大淘宝技术
TAO TECHNOLOGY

服务端开发与面试知识手册

大淘宝技术工程师个人知识整理与经验总结

编者
大淘宝技术开发工程师
刘苏宏(淘苏)

内容简介：

- 主体内容覆盖「JAVA 体系」和「架构能力」两大部分
- 包含网络和操作系统基础；JVM、多线程等中间件；Spring、Netty 主流框架的重点知识等，以及结合实践给出的各类难点问题和解决方案等。

出品方：大淘宝技术品牌

前言

编者： 大淘宝技术开发工程师 / 刘苏宏(淘苏)

淘苏（花名）目前是大淘宝技术的一名开发工程师。从国企跳槽来到互联网，【职业规划】是他被问得最多，也思考得最多的问题。

回忆国企的三到五年时间，他完成了最初始的技术和经验的积累。接下来的职业生涯规划里，他希望成为一名优秀的技术专家，想更深入的培养自己技术攻坚能力，想去更高的平台学习大型项目的功能拆分实践，塑造自身的技术品牌，于是，他来到了大淘宝技术。

基于这个目标，在过去的两年里，他系统学习了各种中间件的底层原理及代码实现，更深层次的掌握高可用、高并发场景的通用解决方案，努力锻炼自己业务的抽象、架构能力、加深对分布式服务设计原则的理解等。学习的同时，不断保持技术的敏锐性，不断将优秀的实践应用到自身的项目中，有意识的补足欠缺的知识理论，对业务的涉及到的亮点工作进行抽象和升华。

在通往梦想的努力道路上，他总结出纯手工的128362字，汇集成这本300+页的《服务端开发与面试知识手册》，内容分为JAVA 体系和架构能力两大部分，

- JAVA体系包含了网络和操作系统基础，JVM、多线程、Mysql、Redis、Kafka等中间件，框架以介绍Spring为主，讲解框架理念与应用场景，也包括一些重点机制的实现方式。重点讲解了分布式系统经常容易出现的难点问题和解决方案。新技术方面：Java 8~11 新特性，G1、ZGC 垃圾回收算法、最新网络协议 HTTP2；知识深度方面：内存屏障、指令重排，JIT 编译器、逃逸分析等。这部分的讲解属于亮点知识。
- 架构能力，主要以经典架构的常考点为主，包括 Spring、Netty 等主流框架的重点知识。大数据场景中的ES，云原生场景的中Docker和K8S的应用。同时结合自身具体的项目，讲好如何使用缓存、队列、数据库来优化已有项目。同时整理了LeetCode高频算法题的python模板，这套模板曾帮助他在三个月左右的时间，进入Leet-Code前两千名。

为了知识结构的完整性和系统性，他牺牲了半年的假期时间，查阅了大量学习资料，加上自己的经验总结和技术理解，在不断蜕变的过程中，延迟获得满足。

淘苏的想法很朴素，技术能力的学习进阶很苦，但是在正确道路上的坚持很酷，希望他的这份精神和手册一样，可

以给同行的技术同学一份参考，作为日常基础知识的查漏补缺，或者个人系统能力的总结思考，一起进步。

PS：
本书内容为工程师个人学习知识积累和经验总结，部分内容来源于互联网，仅作为知识梳理与分享，非盈利广告，无商业用途。结尾页附淘苏的学习途径和参考书籍资料，共享给同行学习，也感谢所有知识的分享者，希望大家都能在实现梦想的道路上越走越好。⛽

扫一扫，关注公众号【大淘宝技术】
了解更多大淘宝技术干货沉淀

大淘宝技术是阿里巴巴集团中国数字商业板块的技术王牌军，支撑淘宝、天猫等核心电商业务。依托大淘宝丰富的业务形态和海量的用户，大淘宝技术通过持续的技术创新和突破，不断探索和衍生颠覆性互联网新体验技术，以更加智能、友好、普惠的科技帮助商家更好的经营，让用户享受更好的消费体验。

CONTENTS
目录

第一部分
关于Java语言的方方面面

01
基础篇

网络基础 ---------- 02

操作系统基础 ---------- 13

Java基础 ---------- 18

面试题 ---------- 28

02
JVM篇

JVM内存划分 ---------- 30

JVM类加载过程 ---------- 34

JVM垃圾回收 ---------- 36

线上故障排查 ---------- 43

03
多线程篇

线程调度 ---- 53

线程池 ---- 56

线程安全 ---- 60

内存模型 ---- 66

04
MySQL篇

WhyMysql？ ---- 69

事务 ---- 77

索引 ---- 81

SQL查询 ---- 83

集群 ---- 86

面试题 ---- 88

线上故障及优化 ---- 90

05
Redis篇

WhyRedis ---- 94

Redis底层 ---- 100

Redis可用性 ---- 102

Redis七大经典问题 ---- 107

Redis分区容错 ---- 109

Redis实践 ---- 116

06
Kafka篇

Why kafka ---- 118
What Kafka ---- 120
How Kafka ---- 122
生产消费基本流程 ---- 123
一致性 ---- 125
可用性 ---- 126
面试题 ---- 127

07
Spring篇

设计思想&Beans ---- 130
Spring注解 ---- 135
Spring源码阅读 ---- 139

08
SpringCloud篇

Why SpringCloud ---- 140
Spring Boot ---- 141
GateWay / Zuul ---- 142
Eureka / Zookeeper ---- 143
Feign / Ribbon ---- 146
Hystrix / Sentinel ---- 149
Config / Nacos ---- 151
Bus / Stream ---- 152
Sleuth / Zipkin ---- 152
安全认证 ---- 153

灰度发布 155
多版本隔离 156

09 分布式篇

发展历程 158
CPA 159
一致性 159
可用性 163
分区容错性 164
分布式事物 165
面试题 167

第二部分 关于DESIGN的方方面面

01 ES篇

概述 169
基本概念 171
高级特性 172
实战 173

02 Docker&K8S篇

Why Docker 176
核心概念 177
基本操作 178
实战 179

03 Netty篇

核心组件 180
网络传输 185
内存管理 190
高性能数据结构 199

04 LEETCODE

Python语法 207
背包模板 210
回溯模板 211
并查集模板 214
拓扑排序模板 215
单调栈模板 215
二分模板 216
动态规划模板 217
滑动窗口 220
前缀和 221

双指针 222
深度优先 223
广度优先 228
图论 229

05 实战算法篇

URL黑名单（布隆过滤器） 231
词频统计（分文件） 231
未出现的数（bit数组） 232
重复URL（分机器） 233
TOPK搜索（小根堆） 233
中位数（单向二分查找） 233
短域名系统（缓存） 234
海量评论入库（消息队列） 234
在线 / 并发用户数（Redis） 235
热门字符串（前缀树） 235
红包算法 236
手写快排 237
手写归并 239
手写堆排 240
手写单例 241
手写LRUcache 242
手写线程池 244
手写消费者生产者模式 246
手写阻塞队列 248
手写多线程交替打印ABC 250
交替打印FooBar 252

06
个人项目

一站到底 256
秒杀项目 261
即时通信 268
智慧社区 272

07
架构设计

社区系统的架构 284
商城系统–亿级商品如何存储 286
对账系统–分布式事务一致性 286
用户系统–多线程数据割接 287
秒杀系统场景设计 288
统计系统–海量计数 288
系统设计–微软 289
如何设计一个微博 290

08
领域模型落地

拆分微服务 293
关联微服务 293
微服务的落地 294
领域模型的意义 295
战略建模 296
相关名词 296

第一部分
关于Java语言的方方面面

- 基础篇
- JVM篇
- 多线程篇
- MySQL篇
- Redis篇
- Kafka篇
- Spring篇
- SpringCloud篇
- 分布式篇

基础篇

网络基础

• TCP三次握手

三次握手过程

客户端——发送带有SYN标志的数据包——服务端 一次握手 Client进入syn_sent状态
服务端——发送带有SYN/ACK标志的数据包——客户端 二次握手 服务端进入syn_rcvd
客户端——发送带有ACK标志的数据包——服务端 三次握手 连接就进入Established状态

为什么三次

主要是为了建立可靠的通信信道，保证客户端与服务端同时具备发送、接收数据的能力

为什么两次不行?

- 防止已失效的请求报文又传送到了服务端，建立了多余的链接，浪费资源
- 两次握手只能保证单向连接是畅通的。（为了实现可靠数据传输， TCP 协议的通信双方， 都必须维护一个序列号， 以标识发送出去的数据包中， 哪些是已经被对方收到的。 三次握手的过程即是通信双方相互告知序列号起始值， 并确认对方已经收到了序列号起始值的必经步骤；如果只是两次握手， 至多只有连接发起方的起始序列号能被确认， 另一方选择的序列号则得不到确认）

TCP四次挥手过程

四次挥手过程

客户端——发送带有FIN标志的数据包——服务端，关闭与服务端的连接 ，客户端进入FIN-WAIT-1状态
服务端收到这个 FIN，它发回一 个 ACK，确认序号为收到的序号加1，服务端就进入了CLOSE-WAIT状态
服务端——发送一个FIN数据包——客户端，关闭与客户端的连接，客户端就进入FIN-WAIT-2状态
客户端收到这个 FIN，发回 ACK 报文确认，并将确认序号设置为收到序号加1，TIME-WAIT状态

为什么四次

因为需要确保客户端与服务端的数据能够完成传输。

CLOSE-WAIT

这种状态的含义其实是表示在等待关闭

TIME-WAIT

为了解决网络的丢包和网络不稳定所带来的其他问题，确保连接方能在时间范围内，关闭自己的连接

如何查看TIME-WAIT状态的链接数量?

netstat -an |grep TIMEWAIT|wc -l 查看连接数等待timewait状态连接数

为什么会TIME-WAIT过多？解决方法是怎样的?

可能原因： 高并发短连接的TCP服务器上，当服务器处理完请求后立刻按照主动正常关闭连接

解决：负载均衡服务器；Web服务器首先关闭来自负载均衡服务器的连接

1. OSI与TCP/IP 模型

OSI七层：物理层、数据链路层、网络层、传输层、会话层、表示层、应用层

TCP/IP五层：物理层、数据链路层、网络层、传输层、应用层

2. 常见网络服务分层

应用层：HTTP、SMTP、DNS、FTP

传输层：TCP 、UDP

网络层：ICMP 、IP、路由器、防火墙

数据链路层：网卡、网桥、交换机

物理层：中继器、集线器

3. TCP与UDP区别及场景

类型	特点	性能	应用过场景	首部字节
TCP	面向连接、可靠、字节流	传输效率慢、所需资源多	文件、邮件传输	20-60
UDP	无连接、不可靠、数据报文段	传输效率快、所需资源少	语音、视频、直播	8个字节

基于TCP的协议：HTTP、FTP、SMTP

基于UDP的协议：RIP、DNS、SNMP

4. TCP滑动窗口，拥塞控制

TCP通过：应用数据分割、对数据包进行编号、校验和、流量控制、拥塞控制、超时重传等措施保证数据的可靠传输；

拥塞控制目的：为了防止过多的数据注入到网络中，避免网络中的路由器、链路过载

拥塞控制过程：TCP维护一个拥塞窗口，该窗口随着网络拥塞程度动态变化，通过慢开始、拥塞避免等算法减少网络拥塞的发生。

5. TCP粘包原因和解决方法

TCP粘包是指：发送方发送的若干包数据到接收方接收时粘成一包

发送方原因

TCP默认使用Nagle算法（主要作用：减少网络中报文段的数量）：

收集多个小分组，在一个确认到来时一起发送、导致发送方可能会出现粘包问题

接收方原因

TCP将接收到的数据包保存在接收缓存里，如果TCP接收数据包到缓存的速度大于应用程序从缓存中读取数据包的速度，多个包就会被缓存，应用程序就有可能读取到多个首尾相接粘到一起的包。

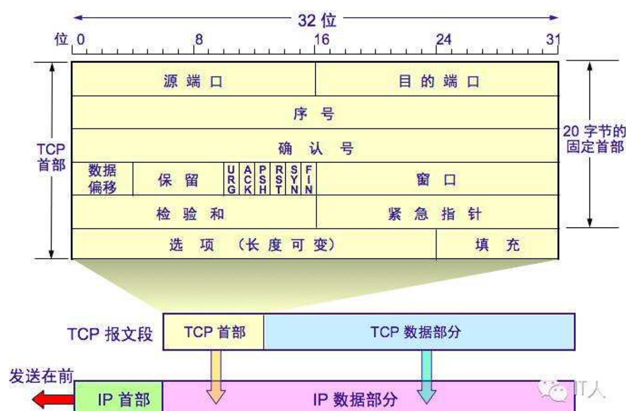
解决粘包问题

最本质原因在与接收对等方无法分辨消息与消息之间的边界在哪，通过使用某种方案给出边界，例如：

- 发送定长包。每个消息的大小都是一样的，接收方只要累计接收数据，直到数据等于一个定长的数值就将它作为一个消息。
- 包尾加上\r\n标记。FTP协议正是这么做的。但问题在于如果数据正文中也含有\r\n，则会误判为消息的边界。
- 包头加上包体长度。包头是定长的4个字节，说明了包体的长度。接收对等方先接收包体长度，依据包体长度来接收包体。

6. TCP、UDP报文格式

TCP报文格式：



源端口号和目的端口号

用于寻找发端和收端应用进程。这两个值加上ip首部源端ip地址和目的端ip地址唯一确定一个tcp连接。

序号字段

序号用来标识从T C P发端向T C P收端发送的数据字节流，它表示在这个报文段中的的第一个数据字节。如果将字节流看作在两个应用程序间的单向流动，则 T C P用序号对每个字节进行计数。序号是32 bit的无符号数，序号到达2^32−1后又从0开始。

当建立一个新的连接时，SYN标志变1。序号字段包含由这个主机选择的该连接的初始序号ISN（Initial Sequence Number）。该主机要发送数据的第一个字节序号为这个ISN加1，因为SYN标志消耗了一个序号

确认序号

既然每个传输的字节都被计数，确认序号包含发送确认的一端所期望收到的下一个序号。因此，确认序号应当是上次已成功收到数据字节序号加 1。只有ACK标志为 1时确认序号字段才有效。发送ACK无需任何代价，因为 32 bit的确认序号字段和A C K标志一样，总是T C P首部的一部分。因此，我们看到一旦一个连接建立起来，这个字段总是被设置，ACK标志也总是被设置为1。TCP为应用层提供全双工服务。这意味数据能在两个方向上独立地进行传输。因此，连接的每一端必须保持每个方向上的传输数据序号。

首部长度

首部长度给出首部中 32 bit字的数目。需要这个值是因为任选字段的长度是可变的。这个字段占4 bit，因此T C P最多有6 0字节的首部。然而，没有任选字段，正常的长度是 2 0字节。

标志字段

在T C P首部中有 6个标志比特。它们中的多个可同时被设置为1.　　URG紧急指针（u rgent pointer）有效　　ACK确认序号有效。　　PSH接收方应该尽快将这个报文段交给应用层。　　RST重建连接。　　SYN同步序号用来发起一个连接。这个标志和下一个标志将在第 1 8章介绍。　　FIN发端完成发送任务。

窗口大小

T C P的流量控制由连接的每一端通过声明的窗口大小来提供。窗口大小为字节数，起始于确认序号字段指明的值，这个值是接收端期望接收的字节。窗口大小是一个 16 bit字段，因而窗口大小最大为 65535字节。

检验和

检验和覆盖了整个的 T C P报文段：T C P首部和T C P数据。这是一个强制性的字段，一定是由发端计算和存储，并由收端进行验证。

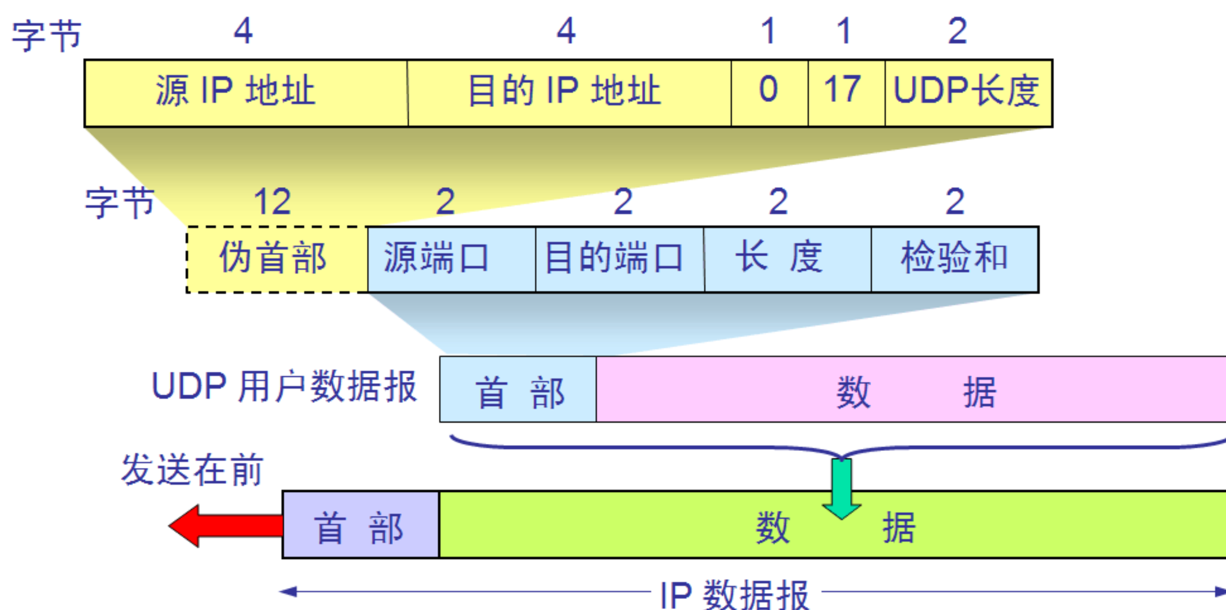
紧急指针

只有当URG标志置1时紧急指针才有效。紧急指针是一个正的偏移量，和序号字段中的值相加表示紧急数据最后一个字节的序号。 T C P的紧急方式是发送端向另一端发送紧急数据的一种方式。

选项

最常见的可选字段是最长报文大小，又称为 MSS (Maximum Segment Size)。每个连接方通常都在通信的第一个报文段（为建立连接而设置 S Y N标志的那个段）中指明这个选项。它指明本端所能接收的最大长度的报文段。

UDP报文格式



端口号

用来表示发送和接受进程。由于 I P层已经把I P数据报分配给T C P或U D P（根据I P首部中协议字段值），因此T C P端口号由T C P来查看，而 U D P端口号由UDP来查看。T C P端口号与UDP端口号是相互独立的。

长度

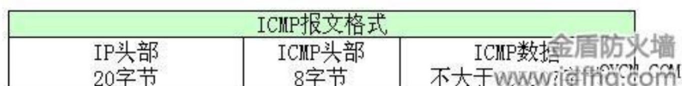
UDP长度字段指的是UDP首部和UDP数据的字节长度。该字段的最小值为 8字节（发送一份0字节的UDP数据报是 O K）。

检验和

UDP检验和是一个端到端的检验和。它由发送端计算，然后由接收端验证。其目的是为了发现UDP首部和数据在发送端到接收端之间发生的任何改动。

IP报文格式

普通的IP首部长为20个字节，除非含有可选项字段。



4位版本

目前协议版本号是4，因此IP有时也称作IPV4.

4位首部长度

首部长度指的是首部占32bit字的数目，包括任何选项。由于它是一个4比特字段，因此首部长度最长为60个字节。

服务类型（TOS）

服务类型字段包括一个3bit的优先权字段（现在已经被忽略），4bit的TOS子字段和1bit未用位必须置0。4bit的TOS分别代表：最小时延，最大吞吐量，最高可靠性和最小费用。4bit中只能置其中1比特。如果所有4bit均为0，那么就意味着是一般服务。

总长度

总长度字段是指整个IP数据报的长度，以字节为单位。利用首部长度和总长度字段，就可以知道IP数据报中数据内容的起始位置和长度。由于该字段长16bit，所以IP数据报最长可达65535字节。当数据报被分片时，该字段的值也随着变化。

标识字段

标识字段唯一地标识主机发送的每一份数据报。通常每发送一份报文它的值就会加1。

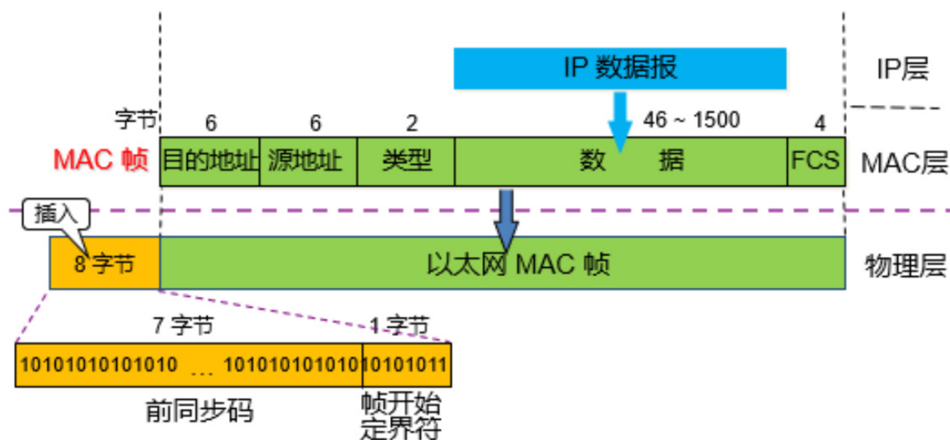
生存时间

TTL（time-to-live）生存时间字段设置了数据报可以经过的最多路由器数。它指定了数据报的生存时间。TTL的初始值由源主机设置（通常为 3 2或6 4），一旦经过一个处理它的路由器，它的值就减去 1。当该字段的值为 0 时，数据报就被丢弃，并发送 ICMP 报文通知源主机。

首部检验和

首部检验和字段是根据 I P首部计算的检验和码。它不对首部后面的数据进行计算。 ICMP、IGMP、UDP和TCP在它们各自的首部中均含有同时覆盖首部和数据检验和码。

以太网报文格式



目的地址和源地址

是指网卡的硬件地址（也叫MAC 地址），长度是48 位，是在网卡出厂时固化的。

数据

以太网帧中的数据长度规定最小46 字节，最大1500 字节，ARP 和RARP 数据包的长度不够46 字节，要在后面补填充位。最大值1500 称为以太网的最大传输单元（MTU），不同的网络类型有不同的MTU，如果一个数据包从以太网路由到拨号链路上，数据包度大于拨号链路的MTU了，则需要对数据包进行分片fragmentation）。ifconfig 命令的输出中也有“MTU:1500”。注意，MTU 个概念指数据帧中有效载荷的最大长度，不包括帧首部的长度。

• HTTP协议

1. HTTP协议1.01.12.0

HTTP1.0：服务器处理完成后立即断开TCP连接（无连接），服务器不跟踪每个客户端也不记录过去的请求（无状态）
HTTP1.1：KeepAlived长连接避免了连接建立和释放的开销；通过Content-Length来判断当前请求数据是否已经全部接受（有状态）
HTTP2.0：引入二进制数据帧和流的概念，其中帧对数据进行顺序标识；因为有了序列，服务器可以并行的传输数据。

http1.0和http1.1的主要区别如下：

- 缓存处理：1.1添加更多的缓存控制策略（如：Entity tag，If-Match）
- 网络连接的优化：1.1支持断点续传
- 错误状态码的增多：1.1新增了24个错误状态响应码，丰富的错误码更加明确各个状态
- Host头处理：支持Host头域，不在以IP为请求方标志
- 长连接：减少了建立和关闭连接的消耗和延迟。

http1.1和http2.0的主要区别：

- 新的传输格式：2.0使用二进制格式，1.0依然使用基于文本格式
- 多路复用：连接共享，不同的request可以使用同一个连接传输（最后根据每个request上的id号组合成正常的请求）
- header压缩：由于1.X中header带有大量的信息，并且得重复传输，2.0使用encoder来减少需要传输的hearder大小
- 服务端推送：同google的SPDUY（1.0的一种升级）一样

2. HTTP与HTTPS之间的区别

HTTP与HTTPS之间的区别

方法	描述
默认端口80	HTTPS默认使用端口443
POST明文传输、数据未加密、安全性差	传输过程ssl加密、安全性较好
响应速度快、消耗资源少	响应速度较慢、消耗资源多、需要用到CA证书

HTTPS链接建立的过程

- 首先客户端先给服务器发送一个请求
- 服务器发送一个SSL证书给客户端，内容包括：证书的发布机构、有效期、所有者、签名以及公钥
- 客户端对发来的公钥进行真伪校验，校验为真则使用公钥对对称加密算法以及对称密钥进行加密
- 服务器端使用私钥进行解密并使用对称密钥加密确认信息发送给客户端
- 随后客户端和服务端就使用对称密钥进行信息传输

对称加密算法

双方持有相同的密钥，且加密速度快，典型对称加密算法：DES、AES

非对称加密算法

密钥成对出现（私钥、公钥），私钥只有自己知道，不在网络中传输；而公钥可以公开。相比对称加密速度较慢，典型的非对称加密算法有：RSA、DSA

3. Get和Post请求区别

HTTP请求

方法	描述
GET	向特定资源发送请求，查询数据，并返回实体
POST	向指定资源提交数据进行处理请求，可能会导致新的资源建立、已有资源修改
PUT	向服务器上传新的内容
HEAD	类似GET请求，返回的响应中没有具体的内容，用于获取报头
DELETE	请求服务器删除指定标识的资源
OPTIONS	可以用来向服务器发送请求来测试服务器的功能性
TRACE	回显服务器收到的请求，用于测试或诊断
CONNECT	HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器

get和Post区别

	GET	POST
可见性	数据在URL中对所有人可见	数据不会显示在URL中
安全性	与post相比，get的安全性较差，因为所发送的数据是URL的一部分	安全，因为参数不会被保存在浏览器历史或web服务器日志中
数据长度	受限制，最长2kb	无限制
编码类型	application/x-www-form-urlencoded	multipart/form-data
缓存	能被缓存	不能被缓存

4. HTTP常见响应状态码

100：Continue --- 继续。客户端应继续其请求。
200：OK --- 请求成功。一般用于GET与POST请求。
301：Moved Permanently --- 永久重定向。
302：Found --- 暂时重定向。
400：Bad Request --- 客户端请求的语法错误，服务器无法理解。
403：Forbideen --- 服务器理解请求客户端的请求，但是拒绝执行此请求。
404：Not Found --- 服务器无法根据客户端的请求找到资源（网页）。
500：Internal Server Error --- 服务器内部错误，无法完成请求。
502：Bad Gateway --- 作为网关或者代理服务器尝试执行请求时，从远程服务器接收到了无效的响应。

5. 重定向和转发区别

重定向：redirect

地址栏发生变化
重定向可以访问其他站点（服务器）的资源
重定向是两次请求。不能使用request对象来共享数据

转发：forward

转发地址栏路径不变
转发只能访问当前服务器下的资源
转发是一次请求，可以使用request对象共享数据

6. Cookie和Session区别

Cookie 和 Session都是用来跟踪浏览器用户身份的会话方式，但两者有所区别：

Cookie 数据保存在客户端(浏览器端)，Session 数据保存在服务器端。

cookie不是很安全，别人可以分析存放在本地的COOKIE并进行欺骗,考虑到安全应当使用session。

Cookie 一般用来保存用户信息，Session 的主要作用就是通过服务端记录用户的状态

• 浏览器输入URL过程

过程：DNS解析、TCP连接、发送HTTP请求、服务器处理请求并返回HTTP报文、浏览器渲染、结束

过程	使用的协议
1、浏览器查找域名DNS的IP地址DNS查找过程（浏览器缓存、路由器缓存、DNS缓存）	DNS：获取域名对应的ip
2、根据ip建立TCP连接	TCP：与服务器建立连接
3、浏览器向服务器发送HTTP请求	HTTP：发送请求
4、服务器响应HTTP响应	HTTP
5、浏览器进行渲染	

操作系统基础

进程和线程的区别

进程：是资源分配的最小单位，一个进程可以有多个线程，多个线程共享进程的堆和方法区资源，不共享栈、程序计数器

线程：是任务调度和执行的最小单位，线程并行执行存在资源竞争和上下文切换的问题

协程：是一种比线程更加轻量级的存在，正如一个进程可以拥有多个线程一样，一个线程可以拥有多个协程。

1. 进程间通信方式IPC

管道pipe

亲缘关系使用匿名管道，非亲缘关系使用命名管道，管道遵循FIFO，半双工，数据只能单向通信；

信号

信号是一种比较复杂的通信方式，用户调用kill命令将信号发送给其他进程。

消息队列

消息队列克服了信号传递信息少，管道只能承载无格式字节流以及缓冲区大小受限等特点。

共享内存(share memory)

- 使得多个进程可以可以直接读写同一块内存空间，是最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的。
- 由于多个进程共享一段内存，因此需要依靠某种同步机制（如信号量）来达到进程间的同步及互斥。

信号量(Semaphores)

信号量是一个计数器，用于多进程对共享数据的访问，这种通信方式主要用于解决与同步相关的问题并避免竞争条件。

套接字(Sockets)

简单的说就是通信的两方的一种约定，用套接字中的相关函数来完成通信过程。

2. 用户态和核心态

用户态：只能受限的访问内存，运行所有的应用程序

核心态：运行操作系统程序，cpu可以访问内存的所有数据，包括外围设备

为什么要有用户态和内核态

由于需要限制不同的程序之间的访问能力, 防止他们获取别的程序的内存数据, 或者获取外围设备的数据, 并发送到网络

用户态切换到内核态的3种方式

a. 系统调用

主动调用，系统调用的机制其核心还是使用了操作系统为用户特别开放的一个中断来实现，例如Linux的int 80h中断。

b. 异常

当CPU在执行运行在用户态下的程序时，发生了某些事先不可知的异常，比如缺页异常，这时会触发切换内核态处理异常。

c. 外围设备的中断

当外围设备完成用户请求的操作后，会向CPU发出相应的中断信号，这时CPU会由用户态到内核态的切换。

3. 操作系统的进程空间

栈区（stack）— 由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。

堆区（heap）— 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收 。

静态区（static）—存放全局变量和静态变量的存储

代码区(text)—存放函数体的二进制代码。

线程共享堆区、静态区

操作系统内存管理

存管理方式：页式管理、段式管理、段页式管理

分段管理

将程序的地址空间划分为若干段（segment），如代码段，数据段，堆栈段；这样每个进程有一个二维地址空间，相互独立，互不干扰。段式管理的优点是：没有内碎片（因为段大小可变，改变段大小来消除内碎片）。但段换入换出时，会产生外碎片（比如4k的段换5k的段，会产生1k的外碎片）

分页管理

在页式存储管理中，将程序的逻辑地址划分为固定大小的页（page），而物理内存划分为同样大小的页框，程序加载时，可以将任意一页放入内存中任意一个页框，这些页框不必连续，从而实现了离散分离。页式存储管理的优点是：没有外碎片（因为页的大小固定），但会产生内碎片（一个页可能填充不满）

段页式管理

段页式管理机制结合了段式管理和页式管理的优点。简单来说段页式管理机制就是把主存先分成若干段，每个段又分成若干页，也就是说 段页式管理机制 中段与段之间以及段的内部的都是离散的

1. 页面置换算法FIFO、LRU

置换算法：先进先出FIFO、最近最久未使用LRU、最佳置换算法OPT

先进先出FIFO

缺点：没有考虑到实际的页面使用频率，性能差、与通常页面使用的规则不符合，实际应用较少

最近最久未使用LRU

原理：选择最近且最久未使用的页面进行淘汰

优点：考虑到了程序访问的时间局部性，有较好的性能，实际应用也比较多

缺点：没有合适的算法，只有适合的算法，IFU、random都可以

```
/**
 * @program: Java
 * @description: LRU最近最久未使用置换算法，通过LinkedHashMap实现
 * @author: Mr.Li
 * @create: 2020-07-17 10:29
 **/
public class LRUCache {
    private LinkedHashMap<Integer,Integer> cache;
    private int capacity;   //容量大小

    /**
     *初始化构造函数
     * @param capacity
     */
    public LRUCache(int capacity) {
        cache = new LinkedHashMap<>(capacity);
        this.capacity = capacity;
    }

    public int get(int key) {
        //缓存中不存在此key，直接返回
        if(!cache.containsKey(key)) {
            return -1;
        }

        int res = cache.get(key);
```

```
        cache.remove(key);  //先从链表中删除
        cache.put(key,res); //再把该节点放到链表末尾处
        return res;
    }

    public void put(int key,int value) {
        if(cache.containsKey(key)) {
            cache.remove(key); //已经存在，在当前链表移除
        }
        if(capacity == cache.size()) {
            //cache已满，删除链表头位置
            Set<Integer> keySet = cache.keySet();
            Iterator<Integer> iterator = keySet.iterator();
            cache.remove(iterator.next());
        }
        cache.put(key,value);  //插入到链表末尾
    }
}
```

```
/**
 * @program: Java
 * @description: LRU最近最久未使用置换算法，通过LinkedHashMap内部removeEldestEntry方法实现
 * @author: Mr.Li
 * @create: 2020-07-17 10:59
 **/
class LRUCache {
    private Map<Integer, Integer> map;
    private int capacity;

    /**
     *初始化构造函数
     * @param capacity
     */
    public LRUCache(int capacity) {
        this.capacity = capacity;
        map = new LinkedHashMap<Integer, Integer>(capacity, 0.75f, true) {
            @Override
            protected boolean removeEldestEntry(Map.Entry eldest) {
                return size() > capacity;  // 容量大于capacity 时就删除
            }
        };
    }
    public int get(int key) {
        //返回key对应的value值，若不存在，返回-1
        return map.getOrDefault(key, -1);
```

```
    }

    public void put(int key, int value) {
        map.put(key, value);
    }
}
```

最佳置换算法OPT

原理：每次选择当前物理块中的页面在未来长时间不被访问的或未来不再使用的页面进行淘汰

优点：具有较好的性能，可以保证获得最低的缺页率

缺点：过于理想化，但是实际上无法实现（没办法预知未来的页面）

2. 死锁条件、解决方式。

死锁是指两个或两个以上进程在执行过程中，因争夺资源而造成的下相互等待的现象；

死锁的条件

互斥条件：进程对所分配到的资源不允许其他进程访问，若其他进程访问该资源，只能等待至占有该资源的进程释放该资源；

请求与保持条件：进程获得一定的资源后，又对其他资源发出请求，阻塞过程中不会释放自己已经占有的资源

非剥夺条件：进程已获得的资源，在未完成使用之前，不可被剥夺，只能在使用后自己释放

循环等待条件：系统中若干进程组成环路，环路中每个进程都在等待相邻进程占用的资源

解决方法：破坏死锁的任意一条件

乐观锁，破坏资源互斥条件，CAS

资源一次性分配，从而剥夺请求和保持条件、tryLock

可剥夺资源：即当进程新的资源未得到满足时，释放已占有的资源，从而破坏不可剥夺的条件，数据库deadlock超时

资源有序分配法：系统给每类资源赋予一个序号，每个进程按编号递增的请求资源，从而破坏环路等待的条件，转账场景

Java基础

• 面向对象三大特性

特性：封装、继承、多态
封装：对抽象的事物抽象化成一个对象，并对其对象的属性私有化，同时提供一些能被外界访问属性的方法；
继承：子类扩展新的数据域或功能，并复用父类的属性与功能，单继承，多实现；
多态：通过继承（多个子类对同一方法的重写）、也可以通过接口（实现接口并覆盖接口）

1. Java与C++区别

不同点：c++支持多继承，并且有指针的概念，由程序员自己管理内存；Java是单继承，可以用接口实现多继承，Java 不提供指针来直接访问内存，程序内存更加安全，并且Java有JVM自动内存管理机制，不需要程序员手动释放无用内存

2. 多态实现原理

多态的底层实现是动态绑定，即在运行时才把方法调用与方法实现关联起来。

静态绑定与动态绑定
一种是在编译期确定，被称为静态分派，比如方法的重载；
一种是在运行时确定，被称为动态分派，比如方法的覆盖（重写）和接口的实现。

多态的实现
虚拟机栈中会存放当前方法调用的栈帧（局部变量表、操作栈、动态连接 、返回地址）。多态的实现过程，就是方法调用动态分派的过程，如果子类覆盖了父类的方法，则在多态调用中，动态绑定过程会首先确定实际类型是子类，从而先搜索到子类中的方法。这个过程便是方法覆盖的本质。

3. static和final关键字

static：可以修饰属性、方法

static修饰属性
类级别属性，所有对象共享一份，随着类的加载而加载（只加载一次），先于对象的创建；可以使用类名直接调用。

static修饰方法
随着类的加载而加载；可以使用类名直接调用；静态方法中，只能调用静态的成员，不可用this；

final：关键字主要用在三个地方：变量、方法、类。

final修饰变量
如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；
如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。

final修饰方法
把方法锁定，以防任何继承类修改它的含义（重写）；类中所有的 private 方法都隐式地指定为 final。

final修饰类
final 修饰类时，表明这个类不能被继承。final 类中的所有成员方法都会被隐式地指定为 final 方法。
一个类不能被继承，除了final关键字之外，还有可以私有化构造器。（内部类无效）

4. 抽象类和接口

抽象类：包含抽象方法的类，即使用abstract修饰的类；抽象类只能被继承，所以不能使用final修饰，抽象类不能被实例化，

接口：接口是一个抽象类型，是抽象方法的集合，接口支持多继承，接口中定义的方法，默认是public abstract修饰的抽象方法

相同点
① 抽象类和接口都不能被实例化
② 抽象类和接口都可以定义抽象方法，子类/实现类必须覆写这些抽象方法

不同点
① 抽象类有构造方法，接口没有构造方法
③抽象类可以包含普通方法，接口中只能是public abstract修饰抽象方法（Java8之后可以）
③ 抽象类只能单继承，接口可以多继承
④ 抽象类可以定义各种类型的成员变量，接口中只能是public static final修饰的静态常量

抽象类的使用场景
既想约束子类具有共同的行为（但不再乎其如何实现），又想拥有缺省的方法，又能拥有实例变量

接口的应用场景

约束多个实现类具有统一的行为，但是不在乎每个实现类如何具体实现；实现类中各个功能之间可能没有任何联系

5. 泛型以及泛型擦除

参考：https://blog.csdn.net/baoyinwang/article/details/107341997

泛型

泛型的本质是参数化类型。这种参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口和泛型方法。

泛型擦除

Java的泛型是伪泛型，使用泛型的时候加上类型参数，在编译器编译生成的字节码的时候会去掉，这个过程成为类型擦除。

如List等类型，在编译之后都会变成 List。JVM 看到的只是 List，而由泛型附加的类型信息对 JVM 来说是不可见的。

可以通过反射添加其它类型元素

6. 反射原理以及使用场景

Java反射

是指在运行状态中，对于任意一个类都能够知道这个类所有的属性和方法；并且都能够调用它的任意一个方法；

反射原理

反射首先是能够获取到Java中的反射类的字节码，然后将字节码中的方法，变量，构造函数等映射成相应的 Method、Filed、Constructor 等类

如何得到Class的实例

```
1.类名.class(就是一份字节码)
2.Class.forName(String className);根据一个类的全限定名来构建Class对象
3.每一个对象多有getClass()方法:obj.getClass();返回对象的真实类型
```

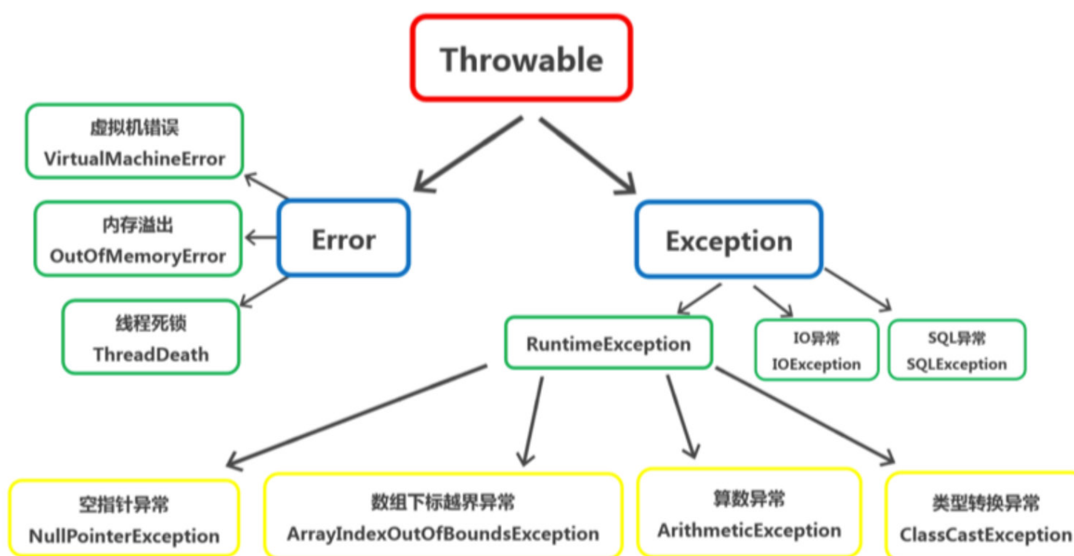
使用场景：

- 开发通用框架 – 反射最重要的用途就是开发各种通用框架。很多框架（比如 Spring）都是配置化的（比如通过 XML 文件配置 JavaBean、Filter 等），为了保证框架的通用性，需要根据配置文件运行时动态加载不同的对象或类，调用不同的方法。

- 动态代理 – 在切面编程（AOP）中，需要拦截特定的方法，通常，会选择动态代理方式。这时，就需要反射技术来实现了。
  JDK：spring默认动态代理，需要实现接口
  CGLIB：通过asm框架序列化字节流，可配置，性能差

- 自定义注解 – 注解本身仅仅是起到标记作用，它需要利用反射机制，根据注解标记去调用注解解释器，执行行为。

7. Java异常体系



Throwable 是 Java 语言中所有错误或异常的超类。下一层分为 Error 和 Exception

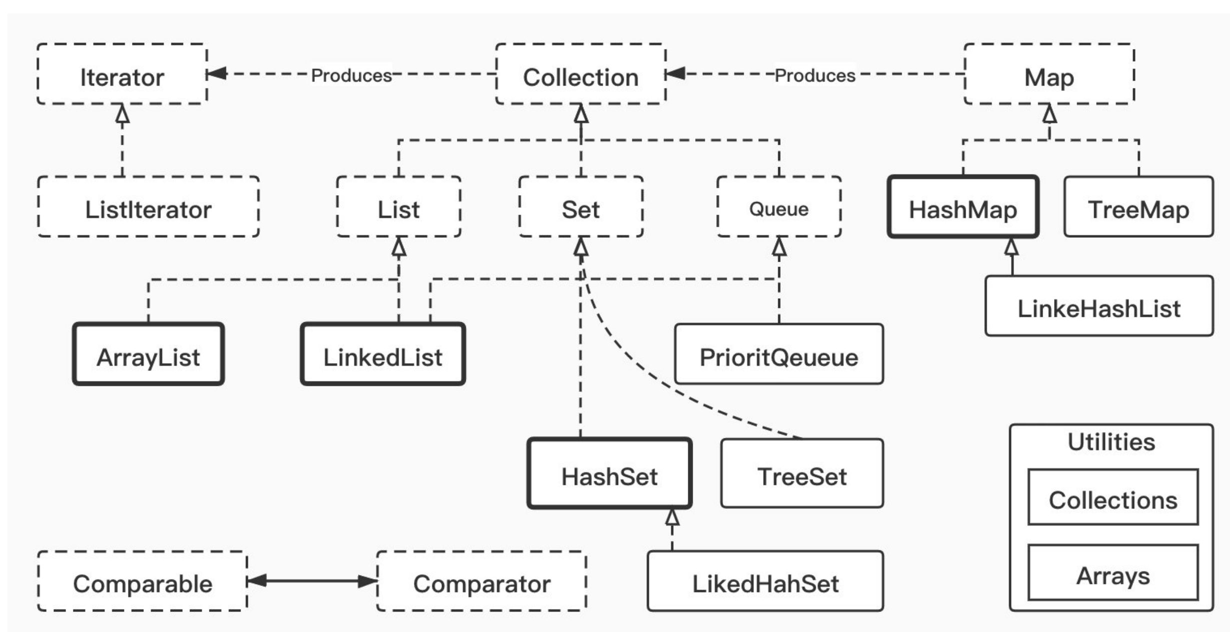
Error：
是指 java 运行时系统的内部错误和资源耗尽错误。应用程序不会抛出该类对象。如果出现了这样的错误，除了告知用户，剩下的就是尽力使程序安全的终止。

Exception 包含：RuntimeException 、CheckedException

编程错误可以分成三类：语法错误、逻辑错误和运行错误。
语法错误（也称编译错误）是在编译过程中出现的错误，由编译器检查发现语法错误
逻辑错误指程序的执行结果与预期不符，可以通过调试定位并发现错误的原因
运行错误是引起程序非正常终端的错误，需要通过异常处理的方式处理运行错误
RuntimeException： 运行时异常，程序应该从逻辑角度尽可能避免这类异常的发生。

如 NullPointerException 、 ClassCastException ；
CheckedException：受检异常，程序使用trycatch进行捕捉处理
如IOException、SQLException、NotFoundException；

• 数据结构



1. ArrayList和LinkedList

ArrayList

底层基于数组实现，支持对元素进行快速随机访问，适合随机查找和遍历，不适合插入和删除。（提一句实际上）默认初始大小为10，当数组容量不够时，会触发扩容机制（扩大到当前的1.5倍），需要将原来数组的数据复制到新的数组中；当从 ArrayList 的中间位置插入或者删除元素时，需要对数组进行复制、移动、代价比较高。

LinkedList

底层基于双向链表实现，适合数据的动态插入和删除；　内部提供了 List 接口中没有定义的方法，用于操作表头和表尾元素，可以当作堆栈、队列和双向队列使用。（比如jdk官方推荐使用基于linkedList的Deque进行堆栈操作）

ArrayList与LinkedList区别

都是线程不安全的，ArrayList 适用于查找的场景，LinkedList 适用于增加、删除多的场景

实现线程安全

可以使用原生的Vector，或者是Collections.synchronizedList(List list)函数返回一个线程安全的ArrayList集合。 建议使用concurrent并发包下的CopyOnWriteArrayList的。

①Vector: 底层通过synchronize修饰保证线程安全，效率较差

②CopyOnWriteArrayList：写时加锁，使用了一种叫写时复制的方法；读操作是可以不用加锁的

2. List遍历快速和安全失败

① 普通for循环遍历List删除指定元素

```
for(int i=0; i < list.size(); i++){
   if(list.get(i) == 5)
       list.remove(i);
}
```

② 迭代遍历,用list.remove(i)方法删除元素

```
Iterator<Integer> it = list.iterator();
while(it.hasNext()){
    Integer value = it.next();
    if(value == 5){
        list.remove(value);
    }
}
```

③foreach遍历List删除元素

```
for(Integer i:list){
    if(i==3) list.remove(i);
}
```

fail—fast：快速失败

当异常产生时，直接抛出异常，程序终止;

fail-fast主要是体现在当我们在遍历集合元素的时候，经常会使用迭代器，但在迭代器遍历元素的过程中，如果集合的结构（modCount）被改变的话，就会抛出异常ConcurrentModificationException，防止继续遍历。这就是所谓的快速失败机制。

fail—safe：安全失败
采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。由于在遍历过程中对原集合所作的修改并不能被迭代器检测到，所以不会触发ConcurrentModificationException。
缺点：基于拷贝内容的优点是避免了ConcurrentModificationException，但同样地，迭代器并不能访问到修改后的内容，即：迭代器遍历的是开始遍历那一刻拿到的集合拷贝，在遍历期间原集合发生的修改迭代器是不知道的。
场景：java.util.concurrent包下的容器都是安全失败，可以在多线程下并发使用，并发修改。

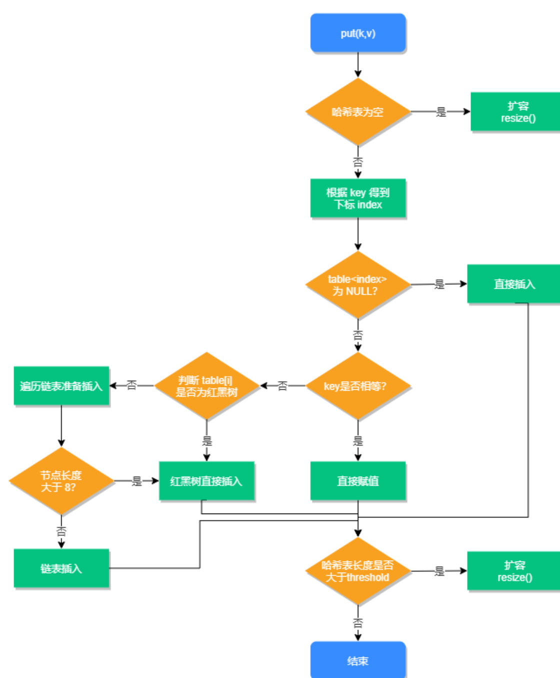
3. 详细介绍HashMap

角度：数据结构+扩容情况+put查找的详细过程+哈希函数+容量为什么始终都是2^N，JDK1.7与1.8的区别。
参考：https://www.jianshu.com/p/9fe4cb316c05

数据结构
HashMap在底层数据结构上采用了数组＋链表＋红黑树，通过散列映射来存储键值对数据

扩容情况
默认的负载因子是0.75，如果数组中已经存储的元素个数大于数组长度的75%，将会引发扩容操作。
【1】创建一个长度为原来数组长度两倍的新数组。
【2】1.7采用Entry的重新hash运算，1.8采用高于与运算。

put操作步骤



① 判断数组是否为空，为空进行初始化;
② 不为空，则计算 key 的 hash 值，通过(n - 1) & hash计算应当存放在数组中的下标 index;
③ 查看 table[index] 是否存在数据，没有数据就构造一个Node节点存放在 table[index] 中;
④ 存在数据，说明发生了hash冲突(存在二个节点key的hash值一样)，继续判断key是否相等，相等，用新的value替换原数据;
⑤ 若不相等，判断当前节点类型是不是树型节点，如果是树型节点，创造树型节点插入红黑树中;
⑥ 若不是红黑树，创建普通Node加入链表中；判断链表长度是否大于 8，大于则将链表转换为红黑树;
⑦ 插入完成之后判断当前节点数是否大于阈值，若大于，则扩容为原数组的二倍

哈希函数

通过hash函数（优质因子31循环累加）先拿到 key 的hashcode，是一个32位的值，然后让hashcode的高16位和低16位进行异或操作。该函数也称为扰动函数，做到尽可能降低hash碰撞，通过尾插法进行插入。

容量为什么始终都是2^N

先做对数组的长度取模运算，得到的余数才能用来要存放的位置也就是对应的数组下标。这个数组下标的计算方法是“ (n - 1) & hash ”。（n代表数组长度）。方便数组的扩容和增删改时的取模。

JDK1.7与1.8的区别

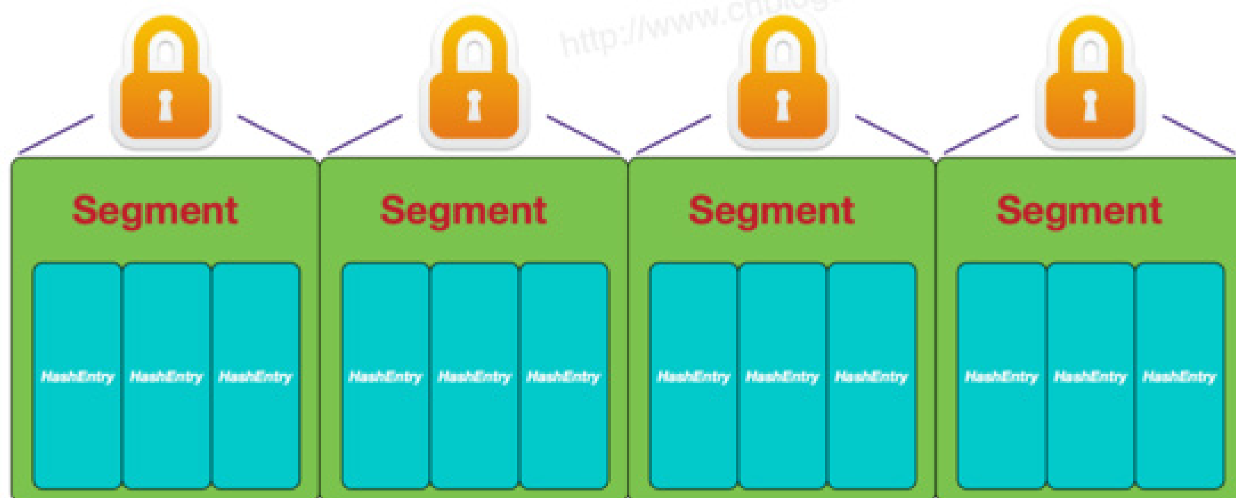
JDK1.7 HashMap:
底层是 数组和链表 结合在一起使用也就是链表散列。如果相同的话，直接覆盖，不相同就通过拉链法解决冲突。扩容翻转时顺序不一致使用头插法会产生死循环，导致cpu100%
JDK1.8 HashMap:
底层数据结构上采用了数组＋链表＋红黑树；当链表长度大于阈值（默认为 8-泊松分布），数组的长度大于 64时，链表将转化为红黑树，以减少搜索时间。（解决了tomcat臭名昭著的url参数dos攻击问题）

4. ConcurrentHashMap

可以通过ConcurrentHashMap 和 Hashtable来实现线程安全；Hashtable 是原始API类，通过synchronize同步修饰，效率低下；ConcurrentHashMap 通过分段锁实现，效率较比Hashtable要好;

ConcurrentHashMap的底层实现

JDK1.7的 ConcurrentHashMap 底层采用 分段的数组+链表 实现；采用 分段锁（Sagment） 对整个桶数组进行了分割分段(Segment默认16个)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。



JDK1.8的ConcurrentHashMap采用的数据结构跟HashMap1.8的结构一样，数组+链表/红黑树；摒弃了Segment的概念，而是直接用 Node 数组+链表+红黑树的数据结构来实现，通过并发控制 synchronized 和CAS来操作保证线程的安全。

5. 序列化和反序列化

序列化的意思就是将对象的状态转化成字节流，以后可以通过这些值再生成相同状态的对象。对象序列化是对象持久化的一种实现方法，它是将对象的属性和方法转化为一种序列化的形式用于存储和传输。反序列化就是根据这些保存的信息重建对象的过程。

序列化：将java对象转化为字节序列的过程。

反序列化：将字节序列转化为java对象的过程。

优点：

a、实现了数据的持久化，通过序列化可以把数据永久地保存到硬盘上（通常存放在文件里）Redis的RDB

b、利用序列化实现远程通信，即在网络上传送对象的字节序列。 Google的protoBuf

反序列化失败的场景：

序列化ID：serialVersionUID不一致的时候，导致反序列化失败

6. String

String 使用数组存储内容，数组使用 final 修饰，因此 String 定义的字符串的值也是不可变的

StringBuffer 对方法加了同步锁，线程安全，效率略低于 StringBuilder

设计模式与原则

1. 单例模式

某个类只能生成一个实例，该实例全局访问，例如Spring容器里一级缓存里的单例池。
优点：
唯一访问：如生成唯一序列化的场景、或者spring默认的bean类型。
提高性能：频繁实例化创建销毁或者耗时耗资源的场景，如连接池、线程池。
缺点：
不适合有状态且需变更的
实现方式：
饿汉式：线程安全速度快
懒汉式：双重检测锁，第一次减少锁的开销、第二次防止重复、volatile防止重排序导致实例化未完成
静态内部类：线程安全利用率高
枚举：effictiveJAVA推荐，反射也无法破坏

2. 工厂模式

定义一个用于创建产品的接口，由子类决定生产何种产品。
优点：解耦：提供参数即可获取产品，通过配置文件可以不修改代码增加具体产品。
缺点：每增加一个产品就得新增一个产品类

3. 抽象工厂模式

提供一个接口，用于创建相关或者依赖对象的家族，并由此进行约束。
优点：可以在类的内部对产品族进行约束
缺点：假如产品族中需要增加一个新的产品，则几乎所有的工厂类都需要进行修改。

面试题

构造方法

构造方法可以被重载，只有当类中没有显性声明任何构造方法时，才会有默认构造方法。

构造方法没有返回值，构造方法的作用是创建新对象。

初始化块

静态初始化块的优先级最高，会最先执行，在非静态初始化块之前执行。

静态初始化块会在类第一次被加载时最先执行，因此在 main 方法之前。

This

关键字 this 代表当前对象的引用。当前对象指的是调用类中的属性或方法的对象

关键字 this 不可以在静态方法中使用。静态方法不依赖于类的具体对象的引用

重写和重载的区别

重载指在同一个类中定义多个方法，这些方法名称相同，签名不同。

重写指在子类中的方法的名称和签名都和父类相同，使用override注解

Object类方法

toString 默认是个指针，一般需要重写

equals 比较对象是否相同，默认和==功能一致

hashCode 散列码，equals则hashCode相同，所以重写equals必须重写hashCode

finalize 用于垃圾回收之前做的遗嘱，默认空，子类需重写

clone 深拷贝，类需实现cloneable的接口

getClass 反射获取对象元数据，包括类名、方法、

notify、wait 用于线程通知和唤醒

基本数据类型和包装类

基本数据类型	存储大小	取值范围	默认值
byte	8 位有符号数	-2^7 到 2^7-1	0
short	16 位有符号数	-2^{15} 到 $2^{15}-1$	0
int	32 位有符号数	-2^{31} 到 $2^{31}-1$	0
long	64 位有符号数	-2^{63} 到 $2^{63}-1$	0L
float	32 位，符合 IEEE 754 标准	负数 -3.402823e+38 到 -1.401298e-45，正数 1.401298e-45 到 3.402823e+38	0.0f
double	64 位，符合 IEEE 754 标准	负数 -1.797693e+308 到 -4.9000000e-324，正数 4.9000000e-324 到 1.797693e+308	0.0d
char	16 位	0 到 $2^{16}-1$	'\u0000'
boolean	1 位	true 和 false	false

类型	缓存范围
Byte,Short,Integer,Long	[−128, 127]
Character	[0, 127]
Boolean	[false, true]

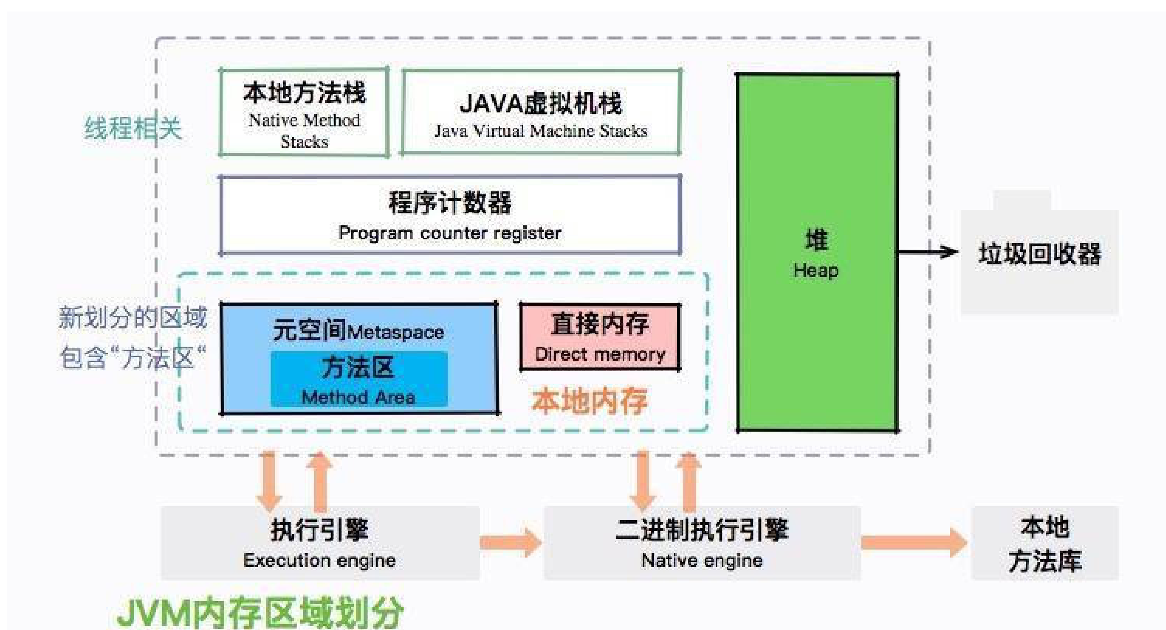
大淘宝技术 TAO TECHNOLOGY

JVM篇

JVM内存划分

1. JVM运行时数据区域

堆、方法区（元空间）、虚拟机栈、本地方法栈、程序计数器



Heap(堆)：

对象的实例以及数组的内存都是要在堆上进行分配的，堆是线程共享的一块区域，用来存放对象实例，也是垃圾回收（GC）的主要区域；开启逃逸分析后，某些未逃逸的对象可以通过标量替换的方式在栈中分配

堆细分：新生代、老年代，对于新生代又分为：Eden区和Surviver1和Surviver2区；

方法区

对于JVM的方法区也可以称之为永久区，它储存的是已经被java虚拟机加载的类信息、常量、静态变量；Jdk1.8以后取消了方法区这个概念，称之为元空间（MetaSpace）；

当应用中的 Java 类过多时，比如 Spring 等一些使用动态代理的框架生成了很多类，如果占用空间超出了我们的设定值，就会发生元空间溢出

虚拟机栈

虚拟机栈是线程私有的，他的生命周期和线程的生命周期是一致的。里面装的是一个一个的栈帧，每一个方法在执行的时候都会创建一个栈帧，栈帧中用来存放（局部变量表、操作数栈 、动态链接 、返回地址）；在Java虚拟机规范中，对此区域规定了两种异常状况：如果线程请求的栈深度大于虚拟机所允许的深度，将会抛出StackOverflowError异常；如果虚拟机栈动态扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。

- 局部变量表：局部变量表是一组变量值存储空间，用来存放方法参数、方法内部定义的局部变量。底层是变量槽（variable slot）
- 操作数栈：是用来记录一个方法在执行的过程中，字节码指令向操作数栈中进行入栈和出栈的过程。大小在编译的时候已经确定了，当一个方法刚开始执行的时候，操作数栈中是空发的，在方法执行的过程中会有各种字节码指令往操作数栈中入栈和出栈。
- 动态链接：因为字节码文件中有很多符号的引用，这些符号引用一部分会在类加载的解析阶段或第一次使用的时候转化成直接引用，这种称为静态解析；另一部分会在运行期间转化为直接引用，称为动态链接。
- 返回地址（returnAddress）：类型（指向了一条字节码指令的地址）
  JIT即时编译器（Just In Time Compiler），简称 JIT 编译器:
- 为了提高热点代码的执行效率，在运行时，虚拟机将会把这些代码编译成与本地平台相关的机器码，并进行各种层次的优化，比如锁粗化等

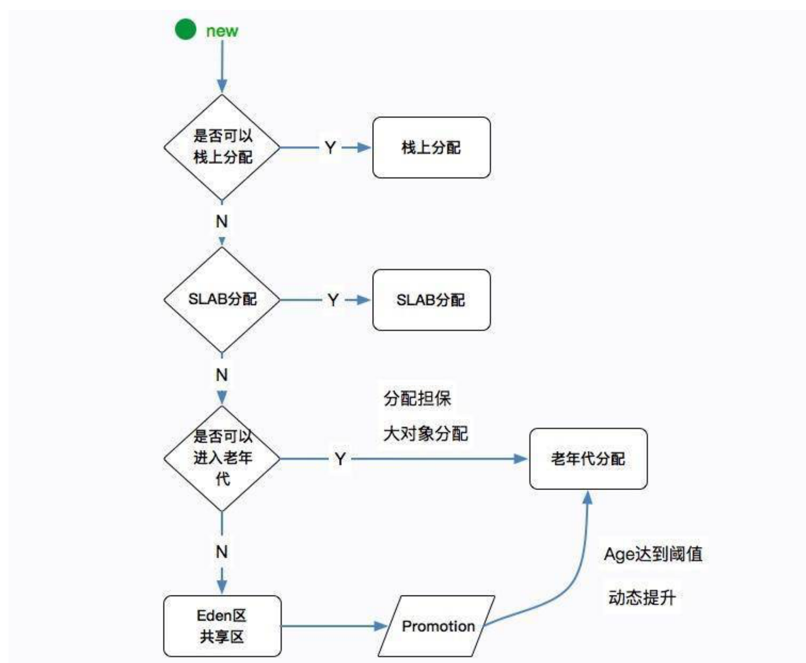
本地方法栈

本地方法栈和虚拟机栈类似，不同的是虚拟机栈服务的是Java方法，而本地方法栈服务的是Native方法。在HotSpot虚拟机实现中是把本地方法栈和虚拟机栈合二为一的，同理它也会抛出StackOverflowError和OOM异常。

PC程序计数器

PC，指的是存放下一条指令的位置的一个指针。它是一块较小的内存空间，且是线程私有的。由于线程的切换，CPU在执行的过程中，需要记住原线程的下一条指令的位置，所以每一个线程都需要有自己的PC。

2. 堆内存分配策略



- 对象优先分配在Eden区，如果Eden区没有足够的空间进行分配时，虚拟机执行一次MinorGC。而那些无需回收的存活对象，将会进到 Survivor 的 From 区（From 区内存不足时，直接进入 Old 区）。
- 大对象直接进入老年代（需要大量连续内存空间的对象）。这样做的目的是避免在Eden区和两个Survivor区之间发生大量的内存拷贝（新生代采用复制算法收集内存）。
- 长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄（Age Count）计数器，如果对象经过了1次Minor GC那么对象会进入Survivor区，之后每经过一次Minor GC那么对象的年龄加1，直到达到阀值（默认15次），对象进入老年区。
  （动态对象年龄判定：程序从年龄最小的对象开始累加，如果累加的对象大小，大于幸存区的一半，则将当前的对象 age 作为新的阈值，年龄大于此阈值的对象则直接进入老年代）
- 每次进行Minor GC或者大对象直接进入老年区时，JVM会计算所需空间大小如小于老年区的剩余值大小，则进行一次Full GC。

3. 创建一个对象的步骤

步骤：类加载检查、分配内存、初始化零值、设置对象头、执行init方法

①类加载检查：
虚拟机遇到 new 指令时，首先去检查是否能在常量池中定位到这个类的符号引用，并且检查这个符号引用代表的类是否已被加载过、解析和初始化过。如果没有，那必须先执行相应的类加载过程。

②分配内存：

在类加载检查通过后，接下来虚拟机将为新生对象分配内存，分配方式有“指针碰撞”和“空闲列表”两种，选择那种分配方式由Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

③初始化零值：

内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值，这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

④设置对象头：

初始化零值完成之后，虚拟机要对对象进行必要的设置，例如这个对象是那个类的实例、如何才能找到类的元数据信息、对象的哈希吗、对象的 GC 分代年龄等信息。 这些信息存放在对象头中。 另外，根据虚拟机当前运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。

⑤执行 init 方法：

从虚拟机的视角来看，一个新的对象已经产生了，但从Java 程序的视角来看， 方法还没有执行，所有的字段都还为零。所以一般来说（除循环依赖），执行 new 指令之后会接着执行 方法，这样一个真正可用的对象才算产生出来。

4. 对象引用

普通的对象引用关系就是强引用。

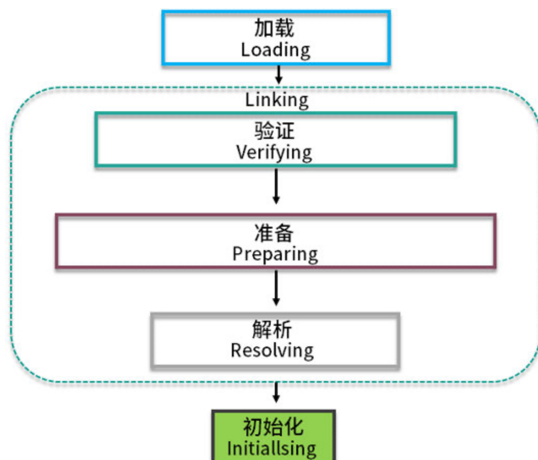
软引用用于维护一些可有可无的对象。只有在内存不足时，系统则会回收软引用对象，如果回收了软引用对象之后仍然没有足够的内存，才会抛出内存溢出异常。

弱引用对象相比软引用来说，要更加无用一些，它拥有更短的生命周期，当 JVM 进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象。

虚引用是一种形同虚设的引用，在现实场景中用的不是很多，它主要用来跟踪对象被垃圾回收的活动。

JVM类加载过程

过程：加载、验证、准备、解析、初始化



加载阶段

1.通过一个类的全限定名来获取定义此类的二进制字节流。

2.将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。

3.在Java堆中生成一个代表这个类的java.lang.class对象，作为方法区这些数据的访问入口。

验证阶段

1.文件格式验证（是否符合Class文件格式的规范，并且能被当前版本的虚拟机处理）

2.元数据验证（对字节码描述的信息进行语意分析，以保证其描述的信息符合Java语言规范要求）

3.字节码验证（保证被校验类的方法在运行时不会做出危害虚拟机安全的行为）

4.符号引用验证（虚拟机将符号引用转化为直接引用时，解析阶段中发生）

准备阶段

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段。将对象初始化为“零”值

解析阶段

解析阶段时虚拟机将常量池内的符号引用替换为直接引用的过程。

字符串常量池：堆上，默认class文件的静态常量池

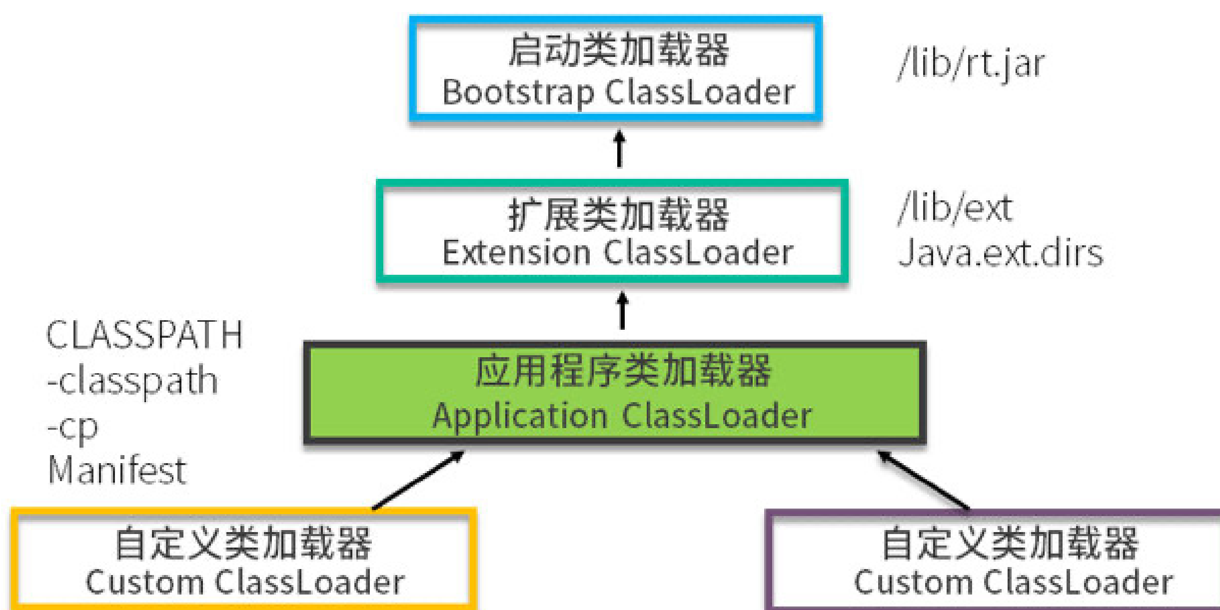
v运行时常量池：在方法区，属于元空间

初始化阶段

初始化阶段时加载过程的最后一步，而这一阶段也是真正意义上开始执行类中定义的Java程序代码。

1. 双亲委派机制

每一个类都有一个对应它的类加载器。系统中的 ClassLoder 在协同工作的时候会默认使用 双亲委派模型 。即在类加载的时候，系统会首先判断当前类是否被加载过。已经被加载的类会直接返回，否则才会尝试加载。加载的时候，首先会把该请求委派该父类加载器的 loadClass() 处理，因此所有的请求最终都应该传送到顶层的启动类加载器 BootstrapClassLoader 中。当父类加载器无法处理时，才由自己来处理。当父类加载器为null时，会使用启动类加载器 BootstrapClassLoader 作为父类加载器。



使用好处

此机制保证JDK核心类的优先加载；使得Java程序的稳定运行，可以避免类的重复加载，也保证了 Java 的核心 API 不被篡改。如果不用没有使用双亲委派模型，而是每个类加载器加载自己的话就会出现一些问题，比如我们编写一个称为 java.lang.Object 类的话，那么程序运行的时候，系统就会出现多个不同的Object 类。

破坏双亲委派机制

可以自己定义一个类加载器，重写loadClass方法；
Tomcat 可以加载自己目录下的 class 文件，并不会传递给父类的加载器；
Java 的 SPI，发起者 BootstrapClassLoader 已经是最上层了，它直接获取了 AppClassLoader 进行驱动加载，和双亲委派是相反的。

2. tomcat的类加载机制

步骤

1. 先在本地cache查找该类是否已经加载过，看看 Tomcat 有没有加载过这个类。
2. 如果Tomcat 没有加载过这个类，则从系统类加载器的cache中查找是否加载过。
3. 如果没有加载过这个类，尝试用ExtClassLoader类加载器类加载，重点来了，这里并没有首先使用 AppClassLoader 来加载类。这个Tomcat 的 WebAPPClassLoader 违背了双亲委派机制，直接使用了 ExtClassLoader 来加载类。这里注意 ExtClassLoader 双亲委派依然有效，ExtClassLoader 就会使用 Bootstrap ClassLoader 来对类进行加载，保证了 Jre 里面的核心类不会被重复加载。 比如在 Web 中加载一个 Object 类。WebAppClassLoader → ExtClassLoader → Bootstrap ClassLoader，这个加载链，就保证了 Object 不会被重复加载。
4. 如果BoostrapClassLoader，没有加载成功，就会调用自己的findClass方法由自己来对类进行加载，findClass 加载类的地址是自己本 web 应用下的 class。
5. 加载依然失败，才使用 AppClassLoader 继续加载。
6. 都没有加载成功的话，抛出异常。

总结一下以上步骤，WebAppClassLoader 加载类的时候，故意打破了JVM 双亲委派机制，绕开了 AppClassLoader，直接先使用 ExtClassLoader 来加载类。

JVM垃圾回收

1. 存活算法和两次标记过程

引用计数法

给对象添加一个引用计数器，每当由一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器为0的对象就是不可能再被使用的。

优点：实现简单，判定效率也很高

缺点：他很难解决对象之间相互循环引用的问题，基本上被抛弃

可达性分析法

通过一系列的成为“GC Roots”(活动线程相关的各种引用，虚拟机栈帧引用，静态变量引用，JNI引用)的对象作为起始点，从这些节点ReferenceChains开始向下搜索，搜索所走过的路径成为引用链，当一个对象到GC ROOTS没有任何引用链相连时，则证明此对象时不可用的；

两次标记过程

对象被回收之前，该对象的finalize()方法会被调用；两次标记，即第一次标记不在“关系网”中的对象。第二次的话就要先判断该对象有没有实现finalize()方法了，如果没有实现就直接判断该对象可回收；如果实现了就会先放在一个

队列中，并由虚拟机建立的一个低优先级的线程去执行它，随后就会进行第二次的小规模标记，在这次被标记的对象就会真正的被回收了。

2. 垃圾回收算法

垃圾回收算法：复制算法、标记清除、标记整理、分代收集

复制算法：(young)

将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收；
优点：实现简单，内存效率高，不易产生碎片
缺点：内存压缩了一半，倘若存活对象多，Copying 算法的效率会大大降低

标记清除：(cms)

标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象
缺点：效率低，标记清除后会产生大量不连续的碎片，需要预留空间给分配阶段的浮动垃圾

标记整理：(old)

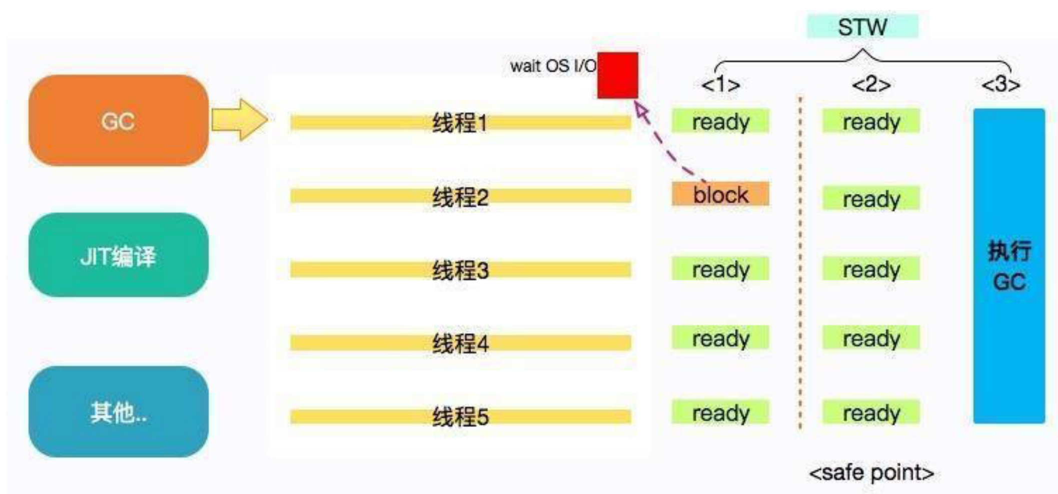
标记过程仍然与“标记-清除”算法一样，再让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存；解决了产生大量不连续碎片问题

分代收集

根据各个年代的特点选择合适的垃圾收集算法。
新生代采用复制算法，新生代每次垃圾回收都要回收大部分对象，存活对象较少，即要复制的操作比较少，一般将新生代划分为一块较大的 Eden 空间和两个较小的 Survivor 空间(From Space, To Space)，每次使用Eden 空间和其中的一块 Survivor 空间，当进行回收时，将该两块空间中还存活的对象复制到另一块 Survivor 空间中。

老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集。

Safepoint 当发生 GC 时，用户线程必须全部停下来，才可以进行垃圾回收，这个状态我们可以认为 JVM 是安全的（safe），整个堆的状态是稳定的。如果在 GC 前，有线程迟迟进入不了 safepoint，那么整个 JVM 都在等待这个阻塞的线程，造成了整体 GC 的时间变长



MinorGC、MajorGC、FullGC

MinorGC 在年轻代空间不足的时候发生，

MajorGC 指的是老年代的 GC，出现 MajorGC 一般经常伴有 MinorGC。

FullGC 1、当老年代无法再分配内存的时候；2、元空间不足的时候；3、显示调用 System.gc 的时候。另外，像 CMS 一类的垃圾回收器，在 MinorGC 出现 promotion failure 的时候也会发生 FullGC。

对象优先在 Eden 区分配

大多数情况下，对象在新生代 Eden 区分配，当 Eden 区空间不够时，发起 Minor GC。

大对象直接进入老年代

大对象是指需要连续内存空间的对象，比如很长的字符串以及数组。老年代直接分配的目的是避免在 Eden 区和 Survivor 区之间出现大量内存复制。

长期存活的对象进入老年代

虚拟机给每个对象定义了年龄计数器，对象在 Eden 区出生之后，如果经过一次 Minor GC 之后，将进入 Survivor 区，同时对象年龄变为 1，增加到一定阈值时则进入老年代（阈值默认为 15）

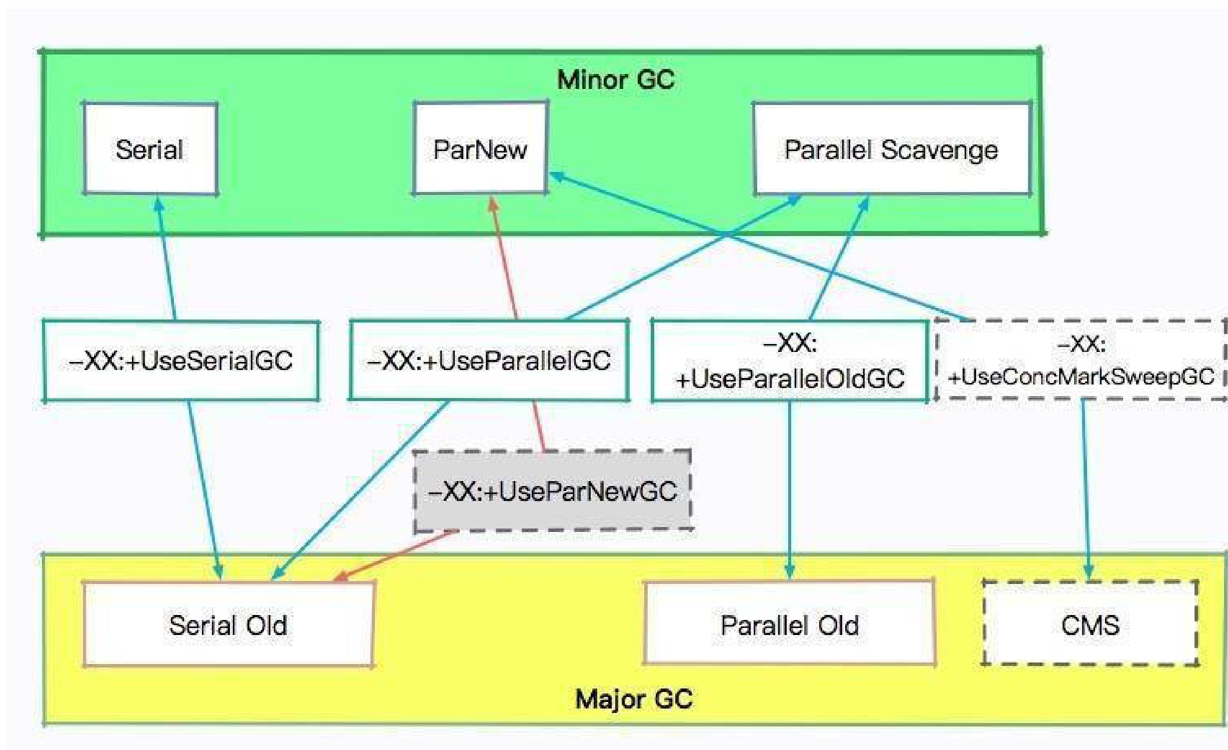
动态对象年龄判定

为了能更好地适应不同程序的内存状况，虚拟机并不总是要求对象的年龄必须达到阈值才能进入老年代。如果在 Survivor 区中相同年龄的所有对象的空间总和大于 Survivor 区空间的一半，则年龄大于或等于该年龄的对象直接进入老年代。

空间分配担保

在发生 Minor GC 之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象的空间总和，如果这个条件成立，那么 Minor GC 可以确保是安全的。如果不成立则进行 Full GC。

3. 垃圾收集器



JDK3：Serial Parnew 关注效率

Serial

Serial 是一个单线程的收集器，它不但只会使用一个 CPU 或一条线程去完成垃圾收集工作，并且在进行垃圾收集的同时，必须暂停其他所有的工作线程，直到垃圾收集结束。适合用于客户端垃圾收集器。

Parnew

ParNew 垃圾收集器其实是 Serial 收集器的多线程版本，也使用复制算法，除了使用多线程进行垃圾收集之外，其余的行为和 Serial 收集器完全一样，ParNew 垃圾收集器在垃圾收集过程中同样也要暂停所有其他的工作线程。

JDK5：parallel Scavenge+（Serial old/parallel old）关注吞吐量

parallel Scavenge：(关注吞吐量)

Parallel Scavenge收集器关注点是吞吐量（高效率的利用CPU）。CMS等垃圾收集器的关注点更多的是用户线程的停顿时间（提高用户体验）；高吞吐量可以最高效率地利用 CPU 时间，尽快地完成程序的运算任务，主要适用于在后台运算而不需要太多交互的任务。

Serial old

Serial收集器的老年代版本，它同样是一个单线程收集器，使用标记-整理算法。主要有两个用途：
在 JDK1.5 之前版本中与新生代的 Parallel Scavenge 收集器搭配使用。
作为年老代中使用 CMS 收集器的后备垃圾收集方案。

parallel old

Parallel Scavenge收集器的老年代版本。使用多线程和“标记-整理”算法。

JDK8-CMS：（关注最短垃圾回收停顿时间）

CMS收集器是一种年老代垃圾收集器，其最主要目标是获取最短垃圾回收停顿时间，和其他年老代使用标记-整理算法不同，它使用多线程的标记-清除算法。最短的垃圾收集停顿时间可以为交互比较高的程序提高用户体验。
CMS 工作机制相比其他的垃圾收集器来说更复杂，整个过程分为以下 4 个阶段：
初始标记：只是标记一下 GC Roots 能直接关联的对象，速度很快，STW。
并发标记：进行 ReferenceChains跟踪的过程，和用户线程一起工作，不需要暂停工作线程。
重新标记：为了修正在并发标记期间，因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，STW。
并发清除：清除 GC Roots 不可达对象，和用户线程一起工作，不需要暂停工作线程。
由于耗时最长的并发标记和并发清除过程中，垃圾收集线程可以和用户现在一起并发工作，所以总体上来看CMS收集器的内存回收和用户线程是一起并发地执行。
优点：并发收集、低停顿
缺点：对CPU资源敏感；无法处理浮动垃圾；使用“标记清除”算法，会导致大量空间碎片产生。

JDK9-G1：（精准控制停顿时间，避免垃圾碎片）

是一款面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存的机器.以极高概率满足GC停顿时间要求的同时,还具备高吞吐量性能特征；相比与 CMS 收集器，G1 收集器两个最突出的改进是：
【1】基于标记-整理算法，不产生内存碎片。
【2】可以非常精确控制停顿时间，在不牺牲吞吐量前提下，实现低停顿垃圾回收。

G1 收集器避免全区域垃圾收集，它把堆内存划分为大小固定的几个独立区域，并且跟踪这些区域的垃圾收集进度，同时在后台维护一个优先级列表，每次根据所允许的收集时间，优先回收垃圾最多的区域。区域划分和优先级区域回收机制，确保 G1 收集器可以在有限时间获得最高的垃圾收集效率。

初始标记：Stop The World，仅使用一条初始标记线程对GC Roots关联的对象进行标记
并发标记：使用一条标记线程与用户线程并发执行。此过程进行可达性分析，速度很慢
最终标记：Stop The World，使用多条标记线程并发执行
筛选回收：回收废弃对象，此时也要 Stop The World，并使用多条筛选回收线程并发执行

JDK11-ZGC:（在不关注容量的情况获取最小停顿时间5TB/10ms）

着色笔技术：加快标记过程

读屏障：解决GC和应用之间并发导致的STW问题

- 支持 TB 级堆内存（最大 4T， JDK13 最大16TB）
- 最大 GC 停顿 10ms
- 对吞吐量影响最大，不超过 15%

4. 配置垃圾收集器

- 首先是内存大小问题，基本上每一个内存区域我都会设置一个上限，来避免溢出问题，比如元空间。
- 通常，堆空间我会设置成操作系统的 2/3，超过 8GB 的堆，优先选用 G1
- 然后我会对 JVM 进行初步优化，比如根据老年代的对象提升速度，来调整年轻代和老年代之间的比例
- 依据系统容量、访问延迟、吞吐量等进行专项优化，我们的服务是高并发的，对 STW 的时间敏感
- 我会通过记录详细的 GC 日志，来找到这个瓶颈点，借用 GCeasy 这样的日志分析工具，定位问题

5. JVM性能调优

对应进程的JVM状态以定位问题和解决问题并作出相应的优化

常用命令：jps、jinfo、jstat、jstack、jmap

jps：查看java进程及相关信息

```
jps -l 输出jar包路径，类全名
jps -m 输出main参数
jps -v 输出JVM参数
```

jinfo：查看JVM参数

```
jinfo 11666
jinfo -flags 11666
Xmx、Xms、Xmn、MetaspaceSize
```

jstat：查看JVM运行时的状态信息，包括内存状态、垃圾回收

```
jstat [option] LVMID [interval] [count]
其中LVMID是进程id，interval是打印间隔时间（毫秒），count是打印次数（默认一直打印）

option参数解释:
-gc 垃圾回收堆的行为统计
-gccapacity 各个垃圾回收代容量(young,old,perm)和他们相应的空间统计
-gcutil 垃圾回收统计概述
-gcnew 新生代行为统计
-gcold 年老代和永生代行为统计
```

jstack：查看JVM线程快照，jstack命令可以定位线程出现长时间卡顿的原因，例如死锁，死循环

```
jstack [-l] <pid> (连接运行中的进程)

option参数解释:
-F 当使用jstack <pid>无响应时，强制输出线程堆栈。
-m 同时输出java和本地堆栈(混合模式)
-l 额外显示锁信息
```

jmap：可以用来查看内存信息 (配合jhat使用)

```
jmap [option] <pid> (连接正在执行的进程)

option参数解释:
-heap 打印java heap摘要
-dump:<dump-options> 生成java堆的dump文件
```

6. JDK新特性

JDK8

支持 Lamda 表达式、集合的 stream 操作、提升HashMap性能

查看比如　CPU、系统内存等，通过历史状态可以体现一个趋势性问题，而这些信息的获取一般依靠监控系统的协作。

```
//Stream API中iterate方法的新重载方法，可以指定什么时候结束迭代
IntStream.iterate(1, i -> i < 100, i -> i + 1).forEach(System.out::println);
```

JDK9
默认G1垃圾回收器

JDK10
其重点在于通过完全GC并行来改善G1最坏情况的等待时间。

JDK11
ZGC (并发回收的策略) 4TB
用于 Lambda 参数的局部变量语法

JDK12
Shenandoah GC (GC 算法)停顿时间和堆的大小没有任何关系，并行关注停顿响应时间。

JDK13
增加ZGC以将未使用的堆内存返回给操作系统，16TB

JDK14
删除cms垃圾回收器、弃用ParallelScavenge+SerialOldGC垃圾回收算法组合
将ZGC垃圾回收器应用到macOS和windows平台

线上故障排查

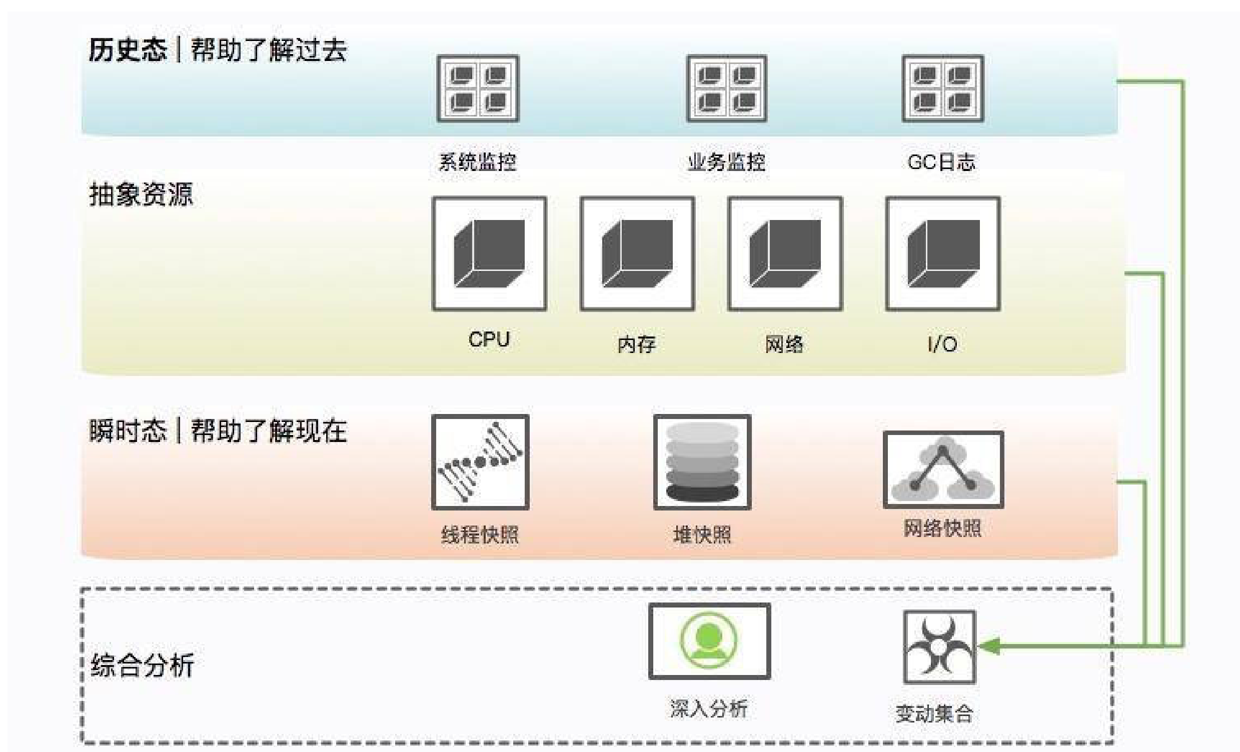
1. 硬件故障排查

如果一个实例发生了问题，根据情况选择，要不要着急去重启。如果出现的CPU、内存飙高或者日志里出现了OOM异常
第一步是隔离，第二步是保留现场，第三步才是问题排查。

隔离
就是把你的这台机器从请求列表里摘除，比如把 nginx 相关的权重设成零。

现场保留

瞬时态和历史态
查看比如 CPU、系统内存等，通过历史状态可以体现一个趋势性问题，而这些信息的获取一般依靠监控系统的协作。



保留信息

（1）系统当前网络连接

```
ss -antp > $DUMP_DIR/ss.dump 2>&1
```

使用 ss 命令而不是 netstat 的原因，是因为 netstat 在网络连接非常多的情况下，执行非常缓慢。
后续的处理，可通过查看各种网络连接状态的梳理，来排查 TIMEWAIT 或者 CLOSEWAIT，或者其他连接过高的问题，非常有用。

（2）网络状态统计

```
netstat -s > $DUMP_DIR/netstat-s.dump 2>&1
```

它能够按照各个协议进行统计输出，对把握当时整个网络状态，有非常大的作用。

```
sar -n DEV 1 2 > $DUMP_DIR/sar-traffic.dump 2>&1
```

在一些速度非常高的模块上，比如 Redis、Kafka，就经常发生跑满网卡的情况。表现形式就是网络通信非常缓慢。

（3）进程资源

```
ss -antp > $DUMP_DIR/ss.dump 2>&1
```

通过查看进程，能看到打开了哪些文件，可以以进程的维度来查看整个资源的使用情况，包括每条网络连接、每个打开的文件句柄。同时，也可以很容易的看到连接到了哪些服务器、使用了哪些资源。这个命令在资源非常多的情况下，输出稍慢，请耐心等待。

（4）CPU 资源

```
mpstat > $DUMP_DIR/mpstat.dump 2>&1
vmstat 1 3 > $DUMP_DIR/vmstat.dump 2>&1
sar -p ALL  > $DUMP_DIR/sar-cpu.dump  2>&1
uptime > $DUMP_DIR/uptime.dump 2>&1
```

主要用于输出当前系统的 CPU 和负载，便于事后排查。

（5）I/O 资源

```
iostat -x > $DUMP_DIR/iostat.dump 2>&1
```

一般，以计算为主的服务节点，I/O 资源会比较正常，但有时也会发生问题，比如日志输出过多，或者磁盘问题等。此命令可以输出每块磁盘的基本性能信息，用来排查 I/O 问题。在第 8 课时介绍的 GC 日志分磁盘问题，就可以使用这个命令去发现。

（6）内存问题

```
free -h > $DUMP_DIR/free.dump 2>&1
```

free 命令能够大体展现操作系统的内存概况，这是故障排查中一个非常重要的点，比如 SWAP 影响了 GC，SLAB 区挤占了 JVM 的内存。

（7）其他全局

```
ps -ef > $DUMP_DIR/ps.dump 2>&1
dmesg > $DUMP_DIR/dmesg.dump 2>&1
sysctl -a > $DUMP_DIR/sysctl.dump 2>&1
```

dmesg 是许多静悄悄死掉的服务留下的最后一点线索。当然，ps 作为执行频率最高的一个命令，由于内核的配置参数，会对系统和 JVM 产生影响，所以我们也输出了一份。

（8）进程快照，最后的遗言（jinfo）

```
${JDK_BIN}jinfo $PID > $DUMP_DIR/jinfo.dump 2>&1
```

此命令将输出 Java 的基本进程信息，包括环境变量和参数配置，可以查看是否因为一些错误的配置造成了 JVM 问题

（9）dump 堆信息

```
${JDK_BIN}jstat -gcutil $PID > $DUMP_DIR/jstat-gcutil.dump 2>&1
${JDK_BIN}jstat -gccapacity $PID > $DUMP_DIR/jstat-gccapacity.dump 2>&1
```

jstat 将输出当前的 gc 信息。一般，基本能大体看出一个端倪，如果不能，可将借助 jmap 来进行分析。

（10）堆信息

```
${JDK_BIN}jmap $PID > $DUMP_DIR/jmap.dump 2>&1
${JDK_BIN}jmap -heap $PID > $DUMP_DIR/jmap-heap.dump 2>&1
${JDK_BIN}jmap -histo $PID > $DUMP_DIR/jmap-histo.dump 2>&1
${JDK_BIN}jmap -dump:format=b,file=$DUMP_DIR/heap.bin $PID > /dev/null  2>&1
```

jmap 将会得到当前 Java 进程的 dump 信息。如上所示，其实最有用的就是第 4 个命令，但是前面三个能够让你初步对系统概况进行大体判断。因为，第 4 个命令产生的文件，一般都非常的大。而且，需要下载下来，导入 MAT 这样的工具进行深入分析，才能获取结果。这是分析内存泄漏一个必经的过程。

（11）JVM 执行栈

```
${JDK_BIN}jstack $PID > $DUMP_DIR/jstack.dump 2>&1
```

jstack 将会获取当时的执行栈。一般会多次取值，我们这里取一次即可。这些信息非常有用，能够还原 Java 进程中的线程情况。

```
top -Hp $PID -b -n 1 -c >  $DUMP_DIR/top-$PID.dump 2>&1
```

为了能够得到更加精细的信息，我们使用 top 命令，来获取进程中所有线程的 CPU 信息，这样，就可以看到资源到底耗费在什么地方了。

（12）高级替补

```
kill -3 $PID
```

有时候，jstack 并不能够运行，有很多原因，比如 Java 进程几乎不响应了等之类的情况。我们会尝试向进程发送 kill -3 信号，这个信号将会打印 jstack 的 trace 信息到日志文件中，是 jstack 的一个替补方案。

```
gcore -o $DUMP_DIR/core $PID
```

对于 jmap 无法执行的问题，也有替补，那就是 GDB 组件中的 gcore，将会生成一个 core 文件。我们可以使用如下的命令去生成 dump：

```
${JDK_BIN}jhsdb jmap --exe ${JDK}java  --core $DUMP_DIR/core --binaryheap
```

（13）内存泄漏的现象

稍微提一下 jmap 命令，它在 9 版本里被干掉了，取而代之的是 jhsdb，你可以像下面的命令一样使用。

```
jhsdb jmap  --heap --pid  37340
jhsdb jmap  --pid  37288
jhsdb jmap  --histo --pid  37340
jhsdb jmap  --binaryheap --pid  37340
```

一般内存溢出，表现形式就是 Old 区的占用持续上升，即使经过了多轮 GC 也没有明显改善。比如ThreadLocal里面的GC Roots，内存泄漏的根本就是，这些对象并没有切断和 GC Roots 的关系，可通过一些工具，能够看到它们的联系。

2. 报表异常 | JVM调优

有一个报表系统，频繁发生内存溢出，在高峰期间使用时，还会频繁的发生拒绝服务，由于大多数使用者是管理员角色，所以很快就反馈到研发这里。

业务场景是由于有些结果集的字段不是太全，因此需要对结果集合进行循环，并通过 HttpClient 调用其他服务的接口进行数据填充。使用 Guava 做了 JVM 内缓存，但是响应时间依然很长。

初步排查，JVM 的资源太少。接口 A 每次进行报表计算时，都要涉及几百兆的内存，而且在内存里驻留很长时间，有些计算又非常耗 CPU，特别的“吃”资源。而我们分配给 JVM 的内存只有 3 GB，在多人访问这些接口的时候，内存就不够用了，进而发生了 OOM。在这种情况下，没办法，只有升级机器。把机器配置升级到 4C8G，给 JVM 分配 6GB 的内存，这样 OOM 问题就消失了。但随之而来的是频繁的 GC 问题和超长的 GC 时间，平均 GC 时间竟然有 5 秒多。

进一步，由于报表系统和高并发系统不太一样，它的对象，存活时长大得多，并不能仅仅通过增加年轻代来解决；而且，如果增加了年轻代，那么必然减少了老年代的大小，由于 CMS 的碎片和浮动垃圾问题，我们可用的空间就更少了。虽然服务能够满足目前的需求，但还有一些不太确定的风险。

第一，了解到程序中有很多缓存数据和静态统计数据，为了减少 MinorGC 的次数，通过分析 GC 日志打印的对象年龄分布，把 MaxTenuringThreshold 参数调整到了 3（特殊场景特殊的配置）。这个参数是让年轻代的这些对象，赶紧回到老年代去，不要老呆在年轻代里。

第二，我们的 GC 时间比较长，就一块开了参数 CMSScavengeBeforeRemark，使得在 CMS remark 前，先执行一次 Minor GC 将新生代清掉。同时配合上个参数，其效果还是比较好的，一方面，对象很快晋升到了老年代，另一方面，年轻代的对象在这种情况下是有限的，在整个 MajorGC 中占的时间也有限。

第三，由于缓存的使用，有大量的弱引用，拿一次长达 10 秒的 GC 来说。我们发现在 GC 日志里，处理 weak refs 的时间较长，达到了 4.5 秒。这里可以加入参数 ParallelRefProcEnabled 来并行处理Reference，以加快处理速度，缩短耗时。

优化之后，效果不错，但并不是特别明显。经过评估，针对高峰时期的情况进行调研，我们决定再次提升机器性能，改用 8core16g 的机器。但是，这带来另外一个问题。

高性能的机器带来了非常大的服务吞吐量，通过 jstat 进行监控，能够看到年轻代的分配速率明显提高，但随之而来的 MinorGC 时长却变的不可控，有时候会超过 1 秒。累积的请求造成了更加严重的后果。

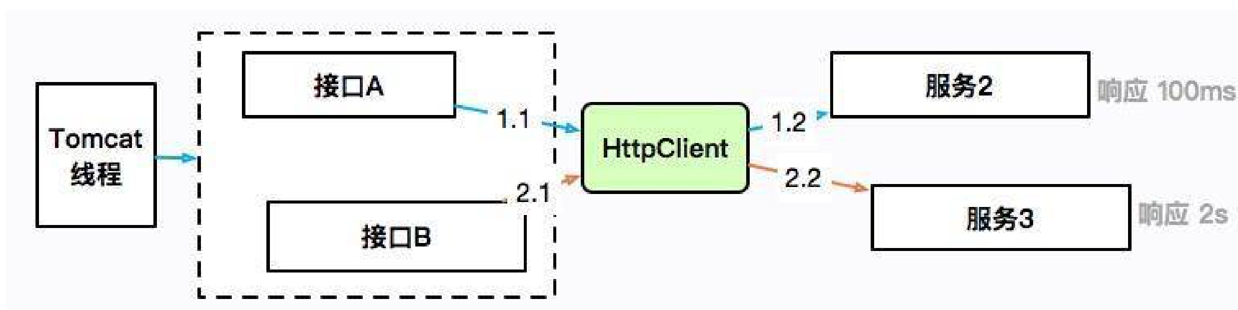
这是由于堆空间明显加大造成的回收时间加长。为了获取较小的停顿时间，我们在堆上改用了 G1 垃圾回收器，把它的目标设定在 200ms。G1 是一款非常优秀的垃圾收集器，不仅适合堆内存大的应用，同时也简化了调优的工作。通过主要的参数初始和最大堆空间、以及最大容忍的 GC 暂停目标，就能得到不错的性能。修改之后，虽然 GC 更加频繁了一些，但是停顿时间都比较小，应用的运行较为平滑。

到目前为止，也只是勉强顶住了已有的业务，但是，这时候领导层面又发力，要求报表系统可以支持未来两年业务10到100倍的增长，并保持其可用性，但是这个“千疮百孔”的报表系统，稍微一压测，就宕机，那如何应对十倍百倍的压力呢？硬件即使可以做到动态扩容，但是毕竟也有极限。

使用 MAT 分析堆快照，发现很多地方可以通过代码优化，那些占用内存特别多的对象：
1、select * 全量排查，只允许获取必须的数据
2、报表系统中cache实际的命中率并不高，将Guava 的 Cache 引用级别改成弱引用（WeakKeys）
3、限制报表导入文件大小，同时拆分用户超大范围查询导出请求。
每一步操作都使得JVM使用变得更加可用，一系列优化以后，机器相同压测数据性能提升了数倍。

3. 大屏异常 | JUC调优

有些数据需要使用 HttpClient 来获取进行补全。提供数据的服务提供商有的响应时间可能会很长，也有可能会造成服务整体的阻塞。



接口 A 通过 HttpClient 访问服务 2，响应 100ms 后返回；接口 B 访问服务 3，耗时 2 秒。HttpClient 本身是有一个最大连接数限制的，如果服务 3 迟迟不返回，就会造成 HttpClient 的连接数达到上限，概括来讲，就是同一服务，由于一个耗时非常长的接口，进而引起了整体的服务不可用

这个时候，通过 jstack 打印栈信息，会发现大多数竟然阻塞在了接口 A 上，而不是耗时更长的接口 B，这个现象起初十分具有迷惑性，不过经过分析后，我们猜想其实是因为接口 A 的速度比较快，在问题发生点进入了更多的请求，它们全部都阻塞住的同时被打印出来了。

为了验证这个问题，我搭建了一个demo 工程，模拟了两个使用同一个 HttpClient 的接口。fast 接口用来访问百度，很快就能返回；slow 接口访问谷歌，由于众所周知的原因，会阻塞直到超时，大约 10 s。 利用ab对两个接口进行压测，同时使用 jstack 工具 dump 堆栈。首先使用 jps 命令找到进程号，然后把结果重定向到文件（可以参考 10271.jstack 文件）。

过滤一下 nio 关键字，可以查看 tomcat 相关的线程，足足有 200 个，这和 Spring Boot 默认的 maxThreads 个数不谋而合。更要命的是，有大多数线程，都处于 BLOCKED 状态，说明线程等待资源超时。通过grep fast | wc –l 分析，确实200个中有150个都是blocked的fast的进程。

问题找到了，解决方式就顺利成章了。

1、fast和slow争抢连接资源，通过线程池限流或者熔断处理

2、有时候slow的线程也不是一直slow，所以就得加入监控

3、使用带countdownLaunch对线程的执行顺序逻辑进行控制

4. 接口延迟 | SWAP调优

有一个关于服务的某个实例，经常发生服务卡顿。由于服务的并发量是比较高的，每多停顿 1 秒钟，几万用户的请求就会感到延迟。

我们统计、类比了此服务其他实例的 CPU、内存、网络、I/O 资源，区别并不是很大，所以一度怀疑是机器硬件的

问题。

接下来我们对比了节点的 GC 日志，发现无论是 Minor GC，还是 Major GC，这个节点所花费的时间，都比其他实例长得多。

通过仔细观察，我们发现在 GC 发生的时候，vmstat 的 si、so 飙升的非常严重，这和其他实例有着明显的不同。

使用 free 命令再次确认，发现 SWAP 分区，使用的比例非常高，引起的具体原因是什么呢?

更详细的操作系统内存分布，从 /proc/meminfo 文件中可以看到具体的逻辑内存块大小，有多达 40 项的内存信息，这些信息都可以通过遍历 /proc 目录的一些文件获取。我们注意到 slabtop 命令显示的有一些异常，dentry（目录高速缓冲）占用非常高。

问题最终定位到是由于某个运维工程师删除日志时，定时执行了一句命令：

find / | grep "xxx.log"

他是想找一个叫做 要被删除 的日志文件，看看在哪台服务器上，结果，这些老服务器由于文件太多，扫描后这些文件信息都缓存到了 slab 区上。而服务器开了 swap，操作系统发现物理内存占满后，并没有立即释放 cache，导致每次 GC 都要和硬盘打一次交道。

解决方式就是关闭 SWAP 分区。

swap 是很多性能场景的万恶之源，建议禁用。在高并发 SWAP 绝对能让你体验到它魔鬼性的一面：进程倒是死不了了，但 GC 时间长的却让人无法忍受。

5. 内存溢出 | Cache调优

> 有一次线上遇到故障，重新启动后，使用 jstat 命令，发现 Old 区一直在增长。我使用 jmap 命令，导出了一份线上堆栈，然后使用 MAT 进行分析，通过对 GC Roots 的分析，发现了一个非常大的 HashMap 对象，这个原本是其他同事做缓存用的，但是做了一个无界缓存，没有设置超时时间或者 LRU 策略，在使用上又没有重写key类对象的hashcode和equals方法，对象无法取出也直接造成了堆内存占用一直上升，后来，将这个缓存改成 guava 的 Cache，并设置了弱引用，故障就消失了。
>
> 关于文件处理器的应用，在读取或者写入一些文件之后，由于发生了一些异常，close 方法又没有放在 finally 块里面，造成了文件句柄的泄漏。由于文件处理十分频繁，产生了严重的内存泄漏问题。

内存溢出是一个结果，而内存泄漏是一个原因。内存溢出的原因有内存空间不足、配置错误等因素。一些错误的编程方式，不再被使用的对象、没有被回收、没有及时切断与 GC Roots 的联系，这就是内存泄漏。

举个例子，有团队使用了 HashMap 做缓存，但是并没有设置超时时间或者 LRU 策略，造成了放入 Map 对象的数据越来越多，而产生了内存泄漏。

再来看一个经常发生的内存泄漏的例子，也是由于 HashMap 产生的。代码如下，由于没有重写 Key 类的 hash-Code 和 equals 方法，造成了放入 HashMap 的所有对象都无法被取出来，它们和外界失联了。所以下面的代码结果是 null。

```
//leak example
import java.util.HashMap;
import java.util.Map;
public class HashMapLeakDemo {
    public static class Key {
        String title;
    public Key(String title) {
        this.title = title;
    }
}

public static void main(String[] args) {
    Map<Key, Integer> map = new HashMap<>();
    map.put(new Key("1"), 1);
    map.put(new Key("2"), 2);
    map.put(new Key("3"), 2);
    Integer integer = map.get(new Key("2"));
    System.out.println(integer);
    }
}
```

即使提供了 equals 方法和 hashCode 方法，也要非常小心，尽量避免使用自定义的对象作为 Key。

再看一个例子，关于文件处理器的应用，在读取或者写入一些文件之后，由于发生了一些异常，close 方法又没有放在 finally 块里面，造成了文件句柄的泄漏。由于文件处理十分频繁，产生了严重的内存泄漏问题。

6. CPU飙高 | 死循环

我们有个线上应用，单节点在运行一段时间后，CPU 的使用会飙升，一旦飙升，一般怀疑某个业务逻辑的计算量太大，或者是触发了死循环（比如著名的 HashMap 高并发引起的死循环），但排查到最后其实是 GC 的问题。

（1）使用 top 命令，查找到使用 CPU 最多的某个进程，记录它的 pid。使用 Shift + P 快捷键可以按 CPU 的使用率进行排序。

```
top
```

（2）再次使用 top 命令，加 -H 参数，查看某个进程中使用 CPU 最多的某个线程，记录线程的 ID。

```
top -Hp $pid
```

（3）使用 printf 函数，将十进制的 tid 转化成十六进制。

```
printf %x $tid
```

（4）使用 jstack 命令，查看 Java 进程的线程栈。

```
jstack $pid >$pid.log
```

（5）使用 less 命令查看生成的文件，并查找刚才转化的十六进制 tid，找到发生问题的线程上下文。

```
less $pid.log
```

我们在 jstack 日志搜关键字DEAD，以及中找到了 CPU 使用最多的几个线程id。

可以看到问题发生的根源，是我们的堆已经满了，但是又没有发生 OOM，于是 GC 进程就一直在那里回收，回收的效果又非常一般，造成 CPU 升高应用假死。接下来的具体问题排查，就需要把内存 dump 一份下来，使用 MAT 等工具分析具体原因了。

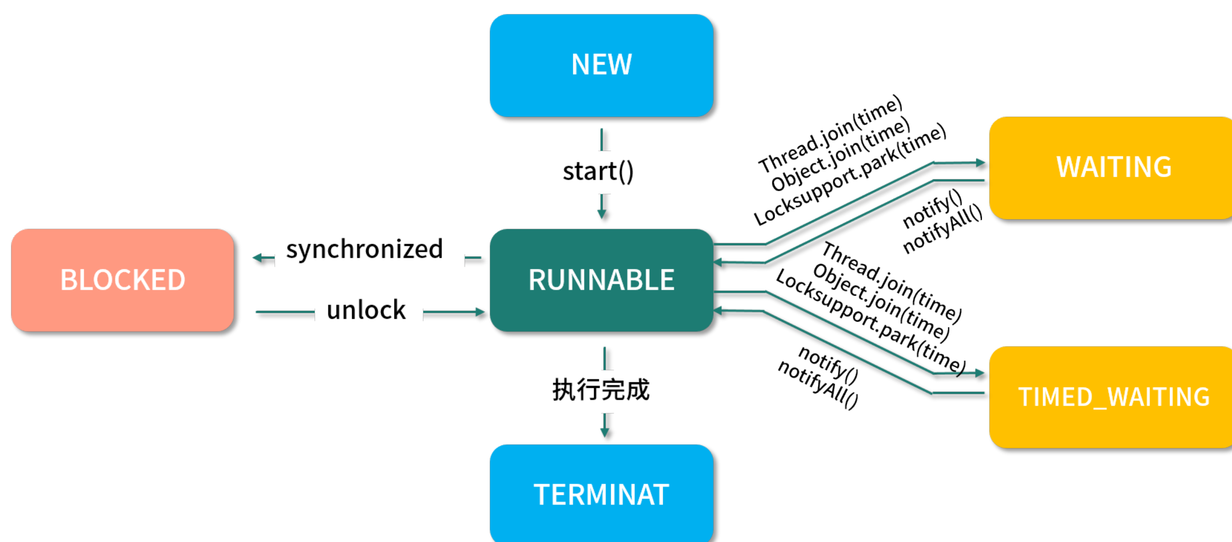
多线程篇

线程调度

1. 线程状态

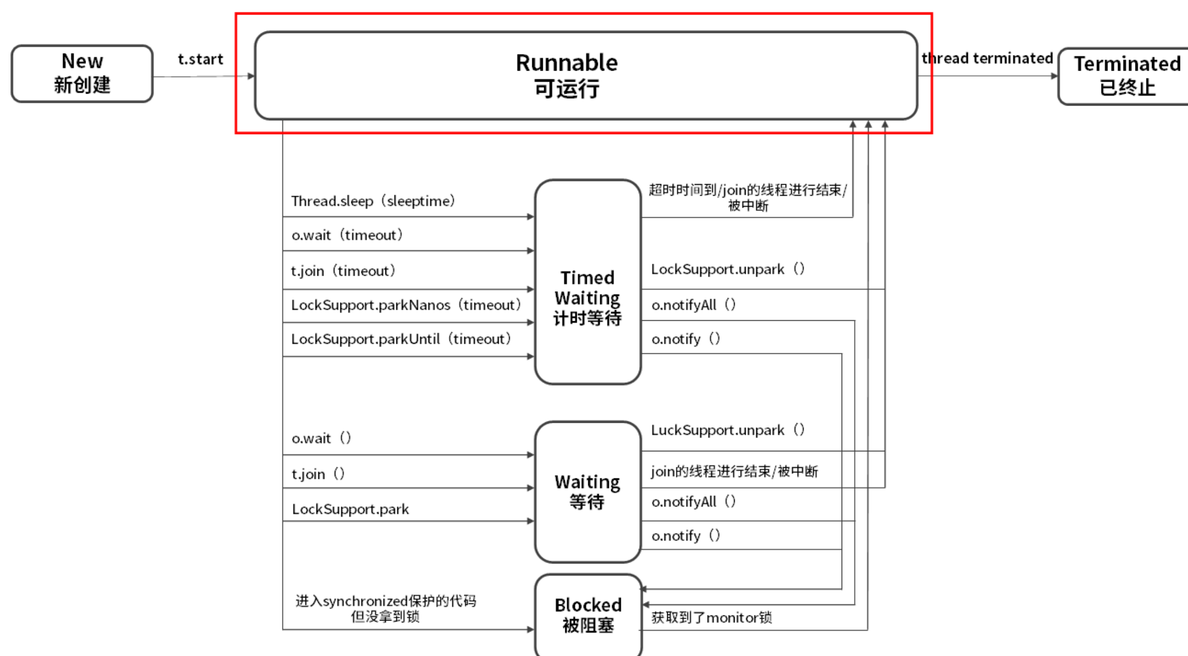
线程是cpu任务调度的最小执行单位，每个线程拥有自己独立的程序计数器、虚拟机栈、本地方法栈

线程状态：创建、就绪、运行、阻塞、死亡



2. 线程状态切换

方法	作用	区别
start	启动线程，由虚拟机自动调度执行run()方法	线程处于就绪状态
run	线程逻辑代码块处理，JVM调度执行	线程处于运行状态
sleep	让当前正在执行的线程休眠（暂停执行）	不释放锁
wait	使得当前线程等待	释放同步锁
notify	唤醒在此对象监视器上等待的单个线程	唤醒单个线程
notifyAll	唤醒在此对象监视器上等待的所有线程	唤醒多个线程
yiled	停止当前线程，让同等优先权的线程运行	用Thread类调用
join	使当前线程停下来等待，直至另一个调用join方法的线程终止	用线程对象调用



3. 阻塞唤醒过程

阻塞

这三个方法的调用都会使当前线程阻塞。该线程将会被放置到对该Object的请求等待队列中，然后让出当前对Object所拥有的所有的同步请求。线程会一直暂停所有线程调度，直到下面其中一种情况发生:

① 其他线程调用了该Object的notify方法，而该线程刚好是那个被唤醒的线程;

② 其他线程调用了该Object的notifyAll方法;

唤醒

线程将会从等待队列中移除，重新成为可调度线程。它会与其他线程以常规的方式竞争对象同步请求。一旦它重新获得对象的同步请求，所有之前的请求状态都会恢复，也就是线程调用wait的地方的状态。线程将会在之前调用wait的地方继续运行下去。

为什么要出现在同步代码块中

由于wait()属于Object方法，调用之后会强制释放当前对象锁，所以在wait()　调用时必须拿到当前对象的监视器monitor对象。因此，wait()方法在同步方法/代码块中调用。

4. wait和sleep区别

- wait 方法必须在 synchronized 保护的代码中使用，而 sleep 方法并没有这个要求。
- wait 方法会主动释放 monitor 锁，在同步代码中执行 sleep 方法时，并不会释放 monitor 锁。
- wait 方法意味着永久等待，直到被中断或被唤醒才能恢复，不会主动恢复，sleep 方法中会定义一个时间，时间到期后会主动恢复。
- wait/notify 是 Object 类的方法，而 sleep 是 Thread 类的方法。

5. 创建线程方式

实现 Runnable 接口（优先使用）

```
public class RunnableThread implements Runnable {
    @Override
    public void run() {System.out.println('用实现Runnable接口实现线程');}
}
```

实现Callable接口（有返回值可抛出异常）

```
class CallableTask implements Callable<Integer> {
    @Override
    public Integer call() throws Exception { return new Random().nextInt();}
}
```

继承Thread类（java不支持多继承）

```
public class RunnableThread implements Runnable {
    @Override
    public void run() {System.out.println('用实现Runnable接口实现线程');}
}
```

使用线程池（底层都是实现run方法）

```
static class DefaultThreadFactory implements ThreadFactory {
    DefaultThreadFactory() {
        SecurityManager s = System.getSecurityManager();
        group = (s != null) ? s.getThreadGroup() : Thread.currentThread().getThreadGroup();
        namePrefix = "pool-" + poolNumber.getAndIncrement() +"-thread-";
    }
    public Thread newThread(Runnable r) {
        Thread t = new Thread(group, r,namePrefix + threadNumber.getAndIncrement(),0);
        if (t.isDaemon()) t.setDaemon(false);  //是否守护线程
        if (t.getPriority() != Thread.NORM_PRIORITY) t.setPriority(Thread.NORM_PRIORITY); //线程优先级
        return t;
    }
}
```

线程池

优点：通过复用已创建的线程，降低资源损耗、线程可以直接处理队列中的任务加快响应速度、同时便于统一监控和管理。

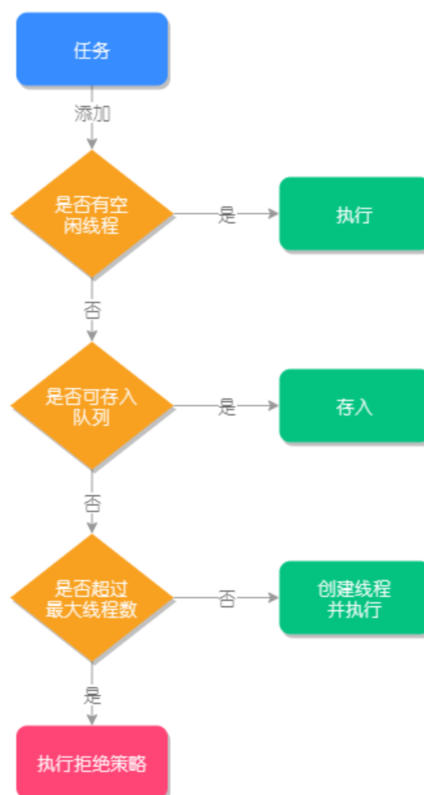
1. 线程池构造函数

```
/**
* 线程池构造函数7大参数
*/
public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,
    TimeUnit unit,BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory,
    RejectedExecutionHandler handler) {}
```

参数介绍

方法	作用
corePoolSize	核心线程池大小
maximumPoolSize	最大线程池大小
keepAliveTime	线程池中超过 corePoolSize 数目的空闲线程最大存活时间；
TimeUnit	keepAliveTime 时间单位
workQueue	阻塞任务队列
threadFactory	新建线程工厂
RejectedExecutionHandler	拒绝策略。当提交任务数超过 maxmumPoolSize+workQueue 之和时，任务会交给RejectedExecutionHandler 来处理

2. 线程处理任务过程：



1. 当线程池小于corePoolSize，新提交任务将创建一个新线程执行任务，即使此时线程池中存在空闲线程。
2. 当线程池达到corePoolSize时，新提交任务将被放入 workQueue 中，等待线程池中任务调度执行。
3. 当workQueue已满，且 maximumPoolSize 大于 corePoolSize 时，新提交任务会创建新线程执行任务。
4. 当提交任务数超过 maximumPoolSize 时，新提交任务由 RejectedExecutionHandler 处理。
5. 当线程池中超过corePoolSize 线程，空闲时间达到 keepAliveTime 时，关闭空闲线程 。

3. 线程拒绝策略

线程池中的线程已经用完了，无法继续为新任务服务，同时，等待队列也已经排满了，再也塞不下新任务了。这时候我们就需要拒绝策略机制合理的处理这个问题。

JDK 内置的拒绝策略如下:
AbortPolicy：直接抛出异常，阻止系统正常运行。可以根据业务逻辑选择重试或者放弃提交等策略。
CallerRunsPolicy ： 只要线程池未关闭，该策略直接在调用者线程中，运行当前被丢弃的任务。
不会造成任务丢失，同时减缓提交任务的速度，给执行任务缓冲时间。
DiscardOldestPolicy ： 丢弃最老的一个请求，也就是即将被执行的任务，并尝试再次提交当前任务。
DiscardPolicy ： 该策略默默地丢弃无法处理的任务，不予任何处理。如果允许任务丢失，这是最好的一种方案。

4. Execuors类实现线程池

FixedThreadPool	LinkedBlockingQueue
SingleThreadExecutor	LinkedBlockingQueue
CachedThreadPool	SynchronousQueue
ScheduledThreadPool	DelayedWorkQueue
SingleThreadScheduledExecutor	DelayedWorkQueue

- newSingleThreadExecutor()：只有一个线程的线程池，任务是顺序执行，适用于一个一个任务执行的场景
- newCachedThreadPool()：线程池里有很多线程需要同时执行，60s内复用，适用执行很多短期异步的小程序或者负载较轻的服务

- newFixedThreadPool()：拥有固定线程数的线程池，如果没有任务执行，那么线程会一直等待，适用执行长期的任务。
- newScheduledThreadPool()：用来调度即将执行的任务的线程池
- newWorkStealingPool()：底层采用forkjoin的Deque，采用独立的任务队列可以减少竞争同时加快任务处理

参数	FixedThreadPool	CachedThreadPool	ScheduledThreadpool	SingleThreadExecutor	SingleThreadScheduledExecutor
corePoolSize	构造函数传入	0	构造函数传入	1	1
maxPoolSize	同corePoolSize	Integer.MAX_VALUE	Integer.MAX_VALUE	1	Integer.MAX_VALUE
keepAliveTime	0	60秒	0	0	0

因为以上方式都存在弊端

FixedThreadPool 和 SingleThreadExecutor： 允许请求的队列长度为 Integer.MAXVALUE，会导致OOM。
CachedThreadPool 和 ScheduledThreadPool ： 允许创建的线程数量为 Integer.MAXVALUE，会导致OOM。

手动创建的线程池底层使用的是ArrayBlockingQueue可以防止OOM。

5. 线程池大小设置

- CPU 密集型（n+1）

  CPU 密集的意思是该任务需要大量的运算，而没有阻塞，CPU 一直全速运行。

  CPU 密集型任务尽可能的少的线程数量，一般为 CPU 核数 + 1 个线程的线程池。

- IO 密集型（2*n）

  由于 IO 密集型任务线程并不是一直在执行任务，可以多分配一点线程数，如 CPU * 2

  也可以使用公式：CPU 核心数 *（1+平均等待时间/平均工作时间）。

线程安全

1. 乐观锁，CAS思想

java乐观锁机制

乐观锁体现的是悲观锁的反面。它是一种积极的思想，它总是认为数据是不会被修改的，所以是不会对数据上锁的。但是乐观锁在更新的时候会去判断数据是否被更新过。乐观锁的实现方案一般有两种（版本号机制和CAS）。乐观锁适用于读多写少的场景，这样可以提高系统的并发量。在Java中 java.util.concurrent.atomic下的原子变量类就是使用了乐观锁的一种实现方式CAS实现的。

乐观锁，大多是基于数据版本 (Version)记录机制实现。即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个 “version” 字段来 实现。 读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提 交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如果提交的数据 版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

CAS思想

CAS就是compare and swap（比较交换），是一种很出名的无锁的算法，就是可以不使用锁机制实现线程间的同步。使用CAS线程是不会被阻塞的，所以又称为非阻塞同步。CAS算法涉及到三个操作：
需要读写内存值V；进行比较的值A；准备写入的值B
当且仅当V的值等于A的值等于V的值的时候，才用B的值去更新V的值，否则不会执行任何操作（比较和替换是一个原子操作-A和V比较，V和B替换），一般情况下是一个自旋操作，即不断重试

缺点

ABA问题-知乎
高并发的情况下，很容易发生并发冲突，如果CAS一直失败，那么就会一直重试，浪费CPU资源

原子性

功能限制CAS是能保证单个变量的操作是原子性的，在Java中要配合使用volatile关键字来保证线程的安全；当涉及到多个变量的时候CAS无能为力；除此之外CAS实现需要硬件层面的支持，在Java的普通用户中无法直接使用，只能借助atomic包下的原子类实现，灵活性受到了限制

2、synchronized底层实现

使用方法：主要的三种使用方式
修饰实例方法: 作用于当前对象实例加锁，进入同步代码前要获得当前对象实例的锁

修饰静态方法: 也就是给当前类加锁，会作用于类的所有对象实例，因为静态成员不属于任何一个实例对象，是类成员。
修饰代码块: 指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。
总结: synchronized锁住的资源只有两类：一个是对象，一个是类。

底层实现:
对象头是我们需要关注的重点，它是synchronized实现锁的基础，因为synchronized申请锁、上锁、释放锁都与对象头有关。对象头主要结构是由Mark Word 组成，其中Mark Word存储对象的hashCode、锁信息或分代年龄或GC标志等信息。
锁也分不同状态，JDK6之前只有两个状态：无锁、有锁（重量级锁），而在JDK6之后对synchronized进行了优化，新增了两种状态，总共就是四个状态：无锁状态、偏向锁、轻量级锁、重量级锁，其中无锁就是一种状态了。锁的类型和状态在对象头Mark Word中都有记录，在申请锁、锁升级等过程中JVM都需要读取对象的Mark Word数据。
同步代码块是利用 monitorenter 和 monitorexit 指令实现的，而同步方法则是利用 flags 实现的。

3. ReenTrantLock底层实现

由于ReentrantLock是java.util.concurrent包下提供的一套互斥锁，相比Synchronized，ReentrantLock类提供了一些高级功能

使用方法:
基于API层面的互斥锁，需要lock()和unlock()方法配合try/finally语句块来完成

底层实现:
ReenTrantLock的实现是一种自旋锁，通过循环调用CAS操作来实现加锁。它的性能比较好也是因为避免了使线程进入内核态的阻塞状态。想尽办法避免线程进入内核的阻塞状态是我们去分析和理解锁设计的关键钥匙。

和synchronized区别:
1. 底层实现：synchronized 是JVM层面的锁，是Java关键字，通过monitor对象来完成（monitorenter与monitorexit），ReentrantLock 是从jdk1.5以来（java.util.concurrent.locks.Lock）提供的API层面的锁。
2. 实现原理：synchronized 的实现涉及到锁的升级，具体为无锁、偏向锁、自旋锁、向OS申请重量级锁；ReentrantLock实现则是通过利用CAS**（CompareAndSwap）自旋机制保证线程操作的原子性和volatile保证数据可见性以实现锁的功能。
3. 是否可手动释放：synchronized 不需要用户去手动释放锁，synchronized 代码执行完后系统会自动让线程释放对锁的占用； ReentrantLock则需要用户去手动释放锁，如果没有手动释放锁，就可能导致死锁现象。
4. 是否可中断synchronized是不可中断类型的锁，除非加锁的代码中出现异常或正常执行完成； ReentrantLock则可以中断，可通过trylock(long timeout,TimeUnit unit)设置超时方法或者将lockInterruptibly()放到代码块中，调用interrupt方法进行中断。

5. 是否公平锁synchronized为非公平锁　ReentrantLock则即可以选公平锁也可以选非公平锁，通过构造方法new ReentrantLock时传入boolean值进行选择，为空默认false非公平锁，true为公平锁,公平锁性能非常低。

4. 公平锁和非公平锁区别

公平锁:

公平锁自然是遵循FIFO（先进先出）原则的，先到的线程会优先获取资源，后到的会进行排队等待

优点：所有的线程都能得到资源，不会饿死在队列中。适合大任务

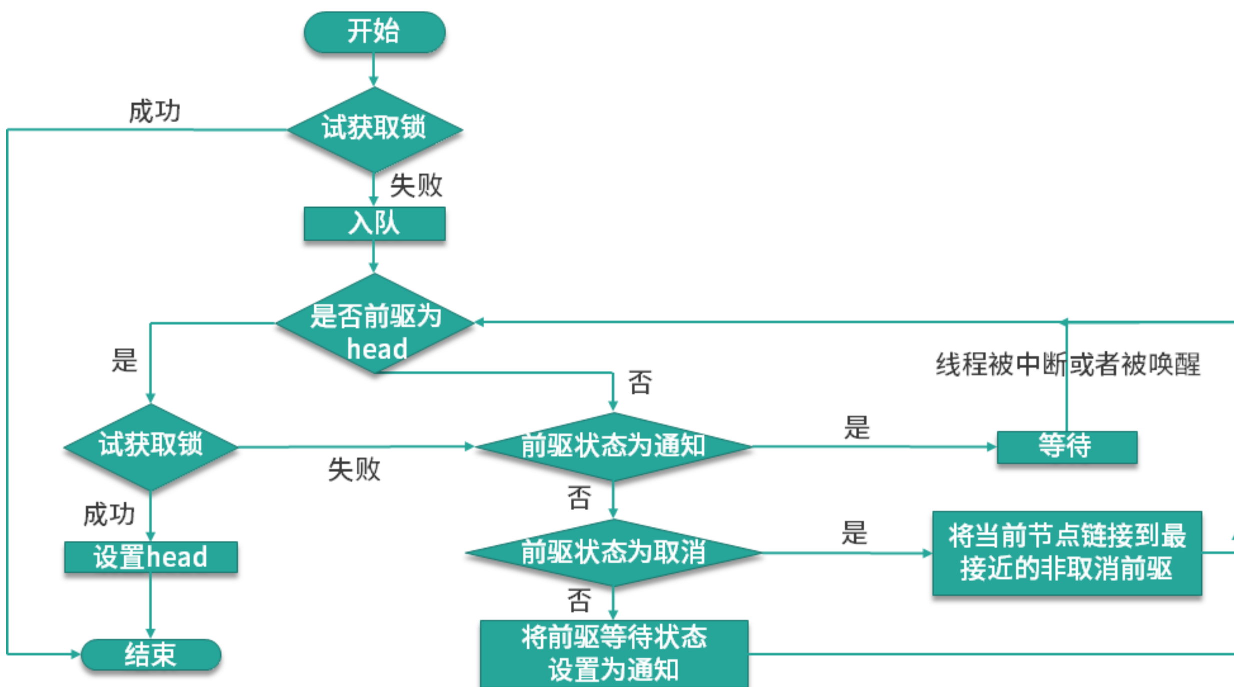
缺点：吞吐量会下降，队列里面除了第一个线程，其他的线程都会阻塞，cpu唤醒阻塞线程的开销大

非公平锁:

多个线程去获取锁的时候，会直接去尝试获取，获取不到，再去进入等待队列，如果能获取到，就直接获取到锁。

优点：可以减少CPU唤醒线程的开销，整体的吞吐效率会高点，CPU也不必取唤醒所有线程，会减少唤起线程的数量。

缺点：你们可能也发现了，这样可能导致队列中间的线程一直获取不到锁或者长时间获取不到锁



公平锁效率低原因:

公平锁要维护一个队列，后来的线程要加锁，即使锁空闲，也要先检查有没有其他线程在　wait，如果有自己要挂起，加到队列后面，然后唤醒队列最前面线程。这种情况下相比较非公平锁多了一次挂起和唤醒。

线程切换的开销，其实就是非公平锁效率高于公平锁的原因，因为非公平锁减少了线程挂起的几率，后来的线程有一定几率逃离被挂起的开销。

5. 使用层面锁优化

【1】减少锁的时间：

不需要同步执行的代码，能不放在同步快里面执行就不要放在同步快内，可以让锁尽快释放；

【2】减少锁的粒度：

它的思想是将物理上的一个锁，拆成逻辑上的多个锁，增加并行度，从而降低锁竞争。它的思想也是用空间来换时间；java中很多数据结构都是采用这种方法提高并发操作的效率，比如：

ConcurrentHashMap：

java中的ConcurrentHashMap在jdk1.8之前的版本，使用一个Segment 数组：Segment< K,V >[] segments Segment继承自ReenTrantLock，所以每个Segment是个可重入锁，每个Segment 有一个HashEntry< K,V >数组用来存放数据，put操作时，先确定往哪个Segment放数据，只需要锁定这个Segment，执行put，其它的Segment不会被锁定；所以数组中有多少个Segment就允许同一时刻多少个线程存放数据，这样增加了并发能力。

【3】锁粗化：

大部分情况下我们是要让锁的粒度最小化，锁的粗化则是要增大锁的粒度；
假如有一个循环，循环内的操作需要加锁，我们应该把锁放到循环外面，否则每次进出循环，都进出一次临界区，效率是非常差的；

【4】使用读写锁：

ReentrantReadWriteLock 是一个读写锁，读操作加读锁，可并发读，写操作使用写锁，只能单线程写；

【5】使用CAS：

如果需要同步的操作执行速度非常快，并且线程竞争并不激烈，这时候使用cas效率会更高，因为加锁会导致线程的上下文切换，如果上下文切换的耗时比同步操作本身更耗时，且线程对资源的竞争不激烈，使用volatiled+cas操作会是非常高效的选择；

6. 系统层面锁优化

自适应自旋锁：

自旋锁可以避免等待竞争锁进入阻塞挂起状态被唤醒造成的内核态和用户态之间的切换的损耗，它们只需要等一等（自旋），但是如果锁被其他线程长时间占用，一直不释放CPU，死等会带来更多的性能开销；自旋次数默认值是10
对上面自旋锁优化方式的进一步优化，它的自旋的次数不再固定，其自旋的次数由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定，这就解决了自旋锁带来的缺点

锁消除：

锁削除是指虚拟机即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行削除。Netty中无锁化设计pipeline中channelhandler会进行锁消除的优化。

锁升级：

偏向锁：

如果线程已经占有这个锁，当他在次试图去获取这个锁的时候，他会已最快的方式去拿到这个锁，而不需要在进行一些monitor操作，因为在大部分情况下是没有竞争的，所以使用偏向锁是可以提高性能的；

轻量级锁：

在竞争不激烈的情况下，通过CAS避免线程上下文切换，可以显著的提高性能。

重量级锁：

重量级锁的加锁、解锁过程造成的损耗是固定的，重量级锁适合于竞争激烈、高并发、同步块执行时间长的情况。

7. ThreadLocal原理

ThreadLocal简介：

通常情况下，我们创建的变量是可以被任何一个线程访问并修改的。如果想实现每一个线程都有自己的 专属本地变量该如何解决呢？ JDK中提供的 ThreadLocal 类正是为了解决这样的问题。类似操作系统中的TLAB

原理：

首先 ThreadLocal 是一个泛型类，保证可以接受任何类型的对象。因为一个线程内可以存在多个 ThreadLocal 对象，所以其实是 ThreadLocal 内部维护了一个 Map ，是 ThreadLocal 实现的一个叫做 ThreadLocalMap 的静态内部类。

最终的变量是放在了当前线程的 ThreadLocalMap 中，并不是存在 ThreadLocal 上，ThreadLocal 可以理解为只是ThreadLocalMap的封装，传递了变量值。

我们使用的 get()、set() 方法其实都是调用了这个ThreadLocalMap类对应的 get()、set() 方法。例如下面的

如何使用：

1）存储用户Session

```
private static final ThreadLocal threadSession = new ThreadLocal();
```

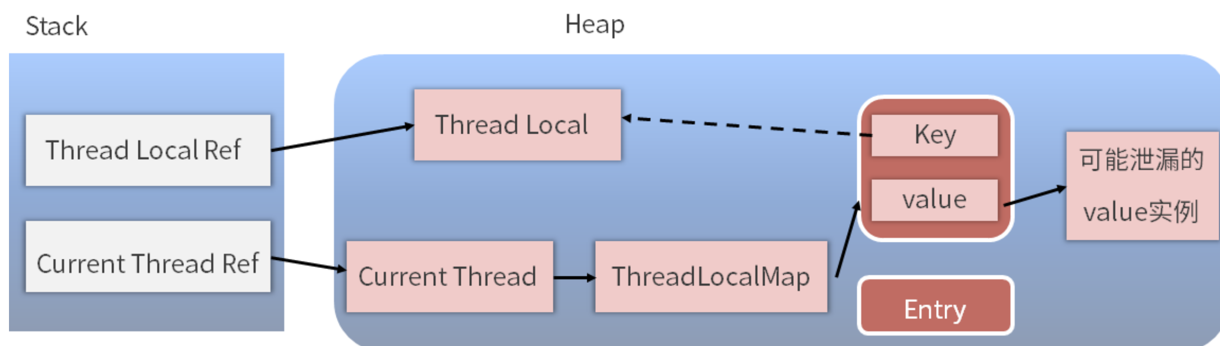
2）解决线程安全的问题

```
private static ThreadLocal<SimpleDateFormat> format1 = new ThreadLocal<SimpleDateFormat>()
```

ThreadLocal内存泄漏的场景

实际上 ThreadLocalMap 中使用的 key 为 ThreadLocal 的弱引用，而 value 是强引用。弱引用的特点是，如果这个对象持有弱引用，那么在下一次垃圾回收的时候必然会被清理掉。

所以如果 ThreadLocal 没有被外部强引用的情况下，在垃圾回收的时候会被清理掉的，这样一来 ThreadLocalMap中使用这个 ThreadLocal 的 key 也会被清理掉。但是，value 是强引用，不会被清理，这样一来就会出现 key 为 null 的 value。 假如我们不做任何措施的话，value 永远无法被GC 回收，如果线程长时间不被销毁，可能会产生内存泄露。



ThreadLocalMap实现中已经考虑了这种情况，在调用 set()、get()、remove() 方法的时候，会清理掉 key 为 null 的记录。如果说会出现内存泄漏，那只有在出现了 key 为 null 的记录后，没有手动调用 remove() 方法，并且之后也不再调用 get()、set()、remove()方法的情况下。因此使用完ThreadLocal方法后，最好手动调用 remove () 方法。

8. HashMap线程安全

死循环造成 CPU 100%

HashMap 有可能会发生死循环并且造成 CPU 100% ，这种情况发生最主要的原因就是在扩容的时候，也就是内部新建新的 HashMap 的时候，扩容的逻辑会反转散列桶中的节点顺序，当有多个线程同时进行扩容的时候，由于 HashMap 并非线程安全的，所以如果两个线程同时反转的话，便可能形成一个循环，并且这种循环是链表的循环，相当于 A 节点指向 B 节点，B 节点又指回到 A 节点，这样一来，在下一次想要获取该 key 所对应的 value 的时候，便会在遍历链表的时候发生永远无法遍历结束的情况，也就发生 CPU 100% 的情况。

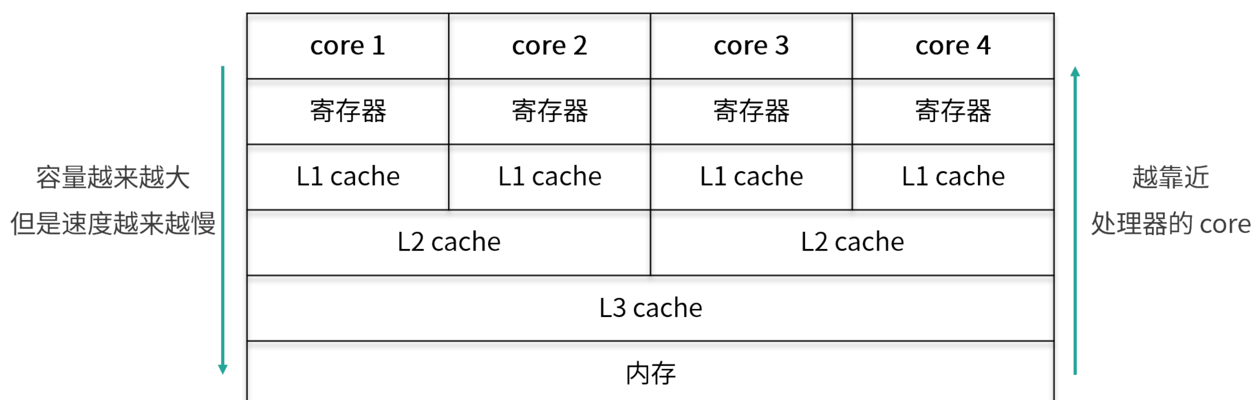
所以综上所述，HashMap 是线程不安全的，在多线程使用场景中推荐使用线程安全同时性能比较好的 ConcurrentHashMap。

9. String不可变原因

1. 可以使用字符串常量池，多次创建同样的字符串会指向同一个内存地址
2. 可以很方便地用作 HashMap 的 key。通常建议把不可变对象作为 HashMap的 key
3. hashCode生成后就不会改变，使用时无需重新计算
4. 线程安全，因为具备不变性的对象一定是线程安全的

内存模型

Java 内存模型（Java Memory Model，JMM）就是一种符合内存模型规范的，屏蔽了各种硬件和操作系统的访问差异的，保证了 Java 程序在各种平台下对内存的访问都能保证效果一致的机制及规范。



JMM 是一种规范，是解决由于多线程通过共享内存进行通信时，存在的本地内存数据不一致、编译器会对代码指令重排序、处理器会对代码乱序执行等带来的问题。目的是保证并发编程场景中的原子性、可见性和有序性。

原子性：

在 Java 中，为了保证原子性，提供了两个高级的字节码指令 Monitorenter 和 Monitorexit。这两个字节码，在 Java 中对应的关键字就是 Synchronized。因此，在 Java 中可以使用 Synchronized 来保证方法和代码块内的操作是原子性的。

可见性：

Java 中的 Volatile 关键字修饰的变量在被修改后可以立即同步到主内存。被其修饰的变量在每次使用之前都从主内存刷新。因此，可以使用 Volatile 来保证多线程操作时变量的可见性。除了 Volatile，Java 中的 Synchronized 和 Final 两个关键字也可以实现可见性。只不过实现方式不同

有序性

在 Java 中，可以使用 Synchronized 和 Volatile 来保证多线程之间操作的有序性。区别：Volatile 禁止指令重排。Synchronized 保证同一时刻只允许一条线程操作。

1. volatile底层实现

作用：

保证数据的“可见性”：被volatile修饰的变量能够保证每个线程能够获取该变量的最新值，从而避免出现数据脏读的现象。

禁止指令重排：在多线程操作情况下，指令重排会导致计算结果不一致

底层实现：

“观察加入volatile关键字和没有加入volatile关键字时所生成的汇编代码发现，加入volatile关键字时，会多出一个lock前缀指令”

lock前缀指令实际上相当于一个内存屏障（也成内存栅栏），内存屏障会提供3个功能：

1. 它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；
2. 它会强制将对缓存的修改操作立即写入主存；
3. 如果是写操作，它会导致其他CPU中对应的缓存行无效。

单例模式中volatile的作用：

防止代码读取到instance不为null时，instance引用的对象有可能还没有完成初始化。

```
class Singleton{
    private volatile static Singleton instance = null;   //禁止指令重排
    private Singleton() {

    }
    public static Singleton getInstance() {
        if(instance==null) { //减少加锁的损耗
            synchronized (Singleton.class) {
                if(instance==null) //确认是否初始化完成
                    instance = new Singleton();
            }
        }
        return instance;
    }
}
```

2. AQS思想

AQS的全称为（AbstractQueuedSynchronizer）抽象的队列式的同步器，是一个用来构建锁和同步器的框架，使用AQS能简单且高效地构造出应用广泛的大量的同步器，如：基于AQS实现的lock，　CountDownLatch、CyclicBarrier、Semaphore需解决的问题：

```
状态的原子性管理
线程的阻塞与解除阻塞
队列的管理
```

AQS核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源

设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制AQS是用CLH（虚拟的双向队列）队列锁实现的，即将暂时获取不到锁的线程加入到队列中。

lock：
是一种可重入锁，除了能完成 synchronized 所能完成的所有工作外，还提供了诸如可响应中断锁、可轮询锁请求、定时锁等避免多线程死锁的方法。默认为非公平锁，但可以初始化为公平锁； 通过方法 lock()与 unlock()来进行加锁与解锁操作；

CountDownLatch：
通过计数法（倒计时器），让一些线程堵塞直到另一个线程完成一系列操作后才被唤醒；该工具通常用来控制线程等待，它可以让某一个线程等待直到倒计时结束，再开始执行。具体可以使用countDownLatch.await()来等待结果。多用于多线程信息汇总。

CompletableFuture：
通过设置参数，可以完成CountDownLatch同样的多平台响应问题，但是可以针对其中部分返回结果做更加灵活的展示。

CyclicBarrier：
字面意思是可循环(Cyclic)使用的屏障（Barrier）。他要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活，线程进入屏障通过CyclicBarrier的await()方法。可以用于批量发送消息队列信息、异步限流。

Semaphore：
信号量主要用于两个目的，一个是用于多个共享资源的互斥作用，另一个用于并发线程数的控制。SpringHystrix限流的思想

3. happens-before

用来描述和可见性相关问题：如果第一个操作 happens-before 第二个操作，那么我们就说第一个操作对于第二个操作是可见的
常见的happens-before：volatile 、锁、线程生命周期。

MySQL篇

WhyMysql?

NoSQL数据库四大家族

- 列存储 Hbase
- K-V存储 Redis
- 图像存储 Neo4j
- 文档存储 MongoDB

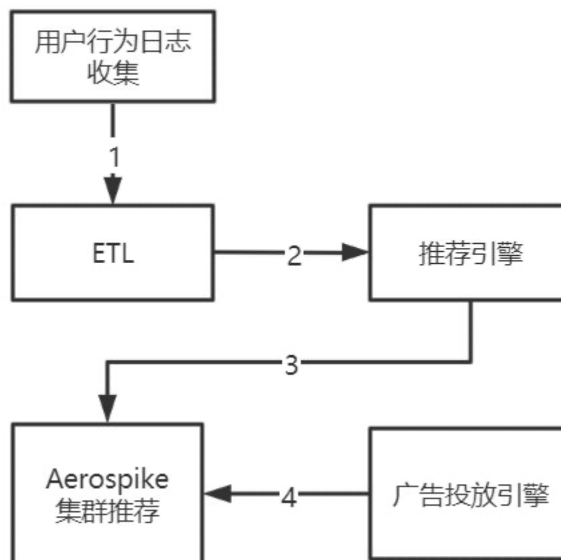
云存储OSS

海量Aerospike

Aerospike（简称AS）是一个分布式，可扩展的键值存储的NoSQL数据库。T级别大数据高并发的结构化数据存储，采用混合架构，索引存储在内存中，而数据可存储在机械硬盘(HDD)或固态硬盘(SSD) 上，读写操作达微妙级，99%的响应可在1毫秒内实现。

	Aerospike	Redis
类型	Nosql数据库	缓存
线程数	多线程	单线程
数据分片	自动处理相当于分片	提供分片算法、平衡各分片数据
数据扩容	动态增加数据卷平衡流量	需停机
数据同步	设置复制因子后可以透明的完成故障转移	手动故障转移和数据同步
载体	内存存储索引+SSD存储数据	内存

Aerospike作为一个大容量的NoSql解决方案，适合对容量要求比较大，QPS相对低一些的场景，主要用在广告行业，个性化推荐广告是建立在了和掌握消费者独特的偏好和习性的基础之上，对消费者的购买需求做出准确的预测或引导，在合适的位置、合适的时间，以合适的形式向消费者呈现与其需求高度吻合的广告，以此来促进用户的消费行为。



（ETL数据仓库技术）抽取（extract）、转换（transform）、加载（load）

- 用户行为日志收集系统收集日志之后推送到ETL做数据的清洗和转换
- 把ETL过后的数据发送到推荐引擎计算每个消费者的推荐结果，其中推荐逻辑包括规则和算法两部分
- 收集用户最近浏览、最长停留等特征，分析商品相似性、用户相似性、相似性等算法。
- 把推荐引擎的结果存入Aerospike集群中，并提供给广告投放引擎实时获取

分别通过HDFS和HBASE对日志进行离线和实时的分析，然后把用户画像的标签(tag：程序猿、宅男...)结果存入高性能的Nosql数据库Aerospike中，同时把数据备份到异地数据中心。前端广告投放请求通过决策引擎（投放引擎）向用户画像数据库中读取相应的用户画像数据，然后根据竞价算法出价进行竞价。竞价成功之后就可以展现广告了。而在竞价成功之后，具体给用户展现什么样的广告，就是有上面说的个性化推荐广告来完成的。

	Aerospike	Mysql
库名	Namespace	Database
表名	Set	Table
记录	Bin	Column
字段	Record	Row
索引	key 、 pk 、kv	pk

图谱Neo4j

> Neo4j是一个开源基于java开发的图形noSql数据库，它将结构化数据存储在图中而不是表中。它是一个嵌入式的、基于磁盘的、具备完全的事务特性的Java持久化引擎。程序数据是在一个面向对象的、灵活的网络结构下，而不是严格的表中，但具备完全的事务特性、企业级的数据库的所有好处。

一种基于图的数据结构，由节点(Node)和边(Edge)组成。其中节点即实体，由一个全局唯一的ID标示，边就是关系用于连接两个节点。通俗地讲，知识图谱就是把所有不同种类的信息，连接在一起而得到的一个关系网络。知识图谱提供了从“关系”的角度去分析问题的能力。

互联网、大数据的背景下，谷歌、百度、搜狗等搜索引擎纷纷基于该背景，创建自己的知识图Knowledge Graph（谷歌）、知心（百度）和知立方（搜狗），主要用于改进搜索质量。

自己项目主要用作好友推荐，图数据库(Graph database)指的是以图数据结构的形式来存储和查询数据的数据库。关系图谱中，关系的组织形式采用的就是图结构，所以非常适合用图库进行存储。

例如在下面这个例子中，我们希望在一个社交网络里找到最大深度为5的朋友的朋友。假设随机选择两个人，是否存在一条路径，使得关联他们的关系长度最多为5？对于一个包含100万人，每人约有50个朋友的社交网络，我们就以典型的开源图数据库Neo4j参与测试，结果明显表明，图数据库是用于关联数据的最佳选择，如表1所示。

深度	关系型数据库的执行时间（s）	Neo4j的执行时间（s）	返回的记录条数
2	0.016	0.01	～2500
3	30.267	0.168	～110 000
4	1543.505	1.359	～600 000
5	未完成	2.132	～800 000

表1 图数据库与关系型数据库执行时间对比

在深度为2时（即朋友的朋友），假设在一个在线系统中使用，无论关系型数据库还是图数据库，在执行时间方面都表现得足够好。虽然Neo4j的查询时间为关系数据库的2/3，但终端用户很难注意到两者间毫秒级的时间差异。当深度为3时（即朋友的朋友的朋友），很明显关系型数据库无法在合理的时间内实现查询：一个在线系统无法接受30s的查询时间。相比之下，Neo4j的响应时间则保持相对平坦：执行查询仅需要不到1s，这对在线系统来说足够快了。

- 优势总结:
- 性能上，使用cql查询，对长程关系的查询速度快
- 擅于发现隐藏的关系，例如通过判断图上两点之间有没有走的通的路径，就可以发现事物间的关联

S.No.	功能	描述
1	STARTNODE	它用于知道关系的开始节点。
2	ENDNODE	它用于知道关系的结束节点。
3	ID	它用于知道关系的ID。
4	TYPE	它用于知道字符串表示中的一个关系的TYPE。

```
match p = (:Person {name:"林婉儿"})-[r:Couple]-(:Person)
RETURN STARTNODE(r)
```

```
// 查询三层级关系节点如下：with可以将前面查询结果作为后面查询条件
match (na:Person)-[re]-(nb:Person) where na.name="林婉儿" WITH na,re,nb match (nb:Person)- [re2:-Friends]->(nc:Person) return na,re,nb,re2,nc
// 直接拼接关系节点查询
match data=(na:Person{name:"范闲"})-[re]->(nb:Person)-[re2]->(nc:Person) return data
// 使用深度运算符
显然使用以上方式比较繁琐,可变数量的关系->节点可以使用-[:TYPE*minHops..maxHops]-。
match data=(na:Person{name:"范闲"})-[*1..2]-(nb:Person) return data
```

文档MongoDB

> MongoDB 是一个基于分布式文件存储的数据库，是非关系数据库中功能最丰富、最像关系数据库的。在高负载的情况下，通过添加更多的节点，可以保证服务器性能。由 C++ 编写，可以为 WEB 应用提供可扩展、高性能、易部署的数据存储解决方案。

Mysql	MongoDB
database(数据库)	database（数据库）
table（表）	collection（ 集合）
row（ 行）	document（ BSON 文档）
column（列）	field（字段）
index（唯一索引、主键索引）	index（支持地理位置索引、全文索引 、哈希索引）
join（主外键关联）	embedded Document (嵌套文档)
primary key(指定1至N个列做主键)	primary key (指定_id field做为主键)

什么是BSON

> {key:value,key2:value2}和Json类似，是一种二进制形式的存储格式，支持内嵌的文档对象和数组对象，但是BSON有JSON没有的一些数据类型，比如 value包括字符串,double,Array,DateBSON可以做为网络数据交换的一种存储形式,它的优点是灵活性高，但它的缺点是空间利用率不是很理想。

BSON有三个特点：轻量性、可遍历性、高效性

```
/* 查询 find() 方法可以传入多个键(key)，每个键(key)以逗号隔开*/
db.collection.find({key1:value1, key2:value2}).pretty()
/* 更新 $set ： 设置字段值 $unset :删除指定字段 $inc：对修改的值进行自增*/
db.collection.update({where},{$set:{字段名:值}},{multi:true})
/* 删除 justOne :如果设为true，只删除一个文档，默认false，删除所有匹配条件的文档*/
db.collection.remove({where}, {justOne: <boolean>, writeConcern: <回执> } )
```

优点：

- 文档结构的存储方式，能够更便捷的获取数据。
  对于一个层级式的数据结构来说，使用扁平式的，表状的结构来查询保存数据非常的困难。
- 内置GridFS，支持大容量的存储。
  GridFS是一个出色的分布式文件系统，支持海量的数据存储，满足对大数据集的快速范围查询。
- 性能优越
  千万级别的文档对象，近10G的数据，对有索引的ID的查询 不会比mysql慢，而对非索引字段的查询，则是全面胜出。 mysql实际无法胜任大数据量下任意字段的查询，而mongodb的查询性能实在牛逼。写入性能同样很令人满意，同样写入百万级别的数据，mongodb基本10分钟以下可以解决。

缺点：

- 不支持事务
- 磁盘占用空间大

MySQL 8.0 版本

1. 性能：MySQL 8.0 的速度要比 MySQL 5.7 快 2 倍。
2. NoSQL：MySQL 从 5.7 版本开始提供 NoSQL 存储功能，在 8.0 版本中nosql得到了更大的改进。
3. 窗口函数：实现若干新的查询方式。窗口函数与 SUM()、COUNT() 这种集合函数类似，但它不会将多行查询结果合并为一行，而是将结果放回多行当中，即窗口函数不需要 GROUP BY。
4. 隐藏索引：在 MySQL 8.0 中，索引可以被“隐藏”和“显示”。当对索引进行隐藏时，它不会被查询优化器所使用。我们可以使用这个特性用于性能调试，例如我们先隐藏一个索引，然后观察其对数据库的影响。如果数据库性能有所下降，说明这个索引是有用的，然后将其“恢复显示”即可；如果数据库性能看不出变化，说明这个索引是多余的，可以考虑删掉。

云存储

	OSS	自建
可靠性	可用性不低于99.995% 数据设计持久性不低于99.9999999999%（12个9）	受限于硬件可靠性，易出问题，一旦出现磁盘坏道，容易出现不可逆转的数据丢失。人工数据恢复困难、耗时、耗力。
安全	服务端加密、客户端加密、防盗链、IP黑白名单等。多用户资源隔离机制，支持异地容灾机制。	需要另外购买清洗和黑洞设备。需要单独实现安全机制。
成本	多线BGP骨干网络，无带宽限制，上行流量免费。无需运维人员与托管费用，0成本运维。	单线或双线接入速度慢，有带宽限制，峰值时期需人工扩容。需专人运维，成本高。

使用步骤

1. 开通服务
2. 创建存储空间
3. 上传文件、下载文件、删除文件
4. 域名绑定、日志记录
5. 根据开放接口进行鉴权访问

功能

图片编辑（裁剪、模糊、水印）

视频截图

音频转码、视频修复

CDN加速

对象存储OSS与阿里云CDN服务结合，可优化静态热点文件下载加速的场景（即同一地区大量用户同时下载同一个静态文件的场景）。可以将OSS的存储空间（Bucket）作为源站，利用阿里云CDN将源内容发布到边缘节点。当大量终端用户重复访问同一文件时，可以直接从边缘节点获取已缓存的数据，提高访问的响应速度

FastDFS

> 开源的轻量级分布式文件系统。它对文件进行管理，功能包括：文件存储、文件同步、文件访问（文件上传、文件下载）等，解决了大容量存储和负载均衡的问题。使用FastDFS很容易搭建一套高性能的文件服务器集群提供文件上传、下载等服务。如相册网站、视频网站等

扩展能力: 支持水平扩展，可以动态扩容；
高可用性: 一是整个文件系统的可用性，二是数据的完整和一致性；
弹性存储: 可以根据业务需要灵活地增删存储池中的资源，而不需要中断系统运行。

指标	适合类型	文件分布	复杂度	FUSE	POSIX	备份机制	通讯协议接口	社区支持	开发语言
FastDFS	4KB~500MB	小文件合并存储不分片处理	简单	不支持	不支持	组内冗余备份	Api http	国内用户群	C
TFS	所有文件	小文件合并，以block组织分片	复杂	不支持	不支持	Block存储多份,主辅灾备	API http	少	C++
MFS	大于64K	分片存储	复杂	支持	支持	多点备份动态冗余	使用fuse	较多	Perl
HDFS	大文件	大文件分片分块存储	简单	支持	支持	多副本	原生api	较多	Java
Ceph	对象文件块	OSD一主多从	复杂	支持	支持	多副本	原生api	较少	C++
MogileFS	海量小图片	不分片存储	复杂	可以支持	不支持	动态冗余	http原生api	文档少	Perl

特性

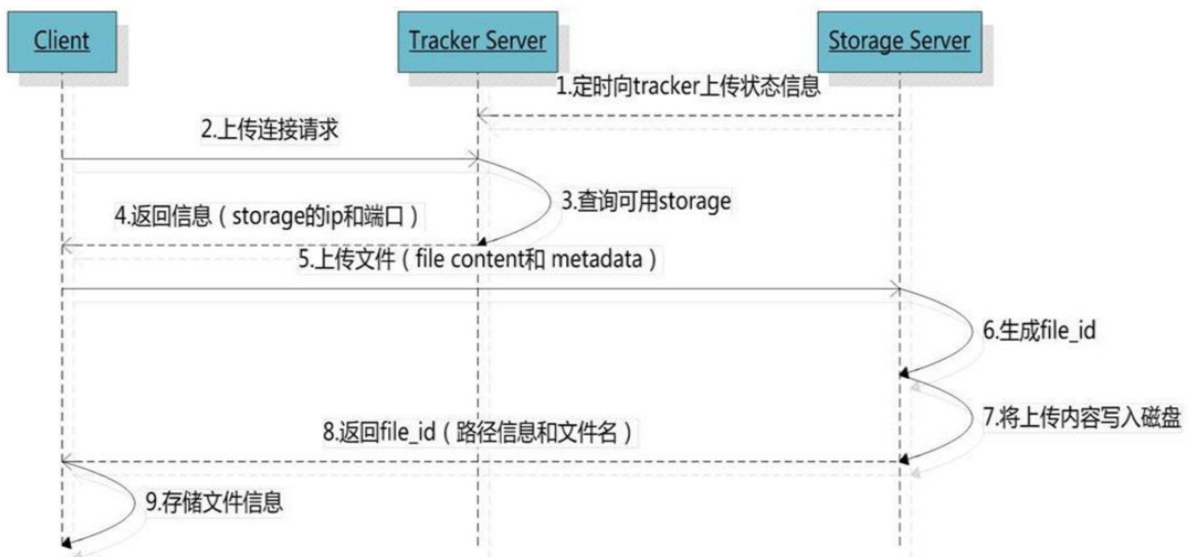
- 和流行的web server无缝衔接，FastDFS已提供apache和nginx扩展模块
- 文件ID由FastDFS生成，作为文件访问凭证，FastDFS不需要传统的name server
- 分组存储，灵活简洁、对等结构，不存在单点
- 文件不分块存储，上传的文件和OS文件系统中的文件一一对应
- 中、小文件均可以很好支持，支持海量小文件存储
- 支持相同内容的文件只保存一份，节约磁盘空间
- 支持多块磁盘，支持单盘数据恢复
- 支持在线扩容 支持主从文件
- 下载文件支持多线程方式，支持断点续传

组成

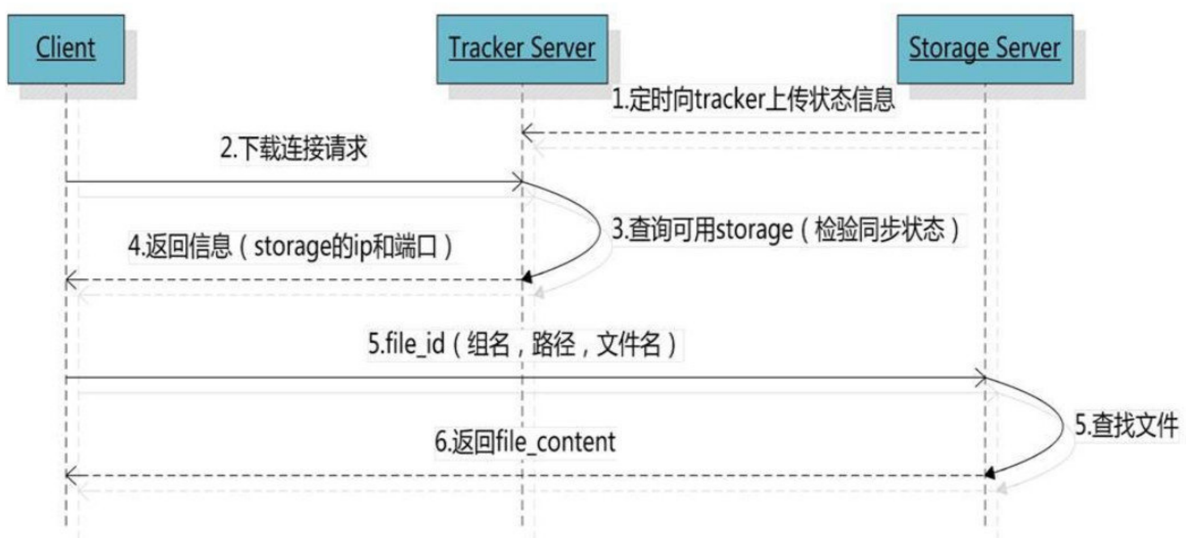
- 客户端（client）
  通过专有接口，使用TCP/IP协议与跟踪器服务器或存储节点进行数据交互。
- 跟踪器（tracker）
  Trackerserver作用是负载均衡和调度，通过Tracker server在文件上传时可以根据策略找到文件上传的地址。Tracker在访问上起负载均衡的作用。

- 存储节点（storage）
  Storageserver作用是文件存储，客户端上传的文件最终存储在Storage服务器上，Storage server没有实现自己的文件系统而是利用操作系统的文件系统来管理文件。存储节点中的服务器均可以随时增加或下线而不会影响线上服务。

上传



下载



断点续传

续传涉及到的文件大小MD5不会改变。续传流程与文件上传类似，先定位到源storage，完成完整或部分上传，再通过binlog进行同group内server文件同步。

配置优化

配置文件：tracker.conf 和 storage.conf

```
// FastDFS采用内存池的做法。
// v5.04对预分配采用增量方式，tracker一次预分配 1024个，storage一次预分配256个。
max_connections = 10240
// 根据实际需要将 max_connections 设置为一个较大的数值，比如 10240 甚至更大。
// 同时需要将一个进程允许打开的最大文件数调大
vi /etc/security/limits.conf 重启系统生效
* soft nofile 65535
* hard nofile 65535
```

```
work_threads = 4
// 说明：为了避免CPU上下文切换的开销，以及不必要的资源消耗，不建议将本参数设置得过大。
// 公式为： work_threads + (reader_threads + writer_threads) = CPU数
```

```
// 对于单盘挂载方式，磁盘读写线程分 别设置为 1即可
// 如果磁盘做了RAID，那么需要酌情加大读写线程数，这样才能最大程度地发挥磁盘性能
disk_rw_separated：磁盘读写是否分离
disk_reader_threads：单个磁盘读线程数
disk_writer_threads：单个磁盘写线程数
```

避免重复

如何避免文件重复上传 解决方案 上传成功后计算文件对应的MD5然后存入MySQL,添加文件时把文件MD5和之前存入MYSQL中的存储的信息对比 。DigestUtils.md5DigestAsHex(bytes)。

事务

1. 事务4大特性

事务4大特性：原子性、一致性、隔离性、持久性

原子性： 事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么全不执行

一致性： 执行事务前后，数据保持一致，多个事务对同一个数据读取的结果是相同的；
隔离性： 并发访问数据库时，一个用户的事务不被其他事务所干扰，各并发事务之间数据库是独立的；
持久性： 一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。
实现保证： MySQL的存储引擎InnoDB使用重做日志保证一致性与持久性，回滚日志保证原子性，使用各种锁来保证隔离性。

2. 事务隔离级别

读未提交： 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读。
读已提交： 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生。
可重复读： 同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，会有幻读。
串行化： 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰。

隔离级别	并发问题
读未提交	可能会导致脏读、幻读或不可重复读
读已提交	可能会导致幻读或不可重复读
可重复读	可能会导致幻读
可串行化	不会产生干扰

3. 默认隔离级别-RR

默认隔离级别： 可重复读；
同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改；

可重复读是有可能出现幻读的，如果要保证绝对的安全只能把隔离级别设置成SERIALIZABLE；这样所有事务都只能顺序执行，自然不会因为并发有什么影响了，但是性能会下降许多。

第二种方式，使用MVCC解决快照读幻读问题（如简单select），读取的不是最新的数据。维护一个字段作为version，这样可以控制到每次只能有一个人更新一个版本。

```
select id from table_xx where id = ? and version = V
update id from table_xx where id = ? and version = V+1
```

第三种方式，如果需要读最新的数据，可以通过GapLock+Next-KeyLock可以解决当前读幻读问题，

```
select id from table_xx where id > 100 for update;
select id from table_xx where id > 100 lock in share mode;
```

4. RR和RC使用场景

事务隔离级别RC (read commit) 和RR (repeatable read) 两种事务隔离级别基于多版本并发控制MVCC (multi-version concurrency control) 来实现。

	RC	RR
实现	多条查询语句会创建多个不同的ReadView	仅需要一个版本的ReadView
粒度	语句级读一致性	事务级读一致性
准确性	每次语句执行时间点的数据	第一条语句执行时间点的数据

5. 行锁，表锁，意向锁

InnoDB支持行级锁(row-level locking)和表级锁,默认为行级锁

InnoDB按照不同的分类的锁：

共享/排它锁(Shared and Exclusive Locks)：行级别锁，

意向锁(Intention Locks)，表级别锁

间隙锁(Gap Locks)，锁定一个区间

记录锁(Record Locks)，锁定一个行记录

表级锁：（串行化）

Mysql中锁定 粒度最大的一种锁，对当前操作的整张表加锁，实现简单 ，资源消耗也比较少，加锁快，不会出现死锁 。其锁定粒度最大，触发锁冲突的概率最高，并发度最低，MyISAM和 InnoDB引擎都支持表级锁。

行级锁：（RR、RC）

Mysql中锁定 粒度最小 的一种锁，只针对当前操作的行进行加锁。 行级锁能大大减少数据库操作的冲突。其加锁粒度最小，并发度高，但加锁的开销也最大，加锁慢，会出现死锁。InnoDB支持的行级锁，包括如下几种：

记录锁（Record Lock）：对索引项加锁，锁定符合条件的行。其他事务不能修改和删除加锁项；

间隙锁（Gap Lock）：对索引项之间的“间隙”加锁，锁定记录的范围，不包含索引项本身，其他事务不能在锁范围内插入数据。

Next-key Lock： 锁定索引项本身和索引范围。即Record Lock和Gap Lock的结合。可解决幻读问题。

InnoDB 支持多粒度锁（multiple granularity locking），它允许行级锁与表级锁共存，而意向锁就是其中的一种表锁。

共享锁（ shared lock, S ）锁允许持有锁读取行的事务。加锁时将自己和子节点全加S锁，父节点直到表头全加IS锁

排他锁（ exclusive lock， X ）锁允许持有锁修改行的事务。 加锁时将自己和子节点全加X锁，父节点直到表头全加IX锁

意向共享锁（intention shared lock, IS）：事务有意向对表中的某些行加共享锁（S锁）

意向排他锁（intention exclusive lock, IX）：事务有意向对表中的某些行加排他锁（X锁）

互斥性	共享锁（S）	排它锁（X）	意向共享锁IS	意向排他锁IX
共享锁（S）	✅	❌	✅	❌
排它锁（X）	❌	❌	❌	❌
意向共享锁IS	✅	❌	✅	✅
意向排他锁IX	❌	❌	✅	✅

6. MVCC多版本并发控制

MVCC是一种多版本并发控制机制，通过事务的可见性看到自己预期的数据，能降低其系统开销.（RC和RR级别工作）

InnoDB的MVCC,是通过在每行记录后面保存系统版本号(可以理解为事务的ID)，每开始一个新的事务，系统版本号就会自动递增，事务开始时刻的系统版本号会作为事务的ID。这样可以确保事务读取的行，要么是在事务开始前已经存在的，要么是事务自身插入或者修改过的，防止幻读的产生。

1. MVCC手段只适用于Msyql隔离级别中的读已提交（Read committed）和可重复读（Repeatable Read）.
2. Read uncimmitted由于存在脏读，即能读到未提交事务的数据行，所以不适用MVCC.
3. 简单的select快照度不会加锁，删改及select for update等需要当前读的场景会加锁

原因是MVCC的创建版本和删除版本只要在事务提交后才会产生。客观上，mysql使用的是乐观锁的一整实现方式，就是每行都有版本号，保存时根据版本号决定是否成功。Innodb的MVCC使用到的快照存储在Undo日志中，该日志通过回滚指针把一个数据行所有快照连接起来。

版本链

在InnoDB引擎表中，它的聚簇索引记录中有两个必要的隐藏列：

trx_id

这个id用来存储的每次对某条聚簇索引记录进行修改的时候的事务id。

roll_pointer

每次对哪条聚簇索引记录有修改的时候，都会把老版本写入undo日志中。这个roll_pointer就是存了一个指针，它指向这条聚簇索引记录的上一个版本的位置，通过它来获得上一个版本的记录信息。(注意插入操作的undo日志没有这个属性，因为它没有老版本)

每次修改都会在版本链中记录。SELECT可以去版本链中拿记录，这就实现了读-写，写-读的并发执行，提升了系统的性能。

索引

1. Innodb和Myisam引擎

Myisam：支持表锁，适合读密集的场景，不支持外键，不支持事务，索引与数据在不同的文件

Innodb：支持行、表锁，默认为行锁，适合并发场景，支持外键，支持事务，索引与数据同一文件

2. 哈希索引

哈希索引用索引列的值计算该值的hashCode，然后在hashCode相应的位置存执该值所在行数据的物理位置，因为使用散列算法，因此访问速度非常快，但是一个值只能对应一个hashCode，而且是散列的分布方式，因此哈希索引不支持范围查找和排序的功能

3. B+树索引

优点：

B+树的磁盘读写代价低，更少的查询次数，查询效率更加稳定，有利于对数据库的扫描

B+树是B树的升级版，B+树只有叶节点存放数据，其余节点用来索引。索引节点可以全部加入内存，增加查询效率，叶子节点可以做双向链表，从而提高范围查找的效率，增加的索引的范围

在大规模数据存储的时候，红黑树往往出现由于树的深度过大而造成磁盘IO读写过于频繁，进而导致效率低下的情况。所以，只要我们通过某种较好的树结构减少树的结构尽量减少树的高度，B树与B+树可以有多个子女，从几十到上千，可以降低树的高度。

磁盘预读原理：将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入。为了达到这个目的，在实际实现B-Tree还需要使用如下技巧：每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个node只需一次I/O。

4. 创建索引

```
CREATE [UNIQUE | FULLTEXT] INDEX 索引名 ON 表名(字段名) [USING 索引方法];

说明:
UNIQUE:可选。表示索引为唯一性索引。
FULLTEXT:可选。表示索引为全文索引。
INDEX和KEY:用于指定字段为索引，两者选择其中之一就可以了，作用是一样的。
索引名:可选。给创建的索引取一个新名称。
字段名1:指定索引对应的字段的名称，该字段必须是前面定义好的字段。
注：索引方法默认使用B+TREE。
```

5. 聚簇索引和非聚簇索引

聚簇索引：将数据存储与索引放到了一块，索引结构的叶子节点保存了行数据（主键索引）
非聚簇索引：将数据与索引分开存储，索引结构的叶子节点指向了数据对应的位置（辅助索引）
聚簇索引的叶子节点就是数据节点，而非聚簇索引的叶子节点仍然是索引节点，只不过有指向对应数据块的指针。

6. 最左前缀问题

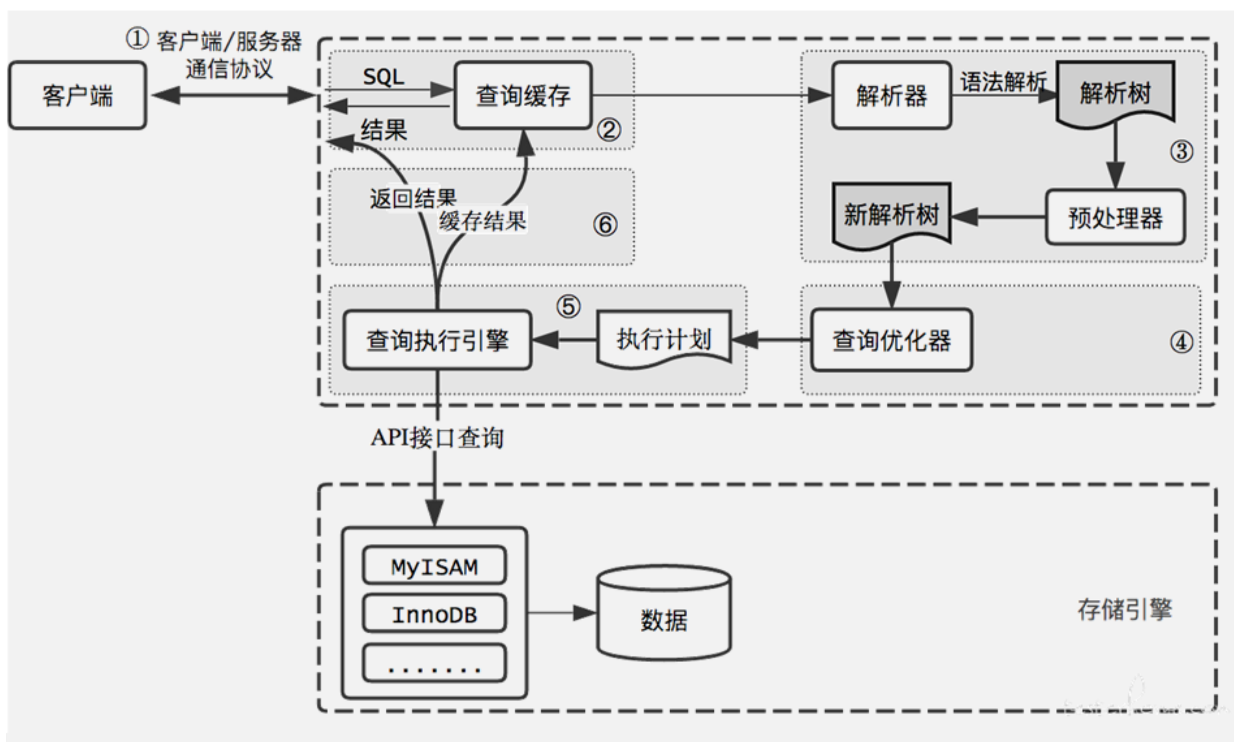
最左前缀原则主要使用在联合索引中，联合索引的B+Tree是按照第一个关键字进行索引排列的。
联合索引的底层是一颗B+树，只不过联合索引的B+树节点中存储的是键值。由于构建一棵B+树只能根据一个值来确定索引关系，所以数据库依赖联合索引最左的字段来构建。
采用>、<等进行匹配都会导致后面的列无法走索引，因为通过以上方式匹配到的数据是不可知的。

SQL查询

1. SQL语句的执行过程

查询语句：

```
select * from student  A where A.age='18' and A.name='张三';
```



结合上面的说明，我们分析下这个语句的执行流程：

①通过客户端/服务器通信协议与 MySQL 建立连接。并查询是否有权限

②Mysql8.0之前开看是否开启缓存，开启了 Query Cache 且命中完全相同的 SQL 语句，则将查询结果直接返回给客户端；

③由解析器进行语法语义解析，并生成解析树。如查询是select、表名tb_student、条件是id='1'

④查询优化器生成执行计划。根据索引看看是否可以优化

⑤查询执行引擎执行 SQL 语句，根据存储引擎类型，得到查询结果。若开启了 Query Cache，则缓存，否则直接返回。

2. 回表查询和覆盖索引

普通索引（唯一索引+联合索引+全文索引）需要扫描两遍索引树

（1）先通过普通索引定位到主键值id=5;

（2）在通过聚集索引定位到行记录;

这就是所谓的回表查询，先定位主键值，再定位行记录，它的性能较扫一遍索引树更低。

覆盖索引：主键索引==聚簇索引==覆盖索引

如果where条件的列和返回的数据在一个索引中，那么不需要回查表，那么就叫覆盖索引。

实现覆盖索引：常见的方法是，将被查询的字段，建立到联合索引里去。

3. Explain及优化

参考：https://www.jianshu.com/p/8fab76bbf448

```
mysql> explain select * from staff;
+----+-------------+-------+------+---------------+------+---------+------+------+-------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra |
+----+-------------+-------+------+---------------+------+---------+------+------+-------+
|  1 | SIMPLE      | staff | ALL  | NULL          | 索引 | NULL    | NULL |    2 | NULL  |
+----+-------------+-------+------+---------------+------+---------+------+------+-------+
1 row in set
```

索引优化：

①最左前缀索引：like只用于'string%'，语句中的=和in会动态调整顺序

②唯一索引：唯一键区分度在0.1以上

③无法使用索引：!= 、is null 、 or、>< 、（5.7以后根据数量自动判定）in 、not in

④联合索引：避免select * ，查询列使用覆盖索引

```
SELECT uid From user Where gid = 2 order by ctime asc limit 10
ALTER TABLE user add index idx_gid_ctime_uid(gid,ctime,uid) #创建联合覆盖索引，避免回表查询
```

语句优化：

①char固定长度查询效率高，varchar第一个字节记录数据长度

②应该针对Explain中Rows增加索引

③group/order by字段均会涉及索引

④Limit中分页查询会随着start值增大而变缓慢，通过子查询+表连接解决

```
select * from mytbl order by id limit 100000,10  改进后的SQL语句如下:
select * from mytbl where id >= ( select id from mytbl order by id limit 100000,1 ) limit 10
select * from mytbl inner ori join (select id from mytbl order by id limit 100000,10) as tmp on tmp.id=ori.id;
```

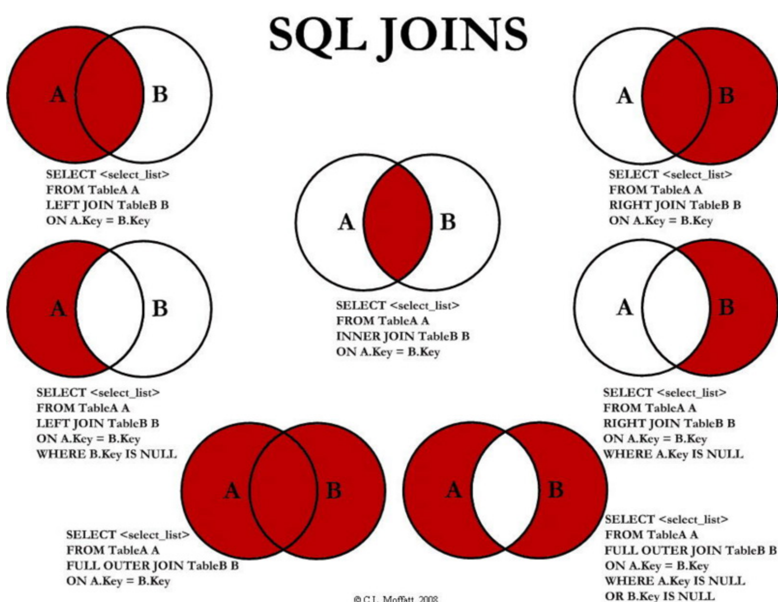
⑤count会进行全表扫描，如果估算可以使用explain
⑥delete删除表时会增加大量undo和redo日志， 确定删除可使用trancate

表结构优化：
①单库不超过200张表
②单表不超过500w数据
③单表不超过40列
④单表索引不超过5个

数据库范式：
①第一范式（1NF）列不可分割
②第二范式（2NF）属性完全依赖于主键 [消除部分子函数依赖]
③第三范式（3NF）属性不依赖于其它非主属性 [消除传递依赖]

配置优化：
配置连接数、禁用Swap、增加内存、升级SSD硬盘

4. JOIN查询



left join(左联接) 返回包括左表中的所有记录和右表中关联字段相等的记录
right join(右联接) 返回包括右表中的所有记录和左表中关联字段相等的记录
inner join(等值连接) 只返回两个表中关联字段相等的行

集群

1. 主从复制过程

MySQl主从复制：

- 原理：将主服务器的binlog日志复制到从服务器上执行一遍，达到主从数据的一致状态。
- 过程：从库开启一个I/O线程，向主库请求Binlog日志。主节点开启一个binlog dump线程，检查自己的二进制日志，并发送给从节点；从库将接收到的数据保存到中继日志（Relay log）中，另外开启一个SQL线程，把Relay中的操作在自身机器上执行一遍
- 优点：
  作为备用数据库，并且不影响业务
  可做读写分离，一个写库，一个或多个读库，在不同的服务器上，充分发挥服务器和数据库的性能，但要保证数据的一致性

binlog记录格式：statement、row、mixed

基于语句statement的复制、基于行row的复制、基于语句和行（mix）的复制。其中基于row的复制方式更能保证主从库数据的一致性，但日志量较大，在设置时考虑磁盘的空间问题

2. 数据一致性问题

"主从复制有延时"，这个延时期间读取从库，可能读到不一致的数据。

缓存记录写key法：

在cache里记录哪些记录发生过的写请求，来路由读主库还是读从库

异步复制：

在异步复制中，主库执行完操作后，写入binlog日志后，就返回客户端，这一动作就结束了，并不会验证从库有没有收到，完不完整，所以这样可能会造成数据的不一致。

半同步复制：

当主库每提交一个事务后，不会立即返回，而是等待其中一个从库接收到Binlog并成功写入Relay-log中才返回客户端，通过一份在主库的Binlog，另一份在其中一个从库的Relay-log，可以保证了数据的安全性和一致性。

全同步复制：

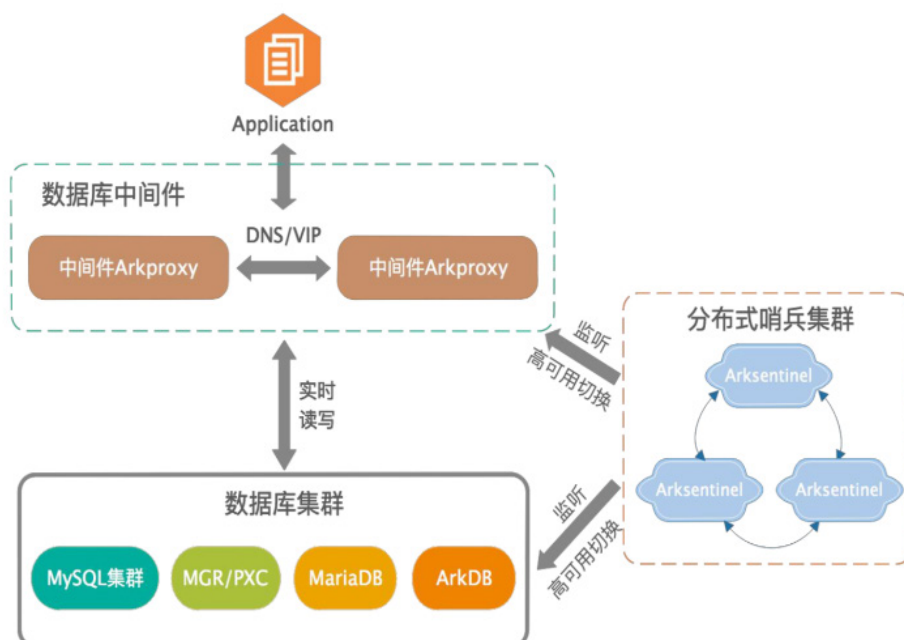
指当主库执行完一个事务，所有的从库都执行了该事务才返回给客户端。因为需要等待所有从库执行完该事务才

能返回，所以全同步复制的性能必然会收到严重的影响。

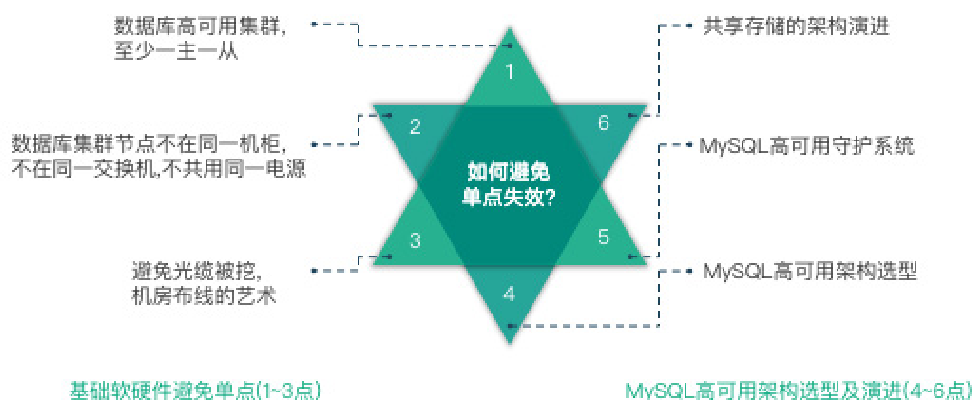
3. 集群架构

Keepalived + VIP + MySQL 主从/双主

当写节点 Master db1 出现故障时，由 MMM Monitor 或 Keepalived 触发切换脚本，将 VIP 漂移到可用的 Master db2 上。当出现网络抖动或网络分区时，MMM Monitor 会误判，严重时来回切换写 VIP 导致集群双写，当数据复制延迟时，应用程序会出现数据错乱或数据冲突的故障。有效避免单点失效的架构就是采用共享存储，单点故障切换可以通过分布式哨兵系统监控。



架构选型：MMM 集群 -> MHA集群 -> MHA+Arksentinel。



4. 故障转移和恢复

转移方式及恢复方法

1. 虚拟IP或DNS服务（Keepalived +VIP/DNS 和 MMM 架构）

问题：在虚拟 IP 运维过程中，刷新ARP过程中有时会出现一个 VIP 绑定在多台服务器同时提供连接的问题。这也是为什么要避免使用 Keepalived+VIP 和 MMM 架构的原因之一，因为它处理不了这类问题而导致集群多点写入。

2. 提升备库为主库（MHA、QMHA）

尝试将原 Master 设置 read_only 为 on，避免集群多点写入。借助 binlog server 保留 Master 的 Binlog；当出现数据延迟时，再提升 Slave 为新 Master 之前需要进行数据补齐，否则会丢失数据。

面试题

分库分表

如何进行分库分表

> 分表用户id进行分表，每个表控制在300万数据。
> 分库根据业务场景和地域分库，每个库并发不超过2000

Sharding-jdbc 这种 client 层方案的优点在于不用部署，运维成本低，不需要代理层的二次转发请求，性能很高，但是各个系统都需要耦合 Sharding-jdbc 的依赖，升级比较麻烦

Mycat 这种 proxy 层方案的缺点在于需要部署，自己运维一套中间件，运维成本高，但是好处在于对于各个项目是透明的，如果遇到升级之类的都是自己中间件那里搞就行了

水平拆分：一个表放到多个库，分担高并发，加快查询速度

- id保证业务在关联多张表时可以在同一库上操作
- range方便扩容和数据统计
- hash可以使得数据更加平均

垂直拆分：一个表拆成多个表，可以将一些冷数据拆分到冗余库中

> 不是写瓶颈优先进行分表

- 分库数据间的数据无法再通过数据库直接查询了。会产生深分页的问题
- 分库越多，出现问题的可能性越大，维护成本也变得更高。
- 分库后无法保障跨库间事务，只能借助其他中间件实现最终一致性。

分库首先需考虑满足业务最核心的场景：

1. 订单数据按用户分库，可以提升用户的全流程体验
2. 超级客户导致数据倾斜可以使用最细粒度唯一标识进行hash拆分
3. 按照最细粒度如订单号拆分以后，数据库就无法进行单库排重了

三个问题：

- 富查询：采用分库分表之后，如何满足跨越分库的查询？使用ES的宽表
  借助分库网关+分库业务虽然能够实现多维度查询的能力，但整体上性能不佳且对正常的写入请求有一定的影响。业界应对多维度实时查询的最常见方式便是借助 ElasticSearch
- 数据倾斜：数据分库基础上再进行分表
- 分布式事务：跨多库的修改及多个微服务间的写操作导致的分布式事务问题？
- 深分页问题：按游标查询，或者叫每次查询都带上上一次查询经过排序后的最大 ID

如何将老数据进行迁移

双写不中断迁移

- 线上系统里所有写库的地方，增删改操作，除了对老库增删改，都加上对新库的增删改
- 系统部署以后，还需要跑程序读老库数据写新库，写的时候需要判断updateTime
- 循环执行，直至两个库的数据完全一致，最后重新部署分库分表的代码就行了

系统性能的评估及扩容

和家亲目前有1亿用户：场景 10万写并发，100万读并发，60亿数据量

设计时考虑极限情况，32库*32表~64个表，一共1000 ~ 2000张表

- 支持3万的写并发，配合MQ实现每秒10万的写入速度
- 读写分离6万读并发，配合分布式缓存每秒100读并发
- 2000张表每张300万，可以最多写入60亿的数据
- 32张用户表，支撑亿级用户，后续最多也就扩容一次

动态扩容的步骤

1. 推荐是 32 库 * 32 表，对于我们公司来说，可能几年都够了。
2. 配置路由的规则，uid % 32 = 库，uid / 32 % 32 = 表

3. 扩容的时候，申请增加更多的数据库服务器，呈倍数扩容
4. 由 DBA 负责将原先数据库服务器的库，迁移到新的数据库服务器上去
5. 修改一下配置，重新发布系统，上线，原先的路由规则变都不用变
6. 直接可以基于 n 倍的数据库服务器的资源，继续进行线上系统的提供服务。

如何生成自增的id主键

- 使用redis可以
- 并发不高可以单独起一个服务，生成自增id
- 设置数据库step自增步长可以支撑水平伸缩
- UUID适合文件名、编号，但是不适合做主键
- snowflake雪花算法，综合了41时间（ms）、10机器、12序列号（ms内自增）

其中机器预留的10bit可以根据自己的业务场景配置

线上故障及优化

更新失败 | 主从同步延时

以前线上确实处理过因为主从同步延时问题而导致的线上的 bug，属于小型的生产事故。

是这个么场景。有个同学是这样写代码逻辑的。先插入一条数据，再把它查出来，然后更新这条数据。在生产环境高峰期，写并发达到了 2000/s，这个时候，主从复制延时大概是在小几十毫秒。线上会发现，每天总有那么一些数据，我们期望更新一些重要的数据状态，但在高峰期时候却没更新。用户跟客服反馈，而客服就会反馈给我们。

我们通过 MySQL 命令：

```
show slave status
```

查看 `Seconds_Behind_Master` ，可以看到从库复制主库的数据落后了几 ms。

一般来说，如果主从延迟较为严重，有以下解决方案：

- 分库，拆分为多个主库，每个主库的写并发就减少了几倍，主从延迟可以忽略不计。
- 重写代码，写代码的同学，要慎重，插入数据时立马查询可能查不到。
- 如果确实是存在必须先插入，立马要求就查询到，然后立马就要反过来执行一些操作，对这个查询设置直连主库或者延迟查询。主从复制延迟一般不会超过50ms

应用崩溃 | 分库分表优化

我们有一个线上通行记录的表，由于数据量过大，进行了分库分表，当时分库分表初期经常产生一些问题。典型的就是通行记录查询中使用了深分页，通过一些工具如MAT、Jstack追踪到是由于sharding-jdbc内部引用造成的。

通行记录数据被存放在两个库中。如果没有提供切分键，查询语句就会被分发到所有的数据库中，比如查询语句是 limit 10、offset 1000，最终结果只需要返回 10 条记录，但是数据库中间件要完成这种计算，则需要(1000+10)2=2020 条记录来完成这个计算过程。如果 offset 的值过大，使用的内存就会暴涨。虽然 sharding-jdbc 使用归并算法进行了一些优化，但在实际场景中，深分页仍然引起了内存和性能*问题。

这种在中间节点进行归并聚合的操作，在分布式框架中非常常见。比如在 ElasticSearch 中，就存在相似的数据获取逻辑，不加限制的深分页，同样会造成 ES 的内存问题。

业界解决方案：

方法一：全局视野法

（1）将order by time offset X limit Y，改写成order by time offset 0 limit X+Y

（2）服务层对得到的N*(X+Y)条数据进行内存排序，内存排序后再取偏移量X后的Y条记录

这种方法随着翻页的进行，性能越来越低。

方法二：业务折衷法-禁止跳页查询

（1）用正常的方法取得第一页数据，并得到第一页记录的time_max

（2）每次翻页，将order by time offset X limit Y，改写成order by time where time>$time_max limit Y

以保证每次只返回一页数据，性能为常量。

方法三：业务折衷法-允许模糊数据

（1）将order by time offset X limit Y，改写成order by time offset X/N limit Y/N

方法四：二次查询法

（1）将order by time offset X limit Y，改写成order by time offset X/N limit Y

（2）找到最小值time_min

（3）between二次查询，order by time between $timemin and $timei_max

（4）设置虚拟timemin，找到timemin在各个分库的offset，从而得到time_min在全局的offset

（5）得到了time_min在全局的offset，自然得到了全局的offset X limit Y

查询异常 | SQL 调优

分库分表前，有一段用用户名来查询某个用户的 SQL 语句：

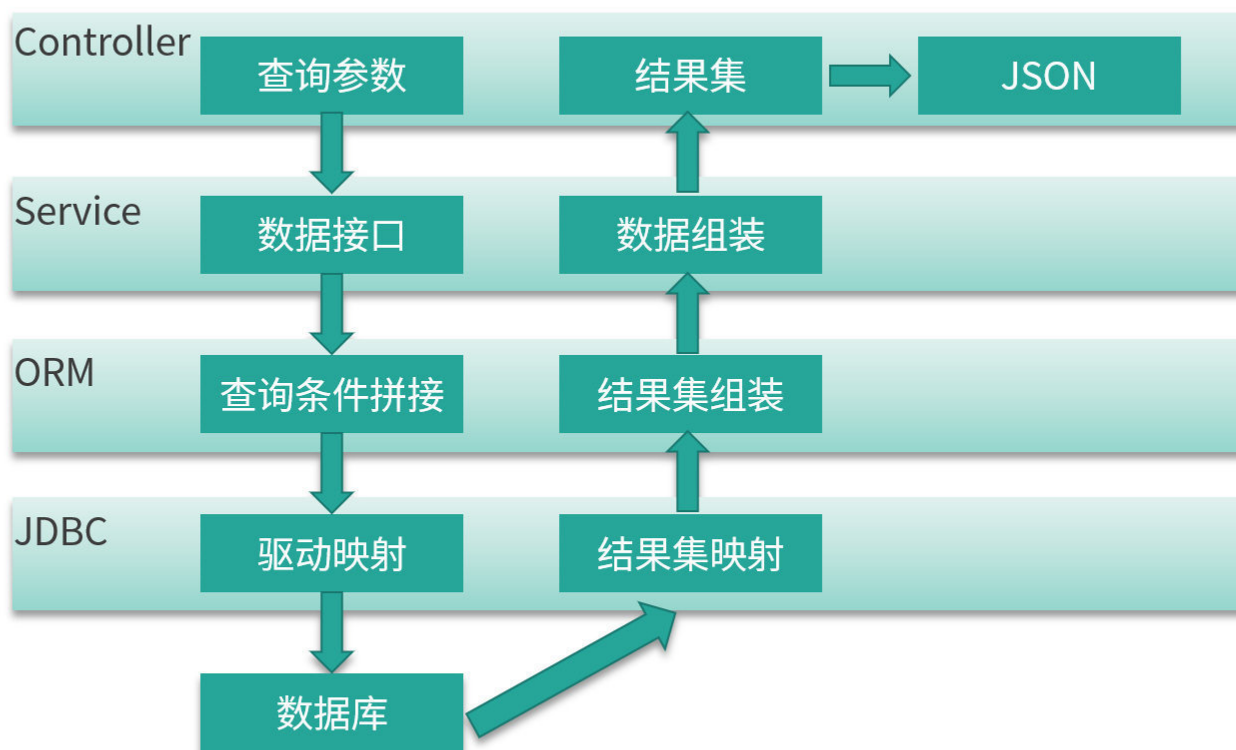
```
select * from user where name = "xxx" and community="other";
```

为了达到动态拼接的效果，这句 SQL 语句被一位同事进行了如下修改。他的本意是，当 name 或者 community 传入为空的时候，动态去掉这些查询条件。这种写法，在 MyBaits 的配置文件中，也非常常见。大多数情况下，这种写法是没有问题的，因为结果集合是可以控制的。但随着系统的运行，用户表的记录越来越多，当传入的 name 和 community 全部为空时，悲剧的事情发生了:

```
select * from user where 1=1
```

数据库中的所有记录，都会被查询出来，载入到 JVM 的内存中。由于数据库记录实在太多，直接把内存给撑爆了。由于这种原因引起的内存溢出，发生的频率非常高，比如导入Excel文件时。

通常的解决方式是强行加入分页功能，或者对一些必填的参数进行校验



Controller 层

现在很多项目都采用前后端分离架构，所以 Controller 层的方法，一般使用 @ResponseBody 注解，把查询的结果，解析成 JSON 数据返回。这在数据集非常大的情况下，会占用很多内存资源。假如结果集在解析成 JSON 之前，占用的内存是 10MB，那么在解析过程中，有可能会使用 20M 或者更多的内存

因此，保持结果集的精简，是非常有必要的，这也是 DTO（Data Transfer Object）存在的必要。互联网环境不怕小结果集的高并发请求，却非常恐惧大结果集的耗时请求，这是其中一方面的原因。

Service 层

Service 层用于处理具体的业务，更加贴合业务的功能需求。一个 Service，可能会被多个 Controller 层所使用，也可能会使用多个 dao 结构的查询结果进行计算、拼装。

```
int getUserSize() {
    List<User> users = dao.getAllUser();
    return null == users ? 0 : users.size();
}
```

代码review中发现了定时炸弹，这种在数据量达到一定程度后，才会暴露问题。

ORM 层

比如使用Mybatis时，有一个批量导入服务，在 MyBatis 执行批量插入的时候，竟然产生了内存溢出，按道理这种插入操作是不会引起额外内存占用的，最后通过源码追踪到了问题。

这是因为 MyBatis 循环处理 batch 的时候，操作对象是数组，而我们在接口定义的时候，使用的是 List；当传入一个非常大的 List 时，它需要调用 List 的 toArray 方法将列表转换成数组（浅拷贝）；在最后的拼装阶段，又使用了 StringBuilder 来拼接最终的 SQL，所以实际使用的内存要比 List 多很多。

事实证明，不论是插入操作还是查询动作，只要涉及的数据集非常大，就容易出现问题。由于项目中众多框架的引入，想要分析这些具体的内存占用，就变得非常困难。所以保持小批量操作和结果集的干净，是一个非常好的习惯。

Redis篇

WhyRedis

速度快，完全基于内存，使用C语言实现，网络层使用epoll解决高并发问题，单线程模型避免了不必要的上下文切换及竞争条件；

	GuavaCache	Tair	EVCache	Aerospike
类别	本地JVM缓存	分布式缓存	分布式缓存	分布式nosql数据库
应用	本地缓存	淘宝	Netflix、AWS	广告
性能	非常高	较高	很高	较高
持久化	无	有	有	有
集群	无	灵活配置	有	自动扩容

与传统数据库不同的是 Redis 的数据是存在内存中的，所以读写速度非常快，因此 redis 被广泛应用于缓存方向，每秒可以处理超过 10万次读写操作，是已知性能最快的Key-Value DB。另外，Redis 也经常用来做分布式锁。除此之外，Redis 支持事务 、持久化、LUA脚本、LRU驱动事件、多种集群方案。

1. 简单高效

1）完全基于内存，绝大部分请求是纯粹的内存操作。数据存在内存中，类似于 HashMap，查找和操作的时间复杂度都是O(1)；

2）数据结构简单，对数据操作也简单，Redis 中的数据结构是专门进行设计的；

3）采用单线程，避免了多线程不必要的上下文切换和竞争条件，不存在加锁释放锁操作，减少了因为锁竞争导致的性能消耗；（6.0以后多线程）

4）使用EPOLL多路 I/O 复用模型，非阻塞 IO；

5）使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis 直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求；

2. Memcache

redis	Memcached
内存高速数据库	高性能分布式内存缓存数据库
支持hash、list、set、zset、string结构	只支持key-value结构
将大部分数据放到内存	全部数据放到内存中
支持持久化、主从复制备份	不支持数据持久化及数据备份
数据丢失可通过AOF恢复	挂掉后，数据不可恢复
单线程（2~4万TPS）	多线程（20-40万TPS）

使用场景：

1. 如果有持久方面的需求或对数据类型和处理有要求的应该选择redis。
2. 如果简单的key/value 存储应该选择memcached。

3. Tair

Tair(Taobao Pair)是淘宝开发的分布式Key-Value存储引擎，既可以做缓存也可以做数据源（三种引擎切换）

- MDB（Memcache）属于内存型产品,支持kv和类hashMap结构,性能最优
- RDB（Redis）支持List.Set.Zset等复杂的数据结构,性能次之,可提供缓存和持久化存储两种模式
- LDB（levelDB）属于持久化产品,支持kv和类hashmap结构,性能较前两者稍低,但持久化可靠性最高

分布式缓存

大访问少量临时数据的存储（kb左右）

用于缓存，降低对后端数据库的访问压力

session场景

高速访问某些数据结构的应用和计算（rdb）

数据源存储

快速读取数据（fdb）

持续大数据量的存入读取（ldb），交易快照

高频度的更新读取（ldb），库存

痛点：redis集群中，想借用缓存资源必须得指明redis服务器地址去要。这就增加了程序的维护复杂度。因为redis服务器很可能是需要频繁变动的。所以人家淘宝就想啊，为什么不能像操作分布式数据库或者hadoop那样。增加一个中央节点，让他去代理所有事情。在tair中程序只要跟tair中心节点交互就OK了。同时tair里还有配置服务器概念。又免去了像操作hadoop那样，还得每台hadoop一套一模一样配置文件。改配置文件得整个集群都跟着改。

4. Guava

分布式缓存一致性更好一点，用于集群环境下多节点使用同一份缓存的情况；有网络IO，吞吐率与缓存的数据大小有较大关系；
本地缓存非常高效，本地缓存会占用堆内存，影响垃圾回收、影响系统性能。

本地缓存设计：
以 Java 为例，使用自带的 map 或者 guava 实现的是本地缓存，最主要的特点是轻量以及快速，生命周期随着 jvm 的销毁而结束，并且在多实例的情况，每个实例都需要各自保存一份缓存，缓存不具有一致性。

解决缓存过期：
1、将缓存过期时间调为永久
2、将缓存失效时间分散开，不要将缓存时间长度都设置成一样；比如我们可以在原有的失效时间基础上增加一个随机值，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

解决内存溢出：
第一步，修改JVM启动参数，直接增加内存。(-Xms，-Xmx参数一定不要忘记加。)
第二步，检查错误日志，查看“OutOfMemory”错误前是否有其它异常或错误。
第三步，对代码进行走查和分析，找出可能发生内存溢出的位置。

Google Guava Cache
自己设计本地缓存痛点：
- 不能按照一定的规则淘汰数据，如 LRU，LFU，FIFO 等。
- 清除数据时的回调通知
- 并发处理能力差，针对并发可以使用CurrentHashMap，但缓存的其他功能需要自行实现
- 缓存过期处理，缓存数据加载刷新等都需要手工实现

Guava Cache 的场景：
- 对性能有非常高的要求

- 不经常变化，占用内存不大
- 有访问整个集合的需求
- 数据允许不实时一致

Guava Cache 的优势：

- 缓存过期和淘汰机制
  在GuavaCache中可以设置Key的过期时间，包括访问过期和创建过期。GuavaCache在缓存容量达到指定大小时，采用LRU的方式，将不常使用的键值从Cache中删除
- 并发处理能力
  GuavaCache类似CurrentHashMap，是线程安全的。提供了设置并发级别的api，使得缓存支持并发的写入和读取，采用分离锁机制，分离锁能够减小锁力度，提升并发能力，分离锁是分拆锁定，把一个集合看分成若干partition, 每个partiton一把锁。更新锁定
- 防止缓存击穿
  一般情况下，在缓存中查询某个key，如果不存在，则查源数据，并回填缓存。（Cache Aside Pattern）在高并发下会出现，多次查源并重复回填缓存，可能会造成源的宕机（DB），性能下降 GuavaCache可以在CacheLoader的load方法中加以控制，对同一个key，只让一个请求去读源并回填缓存，其他请求阻塞等待。（相当于集成数据源，方便用户使用）
- 监控缓存加载/命中情况
  统计

问题：

OOM->设置过期时间、使用弱引用、配置过期策略

5. EVCache

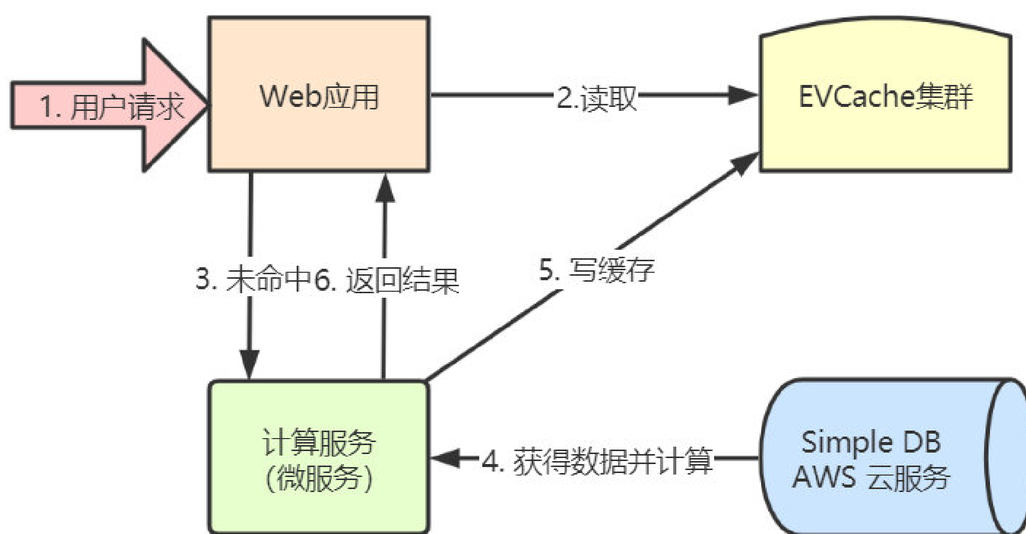
EVCache是一个Netflflix（网飞）公司开源、快速的分布式缓存，是基于Memcached的内存存储实现的，用以构建超大容量、高性能、低延时、跨区域的全球可用的缓存数据层。

E：Ephemeral：数据存储是短暂的，有自身的存活时间

V：Volatile：数据可以在任何时候消失

EVCache典型地适合对强一致性没有必须要求的场合

典型用例：Netflflix向用户推荐用户感兴趣的电影



EVCache集群在峰值每秒可以处理200kb的请求，

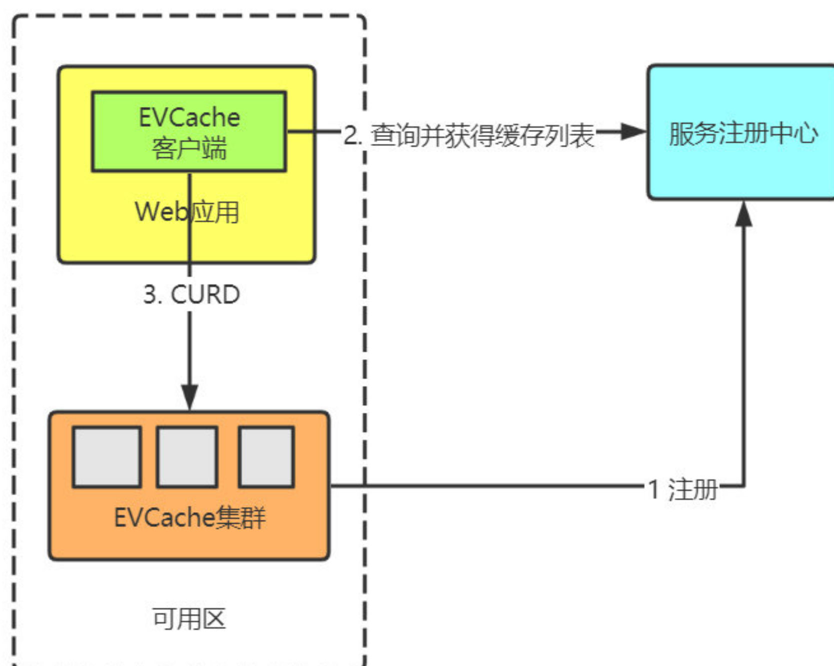
Netflflix生产系统中部署的EVCache经常要处理超过每秒3000万个请求，存储数十亿个对象，跨数千台memcached服务器。整个EVCache集群每天处理近2万亿个请求。

EVCache集群响应平均延时大约是1-5毫秒，最多不会超过20毫秒。

EVCache集群的缓存命中率在99%左右。

典型部署

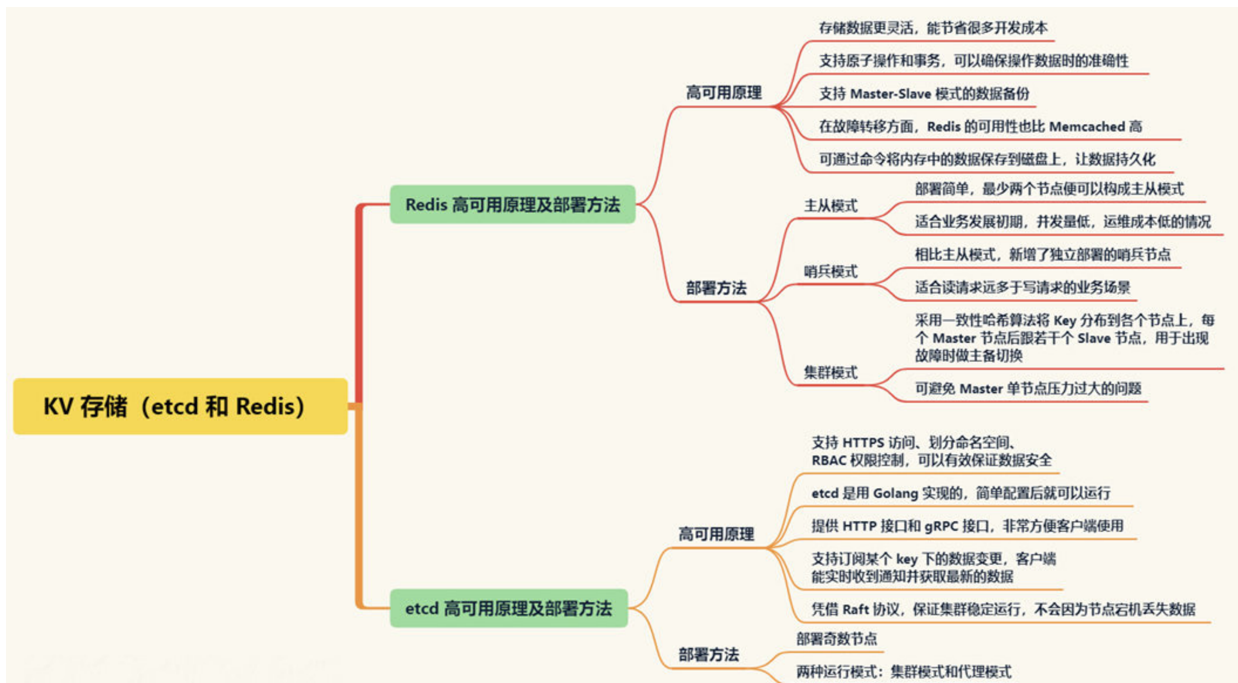
EVCache 是线性扩展的，可以在一分钟之内完成扩容，在几分钟之内完成负载均衡和缓存预热。



1. 集群启动时，EVCache向服务注册中心（Zookeeper、Eureka）注册各个实例
2. 在web应用启动时，查询命名服务中的EVCache服务器列表，并建立连接。
3. 客户端通过key使用一致性hash算法，将数据分片到集群上。

6. ETCD

和Zookeeper一样，CP模型追求数据一致性，越来越多的系统开始用它保存关键数据。比如，秒杀系统经常用它保存各节点信息，以便控制消费 MQ 的服务数量。还有些业务系统的配置数据，也会通过 etcd 实时同步给业务系统的各节点，比如，秒杀管理后台会使用 etcd 将秒杀活动的配置数据实时同步给秒杀 API 服务各节点。



Redis底层

1. redis数据类型

类型	底层	应用场景	编码类型
String	SDS数组	帖子、评论、热点数据、输入缓冲	RAW << EMBSTR << INT
List	QuickList	评论列表、商品列表、发布与订阅、慢查询、监视器	LINKEDLIST << ZIPLIST
Set	intSet	适合交集、并集、查集操作，例如朋友关系	HT << INSET
Zset	跳跃表	去重后排序，适合排名场景	SKIPLIST << ZIPLIST
Hash	哈希	结构化数据，比如存储对象	HT << ZIPLIST
Stream	紧凑列表	消息队列	

2. 相关API

> http://redisdoc.com

String	SET	SETNX	SETEX	GET	GETSET	INCR	DECR	MSET	MGET
Hash	HSET	HSETNX	HGET	HDEL	HLEN	HMSET	HMGET	HKEYS	HGETALL
LIST	LPUSH	LPOP	RPUSH	RPOP	LINDEX	LREM	LRANGE	LLEN	RPOPLPUSH
ZSET	ZADD	ZREM	ZSCORE	ZCARD	ZRANGE	ZRANK	ZREVRANK		ZREVRANGE
SET	SADD	SREM	SISMEMBER	SCARD	SINTER	SUNION	SDIFF	SPOP	SMEMBERS
事务	MULTI	EXEC	DISCARD	WATCH	UNWATCH				

3. redis底层结构

SDS数组结构，用于存储字符串和整型数据及输入缓冲。

```
struct sdshdr{
  int len;//记录buf数组中已使用字节的数量
  int free; //记录 buf 数组中未使用字节的数量
  char buf[];//字符数组，用于保存字符串
}
```

跳跃表：将有序链表中的部分节点分层，每一层都是一个有序链表。

1、可以快速查找到需要的节点 O(logn)，额外存储了一倍的空间

2、可以在O(1)的时间复杂度下，快速获得跳跃表的头节点、尾结点、长度和高度。

字典dict: 又称散列表(hash)，是用来存储键值对的一种数据结构。

Redis整个数据库是用字典来存储的(K-V结构) —Hash+数组+链表

Redis字典实现包括:字典(dict)、Hash表(dictht)、Hash表节点(dictEntry)。

字典达到存储上限(阈值 0.75)，需要rehash(扩容)

1、初次申请默认容量为4个dictEntry，非初次申请为当前hash表容量的一倍。

2、rehashidx=0表示要进行rehash操作。

3、新增加的数据在新的hash表h[1] 、修改、删除、查询在老hash表h[0]

4、将老的hash表h[0]的数据重新计算索引值后全部迁移到新的hash表h[1]中，这个过程称为 rehash。

渐进式rehash

```
由于当数据量巨大时rehash的过程是非常缓慢的，所以需要进行优化。 可根据服务器空闲程度批量rehash部分节点
```

压缩列表zipList

压缩列表(ziplist)是由一系列特殊编码的连续内存块组成的顺序型数据结构，节省内容

sorted-set和hash元素个数少且是小整数或短字符串(直接使用)

list用快速链表(quicklist)数据结构存储，而快速链表是双向列表与压缩列表的组合。(间接使用)

整数集合intSet

整数集合(intset)是一个有序的(整数升序)、存储整数的连续存储结构。

当Redis集合类型的元素都是整数并且都处在64位有符号整数范围内(2^64)，使用该结构体存储。

快速列表quickList

快速列表(quicklist)是Redis底层重要的数据结构。是Redis3.2列表的底层实现。

(在Redis3.2之前，Redis采 用双向链表(adlist)和压缩列表(ziplist)实现。)

Redis Stream的底层主要使用了listpack(紧凑列表)和Rax树(基数树)。
listpack表示一个字符串列表的序列化，listpack可用于存储字符串或整数。用于存储stream的消息内 容。
Rax树是一个有序字典树 (基数树 Radix Tree)，按照 key 的字典序排列，支持快速地定位、插入和删除操作。

4. Zset底层实现

跳表(skip List)是一种随机化的数据结构，基于并联的链表，实现简单，插入、删除、查找的复杂度均为O(logN)。简单说来跳表也是链表的一种，只不过它在链表的基础上增加了跳跃功能，正是这个跳跃的功能，使得在查找元素时，跳表能够提供O(logN)的时间复杂度
Zset数据量少的时候使用压缩链表ziplist实现，有序集合使用紧挨在一起的压缩列表节点来保存，第一个节点保存member，第二个保存score。ziplist内的集合元素按score从小到大排序，score较小的排在表头位置。 数据量大的时候使用跳跃列表skiplist和哈希表hash_map结合实现，查找删除插入的时间复杂度都是O(longN)
Redis使用跳表而不使用红黑树，是因为跳表的索引结构序列化和反序列化更加快速，方便持久化。

搜索
跳跃表按 score 从小到大保存所有集合元素，查找时间复杂度为平均 O(logN)，最坏 O(N) 。

插入
选用链表作为底层结构支持，为了高效地动态增删。因为跳表底层的单链表是有序的，为了维护这种有序性，在插入前需要遍历链表，找到该插入的位置，单链表遍历查找的时间复杂度是O(n)，同理可得，跳表的遍历也是需要遍历索引数，所以是O(logn)。

删除
如果该节点还在索引中，删除时不仅要删除单链表中的节点，还要删除索引中的节点；单链表在知道删除的节点是谁时，时间复杂度为O(1)，但针对单链表来说，删除时都需要拿到前驱节点O(logN)才可改变引用关系从而删除目标节点。

Redis可用性

1. redis持久化

持久化就是把内存中的数据持久化到本地磁盘，防止服务器宕机了内存数据丢失
Redis 提供两种持久化机制 RDB（默认） 和 AOF 机制，Redis4.0以后采用混合持久化，用 AOF 来保证数据不丢失，作为数据恢复的第一选择; 用 RDB 来做不同程度的冷备

RDB：是Redis DataBase缩写快照

RDB是Redis默认的持久化方式。按照一定的时间将内存的数据以快照的形式保存到硬盘中，对应产生的数据文件为dump.rdb。通过配置文件中的save参数来定义快照的周期。

优点：

1）只有一个文件 dump.rdb，方便持久化；

2）容灾性好，一个文件可以保存到安全的磁盘。

3）性能最大化，fork 子进程来进行持久化写操作，让主进程继续处理命令，只存在毫秒级不响应请求。

4）相对于数据集大时，比 AOF 的启动效率更高。

缺点：

数据安全性低，RDB 是间隔一段时间进行持久化，如果持久化之间 redis 发生故障，会发生数据丢失。

AOF：持久化

AOF持久化(即Append Only File持久化)，则是将Redis执行的每次写命令记录到单独的日志文件中，当重启Redis会重新将持久化的日志中文件恢复数据。

优点：

1）数据安全，aof 持久化可以配置 appendfsync 属性，有 always，每进行一次 命令操作就记录到 aof 文件中一次。

2）通过 append 模式写文件，即使中途服务器宕机，可以通过 redis-check-aof 工具解决数据一致性问题。

缺点：

1）AOF 文件比 RDB 文件大，且恢复速度慢。

2）数据集大的时候，比 rdb 启动效率低。

2. redis事务

事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

Redis事务的概念

Redis 事务的本质是通过MULTI、EXEC、WATCH等一组命令的集合。事务支持一次执行多个命令，一个事务中所有命令都会被序列化。在事务执行过程，会按照顺序串行化执行队列中的命令，其他客户端提交的命令请求不会插入到事务执行命令序列中。总结说：redis事务就是一次性、顺序性、排他性的执行一个队列中的一系列命令。

Redis的事务总是具有ACID中的一致性和隔离性，其他特性是不支持的。当服务器运行在AOF持久化模式下，并且appendfsync选项的值为always时，事务也具有耐久性。

Redis事务功能是通过MULTI、EXEC、DISCARD和WATCH 四个原语实现的

事务命令：

MULTI：用于开启一个事务，它总是返回OK。MULTI执行之后，客户端可以继续向服务器发送任意多条命令，这

些命令不会立即被执行，而是被放到一个队列中，当EXEC命令被调用时，所有队列中的命令才会被执行。

EXEC：执行所有事务块内的命令。返回事务块内所有命令的返回值，按命令执行的先后顺序排列。当操作被打断时，返回空值 nil 。

WATCH ：是一个乐观锁，可以为 Redis 事务提供 check-and-set （CAS）行为。可以监控一个或多个键，一旦其中有一个键被修改（或删除），之后的事务就不会执行，监控一直持续到EXEC命令。（秒杀场景）

DISCARD：调用该命令，客户端可以清空事务队列，并放弃执行事务，且客户端会从事务状态中退出。

UNWATCH：命令可以取消watch对所有key的监控。

3. redis失效策略

内存淘汰策略

1）全局的键空间选择性移除

noeviction：当内存不足以容纳新写入数据时，新写入操作会报错。（字典库常用）

allkeys-lru：在键空间中，移除最近最少使用的key。（缓存常用）

allkeys-random：在键空间中，随机移除某个key。

2）设置过期时间的键空间选择性移除

volatile-lru：在设置了过期时间的键空间中，移除最近最少使用的key。

volatile-random：在设置了过期时间的键空间中，随机移除某个key。

volatile-ttl：在设置了过期时间的键空间中，有更早过期时间的key优先移除。

缓存失效策略

定时清除：针对每个设置过期时间的key都创建指定定时器

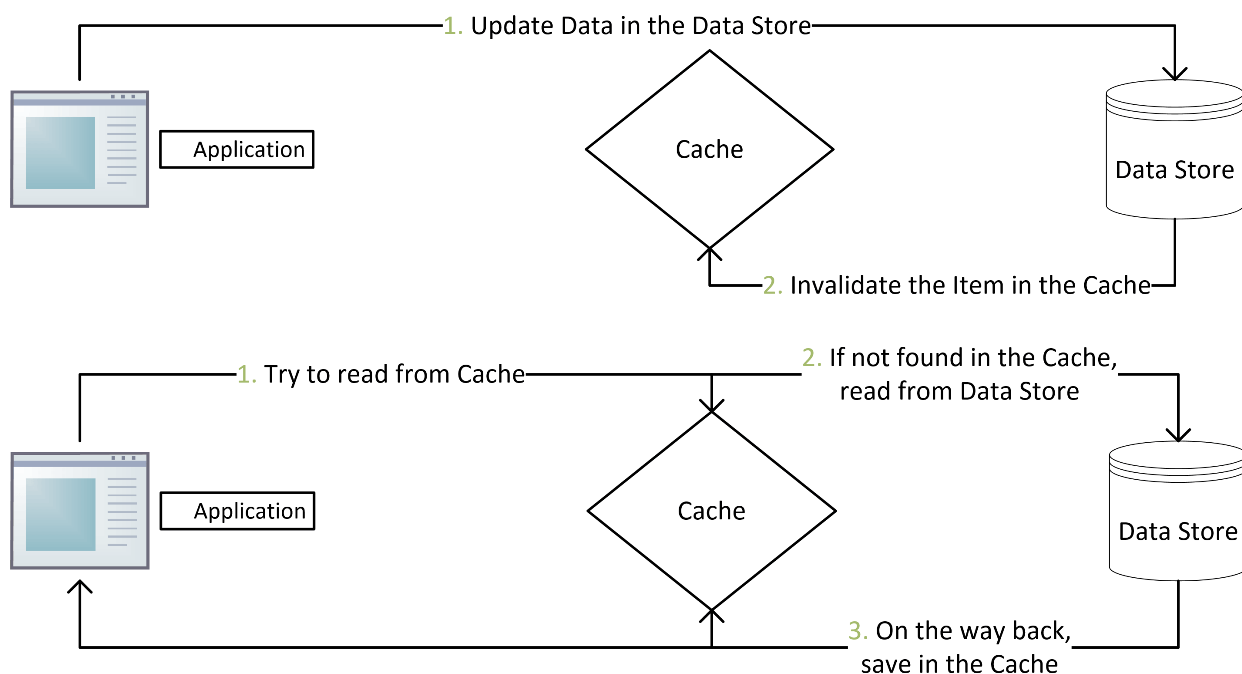
惰性清除：访问时判断，对内存不友好

定时扫描清除：定时100ms随机20个检查过期的字典，若存在25%以上则继续循环删除。

4. redis读写模式

CacheAside旁路缓存

写请求更新数据库后删除缓存数据。读请求不命中查询数据库，查询完成写入缓存



业务端处理所有数据访问细节，同时利用 Lazy 计算的思想，更新 DB 后，直接删除 cache 并通过 DB 更新，确保数据以 DB 结果为准，则可以大幅降低 cache 和 DB 中数据不一致的概率

如果没有专门的存储服务，同时是对数据一致性要求比较高的业务，或者是缓存数据更新比较复杂的业务，适合使用 Cache Aside 模式。如微博发展初期，不少业务采用这种模式

```
// 延迟双删，用以保证最终一致性,防止小概率旧数据读请求在第一次删除后更新数据库
public void write(String key,Object data){
    redis.delKey(key);
    db.updateData(data);
    Thread.sleep(1000);
    redis.delKey(key);
}
```

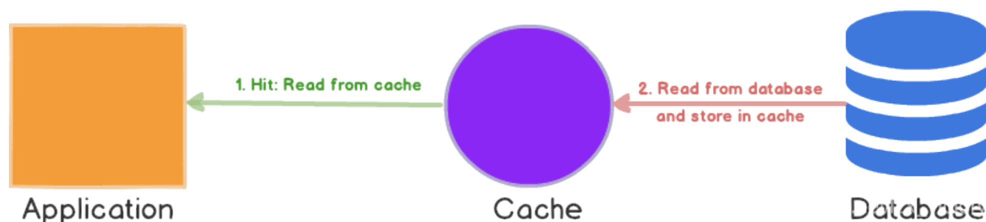
高并发下保证绝对的一致，先删缓存再更新数据，需要用到内存队列做异步串行化。非高并发场景，先更新数据再删除缓存，延迟双删策略基本满足了

- 先更新db后删除redis：删除redis失败则出现问题
- 先删redis后更新db：删除redis瞬间，旧数据被回填redis
- 先删redis后更新db休眠后删redis：同第二点，休眠后删除redis 可能宕机
- java内部jvm队列：不适用分布式场景且降低并发

Read/Write Though（读写穿透）

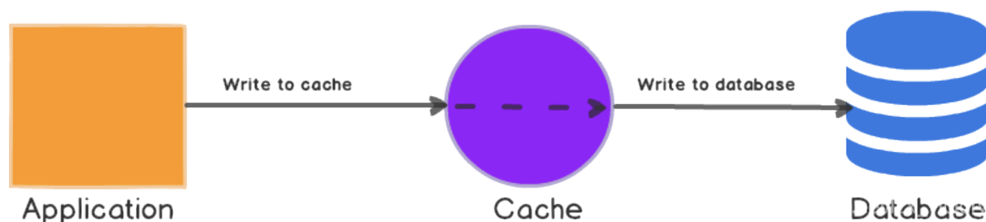
先查询缓存中数据是否存在,如果存在则直接返回,如果不存在,则由缓存组件负责从数据库中同步加载数据.

Read-Through



先查询要写入的数据在缓存中是否已经存在,如果已经存在,则更新缓存中的数据，
并且由缓存组件同步更新到数据库中。

Write-Through

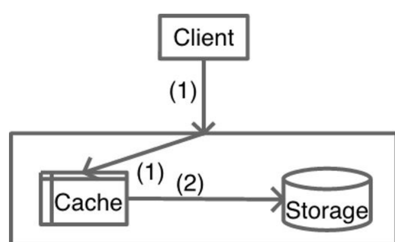


用户读操作较多.相较于Cache aside而言更适合缓存一致的场景。使用简单屏蔽了底层数据库的操作,只是操作缓存.

场景：

微博 Feed 的 Outbox Vector（即用户最新微博列表）就采用这种模式。一些粉丝较少且不活跃的用户发表微博后，Vector 服务会首先查询 Vector Cache，如果 cache 中没有该用户的 Outbox 记录，则不写该用户的 cache 数据，直接更新 DB 后就返回，只有 cache 中存在才会通过 CAS 指令进行更新。

Write Behind Caching（异步缓存写入）



Write Behind Caching

1. Write：只更新缓存，缓存服务异步更新DB
2. Read：miss后由缓存服务加载+写cache
3. 特点：写性能最高，定期异步刷新，存在数据丢失概率
4. 适合场景：写频率超高，需要合并

比如对一些计数业务，一条 Feed 被点赞 1万 次，如果更新 1万 次 DB 代价很大，而合并成一次请求直接加 1万，则是一个非常轻量的操作。但这种模型有个显著的缺点，即数据的一致性变差，甚至在一些极端场景下可能会丢失数据。

5. 多级缓存

浏览器本地内存缓存： 专题活动，一旦上线，在活动期间是不会随意变更的。
浏览器本地磁盘缓存： Logo缓存，大图片懒加载
服务端本地内存缓存： 由于没有持久化，重启时必定会被穿透
服务端网络内存缓存： Redis等，针对穿透的情况下可以继续分层，必须保证数据库不被压垮

为什么不是使用服务器本地磁盘做缓存？
当系统处理大量磁盘 IO 操作的时候，由于 CPU 和内存的速度远高于磁盘，可能导致 CPU 耗费太多时间等待磁盘返回处理的结果。对于这部分 CPU 在 IO 上的开销，我们称为 iowai

Redis七大经典问题

1. 缓存雪崩

指缓存同一时间大面积的失效，所以，后面的请求都会落到数据库上，造成数据库短时间内承受大量请求而崩掉。

解决方案：
- Redis 高可用，主从+哨兵，Redis cluster，避免全盘崩溃
- 本地 ehcache 缓存 + hystrix 限流&降级，避免 MySQL 被打死
- 缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。
- 逻辑上永不过期给每一个缓存数据增加相应的缓存标记，缓存标记失效则更新数据缓存
- 多级缓存，失效时通过二级更新一级，由第三方插件更新二级缓存。

2. 缓存穿透

https://blog.csdn.net/lin777lin/article/details/105666839
缓存穿透是指缓存和数据库中都没有的数据，导致所有的请求都落到数据库上，造成数据库短时间内承受大量请求而崩掉。

解决方案：
1）接口层增加校验，如用户鉴权校验，id做基础校验，id<=0的直接拦截；
2）从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-value对写为key-null，缓存有效时间可以设置短点，如30秒。这样可以防止攻击用户反复用同一个id暴力攻击；

3）采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力。（宁可错杀一千不可放过一人）

3. 缓存击穿

这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力。和缓存雪崩不同的是，缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库

解决方案：

1）设置热点数据永远不过期，异步线程处理。

2）加写回操作加互斥锁，查询失败默认值快速返回。

3）缓存预热

系统上线后，将相关可预期（例如排行榜）热点数据直接加载到缓存。

写一个缓存刷新页面，手动操作热点数据（例如广告推广）上下线。

4. 数据不一致

在缓存机器的带宽被打满，或者机房网络出现波动时，缓存更新失败，新数据没有写入缓存，就会导致缓存和 DB 的数据不一致。缓存 rehash 时，某个缓存机器反复异常，多次上下线，更新请求多次 rehash。这样，一份数据存在多个节点，且每次 rehash 只更新某个节点，导致一些缓存节点产生脏数据。

- Cache 更新失败后，可以进行重试，则将重试失败的 key 写入mq，待缓存访问恢复后，将这些 key 从缓存删除。这些 key 在再次被查询时，重新从 DB 加载，从而保证数据的一致性
- 缓存时间适当调短，让缓存数据及早过期后，然后从 DB 重新加载，确保数据的最终一致性。
- 不采用 rehash 漂移策略，而采用缓存分层策略，尽量避免脏数据产生。

5. 数据并发竞争

数据并发竞争在大流量系统也比较常见，比如车票系统，如果某个火车车次缓存信息过期，但仍然有大量用户在查询该车次信息。又比如微博系统中，如果某条微博正好被缓存淘汰，但这条微博仍然有大量的转发、评论、赞。上述情况都会造成并发竞争读取的问题。

- 加写回操作加互斥锁，查询失败默认值快速返回。
- 对缓存数据保持多个备份，减少并发竞争的概率

6. 热点key问题

明星结婚、离婚、出轨这种特殊突发事件，比如奥运、春节这些重大活动或节日，还比如秒杀、双12、618 等线上促销活动，都很容易出现 Hot key 的情况。

如何提前发现HotKey?

- 对于重要节假日、线上促销活动这些提前已知的事情，可以提前评估出可能的热 key 来。
- 而对于突发事件，无法提前评估，可以通过 Spark，对应流任务进行实时分析，及时发现新发布的热点 key。而对于之前已发出的事情，逐步发酵成为热 key 的，则可以通过 Hadoop 对批处理任务离线计算，找出最近历史数据中的高频热 key。

解决方案：

- 这 n 个 key 分散存在多个缓存节点，然后 client 端请求时，随机访问其中某个后缀的 hotkey，这样就可以把热 key 的请求打散，避免一个缓存节点过载
- 缓存集群可以单节点进行主从复制和垂直扩容
- 利用应用内的前置缓存，但是需注意需要设置上限
- 延迟不敏感，定时刷新，实时感知用主动刷新
- 和缓存穿透一样，限制逃逸流量，单请求进行数据回源并刷新前置
- 无论如何设计，最后都要写一个兜底逻辑，千万级流量说来就来

7. BigKey问题

比如互联网系统中需要保存用户最新 1万 个粉丝的业务，比如一个用户个人信息缓存，包括基本资料、关系图谱计数、发 feed 统计等。微博的 feed 内容缓存也很容易出现，一般用户微博在 140 字以内，但很多用户也会发表 1千 字甚至更长的微博内容，这些长微博也就成了大 key

- 首先Redis底层数据结构里，根据Value的不同，会进行数据结构的重新选择
- 可以扩展新的数据结构，进行序列化构建，然后通过 restore 一次性写入
- 将大 key 分拆为多个 key，设置较长的过期时间

Redis分区容错

1. redis数据分区

Hash：（不稳定）

客户端分片：哈希+取余

节点伸缩：数据节点关系变化，导致数据迁移

迁移数量和添加节点数量有关：建议翻倍扩容

一个简单直观的想法是直接用Hash来计算，以Key做哈希后对节点数取模。可以看出，在key足够分散的情况下，均匀性可以获得，但一旦有节点加入或退出，所有的原有节点都会受到影响，稳定性无从谈起。

一致性Hash：（不均衡）

客户端分片：哈希+顺时针（优化取余）

节点伸缩：只影响邻近节点，但是还是有数据迁移

翻倍伸缩：保证最小迁移数据和负载均衡

一致性Hash可以很好的解决稳定问题，可以将所有的存储节点排列在收尾相接的Hash环上，每个key在计算Hash后会顺时针找到先遇到的一组存储节点存放。而当有节点加入或退出时，仅影响该节点在Hash环上顺时针相邻的后续节点，将数据从该节点接收或者给予。但这又带来均匀性的问题，即使可以将存储节点等距排列，也会在存储节点个数变化时带来数据的不均匀。

Codis的Hash槽

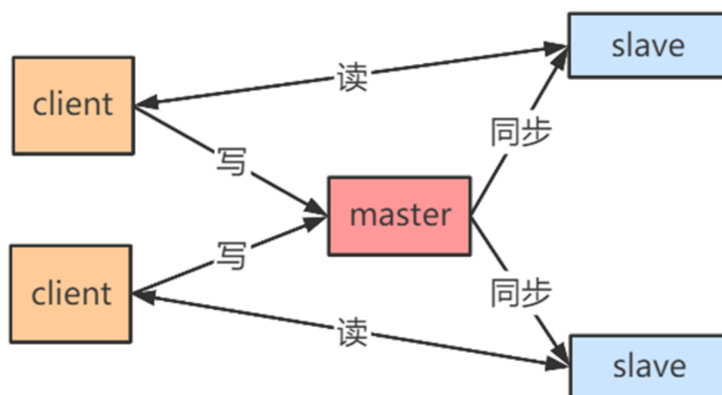
Codis 将所有的 key 默认划分为 1024 个槽位(slot)，它首先对客户端传过来的 key 进行 crc32 运算计算 哈希值，再将 hash 后的整数值对 1024 这个整数进行取模得到一个余数，这个余数就是对应 key 的槽位。

RedisCluster

Redis-cluster把所有的物理节点映射到[0-16383]个slot上,对key采用crc16算法得到hash值后对16384取模，基本上采用平均分配和连续分配的方式。

2. 主从模式=简单

主从模式最大的优点是部署简单，最少两个节点便可以构成主从模式，并且可以通过读写分离避免读和写同时不可用。不过，一旦 Master 节点出现故障，主从节点就无法自动切换，直接导致 SLA 下降。所以，主从模式一般适合业务发展初期，并发量低，运维成本低的情况



Redis 主从模式示意图

主从复制原理：

①通过从服务器发送到PSYNC命令给主服务器

②如果是首次连接，触发一次全量复制。此时主节点会启动一个后台线程，生成 RDB 快照文件

③主节点会将这个 RDB 发送给从节点，slave 会先写入本地磁盘，再从本地磁盘加载到内存中

④master会将此过程中的写命令写入缓存，从节点实时同步这些数据

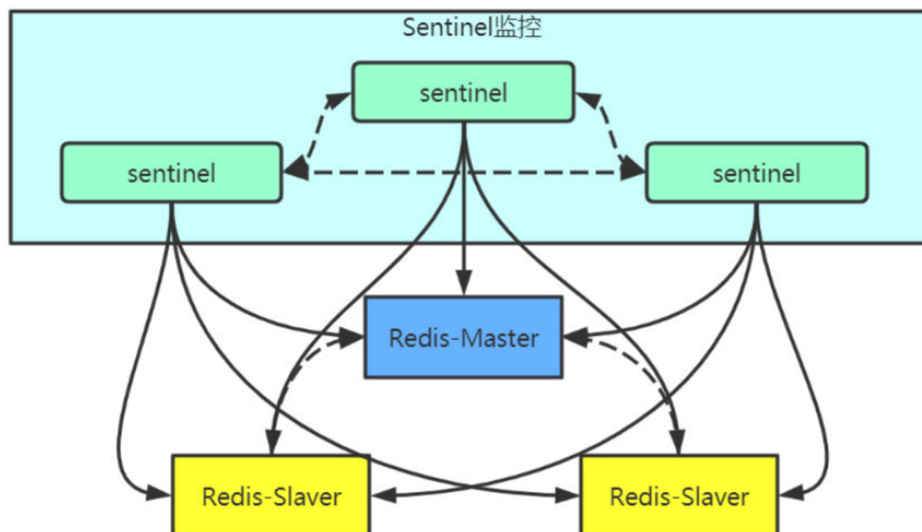
⑤如果网络断开了连接，自动重连后主节点通过命令传播增量复制给从节点部分缺少的数据

缺点

所有的slave节点数据的复制和同步都由master节点来处理，会照成master节点压力太大，使用主从从结构来解决，redis4.0中引入psync2 解决了slave重启后仍然可以增量同步。

3. 哨兵模式=读多

由一个或多个sentinel实例组成sentinel集群可以监视一个或多个主服务器和多个从服务器。哨兵模式适合读请求远多于写请求的业务场景，比如在秒杀系统中用来缓存活动信息。 如果写请求较多，当集群 Slave 节点数量多了后，Master 节点同步数据的压力会非常大。



当主服务器进入下线状态时，sentinel可以将该主服务器下的某一从服务器升级为主服务器继续提供服务，从而保证redis的高可用性。

检测主观下线状态

Sentinel每秒一次向所有与它建立了命令连接的实例(主服务器、从服务器和其他Sentinel)发送PING命 令

实例在down-after-milliseconds毫秒内返回无效回复Sentinel就会认为该实例主观下线(SDown)

检查客观下线状态

当一个Sentinel将一个主服务器判断为主观下线后 ，Sentinel会向监控这个主服务器的所有其他Sentinel发送查询主机状态的命令
如果达到Sentinel配置中的quorum数量的Sentinel实例都判断主服务器为主观下线，则该主服务器就会被判定为客观下线(ODown)。

选举Leader Sentinel
当一个主服务器被判定为客观下线后，监视这个主服务器的所有Sentinel会通过选举算法(raft)，选出一个Leader Sentinel去执行failover(故障转移)操作。

Raft算法
Raft协议是用来解决分布式系统一致性问题的协议。 Raft协议描述的节点共有三种状态:Leader，Follower，Candidate。 Raft协议将时间切分为一个个的Term(任期)，可以认为是一种“逻辑时间”。 选举流程:
①Raft采用心跳机制触发Leader选举系统启动后，全部节点初始化为Follower，term为0
②节点如果收到了RequestVote或者AppendEntries，就会保持自己的Follower身份
③节点如果一段时间内没收到AppendEntries消息，在该节点的超时时间内还没发现Leader，Follower就会转换成Candidate，自己开始竞选Leader。一旦转化为Candidate，该节点立即开始下面几件事情:
--增加自己的term，启动一个新的定时器
--给自己投一票，向所有其他节点发送RequestVote，并等待其他节点的回复。
④如果在计时器超时前，节点收到多数节点的同意投票，就转换成Leader。同时通过 AppendEntries，向其他节点发送通知。
⑤每个节点在一个term内只能投一票，采取先到先得的策略，Candidate投自己， Follower会投给第一个收到RequestVote的节点。
⑥Raft协议的定时器采取随机超时时间（选举的关键），先转为Candidate的节点会先发起投票，从而获得多数票。

主服务器的选择
当选举出Leader Sentinel后，Leader Sentinel会根据以下规则去从服务器中选择出新的主服务器。
1. 过滤掉主观、客观下线的节点
2. 选择配置slave-priority最高的节点，如果有则返回没有就继续选择
3. 选择出复制偏移量最大的系节点，因为复制偏移量越大则数据复制的越完整
4. 选择runid最小的节点，因为runid越小说明重启次数越少

故障转移
当Leader Sentinel完成新的主服务器选择后，Leader Sentinel会对下线的主服务器执行故障转移操作，
主要有三个步骤:
1. 它会将失效Master的其中一个Slave 升级为新的 Master , 并让失效 Master 的其他 Slave 改为复制新的 Master ;
2. 当客户端试图连接失效的Master时，集群会向客户端返回新Master的地址，使得集群当前状态只有一个Master。
Master 和 Slave 服务器切换后， Master 的 redis.conf 、 Slave 的 redis.conf 和 sentinel.conf 的配置文件的内容

都会发生相应的改变，即 Master 主服务器的 redis.conf配置文件中会多一行 replicaof 的配置， sentinel.conf 的监控目标会随之调换。

4. 集群模式=写多

为了避免单一节点负载过高导致不稳定，集群模式采用一致性哈希算法或者哈希槽的方法将 Key 分布到各个节点上。其中，每个 Master 节点后跟若干个 Slave 节点，用于出现故障时做主备切换，客户端可以连接任意 Master 节点，集群内部会按照不同 key 将请求转发到不同的 Master 节点

集群模式是如何实现高可用的呢？集群内部节点之间会互相定时探测对方是否存活，如果多数节点判断某个节点挂了，则会将其踢出集群，然后从 Slave 节点中选举出一个节点替补挂掉的 Master 节点。整个原理基本和哨兵模式一致

虽然集群模式避免了 Master 单节点的问题，但集群内同步数据时会占用一定的带宽。所以，只有在写操作比较多的情况下人们才使用集群模式，其他大多数情况，使用哨兵模式都能满足需求

5. 分布式锁

利用Watch实现Redis乐观锁

乐观锁基于CAS(Compare And Swap)比较并替换思想，不会产生锁等待而消耗资源，但是需要反复的重试，但也是因为重试的机制，能比较快的响应。因此我们可以利用redis来实现乐观锁（秒杀）。具体思路如下:

1. 利用redis的watch功能，监控这个redisKey的状态值
2. 获取redisKey的值，创建redis事务，给这个key的值+1
3. 执行这个事务，如果key的值被修改过则回滚，key不加1

利用setnx防止库存超卖

分布式锁是控制分布式系统之间同步访问共享资源的一种方式。 利用Redis的单线程特性对共享资源进行串行化处理

```
// 获取锁推荐使用set的方式
String result = jedis.set(lockKey, requestId, "NX", "EX", expireTime);
String result = jedis.setnx(lockKey, requestId); //如线程死掉，其他线程无法获取到锁
```

```
// 释放锁，非原子操作，可能会释放其他线程刚加上的锁
if (requestId.equals(jedis.get(lockKey))) {
  jedis.del(lockKey);
}
// 推荐使用redis+lua脚本
String lua = "if redis.call('get',KEYS[1]) == ARGV[1] then return redis.call('del',KEYS[1]) else return 0 end";
Object result = jedis.eval(lua, Collections.singletonList(lockKey),
```

分布式锁存在的问题：

- 客户端长时间阻塞导致锁失效问题
  计算时间内异步启动另外一个线程去检查的问题，这个key是否超时，当锁超时时间快到期且逻辑未执行完，延长锁超时时间。
- Redis服务器时钟漂移问题导致同时加锁 redis的过期时间是依赖系统时钟的，如果时钟漂移过大时 理论上是可能出现的 会影响到过期时间的计算。
- 单点实例故障，锁未及时同步导致丢失
  RedLock算法

1. 获取当前时间戳T0，配置时钟漂移误差T1
2. 短时间内逐个获取全部N/2+1个锁，结束时间点T2
3. 实际锁能使用的处理时长变为：TTL –（T2 – T0）– T1

该方案通过多节点来防止Redis的单点故障，效果一般，也无法防止：

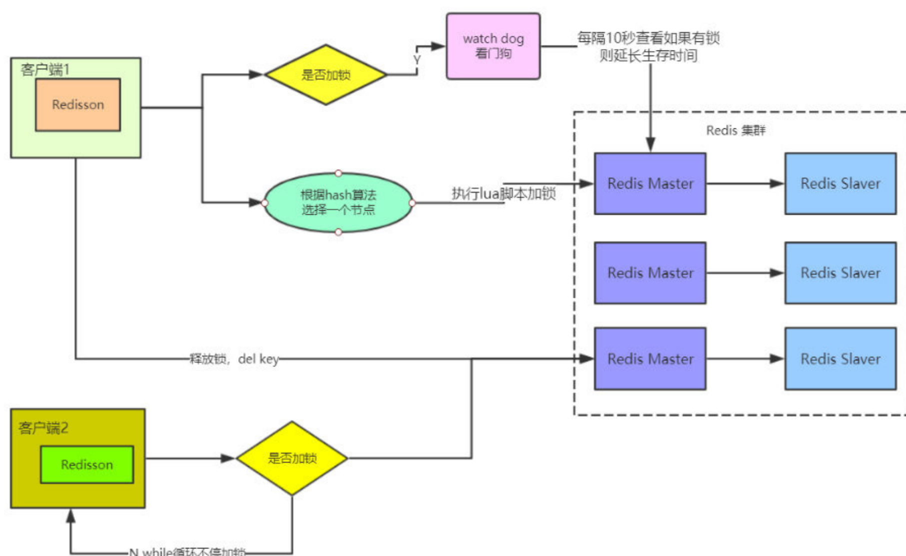
- 主从切换导致的两个客户端同时持有锁
  大部分情况下持续时间极短，而且使用Redlock在切换的瞬间获取到节点的锁，也存在问题。已经是极低概率的时间，无法避免。Redis分布式锁适合幂等性事务，如果一定要保证安全，应该使用Zookeeper或者DB，但是，性能会急剧下降。

与zookeeper分布式锁对比

- redis 分布式锁，其实需要自己不断去尝试获取锁，比较消耗性能。
- zk 分布式锁，注册个监听器即可，不需要不断主动尝试获取锁，ZK获取锁会按照加锁的顺序，所以是公平锁，性能和mysql差不多，和redis差别大

Redission生产环境的分布式锁

Redisson是基于NIO的Netty框架上的一个Java驻内存数据网格(In–Memory Data Grid)分布式锁开源组件。



但当业务必须要数据的强一致性，即不允许重复获得锁，比如金融场景(重复下单，重复转账)，请不要使用redis分布式锁。可以使用CP模型实现，比如:zookeeper和etcd。

	Redis	zookeeper	etcd
一致性算法	无	paxos(ZAB)	raft
CAP	AP	CP	CP
高可用	主从集群	n+1	n+1
实现	setNX	createNode	restfulAPI

6. redis心跳检测

在命令传播阶段，从服务器默认会以每秒一次的频率向主服务器发送ACK命令:

1. 检测主从的连接状态 检测主从服务器的网络连接状态
   lag的值应该在0或1之间跳动，如果超过1则说明主从之间的连接有 故障。
2. 辅助实现min-slaves,Redis可以通过配置防止主服务器在不安全的情况下执行写命令

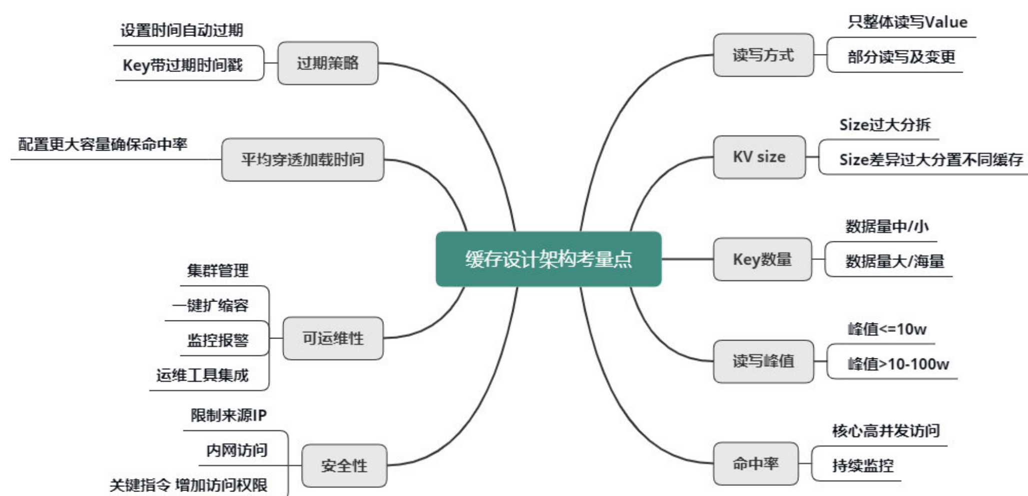
```
min-slaves-to-write 3 (min-replicas-to-write 3 )

min-slaves-max-lag 10 (min-replicas-max-lag 10)
```

   上面的配置表示:从服务器的数量少于3个，或者三个从服务器的延迟(lag)值都大于或等于10 秒时，主服务器将拒绝执行写命令。
3. 检测命令丢失，增加重传机制

如果因为网络故障，主服务器传播给从服务器的写命令在半路丢失，那么当从服务器向主服务器发 送REPLCONF ACK命令时，主服务器将发觉从服务器当前的复制偏移量少于自己的复制偏移量， 然后主服务器就会根据从服务器提交的复制偏移量，在复制积压缓冲区里面找到从服务器缺少的数据，并将这些数据重新发送给从服务器。

Redis实战

1. Redis优化



读写方式

简单来说就是不用keys等，用range、contains之类。比如，用户粉丝数，大 V 的粉丝更是高达几千万甚至过亿，因此，获取粉丝列表只能部分获取。另外在判断某用户是否关注了另外一个用户时，也只需要关注列表上进行检查判断，然后返回 True/False 或 0/1 的方式更为高效。

KV size

如果单个业务的 KV size 过大，需要分拆成多个 KV 来缓存。拆分时应考虑访问频率

key 的数量

如果数据量巨大，则在缓存中尽可能只保留频繁访问的热数据，对于冷数据直接访问 DB。

读写峰值

如果小于 10万 级别，简单分拆到独立 Cache 池即可 如果达到 100万 级的QPS，则需要对 Cache 进行分层处理，可以同时使用 Local-Cache 配合远程 cache，甚至远程缓存内部继续分层叠加分池进行处理。（多级缓存）

命中率

缓存的命中率对整个服务体系的性能影响甚大。对于核心高并发访问的业务，需要预留足够的容量，确保核心业务缓存维持较高的命中率。比如微博中的 Feed Vector Cache（热点资讯），常年的命中率高达 99.5% 以上。为了持续

保持缓存的命中率，缓存体系需要持续监控，及时进行故障处理或故障转移。同时在部分缓存节点异常、命中率下降时，故障转移方案，需要考虑是采用一致性 Hash 分布的访问漂移策略，还是采用数据多层备份策略。

过期策略

可以设置较短的过期时间，让冷 key 自动过期；也可以让 key 带上时间戳，同时设置较长的过期时间，比如很多业务系统内部有这样一些 key：key_20190801。

缓存穿透时间

平均缓存穿透加载时间在某些业务场景下也很重要，对于一些缓存穿透后，加载时间特别长或者需要复杂计算的数据，而且访问量还比较大的业务数据，要配置更多容量，维持更高的命中率，从而减少穿透到 DB 的概率，来确保整个系统的访问性能。

缓存可运维性

对于缓存的可运维性考虑，则需要考虑缓存体系的集群管理，如何进行一键扩缩容，如何进行缓存组件的升级和变更，如何快速发现并定位问题，如何持续监控报警，最好有一个完善的运维平台，将各种运维工具进行集成。

缓存安全性

对于缓存的安全性考虑，一方面可以限制来源 IP，只允许内网访问，同时加密鉴权访问。

2. Redis热升级

> 在 Redis 需要升级版本或修复 bug 时，如果直接重启变更，由于需要数据恢复，这个过程需要近 10 分钟的时间，时间过长，会严重影响系统的可用性。面对这种问题，可以对 Redis 扩展热升级功能，从而在毫秒级完成升级操作，完全不影响业务访问。

热升级方案如下，首先构建一个 Redis 壳程序，将 redisServer 的所有属性（包括redisDb、client等）保存为全局变量。然后将 Redis 的处理逻辑代码全部封装到动态连接库 so 文件中。Redis 第一次启动，从磁盘加载恢复数据，在后续升级时，通过指令，壳程序重新加载 Redis 新的 redis-4.so 到 redis-5.so 文件，即可完成功能升级，毫秒级完成 Redis 的版本升级。而且整个过程中，所有 Client 连接仍然保留，在升级成功后，原有 Client 可以继续进行读写操作，整个过程对业务完全透明。

Kafka篇

Why kafka

消息队列的作用：异步、削峰填谷、解耦

中小型公司，技术实力较为一般，技术挑战不是特别高，用 RabbitMQ （开源、社区活跃）是不错的选择；大型公司，基础架构研发实力较强，用 RocketMQ（Java二次开发） 是很好的选择。

如果是大数据领域的实时计算、日志采集等场景，用 Kafka 是业内标准的，绝对没问题，社区活跃度很高，绝对不会黄，何况几乎是全世界这个领域的事实性规范。

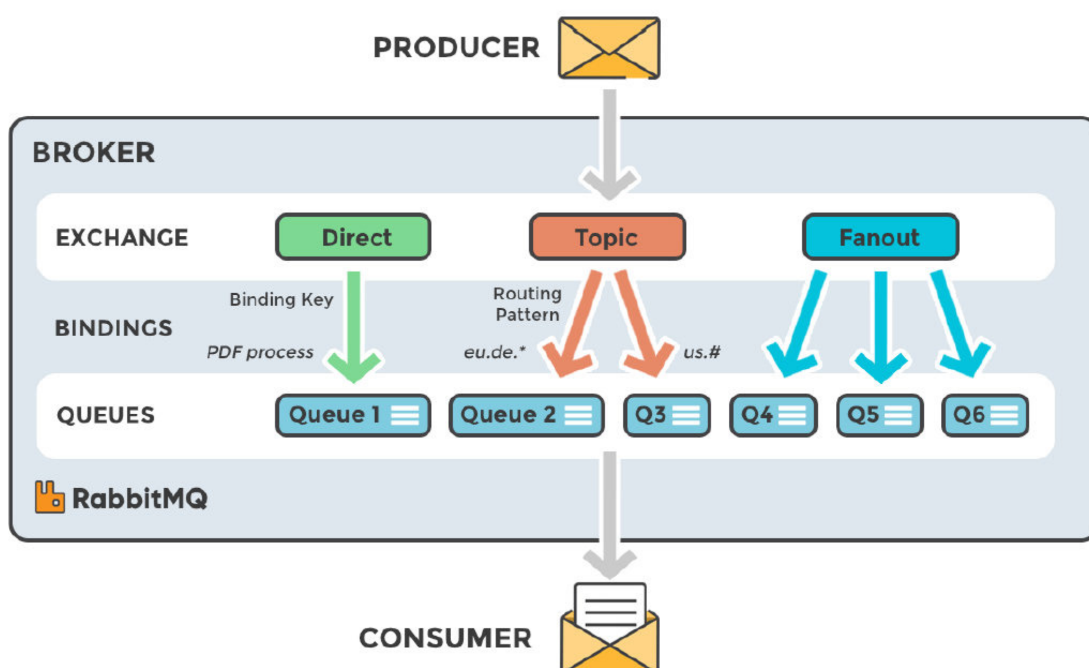
	RabbitMQ	RocketMQ	Kafka
单机吞吐量	1w量级	10w量级	10w量级
开发语言	Erlang	Java	Java和Scala
消息延迟	微秒	毫秒	毫秒
消息丢失	可能性很低	参数优化后可以0丢失	参数优化后可以0丢失
消费模式	推拉	推拉	拉取
主题数量对吞吐量的影响	\	几百上千个主题会对吞吐量有一个小的影响	几十上百个主题会极大影响吞吐量
可用性	高（主从）	很高（主从）	很高（分布式）

RabbitMQ

RabbitMQ开始是用在电信业务的可靠通信的，也是少有的几款支持AMQP协议的产品之一。

优点：

- 轻量级，快速，部署使用方便
- 支持灵活的路由配置。RabbitMQ中，在生产者和队列之间有一个交换器模块。根据配置的路由规则，生产者发送的消息可以发送到不同的队列中。路由规则很灵活，还可以自己实现。
- RabbitMQ的客户端支持大多数的编程语言，支持AMQP协议。



缺点：

- 如果有大量消息堆积在队列中，性能会急剧下降
- 每秒处理几万到几十万的消息。如果应用要求高的性能，不要选择RabbitMQ。
- RabbitMQ是Erlang开发的，功能扩展和二次开发代价很高。

RocketMQ

借鉴了Kafka的设计并做了很多改进，几乎具备了消息队列应该具备的所有特性和功能。

- RocketMQ主要用于有序，事务，流计算，消息推送，日志流处理，binlog分发等场景。
- 经过了历次的双11考验，性能，稳定性可靠性没的说。
- java开发，阅读源代码、扩展、二次开发很方便。
- 对电商领域的响应延迟做了很多优化。
- 每秒处理几十万的消息，同时响应在毫秒级。如果应用很关注响应时间，可以使用RocketMQ。

- 性能比RabbitMQ高一个数量级，。
- 支持死信队列，DLX 是一个非常有用的特性。它可以处理异常情况下，消息不能够被消费者正确消费而被置入死信队列中的情况，后续分析程序可以通过消费这个死信队列中的内容来分析当时所遇到的异常情况，进而可以改善和优化系统。

缺点：

跟周边系统的整合和兼容不是很好。

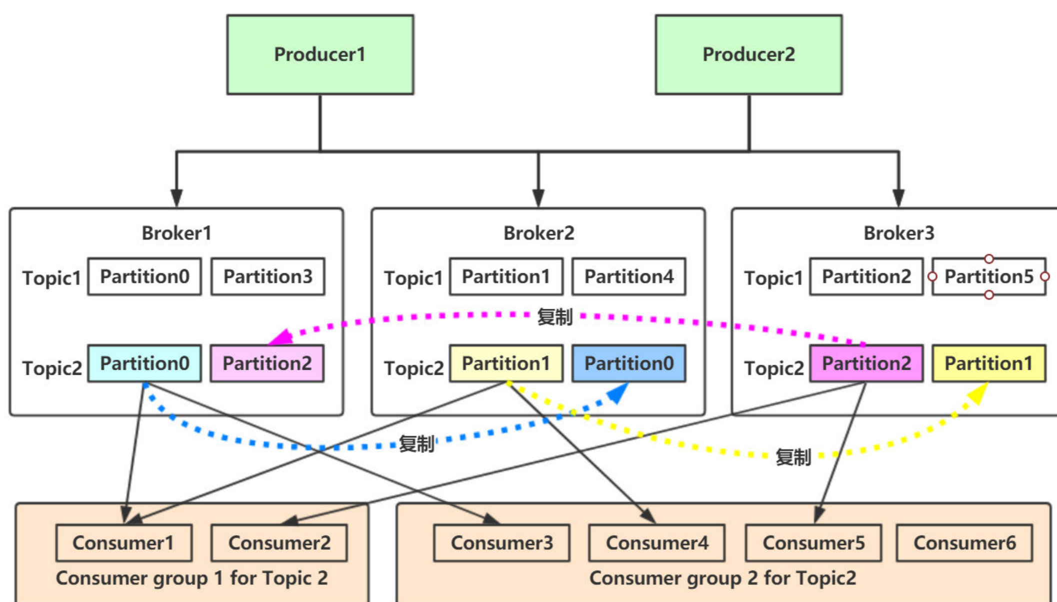
Kafka

高可用，几乎所有相关的开源软件都支持，满足大多数的应用场景，尤其是大数据和流计算领域，

- Kafka高效，可伸缩，消息持久化。支持分区、副本和容错。
- 对批处理和异步处理做了大量的设计，因此Kafka可以得到非常高的性能。
- 每秒处理几十万异步消息消息，如果开启了压缩，最终可以达到每秒处理2000w消息的级别。
- 但是由于是异步的和批处理的，延迟也会高，不适合电商场景。

What Kafka

- Producer API：允许应用程序将记录流发布到一个或多个Kafka主题。
- Consumer API：允许应用程序订阅一个或多个主题并处理为其生成的记录流。
- Streams API：允许应用程序充当流处理器，将输入流转换为输出流。



消息Message

Kafka的数据单元称为消息。可以把消息看成是数据库里的一个“数据行”或一条“记录”。

批次

为了提高效率，消息被分批写入Kafka。提高吞吐量却加大了响应时间

主题Topic

通过主题进行分类，类似数据库中的表，

分区Partition

Topic可以被分成若干分区分布于kafka集群中，方便扩容
单个分区内是有序的，partition设置为一才能保证全局有序

副本Replicas

每个主题被分为若干个分区，每个分区有多个副本。

生产者Producer

生产者在默认情况下把消息均衡地分布到主题的所有分区上：

- 直接指定消息的分区
- 根据消息的key散列取模得出分区
- 轮询指定分区。

消费者Comsumer

消费者通过偏移量来区分已经读过的消息，从而消费消息。把每个分区最后读取的消息偏移量保存在Zookeeper 或Kafka上，如果消费者关闭或重启，它的读取状态不会丢失。

消费组ComsumerGroup

消费组保证每个分区只能被一个消费者使用，避免重复消费。如果群组内一个消费者失效，消费组里的其他消费者可以接管失效消费者的工作再平衡，重新分区

节点Broker

连接生产者和消费者，单个broker可以轻松处理数千个分区以及每秒百万级的消息量。

- broker接收来自生产者的消息，为消息设置偏移量，并提交消息到磁盘保存。
- broker为消费者提供服务，响应读取分区的请求，返回已经提交到磁盘上的消息。

集群

每隔分区都有一个首领，当分区被分配给多个broker时，会通过首领进行分区复制。

生产者Offset

消息写入的时候，每一个分区都有一个offset，即每个分区的最新最大的offset。

消费者Offset

不同消费组中的消费者可以针对一个分区存储不同的Offset，互不影响

LogSegment

- 一个分区由多个LogSegment组成，
- 一个LogSegment由 `.log .index .timeindex` 组成
- `.log` 追加是顺序写入的，文件名是以文件中第一条message的offset来命名的
- `.Index` 进行日志删除的时候和数据查找的时候可以快速定位。
- `.timeStamp` 则根据时间戳查找对应的偏移量。

How Kafka

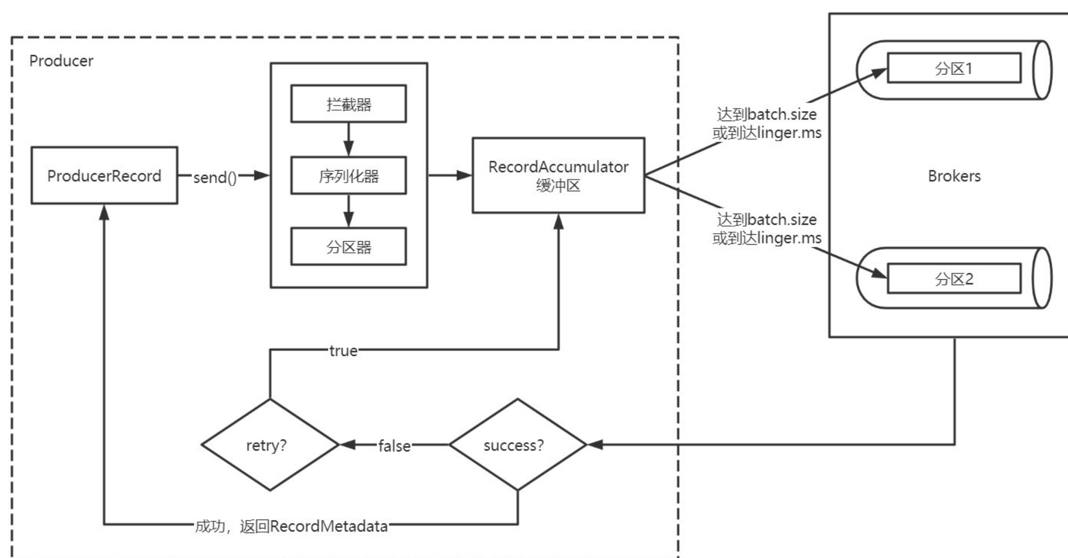
优点

- 高吞吐量：单机每秒处理几十上百万的消息量。即使存储了TB及消息，也保持稳定的性能。
  零拷贝 减少内核态到用户态的拷贝，磁盘通过sendfile实现DMA 拷贝Socket buffer
  顺序读写 充分利用磁盘顺序读写的超高性能
  页缓存mmap，将磁盘文件映射到内存, 用户通过修改内存就能修改磁盘文件。
- 高性能：单节点支持上千个客户端，并保证零停机和零数据丢失。
- 持久化：将消息持久化到磁盘。通过将数据持久化到硬盘以及replication防止数据丢失。
- 分布式系统，易扩展。所有的组件均为分布式的，无需停机即可扩展机器。
- 可靠性 – Kafka是分布式，分区，复制和容错的。
- 客户端状态维护：消息被处理的状态是在Consumer端维护，当失败时能自动平衡。

应用场景

- 日志收集：用Kafka可以收集各种服务的Log，通过大数据平台进行处理；
- 消息系统：解耦生产者和消费者、缓存消息等；
- 用户活动跟踪：Kafka经常被用来记录Web用户或者App用户的各种活动，如浏览网页、搜索、点击等活动，这些活动信息被各个服务器发布到Kafka的Topic中，然后消费者通过订阅这些Topic来做运营数据的实时的监控分析，也可保存到数据库；

生产消费基本流程



1．Producer创建时，会创建一个Sender线程并设置为守护线程。

2．生产的消息先经过拦截器->序列化器->分区器，然后将消息缓存在缓冲区。

3．批次发送的条件为：缓冲区数据大小达到batch.size或者linger.ms达到上限。

4．批次发送后，发往指定分区，然后落盘到broker；

   - acks=0只要将消息放到缓冲区，就认为消息已经发送完成。
   - acks=1表示消息只需要写到主分区即可。在该情形下，如果主分区收到消息确认之后就宕机了，而副本分区还没来得及同步该消息，则该消息丢失。
   - acks=all（默认）首领分区会等待所有的ISR副本分区确认记录。该处理保证了只要有一个ISR副本分区存活，消息就不会丢失。

5．如果生产者配置了retrires参数大于0并且未收到确认，那么客户端会对该消息进行重试。

6．落盘到broker成功，返回生产元数据给生产者。

Leader选举

- Kafka会在Zookeeper上针对每个Topic维护一个称为ISR（in-sync replica）的集合
- 当集合中副本都跟Leader中的副本同步了之后，kafka才会认为消息已提交
- 只有这些跟Leader保持同步的Follower才应该被选作新的Leader
- 假设某个topic有N+1个副本，kafka可以容忍N个服务器不可用，冗余度较低

  如果ISR中的副本都丢失了，则：
  - 可以等待ISR中的副本任何一个恢复，接着对外提供服务，需要时间等待
  - 从OSR中选出一个副本做Leader副本，此时会造成数据丢失

副本消息同步

首先，Follower 发送 FETCH 请求给 Leader。接着，Leader 会读取底层日志文件中的消 息数据，再更新它内存中的 Follower 副本的 LEO 值，更新为 FETCH 请求中的 fetchOffset 值。最后，尝试更新分区高水位值。Follower 接收到 FETCH 响应之后，会把消息写入到底层日志，接着更新 LEO 和 HW 值。

相关概念：LEO和HW。

- LEO：即日志末端位移(log end offset)，记录了该副本日志中下一条消息的位移值。如果LEO=10，那么表示该副本保存了10条消息，位移值范围是[0, 9]
- HW：水位值HW（high watermark）即已备份位移。对于同一个副本对象而言，其HW值不会大于LEO值。小于等于HW值的所有消息都被认为是“已备份”的（replicated）

Rebalance

- 组成员数量发生变化
- 订阅主题数量发生变化
- 订阅主题的分区数发生变化

leader选举完成后，当以上三种情况发生时，Leader根据配置的RangeAssignor开始分配消费方案，即哪个consumer负责消费哪些topic的哪些partition。一旦完成分配，leader会将这个方案封装进SyncGroup请求中发给coordinator，非leader也会发SyncGroup请求，只是内容为空。coordinator接收到分配方案之后会把方案塞进SyncGroup的response中发给各个consumer。这样组内的所有成员就都知道自己应该消费哪些分区了。

分区分配算法RangeAssignor

原理是按照消费者总数和分区总数进行整除运算平均分配给所有的消费者。

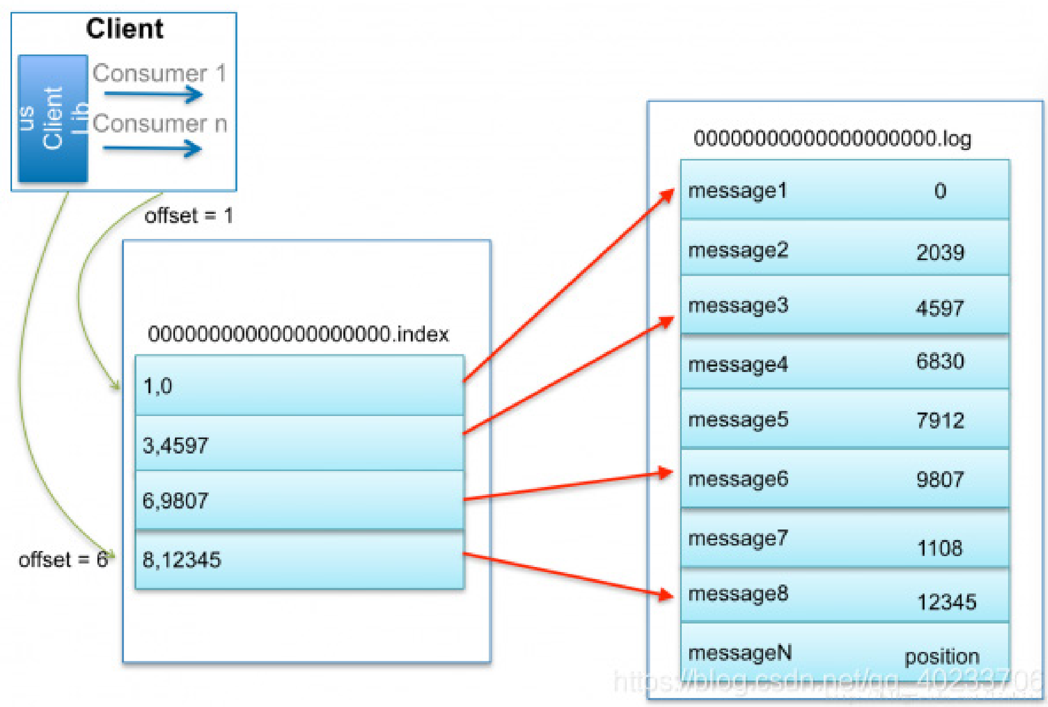
订阅Topic的消费者按照名称的字典序排序，分均分配，剩下的字典序从前往后分配

增删改查

```
kafka-topics.sh --zookeeper localhost:2181/myKafka --create --topic topic_x
                 --partitions 1 --replication-factor 1
kafka-topics.sh --zookeeper localhost:2181/myKafka --delete --topic topic_x
kafka-topics.sh --zookeeper localhost:2181/myKafka --alter --topic topic_x
                 --config max.message.bytes=1048576
kafka-topics.sh --zookeeper localhost:2181/myKafka --describe --topic topic_x
```

如何查看偏移量为23的消息?

通过查询跳跃表ConcurrentSkipListMap，定位到在00000000000000000000.index ，通过二分法在偏移量索引文件中找到不大于 23 的最大索引项，即offset 20 那栏，然后从日志分段文件中的物理位置为320 开始顺序查找偏移量为 23 的消息。



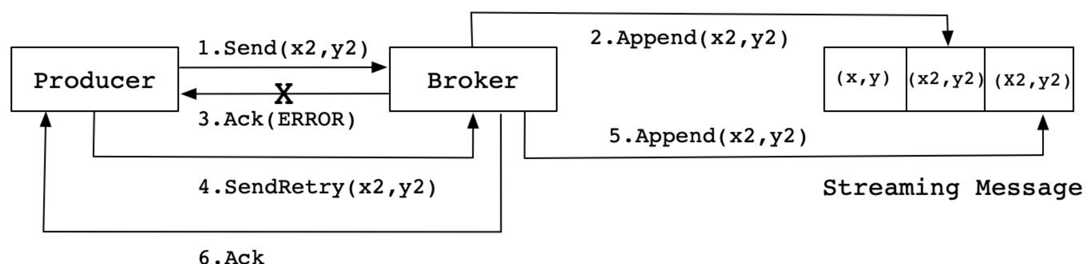
切分文件

- 大小分片 当前日志分段文件的大小超过了 broker 端参数 log.segment.bytes 配置的值
- 时间分片 当前日志分段中消息的最大时间戳与系统的时间戳的差值大于log.roll.ms配置的值
- 索引分片 偏移量或时间戳索引文件大小达到broker端 log.index.size.max.bytes配置的值
- 偏移分片 追加的消息的偏移量与当前日志分段的偏移量之间的差值大于 Integer.MAX_VALUE

一致性

幂等性

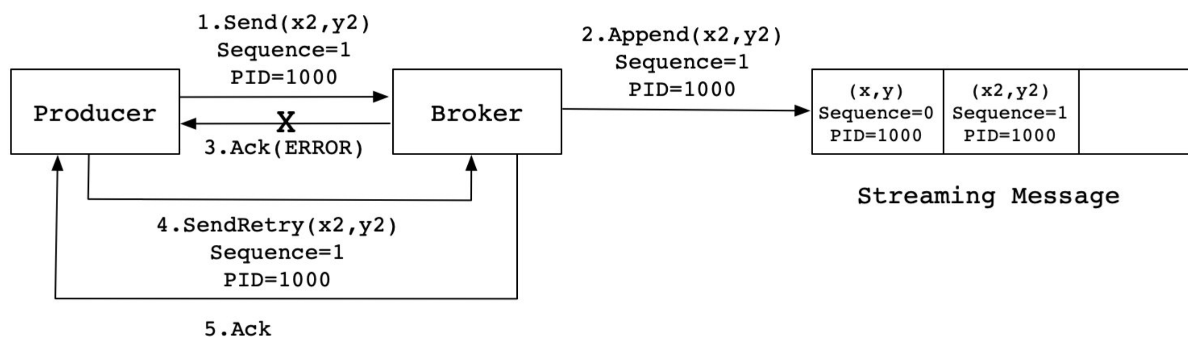
保证在消息重发的时候，消费者不会重复处理。即使在消费者收到重复消息的时候，重复处理，也要保证最终结果的一致性。所谓幂等性，数学概念就是：f(f(x)) = f(x)



如何实现？

添加唯一ID，类似于数据库的主键，用于唯一标记一个消息。

```
ProducerID：#在每个新的Producer初始化时，会被分配一个唯一的PID
SequenceNumber：#对于每个PID发送数据的每个Topic都对应一个从0开始单调递增的SN值
```



如何选举

1. 使用 Zookeeper 的分布式锁选举控制器，并在节点加入集群或退出集群时通知控制器。
2. 控制器负责在节点加入或离开集群时进行分区Leader选举。
3. 控制器使用epoch忽略小的纪元来避免脑裂：两个节点同时认为自己是当前的控制器。

可用性

- 创建Topic的时候可以指定 --replication-factor 3 ，表示不超过broker的副本数
- 只有Leader是负责读写的节点，Follower定期地到Leader上Pull数据。
- ISR是Leader负责维护的与其保持同步的Replica列表，即当前活跃的副本列表。如果一个Follow落后太多，Leader会将它从ISR中移除。选举时优先从ISR中挑选Follower。
- 设置 acks=all 。Leader收到了ISR中所有Replica的ACK，才向Producer发送ACK。

面试题

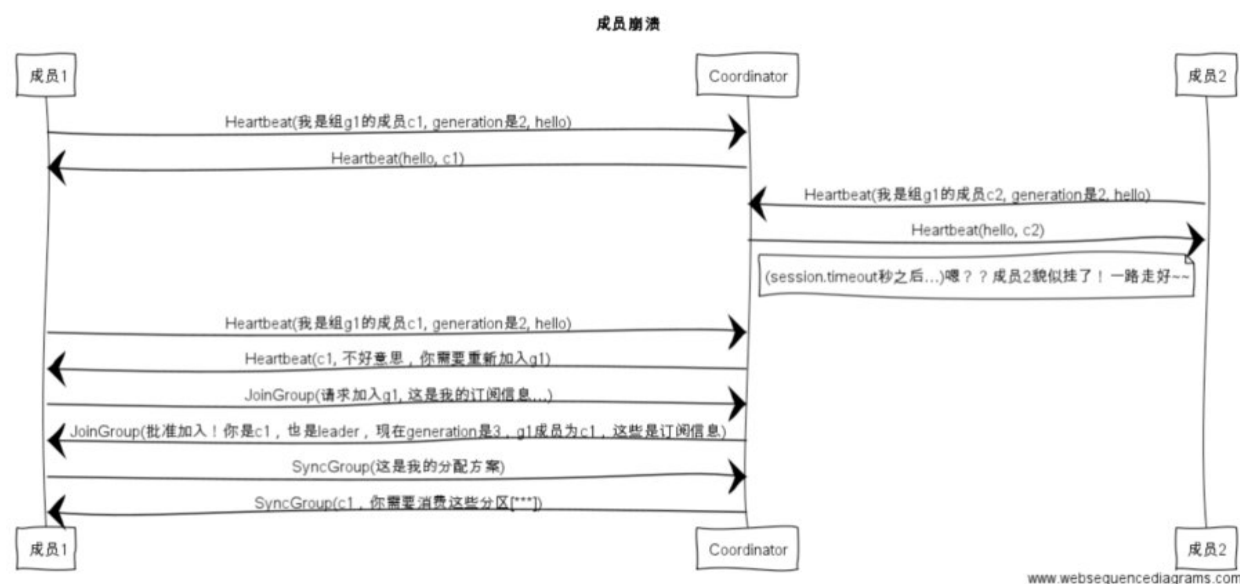
线上问题rebalance

> 因集群架构变动导致的消费组内重平衡，如果kafka集内节点较多，比如数百个，那重平衡可能会耗时导致数分钟到数小时，此时kafka基本处于不可用状态，对kafka的TPS影响极大

产生的原因：

- 组成员数量发生变化
- 订阅主题数量发生变化
- 订阅主题的分区数发生变化

  组成员崩溃和组成员主动离开是两个不同的场景。因为在崩溃时成员并不会主动地告知coordinator此事，coordinator有可能需要一个完整的session.timeout周期(心跳周期)才能检测到这种崩溃，这必然会造成consumer的滞后。可以说离开组是主动地发起rebalance；而崩溃则是被动地发起rebalance。



解决方案：

```
加大超时时间 session.timout.ms=6s
加大心跳频率 heartbeat.interval.ms=2s
增长推送间隔 max.poll.interval.ms=t+1 minutes
```

这些年，为了进阿里背过的面试题

ZooKeeper的作用

目前，Kafka 使用 ZooKeeper 存放集群元数据、成员管理、Controller 选举，以及其他一些管理类任务。之后，等 KIP-500 提案完成后，Kafka 将完全不再依赖于 ZooKeeper。

- **存放元数据**是指主题分区的所有数据都保存在 ZooKeeper 中，其他“人”都要与它保持对齐。
- **成员管理**是指 Broker 节点的注册、注销以及属性变更等 。
- **Controller** 选举是指选举集群 Controller，包括但不限于主题删除、参数配置等。

一言以蔽之:**KIP-500 ，是使用社区自研的基于 Raft 的共识算法，实现 Controller 自选举**。
同样是存储元数据，这几年**基于Raft算法的etcd**认可度越来越高
越来越多的系统开始用它保存关键数据。比如，秒杀系统经常用它保存各节点信息，以便控制消费 MQ 的服务数量。还有些**业务系统的配置数据**，也会通过 etcd 实时**同步给业务系统的各节点**，比如，秒杀管理后台会使用 etcd 将**秒杀活动的配置数据实时同步给秒杀 API 服务各节点。**

Replica副本的作用

Kafka 只有 Leader 副本才能 对外提供读写服务，响应 Clients 端的请求。Follower 副本只是采用拉(PULL)的方式，被动地同步 Leader 副本中的数据，并且在 Leader 副本所在的 Broker 宕机后，随时准备应聘 Leader 副本。

- 自 **Kafka 2.4** 版本开始，社区可以通过配置参数，允许 Follower 副本有限度地提供读服务。
- 之前确保一致性的主要手段是高水位机制， 但高水位值无法保证 Leader 连续变更场景下的数据一致性，因此，社区引入了 **Leader Epoch** 机制，来修复高水位值的弊端。

为什么不支持读写分离?

- **自 Kafka 2.4 之后**，Kafka 提供了有限度的读写分离。
- **场景不适用**。读写分离适用于那种读负载很大，而写操作相对不频繁的场景。
- **同步机制**。Kafka 采用 PULL 方式实现 Follower 的同步，同时复制延迟较大。

如何防止重复消费

- 代码层面每次消费需提交offset
- 通过Mysql的唯一键约束，结合Redis查看id是否被消费，存Redis可以直接使用set方法
- 量大且允许误判的情况下，使用布隆过滤器也可以

如何保证数据不会丢失

- **生产者**生产消息可以通过comfirm配置ack=all解决
- Broker同步过程中leader宕机可以通过配置ISR副本+重试解决
- **消费者**丢失可以关闭自动提交offset功能，系统处理完成时提交offset

如何保证顺序消费

- 单 topic，单partition，单 consumer，单线程消费，吞吐量低，不推荐
- **如只需保证单key有序**，为每个key申请单独内存 queue，每个线程分别消费一个内存 queue 即可，这样就能保证单key（例如用户id、活动id）顺序性。

【线上】如何解决积压消费

- **修复consumer**，使其具备消费能力，并且扩容N台
- 写一个**分发的程序，**将Topic均匀分发到临时Topic中
- 同时**起N台consumer，**消费不同的**临时Topic**

如何避免消息积压

- 提高消费并行度
- 批量消费
- 减少组件IO的交互次数
- 优先级消费

```
if (maxOffset - curOffset > 100000) {
  // TODO 消息堆积情况的优先处理逻辑
  // 未处理的消息可以选择丢弃或者打日志
  return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
}
// TODO 正常消费过程
return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
```

如何设计消息队列

需要支持快速水平扩容，broker+partition，partition放不同的机器上，增加机器时将数据根据topic做迁移，分布式需要考虑一致性、可用性、分区容错性

- **一致性：**生产者的消息确认、消费者的幂等性、Broker的数据同步
- **可用性：**数据如何保证不丢不重、数据如何持久化、持久化时如何读写
- **分区容错：**采用何种选举机制、如何进行多副本同步
- **海量数据：**如何解决消息积压、海量Topic性能下降

性能上，可以借鉴时间轮、零拷贝、IO多路复用、顺序读写、压缩批处理

Spring篇

设计思想&Beans

1. IOC 控制反转

IoC（Inverse of Control:控制反转）是一种设计思想，就是将原本在程序中手动创建对象的控制权，交由Spring框架来管理。IoC 在其他语言中也有应用，并非 Spring 特有。

IoC 容器是 Spring用来实现 IoC 的载体，IoC 容器实际上就是个Map（key，value）,Map 中存放的是各种对象。将对象之间的相互依赖关系交给 IoC 容器来管理，并由 IoC 容器完成对象的注入。这样可以很大程度上简化应用的开发，把应用从复杂的依赖关系中解放出来。 IoC 容器就像是一个工厂一样，当我们需要创建一个对象的时候，只需要配置好配置文件/注解即可，完全不用考虑对象是如何被创建出来的。

DI 依赖注入

DI:（Dependancy Injection：依赖注入)站在容器的角度，将对象创建依赖的其他对象注入到对象中。

2. AOP 动态代理

AOP(Aspect-Oriented Programming:面向切面编程)能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可拓展性和可维护性。

Spring AOP就是基于动态代理的，如果要代理的对象，实现了某个接口，那么Spring AOP会使用JDKProxy，去创建代理对象，而对于没有实现接口的对象，就无法使用 JDK Proxy 去进行代理了，这时候Spring AOP会使用基于asm框架字节流的Cglib动态代理，这时候Spring AOP会使用 Cglib 生成一个被代理对象的子类来作为代理。

3. Bean生命周期

单例对象： singleton

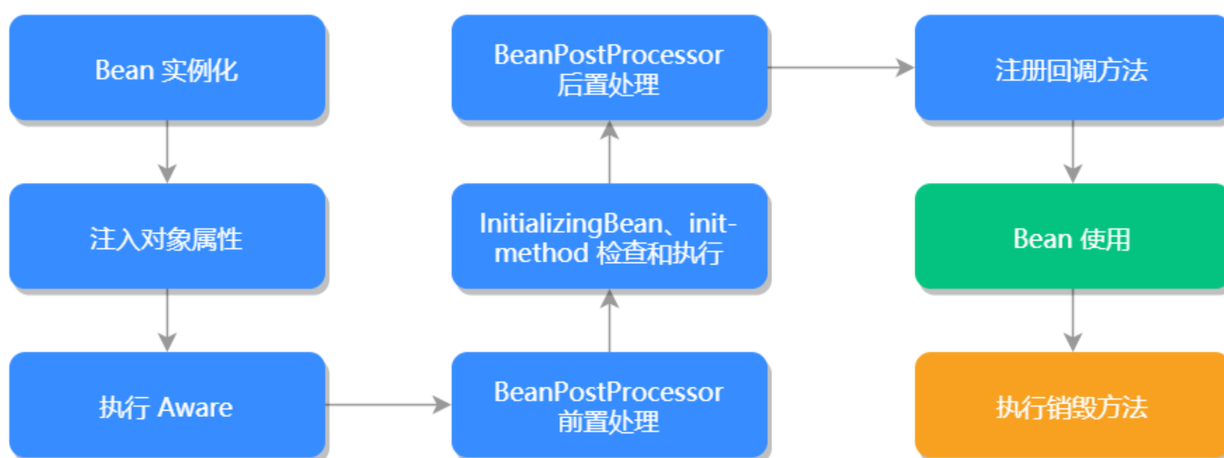
总结：单例对象的生命周期和容器相同

多例对象： prototype

出生：使用对象时spring框架为我们创建

活着：对象只要是在使用过程中就一直活着

死亡：当对象长时间不用且没有其它对象引用时，由java的垃圾回收机制回收



IOC容器初始化加载Bean流程：

```
@Override
public void refresh() throws BeansException, IllegalStateException { synchronized (this.startupShutdown-
Monitor) {
  // 第一步:刷新前的预处理
  prepareRefresh();
  //第二步: 获取BeanFactory并注册到 BeanDefitionRegistry
  ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
  // 第三步:加载BeanFactory的预准备工作(BeanFactory进行一些设置，比如context的类加载器等)
  prepareBeanFactory(beanFactory);
  try {
    // 第四步:完成BeanFactory准备工作后的前置处理工作
    postProcessBeanFactory(beanFactory);
    // 第五步:实例化BeanFactoryPostProcessor接口的Bean
    invokeBeanFactoryPostProcessors(beanFactory);
    // 第六步:注册BeanPostProcessor后置处理器，在创建bean的后执行
    registerBeanPostProcessors(beanFactory);
    // 第七步:初始化MessageSource组件(做国际化功能;消息绑定，消息解析);
    initMessageSource();
    // 第八步:注册初始化事件派发器
```

```
    initApplicationEventMulticaster();
    // 第九步:子类重写这个方法，在容器刷新的时候可以自定义逻辑
    onRefresh();
    // 第十步:注册应用的监听器。就是注册实现了ApplicationListener接口的监听器
    registerListeners();
    //第十一步:初始化所有剩下的非懒加载的单例bean 初始化创建非懒加载方式的单例Bean实例(未设置属性)
    finishBeanFactoryInitialization(beanFactory);
    //第十二步: 完成context的刷新。主要是调用LifecycleProcessor的onRefresh()方法，完成创建
    finishRefresh();
    }
   ......
 }
```

总结：

四个阶段

- **实例化** Instantiation
- **属性赋值** Populate
- **初始化** Initialization
- **销毁** Destruction

多个扩展点

- 影响多个Bean
  - BeanPostProcessor
  - InstantiationAwareBeanPostProcessor
- 影响单个Bean
  - Aware

完整流程

1. 实例化一个Bean－－也就是我们常说的new；
2. 按照Spring上下文对实例化的Bean进行配置－－也就是IOC注入；
3. 如果这个Bean已经实现了BeanNameAware接口，会调用它实现的setBeanName(String)方法，也就是根据就是Spring配置文件中Bean的id和name进行传递
4. 如果这个Bean已经实现了BeanFactoryAware接口，会调用它实现setBeanFactory(BeanFactory)也就是Spring配置文件配置的Spring工厂自身进行传递；
5. 如果这个Bean已经实现了ApplicationContextAware接口，会调用setApplicationContext(Application-Context)方法，和4传递的信息一样但是因为ApplicationContext是BeanFactory的子接口，所以更加灵活
6. 如果这个Bean关联了BeanPostProcessor接口，将会调用postProcessBeforeInitialization()方法，Bean PostProcessor经常被用作是Bean内容的更改，由于这个是在Bean初始化结束时调用那个的方法，也可以被

应用于内存或缓存技术
7. 如果Bean在Spring配置文件中配置了init-method属性会自动调用其配置的初始化方法。
8. 如果这个Bean关联了BeanPostProcessor接口，将会调用postProcessAfterInitialization()，打印日志或者三级缓存技术里面的bean升级
9. 以上工作完成以后就可以应用这个Bean了，那这个Bean是一个Singleton的，所以一般情况下我们调用同一个id的Bean会是在内容地址相同的实例，当然在Spring配置文件中也可以配置非Singleton，这里我们不做赘述。
10. 当Bean不再需要时，会经过清理阶段，如果Bean实现了DisposableBean这个接口，或者根据spring配置的destroy-method属性，调用实现的destroy()方法

4. Bean作用域

名称	作用域
singleton	**单例对象，默认值的作用域**
prototype	**每次获取都会创建一个新的 bean 实例**
request	每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP request内有效。
session	在一次 HTTP session 中，容器将返回同一个实例
global-session	将对象存入到web项目集群的session域中,若不存在集群,则global session相当于session

默认作用域是singleton，多个线程访问同一个bean时会存在线程不安全问题

保障线程安全方法：

1. 在Bean对象中尽量避免定义可变的成员变量（不太现实）。
2. 在类中定义一个ThreadLocal成员变量，将需要的可变成员变量保存在 ThreadLocal 中

ThreadLocal：

每个线程中都有一个自己的ThreadLocalMap类对象，可以将线程自己的对象保持到其中，各管各的，线程可以正确的访问到自己的对象。

将一个共用的ThreadLocal静态实例作为key，将不同对象的引用保存到不同线程的ThreadLocalMap中，然后在线程执行的各处通过这个静态ThreadLocal实例的get()方法取得自己线程保存的那个对象，避免了将这个对象作为参数传递的麻烦。

5. 循环依赖

循环依赖其实就是循环引用，也就是两个或者两个以上的 Bean 互相持有对方，最终形成闭环。比如A 依赖于B，B又依赖于A

Spring中循环依赖场景有:

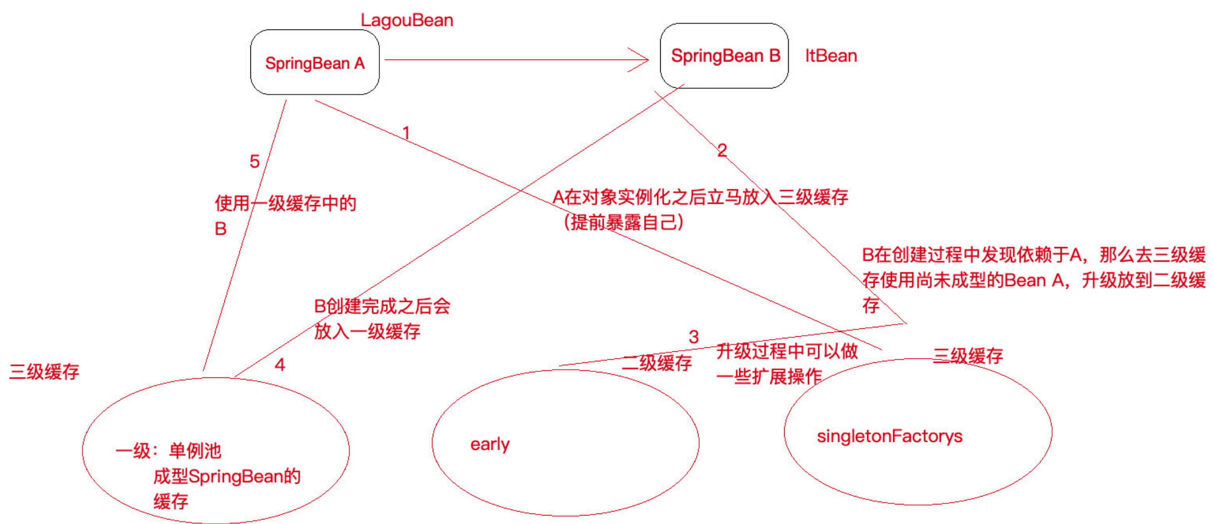
- prototype 原型 bean循环依赖
- 构造器的循环依赖（构造器注入）
- Field 属性的循环依赖（set注入）

其中，构造器的循环依赖问题无法解决，在解决属性循环依赖时，可以使用懒加载，spring采用的是提前暴露对象的方法。

懒加载@Lazy解决循环依赖问题

Spring启动的时候会把所有bean信息(包括XML和注解)解析转化成Spring能够识别的BeanDefinition并存到Hashmap里供下面的初始化时用，然后对每个 BeanDefinition 进行处理。普通 Bean 的初始化是在容器启动初始化阶段执行的，而被lazy-init=true修饰的 bean 则是在从容器里第一次进行context.getBean() 时进行触发。

三级缓存解决循环依赖问题



1. Spring容器初始化ClassA通过构造器初始化对象后提前暴露到Spring容器中的singletonFactorys（三级缓存中）。
2. ClassA调用setClassB方法，Spring首先尝试从容器中获取ClassB，此时ClassB不存在Spring 容器中。
3. Spring容器初始化ClassB，ClasssB首先将自己暴露在三级缓存中，然后从Spring容器一级、二级、三级缓存中一次中获取ClassA 。
4. 获取到ClassA后将自己实例化放入单例池中，实例 ClassA通过Spring容器获取到ClassB，完成了自己对象初始化操作。
5. 这样ClassA和ClassB都完成了对象初始化操作，从而解决了循环依赖问题。

Spring注解

1. @SpringBoot

声明bean的注解

@Component 通用的注解，可标注任意类为 Spring 组件

@Service 在业务逻辑层使用（service层）

@Repository 在数据访问层使用（dao层）

@Controller 在展现层使用，控制器的声明（controller层）

注入bean的注解

@Autowired：默认按照类型来装配注入，@Qualifier：可以改成名称

@Resource：默认按照名称来装配注入，JDK的注解，新版本已经弃用

@Autowired注解原理

@Autowired的使用简化了我们的开发，

实现 AutowiredAnnotationBeanPostProcessor 类，该类实现了 Spring 框架的一些扩展接口。
实现 BeanFactoryAware 接口使其内部持有了 BeanFactory（可轻松的获取需要依赖的的 Bean）。
实现 MergedBeanDefinitionPostProcessor 接口，实例化Bean 前获取到里面的 @Autowired 信息并缓存下来；
实现 postProcessPropertyValues 接口， 实例化Bean 后从缓存取出注解信息，通过反射将依赖对象设置到 Bean 属性里面。

@SpringBootApplication

```
@SpringBootApplication
public class JpaApplication {
    public static void main(String[] args) {
        SpringApplication.run(JpaApplication.class, args);
    }
}
```

@SpringBootApplication注解等同于下面三个注解：

- @SpringBootConfiguration： 底层是Configuration注解，说白了就是支持JavaConfig的方式来进行配置
- @EnableAutoConfiguration：开启自动配置功能
- @ComponentScan：就是扫描注解，默认是扫描当前类下的package

其中 `@EnableAutoConfiguration` 是关键(启用自动配置)，内部实际上就去加载META-INF/spring.factories文件的信息，然后筛选出以 `EnableAutoConfiguration` 为key的数据，加载到IOC容器中，实现自动配置功能！

它主要加载了@SpringBootApplication注解主配置类，这个@SpringBootApplication注解主配置类里边最主要的功能就是SpringBoot开启了一个@EnableAutoConfiguration注解的自动配置功能。

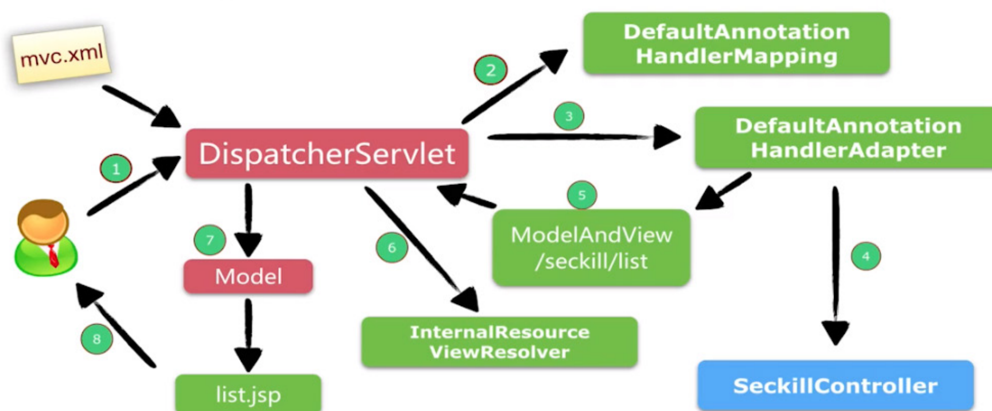
@EnableAutoConfiguration作用：
它主要利用了一个
EnableAutoConfigurationImportSelector选择器给Spring容器中来导入一些组件。

```
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration
```

2. @SpringMVC

```
@Controller 声明该类为SpringMVC中的Controller
@RequestMapping 用于映射Web请求
@ResponseBody 支持将返回值放在response内，而不是一个页面，通常用户返回json数据
@RequestBody 允许request的参数在request体中，而不是在直接连接在地址后面。
@PathVariable 用于接收路径参数
@RequestMapping("/hello/{name}")申明的路径，将注解放在参数中前，即可获取该值，通常作为Restful的接口实现方法。
```

SpringMVC原理



- 客户端（浏览器）发送请求，直接请求到 DispatcherServlet 。
- DispatcherServlet 根据请求信息调用 HandlerMapping ，解析请求对应的 Handler 。
- 解析到对应的 Handler （也就是 Controller 控制器）后，开始由HandlerAdapter 适配器处理。
- HandlerAdapter 会根据 Handler 来调用真正的处理器开处理请求，并处理相应的业务逻辑。
- 处理器处理完业务后，会返回一个 ModelAndView 对象， Model 是返回的数据对象
- ViewResolver 会根据逻辑 View 查找实际的 View 。
- DispaterServlet 把返回的 Model 传给 View （视图渲染）。
- 把 View 返回给请求者（浏览器）

3. @SpringMybatis

```
@Insert： 插入sql ,和xml insert sql语法完全一样
@Select： 查询sql, 和xml select sql语法完全一样
@Update： 更新sql, 和xml update sql语法完全一样
@Delete： 删除sql, 和xml delete sql语法完全一样
@Param： 入参
@Results： 设置结果集合@Result： 结果
@ResultMap： 引用结果集合
@SelectKey： 获取最新插入id
```

mybatis如何防止sql注入?

简单的说就是#{}是经过预编译的，是安全的，${}是未经过预编译的，仅仅是取变量的值，是非安全的，存在SQL注入。在编写mybatis的映射语句时，尽量采用“#{xxx}”这样的格式。如果需要实现动态传入表名、列名，还需要做如下修改：添加属性statementType="STATEMENT"，同时sql里的属有变量取值都改成${xxxx}

Mybatis和Hibernate的区别

Hibernate 框架：

Hibernate是一个开放源代码的对象关系映射框架,它对JDBC进行了非常轻量级的对象封装,建立对象与数据库表的映射。是一个全自动的、完全面向对象的持久层框架。

Mybatis框架：

Mybatis是一个开源对象关系映射框架，原名：ibatis,2010年由谷歌接管以后更名。是一个半自动化的持久层框架。

区别：

开发方面

在项目开发过程当中，就速度而言：

hibernate开发中，sql语句已经被封装，直接可以使用，加快系统开发；

Mybatis 属于半自动化，sql需要手工完成，稍微繁琐；
但是，凡事都不是绝对的，如果对于庞大复杂的系统项目来说，复杂语句较多，hibernate 就不是好方案。

sql优化方面
Hibernate 自动生成sql,有些语句较为繁琐，会多消耗一些性能；
Mybatis 手动编写sql，可以避免不需要的查询，提高系统性能；

对象管理比对
Hibernate 是完整的对象-关系映射的框架，开发工程中，无需过多关注底层实现，只要去管理对象即可；
Mybatis 需要自行管理映射关系；

4. @Transactional

```
@EnableTransactionManagement
@Transactional
```

注意事项：
①事务函数中不要处理耗时任务，会导致长期占有数据库连接。
②事务函数中不要处理无关业务，防止产生异常导致事务回滚。

事务传播属性
1) REQUIRED（默认属性） 如果存在一个事务，则支持当前事务。如果没有事务则开启一个新的事务。
2) MANDATORY 支持当前事务，如果当前没有事务，就抛出异常。
3) NEVER 以非事务方式执行，如果当前存在事务，则抛出异常。
4) NOT_SUPPORTED 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
5) REQUIRES_NEW 新建事务，如果当前存在事务，把当前事务挂起。
6) SUPPORTS 支持当前事务，如果当前没有事务，就以非事务方式执行。
7) NESTED （局部回滚） 支持当前事务，新增Savepoint点，与当前事务同步提交或回滚。 **嵌套事务一个非常重要的概念就是内层事务依赖于外层事务。外层事务失败时，会回滚内层事务所做的动作。而内层事务操作失败并不会引起外层事务的回滚。**

Spring源码阅读

1. Spring中的设计模式

参考：spring中的设计模式

单例设计模式：Spring 中的 Bean 默认都是单例的。

工厂**设计模式：**Spring使用工厂模式通过 BeanFactory 、 ApplicationContext 创建bean 对象。

代理设计模式：Spring AOP 功能的实现。

观察者模式： Spring 事件驱动模型就是观察者模式很经典的一个应用。

适配器模式：Spring AOP 的增强或通知(Advice)使用到了适配器模式、spring MVC 中也是用到了适配器模式适配 Controller 。

SpringCloud篇

Why SpringCloud

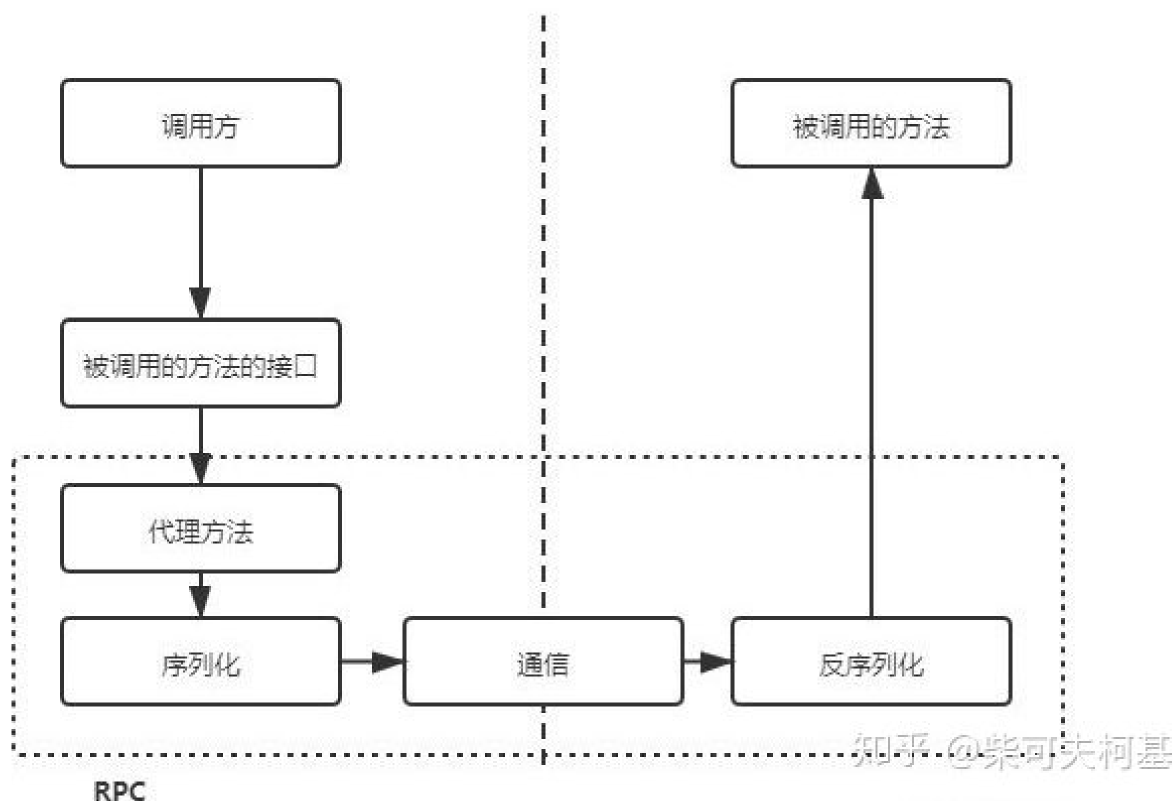
> Spring cloud 是一系列框架的有序集合。它利用 spring boot 的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用 spring boot 的开发风格做到一键启动和部署。

SpringCloud（微服务解决方案）	**Dubbo（分布式服务治理框架）**
Rest API （轻量、灵活、swagger）	RPC远程调用（高效、耦合）
Eureka、Nacos	Zookeeper
使用方便	性能好
即将推出SpringCloud2.0	断档5年后17年重启

SpringBoot是Spring推出用于解决传统框架配置文件冗余,装配组件繁杂的基于Maven的解决方案,旨在快速搭建单个微服务，SpringCloud是依赖于SpringBoot的,而SpringBoot并不是依赖与SpringCloud,甚至还可以和Dubbo进行优秀的整合开发

MartinFlower **提出的微服务之间是通过RestFulApi进行通信，具体实现**

- RestTemplate：**基于HTTP协议**
- Feign：封装了ribbon和Hystrix 、RestTemplate **简化了客户端开发工作量**
- RPC：**基于TCP协议，序列化和传输效率提升明显**
- MQ：**异步解耦微服务之间的调用**



Spring Boot

> Spring Boot 通过简单的步骤就可以创建一个 Spring 应用。
> Spring Boot 为 Spring 整合第三方框架提供了开箱即用功能。
> Spring Boot 的核心思想是约定大于配置。

Spring Boot 解决的问题

- 搭建后端框架时需要手动添加 Maven 配置，涉及很多 XML 配置文件，增加了搭建难度和时间成本。
- 将项目编译成 war 包，部署到 Tomcat 中，项目部署依赖 Tomcat，这样非常不方便。
- 应用监控做的比较简单，通常都是通过一个没有任何逻辑的接口来判断应用的存活状态。

Spring Boot 优点

自动装配：Spring Boot 会根据某些规则对所有配置的 Bean 进行初始化。可以减少了很多重复性的工作。

比如使用 MongoDB 时，只需加入 MongoDB 的 Starter 包，然后配置 的连接信息，就可以直接使用 Mongo-Template 自动装配来操作数据库了。简化了 Maven Jar 包的依赖，降低了烦琐配置的出错几率。

内嵌容器：Spring Boot 应用程序可以不用部署到外部容器中，比如 Tomcat。

应用程序可以直接通过 Maven 命令编译成可执行的 jar 包，通过 java-jar 命令启动即可，非常方便。

应用监控： Spring Boot 中自带监控功能 Actuator，可以实现对程序内部运行情况进行监控，比如 Bean 加载情况、环境变量、日志信息、线程信息等。当然也可以自定义跟业务相关的监控，通过Actuator 的端点信息进行暴露。

```
spring-boot-starter-web         //用于快速构建基于 Spring MVC 的 Web 项目。
spring-boot-starter-data-redis  //用于快速整合并操作 Redis。
spring-boot-starter-data-mongodb //用于对 MongoDB 的集成。
spring-boot-starter-data-jpa    //用于操作 MySQL。
```

自定义一个Starter

1. 创建 Starter 项目，定义 Starter 需要的配置（Properties）类，比如数据库的连接信息；
2. 编写自动配置类，自动配置类就是获取配置，根据配置来自动装配 Bean；
3. 编写 spring.factories 文件加载自动配置类，Spring 启动的时候会扫描 spring.factories 文件，；
4. 编写配置提示文件 spring-configuration-metadata.json（不是必须的），在添加配置的时候，我们想要知道具体的配置项是什么作用，可以通过编写提示文件来提示；
5. 在项目中引入自定义 Starter 的 Maven 依赖，增加配置值后即可使用。

Spring Boot Admin（将 actuator 提供的数据进行可视化）

- 显示应用程序的监控状态、查看 JVM 和线程信息
- 应用程序上下线监控
- 可视化的查看日志、动态切换日志级别
- HTTP 请求信息跟踪等实用功能

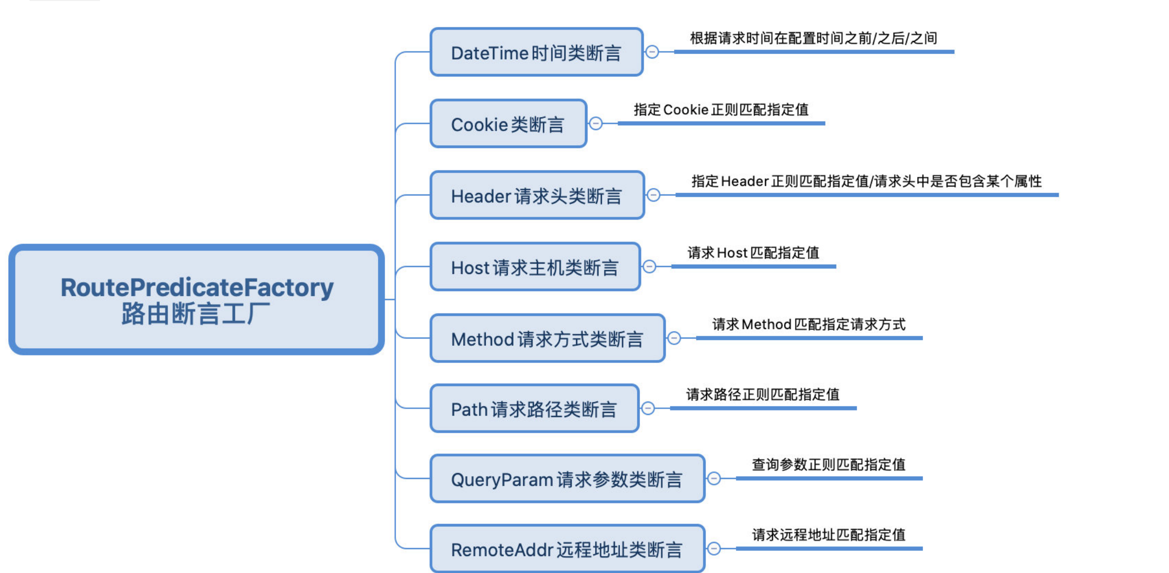
GateWay / Zuul

> GateWay目标是取代Netflflix Zuul，它基于Spring5.0+SpringBoot2.0+WebFlux等技术开发，提供统一的路由方式（反向代理）并且基于 Filter(定义过滤器对请求过滤，完成一些功能) 链的方式提供了网关基本的功能，例如：鉴权、流量控制、熔断、路径重写、日志监控。

组成：

- 路由route： 网关最基础的工作单元。路由由一个ID、一个目标URL、一系列的断言（匹配条件判断）和Filter过滤器组成。如果断言为true，则匹配该路由。
- 断言predicates：参考了Java8中的断言Predicate，匹配Http请求中的所有内容（类似于nginx中的location匹配一样），如果断言与请求相匹配则路由。
- 过滤器filter：标准的Spring webFilter，使用过滤器在请求之前或者之后执行业务逻辑。

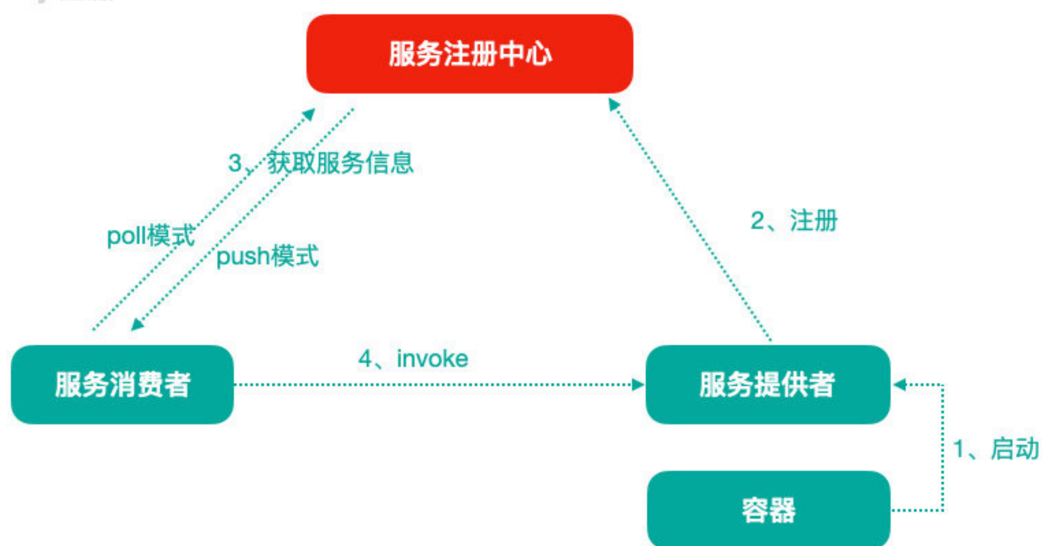
请求前 `pre` 类型过滤器：做**参数校验、权限校验、流量监控、日志输出、协议转换**等，
请求前 `post` 类型的过滤器：做**响应内容、响应头**的修改、**日志的输出、流量监控**等。



GateWayFilter 应用到单个路由路由上 、**GlobalFilter** 应用到所有的路由上

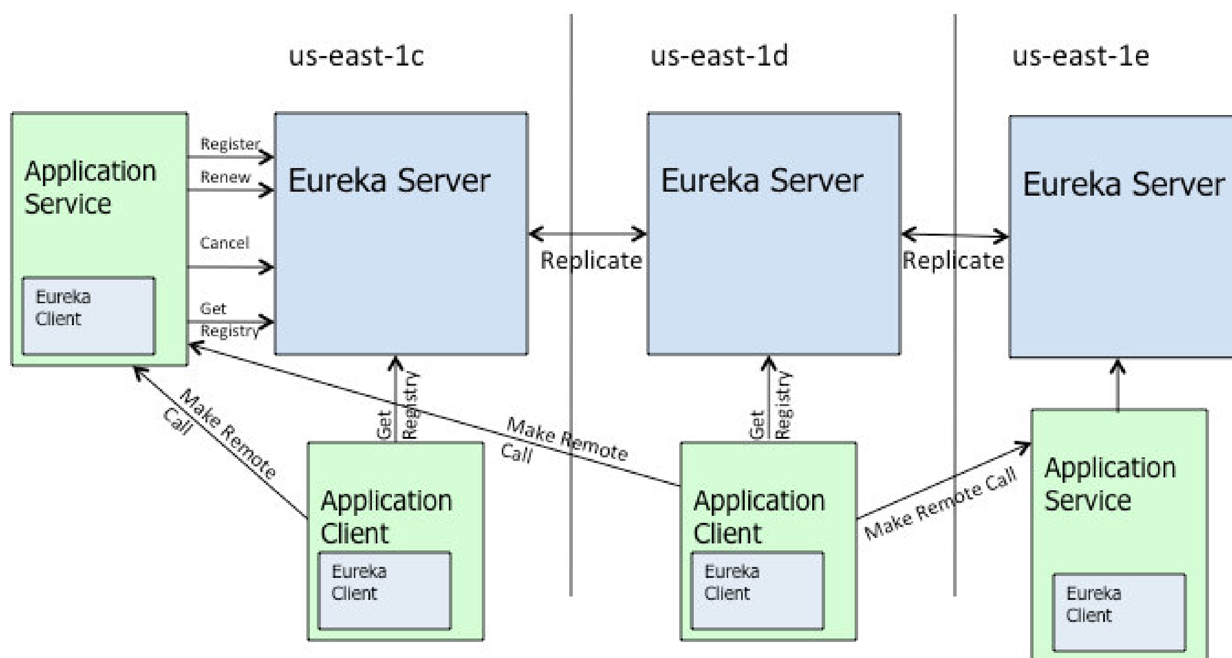
Eureka / Zookeeper

> 服务注册中心本质上是为了解耦服务提供者和服务消费者，为了支持弹性扩缩容特性，一个微服务的提供者的数量和分布往往是动态变化的。



区别	Zookeeper	Eureka	Nacos
CAP	CP	AP	CP/AP切换
可用性	选举期间不可用	自我保护机制，数据不是最新的	
组成	Leader和Follower	节点平等	
优势	分布式协调	注册与发现	注册中心和配置中心
底层	进程	服务	Jar包

Eureka通过**心跳检测、健康检查和客户端缓存**等机制，提高系统的灵活性、可伸缩性和可用性。

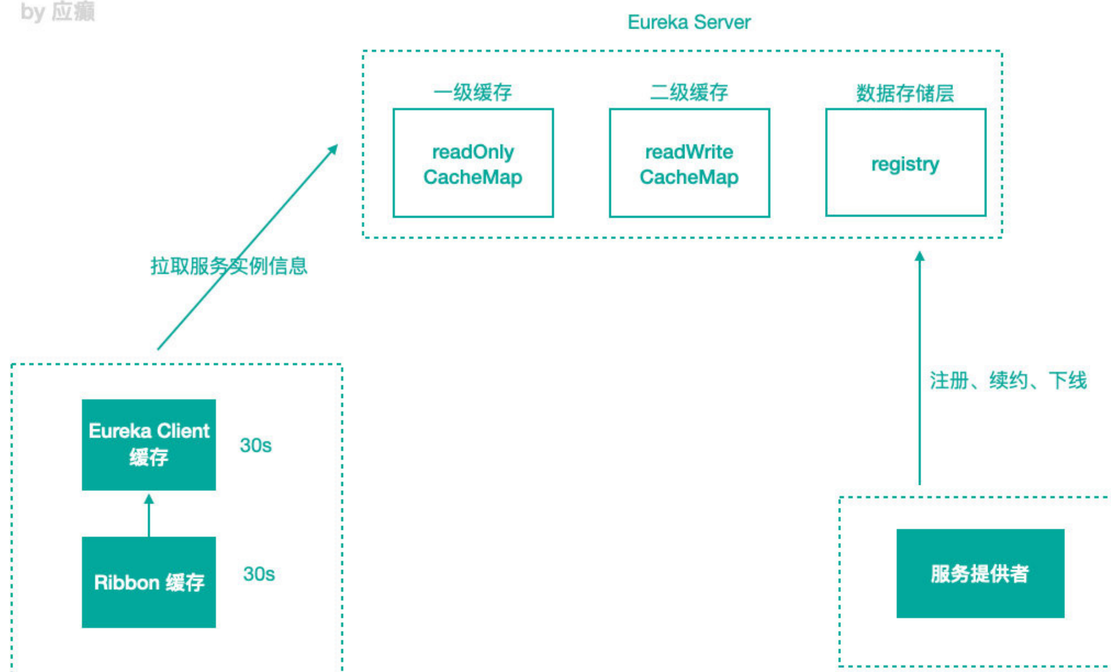


1. us-east-1c、us-east-1d，us-east-1e代表不同的机房，每一个Eureka Server都是一个集群。
2. Service作为服务提供者向Eureka中注册服务，Eureka接受到注册事件会在集群和分区中进行数据同步，Client作为消费端（服务消费者）可以从Eureka中获取到服务注册信息，进行服务调用。
3. 微服务启动后，会周期性地向Eureka发送心跳（默认周期为30秒）以续约自己的信息
4. Eureka在一定时间内（默认90秒）没有接收到某个微服务节点的心跳，Eureka将会注销该微服务节点
5. Eureka Client会缓存Eureka Server中的信息。即使所有的Eureka Server节点都宕掉，服务消费者依然可以使用缓存中的信息找到服务提供者

Eureka缓存

> 新服务上线后，服务消费者不能立即访问到刚上线的新服务，需要过一段时间后才能访问？或是将服务下线后，服务还是会被调用到，一段时候后才彻底停止服务，访问前期会导致频繁报错！

by 应癫



服务注册到注册中心后，服务实例信息是存储在Registry表中的，也就是内存中。但Eureka为了提高响应速度，在内部做了优化，加入了两层的缓存结构，将Client需要的实例信息，直接缓存起来，获取的时候直接从缓存中拿数据然后响应给 Client。

- 第一层缓存是readOnlyCacheMap，采用ConcurrentHashMap来存储数据的，主要负责定时与readWrite-CacheMap进行数据同步，默认同步时间为 30 秒一次。
- 第二层缓存是readWriteCacheMap，采用Guava来实现缓存。缓存过期时间默认为180秒，当服务下线、过期、注册、状态变更等操作都会清除此缓存中的数据。
- 如果两级缓存都无法查询，会触发缓存的加载，从存储层拉取数据到缓存中，然后再返回给 Client。

  Eureka之所以设计二级缓存机制，也是为了提高 Eureka Server 的响应速度，缺点是缓存会导致 Client获取不到最新的服务实例信息，然后导致无法快速发现新的服务和已下线的服务。

解决方案

- 我们可以缩短读缓存的更新时间让服务发现变得更加及时，或者直接将只读缓存关闭，同时可以缩短客户端如ribbon服务的定时刷新间隔，多级缓存也导致C层面（数据一致性）很薄弱。
- Eureka Server 中会有定时任务去检测失效的服务，将服务实例信息从注册表中移除，也可以将这个失效检测的时间缩短，这样服务下线后就能够及时从注册表中清除。

自我保护机制开启条件

- 期望最小每分钟能够续租的次数（实例* 频率 * 比例==10* 2 *0.85）
- 期望的服务实例数量（10）

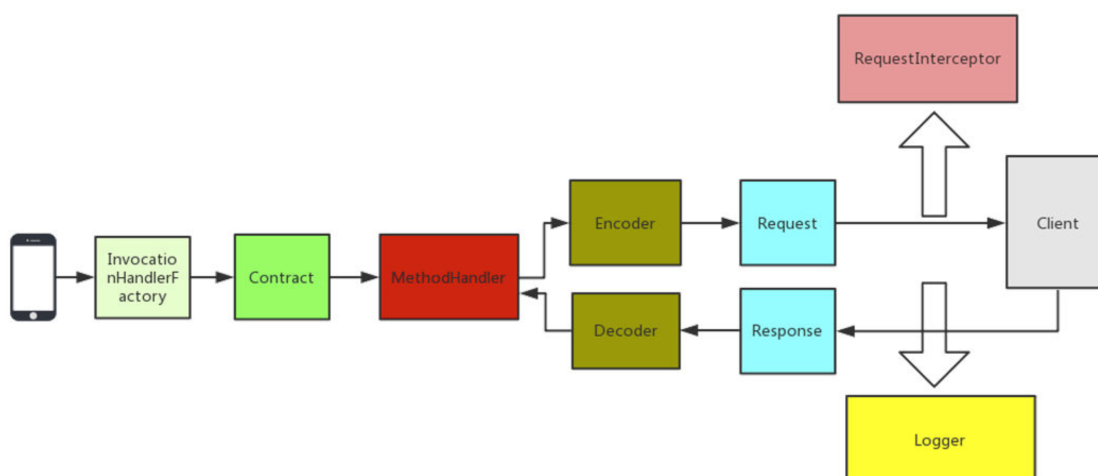
健康检查

Eureka Client 会定时发送心跳给 Eureka Server 来证明自己处于健康的状态

集成SBA以后可以把所有健康状态信息一并返回给eureka

Feign / Ribbon

- Feign 可以与 Eureka 和 Ribbon 组合使用以支持负载均衡，
- Feign 可以与 Hystrix 组合使用，支持熔断回退
- Feign 可以与ProtoBuf实现快速的RPC调用



- **InvocationHandlerFactory 代理**

  采用 JDK 的动态代理方式生成代理对象，当我们调用这个接口，实际上是要去调用远程的 HTTP API
- **Contract 契约组件**

  比如请求类型是 GET 还是 POST，请求的 URI 是什么
- **Encoder 编码组件 \ Decoder 解码组件**

  通过该组件我们可以将请求信息采用指定的编码方式进行编解码后传输
- **Logger 日志记录**

  负责 Feign 中记录日志的，可以指定 Logger 的级别以及自定义日志的输出
- **Client 请求执行组件**

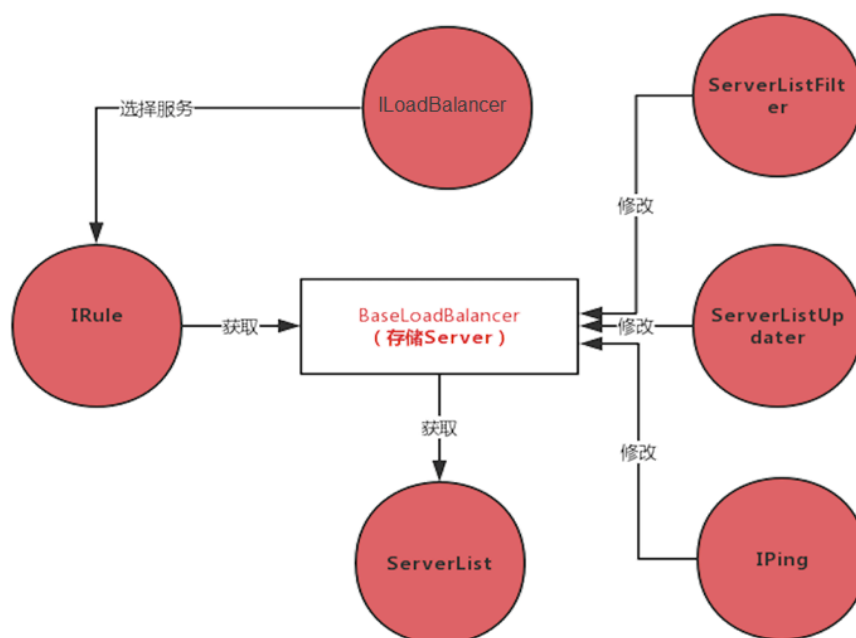
  负责 HTTP 请求执行的组件，Feign 中默认的 Client 是通过 JDK 的 HttpURLConnection 来发起请求的，在每次发送请求的时候，都会创建新的 HttpURLConnection 链接，Feign 的性能会很差，可以通过扩展该接口，使用 Apache HttpClient 等基于连接池的高性能 HTTP 客户端。

- **Retryer 重试组件**
  负责重试的组件，Feign 内置了重试器，当 HTTP 请求出现 IO 异常时，Feign 会限定一个最大重试次数来进行重试操作。
- **RequestInterceptor 请求拦截器**
  可以为 Feign 添加多个拦截器，在请求执行前设置一些扩展的参数信息。

Feign最佳使用技巧

- 继承特性
- 拦截器
  比如添加指定的请求头信息，这个可以用在服务间传递某些信息的时候。
- GET 请求多参数传递
- 日志配置
  FULL 会输出全部完整的请求信息。
- 异常解码器
  异常解码器中可以获取异常信息，而不是简单的一个code，然后转换成对应的异常对象返回。
- 源码查看是如何继承Hystrix
  HystrixFeign.builder 中可以看到继承了 Feign 的 Builder，增加了 Hystrix的SetterFactory， build 方法里，对 invocationHandlerFactory 进行了重写， create 的时候**返回HystrixInvocationHandler**， 在 invoke 的时候**会将请求包装成 HystrixCommand** 去执行，这里就自然的集成了 Hystrix

Ribbon



使用方式

- 原生 API，Ribbon 是 Netflix 开源的，没有使用 Spring Cloud，需要使用 Ribbon 的原生 API。
- Ribbon + RestTemplate，整合Spring Cloud 后，可以基于 RestTemplate 提供负载均衡的服务
- Ribbon + Feign

组件	作用
LoadBalancer	定义了一系列的操作接口，比如选择服务实例。
IRule	算法策略，内置了很多的算法策略来为服务实例的选择提供服务。
ServerList	负责服务实例信息的获取，可以获取配置文件中的，也可以从注册中心获取。
ServerListFilter	可以过滤掉某些不想要的服务实例信息。
ServerListUpdater	负责更新本地缓存的服务实例信息。
IPing	对已有的服务实例进行可用性检查，保证选择到的服务都是可用的。

负载均衡算法

- RoundRobinRule 是**轮询的算法**，A和B轮流选择。
- RandomRule 是**随机算法**，这个就比较简单了，在服务列表中随机选取。
- BestAvailableRule 选择一个最**小的并发请求 server**

自定义负载均衡算法

- 实现 Irule 接口
- 继承 AbstractLoadBalancerRule 类

自定义负载均衡使用场景（核心）

- **灰度发布**

  灰度发布是能够平滑过渡的一种发布方式，在发布过程中，先发布一部分应用，让指定的用户使用刚发布的应用，等到测试没有问题后，再将其他的全部应用发布。如果新发布的有问题，只需要将这部分恢复即可，不用恢复所有的应用。
- **多版本隔离**

  多版本隔离跟灰度发布类似，为了兼容或者过度，某些应用会有多个版本，这个时候如何保证 1.0 版本的客户端不会调用到 1.1 版本的服务，就是我们需要考虑的问题。
- **故障隔离**

  当线上某个实例发生故障后，为了不影响用户，我们一般都会先留存证据，比如：线程信息、JVM 信息等，然后将这个实例重启或直接停止。然后线下根据一些信息分析故障原因，如果我能做到故障隔离，就可以直接将出问题的实例隔离，不让正常的用户请求访问到这个出问题的实例，只让指定的用户访问，这样就可以单独用特定的用户来对这个出问题的实例进行测试、故障分析等。

Hystrix / Sentinel

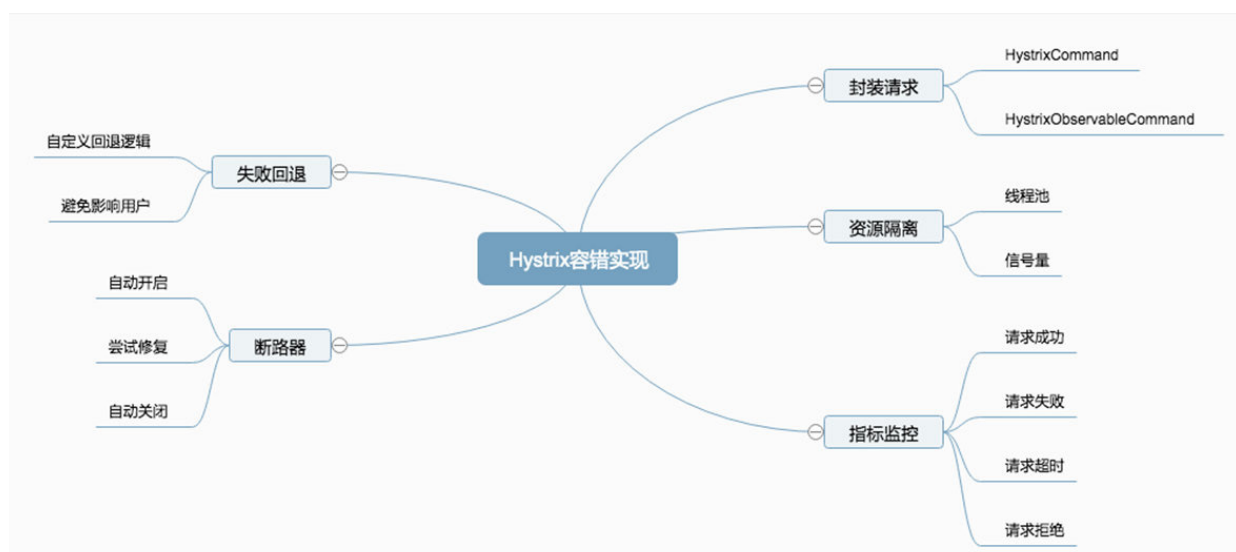
服务雪崩场景

自己即是服务消费者，同时也是服务提供者，同步调用等待结果导致资源耗尽

解决方案

服务方：扩容、限流，排查代码问题，增加硬件监控

消费方：使用Hystrix资源隔离，熔断降级，快速失败



Hystrix断路保护器的作用

- **封装请求**会将用户的操作进行统一封装，统一封装的目的在于进行统一控制。
- **资源隔离限流**会将对应的资源按照指定的类型进行隔离，比如**线程池**和**信号量**。
  - 计数器限流，例如5秒内技术1000请求，超数后限流，未超数重新计数
  - 滑动窗口限流，解决计数器不够精确的问题，把一个窗口拆分多滚动窗口
  - 令牌桶限流，类似景区售票，售票的速度是固定的，拿到令牌才能去处理请求
  - 漏桶限流，生产者消费者模型，实现了恒定速度处理请求，能够绝对防止突发流量
- **失败回退**其实是一个备用的方案，就是说当请求失败后，有没有备用方案来满足这个请求的需求。
- **断路器**这个是**最核心**的，，如果断路器处于打开的状态，那么所有请求都将失败，执行回退逻辑。如果断路器处于关闭状态，那么请求将会被正常执行。有些场景我们需要手动**打开断路器强制降级**。
- **指标监控**会对请求的**生命周期进行监控**，请求成功、失败、超时、拒绝等状态，都会被监控起来。

Hystrix使用上遇到的坑

- 配置可以对接**配置中心**进行动态调整

Hystrix 的配置项非常多，如果不对接配置中心，所有的配置只能在代码里修改，在集群部署的难以应对紧急情况，我们项目只设置一个 CommandKey，其他的都在配置中心进行指定，紧急情况如需隔离部分请求时，只需在配置中心进行修改以后，强制更新即可。

- 回退逻辑中可以**手动埋点**或者通过**输出日志**进行告警

  当请求失败或者超时，会执行回退逻辑，如果有大量的回退，则证明某些服务出问题了，这个时候我们可以在回退的逻辑中进行埋点操作，上报数据给监控系统，也可以输出回退的日志，统一由日志收集的程序去进行处理，这些方式都可以将问题暴露出去，然后通过实时数据分析进行告警操作

- 用 ThreadLocal配合**线程池隔离**模式需当心

  当我们用了线程池隔离模式的时候，被隔离的方法会包装成一个 Command 丢入到独立的线程池中进行执行，这个时候就是从 A 线程切换到了 B 线程，ThreadLocal 的数据就会丢失

- Gateway中多用信号量隔离

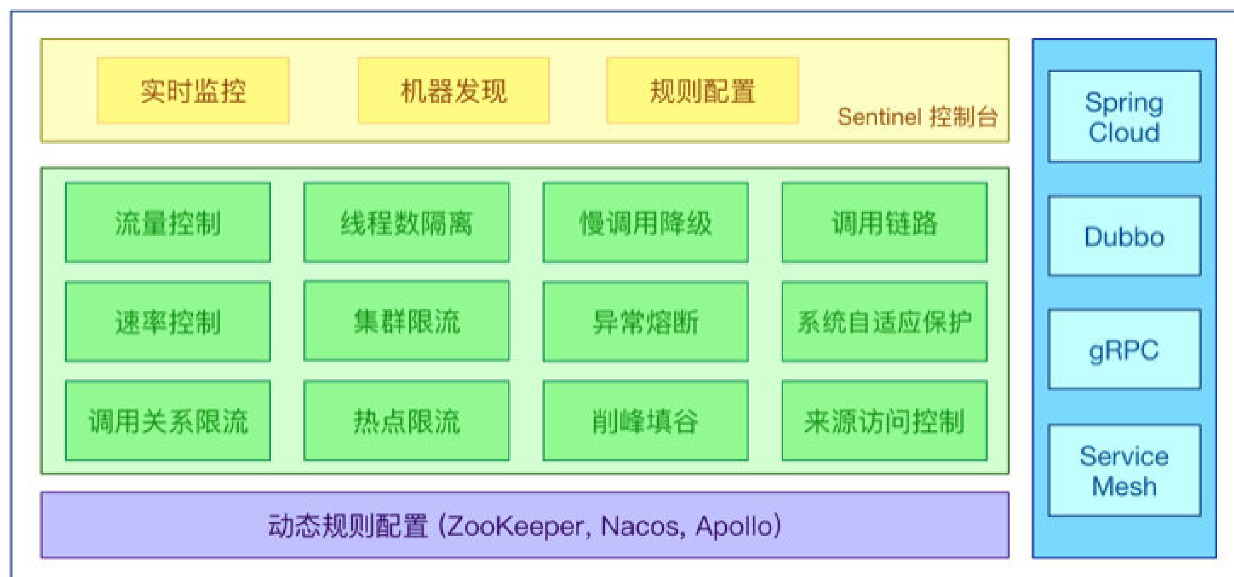
  网关是所有请求的入口，路由的服务数量会很多，几十个到上百个都有可能，如果用线程池隔离，那么需要创建上百个独立的线程池，开销太大，用信号量隔离开销就小很多，还能起到限流的作用。

Sentinel

> Sentinel是一个面向云原生微服务的流量控制、熔断降级组件。
> 替代Hystrix，针对问题：服务雪崩、服务降级、服务熔断、服务限流

Hystrix区别：

- 独立可部署Dashboard（基于 Spring Boot 开发）控制台组件
- 不依赖任何框架/库，减少代码开发，通过UI界面配置即可完成细粒度控制



丰富的应用场景：Sentinel 承接了阿里巴巴近 10 年的双十一大促流量的核心场景，例如秒杀、消息削峰填谷、集群流量控制、实时熔断下游不可用应用等。

完备的实时监控：可以看到500 台以下规模的集群的汇总也可以看到单机的秒级数据。

广泛的开源生态：与 SpringCloud、Dubbo的整合。您只需要引入相应的依赖并进行简单的配置即可快速地接入Sentinel。

区别：

- Sentinel不会像Hystrix那样放过一个请求尝试自我修复，就是明明确确按照时间窗口来，熔断触发后，时间窗口内拒绝请求，时间窗口后就恢复。
- Sentinel Dashboard中添加的规则数据存储在内存，微服务停掉规则数据就消失，在生产环境下不合适。可以将Sentinel规则数据持久化到Nacos配置中心，让微服务从Nacos获取。

#	Sentinel	Hystrix
隔离策略	信号量隔离	线程池隔离/信号量隔离
熔断降级策略	基于响应时间或失败比率	基于失败比率
实时指标实现	滑动窗口	滑动窗口（基于 RxJava）
扩展性	多个扩展点	插件的形式
限流	基于 QPS，支持基于调用关系的限流	不支持
流量整形	支持慢启动、匀速器模式	不支持
系统负载保护	支持	不支持
控制台	开箱即用，可配置规则、查看秒级监控、机器发现等	不完善
常见框架的适配	Servlet、Spring Cloud、Dubbo、gRPC	Servlet、Spring Cloud Netflix

Config / Nacos

> Nacos是阿里巴巴开源的一个针对微服务架构中服务发现、配置管理和服务管理平台。
> Nacos就是注册中心+配置中心的组合（Nacos=Eureka+Confifig+Bus）

Nacos功能特性

- 服务发现与健康检查

- 动态配置管理
- 动态DNS服务
- 服务和元数据管理

保护阈值：

当服务A健康实例数/总实例数 < 保护阈值 的时候，说明健康实例真的不多了，这个时候保护阈值会被触发（状态true），nacos将会把该服务所有的实例信息（健康的+不健康的）全部提供给消费者，消费者可能访问到不健康的实例，请求失败，但这样也比造成雪崩要好，牺牲了一些请求，保证了整个系统的一个可用。

Nacos 数据模型（领域模型）

- **Namespace** 代表不同的环境，如开发dev、测试test、生产环境prod
- **Group** 代表某项目，比如爪哇云项目
- **Service** 某个项目中具体xxx服务
- **DataId** 某个项目中具体的xxx配置文件

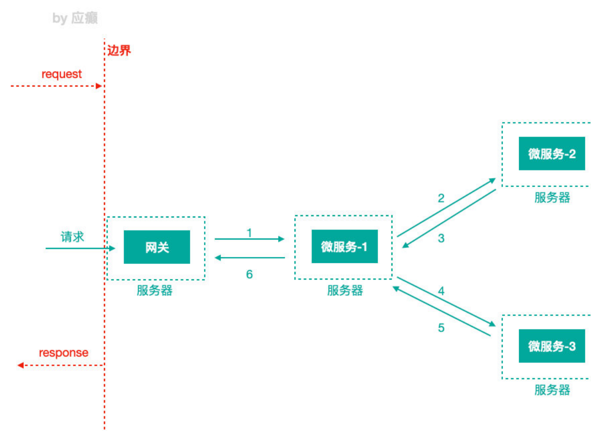
可以通过 Spring Cloud 原生注解 @RefreshScope 实现配置自动更新

Bus / Stream

> Spring Cloud Stream 消息驱动组件帮助我们更快速，更方便的去构建消息驱动微服务的
> 本质：屏蔽掉了底层不同MQ消息中间件之间的差异，统一了MQ的编程模型，降低了学习、开发、维护MQ的成本，目前支持Rabbit、Kafka两种消息

Sleuth / Zipkin

全链路追踪



Trace：服务追踪的追踪单元是从客户发起请求（request）抵达被追踪系统的边界开始到被追踪系统向客户返回响应（response）为止的过程
Trace ID：为了实现请求跟踪，当请求发送到分布式系统的入口端点时，只需要服务跟踪框架为该请求创建一个唯一的跟踪标识Trace ID，同时在分布式系统内部流转的时候，框架失踪保持该唯一标识，直到返回给请求方

Trace ID：当请求发送到分布式系统的入口端点时，Sleuth为该请求创建一个唯一的跟踪标识Trace ID，在分布式系统内部流转的时候，框架始终保持该唯一标识，直到返回给请求方

Span ID：为了统计各处理单元的时间延迟，当请求到达各个服务组件时，也是通过一个唯一标识SpanID来标记它的开始，具体过程以及结束。

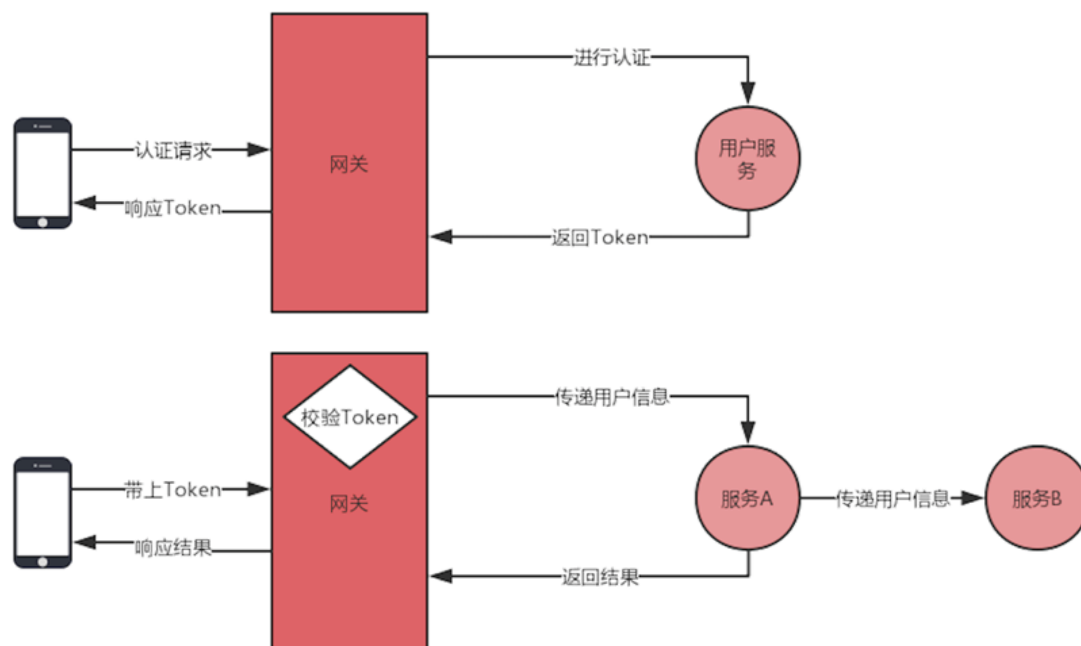
Spring Cloud Sleuth （追踪服务框架）可以追踪服务之间的调用，Sleuth可以记录一个服务请求经过哪些服务、服务处理时长等，根据这些，我们能够理清各微服务间的调用关系及进行问题追踪分析。

耗时分析：通过 Sleuth 了解采样请求的耗时，分析服务性能问题（哪些服务调用比较耗时）
链路优化：发现频繁调用的服务，针对性优化等
聚合展示：数据信息发送给 Zipkin 进行聚合，利用 Zipkin 存储并展示数据。

安全认证

- Session
  认证中最常用的一种方式，也是最简单的。存在多节点session丢失的情况，可通过nginx粘性Cookie和Redis集中式Session存储解决
- HTTP Basic Authentication
  服务端针对请求头中base64加密的Authorization 和用户名和密码进行校验。
- Token
  Session 只是一个 key，会话信息存储在后端。而 Token 中会存储用户的信息，然后通过加密算法进行加密，只有服务端才能解密，服务端拿到 Token 后进行解密获取用户信息。
- JWT认证

> JWT（JSON Web Token）用户提供用户名和密码给认证服务器，服务器验证用户提交信息的合法性；如果验证成功，会产生并返回一个 Token，用户可以使用这个 Token 访问服务器上受保护的资源。



1. 认证服务提供认证的 API，校验用户信息，返回认证结果
2. 通过JWTUtils中的RSA算法，生成JWT token，token里封装用户id和有效期
3. 服务间参数通过请求头进行传递，服务内部通过 ThreadLocal 进行上下文传递。
4. Hystrix导致ThreadLocal失效的问题可以通过，重写 Hystrix 的 Callable 方法，传递需要的数据。

Token最佳实践

- 设置**较短（合理）的过期时间。**
- 注销的 Token **及时清除**（放入 Redis 中做一层过滤）。
  虽然不能修改 Token 的信息，但是能在验证层面做一层过滤来进行处理。
- 监控 Token 的**使用频率。**
  为了防止数据被别人爬取，最常见的就是监控使用频率，程序写出来的爬虫程序访问频率是有迹可循的
- 核心功能敏感操作可以使用**动态验证**（验证码）。
  比如提现的功能，要求在提现时再次进行验证码的验证，防止不是本人操作。
- **网络环境、浏览器**信息等识别。
  银行 APP 对环境有很高的要求，使用时如果断网，APP 会自动退出，重新登录，因为网络环境跟之前使用的不一样了，还有一些浏览器的信息之类的判断，这些都是可以用来保证后端 API 的安全。
- **加密密钥**支持动态修改。
  如果 Token 的加密密钥泄露了，也就意味着别人可以伪造你的 Token，可以将密钥存储在配置中心，以支持动态修改刷新，需要注意的是建议在流量低峰的时候去做更换的操作，否则 Token 全部失效，所有在线的请求都会重新申请 Token，并发量会比较大。

灰度发布

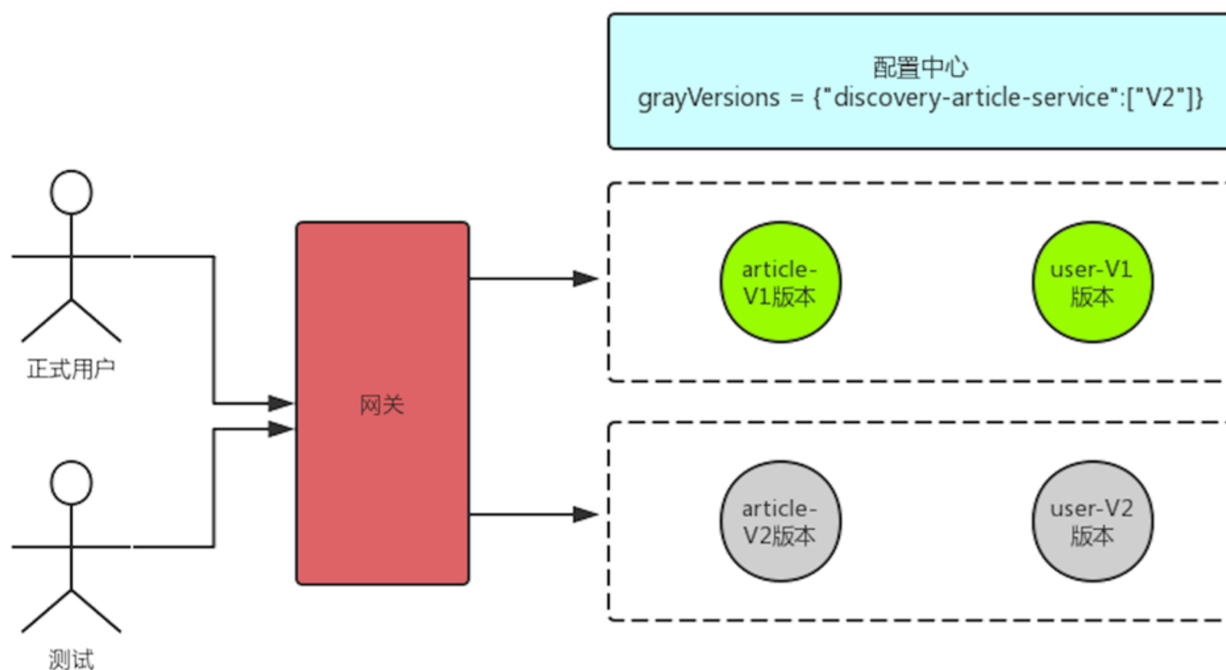
痛点：

- 服务数量多，业务变动频繁，一周一发布
- 灰度发布能降低发布失败风险，减少影响范围
  通过灰度发布，先让一部分用户体验新的服务，或者只让测试人员进行测试，等功能正常后再全部发布，这样能降低发布失败带来的影响范围。
- 当发布出现故障时，可以快速回滚，不影响用户
  灰度后如果发现这个节点有问题，那么只需回滚这个节点即可，当然不回滚也没关系，通过灰度策略隔离，也不会影响正常用户

可以通过Ribbon的负载均衡策略进行灰度发布，可以使用更可靠的Discovery

Discovery

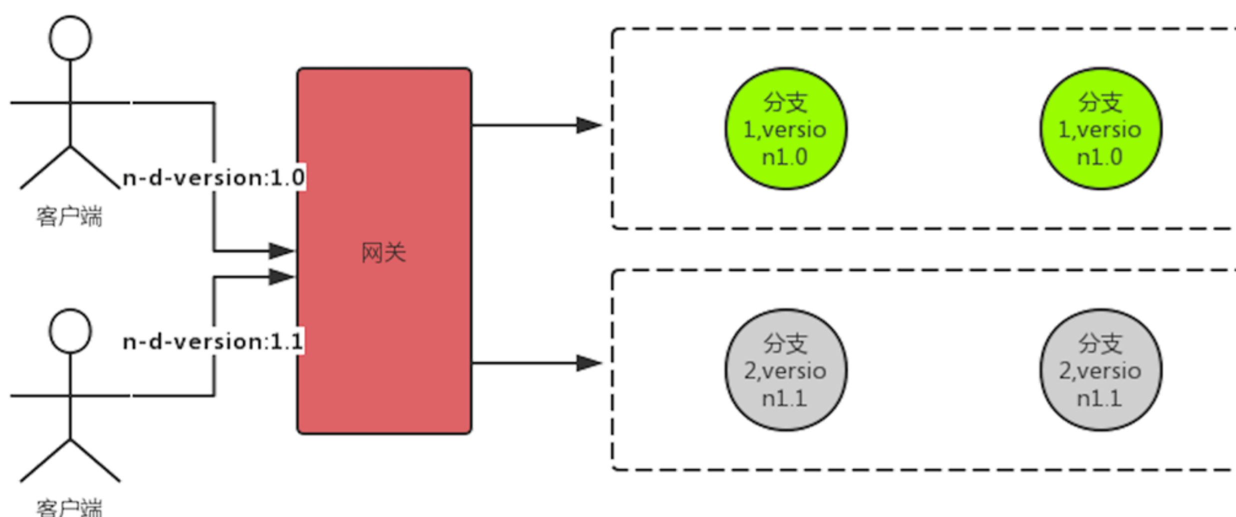
> 基于Discovery 服务注册发现、Ribbon 负载均衡、Feign 和 RestTemplate 调用等组件的企业级微服务开源解决方案，包括灰度发布、灰度路由、服务隔离等功能



1. 首先将需要发布的服务从转发过程中移除，等流量剔除之后再发布。
2. 部分机器中的版本进行升级，用户默认还是请求老的服务，通过版本来支持测试请求。
3. 测试完成之后，让新的版本接收正常流量，然后部署下一个节点，以此类推。

```
grayVersions = {"discovery-article-service":["1.01"]}
```

多版本隔离



本地复用测试服务–Eureka Zone亮点

region 地理上的分区，比如北京、上海等

zone 可以简单理解为 region 内的具体机房

在调用的过程中会优先选择相同的 zone 发起调用，当找不到相同名称的 zone 时会选择其他的 zone 进行调用，我们可以利用这个特性来解决本地需要启动多个服务的问题。

各组件调优

当你对网关进行压测时，会发现并发量一直上不去，错误率也很高。因为你用的是默认配置，这个时候我们就需要去调整配置以达到最优的效果。

首先我们可以对容器进行调优，最常见的就是**内置的Tomcat** 容器了，

```
server.tomcat.accept-count //请求队列排队数
server.tomcat.max-threads //最大线程数
server.tomcat.max-connections //最大连接数
```

Hystrix 的信号量（semaphore）隔离模式，并发量上不去很大的原因都在这里，信号量默认值是 100，也就是最大并发只有 100，超过 100 就得等待。

```
//信号量
zuul.semaphore.max-semaphores //信号量：最大并发数
//线程池
hystrix.threadpool.default.coreSize //最大线程数
hystrix.threadpool.default.maximumSize //队列的大
hystrix.threadpool.default.maxQueueSize //等参数
```

配置Gateway并发信息，

```
gateway.host.max-per-route-connections //每个路由的连接数
gateway.host.max-total-connections //总连接数
```

调整 **Ribbon** 的并发配置，

```
ribbon.MaxConnectionsPerHost //单服务并发数
ribbon.MaxTotalConnections   //总并发数
```

修改**Feign**默认的HttpURLConnection 替换成 httpclient 来提高性能

```
feign.httpclient.max-connections-per-route//每个路由的连接数
feign.httpclient.max-connections //总连接数
```

Gateway+配置中心实现动态路由

Feign+配置中心实现动态日志

分布式篇

> 分布式系统是一个硬件或软件组件分布在不同的网络计算机上，彼此之间仅仅通过消息传递进行通信和协调的系统。

发展历程

- 入口级负载均衡
  - 网关负载均衡
  - 客户端负载均衡
- 单应用架构
  - 应用服务和数据服务分离
  - 应用服务集群
  - 应用服务中心化SAAS
- 数据库主备读写分离
  - 全文搜索引擎加快数据统计
  - 缓存集群缓解数据库读压力
  - 分布式消息中间件缓解数据库写压力
  - 数据库水平拆分适应微服务
  - 数据库垂直拆分解决慢查询
- 划分上下文拆分微服务
  - 服务注册发现（Eureka、Nacos）
  - 配置动态更新（Config、Apollo）
  - 业务灰度发布（Gateway、Feign）
  - 统一安全认证（Gateway、Auth）
  - 服务降级限流（Hystrix、Sentinel）

  - 接口检查监控（Actuator、Prometheus）
  - 服务全链路追踪（Sleuth、Zipkin）

CAP

- **一致性**（2PC、3PC、Paxos、Raft）
  - 强一致性：**数据库一致性**，牺牲了性能
    - **ACID**：原子性、一致性、隔离性、持久性
  - 弱一致性：**数据库和缓存，延迟双删、重试**
  - 单调读一致性：**缓存一致性**，ID或者IP哈希
  - 最终一致性：**边缘业务**，消息队列
- **可用性**（多级缓存、读写分离）
  - **BASE** 基本可用：限流导致响应速度慢、降级导致用户体验差
    - Basically Availabe 基本可用
    - Soft state 软状态
    - Eventual Consistency 最终一致性
- 分区容忍性（一致性Hash解决扩缩容问题）

一致性

XA方案

2PC协议：两阶段提交协议，P是指准备阶段，C是指提交阶段
- 准备阶段：询问是否可以开始，写Undo、Redo日志，收到响应
- 提交阶段：执行Redo日志进行**Commit**，执行Undo日志进行**Rollback**

3PC协议：将提交阶段分为**CanCommit、PreCommit、DoCommit**三个阶段
CanCommit：发送canCommit请求，并开始等待
PreCommit：收到全部Yes，写Undo、Redo日志。超时或者No，则中断
DoCommit：执行Redo日志进行Commit，执行Undo日志进行Rollback
区别是第二步，参与者**自身增加了超时，如果失败可以及时释放资源**

Paxos算法

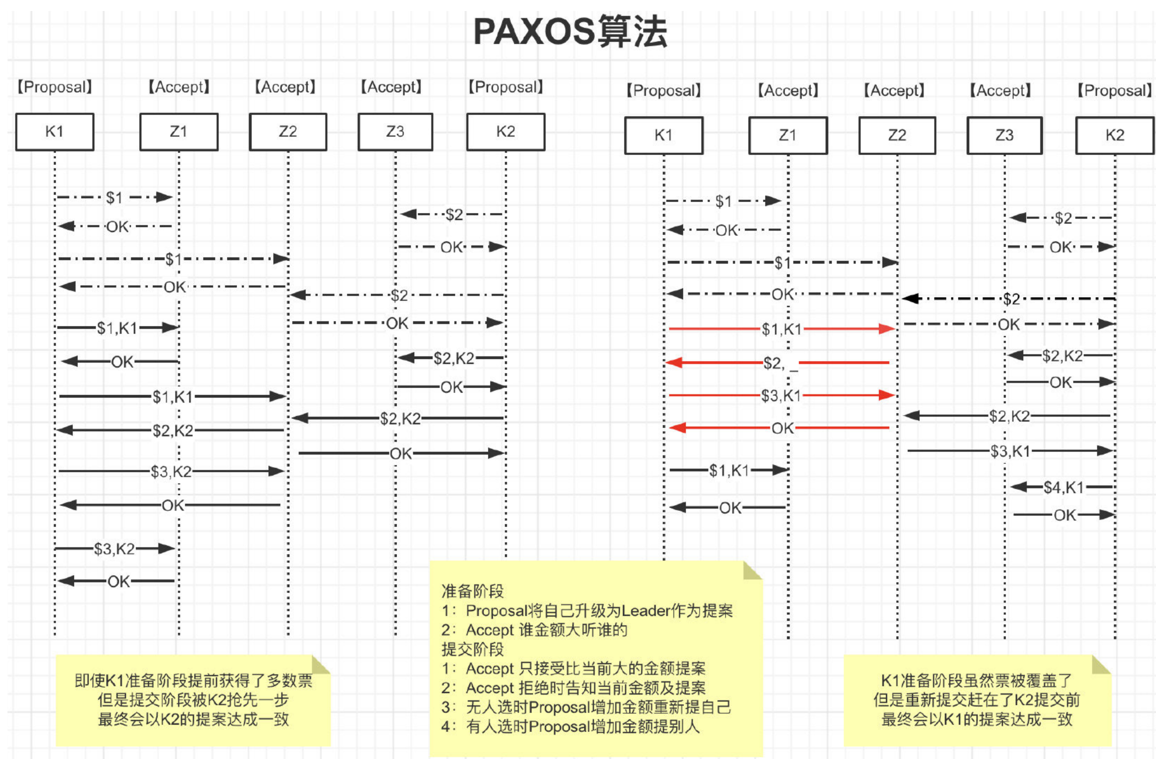
> 分布式系统是一个硬件或软件组件分布在不同的网络计算机上，彼此之间仅仅通过消息传递进行通信和协调的系统。

参与者（例如Kafka）的一致性可以由协调者（例如Zookeeper）来保证，协调者的一致性就只能由Paxos保证了

Paxos算法中的角色：

- **Client：**客户端、例如，对分布式文件服务器中文件的写请求。
- **Proposer：**提案发起者，根据Accept返回选择最大N对应的V，发送[N+1,V]
- **Acceptor：**决策者，Accept以后会拒绝小于N的提案，并把自己的[N,V]返回给Proposer
- **Learners：**最终决策的学习者、学习者充当该协议的复制因素

```
//算法约束
P1:一个Acceptor必须接受它收到的第一个提案。
//考虑到半数以上才作数，一个Accpter得接受多个相同v的提案
P2a:如果某个v的提案被accept，那么被Acceptor接受编号更高的提案必须也是v
P2b:如果某个v的提案被accept，那么从Proposal提出编号更高的提案必须也是v
//如何确保v的提案Accpter被选定后，Proposal都能提出编号更高的提案呢
针对任意的[Mid,Vid]，有半数以上的Accepter集合S，满足以下二选一：
  S中接受的提案都大于Mid
  S中接受的提案若小于Mid，编号最大的那个值为Vid
```



面试题：如何保证Paxos算法活性

假设存在这样一种极端情况，有两个Proposer依次提出了一系列编号递增的提案，导致最终陷入死循环，没有value被选定

- **通过选取主Proposer**，规定只有主Proposer才能提出议案。只要主Proposer和过半的Acceptor能够正常网络通信，主Proposer提出一个编号更高的提案，该提案终将会被批准。
- 每个Proposer发送提交提案的时间设置为**一段时间内随机**，保证不会一直死循环

ZAB算法

Raft算法

> Raft 是一种为了管理复制日志的一致性算法

Raft使用**心跳机制**来触发选举。当server启动时，初始状态都是**follower**。每一个server都有一个定时器，超时时间为election timeout（**一般为150-300ms**），如果某server**没有超时的情况下收到**来自领导者或者候选者的任何消息，**定时器重启**，如果超时，它就开始一次选举。

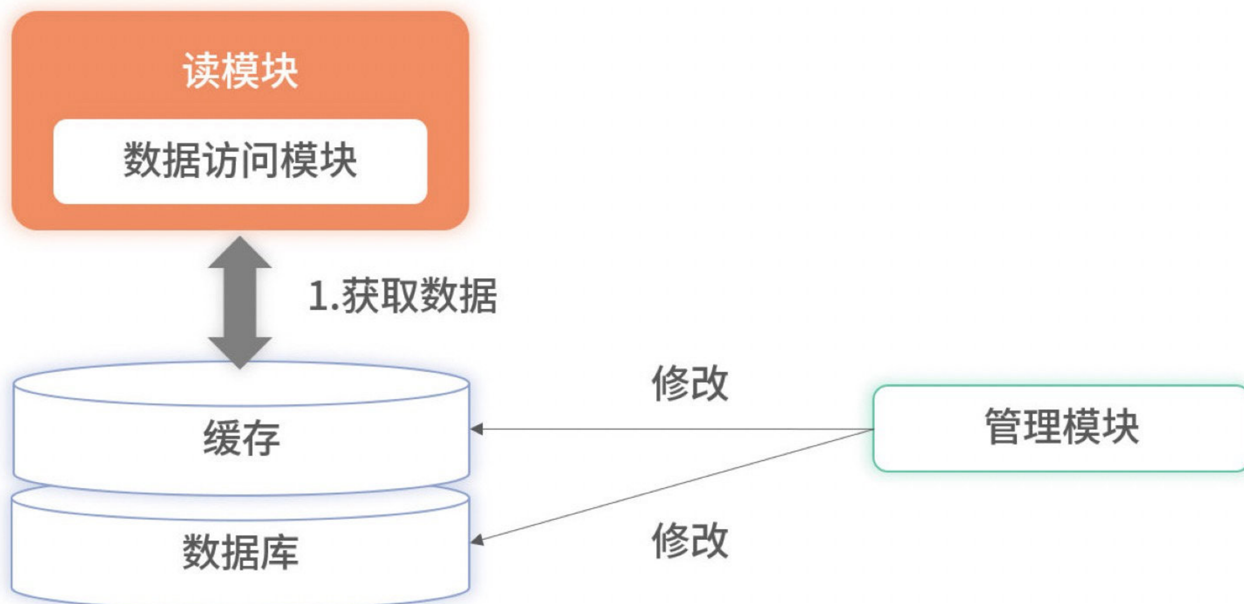
Leader异常：异常期间Follower会超时选举，完成后Leader比较彼此步长

Follower异常：恢复后直接同步至Leader当前状态

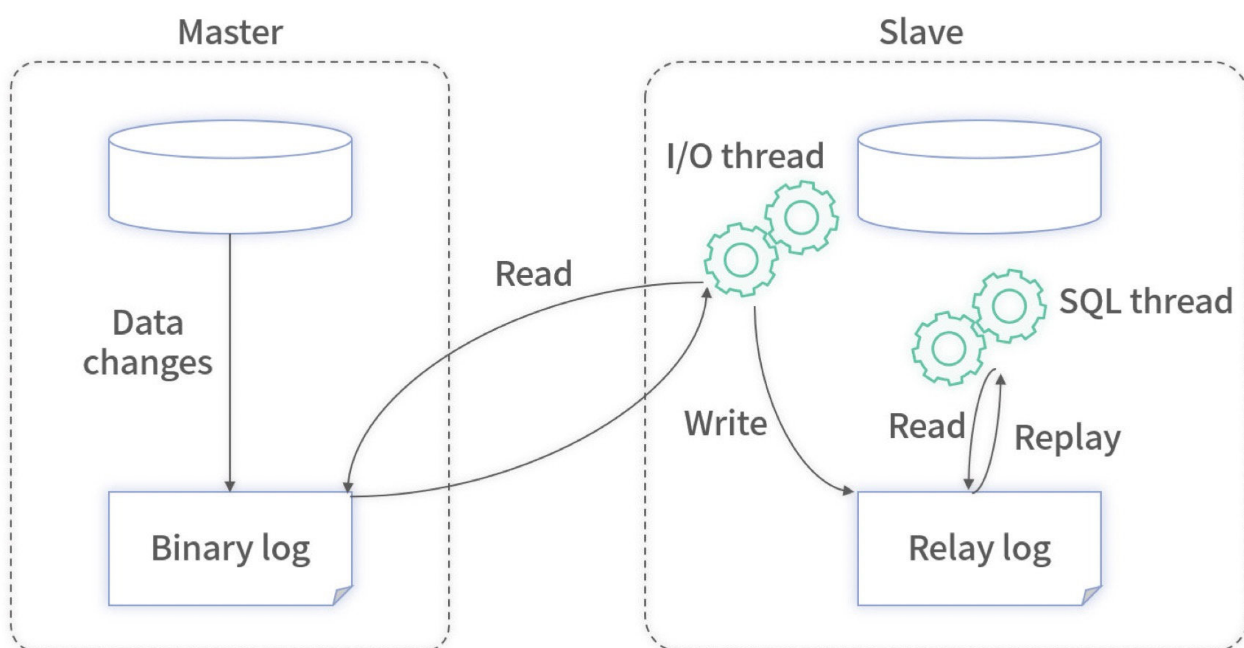
多个Candidate：选举时失败，失败后超时继续选举

数据库和Redis的一致性

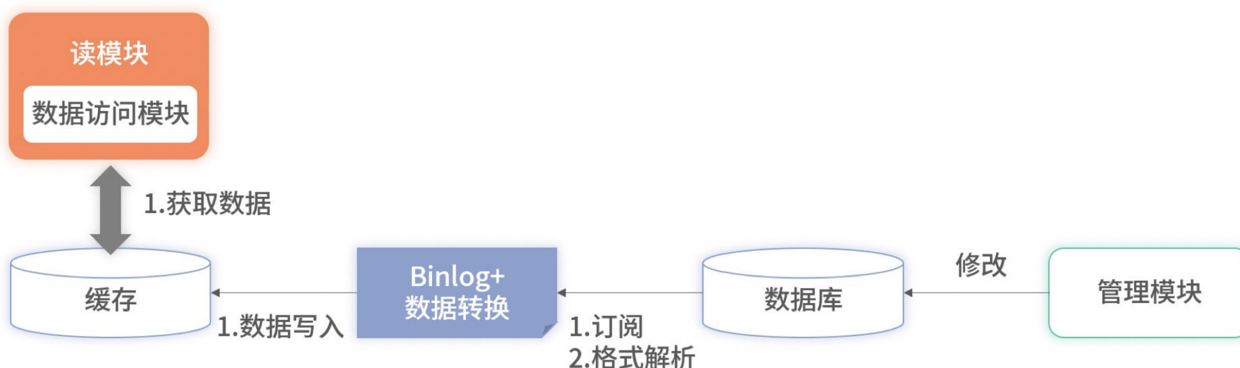
全量缓存保证高效读取



所有数据都存储在缓存里，读服务在查询时不会再降级到数据库里，所有的请求都完全依赖缓存。此时，因降级到数据库导致的毛刺问题就解决了。但全量缓存并**没有解决更新时的分布式事务**问题，反而把问题放大了。因为全量缓存**对数据更新要求更加严格**，要求所有数据库**已有数据和实时更新**的数据必须完全同步至缓存，不能有遗漏。对于此问题，一种有效的方案是采用**订阅数据库的 Binlog** 实现数据同步



现在很多开源工具（如**阿里的 Canal**等）可以模拟主从复制的协议。通过模拟协议读取主数据库的 Binlog 文件，从而获取主库的所有变更。对于这些变更，它们开放了各种接口供业务服务获取数据。

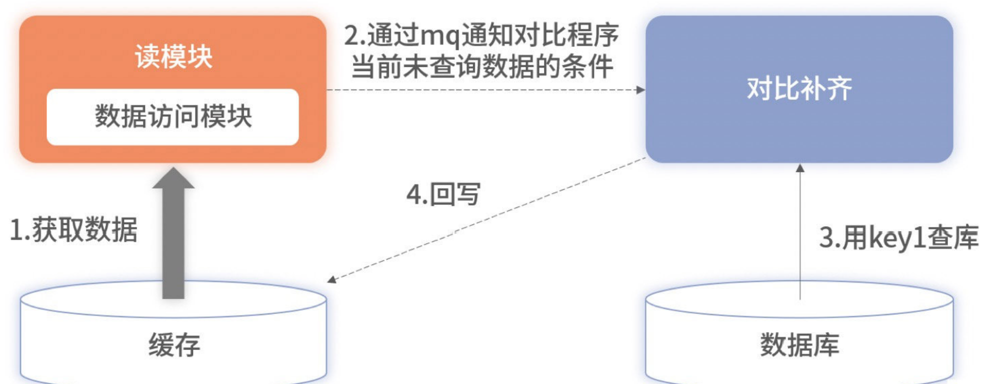


将 Binlog 的中间件挂载至目标数据库上，就可以**实时获取该数据库的所有变更数据**。对这些变更数据解析后，便可**直接写入缓存里**。优点还有：

- 大幅提升了读取的速度，降低了延迟
- Binlog 的主从复制是基于 ACK 机制， 解决了分布式事务的问题

  如果同步缓存失败了，被消费的 Binlog 不会被确认，下一次会重复消费，数据最终会写入缓存中

缺点不可避免：1、增加复杂度 2、消耗缓存资源 3、需要筛选和压缩数据 4、极端情况数据丢失



可以通过异步校准方案进行补齐，但是会损耗数据库性能。但是此方案会隐藏中间件使用错误的细节，线上环境前期更重要的是记录日志排查在做后续优化，不能本末倒置。

可用性

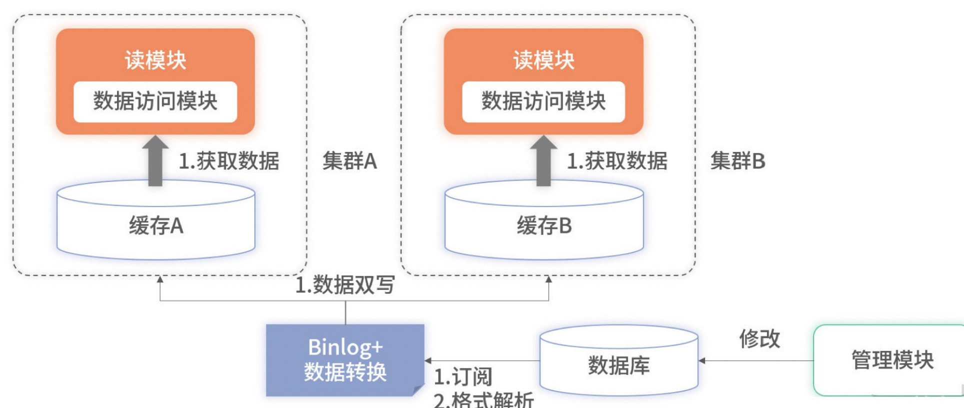
心跳检测

> 以固定的频率向其他节点汇报当前节点状态的方式。收到心跳，说明网络和节点的状态是健康的。心跳汇报时，一般会携带一些附加的状态、元数据，以便管理

周期检测心跳机制：超时未返回

累计失效检测机制：重试超次数

多机房实时热备



两套缓存集群可以分别部署到不同城市的机房。读服务也相应地部署到不同城市或不同分区。在承接请求时，不同机房或分区的读服务只依赖同样属性的缓存集群。此方案有两个好处。

1. 提升了性能。读服务不要分层，读服务要尽可能地和缓存数据源靠近。
2. 增加了可用。当单机房出现故障时，可以秒级将所有流量都切换至存活的机房或分区

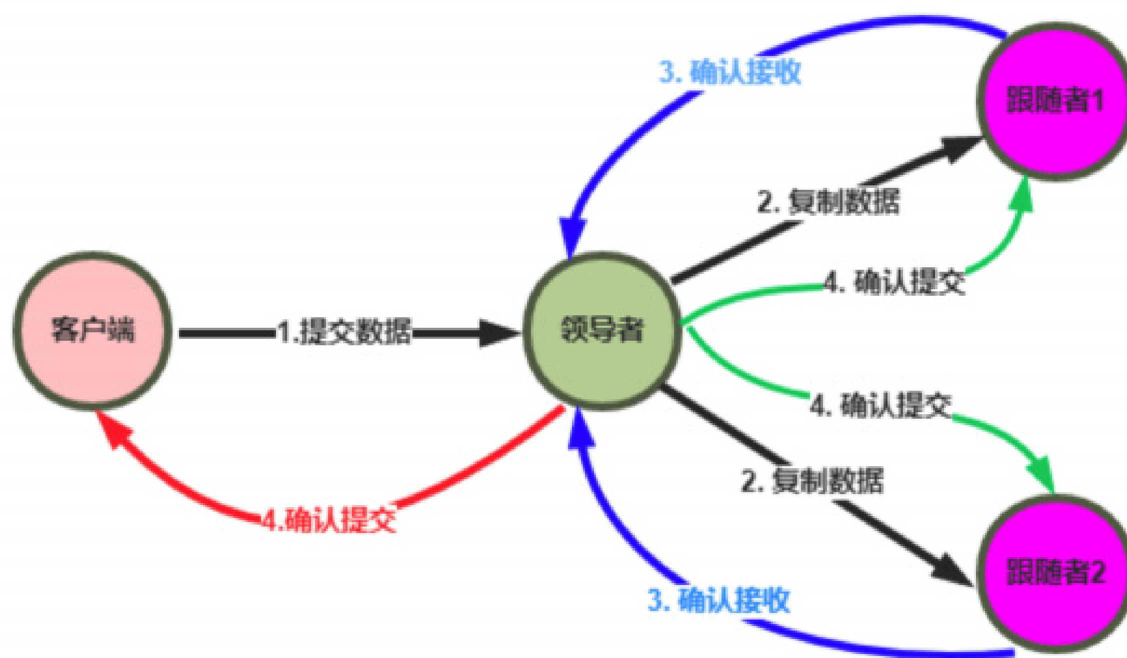
此方案虽然带来了性能和可用性的提升，但代价是资源成本的上升。

分区容错性

> 分布式系统对于错误包容的能力

通过限流、降级、兜底、重试、负载均衡等方式增强系统的健壮性

日志复制



1. Leader把指令添加到日志中，发起 RPC 给其他的服务器，让他们复制这条信息
2. Leader会不断的重试，直到所有的 Follower响应了ACK并复制了所有的日志条目
3. 通知所有的Follower提交，同时Leader该表这条日志的状态，并返回给客户端v

主备（Master-Slave）

主机宕机时，备机接管主机的一切工作，主机恢复正常后，以自动（热备）或手动（冷备）方式将服务切换到主机上运行，Mysql和Redis中常用。

MySQL之间数据复制的基础是**二进制日志文件**（binary log fifile）。它的数据库中所有操作都会以“事件”的方式记录在二进制日志中，其他数据库作为slave通过一个**I/O线程与主服务器保持通信，并监控**master的二进制日志文件的变化，如果发现master二进制日志文件发生变化，则会把变化复制到自己的**中继日志**中，然后slave的一个SQL线程会把相关的“事件”执行到自己的数据库中，以此实现从数据库和主数据库的**一致性**，也就实现了**主从复制**

互备（Active-Active）

指两台主机同时运行各自的服务工作且**相互监测**情况。在数据库高可用部分，常见的互备是MM模式。MM模式即Multi-Master模式，指一个系统存在多个master，每个master都具有read-write能力，会根据**时间戳**或**业务逻辑**合并版本。

集群（Cluster）模式

是指有多个节点在运行，同时可以通过主控节点**分担服务**请求。如Zookeeper。集群模式需要解决主控节点**本身的高可用**问题，一般采用主备模式。

分布式事务

XA方案

两阶段提交 | 三阶段提交

- 准备阶段的资源锁定，存在性能问题，严重时会造成死锁问题
- 提交事务请求后，出现网络异常，部分数据收到并执行，会造成一致性问

TCC方案

Try Confirm Cancel / 短事务

- Try 阶段：这个阶段说的是对各个服务的资源做检测以及对资源进行锁定或者预留
- Confirm 阶段：这个阶段说的是在各个服务中执行实际的操作
- Cancel 阶段：如果任何一个服务的业务方法执行出错，那么就需要进行补偿/回滚

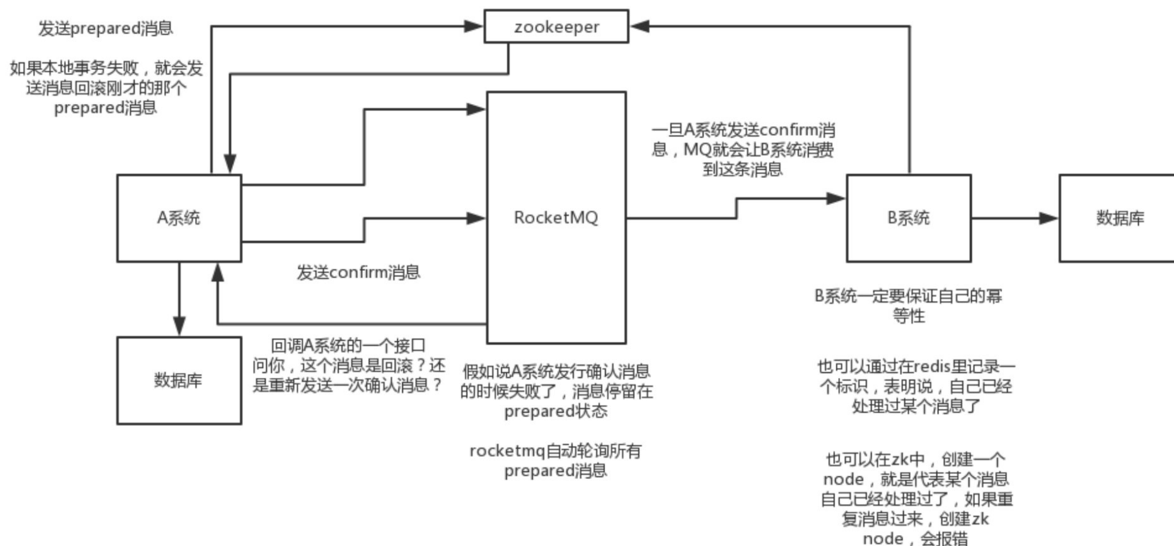
Saga方案

事务性补偿 / 长事务

- 流程长、流程多、调用第三方业务

本地消息表（eBay）

MQ最终一致性



比如阿里的 RocketMQ 就支持消息事务（核心：双端确认，重试幂等）

1. A(订单) 系统先发送一个 prepared 消息到 mq，prepared 消息发送失败则取消操作不执行了
2. 发送成功后，那么执行本地事务，执行成功和和失败发送确认和回滚消息到mq
3. 如果发送了确认消息，那么此时 B(仓储) 系统会接收到确认消息，然后执行本地的事务
4. mq 会自动定时轮询所有 prepared 消息回调的接口，确认事务执行状态
5. B 的事务失败后自动不断重试直到成功，达到一定次数后发送报警由人工来手工回滚和补偿

最大努力通知方案（订单 -> 积分）

1. 系统 A 本地事务执行完之后，发送个消息到 MQ；
2. 这里会有个专门消费 MQ 的最大努力通知服务，接着调用系统 B 的接口；
3. 要是系统 B 执行失败了，就定时尝试重新调用系统 B，反复 N 次，最后还是不行就放弃

你找一个严格资金要求绝对不能错的场景，你可以说你是用的 TCC 方案；

如果是一般的分布式事务场景，例如积分数据，可以用可靠消息最终一致性方案

如果分布式场景允许不一致，可以使用最大努力通知方案

面试题

分布式Session实现方案

- 基于JWT的Token，数据从cache或者数据库中获取
- 基于Tomcat的Redis，简单配置conf文件
- 基于Spring的Redis，支持SpringCloud和Springb

第二部分
关于DESIGN的方方面面

- ES篇
- Docker&K8S篇
- Netty篇
- LEETCODE
- 实战算法篇
- 个人项目
- 架构设计
- 领域模型落地

ES篇

因集群架构变动导致的消费组内重平衡，如果kafka集内节点较多，比如数百个，那重平衡可能会耗时导致数分钟到数小时，此时kafka基本处于不可用状态，对kafka的TPS影响极大

概述

特点

1. **安装方便：**没有其他依赖，下载后安装非常方便；只用修改几个参数就可以搭建起来一个集群
2. **JSON：**输入/输出格式为 JSON，意味着不需要定义 Schema，快捷方便
3. **RESTful：**基本所有操作（索引、查询、甚至是配置）都可以通过 HTTP 接口进行
4. **分布式：**节点对外表现对等（每个节点都可以用来做入口） 加入节点自动负载均衡
5. **多租户：**可根据不同的用途分索引，可以同时操作多个索引
6. **支持超大数据：** 可以扩展到 PB 级的结构化和非结构化数据 海量数据的近实时处理

功能

- **分布式的搜索引擎**
  分布式：Elasticsearch自动将海量数据分散到多台服务器上去存储和检索
- **全文检索**
  提供模糊搜索等自动度很高的查询方式，并进行相关性排名，高亮等功能
- **数据分析引擎（分组聚合）**
  社区网站，最近一周用户登录、最近一个月各功能使用情况
- **对海量数据进行近实时（秒级）的处理**
  海量数据的处理：因为是分布式架构，可以采用大量的服务器去存储和检索数据

场景

- **搜索类**场景
  比如说人员检索、设备检索、App内的搜索、订单搜索。
- **日志分析**类场景
  经典的ELK组合（Elasticsearch/Logstash/Kibana），实现**日志收集，日志存储，日志分析**
- **数据预警平台**及数据分析场景
  例如社区团购提示，当优惠的价格低于某个值时，自动触发通知消息，通知用户购买。
  分析竞争对手商品销量Top10，供运营分析等等。
- **商业BI(Business Intelligence)**系统
  比如社区周边，需要分析某一地区用户消费金额及商品类别，输出相应的报表数据，并预测该地区的热卖商品，通过区域和人群特征划分进行定向推荐。Elasticsearch执行数据分析和挖掘，Kibana做数据可视化。

竞品分析

Lucene
Java编写的信息搜索工具包（Jar包），Lucene只是一个框架，熟练运用Lucene非常复杂。

Solr
基于**Lucene**的HTTP接口查询服务器，是一个封装了很多Lucene细节搜索引擎系统

Elasticsearch
基于**Lucene**分布式海量数据近实时搜索引擎。采用的策略是将每一个字段都编入索引，使其可以被搜索。

对比

1）Solr利用Zookeeper进行分布式管理，而Elasticsearch自身带有分布式协调管理功能
2）Solr比Elasticsearch实现更加全面，而Elasticsearch本身更注重于核心功能， 高级功能多由第三方插件提供
3）Solr在传统的搜索应用中表现好于Elasticsearch，而Elasticsearch在实时搜索应用方面比Solr表现好
目前主流依然是**Elasticsearch**7.x 最新的是7.8
优化：**默认集成JDK**、升级Lucene8大幅提升**TopK性能**、引入熔断机制**避免OOM**发生

基本概念

IK分词器

IKAnalyzer是一个开源的，基于java语言开发的轻量级的中文分词工具包。新版本的IKAnalyzer3.0则发展为 面向Java的公用分词组件，独立于Lucene项目，同时提供了对Lucene的默认优化实现。

IK分词器3.0的特性如下：

1. 采用了特有的“正向迭代最细粒度切分算法“，具有60万字/秒的高速处理能力。
2. 采用了多子处理器分析模式，支持：英文字母（IP地址、Email、URL）、数字（日期，常用中文数量词，罗马数字，科学计数法），中文词汇（姓名、地名处理）等分词处理。
3. 支持个人词条的优化的词典存储，更小的内存占用。
4. 针对Lucene全文检索优化的查询分析器IKQueryParser；采用歧义分析算法优化查询关键字的搜索
5. 排列组合，能极大的提高Lucene检索的命中率。

- **扩展词典：ext_dict**
- **停用词典：stop_dict**
- **同义词典：same_dict**

索引（类数据库）

settings：设置索引库，定义索引库的分片数副本数等

映射（类表设计）

- 字段的数据类型
- 分词器类型
- 是否要进行存储或者创造索引

文档（数据）

- 全量更新用Put
- 局部更新用Post

高级特性

映射高级

地理坐标点数据类型

> 地理坐标点是指地球表面可以用经纬度描述的一个点。 地理坐标点可以用来计算两个坐标间的距离，还可以判断一个坐标是否在一个区域中。地理坐标点需要显式声明对应字段类型为 geo_point

动态映射

> 使用dynamic mapping 来确定字段的数据类型并自动把新的字段添加到类型映射

DSL高级

- **查询所有(match_all query)**
- **全文搜索(full-text query)**
  - 匹配搜索(match query)
  - 短语搜索(match phrase query)
  - 默认查询(query string)
  - 多字段匹配搜索(multi match query)
- **词条级搜索(term-level query)**
  - 精确搜索term
  - 集合搜索idx
  - 范围搜索range
  - 前缀搜索prefix
  - 通配符搜索wildcard
  - 正则搜索regexp
  - 模糊搜索fuzzy
- **复合搜索**
- **排序sort&分页size&高亮highLight&批量bluk**

聚合分析

> 聚合分析是数据库中重要的功能特性，完成对一个查询的数据集中数据的聚合计算，如：找出某字段（或计算表达式的结果）的最大值、最小值，计算和、平均值等

- 对一个数据集求最大、最小、和、平均值等指标的聚合，在ES中称为指标聚合 metric
- 对查询出的数据进行分桶group by，再在桶上进行指标桶聚合 bucketing

智能搜索

- Term Suggester
- Phrase Suggester
- Completion Suggester
- Context Suggester

如果Completion Suggester已经到了零匹配，可以猜测用户有输入错误，这时候可以尝试一下Phrase Suggester。如果还是未匹配则尝试Term Suggester。

精准程度上(Precision)看： Completion > Phrase > Term， 而召回率上(Recall)则反之。

从性能上看，Completion Suggester是最快的，如果能满足业务需求，只用Completion Suggester做前缀匹配是最理想的。 Phrase和Term由于是做倒排索引的搜索，相比较而言性能应该要低不少，应尽量控制Suggester用到的索引的数据量，最理想的状况是经过一定时间预热后，索引可以全量map到内存。

实战

写优化

- **副本数量0**
  首次 初始化数据时，将副本设置为0，写入完毕再改回，避免了副本建立索引的过程
- **自动生成id**
  可以避免写前判断是否存在的过程
- **合理使用分词器**
  binary类型不适用，title和text使用不同的分词器加快速度
- **禁用评分，延长索引刷新间隔**
- **将多个索引操作放入到batch进行处理**

读优化

- **使用Filter代替Query，减少打分缓解，使用bool组合query和filter查询**
- **对数据进行分组，按照日，月，年不同维度分组，查询可集中在局部index中**

零停机索引重建方案

- **外部数据导入**
  - 通过MQ的web控制台或cli命令行，发送指定的MQ消息
  - MQ消息被微服务模块的消费者消费，触发ES数据重新导入功能
  - 微服务模块从数据库里查询数据的总数及分页信息，并发送至MQ
  - 微服务从MQ中根据分页信息从数据库获取到数据后，根据索引结构的定义，将数据组装成ES支持的JSON格式，并通过bulk命令将数据发送给Elasticsearch集群进行索引的重建工作。

- **基于Scroll+bulk+索引别名的方案**
  - 新建索引book_new，将mapping信息，settings信息等按新的要求全部定义好
  - 使用scroll api将数据批量查询出来，指定scroll查询持续时间
  - 采用bulk api将scoll查出来的一批数据，批量写入新索引
  - 查询一批导入一批，注意每次都使用上次结束时的scoll_id
  - 切换别名bookalias到新的索引booknew上面，此时Java客户端仍然使用别名访问，也不需要修

  改任何代码，不需要停机。验证别名查询的是否为新索引的数据

- **Reindex API方案**
  - Elasticsearch v6.3.1已经支持Reindex API，它对scroll、bulk做了一层封装，能够 对文档重建索引而不需要任何插件或外部工具。

参与度 & 灵活性：自研 > scroll+bulk > reindex

稳定性 & 可靠性：自研 < scroll+bulk < reindex

DeepPaging性能解决方案

> 比如超级管理员，要给某个省份用户发送公告或者广告，最容易想到的就是利用 from + size 来实现，但这是不现实的

分页方式	性能	优点	缺点	场景
from + size	低	灵活性好，实现简单	深度分页问题	数据量比较小，能容忍深度分页问题
scroll	中	解决了深度分页问题	无法反映数据的实时性（快照版本）维护成本高，需要维护一个 scroll_id	海量数据的导出 需要查询海量结果集的数据
search_after	高	性能最好 不存在深度分页问题 能够反映数据的实时变更	实现连续分页的实现会比较复杂，因为每一次查询都需要上次查询的结果	海量数据的分页

Docker&K8S篇

chroot 是在 Unix 和 Linux 系统的一个操作，针对正在运作的软件行程和它的子进程，改变它外显的根目录。一个运行在这个环境下，经由 chroot 设置根目录的程序，它不能够对这个指定根目录之外的文件进行访问动作，不能读取，也不能更改它的内容。

虚拟化技术_VMware 、VirtualBox、KVM

虚拟化技术就是在操作系统上多加了一个虚拟化层（Hypervisor），可以将物理机的CPU、内存、硬盘、网络等资源进行虚拟化，再通过虚拟化出来的空间上安装操作系统，构建虚拟的应用程序执行环境。这就是我们通常说的虚拟机。

虚拟机的优点：

- 提升IT效率、降低运维成本
- 更快地部署工作负责
- 提高服务器可用性

虚拟机的缺点：

- 占用资源较多、性能较差
- 扩展、迁移能力较差

Why Docker

场景

- 开发人员在本地编写代码，并使用Docker容器与其他同事共享劳动成果。
- 使用Docker将应用程序推送到测试环境中，并执行自动和手动测试。
- 开发人员可以在开发环境中对其进行修复，然后将其重新部署到测试环境中以进行测试和验证。
- 测试完成后，将修补程序推送给生产环境就像将更新的镜像推送到生产环境一样简单。

需求

> 快速，一致地交付应用程序、镜像打包环境，避免了环境不一致的问题，简化开发的生命周期，适合于快速迭代敏捷开发的场景

特性	容器	虚拟机
启动速度	秒级	分钟级
性能	接近原生	较弱
内存代价	很小	较多
硬盘使用	一般为MB	一般为GB
运行密度	单机支持上千个容器	一般几十个
隔离性	安全隔离	完全隔离
迁移性	优秀	一般

核心概念

Docker引擎-守护进程

Docker使用C/S架构：用户通过Docker客户端与Docker守护进程（Docker引擎）通过Unix套接字或者RESTAPI进行通信，Docker引擎完成了构建，运行和分发Docker容器的繁重工作

Docker镜像-Dockerfile

Docker镜像类似于虚拟机镜像，是一个只读的模板，是创建Docker容器的基础

镜像是基于联合（Union）文件系统的一种层式的结构，由一系列指令一步一步构建出来。

比如：拷贝文件、执行命令

Docker仓库-Hub

Docker仓库可以分为公开仓库（Public）和私有仓库（Private）两种形式。

最大的公开仓库是官方提供的Docker Hub，其中存放了数量庞大的镜像供用户下载。

基本操作

镜像

```
[root@localhost ~]# docker pull mysql:5.7.30
5.7.30: Pulling from library/mysql ……
[root@localhost ~]# docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
mysql 5.7.30 9cfcce23593a 6 weeks ago 448MB
[root@localhost ~]# docker tag mysql:5.7.30 mysql5
[root@localhost ~]# docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
mysql5 latest 9cfcce23593a 6 weeks ago 448MB
mysql 5.7.30 9cfcce23593a 6 weeks ago 448MB
[root@localhost ~]# docker inspect mysql:5.7.30
[{显示docker 详细信息}]
[root@localhost ~]# docker search mysql
[root@localhost ~]# docker rmi mysql:5.7.30
[root@localhost ~]# docker push mysql[:TAG]
```

容器

```
[root@localhost ~]# docker create -it nginx
[root@localhost ~]# docker start 9cfcce23593a

#查看运行的容器
[root@localhost ~]# docker ps
#查看所有容器
[root@localhost ~]# docker ps -a
#新建并启动容器
[root@localhost ~]# docker run -it --rm --network host tomcat:8.5.56-jdk8-openjdk
```

实战

1. 创建一个卷，待后边使用

```
docker volume create test_volume
```

1. 分别启动2个容器挂在上卷,

```
在2个终端窗口启动2个容器
docker run -it --rm -v test_volume:/test nginx:latest /bin/bash
docker run -it --rm -v test_volume:/test nginx:latest /bin/bash
cd /test;
touch a.txt
ls /test # 在两个容器中我们均可以看到我们创建的文件，shixian在多个容器之间实现数据共享
```

挂载在容器 /test 目录内创建。 Docker 不支持容器内安装点的相对路径。 多个容器可以在同一时间段内使用相同的卷。如果两个容器需要访问共享数据，例如，如果一个容器写入而另一个容器读取数据。 卷名 在驱动程序test必须唯一。这意味着不能将相同的卷名与两个不同的驱动程序一起使用。 如果我们指定了当前testvolume程序上已在使用的卷名，则Docker会假定我们要重用现有卷，并且不会返回错误。如果开始无 testvolume 则会创建这个卷当然除了使用卷，也可以使用将宿主机的文件映射到容器的卷，命令类似，只不过不用提前创建卷，而且数据会映射到宿主机上注意如果宿主机上的目录可以不存在，会在启动容器的时候创建

Netty篇

核心组件

1. 整体结构

Core 核心层

Core 核心层是 Netty 最精华的内容，它提供了底层网络通信的通用抽象和实现，包括事件模型、通用API、支持零拷贝的 ByteBuf 等。

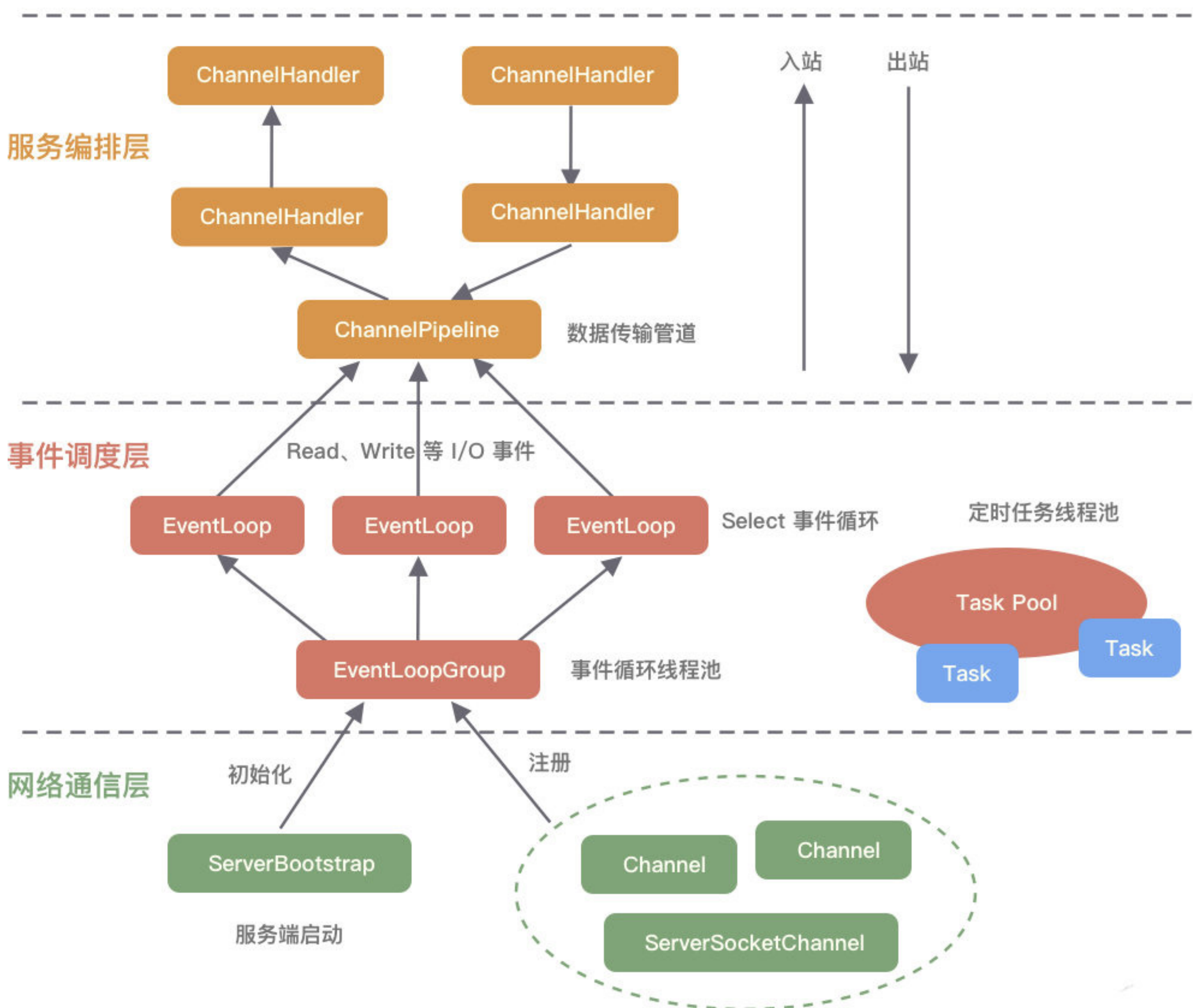
Protocol Support 协议支持层

协议支持层基本上覆盖了主流协议的编解码实现，如 HTTP、Protobuf、WebSocket、二进制等主流协议，此外 Netty 还支持自定义应用层协议。Netty 丰富的协议支持降低了用户的开发成本，基于 Netty 我们可以快速开发 HTTP、WebSocket 等服务。

Transport Service 传输服务层

传输服务层提供了网络传输能力的定义和实现方法。它支持 Socket、HTTP 隧道、虚拟机管道等传输方式。Netty 对 TCP、UDP 等数据传输做了抽象和封装，用户可以更聚焦在业务逻辑实现上，而不必关系底层数据传输的细节。

2. 逻辑架构



网络通信层

网络通信层的职责是执行网络 I/O 的操作。它支持多种网络协议和 I/O 模型的连接操作。当网络数据读取到内核缓冲区后，会触发各种网络事件，这些网络事件会分发给事件调度层进行处理。

网络通信层的核心组件包含BootStrap、ServerBootStrap、Channel三个组件。

Bootstrap 是“引导”的意思，负责 Netty 客户端程序的启动、初始化、服务器连接等过程，串联了 Netty 的其他核心组件。

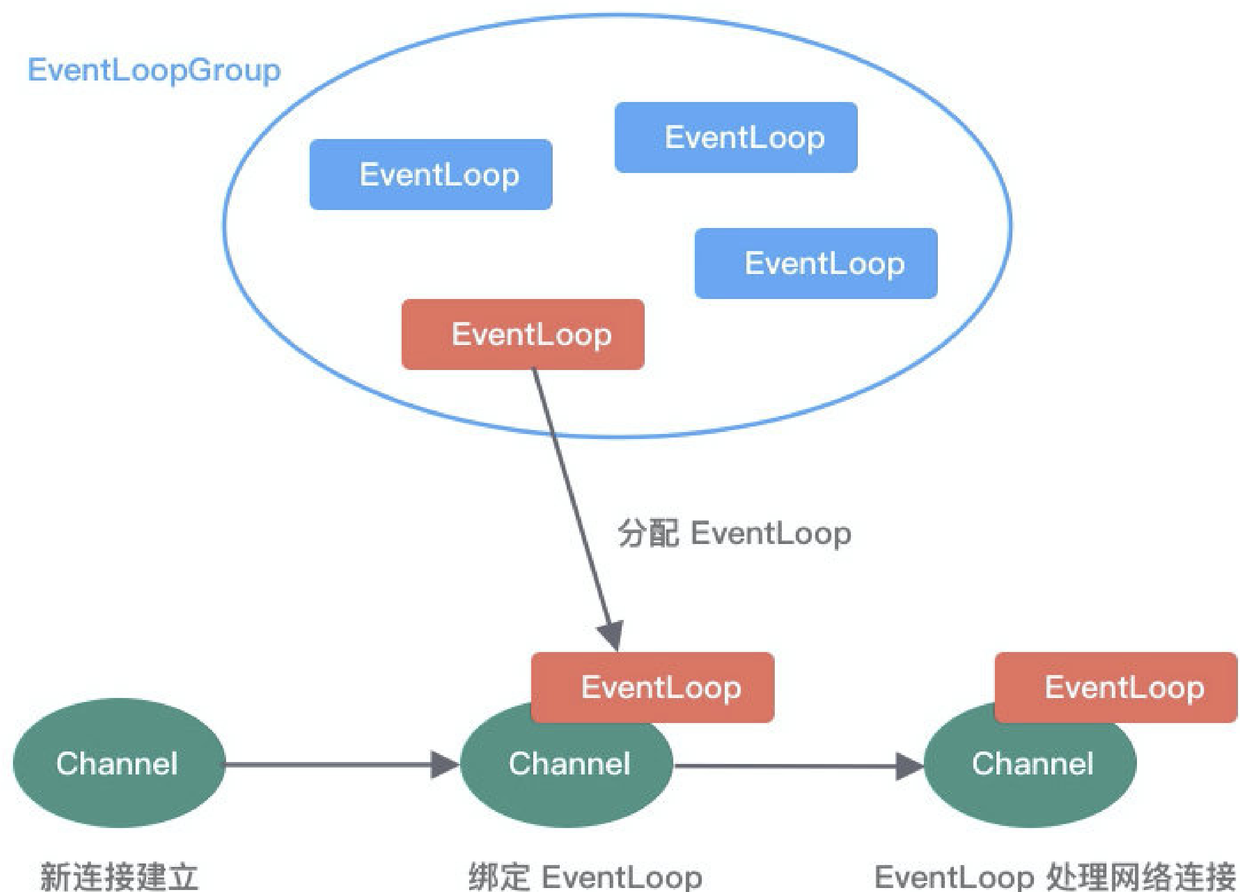
ServerBootStrap 用于服务端启动绑定本地端口，会绑定Boss 和 Worker两个 EventLoopGroup。

Channel 的是“通道”，Netty Channel提供了基于NIO更高层次的抽象，如 register、bind、connect、read、write、flush 等。

事件调度层

事件调度层的职责是通过 Reactor 线程模型对各类事件进行聚合处理，通过 Selector 主循环线程集成多种事件（ I/O 事件、信号事件、定时事件等），实际的业务处理逻辑是交由服务编排层中相关的 Handler 完成。

事件调度层的核心组件包括 EventLoopGroup、EventLoop。



EventLoop

负责处理 Channel 生命周期内的所有 I/O 事件，如 accept、connect、read、write 等 I/O 事件

①一个 EventLoopGroup 往往包含**一个或者多个** EventLoop。

②EventLoop 同一时间会与一个Channel绑定，每个 EventLoop 负责**处理一种类型 Channel。**

③Channel 在生命周期内可以对和多个 EventLoop 进行**多次绑定和解绑。**

EventLoopGroup

是Netty的**核心处理引擎**，本质是一个线程池，主要负责接收 I/O 请求，并分配线程执行处理请求。通过创建不同的 EventLoopGroup 参数配置，就可以支持 Reactor 的三种线程模型：

- 单线程模型：EventLoopGroup 只包含一个 EventLoop，Boss 和 Worker 使用同一个EventLoopGroup；
- 多线程模型：EventLoopGroup 包含多个 EventLoop，Boss 和 Worker 使用同一个EventLoopGroup；
- 主从多线程模型：EventLoopGroup 包含多个 EventLoop，Boss 是主 Reactor，Worker 是从 Reactor，它们分别使用不同的 EventLoopGroup，主 Reactor 负责新的网络连接 Channel 创建，然后把 Channel 注册到从 Reactor。

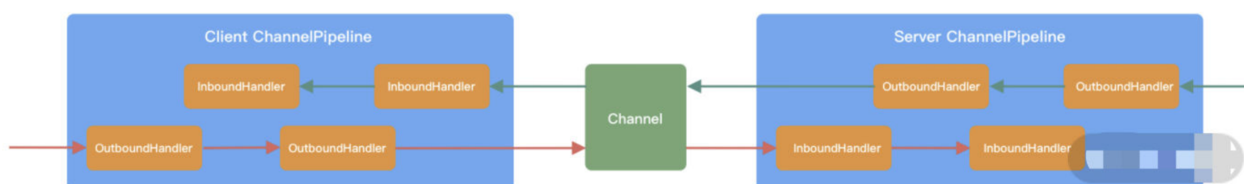
服务编排层

服务编排层的职责是负责组装各类服务，它是 Netty 的核心处理链，用以实现网络事件的动态编排和有序传播。

服务编排层的核心组件包括 **ChannelPipeline、ChannelHandler、ChannelHandlerContext。**

ChannelPipeline

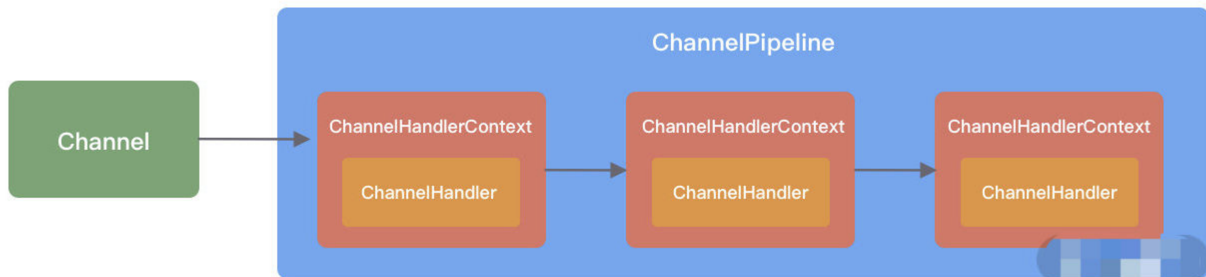
是 Netty 的核心编排组件，负责组装各种 ChannelHandler，ChannelPipeline 内部通过双向链表将不同的 ChannelHandler 链接在一起。当 I/O 读写事件触发时，Pipeline 会依次调用 Handler 列表对 Channel 的数据进行拦截和处理。



客户端和服务端都有各自的 ChannelPipeline。客户端和服务端一次完整的请求：客户端出站（Encoder 请求数据）、服务端入站（Decoder接收数据并执行业务逻辑）、服务端出站（Encoder响应结果）。

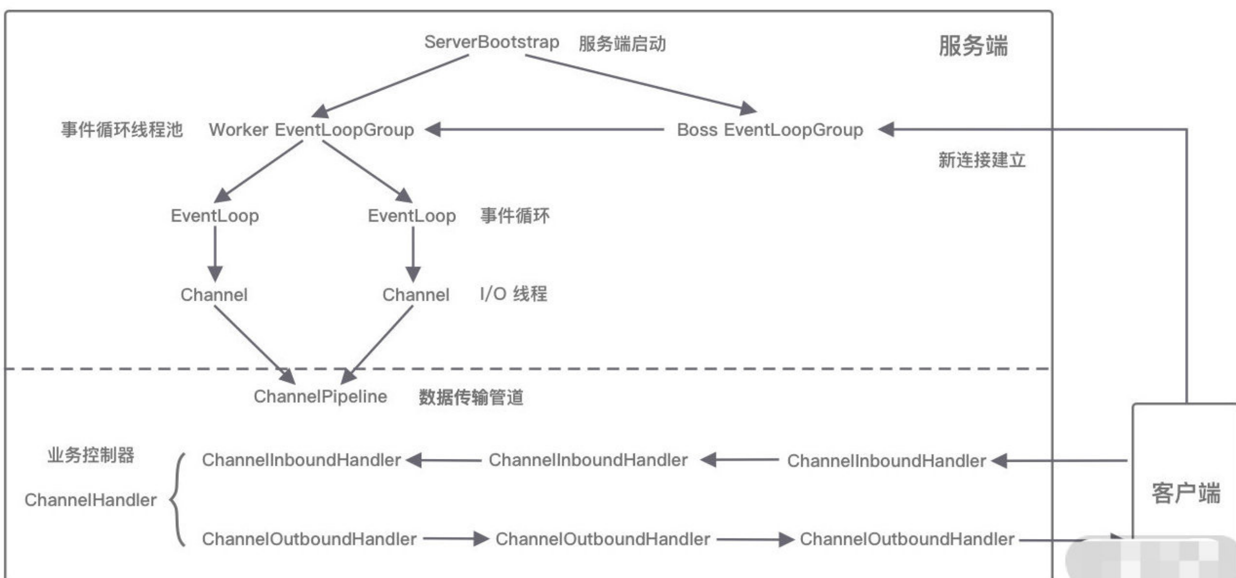
ChannelHandler

完成数据的编解码以及处理工作。



ChannelHandlerContext

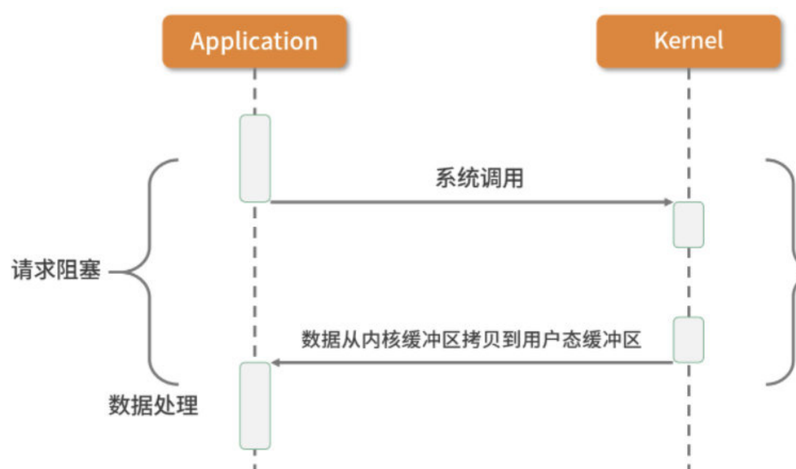
用于保存Handler 上下文，通过 HandlerContext 我们可以知道 Pipeline 和 Handler 的关联关系。HandlerContext 可以实现 Handler 之间的交互，HandlerContext 包含了 Handler 生命周期的所有事件，如 connect、bind、read、flush、write、close 等。同时，HandlerContext 实现了Handler通用的逻辑的模型抽象。



网络传输

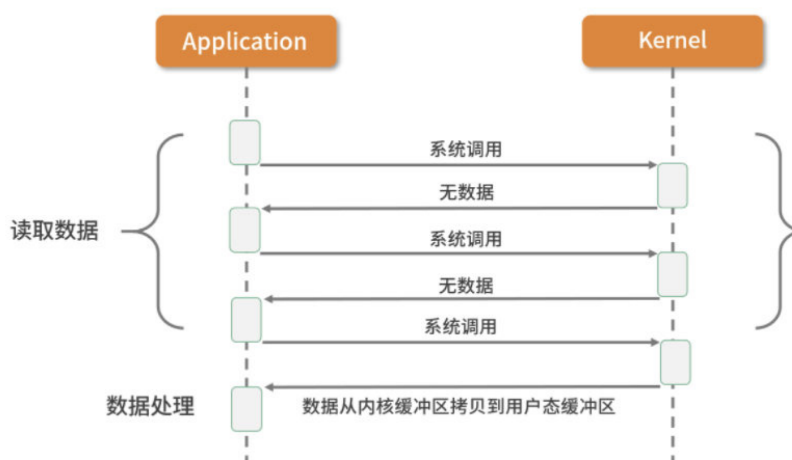
1. 五种IO模型的区别

阻塞I/O：（BIO）



应用进程向内核发起 I/O 请求，发起调用的线程一直等待内核返回结果。一次完整的 I/O 请求称为BIO（Blocking IO，阻塞 I/O），所以 BIO 在实现异步操作时，只能使用多线程模型，一个请求对应一个线程。但是，**线程的资源是有限且宝贵的，创建过多的线程会增加线程切换的开销。**

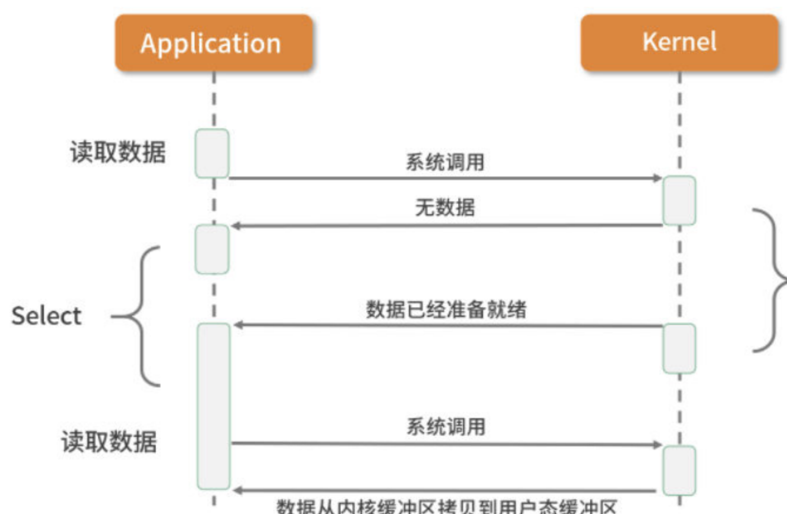
同步非阻塞I/O（NIO）：



应用进程向内核发起 I/O 请求后不再会同步等待结果，而是会立即返回，通过轮询的方式获取请求结果。NIO 相比 BIO 虽然大幅提升了性能，但是轮询过程中大量的系统调用导致上下文切换开销很大。所以，单独使用非阻塞 I/O 时

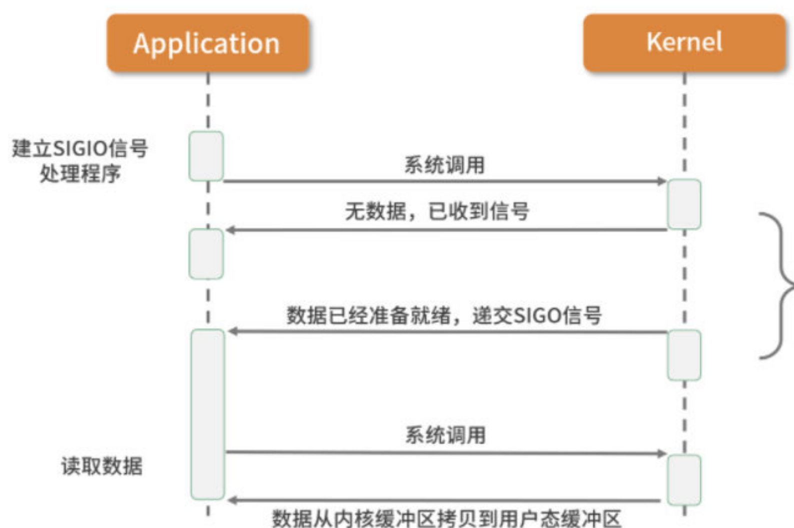
效率并不高，而且**随着并发量的提升，非阻塞 I/O 会存在严重的性能浪费。**

多路复用I/O（select和poll）：



多路复用实现了**一个线程处理多个 I/O 句柄的操作。**多路指的是多个数据通道，复用指的是使用一个或多个固定线程来处理每一个 Socket。select、poll、epoll 都是 I/O 多路复用的具体实现，线程一次 select 调用可以获取内核态中多个数据通道的数据状态。其中，select只负责等，recvfrom只负责拷贝，阻塞IO中可以对多个文件描述符进行阻塞监听，是一种非常高效的 I/O 模型。

信号驱动I/O（SIGIO）：

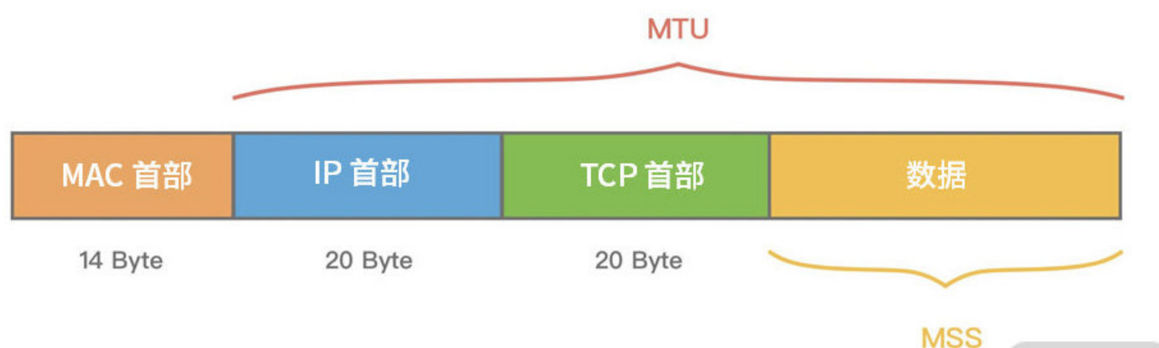


信号驱动IO模型，应用进程告诉内核：当数据报准备好的时候，给我发送一个信号，对SIGIO信号进行捕捉，并且调用我的信号处理函数来获取数据报。

Reactor 实现相对简单，适合处理耗时短的场景，对于耗时长的 I/O 操作容易造成阻塞。Proactor 性能更高，但是实现逻辑非常复杂，适合图片或视频流分析服务器，目前主流的事件驱动模型还是依赖 select 或 epoll 来实现。

3. 拆包粘包问题

拆包TCP 传输协议是面向流的，没有数据包界限。 MTU（Maxitum Transmission Unit）是链路层一次最大传输数据的大小。MTU 一般来说大小为 1500 byte。MSS（Maximum Segement Size）是指 TCP 最大报文段长度，它是传输层一次发送最大数据的大小。



如上图所示，如果 MSS + TCP 首部 + IP 首部 > MTU，那么数据包将会被拆分为多个发送。这就是拆包现象。

Nagle 算法

Nagle算法可以理解为批量发送，也是我们平时编程中经常用到的优化思路，它是在数据未得到确认之前先写入缓冲区，等待数据确认或者缓冲区积攒到一定大小再把数据包发送出去。Netty　中为了使数据传输延迟最小化，就默认禁用了 Nagle 算法。

拆包/粘包的解决方案

在客户端和服务端通信的过程中，服务端一次读到的数据大小是不确定的。需要确定边界：

消息长度固定

特定分隔符

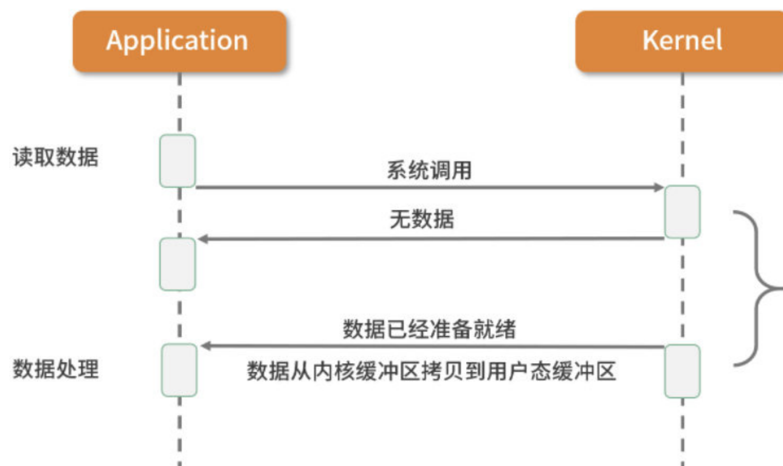
消息长度 + 消息内容(Netty)

4. 自定义协议

Netty 常用编码器类型：

```
MessageToByteEncoder //对象编码成字节流；
MessageToMessageEncoder //一种消息类型编码成另外一种消息类型。
```

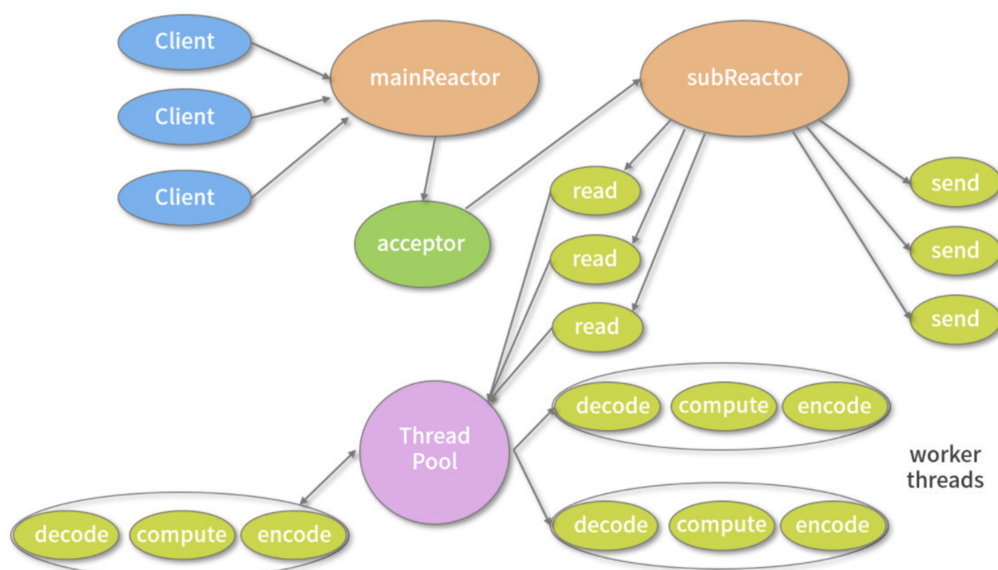
异步I/O（Posix.1的aio_系列函数）：



当应用程序调用aioread时，内核一方面去取数据报内容返回，另一方面将程序控制权还给应用进程，应用进程继续处理其他事情，是一种非阻塞的状态。当内核中有数据报就绪时，由内核将数据报拷贝到应用程序中，返回aioread中定义好的函数处理程序。

2. Reactor多线程模型

Netty 的 I/O 模型是基于非阻塞 I/O 实现的，底层依赖的是 NIO 框架的多路复用器 Selector。采用 epoll 模式后，只需要一个线程负责 Selector 的轮询。当有数据处于就绪状态后，需要一个事件分发器（Event Dispather），它负责将读写事件分发给对应的读写事件处理器（Event Handler）。事件分发器有两种设计模式：Reactor 和 Proactor，Reactor 采用同步 I/O， Proactor 采用异步 I/O。



Netty 常用解码器类型：

```
ByteToMessageDecoder/ReplayingDecoder //将字节流解码为消息对象；
MessageToMessageDecoder //将一种消息类型解码为另外一种消息类型。
```

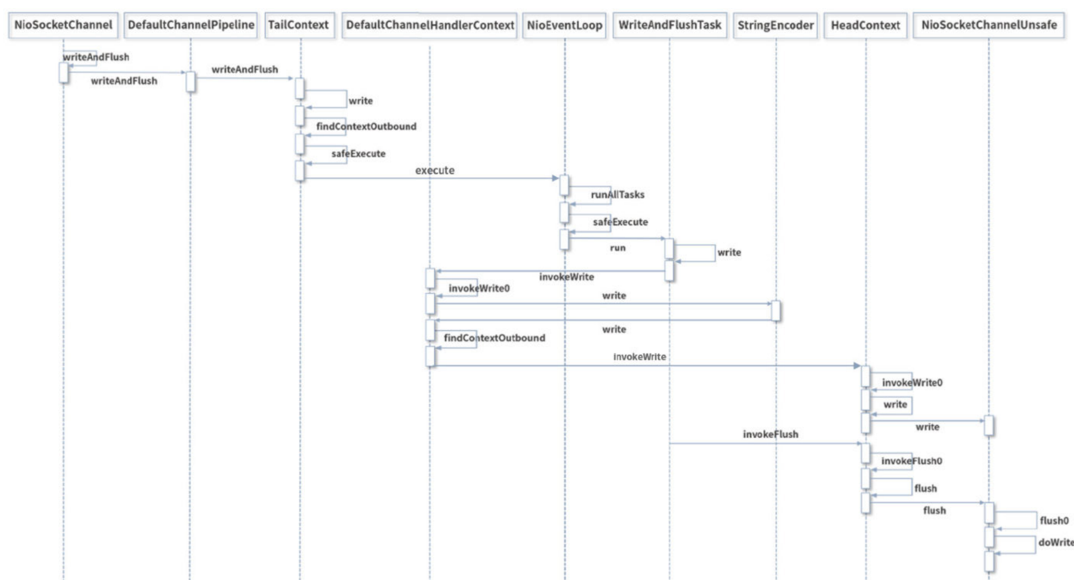
编解码器可以分为一次解码器和二次解码器，一次解码器用于解决 TCP 拆包/粘包问题，按协议解析后得到的字节数据。如果你需要对解析后的字节数据做对象模型的转换，这时候便需要用到二次解码器，同理编码器的过程是反过来的。

Netty自定义协议内容：

```
/*
+---------------------------------------------------------------+
| 魔数 2byte | 协议版本号 1byte | 序列化算法 1byte | 报文类型 1byte  |
+---------------------------------------------------------------+
| 状态 1byte |        保留字段 4byte     |      数据长度 4byte     |
+---------------------------------------------------------------+
|                   数据内容 （长度不定）                          |
+---------------------------------------------------------------+
 */
```

如何判断 ByteBuf 是否存在完整的报文？最常用的做法就是通过读取消息长度 dataLength 进行判断。如果 ByteBuf 的可读数据长度小于 dataLength，说明 ByteBuf 还不够获取一个完整的报文。

5. WriteAndFlush



① writeAndFlush 属于出站操作，它是从 Pipeline 的 Tail 节点开始进行事件传播，一直向前传播到 Head 节点。不管在 write 还是 flush 过程，Head 节点都中扮演着重要的角色。

② write 方法并没有将数据写入 Socket 缓冲区，只是将数据写入到 ChannelOutboundBuffer 缓存中，ChannelOutboundBuffer 缓存内部是由单向链表实现的。

③ flush 方法才最终将数据写入到 Socket 缓冲区。

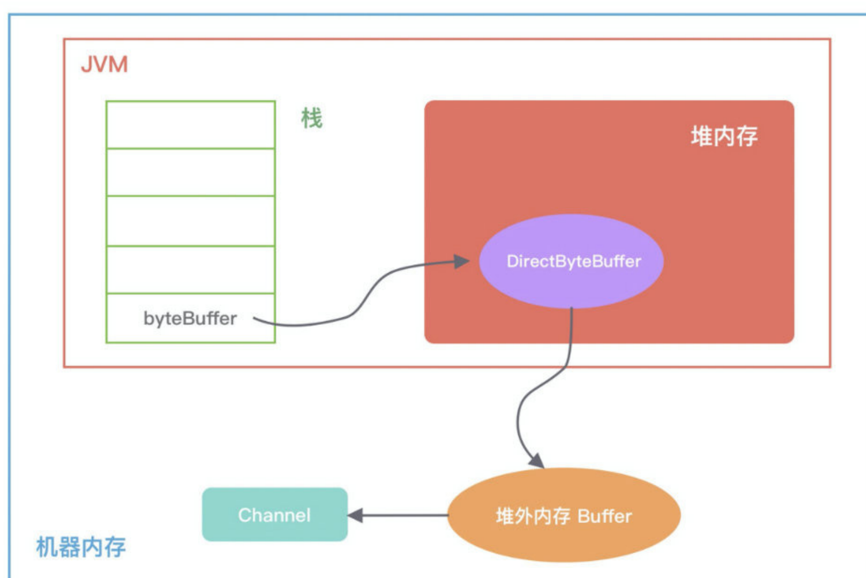
内存管理

1. 堆外内存

在 Java 中对象都是在堆内分配的，通常我们说的JVM 内存也就指的堆内内存，堆内内存完全被JVM 虚拟机所管理，JVM 有自己的垃圾回收算法，对于使用者来说不必关心对象的内存如何回收。堆外内存与堆内内存相对应，对于整个机器内存而言，除堆内内存以外部分即为堆外内存。堆外内存不受 JVM 虚拟机管理，直接由操作系统管理。使用堆外内存有如下几个优点：

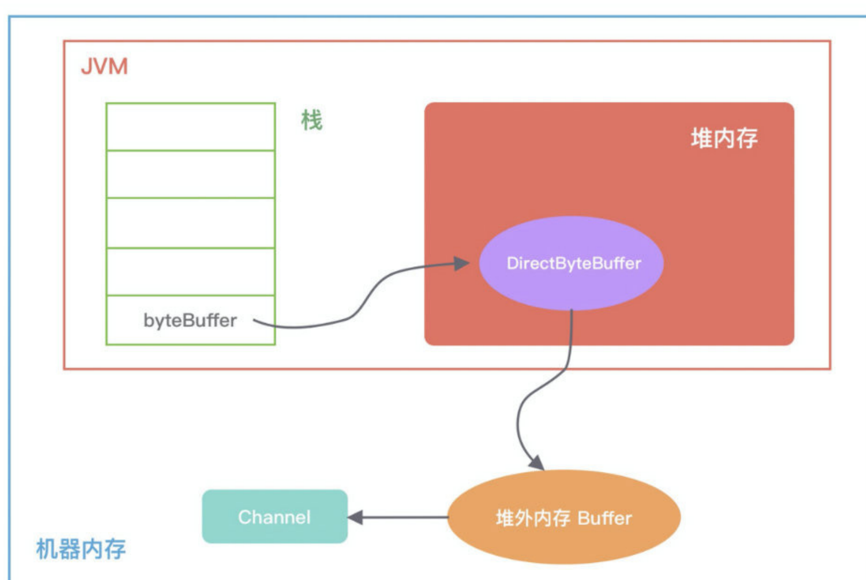
1. 堆内内存由 JVM GC 自动回收内存，降低了 Java 用户的使用心智，堆外内存由于不受 JVM 管理，所以在一定程度上可以降低 GC 对应用运行时带来的影响。
2. 堆外内存需要手动释放，这一点跟 C/C++ 很像，稍有不慎就会造成应用程序内存泄漏，当出现内存泄漏问题时排查起来会相对困难。
3. 当进行网络 I/O 操作、文件读写时，堆内内存都需要转换为堆外内存，然后再与底层设备进行交互，所以直接使用堆外内存可以减少一次内存拷贝。
4. 堆外内存可以方便实现进程之间、JVM 多实例之间的数据共享。

在堆内存放的 DirectByteBuffer 对象并不大，仅仅包含堆外内存的地址、大小等属性，同时还会创建对应的 Cleaner 对象，通过 ByteBuffer 分配的堆外内存不需要手动回收，它可以被 JVM 自动回收。当堆内的 DirectByteBuffer 对象被 GC 回收时，Cleaner 就会用于回收对应的堆外内存。



从 DirectByteBuffer 的构造函数中可以看出，真正分配堆外内存的逻辑还是通过 unsafe.allocateMemory(size)，Unsafe 是一个非常不安全的类，它用于执行内存访问、分配、修改等敏感操作，可以越过 JVM 限制的枷锁。Unsafe 最初并不是为开发者设计的，使用它时虽然可以获取对底层资源的控制权，但也失去了安全性的保证，使用 Unsafe 一定要慎重（Java 中是不能直接使用 Unsafe 的，但是可以通过反射获取 Unsafe 实例）。Netty 中依赖了 Unsafe 工具类，是因为 Netty 需要与底层 Socket 进行交互，Unsafe 提升 Netty 的性能

因为DirectByteBuffer 对象的回收需要依赖 Old GC 或者 Full GC 才能触发清理，如果长时间没有 GC执行，那么堆外内存即使不再使用，也会一直在占用内存不释放，很容易将机器的物理内存耗尽。-XX:MaxDirectMemorySize 指定堆外内存的上限大小，超出时触发GC，仍无法释放抛出OOM异常。

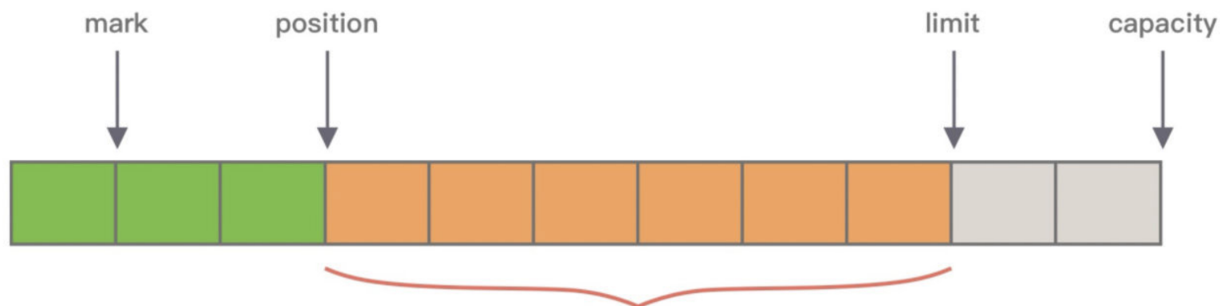


当初始化堆外内存时，内存中的对象引用情况如下图所示，first 是 Cleaner 类中的静态变量，Cleaner 对象在初始化时会加入 Cleaner 链表中。DirectByteBuffer 对象包含堆外内存的地址、大小以及 Cleaner 对象的引用，ReferenceQueue 用于保存需要回收的 Cleaner 对象。

2. 数据载体ByteBuf

JDK NIO 的 ByteBuffer

- mark：为某个读取过的关键位置做标记，方便回退到该位置；
- position：当前读取的位置；
- limit：buffer 中有效的数据长度大小；
- capacity：初始化时的空间容量。

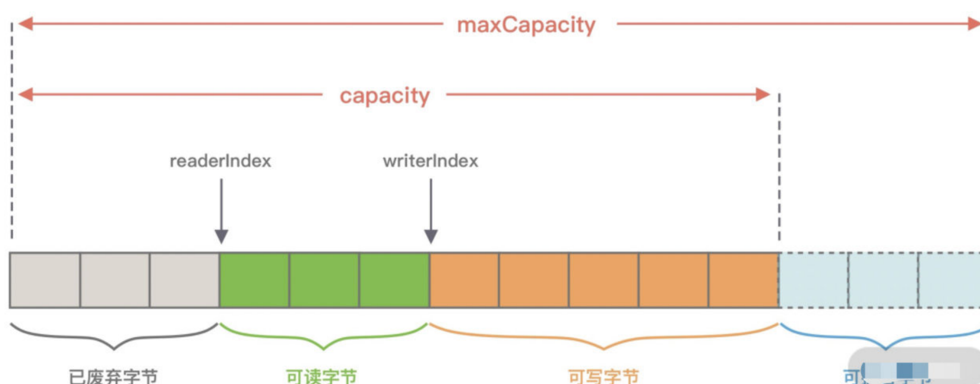


第一，ByteBuffer 分配的长度是固定的，无法动态扩缩容，每次在存放数据的时候对容量大小做校验，扩容需要将已有的数据迁移。

第二，ByteBuffer 只能通过 position 获取当前可操作的位置，因为读写共用的 position 指针，所以需要频繁调用 flip、rewind 方法切换读写状态。

Netty中的ByteBuf

- **废弃字节**，表示已经丢弃的无效字节数据。
- **可读字节**，表示 ByteBuf 中可以被读取的字节内容，可以通过 writeIndex – readerIndex 计算得出。当读写位置重叠时时，表示 ByteBuf 已经不可读。
- **可写字节**，向 ByteBuf 中写入数据都会存储到可写字节区域。当 writeIndex 超过 capacity，表示 ByteBuf 容量不足，需要扩容。
- **可扩容字节**，表示 ByteBuf 最多还可以扩容多少字节，最多扩容到 maxCapacity 为止，超过 maxCapacity 再写入就会出错。



引用计数

当byteBuf当引用计数为 0，该 ByteBuf 可以被放入到对象池中，避免每次使用 ByteBuf 都重复创建。

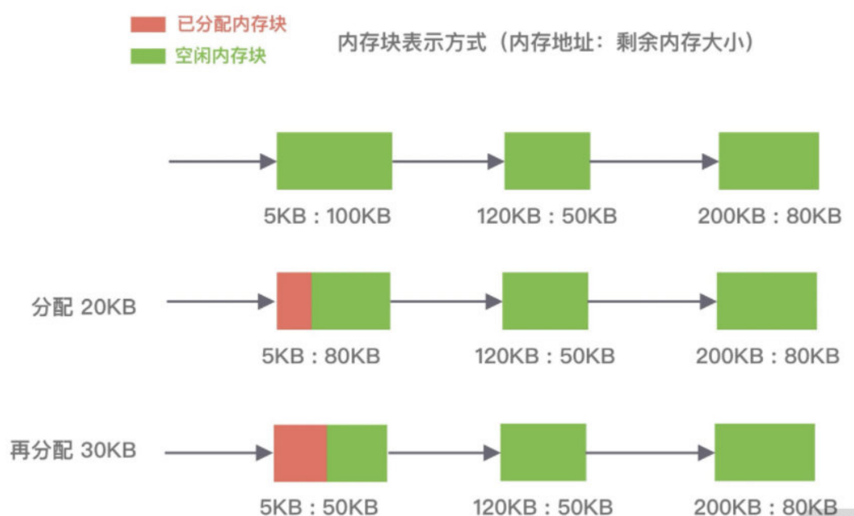
JVM 并不知道 Netty 的引用计数是如何实现的，当 ByteBuf 对象不可达时，一样会被 GC 回收掉，但是如果此时 ByteBuf 的引用计数不为 0，那么该对象就不会释放或者被放入对象池，从而发生了内存泄漏。Netty 会对分配的 ByteBuf 进行抽样分析，检测 ByteBuf 是否已经不可达且引用计数大于 0，判定内存泄漏的位置并输出到日志中，通过关注日志中 LEAK 关键字可以找到内存泄漏的具体对象。

3. 内存分配jemalloc

为了减少分配时产生的内部碎片和外部碎片，常见的内存分配算法动态内存分配、伙伴算法和Slab 算法

动态内存分配（DMA）

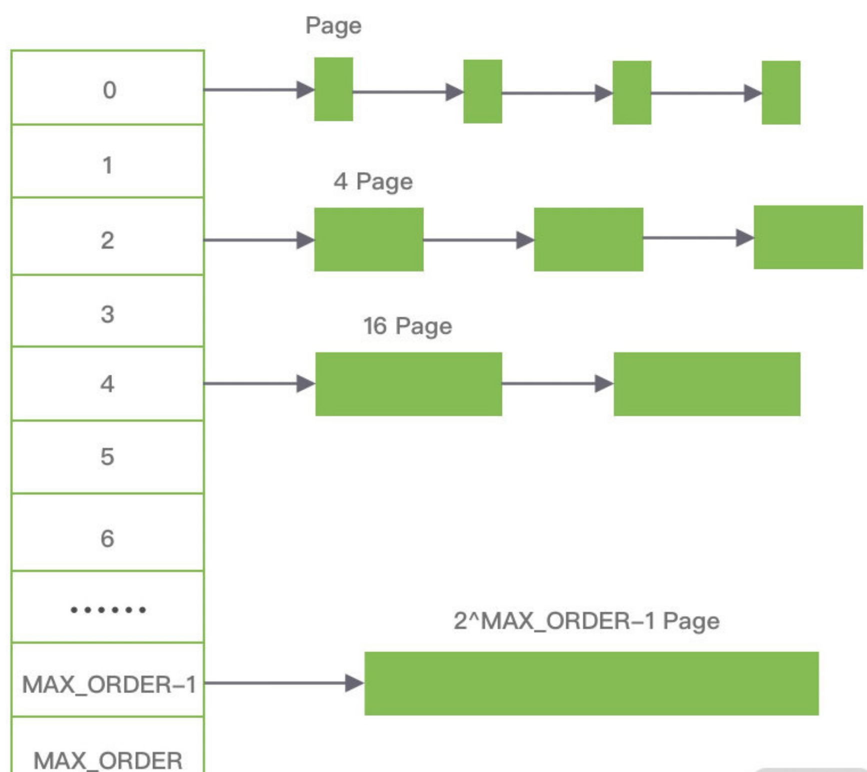
首次适应算法（first fit），空闲分区链以地址递增的顺序将空闲分区以双向链表的形式连接在一起，从空闲分区链中找到第一个满足分配条件的空闲分区，然后从空闲分区中划分出一块可用内存给请求进程，剩余的空闲分区仍然保留在空闲分区链中。



循环首次适应算法（next fit）不再是每次从链表的开始进行查找，而是从上次找到的空闲分区的以后开始查找。查找效率提升，会产生更多的碎片。

最佳适应算法（best fit），空闲分区链以空闲分区大小递增的顺序将空闲分区以双向链表的形式连接在一起，每次从空闲分区链的开头进行查找。

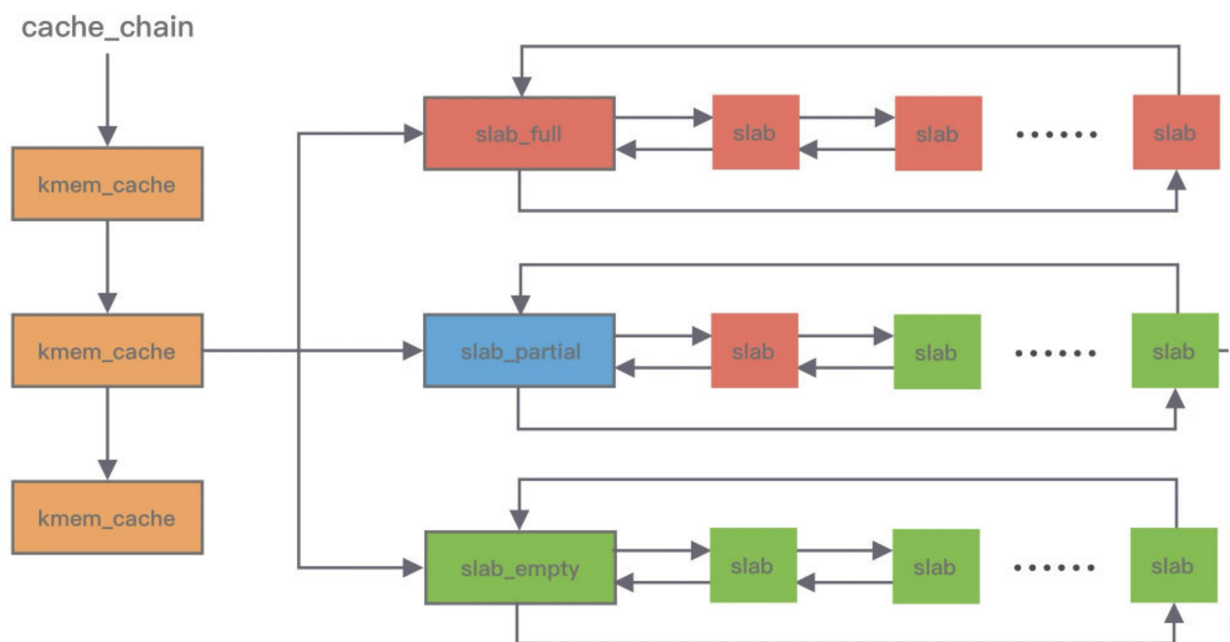
伙伴算法（外部碎片少，内部碎片多）是一种非常经典的内存分配算法，它采用了分离适配的设计思想，将物理内存按照 2 的次幂进行划分，内存分配时也是按照 2 的次幂大小进行按需分配



1. 首先需要找到存储 2^4 连续 Page 所对应的链表，即数组下标为 4；
2. 查找 2^4 链表中是否有空闲的内存块，如果有则分配成功；
3. 如果 2^4 链表不存在空闲的内存块，则继续沿数组向上查找，即定位到数组下标为 5 的链表，链表中每个节点存储 2^5 的连续 Page；
4. 如果 2^5 链表中存在空闲的内存块，则取出该内存块并将它分割为 2 个 2^4 大小的内存块，其中一块分配给进程使用，剩余的一块链接到 2^4 链表中。

Slab 算法（解决伙伴算法内部碎片问题）

Slab 算法在伙伴算法的基础上，对小内存的场景专门做了优化，采用了内存池的方案，解决内部碎片问题。

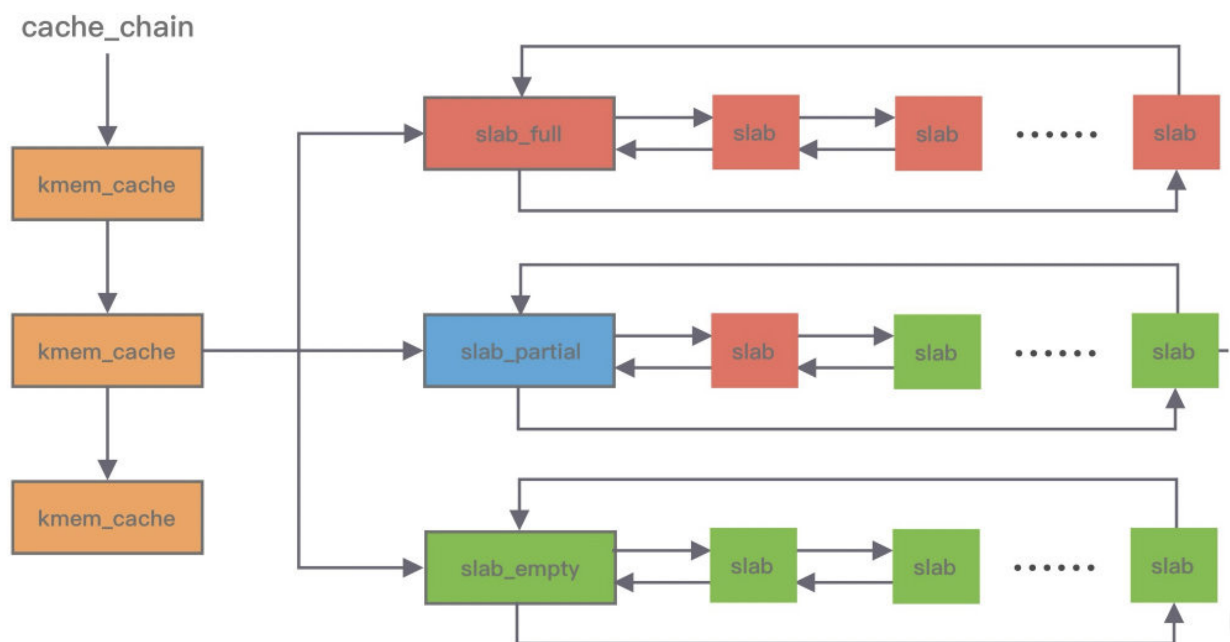


在 Slab 算法中维护着大小不同的 Slab 集合，将这块内存划分为大小相同的 slot，不会对内存块再进行合并，同时使用位图 bitmap 记录每个 slot 的使用情况。

kmemcache 中包含三个 Slab 链表：**完全分配使用 slabfull、部分分配使用 slabpartial和完全空闲 slabsempty，这三个链表负责内存的分配和释放。Slab 算法是基于对象进行内存管理的，它把相同类型的对象分为一类。当分配内存时，从 Slab 链表中划分相应的内存单元；单个 Slab 可以在不同的链表之间移动，例如当一个 Slab 被分配完，就会从 slabpartial 移动到 slabsfull，当一个 Slab 中有对象被释放后，就会从 slabfull 再次回到 slabpartial，所有对象都被释放完的话，就会从 slabpartial 移动到 slabempty。当释放内存时，Slab 算法并不会丢弃已经分配的对象，而是将它保存在缓存中，当下次再为对象分配内存时，直接会使用最近释放的内存块**。

4. jemalloc 架构

内存是由一定数量的 arenas 负责管理，线程均匀分布在 arenas 当中；
每个 arena 都包含一个 bin 数组，每个 bin 管理不同档位的内存块；
每个 arena 被划分为若干个 chunks，每个 chunk 又包含若干个 runs，每个 run 由连续的 Page 组成，run 才是实际分配内存的操作对象；
每个 run 会被划分为一定数量的 regions，在小内存的分配场景，region 相当于用户内存；
每个 tcache 对应一个 arena，tcache 中包含多种类型的 bin。



内存管理Arena，内存由一定数量的 arenas 负责管理。每个用户线程采用 round-robin 轮询的方式选择可用的 arena 进行内存分配。

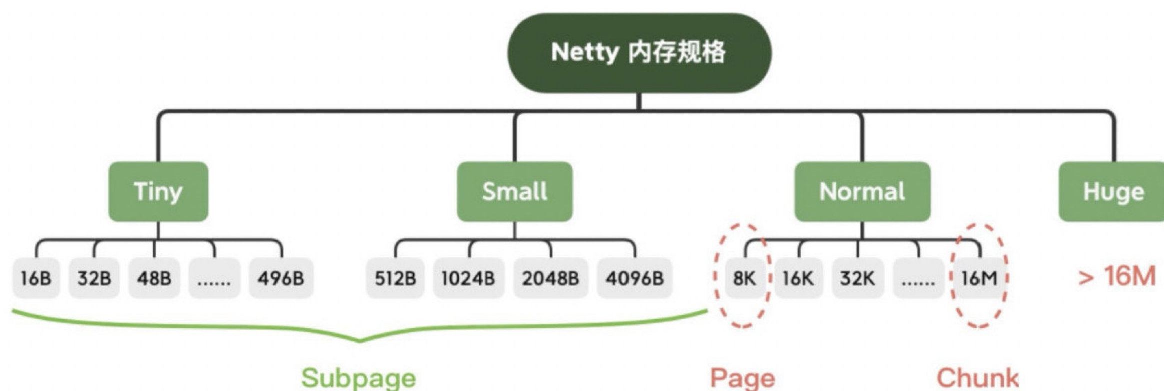
分级管理Bin，每个 bin 管理的内存大小是按分类依次递增。jemalloc 中小内存的分配是基于 Slab 算法完成的，会产生不同类别的内存块。

Page集合chunk，chunk 以 Page 为单位管理内存。每个 chunk 可被用于多次小内存的申请，但是在大内存分配的场景下只能分配一次。

实际分配单位run，run 结构具体的大小由不同的 bin 决定，例如 8 字节的 bin 对应的 run 只有一个 Page，可以从中选取 8 字节的块进行分配。

run 细分region，每个 run 会将划分为若干个等长的 region，每次内存分配也是按照 region 进行分发。

tcache 是每个线程私有的缓存，tcache 每次从 arena 申请一批内存，在分配内存时首先在 tcache 查找，避免锁竞争，分配失败才会通过 run 执行内存分配。



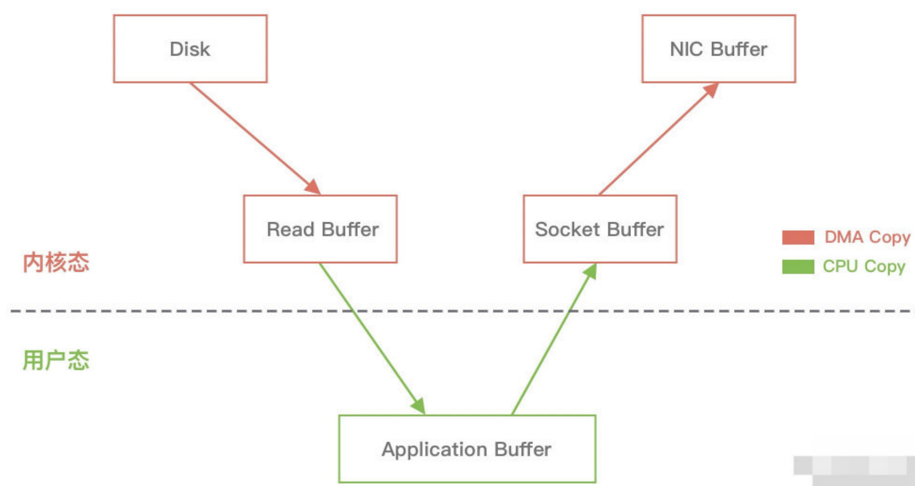
Small 场景，如果请求分配内存的大小小于 arena 中的最小的 bin，那么优先从线程中对应的 tcache 中进行分配。首先确定查找对应的 tbin 中是否存在缓存的内存块，如果存在则分配成功，否则找到 tbin 对应的 arena，从 arena 中对应的 bin 中分配 region 保存在 tbin 的 avail 数组中，最终从 availl 数组中选取一个地址进行内存分配，当内存释放时也会将被回收的内存块进行缓存。

Large 场景的内存分配与 Smalll 类似，如果请求分配内存的大小大于 arena 中的最小的 bin，但是不大于 tcache 中能够缓存的最大块，依然会通过 tcache 进行分配，但是不同的是此时会分配 chunk 以及所对应的 run，从 chunk 中找到相应的内存空间进行分配。内存释放时也跟 samll 场景类似，会把释放的内存块缓存在 tacache 的 tbin 中。此外还有一种情况，当请求分配内存的大小大于tcache 中能够缓存的最大块，但是不大于 chunk 的大小，那么将不会采用 tcache 机制，直接在 chunk 中进行内存分配。

Huge场景，如果请求分配内存的大小大于chunk的大小，那么直接通过 mmap 进行分配，调用 munmap 进行回收。

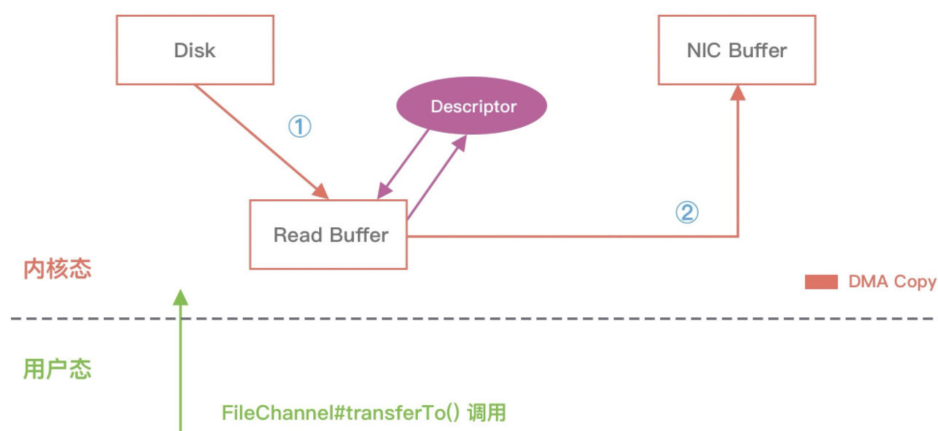
5. 零拷贝技术

1. 当用户进程发起 read() 调用后，上下文从用户态切换至内核态。DMA 引擎从文件中读取数据，并存储到内核态缓冲区，这里是第一次数据拷贝。
2. 请求的数据从内核态缓冲区拷贝到用户态缓冲区，然后返回给用户进程。第二次数据拷贝的过程同时，会导致上下文从内核态再次切换到用户态。
3. 用户进程调用 send() 方法期望将数据发送到网络中，用户态会再次切换到内核态，第三次数据拷贝请求的数据从用户态缓冲区被拷贝到 Socket 缓冲区。
4. 最终 send() 系统调用结束返回给用户进程，发生了第四次上下文切换。第四次拷贝会异步执行，从 Socket 缓冲区拷贝到协议引擎中。



在 Linux 中系统调用 sendfile() 可以实现将数据从一个文件描述符传输到另一个文件描述符，从而实现了零拷贝技术。

在 Java 中也使用了零拷贝技术，它就是 NIO FileChannel 类中的 transferTo() 方法，它可以将数据从 FileChannel 直接传输到另外一个 Channel。



Netty 中的零拷贝技术除了操作系统级别的功能封装，更多的是面向用户态的数据操作优化，主要体现在以下5个方面：

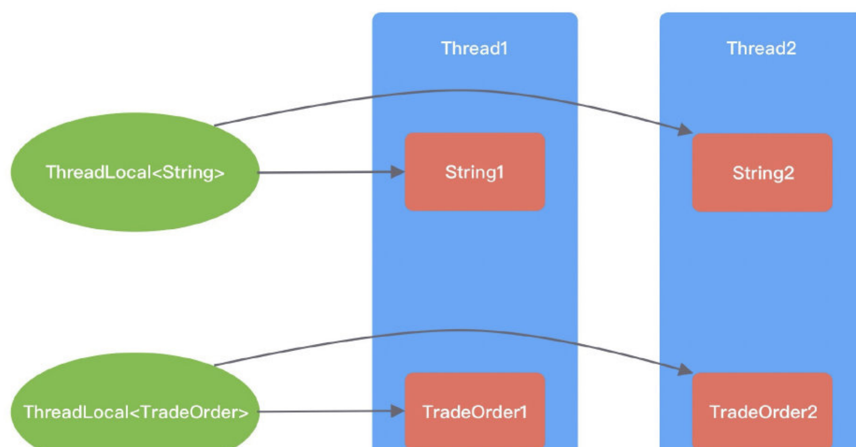
- 堆外内存，避免 JVM 堆内存到堆外内存的数据拷贝。
- CompositeByteBuf 类，可以组合多个 Buffer 对象合并成一个逻辑上的对象，避免通过传统内存拷贝的方式将几个 Buffer 合并成一个大的 Buffer。
- 通过 Unpooled.wrappedBuffer 可以将 byte 数组包装成 ByteBuf 对象，包装过程中不会产生内存拷贝。
- ByteBuf.slice ，slice 操作可以将一个 ByteBuf 对象切分成多个 ByteBuf 对象，切分过程中不会产生内存拷贝，底层共享一个 byte 数组的存储空间。
- Netty 使用 封装了transferTo() 方法 FileRegion，可以将文件缓冲区的数据直接传输到目标 Channel，避免内核缓冲区和用户态缓冲区之间的数据拷贝。

高性能数据结构

1. FastThreadLocal

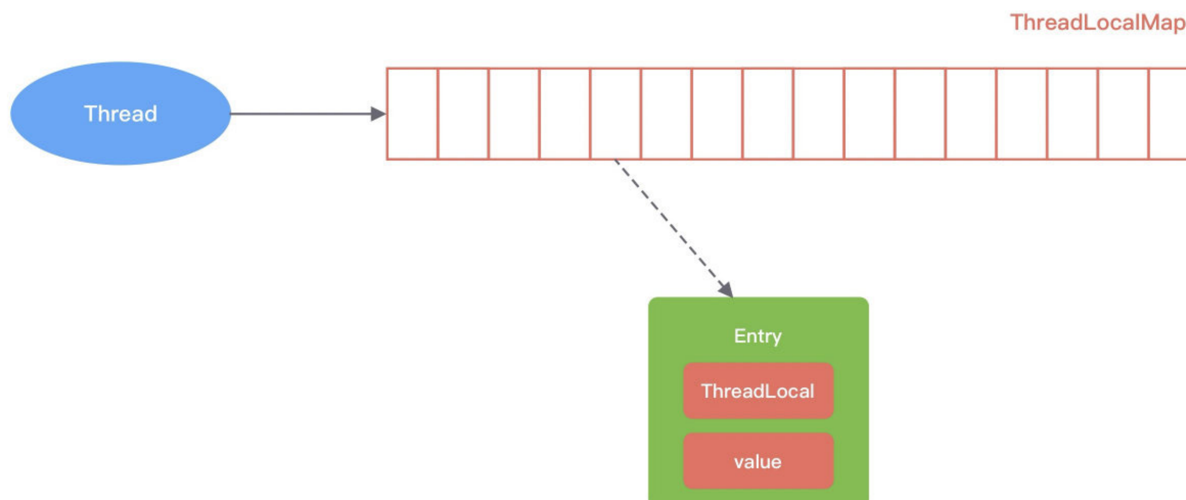
ThreadLocal 可以理解为线程本地变量。ThreadLocal 为变量在每个线程中都创建了一个副本，该副本只能被当前线程访问，多线程之间是隔离的，变量不能在多线程之间共享。这样每个线程修改变量副本时，不会对其他线程产生影响。

既然多线程访问 ThreadLocal 变量时都会有自己独立的实例副本，那么很容易想到的方案就是在 ThreadLocal 中维护一个 Map，记录线程与实例之间的映射关系。当新增线程和销毁线程时都需要更新 Map 中的映射关系，因为会存在多线程并发修改，所以需要保证 Map 是线程安全的。但是在高并发的场景并发修改 Map 需要加锁，势必会降低性能。

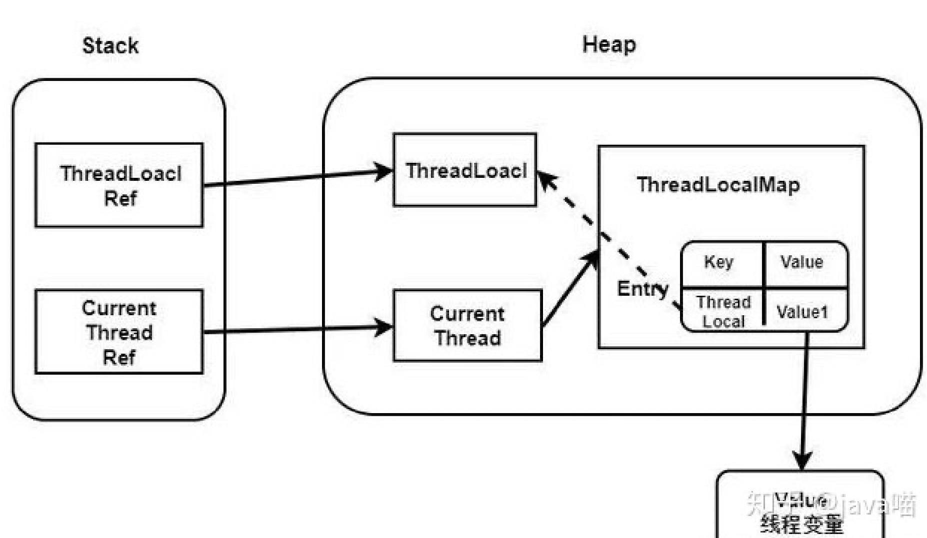


JDK 为了避免加锁，采用了相反的设计思路。以 Thread 入手，在 Thread 中维护一个 Map，记录 ThreadLocal 与实例之间的映射关系，这样在同一个线程内，Map 就不需要加锁了。

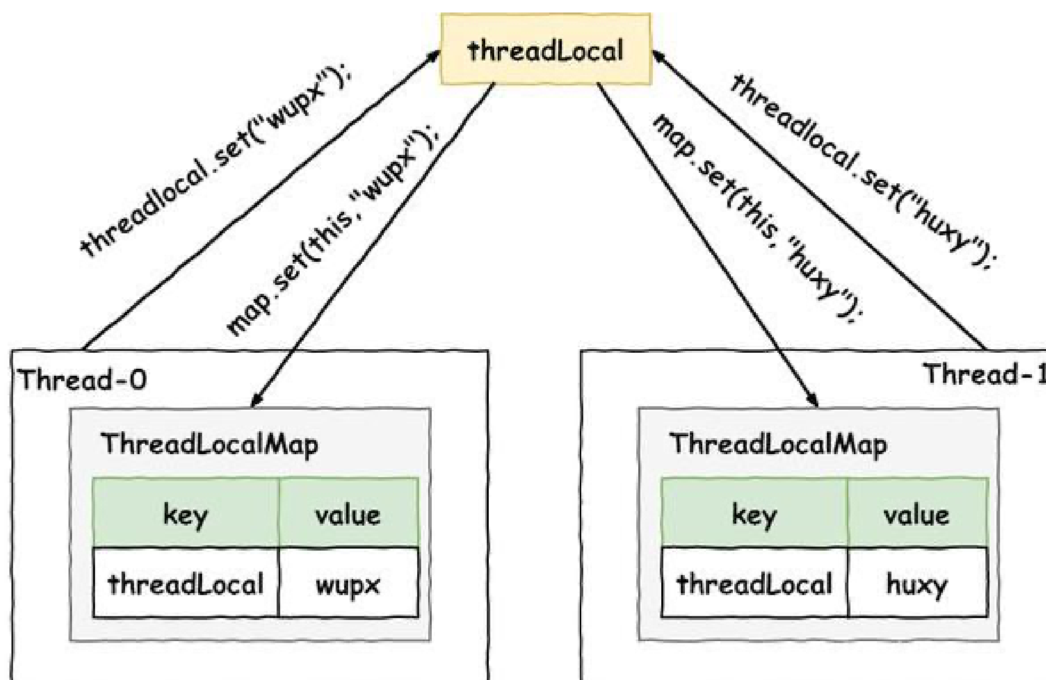
ThreadLocalMap 是一种使用线性探测法实现的哈希表，底层采用数组存储数据，通过魔数0x61c88647来使散列更加平衡。ThreadLocalMap 初始化一个长度为 16 的 Entry 数组。与 HashMap 不同的是，Entry 的 key 就是 ThreadLocal对象本身，value 就是用户具体需要存储的值。



Entry 继承自弱引用类 WeakReference，Entry 的 key 是弱引用，value 是强引用。在 JVM 垃圾回收时，只要发现了弱引用的对象，不管内存是否充足，都会被回收。那么为什么 Entry 的 key 要设计成弱引用呢？如果 key 都是强引用，当线 ThreadLocal 不再使用时，然而 ThreadLocalMap 中还是存在对 ThreadLocal 的强引用，那么 GC 是无法回收的，从而造成内存泄漏。



虽然 Entry 的 key 设计成了弱引用，但是当 ThreadLocal不再使用(业务逻辑走完，但是由于线程复用导致线程并没有结束)被 GC 回收后，ThreadLocalMap 中可能出现 Entry 的 key 为 NULL，那么 Entry 的 value 一直会强引用数据而得不到释放，只能等待线程销毁。那么应该如何避免 ThreadLocalMap 内存泄漏呢？ThreadLocal 已经帮助我们做了一定的保护措施，在执行 ThreadLocal.set()/get() 方法时，ThreadLocal 会清除 ThreadLocalMap 中 key 为 NULL 的 Entry 对象，让它还能够被 GC 回收。除此之外，当线程中某个 ThreadLocal 对象不再使用时，立即调用 remove() 方法删除 Entry 对象。如果是在异常的场景中，应在 finally 代码块中进行清理，保持良好的编码意识。在Netty中，可以方便的使用FashThreadLocal来防止内存泄漏



FastThreadLocal

FastThreadLocal 使用 Object 数组替代了 Entry 数组，Object[0] 存储的是一个Set> 集合，从数组下标 1 开始都是直接存储的 value 数据，不再采用 ThreadLocal 的键值对形式进行存储。主要是针对set方法，增加了两个额外的行为。

1. 找到数组下标 index 位置，设置新的 value。
2. 将 FastThreadLocal 对象保存到待清理的 Set 中。

	value1	value2	value3	value4	UNSET	UNSET	UNSET
0	1	2	3	4	5	6	

- **高效查找**。FastThreadLocal 在定位数据的时候可以直接根据数组下标 index 获取，时间复杂度 O(1)。而 JDK 原生的 ThreadLocal 在数据较多时哈希表很容易发生 Hash 冲突，线性探测法在解决 Hash 冲突时需要不停地向下寻找，效率较低。此外，FastThreadLocal 相比 ThreadLocal 数据扩容更加简单高效，FastThreadLocal 以 index 为基准向上取整到 2 的次幂作为扩容后容量，然后把原数据拷贝到新数组。而 ThreadLocal 由于采用的哈希表，所以在扩容后需要再做一轮 rehash。
- **安全性更高**。JDK 原生的 ThreadLocal 使用不当可能造成内存泄漏，只能等待线程销毁。在使用线程池的场景下，ThreadLocal 只能通过主动检测的方式防止内存泄漏，从而造成了一定的开销。然而 FastThreadLocal 不仅提供了 remove() 主动清除对象的方法，而且在线程池场景中 Netty 还封装了 FastThreadLocalRunnable，**任务执行完毕后一定会执行 FastThreadLocal.removeAll() 将 Set 集合中所有 FastThreadLocal 对象都清理掉**

2. HashedTimerWheel

生成月统计报表、每日得分结算、邮件定时推送

定时任务三种形式：

1. 按固定周期定时执行
2. 延迟一定时间后执行
3. 指定某个时刻执行

定时任务的三个关键方法：

1. Schedule 新增任务至任务集合；
2. Cancel 取消某个任务；
3. Run 执行到期的任务

JDK自带的三种定时器：Timer、DelayedQueue 和 ScheduledThreadPoolExecutor

> Timer小根堆队列，deadline　任务位于堆顶端，弹出的始终是最优先被执行的任务。Run　操作时间复杂度　O(1)，Schedule 和Cancel 操作的时间复杂度都是 O(logn)。

不论有多少任务被加入数组，始终由 异步线程TimerThread 负责处理。TimerThread 会定时轮询 TaskQueue 中的任务，如果堆顶的任务的 deadline 已到，那么执行任务；如果是周期性任务，执行完成后重新计算下一次任务的 deadline，并再次放入小根堆；如果是单次执行的任务，执行结束后会从 TaskQueue 中删除。

> DelayedQueue　采用优先级队列　PriorityQueue延迟获取对象的阻塞队列。DelayQueue中的每个对象都必须实现 Delayed 接口，并重写 compareTo 和 getDelay 方法。

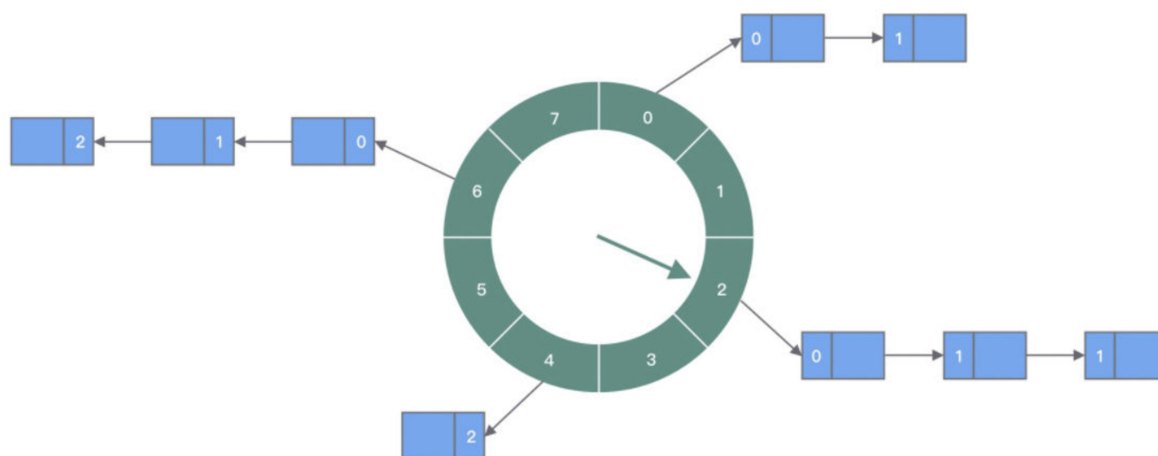
DelayQueue 提供了 put() 和 take() 的阻塞方法，可以向队列中添加对象和取出对象。对象被添加到 DelayQueue 后，会根据 compareTo() 方法进行优先级排序。getDelay() 方法用于计算消息延迟的剩余时间，只有 getDelay <=0 时，该对象才能从 DelayQueue 中取出。

DelayQueue 在日常开发中最常用的场景就是实现重试机制。例如，接口调用失败或者请求超时后，可以将当前请求对象放入 DelayQueue，通过一个异步线程 take() 取出对象然后继续进行重试。如果还是请求失败，继续放回 DelayQueue。可以设置重试的最大次数以及采用指数退避算法设置对象的 deadline，如 2s、4s、8s、16s ……以此类推。DelayQueue的时间复杂度和Timer基本一致。

> 为了解决 Timer 的设计缺陷，JDK 提供了功能更加丰富的 ScheduledThreadPoolExecutor，多线程、相对时间、对异常

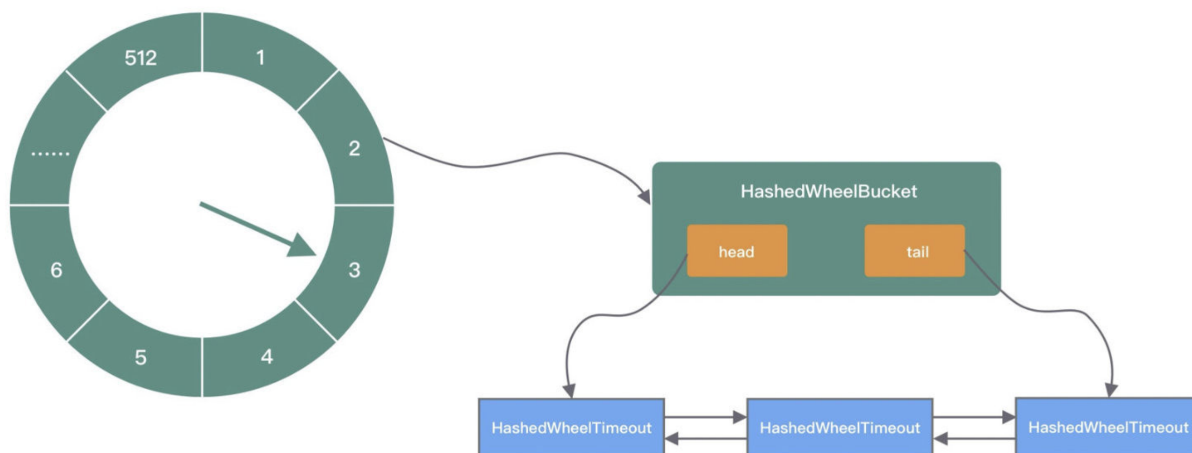
Timer 是单线程模式。如果某个 TimerTask 执行时间很久，会影响其他任务的调度。
Timer 的任务调度是基于系统绝对时间的，如果系统时间不正确，可能会出现问题。
TimerTask 如果执行出现异常，Timer 并不会捕获，会导致线程终止，其他任务永远不会执行。

时间轮原理分析



根据任务的到期时间进行取余和取模，然后根据取余结果将任务分布到不同的 slot 中，每个slot中根据round值决定是否操作，每次轮询到指定slot时，总时遍历最少round的对象进行执行，这样新增、执行两个操作的时间复杂度都近似O(1)。如果冲突较大可以增加数组长度，或者采用多级时间轮的方式处理。

```
public HashedWheelTimer(
        ThreadFactory threadFactory, //线程池，但是只创建了一个线程
        long tickDuration, //时针每次 tick 的时间，相当于时针间隔多久走到下一个 slot
        TimeUnit unit,         //表示 tickDuration 的时间单位，tickDuration * unit
        int ticksPerWheel, //时间轮上一共有多少个 slot，默认 512 个。
        boolean leakDetection,
        long maxPendingTimeouts) {//最大允许等待任务数
    // 省略其他代码
    wheel = createWheel(ticksPerWheel); // 创建时间轮的环形数组结构
    mask = wheel.length - 1; // 用于快速取模的掩码
    long duration = unit.toNanos(tickDuration); // 转换成纳秒处理
    workerThread = threadFactory.newThread(worker); // 创建工作线程
    leak = leakDetection || !workerThread.isDaemon() ? leakDetector.track(this) : null; // 是否开启内存泄漏检测
   this.maxPendingTimeouts = maxPendingTimeouts; // 最大允许等待任务数，HashedWheelTimer 中任务超出该
阈值时会抛出异常
}
```



时间轮空推进问题

Netty 中的时间轮是通过固定的时间间隔 tickDuration 进行推动的，如果长时间没有到期任务，那么会存在时间轮空推进的现象，从而造成一定的性能损耗。此外，如果任务的到期时间跨度很大，例如 A 任务 1s 后执行，B 任务 6 小时之后执行，也会造成空推进的问题。

Kafka解决方案

为了解决空推进的问题，Kafka 借助 JDK 的 DelayQueue 来负责推进时间轮。DelayQueue 保存了时间轮中的每

个 Bucket，并且根据 Bucket 的到期时间进行排序，最近的到期时间被放在 DelayQueue 的队头。Kafka 中会有一个线程来读取 DelayQueue 中的任务列表，如果时间没有到，那么 DelayQueue 会一直处于阻塞状态，从而解决空推进的问题。虽然DelayQueue 插入和删除的性能不是很好，但这其实就是一种权衡的策略，但是DelayQueue 只存放了 Bucket，Bucket 的数量并不多，相比空推进带来的影响是利大于弊的。

为了解决任务时间跨度很大的问题，Kafka 引入了层级时间轮，如下图所示。当任务的 deadline 超出当前所在层的时间轮表示范围时，就会尝试将任务添加到上一层时间轮中，跟钟表的时针、分针、秒针的转动规则是同一个道理。

4、select、poll、epoll的区别

select （windows）**poll **(linux)本质上和select没有区别，查询每个fd对应的设备状态，如果设备就绪则在设备等待队列中加入一项并继续遍历，如果遍历完所有fd后没有发现就绪设备，则挂起当前进程，直到设备就绪或者主动超时，被唤醒后它又要再次遍历fd。

**epoll **支持水平触发和边缘触发，最大的特点在于边缘触发，它只告诉进程哪些fd刚刚变为就绪态，并且只会通知一次。还有一个特点是，epoll使用“事件”的就绪通知方式，通过epollctl注册fd，一旦该fd就绪，内核就会采用类似callback的回调机制来激活该fd，epollwait便可以收到通知。

Epoll空轮询漏洞

在 JDK 中， Epoll 的实现是存在漏洞的，即使 Selector 轮询的事件列表为空，NIO 线程一样可以被唤醒，导致 CPU 100% 占用。实际上 Netty 并没有从根源上解决该问题，而是巧妙地规避了这个问题。

```
long time = System.nanoTime();
if (/*事件轮询的持续时间大于等于 timeoutMillis*/) {
    selectCnt = 1;
} else if (/*不正常的次数 selectCnt 达到阈值 512*/) {
    //重建Select并且SelectionKey重新注册到新Selector上
    selector = selectRebuildSelector(selectCnt);
}
```

NioEventLoop 线程的可靠性至关重要，一旦 NioEventLoop 发生阻塞或者陷入空轮询，就会导致整个系统不可用。

LEETCODE

Python语法

```
reduce(function, iterable[, initializer])
    reduce(lambda x,y:x * y,ns) # 数组之乘积 (ns[0] * ns[1]) * ns[2]
    reduce(lambda x,y:x + y,ns) # 数组之和
# 记忆化搜索
@functools.lru_cache(None)
res = helper(0,N,0)
helper.cache_clear()
tuple(ns) 可以hash做参数
# 大根堆
q = list(map(lambda x:-x,ns))
heapq.heapify(q)
key = -heapq.heappop(q)
# 过滤函数
filter(function, iterable)
    filter(lambda x: 2 < x < 10 and x % 2 == 0, range(18))
    filter(dfs, range(len(graph)))
# 除数
div, mod = divmod(sum(ns), 4)
random.randint(i,len(self.ns)-1)
#第一个降序，第二个升序
sorted(pss,key = lambda x:[x[0],-x[1]])

# 不可变str 常见函数
split(sep=None, maxsplit=-1)  # 以sep来分割字符串
strip([chars])  # 去除首末两端的字符, 默认是 \r,\n," "
join(iterable)  # 将iterable内的元素拼接成字符串,如','.join(['leet', 'code'])="leet,code"
replace(old, new[, count])  # 字符串替换, old to new
count(sub[, start[, end]])  # 统计子字符串sub的个数
startswith(prefix[, start[, end]])  # 以prefix开始的字符串
```

```
endswith(suffix[, start[, end]]) # 以suffix结束的字符串
cs in chrs: # chrs 中包含 cs

# deque 常见函数
queue = deque([iterable[, maxlen]])
queue.append(val) # 往右边添加一个元素
queue.appendleft(val) # 往左边添加一个元素
queue.clear() # 清空队列
queue.count(val) # 返回指定元素的出现次数
queue.insert(val[, start[, stop]]) # 在指定位置插入元素
queue.pop() # 获取最右边一个元素，并在队列中删除
queue.popleft() # 获取最左边一个元素，并在队列中删除
queue.reverse() # 队列反转
queue.remove(val) # 删除指定元素
queue.rotate(n=1) # 把右边元素放到左边

# list 常见函数
lst.sort(*, key=None, reverse=False)
lst.append(val) # 也可以 lst = lst + [val]
lst.clear() # 清空列表
lst.count(val) # val个数
lst.pop(val=lst[-1]) # (默认)从末端移除一个值
lst.remove(val) # 移除 val
lst.reverse() # 反转
lst.insert(i, val) # 在 i 处插入 val

# 字典dict 常见函数
d = defaultdict(lambda : value) # 取到不存在的值时不会报错，用{}时、需要设置get的default值
pop(key[, default]) # 通过键去删除键值对(若没有该键则返回default(没有设置default则报错))
setdefault(key[, default]) # 设置默认值
update([other]) # 批量添加
get(key[, default]) # 通过键获取值(若没有该键可设置默认值, 预防报错)
clear() # 清空字典
keys() # 将字典的键组成新的可迭代对象
values() # 将字典中的值组成新的可迭代对象
items() # 将字典的键值对凑成一个个元组, 组成新的可迭代对象
dict1 = dict2 #两个字典完全相等，滑窗时可用

# 集合set 常见函数
s = set(lambda : value)
add(elem) # 向集合中添加数据
update(*others) # 迭代着增加
clear() # 清空集合
discard(elem) # 删除集合中指定的值(不存在则不删除)

# 堆heapq 常见函数
heap = [] # 建堆
```

```
heapq.heappush(heap,item) # 往堆中插入新值
heapq.heappop(heap) # 弹出最小的值
heap[0] # 查看堆中最小的值, 不弹出
heapq.heapify(x) # 以线性时间将一个列表转为堆
heapq.heappoppush(heap, item) # 弹出最小的值.并且将新的值插入其中.
heapq.merge(*iterables, key=None, reverse=False) # 将多个堆进行合并
heapq.nlargest(n, iterable, key=None) # 从堆中找出最大的 n 个数，key的作用和sorted( )方法里面的key类似, 用列
表元素的某个属性和函数作为关键字
heapq.nsmallest(n, iterable, key=None) # 从堆中找出最小的 n 个数, 与 nlargest 相反


# 二分查找函数
bisect.bisect_left(ps, T, L=0, R=len(ns)) #二分左边界
bisect.bisect_right(ps, T, L=0, R=len(ns)) #二分右边界
bisect.insort_left(a, x, lo=0, hi=len(a)) # 二分插入到左侧
bisect.insort_right(a, x, lo=0, hi=len(a)) # 二分插入到右侧

# bit操作
& 符号，x & y ，会将两个十进制数在二进制下进行与运算
| 符号，x | y ，会将两个十进制数在二进制下进行或运算
^ 符号，x ^ y ，会将两个十进制数在二进制下进行异或运算
<< 符号，x << y 左移操作，最右边用 0 填充
>> 符号，x >> y 右移操作，最左边用 0 填充
~ 符号，~x ，按位取反操作，将 x 在二进制下的每一位取反

# 整数集合set位运算
# 整数集合做标志时，可以做参数加速运算
vstd 访问 i： vstd | (1 << i)
vstd 离开 i： vstd & ~(1 << i)
vstd 不包含 i : not vstd & (1 << i)

并集： A | B
交集： A & B
全集： (1 << n) - 1
补集： ((1 << n) - 1) ^ A
子集： (A & B) == B
判断是否是 2 的幂： A & (A - 1) == 0
最低位的 1 变为 0： n &= (n - 1)
    while n:
        n &= n - 1
        ret += 1
最低位的 1：A & (-A)，最低位的 1 一般记为 lowbit(A)

# ^    ：匹配字符串开头
# [\+\-]：代表一个+字符或-字符
# ?    ：前面一个字符可有可无
# \d   ：一个数字
```

```
# +    ：前面一个字符的一个或多个
# \D   ：一个非数字字符
# *    ：前面一个字符的0个或多个
matches = re.match('[ ]*([+−]?\d+)', s)
```

背包模板

「力扣」上的 0-1 背包问题

- 组合问题模板

```
#0−1背包，不可重复
for n in ns:
  for i in range(T, n−1, −1):
      dp[i] = max(dp[i], dp[i − n] + ws[i])
#完全背包，可重复，无序，算重量
for n in ns:
  for i in range(n, T+1):
      dp[i] = max(dp[i], dp[i − n] + ws[i])
#完全背包，可重复，有序，算次数
for i in range(1, T+1):
  for n in ns:
    dp[i] += dp[i−n]
```

✅377 组合总和 Ⅳ ✅494 目标和 ✅518 零钱兑换 II

- True、False问题

```
dp[i] |= dp[i−num]
```

✅139 单词拆分 ✅416 分割等和子集

```
#特殊的可以使用bit数组
```

- 最大最小问题：

```
dp[i] = min(dp[i], dp[i−num]+1)
dp[i] = max(dp[i], dp[i−num]+1)
```

✅474 一和零 ✅322零钱兑换

「力扣」第 879 题：盈利计划（困难）；「力扣」第 1449 题：数位成本和为目标值的最大数字（困难）。

回溯模板

```
# 回溯算法，复杂度较高2^n或者N！，因为回溯算法就是暴力穷举，可用lru剪枝
@functools.lru_cache(None)
def backtrack(路径, 选择列表):
    if 满足结束条件:
        结果.append(路径)
        return
    for 选择 in 选择列表:   # 核心代码段
        if vst[i]:  # 辅助数组，减枝
          continue
        做出选择
        递归执行backtrack
        撤销选择
```

「**剪枝**」第 46 题 全排列 第 47 题 全排列②

```
# 剪枝
def backtrack(temp_list, length):
    if length == n:
      res.append(temp_list)
    for i in range(n):
      if not visited[i]:
        visited[i] = 1
        backtrack(temp_list + [nums[i]], length + 1)
        visited[i] = 0
```

「**索引遍历**」第 78 题 子集 | 第 47 题 子集② | 第 131 题 分割字符串

```
第 **39 **题 组合 | 第 **40** 题 组合② | 第 **216** 题 组合③
```

```
# 索引遍历
def helper1(idx, n, temp_list):
  if temp_list not in res:
    res.append(temp_list)
  for i in range(idx, n):
    helper1(i + 1, n, temp_list + [nums[i]])
```

「 **资源消耗** 」第 22 题 夸号生成

```
# 资源消耗
def backtrack(S, L, R):
  if not L and not R:
    ans.append(''.join(S))
    return
  if L :     backtrack(S + ['('], L−1, R)
  if R > L : backtrack(S + [')'], L, R−1)
```

「 **资源消耗** 」第 93 题 复原IP

```
资源消耗
def backtrack(i, tmp, flag):
  if i == n and flag == 0:
    res.append(tmp[:−1])
  elif i<n and s[i] == '0':
    backtrack(i + 1, tmp + s[i] + ".", flag − 1)
  elif flag :
    for j in range(i, min(n,i + 3)):
      if 0 < int(s[i:j + 1]) <= 255:
        backtrack(j + 1, tmp + s[i:j + 1] + ".", flag − 1)
```

「 **资源消耗** 」第 17 题 电话号码

```
# 资源消耗
def dfs(path, remains):
  if not remains:
    res.append(path[:])
    return
  for i in range(len(remains)):
    dfs(path + [remains[i]], remains[:i] + remains[i+1:])

# 套模板
def dfs(pth,idx):
    if idx == len(ds):
        res.append(pth)
        return
    for c in dic[ds[idx]]:
        dfs(pth + c, idx + 1)
```

「**多重限制**」第 37 题 解数独 | 第 51 题 N皇后

```
# 多重限制
def backtrack(pos):
  if pos == n:
    return True
  i, j = empty[pos]
  for num in row[i] & col[j] & block[bidx(i, j)]:
    row[i].remove(num)
    col[j].remove(num)
    block[bidx(i, j)].remove(num)
    board[i][j] = str(num)
    if backtrack(pos + 1): return True
    row[i].add(num)
    col[j].add(num)
    block[bidx(i, j)].add(num)
```

「**递归**」第 10 题 正则匹配

```
# 递归
def isMatch(self, s: str, p: str) -> bool:
  if not p:
    return not s
  f = bool(s and p[0] in {s[0],'.'})
  if len(p) >= 2 and p[1] == "*":
    return self.isMatch(s, p[2:]) or f and self.isMatch(s[1:], p)
  else:
    return f and self.isMatch(s[1:], p[1:])
```

并查集模板

```
#虚拟节点用以连接某一特征的全部节点，类似于链表的preHead
dummy
parent = {}
size = collections.defaultdict(lambda:1)
cnt = 0
def find(x):
    parent.setdefault(x,x)
    while x != parent[x]:
        x = parent[x]
        #路径压缩 parent[x] = parent[parent[x]];
    return x
def union(x,y):
    nonlocal cnt
    if connected(x,y): return
    # 小的树挂到大的树上， 使树尽量平衡
    xP = find(x)
    yP = find(y)
    if size[hP] < size[yP]:
        parent[xP] = yP
    else:
        parent[yP] = xP
    size[xP] += size[yP]
    # 优化结束
    parent[find(x)] = find(y)
    # 不优化
    cnt -= 1
    return size[xP]
def connected(x, y):
    return find(x) == find(y)
def add(self,x):
    if x not in parent:
        parent[x] = None
        cnt += 1
# 检查是否有环
for a, b in edges:
    if connected(a, b):
        return True
    union(a, b)
# 将每个集合组成以头为key的字典
res = collections.defaultdict(list)
for e in e2n:
    res[uf.find(e)].append(e)
```

拓扑排序模板

```
# 【拓扑排序模板】
ins = [0] * n
ous = collections.defaultdict(list)
for cur, pre in ps:
    ins[cur] += 1          #入度
    ous[pre].append(cur)    #出度
res = list(filter(lambda x:ins[x]==0, range(n)))
q = collections.deque(res)
while q:
    pre = q.popleft()
    for cur in ous[pre]:   #释放出度队列
        ins[cur] -= 1
        if not ins[cur]:
            q.append(cur)  #入度为0解锁
            res.append(cur)
```

单调栈模板

```
# s中一般存索引
for i in range(len(ns):
    while stack and ns[stack[-1]] <= ns[i]: # 单调递减栈
        stack.pop()
    # 业务逻辑
    stack.append(i)
```

「**单调递增**」第 84 题 求最大矩形

```
# 第 **84** 题 求最大矩形
for i in range(len(hs)):
    while s and hs[i] < hs[s[-1]]:
        base = s.pop()
        if s:
            H = hs[base]
            W = i - s[-1] - 1 # 当前弹出的做高，当前与次小做宽
            res = max(res, H * W)
    s.append(i)
```

「**单调递增,考虑剩余**」第 316 题 去除重复字符

```
# 第 **316** 题 去除重复字符
for i,c in enumerate(ss):
    if c not in s:
        while s and c < s[-1] and s[-1] in ss[i:]:
            s.pop()
        s.append(c)
```

「**单调递减**」第 42 题 接雨水

```
# 第 **42** 题 接雨水
for i in range(len(hgt)):
    while stack and hgt[i] > hgt[stack[-1]]: #递减栈
        base = stack.pop()
        if stack:
            LH = hgt[stack[-1]]
            W = i - stack[-1] - 1
            H = min(LH,hgt[i]) - hgt[base]
            res += W * H
    stack.append(i)
```

「**单调递减**」第 739 题 每日温度

```
# 第 **739** 题 每日温度
for i in range(len(T)-1,-1,-1):
    while s and T[s[-1]] <= T[i] :  #递减栈
        s.pop()
    res[i] = s[-1] - i if s else 0
    s.append(i)
```

二分模板

```
# 1355579 T=5 => 13(5)55579 返回2
# ps[i-1] < ps[i] <= ps[i+1]
bisect.bisect_left(ps, T, L=0, R=len(ns))
# 1355579 T=5 => 13555(5)79 返回5
# ps[i-1] <= ps[i] < ps[i+1]
bisect.bisect_right(ps, T, L=0, R=len(ns))
bisect.bisect(ps, T, L=0, R=len(ns))
```

「 **中位返回** 」第 33 题 搜索旋转排序数组 | 第374题 猜数字大小 | 第69题 x平方根

```
# 中位返回
while L <= R:
    M = (L + R) // 2
    if nums[M] == T:
        return M
    elif nums[M] < T:
        L = M + 1
    else:
        R = M − 1
```

「 **区域压缩** 」第278题 第一个错误版本| 第162题 寻找峰值 | 第153题 寻找数组最小值

```
# 区域压缩
while L < R:
    M = (L + R) // 2
    if need in s[L:M]:
        R = M
    else:
        L = M + 1
```

动态规划模板

「 单串问题 」

- 70 爬楼梯问题
- 801 使序列递增的最小交换次数
- 746 使用最小花费爬楼梯
- 300 最长上升子序列

```
# 依赖前单个元素
dp[i] = dp[i−1] + ns[i]
# 依赖前部区域元素
for i in range(n)
    for j in range(i)
        dp[i] = min(dp[i], f(dp[j])
```

「单串加状态问题」

- 887 鸡蛋掉落

```
# 鸡蛋掉落
while cur[K] < N:           # 还剩 j 个蛋 测 ans 次 覆盖多少层
  for j in range(1, K + 1): # 覆盖总层数 碎了 −1 次层数 + 1 + 没碎 −1 次层数
      cur[j] = prev[j − 1] + 1 + prev[j]
  ans += 1
  prev = copy.deepcopy(cur)
```

- 813 最大平均值分组

```
# 813 最大平均值分组
for k in range(K−1):          #循环k次
  for i in range(N):          #每次均依赖上次的结果
      for j in range(i+1, N):
          dp[i] = max(dp[i], avrg(i, j) + dp[j])
```

- 410 分割数组最大值

```
# 410 分割数组最大值
for k in range(1,K):
  for i in range(N):
      for j in range(i):
          # 0~i中分 k 段最大 即为
          # 0~j中分k−1段最大 和 j到i的前缀和的最大
          dp[i][k] = min(dp[i][k], max(dp[j][k−1], ps[i+1] − ps[j+1]))
```

「经典双串LCS问题」

```
# 经典双串LCS问题
dp = [[0] * (M+1) for _ in range(N+1)]
for i in range(N):
    for j in range(M):
        if t1[i] == t2[j] : dp[i+1][j+1] = dp[i][j] + 1
        else : dp[i+1][j+1] = max(dp[i][j+1],dp[i+1][j])
```

「 区间动态规划 」

- 5 最长回文子串
- 647 最多回文子串
- 516 最长回文子序列
- 1312 最长回文插入次数

```
# dp[i][j] 代表从 i 到 j 的最长子串满足条件的数量
# i-- < j++  ==> i 在 0~j 范围内 --
dp = [[0] * (N) for _ in range(N)]
for j in range(N):
    dp[j][j] = 1
    for i in range(j-1,-1,-1):
        if ss[i] == ss[j]:
            dp[i][j] = dp[i+1][j-1] +2
        else :
            dp[i][j] = max(dp[i+1][j],dp[i][j-1])
```

「 区间分治动态规划 」

486 预测赢家
312 戳气球
664 奇怪的打印机
546 移除盒子

```
# 区间分治动态规划
def helper(self, ns: List[int]) :
    N = len(ns)
    dp = [[0] * N for _ in range(N+1)]
    for l in range(N): # 长度从小到大
        for i in range(N-l): # 以 i 为 开头
            j = i + l           # 以 j 为 终点
            for k in range(i,j): # 以 k 为分割点，进行分治
                // Todo 业务逻辑
```

「 卡特兰数 」

```
# 卡特兰数
g(n) = g(0)*g(n-1) + g(1)*g(n-2) ...g(n-1)*g(0)
dp=[1] + [0] * n
for i in range(1,n+1):
    for j in range(1,i+1):
        dp[i] += dp[j-1] * dp[i-j]
```

滑动窗口

```
"""给定待查串s和目标串t"""
nd, wd = {}, {}
nd = collections.Counter(s1)
L, R = 0, 0
cnt = 0 # 满足条件个数
while R < len(s):        # 窗口右边界不断扩大，本质是搜索问题的可能解
    c = s[R]      # 即将加入到窗口中的字符
    R += 1
    更新窗口中的数据
    while 满足窗口收缩条件：  # 窗口的左边界收缩，本质是优化可行解
        记录或返回结果
        d = s[L]   # 即将从窗口中删除的字符
        L += 1
        更新窗口中的数据
return 结果

# 固定窗口 ,比滑动窗口更快一些
i = j = cnt = 0
for j in range(len(A)):
    if A[j] == 0:
        cnt += 1
    if cnt > K: #不满足时 平移
        if A[i] == 0:
            cnt -= 1
        i += 1
return j - i + 1

for j in range(len(A)):
    if A[j] == 0:
        cnt += 1
    while cnt > K:
        if A[i] == 0:
            cnt -= 1
        i += 1
    res = max(res, j - i + 1)
return res
```

前缀和

「累加和存位置」

1371 最长偶数元音子数组

525 最长相等01子数组

325 最长和为k 子数组

```
# 前缀和初始化
psd = {0: -1}
for i in range(len(s)):
    t ^= cd.get(s[i], 0) # 业务逻辑
    if t not in psd:
        psd[t] = i      # 第一次存入数组
    else:
        ans = max(ans, i - psd[t]) #已存入则开始计算
```

「累加和存数量」

560 和为K的子数组数量

统计优美子数组

```
# 累加和存数量
psd = {0:1}
for i in range(len(ns)):
    s += ns[i]
    if s - T in psd:
        ans += psd[s - T] # 存数量
    psd[s] = psd.get(s,0) + 1
```

「模K状态前缀和」

523 连续和为 k 倍 的子数组（存索引）

974 和被k 整除 子数组数量（存数量）

```
# 模K状态前缀和
psd = {0:-1}
ans = s = 0
for i in range(len(ns)):
    s += ns[i]              # 业务逻辑
    if T != 0: s %= abs(T)  # 模k状态做key，索引做值
    if s not in psd:
        psd[s] = i
    elif i - psd[s] > 1:
        return True
```

「矩阵前缀和」

- 363 不超过K的最大数值和
- 1074 和为目标值的子矩阵数量

```
# 矩阵前缀和
for i in range(m):        #固定左边界
    ps = [0] * n
    for j in range(i, m): #固定右边界
        psS = 0
        dct = {0:1}        #初始只有一种可能
        for k in range(n): # 以高做前缀和
            ps[k] += mtx[j][k]          # 每行前缀和
            psS += ps[k]                # n行前缀和
            cnt += dct.get(psS - T, 0)  # 满足条件cnt
            dct[psS] = dct.get(psS,0) + 1 # 保存当前状态
return cnt
```

双指针

```
# 双指针
def removeElement(self, ns: List[int], val: int) -> int:
    slow = 0
    n = len(ns)
    for fast in range(n):
        if ns[fast] != val:
            ns[slow] = ns[fast]
            slow += 1
    return slow
```

深度优先

「二叉树遍历模板」

```
# 递归
# 时间复杂度：O(n)，n为节点数，访问每个节点恰好一次。
# 空间复杂度：空间复杂度：O(h)，h为树的高度。最坏情况下需要空间O(n)，平均情况为O(logn)

# 递归1：二叉树遍历最易理解和实现版本
class Solution:
    def preOrd(self, root: TreeNode) -> List[int]:
        if not root:
            return []
        # 前序递归
        return [root.val] + self.preOrd(root.left) + self.preOrd(root.right)
        # # 中序递归
        # return self.inOrd(root.left) + [root.val] + self.inOrd(root.right)
        # # 后序递归
        # return self.postOrd(root.left) + self.postOrd(root.right) + [root.val]

# 递归2：通用模板，可以适应不同的题目，添加参数、增加返回条件、修改进入递归条件、自定义返回值
class Solution:
    def preOrd(self, root: TreeNode) -> List[int]:
        def dfs(cur):
            if not cur:
                return
            # 前序递归
            res.append(cur.val)
            dfs(cur.left)
            dfs(cur.right)
            # # 中序递归
            # dfs(cur.left)
            # res.append(cur.val)
            # dfs(cur.right)
            # # 后序递归
            # dfs(cur.left)
            # dfs(cur.right)
            # res.append(cur.val)
        res = []
        dfs(root)
        return res


# 迭代
# 时间复杂度：O(n)，n为节点数，访问每个节点恰好一次。
```

```
# 空间复杂度：O(h)，h为树的高度。取决于树的结构，最坏情况存储整棵树，即O(n)
# 迭代1：前序遍历最常用模板（后序同样可以用）
class Solution:
    def preOrd(self, root: TreeNode) -> List[int]:
        if not root:
            return []
        res = []
        stack = [root]
        # # 前序迭代模板：最常用的二叉树DFS迭代遍历模板
        while stack:
            cur = stack.pop()
            res.append(cur.val)
            if cur.right:
                stack.append(cur.right)
            if cur.left:
                stack.append(cur.left)
        return res

        # # 后序迭代，相同模板：将前序迭代进栈顺序稍作修改，最后得到的结果反转
        # while stack:
        #     cur = stack.pop()
        #     if cur.left:
        #         stack.append(cur.left)
        #     if cur.right:
        #         stack.append(cur.right)
        #     res.append(cur.val)
        # return res[::-1]

# 迭代1：层序遍历最常用模板
class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        if not root:
            return []
        q = deque([root])
        res = []
        while q :
            l = []
            for i in range(len(q)) :
                t = q.popleft()
                l.append(t.val)
                if t.left : q.append(t.left)
                if t.right : q.append(t.right)
            res.append(l)
        return res


# 迭代2：前、中、后序遍历通用模板（只需一个栈的空间）
```

```
class Solution:
    def inOrd(self, root: TreeNode) -> List[int]:
        res = []
        stack = []
        cur = root
        # 中序，模板：先用指针找到每颗子树的最左下角，然后进行进出栈操作
        while stack or cur:
            while cur:
                stack.append(cur)
                cur = cur.left
            cur = stack.pop()
            res.append(cur.val)
            cur = cur.right
        return res

        # # 前序，相同模板
        # while stack or cur:
        #     while cur:
        #         res.append(cur.val)
        #         stack.append(cur)
        #         cur = cur.left
        #     cur = stack.pop()
        #     cur = cur.right
        # return res

        # # 后序，相同模板
        # while stack or cur:
        #     while cur:
        #         res.append(cur.val)
        #         stack.append(cur)
        #         cur = cur.right
        #     cur = stack.pop()
        #     cur = cur.left
        # return res[::-1]


# 迭代3：标记法迭代（需要双倍的空间来存储访问状态）：
# 前、中、后、层序通用模板，只需改变进栈顺序或即可实现前后中序遍历，
# 而层序遍历则使用队列先进先出。0表示当前未访问，1表示已访问。
class Solution:
    def preOrd(self, root: TreeNode) -> List[int]:
        res = []
        stack = [(0, root)]
        while stack:
            flag, cur = stack.pop()
            if not cur: continue
            if flag == 0:
```

```
            # 前序，标记法
            stack.append((0, cur.right))
            stack.append((0, cur.left))
            stack.append((1, cur))

            # # 后序，标记法
            # stack.append((1, cur))
            # stack.append((0, cur.right))
            # stack.append((0, cur.left))

            # # 中序，标记法
            # stack.append((0, cur.right))
            # stack.append((1, cur))
            # stack.append((0, cur.left))
        else:
            res.append(cur.val)
    return res

    # # 层序，标记法
    # res = []
    # queue = [(0, root)]
    # while queue:
    #     flag, cur = queue.pop(0)  # 注意是队列，先进先出
    #     if not cur: continue
    #     if flag == 0:
              # 层序遍历这三个的顺序无所谓，因为是队列，只弹出队首元素
    #         queue.append((1, cur))
    #         queue.append((0, cur.left))
    #         queue.append((0, cur.right))
    #     else:
    #         res.append(cur.val)
    # return res



# 莫里斯遍历
# 时间复杂度：O(n)，n为节点数，看似超过O(n)，有的节点可能要访问两次，实际分析还是O(n)
# 空间复杂度：O(1)，如果在遍历过程中就输出节点值，则只需常数空间就能得到中序遍历结果，空间只需两个指针。
# 如果将结果储存最后输出，则空间复杂度还是O(n)。

# PS：莫里斯遍历实际上是在原有二叉树的结构基础上，构造了线索二叉树，
#   线索二叉树定义为：原本为空的右子节点指向了中序遍历顺序之后的那个节点，把所有原本为空的左子节点都指向了中序遍历之前的那个节点

# 此处只给出中序遍历，前序遍历只需修改输出顺序即可
#   而后序遍历，由于遍历是从根开始的，而线索二叉树是将为空的左右子节点连接到相应的顺序上，使其能够按照相应准则输出
```

```
# 但是后序遍历的根节点却已经没有额外的空间来标记自己下一个应该访问的节点，
# 所以这里需要建立一个临时节点dump，令其左孩子是root。并且还需要一个子过程，就是倒序输出某两个节点之间路径上的各个节点。

# 莫里斯遍历，借助线索二叉树中序遍历（附前序遍历）
class Solution:
    def inOrd(self, root: TreeNode) -> List[int]:
        res = []
        # cur = pre = TreeNode(None)
        cur = root

        while cur:
            if not cur.left:
                res.append(cur.val)
                # print(cur.val)
                cur = cur.right
            else:
                pre = cur.left
                while pre.right and pre.right != cur:
                    pre = pre.right
                if not pre.right:
                    # print(cur.val) 这里是前序遍历的代码，前序与中序的唯一差别
                    pre.right = cur
                    cur = cur.left
                else:
                    pre.right = None
                    res.append(cur.val)
                    # print(cur.val)
                    cur = cur.right
        return res


# N叉树遍历
# 时间复杂度：时间复杂度：O(M)，其中 M 是 N 叉树中的节点个数。每个节点只会入栈和出栈各一次。
# 空间复杂度：O(M)。在最坏的情况下，这棵 N 叉树只有 2 层，所有第 2 层的节点都是根节点的孩子。
# 将根节点推出栈后，需要将这些节点都放入栈，共有 M⊠1个节点，因此栈的大小为 O(M)。


# N叉树简洁递归
class Solution:
    def preorder(self, root: 'Node') -> List[int]:
        if not root: return []
        res = [root.val]
        for node in root.children:
            res.extend(self.preorder(node))
        return res
```

```
# N叉树通用递归模板
class Solution:
    def preorder(self, root: 'Node') -> List[int]:
        res = []
        def helper(root):
            if not root:
                return
            res.append(root.val)
            for child in root.children:
                helper(child)
        helper(root)
        return res

# N叉树迭代方法
class Solution:
    def preorder(self, root: 'Node') -> List[int]:
        if not root:
            return []
        s = [root]
        # s.append(root)
        res = []
        while s:
            node = s.pop()
            res.append(node.val)
            # for child in node.children[::-1]:
            #     s.append(child)
            s.extend(node.children[::-1])
        return res
```

广度优先

```
# 「**无向图的遍历**」
q = collections.deque([i])
while q:
    cur = q.popleft()
    for nxt in dt[cur]:
        if not vst[nxt]:
            vstd[nxt] = True
            q.append(nxt)
```

```
# 「**二叉树层序遍历**」
q = deque([root])
res = []
while q :
    l = []
    for i in range(len(q)) :
        t = q.popleft()
        l.append(t.val)
        if t.left : q.append(t.left)
        if t.right : q.append(t.right)
    res.append(l)
return res
```

图论

```
#「Dijkstra最短路径」
dic = collections.defaultdict(list)
for u, v, w in edges:
    dic[u].append([v, w])
    dic[v].append([u, w])
q = [(0, n)]
dist = [-1] * (n + 1)
while q:
    dis, cur = heapq.heappop(q)
    if dist[cur] < 0:
        dist[cur] = dis
        for nxt, wi in dic[cur]:
            heapq.heappush(q, [dis + wi, nxt])
```

「 Floyd 求图中路径 」

```
# Floyd算法 求图中任意2点距离
ds = defaultdict(int)
st = set()
for i, (x, y) in enumerate(ess):
    ds[(x, y)] = vs[i]
    ds[(y, x)] = 1 / vs[i]
    st.update({x,y})
arr = list(st)
for k in arr:
    for i in arr:
        for j in arr:
            if ds[(i, k)] and ds[(k, j)]:
                ds[(i, j)] = ds[(i, k)] * ds[(k, j)]
```

实战算法篇

1. URL黑名单（布隆过滤器）

100亿黑名单URL，每个64B，问这个黑名单要怎么存？判断一个URL是否在黑名单中

散列表：

如果把黑名单看成一个集合，将其存在 hashmap 中，貌似太大了，需要 640G，明显不科学。

布隆过滤器：

它实际上是一个很长的二进制矢量和一系列随机映射函数。

它**可以用来判断一个元素是否在一个集合中**。它的优势是只需要占用很小的内存空间以及有着高效的查询效率。对于布隆过滤器而言，它的本质是一个**位数组**：位数组就是数组的每个元素都只占用1bit，并且每个元素只能是0或者1。

在数组中的每一位都是二进制位。布隆过滤器除了一个位数组，还有 K 个哈希函数。当一个元素加入布隆过滤器中的时候，会进行如下操作：

- 使用 K 个哈希函数对元素值进行 K 次计算，得到 K 个哈希值。
- 根据得到的哈希值，在位数组中把对应下标的值置为 1。

2. 词频统计（分文件）

2GB内存在20亿整数中找到出现次数最多的数

通常做法是使用哈希表对出现的每一个数做词频统计，哈希表的key是某个整数，value记录整数出现的次数。本题的数据量是20亿，有可能一个数出现20亿次，则为了避免溢出，哈希表的key是32位（4B）,value也是 32位（4B），那么一条哈希表的记录就需要占用8B。

当哈希表记录数为2亿个时，需要16亿个字节数（8*2亿），需要至少1.6GB内存(16亿/2^30,1GB== 2 ^30个字节 == 10亿)。则20亿个记录，至少需要16GB的内存，不符合题目要求。

解决办法是将20亿个数的大文件利用哈希函数分成16个小文件，根据哈希函数可以把20亿条数据均匀分布到16个文

件上，同一种数不可能被哈希函数分到不同的小文件上，假设哈希函数够好。然后对每一个小文件用哈希函数来统计其中每种数出现的次数，这样我们就得到16个文件中出现次数最多的数，接着从16个数中选出次数最大的那个key即可。

3. 未出现的数（bit数组）

40亿个非负整数中找到没有出现的数

对于原问题，如果使用哈希表来保存出现过的数，那么最坏情况下是40亿个数都不相同，那么哈希表则需要保存40亿条数据，一个32位整数需要4B，那么40亿*4B = 160亿个字节，一般大概10亿个字节的数据需要1G的空间，那么大概需要16G的空间，这不符合要求。

我们换一种方式，申请一个bit数组，数组大小为4294967295，大概为40亿bit，40亿/8 = 5亿字节，那么需要0.5G空间， bit数组的每个位置有两种状态0和1，那么怎么使用这个bit数组呢？呵呵，数组的长度刚好满足我们整数的个数范围，那么数组的每个下标值对应4294967295中的一个数，逐个遍历40亿个无符号数，例如，遇到20，则bitArray[20] = 1；遇到666，则bitArray[666] = 1,遍历完所有的数，将数组相应位置变为1。

40亿个非负整数中找到一个没有出现的数，内存限制10MB

10亿个字节的数据大概需要1GB空间处理，那么10MB内存换算过来就是可以处理1千万字节的数据，也就是8千万bit，对于40亿非负整数如果申请bit数组的话，40亿bit / 0.8亿bit = 50，那么这样最少也得分50块来处理，下面就以64块来进行分析解答吧。

总结一下进阶的解法：

1. 根据10MB的内存限制，确定统计区间的大小，就是第二次遍历时的bitArr大小。
2. 利用区间计数的方式，找到那个计数不足的区间，这个区间上肯定有没出现的数。
3. 对这个区间上的数做bit map映射，再遍历bit map，找到一个没出现的数即可。

自己的想法

如果只是找一个数，可以高位模运算，写到64个不同的文件，然后在最小的文件中通过bitArray一次处理掉。

40亿个无符号整数，1GB内存，找到所有出现两次的数

对于原问题，可以用bit map的方式来表示数出现的情况。具体地说，是申请一个长度为4294967295×2的bit类型的数组bitArr，用2个位置表示一个数出现的词频，1B占用8个bit，所以长度为4294967295×2的bit类型的数组占用1GB空间。怎么使用这个bitArr数组呢？遍历这40亿个无符号数，如果初次遇到num，就把bitArr[num2 + 1]和bitArr[num2]设置为01，如果第二次遇到num，就把bitArr[num2+1]和bitArr[num2]设置为10，如果第三次遇到num，就把bitArr[num2+1]和bitArr[num2]设置为11。以后再遇到num，发现此时bitArr[num2+1]和bitArr[num2]已经被设置为11，就不再做任何设置。遍历完成后，再依次遍历bitArr，如果发现bitArr[i2+1]和bitArr[i2]设置为10，那么i 就是出

现了两次的数。

4. 重复URL（分机器）

找到100亿个URL中重复的URL

原问题的解法使用解决大数据问题的一种常规方法：把大文件通过哈希函数分配到机器，或者通过哈希函数把大文件拆成小文件。一直进行这种划分，直到划分的结果满足资源限制的要求。首先，你要向面试官询问在资源上的限制有哪些，包括内存、计算时间等要求。在明确了限制要求之后，可以将每条URL通过哈希函数分配到若干机器或者拆分成若干小文件，这里的“若干”由具体的资源限制来计算出精确的数量。

例如，将100亿字节的大文件通过哈希函数分配到100台机器上，然后每一台机器分别统计分给自己的URL中是否有重复的URL，同时哈希函数的性质决定了同一条URL不可能分给不同的机器；或者在单机上将大文件通过哈希函数拆成1000个小文件，对每一个小文件再利用哈希表遍历，找出重复的URL；或者在分给机器或拆完文件之后，进行排序，排序过后再看是否有重复的URL出现。总之，牢记一点，很多大数据问题都离不开分流，要么是哈希函数把大文件的内容分配给不同的机器，要么是哈希函数把大文件拆成小文件，然后处理每一个小数量的集合。

5. TOPK搜索（小根堆）

海量搜索词汇，找到最热TOP100词汇的方法

最开始还是用哈希分流的思路来处理，把包含百亿数据量的词汇文件分流到不同的机器上，具体多少台机器由面试官规定或者由更多的限制来决定。对每一台机器来说，如果分到的数据量依然很大，比如，内存不够或其他问题，可以再用哈希函数把每台机器的分流文件拆成更小的文件处理。

处理每一个小文件的时候，哈希表统计每种词及其词频，哈希表记录建立完成后，再遍历哈希表，遍历哈希表的过程中使用大小为100的小根堆来选出每一个小文件的top 100（整体未排序的top 100）。每一个小文件都有自己词频的小根堆（整体未排序的top 100），将小根堆里的词按照词频排序，就得到了每个小文件的排序后top 100。然后把各个小文件排序后的top 100进行外排序或者继续利用小根堆，就可以选出每台机器上的top 100。不同机器之间的top100再进行外排序或者继续利用小根堆，最终求出整个百亿数据量中的top 100。对于top K 的问题，除哈希函数分流和用哈希表做词频统计之外，还经常用堆结构和外排序的手段进行处理。

6. 中位数（单向二分查找）

10MB内存，找到100亿整数的中位数

①内存够：内存够还慌什么啊，直接把100亿个全部排序了，你用冒泡都可以...然后找到中间那个就可以了。但是你以

为面试官会给你内存? ?
②内存不够：题目说是整数，我们认为是带符号的int,所以4字节，占32位。

假设100亿个数字保存在一个大文件中，依次读一部分文件到内存(不超过内存的限制)，将每个数字用二进制表示，比较二进制的最高位(第32位，符号位，0是正，1是负)，如果数字的最高位为0，则将这个数字写入 file0文件中；如果最高位为 1，则将该数字写入file1文件中。

从而将100亿个数字分成了两个文件，假设 file0文件中有 60亿 个数字，file1文件中有 40亿 个数字。那么中位数就在 file0 文件中，并且是 file0 文件中所有数字排序之后的第 10亿 个数字。（file1中的数都是负数，file0中的数都是正数，也即这里一共只有40亿个负数，那么排序之后的第50亿个数一定位于file_0中）

现在，我们只需要处理 file0 文件了（不需要再考虑file1文件）。对于 file0 文件，同样采取上面的措施处理：将file0文件依次读一部分到内存(不超内存限制)，将每个数字用二进制表示，比较二进制的 次高位（第31位），如果数字的次高位为0，写入file00文件中；如果次高位为1，写入file01文件 中。

现假设 file00文件中有30亿个数字，file01中也有30亿个数字，则中位数就是：file00文件中的数字从小到大排序之后的第10亿个数字。

抛弃file01文件，继续对 file00文件 根据 次次高位(第30位) 划分，假设此次划分的两个文件为：file000中有5亿个数字，file001中有25亿个数字，那么中位数就是 file00_1文件中的所有数字排序之后的 第 5亿 个数。

按照上述思路，直到划分的文件可直接加载进内存时，就可以直接对数字进行快速排序，找出中位数了。

7. 短域名系统（缓存）

设计短域名系统，将长URL转化成短的URL.

（1）利用放号器，初始值为0，对于每一个短链接生成请求，都递增放号器的值，再将此值转换为62进制（a-zA-Z0-9），比如第一次请求时放号器的值为0，对应62进制为a，第二次请求时放号器的值为1，对应62进制为b，第10001次请求时放号器的值为10000，对应62进制为sBc。
（2）将短链接服务器域名与放号器的62进制值进行字符串连接，即为短链接的URL，比如：t.cn/sBc。
（3）重定向过程：生成短链接之后，需要存储短链接到长链接的映射关系，即sBc -> URL，浏览器访问短链接服务器时，根据URL Path取到原始的链接，然后进行302重定向。映射关系可使用K-V存储，比如Redis或Memcache。

8. 海量评论入库（消息队列）

假设有这么一个场景，有一条新闻，新闻的评论量可能很大，如何设计评论的读和写

前端页面直接给用户展示、通过消息队列异步方式入库

读可以进行读写分离、同时热点评论定时加载到缓存

9. 在线/并发用户数（Redis）

显示网站的用户在线数的解决思路

维护在线用户表

使用Redis统计

显示网站并发用户数

1. 每当用户访问服务时，把该用户的 ID 写入ZSORT队列，权重为当前时间
2. 根据权重(即时间)计算一分钟内该机构的用户数Zrange
3. 删掉一分钟以上过期的用户Zrem

10. 热门字符串（前缀树）

假设目前有 1000w 个记录（这些查询串的重复度比较高，虽然总数是 1000w，但如果除去重复后，则不超过 300w 个）。请统计最热门的 10 个查询串，要求使用的内存不能超过 1G。（一个查询串的重复度越高，说明查询它的用户越多，也就越热门。）

HashMap 法

虽然字符串总数比较多，但去重后不超过 300w，因此，可以考虑把所有字符串及出现次数保存在一个 HashMap 中，所占用的空间为 300w*(255+4)≈777M（其中，4 表示整数占用的 4 个字节）。由此可见，1G 的内存空间完全够用。

思路如下：

首先，遍历字符串，若不在 map 中，直接存入 map，value 记为 1；若在 map 中，则把对应的 value 加 1，这一步时间复杂度 O(N)。

接着遍历 map，构建一个 10 个元素的小顶堆，若遍历到的字符串的出现次数大于堆顶字符串的出现次数，则进行替换，并将堆调整为小顶堆。

遍历结束后，堆中 10 个字符串就是出现次数最多的字符串。这一步时间复杂度 O(Nlog10)。

前缀树法

当这些字符串有大量相同前缀时，可以考虑使用前缀树来统计字符串出现的次数，树的结点保存字符串出现次数，0

表示没有出现。

思路如下：
在遍历字符串时，在前缀树中查找，如果找到，则把结点中保存的字符串次数加 1，否则为这个字符串构建新结点，构建完成后把叶子结点中字符串的出现次数置为 1。

最后依然使用小顶堆来对字符串的出现次数进行排序。

11. 红包算法

线性切割法，一个区间切N-1刀。越早越多
二倍均值法，【 0 ~ 剩余金额 / 剩余人数 * 2 】中随机，相对均匀

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

名称	数据对象	稳定性	时间复杂度		额外空间复杂度	描述
			平均	最坏		
冒泡排序	数组	✓	$O(n^2)$		$O(1)$	（无序区，有序区）。 从无序区透过交换找出最大元素放到有序区前端。
选择排序	数组	✗	$O(n^2)$		$O(1)$	（有序区，无序区）。 在无序区里找一个最小的元素跟在有序区的后面。对数组：比较得多，换得少。
	链表	✓				
插入排序	数组、链表	✓	$O(n^2)$		$O(1)$	（有序区，无序区）。 把无序区的第一个元素插入到有序区的合适的位置。对数组：比较得少，换得多。
堆排序	数组	✗	$O(n \log n)$		$O(1)$	（最大堆，有序区）。 从堆顶把根卸出来放在有序区之前，再恢复堆。
归并排序	数组	✓	$O(n \log^2 n)$		$O(1)$	把数据分为两段，从两段中逐个选最小的元素移入新数据段的末尾。 可从上到下或从下到上进行。
			$O(n \log n)$		$O(n) + O(\log n)$ 如果不是从下到上	
	链表				$O(1)$	
快速排序	数组	✗	$O(n \log n)$	$O(n^2)$	$O(\log n)$	（小数，基准元素，大数）。 在区间中随机挑选一个元素作基准，将小于基准的元素放在基准之前，大于基准的元素放在基准之后，再分别对小数区与大数区进行排序。
希尔排序	数组	✗	$O(n \log^2 n)$	$O(n^2)$	$O(1)$	每一轮按照事先决定的间隔进行插入排序，间隔会依次缩小，最后一次一定要是1。
计数排序	数组、链表	✓	$O(n+m)$		$O(n+m)$	统计小于等于该元素值的元素的个数i，于是该元素就放在目标数组的索引i位（i≥0）。
桶排序	数组、链表	✓	$O(n)$		$O(m)$	将值为i的元素放入i号桶，最后依次把桶里的元素倒出来。
基数排序	数组、链表	✓	$O(k \times n)$	$O(n^2)$		一种多关键字的排序算法，可用桶排序实现。

12. 手写快排

```
public class QuickSort {
    public static void swap(int[] arr, int i, int j) {
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }
    /* 常规快排 */
    public static void quickSort1(int[] arr, int L , int R) {
        if (L > R)  return;
        int M = partition(arr, L, R);
        quickSort1(arr, L, M - 1);
        quickSort1(arr, M + 1, R);
    }
    public static int partition(int[] arr, int L, int R) {
        if (L > R) return -1;
        if (L == R) return L;
        int lessEqual = L - 1;
        int index = L;
        while (index < R) {
            if (arr[index] <= arr[R])
                swap(arr, index, ++lessEqual);
            index++;
        }
        swap(arr, ++lessEqual, R);
        return lessEqual;
    }
    /* 荷兰国旗 */
    public static void quickSort2(int[] arr, int L, int R) {
        if (L > R)  return;
        int[] equalArea = netherlandsFlag(arr, L, R);
        quickSort2(arr, L, equalArea[0] - 1);
        quickSort2(arr, equalArea[1] + 1, R);
    }
    public static int[] netherlandsFlag(int[] arr, int L, int R) {
        if (L > R) return new int[] { -1, -1 };
        if (L == R) return new int[] { L, R };
        int less = L - 1;
        int more = R;
        int index = L;
        while (index < more) {
            if (arr[index] == arr[R]) {
                index++;
            } else if (arr[index] < arr[R]) {
                swap(arr, index++, ++less);
```

```
            } else {
                swap(arr, index, --more);
            }
        }
        swap(arr, more, R);
        return new int[] { less + 1, more };
    }

    // for test
    public static void main(String[] args) {
        int testTime = 1;
        int maxSize = 10000000;
        int maxValue = 100000;
        boolean succeed = true;
        long T1=0,T2=0;
        for (int i = 0; i < testTime; i++) {
            int[] arr1 = generateRandomArray(maxSize, maxValue);
            int[] arr2 = copyArray(arr1);
            int[] arr3 = copyArray(arr1);
//          int[] arr1 = {9,8,7,6,5,4,3,2,1};
            long t1 = System.currentTimeMillis();
            quickSort1(arr1,0,arr1.length-1);
            long t2 = System.currentTimeMillis();
            quickSort2(arr2,0,arr2.length-1);
            long t3 = System.currentTimeMillis();
            T1 += (t2-t1);
            T2 += (t3-t2);
            if (!isEqual(arr1, arr2) || !isEqual(arr2, arr3)) {
                succeed = false;
                break;
            }
        }
        System.out.println(T1+" "+T2);
//      System.out.println(succeed ? "Nice!" : "Oops!");
    }

    private static int[] generateRandomArray(int maxSize, int maxValue) {
        int[] arr = new int[(int) ((maxSize + 1) * Math.random())];
        for (int i = 0; i < arr.length; i++) {
            arr[i] = (int) ((maxValue + 1) * Math.random())
                    - (int) (maxValue * Math.random());
        }
        return arr;
    }
    private static int[] copyArray(int[] arr) {
        if (arr == null)  return null;
        int[] res = new int[arr.length];
```

```
        for (int i = 0; i < arr.length; i++) {
            res[i] = arr[i];
        }
        return res;
    }
    private static boolean isEqual(int[] arr1, int[] arr2) {
        if ((arr1 == null && arr2 != null) || (arr1 != null && arr2 == null))
            return false;
        if (arr1 == null && arr2 == null)
            return true;
        if (arr1.length != arr2.length)
            return false;
        for (int i = 0; i < arr1.length; i++)
            if (arr1[i] != arr2[i])
                return false;
        return true;
    }
    private static void printArray(int[] arr) {
        if (arr == null)
            return;
        for (int i = 0; i < arr.length; i++)
            System.out.print(arr[i] + " ");
        System.out.println();
    }
}
```

13. 手写归并

```
public static void merge(int[] arr, int L, int M, int R) {
    int[] help = new int[R - L + 1];
    int i = 0;
    int p1 = L;
    int p2 = M + 1;
    while (p1 <= M && p2 <= R)
        help[i++] = arr[p1] <= arr[p2] ? arr[p1++] : arr[p2++];
    while (p1 <= M)
        help[i++] = arr[p1++];
    while (p2 <= R)
        help[i++] = arr[p2++];
    for (i = 0; i < help.length; i++)
        arr[L + i] = help[i];
}
```

```
public static void mergeSort(int[] arr, int L, int R) {
    if (L == R)
        return;
    int mid = L + ((R - L) >> 1);
    process(arr, L, mid);
    process(arr, mid + 1, R);
    merge(arr, L, mid, R);
}
public static void main(String[] args) {
    int[] arr1 = {9,8,7,6,5,4,3,2,1};
    mergeSort(arr, 0, arr.length - 1);
    printArray(arr);
}
```

14. 手写堆排

```
// 堆排序额外空间复杂度O(1)
public static void heapSort(int[] arr) {
    if (arr == null || arr.length < 2)
        return;
    for (int i = arr.length - 1; i >= 0; i--)
        heapify(arr, i, arr.length);
    int heapSize = arr.length;
    swap(arr, 0, --heapSize);
    // O(N*logN)
    while (heapSize > 0) { // O(N)
        heapify(arr, 0, heapSize); // O(logN)
        swap(arr, 0, --heapSize); // O(1)
    }
}
// arr[index]刚来的数，往上
public static void heapInsert(int[] arr, int index) {
    while (arr[index] > arr[(index - 1) / 2]) {
        swap(arr, index, (index - 1) / 2);
        index = (index - 1) / 2;
    }
}
// arr[index]位置的数，能否往下移动
public static void heapify(int[] arr, int index, int heapSize) {
    int left = index * 2 + 1; // 左孩子的下标
    while (left < heapSize) { // 下方还有孩子的时候
        // 两个孩子中，谁的值大，把下标给largest
```

```
            // 1 ) 只有左孩子，left -> largest
            // 2) 同时有左孩子和右孩子，右孩子的值<= 左孩子的值，left -> largest
            // 3) 同时有左孩子和右孩子并且右孩子的值> 左孩子的值， right -> largest
            int largest = left+1 < heapSize && arr[left+1]> arr[left] ? left+1 : left;
            // 父和较大的孩子之间，谁的值大，把下标给largest
            largest = arr[largest] > arr[index] ? largest : index;
            if (largest == index)
                break;
            swap(arr, largest, index);
            index = largest;
            left = index * 2 + 1;
        }
    }
    public static void swap(int[] arr, int i, int j) {
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }
    public static void main(String[] args) {
        int[] arr1 = {9,8,7,6,5,4,3,2,1};
        heapSort(arr1);
        printArray(arr1);
    }
```

15. 手写单例

```
public class Singleton {
        private volatile static Singleton singleton;
        private Singleton() {}
        public static Singleton getSingleton() {
        if (singleton == null) {
            synchronized (Singleton.class) {
            if (singleton == null) {
                singleton = new Singleton();
            }
        }
        }
        return singleton;
    }
}
```

16. 手写LRUcache

```
// 基于linkedHashMap
public class LRUCache {
    private LinkedHashMap<Integer,Integer> cache;
    private int capacity;   //容量大小
    public LRUCache(int capacity) {
        cache = new LinkedHashMap<>(capacity);
        this.capacity = capacity;
    }
    public int get(int key) {
        //缓存中不存在此key，直接返回
        if(!cache.containsKey(key)) {
            return -1;
        }
        int res = cache.get(key);
        cache.remove(key);   //先从链表中删除
        cache.put(key,res);  //再把该节点放到链表末尾处
        return res;
    }
    public void put(int key,int value) {
        if(cache.containsKey(key)) {
            cache.remove(key); //已经存在，在当前链表移除
        }
        if(capacity == cache.size()) {
            //cache已满，删除链表头位置
            Set<Integer> keySet = cache.keySet();
            Iterator<Integer> iterator = keySet.iterator();
            cache.remove(iterator.next());
        }
        cache.put(key,value);  //插入到链表末尾
    }
}
```

```
//手写双向链表
class LRUCache {
    class DNode {
        DNode prev;
        DNode next;
        int val;
        int key;
    }
    Map<Integer, DNode> map = new HashMap<>();
```

```
DNode head, tail;
int cap;
public LRUCache(int capacity) {
    head = new DNode();
    tail = new DNode();
    head.next = tail;
    tail.prev = head;
    cap = capacity;
}
public int get(int key) {
    if (map.containsKey(key)) {
        DNode node = map.get(key);
        removeNode(node);
        addToHead(node);
        return node.val;
    } else {
        return -1;
    }
}
public void put(int key, int value) {
    if (map.containsKey(key)) {
        DNode node = map.get(key);
        node.val = value;
        removeNode(node);
        addToHead(node);
    } else {
        DNode newNode = new DNode();
        newNode.val = value;
        newNode.key = key;
        addToHead(newNode);
        map.put(key, newNode);
        if (map.size() > cap) {
            map.remove(tail.prev.key);
            removeNode(tail.prev);
        }
    }
}
public void removeNode(DNode node) {
    DNode prevNode = node.prev;
    DNode nextNode = node.next;
    prevNode.next = nextNode;
    nextNode.prev = prevNode;
}
public void addToHead(DNode node) {
    DNode firstNode = head.next;
    head.next = node;
    node.prev = head;
```

```
        node.next = firstNode;
        firstNode.prev = node;
    }
}
```

17. 手写线程池

```
package com.concurrent.pool;
import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
public class MySelfThreadPool {
    //默认线程池中的线程的数量
    private static final int WORK_NUM = 5;
    //默认处理任务的数量
    private static final int TASK_NUM = 100;
    private int workNum;//线程数量
    private int taskNum;//任务数量
    private final Set<WorkThread> workThreads;//保存线程的集合
    private final BlockingQueue<Runnable> taskQueue;//阻塞有序队列存放任务
    public MySelfThreadPool() {
        this(WORK_NUM, TASK_NUM);
    }
    public MySelfThreadPool(int workNum, int taskNum) {
        if (workNum <= 0) workNum = WORK_NUM;
        if (taskNum <= 0) taskNum = TASK_NUM;
        taskQueue = new ArrayBlockingQueue<>(taskNum);
        this.workNum = workNum;
        this.taskNum = taskNum;
        workThreads = new HashSet<>();
        //启动一定数量的线程数，从队列中获取任务处理
        for (int i=0;i<workNum;i++) {
            WorkThread workThread = new WorkThread("thead_"+i);
            workThread.start();
            workThreads.add(workThread);
        }
    }
    public void execute(Runnable task) {
        try {
            taskQueue.put(task);
        } catch (InterruptedException e) {
```

```
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    public void destroy() {
        System.out.println("ready close thread pool...");
        if (workThreads == null || workThreads.isEmpty()) return ;
        for (WorkThread workThread : workThreads) {
            workThread.stopWork();
            workThread = null;//help gc
        }
        workThreads.clear();
    }
    private class WorkThread extends Thread{
        public WorkThread(String name) {
            super();
            setName(name);
        }
        @Override
        public void run() {
            while (!interrupted()) {
                try {
                    Runnable runnable = taskQueue.take();//获取任务
                    if (runnable !=null) {
                        System.out.println(getName()+" readyexecute:"+runnable.toString());
                        runnable.run();//执行任务
                    }
                    runnable = null;//help gc
                } catch (Exception e) {
                    interrupt();
                    e.printStackTrace();
                }
            }
        }
        public void stopWork() {
            interrupt();
        }
    }
}

package com.concurrent.pool;

public class TestMySelfThreadPool {
    private static final int TASK_NUM = 50;//任务的个数
    public static void main(String[] args) {
        MySelfThreadPool myPool = new MySelfThreadPool(3,50);
        for (int i=0;i<TASK_NUM;i++) {
```

```
                myPool.execute(new MyTask("task_"+i));
            }
        }
        static class MyTask implements Runnable{
            private String name;
            public MyTask(String name) {
                this.name = name;
            }
            public String getName() {
                return name;
            }
            public void setName(String name) {
                this.name = name;
            }
            @Override
            public void run() {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                System.out.println("task :"+name+" end...");
            }
            @Override
            public String toString() {
                // TODO Auto-generated method stub
                return "name = "+name;
            }
        }
}
```

18. 手写消费者生产者模式

```
public class Storage {
    private static int MAX_VALUE = 100;
    private List<Object> list = new ArrayList<>();
    public void produce(int num) {
        synchronized (list) {
            while (list.size() + num > MAX_VALUE) {
                System.out.println("暂时不能执行生产任务");
                try {
```

```
                list.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        for (int i = 0; i < num; i++) {
            list.add(new Object());
        }
        System.out.println("已生产产品数"+num+" 仓库容量"+list.size());
        list.notifyAll();
    }
}

public void consume(int num) {
    synchronized (list) {
        while (list.size() < num) {
            System.out.println("暂时不能执行消费任务");
            try {
                list.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        for (int i = 0; i < num; i++) {
            list.remove(0);
        }
        System.out.println("已消费产品数"+num+" 仓库容量" + list.size());
        list.notifyAll();
    }
}
}

public class Producer extends Thread {
    private int num;
    private Storage storage;
    public Producer(Storage storage) {
        this.storage = storage;
    }
    public void setNum(int num) {
        this.num = num;
    }
    public void run() {
        storage.produce(this.num);
    }
}

public class Customer extends Thread {
```

```
    private int num;
    private Storage storage;
    public Customer(Storage storage) {
        this.storage = storage;
    }
    public void setNum(int num) {
        this.num = num;
    }
    public void run() {
        storage.consume(this.num);
    }
}

public class Test {
    public static void main(String[] args) {
        Storage storage = new Storage();
        Producer p1 = new Producer(storage);
        Producer p2 = new Producer(storage);
        Producer p3 = new Producer(storage);
        Producer p4 = new Producer(storage);
        Customer c1 = new Customer(storage);
        Customer c2 = new Customer(storage);
        Customer c3 = new Customer(storage);
        p1.setNum(10);
        p2.setNum(20);
        p3.setNum(80);
        c1.setNum(50);
        c2.setNum(20);
        c3.setNum(20);
        c1.start();
        c2.start();
        c3.start();
        p1.start();
        p2.start();
        p3.start();
    }
}
```

19. 手写阻塞队列

```
public class blockQueue {
    private List<Integer> container = new ArrayList<>();
    private volatile int size;
```

```
    private volatile int capacity;
    private Lock lock = new ReentrantLock();
    private final Condition isNull = lock.newCondition();
    private final Condition isFull = lock.newCondition();
    blockQueue(int capacity) {
        this.capacity = capacity;
    }
    public void add(int data) {
        try {
            lock.lock();
            try {
                while (size >= capacity) {
                    System.out.println("阻塞队列满了");
                    isFull.await();
                }
            } catch (Exception e) {
                isFull.signal();
                e.printStackTrace();
            }
            ++size;
            container.add(data);
            isNull.signal();
        } finally {
            lock.unlock();
        }
    }
    public int take() {
        try {
            lock.lock();
            try {
                while (size == 0) {
                    System.out.println("阻塞队列空了");
                    isNull.await();
                }
            } catch (Exception e) {
                isNull.signal();
                e.printStackTrace();
            }
            --size;
            int res = container.get(0);
            container.remove(0);
            isFull.signal();
            return res;
        } finally {
            lock.unlock();
        }
    }
}
```

```
public static void main(String[] args) {
    AxinBlockQueue queue = new AxinBlockQueue(5);
    Thread t1 = new Thread(() -> {
        for (int i = 0; i < 100; i++) {
            queue.add(i);
            System.out.println("塞入" + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
    Thread t2 = new Thread(() -> {
        for (; ; ) {
            System.out.println("消费"+queue.take());
            try {
                Thread.sleep(800);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    });
    t1.start();
    t2.start();
}
```

20. 手写多线程交替打印ABC

```
package com.demo.test;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
public class syncPrinter implements Runnable{
    // 打印次数
    private static final int PRINT_COUNT = 10;
    private final ReentrantLock reentrantLock;
    private final Condition thisCondtion;
    private final Condition nextCondtion;
    private final char printChar;
     public syncPrinter(ReentrantLock reentrantLock, Condition thisCondtion, Condition nextCondition, char 
printChar) {
```

```
        this.reentrantLock = reentrantLock;
        this.nextCondtion = nextCondition;
        this.thisCondtion = thisCondtion;
        this.printChar = printChar;
    }
    @Override
    public void run() {
        // 获取打印锁 进入临界区
        reentrantLock.lock();
        try {
            // 连续打印PRINT_COUNT次
            for (int i = 0; i < PRINT_COUNT; i++) {
                //打印字符
                System.out.print(printChar);
                // 使用nextCondition唤醒下一个线程
                // 因为只有一个线程在等待，所以signal或者signalAll都可以
                nextCondtion.signal();
                // 不是最后一次则通过thisCondtion等待被唤醒
                // 必须要加判断，不然虽然能够打印10次，但10次后就会直接死锁
                if (i < PRINT_COUNT - 1) {
                    try {
                        // 本线程让出锁并等待唤醒
                        thisCondtion.await();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        } finally {
            reentrantLock.unlock();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        ReentrantLock lock = new ReentrantLock();
        Condition conditionA = lock.newCondition();
        Condition conditionB = lock.newCondition();
        Condition conditionC = lock.newCondition();
        Thread printA = new Thread(new syncPrinter(lock, conditionA, conditionB,'A'));
        Thread printB = new Thread(new syncPrinter(lock, conditionB, conditionC,'B'));
        Thread printC = new Thread(new syncPrinter(lock, conditionC, conditionA,'C'));
        printA.start();
        Thread.sleep(100);
        printB.start();
        Thread.sleep(100);
        printC.start();
    }
}
```

21. 交替打印FooBar

```
//手太阴肺经 BLOCKING Queue
public class FooBar {
    private int n;
    private BlockingQueue<Integer> bar = new LinkedBlockingQueue<>(1);
    private BlockingQueue<Integer> foo = new LinkedBlockingQueue<>(1);
    public FooBar(int n) {
        this.n = n;
    }
    public void foo(Runnable printFoo) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            foo.put(i);
            printFoo.run();
            bar.put(i);
        }
    }
    public void bar(Runnable printBar) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            bar.take();
            printBar.run();
            foo.take();
        }
    }
}

//手阳明大肠经CyclicBarrier 控制先后
class FooBar6 {
    private int n;
    public FooBar6(int n) {
        this.n = n;
    }
    CyclicBarrier cb = new CyclicBarrier(2);
    volatile boolean fin = true;
    public void foo(Runnable printFoo) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            while(!fin);
            printFoo.run();
            fin = false;
            try {
                cb.await();
            } catch (BrokenBarrierException e) {}
        }
    }
    public void bar(Runnable printBar) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            try {
```

```
                cb.await();
            } catch (BrokenBarrierException e) {}
            printBar.run();
            fin = true;
        }
    }
}

//手少阴心经 自旋 + 让出CPU
class FooBar5 {
    private int n;

    public FooBar5(int n) {
        this.n = n;
    }
    volatile boolean permitFoo = true;
    public void foo(Runnable printFoo) throws InterruptedException {
        for (int i = 0; i < n; ) {
            if(permitFoo) {
                printFoo.run();
                i++;
                permitFoo = false;
            }else{
                Thread.yield();
            }
        }
    }
    public void bar(Runnable printBar) throws InterruptedException {
        for (int i = 0; i < n; ) {
            if(!permitFoo) {
            printBar.run();
            i++;
            permitFoo = true;
            }else{
                Thread.yield();
            }
        }
    }
}



//手少阳三焦经 可重入锁 + Condition
class FooBar4 {
    private int n;

    public FooBar4(int n) {
        this.n = n;
    }
```

```
    Lock lock = new ReentrantLock(true);
    private final Condition foo = lock.newCondition();
    volatile boolean flag = true;
    public void foo(Runnable printFoo) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            lock.lock();
            try {
                while(!flag) {
                    foo.await();
                }
                printFoo.run();
                flag = false;
                foo.signal();
            }finally {
                lock.unlock();
            }
        }
    }

    public void bar(Runnable printBar) throws InterruptedException {
        for (int i = 0; i < n;i++) {
            lock.lock();
            try {
                while(flag) {
                    foo.await();
                }
                printBar.run();
                flag = true;
                foo.signal();
            }finally {
                lock.unlock();
            }
        }
    }
}

//手撅阴心包经 synchronized + 标志位 + 唤醒
class FooBar3 {
    private int n;
    // 标志位，控制执行顺序，true执行printFoo，false执行printBar
    private volatile boolean type = true;
    private final Object foo=  new Object(); // 锁标志

    public FooBar3(int n) {
        this.n = n;
    }
    public void foo(Runnable printFoo) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            synchronized (foo) {
```

```
                while(!type){
                    foo.wait();
                }
                printFoo.run();
                type = false;
                foo.notifyAll();
            }
        }
    }

    public void bar(Runnable printBar) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            synchronized (foo) {
                while(type){
                    foo.wait();
                }
                printBar.run();
                type = true;
                foo.notifyAll();
            }
        }
    }
}


//手太阳小肠经 信号量 适合控制顺序
class FooBar2 {
    private int n;
    private Semaphore foo = new Semaphore(1);
    private Semaphore bar = new Semaphore(0);
    public FooBar2(int n) {
        this.n = n;
    }

    public void foo(Runnable printFoo) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            foo.acquire();
            printFoo.run();
            bar.release();
        }
    }
    public void bar(Runnable printBar) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            bar.acquire();
            printBar.run();
            foo.release();
        }
    }
}
```

个人项目

一站到底

采用SpringBoot构建项目，主要通过分布式缓存、队列、限流保证系统高可用，Netty、缓存、反向代理保证高并发。

双人对战答题、公司对战抢答

1. 如何设计排行榜

- 个人总得分和总排名实时更新
- 个人排行榜按分数、时间、次数、正确率展示
- 日榜、过去N日榜滚动更新

性能优化过程

第一条需求很简单，使用了Redis的Zset实现不过这里总得分采用了基于分数、时间、次数和正确率的混合加权。考虑到数据的持久化，以及关系数据库和缓存的一致性导致的设计的复杂性，使用了谷歌开源的JamsRanking

优点是可以直接使用现成的setScores和getRanking接口封装了Redis和Mysql和消息队列的完成事务和一致性的使用细节。缺点是并发比较低使用Jmeter进行压测，单机只有20左右的TPS**

后来看了下源码，主要是它针对每一次设置都进行了分布式事务处理，并且会返回事务提交或回滚的结果。了解了底层实现以后就去谷歌的开源社区去查阅了相关的解决方案，当时官方对这个问题并没有通过配置能直接解决问题的快捷方式，不过推荐了使用者自身如果对响应时间不高的场景下可以采用批量合并事务的方式进行优化。基于这个思路，我们把写操作进行了封装并放入了队列，然后在消费者端批量取得数据后进行事务的批量处理，压测环境下整体性能达到了500TPS。已经基本满足了线上更新的需求，但是当时压测的过程中，队列偶尔的吞吐量会大范围波动，

经常会持续数十秒，然后业务一次性处理完再响应，导致局部响应时间大幅度增长

后来也是在官网上查询，了解到谷歌开源组件使用的队列服务底层是使用BigTable作为持久层，但是当BigTable分片过大时，会触发再分片的过程，再分片的过程中，是不会进行任务分发的，所以就会导致先前的问题。针对这个问题，谷歌官方的建议是提前配置队列的数量、负载策略和最大容量等信息，保证所有队列不同时触发再分片

进行两次优化后，压测环境已经基本可以满足预期了，在实际生产环境的部署中，发现对于事务更新失败时，Jams-Ranking会对失败的事务进行切分和重试，整个过程对于研发人员是透明的，不利于线上问题排查，所以我们当时特地写了一个watchdog的工具，监控事务回滚达到十次以上的事务，查明原因后通过后台管理系统进行相应补偿，保证最终一致性

最终结果：

- 高效快速：能在数百毫秒内找到玩家排名以及进行更新
- 强一致性以及持久化、排名准确
- 可以扩展到任意数量的玩家
- 吞吐量有限制，只能支持约每秒 500次更新。

针对这个缺点谷歌官方也是给出了使用分片树和近似排名的解决方案，当然复杂的方案有更高的运维成本，所以我们优化工作也就到此为止

方案优化过程

方案1：每日一个滚动榜，当日汇聚（费时间）

首先记录每天的排行榜和一个滚动榜，加分时同时写入这两个榜单，每日零点后跑工具将前几天数据累加写入当日滚动榜，该方案缺点是时间复杂度高，7天榜还好，只需要读过去6天数据，如果是100天榜，该方案需要读过去99天榜，显然不可接受

方案2：全局N个滚动榜同时写（费空间）

要做到每日零点后榜单实时生效，而不需要等待离线作业的完成，一种方案是预写未来的榜单。可以写当天的滚动榜的同时，写往后N-1天的滚动榜一起写入该方案不仅能脱离离线作业做到实时更新，且可以省略每天的日榜。但缺点也不难看出，对于7天滚动榜，每次写操作需要更新7个榜单，但是对于百日榜，空间消耗无法接受，1000万榜单大约消耗1G内存

方案3：实时更新，常数次写操作

有不有办法做到既能实时更新，写榜数量也不随N的增加而增加呢?

仍然是记录每天的排行榜和一个滚动榜，加分操作也还是同时操作当日榜和全局榜，但每日零点的离线作业改为从全局榜中减去之前过期的数据，从而实现先滚动更新。 此方案每次只需读取一个日榜做减法，时间复杂度为O(1)；但是无法做到实时更新。 这个方案的优点是在十二点前提前准备好差分榜，到了十二点直接加上当天数据就是滚动榜内

容 ，这样就在常数次写操作的前提下，实现了滚动榜的实时更新

2. 如何解决重复答题

利用setnx防止重复答题 分布式锁是控制分布式系统之间同步访问共享资源的一种方式。 利用Redis的单线程特性对共享资源进行串行化处理

```
// 获取锁推荐使用set的方式
String result = jedis.set(lockKey, requestId, "NX", "EX", expireTime);
```

```
// 推荐使用redis+lua脚本
String lua = "if redis.call('get',KEYS[1]) == ARGV[1] then return redis.call('del',KEYS[1]) else return 0 end";
Object result = jedis.eval(lua, Collections.singletonList(lockKey))
```

3. 一个题目被多个人抢答

利用redis来实现乐观锁（抢答），好处是答错的人不影响状态，第一个秒杀答对的人才能得分。

1、利用redis的watch功能，监控这个 Corp:Activ:Qust: 的状态值

2、获取Corp:Activ:Qust: 的值，创建redis事务，给这个key的值-1

3、执行这个事务，如果key的值被修改过则回滚，key不变

4. 如何管理昵称重复

使用布隆过滤器：

它实际上是一个很长的二进制矢量数组和 K 个哈希函数。当一个昵称加入布隆过滤器中的时候，会进行如下操作：

- 使用 K 个哈希函数对元素值进行 K 次计算，得到 K 个哈希值。
- 根据得到的哈希值，在位数组中把对应下标的值置为 1。 Na

用户新增昵称时需要首先计算K个哈希值，如果K个哈希值有一个不为0则通过，否则不通过，不通过时通过加随机字符串再次检验，检测通过后返回给前端，帮助用户自动填写。

布隆过滤器的好处是它可以用来判断一个元素是否在一个集合中。它的优势是只需要占用很小的内存空间以及有着高效的查询效率。对于布隆过滤器而言，它的本质是一个位数组：位数组就是数组的每个元素都只占用 1 bit ，并且每个元素只能是 0 或者 1。

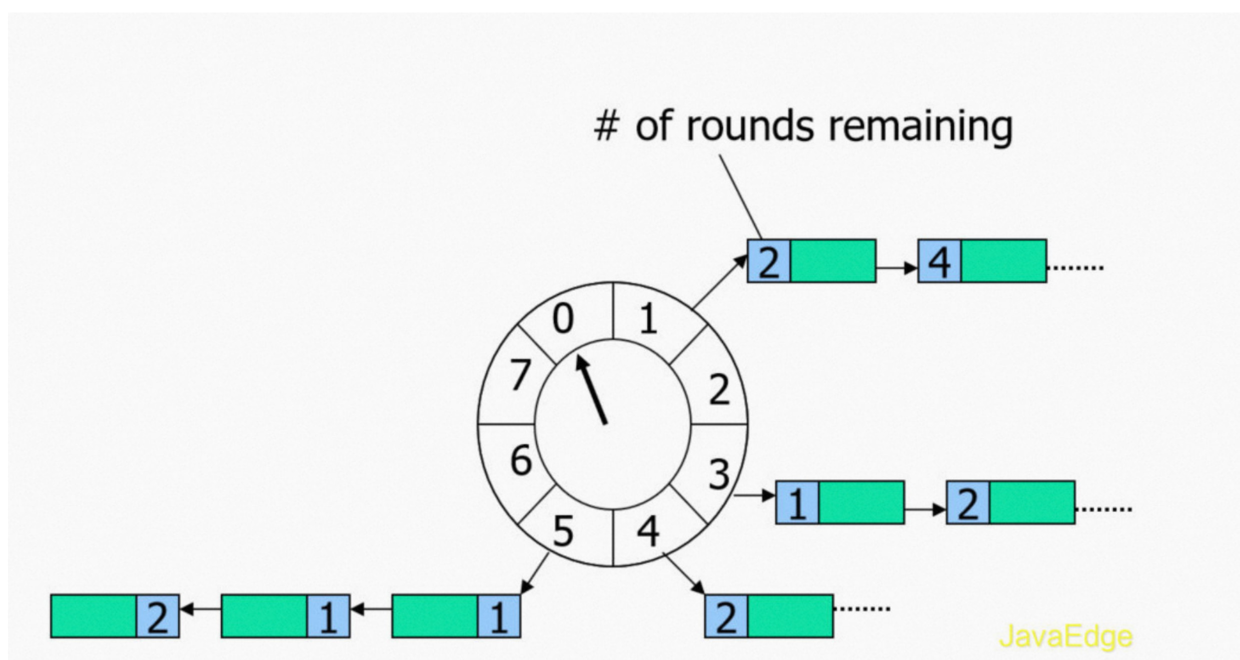
BloomFilter 的优势是，全内存操作，性能很高。另外空间效率非常高，要达到 1% 的误判率，平均单条记录占用 1.2 字节即可。而且，平均单条记录每增加 0.6 字节，还可让误判率继续变为之前的 1/10，即平均单条记录占用 1.8 字节，误判率可以达到 1/1000；平均单条记录占用 2.4 字节，误判率可以到 1/10000，以此类推。这里的误判率是指，BloomFilter 判断某个 key 存在，但它实际不存在的概率，因为它存的是 key 的 Hash 值，而非 key 的值，所以有概率存在这样的 key，它们内容不同，但多次 Hash 后的 Hash 值都相同。对于 BloomFilter 判断不存在的 key ，则是 100% 不存在的，反证法，如果这个 key 存在，那它每次 Hash 后对应的 Hash 值位置肯定是 1，而不会是 0

5. 如何管理出题定时任务

压测环境中服务器通过Netty的主从Reactor多路复用NIO事件模型，单机可以轻松应对十万长连接，但是每个业务中，由于每个用户登录系统后需要按照指定顺序答题，例如一共要答十道，那么服务器针对这一个用户就会产生十个定时任务，所以对于系统来说，定时器的数量就是百万级别的。

通过压测结果发现：JDK自带的Timer，在大概三万并发时性能就急剧下降了。也是此时根据业务场景的需要，将定时任务改成了Netty自带的HashedWheelTimer时间轮方案，通过压测单机在50万级别下依然能够平滑的执行。

也是这个强烈的反差，使我在强烈的好奇心促使下，阅读源码了解到常规的JDK 的Timer 和 DelayedQueue 等工具类，可实现简单的定时任务，单底层用的是堆数据结构，存取复杂度都是 O(NlogN)，无法支撑海量定时任务。Netty 经典的时间轮方案，正是通过将任务存取及取消操作时间复杂度降为 O(1)，而广泛应用在定时任务量大、性能要求高的场景中。



基于Netty的Websocket底层，服务器端维护一个高效批量管理定时任务的调度模型。时间轮一般会实现成一个环形数组结构，类似一个时钟，分为很多槽，一个槽代表一个时间间隔，每个槽使用双向链表存储定时任务。指针周期性地跳动，跳动到一个槽位，就执行该槽位的定时任务。

单层时间轮的容量和精度都是有限的，对于精度要求特别高、时间跨度特别大或是海量定时任务需要调度的场景，可以考虑使用多级时间轮以及持久化存储与时间轮结合的方案。时间轮的定时任务处理逻辑如下：

1. 将缓存在 timeouts 队列中的定时任务转移到时间轮中对应的槽中
2. 根据当前指针定位对应槽，处理该槽位的双向链表中的定时任务，从链表头部开始迭代：
   - 属于当前时钟周期则取出运行
   - 不属于则将其剩余的时钟周期数减一
3. 检测时间轮的状态。如果时间轮处于运行状态，则循环执行上述步骤，不断执行定时任务。

6. 如何解决客户端断连

使用Netty的重连检测狗ConnectionWatchdog

服务端定义refreshTime，当我们从channel中read到了服务端发来的心跳响应消息的话，就刷新refreshTime为当前时间

客户端在state是WRITER_IDLE的时候每隔一秒就发送一个心跳包到sever端，告诉server端我还活着。

当重连成功时，会触发channelActive方法，在这里我们开启了一个定时任务去判断refreshTime和当前时间的时间差，超过5秒说明断线了，要进行重连，最后计算重连次数，尝试连接2次以上连不上就会修改header信息强制重连去连另一个服务器。

秒杀项目

技术选型

秒杀用到的基础组件，主要有框架、KV 存储、关系型数据库、MQ。
框架主要有 Web 框架和 RPC 框架。

其中，Web 框架主要用于提供 HTTP 接口给浏览器访问，所以 Web 框架的选型在秒杀服务中非常重要。在这里，我推荐Gin，它的性能和易用性都不错，在 GitHub 上的 Star 达到了 44k。对比性能最好的 fasthttp，虽然 fasthttp 在请求延迟低于 10ms 时性能优势明显，但其底层使用的对象池容易让人踩坑，导致其易用性较差，所以没必要过于追求性能而忽略了稳定性

至于 RPC 框架，我推荐选用 gRPC，因为它的扩展性和性能都非常不错。在秒杀系统中，Redis 中的数据主要是给秒杀接口服务使用，以便将配置从管理后台同步到 Redis 缓存中。

KV 存储方面，秒杀系统中主要是用 Redis 缓存活动配置，用 etcd 存储集群信息。

关系型数据库中，MySQL 技术成熟且稳定可靠，秒杀系统用它存储活动配置数据很合适。主要 原因还是秒杀活动信息和库存数据都缓存在 Redis 中，活动过程中秒杀服务不操作数据库， 使用 MySQL 完全能够满足需求。

MQ 有很多种，其中 Kafka 在业界认可度最高，技术也非常成熟，性能很不错，非常适合用在秒杀系统中。Kafka 支持自动创建队列，秒杀服务各个节点可以用它自动创建属于自己的队列

方案设计

背景

- 秒杀业务简单，每个秒杀活动的商品是事先定义好的，商品有明确的类型和数量，卖完即止
- 秒杀活动定时上架，消费者可以在活动开始后，通过秒杀入口进行抢购秒杀活动
- 秒杀活动由于商品物美价廉，开始售卖后，会被快速抢购一空。
-

现象

- 秒杀活动持续时间短，访问冲击量大，秒杀系统需要应对这种爆发性的访问模型
- 业务的请求量远远大于售卖量，大部分是陪跑的请求，秒杀系统需要提前规划好处理策略

- 前端访问量巨大，系统对后端数据的访问量也会短时间爆增，对数据存储资源进行良好设计
- 活动期间会给整个业务系统带来超大负荷，需要制定各种策略，避免系统过载而宕机
- 售卖活动商品价格低廉，存在套利空间，各种非法作弊手段层出，需要提前规划预防策略

秒杀系统设计

首先，要尽力将请求拦截在系统上游，层层设阻拦截，过滤掉无效或超量的请求。因为访问量远远大于商品数量，所有的请求打到后端服务的最后一步，其实并没有必要，反而会严重拖慢真正能成交的请求，降低用户体验。

秒杀系统专为秒杀活动服务，售卖商品确定，因此可以在设计秒杀商品页面时，将商品信息提前设计为静态信息，将静态的商品信息以及常规的 CSS、JS、宣传图片等静态资源，一起独立存放到 CDN 节点，加速访问，且降低系统访问压力，在访问前端也可以制定种种限制策略，比如活动没开始时，抢购按钮置灰，避免抢先访问，用户抢购一次后，也将按钮置灰，让用户排队等待，避免反复刷新。

其次，要充分利用缓存，提升系统的性能和可用性。

用户所有的请求进入秒杀系统前，通过负载均衡策略均匀分发到不同 Web 服务器，避免节点过载。在 Web 服务器中，首先检查用户的访问权限，识别并发刷订单的行为。如果发现售出数量已经达到秒杀数量，则直接返回结束，要将秒杀业务系统和其他业务系统进行功能分拆，尽量将秒杀系统及依赖服务独立分拆部署，避免影响其他核心业务系统。

秒杀系统需要构建访问记录缓存，记录访问 IP、用户的访问行为，发现异常访问，提前进行阻断及返回。同时还需要构建用户缓存，并针对历史数据分析，提前缓存僵尸强刷专业户，方便在秒杀期间对其进行策略限制。这些访问记录、用户数据，通过缓存进行存储，可以加速访问，另外，对用户数据还进行缓存预热，避免活动期间大量穿透。

1. 如何解决超卖?

mysql乐观锁+redis预减库存+redis缓存卖完标记

第一是基于数据库乐观锁的方式保证数据并发扣减的强一致性；

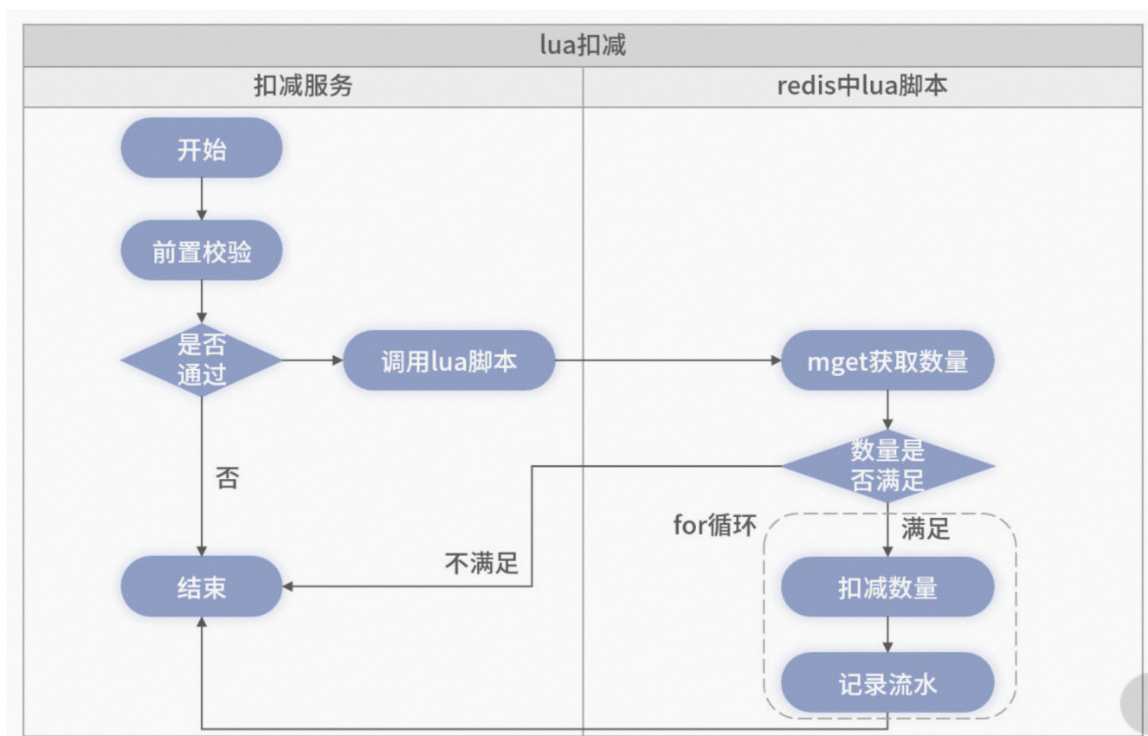
第二是基于数据库的事务实现批量扣减部分失败时的数据回滚。

在扣减指定数量前应先做一次前置数量校验的读请求（参考读写分离 + 全缓存方案）

> 纯数据库乐观锁+事务的方式性能比较差，但是如果不计成本和考虑场景的话也完全够用，因为任何没有机器配置的指标，都是耍流氓。如果我采用 Oracle 的数据库、100 多核的刀锋服务器、SSD 的硬盘，即使是纯数据库的扣减方案，也是可以达到单机上万的 TPS 的。

单线程Redis 的 lua 脚本实现批量扣减

当用户调用扣减接口时，将扣减的 对应数量 + 脚本标示传递至 Redis 即可，所有的扣减判断逻辑均在 Redis 中的 lua 脚本中执行，lua 脚本执行完成之后返还是否成功给客户端。



Redis 中的 lua 脚本执行时，首先会使用 get 命令查询 uuid 进行查重。当防重通过后，会批量获取对应的剩余库存状态并进行判断，如果一个扣减的数量大于剩余数量，则返回错误并提示数量不足。

Redis 的单线程模型，确保不会出现当所有扣减数量在判断均满足后，在实际扣减时却数量不够。同时，单线程保证判断数量的步骤和后续扣减步骤之间，没有其他任何线程出现并发的执行。

当 Redis 扣减成功后，扣减接口会异步的将此次扣减内容保存至数据库。异步保存数据库的目的是防止出现极端情况—— Redis 宕机后数据未持久化到磁盘，此时我们可以使用数据库恢复或者校准数据

最后，运营后台直连数据库，是运营和商家修改库存的入口。商家在运营后台进货物进行补充。同时，运营后台的实现需要将此数量同步的增加至 Redis，因为当前方案的所有实际扣减都在 Redis 中

> 纯缓存方案虽不会导致超卖，但因缓存不具备事务特性，极端情况下会存在缓存里的数据无法回滚，导致出现少卖的情况。且架构中的异步写库，也可能发生失败，导致多扣的数据丢失

可以借助顺序写的特性，将扣减任务同步插入任务表，发现异常时，将任务表作为undolog进行回滚
可以解决由于网络不通、调用缓存扣减超时、在扣减到一半时缓存突然宕机（故障 failover）了。针对上述请求，都有相应的异常抛出，根据异常进行数据库回滚即可，最终任务库里的数据都是准的

更进一步：由于任务库是无状态的，可以进行水平分库，提升整体性能

2. 如何解决重复下单?

mysql唯一索引+分布式锁

3. 如何防刷?

IP限流 | 验证码 | 单用户 | 单设备 | IMEI | 源IP |均设置规则

4. 热key问题如何解决?

redis集群+本地缓存+限流+key加随机值分布在多个实例中

1. 缓存集群可以单节点进行主从复制和垂直扩容
2. 利用应用内的前置缓存，但是需注意需要设置上限
3. 延迟不敏感，定时刷新，实时感知用主动刷新
4. 和缓存穿透一样，限制逃逸流量，单请求进行数据回源并刷新前置
5. 无论如何设计，最后都要写一个兜底逻辑，千万级流量说来就来

5. 应对高并发的读请求

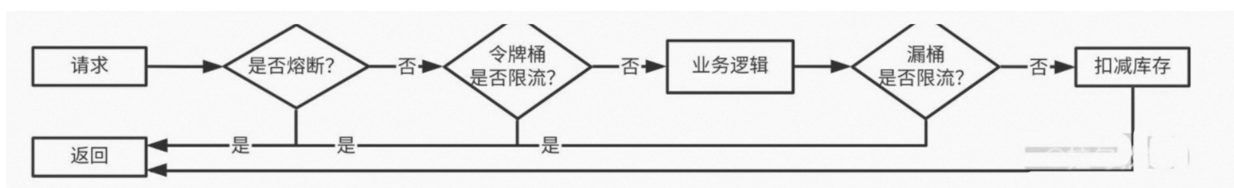
使用缓存策略将请求挡在上层中的缓存中

使用CDN，能静态化的数据尽量做到静态化，

加入限流（比如对短时间之内来自某一个用户，某一个IP、某个设备的重复请求做丢弃处理）

- 资源隔离限流会将对应的资源按照指定的类型进行隔离，比如线程池和信号量。
- 计数器限流，例如5秒内技术1000请求，超数后限流，未超数重新计数
- 滑动窗口限流，解决计数器不够精确的问题，把一个窗口拆分多滚动窗口
- 令牌桶限流，类似景区售票，售票的速度是固定的，拿到令牌才能去处理请求
- 漏桶限流，生产者消费者模型，实现了恒定速度处理请求，能够绝对防止突发流量

流量控制效果从好到差依次是：漏桶限流 > 令牌桶限流 > 滑动窗口限流 > 计数器限流



其中，只有漏桶算法真正实现了恒定速度处理请求，能够绝对防止突发流量超过下游系统承载能力。 不过，漏桶限流也有个不足，就是需要分配内存资源缓存请求，这会增加内存的使用率。而令牌桶限流算法中的“桶”可以用一个整数表示，资源占用相对较小，这也让它成为最常用的限流算法。正是因为这些特点，漏桶限流和令牌桶限流经常在一些大流量系统中结合使用。

6. 应对高并发的写请求

- 削峰：恶意用户拦截
  对于单用户多次点击、单设备、IMEI、源IP均设置规则
- 采用比较成熟的漏桶算法、令牌桶算法，也可以使用guava开箱即用的限流算法
  可以集群限流，但单机限流更加简洁和稳定
- 当前层直接过滤一定比例的请求，最大承载值前需要加上兜底逻辑
- 对于已经无货的产品，本地缓存直接返回
- 单独部署，减少对系统正常服务的影响，方便扩缩容

对于一段时间内的秒杀活动，需要保证写成功，我们可以使用 消息队列。
- 削去秒杀场景下的峰值写流量——流量削峰
- 通过异步处理简化秒杀请求中的业务流程——异步处理
- 解耦，实现秒杀系统模块之间松耦合——解耦

削去秒杀场景下的峰值写流量
- 将秒杀请求暂存于消息队列，业务服务器响应用户“秒杀结果正在处理中。。。”，释放系统资源去处理其它用户的请求。
- 削峰填谷，削平短暂的流量高峰，消息堆积会造成请求延迟处理，但秒杀用户对于短暂延迟有一定容忍度。秒杀商品有 1000 件，处理一次购买请求的时间是 500ms，那么总共就需要 500s 的时间。这时你部署 10 个队列处理程序，那么秒杀请求的处理时间就是 50s，也就是说用户需要等待 50s 才可以看到秒杀的结果，这是可以接受的。这时会并发 10 个请求到达数据库，并不会对数据库造成很大的压力。

通过异步处理简化秒杀请求中的业务流程
先处理主要的业务，异步处理次要的业务。
- 如主要流程是生成订单、扣减库存；
- 次要流程比如购买成功之后会给用户发优惠券，增加用户的积分。
- 此时秒杀只要处理生成订单，扣减库存的耗时，发放优惠券、增加用户积分异步去处理了。

解耦

实现秒杀系统模块之间松耦合将秒杀数据同步给数据团队，有两种思路：

- 使用 HTTP 或者 RPC 同步调用，即提供一个接口，实时将数据推送给数据服务。系统的耦合度高，如果其中一个服务有问题，可能会导致另一个服务不可用。
- 使用消息队列将数据全部发送给消息队列，然后数据服务订阅这个消息队列，接收数据进行处理。

7. 如何保证数据一致性

CacheAside旁路缓存读请求不命中查询数据库，查询完成写入缓存，写请求更新数据库后删除缓存数据。

```
// 延迟双删，用以保证最终一致性,防止小概率旧数据读请求在第一次删除后更新数据库
public void write(String key,Object data){
    redis.delKey(key);
    db.updateData(data);
    Thread.sleep(1000);
    redis.delKey(key);
}
```

为防缓存失效这一信息丢失，可用消息队列确保。

- 更新数据库数据；
- 数据库会将操作信息写入binlog日志当中；
- 另起一段非业务代码，程序订阅提取出所需要的数据以及key；
- 尝试删除缓存操作，若删除失败，将这些信息发送至消息队列；
- 重新从消息队列中获得该数据，重试操作；

订阅binlog程序在mysql中有现成的中间件叫canal，重试机制，主要采用的是消息队列的方式。

终极方案：请求串行化

真正靠谱非秒杀的方案：将访问操作串行化

1. 先删缓存，将更新数据库的写操作放进有序队列中
2. 从缓存查不到的读操作也进入有序队列

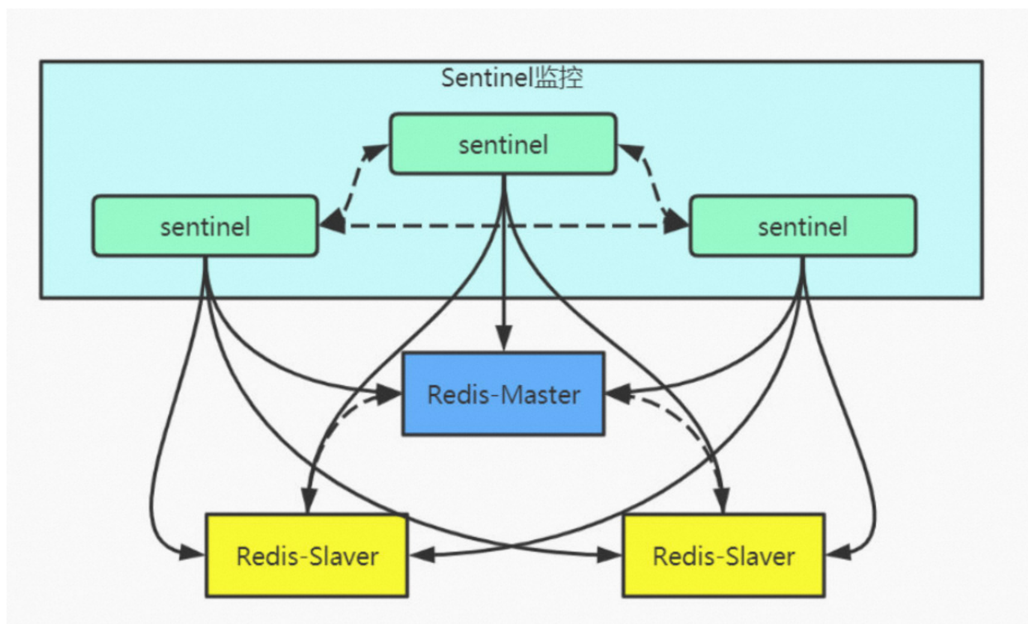
需要解决的问题：

1. 读请求积压，大量超时，导致数据库的压力：限流、熔断
2. 如何避免大量请求积压：将队列水平拆分，提高并行度。

8. 可靠性如何保障**

由一个或多个sentinel实例组成sentinel集群可以监视一个或多个主服务器和多个从服务器。哨兵模式适合读请求远多

于写请求的业务场景，比如在秒杀系统中用来缓存活动信息。 如果写请求较多，当集群 Slave 节点数量多了后，Master 节点同步数据的压力会非常大。



当主服务器进入下线状态时，sentinel可以将该主服务器下的某一从服务器升级为主服务器继续提供服务，从而保证redis的高可用性。

9. 秒杀系统瓶颈-日志

> 秒杀服务单节点需要处理的请求 QPS 可能达到 10 万以上。一个请求从进入秒杀服务到处理失败或者成功，至少会产生两条日志。也就是说，高峰期间，一个秒杀节点每秒产生的日志可能达到 30 万条以上

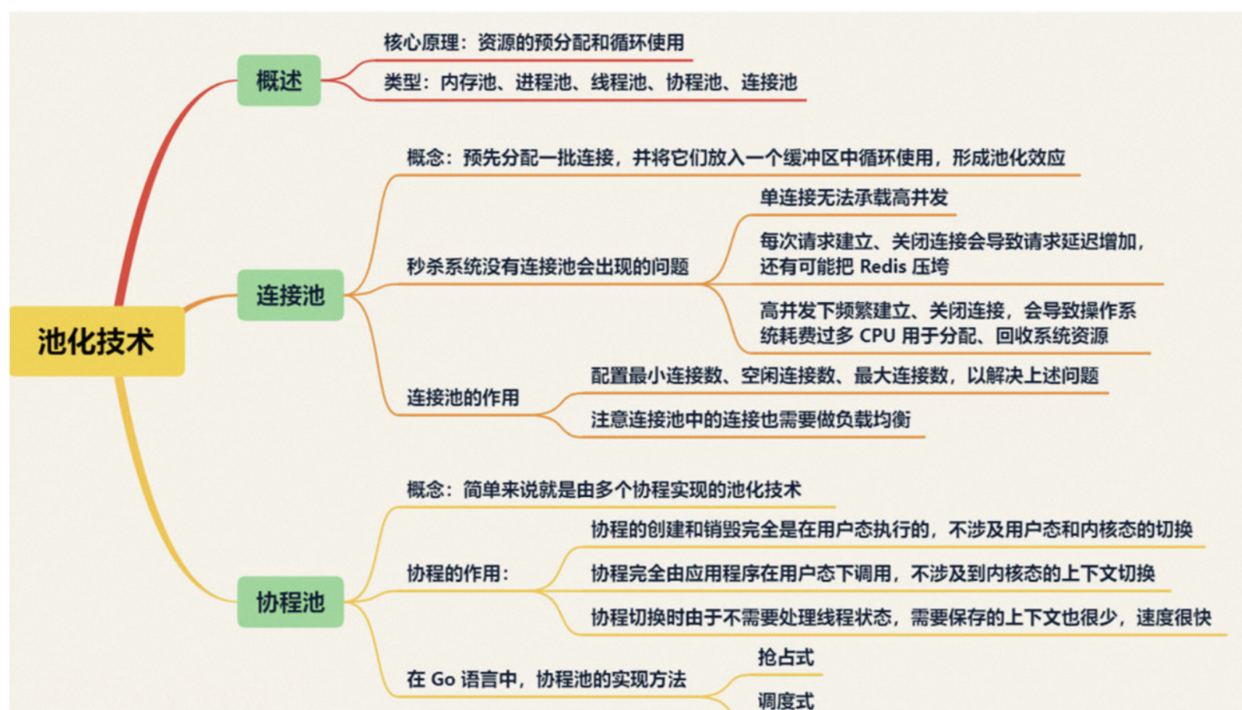
一块性能比较好的固态硬盘，每秒写的IOPS 大概在 3 万左右。也就是说，一个秒杀节点的每秒日志条数是固态硬盘 IOPS 的 10 倍，磁盘都扛不住，更别说通过网络写入到监控系统中。

- 每秒日志量远高于磁盘 IOPS，直接写磁盘会影响服务性能和稳定性
- 大量日志导致服务频繁分配，频繁释放内存，影响服务性能。
- 服务异常退出丢失大量日志的问题

解决方案

- Tmpfs，即临时文件系统，它是一种基于内存的文件系统。我们可以将秒杀服务写日志的文件放在临时文件系统中。相比直接写磁盘，在临时文件系统中写日志的性能至少能提升 100 倍，每当日志文件达到 20MB 的时候，就将日志文件转移到磁盘上，并将临时文件系统中的日志文件清空。
- 可以参考内存池设计，将给logger分配缓冲区，每一次的新写可以复用Logger对象
- 参考kafka的缓冲池设计，当缓冲区达到大小和间隔时长临界值时，调用Flush函数，减少丢失的风险

10. 池化技术



通常可以采用循环队列来保存空闲连接。使用的时候，可以从队列头部取出连接，用完后将空闲连接放到队列尾部。Netty中利用带缓冲区的 channel 来充当队列。

即时通信

1. 单聊消息可靠传输

TCP保证消息可靠传输三板斧：超时、重传、确认。服务端和客户端通信MSG和ACK的共计6个报文

- 请求报文（request，后简称为为R），客户端主动发送给服务端。
- 应答报文（acknowledge，后简称为A），服务器被动应答客户端的报文。
- 通知报文（notify，后简称为N），服务器主动发送给客户端的报文

在线消息流程：

A 消息请求 MSG:R => S 消息应答 MSG:A => S 消息通知B MSG:N

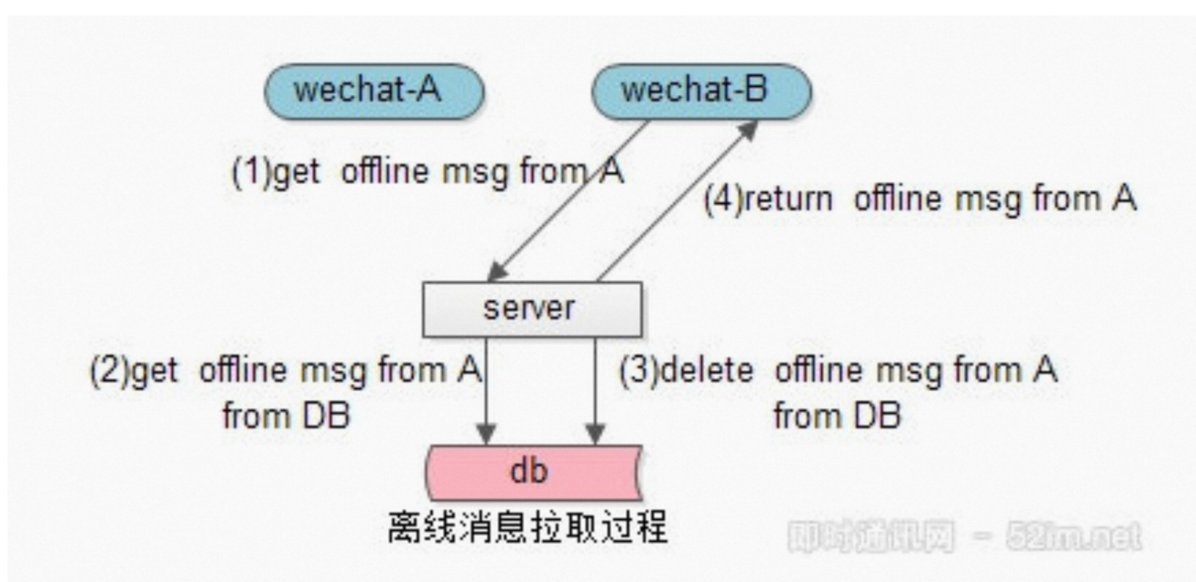
S 确认通知 ACK:N <= S 确认应答 ACK:A <= B确认请求S ACK:R

超时与重传、确认和去重：

A发出了 MSG:R ，收到了MSG:A之后，在一个期待的时间内，如果没有收到ACK:N，A会尝试将 MSG:R 重发。可能A同时发出了很多消息，所以A需要在本地维护一个等待ack队列，并配合timer超时机制，来记录哪些消息没有收到ACK:N，定时重发。确认ACK保证必达，去重保证唯一

离线消息流程

原方案：根据离线好友的标识，交互拉取指定的消息

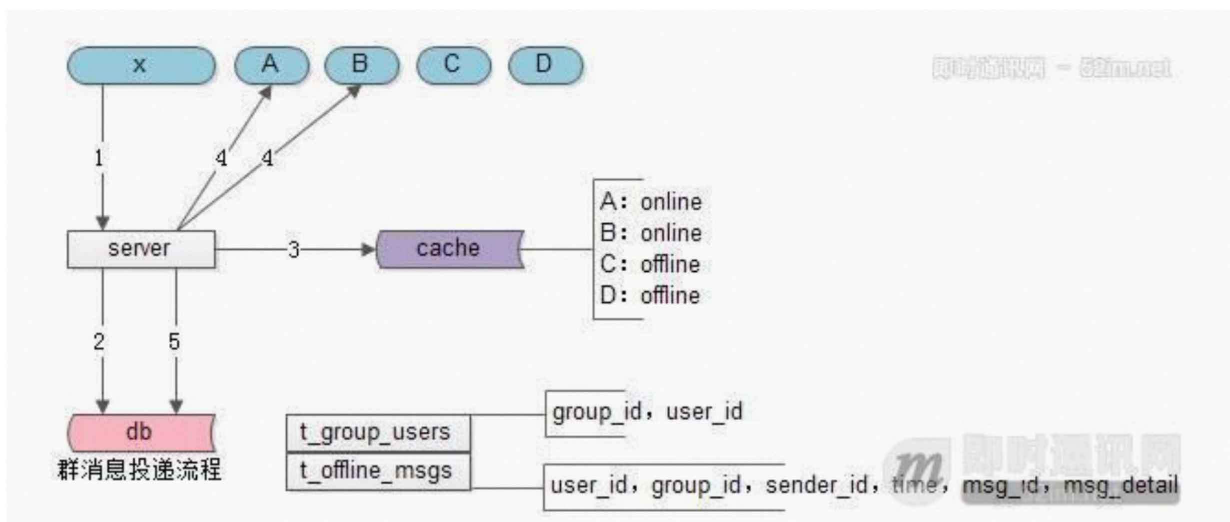


优化的方案：

- 如用户勾选全量则返回计数，在用户点击时拉取。
- 如用户未勾选全量则返回最近全部离线消息，客户端针对用户id进行计算。
- 全量离线信息可以通过客户端异步线程分页拉取，减少卡顿
- 将ACK和分页第二次拉取的报文重合，可以较少离线消息拉取交互的次数

2. 群聊消息如何保证不丢不重

> 在线的群友能第一时间收到消息；
> 离线的群友能在登陆后收到消息。



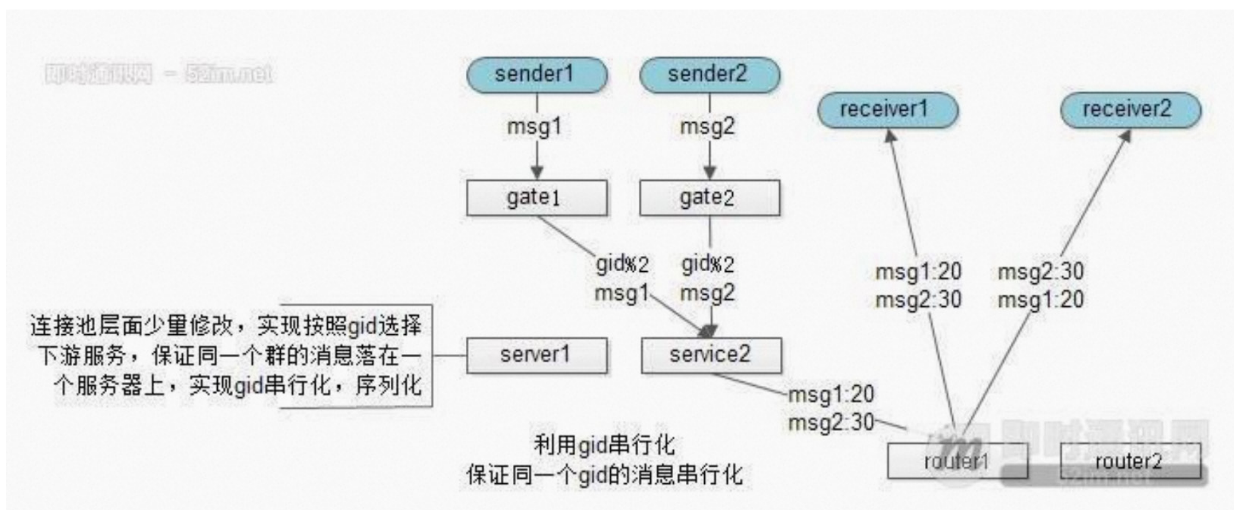
- 群消息发送者x向server发出群消息；
- server去db中查询群中有多少用户(x,A,B,C,D)；
- server去cache中查询这些用户的在线状态；
- 对于群中在线的用户A与B，群消息server进行实时推送；
- 对于群中离线的用户C与D，群消息server进行离线存储。

对于同一份群消息的内容，多个离线用户存储了很多份。假设群中有200个用户离线，离线消息则冗余了200份，这极大的增加了数据库的存储压力

- 离线消息表只存储用户的群离线消息msg_id，降低数据库的冗余存储量
- 加入应用层的ACK，才能保证群消息一定到达，服务端幂等性校验及客户端去重，保证不重复
- 每条群消息都ACK，会给服务器造成巨大的冲击，通过批量ACK减少消息风暴扩散系数的影响
- 群离线消息过多：拉取过慢，可以通过分页懒拉取改善。

3. 如何保证消息的时序性

方案：
- Id通过借鉴微信号段+跳跃的方式保证趋势递增
- 单聊借鉴数据库设计，单点序列化同步到其他节点保证多机时序
- 群聊消息使用单点序列化保证各个发送者的消息相对时序



优化：

- 利用服务器单点序列化时序，可能出现服务端收到消息的时序，与发出序列不一致
- 在A往B发出的消息中，加上发送方A本地的一个绝对时序，来表示接收方B的展现时序。
- 群聊消息保证一个群聊落在一个service上然后通过本地递增解决全局递增的瓶颈问题

4. 推拉结合

历史方案：

- 服务器在缓存集群里存储所有用户的在线状态 -> 保证状态可查
- 用户状态实时变更，任何用户登录/登出时，需要推送所有好友更新状态
- A登录时，先去数据库拉取自己的好友列表，再去缓存获取所有好友的在线状态

“消息风暴扩散系数”是指一个消息发出时，变成N个消息的扩散系数，这个系数与业务及数据相关，一定程度上它的大小决定了技术采用推送还是拉取。

优化方案：

- 好友状态推拉结合，首页置顶亲密、当前群聊，采用推送，否则可以采用轮询拉取的方式同步；
- 群友的状态，由于消息风暴扩散系数过大，可以采用按需拉取，延时拉取的方式同步；
- 系统消息/开屏广告等这种实时产生的消息，可以采用推送的方式获取消息；

5. 好友推荐

Neo4j 图谱数据库

智慧社区

18年初，针对我们Dubbo框架的智慧楼宇项目的单体服务显得十分笨重，需要采用微服务的形式进行架构的重新设计，当时，我阅读了Eric Evans 写的《领域驱动设计：软件核心复杂性应对之道》和Martin fowler的《微服务架构：Microservice》两本重量级书籍，书中了解到转型微服务的重要原因之一就是利用分治的思想减少系统的复杂性，是一种针对复杂问题的宏观设计，来应对系统后来规模越来越大，维护越来越困难的问题。然而，拆分成微服务以后，并不意味着每个微服务都是各自独立地运行，而是彼此协作地组织在一起。这就好像一个团队，规模越大越需要一些方法来组织，这正是我们需要DDD模型为我们的架构设计提供理论并实践的方法。

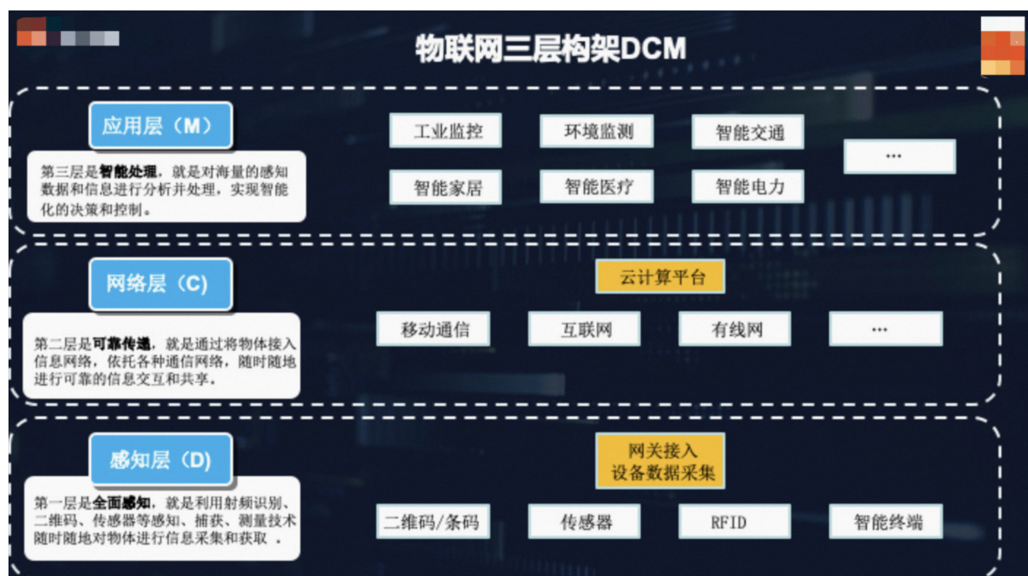
当时每次版本更新迭代动辄十几个微服务同时修改，有时一个简单的数据库字段变更，也需要同时变更多个微服务，引起了团队的反思：微服务化看上去并没有减少我们的工作量。《企业架构设计》中对于微服务的定义是小而专，但在起初的设计时，我们只片面的理解了小却忽视了专，此时我们才意识到拆分的关键是要保证微服务内高内聚，微服务间低耦合。

物联网架构

> 物联网是互联网的外延。将用户端延伸和扩展到物与人的连接。物联网模式中，所有物品与网络连接，并进行通信和场景联动。互联网通过电脑、移动终端等设备将参与者联系起来，形成的一种全新的信息互换方式

DCM系统架构

- 设备感知层（Device）：利用射频识别、二维码、传感器等技术进行数据采集
- 网络传输层（Connect）：依托通信网络和协议，实现可信的信息交互和共享
- 应用控制层（Manage）：分析和处理海量数据和信息，实现智能化的决策和控制

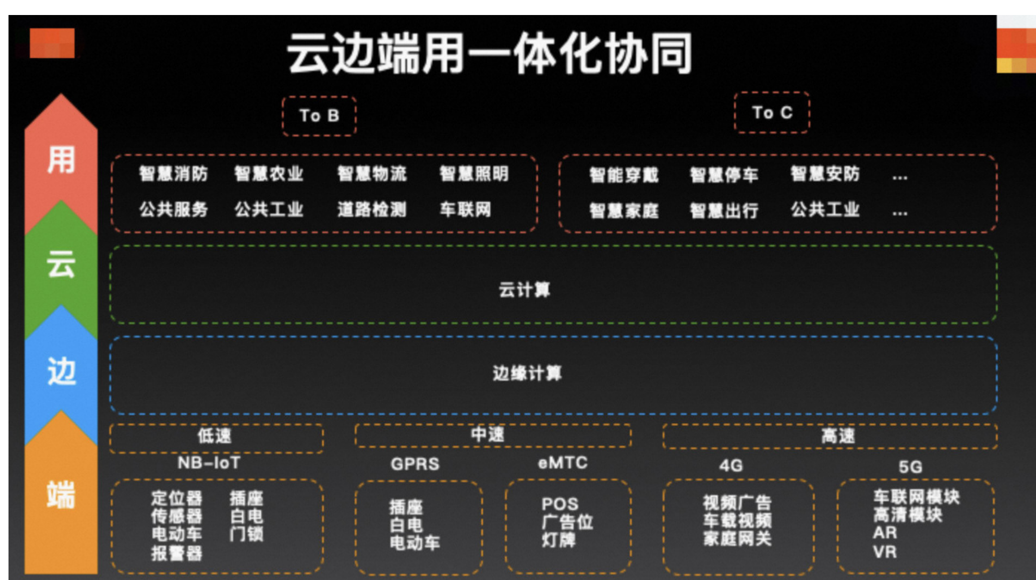


三要素

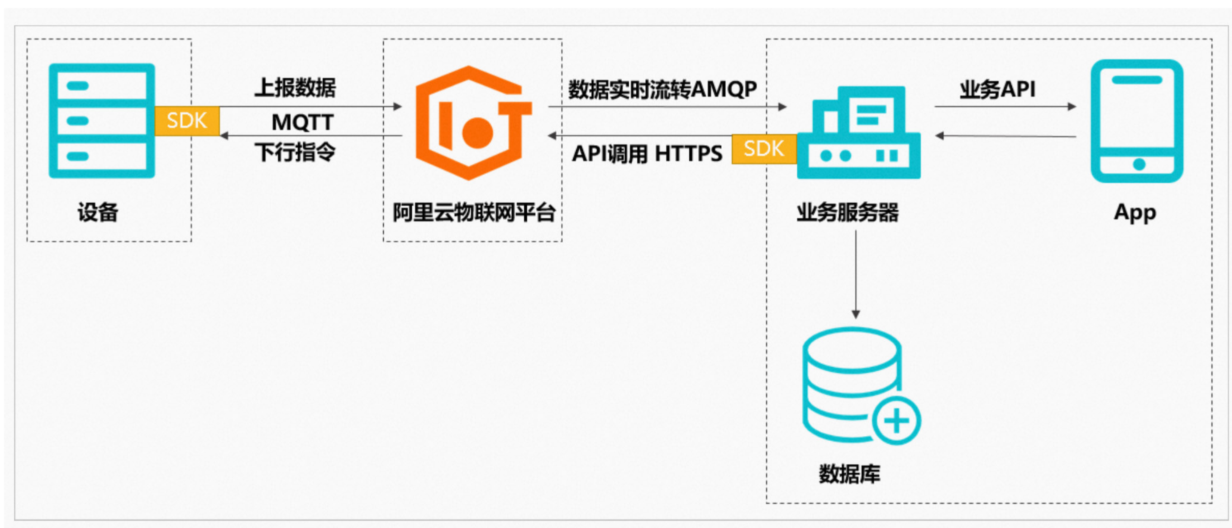
- 设备联网：通过不同的网络协议和通信标准，实现设备与控制端的连接
- 云端分析：提供监控、存储、分析等数据服务，以及保障客户的业务数据安全
- 云边协同：云端接受设备上报数据，下发设备管控指令

云 / 边 / 端协同

云端计算、终端计算和边缘计算是一个协同的系统，根据用户场景、资源约束程度、业务实时性等进行动态调 配，形成可靠、低成本的应用方案。从过去几年的发展积累来看，AI 已在物联网多个层面进行融合，比我们合作的海康威视、旷视宇视、商汤科技等纷纷发布了物联网AI相关平台和产品，和移动进行了紧密的融合。



物联网平台接入



向下连接海量设备，支撑设备数据采集上云；
向上通过调用云端API将指令下发至设备端，实现远程控制。

上行数据链路

- 设备建立MQTT长连接，上报数据（发布Topic和Payload）到物联网平台
- 物联网平台通过配置规则，通过RocketMQ、AMQP等队列转发到业务平台

下行指令链路

- 业务服务器基于HTTPS协议调用的API接口，发布Topic指令到物联网平台。
- 物联网平台通过MQTT协议，使用发布（指定Topic和Payload）到设备端。

门锁接入

WIFI门锁：非保活 平常处于断电休眠状态，需要MCU 唤醒才能传输和发送数据
蓝牙门锁：MCU串口对接和SDK对接，近距离单点登录和远距离网关登录
Zigbee门锁：非保活 但是保持心跳，MCU对接，Zigbee协议控制。
NB-Iot门锁：可以通过公网连接，把门禁变成SAAS服务，MCU

名词	解释
SaaS	Software-as-a-Service，提供给客户的服务是运营商运行在云计算基础设施上的应用程序。**用户可以在各种设备上通过客户端界面访问应用**，例如计算机浏览器。用户不需要管理或控制任何云计算基础设施，包括网络、服务器、操作系统、存储等资源，一切由 SaaS 提供商管理和运维。
PaaS	Platform-as-a-Service，表示平台即服务理念，客户不需要管理或控制底层的云基础设施，包括网络、服务器、操作系统、存储等，**但客户能控制部署的应用程序**，也可能控制运行应用程序的托管环境配置。
IaaS	Infrastructure-as-a-Service，表示基础设施即服务理念，提供的服务是对所有计算基础设施的利用，包括 CPU、内存、存储、网络等其它计算资源。**用户能够部署和运行任意软件，包括操作系统和应用程序。**

各种协议

HTTP协议（CS用户上网）

HTTP协议是典型的CS通讯模式，由客户端主动发起连接，向服务器请求XML或JSON数据。该协议最早是为了适用web浏览器的上网浏览场景和设计的，目前在PC、手机、pad等终端上都应用广泛，但并不适用于物联网场景

- 由于必须由设备主动向服务器发送数据，难以主动向设备推送数据。
- 物联网场景中的设备多样，运算受限的设备，难以实现JSON数据格式的解析

RESTAPI（松耦合调用）

REST/HTTP主要为了简化互联网中的系统架构，快速实现客户端和服务器之间交互的松耦合，降低了客户端和服务器之间的交互延迟。因此适合在物联网的应用层面，通过REST开放物联网中资源，实现服务被其他应用所调用。

CoAP协议（无线传感）

> 简化了HTTP协议的RESTful API，它适用于在资源受限的通信的IP网络。

MQTT协议（低带宽）

> MQTT协议采用发布/订阅模式，物联网终端都通过TCP连接到云端，云端通过主题的方式管理各个设备关注的通讯内容，负责将设备与设备之间消息的转发

适用范围：在低带宽、不可靠的集中星型网络架构（hub-and-spoke），不适用设备与设备之间通信，设备控制能力弱，另外实时性较差，一般都在秒级。协议要足够轻量，方便嵌入式设备去快速地解析和响应。具备足够的灵活性，使其足以为 IoT 设备和服务的多样化提供支持。应该设计为异步消息协议，这么做是因为大多数 IoT 设备的网络延迟很可能非常不稳定，若使用同步消息协议，IoT 设备需要等待服务器的响应，必须是双向通信，服务器和客户端应该可以互相发送消息。

AMQP协议（互操作性）

> 用于业务系统例如PLM，ERP，MES等进行数据交换。

适用范围：最早应用于金融系统之间的交易消息传递，在物联网应用中，主要适用于移动手持设备与后台数据中心的通信和分析。

XMPP协议（即时通信）

> 开源形式组织产生的网络即时通信协议。被IETF国际标准组织完成了标准化工作

适用范围：即时通信的应用程序，还能用在协同工具、游戏等。

XMPP在通讯的业务流程上是更适合物联网系统的，开发者不用花太多心思去解决设备通讯时的业务通讯流程，相对开发成本会更低。但是HTTP协议中的安全性以及计算资源消耗的硬伤并没有得到本质的解决。

JMS（Java消息服务）

Java消息服务（Java Message Service）应用程序接口，是一个Java平台中关于面向消息中间件（MOM）的API，用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。

Zigbee协议

低功耗，它保持IEEE 802.15.4（2003）标准

IOT流量洪峰

智慧社区IOT领域，不管是嵌入式芯片还是应用服务器都需要传递消息，常见上行的消息有：人脸识别开门、烟感雾感告警、共享充电桩充电，下行的广告下发、NB门禁开门指令、超级门板显示等，由于物联网设备时不时会故障和断网导致大量的流量洪峰，传统消息队列需要针对性优化。

- **上下行拆分**

上行消息特征：并发量高、可靠性和时延性要求低
下行消息特征：并发量低、控制指令的成功率要求高

- **海量Topic下性能**

Kafka海量Topic性能会急剧下降，Zookeeper协调也有瓶颈
多泳道消息队列可以实现IoT消息队列的故障隔离

- **实时消息优先处理**

NB门禁实时产生的开门指令必须第一优先级处理，堆积的消息降级
设计成无序、不持久化的，并与传统的FIFO队列隔离

- **连接、计算、存储分离**

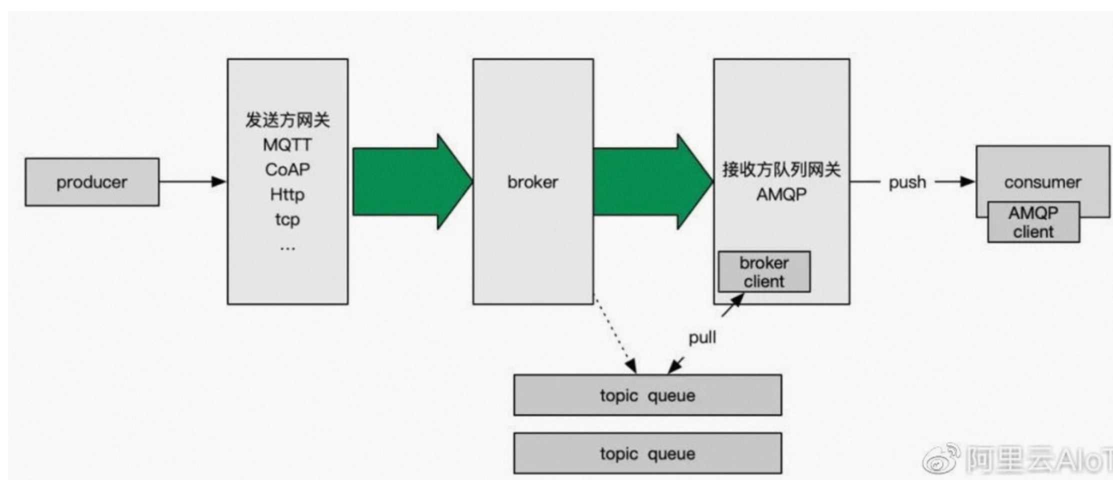
Broker只做流转分发，实现无状态和水平扩展

计算交给Flink，存储交给nosqlDB，实现高吞吐写

- **消息策略-推拉结合**

MQTT针对电池类物联网设备，AMQP针对安全性较高的门禁设备

消费端离线时存到queue，在线时将实时消息和从queue中拉取的消息一起推送



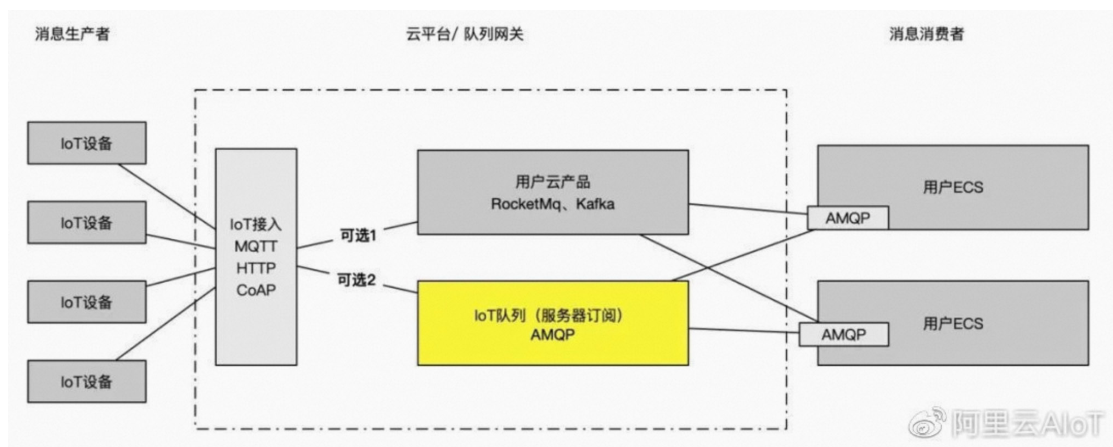
如果解决海量Topic

首先要做的就是分区、分组等水平拆分的方式，接下来考虑单实例如何处理更多Topic，传统消息队列在海量Topic下顺序写会退化成随机写，性能大幅下降

- 人工Sharding：部署多个Kafka集群，通过不同mq连接来隔离
- 合并Topic，客户端封装subTopic。比如一个服务的N个统计项，会消费到无关消息

基于这个思路，使用Kafka Streams或者Hbase列存储来聚合

针对单个Topic海量订阅的问题，可以在上层封装广播组件来协调批量发送



社区直播带货

> 使用端 / 边 / 云三级架构，客户端加密传输，边缘节点转发、云侧转码并持久化

产品的背景

> 上线时间，从调研到正式上线用了 3个月时间，上线后一个月内就要经历双十二挑战。在这么紧的上线时间要求下，需要用到公司提供的所有优势，包括cdn网络，直播牌照等

面临的挑战

- 直播数据是实时生成的，所有不能够进行预缓存
- 直播随时会发生，举办热点活动，相关服务器资源需要动态分配
- 直播的延迟对于用户体验影响很大，需要控制在秒级
- 直播sdk是内嵌在社区应用里的，整体要求不能超过5M

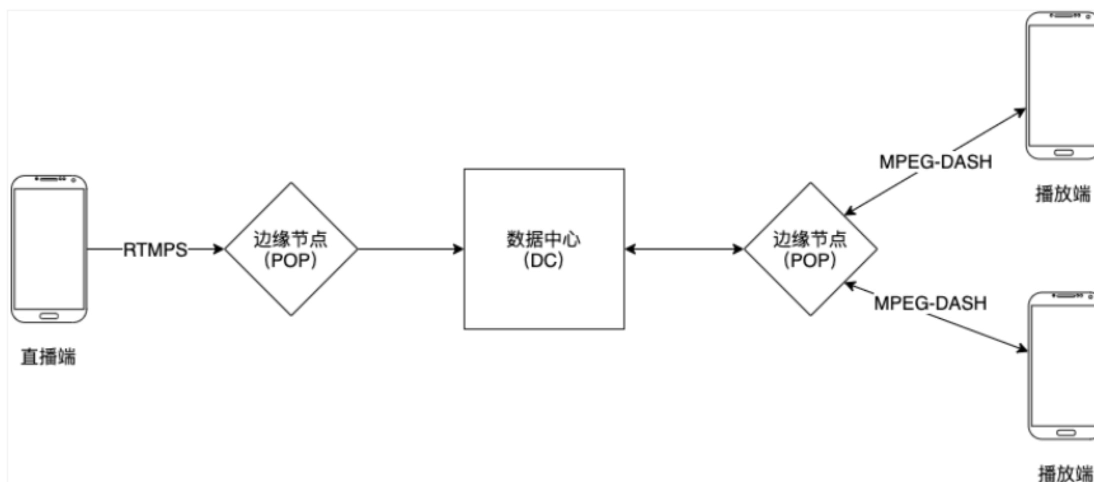
协议的比较

协议	上线时间	网络兼容	端对端延迟	应用大小	问题
WebRTC		✗			Webrtc 基于 UDP，和社区应用的网络架构不兼容
HTTP Upload			✗		会导致网络高延迟
Custom Protocol	✗				工程师需要实现自己的客户端与服务端的库，无法按时上线
Proprietary				✗	协议就需要几兆的空间，超出额度
RTMPS	✔	✔	✔	✔	TCP实时传输消息协议，更安全更可靠

整体流程

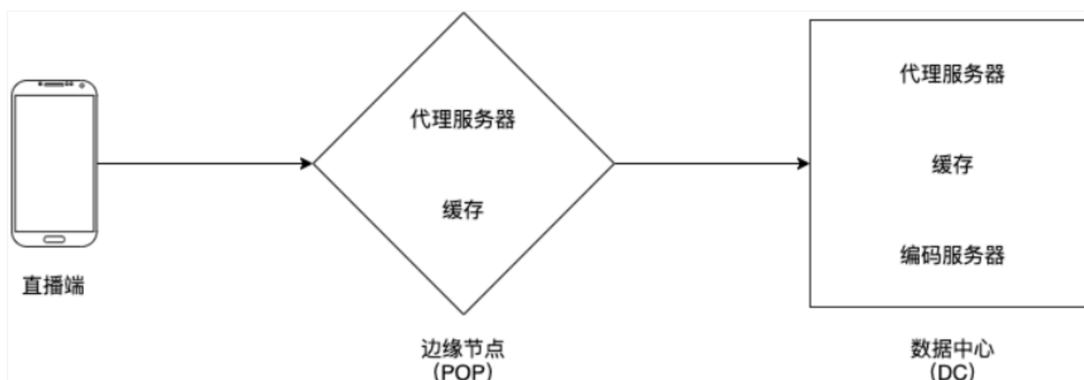
RTMPS：基于TCP实时传输消息协议，更安全更可靠

MPEG-DASH：是一种基于HTTP协议自适应比特率流媒体技术，应对复杂的环境



1. 直播端使用 RTMPS 协议发送直播数据到边缘节点（POP）
2. POP 使用RTMP发送数据到数据中心（DC）
3. DC 将数据编码成不同的清晰度并进行持久化存储
   云端转码主要有两种分辨率400x400 和 720x720.
4. 播放端通过 MPEG-DASH / RTMPS 协议接收直播数据
   如果用户网络不好MPEG-DASH会自动转换成低分辨率

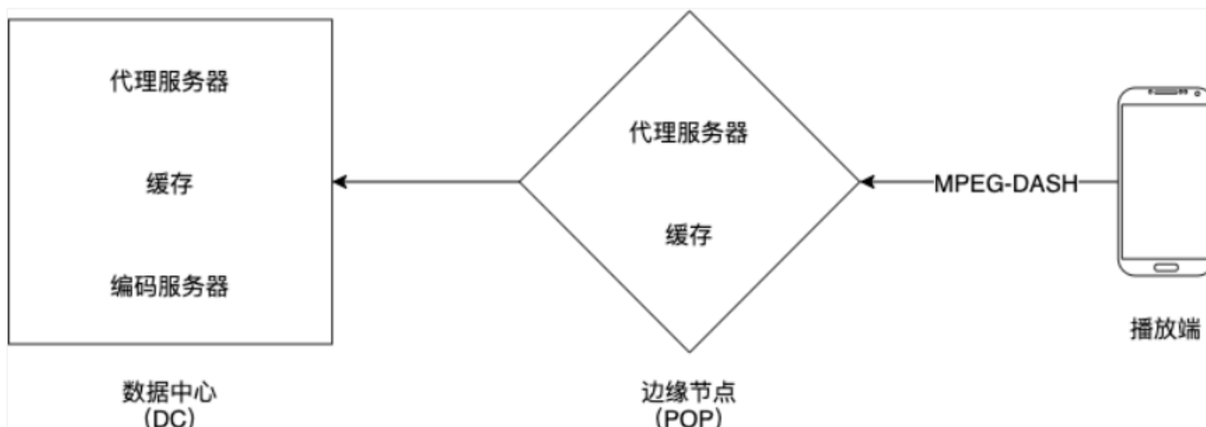
直播流程



1. 直播端使用 RTMPS 协议发送直播流数据到 POP 内的就近的代理服务器
2. 代理服务器转发直播流数据到数据中心的网关服务器（443转80）
3. 网关服务器使用直播 id 的一致性哈希算法发送直播数据到指定的编码服务器
4. 编码服务器有几项职责：
   - 4.1 验证直播数据的格式是否正确。
   - 4.2 关联直播 id 以及编码服务器第一映射，保证客户端即使连接中断或者服务器扩容时，在重新连接的时候依

- 然能够连接到相同的编码服务器
- 4.3 使用直播数据编码成不同解析度的输出数据
- 4.4 使用 DASH 协议输出数据并持久化存储

播放流程



1. 播放端使用 HTTP DASH 协议向 POP 拉取直播数据
2. POP 里面的代理服务器会检查数据是否已经在 POP 的缓存内。如果是的话，缓存会返回数据给播放端，否则，代理服务器会向 DC 拉取直播数据
3. DC 内的代理服务器会检查数据是否在 DC 的缓存内，如果是的话，缓存会返回数据给 POP，并更新 POP 的缓存，再返回给播放端。不是的话，代理服务器会使用一致性哈希算法向对应的编码服务器请求数据，并更新 DC 的缓存，返回到 POP，再返回到播放端。

收获

1. 项目的成功不，代码只是内功，考虑适配不同的网络、利用可利用的资源
2. 惊群效应在热点服务器以及许多组件中都可能发生
3. 开发大型项目需要对吞吐量和时延、安全和性能做出妥协
4. 保证架构的灵活度和可扩展性，为内存、服务器、带宽耗尽做好规划

直播高可用方案

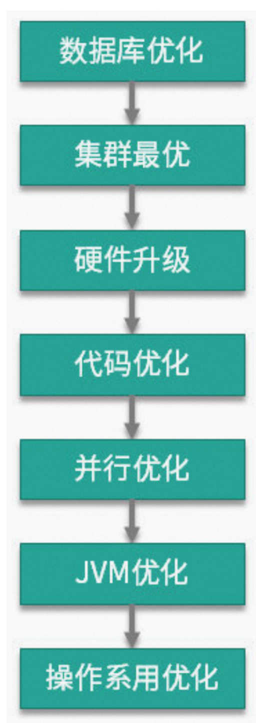
网络可靠性：

- 根据网络连接速度来自动调整视频质量
- 使用短时间的数据缓存来解决直播端不稳定，瞬间断线的问题
- 根据网络质量自动降级为音频直播以及播放

惊群效应：

- 当多个播放端向同一个 POP 请求直播数据的时候，如果数据不在缓存中
- 这时候只有一个请求 A 会到 DC 中请求数据，其他请求会等待结果
- 但是如果请求 A 超时没有返回数据的话，所有请求会一起向 DC 访问数据
- 这时候就会加大 DC 的压力，触发惊群效应
- 解决这个问题的方法就是通过实际的情况来调整请求超时的时间。这个时间如果太长的话会带来直播的延迟，太短的话会经常触发惊群效应（每个时间窗口只允许触发一次，设置允许最大回源数量）

性能优化方案



数据库优化： 数据库是最容易成为瓶颈的组件，考虑从 SQL 优化或者数据库本身去提高它的性能。如果瓶颈依然存

在，则会考虑分库分表将数据打散，如果这样也没能解决问题，则可能会选择缓存组件进行优化

集群最优：存储节点的问题解决后，计算节点也有可能发生问题。一个集群系统如果获得了水平扩容的能力，就会给下层的优化提供非常大的时间空间，由最初的 3 个节点，扩容到最后的 200 多个节点，但由于人力问题，服务又没有什么新的需求，下层的优化就一直被搁置着。

硬件升级：水平扩容不总是有效的，原因在于单节点的计算量比较集中，或者 JVM 对内存的使用超出了宿主机的承

载范围。在动手进行代码优化之前，我们会对节点的硬件配置进行升级。

代码优化：代码优化是提高性能最有效的方式，但需要收集一些数据，这个过程可能是服务治理，也有可能是代码流程优化。比如JavaAgent 技术，会无侵入的收集一些 profile 信息，供我们进行决策。

并行优化：并行优化是针对速度慢的接口进行并行调用。所以我们通常使用 ContDownLatch 对需要获取的数据进行并行处理，效果非常不错，比如在 200ms 内返回对 50 个耗时 100ms 的下层接口的调用。

JVM 优化： JVM 发生问题时，优化会获得巨大的性能提升。但在 JVM 不发生问题时，它的优化效果有限。但在代码优化、并行优化、JVM 优化的过程中，JVM 的知识却起到了关键性的作用

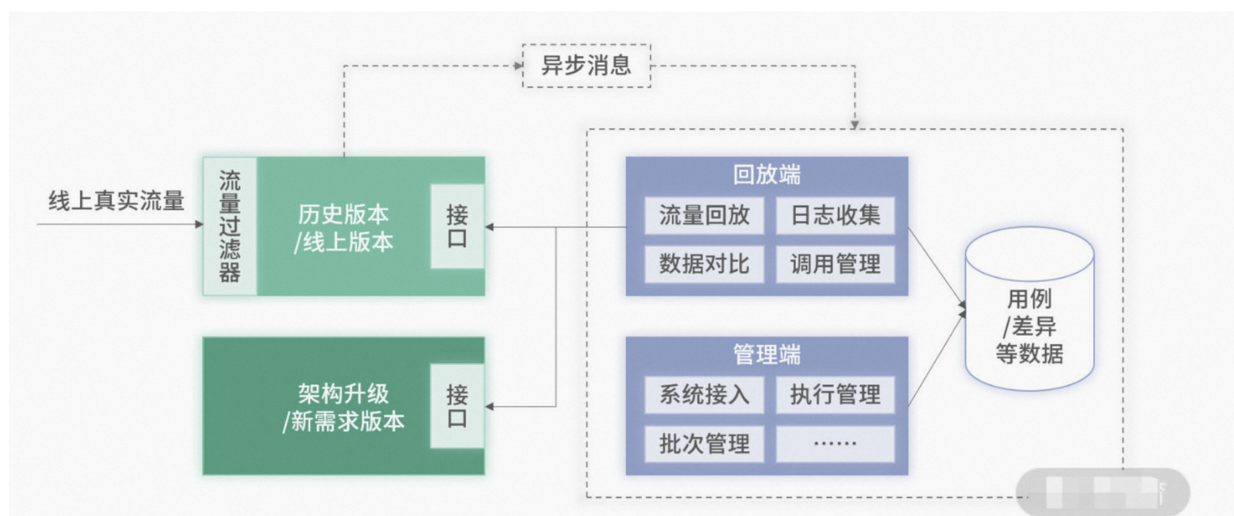
操作系统优化：操作系统优化是解决问题的杀手锏，比如像 HugePage、SWAP、“CPU 亲和性”这种比较底层的优化。但就计算节点来说，对操作系统进行优化并不是很常见。运维在背后会做一些诸如文件句柄的调整、网络参数的修改，这对于我们来说就已经够用了

流量回放自动化测试

> 系统级的重构，测试回归的工作量至少都是以月为单位，对于人力的消耗巨大。一种应对方案是，先不改造，到系统实在扛不住了再想办法。另一种应对方案是，先暂停需求，全力进行改造。但在实际工作场景中，上述应对策略往往很难实现。

场景：

1. 读服务均是查询，它是无状态的。
2. 不管是架构升级还是日常需求，读服务对外接口的出入参格式是没有变化的



- **日志收集，**主要作用是收集被测系统的真实用户请求，基于一定规则处理后作为系统用例；
  Spring 里的 Interceptor 、Servlet 里的 Filter 过滤器，对所有请求的入参和出参进行记录，并通过 MQ 发送出去。（注意错峰、过滤写、去重等）
- 数据回放是基于收集的用例，对被测系统进行数据回放，发起自动化测试回归；
  离线回放：只调用新服务，将返回的数据和日志里的出参进行比较，日志比较大
  实时回放：去实时调用线上系统和被测系统，并存储实时返回回放的结果信息，线上有负担
  并行回放：新版本不即时上线，每次调用老版本接口时概率实时回放新版本接口，耗时间周期
- **差异对比，**通过差异对比自动发现与预期不一致的用例，进而确定 Bug。
  采用文本对比，可以直观地看到哪个字段数据有差异，从而更快定位到问题。正常情况下，只要存在差异的数据，均可认为是 Bug，是需要进行修复的。

方法论

Discovery

考虑企业战略，分析客户需求，制定产品目标
由外到内：竞争对手的方案，为什么做，以后怎么发展，如何去优化。
自上而下：基于公司的战略，考虑自身能力和所处环境。
自下而上：从资源、历史问题、优先级出发，形成一套可行性实施方法。

Define

基于收集的信息，综合跨业务线的抽象能力和服务，先做什么后做什么，怎么做
设计新的架构，重点设计解决痛点问题。
拆分业务领域，重点划分工作临界上下文。

Design

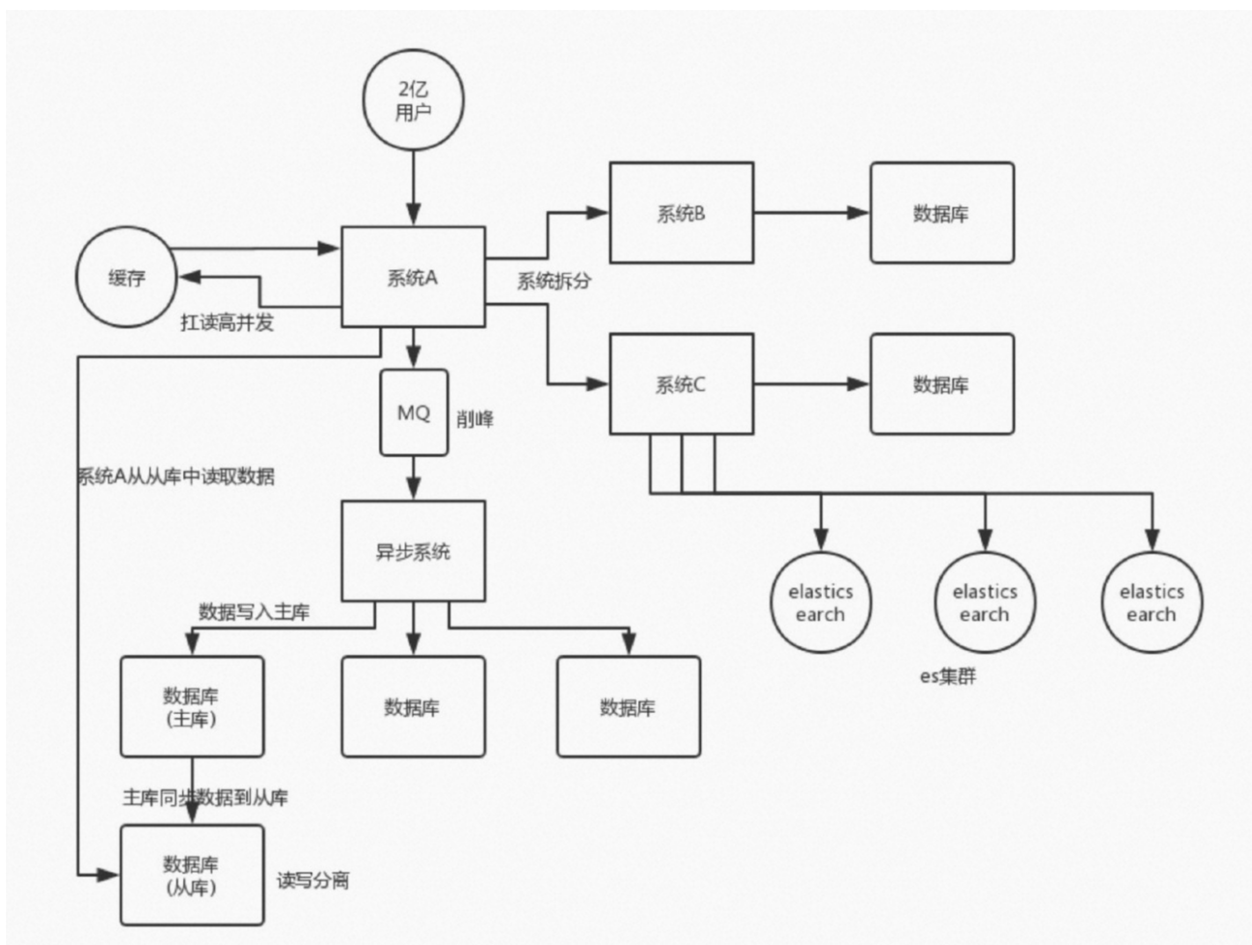
详细的业务设计，功能设计，交付计划，考核计划
产品愿景，产品形态，相关竞品方案对比，价值、优势、收益
梳理业务范围，要知道电商领域四大流（信息流、商流、资金流、物流）
MVP最小可用比，让客户和老大看到结果，最后通编写story把故事编圆

Delivery

交付阶段，根据反馈及时调整中台战略，减少损失和增大收益
合理制定每个阶段的绩效考核目标：
40%稳定+25%业务创新+20%服务接入+15%用户满意度

架构设计

1. 社区系统的架构



系统拆分

通过DDD领域模型，对服务进行拆分，将一个系统拆分为多个子系统，做成SpringCloud的微服务。微服务设计时要尽可能做到少扇出，多扇入，根据服务器的承载，进行客户端负载均衡，通过对核心服务的上游服务进行限流和降级改造。

一个服务的代码不要太多，1 万行左右，两三万撑死了吧。

大部分的系统，是要进行多轮拆分的，第一次拆分，可能就是将以前的多个模块该拆分开来了，比如说将电商系统拆分成订单系统、商品系统、采购系统、仓储系统、用户系统等等吧。

但是后面可能每个系统又变得越来越复杂了，比如说采购系统里面又分成了供应商管理系统、采购单管理系统，订单系统又拆分成了购物车系统、价格系统、订单管理系统。

CDN、Nginx静态缓存、JVM缓存

利用Java的模板thymeleaf可以将页面和数据动态渲染好，然后通过Nginx直接返回。动态数据可以从redis中获取。其中redis里的数据由一个缓存服务来进行消费指定的变更服务。

商品数据，每条数据是 10kb。100 条数据是 1mb，10 万条数据是 1g。常驻内存的是 200 万条商品数据，占用内存是 20g，仅仅不到总内存的 50%。目前高峰期每秒就是 3500 左右的请求量。

缓存

Redis cluster，10 台机器，5主5从，5 个节点对外提供读写服务，每个节点的读写高峰 QPS 可能可以达到每秒 5 万，5 台机器最多是 25 万读写请求每秒。

32G 内存+ 8 核 CPU + 1T 磁盘，但是分配给 Redis 进程的是 10g 内存，一般线上生产环境，Redis 的内存尽量不要超过 10g，超过 10g 可能会有问题。

因为每个主实例都挂了一个从实例，所以是高可用的，任何一个主实例宕机，都会自动故障迁移，Redis 从实例会自动变成主实例继续提供读写服务。

MQ

可以通过消息队列对微服务系统进行解耦，异步调用的更适合微服务的扩展
同时可以应对秒杀活动中应对高并发写请求，比如kafka在毫秒延迟基础上可以实现10w级吞吐量
针对IOT流量洪峰做了一些特殊的优化，保证消息的及时性
同时可以使用消息队列保证分布式系统最终一致性

分库分表

分库分表，可能到了最后数据库层面还是免不了抗高并发的要求，好吧，那么就 将一个数据库拆分为多个库，多个库来扛更高的并发；然后将一个表拆分为多个 表，每个表的数据量保持少一点，提高 sql 跑的性能。在通讯录、订单和商城商品模块超过千万级别都应及时考虑分表分库

读写分离

读写分离，这个就是说大部分时候数据库可能也是读多写少，没必要所有请求都 集中在一个库上吧，可以搞个主从架构，主库写入，从库读取，搞一个读写分离。 读流量太多的时候，还可以加更多的从库。比如统计监控类的微服务通过读写分离，只需访问从库就可以完成统计，例如ES

ElasticSearch

Elasticsearch，简称 es。es 是分布式的，可以随便扩容，分布式天然就可以支撑高并发，因为动不动就可以扩容加机器来扛更高的并发。那么一些比较简单的查询、统计类的操作，比如运营平台上的各地市的汇聚统计，还有一些全文搜索类的操作，比如通讯录和订单的查询。

2. 商城系统-亿级商品如何存储

基于 Hash 取模、一致性 Hash 实现分库分表
高并发读可以通过多级缓存应对
大促销热key读的问题通过redis集群+本地缓存+限流+key加随机值分布在多个实例中
高并发写的问题通过基于 Hash 取模、一致性 Hash 实现分库分表均匀落盘
业务分配不均导致的热key读写问题，可以根据业务场景进行range分片，将热点范围下的子key打散

具体实现：预先设定主键的生成规则，根据规则进行数据的分片路由，但这种方式会侵入商品各条线主数据的业务规则，更好的方式是基于分片元数据服务器（即每次访问分片前先询问分片元服务器在路由到实际分片）不过会带来复杂性，比如保证元数据服务器的一致性和可用性。

3. 对账系统-分布式事务一致性

> 尽量避免分布式事务，单进程用数据库事务，跨进程用消息队列

主流实现分布式系统事务一致性的方案：
1. 最终一致性：也就是基于 MQ 的可靠消息投递的机制，
2. 基于重试加确认的的最大努力通知方案。

理论上也可以使用（2PC两阶段提交、3PC三阶段提交、TCC短事务、SAGA长事务方案），但是这些方案工业上落地代价很大，不适合互联网的业界场景。针对金融支付等需要强一致性的场景可以通过前两种方案实现。（展开说的话参考分布式事务）

类型	特点	性能/吞吐能力	使用复杂度	常见原理	常见方案
【强一致性】型	需要实现rollback	受损严重	中	1、2PC 2、XA	（难度高、较少） 1、阿里Seata 2、Hmily
【最终一致性】型	N/A	受损较少	中/高	1、TCC 2、最大努力通知 3、异步补偿	1、自研补偿方案 2、自研MQ方案 3、RocketMQ 4、阿里Seata 5、Hmily 6、人工介入

本地数据库事务原理：undo log（原子性）+ redo log（持久性）+ 数据库锁（原子性&隔离性）+ MVCC（隔离性）
分布式事务原理：全局事务协调器（原子性）+ 全局锁（隔离性）+ DB本地事务（原子性、持久性）

一、我们公司账单系统和第三方支付系统对账时，就采用“自研补偿/MQ方案 + 人工介入”方式
落地的话：方案最“轻”，性能损失最少。可掌控性好，简单易懂，易维护。 考虑到分布式事务问题是小概率事件，留有补救余地就行，性能的损失可是实打实的反应在线上每一个请求上

二、也了解到业界比如阿里成熟Seata AT模式，平均性能会降低35%以上
我觉得不是特殊的场景不推荐

三、RocketMQ事务消息
听起来挺好挺简单的方案，但它比较挑业务场景，同步性强的处理链路不适合。
【重要】要求下游MQ消费方一定能成功消费消息。否则转人工介入处理。
【重要】千万记得实现幂等性。

4. 用户系统–多线程数据割接

由于项目需要进行数据割接，保证用户多平台使用用户感知的一致，将广东项目的几百万用户及业务数据按照一定的逻辑灌到社区云平台上，由于依赖了第三方统一认证和省侧crm系统，按照之前系统内割接的方法，通过数据库将用户的唯一标识查出来然后使用多线程向省侧crm系统获取结果。

但是测试的过程中，发现每个线程请求的数据发生了错乱，导致每个请求处理的数据有重复，于是立即停止了脚本，当时怀疑是多线程对资源并发访问导致的，于是把ArrayList 改成了CopyOnWriteArrayList，但是折腾了一晚上，不管怎么修改，线程之间一直有重复数据，叫了一起加班的同事也没看出问题来，和同事估算了一下不使用多线程，大概30–40个小时能跑完，想了下也能接受，本来已经准备放弃了。

不过回到家，我还是用多线程仔细单步模拟了下，整个处理的过程，发现在起线程的时候，有些子线程并没有把分配给他的全部id的list处理完，导致最终状态没更新，新线程又去执行了一遍，然后我尝试通过修改在线程外深拷贝一个List再作为参数传入到子线程里，（后续clear的时候也是clear老的List）果然，整个测试过程中再也没出现过重复处理的情况。

事后，我也深究了下原因：

```
if(arrayBuffer.length == 99) {
 val asList = arrayBuffer.toList
 exec.execute ( openIdInsertMethod(asList) )
 arrayBuffer.clear
}
```

在一个线程中开启另外一个新线程，则新开线程称为该线程的子线程，子线程初始优先级与父线程相同。不过主线程先启动占用了cpu资源，因此主线程总是优于子线程。然而，即使设置了优先级，也无法保障线程的执行次序。只不过，优先级高的线程获取CPU资源的概率较大，优先级低的并非没机会执行。

所以主线程上的clear操作有可能先执行，那么子线程中未处理完的数据就变成一个空的数组，所以就出现了多个线程出现了重复数据的原因，所以我们要保证的是子线程每次执行完后再进行clear即可。而不是一开始定位的保证ArrayList的安全性。所以将赋值(buffer->list)操作放在外面去执行后，多线程数据就正常了。

5. 秒杀系统场景设计

见秒杀项目方案设计

6. 统计系统-海量计数

中小规模的计数服务（万级）

最常见的计数方案是采用缓存 + DB 的存储方案。当计数变更时，先变更计数 DB，计数加 1，然后再变更计数缓存，修改计数存储的 Memcached 或 Redis。这种方案比较通用且成熟，但在高并发访问场景，支持不够友好。在互联网社交系统中，有些业务的计数变更特别频繁，比如微博 feed 的阅读数，计数的变更次数和访问次数相当，每秒十万到百万级以上的更新量，如果用 DB 存储，会给 DB 带来巨大的压力，DB 就会成为整个计数服务的瓶颈所在。即便采用聚合延迟更新 DB 的方案，由于总量特别大，同时请求均衡分散在大量不同的业务端，巨大的写压力仍然是 DB 的不可承受之重。

大型互联网场景（百万级）

直接把计数全部存储在 Redis 中，通过 hash 分拆的方式，可以大幅提升计数服务在 Redis 集群的写性能，通过主从

复制，在 master 后挂载多个从库，利用读写分离，可以大幅提升计数服务在 Redis 集群的读性能。而且 Redis 有持久化机制，不会丢数据

一方面 Redis 作为通用型存储来存储计数，内存存储效率低。以存储一个 key 为 long 型 id、value 为 4 字节的计数为例，Redis 至少需要 65 个字节左右，不同版本略有差异。但这个计数理论只需要占用 12 个字节即可。内存有效负荷只有 12/65=18.5%。如果再考虑一个 long 型 id 需要存 4 个不同类型的 4 字节计数，内存有效负荷只有 (8+16)/(65*4)= 9.2%。

另一方面，Redis 所有数据均存在内存，单存储历史千亿级记录，单份数据拷贝需要 10T 以上，要考虑核心业务上 1 主 3 从，需要 40T 以上的内存，再考虑多 IDC 部署，轻松占用上百 T 内存。就按单机 100G 内存来算，计数服务就要占用上千台大内存服务器。存储成本太高。

微博、微信、抖音（亿级）

定制数据结构，共享key 紧凑存储，提升计数有效负荷率
超过阈值后数据保存到SSD硬盘，内存里存索引
冷key从SSD硬盘中读取后，放入到LRU队列中
自定义主从复制的方式，海量冷数据异步多线程并发复制

7. 系统设计

1、需求收集

确认使用的对象（ToC：高并发，ToB：高可用）
系统的服务场景（即时通信：低延迟，游戏：高性能，购物：秒杀--一致性）
用户量级（万级：双机、百万：集群、亿级：弹性分布式、容器化编排架构）
百万读：3主6从，每个节点的读写高峰 QPS 可能可以达到每秒 5 万，可以实现15万，30万读性能
亿级读，通过CDN、静态缓存、JVM缓存等多级缓存来提高读并发
百万写，通过消息队列削峰填谷，通过hash分拆，水平扩展分布式缓存
亿级写，redis可以定制数据结构、SSD+内存LRU、冷数据异步多线程复制
持久化，（Mysql）承受量约为 1K的QPS，读写分离提升读并发，分库分表提升写并发

2、顶层设计

核心功能包括什么：
写功能：发送微博
读功能：热点资讯
交互：点赞、关注

3、系统核心指标

- 系统性能和延迟
  - 边缘计算 | 动静分离 | 缓存 | 多线程 |
- 可扩展性和吞吐量
  - 负载均衡 | 水平扩展 | 垂直扩展 | 异步 | 批处理 | 读写分离
- 可用性和一致性
  - 主从复制 | 哨兵模式 | 集群 | 分布式事务

4、数据存储

键值存储 : Redis (热点资讯)
文档存储 : MongoDB (微博文档分类)
分词倒排：Elasticsearch (搜索)
列型存储：Hbase、BigTable (大数据)
图形存储：Neo4j （ 社交及推荐)
多媒体：FastDfs (图文视频微博)

8．如何设计一个微博

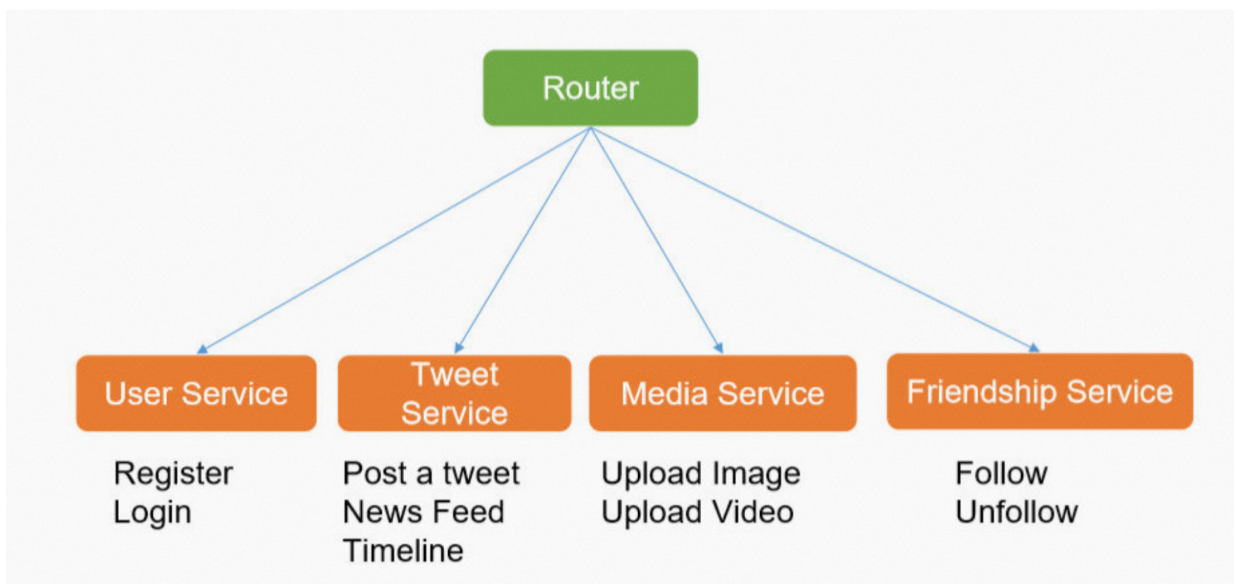
实现哪些功能：

筛选出核心功能 (Post a Tweet，Timeline，News Feed，Follow/Unfollow a user，Register/Login)

承担多大QPS：

QPS = 100，那么用我的笔记本作Web服务器就好了
QPS = 1K，一台好点的Web 服务器也能应付，需要考虑单点故障;
QPS = 1m，则需要建设一个1000台Web服务器的集群，考虑动态扩容、负载分担、故障转移
一台 SQL Database （ Mysql) 承受量约为 1K的QPS;
一台 NoSQL Database (Redis) 约承受量是 20k 的 QPS;
一台 NoSQL Database (Memcache) 约承受量是 200k 的 QPS;

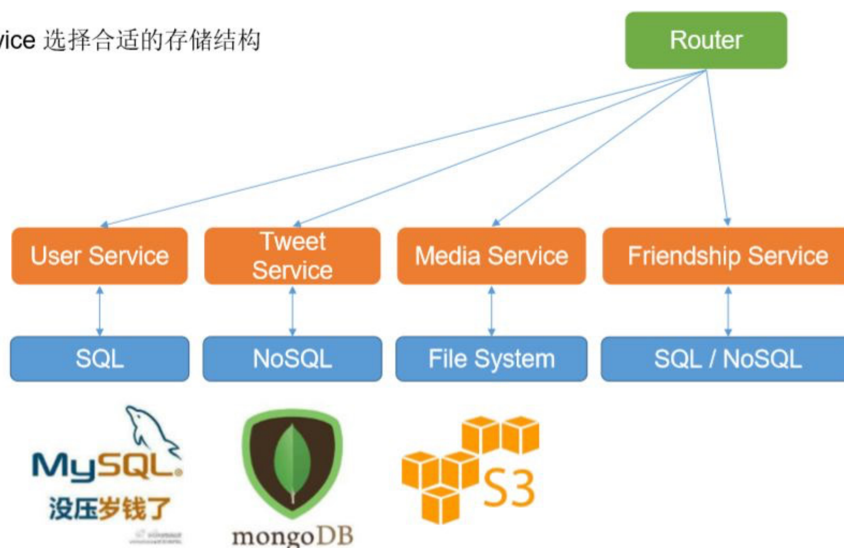
微服务战略拆分



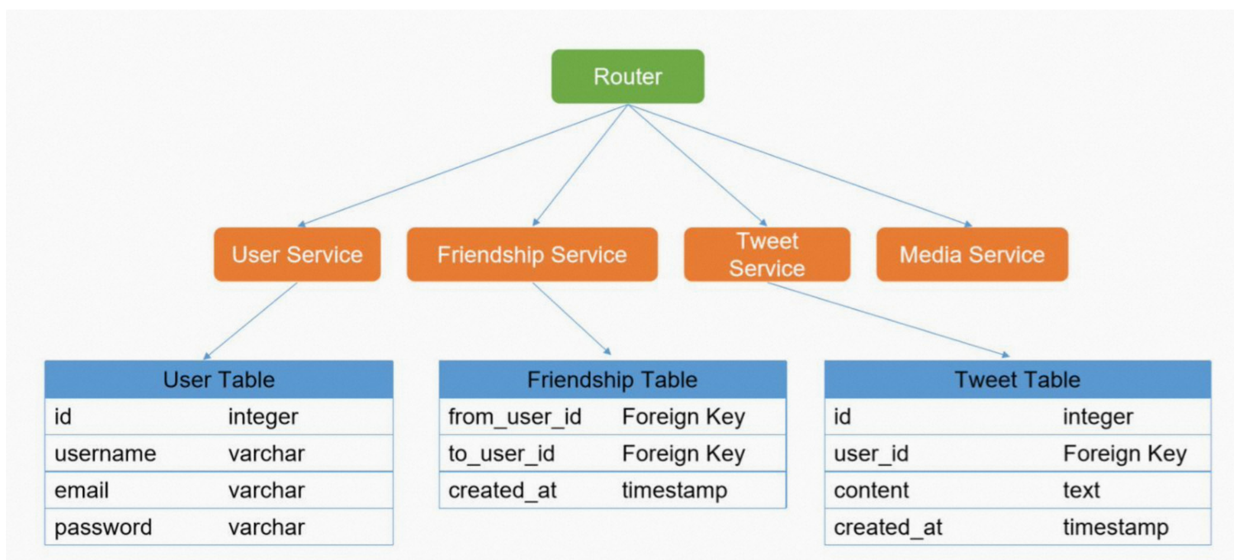
针对不同服务选择不同存储

- 第一步 Step 1: Select
  - 为每个 Application / Service 选择合适的存储结构
- 第二步 Step 2: Schema
  - 细化数据表结构

- 程序 = 算法 + 数据结构
- 系统 = 服务 + 数据存储



设计数据表的结构



基本差不多就形成了一个解决方案，但是并不是完美的，仍然需要小步快跑的不断的针对消息队列、缓存、分布式事务、分表分库、大数据、监控、可伸缩方面进行优化

领域模型落地

1. 拆分微服务

> 微服务内高内聚，微服务间低耦合

微服务内高内聚即单一职责原则

每个微服务中的代码变化都是同一类原因。因这类原因而需要变更的代码都在这个微服务中，与其他微服务无关，那么就可以将代码修改的范围缩小到这个微服务内。把这个微服务修改好了，独立修改、独立发布，该需求就实现了。这样，微服务的优势才能发挥出来。

微服务间低耦合开放封闭原则

就是说在微服务实现自身业务的过程中，如果需要执行的某些过程不是自己的职责，就应当将这些过程交给其他微服务去实现，你只需要对它的接口进行调用。这样，微服务之间的调用就实现了解耦。

领域建模就是将一个系统划分成了多个子域，每个子域都是一个独立的业务场景，每个子域的边界就是“限界上下文”。该业务场景会涉及许多领域对象，但分析建模始终需要围绕着业务场景的上下文进行。

领域事件通知机制最有效的方式就是通过消息队列，实现领域事件在微服务间的通知。

> “核心通讯录”微服务只负责发送变更消息到消息队列，不管谁会接收并处理这些消息；
> “门禁管理”微服务只负责接收照片变更消息，不管谁发送的这个消息。

2. 关联微服务

1. 按照限界上下文进行微服务的拆分，将领域模型划分到多个问题子域
2. 基于充血模型与贫血模型设计各个微服务的业务领域层（Service、Entity、Value）
3. 通过领域事件通知机制和微服务调用的推拉结合，将各个子域进行解耦关联

4. 通讯录 | 短信 | 推送通知 | 支付 | 文件服务

- **智慧通行**

  > 解决物业多品牌、多系统应用造成的信息孤岛，数据混乱的问题

  - 人脸门禁 | 可视对讲 | 电梯梯控 | 停车系统 | 访客预约

- **安全社区**

  > 通过图像视频识别、传感数据采集，实现报警联动和风险预警

  - 视频监控 | 周界报警 | 高空抛物 | 跨域追踪

- **全屋智能**

  > 围绕业主需求，逐步引入社区医疗、社区养老、社区团购、社区家政等服务

  - 超级面板 | 无线门锁 | 烟感雾感

- **增值服务**

  > 实现跨品牌的产品体验，支持基于matrix引擎的智能生活场景裂变能力

  - 智能充电 | 云广播 | 出入提醒 | 定向投放

3. 微服务的落地

> 通过合理的微服务设计，尽量让每次的需求变更都交给某个小团队独立完成，让需求变更落到某个微服务上进行变更。唯有这样，每次变更只需独立地修改这个微服务，独立打包、独立升级，新需求独立实现，才能发挥微服务的优势。

- **数据隔离：**数据库中用户信息表的读写只有通讯录微服务。当其他微服务需要读写用户信息时，就不能直接读取用户信息表，而是通过 API 接口去调用通讯录微服务。
- **接口复用：**因此，当多个团队向你提需求时，必须要对这些接口进行规划，通过复用尽可能少的接口满足他们的需求；当有新的接口提出时，要尽量通过现有接口解决问题。
- **向前兼容：**当调用方需要接口变更时怎么办？变更现有接口应当尽可能向前兼容，即接口的名称与参数都不变，只是在内部增加新的功能。宁愿增加一个新的接口也最好不要去变更原有的接口。
- **本地调用**：在访客申请微服务的本地，增加一个查询用户Service的 feign 接口。这样，访客申请Service就像本地调用一样调用查询用户Service，再通过 feign 接口实现远程调用。这种防腐层的设计，可以隔离当前微服务以外的其他微服务拆分变更导致的接口的失效的影响。
- **数据库去中心化：**
  - 微服务中通讯录服务与健康码服务分别对应的用户库与权限库，它们的共同特点是数据量小但频繁读取，可以选用小型的 MySQL 数据库并在前面架设 Redis 来提高查询性能；
    微服务中访客通行与生活缴费分别对应的通行记录库、订单库，其特点是数据量大并且高并发写，选用一个数据库显然扛不住这样的压力，因此可以选用了 TiDB 这样的 NewSQL 数据库进行分布式存储，将数据压力分散到多个数据节点中，从而解决 I/O 瓶颈；
  - 微服务中数据分析与通讯录查询这样的查询分析业务，则选用 NoSQL 数据库或大数据平台，通过读写分离将

- 生产库上的数据同步过来进行分布式存储，然后宽表一系列的预处理，应对海量历史数据的决策分析与秒级查询。（ NoSQL 为空的字段是不占用空间的，因此字段再多都不影响查询性能 ）

4. 领域模型的意义

贫血模型、充血模型、策略模式、装饰者模式只是DDD实现的方式，而DDD的真谛是领域建模。

做事不能仅凭一腔热血，一定要符合自然规律。其实软件的设计开发过程也是这样。对业务理解不深刻全局架构设计往往是过度设计，这时候应该抓主要流程，开始领域建模。

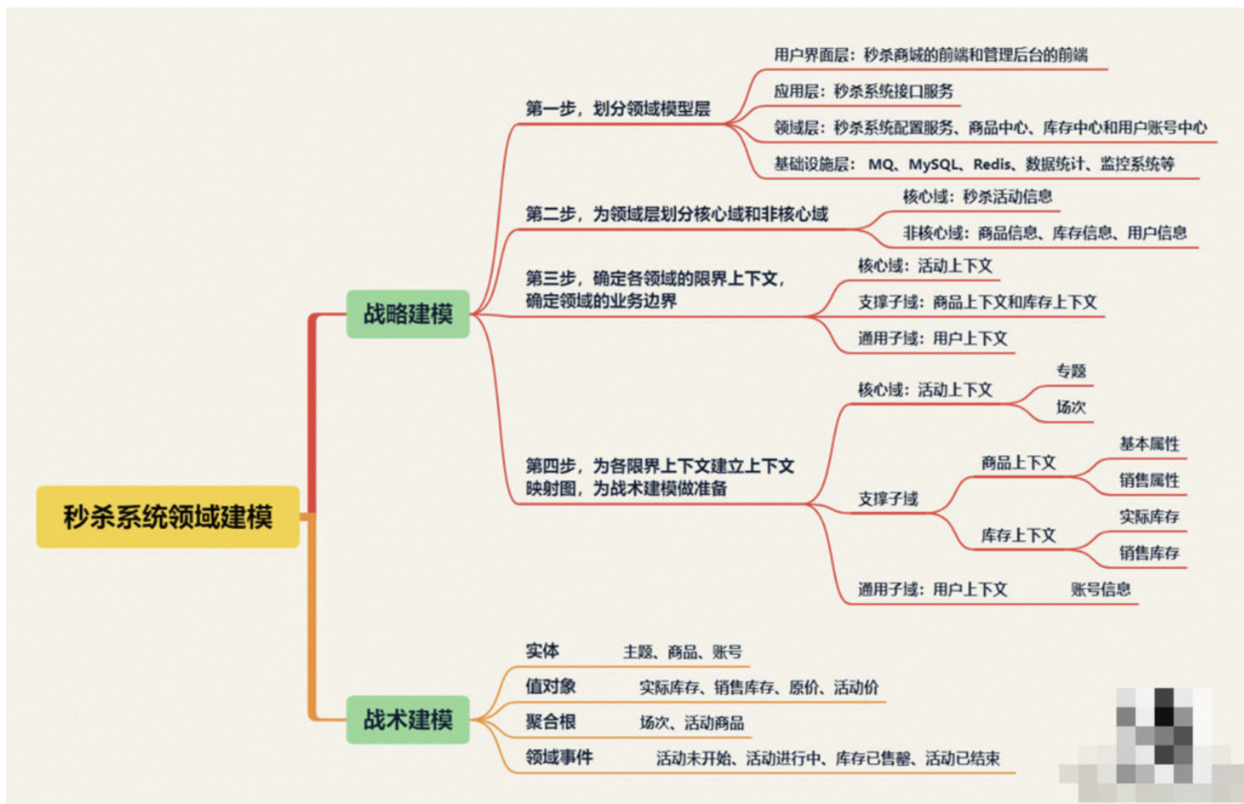
- 接着，每次添加新功能的时候，一方面要满足当前的需求，另一方面业务相关的领域建模设计刚刚满足需求，从而使设计最简化、代码最少。
- 这样的设计过程叫小步快跑。采用小步快跑的设计方法，一开始不用思考那么多问题，从简单问题开始逐步深入。领域模型就像小树一样一点儿一点儿成长，最后完成所有的功能。

> 保持软件设计不退化的关键在于每次需求变更的设计，只有保证每次需求变更时做出正确的设计，才能保证软件以一种良性循环的方式不断维护下去。

有没有一种方法，让我们在第十次变更、第二十次变更、第三十次变更时，依然能够找到正确的设计呢？有，那就是领域驱动设计

那么在每次需求变更时，将变更还原到真实世界中，看看真实世界是什么样子的，根据真实世界进行变更。

5. 战略建模



6. 相关名词

领域和子域（Domain/Subdomain）

在上下文地图构建的领域中，对应模块，使用限界上下文划分领域，对应微服务

限界上下文（Bounded Context）

在一个领域/子域中，有概念上的领域边界，任何领域对象在该边界内部的有不依赖外部的确切含义。

领域对象

服务、实体与值对象是领域驱动设计的领域对象，可以通过贫血模型和充血模型转换为程序设计

实体和值对象

通过一个唯一标识字段来区分真实世界中的每一个个体的领域对象，称为实体。真实世界中那些一成不变的、本质性的事物的领域对象，称为值对象。 可变性是实体的特点，而不变性则是值对象的本质。

贫血模型与充血模型

POJO对象中只保存get/set方法，没有任何业务逻辑，这样的设计被称为贫血模型

充血模型是封装和继承思想的体现，门禁设备实体中，包含特征值下发、广告下发、通行记录回调等方法，不同厂商的实体针对多态进行聚合，并通过工厂或仓库对外提供服务。在充血模型中， Service 只干一件非常简单的事，就是直接去调用对象中的工厂方法生成不同产品，其他的什么都不干。

聚合

聚合体现的是一种整体与部分的关系。正是因为有这样的关系，在操作整体的时候，整体就封装了对部分的操作。如何正确理解是否存在聚合的关系：就是当整体不存在时，部分就变得没有了意义。部分是整体的一个部分，与整体有相同的生命周期。

工厂

通过装配，创建领域对象，是领域对象生命周期的起点。譬如，系统要通过 ID 装载一个访客申请：

1. 表单工厂分别调用表单信息DAO、表单明细 DAO 和用户DAO 去进行查询；
2. 将得到的表单明细对象、用户对象进行装配，分别 set 到表单信息对象的表单明细与用户属性中；
3. 最后，表单工厂将装配好的表单对象返回给表单仓库。

仓库

如果服务器是一个非常强大的服务器，那么我们不需要任何数据库。系统创建的所有领域对象都放在仓库中，当需要这些对象时，通过 ID 到仓库中去获取。

- 当客户程序通过 ID 去获取某个领域对象时，仓库会通过这个 ID 先到缓存中进行查找：
- 查找到了，则直接返回，不需要查询数据库；
- 没有找到，则通知工厂，工厂调用 DAO 去数据库中查询，然后装配成领域对象返回给仓库。
- 仓库在收到这个领域对象以后，在返回给客户程序的同时，将该对象放到缓存中

致谢&声明

Hi，大家好，我是大淘宝技术工程师淘苏，很高兴能和大家一起分享我的学习经验。

本书内容为我个人的学习经验总结，部分内容来源于互联网，在此想要一并感谢所有同行和教育者。首先，特别感谢拉勾教育、爪哇教育、左程云算法平台提供的部分学习资料，也感谢在打怪升级路程中互帮互助的工程师队友们；其次，我在学习过程中参考了以下专业书籍，感谢各位编辑作者和出版社，为我的知识系统完整性带来了极大的提升帮助；最后，本书内容整理中还涉及到一些互联网的材料尚未查询到出处，未能详尽列举之处还请各位同行海涵，也感谢大家在知识分享上的无私和包容。希望我整理的这本知识小手册，能够帮助更多小伙们在学习和技术上精进一步。

淘苏的书籍学习清单：

1.《32个Java面试必考点》张雷
2.《Netty核心原理剖析与RPC实践》若地
3.《深入浅出Java虚拟机》李国
4.《架构师的36项修炼》李智慧
5.《架构设计面试精讲》刘海丰
6.《DDD 微服务落地实战》范钢

出品方：大淘宝技术品牌