一、概述
1.大型网站架构演化
1.1大型网站软件系统的特点
1.2大型网站架构演化过程
1.2.1初始阶段的网站架构
1.2.2应用服务的数据服务分离
1.2.3使用缓存改善网站性能
1.2.4使用应用服务器集群改善网站的并发处理能力
1.2.5数据库读写分离
1.2.6使用反向代理和CDN加速网站响应
1.2.7使用分布式文件系统和分布式数据库
1.2.8使用NoSQL和搜索引擎
1.2.9业务拆分
1.2.10分布式服务
1.3大型网站架构演化的价值观
1.4网站架构设计误区
2.大型网站架构模式
2.1网站的架构模式
2.1.1分层
2.1.2分割
2.1.3分布式
2.1.4集群
2.1.5缓存
2.1.6异步
2.1.7冗余
2.1.8自动化
2.1.9安全
3.大型网站核心架构要素
3.1性能
3.2可用性
3.3伸缩性
3.4扩展性
3.5安全性
二、架构
4.网站的高性能架构
4.1网站性能测试
4.1.1不同视角下的网站性能
4.1.1.1用户视角的网站性能
4.1.1.2开发人员视角的网站性能
4.1.1.3运维人员视角的网站性能
4.1.2性能测试指标
4.1.2.1响应时间
4.1.2.2并发数
4.1.2.3吞吐量
4.1.2.4性能计数器
4.1.3性能测试方法
4.1.4性能测试报告
4.1.5性能优化策略
4.2Web前端性能优化
4.2.1浏览器访问优化
4.2.2CDN加速

4.2.3反向代理
4.3应用服务器性能优化
4.3.1分布式缓存
4.3.1.1缓存的基本原理
4.3.1.2合理使用缓存
4.3.1.3分布式缓存架构
4.3.2异步操作
4.3.3使用集群
4.3.4代码优化
4.3.4.1多线程
4.3.4.2资源复用
4.3.4.3数据结构
4.3.4.4垃圾回收
4.4存储性能优化
4.4.1机械硬盘VS固态硬盘
4.4.2B+树VSLSM树
4.4.3RAID VS HDFS
5.网站的高可用架构
5.1网站可用性的度量和考核
5.1.1网站可用性度量
5.1.2网站可用性考核
5.2高可用的网站架构
5.3高可用应用
5.3.1通过负载均衡进行无状态服务的失效转移
5.3.2应用服务器集群的Session管理
5.3.2.1Session复制
5.3.2.2Session绑定
5.3.2.3利用Cookie记录Session
5.3.2.4Session服务器
5.4高可用服务
5.4.1分级管理
5.4.2超时设置
5.4.3异步调用
5.4.4服务降级
5.4.5幂等性设计
5.5高可用数据
5.5.1CAP原理
5.5.2数据备份
5.5.3失效转移
5.6高可用网站的软件质量保证
5.6.1网站发布
5.6.2自动化测试
5.6.3预发布验证
5.6.4代码控制
5.6.5自动化发布
5.6.6灰度发布
5.7网站运行监控
5.7.1监控数据采集
5.7.2监控管理
6.网站的伸缩性架构
6.1网站架构的伸缩性设计
6.1.1不同功能进行物理分离实现伸缩
6.1.2单一功能通过集群规模实现伸缩

6.2应用服务器集群伸缩性设计
6.2.1HTTP重定向负载均衡
6.2.2DNS域名解析负载均衡
6.2.3反向代理负载均衡
6.2.4IP负载均衡
6.2.5数据链路层负载均衡
6.2.6负载均衡算法
6.3分布式缓存集群伸缩性设计
6.3.1Memcached分布式缓存集群访问模型
6.3.2Memcached分布式缓存集群的伸缩性挑战
6.3.3分布式缓存的一致性Hash算法
6.4数据存储服务器集群伸缩性设计
6.4.1关系数据库集群的伸缩性设计
6.4.2NoSQL数据库伸缩性设计
7.网站的扩展性架构
7.1构建可扩展的网站架构
7.2利用分布式消息队列降低系统耦合性
7.2.1事件驱动架构
7.2.2分布式消息队列
7.3利用分布式服务打造可复用的业务平台
7.3.1Web Service与企业级分布式服务
7.3.2大型网站分布式服务的需求与特点
7.3.3分布式服务框架设计
7.4可扩展的数据结构
7.5利用开放平台建设网站生态圈
8.网站的安全架构
8.1网站应用攻击和防御
8.2信息加密技术及密钥安全管理
8.3信息过滤和反垃圾
8.4电子商务风险控制
三、案例

一、概述

本篇主要是从：

- 演化
- 模式
- 要素

三个方面描述网站的整体架构

1.大型网站架构演化

1.1大型网站软件系统的特点

- **高并发、大流量**

需要面对高并发用户和大量访问

- **高可用**

系统7×24小时不间断服务

- **海量数据**

需要存储管理海量数据，因此需要使用大量服务器

- **用户分布广泛，网络情况复杂**

由于各国网络情况，各运营商情况都不同，所以各地的网络情况也千差万别

- **安全环境恶劣**

由于互联网是开放的，所以网站更容易收到攻击

- **需求快速变更，发布频繁**

由于互联网产品要快速适应市场，满足客户需求，所以其产品的发布频率很高

- **渐进式发展**

所有的大型互联网站都是从一个小网站开始，渐进发展起来的

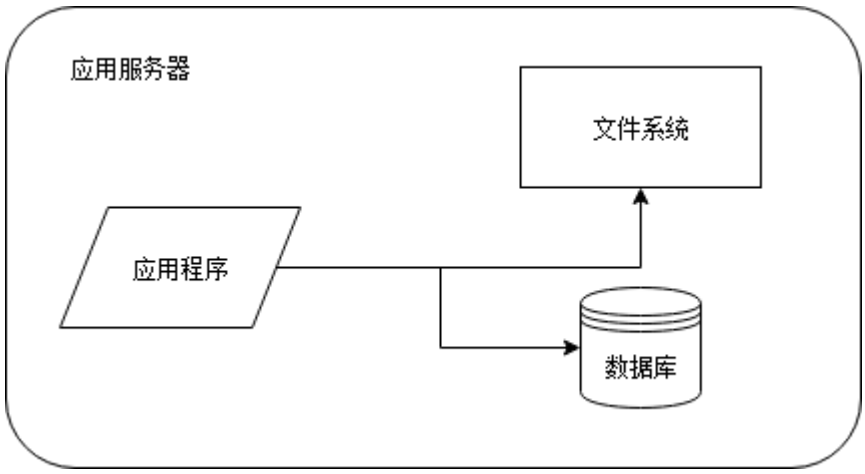
1.2大型网站架构演化过程

大型网站的技术挑战主要是：庞大的用户，高并发的访问，海量的数据

大型网站的架构就是要解决以上挑战的。

1.2.1初始阶段的网站架构

由于开始时，小型网站访问人数不多，只需要一台服务器即可。**将应用程序、数据库、文件等资源都放在该服务器上**，其架构图如下所示：



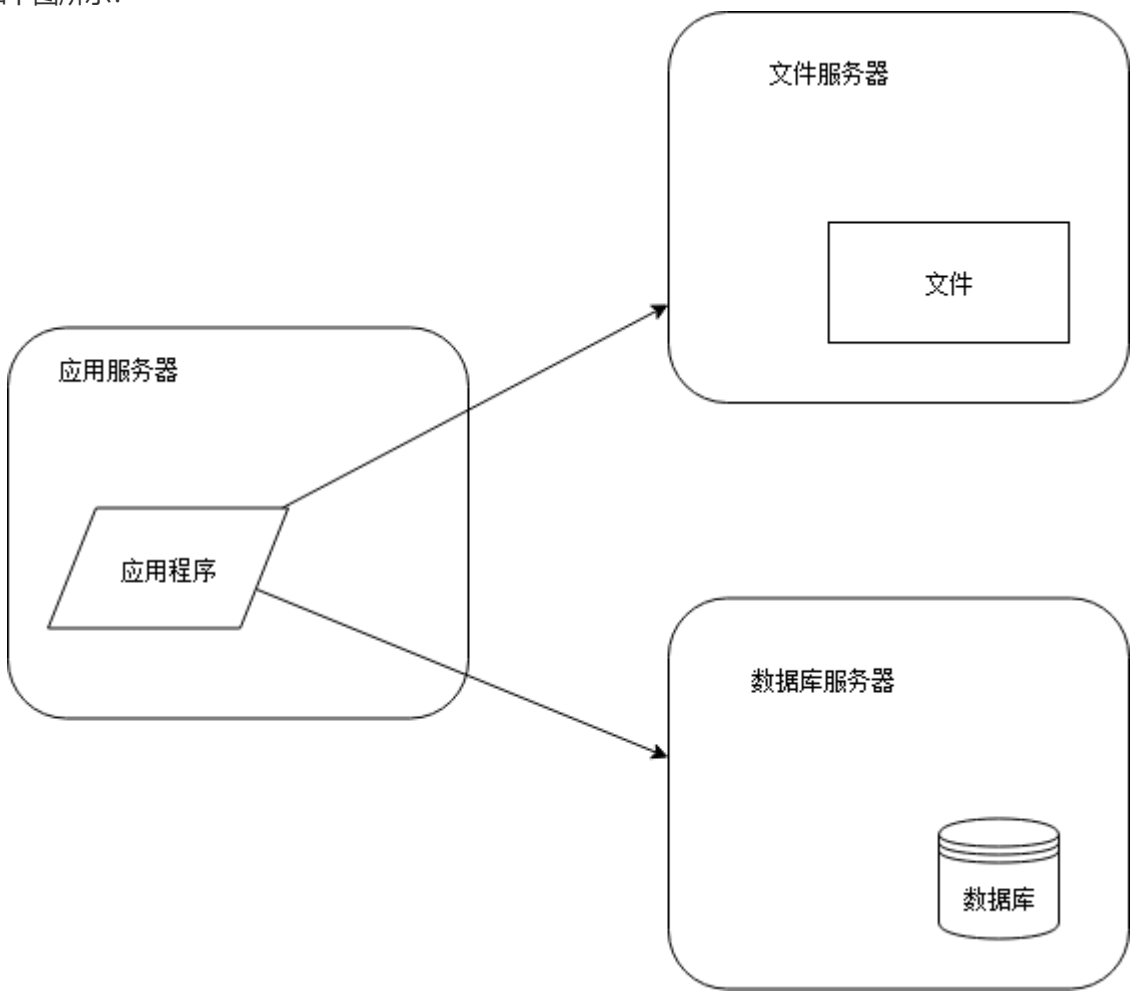
但是随着**越来越多的用户访问，网站的性能会变的越来越差，越来越多的数据也导致网站的存储空间不足**

1.2.2应用服务的数据服务分离

为了解决以上问题，将**应用和数据分离**，应用和数据分离之后，**不同服务器提供不同服务**，网站的并发处理能力和数据存储空间得到了很大改善，这样网站就使用三台服务器：

- 应用服务器：处理大量的业务逻辑---------->强大的CPU；
- 数据库服务器：快速磁盘检索和数据缓存---------->更快的磁盘、更大的内存；
- 文件服务器：存储大量用户上传的文件---------->更大的磁盘；

其架构如下图所示：



但是随着用户的增多，**数据库压力太大而造成的延迟访问会影响网站性能和用户体验。**

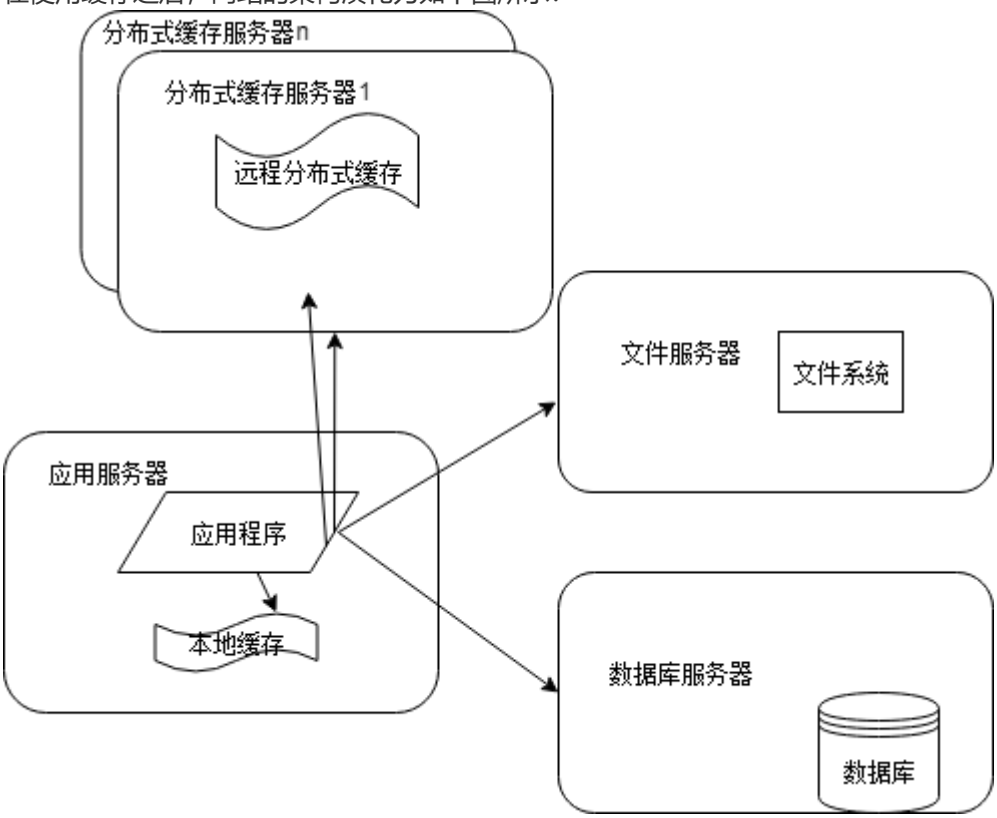
1.2.3使用缓存改善网站性能

由于互联网的数据访问遵循二八定律：80%的业务访问集中在20%的数据上，根据数据的这个特点，我们可以将**经常使用的20%的数据缓存在内存中，从而减少对数据库的访问，从而提升网站的数据访问速度。**

缓存可以分成两种：

- 本地缓存：访问速度更快，但是存储数据量受限；
- 远程缓存：远程分布式缓存使用集群的方式，使用内存较大的缓存服务器；

在使用缓存之后，网站的架构演化为如下图所示：

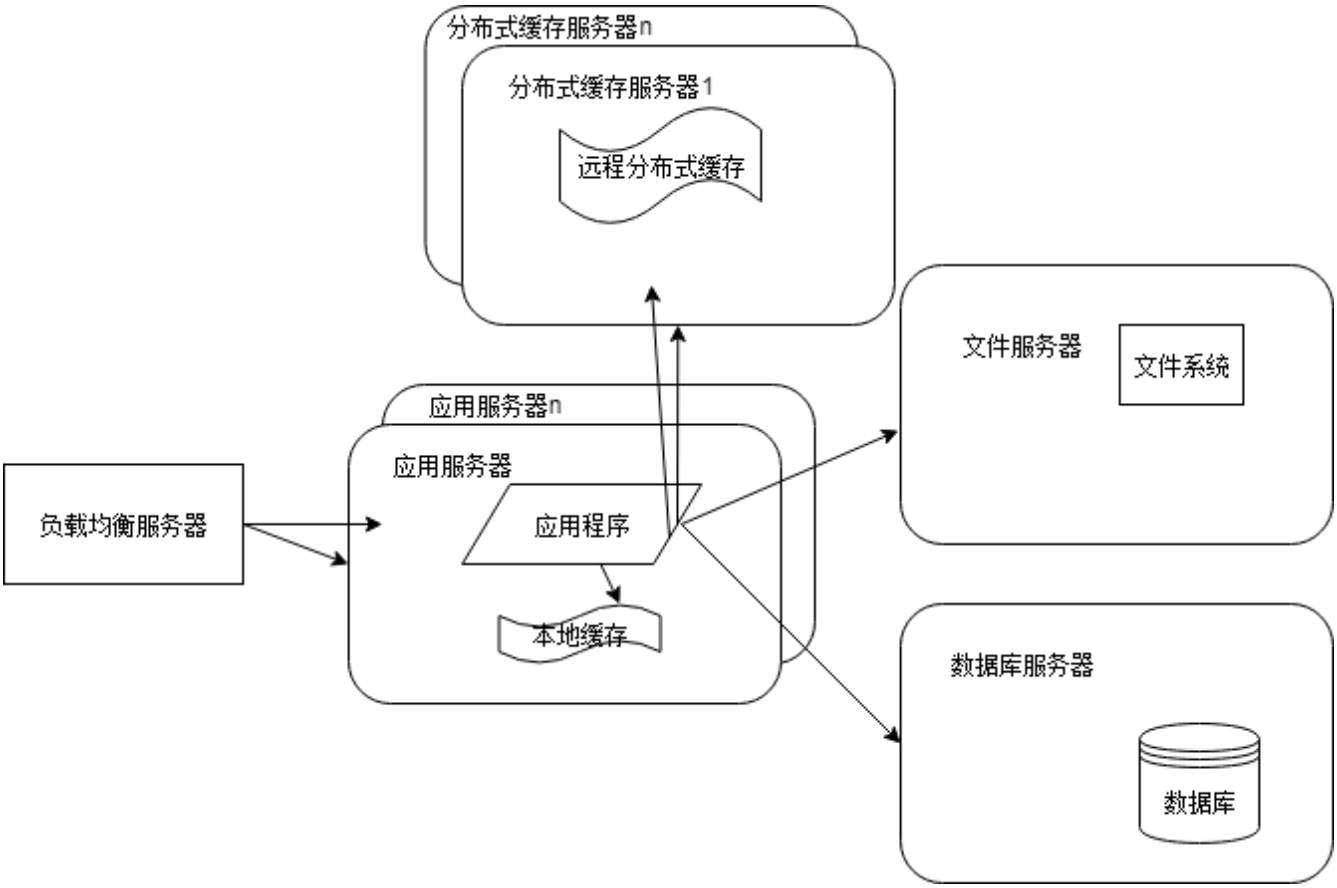


在**网站访问的高峰期，应用服务器**成为整个网站的瓶颈。

1.2.4使用应用服务器集群改善网站的并发处理能力

使用集群是网站解决高并发、海量数据的常用解决手段，当一台服务器处理能力、存储空间无法满足业务需求时，正确的做法是增加一台服务器来分担原来服务器的压力，而不是更换原来的服务器。

系统的伸缩性也是通过服务器集群的数量来体现的，在应用服务器中使用集群后，系统的架构如下图所示：**通过负载均衡服务器，可以将访问请求发送到应用服务器集群中的任何一台服务器中**

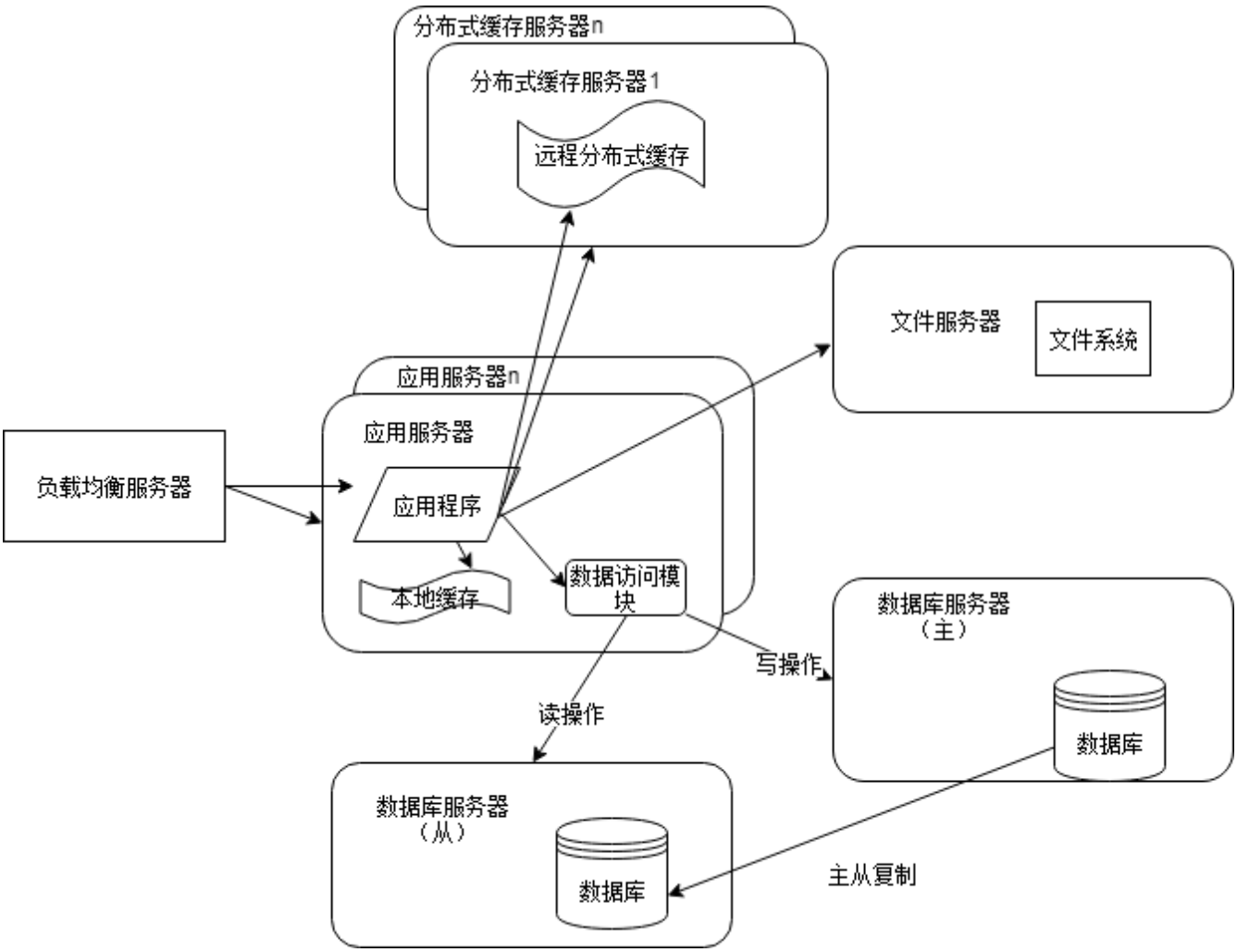


1.2.5数据库读写分离

即使在使用了缓存之后，仍然**有一部分读操作和全部的写操作都要对数据库进行访问**，在一定的访问规模时，数据库会因为负载过高成为系统的瓶颈。

- **主从复制**：配置多台数据库主从关系，可以将一台服务器（主服务器）上的数据同步到另一台服务器（从服务器）上；
- **使用读写分离来缓解数据库负载压力**：应用服务器在写数据时，访问主数据库；在读数据时，访问从数据库；

在使用读写分离之后，网站的架构演化成如下图所示：

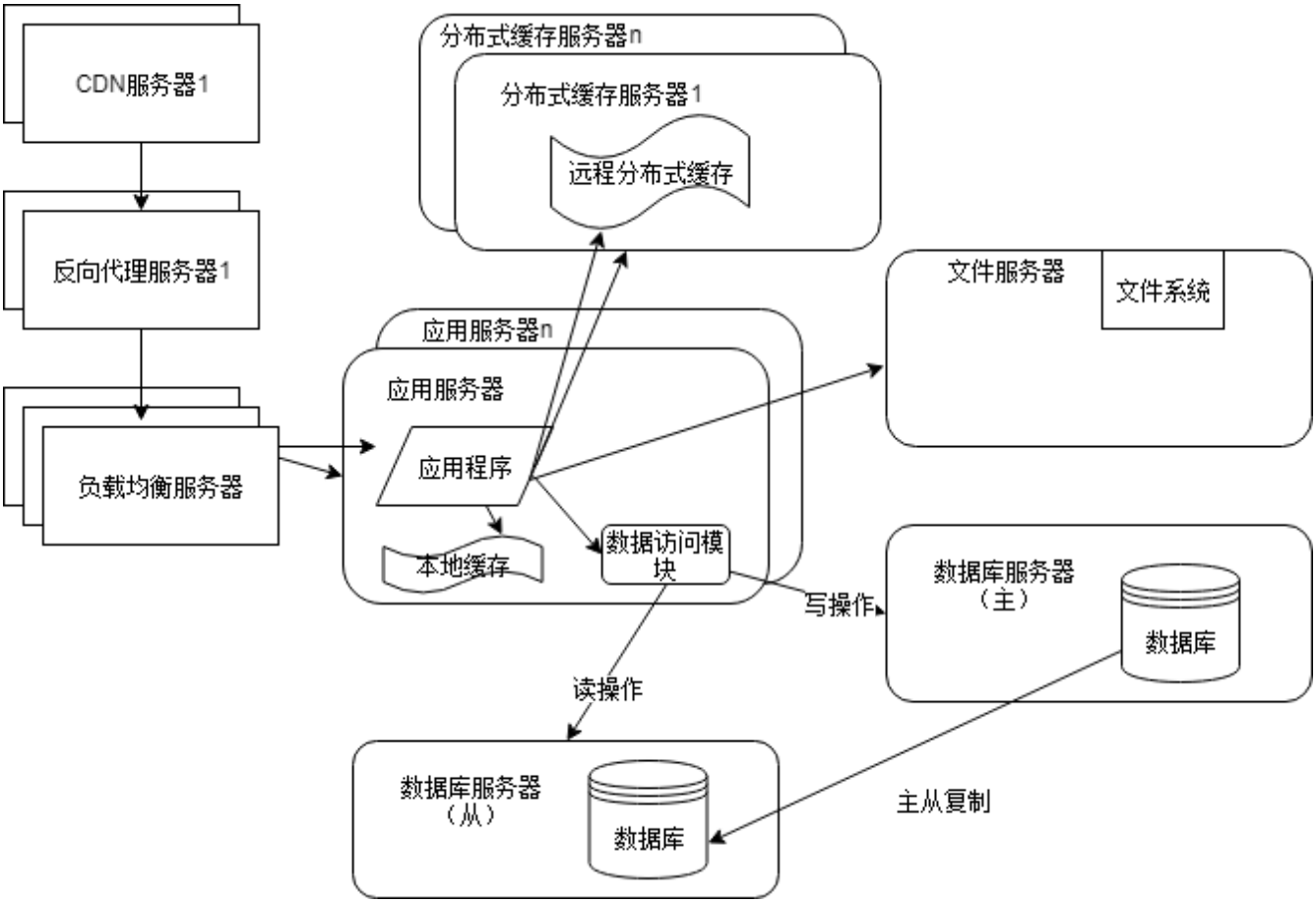


1.2.6使用反向代理和CDN加速网站响应

由于网络环境复杂，不同地区访问网站时，响应速度差别很大。通过使用**CDN和反向代理**，能加速网站访问速度，提升用户体验。 **CDN和反向代理的基本原理都是缓存**：

- **CDN是部署在网络提供商的机房中**，用户请求服务时，可以在离自己最近的网络提供商机房中获取数据；
- **反向代理部署在网站的中心机房**，当用户的请求到达中心机房时，先会访问反向代理服务器，如果反向代理服务器中包含用户请求的数据，则直接返回数据；
- CDN和反向代理的目的都是尽早返回数据给用户，既能加快用户访问速度，又能减轻服务器的负载压力；

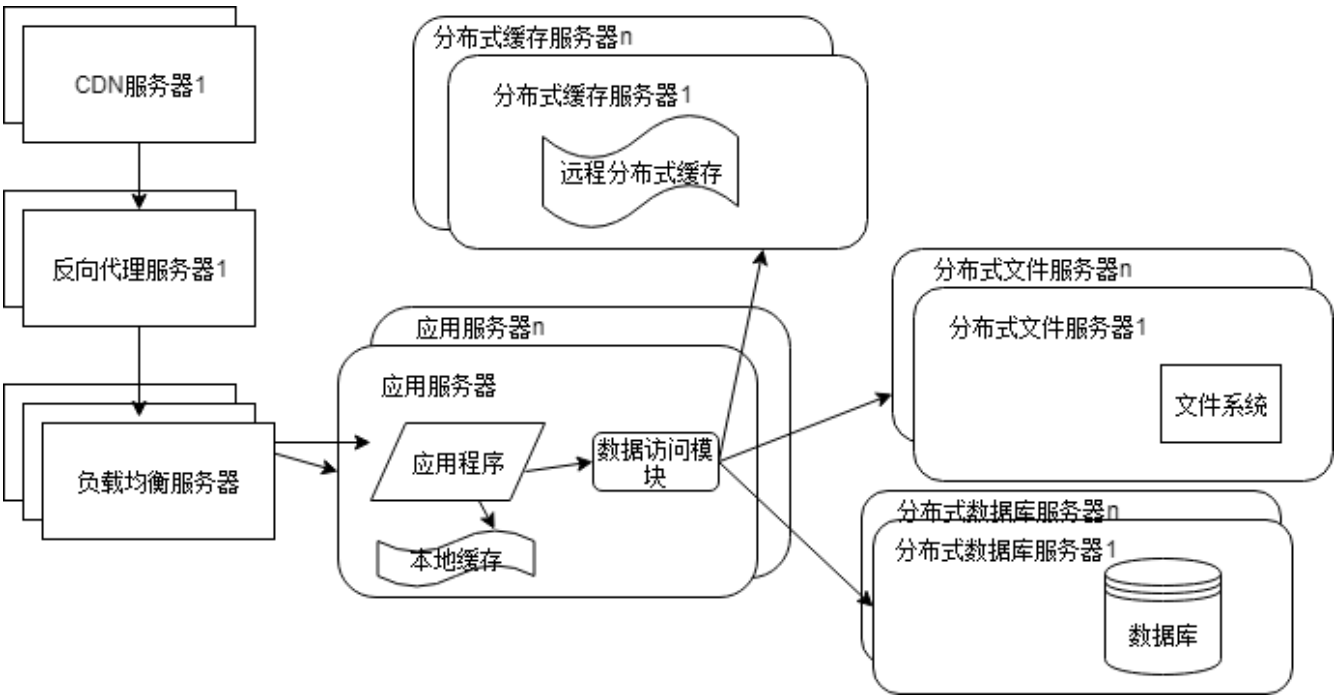
在使用CDN和反向代理之后，网站的架构演化成如下图所示：



1.2.7使用分布式文件系统和分布式数据库

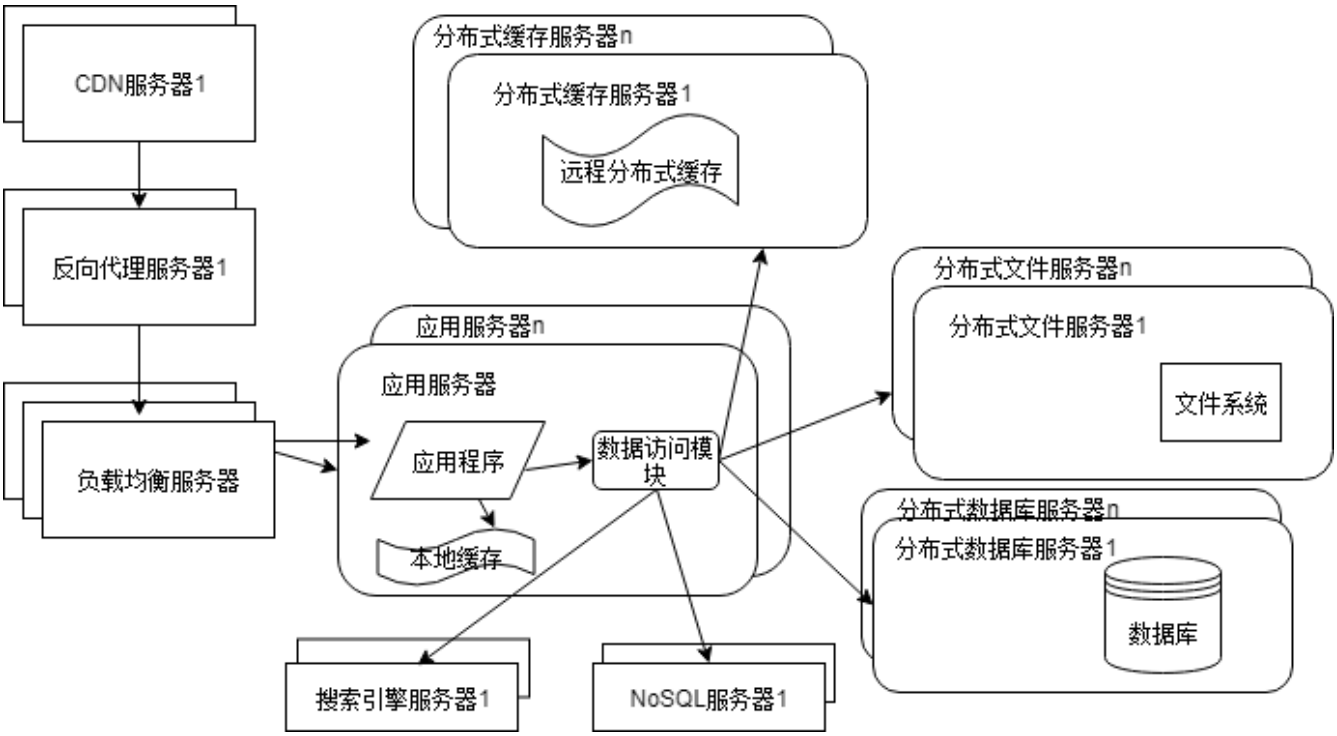
- 分布式数据库：分布式数据库是网站数据库拆分的最后手段，**只有在单表数据规模十分庞大时才使用**，一般情况下，网站更常用的是数据库拆分手段是**业务分库**，将不同业务的数据库部署在不同的物理服务器上
- 分布式文件系统：

系统的架构如下图所示：



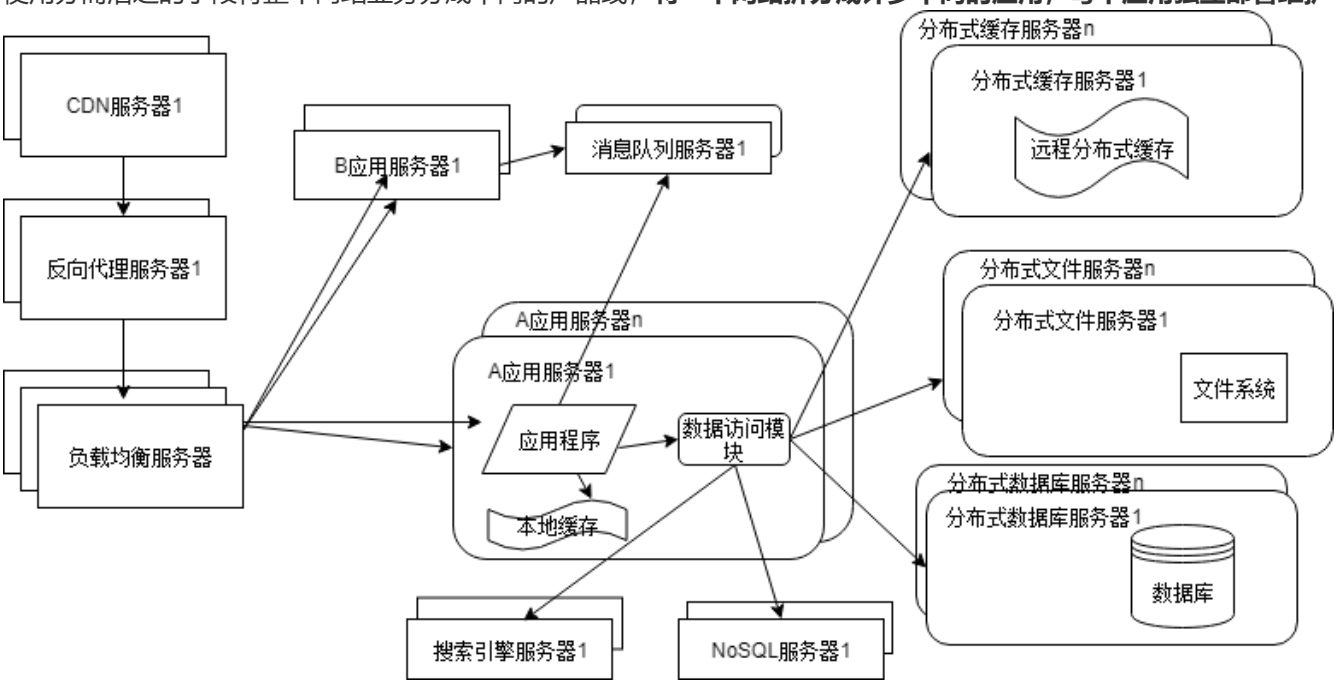
1.2.8使用NoSQL和搜索引擎

由于网站的业务对于数据存储和检索的需求越来越高，所以网站需要采用**非关系型数据库（如NoSQL）和非数据库查询技术（如搜索引擎）**



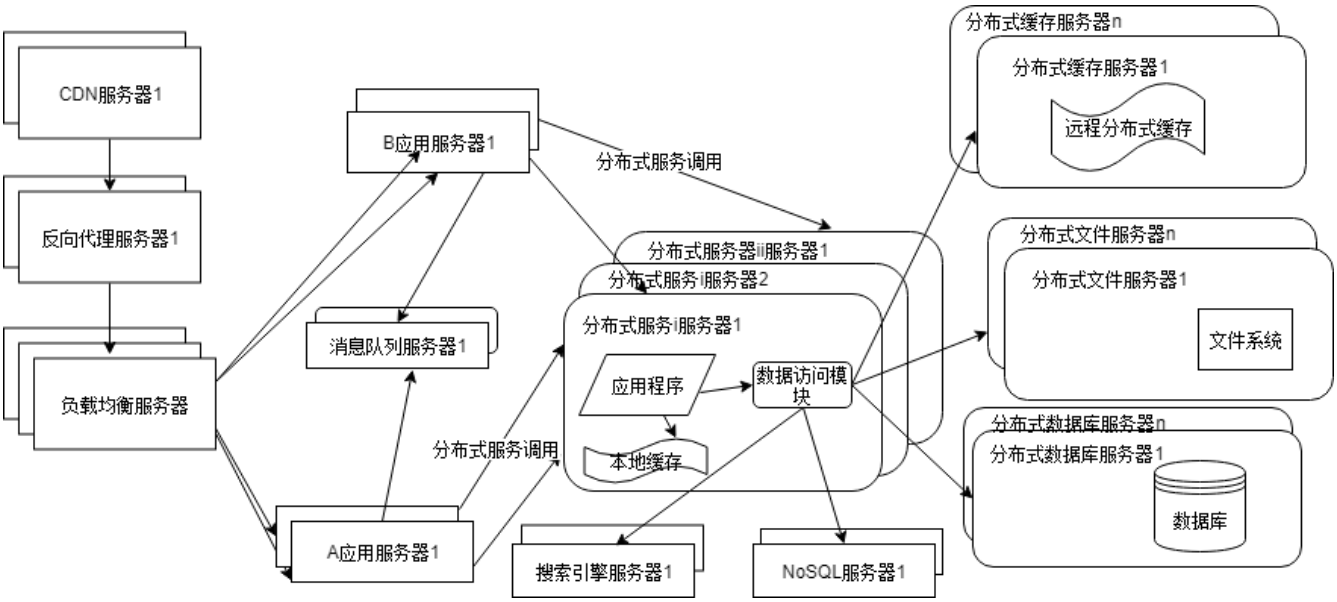
1.2.9业务拆分

使用分而治之的手段将整个网站业务分成不同的产品线，**将一个网站拆分成许多不同的应用，每个应用独立部署维护**



1.2.10分布式服务

每一个应用系统中会执行许多相同的业务操作，比如用户管理、商品管理等，**那么可以将这些应用的公共业务提取出来，独立部署**，这样只需要通过分布式服务调用公共的业务服务来完成具体的业务操作。



1.3大型网站架构演化的价值观

大型网站架构技术的核心价值不在于从无到有的搭建一个大型网站，而是能从小型网站慢慢演化为一个大型网站

1.4网站架构设计误区

- 一味追随大公司的解决方案
- 为了技术而技术
- 企图用技术解决所有问题

2.大型网站架构模式

- 模式：是指描述某个问题以及该问题的解决方案

2.1网站的架构模式

大型网站为了解决高并发、海量数据处理、高可靠运行等问题，提出了许多解决方案，实现网站高性能、高可用、可伸缩、可扩展、安全的特性

2.1.1分层

- 分层是将系统从**横向维度上**切分成几个部分，每个部分负责相对比较单一的职责，然后通过上层对下层的调用和依赖组成一个完整的系统。 在大型网站中，将系统分成：**应用层、服务层、数据层。** 各层之间具有一定的独立性，只要维持调用接口不变，各层可以根据具体问题内部进行演化。 禁止跨层次调用（应用层调用数据层）、逆向调用（数据层调用服务层）
- 分层架构是逻辑上的，物理部署时，三层架构既可以部署在同一台机器上，也可以部署在不同服务器上。

2.1.2分割

- 分割是从**纵向**对软件系统进行切分，将这些不同功能和服务分割开来，得到**高内聚低耦合的模块单元**；
- 方便软件的开发和维护，也方便不同模块的分布式部署，提高网站的并发处理能力和扩展能力；

2.1.3分布式

- 经过分层和分割之后，可以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**，这就是分布式部署的好处；
- 而且大量的服务器表示有大量的硬件资源，能处理并发访问数和数据量就更大；

- 但是分布式调用要通过网络，会影响性能；
- 服务器越多，宕机的机率越高，可靠性越差；
- 数据在分布式环境中保持一致性困难；
- 网站的依赖错综复杂，开发维护困难；

常用的分布式方案有：

- **分布式应用和服务**： 将分层和分割的应用和服务模块分布式部署。
- **分布式静态资源**： 将网站的静态资源如：JS、CSS、图片等资源独立部署，采用独立的域名。**独立部署能减轻应用服务器的负载压力，独立域名能利用浏览器的多线程加快响应速度。**
- 分布式数据和存储
- 分布式计算

2.1.4集群

- **集群化**：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；
- **可扩展性**:可以无限制的像集群中添加服务器；
- **高可用性**：当某台服务器发生故障时，负载均衡设备会将请求转发到其他服务器上

2.1.5缓存

缓存就是将数据存放在距离计算最近的位置以加快处理速度，缓存是改善软件性能的第一手段； 缓存使用的两个条件： **数据访问热点不均匀，即符合二八定律 数据在某个时间段都有效，不会很快过期**

常用的缓存有：

- CDN：内容分发网络，部署在距离用户最近的网络提供商的机房中，**缓存网站的静态资源**；
- 反向代理：部署在网站中心机房中，**用户的请求会第一个到达反向代理服务器，缓存网站的静态资源**
- 本地缓存：**在应用服务器中缓存本地的热点数据**；
- 分布式缓存：**将数据缓存在一个专门的分布式缓存集群中，应用程序通过网络访问缓存数据**；

2.1.6异步

异步：业务之间的消息传递不是同步调用，而是**将一个业务操作分层多个阶段，每个阶段通过共享数据的方式异步执行进行协作**；

同一个服务器内，可以通过**多线程共享内存队列的方式实现异步**；分布式系统中，**多个服务器集群通过分布式消息队列实现异步**；

异步典型架构就是生产者消费者模式，二者不存在直接调用，通过消息队列进行通信异步消息队列的特性：

- 提高系统可用性：消费者服务器发生故障时，数据会在消息队列中堆积，生产者服务器可以继续处理业务不用阻塞，系统整体表现无故障；
- 加快网站响应速度：生产者服务器处理完业务请求后，将数据写入消息队列，可以直接返回响应延迟减少；
- 消除并发高峰：消息队列会将突然增加的访问请求数据放入消息队列中，等待消费者服务器依次处理，不会对整个网站造成太大压力；

2.1.7冗余

冗余是保障网站7×24小时连续运行的重要手段，实现服务的高可用

- 服务器冗余：由多台服务器构成集群，保证在有服务器宕机的情况下，网站仍是可用的；
- 数据冗余：将数据存储在多个地方，保证数据能够获取；

2.1.8自动化

- 目前大型网站的自动化架构设计主要集中在运维发布方面

2.1.9安全

- 通过密码和手机验证码进行身份认证；
- 登录、交易等操作需要对网络通信进行加密；
- 使用验证码防止机器人程序滥用网络资源攻击网站

3.大型网站核心架构要素

软件架构的要素：

- 性能
- 可用性
- 伸缩性
- 扩展性
- 安全性

3.1性能

性能是网站的一个重要指标，是网站结构设计的重要方面；

- 在浏览器端：浏览器缓存、使用页面压缩、合理布局页面、减少Cookies传输；
- 在应用服务器端：使用本地缓存和分布式缓存，通过异步操作将用户请求发送到消息队列，多台应用服务器构成集群，使用多线程、改善内存管理；
- 在数据库服务器端：索引、缓存、SQL优化；
- 衡量网站性能指标：响应时间、TPS、系统性能计数器

性能符合预期是应该的，还要考虑系统在高并发访问的情况下，超出负载设计能力的情况下可能会出现的性能问题。

3.2可用性

可用性是指：在7×24可用内，扣除故障时间，网站的总可用时间对应的可用性指标； 大型网站拥有大量的服务器，每天必定会有一些服务器宕机，所有网站可用性设计的前提就是**必然有服务器会宕机，高可用网站就是指，即使在服务器宕机时，服务或者应用任然可用。 实现网站的高可用的主要手段就是冗余，应用服务部署多台服务器，数据存储在多台服务器上备份**：

- 应用服务器：通过**负载均衡**设备，将多台应用服务器构成一个**集群**对外提供服务。当有服务器宕机时，将该服务器上的请求转发到其他服务器上从而实现**高可用**，这样做的前提条件是，**应用服务器必须是无状态的，即应用服务器上不能保存当前会话的信息。**
- 存储服务器：对服务器上的数据进行**实时备份**，当服务器宕机时，需要将数据**转移**到可用的服务器上，并进行**数据恢复**保证有服务器能提供数据存储。

- 软件开发过程：通过预发布验证、自动化测试、自动化发布、灰度发布等手段，减少故障引入线上环境的可能，避免故障范围扩大。

3.3伸缩性

网站是通过集群的方式将多台服务器变成一个整体对外提供服务。伸缩性就是指：通过向集群中添加或减少服务器，能使得网站的性能线性变化。 衡量伸缩性的主要标准是：是否使用多台服务器构成集群；是否容易向集群中添加新的服务器；加入新的服务器是否能提供和原来服务器无差别的服务；集群中可容纳的服务器总数是否有限制。

- 应用服务器集群：**只要保证服务器上不保存数据，每个服务器都是对等的，通过负载均衡设备就能向集群中不断加入服务器**；
- 缓存服务器集群：**加入新的缓存服务器会导致缓存路由失效，使得集群中的大部分缓存数据都无法访问，需要改进缓存路由算法保证缓存数据的可访问性**；
- 关系数据库：很难做到大规模集群的可伸缩性，**通过路由分区等手段将部署有多个数据库服务器组成一个集群**；
- 非关系数据库：NoSQL数据库天生就是为了处理海量数据，因此其伸缩性支持非常好；

3.4扩展性

网站的扩展性关注的是**网站的功能需求**，即如何设计网站架构使得其能快速响应需求的变更； 衡量网站架构的扩展性的主要标准是：**网站添加新功能时，是否可以实现对现有产品透明无影响；不需要改动或很少改动既有业务功能就能上线新产品。 网站的扩展性主要手段就是：事件驱动架构和分布式服务**

- 事件驱动架构：**通常使用消息队列实现**，将用户请求或其他业务事件构成的消息发布到消息队列，消息的处理者作为消费者从消息队列中获取消息进行处理。**通过消息队列将消息产生和消息处理分离开，可以透明的添加新的消息生产者或新的消息消费者**；
- 分布式服务：**通过分布式服务调用，将业务和可复用服务分离开。**新增的功能通过调用可复用的服务实现自身的逻辑功能，对现有功能不会造成任何影响。

3.5安全性

网站的安全架构就是保护网站不受恶意访问和攻击，保护网站的重要数据不被窃取； 衡量网站安全架构的标准是：针对现存和潜在的各种攻击与窃密手段，是否有可靠的应对策略；

二、架构

- **性能**
- **可用性**
- **伸缩性**
- **扩展性**
- **安全 是网站架构最核心的要素，这几个问题解决了，大型网站架构设计的大部分挑战也都克服了。**

4.网站的高性能架构

网站性能既可以客观的表示，如：响应时间、吞吐量等；也可以是主观感受，如：用户感受和工程师感受。

4.1网站性能测试

性能测试是性能优化的基础和前提，也是性能优化结果的检查和度量标准

4.1.1不同视角下的网站性能

4.1.1.1用户视角的网站性能

从用户的角度来说，网站性能就是直观上感受的**网站响应速度的快慢。**具体来说，就是: 用户感受的时间 = 用户计算机和网站服务器通信的时间 + 网站服务器处理时间 + 用户浏览器解析响应数据的时间

在实践中，使用前端架构优化手段：

- 优化页面HTML样式
- 利用浏览器的并发和异步特性
- 调整浏览器缓存策略
- 使用CDN服务
- 使用反向代理

4.1.1.2开发人员视角的网站性能

开发人员主要关注的是**应用系统本身及其子系统的性能**，包括：

- *响应延迟*
- *系统吞吐量*
- *并发处理能力*
- *系统稳定性*

使用的**主要优化手段**有：

- *使用缓存加速数据读取*
- *使用集群提高系统吞吐能力*
- *使用异步消息加快请求响应及实现削峰*
- *使用代码优化改善程序性能*

4.1.1.3运维人员视角的网站性能

运维人员主要关注**网站的基础设施性能和资源利用率**：

- *网络运营商的宽带能力*
- *服务器硬件配置*
- *数据中心网络架构*
- *服务器和网络宽带的资源利用率*

主要的优化手段是：

- *建设优化骨干网*
- *使用高性价比定制服务器*
- *利用虚拟化技术优化资源利用率*

4.1.2性能测试指标

从开发和测试人员的视角，网站的性能指标主要有：响应时间、并发数、吞吐量、性能计数器。

4.1.2.1响应时间

响应时间是指**从发出请求开始到收到最后响应数据所需要的时间**，响应时间是系统最重要的性能指标，直观的反应了系统的快慢。

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的响应时间。

4.1.2.2并发数

并发数是指**系统能够同时处理请求的数目，对于网站就是同时提交请求的用户数。**

- *网站并发用户数：同时向网站提交请求的用户数；*
- *网站在线用户数：当前登录网站的用户数；*
- *网站系统用户数：可能访问系统的总用户数，也就是注册用户数；*

网站系统用户数 >> 网站在线用户数 >> 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。

4.1.2.3吞吐量

吞吐量是指**单位时间内系统处理的请求数量**，对于网站可以使用：**请求数/秒，页面数/秒，访问人数/天，处理的业务数/小时**等指标来衡量。 *TPS（每秒事务数）*是吞吐量的一个常用量化指标，此外还有*HPS（每秒HTTP请求数）*、*QPS（每秒查询数）*等。

在系统的并发数由小增大的过程中，系统的吞吐量先是逐渐增加，到达一个极限之后，随着并发数的增加，系统吞吐量下降，到达系统崩溃点后，系统资源耗尽，系统吞吐量为零。

而系统的响应时间先是随并发数小幅上升，当到达吞吐量极限之后，响应时间快速上升，到达系统崩溃点时，响应时间无穷大。

4.1.2.4性能计数器

性能计数器是指**描述服务器或操作系统的一些数据指标，包括System Load、对象和线程数、内存使用、CPU使用、磁盘与网络I/O等指标。**

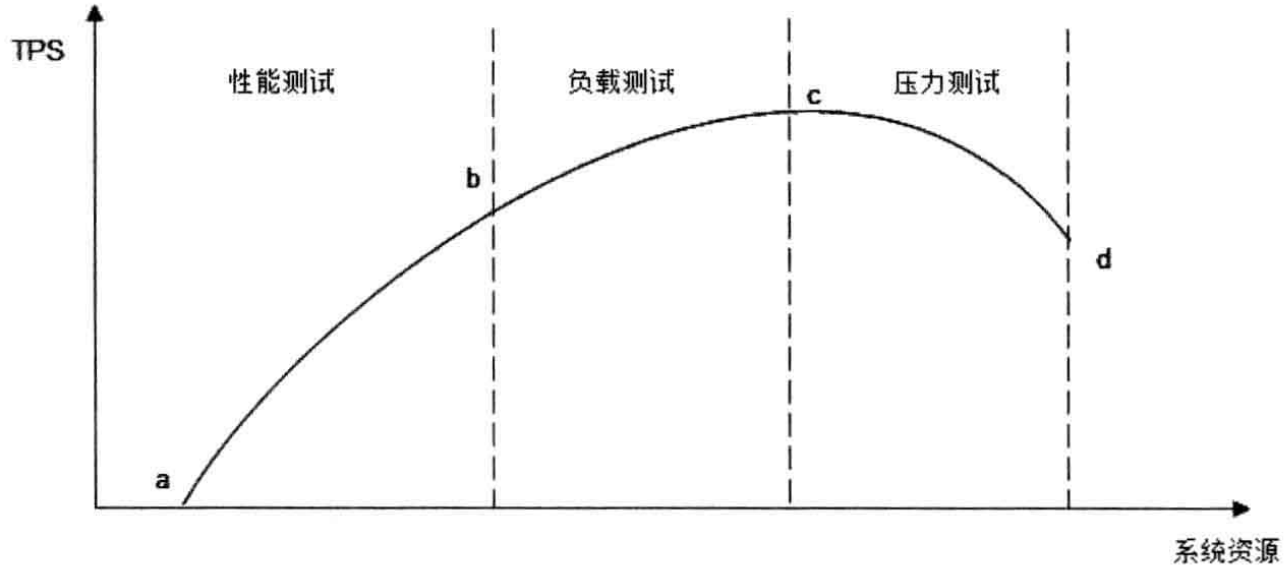
System Load即系统负载，指**当前正在被CPU执行和等待CPU执行的进程数的总和，反应了系统忙闲程度。** 在多核CPU的情况下，理想情况是所有CPU都在使用，没有等待的进程，即Load的值为CPU的数目； Load值 > CPU数 ---------> 表示有CPU空闲，资源存在浪费； Load值 < CPU数 --------> 表示有进程排队等待CPU调度，系统资源不足；

4.1.3性能测试方法

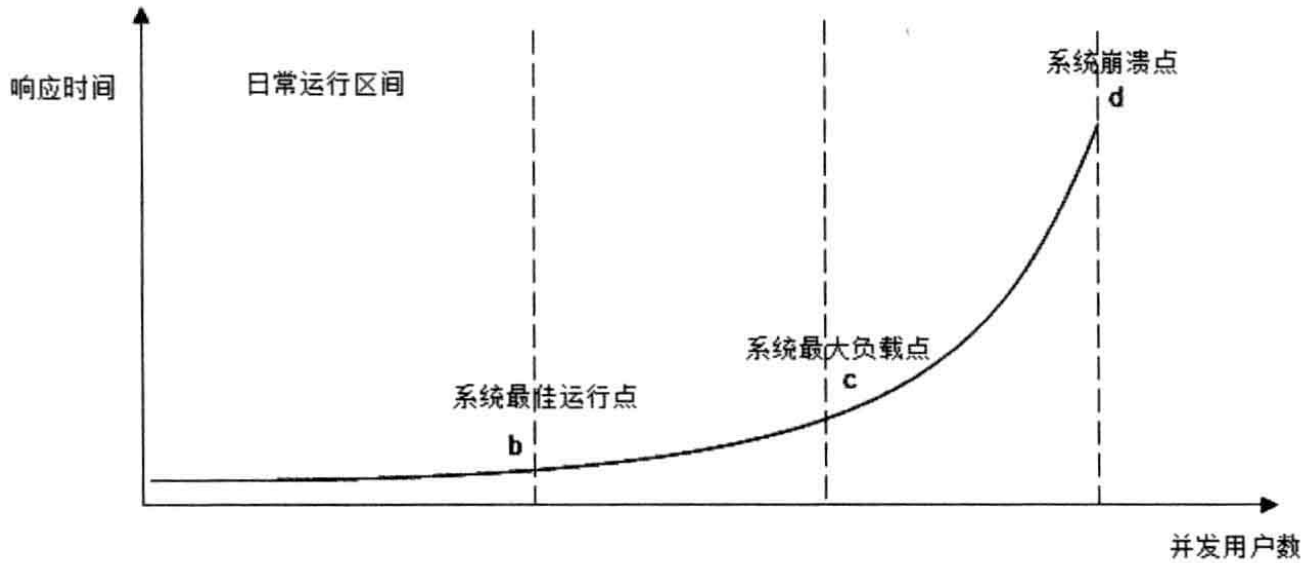
性能测试就是通过不断对系统增加访问压力，以获得系统性能指标、最大负载能力、最大压力承受能力的过程。

- **性能测试**：测试性能指标，系统性能是否符合需求及设计目标；
- **负载测试**：测试最大负载能力，保证系统正常运行能承受的最大访问负载压力；
- **压力测试**：测试最大承受能力，可能导致系统崩溃的最大访问负载压力；
- **稳定性测试**：测试不同硬件、软件、网络环境下，系统的稳定性；

下图是性能测试曲线：



下图是响应时间曲线：



4.1.4性能测试报告

以报告的形式呈现性能测试曲线和响应时间曲线的数据

4.1.5性能优化策略

如果性能测试结果不满足设计或业务需求，则**寻找系统瓶颈，分而治之，逐步优化。**

- 性能分析：若存在性能问题，则要对请求经历的各个环节进行分析，找出可能出现的性能瓶颈的位置，定位问题；
- 性能优化：根据网站分层架构，可分为**Web前端性能优化、应有服务器性能优化、存储服务器性能优化。**

4.2Web前端性能优化

Web前端是指网站业务逻辑之前的部分，包括浏览器加载、网站试图模型、图片服务、CDN服务等，主要的优化手段有：优化浏览器访问、使用反向代理、CDN等

4.2.1浏览器访问优化

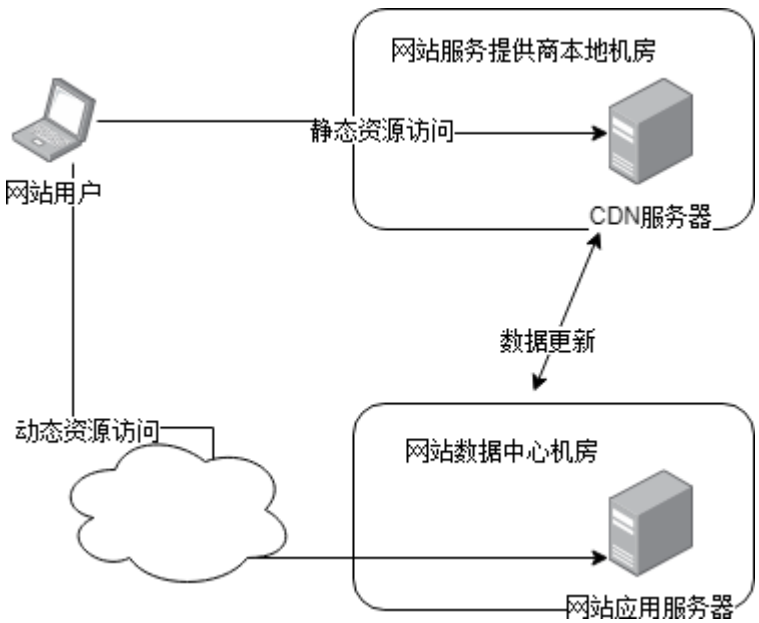
- **减少HTTP请求**： 由于HTTP协议是无状态的，即每次请求服务器都会用线程进行处理，减少HTTP请求能有效提高访问性能。 *具体的方法有*：**合并CSS、合并JavaScript、合并图片。**即将请求需要的CSS、JavaScript合并成一个文件，多张图片合并成一张。
- **使用浏览器缓存**： 由于网站的**静态资源（CSS、JavaScript、Logo、图片等）更新的频率较低**，但是静态资源又是每次请求都需要的，如果**将静态资源缓存在浏览器中**，可以极好的改善性能。 通过设置HTTP响应头的Cache-Control和Expires属性，可设置缓存有效时间。 使用浏览器缓存时，在更新静态资源时需要逐量更新，而且有一定的时间间隔，这样是为了防止浏览器缓存大量失效，使得服务器负载骤增。
- **启用压缩 在服务器端对文件进行压缩，在浏览器端解压文件**，可有效减少通信传输的数据量。HTML、CSS、JavaScript文件启用GZip压缩可达到较好的效果。 但是压缩对服务器和浏览器都会产生一定的压力，服务器资源不足的情况下不考虑。
- **CSS放在页面最上面，JavaScript放在页面最下面 浏览器会下载完CSS文件之后对页面进行渲染**，所以最好是将CSS放在页面最上面。 **JavaScript会在加载后立即执行，可能会阻塞整个页面，**所以最好将JavaScript放在页面最下面。
- **减少Cookie传输 Cookie包含在每次的请求和响应中，太大Cookie影响数据传输。 对于静态资源发送Cookie没有意义，可以考虑静态资源使用独立域名访问，减少请求静态资源时发送Cookie。**

4.2.2CDN加速

CDN(Content Distribute Network，内容分发网络)本质上就是**缓存**，且CDN的位置在用户网络运营商机房，是离用户最近的地方，即网络的第一跳。

CDN中存储**静态资源，例如：图片、文件、CSS等**

具体结构如下图所示：



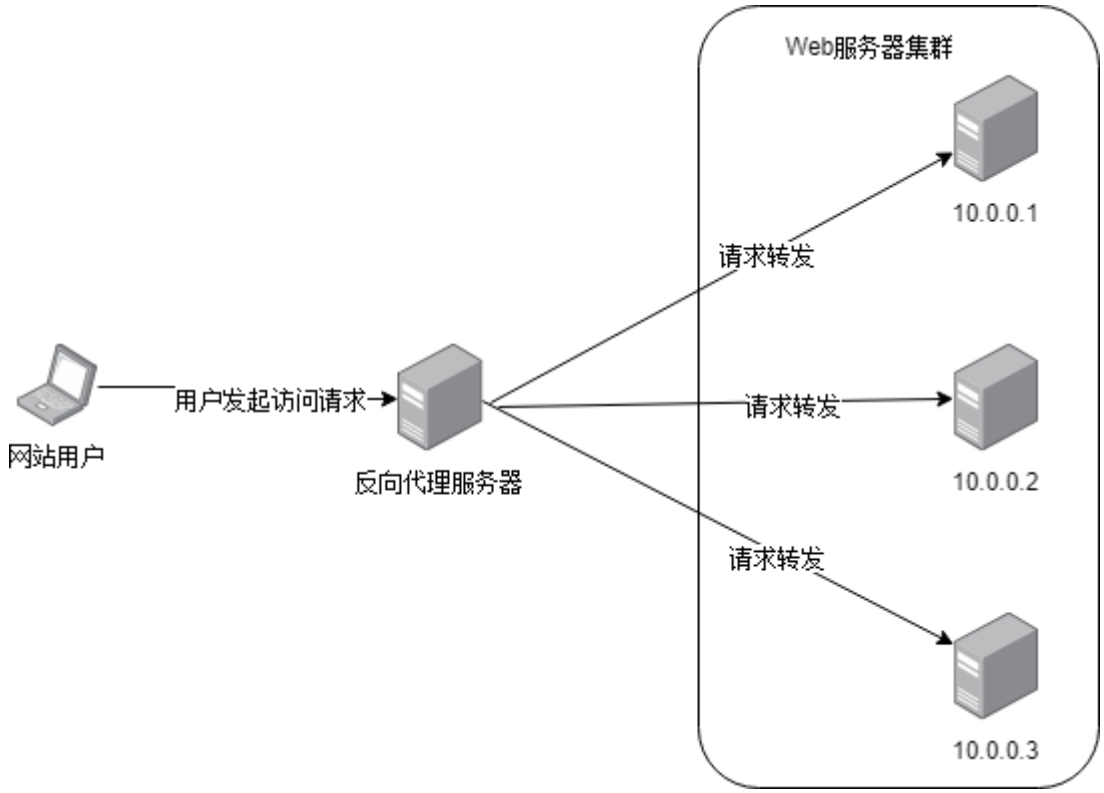
4.2.3反向代理

传统代理服务器位于浏览器一侧，代理服务器将浏览器的HTTP请求发送到互联网上；而反向代理服务器则是位于网站机房一侧，**代理网站Web服务器接收HTTP请求。**

反向代理服务器有以下作用：

- **保护网站安全**：所有请求都必须经过反向代理服务器，相等于在Web服务器和网络攻击之间建立了屏障；
- **配置缓存加速网站响应**：当用户第一次访问静态内容收到响应时，反向代理服务器会将静态内容缓存起来，有其他用户访问的时候能够直接返回，加速Web请求响应；
- **负载均衡**：反向代理服务器能实现请求的转发，这和负载均衡服务器的作用一致；

使用反向代理服务器的网站架构如下图所示：



4.3应用服务器性能优化

应用服务器的主要优化手段有：**缓存、集群、异步**

4.3.1分布式缓存

缓存是在系统性能出现瓶颈时的第一解决方案。

4.3.1.1缓存的基本原理

缓存是指，将数据存储在访问速度相对较高的存储介质中，以加快系统的处理，缓存的本质就是一个Hash表。

在网站应用中，数据缓存以<Key, Value>的形式存储在Hash表中。

缓存主要是存放那些**读写比很高、很少变化的数据**，应用在**读取数据时，先到缓存中读，如果缓存中不存在或者数据失效，再访问数据库，重新写入缓存。**

由于网站的数据访问符合二八定律，即80%的访问落在20%的数据上，所以利用缓存读数据快速的特性能很好的改善系统性能。

4.3.1.2合理使用缓存

缓存虽然优点很多，但是不合理使用缓存不仅不会提升系统性能，而且还会成为系统累赘。

- **频繁修改的数据**

  频繁修改的数据存入缓存的意义不大，**会出现应用还来不及取数据，数据就失效的场景**，一般来说，读写比在2：1以上的数据存入缓存才有意义。

- **没有热点的访问**

  应用系统访问数据没有热点，几乎是平均访问的话，也不适合使用缓存，此时缓存就没有意义，因为大部分缓存中的数据还没来得及再次访问就被挤出缓存了。

- **数据不一致与脏读**

  一般会对缓存中的数据设置**失效时间**，超过失效时间的数据要重新从数据库中加载，所以应用要容忍一定时间的数据不一致。

- **缓存可用性**

  缓存是**提高系统的读取性能**的，理论上缓存数据丢失或者缓存不可用不会影响到应用程序的处理，毕竟应用程序可以从数据库中获取数据。

  但是实际中，由于*缓存承担了大部分的数据访问压力*，**当缓存崩溃时，数据库会因为无法承受大量的数据访问而宕机，进而导致整个网站不可访问，这就是缓存雪崩。**

  解决缓存雪崩的方法主要是：*通过分布式缓存服务器集群，将缓存数据分布到集群的多个服务器上，可在一定程度上改善缓存的可用性*。即使有缓存服务器宕机，只会影响小部分的缓存数据，大部分数据仍是可访问的。

- **缓存预热**

  缓存中存放的是热点数据，而热点数据又是缓存系统利用LRU算法经过*很长时间*筛选出来的。新启动的缓存服务器如果没有任何数据，在重建缓存的过程中还是会对系统性能造成负荷。

  可以使用**缓存预热**来解决以上问题，即**在缓存系统启动时，就把热点数据加载好。**

- **缓存穿透**

  缓存穿透是指：**持续高并发的请求某个不存在的数据，由于缓存没有该数据，导致每次请求都是访问数据库会使得数据库压力过大而崩溃。**

  一个简单的解决办法就是：将不存在的数据也缓存起来，其值设置为null。

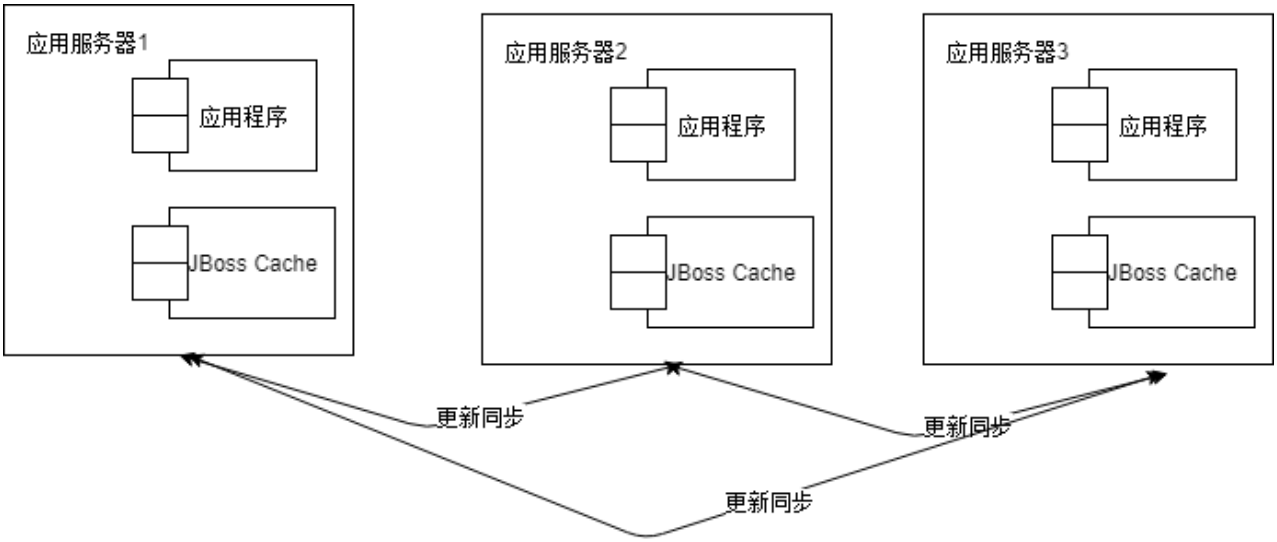
4.3.1.3分布式缓存架构

分布式缓存架构是指：缓存部署在多个服务器组成的集群中，以集群整体的方式向外提供缓存服务。有两种架构方式：

- 需要同步更新的分布式缓存：JBoss Cache

  在集群中的所有服务器都保存相同的数据，当某台服务器上的缓存数据更新时，要通知其他所有服务器同步进行更新。

  JBoss Cache通常将应用程序和缓存部署在同一台服务器上，应用程序可以快速的获取缓存数据，但是缓存的数据量受限于单一服务器的内存空间，而且当集群规模很大的时候，**缓存更新数据的同步过程代价过大。**



- 不互相通信的分布式缓存：Memcached

  Memcached采用集中式的缓存集群管理，主要有以下特点：

  - 简单的通信协议

    远程通信需要考虑两个要素：一是通信协议：选择TCP、UDP、HTTP等中某个协议；另一个是通信序列化协议：数据传输的两端要使用彼此可识别的数据序列化方式。

    Memcached使用TCP（UDP）协议通信；其序列化协议是基于文本的自定义协议，以一个关键字开头，后面一组命令操作数。

  - 丰富的客户端程序

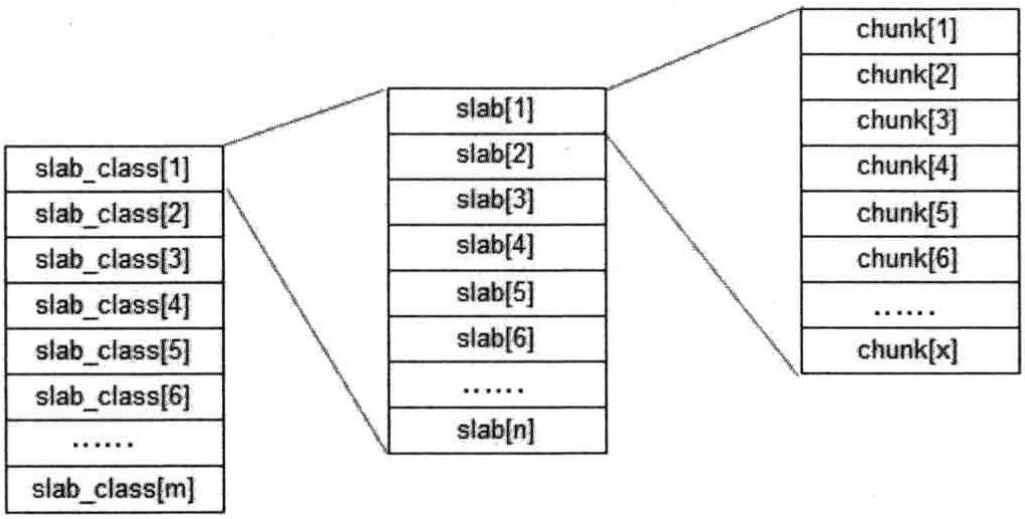
    由于Memcached通信协议很简单，所以只要支持该协议的客户端都能够和Memcached服务器通信。

  - 高性能的网络通信

    Memcached服务端通信模块基于Libevent，一个支持事件触发的网络通信程序库。

  - 高效的内存管理

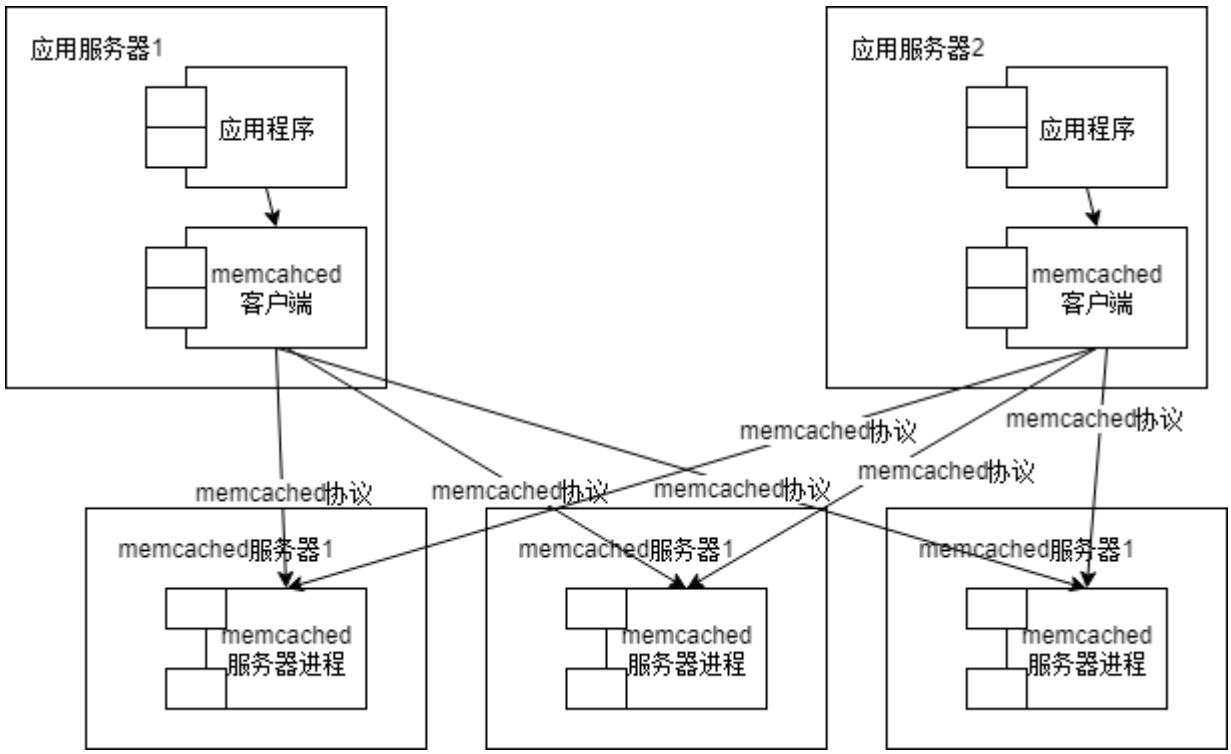
    Memcached使用**固定空间分配**来解决内存碎片问题：将内存空间分成一组slab，每组slab里包含一组chunk，同一个slab里的chunk大小相等且固定，拥有相同大小chunk的slab组织在一起，被称为slab_class。**存储数据时根据数据的size大小，寻找一个大于size的最小chunk将数据写入，内存的分配和释放都是以chunk为单位的。**



- 互不通信的服务器集群架构

  客户端路由算法一致性Hash算法能保证集群内服务器宕机时，大部分数据都是有效的，不会造成缓存雪崩。集群内服务器互不通信使得集群可以做到几乎无限制的线性伸缩。

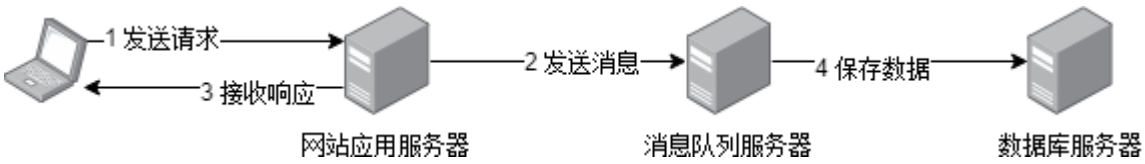
Memcached的架构如下图所示：



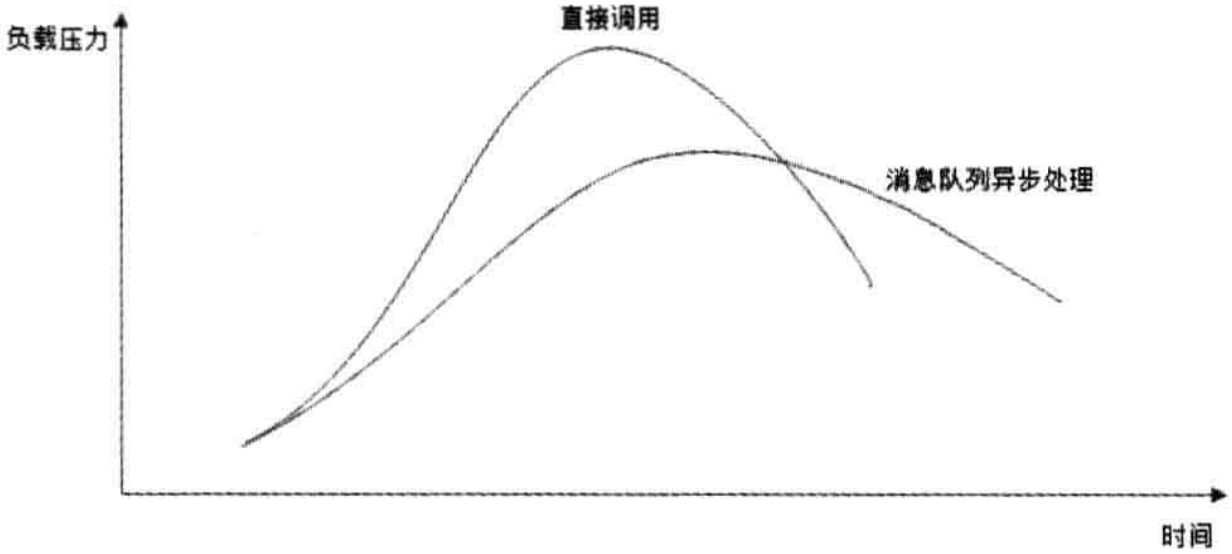
4.3.2异步操作

使用消息队列将调用异步化，可改善网站的扩展性，也可以改善网站的性能。

在不使用消息队列的情况下，用户请求数据直接写入数据库，在高并发的情况下会对数据库造成巨大压力，使得延迟加剧。**使用消息队列后，用户请求的数据发送给消息队列后立即返回，再由消息队列的消费者进程从消息队列中获取数据，异步写入数据库。**

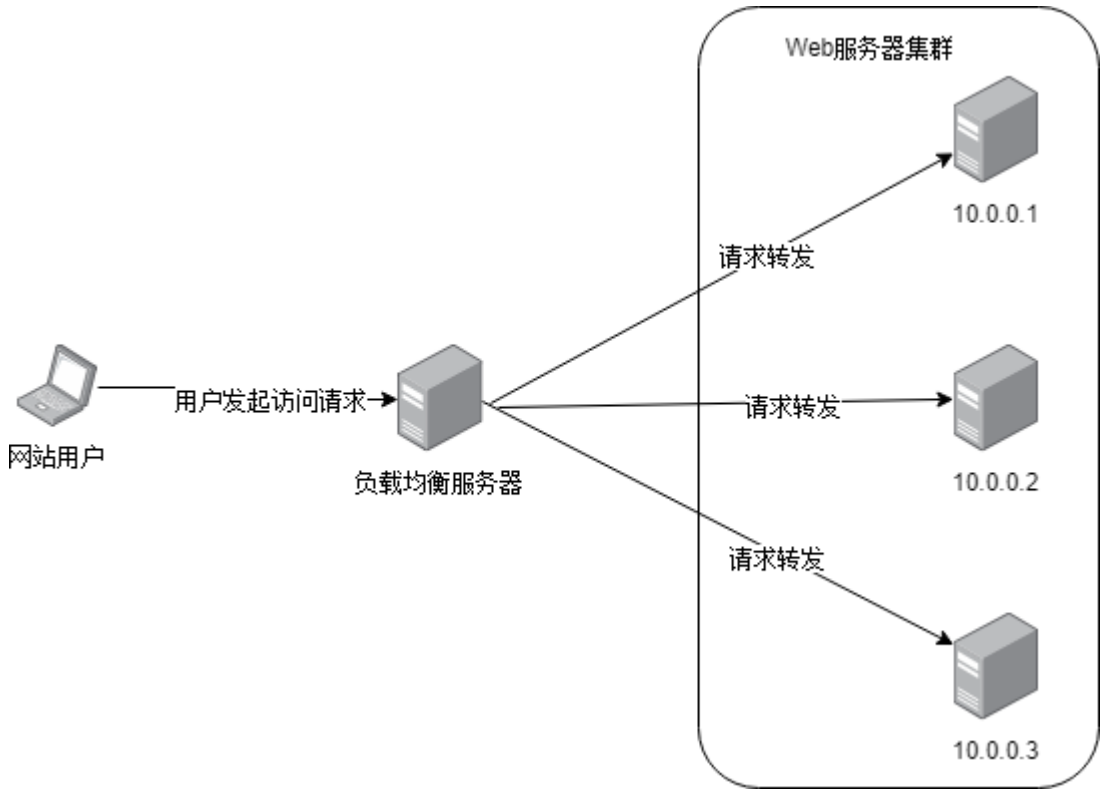


消息队列具有很好的削峰作用，通过异步处理将短时间高并发的事务消息存储在消息队列中，从而削平高峰期并发事务。由于数据写入消息队列后立即返回给用户，数据在后续的业务校验、写入数据库等操作可能失败，因此在使用消息队列进行异步处理后，需要适当修改业务流程进行配合。



4.3.3使用集群

负载均衡技术是指：**为一个应用构建一个由多台服务器组成的服务器集群，将并发访问请求分发到多台服务器上处理，避免单一服务器因负载压力过大而响应缓慢，使用户请求具有更好的响应延迟特性。**



4.3.4代码优化

合理优化业务代码，能够很好的改善网站性能。

4.3.4.1多线程

线程比进程更轻量，更少占用系统资源，切换代价小，目前主要的Web应用服务器都采用多线程的方式响应并发用户请求。

但是多线程会带来线程安全的问题：多线程同时对某个资源进行修改会造成数据不一致的情况。

解决多线程安全的方法：

- **对象设计为无状态对象**：

  无状态对象是指，对象本身不存储状态信息，这样多线程访问的时候就不会出现状态不一致的情况。

  Java Web中的Servlet对象就是无状态的，而且Web开发中常用的贫血模型对象都是无状态对象。
- **使用局部对象**：

  在方法内部创建对象，这些对象会被每个进入方法的线程创建，不会出现多线程访问的问题。
- **并发访问资源时使用锁**：

  多线程访问的时候，通过锁的方式使得多线程并发操作转化为顺序操作，从而避免资源并发修改时状态不确定。

4.3.4.2资源复用

系统运行时，要尽量避免开销很大的对象的创建和销毁，比如数据库连接、网络通信连接、线程、复杂对象等。

资源复用的方式有两种：

- **单例（Singleton）**：只对对象创建一次，之后使用的都是该对象实例。
- **对象池（Object Pool）**：通过复用对象实例，减少对象创建和资源消耗。数据库连接池、线程池都是使用对象池的方式。

4.3.4.3数据结构

在不同场景下使用合适的数据结构，灵活组合各种数据结构改善数据读写和计算性能可极大优化程序的性能。

4.3.4.4垃圾回收

如果Web应用运行在JVM等具有垃圾回收的环境中，那么垃圾回收可能会对系统的性能特性产生巨大的影响。理解垃圾回收机制有助于优化程序和参数调优，以及编写内存安全的代码。

4.4存储性能优化

在网站应用中，海量数据的读写对磁盘访问造成了巨大的压力，虽然可以通过cache来缓解大部分的压力，但是在很多情况下，磁盘仍然是系统访问的最大瓶颈。

4.4.1机械硬盘VS固态硬盘

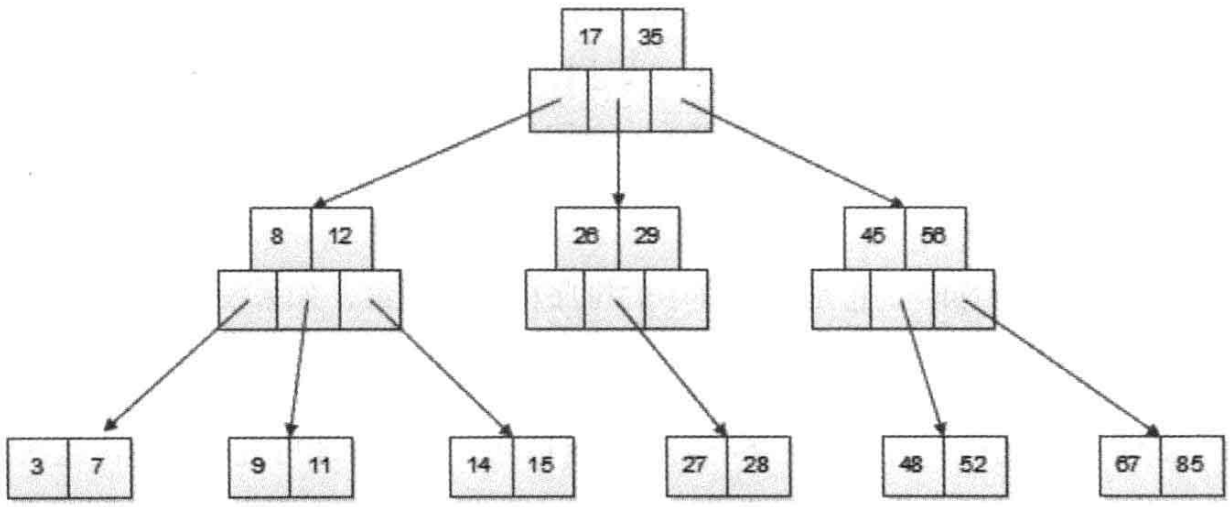
- 机械硬盘：连续访问性能较好，随机访问性能较差；
- 固态硬盘：支持快速随机访问；

4.4.2B+树VSLSM树

由于**磁盘的快顺序读写、慢随机读写的特性**，所以要使用合适的数据结构使得这种特性能加快系统性能。

- **B+树**：

  为了改善数据访问速度，数据库系统或文件系统会对数据先排序后存储。使用的数据结构就是B+树。

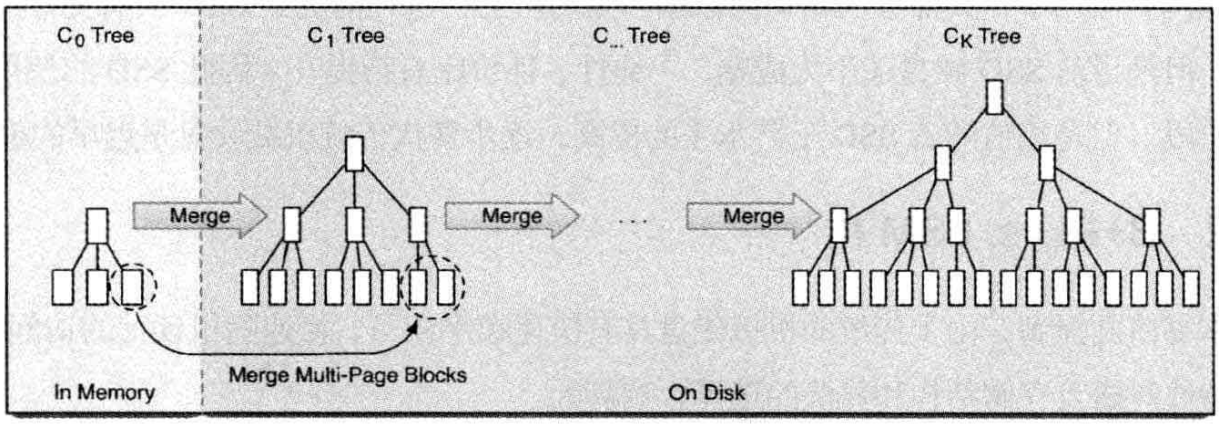


B+树是一种**专门针对磁盘存储而优化的数据结构**，以树节点为单位存储在磁盘中，从根节点开始查询所需要的数据。

- **LSM树**：

  LSM树可以看成一个N阶合并树，**数据的写操作**：都在内存中进行，而且都会创建一个新记录，这些数据在内存中仍是一颗排序树，当数据量超过设定的阈值以后，会将这颗排序树和磁盘上最新的排序树合并。当这颗排序树的数据量也超过设定阈值后，和磁盘的下一级的排序树合并。合并过程中，会用最新更新的数据覆盖旧的数据。

  数据的读操作：从内存中的排序树开始搜索，如果没用找到，就从磁盘上的排序树顺序查找。



4.4.3RAID VS HDFS

- RAID（廉价磁盘冗余阵列）：主要是为了改善磁盘访问延迟，增强磁盘的可用性和容错能力。通过使用RAID技术，实现数据在多块磁盘上的并发读写和数据备份。
- HDFS（Hadoop分布式文件系统）：系统在整个存储集群的多台服务器上进行数据并发读写和备份，可以看作在服务器集群规模上实现了类似RAID的功能。

5.网站的高可用架构

网站的可用性（Availability）是指网站可有效访问的特性

5.1网站可用性的度量和考核

一个网站要想展现在用户面前，需要经过很多环节，其中任意一个环节出现错误，都会导致网站页面不可用。

5.1.1网站可用性度量

网站不可用也称为网站故障，业界通常用多少个9来衡量网站的可用性，如QQ的可用性是4个9，即QQ服务99.99%可用。这意味着QQ服务要保证在所有运行时间中，只有0.01%的时间不可用，也就是一年只有大约53分钟不可用。

可用性的度量公式如下：

网站不可用时间（故障时间） = 故障修复时间点 - 故障发现（报告）时间点

网站年度可用性指标 = （1 - 网站不可用时间/年度总时间） × 100%

对于大多数网站：

- 2个9基本可用，年度不可用时间大概小于88个小时；
- 3个9是较高可用，年度不可用时间小于9小时；
- 4个9是具有自动恢复能力的高可用，网站年度不可用时间小于53分钟；
- 5个9是极高可用性，网站年度不可用时间小于5分钟。

5.1.2网站可用性考核

网站的可用性指标是网站架构设计的重要指标。可用性指标是网站或产品的整体考核指标。

5.2高可用的网站架构

由于互联网公司采用的都是PC级服务器、开源的数据库和操作系统，这些廉价的设备在保证节约成本的同时会降低了系统的可用性。

高可用的目的就是：在硬件故障时常态的情况下，保证服务器硬件故障时服务任然可用、数据依然被保存且能访问。

高可用架构的主要手段就是数据和冗余备份和失效转移：一旦某些服务器宕机，就将服务切换到其他可用服务器上；如果磁盘损坏，则从备份的磁盘读取数据。

通常把网站分成**应用层、服务层、数据层**基本层次架构，应用层主要负责具体业务逻辑处理，服务层主要负责提供可复用的服务，数据层负责数据的存储和访问。

- 应用层服务器通常是为了应对**高并发的访问请求**，采用**负载均衡设备**将一组服务器组成一个集群共同对外提供服务；负载均衡设备通过**心跳检测**等手段监控到某台服务器不可用时，将其从集群列表中删除，将请求转发到其他可用的服务器上，使得整个集群保证可用，实现**高可用**；
- 服务层服务器和应用层服务器情况类似，也是**通过集群的方式实现高可用**，服务层服务器会被应用层通过分布式服务调用框架调用，**分布式服务调用框架会在服务层实现负载均衡**，通过服务注册中心对提供服务的服务器进行**心跳检测**，发现服务不可用，则剔除不可用服务器，实现**高可用**；
- 数据层服务器存储着数据，为了保证服务器宕机时**数据不丢失且仍然可访问**，需要在数据写入时进行**数据同步复制**，将数据复制到多台服务器上，实现数据冗余备份；

而且服务器宕机不仅可以是硬件问题，还有由于网站升级发布引起的宕机，后者发生更加频繁。

5.3高可用应用

应用层保证高可用性的一种重要手段就是**应用的无状态性**，即应用服务器不保存业务上下文信息，仅根据每次请求的数据进行相关业务处理，多个服务实例（服务器）之间完全对等，**请求提交任意服务器处理结果都是一样的。**

但是并不是所有场景下都是无状态的，比如在交易类网站中，购物车记录就是上下文信息，即有状态的。

5.3.1通过负载均衡进行无状态服务的失效转移

不保存状态的应用给高可用的架构设计带来了巨大便利，所有的服务器完全对等，当集群中任意一台或多台服务器宕机，请求提交给集群中其他服务器，对于用户而言，请求总是成功的。

实现服务器可用状态实时监控、自动转移失败任务的机制就是**负载均衡。**

负载均衡本来是将数据和流量分摊到集群中的多台服务器上的，但由于集群中服务器是无状态、对等的，负载均衡也起到了高可用的作用。

5.3.2应用服务器集群的Session管理

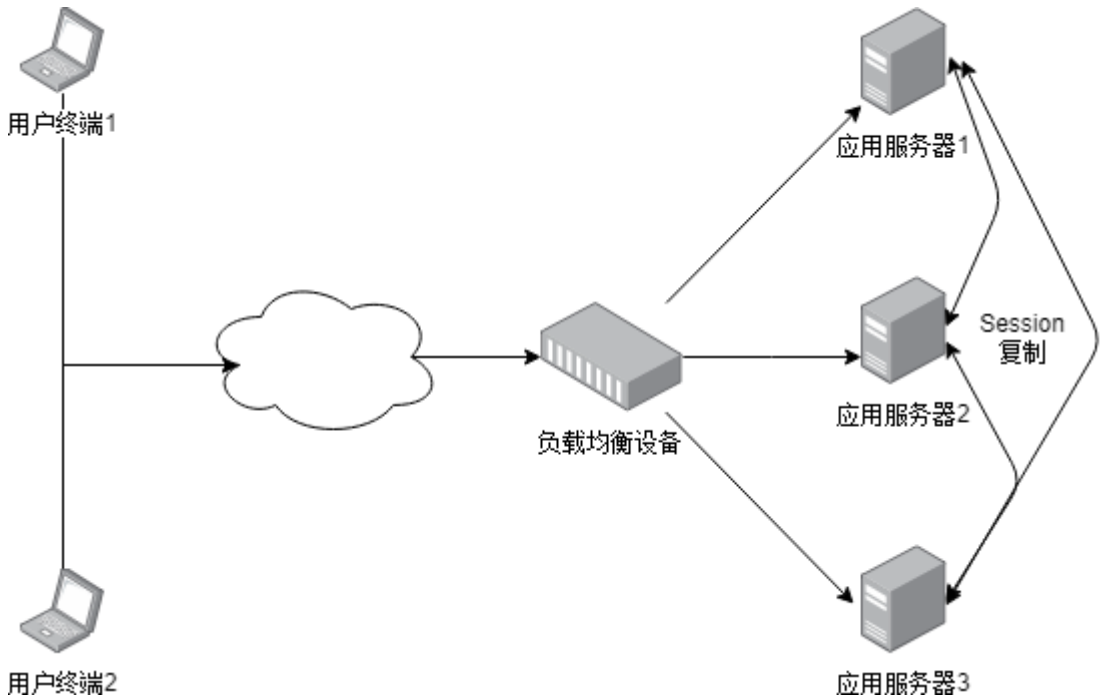
由于事实上业务一定是有状态的，把业务中上下文对象称为会话（Session）。在单机情况下，Session可以由部署在服务器上的Web容器（如Tomcat）管理；在负载均衡的集群环境下，由于负载均衡服务器可能将请求发送到任意应用服务器，所以不能单纯的交由Web容器管理。

集群情况下。Session管理有以下几种手段：

- **Session复制**
- **Session绑定**
- **利用Cookie记录Session**
- **Session服务器**

5.3.2.1Session复制

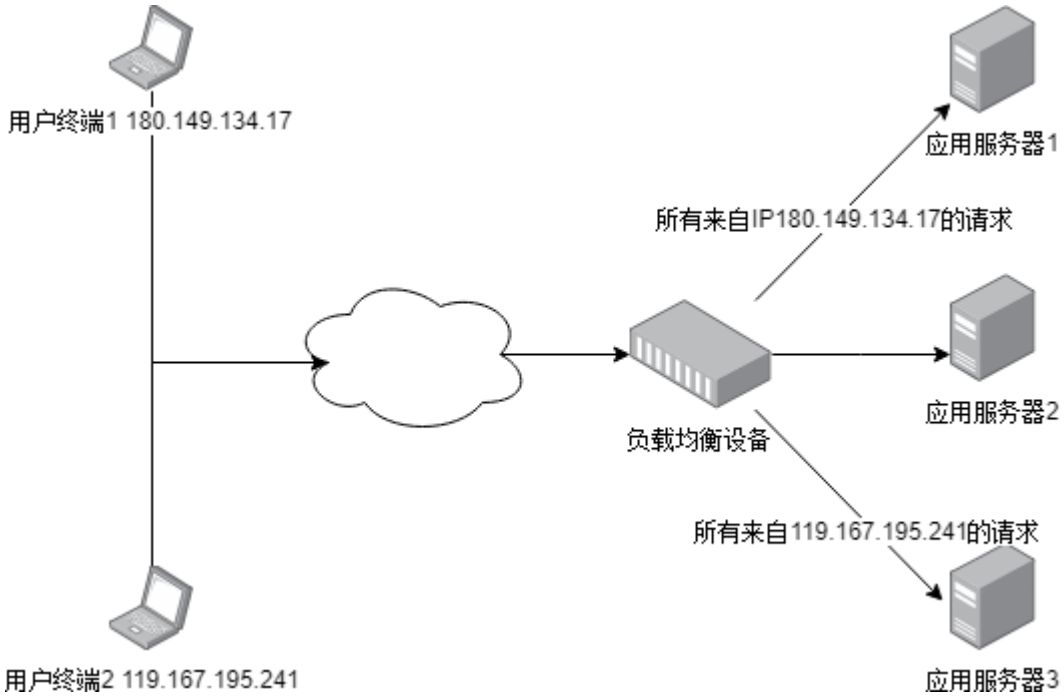
Session复制是指，在集群的服务器中进行Session同步，使得每台服务器上都保存所有用户的Session信息。



Session复制只能在集群规模较小的情况下使用，在集群规模较大的情况下，Session复制要占用大量服务器和网络资源，而且也会导致服务器内存不够Session使用的情况。

5.3.2.2Session绑定

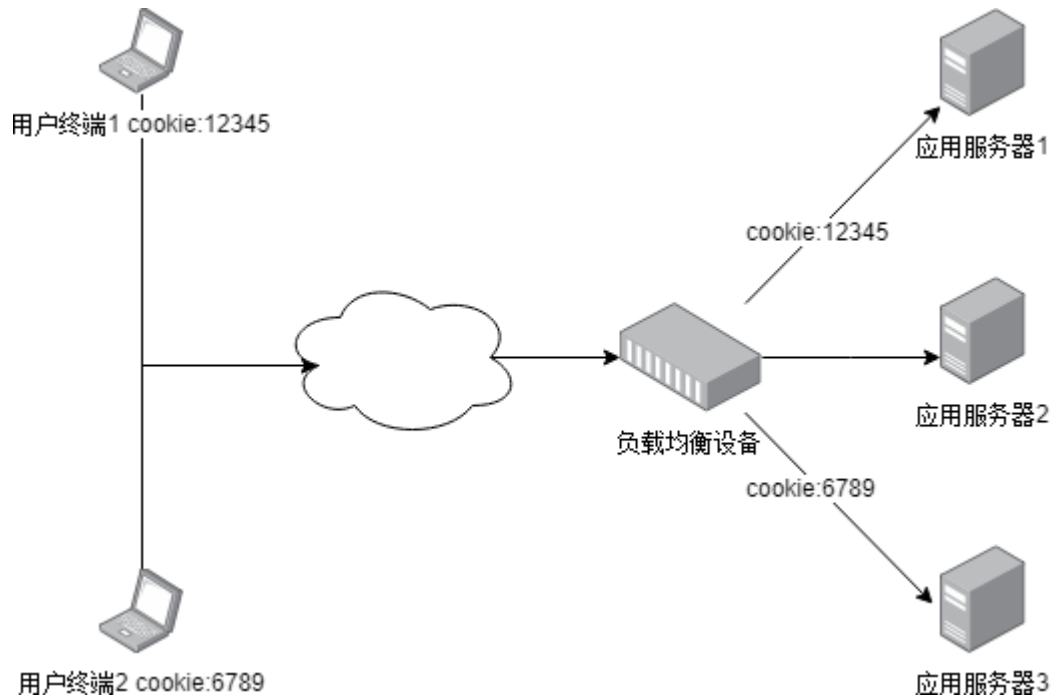
Session绑定可以利用负载均衡的源地址Hash算法实现，即将同一IP的请求发送到同一服务器上，在整个会话期间，该用户的所有请求都在该服务器上完成，Session绑定在该服务器上保证Session的获取，这种方法也叫会话粘滞。



这种设计**不符合高可用的要求**，一旦某台服务器宕机，该服务器绑定的Session将丢失，用户切换到其他服务器无法继续完成请求。

5.3.2.3利用Cookie记录Session

利用浏览器中的Cookie记录Session，在每次发送请求时，将记录的Session的Cookie顺带发送出来，保证应用服务器能得到会话的上下文信息。



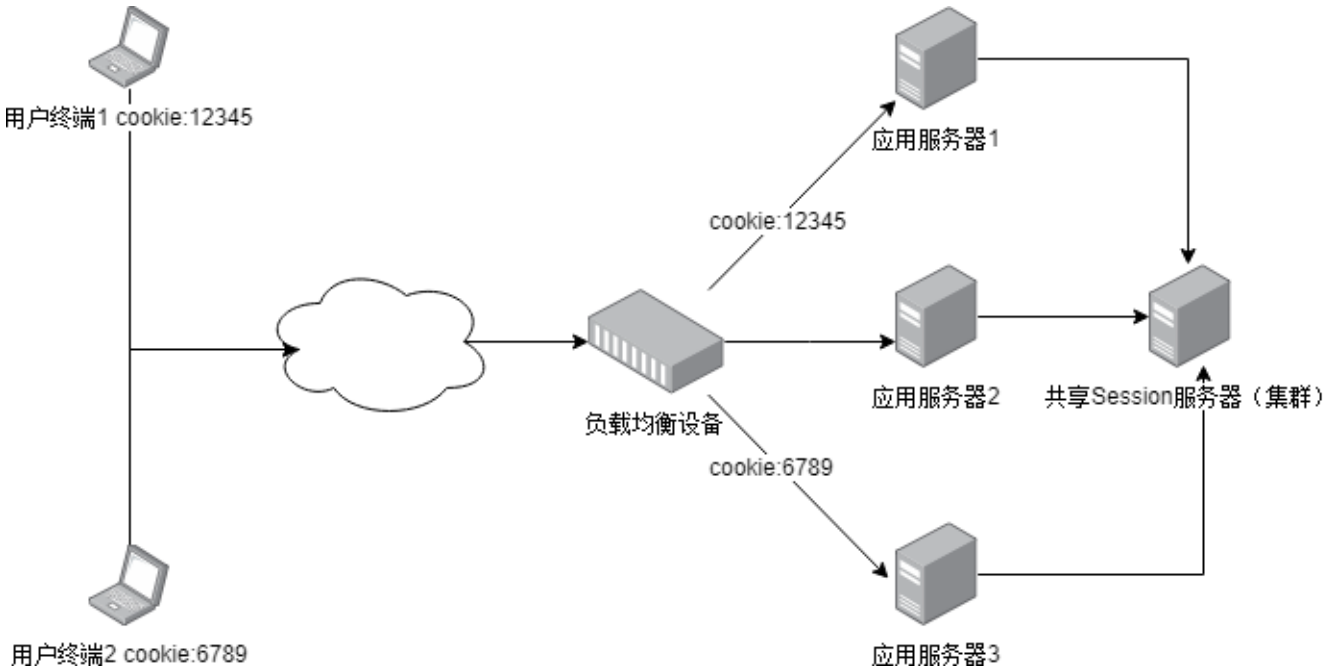
Cookie记录Session也有一些缺点：

- Cookie大小受限，能记录的信息有限；
- 每次请求都要传输Cookie，性能受限；
- 用户关闭Cookie，访问会不正常；

但是，Cookie的简单易用，可用性高，支持应用服务器线性伸缩，许多网站都使用Cookie记录Session。

5.3.2.4Session服务器

利用独立部署的Session服务器（集群）统一管理Session，应用服务器每次读写Session时，都访问Session服务器。



实际上，Session服务器是**将应用服务器状态分离**，分为无状态的应用服务器和有状态的Session服务器，然后再针对这两种服务器的不同特性分别设计架构。

5.4高可用服务

可复用服务模块为业务产品提供基础公共服务，服务通常都**独立分布式部署**，被具体应用远程调用。**可复用服务和应用一样，都是无状态的服务，所以可以使用负载均衡的失效转移策略实现高可用服务。**

5.4.1分级管理

将服务器进行分级管理，**核心应用和服务优先使用更好的硬件**，在运维响应上速度也迅速。

在服务部署上也要进行必要**隔离**，避免故障的连锁反应：低优先级的服务**通过启动不同的线程或者部署在不同的虚拟机上进行隔离**，高优先级的服务**则需要部署在不同物理机上，核心服务和数据甚至要部署在不同地域的数据中心。**

5.4.2超时设置

由于服务器宕机，线程死锁等原因，可能导致**无法响应调用服务的应用程序**，进而使得用户长时间得不到响应，而且还会占用应用程序的资源。

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，可选择**继续重试或将请求转移到其他提供相同服务的服务器上。**

5.4.3异步调用

应用对服务的调用通过**消息队列等异步方式完成**，避免一个服务失败导致整个应用请求失败的情况。

如提交一个新用户注册的请求，应用需要调用三个服务：将用户信息写入数据库；发送账户注册成功邮件；开通相应权限。

- 如果采用同步服务调用，当邮件队列阻塞不能发送邮件时，会导致其他两个服务无法执行，最后导致用户注册失败；
- 如果采用异步服务调用，应用程序将用户注册信息发送给消息队列服务器之后立即返回用户注册成功响应，而记录用户信息到数据库，发送用户注册成功邮件，调用用户服务开通权限这三个服务作为消息队列的消费者任务异步执行，即使邮件服务队列阻塞，邮件不能发送也不会影响其他服务执行，用户注册可以顺利完成，只是晚一点收到邮件而已。

5.4.4服务降级

在网站访问的高峰期，服务可能因为大量的并发调用而性能急剧下降，严重时可能导致服务宕机。**为了保证核心服务和功能的正常运行，需要对服务进行降级。**服务降级有两种手段：拒绝服务和关闭服务。

- 拒绝服务：**拒绝低优先级的服务，减少服务的并发数**，保证核心服务和功能的正常使用；或者**随机拒绝部分请求调用，节省资源**；
- 关闭服务：**关闭部分不重要的功能，或者服务内部关闭部分不重要的功能**，以节约系统开销，为重要的服务和功能让出资源；

5.4.5幂等性设计

幂等性是指：**重复发送多次请求得到的响应结果都是一样的。**

应用程序调用服务失败后，会将调用请求重新发送到另一个服务器上，**但是这个失败可能是由于网络问题造成的，也就是说，服务处理已经成功只是在返回结果时网络出现了问题**，这时应用重新提交请求就会导致服务重复调用。

服务重复调用是不可避免的，因此在服务层必须保证**服务重复调用和调用一次产生的结果相同，即服务具有幂等性。**

有些服务天生就有幂等性，如用户的性别设置，设置一次男性和设置多次男性结果都是一样的；而有些服务如转账服务就不具备幂等性，这时候就需要通过交易编号等信息进行**服务调用有效性检验**，只有有效的操作才能继续进行。

5.5高可用数据

不同于高可用应用和服务，**由于数据存储服务器上保存的数据不同，当某台服务器宕机时，数据访问请求不能任意切换到集群中其他机器上。**

保证数据存储高可用的手段主要是**数据备份、失效转移机制**：

- *数据备份*：保证数据有多个副本，任意副本的丢失都不会导致数据的永久丢失，实现数据完全持久化；
- *失效转移机制*：保证当一个数据副本无法访问时，可以快速切换到其他数据副本，保证系统可用；

5.5.1CAP原理

高可用数据有以下几层含义：

- **数据一致性（Consistency）**：

  为了实现数据持久性，不但每次写入数据时都要**写入持久性存储**，还需要将**数据备份一个或多个副本，存放在不同的物理设备上。**

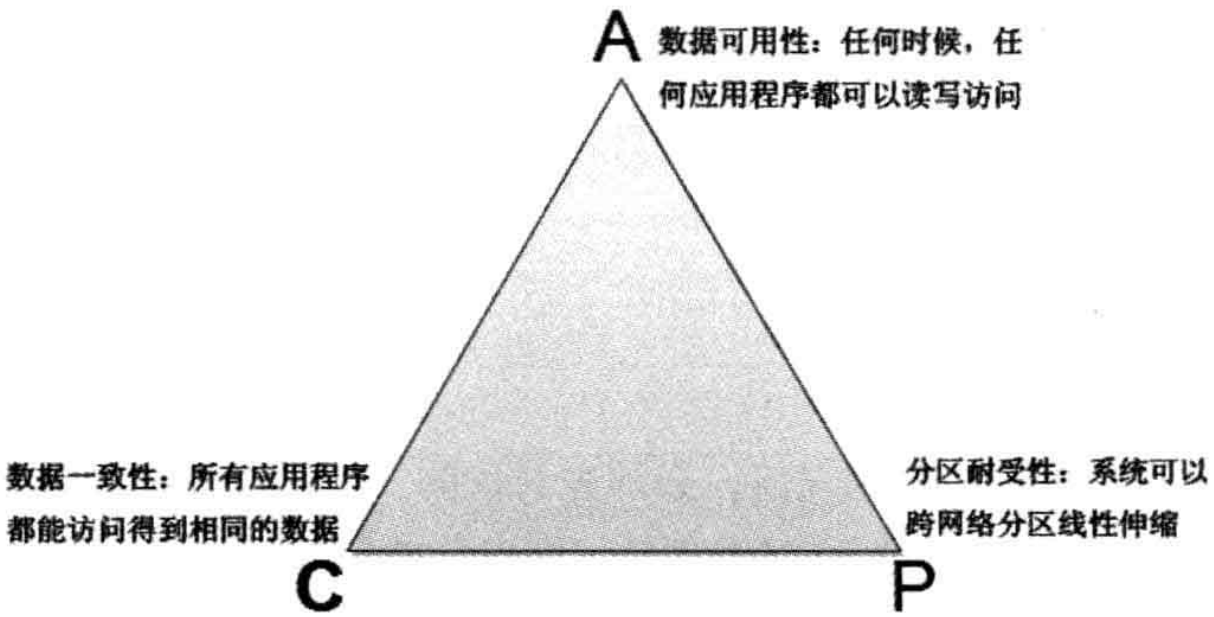
- **数据可用性（Availability）**：

  将多份数据副本放在不同存储设备的情况下，如果某一数据存储设备损坏，需要**将数据访问切换到另一台数据存储设备上。**

- **分区容错性（Partition Tolerance）**：

  在有多个数据备份的情况下，由于网络、服务器或者软件故障，会导致部分副本没有写入成功，**这会造成各个副本之间的数据不一致，数据内容冲突。**

CAP原理认为：**一个提供数据服务的数据存储系统无法同时满足数据一致性（Consistency）、数据可用性（Availability）、分区容错性（Partition Tolerance）这三个条件。**



在大型网站中，数据规模总是在扩展的，因此可伸缩性即**分区容错性**是必不可少的，即P是一定存在的；网站规模变大后，网络和服务器的故障频繁出现，所以必须要保证分布式处理系统的高可用性。

在大型网站中，一般都会保证分布式存储系统的可用性（A）和分区容错性（P）而在某种程度上放弃数据一致性（C）。

数据一致性又包括：

- 数据强一致性：

  各个副本中的数据始终是一致的，数据更新操作结果和操作响应总是一致的。

- 数据用户一致性：

  数据在各个副本中可能是不一致的，但是在用户访问的时候，通过纠错和校验机制，可以得到一个一致且正确的数据给用户。

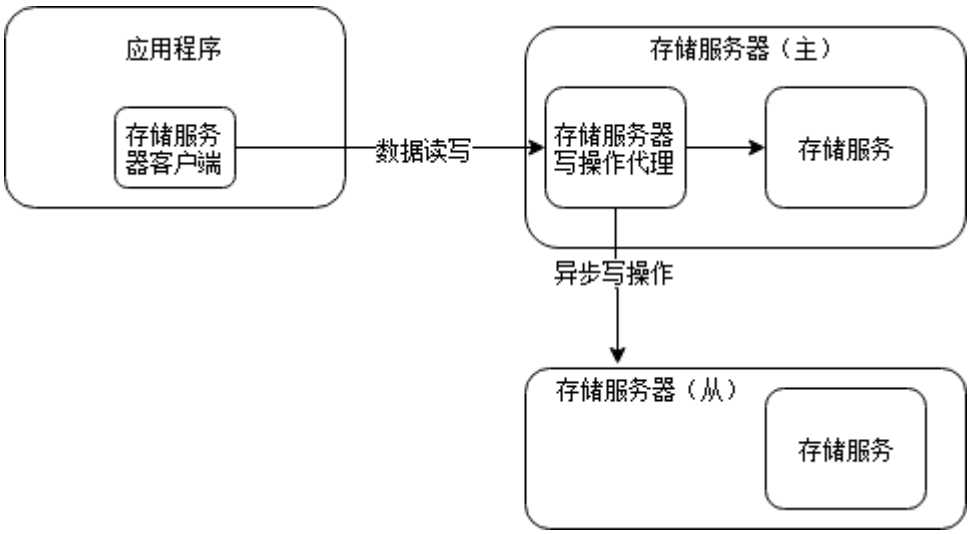
- 数据最终一致性：

  各个副本中的数据可能是不一致的，用户访问的数据也可能是不一致的，但系统进过一段时间（通常比较短）的自我恢复和修正，数据最终会达到一致。

一般会使系统达到用户一致性，保证最终用户访问得到正确的数据。

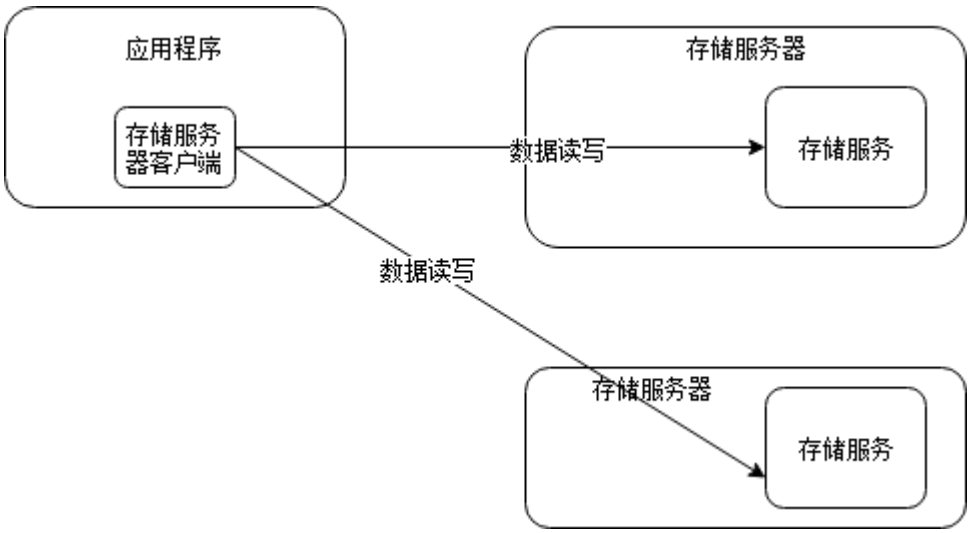
5.5.2数据备份

- *数据冷备*：即定期的将数据复制到某种存储介质上并物理存档保管。但是无法保证在长时间未备份后数据的一致性，而且数据恢复时间较长。
- *数据热备*：
  - **异步热备**：多份数据备份的写入操作异步完成，应用程序收到数据服务系统的写操作成功响应时，只写成功了一份，存储系统将会异步的写其他副本。



    异步写入方式下，存储服务器分为**主存储服务器（Master）和从存储服务器（Slave）**，应用程序正常情况下只连接主存储服务器，**数据写入时，由主存储服务器的写操作代理模块将数据写入本机存储系统后立即返回写操作成功响应，然后通过异步线程将写操作数据同步到从存储服务器。**

  - **同步热备**：多份数据副本的写入方式同步完成，**应用程序收到数据服务系统的写成功响应时，多份数据都已经写操作成功。**



同步热备具体实现的时候，为了提高性能，**在应用程序客户端并发向多个存储服务器同时写入数据**，等待所有存储服务器都返回操作成功的响应后，再通知应用程序写操作成功。

- **关系数据库热备机制就是Master-Slave同步机制，该机制不仅解决了数据库备份问题，还改善了数据库系统性能，即通过读写分离的方法，写操作访问Master数据库，读操作访问Slave数据库。**

5.5.3失效转移

失效转移：如果数据服务器集群中任意一台服务器宕机，那么应用程序**对于该服务器的所有读写操作都要重新路由到其他服务器**，才能保证数据访问不会失败。

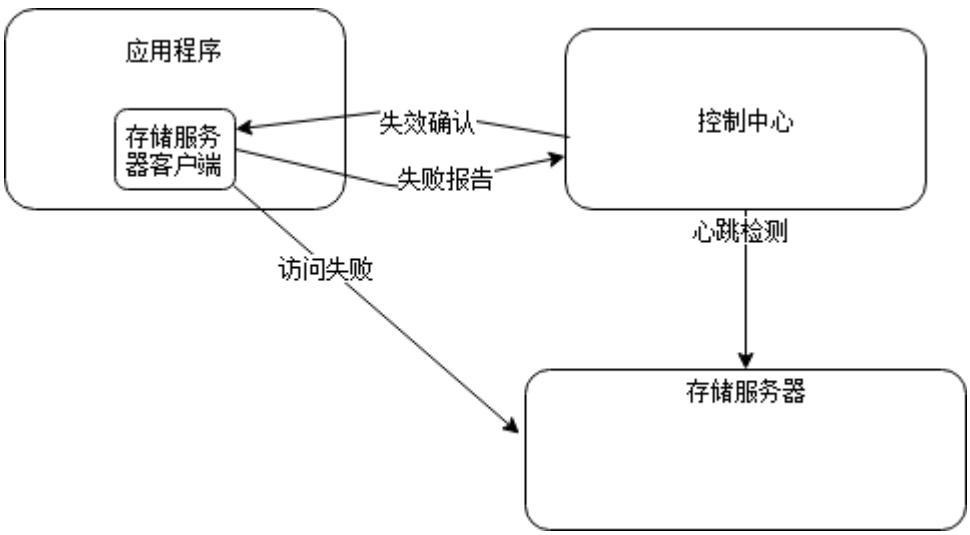
失效转移操作由三部分组成：失效确认、访问转移、数据恢复。

- **失效确认**：

  失效确认是判断服务器是否宕机，主要手段有两种：心跳检测和应用程序访问失败报告；

  - **心跳检测**：
  - **应用程序访问失败报告**：

    对于应用程序的访问失败报告，控制中心还要再发送一次心跳检测确认，以面误判服务器宕机。



- **访问转移**：

  确认某台数据存储服务器宕机以后，就需要将数据读写访问**重新路由到其他服务器上。**

  - 对于完全对等的服务器，应用程序根据配置直接切换到对等服务器上；
  - 对于不对等的服务器，就需要重新计算路由，选择存储服务器；
- **数据恢复**：

  因为某台服务器宕机，所以数据存储的副本数目会减少，必须将副本数目恢复到系统设定的值；

  系统需要从健康的服务器复制数据，将数据副本数目恢复到设定值。

5.6高可用网站的软件质量保证

5.6.1网站发布

- 网站发布任何新功能或修改布局，都需要在服务器上关闭原有应用，然后重新部署启动新的应用，这个过程要求不能影响用户使用；
- 网站的发布过程实际上和服务器宕机过程相当，对于系统可用性的影响和服务器宕机相似；在高可靠网站系统设计时，需要考虑网站发布带来的服务器宕机的频率。
- **发布过程中，每次关闭的服务器是集群中的一小部分，并在发布完成后立即可以访问**，因此整个发布过程不影响用户使用。

5.6.2自动化测试

- 目前大部分网站都采用Web自动化测试技术，使用自动测试工具或脚本完成测试。

5.6.3预发布验证

- 由于测试环境和线上环境不同，软件部署后经常会出现各种问题，**因此需要将代码先发布到预发布服务器上**，然后执行一些典型的业务流程，确认系统没问题后正式发布。
- 预发布服务器和真实服务器除了用户无法访问以外，其他都完全相同。

5.6.4代码控制

网站代码控制的核心是**既保证代码发布版本的稳定正确，又保证不同团队开发不受影响。**

- **主干开发，分支发布**：

  代码的修改都在主干上进行，需要发布时，就从主干上拉下一个分支发布，该分支就成为一个发布版本。
- **分支开发，主干发布**：

  任何修改都不能在主干上进行，需要开发一个新功能或者修复Bug时，都是从主干拉出一个分支，开发完成且测试通过后，合并到主干，然后从主干进行发布，主干上代码永远是最新的发布版本。

5.6.5自动化发布

5.6.6灰度发布

应用发布成功后，任然有可能出现故障，这时就要进行**发布回滚**，由于大型网站主要业务的应用服务器集群庞大，所以采用**灰度发布模式**，**将集群服务器分成若干部分，持续几天才把整个集群全部发布完，期间如果出现问题，只需要回滚已发布的服务器。**

5.7网站运行监控

网站的运行监控对于网站运维和架构设计至关重要。

5.7.1监控数据采集

- 用户行为日志收集：用户在浏览器上所有的操作以及所在的环境，包括用户操作系统与浏览器版本信息，IP地址，页面访问路径，页面停留时间等。这些数据对于统计网站PV/UV指标，分析用户行为，优化网站设计，个性化营销，推荐等非常重要。
    - 服务器端日志收集
    - 客户端浏览器日志收集
- 服务器性能监控：收集服务器性能指标，包括Load，内存占用，磁盘IO，网络IO等。
- 运行数据报告：监控一些与具体业务场景相关的技术和业务指标，如缓存命中率，平均响应时延，每分钟发送邮件数目，待处理的任务总数等。

5.7.2监控管理

- 系统报警：在服务器运行正常的情况下，各个监控指标都稳定在一个特定水平，如果某些指标超过特定阈值，意味着系统可能出现故障，需要对相关人员报警，及时采取措施。
- 失效转移：监控系统还可以在发现故障的情况下主动通知应用，进行失效转移。
- 自动优雅降级：监控系统会根据实时监控的运行状态，会适当卸载某些低负载的服务器，重新安装启动部分高负载应用，使应用负载总体均衡。

6.网站的伸缩性架构

网站的伸缩性指：不需要改变网站软硬件设计，仅仅通过改变部署服务器数量就可以扩大或缩小服务处理能力。

大型网站是由小型网站渐近发展而来的，在渐近过程中，最重要的技术手段就是**服务器集群**，通过不断向集群中添加服务器来增强整个集群的处理能力。

只要技术上能做到向集群中加入服务器数量和集群的处理能力成线性相关，那么网站就能不断提升自己的规模。

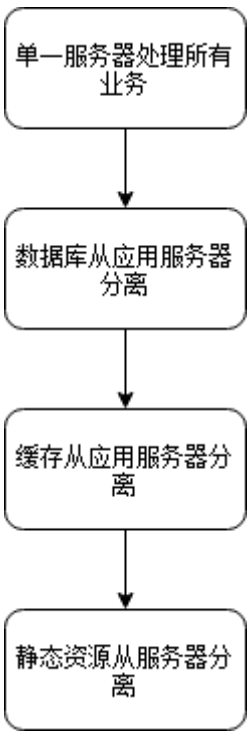
6.1网站架构的伸缩性设计

网站的伸缩性设计分为两类：

- **根据功能进行物理分离实现伸缩**：不同服务器部署不同服务，提供不同功能；
- **单一功能通过集群实现伸缩**：集群内的多台服务器部署相同的服务，提供相同功能；

6.1.1不同功能进行物理分离实现伸缩

通过增加新的服务器从现有服务器中分离出部分功能和服务，最终达到每个服务器（集群）负责某个特定服务的情况。每次分离都会有更多的服务器加入网站，使用新增的服务器处理某种特定服务。



- **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，实现伸缩性；

- **横向分离（业务分割后分离）**：将不同业务模块分离部署；

6.1.2单一功能通过集群规模实现伸缩

物理分离可以一定程度上实现伸缩性，但单一服务器的处理能力终究有限，因此必须使用**服务器集群**，**将相同服务部署在多台服务器上构成一个集群对外提供服务。**集群伸缩性分为应用服务器集群伸缩性和数据库服务器伸缩性。

6.2应用服务器集群伸缩性设计

由于应用服务器被设计为**无状态的**，即应用服务器不存储请求上下文，所以将用户请求发送到集群中任意一台服务器中，处理结果都是相同的。

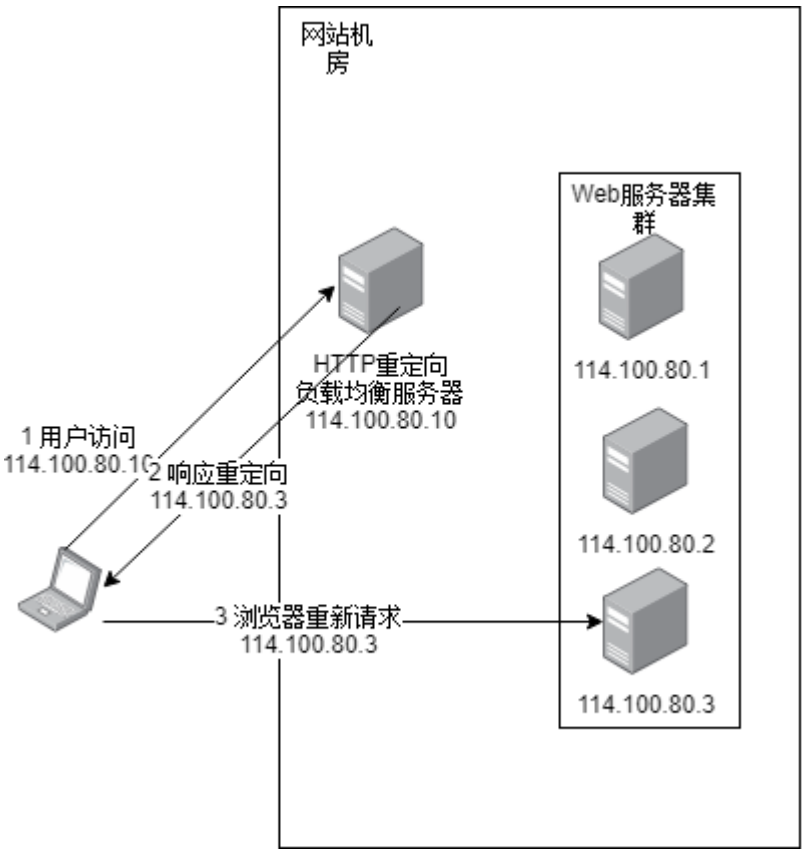
应用服务器的伸缩性：HTTP请求分发装置能够感知或配置集群中服务器数量，可以及时发现集群中新上线或下线的服务器，那么就实现了应用服务器集群的伸缩性。

负载均衡是网站不可缺少的基础手段，不但可以实现网站的伸缩性，也可以改善网站的可用性。具体手段如下：

- HTTP重定向负载均衡
- DNS域名解析负载均衡
- 反向代理负载均衡
- IP负载均衡
- 数据链路层负载均衡

6.2.1HTTP重定向负载均衡

利用HTTP重定向协议实现负载均衡，用户请求首先发送给HTTP重定向负载均衡服务器，该服务器会根据用户的HTTP请求计算一台真实应用服务器IP地址，并将该IP地址写入HTTP重定向响应中返回给用户浏览器。



该方案优点是：简单易实现；

缺点是：

- 浏览器需要两次请求才能完成一次访问，性能较差；
- 重定向服务器的处理能力会成为系统瓶颈；
- 使用302响应码重定向，可能会被搜索引擎判定为SEO作弊；

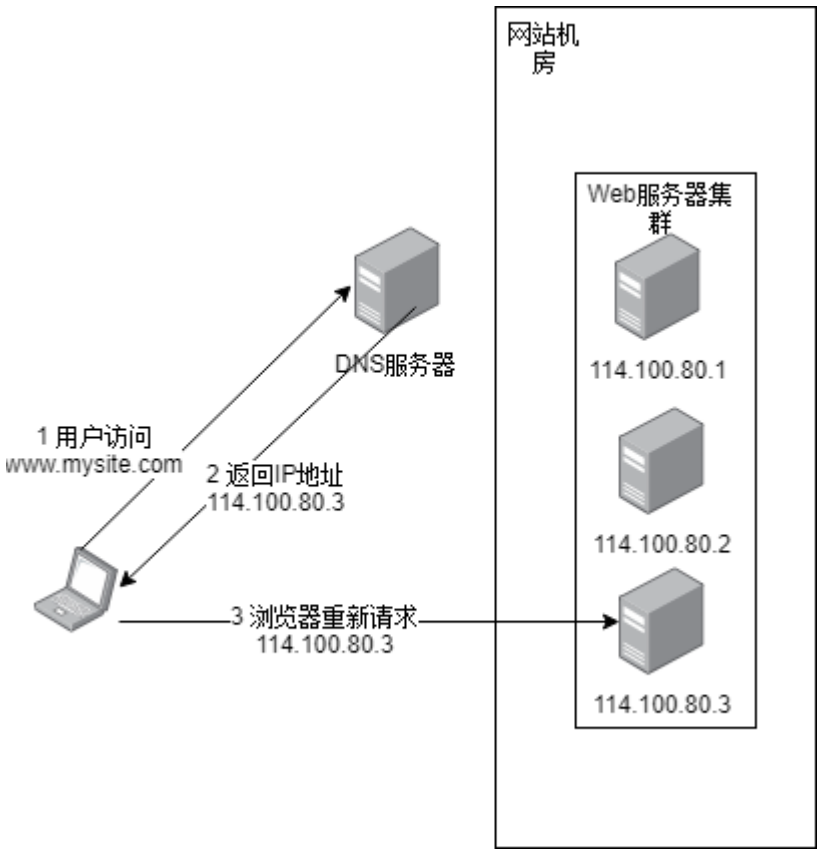
所以实际中很少使用。

6.2.2DNS域名解析负载均衡

利用DNS处理域名解析时同时进行负载均衡处理，在DNS服务器中配置多条A记录：

记录名	域名	IP地址
A	www.mysite.com	114.100.80.1
A	www.mysite.com	114.100.80.2
A	www.mysite.com	114.100.80.3

每次域名解析时，**根据负载均衡算法计算一个不同的IP地址返回**，这样A记录中配置的多个服务器就构成了一个集群实现负载均衡。



优点：

- 将负载均衡工作交给DNS服务器，节省了网站的维护；
- DNS支持基于地理位置的域名解析，会将域名解析为距离用户地理位置最近的服务器地址，加快用户访问；

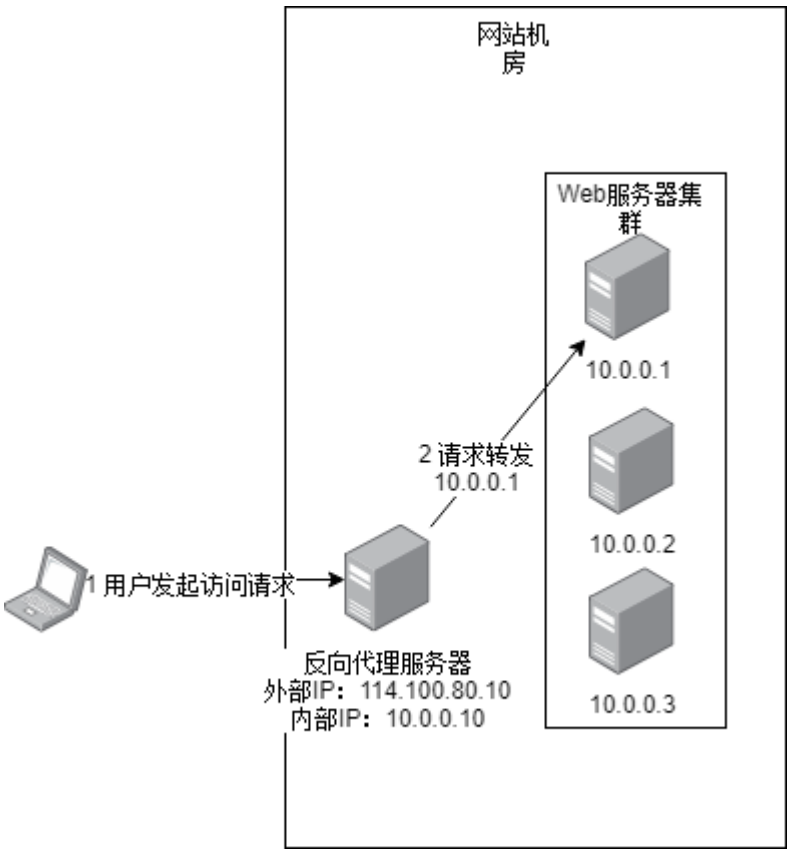
缺点：

- DNS解析是多级解析，每一级DNS都要缓存A记录，当某一台服务器宕机下线时，即使修改了A记录，要使其生效还需要一定时间，该时间内用户有访问失败的风险。
- DNS负载均衡的控制权在运营商手中，网站无法做出太大改善和管理；

大型网站总是部分使用DNS域名解析负载均衡，将其作为第一级负载均衡手段。即DNS解析得到是提供负载均衡的内部服务器，这组服务器再进行负载均衡，将请求发送到真实的Web服务器上。

6.2.3反向代理负载均衡

利用反向代理服务器进行负载均衡，反向代理服务器处于Web服务器前面，所以反向代理服务器其实承担了负载均衡服务器的功能，将请求根据负载均衡算法转发到不同Web服务器上。



由于反向代理转发请求在HTTP协议层面，也叫应用层负载均衡。

优点：

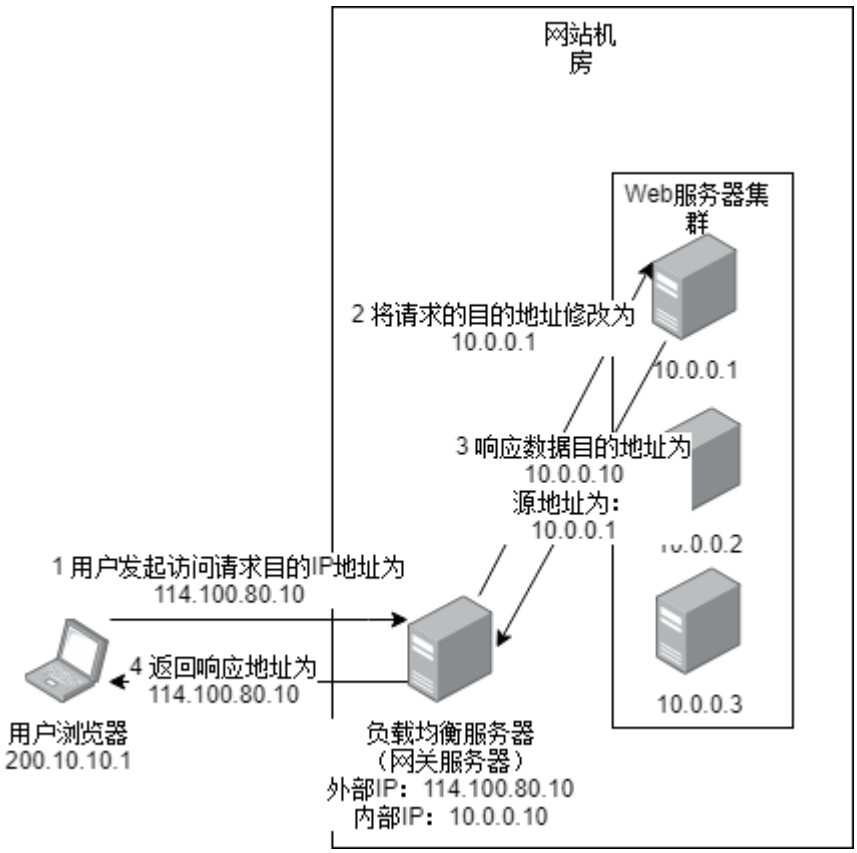
- 负载均衡和反向代理功能结合到一起，部署简单；

缺点：

- 反向代理服务器要处理所有请求，会成为系统瓶颈；

6.2.4IP负载均衡

在网络层修改请求的目的地址进行负载均衡，直接在OS内核进程中进行IP地址替换。



用户请求到达负载均衡服务器时，**负载均衡服务器在操作系统内核进程中获取网络数据包，根据负载均衡算法计算得到一台Web服务器的IP地址，并且将请求的目的地址修改为该服务器的IP地址**，全程无需用户进程参与；

Web服务器处理完成后，将响应数据发送给负载均衡服务器，**负载均衡服务器再将响应数据中的源地址修改为负载均衡服务器的IP地址**，然后发送给用户浏览器。

IP负载均衡中最关键的是，**响应数据如何返回给负载均衡服务器**，因为从Web服务器返回的数据包目的地址是用户的IP地址，负载均衡服务器是肯定无法得到的。有以下两种方法：

- **源地址转换（SNAT）**：在负载均衡服务器修改请求的目的IP地址时同时修改请求的源IP地址，设置为负载均衡服务器的IP地址，这样Web服务器就会将响应数据发送给负载均衡服务器；
- **负载均衡服务器作为网关服务器**：将负载均衡服务器同时作为物理集群的网关服务器，所以响应都要经由负载均衡服务器；

优点：

- 在操作系统内核进行数据分发，性能更好；

缺点：

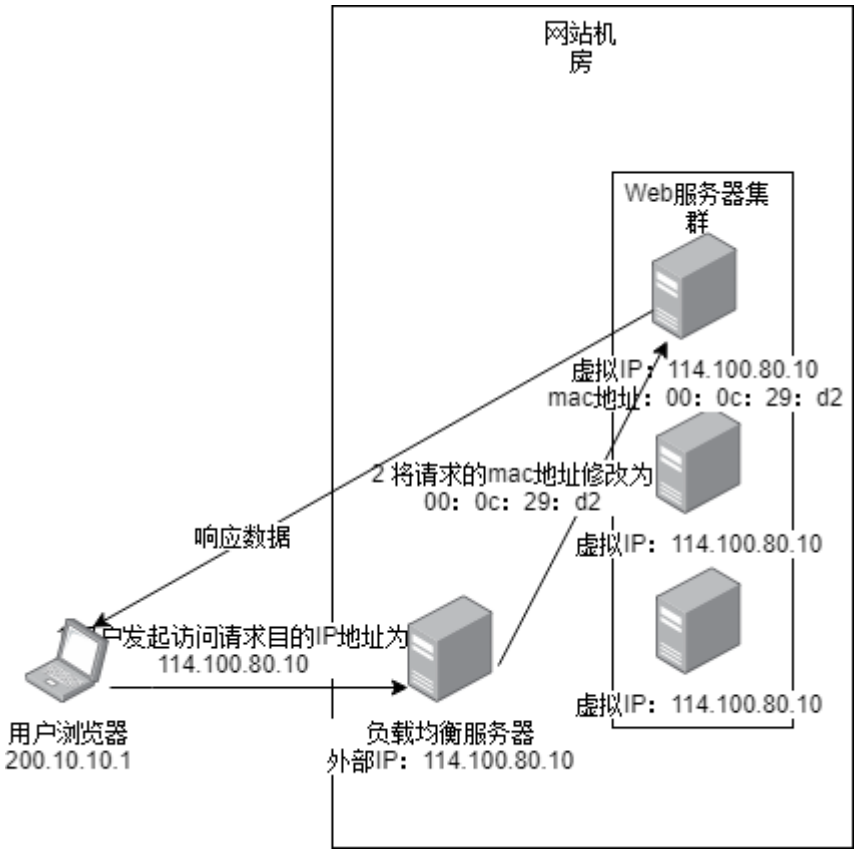
- 所有请求的响应都要经由负载均衡服务器，集群的最大数据吞吐量受限于负载均衡服务器网卡带宽，对于提供下载和视频服务等传输大量数据的网站而言，难以满足需求；

6.2.5数据链路层负载均衡

在通信协议中的数据链路层修改mac地址进行负载均衡，用户请求发送给负载均衡服务器，**通过配置真实物理服务器集群中所有机器的虚拟IP地址和负载均衡服务器IP地址一致，**负载均衡数据分发过程中不修改IP，只修改mac地址，这样就能达到数据分发的目的。

由于请求的源IP地址和目的IP地址均未修改，于是可以**直接从Web服务器将响应数据发送给用户浏览器**，避免了负载均衡服务器网卡带宽成为瓶颈；

这种负载均衡方式又称为**直接路由方式（DR）**



该方式是大型网站使用最广的负载均衡手段，Linux平台上最好的数据链路层负载均衡开源产品是LVS（Linux Virtual Server）

6.2.6负载均衡算法

负载均衡服务器主要完成下面两步工作：

- **根据负载均衡算法和Web服务器列表计算得到集群中一台Web服务器的地址**；
- **将数据请求发送到该地址对应的Web服务器上；**

负载均衡算法主要有：

- **轮询（Round Robin，RR）**：

  所有请求依次分发到集群的每台服务器上，每台服务器处理数量近似的请求，适合于服务器所有硬件都相同的场景。

- **加权轮询（Weighted Round Robin，WRR）**：

依据应用服务器硬件性能，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，高性能服务器分配更多请求；

- **随机（Random）**：

  请求会被随机分配到各个应用服务器，在很多场景下，该方案都简单实用；

- **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）**：

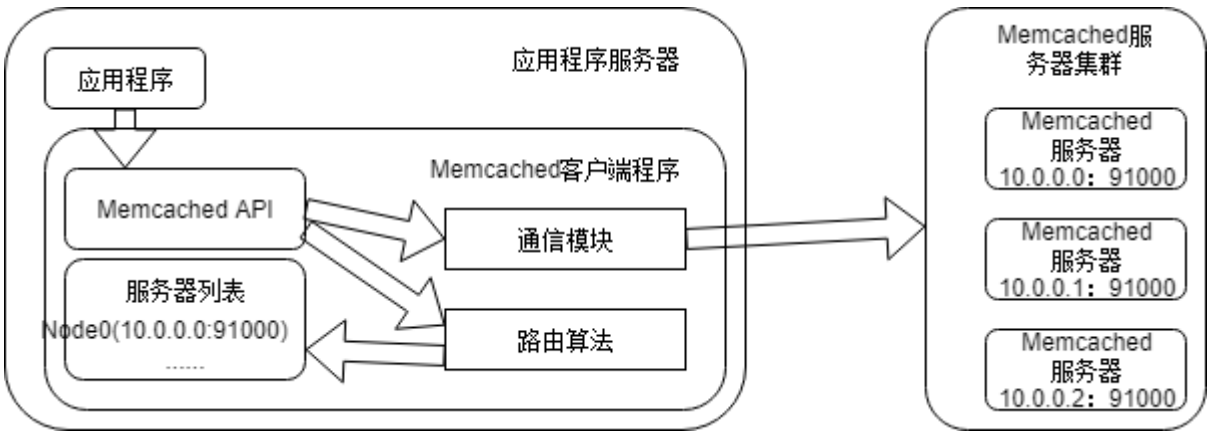
  根据请求的源IP地址进行Hash计算，映射到对应的服务器，这样来自同一个IP地址的请求总会在同一个服务器上处理，该请求的上下文信息可以存储在这台服务器上，但是这样会导致**会话粘滞**；

6.3分布式缓存集群伸缩性设计

分布式缓存集群的伸缩性不能简单的使用负载均衡手段来实现，因为分布式缓存服务器中存储的数据各不相同，缓存访问请求不可能在缓存集群中任意服务器上都能处理，这一特点严重制约了分布式缓存集群的伸缩性设计。

所以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，即**新加入缓存服务器后应使得整个缓存服务器中已被缓存的数据尽可能还能被访问到。**

6.3.1Memcached分布式缓存集群访问模型



应用程序通过Memcached客户端访问Memcached服务器集群，Memcached客户端主要由一组API，Memcached服务器路由算法，Memcached服务器列表及通信模块组成。

其中路由算法负责根据应用程序输入缓存数据KEY计算得到应该将数据写入到Memcached的哪台服务器或从哪台服务器读数据。

6.3.2Memcached分布式缓存集群的伸缩性挑战

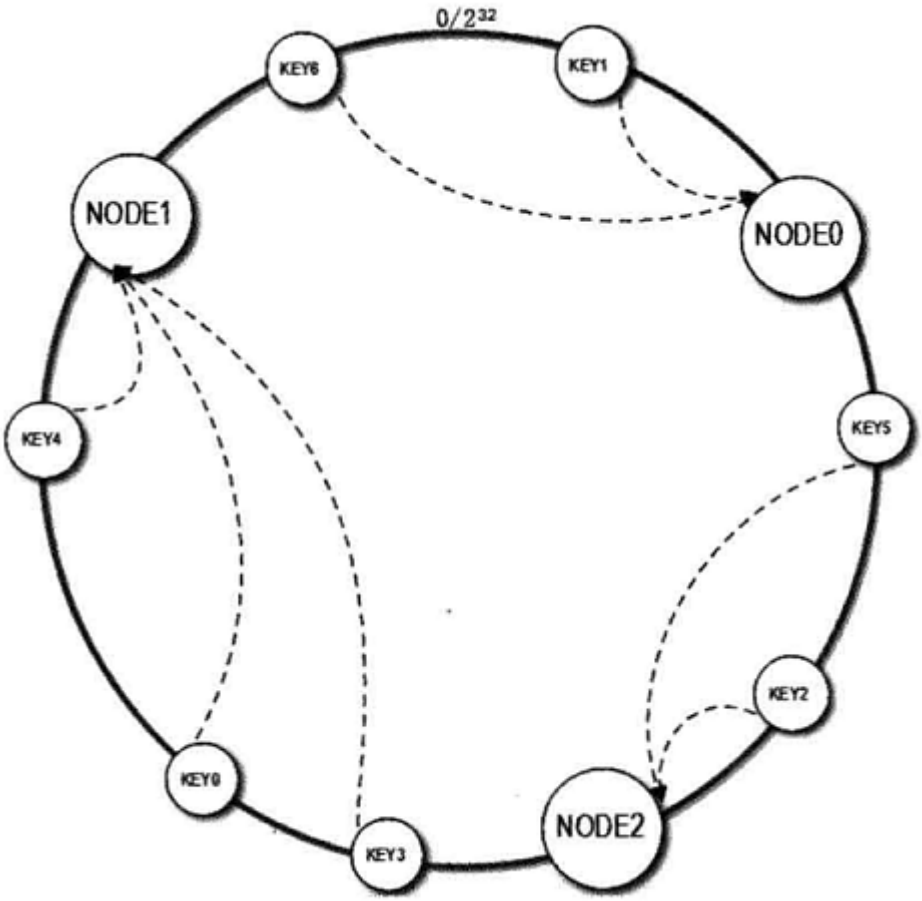
Memcached分布式缓存系统中，**路由算法至关重要**，决定着究竟该访问哪个Memcached服务器；

简单的路由算法有**余数Hash**，如果不考虑缓存服务器集群的伸缩性，余数Hash几乎可以满足大部分缓存路由需求。

当集群扩容时，缓存集群使用余数Hash的挑战就来了，**集群扩容后，会导致大部分读操作失败，从而使得对数据库的访问激增，可能会导致数据库服务器宕机。**

6.3.3分布式缓存的一致性Hash算法

解决分布式集群缓存扩容问题的方法是**一致性Hash算法**，实现KEY到缓存服务器的Hash映射。



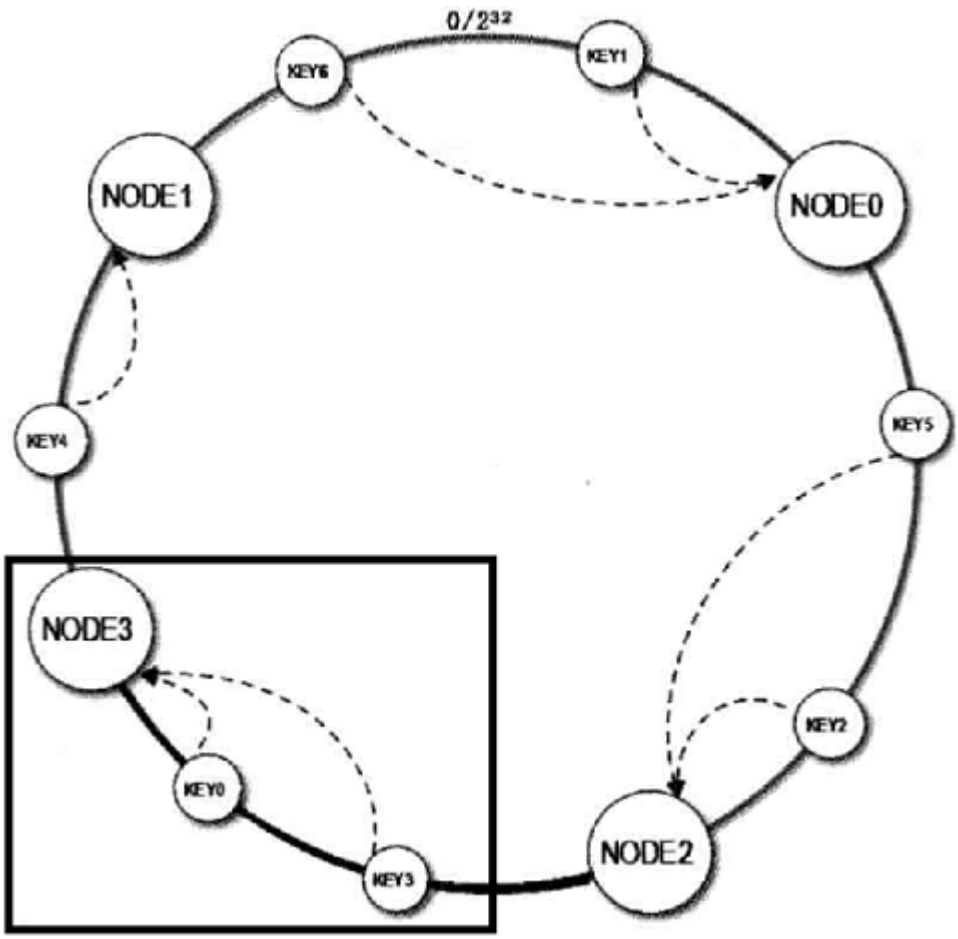
具体算法过程为：

- 构造一个长度为2^{32}的整数环，根据结点名称的Hash值（其分布范围为[0, $2^{32}-1$]）将缓存服务器结点放置在这个Hash环上；
- 在Hash环上顺时针查找距离这个KEY的Hash最近的缓存服务器结点，完成KEY到服务器的Hash映射查找；

例如：

如上图所示，NODE1的Hash值为3 594 963 423，NODE2的Hash值为1 845 328 979，而KEY0的Hash值为2 534 256 785，那么KEY0在环上顺时针查找，找到最近的结点为NODE1；

当服务器需要扩容时：只需要将新加入的结点名称（NODE3）的Hash值加入一致性Hash环中，假设NODE3的Hash值是2 790 324 235，加入NODE3后，KEY0（Hash值2 534 256 785）和KEY3（Hash值2 235 412 563）顺时针查找到的结点就是NODE3；



加入新结点NODE3后，原来的KEY大部分还能继续计算到原来的结点，只有KEY3、KEY0从原来的NODE1重新计算到NODE3，这样就能保证大部分被缓存的数据还能命中。

在具体应用中，这个长度为2^{32}的一致性Hash环通常使用**二叉查找树**实现，查找二叉查找树中大于等于KEY的Hash值的最小结点；

但是上面的算法任然有一些小问题，新加入结点NODE3只影响了结点NODE1，也就是说NODE3缓解了NODE1的压力，即NODE0和NODE2缓存数据量和负载压力是NODE1和NODE3的两倍。

解决上述一致性Hash算法带来的负载不均衡问题，**通过使用虚拟层的手段**：将每台物理缓存服务器虚拟为一组虚拟缓存服务器，将虚拟服务器的Hash值放置在Hash环上，KEY在环上先找到虚拟服务器结点，再得到物理服务器信息；

6.4数据存储服务器集群伸缩性设计

缓存的目的是加速数据读取并减轻数据存储服务器的压力，而数据存储服务器必须保证**数据可靠存储**，**必须保证在任何情况下数据的可用性和正确性**，所以数据存储服务器集群的伸缩性对数据的持久性和可用性提出了更高的要求。

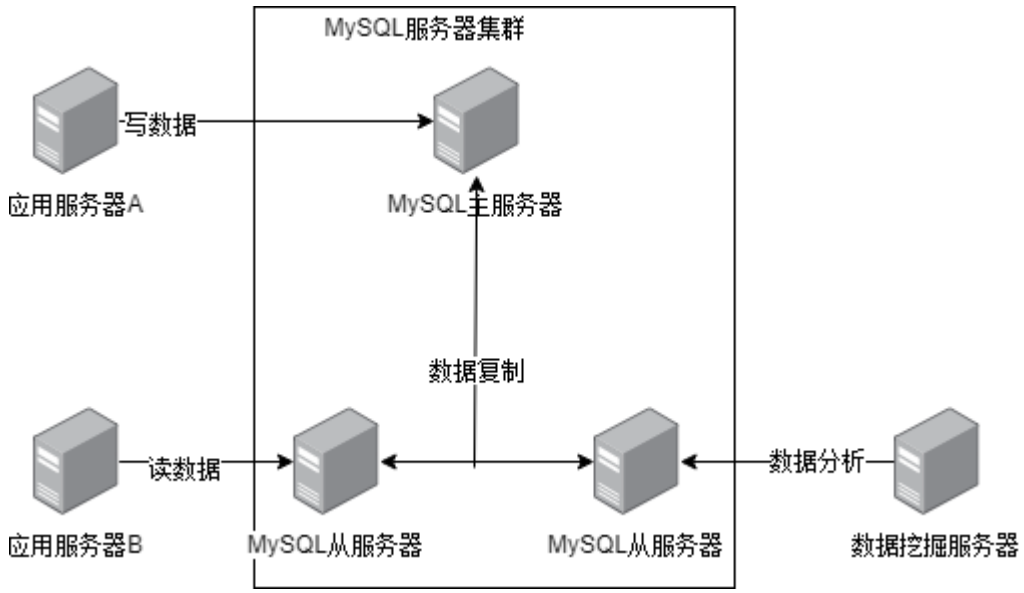
6.4.1关系数据库集群的伸缩性设计

主要来说关系数据库伸缩性的保证主要由以下三点做到：

- **主从复制，读写分离**

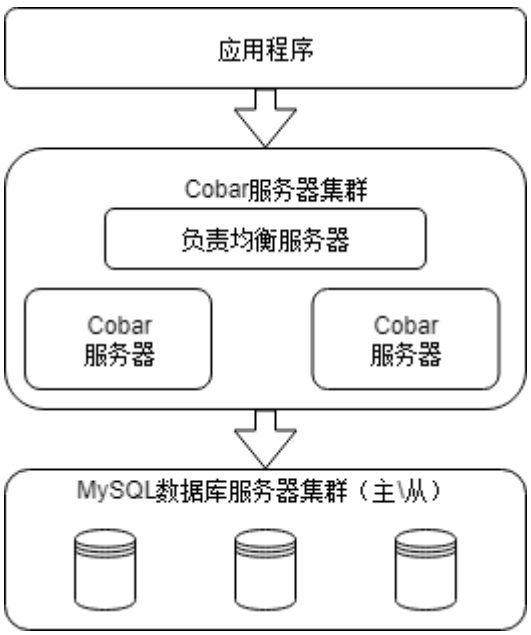
- **数据分库**
- **单表分片**

以MySQL为例说明关系数据库集群伸缩性的设计，在MySQL服务器集群中，虽然部署着多台MySQL实例，但是它们的角色有主从之分，**数据写操作都在主服务器上进行，由主服务器向从服务器进行数据同步，数据读操作、数据分析等操作都在从服务器上进行**，这就是主从复制，读写分离。

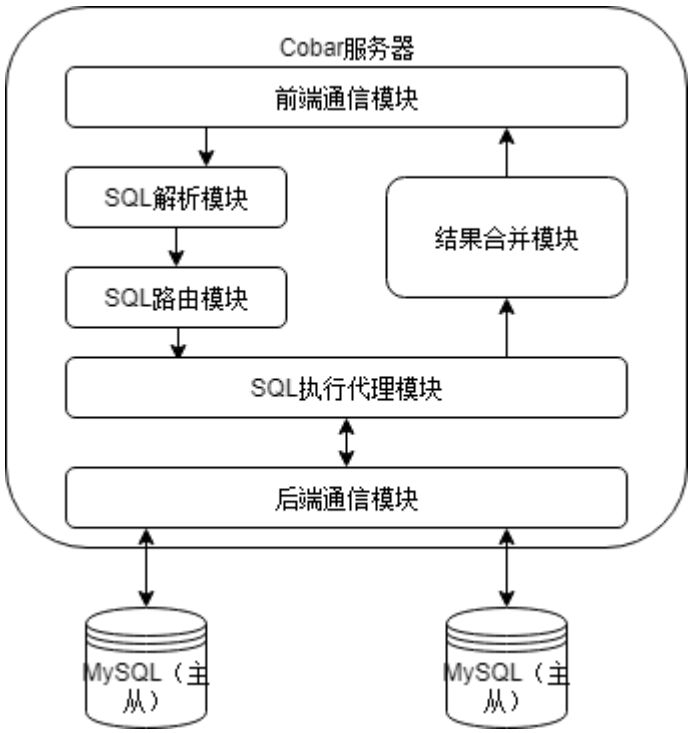


数据分库：根据业务的不同，将不同业务数据表部署在不同的数据库集群上。

单表分片：对于某些单表数据很多的表，则需要进行分片，**将一张表拆分开分别存储在多个数据库中。**Cobar是一个分布式关系数据库访问代理，介于应用服务器和数据库服务器之间。



应用程序通过JDBC驱动访问Cobar集群，Cobar服务器根据SQL和分库规则分解SQL，分发到MySQL集群不同的数据库实例上执行。上图为Cobar部署模型。



分布式数据库的一次访问请求分为以下两步：

- 查询过程**：前端通信模块负责和应用程序通信，接收到SQL请求(select * from users where userid in (12,22,23))，然后将SQL语句交给SQL解析模块得到路由规则查询条件（userid in (12,22,23)），SQL路由模块根据路由配置（userid为偶数路由至数据库A，userid为奇数路由至数据库B）结合规则查询条件将SQL语句分解为两条SQL语句，交给SQL执行代理模块发送至数据库A和数据库B执行；
- **返回查询结果过程**：数据库A和数据库B将执行结果返回至SQL执行代理模块，通过结果合并模块将两个返回结果集合并成一个结果集，最终返回给应用程序；

Cobar如何实现集群伸缩，有以下两种方案：

- **Cobar服务器集群伸缩**：Cobar服务器可以看成无状态的应用服务器，因此使用负载均衡设备完成；
- **MySQL服务器集群伸缩**：MySQL中存储着数据，想要保证集群扩容后数据的一致性，必须要进行数据迁移，将原来存储服务器上的数据迁移到新的服务器上（具体的迁移哪些数据可以使用一致性Hash算法）；

在实践中，Cobar使用了MySQL的数据同步进行数据迁移，数据迁移的单位是Schema，在Cobar集群初始化时，在每个MySQL实例创建多个Schema，集群扩容的时候，从每个服务器中迁移部分Schema到新机器中；

同步完成时，修改Cobar服务器的路由配置，将这些Schema的IP修改为新机器的IP，然后从原机器中删除相关Schema，完成MySQL集群扩容；

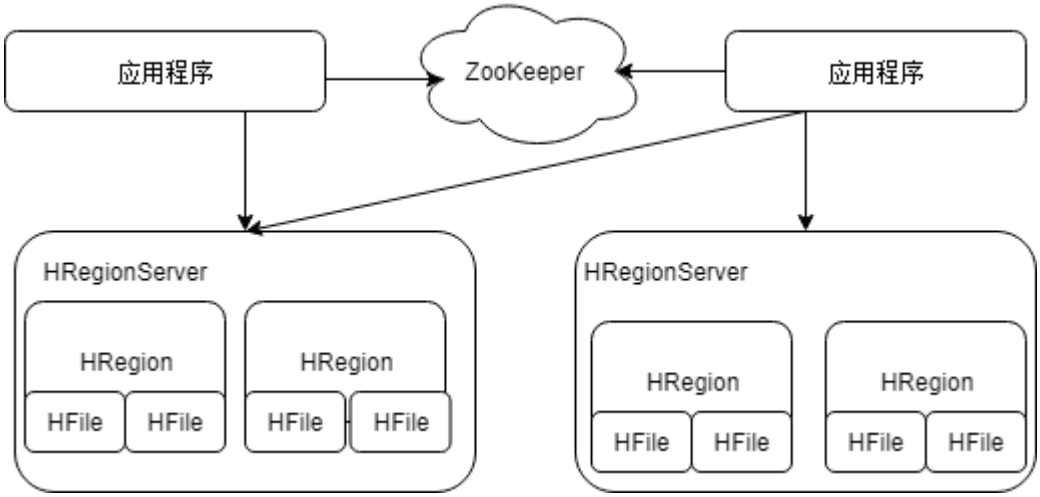
但是Cobar路由之后只能在单一数据库实例上处理查询请求，因此无法执行跨库JOIN查询，也不能执行跨库事务。

6.4.2NoSQL数据库伸缩性设计

NoSQL主要是指非关系的、分布式的数据库设计模式，NoSQL产品放弃了关系数据库两大重要基础：**结构化查询语言（SQL）和事务一致性保证（ACID）**，强化了大型网站更关注的特性：**高可用性和可伸缩性。**

下面以HBase为例，说明NoSQL数据库伸缩性设计：

HBase的伸缩性依赖的主要是：**可分裂的HRegion和可伸缩的分布式文件系统HDFS**，



HBase中，数据以HRegion为单位进行管理，也就是说应用程序如果访问一个数据，必须先要找到HRegion，然后将数据读写操作提交给HRegion，由HRegion完成存储层面的数据操作；

每个HRegion中存储一段Key值区间[Key1, Key2）的数据，HRegionServer是物理服务器，每个HRegionServer上启动多个HRegion实例，当一个HRegion中写入的数据太多，达到配置的阈值时，HRegion会分裂成两个HRegion，并将HRegion在整个集群中进行迁移，使得HRegionServer的负载均衡。

所有HRegion信息（存储的Key值区间、所在HRegionServer地址、访问端口号等）都记录在HMaster服务器上，

- **读操作**：**为了保证高可用，HBase启动多个HMaster，并通过ZooKeeper（一个支持分布式一致性的数据管理服务）选举出一个主服务器，应用程序通过ZooKeeper获得主HMaster的地址，输入Key值获得这个Key所在的HRegionServer地址，然后请求HRegionServer上的HRegion，获得需要的数据**；
- **写操作**：**首先先得到HRegion才能继续操作，HRegion会把数据存储在若干个叫做HFile格式的文件中，这些文件使用HDFS分布式文件系统存储，在整个集群内分布并高可用**；

如果集群中有新加入的服务器，即有新的HRegionServer，由于其负载较低，也会把HRegion迁移过去并记录到HMaster，从而实现HBase的线性伸缩；

7.网站的扩展性架构

扩展性是指：**在对现有系统影响最小的情况下，系统功能可持续扩展和提升的能力**；

扩展性和伸缩性的区别：

- **扩展性**：针对功能，是指对现有系统影响最小的情况下，系统功能可持续扩展或提升的能力；
- **伸缩性**：针对资源，是指系统通过增加（减少）自身资源规模的方式增强（减少）自己处理事务的能力；

7.1构建可扩展的网站架构

低耦合系统更容易扩展，低耦合系统模块更容易复用，设计网站可扩展的核心思想就是**模块化**，并在此基础上，**降低模块间的耦合性，提高模块的复用性**；

分层和**分割**是模块化设计的重要手段，**利用分层和分割的方式将软件分割为若干个低耦合的独立的组件模块，这些组件模块以消息传递及依赖调用的方式聚合成一个完整系统**；

模块通过分布式部署的方式，独立的模块部署在独立的服务器（集群）上，从物理上分离模块之间的耦合关系，进一步降低耦合提高复用性；

模块分布式部署以后具体聚合方式主要有：分布式消息队列和分布式服务。

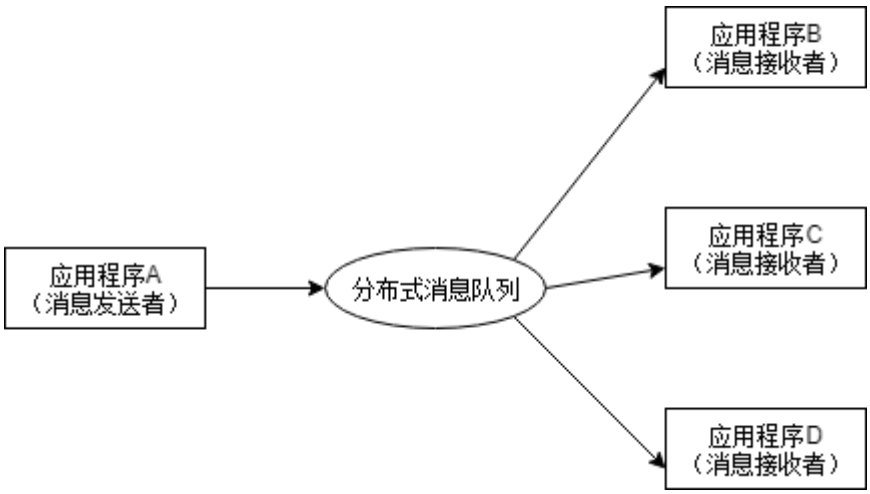
7.2利用分布式消息队列降低系统耦合性

若模块之间不存在直接调用，那么新增模块或修改模块就对其他模块影响很小，这样系统的扩展性更好；

7.2.1事件驱动架构

事件驱动架构（Event Driven Architecture）：通过**在低耦合的模块之间传输事件消息**，以保持模块的松散耦合，并借助事件消息的通信完成模块间合作；

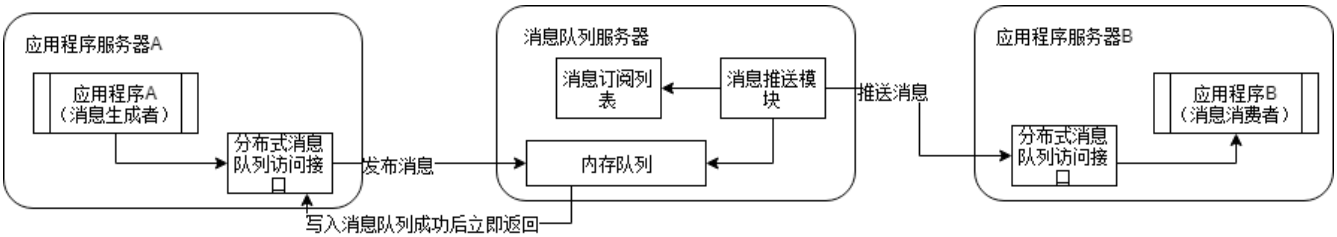
典型的EDA架构就是操作系统中常见的生产者消费者模式；



消息队列使用的是**发布-订阅模式**，消息发送者发布消息，一个或多个消息接收者订阅消息；消息发送者和消息接收者之间没有直接耦合，**消息发送者将消息发送至分布式消息队列即结束对消息的处理，消息接收者只需要从分布式消息队列获取消息后进行处理，不需要知道消息从何而来。**

7.2.2分布式消息队列

分布式消息队列可以看成将队列数据结构部署到独立的服务器上，应用程序通过远程访问接口使用分布式消息队列，进行消息存取操作，进而实现分布式的异步调用；



消息生产者应用程序通过**远程访问接口**将消息队列推送给消息队列服务器，消息队列服务器将消息写入本地内存队列后立即返回成功响应给消息生产者；消息队列服务器根据**消息订阅列表**查找订阅该消息的消息消费者应用程序，将消息队列中的消息按照先进先出的原则通过**远程通信接口**发送给消息消费者程序；

- **伸缩性：**消息队列服务器上的数据可以看成被即使处理的，因此类似于无状态服务器，伸缩性设计比较简单；
- **可用性：**分布式消息队列服务器内存空间不足时，会将消息写入磁盘；

为了避免消息队列服务器宕机造成消息丢失，会将消息成功发送到消息队列的消息存储在消息生产者服务器，等消息真正被消息消费者服务器处理后才删除消息；在消息队列服务器宕机后，生产者服务器会选择分布式消息队列服务器集群中其他服务器发布消息；

7.3利用分布式服务打造可复用的业务平台

使用分布式服务是降低系统耦合性的另一个重要手段，分布式消息队列是通过消息对象分解系统耦合性，**不同子系统处理同一个消息**；分布式服务调用则是通过**接口**分解系统耦合性，不同子系统通过相同的接口描述进行服务调用；

巨无霸应用系统的问题：

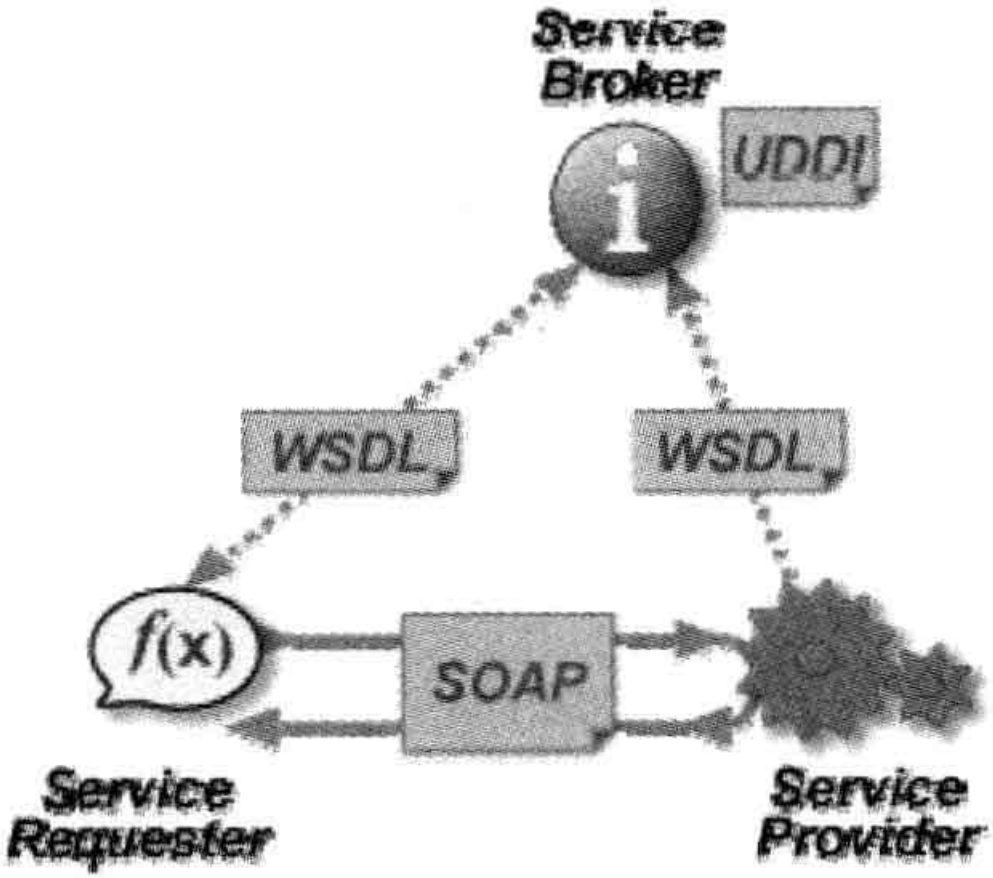
- 编译、部署困难
- 代码分支管理困难
- 数据库连接耗尽
- 新增业务困难

解决方案就是拆分，**将模块独立部署，降低系统耦合性**：

- 纵向拆分：将一个大应用拆分为多个小应用；
- 横向拆分：**将复用的业务拆分出来，独立部署为分布式业务**，新增业务只需要调用这些分布式服务，不需要依赖具体模块代码；

7.3.1Web Service与企业级分布式服务

Web Service是用以整合异构系统以及构建分布式系统的；



服务提供者（Service Provider）通过WSDL向注册中心（Service Broker）描述自身提供的服务接口属性，注册中心使用UDDI发布服务提供者提供的服务，服务请求者（Service Requester）从注册中心检索到服务消息后，通过SOAP和服务提供者通信，使用相关业务；

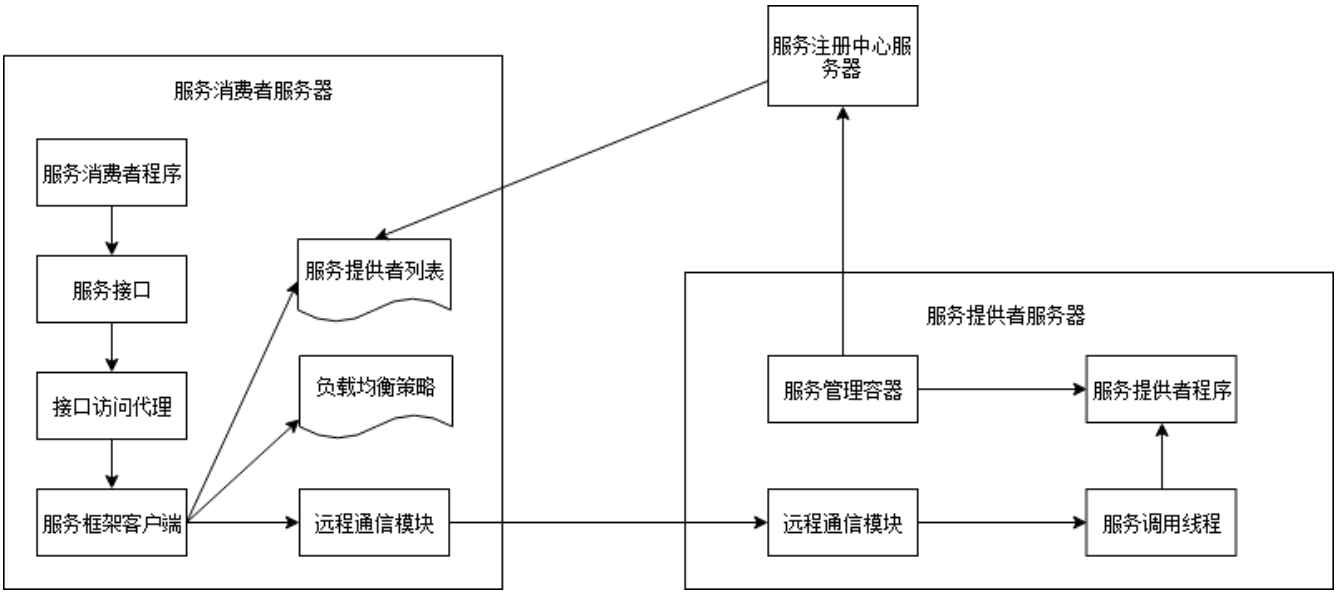
7.3.2大型网站分布式服务的需求与特点

对于大型网站，分布式服务框架要提供的功能有：

- 服务注册于发现：
- 服务调用：
- 负载均衡：分布式服务架构要能够支持服务请求者**使用可配置的负载均衡算法访问服务**，使服务提供者集群实现负载均衡；
- 失效转移：一旦服务不可用，就会影响到很多应用的可用性，分布式服务框架支持服务提供者的失效转移机制，当某个服务实例不可用，就将访问切换到其他服务实例上，实现服务整体高可用；
- 高效远程通信：对于大型网站，核心服务每天调用次数会达到数以亿次，如果没有高效的远程通信手段，服务调用会成为系统性能瓶颈；
- 整合异构系统：
- 对应用最少侵入：
- 版本管理：
- 实时监控：

7.3.3分布式服务框架设计

下面以开源的分布式服务框架Dubbo为例，说明分布式服务框架的设计。



服务消费者程序通过**服务接口**使用服务，而服务接口通过**代理**加载具体服务，具体服务可以是本地的代码模块，也可以是远程的服务；

服务框架客户端模块通过**服务注册中心**加载服务提供者列表（服务提供者启动后自动向服务注册中心注册自己可提供的服务接口列表），查找需要的服务接口，并根据配置的负载均衡策略将服务调用请求发送到某台服务提供者服务器，如果服务调用失败，客户端会自动从服务提供者列表选择一个可提供同样服务的另一台服务器重新请求服务，实现服务的自动失效转移，保证服务高可用；

7.4可扩展的数据结构

传统数据库为了保证关系运算的正确性，在设计数据库表结构的时候，就需要指定表的schema（字段名称、数据类型等），但是这会在需求变更时变得很僵硬；

NoSQL数据库的ColumnFamily能够无需修改表结构就新增字段；

7.5利用开放平台建设网站生态圈

开放平台时网站内部和外部交互的接口，外部需要面对众多的第三方开发者，内部需要面对网站内诸多的业务服务，开放平台的设计包括：

- API接口：
- 协议转换：
- 安全：
- 审计：
- 路由：
- 流程：

8.网站的安全架构

8.1网站应用攻击和防御

8.2信息加密技术及密钥安全管理

8.3信息过滤和反垃圾

8.4电子商务风险控制

三、案例