# c13n #93

c13n

2026 年 8 月 30 日

# 第 I 部

# 对象存储上的持久流式传输

马浩琨

Aug 24, 2026

对象存储长期被视为「存文件」的地方，但随着日志、监控、视频以及物联网数据的爆发式增长，人们开始追问一个问题：是否能在保留对象存储低成本、无限扩容特性的同时，让它像消息队列一样提供低延迟的顺序读写？本文尝试回答这一问题，并给出可落地的技术路径。

## 1 对象存储与流式传输的语义差异

对象存储的核心操作是 PUT 与 GET，写操作以整个对象为单位，读操作同样面向完整对象或固定范围。这种模型天然支持最终一致性，但缺乏对「偏移量」和「提交点」的原生表达。流式传输则要求把连续字节流切分为可编号的分片，并支持在任意偏移处继续读写。把两者结合的思路在于：把对象内部继续切分为逻辑分片，再用独立的元数据对象把分片串成一条「流」。

## 2 三层模型的职责划分

应用层负责把业务数据包装成「流」的语义，向下提供 append、log、read 等接口。协议层把这些调用翻译成 S3 的 Multipart Upload 原语，同时维护一个轻量级索引，记录每个分片的 PartNumber、ETag 以及字节范围。存储层仍然是标准的对象存储桶，只需开启跨区域复制与生命周期策略即可。

## 3 分段写入的实现细节

最常见的做法是利用 Multipart Upload 的 PartNumber 充当逻辑偏移。假设我们把 PartSize 固定为 16 MiB，那么第 n 个分片对应的字节范围就是 `16 MiB * n` 到 `16 MiB * (n + 1)`。当生产者需要追加数据时，只需发起一个新的 Part，并把 PartNumber 设为当前最大值加一即可。需要注意的是，S3 对 PartSize 的限制是 5 MiB 到 5 GiB 之间，过小的 Part 会导致 Part 数量膨胀，过大的 Part 则会增加重传成本。因此实际部署时往往会根据实时带宽动态调整 PartSize。

## 4 流式索引的结构与版本控制

索引本身也是一个对象，通常命名为 `stream-name/manifest/latest.json`。文件内容采用 JSON Lines 格式，每一行记录一个 Part 的元数据：

```
{"part": 42, "etag": "\"abc123\"", "start": 688128000, "size":
    ↪ 16777216}
```

当索引体积超过阈值后，可以按天或按小时切分，并用一个「热索引」放在 Redis 中做缓存。写时复制策略要求每次更新索引时，先把旧索引拷贝一份，再在副本上追加新行，最后用原子 PUT 覆盖主索引。这样即使写入过程中发生崩溃，系统也能回退到最近一个完整版本。

## 5　一致性与持久化保证

一旦某个 Part 的 PUT 请求返回 200，对象存储就承诺该 Part 已在至少三个可用区持久化。此时生产者可以安全地向消费者 ACK。跨可用区复制策略（CRR）可以把同一个 Part 异步复制到另一个区域，实现真正的多活。断点续传则依赖 ListParts API：消费者启动时先列出所有未完成的分片，找到最大的 PartNumber，之后即可从下一个 Part 继续写入。

## 6　读取模式的匹配策略

顺序读取时，消费者先从索引里拿到 Part 列表，再依次发起 Range-GET 请求。为了降低延迟，可以在后台预取未来若干个 Part，并把它们缓存在本地内存。随机读取时，消费者先在索引里做二分查找，定位目标字节所在的 Part，再发起单次 Range 请求即可。零拷贝传输可以在服务端开启 SSE-Customer-Key 加密后仍然保持：对象存储直接把加密后的字节流返回给客户端，客户端在内核态完成解密，避免用户态内存拷贝。

## 7　元数据与生命周期管理

在对象上打标签 `topic=logs`、`partition=0`、`offset=10086`，可以让后续的垃圾回收脚本快速定位过期分片。生命周期策略可配置为：标准存储保留 7 天后转低频访问，再过 23 天转归档。归档后的 Part 不会被主动删除，而是由独立的 GC 进程在索引里做标记，等所有消费者确认消费完毕后再真正发出 Delete 请求。

## 8　性能调优的关键参数

PartSize 与并发度的关系可近似表示为：

$$\text{吞吐} = \min(\text{带宽}, \text{并发数} \times \text{PartSize} / \text{往返延迟})$$

实际测试表明，在 1 Gbps 带宽、平均 RTT 30 ms 的环境下，把 PartSize 设为 16 MiB、并发度设为 8，追加延迟的 P99 可稳定在 180 ms 左右。读写分离的另一个技巧是把索引和数据放在不同桶，避免 List 操作扫描海量数据对象。

## 9　真实场景的落地效果

某日志平台每秒产生 10 GB 文本，先在边缘节点做 Snappy 行式压缩，再以 16 MiB 为单位切分上传到 S3。索引存放在 ElastiCache 中，查询引擎 Athena 可以直接引用清单文件做即席分析，端到端延迟从原来的 5 分钟缩短到 30 秒。视频直播场景则把 HLS 切片直接映射为 Part，PartNumber 即为媒体序列号，播放器用 Range 请求即可实现任意倍速回放。

## 10 方案对比与选型建议

与 Kafka 相比，对象存储流式方案的追加延迟从 10 ms 上升到 50-200 ms，但容量可达 EB 级且成本降低一个数量级。MinIO 在局域网环境下可把延迟压到 20 ms，但仍然受限于单集群容量。HDFS 适合批处理，对小文件追加并不友好。因此在「先持久化、后回放」或「先存后算」的场景下，对象存储流式方案往往成为最优解。

## 11 未来演进与开放问题

S3 Express One Zone 把数据放在单可用区的高性能存储上，理论上可以将追加延迟再降一个数量级。Apache Arrow 把 Parquet 文件直接放在 S3 上，结合 Object Lambda 可以在服务端完成列裁剪和过滤，进一步降低分析延迟。另一个尚未标准化的方向是 Conditional Write：如果能让 PUT 在「当前 PartNumber 不存在」时才成功，就能用一条请求同时完成追加与冲突检测。

对象存储通过「分段 + 索引」即可在保留对象语义的同时实现持久流式传输。实施时只需四步：选定 PartSize 与索引介质、封装支持 append/log/read 的 SDK、配置生命周期与跨区复制、监控 P99 延迟与 GC 频率。如此即可把「大文件」变成「永不丢失的数据流」。

# 第 II 部

# Linux 内核的模块加载机制

马浩琨

Aug 25, 2026

Linux 内核之所以能在几十年间保持「小巧却无限扩展」的特性，核心原因之一就在于它提供了一套完整的动态代码装载机制，让开发者不必重新编译整个内核，就能把新的驱动、文件系统或协议栈以模块形式热插拔地加入运行中的系统。本文将沿着从用户空间命令到内核内部的完整路径，逐一剖析模块加载、符号解析、依赖管理与卸载的全过程。

## 12 什么是 LKM

LKM（Loadable Kernel Module）是编译后以 `.ko` 文件形式存在的一段可执行代码，它在加载时被内核动态链接进地址空间，卸载后则彻底释放占用的内存。与内建（built-in）到 `vmlinux` 的代码不同，LKM 在运行时可被按需载入或移除，从而让内核体积保持最小，同时具备按需扩展的能力。从 ELF 文件结构来看，`.ko` 本质上是一个可重定位的目标文件，包含代码段、数据段、符号表和重定位表；此外，它还带有内核专用的 `.modinfo` 段，用于存放模块作者、许可证、依赖关系等元信息。

## 13 用户空间工具链

当用户在命令行执行 `insmod` 或 `modprobe` 时，实际上是在调用 `finit_module` 或 `init_module` 系统调用。busybox 中的简易实现与 `kmod` 库的完整实现存在差异：前者仅做最基本的文件映射，后者则会解析 `modules.dep` 依赖数据库、自动加载前置模块，并支持模块签名验证。模块签名机制通过 `CONFIG_MODULE_SIG_FORCE` 选项强制要求所有模块必须由受信任的 X.509 证书签名，否则拒绝加载，从而在 Secure Boot 环境中保证内核完整性。

## 14 系统调用入口

进入内核后，`load_module()` 函数首先读取 ELF 文件头，验证魔数与架构兼容性，然后分配一个 `struct module` 对象，用于记录模块在内核中的全部状态。接下来，内核通过 `module_alloc()` 在专用虚拟地址区间分配可执行内存，并把各个段按权限映射到相应位置。此时模块状态被置为 `COMING`，表示正在初始化，尚未对外部可见。

## 15 核心装载步骤

布局阶段完成后，内核调用 `apply_relocate_add()` 对模块中的重定位表进行处理，把所有外部符号地址回填为内核真实地址。符号解析依赖于内核维护的两张表：`__ksymtab` 保存符号地址，`__kcrctab` 保存 CRC32 校验值，用于在 `CONFIG_MODVERSIONS` 开启时检测模块与内核版本是否匹配。开发者通过 `EXPORT_SYMBOL` 或 `EXPORT_SYMBOL_GPL` 宏导出的符号会被自动加入上述两张表，供其他模块引用。

模块参数的解析则由 `parse_args()` 完成：内核在模块加载时扫描 `__param` 段，把用户通过 `modprobe` 传递的 `key=value` 键值对写入对应变量。参数解析成功后，`do_init_module()` 调用模块通过 `module_init()` 宏注册的初始化回调；若回调返回非零值，内核会自动回滚已分配的资源，并将模块状态回退到 `COMING` 之前的阶段，避免半初始化状态残留。

## 16 依赖与引用计数

模块之间的符号依赖形成了有向图，内核用 drivers/base/module.c 中的引用计数机制来维护拓扑关系。当模块 A 导出符号被模块 B 使用时，B 在加载时会调用 try_module_get() 增加 A 的引用计数；卸载时则通过 module_put() 递减。只有当引用计数为零，且没有设备文件或文件系统仍持有该模块时，delete_module() 系统调用才能真正执行卸载流程。

## 17 卸载流程

rmmod 命令最终映射到 sys_delete_module 系统调用。内核首先检查引用计数与设备占用情况，确认安全后调用 free_module()。该函数依次执行模块通过 module_exit() 注册的清理回调，释放所有 section 占用的内存，最后调用 vfree() 归还虚拟地址空间。整个过程结束后，模块状态被置为 GOING，随后从内核模块链表中移除。

## 18 安全与调试

为防止恶意模块加载，内核提供了 lockdown 模式与 SELinux/AppArmor 的细粒度控制。lockdown 模式下，即使拥有 root 权限也无法加载未签名的模块；SELinux 则通过 module_request 钩子对特定域发起的模块加载请求进行审计与拦截。调试时，dmesg 中「Loading module」与「Freeing module」日志、/proc/kallsyms 中的符号表以及 /sys/module/ 下的模块目录，都是定位问题的第一手资料。crash 工具可通过 mod 命令列出当前已加载模块，并用 lx-symbols 自动加载符号文件，极大简化了现场分析。

## 19 进阶主题

Livepatch 技术允许在不重启系统的情况下，用新函数替换内核既有函数，其实现同样建立在模块加载框架之上。eBPF 则提供了更轻量的内核扩展方式，通过字节码即时编译执行，避免了传统 LKM 的编译与签名流程。Rust for Linux 项目正在把类型安全的 Rust 代码编译为 .ko 文件，并通过稳定的 FFI 边界导出符号，为内核模块开发引入内存安全的新范式。

## 20 实践示例

编写最小的 hello 模块只需几行代码：

```
#include <linux/init.h>
#include <linux/module.h>

static int __init hello_init(void)
{
   pr_info("Hello, kernel!\n");
```

```
    return 0;
}

static void __exit hello_exit(void)
{
    pr_info("Goodbye, kernel!\n");
}

module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
```

这段代码中，module_init 与 module_exit 宏分别把初始化与清理函数注册到内核的回调链表；MODULE_LICENSE 宏则把许可证字符串写入 .modinfo 段，供 modinfo 命令读取。

若要为模块添加参数，可使用：

```
static int debug = 0;
module_param(debug, int, 0644);
MODULE_PARM_DESC(debug, "Enable debug messages");
```

module_param 宏在编译时生成参数描述符，加载时由内核解析并赋值给 debug 变量。

从用户空间的 modprobe 到内核的 load_module()，再到符号重定位、初始化回调与引用计数管理，Linux 模块加载机制形成了一套严密且可扩展的动态链接体系。理解这一体系，不仅能帮助开发者编写可靠的内核模块，也为探索 livepatch、eBPF 等前沿技术奠定了基础。

# 第 III 部

# HTTP Accept 头驱动的内容协商机制

黄梓淳

Aug 26, 2026

在 Web 世界里，同一个资源往往需要以多种形式呈现给不同客户端。浏览器可能希望得到 HTML，移动应用可能期待 JSON，爬虫则可能只关心纯文本。这种「同一 URI、多种表示」的需求，催生了 HTTP 内容协商机制。内容协商分为主动协商与被动协商两种形式。主动协商由服务端根据请求头自动选择最合适的表示，被动协商则让客户端在多个可选响应中二次选择。本文聚焦主动协商中扮演核心角色的 Accept 系列请求头，剖析其语法、实现策略与工程实践，并探讨性能、安全与未来演进。

## 21 HTTP 内容协商基础

RFC 7231 明确了主动协商的流程：服务端在收到请求后，先解析 Accept、Accept-Language、Accept-Encoding 等头字段，再结合自身能力与资源可用形式，选出最优表示并返回 200 响应，同时在响应头中携带 Content-Type、Content-Language、Content-Encoding 以告知客户端实际使用的表示。Vary 响应头则用于告知缓存系统哪些请求头会影响响应结果，从而避免错误命中。

Accept 头声明客户端可接受的 MIME 类型及其优先级；Accept-Language 声明自然语言偏好；Accept-Encoding 则用于表达对 gzip、Brotli 等压缩方式的支持。Accept-Charset 已在现代实践中被废弃，因为 UTF-8 已成事实标准。服务端返回的 Content-* 系列头与请求中的 Accept 系列头形成镜像，共同构成完整的协商闭环。

## 22 Accept 头的完整解析

Accept 头的语法遵循「媒体范围」列表，每个范围可附加 q 参数表示质量因子。典型示例 Accept: text/html, application/xhtml+xml, application/xml;q=0.9, image/webp, /;q=0.8 表达了客户端对 HTML、XHTML、XML、WebP 图片以及任意类型的偏好顺序。质量因子取值范围为 0 到 1，默认 1。RFC 7231 第 5.3.2 节规定：当多个媒体范围匹配同一资源表示时，q 值最高者胜出；若 q 值相同，则按出现顺序决定。

星号通配符 / 表示接受任意类型，image/* 表示接受任意图片子类型。参数如 profile= 或 charset= 可进一步约束模式，例如 Accept: application/ld+json;profile=https://schema.org/ 用于请求符合 Schema.org 的 JSON-LD。浏览器通常发送包含多种现代格式的 Accept 列表，而 API 客户端往往只声明 application/json；爬虫则可能只发送 / 以获取最通用的文本形式。

## 23 服务端实现策略

服务端解析 Accept 头一般遵循「拆分—排序—过滤—打分—选择」五步。算法先将逗号分隔的媒体范围拆成数组，按 q 值降序排列，再逐一与服务端支持的 MIME 类型做匹配。匹配成功后累加 q 值与特异度得分，最终得分最高者即为协商结果。边界情况包括：请求不携带 Accept 头时，默认使用服务端首选类型；q=0 表示显式拒绝；多个类型得分相同时可按服务端偏好顺序或响应大小决定。

以 Node.js 的 negotiator 库为例，代码 negotiator = require('negotiator'); available = ['text/html','application/json']; best = negotiator.mediaType(available) 完成了从请求头到最优类型的映射。Go 语言可用 github.com/julien-

schmidt/httprouter 结合 mime 包实现类似逻辑。Spring 框架则通过 ContentNegotiationManager 统一管理，在 WebMvcConfigurer 中重写 configureContentNegotiation 方法即可声明扩展名、参数或 Accept 头的优先级。

与路由框架集成时，Express 可将协商逻辑封装为中间件，放在路由之前执行；Gin 在 Go 中可通过 c.Negotiate 系列方法直接返回 HTML 或 JSON；Spring MVC 则依靠 @ResponseBody 与 HttpMessageConverter 自动选择序列化器。无论哪种实现，缓存系统都需要看到 Vary: Accept，否则可能把 JSON 响应错误地提供给请求 HTML 的客户端。

## 24 实战案例

多语言文档站通常同时依赖 Accept-Language 与 Accept 头。服务端可先按 Accept-Language 确定语言，再按 Accept 选择 HTML 或 JSON 格式的 API 响应。URL 设计上，子路径方案 /zh-cn/docs 直观但需维护多套路由；内容协商方案 URI 唯一但缓存碎片化严重；Cookie 方案则在无状态场景下难以被 CDN 理解。

同一资源同时提供 HTML 与 JSON 的 HATEOAS API，可通过 Accept: application/json;profile=https://example.com/hateoas 与 Accept: text/html 区分不同表示。profile 参数可映射到不同 JSON Schema，从而在不改变媒体类型的前提下实现版本演进。

渐进式图片场景中，服务端可检查 Accept 是否包含 image/webp 或 image/avif，若命中则由 Cloudflare 或对象存储的边缘函数实时转换并返回对应格式。GraphQL 端点通常只声明 application/json，但仍需处理 Accept: application/graphql-response+json 等实验性类型。

## 25 性能与可维护性

Accept 解析本身开销极低，但 Vary: Accept 会把不同 Accept 值的请求分散到不同缓存条目，导致公共资源缓存命中率下降。缓解方法包括：标准化客户端 Accept 列表、忽略无影响的参数、或在反向代理层对 Accept 做归一化后再转发。

Nginx 可通过 map 指令将 Accept 映射为简短标记，再结合 try_files 实现轻量协商；Cloudflare Workers 则能在边缘执行完整协商逻辑，减少源站压力；Varnish 的 VCL 语言同样支持基于 Accept 的分支缓存。无论哪种方案，均需记录协商决策日志以便 A/B 测试不同 q 值策略对用户体验的影响。

## 26 安全与隐私

Accept 头可与其他指纹信息组合，提升跨站追踪精度。研究表明，Accept + User-Agent + Accept-Language 三元组在未登录用户中具备较高唯一性。服务端应避免将敏感信息写入 Content-Type 参数，如将内部版本号或用户 ID 编码其中。拒绝服务方面，构造数千个媒体范围的 Accept 头可放大解析开销，需在网关层设置字段长度上限或使用流式解析器。

## 27 演进与替代方案

传统 Accept 头无法表达「只返回部分字段」或「字段级压缩」等细粒度需求。RFC 7240 引入 Prefer 与 Preference-Applied 头，可声明 return=minimal 或 respond-async 等偏好。链接头中的 profile 与 describes 参数则提供了更语义化的资源描述方式。
HTTP/3 与 QUIC 对内容协商的影响主要体现在 0-RTT 与连接复用层面，但协商算法本身保持不变。RFC 8942 定义的 Variants 与 Variant-Key 头让服务端在响应中预先声明资源变体，客户端可据此直接选择，避免二次协商。

## 28 最佳实践清单

始终返回 Vary: Accept，以确保缓存正确区分不同表示。提供 406 Not Acceptable 兜底响应，并在响应体中列出服务端支持的类型列表。Accept 解析逻辑必须幂等，避免产生副作用或触发写操作。公开文档中应明确声明支持的 MIME 类型及默认 q 值策略。最后，结合 ETag 或 Last-Modified 实现条件请求，可在客户端缓存有效时跳过协商，降低延迟。
内容协商是「一个 URI，多种表示」的基石，在可维护性、安全与性能之间持续权衡。随着语义 Web 与机器可读描述的演进，未来或将出现 Accept-Schema 等新头字段，让客户端能够声明对数据结构的精确期望。理解 Accept 机制的细节，有助于构建更优雅、更高效的 Web 服务。

# 第 IV 部

# HTTP 客户端库的演进与迁移实践

杨其臻

Aug 29, 2

随着微服务架构与云原生环境的普及，分布式系统对 HTTP 交互提出了更高要求。遗留代码库中仍广泛使用的 JDK `HttpURLConnection`、Apache HttpClient 3/4 以及 OkHttp 2.x 等组件，因其阻塞式 API、缺失连接池以及有限的 HTTP/2 支持，已成为系统可维护性、性能与安全性的瓶颈。本文面向后端、平台与 DevOps 工程师及架构师，系统梳理主流 HTTP 客户端库的技术演进脉络，并结合真实迁移案例，提出可落地的选型决策模型与工程化实践。

## 29 HTTP 客户端库的技术演进路线

自 JDK 1.1 引入的 `HttpURLConnection` 起，Java 生态的 HTTP 客户端便沿着「阻塞→异步、多路复用→协议扩展」的轨迹持续演进。该类库仅提供基础的请求/响应抽象，内部既无连接池，也未实现 HTTP/2 协议，导致高并发场景下频繁创建与销毁 TCP 连接，严重制约吞吐。JDK 11 正式引入的 `java.net.http` 包则原生支持异步编程模型与 HTTP/2 多路复用，开发者可通过 `HttpClient.newBuilder().version(HttpClient.Version.HTTP_2)` 启用新协议栈，同时通过 `CompletableFuture` 实现非阻塞回调，显著降低线程上下文切换开销。

Apache HttpClient 的演进同样反映了这种趋势。3.x 版本采用完全阻塞的 I/O 模型，配置分散在大量 `HttpParams` 子类中；4.x 引入了面向连接池的 `PoolingHttpClientConnectionManager`，并在 4.5+ 版本中完善了 Keep-Alive 策略与空闲连接清理；5.x 则通过模块化设计同时提供「经典」与「异步」双栈，底层基于 `Reactor` 实现非阻塞 I/O。开发者可通过 `CloseableHttpAsyncClient` 发起请求，并在 `FutureCallback` 中处理响应，避免了同步等待带来的线程阻塞。

Square 公司推出的 OkHttp 则在移动端与服务端双线演进。2.x 版本引入了强大的「拦截器链」机制，允许开发者在 `Request` 发出前后插入日志、加解密或鉴权逻辑；3.x 开始默认启用 HTTP/2，并通过 `ConnectionPool` 实现多路复用；4.x 全面拥抱 Kotlin，通过扩展函数与 DSL 构建请求，显著提升了代码可读性。核心方法 `RealCall.getResponseWithInterceptorChain()` 将请求依次经过 `AddHeadersInterceptor`、`ConnectInterceptor` 等环节，最终完成网络 I/O。

在响应式编程浪潮下，Jetty Reactive HttpClient、Reactor-Netty 与 Vert.x Web Client 等组件进一步把异步推向极致。它们均基于事件循环与零拷贝缓冲区，避免了用户态-内核态的数据拷贝，适合对延迟与吞吐要求严苛的场景。与此同时，Service Mesh（如 Envoy、Linkerd）将 mTLS 握手、负载均衡与可观测性下沉到 Sidecar，应用层 SDK 趋向轻量化，仅保留协议编解码与序列化职责。

纵观上述演进，可见四大共性：异步化 I/O、HTTP/2 多路复用、协议扩展能力（如 WebSocket、Server-Sent Events）以及零信任网络下的 mTLS 与可观测性集成。

## 30 选型决策模型

面对多套候选方案，团队需建立可量化的决策模型。首先在功能矩阵中对比同步/异步、HTTP 版本、连接池实现、指标暴露与协议扩展能力；其次通过压测工具（如 JMeter、wrk）获取 QPS、P99 延迟与资源占用数据；再次评估社区活跃度、Spring 集成友好度与长期维护承诺；最后进行 License、历史漏洞与 JDK 兼容性风险扫描。决策树示例显示：

遗留 Spring Boot 2 应用若目标升级至 Spring Boot 3，可优先考虑 java.net.http 以减少依赖；移动端应用继续使用 OkHttp 以复用成熟的证书固定与缓存机制；边缘计算节点则倾向于 JDK 11+ 自带客户端以降低镜像体积。

# 31 迁移实践：从 Apache HttpClient 4.5 至 java.net.http

## 31.1 前期准备与方案对比

迁移前需对代码库进行全面盘点，识别自定义 SSL 上下文、Cookie 策略、代理配置、超时与重试逻辑。依赖梳理需确认 Spring 5+ 与 Servlet 容器兼容性。方案 A 将直接替换为 JDK 11+ 原生客户端，优点是零额外依赖、享受长期支持；方案 B 仅升级到 Apache HttpClient 5.x，改动最小但仍需维护第三方依赖；方案 C 通过抽象 HttpClient SPI 层实现多实现可插拔，为未来技术栈演进预留空间。以下以方案 A 为例，详述迁移步骤。

## 31.2 Maven/Gradle 依赖清理与配置映射

首先从 pom.xml 中移除 httpclient 与 httpcore 依赖，JDK 11+ 环境已内置 java.net.http 模块，无需额外声明。原有 SSL 配置代码形如：

```
SSLContext sslContext = SSLContexts.custom()
      .loadTrustMaterial(trustStore, new TrustSelfSignedStrategy())
      .build();
SSLConnectionSocketFactory sslsf = new SSLConnectionSocketFactory(
    ↪ sslContext);
```

在 java.net.http 中，对应逻辑迁移至 SSLParameters 与 HttpClient.Builder：

```
SSLContext sslContext = SSLContext.getInstance("TLSv1.3");
sslContext.init(null, trustManagers, new SecureRandom());
HttpClient client = HttpClient.newBuilder()
      .sslContext(sslContext)
      .version(HttpClient.Version.HTTP_2)
      .build();
```

上述代码首先通过 SSLContext.getInstance(TLSv1.3) 获取支持 TLS 1.3 的上下文，随后调用 init 注入 TrustManager 数组；HttpClient.Builder 的 sslContext 方法将该上下文应用到所有请求，而 version 方法显式声明 HTTP/2 协议偏好。
原有 RequestConfig 中的超时设置：

```
RequestConfig config = RequestConfig.custom()
      .setConnectTimeout(5000)
      .setSocketTimeout(10000)
      .build();
```

在 JDK 新客户端中，超时由 HttpRequest.Builder 的 timeout 方法控制：

```
HttpRequest request = HttpRequest.newBuilder()
      .uri(URI.create("https://api.example.com/users"))
      .timeout(Duration.ofMillis(10000))
      .build();
```

该方法接收一个 Duration 对象，内部转换为 java.net.http 的超时语义；若请求在指定时间内未完成，将抛出 HttpTimeoutException。

## 31.3 异步回调与异常处理改造

Apache HttpClient 的异步用法依赖 FutureCallback，示例：

```
httpClient.execute(request, new FutureCallback<HttpResponse>() {
   @Override
   public void completed(HttpResponse result) { /* 处理响应 */ }
   @Override
   public void failed(Exception ex) { /* 处理异常 */ }
   @Override
   public void cancelled() { /* 处理取消 */ }
});
```

迁移后使用 CompletableFuture：

```
CompletableFuture<HttpResponse<String>> future =
      client.sendAsync(request, HttpResponse.BodyHandlers.ofString
          ↪ ());
future.whenComplete((response, throwable) -> {
   if (throwable != null) {
      if (throwable instanceof HttpTimeoutException) {
         // 超时处理逻辑
      } else if (throwable instanceof ConnectException) {
         // 连接失败处理逻辑
      }
   } else {
      // 正常响应处理
   }
});
```

sendAsync 返回的 CompletableFuture 可链式调用 whenComplete，在 throwable 非空时通过 instanceof 区分 HttpTimeoutException 与 ConnectException，实现与原回调等价的异常分类处理。

## 31.4 重试与熔断集成

为避免重复造轮子，可借助 Resilience4j 包装 CompletableFuture：

```
Retry retry = Retry.ofDefaults("http-client");
Function<HttpRequest, CompletableFuture<HttpResponse<String>>>
    ↪ decorated =
      Retry.decorateCompletionStage(retry, () -> client.sendAsync(
          ↪ request, HttpResponse.BodyHandlers.ofString()));
```

`Retry.decorateCompletionStage` 会在 `CompletableFuture` 失败时按配置策略自动重试，内部通过指数退避降低对下游的冲击。

### 31.5 单元与集成测试

使用 WireMock 录制真实流量并回放，验证新老客户端行为一致性；混沌测试则通过 Chaos Monkey 注入延迟与断连，观察重试与熔断是否生效。

### 31.6 灰度发布与回滚

通过 Feature Toggle 让新旧客户端并行运行，金丝雀发布阶段监控 QPS、错误率与 P99 延迟；若指标异常，可在 1-2 个版本周期内快速回滚至保留的 HttpClient 4.5 依赖。

## 32 迁移实践：从 OkHttp 2.x 至 4.x

OkHttp 4.x 对 Kotlin 友好，但引入若干破坏性变更。原 2.x 拦截器接口 `Interceptor.Chain.proceed(Request)` 在 4.x 中签名不变，但 `CertificatePinner` 的 DSL 由字符串模式改为中缀函数：

```
val pinner = CertificatePinner.Builder()
      .add("example.com", "sha256/
          ↪ AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=")
      .build()
```

Kotlin 扩展函数可进一步简化请求构建：

```
val request = Request.Builder()
      .url("https://api.example.com/users")
      .header("Accept", "application/json")
      .build()
val response = client.newCall(request).execute()
response.use {
   println(it.body?.string())
}
```

`response.use` 确保 `ResponseBody` 及时关闭，避免连接池泄漏。R8/ProGuard 规则需更新以保留新包路径，HTTP/2 多路复用可将移动端 RTT 降低约 30%。

## 33 通用工程化 checklist

为保障长期演进能力，团队应建立接口抽象层，将 `HttpClient`、`HttpRequest` 与 `HttpResponse` 定义为 SPI，具体实现通过依赖注入切换；配置外置到 YAML 或配置中心，实现超时、重试、代理等参数的动态刷新；可观测性方面，使用 Micrometer 采集 QPS 与延迟，并通过 OpenTelemetry 注入分布式追踪 Span；安全层面强制启用 TLS 1.3、证书热更新与 HSTS 头；定期进行混沌与压测，建立性能基线；最后输出迁移指南、示例仓库与 FAQ，形成组织级知识库。

## 34 踩坑实录与避坑指南

连接池泄漏往往源于未关闭 `ResponseBody`，导致 `FIN_WAIT2` 堆积；SNI 缺失在 JDK 8u161 以下默认不发送，导致证书验证失败；代理认证若同时配置 Digest 与 Basic，可能陷入循环挑战；Keep-Alive 空闲时间与负载均衡器不一致会引发 502；指标缺失则需手动封装 Micrometer `Timer` 与 `Counter`。

## 35 未来展望

HTTP/3（QUIC）已在 OkHttp 5、Jetty 12 与 netty-incubator 中获得实验性支持，零 RTT 握手将进一步降低移动端延迟。WebAssembly 与 Sidecar 环境下的 WASI-HTTP 规范，让浏览器与边缘节点可用统一接口发起请求。零信任网络中，SPIFFE/SPIRE 可自动注入 mTLS 证书，SDK 无需感知证书生命周期。AI 辅助方面，基于链路追踪数据训练超时与重试策略，可在异常流量下动态调整参数。

## 36 结论

从阻塞式 `HttpURLConnection` 到原生异步的 `java.net.http`，再到即将普及的 HTTP/3，演进主线清晰可见：协议从单路进化到多路复用与 QUIC，编程模型从同步转向异步与响应式。迁移绝非一刀切，而应评估现状、抽象接口、灰度验证。唯有将 HTTP 客户端视为平台级能力持续演进，方能在云原生与零信任时代保持竞争力。

# 第 V 部

# 连续扩散语言模型（Continuous Diffusion Language Models）

黄京

Aug 30, 2026

## 37 从图像到文本：扩散模型的跨模态尝试

扩散模型在图像生成领域取得的成功已无需赘述，其核心思想是通过逐步添加噪声将数据分布映射到标准高斯，再通过学习逆过程实现高质量样本生成。然而，将这一范式迁移到文本领域面临根本性挑战：文本由离散符号构成，而扩散过程天然依赖连续空间。这促使研究者探索将离散 token 映射到连续嵌入空间的方案，形成了连续扩散语言模型这一新兴方向。该类模型试图在保留扩散模型全局建模优势的同时，解决文本生成的离散性约束问题。

## 38 扩散模型的数学框架

在连续扩散语言模型中，前向加噪过程首先将 one-hot 形式的 token 转换为稠密嵌入向量。给定原始嵌入 $x_0$，扩散过程定义为 $q(x_t|x_0) = \mathcal{N}(\sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)I)$，其中 $\bar{\alpha}_t$ 控制噪声强度随时间步递增。这一设计使得在任意时刻 $t$，我们都能从原始数据直接采样得到加噪版本，无需迭代累积。

反向去噪过程则试图学习参数化的条件分布 $p_\theta(x_{t-1}|x_t) \approx \mathcal{N}(\mu_\theta(x_t,t), \Sigma_\theta(x_t,t))$。实践中通常采用简化目标：直接预测添加的噪声 $\varepsilon$ 或直接预测原始数据 $x_0$。前者对应于最小化 $\mathbb{E}_{t,x_0,\varepsilon}\|\varepsilon - \varepsilon_\theta(x_t,t)\|^2$，后者则优化重建损失。两种目标在不同噪声调度下表现出互补特性，噪声预测在高噪声区域更稳定，而 $x_0$ 预测在低噪声区域重建更精确。

## 39 模型架构与实现细节

连续扩散语言模型的嵌入层将离散词表映射为可学习的连续向量空间。不同于 VQ-VAE 的硬量化策略，CDLM 保持嵌入的连续性，允许梯度在嵌入空间自由流动。嵌入维度通常设置为 768 – 1024，与 Transformer 主干的隐藏维度保持一致。部分工作探索了冻结预训练词嵌入的可能性，但实验表明可学习嵌入在下游任务上通常获得 2 – 3 个点的 PPL 提升。

主干网络采用双向 Transformer encoder，允许每个位置在所有时间步都能看到完整上下文。时间步信息通过 Sinusoidal 编码或可学习嵌入注入，常用 AdaLN（Adaptive Layer Normalization）机制实现：对每个 Transformer 层计算 $\gamma(t)$ 和 $\beta(t)$，对隐藏状态进行 scale-shift 调制。代码实现中，时间步嵌入通常先经过两层 MLP 映射到与隐藏维度相同的空间，再与层归一化参数融合。

```
class TimeEmbedding(nn.Module):
   def __init__(self, dim):
      super().__init__()
      self.dim = dim
      # Sinusoidal 位置编码
      self.proj = nn.Sequential(
         nn.Linear(dim, dim * 4),
         nn.SiLU(),
         nn.Linear(dim * 4, dim)
      )
```

```
    def forward(self, t):
        # t: [batch_size]
        half = self.dim // 2
        freqs = torch.exp(-torch.arange(half, device=t.device) *
                          math.log(10000) / (half - 1))
        args = t[:, None].float() * freqs[None]
        emb = torch.cat([torch.cos(args), torch.sin(args)], dim=-1)
        return self.proj(emb)
```

上述代码实现了时间步的 Sinusoidal 编码与 MLP 投影。输入时间步 $t$ 首先扩展为正弦余弦特征，再通过两层非线性变换得到与模型隐藏维度匹配的向量。该向量随后在 AdaLN 中用于调节 Transformer 层的归一化参数，实现条件生成能力。

## 40 训练策略与工程优化

时间步采样策略对模型收敛至关重要。均匀采样在 $[1, T]$ 区间内随机选择 $t$，但由于不同 $t$ 对应的去噪难度差异巨大，带权采样通常更有效。常见做法是按照噪声调度函数 $\beta_t$ 的密度进行重要性采样，使模型在高噪声和低噪声区域都能获得充分训练。实践中，权重函数常设计为 $w(t) \propto \beta_t / \alpha_t$，平衡重建误差与梯度方差。

序列长度外推是长文本生成的关键瓶颈。标准绝对位置编码难以泛化到训练长度之外，相对位置编码（如 ALiBi）或外推式编码（如位置插值）成为必要选择。ALiBi 通过在注意力分数上添加线性偏置，允许模型在推理时处理更长序列而无需重新训练。实验表明，采用 ALiBi 的 CDLM 模型可以将有效上下文长度从 512 扩展至 2048，同时保持困惑度基本不变。

## 41 推理加速与采样策略

标准 DDPM 采样需要数百至上千步迭代，远慢于自回归模型的一次前向。DDIM 通过将扩散过程重新参数化为非马尔可夫链，允许用数十步甚至数步完成高质量采样。其核心公式为 $x_{t-1} = \sqrt{\alpha_{t-1}} \hat{x}_0 + \sqrt{1 - \alpha_{t-1} - \sigma_t^2} \varepsilon_\theta(x_t, t)$，其中 $\sigma_t$ 控制随机性大小。实践中，设置 $\sigma_t = 0$ 可获得确定性采样，进一步加速推理。

动态 early-stopping 根据当前去噪质量自适应调整步数。当预测的 $x_0$ 与前一步的重建差异低于阈值时，可提前终止扩散过程。该策略在低噪声区域特别有效，因为此时模型已接近最终输出，继续迭代收益递减。结合键值缓存机制，推理吞吐量可提升 3 – 5 倍。

## 42 条件控制机制

Classifier-Free Guidance（CFG）是 CDLM 中最常用的条件控制方法。训练时随机以一定概率丢弃条件信息，强制模型同时学习条件与无条件分布。推理时通过插值公式 $\tilde{\varepsilon}_\theta = \varepsilon_\theta(x_t, t, \emptyset) + s \cdot (\varepsilon_\theta(x_t, t, c) - \varepsilon_\theta(x_t, t, \emptyset))$ 实现可调强度控制，其中 $s$ 为 guidance scale。较大的 $s$ 增强条件遵循度但降低多样性，较小的 $s$ 则相反。

跨模态条件通过预训练编码器将外部信号映射到与文本嵌入相同的空间。CLIP 图像编码器输出可直接作为扩散模型的条件向量，实现图生文任务。结构化条件（如表格、JSON）则需要专门的编码器，将键值对序列化为连续表示，再与时间步嵌入融合。实验表明，条件注

入位置（早期 vs. 晚期层）对生成质量影响显著，早期注入通常获得更强的条件遵循能力。

## 43 长文本建模挑战

标准 Transformer 的平方复杂度限制了 CDLM 处理长文本的能力。层次化扩散通过先在句子级别进行扩散，再在词元级别细化，有效降低了计算复杂度。第一阶段在粗粒度序列上运行，第二阶段以第一阶段输出为条件进行细粒度去噪。状态空间模型（如 Mamba、RetNet）与扩散的结合是另一前沿方向，这些模型提供线性复杂度且保持长程依赖建模能力，适合作为扩散主干的替代架构。

## 44 理论分析与开放问题

尽管扩散模型在图像 FID 上表现优异，但在文本 PPL 上仍落后于自回归模型。这一差距源于信息瓶颈：连续嵌入空间需要通过有限维度表示离散符号的概率分布，导致表示坍缩。理论分析表明，最优噪声调度应与数据流形曲率相关，但实践中常用线性或余弦调度仅为近似。收敛速度分析显示，扩散模型的 ELBO 与对数似然之间存在间隙，该间隙随序列长度增加而扩大。

与能量模型、流模型的统一视角揭示了扩散模型的本质：它们都是通过学习从噪声到数据的映射来隐式建模数据分布。能量模型直接定义概率密度，流模型学习可逆变换，而扩散模型则通过马尔可夫链逐步转换。三者在数学上可相互转化，提供了从不同角度理解生成过程的可能。

## 45 实际应用与生态现状

开源实现中，Diffusion-LM 和 CD-LM 提供了 PyTorch 参考代码，HuggingFace Diffusers 计划集成扩散语言模型支持。学术预训练权重覆盖 110M 至 1B 参数规模，社区微调脚本支持在特定领域数据上继续训练。快速上手示例显示，通过十余行代码即可加载预训练模型并生成句子，CFG scale 参数可直接调节输出多样性。

短期落地场景包括交互式写作助手（利用并行生成与局部编辑能力）、代码自动补全（带语法约束的生成）以及广告文案生成（多属性条件控制）。长期愿景是打破自回归模型在文本生成的主导地位，实现与多模态生成范式的统一。

## 46 结论

连续扩散语言模型通过将离散文本映射到连续空间，成功将扩散模型的并行生成、纠错能力和灵活条件控制引入 NLP 领域。尽管在似然估计上仍落后于自回归模型，其在可控生成和编辑任务上的独特优势已展现出广阔应用前景。随着长文本建模和推理加速技术的成熟，CDLM 有望成为下一代文本生成架构的重要候选方案。