Notes on System Design

LaTeX by Junfan Zhu

2024 年 11 月 7 日

目录

1 百万量级用户扩展 **2**

2 封底估算 Back-of-the-Envelope Estimation：分析系统性能 **3**

3 面试 45min **3**

4 限流器 **3**

5 一致性哈希 **4**

6 键值存储（非关系型数据库） **4**

6.1 分布式哈希 5

6.2 版本控制 5

6.3 系统架构 6

7 唯一 ID 生成器 **6**

8 URL 缩短器 **7**

9 网络爬虫 **8**

10 通知 **9**

11 发帖刷动态 **9**

12 聊天 **10**

13 搜索自动补全 Autocomplete **11**

14 YouTube 视频 **11**

15 云盘 **13**

16 支付宝收付款 **14**

17 指标监控告警 **16**

1 百万量级用户扩展

让网络层无状态，每一层都要冗余，尽量多缓存数据，多个数据中心，CDN 承载静态资源，用分片扩展数据层，不同架构层分成不同服务，监控系统，使用自动化工具。

设计：

用户从 DNS 获取负载均衡器 IP 地址，通过 IP 地址连接负载均衡器，HTTP 请求被转发到服务器 1 或 2 上，Web 服务器从库中读用户数据，Web 服务器把所有增删改操作请求转发到主库。

1. 非关系型数据库 NoSQL (CouchDB, Neo4j, Cassandra, HBase, DynamoDB)：键值存储、图存储、列存储、文档存储。适用于低延时、非结构化数据、只需序列化 (JSON, XML, YAML) 和反序列化数据、海量数据。

2. 负载均衡器。把流量均匀分配到各 Web 服务器，服务器间用私有 IP 地址通信。

3. 数据库复制。Master-Slave 主从关系数据库复制。

4. 缓存：读得多而改得不多。

   缓存可以提高系统性能，减轻数据库工作负载。单缓存服务器可能导致单点故障 Single Point of Failure (SPOF)，它坏了系统就不能工作，所以要部署多个缓存服务器。一旦缓存满了，任何新请求可能导致已有条目被删除，这是缓存驱逐，包括：LRU (Least Recently Used), LFU (Least Frequently Used), FIFO.

5. 内容分发网络 Content Delivery Network (CDN)。

   可以缓存静态内容：用户请求文件，如果文件不在 CDN，就去 Web 服务器拉取，在 CDN 保存下来，返回给用户 A，接下来用户 B 请求时直接返回。

   加入 CDN、缓存后的设计：静态资源不再由 Web 服务提供，而是从 CDN 获取，提供响应速度，并用缓存减轻数据库负载。

6. 横向扩展：无状态网络层 = 把会话数据持久性存储在数据库中。

   无状态架构比有状态架构更简单、可扩展。用户 HTTP 请求可发给任意 Web 服务器，Web 服务器从共享的数据存储中拉取数据，状态数据存储在数据库（NoSQL 易扩展）而非 Web 服务器中。自动扩展：基于网络流量自动增减 Web 服务器。

7. 多个数据中心。流量重定向，如 geoDNS 根据地理位置引流到附近数据中心。数据同步，不同地区用户使用不同的本地数据库/缓存，在多个数据中心异步复制数据。测试部署保证一致性。

8. 消息队列，利用生产者消费者的缓冲区来分配异步请求。

9. 数据库扩展。

   (a) 纵向（向上）：增加算力。导致更大单点故障风险，烧钱。

   (b) 横向（分片）：增加服务器。选择分片键 Sharding/Partition Key，决定把数据均分分到各个 Shard，名人可能要单独分区。一旦数据库通过分片，就很难跨分片连接 join，因此要去规范化 denormalization，把数据冗余存储到多个表。

2 封底估算 Back-of-the-Envelope Estimation：分析系统性能

1. 内存块，硬盘慢。避免在硬盘找数据。
2. 压缩算法快。尽量把数据压缩后再用网络传输。
3. 高可用性：系统长时间运转能力（SLA 服务水平协议）

计算题：推特 3 亿月活，50% 用户每天用，每天平均发两条推文，10% 推文包含 1M 多媒体数据，数据存储 5 年。

1. 每日活跃用户 DAU = 300,000,000 * 50% = 150,000,000
2. 推文每秒查询量 QPS = 150,000,000 * 2 / 24h / 3600s = 3500
3. 峰值 QPS = 2 * 推文 QPS = 7000
4. 多媒体数据存量 = 150,000,000 * 2 * 10% * 1MB = 30 TB/天
5. 5 年多媒体数据量 = 30 TB * 365 * 5 = 55PB

3 面试 45min

1. 理解问题，确定设计边界，3-10min。
2. 提议高层级的设计并获得认同，10-15min。
3. 试探面试官想法，继续深入，提出多个方案，10-25min。
4. 总结时复述设计方案是有用的，唤起面试官的记忆有助于面试结果，3-5min。

4 限流器

设计：低延时、内存少、分布式限流器，限制过量请求，能处理异常，容错性高。

流量限制规则存储在硬盘，工作进程 Worker 从硬盘获取规则并存到缓存。当客户端向服务器发送请求时，请求被发给限流器中间件。限流器中间件从 Redis 缓存中加载规则，获得计数器和上一次请求的时间戳。如果没限流，限流器中间件转发给 API 服务器；如果限流，限流器返回 429 响应码报错，并把请求丢弃或者转发到队列。

1. 云微服务架构，限流器在 API 网关（流量限制中间件）实现身份验证、IP 地址白名单等。
2. 流量限制算法
   (a) 代币桶 Token Bucket. 两个参数：桶大小、每秒放进桶里代币数量。易实现，内存使用率高，可以应对限时促销等流量激增场景。
   (b) 漏桶 Leaking Bucket. FIFO 队列的代币桶。两个参数：桶大小、出栈速度（每秒处理多少请求）。由于请求按固定速度处理，适合出栈速度稳定的场景。突发流量导致大量积压。
   (c) 固定窗口计数 Fixed Window Counter. 如果在时间窗口边界出现大流量爆发，则通过的请求也激增。

(d) 滑动窗口日志 Sliding Window Log. 缓存保存时间戳，流量限制准确，任何滑动窗口都不超过阈值，但内存消耗大，因为即使请求被拒绝，它的时间戳仍然保存在内存。

(e) 滑动窗口计数 Sliding Window Counter. 平滑了流量波动，因为时间窗口内请求速率是基于前一个时间窗口内请求的平均速率计算出来的。

3. 高层级架构。硬盘访问太慢，用 Redis 存储计数器，因为内存上的缓存速度快、支持时间过期策略。

4. 流量限制规则。用户请求过多时，限流器向客户端返回 HTTP 响应码 429、需要等待时间。

5. 分布式系统扩展

(a) 高并发环境产生了竞争条件 Race Condition，若用锁则拖慢系统，可以用 Lua 脚本和 Redis 有序集合数据结构。同步问题。

(b) 限流器之间的同步问题，可以用 Redis 中心化数据存储。

6. 性能优化。多数据中心、最终一致性模型来同步数据。

7. 讨论

(a) 硬流量限制：请求数量不超过阈值。软流量限制：请求数量可以短时间内超过阈值。

(b) 流量限制在 OSI7 层模型中的其他层。

(c) 设计客户端避免被限流：用客户端缓存防止频繁调用 API、别发太多请求、添加代码捕获异常并恢复、重试逻辑中添加退避时间。

5 一致性哈希

1. 服务器宕机时，大部分缓存客户端会连到错误服务器，导致缓存未命中 Cache Miss，所以用一致性哈希，平均只要重新映射键数/槽 Slot 个键，而不用把所有键全部映射。

2. 一致性哈希步骤：用均匀分布哈希函数将服务器和键映射到哈希环，要找某个键的映射位置，就从键的位置顺时针找，直到找到哈希环上第一个服务器。

3. 虚拟节点：实际节点在哈希环上的映射。每个服务器都可以有多个虚拟节点，数量越多，键的分布越均匀，因为标准差/数据分散程度减少。

4. 优点

(a) 加减服务器时，需要重新分配的键最少。

(b) 可以横向扩展，因为数据分布均匀。

(c) 减轻名人热点键问题导致的某分区过载，因为数据分布均匀。

6 键值存储（非关系型数据库）

设计：键值对小，大数据存储，高可用性，高扩展性（大数据），自动伸缩（基于流量加减服务器），可调节的一致性，低延时的键值存储系统。

1. 存储大数据：用一致性哈希把负载分散到各服务器。

2. 高可用读：数据复制、数据中心。
3. 高可用写：版本控制、用向量时钟解决冲突
4. 数据集分区：一致性哈希
5. 增量可扩展性：一致性哈希
6. 异质性：一致性哈希
7. 可调一致性：仲裁一致性
8. 处理暂时故障：松散仲裁、暗示性传递
9. 处理永久故障：Merkle 树
10. 处理数据中心故障：跨数据中心复制

6.1 分布式哈希

CAP 三选二

1. 一致性 Consistency：所有客户端在同一时间看到同样数据。
   (a) 仲裁一致性 Quorum Consensus, 保证读写操作一致性。N= 副本数, W= 写操作 Quorum 大小，获得 W 个副本确认。R= 读操作 Quorum 大小，获得 R 个副本响应，W+R>N 保证强一致性。
   (b) 最终一致性：足够长时间后，所有数据更新都传播开来，所有副本变得一致，如 Dynamo, Cassandra.
2. 可用性 Availability：即便有节点故障，任意客户端发出的请求都能被响应。
3. 分区容错性 Partition Tolerance：分区 = 两个节点通信中断。分区容错性 = 尽管网络被分区，系统仍可继续运行。

用一致性哈希来数据分区

1. 自动伸缩：可以基于负载自动加减服务器
2. 异质性：可以给性能高的服务器分配更多虚拟节点。

6.2 版本控制

每次修改数据都生成一个新的不可变数据版本。

1. 向量时钟：[服务器，版本] 对，可以检查版本先后、是否冲突。
2. Gossip 协议：去中心化故障检测。
   (a) 每个节点维护一个节点成员列表：ID、心跳。
   (b) 每个节点增加心跳，并定期给随机节点发送心跳数，这些节点又将心跳信号传递给另一组节点。

(c) 节点收到心跳，就更新成员列表，如果心跳在预定时间内没增加，该成员就被认为是宕机了。

3. 松散仲裁 Sloppy Quorum，提高可用性。

(a) 在哈希环上最先发现的 W 个正常服务器进行写操作，在哈希环上最先发现的 R 个正常服务器进行读操作，故障服务器被忽略。

(b) 暗示性传递 Hinted Handoff，处理临时故障。如果服务器故障，另一个服务器来临时接班，等服务器恢复时，变更被推送回来实现数据一致性。

(c) 反熵协议 Anti-entropy Protocol，如果副本永久不可用，也能保持副本同步。比较副本上每条数据，并将每个副本都更新到最新版本。Merkle 树（哈希树）能检测不一致性，并最小化数据传输量。

i. Merkle 树（哈希树）可验证大数据结构的内容。对每个非叶节点，它的标记是基于子节点的值进行哈希运算。若根节点哈希值若相同，则匹配；若不同，则先左后右比较子节点哈希值，找出哪些桶不同步，并同步这些桶。

ii. 把键空间分成不同的桶，桶用作根节点维护树的深度。然后把桶里每个键都用一致性哈希 Uniform Hashing 计算哈希值。为每个桶创建哈希节点。向上构建树，直到根节点，通过计算子节点的哈希值得到父节点的哈希值。

6.3 系统架构

1. 客户端与键值存储系统之间用 API 通信：get(key), put(key, value).

2. 协调者是一个节点，在客户端、键值存储之间充当代理。

3. 节点通过一致性哈希分布在哈希环上。

4. 系统去中心化，可以自动加减节点。

5. 数据被复制到多个节点。没有单点故障。

6. 读写路径：基于 Cassandra 架构。

(a) 写请求在提交日志 Commit Log 中被持久化，数据保存在内存缓存。当内存缓存已满，数据被刷新到硬盘上的 SSTable。

(b) 读请求导向节点后，系统检查数据是否在内存缓存，若不在，就用布隆过滤器 Bloom Filter 找硬盘里哪个 SSTable 有这个键。

7 唯一 ID 生成器

1. 多主复制 Multi-master Replication，把数据库下一个 ID+= 服务器数量，但分布式环境下 ID 不连续，且难与多个数据中心扩展。

2. 通用唯一标识符 Universally Unique Identifier (UUID)。生成 ID 简单，易扩展。ID 不随时间增加，长 128 位。

3. 工作服务器 Ticket Server，ID 是数字，但存在单点故障。

4. 推特雪花 Snowflake，把 ID 分解为符号位、时间戳、数据中心 ID、机器 ID、序列号。

讨论

1. 时钟同步。服务器在多核上时，用网络时间协议 Network Time Protocol (NTP) 解决。
2. 调整 ID 各部分长度：减少序列号长度，增加时间戳长度。
3. 高可用性。

8 URL 缩短器

封底估算：每天生成 1 亿 URL，每秒写操作 1160，读操作是写操作 10 倍，读操作 11,600. URL 缩短器工作 10 年，就要支持 3650 亿条记录。URL 平均长度 100 字符，10 年的存储量是 36.5TB.

两个 API 端点，有利于客户端服务器通信。

1. 缩短 URL。客户端发 POST 请求（包含长 URL），返回短 URL。实现：哈希表存储长短 URL 键值对。62 个字符，$62^n < 3650$ 亿，哈希值长度 n = 7. 根据 Char 7 的唯一 ID 生成器把 URL 生成 ID，再用以下两种方法把 ID 转换成短 URL。

哈希 + 解决冲突 Hash Collision	Base62 转换 Base Conversion
CRC32、MD5、SHA-1。用布隆过滤器（高效利用空间概率性）提升性能，检测元素是否属于集合。	把十进制数转化为 Base62 表示
短 URL 长度固定	短 URL 长度不固定，随 ID 变化
不用唯一 ID 生成器	需要唯一 ID 生成器
可能有哈希冲突需要解决	没冲突，因为 ID 唯一
不知道下一个可用短 URL 是什么，因为不依赖 ID	真的下一个短 URL 可用是什么，因为新 ID += 1，可能是安全隐患

2. URL 重定向。发送 GET 请求，返回长 URL。
   (a) 301 重定向：请求的 URL 永久移动到长 URL，浏览器缓存响应，以后对同一个 URL 的请求就不再发给缩短器。因为读操作多于写操作，<shortURL, longURL> 映射存储在缓存中可以降低服务器负载，提高性能。
   (b) 302 重定向：对同一个 URL 后续请求还会发给 URL 缩短器，再倍重定向到长 URL 服务器。可以更轻松跟踪点击率、点击来源，适合数据分析。
3. 其他
   (a) 限流器：过滤拦截恶意海量 URL 请求等安全问题。
   (b) Web 服务器伸缩：因为网络无状态，所以可以加减 Web 服务器。
   (c) 数据库扩展：数据库复制、分片。

9 网络爬虫

给定一组 URL，下载这些 URL 对应的所有网页，从这些网页中提取 URL，将新 URL 添加到需要下载的 URL 列表，重复以上。

1. 把种子 URL 添加到 URL 前线。
2. HTML 下载器从 URL 前线获取 URL 列表。
3. HTML 下载器从 DNS 解析器中获取 URL 对应的 IP 地址并下载。
4. 内容解析器解析 HTML 页面，检查是否有问题。
5. 内容解析器将解析和验证后的内容传给“已见过内容?”组件。
6. “已见过内容?”组件检查 HTML 页面是否存在数据库中。若在，则同样内容的 URL 已被处理过，则丢弃 HTML。若不再，则把页面传递给链接提取器。
7. 链接提取器从 HTML 页面中提取链接。
8. 提取的链接被传递给 URL 过滤器进行筛选。
9. 经过筛选的链接被传递给“已见过 URL?”组件。
10. “已见过 URL?”组件检查 URL 是否在数据库中，若是，则已被处理过，不用再处理。若没处理过，则添加到 URL 前线中。

选择好的种子 URL 就有一个好的起点，爬虫可以遍历尽可能多的链接。把整个 URL 空间按不同国家分块，或按照话题区分。用 FIFO 队列存储即将下载的 URL，要下载网页就要用 DNS 解析器将 URL 转化为 IP 地址。内容解析器比较两个 HTML 文件的哈希值，看是否重复冗余。用布隆过滤器和哈希表判断 URL 是否已见过。

1. DFS 太深了，不好，用 BFS+FIFO 队列。

   注意忙于爬同一个主机会使服务器被淹没，这不礼貌甚至可能被认为是 DoS 攻击。而且考虑 URL 优先级，用队列优先处理网络流量、网络排名高的网页。

2. URL 前线

   礼貌性约束：维护网站主机名和下载线程 Worker 的映射。FIFO 队列路由器确保每个队列只包含来自同一主机的 URL，Worker 一个一个下载源于同一主机的网页，并中间间隔延时。存储在硬盘，在内存维护缓冲区。

3. HTML 下载器

   遵守机器人排除协议 Robots Exclusion Protocol，多服务器多线程分布式爬虫。由于 DNS 接口同步，所以 DNS 请求时间长，DNS 解析器是爬虫的性能瓶颈，线程发出 DNS 请求时会阻塞其他线程，所以要维护 DNS 缓存，维护域名-IP 地址映射，避免频繁请求。

4. 健壮性

   一致性哈希均衡 HTML 下载器负载。保存爬取状态和数据，应对故障中断。处理异常防止崩溃。添加 PNG 图片下载器、网络监视器等模块。哈希、校验和 checksum 可以查重。设置最大 URL 避免蜘蛛陷阱（爬虫因无限深的目录结构而陷入无限循环的网页）。排除噪声的垃圾 UIRL。

5. 讨论

   (a) 动态渲染/服务端渲染。王者通过 JavaScript 脚本动态生成链接，直接下载并不能获取动态链接，因此解析网页前需要服务器渲染。

   (b) 数据库复制分片，增加可扩展性。

   (c) 横向扩展。用成千上万服务器大范围爬取，要保持服务器无状态。

10 通知

1. 把数据库和缓存从通知服务器中移出，加入更多通知服务器并设置自动横向扩展，引入消息队列来解耦系统组件。

2. 消息队列解决了组件间依赖问题，每种通知都有消息队列，若第三方服务故障也不会影响其他类型的通知。

3. 为防止数据丢失，通知系统在数据库中持久化通知数据，使用重试机制，若发送失败就放回消息队列再重新发送。

4. 分布式性质可能重复通知，使用去重 dedupe 机制，在通知第一次到达时用检查事件 ID 判断是否已出现过。

5. 通知服务器具有身份验证（appKey/appSecret）和流量限制功能。

11 发帖刷动态

两个流程：发布 feed、构建 news feed. 用户发帖会写入缓存和数据库，发帖向服务器发送 HTTP POST 请求，该 API 接受帖子内容和身份验证的令牌 auth_token。广播服务 Fanout Service 把帖子推送给好友 news feed.

1. Web 服务器：与客户端通信、验证用户身份和限流。

2. 广播服务

   (a) 写广播。实时生成 news feed 并立刻推给好友，获取 news feed 很快因为写帖子的时候系统已经算好谁会订阅。如果好友众多，为所有人生成 news feed 会很慢，即热键问题 Hotkey Problem. 对不活跃用户预计算 news feed 浪费资源。

   (b) 读广播。用户加载主页时才生成 news feed 的按需模型，对不活跃用户效果好，不存在热键问题因为数据不会推给好友，但获取 news feed 慢。

   (c) 混合方案，对活跃用户粉丝用按需避免系统过载，用一致性哈希均匀分配数据减轻热键问题。

3. 工作流程

   (a) 从图数据库获取好友 ID

   (b) 从用户缓存获取好友信息，筛选被屏蔽好友

   (c) 发送好友列表、动态 ID 给消息队列

(d) 广播 Worker 从消息队列获取数据，将 news feed 数据存储在 news feed 缓存 <post_id, user_id> 表中。

(e) 将 <post_id, user_id> 存储在 news feed 缓存中。

4. 获取 news feed

用户发请求获取 news feed，负载均衡器把请求分配给各 Web 服务器，Web 服务器请求 news feed 服务来获取 news feed，news feed 服务从 news feed 缓存中获取帖子 ID 列表和帖子内容，将整合好的 news feed 及 JSON 格式返回客户端渲染。

5. 缓存架构

news feed 层、内容层（热点内容、常规内容）、社交图谱层（粉丝、关注者）、操作层（点赞、回复）、计数器层（点赞数、回复数）。

12 聊天

1. 轮询 Polling：客户端周期性询问服务器是否有新消息。效率低。

长轮询 Long Polling：客户端始终打开，直到有新消息可用或者达到超时阈值。一旦收到新消息，就将另一个请求发给服务器，重新开始这个流程。缺点：服务器无法判断客户端是否断开连接；发送者、接收者可能没有连接到同一个聊天服务器；对不经常聊天的用户效率低。

2. WebSocket. HTTP+ 握手，客户端发起，双向通信且持久。发送和接收时都使用 WebSocket 协议。

3. 高层级设计

(a) 无状态服务、有状态服务（聊天）、第三方集成（推送通知）

(b) 可扩展性：用云服务器而非单服务器，防止单点故障。

(c) 存储。通用数据用数据库；聊天记录用键值存储 HBase、Cassandra，因为横向扩展、延时低、关系型数据库不易处理长尾 Long Tail 数据。

4. 数据模型。一对一聊天的主键是 message_id（用 MySQL 的 auto_increment 关键字，或 Snowflake 全局 64 位序列号生成器，或群聊中唯一本地 ID 序列号生成器实现）；群聊的复合主键是 channel_id, message_id。

5. 为客户端选择最佳聊天服务器（基于地理位置、服务器性能）。

Apache Zookeeper：用户登录，负载均衡器将登录请求发给 API 服务器，后端验证用户身份后服务器发现最佳聊天服务器，用户通过 WebSocket 与某服务器建立连接。

6. 消息流

(a) 一对一。用户发送聊天消息给服务器，服务器从 ID 生成器获取消息 ID，服务器把消息发送给消息同步队列，将消息存储在键值存储中。若接收用户不在线，则转发到接收用户的服务器；若接收用户在线，则发送推送通知给服务器。接收用户的服务器转发消息给接收用户，两者保持 WebSocket 连接。

(b) 多设备间同步。手机和电脑都与服务器建立 WebSocket 连接，每个设备都有自己的 cur_max_message_id，从键值存储中获取新消息，消息同步。

(c) 群聊。用户消息被复制到每个群员的消息同步队列，每人都有自己的收件箱。

7. 在线状态

客户端和实时服务建立 WebSocket 连接后，在线状态和最后活跃时间戳存储在键值存储中。用户下线时，在线状态键值存储改为离线。连接断开时，客户端每 5 秒给服务器发送心跳，服务器收到心跳则在线，若 30 秒未收到心跳则离线。在线状态服务器用发布-订阅模型，他的每个好友都订阅了用户在线状态的更新。

13 搜索自动补全 Autocomplete

展示 Top k 查询词，快速响应，自动补全的是相关词，根据流行度排序。

1. 数据搜集：查询字符串及其频率的频率表。

(a) Trie 树获取 Top k 查询词：找到前缀节点，从前缀节点遍历子树并获得合适子节点，对它们排序出 Top k. 通过 Worker 使用聚合数据创建，每周更新字典树，过滤敏感词，按首字母分片并用多个服务器实现扩展存储。

(b) 优化：限制前缀长度，每个节点缓存被高频搜索的查询词。不必每个词都更新、频繁更新字典树。

(c) 用数据分析日志 Analytics Log 追加写入 append-only。判断实时结果是否重要，用聚合器聚合数据（如每周重新构建字典树）方便系统处理。Worker 服务器定期执行异步任务。分布式缓存字典树，每周获取一次数据库快照，序列化后的数据存储在 MongoDB。把字典树键值存储映射到哈希表。

2. 查询 Top k。频率表的两个字段：query（字符串）、frequency（搜索次数）。

(a) 负载均衡器把查询请求转发给 API 服务器，API 服务器从字典树缓存中读取数据，构建客户端自动补全建议。若数据不在缓存中，则填充回缓存，若缓存服务器宕机或内存溢出，会发生缓存未命中。

(b) 要快。浏览器发送 AJAX 请求获取自动补全结果，这样不用刷新整个网页。用浏览器缓存自动补全建议，因为短期内不会变化。

3. 其他。支持多语言，在字典树节点存储 Unicode 字符。为不同地区构建不同字典树，存储在 CDN 提高响应速度。不支持爆炸性新闻。改变排序模型，给最新查询词分配高权重。流数据处理用 Apache。

14 YouTube 视频

客户端访问 YouTube，视频存储在 CDN 中，播放时视频流从 CDN 中传输出来。除了视频流之外的请求（推荐视频、生成上传视频 URL、缓存）都被发往 API 服务器。

1. 视频上传

视频元数据存储在元数据库，利用分片和复制满足高可用性。用 Blob (Binary Large Object) 存储系统存储视频。视频转码（编码）服务器，基于不同设备和带宽提供合适的视频流。转码后的视频也用 Blob 存储。消息队列存储视频转码完成的信息。完成处理器由一系列 Worker 组成，从完成队列中拉取事件数据并更新元数据缓存和数据库。

(a) 视频上传流程

转码服务器从原始存储中获取视频并转码，转码结束后，视频被发到转码存储中（分配到 CDN）、转码完成事件被加入完成队列（完成处理器更新元数据库、元数据缓存），API 服务器通知客户端视频上传成功，准备流式传输。

(b) 更新视频元数据。视频上传到原始存储时，客户端发送请求更新视频元数据（视频 URL、大小、分辨率、格式）。

2. 视频流传输 Streaming Flow 流程。视频从 CDN 开始流式传输，一边加载视频一边看视频。比特率 bitrate 是视频中每秒传输的比特数，高比特率就是高清画质和快网速。

3. 视频转码格式

(a) 容器：包含视频文件、音频、元数据的篮子。可以用文件扩展名 mp4 确定容器格式。

(b) 编解码器 Codec：压缩和解压缩算法，减小视频大小还能保证视频质量。

4. 有向无环图 DAG 模型：为了支持不同视频处理流水线、保持高并行，加入的抽象，让客户端定义不同阶段的任务（视频转码、缩略图、水印、音频编码）。

5. 云服务视频编码架构

(a) 预处理器。分割视频成几秒钟的小单元图像组 GOP (Group of Pictures)，确保这些 GOP 在视频流中的位置对齐，并缓存这些分割后的小视频在临时存储中，若视频编码失败，可以用保存的数据重试。

(b) DAG 调度器。把 DAG 分成两个阶段（处理视频、视频编码和缩略图）的任务，并放在资源管理器的任务队列

(c) 资源管理器。管理资源分配的效率。包含任务优先级队列、可用 Worker 队列、当前任务运行队列、任务调度器（选取合适的 Worker 执行任务）。

(d) 任务 Worker

(e) 临时存储。把元数据缓存在内存中，视频放在 Blob 中。

(f) 编码后视频输出

6. 系统优化

(a) 提速：并行上传各 GOP 小视频。

(b) 上传中心：美国用户上传到美国中心。

(c) 松耦合、高并行：用消息队列，编码模块不再需要等待下载模块的输出，若消息队列有任务，编码模块可以并行处理任务。

(d) 预签名 URL（共享访问签名 Shared Access Signature）：只有授权用户才能上传。客户端向 API 服务器发送 HTTP 请求获取预签名 URL，从而获取预签名 URL 中所标识对象的访问权限。客户端收到 API 服务器返回的预签名 URL 后，就用这个预签名 URL 上传视频。

(e) 保护版权。数字版权管理 Digital Rights Management (DRM) 系统、AES 加密视频、可视水印。

(f) CDN 很贵，节约成本。YouTube 视频遵循长尾分布 Long-Tail Distribution，热门视频频繁播放。仅经 CDN 提供最流行视频，冷门视频用大容量视频服务器提供，短视频按需编码，特定地区流行的视频不必分发到其他地区。构建自己的 CDN 并和 ISP (Internet Service Provider) 合作。

7. 高容错系统

   上传、转码、视频分割、DAG 任务调度器错误、Worker 不可用：重试。预处理器错误：重新生成 DAG，API 服务器不可用：重定向到另一个 API 服务器。资源管理器队列不可用：使用副本。元数据缓存服务器不可用：数据复制多次，访问其他节点，换一个服务器。元数据库：若主库不可用，就推举一个从库作为新主库，若从库不可用，换一个。

15 云盘

3 个 API：上传文件，下载文件，获取文件修改信息。验证用户身份、HTTPS 协议，SSL 协议保护客户端和后端的数据传输。

单服务器解耦：

1. Web 服务器：添加负载均衡器后，根据流量负载加减 Web 服务器。
2. 元数据库：把数据库从服务器移出，避免单点故障。设置数据复制、分片满足可用性、可扩展性。
3. 文件存储：用 Amazon S3，在两个分隔的地理区域之间复制。
4. 通知服务：发布者-订阅者系统，文件编辑修改通知客户端文件最新状态。
5. 离线备份队列：客户端离线，无法接收通知，离线备份队列存储信息，当客户端上线时可以同步更新文件。

设计：

1. 同步冲突：系统先处理的版本胜出，后处理的版本收到冲突通知，可以选择把两个文件合并或覆盖。
2. 块服务器 Block Server：文件被分成几块 Block，每块都有唯一哈希值存储在元数据库，每块都是独立对象存储在 Amazon S3。

   最小化传输数据量：

   (a) 增量同步 Delta Sync：文件被修改时，同步算法，仅同步被修改的块而非整个文件。

   (b) 对块进行压缩。

3. 强一致性

   内存缓存用最终一致性模型，不同副本可能有不同数据。为了实现强一致性，确保缓存副本数据和主数据库数据一致，对数据库写入时让缓存失效，确保缓存和数据库相同。

   关系型数据库维护 ACID，但 NoSQL 不支持 ACID。

4. 元数据库：user, device, workspace, file, file_version, block.

5. 上传、编辑流程

   客户端将文件上传到块服务器。块服务器把文件分块，对块压缩加密并上传到云存储。云存储触发上传完成回调，将请求发给 API 服务器。元数据库中，文件状态为已上传。告知服务器又一个文件状态为已上传。通知服务器通知给其他客户端该文件已上传。

6. 下载

   通知服务告知客户端，文件在别处被修改。一旦客户端知道有更新，就发请求获取元数据。API 服务器向元数据库发请求获取改动的元数据，元数据返回给 API 服务器，客户端获取元数据。一旦客户端收到元数据，就向块服务器发请求下载块。块服务器先从云存储下载块，云存储将块返回给块服务器，客户端下载所有新块来重构文件。

7. 通知服务

   (a) 长轮询。若检测到文件变更，客户端关闭长轮询连接，客户端必须连接元数据服务器下载最新变更。

   (b) 不用 WebSocket。因为 WebSocket 适合实时聊天，对于云盘，没有数据爆发时，不会经常发通知；与通知服务的通信非双向，仅服务器发送文件变更信息给客户端，而非反之。

8. 节约存储空间

   (a) 数据库去重，若两个哈希值相同则一样。

   (b) 智能数据备份。限制版本最大数量，若达到阈值则最老版本被替换为新版本。

   (c) 不常用数据放进冷存储。

9. 故障处理

   (a) 负载均衡器之间用心跳信号检测故障。

   (b) 块服务器、云存储、API 服务器、元数据缓存服务器：复制多次、重定向、访问其他服务器。

   (c) 离线备份队列故障：复制队列多次，若一个队列故障，队列消费者重新订阅备份队列。

   (d) 通知服务故障：若一个服务器故障，所有长轮询连接都消失，客户端需要重连另一个服务器。

   (e) 元数据库故障：主节点故障就把一个从节点提升为主节点，从节点故障就启用别的从节点。

16 支付宝收付款

1. 收款。业务事件：触发资金流动。支付系统（支付 API+ 支付核心 + 支付传输）：记录资金流动，但没有真正转移资金。Payment Service Processor (PSP)：转账从 A 到 B。

   (a) 复式记账 Double-Entry System：借记和贷记。

   (b) 路由服务定义支付路由规则，动态将支付交易路由到合适的 PSP。可降低支付处理成本，最大化成功率，减小故障影响，降低支付处理延时。

(c) 支付集成服务，提供统一 API 与第三方 PSP 和银行通信。实时（HTTP 协议）、不实时（批量文件集成 SFTP 协议，异步批处理大数据）。

(d) 令牌保险库，关系型数据库。支付令牌化是用唯一的支付令牌替换敏感的信用卡信息，可以无卡支付。

2. 卖家仪表盘。Apache Kafka 发布-订阅消息系统。

   Kafka 与消息队列区别：

   (a) 消息队列的事件被处理后就从队列中移除，一个事件是一个消费者处理的。

   (b) Kafka 的事件被处理后不会被移除，而是停留在队列，直到队列大小超过限制。同一个支付事件有多个下游服务，如付款流程、报告流水线、会计，所以适合 Kafka。

3. 重试：至少一次交付。

   若网络故障不能短时间解决，则使用指数退避 Exponential Backoff，请求失败则 1 秒、2 秒、4 秒后重试。

4. 幂等：至多一次支付，客户端重复发一个请求且结果相同。

   幂等键 Idempotency Key（如 UUID）是客户端产生位移值，在一定时间后过期。若顾客点了两次付款，第一次的幂等键作为 HTTP 请求发给支付系统，因为支付系统已经见过这个幂等键，所以第二个请求视为重试，只有一个请求会被处理。

   若支付已处理，但网络错误丢失响应：需要和外部系统 PSP 交互时维持幂等，并跟踪交易成功、失败、等待状态。

5. 同步通信（HTTP 协议）：设计简单，不允许服务自治，随着同步依赖增加而性能变差。

   (a) 客户端发送请求，等服务器响应，将连接保持打开状态，直到知道交易结果。

   (b) 性能低，故障隔离差，耦合度高（请求发送者需要知道接受者），很难扩展（若没有队列作为缓冲区）。

6. 异步通信：牺牲简单和一致性，换取可扩展性和故障容忍性，适合大型支付系统。

   (a) 客户端不等服务器响应，一旦发送请求就关闭连接。请求被处理后，结果通过网络钩子/回调 Webhook 发给客户端。

   (b) 客户端通过 HTTP 发送支付请求，触发业务事件，事件放入 Kafka 队列。异步 Worker 消费业务事件。一旦业务事件被异步 Worker 处理完，后端就发送响应给客户端。

   (c) 单接收者：共享的消息队列有多个订阅者。多接收者：消费者收到消息时，消息没有从 Kafka 中移除，同一条消息可被不同服务处理。

7. 数据一致性：支付系统内部与 PSP 中外部状态一致。

   (a) 单体数据库：transaction 记录交易，transaction_entry 存储交易不同状态。

   (b) 分布式数据库：幂等数据先被写入主数据库，再被复制进副本数据库。

      i. 为了避免副本滞后，只在主数据库存储读写幂等数据，但缺乏可扩展性，对此 Airbnb 用幂等键对数据库分片。

      ii. 强一致性模型，客户端看不到过时值，但牺牲了性能，要等待最慢副本的响应。

iii. 用共识协议（多个服务器达成一致）进行复制，如 Two-phase Commit, Raft, Paxos, Saga 模式。

8. 修复不一致

(a) 同步修复：通过后续读写修复不一致。

(b) 异步修复：使用消费者、定时任务、表扫描发现一致性问题并修复，通过匹配交易日志和外部数据 PSP 实现支付对账 Payment Reconciliation。

(c) 网络请求不可靠：当数据库事务活动状态时，不要发送网络请求。
一个 API 请求分为 3 阶段：预 RPC、RPC（Remote Procedure Call，应用程序可以向远端服务器发送请求）、RPC 后。数据库交互只在预 RPC 和 RPC 后阶段，网络请求只在 RPC 阶段。

9. 交易失败

(a) 任何阶段都要保存支付状态，存储在只可追加写的数据库表。

(b) 重试队列（暂时性错误都被路由到重试队列）、死信队列 Dead Letter Queue（反复失败的消息，可以调试和隔离有问题的消息并手动检查）。

(c) 每个事件都有唯一 ID 确保事件只被处理一次，事件被处理之前存储在持久化存储。

17 指标监控告警

1. 指标数据（时间序列）收集

优缺比较	拉	推
容易 debug	【优】应用服务器/metrics 端点用于拉取指标，可在任何特定时间查看指标。	
健康检查	【优】若应用服务器未响应请求，可以判断服务器故障。	若指标收集器未收到指标，可能是网络问题。
短时任务 Short-lived Job		【优】批处理任务可能短暂，不需要长时间拉数据，可以引入拉模型的推送网关。
防火墙		【优】要让服务器拉取指标，需要保证所有指标端点都能访问，因此网络安全配置复杂。
性能	TCP 协议	【优】UDP 协议，数据延时低
数据可靠性	【优】要收集指标数据的应用服务器是事先在配置文件里定义的。	任何类型的客户端都能推送指标数据给指标收集器，可用白名单限定允许哪些服务器接收。

(a) 拉。指标收集器从服务发现组件获取 Web 服务器配置元数据（爬虫时间间隔、IP 地址、超时时间）。指标收集器通过 HTTP/metrics 端点拉取指标数据，为暴露/metrics 端点，要在 Web 服务器上安装客户端库（如 SDK）。(Prometheus)

(b) 推。多个指标来源（Web 服务器、数据库集群）直接将数据发给指标收集器。(Amazon CloudWatch, Graphite)

缺点是给指标收集器带来巨大流量，解决方案是在指标来源上安装 agent 代理。代理是运行在应用服务器上的软件，从应用服务器收集各指标数据并处理，然后推给指标收集器。若推送流量大，代理就进行调整以避免指标收集器被压垮。

2. Kafka 扩展系统

(a) 指标收集器将指标数据发送给队列系统 Kafka，然后消费者或流处理服务（Apache Storm, Flink, Spark）处理数据并推送给时间序列数据库。

Kafka 高可靠、可扩展的分布式消息平台，解耦了数据收集和数据处理服务，若数据库不可用则可以把数据保留在 Kafka 防止丢失。

(b) Kafka 内置分区机制扩展系统。可以基于吞吐量的需求配置分区数量，按指标名字分区聚合，进一步按标签分区，对指标分类排优先级。

3. 查询。InfluxDB 用 Flux 语言，专门针对时间序列分析，比 SQL 简便易懂。

4. 存储

(a) 在时间序列数据库上添加缓存层。预写日志 Write Ahead Log (WAL)、缓存、时间结构合并树 Time-Structured Merge (TSM) Tree、时间序列索引 Time Series Index (TSI)。

(b) 聚合。查询低延时、减少存储，用秒、分钟、小时等粒度聚合、存储时间序列数据。

(c) 空间优化。数据编码压缩、向下采样 Downsampling（高分辨率数据转换为低分辨率数据）、冷存储。

5. 告警

过滤合并重复告警，让值班人员不被告警数量压垮。告警存储是键值数据库 (Cassandra)，保存所有告警的状态机，确保至少发送一次通知。合格的告警被插入 Kafka，告警消费者处理 Kafka 告警事件并发送通知。

参考文献

[1] Alex Xu. 搞定系统设计. 电子工业出版社, 2023.