<https://www.cnblogs.com/davidwang456/articles/5360292.html> 到中间“**在 Jenkins 里面把它部署到多个 Registry 里**”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

存储等偏向于底层的方面，要想能够性能做的不错，需要软硬件集合，类似于华为的Turbo技术。

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）的迅速查询与内容挖掘（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>

**混合云**融合了公有云和私有云，是近年来云计算的主要模式和发展方向。我们已经知道私有云主要是面向企业用户，出于安全考虑，企业更愿意将数据存放在私有云中，但是同时又希望可以获得公有云的计算资源

**IaaS:** Infrastructure as a Service，基础设施即服务，自己搭建或者购买运营商的服务器

**PaaS：**Platform as a Service，平台即服务，阿里云或者腾讯云这种平台。

**SaaS：**Software as a Service，软件即服务，不需要自己处理服务器、域名等问题，只需要购买模板、设计界面等等。

**S3**是Simple Storage Service的缩写，即简单存储服务。最常见的是Amazon S3。AWS

就是亚马逊旗下的全球市场占额最大的云服务提供商。

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

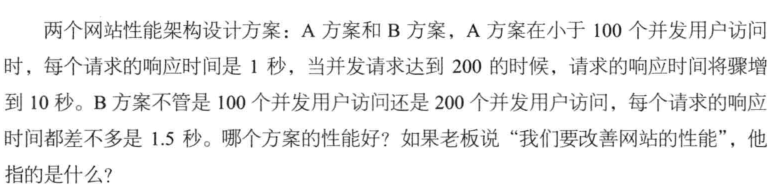
Dubbo 阿里的分布式服务框架

Cobar 阿里的分布式数据库中间件（支持数据的分库分表）

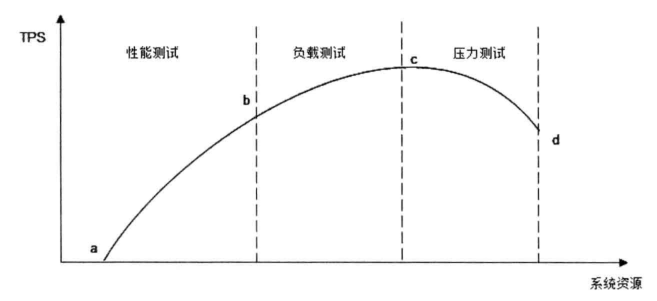
Chubby Google的分布式锁服务

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

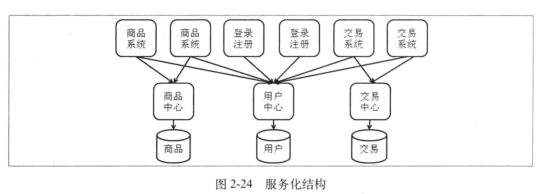
网站技术是为业务服务的，是否使用分布式服务需要根据业务发展规划改变。所以会出现渐进式的演变，使用了分布式之后又改回集中式。



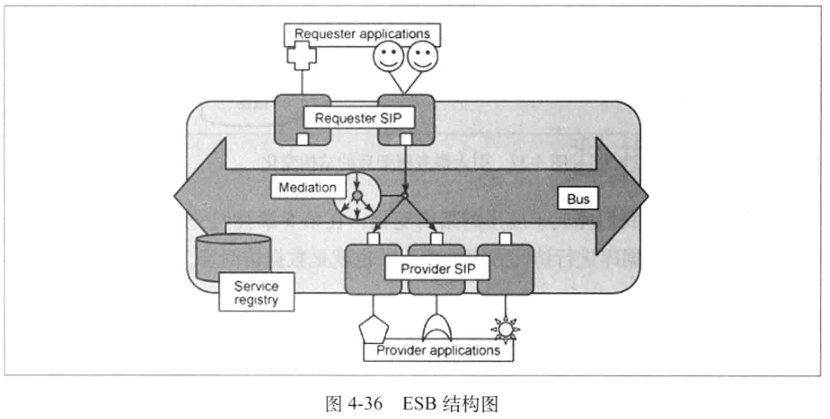
用户、开发人员、运维人员对网站的性能理解不同



随着压力的增长，性能测试阶段错误率就已经不为0了。



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

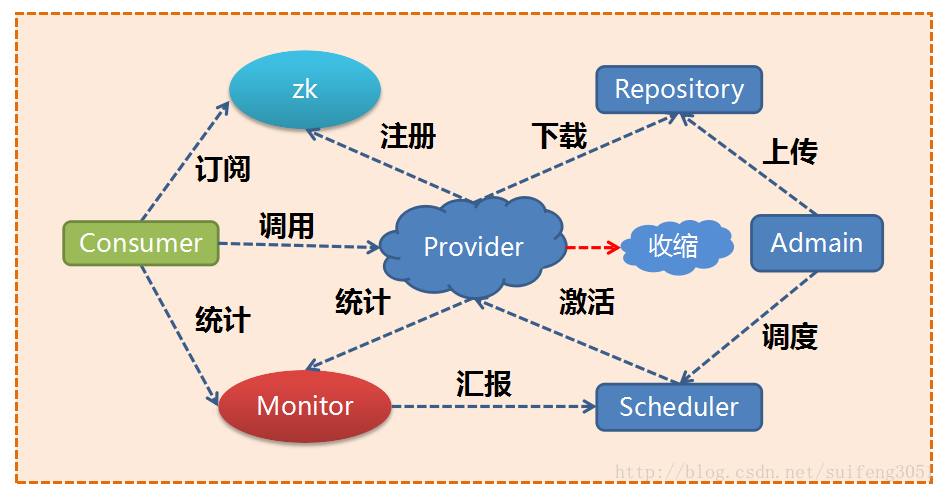


ESB（企业服务总线）与微服务的区别：（1）总线式与点对点；（2）服务框架面向同构系统，ESB会更多考虑不同产商提供的服务的整合。

**API网关**是一个服务器，是系统的唯一入口。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，除此之外，我们一般也会把安全，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。

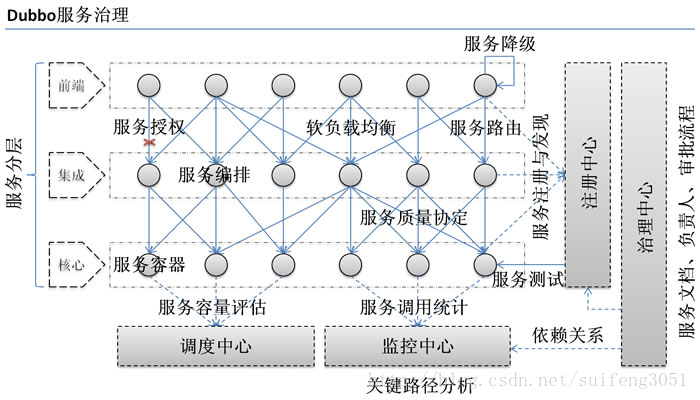
**基于接口的路由**：一个服务提供多个接口，但是有一个接口会比较耗时，所以要将耗时的与不耗时的分开路由。

**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。

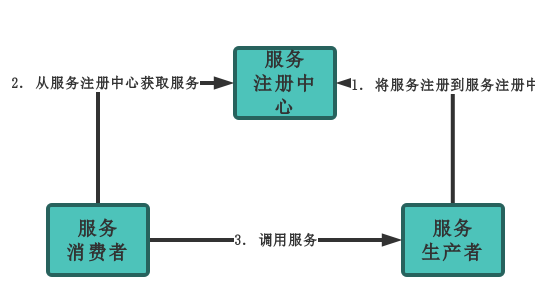


服务治理包含了所有服务相关的控制与相关信息的查看：

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：



关系调用说明：

* 服务生产者启动时，向服务注册中心注册自己提供的服务
* 服务消费者启动时，在服务注册中心订阅自己所需要的服务
* 注册中心返回服务提供者的地址信息给消费者
* 消费者从提供者中调用服务

服务限流：

目的可能是因为灰度发布、也可能是因为服务器的承受能力

CDN、负载均衡、使用消息队列都算是限流

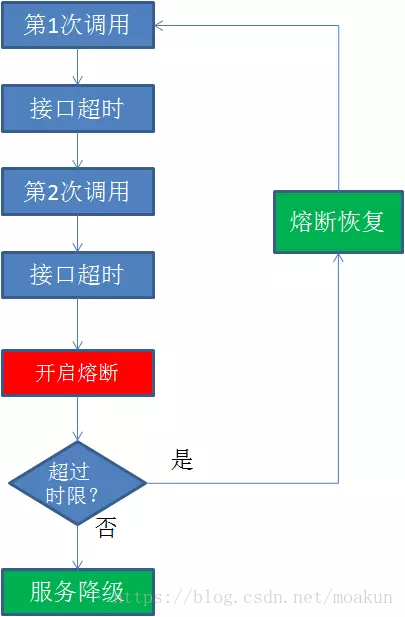
<https://www.cnblogs.com/clds/p/5850086.html> 以及其中的两个URL

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。

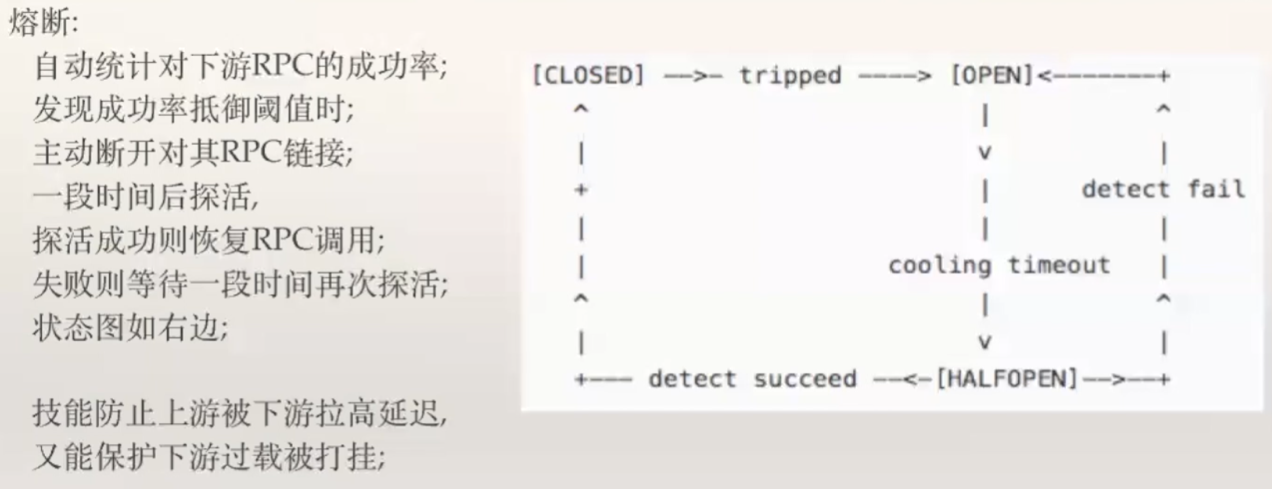
服务熔断（服务隔离）： 为了拯救服务雪崩问题



<https://blog.csdn.net/moakun/article/details/80222325>

<https://blog.csdn.net/whereismatrix/article/details/53465722>

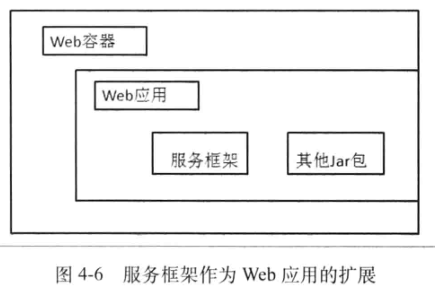
服务降级定义存在问题



我一直觉得数据来源是上游（用户是最上游也是产品服务链的最小游），但是公司里是调用方是上游（用户是最上游）。

服务框架与web应用之间的关系：

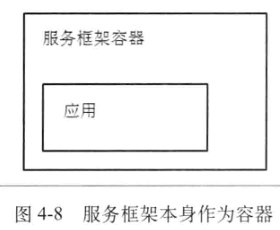
（1）在web应用中编写比如连接数据库的代码



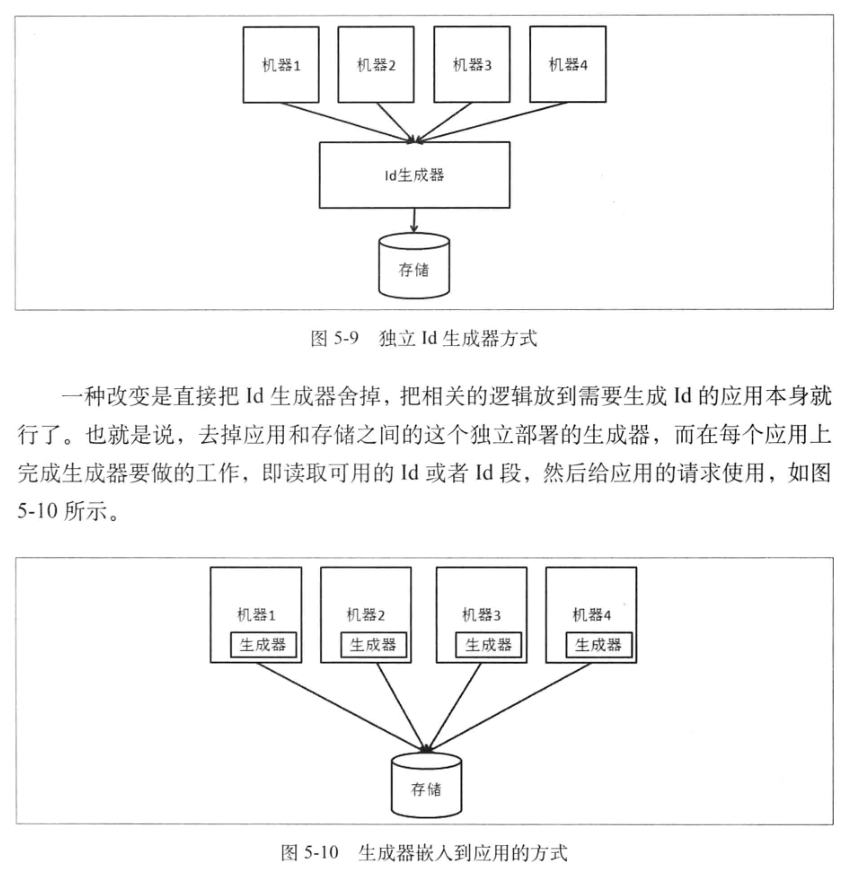
（2）单独作为一个web应用运行



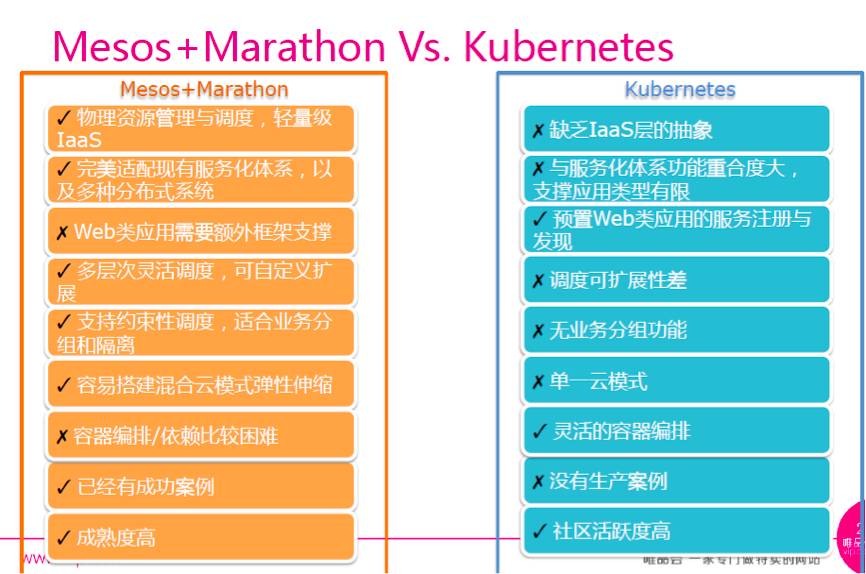
（3） 本身与web服务器结合在一起



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：



Mesos：相关网址： <https://www.cnblogs.com/xiaomaohai/p/6158061.html> Mesos Kernel利用资源管理和调度的API在整个数据中心或云环境中运行和提供引用（例如，Hadoop、Spark、Kafka、Elastic Search）。架构上将资源(CPU、mem、disk)的调度分成两层，Mesos只是监管资源与提供资源给框架（本身不清楚各个应用的资源需求），资源调度由对应的框架实现。分为Master与Slave两种节点，Slave负责对Master进行资源的报告，Master负责将资源的报告汇总向，并向注册到Master的Framework发出资源邀约。



YARN：存在类似于Mesos，但是工作上比Mesos多出对资源的调度，应用程序的ApplicationMaster会把各个任务的资源要求汇报给YARN，YARN则根据需要为应用程序分配资源。不支持disk的资源调度。

Marathon：Marathon是一个Mesos框架（Spark也是一种Mesos框架），能够支持运行长服务，比如Web应用等。它是集群的分布式Init.d，能够原样运行任何Linux二进制发布版本，如Tomcat、Play等等。它也是一种私有的PaaS，实现服务的发现，为部署提供提供REST API服务，有授权和SSL、配置约束，通过HAProxy实现服务发现和负载平衡。

Mesos+ZooKeeper+Marathon+Docker 可以搭建一个分布式PaaS平台。

负载均衡：

HTTP重定向，整个流程需两次请求，速度受限

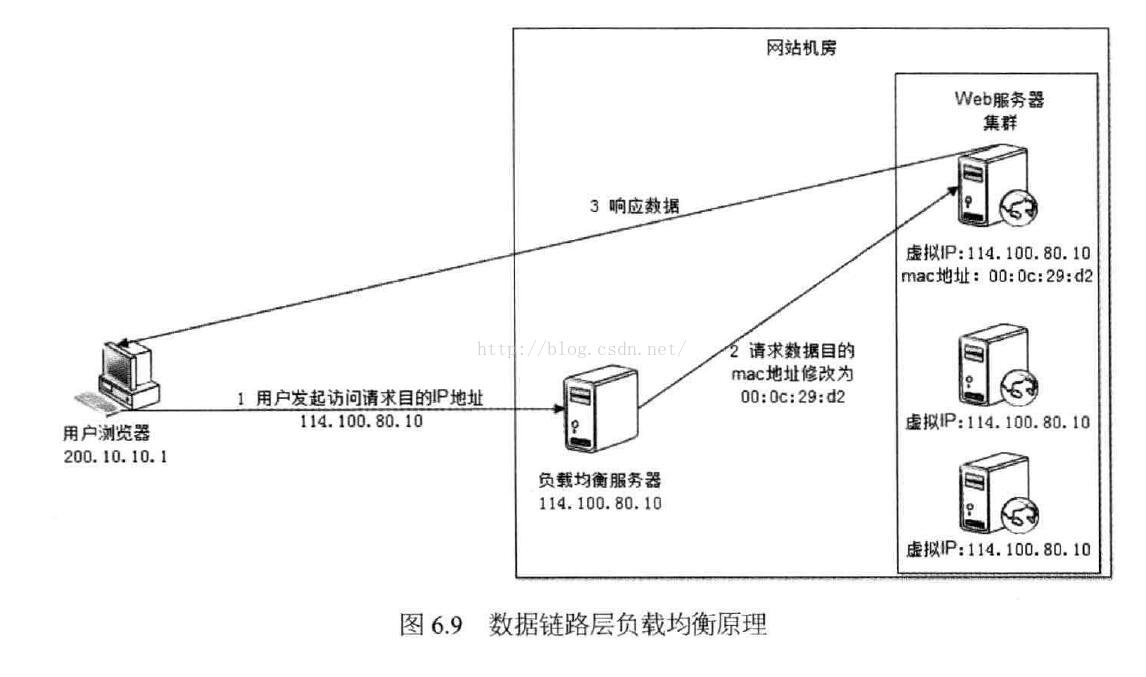
CDN客户端引流，类似于DNS域名解析负载均衡，是客户端层面的解决办法，负载均衡严格来讲主要是服务端。

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，ip层开启了虚拟共享模式，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

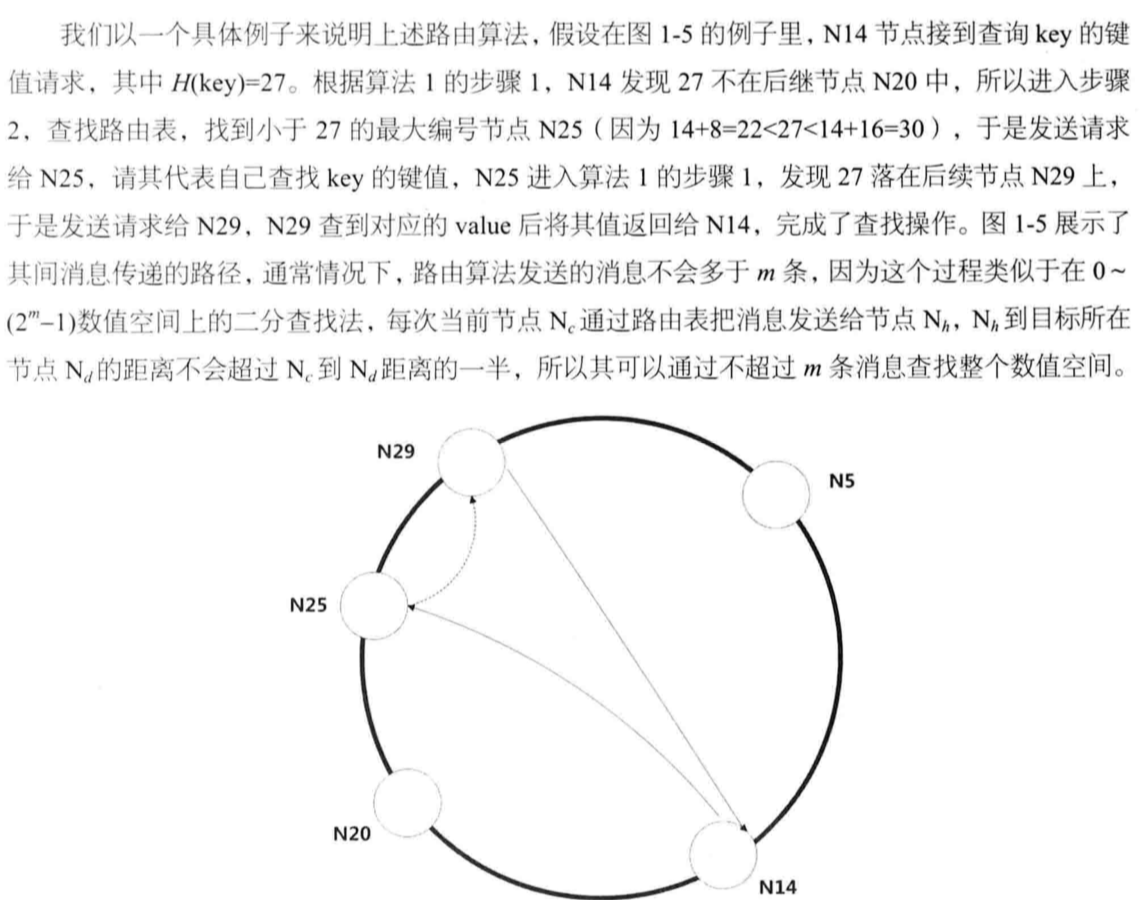
负载均衡的几种算法： 类似于操作系统中的调度算法



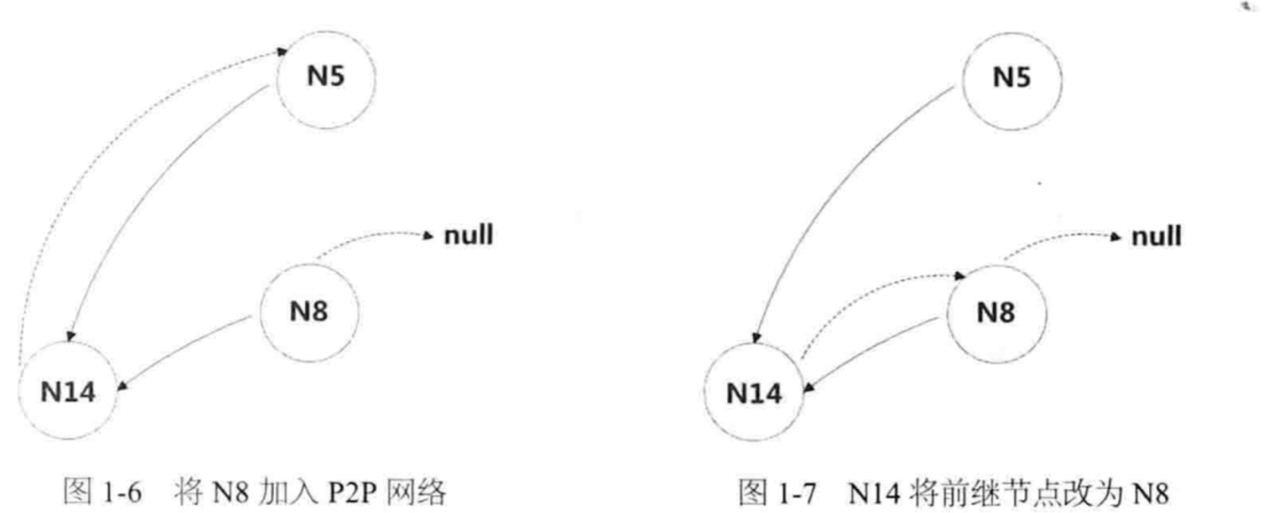
Nginx支持的5中分发策略：轮询、指定权重（权重与访问比率正比）、ip hash、按照服务器的响应时间调整、url hash <https://www.cnblogs.com/andashu/p/6377323.html>   
如果负载均衡有误的话，push给用户的时候就会有一个小尾巴，导致总体push时长会很长。Mongo如果是读取从库，那么负载均衡就要注意机房的从库比例，是否能够真的按比例分配任务到机房。

hash映射支持点查询；范围映射支持范围查询。非一致性hash必须考虑机器个数才能够将让负载均衡起来，一致性hash允许了新节点找不到时向旧节点的渗透。

一致性hash：（机器到环的映射是随机的，所以有“虚拟节点”概念）



以上版本，在并发条件中添加一个节点：（可能同时添加多个节点，在添加或者删除的时候做好各节点中变量改变的锁就好）



本来按照我的思路，数据的分发应该先经过一个总的代理路由器（入口流量这么走，出口流量直接从节点返回给目标），而不是直接落到节点上。添加一个节点，需要修改总路由器，并由总路由器“加锁”通知相关的两个节点，确认修改后结束“事务”。

每个节点都有一两个副本机，串联跟在节点后面，避免节点崩溃导致数据查询不到，及时记性写同步，副本机是虚拟机。

多级缓存：

浏览器缓存、缓存服务器（公司或者运营商）、分布式数据库（内存缓存）、tomcat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

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：

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加密记录一些Session中本来要存储的内容

配置像数据库一样的session服务器集群，甚至直接存在redis数据库缓存中

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

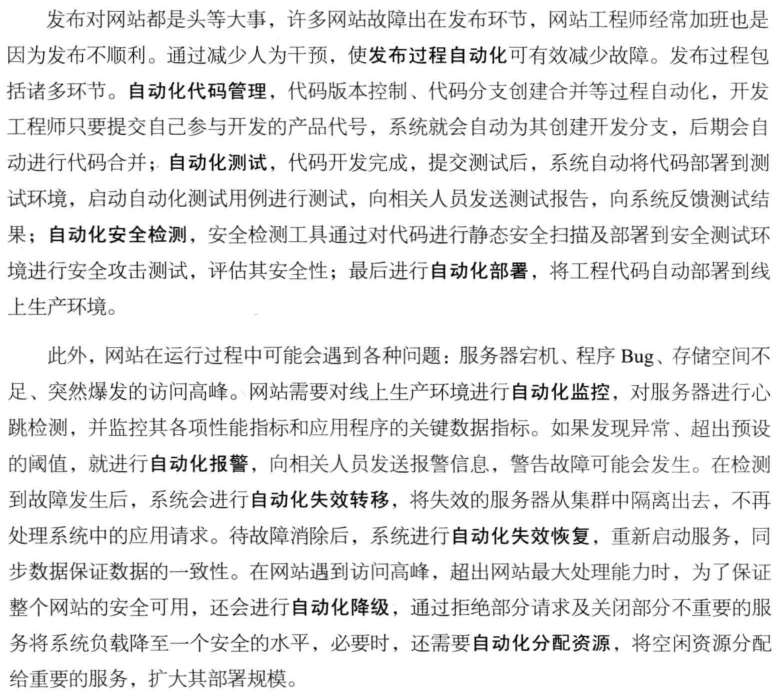
双主复制结构：

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。

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

ORM框架将对象与关系数据库解耦



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：

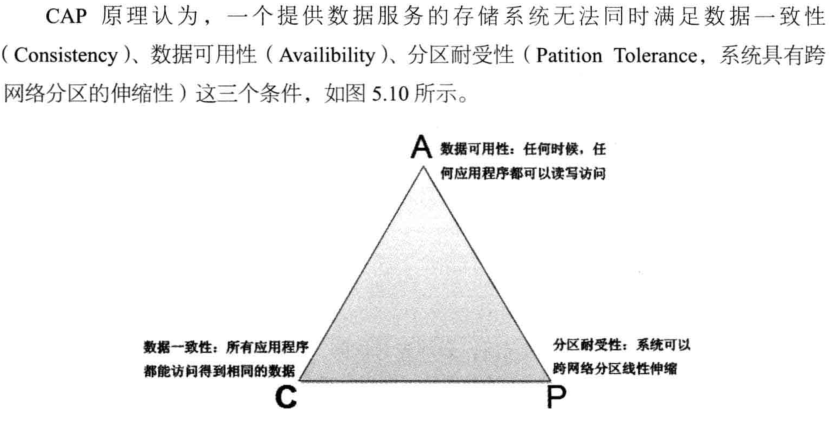
1. 一种消息优先级使用一个队列，一种订阅者优先级使用一个集群
2. 一种消息优先级或者一种订阅者优先级使用一个消息中间件处理，订阅者优先级会导致分级订阅架构：

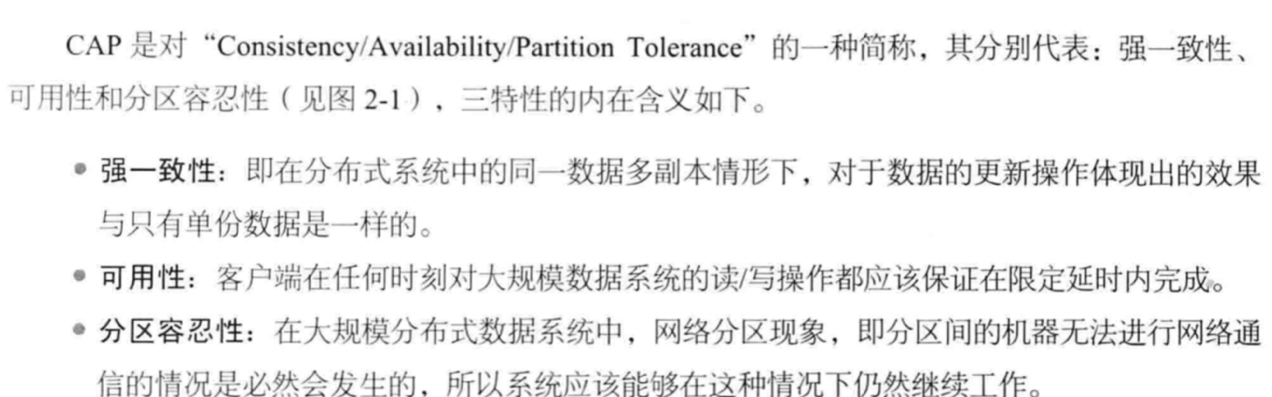
![C:\Users\tmp\AppData\Roaming\Tencent\Users\641370196\QQ\WinTemp\RichOle\J4](%TV]KI@STYQPYS)%{MW.png](data:image/png;base64,)

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。

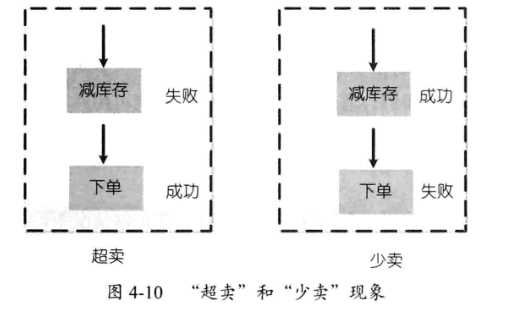
有的数据库数据会带上一个时间戳表示数据被创建或者被修改的时间，在宕机之后进行数据同步。

CAP原理：  


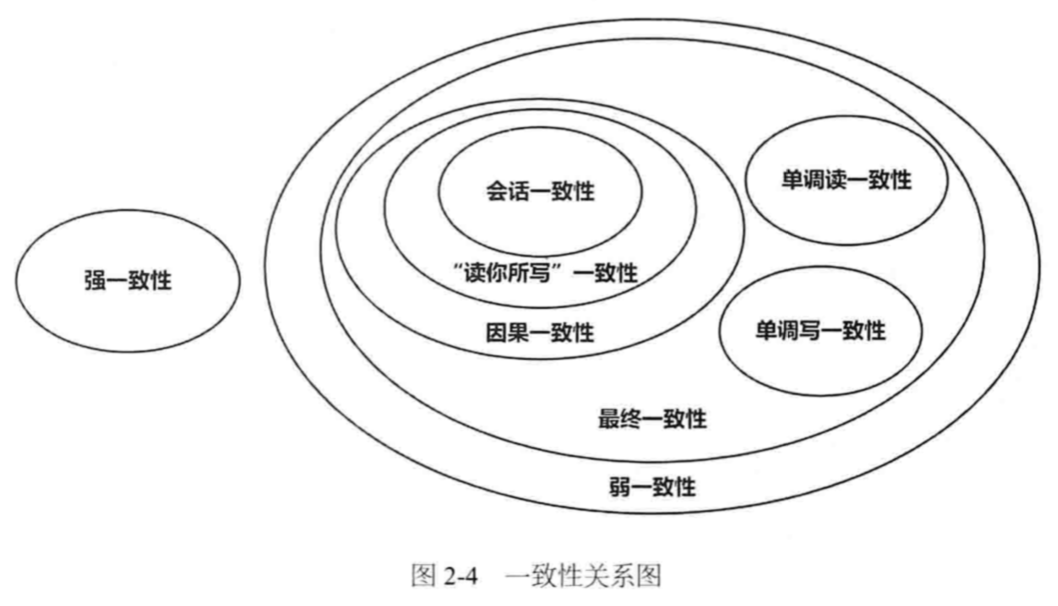


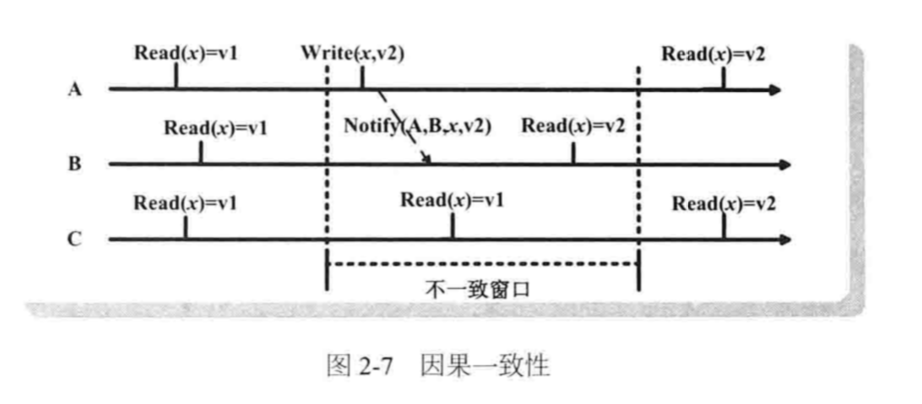
CAP原则很容易让人误解为“系统需要在CAP中选择两个字母而抛弃一个”，实际上因为P是必须的，但是基本不会遇到（根据系统规模），通常会兼顾三者，如果非要做取舍也不会是整个系统级别，而是子系统。

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：

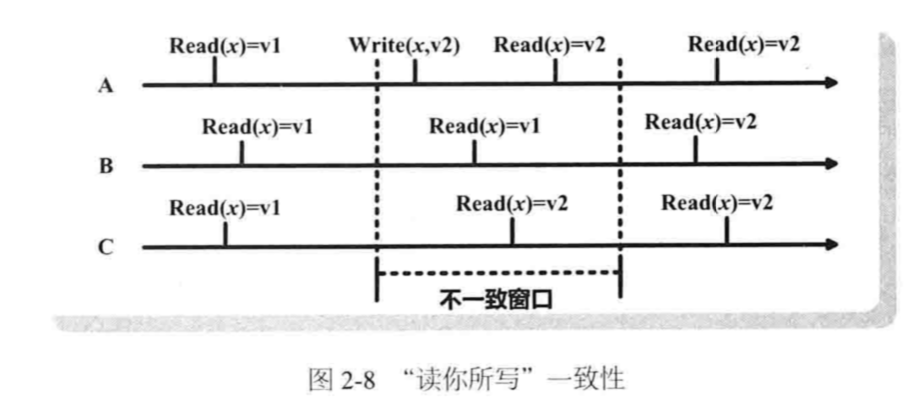
还有一种超卖现象因为数据一致性的延迟导致的。

很多种一致性：

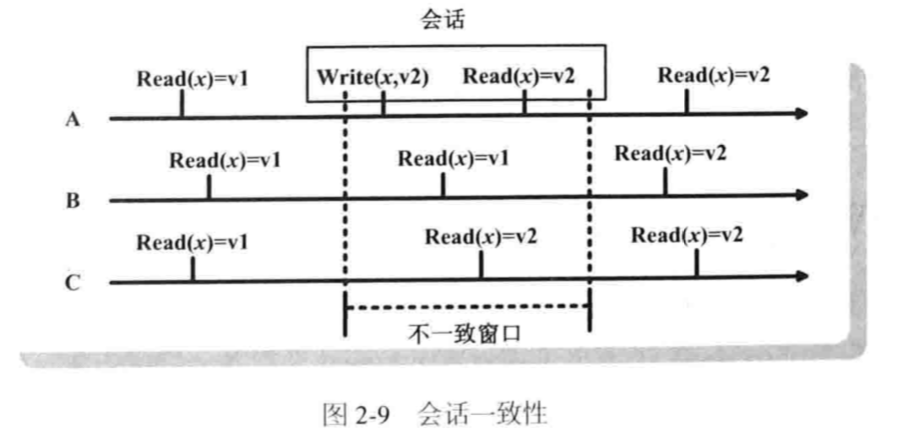




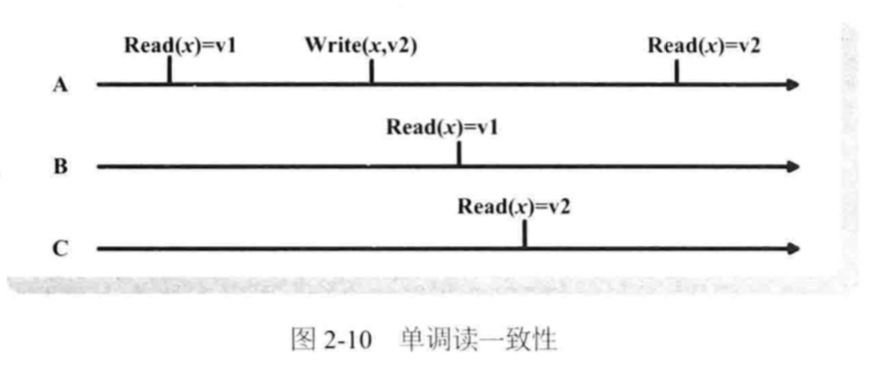
程序之间可以notify主动通知数据的改变



相当于对自己notify



会话内保证“读你所写”，如果会话异常中断，可能导致之后还是旧值



读到一次v2之后，要保证整个系统都是v2

如果数据对一致性要求不高，更新副本的时候不使用热门和协议，每隔一段时间对更新内容进行投票（可以直接全局，也可以先部分后全局）

kafka维护副本数据一致的时候，采用**同步异步混合更新**的方式：先立即同步notify保证更新一部分数据，剩余副本异步自己获取更新。混合同步异步的方式，在需要读取到最新的数据的强一致性要求环境时，也能让请求不集中于主机上。

**总主机与区域主机：**主机notify通知一部分从机更新，这部分从机之后再负责notify各自边上的

**任意节点更新：**主机通知任意一个节点更新后，其余副本都由这个节点通知更新。需要担心两个客户端同时发出对同一记录两个不同的更新要求。当主机故障时，系统可以选择这样（我想的是就近节点更新），冲突通过投票解决。

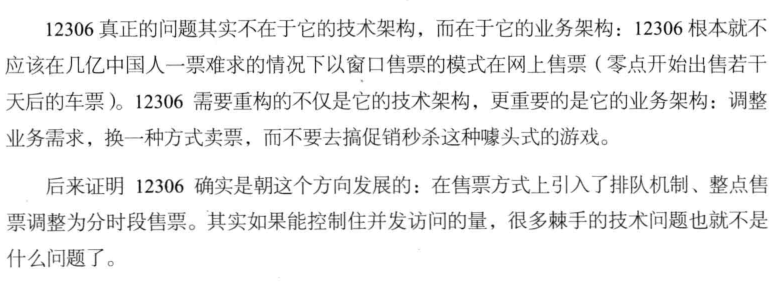
先部分后全局的投票就像是选总统一样

道路每隔一定长度就会有一个红绿灯，对应程序中不断进行数据缓冲与批量写入

案例分析：

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。

（1）



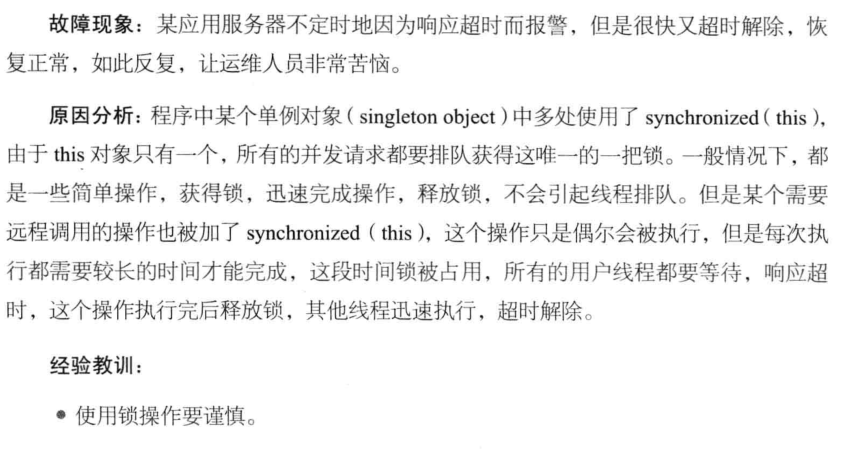
（2） 微博实现中几个要注意的地方

社交APP的常用用户是“固定”的，缓存的内容也就可以确定

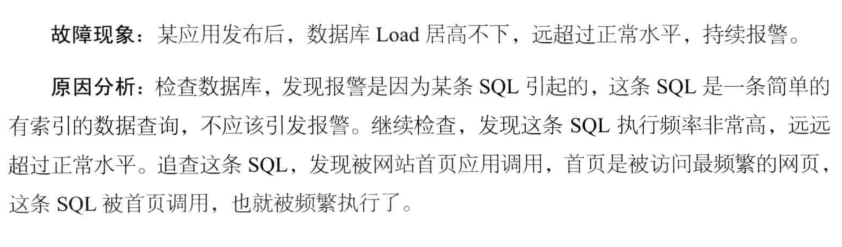
对粉丝的实时消息推送，明星用户存在大量粉丝这会导致一时间产生大量数据库读写操作，明星用户的消息应被缓存，之后再慢慢写入数据库

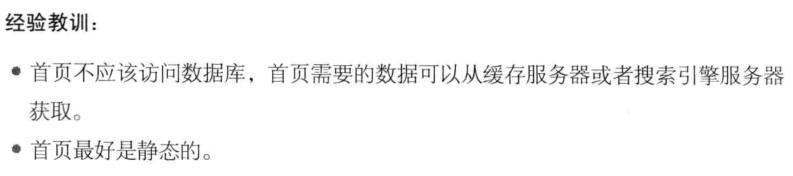
处理僵尸粉丝与垃圾粉丝

（3）

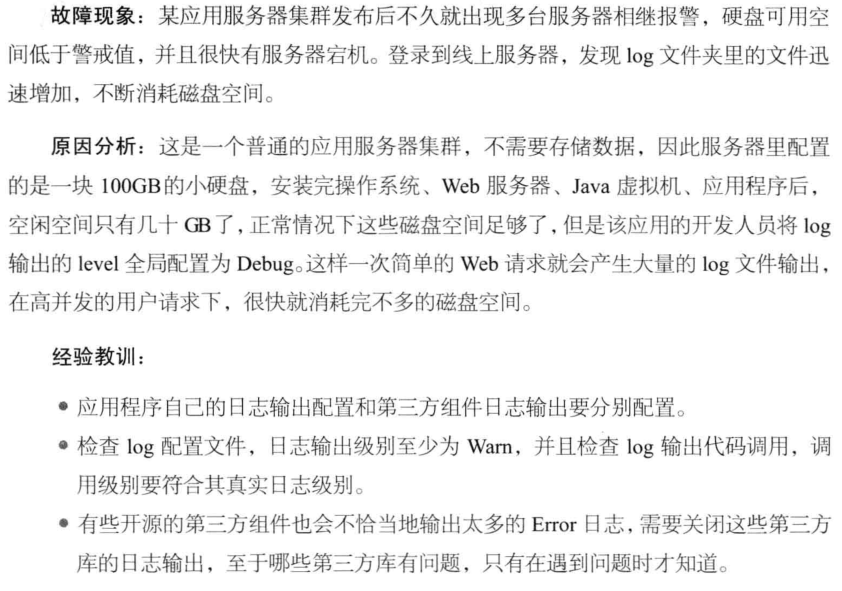


（4）

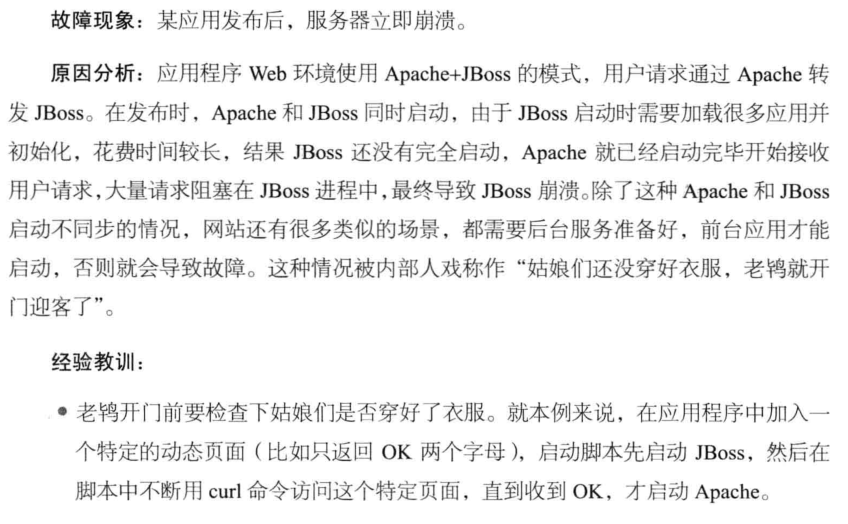




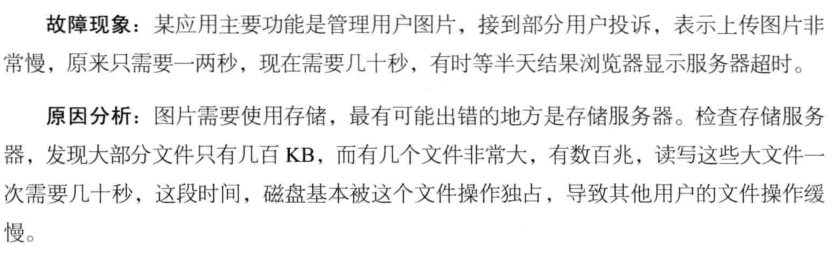
（5）

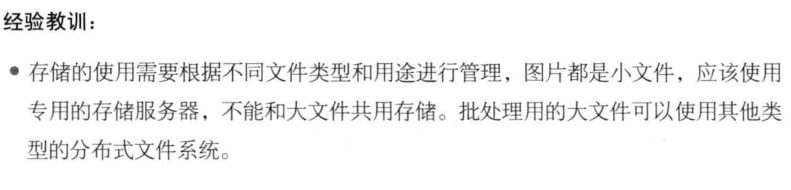


（6）



（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,)

（10） 秒杀页面的URL要动态生成，同时监控用户的访问行为

（11） 出现大量CLOST\_WAIT

我想到的解决办法是设置Linux，使得能够及时释放资源

但是出现大量，说明与预期不符合，这应该要检查程序或者设计一个网络连接池。

1. 如果是周期性卡顿

比较有可能是jvm中周期性Full GC问题。通过调整新生代与老年代的比例（该值可以通过参数 –XX:NewRatio 来指定），调低老年代占的内存大小，增大新生代内存，这样老年代很快就满了，新生代不容易满，直到调整到发生fullGC时候对于系统性能影响不大的时候（用户察觉不出来），那么调优结束。

1. 在服务启动的时候会占用大量CPU资源

经过分析，确定是C2 CompilerThread线程。 Java程序在启动的时候所有代码的执行都处于解释执行模式，只有在运行了一段时间后，根据代码方法执行的次数，或代码里循环的执行次数等达到一定的阈值才会编译成机器码。默认编译线程个数为2 。

有以下三种处理方式：

1）预热 真正的处理，但是做起来不简单

      如果在服务接受线上请求之前提前完成编译优化过程，那么将能避免此种抖动情况。一般的做法是预热，有两种方法：

      a）程序主动预热：在启动完成后，程序主动的访问热点的代码，确保主要的热点代码已被编译成机器码后再放入流量，可通过-XX:+PrintCompilation来确认。

      b）复制流量预热：通过tcpcopy软件拷贝一份线上nginx的流量进行预热，完成之后再导入线上流量。

2）启动多个线程进行编译优化

     如果能加快编译优化速度，那也能降低解释执行阶段导致的抖动时间。因此可以多拿几个线程来做编译，加快达到高峰性能的速度。

     可以使用-XX:CICompilerCount参数来设置编译线程数目，这个值默认是2（之前在栈里看到有两个编译线程），我们可以加到4。

3）采用多层编译

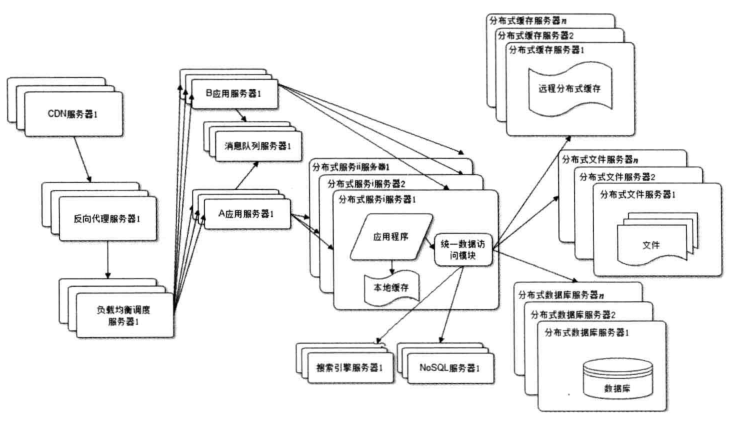
      编译方式有三种：1）Client模式；2）Server模式；3）Tiered模式。我们服务默认是Server模式。

      Server模式是采用c2高级编译的，会比较耗时且要运行一段时间才会触发编译。 Server模式的优点是编译后程序效率较高；

      Client模式比较轻量也比较快触发（比Server模式触发快），编译优化后程序效率不如Server模式；

      Tiered模式是Client模式和Server模式的折中，一开始会启用Client模式，可以在启动后更快的让部分代码先进入编译优化阶段，之后会启动Server模式，达到程序效率最大优化的目的。

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，不过只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）（甚至按照接口划分，但是优先级应该按照服务设置），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

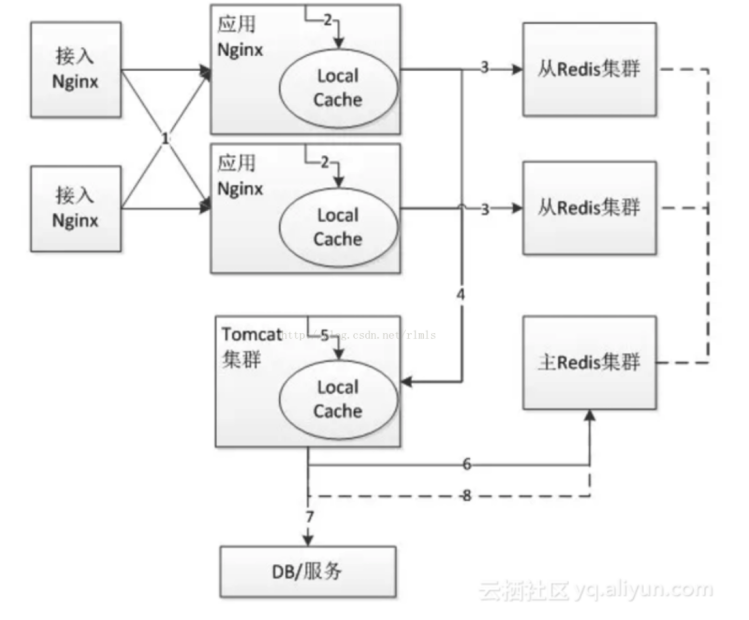
将程序与存储、具体数据分离

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）

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

升级的时候通过版本号区

**规则引擎：**通过UI操作能够设定不同的规则，联想到腾讯云的防火墙的规则设置

**风控团队：**将高风险的交易识别出来，进行人工审核

**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）**：将一小部分服务器更新作为金丝雀，观察故障看是否回滚，如果成功就全范围更新。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。

监控服务器的程序称为探针