是测「封没封住」
cat /etc/resolv.conf
dig @127.0.0.1 -p 5353 whoami.example.org A +short
tcpdump -ni any 'tcp port 443 and not dst net 192.0.2.0/24' -c 20
```

为什么这么设：把判据从「denylist 有没有命中」换成「系统解析器是谁、有没有外部 DoH 连接」。前者是配置，后者是行为。

怎么验证：`resolv.conf` 指向自建解析器；抓包里没有客户端直连外部 DoH 端点的连接；改系统 DNS 做一次对照演练后恢复。

失败模式 / 回滚点：演练后忘记把系统 DNS 改回来，会让这台客户端长期绕过所有本地策略。回滚：演练前后各存一份 `/etc/resolv.conf` 快照，并在脚本里用 `trap` 恢复。

13.6　ECS 默认不发

EDNS Client Subnet（ECS，[RFC 7871]）会把客户端地址的一部分随查询送给权威服务器，让 CDN 按「用户所在网段」返回就近节点。代价是：**每发一次 ECS，就等于主动向第三方披露一条客户端网段信息**，与本章的抗元数据目标直接冲突。

Unbound 只有在 `module-config` 里显式加入 `subnetcache` 模块，并且配置了 `send-client-subnet` / `client-subnet-zone` 时才会向权威服务器添加 ECS 选项。默认模块链是 `"validator iterator"`，**默认不发 ECS**。官方手册也建议只在「解析器与客户端属于不同网络」的开放解析器场景启用，因为它会按网段切分缓存、拖累命中率。

```
# /etc/unbound/unbound.conf.d/40-ecs.conf ——默认（不启用 ECS）
server:
    module-config: "validator iterator"
    # 不写 subnetcache，不写 send-client-subnet / client-subnet-zone
```

为什么这么设：把「不发」写成显式的一行 `module-config`，是为了让审计能直接读到结论，而不是靠「默认应该没开吧」。若确实需要 ECS（例如解析器与客户端不同网、且 CDN 调度必须靠它），应把 `max-client-subnet-ipv4` / `max-client-subnet-ipv6` 设成能接受的最粗前缀，并且只把 `send-client-subnet` 列给确需的目标，而不是全局打开 `client-subnet-always-forward: yes`。

怎么验证：

```
unbound-control get_option module-config
tcpdump -ni any 'udp port 53' -c 30 -vvv 2>/dev/null | grep -i 'CLIENT-SUBNET' || true
dig @127.0.0.1 -p 5353 +subnet=192.0.2.0/24 example.org A | grep -E 'CLIENT-SUBNET|status:'
```

第一条应显示模块链里没有 `subnetcache`；抓包里不应出现 CLIENT-SUBNET 选项。注意即便客户端自己带了 ECS 选项（`dig +subnet=`），未启用 `subnetcache` 时 Unbound 也不会把它转发给未列入 `send-client-subnet` 的权威服务器。

失败模式 / 回滚点：把 `module-config` 写成 `"subnetcache validator iterator"` 却忘了配任何 `send-client-subnet`，表面上「开了 ECS 模块」但什么都没发，容易误判成已生效或已泄露。回滚：恢复 `"validator iterator"`，并复查所有 include 片段里有没有 `client-subnet-zone`。

13.7　dnstap 是自己的查询流副本

dnstap 是结构化的 DNS 事件日志格式，能记录客户端查询、客户端应答、解析器与转发器的出向/入向报文。它的价值是排障与审计；它的风险是**你自己在自己门口生成了一份完整的查询流副本**。前面所有 QMIN、DDR 压制、ECS 默认不发，都是为了减少元数据外泄；如果 dnstap 全量日志无限期留存，等于把削减掉的元数据又完整存了一份在本地磁盘上，还会进入备份、日志采集系统、SIEM。

```
# /etc/unbound/unbound.conf.d/50-dnstap.conf
# 教学示例：开启时写明保留期（例如 30 天），由外部采集器落盘
dnstap:
    dnstap-enable: yes
    dnstap-socket-path: "/run/dnstap/dnstap.sock"
    dnstap-log-client-query-messages: yes
    dnstap-log-client-response-messages: yes
    dnstap-log-resolver-response-messages: yes
    dnstap-log-forwarder-response-messages: yes
    dnstap-bidirectional: no
```

为什么这么设：明确列出记录了哪几类消息，便于审计「到底留了什么」；`dnstap-socket-path` 由外部采集器接收，Unbound 自己不直接写盘，因此**保留期由采集器与 logrotate 落实**，不能只写在注释里。`dnstap-bidirectional: no` 避免重复记录同一条流的两个方向。

怎么验证：

```
unbound-checkconf /etc/unbound/unbound.conf.d/50-dnstap.conf
ls -l /run/dnstap/dnstap.sock
# 保留期是否真的执行（示例：30 天）
find /var/log/dnstap -type f -mtime +30 -print
systemctl cat dnstap-collector 2>/dev/null | grep -i -E 'retention|rotate' || true
```

`find` 在稳定运行 30 天以上的系统上应该为空；若列出文件，说明保留期只是写在文档里。

失败模式 / 回滚点：三个高频问题：①编译 Unbound 时没带 `--enable-dnstap`，配置能过但功能不存在；②采集器挂了导致 dnstap 线程阻塞或事件丢失，表现为「日志有缺口」而不是「DNS 不通」；③只给 nginx access log 设了保留期，dnstap 归档无限增长。回滚：`dnstap-enable: no` 并 restart，先恢复可观测性的最小集，再修采集器。

13.7.1 把保留期写成可执行断言

```
# 教学示例：采集器侧的清理任务，保留 30 天
find /var/log/dnstap -type f -name '*.dnstap' -mtime +30 -delete
# 审计：确认配置文件里写明了保留期，且任务确实被调度
grep -rn '30' /etc/logrotate.d/dnstap /etc/systemd/system/dnstap-*.service 2>/dev/null || true
systemctl list-timers | grep -i dnstap || true
```

为什么这么设：把「保留期」从一句承诺变成一条可 grep、可看定时器、可看文件年龄的事实。第 12 章的 nginx access log 与这里的 dnstap 要用同一套口径，否则会出现「前端日志 30 天、查询流副本无限期」的落差。

怎么验证：`systemctl list-timers` 里能看到清理任务；`find -mtime +30` 无输出。

失败模式 / 回滚点：`-delete` 类命令写错路径会误删别的日志。回滚：先不带 `-delete` 跑一遍 `-print` 看清单，确认路径后再启用删除；并保留采集器配置的备份。

13.8 一张验收清单

检查项	期望	命令 / 依据
QMIN	`yes` 且 `strict: no`	`unbound-control get_option ...`
兼容性	静态资源域名有 ANSWER	`dig assets.cdn.example.org A`
负缓存	strict 改动后逐层刷过	每台实例 `unbound-control flush_zone`
单上游	每 zone 一条 `forward-addr`	`unbound-control list_forwards`
fail-closed	断 853 后 SERVFAIL、抓包无 53	断链演练 + `tcpdump`
fallback	定义了但无引用路径	`showRules()` / `list_forwards`
DDR	`_dns.resolver.arpa.` NODATA	`dig ... SVCB`
canary	`use-application-dns.net` NXDOMAIN	`dig ... A`
绕过可测	改系统 DNS 能观察到行为变化	对照演练
ECS	模块链无 `subnetcache`，抓包无 CLIENT-SUBNET	`get_option` + `tcpdump`
dnstap	记录类型明确、保留期执行	`find -mtime +30` + 定时器

13.9 本章要点

- 查询名最小化（QMIN）**抗的是元数据聚合，不抗全路径嗅探**，也不是加密；本地根区镜像时根这一跳本来不出口，收益主要落在 TLD 层。`harden-below-nxdomain` 只对 DNSSEC-secure 的 NXDOMAIN 生效，无验证器时是空转。
- `qname-minimisation: yes` 是尽力而为：中间层返回非 NOERROR 就回退完整 QNAME；`strict: yes` **不回退**，官方手册明确警告「很多域将无法解析」。
- 某大型 CDN 的 CNAME 区违反 [RFC 8020]（父名 NXDOMAIN、子名却存在）→ strict 模式问中间层拿到 NXDOMAIN 就收工，整站 CSS/JS 解析失败；改 `strict: no` 后恢复，**且链路上每层缓存都要刷**，否则前一层照样返回缓存的 NXDOMAIN（负缓存按 SOA minimum 缓存，常见 3600 秒）。

- fail-closed 三件套：`forward-first: no` 禁止静默明文降级、两波都失败就 SERVFAIL、`forward-addr` 只留一条（多条是并发竞速取最快而非主备）。**`fallback` 定义了但无人引用也算安全**，但要用 `showRules()` / `list_forwards` 与断链演练证明无引用路径。
- `local-zone: "_dns.resolver.arpa." always_nodata` 压制 DDR，`"use-application-dns.net." always_nxdomain` 关闭 Firefox 内置 DoH；Chrome 靠系统解析器匹配已知提供商列表，指向私有解析器时不升级。**按 IP 封 443 是 denylist 不是证明**，验收要能测出绕过。
- ECS 默认不发（`module-config` 不含 `subnetcache`）；dnstap 是**自己的查询流副本**，全量日志必须写明记录类型与保留期（例如 30 天）并验证真的执行，否则抗元数据聚合在自己门口破功。

13.10　参考资料

- RFC 9156, *DNS Query Name Minimisation to Improve Privacy*: https://www.rfc-editor.org/rfc/rfc9156
- RFC 7816, *DNS Query Name Minimisation to Improve Privacy*（前一版）: https://www.rfc-editor.org/rfc/rfc7816
- RFC 8020, *NXDOMAIN: There Really Is Nothing Underneath*: https://www.rfc-editor.org/rfc/rfc8020
- Mozilla Support, *Canary domain* `use-application-dns.net`（Firefox 内置 DoH 探测域的官方说明）[1]: https://support.mozilla.org/en-US/kb/canary-domain-use-application-dnsnet
- RFC Editor, [RFC 9460] *Service Binding and Parameter Specification via the DNS (SVCB and HTTPS Resource Records)*: https://www.rfc-editor.org/rfc/rfc9460
- RFC Editor, [RFC 2132] *DHCP Options and BOOTP Vendor Extensions*（IPv4 侧的 option 6 DNS）: https://www.rfc-editor.org/rfc/rfc2132
- RFC Editor,[RFC 8415] *Dynamic Host Configuration Protocol for IPv6 (DHCPv6)*:https://www.rfc-editor.org/rfc/rfc8415
- RFC Editor, [RFC 9462] *Discovery of Designated Resolvers (DDR)*: https://www.rfc-editor.org/rfc/rfc9462
- RFC 7871, *Client Subnet in DNS Queries*: https://www.rfc-editor.org/rfc/rfc7871
- Unbound 官方手册, `unbound.conf(5)` 的 `qname-minimisation`、`qname-minimisation-strict`、`harden-below-nxdomain`、`local-zone` 类型、EDNS Client Subnet 模块与 dnstap 段: https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html
- Unbound 官方手册, `unbound-control(8)` 的 `list_forwards` / `flush_zone` / `get_option`: https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound-control.html
- Mozilla Support, Canary domain `use-application-dns.net`: https://support.mozilla.org/en-US/kb/canary-domain-use-application-dnsnet
- dnstap 官方站点（结构化 DNS 事件日志格式）: https://dnstap.info/
- dnsdist 官方手册, Actions / Selectors / Constants 参考（`PoolAction`、`RCodeAction`、`DNSRCode.NXDOMAIN`）: https://www.dnsdist.org/reference/actions.html

第十四章

加密通道选路与 fail-closed 工程

前几章把 DoT、DoH、QMIN、fail-closed 分别讲成了可独立的配置片段。这一章把它们放回一**条完整链路**里：客户端的 :53 被 nftables 劫持进本地解析器，解析器按国内/国外分流，转发器把查询送进加密隧道，隧道另一端的递归验证器继续向权威服务器迭代。任何一段的规则写错，表现都像「上游挂了」——所以本章的排查顺序是固定的：**先看 nftables 规则，再怀疑别的**。

14.1 整体链路：从客户端:53 到远端递归验证器

```
[LAN 客户端 / WG 对等端]
   | ① 明文 :53（写死 DNS 的物联网设备、不理 RA 的存根解析器）
   ▼
[gw-01 策略层]  nft: hijack-dns 把 LAN/WG 的 :53 tproxy 到本地 :5353
   |              ★ 本机解析器进程（unbound / nsd）出向 53 按 uid 豁免
   ▼
[本地策略解析器 A]  过滤 / canary / DDR 压制 / 国内-国外分流
   | ② 按域名选路：国内域 → 本地明文上游；其余 → 转发器 B
   ▼
[转发器 B]  单上游、forward-first: no、fail-closed
   | ③ 加密隧道入口（本机回环 :5354 → SSH SOCKS / DTLS / DoT）
   ▼
[加密隧道]  TCP 承载优先（SSH SOCKS / DoT 853），UDP/QUIC 只在链路允许时
   | ④ 跨境链路
   ▼
[cloud-01 递归验证器]  DNSSEC 验证 + QMIN + EDNS 1232
   | ⑤ 明文 53 迭代查询（根 / TLD / 权威）
   ▼
[权威服务器]
```

这张图里有两个容易搞混的「加密」：③→④是**传输加密/隐私**，⑤的 DNSSEC 是**数据完整性/来源认证**。第 11、12 章强调过的边界在这里依然成立：整条链路里只有「客户端↔本地解析器」和「网关↔云主机」这两段能被我们加密，云主机向权威服务器发出的迭代查询通常仍是明文 53。想让根那一跳不出本机，走第 6 章的本地根区镜像。

```
# /etc/unbound/policy.conf ——本地策略解析器 A（节选，教学用示例地址）
server:
    interface: 127.0.0.1@5353
    access-control: 127.0.0.0/8 allow
    access-control: 192.0.2.0/24 allow
    access-control: fd00::/8 allow
    access-control: 0.0.0.0/0 refuse

    edns-buffer-size: 1232
    max-udp-size: 1232
    qname-minimisation: yes
    qname-minimisation-strict: no
    harden-glue: yes
```

```
    local-zone: "_dns.resolver.arpa." always_nodata
    local-zone: "use-application-dns.net." always_nxdomain

# 国内域走本地上游（教学示例：198.51.100.53）
forward-zone:
    name: "cn.example."
    forward-addr: 198.51.100.53
    forward-first: no

# 其余走转发器 B（本机回环），B 再进加密隧道
forward-zone:
    name: "."
    forward-addr: 127.0.0.1@5354
    forward-first: no
    do-not-query-localhost: no
```

为什么这么设：A 只做策略与过滤，把「是否出隧道」的决定集中在一处；`forward-first: no` 保证 SERVFAIL 时不会逃出该区自行递归（第 13 章的 fail-closed）。`do-not-query-localhost: no` 只在「本机另一实例确实是设计中的下一跳」时才放开，且必须保证端口、pidfile、控制接口不重叠、拓扑不成环（第 5、7 章）。canary 与 DDR 两条压在 A 上，是因为**客户端的 DNS 首先到 A**；压在 B 上就晚了。

怎么验证：

```
unbound-checkconf /etc/unbound/policy.conf
dig @127.0.0.1 -p 5353 www.cn.example. A +short
dig @127.0.0.1 -p 5353 example.org A +short
unbound-control -c /etc/unbound/policy.conf list_forwards
```

失败模式 / 回滚点：`unbound-checkconf` 对 `forward-addr` 指向回环且 `do-not-query-localhost: yes` 时会给出警告「地址不会被用于查询」——不会报错，但转发会静默失败，表现为全量 SERVFAIL。这是最容易被忽略的一条。回滚：明确写 `do-not-query-localhost: no`，并确认 A 与 B 的端口不重叠。

14.2　nft 规则：劫持:53 与 uid 豁免

14.2.1　完整规则示例

```
#!/usr/sbin/nft -f
# /etc/nftables.conf ——教学用示例地址段，勿直接用于生产
table inet mangle {
    set domestic-dns {
        type ipv4_addr
        flags interval
        elements = { 198.51.100.53 }
    }

    chain hijack-dns {
        type nat hook prerouting priority dstnat; policy accept;
        # LAN 与 WG 网段里写死 :53 的流量，tproxy 到本地策略解析器
        iifname { "lan0", "wg0" } udp dport 53 \
            meta mark set 0x2333 tproxy ip to 127.0.0.1:5353 counter accept
        iifname { "lan0", "wg0" } tcp dport 53 \
            meta mark set 0x2333 tproxy ip to 127.0.0.1:5353 counter accept
    }

    chain output-tproxy {
        type route hook output priority mangle; policy accept;

        # ★ 链首：WG 自己的 udp sport 51820 出包必须显式放行（见 14.5）
        udp sport 51820 counter accept

        # ★ 本机解析器进程出向 53 按 uid 豁免，否则自己吃掉自己的递归查询
        meta skuid unbound udp dport 53 counter accept
```

```
        meta skuid unbound tcp dport 53 counter accept
        meta skuid nsd udp dport 53 counter accept

        # 需要走代理的流量：打标后交给策略路由（教学示例：目的为文档地址段）
        ip daddr 203.0.113.0/24 tcp dport { 22, 853, 443 } \
            meta mark set 0x2332 counter accept
    }
}

table inet filter {
    chain output {
        type filter hook output priority filter; policy accept;
        # fail-closed 演练用：临时 drop 853（14.4 节），平时保持注释
        # tcp dport 853 counter drop
    }
}
```

为什么这么设：hijack-dns 用 tproxy 把 LAN/WG 的 53 交给本地 5353，客户端无需改任何配置；output-tproxy 处理**本机自身发出的包**，两条 meta skuid 豁免在第 5、7 章已作为结论给出——不豁免的话，Unbound 自己向权威服务器发的递归查询会被自己的 hijack 吃掉，表现为根区镜像永远不更新、AXFR 失败。udp sport 51820 的链首 accept 是 14.5 节的坑，必须放在打标规则**之前**（链尾规则会连它一起打标）。

怎么验证：

```
nft -c -f /etc/nftables.conf          # 只检查语法，不加载
nft -f /etc/nftables.conf
nft list ruleset | sed -n '1,120p'
nft list chain inet mangle output-tproxy
nft reset counters
# 触发流量后再看计数器
dig @127.0.0.1 -p 5353 example.org A +short
nft list chain inet mangle output-tproxy
```

失败模式 / 回滚点：nft -f 是**重建表**，会清空集合（见 14.3）；meta skuid unbound 要求系统里确实存在该用户，写成别的名字等于没写。回滚：改动前 nft list ruleset > /etc/nftables.conf.bak，出问题时 nft -f /etc/nftables.conf.bak。

14.2.2 排查顺序：先看 nftables

「根区镜像不更新」「AXFR 失败」「WG 握手不通」「上游看起来全挂」这四个现象里，至少三个的根因在规则表，不在 Unbound 配置。固定排查顺序：

```
# ① 规则与集合是不是还在
nft list ruleset | sed -n '1,120p'
nft list chain inet mangle hijack-dns
nft list chain inet mangle output-tproxy

# ② uid 豁免有没有命中（本机解析器出向 53 不应被 tproxy）
nft list ruleset | grep -i skuid
nft reset counters && dig @127.0.0.1 -p 5353 example.org A +short
nft list ruleset | grep -i -E 'counter|packets' | head -20

# ③ 抓包看包究竟去了哪里（mark 是否被打上）
tcpdump -ni any 'udp port 53 or tcp port 853 or udp port 51820' -c 30
```

为什么这么设：把「配置写了」与「规则命中了」分开。计数器是唯一能证明包实际走过哪条规则的证据；抓包用来确认打标后是否真的进了隧道。

怎么验证：LAN 客户端查询使 hijack-dns 计数增加；本机解析器出向 53 的计数**不**增加；udp sport 51820 在 output-tproxy 链首有 accept 且计数增加。

失败模式 / 回滚点：nft reset counters 会清零所有计数，排查前先记录基线。回滚：优先恢复备份规则集，而不是逐条删规则——逐条删容易留下半截状态。

14.3　`nft -f` 重建表会清空集合

一个很实际但很少写进文档的坑：`nft -f file` 加载规则文件时，如果文件里定义了 `table inet mangle { ... }` 且包含 `set`，它是按文件内容**重建**这张表，而不是增量合并。这意味着：

- 通过 `nft add element inet mangle domestic-dns { 203.0.113.7 }` 在运行时往集合里加的地址，加载文件后**全部消失**；
- 由脚本（DDNS 更新、策略域名列表）动态维护的集合，加载后回到文件里写的那几个初始值；
- 计数器、quota 等有状态元素同样被重置。

```
# 危险做法：直接 nft -f 覆盖，动态集合会被清空
nft -f /etc/nftables.conf

# 安全做法：加载前导出含运行时集合的规则集，加载后补齐
nft list ruleset > /var/tmp/nft-before-lab.txt
nft -f /etc/nftables.conf
nft add element inet mangle domestic-dns { 203.0.113.7 }
nft list table inet mangle | grep -A5 'set domestic-dns'
```

为什么这么设：把「补齐」写成加载流程的一部分，而不是事后想起来。脚本化时应把动态集合的初始内容写进一个单独的 `.nft` 片段，并在主文件加载后统一 `nft -f` 追加，避免依赖人工记忆。

怎么验证：加载前后各跑一次 `nft list table inet mangle`，比对 `elements = { ... }`；再触发一条针对动态成员地址的查询，确认命中对应规则。

失败模式 / 回滚点：现象通常是「重启后分流突然只对几个默认域生效」「策略路由部分失效」，很容易被误判成上游或 DDNS 脚本坏了。回滚：`nft -f /var/tmp/nft-before-lab.txt`（导出的规则集包含当时的集合内容），并把动态集合的维护脚本纳入开机流程。

14.4　跨境链路实测：为什么加密通道优先 TCP

这一节给出本项目在「一台家庭网关 + 一台境外云主机的实验环境」里的实测结论。同一条跨境链路上，用一个 1 MB 的测试文件对比两种承载：

承载	传输层	实测吞吐	现象
UDP / QUIC 下载	UDP	~300 B/s	1 MB 要跑很久，丢包与限速叠加，几乎不可用
SSH SOCKS（TCP 承载）	TCP	~660 KB/s	同一链路同一时段，稳定在数百 KB/s

差距约三个数量级。机理不是「TCP 比 UDP 快」，而是这条链路对 UDP 做了明显的 QoS / 限速：UDP 没有拥塞控制反馈，被策略性丢弃后只能靠上层重试，速率塌到几百字节每秒；TCP 有拥塞控制与重传，能收敛到一个可用速率。**DNS 隧道同理**：把 DNS 查询装进 UDP/QUIC 上跑，丢包后要等超时重试，客户端表现为「解析时好时坏、慢得离谱」；装进 TCP 承载的通道（SSH SOCKS、DTLS 之外的 TLS 流、DoT 853 本身就是 TCP），反而稳。

```
# 教学示例：用 SSH SOCKS 把转发器的出向流量塞进 TCP 隧道
ssh -N -D 127.0.0.1:1080 -p 22 \
    -o ServerAliveInterval=30 -o ServerAliveCountMax=3 \
    -o ExitOnForwardFailure=yes \
    user@203.0.113.10

# 转发器侧：只用 TCP 出向，避免掉进 UDP 被 QoS 的路径
# /etc/unbound/exit.conf
#   server:
#     tcp-upstream: yes
#     forward-zone:
#       name: "."
#       forward-addr: 192.0.2.53@853#dns.example.org
#       forward-tls-upstream: yes
#       forward-first: no
```

为什么这么设：tcp-upstream: yes 让 Unbound 向权威/上游侧只用 TCP（官方手册称其「useful in tunneling scenarios」）;DoT 853 天然是 TCP/TLS,两者叠加后整条出向路径没有 UDP 承载的 DNS 报文。ServerAliveInterval / ExitOnForwardFailure 保证隧道挂掉时快速失败而不是静默卡住,配合 fail-closed 才能在两波失败后明确 SERVFAIL。

怎么验证：

```
# 吞吐对照：同一 1MB 文件，两种承载各测一次（教学示例命令形态）
dd if=/dev/urandom of=/tmp/test1m.bin bs=1M count=1
curl -sS -o /dev/null -w 'socks=%{speed_download} B/s\n' \
  --socks5-hostname 127.0.0.1:1080 --max-time 60 \
  https://dns.example.org/test1m.bin
# 断链演练：隧道断开后 DNS 必须 SERVFAIL，不能静默走明文
pkill -f 'ssh -N -D 127.0.0.1:1080' || true
dig @127.0.0.1 -p 5353 example.org A | grep 'status:'
tcpdump -ni any 'udp port 53' -c 10
```

期望 status: SERVFAIL，抓包里没有明文 53 出向；curl 的 speed_download 用来对照 TCP 通道的可用速率。

失败模式 / 回滚点：tcp-upstream: yes 是全局开关，会同时影响本地国内域的解析（如果 A 也用它）；更稳的做法是把「出隧道的那一段」限定在 B 的配置里（第 5 章的 forward-tcp-upstream 可按 zone 设置）。另一个坑是 SSH SOCKS 只代理 TCP，DNS 若还有 UDP 段（例如 A 的国内上游）照旧走明文，别把「开了 SOCKS」当成「全链路加密」。回滚：恢复 tcp-upstream: no 的配置快照，并重新拉起隧道。

14.4.1 fail-closed 在整条链路上怎么落地

第 13 章给的是单点的 fail-closed，这里补链路版：

```
① 客户端 :53 → A              nft 劫持；A 自身出向 53 按 uid 豁免
② A → 国内上游 / B             两条 forward-zone 都 forward-first: no
③ B → 加密隧道                 单上游、forward-tls-upstream: yes、# 认证名
④ 隧道断开                     两波都失败 → SERVFAIL，绝不回退明文 53
⑤ 兜底配置                     fallback 定义了但无人引用也算安全（第 13 章）
```

```
# /etc/unbound/exit.conf ——转发器 B：唯一出隧道的出口
server:
    interface: 127.0.0.1@5354
    access-control: 127.0.0.0/8 allow
    do-not-query-localhost: yes
    tcp-upstream: yes
    tls-cert-bundle: "/etc/ssl/certs/ca-certificates.crt"
    edns-buffer-size: 1232
    qname-minimisation: yes
    qname-minimisation-strict: no

forward-zone:
    name: "."
    forward-addr: 192.0.2.53@853#dns.example.org
    forward-tls-upstream: yes
    forward-first: no
```

为什么这么设：B 只有一个上游、只有一个出口，切换交给外部巡检脚本（第 5 章）；forward-first: no 把「隧道断了」变成可见的 SERVFAIL。edns-buffer-size: 1232 是第 11 章 IPv6 最小 MTU 1280 的推导结果，套上 WireGuard 后 MTU 更小，1472 会分片被丢。

怎么验证：

```
unbound-checkconf /etc/unbound/exit.conf
unbound-control -c /etc/unbound/exit.conf list_forwards
# 断链演练：drop 853，观察 SERVFAIL 与抓包
nft add rule inet filter output tcp dport 853 counter drop
dig @127.0.0.1 -p 5353 example.org A | grep 'status:'
tcpdump -ni any 'tcp port 853 or udp port 53 or tcp port 53' -c 20
nft flush chain inet filter output
```

失败模式 / 回滚点：演练时 nft 规则务必恢复（先 nft list ruleset > /etc/nftables.conf.bak）。若出现「慢一点但仍返回答案」，检查 A 的 forward-first 是否被别的文件覆盖、以及有没有第二条 forward-addr。

14.5　真实形态的坑：WG 回包被 output-tproxy 吃掉

这是策略网关里最反直觉的一个故障。现象是：**WireGuard 隧道起不来**，握手一直不完成；但 wg show 里能看到端点地址、配置看起来完全正确，云主机侧也没收到握手包。

14.5.1　因果链

① 策略网关的 output-tproxy 链尾有一条兜底打标规则：meta mark set 0x2332
② 这条规则本意是「本机发出的、需要走代理的流量打标后交给策略路由」
③ 但 WG 自己的 udp sport 51820 出包也会经过 output 链
④ 于是 WG 发出的握手 / 响应包被打上 0x2332 → 被送进代理
⑤ 代理路径不会把它送回云主机的 WG 端点 → 对端收不到握手响应
⑥ 隧道起不来；此时任何「DNS 走隧道」的设计都无从谈起

第一反应通常是加一条 meta skuid 0 accept 放行「内核发出的包」，因为直觉上内核没有用户态 uid。**这个匹配不到**：meta skuid 只匹配用户态进程的 socket owner，WG 在内核态发出的这类包没有对应的 skuid 元数据，链尾的 mark 规则照样命中。

14.5.2　修法：链首按 sport 显式 accept

```
#!/usr/sbin/nft -f
# /etc/nftables.d/10-wg-exempt.nft ——教学示例
table inet mangle {
    chain output-tproxy {
        # 必须放在所有 mark / tproxy 规则之前
        udp sport 51820 counter accept
        # 若隧道还承载其它端口，一并在这里列出
    }
}
```

为什么这么设：udp sport 51820 是 WG 的监听端口，出包的源端口就是它，这个字段在包里一定存在、与发包主体是内核还是进程无关。accept 会终止本链后续规则，包不再被打标、不再进 tproxy。关键在**链首**：nftables 链内规则按添加顺序匹配，链尾的兜底 mark 会命中任何走到那里的包。

怎么验证：

```
nft list chain inet mangle output-tproxy | sed -n '1,40p'
# 确认 sport accept 在 mark 之前
nft list chain inet mangle output-tproxy | grep -n -E '51820|mark set'
# 触发握手并观察计数器与对端
wg show wg0 latest-handshakes
nft reset counters && wg show wg0 peers | head
nft list chain inet mangle output-tproxy | grep -A1 '51820'
tcpdump -ni any 'udp port 51820' -c 20
```

期望：grep -n 里 51820 的行号**小于** mark set 的行号；latest-handshakes 出现近期时间戳；抓包能看到双向的 51820 流量。

失败模式 / 回滚点：两种常见复发方式——①有人把「优化后的规则文件」重新 nft -f 加载，链首 accept 被冲掉（14.3 节）；②换了 WG 端口却没改规则。回滚：nft -f /etc/nftables.conf.bak，并把这条 accept 写进主规则文件而不是临时命令。另一个误判来源是「以为是云主机防火墙没放行 51820」，结果在云主机侧查了半天——按 14.2.2 的顺序，先看网关自己的 nftables。

14.5.3　与 uid 豁免的对照

豁免对象	匹配方式	原因
本机解析器（unbound / nsd）出向 53	`meta skuid unbound` / `meta skuid nsd`	用户态进程，有 socket owner
WG 内核发出的 udp sport 51820	`udp sport 51820`	内核发包，`meta skuid 0` 匹配不到

为什么这么设：把两条并列写清楚，避免把「按 uid 豁免」当成通用方案。前者管的是「别让解析器吃掉自己的递归查询」(14.2.1)，后者管的是「别让隧道自己的握手包被送进代理」，两者都是链首/链首附近的 accept，但匹配字段完全不同。

怎么验证：`nft list ruleset | grep -i -E 'skuid|51820'` 能同时看到两类规则；计数器上，解析器出向 53 不进 tproxy、WG 出包不打标。

失败模式 / 回滚点：如果为了图省事写了 `udp dport 53 accept` 而不带 `meta skuid`，等于把 LAN 客户端的 53 也放行了，劫持策略失效。回滚：恢复带 `meta skuid` 的精确豁免。

14.6 一张验收清单

检查项	期望	命令
规则语法	`nft -c -f` 无错误	`nft -c -f /etc/nftables.conf`
劫持生效	LAN :53 进本地 5353	`nft list chain inet mangle hijack-dns` + 计数器
uid 豁免	解析器出向 53 不进 tproxy	`nft list ruleset | grep skuid` + 计数器
WG 豁免	`sport 51820` 在链首、mark 之前	`nft list chain inet mangle output-tproxy | grep -n`
集合完整	`-f` 加载后动态成员补齐	`nft list table inet mangle` 前后比对
单上游	A / B 每个 zone 一条	`unbound-control -c ... list_forwards`
fail-closed	断 853 / 隧道后 SERVFAIL	断链演练 + `tcpdump`
无明文出向	抓包无 `dst port 53`	`tcpdump -ni any 'udp port 53 or tcp port 53'`
优先 TCP	出向走 853 / SOCKS，不走 UDP/QUIC	吞吐对照 + `curl --socks5-hostname`
MTU	出向 EDNS 1232	`dig +bufsize=4096 ... | grep udp:`
回滚可用	规则集备份存在且可加载	`nft -c -f /etc/nftables.conf.bak`

14.7 本章要点

- 完整链路是「客户端:53 → nft 劫持→本地策略解析器 A（分流 / canary / DDR 压制）→转发器 B →加密隧道（SSH SOCKS / DTLS / DoT）→远端递归验证器→权威服务器」；**加密传输只覆盖其中几段**，远端向权威发出的迭代查询通常仍是明文 53，DNSSEC 提供的是数据完整性/来源认证，两者不互相替代。
- nft `hijack-dns` 用 tproxy 把 LAN/WG 的 53 送进本地解析器；**本机解析器进程（unbound、nsd）的出向 53 必须按 `meta skuid` 豁免**，否则自己的递归查询被自己的劫持吃掉，表现为根区镜像不更新、AXFR 失败。
- **`nft -f` 是重建表，会清空集合**：运行时用 `nft add element` 加的地址、计数器、动态策略列表加载后全部消失，必须在加载流程里补齐，并把导出 `nft list ruleset` 作为回滚基线。排查网络故障时**先看 nftables 规则，再怀疑别的**。
- 跨境实测：同一链路 UDP/QUIC 下载 1 MB 只到 ~300 B/s，而 TCP 承载的 SSH SOCKS 达 ~660 KB/s，所以对 UDP 做 QoS 的网络里**加密通道优先 TCP**（`tcp-upstream: yes`、DoT 853、SSH SOCKS）。fail-closed 要求隧道断开两波都失败就 SERVFAIL，禁止静默明文降级；单上游是策略唯一的断言，fallback 定义了但无人引用也算安全。
- 策略网关 `output-tproxy` 链尾的 `meta mark set 0x2332` 会连 WireGuard 自己的 `udp sport 51820` 出包一起打标送进代理，导致握手响应到不了对端、隧道起不来；`meta skuid 0` 匹配不到内核发出的这类包，**必须在链首按 `udp sport 51820` 显式 accept**，且 accept 要排在 mark 规则之前。

14.8　参考资料

- nftables 官方 wiki,Setting packet metainformation(`meta mark` / `meta skuid`): https://wiki.nftables.org/wiki-nftables/index.php/Setting_packet_metainformation
- nftables 官方 wiki，规则集管理与 `nft -f` 加载语义：https://wiki.nftables.org/wiki-nftables/index.php/Main_Page
- Unbound 官方手册，`unbound.conf(5)` 的 `tcp-upstream`、`forward-tcp-upstream`、`do-not-query-localhost`、`forward-first` 与 `edns-buffer-size`：https:// unbound.docs.nlnetlabs.nl/ en/ latest/ manpages/ unbound.conf.html
- Unbound 官方手册,`unbound-control(8)` 的 `list_forwards` 与统计命令:https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound-control.html
- NLnet Labs，Unbound 项目文档入口：https://nlnetlabs.nl/documentation/unbound/
- RFC 7858, *Specification for DNS over Transport Layer Security (TLS)*: https://www.rfc-editor.org/rfc/rfc7858
- RFC 7766, *DNS Transport over TCP - Implementation Requirements*: https://www.rfc-editor.org/rfc/rfc7766
- RFC 5737,*IPv4 Address Blocks Reserved for Documentation*:https://www.rfc-editor.org/rfc/rfc5737
- WireGuard 官方手册（端点端口与握手）：https://www.wireguard.com/quickstart/

第四篇　RFC 与协议理论



第十五章

DNS 基础 RFC 族

[RFC 1034] 与 [RFC 1035] 发布于 1987 年 11 月，前者叫 *Domain Names - Concepts and Facilities*，后者叫 *Domain Names - Implementation and Specification*。两者加起来不到两百页，却规定了此后三十多年几乎所有 DNS 行为的骨架：命名空间是树、解析是自上而下的委派、报文是固定首部加四个段、资源记录（resource record，RR）是类型化的键值。

这一章不复述报文格式表，也不给任何配置块。它要回答的是另一类问题：为什么后来的十几份 RFC 都要回来”澄清”1987 年的原始文本；这些澄清各自堵住了哪个实现漏洞；以及当我们在第三篇里打开某个开关时，究竟是踩在哪一条条文逻辑上。全章对每份 RFC 采用统一的四段式：**它要解决什么→关键条文→对实现的约束→常见误读**。

第四篇统一按 RFC 编号引用，不写完整 URL；可核验的公开链接集中在本章末尾《参考资料》。

15.1 命名空间与报文：[RFC 1034] / [RFC 1035]

15.1.1 它要解决什么

1980 年代初的主机名到地址映射靠一份集中维护的 `HOSTS.TXT`。规模一大，分发延迟、名字冲突、单点编辑权都成了问题。[RFC 1034] 的答案是把名字空间做成**倒置的树**，并把管理权按子树切分：根不存主机，只存指向顶级域的委派；每一级只对自己这一层负责。[RFC 1035] 把这套概念落成线格式与算法，定义了报文首部、四类段（Question / Answer / Authority / Additional）、以及资源记录的通用编码。

两者是一体的：[RFC 1034] 明确写着自己”省略了许多细节”，细节要到配套的 [RFC 1035] 里找；反过来 [RFC 1035] 假定读者已经读过 [RFC 1034] 的概念部分。把它们拆成”概念 RFC”与”实现 RFC”是理解后续所有澄清 RFC 的前提——**后来大量的争议不是概念错了，而是线格式层面的歧义没写死**。

15.1.2 关键条文

- 域名是从叶向根读的标签序列，书写时用点分隔；根用零长度标签表示，绝对域名以根标签结尾。树上的每个节点可以有一组 RR，节点的存在与否和它有没有某个类型是两件事。
- 委派（delegation）通过 NS RRset 与胶水记录（glue record）实现：父区在 Authority 段给出子区的 NS，并在 Additional 段附上这些 NS 名字对应的地址，让递归解析器不必先解析 NS 名字就能继续往下走。
- 报文首部里的 RD（Recursion Desired）、RA（Recursion Available）、RCODE、以及四个计数，规定了存根解析器与递归解析器之间的分工。CNAME 与 DNAME 之类的重定向由应答内容表达，不是独立 RCODE。
- [RFC 1034] 定义了 TTL 与缓存的必要性：任何应答都可以被缓存，缓存时长由 RRset 的 TTL 决定；[RFC 1035] 规定了压缩指针（compression pointer），允许报文里重复出现的域名用两字节偏移引用。

15.1.3 对实现的约束

递归解析器必须实现**迭代**（iteration）：从根 hints 出发，一次一个区切（zone cut）往下问，把每次得到的委派当作新的权威服务器集合。权威服务器在应答里塞进不属于自己权威范围的附加数据，接收方不能直接信任——这条约束在 [RFC 2181] 里被再次强调，也是现代清理器（scrubber）只把”可信”数据写入缓存的条文源头。

压缩指针意味着解析器不能简单地按字节比较域名：同一个名字在报文里可能有多种编码形式。大小写不敏感与压缩叠加，直接催生了 [RFC 4343] 的澄清。

15.1.4　常见误读

误读一：RD=1 表示”请递归”，所以权威服务器必须递归。权威服务器对 RD 置位的查询仍可只回答自己权威范围内的数据，并把 RA 清零；RD 是请求方的意图，不是命令。把”收到带 RD 的查询”当成”必须代为解析”是开放解析器（open resolver）常见的成因之一。

误读二：Additional 段里的地址是可信数据。不是。只有落在当前委派所必需的胶水范围内的地址才有理由被使用；域外夹带的 glue 是经典缓存投毒面。这一点与 DNSSEC 是否开启无关，任何模式都该清理。

误读三：NXDOMAIN 只说”这个名字不存在”。原始文本表述含糊，直到 [RFC 8020] 才把”其下所有名字也不存在”钉死。这个歧义是第三篇里严格查询名最小化（strict QNAME minimisation）踩坑的条文根源。

15.2　操作与规范的澄清：[RFC 1912] / [RFC 2181]

15.2.1　它要解决什么

[RFC 1912]（1996，*Common DNS Operational and Configuration Errors*）是 Informational，它不定义协议，而是把当时实际部署里反复出现的配置错误列成清单：SOA 参数不合理、CNAME 与其他类型共存、NS 指向 CNAME、PTR 缺失、胶水不匹配、把私有地址写进公共区等。它的动机很朴素：协议没错，但人写错了，而且错法高度重复。

[RFC 2181]（1997，*Clarifications to the DNS Specification*）性质不同，它是把 [RFC 1034]/[RFC 1035] 里”似乎规定了但各家实现理解不一致”的地方重新写明确的规范文本。

15.2.2　关键条文

[RFC 2181] 里对后续实现影响最大的几条：

- **RRset 是缓存与签名的单位**。同一 owner name、同一 class、同一 type 的所有记录构成一个 RRset，必须整体处理：TTL 必须一致，不能只缓存其中一条，也不能把从不同应答里拼起来的记录混成一个 RRset。后来 DNSSEC 的 RRSIG 签的是 RRset 而不是单条记录，根子就在这里。
- **TTL 的语义被收紧为”最多可缓存的秒数”**，并明确 16 位无符号解释、以及高位被置时的处理，避免把大 TTL 当负数。
- **RCODE 与区段内容的组合**被逐类说明：什么是 referral、什么是权威否定、什么时候 Authority 段的 NS 属于引用而非答案。
- **class 不是只有 IN**，类型码与 class 码是独立的命名空间；实现不该假定 QCLASS 恒为 IN。
- **名字的长度上限、标签的字节语义**被明确：DNS 名字比较是逐标签的、大小写不敏感的二进制比较，不是本地化的字符串比较。

[RFC 1912] 则给出了大量”应当避免”的操作惯例，其中与今天仍强相关的包括：NS 与胶水记录必须一致、SOA 的序列号与刷新参数要能真正驱动传送、不要把 CNAME 当作可与其他数据共存的别名。

15.2.3　对实现的约束

解析器在缓存层必须以 RRset 为粒度做插入、淘汰与 TTL 递减；以”单条记录”为粒度的缓存会在 TTL 不一致时产生半新半旧的应答。清理器还需检查应答里 owner name 是否落在查询所期待的范围内，避免把无关联的 RRset 顺手写进缓存。

权威侧与递归侧共用一套线格式解析，但信任边界不同：权威服务器对本地区数据天然可信，递归解析器对收到的任何东西都要重新判定归属。

15.2.4　常见误读

误读一：[RFC 1912] 是标准，必须逐条照做。它是 Informational，内容是运维经验清单，不是协议一致性判据。但其中”NS 不能指向 CNAME”“CNAME 不能与其他数据共存”后来在别处被规范化，实际约束力比它的类别看起来强。

误读二：“澄清”意味着 [RFC 1034] 写错了。更准确的说法是：原文留了多种自洽的读法，[RFC 2181] 选定一种并要求实现收敛。把澄清 RFC 当”补丁列表”读，会漏掉它最有价值的部分——RRset 作为原子单位这一条，后来成了整个 DNSSEC 数据模型的地基。

15.3 负缓存：[RFC 2308]

15.3.1 它要解决什么

[RFC 1034] 提到了可以缓存否定应答，但没说清楚缓存多久、按什么判据判定”这是否定应答”、以及 NXDOMAIN 与”名字存在但类型不存在”要不要区别对待。结果是否定查询反复打向权威服务器，既浪费带宽也放大故障。[RFC 2308]（1998，*Negative Caching of DNS Queries (DNS NCACHE)*）把负缓存（negative caching）从”可选优化”提升为解析器应有的常规行为，并明确说” negative caching should no longer be seen as an optional part of a resolver”。

15.3.2 关键条文

- 区分两种否定：**NXDOMAIN**（RCODE = Name Error，名字不存在）与 **NODATA**（RCODE = NOERROR、Answer 段无相关记录、Authority 段有 SOA 或无 NS）。NODATA 是伪 RCODE，必须从应答内容算法式地判定，没有独立编码。
- 负缓存的 TTL 取自 Authority 段 SOA 记录的 MINIMUM 字段与 SOA 自身 TTL 中**较小者**；[RFC 2308] 还重定义了 SOA MINIMUM 的语义为”负缓存 TTL”，不再被当作区刷新间隔使用。
- 负应答的 RCODE 不能被解析器”美化”成 NOERROR 空应答；缓存的否定要能在后续同域查询时直接复用，包括对同名不同 QTYPE 的查询。
- [RFC 2308] 也允许对 SERVFAIL 一类失败做短时缓存（上限很小，常见实现取数分钟），但明确这属于实现选择。

15.3.3 对实现的约束

解析器必须为否定结果建立独立的缓存条目，并且该条目要有明确的过期时间；不能靠”没有正向记录”来隐式表达否定。缓存键要同时覆盖 QNAME、QTYPE 的否定语义与 class，否则 NODATA 会被误用于其他类型。

运维上最要紧的推论：**上游一次瞬时 NXDOMAIN 会被按 SOA minimum 缓存**——现实中这个字段常见取 3600 秒。权威侧修好之后，客户端仍会在整段 TTL 内继续失败。而链路上的每一层缓存（本地 Unbound、前置缓存、系统存根）各自持有一份否定条目，只刷其中一层不会让故障消失。第三篇与第五篇讨论的”逐层刷缓存”就是从这条条文里直接推出来的。

15.3.4 常见误读

误读一：负缓存只缓存 NXDOMAIN。 NODATA 同样要缓存，判定逻辑反而更复杂。只实现前者的解析器会在”域名存在但没有 AAAA”这类高频查询上反复外发。

误读二：把 SOA MINIMUM 当刷新计时器。 [RFC 2308] 之后它在负缓存语境里的含义是 TTL 上限；区传送节奏由 SOA 的 REFRESH / RETRY / EXPIRE 决定。

误读三：负缓存是”错误”，关掉更安全。关掉等于把否定查询全部变成实时递归，权威侧抖动会被原样放大到客户端；问题不在机制，而在 TTL 取值与故障时的可观测性。

15.4 EDNS0: [RFC 2671] → [RFC 6891]

15.4.1 它要解决什么

1987 年的报文首部把不少字段用满了：RCODE 只有 4 位、标志位没有余量、UDP 应答默认被限制在 512 字节。DNSSEC 一出现，签名与密钥让应答轻易超过 512 字节，必须走 TCP，代价是连接开销。[RFC 2671]（1999，*Extension Mechanisms for DNS (EDNS0)*）用”伪资源记录”的方式扩展报文而不改首部；[RFC 6891]（2013，*Extension Mechanisms for DNS (EDNS(0))*, STD 75）**废弃** [RFC 2671] 与 [RFC 2673]，成为现行规范。

15.4.2 关键条文

- OPT 伪 RR 放在 Additional 段，owner name 为根，type 为 OPT；它的 CLASS 字段被借作**请求方 UDP 载荷上限**，TTL 字段被借作扩展标志（其中最高位是 DO 位，供 DNSSEC 使用），RDATA 承载若干 `option-code / option-length / option-value` 三元组。
- 扩展 RCODE 由 OPT 里的 8 位 EXTENDED-RCODE 与原首部 4 位组合，形成 12 位 RCODE 空间；BADVERS 用于版本不匹配。

- 不支持 EDNS 的一方会忽略或拒绝 OPT；**发送方必须能在超时后回退**：先降载荷、再退回无 EDNS 的 512 字节 UDP、必要时走 TCP。[RFC 6891] 把这种回退与”不要对同一服务器反复重试已知不支持的扩展”写成实现要求。
- OPT 不是真实 RR，不进缓存、不参与签名、不出现在区文件里。它只在这一个报文的这一跳上有效。

15.4.3　对实现的约束

解析器出向通告的 UDP 载荷与入向可接受的最大值是**两个方向、两个参数**。只调入向参数不会缩小自己发出的查询，路径 MTU 出问题时分片照样发生。这也是本书把出向 EDNS 载荷压到 1232 字节、而不是沿用早期 1472 字节的条文依据：IPv6 规定的最小链路 MTU 是 1280 字节（见 [RFC 8200]），减去 IPv6 首部与 UDP 首部后，1232 是能在不分片前提下装下的 DNS 载荷；PPPoE 1492、各类隧道 MTU 也都比 1472 更紧。

此外，EDNS 是逐跳的：转发器与上游之间、上游与权威之间可以各自协商不同的载荷。任何一跳丢失分片，表现都是”等超时→重试 TCP”，而不是清晰的报错。

15.4.4　常见误读

误读一：EDNS0 等于 DNSSEC。EDNS0 只是承载机制；DO 位表示”请把 DNSSEC 相关记录给我”，但数据完整性与来源认证由 DNSSEC 自己完成。反过来，DoT / DoH / DoQ 提供的是**传输加密与隐私**，与 EDNS0 也毫无替代关系。

误读二：把载荷调大能提升性能。在存在分片丢弃的链路上，调大只会把一次 UDP 往返变成一次超时加一次 TCP 连接。

误读三：OPT 里的选项会被中间解析器透传。不会。每个解析器按自己与下一跳的协商重新构造 OPT；ECS（[RFC 7871]）这类选项的隐私代价正来自”它会被有意识地重新生成并沿链路传递”，见第 17 章。

15.5　未知 RR 类型：[RFC 3597]

15.5.1　它要解决什么

[RFC 1035] 只定义了当时存在的类型码。新类型（后来有 SRV、NAPTR、DS、RRSIG、NSEC、SVCB、HTTPS 等）出现时，如果每个中间节点都要升级软件才能”看懂”新记录，协议就无法演进。[RFC 3597]（2003，*Handling of Unknown DNS Resource Record (RR) Types*）给出的是**透明转发**规则：不认识类型时，按通用二进制格式搬运，不要猜测、不要丢弃、不要把 RDATA 当名字解析。

15.5.2　关键条文

- 定义 `\# <length> <hex>` 的通用 RDATA 表示法，供区文件与展示工具使用。
- 明确**名字是否可压缩取决于类型**，未知类型的 RDATA 里不得做压缩指针解析，也不得对 RDATA 里的字节做大小写折叠。未知类型的记录按不区分大小写处理其 owner name，但 RDATA 保持原样。
- 服务器对未知 QTYPE 应正常应答（有则给记录，无则 NODATA），不得因为”不认识”就返回 FORMERR 或 REFUSED。
- 递归解析器缓存未知类型时按 RRset 整体缓存，与已知类型一致。

15.5.3　对实现的约束

这条 RFC 是 SVCB / HTTPS 记录能被逐步部署的前提，也因此是第 17 章 DDR 发现机制能跑在现有解析器上的前提。它要求缓存与转发路径对类型码保持”数据不可知”：只有当实现确实知道某类型的语义（例如 DS 用于构建信任链）时才做特殊处理。

它也解释了 DNSSEC 的一个设计选择：RRSIG、NSEC、DS 等在 [RFC 4034] 中被明确定义之前，中间设备若按 [RFC 3597] 透明搬运，就不会成为部署障碍。

15.5.4　常见误读

误读一：不认识的类型可以丢掉以”保持安全”。丢弃会制造人为的 NODATA，客户端会据此得出错误结论（例如认为某域不支持某种服务），而且这种错误会被负缓存放大。

误读二：所有 RDATA 里的域名都能压缩。只有类型定义允许时才可以；[RFC 3597] 明确禁止对未知类型做压缩解析，避免把任意二进制误当成指针。

15.6 大小写：[RFC 4343]

15.6.1 它要解决什么

[RFC 1034] 说域名比较”case-insensitive”，但没说清边界：是 ASCII 字母吗？非 ASCII 字节呢？大小写不敏感是否适用于 RDATA 里的名字？压缩指针里的名字呢？0x20 编码这类把大小写当随机熵使用的技巧是否合法？[RFC 4343]（2006, *Domain Name System (DNS) Case Insensitivity Clarification*, 更新 [RFC 1034]/[RFC 1035]/[RFC 2181]）逐条收口。

15.6.2 关键条文

- DNS 名字的大小写不敏感**仅指 ASCII 的 A–Z 与 a–z**，且只作用于 owner name 与 RDATA 中”域名类型”的字段；其他字节不折叠，也不做 Unicode 大小写或本地化规则。
- 比较与缓存键应当按规范化形式进行，但**应答里应尽量保留查询方原本的大小写写法**（所谓 0x20 兼容）：解析器把查询名里的大小写当作额外熵，权威服务器回显它，接收方据此判别应答是不是针对自己这次查询的伪造包。
- 大小写不敏感不改变线格式：压缩指针、标签长度、字节序列都照旧。
- 明确”case-insensitive”不等于”case-preserving 是必需的”，但把大小写原样回显是实现可以合法利用的信道。

15.6.3 对实现的约束

缓存键必须规范化，否则 `Example.ORG` 与 `example.org` 会变成两份缓存条目，负缓存与 DNSSEC 验证结果也会分裂。同时，出向查询名可以携带随机化的大小写，只要接收与比较时按不敏感规则处理；这要求实现把”比较用的规范形式”与”线上实际发的字节”分开保存。

对 DNSSEC 也有一层影响：规范名（canonical name）用于签名计算时统一按小写处理，因此大小写差异不应导致签名验证失败；但把 RDATA 里非名字字段也一起折叠，就会破坏签名。

15.6.4 常见误读

误读一：大小写不敏感 = 全部转小写后连 RDATA 一起改。会破坏二进制 RDATA，尤其对未知类型（[RFC 3597]）。

误读二：0x20 是加密或认证。它只是提高伪造应答的猜测成本，属于熵增强；[RFC 4343] 讲清它合法，不等于讲清它安全。真正给数据来源认证的是 DNSSEC，给传输加密与隐私的是 DoT / DoH / DoQ。

15.7 特殊用途域名：[RFC 6761]

15.7.1 它要解决什么

`.local`、`.test`、`localhost`、`onion`、`invalid`、以及后来的 `resolver.arpa`、`home.arpa` 这类名字，散落在各份 RFC 里被”保留”，但保留到底意味着什么并不统一：是权威服务器上永远不该出现？是递归解析器必须本地回答？还是仅仅”建议别用”？[RFC 6761]（2013, *Special-Use Domain Names*）定义了”特殊用途域名”这一概念、注册程序，并建立了 IANA 注册表，把若干已有名字作为初始条目写入。

15.7.2 关键条文

[RFC 6761] 为每个特殊用途域名按七类问题给出明确答案，其中最要紧的三类是：

- **用户与应用**是否应把它当普通域名对待；
- **存根解析器 / 递归解析器**是否应本地生成应答、是否允许向上游查询；
- **权威服务器**是否应对该名字下的查询给出 NXDOMAIN 或 NODATA。

例如 `invalid` 被设计为”保证解析失败”，用于测试；`test` 保留给测试与文档；`localhost` 要求本地解析到环回地址、不许出网。注册表是活的表，后续 RFC 可以往里加条目（第 17 章的 `resolver.arpa`、第 18 章涉及的 `home.arpa` 都是通过各自 RFC 注册或引用的）。

15.7.3 对实现的约束

递归解析器需要一份本地策略来处理这类名字：**在本地给出规定好的应答，不向外发送查询**。这既避免把内部查询泄露到公网，也避免让上游的意外应答污染本地语义。第三篇里对 `_dns.resolver.arpa.` 做 `always_nodata`、以及用 canary 域让

浏览器退回系统解析器的做法，条文依据都落在这套”本地服务区（locally served zone）“思路上，而不是”按 IP 封端口”。

需要注意 [RFC 6761] 建立的是**框架**：某个具体名字该返回什么，要看注册表中该条目所引用的具体 RFC。把”特殊用途”笼统理解成”一律 NXDOMAIN”是错的。

15.7.4　常见误读

误读一：保留名只是”没人注册所以解析不了”。不是。它要求实现主动做本地处理；依赖”公网恰好没有这个区”是不可靠的。

误读二：特殊用途域名可以随意拿来当内网域名。恰恰相反，注册过的名字有规定语义；自建内网应使用自己拥有的域名或使用明确规定可用于此目的的名字，并把文档示例域名（`example.net` 等）与实际部署区分开。

15.8　NXDOMAIN cut：[RFC 8020]

15.8.1　它要解决什么

原始规范对 NXDOMAIN 的范围表述含糊：如果 `foo.example` 返回 NXDOMAIN，那么 `bar.foo.example` 是否必然不存在？按树形命名空间的逻辑应该是，但 [RFC 1034] 没写死，[RFC 2308] 的负缓存规则也没要求据此剪枝。[RFC 8020]（2016，*NXDOMAIN: There Really Is Nothing Underneath*，更新 [RFC 1034] 与 [RFC 2308]）把这句话变成规范：**NXDOMAIN 意味着被否定的名字以及它下面所有名字都不存在**。

15.8.2　关键条文

- 递归解析器收到 NXDOMAIN 后，**可以且应当**据此判定该名字的整个子树都不存在，从而停止对该子树的后续查询。
- 这条更新同时修正了 [RFC 2308] 中负缓存适用范围的表述：负缓存条目可以覆盖整个被否定的子树，而不只是被查询的那一个名字。
- 权威服务器若让某个中间名返回 NXDOMAIN、却在更深的子名上挂数据，属于违反本协议的行为；[RFC 8020] 的附录专门解释了为什么”用返回应答的 owner name 来判断”是不可靠的 workaround。

15.8.3　对实现的约束

对查询名最小化（QNAME minimisation）影响最直接。最小化解析器逐标签往上问，中间某一层若返回 NXDOMAIN，按 [RFC 8020] 就可以直接结束，无需再问完整域名。**严格模式（strict）**正是把这条推理用到极致：拿到中间层 NXDOMAIN 就断定整条链不存在，不做回退。

现实里存在违反 [RFC 8020] 的权威配置——典型是某些大型 CDN 的 CNAME 区，父名返回 NXDOMAIN、子名却真实存在。此时严格模式会把整站的资源域名判成不存在，页面样式与脚本全部解析失败。而且这个错误结论会被链路上的每一层负缓存按 [RFC 2308] 的 TTL 记住，所以改回宽松模式后**还需要逐层刷新缓存**。第三篇第 13 章记录了这个兼容坑的参数处理；本章只讲条文：出问题的是权威侧违反 [RFC 8020]，不是解析器”多疑”。

另有一条容易混淆的边界：`harden-below-nxdomain` 一类剪枝选项只对 **DNSSEC-secure 的 NXDOMAIN** 生效，无签名区的 NXDOMAIN 不触发它。这是实现把安全强度分级的结果，不是 [RFC 8020] 本身的要求。

15.8.4　常见误读

误读一：NXDOMAIN cut 是解析器可选的优化，不开也没损失。它对最小化查询量与负缓存范围有实际影响；但开启严格模式的收益，建立在”所有权威都守规矩”这个前提上。

误读二：把 [RFC 8020] 当成 DNSSEC 的一部分。它是纯 DNS 解析语义的澄清，与签名无关；[RFC 8499] 也把它归入术语与语义收敛的一条线。

误读三：改了模式就立刻恢复。负缓存是分布式的，每一层实例都得刷。

15.9 术语统一：[RFC 8499]

15.9.1 它要解决什么

DNS 由几十份 RFC 共同定义，同一个词在不同年代的文档里含义漂移：“resolver”有时指递归侧、有时指存根；“zone”与“domain”被混用；“negative answer”、“NODATA”、“NXDOMAIN”、“referral”、“delegation”的边界也常被写混。[RFC 8499]（2019，*DNS Terminology*，废弃 [RFC 7719]、更新 [RFC 2308]）把当前用法集中成一份术语表，供后续 RFC 引用。

15.9.2 关键条文

[RFC 8499] 逐条给出当前定义，与本书用词直接相关的有：

- **存根解析器（stub resolver）**、**递归解析器（recursive resolver）**、**转发器（forwarder）**、**权威服务器（authoritative server）**、**缓存解析器（caching resolver）**是不同角色，一台进程可以同时扮演多个，但概念上不能互换。
- **区（zone）**是管理边界，**域（domain）**是命名空间中的子树；**委派（delegation）**是父区把子区的权威指出去，不等同于区切本身。
- **肯定应答 / 否定应答 / 引用（referral）**的判据被重新表述，并顺带更新了 [RFC 2308] 对 NODATA 的措辞。
- 明确“QNAME minimisation”、“aggressive use of DNSSEC-validated cache”等较新术语的所指。

15.9.3 对实现的约束

术语表不产生线格式变化，但它决定了配置文档与日志的可读性：当手册说“validator”、“iterator”、“scrubber”时，指的是验证器、迭代器、清理器三个可分离组件；当说“AD bit”时指的是验证结果标志而非请求标志。本书采用同一套中文译名（见术语统一表），并把 **DNSSEC = 数据完整性/来源认证**与 **DoT/DoH/DoQ = 传输加密/隐私**作为不可互换的两条线，正是遵循 [RFC 8499] 的收敛方向。

15.9.4 常见误读

误读一：术语 RFC 没有技术内容，可以忽略。恰恰相反，很多”配置看起来对但结论错”的排查，根源是把 `forwarder` 当成 `recursive resolver`，或者把 `NODATA` 当成 `NXDOMAIN`——这两种否定在负缓存、在查询名最小化、在 DNSSEC 否认存在里的处理都不同。

误读二：把“caching resolver”当产品名。它是角色。一个进程可以既是递归解析器又是缓存解析器，也可以只做转发而不自行迭代。

15.10 过期缓存应答：[RFC 8767]

15.10.1 它要解决什么

权威服务器不可达（DDoS、链路故障、配置事故）时，传统解析器在 TTL 到期后只能返回 SERVFAIL。但缓存里那份数据往往仍然正确——变化的只是”没来得及重新确认”。[RFC 8767]（2020，*Serving Stale Data to Improve DNS Resiliency*）把业界早已存在的做法写成规范：在刷新失败时，允许把过期缓存应答（serve-stale）返回给客户端，用短暂的数据陈旧换取可用性。

15.10.2 关键条文

- 定义三个计时器：**查询解析计时器**（一次解析努力的总时长上限）、**最大过期计时器**（数据最多能被当作过期数据使用多久）、以及已有的常规解析计时器。
- 建议流程是**先尝试刷新，失败或超时才回过期缓存**，而不是一开始就回旧数据；[RFC 8767] 也讨论了”先回旧数据再后台刷新”的变体。
- 返回过期数据时应让客户端能感知：TTL 被置得很小，且应答不应被误标为权威。
- 明确过期数据不是”权威数据”，不得用它去覆盖 DNSSEC 验证失败，也不得把它当作新的信任依据。

15.10.3 对实现的约束

实现必须把”缓存条目的 TTL 到期”与”条目被删除”分开：前者之后条目仍要留在缓存里一段时间，才能被 serve-stale 复用。同时要先启动刷新、在计时器到期后才回退，否则会把一次瞬时抖动变成全站旧数据。

这里最容易被误读的是计时器单位：`serve-expired-client-timeout` 的取值单位是**毫秒**，默认 1800 即 1.8 秒，语义是”先尝试刷新，超过 1.8 秒还没结果才回过期缓存”，不是 1800 秒。把它当秒来调，等于把回退窗口放大三个数量级。

还要注意与 [RFC 2308] 的关系：serve-stale 复用的是**过期的肯定数据**，负缓存里的否定结果不应被同样复用——把一次瞬时 NXDOMAIN 当”过期但可继续用”的旧答案，会把负缓存放大故障的问题再叠一层。

15.10.4　常见误读

误读一：serve-stale 等于”忽略 TTL”。它只在刷新努力失败时启用，且有最大过期时长上限。

误读二：过期数据一定是错的。大多数 DNS 记录的 TTL 远短于实际变更频率；但”通常正确”不等于”可以认证”，DNSSEC 验证状态必须重新判定。

误读三：可用性机制能替代安全机制。[RFC 8767] 解决的是权威不可达时的可用性，与 DNSSEC 的数据完整性、与加密传输的隐私，三者各补一块。

15.11　文档地址段：[RFC 5737]

15.11.1　它要解决什么

技术文档、示例配置、教材里的 IP 地址如果随手写成可路由的公网地址，读者复制粘贴后可能真的把流量送到无辜的第三方，也会让”这个地址属于谁”变成误导。[RFC 5737]（2010, *IPv4 Address Blocks Reserved for Documentation*, 更新 [RFC 1166]）指定三段专门用于文档的地址，并在 IANA 保留。

15.11.2　关键条文

- `192.0.2.0/24`（TEST-NET-1）、`198.51.100.0/24`（TEST-NET-2）、`203.0.113.0/24`（TEST-NET-3）专门留给文档与示例。
- 这些地址**不应出现在公共互联网路由表**中，也不应被当作可到达的目的地；文档作者可以放心使用，不需要向任何人申请。
- 对应的 IPv6 文档前缀是 `2001:db8::/32`（[RFC 3849]），私有网络地址用 RFC 1918 段与 [RFC 4193] 的 ULA 通用写法。

15.11.3　对实现的约束

本书所有示例统一使用上述段，并明确标注”以下为教学用示例地址”。域名侧使用 `gw.example.net`、`dns.example.org`、`resolver.example.net` 这类保留示例名；ULA 用 `fd00:…`的通用写法；API token 一律写作 `<TOKEN>` 或 `example-token-0000`。

这条约束不只是脱敏，也是可复现性要求：读者在自己的实验环境里看到 `192.0.2.x` 时，应当立刻知道”这是要被替换的占位符”，而不会误以为某台真实主机在应答。

15.11.4　常见误读

误读一：文档地址段就是私有地址。不是。RFC 1918 的地址（如 `192.168.x.x`）可以在真实内网使用；[RFC 5737] 的三段只应出现在文档里，不该被配置到接口上。

误读二：只要换掉 IP 就不算泄露。主机名、域名、token、拓扑全貌同样属于需要泛化的信息；本书把它们统一处理为 `gw-01`、`edge-01`、`cloud-01` 这类中性教学标识，并明确写为教学环境。

15.12　条文之间的因果链

把上面十一组 RFC 串起来，能看到一条清晰的演进逻辑，而不只是编号列表：

1. [RFC 1034]/[RFC 1035] 定义了树、委派、报文与缓存，但把许多边界留给实现自己判断。
2. [RFC 1912] 收拾运维错误，[RFC 2181] 把 RRset、TTL、RCODE 组合这些歧义收口；RRset 作为原子单位后来成了 DNSSEC 签名的数据模型基础（第 16 章）。
3. [RFC 2308] 补上负缓存，[RFC 8020] 补上 NXDOMAIN 的作用范围；两者叠加，既让否定查询不再反复外发，也埋下了”瞬时否定被长时间记住”“严格最小化遇到不守规矩的权威”两类坑。

4. [RFC 2671] → [RFC 6891] 用 OPT 伪 RR 打开扩展空间，才有 DNSSEC 的大应答、ECS 的选项、SVCB 的承载；[RFC 3597] 保证新类型能穿过不认识它的中间节点。
5. [RFC 4343] 把大小写不敏感的边界划清，使 0x20 熵增强与缓存规范化能共存；[RFC 6761] 把”这个名字不要出网”变成显式本地策略；[RFC 8499] 最后把三十年里漂移的词汇统一。
6. [RFC 8767] 在可用性维度上给缓存加了一条”过期但可回退”的路径，与 [RFC 2308] 的否定缓存形成互补而非重复。
7. [RFC 5737] 与 IPv6 文档前缀是写作层的基础设施，保证前六步讨论的所有机制都能用不会伤及第三方的示例地址讲清楚。

这条链也解释了第四篇与其余各篇的分工：第二、三、五篇处理”怎么做”，本卷只处理”为什么这么规定”。例如 EDNS 1232 的取值在第三篇是配置，在这里是 [RFC 6891] 的逐跳载荷协商加上 [RFC 8200] 的 1280 字节最小 MTU；严格最小化的兼容坑在第三篇是参数，在这里是 [RFC 8020] 与 [RFC 2308] 的组合推论。

15.13 本章要点

- [RFC 1034]/[RFC 1035] 分工是”概念”与”实现”；后续大量 RFC 不是推翻它们，而是把线格式与语义的歧义收口。[RFC 2181] 确立的 **RRset 原子性**后来成为 DNSSEC 签名的数据模型地基。
- [RFC 2308] 让负缓存成为常规行为：NXDOMAIN 与 NODATA 判据不同，TTL 取 SOA MINIMUM 与 SOA TTL 的较小者；瞬时否定会被长时间记住，且链路上每层缓存都要单独刷新。
- [RFC 2671] 被 [RFC 6891] 废弃；OPT 伪 RR 借 CLASS 通告 UDP 载荷、借 TTL 位承载 DO/扩展 RCODE。**出向载荷与入向最大值是两个方向**，IPv6 最小 MTU 1280 是 1232 字节取值的根因。
- [RFC 8020] 把 NXDOMAIN 的范围钉死为”该名字及其下所有名字均不存在”，这是严格查询名最小化的条文依据；权威侧违反它会造成整站资源解析失败，且错误结论被 [RFC 2308] 的负缓存逐层记住。
- [RFC 3597] 的未知类型透明转发、[RFC 4343] 的 ASCII 大小写边界、[RFC 6761] 的特殊用途域名本地服务区、[RFC 8499] 的术语收敛，分别支撑了 SVCB/HTTPS 逐步部署、0x20 熵增强、DDR 压制与全书用词一致。
- [RFC 8767] 的过期缓存应答是**可用性**机制：先刷新、超时才回旧数据，计时器单位为毫秒；与 DNSSEC 的数据完整性/来源认证、DoT/DoH/DoQ 的传输加密/隐私互不替代。
- 本书示例统一采用 [RFC 5737] 的 `192.0.2.0/24`、`198.51.100.0/24`、`203.0.113.0/24` 与 `2001:db8::/32`，并明确标注为教学用示例地址。

15.14 参考资料

- RFC Editor, [RFC 1034] *Domain Names - Concepts and Facilities*: https://www.rfc-editor.org/rfc/rfc1034.txt
- RFC Editor, [RFC 1035] *Domain Names - Implementation and Specification*: https://www.rfc-editor.org/rfc/rfc1035.txt
- RFC Editor, [RFC 2181] *Clarifications to the DNS Specification*: https://www.rfc-editor.org/rfc/rfc2181.txt
- RFC Editor, [RFC 2308] *Negative Caching of DNS Queries (DNS NCACHE)*: https://www.rfc-editor.org/rfc/rfc2308.txt
- RFC Editor, [RFC 6891] *Extension Mechanisms for DNS (EDNS(0))*: https://www.rfc-editor.org/rfc/rfc6891.txt
- RFC Editor, [RFC 8020] *NXDOMAIN: There Really Is Nothing Underneath*: https://www.rfc-editor.org/rfc/rfc8020.txt
- RFC Editor, [RFC 8499] *DNS Terminology*: https://www.rfc-editor.org/rfc/rfc8499.txt
- RFC Editor, [RFC 8767] *Serving Stale Data to Improve DNS Resiliency*: https://www.rfc-editor.org/rfc/rfc8767.txt

第十六章

DNSSEC RFC 族

第三篇第 10 章讲的是怎么配验证器：信任锚（trust anchor）从哪来、放行与拒绝怎么调、坏签名怎么观测。本章只讲条文逻辑与设计动机——**为什么这套协议被设计成”验证失败必须拒绝”，而不是”尽量给个答案”**；为什么”签名被剥离”在这些条文里等同于降级攻击；以及为什么一个 2005 年定稿的协议族，在此后十几年里还需要七八份 RFC 来澄清、补参数、收编运维经验。

贯穿全章的区分，全书一致：**DNSSEC 提供数据完整性与来源认证**，DoT / DoH / DoQ 提供**传输加密与隐私**，查询名最小化减少元数据外泄，过期缓存应答提供可用性。四者互不替代。把任何一项当成”已经加密了所以数据可信”，会在排查时得出完全错误的结论。

与第 15 章一样，本章按”它要解决什么→关键条文→对实现的约束→常见误读”展开关键 RFC；运维类与历史类小节合并叙述，避免为凑格式而堆砌名词。

16.1 DNSSECbis: [RFC 4033] / [RFC 4034] / [RFC 4035]

16.1.1 它要解决什么

最初的 DNSSEC（[RFC 2535] 及其一组配套文档）在 1990 年代末定稿，但部署几乎没推开：密钥管理复杂、与动态更新和委派的组合不清晰、应答尺寸与实现负担都偏重。2005 年 3 月发布的 [RFC 4033]、[RFC 4034]、[RFC 4035] 合称 **DNSSECbis**，一次性废弃 [RFC 2535]、[RFC 3008]、[RFC 3090]、[RFC 3445]、[RFC 3655]、[RFC 3658]、[RFC 3755]、[RFC 3757]、[RFC 3845]，并更新 [RFC 1034]、[RFC 1035]、[RFC 2181]、[RFC 2308] 等多份基础文档。三份分工明确：

- [RFC 4033] *DNS Security Introduction and Requirements*: 威胁模型、术语、验证器的四种状态、以及”最后一跳”问题；
- [RFC 4034] *Resource Records for the DNS Security Extensions*: DNSKEY、RRSIG、NSEC、DS 四种记录类型及其线格式；
- [RFC 4035] *Protocol Modifications for the DNS Security Extensions*: DO / CD / AD 标志位、验证流程、名字服务器的发送与应答规则。

设计目标不是”让 DNS 保密”，而是让**应答数据可被验证**：任何人都能伪造报文，但伪造者拿不到私钥，因此签不上名。

16.1.2 关键条文

[RFC 4034] 的四种记录构成一条链：

- **DNSKEY**：区里发布的公钥，flags 区分 KSK（密钥签名密钥，SEP 位）与 ZSK（区签名密钥）。
- **RRSIG**：对某个 **RRset** 的签名，签的是第 15 章 [RFC 2181] 确立的原子单位，不是单条记录；RRSIG 自带签名有效期（inception / expiration），过期即不可验证。
- **DS**（Delegation Signer）：父区里对子区 DNSKEY 的摘要，把信任从父区传到子区。这是信任链的关键——验证器不必预先知道每个区的密钥，只需知道根。
- **NSEC**：按规范名排序的”下一个名字”链，用来**证明某个名字或某个类型不存在**。

[RFC 4033] 定义验证器可判定的四种状态，这四个词在本书中保持一致：

状态	判定条件
Secure	有信任锚、有信任链、应答中所有签名均能验证
Insecure	有信任锚与信任链，但在某个委派点有签名证明 DS 不存在，该分支被证明性地未签名
Bogus	有信任锚且委派指示应签名，但验证失败（缺签名、签名过期、算法不支持、NSEC 说该有却没有等）
Indeterminate	没有可覆盖该部分的信任锚；这是未配置验证时的默认状态

[RFC 4035] 规定三个标志位：**DO**（DNSSEC OK，在 OPT 伪 RR 里，请求 DNSSEC 相关记录）、**CD**（Checking Disabled，存根请求”别替我验证”）、**AD**（Authentic Data，服务器告知”我已验证这些数据为真”）。它还明确：安全感知的名字服务器 **MUST NOT** 在 Answer 与 Authority 段里有任何未判定为真的数据时置 AD 位；对 Bogus 数据，用 RCODE=2（SERVFAIL）通知非验证型存根解析器。

16.1.3 对实现的约束

验证器必须自己能完成从信任锚到目标区的整条链构建：向父区取 DS、向子区取 DNSKEY、用 KSK 验证 DNSKEY RRset、用 ZSK 验证数据 RRset、检查 RRSIG 有效期与算法支持、用 NSEC/NSEC3 验证否定应答。这意味着**它必须能自己向根发起 priming 查询取根的 DNSKEY**。

由此直接推出第三篇那条运维约束：把 `forward-zone "."` 配成”一切都交给上游”的转发器，与”自己配置信任锚”在架构上是二选一。转发器把根查询也交出去了，无法完成自己的信任锚 priming，日志会持续报 `failed to prime trust anchor`，并间歇性全量 SERVFAIL。这不是软件缺陷，是 [RFC 4035] 的验证流程在转发模式下无法闭环。

另一条约束是 [RFC 4035] §4 附近的 BAD cache 思路（[RFC 6840] 把它写得更明确）：对持续验证失败的 RRset 应有短时负面缓存，避免每次查询都重跑一遍昂贵的验证。但这属于性能与抗放大，不改变”必须拒绝”的结论。

16.1.4 常见误读

误读一：DNSSEC 加密了查询内容。完全没有。DNSKEY、RRSIG、DS、NSEC 都是明文记录，任何路径上的观察者都能读到完整 QNAME。加密传输要靠 DoT / DoH / DoQ（第 17 章）。

误读二：只要区里有 RRSIG 就是 Secure。没有信任锚时状态是 Indeterminate，验证器压根不验证；坏签名域也照样返回答案。这就是第三篇记录的”无 trust anchor = 透传不验证”。

误读三：AD 位可以由客户端自己置。[RFC 6840] §5.7 专门澄清：服务器应忽略查询里的 AD 位，它只由服务器在应答里根据验证结果设置。客户端置 AD 不能”要求”服务器验证。

16.2 验证器为什么必须 fail-closed

这一节是本章的核心，也是 DNSSEC 与几乎所有”尽力而为”的网络协议最不一样的地方。

16.2.1 它要解决什么

DNS 的传统行为是 **fail-open 的可用性优先**：一个权威服务器没应答，就换下一个；TTL 到了就重查；实在不行返回 SERVFAIL，客户端自己重试或换解析器。DNSSEC 若沿用这个思路，会出现一个致命的退化：攻击者只要在链路上**把签名相关记录全部剥掉**（去掉 RRSIG、去掉 DNSKEY、把 DO 位清掉、或干脆伪造一个”该区未签名”的 Insecure 证明），验证器就会从”验证失败”滑到”没什么可验证”，然后把未经验证的数据放行。

换句话说，安全机制的强度取决于**它能否被绕过**，而不是它能否正确工作。一个”验不过就放行”的验证器，等于给攻击者发了一张”如何让我闭嘴”的说明书。

16.2.2 关键条文

条文本身并不写”fail-closed”这个词，它分散在几处：

- [RFC 4033] 把 **Bogus** 定义为”有信任锚且委派指示应签名、但验证失败”，并明确该规范只定义”如何用 SERVFAIL 通知非验证型存根解析器数据为 Bogus”——即 Bogus 的对外表现就是拒绝服务，不是降级返回。

- [RFC 4035] 要求 AD 位只在所有相关数据被判定为真时才置位；不置 AD 不等于”可以给数据”，而是”不能宣称可信”。
- [RFC 4035] 明确对验证失败的应答使用 RCODE=2，并建议对重复验证失败做短时缓存（[RFC 6840] §3.1 把这些整理成 BAD cache 的实现要求）。
- [RFC 4033] 的 Insecure 判定要求**有签名证明 DS 不存在**：一个区”看起来没签名”与”被证明未签名”是两件事，只有后者才允许放行。这一条正是抵御”伪造未签名”的关键闸门。

16.2.3 对实现的约束

因此验证器在三种状态之间的行为必须是：

配置状态	行为	结论
无信任锚	透传，不验证	坏签名域也返回答案，等于没有 DNSSEC
permissive	记录日志但放行	可观测，但不提供保护
真验证（`val-permissive-mode: no`）	坏签名→ SERVFAIL，好签名带 AD 标志	唯一能提供保护的模式

第三篇里的 `harden-dnssec-stripped: no` 之所以被列为高危，条文依据就在这里：在有信任锚的前提下，它允许”本应签名却看不到签名”的应答被接受，等于主动把 [RFC 4033] 的 Bogus 判定改成放行。

同理，`harden-below-nxdomain` 只对 **DNSSEC-secure 的 NXDOMAIN** 生效：无签名区的 NXDOMAIN 不构成可验证的否定证明，验证器没有依据去做子树剪枝。这不是开关失灵，是”没有可验证的信任链时不能凭空制造结论”的必然结果。它跟 [RFC 8020] 的 NXDOMAIN cut 是两条独立的线，别混。

16.2.4 常见误读

误读一：SERVFAIL 是验证器坏了。对 Bogus 而言 SERVFAIL 是**规定的正确结果**。排查方向应该是查信任锚、查时间同步（RRSIG 有效期对时钟极敏感）、查算法支持、查上游有没有剥签名，而不是关掉验证。

误读二：permissive 模式是”温和上线”，可以长期留着。它只是把 Bogus 从”拒绝”改成”记一笔日志然后放行”。长期运行时，攻击者不需要绕过日志，只需要让数据通过。

误读三：fail-closed 太激进，会影响可用性。这是真实的取舍，但应由**过期缓存应答**（[RFC 8767]，第 15 章）与信任锚滚动机制来补，而不是靠放宽验证判定。用可用性机制补可用性、用安全机制保安全，边界不能交叉。

16.3 被剥离签名为什么等于降级

16.3.1 它要解决什么

“降级”（downgrade）指攻击者不破解密码学，而是让双方退回一个更弱、更容易攻击的通信模式。对 DNSSEC 来说，最省力的降级不是伪造 RRSIG（做不到），而是让验证器**看不到该有签名的证据**：

- 在应答里删掉 RRSIG 与 DNSKEY；
- 清掉 OPT 里的 DO 位，让上游不返回 DNSSEC 记录；
- 用伪造的 NODATA 冒充”该区没有 DS”（把 Secure 分支伪装成 Insecure）；
- 在链路上做明文 53 的中间人，直接改写应答。

这些手法与第 17 章的”静默明文降级”是同一类思维：不需要赢下密码学，只需要让保护机制**不被触发**。

16.3.2 关键条文

[RFC 4033] 的 Insecure 定义是防御的支点：Insecure 不是”没看到 DS”，而是”在某个委派点拿到了**带签名的** DS 不存在证明”。父区签名过的 NSEC/NSEC3 记录证明该委派未签名，验证器才能合法地放行未签名数据。缺少这份证明，却观察到本应签名的区没签名，只能是 Bogus。

[RFC 6840] §4.3（Check for CNAME）、§4.4（Insecure Delegation Proofs）、§5.11（Mandatory Algorithm Rules）等多处都在堵同一类漏洞：如果实现允许”部分应答”“少一条签名”“引用里少个 DS”就滑向放行，降级就成立。

§5.11 尤其重要——验证器不能因为”我不认识这个算法”就把整个区当成未签名，算法支持与否是验证器的能力问题，不是区的安全状态。

[RFC 4035] 关于 AD 位的规则也从另一侧堵漏：AD 是服务器**自己验证过**才能置的。如果上游悄悄把签名剥了，本地验证器不会置 AD，下游据此可以判别”这一跳没被验证过”。

16.3.3 对实现的约束

因此实现必须把”未见签名”与”证明未签名”严格分开，并在应答组装时防止签名被上游或中间设备剥掉。这就是 `harden-dnssec-stripped` 这类加固项存在的理由：它要求验证器在”信任链说该有签名、应答里却没有”时坚持拒绝。

还有一层与传输层的关系值得写清：**加密传输能防剥签名，但不能替代 DNSSEC**。DoT / DoH / DoQ 保护的是客户端到某一台解析器之间这一跳；解析器自己去递归时，如果那一跳仍是明文 53，中间人依然可以在权威应答到达解析器之前动手。反过来，DNSSEC 也保护不了”查询内容被谁看见”。两条线各管一段，任何一条单独成立都不构成完整保护。

第三篇里”上游超时后允许回退到明文 53 会让整条查询流裸奔”的结论，在本章的语言里就是：降级把保护面从”有完整性校验 + 有传输加密”退回到”什么都没有”，而应用层很难从返回码里察觉这件事。

16.3.4 常见误读

误读一：有 RRSIG 才算安全，没有就是不安全。正确说法是：有信任链且应答可验证才算 Secure；有签名的 Insecure 证明才是 Insecure；其余情况要么 Bogus 要么 Indeterminate。

误读二：把 DO 位当安全开关。DO 只是”请把 DNSSEC 记录给我”的请求标志（[RFC 4035] 借 [RFC 6891] 的 OPT TTL 字段承载）。清掉它确实会造成剥签名，但加上它并不能验证任何东西。

误读三：用了 DoH 就不用管 DNSSEC 了。DoH 保证”我到这台解析器之间没被改”，不保证”这台解析器给我的数据本来就是真的”。

16.4 信任锚滚动：[RFC 5011]

16.4.1 它要解决什么

信任锚是验证器的起点，通常很少几个（根的 KSK）。但如果这个锚需要更换——密钥泄露、算法升级、例行轮换——分布在全世界、无人值守的验证器怎么知道？逐台手工改配置不可行。[RFC 5011]（2007，*Automated Updates of DNS Security (DNSSEC) Trust Anchors*）定义了一套**自动、经认证、经授权**的滚动机制。

16.4.2 关键条文

- 基于当前锚所建立的信任，把**新的候选锚**加入同一信任点；新密钥先以”待观察”状态出现。
- 引入 `addHoldDownTime`（观察期）与 `removeHoldDownTime`（撤销期）两类计时：新锚必须在连续若干个刷新周期内持续以经签名的方式出现，才能被提升为有效锚；撤销一个锚同样要走完等待期。
- 机制提供对 N 个密钥中 N − 1 个被攻破的容忍：只要还有一个可信的当前锚，就能授权加入或撤销其他锚。
- [RFC 5011] 明确说自己改变的是**解析器的管理行为，不是解析行为**：验证流程不变，变的是”锚集合怎么随时间演化”。

16.4.3 对实现的约束

实现必须持久化锚的状态机（当前有效 / 待添加 / 待撤销），并在重启后继续计时——把状态只放在内存里，等于每次重启都把观察期清零。这也是为什么自动信任锚需要一个**可写且持久**的锚文件，而不是只读的配置片段。

时间同步在这里同样是硬依赖：观察期与 RRSIG 有效期都按墙钟计算。第三篇提到的时钟漂移导致间歇性全量 SERVFAIL，在 [RFC 5011] 语境下会更隐蔽——锚状态可能被误判。

16.4.4 常见误读

误读一：[RFC 5011] 用于所有密钥轮换。它管的是**信任锚**（通常是 KSK / 根锚）。区内部的 ZSK 轮换、以及 KSK/ZSK 的常规滚动时序属于 [RFC 6781] / [RFC 7583] 的范围。

误读二：看到新 DNSKEY 就该立刻启用。恰恰相反，观察期是为了抵御”攻击者短时间注入一个假锚”。立刻启用等于把自动滚动变成单点注入面。

16.5 澄清与实现说明：[RFC 6840]

16.5.1 它要解决什么

DNSSECbis 三份文档发布后，各家实现在细节上又分叉了：NSEC3 支持与否、SHA-2 支持、AD/CD 位该怎么设、DS 摘要算法不认识时怎么办、CNAME 链怎么验、嵌套信任锚怎么处理。[RFC 6840]（2013, *Clarifications and Implementation Notes for DNS Security (DNSSEC)*）把部署经验里踩过的坑逐条写成实现说明。

16.5.2 关键条文

按小节看，最影响验证器正确性的几条：

- **§2.1 / §2.2**：明确实现应支持 NSEC3 与 SHA-2 系列，避免把”不支持”误判成”未签名”。
- **§3.1**：实现 BAD cache（验证失败短时负面缓存），既抗放大也避免日志洪泛。
- **§4.1–§4.4**：否认存在证明的澄清、ANY 查询的验证、CNAME 检查、Insecure 委派证明——这几节基本是在堵 16.3 讲的降级路径。
- **§5.2 / §5.11 / §5.12**：未知的 DS 摘要算法、强制算法规则、忽略未知密钥的多余签名。核心立场一致：**验证器的能力缺口不能成为放行的理由**。
- **§5.6–§5.9**：DO 位应答设置、查询里的 AD 位应被忽略、应答里 AD 位的设置条件、以及”验证器应在每一次上游查询都置 CD 位”。最后一条的理由写得很实在：若上游某台已验证的解析器因 Bogus 返回 SERVFAIL，而 [RFC 2308] 允许对 SERVFAIL 短时缓存，本地验证器需要能用 CD 位重新取回原始数据自己判定，而不是继承别人的失败结论。
- **§5.10**：嵌套信任锚的处理。

16.5.3 对实现的约束

[RFC 6840] 把 [RFC 4035] 里若干”可以自行理解”的地方变成 SHOULD / MUST 级别的实现建议，实际效果是**让不同实现的验证结果趋于一致**。对运维的含义是：两个解析器对同一个坏签名域给出不同结论时，先查实现版本与算法支持，再怀疑配置——差异往往来自澄清条款是否被实现。

16.5.4 常见误读

误读一：澄清 RFC 是可选的”最佳实践”。其中不少条目直接决定”这个应答该不该被判为 Bogus”，属于正确性范畴。

误读二：CD 位是给客户端关验证用的。客户端置 CD 表示”请把原始数据给我，我自己判”；而 [RFC 6840] §5.9 讲的是**验证器对上游**也应置 CD，方向不同，别混。

16.6 算法信号与算法要求：[RFC 6975] / [RFC 8624]

16.6.1 它要解决什么

区管理员想从旧算法换到新算法（例如 RSA/SHA-1 → ECDSA P-256），但不敢贸然切换：万一大量验证器还不支持新算法，整区就会对它们变成 Bogus。[RFC 6975]（2013, *Signaling Cryptographic Algorithm Understanding in DNS Security Extensions (DNSSEC)*）让验证中的端系统通过 EDNS 选项向服务器**声明自己支持哪些签名与摘要算法**，使管理员能观测新算法的覆盖率，判断何时可以完成滚动。

[RFC 8624]（2019, *Algorithm Implementation Requirements and Usage Guidance for DNSSEC*, 废弃 [RFC 6944]）则从另一侧给出实现要求表：哪些算法签名侧 MUST NOT、哪些验证侧 MUST、哪些只是 RECOMMENDED。

16.6.2 关键条文

[RFC 8624] §3.1 的结论值得完整记住（这是可核验的表格，不是推测）：

编号	助记符	DNSSEC 签名	DNSSEC 验证
1	RSAMD5	MUST NOT	MUST NOT
3 / 6	DSA / DSA-NSEC3-SHA1	MUST NOT	MUST NOT
5 / 7	RSASHA1 / RSASHA1-NSEC3-SHA1	NOT RECOMMENDED	MUST
8	RSASHA256	MUST	MUST

编号	助记符	DNSSEC 签名	DNSSEC 验证
10	RSASHA512	NOT RECOMMENDED	MUST
13	ECDSAP256SHA256	MUST	MUST
14	ECDSAP384SHA384	MAY	RECOMMENDED
15 / 16	ED25519 / ED448	RECOMMENDED / MAY	RECOMMENDED

表里那个”签名侧不建议、验证侧仍 MUST”的组合是刻意的：老算法虽不该再用于新签名，但仍有存量区在用，验证器必须能验证它们，否则会造成大面积 Bogus。这正是 16.5 §5.11”能力缺口不能成为放行理由”的反面——**该支持的必须支持，不支持的不能假装未签名**。

[RFC 6975] 的信号本身是 EDNS0 选项，属于 [RFC 6891] 的扩展机制；它不改变验证结果，只提供部署侧的观测数据。

16.6.3 对实现的约束

实现要维护算法支持表并随版本更新；同时注意 [RFC 8624] 是**时点性文档**，算法推荐会随密码学进展被后续 RFC 更新。本书不复述更晚的版本结论，具体实现支持哪些算法以所用版本的官方手册为准。

应答尺寸也是算法选择的现实约束：ECDSA 相比 RSA 密钥与签名更短，更容易装进 1232 字节的 EDNS 载荷而不分片（第 15 章 [RFC 6891]）。算法不只是密码学问题，也是 MTU 问题。

16.6.4 常见误读

误读一：[RFC 6975] 的信号让服务器按客户端能力”挑算法”。不是，它用于统计覆盖率以决定滚动时机，区的签名算法不是按单个查询协商的。

误读二：验证器只实现”当前推荐”的算法就够。恰恰要兼容存量算法，否则会把仍在使用老算法的有效区误判成 Bogus。

16.7 签名区的否认存在：[RFC 7129] / [RFC 9276]

16.7.1 它要解决什么

“证明一个东西不存在”比”证明它存在”难。DNSSEC 的 NSEC 链通过”按规范名排序，给出前后两个名字，说明目标落在两者之间”来证明不存在；代价是**允许区枚举**（zone walking）：顺着 NSEC 链一路走就能把区里所有名字列出来。NSEC3（[RFC 5155]）用哈希名字代替明文名字，并引入 salt 与迭代次数提高枚举成本。

[RFC 7129]（2014，*Authenticated Denial of Existence in the DNS*，Informational）是背景与机理说明：为什么否定应答需要最多两条 NSEC（NSEC3 时三条）、通配符与 Empty Non-Terminal 怎么参与、NODATA 与 NXDOMAIN 的证明有何不同。[RFC 9276]（2022，*Guidance for NSEC3 Parameter Settings*）则纠正了一个长期的运维误区——**堆高迭代次数并不会带来相应的安全收益**。

16.7.2 关键条文

[RFC 9276] 的核心论证：

- 名字在 NSEC3 记录里本来就被哈希隐式”加盐”；额外 salt 只有在**在线攻击阶段持续变化**才能抬高成本，而换 salt 要求重建整条哈希链、且必须等链上每个名字都更新完才能生效，操作极其复杂，频繁换盐不现实。
- 额外迭代次数带来的成本对防御方是线性、对攻击者是一次性预计算，收益远小于直觉；文档明确指出”applying additional hash iterations beyond the first”缺乏收益。
- 结论写得很直接：**若必须使用 NSEC3，迭代次数 MUST 取 0**。同时指出高迭代会让验证器与权威服务器都付出 CPU 代价，反而放大 DoS 面。

[RFC 7129] 说明了为什么验证器不能把”没拿到 NSEC 记录”当成”不存在证明缺失无所谓”——没有可验证的否认存在证据，NXDOMAIN 就不能被判为 Secure，这直接影响 `harden-below-nxdomain` 一类剪枝是否生效（见 16.2）。

16.7.3 对实现的约束

权威侧签名工具应默认不用 NSEC3 的高迭代与频繁换盐；验证侧要对高迭代保持警惕（验证成本放大），并按 [RFC 8624] 的算法表处理 NSEC3 相关算法。运维侧要理解”用 NSEC3 防枚举”本身就是取舍：若区里名字并非秘密，用 NSEC 更简单；若确实需要防枚举，可用 NSEC3 的 opt-out 等机制，但参数不能靠”调大迭代”来补。

16.7.4 常见误读

误读一：NSEC3 是 NSEC 的加密版本。它是哈希化的否认存在机制，哈希是为了抬高枚举成本，不是加密，也不提供保密性。

误读二：迭代次数越大越安全。[RFC 9276] 专门推翻这一点，并给出 0 迭代的 MUST 级要求。

误读三：否认存在与 [RFC 8020] 的 NXDOMAIN cut 是一回事。前者是 DNSSEC 的密码学证明，后者是纯解析语义的子树剪枝；一个管”能不能信”，一个管”还要不要继续问”。

16.8 运维实践：[RFC 6781] / [RFC 7583]

16.8.1 它要解决什么

协议定稿之后，真正难的是”每天怎么跑”：密钥多久换一次、KSK 与 ZSK 分工、签名有效期多长、滚动时要等几个 TTL、父区 DS 什么时候更新、出错了怎么回滚。[RFC 6781]（2012，*DNSSEC Operational Practices, Version 2*，废弃 [RFC 4641]）是面向区管理员的实践集；[RFC 7583]（2015，*DNSSEC Key Rollover Timing Considerations*）专攻**滚动时序**，把各事件之间的依赖关系画成时间线。

16.8.2 关键条文

- KSK 与 ZSK 分离：ZSK 可较频繁滚动且不必惊动父区，KSK 滚动必须同步更新父区 DS，代价高、周期长。
- 滚动时序由 TTL 与签名有效期共同约束：新密钥发布后要等旧数据的 TTL 全部过期，才能让旧密钥退场；`double-signing`、`pre-publish` 等策略各有不同的等待窗口。
- [RFC 7583] 明确列出各参数（签名有效期、区刷新、父区 DS 的 TTL、验证器缓存）之间的数学关系，并指出**顺序搞错就会在一段窗口内让部分验证器判为 Bogus**。
- 两份文档都强调监控：签名临期、DS 与 DNSKEY 不一致、信任锚状态，都应在变成故障前被发现。

16.8.3 对实现的约束

对递归解析器运维者，最实用的一条是理解”远端滚动会波及本地”：某个上游区做 KSK 滚动时，若本地验证器缓存了旧 DNSKEY / 旧 DS，且滚动窗口内有验证失败，表现就是间歇性 SERVFAIL——而不是本地配置坏了。这也是 [RFC 5011] 自动锚状态机必须持久化的原因之一。

时钟同步在这里第三次出现：RRSIG 的 inception/expiration、滚动等待窗口、[RFC 5011] 的 hold-down，全都按墙钟计。把 NTP 当成”最好有”的设施，是 DNSSEC 运维里最常见的低估。

16.8.4 常见误读

误读一：密钥滚动是权威侧的事，与递归侧无关。递归侧若锚状态或缓存没跟上，会直接把滚动窗口变成 Bogus 窗口。

误读二：KSK 与 ZSK 可以随便互换滚动。二者对父区 DS 的依赖不同，时序约束不同。

16.9 DLV 的历史与退役：[RFC 4431] / [RFC 5074]

16.9.1 它要解决什么

根区在 2010 年签名之前，DNSSEC 面临鸡生蛋问题：验证器有根锚也没用，因为根下面的委派大多没有 DS，信任链从第一步就断了。[RFC 4431]（2006，*The DNSSEC Lookaside Validation (DLV) DNS Resource Record*，Informational）定义了 DLV 记录类型，[RFC 5074]（2007，*DNSSEC Lookaside Validation (DLV)*，Informational）定义了整套旁路验证机制：**在 DNS 委派链之外发布信任锚**，让验证器对”祖先未签名或不发布 DS”的区也能验证。

16.9.2 关键条文

- DLV 记录本质是放在一个旁路域里的 DS，验证器在找不到常规信任链时，去 DLV 域查对应名字的锚。
- [RFC 5074] 明确说这是**过渡机制**：为根与 TLD 尚未全面签名的阶段提供起点；同时提到它可用于”在一个组织内集中管理多个验证器的锚”或”外包信任锚管理”。
- 两份都是 Informational，不是 Standards Track——这个类别本身就说明它们不是长期架构。

16.9.3 对实现的约束与退役

根区于 2010 年签名、根的信任锚可通过 IANA 公开发布后，常规信任链在绝大多数路径上可用；DLV 的历史任务完成。此后各实现陆续移除 DLV 支持，公共 DLV 注册服务停止运营；**新部署不应再依赖 DLV**，历史配置里的 DLV 锚应清理，否则会留下一个”以为在验证、其实查的是一个已不存在的旁路域”的假象。

本章不复述具体停用日期与版本号——这类事实以各实现的发布说明与官方手册为准；可核验的根信任锚信息以 IANA 的 `root-anchors.xml` 为准（见《参考资料》）。

16.9.4 常见误读

误读一：DLV 是 DNSSEC 的一部分。它是特定历史阶段的补丁，不是 [RFC 4033]/[RFC 4034]/[RFC 4035] 定义的协议组件。

误读二：留着 DLV 配置”总没坏处”。一个指向已退役旁路域的锚配置，会让配置审计失真，也会在排查时把注意力引向错误方向。

16.10 根 KSK 轮换（2018）

16.10.1 它要解决什么

根 KSK 是全球所有 DNSSEC 验证器共同的起点。它一旦更换，任何没有及时获得新锚的验证器都会把整个签名 DNS 判成 Bogus——这是 DNSSEC 部署史上影响面最大的一次协调动作。2018 年的轮换是**根 KSK 自 2010 年根区签名以来的首次更换**，其意义不在密码学，而在验证 [RFC 5011] 设计的自动化滚动机制能不能在真实世界跑通。

16.10.2 关键条文与事实

可核验的锚数据来自 IANA 的 `root-anchors.xml`：KeyTag **19036**（2010 年起的初始锚，其 `validUntil` 为 2019-01-11）与 KeyTag **20326**（`validFrom` 为 2017-02-02），两者均为算法 8 / 摘要类型 2。新锚提前一年多发布并进入 [RFC 5011] 的观察期，正是 16.4 讲的 hold-down 机制在根这一层的实例。

轮换过程中还配套处理了”根区应答变大”的工程问题：新 KSK 引入期间根 DNSKEY 应答尺寸上升，触发了对 TCP 回退、EDNS 载荷与分片路径的再检查。这与第 15 章 [RFC 6891] 的逐跳载荷协商是同一条线。

16.10.3 对实现的约束

- 验证器必须**持久化 [RFC 5011] 状态**并在轮换窗口内正常运行；长期关机或只读锚文件的实例，可能错过观察期。
- 静态写死旧锚、又不启用自动滚动的配置，在旧锚失效后会进入大面积 SERVFAIL。
- 这次事件把”信任锚不是一次性配置、而是一个随时间演化的状态机”变成了运维常识。

16.10.4 常见误读

误读一：轮换是”换一把更长的密钥”。关键不是密钥强度，而是全球验证器能否在不停机的情况下同步更换起点。

误读二：2018 年之后根锚就固定了。不是。`root-anchors.xml` 里还有更晚 `validFrom` 的条目，说明锚集合继续演化；具体实现与当前状态以 IANA 数据与所用版本的官方手册为准。

16.11 条文之间的因果链

把本卷的 RFC 按”为什么需要下一份”重排，逻辑比编号列表清楚：

1. [RFC 4033]/[RFC 4034]/[RFC 4035] 定义信任链、四种状态与三种标志位，确立 **Bogus → SERVFAIL** 的基本契约。
2. 契约一旦确立，“能否被绕过”就成了主要攻击面：16.2 与 16.3 分别从 [RFC 4033] 的 Insecure 定义、[RFC 4035] 的 AD 位规则、[RFC 6840] 的一系列澄清里推导出 **fail-closed** 与 **剥签名即降级**。
3. 信任锚不能是静态常量，于是有 [RFC 5011] 的自动滚动；滚动在真实世界的检验是 2018 年根 KSK 轮换。
4. 算法会老化，于是有 [RFC 6975] 的覆盖率信号与 [RFC 8624] 的实现要求表；应答会变大，于是回到 [RFC 6891] 的载荷与 MTU 约束。
5. 否认存在的证明有额外复杂度与参数误区，于是有 [RFC 7129] 的机理说明与 [RFC 9276] 的 0 迭代要求。
6. 日常运维的时序问题由 [RFC 6781] / [RFC 7583] 收编；根未签名时代的过渡方案 DLV 完成使命后退役。

这里给出的都是”开关为什么存在”。

16.12 一个设计哲学的代价：算法复杂度攻击

把本章的条文串起来会看到一条不太舒适的事实：[RFC 4035] 要求验证器尝试可用的密钥，[RFC 6840] 进一步要求尝试所有签名组合——这是为了在密钥轮换、算法轮换、多签名者场景下「尽量验证成功」，是 Postel 律在 DNSSEC 里的体现。但「尽力」没有上限时，就成了攻击者可以单向放大的算力入口：构造大量碰撞 key 与无效签名，让验证器做平方级无用功。这就是第 10 章 KeyTrap / NSEC3 两个 CVE 的协议根因，[RFC 9276] 对 NSEC3 迭代参数的限制也是同一类修正。

因此补丁的形态不是改协议，而是**给实现加工作量上限**：限制 key 碰撞数、每 RRSET 验证次数、NSEC3 哈希次数，并让验证过程可挂起。这也解释了为什么 Unbound 的加固史一以贯之是「限额 + 让出 CPU + fail-closed」，而不是增加更多验证尝试。理论层要记住的结论是：**安全协议的可用性也是安全属性**；一个能被单包拖垮的验证器，与一个不验证的验证器同样危险。

16.13 本章要点

- DNSSECbis 三份文档([RFC 4033]/[RFC 4034]/[RFC 4035],2005)废弃 [RFC 2535] 系列,定义 DNSKEY / RRSIG / DS / NSEC 与 Secure / Insecure / Bogus / Indeterminate 四态；**RRSIG 签的是 RRset**，源自第 15 章 [RFC 2181] 的原子单位定义。
- **验证器必须 fail-closed**：[RFC 4033] 规定 Bogus 以 SERVFAIL 通知，[RFC 4035] 规定 AD 位只在全部数据判定为真时置位。无锚 = 透传、permissive = 记日志放行，都不提供保护；`harden-dnssec-stripped: no` 在有锚时主动制造降级漏洞。
- **被剥离签名等于降级**：Insecure 要求父区**带签名的 DS 不存在证明**，“没看到签名”与”证明未签名”必须严格分开。[RFC 6840] §4/§5 多处堵的正是”少一条证据就滑向放行”。
- 加密传输防剥签名但不替代 DNSSEC；DNSSEC 保护数据与来源、DoT/DoH/DoQ 保护传输隐私、[RFC 8767] 补可用性，四者互不替代。
- [RFC 5011] 让信任锚成为可持久化的状态机（观察期 / 撤销期），2018 年根 KSK 首次轮换（旧锚 KeyTag 19036 →新锚 20326）是它在全球规模上的验证；静态锚或错过观察期的实例会大面积 SERVFAIL。
- [RFC 8624] 的算法表刻意让”签名侧 NOT RECOMMENDED、验证侧 MUST”并存；[RFC 9276] 要求 NSEC3 迭代次数取 **0**，堆高迭代缺乏安全收益却放大 CPU 与 DoS 面。
- [RFC 6781]/[RFC 7583] 把 KSK/ZSK 分工与滚动时序写成可计算的时间线；DLV（[RFC 4431]/[RFC 5074]）是根未签名时代的 Informational 过渡机制，已随根签名完成而退役。
- 转发器无法自己完成信任锚 priming，因此”转发器”与”验证器”在架构上二选一；这是 [RFC 4035] 验证流程的推论，不是软件缺陷。

16.14 参考资料

- RFC Editor, [RFC 4033] *DNS Security Introduction and Requirements*: https://www.rfc-editor.org/rfc/rfc4033.txt
- RFC Editor, [RFC 4034] *Resource Records for the DNS Security Extensions*: https://www.rfc-editor.org/rfc/rfc4034.txt
- RFC Editor, [RFC 4035] *Protocol Modifications for the DNS Security Extensions*: https://www.rfc-editor.org/rfc/rfc4035.txt
- RFC Editor, [RFC 5011] *Automated Updates of DNS Security (DNSSEC) Trust Anchors*: https://www.rfc-editor.org/rfc/rfc5011.txt
- RFC Editor, [RFC 6840] *Clarifications and Implementation Notes for DNS Security*: https://www.rfc-editor.org/rfc/rfc6840.txt
- RFC Editor,[RFC 8624] *Algorithm Implementation Requirements and Usage Guidance for DNSSEC*: https://www.rfc-editor.org/rfc/rfc8624.txt
- RFC Editor, [RFC 9276] *Guidance for NSEC3 Parameter Settings*: https://www.rfc-editor.org/rfc/rfc9276.txt
- IANA，根区信任锚数据（KeyTag / validFrom / validUntil）：https://data.iana.org/root-anchors/root-anchors.xml

第十七章

加密与隐私 RFC 族

这一章处理第三条线：**谁看见了查询**。它与前两条线不重叠，也不互相替代——DNSSEC 提供数据完整性与来源认证，查询名最小化（QNAME minimisation，QMIN）减少发给上级的元数据，DoT / DoH / DoQ / ODoH 提供传输加密与隐私。端口、证书、转发、fail-closed 与 DDR 压制的落地配置在第三篇第 11–14 章。这里回答”条文为什么这么写、实现因此被约束在哪、以及哪些读法是错的”。

全章对每份 RFC 采用统一的四段式：**它要解决什么→关键条文→对实现的约束→常见误读**。第四篇统一按 `[RFC XXXX]` 形式引用编号，不写完整 URL；可核验链接集中在章末《参考资料》。涉及具体发布年份、版本状态与实现默认值时，一律以 RFC Editor 索引与所用版本的官方手册对应章节为准，本章不猜测。书中出现的地址只取 [RFC 5737] 的 `192.0.2.0/24`、`198.51.100.0/24`、`203.0.113.0/24` 与 `2001:db8::/32`，域名只用 `example.net` / `example.org` 及其子名，均为教学用示例。

17.1 查询名最小化的两版规范：[RFC 7816] → [RFC 9156]

17.1.1 它要解决什么

未开 QMIN 时，递归解析器解析 `assets.cdn.example.org A` 会把完整 QNAME 依次送给根、TLD、以及每一级权威服务器。这些服务器各自只负责命名空间的一小段，本不需要知道完整名字；但完整名字让它们能长期积累”谁在查什么”的元数据，且不同层的运营者可以把日志拼起来。[RFC 7816] 提出的是**数据最小化**：向每一级只发送推进解析所必需的最少标签数，把”必须知道”限制在区切（zone cut）这一层。

[RFC 9156] 废弃 [RFC 7816]，不是推翻目标，而是修正算法：第一版的固定 QTYPE 策略在真实权威服务器上引发了足够的兼容问题，需要重新规定；同时把”遇到异常应答怎么办”写成明确的两种模式。具体发布年份见 RFC Editor 索引。

17.1.2 关键条文

- [RFC 7816] 的算法是：QNAME 逐标签增长，每次只问到当前已知的区切；**中间查询把 QTYPE 固定为 NS**，理由是解析到区切时本来就想要委派信息，且固定类型可以避免把原始 QTYPE 暴露给上级。
- [RFC 9156] 改为**保留原始 QTYPE**：中间查询只缩短 QNAME，不再替换 QTYPE。判断”这是引用（referral）还是最终应答”依据的是 Answer 段为空、Authority 段给出下级 NS 这一组合，与 QTYPE 无关。
- 为什么要改回去：NS 是**被特殊对待的类型**。不少权威服务器收到对某个中间名的 NS 查询时，会直接把本区 apex 的 NS RRset 放进 Answer 段并置 AA 位，而不是给引用；最小化解析器若把这种应答当成”已经到了最终区”，就会在错误的层级停下。此外 CNAME / DNAME、通配符、以及某些运营者按 QTYPE 做的限速与过滤，对不同 QTYPE 的答案并不一致，固定 NS 会让中间层看到的世界与最终查询要的答案脱节。保留原始 QTYPE 的代价是上级多知道一个类型，换来的是应答语义一致。
- [RFC 9156] 把回退策略分成 **relaxed** 与 **strict**：relaxed 在中间查询返回非预期的 RCODE（NXDOMAIN、REFUSED、SERVFAIL 等）或非引用式应答时，退回发送完整 QNAME 与原始 QTYPE 重新问一次；strict 不回退，直接依据 [RFC 8020] 的 NXDOMAIN cut 断言”该名字及其下所有名字都不存在”并结束解析。
- 两版都明确：最小化只作用于**解析器到权威侧**的迭代查询，不改变存根解析器到递归解析器这一跳，也不改变报文的明文属性。

17.1.3 对实现的约束

实现必须把”已知区切”作为状态维护：每收到一次引用，才把 QNAME 往下推进一个标签；不能靠预先猜域名有几段。对 CNAME / DNAME 链、通配符命中、以及权威侧返回 NODATA 的情形，要有明确的推进或回退规则，否则会停在半路。

strict 模式的正确性建立在”所有权威都遵守 [RFC 8020]“这个前提上。现实里存在违反它的区：父名返回 NXDOMAIN、更深的子名却挂着数据（某些大型 CDN 的 CNAME 区是典型）。最小化恰好先问中间那个”空父名”，strict 拿到 NXDOMAIN 就断定整条链不存在，于是整站的资源域名解析失败，页面样式与脚本全丢。更麻烦的是这个错误结论会被 [RFC 2308] 的负缓存按 SOA minimum 记住，改回 relaxed 之后**链路上的每一层缓存都要单独刷**才恢复。第三篇第 13 章记录的是参数处理，这里的条文结论是：出问题的是权威侧违反 [RFC 8020]，不是解析器多疑。

还有一条边界必须写清：**QMIN 不是加密**。它减少的是发给上级的标签数，抗的是”多个权威运营者长期聚合元数据”；最终那个权威服务器仍然必须看到完整 QNAME 才能给出答案，解析器自己也看到全部。若本地做了根区镜像，根这一跳本来就不出口，收益主要落在 TLD 与二级权威层——这与第三篇的实测观察一致。

17.1.4 常见误读

误读一：QMIN 让查询内容保密。它只缩短 QNAME，QTYPE、源地址、时序、报文长度都还在，链路上任何能看到这一跳的观察者都不受影响。

误读二：strict 更严格所以更隐私。 strict 的差别是”不回退”，不是”发得更少”；它以兼容性换掉了一次回退机会，遇到不守规矩的权威反而会把整条查询判死。

误读三：改回原始 QTYPE 是倒退。固定 NS 的初衷是少泄露一点类型信息，但它让应答语义失真；[RFC 9156] 是拿这一点理论收益换实现正确性。

误读四：relaxed 等于没开 QMIN。只在异常路径上回退，正常路径仍然是最小化的；把偶发回退当成”机制失效”，会误判观测日志。

17.2 DoT：[RFC 7858]

17.2.1 它要解决什么

存根解析器到递归解析器这一跳长期是明文 UDP/TCP 53：同一局域网、同一接入链路上的被动观察者能读到全部查询，主动观察者还能改写应答。[RFC 7858] 给出最小改动的答案——把 [RFC 1035] 的 DNS 线格式**原样**放进 TLS 记录，用专用端口承载，不引入 HTTP 层、不重新定义报文。

17.2.2 关键条文

- 传输是 TCP 上的 TLS，专用端口 853；DNS 报文沿用 [RFC 1035] 为 TCP 定义的两字节长度前缀分帧，首部、段结构与压缩指针原样保留，仅由 TLS 提供机密性、完整性与服务器认证。
- 客户端**必须**按 PKIX 校验服务器证书与其身份（主机名或预先约定的 SPKI 指纹）；不满足校验即连接失败，不允许”先加密起来再说”。
- 连接可以复用、可以会话恢复；多个查询可共用一个 TLS 连接，应答**不必按请求顺序返回**，接收方靠 DNS 报文首部的 ID 匹配。实现需要处理超时、半开连接与重连，以及一条连接上的并发上限。
- 报文长度会暴露查询规模，[RFC 7858] 因此指向 EDNS(0) 填充选项（填充策略见官方手册对应章节），让填充成为可用手段而非必选行为。
- 规范只定义**这一跳**；解析器拿到查询后如何向权威服务器迭代，不在本文范围内。

17.2.3 对实现的约束

服务端与客户端都要把”认证失败”当作硬失败，而不是把 TLS 握手失败降级成明文 53。第三篇反复强调的静默明文降级，条文对应就在这里：一旦允许回退，[RFC 7858] 提供的保护面归零，而应用侧几乎察觉不到。实现还需要为长连接设定空闲超时与并发上限，否则一个 TLS 连接就能占住解析器的一条工作槽。

填充不是免费的：它增加带宽，且填充长度要与 EDNS(0) 通告的载荷上限、以及所用承载的 MTU 一起算。DoT 本身跑在 TCP 上，不存在 UDP 分片问题，但填充后的报文若被同一实现复用于 UDP 侧承载（DTLS、DoQ），仍可能触发分片被丢→等超时→重试。填充是缓解长度画像的手段，不是必选项。

17.2.4 常见误读

误读一：853 上跑的就是”加密 DNS”，所以数据可信。加密与认证的是链路对端，DNS 应答数据的来源认证仍要靠 DNSSEC（第 16 章）。

误读二：连上 TLS 就完成了 [RFC 7858]。不校验证书等于只防被动嗅探、不防主动中间人；规范把身份认证写成了必须项。

误读三：DoT 保护整条解析路径。它只覆盖存根到这一台解析器；解析器往权威侧发出的迭代查询仍可能是明文 53。

17.3 DoH: [RFC 8484]

17.3.1 它要解决什么

[RFC 7858] 的专用端口易于被按端口识别、限速或阻断。[RFC 8484] 把 DNS 报文作为 HTTP 消息体、经 HTTPS 送出，使其能与普通 Web 流量共用 443 端口，降低”仅因是 DNS 就被区别对待”的概率。代价是把 HTTP 层的元数据一起带进了 DNS 的信任模型。

17.3.2 关键条文

- DNS 线格式报文作为 `application/dns-message` 的实体：POST 放在请求体，GET 用 base64url 编码的 `dns` 查询参数；URI 模板形式（常见 `/dns-query`）由服务自行声明。
- 使用 HTTP/2 及以上更为合适；HTTP 缓存语义可用，但缓存键必须考虑 DNS 应答自身的 TTL，不能让 HTTP 缓存把短 TTL 的记录留得更久。
- 服务器认证沿用 HTTPS 的 PKIX 体系；规范没有另外定义一套 DNS 专用的身份模型。
- 与 DoT 一样，规范只定义**存根到端点**这一跳，端点之后的递归与迭代不在范围内。

17.3.3 对实现的约束

端点运营者同时掌握查询内容、客户端 IP、以及 HTTP 层信息（SNI、ALPN、User-Agent、可能的 Cookie 与鉴权头）。因此实现侧必须允许”不写这些字段”：不注入 Cookie、不把鉴权 token 当成跟踪标识、不把客户端指纹写进长期日志。第三篇把 DoH 前端与 Unbound 后端分开，条文理由就在这里——HTTP 层能收集的东西远多于 DNS 层，日志归属要在架构上切清楚。

同一 443 上 Web 与 DoH 共存，也让”按 IP 或按端口封禁”既容易误伤又容易漏：端点可能挂在共享 IP、CDN 或 anycast 之后，IP 与”这是不是一个 DoH 端点”不再一一对应。这一点在 17.11 会展开。

17.3.4 常见误读

误读一：DoH 是匿名。它把查询从本地网络观察者手里移到了端点运营者手里，并且新增了 HTTP 元数据；谁被信任变了，被收集的量没变少。

误读二：混在 443 里就查不出来。SNI、目标 IP、连接时序、报文长度与 HTTP 特征仍可观测，“与 Web 同形”降低的是分类的确定性，不是可见性。

误读三：DoH 端点天然不记日志。记不记是运营策略，不是协议保证；[RFC 8932] 整份文档都在处理这个问题。

17.4 DoQ: [RFC 9250]

17.4.1 它要解决什么

DoT 的 TCP + TLS 在丢包、握手往返与队头阻塞上有成本；DoH 又背了 HTTP 层的复杂度。[RFC 9250] 用 QUIC 承载 DNS：保留 TLS 1.3 的加密与认证，换用 UDP 之上的多路复用流，让多个查询互不阻塞，并支持连接迁移与更快的连接建立。

17.4.2 关键条文

- 使用**专用 QUIC 连接**，ALPN 标识与 IANA 分配的 UDP 端口见 RFC Editor 原文（具体编号以官方手册对应章节为准）；不复用 HTTP/3 的语义。

- 每个 DNS 查询映射到一条 QUIC 双向流，流本身提供顺序与对应关系；同一连接可并发多条流，DNS 报文首部里的 ID **必须置 0**，不再承担匹配职责。
- QUIC 内置 TLS 1.3，加密与服务器认证是强制的；没有”不加密的 DoQ”这一档，也没有明文回退档。
- 0-RTT 数据、空闲超时、流数与流量控制都有对应约束；实现要按 QUIC 的资源模型做限额，防止一条连接独占解析器。
- 与 DoT / DoH 相同，只覆盖存根到解析器这一跳。

17.4.3 对实现的约束

fail-closed 在这里不是可选项：QUIC 握手失败、被中间设备丢弃、或端口不可达时，客户端若自动退回明文 53，等于把 [RFC 9250] 的保护整段撤掉。第三篇记录的实测结论也指向同一方向：跨境链路上 UDP / QUIC 被 QoS 时，同一路径 TCP 承载更稳（实测 1MB 下载差距在三个数量级附近），因此”加密传输选哪条承载”是链路实测问题，不是协议优劣的口号。

服务端需要把”一条 QUIC 连接上的并发流数”纳入与 DoT 长连接同等的资源预算；连接迁移让基于五元组的限速与会话绑定失效，策略要按连接 ID 而不是按源端口来做。

17.4.4 常见误读

误读一：QUIC 用 UDP 所以更快、一定更好。在丢包或被限速的 UDP 路径上，它可能比 TCP 承载更差；协议特性不等于链路实测结果。

误读二：0-RTT 等于零风险。0-RTT 数据有重放语义上的限制，实现要按 QUIC 的约束处理，不能把任何查询都塞进早期数据。

误读三：DoQ 是”DoH over QUIC”。它是独立的专用 QUIC 映射，不经过 HTTP 层，也没有 HTTP 方法、路径与状态码。

17.5 使用配置：[RFC 8310]

17.5.1 它要解决什么

[RFC 7858] 定义了”怎么把 DNS 装进 TLS”，但没回答一个部署问题：**当加密连不上时该怎么办**。现实中证书过期、端口被封、对端只支持旧版本都会发生；若每个实现自己决定”要不要退回明文”，用户以为自己开着加密，实际却在裸奔。[RFC 8310] 把使用方式收敛成两份可声明的 profile，并明确谁该对身份负责。

17.5.2 关键条文

- **Strict Privacy（严格隐私）**：要求加密且要求服务器认证；认证或握手失败即失败，不允许回退到明文，也不允许”先连上再补校验”。
- **Opportunistic Privacy（机会隐私）**：尽可能使用加密，连不上时允许退回明文；它明确不承诺隐私，只承诺”条件允许时比明文好一点”，认证要求也相应放宽。
- 身份验证方式包括常规 PKIX 证书校验（对端点主机名）与 SPKI 指纹固定（pin），供无法或不想依赖公共 CA 体系的部署使用。
- 文档同时覆盖 DNS-over-TLS 与 DNS-over-DTLS 两类承载（DTLS 侧的规范编号见官方手册对应章节），并要求配置方能**明确声明**自己采用哪一档 profile。

17.5.3 对实现的约束

实现必须把 profile 作为**可观测、不可静默改变**的状态：严格档下任何失败都要以连接失败或 SERVFAIL 呈现，不能悄悄换端口、换协议、换证书校验策略。第三篇里 `forward-first: no`、fail-closed、“两波都失败就 SERVFAIL”的工程做法，条文依据是这一条——降级不是一种容错策略，而是把安全声明改写了。

运营侧还要区分”客户端声明的 profile”与”服务端实际提供的能力”：服务端只支持 opportunistic，客户端却以为自己在 strict，是配置审计里最常见的错配。

17.5.4 常见误读

误读一：opportunistic 是”轻量版严格档”。它不承诺隐私，只是可用性优先的尽力而为；把它写进”已启用加密传输”的清单会造成虚假结论。

误读二：证书校验可以等连上以后再做。严格档要求先认证再使用，认证失败的连接不能被拿来发查询。

误读三：[RFC 8310] 只管 DoH。它面向的是 DNS-over-TLS / DNS-over-DTLS 的使用配置；DoH 沿用 HTTPS 的 PKIX 体系，不在这份 profile 的定义范围内。

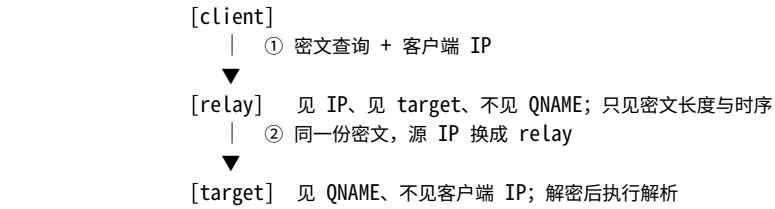
17.6 两跳分离：[RFC 9230]

17.6.1 它要解决什么

DoH 把查询从本地观察者手里交给了端点运营者：端点同时看到**客户端 IP** 与**查询内容**。换一个公共端点只是换一个信任对象。[RFC 9230] 想做的是把这两样信息拆到两个不同的参与方手里，使得单看任何一方的日志都无法把”谁”和”查了什么”关联起来。

17.6.2 关键条文

- 引入 **client → relay → target** 两跳。客户端用 target 的公钥（HPKE）把 DNS 报文封装成密文，连同一个加密后的应答密钥一起交给 relay；relay 只负责转发，看不到明文查询。
- **relay 见 IP 不见查询**：它知道客户端 IP、知道 target 是谁、能看到报文长度与时序，但拿不到查询内容。
- **target 见查询不见 IP**：它解密、执行解析、把应答封装回密文经 relay 交回客户端；它看到的是 relay 的 IP，不是原始客户端的。
- 分离成立的前提是 **relay 与 target 不共谋、不由同一主体控制**，且两者之间不能通过时序、长度、连接复用把请求重新配对；规范也明确这类流量分析不在密码学保护范围内。
- 与 DoH 一样，只解决存根到 target 这一跳；target 之后向权威服务器的迭代依然按普通 DNS 进行。
- [RFC 9230] 的作者来自 Apple、Fastly 与 Cloudflare，但走的是 **Independent Submission**：由 RFC Editor 自行决定发表，未经 IETF 背书，状态为 **Experimental**，不是 Standards Track。
- 客户端加密用 [RFC 9180] 的 HPKE，规范要求 DHKEM(X25519, HKDF-SHA256) / HKDF-SHA256 / AES-128-GCM。[RFC 9458] 的 OHTTP 把同一套 relay + 加密封装从 DNS 泛化到任意 HTTP 消息，是后来的 Proposed Standard；9458 的致谢写明设计基于 [RFC 9230]，9230 的实验概述也把 OHTTP 列为后续方向。时间线是 ODoH（2022，Experimental）在前，OHTTP（2024，Standards Track）在后。



17.6.3 对实现的约束

客户端需要取得并校验 target 的公钥配置（密钥如何分发、如何轮换、如何撤销是部署问题，见官方手册对应章节）；relay 必须不解密、不改写、不按查询内容做路由；target 必须不依赖客户端 IP 做应答策略，否则”看不到 IP”只是纸面说法。任何一方记录可关联的元数据（精确时间戳、稳定连接标识、长度序列），都会把两跳重新拼成一跳。

还有一条常被忽略的约束，值得单独论证：**自建可信 target 时收益约等于零**。ODoH 的收益定义在”target 是你不信任的第三方”这一场景——你用它解析，但不愿让它知道你是谁。若 target 是你自己搭建并完全信任的解析器，你本来就把全部查询交给了它，隐藏客户端 IP 对它没有意义；而为了让 relay 与 target 不共谋，relay 必须另找独立主体，可这个外部 relay 反而成了新的、能看到你 IP 的第三方。若为了省事把 relay 也建在自己这边，两跳就落在同一管理域里，日志一关联即还原，“分离”只剩名义。与此同时你多付了一跳延迟、一套密钥分发、一处新的失败点，还失去了本地解析器直接可观测的查询流。结论不是”ODoH 无用”，而是**它的适用前提是把查询交给一个你不愿暴露身份的第三方解析器**；自建可信解析器用 DoT / DoQ 加 QMIN 已经能拿到该拿的东西。

17.6.4 常见误读

误读一：ODoH 是 DoH 的加密加强版。它多出来的是**参与方分离**，不是更强的密码；DoH 的链路本来就由 TLS 加密。

误读二：relay 看不到查询，所以 relay 可以随意选。 relay 仍看到客户端 IP 与 target 身份，且能通过时序做关联；选一个与 target 有共同运营主体的 relay，等于把两跳合并。

误读三：ODoH 防流量分析。长度、时序、并发模式仍在，[RFC 9230] 并不承诺抗全局被动观察者；填充只能缓解。

误读四：ODoH 是某个开源项目发明的协议。它是有 RFC 编号的公开规范，开源项目只是实现。Experimental 与 Independent Submission 两个标记也说明它尚未进入 IETF 标准流程。

17.7　发现与压制：[RFC 9462]

17.7.1　它要解决什么

加密传输普及后，客户端仍卡在第一步：怎么知道**眼前这台**解析器（通常是网络自动分配的明文 53 地址）有没有对应的加密端点、端点叫什么、支持哪种协议。[RFC 9462] 的 DDR（Discovery of Designated Resolvers）让客户端向已配置的解析器自己发一个特殊名字的查询，从应答里拿到它声明的加密解析器信息，从而在不改网络配置的情况下升级到加密传输。

17.7.2　关键条文

- 保留名 `_dns.resolver.arpa.` 用于发现；客户端向**当前已配置的解析器**查询该名字的 SVCB / HTTPS RRset（借 [RFC 3597] 建立的未知类型透明转发能力，见第 15 章）。
- 应答里携带端点主机名、端口、ALPN、以及 DoH 的 URI 路径模板之类的参数，客户端据此选择自己支持的加密协议并验证指定解析器的身份。
- 规范要求客户端确认”指定解析器”与”当前解析器”之间存在可验证的关联（同一运营方、同一网络授权），并保留回退到原解析器的路径；发现结果是**声明**，不是强制切换指令。
- 特殊用途域名的本地处理框架来自 [RFC 6761]（第 15 章）；`resolver.arpa` 的语义由本 RFC 及 IANA 注册表确定。

```
(示意图，非配置；端点名均为 example.net / example.org 教学用示例)
[客户端] --① 查 _dns.resolver.arpa. SVCB--------> [当前解析器]
[客户端] <--② 声明的加密端点参数----------------- [当前解析器]
[客户端] --③ 按参数连接并校验身份--------------> [resolver.example.net]
```

17.7.3　对实现的约束

发现机制给客户端一条**自动绕过本地解析器**的路径，这正是本书在第三篇压制它的原因。本地解析器承担过滤、分流、DNSSEC 验证、QMIN、负缓存控制、dnstap 审计与 canary 判定；一旦客户端按 DDR 声明直连外部加密端点，这些策略全部落空，而且落空得悄无声息——网络侧只看到 443 流量，看不到查询。

压制手段是语义层的：让本地解析器对 `_dns.resolver.arpa.` 的发现查询返回 NODATA（本地服务区策略），客户端得不到声明就不会升级；浏览器侧再配合 canary 域（Firefox 的 `use-application-dns.net` 回 NXDOMAIN、Chrome 靠系统解析器匹配不到已知 DoH 提供商）让应用退回系统解析器。**这两者都是按名字做判定，不是按 IP 封 443**；后者的局限在 17.11 展开。注意压制的是”自动发现绕过”，不是”禁止用户显式配置加密解析器”——显式配置、且指向自己信任的端点，属于正常用法。

17.7.4　常见误读

误读一：DDR 会强制客户端改用加密解析器。它提供的是可验证的声明与升级路径，客户端有策略决定权，网络运营者也可以选择不提供声明。

误读二：压掉 DDR 就等于禁用了 DoH。只挡住了”由当前解析器自动发现”这一条路；用户手动配置的端点、应用内置的端点列表不受影响，所以还需要 canary 与出向策略配合。

误读三：`_dns.resolver.arpa.` 是某厂商的私有域名。它是注册过的特殊用途名，语义由规范与 IANA 注册表给定，本地处理依据是 [RFC 6761] 一类框架，不是”公网恰好没这个区”。

17.8　运营者视角：[RFC 8932]

17.8.1　它要解决什么

前面几份 RFC 都在定义”两个端点之间怎么传”。加密 DNS 一旦作为服务对外提供，一个新的汇聚点就出现了：服务运营者同时握有客户端 IP、查询内容、精确时间与连接特征，而且用户往往误以为”用了加密就不用管日志了”。[RFC 8932] 是写给**运营者**的建议集，处理的是协议条文管不到的策略面。

17.8.2 关键条文

- **数据最小化与保留期**：默认不收集可识别客户端的字段；确需收集时明确目的、范围与保留时长，并让外部可核验（隐私声明、透明报告）。
- **不要引入新的标识符**：慎用或不用 ECS；不注入 Cookie、不用 User-Agent / 鉴权头做长期关联；DoH 层的 HTTP 元数据要与 DNS 层一样受最小化约束。
- **传输与协议选择**：使用当前版本的 TLS、支持填充、支持 QMIN、不提供静默明文回退；对 DoH 端点还要注意 HTTP 缓存与 CDN 会不会把身份信息带到第三方。
- **可核查性**：把”记什么、留多久、谁能看、如何删除”写成公开策略，而不是只写在内部运维手册里。
- 文档性质是建议（具体类别与状态见 RFC Editor 索引），但其中多条与 [RFC 8310] 的严格档、[RFC 9076] 的威胁模型直接互为支撑。

17.8.3 对实现的约束

对自建解析器同样适用：**dnstap 之类的全量查询日志就是自己查询流的一份副本**。开全量日志却不写保留期（例如 30 天）、不限制访问，等于在”抗元数据聚合”这件事上于自己门口破功——上级权威少拿了一份元数据，本地却攒了一份更完整的。日志字段裁减、聚合统计优先、明确保留期与销毁动作，是同一套最小化的落地。

另一条约束是”信任边界要写进设计文档”：端点运营者是谁、日志落在哪个运维域、是否有第三方 CDN 参与，决定了用户实际把隐私交给了谁；这些问题不是 TLS 版本能回答的。

17.8.4 常见误读

误读一：加密 DNS 服务天然不记录。记录与否是策略；协议只保证传输不被路径观察者读到。

误读二：只要关掉访问日志就符合建议。还要处理保留期、字段最小化、填充、ECS、以及 DoH 侧的 HTTP 指纹与第三方参与方。

误读三：这份文档只约束公共大厂。自建一台给家里或机房用的解析器，同样会成为该范围内所有查询的汇聚点；范围小不等于责任小。

17.9 客户端子网：[RFC 7871]

17.9.1 它要解决什么

递归解析器代很多客户端查询时，权威服务器（尤其 CDN）看到的是**解析器的 IP**，于是按解析器位置选节点，客户端可能被指到很远的边缘。[RFC 7871] 定义 EDNS(0) 的 Client Subnet 选项：解析器在出向查询里附带客户端源地址的一个**截断前缀**，让权威侧按客户端大致位置做调度，同时用应答里的 scope 前缀长度约束缓存复用范围。

17.9.2 关键条文

- 选项携带地址族、源前缀长度（source prefix-length）与 scope 前缀长度（scope prefix-length）；请求里放的是截断后的前缀，不是完整地址。
- 应答里的 scope 告知”这个答案适用于多宽的前缀”，解析器必须按它构造缓存键；scope 越窄，缓存命中率越低。
- 权威服务器未表明支持时不应发送；解析器对来自不可信客户端的 ECS 选项不应原样透传，也不应把前缀加长到超出自己策略允许的范围。
- 规范给出的是可互操作的机制与部署建议，不是”所有解析器必须开启”的要求（类别与状态见 RFC Editor 索引）。

17.9.3 对实现的约束

ECS 与 QMIN 的目标**直接冲突**：QMIN 在减少对上级暴露的元数据，ECS 却在同一个出向报文里主动加入一个按客户端划分的标识。每多一个权威服务器收到该前缀，就多一处能长期按前缀聚合”某网段在查什么”的日志；路径上能读到这一跳的观察者同样拿得到。IPv4 常见的 /24 或 IPv6 常见的 /56 看起来”不精确”，但在小网段、家宽前缀、固定 ULA 环境里足以构成稳定准标识。

运维约束有三条：能不开就不开，把启用范围限制在确实需要地理位置调度的权威服务器白名单内；按 scope 严格分缓存，避免把带 ECS 的应答错误复用到别的客户端；明确记录哪些上游收到了 ECS、保留多久。第三篇把 ECS 列为隐私代价项，条文依据在这里。

17.9.4　常见误读

误读一：只给前缀，所以不算个人信息。稳定的前缀加时间加查询内容足以做关联，“截断”不等于”匿名”。

误读二：不开 ECS，CDN 就完全不工作了。调度还有 anycast、解析器自身位置、以及应用侧的其他信号；ECS 是提高精度的选项，不是解析的前提。

误读三：ECS 是 [RFC 6891] 的一部分所以必须支持。它是 EDNS(0) 的一个可选选项，支持与否、转发与否都是策略决定。

17.10　整体隐私考量：[RFC 9076]

17.10.1　它要解决什么

前面每份 RFC 各补一段：QMIN 补元数据最小化、DoT / DoH / DoQ 补一跳加密、ODoH 补参与方分离、ECS 补调度精度、[RFC 8932] 补运营策略。但”DNS 隐私”作为一个整体问题，需要一份把**所有位置、所有参与方、所有非加密维度**列全的威胁模型。[RFC 9076] 承担的就是这个横向梳理：它不定义新协议，而是说明加密之后还剩哪些泄露面。

17.10.2　关键条文

- 泄露点遍布全链路：存根到解析器、解析器到各级权威、缓存与转发链、区传送、被动 DNS 采集、日志系统、以及解析器运营者与权威运营者各自的数据留存。
- 可见的不止 QNAME：QTYPE、EDNS 选项、源 / 目的 IP、报文长度、时序、并发模式、失败与重试模式，都能被用于关联与画像；DNSSEC 增加的是可验证的数据，不是保密性。
- 加密一跳会把信任**转移**而不是消除：从路径观察者转移到端点运营者；若端点之后的迭代仍是明文，泄露面只是挪了个位置。
- 参与方数量本身是风险：多个层级的运营者若共享或交换日志，单点的最小化措施会被聚合抵消；共谋与全局被动观察者属于更强的对手，很多机制并不承诺抵御它们。
- 因此文档强调数据最小化、保留期、日志访问控制和可审计的策略，而不是把”是否用了 TLS”当作隐私结论。

17.10.3　对实现的约束

对自建部署，最实用的一条是**先画数据流图再谈加密**：列出每个实例各自能看到什么、记什么、留多久、谁能读，然后才决定哪一跳需要 DoT / DoQ、哪一跳可以靠本地根区镜像根本不出网。否则很容易出现”客户端到网关加密了、网关到上游明文、上游又开了全量 dnstap”的组合，而配置审计表上每一格都写着”已加密”。

另一条是明确写出不承诺：QMIN 不承诺抗全路径嗅探，ODoH 不承诺抗流量分析，填充不承诺抗长度画像，DNSSEC 不承诺保密。把承诺边界写进设计文档，比事后解释”为什么还是会泄露”省事得多。

17.10.4　常见误读

误读一：这份文档是某一份协议的实施细则。它是横向的隐私考量梳理，用来检查其它 RFC 的覆盖面，不提供开关或参数。

误读二：只要全网都上加密传输，DNS 隐私问题就解决了。端点运营者、权威服务器、日志留存与流量分析仍在模型里。

误读三：隐私与可观测性二选一，只能牺牲一边。真实取舍是”最小化 + 明确保留期 + 聚合优先”，而不是关掉全部日志或全量留存两个极端。

17.11　两个收尾判断

17.11.1　denylist 不是证明

第三篇里”按 IP 封 443 不如 canary 域准”这条实测结论，值得在理论层说透。把已知 DoH 端点 IP 收进一张拒绝列表，得到的是一个**枚举型否定**：“匹配到列表里的地址时才拦截”。它既覆盖不全——新端点、共享 IP、CDN 与 anycast 之后的同一地址、以及和 Web 共端口的自建端点，都可以不在表里；也容易过度——同一个 IP 上跑着别的服务时，封禁会误伤。更关键的是，**“没有被列表命中”不是”查询确实经过了本地解析器”的证据**：客户端可以直连一个表里没有的端点，网络侧只看到 443，什么也证明不了。

能称为证据的是**肯定性的观测**：本地解析器自己的查询日志或 dnstap 流里出现了这条查询；canary 域按预期返回了规定的 NXDOMAIN / NODATA；或者受控客户端的连接目标与解析器地址一致。名字层的本地服务区判定

（`_dns.resolver.arpa`. 返回 NODATA、浏览器 canary 回 NXDOMAIN）之所以比封 IP 更可靠，是因为它作用于**客户端的决策逻辑**，而不是试图穷举客户端可能连到哪里。这一区分在加密流量里尤其重要：报文不可读之后，唯一可靠的观测点就只剩自己这一侧的日志。

一句话：**denylist 是一种拦截手段，不是一种证明方法**。把它写进合规或验收结论里，等于用”没抓到”代替”没发生”。

17.11.2 抗聚合≠抗全路径嗅探

QMIN 的收益要放在正确的对手模型下衡量。它把完整 QNAME 从根、TLD 等”本不需要知道”的层级上拿掉，使这些运营者难以长期按完整域名聚合”谁在查什么”；这是**抗元数据聚合**，对手是分散在多个层级、各自只看一段、但会把日志留很久的运营者。它不改变任何一跳的明文属性，也不改变最终权威服务器必须看到完整 QNAME 这一事实。

而一个在解析器出向链路上的被动观察者，看到的是**全部**迭代查询：每一次最小化后的中间名、最后一次完整 QNAME、QTYPE、时序、长度、目的地址。本地根区镜像能把根这一跳从出向链路里消掉，收益落在 TLD 与二级权威层；但链路其余部分仍在该观察者的视野里。这就是”抗聚合”与”抗全路径嗅探”的差别：

维度	抗元数据聚合（QMIN 等）	抗全路径嗅探
对手	多个只看见局部的权威运营者	单一链路上看见全部报文的观察者
手段	减少每跳携带的标签、限制留存	该跳的传输加密，或让该跳根本不出网
覆盖	解析器到各级权威（部分层级）	某一跳的全部字节与元数据
剩余可见	最终权威仍见完整 QNAME、解析器见全部	加密后仍见长度、时序、目的 IP
与 DNSSEC 关系	无关	无关（DNSSEC 不提供保密性）

因此全书反复用两句话收口：**QMIN 不是加密**；**加密传输只覆盖它被部署的那一跳**。把”上级少拿了一份元数据”读成”链路上没人能看见”，会把威胁模型里的一个对手换成另一个，却以为已经赢了。同理，ODoH 的两跳分离对付的是”端点同时看到 IP 与查询”，不是”有人能看到全部流量”；自建可信 target 时收益约等于零，也是同一个道理——对手模型没变，加一跳不会凭空产生保护。

17.12 本章要点

- [RFC 7816] 用固定 NS 类型问中间层；[RFC 9156] 废弃它并**改回保留原始 QTYPE**，因为 NS 被权威服务器特殊对待、会让应答语义失真。relaxed 遇异常回退完整 QNAME，strict 依 [RFC 8020] 不回退，因此会撞上”父名 NXDOMAIN、子名却存在”的违规区，且错误结论被 [RFC 2308] 负缓存逐层记住。
- DoT（[RFC 7858]，TCP 853）、DoH（[RFC 8484]，HTTPS 443）、DoQ（[RFC 9250]，专用 QUIC）都只保护**存根到解析器这一跳**；[RFC 8310] 的 Strict / Opportunistic 两档决定认证或握手失败时能否静默退回明文，严格档必须 fail-closed。
- [RFC 9230] 的 ODoH 靠 relay 与 target 分离实现”见 IP 不见查询 / 见查询不见 IP”；**自建可信 target 时收益约等于零**——你本来就信任它，而独立 relay 又成为新的 IP 观察者，两跳自建则日志一关联即还原。
- [RFC 9462] 的 `_dns.resolver.arpa`. 发现机制给客户端一条自动绕过本地解析器的路径，本书以本地 NODATA 与浏览器 canary 域压制它；[RFC 7871] 的 ECS 与 QMIN 目标直接冲突，[RFC 8932] 与 [RFC 9076] 则把日志、保留期、流量分析与运营者责任补进威胁模型。
- **denylist 不是证明**：按 IP 封禁是枚举型否定，“没命中”不等于”查询走了本地解析器”，证据只能来自自己这一侧的肯定性观测。**抗聚合≠抗全路径嗅探**：QMIN 抗的是多个权威运营者的元数据聚合，不加密任何一跳，链路上一个被动观察者仍看到全部迭代查询。

17.13 参考资料

- RFC Editor, [RFC 7816] *DNS Query Name Minimisation to Improve Privacy*: https://www.rfc-editor.org/rfc/rfc7816.txt
- RFC Editor, [RFC 9156] *DNS Query Name Minimisation to Improve Privacy*: https://www.rfc-editor.org/rfc/rfc9156.txt
- RFC Editor, [RFC 7858] *Specification for DNS over Transport Layer Security (TLS)*: https://www.rfc-editor.org/rfc/rfc7858.txt
- RFC Editor, [RFC 8484] *DNS Queries over HTTPS (DoH)*: https://www.rfc-editor.org/rfc/rfc8484.txt

- RFC Editor, [RFC 9250] *DNS over Dedicated QUIC Connections*: https://www.rfc-editor.org/rfc/rfc9250.txt
- RFC Editor, [RFC 8310] *Usage Profiles for DNS over TLS and DNS over DTLS*: https://www.rfc-editor.org/rfc/rfc8310.txt
- RFC Editor, [RFC 9230] *Oblivious DNS over HTTPS* (Independent Submission / Experimental, 状态见 Info 页): https://www.rfc-editor.org/rfc/rfc9230.txt
- RFC Editor, [RFC 9180] *Hybrid Public Key Encryption*: https://www.rfc-editor.org/rfc/rfc9180.txt
- RFC Editor, [RFC 9458] *Oblivious HTTP* (Proposed Standard, 基于 ODoH 泛化): https://www.rfc-editor.org/rfc/rfc9458.txt
- RFC Editor, [RFC 9462] *Discovery of Designated Resolvers*: https:// www.rfc-editor.org/ rfc/rfc9462.txt

其余 [RFC 8932]、[RFC 7871]、[RFC 9076]，以及章内引用的 [RFC 8020]、[RFC 2308]、[RFC 6761]、[RFC 5737] 等原文，均在 RFC Editor 按编号索引；完整链接由第四篇的统一 RFC 索引章收录。

第十八章

IPv6 / RA / RDNSS RFC 族与实现差异

IPv6 主机上网之前要回答四个问题，而 IPv4 时代这四个问题大多由一个 DHCP 报文包办：

1. 我该用哪个地址（SLAAC 与 PIO）；
2. 这个地址能用多久（Preferred / Valid Lifetime）；
3. 出网的默认路由器是谁、还有哪些更具体的前缀要走谁（RA、默认路由器优先级、RIO）；
4. 域名该问谁（RDNSS / DNSSL）。

它们的答案全部装在 ICMPv6 路由器广告（Router Advertisement，RA）及其选项里。DNS 部分之所以值得单独写一章，是因为本书的主角是递归解析器：**一个 Unbound 实例再怎么加固，只要客户端的存根解析器（stub resolver）把查询发给了别人，加固就等于没做**。RA 是决定「客户端把查询发给谁」的那一层协议，而这一层的语义由一组彼此咬合的 RFC 规定。本章按「解决什么→关键条文→对实现的约束→常见误读」逐个拆开。

18.1 底盘：[RFC 4291] 地址架构与 [RFC 8200] 1280 字节最小 MTU

18.1.1 它要解决什么

[RFC 4291] 定义 IPv6 的地址模型：128 位地址、单播 / 组播 / 任播三类、地址文本表示、以及必需的地址（每个接口的链路本地地址、环回、未指定地址、被请求节点组播地址等）。它把「地址」和「接口」解耦：一个接口可以同时挂链路本地、ULA、一个或多个全局地址，每个地址还带 preferred / deprecated 状态。后面的源地址选择、临时地址、ULA 都建立在这个多地址模型之上。

[RFC 8200] 是 IPv6 基本首部规范。它把分片责任从路由器收回到源端：中间路由器不重组、不分片，超出路径 MTU 的报文由源端通过 Packet Too Big 学习后重发。

18.1.2 关键条文

- [RFC 4291] 规定地址前缀长度写法、链路本地前缀 `fe80::/10`、组播前缀 `ff00::/8`，以及接口标识符与前缀拼成 128 位地址的规则；链路本地地址只在同一链路上有意义，因此带链路本地地址时必须带 zone / scope 信息。
- [RFC 8200] 第 5 节规定：**每条链路的 MTU 必须至少为 1280 字节**，这就是 IPv6 最小链路 MTU。链路层不足 1280 时必须在 IPv6 之下自己做适配。源端可以只发不超过 1280 字节的报文并省略 PMTUD 实现，但那样就不能利用更大 MTU。

18.1.3 对实现的约束

1280 不是「建议值」，是所有上层协议必须能落脚的地板。DNS 走 UDP 时，IPv6 报文 = 40 字节 IPv6 首部 + 8 字节 UDP 首部 = 48 字节开销，`1280 - 48 = 1232`，这正是第三篇反复出现的 `edns-buffer-size: 1232` 的条文根因。把值调到 1472 会在 1280 MTU 的链路上分片；而 [RFC 8200] 的分片只有源端能做，被中间设备丢弃的分片不会有任何路由器替你重组，表现就是等超时再退到 TCP。

链路本地地址与 zone 的约束同样会漏到实现里：RA 里若广告链路本地 RDNSS 地址，[RFC 8106] 要求把它连同源接口的 zone index 一起登记进解析器仓储，否则同一个 `fe80::1` 在多接口主机上无法区分该从哪个接口发出。

18.1.4　常见误读

- 「**IPv6 不分片，所以不会丢包**」：不分片的是中间路由器，源端仍能发分片首部；分片丢失照样发生，且更难排查。
- 「**1280 是推荐 MTU**」：是下限，不是性能指标；链路 MTU 更大不代表 UDP 载荷能自动放大。
- 「**文档地址 `2001:db8::/32` 可以随便配到接口上**」：它是 [RFC 3849] 保留的文档前缀，只用于写作与教学，不是可路由地址。

18.2　[RFC 4861] NDP 与 RA：报文为什么必须是 hop limit 255

18.2.1　它要解决什么

IPv6 取消了 ARP，把「地址解析、邻居不可达检测、路由器发现、重定向、前缀与参数通告」合并为邻居发现协议（Neighbor Discovery Protocol，NDP），承载在 ICMPv6 上。RA 是其中的路由器广告消息：路由器周期性组播到 `ff02::1`（所有节点），或应答主机发往 `ff02::2`（所有路由器）的路由器请求 RS。

18.2.2　关键条文

- 所有 NDP 消息的 IP Hop Limit **必须为 255**；接收方必须校验该值为 255，否则静默丢弃。[RFC 4861] 给的理由很直白：255 意味着「该报文不可能被路由器转发过」，因此 NDP 天然被限制在本链路内，不需要额外信任机制就能挡住来自链路外的伪造 RA / NA。这是一条**协议级安全边界**，不是可协商参数。
- RA 的接收有效性检查还包括：ICMP 校验和有效、Code 为 0、ICMP 长度不小于规定值、所有已含选项长度非零。
- RA 首部带 `Cur Hop Limit`、`M` / `O` 位、`Router Lifetime`、`Reachable Time`、`Retrans Timer`；`Router Lifetime` 非零才表示该路由器可作为默认路由器，为零表示「撤销我作为默认路由器」。
- 源链路层地址选项（Source Link-Layer Address Option，SLLAO）：路由器在 RA 中**可以**携带发送接口的链路层地址（为便于入向负载均衡等情况也允许省略）；NS 发往被请求节点组播地址时则**必须**携带，好让目标有地方回 NA。
- 前缀信息选项（Prefix Information Option，PIO）带 `L`（on-link）、`A`（autonomous，即允许 SLAAC）两个标志位、Valid Lifetime 与 Preferred Lifetime。
- 主机维护默认路由器列表、前缀列表、邻居缓存与目的地缓存；未识别选项必须忽略，保证向后兼容——这正是 [RFC 4191]、[RFC 8106] 能以后续 RFC 追加选项而不改动 [RFC 4861] 的原因。

18.2.3　对实现的约束

任何转发、隧道、桥接、Netfilter 规则都不许改写 NDP 报文的 hoplimit。第 19 章实测过的 iOS 硬性要求只是这条通用条文的一个可观测后果：抓包里若看到 254 或 64，客户端会整包丢弃，且不会给出「你的 hoplimit 错了」这种提示，只会表现为「拿不到 IPv6 默认路由 / 拿不到 DNS」。发送侧也别自作聪明把 hoplimit 交给应用填：由内核构造 RA 时置 255 最稳。

`Router Lifetime=0` 是「撤销默认路由器」，不是「关闭 RA」。一台路由器仍可以发带 PIO / RDNSS 选项、但 `Router Lifetime=0` 的 RA，用来只广告 DNS 或撤销旧前缀而不参与默认路由选举。实现若把两者混为一谈，会造成「改了 DNS 却把默认路由弄没了」。

18.2.4　常见误读

- 「**hoplimit 255 只是防环**」：它首先是链路边界证明，用来在不引入 SEND / RA-Guard 的前提下把 NDP 攻击面限制在本链路。
- 「**RA 只能组播**」：RS 触发的单播 RA 是 [RFC 4861] 的正常路径，关系到「刚连上 Wi-Fi 能不能立刻拿到配置」。
- 「**SLLAO 是必填项**」：RA 里是 MAY；真正 MUST 的是发往被请求节点组播地址的 NS。把 MAY 当 MUST，会在多接口 / 负载均衡拓扑下误判合规。
- 「**没收到 RA 就退回 DHCPv4 思维**」：`M` / `O` 位只表示「可用有状态 DHCPv6」，IPv6 默认路由仍然只能来自 RA，DHCPv6 本身不下发默认网关——这是它与 DHCPv4 最根本的差别。

18.3 [RFC 4862] SLAAC: 为什么 PIO lifetime=0 缩不掉一个长寿命地址

18.3.1 它要解决什么

SLAAC 规定主机如何把 RA 里的前缀与本接口标识符拼成地址、如何做重复地址检测（DAD）、以及地址从 preferred 到 deprecated 再到 invalid 的生命周期。它是无状态的：路由器不记录谁用了哪个地址，主机自己算、自己检测冲突。

18.3.2 关键条文

[RFC 4862] 第 5.5.3 节的 e) 步是本章最该逐字读的一段。当 RA 里的前缀与主机已有的某个 SLAAC 地址前缀相同时：

- **Preferred Lifetime 总是**重置为收到的值（这样管理员可以用短 preferred lifetime 主动把地址标记为 deprecated）；
- **Valid Lifetime 的处理分三种情况**，设已存在地址的剩余寿命为 `RemainingLifetime`：
  1. 收到的 Valid Lifetime 大于 2 小时，或大于 `RemainingLifetime`，就把有效寿命设为收到值；
  2. 若 `RemainingLifetime` ⩽ 2 小时，则**忽略**该 PIO 对 valid lifetime 的修改，除非 RA 经过了认证（如 [RFC 3971] SEND）；
  3. 否则（即想缩短、且剩余寿命还长），把有效寿命重置为 **2 小时**，而不是收到的那个小值。

条文明说这是为了防一种拒绝服务攻击：一份伪造的、带极短 Valid Lifetime 的 RA 不该让全网主机的地址瞬间过期。合法路由器周期性发的 RA 会在短寿命真正生效之前把它「抵消」掉。注意 preferred lifetime 不受此保护，因为「把地址标记 deprecated」造成的危害相对小，且管理员确实需要这个能力。

18.3.3 对实现的约束

这直接解释了第 19 章那个最反直觉的观测：**运营商换前缀之后，旧前缀的地址不会因为你补发一份 `lifetime=0` 的 PIO 就立刻消失**。按第 3 条，一个原本还有几天寿命的地址最多被压到 2 小时；按第 2 条，剩余寿命已经很短的甚至直接忽略。也就是说「PIO lifetime=0 = 立即删除地址」在 [RFC 4862] 里是不存在的语义。

于是 RFC 族里出现了一个重要分工，后文会反复用到：

对象	`lifetime=0` 的含义	依据
PIO 中的前缀	把已有 SLAAC 地址的 valid lifetime 压到最多 2 小时（或忽略），不是立即删除	[RFC 4862] §5.5.3 e)
RDNSS 选项中的解析器地址	从 DNS Server List 与 Resolver Repository 中**删除该条目**	[RFC 8106] §6.2 步骤 (b)
RA 首部 `Router Lifetime=0`	撤销该路由器作为默认路由器	[RFC 4861]
RIO 中的路由	撤销该更具体路由	[RFC 4191]

不同选项的「0」语义不同，是因为它们保护的对象不同：地址是长期状态，撤销必须给足在途连接收敛时间；DNS 服务器列表是配置指针，换掉它不影响已有 TCP 连接，可以即时生效。

18.3.4 常见误读

- 「**PIO lifetime=0 就是让地址马上失效**」：条文给出的下限是 2 小时，且这条下限是防 DoS 设计，不是实现偷懒。
- 「**前缀轮换后只要发撤销 RA 就完事**」：旧地址仍会存活，配合 [RFC 6724] 的临时地址优先，客户端可能继续拿旧前缀当源地址，上游不路由就全断。这正是第 19 章要设三处 sysctl 的机理。
- 「**deprecated 地址会被立刻删掉**」：deprecated 只是不再用于**新**连接，已有通信 SHOULD 继续用它；要等 valid lifetime 到期才是 invalid。
- 「**DAD 通过就万事大吉**」：DAD 只保证本链路上没有冲突，不保证前缀可路由、也不保证上游仍认这个前缀。

18.4 [RFC 6106] → [RFC 8106]: RDNSS / DNSSL 与 lifetime=0

18.4.1 它要解决什么

RA 原本不带 DNS 信息，IPv6 主机只能靠有状态 DHCPv6 拿递归解析器地址，而不少客户端（尤其移动系统）根本不实现 DHCPv6 客户端。[RFC 6106] 定义了两个 RA 选项：RDNSS（Recursive DNS Server，携带一到多个递归解析器 IPv6 地址 + Lifetime）与 DNSSL（DNS Search List，携带搜索域 + Lifetime），让纯 SLAAC 网络也能配好 DNS。[RFC 8106] 废止并取代 [RFC 6106]，沿用选项格式与语义，主要做澄清与默认值调整。

18.4.2 关键条文

- RDNSS / DNSSL 选项各有自己的 Lifetime 字段，**与 RA 首部的 Router Lifetime 无关**；路由器不再是默认路由器，不代表它广告的 DNS 信息要一起过期。
- Lifetime 以秒计，是「自收到该 RA 起的有效时长」，主机为每条 DNS 记录维护 Expiration-time；到期后该条目不应再使用。
- [RFC 8106] §6.1 / §6.2 规定主机维护两张结构：DNS Server List（按 RA 顺序保存学到的地址与 Expiration-time）与 Resolver Repository（真正供存根解析器使用的有序列表）。§5.1 的字段定义把零值写为「该地址 MUST 不再被使用」；§6.2 步骤 (b) 把它落到实现上：**若某地址已在 DNS Server List 中、且该 RDNSS 选项的 Lifetime 字段为 0，则从两张表里删除该条目**，用于 DNS 服务终止或重编号场景。步骤 (c) 则规定同接口收到相同地址的非零 Lifetime 时只刷新 Expiration-time；§5.3.1 要求主机 SHOULD 保留来自多个来源（RA、DHCP）的 DNS 信息，保留多少属本地策略。
- [RFC 8106] §5.1 规定 RDNSS 的 Lifetime 是「自收到该报文起的秒数」，SHOULD 默认**至少**为 `3 × MaxRtrAdvInterval`；全 1 值（`0xffffffff`）表示无限，零值表示该地址 MUST 不再被使用。DNSSL 选项的 Lifetime 字段语义相同。
- [RFC 6106] 曾把 Lifetime 约束成 `MaxRtrAdvInterval <= Lifetime <= 2*MaxRtrAdvInterval` 的**双侧区间**。[RFC 8106] 附录 A 说明这次修订针对的是：高丢包链路上按旧上界取值会让选项频繁过期，于是改成以 `3 × MaxRtrAdvInterval` 为下界、不再设上界，同一节还列出了「至少能扛住连续三次含 DNS 选项的 RA 丢失」的理由。其余改动：删除「过期前主动发 RS 刷新」的建议以免增加组播流量；明确 RDNSS 地址可以是链路本地地址（须带 zone index）；取消「最多维护 3 个 RDNSS 地址 / 3 个搜索域」的推荐，交给实现者的本地策略；不再把 SEND 写成 SHOULD / RECOMMENDED 级别的安全机制。

18.4.3 对实现的约束

Lifetime=0 是「删除」，不是「设为无限」、也不是「优先级最低」。但它只在「该地址已存在于 DNS Server List」时生效：对一个从未学过的地址发 `lifetime=0`，按步骤 (b) 的条件不成立，实现会走到步骤 (d) 的注册分支。因此网关若要压掉同二层域里另一台路由器广告的 DNS 地址，必须：

- 用**完全相同**的地址值构造撤销选项（前缀、压缩写法可以不同，数值必须一致）；
- **持续、周期性重复发送**，因为客户端重新联网或对方重新发 RA 后该地址会被重新注册；
- 自己的地址用正常 Lifetime 随每次 RA 续期。

还有一层实现自由度：主机把多个路由器广告的 RDNSS 合并成一张列表，而不是「后到覆盖先到」，且 [RFC 8106] 不再限制条目数。但「是否接受 A 路由器撤销 B 路由器广告的地址」在条文里并非强约束，各家网络栈行为不完全一致——这是第 19 章把「阻断 WAN 侧 RA」列为退路的原因。

DNSSL 与 RDNSS 的 lifetime 语义一致，但搜索域会改变**绝对域名被怎样追加后缀**，对查询放大与 QNAME 最小化的影响属于第三篇第 13 章的范围，此处不展开。

18.4.4 常见误读

- 「**[RFC 6106] 和 [RFC 8106] 是两套可任选的机制**」：后者 obsoletes 前者，选项类型码沿用，引用时应以 [RFC 8106] 为准；旧资料里「最多 3 个 RDNSS」的说法来自 [RFC 6106]。
- 「**lifetime=0 表示永不过期**」：与 TTL / DHCP lease 的直觉相反，它是显式撤销；无限寿命的编码是全 1 值 `0xffffffff`，两者别弄反。
- 「**发一次撤销就够了**」：条目会被对方后续 RA 重新注册，撤销必须持续。
- 「**RDNSS 只能广告全局地址**」：[RFC 8106] 明确允许链路本地地址；是否选 ULA 是稳定性考量（GUA 前缀会轮换，ULA 不变），不是条文限制。

18.5 [RFC 4191]: RIO 与默认路由器优先级

18.5.1 它要解决什么

[RFC 4861] 只允许主机从多条默认路由器里挑一条，没有「哪条更优先」和「某个具体前缀要走谁」的表达能力。多宿主、双上行、以及「网关后面还有一段内网」的场景都需要更细的路由信息。[RFC 4191] 在 RA 首部保留位里加入 2 位 `Prf`（Default Router Preference），并新增类型 24 的路由信息选项（Route Information Option，RIO）。

18.5.2 关键条文

- `Prf` 是 2 位有符号整数（Low / Medium / High）；`Router Lifetime=0` 时发送方 MUST 置 (00)、接收方 MUST 忽略它；保留值 (10) 按 (00) 处理。
- RIO 带 Prefix Length、独立的 `Prf`（Route Preference）、Route Lifetime 与可变长 Prefix；`Prefix Length > 64` 时 Length 必须为 3；收到 (10) 保留值时整个 RIO MUST 被忽略。
- 同一 RA 里不得出现两个前缀相同的 RIO。RIO 只在多个路由器广告**相同前缀**时参与比较，它不是默认路由的替代品。
- [RFC 4191] 把主机分为 Type A / B / C：A 完全忽略优先级与 RIO，B 用默认路由器优先级但忽略 RIO，C 两者都实现。也就是说 RIO 支持度是**实现可选的能力**，条文自己就预留了不支持的合规主机。

18.5.3 对实现的约束

Linux 内核把 RIO 的处理交给 `accept_ra_rt_info_min_plen` 等 sysctl 控制;Android 侧的 `accept_ra_rt_info_min_plen=48` 要求意味着**前缀长度小于 48 的 RIO 会被静默丢弃**。注意它是「最小前缀长度」的下限：/32、/40 这类更粗的聚合不满足，/48 或更细才进路由表。这不是条文规定，而是某个具体网络栈的本地策略，两者别混为一谈。

由此产生一个粒度分工：RIO 用于**路由聚合**（/48），PIO 里的 /64 用于**地址配置与前缀变更检测**。聚合前缀在运营商轮换 LAN /64 时通常不变，因此拿 RIO 的 /48 去判断「前缀是否变化」会把两次故障的根因藏成同一个变量——这个机理在第 19 章展开，本章只标出条文依据：PIO 与 RIO 是不同选项、不同 Lifetime、不同用途，[RFC 4191] 明确 RIO 不能取代 on-link / SLAAC 前缀判定。

18.5.4 常见误读

- 「**设了 High 优先级就一定能抢到默认路由**」：`Prf` 只在路由器都有效（Router Lifetime 非零）时参与比较；且 Type A 主机完全忽略它。
- 「**RIO 可以当 PIO 用**」：RIO 只装路由，不给主机配地址；反过来 PIO 也不能替代 RIO 表达「去这个前缀走我」。
- 「**Android 的 48 是 RFC 规定的最小值**」：是具体实现策略，条文里的 Prefix Length 范围是 0–128。

18.6 [RFC 4941] 临时地址与它的后续修订

18.6.1 它要解决什么

SLAAC 默认把 64 位接口标识符（传统上由链路层地址派生，或按 [RFC 4291] 相关规则生成）长期固定，配合全局前缀就能跨网络、跨时间关联同一台设备。[RFC 4941] 定义临时地址扩展：在稳定的「公共地址」之外，额外生成随机接口标识符、寿命有限、周期性重新生成的临时地址，用于对外发起的新连接，降低被动流量分析的关联能力。

18.6.2 关键条文

- 临时地址由随机接口标识符 + 当前前缀生成，生成后仍要做 DAD。
- 它带 `TEMP_VALID_LIFETIME` 与 `TEMP_PREFERRED_LIFETIME`；[RFC 4941] 给出的 `TEMP_VALID_LIFETIME` 默认值是**一周**，并明确该值未必适合所有场景、用户应能调整。
- 实现必须提供让应用反转偏好、改用公共地址的机制（例如 [RFC 5014] 一类的 API 扩展），且该机制只影响调用它的应用。
- 临时地址与「隐私」的边界要说清：它降低的是**按地址做长期关联**的能力，不是加密、也不是匿名；上层协议里的标识符、DNS 查询名、时序仍可被关联。
- 版本说明：[RFC 8981] 废止并取代 [RFC 4941]（同系列、标题相近的 *Temporary Address Extensions for Stateless Address Autoconfiguration in IPv6*）。本书提到「临时地址」这一机制时沿用 [RFC 4941] 这个名字以和第 19 章的实测结论对齐；具体默认参数、生成算法与弃用条件以 [RFC 8981] 及官方手册对应章节为准。

18.6.3　对实现的约束

临时地址的寿命是**独立于前缀寿命**的另一只时钟。前缀轮换后，旧前缀上已生成的临时地址并不会因为前缀撤销而立刻消失（见 18.3），它的 `TEMP_VALID_LIFETIME` 还在走；Linux 侧 `temp_valid_lft` 默认 48 小时这类取值是实现默认值，不是 [RFC 4941] 的一周，两者不要互相套用。

对服务器侧还有一个常被忽略的约束：临时地址不适合做**被访问**的端点。作为 RDNSS 里广告的递归解析器地址，应当用不随前缀轮换、也不随隐私时钟变化的 ULA；让客户端的解析器地址自己每两天变一次，等于把第 9 章的访问控制列表变成一张永远过期的表。

18.6.4　常见误读

- 「**开了临时地址就不用管 DNS 隐私**」：它只管源地址，与 DNSSEC 的数据完整性、DoT / DoH / DoQ 的传输加密是三件事。
- 「**临时地址会让服务器连不上我**」：它用于主动出连接；监听、被访问应绑定稳定地址。
- 「**旧前缀的临时地址会随 RA 撤销一起消失**」：见 18.3，PIO `lifetime=0` 压不掉长寿命地址。

18.7　[RFC 6724] 源地址选择：规则编号的迁移与两条关键偏好

18.7.1　它要解决什么

一台主机同时有链路本地、ULA、一个或多个 GUA、临时地址、deprecated 地址，甚至还有 IPv4 地址。给定目的地址，选哪个做源？[RFC 3484] 定义了默认源地址选择与默认目的地址选择算法及策略表（policy table）；[RFC 6724] 废止并取代它，规则排序与内容有调整。这个选择发生在应用 `connect()` / 发包之前，普通用户几乎看不到，却直接决定「流量从哪个前缀出去」。

18.7.2　关键条文

先说编号迁移，因为运维资料里常把两代规则混着引用：

语义	[RFC 3484]（旧）	[RFC 6724]（现行）
避开 deprecated 地址	Rule 3: Avoid deprecated addresses	**Rule 3: Avoid deprecated addresses**（编号未变）
优先出接口	Rule 5: Prefer outgoing interface	**Rule 5: Prefer outgoing interface**
优先匹配标签	Rule 6	**Rule 6: Prefer matching label**（编号未变）
临时地址优先	旧版 Rule 7 是 *Prefer public addresses*（**相反**）	**Rule 7: Prefer temporary addresses**

也就是说「规则 5 非 deprecated」是旧资料里常见的说法混淆：[RFC 3484] 里避开 deprecated 是 Rule 3、优先出接口是 Rule 5；[RFC 6724] 保留了这两个编号，真正反转的是 Rule 7——从「优先公共地址」改成「优先临时地址」。此外 [RFC 6724] 新增了 Rule 5.5（优先由所选 next-hop 广告的前缀所分配的地址），并明确该规则**只适用于会追踪「哪个 next-hop 广告了哪个前缀」的实现**，不要求所有实现都记这份信息。

两条与本书主线最相关的偏好：

- **避开 deprecated**（Rule 3）：preferred 与 deprecated 二选一时取 preferred。这是「旧前缀地址被标记 deprecated 后不再用于新连接」的条文依据。
- **临时地址优先**（Rule 7）：临时地址与公共地址二选一时取临时地址；实现 MUST 提供让应用反转该偏好的机制。

策略表同样关键：默认表里 `::/0` 优先级 40、`fc00::/7`（即 ULA 段）优先级 3、`::ffff:0:0/96`（IPv4 映射）35。低优先级 + 标签不匹配，会让「有 IPv6 全局地址时优先不用 ULA 做源」成为默认行为——这正是「ULA 做 DNS 地址、GUA 做出网地址」能共存的原因：RDNSS 里是 ULA，但客户端发起查询时选的源地址未必是它，防火墙与访问控制要按这个现实写。

18.7.3 对实现的约束

把 18.3、18.6、18.7 三段连起来就是那条著名的故障链：运营商换 /64 →旧 PIO 的长寿命地址只能被压到 2 小时→旧前缀上的临时地址仍在 `TEMP_VALID_LIFETIME` 内且仍是 preferred → Rule 7 让它优先于公共地址→ Rule 3 不把它排除（它还没 deprecated）→客户端拿旧前缀当源→上游不路由→全断。任何一环的条文单看都合理，组合起来才成为故障。

实现还 MUST / SHOULD 提供策略表可配置能力（如 `gai.conf` 一类机制），且自动添加的行不得覆盖管理员显式配置的行。这条约束的含义是：策略表是本地策略，不是网络下发的，排查跨机器行为差异时先比对各自策略表。

18.7.4 常见误读

- 「**Rule 7 一直都是临时地址优先**」：[RFC 3484] 的 Rule 7 恰好相反；引用旧抓包资料或旧内核文档时要先看它依据哪一代 RFC。
- 「**Rule 5 是避开 deprecated**」：编号对应的是优先出接口；避开 deprecated 是 Rule 3。
- 「**源地址选择由路由器决定**」：算法在主机本地跑，路由器只能通过 RA / RIO / 策略表默认值间接影响。
- 「**ULA 优先级低说明 ULA 没用**」：低优先级针对的是选源；ULA 稳定不变，适合做被访问的解析器地址。

18.8 [RFC 4193] ULA：稳定的「内部编号」，不是可路由公网地址

18.8.1 它要解决什么

RFC 1918 私有地址在 IPv6 里需要一个对应物，但设计目标更严格：既要在站点内部稳定、不随运营商前缀轮换变化，又要尽量避免两个站点合并 / 建 VPN 时前缀撞车。[RFC 4193] 定义唯一本地地址（Unique Local Address，ULA）。

18.8.2 关键条文

- 前缀 `fc00::/7`；其中 L 位为 1 的 `fd00::/8` 用于本地分配，`fc00::/8` 的分配方式见官方手册。实际书里写的 `fd00:` …属于前者。
- 结构 = 前缀 + **40 位 Global ID** + 16 位 Subnet ID + 64 位接口标识符。Global ID 用伪随机算法生成，目的是让不同站点「碰巧选到同一个 /48」的概率极低，而不是靠中央机构分配。
- 条文明确 ULA **不预期在全局互联网上被路由**，也不应与全局地址做等价的路由通告；站点边界应过滤。它是「本地唯一」，不是「全球唯一」的同义词。
- [RFC 6724] 默认策略表给 `fc00::/7` 优先级 3，低于 `::/0` 的 40，见上一节。

18.8.3 对实现的约束

ULA 的价值在于**生命周期与运营商无关**：GUA /64 会被轮换，ULA 不会。因此把递归解析器的地址选在 ULA 段、通过 RDNSS 发布，是让「DNS 配置不随前缀轮换漂移」的条文级理由。它同时影响访问控制：第 9 章的 ACL 可以写成一个长期不变的 ULA 网段，不必每次前缀轮换都重写。

代价是实现必须自己管好两件事：一是边界过滤，别把 ULA 路由通告到公网侧；二是 Global ID 不要手工写成 `fd00::1/64` 这种人人相同的「伪 ULA」——那正是 40 位随机 Global ID 要防的冲突。教学示例里写 `fd00:53::1` 只为可读性，真实部署应生成随机 Global ID。

18.8.4 常见误读

- 「**ULA 就是 IPv6 版的 192.168**」：机制类似，但 40 位 Global ID 的防冲突目标、以及策略表里的低优先级，都是 RFC 1918 没有的。
- 「**ULA 可以 NAT66 出去上网**」：条文不预期全局路由；出网上网靠 GUA。
- 「**ULA 与 GUA 同时存在时一定用 ULA**」：默认策略表相反，见 Rule 6 与优先级 3。

18.9 PIO 的后续修订：编号核对与 [RFC 8978] / [RFC 9096]

18.9.1 它要解决什么

18.3 讲的那条 2 小时保护，在「前缀突然失效」的场景里会变成麻烦：CE 路由器崩溃重启、或运营商瞬间换发 DHCPv6-PD 前缀，主机却仍按旧 PIO 的 Valid Lifetime 认为旧前缀有效，于是继续拿旧地址当源，出现长时间的全断。这组文档处理的正是**重编号事件（renumbering event）**下 SLAAC 的收敛问题。

18.9.2　关键条文与编号说明

先做一次编号核对。这几个编号在二手资料里常被混用，按「拿不准的编号不编、以 RFC Editor 官方页面为准」的原则，这里给出可核验的对应关系：

文档	主题
[RFC 8971]	*Bidirectional Forwarding Detection (BFD) for Virtual eXtensible Local Area Network (VXLAN)*；属 BFD/VXLAN 范畴，**不涉及 PIO**
[RFC 8978]	*Reaction of IPv6 Stateless Address Autoconfiguration (SLAAC) to Flash-Renumbering Events*；分析主机在突发重编号下的行为与成因
[RFC 9096]	*Improving the Reaction of Customer Edge Routers to IPv6 Renumbering Events*（BCP）；给出 CE 路由器侧的规范性建议
[RFC 4862] §5.5.3	前述 2 小时下限，是所有这些修订要绕开/配合的原始条文

值得专门提醒的是：**[RFC 8971] 与 PIO 无关**，它的主题是 VXLAN 上的 BFD；讨论「PIO 与前缀轮换」时应查 [RFC 8978] 与 [RFC 9096]，不要按相近编号误引。下表把四者的位置一次列清，避免读者照着错误编号去找条文。年份、BCP 编号与「是否 obsoletes / updates 其他 RFC」以 RFC Editor 官方页面为准。

[RFC 9096] 与 PIO 直接相关的建议集中在第 3.4 / 3.5 节：

- PIO 的 Preferred / Valid Lifetime **不得超过**从上游 DHCPv6-PD 学到的对应剩余寿命;CE 路由器 SHOULD 用**较短**的 PIO 寿命，因为后续 RA 会不断续期，效果等价于上游长寿命，同时把「主机可能持有陈旧配置」的窗口缩短。
- 一旦前缀变为陈旧（stale），CE 路由器 MUST 继续以 PIO 广告它，且把 **Valid Lifetime 与 Preferred Lifetime 都置 0**、A / L 位保持不变；这类广告 MUST 持续至少先前用过的 Valid Lifetime 那么久，并用非请求 RA 发送。
- 收到上游 `valid-lifetime=0` 的 IA Prefix 时，也要把对应子前缀以双 0 的 PIO 广告出去。

18.9.3　对实现的约束

把这一节与 18.3 并排读，才看得出 RFC 族的完整设计意图：**撤销不是「发一次就删」，而是「在足够长的时间里持续声明它已作废」**。因为 [RFC 4862] §5.5.3 保证单次短寿命 RA 压不掉长寿命地址，唯一可靠的做法是从源头把 PIO 寿命设短（让地址本来就不会有几天寿命），并且把撤销声明持续发到覆盖旧寿命。这与 18.4 里 RDNSS `lifetime=0` 必须持续重复发送，是同一个工程结论在两种选项上的投影。

对 DNS 侧的约束是: 解析器地址若用 GUA, 前缀轮换时 RDNSS 里那个地址会随前缀一起失效, 客户端的 DNS Server List 里留着一条不可达条目；改用 ULA 就绕开了整条链。这是「ULA 做 DNS 地址」在 RFC 层面的第二个理由。

18.9.4　常见误读

- 「**PIO lifetime=0 能立刻清掉旧地址**」：[RFC 4862] 不允许；[RFC 9096] 的对策是缩短 PIO 寿命 + 持续撤销，而不是依赖单次双 0。
- 「**把 PIO 寿命设得跟上游一样长最稳妥**」：[RFC 9096] 的建议正相反，长寿命会放大陈旧配置窗口。
- 「**重编号只影响地址，不影响 DNS**」：RDNSS 里若放 GUA，客户端的解析器地址同样失效。

18.10　横向对比：同一组 RFC 在六种实现里的行为差异

这一节只比较**协议行为**：谁构造并发送 RA、谁消费 RA 里的 RDNSS、谁做 DNSSEC 验证、谁参与 DDR（Discovery of Designated Resolvers，[RFC 9462]）。

先立一条最容易被忽略的划分：**DNS 服务器软件与 RA 发送者是两个完全不同的角色**。RA 是 ICMPv6 消息，由内核网络栈或专门的 RA 守护进程（如 `radvd` 一类）构造；BIND、Unbound、CoreDNS 这类进程工作在 DNS 层，既不解析 RA，也不发 RA。把「我装了 Unbound」当成「客户端就会用它」，是这一章所有误读里代价最大的一个。

18.10.1 谁发 RA / 谁消费 RDNSS

实现	发 RA（构造 PIO / RIO / RDNSS 选项）	消费 RDNSS（把 RA 里的 DNS 地址装进存根解析器）	条文落点
BIND	否，纯 DNS 进程	否；其所在主机由操作系统的网络栈决定 resolver	[RFC 8106] 的主机侧行为不在 BIND 内
dnsmasq	**可发**：内置 RA 能力，可广告 RDNSS / DNSSL / PIO	否（作为本机进程它读系统 resolver 配置，不是 RA 消费者）	少数把「DNS 服务」与「RA 广告」放在同一进程的实现
systemd-resolved	否	**是**（配合 systemd-networkd / NetworkManager 接收 RA，维护 per-link DNS 服务器列表）	直接实现 [RFC 8106] §6.1/§6.2 的 DNS Server List 语义
CoreDNS	否	否；集群内作为被指向的 DNS 端点	Pod 的 DNS 由 Kubernetes 注入，与 RA 无关
Unbound	否	否；它是被 RDNSS 广告的**目标地址**	依赖外部 RA 发送者把 ULA 地址写进客户端
路由器固件	**是**（内核 / `radvd` / 厂商自研 RA 守护进程）	通常否（作为路由器角色）	[RFC 4861] / [RFC 4191] / [RFC 8106] 的主要实现方

由此产生一个很实际的差别：dnsmasq 把「发 RA」和「跑 DNS」绑在一起，改一个进程的生命周期会同时影响客户端的地址配置与 DNS；而 Unbound / BIND / CoreDNS 必须与独立的 RA 发送者配合，两者可以分别升级、分别回滚。第 19 章选择「独立的 RA 发送进程 + 独立的 Unbound 实例」，正是利用了这层解耦。

谁真正把 RDNSS 装进内核之外的 resolver 仓储，各家网络栈差异最大：Linux 的 systemd-resolved 按 per-link 维护 DNS 服务器并做路由域分流；Android、iOS、Windows、macOS 各自有独立的 RA 消费实现，对「由第三方撤销别人的 RDNSS 条目」「是否接受链路本地 RDNSS」「hoplimit 校验的严格程度」处理并不完全一致。这些属于实现本地策略，RFC 条文只规定了 MUST 的最小集合，不保证跨栈一致性——所以 18.4 才强调撤销要持续发、验证要按客户端分别做。

18.10.2 谁做 DNSSEC 验证

这里的分界必须和全书口径一致：DNSSEC = **数据完整性与来源认证**，DoT / DoH / DoQ = **传输加密与隐私**，互不替代。

实现	DNSSEC 验证角色	与 RA / RDNSS 的关系
Unbound	完整验证器（validator），信任锚 + 签名校验，坏签名→ SERVFAIL	自己不收 RA；作为 RDNSS 广告地址的目标
BIND	可作验证器（也可作权威服务器）	同上，与 RA 无耦合
dnsmasq	支持 DNSSEC 验证（通常作为转发侧的小型验证器，能力定位见官方手册）	是唯一同时发 RA 且能做验证的常见组合
systemd-resolved	支持 DNSSEC，但默认行为与降级策略按发行版配置而异，见官方手册	它是 RDNSS 的消费者，验证发生在同一进程内
CoreDNS	本身不做完整递归验证；可由插件链把验证交给上游	集群内服务端点，与 RA 无关
路由器固件	绝大多数只做转发 / 代理，不做验证	它是 RDNSS 的**广告者**，广告谁就决定了客户端把验证托付给谁

关键结论：**RDNSS 选项里只放得下 IPv6 地址，放不下「这个解析器会不会验证」这种能力声明**。客户端无法从 RA 里得知对端是不是验证器。所以「全屋 DNSSEC 验证」必须由网关侧保证 RDNSS 指向的那个实例确实在做验证，而不是靠客户端选择；第 10 章的三种验证状态在这里同样适用。

同理，RA 里也没有「加密传输」能力字段。DDR（[RFC 9462]）与 DoT / DoH / DoQ 解析器的自动发现是 DNS / HTTP 层的另一套机制（第 17 章讲协议对比，第 13 章讲压制），与 RDNSS 是两条独立通道。

18.10.3　谁支持 DDR

实现	DDR 相关行为	说明
路由器固件	一般不参与；DDR 是主机与指定解析器之间的发现机制	但固件若提供 DNS 代理，可能成为客户端绕过本地解析器的路径
systemd-resolved	与网络级「指定解析器 / 每个链路的加密解析器配置」关联最紧，具体支持版本见官方手册	它同时是 RDNSS 消费者，因此是两条通道的交汇点
BIND / Unbound / CoreDNS / dnsmasq	是否为 DDR 提供发现侧响应取决于版本与配置，见各自官方手册	这些进程主要扮演「被发现的解析器」，不是发起发现的客户端

DDR 与本章主线的关系是一处**绕过风险**：RDNSS 把客户端指向了网关 ULA 上的 Unbound，但客户端若通过 DDR 自动发现某个加密解析器并改用 DoH / DoT，网关侧的过滤、日志与验证就一起被绕过。第 13 章的 `local-zone` 压制与 canary 域是工程对策；协议层的成因是——**RA 只规定了「问谁」，没有规定「必须用明文 53 问、且不许另找别人」**。

18.10.4　三组条文 × 实现职责速查

条文	由谁实现	常见缺口
[RFC 4861] hop limit 255 / 有效性检查	主机内核 + 路由器固件 / RA 发送者	中间设备改写 hoplimit；SLLAO 被误当必填
[RFC 4862] §5.5.3 的 2 小时下限	主机内核	误以为 PIO `lifetime=0` 立即删地址
[RFC 8106] RDNSS + Lifetime=0 撤销	systemd-resolved 等主机网络栈（消费）、dnsmasq / 固件（发送）	撤销只发一次；跨栈对第三方撤销接受度不一
[RFC 4191] RIO / `Prf`	主机网络栈（Type A/B/C 支持度不同）、路由器固件	Android `/48` 下限被当成 RFC 条文
[RFC 4941] / [RFC 8981] 临时地址	主机内核	前缀轮换后旧临时地址仍在 preferred 状态
[RFC 6724] Rule 3 / 7	主机内核 + 本地策略表	把旧版 [RFC 3484] 的规则编号套到现行实现
[RFC 8200] 1280 最小 MTU	全链路	EDNS 载荷超过 1232 导致分片被丢
[RFC 9462] DDR	客户端网络栈 + 解析器侧	与 RDNSS 并行造成绕过

这张表要传达的不是「谁功能多」，而是**一条 RA 从构造到被消费，中途至少跨了三四个互不知情的实现**。RA 发送者按 [RFC 4861] 置 255，主机内核按 [RFC 4862] 决定地址寿命，用户态 resolved 按 [RFC 8106] 装 DNS 列表，最后应用按 [RFC 6724] 选源地址——任何一环的本地策略差异，都不会在别处留下显眼报错。排查 IPv6 DNS 问题时先看报文、再看客户端状态，而不是先怀疑 Unbound 配置，原因就在这里。

部署侧的可复制命令、验收清单与回滚点在第 19 章；第 20、21 章把这套分工搬到集群与「一台家庭网关 + 一台云主机」的实验环境；真实故障的教学案例集中在第 22 章。本章到此只回答「为什么这么规定」。

18.11　[RFC 6724] 的另一面：`getaddrinfo()` 的返回顺序

本章前面谈 [RFC 6724] 时，重点在**选哪个源地址**。但同一套策略还决定了另一件更容易被忽视的事：应用调用 `getaddrinfo()` 时，拿到的**候选目的地址列表是有顺序的**，双栈域名下 v6 往往排在 v4 前面。程序若只取前几个（例如常见写法 `infos[:3]`）就只试了 v6，于是「解析成功」和「能连上」变成两件事。

一个已脱敏的真实形态：某发片脚本在开工前对某国内平台域名做 `socket.getaddrinfo(host, 443)` + TCP 探测，只试前三个返回地址；该域名前 4 个全是 IPv6 且在这条链路上被 RST（`ECONNREFUSED`），后面 5 个 IPv4 全通。结果探测误报「TCP 连不上」，而浏览器走 v4 正常打开页面——日志于是出现「报故障→四秒后网络已恢复」的假振荡。**恢复并没有发生，只是上层页面走了另一条路**。

三条结论要记住：①探测必须**去重后全量遍历**（或至少 v4 优先），任一地址连通即视为通；②「已恢复」这类日志只能基于重探测结果，不能只看上层页面能否打开；③ `getaddrinfo` 的顺序是**策略排序**，不是「第一个最可用」。这个案例的完整排查决策表在第 36 章，源地址选择与前缀轮换见第 19、22 章，告警误报治理见第 31 章。

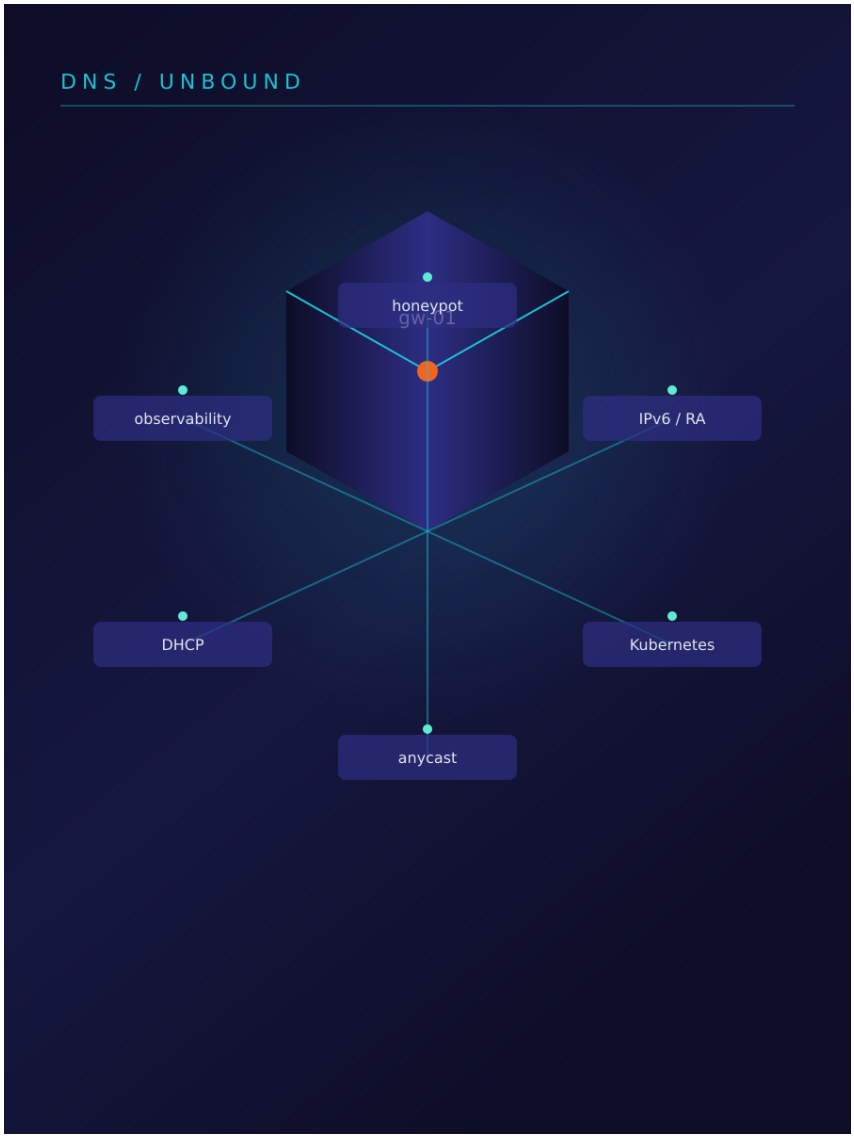
18.12 本章要点

- [RFC 4861] 强制所有 NDP 报文 hop limit 为 255 并由接收方校验，这是把 NDP 限制在本链路的协议级安全边界；`Router Lifetime=0` 只撤销默认路由器，不等于关闭 RA；SLLAO 在 RA 中是 MAY。
- [RFC 4862] §5.5.3 e) 明确规定 PIO 的短 / 零 Valid Lifetime 最多把已有的长寿命地址压到 2 小时（或忽略），防的是伪造 RA 造成全网地址瞬间过期的 DoS；因此「发一次 `lifetime=0` 就清掉旧前缀」不成立。[RFC 9096] 的对策是缩短 PIO 寿命并持续双 0 撤销。
- [RFC 8106] 取代 [RFC 6106]：RDNSS / DNSSL 有独立 Lifetime，§6.2 步骤 (b) 规定已存在地址配 `Lifetime=0` 时从 DNS Server List 与 Resolver Repository 中删除；这是「压掉光猫广告的 DNS」的条文依据，但撤销必须持续重复发送。
- [RFC 4191] 的 RIO 与默认路由器 `Prf` 是实现可选能力（Type A/B/C 主机）；Android `accept_ra_rt_info_min_plen=48` 是本地策略而非 RFC 条文，/48 用于路由聚合、/64 用于地址变更检测，两者不可混用。
- [RFC 6724] 里避开 deprecated 是 Rule 3、优先出接口是 Rule 5，Rule 7 已从 [RFC 3484] 的「优先公共地址」反转为「优先临时地址」；配合 [RFC 4941] / [RFC 8981] 的独立寿命时钟，构成前缀轮换后旧临时地址仍被选为源的故障链。
- [RFC 4193] ULA 不随运营商前缀轮换变化，适合作为 RDNSS 里发布的解析器地址；[RFC 8200] 的 1280 字节最小 MTU 是 EDNS 1232 取值的根因。横向对比的关键是角色划分：BIND / Unbound / CoreDNS 既不发 RA 也不消费 RDNSS，dnsmasq 与路由器固件发 RA，systemd-resolved 消费 RDNSS，DNSSEC 验证与 DDR 支持各自独立，互不替代。

18.13 参考资料

- RFC Editor, [RFC 4291] *IP Version 6 Addressing Architecture*: https://www.rfc-editor.org/rfc/rfc4291
- RFC Editor, [RFC 4861] *Neighbor Discovery for IP version 6 (IPv6)*: https://www.rfc-editor.org/rfc/rfc4861
- RFC Editor, [RFC 4862] *IPv6 Stateless Address Autoconfiguration*: https://www.rfc-editor.org/rfc/rfc4862
- RFC Editor, [RFC 8106] *IPv6 Router Advertisement Options for DNS Configuration*: https://www.rfc-editor.org/rfc/rfc8106
- RFC Editor, [RFC 6724] *Default Address Selection for Internet Protocol Version 6 (IPv6)*: https://www.rfc-editor.org/rfc/rfc6724
- RFC Editor, [RFC 9096] *Improving the Reaction of Customer Edge Routers to IPv6 Renumbering Events*: https://www.rfc-editor.org/rfc/rfc9096
- RFC Editor, [RFC 8200] *Internet Protocol, Version 6 (IPv6) Specification*: https://www.rfc-editor.org/rfc/rfc8200
- RFC Editor 官方 RFC 索引（编号、年份、obsoletes / updates 关系以此为准）：https://www.rfc-editor.org/rfc-index.html

第五篇　工程实践


DNS / UNBOUND
honeypot
observability
IPv6 / RA
DHCP
Kubernetes
anycast

第十九章

IPv6 部署：RA、RDNSS、ULA 与前缀轮换

19.1 「全屋无污染 DNS」的真正机制

把家里所有终端的 DNS 都干净地引到自己的解析器上，最容易想到的做法是「进光猫管理页面，把 DHCPv6 / DNS 选项改掉」。这条路既脆弱又常常走不通：光猫的固件不一定允许改，改了会在运营商下发配置时被覆盖，而且为了进这个页面去切换光猫的工作模式，会踩到 19.6 节那个代价极高的数据面缺陷。

真正稳定、且与客户端操作系统无关的机制只有一句话：

> **网关自己发 ICMPv6 路由器广告（Router Advertisement，RA），在 RA 的 RDNSS 选项里广告自己的 ULA 解析器地址；光猫里不需要配任何 DNS。**

这就是本书把 `ra-steer` 这类小程序放在网关上的原因。它不是 DHCPv6 服务器，也不碰光猫配置，只做一件事：周期性地在 LAN 侧发出一份内容可控的 RA。RA 是 IPv6 无状态地址自动配置（SLAAC）的原生入口，任何支持 IPv6 的终端——Android、iOS、Windows、macOS、Linux、智能电视——都必须处理它，才能拿到默认路由。既然一定要收，就可以顺带把 DNS 一起交给它。

这里有两层含义值得先分清：

层	谁负责	与 DNS 的关系
数据面	光猫（桥接/透传）、网关	只转发报文，不解释 DNS
控制面	网关上的 RA 发送进程 + 网关上的 Unbound 实例	决定终端把查询发给谁

「无污染」不是靠在光猫上屏蔽某个端口实现的，而是靠**客户端自己把递归解析器地址填成了网关的 ULA**。第 13 章讲的 DDR 压制、浏览器 canary 域、以及第 9 章的:53 劫持，都是这条主线之外的补漏手段：前者防止客户端绕过，后者兜住那些不理 RA 的存根解析器。

从部署视角列一份「报文里必须有、且必须验」的清单。

19.2 一份合格 RA 里的六项内容

19.2.1 组播与单播

RA 的默认目的地是 `ff02::1`（所有节点组播地址），周期由 `MaxRtrAdvInterval` 控制，典型值 600 秒。这个周期对「刚插上网线的手机」太慢，所以还要支持**单播 RA**：终端启动时发 RS（Router Solicitation）到 `ff02::2`，网关收到后立刻以单播回一份 RA，不必等下一个组播周期。

```
# ra-steer 侧的时间参数（教学骨架，见 19.7 的脚本）
AdvSendAdvert=yes
```

```
MaxRtrAdvInterval=60        # 组播周期压到 60s，缩短新设备收敛时间
MinRtrAdvInterval=20
AdvManagedFlag=no           # 不用 DHCPv6 管地址，纯 SLAAC
AdvOtherConfigFlag=yes      # 允许客户端继续取「其他信息」（含 RDNSS）
```

为什么这么设：600 秒的默认周期意味着一台刚回到家的设备可能要等十分钟才拿到正确 DNS；压到 60 秒并配合单播回应 RS，可以把收敛拉到秒级。`AdvOtherConfigFlag=yes` 保证只靠 RA 也能携带 DNS，不依赖 DHCPv6 服务器。

怎么验证：`tcpdump -i lan0 -n 'icmp6 && ip6[40] == 134' -vv -c 4`，观察是否同时出现组播 RA 与回应 RS 的单播 RA。

失败模式 / 回滚点：周期过短会在电池供电的终端上造成可感知的唤醒开销；若观察到终端耗电异常，把 `MaxRtrAdvInterval` 退回 600，并保留单播回应 RS 作为快速通道。改坏时停掉发送进程即可，网关上原有的 `radvd` 配置备份在 `/etc/ra-steer/ra.conf.bak`。

19.2.2　SLLAO：源链路层地址选项

RA 里必须带 SLLAO（Source Link-Layer Address Option）。缺了它，客户端无法把「默认路由器」映射成邻居缓存里的一个可达项，表现为能收到 RA、能配上地址，但发不出去包，或者每包都要先做一次邻居发现。

为什么这么设：SLLAO 让客户端一次性建立 `网关 IPv6 链路本地地址 → LAN MAC` 的邻居表项，省掉一次 NDP 往返；在 Wi-Fi 上还能减少组播邻居请求。

怎么验证：`tcpdump -i lan0 -n 'icmp6 && ip6[40] == 134' -vv | grep -i 'source link-layer'`。

失败模式 / 回滚点：SLLAO 里填错 MAC（例如填了 WAN 口而不是 LAN 口）会导致全屋 IPv6 黑洞，且现象很容易被误判成「运营商没给前缀」。核对 `ip -6 neigh show` 里的路由器表项；回滚时把 SLLAO 改为由脚本从 `lan0` 的 `address` 属性读取，不要手写常量。

19.2.3　hop limit 255：iOS 的硬性要求

[RFC 4861] 要求接收端校验 RA 的 IPv6 Hop Limit 必须是 255，用来保证报文没有跨过路由器。多数 Linux 客户端只是宽松检查，而 iOS 是**强制**的：hop limit 不是 255 的 RA 会被静默丢弃，其他一切都正常，只有 iPhone / iPad 拿不到 DNS。正因为这个差异，故障很容易被归到「iOS 兼容性问题」而拖很久。

```
# 发送 RA 前必须确认出站报文的 hoplimit 未被 nftables/ip6tables 改写
ip6tables -t mangle -L -n -v | grep -i hl || true
nft list ruleset | grep -i 'hoplimit' || true
```

为什么这么设：任何在 `mangle/inet` 表里做 `hoplimit set` 的 QoS 或策略路由规则，都可能把 RA 的 255 改掉；先确认没有这类规则，再谈客户端兼容。

怎么验证：`tcpdump -i lan0 -n 'icmp6 && ip6[40] == 134' -vv -c 1` 的输出里 `hlim 255`（有的版本写作 `hop limit 255`）。

失败模式 / 回滚点：若抓到 254 或 64，回滚点是 `nft flush chain inet mangle ...` 之前保存的 `nft list ruleset > /etc/nftables.conf.bak`；不要靠重启客户端掩盖，iOS 不会改变这个要求。

19.2.4　RDNSS：广告自己的 ULA，并用 lifetime=0 压掉上级广告

RDNSS 选项（[RFC 8106]，取代 [RFC 6106]）允许在一个 RA 里携带 1–3 个递归解析器地址。客户端会把**所有路由器**广告的 RDNSS 合并成一张列表，而不是「后到覆盖先到」。所以只发自己的地址是不够的：光猫还在同一个二层域里发 RA，它的 DNS 地址会一直留在列表里，客户端随时可能挑中它。

工程上的做法是让网关的 RA 同时携带两类 RDNSS 选项：

```
# /etc/ra-steer/ra.conf 片段（教学骨架）
RDNSS fd00:53::1 fd00:53::2 {
    AdvRDNSSLifetime 60;        # 自己的两个 ULA 解析器地址，正常寿命
};
RDNSS fd00:53::9 {
    AdvRDNSSLifetime 0;         # 光猫广告的那个地址：显式撤销
};
```

为什么这么设：`Lifetime=0` 的语义是「该地址不再作为 RDNSS 使用」，用来在客户端的合并表里把上级路由器广告的地址标为失效；自己的两个 ULA 地址给 60 秒寿命，随每次 RA 续期。两条都是网关自己的 ULA 实例地址，对应第 7 章的多实例分工。

怎么验证：在 Android 上 `adb shell dumpsys connectivity | grep -i -A3 'DnsAddresses'`，期望只列出 `fd00:53::1` 与 `fd00:53::2` 两条；iOS 侧用 `tcpdump` 抓 RA 核对选项内容，或用描述文件下发前的「Wi-Fi →配置 DNS」页面确认没有第三条。

失败模式 / 回滚点：并非所有客户端都接受「由另一台路由器撤销第三方广告的地址」，某些实现只认原广告者的撤销。若验证时仍看到第三条，退路是在网关上阻断来自 WAN 侧的 RA（只放行自己的 RA），而不是去改光猫 DNS；这一步会改变二层拓扑，务必先备份 `nft list ruleset`。另一个坑是撤销必须**持续重复发送**：终端重新联网后会重新接收光猫的 RA，一次性撤销很快失效。

19.2.5 PIO 与 RIO：/64 和 /48 各管一件事

PIO（Prefix Information Option）告诉客户端「你可以在这个 /64 上配 SLAAC 地址」；RIO（Route Information Option，[RFC 4191]）告诉客户端「去这个前缀要走我」。两者的粒度不能混用，这里有个一天之内栽两次的坑。

Android 内核的 `accept_ra_rt_info_min_plen=48` 会**丢弃前缀长度小于 48 的 RIO**。因此下游路由广告必须写成 /48（或更细），写 /32 之类的聚合会被静默忽略，现象是「路由表里什么都没多」。但前缀变更检测脚本必须盯的是 PIO 里的 **/64**：运营商轮换的是 LAN 的 /64，而 /48 聚合在这个轮换中不变。

第一次栽在 DDNS 更新脚本：脚本从 RIO 里取 /48 拼出记录值，运营商换 /64 后 DDNS 纹丝不动。第二次栽在 RA 通知脚本：同一个 /48 变量被用来判断「前缀是否变化」，于是既不发新 RA 通知，也不触发临时地址重建。两处故障表现完全不同，根因是同一个。

```
# 只从 PIO 取 LAN /64，绝不从 RIO 推导
rdisc6 -1 lan0 | grep -i 'Prefix' || true
ip -6 route show | grep '^2001:db8' || true
```

为什么这么设：把「路由聚合粒度」和「地址变更粒度」拆成两个独立数据源，避免一个变量被两处脚本复用而同时失效。

怎么验证：`ip -6 addr show dev lan0 scope global`，确认地址来自当前下发的 /64；轮换后该命令输出的 /64 必须变化，而 RIO 生成的 /48 路由保持不变即为正确。

失败模式 / 回滚点：若脚本已经把 /48 写进 DDNS，先手工更新一次记录再回滚 DDNS 脚本变量；不要靠清缓存解决，负缓存会按 SOA minimum 续命（第 22 章案例⑤）。

19.3 ULA 做 DNS 地址，GUA 做上网地址

家宽下发的 GUA（Global Unicast Address）前缀会轮换，教学示例里从 `2001:db8:1:1::/64` 变成 `2001:db8:1:9::/64` 是常态。若把 RDNSS 里的解析器地址写成 GUA，那么每次轮换：

1. 所有客户端手里的 DNS 地址失效；
2. 新 RA 到达前的一段时间里查询全部超时；
3. 即便新 RA 到达，负缓存和应用的连接池还可能继续用旧地址（见第 22 章案例⑤）。

ULA（`fd00:53::/48` 这一段）由网关自己生成，[RFC 4193] 要求用随机 40 位全局 ID 生成 /48，一旦选定就不随运营商变化。它不用于访问公网，只用于「在自家二层域里稳定地寻址自己的服务」——DNS 正好是这种服务。

```
# 网关 LAN 口的地址分工（教学示例）
lan0: fd00:53::1/64        # RDNSS 第一条，永久
lan0: fd00:53::2/64        # RDNSS 第二条（第二个 Unbound 实例），永久
lan0: 2001:db8:1:1::1/64  # 由运营商下发的 GUA，会轮换，不作 DNS 地址
```

为什么这么设：把「会变的」和「不能变的」彻底分开，DNS 引导路径上不出现任何依赖运营商下发状态的常量。

怎么验证：`dig @fd00:53::1 +short signed.dns.example.org A` 应成功，且 `ip -6 route get fd00:53::1` 显示直连 LAN，不经过默认路由。

失败模式 / 回滚点：ULA 不是公网可路由地址，若某台终端把 `fd00:53::1` 当成公网目标往隧道里送，会立刻超时；回滚点在 nftables 的策略路由规则里保留一条「目的 ULA →直连 LAN」的豁免。生成 ULA 时不要图省事写全零全局 ID，按 [RFC 4193] 随机生成，避免以后合并网络时撞前缀。

19.4　前缀轮换 × 临时地址寿命

[RFC 4941] 隐私扩展生成的临时地址，在 Linux 上默认 `temp_valid_lft` 是 **48 小时**（172800 秒），`temp_prefered_lft` 是 24 小时。运营商换前缀之后，旧前缀上的临时地址依然有长达两天的有效寿命，而 [RFC 6724] 的地址选择规则**优先选临时地址作为源地址**。结果是一个特别难查的现象：

> 客户端拿到了新前缀、配了新地址、默认路由也正确，但出站连接的源地址仍是旧前缀→上游不路由回包→「IPv6 全断，ping6 网关却通」。

把寿命压到 30 分钟，让旧临时地址在轮换后尽快退场：

```
# /etc/sysctl.d/90-ipv6-privacy.conf
net.ipv6.conf.all.use_tempaddr      = 2
net.ipv6.conf.default.use_tempaddr  = 2
net.ipv6.conf.all.temp_valid_lft    = 1800
net.ipv6.conf.default.temp_valid_lft= 1800
net.ipv6.conf.all.temp_prefered_lft = 600
net.ipv6.conf.default.temp_prefered_lft = 600
```

为什么这么设：`use_tempaddr = 2` 表示「生成并优先使用临时地址」；`temp_valid_lft = 1800` 把旧前缀的最长残留时间从 48 小时压到 30 分钟，把一次前缀轮换的故障窗口收进可接受范围。

怎么验证：`sysctl net.ipv6.conf.lan0.temp_valid_lft` 必须返回 `1800`；只查 `all.*` 不算数。

失败模式 / 回滚点：这份文件只对**之后新建的接口**生效。已存在的接口不会被 `all.*` 回溯改写，NetworkManager 还会在连接激活时把 `use_tempaddr` 写成 `-1`（继承内核默认）覆盖你的值。发现不生效时按 19.7 的三处补齐；回滚只需删除该文件并 `sysctl --system`，但注意删除后旧接口的值仍留在运行时，要显式写回。

三处都要设，这是 Linux sysctl 的一个反直觉边界：

写入位置	作用范围	只写它会怎样
`all.*`	全局「上限/聚合」视图	**对已存在的接口无效**，最典型的假成功
`default.*`	之后新建的接口	当前 LAN 接口不受影响
`<iface>.*`	这个具体接口	只写它，新接口（USB 网卡、热点）又漏掉

```
# 对已存在接口补写，幂等，可重复执行
IFACE=lan0
for k in use_tempaddr temp_valid_lft temp_prefered_lft; do
  v=$(awk -v k="$k" '$0 ~ "net.ipv6.conf.all."k {print $3}' /etc/sysctl.d/90-ipv6-privacy.conf)
  sysctl -w "net.ipv6.conf.${IFACE}.${k}=${v}"
done
```

为什么这么设：把「配置文件的意图」逐项投放到运行时已存在的接口上，绕开 `all.*` 不回溯的限制；写成循环而不是硬编码，避免以后改了文件却忘了改命令。

怎么验证：`sysctl -a | grep 'net.ipv6.conf.lan0.temp'` 三行都应为 `2` / `1800` / `600`。

失败模式 / 回滚点：NetworkManager 连接激活会再次覆盖接口级值，长期方案是给连接加 `ipv6.ip6-privacy` 或在 `NetworkManager.conf` 里禁用该连接的 sysctl 改写。回滚时把接口级值写回内核默认，并保留一份 `sysctl -a > /var/tmp/sysctl-before.txt` 快照。

19.4.1　STALE PIO 为什么清不掉

一个自然的想法是：轮换时发一个 `Valid Lifetime = 0` 的 PIO，让客户端立刻丢掉旧前缀地址。在 Linux 上这**不生效**。[RFC 4862] 第 5.5.3 节为抵御伪造 RA 的拒绝服务攻击，规定对仍剩较长有效期的地址不允许被一个短寿命（或零寿命）的 PIO 立刻缩短——只能把有效期夹到 2 小时量级，具体夹取规则见该节。于是不管怎么发「撤销」，旧地址都会赖着不走，这就是所谓 STALE PIO 清理无效。

真正有效的只有两条路：**等**（靠上面把 `temp_valid_lft` 压短），以及**在客户端侧主动删地址**（由前缀变更检测脚本调用 `ip -6 addr flush ... scope global dynamic` 一类的动作，且要避开 ULA）。不要指望 RA 单方面完成清理。

19.5 光猫只当管道用：一次数据面缺陷的教训

为了进管理页面改 DNS，实验里曾把光猫从桥接模式改回路由模式。配置页面确实进去了，IPv6 地址也照常下发，随后出现一个非常干净、也非常迷惑的故障分层：

测试	结果	说明
ICMPv6 ping 通外网文档地址	通	控制面看起来完好
TCP 三次握手	通	`connect()` 成功，SYN/SYN-ACK/ACK 完整
TCP 数据阶段（首个有载荷的段）	**静默丢包**	无 RST、无 ICMPv6 报错，连接挂住直到超时
换回桥接模式	全部恢复	与 DNS、RA、Unbound 配置无关

抓包看到的现象是握手包很小所以能通过，一旦出现接近 MTU 的载荷就再也收不到 ACK，重传也无回应；换成小包手动分片又能零星走一点。这是光猫在路由模式下某个 IPv6 数据面路径（分段/重组或硬件加速表项）的缺陷，不是 DNS 配置错误，也不是 MTU 设错——同一条链路在桥接模式下完全一致。

代价是：一个本来只想改 DNS 的动作，把整条 IPv6 数据面换成了不可信的实现，排查方向还被「ping 通 / 握手通」误导了好几个小时。由此定下一条部署纪律：

> **光猫只当管道用**。保持桥接/透传，所有三层及以上能力（RA、DNS、分流、加密隧道、防火墙）都放在自己可控的网关 `gw-01` 上；任何需要「进光猫改点什么」的方案，先问一句能不能用网关侧发 RA 替代。

这条纪律与 19.1 是同一个结论的两面：RDNSS 方案之所以值得写透，正是因为它让「改光猫」这个动作变得完全不必要。第 14 章的加密通道选路、第 21 章的拓扑拼装，都建立在这个前提上。

如果已经改了工作模式且暂时无法改回，不要试图在光猫里调 MTU 兜底：把 IPv6 出向流量整体绕过它（例如只走加密隧道承载的 IPv4 通道）是更可控的临时方案，同时把隧道 MTU 与 `edns-buffer-size: 1232` 一起核对（见第 22 章案例③与第 13 章）。

19.6 可复制的 sysctl 与 RA 引导脚本骨架

下面是一份**骨架**，不是开箱即用的发行版包：它把需要显式决策的量都留成变量，避免把某个环境的接口名、ULA、上游写死。配置块只给引导层，Unbound 实例本身的配置见第 5、7 章。

```
#!/bin/sh
# /usr/local/sbin/ra-steer-boot.sh ——教学骨架
set -eu
IFACE="${IFACE:-lan0}"
ULA_DNS1="${ULA_DNS1:-fd00:53::1}"
ULA_DNS2="${ULA_DNS2:-fd00:53::2}"
UPSTREAM_DNS="${UPSTREAM_DNS:-fd00:53::9}"    # 需要撤销的上级广告地址
CONF=/etc/ra-steer/ra.conf
BAK=/etc/ra-steer/ra.conf.bak

# 1) 快照，回滚点
sysctl -a > /var/tmp/sysctl-before-ra-steer.txt
[ -f "$CONF" ] && cp -a "$CONF" "$BAK"

# 2) 三处 sysctl：全局 / 默认 / 接口级
sysctl -w net.ipv6.conf.all.accept_ra=2
sysctl -w net.ipv6.conf.default.accept_ra=2
sysctl -w "net.ipv6.conf.${IFACE}.accept_ra=2"
for k in use_tempaddr:2 temp_valid_lft:1800 temp_prefered_lft:600; do
  key=${k%%:*}; val=${k##*:}
  sysctl -w "net.ipv6.conf.all.${key}=${val}"
  sysctl -w "net.ipv6.conf.default.${key}=${val}"
  sysctl -w "net.ipv6.conf.${IFACE}.${key}=${val}"
done
```

```
# 3) 生成 RA 配置：自己的 ULA 正常寿命 + 上级地址 lifetime=0
cat > "$CONF" <<EOF
interface ${IFACE} {
  AdvSendAdvert on;
  MaxRtrAdvInterval 60;
  MinRtrAdvInterval 20;
  AdvDefaultLifetime 180;
  AdvSourceLLAddress on;
  AdvManagedFlag off;
  AdvOtherConfigFlag on;
  prefix 2001:db8:1:1::/64 { AdvAutonomous on; AdvValidLifetime 3600; };
  route 2001:db8:1::/48 { AdvRouteLifetime 3600; };
  RDNSS ${ULA_DNS1} ${ULA_DNS2} { AdvRDNSSLifetime 60; };
  RDNSS ${UPSTREAM_DNS} { AdvRDNSSLifetime 0; };
};
EOF

# 4) 交给 RA 发送器；hop limit 由内核置 255，禁止任何规则改写
exec /usr/local/sbin/ra-steer --conf "$CONF" --iface "$IFACE" --hoplimit 255
```

为什么这么设：`accept_ra=2` 是网关开启 IPv6 转发时仍接收 RA 的必要条件；PIO 用 /64、RIO 用 /48，落实 19.2 的粒度分离；`; AdvSourceLLAddress on` 对应 SLLAO；`--hoplimit 255` 对应 iOS 的硬性要求；第 1 步先做快照，保证任何一步失败都有可退回的状态。

怎么验证：`tcpdump -i lan0 -n -vv 'icmp6 && ip6[40] == 134' -c 1` 应同时看到 `hlim 255`、SLLAO、/64 PIO、/48 RIO、一条正常 RDNSS 与一条 lifetime 0 的 RDNSS。

失败模式 / 回滚点：`prefix` 行写死的教学 /64 必须替换成运行时从上游下发的真实值，否则轮换后会把旧前缀继续广告出去（正是 19.2 那个坑）；生产脚本应从 `dhcpcd/dhclient` 的钩子读入变量。回滚顺序：停掉 `ra-steer` →用 `$BAK` 覆盖 `$CONF` → `sysctl --system` 加上 `/var/tmp/sysctl-before-ra-steer.txt` 逐项对照→让光猫原有的 RA 重新填满客户端列表。若担心操作失误把自己锁在外面，用 `systemd-run --on-active=10m systemctl stop ra-steer` 设一个十分钟的自动回滚定时器，确认无误后再取消。

19.7　验证命令与验收清单

引导脚本跑起来之后，按「链路→报文→客户端→解析器」四层验，不要跳层。

层	命令	通过标准
链路	`ip -6 addr show dev lan0 scope global`	有 ULA `fd00:53::1`，GUA 为当前下发的 /64
报文	`tcpdump -i lan0 -n -vv 'icmp6 && ip6[40] == 134' -c 2`	`hlim 255`、SLLAO、PIO /64、RIO /48 齐全
报文	`rdisc6 -1 lan0`	能看到两条 RDNSS，且撤销条目的 lifetime 为 0
客户端（Android）	`adb shell dumpsys connectivity | grep -i -A3 DnsAddresses`	**只显示自己的两条** ULA 地址
客户端（Linux）	`resolvectl dns lan0` 或 `cat /etc/resolv.conf`	同上，无第三条
客户端（iOS）	抓 RA 核对；描述文件未接管时查看 Wi-Fi 的 DNS 栏	无光猫下发的 DNS
解析器	`dig @fd00:53::1 +dnssec signed.dns.example.org A`	返回答案且 `flags:` 含 `ad`（第 10 章）
解析器	`dig @fd00:53::2 +short +dnssec signed.dns.example.org A`	第二个实例同样可用（第 7 章）

关于「手机 DNS 应只显示自己的两条」需要说清期望从哪来：`fd00:53::1` 与 `fd00:53::2` 是网关上两个 Unbound 实例的 ULA 地址。如果只部署了一个实例，列表里就只该有一条；出现第三条（尤其是光猫的链路本地地址或一个 GUA 地址），说明 RDNSS 撤销没被接受、或者光猫的 RA 没被隔离，回到 19.2 的退路。

还要验一次**轮换演练**，否则前面所有配置都只是静态正确：

```
# 模拟一次 /64 轮换（教学：手工触发，不改光猫）
ip -6 addr flush dev lan0 scope global dynamic
ip -6 addr add 2001:db8:1:9::1/64 dev lan0
/usr/local/sbin/ra-steer-boot.sh >/var/tmp/ra-steer-reload.log 2>&1
sleep 35
ip -6 addr show dev lan0 scope global | grep -c '2001:db8:1:1' || true   # 期望 0
```

为什么这么设：静态验证查不出「旧临时地址赖着不走」和「DDNS 用 /48 推不出变化」这两类问题，必须让前缀真的变一次。35 秒略大于 `temp_valid_lft` 之外的观察窗，用来确认旧地址没有被重新生成。

怎么验证：`ip -6 addr show dev lan0 scope global` 里不再出现旧 /64；`adb shell dumpsys connectivity | grep -i -A3 DnsAddresses` 仍是那两条 ULA。

失败模式 / 回滚点：`flush` 会把 ULA 之外的动态地址一并清掉，若误加到 `fd00:53::1` 会立刻让全屋 DNS 失联；执行前确认 `grep -q 'fd00:53::1' /var/tmp/sysctl-before-ra-steer.txt` 之类的快照存在，回滚用 `ip -6 addr add fd00:53::1/64 dev lan0` 先救回 DNS，再谈其余。

19.8 回滚点汇总

变更	快照	回滚动作	影响面
RA 发送进程	`/etc/ra-steer/ra.conf.bak`	停进程 + 还原配置 + 让光猫 RA 恢复	客户端 DNS 列表回到改动前
sysctl	`/var/tmp/sysctl-before-ra-steer.txt`	`sysctl --system` 并逐项写回接口级值	临时地址寿命恢复 48h
GUA / ULA 地址	`ip -6 addr show > /var/tmp/addr-before.txt`	`ip -6 addr add` 逐条补回	全屋 DNS 可达性
nftables（若做了 RA 隔离）	`nft list ruleset > /etc/nftables.conf.bak`	`nft -f /etc/nftables.conf.bak`	二层 RA 来源
光猫工作模式	改动前拍照记录	改回桥接/透传	整个 IPv6 数据面

三点纪律：其一，回滚顺序**先救 DNS 地址，再救路由，最后救策略**，因为所有远程排障手段都依赖 DNS；其二，改动前设十分钟自动回滚定时器，防止配置错误把自己关在门外；其三，遇到任何「看起来像 DNS、又不太像」的故障，按第 22 章那条经验——**先看 nftables，再怀疑别的**。

19.9 本章要点

1. 「全屋无污染 DNS」的机制是网关自己发 RA 广告 RDNSS（ULA 地址），**不需要去光猫里配任何 DNS**；光猫只当管道用。
2. 一份合格 RA 要有：组播 + 单播回应 RS、SLLAO、**hop limit 255**（iOS 强制）、自己的 ULA RDNSS、对上级广告地址的 `lifetime=0` 撤销；PIO 用 /64、RIO 用 /48，两种粒度不能互推。
3. ULA 永久不变适合作 DNS 地址，GUA /64 会轮换绝不能写进 RDNSS；前缀变更检测必须盯 /64，否则 DDNS 与 RA 通知会同一天各栽一次。
4. Linux 上 `temp_valid_lft` 默认 48h，前缀轮换后旧临时地址仍被 [RFC 6724] 优先选作源地址；改 30min 且 `all.*` / `default.*` / 接口级三**处都要设**，NetworkManager 会把 `use_tempaddr` 写成 `-1`。
5. 发零寿命 PIO 清不掉旧地址：[RFC 4862] 第 5.5.3 节不允许把已存在的长寿命地址立刻缩短，只能靠短寿命 + 客户端侧主动清理。
6. 为改 DNS 去动光猫工作模式，可能踩到「ICMP 通、TCP 握手通、数据阶段静默丢包」的数据面缺陷；这是把可控性交出去的代价。

19.10 参考资料

- RFC 4861, *Neighbor Discovery for IP version 6*（SLLAO、hop limit 255）：https://www.rfc-editor.org/rfc/rfc4861

- RFC 4862, *IPv6 Stateless Address Autoconfiguration*（第 5.5.3 节地址寿命）：https://www.rfc-editor.org/rfc/rfc4862
- RFC 8106, *IPv6 Router Advertisement Options for DNS Configuration*（取代 [RFC 6106]）：https://www.rfc-editor.org/rfc/rfc8106
- RFC 4191, *Default Router Preferences and More-Specific Routes*（RIO）：https://www.rfc-editor.org/rfc/rfc4191
- RFC 4941, *Privacy Extensions for SLAAC*（临时地址；另见 [RFC 6724] 源地址选择、[RFC 4193] ULA）：https://www.rfc-editor.org/rfc/rfc4941
- RFC 6724, *Default Address Selection for IPv6*：https://www.rfc-editor.org/rfc/rfc6724
- Linux 内核 `ip-sysctl` 文档（`temp_valid_lft`、`accept_ra_rt_info_min_plen`）：https://docs.kernel.org/networking/ip-sysctl.html
- Unbound 配置手册：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html

第二十章

Kubernetes 中的 DNS 选型

20.1 先划清一条边界：服务发现≠递归解析

Kubernetes 里的 DNS 问题和本书前四篇讨论的不是同一个问题。前四篇的递归解析器（recursive resolver）面对的是**整个公网名字空间**：从根开始迭代、做 DNSSEC 验证、缓存、过滤、走加密上游。集群里的 DNS 面对的是一个 **closed world**：`svc.example.internal`、`kube-dns.kube-system.svc.example.internal`、`<pod-ip>.<namespace>.pod.example.internal` 这些名字只在集群内有意义，答案来自 API Server 里的 Service / EndpointSlice 对象，而不是任何权威服务器。

由此得出本章最重要的结论，后面所有拓扑都围绕它展开：

> **CoreDNS 负责集群内服务发现，Unbound 负责外部递归 / 验证 / 过滤 / 加密上游。两者不是替代关系。**

把 Unbound 直接装成 `kube-system` 里的 DNS Service 是行不通的：它不知道 Service 何时扩缩容，也不会 watch EndpointSlice，`dig @unbound mysvc.default.svc.example.internal` 只会往公网根区问一个根本不存在的名字。反过来，把 CoreDNS 当成公网递归解析器用也不成立：它的 `forward` 插件只是把查询转交给上游，验证、QMIN、负缓存策略、`edns-buffer-size` 这些都是交给别人的事；CoreDNS 自己没有 DNSSEC 验证链、没有根区 priming、没有信任锚（trust anchor）管理。第 5 章讲转发器、第 10 章讲验证器，这两件事在 Kubernetes 里分别落在 CoreDNS 的 `forward` 与集群外的 Unbound 上。

一个常见的误判是「CoreDNS 已经做了缓存，为什么还要 NodeLocal 或 Unbound」。因为缓存的**位置**决定了它能否绕开内核的 conntrack 竞态（20.5），而缓存的**语义**（验证与否、负缓存多久、是否 serve-stale）决定了它能否满足第三篇的加固基线（20.7）。

20.2 从 kube-dns 到 CoreDNS

早期集群里 `kube-dns` 是一个三容器组合：`kubedns`（watch Service/Endpoints 并维护一棵内存树）、`dnsmasq`（缓存 + 存根转发）、`sidecar`（健康与指标）。CoreDNS 自 Kubernetes 1.11 起成为可选、1.13 起成为默认（具体版本以官方发布说明为准，见参考资料）。更替不是因为功能表多了一行，而是四个工程原因：

维度	kube-dns	CoreDNS
结构	三个容器 + 进程间 DNS 转发	单进程，插件链（plugin chain）按 Corefile 顺序执行
性能	dnsmasq 与 kubedns 之间多一次本机 UDP 转发，内存与 GC 开销更高	单进程内完成匹配，资源占用更低
可维护性	改行为要动 Go 代码、重新构建三个镜像	改 Corefile 增删插件即可，`reload` 热加载
生态	项目基本停止演进	CNCF 毕业项目，插件与上游生态持续维护

「插件链」是理解 Corefile 的钥匙。CoreDNS 收到一个查询后，按每个 server block 的插件顺序逐个尝试；`kubernetes` 插件命中集群内名字就作答并终止，`forward` 插件把剩余名字交给上游，`cache` 在链中位置决定了它缓存的是「插件处理后的答案」。顺序写错会直接改变语义，例如把 `cache` 放在 `kubernetes` 之前，就会把 Service 答案按 TTL 缓存住而感知不到 EndpointSlice 的变化（CoreDNS 的 `kubernetes` 插件默认给很低/零 TTL，但链的顺序仍应谨慎）。

另需记住：CoreDNS 是 CNCF 毕业项目，这一身份说明它解决的是**云原生服务发现**问题，而不是「公网递归解析器」问题。CNCF 的成熟度评价不构成把它当验证器使用的依据。

20.3　四种 dnsPolicy 与 resolv.conf 的形态

Pod 的 `dnsPolicy` 决定 kubelet 往容器里写一份什么样的 `/etc/resolv.conf`。这是一个很具体的、可以 `kubectl exec` 出来看的差异。

```yaml
# 教学示例：四种 dnsPolicy 的最小对照
apiVersion: v1
kind: Pod
metadata:
  name: dns-policy-demo
  namespace: default
spec:
  dnsPolicy: ClusterFirst          # 依次改成下列四值观察
  containers:
    - name: tools
      image: registry.example.internal/net-tools:latest
      command: ["sleep", "3600"]
```

为什么这么设：用一个只有 `sleep` 的探针 Pod，把变量收敛到 `dnsPolicy` 一项，避免业务镜像自带的 `/etc/resolv.conf` 挂载干扰观察。

怎么验证：`kubectl exec dns-policy-demo -- cat /etc/resolv.conf`。

失败模式 / 回滚点：`dnsPolicy` 是 Pod 级不可变字段，改它要靠重建 Pod（Deployment 改模板触发滚动更新）；回滚就是把模板改回 `ClusterFirst`。`None` 配错时 Pod 会完全无法解析任何名字，包括集群内名字，业务容器可能起不来，务必先在探针 Pod 上验。

dnsPolicy	/etc/resolv.conf 形态	适用
`ClusterFirst`（默认）	`nameserver` 指向集群 DNS Service 的 ClusterIP（教学示例 `192.0.2.10`），`search` 为 `<ns>.svc.example.internal svc.example.internal example.internal`，`options ndots:5`	普通 Pod
`ClusterFirstWithHostNet`	与 `ClusterFirst` 相同	使用 `hostNetwork: true` 的 Pod（如 CNI 组件、节点级 agent）
`Default`	继承节点宿主机的 `/etc/resolv.conf`（可能是云厂商 VPC 内的解析器）	明确要走节点解析器、不查集群内名字的 Pod
`None`	不自动生成；必须配合 `dnsConfig` 手写 `nameserver` / `searches` / `options`	需要完全自定义的应用，或接 NodeLocal / 节点级 Unbound

两个容易踩的点：

其一，`hostNetwork: true` 的 Pod 若不显式写 `ClusterFirstWithHostNet`，kubelet 会按 `Default` 处理，容器直接读宿主机 resolv.conf，于是 `mysvc.default.svc.example.internal` 解析失败，而同节点其他 Pod 都正常。这是「只有某几个系统组件解析不了」的典型根因。

其二，`None` 不是「禁用 DNS」，而是「kubelet 完全放手」。写 `None` 却忘了 `dnsConfig`，容器里会是一份空的或继承镜像自带配置的 resolv.conf。

```yaml
# dnsPolicy: None 的完整写法（教学示例，指向节点级缓存地址）
spec:
  dnsPolicy: None
  dnsConfig:
    nameservers:
      - "192.0.2.10"       # NodeLocal DNSCache 的节点本地监听地址示例
    searches:
      - "default.svc.example.internal"
      - "svc.example.internal"
    options:
      - name: ndots
        value: "2"
      - name: edns0
```

为什么这么设：显式把 ndots 降到 2 并自己给全 searches，既保留集群内短名解析，又避免默认 ndots:5 带来的查询放大（20.4）。

怎么验证：kubectl exec <pod> -- cat /etc/resolv.conf 应只出现上面三个 search 与 ndots:2；kubectl exec <pod> -- getent hosts mysvc 成功。

失败模式 / 回滚点：searches 写漏 svc.example.internal 会让跨命名空间的短名失败；回滚把 dnsPolicy 改回 ClusterFirst 并删掉 dnsConfig。注意 dnsConfig 的 searches 最多 6 项（上限以官方 API 文档为准），不要照抄一份很长的域列表。

20.4 ndots:5 的查询放大，以及 search 域与绝对域名

ndots: n 的语义是：当查询名里的点数**少于 n** 时，存根解析器先按 search 列表逐项拼接，全部试过之后才把原始名字当绝对域名（FQDN）再查一次。默认 ndots:5 配上 3 个 search 域，意味着：

- mysvc（0 个点）→最多 3 次带 search 的查询 + 1 次原名 = **4 次**；
- api.dns.example.org（3 个点）→仍然先做 3 次 api.dns.example.org.default.svc.example.internal 之类的无效查询，再查原名 = **4 次**；
- 只有带**尾点**的绝对域名 api.dns.example.org.（4 个点且以点结尾表示 FQDN）才会跳过 search 直接查一次。

对外部域名，前 3 次查询会在集群内被判为 NXDOMAIN 并可能进入负缓存，然后才轮到真正那一次。这不是「多几个 UDP 包」这么简单：它把每个公网查询放大成 4 倍，直接压在 CoreDNS 与 conntrack 上，也是 20.5 里 UDP 竞态被触发的放大器。

```
# 观察放大：在 Pod 内抓本机出向 53
kubectl exec dns-policy-demo -- tcpdump -ni any -c 12 'port 53' &
kubectl exec dns-policy-demo -- getent hosts api.dns.example.org
```

为什么这么设：用最朴素的 getent 触发存根解析器行为，数一数实际发出的查询数，比读文档更直观。

怎么验证：抓包里应出现 4 个不同的 QNAME（3 个带 search 后缀 + 1 个原名）；把 ndots 改成 2 后，api.dns.example.org 只剩 1 个 QNAME。

失败模式 / 回滚点：降 ndots 到 2 之后，**集群内短名** mysvc（0 点）仍会走 search，没问题；但形如 mysvc.default（1 点）的写法会先被当成绝对域名查一次，可能拿到公网 NXDOMAIN 或慢一拍。回滚点是把该 Pod 的 dnsConfig.options 改回 ndots:5，或要求业务统一使用带尾点的 FQDN。

什么时候该降：

场景	建议
大量调用外部 API（3 个点以内的域名居多）	降到 2 或 3，或代码里统一写带尾点 FQDN
主要访问集群内服务，外部调用少	保持 5，收益不明显，改动反而有风险
混合型，且能改应用代码	优先改代码用 FQDN 带尾点，不动集群默认值
已上 NodeLocal 且 CoreDNS 压力仍高	降 ndots + 缩短 searches（只保留真实用到的）

「绝对域名」的正确写法是末尾那个点：dig api.dns.example.org. +short。在 Corefile 与 dnsConfig 里写域名时也应保持一致：CoreDNS 的 kubernetes 插件用 cluster.local 风格的 example.internal 作为集群域，写完整域名时带尾点可避免被 server block 的后缀匹配误吞。

20.5　NodeLocal DNSCache：大规模集群几乎必需

NodeLocal DNSCache 以 DaemonSet 方式在每个节点跑一个轻量缓存（通常用 CoreDNS），在节点本地监听一个地址，Pod 的 DNS 查询先到这里，缓存未命中才转发给集群 DNS Service。它解决的不是「缓存命中率」这种锦上添花的事，而是两个很硬的问题：

其一，绕开 conntrack 的 UDP 竞态。在 Linux 上，ClusterIP 到 Endpoint 的 DNAT 走 conntrack。UDP 五元组在并发查询下可能出现竞态：同一 Pod 几乎同时发出的两个查询（正是 `ndots:5` 放大出来的那种）被映射到同一条 conntrack 表项，或回包到达时表项已过期，表现为**随机 5 秒超时**、偶发 SERVFAIL。DNS 的默认超时与重试是 5s 量级，于是应用层的毛刺被放大成接口超时。NodeLocal 把查询终止在本机（`iptables/ipvs` 规则把节点本地地址直接交给本机进程，不经过 Service DNAT），这条链上的 conntrack 竞态就不存在了。

其二，减少对中心 DNS Service 的压力。每个节点吸收掉热查询与放大的 NXDOMAIN 查询，CoreDNS 副本看到的 QPS 从「Pod 数 × 4」降到「节点数 × 未命中率」。上百节点的集群里，这个差别决定了要不要给 CoreDNS 堆副本。

```yaml
# 教学骨架：NodeLocal 的关键参数（完整清单见官方 addon 清单）
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-local-dns
  namespace: kube-system
spec:
  template:
    spec:
      hostNetwork: true
      dnsPolicy: ClusterFirstWithHostNet   # 20.3：hostNetwork 必须显式写这一项
      containers:
        - name: node-cache
          image: registry.example.internal/dns/k8s-dns-node-cache:latest
          args:
            - "-localip"
            - "192.0.2.10,fd00:53::10"     # 节点本地监听（教学示例与文档段地址）
            - "-conf"
            - "/etc/Corefile"
            - "-upstreamsvc"
            - "kube-dns-upstream-service"  # 未命中时转发给该 Service
```

为什么这么设：`hostNetwork: true` 让节点本地地址不经过 Service 网络；`-localip` 同时给 IPv4 与 ULA IPv6 地址，与第 19 章的 ULA 用法一致；`-upstreamsvc` 指向一个**绕过自身**的上游 Service，避免节点缓存与中心 CoreDNS 形成环。

怎么验证：`kubectl -n kube-system get pods -l k8s-app=node-local-dns -o wide` 应每节点一个 Running；在节点上 `ss -lunp | grep 192.0.2.10` 能看到监听；`dig @192.0.2.10 mysvc.default.svc.example.internal +short` 返回 ClusterIP。

失败模式 / 回滚点：最典型的故障是**转发环**——NodeLocal 转发到中心 CoreDNS，中心 CoreDNS 又被规则引回节点本地地址。表现为所有查询 5 秒超时且 CoreDNS 日志里查询数暴涨。回滚点：保留原 `kube-dns` Service 的 ClusterIP 不动，通过修改节点上 NodeLocal 的 iptables 规则（`iptables-save > /var/tmp/iptables-before-nodelocal.txt` 快照）或直接 `kubectl delete ds node-local-dns` 恢复；删除后 Pod 的 resolv.conf 需要滚动更新才会回到中心 DNS，别指望即时生效。

第二个坑是 `dnsPolicy` 写成 `Default`（20.3）：DaemonSet 用了 `hostNetwork`，若漏掉 `ClusterFirstWithHostNet`，node-cache 容器自己解析 `kube-dns-upstream-service` 就会去问宿主机解析器，直接失败。

20.6　一份可复制的 Corefile

下面这份 Corefile 用于**中心 CoreDNS**（服务发现 + 转发给外部验证器），不是 NodeLocal 的那份。

```
# CoreDNS ConfigMap（教学示例，集群域用 example.internal）
.:53 {
    errors
    health {
        lameduck 5s
```

```
    }
    ready
    log
    kubernetes example.internal in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
        ttl 30
    }
    prometheus :9153
    cache {
        success 9984 30
        denial 9984 5
    }
    rewrite stop {
        name regex (.*)\.gw\.example\.internal dns.example.org
        answer name dns.example.org {1}.gw.example.internal
    }
    forward . 192.0.2.53 {
        policy sequential
        expire 10s
        max_fails 1
        health_check 5s
    }
}
```

为什么这么设：把 kubernetes 放在链首保证集群内名字不出集群；fallthrough 只放行反向域，不把整个集群域漏给公网；cache 的 denial TTL 压到 5s 防负缓存放大；forward 单上游且 max_fails 1 + health_check 5s 快速摘除，避免把查询挂在黑掉的验证器上。注意 forward 到 Unbound 走的是明文 53（集群内网络），跨节点的加密传输、fail-closed 与 DoT 上游由 Unbound 侧负责（第 11、14 章），不要指望 CoreDNS 的 forward 提供 TLS。

失败模式 / 回滚点：rewrite 规则写错会把集群内名字改写到公网，cache 放到 kubernetes 之前会让 EndpointSlice 变化滞后；回滚点是 kubectl -n kube-system get cm coredns -o yaml > /var/tmp/coredns-cm-before.yaml 快照 + kubectl apply -f 还原并 rollout restart；reload 插件可热加载 Corefile，但插件增删仍建议滚动重启。

怎么验证（一组可复制命令）：

```
kubectl -n kube-system get configmap coredns -o yaml | grep -A40 'Corefile'
kubectl -n kube-system rollout status deployment/coredns --timeout=90s
kubectl -n kube-system rollout restart deployment/coredns
kubectl -n kube-system rollout status deployment/coredns --timeout=90s

# 集群内名字走 kubernetes 插件
kubectl run -it --rm dnsprobe --image=registry.example.internal/net-tools:latest \
  --restart=Never -- dig +short mysvc.default.svc.example.internal A

# 外部名字走 forward → Unbound；带尾点避开 search 放大
kubectl run -it --rm dnsprobe --image=registry.example.internal/net-tools:latest \
  --restart=Never -- dig +short api.dns.example.org. A

# 负缓存只留 5 秒：连查两次不存在的名字，比较时间
kubectl run -it --rm dnsprobe --image=registry.example.internal/net-tools:latest \
  --restart=Never -- sh -c 'dig +short no-such.dns.example.org.; sleep 6; dig +short no-such.dns.example.org.'

# 指标与上游健康
kubectl -n kube-system port-forward deploy/coredns 9153:9153 &
curl -s 127.0.0.1:9153/metrics | grep -E 'coredns_(cache|forward)_' | head -20
```

为什么这么设（验证命令）：先确认 ConfigMap 已生效与副本就绪，再分别从 kubernetes 插件、forward 插件、负缓存、指标四条路径核对，任一条不对都能定位到具体插件而不是整份 Corefile。

怎么验证(验证命令):kubectl -n kube-system rollout status deployment/coredns --timeout=90s 显示 successfully rolled out；三条 dig 分别返回 ClusterIP、公网地址、以及 6 秒后重新发起的查询。

失败模式 / 回滚点（验证命令）：kubectl run -it --rm dnsprobe 在集群里没有 RBAC 或镜像拉不到时会失败，此时

改用已有 Pod 内 `exec` 执行 `dig`；`port-forward` 未建立时 `curl` 会连不上 9153，先确认 CoreDNS 副本就绪。Corefile 本身的失败模式（`rewrite` 写错把集群内名字改写到公网、`cache` 放到 `kubernetes` 之前导致 EndpointSlice 滞后 30s）与回滚点（`kubectl -n kube-system get cm coredns -o yaml > /var/tmp/coredns-cm-before.yaml` 快照 + `rollout restart`）见 20.6 开头的那段；`reload` 插件可热加载 Corefile，但插件增删仍建议走一次滚动重启。

逐项说明这条插件链：

插件	作用	与本书前文的呼应
`errors` / `log`	错误与查询日志	对应第 7 章可观测性；生产环境 `log` 要限量，避免 dnstap 式的元数据自留问题
`health` / `ready`	存活与就绪探针（`:8080` / `:8181`）	滚动更新时不把未就绪副本挂进 Service
`kubernetes`	唯一的服务发现来源	见 20.1，Unbound 不能替代
`prometheus :9153`	指标导出	结合 `coredns_cache_*`、`coredns_forward_*` 看命中率与上游时延
`cache`	`success 9984 30`（成功缓存 30s）、`denial 9984 5`（NXDOMAIN 只缓存 5s）	**负缓存**故意压到 5 秒，对应第 22 章案例⑤：上游瞬时 NXDOMAIN 不应在集群里续命 3600s
`rewrite`	把内部别名改写到外部域名，并改写应答里的名字	只做名字整形，不做验证
`forward`	把所有集群外名字交给 Unbound 验证器（教学示例 `192.0.2.53`）	单个上游、fail-closed：见下文

`forward` 这里只给一**个目标地址**（教学示例 `192.0.2.53`；双栈部署由该地址对应的服务同时提供 v4/v6 可达性，不列第二个目标）。`policy sequential` 在只有一个目标时等价于固定目标。这与第 5 章的教训完全一致：给转发器配第二个 `forward-addr`，Unbound 是「并发竞速取最快」而非主备，会把查询送到不该去的解析器，过滤表失效。CoreDNS 的 `forward` 也有多上游与 `policy random` / `round_robin` 等选项，在「上游即策略边界」的场景里同样不该开竞速。要做切换，用外部巡检脚本改 ConfigMap，不要靠多上游。

20.6.1　stubDomains 与 upstreamNameservers 的历史写法

在 kube-dns 时代，`kube-dns` 的 ConfigMap 支持两个字段：

```yaml
# 历史写法（kube-dns ConfigMap，仅作对照，勿在新集群使用）
apiVersion: v1
kind: ConfigMap
metadata:
  name: kube-dns
  namespace: kube-system
data:
  stubDomains: |
    {"corp.example.internal": ["192.0.2.54"]}
  upstreamNameservers: |
    ["192.0.2.53", "2001:db8::53"]
```

为什么这么设（仅作历史对照）：这两个字段是 kube-dns 时代把「特定域走指定解析器」与「其余走上游」塞进一个 ConfigMap 的办法，列出它是为了给出迁移映射，不是建议在新集群使用。

怎么验证：`kubectl -n kube-system get cm coredns -o yaml | grep -E 'stubDomains|upstreamNameservers' || true` 应无输出，说明集群已走 CoreDNS 的 server block 写法。

失败模式 / 回滚点：迁移时若把 `upstreamNameservers` 的多个地址照抄成 CoreDNS `forward` 的多个目标，会重新引入案例①的竞速；回滚点同样是 `kubectl -n kube-system get cm coredns -o yaml > /var/tmp/coredns-cm-before.yaml` 快照。

它们分别表达「某些域名走指定解析器」和「其余域名走这些上游」。CoreDNS 里没有这两个字段，对应关系是：

kube-dns 字段	CoreDNS 对应写法
`stubDomains: {"corp...": ["192.0.2.54"]}`	另起一个 server block: `corp.example.internal:53 { forward . 192.0.2.54 ; cache ; errors }`
`upstreamNameservers: [...]`	根 server block 里的 `forward . < 地址 > ...`
两者的优先级	CoreDNS 按 **server block 的最长后缀匹配**选择，天然实现「更具体的域先命中」

```
# CoreDNS 里 stubDomains 的等价物
corp.example.internal:53 {
    errors
    cache 30
    forward . 192.0.2.54
}
```

为什么这么设：用独立的 server block 而不是在根 block 里堆 `rewrite`，可以让 `log` 与 `prometheus` 按域分别统计，也避免 rewrite 规则互相干扰。

怎么验证：`kubectl run -it --rm dnsprobe --image=registry.example.internal/net-tools:latest --restart=Never -- dig +short host.corp.example.internal A`，并在 `192.0.2.54` 侧确认收到了查询。

失败模式 / 回滚点：server block 的域名必须与 `kubernetes` 插件的集群域不重叠，否则服务发现被遮蔽；写 `corp.example.internal:53` 时不要漏 `:53`。回滚同样是还原 ConfigMap 快照。另一个历史坑：`upstreamNameservers` 里列多个地址会被当成竞速/随机，迁移到 CoreDNS 时**不要**照抄成多个 `forward` 目标，按 20.1 的分工只留一个验证器入口。

20.7 三种拓扑与取舍

分工确定之后，Unbound 放在链路的哪一层仍有三个可行位置。下表所有地址都是教学示例。

拓扑	结构	优点	代价 / 取舍
A. NodeLocal + Unbound 上游	Pod → NodeLocal(节点本机) → CoreDNS(服务发现) → Unbound(集群内 Service 或集群外) →权威	改动最小，复用现有 DNS Service；conntrack 竞态在节点侧已消除	Unbound 仍是共享入口，故障域覆盖全集群；DNSSEC 验证集中在少数副本
B. 每节点 Unbound sidecar	Pod →节点级 Unbound(本机:53) → CoreDNS(仅集群域名，另一 server block) →加密上游	验证/过滤/QMIN/serve-stale 全部节点本地；单点故障域缩到一个节点；可做每节点缓存	DaemonSet 数量随节点增长，信任锚与配置分发、版本一致性、内存开销都要自己管
C. 集群外中央验证器 + DoT 上游	Pod → NodeLocal → CoreDNS →集群边界的 `cloud-01` Unbound → DoT/DoQ 到外部→权威	配置集中、审计集中；DoT 上游与第 11、14 章一致；跨集群复用	跨节点/跨机房多一跳；链路被 QoS 时 UDP 不稳，需要按第 22 章案例⑩优先 TCP 承载

三者不互斥：常见落地是 **A 起步、C 收口**——先在节点上消除 conntrack 竞态，再把 CoreDNS 的 `forward` 指到集群外那台做验证与 DoT 的 Unbound；B 适合对隐私与故障域有更严要求、且已有配置分发体系的团队。第 21 章的综合实验拓扑本质上是 C 的缩小版。

无论哪一种，落到 Unbound 侧的控制项应与第三篇逐条对齐：

控制项	放在哪一层	要点
DNSSEC 验证	**Unbound**（验证器），或集群外中央验证器	CoreDNS 不做验证；有信任锚才会对坏签名域返回 SERVFAIL；不要把 `auto-trust-anchor-file` 配在纯转发器上（第 22 章案例④）

控制项	放在哪一层	要点
DoT / DoH / DoQ 上游	Unbound 出向	加密传输与 DNSSEC 是两件事，不互相替代；`forward-tls-upstream: yes` 一类配置见第 11 章
查询名最小化（QMIN）	Unbound 出向	别开 `qname-minimisation-strict: yes`，违法 [RFC 8020] 的 CDN 区会整片解析失败，且每层缓存都要刷（第 13、22 章）
EDNS 1232	全链路	`edns-buffer-size: 1232`（官方默认，DNS Flag Day 2020）；**出向**靠它，`max-udp-size` 只管入向。节点网络、隧道 MTU 1280 下尤其重要（第 22 章案例③）
负缓存	每层都要单独设	CoreDNS `cache { denial ... 5 }`、NodeLocal 的 cache、Unbound `cache-max-negative-ttl` 三层各自有默认值；上游瞬时 NXDOMAIN 会被按 SOA minimum 缓存，**链路上每个实例都要刷**，见第 22 章案例⑤
过期缓存应答（serve-stale）	Unbound	`serve-expired-client-timeout` 的单位是**毫秒**（默认 1800 = 1.8 秒），不是 1800 秒；CoreDNS 的 `cache` 没有等价的 serve-stale 语义，别误以为开了 `cache` 就等于有了抗上游抖动能力

特别提醒「每层缓存都要刷」在 Kubernetes 里比裸机更麻烦：一次错误的 NXDOMAIN 可能同时躺在 NodeLocal、中心 CoreDNS、Unbound、以及应用自身的解析器里。刷缓存时要按 `NodeLocal → CoreDNS → Unbound` 的顺序逐层来，只刷最后一层会看到「明明修好了却还在失败」。

```
# 逐层刷缓存（教学示例；具体命令以各组件实际启用方式为准）
kubectl -n kube-system rollout restart ds/node-local-dns
kubectl -n kube-system rollout restart deploy/coredns
unbound-control -c /etc/unbound/unbound.conf flush_negative
unbound-control -c /etc/unbound/unbound.conf flush_zone dns.example.org
```

为什么这么设：从离客户端最近的一层开始清，避免清完最外层后，近端缓存又把旧答案吐出来。

怎么验证：`dig +short no-such.dns.example.org.` 在每一层分别 `@192.0.2.10`（NodeLocal）、`@<CoreDNS-ClusterIP>`、`@192.0.2.53`（Unbound）直到结果一致。

失败模式 / 回滚点：`unbound-control` 没启用远程控制时只能重启实例刷缓存（第 22 章案例⑤）；`rollout restart` 会让该层短暂降级，滚动窗口内要保证 `serve-expired` 开着。回滚点同上：保留 ConfigMap 与 Unbound 配置快照。

20.8　验收清单

检查	命令	通过标准
dnsPolicy 生效	`kubectl exec <pod> -- cat /etc/resolv.conf`	与 20.3 表格一致；hostNetwork Pod 是 `ClusterFirstWithHostNet`
集群内服务发现	`dig +short mysvc.default.svc.example.internal A`	返回 Service ClusterIP
外部名字分流	`dig +short api.dns.example.org. A`	由 Unbound 递归得到答案，CoreDNS `forward` 计数上升
search 放大已消除	抓 `port 53` 数 QNAME	带尾点域名只剩 1 个 QNAME

检查	命令	通过标准
负缓存时长	CoreDNS `/metrics` 的 `coredns_cache_*` + 两次连查对比	denial 缓存不超过 5s
DNSSEC 在正确的层	`dig +dnssec badsig.dns.example.org. A`	返回 SERVFAIL（验证器侧真验证时）
EDNS 尺寸	`dig +bufsize=4096 +dnssec @192.0.2.53 dns.example.org DNSKEY`	不出现分片丢弃后的 TCP 重试
conntrack 竞态	`conntrack -S | grep -i -E 'insert_failed|drop'`	上 NodeLocal 前后对比，随机 5s 超时消失

最后再强调一次选型结论：**别把 Unbound 塞进 `kube-system` 去替换 CoreDNS**。它该出现在 `forward` 的另一端、节点本地的 sidecar 里、或者集群边界那台跑 DoT 的验证器上。集群内服务发现只有一个数据源，就是 API Server；公网递归与验证只有一个负责人，就是 Unbound。第 21 章会把这个分工和第 19 章的网关引导、第 3 篇的加固控制项拼成一台网关加一台云主机的完整实验。

20.9 本章要点

1. **分工**：CoreDNS 负责集群内服务发现（数据源是 API Server），Unbound 负责外部递归 / DNSSEC 验证 / 过滤 / 加密上游；二者不是替代关系，Unbound 不替换 `kubernetes` 插件。
2. **四种 dnsPolicy**：`ClusterFirst`（默认）、`ClusterFirstWithHostNet`（`hostNetwork: true` 必须显式写）、`Default`（继承节点）、`None`（必须配 `dnsConfig`）；`None` 不是禁用 DNS。
3. **`ndots:5` 把 3 点以内的外部域名放大成最多 4 次查询**；外部调用密集的 Pod 降到 2–3，或统一使用带尾点的绝对域名，同时缩短 `searches`。
4. **NodeLocal DNSCache** 在大规模集群几乎必需：节点本地终止查询以绕开 ClusterIP DNAT 的 conntrack UDP 竞态（随机 5s 超时），并大幅削减中心 DNS 的 QPS；注意 `hostNetwork` 与避免转发环。
5. **Corefile 是一条插件链**，顺序改变语义；`cache` 的 denial TTL 要压到秒级，`forward` 只留一个上游（竞速会破坏策略边界），`kubernetes` 放链首。
6. **第三篇的控制项落在 Unbound 那一层**：DNSSEC 验证、DoT 上游、QMIN（不开 strict）、`edns-buffer-size: 1232`、负缓存与 serve-stale；且**每层缓存都要单独刷**、单独设。

20.10 参考资料

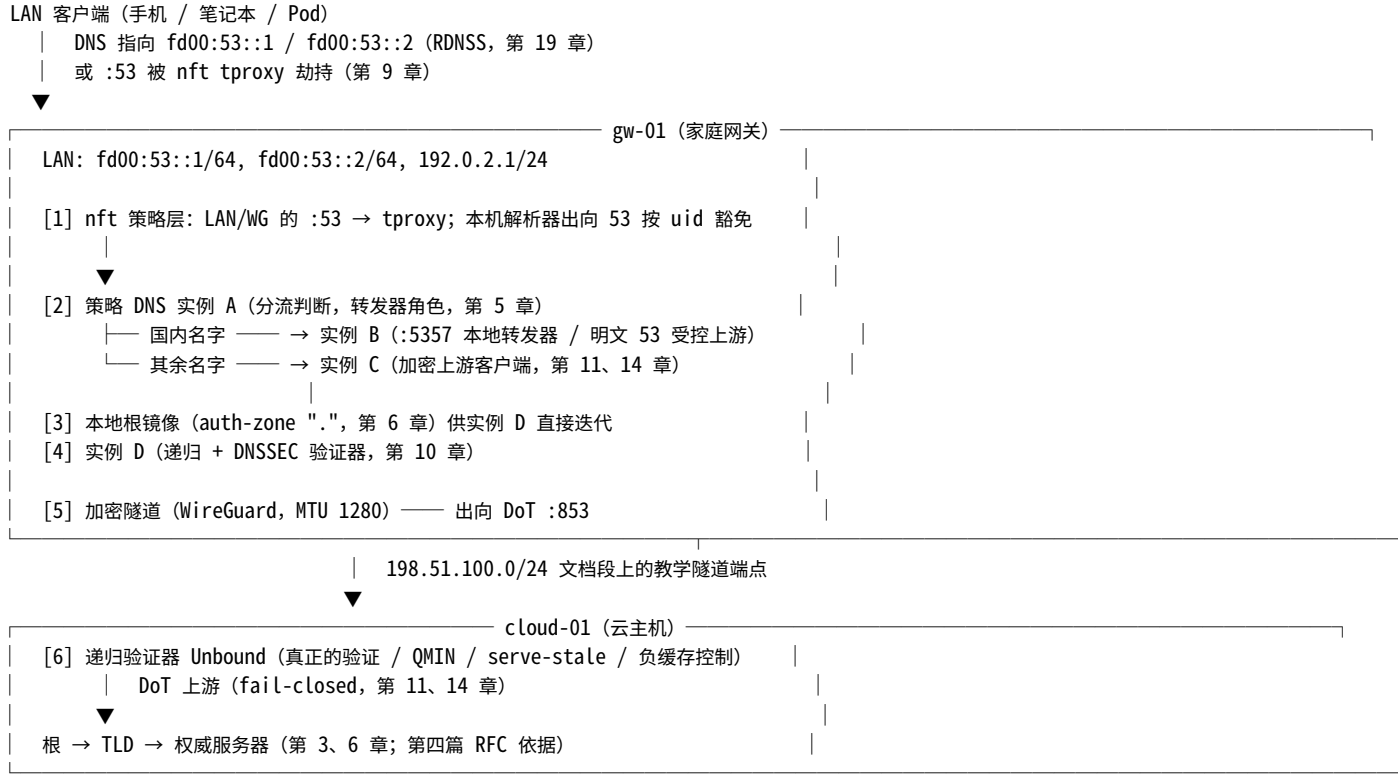
- Kubernetes 官方文档 *DNS for Services and Pods*: https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/
- Kubernetes 官方文档 *Using NodeLocal DNSCache in Kubernetes clusters*: https://kubernetes.io/docs/tasks/administer-cluster/nodelocaldns/
- Kubernetes 官方文档 *Customizing DNS Service*（stubDomains / upstreamNameservers 迁移说明）: https://kubernetes.io/docs/tasks/administer-cluster/dns-custom-nameservers/
- CoreDNS 官方站点与插件文档: https://coredns.io/plugins/
- CoreDNS `forward` 插件文档: https://coredns.io/plugins/forward/
- Kubernetes 官方博客 *CoreDNS for Kubernetes Service Discovery*: https://kubernetes.io/blog/2018/07/10/coredns-ga-for-kubernetes-service-discovery/
- Unbound 配置手册: https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html
- RFC 8020, *NXDOMAIN: There Really Is Nothing Underneath*: https://www.rfc-editor.org/rfc/rfc8020

第二十一章

综合实验：一台网关 ＋ 一台云主机

21.1 拓扑总览：把前四篇拼成一条链

这台实验环境只由两台机器构成：家里的一台网关 gw-01，以及一台境外云主机 cloud-01。它把第一篇的角色划分、第二篇的多实例配置、第三篇的加固控制项、第四篇的 RFC 依据串成一条可复现的查询路径。



为什么这么设：把每个控制项钉在唯一一个组件上（引导归 RA、分流归 A、验证归 cloud-01），任何一次故障都能按 21.5 的五个观测点切成互不重叠的一段，而不是在两台机器之间来回猜。

怎么验证：`ss -lunp -t | grep -E ':(53|853|5357|5358|5359)\b'` 的输出应与 21.4 监听表逐行一致。

失败模式 / 回滚点：把验证与转发堆到同一个实例会重现第 22 章案例④；回滚点见 21.7 的分层快照表，顺序先救 DNS 可达性再救策略。

先明确各角色的边界，避免把两台机器上的实例混成一个「大 Unbound」：

组件	角色（第一篇术语）	唯一职责	不能做什么
实例 A	转发器	按域名/标签分流，决定往哪条链路走	不做验证、不配信任锚（第 22 章案例④）
实例 B	转发器	国内名字的单上游受控转发	不配第二个上游（第 22 章案例①）
实例 C	转发器	加密上游客户端，fail-closed	不允许静默明文降级（第 22 章案例②）
实例 D	递归解析器 + 验证器	本地根镜像迭代 + DNSSEC 验证	与 A 的转发器身份二选一
`cloud-01` Unbound	递归解析器 + 验证器	出向递归、DoT 上游、QMIN、serve-stale	不负责 LAN 的 DNS 引导

「策略 DNS」在本章特指实例 A 做的那件事：**按查询名与来源标签把查询分给不同的下游实例**，它不是 policy routing 意义上的路由表操作，也不解释应答内容。分流规则、监听表、失败注入点见 21.3–21.5。

与第 20 章的关系也顺带说清：如果客户端是集群里的 Pod，把上图的「实例 A + CoreDNS」换成一个组合——CoreDNS 只答 `svc.example.internal`，其余 `forward` 到本图的 `cloud-01`。第 20 章的三种拓扑在这里对应「集群外中央验证器 + DoT 上游」那种。

21.2　客户端 DNS 指向：引导层只做一件事

客户端侧不需要逐台配置。第 19 章已经把机制写透：网关自己发 RA，在 RDNSS 选项里广告自己的 ULA 地址，并用 `RDNSS lifetime=0` 压掉光猫广告的 DNS。**光猫里不配任何 DNS**。

```
LAN 客户端最终应看到的 DNS:
  fd00:53::1        # 实例 A 的监听地址
  fd00:53::2        # 备用/第二个实例，仅当确实部署了第二个实例
  （不应出现第三条，尤其不应出现光猫的链路本地地址或 GUA）
```

为什么这么设：引导路径上只放 ULA（`fd00:53::/48`，第 19 章），不出现会随运营商轮换的 GUA，也不留给光猫广告的 DNS 任何位置。

怎么验证：`adb shell dumpsys connectivity | grep -i -A3 DnsAddresses`，期望只列出这两条。

失败模式 / 回滚点：出现第三条说明 RDNSS 的 `lifetime=0` 撤销未被接受，回到第 19 章的退路；回滚用 `/etc/ra-steer/ra.conf.bak`。

对不理 RA 的存根解析器（写死:53 的物联网设备、某些容器），用第 9 章的 nftables :53 劫持兜底。但必须记住第 22 章案例⑥那条边界：**本机解析器进程（unbound、nsd）的出向 53 要按 uid 豁免**，否则自己的递归查询被自己的 hijack-dns 吃掉，表现为根区镜像永远不更新。

验收（客户端侧，命令可直接复制）：

```
# Android
adb shell dumpsys connectivity | grep -i -A3 DnsAddresses
# Linux
resolvectl dns lan0 2>/dev/null || cat /etc/resolv.conf
# 抓 RA 核对 hop limit 255 / SLLAO / RIO / RDNSS
tcpdump -i lan0 -n -vv 'icmp6 && ip6[40] == 134' -c 2
# 走通一次查询
dig @fd00:53::1 +short dns.example.org A
```

为什么这么设：把「引导正确性」和「解析正确性」分成两次验收，前者只看客户端的 DNS 列表与 RA 报文，后者才看应答。

怎么验证：`adb shell dumpsys connectivity | grep -i -A3 DnsAddresses` 只列出自己的两条 ULA；`dig @fd00:53::1 +short dns.example.org A` 返回答案。

失败模式 / 回滚点：出现第三条 DNS 说明 RDNSS 撤销没被接受，回到第 19 章的退路；`dig` 超时但列表正确，说明问题在网关内部（跳到 21.3 的 nft 计数器），不要在客户端上反复改 `/etc/resolv.conf`。回滚点：停掉 RA 发送进程，用第 19 章的 `/etc/ra-steer/ra.conf.bak` 恢复。

21.3 分流规则：策略 DNS 只留一个出口

分流表是拓扑级的拼装，具体 unbound.conf 的完整配置见第 5、7 章，这里不重复贴配置块。

判定顺序	匹配条件	交给	依据
1	来源 uid 是本机解析器（unbound / nsd）	直接放行出向 53，不进 tproxy	第 22 章案例⑥
2	查询名命中本地权威区（example.internal、_dns.resolver.arpa.）	本地应答 / always_nodata	第 13 章 DDR 压制
3	命中「国内名字」清单	实例 B（单上游受控转发器）	第 5 章单上游竞速教训
4	其余全部	实例 C →加密隧道→ cloud-01	第 11、14 章 fail-closed
5	以上任一步 SERVFAIL	返回 SERVFAIL，不回退明文	第 22 章案例②

三条拓扑级纪律：

- **每个转发器只留一个上游**。给转发器配第二个 forward-addr，Unbound 是「并发竞速取最快」而非主备，查询会被送到不该去的解析器，过滤表失效（第 22 章案例①）。切换交给外部巡检脚本改配置 + reload。
- **fail-closed 不是一句口号**。 forward-first: no、加密上游失败后两波都失败就 SERVFAIL；允许回退到明文 53 会让整条查询流在无人察觉的情况下裸奔（第 22 章案例②）。
- **分流清单本身要能回滚**。清单用单独文件（例如 /etc/unbound/policy-domains.list）加载，改动前 cp -a 一份，回滚只还原这一个文件，不动实例配置。

```
# 拓扑级分流可视性：看每个实例各自收到的查询数
unbound-control -c /etc/unbound/instance-a.conf stats_noreset | grep -E 'num.query|num.cache'
unbound-control -c /etc/unbound/instance-b.conf stats_noreset | grep -E 'num.query|num.cache'
unbound-control -c /etc/unbound/instance-c.conf stats_noreset | grep -E 'num.query|num.cache'
```

为什么这么设：三个实例的查询计数能直接证明「分流是否真的按表走」，比在日志里 grep 域名更可靠，也不会把查询内容写进日志（呼应第 7 章 dnstap 保留期那条）。

怎么验证：连续查一个国内名字和一个外部名字，A 的 num.query 增 2，B 与 C 各增 1。

失败模式 / 回滚点：若 A 增 2 而 B、C 都是 0，多半是 A 自己缓存了答案或 tproxy 没把包交进来；按 21.6 的顺序先看 nftables 计数器。回滚点：/etc/unbound/policy-domains.list.bak 与三个实例的配置快照。

21.4 各实例监听表

监听表是整个拓扑最容易写错的一张表：地址重叠、端口撞车、把 ULA 绑到会轮换的 GUA 上，都会造成「部分客户端能用、部分不能」。以下只列拓扑级的地址与端口分工，**完整的 server: / forward-zone: 配置见第 5、6、7 章**。

实例	监听地址	端口	角色	上游 / 出向
A 策略 DNS	fd00:53::1、192.0.2.1	53	转发器（分流）	B / C
A’ 备用引导	fd00:53::2	53	转发器（同 A 配置，仅地址不同）	B / C
B 国内分流	127.0.0.1、::1	5357	转发器	单上游，受控明文 53
C 加密客户端	127.0.0.1、::1	5358	转发器	WG 隧道→ cloud-01 DoT :853
D 本地递归	127.0.0.1、::1	5359	递归解析器 + 验证器	本地根镜像→权威（第 6 章）
cloud-01 Unbound	教学隧道地址 198.51.100.2、ULA fd00:53::53	853 (DoT) / 53 (仅隧道内)	递归解析器 + 验证器	外部 DoT 上游→根/TLD/权威

三点约束：

1. **RDNSS 里只出现 ULA**（fd00:53::1 / fd00:53::2），不出现 2001:db8::/32 这类会轮换的 GUA，见第 19 章。
2. **B / C / D 只监听 loopback**，不暴露到 LAN；LAN 上唯一的:53 入口是 A（加上 nft tproxy 的兜底）。
3. **cloud-01 的 53 只在隧道内可达**，公网侧只留:853；否则它就成了一台开放解析器（第 9 章）。

```
# 监听表核对：一次列出本机所有 53/853/535x 监听
ss -lunp -t | grep -E ':(53|853|5357|5358|5359)\b' || true
# ULA 是否真的绑在 LAN 口（不是 GUA）
ip -6 addr show dev lan0 scope global
```

为什么这么设：用一张命令就能核对「谁在哪个地址上听」，避免逐个实例翻配置文件；ip -6 addr 同时验证 ULA 与 GUA 的分工。

怎么验证：ss -lunp 输出与上表逐行一致；ip -6 addr show dev lan0 里 fd00:53::1/64 存在。

失败模式 / 回滚点：两个实例绑同一 127.0.0.1:5357 时后启动者失败，日志里是 bind: Address already in use，现象却是「分流一半失效」。回滚点：四个实例的配置文件快照 + systemctl stop 顺序表；改端口前先 ss -lunp 确认没有残留进程。

21.5　失败注入与观测点

「可复现」的关键不是配置写得多全，而是能**主动把某一层打断，并看到预期中的降级行为**。下面这张表就是本实验的注入矩阵，每项都对应前面某一章的结论。

#	注入动作	预期可观测结果	关联章节 / 案例
1	停掉 cloud-01 的 Unbound	C 侧 DoT 失败→ A 返回 SERVFAIL；**不出现明文 53 出向**	第 5、14 章；案例②
2	上游权威短暂返回 NXDOMAIN 后恢复	应答在 SOA minimum 期内仍是 NXDOMAIN；逐层刷缓存后恢复	第 22 章案例⑤
3	把某条路径 MTU 改到 1280 以下	edns-buffer-size: 1232 仍不分片；若误设 1472 则出现超时→ TCP 重试	第 13、17 章；案例③
4	断开 cloud-01 出向，但保留已缓存记录	serve-expired 返回过期缓存应答，客户端仍有答案	第 7 章；注意 serve-expired-client-timeout 单位是毫秒
5	打开 qname-minimisation-strict: yes	违反 [RFC 8020] 的 CDN 区 CNAME 整片解析失败；关掉后仍要刷每层缓存	第 13 章；案例③
6	在 A 上加第二个 forward-addr	查询开始出现在不该去的解析器上，过滤表失效	第 5 章；案例①
7	去掉 nft 的 uid 豁免	根区镜像停止更新、AXFR 失败、递归查询被自己劫持	第 9 章；案例⑥
8	在 output-tproxy 链尾打标不排除 WG	WG 握手响应到不了对端，隧道起不来	第 14 章；案例⑦

注入点的物理位置也要钉死，否则「断了但没断对」会浪费大量时间：

```
观测点 O1：LAN 客户端 → A 之间（tcpdump，看 DNS 列表与 QNAME）
观测点 O2：A → B/C 之间（各实例 stats / dnstap，看分流）
观测点 O3：C → WG 隧道口（tcpdump + nft 计数器，看 mark 与加密封装）
观测点 O4：cloud-01 隧道口（tcpdump :853，看 DoT 与 EDNS 尺寸）
观测点 O5：cloud-01 → 权威之间（dig +dnssec、QMIN 层数、根区 priming）
```

为什么这么设：五个观测点把链路切成互不重叠的四段，任何一次故障都能用「哪一段的计数器不动了」定位，而不是靠猜。

怎么验证：注入 #1 时，O3 的隧道计数器仍在（C 在重试），O4 无:853 新流，O1 收到 SERVFAIL；三者齐备才说明 fail-closed 真的生效。

失败模式 / 回滚点：注入测试会污染缓存，测完必须按 #2 的顺序逐层刷；所有注入前先 `nft list ruleset > /var/tmp/nft-before-lab.txt`、`cp -a /etc/unbound /var/tmp/unbound-before-lab/`。**排查顺序：先看 nftables，再怀疑别的**——注入 #7、#8 的现象都极易被误判成上游故障。

21.6 验收命令清单

以下命令按「由近及远、由数据面到语义」排列，全部使用文档段地址与教学域名，可直接复制。它们只检查拓扑是否拼装正确，不重复前面章节的完整配置。

21.6.1 1) DNSSEC 与信任锚

```
# 好签名域：应答应带 ad 标志（验证器真验证时）
dig @fd00:53::1 +dnssec signed.dns.example.org A
# 坏签名域：应返回 SERVFAIL，而不是照样给答案
dig @fd00:53::1 +dnssec badsig.dns.example.org A
# 验证器侧确认信任锚已 priming（不应有 failed to prime 日志）
unbound-control -c /etc/unbound/instance-d.conf status | grep -i anchor || true
journalctl -u unbound@instance-d --since '10 min ago' | grep -i 'prime\|bogus' || true
```

为什么这么设：第 10 章的三种状态（无锚透传 / permissive 记日志放行 / 真验证）用一好一坏两个域就能一次判完；`dig +dnssec` 的 `ad` 标志是「这一跳真的验证了」的证据，而 SERVFAIL 是坏签名被拒的证据。

怎么验证：好签名域 `flags:` 含 `ad`；坏签名域 `status: SERVFAIL`。

失败模式 / 回滚点：坏签名域返回答案说明信任锚没生效或处于 permissive；**不要在纯转发器（实例 A）上配 `auto-trust-anchor-file`**——信任锚需要自己向根 priming DNSKEY，而 `forward-zone "."` 把一切交给上游，只会得到持续的 `failed to prime trust anchor` 与间歇性全量 SERVFAIL（第 22 章案例④）。回滚点：`/var/tmp/unbound-before-lab/` 快照。

21.6.2 2) 坏 MTU 路径

```
# 在隧道口把 MTU 压到 1280（IPv6 最小 MTU / WireGuard 典型值）
ip link set dev wg0 mtu 1280
# 用大应答触发分片判定；1232 应一次过，1472 会超时后重试 TCP
dig @fd00:53::1 +bufsize=4096 +dnssec dns.example.org DNSKEY | tail -5
# 观察是否出现 TCP 回退
tcpdump -ni wg0 -c 20 'port 53 or port 853' | grep -i -E 'tcp|frag' || true
# 出向通告尺寸：edns-buffer-size（官方默认 1232），max-udp-size 只管入向
unbound-control -c /etc/unbound/instance-d.conf get_option edns-buffer-size
```

为什么这么设：把 MTU 与 EDNS 尺寸放在同一组命令里，能直接区分「分片被丢」和「上游超时」这两种长得一模一样的现象。`edns-buffer-size: 1232` 是 DNS Flag Day 2020 的官方默认，1472 在 MTU 1280 与 PPPoE 1492 上都会分片。

怎么验证：`unbound-control ... get_option edns-buffer-size` 显示 1232（若该 `get_option` 不受支持，直接查配置文件）；抓包里无 IPv6 分片头、无 53/TCP 重试。

失败模式 / 回滚点：误设 1472 后不要靠调 `max-udp-size` 补救——它只管入向，出向要靠 `edns-buffer-size`。回滚：`ip link set dev wg0 mtu < 原值 >`，原值存在 `/var/tmp/nft-before-lab.txt` 旁边的 `ip link show > /var/tmp/link-before-lab.txt` 快照里。

21.6.3 3) 上游断开时的过期缓存应答

```
# 先预热一个记录
dig @fd00:53::1 +short api.dns.example.org A
# 模拟上游断开（教学：在 cloud-01 侧临时停服务，不真的改公网）
ssh cloud-01 'systemctl stop unbound-upstream' || true
# 立刻再查：应仍得到答案（serve-expired），但可能带过期标记
dig @fd00:53::1 +short api.dns.example.org A
```

```
# 注意单位：serve-expired-client-timeout 默认 1800 = 1.8 秒（毫秒），不是 1800 秒
unbound-control -c /etc/unbound/instance-d.conf get_option serve-expired-client-timeout || true
ssh cloud-01 'systemctl start unbound-upstream' || true
```

为什么这么设：把「上游断开」和「应答仍在」放在同一个 shell 片段里，直接证明 serve-stale 生效；同时把那个最容易读错的单位写在命令旁边。

怎么验证：第二次 `dig` 仍返回地址；`journalctl` 里能看到 `serve expired` 一类记录。

失败模式 / 回滚点：若第二次 `dig` 变成 SERVFAIL，检查 `serve-expired: yes` 与 `serve-expired-ttl` 是否被上层（A/C 的 `cache`）提前截断——**链路上每一层的缓存策略都要一致**。回滚：恢复 `cloud-01` 服务并逐层刷缓存。

21.6.4　4) nft 计数器与劫持边界

```
# 先看规则与计数器（排查顺序：先看 nftables，再怀疑别的）
nft list ruleset | sed -n '1,120p'
nft list chain inet mangle hijack-dns 2>/dev/null || true
nft list chain inet mangle output-tproxy 2>/dev/null || true
# 清零点后发一次查询，看每条规则的 pkts 增量
nft reset counters
dig @fd00:53::1 +short dns.example.org A
nft list ruleset | grep -i 'counter\|packets' | head -20
# uid 豁免是否命中（本机解析器出向 53 不应被 tproxy）
nft list ruleset | grep -i 'skuid' | head -10
```

为什么这么设：计数器把「包到底走了哪条链」变成可数的数字，比看日志里的现象更直接；`skuid` 一行专门验证案例⑥的豁免，链尾 `meta mark set 0x2332` 一行专门验证案例⑦的 WG 回包是否被误打标。

怎么验证：LAN 查询使 hijack-dns 计数增加；本机解析器出向 53 的计数不增加；WG 的 `udp sport 51820` 在 `output-tproxy` 链首有显式 accept。

失败模式 / 回滚点：`meta skuid 0` 匹配不到内核发出的 WG 握手响应包，必须按 `udp sport` 显式 accept；改规则前 `nft list ruleset > /etc/nftables.conf.bak`，回滚 `nft -f /etc/nftables.conf.bak`。

21.6.5　5) 一张表收口

验收项	命令（简写）	通过标准
客户端 DNS 列表	`adb shell dumpsys connectivity | grep DnsAddresses`	只两条 ULA
分流生效	三个实例 `stats_noreset`	A+2，B/C 各 +1
DNSSEC 好签名	`dig +dnssec signed.dns.example.org A`	`ad` 标志
DNSSEC 坏签名	`dig +dnssec badsig.dns.example.org A`	SERVFAIL
坏 MTU	`dig +bufsize=4096 ... DNSKEY` + 抓包	无分片、无 TCP 重试
上游断开	`dig +short api.dns.example.org A` 两次	第二次仍返回（serve-stale）
nft 劫持边界	`nft list ruleset` + `nft reset counters`	uid 豁免命中，WG 出包未被打标
隧道加密传输	`tcpdump -ni wg0 'port 853'`	只有:853，无明文 53 出公网

21.7　回滚点汇总与实验纪律

层	快照	回滚动作	优先级
客户端引导（RA / RDNSS）	`/etc/ra-steer/ra.conf.bak`	停发送进程、还原配置	2

层	快照	回滚动作	优先级
nftables（劫持 / tproxy / mark）	`/etc/nftables.conf.bak`	`nft -f` 还原	1
分流清单	`policy-domains.list.bak`	单文件还原 + reload	1
四个 Unbound 实例	`/var/tmp/unbound-before-lab/`	逐实例还原配置并重启	2
隧道 MTU / 链路	`ip link show > /var/tmp/link-before-lab.txt`	`ip link set dev wg0 mtu < 原值 >`	3
`cloud-01` 上游服务	`systemctl` 状态快照	`systemctl start`	3
各层缓存	`dump_cache` 输出（可选）	NodeLocal/CoreDNS/Unbound 逐层 `flush_negative` / `flush_zone`	4

四条实验纪律：

1. **先看 nftables，再怀疑别的**。根区镜像不更新、AXFR 失败、WG 握手不通、上游「看起来全挂」——这四个现象里至少三个的根因在规则表，不在 Unbound 配置。
2. **回滚先救 DNS 可达性，再救策略，最后救缓存**，因为所有远程排障手段都依赖 DNS 能用。
3. **改动前设一个十分钟的自动回滚定时器**（`systemd-run --on-active=10m ...`），确认无误再取消，防止配置错误把自己关在门外。
4. **每次注入测试后逐层刷缓存**，否则前一层缓存的 NXDOMAIN 会让你误判「修复无效」（第 22 章案例③、⑤）。

各实例的完整 `unbound.conf`、DoT 服务端/客户端配置、DNSSEC 信任锚操作、nftables 规则全文、RA 引导脚本，分别见第 5、6、7、9、10、11、13、14、19 章及第四篇对应 RFC 章。

21.8 本章要点

1. 拓扑是一条链：LAN 客户端→（RDNSS 引导 / nft 劫持）→策略 DNS 实例 A → B/C 分流→加密隧道→`cloud-01` 递归验证器→权威；本地根镜像与实例 D 提供不依赖上游的迭代路径。
2. 客户端引导只靠网关自己发 RA 广告 ULA RDNSS，**光猫里不配任何 DNS**；监听表里 LAN 上唯一的:53 入口是 A，B/C/D 只听 loopback，ULA 不随 GUA 轮换。
3. 分流表必须单上游、fail-closed；策略 DNS 用各实例的 `stats_noreset` 计数验证，而不是靠日志猜。
4. 五个观测点 O1–O5 把链路切成四段，八项失败注入逐项验证降级行为；**排查顺序：先看 nftables，再怀疑别的**。
5. 验收命令覆盖 DNSSEC 好/坏签名、坏 MTU 下的 EDNS 1232、上游断开时的 serve-stale、nft 计数器与 uid 豁免，以及「公网侧只有:853」。
6. 每次注入后按层刷缓存；回滚按「DNS 可达性→策略→缓存」排序，改动前留快照与自动回滚定时器。

21.9 参考资料

- RFC 5737, *IPv4 Address Blocks Reserved for Documentation*: https://www.rfc-editor.org/rfc/rfc5737
- RFC 3849, *IPv6 Address Prefix Reserved for Documentation*: https://www.rfc-editor.org/rfc/rfc3849
- RFC 4193, *Unique Local IPv6 Unicast Addresses*: https://www.rfc-editor.org/rfc/rfc4193
- Unbound 官方手册（配置选项与 `unbound-control`）: https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html
- Unbound `unbound-control` 手册: https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound-control.html
- nftables 官方 wiki（计数器与规则集管理）: https://wiki.nftables.org/wiki-nftables/index.php/Main_Page
- Kubernetes 官方文档 *Using NodeLocal DNSCache in Kubernetes clusters*（第 20 章拓扑 C 的对应）: https://kubernetes.io/docs/tasks/administer-cluster/nodelocaldns/
- DNS Flag Day 2020（EDNS 缓冲区尺寸建议）: https://dnsflagday.net/2020/

第二十二章

故障案例集

22.1 案例索引与排查纪律

#	一句话现象	根因类别	关联章节
①	加了第二个上游，过滤表失效	转发器语义误读	第 5 章
②	上游超时后整条查询流裸奔	允许静默明文降级	第 5、14 章
③	某站 CSS/JS 全部解析失败	strict QMIN 撞违反 [RFC 8020] 的区	第 13、17 章
④	间歇性全量 SERVFAIL	转发器上配了信任锚	第 6、10 章
⑤	权威修好了客户端仍失败	负缓存按 SOA minimum 续命	第 7 章
⑥	根区镜像永远不更新、AXFR 失败	本机解析器出向 53 被自己劫持	第 9 章
⑦	WireGuard 隧道起不来	回包被 `output-tproxy` 打标	第 14 章
⑧	换了前缀但 IPv6 全断	临时地址寿命 × [RFC 6724] 源地址选择	第 19 章
⑨	DDNS 与 RA 通知同一天各失效一次	/48 与 /64 粒度混用	第 19 章
⑩	跨境 DNS 时快时慢、UDP 几乎不通	跨境 UDP QoS	第 14、19 章

贯穿全部案例的一条经验，先写在最前面：

> **排查顺序：先看 nftables，再怀疑别的。**

十根因里有三根（⑥、⑦，以及③里「刷了缓存却没生效」的一部分）在规则表里，还有两根（②、⑩）的第一反应会把人引到 DNS 配置上，而真正要核的是规则与链路。策略网关上的一次 `nft list ruleset` 加上计数器清零重测，成本远低于重启解析器、换上游、刷缓存这三件套。下文每个案例的「实测定位」都按这个顺序走。

22.2 案例①单上游变竞速

- **现象**：为了「冗余」，在策略转发器里加了第二个 `forward-addr`。改完之后，一部分本该被过滤的名字返回了应答，日志里查询出现在两个上游中的任意一个；过滤表时灵时不灵。
- **第一反应（错误判断）**：以为这是主备切换生效——「第二个上游顶上了，说明高可用做对了」。也有人怀疑是过滤表清单加载失败，反复重载清单。
- **实测定位**：先 `nft list ruleset` 确认没有规则把包分流到别处；再在两个上游各自开查询日志，用同一个名字连查十次，发现两边都收到了查询，且哪边先回就用哪边的应答。这不是切换，是并发。

- **根因**：给转发器配第二个 forward-addr，Unbound 是**并发竞速取最快**而非主备。查询会被送到不该去的解析器，过滤表只在「恰好选中那个上游」时生效。冗余的语义被误当成故障转移的语义。
- **修复**：转发器只留一个上游，切换交给外部巡检脚本改配置 + reload。

```
# /etc/unbound/instance-b.conf ——转发器只留一个上游
forward-zone:
    name: "."
    forward-addr: 192.0.2.53
    forward-first: no
```

为什么这么设：forward-first: no 保证转发失败时不退回普通递归（案例②），单条 forward-addr 保证策略边界只有一个出口。

怎么验证：unbound-control -c /etc/unbound/instance-b.conf stats_noreset | grep num.query，并在两个候选上游分别抓包，确认只有一个收到查询。

失败模式 / 回滚点：只留一个上游意味着该上游故障时这条链路全停，这是有意的取舍，靠外部脚本切换而不是靠竞速。回滚点：cp -a /etc/unbound/instance-b.conf /var/tmp/instance-b.conf.bak，改坏时还原并 reload。

- **验证**：连续查 20 次同一个名字，两个上游的查询计数应为 20 / 0；过滤域始终返回预设应答。
- **如何避免**：把「多上游」写进变更评审的禁用项；任何「为了冗余加一行 forward-addr」的提案，先问一句「这是竞速还是切换」。第 20 章 CoreDNS 的 forward 同样适用。

22.3　案例②静默明文降级

- **现象**：cloud-01 上的 DoT 上游超时后，客户端查询仍能成功，只是变慢了一点；抓公网侧的包发现出现了明文 53 的查询，里面是完整 QNAME。没有任何告警。
- **第一反应（错误判断）**：「上游挂了但服务还活着，说明容错做得不错」。甚至有人把「自动回退」当成设计特性写进文档。
- **实测定位**：nft list ruleset 看是否有规则放行明文 53 出向；在 WAN 口 tcpdump -ni wan0 'port 53 or port 853'；然后 dig @fd00:53::1 dns.example.org A 同时停掉 DoT 服务端，捕获到:53 流量与成功应答同时出现。
- **根因**：上游超时后允许回退到明文 53（或 forward-first: yes 之类的宽松设置），fail-closed 没落实。**加密传输（DoT / DoH / DoQ）与 DNSSEC 是两件事**：这里丢的是传输加密与隐私，DNSSEC 的 ad 标志可能还在，于是监控看不出异常。
- **修复**：forward-first: no、加密上游两波都失败就返回 SERVFAIL，并在 nft 侧拒绝本机解析器之外的明文 53 出向。

```
# 转发器：禁止一切向明文递归的逃逸
forward-zone:
    name: "."
    forward-addr: 198.51.100.2@853
    forward-tls-upstream: yes
    forward-first: no
```

为什么这么设：forward-tls-upstream: yes 把这条上游钉在 TLS 上，forward-first: no 关掉「转发失败就退回普通递归」的出口；两者合起来才是 fail-closed。

怎么验证：停掉 cloud-01 的 DoT 服务后 dig @fd00:53::1 dns.example.org A 应返回 SERVFAIL，且 tcpdump -ni wan0 'port 53' 无任何新包。

失败模式 / 回滚点：fail-closed 的代价是上游故障时**业务真的会失败**，这是设计意图，不要为了可用性再开回退。回滚点：保留原始配置快照；若临时需要应急，用一条有明确到期时间的规则而不是改 forward-first，并设 systemd-run --on-active=10m 自动撤销。

- **验证**：注入上游断开，确认「无明文 53 出公网 + 客户端 SERVFAIL」两条同时成立；恢复 DoT 后查询恢复且 ad 标志正常。
- **如何避免**：把「明文 53 出向包数」做成常驻告警；任何健康检查只测「能否解析」而不测「是否加密」，都会漏掉这类降级。

22.4 案例③ strict QMIN 撞违反 RFC 8020 的 CDN 区

- **现象**：开启 `qname-minimisation-strict: yes` 后，某大型 CDN 承载的整站 CSS / JS / 图标全部解析失败，页面只剩纯文本；主域却能解析。关掉 strict 之后**仍然失败**，重启解析器之后**还是失败**。
- **第一反应（错误判断）**：「CDN 节点故障」「运营商劫持」「strict 关了没生效，配置没重载」。于是反复 reload、换上游、清客户端 DNS 缓存，问题依旧。
- **实测定位**：`nft list ruleset` 排除规则因素；然后逐层 `dig`：`dig @<NodeLocal> +short asset.dns.example.org.`、`dig @<CoreDNS> ...`、`dig @<Unbound> ...`，发现**每一层**都返回 NXDOMAIN，但最外层向权威直接查却能得到正确 CNAME。定位到缓存，而不是配置。
- **根因**：查询名最小化（QNAME minimisation，QMIN）在 strict 模式下，问中间层拿到的 NXDOMAIN 不回退到完整域名。而该 CDN 的 CNAME 区违反 [RFC 8020]（父名返回 NXDOMAIN、子名却存在），于是整片静态资源被判为不存在。更关键的是：NXDOMAIN 已经作为负缓存写进了**链路上每一层**——NodeLocal、中心 CoreDNS、Unbound、以及应用自身的解析器。只改配置、只刷一层，前一层照样吐缓存的 NXDOMAIN。
- **修复**：把 `qname-minimisation-strict` 改回 `no`（保留 QMIN 本身），然后按 `NodeLocal → CoreDNS → Unbound` 的顺序逐层刷负缓存。

```
# 兼容优先：不开 strict
server:
    qname-minimisation: yes
    qname-minimisation-strict: no
```

为什么这么设：QMIN 仍保留（减少发给上级的 QNAME 标签数，抗元数据聚合），但允许在遇到坏区时回退到完整域名。注意 QMIN **不是加密**：它不抗全路径嗅探；本地根镜像时根这一跳本来就不出口，收益落在 TLD 层。

怎么验证：`dig @192.0.2.53 +short asset.dns.example.org.` 返回 CNAME；`unbound-control -c /etc/unbound/instance-d.conf get_option qname-minimisation-strict` 显示 no（若 `get_option` 不支持则直接查配置文件）。

失败模式 / 回滚点：关 strict 会略微增加发送给上级的标签数，这是兼容性代价。回滚点：配置文件快照 + 逐层刷缓存脚本；**不要**用「重启单个实例」代替逐层刷。

```
# 逐层刷，顺序从离客户端最近的一层开始
kubectl -n kube-system rollout restart ds/node-local-dns     # 若有 NodeLocal（第 20 章）
kubectl -n kube-system rollout restart deploy/coredns        # 若有 CoreDNS
unbound-control -c /etc/unbound/instance-a.conf flush_negative
unbound-control -c /etc/unbound/instance-d.conf flush_negative
unbound-control -c /etc/unbound/instance-d.conf flush_zone dns.example.org
```

为什么这么设：从离客户端最近的一层开始清，避免清完最外层后近端缓存又把旧 NXDOMAIN 吐出来（案例③、⑤的共同要求）。

怎么验证：`dig @192.0.2.10 +short asset.dns.example.org.`、`dig @<CoreDNS-ClusterIP> +short asset.dns.example.org.`、`dig @192.0.2.53 +short asset.dns.example.org.` 三层都返回 CNAME。

失败模式 / 回滚点：`unbound-control` 未启用远程控制时只能重启实例，且每个实例都要刷；`rollout restart` 会让该层短暂降级，滚动窗口内要保证 `serve-expired` 开着。回滚点：刷缓存前的 `dump_cache` 输出与 ConfigMap 快照。

- **验证**：每一层分别 `dig @< 该层地址 > +short asset.dns.example.org.` 都返回 CNAME；页面静态资源加载完整。
- **如何避免**：默认不开 strict；把它列为「只在确认全链路名字空间都合规时才考虑」的选项。凡是出现「NXDOMAIN 修不好」，先列出链路上所有缓存层再动手，逐层刷、逐层验。

22.5 案例④转发器上配 trust anchor 导致 priming 失败

- **现象**：某个实例间歇性全量 SERVFAIL，日志里持续刷 `failed to prime trust anchor`；重启后短暂正常，几分钟后又坏。
- **第一反应（错误判断）**：「根区镜像坏了」「信任锚文件过期」「上游 DNSKEY 应答被丢」，于是反复换 `root.hints`、重新下载锚文件。

- **实测定位**：`nft list ruleset` 排除出向 53 被劫持（那是案例⑥）后，看该实例的 `forward-zone` 与 `server:` 段，发现两者同时存在：`forward-zone "."` 把一切交给上游，而 `auto-trust-anchor-file` 要求自己向根 priming DNSKEY。抓包确认该实例从未向根发过 DNSKEY 查询。
- **根因**：信任锚（trust anchor）需要验证器自己向根 priming DNSKEY，而纯转发器把全部查询交给上游，无法完成 priming。**转发器与验证器二选一**，不能在同一实例里既全量转发又做验证。
- **修复**：拆成两个实例——纯转发器去掉所有验证器选项，递归验证器去掉 `forward-zone "."` 并自己持有信任锚（配置见下方）。

```
# 方案 A：纯转发器，去掉所有验证器选项
# server: 段里不要出现 auto-trust-anchor-file / val-permissive-mode
forward-zone:
    name: "."
    forward-first: no
    forward-addr: 192.0.2.53

# 方案 B：递归验证器，去掉 forward-zone "."，自己迭代并持有信任锚
# server:
#     auto-trust-anchor-file: "/var/lib/unbound/root.key"
# auth-zone:
#     name: "."
#     ...（本地根镜像见第 6 章）
```

为什么这么设：把两个角色拆到不同实例（第 7 章的多实例分工），各自只需满足自己那套前提；方案 B 的递归能力由本地根镜像支撑，不依赖转发上游。

怎么验证：`journalctl -u unbound@instance-a --since '10 min ago' | grep -i 'prime\|bogus'` 无新记录；`dig @192.0.2.53 +dnssec badsig.dns.example.org A` 在验证器侧返回 SERVFAIL、在转发器侧不假装验证。

失败模式 / 回滚点：拆成两个实例后，转发器的应答不带 `ad` 标志是**正常的**（它没有验证），别误判成 DNSSEC 失效。回滚点：两个实例的配置文件快照，以及 `/var/lib/unbound/root.key` 的备份。

- **验证**：连续观察 30 分钟无 `failed to prime`；坏签名域只在验证器那条链路上被拒。
- **如何避免**：配置模板按角色分离，转发器模板里物理删除验证器选项，不要靠注释「以后可能要用」。第 20 章强调的「DNSSEC 验证放在 Unbound 那一层」在裸机拓扑里同样成立。

22.6　案例⑤负缓存放大

- **现象**：上游权威瞬时返回过一次 NXDOMAIN，运维修好权威之后，客户端仍然持续失败约一小时；不同客户端「有的好了有的没好」。
- **第一反应（错误判断）**：「权威没修干净」「传播延迟」「客户端自己缓存了」，于是反复改权威记录、清浏览器缓存。
- **实测定位**：`nft list ruleset` 排除规则后，逐层 `dig +nssearch` 或直接 `dig @<每层> no-such.dns.example.org.`，看哪一层还在返回 NXDOMAIN；再 `unbound-control ... dump_cache | grep -A3 no-such` 看到带 SOA 的负缓存条目及其剩余 TTL。
- **根因**：上游瞬时 NXDOMAIN 被按 SOA minimum 缓存（常见 3600 秒），`cache-max-negative-ttl` 官方默认也是 3600。修好权威不会让已有负缓存失效。且**链路上每个实例都有自己的负缓存**，只刷一个实例就会看到「有的客户端好了」。
- **修复**：把 `cache-max-negative-ttl` 压到 60 秒，并用 `flush_negative` / `flush_zone` 逐实例刷掉已经写进去的负缓存（配置见下方）。

```
# 缩短负缓存上限；不要指望关掉它（NXDOMAIN flood 需要它）
server:
    cache-max-negative-ttl: 60
    # 出问题时用 unbound-control flush_negative / flush_zone 逐实例刷
```

为什么这么设：把负缓存从 3600 秒压到 60 秒，让「修好权威」到「客户端恢复」的窗口可控；仍然保留负缓存以吸收不存在的名字查询。

怎么验证：`dig @192.0.2.53 +short no-such.dns.example.org.` 后立刻 `unbound-control -c /etc/unbound/instance-d.conf flush_negative`，再查应改为向上游发起新查询；`unbound-control ... dump_cache` 里该条目消失。

失败模式 / 回滚点：`unbound-control` 没启用远程控制时**只能重启实例刷缓存**，且每个实例都要刷。压太短（例如 1 秒）会让不存在的名字查询打穿到上游，遇扫描流量时放大。回滚点：配置快照 + 一份 `dump_cache` 输出便于对照。

- **验证**：修好权威后 60 秒内所有实例都返回正确应答；逐层 `dig` 结果一致。
- **如何避免**：把「逐层刷缓存」写成固定 runbook 步骤；第 20 章的 CoreDNS `cache { denial ... 5 }` 与 Unbound 的 `cache-max-negative-ttl` 要**分别**设置，别以为设了一处就全链路生效。

22.7 案例⑥本机解析器出向 53 被自己 hijack

- **现象**：本地根区镜像永远不更新，AXFR 失败；递归查询时好时坏。同一台网关上，LAN 客户端的 DNS 却完全正常。
- **第一反应（错误判断）**：「上游根服务器不通」「AXFR 的 TSIG 密钥错了」「镜像源站改了地址」。有人甚至把 `<TOKEN>` 换了三遍。
- **实测定位**：**先看 nftables**——`nft list ruleset | grep -A6 hijack-dns`，发现 LAN/WG 的:53 一律送进 tproxy，而规则里没有对本机解析器进程的 uid 豁免。再 `nft reset counters` 后发起一次递归，看到 hijack 计数增加，而 WAN 口:53 出向计数为零：自己的递归查询被自己吃掉了。
- **根因**：策略网关把 LAN/WG 的:53 送进 tproxy 是为了兜住写死:53 的存根解析器，但**本机解析器进程（unbound、nsd）的出向 53 必须按 uid 豁免**，否则它自己的递归查询又被劫持回自己，形成环。
- **修复**：在 `hijack-dns` 链的 tproxy 规则之前按 uid 插入豁免，例如 `meta skuid $UNBOUND_UID accept` / `meta skuid $NSD_UID accept`（`$*_UID` 为运行时查到的数值 uid，骨架见下方）。

```
# nft 策略层骨架：uid 豁免必须写在 tproxy 之前
table inet mangle {
  chain hijack-dns {
    meta skuid 953 accept      # 示例：unbound 服务用户的数值 uid，按实际替换
    meta skuid 954 accept      # 示例：nsd 服务用户的数值 uid，按实际替换
    # 其余 LAN/WG 的 :53 才进 tproxy
    udp dport 53 meta l4proto udp tproxy to :5355 meta mark set 0x2333
  }
  chain output-tproxy {
    # 链首：WG 自己的出包显式放行（案例 ⑦）
    udp sport 51820 accept
    meta skuid 953 accept      # 与 hijack-dns 同一数值 uid 示例
    # ... 其余规则
    meta mark set 0x2332
  }
}
```

为什么这么设：策略网关把 LAN/WG 的 53 送进 tproxy 是为了兜住写死:53 的存根解析器，但**本机解析器进程（unbound、nsd）的出向 53 必须按 uid 豁免**，否则它自己的递归查询又被劫持回自己，形成环。

怎么验证：`nft reset counters` 后 `dig @127.0.0.1 -p 5359 +short dns.example.org A`，hijack-dns 计数不增加，WAN 口 `tcpdump -ni wan0 'port 53'` 能看到出向查询。

失败模式 / 回滚点：`meta skuid 0` 匹配不到内核发出的包（案例⑦会再踩一次）；`meta skuid` 匹配的是数值 uid，上线前用 `id -u unbound` / `id -u nsd` 查真实值替换上面的示例数字，写错会静默不命中。回滚点：`nft list ruleset > /etc/nftables.conf.bak`，出问题时 `nft -f /etc/nftables.conf.bak`。

- **验证**：根区镜像的 SOA 序列号随权威更新；AXFR 成功；递归查询稳定。
- **如何避免**：把 uid 豁免写进 hijack 规则模板的第一条，任何新增本机解析服务时同步加豁免。这条经验本身就是本章开头那句「先看 nftables」的由来。

22.8　案例⑦ WG 回包被 output-tproxy 打标

- **现象**：WireGuard 隧道起不来，握手反复重试；`wg show` 里 latest handshake 一直是空。ping 隧道对端地址不通，但网关本身出公网正常。
- **第一反应（错误判断）**：「云主机防火墙没放行 UDP」「密钥配错了」「运营商封了 UDP」。于是换端口、换密钥、换云主机区域。
- **实测定位**：**先看 nftables**——`nft list chain inet mangle output-tproxy`，发现链尾有 `meta mark set 0x2332`，把 WG 自己的 `udp sport 51820` 出包也一并打标送进代理。抓包看到握手包发出去了，但响应回不来；对端日志显示从未收到可用的握手响应。
- **根因**：策略网关的 `output-tproxy` 链尾统一打标，连内核为 WG 发出的报文一起命中。试过用 `meta skuid 0` 排除，**匹配不到内核发出的这类包**，因为这类包没有用户态 socket 归属。必须在链首按 `udp sport` 显式 accept（见案例⑥的规则骨架）。
- **修复**：把 `udp sport 51820 accept` 放到 `output-tproxy` 链首，放在链尾的 `meta mark set 0x2332` 之前（骨架见下方）。

```
# output-tproxy 链首，顺序不可交换
udp sport 51820 accept
# 下面才是普通的本机出向策略与链尾打标
meta mark set 0x2332
```

为什么这么设：按源端口做无状态例外，不依赖 socket 归属，能覆盖内核发出的握手响应；放在链首保证不会被后面的打标规则追上。

怎么验证：`nft reset counters` 后 `wg show wg0 | grep -i handshake` 应出现时间戳；`tcpdump -ni wan0 'udp port 51820'` 看到双向报文。

失败模式 / 回滚点：把 `udp sport 51820 accept` 放到链尾就没用了，链尾打标会先命中。若隧道端口是变量，用 `define WG_PORT = 51820` 并在规则里引用，避免改端口时漏改。回滚点：`/etc/nftables.conf.bak`。

- **验证**：`wg show` 有 latest handshake；`ping` 隧道对端通；隧道内 DoT 查询成功。
- **如何避免**：任何「统一打标」的链尾规则上线前，列出本机所有由内核发起的出向协议（WG、IPsec、NDP 相关）逐条做链首例外。案例⑥和⑦是同一条纪律的两个侧面：**先看 nftables，再怀疑别的**。

22.9　案例⑧临时地址寿命 × 前缀轮换

- **现象**：运营商换 IPv6 前缀之后，客户端拿到了新前缀、配了新地址、默认路由也正确，但 IPv6 出站全断；`ping6` 网关 ULA 却通。约两天后自行恢复。
- **第一反应（错误判断）**：「运营商路由没收敛」「上游不回程路由」「新前缀没下发完整」。于是等、重启光猫、改默认路由，都没用。
- **实测定位**：`nft list ruleset` 排除出向过滤后，`ip -6 addr show dev lan0 scope global` 看到新旧两个 /64 地址并存；`ip -6 route get 2001:db8::1` 显示**源地址仍是旧前缀**。抓包确认出向包源地址属于已撤销的旧 /64。
- **根因**：[RFC 4941] 隐私扩展生成的临时地址，Linux 上 `temp_valid_lft` 默认 48 小时。换成新前缀后旧前缀的临时地址仍存活，而 [RFC 6724] 的地址选择规则**优先选临时地址作源地址**，于是上游不路由回包。所谓「两天后自行恢复」正是旧地址寿命耗尽。
- **修复**：把 `temp_valid_lft` 压到 1800 秒，且 `all.*` / `default.*` / 接口级三处都要设（配置见下方，完整脚本见第 19 章）。

```
# /etc/sysctl.d/90-ipv6-privacy.conf
net.ipv6.conf.all.use_tempaddr       = 2
net.ipv6.conf.default.use_tempaddr   = 2
net.ipv6.conf.all.temp_valid_lft     = 1800
net.ipv6.conf.default.temp_valid_lft = 1800
```

为什么这么设：`use_tempaddr = 2` 生成并优先使用临时地址；`temp_valid_lft = 1800` 把旧前缀残留从 48 小时压到 30 分钟，把故障窗口收进可接受范围。

怎么验证：`sysctl net.ipv6.conf.lan0.temp_valid_lft` 必须返回 1800——**只查 `all.*` 不算数**。

失败模式 / 回滚点：这份文件只对之后新建的接口生效，`all.*` 不回溯改写已存在的接口；NetworkManager 连接激活时还会把 `use_tempaddr` 写成 `-1` 覆盖。因此 `all.*` / `default.*` / **接口级**三处都要设（完整脚本见第 19 章）。回滚：删除文件 + `sysctl --system`，并把接口级运行时值写回。

- **验证**：轮换演练后 `ip -6 route get 2001:db8::1` 的源地址属于新 /64；30 分钟内旧地址消失。
- **如何避免**：把接口级 sysctl 写进 NetworkManager 的激活钩子，别只依赖 `/etc/sysctl.d/`。注意发零寿命 PIO 清不掉旧地址——[RFC 4862] 第 5.5.3 节不允许把已存在的长寿命地址立刻缩短（STALE PIO 清理在 Linux 上无效），只能靠短寿命加客户端侧主动清理。

22.10 案例⑨ /48 与 /64 粒度

- **现象**：同一天出现两次互不相干的故障。第一次：运营商换前缀后 DDNS 记录没更新，外部访问断。第二次：RA 通知脚本没触发，客户端迟迟拿不到新配置。两次的现象、报错、影响面完全不同。
- **第一反应（错误判断）**：第一次判为「DDNS 服务商 API 限流」，检查 `curl` 与 `<TOKEN>`；第二次判为「RA 发送进程卡死」，重启进程。都短暂好转，很快复发。
- **实测定位**：`nft list ruleset` 排除后，检查两个脚本共用的变量，发现都取自 RIO（Route Information Option）广告的 `2001:db8:1::/48`。而运营商轮换的是 LAN 的 **/64**（`2001:db8:1:1::/64` → `2001:db8:1:9::/64`），/48 聚合始终不变，于是两个「前缀是否变化」的判断都恒为假。
- **根因**：Android 的 `accept_ra_rt_info_min_plen=48` 会丢弃前缀长度小于 48 的 RIO，所以 **RIO 必须用 /48**（或更细）；但**前缀变更检测必须用 PIO 的 /64**。把「路由聚合粒度」和「地址变更粒度」共用一个变量，导致一处错、两处同时错。
- **修复**：把两个脚本的数据源拆开——DDNS 与 RA 通知只读 PIO 的 /64，RIO 的 /48 只用于生成下游路由，命名上就区分 `LAN_PREFIX_64` 与 `ROUTE_AGG_48`。

```
# 只从 PIO 取 LAN /64，绝不从 RIO 推导
ip -6 addr show dev lan0 scope global | grep '^    inet6 2001:db8'
# 变更检测的输入：当前 /64；RIO 的 /48 只用于下游路由
```

为什么这么设：把两个数据源彻底拆开，DDNS 与 RA 通知各自读自己的那一份，避免一个 /48 变量被两处复用。

怎么验证：轮换后 `ip -6 addr show dev lan0 scope global` 的 /64 变化、`ip -6 route show | grep 2001:db8:1::/48` 保持不变，同时 DDNS 记录更新、RA 通知触发——三者同时成立才算修好。

失败模式 / 回滚点：若脚本已把 /48 写进 DDNS，先手工更新一次记录再回滚脚本变量；不要靠清缓存解决，负缓存（案例⑤）会让旧值续命。回滚点：两个脚本的快照与 DDNS 记录的历史值。

- **验证**：连续两次轮换演练，DDNS 与 RA 通知都各自触发一次，且 RIO 生成的 /48 路由不抖动。
- **如何避免**：变量名里就写清粒度（`LAN_PREFIX_64` 与 `ROUTE_AGG_48`），禁止用泛化的 `PREFIX`。第 19 章把这个坑记为「同一天栽两次」；这类「两个故障一个根因」的情形，值得专门查一次共用变量。

22.11 案例⑩跨境 UDP QoS

- **现象**：跨境链路上 DNS 时快时慢，UDP 承载的加密通道几乎不可用；大一点的应答直接超时后走 TCP 重试。同一时间 TCP 上的 SSH 却很流畅。
- **第一反应（错误判断）**：「云主机解析器性能不够」「递归层数太多」「DoT 服务端线程数太小」。于是加副本、调 `num-threads`、换上游，UDP 依旧。
- **实测定位**：`nft list ruleset` 确认没有 mark / 限速规则后，在同一条链路上做对照实测：UDP/QUIC 下载 1MB 只到约 300 B/s，而 SSH SOCKS（TCP）达约 660 KB/s。差别不是解析器，是链路对 UDP 的 QoS。再抓包看是否有 IPv6 分片被丢（叠加 `edns-buffer-size: 1472` 时更明显）。
- **根因**：该链路对跨境 UDP 做了限速/劣化。DNS 与 QUIC 都走 UDP，首当其冲；而 TCP 承载的通道有重传与拥塞控制，反而稳。DNS 隧道同理：**加密传输优先选 TCP 承载的通道**。另外 `edns-buffer-size: 1472` 会在 MTU 1280（WireGuard / IPv6 最小 MTU）与 PPPoE 1492 上分片，分片被丢后等超时重试 TCP，与 QoS 叠加成「双重慢」；官方默认 1232（DNS Flag Day 2020）才是对的，且 **`max-udp-size` 只管入向，出向靠 `edns-buffer-size`**。

- **修复**：加密传输选 TCP 承载的通道（DoT），出向 `edns-buffer-size` 钉在 1232（配置见下方），不要拿 `max-udp-size` 当出向修复。

```
# 验证器 / 递归解析器出向尺寸
server:
    edns-buffer-size: 1232
    # max-udp-size 只影响入向，不要拿它当出向修复
```

为什么这么设：1232 让大应答在 MTU 1280 的隧道里不分片，去掉「超时→ TCP 重试」这一层额外延迟；配合 TCP 承载的加密通道，绕开 UDP QoS。

怎么验证：`dig @192.0.2.53 +bufsize=4096 +dnssec dns.example.org DNSKEY` 一次返回，抓包里无分片头、无 53/TCP 重试；`unbound-control ... get_option edns-buffer-size` 为 1232。

失败模式 / 回滚点：把 1232 再往下压（例如 512）会产生大量 TCP fallback，属于过度修正。回滚点：配置快照；若临时必须走 UDP，给它单独一条链路而不是改全局尺寸。

- **验证**：同一链路上 DNS 平均时延稳定；大 DNSKEY 应答不触发 TCP 重试。
- **如何避免**：第 14 章的加密通道选路把「链路对 UDP 的待遇」当作一等输入；选 DoT（TCP 853）而非 QUIC/DoQ 承载的 UDP 通道时，先做一次 1MB 对照实测，别凭协议理论下结论。第 19 章那次「为改 DNS 去改光猫工作模式」踩到的「ICMP 通、TCP 握手通、数据阶段静默丢包」也是同一类教训：**可控性交给别人的数据面，排查成本会翻倍**。

22.12　跨案例的三条通用纪律

1. **先看 nftables，再怀疑别的**。十例里⑥⑦直接是规则表问题，②⑩的第一反应会被规则/链路误导，③⑤的「刷了没生效」也要先确认没有规则把查询引回旧路径。`nft list ruleset` + `nft reset counters` 重测，是成本最低的第一步。
2. **缓存是有层的，修复必须逐层**。③（NXDOMAIN）与⑤（负缓存）都要求 NodeLocal / CoreDNS / Unbound / 应用解析器逐层刷、逐层验；只刷最外层会看到「明明修好了却还在失败」。
3. **先分清两种「安全」，再谈降级**。DNSSEC 是数据完整性/来源认证，DoT/DoH/DoQ 是传输加密/隐私，二者不互相替代。案例②里 `ad` 标志还在却已经明文裸奔，案例④里转发器假装验证反而全量 SERVFAIL——都是把两件事混成一件的结果。

另有一条配置层面的底线：`harden-below-nxdomain` 只对 DNSSEC-secure 的 NXDOMAIN 生效（官方 man page 的 NOTE），无签名区的 NXDOMAIN 不触发剪枝，别把它当成「已生效」的防护；`harden-glue` 与 DNSSEC 无关，域外夹带的 glue 是经典缓存投毒面，任何模式都该开。这两条不改变本章任何案例的结论，但会影响你对「防护是否到位」的判断。

22.13　本章要点

1. **排查顺序：先看 nftables，再怀疑别的**——规则表能解释案例⑥（本机出向 53 被自己 hijack）、⑦（WG 回包被 `output-tproxy` 打标）以及多处「刷缓存不生效」。
2. **转发器只留一个上游、fail-closed**：多上游是并发竞速不是主备（①）；允许回退明文 53 会造成无人告警的静默降级（②）。
3. **缓存分层**：strict QMIN 撞违反 [RFC 8020] 的 CDN 区（③）与负缓存按 SOA minimum 续命（⑤），都要求逐层刷、逐层验；`serve-expired-client-timeout` 单位是毫秒，`max-udp-size` 只管入向。
4. **角色二选一**：转发器上配 `auto-trust-anchor-file` 无法完成 priming（④）；DNSSEC 验证放在验证器那一层，与加密传输不是一回事。
5. **IPv6 两处粒度**：临时地址默认 48h 寿命叠加 [RFC 6724] 源地址优选，前缀轮换后要三处 sysctl 都设（⑧）；RIO 用 /48、变更检测用 /64，共用变量会一天栽两次（⑨）。
6. **链路实测优先于协议推断**：跨境 UDP QoS 下 TCP 承载更稳（⑩），`edns-buffer-size: 1232` 避免 MTU 1280 分片；「光猫只当管道用」（第 19 章）是同一条可控性纪律。

22.14　参考资料

- Unbound 配置手册（`forward-first`、`edns-buffer-size`、`harden-below-nxdomain`、`qname-minimisation-strict`、`cache-max-negative-ttl` 等选项与 NOTE）：https://unbound.docs.nlnetlabs.nl/en/latest/manpages

/unbound.conf.html

- Unbound `unbound-control` 手册（`flush_negative` / `flush_zone` / `dump_cache`）：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound-control.html
- RFC 8020, *NXDOMAIN: There Really Is Nothing Underneath*：https://www.rfc-editor.org/rfc/rfc8020
- RFC 9156, *DNS Query Name Minimisation to Improve Privacy*（更新 [RFC 7816]）：https://www.rfc-editor.org/rfc/rfc9156
- RFC 4862 第 5.5.3 节（地址寿命与 STALE PIO）：https://www.rfc-editor.org/rfc/rfc4862
- Linux 内核 `ip-sysctl` 文档（临时地址与 RA 相关 sysctl）：https://docs.kernel.org/networking/ip-sysctl.html
- nftables 官方 wiki（规则集、计数器与 mark）：https://wiki.nftables.org/wiki-nftables/index.php/Main_Page
- DNS Flag Day 2020（EDNS 缓冲区尺寸 1232 的建议）：https://dnsflagday.net/2020/

第二十三章

WPAD 与 PAC：自动代理发现的实践

一台客户端要决定”这个请求走不走代理”，得先拿到一份能执行的小文件；**WPAD（Web Proxy Auto-Discovery）负责找到这份文件的 URL，PAC（Proxy Auto-Config）负责在文件里给出判断**。这两件事常被混成一个词，混完之后排障几乎总会找错层：DNS 里查不到 `wpad` 名字是发现层的问题，`FindProxyForURL()` 返回了错的分流结果是配置层的问题，两者的日志、验证命令和回滚点完全不同。

先把边界立清楚。本书第五篇的主线是策略网关上的**透明代理**：流量在网关被接管，客户端不需要知道代理存在，也不需要谁来告诉它”去哪里找配置”。WPAD 在这条主线里只有两个位置——给那些必须靠自动配置才能工作的**遗留客户端兼容**，以及把”名字→配置→流量路径”这条链讲透的**原理科普**。它不该出现在新架构的依赖图里，最后一节会把理由摊开讲。

23.1 两层：找 URL 与决定怎么走

23.1.1 WPAD 层

输入是客户端所处的网络上下文（DHCP 应答、DNS 域名、search 域），输出是一**个 URL**，典型形态是 `http://wpad.example.net/wpad.dat`。它不解析这个 URL 里的内容，也不做任何分流判断；它成功与否，只决定客户端手里有没有一个可以取 PAC 的地址。

23.1.2 PAC 层

输入是 `url` 与 `host`，输出是一串**代理指令字符串**：`DIRECT`、`PROXY 198.51.100.10:3128`，或用分号串起来的回退链 `PROXY 198.51.100.10:3128; DIRECT`。文件本身是一段 JavaScript，入口函数固定为 `FindProxyForURL()`。客户端每发起一个请求就调用一次它，所以 PAC 里每一次 `dnsResolve()` 都是一次额外的域名查询。

维度	WPAD（发现层）	PAC（配置层）
输入	DHCP / DNS / 本地域名后缀	请求的 URL 与主机名
输出	PAC 文件的 URL	`DIRECT` / `PROXY` / 回退链
失败后果	客户端退回”不配置代理”或沿用旧配置	单次请求走错路径或被拒
标准状态	过期的工作组草案（见下节）	Netscape 1996 的事实约定，无正式 RFC
能否替换	能用显式 PAC URL 或手工配置绕开	能用静态代理设置绕开
排障入口	DHCP 包、`dig`、系统代理开关	`pactester`、浏览器开发者工具

这一分层是全章的骨架：后面所有”配好了却不生效”的问题，先问一句”是没找到 URL，还是找到了 URL 但文件里的判断不对”，能省掉一半的瞎试。

23.2 标准状态：一份过期的工作组草案

23.2.1 draft-ietf-wrec-wpad-01

WPAD **从未成为 RFC**。它唯一成文的定义是 IETF WREC 工作组的 Internet-Draft `draft-ietf-wrec-wpad-01`：**1999-07-29 发布，1999-12 过期**，之后没有推进到 RFC 序列。本书凡提到”WPAD 规范要求”时，指的都是这份**已过期草案**里的条文，不是任何 RFC 的规范性要求——这一点在写文档、做验收、跟厂商对口径时最容易出错。

因此本章不会给 WPAD 编一个 RFC 编号，也不会把草案里的 `MUST` / `SHOULD` 当成 RFC 级别的互操作义务。草案引用的周边协议倒是有正式编号：DHCP 选项见 [RFC 2132]，SLP 见 [RFC 2608] / [RFC 2609]，Well-Known Alias 的约定见 [RFC 2219]。DNS SRV 记录类型与其余拿不准的编号一律**见官方资料**，章内不填。

23.2.2 过期的东西为什么还在跑

一份 1999 年过期的草案能活到今天，靠的是客户端实现的默认行为，不是标准推进。主流操作系统与浏览器里的”自动检测代理设置”开关，实现的是草案描述的发现流程；只要开关默认勾上，网络里存在一条 `wpad` A 记录或一个 252 选项，客户端就会自己去取 `wpad.dat`。**事实标准不等于标准**：它没有版本协商、没有能力通告、没有失败语义的统一定义，各家实现在细节上分叉，正是后面”缓存与降级”一节里行为不一致的来源。

23.3 草案 §4.4：五种发现机制与 MUST / SHOULD

草案 §4.4 列了五种发现途径，并用规范性用词分了级。下表按草案原文的分级抄录，注解是本章补的实践说明：

#	机制	草案分级	客户端动作	依赖 / 注解
1	DHCP option 252	**MUST**	从 DHCP 应答里取出 PAC URL	252 **不在 [RFC 2132] 正式选项表**；落在 IANA 私有/保留编号段
2	SLP	**SHOULD**	按服务类型查 `service: URLs` 属性	[RFC 2608] / [RFC 2609]
3	Well-Known Alias `wpad.TGTDOM.` 的 A 记录	**MUST**	解析 A 记录，再 HTTP GET `/wpad.dat`	[RFC 2219]；最常被触发的一条
4	DNS SRV	**SHOULD**	查 SRV 记录得到主机与端口	记录类型定义见官方资料
5	DNS TXT `service: URLs`	**SHOULD**	读 TXT 中的 URL 列表	与 SLP 共用同一属性名

再强调一次：这些 `MUST` / `SHOULD` 是**草案内部**的分级，不是 RFC 义务。实现里最常见的组合其实是第 1 条加第 3 条——先问 DHCP，拿不到再拼 `wpad.< 域名 >` 去查 DNS。

23.3.1 DHCP option 252 的尴尬位置

252 这个编号不在 [RFC 2132] 的正式 DHCP 选项表里，属于 IANA 编号空间中的私有/保留用途段（具体区间划分见官方资料），靠的是各家实现之间的事实约定。后果有两个：

- 服务端虽然”必须”给，但没有任何标准条文规定值的格式、编码与超长时的处理，不同 DHCP 服务端填法略有差别；
- 客户端若在租约期内才启用自动检测，或第一次应答没带上 252，可以另发 **DHCPINFORM** 单独补取该选项，不必等续租。

所以”我明明配了 252，客户端却没拿到”这类问题，第一现场是 DHCP 报文，不是 DNS 日志。抓包确认 252 是否出现在 OFFER / ACK / DHCPINFORM 的应答里，再往下走。

23.3.2 SLP 与两条 DNS 附加机制

SLP 是草案里排在 SHOULD 档的服务发现协议，规范见 [RFC 2608] 与 [RFC 2609]。它需要一个可用的服务目录，部署成本明显高于”加一条 A 记录”，因此在小型网络里基本见不到。DNS SRV 与 DNS TXT `service: URLs` 同样只

是 SHOULD：前者把主机与端口写进记录，后者直接把 URL 列表放进 TXT。三者都要求客户端主动实现相应查询，而草案过期之后没有推动力去统一实现，实际覆盖率远低于第 1、3 条。

23.3.3 Well-Known Alias：一次 A 记录加一次明文 HTTP GET

第 3 条是本章实践部分要落地的那条。流程极简：把 `wpad.` 拼在目标域名（草案称 `TGTDOM`）前面，查 **A 记录**，取到地址后向该地址的 80 端口请求 `/wpad.dat`。没有 TLS、没有签名、没有内容校验；整条链的安全性全押在”这个 `wpad` 名字归谁”上。[RFC 2219] 讲的是 Well-Known Alias 这一命名约定本身，不是 WPAD 的安全模型——把两者混读，会误以为这条链有标准背书。

23.4 DNS devolution：会走出组织边界的那一跳

23.4.1 逐级剥左侧标签

`TGTDOM` 从客户端的 DNS 域名（或 search 域）来。草案允许的 **devolution（域名退化）**是：查不到就剥掉最左侧一个标签再查，**一直剥到只剩 TLD** 为止，任一步取到 A 记录即停止。

```
客户端所属域：corp.example.net
TGTDOM 初值：corp.example.net.

  第 1 跳   wpad.corp.example.net.   -> 组织内 DNS，可控
  第 2 跳   wpad.example.net.        -> 组织内 DNS，可控
  第 3 跳   wpad.< 公共后缀 >.         -> 公共命名空间，不可控
  命中任一跳的 A 记录即停止，向该地址 HTTP GET /wpad.dat
```

问题就在第 3 跳：`example.net` 这一层在真实部署里往往已经不是组织可写的区；再往外剥到 TLD，名字彻底落到公共命名空间。**devolution 的设计假定”越往上越靠近组织自己”，实际恰好相反**——剥标签是在往组织的控制边界外面走。

23.4.2 `wpad.< 公共后缀 >` 与 W-Bad 类攻击

历史上确实有人把公共后缀下的 `wpad` 名字注册下来并指向自己的服务器；客户端一旦 devolution 越界，就会把 `wpad.dat` 的**内容归属**交给一个组织外的注册者。这类问题在文献里被归到 **W-Bad / WPAD 名称冲突攻击**一类：它不需要攻破 DNS，不需要伪造应答，只需要等客户端自己走完最后一跳，然后在返回的 PAC 里把感兴趣的目标指向攻击者控制的代理。具体事件年份、编号与命名细节**见官方资料**，本章不填不确定的数字。

这里的关键不是”某个后缀被抢注过”，而是模型本身：**配置内容的可信度，由一个可能被外部持有的名字决定**。DNSSEC 也无法救这一格——它保证应答没被篡改、来源可认证，但不保证”这个合法持有者就是你的网络管理员”。这与全书反复区分的两件事一致：DNSSEC 管数据完整性与来源认证，DoT / DoH / DoQ 管传输加密与链路隐私，谁都不管**命名归属的授权**。

23.4.3 三条缓解，按优先级排

1. **用内网保留域，并把所有派生名字本地接管**。客户端域名后缀收敛到自己能完全控制的区；Unbound 上用 `local-zone` + `local-data` 把 `wpad.< 内网域 >` 钉死，同时把 devolution 可能落到的外层候选名字也在本地判死，让”最后一跳”根本不出网。
2. **显式关闭 WPAD**。对不需要它的客户端，直接关掉系统/浏览器里的自动检测开关，比在 DNS 里围堵更彻底：没有发现流程，就没有 URL，也就没有取 PAC 这一步。开关位置见后文，具体路径随版本变化，以官方资料为准。
3. **不要依赖 search 域**。别把”客户端会拼出什么名字”寄托在 DHCP 下发的域名后缀上；后缀是输入，不是策略。需要固定行为时给显式 PAC URL 或静态代理配置。

三者的共同点是把**决定权从名字搬回配置**。第 1 条仍在 WPAD 框架内打补丁，第 2、3 条是直接绕开 WPAD——对遗留客户端之外的机器，后者才是本书推荐的取向。

23.5 现代化尝试：draft-joshco-wpadng-02

2024 年出现过 `draft-joshco-wpadng-02`：一份**个人草案**（非工作组立项），**同样已过期**。它更像一份”如果重做一遍该改什么”的设计笔记，值得读的是它点出的四个方向：

- 用 **DNS-SD** 替代广播式 / 猜测式的服务发现，让发现结果有明确的服务实例语义；
- 补上 **DHCPv6** 侧的配置途径，1999 年的草案只考虑了 IPv4 的 DHCP 选项；

- 引入 **urn:wpadng:none**，显式表达”本网络没有代理配置”这一状态，避免”找不到就猜、猜不到就沿用旧值”的含糊地带；
- **去掉 devolution**，从根上切断上一节那个越出组织边界的最后一跳。

这四个方向恰好对应本章前三节的每一处别扭：发现语义含糊、v6 缺席、失败状态未定义、名字归属失控。但它是过期个人草案，**没有任何实现承诺或互操作保证**，本书不落地、不写配置、也不建议拿它做选型依据。它的用处是当一面镜子：一份想修好 WPAD 的草案，要动的地方全在”自动发现”这件事本身。

23.6　PAC 层：FindProxyForURL() 与四个常用函数

23.6.1　函数与坑

PAC 来自 **Netscape 1996** 的代理自动配置约定，入口函数固定为 `FindProxyForURL(url, host)`，**没有正式 RFC**。它是一段受限的 JavaScript 环境：没有 DOM、没有网络 API，只有几个由宿主提供的辅助函数。

函数	作用	本章要留意的坑
`dnsResolve(host)`	把主机名解析成点分 IPv4 字符串，失败返回 `null`	每次调用都是一次额外 DNS 查询；它的解析路径未必与系统解析器一致
`isInNet(ip, addr, mask)`	判断 IP 是否落在点分十进制网段内	面向 IPv4 字面量设计；IPv6 判断 historically 弱，别依赖
`shExpMatch(str, pattern)`	shell 通配符匹配	只做字符串匹配，不校验 DNS 层级，也不是任何形式的来源认证
`myIpAddress()`	返回本机一个 IPv4 地址	多网卡、双栈、VPN 并存时返回值不确定

IPv6 支持弱不是一句带过的历史注脚，而是会直接改变分流结果：`dnsResolve()` 多数实现只返回 A 记录，`myIpAddress()` 只给一个 IPv4，`isInNet()` 的掩码写法也围着 IPv4 转。于是纯 IPv6 目标常常解析不到可用的点分地址，静默落到 `FindProxyForURL()` 的默认分支。写 PAC 时应当**显式处理解析失败**，而不是假设每个 `host` 都能变成 IPv4。

23.6.2　完整可复制的 wpad.dat

下面这份是教学用的 DIRECT / PROXY 分流示例，地址全部来自文档地址段，域名只有 `example.net` / `example.org`；`198.51.100.10:3128` 是示例代理端点，`192.0.2.10` 是提供 PAC 的站点地址。

```
function FindProxyForURL(url, host) {
    var DIRECT = "DIRECT";
    var PROXY  = "PROXY 198.51.100.10:3128";

    // 1) 内网保留域与本机名：直连，避免把管理流量绕进代理
    if (shExpMatch(host, "*.example.net") || shExpMatch(host, "example.net") ||
        shExpMatch(host, "*.example.org") || shExpMatch(host, "example.org") ||
        shExpMatch(host, "localhost")) {
        return DIRECT;
    }

    // 2) 名字 -> 地址；解析失败不硬送代理，显式处理 IPv6 弱支持的后果
    var ip = dnsResolve(host);
    if (ip === null) {
        return DIRECT;
    }

    // 3) 落在文档地址段内的目标直连
    if (isInNet(ip, "192.0.2.0",    "255.255.255.0") ||
        isInNet(ip, "198.51.100.0", "255.255.255.0") ||
        isInNet(ip, "203.0.113.0", "255.255.255.0")) {
        return DIRECT;
    }
```

```
    // 4) myIpAddress() 仅作条件分流演示：本机在示例网段时尝试代理
    if (isInNet(myIpAddress(), "192.0.2.0", "255.255.255.0")) {
        return PROXY + "; " + DIRECT;
    }

    return DIRECT;
}
```

为什么这么设：四个函数各出现一次，便于逐个验证；dnsResolve() 返回 null 时直接 DIRECT，是为了让”解析不到就不猜”成为显式策略，而不是掉进默认分支；文档地址段一律直连，保证示例环境里不会出现代理回环。

怎么验证：pactester -p ./wpad.dat -u https://example.org/index.html -c 192.0.2.20（参数以工具官方资料为准），期望输出 DIRECT；把示例域名换成清单外的名字再跑一次，检查是否按预期落到代理分支。

失败模式 / 回滚点：第 4 步的 PROXY ...; DIRECT 是**教学演示的回退链，不是生产写法**——代理不可用时它会静默直连，正是后文”缓存与降级”批评的 fail-open 行为。生产里应把它改成单一 PROXY 并接受失败，或干脆不用 PAC。回滚只需保留上一版 wpad.dat，按下一节的原子替换流程换回。

23.7 落地：把 wpad.dat 放到一个可控的 URL 上

23.7.1 nginx 提供 /wpad.dat

```
# /etc/nginx/conf.d/wpad.conf
server {
    listen 80;
    server_name wpad.example.net;

    location = /wpad.dat {
        default_type application/x-ns-proxy-autoconfig;
        alias /srv/wpad/wpad.dat;
        add_header Cache-Control "no-store, max-age=0" always;
        access_log /var/log/nginx/wpad-access.log;
    }

    location / {
        return 404;
    }
}
```

为什么这么设：default_type 直接给出 application/x-ns-proxy-autoconfig，不必往 types 里加映射；location = 精确匹配，避免别的路径误命中；no-store 压掉客户端与中间件的缓存；单独一份 access log 便于回答”到底哪些客户端来取过”。**这里用 80 端口不是建议**，而是因为 Well-Known Alias 流程默认就是明文 HTTP GET，改成只允许 HTTPS 会让一部分遗留客户端直接取不到——这个妥协本身就是后文”不推荐”的理由之一。

怎么验证：curl -I http://wpad.example.net/wpad.dat，应答里应有 200、Content-Type: application/x-ns-proxy-autoconfig 与 Cache-Control: no-store, max-age=0。

失败模式 / 回滚点：default_type 被上层 types 覆盖时客户端拿到 text/plain，部分实现会拒绝执行；alias 路径写错则 404，客户端表现为”自动检测开着但没有代理”。回滚：nginx -t 通过后再 nginx -s reload，并保留旧 wpad.dat 副本；最坏情况下注释掉该 server 块，让发现流程拿不到 URL。

23.7.2 Unbound 把名字收在本地

```
# /etc/unbound/unbound.conf.d/wpad.conf
server:
    local-zone: "example.net." static
    local-data: "wpad.example.net. 300 IN A 192.0.2.10"
    local-data: "wpad.example.net. 300 IN AAAA 2001:db8::10"

    # 演示：把不该外泄的 devolution 候选名字在本地判死
    # 此处必须换成你自己客户端会派生出的那些名字，照抄示例没有意义
    local-zone: "devolution-candidate.example.org." always_nxdomain
```

为什么这么设：`local-zone` 让 `example.net.` 成为本地接管的区，`local-data` 把 `wpad.example.net.` 同时钉到 IPv4 与 IPv6 文档地址；这样第 3 条发现机制解析出的地址始终指向自己那台 nginx，`wpad` 名字的归属不外包给公共命名空间。`always_nxdomain` 那一行演示的是”把外层落点也在本地判死”的思路——**它是占位示例**，名字写成一个明显不存在的标签以免被照抄；真实环境里应把客户端 devolution 会派生出的候选名字逐一枚举后本地接管。

怎么验证：`unbound-checkconf && unbound-control reload && dig @127.0.0.1 wpad.example.net A +short`，期望 `192.0.2.10`；再跑 `dig @127.0.0.1 wpad.example.net AAAA +short`，期望 `2001:db8::10`；最后查 `dig @127.0.0.1 wpad.devolution-candidate.example.org A +short`，期望无应答，说明外层落点没有出网。

失败模式 / 回滚点：只写 `local-data` 不写 `local-zone`，Unbound 会为这些数据隐含建一个 `transparent` 区，域内**其他**名字仍会走递归——devolution 的外层落点照样可能出网，这是最隐蔽的一种”配了等于没配”。反过来，`static` 区里没有本地数据的名字会被判无应答，若 `example.net.` 下还有别的业务名，会一起断掉；回滚把它改成 `transparent` 让未匹配名字落回递归。`local-zone` 各类型的完整语义见第 24 章，此处不展开。

23.7.3　生成脚本骨架：域名清单→ PAC 规则

手工维护 `shExpMatch` 列表迟早会漏项。骨架思路是：一份纯文本域名清单，生成一个只含直连规则的 `wpad.dat` 新文件，校验通过后再安装。

```
#!/usr/bin/env python3
""" 域名清单 -> PAC 规则（骨架）。示例地址仅为文档地址段。 """
from pathlib import Path

LIST = Path("/srv/wpad/domains.txt")      # 每行一个域名，# 开头为注释
OUT = Path("/srv/wpad/wpad.dat.new")     # 临时产物，校验后再 install
PROXY = "PROXY 198.51.100.10:3128"
DIRECT_SUFFIXES = ("example.net", "example.org")

def read_domains(path):
    for line in path.read_text(encoding="utf-8").splitlines():
        name = line.strip().lower().rstrip(".")
        if not name or name.startswith("#"):
            continue
        if "." not in name or len(name) <= 2:
            continue                      # 只收含点的整域名
        yield name

def render(names):
    patterns = list(dict.fromkeys(list(DIRECT_SUFFIXES) + list(names)))
    rules = "\n".join(
        f'        shExpMatch(host, "*.{d}") || shExpMatch(host, "{d}") ||'
        for d in patterns
    )
    return f"""// 由 {LIST.name} 生成，请勿手工编辑
function FindProxyForURL(url, host) {{
    var direct = "DIRECT";
    if (
{rules}
        false) {{
        return direct;
    }}
    return "{PROXY}";
}}
"""

if __name__ == "__main__":
    OUT.write_text(render(read_domains(LIST)), encoding="utf-8")
    print(f"written: {OUT}")
```

为什么这么设：输出到 `.new` 而不是线上文件，把”生成”与”生效”分开；清单里先过滤注释、空行、裸标签和短名，避免把脏数据拼进规则；末尾的 `false` 让拼接出来的条件链在任何清单长度下都语法合法。

怎么验证：`python3 /srv/wpad/gen-pac.py && pactester -p /srv/wpad/wpad.dat.new -u https://example.org/`，

确认输出 DIRECT；确认无误再 `install -m 0644 /srv/wpad/wpad.dat.new /srv/wpad/wpad.dat`。

失败模式 / 回滚点：清单里混入带路径或通配符的条目会生成永远不匹配的规则；空的 `domains.txt` 会生成”所有请求都走代理”的文件，等于把全量流量推给示例端点。回滚点就是上一版 `wpad.dat` 与 `.new` 并存：只要不执行 `install`，线上文件没有任何变化。原子替换与完整性校验的完整工程做法见第 24 章。

23.8 校验、开关与排障

23.8.1 pactester

`pactester`（pacparser 项目提供的命令行工具）在本地执行 PAC，不需要真的开浏览器、也不需要真的有代理在跑，适合把”文件逻辑对不对”和”发现流程对不对”切成两半验证。常用姿势是固定 `-u` 给一个 URL、必要时用 `-c` 指定客户端 IP，观察返回串：

```
pactester -p /srv/wpad/wpad.dat -u https://example.org/index.html -c 192.0.2.20
# 期望：DIRECT

pactester -p /srv/wpad/wpad.dat -u https://example.org/index.html -c 203.0.113.20
# 期望：按第 4 步条件分流的结果，用于确认 myIpAddress 分支
```

返回 DIRECT 不代表发现层成功，只代表**这份文件**对这组输入给出直连。工具的具体参数与安装方式**见项目官方资料**，本章不写死版本号。

23.8.2 浏览器与系统开关

位置	相关开关（名称随版本变化）	关闭方式
系统网络设置	“自动检测代理设置” / “自动代理发现”	取消勾选；企业环境用配置描述文件或组策略统一下发
浏览器连接设置	“使用系统代理设置” / “自动检测此网络的代理设置”	改为”不使用代理”或填显式 PAC URL
命令行与库	`http_proxy` / `https_proxy` 等环境变量	显式清空；这类客户端通常不读 WPAD

别把”关掉”当成一次性动作：新接入的终端、重装后的系统、临时借用的设备都会带着默认勾选回来。可行做法是把”自动检测关闭”写进终端基线，并在网关侧保留显式关闭的兜底（第 2 条缓解）。具体开关名、组策略项与配置描述文件字段**见官方资料**，不同版本差异较大。

23.8.3 日志与排障三板斧

按”发现层→传输层→配置层”的顺序查，不要跳步：

```
curl -I http://wpad.example.net/wpad.dat
# 看 HTTP 状态、Content-Type、Cache-Control；404 或 text/plain 都是配置层前置失败

dig @127.0.0.1 wpad.example.net A +short
dig @127.0.0.1 wpad.example.net AAAA +short
# 看名字是否被本地区接管；返回空或落到公共地址，说明 devolution 未被收住

grep 'wpad.dat' /var/log/nginx/wpad-access.log
# 看客户端到底来取过没有；没有这一行，问题在发现层而非 PAC 内容
```

三者对应三种完全不同的故障：`curl` 失败是服务端配置，`dig` 偏离预期是名字归属，`grep` 无记录是客户端根本没发起发现。浏览器开发者工具用于最后一步：在请求列表里确认代理连接实际连向哪个地址，以及 PAC 是否被重新取过（缓存命中时不会再次出现在日志里）。若怀疑 DHCP 252，则回到报文层面抓 OFFER / ACK / DHCPINFORM，这一层 `dig` 帮不上忙。

排障时最容易被误导的是**缓存**：改完 `wpad.dat` 而客户端毫无变化，多半不是”没生效”，而是它手里还拿着上一版。先清浏览器与系统的代理配置缓存、或用全新的无痕会话复测，再判断文件本身有没有问题。

23.9　为什么不推荐它

前几节把 WPAD 配到了能跑，这一节说明为什么本书不把它写进新架构。四条理由都是结构性的，不是”配置不够仔细”能补掉的。

23.9.1　默认开启，等于没有决策点

自动检测往往默认勾选。运维没有主动选择”启用 WPAD”，网络里出现一条 `wpad` A 记录或一个 252 选项，一批客户端的出向路径就变了。透明代理主线里，路径由网关策略决定、变更有配置与审计；这里路径由**客户端的一次名字查询**决定，变更点分散在 DHCP、DNS 和每台终端的开关上，很难给出一个”谁在什么时候改了它”的答案。

23.9.2　名称劫持：可信度押在名字归属上

从 `wpad.< 公共后缀 >` 到 W-Bad 类攻击，问题不是某个后缀没注册好，而是**配置内容的归属由 DNS 名字的归属决定**。devolution 又把这个归属边界推到组织之外。内网保留域 + 本地接管能把边界拉回来，但没改变”靠猜名字拿配置”这个模型；一旦有客户端绕过本地解析器、或带了别的 search 域，边界立刻重新打开。DNSSEC 也补不上这一格：它证明应答来自合法持有者，不证明合法持有者是你的管理员。

23.9.3　明文 HTTP 取 PAC

取 `wpad.dat` 的那一跳是明文 HTTP，链路上改内容不需要破解任何东西。PAC 不是数据，是**决定后续所有连接走向的程序**：改一行 `FindProxyForURL()`，就能把指定目标导向别处。业务连接上的 TLS 只保护业务连接，不保护”取配置”这一步；这与第 12 章讨论的加密解析是两码事，不要混为一谈。给 `wpad.dat` 上 HTTPS 也只能覆盖愿意配合的客户端，遗留实现仍会走 80。

23.9.4　缓存与降级，和 fail-closed 正面冲突

PAC 在浏览器与系统层都有缓存，`no-store` 只能约束自己这台 nginx 的应答，管不到下游各自的实现；改了文件，客户端可能一段时间内仍执行旧规则。更麻烦的是失败语义：发现不到 URL 时，有的客户端退回”无代理”，有的沿用上次配置；PAC 里写 `PROXY ...; DIRECT` 时，代理挂掉会静默直连。**同一个故障，不同客户端给出不同结果**，而且都是 fail-open。第三篇反复强调的 `forward-first: no`、严格档 fail-closed，在这里没有任何对应物——协议层就没定义”失败时该怎么办”。

23.9.5　收口

把四条并起来看：WPAD 把一个本该由网关集中决策的策略，拆成”名字归属 + 明文取件 + 客户端缓存 + 各家失败语义”四段，每段都能单独改变结果。透明代理不需要客户端配合，也不存在”谁来告诉它去哪里”这个问题；本书第五篇的网关策略、按 uid 豁免本机解析器出向查询那一套，才是主线。

因此本章的定位不变：**遗留客户端兼容**与**原理科普**。前者是为少数只会读自动配置的旧终端留一个受控的兼容口——内网保留域、本地接管、显式关闭其余终端、单独日志；后者是用它把”名字→配置→流量路径”这条链演示清楚。新架构的依赖图里不该有 WPAD；需要自动下发配置时，用范围明确、可审计、失败时行为确定的机制，而不是一次 `wpad.dat` 的猜测。

23.10　本章要点

- **两层必须分开**：WPAD 只负责找到 PAC 的 URL（DHCP 252、DNS、devolution），PAC 的 `FindProxyForURL()` 才决定 `DIRECT` / `PROXY`；”没找到 URL”与”文件判断错了”的日志、命令与回滚点完全不同。
- **标准状态要写准**：WPAD 从未成为 RFC，唯一定义是 WREC 工作组草案 `draft-ietf-wrec-wpad-01`（1999-07-29 发布、1999-12 过期）；§4.4 的 MUST / SHOULD 是草案内部分级，252 不在 [RFC 2132] 正式选项表，PAC 本身也没有正式 RFC。
- **devolution 会越出组织边界**：逐级剥左标签直到 TLD，最后一跳落到公共命名空间，于是 `wpad.< 公共后缀 >` 的注册者可以决定 PAC 内容（W-Bad 类攻击）；缓解靠内网保留域 + 本地接管、显式关闭自动检测、不依赖 search 域。
- **落地三件套**：nginx 以 `application/x-ns-proxy-autoconfig` 提供 `/wpad.dat`，Unbound 用 `local-zone` + `local-data` 把 `wpad.example.net` 钉在文档地址段，域名清单生成脚本产出 `.new` 后经 `pactester` 校验再 `install`；排障按 `curl -I` → `dig` → access log →开发者工具逐层推进。
- **不推荐作为新架构依赖**：默认开启、名称劫持、明文 HTTP 取 PAC、缓存与各家 fail-open 降级，都是结构性缺陷；本书主线是网关侧的透明代理，WPAD 只用于遗留客户端兼容与原理科普。

- 2024 年的 `draft-joshco-wpadng-02` 是已过期的个人草案，提出 DNS-SD、DHCPv6、`urn:wpadng:none` 与去掉 devolution；只作设计方向的记录，不落地、不做选型依据。

23.11 参考资料

- IETF Datatracker, `draft-ietf-wrec-wpad-01`（WPAD 唯一成文定义）: https://datatracker.ietf.org/doc/html/draft-ietf-wrec-wpad-01
- IETF Datatracker, `draft-joshco-wpadng-02`（2024 个人草案）: https://datatracker.ietf.org/doc/html/draft-joshco-wpadng-02
- RFC Editor, [RFC 2132] *DHCP Options and BOOTP Vendor Extensions*: https://www.rfc-editor.org/rfc/rfc2132.txt
- RFC Editor, [RFC 2608] *Service Location Protocol, Version 2*: https://www.rfc-editor.org/rfc/rfc2608.txt
- RFC Editor, [RFC 2219]（Well-Known Alias 的 DNS 别名约定）: https://www.rfc-editor.org/rfc/rfc2219.txt
- RFC Editor, [RFC 5737] *IPv4 Address Blocks Reserved for Documentation*: https://www.rfc-editor.org/rfc/rfc5737.txt
- NLnet Labs,Unbound 官方文档(`local-zone` / `local-data` 语义以此为准):https://unbound.docs.nlnetlabs.nl/en/latest/
- nginx 官方文档（`default_type`、`alias`、`add_header` 指令以此为准）: https://nginx.org/en/docs/

[RFC 2609] 与章内提到的 DNS SRV 记录类型原文，均在 RFC Editor 按编号索引，完整链接由第四篇统一 RFC 索引章收录；`pactester` 的安装与参数、浏览器与系统代理开关的具体名称与路径、W-Bad 类攻击的事件年份与编号，一律**见官方资料**。若 Datatracker 草案链接发生重定向，按草案名在 IETF Datatracker 检索即可。

第二十四章

Adhole：DNS 层过滤表的工程实践

把 58 万条域名塞进 `local-zone`，难点不在”怎么写一行配置”，而在两件事：**上游清单格式互不相通，转换错的代价是静默漏放**；以及 **Unbound 在哪一步判定这些名字**。前者决定表是否可信，后者决定表是否真的生效。本章把一套 Adhole 式过滤表从原始清单走到两台解析器上的全过程拆开，重点放在格式转换与 `local-zone` 的优先级。

最小暴露面见第 9 章，可观测性见第 7 章，DoH 端点分工见第 12 章。

24.1 能力边界：DNS 层只能表达”名字”

浏览器扩展里的 Adblock Plus / uBlock 规则看见的是完整 URL，Unbound 看见的只有 QNAME。这条差异不是”少几个字段”，而是一**整类规则在 DNS 层没有对应物**。写转换器之前必须把这类规则判为跳过，而不是”尽量近似”——近似的后果是把不该挡的域名挡了，或把该挡的域名放行。

规则形态	例子（教学用虚构域名）	能否表达	处理
整域名	`ads.example.net`	能	转成 `local-zone`
通配子域	`*.tracker.example.org`	部分能	剥 `*.` 后按父域建区，覆盖其下所有名字
URL 路径	`example.net/path/ad.js`	**不能**	跳过；QNAME 里没有路径
限定来源页	`||ads.example.net^$domain=news.example.org`	**不能**	跳过；`$domain=` 依赖被加载页面的域
元素隐藏	`##.ad-banner`	**不能**	跳过；这是 DOM/CSS 层
脚本注入	`##+js(...)`	**不能**	跳过；这是页面脚本层
白名单例外	`@@||allow.example.net^`	**不能**	跳过整条，别把 `@@` 误当允许项
IP / 端口	`0.0.0.0 ads.example.net` 里的地址	地址无意义	只取右侧域名

`@@` 尤其容易出错：它在浏览器规则里表示”这条不挡”，而 DNS 层没有”按请求上下文放行”的概念。把 `@@` 后的域名收进表，等于把白名单写反了。本项目单独维护 `unblock_domains.txt`，**在最终去重前应用**：先精确删除，再合并去重，这样白名单既不会被 cn/noncn 两份表重新拼回来，也不会因父域通配而漏掉。一句话记住边界：过滤表决定”这个名字要不要给答案”，不决定”这次请求从哪个页面来、URL 里有什么、页面上哪块元素要隐藏”。

24.2 六类上游格式逐条拆

所有格式最终都归一到同一形态（选 `always_null` 而非 `always_nxdomain`，是让客户端拿到 `0.0.0.0` / `::` 的空答而不是 NXDOMAIN；NXDOMAIN 常引发客户端重试、换搜索域甚至换解析器，语义见 24.6.2）：

```
local-zone: "ads.example.net." always_null
```

为什么这么设：include 展开后等价于文本原样插入，表文件里只能出现 server: 段内合法的语句；统一 FQDN 尾点与统一类型，才能用 unbound-control list_local_zones 直接核对命中了哪条。**怎么验证**：unbound-checkconf /etc/unbound/unbound.conf && unbound-control list_local_zones | grep 'ads.example.net'。**失败模式 / 回滚点**：混用 always_nxdomain / always_refuse 会让一张表出现三种应答语义，排障时无法用一条 dig 判断是谁挡的；回滚到上一版表并 reload 即可。

24.2.1　HOSTS 格式

```
0.0.0.0 ads.example.net
127.0.0.1 tracker.example.org
0.0.0.0 ads.example.net # 广告位
```

可提取信息：每行右侧第一个域名 token（一行多个域名则逐个取出）。丢弃项：左侧 IP（0.0.0.0 与 127.0.0.1 都只是 HOSTS 的本地重定向占位）、# 之后的注释。127.0.0.1 domain **不能**直译成指向回环的 local-data：那会让客户端连自己的 127.0.0.1，把”连接被拒绝”伪装成”服务存在”。输出：local-zone: "ads.example.net." always_null、local-zone: "tracker.example.org." always_null。

24.2.2　纯域名 TXT

```
ads.example.net
tracker.example.org
# 注释行
```

可提取信息：整行本身。丢弃项：# / ! 开头的注释、空行、行尾空白、不含点的裸标签（如 localhost）。这是最干净的一类，也是转换器应把其他所有格式**归一化到的目标形态**：先变成”一行一个域名”，再走同一段校验与输出。

24.2.3　Adblock Plus / uBlock 格式

```
||ads.example.net^
||tracker.example.org^$domain=news.example.org
||cdn.example.net/path/ad.js^
##.ad-banner
##+js(set-constant)
@@||allow.example.net^
```

原始行	判定	输出
||ads.example.net^	纯域名规则，收	local-zone: "ads.example.net." always_null
||tracker.example.org^$domain=news.example.org	含 $domain=，**跳过整条**	无
||cdn.example.net/path/ad.js^	含路径，**跳过整条**	无
##.ad-banner / ##+js(...)	元素隐藏 / 脚本注入，跳过	无
@@||allow.example.net^	白名单例外，跳过	无

”跳过整条”是关键：不要把 cdn.example.net 从带路径的规则里抠出来收进表，那会把整个 CDN 域挡掉；$domain= 的规则若只取左侧域名，就变成”对所有来源页都挡”，比原规则严格得多。

24.2.4　Unbound 原生格式

```
local-zone: "ads.example.net." always_null
local-zone: "tracker.example.org." always_nxdomain
```

可提取信息：引号里的区名。丢弃项：原类型（统一改写为 always_null）、local-zone: 关键字、多余空格。上游已给 always_null 也要**重新过一遍校验**：这类清单常混着别类型，混用会破坏单一应答语义。改写同时强制补齐 FQDN 尾点，避免 example.net 与 example.net. 在树里被当成两个东西。

24.2.5　威胁情报 TXT

```
# 恶意 C2 Indicator，教学示例
https://c2.example.org/beacon
```

```
http://198.51.100.7/payload
c2.example.org
```

可提取信息：URL 里的主机名。丢弃项：`http://` / `https://` 前缀、尾 `/` 及其后的路径、端口、查询串、纯 IP 行、注释。输出：`local-zone: "c2.example.org." always_null`。纯 IP 的 IOC 属于防火墙 / 代理层，不属于 DNS 过滤表：拼一条 `local-zone: "198.51.100.7." always_null` 是个永远查不到的名字，纯浪费表项。去协议后要先去尾 `/` 再取主机名，顺序反了会把斜杠留在域名里。

24.2.6 OISD 的 unbound 输出

OISD 等项目直接提供 unbound 格式清单，形态与 24.2.4 相同。可提取信息：引号内区名；丢弃项：文件头 `#` 注释与元数据行、非 `local-zone` 的杂项行。走同一个解析器即可，但**单独记一个来源标签**——`domain-state.tsv` 的 `category` 要能区分”这条来自哪个上游”，回滚时才能只撤一份清单。具体上游 URL 与清单内容会变动，教学脚本不依赖任何真实清单地址；接入时把下载好的本地文件放进 `sources/` 即可。

24.2.7 归一化六步

所有格式最后汇到这六步，顺序不能随意交换：① **去 `||` 与 `^`**（只处理 ABP 边界符号）；② **剥离 `*.`**（`*.tracker.example.org` → `tracker.example.org`，这是**降级为父域**，会覆盖其下所有子域；若上游本意只匹配一级子域，DNS 层无法精确表达，只能接受更宽的拦截）；③ **去 `http(s)://`**；④ **去尾 `/`**（先截断路径，只剩主机名）；⑤ **去 `www.`**（减少重复表项，但这是可逆性最弱的一步——`www.` 与裸域若由不同运营方托管，合并会让裸域也被挡，必须能从白名单回滚）；⑥ **小写化、去尾点、只保留含点且长度 > 2 的整域名**（每个标签只含字母数字与连字符、不以连字符开头/结尾，通不过的行计入跳过计数而不是静默丢）。`" 含点"` 排除 `localhost` 一类裸标签，`" 长度 > 2"` 排除 `a.b` 这类短串；两者只是粗筛，转换器不判断公共后缀，因为那需要额外的后缀列表，不在本书已核实范围内。

24.2.8 源格式→可提取信息→丢弃项→输出行对照表

源格式	可提取信息	丢弃项	输出行
HOSTS `0.0.0.0 domain`	右侧整域名	左侧 IP、`#` 注释	`local-zone: "ads.example.net." always_null`
HOSTS `127.0.0.1 domain`	右侧整域名	左侧回环地址（不转 `local-data`）	`local-zone: "tracker.example.org." always_null`
纯域名 TXT	整行域名	注释、空行、裸标签	`local-zone: "ads.example.net." always_null`
ABP `||domain.tld^`	`||` 与 `^` 之间的主机名	`||`、`^`	`local-zone: "ads.example.net." always_null`
ABP 带 `$domain=`	无（整条不可用）	`$domain=` 及左侧域名一并跳过	无输出
ABP 带路径 / `##` / `##+js` / `@@`	无（整条不可用）	整行	无输出
Unbound `local-zone: "..." always_null`	引号内区名	原类型、关键字、多余空格	统一改写为 `always_null`
威胁情报 URL	URL 中的主机名	协议、路径、端口、IP 行	`local-zone: "c2.example.org." always_null`
OISD unbound 输出	引号内区名	文件头注释、非 `local-zone` 行	`local-zone: "..." always_null`

24.3 可复制的转换骨架

教学骨架：只读本地 `sources/`，不访问网络，仅依赖 Python 标准库，输出归一化域名列表。

```python
#!/usr/bin/env python3
# normalize.py ——本地清单 → 一行一个域名；不依赖真实清单 URL
import re, sys
from pathlib import Path

SKIP_PREFIX = ("##", "@@", "!", "#")
LABEL = re.compile(r"^[a-z0-9](?:[a-z0-9-]{0,61}[a-z0-9])?$")
IPV4 = re.compile(r"^\d{1,3}(?:\.\d{1,3}){3}$")

def extract(line: str) -> str | None:
    """ 抽出候选主机名；DNS 层表达不了的规则整条返回 None。"""
    s = line.strip()
    if not s or s.startswith(SKIP_PREFIX):
        return None
    m = re.match(r'^local-zone:\s*"([^"]+)"', s, re.I)
    if m:
        return m.group(1)                    # Unbound 原生 / OISD
    if "##" in s or "+js" in s or "$" in s:  # 元素隐藏、脚本、$domain=
        return None
    if s.startswith("||"):
        s = s[2:].split("^", 1)[0]
    s = re.sub(r"^https?://", "", s, flags=re.I)
    if "/" in s:                             # URL 路径，整条跳过
        return None
    s = s.split("?", 1)[0].split(":", 1)[0]
    parts = s.split()                        # HOSTS：取右侧第一个域名
    if len(parts) >= 2 and IPV4.match(parts[0]):
        s = parts[1]
    s = s.split("#", 1)[0].strip().lower().rstrip(".")
    if s.startswith("*."):
        s = s[2:]
    if s.startswith("www."):
        s = s[4:]
    return s or None

def valid(host: str) -> bool:
    """ 只保留含点且长度 > 2 的整域名。"""
    if "." not in host or len(host) <= 2 or IPV4.match(host):
        return False
    return all(LABEL.match(l) for l in host.split("."))

def main() -> int:
    skipped, seen = 0, set()
    for p in sorted(Path("sources").glob("*.txt")):
        for line in p.read_text(encoding="utf-8", errors="replace").splitlines():
            h = extract(line)
            if h is None or not valid(h):
                skipped += 1
            else:
                seen.add(h)
    allow = {l.strip().lower() for l in
             Path("unblock_domains.txt").read_text(encoding="utf-8").splitlines()
             if l.strip() and not l.startswith("#")}
    for h in sorted(seen - allow):           # 白名单在最终去重前应用
        print(h)
    print(f"# skipped={skipped} kept={len(seen - allow)}", file=sys.stderr)
    return 0

if __name__ == "__main__":
    raise SystemExit(main())
```

为什么这么设：`extract()` 把六类格式压成一条归一化流水线，`$` 与 `/` 检查放在拆 `||` / `^` **之前或紧随其后**，保证带

`$domain=` 与带路径的规则整条返回 None 而不是被抠出主机名；`valid()` 只做与 DNS 相关的最少校验；`seen` 是集合，去重一次完成，避免 cn/noncn 各自去重后再合并时重复；纯 IP 的 IOC 计入 `skipped`，报告里能看见”丢了多少、为什么丢”。**怎么验证**：`mkdir -p sources && printf '0.0.0.0 ads.example.net\n||tracker.example.org^\n##.banner\n' > sources/demo.txt && : > unblock_domains.txt && python3 normalize.py` ——应只输出 `ads.example.net` 与 `tracker.example.org`，stderr 里 `skipped` 至少为 1。**失败模式 / 回滚点**：常见错是把”跳过”写成”提取”，例如只取 `cdn.example.net` 而丢掉路径；`valid()` 会放行这种错误，所以路径判断必须在 `extract()` 里。骨架只读 `sources/`、只写 stdout，不落盘，删掉输入文件即完全复原。

配套的 sed 骨架用于把已有 unbound 清单刷成统一类型（正式流程走上面的 Python）：

```
# 类型改写 + 补齐尾点 + 去重；输出只在 /tmp
sed -n 's/^local-zone:[[:space:]]*"\([^"]*\)"[[:space:]].*/local-zone: "\1." always_null/p' \
    sources/oisd-unbound.txt | sed 's/\.\."/."/' | sort -u > /tmp/adhole-normalized.conf
```

为什么这么设：`\([^"]*\)` 捕获引号内区名，输出统一补 FQDN 尾点，第二道 `sed` 修掉原本已带点时的双点，`sort -u` 是重复定义的最后防线（24.6.5）。**怎么验证**：`grep -c 'always_null' /tmp/adhole-normalized.conf ; grep -vc 'always_null' /tmp/adhole-normalized.conf` ——第二条应为 0。**失败模式 / 回滚点**：上游若用单引号或不带引号的区名会静默丢行、计数却不报错；输出只在 `/tmp`，`install` 前不影响运行中配置，正式流程以 Python 骨架为准。

24.4 输出三份表与 domain-state.tsv

文件	规模	内容
`adhole-cn.conf`	约 6k 条	归类为 cn 的域名
`adhole-noncn.conf`	约 27MB	归类为 noncn 的域名
`adhole-full.conf`	约 28MB / 58 万条	cn + noncn 合并去重后的完整表
`domain-state.tsv`	与表项同量级	状态与分类依据

`adhole-full.conf` 是两台解析器实际 `include` 的那一份；cn / noncn 分开保留，是为了按来源灰度与单独回滚。三份都写成 24.2 开头那种单一行形态，不设表头。`unbound-checkconf` 能查语法，但**不校验语义重复**：同名区出现两次它照样通过；若表文件里混进 `local-zone: "example.net." always_null`，会把本章验收用的示例域一起挡掉。回滚：保留上一版 `adhole-full.conf`，`install` 回退后 reload（24.9）。

`domain-state.tsv` 五个字段固定，以制表符分隔：

```
domain  category    last_seen   last_checked    ip
ads.example.net noncn   2026-11-07  2026-11-07  192.0.2.7
tracker.example.org cn  2026-11-07  2026-11-07  198.51.100.9
```

`domain` 是归一化后的整域名，与表项一一对应作主键；`category` 为 `cn` / `noncn` / 来源标签，决定写入哪份表、回滚时撤哪份清单；`last_seen` 是该域名最近一次出现在上游清单的日期，用于老化下线；`last_checked` 是最近一次主动解析校验的日期——**出现≠仍可解析**，两者必须分开；`ip` 是解析得到的 A 记录地址，既是分类依据也是误伤排查线索。上面的日期与地址均为教学示例，实际值由生成时的解析结果写入。

为什么这么设：五个字段把”表项是什么、为什么归这一类、上一次证据何时取”分开存，`unbound-checkconf` 只认表文件、不读 TSV，因此 TSV 的格式错误不会影响解析、却会让回滚与误伤排查失去依据；固定制表符分隔是为了能用 `awk -F'\t'` 直接查。

怎么验证：`unbound-checkconf /etc/unbound/unbound.conf && grep -c '^local-zone:' /etc/unbound/adhole-full.conf`（应接近 58 万），再用 `unbound-control list_local_zones | grep -c always_null` 核对生效条目数。**失败模式 / 回滚点**：`domain-state.tsv` 是生成物，每次生成保留带日期副本（24.11），回退到上一版即可恢复分类。

24.5 分类靠解析 A 记录，不靠域名后缀猜

`category` 的判定流程是：**对每个候选域名做一次 A 记录解析，拿到应答里的 IP，再对照本地 `geoip-cn.json` 判断该地址是否属于 cn 网段**。命中则 `cn`，否则 `noncn`；解析失败（NXDOMAIN / SERVFAIL / 超时）的条目记入单独状态，下一轮 `last_checked` 再试，不要凭后缀猜一个分类塞进去。

为什么要绕这一圈：域名后缀是**注册管理**信息，IP 归属是**托管位置**信息，两者没有必然关系。以 .cn 结尾的域名可以托管在境外，以通用后缀结尾的域名可以托管在境内 CDN；某些服务还会按解析来源返回不同 IP，后缀完全不携带这一层信息。过滤表要分的是”这一批域名由哪份清单负责、回滚时撤哪一份”，按后缀猜会让分类与实际托管脱节，灰度时看不出影响面。

三个约束：①解析必须走**受控的递归解析器**（第 4–6 章的实例），不要用本机存根解析器随手 getaddrinfo——后者可能被搜索域、mDNS、/etc/hosts 污染；② geoip-cn.json 按网段前缀匹配，自身要有版本与更新记录以保证比对可复现，具体数据来源与格式见官方资料，本章不假定任何特定提供方；③只重解析 last_checked 超阈值的条目，58 万条全量每天重查既慢又给上游制造无意义负载。

怎么验证：awk -F'\t' 'NR>1 && ($5 == "" || $4 == "") {print}' domain-state.tsv | head 应无输出；再抽检若干 cn 条目的 ip 是否落在 geoip-cn.json 的 cn 段。**失败模式 / 回滚点**：上游临时 NXDOMAIN 会把域名误判成”无分类”，若转换器直接丢弃就产生静默漏放；正确做法是保留条目、标记待复查、本轮沿用上一次 category。回滚：回退上一版 domain-state.tsv。

24.6 local-zone 与 forward-zone 的优先级（本章核心）

一个看似简单、最容易配错的问题：**被写进过滤表的名字，会不会先被送到上游 forward-zone？** 不会。

24.6.1 查询处理顺序

① 收包 → access-control（谁可以问；最具体网段匹配，无匹配即 refuse）
② local-zone 树查找（按最具体的区名匹配；tag / view 在此层参与）
③ 命中且类型要求本地应答 → 直接生成权威应答，返回客户端
④ 未命中 / 类型为 transparent 类 → 查缓存
⑤ 缓存未命中 → RPZ QNAME 触发器（在 local-zone 之后、auth-zone 之前）
⑥ auth-zone / forward-zone / stub-zone / 递归 → 出向查询
⑦ 应答回来 → 缓存 → 返回客户端

官方手册明确写了三条次序：权威区在 for-downstream: yes（默认）时”在 local-zones 之后、cache 之前处理”；RPZ 的 QNAME 触发器”在任意 local-zone 之后、任意 auth-zone 之前应用”；auth-zone 的 for-upstream 阶段在缓存查找之后、出网之前。合起来就是上面的顺序。

结论：**local-zone 匹配发生在递归与转发之前**。always_null 在②命中后于③直接返回 0.0.0.0 / ::，查询根本不会抵达⑥的 forward-zone。过滤表不是一个”应答回来后再改写”的钩子，而是查询路径上的提前出口，所以它能零上游查询地生效。由此得分工：**forward-zone 只对未被本地区接管的名字生效**。写 forward-zone: "." 把一切交给单一上游（第 5 章的单上游约束）并不意味着过滤表失效；恰恰相反，命中表的名字在到达这个 forward-zone 之前就被终结了。反过来，既没进表、又被 transparent 类区放过的名字才会真正走到上游。

24.6.2 always_* 与 static / transparent 的语义表

local-zone 第二个参数决定”命中后若无 local-data 精确匹配，给什么应答”：

类型	命中后行为（无 local-data 精确匹配时）	local-data	适用场景
always_null	对区内每个名字无条件返回 0.0.0.0 / ::	忽略	**过滤表默认类型**；空答对客户端最友好
always_nodata	无条件 NOERROR/NODATA（名字存在、该类型无数据）	忽略	只想让 A/AAAA 查不到、又不想给 NXDOMAIN
always_nxdomain	无条件 NXDOMAIN	忽略	明确宣告”此域不存在”，可能触发重试
always_transparent	按正常流程向上解析	忽略	临时关掉这个区的本地接管
always_refuse	返回 REFUSED，连该区名的 DS 查询也挡	忽略	强阻断；手册提示会打断 DNSSEC 信任链
always_deny	静默丢包	忽略	不推荐：DNS 不擅长按策略丢包，引发重试
static	有 local-data 就答，否则 NODATA / NXDOMAIN（带 SOA）	使用	本地权威小 Zone，名字固定

类型	命中后行为（无 local-data 精确匹配时）	local-data	适用场景
transparent	有 local-data 就答；**不同名字**正常解析；同名不同类型回 NODATA	使用	只覆盖少数名字、其余照常上网
typetransparent	有 local-data 就答；不同名字**或同名不同类型**都正常解析	使用	只覆盖 A，不希望 AAAA 被 NODATA 连带影响

最后两行的差异在双栈环境里是实打实的：用 transparent 给 gw.example.net. 写了 A 记录，客户端问 AAAA 会收到 NODATA；用 typetransparent 则正常向上解析 AAAA。always_* 前缀的含义是**忽略 local-data**；手册对 always_null 的原文是”对区内每个名字总返回 0.0.0.0 或 ::0，类似带零数据的 redirect，忽略区里的 local data，用于某些屏蔽清单”——这正是本章选它作表项类型的依据。

24.6.3 local-data 与 local-zone 配合

local-data 提供精确的资源记录，local-zone 决定”没精确匹配时怎么办”。两条边界：若只写 local-data 而不写覆盖它的 local-zone，手册说明 Unbound 会自动创建 transparent 区，所以**“没写区”不等于”没接管”**，而是以最宽松的类型接管；always_null / always_nodata / always_refuse 等忽略 local data，在这些区里配 local-data 是无效的。想让某个被屏蔽域返回特定地址，应改用 redirect 或 static + local-data 并把它从过滤表移除。常见需求”挡掉广告域但 sso.example.net 要能解析”的正确做法，不是在 always_null 区里塞 local-data，而是把该域放进 unblock_domains.txt（24.1），或给父域用 typetransparent 配例外；前者更简单可审计，是本项目默认路径。

24.6.4 access-control 与 local-zone 的先后

access-control 在①、local-zone 在②，因此**访问控制先于本地接管**：被 deny / refuse 的源地址连本地应答都拿不到，谈不上”命中过滤表”。这解释了一个常见疑惑——从没放行的网段 dig 一个被屏蔽域，看到的是 REFUSED 而不是 0.0.0.0，别据此判断过滤表没生效。手册给出的 deny_non_local / refuse_non_local 只允许查询权威 local-data、不给递归，适合”只想让某网段拿到本地静态应答”；普通 allow 对非递归查询也允许访问 local-data。access-control 按**最具体网段**匹配且与语句顺序无关，这与 include 顺序的规则正相反，两者不要混记。完整基线见第 9 章。

24.6.5 include 顺序与重复定义

include: 可出现在配置任意位置，语义是”把被包含文件的文本像复制粘贴一样插到这一点”，也支持通配符。三条工程结论：① **段上下文会延续**——include 写在 server: 段中间，表文件里的裸 local-zone: 行才属于 server:；写在 forward-zone: 段之后会被解析进错误的段。稳妥做法是用 include-toplevel:（手册说明它会先关闭当前段、强制被包含文件自带段声明），或确保 include 位于 server: 段内。② **chroot 下路径必须可见**，相对路径依赖启动目录 / directory: 设置，配合 AppArmor 还要求落在白名单内（24.9）；第 7 章已给过”三处同时满足”的约束，这里不重复。③ **同名区重复定义不应依赖”谁生效”**：手册对 unbound-control local_zone 的说明是”若该区已存在，类型被改为给定参数”，即后写入者覆盖先前类型；但把 58 万行表的正确性寄托在覆盖顺序上是脆弱的，unbound-checkconf 不报重复、日志也不一定显眼。正确做法是生成器里用集合去重、sort -u 收尾，保证每份表内每个区名只出现一次；跨文件重复（例如 adhole-full.conf 与手工例外文件同时定义同一域）用 list_local_zones 核对实际生效类型。一句话：include 顺序决定”哪些语句在哪个段、哪份表先被读到”，但**业务上的覆盖关系由生成器保证，别靠配置顺序碰运气**。

24.6.6 auth-zone 与 local-zone 的边界

auth-zone 是从文件加载的权威区副本，两个可同时开启的阶段：for-downstream: yes（默认）在 local-zone 之后、cache 之前，让 Unbound 像权威服务器一样直接回答下游；for-upstream: yes（默认）在缓存查找之后、出网之前，供递归过程查本地副本。它与 local-zone 位置相邻但职责不同：**local-zone 是策略（该不该答、答什么），auth-zone 是数据（一整份区文件内容）**。手册另说明：需要带引荐、通配、CNAME/DNAME、DNSSEC 权威服务的复杂权威数据，应当用 stub-zone 而不是 local-zone。所以第 6 章的本地根区镜像用 auth-zone，本章的过滤表只承担”按名字给空答”，别因为”都叫 zone”就混用。forward-zone 与 auth-zone 的区别也清楚：前者把查询交给另一台**递归**服务器，后者提供本地区文件内容；二者都在⑥附近参与，而 local-zone 在②就可能终结查询。

24.6.7　RPZ 与 local-zone 的区别

RPZ（Response Policy Zone，响应策略区）把策略写成标准 DNS 区文件格式，可通过区传送从外部来源加载，不同解析器实现可共用；local-zone 是 Unbound 专有语法。官方手册给出的次序是 **RPZ QNAME 触发器在任意 local-zone 之后、任意 auth-zone 之前应用**，且 RPZ 依赖 respip 模块、需在 module-config 中加载。因此同时配了二者的实例里 local-zone 先判，被 always_null 终结的名字不会再经过 RPZ。RPZ 的优势是标准化、可传送、能按应答 IP / 客户端 IP 等更多触发器做策略，代价是多一层模块与区文件管理；本项目选 local-zone 大表，理由是生成链路全在本地、可原子替换、可按来源灰度。触发器精确匹配语义、动作类型与加载顺序以官方手册 RPZ 章节为准。

24.6.8　用 dig 证明”命中 local-zone 时上游零查询”

```
# ① 表是否已加载
unbound-control list_local_zones | grep 'ads.example.net'
# ② 记录统计基线
unbound-control stats_noreset > /tmp/adhole-stats-before.txt
# ③ 只查被屏蔽的名字，连查 5 次
for i in 1 2 3 4 5; do dig @192.0.2.1 ads.example.net A +short; done
# ④ 对比出向计数
unbound-control stats_noreset > /tmp/adhole-stats-after.txt
diff /tmp/adhole-stats-before.txt /tmp/adhole-stats-after.txt | grep -E 'cachemiss|queries|recursivereplies'
# ⑤ 对照组：查未进表的名字
dig @192.0.2.1 open.example.org A +short
unbound-control stats_noreset | grep -E 'total.num.cachemiss|total.num.queries'
```

为什么这么设：always_null 生成的是本地权威应答，不进入递归路径，因此 total.num.cachemiss 与 total.num.recursivereplies 不应因③增长；total.num.queries 会增长，因为查询确实被收下了——这正是要区分的两个量。手册说明 recursivereplies 是”发给需要递归处理的查询的应答数”，缓存应答不计入，所以它是比”延迟”更硬的证据。⑤的对照组证明统计本身是活的、不是计数器没动。**怎么验证**：③的五次应答都应是 0.0.0.0；④的 cachemiss 增量为 0；⑤之后 cachemiss 至少 +1。**失败模式 / 回滚点**：若实例之前查过这个被屏蔽名字、缓存里有条目，cachemiss 本来就不会涨，容易得出假阳性，所以要先确认是冷查询（多实例链路上逐层刷缓存的要求见第 7 章，这里不重复配置）；另一个坑是 dig 打到了别的实例，用 unbound-control -c < 配置 > 或核对 @ 后的地址与端口。这组命令只读统计、不改配置，唯一副作用是 dig 本身产生的查询记录。

补充定性验证：在受控上游侧临时观察这几个名字的查询是否出现；dnstap 的采集与保留期配置在第 7 章，这里只把它当**证据来源**。

24.7　原子生成：临时文件 + 校验 + install

58 万行的表绝不能”边下载边覆盖正在被读的文件”。完整性校验的最低要求是：**最后一行必须是合法的 local-zone 行**。这一条能抓住绝大多数”下载中断 / 磁盘写满 / 管道被 head 截断”的不完整表——截断在行边界时 unbound-checkconf 语法仍合法，却会静默漏放大批域名。

```
#!/usr/bin/env bash
# build-adhole.sh ——输入是本地 sources/ 与 unblock_domains.txt，不访问网络
set -euo pipefail
OUT=/etc/unbound/adhole-full.conf
TMP=$(mktemp /etc/unbound/.adhole-full.XXXXXX)
trap 'rm -f "$TMP" /tmp/adhole-names.txt' EXIT

python3 normalize.py > /tmp/adhole-names.txt
awk '{printf "local-zone: \"%s.\" always_null\n", $0}' /tmp/adhole-names.txt > "$TMP"

# 完整性校验：非空、行数下限、最后一行必须是合法 local-zone
[[ -s "$TMP" ]]
[[ $(wc -l < "$TMP") -ge 500000 ]]
tail -n 1 "$TMP" | grep -Eq '^local-zone: "[a-z0-9.-]+\." always_null$'

install -m 0644 -o root -g root "$TMP" "$OUT"
unbound-checkconf /etc/unbound/unbound.conf
unbound-control reload || systemctl restart unbound
echo "installed $(grep -c '^local-zone:' "$OUT") zones"
```

为什么这么设：mktemp 把临时文件放在同一文件系统，install 的落位才可能是原子替换；set -euo pipefail 让任何一步失败都中止，绝不把半成品推到 OUT；trap 保证异常退出时清掉临时文件。tail -n 1 | grep -Eq 是本章刻意要求的那道校验，它验证的是”文件被完整写完”这个 unbound-checkconf 不负责的性质；install -m 0644 -o root -g root 明确权限与属主，配合 24.8 的 AppArmor 白名单。**怎么验证**：ls -l "$OUT" ; tail -n 1 "$OUT" ; unbound-control list_local_zones | grep -c always_null，三处形态与数量应一致；再跑 24.6.8 的 dig 组。**失败模式 / 回滚点**：wc -l 的 500000 是本项目 58 万条口径的宽松阈值，上游规模变化时会误报”表变小”，此时应人工确认而不是直接放宽；unbound-checkconf 只查语法、不查重复定义与语义。回滚：保留上一版 adhole-full.conf（24.11），install 回退旧文件后 reload；手册列出 interface 等少数选项 reload 不生效，local-zone 本身可 reload，但混在其他变更里时 restart 更稳。

24.8 大表的内存代价、加载时间与 AppArmor

local-zone 大表**全量常驻内存**：Unbound 把区名组织成一棵树，每个节点要存标签、类型、可选 tag / local-data 指针，再加分配器开销。磁盘上约 28MB，进内存后远不止 28MB——一行约五十字节的文本变成结构体、树指针与对齐填充，量级通常按数倍估算；具体数值取决于版本、编译选项与是否启用 tag/view，**不要照搬任何固定倍数，以本机基线为准**。三个可观测代价：启动 / reload 时解析 58 万行建树，耗时明显长于小配置；手册说明 fast_reload 期间需**约两倍**配置内存（新旧两份并存），峰值可能触发 OOM；稳态下与缓存争内存，缓存尺寸需相应下调。

怎么验证：unbound-control stats_noreset | grep -E 'mem|queries' ; ps -o rss= -p "$(pidof unbound)" ——改动前后各取一次基线，记录 RSS 增量与启动耗时；给大表留出的内存要从缓存预算里扣，不能两套都按满配（缓存尺寸与线程/slabs 规划见第 7 章）。另有一个隐蔽成本：每次查询都要在这棵树里做最具体匹配，它与缓存命中、RPZ 判定、日志采集一起构成每查询固定开销；加载大表前后各测一次命中数与 recursion.time.median，才能把”变慢”归因到具体环节。**失败模式 / 回滚点**：最危险的不是慢，而是 fast_reload 的双倍内存峰值触发 OOM，表现为进程被杀、反复重启；内存紧张时宁可低峰期 restart（restart 不需要同时持有新旧两份配置）。回滚：回退上一版表或先只加载 adhole-cn.conf（约 6k 条）观察，再逐步放开。

第 9 章已把 AppArmor 基线讲透：**profile 只允许 Unbound 读 /etc/unbound/****，把表文件放到 /var/lib/unbound/ 或 /tmp 会被 DENIED，而 unbound-checkconf 照样通过——它不检查内核的强制访问控制判决。本章补三条：临时文件必须在 /etc/unbound/ 下（mktemp /etc/unbound/.adhole-full.XXXXXX）；install 后属主 / 权限要与 profile 一致，profile 允许的读路径与 Unix 权限是两道独立的门；排查顺序固定为 unbound-checkconf → journalctl -k | grep DENIED →文件路径→ include 顺序，不要先怀疑上游或表内容。**怎么验证**：unbound-checkconf /etc/unbound/unbound.conf ; journalctl -k --since '10 min ago' | grep -i denied || true ; aa-status | grep unbound ——语法通过、无 DENIED、进程处于 enforce 模式，三者齐备才算加载成功。**失败模式 / 回滚点**：reload 后 list_local_zones 为空且日志无报错，第一嫌疑是 DENIED 被 audit 级别吞掉，第二嫌疑是 include 写在错误的段里（24.6.5）；把表移回 /etc/unbound/** 并重启，不要为此放宽 profile，那等于把第 9 章的最小暴露面拆了。

24.9 定时同步到另一台解析器

两台解析器（家庭网关 gw-01 与云主机 edge-01）用同一张表，靠定时同步保证口径一致。

```
#!/usr/bin/env bash
# sync-adhole.sh ——在生成端执行；目标主机名与密钥均为教学示例
set -euo pipefail
SRC=/etc/unbound/adhole-full.conf
STATE=domain-state.tsv
tail -n 1 "$SRC" | grep -Eq '^local-zone: "[a-z0-9.-]+\." always_null$'
# 先推到目标端的暂存路径，校验通过后才覆盖 include 指向的路径
scp -q "$SRC" "edge-01:/etc/unbound/.adhole-full.incoming"
scp -q "$STATE" "edge-01:/etc/unbound/domain-state.tsv"
ssh edge-01 'tail -n 1 /etc/unbound/.adhole-full.incoming | grep -Eq "^local-zone: .* always_null$" && \
             install -m 0644 /etc/unbound/.adhole-full.incoming /etc/unbound/adhole-full.conf && \
             rm -f /etc/unbound/.adhole-full.incoming && \
             unbound-checkconf /etc/unbound/unbound.conf && \
             systemctl restart unbound'
```

为什么这么设：**被覆盖的 include 路径必须先讲清**——目标端主配置里 include: "/etc/unbound/adhole-full.conf" 是唯一入口，scp 先落到 .adhole-full.incoming 暂存，校验通过后由 install 覆盖的正是这个路径；edge-01 上不要另有一份手工维护的同名文件，也不要让 include 用通配符把备份（adhole-full.conf.bak.*）一起读进来，否则出现 24.6.5 的重复定义。同步的是**生成物**而不是 sources/，口径才不会因两端脚本版本不同而漂移；domain-state.tsv 一并

同步，让目标端具备同样的误伤排查线索。目标端先在暂存路径上查最后一行，再 `install` 覆盖 `include` 指向的目标、固定权限，`unbound-checkconf` 验语法，最后 **restart**（`fast_reload` 大表时的双倍内存峰值见 24.8，同步窗口宁可短时重启）；顺序反了会留下属主错误或半截文件，且 `restart` 才能让新表稳定生效。凭据用 `<TOKEN>` 或等价密钥认证，不写明文口令；定时用 systemd timer 或 cron 均可，间隔与上游清单更新节奏一致。**怎么验证**：`ssh edge-01 'unbound-control list_local_zones | grep -c always_null' ; unbound-control list_local_zones | grep -c always_null` 两端计数应一致，再各自跑 24.6.8 的被屏蔽名字，都应答 `0.0.0.0`。**失败模式 / 回滚点**：网络中断会让 `scp` 写出半截文件，所以先推暂存路径、校验通过后才 `install` 到被 `include` 的路径；两端版本不一致时先让目标端回退上一版表（24.11），不要在线修补。回滚：目标端 `install` 旧备份并 restart；若只有一端失败，暂停同步任务，避免下一轮把好端也覆盖。

24.10　误伤排查、白名单回滚点与保留期

过滤表最常见的生产事故不是”没挡住”，而是”挡错了”。排查四步：① **dig 定性**——`dig @192.0.2.1 sso.example.net A +short` 若返回 `0.0.0.0`，先确认是表在起作用而不是源站真挂了，再用 `unbound-control list_local_zones | grep sso.example.net` 查是精确区还是父域区命中；② **dnstap 找证据**——看这次查询的 QNAME、类型、客户端地址与应答，确认哪一层给的空答，采集配置见第 7 章；③ **定位来源**——`grep sso.example.net domain-state.tsv` 看 `category` 与 `ip`，再回查 `sources/` 里是哪份清单引入了它（24.2.6 的来源标签在这里派上用场）；④ **白名单回滚**——把域名加进 `unblock_domains.txt`，重跑 24.3 生成器与 24.7 原子落位。因为白名单在**最终去重前**应用，重生成后它会从三份表里一起消失，不会被 cn/noncn 合并拼回来。

```
# 误伤应急：写白名单并重生成（教学示例）
printf 'sso.example.net\n' >> unblock_domains.txt
bash build-adhole.sh
dig @192.0.2.1 sso.example.net A +short
```

为什么这么设：应急改动只落在 `unblock_domains.txt` 一个文件，重生成走与日常完全相同的流水线，不引入”手工改一行表文件”这类无法复现的状态；`dig` 复核放最后，避免”白名单写了但没生效”被当成已修复。**怎么验证**：`grep sso.example.net /etc/unbound/adhole-full.conf` 应无输出，`unbound-control list_local_zones | grep sso.example.net` 应无输出，`dig` 应返回真实应答。**失败模式 / 回滚点**：reload 后链路上其他实例可能仍留有旧应答，要按第 7 章逐层确认缓存；父域级拦截（`*.` 降级或 `www.` 归一化）造成的误伤，加精确白名单未必够，需要把父域一并加入或改回更窄的清单。回滚点两处：`unblock_domains.txt` 去掉该行即恢复拦截，`adhole-full.conf` 回退上一版即整体撤销本轮更新。

24.10.1　保留期与版本化

产物	保留策略	理由
`adhole-*.conf` 历史版本	按日期保留有限份（如最近 7 天 / 若干版），到期删除	回滚点要够用，但不能成为无期限的域名画像存档
`domain-state.tsv` 历史版本	同上；`last_seen` / `last_checked` 用于老化下线	长期不再出现的条目应退出表
dnstap 全量查询日志	**30 天**，配到期删除	与第 7 章一致；查询流副本最敏感
`sources/` 上游清单快照	与表版本对应保留	生成可复现，才谈得上审计

版本化做法：`install` 前把当前 `OUT` 复制成带日期的备份，备份文件本身**不要被 `include` 的通配符扫到**（24.9）。每次生成写一份 manifest，记录表行数、跳过计数、三份表规模、`geoip-cn.json` 版本与生成时间；误伤排查先看 manifest，能立刻判断是不是这一轮引入的。`last_seen` 的老化要保守：连续多轮未出现在上游清单才从表中移除，移除同样走正常重生成流程并保留旧版本以便回滚。不要因为怕漏放就永远不删条目——表越大，24.8 的内存与加载代价越高，误伤面也越宽。

24.11　与第三篇的三处呼应

- **第 9 章最小暴露面**：过滤表是”答什么”的策略，`access-control` 是”谁可以问”的门，门在策略之前（24.6.4）。表文件放在 `/etc/unbound/**` 内是 AppArmor 白名单的硬要求；任何为图方便放宽 profile 的做法都等于把最小暴露面拆掉。
- **第 7 章可观测性**：dnstap 是自己的查询流副本，本章只用它做误伤证据与”上游零查询”的定性验证，采集开关、六类消息与 30 天保留期一律沿用第 7 章；大表加载前后的 RSS、命中数与 `recursion.time.median` 基线也沿用该章方法。

- **第 12 章 DoH 端点与过滤表的关系**：DoH 只是把 DNS 报文装进 HTTPS，**不改变解析器内部的查询处理顺序**。第 12 章的分工是 dnsdist / nginx 做前端、Unbound 只监听回环做递归验证；只要客户端最终落到这台 Unbound，`local-zone` 就在递归/转发之前生效，过滤表照常工作。反过来，若浏览器绕过本地解析器直连第三方 DoH 端点，过滤表完全看不见这次查询——那属于第 13 章的 DDR 压制与 canary 域要解决的问题，不是把表加大能补救的。这里也再次体现全书一贯区分：DoH / DoT 是**传输加密**，DNSSEC 是**数据完整性与来源认证**，过滤表是**本地策略**，三者互不替代。

24.12 本章要点

- **DNS 层只能按名字拦截**：URL 路径、`$domain=`、元素隐藏 `##`、脚本 `##+js`、白名单 `@@` 都表达不了，必须整条跳过而不是近似提取；`unblock_domains.txt` 在最终去重前应用，避免白名单被 cn/noncn 合并拼回来。
- 六类上游格式（HOSTS 两种写法、纯域名 TXT、ABP `||domain.tld^`、Unbound 原生、威胁情报 TXT、OISD unbound 输出）统一归一为 `local-zone: "domain." always_null`；顺序是去 `||`/`^` →剥 `*.` →去 `http(s)://` →去尾 `/` →去 `www.` →只保留含点且长度 > 2 的整域名。`always_null` 忽略 `local-data`，例外要走白名单。
- **`local-zone` 匹配发生在递归与转发之前**：`always_null` 在访问控制之后直接返回 `0.0.0.0` / `::`，不会再走 `forward-zone`；`forward-zone` 只对未被本地区接管的名字生效。用 `stats_noreset` 的 `cachemiss` / `recursivereplies` 前后对比加未命中对照组，可以证明”命中时上游零查询”。
- `always_nodata` / `always_transparent` / `always_refuse` / `always_deny` 都忽略 `local-data`；`static` 无匹配给 NODATA/NXDOMAIN，`transparent` 与 `typetransparent` 的区别在”同名不同类型”是否继续向上解析。`include` 是文本插入、同名重复别指望覆盖顺序，靠生成器集合去重；RPZ QNAME 触发器在 `local-zone` 之后、`auth-zone` 之前。
- 三份表（cn 约 6k 条、noncn 约 27MB、full 约 28MB / 58 万条）与 `domain-state.tsv` 五字段配套；分类靠**解析 A 记录并对照 `geoip-cn.json`**，不靠域名后缀猜，因为后缀是注册管理信息、IP 才是托管位置信息。
- 工程闭环：临时文件 + 校验最后一行合法 `local-zone` + `install` 原子落位；AppArmor 只许读 `/etc/unbound/**`；58 万行表全量常驻内存、`fast_reload` 有约两倍配置内存的峰值；同步时讲清被覆盖的 `include` 路径；误伤走 `dig` → dnstap →白名单重生成；历史表、状态、dnstap 各有保留期（查询流 30 天）与版本化回滚点。

24.13 参考资料

- Unbound 官方手册，`unbound.conf(5)`（`local-zone` 各类型、`local-data`、`include:` / `include-toplevel:`、`access-control`、`forward-zone`、`auth-zone`）：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html
- Unbound 官方手册，Response Policy Zones 章节（RPZ 触发器次序、`respip` 模块、与 `local-zone` 的关系）：https://unbound.docs.nlnetlabs.nl/en/latest/topics/filtering/rpz.html
- Unbound 官方手册，`unbound-control(8)`（`list_local_zones`、`local_zones` / `local_zones_remove`、`stats_noreset`、`fast_reload` 的内存说明）：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound-control.html
- Unbound 官方手册，`unbound-checkconf(8)`（检查范围与用法）：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound-checkconf.html
- Unbound 官方手册,Performance Tuning(缓存内存、线程与 slabs 的规划):https://unbound.docs.nlnetlabs.nl/en/latest/topics/core/performance.html
- NLnet Labs，Unbound 项目文档入口（版本相关的默认区清单、AppArmor 与模块配置说明）：https://nlnetlabs.nl/documentation/unbound/
- RFC 5737，*IPv4 Address Blocks Reserved for Documentation*（本章示例地址段）：https://www.rfc-editor.org/rfc/rfc5737
- RFC 1034，*Domain Names — Concepts and Facilities*（区、权威应答与递归的基本概念）：https://www.rfc-editor.org/rfc/rfc1034

第二十五章

内网 DDNS 与微软 DNS 对照

内部名字不是凭空出现的：一台机器拿到 DHCP 租约，或者一个服务注册了自己，总得有人把「名字→地址」写进某个权威区。Unbound 在这条链路上扮演的角色经常被误读，最常见的一句话是「把 Unbound 的 DDNS 配一下」。这句话没有对应的配置项，也没有对应的功能。先把这个立论钉死，再看微软 DNS 那套真正实现了动态更新的模型，最后给出纯 Unbound 架构里让内网动态名字落地的三条绕行路径。

与相邻章节的边界：第 2 章已经做过主流实现的功能横评，本章不再贴产品对照表；第 6 章给了本地根区镜像的完整配置，本章只在必要处一句交叉引用，不重画那份配置。

25.1 Unbound 不接受动态更新

25.1.1 FEATURES 文件里的原文口径

NLnet Labs 在 Unbound 仓库的 `doc/FEATURES` 里把能力边界写得很直接：

```
Big Features: Recursive service. Caching service.
Forwarding and stub zones. Very limited authoritative service.

RFC 1995, 1996, 2136: not authoritative, so no AXFR, IXFR,
NOTIFY or dynamic update services are appropriate.
```

这段话有两层意思，不能只截前半句：

- Unbound 的定位是递归 / 缓存 / 转发 / stub，**权威服务是「非常有限的」（very limited）**，不是「兼作权威服务器」；
- 因为它不是权威服务器，AXFR、IXFR、NOTIFY 与 [RFC 2136] 的 dynamic update 服务「不适用」（not appropriate）——这是设计结论，不是「还没实现」的待办项。

于是：Unbound 不处理 `UPDATE` 报文、不做区传送服务端、不发 NOTIFY。官方手册里检索 `DDNS` / `RFC 2136` 命中为零；唯一一处“dynamic update”字样是 `num.query.opcode.UPDATE` 统计计数器的说明——它只用来数收到了多少个此类报文，不代表会应答。

不要把本章后面任何一段称作「Unbound 的 DDNS 配置」。后面写的都是把更新交给别人做、Unbound 只负责查到结果，是架构绕行，不是功能开关。

25.1.2 「非常有限的权威服务」具体是什么

只剩两样东西：`local-zone` / `local-data` 的静态配置，以及 `unbound-control local_zone` / `local_data` 的运行时增删。先看静态形式：

```
# /etc/unbound/unbound.conf.d/10-local-data.conf（教学示例）
server:
    local-zone: "home.arpa." static
    local-data: "gw-01.home.arpa.   IN A    192.0.2.1"
    local-data: "edge-01.home.arpa. IN A    192.0.2.2"
```

```
local-data: "gw-01.home.arpa.   IN AAAA 2001:db8::1"
include: "/etc/unbound/local-data.generated.conf"
```

- **为什么这么设**：static 让本地区直接应答、不向外部递归；local-data 是逐条写死的资源记录；include 把「可再生成的那部分」单独放一个文件，为路径③留出替换接缝。
- **怎么验证**：dig @198.51.100.53 gw-01.home.arpa. A +short
- **失败模式 / 回滚点**：local-zone 类型或 local-data 拼写错会让 unbound-checkconf 直接报错；把 include 指向空文件或上一版本即可回滚，改完 unbound-control reload 生效。注意 local-zone 匹配发生在递归 / 转发之前，写错区名会静默接管本来该外发的查询（第 24 章讲过这个优先级）。

运行时增删靠控制通道，它改的是内存里的本地区，不是配置文件：

```
unbound-control local_zone home.arpa. static
unbound-control local_data gw-01.home.arpa. IN A 192.0.2.1
unbound-control local_data_remove gw-01.home.arpa.
```

- **为什么这么设**：适合临时加一条、临时挡一个名字做验证，不必 reload 整个实例。
- **怎么验证**：unbound-control list_local_data | grep 'gw-01.home.arpa'
- **失败模式 / 回滚点**：**不持久化**——进程重启或 reload 后运行时增删全部消失；用 local_data_remove 逐条撤回，或重启实例回到配置文件定义的状态。控制通道本身要按第 9 章收在受保护网络里，别把它当公开写入接口。

25.1.3　不持久化是设计，不是缺陷

把上面两条合起来看，Unbound 的本地区是「启动时从文件读、运行中可临时改、重启即回到文件」的模型。它天然缺少动态更新协议里最要紧的两件事：多客户端并发写入的鉴权，以及写入结果的持久落盘。所以内网名字要「动」起来，只有两条路：让别的进程去写权威区（路径①、②），或者让别的进程去写那个被 include 的文件（路径③）。配置管理——版本化文件 + 原子 install + reload——在这三种方案里都是兜底，不是可选项。

unbound-control 提供的 local_zone / local_data 名字里带 “local”，是因为它们服务于本地区，不是因为它们是 [RFC 2136] 的实现。把这两组命令当成「DDNS 接口」用，第一次重启就会丢数据。

25.2　微软 DNS：真正实现了 [RFC 2136] 的那套模型

这一节是事实对照，说明「动态更新」在成熟实现里长什么样，好让后面三条绕行路径知道自己省掉了什么。资料来源是 Microsoft Learn 的《Dynamic DNS Update in Windows and Windows Server》及同系列文档，具体条目按标题检索。

25.2.1　UPDATE 的原子性与 prerequisite

[RFC 2136] 的 UPDATE 报文不是「改一条记录」的远程调用，而是**对一个区的一次原子事务**：一个报文里可以同时带删除、添加、以及 prerequisite（前置条件）。prerequisite 形如「这个名字必须不存在」「这个名字必须已有这条记录」，服务器先逐条检查前置条件，任一不满足就整包拒绝，一个字节都不改。

这一点决定了它和「脚本改文件再 reload」的根本差别：

维度	[RFC 2136] UPDATE	静态文件生成
一致性	单报文原子提交，前置条件失败则整体拒绝	文件级全量替换，靠生成脚本自己保证
并发写入	服务器串行化处理，带鉴权	多写者需外部加锁
生效点	服务器应答即生效	reload 之后生效
部分失败	不存在	可能写出半截文件（须原子 install）

后面路径①的 nsupdate 骨架正是靠这个原子性拿到「失败即未修改」的回滚语义；路径③没有这层保护，只能自己造哨兵和备份。

25.2.2 更新必须落到主服务器

标准主辅模型里，`UPDATE` 只能发给该区的**主服务器**（primary），由它改区数据再经区传送扩散到辅服务器。AD 集成区改变了这一点：区数据存在 Active Directory 里并可复制到多台域控制器，因此更新可以发给**任意一台持有该区副本的 DC**，由 AD 复制机制统一收敛。

这条差异直接影响规划：如果内部区是文件型主区，就必须保证客户端和 DHCP 都能稳定到达那一台主服务器，它是一个明确单点；换成 AD 集成区，写入点变成多台，但代价是引入了 AD 复制延迟与 DC 定位。

25.2.3 安全动态更新只对 AD 集成区可用

三种状态的边界要分清楚，最常见的误判是把「开了动态更新」等同于「开了安全动态更新」：

- **文件型主区**：默认不开放动态更新；可以手动改成允许，但没有 AD 身份体系可依托，只能靠「允许任何更新」或按地址粗粒度放行。
- **AD 集成区**：集成之后默认就是「**仅安全更新**」**(secure only)**；也可以放宽为「非安全 + 安全」或「不允许」，但那是显式降级，不是默认行为。
- **安全动态更新只对 AD 集成区可用**——没有 AD，就没有这套基于 Kerberos 的客户端身份与 ACL。

换句话说，「把区转成 AD 集成」这一步同时改了两件事：写入点从单点变成多点，默认策略从「不开放」变成「仅安全更新」。回滚时要一起考虑，别只导回 zone 文件。

25.2.4 TKEY → TSIG → ACL →签名应答

微软文档给出的安全更新时序（Windows 侧只用 Kerberos，不用其他 TKEY 模式）：

```
1. 客户端先尝试一次不带签名的 UPDATE
2. 服务器按「仅安全更新」拒绝
3. 客户端与服务器做 TKEY 协商，建立安全上下文（Windows 只用 Kerberos）
4. 用协商出的密钥以 TSIG 签名正式 UPDATE 报文
5. 服务器按 ACL 校验「这个身份能不能改这个名字」，通过后写入 AD
6. 服务器返回同样带签名的应答
```

相关标准是 [RFC 3645]（GSS-TSIG），它把 GSS-API 与 TSIG 接起来；TKEY 与 TSIG 本身的规范按名称在 RFC Editor 检索，本章不臆断编号。第 3 步与第 5 步是安全性的两个支点：**前者证明「你是谁」，后者决定「你能改哪些名字」**。缺了 ACL，仅仅「能签名」就等于任何域用户都能覆盖别人的记录。

对照 Unbound：`local_data` 既没有 TKEY 也没有 ACL，它唯一的「权限」是能不能连上控制端口。这也是为什么运行时增删只适合调试、不适合做生产写入通道。

25.2.5 配套机制清单

动态更新不是孤立功能，微软 DNS 围绕它配了一整套东西：

- **DHCP 代注册**：客户端不自己发 UPDATE，由 DHCP 服务器在租约事件里代写 A / AAAA / PTR。
- **`DnsUpdateProxy` 组**：让 DHCP 代注册的记录不被当成「某台 DHCP 服务器私有」，否则后续真机自己来更新时会被 owner 检查挡掉。
- **`DHCID` 记录与 Name Protection**：用 `DHCID` 把名字和租约绑定，Name Protection 防止一个客户端抢注另一个客户端已有的名字（尤其 PTR）。
- **aging / scavenging（老化与清理）**：**默认关闭**；启用后 no-refresh 与 refresh 两个间隔**各 7 天**。默认关的意思是「陈旧记录会一直留着」，不是「自动清理已打开」。
- **`_msdcs` 与 DC locator**：域控定位专用的子域与 SRV 记录集，是 AD 自己最依赖的一组动态名字。
- **GlobalNames 区**：为不带后缀的单标签名字提供一套受控解析。

路径①若接的是 AD DNS，这套东西仍然由 AD 侧负责；Unbound 的 `stub-zone` 只是把查询送过去，不参与其中任何一环。

25.2.6 运维上真实会踩的坑

坑	现象	处理 / 回滚点
时钟偏差 > 5 分钟	Kerberos 认证失败→ TKEY 协商失败→更新被拒，客户端名字不出现	先对齐域内时间源；排查时把时间同步列为第一检查项
DHCP 用了高权限凭据	记录 owner 变成服务账户，真机后续无法自行更新或删除	改用**低权限专用凭据**；已写错的记录按 owner 批量清理后重建
反向区漏建	正向 A 记录正常、PTR 全无，反向校验与部分服务发现失败	正向区上线时同步建反向区并纳入巡检清单
误以为 scavenging 默认开	陈旧记录长期堆积，同一名字指向已释放地址	显式确认老化开关与两个 7 天间隔，先小范围试运行
区类型选错	想用「仅安全更新」却发现是文件型主区	区类型转换是变更动作，回滚点留在转换前导出的 zone 文件

前两行在路径①里同样适用：TSIG 共享密钥不依赖时钟，但 GSS-TSIG 依赖 Kerberos，一旦把脚本从 TSIG 切到 GSS-TSIG，时间同步就从「建议」变成「硬前提」。

25.2.7　社区观点，不是官方规定

NLnet Labs 邮件列表 2014-06 的 [Unbound-users] Unbound vs MS Resolver 讨论里有一种常见建议：AD 假设 DNS 服务器能接受 AD 认证的动态更新，所以域控与成员机最好指向微软 DNS；Unbound 放在外侧做**对外递归验证层**，把 AD 域用 `stub-zone` / `forward-zone` 指过去。

这是社区观点，不是 NLnet Labs 的官方规定，也不是微软的强制要求。它描述的是一种减少摩擦的部署习惯：把「谁负责写内部区」和「谁负责解析外部名字」拆开，避免让一个递归验证器去承担它不做的写入职责。是否照做取决于现有 AD 拓扑，不属于协议约束。把它写成「官方要求域控必须指向微软 DNS」是错的。

25.3　三条绕行路径

> **以下三方案是本书对工程做法的自定义归纳，不是任何软件的官方功能列表，也不是「Unbound 的 DDNS」。**
> 共同前提是：Unbound 只做递归 / 验证 / 转发，写入动作全部发生在它之外。

25.3.1　路径①外部权威 + DHCP 脚本写 nsupdate

租约事件触发脚本，脚本用 `nsupdate` 把记录写进一个真正的权威区；Unbound 用 `stub-zone` 指向那个权威。已有 BIND / NSD / AD DNS 的环境最省事。

`nsupdate` 教学骨架（TSIG 密钥放在权限受限的文件里，共享秘密用占位符）：

```
#!/usr/bin/env bash
# /usr/local/sbin/ddns-lease.sh ——由 DHCP 租约事件调用，教学骨架
set -euo pipefail

ZONE="home.arpa."
SERVER="192.0.2.53"               # 权威服务器，教学用 RFC 5737 地址
KEYFILE="/etc/ddns/home-arpa.key"  # key "ddns-key" { algorithm hmac-sha256; secret "***"; };
NAME="gw-01.home.arpa."
ADDR="192.0.2.1"
TTL=300

/usr/bin/nsupdate -k "$KEYFILE" <<EOF
server $SERVER
zone $ZONE
update delete $NAME A
update add $NAME $TTL A $ADDR
send
EOF
```

- **为什么这么设**：先 `delete` 再 `add` 保证同一名字不留旧值；`zone` 明确目标区避免脚本打错区；`-k` 走密钥文件而不是把秘密写进命令行（命令行会进进程列表）；短 TTL 让租约变化能较快扩散。需要 AD 身份验证时改用 GSS-TSIG 模式，按 [RFC 3645] 与所用客户端文档检索，别把 TSIG 与 Kerberos 混成一种东西。

- **怎么验证**：`dig @192.0.2.53 gw-01.home.arpa. A +short`
- **失败模式 / 回滚点**：密钥错 / 时钟偏差 / ACL 不放行→ `nsupdate` 报 `update failed: REFUSED` 或 TSIG 校验错，此时区数据**未被修改**（原子性帮了忙）；脚本要记录租约旧值，回滚时用同一骨架把 `ADDR` 换回旧地址重放一次。密钥文件权限不是 `600` 就别上线。

Unbound 侧只需要一条指向权威的 stub：

```
# /etc/unbound/unbound.conf.d/20-internal-stub.conf（教学示例）
stub-zone:
    name: "home.arpa."
    stub-addr: 192.0.2.53
    stub-addr: 2001:db8::53
    stub-first: no
```

- **为什么这么设**：`stub-zone` 让 Unbound 对 `home.arpa.` 直接问权威、不走公网根；`stub-first: no` 是官方默认值，表示 stub 失败时**不**回退到常规递归，避免内部名字在权威故障时漏到公网。它与 `forward-zone` 的差别在于收到的是权威应答而非转发结果。
- **怎么验证**：`dig @198.51.100.53 gw-01.home.arpa. A +short`，与直连权威的结果比对，应一致。
- **失败模式 / 回滚点**：权威不可达时该区立即解析失败（不会自动回公网），这是刻意的 fail-closed；把这段 `stub-zone` 注释掉并 `unbound-control reload` 即回滚，但回滚后 `home.arpa.` 会落到公网递归、多半得到 NXDOMAIN——所以更稳妥的回滚是修权威，不是删 stub。

25.3.2 路径② NSD 权威 + Unbound 递归

同一台或另一台机器上跑 NSD 管内部区，Unbound 仍用 `stub-zone` / `forward-zone` 指过去；更新走 NSD 的 zonefile 生成，或由外部 `nsupdate` 写 NSD 的区。**明确一句：`auth-zone` 是只读的本地根区镜像，不是可写权威区**，配置与用途见第 6 章，本章不再贴那段配置。这条路径与①的差别只在「权威是自己人、且和递归器同机或近邻」，写入责任依旧不在 Unbound。

选它时要注意一个工程边界：权威与递归同机时，两者都监听 53 端口需要分地址或分端口；若还用了第 24 章那类:53 劫持规则，本机解析器进程的出向 53 必须按 uid 豁免，否则递归查询会被自己的规则吃掉。

25.3.3 路径③静态生成 local-data include

DHCP 租约表 / 资产清单→生成 `local-data` 行→原子 `install` → reload。没有实时 UPDATE，但零额外组件。第 23 章 WPAD 的 `wpad.< 内网域 >`（本章写作 `wpad.home.arpa.`）就是这类静态产物的例子。

```
#!/usr/bin/env bash
# /usr/local/sbin/gen-local-data.sh ——租约/资产清单 → include 文件，教学骨架
set -euo pipefail

SRC="/var/lib/ddns/leases.tsv"      # 列: name<TAB>addr
ZONE="home.arpa."
OUT="/etc/unbound/local-data.generated.conf"
BACKUP="${OUT}.prev"
TMP="$(mktemp "${OUT}.XXXXXX")"
trap 'rm -f "$TMP"' EXIT

{
    printf '# generated by gen-local-data.sh; do not edit\n'
    awk -F'\t' -v zone="$ZONE" 'NF==2 && $1 ~ /^[a-zA-Z0-9-]+$/ {
        printf "local-data: \"%s.%s  300 IN A  %s\"\n", $1, zone, $2
    }' "$SRC"
    printf '# EOF\n'                # 完整性哨兵：最后一行必须存在
} > "$TMP"

test -s "$TMP"                                      # 非空
tail -n 1 "$TMP" | grep -qx '# EOF'                 # 未被截断
if [ -f "$OUT" ]; then cp -p "$OUT" "$BACKUP"; fi   # 保留上一版本，回滚点
install -m 0644 -o root -g root "$TMP" "$OUT"        # 原子替换

if ! /usr/sbin/unbound-checkconf /etc/unbound/unbound.conf >/dev/null; then
```

```
    install -m 0644 -o root -g root "$BACKUP" "$OUT"    # 立即回滚
    exit 1
fi
/usr/sbin/unbound-control reload
```

- **为什么这么设**：先写临时文件、校验完整性与配置语法、最后 `install` 原子替换，避免 reload 读到半截文件；`# EOF` 哨兵让「生成被截断」可被一条命令检出；`awk` 里过滤名字字符，防止清单里的怪字符注入配置；`unbound-checkconf` 放在 `install` 之后、`reload` 之前，失败就把 `.prev` 装回去。
- **怎么验证**：`tail -n 1 /etc/unbound/local-data.generated.conf && dig @198.51.100.53 gw-01.home.arpa. A +short`
- **失败模式 / 回滚点**：清单格式错→ `awk` 少产记录，表现为个别名字消失；生成被截断→哨兵检查失败，旧文件**保持不变**，这是设计好的自动回滚；`unbound-checkconf` 不过→脚本立即还原 `.prev` 并退出，不 reload。人工回滚只需把上一版本 `install` 回去再 reload，所以每次生成都保留带版本号的副本。

25.3.4　三方案对照表

维度	①外部权威 + `nsupdate`	② NSD 权威 + Unbound	③静态生成 `local-data`
实时性	租约事件即写，秒级（受 TTL 与缓存影响）	取决于 zonefile 生成或 `nsupdate` 频率	无实时性，等于生成 + reload 的周期
认证	TSIG 共享密钥 / GSS-TSIG（[RFC 3645]）	取决于权威侧配置，可沿用 TSIG	无协议级认证，靠文件权限与生成机管控
额外组件	权威服务器、DHCP 脚本、密钥分发	NSD（或同类权威）+ 生成 / 更新通道	无（只需 cron / systemd timer 与脚本）
故障域	权威不可达→内部区整区解析失败	同左，但权威与递归可同机，链路更短	生成机故障只影响「新名字」，旧文件仍可解析
适用规模	中大型、已有权威体系或 AD	中小规模、希望自管内区	小规模、名字变化不频繁（第 23 章 WPAD 类）
回滚点	重放 `nsupdate` 写回旧值；密钥 / ACL 变更前快照	回滚 zonefile 版本并 reload	`install` 上一版本生成文件 + reload

三行里最容易被低估的是「故障域」：①和②把内部名字的可用性绑到了权威服务器的可用性上，③把新名字的可用性绑到了生成机，但**已生成的名字活在本地文件里**。小规模场景偏爱③，不是因为它更先进，而是因为它的失败模式最温和。

25.3.5　验证命令与回滚点总表

```
# 权威侧：确认写入生效
dig @192.0.2.53 gw-01.home.arpa. A +short
dig @192.0.2.53 1.2.0.192.in-addr.arpa. PTR +short    # 教学用 RFC 5737 地址的反查名

# 递归侧：确认 Unbound 确实问的是权威而不是公网
dig @198.51.100.53 gw-01.home.arpa. A +short
dig @198.51.100.53 home.arpa. SOA +noall +answer

# 配置侧：改配置前先过语法
unbound-checkconf
unbound-control reload
unbound-control list_local_data | grep 'home.arpa'
unbound-control stats | grep 'num.query.opcode.UPDATE'
```

- **为什么这么设**：权威侧和递归侧要**分别**验，两边结果一致才能证明 stub 指对了；`SOA` 用来确认应答来自内部区；`num.query.opcode.UPDATE` 若非零，说明有人把 `UPDATE` 报文打到了 Unbound——它不会应答，这属于链路配置错误，应把写入端改指权威。
- **怎么验证**：`unbound-checkconf && dig @198.51.100.53 gw-01.home.arpa. A +short`
- **失败模式 / 回滚点**：负缓存会把一次 NXDOMAIN 按 SOA minimum 缓存住（第 4、7 章的坑），修好权威后客户端仍可能失败，必要时刷链路上每个实例的缓存；三条路径的通用回滚顺序都是「先停写入端→恢复上一版本数据 / 文件→ reload →两侧 `dig` 比对」。

一个容易忽略的收尾：无论选哪条路径，内部区名都该用 `home.arpa.` 这类不可公网解析的名字；若把内部名字挂在可公开解析的域下，就得额外维护内外两套答案，排障复杂度会明显上升（第 28 章会展开这个故障域问题）。

25.4 本章要点

- Unbound **不支持 [RFC 2136] 动态更新**：官方 `doc/FEATURES` 写明「not authoritative, so no AXFR, IXFR, NOTIFY or dynamic update services are appropriate」；它只有「非常有限的权威服务」，不存在「Unbound 的 DDNS 配置」这种东西。
- 「非常有限的权威服务」= `local-zone` / `local-data` 静态配置 + `unbound-control local_zone` / `local_data` 运行时增删；**运行时改动不持久化**，重启或 reload 即回到配置文件，必须靠配置管理兜底。
- 微软 DNS 走的是完整 [RFC 2136] 模型：原子 UPDATE + prerequisite、更新落到主服务器（AD 集成区可发给任意持副本的 DC）、**安全动态更新只对 AD 集成区可用且集成后默认「仅安全更新」**、文件型主区默认不开放；流程是 TKEY（Windows 只用 Kerberos）→ TSIG 签名 UPDATE →按 ACL 写 AD →签名应答，相关 [RFC 3645] GSS-TSIG。
- 配套要点与坑：`DnsUpdateProxy` / `DHCID` + Name Protection / aging-scavenging **默认关**、no-refresh 与 refresh 各 7 天 / `_msdcs` 与 DC locator / GlobalNames；时钟偏差 > 5 分钟 Kerberos 失败、DHCP 须用低权限专用凭据否则 owner 错乱、反向区容易漏建。2014-06 邮件列表的「域控指向微软 DNS、Unbound 作对外递归验证层」只是**社区观点**。
- 纯 Unbound 架构的三条绕行路径：①外部权威 + DHCP 脚本 `nsupdate` + `stub-zone`；② NSD 权威 + Unbound 递归（`auth-zone` 是只读根镜像，不是可写权威区，见第 6 章）；③静态生成 `local-data` include（第 23 章 `wpad.< 内网域 >` 即此类）。三者差别集中在实时性、认证、额外组件与回滚点。

25.5 参考资料

1. `https://github.com/NLnetLabs/unbound` —NLnet Labs / Unbound 官方仓库，仓库内 `doc/FEATURES`（"Very limited authoritative service" 与 RFC 1995 / 1996 / 2136 的原文口径）。
2. `https://unbound.docs.nlnetlabs.nl/en/latest/` —Unbound 官方在线手册入口（`unbound.conf` 的 `local-zone` / `local-data` / `stub-zone` / `auth-zone`，`unbound-control` 的 `local_zone` / `local_data` / `list_local_data`，以及 `num.query.opcode.UPDATE`，按名称检索）。
3. `https://nlnetlabs.nl/documentation/unbound/` —NLnet Labs：Unbound 文档与资料页。
4. `https://www.rfc-editor.org/rfc/rfc2136` —[RFC 2136] *Dynamic Updates in the Domain Name System (DNS UPDATE)*：UPDATE 的原子性、prerequisite 与应答语义。
5. `https://www.rfc-editor.org/rfc/rfc3645` —[RFC 3645]：GSS-TSIG，安全动态更新里 GSS-API 与 TSIG 的衔接。
6. `https://learn.microsoft.com/` —Microsoft Learn，按标题检索《Dynamic DNS Update in Windows and Windows Server》及 secure dynamic update、`DnsUpdateProxy`、DHCID / Name Protection、aging and scavenging、GlobalNames、DC locator / `_msdcs` 等条目。
7. `https://lists.nlnetlabs.nl/` —NLnet Labs 邮件列表归档入口，`[Unbound-users]` 按主题 `Unbound vs MS Resolver` 与 2014-06 月份检索（本章仅作社区观点引用）。
8. `https://www.rfc-editor.org/rfc/rfc5737` —[RFC 5737]：文档用 IPv4 地址块（本章全部示例地址依据）。

第二十六章

企业级大规模部署：anycast 与集群

把一台 Unbound 扩展成「对外只表现为一个服务地址」的一组节点，难点不在 `num-threads` 调到几，而在：客户端的这一次查询和下一次查询可能落到不同节点，节点之间必须能互相承认对方签过的状态。本章只讲解析层怎么横向扩，落到配置、命令、验收清单与回滚点。

26.1 前提与边界

26.1.1 Unbound 是递归验证器，不是权威服务器

四章统一的口径在这里是 anycast（任播）的前提：Unbound 是递归验证器（recursive resolver），`local-zone` / `local-data` 只提供「非常有限的权威服务」。因此：

- anycast 扩展的是**递归 / 验证 / 转发这一层**，不是权威数据面；
- Unbound 不做 AXFR / IXFR / NOTIFY、不接受 `UPDATE` 报文（见官方 `FEATURES` 与手册），所以「主辅区同步」「辅升主」这些事不属于 Unbound 节点之间的同步，而是权威面的事；
- 本章的「隐藏主服务器」「一主多辅」落在 NSD / BIND / AD DNS 那一侧，Unbound 通过 `stub-zone` / `forward-zone` 指过去即可。

`auth-zone` 是只读根镜像，不是可写权威区（第 6 章已讲），别把它当成 anycast 集群里「共享区数据」的手段。

26.1.2 与相邻章节的边界

- 第 7 章讲一台机器上多实例、缓存与观测，本章讲多台机器对一个地址；`num-threads`、libevent、forked operation、per-thread 缓存只是单机性能基线，本章当作每个节点已调好的既定前提一句带过（参数见官方手册对应章节）。
- 第 20 章讲 Kubernetes 里的服务发现与 NodeLocal DNSCache，本章不进容器编排；
- 第 21 章是「一台网关 + 一台云主机」的实验拓扑，本章是它的生产化放大，不重画那张拓扑。

26.2 官方手册里的 anycast 事实

这一节只写官方手册写明的事实，不掺本书推断。DNS Cookie 机制的规范编号本章不臆断，按名称在 RFC Editor 检索；下列选项与控制命令的**名称**来自官方手册，参数取值见手册页。

26.2.1 同一集合必须共享 cookie-secret

官方手册对 `cookie-secret` 的要求非常直白：**同一 anycast 集合的节点必须共享同一个密钥**，才能互相验证 Server Cookie。原因不是配置项洁癖：客户端第一次查询被 BGP/ECMP 送到 `edge-01`，`edge-01` 用它手里的密钥签发一个 Server Cookie；下一次查询可能被送到 `edge-02`，`edge-02` 要能校验这个 Cookie 是不是自己人签的。密钥不一致，跨节点的 Server Cookie 校验就失败，Cookie 带来的放大攻击缓解随之失效。

```
server:
    # 方式一：持久化到文件（推荐，便于配置分发与轮换）
    cookie-secret-file: "/etc/unbound/cookie-secret.key"
```

```
    # 方式二：内联。生产环境不要把真实值写进公开配置或版本库：
    # cookie-secret: "<128_BIT_HEX_FROM_SECRET_STORE>"
```

为什么这么设：`cookie-secret-file` 把 128 位密钥落到文件，配置分发系统可以把「同一把」密钥幂等地推到每个节点；`cookie-secret` 内联是等价选项，但真实值一旦进版本库就等于泄露。二者都要求全集合一致。

怎么验证：`unbound-control status` 逐节点确认实例在线，再比对各节点 `/etc/unbound/cookie-secret.key` 的**校验和是否一致**（`sha256sum /etc/unbound/cookie-secret.key`），并用 `stat -c '%a %U' /etc/unbound/cookie-secret.key` 确认权限（应为 600、属主为 unbound）；只比对存在性与权限即可，不要打印真实密钥内容。

失败模式 / 回滚点：某个节点漏分发、拿到不同密钥，表现为「客户端换节点后 Cookie 保护消失」。回滚点在分发阶段——在激活新密钥之前，把正确文件重推到全部节点再继续；一旦对外服务，只能按下一节的分阶段流程重新收敛。

> 注意区分：`cookie-secret` 属于 DNS Cookie（报文层的来源校验 / 放大缓解），证书轮换属于 DoT / DoH / DoQ 的**传输加密**，DNSSEC 则是**数据完整性 / 来源认证**。三者不互相替代，不要把这里的密钥滚动和 DNSSEC 信任锚混为一谈。

26.2.2　cookie-secret-file 与分阶段滚动

官方手册给出的滚动流程是三个 `unbound-control` 命令，顺序不能乱：

阶段	命令	状态	目的
1 全部 staging	`add_cookie_secret`	新密钥加入但**不生效**	先把新密钥摆到所有节点上
2 全部激活	`activate_cookie_secret`	新密钥开始签发 Server Cookie	统一切换签发方
3 删旧	`drop_cookie_secret`	移除旧密钥	收敛到只剩新密钥

顺序的约束是：**先全部 staging、再激活；激活满 1 小时后才删旧**。两个理由都来自「anycast 节点不会同一秒更新完」：

- 如果先在一台节点激活、再慢慢推其余节点，客户端完全可能先拿到「新密钥签的 Cookie」，下一跳撞到还在用旧密钥的节点，校验失败；
- 激活之后，仍有一批客户端手里握着**旧密钥签发的** Server Cookie，必须留够窗口让它们自然过期。官方手册给的是 1 小时，别自作主张缩短。

```
#!/usr/bin/env bash
# cookie-rotate.sh ——教学骨架：分阶段滚动 anycast 集合的 DNS Cookie 密钥
# 命令名称与顺序按官方手册；参数细节见 unbound-control 手册页。
set -euo pipefail
NODES=(edge-01 edge-02 edge-03)
NEW_SECRET="<128_BIT_HEX_FROM_SECRET_STORE>"   # 占位，由密钥管理下发，勿入版本库

# 阶段 1：全部节点先把新密钥加入 staging（此时不生效）
for n in "${NODES[@]}"; do
    ssh "ops@${n}" unbound-control add_cookie_secret "\"${NEW_SECRET}\""
done

# 阶段 2：全部节点激活（此后新应答用新密钥签发 Server Cookie）
for n in "${NODES[@]}"; do
    ssh "ops@${n}" unbound-control activate_cookie_secret
done

# 阶段 3：激活满 1 小时、旧 Cookie 已过期，才删旧密钥
sleep 3600
for n in "${NODES[@]}"; do
    ssh "ops@${n}" unbound-control drop_cookie_secret
done
```

为什么这么设：把「全部节点一致」当成每一步的门禁，而不是「每台各自走完三步」。逐台走完三步会让集合在几分钟里同时存在新旧两种签发方，正是 anycast 最怕的状态。

怎么验证：阶段 2 之后 `for n in edge-01 edge-02 edge-03; do ssh "ops@${n}" unbound-control status; done`，并从客户端连续查询、观察 OPT 伪段的 COOKIE 选项（`dig @192.0.2.53 example.net A +short`，带 Cookie 的方法按名称在 `dig` 手册检索），确认跨节点一致。

失败模式 / 回滚点：阶段 1、2 出错都可「暂停、补齐、重来」，因为旧密钥还在服务。真正的**不可逆点**是 `drop_cookie_secret`——一旦删早了，仍持旧 Cookie 的客户端会被拒；此时没有「撤销 drop」的捷径，只能重新 `add_cookie_secret` 把旧密钥加回再激活（见官方手册）。所以 1 小时等待与「删前再确认」是硬回滚点。

26.2.3 证书轮换：新旧配置并存的窗口

官方手册提到，TLS 证书轮换期间可用 `tls-additional-port` 以及「客户端 magic」一类机制，让**新旧证书（及对应的客户端识别）在轮换窗口内并存**；理由是 anycast 各节点的配置不会在同一秒更新完，必须允许过渡期两种配置都能服务。具体参数、端口取值与 magic 的定义，按名称在官方手册与 RFC Editor 检索，本章不臆断编号与取值。

证书轮换和 Cookie 密钥滚动是同一种「集合内暂时不一致」问题，都要按「先铺满、再切换、留足过期窗口、最后收敛」的节奏走，不能按「改一台算一台」的节奏走。

26.3 NIST SP 800-81r3 的部署指导

`NIST SP 800-81r3`（*Secure Domain Name System (DNS) Deployment Guide*）给出的是**控制目标**，不是 Unbound 的配置项。本节只列控制目标名称与含义，**不写条款号、不编控制项编号**；编号与条文细节见官方资料，或按名称在 NIST CSRC 检索。

控制目标名称	含义	在本章的落点
部署专用 DNS 服务器以缩小攻击面	DNS 主机不同置无关服务；不可行时才与 DHCP 等强相关服务同置	anycast 节点只跑 Unbound + 路由守护进程，不叠别的服务
每个区至少一个主 + 多个辅	权威数据不单点	权威面（NSD / AD DNS），Unbound 不做主辅同步
至少两台权威在不同网段、不同地理位置	网络与地理双重分散	与路由宣告的故障域设计对齐
隐藏主服务器（hidden primary）	主服务器不对公网宣告，只让辅服务器取区	权威面单独设计，不并入 anycast VIP
存根 / 客户端至少配两个递归器	客户端侧不要只写一个解析器地址	客户端配置层面，见下文对单地址客户端的讨论
anycast 可隐藏单节点故障	单个节点宕掉时，路由把流量送到其余节点，客户端不感知	本章工程骨架的核心，尤其对**只配一个地址的客户端**重要
有书面的辅升主流程	主不可用时，如何把辅提升为主，要事先写清谁做、怎么做	权威面预案，配演练记录
组件故障告警	组件级故障要能被发现，不能只靠用户报障	`unbound-control` 集中观测 + 路由告警
对齐 NIST SP 800-53 SC-22 的地理分离讨论	至少两台权威分处地理分离的子网	只做名称层面的对齐讨论，条款细节见官方资料

两点值得展开：

其一，指南把「客户端至少配两个递归器」和「anycast 可隐藏单节点故障」放在一起讲，是因为现实里大量存根解析器（stub resolver）**只配了一个地址**。对这类客户端，anycast 是唯一能让「换一台服务器」这件事不依赖客户端改配置的办法——路由层把同一个地址送到健康的节点，客户端毫无感知。代价是：客户端没有第二个地址可退，因此 anycast 集合本身的健康检查和撤路由必须更可靠。

其二，地理分离讨论对齐到 `NIST SP 800-53` 的 `SC-22`。本章只引用其名称，不写条款号；读者在官方资料里按名称检索即可。本书的立场是：这些控制目标描述的是**目标**，anycast 是达成其中「隐藏单节点故障」的一种手段，不是替代「一主多辅」「异地权威」的捷径。

26.4　工程骨架

以下做法是本书对常见部署的**归纳**，不是 Unbound 官方规定，也不是 NIST 条款：anycast 服务地址绑 loopback + BGP/OSPF 宣告、健康检查失败即撤路由、缓存不共享带来的冷缓存抖动、配置与过滤表分发的幂等与版本化、`unbound-control` 集中观测、DNS Cookie 密钥滚动、证书轮换、灰度与回滚、权威面与递归面分开。骨架刻意只用文档地址与示例主机名，便于照抄到实验环境再替换。

26.4.1　服务地址绑 loopback

每台节点把服务 VIP 绑在 `lo`，而不是某块物理网卡：网卡抖动不影响 VIP 存在，路由守护进程宣告的是这个 /32、/128 主机路由。

```
# 每个 anycast 节点执行（教学用示例地址）
ip addr add 192.0.2.53/32 dev lo
ip addr add 2001:db8::53/128 dev lo
```

为什么这么设：VIP 与物理接口解绑，接口 flap 不会让本机服务地址消失；上层宣告的只是「到这个主机路由」的可达性。

怎么验证：`ip -4 addr show dev lo | grep 192.0.2.53` 与 `ip -6 addr show dev lo | grep 2001:db8::53`。

失败模式 / 回滚点：地址被误配成 /24 会与其他节点冲突；回滚为 `ip addr del 192.0.2.53/32 dev lo`，并确认路由已撤销后再删。

26.4.2　路由宣告伪配置

```
# 路由宣告伪配置（示意，非任何厂商真实语法；具体命令见所用路由守护进程官方资料）
# 节点 underlay: 198.51.100.11 (edge-01)、198.51.100.12 (edge-02)
# 服务 VIP: 192.0.2.53/32、2001:db8::53/128（各自绑在 loopback）
# 集中观测端: 203.0.113.10

router bgp <LOCAL_AS>
  neighbor 198.51.100.1 remote-as <UPSTREAM_AS>
  address-family ipv4 unicast
    network 192.0.2.53/32        # 仅当 anycast-healthcheck.sh 通过时宣告
  address-family ipv6 unicast
    network 2001:db8::53/128     # 同上，条件宣告
# 条件：健康检查失败 -> 撤销对应 network 宣告（即「撤路由」）
# 同一 VIP 被多台节点宣告，上游按 BGP/IGP 选路把流量分摊到健康节点
```

为什么这么设：宣告与**健康状态绑定**，而不是「进程起来就一直宣告」，这样单节点故障能自动从路由表消失，正是 anycast 隐藏故障的机制。

怎么验证：在上游设备查看 `192.0.2.53/32` 是否只在健康检查通过的节点上出现（命令按所用守护进程检索），客户端侧 `dig @192.0.2.53 example.net A +short +time=1 +tries=1` 应稳定返回。

失败模式 / 回滚点：条件宣告没接上健康检查→故障节点仍在表里→流量黑洞。回滚点是手工撤销该 `network` 宣告，先让流量绕开，再修脚本。

26.4.3　健康检查脚本

```
#!/usr/bin/env bash
# anycast-healthcheck.sh ——教学骨架：健康检查失败即撤路由
# 撤/宣告路由的落地命令取决于路由守护进程（FRR / BIRD 等），此处用钩子函数占位。
set -euo pipefail
VIP4="192.0.2.53"
VIP6="2001:db8::53"
SECRET_FILE="/etc/unbound/cookie-secret.key"

withdraw_anycast_route() {
    logger -t anycast-health "withdraw ${VIP4} ${VIP6}"   # 伪实现：替换为真实撤宣告命令
}
announce_anycast_route() {
    logger -t anycast-health "announce ${VIP4} ${VIP6}"   # 伪实现：替换为真实宣告命令
```

```
}

# 1) 本地控制通道与进程存活
if ! unbound-control status >/dev/null 2>&1; then
    withdraw_anycast_route; exit 1
fi
# 2) 本机 VIP 上真的能解析（不能只看进程在不在）
if ! dig "@${VIP4}" example.net A +short +time=1 +tries=1 >/dev/null 2>&1; then
    withdraw_anycast_route; exit 1
fi
# 3) 共享密钥文件存在且非空（缺了会导致跨节点 Server Cookie 校验失败）
if [[ ! -s "${SECRET_FILE}" ]]; then
    withdraw_anycast_route; exit 1
fi
announce_anycast_route
```

为什么这么设：三层检查分别挡住「进程死了」「进程活着但不解析」「密钥漏分发」三类黑洞；只查进程存活是最常见的假阳性来源。

怎么验证：临时停掉 Unbound 后运行脚本，`logger` 里出现 `withdraw`（`journalctl -t anycast-health -n 5`）；恢复后重新出现 `announce`。

失败模式 / 回滚点：判定过严（一次 `dig` 抖动就撤）会造成频繁切流、放大冷缓存抖动；回滚为放宽成「连续 N 次失败才撤」，并保留手动 `announce_anycast_route` 入口。

26.4.4 单节点组装配置

```
server:
    interface: 192.0.2.53
    interface: 2001:db8::53
    # 同一 anycast 集合共享同一把密钥（见上一节）
    cookie-secret-file: "/etc/unbound/cookie-secret.key"
    # 过滤表 / local-data 由配置分发按版本 include，不手工改
    include: "/etc/unbound/conf.d/filter-v2026-11.conf"
    num-threads: 4            # 单机基线，见第 7 章
    access-control: 198.51.100.0/24 allow
    access-control: 2001:db8::/32 allow

remote-control:
    control-enable: yes       # 供 unbound-control 集中观测；控制端口不要对公网开放（第 9 章）

stub-zone:
    name: "home.arpa."
    stub-addr: 198.51.100.53  # 指向内部权威面，不是 Unbound 自己
```

为什么这么设：把「服务地址 / 共享密钥 / 过滤表版本 / 观测通道 / 内部权威指向」收在一个文件里，配置分发只需幂等地推这一个版本；`stub-zone` 明确划分递归面与权威面。

怎么验证：`unbound-checkconf /etc/unbound/unbound.conf` 与 `unbound-control status`。

失败模式 / 回滚点：版本号写错或 include 文件缺失→ reload 失败、节点被健康检查撤路由；回滚为 pin 回上一个版本目录并重放分发（见「幂等与版本化」）。

26.4.5 缓存不共享：冷缓存抖动

每个节点的缓存是**各自进程内**的（per-thread 缓存也不跨进程、更不跨主机）。撤掉一台节点的路由后，流量落到另一台节点，那台节点对这批查询是**冷缓存**：上游递归量突然上升、延迟抬高、命中率要等 TTL 爬一段时间才恢复。这是 anycast 最容易被忽略的代价。

缓解办法：新节点上线先预热再宣告路由；开启过期缓存应答（serve-stale）撑过回源抖动；避免健康检查抖动造成频繁切流；灰度期间只切一小部分流量观察命中率。注意别把「缓存不共享」误当成配置 bug——它是无状态横向扩的固有属性。

26.4.6　配置与过滤表分发：幂等 + 版本化

`unbound.conf`、`cookie-secret.key`、`local-data` / 过滤表 include 都由配置分发系统推送，要求两点：

- **幂等**：同一版本重复推送结果一致，不产生重复 include、不改文件权限；用临时文件 + 完整性校验 + `install` 原子落盘（第 24 章的原子生成思路可复用）。
- **版本化**：每次推送带版本标签，节点上报「当前版本」，集中端能一眼看出哪台漂移了。

分发顺序也要管：先铺 `cookie-secret` 的 staging（上一节阶段 1），再推配置，最后才激活。第 25 章的静态 `local-data` 生成、第 24 章的过滤表生成，都是这个分发面的输入，不在本章重复。

26.4.7　unbound-control 集中观测

每个节点开 `unbound-control`（走本地控制通道或受保护的 TLS，不对公网暴露），集中端定期拉 `unbound-control stats`，比对各节点 QPS、SERVFAIL 率、缓存命中率。切流之后若某节点命中率骤降、SERVFAIL 抬升，基本就是冷缓存抖动或配置漂移的信号。组件故障告警（NIST 控制目标之一）就挂在这些指标上，而不是靠用户报障。

26.4.8　灰度与回滚

一次只动一台节点：推版本→观察指标→再动下一台。Cookie 密钥滚动、证书轮换、过滤表更新都按这个节奏。回滚点按「不可逆程度」排序：配置版本可 pin 回上一版；路由可重新宣告；**Cookie 密钥在 `drop_cookie_secret` 之前**都可安全回退，之后只能重新加回旧密钥。每条变更都要写清「在哪一步之前可以无损回滚」。

26.4.9　权威面与递归面分开

把两个面拆开是本章的收口：

- 权威面（NSD / BIND / AD DNS）：一主多辅、隐藏主服务器、异地分布、书面的辅升主流程；它的数据同步走区传送，与 Unbound 无关。
- 递归面（本章的 anycast 集合）：无状态、可横向扩、共享 `cookie-secret-file`、各自缓存、靠路由宣告伸缩。
- 两者用 `stub-zone` / `forward-zone` 连接；别把权威区塞进 anycast VIP，也别让 Unbound 承担「主辅同步」。

这样，权威面的地理分离（对齐 `SC-22` 讨论）与递归面的 anycast 高可用，是两个独立可演练的故障域。

26.5　验收清单

项目	判定命令	通过标准
VIP 绑在 loopback	`ip -4 addr show dev lo | grep 192.0.2.53`	出现 `/32`，非 `/24`
本机 VIP 可解析	`dig @192.0.2.53 example.net A +short +time=1 +tries=1`	稳定返回 A 记录
密钥全集合一致	`sha256sum /etc/unbound/cookie-secret.key` 逐节点比对	校验和一致、权限 `600`
条件宣告生效	上游查看 `192.0.2.53/32` 与 `2001:db8::53/128`	仅在健康检查通过的节点出现
撤路由生效	手动停健康检查后查上游路由表	对应主机路由消失
跨节点 Server Cookie	连续查询并观察 OPT 伪段 COOKIE 选项	换节点后仍被接受
证书并存窗口	轮换期对两种端点分别握手（方法按官方手册检索）	新旧端点均可服务
观测汇聚	集中端拉各节点 `unbound-control stats`	全节点都有数据、无漂移
两平面分离	`unbound-checkconf` + 检查 `stub-zone` 指向	权威面不在 anycast VIP 内
回滚演练	按下一节逐项演练并留记录	有书面回滚点与演练记录

26.6 失败模式与回滚点

失败模式	现象	回滚点
密钥漏分发 / 不一致	换节点后 Server Cookie 校验失败、Cookie 保护消失	在 `activate_cookie_secret` 之前补齐 staging 重来
过早 `drop_cookie_secret`	持旧 Cookie 的客户端被拒	只在激活满 1 小时后删；删后只能重新 `add_cookie_secret` 加回旧密钥
健康检查假阴性	误撤路由，其余节点冷缓存抖动	放宽为连续 N 次失败才撤，手动 `announce_anycast_route`
健康检查假阳性	进程活着但不解析，VIP 成黑洞	脚本保留真实 `dig` 校验，补密钥文件存在性检查
路由未撤干净	流量仍打到故障节点	上游确认 withdraw，检查条件宣告是否接上健康检查
证书轮换不同步	部分节点新证书、部分旧证书	pin 回旧证书版本，逐节点重推，留并存窗口
配置 / 过滤表版本漂移	同一名字在不同节点答案不一致	pin 上一版本、幂等重放分发
冷缓存抖动	切流后上游 QPS 放大、延迟上升	预热 + serve-stale + 降低切流频率
控制端口暴露	观测通道变成攻击面	收回 `access-control`，控制通道只走受保护网络（第 9 章）

26.7 本章要点

- anycast 扩的是 Unbound 的**递归 / 验证层**，不是权威数据面；隐藏主服务器、一主多辅、辅升主属于权威面，Unbound 不做 AXFR / 主辅同步。
- 官方手册要求同一 anycast 集合**共享 `cookie-secret`**，用 `cookie-secret-file` 持久化；滚动必须按 `add_cookie_secret` → `activate_cookie_secret` → `drop_cookie_secret`，先全部 staging、再激活、激活满 1 小时才删旧。
- 证书轮换允许新旧配置并存，因为节点不会同一秒更新完；具体取值按名称在官方手册 / RFC Editor 检索，不臆断编号。
- `NIST SP 800-81r3` 给的是控制目标名称（专用服务器、异地权威、隐藏主、双递归器、anycast 隐藏单节点故障、书面辅升主、组件告警、对齐 `SC-22`），本章只写名称不编条款号。
- 工程上：VIP 绑 loopback + 条件路由宣告、健康检查失败即撤路由、缓存不共享的冷缓存抖动、配置分发幂等版本化、`unbound-control` 集中观测、灰度与明确回滚点、权威面与递归面分开。

26.8 参考资料

1. `https://unbound.docs.nlnetlabs.nl/en/latest/` —Unbound 官方在线手册入口（含 `unbound.conf`、`unbound-control` 手册页；`cookie-secret`、`cookie-secret-file`、`add_cookie_secret` / `activate_cookie_secret` / `drop_cookie_secret`、`tls-additional-port` 按名称检索）。
2. `https://nlnetlabs.nl/documentation/unbound/` —NLnet Labs: Unbound 文档与资料页。
3. `https://www.rfc-editor.org/rfc/rfc5737` —[RFC 5737]: 文档用 IPv4 地址块（本章示例地址依据）。
4. `https://www.rfc-editor.org/rfc/rfc1034` —[RFC 1034]: 域名概念与设施（主 / 辅区、区传送的背景）。
5. `https://csrc.nist.gov/` —NIST CSRC: `NIST SP 800-81r3`《Secure Domain Name System (DNS) Deployment Guide》与 `NIST SP 800-53`（按名称检索，`SC-22` 地理分离讨论）。
6. `https://www.rfc-editor.org/` —RFC Editor: DNS Cookie / anycast 相关规范按名称检索（本章不臆断规范编号）。

第二十七章

单播 DNS 与多播 DNS：从 mDNS 到 DNS-SD

一台机器在同一个网卡上其实同时开着两套名字系统：一套走 UDP/TCP 53，由递归解析器与权威服务器分工完成；另一套走 UDP 5353，只在同一段链路里喊话。它们共用 DNS 的报文格式，却不是同一条解析路径。第 9 章讲最小暴露面时，隐含前提之一是「名字查询都会经过 Unbound」；这一章要把这个前提拆开：**`.local` 那类名字根本不会到 Unbound 这里来**。

这里只讲部署边界：谁在哪个通道里应答、跨链路时该换哪条通道、以及企业网络里最常见的三种「时通时不通」。

27.1 两条互不相交的解析通道

27.1.1 一台网关上到底谁在监听

通道	端口	谁监听	谁应答	名字长什么样
单播 DNS	UDP/TCP 53	Unbound（或转发器）	递归解析器 / 权威服务器	`gw-01.home.arpa.`、`dns.example.org.`
多播 DNS	UDP 5353 组播	存根解析器（Avahi / systemd-resolved / mDNSResponder）	链路上拥有该名字的那台主机	`gw-01.local.`、`_demo._tcp.local.`
加密传输	853 / 443	Unbound 的 DoT / DoH 端点	递归解析器	与单播 DNS 同一命名空间

这张表的重点不是端口，而是「**谁应答**」**这一列换了人**：单播侧应答者是一台被配置出来的服务器，多播侧应答者是名字的拥有者本人。前者可以集中加固、集中观测，后者不能。

27.1.2 Unbound 的位置：它不在多播通道里

Unbound 不参与 mDNS：它是单播递归验证器，不加入 `224.0.0.251` / `FF02::FB` 组播组，不监听 UDP 5353 上的多播查询，也不为 `.local` 名字生成应答。第 25 章已经说过 Unbound 是递归验证器而不是权威服务器，`local-zone` / `local-data` 只是「非常有限的权威服务」；这个定位在多播场景里更明显——多播应答由「链路上拥有这个名字的那台主机自己」发出，与递归、缓存、转发全无关系。

把两条通道分开的**不是 Unbound**，而是存根解析器（stub resolver）：Linux 上常见 systemd-resolved 或 Avahi，macOS / Windows 侧常见 mDNSResponder。存根解析器决定「这个名字该往 53 单播发，还是往 5353 多播发」，Unbound 只收到前者。换句话说，分流发生在客户端，发生在查询离开网卡之前。

一条能立刻看清边界的命令：

```
ss -lunp | grep -E '(:53|:5353)'
```

预期里 `:5353` 的持有者是 avahi-daemon 或 systemd-resolved，`:53` 的持有者是 Unbound；**不应该出现 Unbound 占着 5353 多播**。若只看到 53 而没有 5353，说明本机没有 mDNS 应答者，`.local` 查询会整条落到单播侧，随后被当作一个不存在的顶级域处理。

27.2　mDNS：只在本链路里成立的名字

27.2.1　组播地址、端口与 TTL=1

mDNS [RFC 6762] 的固定参数很少，但每一个都是部署边界：

项	取值	部署含义
IPv4 组播地址	`224.0.0.251`	属于 `224.0.0.0/24` 本地网络控制块
IPv6 组播地址	`FF02::FB`	链路本地范围
传输	UDP 5353	与单播 53 完全分离
IPv4 TTL	1	路由器不转发；`224.0.0.0/24` 的报文天然不出本链路
专用后缀	`.local`	专用于本链路，不是可全球解析的顶级域

TTL=1 是最容易被误当成「配置没调好」的一条：隔了两个子网之后发现不了设备，**不是交换机没开组播、也不是防火墙漏了端口**，而是 mDNS 按定义就只在一条链路上有效。想跨子网，只能换到单播 DNS-SD（见后文），不是想办法把多播「打通」。把时间和精力花在调 IGMP / MLD snooping 上，方向就错了。

`.local` 被 [RFC 6762] 留给 mDNS，这带来一个长期运维坑：企业若把内部单播 DNS 区也叫 `.local`，存根解析器就可能在两条通道之间摇摆，表现为时通时不通。给家庭与小型内网用的名字是 `home.arpa.` [RFC 8375]，后面单独讲。

27.2.2　wire format 复用，语义不复用

mDNS 报文复用 DNS 的 wire format，所以 tcpdump / Wireshark 能直接解出来；但排障时若按单播 DNS 的语义去读这些位，几乎每个结论都会错。下面这张表是**部署排障用的对照**，不是条文复述：

字段	单播 DNS 的习惯	mDNS 多播查询 / 应答
Message ID	每查询一个随机数，用来配对请求与应答	恒 0；多播应答不靠 ID 配对
AA	权威服务器才置 1	应答恒 1——每个应答者对自己名字都是权威
RD / RA	存根置 RD=1 请求递归，递归器回 RA=1	恒 0 / 0，**mDNS 不做递归**
RCODE	NXDOMAIN / SERVFAIL 等承载错误	恒 0；「没这个名字」表现为「没人应答」
OPCODE	可承载 UPDATE 等非查询操作	非 0 的 OPCODE 静默忽略，不应答也不报错
TC	应答放不下时置位，提示改用 TCP	在**查询**里置位表示 Known-Answer 列表还有后续包

几个直接能用的排障推论：

- 抓包里看到 5353 上的应答没有对应 ID，别当错包；ID 恒 0 是规定动作。
- 拿 `dig` 的思路去等 NXDOMAIN 是等不到的：mDNS 里名字不存在 = 静默无应答，客户端靠超时收尾。
- 若看到 RD=1 的 5353 包，多半是某个实现把单播查询误投到多播口，或者两条通道被混在了同一条链路上。
- 应答者不会替你去问别人；「这个名字可能在别处」在 mDNS 里不成立，缓存由存根解析器自己维护。
- TC 位在查询里的含义与单播完全不同：看到它应该去找后续包里的 Known-Answer 记录，而不是准备切 TCP。
- AA=1 不代表链路上有一台权威服务器，只代表「发包的这台主机拥有它应答的名字」，别据此画拓扑。

27.2.3 probing、announcing 与 goodbye

一台主机想用 `gw-01.local.` 这个名字，不是直接开喊，而是先 **probing**：连续发出若干查询，问「有没有人已经在用这个名字」，同时监听是否有别人应答；确认无冲突后才 **announcing**，把 PTR / SRV / TXT / A / AAAA 记录主动多播出去（通常不止发一次，让链路上各缓存都收到）。运行中若发现同名冲突，按规则退让并重新选名。

服务正常下线时发 **goodbye**：把该记录的 TTL 置 0 后多播出去，等于告诉邻居「立刻把这条从缓存里删掉」。

部署上要注意 goodbye 的脆弱性：它是尽力而为的通告，没有确认、没有重传保证。主机断电、网线被拔、无线漫游的瞬间都发不出来，邻居缓存里就会残留一条已经不存在的记录，直到本地缓存老化。清理只能靠存根解析器侧的缓存超时，**不要指望 Unbound 帮 mDNS 刷缓存**——Unbound 从没收到过这些记录。这也是第 7 章讲的「链路上每一层缓存都要刷」在这里失效的原因：这一层根本不在 Unbound 的缓存里。

27.2.4 别把它叫成 LLMNR

LLMNR（Link-Local Multicast Name Resolution）是另一套链路本地名字解析协议，与 mDNS 不是同一个东西：端口不同、报文格式不同、服务发现模型也不同。运维里最常见的误操作是「关掉 LLMNR 就等于关掉 mDNS」，或者反过来。systemd-resolved 里这两个开关本来就是分开的（后面的配置块会把 `LLMNR=no` 与 `MulticastDNS=yes` 同时写出来），排查时也要分别看：

```
resolvectl status | grep -iE 'MulticastDNS|LLMNR'
```

若输出里 `MulticastDNS` 是 no 而 `LLMNR` 是 yes，那么关 LLMNR 对 `.local` 一点帮助都没有；反之亦然。把两者混为一谈，最典型的后果是「明明按文档关了链路本地解析，打印机还是能被发现」，然后开始在错误的方向上查防火墙。

27.3 DNS-SD：三件套，以及它并不依赖 mDNS

27.3.1 PTR / SRV / TXT 各管什么

DNS-SD [RFC 6763] 定义的是「怎么用 DNS 记录描述一个服务」，不是「怎么发多播包」。三件套分工固定：

记录	回答的问题	查询形态
PTR	这类服务有哪些实例?	`_demo._tcp.example.org.` PTR → `Demo Service._demo._tcp.example.org.`
SRV	这个实例跑在哪台主机、哪个端口?	`Demo Service._demo._tcp.example.org.` SRV → `edge-01.example.org.` + 端口
TXT	实例有哪些附加参数?	同上名的 TXT，键 = 值文本，例如 `note=demo`

之后才用 A / AAAA 把 `edge-01.example.org.` 解析成地址。注意这套记录**完全可以用普通的单播 DNS 区承载**：把 `_demo._tcp.example.org.` 的 PTR、SRV、TXT 写进权威区，客户端用常规 `dig` 就能完成服务发现，全程不碰 5353。这一点是理解后面三种跨链路方案的前提。

27.3.2 关键一句：DNS-SD 不依赖 mDNS

DNS-SD 不依赖 mDNS，mDNS 也不是 DNS-SD 的唯一承载方式。把两者绑成「Bonjour = mDNS + DNS-SD」是家庭网络里最常见的直觉，但在企业网里恰恰要拆开：

- 单链路、零配置：DNS-SD 记录由 mDNS 多播分发，最省事，也最不可控。
- 多子网、要审计、要权限：同一套 PTR/SRV/TXT 记录放进单播权威区，就是 Wide-Area DNS-SD。
- 设备多、电池供电、无线多播代价高：换成 SRP 的纯单播注册。

三种场景下记录形状是一样的，变的只有**传输与注册方式**。所以排障时先问「这条 PTR 是从哪来的」，比先问「组播通不通」有用得多。

27.3.3 Wide-Area DNS-SD：把发现搬回单播

Wide-Area DNS-SD 用 [RFC 2136] 的 DNS `UPDATE` 报文注册服务，并用 TSIG 做认证（TSIG 规范编号见官方资料，按名称在 RFC Editor 检索即可）。浏览侧仍然是普通的 PTR / SRV / TXT 单播查询，走常规递归解析器，因此能跨子网、能进日志、能按区做访问控制。

对本书的架构来说，这条路径最自然：内部权威区由 NSD 或微软 DNS 之类持有（第 25 章的三条路径），Unbound 用 `stub-zone` / `forward-zone` 指过去，只做单播递归与验证。但 **Unbound 自己不接受 `UPDATE` 报文**——[RFC 2136] 动态更新是权威服务器的能力，Unbound 没有。注册必须由客户端或 DHCP 脚本直接发给权威服务器，别把 `UPDATE` 往 Unbound 的 53 口发。

27.3.4 Discovery Proxy：跨链路的翻译器

Discovery Proxy [RFC 8766] 解决的是「设备只会多播、客户端在另一个子网」这个过渡问题：代理身处设备所在的链路，替单播客户端发多播查询，把收到的多播应答整理成单播 DNS-SD 记录再回给客户端。它本质上是一个**通道翻译器**，不是新的名字系统，也不是中继转发器——它输出的是可被单播缓存、可被递归的标准 DNS-SD 记录。

部署边界：代理必须在每条需要被发现的链路上都有一个「腿」，单播侧则可以共用一套权威区与递归器。它适合打印机、摄像头这类改不了协议栈的设备；新设备应优先直接走 SRP 或 Wide-Area 注册，少一层状态。

27.3.5 SRP：为 Wi-Fi 与低功耗网络做的纯单播注册

SRP [RFC 9665]（Service Registration Protocol）是**纯单播**的 DNS-SD 注册协议：设备用 DNS Update 把 PTR / SRV / TXT 提交给 SRP 注册服务器，认证用 SIG(0) 公钥而不是共享密钥，命名采用 FCFS（First Come, First Served，先到先得）规则来处理同名冲突。设计动机很直接：在 Wi-Fi 与低功耗、电池供电的网络里，多播收发既费电又容易被无线侧丢弃或限速，与其优化多播，不如彻底不发多播。

对部署的含义：设备侧没有 probing / announcing 那一套，名字冲突由注册服务器按 FCFS 裁定，浏览侧仍是单播 DNS-SD。于是「跨子网发现」不再需要打通多播，也不需要每条链路放一个代理，规模可以上到大量终端。密钥分发与 FCFS 命名策略是新的运维面，别把它当成零配置方案。

27.3.6 LLQ：见 RFC Editor 索引

长轮询（Long-Lived Queries，LLQ）用于让单播客户端及时知道服务记录的变化，避免反复轮询。编号与当前状态**见 RFC Editor 索引 / 按名称在 RFC Editor 检索**，本章不写具体编号与条款；部署时优先确认自己用的注册服务器与客户端是否都支持该机制，而不是假定它默认可用。

27.4 四维对照表

传输、作用域、注册方式、递归、谁应答、适用规模，把这四种方案一次摆平：

方案	传输	作用域	注册方式	递归	谁应答	适用规模
单播 DNS	UDP/TCP 53（可承载 DoT / DoH / DoQ）	路由可达即全网	zonefile、[RFC 2136] UPDATE + TSIG，或资产清单生成 `local-data`	有，递归解析器逐级查询	权威服务器 + 递归解析器	单台到全球 anycast 集群（第 26 章）
mDNS	UDP 5353 组播 `224.0.0.251` / `FF02::FB`，TTL=1	单一链路	主机自行 probing / announcing，goodbye 用 TTL=0	**无**，RD/RA=0	链路上拥有该名字的那台主机	一个广播域，几十台以内
Wide-Area DNS-SD	单播 UDP/TCP 53	跨子网、跨园区	[RFC 2136] UPDATE + TSIG 写入权威区	有，走常规单播递归	单播权威服务器（可叠加 Discovery Proxy）	园区、多子网、需要审计与权限

方案	传输	作用域	注册方式	递归	谁应答	适用规模
SRP	纯单播 DNS Update	由注册服务器决定，可跨链路	DNS Update + SIG(0) 公钥，FCFS 命名	有，单播侧	SRP 注册服务器 / 代理	Wi-Fi、低功耗与电池设备、大量终端

读表时抓两条：**只有 mDNS 那一行的「递归」是无；只有 mDNS 那一行的「谁应答」不是服务器**。其余三行都落在本书前 26 章已经覆盖的单播架构里，只是记录内容换成了 DNS-SD 三件套。

27.5 一段可复制的教学配置：把分流钉死

下面用 Avahi 发布一个示例服务，再用 systemd-resolved 把 `.local` 与单播名字分开，最后确认 Unbound 只管单播侧。三块配置各自紧跟「为什么 / 怎么验证 / 失败模式与回滚点」，可按顺序照抄。

27.5.1 前置假设

- 网关单播侧地址 `192.0.2.53`（Unbound 监听 53），LAN 段 `192.0.2.0/24`；IPv6 用 `2001:db8::/32` 中的地址。
- 内部单播域用 `home.arpa.`，对外示例域用 `example.org.`；**不使用 `.local` 作为单播区名**。
- 参与演示的主机名统一写作 `gw-01`、`edge-01`、`cloud-01`。

27.5.2 配置块一：限定 Avahi 只在 LAN 接口上说话

`/etc/avahi/avahi-daemon.conf`：

```
[server]
use-ipv4=yes
use-ipv6=yes
allow-interfaces=lan0
deny-interfaces=wan0
domain-name=local
enable-dbus=yes

[publish]
publish-addresses=yes
publish-hinfo=no
publish-workstation=no

[reflector]
enable-reflector=no
```

①为什么这么设：`allow-interfaces` 把多播限制在 LAN 一侧，避免 mDNS 通告从 WAN 口漏出；`domain-name=local` 明确后缀；`enable-reflector=no` 是关键——反射器会把多播在多个接口间转发，等于人为把「链路本地」改成了「跨链路」，既违背 TTL=1 的边界，也会和后面的单播 DNS-SD 抢同一批名字。这一项默认就该关。

②怎么验证（一条可复制命令）：

```
avahi-browse -art | grep -i '_demo._tcp'
```

③失败模式 / 回滚点：若 `avahi-browse` 输出为空但服务进程在跑，先查 `deny-interfaces` 是否把 LAN 口也排除了，再查防火墙；反射器误开时会在多个网段看到同一批实例，回滚就是把 `enable-reflector` 改回 `no` 并重启 avahi-daemon。任何改动前先备份该配置文件，`systemctl restart avahi-daemon` 是回滚点。

27.5.3 配置块二：发布一个示例服务

`/etc/avahi/services/demo.service`：

```
<?xml version="1.0" standalone='no'?>
<!DOCTYPE service-group SYSTEM "avahi-service.dtd">
<service-group>
  <name replace-wildcards="yes">Demo Service on %h</name>
```

```
  <service>
    <type>_demo._tcp</type>
    <port>12345</port>
    <txt-record>note=demo</txt-record>
    <txt-record>site=example.org</txt-record>
  </service>
</service-group>
```

①**为什么这么设**：这是 DNS-SD 三件套的本地发布形式——<type> 对应 PTR 的服务类型，<port> 与主机名一起进 SRV，两条 <txt-record> 进 TXT。TXT 里只放元数据，不放地址；示例域 example.org 只作为标记，不参与解析。用 _demo._tcp 而不是真实服务类型，避免和链路上真实设备撞名。

②**怎么验证**（一条可复制命令）：

```
avahi-resolve -n "gw-01.local" && avahi-browse -rt _demo._tcp
```

③**失败模式 / 回滚点**：avahi-resolve 超时说明 .local 没被存根解析器交给 mDNS（回到前面 resolvectl status 那一节），此时查服务文件没有意义；avahi-browse 能看到实例但解析不出地址，通常是多播被防火墙挡在入向。回滚点：删除该 .service 文件并重启 avahi-daemon，Avahi 会发出 goodbye（TTL=0）；若进程被强杀没来得及发，邻居侧残留记录只能等缓存老化，这一点前面已经说明。

27.5.4　配置块三：systemd-resolved 明确两条通道

/etc/systemd/resolved.conf.d/10-mdns-split.conf：

```
[Resolve]
DNS=192.0.2.53
Domains=home.arpa. example.org.
MulticastDNS=yes
LLMNR=no
```

①**为什么这么设**：DNS= 把单播查询固定指向 Unbound 的文档示例地址；Domains= 只列单播侧的名字，**.local 不写进去**——它由存根解析器内部判定走 mDNS 通道，写进路由域反而会把 .local 强行送去单播，制造后面要讲的抢解析。MulticastDNS=yes 与 LLMNR=no 并列，正是为了把两套链路本地协议分开，避免「关错了开关」。不同发行版与版本的默认值有差异，取值见官方手册对应章节。

②**怎么验证**（一条可复制命令）：

```
resolvectl query gw-01.home.arpa. && resolvectl status | grep -iE 'MulticastDNS|LLMNR|DNS Servers'
```

③**失败模式 / 回滚点**：gw-01.home.arpa. 解析失败而 .local 正常，说明单播路由域没生效或 Unbound 未监听 192.0.2.53；反过来 .local 落到单播、返回 NXDOMAIN，说明 .local 被误写进了 Domains=。回滚点：删除该 drop-in 文件后 systemctl restart systemd-resolved，并确认 /etc/resolv.conf 仍指向存根解析器而不是被别的管理工具覆盖。

27.5.5　配置块四：Unbound 只补单播侧，绝不碰 .local

/etc/unbound/unbound.conf.d/10-internal.conf：

```
server:
    interface: 192.0.2.53
    access-control: 192.0.2.0/24 allow

local-zone: "home.arpa." static
local-data: "gw-01.home.arpa. IN A 192.0.2.1"
local-data: "edge-01.home.arpa. IN A 192.0.2.2"
```

①**为什么这么设**：这两条 local-data 只服务于单播内部域，是第 25 章「静态生成 local-data」路径的最小例子。**故意不为 .local 配任何 local-zone / local-data**：一旦配上，Unbound 就会开始对 .local 给出权威性应答，与 mDNS 形成两条都自称权威的通道，「时通时不通」就从别人的问题变成自己的配置问题。第 9 章的默认本地区清单要与这里的取舍一起看，别把清单里的保留名当成可随便接管的可写区。

②**怎么验证**（一条可复制命令）：

```
unbound-checkconf && dig @192.0.2.53 gw-01.home.arpa. A +short
```

③**失败模式 / 回滚点**：unbound-checkconf 报错就别 reload，先按行号修配置；dig 能拿到 192.0.2.1 说明单播侧正常。再补一条反向验证：

```
dig @192.0.2.53 gw-01.local. A +short
```

预期是**拿不到 A 记录**；无论返回 NXDOMAIN、SERVFAIL 还是 REFUSED，都说明这次查询走的是单播而不是 mDNS，边界成立。若这里返回了一条地址，说明有人给 Unbound 塞了 .local 数据，或者上游单播 DNS 占了 .local——立刻按下一节排查。回滚点：删掉相应 local-zone / local-data 行并 reload，配置管理侧同步回滚源文件。

27.5.6 验证命令汇总

```
# 1. 谁在监听，确认 Unbound 不在 5353 上
ss -lunp | grep -E '(:53|:5353)'

# 2. 存根解析器的两个开关是不是各自独立
resolvectl status | grep -iE 'MulticastDNS|LLMNR'

# 3. 多播侧能不能看到实例与地址
avahi-browse -art
avahi-resolve -n "gw-01.local"

# 4. 单播侧能不能拿到内部名字
dig @192.0.2.53 gw-01.home.arpa. A +short
dig @192.0.2.53 _demo._tcp.example.org. PTR +short

# 5. 抓包看 5353 与 224.0.0.251 上有没有真实流量
tcpdump -ni any 'udp port 5353'
```

第 5 条最有说服力：把一台客户端接进 LAN 后触发一次发现，若 -ni any 里完全没有 5353 包，那么无论 avahi-browse 报什么，问题都在链路或防火墙，不在 DNS 记录本身。

27.6 边界与排障：企业里最常见的三种「时通时不通」

27.6.1 一、.local 被企业单播 DNS 占用

[RFC 6762] 把 .local 留给 mDNS；但历史上不少企业内部单播区也叫 .local。结果同一名字有两条通道都可能应答：存根解析器按实现与路由域的不同，有时把查询送到 5353、有时送到 53。表现就是典型的时通时不通——同一台打印机，上午能连、下午 NXDOMAIN；同一条命令，在 A 机器上成功、在 B 机器上失败。

排查顺序：

```
# 看这个名字被判定到哪条通道
resolvectl query gw-01.local

# 看单播侧是否真的有人在应答 .local
dig @192.0.2.53 gw-01.local. A +short

# 看多播侧有没有包
tcpdump -ni any 'udp port 5353'
```

结论只能有一个：**不要再让 .local 同时属于两套系统**。要么把内部单播区迁到 home.arpa. 或组织自己的子域（example.org. 下的内部子域），要么在存根解析器上把 .local 固定交给 mDNS 并保证单播侧不给它权威应答。不要试图用 Unbound 的 local-zone 去「补」这个洞，那只会再加一个应答者。

27.6.2 二、防火墙整体禁用 mDNS

企业里 mDNS 常被防火墙整体禁用。微软文档给出的做法是禁用入站 mDNS (UDP-In) 规则（按标题在 Microsoft Learn 检索可见，具体规则名与路径见官方资料）。这类策略一开，链路上 5353 的入向应答全被丢掉，avahi-browse 列表为空，但单播 dig 完全正常。

要分清两种「看不到」：

现象	单播 `dig`	5353 抓包	判断
防火墙禁 mDNS	正常	有查询、无应答	链路被剪，名字本身没错
没有 mDNS 应答者	正常	无查询、无应答	本机压根没开多播解析
`.local` 被单播占用	有时返回地址	时有时无	两条通道在抢解析

跨子网才靠单播 DNS-SD：这一条要和 TTL=1 连起来读。企业禁掉 mDNS 之后，跨子网的发现需求不会消失，它会转到 Wide-Area DNS-SD（UPDATE + TSIG 写权威区）、Discovery Proxy（翻译已有设备）或 SRP（纯单播注册）上。也就是说，禁多播不是把服务发现关掉，而是把它从「不可控的链路行为」搬回「可审计的单播区」——这正是本书主张的架构方向。

27.6.3　三、跨子网还想发现：不是打通多播

第三类误判是把「跨 VLAN 发现不了」当成交换机配置问题，于是去开组播反射、去调 snooping、去放通 `224.0.0.251`。这些做法即使短期有效，也把 mDNS 的链路本地前提破坏了：同一名字会在多条链路上被不同主机争用，goodbye 包又跨不过去，残留记录越积越多。

正确做法按设备类型分：

- 改得了协议栈的新设备→直接 SRP 纯单播注册，或 Wide-Area DNS-SD 的 `UPDATE` + TSIG。
- 只会上多播的老设备→在它所在链路放 Discovery Proxy [RFC 8766]，单播侧共用权威区与 Unbound。
- 只想浏览、不需要注册→单播侧查 PTR / SRV / TXT 三件套即可，不需要任何 5353 流量。

27.6.4　`home.arpa.` 才是该用的名字

`home.arpa.` [RFC 8375] 是给家庭与小型内网专门留的名字，不会与 mDNS 的 `.local` 抢解析，也不会与公网可解析的域混淆。它和第 9 章的默认本地区清单是同一件事的两面：第 9 章管「哪些名字不该出网」，这里管「内部名字该叫什么」。

落地时的三条建议：

- 内部单播区统一用 `home.arpa.`（家庭/小型）或组织自有域的内部子域，不用 `.local`。
- Unbound 侧用 `local-zone` + `local-data` 静态承载，或按第 25 章用 `stub-zone` 指向 NSD / 微软 DNS 权威区；`home.arpa.` 不该出现在公网递归路径上。
- 存根解析器的 `Domains=` 只列单播名字，`.local` 留给 mDNS，两边互不越界。

把名字选对，前面三类「时通时不通」里至少两类在部署阶段就消失了。剩下的那一类——防火墙策略——反而是最容易验证的：一条 `tcpdump -ni any 'udp port 5353'` 就能给出答案。

27.7　本章要点

1. **Unbound 不参与 mDNS**：不监听 5353 多播、不应答 `.local`；分流由存根解析器（systemd-resolved / mDNSResponder / Avahi）在客户端完成，Unbound 只收到单播那一半。
2. **mDNS 是链路本地、无递归的另一套名字系统**：`224.0.0.251` / `FF02::FB` / UDP 5353 / TTL=1；wire format 复用但语义不同（Message ID 恒 0、AA=1、RD/RA=0、RCODE 恒 0、非 0 OPCODE 静默忽略、TC 表示 Known-Answer 续包），名字靠 probing / announcing 争用、靠 goodbye（TTL=0）撤销。
3. **DNS-SD 不依赖 mDNS**：PTR / SRV / TXT 三件套可以完整跑在单播 DNS 上；跨链路用 Wide-Area DNS-SD（[RFC 2136] UPDATE + TSIG）、Discovery Proxy（[RFC 8766]）或 SRP（[RFC 9665] 纯单播、SIG(0) 公钥、FCFS 命名），LLQ 见 RFC Editor 索引。
4. **`.local` 只能属于一条通道**：被企业单播 DNS 占用就会与 mDNS 抢解析，表现为时通时不通；家庭与小型内网该用 `home.arpa.` [RFC 8375]，与第 9 章默认本地区清单一起规划。
5. **企业禁用 mDNS 是常态**：禁用入站 `mDNS (UDP-In)` 规则后，跨子网发现应转到单播 DNS-SD，而不是去打通多播；排障先用 `ss -lunp`、`resolvectl status`、`tcpdump -ni any 'udp port 5353'` 分清是链路、开关还是名字冲突。
6. **LLMNR 不是 mDNS**：两者端口、报文与发现模型都不同，开关必须分别设置、分别排查，别把「关掉了 LLMNR」当成「关掉了 mDNS」。

27.8 参考资料

1. RFC 6762, *Multicast DNS*: https://www.rfc-editor.org/rfc/rfc6762
2. RFC 6763, *DNS-Based Service Discovery*: https://www.rfc-editor.org/rfc/rfc6763
3. RFC 2136, *Dynamic Updates in the Domain Name System (DNS UPDATE)*: https://www.rfc-editor.org/rfc/rfc2136
4. RFC 8766, *DNS-SD Discovery Proxy*: https://www.rfc-editor.org/rfc/rfc8766
5. RFC 9665, *Service Registration Protocol for DNS-Based Service Discovery*: https:// www.rfc-editor.org/rfc/rfc9665
6. RFC 8375, *Special Use Domain* '*home.arpa.*': https://www.rfc-editor.org/rfc/rfc8375
7. Unbound 官方手册（章节与默认值以此为准）: https://unbound.docs.nlnetlabs.nl/en/latest/
8. Microsoft Learn（按标题检索 `mDNS (UDP-In)` 入站规则与禁用方式）: https://learn.microsoft.com/

第二十八章

DNS 故障域隔离：从一次全球性中断说起

把 2021 年 10 月 4 日那次近 6–7 小时的中断，记成「内部 DNS 和公网用了同一个域名，DNS 没隔离，所以炸了」，是一句很顺手的总结，也是**不准确的归因**。它把一场路由与控制面的连锁事故，压缩成了一个命名空间配置问题；按这个结论去整改，会去做 split-horizon、改几个内部域名，却在真正断掉的那一层什么都没动。

28.1 先把归因摆正：不是「同域名」

28.1.1 官方确认的因果链

Meta 工程博客 2021-10-05 的复盘《More details about the October 4 outage》给出的链条是五步，顺序不能调换：

```
[1] 例行骨干网维护：一条「评估全球骨干容量」的命令
        ↓ 意外生效
[2] 所有数据中心之间的连接被切断；命令审计工具里的 bug 没拦住
        ↓ 数据中心不可达
[3] 小型设施里的权威 DNS 服务器连不上数据中心
        → 按设计「判定自己不健康」
        → 主动撤回自己的 BGP 路由宣告
        ↓ 前缀从全球路由表消失
[4] facebook.com / instagram.com / whatsapp.net 的权威 NS 不可达
    （服务器进程仍在运行，是「不可达」不是「宕机」）
        ↓ 修复动作本身依赖网络与名字解析
[5] 官方明说的两个恢复障碍：
      a. 正常网络访问与带外（OOB）网络访问都断了
      b. DNS 全面失效，打断了大量内部工具（排障/修复要用的工具自己依赖 DNS）
   → 只能派工程师物理到场；数据中心的高物理/系统安全设计
     拖慢了进场与改设备，官方认为这个「日常高安全 vs 罕见事件慢恢复」
     的取舍值得拿出来讨论
```

注意第 [3] 步：撤路由是**健康检查按设计做出的正确动作**，不是配置失误。权威服务器判断「我连不上数据中心，我不该继续对外宣告」，这个逻辑本身没问题；出问题的是「内部可达性、外部可达性、控制面可达性」被绑在了同一个判定条件上。第 [5] 步才是这次事故从「一次路由故障」升级成「近 6–7 小时全局中断」的关键：**修东西要用的路，和坏掉的东西是同一条路**。

28.1.2 必须写明的一处纠正

官方复盘里**没有**「内外同域名导致 split-horizon 误配」这一条。既没有说内部视图配错，也没有说因为内部名字和对外名字重名而解析串了。把「同域名」写成根因，等于凭空给官方结论加了一个不存在的环节。

正确的提法分两层，别混：

- **根因不是「同域名」，而是内外共用同一套名字解析基础设施与同一故障域**：骨干网断连→权威 DNS 自动撤 BGP 路由→内部工具、认证、带外管理跟着一起失效。这是本书据官方链条做出的**归纳**。
- 「**内部名字若挂在可公开解析的同一命名空间下，且没有独立的内部权威与内部视图，就会把内部系统的可用性绑到公网路由与公网权威的可达性上**」，是从案例里提炼的**设计推论**，不是已确认根因。它是一条值得照着做的设计教训，不能倒过来当成「官方说事故原因是这个」。

这个区分不是抠字眼。若当成根因，整改会停在「换个内部域名」；若当成推论，整改会落到「把权威面、递归面、控制面的故障域切开」。后者才是本章后面六层要做的事。

顺带把结论的性质钉死：这是**架构问题**，与选用哪个 DNS 软件无关。换成任何一款权威服务器、任何一款递归解析器，只要「权威可达性判定」「内部工具的解析依赖」「带外通道」「进门凭据」共用一个故障域，同样的链条仍然成立。这里不做产品批判，也不给任何实现打分；Unbound 在本书里的定位始终是**递归验证器，不是权威服务器**，`local-zone` / `local-data` 只是「非常有限的权威服务」。

28.2　证据分级表：四类，不许混

写复盘最容易出的错，是把外部观测当成官方结论、把记者的现场描述当成厂商逐条确认、再把自己的推断写成「由此可见，根因是……」。下面这张表把本章用到的每一条陈述逐条归类，后面所有论述都按这个口径引用。

#	陈述	证据等级	来源 / 说明
A1	例行骨干维护中一条评估全球骨干容量的命令，意外切断所有数据中心互联	官方确认	Meta 工程博客复盘
A2	命令审计工具里的 bug 没能拦住该命令	官方确认	同上
A3	小型设施里的权威 DNS 连不上数据中心，按设计判定不健康并**主动撤回 BGP 路由宣告**	官方确认	同上
A4	对外业务域的权威 NS 从全球路由表消失；服务器仍在运行但不可达	官方确认	同上
A5	正常与带外网络访问都断；DNS 全面失效打断大量内部工具	官方确认	同上，明列为两个恢复障碍
A6	只能派工程师物理到场；高物理/系统安全设计拖慢恢复	官方确认	同上
B1	15:51 UTC 内部报告公共解析器返回 SERVFAIL；15:58 UTC 发现 DNS 前缀路由被撤回	外部观测	Cloudflare 博客《Understanding how Facebook disappeared from the Internet》
B2	全球用户/应用重试使解析查询量约 **30 倍**	外部观测	同上；全球解析器缓存 SERVFAIL
B3	15:39 UTC 起数百条 BGP 撤回，**7 个覆盖内部 DNS 的 IPv4 前缀消失**（含更具体的 /24）	外部观测	ThousandEyes《Facebook Outage Analysis》
B4	**控制面被切断**才是恢复慢的根	外部观测（分析判断）	同上
B5	「内部 DNS 可能是认证服务的依赖」	外部观测（原文即推测）	同上；**不得写成已证实**
B6	`.com` / `.net` 权威侧对相关域的查询从约 **7,000 qps 涨到 900,000 qps（>100 倍）**	外部观测	Verisign 博客《Observations on Resolver Behavior During DNS Outages》
B7	相关域 TTL 只有 1–5 分钟，缓存几分钟内耗尽	外部观测	同上
B8	呼吁递归器在「权威全不响应但父区委派正常」时限流 + 指数退避	外部观测（建议）	同上；是建议，不是已部署事实

#	陈述	证据等级	来源 / 说明
C1	员工工牌失效、无法进入建筑	**公开报道**	NYT 记者 Sheera Frenkel 的公开报道，**非 Meta 官方逐条确认**
C2	派工程师到现场	**公开报道**	The Verge 报道，与官方 A6「物理到场」互相印证
D1	根因是内外共用同一套名字解析基础设施与同一故障域	**本书推论**	由 A1–A5 归纳，官方无此措辞
D2	内部名字挂在可公开解析的同一命名空间且无独立内部权威/视图→内部可用性被绑到公网路由与公网权威	**本书推论（设计教训）**	见上节，不是已确认根因
D3	「DNS 故障→认证/门禁失效」的完整依赖链	**本书推论（不作事实断言）**	由 B5 的推测 + C1/C2 拼出的可能性，中间环节未公开
D4	本章六层隔离框架、注入清单、恢复顺序	**归纳框架**	非任何官方规范

28.2.1 推论就是推论

表里 D 类四条，写作上必须自带标签，不能出现「由此可见，事故原因是……」这类把推断伪装成结论的句子。尤其 D3：门禁与工牌的失效，**公开可核验的只有记者的现场描述**（C1）与「派工程师到场」（C2、A6），以及 ThousandEyes 那条标着推测的分析（B5）。把这三者拼成「因为内部 DNS 挂了所以工牌刷不开」是一条合理的假设，适合拿来做演练输入，**不适合写成事实**。

同理，B2 的 30 倍与 B6 的 100 倍是两处不同观测点的放大倍数，前者看的是公共递归器的入向查询，后者看的是顶级域权威侧的查询，不要相加、不要当成同一个数字。B4 的「控制面被切断」是 ThousandEyes 的分析判断，不是 Meta 对自身控制面的说明。

28.3 六层隔离框架

下面六层是本书为「把故障域切开」自定义的检查框架，标注为**归纳**：它不来自某份标准，也不宣称覆盖全部风险。顺序有意为之——先切名字，再切权威，再切递归，然后才轮到控制面与失败语义；因为前三层的隔离再干净，只要第四层的「进门」还绑在同一条路上，第五、六层的预案就执行不了。

28.3.1 ①命名空间隔离

目标是让内部名字的可解析性**不依赖公网命名空间的可达性**。

- 内部用独立且**不可公网解析**的域：`home.arpa.`（[RFC 8375] 为家庭/小型内网保留）或自己持有的域下划出的**内部专用子域**，配合本地区与内部分区。不要复用对外业务主域。
- 内部专用子域要有明确边界：对外权威上不出现该子域的委派，内部递归器上把它钉到内部权威，公共视角查不到记录。
- split-horizon（同一名字按来源给两套答案）是可选工具，**不是隔离本身**。它的代价要写进运维手册：同一个 FQDN 在两个视图里答案不同，排障时必须先问「这次查询来自哪个视图」，缓存、`dig` 结果、工单记录都会出现「时通时不通」的表述；视图数量一多，变更流程与审计日志都要跟着翻倍。
- 明确不做的事：不靠「改个内部域名」解决可达性。名字换了，若权威面和路由面仍共用，故障域没变。

第 27 章会给出另一条腿：mDNS 的 `.local` 是链路本地多播名字，本链路自洽，天然不依赖单播权威与公网路由；但它出不了子网，也**不能替代**内部权威。

28.3.2 ②权威面隔离

目标是让「对外权威不可达」不自动等价于「内部名字也解析不了」。

- 内部权威（NSD、AD 集成区等）与对外权威**分开部署、分开路由宣告、分开变更流程**。第 25 章讲的内部名字产生路径，落点就是这台内部权威；Unbound 用 `stub-zone` / `forward-zone` 指过去，自己不承担权威角色。

- 隐藏主服务器（hidden primary）：真正可写的主不对外宣告，对外只见辅；变更先在主上做，再分发。
- 权威至少两台，分处不同网段与不同地理位置。NIST SP 800-81r3 的部署指导里有同类控制目标（专用 DNS 服务器、每区一主多辅、至少两台异地权威、隐藏主、书面的辅升主流程与组件告警）；本章只引**控制目标名称**，不引条款号，具体条文见官方资料。
- 关键边界：健康检查的判定条件要问清楚「健康」指的是什么。案例里权威服务器「连不上数据中心」就撤自己的路由，等于把跨数据中心的 backbone 状态当成了自身服务可用性的代理指标。内部权威的宣告条件里，不应包含「能否连上对外业务系统」这类同故障域依赖。
- 变更流程分开还有一层意思：对外权威的一次误操作，不应同时改动内部权威的区数据或路由策略。配置与区数据分发要版本化、可回滚，见第 26 章。

28.3.3　③递归面隔离

目标是让内部客户端的解析**不托管给对外权威或外部网络**。

- 内部递归解析器独立部署，不由对外权威服务器兼任，也不放在「只有公网路由可达」的位置。存根解析器至少配两个递归器地址（第 8 章的威胁模型与第 9 章的访问控制给出基线）。
- 本地根区镜像 / `auth-zone` 让根这一跳不出口（第 6 章）：公网递归链路抖动时，内部仍有根层数据。注意 `auth-zone` 是**只读根镜像，不是可写权威区**，别拿它当内部权威用。
- 缓存与 TTL 撑过上游抖动：第 4 章的最小可用配置、第 7 章的多实例与可观测性，加上 `serve-expired`（过期缓存应答，即 serve-stale 机制，[RFC 8767]），构成「上游短暂不可用时仍给答案」的能力。
- 这里有个常被忽略的边界：递归面隔离**不提供**控制面隔离。递归器再冗余，若工程师的登录、工牌、带外终端仍要靠同一个名字空间进来，第④层空缺照样会把恢复卡住。

28.3.4　④控制面隔离

这一层是官方 A5、A6 直接指出的缺口，也是本书前 27 章几乎没碰的一层。

- 带外管理（OOB）要有**独立路径与独立名字**：独立链路、独立地址段、独立的静态名字解析（本地 `hosts` 或 OOB 自带的小型解析），不查业务 DNS，不走业务路由。案例里「正常与带外网络访问都断」说明两条路在某处共用了物理/路由资源，OOB 的「外」必须是真的外。
- 门禁 / 认证 / 工牌系统要有**本地校验与离线兜底**：门禁控制器能在与中心认证断开时按本地缓存的白名单放行应急人员，而不是「刷不开就是拒」。这是从 C1、C2、B5 提炼的**演练假设**（D3），不是对事故原因的断言。
- **物理到场预案**：谁有权限进场、备用凭据放在哪里、离线怎么联系、设备控制台怎么接、多久能到现场。A6 明说高物理/系统安全设计拖慢了恢复，所以预案要提前把「安全流程在应急下如何走」写清楚，而不是出事时临时找人签字。
- 一句实话：这一层**超出 DNS 书的范围**。DNS 章节能做的是「不让 DNS 成为进门的前置依赖」，做不了门禁控制器与身份系统的设计。书里只标注接口与验收条件，实现交给机房与身份团队。

28.3.5　⑤缓存与失败语义

目标是把「权威不可达」从一次局部抖动，限制成一次局部抖动，而不是放大成 100 倍查询风暴。

- TTL 策略：B7 观测到 1–5 分钟的 TTL 让缓存几分钟内耗尽。对外业务记录给极短 TTL，等于把权威侧的可用性直接暴露给全球重试流量；关键内部记录则应留足可用窗口。TTL 是**故障域的时间维度**。
- 负缓存（[RFC 2308]）：上游瞬时 NXDOMAIN 会按 SOA minimum 被缓存，修好权威后客户端仍失败；本书第 4/7 章与第 22 章案例都踩过，链路上**每个实例都要刷**。故障期尤其要避免把 SERVFAIL/NXDOMAIN 长时间钉在缓存里。
- SERVFAIL 限流与指数退避：B8 的建议正对 B2、B6 的放大。递归器在「父区委派正常、但权威全不响应」时应收敛重试而非线性重发；客户端侧的应用重试同样要有抖动退避。这里没有「一个开关买齐」的做法，参数名称与默认值**见官方手册对应章节**，本节只表达策略意图。
- `serve-expired` 与限流是一对，不是二选一：前者让已有缓存继续服务，后者让没有缓存时不去压垮上游。两者都要配合可观测性（第 7 章）确认真的生效。

下面这个短片段只用来示意第③、⑤层的衔接，是**教学用示意**，不是完整配置：

```
# /etc/unbound/unbound.conf.d/99-fault-domain.conf（节选，教学用示例）
server:
    serve-expired: yes
    serve-expired-client-timeout: 1800
```

为什么这么设：`serve-expired: yes` 允许上游不可达时回一份过期缓存应答，把「解析全失败」降级为「答案稍旧」；`serve-expired-client-timeout` 的**单位是毫秒**（1800 即 1.8 秒），语义是「先尝试刷新，超时才回过期缓存」，不是 1800 秒。过期应答的保留上限还有单独参数控制，**名称与默认值见官方手册对应章节**，本节不代填。

怎么验证：断开上游前先预热一条记录，再断链并观察 TTL 与应答标志——

```
dig @192.0.2.53 www.example.org A +noall +answer +comment
```

断链后应答仍在、TTL 明显偏小或为 0，即说明过期缓存应答在起作用；参数名与默认值以官方手册对应章节为准。

失败模式 / 回滚点：过期缓存会掩盖上游已恢复的事实，负缓存还可能把瞬时的 NXDOMAIN 钉住；恢复后必须按「控制面→权威→递归→客户端」的顺序逐层刷缓存（`unbound-control` 相关命令见第 7 章），再复测。回滚只需把该 include 文件移除并 reload，配置管理里保留上一版。

28.3.6 ⑥演练与可恢复性

前五层是设计，第六层是**证明设计成立**。没注入过故障的隔离，只能算图纸。

- 注入清单至少覆盖四类：**断网**（业务链路）、**撤路由**（模拟权威前缀消失）、**关权威**（内部/对外分别关）、**关认证**（中心认证不可达）。每类都要有明确的预期可观测结果与回滚点，见下一节表格。
- **恢复顺序固定：先控制面，再 DNS，再业务**。理由直接来自 A5：DNS 是修复动作的前置依赖，业务先起来而排障工具仍不可用，只会把事故拖长；控制面（OOB、门禁、登录、离线凭据）先可用，工程师才进得去、改得动。
- 演练前钉死三件事：**谁有权限**（变更与撤路由的授权人）、**怎么联系**（不依赖业务 DNS 与业务网络的通信录，纸质或离线副本）、**离线凭据**（`<TOKEN>` 类凭据与应急账号的存放位置、有效期、轮换记录）。这三项恰恰是 A6 里拖慢恢复的部分。
- 演练要记录「实际恢复耗时」与「预期耗时」的差，差在哪一层就把哪一层拎出来重做。别把演练做成「跑一遍命令都绿了」——绿的是命令，不是恢复能力。

28.4 映射到前 27 章

本章是前 27 章的**失败模式总检验**：把每一章已经做过的隔离，放回「一次骨干断连 + 权威撤路由 + 控制面一起失效」的场景里，看它到底挡住了哪一层。

章	主要提供	对应隔离层	在这个场景下能挡住什么 / 挡不住什么
第 25 章内网 DDNS 与微软 DNS 对照	内部名字的产生路径：外部权威 + `nsupdate`（[RFC 2136]）、NSD 权威 + Unbound 递归、静态生成 `local-data`	①命名空间隔离、②权威面隔离	独立内部区 + 内部视图让内部名字不去公网解析；但「更新必须落到主服务器」「AD 集成区发给任意 DC」说明**写路径仍依赖可达性**，主区不可达时动态更新停摆，需靠静态产物兜底
第 26 章企业级大规模部署：anycast 与集群	anycast 服务地址绑 loopback + 路由宣告、健康检查失败即撤路由、隐藏主 + 辅、DNS Cookie 密钥分阶段滚动、配置与过滤表版本化分发	②权威面隔离、③递归面隔离（冗余）	anycast 能藏住单节点故障、多台异地权威能扛住单机宕机；但案例里**故障恰恰是「撤路由」本身**。同一 anycast 集合若共用同一套配置分发与同一套健康判定，仍是同一故障域；且缓存不共享，切流量后有冷缓存抖动
第 27 章单播 DNS 与多播 DNS	`.local` 的链路本地多播（[RFC 6762]）、`home.arpa.`（[RFC 8375]）与单播名字的边界、Unbound 不参与 mDNS	①命名空间隔离（本链路自洽的一支）	本链路发现不依赖单播权威与公网路由，链路内设备仍可互相找到；但 mDNS 默认不出子网，跨子网要靠单播 DNS-SD / Discovery Proxy，**不能替代内部权威与递归面**

章	主要提供	对应隔离层	在这个场景下能挡住什么 / 挡不住什么
第 21 章综合实验拓扑	一台网关 + 一台云主机，A/B/C 三层实例、五个观测点、八项失败注入矩阵、`serve-expired` / 负缓存 / 逐层刷缓存的回滚纪律	③递归面隔离、⑤缓存与失败语义、⑥演练与可恢复性	注入矩阵已覆盖「停上游」「断出向」「瞬时 NXDOMAIN」「上游断开时过期缓存」；可直接扩展成下一节的撤路由/关认证项。但它是一台网关的教学拓扑，**没有独立的 OOB 链路与门禁系统**，第④层无法在拓扑内验证

28.4.1　仍然空着的那一格

把表竖着看，第④层「控制面隔离」只有接口、没有实现。**物理门禁的离线兜底、带外管理网的独立链路、工牌控制器的本地白名单、离线凭据的保管与轮换，都不在 DNS 书范围内**。第 21 章的拓扑能演示「DNS 挂了业务怎么降级」，演示不了「人进不去机房怎么修」。

这不是遗漏，是边界。本书能给的承诺只有一条：**别让 DNS 成为「进门」和「修 DNS」的前置依赖**。剩下的部分要在机房、身份与物理安全的方案里闭环；第 28 章把这一格明确留白，比假装六层都配齐了更诚实。

28.5　故障注入与验收清单

以下为**教学用示例地址与域名**，在「一台家庭网关 + 一台云主机」的实验环境里执行；`192.0.2.53` 代表内部递归器，`198.51.100.53` 代表对外权威视角，`203.0.113.53` 代表外部公共视角（教学桩），`2001:db8::53` 代表内部递归器的 IPv6 地址。实验主机名用 `gw-01`、`edge-01`、`cloud-01` 这类中性写法，不声明是真实环境。执行前先备份：`cp -a /etc/unbound /var/tmp/unbound-before-fault-lab/`。

28.5.1　可复制的清单表

#	注入动作	对应层	预期可观测结果	验收命令	回滚点
1	断开网关到云主机的业务链路，保留内网	⑥ / ③	内部名字仍可解析；外部名字靠过期缓存应答撑住，不出明文 53 兜底	`dig @192.0.2.53 svc.home.arpa. A +short` 有答案	恢复链路；逐层刷缓存
2	撤回对外权威的教学前缀（模拟 B1/B3）	② / ⑥	外部视角 SERVFAIL；**内部权威的 NS 与区数据不受影响**	`dig @203.0.113.53 www.example.net A +short` 无答案，同时 `dig @192.0.2.53 svc.home.arpa. A +short` 仍有答案	重新宣告前缀；确认内部权威路由未被动过
3	分别停掉对外权威、再停内部权威	②	两次故障的影响范围不同：前者不影响内部名字，后者影响	两次 `dig` 对比，记录差异	按「先内部权威，再对外权威」恢复
4	对一条内部名字瞬时返回 NXDOMAIN 后恢复	⑤	应答在 SOA minimum 期内仍是 NXDOMAIN；刷缓存后恢复	`dig @192.0.2.53 svc.home.arpa. A +noall +comment`	逐实例刷负缓存（第 7、22 章）

#	注入动作	对应层	预期可观测结果	验收命令	回滚点
5	断上游后连续查询同一名字 100 次	⑤	查询计数**不出现 100 倍放大**；有限流/退避或过期缓存应答命中	`unbound-control stats_noreset | grep -i servfail`	关闭注入脚本；核对计数基线
6	让中心认证/工牌教学桩不可达	④ / ⑥	门禁本地兜底放行应急账号；OOB 名字由静态解析给出，不查业务 DNS	`getent hosts oob.example.net` 有静态答案	恢复认证桩；核对本地白名单未越权
7	切断业务 DNS，只留 OOB 通道	④ / ⑥	工程师仍能登录控制台、取到离线凭据、按预案进场	`dig @2001:db8::53 gw-01.home.arpa. A +short` 在 OOB 侧仍成立	恢复业务 DNS；归档离线凭据使用记录
8	全流程恢复演练	⑥	严格按「控制面→DNS →业务」顺序，每步有验收命令与耗时记录	见下表命令组逐条复测	保留演练记录与配置备份，回滚到基线

表中第 6、7 项依赖机房/身份系统的教学桩，不在 DNS 实现范围内；这里只给验收条件。任何一项出现「预期结果不符」，先按第 21 章的纪律排查观测点与 nftables 规则，再怀疑上游。

28.5.2 一组教学用验证命令

```
# 1）命名空间隔离：内部保留域在公共视角不可解析
dig @203.0.113.53 home.arpa. SOA +noall +comment
dig @192.0.2.53  svc.home.arpa. A +short

# 2）权威面隔离：内部名字与对外名字指向不同的权威
dig @192.0.2.53     home.arpa. NS +short
dig @198.51.100.53  example.net NS +short

# 3）递归面隔离：根层由本地镜像给出，不依赖公网出向
dig @192.0.2.53 . NS +noall +answer
unbound-control status

# 4）缓存与失败语义：过期缓存应答、负缓存、SERVFAIL 计数
dig @192.0.2.53 www.example.org A +noall +answer +comment
unbound-control dump_cache | grep -i 'example.org' | head
unbound-control stats_noreset | grep -iE 'servfail|expired'

# 5）IPv6 与 ULA：内部递归器的文档地址段可达性
dig @2001:db8::53 svc.home.arpa. AAAA +short

# 6）控制面隔离：OOB 名字走静态解析，不经业务 DNS
getent hosts oob.example.net
grep -n 'oob.example.net' /etc/hosts

# 7）恢复顺序自检：控制面 → DNS → 业务，逐层确认再放行
dig @192.0.2.53 oob.example.net A +short
dig @192.0.2.53 svc.home.arpa. A +short
dig @192.0.2.53 www.example.org A +short
```

为什么这么设：七组命令各自只测一层，且都由近及远——先看内部名字、再看权威委派、再看根层、再看缓存语义、最后看控制面与恢复顺序。把注入前后的同一条命令输出留档，差分就是验收证据，不需要额外搭监控。

怎么验证：注入 #2 时，第 1 组里公共视角应无答案、内部视角仍有答案；第 2 组两组 NS 应指向不同集合；第 4 组的 `expired` 计数应上升而 `servfail` 计数不呈倍数增长。三处齐备，才说明命名空间、权威面、失败语义三层都真的切开了。

失败模式 / 回滚点：注入会污染缓存（尤其负缓存），测完必须按「控制面→权威→递归→客户端」逐层刷；`unbound-control` 不可用时只能重启实例刷缓存，且链路上每个实例都要刷（第 7、22 章）。所有注入前备份 `/etc/unbound`，所有路由/宣告类注入先在实验段做，回滚点是重新宣告教学前缀并恢复链路。演练记录与 `<TOKEN>` 类凭据的使用记录一并归档。

28.6　本章要点

- **归因必须摆正**：官方复盘里没有「内外同域名导致 split-horizon 误配」这一条；正确链条是骨干维护命令误切断数据中心互联→权威 DNS 按设计判定不健康并主动撤回 BGP 路由→权威 NS 从全球路由表消失→正常与带外访问皆断、DNS 失效打断内部工具→只能物理到场且高安全设计拖慢恢复。根因不是「同域名」，而是**内外共用同一套名字解析基础设施与同一故障域**。
- **推论不许写成事实**：「内部名字挂在可公开解析的同一命名空间且无独立内部权威/视图，会把内部可用性绑到公网路由与公网权威」是从案例提炼的**设计推论**；门禁/工牌失效属**公开报道**（NYT、The Verge）与官方「物理到场」互相印证，不是 Meta 逐条确认。证据分官方确认 / 外部观测 / 公开报道 / 本书推论四类。
- **放大与语义**：Cloudflare 观测约 30 倍重试、15:51 SERVFAIL / 15:58 路由撤回；ThousandEyes 观测 15:39 起数百条撤回、7 个覆盖内部 DNS 的 IPv4 前缀消失并指出控制面被切断；Verisign 观测约 7,000 → 900,000 qps（>100 倍）、TTL 1–5 分钟缓存几分钟耗尽，建议限流 + 指数退避。两组倍数是不同观测点，不可混用。
- 六层隔离：①命名空间（`home.arpa.` / 内部专用子域，split-horizon 只算工具且排障代价高）②权威面（分开部署/宣告/变更、隐藏主、至少两台异地）③递归面（独立递归器、本地根镜像 / `auth-zone`、`serve-expired`）④控制面（OOB、门禁与认证的本地校验与离线兜底、物理到场预案）⑤缓存与失败语义（TTL、负缓存、SERVFAIL 限流与退避）⑥演练与可恢复性（断网/撤路由/关权威/关认证，恢复顺序**先控制面再 DNS 再业务**）。
- **结论落在架构**：这是故障域划分问题，与选用哪个 DNS 软件无关，不写成产品批判；第 25 章给①②、第 26 章给②③的冗余（但 anycast 撤路由与共用配置分发仍可能是同一故障域）、第 27 章给①的链路本地一支、第 21 章给③⑤⑥，**第④层物理门禁离线兜底不在 DNS 书范围内**，只标注接口与验收条件。
- **验收靠注入不靠声明**：八项注入清单 + 七组教学命令，全部使用 RFC 5737 / `2001:db8::/32` 文档地址与 `example.net` / `example.org` / `home.arpa.`；注入必污染缓存，恢复后逐层刷，配置与凭据使用记录归档。

28.7　参考资料

- Meta 工程博客，《More details about the October 4 outage》（2021-10-05，本章 A1–A6 的唯一官方依据）：https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/
- Cloudflare 博客，《Understanding how Facebook disappeared from the Internet》（B1、B2）：https://blog.cloudflare.com/october-2021-facebook-outage/
- ThousandEyes，《Facebook Outage Analysis》（B3–B5；该站对部分自动抓取返回 403，按文章标题在其官方博客检索即可）：https://www.thousandeyes.com/blog/facebook-outage-analysis
- Verisign 博客，《Observations on Resolver Behavior During DNS Outages》（B6–B8；该文现位于 `blog.verisign.com/security/` 路径下）：https://blog.verisign.com/security/facebook-dns-outage/
- RFC Editor, [RFC 8375] *Special Use Domain 'home.arpa.'*（①命名空间隔离）：https://www.rfc-editor.org/rfc/rfc8375
- RFC Editor, [RFC 2308] *Negative Caching of DNS Queries (DNS NCACHE)*（⑤负缓存）：https://www.rfc-editor.org/rfc/rfc2308
- RFC Editor, [RFC 8767] *Serving Stale Data to Improve DNS Resiliency*（③⑤ `serve-expired` / 过期缓存应答）：https://www.rfc-editor.org/rfc/rfc8767
- NLnet Labs, Unbound 官方文档（`serve-expired` 等参数名称与默认值以此为准）：https://unbound.docs.nlnetlabs.nl/en/latest/

[RFC 5737]、[RFC 3849]、[RFC 2136]、[RFC 6762]、[RFC 1034]、[RFC 1035] 与章内提到的 NIST SP 800-81r3 控制目标、The Verge 与 NYT 公开报道的具体篇目，一律按名称在 RFC Editor / 出版方站点检索，完整链接由第四篇统一 RFC 索引章收录；`serve-expired-client-timeout` 之外的限流参数、门禁控制器的离线兜底实现、Meta 内部认证与 DNS 的依赖链，以官方资料为准。C1/C2 属公开报道、D1–D4 是归纳推论，不是官方结论。

第二十九章

DNS 作为攻击面：隧道、扫描与资产暴露

把 DNS 只当成「把名字换成地址」的服务，会漏掉它的另一面：它是几乎所有网络都默认放行的一条双向通道，也是一份公开可读、长期缓存、跨组织消费的资产清单。前者让 DNS 成为数据外泄与命令控制（command and control，C2）的载体，让递归解析器成为可被滥用的公共资源；后者让一条忘记删除的记录变成别人的入口。这两件事共用同一套观测手段，所以放在一章里讲。

29.1 DNS 隧道：把数据编码进查询名

29.1.1 一次查询就是一封信

隧道的核心不是「偷偷发一个包」，而是把数据塞进本来就要发出去的查询名（QNAME）里。被控主机把待传数据切成块、做 base32/base64url 一类的编码，当作最左侧的子域标签拼上去：

```
[被控主机 192.0.2.20]
   |  A?  < 编码块 >.exfil.example.net
   v
[内部递归解析器 gw-01] -- 出向 53 --> [互联网] --> [攻击者权威服务器 ns.exfil.example.net]
   ^                                                          |
   |  TXT / NULL / CNAME 应答 <----------------- 下行指令 --+
```

由于 `exfil.example.net` 的权威服务器由攻击者控制，每一个查询都会把最左标签写进他的权威日志里——**即使应答是 NXDOMAIN，上行也已完成**。下行则靠 TXT、NULL、CNAME 的 RDATA 携带确认、分片序号与指令，于是形成双向 C2。公开工具形态包括 iodine、dnscat2、dns2tcp 一类，区别主要在编码方式、承载 QTYPE 与会话管理，不在原理。

单标签上限 63 字节、完整 QNAME 约 253 字节，所以每次查询只能带走几十字节；但几 KB/s 到几十 KB/s 的低吞吐已经足够偷凭据、配置文件与少量数据库记录。低速恰恰是它的优势：不容易触发带宽告警，也不像大流量外传那样显眼。

29.1.2 为什么常规防线看不见

- **出站策略通常放行 UDP/53**。很多网络允许任意主机直连公网解析器，防火墙看不到「这是 DNS 还是隧道」。
- **DNS 日志常被忽略**。默认 `verbosity: 1` 且 `log-queries: no` 时（第 9 章基线），查询名根本不落盘；只留 SERVFAIL 日志等于只记失败、不记成功的外泄。
- **域名会轮换**。DGA（domain generation algorithm）或预置域池按时间/种子换域，让「封掉几个域名」在小时级就失效。
- **承载可以换**。明文 53 被收敛后，客户端仍可能改用加密传输端点绕过，所以出向策略要同时覆盖明文 53 与加密传输发现机制（第 12、13 章）。这里要守住全书的关键区分：DNSSEC 提供数据完整性与来源认证，DoT / DoH / DoQ 提供传输加密与隐私，二者互不替代。

29.1.3 Payload 分析：看一条查询长什么样

Payload 分析针对**单条查询**的内容特征，对已知隧道工具最有效：

- 超长 QNAME（接近 253 字节上限）、单标签接近 63 字节；

- 高熵 / 随机字符串、异常字符分布（大小写混排的 base32、连续的十六进制串）；
- 罕见 QTYPE：TXT、NULL、CNAME 在普通客户端查询里占比很低；
- 编码模式正则，以及同一域下标签长度高度一致（分片定长的痕迹）。

局限也很明确：换一种编码或把块切短就绕过了；而且 ACME DNS-01 的长 token、CDN 健康检查、服务发现类长名字都会产生高熵长标签。第 31 章会讲告警误报治理与白名单，这里先记住**特征命中只触发人工复核，不自动阻断**。

29.1.4　流量分析：看一批查询呈什么形状

流量分析针对**一段时间内的聚合行为**，能覆盖未知隧道：

- **单域唯一 FQDN 数**：最通用的信号。隧道必须不断换名字来传新数据、并避开缓存命中，因此「同一个注册域下出现了大量互不重复的名字」几乎与编码方式、QTYPE、长度无关。
- qps 突增，尤其是单客户端对某个新域的持续查询；
- NXDOMAIN 比例升高（编造子域 / DGA 的副产物）；
- 低 TTL（便于复用名字、避免被长时间缓存）；
- 单源集中查询某新域、地理异常（客户端位置与该域权威侧历史分布不符，需要 IP 归属数据支撑）。

GIAC / SANS 一类资料把它归成 payload 分析与流量分析两类；两类检测要一起用：payload 分析给出「这条像不像隧道」的可解释证据，流量分析给出「这个域该不该进复核队列」的排序依据。公开学术工作（Ben-Gurion University 关于低吞吐 DNS 外泄检测）走的是更长的观察窗口 + 按域聚合特征 + Isolation Forest 一类异常检测；具体实现与数据集见论文原文，本书不据此宣称任何检测率。

29.1.5　可复制的 dnstap / jq 巡检骨架

dnstap 是自己的查询流副本，落的是二进制帧。下面的骨架假设 consumer 已把帧转成 JSON Lines，每行含 `ts` / `type` / `client_address` / `qname` / `qtype` / `rcode`；**字段名随 consumer 实现而变，先跑 `head -1 /var/log/dnstap/queries.jsonl | jq 'keys'` 对齐**。全量查询日志本身就是隐私数据，保留期与访问范围按第 7、31 章的边界执行，不要无条件推给第三方 SaaS。

```
#!/bin/sh
# dns-tunnel-triage.sh ——教学用巡检骨架，不是即插即用的检测产品
LOG=${1:-/var/log/dnstap/queries.jsonl}
WINDOW=${2:-3600}                          # 观察窗口（秒）
SINCE=$(date -u -d "-${WINDOW} seconds" +%s)

# ① 按「客户端 + 注册域」聚合：唯一 FQDN 数 / qps / NXDOMAIN / 罕见 QTYPE / 长 QNAME
#   输出首列是纯数字的排序键，后面才是带标签的指标列
jq -r --argjson since "$SINCE" '
    select(.type == "CLIENT_QUERY" and (.ts >= $since))
    | [(.client_address // "unknown"),
       ((.qname // "") | ascii_downcase | sub("\\.$"; "")),
       (.qtype // ""), (.rcode // "")] | @tsv' "$LOG" \
| awk -F'\t' '
    function reg2(name,   n, a) {
      n = split(name, a, ".");
      return (n >= 2) ? a[n-1] "." a[n] : name;
    }
    {
      client = $1; fqdn = $2; qtype = $3; rcode = $4;
      key = client "\t" reg2(fqdn);
      uniq[key "\t" fqdn] = 1;             # 去重集合，不是查询计数
      q[key]++;
      if (rcode == "NXDOMAIN") nx[key]++;
      if (qtype == "TXT" || qtype == "NULL" || qtype == "CNAME") rare[key]++;
      if (length(fqdn) > 120) longq[key]++;
    }
    END {
      for (u in uniq) { split(u, b, "\t"); cnt[b[1] "\t" b[2]]++ }
      for (k in q)
        printf "%d\t%s\tqueries=%d\tnxdomain=%d\trare_qtype=%d\tlong_qname=%d\n", \
          cnt[k], k, q[k], nx[k], rare[k], longq[k];
```

```
    }' \
| sort -k1,1nr | head -30          # 不加 -t: $'\t' 是 bash 专有写法，#!/bin/sh 下会报错

# ② payload 钻取：把最长的 QNAME 打出来人工看编码形态
jq -r --argjson since "$SINCE" '
    select(.type == "CLIENT_QUERY" and (.ts >= $since))
    | select((.qname | length) > 120)
    | [.client_address, .qname, .qtype] | @tsv' "$LOG" \
| awk -F'\t' '{ n = split($2, a, "."); print length($2), n, $3, $1, $2 }' \
| sort -k1,1nr | head -20
```

为什么这么设：首列是纯数字的 `uniq_fqdn` 值，专门给 `sort -n` 用；把它写成 `uniq_fqdn=NNN` 这种带前后缀的字符串会让数值排序失效。`uniq` 用「客户端 + 注册域 + FQDN」做去重后再计数，正是流量分析里最抗规避的那一项；`q` 同时给出体量，避免把「一次性批量导入」误读成持续外泄。`rare` / `longq` 只是把 payload 信号并到同一行，方便一次排序决定先看谁。`reg2` 取最后两个标签只是教学近似，正式系统必须换成公共后缀表（Public Suffix List），否则 `co.uk` 一类后缀会把大量无关域并成一行。

怎么验证：`bash dns-tunnel-triage.sh /var/log/dnstap/queries.jsonl 3600 | head -5` 应能列出你网络里的正常域；再向自己的解析器发一批教学用高熵查询制造已知信号——把 `exfil.example.net` 换成你有授权的实验子域后执行 `for i in $(seq 1 50); do dig @192.0.2.1 "$(openssl rand -hex 24).exfil.example.net" A +short >/dev/null; done`，重跑脚本时该域的 `uniq_fqdn` 应上升约 50。骨架只读日志、不改配置，验证过程不会产生阻断动作。

失败模式 / 回滚点：字段不对齐会输出空表，先查 `jq 'keys'`；consumer 断流时表是陈旧的，别把「没告警」当「没外泄」，dnstap 断流本身要在第 31 章单独告警。最大误报源是证书续期与 CDN 健康探测，所以脚本只产出排序、不自动写 RPZ。骨架无状态，停跑即回滚；若后续人工加了 RPZ 条目，必须同时登记复核日期，否则过期 IOC 会长期误伤。

29.1.6 检测阈值表

所有数字都是**触发人工复核的起点值，不是阻断阈值**，且**需按你自己 7 天同时段基线调整**；不同网络的客户端数量、CDN 用量差异极大，照抄会淹没在误报里。

信号	计算口径	起始观察值（需按基线调整）	常见误报源
单域唯一 FQDN 数	每客户端、每注册域、每小时去重后的 FQDN 数	大于 200 或超过基线 P99	CDN 多级 CNAME 探测、大规模服务发现
QNAME 长度	长度大于 120 字节的查询占比	大于 1%	ACME DNS-01 token、长 SRV / SVCB 名字
单标签长度	最左标签大于等于 50 字节（上限 63）	出现即进复核队列	一次性验证值、遥测上报名字
罕见 QTYPE 占比	TXT、NULL、CNAME 合计占该域查询比例	大于 5%	邮件策略查询、SPF / DKIM 查询
qps 倍率	相对 7 天同时段基线的倍数	大于 3 倍且持续 5 分钟	发布、批处理、缓存冷启动
NXDOMAIN 比例	该域 NXDOMAIN 数 / 该域查询数	大于 30%	拼写纠错、已下线业务、search 域追加
低 TTL	应答 TTL 小于 60 秒的占比	明显偏离基线	灰度发布、故障切换期
地理异常	客户端归属与历史分布的偏离	出现新地区即复核	出差、VPN 出口变化

高熵与编码正则属于 payload 分析的第二层，骨架只给了长度与字符形态的近似；正式系统再补香农熵与编码正则，并把白名单放在**聚合之后**（第 31 章）。

29.1.7 缓解：收敛出向 53，而不是封几个域名

缓解的顺序是「先关掉通道，再看得见，最后处置已确认的 IOC」：

1. **客户端不能直连公网 53**，出向只允许内部解析器；内部解析器只向受控上游发查询（第 5 章单上游 + fail-closed，第 9 章已插入的出向收敛节）。本机解析器进程自己的出向 53 要按 uid 豁免，否则会被自己的劫持规则吃掉（第 14 章）。

2. **把加密传输端点纳入同一策略**：压制 DDR 自动发现、管住 DoH 端点，否则客户端换一条承载就绕过去了（第 11–13 章）。
3. **dnstap 全量日志 + 明确保留期**（第 7 章的 30 天），这是上面所有检测的来源；没有查询名，payload 与唯一 FQDN 都无从计算。
4. **RPZ 只用于阻断已确认的域**（官方 RPZ 用法见参考资料），配合 `ratelimit` 一类限速（第 9 章）。RPZ 是事后处置，不是控制措施。
5. **不要写「封几个域名」当结论**：DGA 会轮换，新域注册成本极低，黑名单永远慢一步。真正的控制面是「谁能向哪里发起 DNS 解析」；域名清单只是这条控制面上的临时补丁。

29.2　高速扫描与开放解析器：不提供可被滥用的能力

29.2.1　Unbound 能不能当「污染源」或红队工具

先给结论：**Unbound 不是投毒工具，也不会替你把伪造应答注入别人的解析器**。它是递归验证器，设计前提恰恰是「只接受能按 DNSSEC 签名链、或按自身清理规则核验的数据」（第 1 章的 2008 年案例、第 10 章的三态验证）。把一台 Unbound 配成向第三方解析器**主动发送伪造应答**，既不属它的功能，也不是这里会写的做法——那需要自己构造 DNS 报文并按对端在途查询的 TXID/端口抢答，属于 [RFC 5452] 讨论的伪造应答范畴，与「配置一台解析器」是两回事。

但有三条**机理上的边界**值得防守方知道，它们决定了「自己手里的 Unbound 会被怎么用」：

能力	Unbound 是否具备	防守含义
改写**自己**应答给客户端（`local-zone` / `local-data` / RPZ / `respip`）	具备	这是正常的本地策略。一旦该实例被未授权的人控制、或错误地对全网开放递归，策略就能把**信任它的客户端**导向任意地址——所以第 9 章把 `access-control` 默认拒绝放在第一位。
作为**开放解析器**被第三方当反射器 / 扫描器	配错时具备	不是「投毒」，而是放大与免费算力滥用（见下节 [RFC 5358]）。
向**别人的**缓存注入伪造应答	不具备	缓存投毒依赖伪造源、抢在真应答前到达、猜中 TXID/端口；Unbound 没有这个功能面。防守侧靠源端口随机化、0x20、`harden-glue` 与 DNSSEC 验证（第 1、10 章）。

授权的红队测试里，Unbound 更常见的角色反而是**观测点**：用 dnstap 记录自己网络里的查询流、验证 DNS 隧道与随机子域信号是否可见（第 31、34 章），或在自建实验拓扑里做一台**受控的**权威/递归来验证防守规则是否生效。这些都不触碰第三方资产。本书不提供任何针对外部解析器的伪造、抢答或规避检测流程；涉及应答改写的章节（第 24 章过滤表、第 9 章 `respip`）一律限定在「自己持有、自己授权的解析器」上。

29.2.2　一套工具，两种用法

massdns、dnsx、subfinder 一类工具原本用于资产发现：拿一份字典高速枚举子域、批量查 ANY 或随机子域、从应答里挑出存活名字。同一套能力换一个发起者，就是对一个域的高速探测，或把一批解析器当成免费扫描器用。这里讨论防护时，重点不是识别工具指纹，而是**让本地服务不具备被滥用的能力**。

29.2.3　[RFC 5358] 与开放反射器

开放递归解析器是放大攻击与扫描的公共资源。[RFC 5358] 明确反对开放反射器：UDP 源地址可伪造，几十字节的小查询能换回上千字节的大应答，攻击者把源伪造成受害者即可把流量打过去。第 9 章已算过放大倍数并给了 `access-control` / 报文尺寸 / 限速的完整基线，本章只补一层视角：开放解析器的问题不只是「自己被打」，更是「自己被打的同时成为打别人的枪」，因此它属于对外责任，不能只按本地可用性评估。

29.2.4　限速与退避

- `access-control` **默认拒绝**，无匹配即 `refuse`，只给已登记的客户端网段放行（第 9 章）；

- `ratelimit` / `ip-ratelimit` 官方标注为 **experimental**，手册说明前者按目标域回 SERVFAIL、后者按源静默丢弃；**按 `stats_noreset` 的基线调，不要把任何数字当通用结论**。这里不重复第 9 章的配置块。
- **权威侧 RRL**（response rate limiting）：权威服务器对同一查询/源组合的应答限速，压住反射放大与随机子域洪泛。
- **递归侧对失败应答限速与指数退避**：SERVFAIL / 超时引来的重试会自我放大，Verisign 的公开观测里出现过百倍量级的重试放大；负缓存（[RFC 2308]）既是抑制手段也是故障放大器（第 4、22 章）。工程上要让重复失败退避、聚合，而不是无限重试。
- **禁止 AXFR 给未授权源**：AXFR 走 TCP/53，一次就能拿走整区；只在已登记的从服务器之间允许，并单独验证 TCP 而不是只测 UDP。
- **日志里识别单源高速**：按「源地址 + 秒」聚合查询数，和上面的隧道骨架共用同一份 JSON Lines。

```
# 单源高速：按「源地址 + 秒」统计，找持续的高 qps 源
LOG=/var/log/dnstap/queries.jsonl
SINCE=$(date -u -d "-3600 seconds" +%s)
jq -r --argjson since "$SINCE" '
    select(.type == "CLIENT_QUERY" and (.ts >= $since))
    | [(.ts | tostring), (.client_address // "unknown")] | @tsv' "$LOG" \
| awk -F'\t' '{ c[$2 "\t" int($1)]++ }
    END { for (k in c) print k "\t" c[k] }' \
| sort -k3,3nr | head -20
```

为什么这么设：按秒分桶能区分「持续高速」与「一次性突发」，后者常见于发布或缓存冷启动；只按总量排序会把一次性大客户排到最前面，掩盖真正的扫描源。

怎么验证：`unbound-control stats_noreset | grep -iE 'queries|ratelimit|unwanted|discard'` 对照限速相关计数是否随压测上升（计数器名以官方手册为准，不要照抄本行的关键字当结论）；从自家授权网段用 `dig` 造一小批查询，确认脚本能看到该源且限速计数变化。

失败模式 / 回滚点：反射场景里看到的「源」可能是被伪造成的受害者，据此封禁等于帮攻击者完成二次伤害；分布式扫描又会摊薄单源计数，所以这一项只作为复核线索。脚本只读日志，停跑即回滚。

29.2.5 为什么不是「封 IP 就行」

源可以是分布式的（僵尸网络、多出口），UDP 源还可以伪造；封 IP 是 denylist，既不证明服务能力已收紧，也挡不住换一批源再来（第 13 章讨论加密解析器发现时给过同一条判据：封 IP 是 denylist，不是证明）。临时止血可以封，但整改要落在四件事上：**默认拒绝递归、限速与退避、压小应答、收敛出向路径**，让这台机器对未授权源「无服务可滥用」。长期告警交给第 31 章的观测栈，而不是靠人工维护一张 IP 表。

29.3 互联网暴露面排查：只查自己有授权的名字

29.3.1 数据来源

来源	能看到什么	局限
Certificate Transparency 日志	新签发证书里的 SAN，含通配符名、早已下线的名字	只覆盖申请过证书的名字，不覆盖纯内部或未签发证书的子域
被动 DNS	历史上被解析过的名字与 RDATA、首次/末次见到的时间	覆盖取决于采集点位置，看不到未被采集的查询
公开区数据、搜索引擎索引	区文件、页面与链接里出现的主机名	陈旧，且「搜不到」不等于「不存在」
WHOIS / RDAP	注册人、注册商、NS、域状态	描述注册关系，不描述当前解析结果
自己的权威区文件与变更单	设计上应该有哪些记录、谁建的	常与实际运行的 DNS 不一致

把这张表读成一句话：**以运行中的区为准**。CT 与被动 DNS 负责找出「你以为没有、其实有」的名字，权威区与变更单负责解释「它为什么在」。

29.3.2 自查清单

检查项	命令形态	期望结果	异常含义
NS 委派一致性	`dig +nssearch`	父区委派与区自身 NS 一致	存在旧 NS、未清理的委派
AXFR 是否误开	`dig AXFR ... @< 自己权威 >`	未授权源被拒绝	整区可被未授权源取走
CNAME / NS / MX / TXT 目标	`dig +short <name> <type>`	目标仍在服务、仍属自己	指向已下线资源（下一节）
_acme-challenge 残留	`dig +short _acme-challenge... TXT`	无残留或仅当前续期中的值	验证记录与旧自动化凭据长期存在
DS 是否匹配	`dig +short <zone> DS @< 父区权威 >`	与当前 KSK 对应	整个区被判 bogus（第 10 章）

```
# 执行前把 ZONE / AUTH 换成你自己已授权的区与权威服务器
ZONE=example.org
AUTH=192.0.2.53    # [RFC 5737] 文档地址，勿直接用于生产

dig +short NS "$ZONE"
dig +nssearch "$ZONE"
dig +short "_acme-challenge.$ZONE" TXT
dig +short "svc.$ZONE" CNAME
dig +short "$ZONE" DS @"$AUTH"
dig AXFR "$ZONE" @"$AUTH" +time=2 +tries=1 | head -30    # 只对自己有授权的区域执行
```

最后一条的期望是**未授权源拿到拒绝**（transfer failed / REFUSED），只有已登记的从服务器才应看到区内容；把 UDP 与 TCP/53 都测一遍，很多「AXFR 已关」的结论只测了 UDP。DS 一项只做匹配核对，信任锚与验证状态见第 10 章。

失败模式：`dig +short` 输出为空有多种含义（记录不存在、名字不存在、应答被过滤），不能据此判定「资源已下线」；补一次不带 `+short` 的查询看 HEADER 的 rcode 与 AUTHORITY 段。排查结果要落成带日期、执行人的工单，`<TOKEN>` 一类凭据不写进去。

29.3.3　纪律：只枚举有授权的名字，不 claim

OWASP Web Security Testing Guide（WSTG）的子域接管测试项明确要求 **“Do not claim it”**（原文措辞见参考资料里的官方项目页）。这条纪律在本章有两个具体落点：

- **范围**：只读的 CT / 被动 DNS / WHOIS-RDAP 可以自由查；主动 `dig`、AXFR、`+nssearch` 只对自己有授权的名字与自己的权威服务器执行；不对第三方权威服务器做 AXFR 或高速枚举。
- **发现可接管资源时**：只确认指纹（错误页特征、CNAME 目标是否 NXDOMAIN），在自己的资产清单里登记，然后**删掉自己的悬空记录**。不要去服务商那里注册同名资源「验证一下」，不要往该名字写内容、发请求或申请证书——那已经越过授权边界，且可能破坏他人正在使用的服务。

29.4　悬空 CNAME 与子域接管

29.4.1　成因：DNS 记录是公开指针，不是所有权证明

CNAME 常指向第三方资源：对象存储桶、Pages 站点、SaaS 自定义域、CDN 自定义域。服务下线或资源被释放后，DNS 记录留在区里，就成了「悬空 CNAME」（dangling CNAME）。若服务商在新建同名资源时不校验域所有权，任何人注册该资源名，就能在这个子域上提供内容。

要点是**不是所有「查不到」的 CNAME 都可接管**：有的服务商会校验所有权，有的名字已被回收但保留错误页，有的只是暂时解析失败。区分「可接管」与「仅下线」，靠的是指纹而不是猜测。

29.4.2　影响分级

被接管的记录	接管后能做什么	相对危级
NS	控制整区：改任意记录、签证书、改 MX 收信	最高
MX	接收或观察该域邮件	高
CNAME / A	在该子域挂内容、钓鱼、读父域作用域的 cookie	高
TXT（SPF include、域验证）	借用的发件授权、长期残留的验证记录	中

被接管的记录	接管后能做什么	相对危级

父域设了 `Domain=example.org` 的 cookie 会发给所有子域，所以一个被接管的子域能读到会话 cookie；钓鱼页面还额外获得「看起来是自家域名」的 URL 与合法证书。若父域 SPF `include:` 了被接管域，发件授权也会被借用——记录里放了什么、谁消费，第 30 章展开，本章只看「指针被别人拿走」这一层。NS 接管最高危，因为它把**整份资产清单**的控制权交出去，而不是只交出一个子域。

29.4.3 下线服务先删 DNS

顺序反过来就会留窗口：先释放第三方资源名、后删 DNS，中间那段时间记录指着一个人人可注册的名字。正确顺序是：

1. 删掉或改掉 DNS 记录（先处理 CNAME，再处理相关的 TXT / MX / NS）；
2. 等 TTL 与链路上各层缓存过期——负缓存也会缓存 NXDOMAIN，改完仍可能短时间查不到（第 4、22 章）；
3. 再释放第三方资源名、关停实例。

把「删 DNS 已完成」写成下线变更单里的**必选勾选项**，没有这一项不允许关单。靠人记是这类问题反复出现的主要原因：DNS 记录没有「服务已下线」的自动感知，它只会一直指下去。

29.4.4 指纹库、资产清单与记录生命周期

- **对照公开指纹库**：`can-i-take-over-xyz` 一类项目按服务商归集错误页文案、CNAME 目标形态与可接管判据。指纹会随服务商改版变化，要定期复核，不能一次扫描定终身；本章不复制任何指纹条目。
- **资产清单字段**：FQDN、记录类型、RDATA、第三方服务与资源 ID、负责人、创建日期、复核日期、关联变更单。少了「第三方资源 ID」，就无法判断一条 CNAME 是否还指向自己的资源。
- **生命周期绑定**：记录与资源的创建、变更、下线走同一张工单；CNAME 目标 `dig +short` 是否仍存在、目标是否已被释放，纳入定期扫描；告警落到第 31 章的栈里。
- **缩小耦合面**：内部服务尽量用 `home.arpa.` 一类内网域与内部记录，少把内部名字挂到外部第三方资源上。

29.4.5 与第 13、28 章的两个接口

第 13 章那个**违反 [RFC 8020] 的 CDN CNAME 区**案例，本章只取一层含义：CNAME 链可以很长，中间层父名返回 NXDOMAIN 而子名却存在，且链上多跳完全由第三方控制。QMIN strict 的参数、整站 CSS/JS 解析失败与逐层刷缓存的步骤都在第 13 章，这里不重述；本章要补的是——**链上每一跳都是需要盘点、需要监控的资产**，任何一跳被释放都可能造成悬空，链越长越容易漏掉中间那一跳。

第 28 章故障域隔离的结论在这里有了安全侧的对应：内部可用性不该绑到外部 CNAME 链。接管其实是同一枚硬币的反面——别人不仅能改你的内容，也能改你的可用性。把权威面、递归面、控制面切开之后，还要把「外部第三方资源」作为一类独立故障域纳入资产清单。完整的负缓存放大故障在第 22 章，本章只引用不展开。

29.5 历史漏洞与 Unbound 的加固模式

29.5.1 2008 Kaminsky 缓存投毒

2008 年的 Kaminsky 缓存投毒事件把两处熵不足叠到了一起：DNS 事务 ID 只有 16 位，源端口在很多实现里可预测。攻击者只要在真应答返回前猜中这对值并伪造一个带胶水记录的应答，就能把攻击者控制的权威 NS 写进递归解析器的缓存，此后该域的查询全部被引走。行业的加固方向是三条：随机化源端口（扩大可用熵）、0x20 大小写随机化（把查询名的大小写当额外校验位，回显不一致即丢弃）、以及加速 DNSSEC 部署。

[RFC 5452]（*Measures for Making DNS More Resilient against Forged Answers*）把这些伪造韧性建议系统化了；Unbound 从设计上就按这一族建议实现，而不是事后逐个打补丁。要守住两条边界：0x20 与随机源端口只是**提高伪造难度**，不是加密；DNSSEC 提供的是数据完整性与来源认证，DoT / DoH / DoQ 提供的是传输加密与隐私，三者不在同一层。域外夹带的胶水记录是经典投毒面，对应 `harden-glue` 一类默认加固（第 9 章），它与是否启用验证器无关。

29.5.2 四个公开名称：只说机理

下表的名称与机理都来自公开资料；**具体影响版本、公告编号与年份见官方资料**（NLnet Labs 安全公告与对应版本发布说明），本章不列 CVE 编号、年份与版本号，也不把任何版本写成「已修复」。升级判断以你所用发行版的安全公告为准。

名称	公开可核验机理	对应的加固方向
NXNSAttack	恶意委派应答里塞入大量 NS 名，诱使递归器把一次客户端请求放大成海量出向查询	对单个请求可产生的出向查询数设限
DNSBomb	把一批查询的超时/慢应答在时间上对齐，汇聚成脉冲式流量	限速、超时分散，避免一批超时同时到期
NRDelegation	恶意委派让解析器反复迭代、反复回到同一批名字，耗尽解析预算	迭代次数与 referral 限额
ReTrap	NLnet Labs 安全公告里提到的后续研究，利用 DNSSEC 验证路径的算法复杂度形成 degradation of service	限额 + 挂起让出 CPU，避免长计算占死线程

Debian 的 unbound changelog 里出现过 “The DNSBomb attack, via specially timed DNS queries” 这样的措辞，可作为名称可核验的一处线索；除此之外本章不对时间线做断言。

KeyTrap 与 NSEC3 那一组 DNSSEC 计算放大问题在第 10 章结合验证器配置讲，这里只作一句交叉引用：它同样属于「用有限的计算预算对抗无限的输入组合」这一族问题。

29.5.3　Unbound 的加固模式

下面六条是**本书对公开设计说明与版本公告的归纳**，不是官方原话，也不是「Unbound 免疫这些攻击」的承诺：

1. **限额**：把「一次解析能消耗多少」变成有界量——出向查询数、迭代次数、referral 数、挂起请求数、`ratelimit`。NXNSAttack、NRDelegation 之所以能被压住，靠的就是这类上界。
2. **挂起让出 CPU**：解析状态被挂起等待应答，而不是占着线程阻塞；单个慢查询不拖死整个实例，也给超时分散留了余地。
3. **分离线程**：按角色/模块拆开，验证、递归、控制面的资源与故障不互相拖累（呼应第 7 章多实例、第 28 章故障域隔离）。
4. **fail-closed**：上游或加密通道失败时明确 SERVFAIL，不静默回退明文（第 5、14 章）。
5. **默认安全**：不对公网开放递归、DNSSEC 可验、`harden-*` 一族默认打开；运营者要显式做错才会变弱。
6. **小代码面便于审计**：功能集中在递归与验证，不把权威服务、过滤表、HTTP 栈都塞进同一进程；新的攻击族出现时，改动面与复查面都小。

把这六条合起来看，结论不是「逐个给攻击起名字、逐个加黑名单」，而是：上面所有攻击争抢的都是同一份有限资源——CPU 时间、出向查询额度、缓存与线程。**加固的方向是给这份资源设上界、并让超限与失败可观测**；检测与告警的长期工程在第 31 章。

29.6　攻击→观测信号→加固动作→本书章节

攻击 / 风险	观测信号	加固动作	本书章节
DNS 隧道数据外泄	单域唯一 FQDN 数突增、长 QNAME、罕见 QTYPE	出向 53 只准内部解析器、dnstap 全量日志、限速	第 7、9、14、29 章
DGA 轮换域名的 C2	新域集中查询、NXDOMAIN 比例升高、低 TTL	按行为聚合复核，RPZ 只处置已确认域，不靠单域封禁	第 29、31 章
开放反射器放大	未授权源递归成功、大应答 / 小查询比异常	`access-control` 默认拒绝、压小应答、权威侧 RRL	第 9、29 章
massdns 类高速枚举	单源或分布式 qps 突增、随机子域与 ANY 集中	`ratelimit` / `ip-ratelimit`（experimental，按基线调）、失败应答退避	第 9、29、31 章

攻击 / 风险	观测信号	加固动作	本书章节
AXFR 误开	未授权源在 TCP/53 取得整区内容	只向已登记从服务器允许区传送，UDP 与 TCP 分别验证	第 9、29 章
悬空 CNAME 接管	CNAME 目标 NXDOMAIN 或出现服务商「未认领」指纹	下线服务先删 DNS、资产清单与记录生命周期绑定	第 13、28、29、30 章
NS / MX / TXT 接管	委派 NS 不一致、邮件记录指向已释放资源	`dig +nssearch` 一致性核对、清理残留验证记录	第 28、29、30 章
缓存投毒（Kaminsky 类）	非本实例发出的应答、异常胶水记录	随机源端口、0x20、DNSSEC 验证（数据完整性）	第 9、10、16、29 章
NXNSAttack / DNSBomb / NRDelegation / ReTrap	单请求出向查询数、脉冲 qps、迭代数、CPU 时间异常	限额、超时分散、挂起让出 CPU、按官方公告升级	第 29、31 章
KeyTrap / NSEC3 计算放大	DNSSEC 验证路径 CPU 时间异常	版本升级与验证器配置（详见该章）	第 10 章

表中「观测信号」一列刻意只写能在查询流或统计里看到的量：dnstap 的查询名与 QTYPE、`stats_noreset` 的计数、权威日志的应答记录。看不到就谈不上告警，也谈不上加固验收。

29.7 与后续章节的分工

本章站在**防守侧**：我的暴露面怎么自查、怎么加固。同样的工具与信号换一个视角看，就是第 34 章的「对手按什么顺序找你、每一步留下什么痕迹」，以及第 35 章的「怎么用蜜罐把这些痕迹变成高质量观测数据」。三章共享同一套事实，但落点分别是**加固 / 侦察链 / 反向观测**，不重复阈值表与脚本。

29.8 本章要点

- DNS 隧道把数据编码进子域标签，靠攻击者控制的权威服务器收上行、靠 TXT / NULL / CNAME 回下行；payload 分析看单条查询（长 QNAME、高熵、罕见 QTYPE），流量分析看聚合行为，其中**单域唯一 FQDN 数**最通用、最抗规避。所有阈值都是需按 7 天基线调整的复核起点，不是阻断线。
- 缓解的顺序是「出向 53 只准内部解析器→ dnstap 全量查询日志（含保留期）→限速→ RPZ 处置已确认域」，**不是封几个域名**：DGA 会轮换，域名黑名单只是事后 IOC。出向收敛配置见第 9、14 章。
- 开放解析器与高速扫描的整改目标是**不提供可被滥用的服务能力**：默认拒绝递归、experimental 的 `ratelimit` / `ip-ratelimit` 按 `stats_noreset` 基线调、权威侧 RRL、失败应答退避、禁 AXFR 给未授权源。源可分布式、可伪造，所以「封 IP 就行」是 denylist 式的止血而非证明。
- 暴露面排查以运行中的权威区为准，CT / 被动 DNS / WHOIS-RDAP 只负责找出「以为没有、其实有」的名字；纪律是**只枚举有授权的名字、发现可接管资源不要 claim**（OWASP WSTG 明确要求），只确认指纹并删除自己的悬空记录。
- 悬空 CNAME 的成因是「DNS 记录是公开指针，不是所有权证明」；NS 接管最高危，cookie 作用域、钓鱼与邮件次之。下线服务要先删 DNS 并等缓存过期，再释放第三方资源；第 13 章的长 CNAME 链案例与第 28 章的故障域隔离说明了链上每一跳都要盘点。
- 从 2008 Kaminsky 缓存投毒到 [RFC 5452] 伪造韧性，再到 NXNSAttack / DNSBomb / NRDelegation / ReTrap，攻击争抢的都是有限的计算与出向预算；Unbound 的加固模式是限额、挂起让出 CPU、分离线程、fail-closed、默认安全、小代码面便于审计，而不是堆功能或堆黑名单。

29.9 参考资料

- RFC 5358, *Preventing Use of Recursive Nameservers in Reflector Attacks*: https:// www.rfc-editor.org/rfc/rfc5358
- RFC 5452, *Measures for Making DNS More Resilient against Forged Answers*（伪造应答的机理与缓解）: https://www.rfc-editor.org/rfc/rfc5452

- CERT/CC, *VU#800113 —Multiple DNS implementations vulnerable to cache poisoning*（Kaminsky 类投毒的公开披露）：https://www.kb.cert.org/vuls/id/800113
- RFC 5737, *IPv4 Address Blocks Reserved for Documentation*（本章示例地址段）：https://www.rfc-editor.org/rfc/rfc5737
- Unbound 官方手册, `unbound.conf(5)`（`access-control`、`ratelimit` / `ip-ratelimit`、RPZ 与 hardened 相关选项及默认值）：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html
- Unbound 官方手册,Response Policy Zones 专题(已确认域的阻断用法):https://unbound.docs.nlnetlabs.nl/en/latest/topics/filtering/rpz.html
- dnstap 项目主页（查询流结构化日志格式与工具生态）：https://dnstap.info/
- OWASP Web Security Testing Guide 官方项目页（子域接管测试项与“Do not claim it”纪律）：https://owasp.org/www-project-web-security-testing-guide/
- NLnet Labs, Unbound 文档入口（安全公告、版本发布说明与历史资料索引；具体 CVE 与年份见官方资料）：https://nlnetlabs.nl/documentation/unbound/

第三十章

DNS 里的信任数据：邮件、SSH 与自动化

一个区里最不显眼的名字，往往是以下划线开头的那些。它们不指向任何一台主机，却是别人决定是否相信你的依据：这封邮件该不该收、这个 SSH 主机密钥该不该信、这张证书能不能签、邮箱客户端该连哪个端口。这类记录统称**信任数据**（trust records）——不是地址簿，是**公开的声明**。

第 29 章把 DNS 当成攻击面看：记录暴露了什么、悬空记录怎么被接管。这里换一个方向，只看**记录里放了什么、谁在读、读不到或读错了会发生什么**：权威区里那几行字的语义与消费端行为。

30.1 一张大表：谁在读这些记录

表里「需 DNSSEC 吗」一列问的是**这条记录要产生安全结论，是否必须依赖 DNSSEC 验证**，不是问「能不能顺手签名」。这一点是本章后面所有结论的分水岭。「轮换频率」指正常运维下改动的频次，不是 TTL。「失效后果」指记录写错、缺失或与对端状态不同步时，消费端会发生什么。

用途	记录类型	名字形态	谁消费	需 DNSSEC 吗	轮换频率	失效后果	相关 RFC
邮件发件授权	SPF, TXT	`example.org. TXT "v=spf1 mx -all"`	收件方 MTA	否	低，改发件源时	一个域只能有一条 `v=spf1`；两条即永久错误（permerror），全量拒收或降级	[RFC 7208]
邮件签名公钥	DKIM, TXT	`sel1._domainkey.example.org. TXT "v=DKIM1; k=rsa; p=<BASE64_PUBLIC_KEY>"`	收件方 MTA	否	中，按 selector 定期轮换	公钥取不到则签名验证失败，进而 DMARC 不通过；`p=` 里放的是**公钥**不是私钥	[RFC 6376]
邮件认证策略	DMARC, TXT	`_dmarc.example.org. TXT "v=DMARC1; p=none; rua=mailto:dmarc@example.org"`	收件方 MTA、报告接收方	否	低，策略收紧时	取不到则按无策略处理；一上来就 `p=reject` 会把合法信源一起拒掉	见官方资料

用途	记录类型	名字形态	谁消费	需 DNSSEC 吗	轮换频率	失效后果	相关 RFC
SMTP 强制 TLS	MTA-STS, TXT + HTTPS 策略	`_mta-sts.example.org` TXT 与 `https://mta-sts.example.org/.well-known/mta-sts.txt`	发件方 MTA	否，靠 HTTPS 与策略缓存	中，策略变更要改 `id`	策略文件 404 或证书不匹配，发送方回退到机会性 TLS，可被降级	[RFC 8461]
TLS 失败报告	TLS-RPT, TXT	`_smtp._tls.example.org.` TXT `"v=TLSRPTv1; rua=mailto:tlsrpt@example.org"`	发件方 MTA	否	低，改报告地址时	收不到失败报告，MTA-STS 的故障对己方不可见	见官方资料
邮件证书绑定（DANE）	TLSA	`_25._tcp.mx.example.org.` TLSA `3 1 1 <CERT_SHA256_HEX>`	发件方 MTA	**必须，只在 DNSSEC 签名区生效**	高，跟每次证书轮换同步	证书换了而 TLSA 没同步，对端按 DANE 硬失败**拒收**，不是降级警告	[RFC 7672] 一类
邮件主机	MX	`example.org.` MX `10 mx.example.org.`	发件方 MTA	否	极低，换邮件主机时	**目标必须是名字，不能是 IP 字面量，也不能是 CNAME**；写错会导致解析不一致或退信	见官方资料
客户端服务定位	SRV	`_imaps._tcp.example.org.` SRV `0 1 993 imap.example.org.`	邮件客户端自动配置	否	低，改端口/主机名时	自动配置失败，客户端回退到手工填写	[RFC 6186] / [RFC 8314] 一类
邮箱客户端自动配置	A / CNAME + HTTPS XML	`autoconfig.example.org` CNAME 或 A，HTTPS 上 `/mail/config-v1.1.xml`	Thunderbird 一类客户端	否，靠 HTTPS 证书	低，改服务器配置时	配置取不到则配不出邮箱；证书不覆盖该名字则报证书错误	见官方资料（Mozilla 格式）
同族自动发现	A / CNAME + HTTPS	`autodiscover.example.org.`	Outlook 一类客户端	否，靠 HTTPS 证书	低	同上，回退手工配置；名字不同、机制同族	见官方资料
SSH 主机公钥指纹	SSHFP	`host.example.org.` SSHFP `<alg> <fptype> <FINGERPRINT_HEX>`	OpenSSH 客户端	**必须，只有 AD 位为真才摆脱 TOFU**	中，主机重装/换密钥时	指纹不匹配会告警或拒绝连接；AD 为假则仍是首次连接信任（TOFU）	[RFC 4255] 一类

用途	记录类型	名字形态	谁消费	需 DNSSEC 吗	轮换频率	失效后果	相关 RFC
限定可签发 CA	CAA	`example.org. CAA 0 issue "ca.example.net"`	CA 签发前的检查方	否	低，换 CA 时	写错会让合法 CA 拒签；**没有 CAA 不等于禁止签发**	[RFC 6844] 一类
ACME 自动续期	DNS-01, TXT	`_acme-challenge.example.org. TXT "example-token-0000"`	CA 验证器、certbot 一类客户端	否	**高，每次续期自动写/删**	记录未生效或残留导致续期失败、证书过期；写权限泄露等于可被冒签证书	见官方资料
服务绑定	SVCB / HTTPS	`example.org. HTTPS 1 . alpn="h3,h2"`	浏览器、客户端	可选，DNSSEC 对它是 OPTIONAL	中，换端点/ALPN 时	参数错会连到不存在的端点或回落；不是访问控制机制	[RFC 9460]
加密解析器发现	DDR / SVCB	`_dns.resolver.arpa.`	客户端自动发现加密解析器	可选，见官方资料	低，解析器部署变更时	客户端可能绕过本地解析器；第 13 章讲为何压制它	[RFC 9462]
域所有权验证	TXT	`_<vendor>-challenge.example.org. TXT`	第三方服务的验证器	否	一次性，验证后应删除	临时记录长期残留，就是第 29 章说的暴露面	见官方资料

DKIM 那一行常被口语叫作 **Domain Key**，因此 `sel1._domainkey` 里的 `selector` 就是这套机制的关键：它让「换公钥」变成「换一个左标签」，而不是「改一条所有人都在缓存的记录」。名字里的 `_domainkey` 是固定标签，`sel1` / `sel2` 才是可轮换的部分。

表格里最容易被误读的两行是 TLSA 与 SSHFP：它们把「安全结论」直接挂在 DNS 应答上，所以 DNS 应答本身必须可认证。其余各行要么靠公钥密码学（DKIM、ACME），要么靠独立的 HTTPS 通道（MTA-STS、autoconfig、autodiscover），要么只是配置提示（SPF、DMARC、SRV、CAA、SVCB），DNSSEC 对它们是**增强**而非**前提**。把这两类混成一类，就会出现「我签了名所以 SSH 一定安全」或者「没签名所以 DKIM 无效」这类反向错误。

30.1.1 逐条 dig 验证命令

表里每一行都有对应的可复制命令。统一用文档地址段里的假解析器 `192.0.2.53`，域名只用 `example.org` / `example.net`。`+dnssec +multi` 用于要看 AD 位与完整字段的场合，`+short` 用于脚本里做断言。

```
# 递归解析器 192.0.2.53，属 RFC 5737 文档段
R="@192.0.2.53"

# SPF：断言「有且只有一条 v=spf1」
dig $R example.org TXT +short | grep -c 'v=spf1'
# DKIM：selector 在名字里，轮换就是换 sel1/sel2
dig $R sel1._domainkey.example.org TXT +short
dig $R sel2._domainkey.example.org TXT +short
# DMARC
dig $R _dmarc.example.org TXT +short
# MTA-STS：TXT 只给「存在性 + id」，策略正文走 HTTPS
dig $R _mta-sts.example.org TXT +short
curl -fsS https://mta-sts.example.org/.well-known/mta-sts.txt
# TLS-RPT
```

```
dig $R _smtp._tls.example.org TXT +short
# TLSA：一定要带 +dnssec 看 AD 位，并核对证书哈希
dig $R _25._tcp.mx.example.org TLSA +dnssec +multi
# MX：目标必须是名字；再单独查这个目标的 A，确认不是 CNAME
dig $R example.org MX +multi
dig $R mx.example.org A +short
# SRV
dig $R _imaps._tcp.example.org SRV +short
# Thunderbird autoconfig：DNS 只负责指路，正文在 HTTPS
dig $R autoconfig.example.org A +short
dig $R autoconfig.example.org CNAME +short
curl -fsS https://autoconfig.example.org/mail/config-v1.1.xml
# 同族 autodiscover
dig $R autodiscover.example.org A +short
# SSHFP：AD 位才是信任开关，字段含义见官方资料
dig $R host.example.org SSHFP +dnssec +multi
# CAA
dig $R example.org CAA +short
# ACME DNS-01：续期窗口里应能看到一次性的验证值
dig $R _acme-challenge.example.org TXT +short
# SVCB / HTTPS
dig $R example.org HTTPS +short
dig $R example.org SVCB +short
# DDR：特殊用途名字，通常由本地递归解析器应答
dig $R _dns.resolver.arpa. SVCB +dnssec +multi
# 域所有权验证 TXT：拿全量 TXT 做残留巡检
dig $R example.org TXT +short
dig $R _verification-example.example.org TXT +short
```

读法上有一个坑：`dig` 输出里的 `ad` 标志是**递归验证器**给出的结论，不是权威服务器「说自己是安全的」。用不带验证的解析器（例如纯转发器）去查 SSHFP，即使记录本身正确，也不会有 AD 位。另一个坑是 `+short` 会吞掉 TTL 与 flags，做断言方便，做排障不够；排障一律换回不带 `+short` 的完整输出。至于「刚加完却查不到」，那是下一节的负缓存问题，不是命令写错了。

30.2　完整配置片段

下面七组片段都是教学用的权威区片段或客户端配置，**不是 Unbound 配置**。每组后面紧跟三件事：为什么这么设、一条可复制的验证命令、失败模式与回滚点。

30.2.1　邮件发件授权与策略

```
; example.org. 区片段；地址均为 RFC 5737 文档段
$TTL 3600
@                   IN  MX    10 mx.example.org.
mx                  IN  A     198.51.100.25
@                   IN  TXT   "v=spf1 mx -all"
sel1._domainkey     IN  TXT   "v=DKIM1; k=rsa; p=<BASE64_PUBLIC_KEY>"
sel2._domainkey     IN  TXT   "v=DKIM1; k=rsa; p=<BASE64_PUBLIC_KEY>"
_dmarc              IN  TXT   "v=DMARC1; p=none; rua=mailto:dmarc@example.org"
_imaps._tcp         IN  SRV   0 1 993 imap.example.org.
_submission._tcp    IN  SRV   0 1 587 smtp.example.org.
```

- **为什么这么设**：MX 目标写名字不写 IP、也不指 CNAME；SPF 全文件只保留一条 `v=spf1`；DKIM 用两个 selector 并存，就是为了能在不中断收信的前提下换公钥；DMARC 先 `p=none` 收报告观察，再逐步收紧；SRV 让客户端免手工填端口。`p=` 里是**公钥**，公开是设计使然。
- **怎么验证**：`dig @192.0.2.53 example.org TXT +short`（数一下 `v=spf1` 是否恰好一条）。
- **失败模式 / 回滚点**：两条 `v=spf1` 会直接 permerror；DKIM 轮换务必按「先加新 selector →切签名→看 DMARC 报告→再删旧 selector」的顺序，**旧 selector 的 TXT 就是回滚点**；删早了在途邮件会验签失败。

30.2.2 MTA-STS 与 TLS-RPT

```
; example.org. 区片段
_mta-sts     IN  TXT  "v=STSv1; id=chg-0007"
_smtp._tls   IN  TXT  "v=TLSRPTv1; rua=mailto:tlsrpt@example.org"
mta-sts      IN  A    203.0.113.10
```

HTTPS 上 `https://mta-sts.example.org/.well-known/mta-sts.txt` 的策略正文：

```
version: STSv1
mode: enforce
mx: mx.example.org
max_age: 86400
```

- **为什么这么设**：TXT 只是「策略存在 + 版本号 `id`」的信号，真正的策略正文经 HTTPS 取；因此 `mta-sts.example.org` 的证书必须覆盖这个名字。TLS-RPT 是配套的眼睛，没有它，对端 TLS 握手失败对己方就是静默的。`mode` 先观察再 `enforce`，字段取值见官方资料。
- **怎么验证**：`dig @192.0.2.53 _mta-sts.example.org TXT +short` 与 `curl -fsS https://mta-sts.example.org/.well-known/mta-sts.txt`。
- **失败模式 / 回滚点**：策略文件 404、证书不匹配或 `mx:` 与实际 MX 不符，发送方会回退到机会性 TLS，安全性静默降级；**改策略必须同时改 `id`**，否则发送方按 `max_age` 继续用缓存里的旧策略，改了等于没改。回滚点是保留上一版 `id` 的 TXT 与旧策略文件，直到旧缓存过期。

30.2.3 TLSA 与 DANE

```
; 仅当 example.org. 已 DNSSEC 签名、且父区有匹配的 DS 时才有意义
_25._tcp.mx   IN  TLSA  3 1 1 <CERT_SHA256_HEX>
mx            IN  A     198.51.100.25
```

- **为什么这么设**：DANE（DNS-Based Authentication of Named Entities）把证书或公钥的哈希放进 DNS，让发件方不必只依赖公共 CA 体系来判断「这张证书对不对」。三个数字的含义见 [RFC 7672] 一类官方资料。它的可信度**完全来自 DNSSEC**：在**未签名的区里，这条记录没有安全意义**，攻击者改应答即可改哈希。
- **怎么验证**：`dig @192.0.2.53 _25._tcp.mx.example.org TLSA +dnssec +multi`（看 `flags` 里有没有 `ad`）。
- **失败模式 / 回滚点**：证书轮换不同步是这条记录最典型的故障——**不是警告，是拒收**。回滚点是一个「双记录窗口」：先加新证书的 TLSA，再换证书，观察投递成功，最后删旧 TLSA。把 certbot 的自动续期接进来时，这个窗口必须由同一次自动化事务完成，不能靠人记。

30.2.4 Thunderbird autoconfig 与同族 autodiscover

```
; example.org. 区片段；CNAME 与 A 二选一，同一名字不能并存
autoconfig      IN  A      203.0.113.10
; autoconfig   IN  CNAME  config-host.example.net.
autodiscover    IN  A      203.0.113.10
```

HTTPS 上 `https://autoconfig.example.org/mail/config-v1.1.xml` 的正文（Mozilla 的 `config-v1.1` 格式，字段以官方资料为准）：

```
<?xml version="1.0"?>
<clientConfig version="1.1">
  <emailProvider id="example.org">
    <domain>example.org</domain>
    <incomingServer type="imap">
      <hostname>imap.example.org</hostname>
      <port>993</port>
      <socketType>SSL</socketType>
      <username>%EMAILADDRESS%</username>
    </incomingServer>
    <outgoingServer type="smtp">
      <hostname>smtp.example.org</hostname>
      <port>587</port>
      <socketType>STARTTLS</socketType>
      <username>%EMAILADDRESS%</username>
    </outgoingServer>
```

```
  </emailProvider>
</clientConfig>
```

- **为什么这么设**：DNS 只负责把 autoconfig / autodiscover 指到一台能答 HTTPS 的主机，配置内容本身走 HTTPS，证书必须覆盖该名字。这就是常被口语叫作「Thunderbird 自动配置」的那组记录；autodiscover 是 Outlook 一类客户端的同族机制，名字不同、机制同族，别当成同一份 XML。
- **怎么验证**：dig @192.0.2.53 autoconfig.example.org A +short 与 curl -fsS https://autoconfig.example.org/mail/config-v1.1.xml。
- **失败模式 / 回滚点**：若用 CNAME 指向托管配置主机，服务下线而记录没删，就是第 29 章的**悬空 CNAME**；证书不覆盖该名字则客户端直接报证书错误，不会「自动降级到明文」。回滚点是保留旧 A 记录与旧 XML，直到客户端自动配置验证通过；下线服务时**先删 DNS 记录**。

30.2.5　SSHFP 与 VerifyHostKeyDNS

```
; example.org. 区片段；用 ssh-keygen -r 生成，不要手抄指纹
host     IN  A      198.51.100.25
host     IN  SSHFP  <alg> <fptype> <FINGERPRINT_HEX>
```

客户端侧：

```
# ~/.ssh/config 片段
Host host.example.org
    VerifyHostKeyDNS yes
    StrictHostKeyChecking yes
```

- **为什么这么设**：把「首次连接时人工核对指纹」（TOFU, Trust On First Use）换成「从签名区取已验证的指纹」。<alg> 与 <fptype> 的取值来自 IANA 注册表，含义见官方资料；用 ssh-keygen -r host.example.org 生成，避免手抄出错。**只有递归验证器把 AD 位置 1（DNSSEC 验证通过）时，SSHFP 才会被用来自动接受主机密钥**；AD 为假，客户端仍走 TOFU，VerifyHostKeyDNS 并没有替你建立信任。
- **怎么验证**：dig @192.0.2.53 host.example.org SSHFP +dnssec +multi（看 ad），再用 ssh -v host.example.org 看是否报告在 DNS 中找到匹配的主机密钥。
- **失败模式 / 回滚点**：主机重装换密钥而 SSHFP 未更新，客户端会告警甚至拒绝连接；回滚点是换密钥前先并挂新旧两条 SSHFP，确认新密钥生效后再删旧。若本区并未签名，应把 VerifyHostKeyDNS 退回 ask 并明确记录「这里仍是 TOFU」——转发器上配不了信任锚，「配了 SSHFP」和「客户端真的信了」是两件事。

30.2.6　CAA 与 ACME DNS-01

```
; example.org. 区片段；issue 值用 example 域名占位，实际按 CA 官方资料填
@                  IN  CAA  0 issue "ca.example.net"
@                  IN  CAA  0 iodef "mailto:ca-alerts@example.org"
_acme-challenge    IN  TXT  "example-token-0000"
```

续期侧凭据文件（字段名随 DNS 插件而定，见官方资料）：

```
# /etc/letsencrypt/dns-provider.ini，权限 0600，属主为续期专用账号
dns_provider_api_token = <TOKEN>
```

- **为什么这么设**：CAA 让 CA 在签发前检查「是否允许我签」；_acme-challenge 的 TXT 由 certbot 一类客户端在每次续期时**自动写入、验证后删除**。这里的关键判断是：**能写这条记录，约等于能为该域签发证书**。所以 example-token-0000 这个一次性验证值公开可读、短期有效，并不机密；真正的高风险凭据是 dns_provider_api_token = <TOKEN> 这一行，以及它背后的服务商账号。
- **怎么验证**：dig @192.0.2.53 example.org CAA +short 与 dig @192.0.2.53 _acme-challenge.example.org TXT +short（续期窗口内）。
- **失败模式 / 回滚点**：记录还没生效或负缓存没过期→ CA 验证失败→续期失败→证书过期全站不可达；CAA 写错→合法 CA 拒签；凭据泄露→第三方可能为自己签发该域证书。回滚点有三条：续期前保留上一张证书与旧私钥、保留「禁用该 API 凭据」的开关、以及一份凭据轮换记录。CAA 是**限定**而非认证，不要把它当成防冒签的万能锁。

30.2.7　SVCB、HTTPS 与域所有权验证 TXT

```
; example.org. 区片段；参数含义见 RFC 9460
@                        IN  HTTPS  1 . alpn="h3,h2"
```

```
@                       IN  SVCB   1 svc.example.org. alpn="h2"
_verification-example   IN  TXT    "example-token-0000"
```

- **为什么这么设**：SVCB / HTTPS（[RFC 9460]）把服务绑定信息——ALPN、端口、替代端点、apex 别名、ECH 一类参数——放进 DNS，客户端据此选端点；DNSSEC 对它是 OPTIONAL。域所有权验证 TXT 是**一次性**的：验证通过就该删，长期留着等于一直挂着一块「这个子域可能被接管」的牌子（第 29 章）。DDR（[RFC 9462]）用的是特殊用途名字 `_dns.resolver.arpa.`，通常由本地递归解析器应答，第 13 章讲为何压制它，这里不重复那段配置。
- **怎么验证**：`dig @192.0.2.53 example.org HTTPS +short` 与 `dig @192.0.2.53 _verification-example.example.org TXT +short`（应当查不到，说明已清理）。
- **失败模式 / 回滚点**：SVCB 参数写错会让客户端连到不存在的端点或回落；验证 TXT 残留是慢性风险而非急性故障。回滚点是变更前把整区记录导出一份快照，逐条核对后删除临时名字。

30.3 共同结论：四条物理性质

上面十六行记录形态各异，但共享同一组物理约束。把这四条记住，比记住任何单条记录的语法都重要。

30.3.1 公开、可读、被缓存，TTL 决定变更速度

权威区里的每一个字节都是**公开可读**的：任何人一条 `dig` 就能拿到，还会被递归解析器按 TTL 缓存、被被动 DNS 与证书透明日志记录、被第三方镜像。因此区文件里绝不能出现私钥、API token 明文、口令、内部主机名拓扑细节。反过来说，DKIM 的 `p=` 放公钥、`_acme-challenge` 放一次性验证值，公开并不违背设计——设计就是让别人来读的。

第二个推论同样重要：**TTL 决定变更生效速度**。一个 3600 秒 TTL 的记录，改完之后在最长一小时内，不同解析器看到的新旧值可能不同。所有「先切再删」「双记录窗口」的操作节奏，都要按 TTL 而不是按秒表来排。变更前把 TTL 调低、等旧 TTL 自然过期、变更、验证、再把 TTL 调回，是这类记录的标准动作。

30.3.2 一次性验证值不是秘密，写权限才是

最容易搞反的一处，是把 `_acme-challenge` 的 TXT 值当机密锁起来，却把能写区文件的 API 凭据随手写进 CI 日志或共享脚本里。前者是一次性验证值，公开、短期、用完即删；后者是**区级写权限**，持有者可以改 SPF 的 `include`、改 MX、加一条自己的 DKIM 公钥，或者给自己签一张证书。

所以最小权限的粒度不是「整个区」，而是「**这一个名字**」：`_acme-challenge.example.org.` 的 TXT，或某个自动化只负责的那一个子域。CAA 可以在一定程度上缩小可签发 CA 的范围，但它管的是「谁被允许签」，不管「谁能改 DNS」，两者别互相替代。

30.3.3 负缓存让「刚加的记录查不到」成为常态

[RFC 2308] 定义的负缓存（negative caching）会把 NXDOMAIN / NODATA 按 SOA 的 minimum 字段缓存下来，常见取值是 3600 秒。于是自动化里最经典的一幕是：脚本刚写入 `_acme-challenge`，立刻 `dig` 一次，拿回「不存在」，判定续期失败。这不是写失败，是**验证链路里某一层还缓存着之前的『不存在』**。

变更前把相关名字的 TTL 与 SOA minimum 调低，是最省事的预防：

```
; 变更前：调低 TTL 与 SOA minimum（教学用示例值）
$TTL 300
@   IN  SOA  ns.example.org. hostmaster.example.org. (
        1 7200 3600 1209600 300 )

# 变更前先等旧 TTL 过期，再改；改完复查完整应答而不是只看 +short
dig @192.0.2.53 _acme-challenge.example.org TXT
# 变更后若仍查不到，链路上每一层缓存都要刷，否则前一层照样返回缓存的 NXDOMAIN
```

- **为什么这么设**：把「不存在的记忆」缩短，让新增记录尽快可见；SOA 各字段语义见官方资料。
- **怎么验证**：`dig @192.0.2.53 _acme-challenge.example.org TXT` 看 TTL 与 `flags`。
- **失败模式 / 回滚点**：验证脚本必须重试并带超时，不能「查一次就判失败」；书里第 4 项实测结论里那条「链路上每个实例都要刷」在这里同样成立。回滚点是等负缓存自然过期，或在受控实例上刷缓存（第 7 章讲采集与运行时操作，此处不重复配置）。

30.3.4　依赖 DNSSEC 的记录在未签名区没有安全意义

TLSA / DANE、SSHFP 这类记录把安全结论直接挂在 DNS 应答上。应答若只靠 UDP/53 的源端口与事务 ID 保护，改应答就能改指纹或证书哈希，此时那条记录只是「更方便的 TOFU」，不是认证。判断标准只有一个：**递归验证器是否真的验过并把 AD 位置 1**。

「本区已签名」也不等于一劳永逸：签名区里还得管 KSK / ZSK / DS 的轮换，DS 与子区不匹配会让整区变成 bogus，连带把这些信任记录一起打成 SERVFAIL。边界只有一句：**转发器与验证器二选一**，在纯转发器上讨论 SSHFP 的 AD 位没有意义。

30.4　自动化写记录的凭据与故障域

certbot、DNS 服务商 API、动态 DNS 客户端，本质是同一类东西：**一个能改区内容的程序化凭据**。对它按下面五条收紧，逐条都能对应到前面某个失败模式。

- **最小权限**：只授 `example-token-` 那个一次性名字所属的 `_acme-challenge.example.org.`，或自动化实际负责的那一个子域；不授整区、不授 NS、不授 MX、不授 SOA。服务商若不支持按名字授权，就把这个域单独放一个账号。
- **独立凭据**：续期脚本、DDNS 客户端、人工运维各用各的凭据，不复用、不写进同一个 CI 变量；凭据文件权限 0600、属主为续期专用账号，值写 `<TOKEN>` 而非明文。
- **审计**：每次写/删记录都留「谁、何时、改了哪个名字、旧值是什么」；这些事件应该进第 31 章的栈里做告警，尤其是非续期窗口里出现的 `_acme-challenge` 写入。
- **轮换与吊销**：凭据定期轮换，并预先验证「禁用这个凭据」的路径真的可用；轮换本身也要走一次完整续期演练，别等到证书过期那天才发现新凭据没生效。
- **回滚点**：变更前导出区快照、保留上一张证书与旧私钥、保留旧 selector / 旧 TLSA / 旧 SSHFP 直到新值验证通过。任何「先删后加」的顺序在这类记录上都是错的。

与第 25 章 DDNS 对照：那边更新的是 A / AAAA，这边更新的是 TXT / TLSA / SSHFP，**写权限模型完全一样**——一个能改区内容的凭据就是区级权限，第 25 章讲的更新凭据隔离与事务边界，在这里具体化成「只授一个 `_acme-challenge` 名字」。

与第 28 章故障域隔离对照：证书续期、邮件投递、SSH 首次信任，都绑在**同一份区数据的可写与可解析**上。若权威区、自动化 API 账号、续期主机、以及排障要用的解析器共用一条链路和一个凭据，一次服务商 API 故障或一次区推送失败会同时打掉这几件事，而且修复动作本身还要依赖这套 DNS。把「谁能改记录 / 改了谁会读 / 读不到时谁降级」列成一张依赖表，是第 28 章隔离框架在信任数据上的具体化：**别把回滚用的那条路，和坏掉的那条路放在同一个故障域里**。

30.5　本章要点

- 信任数据是**公开声明**：区文件里绝不放私钥、API 凭据或内部拓扑；DKIM 公钥与 ACME 一次性验证值公开是设计使然，机密的是**写权限**。
- 「需 DNSSEC 吗」是全章分水岭：SSHFP 与 TLSA/DANE 必须靠 AD 位为真的验证应答，在未签名区里没有安全意义；其余记录靠公钥、HTTPS 或只是配置提示，DNSSEC 只是增强。
- 失效后果要按消费端写：两条 `v=spf1` 是 permerror，MX 指 IP/CNAME 会解析不一致，TLSA 与证书轮换不同步是**拒收**而非警告，MTA-STS 策略取不到是静默降级到机会性 TLS。
- 负缓存（[RFC 2308]）让「刚加的记录查不到」成为常态：变更前调低 TTL 与 SOA minimum、验证脚本要重试、链路上每层缓存都要刷；所有轮换都用「先加新、再切、后删旧」的双记录窗口。
- 自动化写记录的凭据按「最小权限到单个名字、独立凭据、审计、可吊销轮换、有回滚点」五条收紧；`_acme-challenge` 的写权限约等于该域的证书签发权。
- 与第 25 章 DDNS、第 28 章故障域隔离呼应：这些记录把内部系统绑到外部 DNS 上，回滚路径不能和故障路径共用同一个凭据、同一条链路。

30.6　参考资料

- RFC 7208（SPF）：https://www.rfc-editor.org/rfc/rfc7208
- RFC 6376（DKIM）：https://www.rfc-editor.org/rfc/rfc6376
- RFC 8461（MTA-STS）：https://www.rfc-editor.org/rfc/rfc8461
- RFC 7672（DANE / SMTP TLSA）：https://www.rfc-editor.org/rfc/rfc7672
- RFC 4255（SSHFP）：https://www.rfc-editor.org/rfc/rfc4255
- RFC 9460（SVCB / HTTPS）：https://www.rfc-editor.org/rfc/rfc9460

- RFC 6186（邮件客户端自动配置的 SRV 用法）：https://www.rfc-editor.org/rfc/rfc6186
- certbot 官方站（DNS-01 插件与凭据文件权限见官方资料）：https://certbot.eff.org/

其余 RFC 编号（如 [RFC 2308]、[RFC 9462]、[RFC 6844]、[RFC 8314]、[RFC 5737]）按第四篇统一索引查 RFC Editor；Thunderbird `config-v1.1.xml`、autodiscover、DMARC、TLS-RPT、ACME 与域所有权验证的规范编号、年份与字段取值，本章一律写「见官方资料」，不做断言。

第三十一章

可观测性栈：dnstap、Grafana、eBPF 与告警

一个 `unbound-control stats_noreset` 能告诉你「现在有多少 SERVFAIL」，却回答不了「哪个客户端把某个域刷出了几千个不同名字」，也回答不了「出向 UDP/53 是被 nftables 吃了，还是被上游丢了」。把这三种问题混在一处排查，结局往往是：指标看着正常，用户仍在报障，于是反复重启 Unbound 刷缓存，而真正的断点在别处。可观测性不是一个开关，是三**条互相独立、又能互相印证的链路**：`unbound-control` 按需统计、dnstap 查询流、以及操作系统侧的收发与归因。这里讲怎么接进指标与日志系统、怎么把数字变成能分级/降噪/演练的告警。

31.1 先划边界：采集点已有，缺的是栈

31.1.1 三层数据源与各自的盲区

层	采集点	能回答	回答不了	载体
①应用内	`unbound-control stats_noreset`、`log-servfail`、`get_option`	解析成功率、命中率、递归耗时、DNSSEC bogus、SERVFAIL 原因、生效配置值	「谁问的」「问了什么名字」、单源速率、内核丢包	命令行 / exporter
②查询流	dnstap（socket → consumer →结构化日志）	客户端地址、QNAME、QTYPE、应答码、上游交互、唯一 FQDN 数	报文在内核丢在哪、是否被 tproxy 改写、TCP 重传	JSON / fstrm / Loki / Elasticsearch
③内核与网络	eBPF、`ss`、`conntrack`、nft counter	UDP/53 收发、丢包与重传、按 PID/UID 归因、TCP 回退、mark 路径	QNAME 内容、缓存命中与否、DNSSEC 状态	bpftrace / tcpdump / nft

三层不是「越多越好」，而是**盲区互补**：①是累积计数器，适合趋势与告警；②是高基数明细，适合取证与隧道检测，但体积与隐私代价最大；③不碰 DNS 语义，却恰好能证明「包有没有离开这台机器」。只装一层的误判很有特征：只有①时 `num.queries_timed_out` 上涨像上游挂了，实际可能是第 14 章 `output-tproxy` 把解析器自己的出向 53 打了 mark；只有②时查询一条不少却没有出向消息，既可能是 `local-zone` 接管（第 24 章优先级），也可能是 consumer 断流只收到半边；只有③时看得出包在走，却分不清 NOERROR 与 SERVFAIL。所以验收标准不是「装了 Grafana」，而是**同一次故障注入，三条链路都能各自看到对应信号，且时间戳能对上**；第 7 章的 `extended-statistics` 让统计多出 QTYPE 等维度、`statistics-interval: 0` 关掉周期性打点改由控制通道按需取，本章沿用，不重贴配置块。

31.2　第一层：应用内统计与 SERVFAIL 日志

31.2.1　stats_noreset 的读数纪律

stats 会重置计数（受 statistics-cumulative 控制），基线必须全程用 stats_noreset。集成进栈后多一条纪律：**只读一次，别让两个采集器同时读**。下面这段是可复制的快照骨架；若已装社区 exporter，就不必再跑第二套采集：

```
#!/bin/sh
# /usr/local/sbin/unbound-stats-snapshot.sh
set -eu
CTRL="unbound-control -c /etc/unbound/recursor.conf"
OUT=/var/lib/node_exporter/textfile_collector/unbound.prom
$CTRL stats_noreset | grep -E '^(total\.num\.(queries|cachehits|cachemiss)|num\.(queries|SERVFAIL|answer\.bogus|queries_timed_out|
mv /tmp/ub-stats.$$ "$OUT.snapshot"
```

为什么这么设：grep 只留关键计数器，不把几十个 num.query.type.* 明细推进时序库；临时文件加 mv 是原子替换，采集器不会读到半截内容。num.query.opcode.UPDATE 这类**非 QUERY opcode 计数**（完整字段名见官方手册扩展统计章节）值得单独留意：递归解析器正常收不到 UPDATE，它持续非零通常意味着动态更新报文打错了地址，或第 25 章 DDNS 链路配错了目标。另外 unbound-control 的运行时 local_zone 增删（第 24 章）也可当作一条变更事件记进日志，便于把「过滤表更新」与命中率变化对上。
怎么验证：$CTRL stats_noreset | grep -E 'num.query.opcode|num.answer.bogus|num.queries_timed_out' 应列出字段；两次执行之间做一次 dig @192.0.2.53 example.org A，total.num.queries 增量应为 1 而非被清零。
失败模式 / 回滚点：控制通道未启用或证书权限不对时整条①链路**静默失效且无告警**，因此后面必须有「采集器本身无数据」的告警。回滚：删掉脚本与定时项，回到人工按需执行，不要改成 stats 凑数。

31.2.2　关键指标清单

指标	层	判读要点
qps（total.num.queries 速率）	①	与「单源 qps」分开；总量平稳不代表无单源突增
缓存命中率（cachehits / (cachehits + cachemiss)）	①	骤降常是冷缓存、刚刷过缓存、或查询换成随机子域
SERVFAIL 数 / 率	①	**用比率不用绝对值**，业务量翻倍时绝对值天然翻倍
NXDOMAIN 比	②	与第 29 章 DGA / 扫描特征联动
上游 RTT / 超时数（recursion.time.avg、num.queries_timed_out、num.recursivereplies）	①	超时涨而递归应答不涨，基本可断定出向链路或上游（第 7 章）
递归查询数（total.num.cachemiss）	①	与 qps 比值异常抬升说明缓存失效或被绕过
unwanted 计数	①	被拒 / 不想要的查询，字段名见官方手册对应章节；呼应第 9 章 ACL
单源 qps	②	必须按客户端地址聚合，①层做不到
唯一 FQDN 数（按二级域、按源）	②	第 29 章认定的最通用隧道检测量，与编码 / QTYPE / 长度无关
ratelimit 命中数	①	官方标 experimental，阈值按 stats_noreset 基线调（第 29 章）
DNSSEC bogus 数（num.answer.bogus）	①	突增多半是信任锚 / 上游验证链问题，不是「网络慢」
dnstap 帧速率 / 日志卷可用空间	② / 主机	帧速率归零即断流；30 天保留期写死后磁盘是最常见的静默故障点

31.2.3　日志分级与 NTP 这个前提

verbosity 平时保持 1，排障临时调到 2–3，排完立刻调回；log-servfail 独立于 verbosity、以 error 级别输出，是性价比最高的一行，把 verbosity 拉到 4 换来的多半是无关调试行和打满的磁盘。下面只读 / 临时改生效值，配置块在第 7 章，不重复：

```
CTRL="unbound-control -c /etc/unbound/recursor.conf"
$CTRL get_option verbosity                # 先读再改，才知道回滚到哪
$CTRL get_option log-servfail
$CTRL verbosity 3                         # 排障窗口：专用命令临时提高详细度
timedatectl show -p NTPSynchronized --value # 日志关联的前提
chronyc tracking 2>/dev/null | head -5 || true
```

为什么这么设：日志关联的前提是**所有链路共用同一个可信时钟**。dnstap 帧、journald、nft 计数、Prometheus 抓取时间只要差几秒，「注入→指标抬升→日志出现」的因果链就会拼错，甚至得出「告警早于故障」的荒谬结论。NTP 时间同步（协议规范见 RFC Editor 官方资料）不是可选项；`timedatectl` 只确认本机已同步，跨主机还要确认是否同源。`verbosity` 是 `unbound-control` 的专用命令，与 `set_option` 不同；`log-servfail` 不在 `set_option` 支持的选项清单内，应固化在配置文件（第 7 章）。
怎么验证：`timedatectl show -p NTPSynchronized --value` 应回 `yes`；`gw-01` 与 `cloud-01` 上 `date -u` 一致到秒级；`journalctl -u unbound --since '2 min ago'` 与 dnstap JSON 时间字段能对上。
失败模式 / 回滚点：`verbosity` 命令与 `set_option` 都是运行时临时值、reload 即失效，别当成已固化配置写进演练报告；`chronyc` 未安装时第二行被 `|| true` 吞掉，此时要用别的手段确认同步。回滚：`$CTRL verbosity 1`，配置文件里保留 `log-servfail: yes`。

一条 SERVFAIL 的原因要落到类别：上游超时、DNSSEC 验证失败（bogus）、根区 / 转发区无法建立、缓存里的旧坏条目。`log-servfail` 给「这一条为什么失败」，`num.answer.bogus` 给「这类失败有多少」；告警挂后者速率才稳定，**定性仍要靠前者**，否则 bogus 告警响了还得从头翻日志。第 7 章「哪个实例→访问控制→日志原因→缓存旧条目→ nftables 出向豁免→ EDNS/MTU」的顺序在这里仍是入口。

31.3 第二层：dnstap 查询流的集成与隐私边界

31.3.1 socket → consumer 的拓扑

集成侧要画清的是谁在监听，以及监听者挂了会怎样：



为什么这么设：把 consumer 画成**独立进程**是关键。socket 是写入目标、不是队列；没有 consumer 监听时写入失败，查询流副本整段消失，而解析服务通常照常工作——这正是「静默失效」的典型形状。日志侧与指标侧最终汇到同一告警入口、同一条 Grafana 时间轴，才能做「先看到指标抬升，再下钻到 QNAME」。
怎么验证：`ls -l /run/dnstap.sock` 看 socket，`ss -xlp | grep dnstap` 看谁在监听，consumer 帧计数（随实现而异）在 `dig @192.0.2.53 example.org A` 前后应递增——三者缺一都不算「链路上有数据」。
失败模式 / 回滚点：consumer 崩溃 / 重启窗口丢帧；socket 权限过宽会把全量查询流暴露给本机任意用户。回滚：`dnstap-enable: no` 并 restart（配置在第 7 章），同时**保留到期删除任务**——关掉采集不会让历史文件消失。

31.3.2 消息类别、全量与采样

dnstap 消息按观察点与方向展开：client、resolver、forwarder 等观察点各配 query / response 两个方向，第 7 章六项 `dnstap-log-*-messages` 全开覆盖的正是这三组观察点的两个方向。按**四类**口径使用：客户端查询、客户端应答、解析器（含转发器）出向查询、出向应答；完整消息类型与字段定义见官方 dnstap 资料。

策略	做法	代价	适用
全量	六项全开，全部落盘，30 天到期删除	磁盘与 I/O 最大；副本最完整	本书默认；隧道检测与取证必须全量
按方向采样	只留 client 侧，或按 1/N 抽帧	无法还原「问了哪个上游」	只关心客户端行为

策略	做法	代价	适用
按名字 / 源采样	排除健康探测域与白名单源	**容易把异常一并排除**	容量受限时的退让，需白名单审计

采样最危险的不是丢数据，而是**丢掉的恰好是异常**：隧道流量通常 qps 不高，1/N 抽样后唯一 FQDN 数被系统性低估，第 29 章的流量分析直接失效。宁可明确写死保留期 + 到期删除，也不在采集端悄悄采样。落盘格式常见 JSON / fstrm 两类，字段名与时间字段含义随 consumer 实现而异，见所选实现的官方文档。

31.3.3　查询日志本身就是隐私副本

这一条要在集成方案里**再次明确**，因为它比第 7 章的单机配置更容易破功：单机时日志只在本机磁盘，接进栈之后它会被推送、索引、复制到副本、被 Grafana 跨主机调阅、被备份系统再存一份。「抗元数据聚合」在自己门口破功，往往就发生在这一步。三条硬边界：

- **保留期跟着副本走**：Loki / Elasticsearch 的索引保留、快照、对象存储桶生命周期都要与第 7 章的 30 天一致；只清理源主机 `/var/log/dnstap` 而索引里还留一年，等于没设保留期。
- **访问范围按最小权限**：日志面板账号与指标面板账号分开；能查 QNAME 的人应比能看 qps 的人少一个数量级；`<TOKEN>` 类凭据独立、可轮换、有审计。
- **不无条件推第三方 SaaS**：把全量查询流推到外部托管日志服务，等于把「家里每个客户端问过什么」交给第三方长期保存。这是**数据主权决定**，要在方案评审里显式写下来。

dnstap 里带 QNAME、客户端地址、时间戳、应答码——没有报文正文，但已足够还原一个网络里所有终端的行为轨迹。它和 DNSSEC 完全无关：DNSSEC 是数据完整性 / 来源认证，DoT / DoH / DoQ 是传输加密 / 隐私，而 dnstap 是**你自己主动造的一份明文元数据副本**。三者不能互相替代，也不能用「我们开了 DoT」给日志留存辩护。

31.3.4　轻量方案：JSON 落文件加 `jq` 巡检

不是每个环境都需要 Loki。把 consumer 输出的 JSON Lines 按小时切文件，配一个巡检脚本就能覆盖第 29 章大部分检测量。字段名随 consumer 实现而变，下面是骨架：

```
#!/bin/sh
# /usr/local/sbin/dnstap-json-inspect.sh ——字段名为示意，以所选 consumer 输出为准
set -eu
LOG=/var/log/dnstap/queries.jsonl
# ① 按「源 + 二级域」统计唯一 FQDN 数（隧道检测最通用的量）
jq -r 'select(.type == "CLIENT_QUERY") | [.client_ip, .qname] | @tsv' "$LOG" \
  | awk -F'\t' '{ n = split($2, p, "."); d = (n >= 2 ? p[n-1] "." p[n] : $2);
                  seen[$1 "\t" d "|" $2] = 1 }
                END { for (k in seen) { split(k, a, "|"); cnt[a[1]]++ }
                      for (c in cnt) print cnt[c], c }' | sort -rn | head -20
# ② payload 粗筛：超长单标签（QNAME 总长上限见 [RFC 1034] 相关条文）
jq -r 'select(.type == "CLIENT_QUERY") | .qname' "$LOG" \
  | awk -F'.' '{ for (i = 1; i <= NF; i++) if (length($i) >= 40) print length($i), $0 }' \
  | sort -rn | head -20
# ③ 断流自检：文件超过 10 分钟无新增即视为可疑
find "$LOG" -mmin +10 -print
```

为什么这么设：①只依赖「名字去重」，与编码 / QTYPE / 长度无关，能覆盖未知隧道（第 29 章流量分析一支）；②是 payload 分析一支的快速粗筛，**只产候选不做判定**；③把「没有数据」变成可检查状态，这是整套栈最容易被漏掉的告警。三支都不联网、不依赖外部服务。
怎么验证：先 `dig @192.0.2.53 www.example.org A`，①应输出 1 级计数；再注入一批 `label01.tunnel.example.net` … `label99.tunnel.example.net`，①里该域计数应接近注入条数，②把标签拉长到 40 字节以上应列出候选。
失败模式 / 回滚点：JSON 字段名不是 dnstap 协议强制的，`select(.type == "CLIENT_QUERY")` 在别的 consumer 上可能是 `message.type` 或嵌套结构，上线前必须 `head -1 "$LOG" | jq .` 对一次字段；②的 40 字节只是教学取值，CDN 与证书续期会产生合法长标签，直接当告警必然误报。回滚：脚本不改采集配置，停掉定时项即可。

31.3.5 重量方案：Loki / Elasticsearch 与告警

当节点多到需要跨主机检索 QNAME（第 26 章 anycast 的 per-node 集中观测就在这一层落地），把 JSON 送进日志系统更合适。骨架仍是 consumer → 结构化日志 → 查询语言聚合 → 告警规则。

为什么这么设：明细与取证留在日志系统（保留期与 30 天对齐），趋势交给指标系统（dnstap_unique_fqdn_per_domain、dnstap_frames_total 这类低基数聚合量），告警只挂低基数指标与「无数据」状态、明细由人从告警链接下钻。把 QNAME 直接当 Prometheus label，等于把高基数问题搬进时序库。
怎么验证：dig @192.0.2.53 probe31.example.net A 后应能按该 QNAME 检索到，且 instance / node 标签指回正确节点——同时验收第 26 章要求的 per-node 区分能力。
失败模式 / 回滚点：索引保留期与源主机不一致、副本数放大存储、日志系统宕机导致 consumer 背压堵住 socket。回滚：consumer 保留「同时落本地文件」分支，切回轻量方案，而不是停采集。

31.4 第三层：内核与网络，不改 Unbound

31.4.1 先说明白：Unbound 不原生支持 eBPF

Unbound 不原生支持 eBPF，官方也没有「eBPF 导出器」这类配置项。这里的 eBPF 是**操作系统侧的观测能力**：在内核里挂探针观察 UDP/53 与 TCP/53 的收发、丢弃、重传与进程归因，Unbound 不用改配置、不用重编译、不用重启。代价在另一侧：需要内核支持与相应权限，探针名与可用挂载点随内核版本变化，具体以 bpftrace 与内核官方文档为准（见参考资料），本书不对任何发行版的内核版本作断言。

31.4.2 bpftrace / tcpdump 教学命令

```
# ① 先列可用探针：名字随内核版本变化，别照抄后面那段
bpftrace -l 'kprobe:udp*' | head -20
bpftrace -l 'tracepoint:net:*' | head -20
# ② 按 PID / UID 归因统计 53 端口相关 UDP 收发（探针名请按 ① 的结果替换）
bpftrace -e '
  kprobe:udp_sendmsg /comm == "unbound"/ { @tx[pid, uid, comm] = count(); }
  interval:s:10 { print(@tx); clear(@tx); }
'
# ③ TCP 回退与重传：先看有没有 SYN 到 53，再落盘细看
tcpdump -ni any 'tcp port 53 and tcp[tcpflags] & tcp-syn != 0' -c 20
tcpdump -ni any -s 0 'port 53' -w /tmp/dns-53.pcap
```

为什么这么设：②的价值是「按 PID/UID 归因」——第 14 章 meta skuid unbound 的 uid 豁免是否真生效，与其猜，不如直接看哪个进程在往 53 发包。③抓 TCP/53 的 SYN，是因为 EDNS 报文过大被分片丢弃后会重试 TCP（第 4 章 1472 字节那个坑），「UDP 出去没回来 + 紧跟着 TCP SYN」是一条很干净的证据链。抓包文件同样属于查询流副本，跑完就删。
怎么验证：dig @192.0.2.53 example.org A +short 期间②的 @tx 应出现 unbound 的 pid/uid；dig @192.0.2.53 example.org A +bufsize=4096 观察③是否出现 TCP/53 SYN；bpftrace -l 无输出说明内核或工具链不支持，此时不要硬套探针名。
失败模式 / 回滚点：探针名不存在时 bpftrace 直接报错，不会静默给错数据；但 tcpdump -w 长时间挂着会写满磁盘。回滚：Ctrl-C 结束 bpftrace，rm -f /tmp/dns-53.pcap，不改 Unbound 任何配置。

31.4.3 ss / conntrack / nft counter：先看 nftables

eBPF 需要内核支持，下面这组几乎到处都有，且是第 14、22 章那条经验——**先看 nftables 规则**——的直接延续：

```
# ① 本机谁在听 :53 / :5353（第 14 章 tproxy 目标端口）
ss -u -a -n -p | grep -E ':53\b|:5353\b|:5357\b' || true
# ② UDP/53 的 conntrack 条目（NodeLocal DNSCache 那类竞态的观察点，第 20 章）
conntrack -L -p udp --dport 53 2>/dev/null | head -20 || true
# ③ 只读看规则与计数，确认 mark / skuid 豁免那几行还在
nft list ruleset | grep -i -E 'counter|mark|dport 53|skuid' | head -30
nft list chain inet mangle output-tproxy
```

为什么这么设：①确认「解析器出向豁免」有没有被规则改动顶掉；②看 UDP/53 conntrack 是否堆积，是重传与超时的旁证；③是**只读**查看（不同于 nft reset counters），不会破坏基线。nft counter 能回答①②两层都回答不了的问题：那个包究竟有没有走到 meta mark set 那一行。

怎么验证：先 `nft list ruleset | grep -i counter`，做一次 `dig @192.0.2.53 example.org A`，再读一次，对应规则 packets 应递增；若 `output-tproxy` 链尾计数递增而解析超时，优先怀疑第 14 章那类「自己的出向被自己劫持」。
失败模式 / 回滚点：`nft reset counters` 会清零所有计数，排查前后不要随手执行，否则演练差分证据全没了；conntrack 表为空不代表没有流量（UDP 条目寿命短，或工具未安装被 `|| true` 吞掉）。恢复规则集用第 14 章备份文件整体 `nft -f`，不要逐条删。

31.5　集成骨架

31.5.1　stats → Prometheus → Grafana

常见接法是社区 exporter（如 `unbound_exporter` 一类实现）把 `unbound-control stats` 转成 `/metrics`，Prometheus 抓取，Grafana 出面板；也可走上一节的 textfile 快照。下面 `prometheus.yml` 片段是可复制骨架：

```
# /etc/prometheus/prometheus.yml 片段
global: { scrape_interval: 15s, evaluation_interval: 15s }
rule_files: [ /etc/prometheus/rules/unbound.yml ]
alerting:
  alertmanagers:
    - static_configs:
        - targets: ['192.0.2.53:9093']
scrape_configs:
  - job_name: unbound-recursor
    static_configs:
      - targets: ['192.0.2.53:9167']
        labels: { role: recursor, node: gw-01 }
  - job_name: unbound-forwarder
    static_configs:
      - targets: ['203.0.113.10:9167']
        labels: { role: forwarder, node: cloud-01 }
  - job_name: node
    static_configs:
      - targets: ['192.0.2.53:9100', '203.0.113.10:9100']
```

为什么这么设：`role` 与 `node` 两个 label 是后面所有面板与告警的分组依据——第 7 章多实例、第 26 章 per-node 差异都靠它区分，缺了它两台节点的 SERVFAIL 会被混成一条曲线；`node` job 提供磁盘指标，没有它就写不出「磁盘将满」告警。exporter 的**具体指标名随实现而变**，本节与下一节表达式都是骨架，字段名以 `/metrics` 实际输出为准。
怎么验证：`promtool check config /etc/prometheus/prometheus.yml` 无报错；`curl -s http://192.0.2.53:9167/metrics | head -20` 能看到计数器；Prometheus `/targets` 页两个 unbound job 为 UP。
失败模式 / 回滚点：exporter 与控制通道绑在一起，`unbound-control` 连不上时它往往仍返回 200 但指标停滞——**UP 不等于有数据**，所以要有「序列无变化」类告警。回滚：注释掉对应 `scrape_configs` 条目并 reload；Prometheus 与 Alertmanager 凭据一律 `<TOKEN>`，不要写进可公开仓库。

31.5.2　Grafana 面板清单

面板按「先看总量、再看比例、最后看可疑度」排，不要一屏塞满：

面板	表达式意图（骨架）	判读要点
qps（按 role / node 分组）	`rate(queries_total[5m])`	总量与单源分开；切流后看节点间是否均衡（第 26 章）
SERVFAIL 率	`rate(servfail_total[5m]) / clamp_min(rate(queries_total[5m]), 1)`	用比率；`clamp_min` 避免低流量除零尖刺
缓存命中率	`cachehits / (cachehits + cachemiss)`	骤降关联冷缓存 / 刷缓存 / 随机子域
上游 RTT 与超时	`recursion_time_avg`、`rate(timed_out_total[5m])`	超时涨而递归应答不涨→出向链路（第 7、14 章）
隧道可疑度	`dnstap_unique_fqdn_per_domain` Top N + 长标签候选数	对应第 29 章流量 / payload 两支；必须带白名单过滤

面板	表达式意图（骨架）	判读要点
采集链路健康	`rate(dnstap_frames_total[5m])`、exporter 序列新鲜度	单独一行，避免「没告警」其实是「没数据」
日志卷余量	node exporter `avail_bytes / size_bytes`	30 天保留期的容量前瞻

面板放一个统一时间轴变量，指标面板与 dnstap 日志面板共用同一区间，否则「先看到曲线、再下钻 QNAME」会因来回改时间范围而做不下去。指标是等间隔采样、dnstap 是逐条事件、nft counter 是累计计数，三者天然不同步：以 NTP 同步后的 UTC 为唯一时基，注入动作记一条带 `date -u` 的锚点，指标看锚点后第一个抓取窗口，dnstap 看锚点前后几秒明细，nft counter 看注入前后**读数差**而非绝对值；多节点（第 26 章）还要确认 NTP 同源，否则 per-node 曲线会横向错位。

31.6 告警规则示例与误报治理

31.6.1 八条规则的总表

阈值全是**占位值**，必须按下一节 7 天基线替换，`for` 抑制单窗口抖动；第 8 条最容易被省掉却最值钱。

#	告警	来源	触发条件（骨架）	分级
1	SERVFAIL 率超阈	①	比率连续 5 分钟 > 基线倍数	critical
2	单源 qps 突增	②	某客户端 qps > 基线 N 倍，持续 5 分钟	warning
3	唯一 FQDN 数异常	②	按域 / 按源唯一名字数超基线	warning
4	上游全超时	① + ③	超时涨、递归应答不涨，且 nft 出向计数有包	critical
5	DNSSEC bogus 突增	①	`num.answer.bogus` 速率超基线	critical
6	dnstap 断流	②	帧计数 10 分钟无增量 / socket 无 consumer	warning
7	磁盘将满	③ / 主机	日志卷可用空间 < 15%	warning
8	采集器无数据	①	exporter 序列长时间不更新	warning

前七条全依赖「数据还在来」，采集链路自己断掉时仪表盘会显示一条安静的平线，很容易被误读成「系统很健康」。隧道类规则（#2、#3）刻意留在下面片段之外：它们的量由日志侧聚合，表达式随聚合实现而变，写成一段看似能跑的 PromQL 反而误导。

31.6.2 Prometheus 规则片段

```yaml
# /etc/prometheus/rules/unbound.yml ——阈值为占位，按 7 天基线替换
# 指标名随所选 exporter 而定，以 /metrics 实际输出为准
groups:
  - name: unbound-health
    rules:
      - alert: UnboundSERVFAILRateHigh
        expr: rate(unbound_servfail_total[5m]) / clamp_min(rate(unbound_queries_total[5m]), 1) > 0.05
        for: 5m
        labels: { severity: critical }
        annotations: { summary: "SERVFAIL 率连续 5 分钟超阈（先查 log-servfail 原因）" }
      - alert: UnboundDNSSECBogusSpike
```

```
        expr: rate(unbound_answer_bogus_total[5m]) > 1
        for: 5m
        labels: { severity: critical }
        annotations: { summary: "DNSSEC bogus 突增：查信任锚与上游验证链" }
  - name: dnstap-pipeline
    rules:
      - alert: DnstapNoFrames
        expr: increase(dnstap_frames_total[10m]) == 0
        for: 5m
        labels: { severity: warning }
        annotations: { summary: "dnstap 断流：检查 socket 与 consumer 进程" }
      - alert: DnstapDiskFilling
        expr: node_filesystem_avail_bytes{mountpoint="/var/log/dnstap"} / node_filesystem_size_bytes{mountpoint="/var/log/dnstap"}
        for: 10m
        labels: { severity: warning }
        annotations: { summary: " 日志卷将满：核对 30 天到期删除任务与索引保留期" }
```

为什么这么设：SERVFAILRateHigh 用比率，业务量翻倍不误报；clamp_min(..., 1) 防夜间低流量分母趋零把曲线打飞；critical 规则的 annotation 直接写明第一步查什么，把第 7 章排查顺序塞进告警本身；DnstapNoFrames 用 increase == 0 而非 absent()，因为「指标存在但停滞」正是 consumer 挂掉的典型形态。
怎么验证：promtool check rules /etc/prometheus/rules/unbound.yml 无报错；/alerts 页能看到八条规则（其余在 Alertmanager 侧）处于 inactive；注入演练后对应规则应转 pending 再 firing。
失败模式 / 回滚点：指标名与所选 exporter 不匹配时规则会**永久 inactive 且不报错**，这是最隐蔽的失败；阈值拍脑袋写死会在峰谷反复误报，最后被整体静音——告警被静音比没有告警更危险。回滚：从 rule_files 摘掉该文件并 reload，先恢复人工巡检。

31.6.3　先测 7 天基线

```
#!/bin/sh
# /usr/local/sbin/unbound-baseline.sh ——每小时一次，连续跑 7 天
set -eu
CTRL="unbound-control -c /etc/unbound/recursor.conf"
echo -n "$(date -u +%Y-%m-%dT%H:%M:%SZ) " >> /var/log/unbound-baseline.tsv
$CTRL stats_noreset | grep -E '^(total\.num\.queries|num\.SERVFAIL|num\.answer\.bogus|num\.queries_timed_out|total\.num\.cachehits
echo >> /var/log/unbound-baseline.tsv
```

为什么这么设：7 天覆盖完整周周期——工作日与周末的查询量、夜间批处理、证书续期与健康探测节奏都不同，只测一天会把周期性峰值当异常。文件是**累积快照**，看速率要取相邻两行之差；用 stats_noreset 才不会自己把计数读没。
怎么验证：7 天后 wc -l /var/log/unbound-baseline.tsv 接近 168；画出 qps 与 SERVFAIL 率日曲线，取 p95 / p99 与最大周期峰值作阈值上界，而不是取均值。
失败模式 / 回滚点：基线期若恰含真实故障或演练，阈值会被污染，须标注并剔除该窗口；基线含查询计数但不含 QNAME，仍按内部文档保管。回滚：停掉定时项，保留文件供复盘。

31.6.4　白名单：CDN、健康探测、证书续期

第 29 章 payload 分析（超长 QNAME、高熵标签、罕见 QTYPE）会撞上一批完全合法的流量，白名单要按**产生原因**分类，而不是按「看着像不像坏」临时加。

来源	特征	误报落在	处理
CDN / CNAME 链	长 QNAME、多标签、一批新子域	#2、#3、长标签粗筛	按已知域后缀与已知源白名单，且**定期复审**
健康探测 / 监控	固定高频、周期性、名字可枚举	#2 单源 qps	按源 + 探测名字精确放行，不整段放行该源
证书续期（ACME DNS-01）	突发 _acme-challenge TXT、标签较长	#3、payload 粗筛	按名字形态与时间窗放行（第 30 章）
授权的内部扫描	唯一 FQDN 数骤增	#3	授权窗口内静音该源，窗口外照常告警，**留审计记录**

三条纪律：白名单**尽量窄**（源 + 名字形态 + 时间窗）、**要有到期复审**、变更本身要有审计。把「某个 CDN 域」整段加进

白名单，等于给第 29 章描述的隧道开了一条免检通道——攻击者只要把数据编码进这个域的子域标签即可。告警分级也在这时定：critical 只给「用户已受影响」的信号（SERVFAIL 率、bogus 突增、上游全超时），warning 给「可疑但需人工确认」的（隧道可疑度、单源突增），采集链路健康单独分组、走不同接收人；分级不清的后果是全部塞进同一个群，然后集体静音。

31.7 演练：三条链路都要看见

第 28 章给了故障域隔离的注入清单，这里只做**观测验收**——验证三条链路各自能不能产生可辨认信号。演练前备份配置，凭据用 `<TOKEN>` 且使用记录归档。

31.7.1 注入 1：关掉一个上游

```
date -u +%Y-%m-%dT%H:%M:%SZ | tee /tmp/drill-anchor.txt
# 教学实验里临时停用某一个上游（做法随部署形态而变，见第 5、14 章）
unbound-control -c /etc/unbound/recursor.conf list_forwards
for i in $(seq 1 12); do dig @192.0.2.53 example.org A +short >/dev/null; sleep 1; done
# ① 层：超时与 SERVFAIL
unbound-control -c /etc/unbound/recursor.conf stats_noreset | grep -E 'num.queries_timed_out|num.SERVFAIL|num.recursivereplies'
# ② 层：consumer 侧看出向消息有去无回（命令随实现而定）；③ 层：出向包与 mark
nft list ruleset | grep -i -E 'counter|mark' | head -20
```

为什么这么设：单上游失效是 fail-closed 场景（第 5 章 `forward-first: no`）。验收点不是「会不会 SERVFAIL」，而是**能否把失败定位到出向链路**：①超时涨、②出向查询有去无回、③看到包出门且 mark 正确，三者齐备才算定位成立。
怎么验证：注入前后两组 `stats_noreset` 相减，`num.queries_timed_out` 应上升；`nft` 对应规则 packets 差分非零；dnstap 里应能检出应答变长或缺失的 resolver / forwarder 消息。
失败模式 / 回滚点：多上游时「关一个」未必产生可观测变化（竞速取最快，第 4 章「单上游不是冗余」的含义），演练要在单上游实验实例做。恢复后负缓存可能继续作祟（第 7 章），按控制面→权威→递归→客户端逐层刷，且**链路上每个实例都要刷**。

31.7.2 注入 2：注一批高熵 QNAME

```
# 教学用高熵标签，域名均为示例域
for i in $(seq -w 1 80); do dig @192.0.2.53 "x${i}a9f3c7e1b.tunnel.example.net" A +short >/dev/null; done
/usr/local/sbin/dnstap-json-inspect.sh | head -20          # ② 层：唯一 FQDN 数
# ① 层：命中率是否下滑、递归数是否抬升
unbound-control -c /etc/unbound/recursor.conf stats_noreset | grep -E 'total.num.cachehits|total.num.cachemiss'
bpftrace -l 'kprobe:udp*' | head -5                        # ③ 层：按 PID 归因有发包
```

为什么这么设：80 个互不相同、标签高熵的子域不改变总 qps 多少，却把「按域唯一 FQDN 数」抬高——这正是第 29 章流量分析比 payload 分析更通用的原因。①层只会有轻微命中率下滑；若①毫无变化而告警全响在②层，说明分层是对的。
怎么验证：巡检脚本①里 `tunnel.example.net` 计数应接近 80；表中 #3 应进入 pending/firing；「隧道可疑度」面板在同一 UTC 时间窗出现凸起。
失败模式 / 回滚点：consumer 若做了采样，唯一 FQDN 数会明显小于 80——这正是反对采集端采样的实证；NXDOMAIN 会被负缓存按 SOA minimum 缓存（第 7 章），演练后刷缓存并把该域加进**演练白名单**而非永久白名单。

31.7.3 注入 3：刷掉 nft counter

```
# 记录基线（只读）
nft list ruleset | grep -i -E 'counter|packets' | head -20 | tee /tmp/nft-before.txt
nft reset counters                          # 注入：清零计数，会破坏基线，故意做
dig @192.0.2.53 www.example.org A +short >/dev/null   # 制造一次可观测流量
nft list ruleset | grep -i -E 'counter|packets' | head -20 | tee /tmp/nft-after.txt
diff -u /tmp/nft-before.txt /tmp/nft-after.txt || true
```

为什么这么设：这一条演练的是**观测本身被重置**。`nft reset counters` 是第 14 章明确警告过的动作，清零后所有基于「累计计数差分」的判断短暂失真；③层若没有独立帧计数或 eBPF 计数兜底，就会出现「明明有流量，计数器看起来没动」的假象。
怎么验证：`diff` 应显示计数清零后重新累积；同时①层 `total.num.queries` 仍递增、②层仍能查到 `www.example.org`，证明另外两条链路不受 nft 计数重置影响。

失败模式 / 回滚点：生产环境不要随意 `reset counters`；演练前 `nft list ruleset > /etc/nftables.conf.bak`（第 14 章），恢复用整体 `nft -f` 而非逐条删。临时文件 `/tmp/nft-*.txt`、`/tmp/drill-anchor.txt` 演练后删除。

31.7.4　验收矩阵

注入	①应用内	②查询流	③内核 / 网络	合格标准
关一个上游	超时数、SERVFAIL 抬升	出向消息有去无回	出向包与 mark 可见	能定位到出向链路，不是「重启再试」
高熵 QNAME	命中率轻微下滑	唯一 FQDN 数≈ 80、长标签候选	按 PID 归因有发包	隧道可疑度告警可复现，白名单不误吞
刷 nft counter	查询计数仍递增	QNAME 明细仍在	计数清零后重新累积	另外两层不受影响，能识别「观测被重置」

三项都通过，才可以说这套栈「能观测」；只通过一项，说明装的是仪表盘，不是可观测性。演练记录连同 UTC 锚点、阈值调整依据、白名单变更归档，作为下一轮 7 天基线的输入。

31.8　与其他章的交叉引用

- 第 7 章：采集点、`stats_noreset` 纪律、`log-servfail`、dnstap 配置与 **30 天保留期**，本章沿用不重复；第 9 章：`access-control` / `ratelimit` 计数是 unwanted / 限速命中指标来源（官方标 experimental，阈值按基线调）。
- 第 14、22 章：出向收敛、tproxy mark、`meta skuid` 豁免；第 26 章：anycast 集中观测与 per-node 差异落在 `node` / `role` label 与跨主机日志检索。
- 第 28 章：故障域隔离的注入清单；第 29 章：唯一 FQDN 数、长 QNAME / 高熵标签、单源 qps 的定义与误报来源；第 30 章：`_acme-challenge` 等记录是证书续期突发查询的合法来源，白名单按名字形态写。

31.9　本章要点

- **三层互补，不靠一层包打天下**：① `stats_noreset` / `log-servfail` 给累积计数与 SERVFAIL 原因；② dnstap 给 QNAME、单源速率、唯一 FQDN 数；③ eBPF / `ss` / `conntrack` / nft counter 给「包有没有出门、被谁发的、mark 走到哪」。采集点见第 7 章。
- **两条硬边界**：**Unbound 不原生支持 eBPF**，eBPF 是操作系统侧能力、探针名随内核版本变化；**查询日志本身就是全量查询流副本**，接进 Loki / Elasticsearch 后保留期要跟着索引、副本、快照一起对齐 30 天，不把 dnstap 数据无条件推第三方 SaaS。
- 集成形态：`unbound_exporter` / stats → Prometheus → Grafana（qps / SERVFAIL 率 / 命中率 / 上游 RTT / 隧道可疑度 + 采集链路健康 + 日志卷余量），dnstap consumer → JSON → Loki / Elasticsearch →告警，或轻量落文件 + `jq` 巡检；指标名随所选实现而变，只给骨架表达式。
- **告警靠基线与白名单治理**：先测 7 天基线再定阈值；八条规则覆盖 SERVFAIL 率连续 5 分钟超阈、单源 qps、唯一 FQDN 数、上游全超时、DNSSEC bogus 突增、dnstap 断流、磁盘将满、采集器无数据；CDN / 健康探测 / 证书续期会产生合法长 QNAME 与突发查询，白名单要窄、要复审、要有审计（呼应第 29 章）。
- **日志关联的前提是 NTP**：`verbosity` 平时 1、排障临时 2–3 再调回，`log-servfail` 常开；UTC 锚点 + 同源时钟，才能把指标采样、dnstap 事件、nft 计数差分拼成同一条因果链。
- **验收靠注入，不靠声明**：关一个上游、注一批高熵 QNAME、刷掉 nft counter，逐项确认三条链路都产生可辨认信号且互不受牵连；第 26 章 per-node 集中观测与第 28 章注入清单只作交叉引用。

31.10　参考资料

- Unbound 官方手册，Monitoring and Reporting（`extended-statistics`、`statistics-interval`、按需统计）：https://unbound.docs.nlnetlabs.nl/en/latest/topics/core/monitoring.html
- Unbound 官方手册，`unbound-control(8)` 的 `stats_noreset`、`get_option`、`set_option`、`verbosity`、`list_forwards`：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound-control.html
- Unbound 官方手册，`unbound.conf(5)`（`verbosity`、`log-servfail`、`dnstap` 各节；配置块见第 7 章）：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html
- dnstap 官方站点（消息类型、帧格式与字段定义以此为准，本书「四类」为归纳）：https://dnstap.info/

- `unbound_exporter` 社区实现（stats → `/metrics` 的一种形态，指标名以实际输出为准）：https://github.com/kumina/unbound_exporter
- Prometheus 官方文档，Alerting rules（`for`、`expr`、规则文件校验）：https://prometheus.io/docs/prometheus/latest/configuration/alerting_rules/
- Grafana Loki 官方文档（结构化日志侧聚合、保留期与 LogQL）：https://grafana.com/docs/loki/latest/
- bpftrace 官方参考手册（探针列表、内置变量 `pid` / `uid` / `comm`、内核版本差异）：https://github.com/iovisor/bpftrace/blob/master/docs/reference_guide.md

第三十二章

代理与过渡场景里的 DNS

本书讨论的是 DNS 协议机制与失效边界，不构成绕过网络管理的操作指南。

本章涉及的代理、TUN、DNS64/NAT64、464XLAT 等场景，只作为「DNS 应答形态发生变化」的研究对象出现：看一个名字在哪一步被换成地址、被谁换、换出来的地址能不能真的被网络转发。文中不提供任何出口配置、订阅、账号或连通性规避步骤，也不列可用的公共服务清单；所有地址与域名都是教学用文档值——IPv4 用 [RFC 5737] 的 `192.0.2.0/24`、`198.51.100.0/24`、`203.0.113.0/24`，IPv6 用 `2001:db8::/32` 与 NAT64 知名前缀 `64:ff9b::/96`，域名用 `example.net` / `example.org` / `home.arpa.`，主机名一律不指向真实环境。

在这些场景里，DNS 应答不再是「目标的真实地址」。有时它是本地的占位票据，有时它是网关合成出来的可路由地址，有时它干脆被跳过——应用拿着一个 IPv4 字面量直接发包。判断一套方案能不能工作，只需看三件事：应答在哪一层被改写、改写者是否还掌握原始域名、改写后的地址是否真的能被转发。第 3 章的四种角色（存根解析器 stub resolver、缓存解析器、转发器、递归解析器）在这里仍是分析骨架，只是每多一层中间设备，就多一个可以改写应答的位置。

32.1 Fake-IP：把域名解析推迟到连接建立时

Fake-IP 不是标准，没有 RFC，是代理客户端在工程上发明的技巧。它改动的是 DNS 应答的内容：本该返回真实 A 记录，却返回一个本地保留段里的假地址。

32.1.1 完整数据流

```
1. 应用向本地 DNS 模块查询 api.example.net 的 A 记录
   → 立刻回一个 198.18.0.0/15 里的假地址，例如 198.18.0.10
   → 同时在内存里记下映射：198.18.0.10 ↔ api.example.net
2. 应用对 198.18.0.10:443 发起 connect
   → TUN / 透明代理按目的地址截获这个包，反查映射表得到 api.example.net
3. 代理拿域名（不是那个假地址）去匹配域名规则集
   → 命中后由出口按规则发起真实连接，真实 DNS 解析在出口侧才发生
4. 应答沿原路回到应用；应用自始至终只见过 198.18.0.10
```

`198.18.0.0/15` 是 [RFC 2544] 的基准测试保留段，这里只作协议示例，不指向任何具体产品配置；IPv6 侧的对应物可以用 `2001:db8:1::/48` 一类文档前缀示意。选保留段的原因很实际：它不会与任何真实可达的公网地址冲突，落到路由表里也不会被误送到别的主机。

32.1.2 为什么存在

透明代理和 TUN 只能看到 IP 包。如果客户端先做真实解析再发包，代理拿到的是目标 IP，原始域名已经丢了；可分流规则偏偏是按域名写的（第 14 章的 `route.rules`）。要么让应用把域名显式交给代理——HTTP 绝对 URI、socks 一类带名字的握手——要么就干脆别在客户端解析。Fake-IP 选了后者：用一个保留段地址当占位符，把「域名→ IP」这一步推迟到连接建立的那一刻，由同时持有映射表的代理来还原。

于是 DNS 应答在这里不是「目标地址」，而是「一张票据」。这也决定了它的信任模型：映射表和规则匹配必须在同一个进程或同一台设备上，否则反查不到域名，占位地址立刻变成黑洞。Fake-IP 因此天然和 TUN 绑定，它不是「递归解析器上

可以打开的某个开关」——第 3 章的四种角色里，这里的本地 DNS 模块扮演的是存根加缓存的混合体，而真实递归发生在出口侧。

32.1.3　映射表的生命周期

占位地址只在映射表存在时才有意义，因此这张表的寿命直接决定整套机制的可靠性。它至少要覆盖三段：DNS 应答发出、应用发起 connect、代理完成规则匹配并接管连接。任何一段提前过期，反查就会失败。

几个和 DNS 缓存不同的性质值得单独记：

- **不按 TTL 淘汰**。 DNS TTL 描述记录可以缓存多久，映射表描述的是「哪个名字对应这次连接」，两者没有换算关系。把 DNS TTL 直接当映射寿命，短 TTL 域名会立刻黑洞。
- **重启即丢**。映射表通常在内存里，代理进程重启后旧的占位地址全部失去含义；客户端若还握着连接池里的旧地址，会持续失败到连接重建。
- **一对多与多对一都麻烦**。同一域名多次查询可能分到不同占位地址，同一占位地址也可能在回收后被另一个名字复用；后者更危险，反查出的域名是错的，规则匹配会走到完全不该走的出口。
- **IPv6 侧同理**。用 `2001:db8:1::/48` 一类文档前缀做 IPv6 占位时，映射表要同时维护 A 与 AAAA 两套占位；并行连接一类的地址选择机制会让两条路径几乎同时到达代理（具体行为见官方资料）。

32.1.4　失效边界

凡是依赖「拿到的是真实 IP」的行为都会坏。逐条看：

连通性探测。操作系统和浏览器为判断「有没有网」，会向固定名字发查询并期待一个可预期、可达的应答，有的还会检查返回内容。Fake-IP 给的是占位地址，探测方拿它去连 80/443，得到的响应与预期不符，于是误判为断网，甚至触发 captive portal 弹窗。这类名字必须排除。

服务发现。 mDNS/DNS-SD（第 27 章）、SRV 与 SVCB 记录里的目标名、以及「先解析、再对结果做二次查询」的逻辑，都假定 A/AAAA 里是真实地址。占位地址会让后续步骤连到不存在的主机，或让发现结果在局域网里根本传不出去。

内网域。 `home.arpa.`、`.local`、内网单标签名本该由本地解析器直接给出 RFC 1918 地址。给了 `198.18.0.10` 之后，打印机、NAS、网关管理页全部连不上——那个地址不属于内网，也不是任何一台内网主机。

literal IPv4。 URL 里直接写 `192.0.2.10` 时根本没有 DNS 查询，映射表无从建立，Fake-IP 完全不参与。这条流量要么按 IP 规则走，要么直连，域名规则集对它无效。

UDP 与 QUIC。无连接的 UDP 包没有「建连」这个反查锚点；短生命周期会话、QUIC 把连接迁移到新地址的路径，都容易让映射在需要时已经过期或不匹配。连接跟踪超时一过，反查就失败，而且失败表现为静默丢包，很难从应用侧看出来。

PTR 反查。需要反向解析的协议拿 `198.18.0.10` 去查对应的 `in-addr.arpa` 反向域，本地解析器没有权威数据，返回 NXDOMAIN 或被上游拒绝；依赖 PTR 结果做校验的服务直接失败。

还有两类较隐蔽的：应用自己的地址缓存与 HTTP 连接池会把占位地址复用很久，映射表过期后仍被使用；证书校验要求 SNI 与证书名一致，而占位地址本身不携带名字，一旦代理在还原域名前就完成握手，TLS 会失败。

32.1.5　失效边界速查表

失效场景	为什么坏	可观测症状	处理方式
连通性探测 / captive portal	期待可预期应答与可达地址，得到占位地址	误报「无网络」、反复弹门户页	加入 `fake-ip-filter`
服务发现（mDNS/DNS-SD、SRV、SVCB）	假定 A/AAAA 是真实可达地址	发现结果为空或指向不存在主机	排除内网后缀，走正常解析
内网域与单标签名	`198.18.0.10` 不属于内网任何主机	打印机、NAS、网关管理页连不上	排除 `home.arpa.` / `.local` / `*.lan`
literal IPv4	没有 DNS 查询，映射表无从建立	域名规则完全不生效	按 IP 规则处理，或补域名还原
UDP / QUIC / 连接迁移	无建连锚点，跟踪超时后映射失效	静默丢包，应用侧难以定位	保证映射寿命长于连接寿命并补兜底
PTR 反查	反向域无权威数据	NXDOMAIN，依赖 PTR 校验的服务失败	允许真实解析或配置本地反向区

失效场景	为什么坏	可观测症状	处理方式
地址缓存 / 连接池复用	占位地址被长期复用，映射已过期	间歇性连接失败，重启应用后恢复	让映射寿命与连接跟踪状态同步

表里最后一行说明一个常被低估的约束：Fake-IP 的映射表不是 DNS 缓存，它不能靠 TTL 自然淘汰，必须和连接跟踪状态同步。映射比连接先消失，就会出现「明明解析过、却连不上」的假故障——这类假故障最容易被误判成出口不通。

32.1.6 fake-ip-filter 与本书内的交叉引用

因此实现都会带一个排除清单，常写作 `fake-ip-filter`：把连通性检查域、`*.lan`、`.local`、`home.arpa.`、内网后缀，以及明确要按真实地址访问的名字排除，让它们走正常解析。要理解它的定位——这是「缩小占位范围」，不是「修好占位地址」。排除项之外的名字依然享受 Fake-IP 带来的域名保留，也依然承担上面全部失效边界。

本书已有对应物在此只作交叉引用，不展开任何配置：`dns-fake6` 入站、`fakeip` server（`198.18.0.0/15` 加 `2001:db8:1::/48` 一类文档段）、第 3 章的四种角色、第 14 章的域名规则与 `route.rules`。第 12 章的 DoH 端点与本节无关：Fake-IP 处理的是「应答内容被替换」，不是「应答怎么送」，两者处在不同层，可以同时存在也可以完全不相干。

32.2 DNS64 / NAT64 / 464XLAT: 纯 IPv6 网络访问 IPv4 目标

这一族标准解决的是另一个问题：接入网只有 IPv6，目标却只有 IPv4。它同样改写 DNS 应答，但改写出来的地址是**真的能路由**的。

32.2.1 标准族与各自职责

组件	标准	做什么
DNS64	[RFC 6147]	客户端问 AAAA、权威侧只有 A 时，把 A 里的 IPv4 地址嵌入一个 IPv6 前缀，**合成** AAAA 返回
NAT64	[RFC 6146]	网关收到发往合成地址的包，抽出内嵌的 v4 地址，做**状态式** v6 → v4 报文头转换与地址/端口映射
SIIT	[RFC 2765] 一类	无状态报文头翻译基础，为前两者提供翻译规则
464XLAT / CLAT	[RFC 8683]	给只认 IPv4 字面量、或只调用 v4 socket API 的应用，在客户端侧再补一层本地翻译
DNS proxy	[RFC 5625]	规定 CPE / CLAT 上的 DNS 代理行为，包括哪些查询该转给谁

```
查询:  api.example.net  AAAA
权威:  只有 A = 192.0.2.10，没有 AAAA
DNS64: 用 NAT64 知名前缀 64:ff9b::/96 与 192.0.2.10 合成
       → AAAA = 64:ff9b::c000:20a（等价嵌入写法 64:ff9b::192.0.2.10）
客户端: 向 64:ff9b::c000:20a 发 IPv6 包
NAT64: 抽出 192.0.2.10，建立状态映射，转成 v4 包继续转发
```

`64:ff9b::/96` 是 NAT64 知名前缀；网络也可以使用自己分配的前缀，此时客户端需要前缀发现机制。前缀定义、可用范围与发现方式见官方资料，本书不展开。注意合成是有条件的：只有「查了 AAAA、且权威侧确实没有 AAAA」时才发生，名字本身有原生 AAAA 就不合成。

32.2.2 合成成立的三个前提

把上面的数据流拆开，DNS64 要真正生效，必须同时满足三件事，缺一件就退化成「纯 v6 网络访问不了 v4-only 目标」：

1. **客户端问的是 AAAA**。只问 A、或应用直接用 IPv4 字面量，合成不会发生。
2. **权威侧确实只有 A**。名字本身有原生 AAAA 时，DNS64 应原样返回，不能覆盖；否则会破坏已有的 IPv6 路径。

3. **查询确实经过那台做合成的解析器**。解析器被绕过，或下发的解析器自己不合成，前两条再成立也没用。

三条分别对应后面六个坑里的第 2、4、6 条。把它们列在前面，是为了让排查顺序固定下来：先看查询有没有到、再看名字有没有要求 AAAA、最后才怀疑合成与翻译本身。多数现场故障其实停在第一条，却被当成 NAT64 网关的配置问题查了很久。

32.2.3　DNSSEC 与合成应答的冲突

这是 [RFC 8683] 明列的第一个坑。DNSSEC 的 RRSIG 覆盖的是原始应答数据；DNS64 在网络侧把「无 AAAA」改成了「有这样一个合成 AAAA」，签名却还是原来那一份。安全感知的验证器自己验签，会把合成应答判为 bogus，进而 SERVFAIL 或直接丢弃。第 10 章的三种验证状态在这里有了新含义：验证器越严格，合成应答越活不下来；而关掉验证换来的「能通」，代价是把第 16 章讲的那套来源认证整个让出去。

[RFC 6147] §5.5 一类条文给出的正解是：**安全感知主机应在本地做 DNS64**——先完成 DNSSEC 验证、拿到可信的 A 记录，再由本机合成 AAAA。这样验证器验的是原始签名数据，合成发生在验证之后，两者不再作用于同一段字节。呼应全书那条区分：DNSSEC 管数据完整性与来源认证，DNS64 管可达性；它们作用在不同层，不能互相替代，也不能叠加在同一段被改写的应答上。客户端看到 AD 位并不代表安全，如果网络侧有人改写了应答，AD 位本身也失去意义。

32.2.4　加密 DNS 让网络侧 DNS64 失效

客户端用 DoT 或 DoH 直连外部解析器时，查询根本不经过网络里那台做合成的 DNS64 设备，于是拿不到合成 AAAA。客户端按普通 IPv6 语义理解为「这个目标没有 IPv6 地址」，在纯 v6 接入网里直接失败——NAT64 一次都没被调用。[RFC 8683] 明确把 DoT/DoH 场景列为 out of scope，不是遗漏，是承认这两类设计目标相撞：加密传输保护链路隐私，同时也把「网络侧基于 DNS 的策略」一起绕过了。

这和前面 DoT/DoH 绕过网络侧策略属于同一类信任模型问题，可以并起来记：**谁离查询最近，谁才有机会改写应答**。客户端把查询送到外部加密端点，网络侧的 DNS64、过滤表、本地策略就同时出局。第 12、13 章讨论的「加密 DNS 与本地策略冲突」，在纯 v6 网络里不是策略失效那么温和，而是直接表现为连通性归零。

32.2.5　知名前缀与地址混配

有公开文档的 DNS64 服务使用 `64:ff9b::/96`，并要求客户端只配置它自己的那两个 IPv6 解析器地址；把 IPv4 解析器地址混进去，客户端会时而在支持合成的路径上解析、时而在不支持合成的路径上解析，合成 AAAA 时有时无，症状是「同一个名字一会儿能通、一会儿超时」，极像链路抖动，实际是解析结果不一致。这里只作协议示例，不写成任何服务清单：要点是「一台客户端的所有解析器必须落在同一套合成策略里」，以及「合成前缀必须与 NAT64 网关的前缀一致」——前缀不一致时，包会被路由到别处，NAT64 收不到。

32.2.6　IPv4 字面量不走 DNS

DNS64 只对「走了 DNS 的名字」有效。应用里写死 `192.0.2.10`、用 IP 字面量构造的 URL、或只调用 v4 socket API 的程序，没有 AAAA 查询，也就没有合成机会；在纯 v6 接入网里这些包无处可去。这正是 CLAT 存在的理由：它在客户端本地把 v4 报文头翻译成 v6，让应用以为自己仍在一个有 IPv4 的网络里。464XLAT 因此是两段的组合——客户端侧 CLAT 做无状态翻译，网络侧 NAT64 做状态式翻译，[RFC 8683] 描述这一架构，[RFC 2765] 一类 SIIT 提供报文头翻译基础。注意这恰好是 Fake-IP 那一节「literal IPv4」边界的镜像：两者都因为「没有 DNS 查询」而失去改写机会，只是一个补翻译、一个补规则。

32.2.7　CLAT 的查询翻倍

CLAT 不该把每个查询原样转发两遍。按 [RFC 5625] 的 DNS proxy 指引，它应只把 IPv6 原生查询与 AAAA 查询交给网络的 DNS64 解析器；如果 A 查询也照发、或为同一个名字额外向上游发起 A 与 AAAA 两次查询，查询量直接翻倍。后果不只是带宽：缓存命中率下降，上游超时时的负缓存放大（第 7 章）也跟着翻倍，故障恢复后客户端仍会拿到过期的空应答。判断一个 CLAT 实现是否合格，只看它在「名字只有 A」和「名字有原生 AAAA」两种情况下各向上游发了几个查询就够了。

32.2.8　RDNSS 下发的解析器自己不做 DNS64

第 19 章主张用 RA/RDNSS 给纯 v6 客户端下发自己的 ULA 解析器地址，并压掉上级广告的 DNS。可达性做对了，功能却可能没做对：如果这个解析器只是普通递归解析器、自身不做 DNS64，客户端问 v4-only 名字的 AAAA 只会拿到空应答，合成不发生，NAT64 依然空转。也就是说「客户端用哪个解析器」和「那个解析器会不会合成」是两件必须同时

成立的事。第 19 章解决前者，本节提醒后者；RA/RDNSS 的条文原理见第 18 章。把这一条和「DoT/DoH 绕过」并排看会更有意思：一个是解析器选对了但没能力，一个是解析器有能力但被绕过，症状都是「纯 v6 网络访问不了 v4-only 目标」。

32.2.9 六个坑速查表

#	坑	机理	症状	结构性对策
1	DNSSEC 冲突	网络侧改写应答，RRSIG 覆盖的是原始数据	验证器判 bogus → SERVFAIL	安全感知主机本地做 DNS64（[RFC 6147] §5.5 一类）
2	DoT / DoH 绕过	查询不经过网络侧 DNS64 设备	拿不到合成 AAAA，NAT64 空转	统一解析器来源；[RFC 8683] 明列 out of scope
3	前缀与地址混配	部分解析走合成、部分走普通路径	同一名字时通时断，像链路抖动	客户端只配同一套合成策略里的解析器；前缀与网关一致
4	IPv4 字面量	没有 DNS 查询就没有合成机会	v4-only 应用在纯 v6 网里直接失败	CLAT 在客户端侧做本地翻译
5	CLAT 查询翻倍	A 与 AAAA 一并转发、重复向上游查询	缓存命中率下降、负缓存放大翻倍	按 [RFC 5625] 只送 v6 原生查询与 AAAA 给 DNS64
6	RDNSS 解析器不合成	可达性对了，功能没对	客户端拿到空 AAAA 应答	下发的解析器自身需具备 DNS64 能力（第 19 章）

第 1 条和第 2 条方向相反，值得并排记：一个因为「验证太严」而失败，一个因为「离查询太远」而失败。两者都说明同一件事——**合成必须发生在验证之后、且必须发生在查询真正经过的那台解析器上**。这也回到本章反复出现的那条主线：谁离查询最近，谁才有机会改写应答。

32.3 域名还原：当 DNS 已经不给出名字时

Fake-IP 覆盖不到的路径——literal IPv4、UDP/QUIC、连接跟踪超时后被排除的名字、被 `fake-ip-filter` 放行的名字——有个共同点：代理收到包时只有 IP，没有名字。于是出现三类把名字「找回来」的机理。本节只归纳为什么需要它们以及各自的边界，具体规则与实例在第 14 章。

32.3.1 sniff：从协议首部里读名字

TLS 的 SNI 与明文 HTTP 的 `Host` 首部在连接建立时就会带出域名，代理可以在完全不改动 DNS 的前提下，从首部还原出名字再拿去匹配规则。边界很清楚：只对首部明文可见的流量有效，加密 ClientHello 一类机制会封住这个入口（见官方资料），非 HTTP/TLS 协议根本读不到；而且 sniff 是启发式的，读不到时必须有兜底行为——放行还是拒绝，这个兜底本身就是策略，属于第 5、14 章反复出现的 fail-open / fail-closed 取舍。

32.3.2 resolve：从缓存反查

如果本地缓存里存着「域名→ IP」的映射，代理可以拿目的 IP 反查缓存得到域名。它比 sniff 便宜，不用等首部；但只在缓存未过期、且该 IP 确实来自一次本地可观测的查询时成立。CDN 的共享地址、anycast（第 26 章）、一对多映射都会让反查变成多义。更要紧的是语义：反查到的是「曾经解析过这个名字」，不是「这次连接一定属于这个名字」。所以它能当规则匹配的提示，不能当审计证据。

32.3.3 规则集与 ip_version 固定出口

按域名规则集分流（第 14 章 `route.rules`）解决「同一个名字走哪条路」；`ip_version` 一类参数解决「这次解析要 A 还是 AAAA」，避免出现「规则说走 v6、解析器却给了 v4」的错配。在纯 v6 + NAT64 的场景里，这个开关还决定客户端会不会去问 AAAA——不问就没有合成，合成不做 NAT64 就空转。三者合起来看：域名还原不是 DNS 的替身，而是「DNS 已经把名字丢了」之后的补偿机制；补偿用得越多，说明上游的 DNS 形态越不适合这套代理架构。

32.4　Fake-IP 与 DNS64 的本质区别

32.4.1　本地占位与可路由合成地址

Fake-IP 给出的 `198.18.0.10` 在公网和内网都不存在对应主机：它只在本地映射表里有意义，必须经 TUN / 透明代理反查成域名才能用，一旦离开那台设备就是黑洞。DNS64 合成的 `64:ff9b::c000:20a` 是真的能路由的：网络里有 NAT64 设备为这个前缀准备了状态映射，任何不参与这套代理的普通 IPv6 主机拿到它，一样能发包并到达目标。一个是**本地占位票据**，一个是**可路由合成地址**。

由此推出三条不同的性质：

- **谁做决定**：Fake-IP 由本地代理决定（映射表 + 域名规则集），DNS64 由网络侧解析器与网关决定（合成策略 + 翻译状态）。前者是端点行为，后者是网络行为。
- **对 DNSSEC 的影响**：Fake-IP 替换的是本地应答，压根不进验证器；DNS64 在网络侧改写应答，会和验证器正面冲突。
- **失败模式**：Fake-IP 的失败是「反查不到域名」，表现为黑洞或误判断网，排查看映射表与连接跟踪；DNS64 的失败是「拿不到合成 AAAA」，表现为超时或「无 IPv6 地址」，排查看解析器与前缀是否一致。

两者目的完全不同，也不是互为备选：一个服务于「按域名分流」，一个服务于「纯 v6 网络访问 v4 目标」。同一个名字理论上可以先后经过两种变形（本地占位→出口侧真实解析→出口网络恰好是 NAT64 环境），但每一步的改写者、地址性质与失效点都要分开算，不能因为「都改了 DNS 应答」就归成一类。

32.4.2　排查顺序：先定位改写者

现场遇到「代理/过渡环境里 DNS 不对」时，按下面顺序问，能省掉大半无效排查：

1. 这次流量到底有没有 DNS 查询？没有（IPv4 字面量、缓存命中、连接池复用），所有基于应答改写的机制都不参与。
2. 查询送到了哪台解析器？是本机存根、网关转发器、RDNSS 下发的解析器，还是应用自己选定的外部端点？这一步决定谁有机会改写。
3. 它返回的 A / AAAA 是真实地址、本地占位、还是合成地址？看地址段就能分：`198.18.0.0/15` 一类是占位，`64:ff9b::/96` 一类是合成，其余按真实应答处理。
4. 那个地址在转发路径上有人接吗？占位要代理反查，合成要 NAT64 状态映射，两者缺一即黑洞。
5. 规则匹配用的是域名还是 IP？域名还原失败时，兜底是放行还是拒绝（第 5、14 章的 fail-open / fail-closed 取舍）。

第 2 步最容易漏。管理员盯着自己那台递归解析器的日志，客户端却早把查询送到了别处——日志里什么都没有，恰恰就是答案。

32.4.3　场景总表

场景	DNS 形态	谁做决定	失效点	本书章节
普通递归解析	真实 A / AAAA 应答	递归解析器	上游不可达、负缓存放大	第 3、4、6 章
DoH（自建端点）	二进制 DNS 报文套 HTTP/2	配置里的 URI Template + 端点解析器	端点运营者可见全部查询；HTTP 缓存语义	第 12、17 章
Fake-IP	`198.18.0.0/15` 本地占位	本地代理的映射表 + 域名规则集	探测 / 服务发现 / 内网域 / literal / UDP / QUIC / PTR	第 14 章
DNS64 + NAT64	合成 AAAA（`64:ff9b::/96` + v4）	网络侧 DNS64 解析器 + NAT64 网关	DNSSEC、DoT/DoH 绕过、前缀混配、查询翻倍	第 18、19 章
464XLAT / CLAT	v4 应用本地翻译 + 网络侧合成	CLAT 侧 DNS proxy + 网络 DNS64	literal IPv4、查询翻倍、RDNSS 解析器不合成	第 19 章
域名还原	DNS 已给出 IP，名字从首部或缓存补回	代理的启发式匹配	首部加密、缓存多义、兜底策略	第 14 章

表里最后两列是排查顺序的提示：先定位「谁做决定」，再对照「失效点」。多数现场问题不是协议没实现，而是改写者被换掉了——客户端改用了别的解析器，或网络侧的策略根本没被调用到。

32.5 SSH / SOCKS 通道里，DNS 到底归谁解析

代理场景里最容易漏的一层是：**`ssh -D` 起的 SOCKS 监听并不自动接管系统 DNS**。应用把目标名字交给谁解析，决定了查询是留在本机还是走通道——这就是「DNS 泄漏」的分界。

一类客户端把这层选择写成一个开关，常见形态是 **`resolvdns yes/no` 式的 DNS 归属设置**：`yes` 把主机名直接交给对端或代理去解析，`no` 则由本机先解析成 IP 再把连接交给通道。本书按语义描述这类开关，**不把它归为 OpenSSH `ssh_config` 的原生选项**——OpenSSH 里没有同名配置项，具体名称与归属以所用客户端文档为准。公开可查的等价能力分散在几处：

实现	形式	语义
PuTTY	“Do DNS name lookup at proxy end” = Yes / No / Auto	Yes 传主机名；No 本地解析后传 IP；Auto 下 HTTP/SOCKS5 传主机名、SOCKS4 不传
Firefox	`network.proxy.socks_remote_dns`	让 SOCKS v5 代为解析
curl	`--socks5-hostname` / `socks5h://`	远程解析；`--socks5` / `socks5://` 是本地解析
sshuttle	`--dns` / `--to-ns`	捕获本机 53 查询并转发到远端解析器

判读方法：让客户端访问一个你控制的全新名字，同时在两端分别看 DNS 日志——只有「本机无查询、对端有查询」才能证明名字确实交给了通道，仅凭出口 IP 变化说明不了 DNS 归属。这与第 29 章「客户端绕过」、第 36 章的工具箱是同一条线索：**先分清是哪一层在解析，再谈通不通**。

32.6 本章要点

1. DoH 是「同一份 DNS wire format + 加密传输」，信任从链路转移到端点运营者，本机策略仍可完整保留；端点一旦不在自己手里，本机过滤表、DNSSEC 验证与 dnstap 观测就同时出局。判断三件事即可：应答在哪一层被改写、改写者是否还掌握原始域名、改写后的地址是否真的可路由。
2. Fake-IP 用 `198.18.0.0/15` 的本地占位地址把「域名→ IP」推迟到连接建立时，让 TUN / 透明代理还能按域名分流；它不可路由、必须反查，连通性探测、服务发现、内网域、IPv4 字面量、UDP/QUIC、PTR 都是失效边界，靠 `fake-ip-filter` 缩小范围而非修好地址。
3. DNS64 [RFC 6147] 合成 AAAA、NAT64 [RFC 6146] 做状态式转换、CLAT [RFC 8683] 补 IPv4 字面量、DNS proxy [RFC 5625] 约束转发行为；六个坑是 DNSSEC 冲突（安全感知主机应本地做 DNS64）、DoT/DoH 绕过网络侧合成、知名前缀与 v4 地址不能混配、literal IPv4 不走 DNS、CLAT 查询翻倍、RDNSS 下发的解析器自身不做合成。
4. sniff（SNI / HTTP Host）、resolve（缓存反查）、域名规则集与 `ip_version` 固定出口，都是「DNS 已经不给名字」之后的补偿机理，不是 DNS 的替身；补偿越多，说明上游 DNS 形态越不适合这套代理架构。
5. 判断任何代理或过渡方案只需三问：应答在哪一层被改写、改写者是否还掌握原始域名、改写后的地址是否真的可路由。Fake-IP 是本地占位（不可路由、必须反查），DNS64 是可路由合成地址，二者目的不同，不能互换。
6. 加密传输（DoT/DoH/DoQ）与 DNSSEC 不互相替代：前者管链路隐私，后者管数据完整性与来源认证；DNS64 在网络侧改写应答，破坏的正是后者。

32.7 参考资料

- RFC 8484, *DNS Queries over HTTPS (DoH)* —https://www.rfc-editor.org/rfc/rfc8484
- RFC 6570, *URI Template* —https://www.rfc-editor.org/rfc/rfc6570
- RFC 6147, *DNS64: DNS Extensions for Network Address Translation from IPv6 Clients to IPv4 Servers* —https://www.rfc-editor.org/rfc/rfc6147
- RFC 6146, *Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers* —https://www.rfc-editor.org/rfc/rfc6146
- RFC 8683（464XLAT / CLAT 与 DNS64 交互的适用性说明，标题与年份见官方资料）—https://www.rfc-editor.org/rfc/rfc8683

- RFC 5625, *DNS Proxy Implementation Guidelines* —https://www.rfc-editor.org/rfc/rfc5625
- RFC 5737, *IPv4 Address Blocks Reserved for Documentation* —https://www.rfc-editor.org/rfc/rfc5737
- RFC 2544（基准测试方法论，`198.18.0.0/15` 保留段出处）—https://www.rfc-editor.org/rfc/rfc2544

第三十三章

DHCP 与 DNS 的交接面

一台客户端打开网页时先问谁，不是 `/etc/resolv.conf` 自己决定的。地址、掩码、网关、以及”把 DNS 查询交给谁”，绝大多数情况下是同一个 DHCP 应答一次性写进去的。于是 DNS 的**输入侧**并不在递归解析器手里，而在 DHCP 服务器手里：option 6 填错一个地址，Unbound 配得再严也没有流量进来；option 15/119 多塞一个后缀，根服务器就要替你收一批无效查询；option 81 与 option 42 更是从名字和时钟两个方向决定了动态更新能不能成立。**交接面**指的就是这一段：配置从 DHCP 落到存根解析器，再落到 DNS 行为的全过程。

这一章做 DHCPv4 / DHCPv6 的 option 工程。RA / RDNSS / DNSSL 的条文与实现差异见第 18 章，家庭网关”广告自己 ULA 解析器、压制上级 DNS”的部署做法见第 19 章，WPAD 草案的五种发现机制与 devolution 安全史见第 23 章，微软 DNS 与 DDNS 的更新模型见第 25 章。

33.1 一条配置要穿过四层

“当前生效的 DNS 是谁”不是一个字段，而是四层叠加后的结果。排查时最容易犯的错，就是只看其中一层就下结论：DHCP 服务器明明写对了，客户端却仍在查别处，往往是被更靠上的一层覆盖了。

层	写入者	覆盖关系	观测点
固件 / 镜像默认值	设备厂商	最底层，仅在无人下发时生效；常常塞进一个自造后缀	恢复出厂后立刻抓一次 `resolvectl domain`
DHCPv4 / DHCPv6 租约	网关或独立 DHCP 服务器	覆盖默认值，但按 T1/T2 刷新，不即时生效	租约文件、`resolvectl dns`
RA 的 RDNSS / DNSSL	链路上的任一路由器	与 DHCPv6 并行，谁赢取决于主机实现	`rdisc6` 一类工具按名称在官方资料检索
每链路 / 每应用覆盖	VPN、浏览器 DoH、容器运行时	优先级最高，能把前三层全部绕开	第 13 章客户端绕过、第 32 章加密传输选路

第四层要特别标出来：即便 DHCP 侧做到完美，一个自带 DoH 端点的浏览器仍可以把查询直接送出内网（第 13 章的 DDR 压制与 canary 域就是在处理这件事）；第 32 章讲的 DNS64 / Fake-IP 也都建立在”客户端确实把查询交给本地解析器”这个前提上。**交接面只负责把流量交到本地解析器，交到之后不被绕走是第三篇与第 32 章的事**，两段的责任要分清楚，否则故障归因会来回甩锅。

一个实用的判据：先问”这台主机此刻把查询发给了谁”，再问”为什么会发给他”。前者用 `resolvectl dns` 或抓包一次性回答，后者才需要沿着这张表逐层排除。把两件事混在一起，是本章多数排障时间被浪费的原因。

33.2 一张表：option 编号→ DNS 行为→症状→验证

左半边是”这个 option 是什么、出自哪份标准”，右半边是运维真正关心的三件事：它改动了哪一个 DNS 行为、配错之后在客户端上看到什么、以及用哪条命令能当场确认。

Option	名称	作用	标准	影响哪个 DNS 行为	配错的症状	验证命令
6	Domain Name Server	下发一个或多个递归解析器地址	[RFC 2132]; [RFC 5625] §5.1	决定存根解析器把查询发给谁	查询全被送到不该去的解析器，本地过滤表与验证策略失效；或 WAN 未通时全部超时	`resolvectl dns`
15	Domain Name	单个默认域后缀	[RFC 2132]; [RFC 5625] §5.2	裸主机名被追加一次后缀	固件默认的 `lan` / `local` 类后缀产生大量无效 TLD 查询打根服务器	`resolvectl domain`
119	Domain Search List	多个搜索域，DNS wire 压缩编码	[RFC 3397] / [RFC 6731]	裸名依次追加多个后缀，与 `ndots` 叠加放大查询	一次访问发出四五次查询，NXDOMAIN 比例升高，日志里全是同一批可疑后缀	`resolvectl domain`
81	Client FQDN	客户端自报 FQDN，附 S/N/O 标志	[RFC 4702]	决定是否触发 [RFC 2136] 动态更新、由谁更新	名字解析不到自己；或 DDNS 记录被非授权名字占据	`dig +short gw-01.home.arpa. @192.0.2.53`
252	WPAD URL	指向代理自动配置文件	1999 年过期草案（非 RFC）	决定 PAC 文件从哪取	代理发现落到组织边界外（详见第 23 章）	`curl -I http://wpad.example.net/wpad.dat`
42	NTP Servers	下发时间服务器	[RFC 2132]	时钟偏差决定 GSS-TSIG 更新、DNSSEC 验签、日志关联是否成立	时钟偏差超过 5 分钟，Kerberos / GSS-TSIG 安全动态更新失败；验证器把有效签名误判为过期	`timedatectl timesync-status`
v6-23	OPTION_DNS_SERVERS	v6 侧递归解析器地址	[RFC 3646]	与 option 6 等价，但走 DHCPv6 通道	与 RA 的 RDNSS 各发一套，主机取谁不确定	`resolvectl dns`
v6-24	OPTION_DOMAIN_LIST	v6 侧域搜索列表	[RFC 3646]	与 option 119 等价	同 15/119 污染，且 v4/v6 两侧后缀还不一致	`resolvectl domain`
v6-56	NTP Server 子选项	v6 侧时间同步	[RFC 5908]	与 option 42 等价	v6-only 客户端无时间源→动态更新与 DNSSEC 一并失效	`chronyc tracking`

Option	名称	作用	标准	影响哪个 DNS 行为	配错的症状	验证命令
v6-25	IA_PD	前缀委派	[RFC 8415] 一类	决定 LAN 侧前缀，间接决定 RDNSS 发布的解析器地址可达性	前缀轮换后旧地址失联（第 18、19 章）	`ip -6 addr show`

表里的编号空间要分清：DHCPv4 的 6/15/119/81/42/252、DHCPv6 的 23/24/56/25、以及 RA 里的 RDNSS / DNSSL 是三套独立编号，各自的选项类型号按 [RFC 8106] 与 IANA 注册表核对（见官方资料）。不同载体里出现相同的数字纯属巧合，彼此没有任何关系；看到”25”先问是哪个载体，别把 IA_PD 和 RA 的 DNS 选项混为一谈。

33.2.1 option 6 / 15 / 119：地址、默认后缀、搜索域

option 6 是唯一真正决定”查询去哪”的字段，可以下发多个地址，顺序即存根解析器的尝试顺序。[RFC 5625] §5.1 指出一个很常见的默认值：家用网关把**自己**填进 option 6，因为它同时是一个 DNS 代理（DNS proxy）；这条做法本身合理，问题在于管理员往往改不掉这个默认值，于是”想让客户端直连本地 Unbound”就无从下手。选服务器时要留一个可改的入口，而不是把网关自身硬编码进固件。

option 15 与 119 都不给地址，只给”后缀”。区别是 15 只能给一个默认域，119 给一个有序列表。119 的载荷不是逗号分隔的明文，而是 DNS wire format 的域名标签序列，允许使用压缩指针（[RFC 3397]）；[RFC 6731] 与该编码在客户端侧的处理相关，具体互操作性结论与位级约束见官方资料，本书不臆造字节序列或版本号。正因为编码特殊，不同客户端对压缩指针的支持并不一致，这也是”能不下发就不下发”的现实理由。

[RFC 5625] §5.2 的措辞值得照抄进运维规范：并不存在标准化的 `local` 后缀，默认域**应当为空**；没有显式配置时，服务器就**不要发** option 15 / 119。第 20 章讲过 `ndots` 的代价——绝对域名也会被反复追加 search 域再查一次——把 119 的列表长度乘上去，就是本章要处理的查询放大。确需本地后缀时，用专门保留的 `home.arpa.`，不要自造 `.lan` / `.local`；后者与 mDNS 的 `.local`（第 27 章）还会撞车。

33.2.2 option 81 / 42：名字与时钟

option 81 让客户端在租约里自报 FQDN，并用几个标志位约定”这次 DNS 更新由谁做”。[RFC 4702] 的三个关键标志：

- **S**：客户端请求服务器代为更新正向（A / AAAA）记录，PTR 由客户端自己负责。
- **N**：客户端明确要求服务器**任何** DNS 更新都不要做；N 与 S 互斥。
- **O**：服务器告知客户端，它覆盖了客户端原本的 S 意图（服务器策略优先）。

其余标志位（含域名编码方式）与精确位序见官方资料。S/N/O 共同决定这条租约会不回落到 [RFC 2136] 的 DNS UPDATE 上，也就是第 25 章 DDNS 的入口。这里只强调一件事：**option 81 是”客户端自报名字”进 DNS 的那道门**，下一节的安全问题全部发生在这道门上。

把 option 42 放进 DNS 章不是凑数。AD 集成 DNS 的安全动态更新走 Kerberos / GSS-TSIG，票据对时钟偏差有硬约束，偏差超过 5 分钟即失败；DNSSEC 的 RRSIG 有明确的生效与过期时间，验证器本机时钟错了会把有效签名判成过期、把尚未生效的签名判成 bogus，最终表现为一片 SERVFAIL（第 10 章）；跨设备排查时，dnstap、DHCP 日志、DNS 日志要能按时间对齐（第 31 章）。DHCPv6 侧的 option 56（[RFC 5908]）同理，v6-only 客户端没有 NTP 源时，动态更新与 DNSSEC 会一起坏，且症状看起来像”DNS 挂了”而不是”表没对上”。

33.2.3 option 252 与 v6 侧的编号空间

option 252 只是把 PAC 文件的 URL 塞进 DHCP 应答，草案从未成为 RFC，也不在 [RFC 2132] 的正式 option 表里，属于 IANA 私有/保留段；五种发现机制、`wpad` 这个名字的归属边界、以及 DNS devolution 越出组织的安全史都在第 23 章，放进 option 体系里占一个位置：它和 15 一样，是”名字/位置”而非”解析器地址”。

DHCPv6 的 23 / 24 ([RFC 3646]) 在语义上就是 v6 版的 6 / 119, 56 ([RFC 5908]) 是 v6 版的 42; 25 (IA_PD) 负责前缀委派，条文见 [RFC 8415] 一类，第 18 章已讲，这里只一句带过。要留意的是：**23/24 与 RA 的 RDNSS/DNSSL 是并行通道**，不是互为备份，下一节专门处理这个冲突。

33.3　RDNSS 与 DHCPv6 的优先级战争

IPv6 侧最反直觉的一点是：DNS 配置有两条完全独立的下发通道。[RFC 8106] 定义的 RA 选项（RDNSS 给地址、DNSSL 给搜索域）由路由器在邻居发现阶段周期性广播；[RFC 3646] 的 DHCPv6 option 23/24 由 DHCPv6 服务器随租约下发。两条通道都合法、都能单独工作，但标准并没有规定客户端同时收到两者时该怎么合。

33.3.1　主机怎么合并两路 DNS

合并策略是实现细节，各系统不一样：有的以 RA 为准、DHCPv6 只作补充；有的把两路地址拼接成一张列表；有的按接口维护”路由域”（routing domain），哪个名字走哪条链路取决于后缀匹配；还有的以后到的消息覆盖先到的。systemd-resolved、Windows、macOS 的具体行为与版本相关，见官方资料，不要凭一次抓包就当成通用结论，也不要把某一次观测写成实现承诺。

更麻烦的是生命周期不同步：RDNSS 带自己的 Lifetime，到期就失效；DHCPv6 的 23/24 跟着租约的 T1/T2 走。于是两边刷新节奏天然错开，即使某一刻恰好一致，下一刻也可能漂移。这也解释了第 19 章为什么强调 hop limit 255、SLLAO、`RDNSS lifetime=0` 这些细节——RA 那条通道是要靠参数**主动管理**的，不是发出去就稳定。

33.3.2　为什么不能两边各发一套

第 19 章的结论是：RA 只广告网关自己的 ULA 解析器地址，并用 `RDNSS lifetime=0` 压掉光猫/上级广播进来的 DNS，从而做到”全屋无污染”——所有查询都经过本地 Unbound 的过滤、验证与缓存。在这个前提下发两套不同地址，等于把这个前提拆掉：

- **结果不确定**。同一间屋子里，A 主机可能取 RA 的 ULA 解析器，B 主机可能取 DHCPv6 23 里的上级解析器，C 主机可能把两者拼接后按某种顺序重试。哪一个赢取决于实现与消息到达顺序，无法通过服务端配置固定。
- **策略被绕过**。凡是落到上级解析器的那部分流量，本地过滤表、DNSSEC 验证、QMIN、加密上游、dnstap 观测全部落空，而且客户端没有任何报错——它只是”能上网”。
- **故障排查失去单一事实来源**。第 5 章反复强调转发器只留一个上游；这里同理，DNS 入口必须只有一个可指认的来源，否则”到底查了谁”要逐台主机抓包才能回答。
- **与搜索域叠加放大**。若 RA 的 DNSSL 与 DHCPv6 24 各给一份后缀，合并后列表更长，配合 `ndots`（第 20 章）放大倍数进一步上升。

还有一层容易被忽略：v4 与 v6 是两套独立的下发通道，双栈主机同时持有 option 6 与 RDNSS/DHCPv6 23。理想状态下它们应当指向同一台本地解析器；若 v4 侧给了 `192.0.2.53`、v6 侧给了另一个地址，那么”同一个域名走哪条协议族”就会变成谁拿到 DNS 答案的决定因素，而选哪条地址又是 RFC 6724 地址选择的事（第 18 章）。症状是间歇性的过滤失效：同一台主机，有时命中本地过滤表，有时没有，和域名本身无关。**双栈环境下，option 6 与 v6 侧来源必须一起看**。

所以工程上的做法不是”两边都配上更保险”，而是**选一条通道作为唯一来源**：按第 19 章用 RA 广告 ULA 解析器，DHCPv6 就干脆不下发 23/24（`dhcpd6.conf` 里注释掉），或者两边填**完全相同**的同一个地址并接受它只是冗余而非独立配置。本章后面的 `dhcpd6.conf` 片段采用前一种。

33.4　租约时序陷阱

[RFC 5625] §5.3 描述的是一台家庭网关开机时最容易踩、又最难复现的问题。WAN 侧还没建立时，网关上的 DHCP 服务器已经开始给 LAN 客户端发租约；此刻它还不知道上游 DNS 是什么，只能把**自己**填进 option 6，靠本机 DNS 代理顶着。等 PPPoE / WAN 侧 DHCP 客户端把链路拉起来、网关从上游学到真正的 DNS 地址后，服务器这边的数据变了，但已经拿到租约的客户端不会主动重问——DHCP 是”服务器推送、客户端按 T1 续期”的模型，配置变更不会反向通知。于是客户端继续把查询发给那个尚未就绪的代理，直到下一次续期才更新。

观察到的现象很有迷惑性：开机后头几十秒到几分钟，`dig` 超时或 SERVFAIL，过一阵子自己就好了；`resolvectl dns` 从头到尾显示的都是同一个网关地址，看不出哪里变了。这正好和”DNS 服务没起来”混淆，容易把人引去重配 Unbound，而真正的问题在租约时序上。

[RFC 5625] §5.3 给出的缓解是**短租约**（例如 60 秒量级），让 T1 尽快到来，把不一致窗口压到可接受范围；代价是 DHCP 与续期流量变多，在小规模 LAN 里通常可以接受。另一个更干净的思路是从设计上消除变更：option 6 始终指向网关自身的 DNS 代理，地址永不改变，上游怎么换都由代理内部消化——这本来就是 [RFC 5625] 讨论 DNS 代理的初衷。两条路可以叠加：短租约兜底，稳定的代理地址治本。

```
# /etc/dhcp/dhcpd.conf 片段：把不一致窗口压短
default-lease-time 60;
max-lease-time 60;
```

- ①为什么这么设：T1 约为租约的一半，60 秒租约让 WAN 就绪后的最长失配窗口落在几十秒量级；仅在启动阶段的过渡网段使用。
- ②怎么验证：`dhcpd -t -cf /etc/dhcp/dhcpd.conf && echo OK`，再到客户端看 `resolvectl dns` 与实际 `dig +short example.org @192.0.2.1` 的恢复时间点。
- ③失败模式 / 回滚点：租约过短会让续期流量与服务器日志明显变多，WAN 抖动时还可能反复触发重配置；回滚为 `default-lease-time 600` / `max-lease-time 3600` 并 reload，客户端会按新值在下一次续期收敛。注意改配置不会立即改写已有租约——这本身就是本节要讲的陷阱。

33.5 option 15 / 119 的污染

厂商固件爱给 LAN 客户端塞一个”看起来贴心”的默认后缀。它可能是自造的 `.lan`、`.local`，也可能是运营商标配的域名。客户端存根解析器拿到后，会把不带点的裸名——以及在第 20 章 `ndots` 语义下**连绝对域名**——逐个追加后缀再查一次。这些后缀绝大多数不是有效 TLD，于是查询一路出网打到根服务器，换回一片 NXDOMAIN。[RFC 5625] §5.2 原文就点名了这类”大量无效 TLD 查询打根服务器”的后果：不是安全事件，是全体递归基础设施替一份没必要的固件默认值买单。

放大系数很容易估：一个 119 列表有 3 个后缀，`ndots` 让一个原本该一次命中的名字先试 3 次带后缀的组合，再加绝对名本身，单次访问就是 4 次查询；若 v4 的 119 与 v6 的 24 又各给一份，列表还能更长。对本地来说是无谓的延迟和日志噪声，对上游来说是被放大的无效负载，对排查来说是一堆同后缀 NXDOMAIN 把真正的故障信号淹掉。

排查顺序建议固定下来：

```
# 1）客户端：看是否被强加了搜索域（systemd-resolved 环境）
resolvectl domain
# 2）看当前生效的上游是不是自己期望的那一个
resolvectl dns
# 3）用一个绝对域名（注意末尾的点）确认解析器本身可用
dig +short example.org. @192.0.2.53
# 4）解析器侧：查 dnstap / 日志里以可疑后缀结尾的 NXDOMAIN 是否成片
```

- ①为什么用这几条：把”DHCP 下发了什么”与”解析器本身能不能用”拆成两个独立判断，避免一上来就改 Unbound 配置。
- ②怎么读输出：第 1 步只要出现非预期的后缀，基本可以断定是 DHCP（或 RA 的 DNSSL）塞进来的；第 3 步带尾点是为了**绕开**搜索域追加。第 4 步的成片 NXDOMAIN 是判定污染范围最直接的信号，可喂给第 31 章的告警规则。
- ③失败模式 / 回滚点：直接删掉 15/119 后，依赖裸名互访的旧脚本会解析不到——这是删后缀最常见的回归。回滚点是保留 `home.arpa.` 这一**个**专用后缀，并把依赖裸名的名字改成显式带点的形式；不要靠再加更多后缀去”补救”。

顺带一提：`.local` 归 mDNS（第 27 章）使用，把它当普通单播后缀追加，查询行为会和组播解析抢同一批名字，症状比单纯的 NXDOMAIN 更乱。

把常见症状与交接面上的具体环节对上，可以少走很多弯路：

症状	优先怀疑的环节	确认动作
开机后短暂全超时，过一会自愈	租约时序（[RFC 5625] §5.3）	看租约剩余时间与 WAN 建立时间是否重叠
本地过滤表对某些主机不生效	多条 DNS 来源或上层覆盖	逐台比对 `resolvectl dns`
成片后缀相同的 NXDOMAIN	15 / 119 / DNSSL 污染	`resolvectl domain` + dnstap QNAME 分布
少数域名间歇性解析失败	双栈两侧来源不一致或 `ndots` 放大	分别用 `-4` / `-6` 各测一次
动态更新失败但 DNS 本身正常	option 42 / 56 时钟、S/N/O 配置	`timedatectl timesync-status`
带尾点的绝对名正常、裸名异常	后缀追加，而非解析器故障	`dig example.org.` 与不带尾点对比

最后一行是最快的一个分水岭：带尾点能通、不带尾点不通，问题一定在后缀追加或搜索域，不必再去查 Unbound 的上游、缓存或 DNSSEC 状态。反过来，带尾点也不通，就与本章的 option 无关了，该回到第二篇的角色配置去查。

33.6　option 81 与动态更新注入的攻防

这一节只写机理与防守，不提供任何伪造报文、构造 option 或注入记录的可复制流程；任何验证都只能在自己有书面授权的隔离实验网段进行（纪律见第 34 章）。

33.6.1　机理止于这一层

微软 DHCP 与 AD 集成 DNS 的“DHCP DNS Dynamic Updates”组合里，DHCP 服务器可以**以自身凭据**代客户端向 DNS 发起 [RFC 2136] 更新，好让客户端不用各自持有写权限。触发这条路径的输入，正是 option 81 里客户端自报的 FQDN。若服务器对”这个名字是否属于该客户端”不做归属校验，就出现一个结构性问题：一个自报的名字可能被写成一条正式的 A / AAAA 记录，落在内网权威区里。

攻击者把名字选成 `wpad` 或能被通配匹配到的形式时，这条记录就与第 23 章的 WPAD 名称解析叠加，形成名称劫持面；这与第 25 章 DDNS、第 29 章暴露面自查是同一条链上的三个观察点。Akamai 2023 的《Weaponizing DHCP DNS Spoofing》与 TrustedSec 2021 的《Injecting Rogue DNS Records Using DHCP》都公开分析过这条机理，本书只引用其公开的结论，不复刻流程、不列工具、不给步骤、不写报文字段序列。

关键在于信任边界：DNS 更新请求看起来来自”可信的 DHCP 服务器”，而服务器只是转述了一个未经证实的客户端自报值。**凭据的可信不等于名字的可信**。这一点也解释了为什么 option 81 不该被当成”只是给客户端起个名字”的普通字段。

33.6.2　防守清单

- **Name Protection / DHCID**：把 DNS 记录与 DHCP 租约绑定，用 DHCID 记录（名称与编号按官方资料检索）标记归属，使别的客户端不能覆盖已有名字。
- **DHCP 专用低权限凭据**：代更新用的账号只授予必要的更新权限，不要用域管理员级凭据；凭据泄露时爆炸半径限定在一小片区。
- **Global Query Blocklist**：Windows DNS 侧的全局查询阻断列表默认包含 `wpad` 等敏感名字，保持启用并按需补充，别为了”让 WPAD 好用”把它关掉（第 23 章）。
- **预置占位记录**：对 `wpad`、`isatap` 一类高价值名字预先建立受控的正式记录，让动态更新无法抢占已存在的名字。
- **DNS 更新 ACL**：区与记录两级限制谁能更新什么，把代更新范围收敛到明确的区与名字集合。
- **交换机端口安全 / DHCP snooping**：只让授权端口能进 DHCP 服务器应答，压缩伪造租约进入的入口（见下一节）。
- **审计与观测**：监视非租约来源的名字更新、敏感名字的新增记录、以及同一名字在短时间里被不同客户端改写；信号进第 31 章告警。

这些措施并不互相替代，它们分别封住四个不同环节：端口安全 / snooping 管”谁能发租约”，Name Protection / DHCID 与更新 ACL 管”哪个名字能写”，低权限凭据管”写坏了能坏到哪里”，Global Query Blocklist 与占位记录管”最敏感的那几个名字根本不能被抢”。只做其中一条，其余环节仍会漏。

时间同步（option 42 / 56）在这里也有防守价值：安全动态更新与 GSS-TSIG 依赖可用时钟，把 NTP 源纳入受控范围，本身就是收紧更新链的一环；反过来，若日志无法按时间对齐，“先有租约还是先有更新”这个关键归因就做不出来。

33.6.3　现象层：不靠攻击复现也能自查

防守方不必也没有必要去复现注入流程，只看结果就够了：

- 权威区里出现与任何已知租约都不对应的 A / AAAA 记录；
- 敏感名字（`wpad` 等）出现了未预期的记录，或占位记录被改写；
- 同一名字短时间内被不同客户端反复更新；
- DNS 更新日志里出现来源与 DHCP 服务器绑定表不一致的条目；
- 一批名字同时指向同一个陌生地址，且 TTL 与正常租约节奏不符。

把这些做成定期比对（租约绑定表↔权威区记录↔ DNS 更新日志）即可，也就是第 29 章资产自查的思路在交接面上的具体化。发现异常先按变更流程回滚占位记录与更新 ACL，不要在不清楚归属前批量删记录。

33.7　rogue DHCP 与 DHCPv6 假 DNS

假 DHCP 服务器应答 DISCOVER / SOLICIT，把 option 6（或 v6 的 23）指向攻击者控制的解析器，是经典的内网劫持：客户端全程无报错，只是查询都被别人看见了，且本地过滤、DNSSEC 验证、加密上游一并被绕过。v6 侧还有一

条更隐蔽的路径——伪造 RA 塞进 RDNSS，效果相同且不依赖 DHCP 服务器存在。第 34 章从对手视角看这条链，本节只落防守，不展开任何伪造或投毒的操作序列。

防守依赖二层而非 DHCP 本身：

- **DHCP snooping / DHCPv6 Guard**：交换机把端口分成 trusted 与 untrusted，只在 trusted 口接受 OFFER / ACK / REPLY，并维护租约绑定表；非授权应答在入端口就被丢弃。
- **RA Guard**：过滤非授权端口发出的 RA，避免 RDNSS / DNSSL 绕过 snooping 单独下发（规范按名称在 RFC Editor 检索）。
- **端口安全 / 802.1X**：限制端口上可出现的 MAC 与设备数量，减少私接小路由的机会。
- **客户端侧核对**：把”当前生效的 DNS 是谁”做成常规巡检项，出现陌生上游、多个 OFFER、option 6 与预期不符即告警；`resolvectl dns` 与 dnstap 里的异常 QNAME 分布都是可用的信号（第 31 章）。

这些控制与第 19 章压制上级 DNS 是互补关系：RA Guard / snooping 保证”只有我发的算数”，RDNSS 参数保证”我发的是本地解析器”，两者缺一，另一方都能被绕过。

33.8 教学配置片段（全部文档地址段）

下面几段都是教学骨架，不是生产模板。所有地址取自 RFC 5737 文档段与 `2001:db8::/32`，域名用 `example.net` / `home.arpa.`，主机名 `gw-01` 等仅为示意。**option 15 与 119 一律注释掉**——按 [RFC 5625] §5.2，未配置就不要发，这本身就是正确配置。

33.8.1 ISC dhcpd: option 6 / 15 / 119

```
# /etc/dhcp/dhcpd.conf
authoritative;
default-lease-time 600;
max-lease-time 3600;

subnet 192.0.2.0 netmask 255.255.255.0 {
    range 192.0.2.20 192.0.2.200;
    option routers             192.0.2.1;    # gw-01（示意）
    option domain-name-servers 192.0.2.53;   # option 6：本地递归解析器
    # option domain-name       "home.arpa."; # option 15：未配置就不要发
    # option domain-search     "home.arpa."; # option 119：未配置就不要发
    option ntp-servers         192.0.2.123;  # option 42：时间同步
}
```

- ①为什么这么设：option 6 只给一个本地解析器，避免第 5 章”多上游竞速”把查询送到不该去的地方；15/119 注释掉，避免查询放大与无效 TLD 查询；42 保证 Kerberos / GSS-TSIG 与 DNSSEC 验签有可用时钟。option 119 的 `domain-search` 压缩编码声明与 Kea 的写法不同，按名称在官方手册检索，不要手写字节序列。
- ②怎么验证：`dhcpd -t -cf /etc/dhcp/dhcpd.conf && echo OK`；客户端 `resolvectl dns` 应只列出 `192.0.2.53`，`resolvectl domain` 应为空或只含预期后缀。
- ③失败模式 / 回滚点：误打开 15/119 会出现成片无效 TLD 查询；回滚＝重新注释两行、reload 服务，并留意已有租约要等 T1 才收敛（正是本章”租约时序陷阱”）。若要临时恢复裸名互访，只加 `home.arpa.` 一个后缀，不要堆列表。

33.8.2 Kea: 同一组 option 的等价写法

```
{
  "Dhcp4": {
    "interfaces-config": { "interfaces": [ "eth0" ] },
    "valid-lifetime": 600,
    "subnet4": [
      {
        "subnet": "192.0.2.0/24",
        "pools": [ { "pool": "192.0.2.20 - 192.0.2.200" } ],
        "option-data": [
          { "name": "routers",             "data": "192.0.2.1" },
          { "name": "domain-name-servers", "data": "192.0.2.53" },
          { "name": "ntp-servers",         "data": "192.0.2.123" }
```

```
        ]
      }
    ]
  }
}
```

- ①为什么这么设：与 ISC dhcpd 保持同一意图——只下发 option 6，`domain-name`（15）与 `domain-search`（119）**故意不出现在 `option-data` 里**；Kea 未配置的 option 不会凭空编一个默认值，这正是 §5.2 想要的行为。
- ②怎么验证：`kea-dhcp4 -t /etc/kea/kea-dhcp4.conf` 做配置自检；字段名、code 与 `csv-format` 的取值以 Kea 官方手册的内置 option 定义为准（见官方资料），不要照抄臆造的类型。
- ③失败模式 / 回滚点：手写 `code` / `space` 不匹配会导致启动自检失败或下发空 option；回滚＝还原备份的 JSON，自检通过后再 reload。若确需 119，先在隔离网段用 `csv-format` 验证客户端对压缩编码的支持，再决定是否下发。

33.8.3　dhcpd6.conf: 23 / 24

```
# /etc/dhcp/dhcpd6.conf
default-lease-time 600;
preferred-lifetime 300;

subnet6 2001:db8:1::/64 {
    range6 2001:db8:1::100 2001:db8:1::200;
    option dhcp6.name-servers 2001:db8:1::53;   # option 23
    # option dhcp6.domain-search "home.arpa.";   # option 24: 未配置就不要发
}
```

- ①为什么这么设：按上一节的结论，**v6 侧只保留一条 DNS 来源**。这里给出 23 是为了演示写法；若 LAN 已按第 19 章用 RA 广告 ULA 解析器，就把 23/24 一并注释掉，只由 RA 负责，避免两侧各发一套。option 56（[RFC 5908]）的内置写法见官方手册，本片段不臆造指令名。
- ②怎么验证：`dhcpd -6 -t -cf /etc/dhcp/dhcpd6.conf && echo OK`；客户端 `resolvectl dns` 只应出现一个来源，且 v4/v6 两侧指向同一台解析器。
- ③失败模式 / 回滚点：23 与 RA 的 RDNSS 各发不同地址时，主机合并策略不确定、部分流量绕过本地策略（本章"优先级战争"）；回滚＝注释 23/24、统一由 RA 下发，或两边填完全相同的地址。IA_PD（option 25）不在本片段内，前缀轮换的处理见第 18、19 章。

33.8.4　客户端侧校验

```
# 谁在下发 DNS、下发了什么
resolvectl dns
resolvectl domain
# 时间源与偏差（option 42 / 56 是否真的生效）
timedatectl timesync-status
chronyc tracking
# 解析器本身是否可用（尾点绕开搜索域追加）
dig +short example.org. @192.0.2.53
# v6 侧前缀与地址是否符合预期
ip -6 addr show
```

- ①为什么用这几条：把四组字段（6、15/119、42/56、IA_PD 的间接影响）各留一个可当场确认的观测点，一次巡检就能定位交接面在哪一段断了。
- ②怎么读输出：Windows 侧用 `ipconfig /all` 看"DNS Servers"与"DNS Suffix Search List"，macOS 侧查系统配置里的搜索域；命令形态按名称在官方资料检索，本书不逐平台展开。`resolvectl dns` 出现陌生上游时先怀疑 rogue DHCP 或两侧通道冲突，而不是先改 Unbound。
- ③失败模式 / 回滚点：这套命令只观测、不改写配置，回滚无风险；但要注意它反映的是**当前生效值**，租约未续期前可能与服务器新配置不一致，务必结合 T1 时间点判读。

33.9　本章要点

- **option 6 决定查询去哪、15/119 决定名字被追加几次、81 决定动态更新由谁做、42/56 决定这套机制有没有可信时钟**；四组字段都不是"DNS 服务器配置"，却先行决定了 DNS 行为。

- **[RFC 5625] §5.2 的默认应是空**：没有标准的 `local` 后缀，未配置就不要发 option 15 / 119；固件默认值会造成成片无效 TLD 查询打根服务器，并与 `ndots`（第 20 章）叠加放大。
- **v6 侧只留一条 DNS 来源**：[RFC 8106] 的 RA 选项与 [RFC 3646] 的 DHCPv6 23/24 并行且主机合并策略不一，各发一套必然让部分客户端绕过本地策略；按第 19 章用 RA 广告 ULA 解析器即可。
- **租约是推送模型**：WAN 建立后上游 DNS 变了，客户端要等 T1 续期才感知（[RFC 5625] §5.3）；短租约压窗口，稳定的 DNS 代理地址治本。
- **option 81 的自报 FQDN 是信任边界**：凭据可信不等于名字可信；防守靠 Name Protection / DHCID、低权限凭据、Global Query Blocklist、占位记录、更新 ACL 与二层 DHCP snooping / RA Guard，本书只写机理与防守。
- **改了 DHCP 配置不等于改了客户端**：排查”DNS 忽然好了/坏了”先看 `resolvectl dns` 与 `resolvectl domain`，再动 Unbound。

33.10 参考资料

1. https://www.rfc-editor.org/rfc/rfc2132 ——DHCP Options 与 BOOTP Vendor Extensions（option 6 / 15 / 42）
2. https://www.rfc-editor.org/rfc/rfc5625 ——DNS Proxy Implementation Guidelines，§5.1 网关默认值、§5.2 默认域应为空、§5.3 租约时序
3. https://www.rfc-editor.org/rfc/rfc3397 ——DHCP Domain Search Option（option 119 压缩编码）
4. https://www.rfc-editor.org/rfc/rfc4702 ——DHCP Client FQDN Option（S/N/O 标志）
5. https://www.rfc-editor.org/rfc/rfc3646 ——DHCPv6 的 DNS 配置选项（23 / 24）
6. https://www.rfc-editor.org/rfc/rfc8106 ——RA 的 RDNSS / DNSSL 选项
7. https://www.rfc-editor.org/rfc/rfc2136 ——DNS UPDATE（动态更新）
8. https://kea.readthedocs.io/ ——ISC Kea 官方手册（option-data 与配置自检）

其余在正文按编号引用的 [RFC 6731]、[RFC 5908]、[RFC 8415] 等，按 `https://www.rfc-editor.org/rfc/rfcNNNN` 的地址模式在 RFC Editor 检索原文。Akamai 2023《Weaponizing DHCP DNS Spoofing》与 TrustedSec 2021《Injecting Rogue DNS Records Using DHCP》按标题在厂商研究博客检索；本书只引用其公开的机理结论，不提供可复制的注入流程。

第三十四章

红队视角的 DNS 侦察

授权纪律（本章前置条件）：所有枚举、扫描与主动查询，只在自己持有**书面授权**的名字与网段上进行。只读的证书透明日志、被动 DNS、公开注册数据可以自由查阅；一旦要对目标权威服务器或递归解析器发出主动查询，必须先有把该动作明确写进 scope 的授权。发现疑似可接管的悬空记录或开放服务，**不要 claim**（不注册、不绑定、不占用、不上传验证文件），只记录指纹并通知资产方。本书讨论的是 DNS 协议机制、失效边界与其留下的观测信号，不构成绕过网络管理的操作指南。

34.1 授权纪律与本章边界

34.1.1 书面 scope 要写清的四件事

一次授权测试里的 DNS 部分，scope 至少要写清：允许触碰的**名字与网段**（含上级域与所有子域的边界）、允许的**动作类别**（只读查询、批量名字查询、区传送请求、解析器能力探测，逐项列明）、**时间窗与源地址**、以及**中止条件与联系人**。任何一项含糊，都不算授权。

特别容易漏的是第三方资产：一条 SPF `include:` 链会把你引到别人的域，读那条公开 TXT 记录是允许的，但对那个域发起主动探测就超出了 scope。授权书里应当明确写出「只读公开记录」与「主动查询」的分界，而不是笼统写「DNS 侦察」。

内网侧更严格。广播应答、DHCP 交接面、动态更新这些动作会影响同一网段里的**无关主机**，因此 scope 不只是「哪些域名」，还必须写清「哪些二层域、哪些网段、允许监听还是允许应答」。交接面机制见第 33、23 章。

34.1.2 与第 29 章的分工

第 29 章是**防守侧**：我的暴露面怎么自查、怎么加固，给的是阈值表、nft 骨架、悬空 CNAME 指纹表与自查清单。本章是**对手侧**：他按什么顺序找、每一步在协议上留下什么、防守方怎么把这些残留变成告警。

阈值表、nft 规则骨架与悬空 CNAME 指纹表见第 29 章；告警规则的具体写法、Prometheus 与 Grafana 的集成见第 31 章。这里只说信号的**语义、来源、纯度与去向**。

一个便于记忆的分工：第 29 章回答「我该关掉什么」，本章回答「关不掉的那些，我该看见什么」。

34.1.3 边界

不列字典、候选名字清单、可复制即用的枚举命令串，也不写规避限速或检测的做法，不列真实目标域名。出现的软件名只是**类别与用途**：massdns、dnsx、subfinder、amass、fierce、dnsrecon 一类都属于「批量名字查询」这一族。

本章也不做工具横评。「只写类别」是有意的：侦察链的形态由**查询在协议上长什么样**决定，而不是由某个工具的开关决定。换一个工具，只要查询形态相同，防守侧看到的信号就相同；反过来，同一工具换一种节奏，也可能落进完全不同的告警窗口。防守方该建模的是查询形态，不是工具指纹。

另外，DNSSEC 提供的是**数据完整性与来源认证**，DoT / DoH / DoQ 提供的是**传输加密与隐私**，二者不互相替代。本章谈签名区可被枚举时，说的是前者的设计取舍，不是「加密了就查不到」。

34.2　侦察链总览

对手的顺序大体是固定的，因为每一步的输出是下一步的输入：

- 先从公开资料里拿到一份名字清单，全程不碰目标；
- 再对这份清单做批量查询，把「存在」与「不存在」区分开；
- 然后问区的结构：谁权威、委派是否一致、区能否整体取出、签名链长什么样；
- 接着把解析器本身当成资产，探它的缓存、身份与能力；
- 再把公开可读的信任记录铺开，拼出邮件与证书供应链；
- 若已获得内网落脚点，最后转向广播域与 DHCP / 自动发现交接面。

这个顺序有一个对防守方很有用的性质：**可见性单调递增，但「像正常流量」的程度不单调**。第一阶段目标侧几乎零信号；第二阶段起开始产生查询流，噪声最大；第三阶段是低频但高价值的一次性查询；第四阶段探的是解析器自己的管理面；第五阶段的 QTYPE 几乎全是 TXT、SRV、HTTPS 这类完全正常的类型，最难靠流量异常区分；第六阶段的信号甚至不在公网权威的 dnstap 里。

所以「越往后越容易被看见」并不成立。真正可靠的做法不是盯某一个阶段，而是把六个阶段各自最纯的那一两个信号取出来，分别接到第 31 章的告警里。下面逐节展开。

```
教学用示意：侦察链与信号强度（非操作顺序图，不含任何目标信息）
被动收集   → 目标侧无查询；信号在外部公开日志
名字枚举   → 查询流噪声最大；量类信号
区结构     → 低频高价值；质类信号（单次即值得看）
解析器     → 管理面查询；纯度最高的质类信号
信任记录   → QTYPE 正常；靠台账差异而非流量异常
内网侧     → 信号在广播域 / DHCP / 更新审计，不在公网 dnstap
```

示意里的「量类」与「质类」是本书为告警设计做的归纳：量类要先测基线再定偏离，质类不必等基线——一次 AXFR 尝试、一条 CH 类查询本身就是事件。

34.3　被动收集：不留查询到目标

34.3.1　数据来源的机理

证书透明（Certificate Transparency，CT）日志是只追加、可公开审计的证书签发记录。证书签发时，其中的 SAN 被提交进日志，任何人都能从日志侧读到新名字，**不需要访问目标主机、目标权威服务器或目标递归解析器**。名字在证书签发的那一刻就已成为公开信息，见官方资料（本章不引具体 RFC 编号）。

被动 DNS 是另一类公开或半公开的历史视图：采集点把观测到的「名字→记录→首次/末次见到时间」聚合起来，因此能给出**已经下线**的名字与旧的记录值。它的价值不在于「现在有什么」，而在于「曾经有什么」——下线的记录往往正是配置残留最集中的地方。

WHOIS / RDAP 给的是注册数据：注册商、委派 NS、创建与更新时间。它不含子域，但能划定组织边界与命名习惯。公开配置、代码仓库、文档、搜索引擎索引、证书链里出现的名字，属于同一类「别人替目标公开了」的情报。见官方资料。

34.3.2　为什么目标侧看不到

这一阶段的共同点是：**所有查询都发往第三方数据源，不发往目标**。目标权威服务器的日志、目标递归解析器的 dnstap、目标 Web 访问日志里，都不会出现这次收集的任何痕迹。这是 DNS 侦察里最难防的一段，也是最容易被防守方误判为「没人在看我」的一段。

由此得到一个不太舒服但必须接受的结论：把子域当作秘密来保护是无效的。名字一旦用于签发证书、一旦被任何递归链路解析过，就以你无法召回的方式公开了。防守侧该做的是**资产台账**，不是「不让名字被知道」——台账见第 29 章，本章只点出对手的免费情报来自你自己的公开动作。

34.3.3　防守价值与阶段信号

对手看不见的部分，防守方可以自己看：把 CT 日志当作自己这一侧的被动收集，发现台账里没有的 SAN，要么是影子资产，要么是误签发，要么是有人先一步用了你的名字。这属于第 29 章的自查动作。

阶段信号结论很干脆：**目标侧 dnstap 无信号**。可观测的是外部公开日志里出现新 SAN，它不进 dnstap，也不该被写成解析器告警。把这一阶段硬塞进解析器监控，只会制造找不到对应查询流的空告警。它该接的是证书与资产台账的变更通知。

34.4 名字枚举：从静默到高噪声

34.4.1 类别与机理

批量名字查询这一族工具（massdns / dnsx / subfinder / amass / fierce / dnsrecon 一类）的共同机理只有一句：把一份候选名字集合，以并发方式送进查询，再用应答码把「存在」与「不存在」二分。差别只在候选名字的来源（证书、索引、聚合视图、组合变换）与并发形态，而不在协议。候选集合怎么来、长什么样，本章不写。

命中一个名字所需的协议信息极少：一个 QNAME、一个 QTYPE、一个 RCODE。正因为如此，这一段在防守侧呈现为**纯粹的统计异常**，而不是某条「恶意查询」的内容特征。想靠单条查询的字节特征抓枚举，通常会落空；要靠窗口内的聚合。

34.4.2 四条典型信号

- **单源 qps 突增**：同一源在一个窗口里发出的查询数远高于常态。源也可能是分布式的，此时共享的往往不是源地址，而是同一批 QNAME 的父域与相近的时间分布。
- **随机子域**：左侧标签呈高熵、无字典语义的字符串。这与第 29 章隧道检测里的高熵 QNAME 同源，区别只在目的——这里是为了撞出存在的名字，那里是为了编码数据。两者在指标上很难分开，因此第 29 章的检测模型可以直接复用。
- **NXDOMAIN 比骤升**：批量探测天然以失败应答为主。顺带一提，失败应答会按 SOA minimum 进入负缓存（[RFC 2308]），所以洪泛不只污染日志，也会占用缓存条目；缓存与刷新的工程细节见第 7 章。
- **ANY 或某个罕见 QTYPE 集中**：一次问全类型的查询形态在正常业务里很少见，是纯度较高的类别信号。相关应答行为的标准化讨论见官方资料，本章不引编号。

这四条都能从 dnstap 聚合得到：唯一 FQDN 数、QTYPE 分布、NXDOMAIN 比、单源 qps。第 31 章已把这些列为指标，本章只说明**它们为什么恰好对应这一段**，以及「唯一 FQDN 数」为什么比 qps 更稳：qps 会被并发度与重试影响，唯一 FQDN 数描述的是对手在覆盖多大的名字空间。

34.4.3 防守价值与授权边界

防守动作不在本章：限速、ACL、权威侧响应限速、基线测量与告警规则分别在第 9、29、31 章。本章的落点是「把这四条信号明确归因到枚举阶段」，让告警带上语义，而不是只报「qps 高了」。归因清楚了，误报治理才有依据——证书续期、CDN 健康探测、监控拨测、批量资产同步都会产生突发查询，白名单与基线在第 31 章。

主动枚举必须先有写进 scope 的授权。本章不写任何规避限速或规避检测的做法，这类内容既不属于防守方需要，也不属于本书范围。

34.5 区结构探测：委派、传送与签名链

34.5.1 NS / SOA 与委派一致性

区的骨架是公开的：NS 给出委派点，SOA 给出主权威、序列号与几组计时器。`+nssearch` 这一类查询形态的含义是「逐个问各台 NS，看它们对同一个区的说法是否一致」，用来发现委派与权威应答之间的偏差。本章只写这个形态的名字与语义，不给命令。

不一致本身就是情报：它意味着有些权威服务器没同步、有些还在应答旧数据、或者区被拆分维护。对防守方而言，这属于配置正确性问题，第 29 章的自查清单会覆盖；本章关注它是「一次查询就能读出结构」的低噪声动作。

34.5.2 AXFR 误开

区传送（AXFR）是主备权威服务器之间同步整区的正常机制，不是漏洞。出问题的是访问控制：若没把传送限制在从服务器清单内，一次 AXFR 就把整个区交了出去，对手不必再枚举任何名字。第 29 章讲防守侧怎么自查与收紧，本章只讲它在侦察链上的位置——**它是对枚举阶段的替代，而不是补充**。

这一点直接决定告警设计：AXFR 是**低频高价值**事件，靠 qps 窗口永远抓不到。该盯的是 QTYPE 为 AXFR、源不在从服务器清单、以及承载它的 TCP/53 连接本身。单条即成事件，不需要基线，也不需要阈值表。

34.5.3 NSEC / NSEC3 walking 与 [RFC 5155]

DNSSEC 要证明「某个名字不存在」，又不能对每个不存在的名字预先签名，于是 NSEC 链给出**相邻的两个已签名名字**：只要待证名字落在两者之间，就能证明它不存在。副作用是链可以顺着走下去——从一条链跳到下一条，把整个签名区的名

字逐个读出来。这是「签名的否定应答」在设计上的必然产物。

[RFC 5155] 的 NSEC3 用哈希后的名字替代明文名字，目的正是**抗枚举**：链上不再直接给出可读的名字，顺着走得到的是哈希值。要强调两点，以免混淆：哈希是公开、可离线计算的，不是加密；NSEC3 提高的是枚举成本，不是把区内容变成机密。更进一步，DNSSEC 的整个目标都是数据完整性与来源认证，从来不包含保密性——把「签名区可被枚举」当成 DNSSEC 的漏洞，或以为上了 DoT / DoH 就查不到，都是把两类机制搞混了。

对手在这一步的信号不是失败应答洪泛，而是对**相邻名字**的连续查询、以及成组的 NSEC / NSEC3 相关记录读取。它比枚举阶段安静得多，但查询序列有清晰的结构特征。

34.5.4　DS / DNSKEY 暴露与阶段信号

DS 与 DNSKEY 是信任链上的公开材料，暴露的是算法选择、KSK / ZSK 结构、密钥轮换节奏与签名有效期窗口。这里没有私钥泄露——私钥本就不在 DNS 里。它的情报价值在于让对手判断签名区的成熟度与轮换窗口，从而挑选变更前后最脆弱的时间点。验证器与信任锚的配置见第 10 章。

本阶段的信号汇总：AXFR 尝试、TCP/53 连接、对相邻名字的连续查询、DS / DNSKEY 集中读取、以及委派一致性查询。其中前两项是质类信号，后几项需要短窗口聚合。全部交给第 31 章的告警，检测模型沿用第 29 章。

34.6　解析器探测：把解析器本身当资产

34.6.1　缓存窥探

递归解析器会缓存应答。若它对 RD=0 的非递归查询也直接返回缓存内容，查询者就能判断「某个名字最近被别人查过」：有应答、TTL 在递减、且没有触发新的迭代，基本就是缓存命中。这就是缓存窥探（cache snooping）的机理——解析器把**自己缓存的存在性**暴露给了外部。

泄露的是元数据，不是内容，也取不到原查询者的身份。但对内网或共享解析器来说，「谁最近访问过哪些名字」本身就是敏感信息。防守侧的正确姿态是把递归服务只开放给授权客户端（第 9 章 `access-control` 默认拒绝），而不是试图给缓存加混淆。

34.6.2　开放解析器识别

从外部发一个本不该被服务的递归请求，看是否得到应答，就能识别开放递归解析器。[RFC 5358] 明确反对可被用作反射放大器的开放递归服务。这类解析器既会被拿去放大流量，也会被当作免费的批量查询通道——后者正好接上本章的名字枚举阶段。防守动作在第 9 章，本章只指出：**一个配置疏漏会同时喂给两个不同阶段的对手**。

34.6.3　CH 类查询与第 9 章

CHAOS（CH）类里的 `version.bind` 一类 TXT 名字，是历史遗留的软件版本与身份自报接口。第 9 章的 `hide-identity` / `hide-version` 就是针对它：要么不给应答，要么返回中性值。机理上，它是「解析器主动回答关于自己的问题」。

对防守方来说，这是整条侦察链上**纯度最高**的信号：QCLASS 不是 IN、QNAME 形态固定、QTYPE 多为 TXT，正常业务流量几乎不会长成这样。一次就值得记一条事件，不需要基线、不需要窗口、误报极低。代价是它也可能来自合规的资产扫描，所以告警应指向「核对来源」，而不是「判定入侵」。

34.6.4　EDNS 能力与限速行为

EDNS(0) 通告的缓冲大小、扩展选项、DNS Cookie 一类信息，能反映解析器实现与能力集；持续观察应答节奏还能读出限速策略。相关标准编号与实现差异见官方资料，本书第 9、13 章分别讲访问控制与查询隐私，本章不引具体编号、不写版本、也不写「如何据此调整查询节奏」——那属于规避检测，不在本书范围。

阶段信号汇总：CH 类（QCLASS ≠ IN）查询、RD=0 的非递归查询占比、来自非授权源的递归请求、异常 EDNS 选项组合。前两项是质类，后两项要按源聚合。这些都不依赖 payload 内容，因此在加密传输场景下依然可见——再强调一次，DoT / DoH 保护的是链路，不改变解析器侧日志里能看到什么。

34.7 信任记录枚举：公开可读的供应链地图

34.7.1 第 30 章的记录族

第 30 章列过一张完整的信任记录表，它们有一个共同属性：**公开、可读、常被缓存**。对手把这张表当成地图来读：

- SPF 的 `include:` 链把一条 TXT 展开成第三方发件来源清单（[RFC 7208]）；
- DKIM 的 `sel._domainkey.example.org.` 形态暴露 selector 命名与轮换习惯（[RFC 6376]）；
- DMARC 的 `rua` 给出报告接收地址；
- MTA-STS 的 `_mta-sts` TXT 与 `mta-sts` 主机上的策略文件（[RFC 8461]），加上 TLS-RPT，说明 SMTP 侧强制 TLS 到什么程度；
- TLSA 的 `_25._tcp.mx.example.org.` 形态绑定证书（[RFC 7672] 一类），**只在 DNSSEC 签名区生效**；
- CAA 限定可为该域签发的 CA；
- `_acme-challenge.example.org.` 一类验证记录是否残留；
- SRV、SVCB / HTTPS（[RFC 9460]）给出服务端点与自动配置入口；
- 各类域所有权验证 TXT，以及 MX、SSHFP 等第 30 章已覆盖的名字形态。

拼起来就是邮件供应链、证书供应链、外包服务商清单，以及一份「哪些记录明显没人维护」的候选表。所有这些都只需要读取公开记录。

34.7.2 为什么「公开」不等于「无害」

必须说准：`_acme-challenge` 里的验证值是一**次性**的，不是秘密；真正的高风险凭据是能写这条记录的 API 权限（第 30 章）。对手读残留记录，读不到可用的 token，读到的是「这个域用 DNS-01、变更流程有没有清理、谁在管 DNS」。这是流程情报，不是凭据泄露。

同理，SPF `include:` 指向别家的域，读那条 TXT 是公开信息；但据此对别家发起主动探测就出了 scope，这一点在授权纪律里已经写明。公开可读不等于可以随便动。

这一阶段在流量上最难抓：QTYPE 是 TXT、SRV、HTTPS，全是正常类型；名字多带下划线前缀，也是正常形态；查询量往往不大。靠「异常 QTYPE」或「qps 突增」基本无效。

34.7.3 防守价值与阶段信号

防守侧该换思路：第 30 章的记录清单是「应该有什么」，对手的查询集合是「他在找什么」，**差异**才是信号。具体表现是——同一个父域下大量下划线前缀名字被连续查询、台账里根本不存在的名字被反复问、TXT 类查询集中在某一小段时间、对已下线记录的连续 NXDOMAIN。

所以这一阶段的告警输入不只是 dnstap，还要有资产台账比对结果；台账本身在第 29、30 章。把「查询了台账里没有的名字」写成一条规则，比调 qps 阈值有效得多。这也解释了为什么本章反复强调「信号带语义」：同样的 NXDOMAIN，在名字枚举阶段是噪声，在这里是台账差异。

残留的 `_acme-challenge`、过期验证 TXT、指向已下线服务的记录，防守动作是清理与生命周期绑定（第 30 章）。

34.8 内网侧授权场景：广播与交接面

34.8.1 场景前提

以下只在**已取得书面授权、且拿到内网落脚点**的渗透测试里成立，scope 必须覆盖具体二层域与网段。机理层面点到为止，操作细节不在本书范围；防守侧配置分别在第 29、33、23 章。

34.8.2 广播应答的机理

LLMNR、NBT-NS、mDNS 一类链路本地名称解析，共同机理是「单播 DNS 解析失败后，在链路本地多播问一句」。问是广播/多播的，答却是单台主机可以抢的：同一二层域里任何主机都能声称自己就是那个名字，应答方由此成为该名称的解析者。相关标准状态见官方资料，本章不引编号。

信号不在公网权威的 dnstap 里，而在广播域观测中：同一名字出现多个应答源、应答者数量突然变化、非 DNS 端口上的名称流量增多。第 27 章讲 mDNS / DNS-SD 的正常用法，本章只点出它在授权测试里为何成为名称解析的竞争面。

34.8.3　第 33 章 DHCP 注入与第 23 章 WPAD 名称劫持

DHCP 的 Client FQDN option 81 与 [RFC 2136] 动态更新连在一起：若更新 ACL、名称保护与凭据权限没配好，客户端可以往区里写名字（包括 `wpad` 一类被特别关注的名字）。机理与防守（DHCP snooping、Name Protection / DHCID、低权限专用凭据、Global Query Blocklist、预置占位记录）在第 33 章。这是一条**从网络配置写到 DNS 数据**的路径，所以对手动静会体现在动态 UPDATE 审计里。

WPAD 侧则是名字归属问题：草案里的 devolution 会逐级剥左侧标签，可能越出组织边界；谁能提供 `wpad` 这个名字的解析，谁就能影响自动代理配置。五种发现机制与草案状态在第 23 章，本章不复述，只把「WPAD 名字的解析请求与突然出现的应答者」列为可观测信号。本书的总体结论不变：WPAD 只作遗留兼容，不该成为新架构的依赖。

34.8.4　阶段信号与去向

内网阶段的信号源是广播域观测、DHCP 服务器日志、DNS 动态 UPDATE 审计，以及内网递归解析器上的名字查询。**不能只盯公网权威**：这些动作里相当一部分根本不产生出向查询。第 35 章的蜜罐与被动观测点，正是作为这类「生产解析器看不见」的补充数据源接进同一套告警。

这也修正了侦察链总览里的直觉：最后一段的可见性不一定最高，它只是**换了采集点**。防守方若只在出口解析器上布 dnstap，会整段漏掉。

34.9　从信号到告警：喂给第 31 章与第 29 章

34.9.1　六类统一字段

把六个阶段的残留压成可计算字段，就是本章反复提到的那几项：dnstap 里的**唯一 FQDN 数**、**QTYPE 分布**、**NXDOMAIN 比**、**单源 qps**、**AXFR 尝试**、**CH 类查询**。内网侧再加两项：动态 UPDATE 审计事件、广播应答源计数。字段的采集在第 7 章，聚合与可视化在第 31 章，检测模型在第 29 章。

```
一次授权演练里聚合出的观测记录（非配置、非告警阈值）
地址均为文档地址段：192.0.2.0/24 [RFC 5737]、2001:db8::/32；域名为 example.org.
window      : 10:00Z ~ 10:05Z
source      : 192.0.2.10
secondary   : 2001:db8::10
parent_zone : example.org.
unique_fqdn : < 窗口内去重后的 QNAME 计数 >
qtype_dist  : < 按 QTYPE 的计数分布 >
nxdomain_r  : < 失败应答占比 >
qps_peak    : < 窗口内单源峰值 >
axfr_try    : <AXFR 请求计数 >
ch_class    : <QCLASS != IN 的查询计数 >
说明：这里只列字段，不列取值门槛；基线测量与阈值见第 29 / 31 章
```

34.9.2　量类与质类分开接

前四类（唯一 FQDN 数、QTYPE 分布、NXDOMAIN 比、单源 qps）是**量类**：必须先有基线，才能谈偏离；基线测量周期、白名单与误报治理在第 31 章，阈值表在第 29 章。后两类（AXFR 尝试、CH 类查询）是**质类**：单次出现即成事件，不必等基线，也不该被聚合窗口稀释掉。把两类混进同一条规则，结果通常是质类信号被平均掉、量类信号被误报淹没。

内网侧的 UPDATE 审计事件属于质类，广播应答源计数属于量类，同样按这个分法接。

34.9.3　告警去向与一个常见误判

链路是：dnstap / 广播域观测 / UPDATE 审计（第 7 章采集）→聚合字段→第 31 章告警规则→第 29 章检测模型与加固动作。每个字段都要对应侦察链的某一段，让告警能带上阶段标签；标签化了之后，处置手册才能按阶段分流。

一个要避免的结论是「把源封掉就行」。源可分布式、可伪造，且开放递归与未限制 AXFR 的问题在于**服务能力本身被滥用**，不在于某一个源。第 29 章已经给出这个结论（第 9 章 `access-control`、禁止未授权 AXFR、权威侧响应限速），本章引用而不复述。

34.10 授权测试纪律的完整闭环

34.10.1 测试前与测试中

测试前：书面 scope 覆盖名字/网段边界、动作类别、时间窗、源地址、联系人、中止条件与止损方案；只读公开数据（CT、被动 DNS、注册数据、公开记录）可以自由查；主动查询、区传送请求、解析器能力探测、任何内网动作，逐项列明才可做。

测试中：只用列明的源，只在时间窗内，不碰 scope 外的名字，不碰 `include:` 链指向的第三方资产，不做任何规避限速或规避检测的动作。所有动作留时间戳，便于防守方事后在自己的 dnstap 里逐条对上——这既是纪律，也是让报告可核验的前提。

34.10.2 发现可接管资源时

发现悬空 CNAME、指向已下线第三方服务的记录、或任何疑似可接管的资源时，**不要 claim**（OWASP WSTG 明确要求 “Do not claim it”）。不注册同名资源、不上传验证文件、不绑定、不占用；只把指纹、记录值、观察时间与公开可核验的证据写进报告，通知资产方。指纹表在第 29 章，本章不复述。

这条纪律不只是合规要求：一旦 claim，就从「只读观测」变成了「对第三方资产产生写入」，同时也毁掉了防守方复现与归因的原始状态。

34.10.3 报告要素

```
报告骨架（占位符，不含真实目标、不含复现命令）
时间窗          ：< 起止时间，含时区 >
源地址          ：< 书面 scope 里列明的源，文档地址段示例 192.0.2.10>
scope 依据      ：< 授权书编号 / 签署方 / 允许的动作类别 >
动作与复现      ：< 按时间顺序列出动作类别，供防守方在日志中核对 >
观测到的信号    ：<dnstap / 审计字段与对应阶段标签 >
影响判断        ：< 只写可核验的事实与推断边界 >
发现的可接管面：< 只记指纹，不 claim，不写入第三方 >
止损建议        ：< 删除悬空记录、收紧 ACL、清理残留验证 TXT 等，指向第 29 / 30 章 >
回滚与清理      ：< 测试期间产生的记录、凭据、临时资源如何撤销 >
```

把「观测到的信号」写成防守方能在自己日志里找到的字段，而不是工具输出截图，是这份报告最有价值的部分：它让一次授权演练直接变成第 31 章告警规则的验收用例。反之，只报「发现 X 个子域」而不给时间窗与源，防守方无法确认自己的监控是否真的看见了。

34.11 阶段 / 工具形态 / 观测信号 / 防守动作 / 本书章节

阶段	工具形态	观测信号	防守动作	本书章节
被动收集	CT 日志监视、被动 DNS 聚合视图、WHOIS/RDAP、公开配置与索引检索	目标侧 dnstap 无信号；外部公开日志出现新 SAN、台账外的名字	资产台账与 CT 变更通知；接受名字已公开，不把子域当秘密	第 29 章（自查）、第 30 章（记录清单）
名字枚举	批量名字查询工具族（massdns / dnsx / subfinder / amass / fierce / dnsrecon 一类），候选名字来自证书、索引与聚合视图	唯一 FQDN 数骤增、QTYPE 分布偏斜、NXDOMAIN 比升高、单源 qps 突增、随机子域、ANY 集中	按窗口聚合后告警；基线、白名单与误报治理；限速与 ACL 只作引用	第 29 章（检测模型）、第 31 章（告警）、第 9 章（ACL）
区结构探测	委派一致性查询（`+nssearch` 一类形态）、AXFR 请求、NSEC / NSEC3 链读取、DS / DNSKEY 读取	AXFR 尝试、TCP/53 连接、相邻名字的连续查询、DS/DNSKEY 集中读取	AXFR 与 CH 类按单条事件告警；区传送限制在从服务器清单；委派一致性自查	第 29 章（自查）、第 10 章（DNSSEC）、第 31 章（告警）

阶段	工具形态	观测信号	防守动作	本书章节
解析器探测	非递归缓存探测、开放递归识别、CH 类版本/身份查询、EDNS 能力探测	CH 类（QCLASS ≠ IN）查询、RD=0 查询占比、非授权源的递归请求、异常 EDNS 选项组合	默认拒绝非授权递归；关闭/中性化版本与身份自报；不提供可被滥用的服务能力	第 9 章（最小暴露面、hide-identity/version）、第 31 章（告警）
信任记录枚举	公开 TXT / SRV / SVCB / HTTPS 记录读取、`include:` 链展开、验证记录残留核对	台账里不存在的下划线前缀名被连续查询、TXT 类查询集中在短窗、对已下线记录的连续 NXDOMAIN	资产台账比对而非流量阈值；清理残留验证 TXT；写记录权限最小权限化	第 30 章（信任记录）、第 29 章（台账）、第 31 章（告警）
内网侧授权场景	链路本地多播名称应答观察、DHCP 交接面观察、自动发现名字观察	同一名字多个应答源、动态 UPDATE 审计事件、广播应答源计数、WPAD 名字解析请求	另接广播域/DHCP/UPDATE 数据源；DHCP snooping、名称保护、预置占位记录	第 33 章（DHCP 交接面）、第 23 章（WPAD）、第 27 章（mDNS）、第 35 章（蜜罐）

表里的「防守动作」一列刻意只写动作类别，不写规则骨架与阈值：这两样分别属于第 29、31 章。真正要带走的是中间两列的对应关系——每一种查询形态，都恰好对应一个可以计算出来的字段；能算出字段，才谈得上告警。

34.12　本章要点

- 授权纪律是前置条件：只读公开数据可自由查，主动查询、区传送请求、解析器探测与任何内网动作必须先写进书面 scope；发现可接管资源不要 claim，只记指纹并通知资产方。
- 侦察链六阶段的可见性不单调：被动收集在目标侧零信号，名字枚举噪声最大，区结构与解析器探测是低频高纯度事件，信任记录枚举在流量上最像正常业务，内网阶段的信号干脆换了采集点。
- 六类统一字段是唯一 FQDN 数、QTYPE 分布、NXDOMAIN 比、单源 qps、AXFR 尝试、CH 类查询；前四类量类要先有基线，后两类质类单次即可成事件，不该混进同一条聚合规则。
- DNSSEC 的签名否定应答让签名区可被枚举，[RFC 5155] 的 NSEC3 是为抗枚举而设计；这是数据完整性机制的取舍，与 DoT / DoH / DoQ 的传输加密无关，两者不能互相替代。
- 第 29 章是防守侧自查与加固（阈值表、nft 骨架、悬空 CNAME 指纹表）；信号最终喂给第 31 章的告警与第 29 章的检测模型。

34.13　参考资料

- RFC 1035, *Domain Names —Implementation and Specification*（区传送与记录类型基础）：https://www.rfc-editor.org/rfc/rfc1035
- RFC 5155, *DNSSEC Hashed Authenticated Denial of Existence*（NSEC3 抗枚举设计）: https://www.rfc-editor.org/rfc/rfc5155
- RFC 5358, *Preventing Use of Recursive Nameservers in Reflector Attacks*（开放递归解析器）: https://www.rfc-editor.org/rfc/rfc5358
- RFC 5737, *IPv4 Address Blocks Reserved for Documentation*（本书示例地址段）：https://www.rfc-editor.org/rfc/rfc5737
- RFC 2308, *Negative Caching of DNS Queries*（负缓存与 NXDOMAIN）: https://www.rfc-editor.org/rfc/rfc2308
- RFC 7208, *Sender Policy Framework (SPF)*（`include:` 链）: https://www.rfc-editor.org/rfc/rfc7208
- OWASP Web Security Testing Guide（授权测试纪律与 “Do not claim it”）: https://owasp.org/www-project-web-security-testing-guide/
- Unbound 官方文档（`access-control`、`hide-identity` / `hide-version` 等条目按名称检索）：https://unbound.docs.nlnetlabs.nl/en/latest/

第三十五章

DNS 蜜罐与被动观测

生产解析器的日志回答的是「我的用户问了什么」，蜜罐回答的是另一个问题：「谁在打听一个本来不该有人打听的名字」。前者的流量里混着大量正常业务，异常只占极小比例，告警必须靠基线和白名单才能压住误报；后者几乎不承载任何业务，落到它上面的每一次查询都自带可疑性。这个差别决定了两者的工程目标不同：生产解析面追求可用与低延迟，蜜罐追求**可归因、可留存、且绝不干扰生产**。把这两件事混在一台实例上做，是本章最想避免的错误。

蜜罐不是「把对手骗进来打」。它不主动发包、不回连、不试探对方端口；它只应答本来就会到达的查询，并把查询的形态记下来。DNSSEC 提供数据完整性与来源认证，DoT / DoH / DoQ 提供传输加密与隐私，二者互不替代——蜜罐里若给诱饵区签了名，也不等于查询链路被加密，这条全书的关键区分在这里同样成立。

35.1 三种形态：把 DNS 变成观测面

同一个「蜜罐」label 下其实有三种互不相同的部署，区别只在于**是否改写应答**以及**改写发生在哪一层**。先分清形态，再谈指标，否则很容易把「被观测」误当成「被拦截」。

35.1.1 假权威：对诱饵域应答，记录谁在查

在上级注册一个专门用于观测的诱饵域（例如 `bait.example.net`），把它的 NS 指向一台由自己控制、与生产完全隔离的权威实例。这个域不出现在任何业务配置、证书、邮件头或文档里，因此任何一次对它（或它的子域）的查询都值得记录。查询先到递归解析器，递归解析器按 NS 找到假权威，于是**假权威日志里看到的是递归解析器的源地址**，不是原始存根解析器——这是最容易被误读的一点。要归因到原始客户端，得靠 ECS（[RFC 7871]）里携带的子网信息，或者靠递归侧的 dnstap 流做关联，不能凭假权威日志直接下结论。

假权威的价值在于「谁在查」和「按什么节奏查」：名字枚举、字典爆破、NXDOMAIN 洪泛都会在这里留下干净的序列，没有业务噪声。

35.1.2 sinkhole：把已知坏域导向受控解析器

sinkhole 不改权威、不注册新域，而是在**自己的递归解析面**上用 RPZ（response policy zone）把已经确认的恶意域、DGA 域、隧道域改写到一个受控地址，例如 `192.0.2.54` 上的一台只记录不转发的解析器。改写之后，被控主机以为自己拿到了应答，实际连向的是观测点；观测点因此能看到回连的周期性、承载 QTYPE、以及同一批域名的轮换规律。

这里必须写清边界：RPZ 是**事后处置**，只用于已确认的域，不是控制措施，也不是靠封单个域就能解决 DGA（第 29 章的结论）。本章用 RPZ 的目的甚至不是「阻断」，而是「把已确认的域导向一个能记日志的地方」。把观测目标和处置目标分开，才不会在误报时把真实客户端一起导向死路。

35.1.3 被动 DNS 采集点：只观测不改写

第三种形态完全不参与应答路径：在递归解析器旁挂一个 dnstap 消费端或镜像口，把「某名字在某时间被查过、返回了什么 RDATA」记成历史索引，不改写任何应答、不新增任何区。它回答的是「这个名字第一次/最后一次出现是什么时候」「和它一起出现的还有哪些名字」，是资产自查与事件回溯的底稿（第 29 章把被动 DNS 列为暴露面排查的四类来源之一）。

被动采集点的覆盖面完全取决于采集点位置：看不到未经过该采集点的查询，也看不到被加密传输绕过去的查询。它是三种形态里侵入性最低、也最容易被「以为全都能看见」的一种。

35.1.4　形态对照

形态	是否改写应答	主要观测位置	能回答的问题	主要代价
假权威	是（对诱饵域给出受控应答）	隔离的权威实例	谁在查、按什么节奏查	需要持有域并维持 NS/胶水记录，日志里看到的多是递归器地址
sinkhole	是（RPZ 改写已确认域）	生产递归解析面 + 受控落点	恶意软件怎么回连、域名怎么轮换	改写会误伤，必须白名单与复核日期
被动 DNS 采集点	否	递归侧日志 / 镜像	名字的历史出现与关联	覆盖有盲区，日志本身就是全量查询流副本

三种形态可以并存，但**日志通道必须分开打标**：来自假权威的 `sensor=fake-auth`、来自 sinkhole 的 `sensor=sinkhole`、来自被动采集的 `sensor=passive`，否则事后无法解释「同一条查询被记了两次」是不是故障。

35.2　蜜罐究竟能观测到什么

蜜罐的输出不是「发现攻击」四个字，而是一组可以喂给指标系统的原始特征。下面按信号类型拆开。

35.2.1　扫描与枚举节奏

单源 qps 突增、随机子域集中出现、NXDOMAIN 比例抬升、ANY 或罕见 QTYPE 集中，是名字枚举阶段的典型形态（第 34 章侦察链第 2 步）。假权威的诱饵域没有合法流量，所以这里**不需要复杂基线**：查询数从 0 变 1 就值得记一条。反过来说，这也意味着任何一次误配置（例如监控探针把诱饵域写进了健康检查清单）都会立刻污染数据，白名单要从第一天就建。

35.2.2　DGA 与隧道的编码形态

DGA（domain generation algorithm）和 DNS 隧道都把数据塞进 QNAME，但形态不同：前者是**大量互不相同、按时间成批出现**的高熵标签，后者是**同一父域下长度高度一致的长标签**、常伴 TXT / NULL / CNAME 一类罕见 QTYPE。单标签上限 63 字节、完整 QNAME 约 253 字节，这个协议约束（[RFC 1034]、[RFC 1035]）让「接近上限且长度整齐」成为很稳的指纹。第 29 章已经给过长 QNAME、唯一 FQDN 数、按域聚合的定义，本章不重列阈值，只强调蜜罐里这些特征的**信噪比远高于生产面**：生产面里 CDN 健康探测和证书续期会产生合法长 QNAME，诱饵域里不会。

35.2.3　缓存探测与解析器能力指纹

对手常常先摸清解析器再下手：发非递归查询看是否命中缓存（cache snooping）、问 CH 类 TXT 探版本、用不同 EDNS 选项试探能力边界。蜜罐侧的对应观测项是：

- **RD 标志与命中/未命中的时序**：非递归查询直奔假权威，往往是缓存探测而非正常解析；
- **ECS**：[RFC 7871] 的 Client Subnet 选项会带出客户端子网前缀，可用来区分「同一个递归器背后有多少不同来源」；但它同时是准个人信息，采集时只保留必要的截断前缀或哈希值；
- **DNS Cookie**：[RFC 7873] 一类机制是否出现、是否稳定复用，能辅助判断请求是否来自同一解析器实例；互操作细节见官方资料，本章不作实现断言；
- **EDNS(0) 版本、UDP 载荷上限、DO 标志**：[RFC 6891] 定义的能力集合，配合第 8 章「`edns-buffer-size: 1472` 会在 MTU 1280 与 PPPoE 上分片」的结论，能看出对端是不是一个配置过的成熟解析器；
- **扩展错误码**：[RFC 8914] 一类 EDE 信息只在特定实现上出现，遇到就按「见官方资料」处理，不要凭它反推产品版本号。

35.2.4　AXFR 尝试与区结构探测

AXFR（[RFC 5936]）本该只允许从授权的次级权威发起。一旦有人对诱饵域发 AXFR，几乎可以断定是在做区结构探测（第 34 章第 3 步、第 29 章的 AXFR 误开自查）。蜜罐应当**拒绝但记录**：记下时间、源、是否带 TSIG、请求的是哪个区，而不是把区内容交出去。同理，NS / SOA 的密集查询、NSEC / NSEC3 遍历形态（[RFC 5155] 的 NSEC3 本就是为抗枚举而设计）都属于同一阶段的信号。

35.2.5 恶意软件回连规律

sinkhole 最擅长的不是抓单条查询，而是抓**周期**：同一批被控主机按固定间隔回连、域名按种子轮换、失败后按退避重试、每次回连的 QTYPE 与标签长度几乎不变。这些规律要按「域 × 时间窗 × 唯一源」聚合才看得出来，单条日志没有意义。也正因如此，采集端不能做采样——第 31 章已经论证过「反对采集端采样」，蜜罐侧同理：采掉 90% 之后，周期性和唯一源数都不可复原。

35.2.6 观测项与本书章节的对应

观测项	直接信号	交给哪一章的消费端
扫描节奏	单源 qps、NXDOMAIN 比、随机子域	第 29 章检测指标、第 31 章告警
DGA / 隧道编码	长标签、高熵、唯一 FQDN 数、QTYPE 分布	第 29 章流量分析、第 31 章「隧道可疑度」
缓存探测	RD=0、CH 类查询、命中时序	第 9 章 hide-identity/version、第 34 章第 4 步
解析器能力	ECS / Cookie / EDNS / DO / EDE	第 8 章 EDNS 缓冲、第 33 章客户端来源
区结构探测	AXFR 尝试、NS/SOA 密集、NSEC 遍历	第 34 章第 3 步、第 29 章 AXFR 自查
回连规律	周期性、域名轮换、重试退避	第 31 章基线告警、第 34 章防守落点

35.3 不能做什么

这一节单列，因为蜜罐天然处在「合法观测」与「越界采集」的边界上。以下四条是硬约束，不是建议：

- **不主动攻击**。蜜罐不应答之外的动作一律不做：不回连查询源、不扫描对方端口、不向对方权威服务器发 AXFR、不做 DNS 反射或投毒测试、不把观测到的地址拿去当扫描目标。它是被动的温度计，不是反击平台。第 32–34 章的授权纪律在这里同样适用。
- **不采集与业务无关的个人信息**。日志只保留排查所需的协议字段：时间、源（截断或哈希后）、QNAME、QTYPE、QCLASS、标志位、RDATA 摘要、传感器标记。不记用户名、设备标识、浏览器指纹、邮件地址；ECS 携带的客户端子网只保留必要前缀，不落全量。诱饵域的查询内容本身可能含有被外泄的数据片段，因此**留存即风险**，字段越窄越好。
- **不把查询内容外发第三方**。查询日志是自己的全量查询流副本（第 7、31 章已明确），不得无条件推给第三方 SaaS、公共被动 DNS 服务或在线分析接口；汇总指标可以出内网，原始 QNAME 不出。若组织已有集中日志平台，也必须先确认它属于同一信任边界，而不是「顺手接过去」。
- **日志留存期与访问范围写进配置，不靠口头约定**。沿用第 7 章的 dnstap **30 天**保留期口径，蜜罐日志、索引、副本、快照、备份一并对齐 30 天；访问范围限定到观测角色，读权限与写权限分离，谁在何时读过要留审计。到期自动删除，不要「先留着以后再说」。

还有一条工程上的「不能」：**不能让蜜罐成为生产链路的一环**。诱饵域的 NS、sinkhole 的落点、被动采集的旁路，都不许出现在生产递归解析器的上游或转发链里。这一条在后面的故障域隔离小节展开。

35.4 与本书其他章的闭环

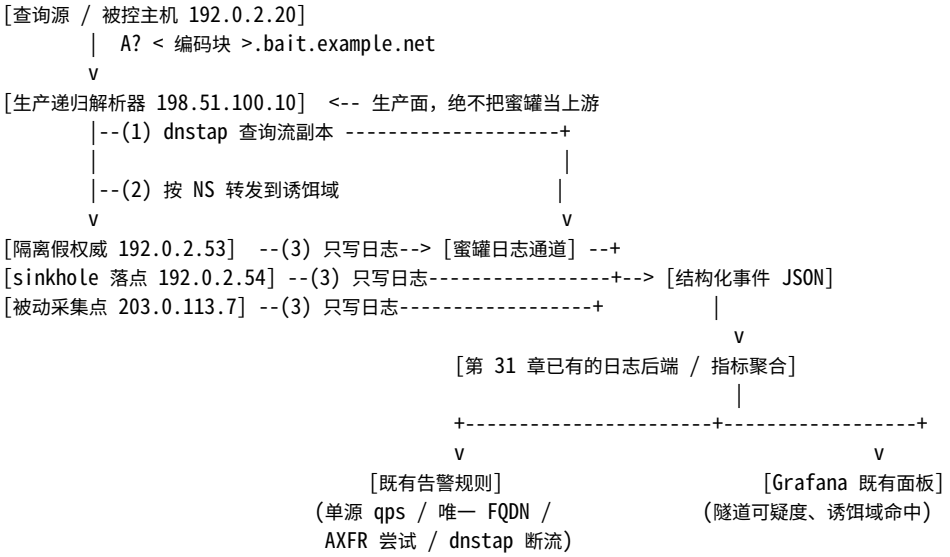
蜜罐不是孤立的新系统，它接的全是本书已经建好的东西。四节分别说清数据从哪来、到哪去、能解释什么、以及不能碰什么。

35.4.1 高质量数据源：第 29 章的隧道与扫描指标

第 29 章定义的长 QNAME、高熵标签、唯一 FQDN 数、单源 qps、NXDOMAIN 比例，在生产面上要靠 7 天基线才能区分 CDN 健康探测与真异常。蜜罐把这些指标的**分母清成了零**：诱饵域没有合法查询，于是「唯一 FQDN 数突然变成几十」不再需要统计判断。蜜罐因此不是第 29 章的替代品，而是给它的一条高质量输入——用它来校准阈值、验证白名单是否会误吞真信号，比在生产面上试错安全得多。

35.4.2　同一套告警：第 31 章的 dnstap 与 Prometheus

蜜罐不新建一套告警。它按第 31 章已经存在的采集链路，把结构化事件交到同一个日志后端、同一组指标 label、同一套告警规则里。下面只给**数据流图与字段契约**，不重复第 31 章的 Prometheus 抓取配置、Grafana 面板 JSON、eBPF 探针、dnstap 采集器配置与保留期设置。



图上三条 (3) 通道汇成同一种事件格式，是集成的关键：告警规则只认识字段，不认识「这是哪台蜜罐」。下面是一组**教学用 JSON 字段示例**，字段名是本书为归纳而定的，不是 dnstap 或任何产品的官方 schema（官方字段以 dnstap.info 为准）。

```
{
  "ts": "2026-11-07T09:14:22Z",
  "sensor": "fake-auth",
  "sensor_addr": "192.0.2.53",
  "zone": "bait.example.net",
  "qname": "x7f3c9e1b2a.tunnel.bait.example.net",
  "qtype": "TXT",
  "qclass": "IN",
  "flags": {"rd": false, "do": false, "edns": true, "udp_payload": 1232},
  "ecs_prefix": "198.51.100.0/24",
  "cookie_seen": true,
  "src_kind": "recursive-resolver",
  "rcode": "NXDOMAIN",
  "axfr_attempt": false,
  "dga_entropy": 4.61,
  "tunnel_score": 0.83,
  "unique_src_5m": 3,
  "retention_days": 30
}
```

字段设计上有三个刻意之处：`sensor` 让三条通道可区分；`src_kind` 明确提醒「源多半是递归解析器而非原始客户端」，避免归因错误；`retention_days` 把留存期写成数据的一部分，而不是留在运维文档里靠人记。`qname` 一类的原始内容**只落在本地后端**；出内网的汇总指标里只保留 `sensor`、`qtype`、计数与分数，不带 `qname`、不带 `ecs_prefix`。

35.4.3　对手在哪一步：第 34 章的侦察链

第 34 章把侦察拆成被动收集、名字枚举、区结构探测、解析器探测、信任记录枚举、内网侧六步。蜜罐能看见的不是全部，而是**会向 DNS 发出查询的那几步**：第 1 步的被动收集不留任何查询到目标，蜜罐天然看不见；第 2 步枚举落在假权威的随机子域序列里；第 3 步的 AXFR / NS / SOA 探测落在假权威与区日志里；第 4 步的缓存窥探与 CH 类指纹落在递归侧的 dnstap 里；第 5 步的 SPF / DKIM / DMARC / TLSA / CAA 若属于诱饵域，同样会留下读取痕迹。换句话说，蜜罐日志可以给「对手走到哪一步」排一个时间线：先密集枚举、再转 AXFR、随后开始解析器能力试探，是相当清晰的递进。这条时间线与第 34 章的防守价值表正好互补——那边讲「每个阶段留下什么信号」，这边讲「哪个传感器真的收到了」。

35.4.4 硬前提：第 28 章的故障域隔离

第 28 章的结论在蜜罐场景里要读成一条禁令：**蜜罐必须与生产解析面隔离**。具体是四层：

- **地址与实例隔离**：假权威 192.0.2.53、sinkhole 192.0.2.54、被动采集点 203.0.113.7 与生产递归 198.51.100.10 不共用进程、不共用缓存、不共用配置目录；主机名只是教学标记。
- **链路隔离**：生产解析器的 forward-zone / stub-zone / 上游清单里绝不出现蜜罐地址。否则诱饵域的高熵查询洪泛会先打满生产缓存，再让负缓存（[RFC 2308]）把 NXDOMAIN 按 SOA minimum 扣住，业务查询跟着遭殃——这正是第 7 章「负缓存会放大故障」的另一副面孔。
- **容量隔离**：蜜罐日志卷独立配额、独立磁盘，写满只影响自己；日志通道限速与丢弃策略要「宁丢蜜罐、不丢生产」。
- **回滚隔离**：停用蜜罐的单点动作是撤掉诱饵域的 NS 委派或删掉 RPZ 条目，不需要重启生产解析器，也不需要刷生产缓存。

一句话检验：把蜜罐整台关掉，生产解析的命中率、SERVFAIL 率、上游 RTT 三条曲线应当没有任何变化。做不到，说明隔离没做完。

35.5 工程骨架（教学用）

下面五块是最小骨架，全部使用文档地址段。它演示的是「边界怎么划」，不是可直接上生产的完整部署。

35.5.1 隔离的权威实例

NSD 与 CoreDNS 都能扮演假权威；这里给出两种等价形态，只保留与隔离相关的行。

```
# /etc/nsd/honeypot.conf ——教学用，NSD 形态
server:
    ip-address: 192.0.2.53
    ip-address: 2001:db8:35::53
    do-ip6: yes
    port: 53
    hide-version: yes
    verbosity: 1
    # 不提供递归，不对非诱饵域应答；不配任何转发/上游
zone:
    name: "bait.example.net"
    zonefile: "/etc/nsd/zones/bait.example.net.zone"
    # 不写任何 provide-xfr: AXFR 放行名单为空
    # 「拒绝但记录」在各实现里的具体写法与日志字段见官方资料
```

```
# Corefile ——教学用，CoreDNS 等价形态
bait.example.net:53 {
    file /etc/coredns/zones/bait.example.net.zone
    log
    errors
}
```

为什么这么设：hide-version: yes 与「不递归」把假权威的能力面压到只剩诱饵区；第 6 章曾用 provide-xfr 显式放行回环地址来建根区镜像，这里反过来——一条都不配，AXFR 放行名单为空，请求应被拒绝并留下审计痕迹，这是本章要的信号，不是要交出区内容。各实现的拒绝码与日志字段见官方资料。CoreDNS 形态里 file 插件只读区文件、log 只记查询，效果一致。两者都不接任何上游，因此不可能被当跳板。

怎么验证：dig @192.0.2.53 bait.example.net SOA +short 应返回该区的 SOA，而 dig @192.0.2.53 www.example.org A +short 不应给出可用的递归应答（只可能是 REFUSED / 无应答）。

失败模式 / 回滚点：误把 provide-xfr 写成放行任意地址，或给实例配了转发上游，等于把一台权威变成开放解析器/跳板，立即停实例并恢复配置。回滚点就是撤销诱饵域的 NS 委派：委派一撤，查询不再到达，生产侧无需任何改动。

35.5.2 只写不读的日志通道

```
# 教学用：日志通道的权限与生命周期契约
channel:
  path: /var/log/dns-honeypot/events.jsonl
  mode: "0240"                # owner 为 -w-（只写），group 为 r--（只读）
```

```
  owner: honeypot-writer        # 属主位生效，即使同属 group 也不能读
  group: sensor-readers         # 只有这一个组能读，且不能写
  append_only: true             # 追加写，蜜罐进程不持有读句柄
  quota: 20GiB                  # 独立卷，写满不影响生产
  retention_days: 30            # 与第 7/31 章 dnstap 口径一致
  export: none                  # 不向第三方推送原始 QNAME
  audit: true                   # 谁在何时读过，单独留审计
```

为什么这么设：「只写不读」是防篡改也是防越界——蜜罐进程被攻陷后无法回读历史查询，更无法把历史数据带出去。`export: none` 把「不外发第三方」写成配置而非口号。独立配额保证日志写满时不会连累生产解析器所在的磁盘。

怎么验证：`sudo -u honeypot-writer sh -c 'cat /var/log/dns-honeypot/events.jsonl'` 应被拒绝；`ls -l /var/log/dns-honeypot/events.jsonl` 应显示属主为 `honeypot-writer`、权限含 `-w-r-----`，且到期清理任务生效后 `find /var/log/dns-honeypot -mtime +30 -type f` 为空。

失败模式 / 回滚点：最常见的失败是「为了方便」把采集进程也塞进 `honeypot-writer`，于是只写不读名存实亡；其次是有人临时开了 rsync / 日志转发到外部平台却没登记。发现即停写、撤销外部转发、把该时段标记为不可信。回滚点是清空该通道重新按契约挂载，历史数据按 30 天策略本来也不该长期保留。

35.5.3　RPZ 把已确认域导向 sinkhole

```
# 教学用：生产递归侧 RPZ 片段（配置形态见第 29 章，此处只给导向意图）
rpz:
    name: "sink.example.org"
    zonefile: "/etc/unbound/rpz/sink.zone"
    # 仅列入已确认域；每条带复核日期
```

```
; /etc/unbound/rpz/sink.zone ——教学用；地址为 sinkhole 落点
$ORIGIN sink.example.org.
$TTL 60
; RPZ 的 owner 是「要匹配的 QNAME」相对本 RPZ 区名，故用相对写法
dga-seed-01.example.net    IN  A      192.0.2.54
```

为什么这么设：RPZ 只装**已确认**的域，且用一条 A 记录把 QNAME 直接改写到受控落点 `192.0.2.54`。注意 RPZ 里 owner 名是「被匹配的 QNAME + RPZ 区后缀」，所以这里写成相对名而不是带尾点的绝对名——写错成绝对名是最常见的静默不命中。TTL 取短值（这里 60 秒）是为了让误伤能被快速撤回：负缓存与 RPZ 命中都可能被中间层缓存，短 TTL 缩小影响窗口（第 7 章）。每条条目的复核日期写在变更单里，避免过期 IOC 长期误伤。

怎么验证：`dig @198.51.100.10 dga-seed-01.example.net A +short` 应返回 `192.0.2.54`；`dig @198.51.100.10 www.example.org A +short` 必须仍是正常业务应答，用来确认白名单没被绕过。若第一条拿到的是真实地址，先查 RPZ 区是否已加载、owner 名是否误写成绝对名。

失败模式 / 回滚点：最大风险是把合法域写进 RPZ，真实客户端被导向 sinkhole。回滚点是**删条目 + 按控制面到客户端逐层刷缓存**（第 7 章强调链路上每个实例都要刷），而不是只删文件。第 29 章已明确 RPZ 是事后处置、不是控制措施，本章不为「提升覆盖率」而批量导入未核实清单。

35.5.4　诱饵记录的 TTL 与缓存策略

```
; /etc/nsd/zones/bait.example.net.zone ——教学用
$ORIGIN bait.example.net.
$TTL 60
@    IN  SOA  ns1.bait.example.net. hostmaster.example.net. (
              1 3600 600 604800 60 )
     IN  NS   ns1.bait.example.net.
ns1  IN  A    192.0.2.53
     IN  AAAA 2001:db8:35::53
@    IN  A    192.0.2.53
*    IN  A    192.0.2.53        ; 通配应答：任何子域都给出受控地址
```

为什么这么设：`$TTL 60` 与 SOA minimum 取小值，让「查询→应答→缓存」的窗口短，后续每一次回连更可能真的到达假权威，从而被逐条记录；若 TTL 设成几小时，递归器命中缓存后蜜罐就**看不见**后续查询了，这是蜜罐特有的「缓存把自己饿死」问题。通配记录让随机子域也有受控应答，避免因 NXDOMAIN 而让对手换一批名字重试、打乱节奏观测。

怎么验证：连续两次 `dig @192.0.2.53 random-abc.bait.example.net A` 后看权威日志应各有一条；用 `dig @198.51.100.10 random-abc.bait.example.net A` 观察 TTL 是否按 60 秒递减，确认缓存窗口符合预期。

失败模式 / 回滚点：TTL 过小会让被大量枚举时查询量全部压到假权威，虽不影响生产但仍要给它自己的限速；TTL 过大则观测出现「空窗」。另一个坑是负缓存：若曾返回 NXDOMAIN，[RFC 2308] 的缓存会让后续同名查询不到达权威，排查时按第 7 章逐层刷缓存。回滚点是改回较长 TTL 或撤掉通配记录。

35.5.5 白名单：避免误伤真实客户端

```yaml
# 教学用：蜜罐侧的豁免与复核清单
whitelist:
  - name: health-probe.example.net
    reason: 监控探针的固定健康检查
    reviewer: sensor-readers
    review_after_days: 30
  - cidr: 198.51.100.10/32
    reason: 自有生产递归解析器，其转发查询不代表原始客户端
    reviewer: sensor-readers
    review_after_days: 30
  - cidr: 203.0.113.7/32
    reason: 被动采集点自身的元数据查询
    reviewer: sensor-readers
    review_after_days: 30
rules:
  - 白名单只影响「是否计入告警」，不影响「是否记录原始事件」
  - 每条必须有 reason / reviewer / review_after_days
  - 演练白名单与永久白名单分开，演练后自动失效
```

为什么这么设：白名单要窄、要登记、要复审（第 31 章的治理纪律）。特别地，自有生产递归解析器的地址必须豁免——它转发的查询源是它自己，不豁免会把全部内部客户端的查询误算成「一个高 qps 攻击源」。第 31 章还强调过：白名单只豁免告警、**不豁免记录**，否则等于给自己开了一段不可观测区。

怎么验证：从 198.51.100.10 发一条查询到诱饵域，确认 `events.jsonl` 里有该事件但告警规则未触发；`grep -c 'review_after_days' whitelist.yaml` 应等于条目数，确认每条都有复审日期。

失败模式 / 回滚点：最常见的失败是用「整段网段 + 永久」图省事，把真信号一起吞掉；其次是演练白名单忘了删，下一轮真实演练全部无效。回滚点是回滚清单文件并重新加载采集端，已记录的原始事件不受影响——这也再次说明「记录与告警分离」的价值。

35.6 指标与判读

蜜罐只需要六个指标，多了反而没人看。字段名沿用本章 JSON 示例，指标名随所选实现而变，本节只给语义。

指标	含义	判读要点	主要误报来源
诱饵域被查次数	`sensor=fake-auth` 的事件总数	从 0 变 1 即有效信号，无需基线	监控探针误把诱饵域写进健康检查
唯一源数	按 `ecs_prefix` / 源聚合的去重计数	突增常对应分布式枚举或一批新被控主机	同一递归器 NAT 后多出口、ECS 缺失
QTYPE 分布	A / AAAA / TXT / NULL / AXFR 等占比	TXT、NULL、ANY、AXFR 抬升比总量更说明问题	证书续期类合法 TXT、运维自查
DGA 熵值	标签字符分布的熵（示例字段 `dga_entropy`）	与标签长度一致性一起看，单看熵会误判	CDN 的随机化子域、合法长名字
隧道嫌疑分	由长标签、唯一 FQDN 数、QTYPE、节奏合成的分数	只用于排序与复核，不自动写 RPZ	与第 29 章同源：健康探测与批量任务
AXFR 尝试数	`axfr_attempt=true` 的事件数	对诱饵域几乎不存在合法 AXFR，出现即高优先级	自有次级权威误配、备份脚本

三条判读纪律：其一，**按域按时间窗聚合**，单条查询不判案；其二，**分数只排序不处置**，是否写 RPZ 必须人工复核并登记

复核日期（第 29 章）；其三，**先测基线再定阈值**，第 31 章的 7 天基线方法照用，但蜜罐基线可以短得多，因为分母本来就接近零。另外要单独给「蜜罐日志断流」留一条告警：传感器静默时「没告警」不等于「没查询」，这一点与第 31 章对 dnstap 断流的要求完全一致。

35.7　演练：三条链路都要有信号

演练的目的不是证明「蜜罐能用」，而是证明**观测链路没有断点**。以下两项注入都只对读者自己持有、且已取得书面授权的诱饵域与蜜罐实例执行，不对任何第三方权威服务器发起。

35.7.1　高熵 QNAME 批次注入

```
# 教学用：仅向自有假权威注入，域名均为示例域
for i in $(seq -w 1 60); do
  dig @192.0.2.53 "x${i}a9f3c7e1b.tunnel.bait.example.net" TXT +short >/dev/null
done
# 链路①：蜜罐日志应出现 60 条 sensor=fake-auth 事件
jq -c 'select(.zone=="bait.example.net")' /var/log/dns-honeypot/events.jsonl | tail -5
# 链路②：第 31 章既有的 dnstap 消费端应看到同一批唯一 FQDN
# 链路③：Grafana 既有「隧道可疑度」面板应在同一 UTC 时间窗出现凸起
```

为什么这么设：60 个互不相同、标签高熵的 TXT 查询几乎不抬高总 qps，却把「按域唯一 FQDN 数」与 `dga_entropy` 一起推高——这正是第 29 章「流量分析比 payload 分析更通用」的复现。用 TXT 而不是 A，是为了同时让 QTYPE 分布产生可辨认变化。

怎么验证：`jq -r 'select(.zone=="bait.example.net") | .qname' events.jsonl | sort -u | wc -l` 应接近 60；`unique_src_5m` 不应异常放大（同一源注入），可据此确认「唯一源数」与「唯一 FQDN 数」没有被混成一个指标。

失败模式 / 回滚点：采集端若做了采样，唯一 FQDN 数会明显小于 60，这是反对采样的实证；若曾返回 NXDOMAIN，负缓存可能让后续同名查询不再到达权威，按第 7 章逐层刷缓存并把该域加入**演练白名单**（不是永久白名单）。注入本身无状态，停跑即回滚。

35.7.2　AXFR 尝试模拟

```
# 教学用：只对自己持有并已授权的蜜罐权威实例发起，用于验证拒绝与记录
dig @192.0.2.53 bait.example.net AXFR
# 预期：被拒绝；events.jsonl 里出现一条 axfr_attempt=true 的事件
jq -c 'select(.axfr_attempt==true)' /var/log/dns-honeypot/events.jsonl | tail -1
```

为什么这么设：这里验证的是「拒绝但记录」这条边界是否真的生效——如果区内容被返回，说明 AXFR 放行名单配错，蜜罐从观测点变成了泄露点；如果被拒绝却没记日志，AXFR 尝试数这个指标就是死的。命令只指向本例中的自有实例，不构成对第三方的区传送测试，也不提供任何绕过授权限制的方法。

怎么验证：应答应为拒绝/失败而非区内容（拒绝形态按所选实现见官方资料）；`jq` 应能取到带 `ts`、源、`zone`、`axfr_attempt: true` 的一条事件，且第 31 章既有告警规则中「AXFR 尝试」一条进入 pending/firing。

失败模式 / 回滚点：若区内容被返回，立刻停实例、清空 `provide-xfr` 放行名单并撤销 NS 委派，同时按「疑似配置泄露」登记时段；若日志没记，检查日志通道配额与 `append_only` 是否把该事件丢了。演练后把该时段标注为测试数据，避免污染下一轮基线。

35.7.3　三条链路验收矩阵

注入	蜜罐日志	dnstap（第 31 章既有链路）	Grafana（第 31 章既有面板）	合格标准
高熵 QNAME 批次	60 条 `sensor=fake-auth, qtype=TXT`	同一批唯一 FQDN 可检索	隧道可疑度同窗凸起	三条链路时间窗对齐到同一 UTC 锚点
AXFR 尝试	1 条 `axfr_attempt=true`，应答被拒	递归侧可见对应查询	「AXFR 尝试」告警触发	区内容未返回，且告警不是靠手工补录

注入	蜜罐日志	dnstap（第 31 章既有链路）	Grafana（第 31 章既有面板）	合格标准
停用蜜罐（附加检验）	停止写入	无新增蜜罐事件	生产面三条曲线无变化	证明第 28 章隔离确实成立

三项都通过，才能说这套蜜罐「可观测」；只有蜜罐日志有信号而告警没响，说明字段契约没接进第 31 章的既有规则；只有告警响而日志查不到明细，说明采集端做了采样或留存期被截断。演练记录连同 UTC 锚点、阈值依据、白名单变更一起归档，作为下一轮基线输入（第 31 章的验收方法）。

35.8 伦理与授权边界

这一节单列，因为蜜罐比普通日志系统更容易越线：它记录的是「别人在查什么」，而查询内容可能夹带个人信息、内部主机名乃至被外泄的数据片段。

- **书面授权先行**。只对自己持有、或已取得书面授权的名字与网段部署诱饵域与 sinkhole；不对第三方域名做委派劫持、不把别人的名字导向自己的落点。演练里的枚举、AXFR、能力探测只在自有实例上做，scope 写进变更单，含时间窗与源地址。
- **最小必要采集**。字段清单按「不能做什么」一节收窄，采集前先问「这条字段少了还能不能判案」，答「能」就删掉。ECS 子网、Cookie、QNAME 都属敏感项，各自要有明确的截断或留存理由。
- **不做反击、不做接管**。观察到可接管的悬空记录、误开 AXFR、暴露的凭据，只作为防守线索上报与止损，不 claim、不利用、不顺着查过去（第 29、34 章同一纪律）。蜜罐的输出是告警与报告，不是目标清单。
- **不外发、不长期留存**。原始 QNAME 不出信任边界；30 天到期自动删除，索引、副本、快照、备份一起对齐；读权限限定观测角色并留审计。
- **可解释与可撤回**。任何一条告警都要能回溯到 `sensor` 标记与原始事件；被误导向 sinkhole 的真实客户端要有明确的撤回路径（删 RPZ 条目 + 逐层刷缓存）。第 31 章的「记录与告警分离」在这里尤其重要：白名单豁免告警，但绝不豁免记录，否则撤回之后连复盘材料都没有。

把这几条写成部署前检查表，比事后补一份合规说明有用得多。

35.9 与其他章的交叉引用

- 第 7 章：dnstap 采集点与 **30** 天保留期口径，本章沿用不重复；第 9 章：`hide-identity` / `hide-version`、`access-control` 的能力收敛思路在假权威上同样适用。
- 第 28 章：故障域隔离与注入验收方法；第 29 章：隧道 / 扫描指标定义与「RPZ 只处置已确认域」。
- 第 31 章：Prometheus / Grafana / dnstap 采集器 / eBPF 的配置、告警规则与面板 JSON，本章一律不重复，只定义蜜罐侧交出的字段与流向；第 34 章：侦察链六步与授权测试纪律，本章负责说明哪几步真的会被传感器收到。
- 第 33 章：客户端从哪里拿到解析器地址，决定了 ECS 里能看到的是哪一层来源；第 8 章：EDNS 缓冲与分片结论用于解读 `udp_payload` 字段。

35.10 本章要点

- **三种形态按「是否改写应答」区分**：假权威对诱饵域应答并记录谁在查、sinkhole 用 RPZ 把已确认域导向受控解析器、被动 DNS 采集点只观测不改写；三者日志要用 `sensor` 标记分开，否则重复记录无法解释。
- **蜜罐的价值是信噪比，不是新指标**：第 29 章的长标签、高熵、唯一 FQDN 数、单源 qps 在诱饵域上分母近乎为零，所以它是校准阈值的高质量数据源；AXFR、ECS / Cookie / EDNS、回连周期都能观测，但假权威看到的多半是递归解析器地址，不能当成原始客户端归因。
- **四条不能做是硬约束**：不主动攻击、不采无关个人信息、不把查询内容外发第三方、留存期与访问范围写进配置（沿用第 7、31 章的 30 天口径）。「只写不读」的日志通道与「记录与告警分离」的白名单，是把约束落到工程上的两个抓手。
- **闭环靠字段契约，不靠新一套告警**：数据流图里的三条只写通道汇成同一份教学用 JSON（`sensor` / `src_kind` / `retention_days` 是刻意设计），接进第 31 章既有的 dnstap 日志后端与 Grafana 面板。
- **隔离是硬前提**：第 28 章的故障域隔离要落实为地址、链路、容量、回滚四层独立；检验标准是「关掉整台蜜罐，生产三条曲线不变」。TTL 取小值是为了避免缓存把蜜罐自己饿死，RPZ 短 TTL 与复核日期是为了让误伤可快速撤回。

- **验收靠注入，伦理靠检查表**：注 60 条高熵 TXT、模拟一次仅针对自有授权实例的 AXFR 尝试，逐项确认蜜罐日志 / dnstap / Grafana 三条链路在同一 UTC 锚点都有信号；书面授权、最小必要采集、不反击不接管、不外发不长期留存、可解释可撤回五条写成部署前检查表。

35.11 参考资料

- Unbound 官方手册，Response Policy Zones 专题（RPZ 用于已确认域的改写用法）：https://unbound.docs.nlnetlabs.nl/en/latest/topics/filtering/rpz.html
- Unbound 官方手册，`unbound.conf(5)`（`access-control`、`hide-version`、EDNS 与 RPZ 相关选项及默认值）：https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html
- NSD 官方手册，`nsd.conf(5)`（区定义、`provide-xfr` / `request-xfr`、隐藏版本）：https://nsd.docs.nlnetlabs.nl/en/latest/manpages/nsd.conf.html
- CoreDNS 官方插件目录（`file` 插件与 `log` / `errors` 的形态以官方说明为准）：https://coredns.io/plugins/
- dnstap 官方站点（消息类型、帧格式与字段定义以此为准，本书 JSON 为教学归纳）：https://dnstap.info/
- RFC 5737，文档用 IPv4 地址块：https://www.rfc-editor.org/rfc/rfc5737
- RFC 5936，DNS Zone Transfer Protocol (AXFR)：https://www.rfc-editor.org/rfc/rfc5936
- RFC 7871，Client Subnet in DNS Queries：https://www.rfc-editor.org/rfc/rfc7871

第三十六章

DNS 故障排查工具箱

一台家庭网关加一台云主机的实验环境里，有个发版前做连通性探测的小脚本：对 `api.example.net` 调 `socket.getaddrinfo(host, 443)`，只试 `infos[:3]` 这前三个地址逐个发起 TCP 连接，全失败就记一条「TCP 连不上」。`getaddrinfo` 返回的九个地址里，前四个是 IPv6，在该链路上被对端直接 RST（`ECONNREFUSED 111`），后面五个 IPv4 全通；浏览器照常走 IPv4 打开了页面。于是日志里出现了一条很难解释的曲线：报故障，4 秒后「网络已恢复」，隔一会儿又报故障。网络从头到尾没坏过，坏的是探测脚本只看了排序列表的前三个。

这不是工具不会用，是**没分清哪一层在骗你**。`getaddrinfo` 给的是策略排序后的候选列表，[RFC 6724] 的默认地址选择规则会让双栈域名解析出的 IPv6 地址排在前面；它从来不保证「第一个最可用」，更不保证「前几个里一定有一个能连上」。名字解析成功了、TCP 连接失败了、上层页面却打开了，这三件事分别发生在三个不同的层，用一条「网络不通」概括，就会把告警和后续排查全带偏。第 19 章的源地址选择与临时地址寿命、第 22 章的前缀轮换故障、第 31 章的告警误报治理，都是同一条线索的不同切面。

从这个案例能抽出三条贯穿全章的结论：

1. **探测要全量遍历，不要取前几个**。对 `getaddrinfo` 的返回先去重，再遍历全部地址（或至少显式 v4 优先），任一地址连通即判定为通；只试 `infos[:3]` 这类切片等于把地址排序策略当成了可用性结论。
2. **「网络已恢复」必须基于重探测**。上层页面能打开，可能只是浏览器换了个地址族、换了条连接、甚至命中了缓存；不重新跑一遍同一套探测并记录实际连通的地址，这条日志不构成恢复证据。
3. **`getaddrinfo` 是策略排序，不是可用性排序**。应用拿到列表后怎么选、失败后怎么回退，由应用自己决定（Happy Eyeballs 一类机制），不由解析器决定。排查时要把「名字给了哪些地址」和「应用试了哪些地址」分开看。

本章不按工具名字母序罗列命令，而是**按排查顺序**串一遍：谁在解析→解析器怎么答→抓到什么包→内核动过没→应用怎么连→归属与边界。全书前三十五章里 `dig`、`tcpdump`、`unbound-control`、`nft` 已经散落出现过几十次，这里做的是系统串讲与决策表，不重复各章的完整配置块。

36.1 第一层：谁在解析

用户说「DNS 挂了」，第一件事不是去 ping，而是确定**这个进程到底把名字交给了谁**。同一台机器上，浏览器、curl、Python 脚本、systemd 服务完全可能走不同的解析路径；把路径搞错，后面每一条命令都在回答另一个问题。

36.1.1 从存根配置读起

```
cat /etc/resolv.conf
cat /etc/nsswitch.conf
cat /etc/hosts
cat /etc/gai.conf
```

`/etc/resolv.conf` 里的 `nameserver` 只是存根解析器（stub resolver）的上游地址，不等于「真正在做递归的那台机器」。若看到 `nameserver 127.0.0.53`，那是 systemd-resolved 提供的**本地存根**，它后面还挂着按接口分配的 DNS、路由域、DNSSEC 与 LLMNR 策略；`127.0.0.53` 自己不解析任何东西，只是把查询接过去再分发。此时直接 `dig @127.0.0.53` 看到的，是 resolved 加工过的结果，而不是应用拿到的原始链路。

systemd-resolved 环境里优先读它的状态：

```
resolvectl status
resolvectl query api.example.net
```

`resolvectl status` 会按网络接口列出 DNS Servers、DNS Domain、DNSSEC 等设置，能解释「为什么这台机器上两个进程问同一个名字得到不同答案」——分接口、分路由域的上游本来就可以不同。若本机没有装 `resolvectl`（本章的实测环境就没装），退回到 `/etc/resolv.conf` 加上按接口的配置文件，`resolvectl status` 的替代路径见所用发行版的官方手册；**不要因为命令不存在就假定系统里没有存根层**，很多容器与精简镜像只是把配套命令行工具裁掉了。

`/etc/nsswitch.conf` 的 `hosts:` 行决定名字先查哪儿：`files dns` 意味着先读 `/etc/hosts`，命中就根本不发 DNS 包。`hosts` 行里若出现 `mdns4_minimal`、`resolve`、`myhostname` 之类模块，解析路径还会再拐弯（第 27 章讲单播 DNS 与多播 DNS 的分界）。`/etc/hosts` 里留的一条旧记录，是「改了权威数据却怎么都不生效」最常见的原因。`/etc/gai.conf` 则影响 `getaddrinfo` 的地址排序（label / precedence 表），章首案例里 IPv6 排前面就与这一层及 [RFC 6724] 的默认策略有关；改过 `precedence` 的机器，排序会跟别的机器不一样。

36.1.2　getent 与应用走的是同一条路吗

```
getent hosts api.example.net
getent ahosts api.example.net
getent ahostsv4 api.example.net
getent ahostsv6 api.example.net
```

`getent` 走 NSS（Name Service Switch），是「以应用视角」取地址最贴近的一条只读命令：`hosts` 通常只给一个系列，`ahosts` 会给出 `getaddrinfo` 风格的多个地址，正好用来复现章首那个「前几个全是 v6」的场景。但要注意判读边界：**`getent` 可能根本不碰 DNS**——命中 `/etc/hosts`、命中缓存模块时就地返回了。因此 `getent` 成功不能证明 DNS 正常，`getent` 失败也不能直接推出 DNS 故障；它证明的是「NSS 这条链路的结果」。

想看应用程序真正调用了什么、拿到几个地址，最直接的办法是让它自己报出来：

```
import socket
infos = socket.getaddrinfo("api.example.net", 443,
                           type=socket.SOCK_STREAM)
for family, socktype, proto, canonname, sockaddr in infos:
    print(family.name, sockaddr[0], sockaddr[1])
```

这段只是把候选列表打全，不做连接。把它和 `getent ahosts` 对照，就能分开「NSS 给了什么」与「`getaddrinfo` 排序后给了什么」。若列表前几项全是 `AF_INET6` 而应用只试前三项，结论已经出来了——不需要再抓包。

怎么验证这一层：`getent ahosts api.example.net` 与上面 Python 脚本的输出逐项对照，再 `cat /etc/resolv.conf` 确认存根指向哪里。**失败模式 / 回滚点**：只读命令没有回滚问题；真正的风险是**误判**——把 `getent` 命中 `/etc/hosts` 的结果当成 DNS 应答，或者把 `127.0.0.53` 当成递归解析器。若为了测试临时改了 `/etc/hosts` 或 `/etc/gai.conf`，改前留一份 `cp -a` 备份，测完立刻还原；这两个文件是全局生效的，忘了还原会污染后面所有章节的实验结果。

这一层的出口：搞清楚名字交给了谁，才能进第二层去问那台解析器。若发现应用根本没走系统解析器（浏览器内置 DoH、App 内嵌 SDK、`local-zone` 接管），转第 12、13、24、29、32 章，不要继续往下查 DNS 服务器。

36.2　第二层：解析器怎么答

确定「谁在解析」之后，第二个问题是**那台解析器给出的答案是什么、以及这个答案从哪来**。`dig` 直连指定服务器、绕开本机存根；`+trace` 从根开始迭代，把委派链每一跳摊开；`delv` 与 `drill` 回答「签名对不对、为什么 bogus」；`dnstracer` 专门追委派链。这一层的分工是**查数据**，不是查连通。

36.2.1　dig：把查询路径拆开

```
dig @192.0.2.53 api.example.net A +short
dig @192.0.2.53 api.example.net A +trace
dig @192.0.2.53 api.example.net A +dnssec
dig @192.0.2.53 api.example.net A +cd
dig @192.0.2.53 api.example.net A +tcp
```

- `@server` 指定服务器，绕过 `/etc/resolv.conf`。这是它最大的价值，也是最大的陷阱（见「哪条命令会骗你」）。示例里的 `192.0.2.53` 是文档地址，实际换成自己的解析器地址。
- `+short` 只给 RDATA，适合塞进脚本；排障时别只用 `+short`，头部里的 `status`、`flags`、TTL 往往才是线索。

- `+trace` 从根迭代到权威，能定位「卡在根、卡在 TLD、还是卡在权威」；它走的是**本机发起的迭代查询**，与递归解析器自己的缓存路径不是一回事（部分 `dig` 版本会连带请求 DNSSEC 记录，选项细节见官方手册）。
- `+dnssec` 请求 DNSSEC 相关记录，看应答里的 `AD` 位与 RRSIG；`+cd`（checking disabled，某些版本写作 `+cdflag`）让上游跳过验证，用来区分「数据本身坏了」和「本地验证器把它判成 bogus」。AD 位是**验证状态**，不是加密传输的标志——DNSSEC 管数据完整性与来源认证，DoT / DoH / DoQ 管传输加密与隐私，二者不互相替代。
- `+tcp` 强制走 TCP/53。UDP 被丢与 TCP 能通，是两种完全不同的故障；[RFC 7766] 之后 TCP 是 DNS 传输的必备能力，具体参数见官方手册。

```
host -t A api.example.net 192.0.2.53
nslookup -type=A api.example.net 192.0.2.53
```

`host` 输出简洁，适合快速看一个记录。`nslookup` 早已不推荐用于诊断（它有自己的解析库与行为，不总跟随 `resolv.conf` 的全部路径），但几乎所有系统上都还在、交互模式也确实方便查多个 QTYPE，所以还得会读它的输出；只是**别拿它的结果当权威判据**。

36.2.2 delv：验签与「为什么 bogus」

```
delv @192.0.2.53 api.example.net A
delv @192.0.2.53 api.example.net A +rtrace
delv @192.0.2.53 api.example.net A +cd
```

`delv` 是 BIND 系工具，自带 DNSSEC 验证器并用自己配置的信任锚（trust anchor）独立验签，输出会给出 `fully validated` / `unsigned answer` / `validation failed` 一类结论。它可能返回 SERVFAIL 并附上原因（签名过期、缺 DS、链断在某一级、系统与权威时钟不同步导致 RRSIG 有效期不合法等），这正是 `dig` 只给一个 `SERVFAIL` 时不提供的信息。`+rtrace` 一类追链选项让验证过程可见，具体选项名与输出格式见所用版本的官方手册，不要凭记忆猜。

判读次序建议固定下来：`dig +dnssec` 看有没有 RRSIG / AD → `dig +cd` 看绕过验证后数据是否回来→ `delv` 看独立验证结论与失败原因。若 `+cd` 有答案、`delv` 报 bogus，故障在验证链（信任锚、时钟、上游剥离签名、`harden-dnssec-stripped` 一类设置），转第 10、16 章；若 `+cd` 也没答案，故障在数据或链路，回到 `+trace` 看断在哪一跳。

36.2.3 drill 与 dnstracer：签名链和委派链

```
drill -S api.example.net @192.0.2.53
drill -T api.example.net @192.0.2.53
dnstracer -q a -r 3 api.example.net
```

`drill` 出自 NLnet Labs 的 ldns 工具集，`-S` 一类选项做 DNSSEC 签名链追踪，`-T` 做逐级 trace；`dnstracer` 专门做**逐级委派追踪**，输出从根到权威每一级由哪些 NS 应答、有没有缺胶水记录（glue record）、有没有环路。注意名字是 **`dnstracer`**，不是 `dnstrace`；命令拼错会让人误以为工具没装。两者在实测环境里属于「需要另装」的一类（按发行版包名安装 `ldnsutils` 一类工具包，具体包名见官方手册或发行版包索引），不是基础镜像自带。

`drill -S` 与 `delv` 的区别在于视角：`delv` 给「验签结论」，`drill` 更偏「把签名链逐段摊开给你看」；`dnstracer` 与 `dig +trace` 的区别在于 `dnstracer` 会把同一级多个 NS 都试一遍，容易发现「只有一个 NS 答错」这类单台权威问题。三者都不替代对方，按「先结论、再链、再逐台」的顺序用。

kdig 不在本书范围内：按用户决定忽略，只在此处与工具实装表标注为「忽略」，不写进书里的命令示例与排查流程。需要类似能力时用上面的 `dig` / `delv` / `drill` 组合。

怎么验证这一层：`dig @192.0.2.53 api.example.net A +short` 与 `delv @192.0.2.53 api.example.net A` 各跑一次，前者确认有应答，后者确认签名链结论。**失败模式 / 回滚点**：全是只读查询，无回滚；但 `@server` 指向错误的地址、或在生产解析器上误跑大量 `+trace`，会拿到「另一台机器」的答案或给权威添压。演练控制在一个自持域名上，别对第三方权威做批量追踪。

这一层的出口：解析器确实返回了预期数据→进第三层看线上报文与缓存；验签失败→第 10、16 章；委派链断在权威→属于对方或链路问题，转第 28 章的故障域隔离。

36.3 第三层：抓到什么包

第二层看到的是「我主动问一次，得到什么」；第三层要看的是**链路上真实发生了什么**，包括那些没人主动问就发出的重传、被悄悄丢掉的包、以及缓存里已经存了很久的旧答案。

36.3.1　抓包：先分清三种「不通」

```
tcpdump -ni any 'port 53 or port 853 or port 443'
tcpdump -ni any -vv 'host 192.0.2.53 and udp port 53'
tshark -ni any -f 'port 53' -Y 'dns.flags.rcode != 0'
```

`tcpdump -ni any` 的 `-n` 关掉反向解析（否则抓包工具自己会发 DNS 查询，把现场搅乱），`any` 覆盖所有接口，过滤表达式把 53（明文 DNS）、853（DoT）、443（DoH / HTTPS）一次收进来。`tshark` 是 Wireshark 的命令行版，若装了可以用显示过滤器直接挑出非零 RCODE；没装就用 `tcpdump -w /tmp/dns-cap.pcap` 存盘后离线分析，抓包文件含完整查询内容，按第 7、31 章的日志纪律设保留期并及时删除。

三种「不通」在包上是完全不同的形状，混淆它们会直接导致改错配置：

现象	抓包看到	典型位置
UDP 被丢	有出向查询、无回包，随后间隔重传同一事务 ID；无 ICMP 差错、无 RST	上游限速、策略路由送进黑洞、QoS 丢 UDP、防火墙静默 drop
被 RST / 拒绝	TCP SYN 后收到 RST，或 UDP 后收到 ICMP port unreachable	端口没监听、被显式 reject、服务进程已退出
超时	出向、回向都看得到，但应答 RCODE 为 SERVFAIL，或递归侧一直在等上游	上游超时、验证链等待、缓存里的旧 SERVFAIL、EDNS 缓冲区分片被丢

第一行最容易误判成「上游挂了」，实际可能是本机 nftables 把出向 53 劫持进了 tproxy（第 14、22 章那两个案例），下一层专门处理。第三行若伴随大应答，要想到 `edns-buffer-size` 过大在 MTU 1280 / PPPoE 链路上分片被丢、随后等超时重试 TCP 的坑（第 14 章）；分片与 QTYPE 分布都能在 `stats_noreset` 里找到旁证。

36.3.2　dnstap 与运行时状态

dnstap 是解析器**自己的查询流副本**：它记录客户端地址、QNAME、QTYPE、应答码与上游交互，不需要从网卡上猜。采集点配置与保留期在第 7、31 章，这里只用它回答「这次故障期间，那个名字到底进来过没有、被判成什么」。全量日志要写明保留期并到期删除，否则「抗元数据聚合」在自己门口破功。

```
unbound-control stats_noreset
unbound-control lookup api.example.net
unbound-control dump_cache
unbound-control flush api.example.net
unbound-control local_zone example.net. always_nodata
```

- `stats_noreset` 只读不重置计数，看命中、SERVFAIL、`num.queries_timed_out`、QTYPE 分布；排障期间千万别用会重置计数的 `stats`，否则差分基线全丢（第 31 章的注入演练专门演示过这点）。
- `lookup` 打印该名字当前记录的权威服务器（nameserver）信息，`cache_lookup` 与 `dump_cache` 才是看缓存内容——「权威改了、解析仍旧返回旧值」时先用 `dump_cache` / `cache_lookup` 确认是不是旧应答还在，而不是继续改配置。
- `flush` **会清缓存**，属于有副作用的操作。它清的不只是正应答：NXDOMAIN 一类否定应答也会按 [RFC 2308] 的负缓存（negative caching）规则留存，SOA minimum 常见是几千秒量级，上游一次瞬时故障能被放大成「修好权威后客户端仍失败」的长时间故障。更麻烦的是链路上每个缓存实例都得刷——只刷一台，前一层照样返回缓存里的 NXDOMAIN。执行前记下当前 `dump_cache` 的相关条目，执行后重测；具体子命令参数见官方手册。
- `local_zone` 是运行时加本地区，用来验证「是不是本地区接管了名字」（第 24 章讲 `local-zone` 的匹配优先级）。运行时改动 reload 即失效，别当成已固化配置写进报告。

怎么验证这一层：`tcpdump -ni any 'port 53'` 开着跑一次 `dig @192.0.2.53 api.example.net A`，确认出向与回向成对出现；再 `unbound-control stats_noreset` 看计数器是否递增。**失败模式 / 回滚点**：`flush` 清掉热缓存后瞬时递归量上升，`unbound-control` 通常需要相应权限（实测环境里要 `sudo -n` 或指定控制密钥路径，见官方手册）；抓包文件与 `dump_cache` 导出文件含查询内容，用完删除。回滚点是提前保存的 `dump_cache` 快照与规则集备份，不是「再刷一次」。

这一层的出口：包根本没出门→第四层查内核；包出门没回来且无 RST →查上游与链路；包回来了但答案是旧的→处理缓存与负缓存（第 7、10 章）。

36.4 第四层：内核动过没

先看 nftables，再怀疑别的。这是第 14、22 章反复用真实故障换来的顺序：本机解析器自己的出向 53 被 tproxy 吃掉、WireGuard 回包被 output 链打标送进代理，现象都长得像「上游不通」。在内核这一层，DNS 只是普通流量，规则不认识 QNAME。

36.4.1 只读查看与临时计数探针

```
nft list ruleset
nft -a list chain inet mangle output
conntrack -L | grep -E 'dport=53|sport=53'
ss -lntup | grep -E ':53|:853|:5353'
ip rule
ip -6 rule
```

`nft list ruleset` 一次看全表、链、集合与规则；`nft -a` 带出 `handle` 编号，是后面精确删除探针的依据。`conntrack -L` 看 UDP/53 条目是否堆积（重传与超时的旁证）；`ss -lntup` 确认本机到底有没有进程在 53 / 853 / 转发端口上监听；`ip rule` 与 `ip -6 rule` 看策略路由，解释「为什么这个源地址的包走了另一张表」。

要看「包究竟走到哪一行」，加一条临时 `counter` 探针：

```
nft -a list chain inet mangle output
nft add rule inet mangle output udp dport 53 counter
nft list chain inet mangle output
nft delete rule inet mangle output handle 12
```

为什么这么设：只读的 `list ruleset` 只能告诉你「规则存在」，不能告诉你「这条规则有没有被命中」；一条带 `counter` 的探针能在不改动现有规则、不 `nft -f` 重建表的前提下给出命中数。

怎么验证：`nft add rule` 后发起一次 `dig @192.0.2.53 example.org A`，再 `nft list chain inet mangle output`，探针的 `packets` 应递增；不递增说明流量根本没经过这条链。

失败模式 / 回滚点：`nft -f` 是**重建表**，会清空运行时加入的集合与计数（第 14 章专节讲过），排查期间不要顺手加载规则文件；`nft reset counters` 会清零所有计数、破坏差分基线（第 31 章）。删规则**必须带链名**（`nft delete rule <table> <chain> handle <N>`），只给 handle 在多数写法里不成立，删错链会留下半截状态。回滚按 handle 精确删除探针，或整体恢复改动前的 `nft list ruleset > /etc/nftables.conf.bak` 备份。

老系统还有 iptables-legacy 的 NAT 表在跑，单独看一眼即可，别在两套框架里同时改：

```
iptables -t nat -L -n -v
ip6tables -t nat -L -n -v
```

完整的劫持、豁免、fail-closed 配置块在第 14、22 章。这里要能判断「规则是否还在、是否被后续加载冲掉、出向豁免是否还在链首」。

怎么验证这一层：`nft list ruleset | grep -i -E 'counter|mark|dport 53|skuid'` 与 `ss -lntup` 对照，确认监听端口与包路径一致。**失败模式 / 回滚点**：把「规则文件里写了」当成「运行时生效了」是这层最典型的误判——`nft -f` 重建后动态集合被清空、链首的显式 accept 被冲掉，现象会伪装成上游或 DDNS 脚本坏了。回滚用整体备份，不逐条删。

这一层的出口：规则与包路径都正常→第五层做端到端；规则有问题→第 14、22 章；监听端口不对→回到第 4–7 章的实例配置。

36.5 第五层：应用怎么连

前四层都在回答「名字和包」的问题；第五层才回答用户真正抱怨的那句「网页打不开」。这一层的核心手法是**逐项把变量摘出方程**：`--resolve` 把 DNS 摘掉，`-4` / `-6` 把地址族摘掉，`--noproxy` 把代理摘掉，`socks5h://` 则反过来把 DNS 明确交给代理。

36.5.1 curl 的四种隔离

```
curl -v https://api.example.net/healthz
curl -v --noproxy '*' https://api.example.net/healthz
curl -v -4 --noproxy '*' https://api.example.net/healthz
```

```
curl -v -6 --noproxy '*' https://api.example.net/healthz
curl -v --resolve api.example.net:443:192.0.2.10 https://api.example.net/healthz
```

`curl -v` 的 `Trying 192.0.2.10:443...` 一行就是章首案例最需要的证据：它把「实际尝试了哪个地址」写出来了。`curl` **默认读取** `http_proxy` / `https_proxy` / `all_proxy` 环境变量，透明代理环境里残留一个指向死端口的变量，会让请求根本不经过本机 DNS，日志里却显示「连接失败」；所以第一组命令一定要带 `--noproxy '*'`，或先 `env | grep -i proxy` 看一眼。`-4` / `-6` 强制地址族，用来复现「v6 被 RST、v4 正常」；`--resolve` 直接指定 IP，把 DNS 从方程里彻底摘掉——若 `--resolve` 能通而原名不通，故障一定在解析链而非 TCP 链。

代理归属是另一条独立的分界。`ssh -D` 起的 SOCKS 监听**不自动接管系统 DNS**，应用要自己把名字交给代理，否则查询仍走本机解析器：

```
curl -v --noproxy '*' --socks5-hostname 127.0.0.1:1080 https://api.example.net/healthz
curl -v --noproxy '*' --proxy socks5h://127.0.0.1:1080 https://api.example.net/healthz
curl -v --noproxy '*' --proxy socks5://127.0.0.1:1080 https://api.example.net/healthz
```

`socks5h://` / `--socks5-hostname` 是**远程解析**（把主机名交给代理端），`socks5://` / `--socks5` 是**本地解析**（本机解析完把 IP 交给通道）。这类「DNS 归属开关」的语义（`resolvdns yes/no` 一类：yes 交对端、no 本机解析）在第 32 章已展开；要特别说明：**OpenSSH 的 `ssh_config` 里没有同名选项**，同类能力散落在不同客户端（PuTTY 的 “Do DNS name lookup at proxy end”、Firefox 的 `network.proxy.socks_remote_dns`、`sshuttle --dns` 等），具体归属以所用客户端文档为准。本章只用它做判读：本机无查询、对端有查询，才能证明名字确实交给了通道。

36.5.2　DoT 与 ICMP

```
openssl s_client -connect 192.0.2.53:853 -servername resolver.example.org
ping -c 3 192.0.2.53
ping6 -c 3 2001:db8::53
```

`openssl s_client` 用来测 DoT 端点能否完成 TLS 握手并出示预期证书，端口是 853，与 DoH 的 443 不是一回事（完整服务端配置见第 11、12 章）。`ping` / `ping6` 只测 ICMP 可达性：**`ping` 通不能判断 DNS 通**——ICMP 可能被单独放行，而 53 被 drop；反过来 DNS 通也不代表 TCP 通，章首 v6 被 RST 就是反例。把 ping 当作「链路层还活着」的一个粗信号即可，绝不写进「DNS 正常」的结论里。

怎么验证这一层：`curl -v --noproxy '*' -4 https://api.example.net/healthz` 与 `-6` 版本对照，再与 `curl --resolve` 版本对照；三者的差异直接定位到地址族或解析链。**失败模式 / 回滚点**：忘了 `--noproxy '*'` 是最常见的自欺；`--resolve` 里写错端口（应与 URL 的 443 一致）会连到不存在的服务；`socks5h://` 与 `socks5://` 只差一个字母却属于完全不同的 DNS 归属，写反了会得到「DNS 泄漏」的相反结论。全是客户端一次性参数，无回滚；但测试脚本里若把这些参数固化，记得发版前还原。

这一层的出口：任一地址能连、上层仍失败→应用层（证书、HTTP、代理 ACL）；全部地址连不上但 DNS 正确→回到第四层查包路径；只有某一地址族失败→第 19 章的源地址选择与前缀轮换。

36.6　第六层：归属与边界

最后一层不新增工具，只做一件容易被跳过的事：**把证据按层归位，指向该修的那一章**。同一条 `SERVFAIL`，可能是本地区接管（第 24 章）、验证链断裂（第 10 章）、上游超时（第 5、14 章）、负缓存（第 7 章）、也可能是本机 nftables 劫持（第 14、22 章）。不归位就会在错误的层反复改配置，越改越乱。

运行时状态与日志是归位的依据：

```
unbound-control stats_noreset
unbound-control get_option verbosity
unbound-control verbosity 3
journalctl -u unbound --since '10 minutes ago' --no-pager
unbound-control verbosity 1
```

`verbosity` 平时保持 1，排障窗口临时调到 2–3，排完立刻调回；它是 `unbound-control` 的运行时临时值，reload 即失效，别当固化配置写进演练报告（第 31 章已说明）。`log-servfail` 一行常开比把 verbosity 拉到很高更有性价比；`journalctl -u unbound` 配合 `--since` 缩小窗口，看 SERVFAIL 原因、验证失败、trust anchor priming 失败等行。日志关联的前提是所有链路共用同一个可信时钟：dnstap 帧、journald、nft 计数、抓包时间戳差几秒，因果链就会拼错（第 31 章）。

六层各自的「下一步去哪一章」：

层	确认的事实	若出问题，去哪一章
①谁在解析	应用走的是哪个存根 / NSS 路径	第 3 章（四种角色）、第 19 章（地址选择）、第 27 章（mDNS）、第 12 / 13 / 24 / 29 / 32 章（客户端绕过与接管）
②解析器怎么答	数据与签名链是否正常	第 10、16 章（DNSSEC）、第 5 章（转发器）、第 6 章（递归与根镜像）
③抓到什么包	报文有没有出门、缓存是不是旧的	第 7 章（缓存与可观测性）、第 14 章（EDNS 缓冲区 / 分片）、第 31 章（三层观测）
④内核动过没	规则、集合、监听端口与策略路由	第 14、22 章（nftables 经验与故障案例）
⑤应用怎么连	实际尝试的地址族、代理归属、TLS 握手	第 11、12 章（DoT / DoH）、第 19 章（源地址选择）、第 32 章（socks5h DNS 归属）
⑥归属与边界	证据落在哪一层、该谁修	第 22 章（故障案例集）、第 28 章（故障域隔离）、第 31 章（告警与误报治理）

与第 14/22 章的 nftables 经验、第 19 章的源地址选择、第 31 章的告警与观测、第 32 章的 socks5h DNS 归属，本章只作交叉引用，不重复那些章的完整配置块。

36.7 哪条命令会骗你

把散落在前面几层的陷阱集中列在这里。它们的共同点是：**命令本身没报错，输出也「看起来对」，但回答的是另一个问题**。

1. **`ping` 通≠ DNS 通**。ICMP 与 53 端口是两套放行规则；ICMP 通只能说明三层可达。`ping` 不通也不必然是故障——很多主机主动不回 ICMP。
2. **DNS 通≠ TCP 通**。章首案例最典型：`getaddrinfo` 全量成功，前四个 v6 地址被 RST，后面五个 v4 全通。「解析成功」只覆盖名字到地址这一步，连接能否建立还要看地址族、路由、对端监听与防火墙。
3. **`curl` 默认吃代理环境变量**。`http_proxy` / `https_proxy` / `all_proxy` 残留时，请求可能根本不走本机 DNS、也不走目标地址；结论会指向一个根本没参与的设备。第一组命令永远带 `--noproxy '*'`。
4. **`nslookup` 与 `getent` 路径不同**。`nslookup` 用自己的解析库，不总跟随 `resolv.conf` 的全部路径；`getent` 走 NSS，可能直接读 `/etc/hosts` 或缓存模块而完全不碰 DNS。两者结果不一致时，先怀疑路径，别怀疑数据。
5. **`dig @server` 看到的是那台服务器，不是应用实际拿到的答案**。指定 `@192.0.2.53` 就绕开了本机存根、绕开了 systemd-resolved 的分接口策略、绕开了 `local-zone` 与过滤表。它能证明「这台服务器这样答」，不能证明「应用收到这个」。要复现应用路径，用不带 `@` 的 `dig` 加上 `getent` / `curl -v`。
6. **「网络已恢复」不基于重探测就不是证据**。上层页面打开、告警被手工清掉、下个周期探测恰好命中了 v4，都可能在故障未修复时产生「恢复」信号。恢复日志必须记录同一套探测的重跑结果与实际连通的地址（第 31 章的误报治理）。
7. **`unbound-control flush` 与负缓存 [RFC 2308]**。权威修好后仍失败，第一反应应该是「缓存里是不是还留着旧的 NXDOMAIN / SERVFAIL」，而不是继续改配置。否定应答按 SOA minimum 缓存，会放大上游的瞬时故障；且链路上每个缓存实例都要刷。`flush` 有副作用，执行前先 `dump_cache` 留底。
8. **`+trace`、`drill -S`、`dnstracer` 看到的是「从本机重新迭代一遍」的结果**，不是递归解析器当时走的缓存路径。用它定位委派链断点可以，用它断言「Unbound 当时就是这么查的」不行。
9. **`nft list ruleset` 里写着，不等于运行时命中了**。规则被后续 `nft -f` 冲掉、动态集合被清空、出向豁免被挪到链尾，文件里都还是「有」的。要看命中得靠 `counter`，要看运行时值得靠 `nft -a list chain ...` 与 `ss -lntup`。

一句话收束：**先确认这条命令回答的是哪一层的问题，再拿它的输出当证据**。

36.8 工具实装表

下表是本章实验环境的实测结果，不是各发行版的通用承诺。包名、版本与选项一律以所用发行版与官方手册为准，拿不准就写「见官方手册」，不要凭记忆写版本号。

工具	本机	另一台网关	备注
`dig` / `host` / `nslookup` / `getent` / `tcpdump`	有	有	基础工具，通常随系统或 `dnsutils` 一类包提供
`delv`	有	有	BIND 系，自带信任锚做 DNSSEC 验签，能给出 bogus 原因
`drill`	有	无	NLnet Labs / ldns，按发行版包名安装（`ldnsutils` 一类）
`dnstracer`	有	无	逐级委派追踪；注意名字是 **`dnstracer`**，不是 dnstrace
`nft` / `unbound-control`	有	需 `sudo -n` / 指定控制密钥路径	权限边界写清：只读查看与有副作用的 `flush` / `verbosity` 分开
`resolvectl`	无	无	systemd-resolved 配套命令；未装时读 `/etc/resolv.conf` 与按接口配置，替代路径见发行版官方手册
`tshark`	视发行版	视发行版	「按发行版包名安装」；没装就用 `tcpdump -w` 存盘离线看
`kdig`	—	—	**按用户决定忽略，不写进书**，也不列入命令示例

「有」只表示命令能跑起来，不代表该机器上启用了对应服务；`resolvectl` 未装也不代表 systemd-resolved 不存在，这两件事要分开。**工具缺失时不要改排查顺序**：第二层缺 `delv` 就用 `dig +dnssec` / `+cd` 组合加服务端日志，第三层缺 `tshark` 就用 `tcpdump` 存盘，第六层缺 `resolvectl` 就读配置文件。顺序是按层固定的，工具只是每层的可替换实现。

36.9　症状→第一条命令→判读→下一层→本书章节

最后给一张收尾决策表。用法是**从上往下按行对症状，每条只跑第一行命令**；判读成立就跳到「下一层」，不成立就继续看下一行，不要一上来把六层命令全跑一遍。

症状	第一条命令	判读	下一层	本书章节
某个应用报「DNS 挂了」，别的应用正常	`getent ahosts api.example.net` 与 `cat /etc/nsswitch.conf`	NSS 路径或 `/etc/hosts` 命中不同，应用可能根本没走系统解析器	①→⑤	第 3、24、29、32 章
双栈域名时通时断、日志出现「报故障→4 秒后恢复」	`curl -v --noproxy '*' -4 https://api.example.net/healthz` 与 `-6` 对照	v6 被 RST / v4 通，是地址族与探测只试前几项的问题	①→⑤	第 19、22、31 章
`dig` 有答案但应用仍失败	`dig @192.0.2.53 api.example.net A` 与不带 `@` 的同名查询对照	指定服务器绕开了存根与 `local-zone`，两者答案不一致说明路径不同	①→②	第 3、24 章
全站 SERVFAIL，`+cd` 能拿到数据	`delv @192.0.2.53 api.example.net A`	验证链问题（信任锚、时钟、签名被剥离），不是数据缺失	②→⑥	第 10、16 章
某个外部域解析不了，本级 NS 有一台答错	`dnstracer -q a -r 3 api.example.net`	委派链或单台权威异常，本机递归器未必有错	②→③	第 6、28 章

症状	第一条命令	判读	下一层	本书章节
出向查询无回包、无 RST、只有重传	`tcpdump -ni any 'port 53'`	UDP 被静默丢，先排除本机规则劫持再怀疑上游 QoS	③→④	第 14、22、31 章
权威已修好但客户端仍返回 NXDOMAIN	`unbound-control dump_cache | grep api.example.net`	负缓存 [RFC 2308] 未过期，链路上每个缓存实例都要刷	③→⑥	第 7、10 章
解析器「根区镜像不更新 / AXFR 失败」	`nft -a list chain inet mangle output`	本机解析器出向 53 被 tproxy 吃掉，需 uid 豁免在链首显式 accept	④→③	第 14、22 章
规则文件里写着豁免，现象却像没生效	`nft add rule inet mangle output udp dport 53 counter`（用完按 handle 删）	计数器不递增 = 包没走到这条链；`nft -f` 重建可能已清空集合	④→①	第 14、31 章
`curl` 连接失败但 `dig` 正常，环境里配过代理	`env | grep -i proxy` 然后 `curl -v --noproxy '*' https://api.example.net/healthz`	残留代理变量把请求送进死端口，DNS 根本没参与	⑤→④	第 23、32 章
用了 SOCKS 通道但本机仍看到 DNS 查询	`curl -v --proxy socks5h://127.0.0.1:1080 https://api.example.net/healthz`	`socks5://` 本地解析、`socks5h://` 远程解析，归属开关没设对	⑤→①	第 32 章
`ping` 通、`dig` 通、DoT 端口连不上	`openssl s_client -connect 192.0.2.53:853 -servername resolver.example.org` 与 `ss -lntup`	ICMP 与 DNS 与 TLS 是三条独立链路，逐项隔离	⑤→④	第 11、12 章
同一故障反复改配置都不见效	`journalctl -u unbound --since '10 minutes ago' --no-pager` + `unbound-control stats_noreset`	证据没按层归位，先用日志与计数确定断点在哪一层	⑥→①	第 22、28、31 章

表里「下一层」写成 ① → ⑤这种形式，是为了强调排查不是单向流水线：确认第五层的应用行为后，常常要回到第一层重看解析路径。真正省时间的不是记住多少命令，而是每次只回答一个问题，并且知道**这条命令会骗你到哪一层为止**。

36.10 本章要点

1. 排查按六层顺序走，不按工具名字母排：谁在解析→解析器怎么答→抓到什么包→内核动过没→应用怎么连→归属与边界；每层只回答一个问题，`dig @server`、`getent`、`ping` 各自只覆盖其中一小段。
2. 章首案例的三条结论贯穿全章：探测要全量去重遍历（或 v4 优先），「网络已恢复」必须基于重探测结果，`getaddrinfo` 给的是 [RFC 6724] 策略排序而非可用性排序。
3. 命令会骗人是因为它回答了另一层的问题：`ping` 只测 ICMP、`curl` 默认吃代理变量、`nslookup` 与 `getent` 路径不同、`dig @server` 绕开本机存根、`+trace` 不等于递归器当时的缓存路径。先用 `--resolve` / `-4` / `-6` / `--noproxy '*'` 把变量逐项摘出方程。
4. 三种「不通」在抓包里形状不同：UDP 被静默丢（有出向无回包、只有重传）、被 RST / 拒绝、以及超时回 SERVFAIL；对应上游丢包、显式拒绝、验证或分片问题，改错了层会越改越乱。

5. **先看 nftables**：`nft -a` 带 handle 的临时 `counter` 探针用完按链名精确删除，`nft -f` 重建表会清空集合、`reset counters` 会破坏差分基线；`unbound-control flush` 与 [RFC 2308] 负缓存是「权威修好仍失败」的第一嫌疑，且链路上每个实例都要刷。
6. 工具实装表按实测写清「有 / 需装 / 未装 / 忽略」（`delv` 有，`drill` 与 `dnstracer` 需装，`resolvectl` 未装，`kdig` 按用户决定忽略）；工具缺失不改排查顺序，收尾决策表给出「症状→第一条命令→判读→下一层→本书章节」。

36.11　参考资料

- Unbound 官方手册,`unbound-control(8)` 的 `stats_noreset`、`lookup`、`dump_cache`、`flush`、`local_zone`、`verbosity`: https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound-control.html
- Unbound 官方手册, `unbound.conf(5)`（`verbosity`、`log-servfail`、`dnstap`、`edns-buffer-size` 各节）: https://unbound.docs.nlnetlabs.nl/en/latest/manpages/unbound.conf.html
- NLnet Labs，Unbound 项目文档入口（`drill` / ldns 相关工具与包名以此为准）：https://nlnetlabs.nl/documentation/unbound/
- RFC 6724, *Default Address Selection for Internet Protocol Version 6 (IPv6)*（`getaddrinfo` 返回顺序与源地址选择）：https://www.rfc-editor.org/rfc/rfc6724
- RFC 2308, *Negative Caching of DNS Queries (DNS NCACHE)*（负缓存与 SOA minimum）：https://www.rfc-editor.org/rfc/rfc2308
- RFC 7766, *DNS Transport over TCP - Implementation Requirements*（TCP/53 与 `+tcp` 判读）: https://www.rfc-editor.org/rfc/rfc7766
- RFC 5737, *IPv4 Address Blocks Reserved for Documentation*（本章全部示例地址）：https://www.rfc-editor.org/rfc/rfc5737
- nftables 官方 wiki, 规则集管理、`nft -f` 加载语义与 handle 用法: https://wiki.nftables.org/wiki-nftables/index.php/Main_Page

第三十七章

HTTPDNS：为什么引入、如何用 Unbound 自建、大厂后端如何架构

本章讨论的是一种**不属于 DNS 协议体系**的解析服务形态。它不是加密传输的又一种写法，也不是某个 RFC 的实现，而是应用用 HTTP(S) 去调一个私有 API、拿回一组 IP、再自己改写网络请求的过程。第 12 章的 DoH [RFC 8484] 与它名字相近，但载荷、客户端形态与信任模型都不同；本书把那层区分收在此章统一讲，其他章不再复述。

37.1 先把它和标准 DNS 分开

一次普通解析是「存根解析器→递归解析器→权威服务器」，载荷始终是 DNS wire format，QNAME、QTYPE、RCODE、TTL、RRSIG 都有固定位置。第 3 章的四种角色、第 10 章的 DNSSEC 验证、第 7 章的缓存与负缓存，全都建立在这份报文结构上。

HTTPDNS 的一次调用则是：

```
GET /d?host=api.example.com&uid=app-001&ts=<TS>&nonce=<NONCE>&sig=<SIG> HTTP/1.1
Host: 192.0.2.53            ← 入口是 IP 字面量，不是待解析的域名

→ {"host":"api.example.com","ips":["192.0.2.10","192.0.2.11"],
   "ttl":60,"client_ip":"198.51.100.23"}
```

请求里没有 DNS 报文，应答里没有 answer section，也没有可以挂 RRSIG 的位置。它是**应用层 API**，只是恰好在做「名字→地址」这件事。因此它没有 RFC，也不该被叫作「某种 DNS 协议」：HTTP 本身有 RFC，但 URL 格式、参数名、返回字段、鉴权、错误码、SDK 行为全部由各家产品自定义。厂商文档自己也是这么区分的——把私有协议写成「支持身份鉴权、业务调度、批量解析加速、便于扩展」，把 [RFC 8484] DoH 写成「标准协议，格式固定，部分网络库原生支持，但无鉴权机制、不可扩展、不支持传递业务参数，因此不支持自定义解析」。

这条界线决定了后面所有取舍：**DoH 能挂进第 12 章那套 dnsdist + nginx + Unbound 链路，是因为它仍是一次规规矩矩的 DNS 查询；HTTPDNS 挂不进去，因为 Unbound 收不到 DNS 报文**。

维度	传统 DNS / DoH	HTTPDNS（私有 API）
载荷	DNS wire format；DoH 外加 HTTPS 封装	JSON 或分号分隔的纯文本 IP 列表
客户端	系统存根、浏览器、支持 DoH 的网络库	App 内嵌 SDK，或改造过的网络库
端点来源	本地配置 / URI Template，管理员可控	SDK 内置，通常硬编码 IP 或自举域名
业务参数	无法携带	可带 `uid`、App 版本、运营商、自定义标识
鉴权	标准协议无	Account ID + 密钥 / HMAC / 参数加密

维度	传统 DNS / DoH	HTTPDNS（私有 API）
自定义解析	无	按客户端出口 IP、UIN、版本返回不同结果
缓存与 TTL	递归解析器与 HTTP 缓存	SDK 自己实现，可预解析、持久化、测速排序
DNSSEC / dnstap / 过滤表	全部仍在链路上	全部被跳过

表里最后一行是运维最该记住的：一旦 App 走了私有 API，管理员在网关上改 DHCP option、改 RDNSS、压 DDR 都影响不到它。第 13 章讨论的「客户端绕过」在这里不是绕过某条规则，而是绕过了整套 DNS 子系统。本章不评价这种取舍的好坏，只把它作为引入动机与自建约束的前提写清楚。

37.2　为什么要引入 HTTPDNS

HTTPDNS 在国内移动端兴起，不是因为「HTTP 比 UDP 快」——恰恰相反，多一次 HTTPS 往返通常更慢。它要解决的是**运营商 LocalDNS 不可控**带来的一组具体问题。厂商产品文档归纳的四条收益是「防劫持、精准调度、0ms 解析延迟、快速生效」，中文实践文章则把 LocalDNS 的痛点列得更细：劫持、TTL 不生效、解析延迟、调度不准、缺少异地容灾。把这些合并成工程语言，是下面五条。

37.2.1　痛点一：劫持与污染

LocalDNS 由运营商下发，链路明文，中间设备可以抢答、改写或注入。对移动 App 来说，轻则被插广告，重则核心 API 返回不可用地址。私有 API 走 HTTPS 到自己的服务端，链路上的中间设备看不到也改不了；入口用 IP 字面量，连「解析解析服务域名」这一步都不交给 LocalDNS。这里的边界要说准：**HTTPS 保护的是 App 到解析服务端这一段，不保护 App 拿 IP 之后的业务连接**，后者仍靠业务自己的 TLS。

37.2.2　痛点二：TTL 不生效、变更慢

权威侧把一条 A 记录从 `192.0.2.10` 改成 `192.0.2.20`，客户端多久能看到？取决于链路上每一层缓存。部分 LocalDNS 不严格遵循权威 TTL，会把旧结果留很久；中文实践材料记录过 TTL 被拉到数十小时量级的情况。故障时这最要命：切流已经做完，一批用户还连着已经下线的 IP。

HTTPDNS 服务端可以给客户端返回一个**独立于权威 TTL 的 `ttl`**，并在控制台改解析配置后主动让边缘缓存失效。第 7 章讲过负缓存会把瞬时故障放大成小时级故障，HTTPDNS 绕开 LocalDNS 后，这类放大不再由不受控的第三方决定——代价是缓存策略变成自己的责任。

37.2.3　痛点三：调度不准

CDN 与 GTM 靠 DNS 把用户导到就近边缘节点，判断依据是**递归解析器的地址**，不是真实客户端地址。当客户端配置了别处的公共递归解析器、或运营商把查询转发到异地节点时，权威侧看到的出口 IP 与用户实际位置可能差得很远，结果是跨网、跨地域访问。

HTTPDNS 服务端在 TCP 连接里直接看到客户端出口 IP，应答里甚至会把 `client_ip` 回显出来（本章开头的示例响应即是这个形状）。这让「按客户端地区与运营商选节点」重新成为可能。厂商文档把这一点写成「直接得到客户端的出口网关 IP，从而更准确地判断客户端地区和运营商」；注意它和 [RFC 7871] 的 EDNS Client Subnet 不是一回事：ECS 是把网段信息塞进 DNS 报文给权威侧，HTTPDNS 是根本不发 DNS 报文，在应用层自己拿。

37.2.4　痛点四：容灾与切换粒度

LocalDNS 对业务方是黑盒：它挂了、被污染了、转发到异地了，App 只能等超时。私有 API 的客户端握着一组服务 IP，可以做轮转重试、按网络切换刷新、失败降级回 `LocalDNS`。厂商的最佳实践文档把这套写成明确要求：内置多个启动 IP、冷启动 / 网络切换 / 每 8 小时 / 全部失败时更新服务 IP 列表、单 IP 失败轮转到下一个、全部失败才降级。这种粒度是 DHCP 下发的单一 DNS 地址给不了的。

37.2.5 痛点五：业务身份参与调度

标准 DNS 报文里放不进「这个用户属于哪个灰度桶」「这个客户端版本要走哪组接入点」。私有 API 可以带 uid、版本、地域等参数，服务端据此返回不同 IP。这是「带业务身份的流量调度」，不是传统意义上的域名解析；也正因为需求超出了 DNS 报文的表达能力，厂商才没有把它做成 RFC，而是各发一套互不兼容的 SDK。

37.2.6 什么时候不该引入

五条痛点不等于所有场景都该上 HTTPDNS。厂商产品文档明确写着：它**适用于 C/S 架构**，因为拿到 IP 后必须能定制网络请求；浏览器 / B/S 架构里网络实现对开发者是黑盒，不适合。再叠上本章后面要讲的客户端改造成本（SNI、证书校验、代理检测、WebView 适配），合理的落地范围通常是「核心 API、支付、登录、直播推拉流」这类少数域名，其余仍走系统 DNS。

这里有一条和第 12 章呼应的判断：**如果你的目标只是「链路加密、别让链路上的设备看到查询」，用 DoH 就够了，而且成本低得多**——系统或浏览器原生支持，本机过滤表、DNSSEC 验证、dnstap 观测全都还在。只有需要「真实出口 IP 调度、按业务身份自定义解析、变更秒级生效、SDK 侧缓存策略」时，才轮到私有 API。把两者混为一谈，最常见的后果是照着 DoH 教程配了一遍 dnsdist，最后发现拿不到 client_ip、也没法按 uid 分流。

37.3 用 Unbound 自建：三层缺一不可

37.3.1 认知前提：Unbound 不是 HTTPDNS 服务器

按第 1 章定下的边界，Unbound 只做递归、缓存、DNSSEC 验证与非常有限的本地权威服务（local-zone / local-data）。它不提供 HTTP JSON API，也不会替客户端改写网络请求。**只把 Unbound 开成 DoH 端口，得到的是标准 DoH，不是 HTTPDNS**。

完整自建是三层：

```
[App / SDK]  ← 第三层：客户端改造，不能省
   | ① HTTPS GET /d?host=...&sig=...（入口为 IP 字面量）
   ▼
[HTTP(S) 前端]  ← 新增的唯一组件：鉴权、取 client_ip、查规则、发 DNS 查询
   | ② 标准 DNS 报文（UDP/53 或 TCP/53，回环）
   ▼
[Unbound]  ← 已有能力：递归 / DNSSEC / 缓存 / local-zone 自定义视图 / dnstap
   ▼
[JSON 响应] → App 拿 IP 直连，保留 Host 头与 SNI
```

前端与 Unbound 之间**仍然是标准 DNS 报文**。这是本章最关键的架构判断：私有 API 的「私有」只体现在 App ↔前端那一段；后端完全可以、也应该复用一台正常的递归解析器，别去重造解析逻辑。

37.3.2 拓扑与端口规划

教学环境用一台网关 gw-01，上面跑三个角色，端口互不冲突：

```
# /etc/unbound/unbound-httpdns.conf ——专供 HTTPDNS 前端的实例（示意）
server:
    interface: 127.0.0.1@5358
    port: 5358
    access-control: 127.0.0.0/8 allow
    access-control: 0.0.0.0/0 refuse
    do-ip4: yes
    do-ip6: no
    # 与第 4 章一致的 EDNS 与缓存基线
    edns-buffer-size: 1232
    max-udp-size: 1232
    msg-cache-size: 32m
    rrset-cache-size: 64m
    prefetch: yes
    serve-expired: yes
    # 本实例仍做真验证，别因为「只是给前端查」就关掉
    auto-trust-anchor-file: "/var/lib/unbound/root.key"
```

```
    val-permissive-mode: no
    # 私有地址应答要放行，否则内网视图会被 private-address 剔除
    private-address: 192.168.0.0/16
    private-address: 10.0.0.0/8
    private-domain: "internal.example.net."
    # 自定义视图：对应厂商收费的「自定义解析 / SDNS」
    local-zone: "internal.example.net." static
    local-data: "api.internal.example.net. 300 IN A 192.0.2.10"
    local-data: "api.internal.example.net. 300 IN A 192.0.2.11"

# 观测：前端额外记 client_ip / view，Unbound 侧 dnstap 照常保留
dnstap:
    dnstap-enable: yes
    dnstap-socket-path: "/run/unbound-httpdns/dnstap.sock"
    dnstap-log-client-query-messages: yes
    dnstap-log-client-response-messages: yes
```

①为什么这么设：单独起一个实例而不是复用网关对 LAN 的 :53，有三个理由。一是对外服务的那个实例带过滤表、dnstap 与面向全网的 access-control，把私有 API 的查询混进去会让计费、限流和日志口径全乱；二是自定义解析可能与过滤表冲突（always_null 会先于递归命中，第 24 章讲过优先级）；三是 HTTPDNS 前端是本机回环调用，只需要 127.0.0.0/8 allow，暴露面可以压到最小。local-zone + local-data 承担「按视图返回不同 IP」，private-domain 给内网地址开白名单，正是第 9 章那条 private-address 坑的标准处理。

②怎么验证：

```
unbound-checkconf /etc/unbound/unbound-httpdns.conf
dig @127.0.0.1 -p 5358 api.internal.example.net. A +short
dig @127.0.0.1 -p 5358 example.org. A +dnssec | grep -E 'status:|flags:'
```

第一条应回两条文档地址；第二条应 NOERROR 且 flags 含 ad，证明这个实例不是「为了专用而关掉验证」的空壳。

③失败模式 / 回滚点：最常见的是 private-address 把自定义视图里的内网地址剔掉，表现为「前端问得到 SOA、拿不到 A」。回滚是补 private-domain；若 local-data 与过滤表同名区撞车，删掉冲突的 local-zone 行并 reload。整章改动只影响 5358，把前端指回原来的 :53 即可退回，不动客户端。

37.3.3　前端：HTTP(S) → DNS 的转换层

网关上若没有 nginx / caddy，用 Python 标准库就能跑最小骨架；生产环境建议把 TLS 交给成熟的反向代理。下面这段只做四件事：鉴权、取真实出口 IP、查自定义规则、把查询转成标准 DNS 报文发给上面的 Unbound 实例。

```
#!/usr/bin/env python3
"""httpdns-front.py ——教学骨架：HTTP(S) 私有 API → 标准 DNS 查询。
仅监听回环；生产须由 TLS 反向代理终止，并补齐限流、共享 nonce 存储与审计日志。"""
import hmac, hashlib, json, time, struct, socket, secrets
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer
from urllib.parse import urlparse, parse_qs, quote

UNBOUND = ("127.0.0.1", 5358)
KEY = b"<TOKEN>"                        # 每 App 一把，由控制台下发，不进客户端包
ALLOWED = {"api.example.com", "api.internal.example.net."}
WINDOW = 300
_seen = {}                              # 教学用；多节点须换带 TTL 的共享存储

def canonical(p):
    return "&".join(f"{k}={quote(str(p[k]), safe='')}"
                    for k in sorted(p) if k != "sig")

def sign(p):
    return hmac.new(KEY, canonical(p).encode(), hashlib.sha256).hexdigest()

def dns_query(name, qtype=1):
    """ 构造最小 DNS 查询并解析 answer 段里的 A/AAAA；教学骨架，不处理 CNAME 链。"""
    qid = secrets.randbelow(65536)
```

```
    labels = b"".join(bytes([len(x)]) + x.encode() for x in name.rstrip(".").split("."))
    pkt = struct.pack("!HHHHHH", qid, 0x0100, 1, 0, 0, 0) + labels + b"\x00" \
          + struct.pack("!HH", qtype, 1)
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.settimeout(2)
    s.sendto(pkt, UNBOUND)
    data, _ = s.recvfrom(4096)
    qd, an = struct.unpack("!HH", data[4:8])
    off = 12
    for _ in range(qd):                          # 跳过 question
        while data[off]: off += data[off] + 1
        off += 5
    ips, ttl = [], 60
    for _ in range(an):                          # 只取 IN/A 的简明形态
        if data[off] & 0xC0 == 0xC0: off += 2
        else:
            while data[off]: off += data[off] + 1
            off += 1
        rt, rc, ttl, ln = struct.unpack("!HHIH", data[off:off+10])
        off += 10
        rd = data[off:off+ln]; off += ln
        if rt == 1 and rc == 1 and ln == 4:
            ips.append(".".join(map(str, rd)))
    return ips, ttl

class H(BaseHTTPRequestHandler):
    def do_GET(self):
        u = urlparse(self.path)
        p = {k: v[0] for k, v in parse_qs(u.query, keep_blank_values=True).items()}
        got = p.pop("sig", "")
        # 四道检查：白名单 / 时间窗 / 常量时间比较 / nonce 去重
        if u.path != "/d" or p.get("host") not in ALLOWED:
            return self._send(403, {"error": "forbidden"})
        try: ts = int(p.get("ts", "0"))
        except ValueError: return self._send(401, {"error": "bad ts"})
        if abs(time.time() - ts) > WINDOW or not p.get("nonce"):
            return self._send(401, {"error": "stale/missing ts or nonce"})
        if not hmac.compare_digest(sign(p), got):
            return self._send(401, {"error": "bad signature"})
        nk = f"{p['nonce']}:{ts}"
        if nk in _seen: return self._send(401, {"error": "replay detected"})
        _seen[nk] = ts + WINDOW

        client_ip = self.client_address[0]       # HTTPDNS 的核心：真实出口 IP
        try:
            ips, ttl = dns_query(p["host"])
        except (socket.timeout, OSError):
            return self._send(503, {"fallback": "localdns"})   # 让 App 降级
        self._send(200, {"host": p["host"], "ips": ips, "ttl": ttl,
                         "client_ip": client_ip,
                         "view": "lan" if client_ip.startswith("192.168.") else "wan"})

    def _send(self, code, obj):
        b = json.dumps(obj).encode()
        self.send_response(code)
        self.send_header("Content-Type", "application/json")
        self.send_header("Cache-Control", "no-store")
        self.send_header("Content-Length", str(len(b)))
        self.end_headers(); self.wfile.write(b)
    def log_message(self, *a): pass

ThreadingHTTPServer(("127.0.0.1", 8080), H).serve_forever()
```

①为什么这么设：`ALLOWED` 白名单不能省，否则这台东西就变成对任意域名开放的解析放大器；HMAC 用 `compare_digest` 做常量时间比较，避免逐字节时序侧信道；`ts` + `nonce` 必须一起验，只有 HMAC 的话攻击者抓一个合法 URL 就能原样重放；`client_ip` 取自 socket peer 而不是请求参数，正是「精准调度」这条收益的来源；`dns_query()` 发的是标准 DNS 报文，所以 Unbound 侧的 DNSSEC 验证、缓存、dnstap 全都照常工作。第 5 章那条「单上游不是冗余」的教训在这里同样适用：前端只指向一个 Unbound 实例，切换交给外部巡检，别在代码里塞两个目标做竞速。

②怎么验证：

```
# 1) 后端先自证可用
dig @127.0.0.1 -p 5358 api.example.com. A +short
# 2) 前端正常请求（ts/nonce/sig 由同一个 canonical() 生成）
python3 - <<'PY'
import time, secrets, hmac, hashlib
from urllib.parse import quote
KEY=b'<TOKEN>'
p={'host':'api.example.com','uid':'app-001','ts':str(int(time.time())),
   'nonce':secrets.token_hex(8)}
c='&'.join(f'{k}={quote(str(p[k]),safe="")}' for k in sorted(p) if k!='sig')
print(c+'&sig='+hmac.new(KEY,c.encode(),hashlib.sha256).hexdigest())
PY
# 3) 反例：改一个 sig 字符应 401；同一串发两次第二次应 replay detected
curl -sS 'http://127.0.0.1:8080/d?< 上面的串 >'
# 4) 确认前端确实问了 Unbound（dnstap 里应出现对应 QNAME）
```

③失败模式 / 回滚点：`dns_query()` 是教学骨架，只解析直接的 A 记录、不追 CNAME、不处理压缩指针的完整形态、不做 TCP 回退，遇到真实 CDN 的 CNAME 链会返回空列表——生产请换成成熟的 DNS 库，并保留 CNAME 链与 AAAA。另一个高频坑是签名规范化：`quote(safe='')` 把空格编成 `%20`，`urlencode()` 默认把空格编成 `+`，两端用不同函数算出的签名永远对不上，必须把 canonical 串固定成同一份实现。回滚：前端进程停掉即可，Unbound 侧无需改动；App 侧按契约收到非 200 就降级回系统 DNS。

37.3.4　鉴权与放大面

HMAC 解决的是「请求来自持密钥的 App / 参数没被改 / 不是旧请求重放」，它是完整性加身份认证，**不是加密**——`host` 仍在 URL 里明文。四道检查缺一不可：

检查	作用	漏掉的后果
域名白名单	限定可解析的名字	变成开放解析器，被拿去扫全网域名
`abs(now - ts) <= 300s`	压缩重放窗口	抓到的 URL 永久有效
`compare_digest`	常量时间比较	时序差异泄露签名前缀，可逐字节爆破
`nonce` 去重	窗口内一次有效	合法 URL 被重放一万次，按次计费直接爆表

多节点部署时 `_seen` 必须换成带 TTL 的共享存储，否则请求轮询到另一台边缘节点就漏了重放检查。密钥粒度按 App 或按 SDK 发放，便于单独轮换、限流与计费；客户端终究可被逆向，所以白名单与频率限制是最后一道，不能指望密钥永远不泄露。要求更严时可以把 `host` 也加密（部分厂商用 AES/DES 加密域名参数），代价是排障时看不到明文 QNAME。

`Cache-Control: no-store` 要显式写：这类响应里带 `client_ip` 与视图信息，落到共享缓存里既泄漏又会让 TTL 语义出现两套时钟——第 12 章讨论 DoH 时遇到过同样的问题。

37.3.5　客户端那一半

拿到 IP 只是上半段。私有 API 的定义性特征在下半段：**URL 里是 IP，Host 头里是域名**。

```
① GET https://192.0.2.53/d?host=api.example.com&...&sig=...
   → {"ips":["192.0.2.10"],"ttl":60,"client_ip":"198.51.100.23"}
② POST https://192.0.2.10/v1/pay HTTP/1.1
   Host: api.example.com          ← 必须手工保留
   （TLS SNI 也设为 api.example.com，证书校验用原域名）
```

HTTP 要手动写 `Host`；HTTPS 要设 SNI 并用**原域名**校验证书，否则 `certificate doesn't match IP`；按 `ttl` 缓存；网络切换时刷新（CDN 线路可能已变）；代理开启时强制关闭私有解析；全部失败时降级 `LocalDNS`。厂商文档给出的降级写法很直白：Android 侧 `okhttp3.Dns.SYSTEM.lookup(host)`，iOS 侧只需不做 host 替换、保持原有请求。

厂商《客户端接入概述》把接入方式归纳成三类，正好对应本章三层的不同取舍：SDK、HTTP API、以及 DoH。SDK 屏蔽交互复杂度并附带完整的高可用与性能优化；HTTP API 留给无法集成 SDK 的场景，但上表那些缓存、预解析、降级全得自己写；DoH 适配成本最低、网络库行为与 LocalDNS 一致，却拿不到业务参数与自定义解析。「**封闭网络库优先用 DoH，自有 App 核心域名用 SDK**」**是厂商文档给出的选型结论，不是本书的偏好**。

37.3.6 生产化清单（把骨架变成能上线）

厂商的 HTTP API 最佳实践几乎就是一份自建需求规格书，逐条对应到本章三层：

要求	落点	本章对应
启动 IP 冗余；冷启动 / 网络切换 / 每 8h / 全失败时更新服务 IP	客户端	前文「容灾粒度」
单 IP 失败轮转下一个，全失败降级 LocalDNS	客户端	降级契约
按 TTL 缓存；TTL 80% 预解析；持久化缓存	客户端	客户端那一半
网络切换主动刷新；按 ICMP/TCP 测速排序	客户端	精准调度
异步化，避免业务主流程同步等待	客户端	骨架里的超时与 503
鉴权解析、HTTPS 证书校验、会话 ID 排查	前端 + 客户端	鉴权与放大面
连接复用	前端	反代 TLS 层

还有两条容易被忽略：测速排序会**改变业务服务器原本的负载均衡策略**；网络切换时刷新所有域名会带来额外流量。它们都是「客户端自己掌握调度权」之后的必然代价，不是实现缺陷。第 26 章讲 anycast 集群时讨论过缓存不共享，这里同理：SDK 的本地缓存与 Unbound 的递归缓存是两套独立的时钟，排查「为什么改了记录还不生效」时两边都要看。

37.4 大厂后端是怎么架构的

公开材料里能看清的层次，可以归纳成「管控层 + 解析层 + 边缘入口」三段。注意这是**从产品文档与公开论文归纳出的共同形态**，不是某一家厂商的内部设计图；不同厂商的实现细节不公开，本章不臆测具体组件名与容量数字之外的细节。

37.4.1 三段职责

```
[控制台 / 管控层]
   添加域名、发放 Account ID 与密钥、配置自定义解析（SDNS）、
   按次计费与额度、解析变更下发、健康检查触发切流
         | 配置 / 策略 / 失效通知
         ▼
[解析层：全球部署的递归服务器集群]
   就近接入、多级递归节点、缓存与预取、按 client_ip / 地域 / 运营商
   决定返回哪组 IP、与权威 DNS / GTM 联动做秒级刷新
         |
         ▼
[边缘入口：Anycast / BGP 任播 IP]
   客户端 IP 直连（避免「先解析解析服务」的鸡生蛋问题），
   多集群互备、故障域隔离、DDoS 防护、SLA
```

产品文档把前两层写得很明确：HTTPDNS 是**递归 DNS 服务**，「与权威 DNS 不同，并不具备决定解析结果的能力，而是主要负责解析过程的实现」；解析层是全球部署的解析服务器集群，配大量二级递归节点覆盖主要城市。这句话值得和第 3 章的角色表并排读——大厂的 HTTPDNS 后端在角色上仍然是**递归解析器**，私有 API 只是它面向 App 的一层外观；真正的权威数据仍在各域名自己的权威服务器上。

这正好解释了本章自建方案为什么成立：Unbound 本来就是递归解析器，前端只是给它换了个入口协议。差别在于规模与管控——厂商多了控制台、策略下发、计费、全球边缘、容灾编排，而这些都不是 DNS 协议本身的能力。

37.4.2　与 CDN 的关系：三个判断要拆开

①它是 CDN 的入口优化层，不是 CDN 本身。CDN 靠 DNS 调度把用户导到边缘节点，LocalDNS 的 ECS 缺失、递归转发与 TTL 乱缓存会让调度失准；HTTPDNS 能看到真实客户端出口 IP，因此返回更准的节点。它只返回 IP，不缓存内容、不加速传输、不卸载源站。

②服务端的部署范式与 CDN 同构。Anycast 入口、全球多集群、就近接入、边缘节点、多地容灾——这些都是第 26 章讨论过的规模化手段，与 CDN 的分布式边缘架构是同一套工程。但同构不等于同物：一个是内容分发网络，一个是解析服务的接入层。

③它和 GTM / 权威 DNS 联动，但不替代它们。解析变更要能秒级刷新到边缘缓存，健康检查要能触发切流；这些动作的「事实来源」仍是权威区与全局流量管理配置。厂商文档明确写 HTTPDNS 不具备决定解析结果的能力，就是这条边界。

37.4.3　大规模解析集群的公开参照

最贴近「大厂解析层」的公开架构论文是 SIGCOMM 2020 的 Akamai DNS（权威侧，不是 HTTPDNS，但规模化范式直接可借鉴）。论文披露的要点：系统在 2019 年 12 月的一个观测窗口里服务约 3600 亿次 DNS 查询；用 24 组 IPv4/IPv6 anycast 前缀对构成 anycast cloud，每个 cloud 前挂路由器与 BGP speaker；每台机器运行专用 nameserver 软件并配本地健康监控，检测到问题时由 BGP speaker 撤回该 anycast 前缀的宣告；采用 **two-tier delegation**，为不同区从 anycast cloud 集合里选一个 delegation set，实现故障域隔离与快速重配置；另有独立的映射/监控系统持续发布元数据给 nameserver，决定「应该返回哪些 IP」。

把这套读成 HTTPDNS 后端的参照，有四条可以直接迁移：

- **入口与解析解耦**：客户端打的是任播 IP，真正干活的是后面的解析集群；入口挂了撤路由，不需要改客户端配置。本章的「IP 字面量入口 + 回环 Unbound」是这个结构的单机最小版。
- **策略与应答分离**：「返回哪些 IP」由独立的映射/健康系统决定，nameserver 订阅它。对应到自建方案，就是前端的自定义规则表与 Unbound 的 `local-data` 分开维护，别把业务策略塞进解析器的缓存。
- **故障域按区切分**：two-tier delegation 让不同区落在不同 anycast cloud 集合上，避免一次故障拖垮全部域名。第 28 章的故障域隔离在这里是同一条思路。
- **健康探测驱动撤路由**：本地 agent 连续探测，异常即 withdraw，比靠 TTL 自然收敛快得多。

要如实标注差异：Akamai DNS 是**权威**基础设施，HTTPDNS 后端是**递归**服务，两者在 DNS 层级里位置不同；论文里的 24 组任播前缀、two-tier delegation 是权威侧的设计，不能直接说成「某厂 HTTPDNS 就是这么搭的」。能借鉴的是规模化范式，不是逐字照搬的组件。ESORICS 2024 关于加密 DNS 基础设施依赖与集中化风险的测量工作则提供了另一面：入口高度集中本身就是新的故障域，多集群互备不是可选项。

37.4.4　为什么必然是「按次售卖的 SaaS + 各发一套 SDK」

把这三层合起来看，就能解释产品形态：边缘入口、全球集群、管控台、策略下发、计费与额度，都是重资产；而客户端必须改造网络库，且各家参数名、签名规范化串、加密方式、错误码互不兼容，换厂商要换 SDK、重做证书与 Host 处理。于是它天然长成「云厂商按解析次数售卖的 SaaS」，而不是一个可以标准化互操作的协议。微信小程序那一类「平台层代调第三方解析服务」（`enableHttpDNS` + 服务 ID）也是同一逻辑的变体：由平台统一接管，服务商须通过平台的安全认证，错误码由平台定义。

这也给出一条务实的选型边界：**自建 HTTPDNS 在工程上完全可行，但你要自己承担的不是「解析」，而是入口冗余、策略下发、SDK 适配、降级契约与容灾编排**。Unbound 把最难的递归与验证解决掉了，剩下的全是分布式系统工程。

37.5　它不可能覆盖 App 的全部解析

最后一节用来给本章的适用面封边。私有 API 在 App 里必然是**分层混合**，不是「全量替换」，原因有五条，全部是设计约束而非实现不彻底。

自举问题。解析服务自己的地址怎么找？主流做法是客户端内置一组 IP 字面量（Anycast），但 IP 变更需要发版，所以厂商都留「调度服务接口 / 服务 IP 动态更新」这条兜底，文档建议冷启动、网络切换、每 8 小时、全失败时各更新一次。那一次更新仍可能走系统 DNS。

把内置 IP 换成「直接向这个 IP 发起加密解析」，就是浏览器与 App 绕过本地解析器的那条路径：硬编码的 bootstrap 端点不走系统 DNS，UDP/53 的劫持规则自然抓不到它。它与第 13 章的 DoH bypass 是同一个工程问题，只是发起者从 App SDK 换成了浏览器或后台进程；那里的结论同样适用——按已知端点 IP 封 443 只是 denylist，不是「所有查询都经过本地解析器」的证明，验收要靠抓包看有没有外部加密连接、有没有明文 53 逃出去。HTTPDNS 因为压根不发 DNS 报文，连这层「绕过」都谈不上，它是另一条通道。

SDK 只能改自己控制的网络栈。 App 自己的 OkHttp / NSURLSession / Cronet 能接管；系统 WebView 里的 H5 只能靠平台专门方案（内核级 DNS 托管一类的能力）；广告、推送、埋点、登录、地图等第三方 SDK 各用自己的网络栈；APNs/FCM、NTP、captive portal、OCSP/CRL 完全不受 App 控制；小程序走平台层。典型 App 里十几个第三方 SDK，绝大多数仍走系统 DNS。

降级是规范，不是异常。所有厂商文档都写死：解析超时、返回空、鉴权失败、额度耗尽→必须回退 LocalDNS。既然兜底路径明写着要走系统 DNS，「一部分查询一定走 OS DNS」就不是缺陷。

代理与企业网络。系统 HTTP 代理开启时，直连 IP 会破坏代理语义；企业 DNS 视图、内网域、802.1X 认证网、IPv6-only 合成等场景，系统 DNS 才是正确路径。部分平台在检测到代理时主动强制关闭私有解析。

TLS 成本。拿到 IP 后要手工设 SNI、保留 Host、改证书校验，成本不低。多数团队只对核心 API / 支付 / 直播域名开启，图片、日志、统计、静态资源仍走系统 DNS。

所以厂商控制台那一步「添加域名到解析列表」不只是计费口径，它同时定义了覆盖范围：白名单之外的域名 SDK 根本不接管。整个形态是：

```
层 0  系统 DNS（必然存在，且先于一切）
      自举 / 调度接口、控制台与上报、推送、系统服务、第三方 SDK、
      降级兜底、代理与内网场景
层 1  私有 HTTP API（按域名白名单接管）
      核心业务 API、支付、登录、直播、需按 uid / 版本调度的域名
层 2  IP 直连（私有 API 的结果）
      URL 是 IP，Host / SNI 是原域名，按返回的 ttl 缓存
```

这也回到本章开头那条界线：HTTPDNS 绕开 LocalDNS 的同时，也绕开了本机过滤表、DNSSEC 验证与 dnstap 观测。对运维而言，判断「这一次解析归谁管」比判断「解析对不对」更靠前——第 32 章给出的排查顺序是「先定位改写者」，第 36 章的工具箱里 `dig @server` 会绕开存根与本地策略，在这里同样适用：**指定服务器看到的是那台服务器怎么答，不是 App 实际拿到了什么。**

37.6 本章要点

1. HTTPDNS 没有 RFC，它是应用层私有 API：载荷是 JSON / 纯文本 IP，不是 DNS wire format，因此 DNSSEC、负缓存、过滤表与 dnstap 全都挂不上去。DoH [RFC 8484] 仍是标准 DNS 报文，两者只是名字相近。
2. 引入动机是 LocalDNS 不可控的五个具体痛点：劫持与污染、权威 TTL 不生效导致变更慢、递归出口与客户端位置不符导致 CDN 调度失准、容灾切换粒度粗、业务身份无法参与调度。只想要链路加密时用 DoH 成本更低。
3. 自建 = **Unbound（递归 / 验证 / `local-zone` 自定义视图 / dnstap）+ HTTP(S) 前端（鉴权、取 `client_ip`、发标准 DNS 查询）+ 客户端改造（IP 直连、保留 `Host` 与 SNI、按 `ttl` 缓存、失败降级）**，三层缺一不可。前端与 Unbound 之间仍是标准 DNS 报文，私有性只在 App ↔前端那一段。
4. 前端必须做四道检查：域名白名单、时间窗、常量时间签名比较、`nonce` 去重；签名规范化串两端要用同一个 URL 编码函数；多节点时 nonce 存储须共享。教学骨架不追 CNAME 链、不做 TCP 回退，不能直接上生产。
5. 大厂后端可归纳为「管控层 + 递归解析层 + Anycast 边缘入口」三段：产品文档明确它是递归服务、不决定权威结果；Akamai DNS（SIGCOMM 2020）的任播前缀、two-tier delegation、健康探测撤路由、策略与应答分离，是可借鉴的规模化范式，但它是权威侧架构，不能逐字照搬。
6. 私有 API 在 App 里必然按域名白名单分层混合：自举、第三方 SDK、系统服务、代理场景与降级兜底都绕不开系统 DNS。控制台「添加域名」既是计费口径，也是覆盖范围。

37.7 参考资料

- RFC 8484, *DNS Queries over HTTPS (DoH)* —https://www.rfc-editor.org/rfc/rfc8484
- RFC 7871, *Client Subnet in DNS Queries* —https://www.rfc-editor.org/rfc/rfc7871
- RFC 5737, *IPv4 Address Blocks Reserved for Documentation* —https://www.rfc-editor.org/rfc/rfc5737

- 阿里云 EMAS, 《HTTP 接口自定义接入最佳实践》—https://help.aliyun.com/zh/document_detail/2868007.html
- 阿里云 EMAS,《客户端接入概述》—https://help.aliyun.com/zh/document_detail/2979687.html
- 阿里云 EMAS, 《DoH 接入指南》（私有协议与 RFC 8484 的官方对比）—https://help.aliyun.com/zh/document_detail/2979559.html
- 阿里云, 《HTTPDNS 产品简介》（C/S 适用、B/S 不适用、递归服务定位）—http://docs-aliyun.cn-hangzhou.oss.aliyun-inc.com/pdf/httpdns-product-introduction-cn-zh-2020-07-01.pdf
- 微信开放文档,《移动解析 HttpDNS》—https://developers.weixin.qq.com/miniprogram/dev/framework/ability/HTTPDNS.html
- Kyle Schomp 等, *Akamai DNS: Providing Authoritative Answers to the World' s Queries*, SIGCOMM 2020 —https://groups.cs.umass.edu/wp-content/uploads/sites/3/2020/07/sigcomm2020-final289.pdf
- RIPE-823, *DNS Resolver Recommendations* —https://www.ripe.net/publications/docs/ripe-823

附录 A　RFC 清单

数据来源: RFC Editor 官方 `rfc-index.txt`。状态取自索引的 `Status:` 字段;「有效」按是否被 `Obsoleted by` 替代判定，`Updated by` 表示有后续修订但仍有效。

RFC	名称	发布日期	状态	有效	描述对象
RFC 1034	Domain names - concepts and facilities	Nov 1987	INTERNET STANDARD (有修订)	有效	DNS 基础 / 术语 / 实现
RFC 1035	Domain names - implementation and specification	Nov 1987	INTERNET STANDARD (有修订)	有效	DNS 基础 / 术语 / 实现
RFC 1166	Internet numbers	Jul 1990	INFORMATIONAL (有修订)	有效	其他 / 网络基础设施
RFC 1912	Common DNS Operational and Configuration Errors	Feb 1996	INFORMATIONAL	有效	DNS 基础 / 术语 / 实现
RFC 1918	Address Allocation for Private Internets	Feb 1996	BEST CURRENT PRACTICE (有修订)	有效	其他 / 网络基础设施
RFC 1995	Incremental Zone Transfer in DNS	Aug 1996	PROPOSED STANDARD (有修订)	有效	DNS 基础 / 术语 / 实现
RFC 2132	DHCP Options and BOOTP Vendor Extensions	Mar 1997	DRAFT STANDARD (有修订)	有效	动态更新 / 事务认证
RFC 2136	Dynamic Updates in the Domain Name System (DNS UPDATE)	Apr 1997	PROPOSED STANDARD (有修订)	有效	动态更新 / 事务认证
RFC 2181	Clarifications to the DNS Specification	Jul 1997	PROPOSED STANDARD (有修订)	有效	DNS 基础 / 术语 / 实现
RFC 2219	Use of DNS Aliases for Network Services	Oct 1997	BEST CURRENT PRACTICE	有效	DNS 基础 / 术语 / 实现

RFC	名称	发布日期	状态	有效	描述对象
RFC 2308	Negative Caching of DNS Queries (DNS NCACHE)	Mar 1998	PROPOSED STANDARD (有修订)	有效	缓存 / 应答语义 / EDNS
RFC 2535	Domain Name System Security Extensions	Mar 1999	PROPOSED STANDARD (有修订)	已被替代	DNS 基础 / 术语 / 实现
RFC 2544	Benchmarking Methodology for Network Interconnect Devices	Mar 1999	INFORMATIONAL (有修订)	有效	其他 / 网络基础设施
RFC 2608	Service Location Protocol, Version 2	Jun 1999	PROPOSED STANDARD (有修订)	有效	其他 / 网络基础设施
RFC 2609	Service Templates and Service: Schemes	Jun 1999	PROPOSED STANDARD	有效	其他 / 网络基础设施
RFC 2671	Extension Mechanisms for DNS (EDNS0)	Aug 1999	PROPOSED STANDARD	已被替代	缓存 / 应答语义 / EDNS
RFC 2673	Binary Labels in the Domain Name System	Aug 1999	HISTORIC (有修订)	已被替代	DNS 基础 / 术语 / 实现
RFC 2765	Stateless IP/ ICMP Translation Algorithm (SIIT)	Feb 2000	PROPOSED STANDARD	已被替代	其他 / 网络基础设施
RFC 3008	Domain Name System Security (DNSSEC) Signing Authority	Nov 2000	PROPOSED STANDARD (有修订)	已被替代	DNSSEC / 信任链
RFC 3090	DNS Security Extension Clarification on Zone Status	Mar 2001	PROPOSED STANDARD (有修订)	已被替代	DNSSEC / 信任链
RFC 3397	Dynamic Host Configuration Protocol (DHCP) Domain Search Option	Nov 2002	PROPOSED STANDARD	有效	动态更新 / 事务认证
RFC 3445	Limiting the Scope of the KEY Resource Record (RR)	Dec 2002	PROPOSED STANDARD	已被替代	其他 / 网络基础设施

RFC	名称	发布日期	状态	有效	描述对象
RFC 3484	Default Address Selection for Internet Protocol version 6 (IPv6)	Mar 2003	PROPOSED STANDARD	已被替代	IPv6 / 地址与 RA
RFC 3597	Handling of Unknown DNS Resource Record (RR) Types	Sep 2003	PROPOSED STANDARD (有修订)	有效	DNS 基础 / 术语 / 实现
RFC 3645	Generic Security Service Algorithm for Secret Key Transaction Authentication for DNS (GSS-TSIG)	Oct 2003	PROPOSED STANDARD	有效	动态更新 / 事务认证
RFC 3646	DNS Configuration options for Dynamic Host Configuration Protocol for IPv6 (DHCPv6)	Dec 2003	PROPOSED STANDARD	有效	IPv6 / 地址与 RA
RFC 3655	Redefinition of DNS Authenticated Data (AD) bit	Nov 2003	PROPOSED STANDARD	已被替代	DNS 基础 / 术语 / 实现
RFC 3658	Delegation Signer (DS) Resource Record (RR)	Dec 2003	PROPOSED STANDARD (有修订)	已被替代	其他 / 网络基础设施
RFC 3755	Legacy Resolver Compatibility for Delegation Signer (DS)	May 2004	PROPOSED STANDARD (有修订)	已被替代	DNS 基础 / 术语 / 实现
RFC 3757	Domain Name System KEY (DNSKEY) Resource Record (RR) Secure Entry Point (SEP) Flag	May 2004	PROPOSED STANDARD	已被替代	DNSSEC / 信任链

RFC	名称	发布日期	状态	有效	描述对象
RFC 3845	DNS Security (DNSSEC) NextSECure (NSEC) RDATA Format	Aug 2004	PROPOSED STANDARD	已被替代	DNSSEC / 信任链
RFC 3849	IPv6 Address Prefix Reserved for Documentation	Jul 2004	INFORMATIONAL（有修订）	有效	IPv6 / 地址与 RA
RFC 3971	SEcure Neighbor Discovery (SEND)	Mar 2005	PROPOSED STANDARD（有修订）	有效	IPv6 / 地址与 RA
RFC 4033	DNS Security Introduction and Requirements	Mar 2005	PROPOSED STANDARD（有修订）	有效	DNSSEC / 信任链
RFC 4034	Resource Records for the DNS Security Extensions	Mar 2005	PROPOSED STANDARD（有修订）	有效	DNSSEC / 信任链
RFC 4035	Protocol Modifications for the DNS Security Extensions	Mar 2005	PROPOSED STANDARD（有修订）	有效	DNSSEC / 信任链
RFC 4191	Default Router Preferences and More-Specific Routes	Nov 2005	PROPOSED STANDARD	有效	IPv6 / 地址与 RA
RFC 4193	Unique Local IPv6 Unicast Addresses	Oct 2005	PROPOSED STANDARD	有效	IPv6 / 地址与 RA
RFC 4255	Using DNS to Securely Publish Secure Shell (SSH) Key Fingerprints	Jan 2006	PROPOSED STANDARD	有效	邮件 / 证书 / 公钥发布
RFC 4291	IP Version 6 Addressing Architecture	Feb 2006	DRAFT STANDARD（有修订）	有效	其他 / 网络基础设施
RFC 4343	Domain Name System (DNS) Case Insensitivity Clarification	Jan 2006	PROPOSED STANDARD（有修订）	有效	DNS 基础 / 术语 / 实现

RFC	名称	发布日期	状态	有效	描述对象
RFC 4431	The DNSSEC Lookaside Validation (DLV) DNS Resource Record	Feb 2006	HISTORIC	有效	DNSSEC / 信任链
RFC 4641	DNSSEC Operational Practices	Sep 2006	INFORMATIONAL	已被替代	DNSSEC / 信任链
RFC 4702	The Dynamic Host Configuration Protocol (DHCP) Client Fully Qualified Domain Name (FQDN) Option	Oct 2006	PROPOSED STANDARD	有效	动态更新 / 事务认证
RFC 4861	Neighbor Discovery for IP version 6 (IPv6)	Sep 2007	DRAFT STANDARD (有修订)	有效	IPv6 / 地址与 RA
RFC 4862	IPv6 Stateless Address Auto-configuration	Sep 2007	DRAFT STANDARD (有修订)	有效	IPv6 / 地址与 RA
RFC 4941	Privacy Extensions for Stateless Address Auto-configuration in IPv6	Sep 2007	DRAFT STANDARD	已被替代	查询隐私 / 加密传输
RFC 5011	Automated Updates of DNS Security (DNSSEC) Trust Anchors	Sep 2007	INTERNET STANDARD	有效	DNSSEC / 信任链
RFC 5014	IPv6 Socket API for Source Address Selection	Sep 2007	INFORMATIONAL	有效	IPv6 / 地址与 RA
RFC 5074	DNSSEC Lookaside Validation (DLV)	Nov 2007	HISTORIC	有效	DNSSEC / 信任链
RFC 5155	DNS Security (DNSSEC) Hashed Authenticated Denial of Existence	Mar 2008	PROPOSED STANDARD (有修订)	有效	DNSSEC / 信任链

RFC	名称	发布日期	状态	有效	描述对象
RFC 5358	Preventing Use of Recursive Nameservers in Reflector Attacks	Oct 2008	BEST CURRENT PRACTICE	有效	DNS 基础 / 术语 / 实现
RFC 5452	Measures for Making DNS More Resilient against Forged Answers	Jan 2009	PROPOSED STANDARD	有效	DNS 基础 / 术语 / 实现
RFC 5625	DNS Proxy Implementation Guidelines	Aug 2009	BEST CURRENT PRACTICE	有效	DNS 基础 / 术语 / 实现
RFC 5737	IPv4 Address Blocks Reserved for Documentation	Jan 2010	INFORMATIONAL	有效	特殊用途域名
RFC 5908	Network Time Protocol (NTP) Server Option for DHCPv6	Jun 2010	PROPOSED STANDARD	有效	动态更新 / 事务认证
RFC 5936	DNS Zone Transfer Protocol (AXFR)	Jun 2010	PROPOSED STANDARD (有修订)	有效	DNS 基础 / 术语 / 实现
RFC 6106	IPv6 Router Advertisement Options for DNS Configuration	Nov 2010	PROPOSED STANDARD	已被替代	IPv6 / 地址与 RA
RFC 6146	Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers	Apr 2011	PROPOSED STANDARD	有效	IPv6 / 地址与 RA
RFC 6147	DNS64: DNS Extensions for Network Address Translation from IPv6 Clients to IPv4 Servers	Apr 2011	PROPOSED STANDARD	有效	IPv6 / 地址与 RA

RFC	名称	发布日期	状态	有效	描述对象
RFC 6186	Use of SRV Records for Locating Email Submission/ Access Services	Mar 2011	PROPOSED STANDARD (有修订)	有效	邮件 / 证书 / 公钥发布
RFC 6376	DomainKeys Identified Mail (DKIM) Signatures	Sep 2011	INTERNET STANDARD (有修订)	有效	邮件 / 证书 / 公钥发布
RFC 6570	URI Template	Mar 2012	PROPOSED STANDARD	有效	其他 / 网络基础设施
RFC 6724	Default Address Selection for Internet Protocol Version 6 (IPv6)	Sep 2012	PROPOSED STANDARD	有效	IPv6 / 地址与 RA
RFC 6731	Improved Recursive DNS Server Selection for Multi-Interfaced Nodes	Dec 2012	PROPOSED STANDARD	有效	DNS 基础 / 术语 / 实现
RFC 6761	Special-Use Domain Names	Feb 2013	PROPOSED STANDARD	有效	特殊用途域名
RFC 6762	Multicast DNS	Feb 2013	PROPOSED STANDARD	有效	服务发现 / 多播 / DDR
RFC 6763	DNS-Based Service Discovery	Feb 2013	PROPOSED STANDARD (有修订)	有效	服务发现 / 多播 / DDR
RFC 6781	DNSSEC Operational Practices, Version 2	Dec 2012	INFORMATIONAL	有效	DNSSEC / 信任链
RFC 6840	Clarifications and Implementation Notes for DNS Security (DNSSEC)	Feb 2013	PROPOSED STANDARD (有修订)	有效	DNSSEC / 信任链
RFC 6844	DNS Certification Authority Authorization (CAA) Resource Record	Jan 2013	PROPOSED STANDARD	已被替代	邮件 / 证书 / 公钥发布
RFC 6891	Extension Mechanisms for DNS (EDNS(0))	Apr 2013	INTERNET STANDARD	有效	缓存 / 应答语义 / EDNS

RFC	名称	发布日期	状态	有效	描述对象
RFC 6944	Applicability Statement: DNS Security (DNSSEC) DNSKEY Algorithm Implementation Status	Apr 2013	PROPOSED STANDARD	已被替代	DNSSEC / 信任链
RFC 6975	Signaling Cryptographic Algorithm Understanding in DNS Security Extensions (DNSSEC)	Jul 2013	PROPOSED STANDARD	有效	DNSSEC / 信任链
RFC 7129	Authenticated Denial of Existence in the DNS	Feb 2014	INFORMATIONAL	有效	DNS 基础 / 术语 / 实现
RFC 7208	Sender Policy Framework (SPF) for Authorizing Use of Domains in Email, Version 1	Apr 2014	PROPOSED STANDARD (有修订)	有效	邮件 / 证书 / 公钥发布
RFC 7301	Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension	Jul 2014	PROPOSED STANDARD (有修订)	有效	其他 / 网络基础设施
RFC 7583	DNSSEC Key Rollover Timing Considerations	Oct 2015	INFORMATIONAL	有效	DNSSEC / 信任链
RFC 7672	SMTP Security via Opportunistic DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS)	Oct 2015	PROPOSED STANDARD	有效	邮件 / 证书 / 公钥发布
RFC 7686	The ".onion" Special-Use Domain Name	Oct 2015	PROPOSED STANDARD	有效	特殊用途域名
RFC 7719	DNS Terminology	Dec 2015	INFORMATIONAL	已被替代	DNS 基础 / 术语 / 实现

RFC	名称	发布日期	状态	有效	描述对象
RFC 7766	DNS Transport over TCP - Implementation Requirements	Mar 2016	PROPOSED STANDARD（有修订）	有效	DNS 基础 / 术语 / 实现
RFC 7816	DNS Query Name Minimisation to Improve Privacy	Mar 2016	EXPERIMENTAL	已被替代	查询隐私 / 加密传输
RFC 7830	The EDNS(0) Padding Option	May 2016	PROPOSED STANDARD	有效	查询隐私 / 加密传输
RFC 7858	Specification for DNS over Transport Layer Security (TLS)	May 2016	PROPOSED STANDARD（有修订）	有效	查询隐私 / 加密传输
RFC 7871	Client Subnet in DNS Queries	May 2016	INFORMATIONAL	有效	查询隐私 / 加密传输
RFC 7873	Domain Name System (DNS) Cookies	May 2016	PROPOSED STANDARD（有修订）	有效	查询隐私 / 加密传输
RFC 7958	DNSSEC Trust Anchor Publication for the Root Zone	Aug 2016	INFORMATIONAL	已被替代	DNSSEC / 信任链
RFC 8020	NXDOMAIN: There Really Is Nothing Underneath	Nov 2016	PROPOSED STANDARD	有效	缓存 / 应答语义 / EDNS
RFC 8106	IPv6 Router Advertisement Options for DNS Configuration	Mar 2017	PROPOSED STANDARD	有效	IPv6 / 地址与 RA
RFC 8200	Internet Protocol, Version 6 (IPv6) Specification	Jul 2017	INTERNET STANDARD（有修订）	有效	IPv6 / 地址与 RA
RFC 8310	Usage Profiles for DNS over TLS and DNS over DTLS	Mar 2018	PROPOSED STANDARD	有效	查询隐私 / 加密传输
RFC 8314	Cleartext Considered Obsolete: Use of Transport Layer Security (TLS) for Email Submission and Access	Jan 2018	PROPOSED STANDARD（有修订）	有效	邮件 / 证书 / 公钥发布

RFC	名称	发布日期	状态	有效	描述对象
RFC 8375	Special-Use Domain 'home.arpa.'	May 2018	PROPOSED STANDARD	有效	特殊用途域名
RFC 8415	Dynamic Host Configuration Protocol for IPv6 (DHCPv6)	Nov 2018	PROPOSED STANDARD	已被替代	IPv6 / 地址与 RA
RFC 8461	SMTP MTA Strict Transport Security (MTA-STS)	Sep 2018	PROPOSED STANDARD	有效	邮件 / 证书 / 公钥发布
RFC 8484	DNS Queries over HTTPS (DoH)	Oct 2018	PROPOSED STANDARD	有效	查询隐私 / 加密传输
RFC 8499	DNS Terminology	Jan 2019	BEST CURRENT PRACTICE	已被替代	DNS 基础 / 术语 / 实现
RFC 8624	Algorithm Implementation Requirements and Usage Guidance for DNSSEC	Jun 2019	PROPOSED STANDARD (有修订)	已被替代	DNSSEC / 信任链
RFC 8683	Additional Deployment Guidelines for NAT64/464XLAT in Operator and Enterprise Networks	Nov 2019	INFORMATIONAL	有效	其他 / 网络基础设施
RFC 8766	Discovery Proxy for Multicast DNS-Based Service Discovery	Jun 2020	PROPOSED STANDARD	有效	服务发现 / 多播 / DDR
RFC 8767	Serving Stale Data to Improve DNS Resiliency	Mar 2020	PROPOSED STANDARD	有效	缓存 / 应答语义 / EDNS
RFC 8914	Extended DNS Errors	Oct 2020	PROPOSED STANDARD	有效	DNS 基础 / 术语 / 实现
RFC 8932	Recommendations for DNS Privacy Service Operators	Oct 2020	BEST CURRENT PRACTICE	有效	查询隐私 / 加密传输

RFC	名称	发布日期	状态	有效	描述对象
RFC 8971	Bidirectional Forwarding Detection (BFD) for Virtual eXtensible Local Area Network (VXLAN)	Dec 2020	INFORMATIONAL	有效	其他 / 网络基础设施
RFC 8978	Reaction of IPv6 Stateless Address Auto-configuration (SLAAC) to Flash-Renumbering Events	Mar 2021	INFORMATIONAL	有效	IPv6 / 地址与 RA
RFC 8981	Temporary Address Extensions for Stateless Address Auto-configuration in IPv6	Feb 2021	PROPOSED STANDARD	有效	IPv6 / 地址与 RA
RFC 9076	DNS Privacy Considerations	Jul 2021	INFORMATIONAL	有效	查询隐私 / 加密传输
RFC 9096	Improving the Reaction of Customer Edge Routers to IPv6 Renumbering Events	Aug 2021	BEST CURRENT PRACTICE	有效	IPv6 / 地址与 RA
RFC 9103	DNS Zone Transfer over TLS	Aug 2021	PROPOSED STANDARD	有效	DNS 基础 / 术语 / 实现
RFC 9156	DNS Query Name Minimisation to Improve Privacy	Nov 2021	PROPOSED STANDARD	有效	查询隐私 / 加密传输
RFC 9180	Hybrid Public Key Encryption	Feb 2022	INFORMATIONAL	有效	其他 / 网络基础设施
RFC 9230	Oblivious DNS over HTTPS	Jun 2022	EXPERIMENTAL	有效	查询隐私 / 加密传输
RFC 9250	DNS over Dedicated QUIC Connections	May 2022	PROPOSED STANDARD	有效	查询隐私 / 加密传输
RFC 9276	Guidance for NSEC3 Parameter Settings	Aug 2022	BEST CURRENT PRACTICE	有效	DNSSEC / 信任链

RFC	名称	发布日期	状态	有效	描述对象
RFC 9458	Oblivious HTTP	Jan 2024	PROPOSED STANDARD	有效	其他 / 网络基础设施
RFC 9460	Service Binding and Parameter Specification via the DNS (SVCB and HTTPS Resource Records)	Nov 2023	PROPOSED STANDARD	有效	DNS 基础 / 术语 / 实现
RFC 9462	Discovery of Designated Resolvers	Nov 2023	PROPOSED STANDARD	有效	服务发现 / 多播 / DDR
RFC 9665	Service Registration Protocol for DNS-Based Service Discovery	Jun 2025	PROPOSED STANDARD	有效	服务发现 / 多播 / DDR

本书正文只引用编号与结论；发布日期、状态与替代关系请以 RFC Editor 索引为准。

附录 B　缩略语

按字母顺序列出本书出现的缩略语。同一缩写在 DNS 语境下可能有歧义（例如 `AD` 既是 DNS 标志位也是 Active Directory），表中按本书用法给出。

缩写	全称	中文 / 说明
A / AAAA	Address / Address (IPv6)	地址记录，IPv4 / IPv6
ACL	Access Control List	访问控制列表；Unbound 的 `access-control`、AD 集成区的 ACL
ACME	Automatic Certificate Management Environment	自动证书管理环境；DNS-01 挑战用 `_acme-challenge` TXT
AD	Authenticated Data	DNS 报文标志位：验证器确认答案通过 DNSSEC 验证
AD	Active Directory	微软目录服务（与上一行同缩写，语境区分）
ADNS	Authoritative DNS	权威 DNS（非正式缩写，本书只作行文简写）
ALPN	Application-Layer Protocol Negotiation	TLS 应用层协议协商；HTTPS/SVCB 记录里的 `alpn` 参数
anycast	—	任播：多个节点宣告同一服务地址
AXFR	Authoritative Zone Transfer	全量区域传送
BGP	Border Gateway Protocol	边界网关协议；anycast 与权威路由宣告的基础
BIND	Berkeley Internet Name Domain	最广泛部署的 DNS 实现
CAA	Certification Authority Authorization	限定可为域名签发证书的 CA
CD	Checking Disabled	DNS 报文标志位：请求验证器跳过验证
CLAT	Customer-side Translator	464XLAT 的客户端侧翻译，[RFC 8683]
CNAME	Canonical Name	别名记录；悬空 CNAME 是子域接管的常见入口
CoreDNS	—	Kubernetes 默认的插件化 DNS 服务器
Corefile	—	CoreDNS 的配置文件
CSRC	Computer Security Resource Center	NIST 下属，SP 800 系列发布方
CT	Certificate Transparency	证书透明日志；资产排查时的子域来源
DANE	DNS-based Authentication of Named Entities	用 DNSSEC 保护的 TLSA 记录绑定证书
DDNS	Dynamic DNS	动态域名；常指 [RFC 2136] UPDATE 或公网前缀更新脚本（两者不同）
DDR	Discovery of Designated Resolvers	[RFC 9462]，`_dns.resolver.arpa` 自动发现加密解析器

缩写	全称	中文 / 说明
delv	DNSSEC validation tool (BIND)	从验证器视角看应答为何通过/为何 bogus
DHCID	DHCP Client Identifier	DHCP 代注册时防止名字被冒用的记录
DHCP	Dynamic Host Configuration Protocol	动态主机配置协议；option 252 与 WPAD 相关
DHCPv6	Dynamic Host Configuration Protocol for IPv6	下发 option 23/24/56 等，[RFC 3646] / [RFC 5908] 一类
DKIM	DomainKeys Identified Mail	邮件签名；公钥发布在 `selector._domainkey` TXT
DMARC	Domain-based Message Authentication, Reporting and Conformance	基于 SPF/DKIM 对齐的邮件策略
DNAME	Delegation Name	整个子树的别名重定向
DNS-SD	DNS-Based Service Discovery	用 PTR/SRV/TXT 做服务发现，[RFC 6763]
DNS64	DNS64	[RFC 6147]，从 A 记录合成 AAAA 供 NAT64 使用
dnstap	—	DNS 查询/应答的结构化日志协议
dnstracer	—	逐级追踪委派链；名字是 dnstracer，不是 dnstrace
DoH	DNS over HTTPS	[RFC 8484]，DNS 报文走 HTTPS
DoQ	DNS over Dedicated QUIC Connections	[RFC 9250]
DoT	DNS over TLS	[RFC 7858]，端口 853
drill	DNS lookup tool (ldns)	NLnet Labs 的 ldns 工具，可追 DNSSEC 链
DS	Delegation Signer	父区里对子区 KSK 的哈希，DNSSEC 信任链的一环
eBPF	extended Berkeley Packet Filter	内核可编程观测；不经应用改代码即可看 UDP/53 与丢包
ECH	Encrypted Client Hello	TLS ClientHello 加密；可由 HTTPS 记录发布公钥
ECS	EDNS Client Subnet	[RFC 7871]，把客户端网段带给权威，隐私代价明显
EDNS(0)	Extension Mechanisms for DNS	[RFC 6891]；`edns-buffer-size`、DO 位、Cookie 都在这里
fake-ip	—	代理客户端用保留段占位地址延迟「域名→ IP」映射（198.18.0.0/15 为常见示例段）
gai.conf	getaddrinfo configuration	影响 [RFC 6724] 下的地址排序与 getaddrinfo 返回顺序
getaddrinfo	—	libc 名字解析接口；返回列表是**策略排序**，不是可用性排序
GSS-TSIG	Generic Security Service TSIG	[RFC 3645]，微软安全动态更新用 Kerberos 协商
HINFO	Host Information	主机信息记录，现代基本弃用
HTTPS RR	HTTPS Service Binding	[RFC 9460] 的 HTTPS 专用 SVCB 形式
IANA	Internet Assigned Numbers Authority	号码与协议参数分配机构
IETF	Internet Engineering Task Force	RFC 与 Internet-Draft 的标准化组织
IXFR	Incremental Zone Transfer	增量区域传送

缩写	全称	中文 / 说明
KSK	Key Signing Key	DNSSEC 区签名密钥中的密钥签名密钥
LLQ	Long-Lived Queries	DNS-SD 的长轮询更新机制
mDNS	multicast DNS	[RFC 6762]，链路本地多播名字解析
MTA-STS	MTA Strict Transport Security	强制 SMTP 走校验过的 TLS
MX	Mail Exchanger	邮件交换器记录
NAT64	Network Address Translation IPv6 → IPv4	[RFC 6146]，与 DNS64 配合
NIST	National Institute of Standards and Technology	美国 NIST，SP 800-81r3 发布方
NSEC / NSEC3	Next Secure (v3)	DNSSEC 的”否认存在”证明；NSEC3 用哈希名
NSEC walking	—	借 DNSSEC 否认存在记录枚举签名区名字
NSS	Name Service Switch	`/etc/nsswitch.conf` 决定 getent 先读 hosts 还是 DNS
NXDOMAIN	Non-Existent Domain	名字不存在的应答码
ODoH	Oblivious DNS over HTTPS	[RFC 9230]，Experimental / Independent Submission；relay 与 target 分离的两跳加密解析
OOB	Out-of-Band	带外管理通道
PAC	Proxy Auto-Config	`FindProxyForURL()` 脚本，WPAD 找到的目标
PVD	Provisioning Domain	网络侧下发配置域的概念
QMIN	QNAME Minimisation	查询名最小化，[RFC 7816] → [RFC 9156]
QNAME / QTYPE / QCLASS	Query Name / Type / Class	DNS 问题段的三个字段
RA	Recursion Available	DNS 报文标志位：服务器支持递归
RCODE	Response Code	DNS 应答码（NOERROR / NXDOMAIN / SERVFAIL 等）
RD	Recursion Desired	DNS 报文标志位：客户端请求递归
RDNSS / DNSSL	Recursive DNS Server / DNS Search List	[RFC 8106] 的 RA 选项，与 DHCPv6 23/24 并行
resolvdns	—	一类 SSH/代理客户端的 DNS 归属开关（yes 交给对端 / no 本地解析）；OpenSSH 无同名选项
resolvectl	—	systemd-resolved 的查询/状态命令（未装时读 /etc/resolv.conf）
RPZ	Response Policy Zone	基于策略改写/阻断应答
RPZ	Response Policy Zone	策略改写/阻断应答，蜜罐 sinkhole 常用
RRL	Response Rate Limiting	权威侧应答限速，抑制放大与扫描
RRSIG	Resource Record Signature	DNSSEC 签名记录
sinkhole	—	把恶意/诱饵域导向受控解析器以记录查询
SLP	Service Location Protocol	[RFC 2608]，WPAD 草案里的发现机制之一
socks5h	SOCKS5 with remote hostname	curl 中把主机名交给代理解析；`socks5://` 是本地解析
SPF	Sender Policy Framework	邮件发件方授权，TXT 记录
SRP	Service Registration Protocol	[RFC 9665]，纯单播的 DNS-SD 注册
SRV	Service Location	`_service._proto` 形式的服务定位记录

缩写	全称	中文 / 说明
SSHFP	SSH Host Key Fingerprint	[RFC 4255] 一类，把 SSH 公钥指纹放 DNS（需 DNSSEC）
SVCB	Service Binding	[RFC 9460] 通用服务绑定记录
TC	TrunCation	DNS 报文标志位；mDNS 查询里语义变为 Known-Answer 续包
TKEY	Transaction Key	建立 TSIG 安全上下文的记录类型
TLS-RPT	SMTP TLS Reporting	SMTP TLS 失败报告
TLSA	TLS Association	DANE 的证书/公钥绑定记录
TOFU	Trust On First Use	首次使用时信任（SSH 默认行为；SSHFP+DNSSEC 才摆脱它）
TSIG	Transaction Signature	DNS 消息级 HMAC 认证，用于 AXFR / UPDATE
TTL	Time To Live	缓存存活时间
TXT	Text	任意文本记录，承载 SPF/DKIM/DMARC/ACME 等
UDP / TCP	User Datagram / Transmission Control Protocol	DNS 的两种传输
ULA	Unique Local Address	[RFC 4193] 的 IPv6 私有点对点地址段
W-Bad	WPAD Name Collision / Bad WPAD	学界对 WPAD 名称冲突攻击的称呼
WPAD	Web Proxy Auto-Discovery Protocol	1999 年过期的 IETF 草案，未成为 RFC
ZSK	Zone Signing Key	DNSSEC 区签名密钥中签数据的那一把
464XLAT	—	CLAT + PLAT 组合，让纯 v6 接入网里的 v4-only 应用可达

本表不收录只出现一次且不构成术语的产品名（如某具体云厂商、工具名）。工具名在正文首次出现时给出全称或用途说明。

版本信息

本页为电子书的自动生成元数据，会出现在 PDF / ePub / HTML 三个格式的最后一页。

字段	值
当前版本	v0.7.2
下一待发布版本	v0.8.0
发布日期	2026 年 11 月 16 日
仓库	https://github.com/LeisureLinux/ebook-unbound
在线阅读	https://leisurelinux.github.io/ebook-unbound/
许可	CC BY-SA 4.0（正文）—© 2026 @LeisureLinux

本次版本修订记录 (v0.7.2)

- v0.7.2: 并发竞速的上游隐私代价、bootstrap DoH IP 与 DoH bypass 交叉引用

历史版本

版本	日期	主要变更
v0.7.2	2026-11-11	v0.7.2: 并发竞速的上游隐私代价、bootstrap DoH IP 与 DoH bypass 交叉引用
v0.7.1	2026-11-10	v0.7.1: 篇插图、考证与概念补正、去重复声明与自我指涉
v0.7.0	2026-11-09	fix(ci): ePub 目录按 ol 嵌套深度识别顶层 li，兼容 Pandoc 3.1 / 3.7
v0.6.0	2026-11-09	feat: 第 37 章 HTTPDNS；PDF 字体符号修复；章数同步到 37
v0.4.0	2026-11-07	feat: 35 chapters —proxy-transition DNS, DHCP options, red-team recon, DNS honeypot
v0.3.0	2026-11-07	feat: 31 chapters + appendices A1/A2 + TOC hardening (v0.3.0 candidate)
v0.2.0	2026-11-07	feat: add DDNS/MSDNS, anycast, mDNS and fault-domain isolation chapters (28 total)
v0.1.0	2026-11-07	fix(ci): quote workflow step name (invalid YAML colon)

本书使用 Pandoc + XeLaTeX 构建，字体采用 Noto Serif CJK SC，通过 GitHub Actions 自动构建并发布到 GitHub Pages。正文以 CC BY-SA 4.0 发布，构建脚本为 MIT。