

① 高并发系统：通用设计方法

横向扩展 Scale out 多线程防御. ← Scale up 提升单机性能.

缓存 延长响应时间.

异步 通用方无需等待方法执行完成，即可返回并执行其它逻辑.

系统设计不断演进，不同层级系统有不同侧重点。
以解决系统中存在为困难和驱动力。

· 最简单系统设计，满足业务需求和流量现状，选择最熟悉技术体系。

· 业务化/流量增加，修正存在问题，选择社区成熟，团队熟悉的组件。

· 小修小补无法满足需求，考虑重构/重写大调整。

② 模块分层

业务越来越复杂，大量代码/模块相互依赖，扩展性差，牵一发而动全身。

层间：
① MVC、Web (表现)
② Service (逻辑)
③ DAO (数据访问)

优势：
① 简化设计，各司其职，~~每层~~ 分别专注对应一层。

② 可以做到高复用。

③ 方便做横向扩展。（只抽取瓶颈的一层做扩展）

如何做：

层次边界的规定

相邻层相互依赖，相互数据流转。

*阿里巴巴Java开发手册 v1.4.0
(详尽版)

不足：

增加代码复杂度。

多层架构性能有损耗（如通过网络进行交互）

选择分层结构，不是可选或可以弥补（解决方案）

与软件设计原则相违。

思：

日常开发中，你如何分层？优势是什么？

③ 系统设计目标（一）如何提升系统性能

高并发、高性能、高可用。

高并发系统设计目标：高性能、高可用、可扩展。

↑ 让系统能够处理更多用户并发请求。

可扩展：在短时间内可以完成扩容，平稳承担峰值流量。

高性能：可表现在体验上。

高可用：系统正常服务用户的时间。

性能优化：

原则：不能盲目、问题导向；“八二原则”，20%精力解决80%的性能问题；

数据支撑：响应时间/吞吐量变化；持续优化，明确目标，找性能瓶颈，制定方案。

度量指标：高的响应时间（一段时间）

平均值（少量敏感度差）参考值。

最大值（过于敏感）

分区值，拥堵，取分位数（如第90位）

吞吐量，响应时间，质量并发与流量。

优化目标示例：每秒1万次的请求量，响应时间 99 百分位值在 200ms 以下。

经验：200ms 是分界点，200ms 以下感觉不到延迟。

1S 是分界点，超过 1S 会有明显等待的感觉。

健康的系统，99% 位数的响应时间 200ms 之内，而不超过 1s 的请求占比要在 99%~99.99% 以上。

高并发下的性能优化：

- ① 提高系统的处理核心。任务中单线程数占比越大，功耗比越大。
性能测试中的资源模型，无法无限剧增处理核心数。
- ② 减少单次任务的响应时间。压力测试，找到瓶颈，知道系统承载能力，找到系统瓶颈持续优化性能。
CPU 集型，使用更高效算法，减少运算次数。
通过 Profile 工具找到消耗时间 CPU 最多的地方 / 模块。

I/O 集型（网络 / 磁盘）数据角，缓存，Web 系统。

LINUX 工具集，排查瓶颈。

监控，发现性能问题。

优化方案：

数据锁，锁表，全局锁，索引，缓存是否有...
不同问题制定不同方案。

思考：

日常工作中，有哪些优化手段与经验。

基础优先，性能监控要做好；掌握优化工具与方法；基础知识。

④ N(2) 怎样做到高可用。

高可用性，系统无故障运行的能力

衡量：MTBF，平均故障间隔（可运行时间）↑ $\text{可用性} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$
MTTR，平均故障恢复时间。（故障时间）↓

× 99 的可用性。

90%，99%，99.9%，... 36.5 天，3.65 天，8 小时，... 2.4 小时，14.4 分，1.44 秒。
可用性 故障

一般，核心业务平行化，非核心业务 3 个冗余。

运维稳定性，故障处理流程大，业务变更流程，发生故障的自动恢复，工具建设。

系统警报与自动快速恢复。

思路：

① 系统设计。

考虑如何自动化的发现故障，之后要如何解决。
优化方法：failover（故障转移），超时机制，降级与限流。

不要过度优化。
可用性与性能上要视具体情况作取舍。

② failover

对等，切换其连接点即可。

非对等（备等架构），故障检测与自动切换。

心跳包。
适当的分布，一致选举法。

RPC 架架。

③ 超时机制。

组件 / 服务间的调用，延迟而导致阻塞。

超时时间，采集 99% 的响应时间，根据该时间进行设定。

有损但有效。

④ 降级，限流 → 限制并发请求（短暂停）

保证核心服务稳定性非核心服务。

⑤ 系统运维。

A. 故障恢复发布。

变更管理，快速回滚恢复，灰度发布。

B. 故障演练。

混沌工程，工具“Chaos Monkey”。

可在测试环境中进行。

思考：工作中有哪些保证系统高可用的设计技巧？

⑤ ~ (三) 易于扩展如何实现

平稳期，预留 30%-50% 资源应对峰值流量。

扩展性提升策略：

分层系统上的瓶颈点制约横向扩展能力。

无状态服务更易于扩展。如 MySQL 则为有状态，会涉及大量数据迁移。

要站在整体架构角度，数据层 / 缓存 / 第三方依赖 / 负载均衡 / 交换机带宽。

高扩展性设计思路

思：

拆分：将复杂的逻辑简单化，原来的系统拆分为单一职责、独立的模块。

存储拆分，考虑业务维度（分库）/ 次拆分，水平拆分（分表），根据数据。

节点的增加要基于长远作预留，避免频繁扩容。

事务、跨库的协调成本高，尽量不要使用。

业务拆分：相同的业务服务拆分成单独的业务池，业务依赖独立的数据源。

根据业务的重要性划分（核心池 / 非核心池）

客户端类型（外网池、H5 池、内网池）

优先扩容

降级

拆分前后简单，拆分后复杂需投入大量人与精力。

思：关系型数据库可扩展性差，常见 NoSQL 数据库是如何解决的？

⑥ 面试：问及组件实现原理的小图与处理（回答）

○ 项目的问题、未来的发展思路、可能走向。

基础数据结构、算法及设计思想的掌握。

基本原理、提供设计思路、充分发挥优点、避免踩坑。

① 书籍，《算法导论》《TCP/IP 协议详解》《深入理解计算机系统》。

② 官方组件、官方文档。

③ 博客、博客园、转发编程网、架构师之路。

⑦ 池化技术

MySQL 拦截包创建连接，造成响应时间长。

用预热连接池预先建立的数据源连接。

数据库（MySQL）连接池、HTTP 连接池、Redis 连接池。

* tcpdump 可以抓取包（报文）

连接池 最小连接数 / 最大连接数。（一般线上为 10 / 20-30）

连接维护问题：① 数据库的 IP 变更，连接池插入了旧的（不可用）的服务器。

方案：② MySQL wait_timeout 主动关闭连接。

① 定期检查连接池连接时是否可用（如发送 select 1，异常则移除并尝试重用）。

C3P0 采用，推荐。

② 检查 SQL 前先检测，DBCP testOnBorrow 的开启，不建议线上使用（引入死锁风险）

线程池预先创建线程（JAVA）

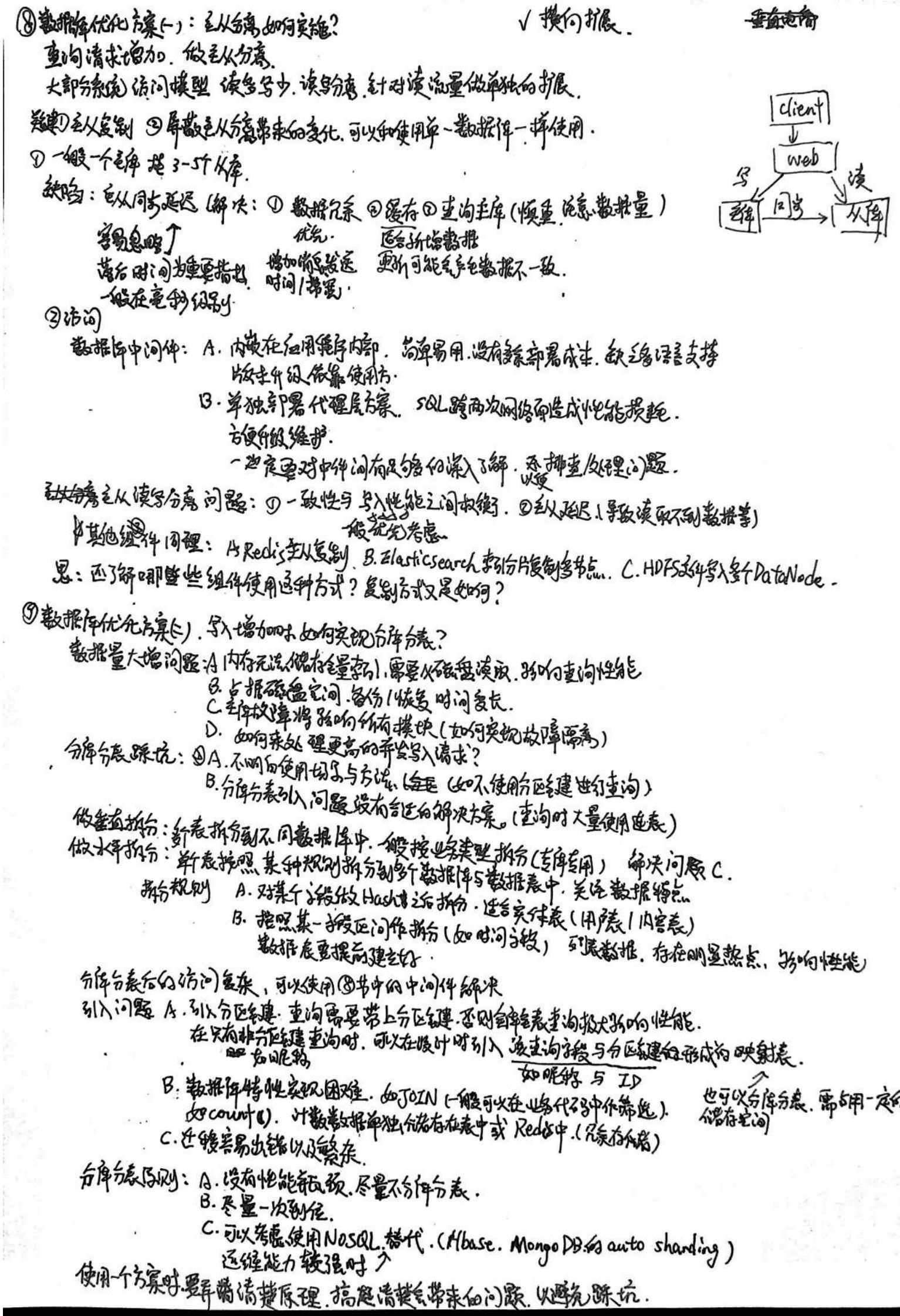
线程堆栈量，可能与 coreThreadCount 与 maxThreadCount 设置过小有关。

不要使用无界队列，内存占满导致频繁的 GC 引起宕机。

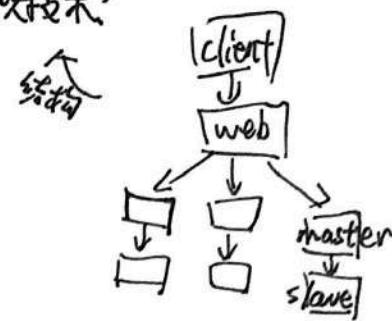
连接 / 线程池的创建耗时 / 消耗资源，使用池技术 提高性能与实用性。

软件设计思想：池化技术（空间换取时间）

思：还使用过其他池化技术吗？



问：分布式ID这样的解决方案，还了解哪些分布式存储组件也使用了类似技术？
实现方式？



⑩ 发号器：如何保证分布式部署后ID的全局唯一性？

分钟方表，全局唯一的全局唯一性。

○ 主键选择：A. 使用业务字段（通常不选）
B. 额外使用生成的唯一ID（推荐）

分钟方表后无法使用自增字段保证全局唯一性。

○ 基于 snowflake 算法搭建发号器。
简单易实现、全局唯一，单调递增，含有业务上意义（可以反解，用于排序）

snowflake 64bit二进制。4位时间戳 2869年。10位机器ID(2-3位IDC, 7-8机器ID)
12位序列号(4096个ID每毫秒) 4位IDC 128/256台机器。
发号器可以多个或单个节点，可以使用keep alive 保证可用性。
加入业务ID，可以反解得出是哪个业务的ID。

○ 其他实现方式。

A. 嵌入业务代码（不用跨网路）需要引入Zookeeper等分布式一致性组件保证机器重启能获得唯一机器ID。
B. 独立服务部署（一个或者多个）只有一台工作 单实例单CPU可达2W每秒。

缺点：依赖时系统时间；发号器QPS不高每毫秒只发一个量，若未级为1，发号不均匀。
①时间戳以秒级计算 ②序列号起始号随机。

D 方案在可选，在最合适，弄懂背后原理，落地。

思：你的系统中，你的ID是如何生成的呢？

⑪ NoSQL 在高并发场景下，数据库和NoSQL如何做到互补。

NoSQL 不使用SQL作为查询语言。

提供优秀的横向扩展能力与读写性能。

Redis, LevelDB 选择的k-v存储，有较高读写性能。

Hbase, Cassandra 列式存储 适用于离线数据统计分析。

MongoDB, CouchDB 文档型数据库 schema free(字段任意扩展)

NoSQL 性能更高，更方便，适用于互联网项目常见的大数据量场景。

SQL 强大查询功能，事务，灵活的索引等。

○ 使用NoSQL提升写入性能

传统数据库不可避免的都做IO。

NoSQL 很多使用LSM树(Log-Structured Merge Tree)，牺牲读性能换取写高性能。

○ 数据写入（按照key排序）的MemTable中（内存），使用 write-ahead log 备份在磁盘
到一定数据量写入一个新文件（sorted String Table）（有序）
sstable文件达到一定数量会合并（有序文件合并速度快）

○ 读取数据先找 MemTable（内存），再找 sstable（多个文件，但有序）。效率低于BT树索引。

△ 模糊搜索。

Elasticsearch 倒排索引。（分词与记录ID和映射关系），提供分布式的全文搜索功能。

△ 提升扩展性

NoSQL分布式，大数据存储场景。

MongoDB. ① Replica. 副本集 ① 负责主从分离，以及读写分离 ② 主从故障切换

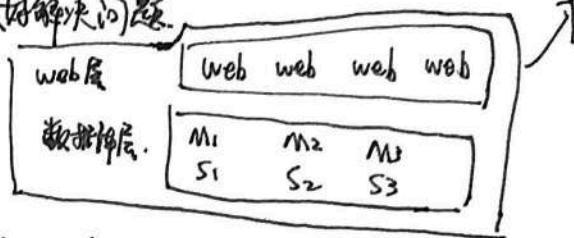
B. Shard, 分片. 美化分库分表

Shard Server 数据节点
Config Server 配置节点, 包含元数据信息
Route Server 仅作为路由, 从 Config Server 收取信息, 将请求路由到 Shard Server

C. 负载均衡子. ① shard 之间数据不均匀重分布 Balance.
② 扩容时, 数据自动做迁移.

造型, 需要对它的实现原理有深入的了解. 运维方面对它熟悉, 以好解决问题.

思: 日常工作中用了哪些 NoSQL 数据库? 基于什么考虑?



⑫ 缓存: 数据伸缩成为瓶颈后, 动态数据查询要如何加速?

并发个数增加, 磁盘 I/O 成为了瓶颈, 这时可以使用缓存.

什么是缓存: 存储数据的组件, 对数据请求更快地返回. 缓存 + 内存.

① 缓存: 容间操作时间.

② 缓存. Linux 内核管理 MMU, 虚拟地址映射到物理地址. TLB 缓存最近转换过的映射. 短视频 APP, 在后面的二、三个视频作缓存, 以提高播放的速度.

③ 缓存. 网站资源的缓存(浏览器)

→ 类似消息队列, 补齐高速设备与低速设备间通信时的速度差.

如内存 → 缓冲区 → 磁盘

④ 缓存分类: A. 随机缓存 (渲染成静态 HTML)

B. 分布式缓存 (动态资源)

C. 热点本地缓存 (根据热点数据查询)

本地缓存方案 HashMap, Guava Cache, Ehcache.

⑤ 缓存不足: 提升访问速度, 适合读多写少, 存在热点的数据; 否则效果不明显.

B. 给操作系统带来复杂度, 故障后不易恢复.

C. 内存带宽资源冲突, 故障后不易恢复.

D. 缓存容量通常比较小, 大数据量要慎用

】过期时间要精细化

缓存可能会造成.

关注缓存命中率.

缓存思想.

思: 日常工作中看到了哪些使用了缓存思想的设计?

⑬ 缓存使用 (一) 如何选择读写策略.

选择策略, 考虑问题: 缓存中脏数据? 策略读写性能? 缓存命中率?

① Cache Side (旁路缓存) 策略. (常用)

缓存与数据源的副本不一致 / 丢失更新. (更新缓存产生的问题)

旁路: 更新数据源并删除缓存. (注意顺序)

先读缓存, 读后则从数据源读取并更新缓存.

写入后 (插入) 即读数据因为空从缓存不到数据. 可在插入后直接更新缓存解决.

写入锁策略时, 缓存会被清理. 影响命中率: 解决方法:

A. 更新缓存, 但在更新前添加分布式锁

B. 给更新数据时再给锁 (过期时间).

② Read/Write Through (读穿 / 写穿) 寄存器 ↔ 缓存 ↔ 数据库 (少用)

写: 更新到缓存, 由缓存更新数据库. Write Miss 可以选择写入缓存并更新数据库或不写缓存直接更新数据库.

Guava Cache 支持类似 Read Through.

① Write Back(写回)策略

更新写缓存，并标记为脏块。在读取对应数据时，如果未命中，则从数据库中取出要读取的数据，写入缓存并返回。不是脏块直接在后端数据库中读取，写入缓存并返回。
类似 Page Cache / Redo Log 的写入等。

问：日常工作中，使用了哪些缓存读写策略？



② 缓存的使用(二) 如何做到高可用

命中率 = 命中请求数 / 总请求数。核心缓存 99% 甚至 99.9%。

○ 方案：A. 客户端方案：客户端中配置多个缓存节点，写入/读取策略实现分布式（多语言无法复用）

B. 中间代理层方案：客户端所有请求通过代理层访问缓存，其中内置高可用策略。

C. 服务端方案：Redis 2.4 以后的 Redis Sentinel 方案。

○ A. 关注贯穿读写两方面：写 → 数据分片 读 → 主从/多副本两种策略。

○ 如何分片：通过分片算法，将数据放在许多不同节点上。

(Hash 分片算法： $hash(key) \% nodeNum$)

(一致性 Hash 算法：Nodes 组成圆环，key 会映射到第一个节点即使用它)

缺点：分布不平均；脏数据；可能造成系统雪崩

引入虚拟节点，设置过期时间，减少脏数据概率。

○ Memcached 主从机制。

更新时 Master-Slave 双写，读取优先读 Slave，之后再读 Master

○ 多副本

Master-Slave 上层添加副本组（多个），优先访问副本组，没有再访问 M-S。

Master-Slave 也可以作为一组副本组以保持数据热点。

一致性哈希法



○ B. 中间代理层

Merouter, Twemproxy, Codis

读写请求都经过代理层完成。

○ C. 服务端方案

读写不经过 Sentinel，作为管理者存在。

监控 Redis 节点和 slave 转主节点的选举。

问：缓存高可用性的重要性，可用性下降会造成什么严重问题？
你们如何来保证缓存的高可用性？

③ 缓存的使用(三) 缓存穿透怎么办？

核心缓存命中率 99%，非核心缓存命中率 90%。

缓存穿透：缓存中没有查到数据从而需要后端系统中查询。

缓存容量有限，只存储 20% 的热点数据。

大量穿透请求，对系统有害。

解决方案：A. 查询的值为空，或查询时发生异常，导致缓存无法命中。

① 回种空值（过期时间要较短，需要评估缓存容量，若无法支撑大量空值，不建议使用）

② 布隆过滤器，通过 Hash(id)，将其对应数组的值设置为 1（表示存在）

缺陷：Hash 冲突，有判断错误几率（不存在以为存在）→ 使用多个 Hash 算法。

不支持删除元素。→ 可以将 0 和 1 变为存储计数，生成多个散列组。

建议：计算多个 Hash。→ 增加空间消耗，都为 1 时才表示存在。

评估业务场景下需要的内存。

存储成本是否接受。

热点缓存失效，造成后端的极大压力，dog-pile effect(狗堆效应) 单发穿透
尽量减少缓存穿透后的并发。

- 后台线程穿透数据库并更新到缓存，而对这个缓存的请求都不再穿透直接返回
- Memcached/Redis中设置分布式锁，获取到锁才穿透。

核心：减少对数据库的并发请求。

问：日常工作中，还有哪些解决缓存穿透的方案呢？



(16) CDN：静态资源如何加速

静态资源 → 就近访问

CDN 静态资源分发到多个地理位置机房中，解决就近访问，也加快访问速度。

问题：① 如何将请求映射到 CDN 节点上 ② 如何根据地理位置选择较近的节点。

DNS → cname.

DNS解析时间可能会较长。

(如果是APP，可以在部署时进行预解析，并定时更新)

关注 CDN 的命中率与源站带宽情况

~ GSLB,全局负载均衡

让流量平均分配以及就近选择。

问：除了厂商对于SLA的保证外，还有什么方案可以保证CDN的可用性？

Special 数据迁移应该怎么做？

数据库迁移：①在线迁移，②要保证数据的完整性，③迁移过程可以回滚。

方案：A. 双写方案（MySQL, Redis）

将新库作为从库来同步数据，多库多表用第三方工具（如Canal）获取binlog增量日志。

改造业务代码：数据写入要同时写入新库（可异步）

抽离数据（抽取部分数据并校验一致）→ 提前写好校验工具或脚本。
灰度切换，到全部切换
在测试环境充分测试，再开始迁移。

出现问题可切换回旧库。

没有出现问题，观察几天可修改成只写新库即可

B. 级联向导方案：

将新库配置为旧库的从库，用外数据同步。

备库配置为新库的从库，用外备备份。

三库写入一致后，将数据库的读写流量切到新库。

暂停应用写入，将写流量切割到新库（业务低峰期）

← 从本地迁移到云

← 短暂停写。

缓存迁移：保证迁移缓存的热度。

使用副本组(memcache)

优化，只有部分（如10%）的流量走新缓存副本组，减少专线的网络占用。

问：你在数据迁移时用了什么方案？优势与劣势？

(17) 消息队列：你该如何处理每秒上万次的下单请求。

高并发写请求：秒杀。

大量读请求：① 热点数据（如秒杀商品）缓存。② 静态化数据 CDN 缓存。③ Web 访问分布式缓存节点。
避免请求业务服务器

④ 限流策略。

消息队列：暂存数据的容器，平衡低速系统与高速系统任务处理时间差的工具。

例子：JAVA 缓解池，RPC 框架网络请求处理。

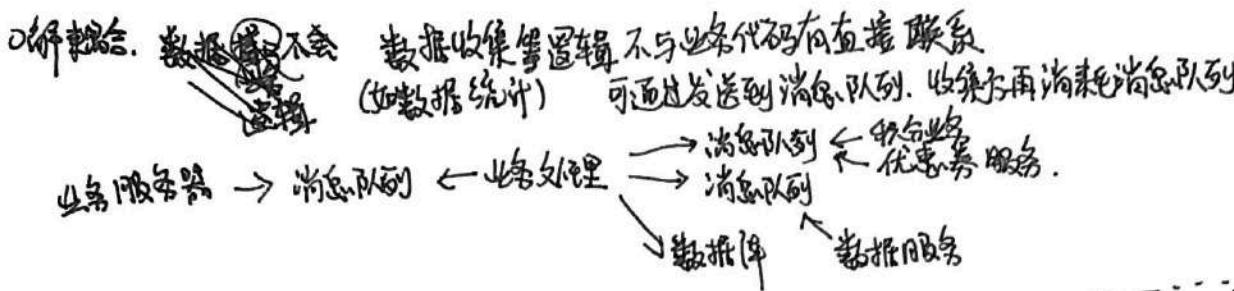
队列内。

降低峰值流量

~ 秒杀请求暂存在消息队列中，释放系统资源后再处理其它请求。（削峰填谷）短暂停延迟处理。

要对处理队列处理时间、前端写入流量大小、数据库处理能力做好评估，根据不同量级部署合适数量的队列。
消费者。

简化请求中的业务流程，次要业务逻辑放入其它消息队列处理（异步处理），主要逻辑得到简化。



问：你在开发过程中，会在什么样的场景下使用消息队列？

(18) 消息投递：如何保证消息仅仅被消费一次。

消息为什么丢失？

- ① 生产者写入过程
- ② 消息队列中有宕机时场景
- ③ 消消息被消费者消费过程。

① 生产者 → 业务服务器，与消息队列通过网络传递。

网络抖动可能导致消息丢失。

方案：消息重传，发送超时后重传，但可能导致消息重复。

② 消息队列与 Page Cache 同步重启掉电则数据丢失。

方案：以集群方式部署消息队列（如 kafka），有性能损耗。

如果业务对消息丢失有一定容忍度（一般也有），可不部署集群。

③ 如 kafka，消费会更新消费进度。

方案：在消息接收并处理后，在更新消费进度，但可能没有即时更新消费进度而重复消费。

如何保证消息只消费一次。

多次消费重复消息与只处理一次结果相同（幂等）

① 如 kafka 0.11 与 pulsar 支持 producer idempotency 来保证消息队列中消息的唯一性。

② 消息生产时可以使用发号器生成全局唯一 ID 作为消息 ID，消费时可根据消息 ID 查询之前是否处理过。

另外引入事务，可以保证处理与写入消息 ID 同时成功或失败。（如处理后消息 ID 放在数据库）

但一般可以不考虑引入事务。

③ 业务层面：如使用乐观锁的方式（数据增加 version 版本号）

根据场景，进行合理的多方案设计。如电商场景，可容忍偶发几条的丢失。不能一概而论。

问：保证消息处理幂等性，还了解哪些方法？

(19) 消息队列：如何降低消息队列系统中消息的延迟？

监控消息延迟：① 使用消息队列提供的工具，监控消息的堆积。

kafka kafka-consumer-groups.sh

JMX，编写代码获取，推荐。

通过生成监控消息的方式监控延迟 } = 增加扩展。

减少消息延迟：消费端：① 提升消费代码性能 ② 增加消费者数量。（kafka 不可用，可另外增加分区）

消息队列：消费端找不到消息时，等待一段时间再拉取（10-100ms）可让消费端增加多线程处理。

消息队列：消息中间件：① 消息的存储 ② 增加拷贝或以一定步长。

问：在研发过程中。

在降低消息延迟方面做过哪些事情？

(20) 面试第二期：当问到项目经历，面试官究竟想要了解什么？

项目背景不宜过多，项目整体架构 一体化？

引入熟悉的领域，多讲相关内容 服务化？提供哪些服务？服务间通信协议？怎么交互？

使用xx解决xx问题，遇到xx问题以及

解决方法？

哪些开源组件？组件选型？

多突出个人对项目的贡献度。

针对复杂的需求，系统设计了哪些方案？有什么技术难点？怎么解决？

遇到问题，如何排查（思路）？

项目的性能问题，怎么优化？

多突出项目中亮点、请求量、数据量。 遇到问题排查问题

如果项目无高请求，抛出假设，与面试官聊聊。 如直播项目大流量。
多极缓存应对热点端热点请求。

几万/几十万同时发言，优化消息延迟。

压测评估目前系统承载流量的能力，如何制定扩容？

平时工作：更多关注系统性能指标，参与排查性能问题与解决诡异 Bug 的过程，与同事讨论系统优化思路。

多关注业界的技术发展，技术可以用在项目中哪些应用场景？

② 系统架构：每秒 1 万次请求的系统要做服务化拆分吗？

微服务拆分，淘宝“玉彩票”项目。

一体化缺陷：① 数据库连接数（最多 16384）达到上限。

应用服务器扩容，使连接数大增。

③ 增加研发成本，抑制研发效率的提升。

功能间耦合严重，模块间相互依赖，一部分影响整体。

④ 系统运维。

上线的编译、单元测试、打包、上传、耗时长（长时间，大项目）

业务/功能的拆分，某些模块请求量很大，容易成为瓶颈。

如用户单，用户逻辑可以部署成单独的服务，其包括用户池的其它池都连接这个服务访问用户信息。

按照业务做横向拆分，解决数据库层面的扩展性问题。

可以将与业务无关的公用服务抽取出来，下沉成单独的服务。

思：你在开发运维过程中遇到哪些问题促使你走上微服务化的道路呢？

③ 微服务风靡后，微服务化和系统拆分要如何改造？

服务拆分遵循原则？

服务边界如何确定？服务的粒度？

服务化后会遇到的问题？如何解决？

原则：① 单一服务内部功能高内聚低耦合。

② 服务拆分粒度，先粗略拆分再逐渐细化。

③ 拆分过程，要尽量避免影响产品的正常功能。

迭代：

A. 优先剥离边界服务，如短信服务。

（非核心服务）

④ 服务存在依赖，优先拆分被依赖服务。

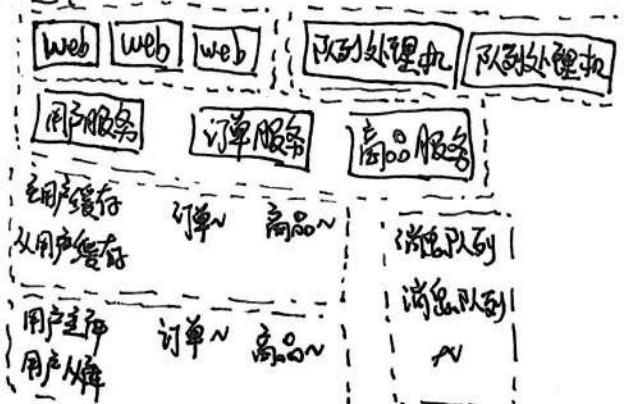
接口的参数类型最好是封装类。

分布式环境协调，调多个服务，引入复杂度：

① 引入了服务注册中心（以了解服务的位置以及服务管理）。

② 服务依赖复杂，一个服务某个问题影响其它服务，直到整个系统。

服务治理体系：采用熔断、降级、限流、超时控制的方法，使问题被限制在单一服务中。



③ 一条请求涉及多个服务，出故障的追踪，引入分布式追踪工具，以及更细致的服务端监控报表。

单一请求性能瓶颈分析

关注依赖服务 / 资源宏观性能表现

思：在微服务拆分后遇到哪些问题？如何解决？

中间件扩容理解：部署运行（感性认识）文档基本原理 / 架构设计部分，必要时阅读源码。
每个模块的小团队不宜过大（6-8），没做好微服务化准备，折中优先的工程拆分。

② RPC 框架：10万 QPS 下如何实现毫秒级的服务调用。拆分成单独包项目中引用。

拆分后拆问题：① 服务间跨网络通信的问题
② 服务如何治理？

问题① 解决的核心组件 RPC 框架。

RPC 框架承载大请求（如10万次/s），
A. 选择合适的网络模型，针对性地调整网络参数优化网络传输性能。
B. 远程过程调用。
C. 选择合适的序列化方式，提升封包/解包的性能。

RPC 可优化：网络传输 / 序列化。

网络适配：选择高性能 I/O 模型。

单次 I/O 请求分为两阶段：① 等待资源阶段 ② 使用资源阶段。

5 种常见的 I/O 模型：A. 同步阻塞 I/O B. 同步非阻塞 I/O C. 同步多路 I/O 复用

D. 信号驱动 I/O E. 异步 I/O

A. 一直等待，直到资源可用才使用资源。

B. 在被通知资源可用时之前，可干别的事，收到通知则使用资源。

C. 同时等待多个资源可用，在干别的事情的时间隙查看哪个资源可用则使用该资源。

D. 在等待资源可用时干别的事情，并在期间回来检查，如果可用则使用资源。

E. 在资源可用时自动使用资源。

使用最广泛的为多路 I/O 复用（linux 的 select, epoll, Java 的 Netty 默认以使用）

网络参数调优：如 tcp_nodelay 关闭 Nagle's 算法。服务端 DelayAck。

序列化：考虑因素：① 性能（时间 / 空间）② 跨语言兼容性 ③ 扩展性。

首选方案：JSON, Thrift / protobuf (需要 IDL 文件)

性能要求不高，传输数据占用带宽不大 → 选 JSON。

性能要求高， Thrift / protobuf。

一体化解决方案（包 RPC 框架） Thrift：

存储场景，数据占用空间较大，可使用 protobuf 带模 JSON。

阅读成熟的 RPC 框架源代码。（阿里 Dubbo, 微博 Motan）

学习编码与设计技巧 → Dubbo → RPC 的抽象。

远程调用的参数。

接收 / 发送缓冲区大小

客户端连接请求缓冲队列大小 (back log)

思：RPC 框架使用过程，遇到哪些问题，如何排查与解决？

④ 服务中心；分布式系统如何寻址？

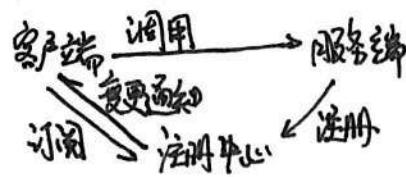
服务注册与发现。RPC 客户端寻服务端部署地址。

地址写在客户端配置。在扩容 / 服务器故障等发生变更的情况下，所有客户端配置要修改并重启所有客户端。

服务端重启也造成了客户端发去的请求造成慢请求甚至失败。

服务中心组件：Zookeeper, kubernetes 的 ETCI，阿里微服务的 Nacos, SpringCloud 的 Eureka。

功能：提供地址解析服务；有节点变更通知（推送）给所有客户端。



优雅关闭服务端：先在注册中心移除服务端，待服务端没有流量后才停止。
 ↓减少对客户端影响
 服务端、服务状态管理。

故障解决思路：

A. 主动探测：注册中心探测服务端某个端口是否可用。

不可用则删除服务器地址

两个问题：服务端要开启某个端口，而端口可能已经在一些机器上被使用
 探测成本高，而且会有延迟才能知道服务不可用。

B. 心跳模式：（检测服务存活，可以考虑使用，通用方法）

服务端定时向注册中心发送心跳包。如果在一定时间内没有收到节点的心跳包则认为服务不可用。（记录每个节点的最后心跳时间，后台定期检查所有节点的时间差值是否超过阈值。）

节点过快摘除，在摘除节点超过一定的比例之后，停止上摘除。（避免误删所有节点等）
 Bug造成

“通知风暴”推送大量消息。解决思路

A. 控制注册中心管理集群规模（指标：注册中心服务器峰值带宽）

B. 扩容注册中心节点

C. 注册中心使用方式，变更一个节点，只通知这个节点的信息。

D. 后端注册中心加入保护策略，通知量达到一个阈值停止通知。

治理，发现是服务治理(service governance)的一环。

服务治理和发现、服务的监控、熔断与引流、分布式追踪、负载均衡

问：你们在服务化框架中使用什么注册中心？基于什么考虑来做选型？

④ 分布式 Trace：横跨几十个分布式组件的慢请求要如何排查。

一体化中的慢请求排查：

A. 打印每步骤耗时情况到日志中，其中增加 requestid 等标识以记录日志的来源请求。

B. 慢请求通常是由子跨网的调用（数据库、缓存、第三方服务），针对这些调用客户端的类做切面手编程

切面编程(AOP)，在不修改源代码的前提下给应用程序添加功能（如监控、日志添加）

静态代理（对运行时性能基本无影响，推荐）AspectJ。

动态代理（运行时作切面注入，性能相对更差）Spring AOP。

避免企业在代码中大量加入日志代码。

C. 日志量大，减少日志量，可根据 requestid 来确定采样。

D. 多服务器节点，无法确认请求落在哪个节点上，难以查找归属。

对代码侵入性较小

日志、收集、集中存储（如 Elasticsearch）ELK那一套，消息队列传输。

分布式的 Trace。

采用 TraceId + spanId

RequestId 记录每一次RPC调用。

spanid 的计算由调用方计算后传给被调用方。

如调用方 spanid 为 1，被调用方可能为 1.1 / 1.2 等。

日志写入时通过上下文获取，后一同写入日志。

影响磁盘 I/O / 网络 I/O

自研 Trace 组件可能设置一个开关，开源组件可以设置一个较低的采样率。

在客户端生成 requestid，请求业务接口时一同传递，则可以把客户端的日志体系也整合进来。（客户端？）

开源解决方案 Zipkin, Jaeger。

问：项目中是否接入过分布式追踪系统？帮助你排查了哪些问题？

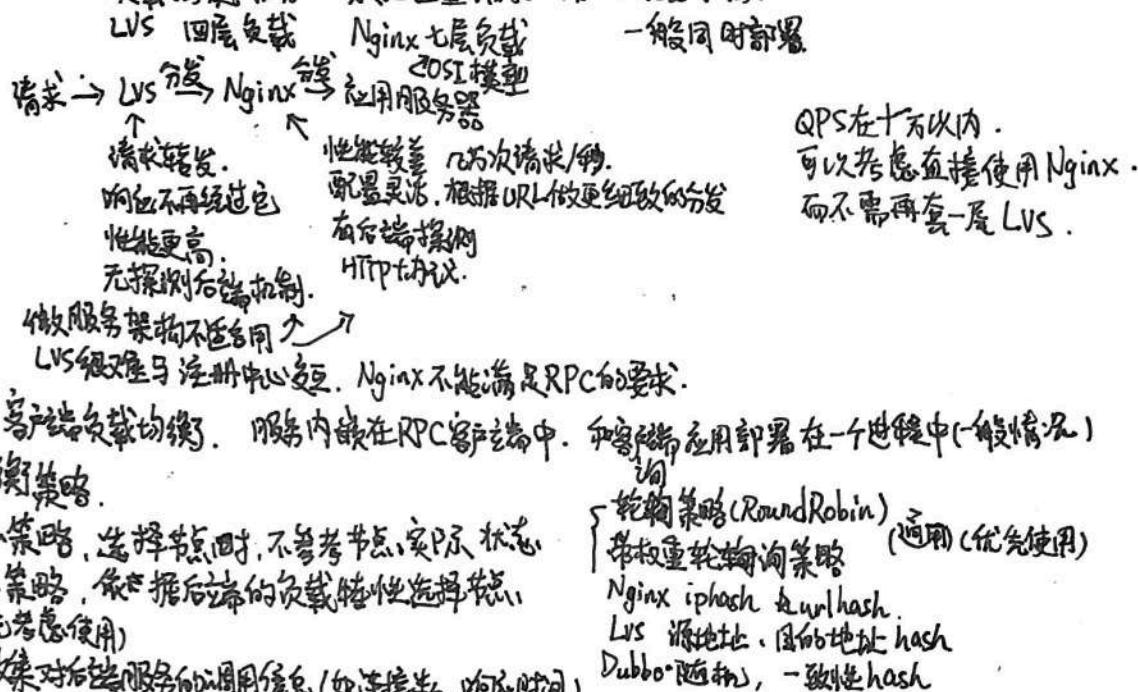
② 负载均衡：怎样提升系统的横向扩展能力？

MySQL 多从库，将查询请求分配到多个从库上（负载均衡，一般为 DNS 服务器）

Nginx，将请求分发到多个后端的业务服务器上。

负载均衡：将请求“均衡”地分配到多个处理节点上，可以对请求方屏蔽后端细节实现灵活扩容。

分类：代理类负载均衡服务：承担全量请求，高性能要求高。



节点故障检测。

服务节点的检测，注册中心处理（心跳包）

Nginx 会检测，nginx-upstream-check-module。Nginx 向后端服务发送请求并根据响应进行判断。

响应码可在配置中心中配置（33 讲）

web 服务的开启关闭。在启动时状态码设置为 500，服务初始化完成后设置为 200，在关闭时先设置状态码为 500，待无流量，无任务后再关闭。（先关闭后重启）

思：在使用负载均衡服务与组件时，遇过哪些问题？有哪些注意的点？

② API 网关，系统的门面要如何做呢？

API 出现大量的爬虫等不正常流量，该如何增加限制策略、跨语言支持？

引入 API 网关：（架构模式）

将服务共有的功能整合在一起，独立部署为单独一层，解决服务治理问题。

分类：入口网关、出口网关。

入口网关：A 提供客户端统一的接入地址，API 网关将用户的动请求动态路由到不同业务，做协议转换工作。

多层微服务的差异。

B. 嵌入服务治理策略，服务熔断、降级、流量控制、分流等。

C. 客户端认证与授权。

D. 黑白名单。

E. 日志记录、RequestID 生成。

出口网关：对调用外部的 API 做统一的认证、授权、审计以及访问控制。

如何实现 API 网关：

① 美化性能：工作模型。

② 扩展性：“热插拔”执行链路上的逻辑。

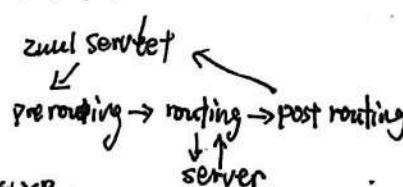
“责任链”连接模式。

③ 使用线程池，为了不让单一服务影响池内所有线程：

A. 为服务划分不同的线程池。

B. 限制一个服务的线程池数量，了可再结合。

开源 API 网关：Kong (Nginx) Zuerl (Spring Cloud), Tyk (Go)

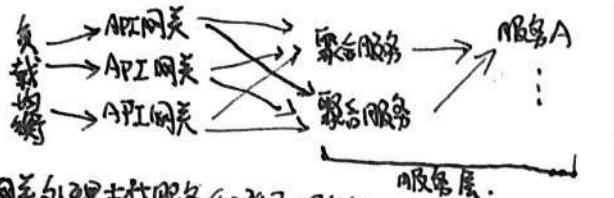


系统中引入API网关：

- A. 网络层接口数据聚合.
- B. HTTP请求转RPC协议. 前端流量限制.

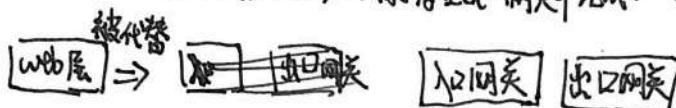
将API网关从Web层独立，协议转换/限流/黑白名单. 使用网关处理. (入口网关层)
服务接口的聚合操作:

- A. 抽立一组业务网关分散服务聚合. 超时控制.
- B. 抽取独立的服务层(聚合服务层) (依赖倾向延伸)



出口网关处理支付服务和登录服务等.

第三方的支付数据加密/签名. 第三方接口调用将在出口网关中完成. ← 使用支付服务则调用出口网关中的统一接口即可.



思：是否使用API网关？在使用API网关时，遇到什么样的问题？

(28) 多机房部署：跨地域的分布式系统如何做？

单机房系统可用性受机房可用性的制约.

多机房部署：在不同IDC机房中，部署多套服务. 这些服务共享同一份业务数据.

跨机房的数据传输：要避免出现跨机房访问数据造成性能衰减问题.

避免数据延迟对接口响应时间的影响

逐步迭代多机房部署方案：A. 同城双活 (机房级强一致性容灾)

在一个机房中部署主库. 各个机房部署从库. 与主库不同机房的从库可在主库故障后做主从切换.

在各个机房部署缓存. 服务读各自机房的缓存与数据库. 写的操作可更新各个机房中的缓存以保持数据一致性. 数据库的写入是写入主库中.

RPC调用中心. 不同机房的服务客户端订阅各自机房的服务(分组). 调用各自机房的服务.

其他服务也尽量只访问当前机房中的节点.

存在跨机房写数据. 请求量不高可容忍.

不要轻易尝试. 数据写入只保证在本机房. 数据写入只写当前机房的数据存储服务. 再采取数据同步.

要高频率要求. A. 基于存储系统的主从复制. MySQL和Redis. ← MySQL, Redis } 双机房使用.
B. 基于消息队列. 另外的机房消费并执行业务逻辑. } 两者结合使用.

读写在同机房内 → 缓存数据, HBase数据.

思：遇到高并发下才考虑壁使用多机房部署？实施中踩到哪些坑？

(29) Service Mesh：如何屏蔽服务化系统的服务治理细节？

- A. 通信协议对多语言友好. 选择合适的序列化方式.
- B. 无法使用之前积累的服务治理策略.

将客户端的服务治理细节单独拆分出来单独部署. “关注点分离”

Service Mesh：处理服务间的通信：

RPC客户端 → sidecar → sidecar → RPC服务器端

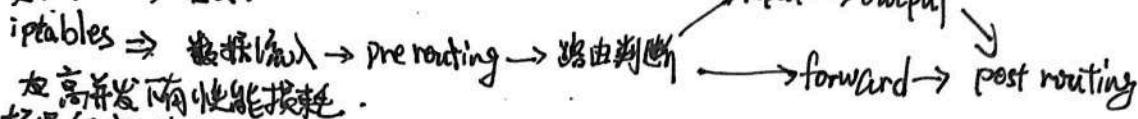
常用手段 → istio. 每次请求都经过控制平面(跨网路调用)

治理策略做到数据平面. 控制平面只下发策略. 改善性能.

将流量转发到Sidecar中。

无感知地引入SideCar网络代理。

A. 使用iptables转发。



B. 轻量级客户端。(推荐)

C. 使用 Cilium Socket 层面请求转发。(探索阶段)

Service Mesh 框架: istio, Linkerd, SOFAMesh
问: 是否使用过 Service Mesh 解决过跨语言的服务治理问题呢?

(30) 维护篇: 给系统加上监控要怎么做?

监控指标的选择: 分布式系统监控: 一般需要的4个指标: 延迟, 通信量, 错误, 饱和度。

延迟: 请求的响应时间 (接口, 数据库, 缓存等)

通信量: 单位时间内请求数量的大小 (访问第三方服务, 消息队列等)

错误: 服务的错误与业务的错误。

饱和度: 服务/资源到达上限的程度。

组件	延迟	通信量	错误	饱和度	其它
Web 服务	响应时间	请求数量	4xx, 5xx 业务错误	Tomcat 线程池 线程数, 活跃 线程数	\
数据库	响应时间 慢请求SQL	请求数量	请求超时 错误数量	连接数	主从延迟
缓存	响应时间, 慢请求	请求数量	同上	连接数	命中率
消息队列	响应时间	请求数量	同上	\	消息堆积
JVM	GC 时间	GC 频率	\	内存区域大小	\
依赖服务	响应时间	请求数量	请求超时 错误数量	\	\

如何采集指标:

A. Agent: 数据源服务器上部署, 获取信息后发送给监控服务器。

B. 埋点 (在代码中): 注意发送监控服务器的请求数量
可以作一段时间内的汇总并发送。

C. 日志, 采集工具: Apache Flume, Fluentd, Filebeat

数据处理与存储。

消息队列存储数据 (削峰填谷), 而后队列处理器进行处理

写入 Elasticsearch, Kibana 展示

流式处理器 (如 spark, storm)

解析数据格式 (日志格式)

对数据聚合运算。

存储在时间序列数据库中 (InfluxDB, OpenTSDB, Graphite)

通过 Grafana 连接时间数据库, 将数据绘制成为报表。

资源报表 (Agent 收集资源运行情况数据, 找出资源出现的问题)

问: 你的服务端监控系统如何搭建? 都有哪些监控报表和监控项?

① 应用性能管理：用户的使用体验应该如何监控？

客户端（用户端）问题以及宽带（网络）的问题。

应用性能管理（Application Performance Management）APM 核心关注点：终端用户的使用体验。

端到端的监控体系：

采集：类似Agent，植入SDK负责数据采集，并定期通过接口发送给服务端。（APM通用服务）

指标：系统部分：数据协议版本、消息头、端消息体、业务消息体。

消息头，包括应用标识（appkey）、消息生成时间戳、签名、加密密钥。

端消息体，客户端版本号、SDK版本号、IDFA、IDFV、IMEI、机器型号、渠道号、运营商。

网络类型、操作系统类型、国家、地区、经纬度等。信息敏感、隐私加密。

业务消息体：真正要采集的数据，也需要加密。

处理与存储：消息的解密（暗）

对得到的数据做解析，得到具体数据，写入消息队列。消费消息，一份写入ElasticSearch → 同服务器端。
一份写到统计平台。

需要监控用户哪些信息？

① 网络数据，大部分问题出现在网络问题上。重要数据！

使用埋点方式，将每个步骤采集时记录，以及错误的发生。

请求→等待时间（正向到本地IP队列，真正处理前的等待），DNS时间，TCP握手时间，SSL时间，发送时间

首包时间（收到第一个响应包时间），包括收时间。

价值：准确、真实、实时，及时反映用户体验。

性能优化的方向。

帮助监控CDN链路质量。

② 异常日志数据（登录失败、下单失败、浏览商品信息失败、评论列表加载失败、无法评分留言）。

思：在工作中，你（团队）如何通过监控，发现客户端上的问题？

② 压力测试：怎样设计全链路压力测试平台？

压力测试：①最好使用线上数据和线上环境 ②要使用线上流量，不要使用模拟的请求。

③不要从一台服务器上发起流量，尽量是各个CDN节点或各机房服务器。

高并发大流量下进行测试，通过观察系统峰值负载的表现，找到系统存在的隐患。

针对整个链路（摄入层/后端服务/数据库/缓存/消息队列/中间件/依赖的第三方服务系统的资源）

统一的压力测试 → 全链路压测

压测作为系统稳定性保障的常规手段，周期性进行。

全链路压测，要联动DBA、运维等多个团队，成本高，甚至会影响线上环境。

→ 搭建自动化全链路压测平台降低成本与风险。

如何搭建：关键点：① 流量的隔离，区分压测流量与正式流量。

② 风险控制。

模块：① 流量构造/产生模块

② 压力测试数据隔离模块。

③ 系统入侵检测模块，压测流量干预模块。

压测数据采集：业务高峰期流量，过滤无效流量存储在NoSQL或者存储服务中（流量数据工厂）

数据即分发多个压测节点上。

数据方式：按照均衡IP地址访问日志（不建议），GoReplay（捕捉与回放）。

在数据捕获时，增加压测的标记项，再写入流量工厂中。（请求头中添加）

数据隔离：针对读数据请求（下行流量）针对不能压测的服务/组件做Mock或特殊处理。

针对写数据请求（上行流量）将写入子节点。

MySQL 在同一个实例创建相同表结构的schema，并导入线上数据。

Redis 统一增加一个前置代理。

Es 将对这部分数据，放在另外单独的索引表中。

压测实施： 实施前，设立压测目标，如整体系统 QPS 20 万/秒。
压测流量按照一定步长增加，如出现瓶颈，回退到上次压测的 QPS，保证服务的稳定性再扩容以及继续后续的压测。
开发流量监控组件，预设性能阈值，达到阈值时，通知压测流量下发组件减少压测流量。
发送警报，在解决问题后再次继续执行压测。

- 价值：**
- 帮助发现系统中可能出现性能瓶颈，方便提前准备预案应对。
 - 做容量评估，提供数据上的支撑。
 - 做预案演练。

思：你的系统压测是如何进行的？过程中发现哪些性能瓶颈点？

(3) 配置管理：成千上万的配置项要如何管理？

对配置项进行管理。A. 通过配置文件管理；B. 使用配置中心管理。mysql Zookeeper

需要重启生效。

通常会把配置文件存储的目录标准化为固定的目录。
并想使用 Git 来管理配置。

开源组件：Apollo, Disconf, Qconf, Spring Cloud Config。
推荐，支持不同环境/集群的配置，完善管理功能。
支持灰度发布，更改热发布功能。

自研配置中心，关键点：

配置信息存储，主要是信息存储与读取。存储组件：MySQL, Zookeeper, Redis, Etcd, etc...

规范化存储组件中的存储结构：全局机器节点配置，配置项存储路径。

new3{project3}{service3}{version3}{module3}.

变更推送：① 应用中心向配置服务器轮播查询配置项，可以给配置项增加一个 MD5 记录项。
在 MD5 不一致时，才实际拉取配置项更新应用中的配置，减少配置中心带宽占用。【更简单】

② 弹性推送。配置中心要记录每个节点的配置项连接，只要配置项发生变更，就向使用了该配置项的节点推送变更通知。【更实时】

配置中心高可用：可用性重要程度大于性能。

配置中心客户端两级缓存
① 内存缓存 ② 文件缓存（灾备，配置中心宕机，应用启动时使用文件
（降低与配置中心的交互频率）
缓存中的配置信息。）
降级方案。

配置的存储是分级的（公共配置 / 个性配置）

如果项目中使用文件方式管理配置信息，只需要将需要动态调整的配置（如启动时间）迁移至配置中心即可。

降低迁移成本

思：你项目中配置管理的方式是怎样的呢？

(4) 降级熔断：如何屏蔽非核心系统故障的影响。

响应慢导致调用者持有资源无法释放，最终拖垮整体。

A. 依赖资源或服务不可用，导致整体宕机。

B. 超过系统承载能力的流量到来而出现拒绝服务情况。（主要思路：限流）

雪崩的发生：一个服务响应缓慢，调用方等它的返回后占用资源，大量请求继续进来，但都处于阻塞状态直到超时的发生。

资源占满后无法处理其它请求，它的上级调用方也同理，直到整个系统无法再处理任何请求。

解决思路：切换到服务的响应正常，切断调用它的服务与它之间的连接，让调用方快速得到调用失败。

释放持有的请求资源。降级和熔断机制。

熔断机制：（断路器模式）

一定数量的请求失败，打开断路器；设置延时器，将打开的断路器设置为半打开状态；半打开状态下，尝试请求。

如果请求失败了，再次打开断路器，如果这个状态下成功了一定次数，则完全关闭断路器。

不仅仅在微服务间可用，在 Redis, memcached 中也可使用。

等资源。

降级机制：

站在整体系统负载的角度，放弃部分非核心功能或服务。有限的系统容错方式。

限流降级 / 开关降级。

也是降级的一
种

开关降级：在代码中预埋开关。关闭时正常服务，打开时则执行降级策略。

开关值保存在配置中心可动态变更而无需重启。

首先要区分核心与非核心服务，只能应用在非核心。具体业务可制定不同的降级策略。

降级开关要在业务低峰期做演练或在压测中演练，保证开关的可用性。

开关降级的实现策略：返回降级数据、降频、异步三种方案

读数据、轮询写数据

功能开关，在出现问题时快速回滚，减少问题持续时间

思：你的项目中制定了哪些降级预案？考虑点是什么？

(35) 流量控制：高并发系统中，我们如何操纵流量？

核心业务不能熔断以及降级（开关降级），只可使限流。

限流坑：A. 限流策略选择不当，导致限流效果不好。

B. 限流后整体性能有损耗

C. 实现单机限流但没有整体限流。

限流：限制到达系统的并发请求数量，保证部分请求正常响应，超出限制部分，通过拒绝服务保证整体系统的可用性。

一般部署在入口层（如API网关）或在RPC客户端中引入限流策略保证单个服务可用。

限流算法：

时间窗口：① 固定窗口，在一段固定时间内限制一定的请求量。

维度：② 滑动窗口。

空间复杂度↑

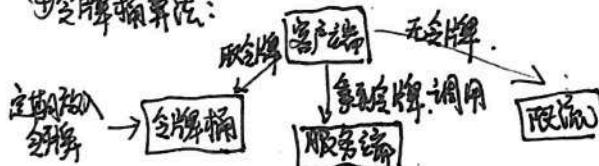
③ 漏桶算法。

流量进入漏桶，按照一定速率将它发向接收方。流量大于漏桶的承载能力将拒绝服务。

（流出流量变得平滑）

使用消息队列实现，会增加流量的响应时间

④ 令牌桶算法：



Guanfa 的限流方案使用该方案。

单机可用度量存储，令牌

分布式可用 Redis 存储令牌

为了减少性能的损耗，可以一次换取多个令牌。

（请求次数）

阈值的确定可以参考压测结果来设置。另外可以将阈值设置在配置中心以方便动态调整。

思：有没有在项目中使用过限流，保护系统？采用了什么样的算法？

(36) 面试：如何准备面试技术面呢？

基础扎实，有良好的编码习惯与能力（现场编码），
5年
1年左右
3年
架构能力

基础知识：算法、操作系统、网络方面、使用语言的特性

Leetcode.

大公司 ⇌ 新手 开拓眼界，有规范的研发流程，高并发大流量的系统。
or
小公司 ⇌ 老鸟 成熟的工具与运维体系等。

(37) 计数系统设计（一）面对海量数据的计数器要如何做？

计数的特点：A. 数据量巨大 B. 访问量大 C. 可用性、准确性要求高

刚开始的计数系统，本着 KISS 原则，简单易维护，用 MySQL 存储

数据量大增，使用分库分表 A. hash(weibo_id)

B. 根据工单解拆时间戳找（数据不同时期）

访问量变得巨大，MySQL无法承受。

Redis加速读请求，多节点提升可用性和性能。hash方式对数据分片。
会导致的数据不一致。

抛弃MySQL，使用Redis存储计数。

读数的写性能，降低读写压力。

使用消息队列，消费者可以将同一计数的多次更新合为一次更新。

如何降低计数系统的存储成本？

Redis存储key使用字符串类型，使用8（sdshdr数据结构长度）+19（8字节数字的长度）+1（'10'）=28个字节。
long类型只要8字节。

会增加Redis的内存消耗，kv存储只要8+4=12字节。

大数据组有如下信息，基于weiboid hash得出。

相同weiboid的计数存储在一起。

冷数据存到SSD中，读取时加载到一块cold cache区域。

3. 解决痛点作优化

支撑使用SSD存储冷数据。Pika, SSDB。（实现原理了解）

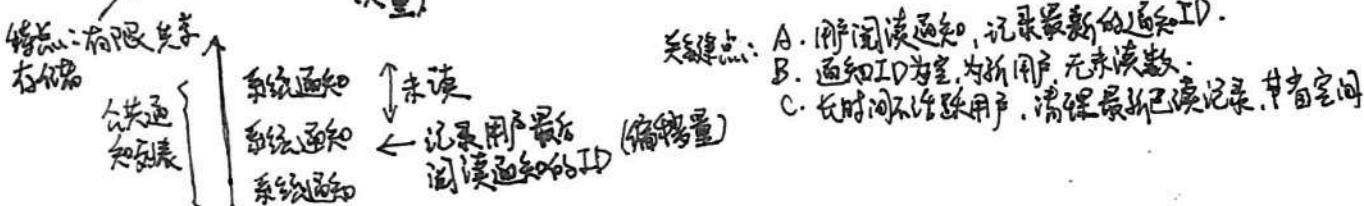
需要大量内存，数据有明显的热点，可使用SSD存储保存冷数据的方案。

思：你的系统中是否有大量计数的需求？如何设计方案来存储与读取？

③ 计数系统设计(2) 50万QPS下如何设计未读计数系统？

○ 系统通知未读计数不同用通用计数的方式。

系统批量为每个用户增加未读计数将是非常耗时的。
（大量）



减少未读数延迟，占用并节省内存。

○ 信息流的未读数：难点：
A. 基于关系系统，关注的人多，增加未读数据操作量大。

B. 指向为双向，请求量大。

C. 无法共享存储。

关注用户有限 方案：
A. 通用计数器记录每个人发布的博文数。

B. 记录一个人所有关注人的博文数，刷屏时将其所有关注人的博文数刷更新（快照）

C. 信息流未读数 = 总所有关注人的博文总数 - 快照中的数量。

(当前) (快照)

缺陷：关注关系更新不及时，将读数不准确。

全缓存存储，在满载时会删除部分数据，影响计数准确。

○ 缓存 提升系统性能，扩大并发量。

○ 围绕系统设计关键困难点想办法。

○ 合理分析场景，权衡可放弃的点（如不活跃用户的未读数被清空等）提升系统性能
减少带宽

○ 一对多的未读数可使用通用计数方案。

○ 分析需求场景，（读数据量级、请求量级、寻找可利用特点）再制订方案。

思：在实际项目中，你的系统有哪些未读计数场景？如何实现方案来实现未读计数？

③ 信息流设计(一): 通用信息流系统的推模式要如何做?

信息流系统倒序 TimeLine (时间线) 美滋点:

- A. 延迟
 - B. 高并发访问支撑
 - C. 挑取性能 (聚合数据, 需要多次查询多次存储)
- (主体模块访问量大)

基于推模模式与拉模式的思路

推模式: 用户发送消息后, 推送给粉丝.

(发件箱) $\times 1$ (收件箱) $\times n$

A. 多线程写入并行写入

读使用缓存, 性能次要.

流量大量写入, 反向缓慢延迟高.

B. 尽量保证数据快写入性能, 采用写入性能更好的引擎.

存储量占用大, 解决思路: 定期清理用户收件箱数据 (如 TokuDB 分形树 / 堆缓存).

扩展性问题: 如关注分组, 导致每个单用户有多个收件箱.

删除消息, 取消关注逻辑复杂: 在读取时增加是否有关注与消息是否删除逻辑.

避免大量的写操作 (不删除收件箱)

推模式适合场景: 粉丝数有限制的场景. (不适合 Weibo)

问: 是否设计过信息流系统? 你如何设计解决推模式推拉结合的延迟问题?

④ 信息流设计(二): 通用信息流系统的拉模式要如何做?

拉模式: 主动拉取关注用户的微博, 并倒序排序/聚合.

索取自发布用户的发件箱.

关注数 1. 解决推模式的推延延迟问题... 存储成本降低, 功能扩展性↑

有上限 拉模式下的问题:

A. 查询聚合成本高. → 缓存 只存储最近5天内发布的微博ID.

B. 缓存节点带宽成本高. 多个节点进行缓存, 聚合时从这些节点并行查询微博ID.

缓存副本降低缓存流量 在应用服务器内存中排序.

推拉模式: 大V用户只推送给活跃用户 (实时)

判断粉丝数 大V用户维护一个活跃用户列表.
有操作的用户将它加入关注者的活跃列表中.

非活跃用户通过异步的方式插入收件箱中.

适合中等体量项目.

问: 项目中是否有使用拉模式? 后来设计过程中遇到过哪些问题? 如何解决?

结束语: 算法导论 TCP/IP 协议.

项目孵化器 Dubbo Spring

公众号文章.

对突发状况深入总结和思考.

问题根本原因如何避免同类问题.

问题最优解. (如何与团队交流)

知识积累. 深入思考

在实践中提升

思考, 内涵, 知识储量.

梯度进阶 (目标)

速度.

高度. 深度 (深入理解)

基础.

精度

慢思之
审问之
细问他问

明辨之

笃行之

高并发下如何发现和排查问题.

如何及时发现问题： 监控，压测

客户端的监控（引讲）重视，如用户端丢包 SOH 在服务端监控中无法查察觉 -

压测： 如压测中发现在一定流量时，Redis 响应变慢，通过监控发现 Redis 的 CPU 占用率大幅上升接近 100%，期间大量逐出 key（逐出会使 CPU 占用上升），并发现逐出前有大量的连接数上升。

内存使用量接近 100%，连接数大量增长，造成逐出大量的 key。

查看端口日志，发现连接数上涨前有一些慢请求，导致大量的重新连接（新连接），旧连接导致关闭。短时间内连接数大增。

tcpdump 抓包发现 Redis 响应时间大幅升高，排查 Redis 实现逻辑，是由于 Redis 3.0 的 jemalloc 释放内存造成偶发的卡顿。短时间内消耗掉 Redis 的所有请求队列，响应时间升高。

监控、错误日志、tcpdump 抓包，Redis 原理/代码查阅 → 途径。

排查问题方法：

排查困难原因所在：容易看到问题外在表现，但根本原因的推断需要分析能力，归内思维能力和经验积累。

主要手段：监控和日志，常见工具使用。（CPU、网络查看等）

内存泄漏：Java 堆内 jmap，top，Java → jstack，pidstat，vmstat，mpstat，Perf
堆外，pmap，GDB。

思：开发维护过程中遇到什么诡异的问题，如何发现以及排查？

我们如何准备抵抗流量峰值？

为什么要考虑抵抗流量峰值？

A. 未雨绸缪，在业务上的成功可能带来大量流量（DAU），如何准备？

B. 为系统发展做准备。

在活动、节日等流量高峰期之前，做预案来应对峰值流量。

确定系统的瓶颈点：

A. 流理系统的调用链路，如得到部署图：

B. 逐一观察数据流转链路上是否存在瓶颈点。 小心评估，避免遗漏

LVS 入口/出口带宽

系统出现问题时快速定位瓶颈点（LVS、Nginx、服务器）

C. 关注链路上的网络带宽和线路的稳定性。

全链路压测为主要途径

制定抵御高并发流量的预案。

不破坏系统架构做大规模调整前提下主要有：

① 切流量到另外的机房，前提是多机房部署

（依赖原机房以及新机房连接）

A. 全部流量先经过原有机房的入口，并通过负载均衡层转发到另外的机房

B. 域名解析切换（切换不及时，DNS 缓存）

② 扩容，横向服务器扩容，组件的扩容（增加数据库从库，缓存多副本组）

③ 专线的扩容。 ↗ 扩容资源要提前准备好。

④ 降级。

⑤ 限流。

思：结合实际工作，你面对突增的流量冲击时如何制定预案？