<https://www.csdn.net/article/2015-03-09/2824135>

<https://www.cnblogs.com/daviddu/articles/7668240.html>

<https://www.cnblogs.com/davidwang456/articles/5360292.html> 到中间“**遵循容器单进程**”后没有看

<https://blog.csdn.net/j3t9z7h/article/details/78800820>

<https://blog.csdn.net/verifocus/article/details/78771067> 几种服务降级

维持高并发、高可用、海量数据（社交网络大量图片、百度大量离线网页的快照）的迅速（促销时候的库存问题）读写与内容挖掘（淘宝的商品推荐）

机房内部与跨机房的数据同步方式不同，后者需保证数据传输的可靠性

<https://www.cnblogs.com/neal-ke/p/8966971.html>

Hessian二进制RPC协议，远程HTTP工具，效率比Protobuf稍微低一点

Protobuf Google的二进制序列化程序

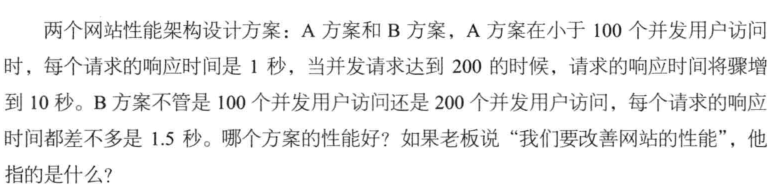
Thrift Facebook的服务注册、远程调用

Dubbo 阿里的分布式服务框架

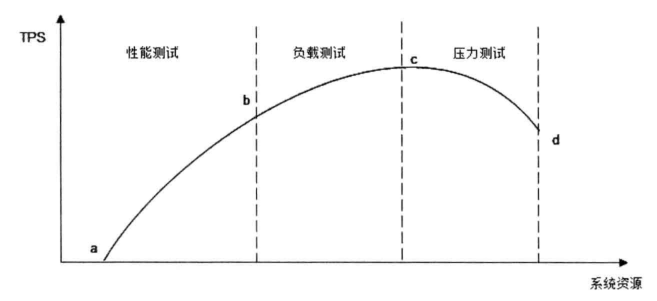
Cobar 阿里的分布式数据库中间件（支持数据的分库分表）

Lucene 垂直全文搜索引擎，用于企业内部搜索数据 Solr是Lucene扩展的工具

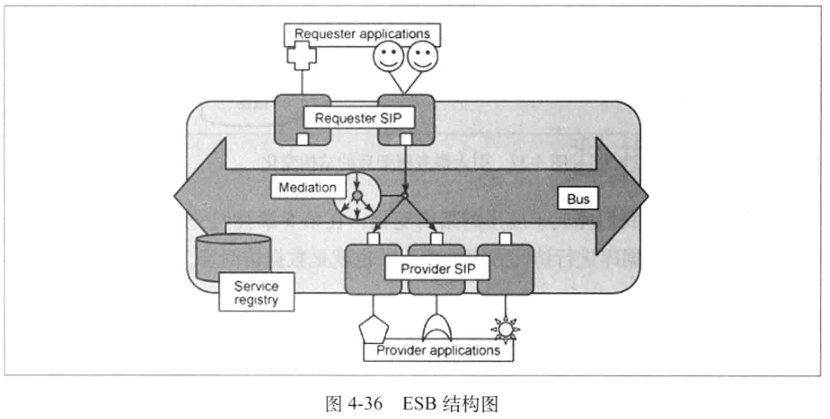
网站技术是为业务服务的，是否使用分布式服务需要根据业务发展规划改变。所以会出现渐进式的演变，使用了分布式之后又改回集中式。



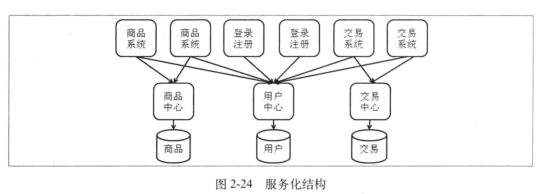
用户、开发人员、运维人员对网站的性能理解不同

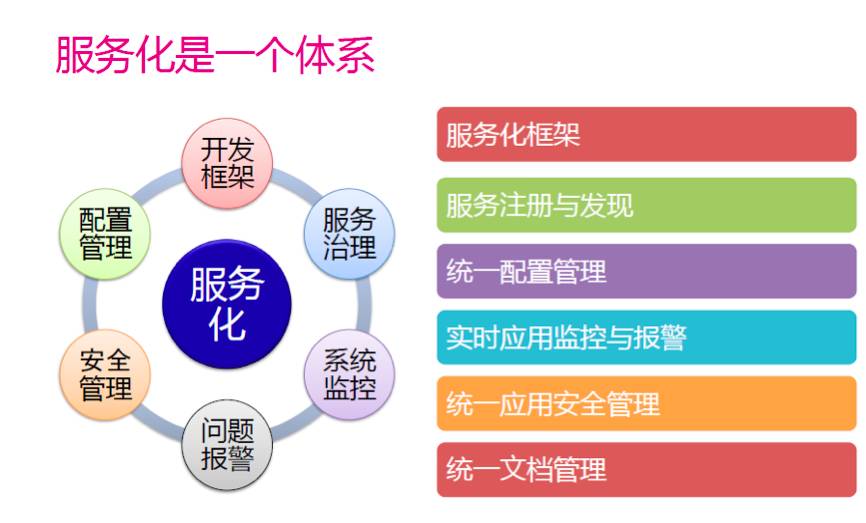


随着压力的增长，性能测试阶段错误率就已经不为0了（获取网站内容的时候会出错）。



ESB（企业服务总线）与微服务的区别：（1）总线式与点对点；（2）服务框架面向同构系统，ESB会更多考虑不同产商提供的服务的整合。



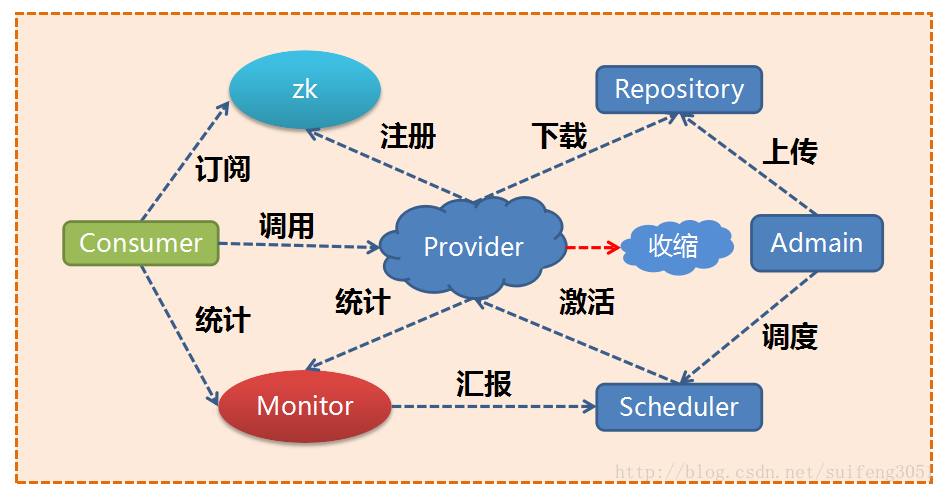


这种形式中文名字叫做“微服务”（微：服务的粒度要细，即服务要细化到API），通常使用 Rest Api，这个过程叫做RPC

**API网关**是一个服务器，是系统的唯一入口。从面向对象设计的角度看，它与外观模式类似。API网关封装了系统内部架构，为每个客户端提供一个定制的API。它可能还具有其它职责，如身份验证、监控、负载均衡、缓存、请求分片与管理、静态响应处理。API网关方式的核心要点是，所有的客户端和消费端都通过统一的网关接入微服务，在网关层处理所有的非业务功能。通常，网关也是提供REST/HTTP的访问API。服务端通过API-GW（API网关）注册和管理服务。它作为一个系统的后端总入口，承载着所有服务的组合路由转换等工作，好处是将结构简化了（类比kafka），如果生产者服务器发生变化，可以很方便又快速地更改。除此之外，一般也会把安全，限流，缓存，日志，监控，重试，熔断等放到 API 网关来做，那么可以试想在高并发的情况下，这里可能会出现一个性能瓶颈。

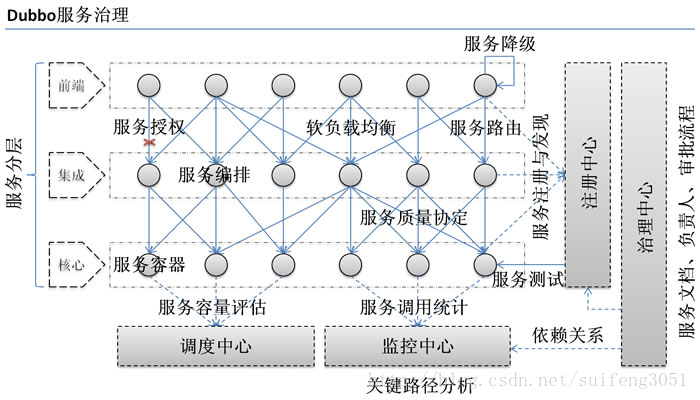
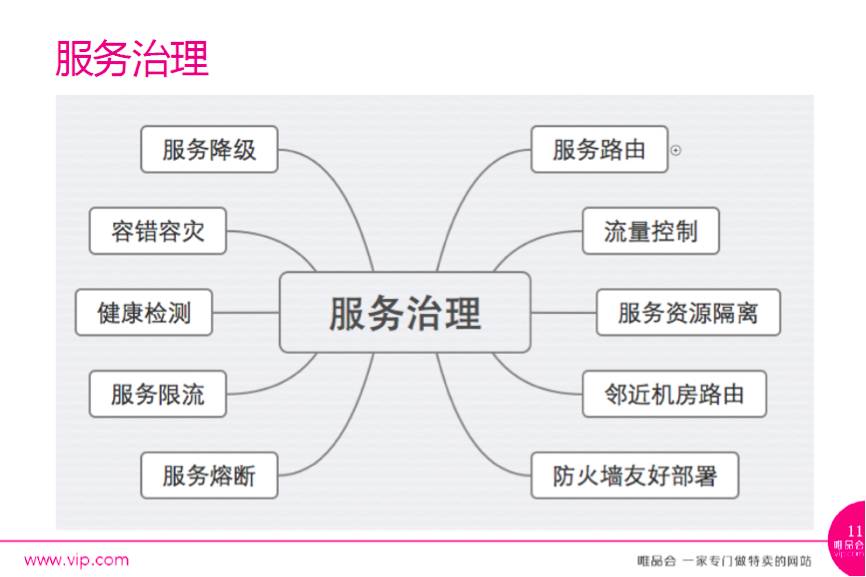
**基于接口的路由**：一个服务提供多个接口，但是有一个接口会比较耗时，所以要将耗时的与不耗时的分开路由到不同的服务器上。API网关路由可以细化到基于服务或者基于接口。

**弹性云**：保证API不能挂掉并且有好的性能。自动化运维解决方案，对访问压力进行监控，根据监控解决调度应用的发布和回收。

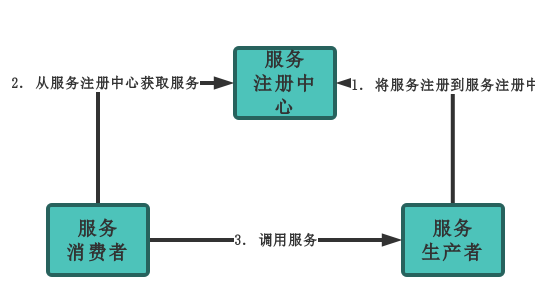


服务治理包含了所有服务相关的控制与相关信息的查看：

查看包含服务的编码、容量、质量（出错率）、依赖、分布、统计等



服务的注册与发现：



服务限流：

目的可能是因为AB测试、也可能是因为服务器的承受能力

CDN、负载均衡、使用消息队列、基于接口的路由都算是限流

限制总并发数（比如数据库连接池、线程池）、限制瞬时并发数（如nginx的limit\_conn模块，用来限制瞬时并发连接数）、限制时间窗口内的平均速率（如Guava的RateLimiter、nginx的limit\_req模块，限制每秒的平均速率）；其他还有如限制远程接口调用速率、限制MQ的消费速率。另外还可以根据网络连接数、网络流量、CPU或内存负载等来限流。

令牌桶算法： 类似于条件变量来限制，使用一个桶来装令牌，令牌可以是每过一定的时间段（限制流入请求）就会生成并在桶没有满的时候加入，如果满了就丢弃与删除。每一次操作需要从桶中取出并消耗一个令牌。允许一定程度的突发，可以一次拿很多个令牌。

漏桶算法:按照常量固定速率流出请求，流入请求速率任意，当流入的请求数累积到漏桶容量时，则新流入的请求被拒绝。

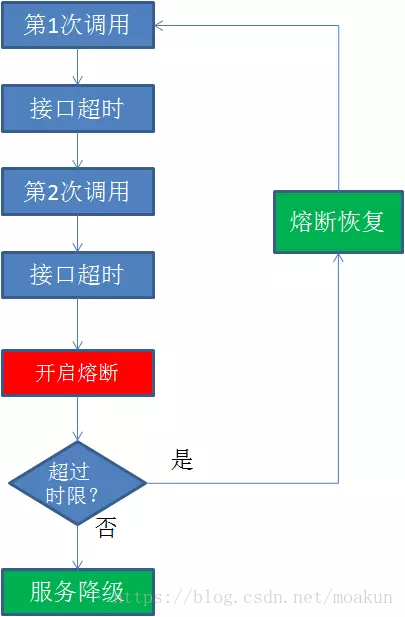
单位时间计数法：使用Memcachd或Redis对每一次访问进行计数统计-》滑动窗口计数法：一定时间段内（比如5个200ms），通过滑动窗口来统计

似乎说关键在于资源隔离，最后使用多个线程池（线程池之间资源隔离）

服务雪崩：

因为服务之间存在依赖关系，构成一张网状结构，如果一个下级服务出现问题会导致最后许多服务都不能使用。

服务熔断（服务隔离）： 为了拯救服务雪崩问题，牺牲局部来保全全局即熔断

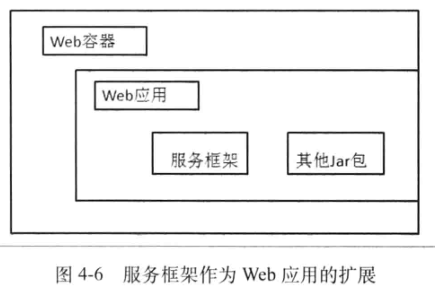


最后一次没有超时，说明是因为当前资源紧张造成，所以应该服务降级（熔断可以视为关闭服务的服务降级）

在进入熔断开启状态一段时间之后（Hystrix默认是5秒），熔断器会进入半熔断状态。所谓半熔断就是尝试恢复服务调用，允许有限的流量调用该服务，并监控调用成功率。如果成功率达到预期，则说明服务已恢复，进入熔断关闭状态；如果成功率仍旧很低，则重新进入熔断关闭状态。熔断状态下，对应用程序的请求会立即返回错误响应。

服务框架与web应用之间的关系：

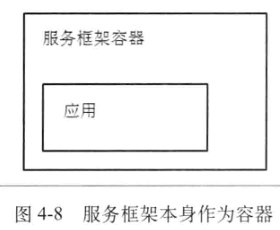
（1）在web应用中编写比如连接数据库的代码



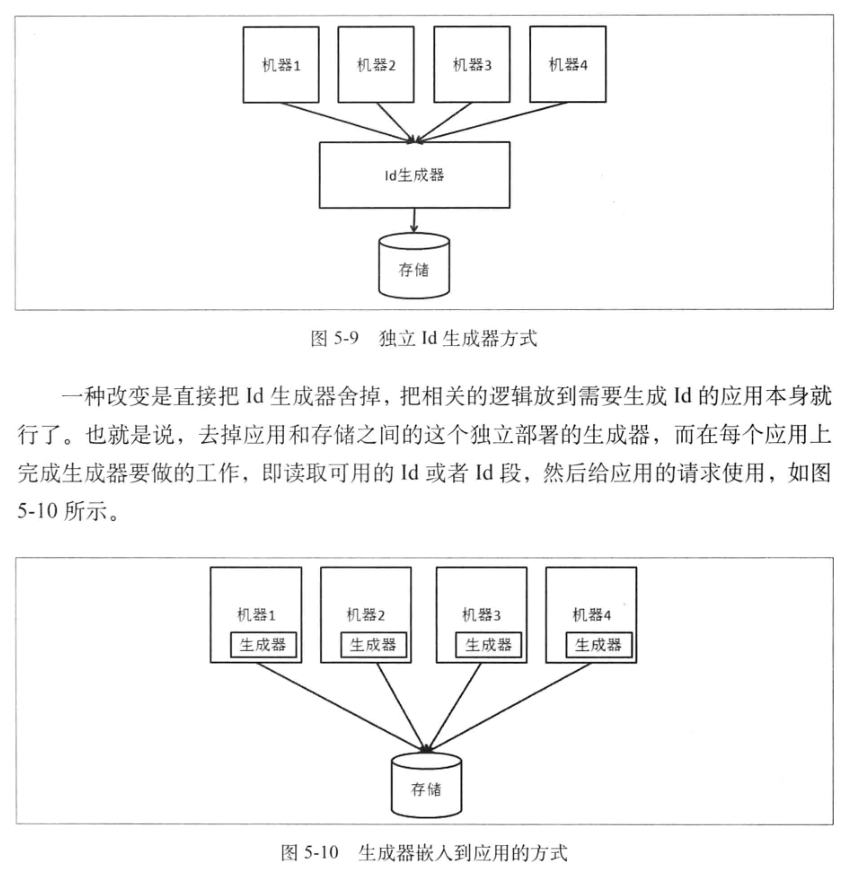
（2）单独作为一个web容器运行



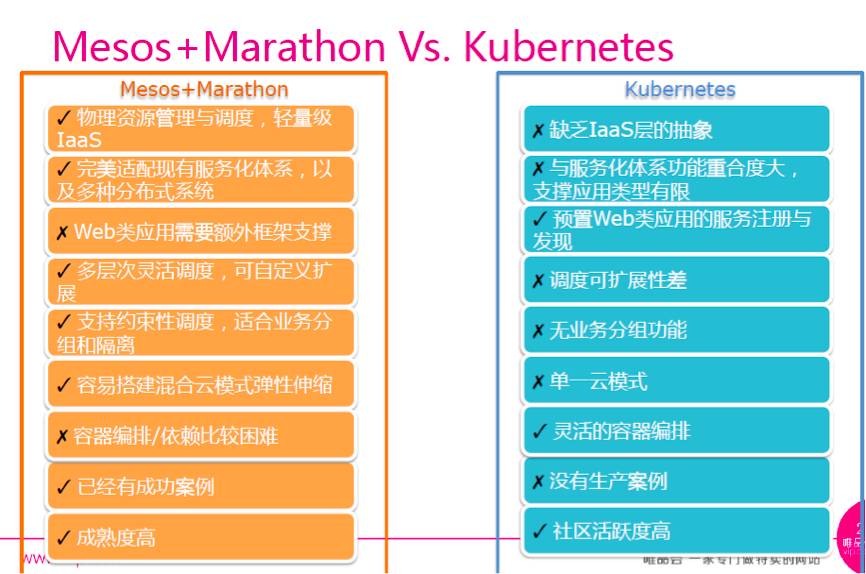
（3） 本身与web服务器结合在一起



分布式产生独立的 ID有两种办法：



下图是唯品会选择Docker容器管理工具的比较：



负载均衡：

HTTP重定向，整个流程需两次请求，速度受限

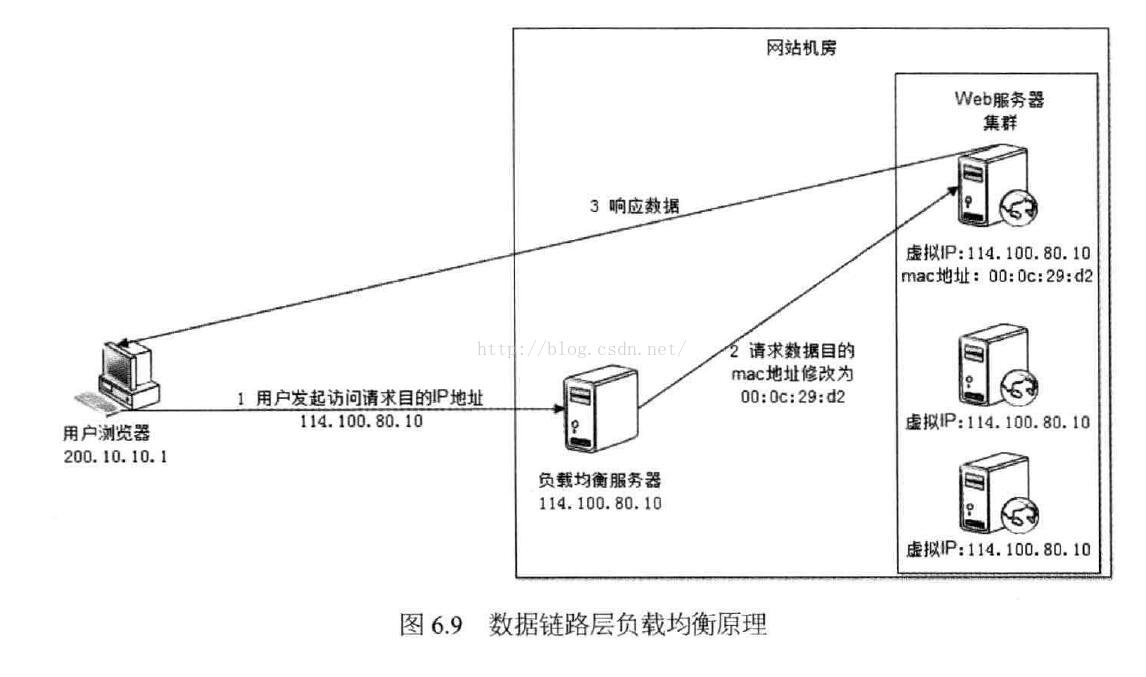
CDN客户端引流，类似于DNS域名解析负载均衡，是客户端层面的解决办法，负载均衡严格来讲主要是服务端。

DNS域名解析负载均衡：设计多条A记录

反向代理负载均衡：使用[代理服务器](https://baike.baidu.com/item/%E4%BB%A3%E7%90%86%E6%9C%8D%E5%8A%A1%E5%99%A8/97996" \t "_blank)可以将请求转发给内部的Web服务器。瓶颈一般出现在代理服务器上，要建立一个对外以及对内连接。好处是方便与代理服务器上的缓存技术结合。这种模式运行在http,所以也叫作应用层负载均衡

ip负载均衡：说明是ip层上实现。需要一个网关将数据包中的目的ip进行更改。

数据链路层负载均衡：链路层修改mac地址的负载均衡。相比较ip层负载均衡的好处在于，算是三角模式，响应的返回不需要经过修改：



CDN只能部署在运营商的机房，反向代理部署在自己的中心机房



多级缓存：

浏览器缓存、缓存服务器、tomcat缓存、分布式数据库（内存缓存）

本地缓存（Java EhCache）是为了减少和远程缓存（redis）的数据交互量，而且本地缓存属于进程内部缓存，访问更加快速

缓存预热：启动的时候就将热点数据加载好

分布式缓存可以分为两种：JBoss Cache代表的需要同步更新；Memcached代表的不互相通信（一致性hash）。

分布式处理session的问题：

忘了web笔记里面记录了没有

session复制，在各个服务器上都有每一个用户的session缓存

session绑定（又称会话黏滞），按照基于源地址的hash算法，可以是一致性hash算法提升服务器的集群变化能力。

使用Cookie记录Session

配置像数据库一样的session服务器集群，甚至直接存在redis数据库中

数据库读写分离带来性能提升的原因：

服务器的数量增加

主从只负责各自的写和读，极大程度的缓解X锁和S锁争用

方便专门针对读或写进行优化，从库可配置myisam引擎，提升查询性能以及节约系统开销

三种数据库分布式架构：

一主多从复制架构+读写分离：

从数据库只能读，写是多余的操作，这种架构是读写分离的产物

数据实时异步更新到从数据库

主数据库出现了问题，则要从从库中选举出一个主库，然后重新动态构建主从关系，但是不容易

多级复制架构：

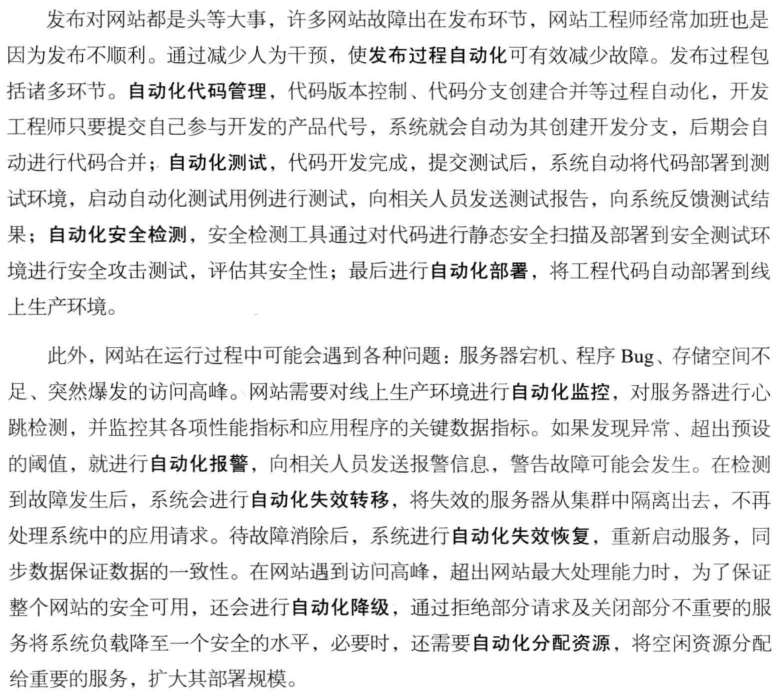
多级复制能够更加友好的除了主库宕机问题，A（主）-->B（从）-->C（从）=》B（主）-->C（从）=》A修复并数据更新之后，B（主）-->C（从）-->A（从）

缺点是增大了传送的深度

双主复制结构：

也是为了解决一主多从宕机时候的麻烦，两台数据库之间互为主从（其中一条默认作为从库，只有当主库宕机后，才作为主库出现），其余数据库作为从库

传统数据库多采用哦同步热备份的模式，同步指从库之间的备份关系，从库全部同时热备份，完成之后只会返回一个响应。异步热备份需要每一个从库返回一个响应。



web自动化测试工具：ThoughtWorks的Selenium，运行在浏览器中模拟用户的行为（py的某一个库也是这样）

服务降级： 高负载的时候保证核心业务

拒绝低优先级应用的调用（Twitter采用了随机拒绝请求的策略）

关闭不重要的服务（双十一的时候关闭删除订单、评价等应用服务）

幂等性设计：应用调用失败之后，会将请求重新发送到其他服务器，但是这个失败可能是虚假的失败。如果将用户性别设置为男性，多次操作结果依旧正确，即幂等性。

失效确认：

1. 多次心跳未接收
2. 应用程序访问错误报告+一次心跳检测确认

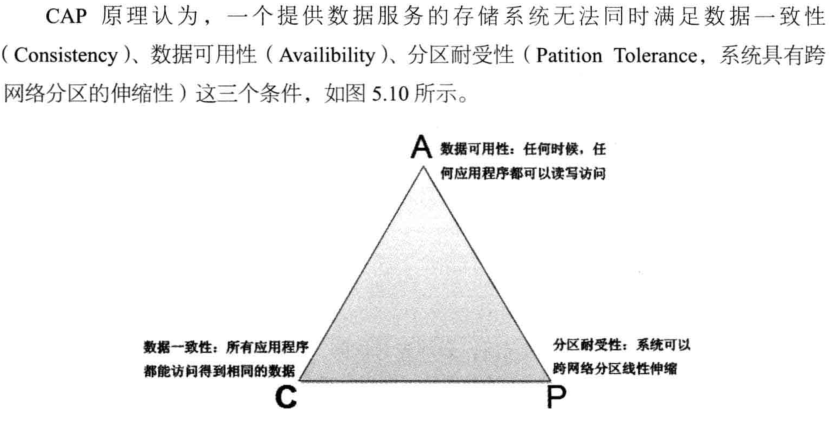
处理消息优先级、订阅者优先级：

1. 一种消息优先级使用一个队列，一种订阅者优先级使用一个集群
2. 一种消息优先级或者一种订阅者优先级使用一个消息中间件处理，订阅者优先级会导致分级订阅架构：

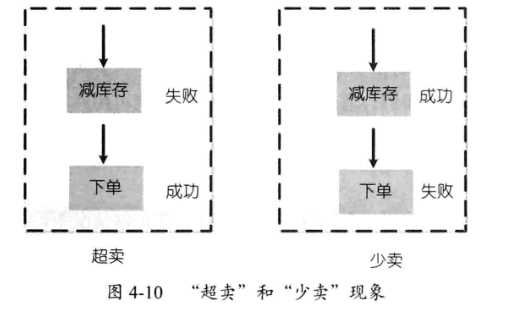
![C:\Users\tmp\AppData\Roaming\Tencent\Users\641370196\QQ\WinTemp\RichOle\J4](%TV]KI@STYQPYS)%{MW.png](data:image/png;base64,)

消息的局部顺序：比如付款要在下订单之后，可以让消息存储中付款一个队列，下订单一个队列，如果都放在一个队列中找起来有很多不必要的数据。

回溯保证消息的安全：消息消费之后还是被保留一段时间的，每一个客户端在队列中都有一个指针，随着消费移动，出现问题可以回溯到之前已经消费过的消息。

CAP原理：  


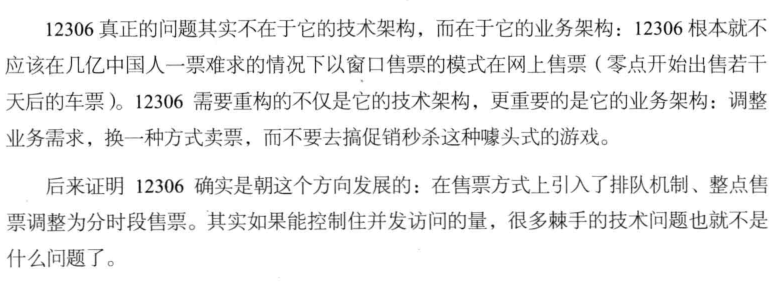
P是必须的，导致了数据一致性的“虚弱”。体现为电商平台交易过程中的“超卖”现象：



2011年刘强东紧急加购十台服务器，但是第二天网站还是不能解决负载问题，就是因为P能力不够。

案例分析：

（1）



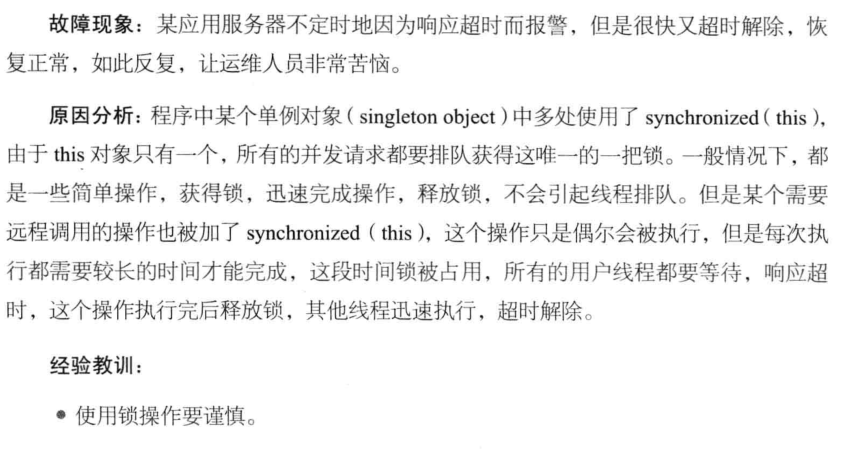
（2） 微博实现中几个要注意的地方

对粉丝的实时消息推送，明星用户存在大量粉丝这会导致一时间产生大量数据库读写操作，明星用户的消息应被缓存，之后再慢慢写入数据库

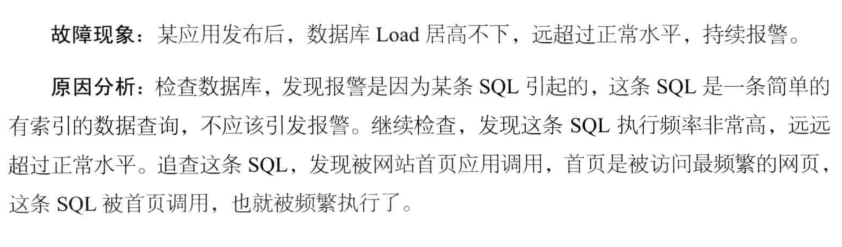
社交APP的常用用户是“固定”的，缓存的内容也就可以确定

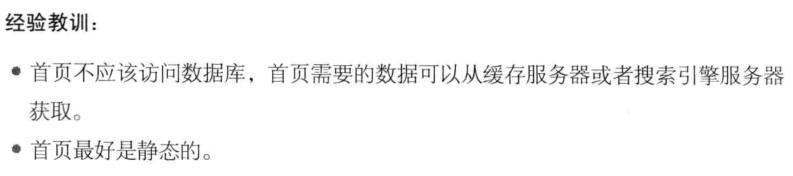
处理僵尸粉丝与垃圾粉丝

（3）

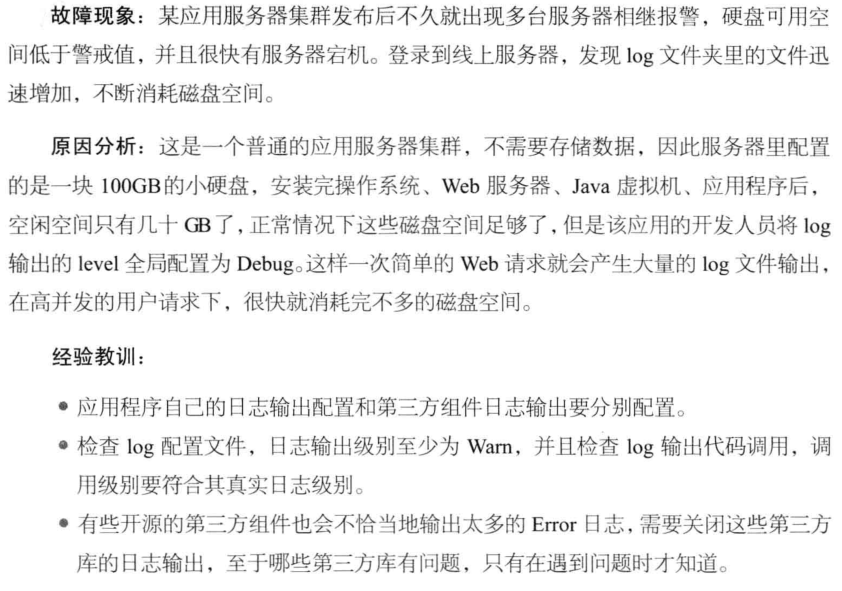


（4）

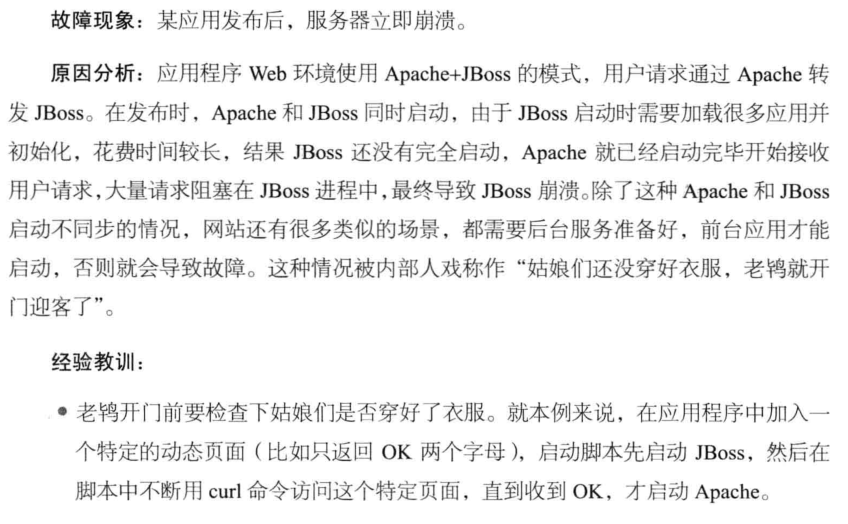




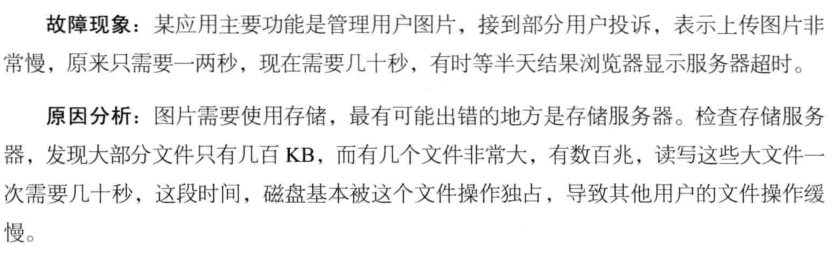
（5）

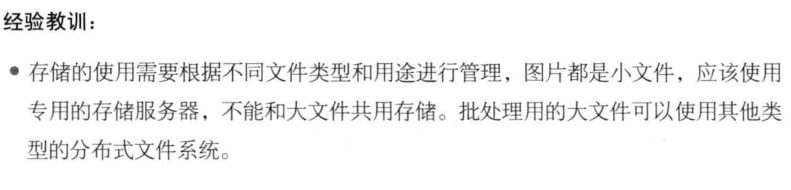


（6）



（7）



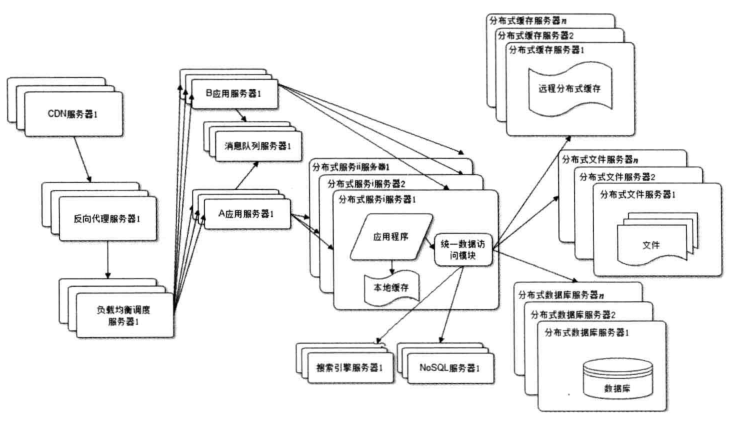


（8）我们服务发现与治理用的 Zookeeper，Zookeeper 瓶颈非常多，如何在跨机房、大数据量情况下如果用好 ZooKeeper？

首先整个系统设计，核心做选举的三个节点一定要放在同一个数据中心部署。不然写数据会造成整个 Zookeeper 集群不稳定。另外所有的业务节点全部挂在观察者模式上，让观察者模式不要影响全局。

（9）不是很认同下面这个，我认为应该Master收到消息就返回给用户“成功”![C:\Users\tmp\AppData\Roaming\Tencent\Users\641370196\QQ\WinTemp\RichOle\$UT9{5`EO3_](GL1T8DDR0Q.png](data:image/png;base64,)

架构分析：



CDN 反向代理（负载均衡+缓存网站中的静态资源）

nginx、tomcat集群分别处理动静态资源（减少服务器的压力，并且最好使用独立的域名加快浏览器的并发访问速度）

用户请求使用了服务器端负载均衡的调度，不过主要出现了多个服务器用于相同的功能就可以部署一个负载均衡调度

应用拆分（重要步骤设置较高优先级，高负载服务必须独立出去避免整个GG），通过消息队列联系（消息队列可以极大地减少消费者服务器的处理压力，将极高的峰值变成比较高的一段平滑曲线）

使用了本地缓存与远程分布式缓存结合

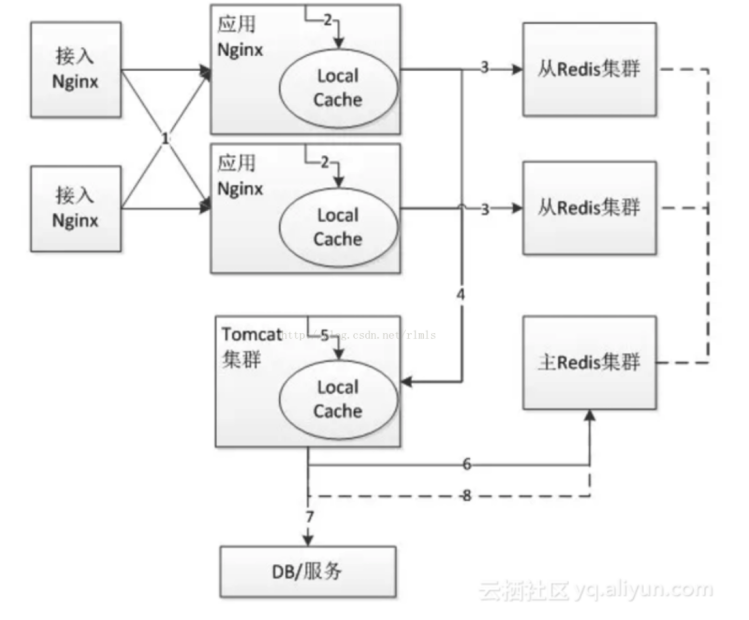
将程序与存储、具体数据分离

统一访问的模块，即中间件，实现变量池

分布式文件存储系统

数据库实现读写分离、分布式的分库分表

采用特殊的数据库完成特殊的功能（图数据库等nosql、非数据查询技术比如搜索引擎）



1、首先接入Nginx将请求负载均衡到应用Nginx，此处常用的负载均衡算法是轮询或者一致性哈希，轮询可以使服务器的请求更加均衡，而一致性哈希可以提升应用Nginx的缓存命中率；后续负载均衡和缓存算法部分我们再细聊；

2、接着应用Nginx读取本地缓存（本地缓存可以使用Lua Shared Dict、Nginx Proxy Cache（磁盘/内存）、Local Redis实现），如果本地缓存命中则直接返回，使用应用Nginx本地缓存可以提升整体的吞吐量，降低后端的压力，尤其应对热点问题非常有效；为什么要使用应用Nginx本地缓存我们将在热点数据与缓存失效部分细聊；

3、如果Nginx本地缓存没命中，则会读取相应的分布式缓存（如Redis缓存，另外可以考虑使用主从架构来提升性能和吞吐量），如果分布式缓存命中则直接返回相应数据（并回写到Nginx本地缓存）；

4、如果分布式缓存也没有命中，则会回源到Tomcat集群，在回源到Tomcat集群时也可以使用轮询和一致性哈希作为负载均衡算法；

5、在Tomcat应用中，首先读取本地堆缓存，如果有则直接返回（并会写到主Redis集群），为什么要加一层本地堆缓存将在缓存崩溃与快速修复部分细聊；

6、作为可选部分，如果步骤4没有命中可以再尝试一次读主Redis集群操作，目的是防止当从有问题时的流量冲击；

7、如果所有缓存都没有命中只能查询DB或相关服务获取相关数据并返回；

8、步骤7返回的数据异步写到主Redis集群，此处可能多个Tomcat实例同时写主Redis集群，可能造成数据错乱，如何解决该问题将在更新缓存与原子性部分细聊。

风控团队：将高风险的交易识别出来，进行人工审核

资源监视与使用分析——高性能探针

软件的线程数量取决于IO等待时间的比重，CPU内核数量，任务负载情况

数据库连接、网络通信连接、线程、复杂对象等都应该复用，复用的两种模式：（1）单例：目前web开发中主要使用贫血模式，Service、DAO、Servlet等都是无状态对象，无需重复创建。Spring对象都是单例对象。（2）对象池

规则引擎：通过UI操作能够设定不同的规则

ORM框架将对象与关系数据库解耦

秒杀页面的URL要动态生成，同时监控用户的访问行为

升级的时候通过版本号区分

**网络流量控制**：一种利用软件或硬件方式来实现对电脑网络流量的控制。

**灰度发布（金丝雀部署）**：将集群服务器划分成若干部分，每一段时间只在一部分服务器上发布，观察故障看是否回滚。游戏中相当于体验服。如果观察的是多个不同版本下用户的行为，收集体验报告确定最后的发布版本，即AB测试。

**Blue/Green Deployment（蓝绿部署）**：创建好蓝色版本之后，将解析从绿色转移到蓝色。在非隔离基础架构（ VM 、 Docker 等）上执行蓝绿部署，蓝色环境和绿色环境有被摧毁的风险。

**滚动发布**：一般是取出一个或者多个服务器停止服务，执行更新，并重新将其投入使用。周而复始，直到集群中所有的实例都更新成新版本。