[一.分布式架构要解决的4大核心问题 4](#_Toc13575)

[二. 高并发问题及解决方案 4](#_Toc20338)

[2.1 系统分类 4](#_Toc20561)

[2.1.1 分类依据 4](#_Toc23232)

[2.1.2 侧重于高并发读的系统 5](#_Toc23545)

[2.1.3 侧重于高并发写的系统 5](#_Toc24495)

[2.1.4同时侧重于高并发读与写的系统 5](#_Toc17309)

[2.2 高并发读解决方案 5](#_Toc31503)

[2.2.1 策略1：动静分离与CDN加速 5](#_Toc22965)

[Nginx作为静态资源服务器 5](#_Toc25280)

[CDN加速 6](#_Toc31966)

[2.2.2 策略2：加缓存 6](#_Toc25145)

[本地缓存或Memcached/Redis集中式缓存 6](#_Toc23735)

[2.2.3 策略3：库读写分离 7](#_Toc23680)

[2.2.4 策略4：并发读与Pipeline 7](#_Toc651)

[多个异步RPC并行调用 7](#_Toc1700)

[冗余请求 8](#_Toc9304)

[缓存Pipeline管道 8](#_Toc26419)

[2.2.5 策略5：批量读 9](#_Toc15599)

[缓存批量读写 9](#_Toc19016)

[Mysql库批量读写 10](#_Toc31301)

[2.2.6 策略6：重写轻读 10](#_Toc5924)

[信息Feeds流 10](#_Toc23348)

[宽表与搜索引擎 11](#_Toc7022)

[2.3高并发写解决方案 12](#_Toc24624)

[2.3.1 策略1：数据分片 12](#_Toc16780)

[数据库的分库分表 12](#_Toc3777)

[JDK的ConcurrenHashMap实现 12](#_Toc4791)

[Kafka的Partition 12](#_Toc17861)

[ES搜索引擎的分布式索引/分片存储 13](#_Toc10934)

[Redis分片存储 13](#_Toc4294)

[2.3.2策略2：任务分片 13](#_Toc12897)

[CPU指令流水线 13](#_Toc23264)

[Map/Reduce大数据 13](#_Toc24417)

[1+N+M网络模型（Netty、Tomcat） 13](#_Toc9748)

[多线程 13](#_Toc10950)

[2.3.3 策略3：异步化 14](#_Toc20989)

[业务层面：轮询+消息中间件 14](#_Toc15718)

[发送验证码 14](#_Toc5427)

[电商的拆合单 14](#_Toc18469)

[广告计费系统 14](#_Toc9968)

[接口层面：异步HTTP\RPC\MYSQL接口 14](#_Toc28570)

[Apache的AsyncHttpClient 14](#_Toc17070)

[异步RPC 15](#_Toc28069)

[异步Mysql 15](#_Toc15830)

[JDK与操作系统层面：NIO/AIO 15](#_Toc24822)

[异步落盘-延时写入 15](#_Toc26546)

[2.3.4 策略4：批量 16](#_Toc16791)

[Kafka百万QPS写入 16](#_Toc28235)

[广告计费系统的合并计费 16](#_Toc27381)

[Mysql库的小事务合并机制 16](#_Toc5847)

[三. 高可靠问题及解决方案 17](#_Toc28412)

[3.1 容量评估与扩容规划 17](#_Toc20189)

[3.1.1 吞吐量、响应时间、并发数的关系 17](#_Toc17214)

[3.1.2 容量规划 17](#_Toc830)

[3.1.3 全链路压测 17](#_Toc8263)

[3.2 过载保护：限流与熔断 18](#_Toc21697)

[3.2.1 限流的两种维度【不让别人调用】 18](#_Toc27779)

[限制系统的最大资源使用数 18](#_Toc15836)

[限制速率 18](#_Toc5988)

[3.2.2 单机限流的算法 18](#_Toc7109)

[漏桶算法 18](#_Toc11945)

[令牌桶算法 18](#_Toc23829)

[3.2.3 单机限流的实现 19](#_Toc1784)

[限制请求队列的长度【网络模型/漏桶算法】 19](#_Toc27270)

[RateLimiter【令牌桶算法】 19](#_Toc23081)

[基于时间窗口的统计限流 19](#_Toc17037)

[3.2.4 中央限流 19](#_Toc4331)

[3.2.5 熔断【不去调别人】 20](#_Toc18147)

[熔断的两种策略 20](#_Toc23222)

[基于请求失败率熔断 20](#_Toc29499)

[基于请求响应时间熔断 21](#_Toc3620)

[3.3 超时与重试 21](#_Toc11953)

[超时时间设置原则 21](#_Toc23840)

[重试原则 22](#_Toc13759)

[3.4 隔离 22](#_Toc915)

[数据隔离--根据数据重要性隔离 22](#_Toc18608)

[机器隔离--吧调用者根据重要性隔离 22](#_Toc3192)

[线程池隔离--业务复杂的服务调用 22](#_Toc26741)

[3.5 有损服务与业务降级 22](#_Toc8638)

[有损服务服务降级要比无服务要好的多 22](#_Toc10869)

[特殊情况下的有损服务和业务降级 22](#_Toc21706)

[微信QQ保障文字通信功能 23](#_Toc8659)

[推荐系统宕机显示静态列表 23](#_Toc19019)

[大促降级导出、批量修改主数据服务 23](#_Toc1814)

[3.6 灰度公测发布、备份、回滚 23](#_Toc16206)

[灰度-公测-线上发布机制 23](#_Toc22348)

[部署包/版本回滚 23](#_Toc26637)

[功能切量回滚--切量开关 23](#_Toc7049)

[3.7 监控体系与日志报警 23](#_Toc16395)

[监控体系 23](#_Toc1301)

[资源监控-CPU内存等 23](#_Toc4085)

[系统监控 24](#_Toc19532)

[SLA核心接口稳定性监控 24](#_Toc32662)

[数据库慢SQL、Redis慢查询 24](#_Toc28478)

[JVM的GC回收频率、时间 24](#_Toc31841)

[业务监控--订单下单成功率等 24](#_Toc5778)

[日志报警 24](#_Toc16739)

[四. 高可用问题及解决方案 24](#_Toc23095)

[4.1 高可用架构的5个核心问题 24](#_Toc29587)

[4.1.1 如何实现故障探测--周期性心跳探测 24](#_Toc11496)

[场景1：Master进程宕机 24](#_Toc7645)

[场景2：Master所在机器宕机 24](#_Toc11410)

[场景3：Master进程暂时僵死，导致master与其他模块间没了通信 24](#_Toc29282)

[场景4：Master进程正常，只是网络故障或抖动，没有心跳 24](#_Toc18200)

[场景1和2是真正宕机，3和4会发生错误决策、误报 24](#_Toc8886)

[MHA如何解决探测--加强探测链路，减少误探概率 25](#_Toc24074)

[MHA-Master,MHA-SLAVE-MASTER两条链路一起检测 25](#_Toc28463)

[4.1.2如何解决脑裂问题 25](#_Toc10573)

[方案1：隔离Fencing--对旧master进行隔离，杀死或不提供服务 25](#_Toc19836)

[方案2：新纪元epoch标识，旧master发现自己旧纪元，主动退位 25](#_Toc22469)

[方案3：租约机制--协调器与新master签租约后旧master失效 25](#_Toc18715)

[MHA方案--隔离,SSH登录master杀死进程 25](#_Toc9849)

[4.1.3 如何做到数据一致性 26](#_Toc5230)

[弱一致性--允许部分数据丢失 26](#_Toc10710)

[最终一致性--切换后，部分数据最终会一致 26](#_Toc14199)

[强一致性--切换后，数据立刻会读到【同步复制】 26](#_Toc8692)

[MHA--使用Mysql的数据同步机制 26](#_Toc29069)

[允许数据丢失--使用异步复制 26](#_Toc14854)

[不允许数据丢失--使用半同步复制 26](#_Toc19837)

[4.1.4 如何做到对客户端透明无察觉 26](#_Toc21606)

[VIP--虚IP漂移，不支持跨网段、跨机房切换 26](#_Toc9206)

[DNS 26](#_Toc20436)

[客户端维护IP列表，自己做健康探测，客户端自己切换 26](#_Toc26173)

[名字服务-如微服务注册中心 26](#_Toc22502)

[增加服务代理层--代理层统一切换 26](#_Toc8634)

[MHA--不解决，客户端自己解决，VIP或DNS 26](#_Toc3289)

[4.1.5 如何解决高可用依赖的连环嵌套问题 27](#_Toc16227)

[为了解决A系统的高可用，引入B系统，那如何解决B系统的高可用 27](#_Toc16959)

[MHA--未解决嵌套高可用问题 27](#_Toc18354)

[4.2 接入层高可用 27](#_Toc13557)

[4.2.1DNS高可用--广域网负载均衡GSLB--跨城容灾 27](#_Toc15683)

[4.2.2网关高可用--局域网负载均衡--F5/LVS 27](#_Toc6467)

[大型公司--F5 + (KEEPALIVED+VIP)+LVS或者自己开发网关 27](#_Toc14073)

[中小型公司 (KEEPALIVED+VIP)+LVS 27](#_Toc22378)

[4.2.3 Nginx高可用 27](#_Toc7499)

[4.2.4 Tomcat高可用 27](#_Toc23102)

[4.3 业务逻辑层高可用--微服务框架自带 28](#_Toc11027)

[4.4 存储层高可用 28](#_Toc18234)

#### 一.分布式架构要解决的4大核心问题

* 高并发
* 高可靠
* 高可用
* 数据一致性

**互联网三大变现模式：**

* 广告
* 电商
* 游戏

#### 高并发问题及解决方案

**让系统可以支撑大规模用户的高并发访问**

##### 2.1 系统分类

###### 2.1.1 分类依据

从三个角度来看系统读与写两端的差异：

* **数量级**：比如百度的读用户可能是亿或数十亿数量级，而写可能是百万或千万级别
* **响应时间**：读的一端通常要求在毫秒级，最差1~2内返回结果；而写则可能会几分钟或几天
* **频率**：读的频率远比写的频率高。对于C端用户来说，可能一个人几分钟就会搜索一次，但网页发布者可能几天才能发布一篇文章或博客。

###### 2.1.2 侧重于高并发读的系统

**搜索引擎**，比如百度、谷歌、、、

**电商的商品搜索**

**电商的商品描述、详情等**

###### 2.1.3 侧重于高并发写的系统

**广告计费系统**，C端用户每次点击或浏览广告，都会计入广告计费系统，然后从广告客户账户里扣费，这种扣费要尽可能实时，如果扣慢了，广告主的账户里明明没有钱了，但广告仍然线上播放，会造成平台流量的损失。

###### 2.1.4同时侧重于高并发读与写的系统

**电商的库存系统与秒杀系统**

**支付系统和微信红包**

**IM、微博和朋友圈**

无论在读还是写的一端，都面临高并发的压力。

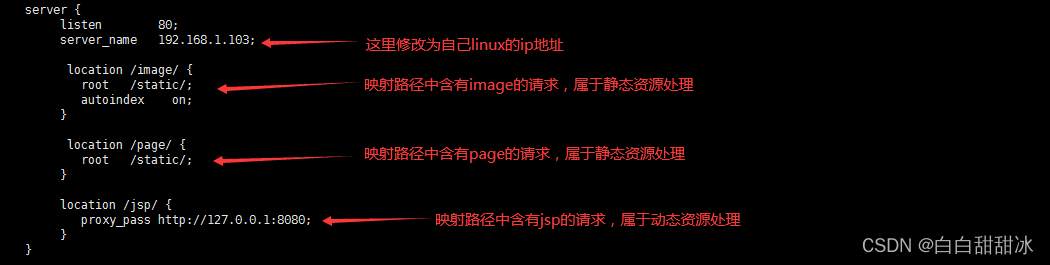
##### 2.2 高并发读解决方案

###### 2.2.1 策略1：动静分离与CDN加速

静态内容：图片、html、js、css、视频、、、、

Nginx作为静态资源服务器

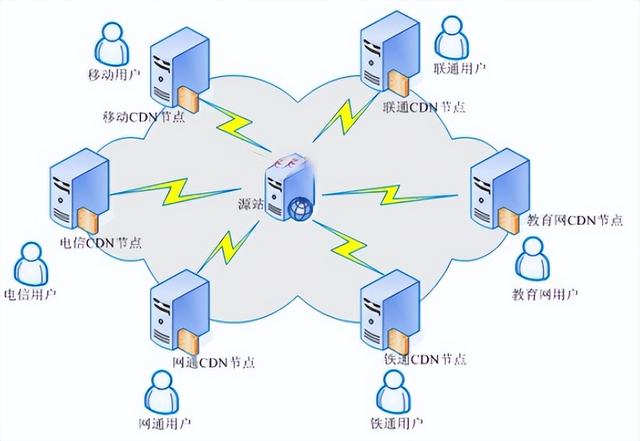
Nginx除了反向代理之外，还可以作为静态资源服务器



nginx 的动静分离，指的是由 nginx 将客户端请求进行分类转发，静态资源请求（如html、css、图片等）由静态资源服务器处理，动态资源请求（如 jsp页面、servlet程序等）由 tomcat 服务器处理，tomcat 本身是用来处理动态资源的，同时 tomcat 也能处理静态资源，但是 tomcat 本身处理静态资源的效率并不高，而且还会带来额外的资源开销。利用 nginx 实现动静分离的架构，能够让 tomcat 专注于处理动态资源，静态资源统一由静态资源服务器处理，从而提升整个服务系统的性能 。

CDN加速

一个静态文件缓存到全网各个节点，当第一个用户访问时，离用户最近的节点还没有缓存数据，CDN就去源系统抓取文件并缓存到该节点。当第二个用户访问时，只需要从这个节点访问即可，不需要再访问原系统。



###### 2.2.2 策略2：加缓存

**缓存的本质**：以空间 换时间

本地缓存或Memcached/Redis集中式缓存

**缓存的数据结构**： <K,V>

**缓存的更新方式**：

* 主动更新：库数据发生变化则主动更新或删除缓存
* 被动更新：当用户查询请求到来，缓存过期，则再更新缓存

**缓存存在的三大问题**：

**缓存穿透**。当用户发起请求的数据在缓存和数据库中都不存在时，会发生缓存穿透问题。解决方案包括：1. 验证拦截。在接口层进行校验，如用户权限和ID等基础校验，拦截无效请求。2. 缓存空数据。当数据库查询结果为空时，将查询结果存入缓存，并设置较短的有效期，以免影响正常数据的缓存。3. 使用布隆过滤器。布隆过滤器是一种特殊的数据结构，能在O(1)时间内判断一个数据是否存在，从而在缓存中过滤掉不存在的数据。12345

**缓存击穿**。当缓存中的数据过期，而数据库中有对应的数据时，会发生缓存击穿问题。解决方案包括：1. 加互斥锁。在并发请求中，只有第一个请求线程能获取锁并执行数据库查询操作，其他线程等待直到第一个线程完成查询并更新缓存。2. 热点数据不过期。将热点数据的缓存设置为不过期，并由定时任务异步加载数据，更新缓存。5

**缓存雪崩**。当大量缓存数据同时过期，导致大量查询请求直接访问数据库，引起数据库压力过大甚至崩溃，发生缓存雪崩问题。也可以是缓存宕机引起。解决方案包括：1. 过期时间打散。为缓存设置带有随机值的过期时间，避免大量缓存数据同时过期。2. 使用Hystrix。Hystrix是一款开源的“防雪崩工具”，通过熔断、降级、限流等手段降低雪崩发生后的损失 3.使用Redis Cluster与Sentinel模式保障高可用。

###### 2.2.3 策略3：库读写分离

京东的商品库做的读写分离，一个master可以多接几个slave机器来分担压力。

注意：这种只适合与没有分库的场景。

###### 2.2.4 策略4：并发读与Pipeline

多个异步RPC并行调用

现在的RPC框架都支持异步RPC，如果有三个RPC接口调用，耗时分别T1,T2,T3，且3个调用之间没有耦合关系。

Dubbo基于 NIO 的非阻塞实现并行调用，客户端不需要启动多线程即可完成并行调用多个远程服务，相对多线程开销较小。



冗余请求

Google公司测试，可仅用2%的额外请求将系统99.9%的请求响应从1800ms降低到74ms

假设100台服务器，其中有1台有延时故障

客户端首先给服务端发送一个请求，并等待服务端返回的响应。如果客户端在一定时间【定义为95%的请求标准响应时间】内没有收到客户端的响应，则马上给另一台或多台服务器发送同样的请求。客户端在等待第一个响应到达之后，立刻终止其他请求的响应。

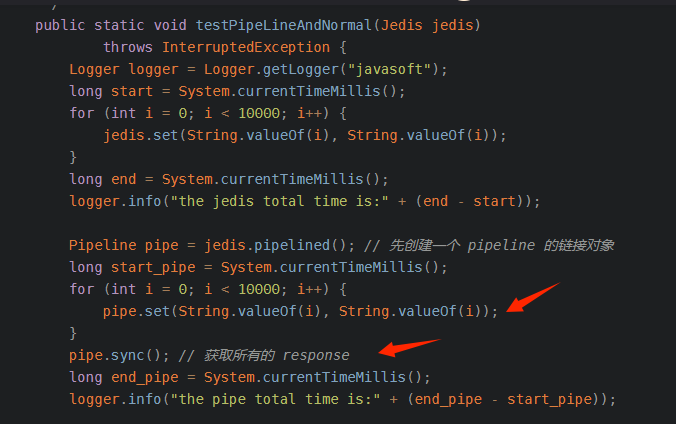
缓存Pipeline管道

Pipeline 在某些场景下非常有用，比如有多个 command 需要被“及时的”提交，**而且他们对相应结果没有互相依赖，对结果响应也无需立即获得**，那么 pipeline 就可以充当这种“批处理”的工具；而且在一定程度上，可以较大的提升性能，性能提升的原因主要是 TCP 连接中减少了“交互往返”的时间。

**适合批量处理且可靠性要求不高的场景**

**可靠性要求不高的场景**：有些系统可能对可靠性要求很高，每次操作都需要立马知道这次操作是否成功，是否数据已经写进 redis 了，那这种场景就不适合。

**短信群发**： 还有的系统，可能是批量的将数据写入 redis，允许一定比例的写入失败，那么这种场景就可以使用了，比如10000条一下进入 redis，可能失败了2条无所谓，后期有补偿机制就行了，比如短信群发这种场景，如果一下群发10000条，按照第一种模式去实现，那这个请求过来，要很久才能给客户端响应，这个延迟就太长了，如果客户端请求设置了超时时间5秒，那肯定就抛出异常了，而且本身群发短信要求实时性也没那么高，这时候用 pipeline 最好了。



需要注意到是用 pipeline 方式打包命令发送，redis 必须在处理完所有命令前先缓存起所有命令的处理结果。打包的命令越多，缓存消耗内存也越多。所以并不是打包的命令越多越好

不过在编码时请注意，pipeline 期间将“独占”链接，此期间将不能进行非“管道”类型的其他操作，直到 pipeline 关闭；如果你的 pipeline 的指令集很庞大，为了不干扰链接中的其他操作，你可以为 pipeline 操作新建 Client 链接，让 pipeline 和其他正常操作分离在2个 client 中

###### 2.2.5 策略5：批量读

缓存批量读写

Redis的multi指令，原子性执行多个指令，减少网络传输，需要注意不能超出以太网MTU大小，否则在网络层还是需要多次传输



首先要明确，redis是单线程工作的；  
1.pipeline的原理是客户端缓冲所有命令，然后一次性发给服务端，目的是减少通信次数；  
2.pipeline不具备原子性，可能部分成功，与其他客户端可以并发执行；  
3.multi操作具备原子性；  
4.实测pipeline效率比multi效率高10倍左右；  
5.multi使用服务端缓冲，pipeline使用客户端缓冲；  
6.multi是一条条发送的；

multi相当于一个redis的transaction的，保证整个操作的原子性，避免由于中途出错而导致最后产生的数据不一致

Mysql库批量读写

Mysql批量查询可以减少网络传输次数，类似原理

###### 2.2.6 策略6：重写轻读

信息Feeds流

Feeds流是一个持续更新并展示给用户的信息流。它将用户主动订阅的若干消息源组合在一起形成内容聚合器，帮助用户持续地获取最新的订阅源内容。所以它通常具有千人千面的个性化特点。

Feeds最核心的能力在于聚合**【其实就是重写，写扩散，在写的时候把一些前置工作做好，当用户查询的时候就不许需要复杂查询】**。他会根据你的行为聚合出你想要的信息，例如：微博是通过你的关注列表了解你可能想要的信息源，而后以时间轴的形式聚合各种信息推给你。后来又出现了抖音的猜你喜欢，它不需要你的手动关注，而是根据你的阅览时长，点赞等信息生成你的用户画像，从而聚合你可能感兴趣的信息。朋友圈的Feeds流则是根据你的好友关系，从而聚合了你可能想要的信息。

比如微博，用户关心的有两个：自己发布的文章列表与关注好友的文章列表，按时间排序。

**一般思维**：

知道用户id，从微博发布表里可以很轻松获取自己发布的微博列表

但是如何获取关注好友的微博列表，而且还需要按时间排序：

1. 从关注关系表中获取此user\_id所有关注的好友ids
2. 然后使用in的方式从微博发布列表中获取所有userID在ids中的文章，之后重新按时间排序 select msg\_ids from Msg where user\_id in (followings) limit offset,count

第二步骤在大数据量的情况下非常耗时

**Feeds流思路**

以微博为例，每个用户都有自己的收件箱与发件箱，假设用户有1000个粉丝，发布1条微博后，只写入自己的发件箱就返回成功。

然后后台异步任务吧这条微博推送到1000个粉丝的收件箱，也就是写扩散。这样每个用户在读取Feeds流的时候就需要在实时进行聚合了，直接读取各自的收件箱即可。

这也即是重写轻读，吧计算逻辑从读的一端转移到写的一端。

至于收件箱可以考虑使用Redis的哈希来实现，value是一个list，key是user\_id,list是msg\_id,至于value不能是无限增长的，在后续第三章案例实战的Feeds流系统中再讲解。

宽表与搜索引擎

若没有分库，则可以通过做读写分离，增加slave的机器来提升读性能。

但是对于已经分库，就需要做宽表，提前吧要关联的数据算好后保存在宽表中，比如Mysql宽表、搜索引擎，可以轻松 实现排序与分页查询功能。

**数据同步的三种策略：**

* 定时任务
* 数据变更通过消息中间件
* Binlog监听同步

也是重写轻读的一种体现

2.2.7 总结

无论是加缓存、动静分离、还是重写轻读，本质上还是CQRS(Command Query Responsibility Separation)读写分离架构。

分别为读和写设计不同的数据结构，在C端同时面临读和写的高并发压力时，吧系统分成读和写两个视角来设计，各自设计适合高并发读和写的数据结构与模型。

读和写有延迟，保障最终一致性，而不是强一致性，并不影响业务。

以库存系统为例，假设库存页面显示10件，但是实际可能9件（有个用户下单扣减1）或11件（有个用户取消订单返回1），但是在后来 用户进行下单操作的时候，会去实时扣减数据库的9件，变成8件，这是没什么问题的，只不过页面显示有1s的延迟。

##### 2.3高并发写解决方案

###### 2.3.1 策略1：数据分片

即对要处理的数据或请求**分成多份**然后**并行处理**

数据库的分库分表

分表后，数据还在一个数据库、一个机器上，可以充分利用CPU、内存等资源。

分库后，可以利用多台机器的资源。

JDK的ConcurrenHashMap实现

ConcurrentHashMap在内部分了诺干槽【个数是2的n次方，默认是16个槽】，即诺干个HashMap，这些槽可以并发的读和写，槽之间是独立的，不会发生数据互斥。

以后设计数据结构可以参考。

Kafka的Partition

Kafka的一个topic在物理空间上会被分成多个Partition分区。Topic在逻辑上可以被认为是一个queue【这里与RocketMQ不同，rocketmq一个topic可以有多个队列，kafka是一个topic就是一个队列】，每条消息都必须指定它的topic，可以简单理解为必须指明把这条消息放进哪个queue里。为了使得Kafka的吞吐率可以水平扩展，物理上把topic分成一个或多个partition，每个partition在物理上对应一个文件夹，该文件夹下存储这个partition的所有消息和索引文件。每个Partition对应磁盘中的一个日志文件，Partition之间相互独立，可以并发的读写，提高Topic的一个并发量。

ES搜索引擎的分布式索引/分片存储

ES中有一个基本策略是分布式索引。比如有10亿个网页和商品，如果建立在一个倒排索引中，则索引会很大

可以吧这10亿个网页或商品分成n份，建立n个小的索引。

当一个请求传来的时候，可以并行的在n个小的索引上查询，然后将 查询结果合并返回。

Redis分片存储

###### 2.3.2策略2：任务分片

数据分片是对要处理的数据/请求进行分片。

**任务分片是对处理程序本身进行分片，其实就是拆分任务并发处理。**

CPU指令流水线

有4条指令，T1,T2,T3,T4

并行执行相较于串行执行的加速比：

（T1+T2+T3+T4）/Max（T1，T2,T3,T4）

Map/Reduce大数据

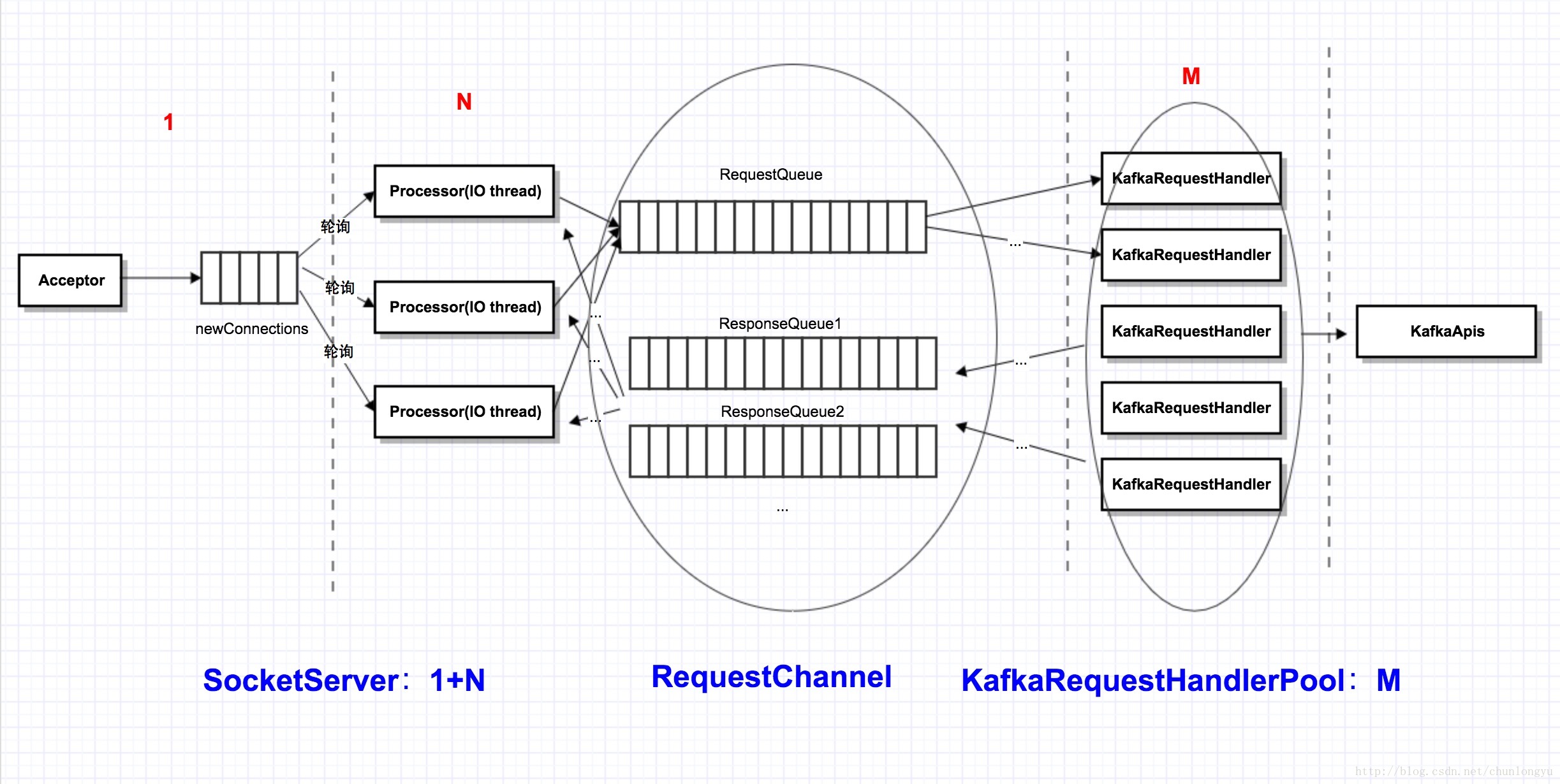
Google公司的Map/Reduce是一种将数据分片与任务分片相结合的典型案例。

1+N+M网络模型（Netty、Tomcat）

1个监听线程监听客户端的Socket连接

N个IO线程处理对Socket读写

M个worker工作线程负责对请求进行逻辑处理



进一步来讲，woker线程还可以拆分为解码、业务逻辑计算、编码等环节，进一步提升并发度。

多线程

注意两点：

* 锁竞争
* 线程切换开销

使用线程池，比如导单逻辑

###### 2.3.3 策略3：异步化

业务层面：轮询+消息中间件

轮询：同步接口+服务器后台任务+客户端轮询

消息中间件：

发送验证码

先存入消息队列，然后调用公网http第三方短信平台发送验证码【调用公网HTTP耗时不定，使用消息中间件，即使tp99超时也不会影响业务】

电商的拆合单

拆单：客户只下一次订单，付款1次，会直接返回接单成功，之后电商平台会在后台使用拆合单引擎进行拆合单处理

广告计费系统

C端用户点击或浏览会被落盘到消息队列持久化

后续的扣费及其他处理都是异步化操作

属于典型的流式计算模型

接口层面：异步HTTP\RPC\MYSQL接口

Apache的AsyncHttpClient

<dependency>

<groupId>org.apache.httpcomponents</groupId>

<artifactId>httpasyncclient</artifactId>

<version>4.1.4</version>

</dependency>

final HttpGet request2 = new HttpGet("http://www.apache.org/");

httpclient.execute(request2, new FutureCallback<HttpResponse>() {

public void completed(final HttpResponse response2) {

latch1.countDown();

System.out.println(request2.getRequestLine() + "->" + response2.getStatusLine());

}

异步RPC

前面已经提过，dubbo的异步RPC

异步Mysql

在java中，JDBC没有提供异步接口。若想要实现mysql的异步调用，需要自己实现mysql的c/s协议。

或者使用其他语言，比如nodejs是有异步调用mysql的接口的。

JDK与操作系统层面：NIO/AIO

异步落盘-延时写入

当客户端写入一个KV数据时候，只需写一条日志【日志文件】+一个内存操作【写入SortedHashMap】，即可告诉客户端写入成功了，实际此时数据并未正式落盘。之后，同步异步任务合并写入磁盘。

如果把数据库看作一个图书馆，B+树就像一个勤快管理员，它总是为新写入的记录找到唯一的正确位置来存放，查找这个记录的时候，只到这个唯一正确位置找就行了。

LSM树就是一个懒惰管理员，新写入的记录暂时存放到内存里。内存里面新记录数量足够多的时候，再把这些记录统一刷写到硬盘上的数据文件中上去。

1）LSM树（Log-Structured MergeTree），日志结构合并树。

2）LSM树存储引擎和B+树存储引擎一样，同样支持增、删、读、改、顺序扫描操作。而且通过批量存储技术规避磁盘随机写入问题。当然凡事有利有弊，**LSM树和B+树相比，LSM树牺牲了部分读性能，用来大幅提高写性能。**

3）LSM树核心思想的核心就是放弃部分读能力，换取写入的最大化能力。LSM Tree ，这个概念就是结构化合并树的意思，它的核心思路其实非常简单，**就是假定内存足够大，因此不需要每次有数据更新就必须将数据写入到磁盘中，而可以先将最新的数据驻留在内存中，等到积累到足够多之后，再使用归并排序的方式将内存内的数据合并追加到磁盘队尾(因为所有待排序的树都是有序的，可以通过合并排序的方式快速合并到一起)。**

4）日志结构的合并树（LSM-tree）是一种基于硬盘的数据结构，与B+tree相比，能显著地减少硬盘磁盘臂的开销，并能在较长的时间提供对文件的高速插入（删除）。然而LSM-tree在某些情况下，特别是在查询需要快速响应时性能不佳。通常LSM-tree适用于索引插入比检索更频繁的应用系统。

LSM树，log-structured，日志结构的，日志是软件系统打出来的，而且系统写日志不会写错，所以不需要更改，只需要在后边追加就好了。各种数据库的写前日志也是追加型的，因此日志结构的基本就指代追加。注意他还是个 “Merge-tree”，也就是“合并-树”，合并就是把多个合成一个

###### 2.3.4 策略4：批量

Kafka百万QPS写入

**Rocketmq与kafka：**

首先**都是顺序写入**，不过 RocketMQ 是把消息都存一个文件中，而 Kafka 是一个分区一个文件。

每个分区一个文件在迁移或者数据复制层面上来说更加得灵活。

但是分区多了的话，写入需要频繁的在多个文件之间来回切换，对于每个文件来说是顺序写入的，但是从全局看其实算随机写入，并且读取的时候也是一样，算随机读。而就一个文件的 RocketMQ 就没这个问题。

**Kafka可以达到百万QPS原因**：

* Partition分区数据分片
* 磁盘顺序写入
* 批量提取消息发送集群

Kafka客户端在内存中为每个Partition分区准备了一个队列，叫RecordAccumulator，Producer线程一条条的发送消息，都会先进入内存队列。然后有个Sender线程从这些队列中批量的提取消息发送给Kafka集群。

广告计费系统的合并计费

扣费模块一次性的从持久化消息队列中拉取多条消息，按照广告主的id进行分组，同一个组内的消息扣费金额进行累加合并，然后从账户数据库里扣除。

Mysql库的小事务合并机制

在MySQL中，小事务合并是一种优化策略，用于减少大量小事务带来的性能开销。这通常通过将多个小事务的提交延迟到一个较大的事务来实现。这样可以减少数据库的锁定和I/O操作次数，从而提高性能。

在多机房的数据库多活【跨数据中心的数据库复制】场景中，事务合并是加速数据库复制的 一个重要策略。

#### 高可靠问题及解决方案

**高可靠是为了让系统更加靠谱，尽可能减少故障的发生次数。尤其是在流量激增的情况下，如电商的大促秒杀、春节发红包等。**

##### 3.1 容量评估与扩容规划

###### 3.1.1 吞吐量、响应时间、并发数的关系

* 吞吐量：QPS\TPS，单位时间内处理的请求数。
* 响应时间：处理每个请求所需的时间，tp99
* 并发数：服务器同时并行处理的请求个数。

并发数 = 吞吐量TPS \* 响应时间TP99

比如CLPS系统的并发数 = 2000 \* 0.3 = 600并发数

###### 3.1.2 容量规划

容量规划，其实就是找出现状与目标的差距，找出差距之后，才能明确应对策略，包括扩容机器、限流等。

一般根据历史的流量曲线，来预估未来的流量。例如，电商双十一大促，去年双11的QPS是20W/s,预计今年涨一倍，就是40W/s，呐机器数量就按照40W/s和压测结果来进行扩容，如果QPS超过这个数值就需要进行限流了。

如何知道是按1.5倍，2倍来算呢，入口出的流量通常是业务团队综合各种商业因素给的一个经验值。

QPS需要用峰值来计算

###### 3.1.3 全链路压测

全链路压测涉及多个团队开发的服务，需要各个团队之间密切写作，制定完备的压测方案。

采用**线上压测**，因为测试环境很多中间件不可用不标准

**读接口压测：**

可以对线上流量进行回放，这没有问题。

**写接口压测：**

会对**线上数据库造成大量测试数据**，解决方案：

线上部署一个与真实数据库一样的“影子数据库”，对测试数据打标签【也可以通过租户隔离】，**测试数据进入影子数据库**。通常由数据库中间件来实现。

##### 3.2 过载保护：限流与熔断

经过容量规划之后，根据未来的预测容量进行扩容，但是没有人保证系统可以100%吹所有用户请求。预测做的再好，实际流量还是有可能大于预测流量，这就需要系统有过载保护功能。

如果系统没有过载保护，系统发生雪崩，则对所有的用户都不能提供服务。做了过载保护，拒绝了少量用户，确保为大多数用户提供正常服务。

###### 3.2.1 限流的两种维度【不让别人调用】

**限流是自己提供的服务设置，是对自己接口提供的一个保护措施。**

限制系统的最大资源使用数

Nginx的limit\_conn限制并发数

秒杀系统库存100件，只允许放500个人进来，对于其他人返回已售完。

限制速率

限制系统的QPS

速率限流分为单机限流和中央限流。

* 单机限流：一般成熟的RPC框架都有相应的限流配置，可以对接口进行限流。

另外，Guava的RateLimiter、Nginx的limt\_req模块，都是单机限流的工具。

* 中央限流：一般需要一个总的中央限流系统，设置一个总的QPS,然后将总的QPS动态的分摊到每台机器上。

###### 3.2.2 单机限流的算法

漏桶算法

漏桶算法保证数据包流出的速率恒定【流入的速率是不定的】

不管你来多少流量数据包，我都是这么一个速率流出

令牌桶算法

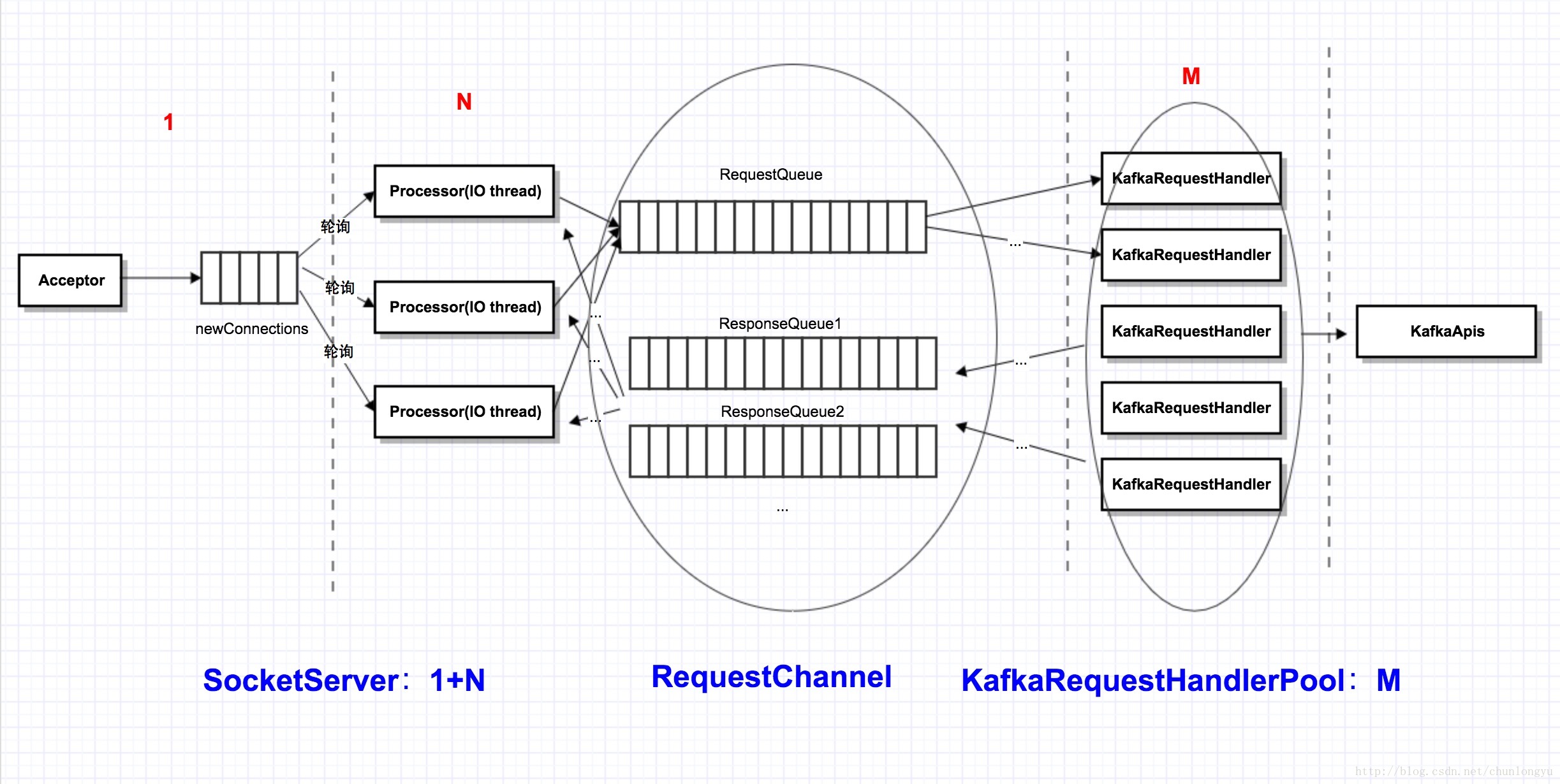
令牌桶算法是流入速率保持恒定，数据包需要从令牌桶里获取令牌

令牌桶产生令牌的速率是固定的，10分钟内就只能产生固定量的令牌数。

###### 3.2.3 单机限流的实现

限制请求队列的长度【网络模型/漏桶算法】

无论是Tomcat等web服务器，还是RPC框架，涉及到网络模型，一般都是1+N+M



IO线程收到请求之后，放入request队列，worker线程从队列中取出请求，进行业务逻辑处理，然后将结果放入response队列，再由IO线程返回给客户端。

通过设置请求队列的长度，可以达到限流的目的。

即当IO线程放入请求时，如果发现队列已满，则直接丢弃请求，然后告诉客户端拒绝服务，这种实现技术，也是漏桶算法的一种。

RateLimiter【令牌桶算法】

Google Guava 库的RateLimiter

比如1s内产生100个令牌permits

RateLimiter limiter = RateLimiter.create(100);

long start = System.currentTimeMillis();

for (int i= 0; i < 30; i++) {

double time = limiter.acquire();

则代表每隔10ms才会产生1个令牌，而且**令牌桶内如果长时间没有流量，只会保留1s内的100个令牌。**

如果积攒100个令牌，面对激增流量，有两种应对方案：

1. SmoothBursty：允许突发流量，可以一瞬间取走所有令牌。
2. SmoothWarmingUp:可以设置预热时间，就是慢慢消耗掉这些令牌。

基于时间窗口的统计限流

时间窗口算法统计每个时间段窗口的请求量，单位可以是1s，1min，5min等，然后吧统计值与阈值进行比较，如果统计值>阈值，则进行限流。

###### 3.2.4 中央限流

单机限流即每台实例维护自己的计数器，而集群限流则是共用一个中央模式的计数器；

单机限流有以下特点：【单机限流需要每台服务器的流量均衡】

1、会出现误限的情况，比如说有两台实例 A 和 B，每个单机限流阈值为 10，那么整体限流

阈值是 20，但是如果出现负载不均，某一秒 A 接收的请求是 15，B 接收的请求是 5，那么根

据单机阈值，A 将放行 10 个请求，B 放行 5 个请求，这一秒内实际只承接了 15 个请求，而

我们的期望是 20；

2、无法很好的设置精确的限流值，一般情况下，单机限流阈值 = 整体限流阈值 / 实例数。比

如说实例数有 50 个，但是想要 80 的限流值就无法精确匹配。 单机限流阈值设置为 1 的情况

下整体限流阈值只有 50 ，单机限流阈值设置为 2 的情况下整体限流阈值则达到了 100。

3、如果整体限流值不变，实例进行扩、缩容是单机限流阈值要跟着更新单机限流阈值；

**单机限流的缺点：**

**如果流量不均衡，限流1000，A服务器流量大，1200，B服务器流量小，800，则B服务器没吃饱，但是A服务器会有200个请求被限流。**

集群限流的特点：

1、使用中央模式的计数器，不会出现单机限流出现的误限、无法精确匹配限流值以及扩、缩

容调整问题，限流值比较准确；

2、依赖提供中央技术器的服务，如果该服务不可用，那么限流功能将不可用，此时可以考虑

降级到单机限流；

###### 3.2.5 熔断【不去调别人】

**限流是提供接口服务一方对自己的保护措施**

**熔断的调用方调用外部服务，调用方对自己的一种保护措施**

Fast Fail原则：一个调用端会调用很多服务，当其中某个服务响应很慢时，调用端所有的worker线程都在等待这个服务返回，所有请求的响应时间都会变慢，最终可能会吧调用端拖死。

熔断的两种策略

基于请求失败率熔断

对于客户端调用的某个服务，如果服务在短时间内大量超时或报错，则客户端直接开启熔断，也就是不再调用该服务。然后经过一段时间，再吧熔断解除，如果还不行，则继续开启熔断。

**以Hystrix为例【Hystrix仅支持以失败率熔断】**

CircuitBreaker.requestVolumeThreshold 滑动窗口的大小，默认20个

CircuitBreaker.sleepWindowInMilliseconds 过多长时间熔断器再次检测是否开启，默认5s

CircuitBreaker。errorThresholdPercentage 失败率，默认50%

每20个请求，当有50%失败时候，熔断器就会开启，直到5s后再次检测

基于请求响应时间熔断

阿里的Sentinel可以支持失败率与响应时间两种熔断策略，且sentinel还支持限流

当资源的平均响应时间超过阈值后，资源进入准降级状态。接下来如果收到5个请求，且他们的RT持续超过阈值，则进行熔断。



##### 3.3 超时与重试

###### 超时时间设置原则

**调用方的超时时间>=服务方的超时时间**

不然差异的时间没什么意义

在整个调用链路上，超时时间应该逐级递减，比如入口500ms，经过1级服务耗掉30ms，接下来的服务最多设置470ms

对于内部服务：

写服务，超时时间在500ms左右

读服务，超时时间在200ms左右

外部服务可以设置1s或更大

###### 重试原则

即使超时时间设置合理，因为网络抖动造成偶尔超时也很常见，此时就需要重试，一般重试3次即可。

但需要注意，不要在调用链的最上层设置重试，会导致底层的服务压力变大。

##### 3.4 隔离

隔离是将系统或资源分开，在系统发生故障时能限定传播范围和影响范围，即发生故障后不会产生滚雪球效应，从而吧故障的影响限定在一个范围内。

###### 数据隔离--根据数据重要性隔离

维度：数据的重要性，非常重要、次重要、不重要

在数据库存储中，将这些数据所在的物理库彻底分开。

###### 机器隔离--吧调用者根据重要性隔离

一个服务，可以吧调用者进行一个重要性排序，对于核心几个调用者，可以为其专门准备一组机器，提供一个专门的别名供其调用。

成熟的RPC框架都带有隔离功能，根据调用方的标识ID，把来自某个调用方的请求都发送到某组固定的机器中。

###### 线程池隔离--业务复杂的服务调用

如果某个服务的接口耗时一直很长，为了不拖累整体RPC的调用，可以为其单独创建一个线程池来进行调用，不会一项其他接口。

##### 3.5 有损服务与业务降级

###### 有损服务服务降级要比无服务要好的多

###### 特殊情况下的有损服务和业务降级

特殊情况：大促、网络故障、应用宕机

微信QQ保障文字通信功能

文字通信、语音通信、视频通信，对带宽的要求越来越高。当网络发生故障，视频通信不可用，但可保证语音通信、文字通信可用。会最大努力提供服务，哪怕是有损服务，也比完全不提供服务强。

推荐系统宕机显示静态列表

推荐系统宕机时，不能显示502.应该显示一个静态页面，这个静态商品页面可以缓存在客户端或者一个非常简单可靠的后备系统中。虽然没有了个性化，但至少用户能看到东西，还可以购买商品。

大促降级导出、批量修改主数据服务

##### 3.6 灰度公测发布、备份、回滚

频繁的系统变更会导致系统不稳定，既然无法避免系统变更，能做的就是让整个过程尽可能平滑、受控，这就是灰度与回滚策略。

###### 灰度-公测-线上发布机制

使用租户id进行流量划分

###### 部署包/版本回滚

当上线新功能出现问题，回滚到上一个版本部署包

###### 功能切量回滚--切量开关

新功能做切量开关，通过配置或缓存开启或关闭

##### 3.7 监控体系与日志报警

###### 监控体系

资源监控-CPU内存等

CPU、内存、硬盘、网络等，资源监控是一件相对标准化的事情，开源软件有Zabbix等，大一些的公司会有运维团队或基础架构团队来搭建专门系统监控。

系统监控

SLA核心接口稳定性监控

数据库慢SQL、Redis慢查询

JVM的GC回收频率、时间

业务监控--订单下单成功率等

业务监控要监控哪些业务指标，需要根据具体业务分析

比如订单系统，比如下单成功率、订单履约时间

###### 日志报警

日志可以用来出现问题定位问题，还可以用来主动报警

日志级别、日志漏打

#### 高可用问题及解决方案

**高可用是为让故障发生之后的修复时间尽可能的短。**

Failover故障转移，冗余备份，主从切换，需要解决以下几个问题

##### 4.1 高可用架构的5个核心问题

###### 4.1.1 如何实现故障探测--周期性心跳探测

场景1：Master进程宕机

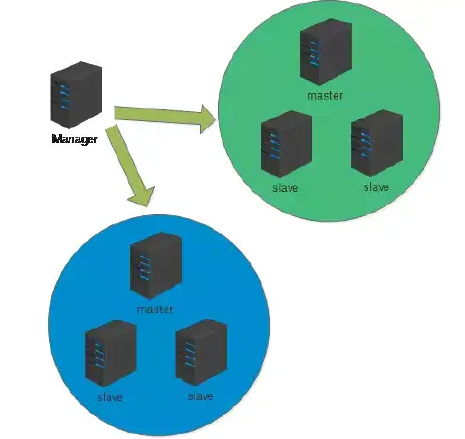
场景2：Master所在机器宕机

场景3：Master进程暂时僵死，导致master与其他模块间没了通信

场景4：Master进程正常，只是网络故障或抖动，没有心跳

场景1和2是真正宕机，3和4会发生错误决策、误报

MHA如何解决探测--加强探测链路，减少误探概率



MHA是Mysql高可用方面的一个成熟方案

在每台Mysql上都安装一个agent，它是一个独立进程，称为MHA NODE。这个进程会监听自己机器上运行的master或slave是否宕机

由MHA Manager来决定是否要切换

MHA-Master,MHA-SLAVE-MASTER两条链路一起检测

仍然不可能做到100%精确探测

###### 4.1.2如何解决脑裂问题

如果发生误报，旧master还活着，有些客户端已经切换到新master，新旧master同时存在就会发生脑裂问题。同时写数据，就会发生错乱。

方案1：隔离Fencing--对旧master进行隔离，杀死或不提供服务

具体的隔离技术，不同系统不一样

方案2：新纪元epoch标识，旧master发现自己旧纪元，主动退位

方案3：租约机制--协调器与新master签租约后旧master失效

MHA方案--隔离,SSH登录master杀死进程

在新msater上位之前，会对旧master执行隔离操作，确保旧master一定死了。通过SSH登录旧master，执行shutdown脚本。

但也有可能SSH登录不成功，强杀命令失败

###### 4.1.3 如何做到数据一致性

Master与slave是异步复制的，有一定的延迟。主从切换后，有一部分数据会丢失。

数据一致性的方案，取决与业务场景对数据一致性有多高的要求。

弱一致性--允许部分数据丢失

如分布式缓存

最终一致性--切换后，部分数据最终会一致

强一致性--切换后，数据立刻会读到【同步复制】

MHA--使用Mysql的数据同步机制

允许数据丢失--使用异步复制

不允许数据丢失--使用半同步复制

在Master宕机后，需要让Slave上的binlog全部重放完毕，然后吧Slave切换成master

###### 4.1.4 如何做到对客户端透明无察觉

客户端连接是master的ip，切换之后，客户端如何感知到改变，并切换到新的IP

VIP--虚IP漂移，不支持跨网段、跨机房切换

DNS

客户端维护IP列表，自己做健康探测，客户端自己切换

名字服务-如微服务注册中心

增加服务代理层--代理层统一切换

MHA--不解决，客户端自己解决，VIP或DNS

使用DNS需要注意，DNS有层层缓存，新Master IP的生效，需要到秒级或分钟级

###### 4.1.5 如何解决高可用依赖的连环嵌套问题

为了解决A系统的高可用，引入B系统，那如何解决B系统的高可用

MHA--未解决嵌套高可用问题

##### 4.2 接入层高可用

###### 4.2.1DNS高可用--广域网负载均衡GSLB--跨城容灾

GSLB--Global Server Load Balance，由网络运营商移动/电信/联通提供的DNS层提供服务

双机房，北京与上海两个IDC(Internet Data Center网络数据中心)机房，各自有一个公网入口IP

将两个公网IP配置到同一个DNS服务，客户端使用DNS域名访问服务器

GSLB可实现，北方用户通过DNS域名访问时，获取到的是北京IDC入口IP，南方获取的是北京IDC入口IP。

DNS系统本身是层层缓存的，可以保证高可用。

###### 4.2.2网关高可用--局域网负载均衡--F5/LVS

请求通过公网IP进入某个IDC后，接下来旧需要接入网关，把来自公网IP的流量分摊到IDC内网多台机器。

大型公司--F5 + (KEEPALIVED+VIP)+LVS或者自己开发网关

中小型公司 (KEEPALIVED+VIP)+LVS

###### 4.2.3 Nginx高可用

请求经过网关，进入内网，接下来可能接入Nginx

多台Nginx配置到网关的路由配置

###### 4.2.4 Tomcat高可用

Nginx的upSTREAM

##### 4.3 业务逻辑层高可用--微服务框架自带

SpringCloud、Dubbo

##### 4.4 存储层高可用

存储层高可用是最难的，因为涉及到数据一致性。

4.4.1 Redis Cluster高可用

4.4.2 HDFS--分布式存储高可用

4.4.3 MHA--数据库高可用

#### 五.DDD领域驱动设计

##### DDD的目的

###### 关注业务，减少技术因素

###### 统一语言解决业务拆分方法和实现标准

架构--开发--产品--用户，大家统一的标准语言

###### 打破沟通壁垒

###### 代码即文档，无需编写文档

###### 大型公司可以搞，不适用小型公司

###### 对查询不友好，可以用于项目实验

###### 20年都没普及

##### MVC架构与DDD的区别

###### MVC是面向过程/数据的喷血模型--实体类model只有属性无行为，目前主流

数据与业务逻辑分离在不同层级



基于数据驱动容易造成贫血失忆症--业务逻辑重复编写

业务层逐渐复杂，可能以前写的业务之后忘记然后在另一个service重写一遍

###### DDD是充血模型--model既有行为也有属性--基于业务驱动

数据与业务放置在一起