从一台 VPS 到邮箱验证码：自建发信链路与轻量账号系统

记录如何用 Cloudflare DNS、Postfix、OpenDKIM、Nginx 和 Netlify Functions 搭建一条只负责发送验证码的邮件链路，并解释 SPF、DKIM、DMARC、PTR、密码哈希与会话 Cookie 背后的安全边界。

2026年7月29日 · Email · Postfix · DKIM · Netlify Functions · 身份认证

为了给 Freshmark 的评论区加入注册和登录，我需要解决一个看似简单的问题：向用户发送六位邮箱验证码。

购买现成的事务邮件服务当然最省事，但我手边已经有一台 VPS，也想借这个机会弄清一封邮件究竟怎样从自己的程序走到 Gmail。最后得到的系统很小：

```
浏览器
  │  异步注册请求
  ▼
Netlify Function
  ├─ 生成验证码
  ├─ 保存验证码和密码的 scrypt 哈希
  └─ HTTPS + Bearer Token
                │
                ▼
VPS 上的 Node 邮件桥接（127.0.0.1:8788）
  │  固定模板写入 sendmail
  ▼
Postfix ── OpenDKIM 签名 ── TLS ── 收件服务器
```

这不是一套通用邮箱服务。它不接收邮件、不提供 IMAP、不允许任意发件人和任意正文，只完成一项工作：把注册验证码交给本机 Postfix 队列。缩小目标之后，配置和攻击面都会清楚很多。

先分清“转发邮箱”和“发信服务器”

我的域名邮箱使用 Cloudflare Email Routing：

```
me@example.org → personal-account@gmail.com
```

这解决的是**收信**：Cloudflare 在域名的 MX 记录上接收邮件，再转发到真实邮箱。它并不意味着 VPS 可以天然以 `noreply@example.org` 的身份发送邮件。

发信是另一条链路。接收方通常会检查：

- 连接过来的 IP 是否被该发件域名的 SPF 授权；
- 邮件是否带有可验证的 DKIM 签名；
- SPF 或 DKIM 是否和用户看到的 From 域名对齐；
- 域名的 DMARC 策略要求怎样处理验证失败的邮件；
- 发送 IP 的 PTR、HELO、历史信誉和邮件内容是否可信。

因此，原有的 Cloudflare MX 记录应当保留。我们只为发信增加记录，不去破坏 Email Routing。

DNS：四类记录各管一件事

假设站点是 `blog.example.org`，发信主机是 `mail.example.org`，VPS 地址用文档专用地址 `203.0.113.10` 表示。Cloudflare 中与发信有关的记录可以整理成：

类型	名称	示例内容	用途
A	`mail`	`203.0.113.10`	让发信主机名指向 VPS
TXT	`mailer`	`v=spf1 ip4:203.0.113.10 -all`	授权 VPS 使用 `mailer.example.org` 作为信封发件域
TXT	`mail._domainkey`	`v=DKIM1; k=rsa; p=...`	发布 DKIM 公钥
TXT	`_dmarc`	`v=DMARC1; p=none; rua=mailto:...`	声明验证与报告策略

邮件相关主机应使用 Cloudflare 的 **DNS only**，不能开启橙色云代理。Cloudflare 的普通 HTTP 代理不会替你代理 SMTP。

为什么 SPF 放在 `mailer` 子域

邮件有两个容易混淆的“发件人”：

```
From: Freshmark <noreply@example.org>
Return-Path: bounce@mailer.example.org
```

`From` 是用户在邮件客户端看到的身份；`Return-Path` 来自 SMTP 信封发件人，用于接收退信。SPF 通常检查后者，所以我把信封发件域固定为 `mailer.example.org`，并在这个子域发布：

```
v=spf1 ip4:203.0.113.10 -all
```

`-all` 表示除了列出的地址之外，其余来源都不被授权。这样不会和根域上 Cloudflare Email Routing 使用的 SPF 记录混在一起，也避免一个域名出现两条 SPF 记录——后者会产生 `PermError`。

DKIM 是怎样工作的

OpenDKIM 在 VPS 上保存私钥，并在邮件头中加入签名。DNS 只发布对应的公钥。签名头里最重要的两个字段是：

```
d=example.org
s=mail
```

它们告诉接收方到下面的位置寻找公钥：

```
mail._domainkey.example.org
```

私钥永远不进入 DNS，也不应提交到 Git。生成密钥后，可用下面的命令检查 DNS 中的公钥是否匹配：

```
opendkim-testkey -d example.org -s mail -vvv
```

DMARC 不等于“反垃圾邮件通行证”

DMARC 检查的是**对齐**。只要通过的 SPF 身份或 DKIM 的 d= 域与 `From` 域满足对齐要求，就能通过 DMARC。

刚部署时适合先使用观察策略：

```
v=DMARC1; p=none; rua=mailto:dmarc@example.org
```

确认合法邮件稳定通过后，再逐步考虑 `quarantine` 或 `reject`。一开始就使用强制拒绝策略，配置中的一个拼写错误便可能让自己的邮件全部被拒收。

最容易忽略的 PTR、HELO 与 IP 信誉

A 记录把主机名解析到 IP，PTR（反向 DNS、rDNS）则把 IP 解析回主机名：

```
mail.example.org → 203.0.113.10
203.0.113.10 → mail.example.org
```

PTR 不在 Cloudflare DNS 中设置，因为 IP 地址属于 VPS 服务商。它通常位于服务商控制面板，找不到时只能提交工单。

理想状态是：

```
PTR = mail.example.org
Postfix HELO = mail.example.org
A(mail.example.org) = VPS IP
```

如果暂时不能修改 PTR，可以让 Postfix 的出站 HELO 使用服务商现有的 PTR 主机名，并确认该主机名能正向解析回同一 IP。这不是最终最漂亮的状态，但比 HELO 与反向解析完全矛盾更可靠。

即使 SPF、DKIM、DMARC 全部通过，Gmail 仍可能把邮件放进垃圾箱。认证证明“这封邮件确实由获授权的服务器发出”，却不证明“这台新服务器长期发送用户想看的邮件”。新 IP 缺少信誉、PTR 名称通用、发送量突然变化、模板过于像垃圾邮件，都会影响分类。

Postfix：只做出站 MTA

在 Ubuntu/Debian 上安装 Postfix 和 OpenDKIM：

```
apt update
apt install postfix opendkim opendkim-tools
```

这台机器不承担公网收信，所以 Postfix 只监听回环地址：

```
inet_interfaces = loopback-only
```

还应确保只有本机被信任，并保留默认的中继限制：

```
mynetworks = 127.0.0.0/8 [::1]/128
smtpd_relay_restrictions = permit_mynetworks, reject_unauth_destination
```

这两个边界非常重要。如果误配成允许互联网中的任意客户端中继邮件，服务器会成为 open relay，很快被滥用并进入黑名单。

出站 TLS 可以采用机会式加密：

```
smtp_tls_security_level = may
smtp_tls_loglevel = 1
```

它会在对方支持 STARTTLS 时加密传输。验证码本身仍应被视为短期秘密，所以应用层还必须设置短有效期和尝试次数限制，不能只依赖传输加密。

用 OpenDKIM 接入 Postfix

OpenDKIM 可以只监听本机端口：

```
Socket inet:8891@localhost
```

典型的签名映射如下：

```
KeyTable      mail._domainkey.example.org example.org:mail:/etc/opendkim/keys/example.org/
mail.private
SigningTable  *@example.org mail._domainkey.example.org
```

然后让 Postfix 把邮件交给 milter：

```
smtpd_milters = inet:127.0.0.1:8891
non_smtpd_milters = inet:127.0.0.1:8891
milter_default_action = accept
milter_protocol = 6
```

验证码邮件由本机 `sendmail` 命令提交，因此 `non_smtpd_milters` 不能漏掉，否则从本机队列发出的邮件可能没有 DKIM 签名。

修改后逐项检查：

```
postfix check
postconf -n
systemctl restart opendkim postfix
systemctl is-active opendkim postfix
ss -lntp
```

公网的 `0.0.0.0:25` 不应出现；本地的 `127.0.0.1:25` 和 OpenDKIM socket 应当存在。

为什么不让 Netlify Function 直接执行 sendmail

Netlify Function 与 VPS 不在同一台机器上，它无法访问 VPS 的本地 Postfix socket。反过来，把 SMTP 提交服务完整暴露到公网，又要处理 SMTP AUTH、证书、暴力破解和账户权限。

我的折中方案是一个很窄的 HTTPS 桥接：

```
POST https://mail.example.org/api/mail/comment-code
Authorization: Bearer <随机长令牌>
Content-Type: application/json

{
  "to": "reader@example.net",
  "code": "123456",
  "locale": "zh",
```

```
  "purpose": "registration"
}
```

桥接服务只接受：

- 一个固定路径和 POST 方法；
- HTTPS 反向代理后的请求；
- 与服务器共享的 Bearer Token；
- 最大 4 KiB 的 JSON；
- 合法邮箱、六位数字验证码和有限的用途字段；
- 每个收件地址每小时不超过固定次数。

未授权请求直接返回 404，而不是暴露“令牌错误”之类的信息。邮件的 `From`、信封发件人、主题和正文模板全部由服务器决定，调用方不能注入任意邮件头或正文。

Node 桥接如何安全调用 sendmail

核心操作不是拼接 shell 字符串，而是使用参数数组启动进程：

```
spawn("/usr/sbin/sendmail", ["-i", "-f", envelopeSender, "--", to])
```

这里有三个细节：

- `-i` 防止正文中单独一行的句点提前终止邮件；
- `-f` 明确设置用于 SPF 和退信的信封发件人；
- `--` 表示后面的收件地址不再解析为命令选项。

程序通过标准输入写入完整的 RFC 5322 邮件，主题用 MIME encoded-word 编码，正文声明 UTF-8。子进程设置十秒超时，错误输出也限制长度，避免异常进程无限挂起或日志无限增长。

服务只绑定：

```
127.0.0.1:8788
```

Nginx 对公网只开放精确 location：

```
location = /api/mail/comment-code {
    proxy_pass http://127.0.0.1:8788;
    client_max_body_size 4k;
    proxy_connect_timeout 2s;
    proxy_read_timeout 12s;
}
```

其他路径统一返回 404。HTTPS 证书由 Certbot 维护，Bearer Token 存放在 root 才能读取的环境文件中，并在 Netlify 中配置为 Functions secret。

systemd：即使程序出错，也限制它能做什么

邮件桥接不需要以 root 身份运行。专用用户只需要读取程序和令牌，并通过 Postfix 的 `postdrop` 机制写入队列。

systemd 可以进一步收紧权限：

```
[Service]
User=freshmail
Group=freshmail
SupplementaryGroups=postdrop
NoNewPrivileges=true
PrivateTmp=true
ProtectSystem=strict
ProtectHome=true
ReadWritePaths=/var/spool/postfix/maildrop /var/spool/postfix/public
RestrictSUIDSGID=true
Restart=on-failure
```

`ProtectSystem=strict` 会把大部分文件系统变为只读，因此必须显式放行 Postfix 提交邮件所需的目录。若 `sendmail` 一直报告 maildrop 是只读文件系统，通常就是这里少写了路径，而不是 Postfix 本身坏了。

从邮箱验证码到真正的登录状态

最初的方案是：某个邮箱验证过一次后，Functions 永久记住它，以后只要填写这个邮箱就放行。它很方便，却有一个根本问题：

> 知道一个已验证邮箱地址的人，也能冒用该地址发表评论。

所以最终实现的是轻量账号，而不是邮箱白名单。

注册阶段

浏览器异步提交昵称、邮箱和密码后，Function 完成：

1. 规范化邮箱并检查字段长度；
2. 为密码生成随机盐，异步计算 scrypt 哈希；
3. 生成密码学安全的六位验证码；
4. 用另一份随机盐计算验证码的 scrypt 哈希；

5. 把待注册记录写入 Netlify Blobs；
6. 调用 VPS 邮件桥接发送验证码。

Blob 中没有明文密码和验证码：

```
{
  "email": "reader@example.net",
  "password": {
    "salt": "<random>",
    "digest": "<scrypt digest>"
  },
  "verification": {
    "salt": "<another random value>",
    "digest": "<scrypt digest>",
    "attempts": 0,
    "expiresAt": "..."
  }
}
```

验证码十分钟失效，连续输错五次便删除待注册记录。更新尝试次数时使用 Blob 的 ETag 条件写入，避免并发请求互相覆盖。

登录与会话

验证成功后才创建账号。登录时重新计算 scrypt 并用 `timingSafeEqual` 比较摘要；不存在的账号也执行一次虚假的 scrypt，以减小通过响应时间枚举邮箱的差异。

会话令牌由 32 字节随机数生成。浏览器收到的 Cookie 类似：

```
freshmark_session=<opaque token>; Path=/; HttpOnly; Secure; SameSite=Lax
```

Functions 不保存令牌明文，只以令牌的 SHA-256 摘要作为 Blob 键。会话 30 天失效，退出登录时同时删除服务端记录并清除 Cookie。

`HttpOnly` 阻止前端 JavaScript 读取令牌，`Secure` 要求只通过 HTTPS 发送，`SameSite=Lax` 降低跨站请求携带 Cookie 的机会。所有修改状态的接口还检查 `Origin`，并分别设置平台级速率限制。

评论提交时，服务器从会话中取得昵称和邮箱，覆盖浏览器提交的同名字段。也就是说，即使有人在开发者工具里把请求改成：

```
{
  "name": "Another User",
  "email": "another@example.net"
}
```

最终保存的仍是当前登录账号的身份。**不要信任隐藏输入框，也不要把“前端不显示”误当成权限控制。**

异步不是“随便加一个 async”

评论、登录和验证码都不应阻塞文章首屏。页面先输出静态 HTML，浏览器完成首帧后才：

- 异步读取评论；
- 异步查询登录会话；
- 在需要登录或提交时动态加载对应 JavaScript chunk；
- 使用 `fetch` 与 Functions 交互。

后端变慢时，文章仍能正常阅读；失败只影响评论区域。密码的 scrypt 也使用 Node 的异步接口，避免在 Function 内用同步计算长时间占住事件循环。

这里的“异步”其实有两层：前端把非关键网络请求移出渲染路径，后端把昂贵的密钥派生工作交给异步执行。只做到其中一层，体验或吞吐量仍可能出现问题。

怎样验证整条链路

我通常按从底层到上层的顺序排查。

先检查 DNS：

```
dig +short A mail.example.org
dig +short TXT mailer.example.org
dig +short TXT mail._domainkey.example.org
dig +short TXT _dmarc.example.org
dig +short -x 203.0.113.10
```

再检查签名和本地服务：

```
opendkim-testkey -d example.org -s mail -vvv
systemctl is-active postfix opendkim nginx freshmark-mailer
postqueue -p
```

然后用无效数据验证公网边界。未授权调用应当像不存在一样返回 404；带正确令牌但邮箱格式错误时，应返回 400，而且不能产生邮件：

```
curl -X POST \
  -H "Authorization: Bearer $MAILER_TOKEN" \
  -H "Content-Type: application/json" \
  --data '{"to":"invalid","code":"123456","purpose":"registration"}' \
  https://mail.example.org/api/mail/comment-code
```

最后才进行真实注册，检查：

- 邮件是否到达；
- `Authentication-Results` 中 SPF、DKIM、DMARC 是否通过；
- DKIM 的 `d=` 和选择器是否正确；
- `Return-Path` 是否是预期的信封发件域；
- 邮件是否进入垃圾箱；
- 验证码过期与错误次数限制是否有效；
- Cookie 是否带有 `HttpOnly`、`Secure` 和 `SameSite`；
- 登出后旧会话是否失效；
- 修改评论请求中的昵称、邮箱是否会被服务器覆盖。

这套方案适合什么，不适合什么

它适合个人网站和低发送量项目：

- 不需要为每月少量验证码购买邮件套餐；
- 数据与发送链路都掌握在自己手里；
- 可以真正理解邮件认证和会话安全；
- 组件小，出问题时能逐层定位。

它不适合大规模业务。自建服务器要自己承担 IP 信誉、退信处理、投诉、限流、监控、密钥轮换和灾难恢复。发送量变大后，成熟的事务邮件服务往往更便宜——不是账单更低，而是运维成本更低。

对我而言，这次配置最有价值的并不是“省下一项订阅”，而是把几个经常被当成黑盒的系统连了起来：DNS 声明身份，DKIM 证明邮件未被篡改，Postfix 负责可靠排队，HTTPS 桥接缩小服务器接口，Functions 管理注册状态，Cookie 维持用户会话。每一层只承担一项清楚的职责，组合后便成了一套足够专业、又没有引入重型框架的评论账号系统。