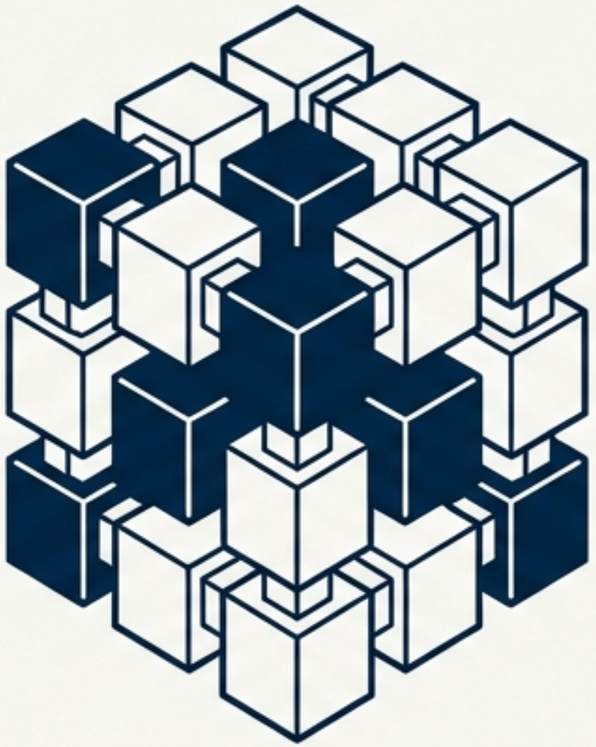


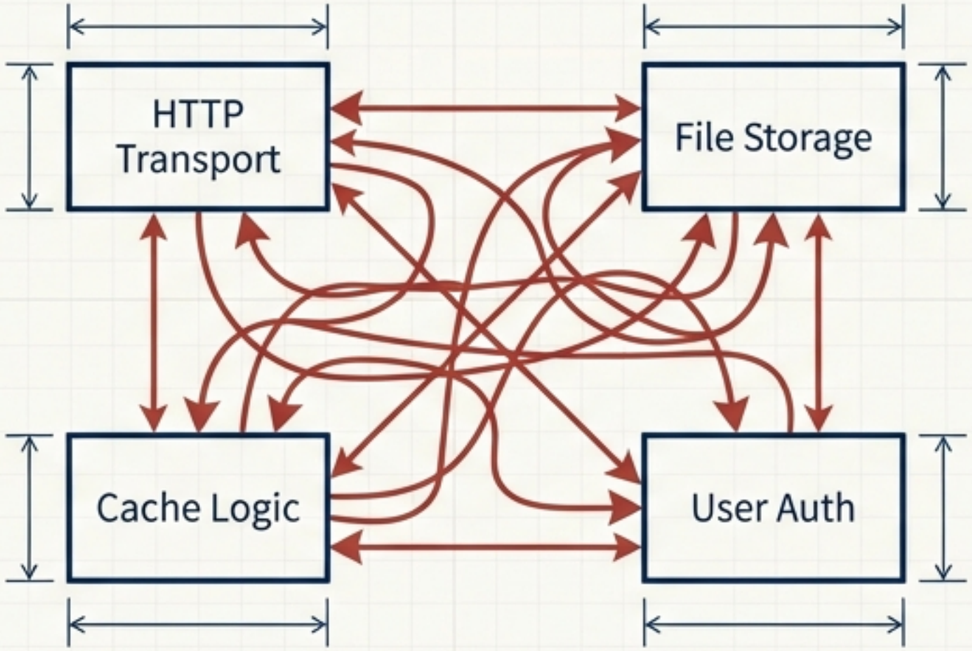
Packice：一个解耦与模块化的P2P缓存系统

深入解析一个优雅系统架构的设计哲学与实践

NotebookLM

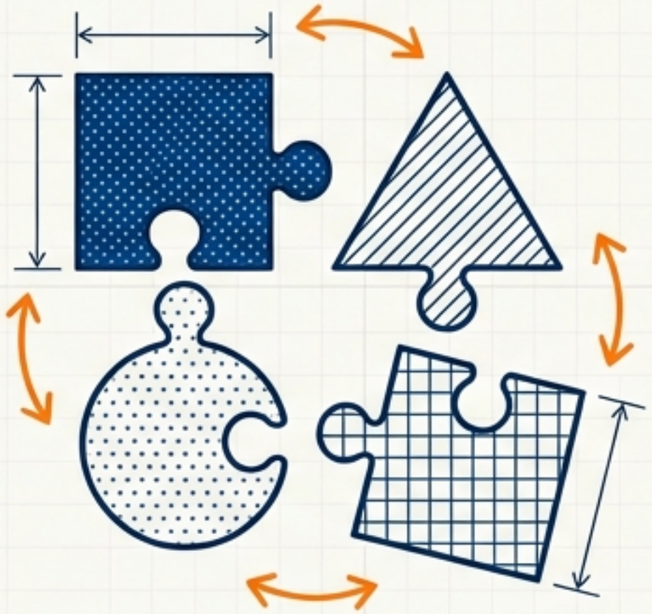
现有缓存架构的普遍裂痕：耦合与低效

紧密耦合的架构



现有的缓存系统往往耦合严重，传输协议、存储逻辑、缓存逻辑没有很好解耦。基础设施的变更往往导致大量代码重构。

缺乏统一抽象

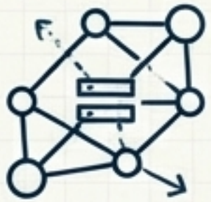


由于缺乏统一的抽象，代码容易出错，难以快速响应业务变化，显著降低了开发效率。

NotebookLM

我们的蓝图：基于三个正交维度的彻底解耦

我们将系统拆分为三个完全独立、可以任意组合的维度，从根本上解决耦合问题。



怎么交互（Transport）

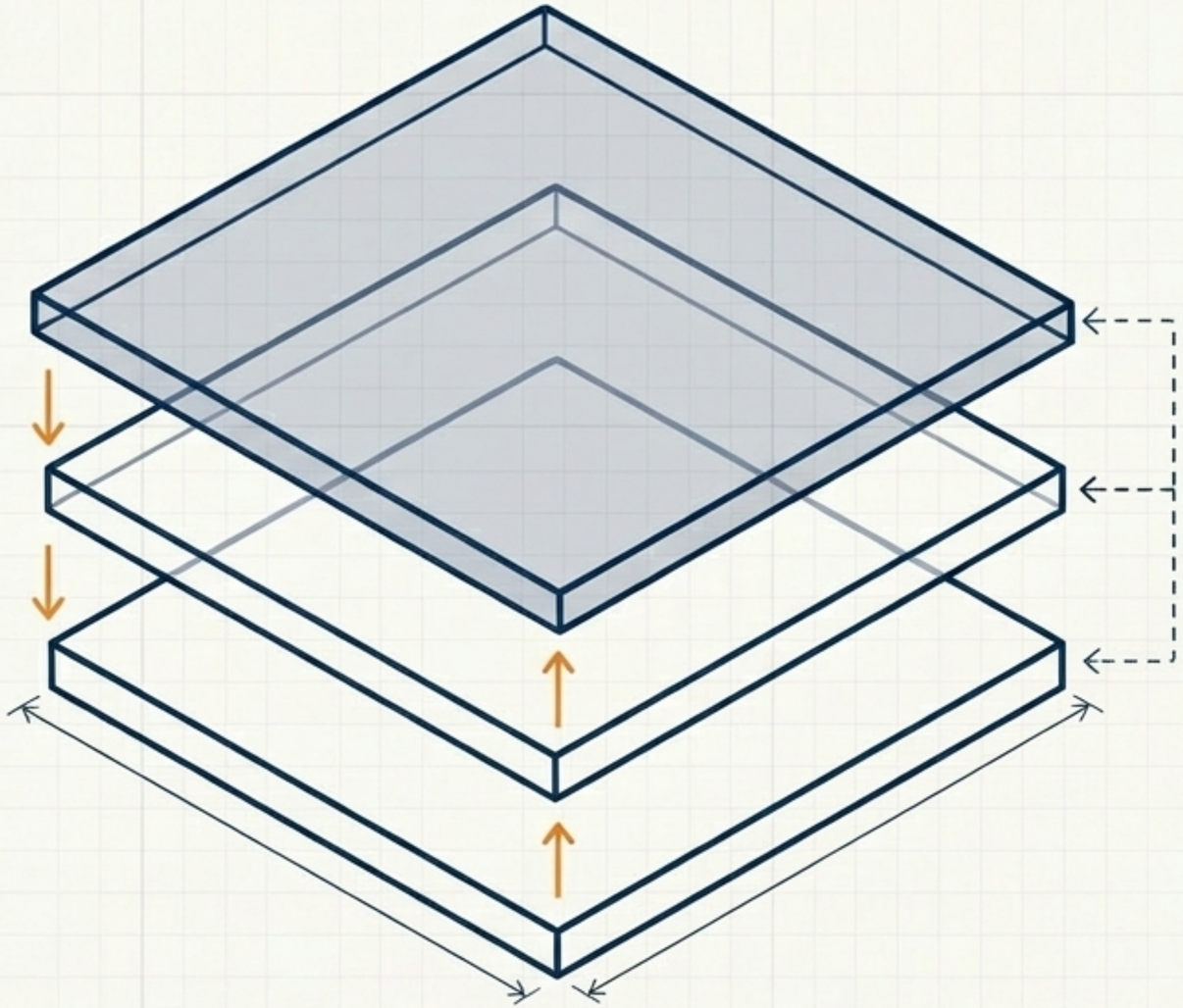
定义客户端与系统间的通信方式。HTTP、UDS、进程内调用...

怎么管（Peer）

负责权限控制、生命周期管理和元数据维护。

怎么存（Backend）

定义数据的物理存储。内存、本地文件、共享文件系统、S3...



NotebookLM

Packice 的核心构件：四大关键抽象

理解这四个位于 `packice/core/′ 的核心抽象，是掌握整个系统的关键。

Peer（管理者）

角色：系统的大脑。

职责：管理 Object 的生命周期，分发 ‘许可证’，但不直接处理数据 I/O。

Lease（许可证）

角色：权限凭证。

职责：拥有 Lease 意味着拥有对 Object 的操作权（读/写/创建）。

Object（对象）

角色：数据的逻辑单元。

职责：包含元数据（Meta）和对物理数据块（Blob）的引用。

Blob（数据块）

角色：数据的物理载体。

职责：实际存储数据，具有多态性（内存，文件等）。

NotebookLM

控制平面：Peer（大脑）与 Lease（许可证）的协同

Peer 的职责

- 管理 Object 生命周期：处理 `acquire´（申请），`seal´（封存），`discard´（丢弃）。
- 分发权限：根据请求授予 Lease，决定谁可以在任何时操作哪个 Object。

关键点：Peer 不直接处理数据 I/O，它只负责元数据和权限管理。

Lease 的关键区分

Lease TTL（租约过期时间）

目的：防止客户端因崩溃或失联而导致死锁，保障系统活性。这是操作权的有效期。

Object TTL（数据过期时间）

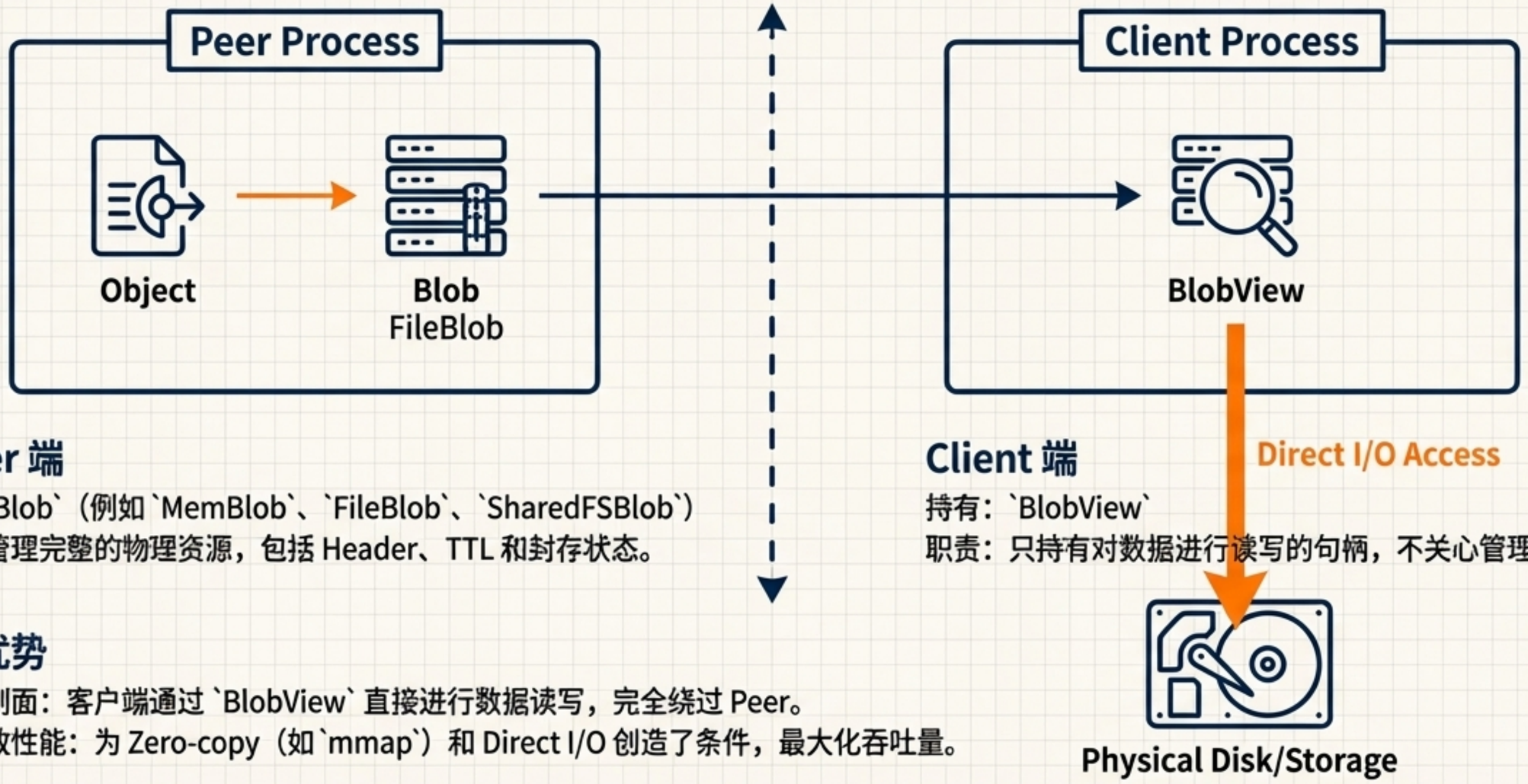
目的：管理业务数据的生命周期，决定数据何时被回收。这是数据本身的有效期。

结论：解耦操作锁与数据生命周期是保障系统鲁棒性的关键设计。

NotebookLM

数据平面：Object（逻辑）与 Blob（物理）的分离

将数据的逻辑表示（Object）与其物理载体（Blob）分开，并通过视图（View）机制实现高性能 I/O。



Server 端

持有：`Blob`（例如 `MemBlob`、`FileBlob`、`SharedFSBlob`）

职责：管理完整的物理资源，包括 Header、TTL 和封存状态。

Client 端

持有：`BlobView`

职责：只持有对数据进行读写的句柄，不关心管理细节。

性能优势

绕过控制面：客户端通过 `BlobView` 直接进行数据读写，完全绕过 Peer。

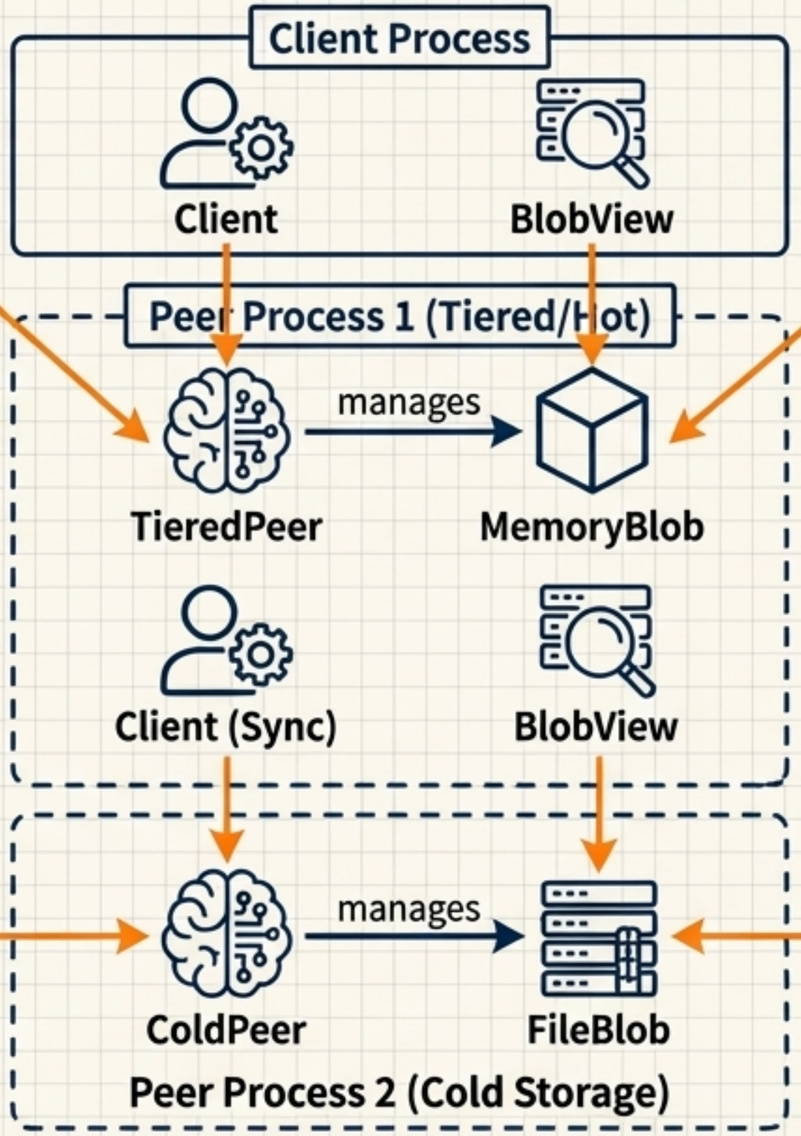
实现极致性能：为 Zero-copy（如`mmap`）和 Direct I/O 创造了条件，最大化吞吐量。

NotebookLM

整体架构蓝图：组件如何协同工作

1. 客户端请求
客户端通过 `Transport` 与 `Peer` (控制平面) 通信，申请 `Lease`。

2. 直接数据访问
客户端获取 `BlobView`，直接与 `Blob` (数据平面) 进行 I/O，实现高性能读写。

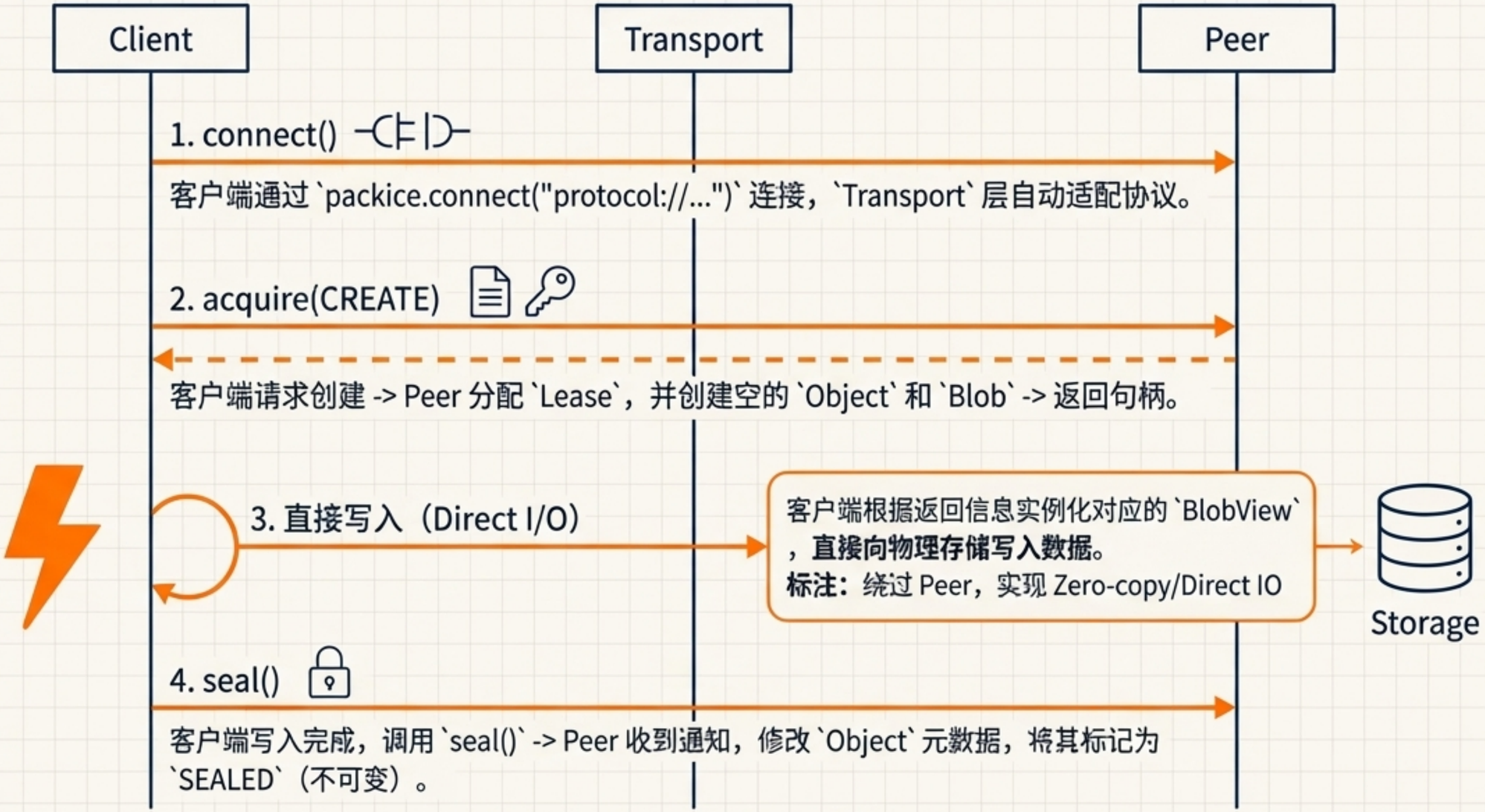


3. Peer 间协作
一个 `Peer` (如 TieredPeer) 可以作为客户端，与另一个 `Peer` 交互，实现数据驱逐、备份等高级功能。

4. 物理存储多态
图中清晰展示不同 Peer 管理不同类型的 Blob (Peer 1 管理 `MemoryBlob`，Peer 2 管理 `FileBlob`)。

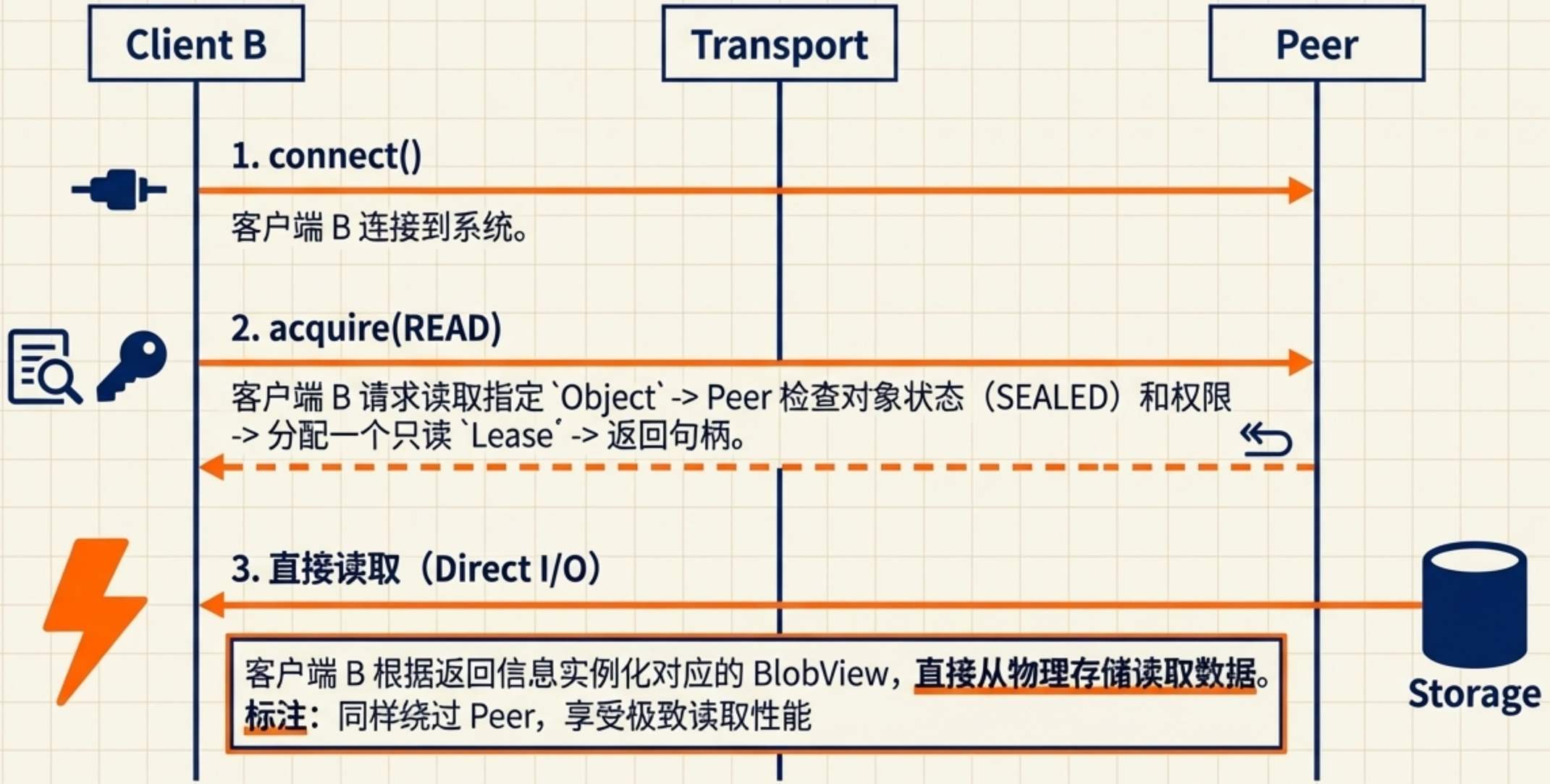
NotebookLM

核心流程（一）：写入一个新对象（Create & Seal）



NotebookLM

核心流程（二）：读取一个已存在的对象（Acquire Read）



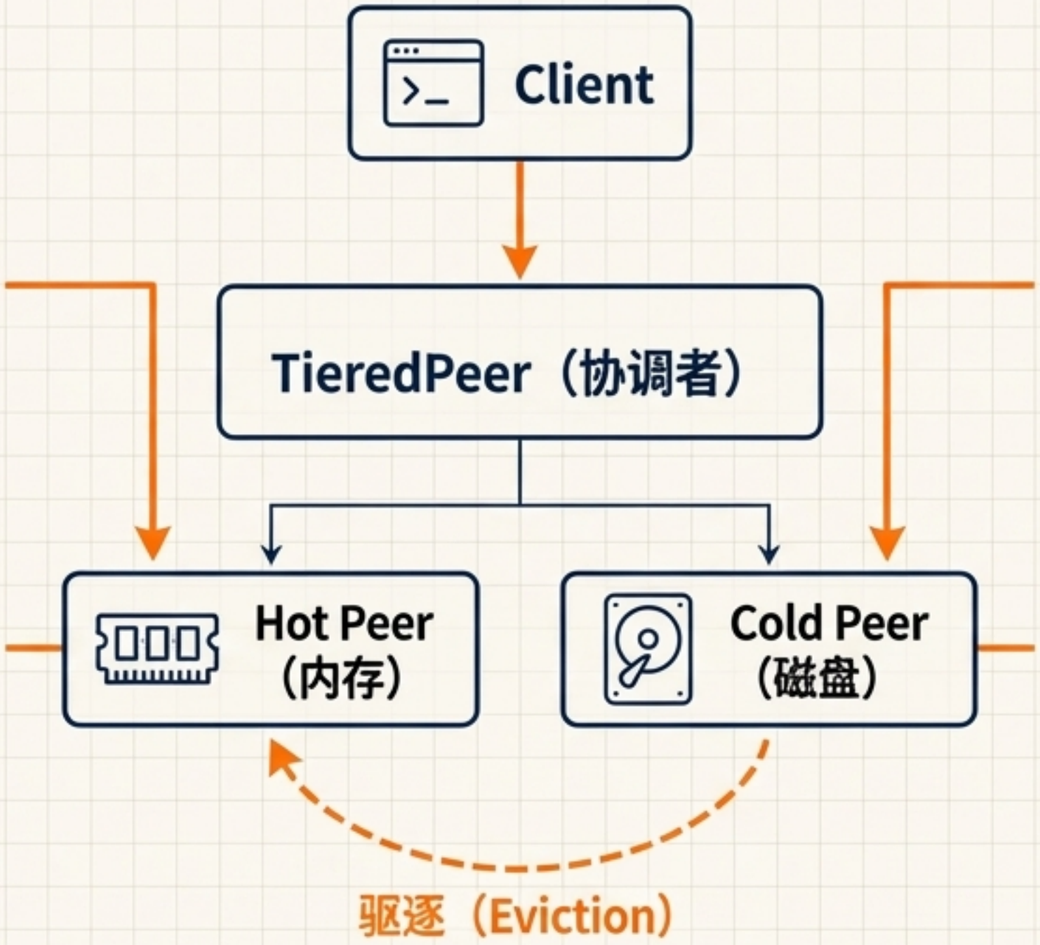
NotebookLM

架构的魔力：用组合实现复杂的分级存储

Packice 的架构以 `Peer´ 为核心交互单元。我们可以将多个基础 Peer 封装为一个复合 `TieredPeer´，作为协调者实现复杂的存储策略。

1. 写入（Write）：Client 请求 -> `TieredPeer´ 优先写入 Hot Peer（内存），实现极速写入。

2. 驱逐（Eviction）：内存满 -> `TieredPeer´ 触发 LRU 策略 -> 将数据从 Hot Peer 搬运至 Cold Peer（磁盘） -> 更新元数据指向。



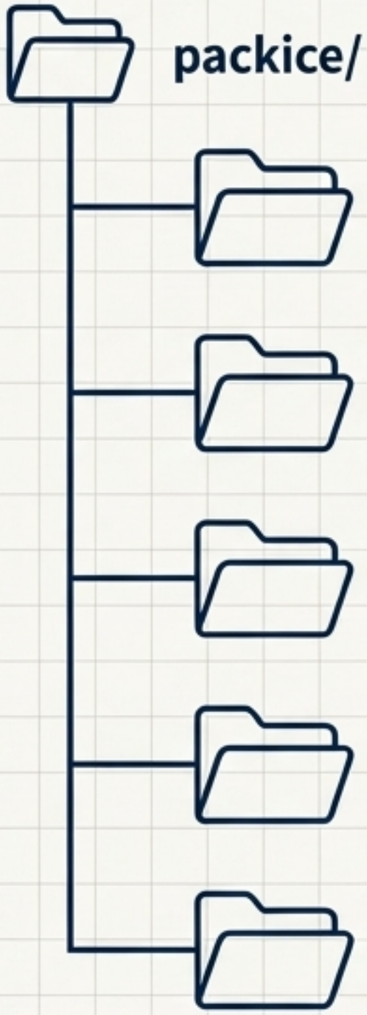
3. 读取命中热数据（Read Hot Hit）：Client 请求 -> 数据在 Hot Peer -> 直接从内存返回。

4. 读取命中冷数据（Read Cold Hit）：Client 请求 -> 数据在 Cold Peer -> 从磁盘返回。

可选优化：`TieredPeer´ 可异步将数据预热回 Hot Peer。

NotebookLM

代码地图：'如果我要贡献代码，应该看哪里？'



packice/

core/
核心接口 (Peer, Blob, Lease) - 系统的契约和骨架。

backends/
存储实现 (Memory, FS, SharedFS) - `Blob` 和 `Lease` 的具体实现。

peers/
逻辑实现 (MemoryPeer, SharedFSPeer, TieredPeer) - `Peer` 的大脑逻辑。

transport/
协议适配 (HTTP, UDS, Direct) - 与外界对话的方式。

interface/
用户接口 (Client, CLI) - 面向最终用户的封装。

NotebookLM

你的扩展蓝图：三个典型的自定义场景

场景 A

我想支持把数据存到 S3

挑战
添加一个新的后端存储。

位置
`packice/backends/`

行动
→ 继承 `Blob` 基类，实现 `S3Blob`，封装 S3 的读写逻辑。

场景 B

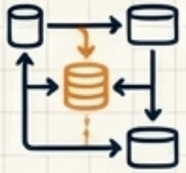
我想用 gRPC 替换 HTTP

挑战
添加一个新的通信协议。

位置
`packice/transport/`

行动
→ 实现 `GrpcServer` 和 `GrpcTransport`，将 gRPC 请求翻译成对现有 `Peer` 接口的调用。

场景 C



我想实现一个新的缓存策略

挑战
自定义数据调度逻辑。

位置
`packice/peers/`

行动
→ 像 `TieredPeer` 一样，组合一个或多个现有 Peer，在新的 Peer 类中实现你的调度逻辑。

NotebookLM

优雅且 Pythonic：简洁强大的开发者体验

```
# 1. 极简的连接方式（自动识别协议）
client = packice.connect(peer)

# 2. Pythonic 的资源管理（Context Manager）
# 自动处理关闭和异常
with client.create(meta={"ttl": 60}) as obj:
    obj.write(b"Hello Packice")

# 3. 显式封存，确保数据一致性
obj.seal()

# 4. 像操作本地对象一样读取
with client.get(obj.id) as obj:
    print(obj.buffer.tobytes())
```

1 极简的连接方式（自动识别协议）

自动识别并连接到对应的 Peer 节点，无需复杂的配置，简化了网络通信的初始化过程。

2 Pythonic 的资源管理（Context Manager）

利用 Python 的上下文管理器（`with` 语句）自动处理资源生命周期。在退出代码块时，系统会自动关闭连接、释放资源并处理潜在的异常，确保代码的健壮性。

3 显式封存，确保数据一致性

通过显式调用 `seal()` 方法，将数据标记为不可变，确保数据在分布式环境中的一致性和完整性，防止意外修改。

4 像操作本地对象一样读取

获取远程数据就像操作本地对象一样直观。通过上下文管理器，系统自动处理数据的检索和缓存，开发者只需关注数据内容本身，无需关心底层细节。

Python实现胶水业务代码非常合适... 此外，'life is short, I use Python"

NotebookLM

前进的道路：现状与未来规划

当前状态

- **核心**：核心抽象（Peer，Lease，Object，Blob）已初步跑通。
- **后端**：已支持 Memory（开发调试）和 SharedFS（POC状态）。
- **客户端**：Python SDK 完成，缺少集成层逻辑。

发展蓝图

性能优化
引入更高效的 object pool 和 object transfer 能力，避免 Python 层的直接复制。

P2P 生态
对接基于 Redis 的 tracker，实现 object 发现等。

生产级鲁棒性
完善 Lease 异常处理；实现生产可用的 Tiered Peer 及生命周期管理。

可观测性
提供生产可用的 Metrics 及相关监控。

NotebookLM

Packice 架构设计的核心权衡

Q：为什么区分 Lease TTL 和 Object TTL?

A：将**操作锁（Lease）**的生命周期与**数据（Object）**的生命周期解耦。前者防止客户 端死锁，保证系统活性；后者服务于业务数据管理。

Q：为什么客户端 I/O 要绕过 Peer?

A：为了极致性能。Peer 专职**控制面（Control Plane）**，负责协调与管理；数据读写应在**数据面（Data Plane）**尽可能直接，以实现 Zero-copy 和 Direct I/O。

Q：SharedFS 和普通 FS 后端有何不同?

A：SharedFS 为多进程/多节点共享同一文件系统而设计，它必须额外处理跨进程的 Header 同步、文件锁管理以及基于文件 mtime 的垃圾回收（GC）。

谢谢

github.com/user/packice

NotebookLM