# 访问一个网站的全过程

# DNS

* 先尝试从host文件中读取域名对应的IP地址，如果找到，则完毕；如果未找到，则使用DNS进行查找。

# TCP

* 三次握手建立连接

# 负载均衡服务器

* Nginx?

# 应用服务器

* Tomcat?

# 浏览器渲染

* 缓存？

# 大型网站架构演进

# 1)单机

# 2)单机负载告警，数据库与应用分离

# 3)应用服务器负载告警，让应用服务器走向集群

# 1)引入负载均衡设备

# 2)分布式Session

## Session Sticky 会话粘滞

## Session Replication 会话复制

## Session 集中存储

## Cookie Based

# 4)数据库读压力变大，读写分离

# 1)数据库主从复制

# 2)搜索引擎

# 3)分布式缓存

* 数据缓存：
* 页面缓存：Apache或Nginx

# 5)弥补关系数据库的不足，引入分布式存储系统

# 6)数据库又遇瓶颈，分库分表

# 1)垂直拆分——分库

* 按业务分库
* 需要解决表关联和分布式事务问题

# 2)水平拆分——分表

# 7)应用拆分与服务化

# 1)应用拆分

# 2)服务化

# 8)消息中间件（异步+解耦)

# 9)CDN

* 内容分发网络。作用是把用户需要的内容分发到离用户近的地方，这样可以使用户能够就近获取所需内容。整个CDN系统分为CDN源站和CDN节点，CDN源站提供CDN节点使用的数据源头，而CDN节点则部署在距离最终用户比较近的地方，加速用户对站点的访问。
* CDN其实就是一种网络缓存技术，能够把一些相对稳定的资源放到距离最终用户较近的机房，一方面可以节省整个广域网的带宽开销，另一方面可以提升用户的访问速度，改进用户体验。我们一般把一些相对静态的文件放在CDN中。

# 引入CDN后浏览器访问网站的流程

- 1)用户向浏览器提交域名  
- 2)浏览器对域名进行解析，由于CDN对域名解析过程进行了调整，所以得到的是该域名对应的CNAME记录。  
- 3)对CNAME再次进行解析，得到实际IP地址。在这次的解析中，会使用全局负载均衡DNS解析，也就是我们需要返回具体IP地址，需要根据地理位置信息以及所在的ISP来确定返回的结果，这个过程才能让身处不同地域、连接不同接入商的用户得到最适合自己访问的CDN地址，才能做到就近访问，从而提升速度。  
- 4)得到实际的IP地址后，向服务器发出访问请求。  
- 5)CDN会根据请求的内容是否在本地缓存进行不同处理：

* 如果存在，则直接返回结果
* 如果不存在，则CDN请求源站，获取内容，然后再返回结果。
* CDN的几个关键技术：

# 全局调度

* 全局调度是完成用户就近访问的第一步，我们需要根据用户地域、接入运营商以及CDN机房的负载情况去调度。前面两个调度因素需要一个尽可能精准的IP地址库，这是正确调用的前提。IP地址库的维护是一个持续和变化的过程，并且调度的策略随着CDN机房的增加也会变化。除了距离，CDN的负载也是调度中的一个影响因素。

# 缓存技术

* 如果CDN机房的请求命中率不高的话，那么起到的加速效果也是相对有限的。
* 要提升命中率，就需要CDN机房中有尽可能全面的数据，这就要求CDN机房的缓存容量要足够大，可以使用内存+SSD+机械硬盘的混合存储方式来提升整体的缓存容量，并且需要做好冷热数据的交换，在提升命中率时也尽量降低缓存的响应数据。
* 此外，当CDN的cache没有命中要回源加载数据时，合并同样数据的请求也是一个很重要的优化，这样可以减少重复的请求，降低源站的压力。
* 最后，新增、变更数据后的CDN预加载也是一个提升命中率的办法，也就是在没有请求进来时，CDN主动去加载数据，做好准备。当然这个主动加载一般也需要源站有一个通知过来。

# 内容分发

* 内容分发主要是对内容全部在CDN上不用回源的数据的管理和分发，例如一些静态页面等。具体做法是在内容管理系统中进行编辑修改后，通过分发系统分发到各个CDN的节点上。分发的效率以及对分发文件一致性、正确性的校验是需要关注的点。

# 带宽优化

* CDN提供了内容加速，很多请求和浏览都压到了CDN上，那么如何能够比较有效地节省贷款是一个很重要的事情，因为这直接关系到流量成本。优化的思路是只返回必要的数据，用更好的压缩算法等。
* 我们可以利用CDN机房距离最终用户近的特点，让一些上传的工作从CDN接入，然后再从CDN传到源站，这一方面可以提升用户的上传速度，另一方面也很好地利用了从CDN机房到源站的上行带宽。

# 整体结构

# 大型网站架构模式

# 分层

* 大型网站架构中采用分层解雇，将网站软件系统分为应用层、服务层、数据层等。

# 分隔

* 如果说分层是将软件在横向方面进行切分，那么分隔就是在纵向方面对软件系统切分。
* 网站越大，功能越复杂，服务和数据处理的种类越多，将这些不同的功能和服务分割开来，包装成高内聚低耦合的模块单元，一方面有助于软件的开发和维护；另一方面，便于不同模块的分布式部署，提高网站的并发处理能力和功能扩展能力。

# 分布式

* 对于大型网站，分层和分隔的一个主要目的是为了切分后的模块便于分布式部署，即将不同模块部署在不同的服务器上，，通过远程调用协同工作。分布式意味着可以使用更多的计算机完成同样的工作。
* 常见的分布式有分布式应用，分布式静态资源（分布式文件系统)，分布式数据和存储（分布式数据库和NoSQL)，分布式计算

# 集群

* 对于用户访问集中模块，还需要将独立部署的服务器集群化，即多台服务器部署相同应用构成一个集群。通过负载均衡设备共同对外提供服务。
* 因为服务器集群有更多服务器提供相同服务，因此可以提供更好的并发特性，当有更多用户访问时，只需要向集群中加入新的机器即可。并且如果某台服务器发生故障时，负载均衡设备会将请求转移到集群中的其他服务器中，使服务器故障不影响用户使用。

# 缓存

* 比如CDN，反向代理（缓存静态资源)，本地缓存，分布式缓存

# 异步

* 内存队列；分布式消息队列
  + 1)提供系统可用性
  + 2)加快网站响应速度
  + 3)削峰

# 冗余

* 访问和负载很小的服务也必须部署至少两台服务器构成一个集群。数据库除了冷备份外，还需要进行主从分离，实现热备份。

# 自动化

* 自动化主要是在发布运维方面。比如自动化发布、自动化代码管理、自动化测试。

# 安全

# 大型网站核心架构要素

# 性能

# 网站性能指标（PV QPS TPS 吞吐量 响应时间)

* 网站并发量是多少？

## 响应时间

* 指应用执行一个操作需要的时间。响应时间是系统最重要的性能指标，直观地反应了系统的快慢。
* 测试程序通过模拟应用程序，记录收到响应和发出请求之间的时间差来计算系统响应时间。

## 并发数

* 指系统能够同时处理请求的数目。对于网站而言，对于网站而言，并发数即网站并发用户数，指同时提交请求的用户数目。
* 网站系统用户数（注册用户数)>>网站在线用户数（当前登录网站的用户总数)>>网站并发用户数
* 测试程序通过多线程模拟并发用户的办法来测试系统的并发处理能力。为了真实模拟用户行为，测试程序并不是启动多线程然后不停地发送请求，而是在两次请求之间加入一个随机等待时间，这个时间被称作思考时间。

## 吞吐量（还有QPS TPS)

* 指单位时间内系统处理的请求数量，体现系统的整体处理能力。对于网站，可以用请求数/秒，或者页面数/每秒来衡量，也可以用访问人数/天，或者处理的业务数/小时等来衡量。TPS（每秒事务数)是吞吐量的一个常用量化指标，此外还有QPS（每秒查询数)、HPS（每秒HTTP请求数)等。

## 性能计数器

* 它是描述服务器或操作系统性能的一些数据指标，包括System Load、对象与线程数、内存使用、CPU使用、磁盘与网络IO等指标。这些指标也是系统监控的重要参数，对这些指标设置报警阈值，当监控系统发现性能计数器超过阈值时，就向运维和开发人员报警，及时发现处理系统异常。
* System Load：系统负载，指当前正在被CPU执行和等待被CPU执行的进程数目总和，是反应系统忙闲程度的重要指标。多核CPU的情况下，完美情况是指所有CPU都在使用，没有进程在等待处理，所以Load的理想值是CPU的数目。当Load值低于CPU数目的时候，表示CPU有空闲，资源存在浪费；高于时表示进程在排队等待CPU调度，表示系统资源不足。
* 在Linux系统中使用top命令查看，该值是三个浮点数，表示最近1分钟、5分钟、15分钟的运行队列平均进程数。

# Web性能优化

## 前端优化

- 1)浏览器缓存  
- 2)HTTP请求合并  
- 3)CSS、JS、图片等资源合并压缩  
- 4)CDN  
- 5)页面静态化  
- 6)减少Cookie传输

## 服务端优化

- 1)接入层：反向代理，负载均衡   
- 2)应用服务器集群  
- 3)缓存：分布式缓存  
- 4)数据库优化，查询优化，索引，主从复制，读写分离，分库分表  
- 5)分布式消息队列，异步操作

# 影响单机并发量的因素与优化

- CPU消耗严重的解决办法：1)减少阻塞，同步转异步，队列等 2)减少线程数，避免大量时间消耗着线程上下文切换上3)降低锁的竞争，尽可能采用lock-free、non-blocking的方式  
- 文件IO消耗严重的解决办法：1)异步写文件2)批量读写3)限流，比如放入队列4)限制文件大小 5)缓存6)RAID  
- 内存消耗严重的解决办法：1)使用不必要的引用2)使用对象缓存池3)使用合理的缓存失效算法  
- 网络IO消耗严重的解决办法：1)减少网络交互次数2)减少网络传输数据量的大小3)尽量采用异步非阻塞的方式

* 本地缓存；数据库优化，索引等

# 抢购/超卖问题

## 前端

* 页面静态化
* CDN
* 禁止用户重复提交请求

## 接入层

* 限制同一个IP的访问频率
* 页面缓存

## 服务层

* 避免恶意刷单，直接绕过App/网页：每个表单分配Token，
* 可以考虑随机将部分请求直接返回失败
* 数据库悲观锁更新库存

## 数据层

* 主从复制，读写分离，分库分表

# 抢红包问题

* 一万个人抢100个红包，如何实现（不用队列)，如何保证2个人不能抢到同一个红包，可用分布式锁
* 微信从财付通拉取金额数据过来，生成个数/红包类型/金额放到redis集群里，app端将红包ID的请求放入请求队列中，如果发现超过红包的个数，直接返回。根据红包的逻辑处理成功得到令牌请求，则由财付通进行一致性调用，通过像比特币一样，两边保存交易记录，交易后交给第三方服务审计，如果交易过程中出现不一致就强制回归。
* 红包如何计算被抢完？ cache会抵抗无效请求，将无效的请求过滤掉，实际进入到后台的量不大。cache记录红包个数，原子操作进行个数递减，到0表示被抢光。

# 可用性/高可用

# 网站可用性度量

* 网站不可用时间=故障修复时间点-故障发现时间点
* 网络年度可用性指标=（1-网站不可用时间/年度总时间) \* 100%
* 对于大多数网站来说，2个9是基本可用，3个9是较高可用，4个9是具有自动恢复能力的高可用。

# 高可用的网站架构（冗余备份+失效转移)

* 实现高可用架构的主要手段是数据和服务的冗余备份和失效转移。
* 位于应用层的服务器通常会使用负载均衡设备，通过心跳检测监控某台服务器不可用时，会将其从集群列表中剔除，使整个集群保持可用。
* 位于服务层的服务器也是使用集群，使用服务注册查找中心对提供服务的服务器进行心跳检测。
* 位于数据层的服务器需要在数据写入时进行同步复制，实现数据冗余备份。

# 高可用的应用（集群)

* 负载均衡；Session管理

# 高可用的服务（集群)

* 除了使用服务注册查找中心进行心跳检测外，还可以：
  + 1)分级管理：核心应用和服务优先使用更好的硬件，在运维响应速度上也格外迅速。
  + 2)超时设置：在应用程序中设置服务调用的超时时间，一旦超时，通信框架就抛出异常，应用程序根据服务调度策略，可以选择继续重试或者将请求转移到提供相同服务的其他服务器上。
  + 3)异步调用：消息队列
  + 4)服务降级：为了保证高并发下核心应用和功能的正常运行，需要对服务进行降级。降级有两种手段：拒绝服务和关闭服务。
* 拒绝服务：拒绝低优先级应用的调用，减少服务调用并发数，或者随机拒绝部分服务额调用
* 关闭服务：关闭部分不重要的服务，或者服务内部关闭部分不重要的功能
  + 5)幂等性设计

# 高可用的数据（主备+互备)

## 主备模式

* Active-Standby模式。当主机宕机时，备机接管主机的一切工作，待主机恢复正常后，按使用者的的设定以热备（自动)或冷备（手动)方式将服务切换到主机上运行。
* 在数据库部分，称之为MS模式（Master-Slave模式)。

## 互备模式

* 两台主机同时运行各自的服务工作且相互检测情况。在数据库部分，常见的互备是MM（多主模式)，指一个系统有多个master，每个master都可以读写，需根据时间戳或业务逻辑合并版本。

## 集群模式

* 有多个节点在运行，同时可以通过主控节点分担服务请求，比如Zookeeper。集群节点要特别解决主控节点的高可用问题。

## Fail\*

### Fail-Over 失效转移

* Fail-Over的含义为“失效转移”，是一种备份操作模式，当主要组件异常时，其功能转移到备份组件。其要点在于有主有备，且主故障时备可启用，并设置为主。如Mysql的双Master模式，当正在使用的Master出现故障时，可以拿备Master做主使用。

### Fail-Safe 失效安全

* Fail-Safe的含义为“失效安全”，即使在故障的情况下也不会造成伤害或者尽量减少伤害。维基百科上一个形象的例子是红绿灯的“冲突监测模块”当监测到错误或者冲突的信号时会将十字路口的红绿灯变为闪烁错误模式，而不是全部显示为绿灯。

### Fail-Fast 快速失败

* 从字面含义看就是“快速失败”，让可能的错误尽早的被发现，对应的方式是“fault-tolerant（错误容忍)”。以JAVA集合（Collection)的快速失败为例，当多个线程对同一个集合的内容进行操作时，就可能会产生fail-fast事件。例如：当某一个线程A通过iterator去遍历某集合的过程中，若该集合的内容被其他线程所改变了；那么线程A访问集合时，就会抛出ConcurrentModificationException异常，产生fail-fast事件。

# 高可用网站的软件质量保证

* 网站发布；自动化测试；预发布验证；代码控制（主干发布、分支开发)
* ；自动化发布；灰度发布

# 网站运行监控

# 伸缩性

# 网站架构的伸缩性设计

## 不同功能进行物理分离实现伸缩

* 纵向分离（分层后分离)：将业务处理流程上的不同部分分离部署
* 横向分离（业务分隔后分离)：将不同的业务模块分离部署

## 单一功能通过集群实现伸缩

* 将相同服务部署在多台服务器上构成一个集群整体对外提供服务。
* 具体来说，集群伸缩性又可分为应用服务器集群伸缩性和数据服务器集群伸缩性。这两周集群由于对数据状态管理的不同，技术实现也有很大区别。而数据服务器集群也可以分为缓存数据服务器集群和存储数据服务器集群，这两种集群的伸缩性也不同。

# 应用服务器集群的伸缩性设计

* 无状态服务器集群，主要依赖于负载均衡。

## 1)HTTP重定向负载均衡

* 利用HTTP重定向协议实现负载均衡。
* HTTP重定向服务器是一台普通的应用服务器，其唯一的功能就是根据用户的HTTP请求计算一台真实的Web服务器的地址，并将该Web服务器地址写入HTTP重定向响应中返回给用户浏览器。
* 优点是比较简单，缺点是浏览器需要两次请求服务器才能完成一次访问，性能较差；重定向服务器自身的处理能力有可能成为瓶颈，整个集群的伸缩性规模有限；使用HTTP302重定向，有可能使搜索引擎判断为SEO作弊，降低搜索排名。

## 2)DNS域名解析负载均衡

* 这是利用DNS处理域名解析请求的同时进行负载均衡处理的一种方案。
* 浏览器访问A记录，DNS服务器保存如下信息：
* www.mysite.com IN A 114.100.80.1
* www.mysite.com IN A 114.100.80.2
* www.mysite.com IN A 114.100.80.3
* 每次域名解析请求都会根据负载均衡算法计算一个不同的IP地址返回，这样多个服务器就构成一个集群，并可以实现负载均衡。
* 优点是将负载均衡的工作转交给DNS，省掉了网站管理维护负载均衡服务器的麻烦，同时很多DNS还支持基于地理位置的解析，即会将域名解析成距离用户最近的一个服务器地址，可以加快用户访问速度。但是缺点是DNS是多级解析，每一级DNS都可能缓存A记录，当下线某台服务器后，即使修改DNS的A记录，要使其生效也需要较长时间。并且DNS负载均衡的控制权在域名服务商那里，网站无法对其做更多改善和更强大的管理。
* 事实上，大型网站总是部分使用DNS域名解析，利用域名解析作为第一层负载均衡手段，即域名解析得到的一组服务器是同样提供负载均衡服务的内部服务器，这组内部负载均衡服务器再进行负载均衡，将请求分发到真实的Web服务器上。

## 3)反向代理负载均衡

* 利用反向代理服务器进行负载均衡。
* 反向代理服务器一般也提供负载均衡的概念，管理一组Web服务器。由于Web服务器不直接对外提供访问，因此Web服务器不需要使用外部IP地址，而反向代理服务器则需要配置双网卡和内部外部两套IP地址。
* 由于反向代理服务器转发请求在HTTP协议层面，因此也叫应用层负载均衡，优点是和反向代理服务器功能集中在一起，部署简单，缺点是反向代理服务器是所有请求和响应的中转站，其性能可能会成为瓶颈。

### 正向代理与反向代理

* 正向代理隐藏了真实的客户端，反向代理隐藏了真实的服务器。

### 反向代理与负载均衡

* 现在许多大型web网站都用到反向代理。除了可以防止外网对内网服务器的恶性攻击、缓存以减少服务器的压力和访问安全控制之外，还可以进行负载均衡，将用户请求分配给多个服务器。

## 4)IP负载均衡

* 在网络层通过修改请求目标地址进行负载均衡。
* 用户请求IP数据报到达负载均衡服务器后，负载均衡服务器在操作系统内核进程获取网络数据包，根据负载均衡算法的得到一台真实Web服务器，然后将数据目的IP地址修改为真实Web服务器的IP地址，不需要通过用户进程处理。真实Web服务器处理完毕后，响应数据包回到负载均衡服务器，负载均衡服务器再将数据包源地址修改为自身的IP地址发送给用户浏览器。
* 这里的关键在于Web服务器响应数据包如何返回给负载均衡服务器。一种方案是负载均衡服务器在修改目的IP地址的同时修改源地址，将数据包源地址设为自身IP，即源地址转换SNAT；另一种方案是将负载均衡服务器作为真实Web服务器集群的网关服务器，这样所有响应数据都会到达负载均衡服务器。
* IP负载均衡在内核进程完成数据分发，较反向代理负载均衡（应用层负载均衡)有更好的处理性能，但是由于所有请求响应都需要经过负载均衡服务器，集群的最大响应数据吞吐量不得不受制于负载均衡服务器网卡带宽。对于提供下载服务或视频服务等需要传输大量数据的网站而言，难以满足需求。能否让负载均衡服务器只分发请求，而使响应数据从真实服务器直接返回给用户呢？

## 5)数据链路层负载均衡

* 数据链路层负载均衡是指在通信协议的数据链路层修改MAC地址进行负载均衡。
* 这种数据传输方式又称为三角传输模式。负载均衡数据分发过程中不修改IP地址，只修改目的MAC地址，通过配置真实Web服务器集群所有机器虚拟IP和负载均衡服务器IP地址一致，从而达到不修改数据报的源地址和目的地址就可以进行数据分发的目的。因为它们的IP地址是一致的（真实Web服务器IP==数据报目的IP)，不需要通过负载均衡服务器进行地址转换，可以将响应数据包直接返回给浏览器。这种负载均衡方式又称为直接路由方式DR。
* 使用三角传输模式的数据链路层负载均衡是目前大型网站使用最广的一种负载均衡手段。在Linux平台上最好的数据链路层负载均衡产品是LVS。

## 四层负载均衡与七层负载均衡

* 四层负载均衡，也就是主要通过报文中的目标地址和端口，再加上负载均衡设备设置的服务器选择方式，决定最终选择的内部服务器。
* 以常见的TCP为例，负载均衡设备在接收到第一个来自客户端的SYN 请求时，即通过上述方式选择一个最佳的服务器，并对报文中目标IP地址改为后端服务器IP，直接转发给该服务器。TCP的连接建立，即三次握手是客户端和服务器直接建立的，负载均衡设备只是起到一个类似路由器的转发动作。
* 上面的IP负载均衡，数据链路层负载均衡都是属于四层负载均衡
* 七层负载均衡：也称为“内容交换”，主要通过报文中真正有意义的应用层内容，再加上负载均衡设备设置的服务器选择方式，决定最终选择的内部服务器。
* 以常见的HTTP为例，负载均衡设备要根据真正的应用层内容再选择服务器，必须先代理实际服务器和客户端建立连接(三次握手)后，才可能接受到客户端发送的真正应用层内容的报文，然后再根据该报文中的特定字段，加上负载均衡设备设置的服务器选择方式，决定最终选择的内部服务器。
* 在这种情况下，负载类似于一个代理服务器与前端的客户端以及后端的服务器会分别建立TCP连接。所以，从技术原理上来看，七层负载均衡明显的对负载均衡设备的要求更高，设备性能消耗也更大。
* 上面的反向代理负载均衡，属于七层负载均衡

## 负载均衡的演进整合

### 硬负载均衡——F5

* 由于小型机和前面的 F5 负载均衡硬件都比较贵，所以出于可靠性、可维护性和成本的综合考虑，一般应用部署两套跑在两台小型机上，在前面共享一个 F5 做负载均衡。而一般 F5 和小型机这类硬件设备都至少是 5 个 9 的可靠性保障，所以整体的系统可靠性基本有保障。
* 进入互联网时代后，应用开发拥抱开源，部署使用更廉价的 PC Server 和免费开源的应用容器。负载均衡也逐步从硬负载向软负载变迁，由于互联网应用的海量特性和部署规模的急剧膨胀，前端负载均衡也开始变得丰富起来。

### 软负载均衡——LVS+Keepalived

* 负载均衡（DNS/nginx/LVS/vipserver)
* 高可用集群（heardbead/keepalived )
* heardbead与keepalived的区别，脑裂问题的解决，高可用方案的优劣
* LVS四种模式

### 两层负载均衡——LVS(+Keepalived)+Nginx

* 为了方便做按域名的分流和适配切流量上线，中间又加了一层 Nginx。这样就变成了两层软负载结构了，LVS 负责 4 层（传输层)，Nginx 负责 7 层（应用层)。 但 Nginx 只负责了单机内多实例的负载均衡，这里主要是因为当时 PC Server 是物理机，CPU 16/32 core，内存 32/64G 不等，为了更充分的利用资源，一台物理机上都部署了多个应用服务实例，而考虑到 Nginx 工作在 7 层的开销远高于 LVS/DR 模式，所以一般在一个 Nginx 后面挂的实例数也不会超过 10 个。

### 三层负载均衡——LVS(+Keepalived)+HAProxy+Nginx

* 随着业务发展和用户流量上升，机器规模也在不断扩张，导致一个网段内的 IP 都不够用了，这套负载结构又遇到了横向扩展的瓶颈，因为 LVS/DR 模式下跨不了网段。所以后来又在 LVS 和 Nginx 之间加了一层 HAProxy。其实加了 HAProxy 之后，它也是工作在 7 层（应用层)，这样 Nginx 这层看起来就不是很有必要。但三层的负载结构能支撑更大规模的集群，而原本在 Nginx 层做了一套方便研发切流量上线的运维管理系统，所以牺牲一点性能换取现在的可维护性和将来扩展性，Nginx 这层就一直保留下来了。而且 Nginx 相比 HAProxy 不是纯粹的负载均衡器，它还能提供 cache 功能，对于某些 HTTP 请求实际只走到 Nginx 这层就可以通过缓存命中而返回。

### 四层负载均衡——DNS+LVS(+Keealived)+HAProxy+Nginx

* 随着业务发展，公司开始了多个 IDC 的建设，考虑到 IDC 级别的容灾，集群开始部署到多个 IDC。跨 IDC 的负载均衡方案可以简单通过 DNS 轮询来实现，但可控性不好。所以我们没有采用这种，而是采用一主加多子域名的方式来基于业务场景实现动态域名调度和负载。主域名下实际是一个动态流量调度器，跨多个 IDC 部署，对于 HTTP 请求基于重定向方式跳子域名，对于 TCP 方式每次建立长连接前请求分配实际连接的子域名。

### 五层负载均衡——CDN+DNS+LVS(+Keepalived)+HAProxy+Nginx

* 最后再加上互联网应用必不可少的 CDN 将静态资源请求的负载分流，那么整个负载的层次结构就完整了。

### SSL带来的负载均衡变化

* 随着互联网的普及，安全问题益发严重，原本早期只有银行网银等使用 HTTPS 方式访问，现在电商类网站也开始启用全站 HTTPS 了。引入 SSL 后对负载结构带来了什么影响么？SSL 属于应用层的协议，所以只能在 7 层上来做，而 HAProxy 也是支持 SSL 协议的，所以一种方式是只需简单的让 HAProxy 开启 SSL 支持完成对内解密对外加密的处理。
* 但 HAProxy 的作者不太赞同这种方案，因为引入 SSL 处理是有额外的性能开销的。那么在承担确定流量的情况下，假设原本需要 M 台 HAProxy，在开启了 SSL 后可能需要 M + N 台 HAProxy。随着流量增长，这种方式的横向扩展成本较高（毕竟 SSL 证书按服务器数量来收费的)。他给出的解决方案是再独立一层 SSL 代理缓存层，像下面这样。
* L4 和 L7 之间独立的 SSL 代理缓存层只负责 SSL 协议的处理，把 HTTPS 转换成 HTTP，并检查本地缓存是否命中。若未命中再转发请求到后端的 L7 层应用负载均衡层。这样的好处是每个层次都可以根据流量来独立伸缩，而且 SSL 层显然可以跨多个应用共享，更节省成本。如果按这个思路来重新调整我们前面的负载均衡结构层次，将会演变成下面这样。
* 其实，这时我觉得应用前面的那层 Nginx 可能就显得多余了点，不是必需的。但如果现实这么演进下来很可能就会有这么一层冗余的东西存在很长一段时间，这就是理想和现实之间的差距吧。

## 负载均衡算法

### 轮询（Round Robin，RR)

* 所有请求被依次分发到每台应用服务器上，即每台服务器需要处理的请求数目都相同，适用于所有服务器硬件都相同的场景。

### 加权轮询（Weighted Round Robin，WRR)

* 根据应用服务器硬件性能的情况，在轮询的基础上，按照配置的权重将请求分发到每个服务器，高性能的服务器能分配更多请求。

### 随机（Random)

* 请求被随机分配到各个应用服务器，在许多场合下，这个方案都很简单实用，因为好的随机数本身就很均衡。即使应用服务器硬件配置不同，也可以使用加权随机算法。

### 最少连接（Least Connections)

* 记录每个应用服务器正在处理的连接数/请求数，将新到的请求分发到最少连接的服务器上。

### 源地址散列（Source Hashing)

* 根据请求来源的IP地址进行Hash计算，得到应用服务器，这样来自同一个IP地址的请求总会在同一个服务器上处理，该请求的上下文信息可以存储在这台应用服务器上，这一个会话周期内重复使用，实现会话粘滞（Session Sticky)。

# 分布式缓存集群的伸缩性设计

* 与应用服务器集群的伸缩性设计不同，分布式缓存服务器集群中不同服务器中缓存的数据不同，缓存访问请求不可以在缓存服务器集群中的任意一台处理，必须先找到有所需数据的缓存服务器，然后才能访问。这个特点会严重制约分布式缓存服务器集群的伸缩性设计，因为新上线的缓存服务器没有缓存任何数据，而已下线的缓存服务器上还缓存着许多热点数据。
* 必须让新上线的缓存服务器对整个分布式缓存集群影响最小，也就是说新加入缓存服务器后应使整个缓存服务器集群中已经缓存的数据尽可能还被访问到，这是分布式缓存集群伸缩性设计的主要目标。
* 使用传统哈希算法（比如余数)的话，假设是3台服务器扩容至4台服务器，大约有3/4被缓存的数据不能命中，随着服务器集群规模的扩大，这个比例线性上升。
* 一致性哈希算法可以解决这个问题。

## 一致性哈希

* 一致性哈希算法在1997年由麻省理工学院提出的一种分布式哈希（DHT)实现算法，设计目标是为了解决因特网中的热点(Hot spot)问题，初衷和CARP十分类似。一致性哈希修正了CARP使用的简 单哈希算法带来的问题，使得分布式哈希（DHT)可以在P2P环境中真正得到应用。
* 一致性hash算法提出了在动态变化的Cache环境中，判定哈希算法好坏的四个定义：
* 1、平衡性(Balance)：平衡性是指哈希的结果能够尽可能分布到所有的缓冲中去，这样可以使得所有的缓冲空间都得到利用。很多哈希算法都能够满足这一条件。
* 2、单调性(Monotonicity)：单调性是指如果已经有一些内容通过哈希分派到了相应的缓冲中，又有新的缓冲加入到系统中。哈希的结果应能够保证原有已分配的内容可以被映射到原有的或者新的缓冲中去，而不会被映射到旧的缓冲集合中的其他缓冲区。
* 3、分散性(Spread)：在分布式环境中，终端有可能看不到所有的缓冲，而是只能看到其中的一部分。当终端希望通过哈希过程将内容映射到缓冲上时，由于不同终端所见的缓冲范围有可能不同，从而导致哈希的结果不一致，最终的结果是相同的内容被不同的终端映射到不同的缓冲区中。这种情况显然是应该避免的，因为它导致相同内容被存储到不同缓冲中去，降低了系统存储的效率。分散性的定义就是上述情况发生的严重程度。好的哈希算法应能够尽量避免不一致的情况发生，也就是尽量降低分散性。
* 4、负载(Load)：负载问题实际上是从另一个角度看待分散性问题。既然不同的终端可能将相同的内容映射到不同的缓冲区中，那么对于一个特定的缓冲区而言，也可能被不同的用户映射为不同 的内容。与分散性一样，这种情况也是应当避免的，因此好的哈希算法应能够尽量降低缓冲的负荷。
* 环形Hash空间
* 按照常用的hash算法来将对应的key哈希到一个具有2^32次方个桶的空间中，即【0,2^32-1】的数字空间中。现在我们可以将这些数字头尾相连，想象成一个闭合的环形。然后根据需要缓存的数据的Key值计算其hash值，然后在Hash环上顺时针查找距离这个key的hash值最近的节点，完成key到节点的hash映射查找。
* 一致性哈希所带来的最大变化是把节点对应的哈希值变成了一个范围。我们会把整个哈希值的范围定义得非常大，然后把这个范围分配给现有的节点。如果有节点加入，那么这个新节点会从原有的某个节点上分管一部分范围的哈希值；如果有节点退出，那么这个节点原来管理的哈希值会给它的下一个节点来管理。
* 3台服务器扩容至4台服务器，可以继续命中原有缓存数据的概率是75%。随着集群规模的扩大，继续命中原有缓存数据的概率也逐渐增大。
* 具体应用中，这个长度为2^32的一致性Hash环通常使用二叉查找树来实现，Hash查找过程实际上是在二叉查找树中查找不小于查找树的最小数值。当然这个二叉树的最右边叶子节点与最左边的叶子节点相连接，构成环。
* 仍存在问题：新增节点时，除了新增的节点外，只有一个节点受影响，这个新增节点和受影响的节点的负载是明显比其他节点低的；减少节点时，除了减去的节点外，只有一个节点受影响，它要承担自己原来的和减去的节点的工作，压力明显比其他结点要高。

## 虚拟节点对一致性哈希的改进

* 每个虚拟节点支持连续的哈希环上的一段，这时如果加入一个物理节点，相应就会加入很多虚拟节点，这些新的虚拟节点是相对均匀地插入到整个哈希环上的，这样，就可以很好地分担现有物理节点的压力了；如果减少一个物理节点，对应的很多虚拟节点就会失效，这样，就会有很多剩余的虚拟节点来承担之前虚拟节点的工作，但对于物理节点来说，增加的负载相对是均衡的。所以可以通过一个物理节点对应非常多的虚拟节点，并且同一个物理节点的虚拟节点尽量均匀分布的方式来解决增加或减少节点时负载不均衡的情况。
* 新加入节点NODE3对应的一组虚拟节点为V30，V31，V32，加入到一致性Hash环上后，影响V01，V12，V22三个虚拟节点，而这三个虚拟节点分别对应NODE0，NODE1，NODE2三个物理节点。因此加入一个节点会影响集群中已存在的三个节点。

## 分片

* 分片将哈希环切割为相同大小的分片，然后将这些分片交给不同的节点负责。注意这里跟上面提到的虚拟节点有着很本质的区别，分片的划分和分片的分配被解耦，一个节点退出时，其所负责的分片并不需要顺时针合并给之后节点，而是可以更灵活的将整个分片作为一个整体交给任意节点，实践中，一个分片多作为最小的数据迁移和备份单位。
* 而也正是由于上面提到的解耦，相当于将原先的key到节点的映射拆成两层，需要一个新的机制来进行分片到存储节点的映射，由于分片数相对key空间已经很小并且数量确定，可以更精确地初始设置，并引入中心目录服务来根据节点存活修改分片的映射关系，同时将这个映射信息通知给所有的存储节点和客户端。
* 常见的存储系统大多采用类似于分片的数据分布和定位方式：
* Dynamo及Cassandra采用分片的方式并通过Gossip在对等节点间同；
* Redis Cluster将key space划分为slots，同样利用Gossip通信；
* Zeppelin将数据分片为Partition，通过Meta集群提供中心目录服务；
* Bigtable将数据切割为Tablet，类似于可变的分片，Tablet Server可以进行分片的切割，最终分片信息记录在Chubby中；
* Ceph采用CRUSH方式，由中心集群Monitor维护并提供集群拓扑的变化。

# 数据存储服务器集群的伸缩性设计

* 分库分表
* 扩容时数据迁移可以利用一致性Hash算法。路由模块使用一致性Hash算法进行路由，尽量使需要迁移的数据最少。但是迁移数据需要遍历数据库中每条记录的索引，重新进行路由计算确定其是否需要迁移，这会对数据库访问造成压力，并且需要解决迁移过程中数据的一致性、可访问性、迁移过程中服务器宕机时的可用性等诸多问题。

# 扩展性

* 主要依赖于分布式消息队列和服务化。

# 分布式消息队列

* 见《面试---8.分布式理论---组件》

# SOA

## 1、简介

* SOA是面向服务架构，它强调系统之间以标准的服务方式进行交互，各系统可采用不同的语言、不同的框架来实现，交互则全部通过服务的方式进行。

## 2、实现

* SCA
* ESB
* Tuscany
* Mule

# 微服务

* 微服务不再强调传统SOA架构里面比较重的ESB企业服务总线，同时SOA的思想进入到单个业务系统内部，实现真正的组件化。
* 微服务只是一种为经过良好架构设计的SOA解决方案，是面向服务的交付方案。
* 微服务更趋向于以自治的方式产生价值。
* 微服务与敏捷开发的思想高度结合在一起，服务的定义更加清晰，同时减少了企业ESB开发的复杂性。
* 微服务是soa思想的一种提炼！
* SOA是重ESB，微服务是轻网关。
* 微服务可以在“自己的程序”中运行，并通过“轻量级设备与HTTP型API进行沟通”。关键在于该服务可以在自己的程序中运行。通过这一点我们就可以将服务公开与微服务架构（在现有系统中分布一个API)区分开来。在服务公开中，许多服务都可以被内部独立进程所限制。如果其中任何一个服务需要增加某种功能，那么就必须缩小进程范围。在微服务架构中，只需要在特定的某种服务中增加所需功能，而不影响整体进程。
* 微服务不需要像普通服务那样成为一种独立的功能或者独立的资源。定义中称，微服务是需要与业务能力相匹配，这种说法完全正确。不幸的是，仍然意味着，如果能力模型粒度的设计是错误的，那么，我们就必须付出很多代价。如果你阅读了Fowler的整篇文章，你会发现，其中的指导建议是非常实用的。在决定将所有组件组合到一起时，开发人员需要非常确信这些组件都会有所改变，并且规模也会发生变化。服务粒度越粗，就越难以符合规定原则。服务粒度越细，就越能够灵活地降低变化和负载所带来的影响。然而，利弊之间的权衡过程是非常复杂的，我们要在配置和资金模型的基础上考虑到基础设施的成本问题。
* Hystrix服务熔断、Zuul服务网关、Ribbon客户端负载均衡、Stream消息驱动、Eureka服务发现、Config配置中心、Consul服务注册

# 安全性

* 可以参考《面试---5. JavaWeb&HTTP&安全》

# 网站攻防

## XSS

## SQL注入

## CSRF

## 其他

## Web应用防火墙

# 信息加密与密钥管理

## 单向散列加密

## 对称加密

## 非对称加密

## 密钥安全管理

# 信息过滤与反垃圾

## 文本匹配

* 敏感词过滤
* 可以使用正则表达式，但效率较低。
* 一般采用Trie树的变种。

## 分类算法

* 贝叶斯分类算法

## 黑名单

* hash表，但是会占用很多内存，如果hash表过大就无法解决了。

### 布隆过滤器

* 通过一个二进制列表和一组随机数映射函数实现。
  + 如果需要处理10亿邮件列表黑名单，在内存中建立一个2GB大小的存储空间，即16G个bit，并全部初始化为0。要将一个邮箱地址加入黑名单时，使用8个随机映射函数（F1~F8)得到0~16G范围内的8个随机数，从而将该邮箱地址勇摄到16Gbit的8个位置上，然后将这些位置值1。当要检查一个邮箱地址是否在黑名单时，使用同样的映射函数，得到16G空间8个位置上的bit，如果这些值都为1，那么该邮箱地址在黑名单上。
* 处理同样数量的信息，布隆过滤器只需要哈希表所需内存的1/8。
* 缺点是可能会误算，比如一个邮箱地址映射的8个bit正好都被其他邮箱地址设为1了。这种可能性极小，但如果需要精确的判断，则不适合布隆过滤器。
* 底层使用的是位图。当一个元素被加入集合时，通过 K 个 Hash 函数将这个元素映射成一个位阵列（Bit array)中的 K 个点，把它们置为 1。检索时，我们只要看看这些点是不是都是 1 就（大约)知道集合中有没有它了：
* 1、如果这些点有任何一个 0，则被检索元素一定不在；
* 2、如果都是 1，则被检索元素很可能在。
* 布隆过滤器的优点 : 空间效率和查询时间都远远超过一般的算法，布隆过滤器存储空间和插入 / 查询时间都是常数O(k)。另外, 散列函数相互之间没有关系，方便由硬件并行实现。布隆过滤器不需要存储元素本身，在某些对保密要求非常严格的场合有优势。
* 布隆过滤器的缺点：误算率是其中之一。随着存入的元素数量增加，误算率随之增加。但是如果元素数量太少，则使用散列表足矣。(误判补救方法是：再建立一个小的白名单，存储那些可能被误判的信息。)
* 另外，一般情况下不能从布隆过滤器中删除元素。我们很容易想到把位数组变成整数数组，每插入一个元素相应的计数器加 1, 这样删除元素时将计数器减掉就可以了。然而要保证安全地删除元素并非如此简单。首先我们必须保证删除的元素在布隆过滤器里面. 这一点单凭这个过滤器是无法保证的。另外计数器回绕也会造成问题。

# 大型网站典型故障案例分析

# 写日志也会引发故障

* log等级设置较低导致大量写入log。
* 经验教训：
  + 1)应用程序自己的日志输出配置和第三方组件日志输出要分别配置
  + 2)检查log配置文件，日志输出级别至少为WARN
  + 3)有可能需要关闭第三方库的日志输出

# 高并发访问数据库引发的故障

* 某条SQL频繁执行导致数据库Load居高不下
* 经验教训：
  + 1)首页不应该访问数据库，首页需要的数据可以从缓存服务器或者搜索引擎服务获取
  + 2)首页最好是静态的

# 高并发情况下锁引发的故障

* 某个单例对象使用了synchronized(this)导致排队。
* 经验教训：使用锁要谨慎。

# 缓存引发的故障

* 缓存服务不受重视，缓存服务器被失误关闭，导致数据库崩溃。
* 经验教训：缓存已经成为网站架构不可或缺的一部分时，对缓存的管理就需要提高到和其他服务器一样的级别。

# 应用启动不同步引发的故障

* 后台服务准备好之前，前台应用就启动了，导致大量请求阻塞。
* 经验教训：在应用程序中加入一个特定的动态页面，启动脚本先启动后台服务，然后在脚本中不断用curl命令访问这个特定页面，直到可以访问成功，才启动前台应用。

# 大文件读写独占磁盘引发的故障

* 数百M的大文件读写时独占磁盘，导致其他用户的文件操作缓慢。
* 经验教训：存储的使用需要根据不同文件类型和用途进行管理，小文件和大文件应该使用专门的存储服务器。

# 滥用生产环境引发的故障

* 线上生产环境进行压测导致网站访问延迟过高。
* 经验教训：访问线上生产环境要规范，不小心就会导致大事故

# 不规范的流程引发的故障

* 注释代码后在上线前忘记把注释去掉，直接提交代码库被发布到线上环境
* 经验教训：代码提交前使用diff命令进行代码比较，确认没有提交不该提交的代码；
* 加强code review，代码在正式提交前必须被至少一个其他工程师做过code review，并且共同承担因代码引起的故障责任。