

分布式

一、大型网站系统的特点

- 高并发，大流量
- 高可用
- 海量数据
- 用户分布广泛，网络情况复杂
- 安全环境恶劣
- 需求快速变更，发布频繁
- 渐进式发展

二、大型网站架构演化发展历程

- 初始阶段的网站架构
- 应用服务和数据服务分离
- 使用缓存改善网站性能
- 使用应用服务器集群改善网站的并发处理能力
- 数据库读写分离
- 使用反向代理和 CDN 加速网站响应
- 使用分布式文件系统和分布式数据库系统
- 使用 NoSQL 和搜索引擎
- 业务拆分
- 分布式微服务

三、拆分 VS 集群

四、微服务 VS SOA

五、前后端完全分离与Rest规范

六、CAP三进二和Base定理

- 关系型数据库遵循ACID规则
- CAP三进二
- BASE定理
- 分布式一致性理论paxos、raft、zab算法

中间件

一、缓存

- 为什么要使用缓存
- 优秀的缓存系统Redis
- redis为什么这么快
- redis的数据类型，以及每种数据类型的使用场景
- redis的过期策略以及内存淘汰机制
- 渐进式ReHash
- 渐进式rehash的原因
- 渐进式rehash的步骤

- 缓存穿透

- 缓存雪崩

二、消息队列

消息队列应用场景

- 异步处理
- 应用解耦
- 流量削锋
- 日志处理
- 消息通讯

消息中间件示例

- 电商系统
- 日志收集系统

- JMS消息服务

- 消息模型
- 消息消费
- 防止消息丢失
 - 同步的事务——停止等待
 - 同步的事务——连续ARQ
 - 异步的事务——回调机制
- 消息的幂等处理
- 消息的按序处理

三、搜索引擎

- 概述
- 特点（优势）：
- 使用场景：
 - 倒排索引
 - 创建索引
- 一些要索引的原文档(Document)
- 将原文档传给分词组件(Tokenizer)
- 将得到的词元(Token)传给语言处理组件(Linguistic Processor)
- 将得到的词(Term)传给索引组件(Indexer)

- 搜索索引
 - 用户输入查询语句
 - 对查询语句进行词法分析，语法分析，及语言处理
 - 搜索索引，得到符合语法树的文档
 - 根据得到的文档和查询语句的相关性，对结果进行排序

- Lucene和ElasticSearch

- 分词器

大数据与高并发

一、秒杀架构设计

- 业务介绍

- 业务特点

- 瞬时并发量大

- 库存量少

- 业务简单

- 技术难点

- 现有业务的冲击

- 直接下订单

- 页面流量突增

- 架构设计思想

- 限流

- 削峰

- 异步

- 缓存

- 整体架构

- 客户端优化

- 秒杀页面

- 防止提前下单

- API 接入层优化

- 限制用户维度访问频率

- 限制商品维度访问频率

- SOA 服务层优化

- 秒杀整体流程图

- 总结

二、数据库架构发展历程

- 单机MySQL的美好年代
- Memcached(缓存)+MySQL+垂直拆分
- Mysql主从复制读写分离
- 分表分库+水平拆分+mysql集群
- 三、MySQL的扩展性瓶颈
- 四、为什么要使用NOSQL NOT ONLY SQL
- 五、传统RDBMS VS NOSQL
- 六、NOSQL数据库的类型
- 七、阿里巴巴中文站商品信息如何存放
 - 商品基本信息
 - 商品描述、详情、评价信息(多文字类)
 - 商品的图片
 - 商品的关键字
 - 商品的波段性的热点高频信息
 - 商品的交易、价格计算、积分累计
 - 大型互联网应用(大数据、高并发、多样数据类型)的难点和解决方案
- 八、数据的水平拆分和垂直拆分
 - 垂直拆分
 - 水平拆分
 - 拆分原则
 - 案例分析
- 九、分布式事务
 - 假如没有分布式事务
 - 什么是分布式事务?
 - XA两阶段提交 (2PC)
 - XA三阶段提交 (3PC)
 - MQ事务
 - TCC事务
- 十、BitMap
 - Bit-map的基本思想
 - Bit-map应用之快速排序
 - Bit-map应用之快速去重
 - Bit-map应用之快速查询
 - Bit-map扩展——Bloom Filter(布隆过滤器)
 - 总结
 - 应用
- 十一、Bloom Filter
- 十二、常见的限流算法
 - 计数器法
 - 滑动窗口
 - 漏桶算法
 - 令牌桶算法
 - 计数器 VS 滑动窗口
 - 漏桶算法 VS 令牌桶算法
- 十三、负载均衡
 - dns域名解析负载均衡
 - 反向代理负载均衡
 - http重定向协议实现负载均衡
 - 分层的负载均衡算法
- 十四、一致性Hash算法

数据库

- 一、数据库范式
- 1NF(第一范式)

- 2NF(第二范式)
- 3NF(第三范式)
- 二、数据库开发规范**
 - 基础规范
 - 命名规范
 - 字段设计规范
 - 总结
- 三、数据库索引**
 - 唯一索引
 - 非唯一索引
 - 主键索引
 - 聚集索引 (聚簇索引)
 - 扩展：聚集索引和非聚集索引的区别？分别在什么情况下使用？
 - 索引实现机制
 - 索引建立原则
- 四、MyISAM vs InnoDB**
- 五、并发事务带来的问题**
 - 丢失更新
 - 脏读 (未提交读)
 - 不可重复读
 - 幻读 (Phantom Read)
- 六、事务隔离级别及锁的实现机制**
 - 一级封锁协议 (对应 read uncommitted)
 - 二级封锁协议 (对应read committed)
 - 三级封锁协议 (对应repeatable read)
 - 最强封锁协议 (对应Serialization)
- 七、MVCC (多版本并发控制)**
- 八、间隙锁与幻读**
 - 间隙锁 (Next-Key锁)
 - RR级别下防止幻读

设计模式与实践

- 一、OOP五大原则SOLID**
 - 单一责任原则
 - 开放封闭原则
 - 里氏替换原则
 - 依赖倒置原则
 - 接口分离原则
- 二、设计模式**
- 三、代理模式**
 - 定义与举例
 - 静态代理
 - 动态代理
 - JDK动态代理
 - CGLIB动态代理
- 四、面向切面编程 (AOP)**
 - 基本思想
 - 登录验证
 - 基于RBAC的权限管理
 - 角色访问控制 (RBAC)
 - 执行流程分析
 - 日志记录
 - 日志记录最佳实践
 - 事务处理

统一异常处理

五、工厂模式

简单工厂

工厂方法

抽象工厂

六、控制反转IOC

七、观察者模式

八、Zookeeper

ZK简介

存储结构

znode

znode中的存在类型

应用场景

统一命名服务

负载均衡

统一配置管理

集群管理

服务器动态上下线

写数据流程

Leader选举

数据结构与算法

一、树

二、BST树

三、BST树

四、AVL树

五、红黑树

六、B-树

七、B+树

八、字典树

九、跳表

十、HashMap

简介

内部实现

存储结构-字段

功能实现-方法

1. 确定哈希桶数组索引位置

2. 分析HashMap的put方法

3. 扩容机制

线程安全性

十一、ConcurrentHashMap

锁分段技术

CAS无锁算法

实现方式

存在的缺点

十二、ConcurrentLinkedQueue

延迟更新tail节点

延迟删除head节点

十三、Topk问题

简述

解决方案

实际运行

(1) 单机+单核+足够大内存

(2) 单机+多核+足够大内存

(3) 单机+单核+受限内存

(4) 多机+受限内存

经常被提及的该类问题

重复问题

十四、资源池思想

作用

线程池

连接池

十五、JVM内存管理算法

判断对象是否存活

引用计数法

可达性分析算法

垃圾回收算法

标记-清除算法(Mark-Sweep)

复制算法(Copying)

标记-整理算法(Mark-Compact)

分代收集算法(Generational Collection)

十六、容器虚拟化技术，Docker思想

为什么会有Docker

Docker理念

实现方式

Docker的组成

镜像

容器

仓库

总结

十七、持续集成、持续发布，jenkins

持续集成

手动部署

自动部署

面试题举例

一、设计一个分布式环境下全局唯一的发号器

1、UUID

2、数据库自增长序列或字段

3、数据库sequence表以及乐观锁

4、Redis生成ID

5、Twitter的snowflake算法

二、设计一个带有过期时间的LRU缓存

问题描述

问题分析

过期时间实现

维护一个线程

惰性删除

三、设计一个分布式锁

什么是分布式锁？

我们需要怎样的分布式锁？

基于数据库做分布式锁

1、基于乐观锁

2、基于悲观锁

基于Redis做分布式锁

1、基于redis的setnx()、expire()方法做分布式锁

2、基于redis的setnx()、get()、getset()方法做分布式锁

基于ZooKeeper做分布式锁

使用分布式锁的注意事项
分布式可重入锁的设计
四、设计一个分布式环境下的统一配置中心
配置中心概述
演进中的配置
配置中心之简版
配置中心之性能改进
配置中心之可用性改进
五、如何准备HR面试

分布式

一、大型网站系统的特点

高并发，大流量

需要面对高并发用户，大流量访问。Google 日均 PV 35 亿，日 IP 访问数 3 亿；腾讯 QQ 的最大在线用户数 1.4 亿（2011年数据）。

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，需要使用大量服务器。Facebook 每周上传的照片数量接近 10 亿，百度收录的网页数目有数百亿，Google 有近百万台服务器为全球用户提供服务。

用户分布广泛，网络情况复杂

许多大型互联网站都是为全球用户提供服务的，用户分布范围广，各地网络情况千差万别。在国内，还有各个运营商网络互通难的问题。

安全环境恶劣

由于互联网的开放性，使得互联网站更容易受到攻击，大型网站几乎每天都会被黑客攻击。

需求快速变更，发布频繁

和传统软件的版本发布频率不同，互联网产品为快速适应市场，满足用户需求，其产品发布频率极高。一般大型网站的产品每周都有新版本发布上线，中小型网站的发布更频繁，有时候一天会发布几十次。

渐进式发展

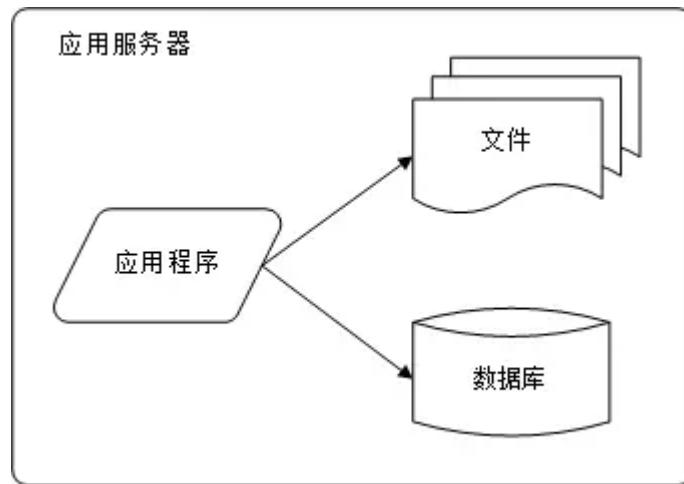
几乎所有的大型互联网站都是从一个小网站开始，渐进地发展起来的。Facebook 是扎克伯格同学在哈佛大学的宿舍里开发的；Google 的第一台服务器部署在斯坦福大学的实验室；阿里巴巴是在马云家的客厅诞生的。好的互联网产品都是慢慢运营出来的，不是一开始就开发好的，这也正好与网站架构的发展演化过程对应。

二、大型网站架构演化发展历程

大型网站的技术挑战主要来自于庞大的用户，高并发的访问和海量的数据，任何简单的业务一旦需要处理数以 P 计的数据和面对数以亿计的用户，问题就会变得很棘手。大型网站架构主要解决这类问题。

初始阶段的网站架构

大型网站都是从小型网站发展而来，网站架构也是一样，是从小型网站架构逐步演化而来。小型网站最开始没有太多人访问，只需要一台服务器就绰绰有余，这时的网站架构如下图所示：



应用程序、数据库、文件等所有资源都在一台服务器上。

应用服务和数据服务分离

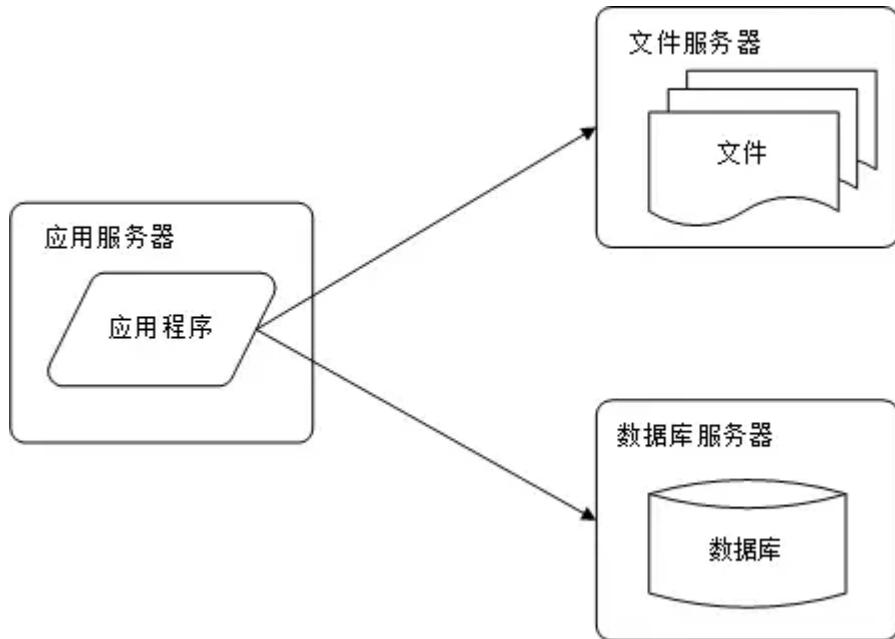
随着网站业务的发展，一台服务器逐渐不能满足需求：越来越多的用户访问导致性能越来越差，越来越多的数据导致存储空间不足。这时就需要将应用和数据分离。应用和数据分离后整个网站使用3台服务器：应用服务器、文件服务器和数据库服务器。这 3 台服务器对硬件资源的要求各不相同：

应用服务器需要处理大量的业务逻辑，因此需要更快更强大的CPU；

数据库服务器需要快速磁盘检索和数据缓存，因此需要更快的磁盘和更大的内存；

文件服务器需要存储大量用户上传的文件，因此需要更大的硬盘。

此时，网站系统的架构如下图所示：



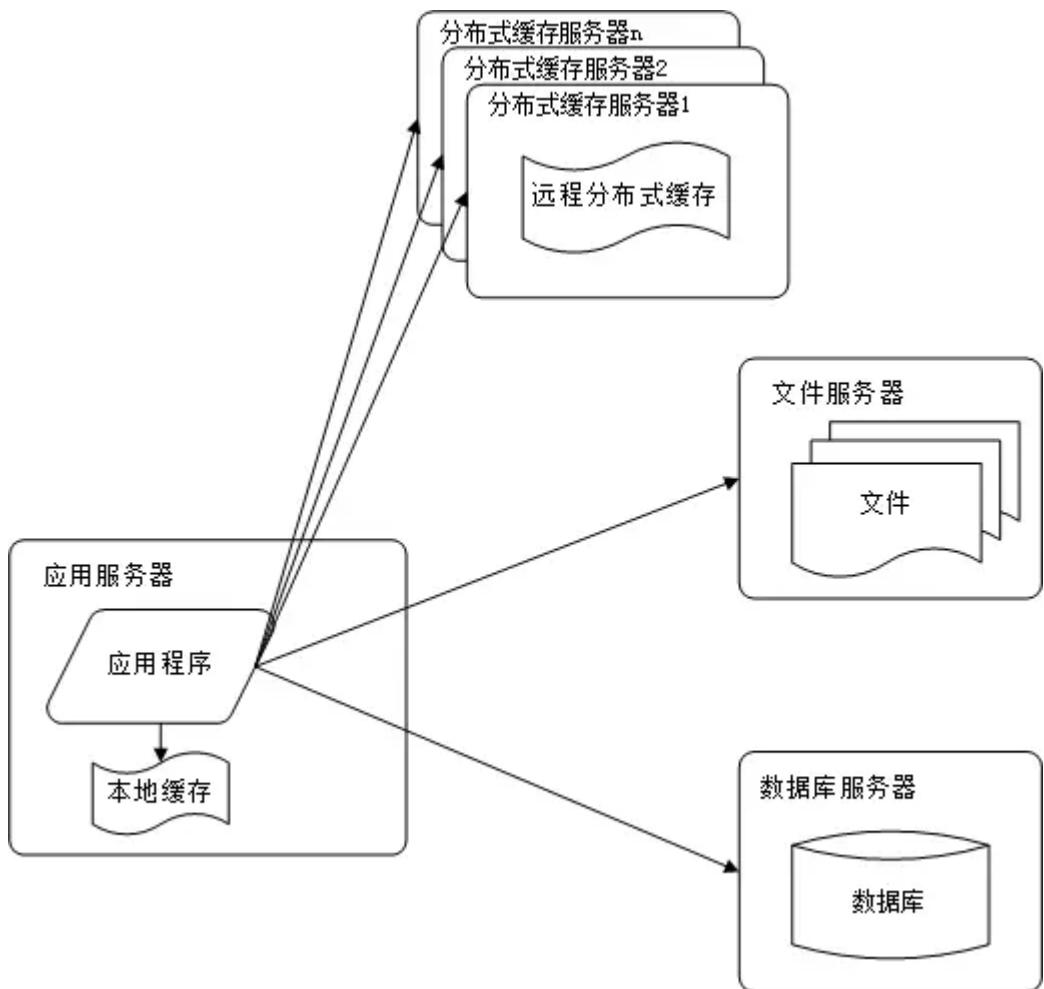
应用和数据分离后，不同特性的服务器承担不同的服务角色，网站的并发处理能力和数据存储空间得到了很大改善，支持网站业务进一步发展。但是随着用户逐渐增多，网站又一次面临挑战：数据库压力太大导致访问延迟，进而影响整个网站的性能，用户体验受到影响。这时需要对网站架构进一步优化。

使用缓存改善网站性能

网站访问的特点和现实世界的财富分配一样遵循二八定律：80% 的业务访问集中在20% 的数据上。既然大部分业务访问集中在一小部分数据上，那么如果把这一小部分数据缓存在内存中，就可以减少数据库的访问压力，提高整个网站的数据访问速度，改善数据库的写入性能了。网站使用的缓存可以分为两种：缓存在应用服务器上的本地缓存和缓存在专门的分布式缓存服务器上的远程缓存。

本地缓存的访问速度更快一些，但是受应用服务器内存限制，其缓存数据量有限，而且会出现和应用程序争用内存的情况。

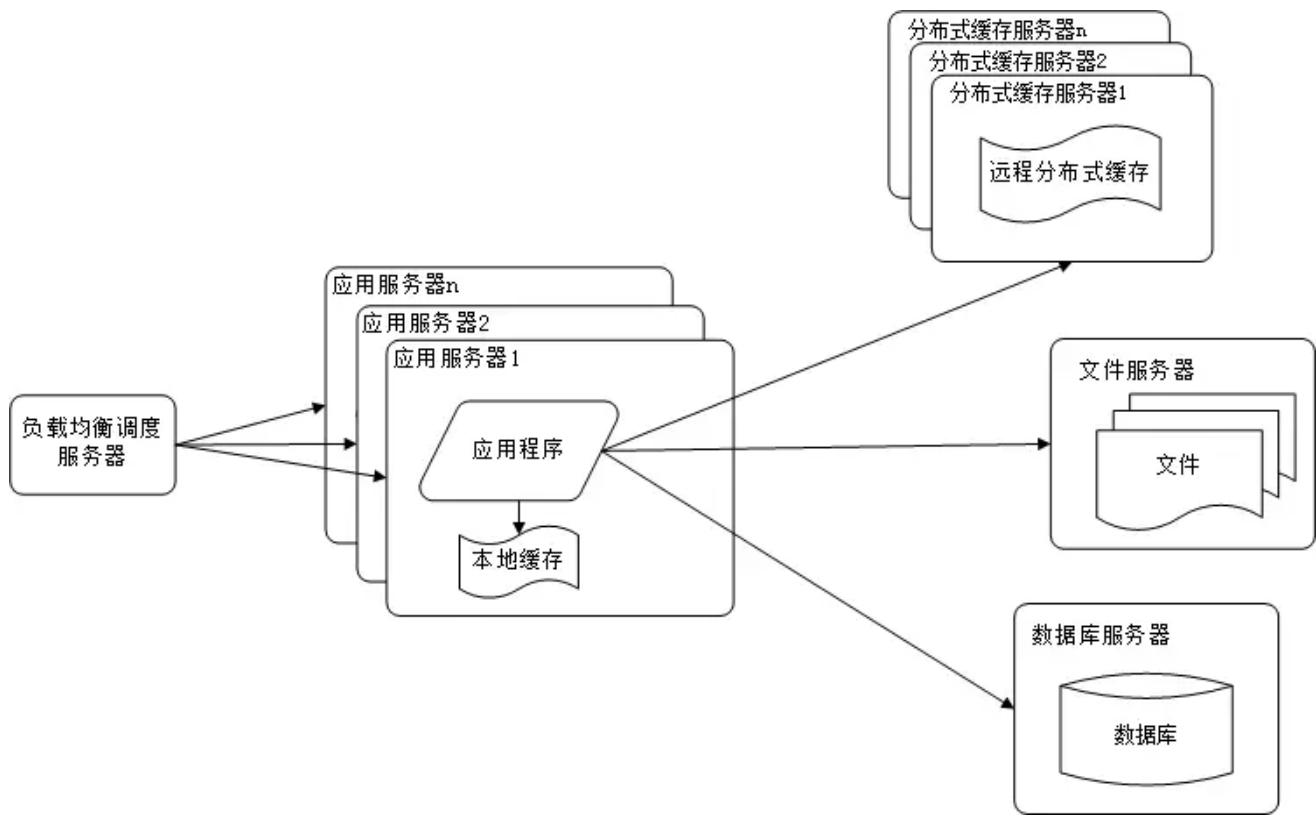
远程分布式缓存可以使用集群的方式，部署大内存的服务器作为专门的缓存服务器，可以在理论上做到不受内存容量限制的缓存服务。



使用缓存后，数据访问压力得到有效缓解，但是单一应用服务器能够处理的请求连接有限，在网站访问高峰期，应用服务器成为整个网站的瓶颈。

使用应用服务器集群改善网站的并发处理能力

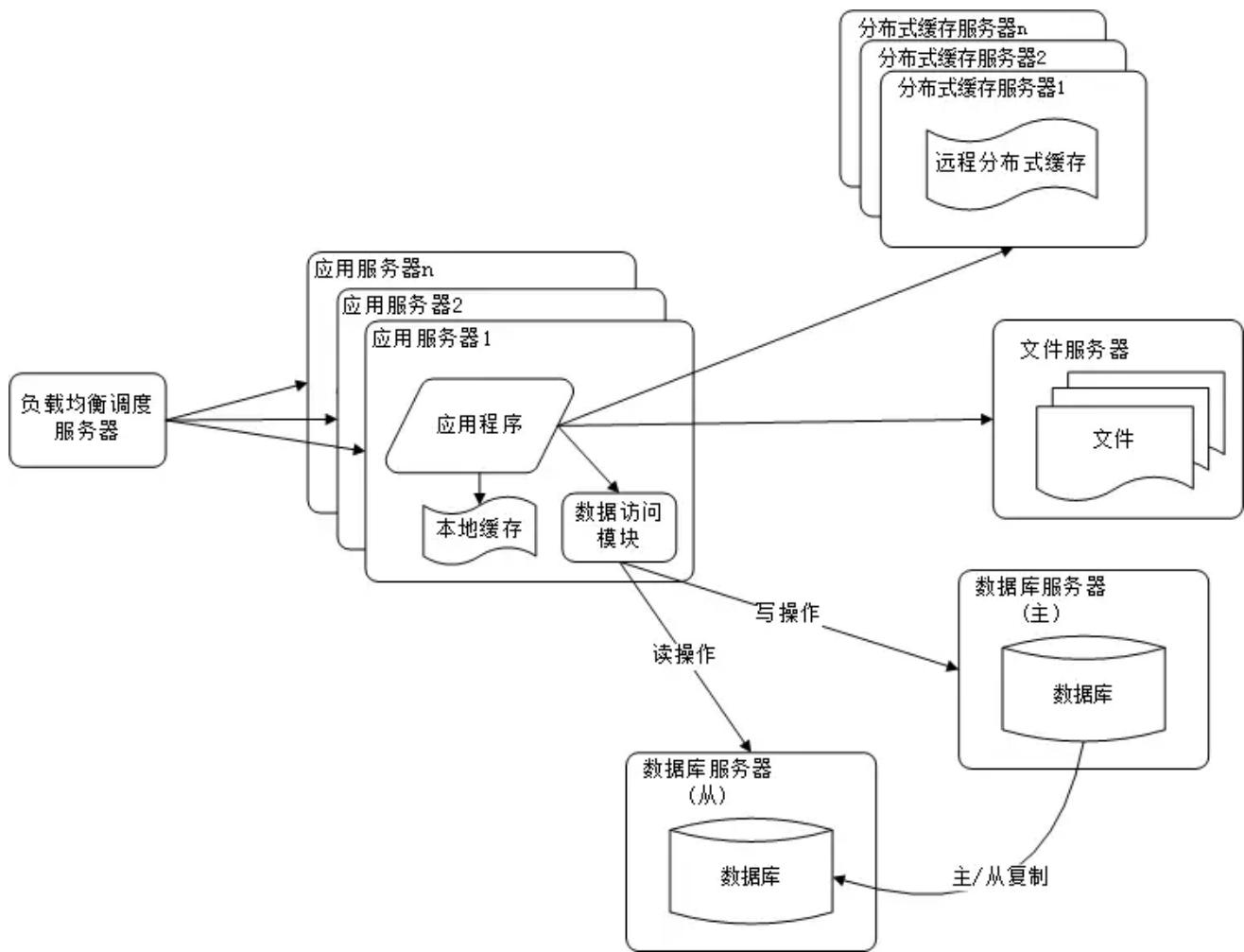
使用集群是网站解决高并发、海量数据问题的常用手段。当一台服务器的处理能力、存储空间不足时，不要企图去更换更强大的服务器，对大型网站而言，不管多么强大的服务器，都满足不了网站持续增长的业务需求。这种情况下，更恰当的做法是增加一台服务器分担原有服务器的访问及存储压力。**对网站架构而言，只要能通过增加一台服务器的方式改善负载压力，就可以以同样的方式持续增加服务器不断改善系统性能，从而实现系统的可伸缩性。**应用服务器实现集群是网站可伸缩架构设计中较为简单成熟的一种，如下图所示：



通过负载均衡调度服务器，可以将来自用户浏览器的访问请求分发到应用服务器集群中的任何一台服务器上，如果有更多用户，就在集群中加入更多的应用服务器，使应用服务器的压力不再成为整个网站的瓶颈。

数据库读写分离

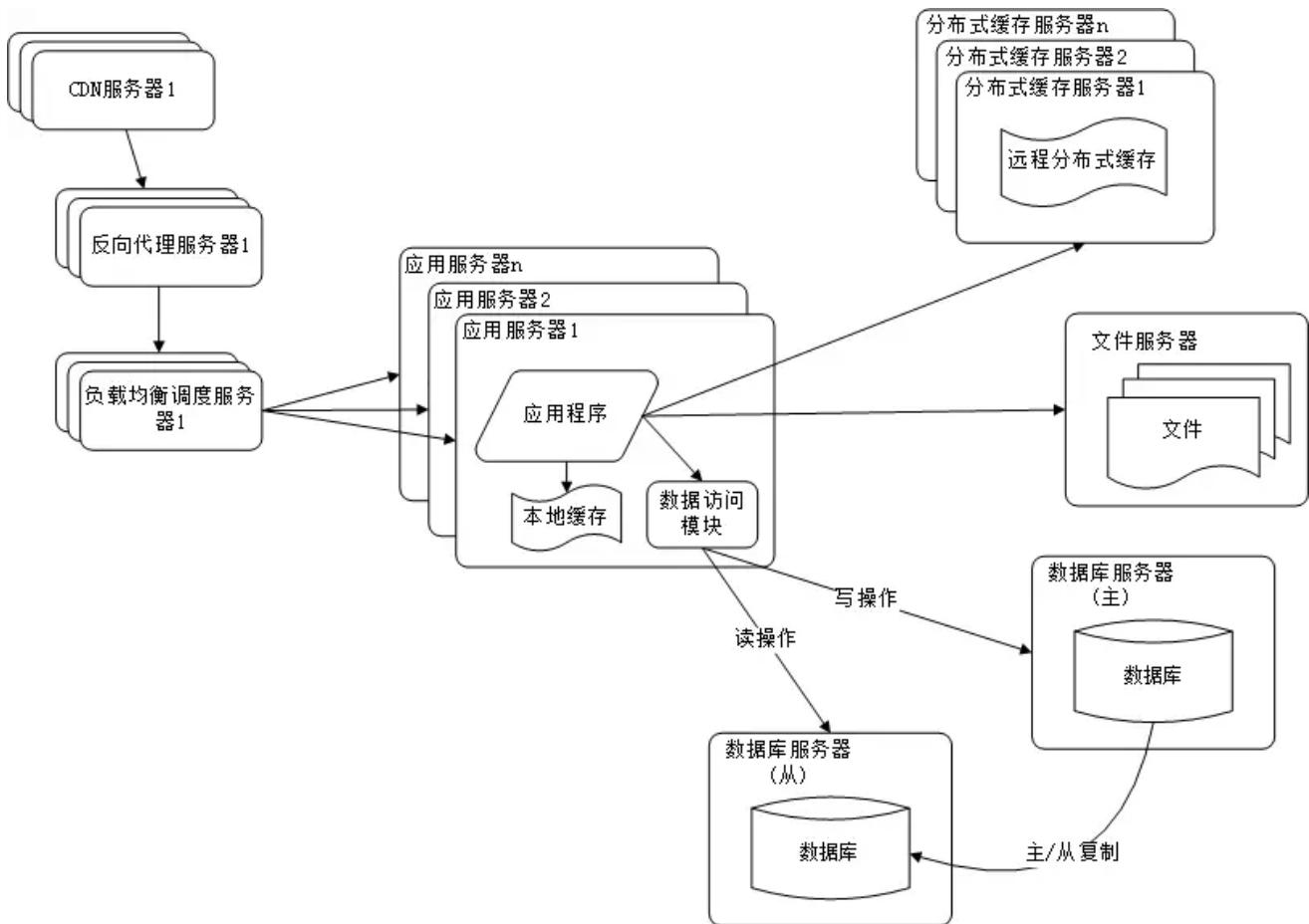
网站在使用缓存后，使对大部分数据读操作访问都可以不通过数据库就能完成，但是仍有一部分读操作（缓存访问不命中、缓存过期）和全部的写操作都需要访问数据库，在网站的用户达到一定规模后，数据库因为负载压力过高而成为网站的瓶颈。目前大部分的主流数据库都提供主从热备功能，通过配置两台数据库主从关系，可以将一台数据库服务器的数据更新同步到另一台服务器上。网站利用数据库的这一功能，实现数据库读写分离，从而改善数据库负载压力。如下图所示：



应用服务器在写数据的时候，访问主数据库，主数据库通过主从复制机制将数据更新同步到从数据库，这样当应用服务器读数据的时候，就可以通过从数据库获得数据。为了便于应用程序访问读写分离后的数据库，通常在应用服务器端使用专门的数据访问模块，使数据库读写分离对应用透明。

使用反向代理和 CDN 加速网站响应

随着网站业务不断发展，用户规模越来越大，由于中国复杂的网络环境，不同地区的用户访问网站时，速度差别也极大。有研究表明，网站访问延迟和用户流失率正相关，网站访问越慢，用户越容易失去耐心而离开。为了提供更好的用户体验，留住用户，网站需要加速网站访问速度。主要手段有使用 CDN 和方向代理。如下图所示：



CDN 和反向代理的基本原理都是缓存。

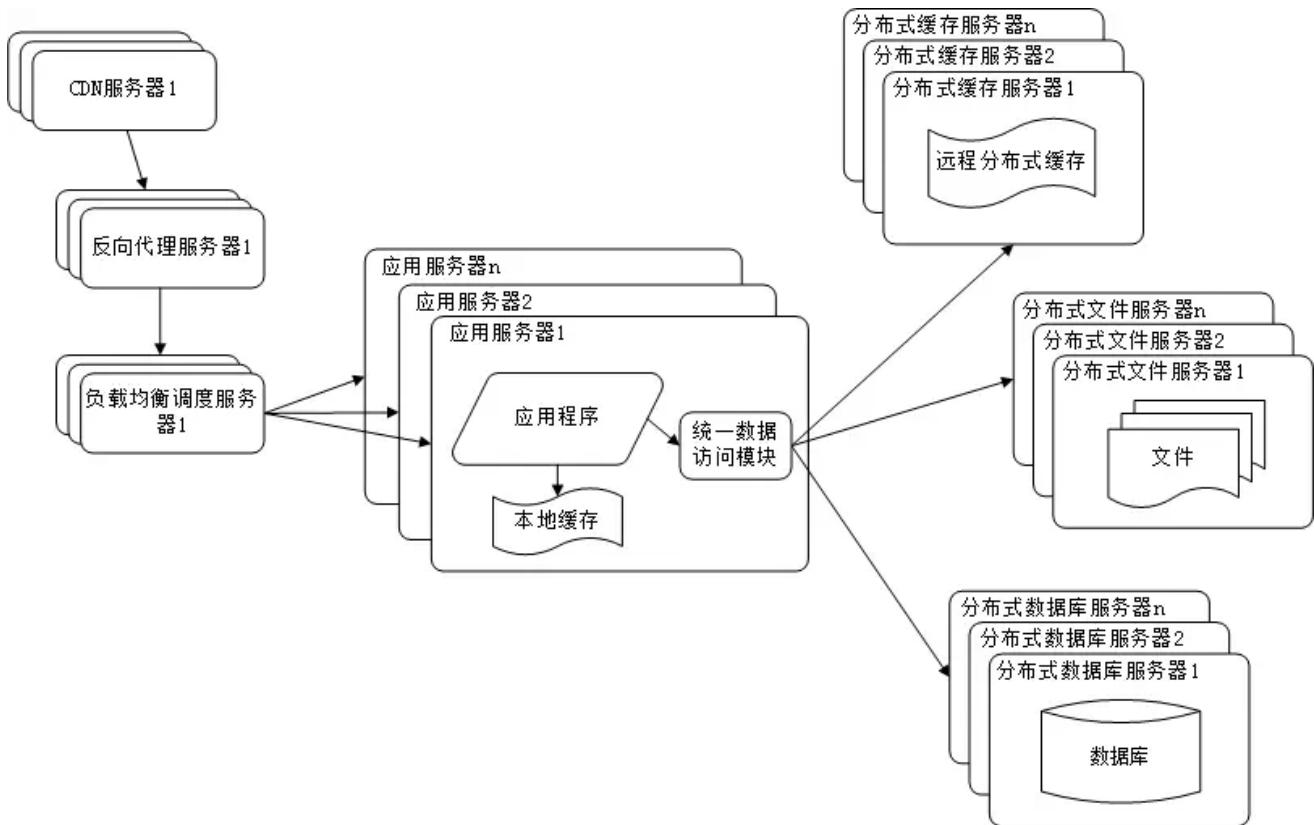
CDN 部署在网络提供商的机房，使用户在请求网站服务时，可以从距离自己最近的网络提供商机房获取数据

反向代理则部署在网站的中心机房，当用户请求到达中心机房后，首先访问的服务器是反向代理服务器，如果反向代理服务器中缓存着用户请求的资源，就将其直接返回给用户

使用 CDN 和反向代理的目的都是尽早返回数据给用户，一方面加快用户访问速度，另一方面也减轻后端服务器的负载压力。

使用分布式文件系统和分布式数据库系统

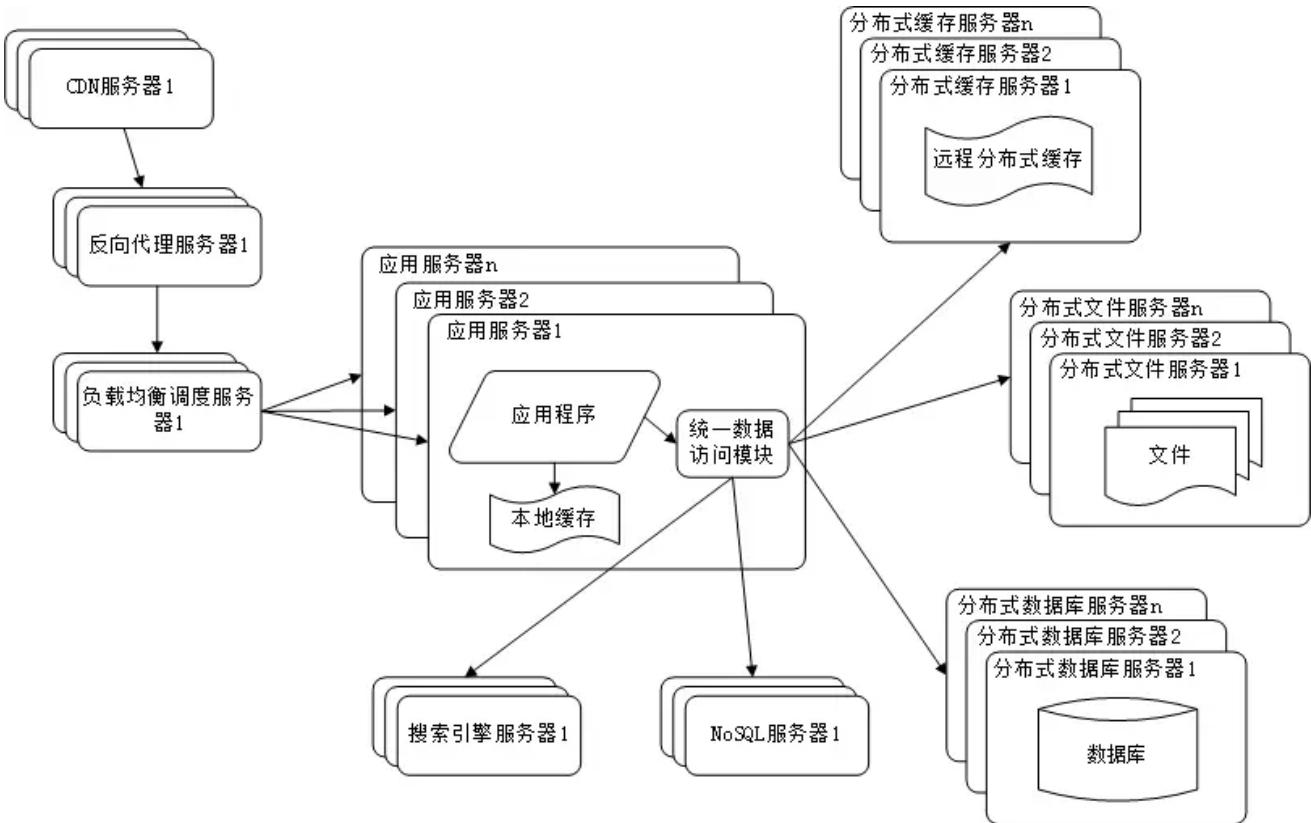
任何强大的单一服务器都满足不了大型网站持续增长的业务需求。数据库经过读写分离后，从一台服务器拆分成两台服务器，但是随着网站业务的发展依然不能满足需求，这时需要使用分布式数据库。文件系统也一样，需要使用分布式文件系统。如下图所示：



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非常庞大的时候才使用。不得已时，网站更常用的数据库拆分手段是业务分库，将不同业务的数据部署在不同的物理服务器上。

使用 NoSQL 和搜索引擎

随着网站业务越来越复杂，对数据存储和检索的需求也越来越复杂，网站需要采用一些非关系数据库技术如 NoSQL 和非数据库查询技术如搜索引擎。如下图所示：

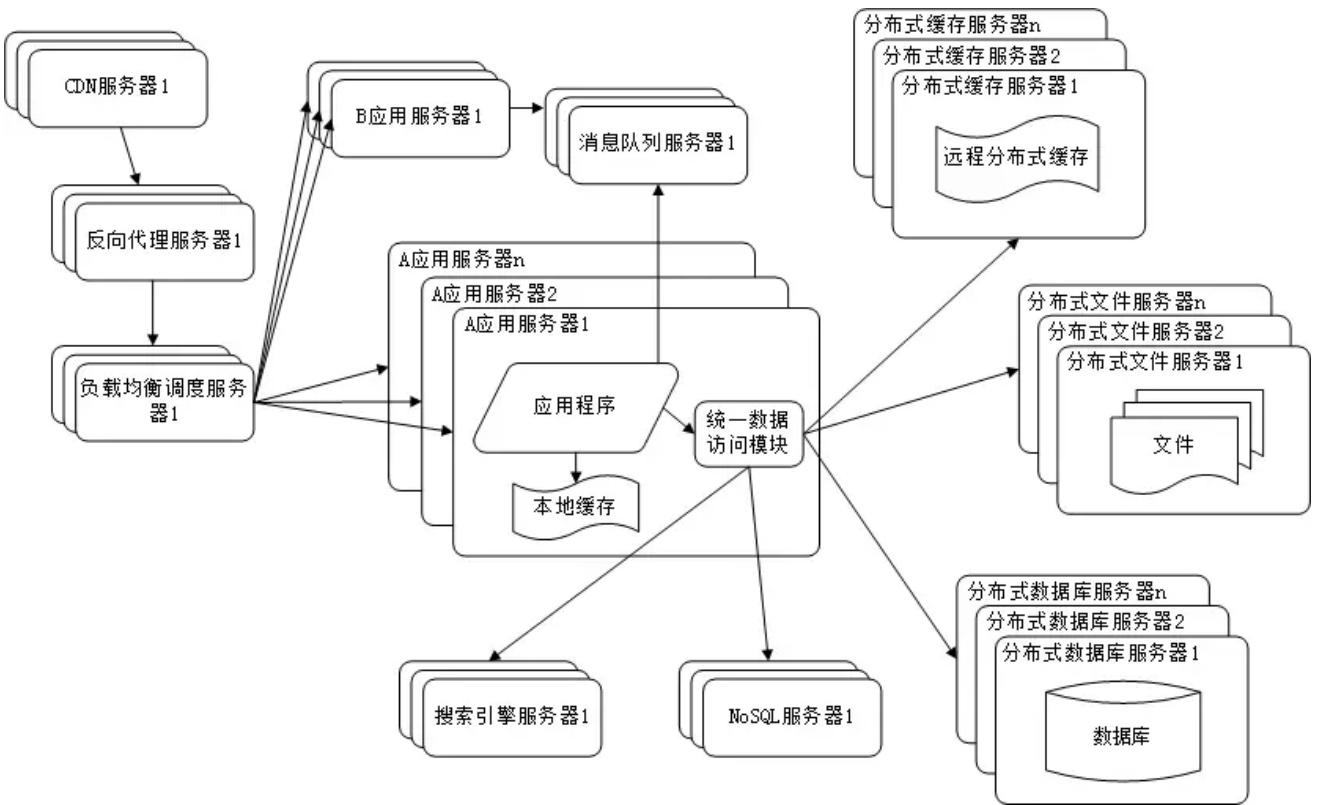


NoSQL 和搜索引擎都是源自互联网的技术手段，对可伸缩的分布式特性具有更好的支持。应用服务器则通过一个统一数据访问模块访问各种数据，减轻应用程序管理诸多数据源的麻烦。

业务拆分

大型网站为了应对日益复杂的业务场景，通过使用分而治之的手段将整个网站业务分成不同的产品线。如大型购物交易网站都会将首页、商铺、订单、买家、卖家等拆分成不同的产品线，分归不同的业务团队负责。

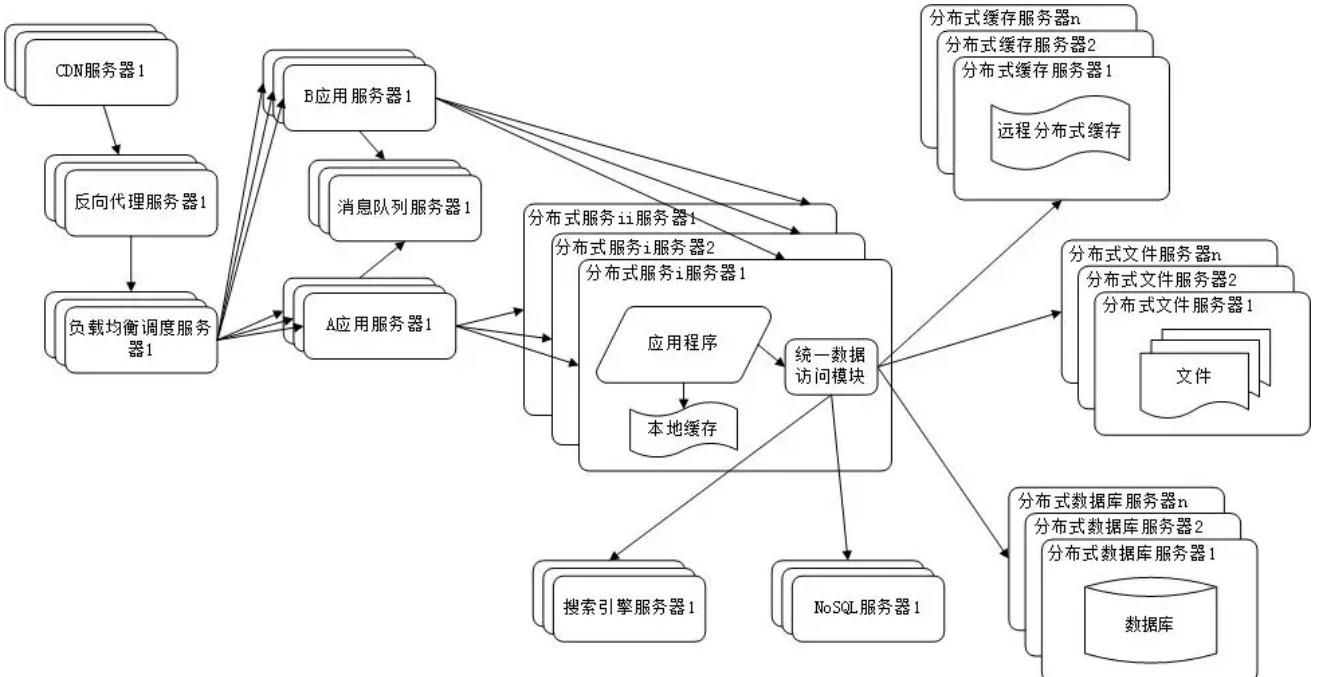
具体到技术上，也会根据产品线划分，将一个网站拆分成许多不同的应用，每个应用独立部署。应用之间可以通过一个超链接建立关系（在首页上的导航链接每个都指向不同的应用地址），也可以通过消息队列进行数据分发，当然最多的还是通过访问同一个数据存储系统来构成一个关联的完整系统，如下图所示：



分布式微服务

随着业务拆分越来越小，存储系统越来越庞大，应用系统的整体复杂度呈指数级增加，部署维护越来越困难。由于所有应用要和所有数据库系统连接，在数万台服务器规模的网站中，这些连接的数目是服务器规模的平方，导致数据库连接资源不足，拒绝服务。

既然每一个应用系统都需要执行许多相同的业务操作，比如用户管理、商品管理等，那么可以将这些共用的业务提取出来，独立部署。由这些可复用的业务连接数据库，提供共用业务服务，而应用系统只需要管理用户界面，通过分布式服务调用共用业务服务完成具体业务操作。如下图所示：



三、拆分 VS 集群

1. 拆分：不同的**多台服务器上面部署不同的服务模块**，模块之间通过RPC通信和调用，用于拆分业务功能，独立部署，多个服务器共同组成一个整体对外提供服务。
2. 集群：不同的**多台服务器上面部署相同的服务模块**，通过分布式调度软件进行统一的调度，用于分流容灾，降低单个服务器的访问压力。

四、微服务 VS SOA

创始人：**martin fowler** <https://martinfowler.com/articles/microservices.html>

单体应用：ALL IN ONE

微服务是一种架构风格，一个大型复杂软件应用由一个或多个微服务组成。系统中的各个微服务可被独立部署，各个微服务之间是松耦合的。每个微服务仅关注于完成一件任务并很好地完成该任务。在所有情况下，每个任务代表着一个小的业务能力

微服务，从本质意义上讲，还是 SOA 架构。但内涵有所不同，微服务并不绑定某种特殊的技术，在一个微服务的系统中，可以有 Java 编写的服务，也可以有 Python 编写的服务，他们是靠 Restful 架构风格统一成一个系统的。所以微服务本身与具体技术实现无关，扩展性强。

五、前后端完全分离与Rest规范

http是目前在互联网上使用最多的协议，没有之一。可是http的创始人一直都觉得，在过去10几年来，所有的人都在错误的使用Http。

这句话怎么说呢？如果说你要删除一个数据，以往的做法通常是 delete/{id}，如果你要更新一个数据，可能是Post 数据放Body，然后方法是 update/{id}，或者是article/{id}?method=update。

这种做法让我很暴躁，我觉得这个世界不该这样的，所有的人都在误解而且在严重错误的误解Http的设计初衷，好像是发明了火药却只用它来做烟花爆竹。

那么正确的使用方式是什么呢？如果你要看Rest各种特性，你恐怕真的很难理解Rest，但是如果你看错误的使用http的人到底儿了哪些错，什么是Rest就特别容易理解了。

第一条，混乱。一万个人心里有一万个Url的命名规则，Url是统一资源定位符，重点是资源。而很多人却把它当成了万金油，每一个独立的虚拟的网页都可以随意使用，各种操作都能够迭加。这是混乱的来源之一。

第二条，贪婪。有状态和无状态全部混在一起。特别是在购物车或者是登录的应用中，经常刷新就丢失带来的用户体验简直棒棒哒。每一个请求并不能单独的响应一些功能，很多的功能混杂在一起。这是人性贪婪的本质，也是各种Hack的起源，只要能够把问题解决掉，总会有人用他认为最方便的方式去解决问题，比如说汽车门把手坏掉了直接系根绳子当把手，emmm这样确实很棒啊。

第三条，无序。返回的结果往往是很随意，各种错误信息本来就是用Http的状态码构成的，可是很多人还是喜欢把错误信息返回在返回值中。最常见的就是Code和Message，当然对于这一点，我个人是保留疑问的，我的观点是，Http本身错误和服务器的内部错误还是需要在不断层面分开的，不能混在一起。可是在大神眼里并非如此，这个再议。

好了我编不下去了。那么怎么解决这些问题呢？强迫症患者的福音就是先颁布规则，第一个规则就是明确Url是什么，该怎么用。就是所有的Url本质来讲，都应该是一种资源。一个独立的Url地址，就是对应一个独一无二的资源。怎么样？这种感觉是不是棒棒哒？一个冰淇淋，一个老师，一间房子，在Url上对应的都是一个资源，不会有太多的Url跟他对应，也不会表示有多个Url地址～注意，这里点的是Url地址，并不是单独的参数，他就是一个/room/{room_id}这样的东西，举个栗子，/room/3242 这就表示3242号房间。

这是一个清爽的世界啊，你想想，之前的Url是什么都要，我开房，可能是/open/room/3242 我要退房可能是/exit/3242/room，我要打理房间，可能是room/3242?method=clean.

够了！这些乱七八糟的东西全够了，让世界回归清爽的本质，一间房，就是/room/3242 没有别的Url地址了。那我想要对这个资源有操作怎么办？这就是棒棒哒大神想出来的了，http有几种Method来着？get, put, post, delete，还有其他隐藏的4种。在过去的混乱世界里，经常用的就是Get和Post。如果不是因为Get不支持大数据传输，我想连Post都会有人使用。（想像一下Roy Fielding在愤怒的对着电脑屏幕喊，Http的Method一共有八个，你们为毛只逮着Get一只羊的毛薅薅薅薅薅）。

而对资源最常见的操作是什么？CRUD，对不对，就是创建，读，更新，删除。再看Http的Method？是不是非常完美？其实也怪Fielding老爷子一开始命名不准确，如果刚开始就是把Get方法叫做Read，Put方法叫做Update，Post叫做Create这该多好。。。

你用一个Get，大家又发现没什么限制没什么所谓，又很难理解Put和Post的差别，法无禁止即可为啊，呃，老爷子不要瞪我，我瞎说的。

总之，这四种方法够不够你浪？你有本身找出来更多的对资源的操作来啊，我还有4个Method没用过呢。如果这4个真的不够了，有什么问题，大不了我再重新更改http协议啊。

其实简单说，对于Rest理解到这里就够了。后续的东西，都是在这一条基础上空想出来的，比强迫症更强强迫症，当然，无状态我是百分百支持的。以上的各种表述可能不太准确，也纯属是我的意淫和各种小道资料，并未考据，但是凭良心讲，我是早就看不惯黑暗年代里的Url命名风格了，所以当时最早接触到Rest的时候，瞬间就找到了真爱，我靠，这不就是我一直想要的答案吗？

但是我一直想的仅仅是命名规范，从来没有把自己的思考角度放在一个url就是一个资源，所有的操作都是对资源的更改而言的角度上啊。所以你能理解到的程度，更多的就是在乎你要弄清楚你要解决的什么问题，如果你的问题只是理解Rest，恐怕你很理解，如果你的问题是怎么解决Url混乱的问题，你反而很快能弄懂了～

Rest操作最佳实践：现在在很多企业中，虽然都在支持Rest规范，但是真正严格遵守的几乎没有，因为按照Rest规范，删除需要发送Delete请求，插入需要发送PUT请求，过于繁琐，并且有些框架，例如ajax, Springmvc等，对Delete和PUT请求的支持不太友好，所以实际应用中很少使用这两种请求，一般还是只是用Get和Post请求，使用接口名字来区分，所以，对于Rest规范，只需要记得传递数据只使用JSON，而不是后端去渲染模板，从而实现前后端的完全分离。

六、CAP三进二和Base定理

关系型数据库遵循ACID规则

事务在英文中是transaction，和现实世界中的交易很类似，它有如下四个特性：

1、A (Atomicity) 原子性 原子性很容易理解，也就是说事务里的所有操作要么全部做完，要么都不做，事务成功的条件是事务里的所有操作都成功，只要有一个操作失败，整个事务就失败，需要回滚。比如银行转账，从A账户转100元至B账户，分为两个步骤：1) 从A账户取100元；2) 存入100元至B账户。这两步要么一起完成，要么一起不完成，如果只完成第一步，第二步失败，钱会莫名其妙少了100元。

2、C (Consistency) 一致性 一致性也比较容易理解，也就是说数据库要一直处于一致的状态，事务的运行不会改变数据库原本的一致性约束。

3、I (Isolation) 独立性 所谓的独立性是指并发的事务之间不会互相影响，如果一个事务要访问的数据正在被另外一个事务修改，只要另外一个事务未提交，它所访问的数据就不受未提交事务的影响。比如现有有个交易是从A账户转100元至B账户，在这个交易还未完成的情况下，如果此时B查询自己的账户，是看不到新增加的100元的

4、D (Durability) 持久性 持久性是指一旦事务提交后，它所做的修改将会永久的保存在数据库上，即使出现宕机也不会丢失。

CAP三进二

在分布式系统中，讲究C:Consistency（强一致性）、A:Availability（可用性）、P:Partition tolerance（分区容错性）

CAP的证明基于异步网络，异步网络也是反映了真实网络中情况的模型。真实的网络系统中，节点之间不可能保持同步，即便是时钟也不可能保持同步，所有的节点依靠获得的消息来进行本地计算和通讯。这个概念其实是相当强的，意味着任何超时判断也是不可能的，因为没有共同的时间标准。之后我们会扩展CAP的证明到弱一点的异步网络中，这个网络中时钟不完全一致，但是时钟运行的步调是一致的，这种系统是允许节点做超时判断的。

CAP的证明很简单，假设两个节点集{G1, G2}，由于网络分片导致G1和G2之间所有的通讯都断开了，如果不满足P，则整个网络不可用，如果在G1中写，在G2中读刚写的数据，G2中返回的值不可能是G1中的写值。由于A的要求，G2一定要返回这次读请求，由于P的存在，导致C一定是不可满足的。

CAP理论就是说在分布式存储系统中，最多只能实现上面的两点。而由于当前的网络硬件肯定会出现延迟丢包等问题，所以

分区容忍性是我们必须需要实现的。

所以我们只能在一致性和可用性之间进行权衡，没有任何分布式系统能同时保证这三点。

C:强一致性 A: 高可用性 P: 分布式容忍性 CA 传统Oracle数据库

AP 大多数网站架构的选择

CP Redis、Mongodb

注意：分布式架构的时候必须做出取舍。一致性和可用性之间取一个平衡。多余大多数web应用，其实并不需要强一致性。

因此牺牲C换取P，这是目前分布式数据库产品的方向

一致性与可用性的抉择

对于web2.0网站来说，关系数据库的很多主要特性却往往无用武之地

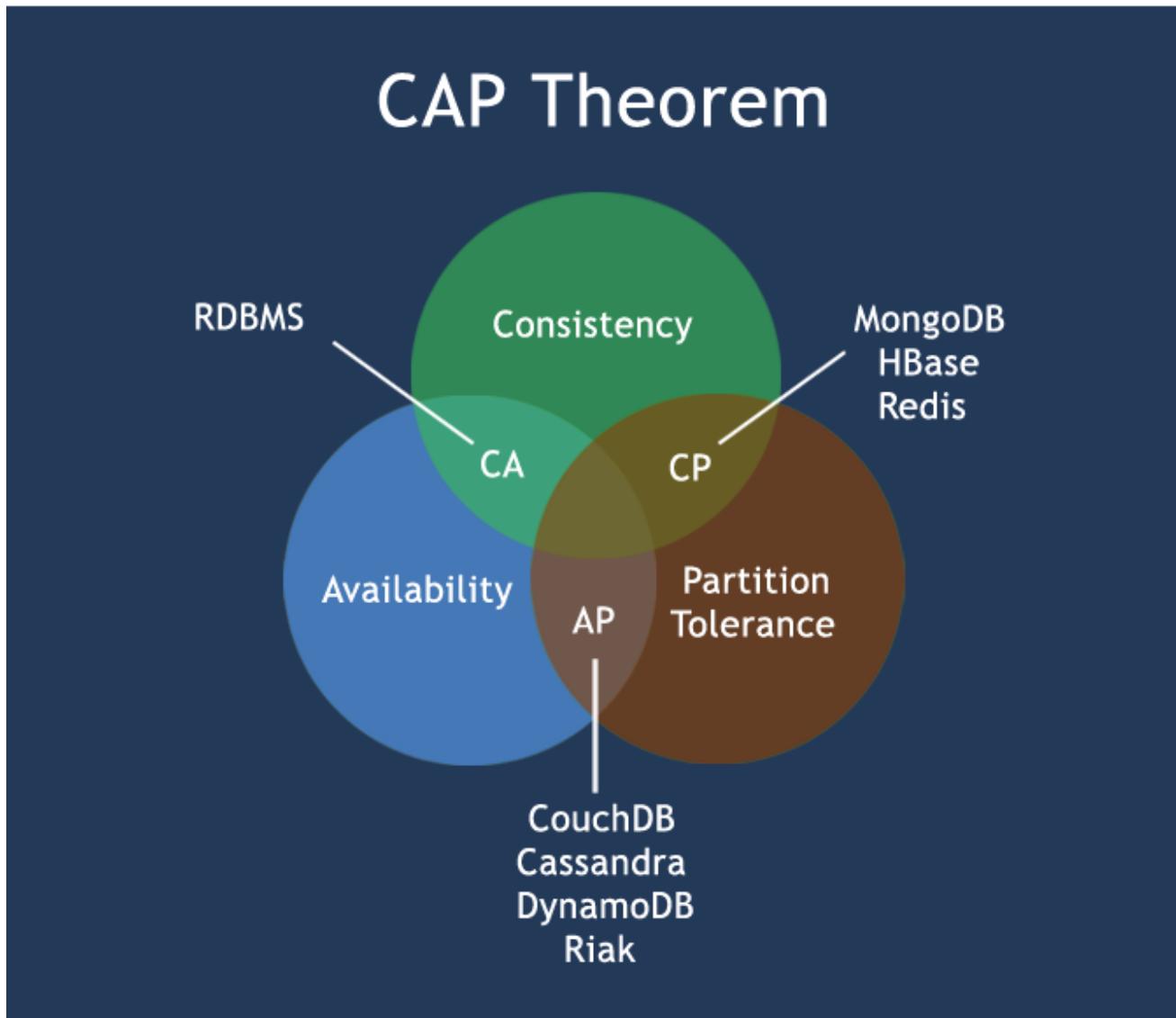
数据库事务一致性需求 很多web实时系统并不要求严格的数据库事务，对读一致性的要求很低，有些场合对写一致性要求并不高。允许实现最终一致性。

数据库的写实时性和读实时性需求 对关系数据库来说，插入一条数据之后立刻查询，是肯定可以读出来这条数据的，但是对于很多web应用来说，并不要求这么高的实时性，比方说发一条消息之后，过几秒乃至十几秒之后，我的订阅者才看到这条动态是完全可以接受的。

对复杂的SQL查询，特别是多表关联查询的需求 任何大数据量的web系统，都非常忌讳多个大表的关联查询，以及复杂的数据分析类型的报表查询，特别是SNS类型的网站，从需求以及产品设计角度，就避免了这种情况的产生。往往更多的只是单表的主键查询，以及单表的简单条件分页查询，SQL的功能被极大的弱化了。

CAP理论的核心是：一个分布式系统不可能同时很好的满足一致性，可用性和分区容错性这三个需求，最多只能同时较好的满足两个。因此，根据 CAP 原理将 NoSQL 数据库分成了满足 CA 原则、满足 CP 原则和满足 AP 原则三大类：

- CA - 单点集群，满足一致性，可用性的系统，通常在可扩展性上不太强大。
- CP - 满足一致性，分区容忍的系统，通常性能不是特别高。
- AP - 满足可用性，分区容忍性的系统，通常可能对一致性要求低一些。



BASE定理

BASE就是为了解决关系数据库强一致性引起的问题而引起的可用性降低而提出的解决方案。

BASE其实是下面三个术语的缩写：

- 基本可用 (Basically Available)
- 软状态 (Soft state)

- 最终一致 (Eventually consistent)

它的思想是通过让系统放松对某一时刻数据一致性的要求来换取系统整体伸缩性和性能上改观。为什么这么说呢，缘由就在于大型系统往往由于地域分布和极高性能的要求，不可能采用分布式事务来完成这些指标，要想获得这些指标，我们必须采用另外一种方式来完成，这里BASE就是解决这个问题的办法

分布式一致性理论paxos、raft、zab算法

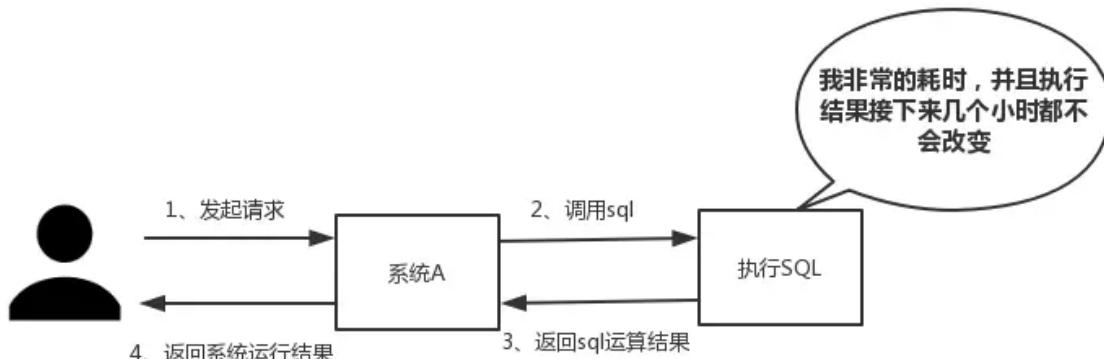
演示 Raft <http://thesecretlivesofdata.com/raft/>

中间件

一、缓存

为什么要使用缓存

(一) 性能 如下图所示，我们在碰到需要执行耗时特别久，且结果不频繁变动的SQL，就特别适合将运行结果放入缓存。这样，后面的请求就去缓存中读取，使得请求能够迅速响应。



题外话：忽然想聊一下这个**迅速响应**的标准。其实根据交互效果的不同，这个响应时间没有固定标准。不过曾经有人这么告诉我：“在理想状态下，我们的页面跳转需要在**瞬间**解决，对于页内操作则需要在**刹那**间解决。另外，超过**一弹指**的耗时操作要有进度提示，并且可以随时中止或取消，这样才能给用户最好的体验。”

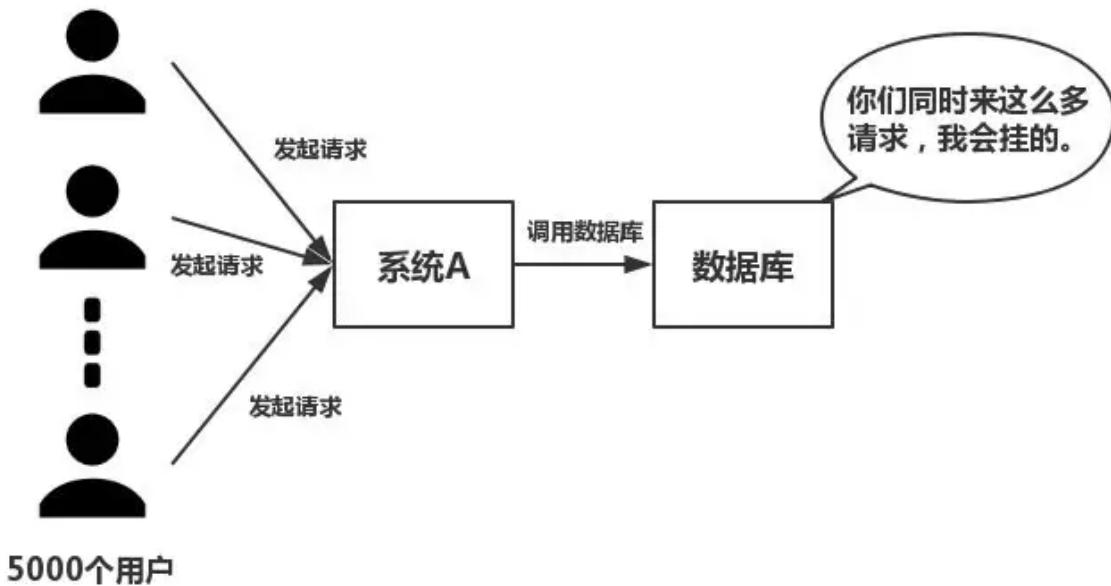
那么**瞬间**、**刹那**、**一弹指**具体是多少时间呢？

根据《摩诃僧祇律》记载

一刹那者为一念，二十念为一瞬，二十瞬为一弹指，二十弹指为一罗预，二十罗预为一须臾，一日一夜有三十须臾。

那么，经过周密的计算，一瞬间为0.36秒，一刹那有0.018秒，一弹指长达7.2秒。

(二) 并发 如下图所示，在大并发的情况下，所有的请求直接访问数据库，数据库会出现连接异常。这个时候，就需要使用redis做一个缓冲操作，让请求先访问到redis，而不是直接访问数据库。



优秀的缓存系统Redis

Redis是完全开源免费的，用C语言编写的，遵守BSD协议，是一个高性能的(key/value)分布式内存数据库，基于内存运行并支持持久化的NoSQL数据库，是当前最热门的NoSql数据库之一，也被人们称为数据结构服务器。

Redis相比同类的其他产品，具有如下优点：

- Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用
- Redis不仅仅支持简单的key-value类型的数据，同时还提供list, set, zset, hash等数据结构的存储
- Redis支持数据的备份，即master-slave模式的数据备份

redis为什么这么快

主要是以下三点

- 纯内存操作
- 单线程操作，避免了频繁的上下文切换
- 采用了非阻塞I/O多路复用机制

题外话：我们现在要仔细的说一说I/O多路复用机制，因为这个说法实在是太通俗了，通俗到一般人都不懂是什么意思。博主打一个比方：小曲在S城开了一家快递店，负责同城快送服务。小曲因为资金限制，雇佣了一批快递员，然后小曲发现资金不够了，只够买一辆车送快递。

经营方式一 客户每送来一份快递，小曲就让一个快递员盯着，然后快递员开车去送快递。慢慢的小曲就发现了这种经营方式存在下述问题

- 几十个快递员基本上时间都花在了抢车上了，大部分快递员都处在闲置状态，谁抢到了车，谁就能去送快递
- 随着快递的增多，快递员也越来越多，小曲发现快递店里越来越挤，没办法雇佣新的快递员了
- 快递员之间的协调很花时间

综合上述缺点，小曲痛定思痛，提出了下面的经营方式

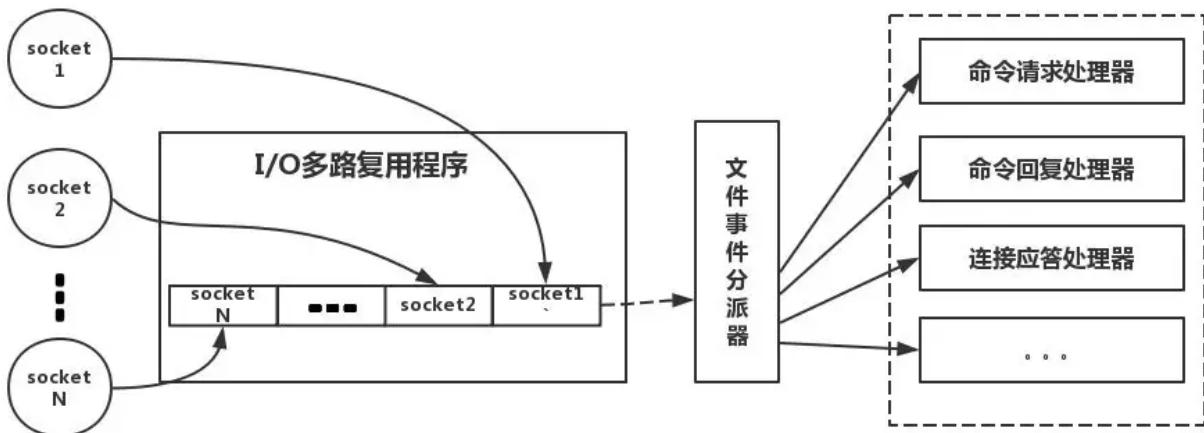
经营方式二 小曲只雇佣一个快递员。然后呢，客户送来的快递，小曲按送达地点标注好，然后依次放在一个地方。最后，那个快递员依次的去取快递，一次拿一个，然后开着车去送快递，送好了就回来拿下一个快递。

对比 上述两种经营方式对比，是不是明显觉得第二种，效率更高，更好呢。在上述比喻中：

- 每个快递员----->每个线程
- 每个快递----->每个socket(I/O流)
- 快递的送达地点----->socket的不同状态
- 客户送快递请求----->来自客户端的请求
- 小曲的经营方式----->服务端运行的代码
- 一辆车----->CPU的核数

于是我们有如下结论 1、经营方式一就是传统的并发模型，每个I/O流(快递)都有一个新的线程(快递员)管理。 2、经营方式二就是I/O多路复用。只有单个线程(一个快递员)，通过跟踪每个I/O流的状态(每个快递的送达地点)，来管理多个I/O流。

下面类比到真实的redis线程模型，如图所示



参照上图，简单来说，就是。我们的redis-client在操作的时候，会产生具有不同事件类型的socket。在服务端，有一段I/O多路复用程序，将其置入队列之中。然后，文件事件分派器，依次去队列中取，转发到不同的事件处理器中。需要说明的是，这个I/O多路复用机制，redis还提供了select、epoll、evport、kqueue等多路复用函数库，大家可以自行去了解。

redis的数据类型，以及每种数据类型的使用场景

(一)String 这个其实没啥好说的，最常规的set/get操作，value可以是String也可以是数字。一般做一些复杂的计数功能的缓存。

(二)hash 这里value存放的是结构化的对象，比较方便的就是操作其中的某个字段。博主在做**单点登录**的时候，就是用这种数据结构存储用户信息，以cookield作为key，设置30分钟为缓存过期时间，能很好的模拟出类似session的效果。

(三)list 使用List的数据结构，可以做**简单的消息队列的功能**。另外还有一个就是，可以利用lrange命令，**做基于redis的分页功能**，性能极佳，用户体验好。

(四)set 因为set堆放的是一堆不重复值的集合。所以可以做**全局去重的功能**。为什么不用JVM自带的Set进行去重？因为我们的系统一般都是集群部署，使用JVM自带的Set，比较麻烦，难道为了一个做一个全局去重，再起一个公共服务，太麻烦了。另外，就是利用交集、并集、差集等操作，可以计算**共同喜好，全部的喜好，自己独有的**喜好等**功能**。

(五)sorted set

sorted set多了一个权重参数score,集合中的元素能够按score进行排列。可以做**排行榜应用，取TOP N操作**。另外，参照另一篇《分布式之延时任务方案解析》，该文指出了sorted set可以用来做**延时任务**。最后一个应用就是可以做**范围查找**。

redis的过期策略以及内存淘汰机制

分析:这个问题其实相当重要，到底redis有没用到家，这个问题就可以看出来。比如你redis只能存5G数据，可是你写了10G，那会删5G的数据。怎么删的，这个问题思考过么？还有，你的数据已经设置了过期时间，但是时间到了，内存占用率还是比较高，有思考过原因么？**回答:** redis采用的是定期删除+惰性删除策略。**为什么不用定时删除策略？** 定时删除,用一个定时器来负责监视key,过期则自动删除。虽然内存及时释放，但是十分消耗CPU资源。在大并发请求下，CPU要将时间应用在处理请求，而不是删除key,因此没有采用这一策略。**定期删除+惰性删除是如何工作的呢？** 定期删除，redis默认每个100ms检查，是否有过期的key,有过期key则删除。需要说明的是，redis不是每个100ms将所有的key检查一次，而是随机抽取进行检查(如果每隔100ms,全部key进行检查，redis岂不是卡死)。因此，如果只采用定期删除策略，会导致很多key到时间没有删除。于是，惰性删除派上用场。也就是说在你获取某个key的时候，redis会检查一下，这个key如果设置了过期时间那么是否过期了？如果过期了此时就会删除。**采用定期删除+惰性删除就没其他问题了么？** 不是的，如果定期删除没删除key。然后你也没即时去请求key，也就是说惰性删除也没生效。这样，redis的内存会越来越高。那么就应该采用**内存淘汰机制**。在redis.conf中有一行配置

```
# maxmemory-policy volatile-lru
```

该配置就是配内存淘汰策略的(什么，你没配过？好好反省一下自己) 1) noeviction: 当内存不足以容纳新写入数据时，新写入操作会报错。**应该没人用吧。** 2) allkeys-lru: 当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的key。**推荐使用，目前项目在用这种。** 3) allkeys-random: 当内存不足以容纳新写入数据时，在键空间中，随机移除某个key。**应该也没人用吧，你不删最少使用Key,去随机删。** 4) volatile-lru: 当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的key。**这种情况一般是把redis既当缓存，又做持久化存储的时候才用。不推荐** 5) volatile-random: 当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个key。**依然不推荐** 6) volatile-ttl: 当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的key优先移除。**不推荐** ps: 如果没有设置 expire 的key, 不满足先决条件(prerequisites); 那么 volatile-lru, volatile-random 和 volatile-ttl 策略的行为, 和 noeviction(不删除) 基本上一致。

渐进式ReHash

渐进式rehash的原因

整个rehash过程并不是一步完成的，而是分多次、渐进式的完成。如果哈希表中保存着数量巨大的键值对时，若一次进行rehash，很有可能会导致服务器宕机。

渐进式rehash的步骤

- 为ht[1]分配空间，让字典同时持有ht[0]和ht[1]两个哈希表
- 维持索引计数器变量rehashidx，并将它的值设置为0，表示rehash开始
- 每次对字典执行增删改查时，将ht[0]的rehashidx索引上的所有键值对rehash到ht[1]，将rehashidx值+1。

- 当ht[0]的所有键值对都被rehash到ht[1]中，程序将rehashidx的值设置为-1，表示rehash操作完成

注：渐进式rehash的好处在于它采取分为而治的方式，将rehash键值对的计算均摊到每个字典增删改查操作，避免了集中式rehash的庞大计算量。

缓存穿透

概念访问一个不存在的key，缓存不起作用，请求会穿透到DB，流量大时DB会挂掉。

解决方案：

- 采用布隆过滤器，使用一个足够大的bitmap，用于存储可能访问的key，不存在的key直接被过滤；
- 访问key未在DB查询到值，也将空值写进缓存，但可以设置较短过期时间。

缓存雪崩

大量的key设置了相同的过期时间，导致在缓存在同一时刻全部失效，造成瞬时DB请求量大、压力骤增，引起雪崩。

解决方案

- 可以给缓存设置过期时间时加上一个随机值时间，使得每个key的过期时间分布开来，不会集中在同一时刻失效；
- 采用限流算法，限制流量；
- 采用分布式锁，加锁访问。

二、消息队列

消息队列中间件是分布式系统中重要的组件，主要解决应用耦合，异步消息，流量削峰等问题

实现高性能，高可用，可伸缩和最终一致性架构

使用较多的消息队列有ActiveMQ，RabbitMQ，ZeroMQ，Kafka，MetaMQ，RocketMQ

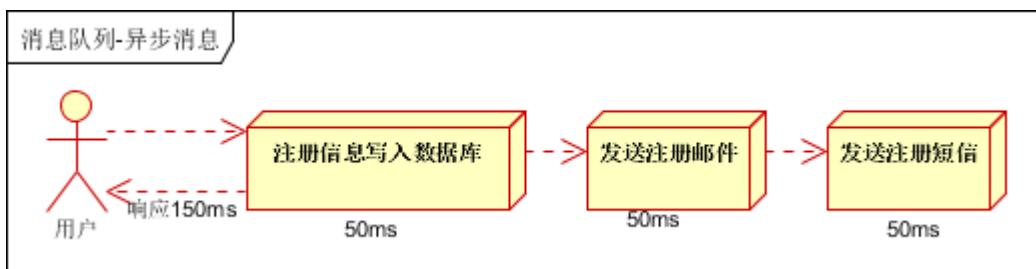
消息队列应用场景

以下介绍消息队列在实际应用中常用的使用场景。异步处理，应用解耦，流量削峰和消息通讯四个场景

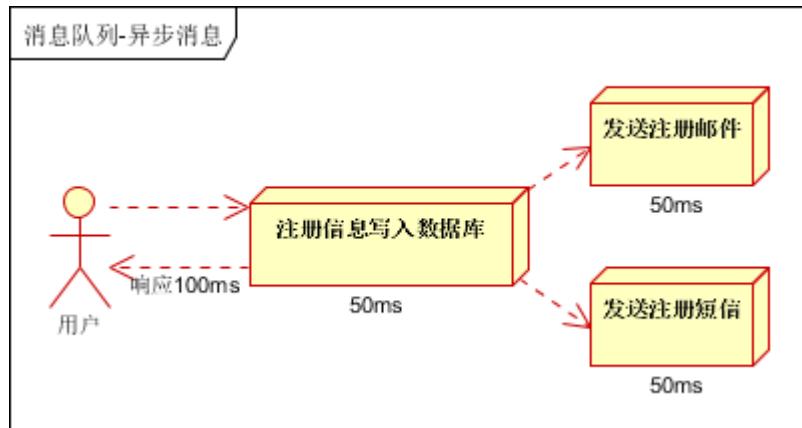
异步处理

场景说明：用户注册后，需要发注册邮件和注册短信。传统的做法有两种 1.串行的方式；2.并行方式

(1) 串行方式：将注册信息写入数据库成功后，发送注册邮件，再发送注册短信。以上三个任务全部完成后，返回给客户端



(2) 并行方式：将注册信息写入数据库成功后，发送注

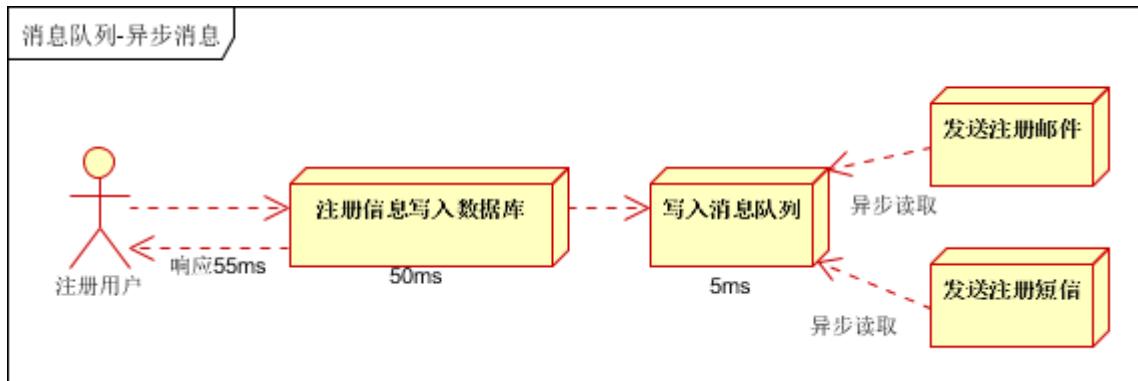


假设三个业务节点每个使用50毫秒，不考虑网络等其他开销，则串行方式的时间是150毫秒，并行的时间可能是100毫秒。

因为CPU在单位时间内处理的请求数是一定的，假设CPU1秒内吞吐量是100次。则串行方式1秒内CPU可处理的请求量是7次 ($1000/150$)。并行方式处理的请求量是10次 ($1000/100$)

小结：如以上案例描述，传统的方式系统的性能（并发量，吞吐量，响应时间）会有瓶颈。如何解决这个问题呢？

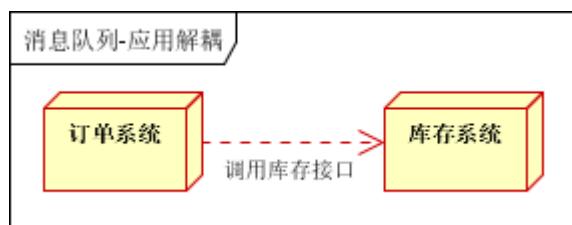
引入消息队列，将不是必须的业务逻辑，异步处理。改造后的架构如下：



按照以上约定，用户的响应时间相当于是注册信息写入数据库的时间，也就是50毫秒。注册邮件，发送短信写入消息队列后，直接返回，因此写入消息队列的速度很快，基本可以忽略，因此用户的响应时间可能是50毫秒。因此架构改变后，系统的吞吐量提高到每秒20 QPS。比串行提高了3倍，比并行提高了两倍

应用解耦

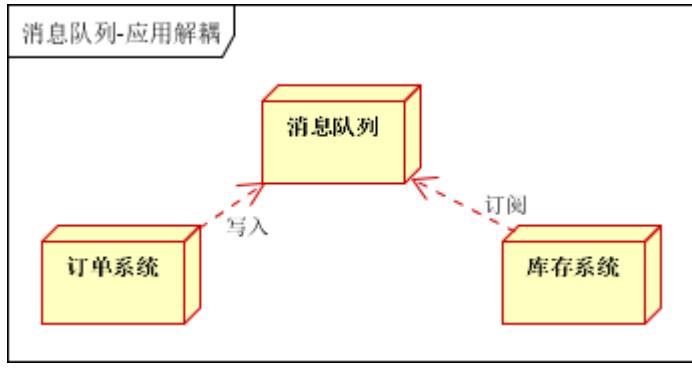
场景说明：用户下单后，订单系统需要通知库存系统。传统的做法是，订单系统调用库存系统的接口。如下图



传统模式的缺点：

- 假如库存系统无法访问，则订单减库存将失败，从而导致订单失败
- 订单系统与库存系统耦合

如何解决以上问题呢？引入应用消息队列后的方案，如下图：



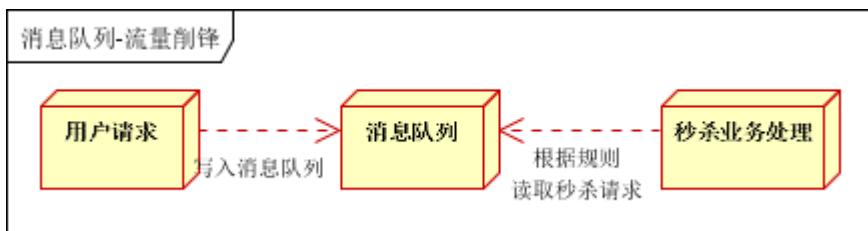
- 订单系统：用户下单后，订单系统完成持久化处理，将消息写入消息队列，返回用户订单下单成功
- 库存系统：订阅下单的消息，采用拉/推的方式，获取下单信息，库存系统根据下单信息，进行库存操作
- 假如：在下单时库存系统不能正常使用。也不影响正常下单，因为下单后，订单系统写入消息队列就不再关心其他的后续操作了。实现订单系统与库存系统的应用解耦

流量削锋

流量削锋也是消息队列中的常用场景，一般在秒杀或团抢活动中使用广泛

应用场景：秒杀活动，一般会因为流量过大，导致流量暴增，应用挂掉。为解决这个问题，一般需要在应用前端加入消息队列。

- 可以控制活动的人数
- 可以缓解短时间内高流量压垮应用



- 用户的请求，服务器接收后，首先写入消息队列。假如消息队列长度超过最大数量，则直接抛弃用户请求或跳转到错误页面
- 秒杀业务根据消息队列中的请求信息，再做后续处理

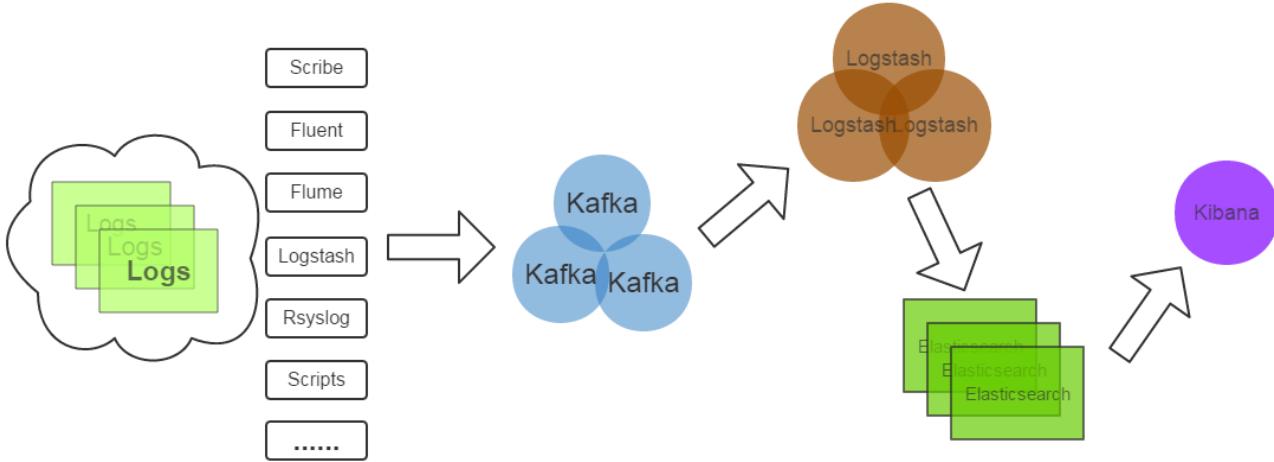
日志处理

日志处理是指将消息队列用在日志处理中，比如Kafka的应用，解决大量日志传输的问题。架构简化如下



- 日志采集客户端，负责日志数据采集，定时写受写入Kafka队列
- Kafka消息队列，负责日志数据的接收，存储和转发
- 日志处理应用：订阅并消费kafka队列中的日志数据

以下是新浪kafka日志处理应用案例：转自 (<http://cloud.51cto.com/art/201507/484338.htm>)



(1) Kafka：接收用户日志的消息队列

(2) Logstash：做日志解析，统一成JSON输出给Elasticsearch

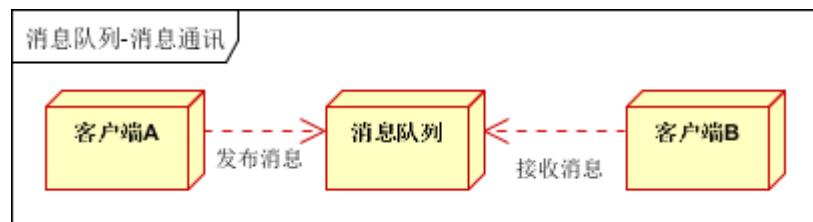
(3) Elasticsearch：实时日志分析服务的核心技术，一个schemaless，实时的数据存储服务，通过index组织数据，兼具强大的搜索和统计功能

(4) Kibana：基于Elasticsearch的数据可视化组件，超强的数据可视化能力是众多公司选择ELK stack的重要原因

消息通讯

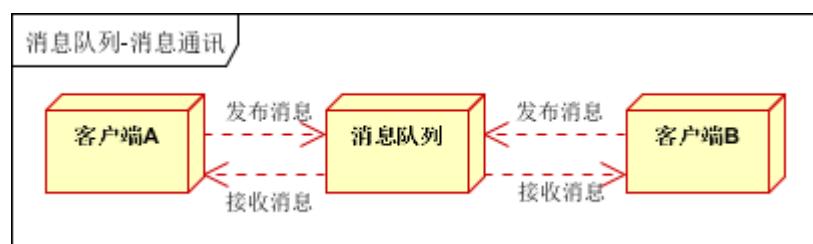
消息通讯是指，消息队列一般都内置了高效的通信机制，因此也可以用在纯的消息通讯。比如实现点对点消息队列，或者聊天室等

点对点通讯：



客户端A和客户端B使用同一队列，进行消息通讯。

聊天室通讯：

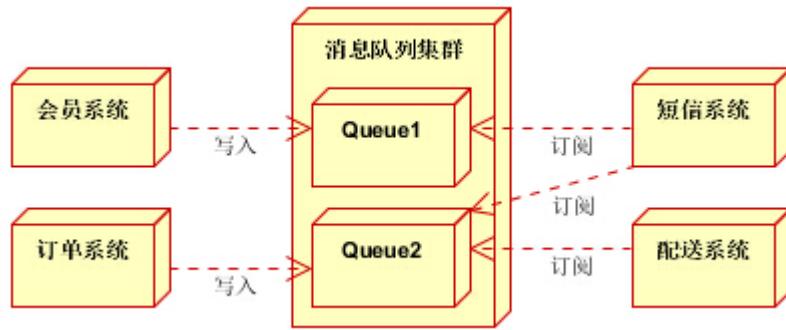


客户端A，客户端B，客户端N订阅同一主题，进行消息发布和接收。实现类似聊天室效果。

以上实际是消息队列的两种消息模式，点对点或发布订阅模式。模型为示意图，供参考。

消息中间件示例

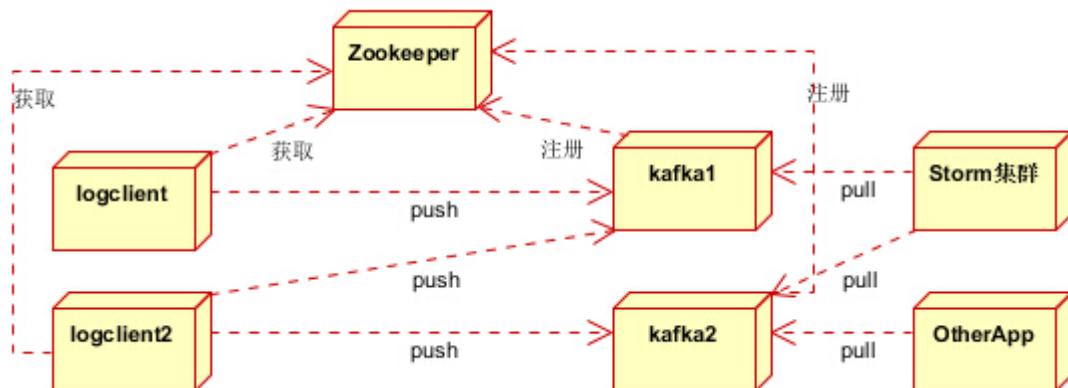
电商系统



消息队列采用高可用，可持久化的消息中间件。比如Active MQ, Rabbit MQ, Rocket Mq。

- (1) 应用将主干逻辑处理完成后，写入消息队列。消息发送是否成功可以开启消息的确认模式。（消息队列返回消息接收成功状态后，应用再返回，这样保障消息的完整性）
- (2) 扩展流程（发短信，配送处理）订阅队列消息。采用推或拉的方式获取消息并处理。
- (3) 消息将应用解耦的同时，带来了数据一致性问题，可以采用最终一致性方式解决。比如主数据写入数据库，扩展应用根据消息队列，并结合数据库方式实现基于消息队列的后续处理。

日志收集系统



分为Zookeeper注册中心，日志收集客户端，Kafka集群和Storm集群（OtherApp）四部分组成。

- Zookeeper注册中心，提供负载均衡和地址查找服务
- 日志收集客户端，用于采集应用系统的日志，并将数据推送到kafka队列
- Kafka集群：接收，路由，存储，转发等消息处理

Storm集群：与OtherApp处于同一年级，采用拉的方式消费队列中的数据

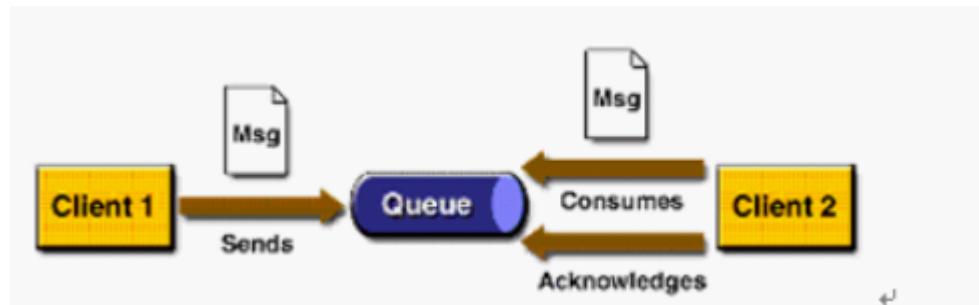
JMS消息服务

讲消息队列就不得不提JMS。JMS (JAVA Message Service, Java消息服务) API是一个消息服务的标准/规范，允许应用程序组件基于JavaEE平台创建、发送、接收和读取消息。它使分布式通信耦合度更低，消息服务更加可靠以及异步性。

在EJB架构中，有消息bean可以无缝的与JM消息服务集成。在J2EE架构模式中，有消息服务器模式，用于实现消息与应用直接的解耦。

消息模型

在JMS标准中，有两种消息模型P2P (Point to Point) ,Publish/Subscribe(Pub/Sub)。

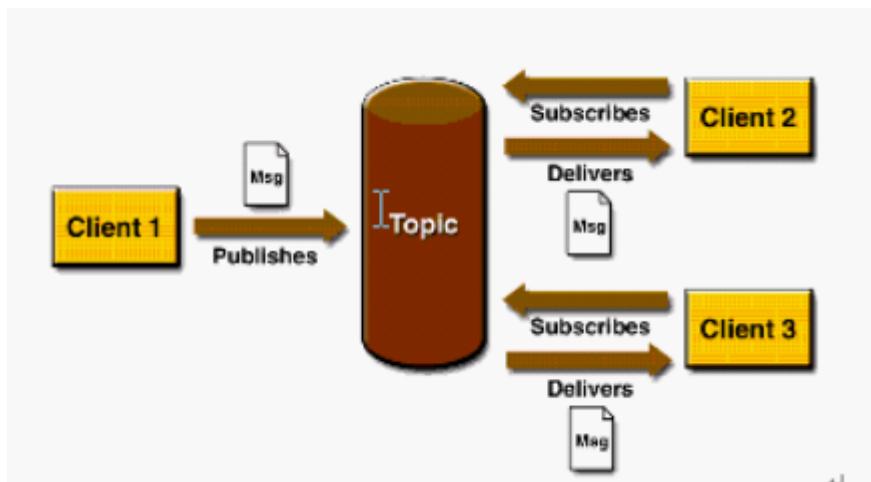


P2P模式包含三个角色：消息队列（Queue），发送者(Sender)，接收者(Receiver)。每个消息都被发送到一个特定的队列，接收者从队列中获取消息。队列保留着消息，直到他们被消费或超时。

P2P的特点

- 每个消息只有一个消费者（Consumer）(即一旦被消费，消息就不再在消息队列中)
- 发送者和接收者之间在时间上没有依赖性，也就是说当发送者发送了消息之后，不管接收者有没有正在运行，它不会影响到消息被发送到队列
- 接收者在成功接收消息之后需向队列应答成功

如果希望发送的每个消息都会被成功处理的话，那么需要P2P模式。



Pub/sub模式包含三个角色主题（Topic），发布者（Publisher），订阅者（Subscriber） 多个发布者将消息发送到Topic,系统将这些消息传递给多个订阅者。

Pub/Sub的特点

- 每个消息可以有多个消费者
- 发布者和订阅者之间有时间上的依赖性。针对某个主题（Topic）的订阅者，它必须创建一个订阅者之后，才能消费发布者的消息
- 为了消费消息，订阅者必须保持运行的状态

为了缓和这样严格的时间相关性，JMS允许订阅者创建一个可持久化的订阅。这样，即使订阅者没有被激活（运行），它也能接收到发布者的消息。

如果希望发送的消息可以不被做任何处理、或者只被一个消息者处理、或者可以被多个消费者处理的话，那么可以采用Pub/Sub模型。

消息消费

在JMS中，消息的产生和消费都是异步的。对于消费来说，JMS的消息者可以通过两种方式来消费消息。

- (1) 同步

订阅者或接收者通过receive方法来接收消息，receive方法在接收到消息之前（或超时之前）将一直阻塞；

(2) 异步

订阅者或接收者可以注册为一个消息监听器。当消息到达之后，系统自动调用监听器的onMessage方法。

防止消息丢失

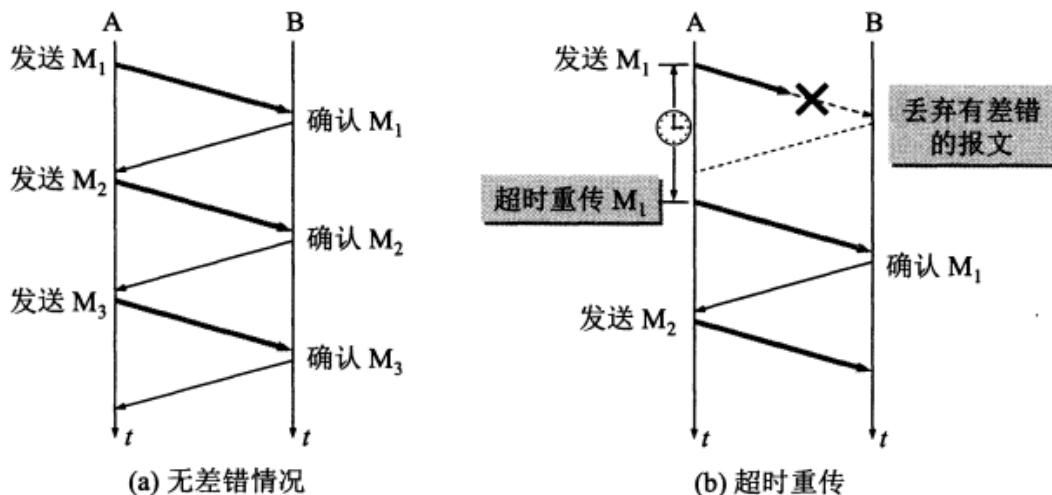
由于网络问题，我们很难保证生产者发送的消息能100%到达消息队列服务器，也就是说有消息丢失的可能性，因此，生产者就必须具有消息丢失检测和重发机制，也就是我们常说的消息队列的事物机制

不能把可靠性的保证全部交给TCP，TCP只保证了传输层的可靠传输，但是无法保证与应用层的交互是否出错

TCP无法给应用层任何反馈，因此必须在应用层处理差错

同步的事务——停止等待

所谓停止等待协议就是没发送完一组数据后，等待对方确认并且收到确认后，再发送下一组数据。



同步的事物——连续ARQ

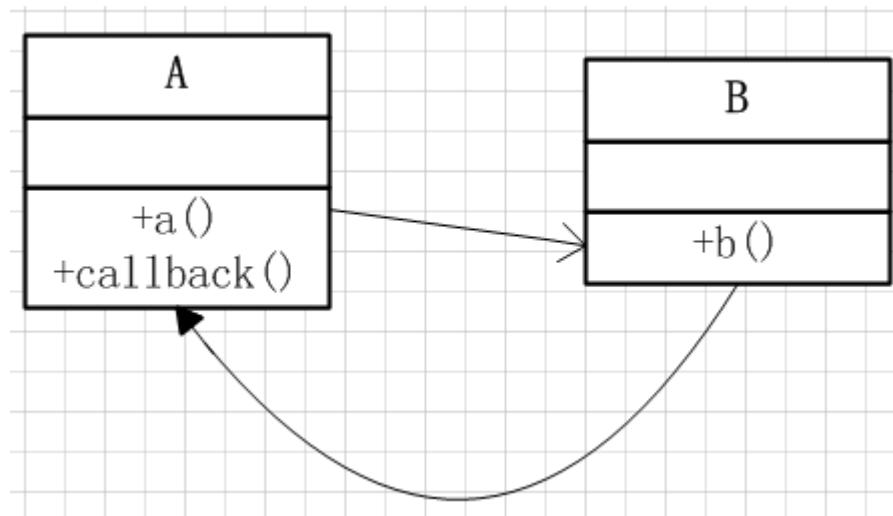
类似于TCP的滑动窗口模型



图 5-13 连续 ARQ 协议的工作原理

异步的事务——回调机制

生产者在发送消息的时候，注册一个回调函数，这样生产者便不用停下来等待确认了，而是可以一直持续发送消息，当消息到达消息队列服务器的时候，服务器便会调用生产者注册的回调函数，告知生产者消息发送成功了还是失败了，进而做进一步的处理，从而提高了并发量。



消息的幂等处理

由于网络原因，生产者可能会重复发送消息，因此消费者方必须做消息的幂等处理，常用的解决方案有：

1. **查询操作**：查询一次和查询多次，在数据不变的情况下，查询结果是一样的。select是天然的幂等操作；
2. **删除操作**：删除操作也是幂等的，删除一次和多次删除都是把数据删除。（注意可能返回结果不一样，删除的数据不存在，返回0，删除的数据多条，返回结果多个）；
3. **唯一索引**，防止新增脏数据。比如：支付宝的资金账户，支付宝也有用户账户，每个用户只能有一个资金账户，怎么防止给用户创建资金账户多个，那么给资金账户表中的用户ID加唯一索引，所以一个用户新增成功一个资金账户记录。要点：唯一索引或唯一组合索引来防止新增数据存在脏数据（当表存在唯一索引，并发时新增报错时，再查询一次就可以了，数据应该已经存在了，返回结果即可）；
4. **token机制**，防止页面重复提交。业务要求：页面的数据只能被点击提交一次；发生原因：由于重复点击或者网络重发，或者nginx重发等情况会导致数据被重复提交；解决办法：集群环境采用token加redis(redis单线程的，处理需要排队）；单JVM环境：采用token加redis或token加jvm内存。处理流程：1. 数据提交前要向服务的申请token，token放到redis或jvm内存，token有效时间；2. 提交后后台校验token，同时删除token，生成新的token返回。token特点：要申请，一次有效性，可以限流。注意：redis要用删除操作来判断token，删除成功代表token校验通过，如果用select+delete来校验token，存在并发问题，不建议使用；
5. 悲观锁——获取数据的时候加锁获取。`select * from table_xxx where id='xxx' for update;` 注意：id字段一定是主键或者唯一索引，而不是锁表，会死人的悲观锁使用时一般伴随事务一起使用，数据锁定时间可能会很长，根据实际情况选用；
6. 乐观锁——乐观锁只是在更新数据那一刻锁表，其他时间不锁表，所以相对于悲观锁，效率更高。乐观锁的实现方式多种多样可以通过version或者其他状态条件：1. 通过版本号实现`update table_xxx set name=#name#,version=version+1 where version=#version#`如下图(来自网上)；2. 通过条件限制`update table_xxx set avai_amount=avai_amount-#subAmount# where avai_amount-#subAmount# >= 0`要求：`quality-#subQuality# >=`，这个情景适合不用版本号，只更新是做数据安全校验，适合库存模型，扣份额和回滚份额，性能更高；

```
update table_xxx set name=#name#,version=version+1 where id=#id# and version=#version#;

update table_xxx set avai_amount=avai_amount-#subAmount# where id=#id# and avai_amount-
#subAmount# >= 0;
```

7.分布式锁——还是拿插入数据的例子，如果是分布式系统，构建全局唯一索引比较困难，例如唯一性的字段没法确定，这时候可以引入分布式锁，通过第三方的系统(redis或zookeeper)，在业务系统插入数据或者更新数据，获取分布式锁，然后做操作，之后释放锁，这样其实是把多线程并发的锁的思路，引入多个系统，也就是分布式系统中得解决思路。要点：某个长流程处理过程要求不能并发执行，可以在流程执行之前根据某个标志(用户ID+后缀等)获取分布式锁，其他流程执行时获取锁就会失败，也就是同一时间该流程只能有一个能执行成功，执行完成后，释放分布式锁(分布式锁要第三方系统提供)；

8.select + insert——并发不高的后台系统，或者一些任务JOB，为了支持幂等，支持重复执行，简单的处理方法是，先查询下一些关键数据，判断是否已经执行过，在进行业务处理，就可以了。注意：核心高并发流程不要用这种方法；

消息的按序处理

同上，消息的按序也不能完全依靠于TCP

在说到消息中间件的时候，我们通常都会谈到一个特性：消息的顺序消费问题。这个问题看起来很简单：Producer发送消息1, 2, 3。。。 Consumer按1, 2, 3。。。顺序消费。

但实际情况却是：无论RocketMQ，还是Kafka，缺省都不保证消息的严格有序消费！

这个特性看起来很简单，但为什么缺省他们都不保证呢？

“严格的顺序消费”有多么困难

下面就从3个方面来分析一下，对于一个消息中间件来说，“严格的顺序消费”有多么困难，或者说不可能。

发送端

发送端不能异步发送，异步发送在发送失败的情况下，就没办法保证消息顺序。

比如你连续发了1, 2, 3。过了一会，返回结果1失败，2, 3成功。你把1再重新发送1遍，这个时候顺序就乱掉了。

存储端

对于存储端，要保证消息顺序，会有以下几个问题：（1）消息不能分区。也就是1个topic，只能有1个队列。在Kafka中，它叫做partition；在RocketMQ中，它叫做queue。如果你有多个队列，那同1个topic的消息，会分散到多个分区里面，自然不能保证顺序。

（2）即使只有1个队列的情况下，会有第2个问题。该机器挂了之后，能否切换到其他机器？也就是高可用问题。

比如你当前的机器挂了，上面还有消息没有消费完。此时切换到其他机器，可用性保证了。但消息顺序就乱掉了。

要想保证，一方面要同步复制，不能异步复制；另1方面得保证，切机器之前，挂掉的机器上面，所有消息必须消费完了，不能有残留。很明显，这个很难！！！

接收端

对于接收端，不能并行消费，也即不能开多线程或者多个客户端消费同1个队列。

总结

从上面的分析可以看出，要保证消息的严格有序，有多么困难！

发送端和接收端的问题，还好解决一点，限制异步发送，限制并行消费。但对于存储端，机器挂了之后，切换的问题，就很难解决了。

你切换了，可能消息就会乱；你不切换，那就暂时不可用。这2者之间，就需要权衡了。

业务需要全局有序吗？

通过上面分析可以看出，要保证一个topic内部，消息严格的有序，是很困难的，或者说条件是很苛刻的。

那怎么办呢？我们一定要使出所有力气、用尽所有办法，来保证消息的严格有序吗？

这里就需要从另外一个角度去考虑这个问题：业务角度。正如在下面这篇博客中所说的：

<http://www.jianshu.com/p/453c6e7ff81c>

实际情况中：（1）不关注顺序的业务大量存在；（2）队列无序不代表消息无序。

三、搜索引擎

概述

全文搜索就是对文本数据的一种搜索方式，文本数据的都多，可以分为顺序搜索法和索引搜索法，，全文检索使用的是索引搜索法

特点（优势）：

- 做了相关度排序
- 对文本中的关键字做了高亮显示
- 摘要截取
- 只关注文本，不考虑语义
- 搜索效果更加精确——基于单词搜索，比如搜索Java的时候找不到JavaScript，因为它们是不同的两个单词

使用场景：

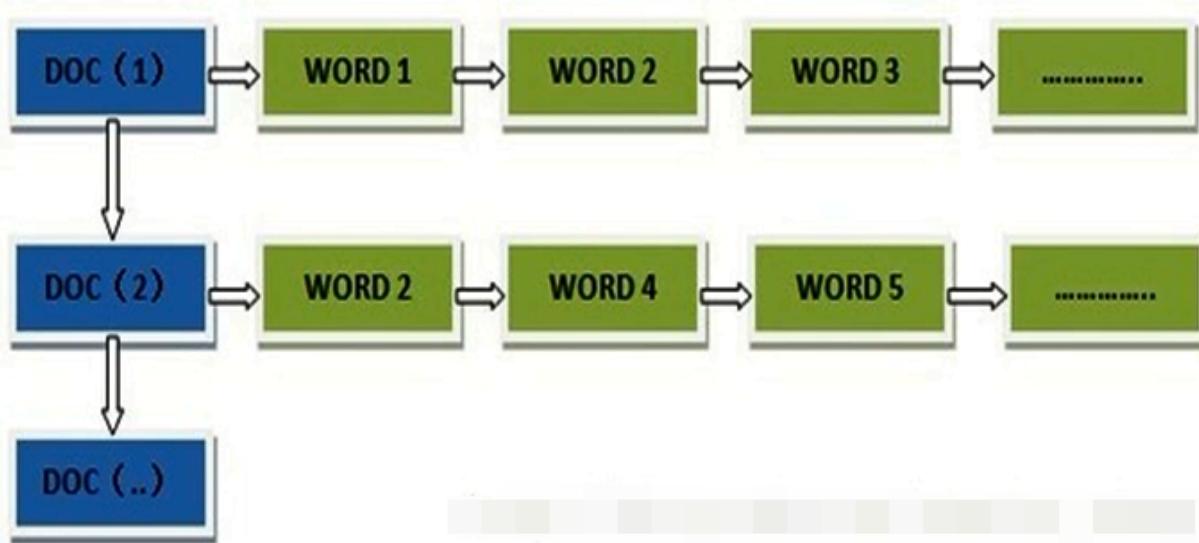
- 替换数据库的模糊查询，提高查询速度，降低数据库压力，增强了查询效率
- 数据库模糊查询缺点：查询速度慢，左模糊和全模糊会使索引失效，没有相关度排序，没有对文本中关键字做高亮显示，搜索效果不好
- 全文检索是搜索引擎的基础
- 只对“指定领域”的网站进行索引和搜索，即垂直搜索
- 可以在word、pdf等各种各样的数据格式中检索内容
- 其他场合，比如输入法等

倒排索引

正向索引的结构如下：

“文档1”的ID > 单词1：出现次数，出现位置列表；单词2：出现次数，出现位置列表；.....。

“文档2”的ID > 此文档出现的关键词列表。



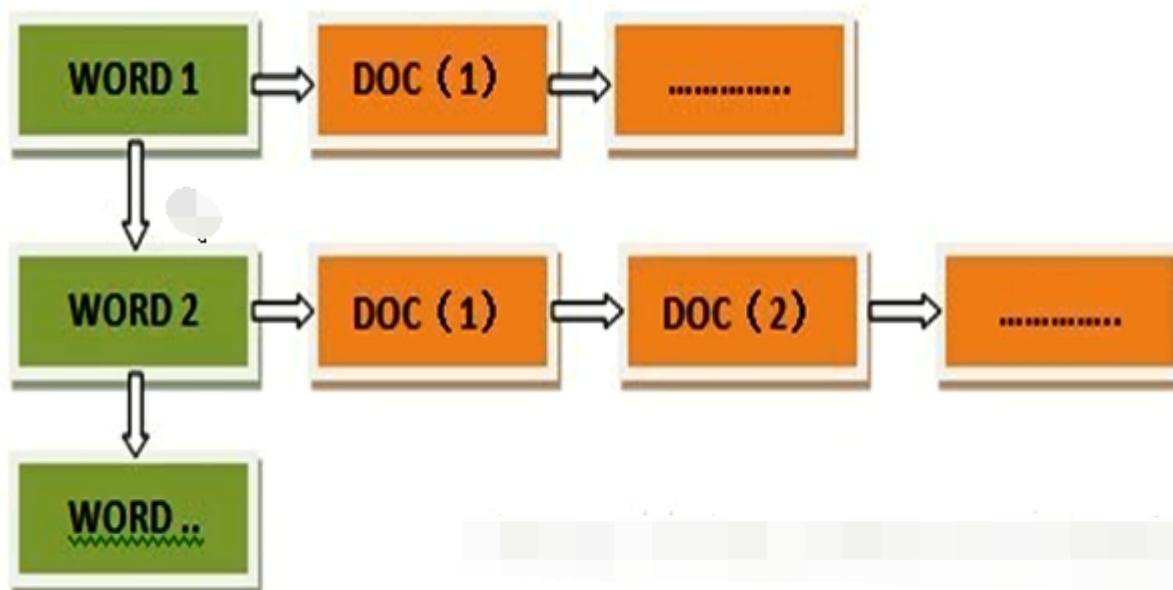
当用户在主页上搜索关键词“华为手机”时，假设只存在正向索引（forward index），那么就需要扫描索引库中的所有文档，找出所有包含关键词“华为手机”的文档，再根据打分模型进行打分，排出名次后呈现给用户。因为互联网上收录在搜索引擎中的文档的数目是个天文数字，这样的索引结构根本无法满足实时返回排名结果的要求。

所以，搜索引擎会将正向索引重新构建为倒排索引，即把文件ID对应到关键词的映射转换为关键词到文件ID的映射，每个关键词都对应着一系列的文件，这些文件中都出现这个关键词。

得到倒排索引的结构如下：

“关键词1”：“文档1”的ID，“文档2”的ID，.....。

“关键词2”：带有此关键词的文档ID列表。



创建索引

全文检索的索引创建过程一般有以下几步：

一些要索引的原文档(Document)

为了方便说明索引创建过程，这里特意用两个文件为例：

文件一：Students should be allowed to [Go](#) out with their friends, but not allowed to drink beer.

文件二：My friend Jerry went to school to see his students but found them drunk which is not allowed.

将原文档传给分词组件(Tokenizer)

分词组件(Tokenizer)会做以下几件事情(此过程称为Tokenize)：

1. 将文档分成一个一个单独的单词。

2. 去除标点符号。

3. 去除停词(Stop word)。

所谓停词(Stop word)就是一种语言中最普通的一些单词，由于没有特别的意义，因而大多数情况下不能成为搜索的关键词，因而创建索引时，这种词会被去掉而减少索引的大小。

英语中挺词(Stop word)如：“the”, “a”, “this”等。

对于每一种语言的分词组件(Tokenizer)，都有一个停词(stop word)集合。

经过分词(Tokenizer)后得到的结果称为词元(Token)。

在我们的例子中，便得到以下词元(Token)：

“Students”, “allowed”, “go”, “their”, “friends”, “allowed”, “drink”, “beer”, “My”, “friend”, “Jerry”, “went”, “school”, “see”, “his”, “students”, “found”, “them”, “drunk”, “allowed”。

将得到的词元(Token)传给语言处理组件(Linguistic Processor)

语言处理组件(linguistic processor)主要是对得到的词元(Token)做一些同语言相关的处理。

对于英语，语言处理组件(Linguistic Processor)一般做以下几点：

1. 变为小写(Lowercase)。

2. 将单词缩减为词根形式，如“cars”到“car”等。这种操作称为：stemming。

3. 将单词转变为词根形式，如“drove”到“drive”等。这种操作称为：lemmatization。

Stemming 和 lemmatization 的异同：

- 相同之处：Stemming和lemmatization都要使词汇成为词根形式。

- 两者的方式不同：

- Stemming采用的是“缩减”的方式：“cars”到“car”，“driving”到“drive”。

- Lemmatization采用的是“转变”的方式：“drove”到“drove”，“driving”到“drive”。

- 两者的算法不同：

- Stemming主要是采取某种固定的算法来做这种缩减，如去除“s”，去除“ing”加“e”，将“ational”变为“ate”，将“tional”变为“tion”。

- Lemmatization主要是采用保存某种字典的方式做这种转变。比如字典中
有“driving”到“drive”，“drove”到“drive”，“am, is, are”到“be”的映射，做转变时，只要查字典就可以了。
- Stemming和lemmatization不是互斥关系，是有交集的，有的词利用这两种方式都能达到相同的转换。

语言处理组件(linguistic processor)的结果称为词(Term)。

在我们的例子中，经过语言处理，得到的词(Term)如下：

“student”, “allow”, “go”, “their”, “friend”, “allow”, “drink”, “beer”, “my”, “friend”, “jerry”, “go”, “school”, “see”, “his”, “student”, “find”, “them”, “drink”, “allow”。

也正是因为有语言处理的步骤，才能使搜索drove，而drive也能被搜索出来。

将得到的词(Term)传给索引组件(Indexer)

索引组件(Indexer)主要做以下几件事情：

1. 利用得到的词(Term)创建一个字典。

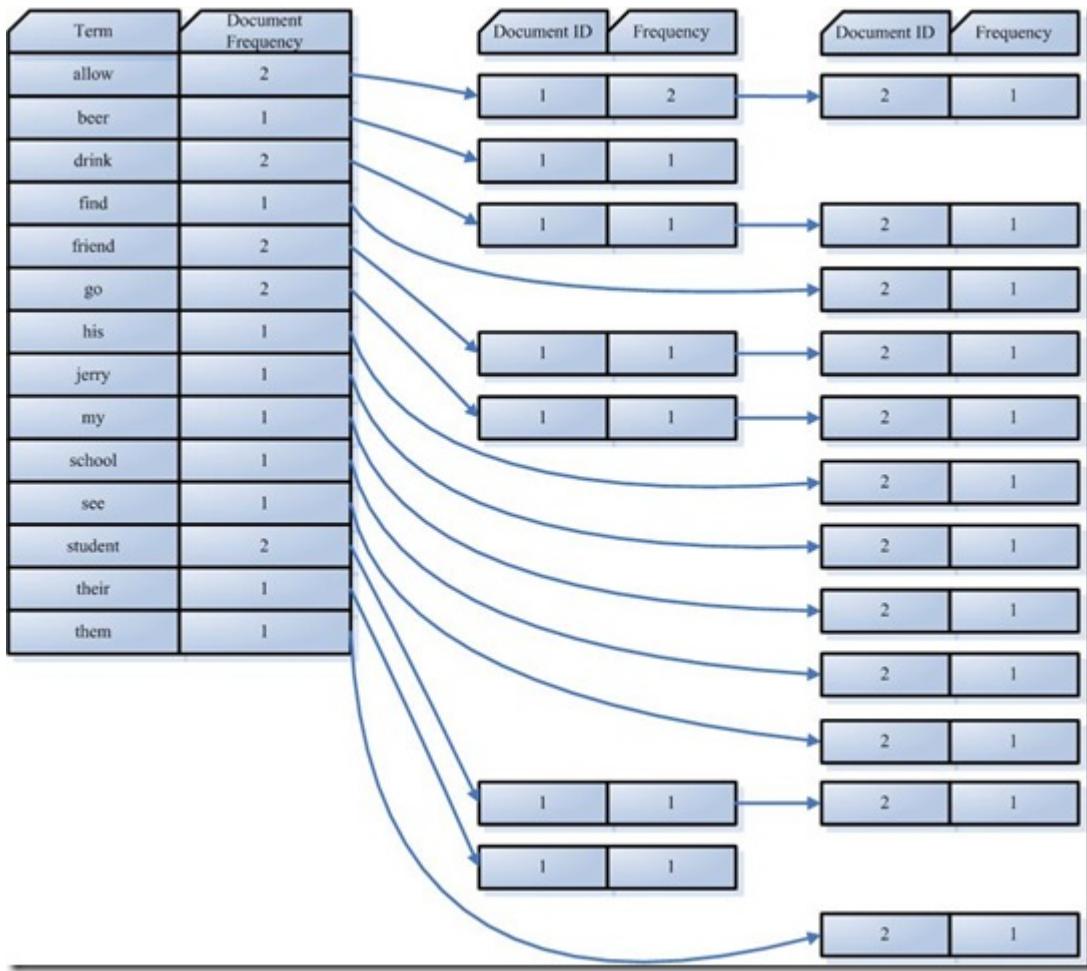
在我们的例子中字典如下：

Term	Document ID
student	1
allow	1
go	1
their	1
friend	1
allow	1
drink	1
beer	1
my	2
friend	2
jerry	2
go	2
school	2
see	2
his	2
student	2
find	2
them	2
drink	2
allow	2

2. 对字典按字母顺序进行排序。

Term	Document ID
allow	1
allow	1
allow	2
beer	1
drink	1
drink	2
find	2
friend	1
friend	2
go	1
go	2
his	2
jerry	2
my	2
school	2
see	2
student	1
student	2
their	1
them	2

3. 合并相同的词(Term) 成为文档倒排(Posting List) 链表。



在此表中，有几个定义：

- Document Frequency 即文档频次，表示总共有多少文件包含此词(Term)。
- Frequency 即词频率，表示此文件中包含了几个此词(Term)。

所以对词(Term) “allow”来讲，总共有两篇文档包含此词(Term)，从而词(Term)后面的文档链表总共有两项，第一项表示包含“allow”的第一篇文档，即1号文档，此文档中，“allow”出现了2次，第二项表示包含“allow”的第二个文档，是2号文档，此文档中，“allow”出现了1次。

到此为止，索引已经创建好了，我们可以通过它很快的找到我们想要的文档。

而且在此过程中，我们惊喜地发现，搜索“drive”，“driving”，“drove”，“driven”也能够被搜到。因为在我们的索引中，“driving”，“drove”，“driven”都会经过语言处理而变成“drive”，在搜索时，如果您输入“driving”，输入的查询语句同样经过我们这里的一到三步，从而变为查询“drive”，从而可以搜索到想要的文档。

搜索索引

到这里似乎我们可以宣布“我们找到想要的文档了”。

然而事情并没有结束，找到了仅仅是全文检索的一个方面。不是吗？如果仅仅只有一个或十个文档包含我们查询的字符串，我们的确找到了。然而如果结果有一千个，甚至成千上万个呢？那个又是您最想要的文件呢？

打开Google吧，比如说您想在微软找份工作，于是您输入“Microsoft job”，您却发现总共有22600000个结果返回。好大的数字呀，突然发现找不到是一个问题，找到的太多也是一个问题。在如此多的结果中，如何将最相关的放在最前面呢？

Web Results 1 - 10 of about 22,600,000 for Microsoft job. (0.15 seconds)

[Microsoft Jobs - Locations - Global Careers](#)
Microsoft Employment Opportunity Web Site: Find information on jobs at Microsoft throughout the domestic United States and at our corporate headquarters in ...
www.microsoft.com/careers/ - 20k - [Cached](#) - [Similar pages](#)

[Microsoft Canada Career Centre - Home](#)
The Microsoft Canada Development Centre will open in the fall of 2007 and will ... Featured Jobs. J0807-0924: Bilingual Pre-Sales Specialist, Dynamics CRM ...
www.microsoft.com/canada/employment/default.mspx - 13k - [Cached](#) - [Similar pages](#)
[[More results from www.microsoft.com](#)]

[Microsoft Careers Home](#)
Create up to five job search agents, which will automatically search for Microsoft® positions that match your specific search criteria and notify you of the ...
members.microsoft.com/careers/default.mspx - 12k - [Cached](#) - [Similar pages](#)

当然Google做的很不错，您一下就找到了jobs at Microsoft。想象一下，如果前几个全部是“Microsoft does a good job at software industry...”将是多么可怕的事情呀。

如何像Google一样，在成千上万的搜索结果中，找到和查询语句最相关的呢？

如何判断搜索出的文档和查询语句的相关性呢？

这要回到我们第三个问题：如何对索引进行搜索？

搜索主要分为以下几步：

用户输入查询语句

查询语句同我们普通的语言一样，也是有一定语法的。

不同的查询语句有不同的语法，如SQL语句就有一定的语法。

查询语句的语法根据全文检索系统的实现而不同。最基本的有比如：AND, OR, NOT等。

举个例子，用户输入语句：lucene AND learned NOT [Hadoop](#)。

说明用户想找一个包含lucene和learned然而不包括hadoop的文档。

对查询语句进行词法分析，语法分析，及语言处理

由于查询语句有语法，因而也要进行语法分析，语法分析及语言处理。

1. 词法分析主要用来识别单词和关键字。

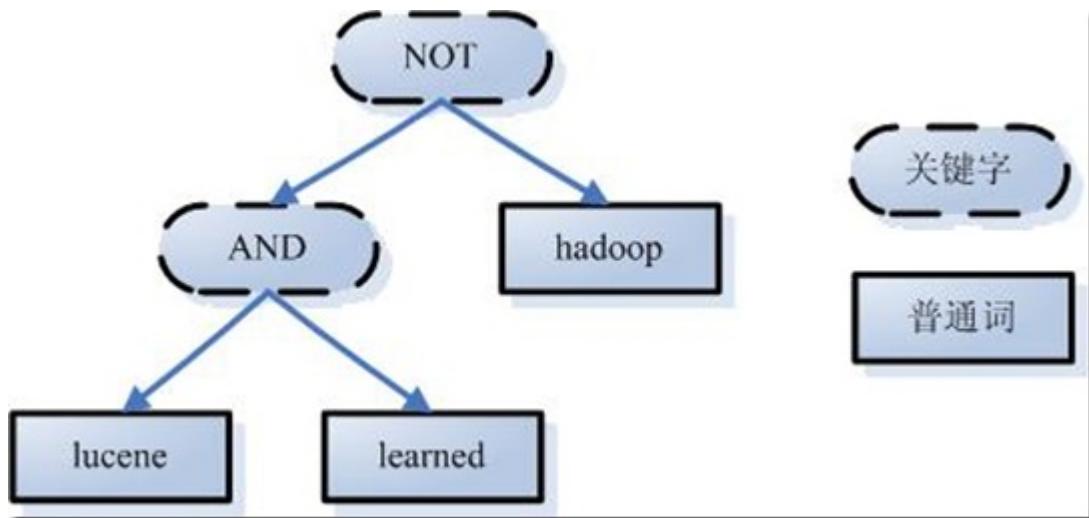
如上述例子中，经过词法分析，得到单词有lucene, learned, hadoop, 关键字有AND, NOT。

如果在词法分析中发现不合法的关键字，则会出现错误。如lucene AMD learned，其中由于AND拼错，导致AMD作为一个普通的单词参与查询。

2. 语法分析主要是根据查询语句的语法规则来形成一棵语法树。

如果发现查询语句不满足语法规则，则会报错。如lucene NOT AND learned，则会出错。

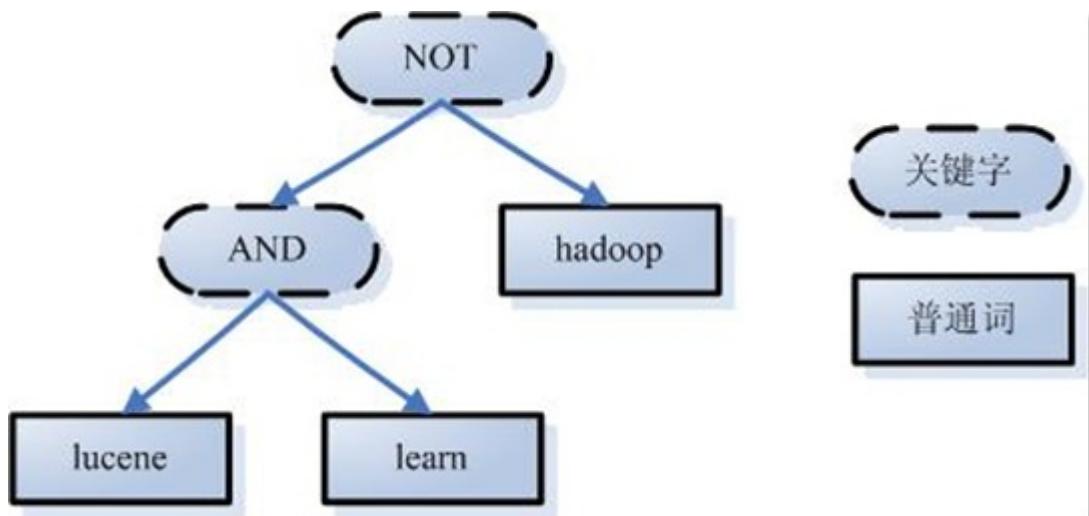
如上述例子，lucene AND learned NOT hadoop形成的语法树如下：



3. 语言处理同素引过程中的语言处理几乎相同。

如learned变成learn等。

经过第二步，我们得到一棵经过语言处理的语法树。



搜索索引，得到符合语法树的文档

此步骤有分几小步：

1. 首先，在反向索引表中，分别找出包含lucene, learn, hadoop的文档链表。
2. 其次，对包含lucene, learn的链表进行合并操作，得到既包含lucene又包含learn的文档链表。
3. 然后，将此链表与hadoop的文档链表进行差操作，去除包含hadoop的文档，从而得到既包含lucene又包含learn而且不包含hadoop的文档链表。
4. 此文档链表就是我们要找的文档。

根据得到的文档和查询语句的相关性，对结果进行排序

虽然在上一步，我们得到了想要的文档，然而对于查询结果应该按照与查询语句的相关性进行排序，越相关者越靠前。

如何计算文档和查询语句的相关性呢？

不如我们把查询语句看作一片短小的文档，对文档与文档之间的相关性(relevance)进行打分(scoring)，分数高的相关性好，就应该排在前面。

那么又怎么对文档之间的关系进行打分呢？

这可不是一件容易的事情，首先我们看一看判断人之间的关系吧。

首先 看一个人，往往有很多要素，如性格，信仰，爱好，衣着，高矮，胖瘦等等。

其次 对于人与人之间的关系，不同的要素重要性不同，性格，信仰，爱好可能重要些，衣着，高矮，胖瘦可能就不那么重要了，所以具有相同或相似性格，信仰，爱好的人比较容易成为好的朋友，然而衣着，高矮，胖瘦不同的人，也可以成为好的朋友。

因而判断人与人之间的关系，首先要找出哪些要素对人与人之间的关系最重要，比如性格，信仰，爱好。其次要判断两个人的这些要素之间的关系，比如一个人性格开朗，另一个人性格外向，一个人信仰佛教，另一个信仰上帝，一个人爱好打篮球，另一个爱好踢足球。我们发现，两个人在性格方面都很积极，信仰方面都很善良，爱好方面都爱运动，因而两个人关系应该会很好。

我们再来看看公司之间的关系吧。

首先 看一个公司，有很多人组成，如总经理，经理，首席技术官，普通员工，保安，门卫等。

其次对于公司与公司之间的关系，不同的人重要性不同，总经理，经理，首席技术官可能更重要一些，普通员工，保安，门卫可能较不重要一点。所以如果两个公司总经理，经理，首席技术官之间关系比较好，两个公司容易有比较好的关系。然而一位普通员工就算与另一家公司的一位普通员工有血海深仇，怕也难影响两个公司之间的关系。

因而判断公司与公司之间的关系，首先要找出哪些人对公司与公司之间的关系最重要，比如总经理，经理，首席技术官。其次要判断这些人之间的关系，不如两家公司的总经理曾经是同学，经理是老乡，首席技术官曾是创业伙伴。我们发现，两家公司无论总经理，经理，首席技术官，关系都很好，因而两家公司关系应该会很好。

分析了两种关系，下面看一下如何判断文档之间的关系了。

首先，一个文档有很多词(Term)组成，如search, lucene, full-text, this, a, what等。

其次对于文档之间的关系，不同的Term重要性不同，比如对于本篇文档，search, Lucene, full-text就相对重要一些，this, a, what可能相对不重要一些。所以如果两篇文档都包含search, Lucene, fulltext，这两篇文档的相关性好一些，然而就算一篇文档包含this, a, what，另一篇文档不包含this, a, what，也不能影响两篇文档的相关性。

因而判断文档之间的关系，首先找出哪些词(Term)对文档之间的关系最重要，如search, Lucene, fulltext。然后判断这些词(Term)之间的关系。

找出词(Term)对文档的重要性过程称为计算词的权重(Term weight)的过程。

计算词的权重(term weight)有两个参数，第一个是词(Term)，第二个是文档(Document)。

词的权重(Term weight)表示此词(Term)在此文档中的重要程度，越重要的词(Term)有越大的权重(Term weight)，因而在计算文档之间的相关性中将发挥更大的作用。

判断词(Term)之间的关系从而得到文档相关性的过程应用一种叫做向量空间模型的算法(Vector Space Model)。

下面仔细分析一下这两个过程：

1. 计算权重(Term weight)的过程。

影响一个词(Term)在一篇文档中的重要性主要有两个因素：

- Term Frequency (tf)：即此Term在此文档中出现了多少次。tf 越大说明越重要。
- Document Frequency (df)：即有多少文档包含次Term。df 越大说明越不重要。

容易理解吗？词(Term)在文档中出现的次数越多，说明此词(Term)对该文档越重要，如“搜索”这个词，在本文档中出现的次数很多，说明本文档主要就是讲这方面的事的。然而在一篇英语文档中，this出现的次数更多，就说明越重要吗？不是的，这是由第二个因素进行调整，第二个因素说明，有越多的文档包含此词(Term)，说明此词(Term)太普通，不足以区分这些文档，因而重要性越低。

这也如我们程序员所学的技术，对于程序员本身来说，这项技术掌握越深越好（掌握越深说明花时间看的越多，tf越大），找工作时越有竞争力。然而对于所有程序员来说，这项技术懂得的人越少越好（懂得的人少df小），找工作越有竞争力。人的价值在于不可替代性就是这个道理。

道理明白了，我们来看看公式：

$$w_{t,d} = tf_{t,d} \times \log(n / df_t)$$

$w_{t,d}$ = the weight of the term t in document d

$tf_{t,d}$ = frequency of term t in document d

n = total number of documents

df_t = the number of documents that contain term t

这仅仅只term weight计算公式的简单典型实现。实现全文检索系统的人会有自己的实现，Lucene就与此稍有不同。

2. 判断Term之间的关系从而得到文档相关性的过程，也即向量空间模型的算法(VSM)。

我们把文档看作一系列词(Term)，每一个词(Term)都有一个权重(Term weight)，不同的词(Term)根据自己的在文档中的权重来影响文档相关性的打分计算。

于是我们把所有此文档中词(term)的权重(term weight)看作一个向量。

Document = {term1, term2, ,term N}

Document Vector = {weight1, weight2, ,weight N}

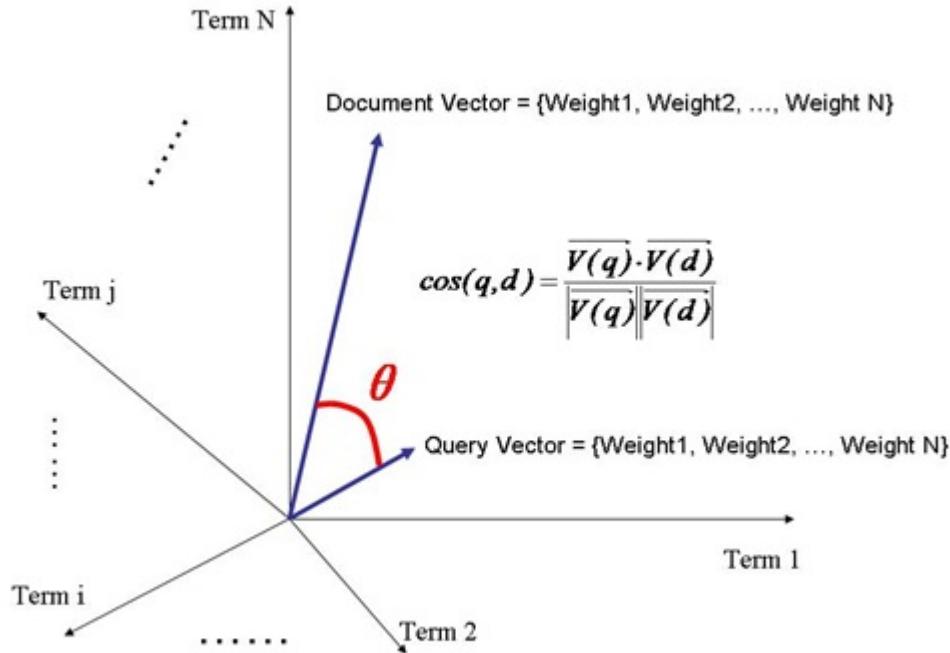
同样我们把查询语句看作一个简单的文档，也用向量来表示。

Query = {term1, term 2, , term N}

Query Vector = {weight1, weight2, , weight N}

我们把所有搜索出的文档向量及查询向量放到一个N维空间中，每个词(term)是一维。

如图：



我们认为两个向量之间的夹角越小，相关性越大。

所以我们计算夹角的余弦值作为相关性的打分，夹角越小，余弦值越大，打分越高，相关性越大。

有人可能会问，查询语句一般是很短的，包含的词(Term)是很少的，因而查询向量的维数很小，而文档很长，包含词(Term)很多，文档向量维数很大。你的图中两者维数怎么都是N呢？

在这里，既然要放到相同的向量空间，自然维数是相同的，不同时，取二者的并集，如果不含某个词(Term)时，则权重(Term Weight)为0。

相关性打分公式如下：

$$score(q, d) = \frac{\vec{V}_q \cdot \vec{V}_d}{\|\vec{V}_q\| \|\vec{V}_d\|} = \frac{\sum_{i=1}^n w_{i,q} w_{i,d}}{\sqrt{\sum_{i=1}^n w_{i,q}^2} \sqrt{\sum_{i=1}^n w_{i,d}^2}}$$

举个例子，查询语句有11个Term，共有三篇文档搜索出来。其中各自的权重(Term weight)，如下表格。

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11
D1	0	0	.477	0	.477	.176	0	0	0	.176	0
D2	0	.176	0	.477	0	0	0	0	.954	0	.176
D3	0	.176	0	0	0	.176	0	0	0	.176	.176
Q	0	0	0	0	0	.176	0	0	.477	0	.176

于是计算，三篇文档同查询语句的相关性打分分别为：

$$SC(Q, D_1) = \frac{(0.176)(0.176)}{\sqrt{0.477^2 + 0.477^2 + 0.176^2 + 0.176^2} \sqrt{0.176^2 + 0.477^2 + 0.176^2}} \approx 0.08$$

$$SC(Q, D_2) = \frac{(0.954)(0.477) + (0.176)^2}{\sqrt{0.176^2 + 0.477^2 + 0.954^2 + 0.176^2} \sqrt{0.176^2 + 0.477^2 + 0.176^2}} \approx 0.825$$

$$SC(Q, D_3) = \frac{(0.176)^2 + (0.176)^2}{\sqrt{0.176^2 + 0.176^2 + 0.176^2 + 0.176^2} \sqrt{0.176^2 + 0.477^2 + 0.176^2}} \approx 0.327$$

于是文档二相关性最高，先返回，其次是文档一，最后是文档三。

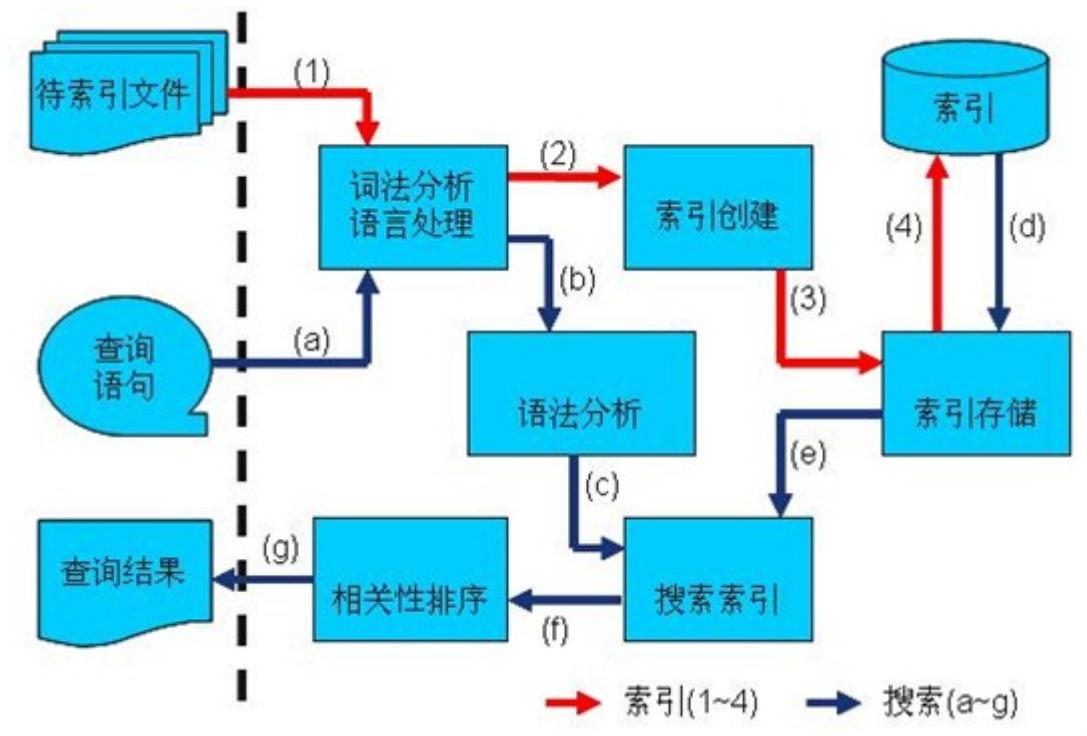
ItemCF UserCF

到此为止，我们可以找到我们最想要的文档了。

说了这么多，其实还没有进入到Lucene，而仅仅是信息检索技术(Information retrieval)中的基本理论，然而当我们看过Lucene后我们会发现，Lucene是对这种基本理论的一种基本的实践。所以在以后分析Lucene的文章，会常常看到以上理论在Lucene中的应用。

在进入Lucene之前，对上述索引创建和搜索过程做一个总结，如图：

此图参照<http://www.lucene.com.cn/about.htm> 中文章《开放源代码的全文检索引擎Lucene》



1. 索引过程：

1) 有一系列被索引文件

2) 被索引文件经过语法分析和语言处理形成一系列词(Term)。

3) 经过索引创建形成词典和反向索引表。

4) 通过索引存储将索引写入硬盘。

2. 搜索过程:

a) 用户输入查询语句。

b) 对查询语句经过语法分析和语言分析得到一系列词(Term)。

c) 通过语法分析得到一个查询树。

d) 通过索引存储将索引读入到内存。

e) 利用查询树搜索索引，从而得到每个词(Term)的文档链表，对文档链表进行交，差，并得到结果文档。

f) 将搜索到的结果文档对查询的相关性进行排序。

g) 返回查询结果给用户。

Lucene和ElasticSearch

lucene:全文搜索工具包，依赖于java，不适用于集群环境

ES：全文搜索服务器，基于lucene，采用Rest **HTTP调用**方式，对集群支持较好

分词器

WhitespaceAnalyzer

仅仅是去掉了空格，没有其他任何操作，不支持中文。

SimpleAnalyzer

讲除了字母以外的符号全部去除，并且讲所有字符变为小写，需要注意的是这个分词器同样把数据也去除了，同样不支持中文。

StopAnalyzer

这个和SimpleAnalyzer类似，不过比他增加了一个的是，在其基础上还去除了所谓的stop words，比如the, a, this这些。这个也是不支持中文的。

StandardAnalyzer

英文方面的处理和StopAnalyzer一样的，对中文支持，使用的是单字切割。

CJKAnalyzer

这个支持中日韩，前三个字母也就是这三个国家的缩写。这个对于中文基本上不怎么用吧，对中文的支持很烂，它是用每两个字作为分割，分割方式个人感觉比较奇葩，我会在下面比较举例。

SmartChineseAnalyzer

中文的分词。比较标准的中文分词，对一些搜索处理的并不是很好

大数据与高并发

一、秒杀架构设计

业务介绍



什么是秒杀？通俗一点讲就是网络商家为促销等目的组织的网上限时抢购活动

比如说京东秒杀，就是一种定时定量秒杀，在规定的时间内，无论商品是否秒杀完毕，该场次的秒杀活动都会结束。这种秒杀，对时间不是特别严格，只要下手快点，秒中的概率还是比较大的。

淘宝以前就做过一元抢购，一般都是限量1件商品，同时价格低到「令人发齿」，这种秒杀一般都在开始时间1到3秒内就已经抢光了，参与这个秒杀一般都是看运气的，不必太强求。

业务特点



瞬时并发量大

- 大量用户会在同一时间进行抢购
- 网站瞬时访问流量激增。



库存少

- 访问请求数量远远大于库存数量
- 只有少部分用户能够秒杀成功



业务流程简单

- 业务逻辑比较简单
- 下订单减库存

瞬时并发量大

秒杀时会有大量用户在同一时间进行抢购，瞬时并发访问量突增10倍，甚至100倍以上都有。

库存量少

一般秒杀活动商品量很少，这就导致了只有极少量用户能成功购买到。

业务简单

流程比较简单，一般都是下订单、扣库存、支付订单。

技术难点

对现有业务冲击

高并发应用负责高

突然增加网络与服务带宽

直接下单

控制商品页面购买按钮点亮

下单前置检查

©51CTO博客

现有业务的冲击

秒杀是营销活动中的一种，如果和其他营销活动应用部署在同一服务器上，肯定会对现有其他活动造成冲击，极端情况下可能导致整个电商系统服务宕机。

直接下订单

下单页面是一个正常的 URL 地址，需要控制在秒杀开始前，不能下订单，只能浏览对应活动商品的信息。简单来说，需要 Disable 订单按钮。

页面流量突增

秒杀活动开始前后，会有很多用户请求对应商品页面，会造成后台服务器的流量突增，同时对应的网络带宽增加，需要控制商品页面的流量不会对后台服务器、DB、Redis 等组件造成过大的压力

架构设计思想

限流

只有少部分用户能够秒杀成功，
所以要限制大部分流量，只允许少部分流量进入服务后端。

削峰

对于秒杀系统瞬时会有大量用户涌入，所以在抢购一开始会有很高的瞬间峰值。
高峰值流量是压垮系统很重要的原因，所以如何把瞬间的高流量变成一段时间平稳的流量也是设计秒杀系统很重要的思路。实现削峰的常用的方法有利用缓存和消息中间件等技术

异步处理

秒杀系统是一个高并发系统，采用异步处理模式可以极大地提高系统并发量，
其实异步处理就是削峰的一种实现方式

内存缓存

秒杀系统最大的瓶颈一般都是数据库读写，由于数据库读写属于磁盘IO，性能很低，如果能够把部分数据或业务逻辑转移到内存缓存，效率会有极大地提升

@51CTO博客

限流

由于活动库存量一般都是很少，对应的只有少部分用户才能秒杀成功。所以我们需要限制大部分用户流量，只准少量用户流量进入后端服务器。

削峰

秒杀开始的那一瞬间，会有大量用户冲击进来，所以在开始时候会有一个瞬间流量峰值。如何把瞬间的流量峰值变得更平缓，是能否成功设计好秒杀系统的关键因素。实现流量削峰填谷，一般的采用缓存和 MQ 中间件来解决。

异步

秒杀其实可以当做高并发系统来处理，在这个时候，可以考虑从业务上做兼容，将同步的业务，设计成异步处理的任务，提高网站的整体可用性。

缓存

秒杀系统的瓶颈主要体现在下订单、扣减库存流程中。在这些流程中主要用到 OLTP 的数据库，类似 MySQL、SQLServer、Oracle。由于数据库底层采用 B+ 树的储存结构，对应我们随机写入与读取的效率，相对较低。如果我们把部分业务逻辑迁移到内存的缓存或者 Redis 中，会极大的提高并发效率。

整体架构



客户端优化

客户端优化主要有两个问题：

秒杀页面

秒杀活动开始前，其实就有很多用户访问该页面了。如果这个页面的一些资源，比如 CSS、JS、图片、商品详情等，都访问后端服务器，甚至 DB 的话，服务肯定会出现不可用的情况。所以一般我们会把这个页面整体进行静态化，并将页面静态化之后的页面分发到 CDN 边缘节点上，起到压力分散的作用。

防止提前下单

防止提前下单主要是在静态化页面中加入一个 JS 文件引用，该 JS 文件包含活动是否开始的标记以及开始时的动态下单页面的 URL 参数。同时，这个 JS 文件是不会被 CDN 系统缓存的，会一直请求后端服务的，所以这个 JS 文件一定要很小。当活动快开始的时候（比如提前），通过后台接口修改这个 JS 文件使之生效。

API 接入层优化

客户端优化，对于不是搞计算机方面的用户还是可以防止住的。但是稍有一定网络基础的用户就起不到作用了，因此服务端也需要加些对应控制，不能信任客户端的任何操作。一般控制分为 2 大类：

限制用户维度访问频率

针对同一个用户（Userid 维度），做页面级别缓存，单元时间内的请求，统一走缓存，返回同一个页面。

限制商品维度访问频率

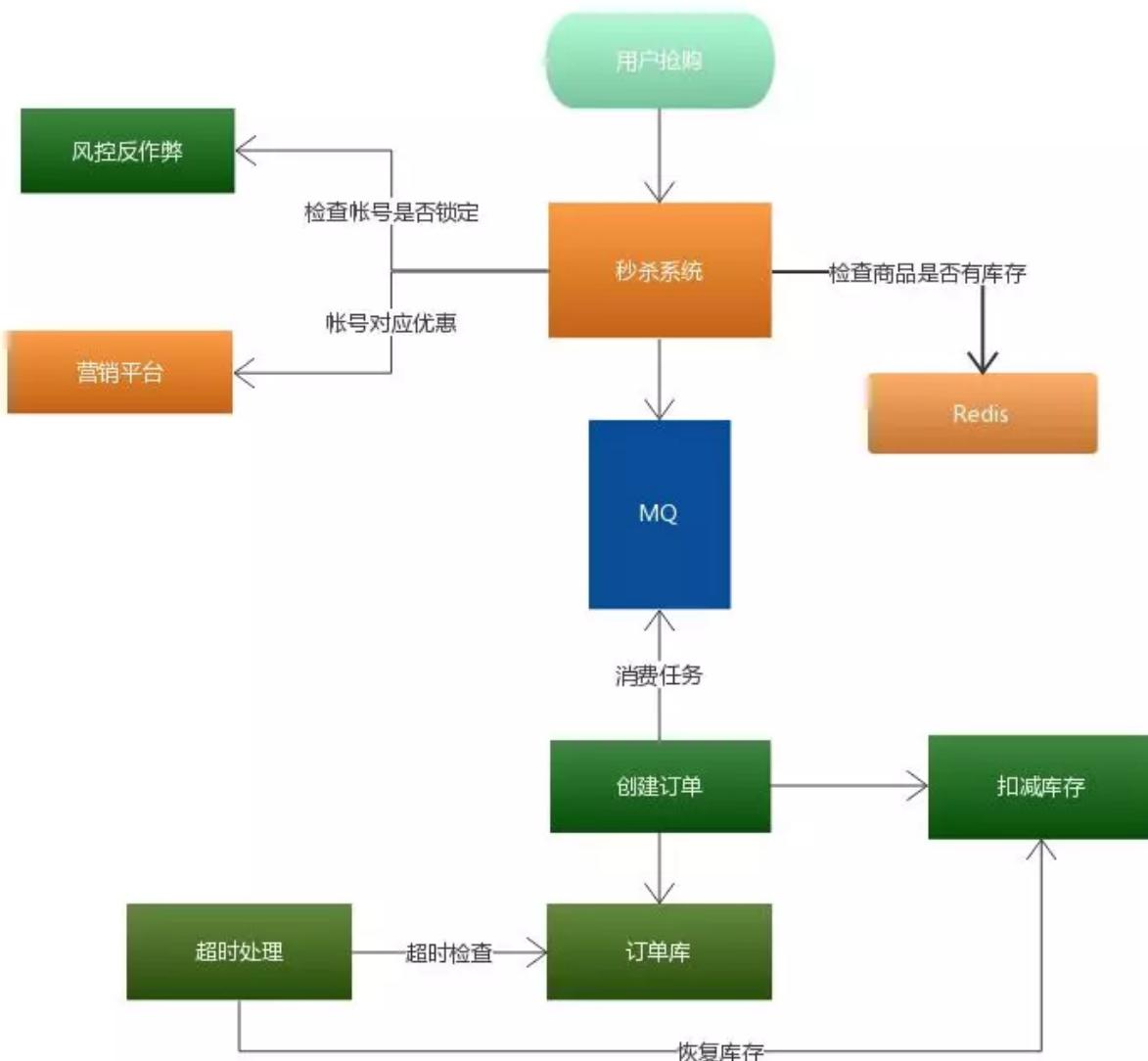
大量请求同时间段查询同一个商品时，可以做页面级别缓存，不管下回是谁来访问，只要是这个页面就直接返回。

SOA 服务层优化

上面两层只能限制异常用户访问，如果秒杀活动运营的比较好，很多用户都参加了，就会造成系统压力过大甚至宕机，因此需要后端流量控制。

对于后端系统的控制可以通过消息队列、异步处理、提高并发等方式解决。对于超过系统水位线的请求，直接采取「Fail-Fast」原则，拒绝掉。

秒杀整体流程图



秒杀系统核心在于层层过滤，逐渐递减瞬时访问压力，减少最终对数据库的冲击。通过上面流程图就会发现压力最大的地方在哪里？

MQ 排队服务，只要 MQ 排队服务顶住，后面下订单与扣减库存的压力都是自己能控制的，根据数据库的压力，可以定制化创建订单消费者的数量，避免出现消费者数据量过多，导致数据库压力过大或者直接宕机。

库存服务专门为秒杀的商品提供库存管理，实现提前锁定库存，避免超卖的现象。同时，通过超时处理任务发现已抢到商品，但未付款的订单，并在规定付款时间后，处理这些订单，将恢复订单商品对应的库存量。

总结

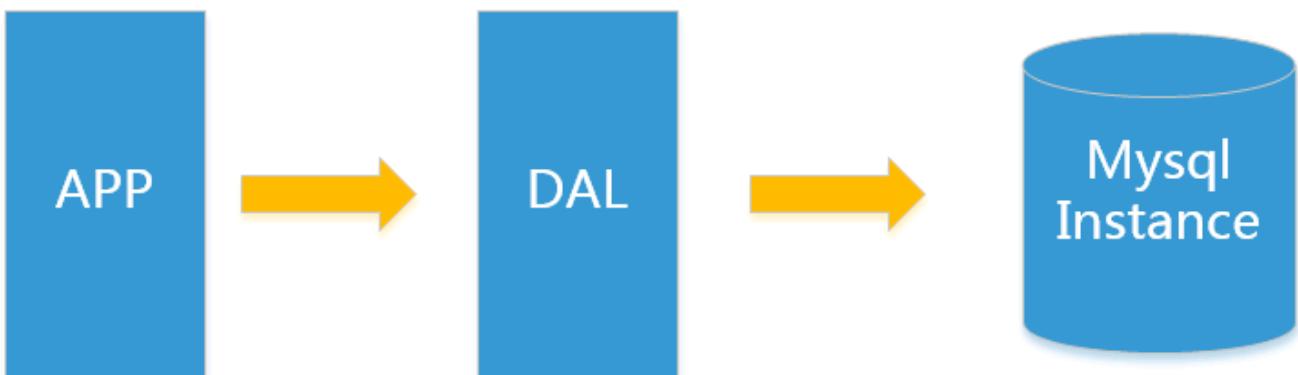
核心思想：层层过滤

- 尽量将请求拦截在上游，降低下游的压力
- 充分利用缓存与消息队列，提高请求处理速度以及削峰填谷的作用

二、数据库架构发展历程

单机MySQL的美好年代

在90年代，一个网站的访问量一般都不大，用单个数据库完全可以轻松应付。在那个时候，更多的都是静态网页，动态交互类型的网站不多。

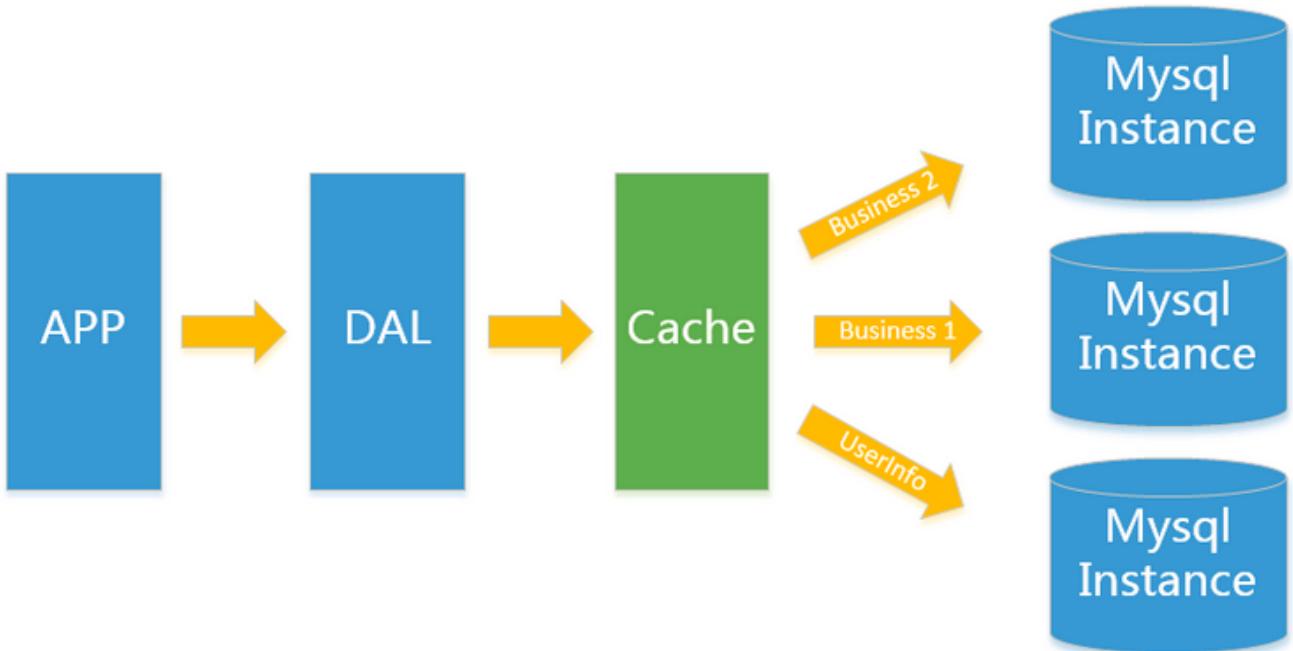


上述架构下，我们来看看数据存储的瓶颈是什么？

1. 数据量的总大小 一个机器放不下时
2. 数据的索引 (B+ Tree) 一个机器的内存放不下时
3. 访问量(读写混合)一个实例不能承受

Memcached(缓存)+MySQL+垂直拆分

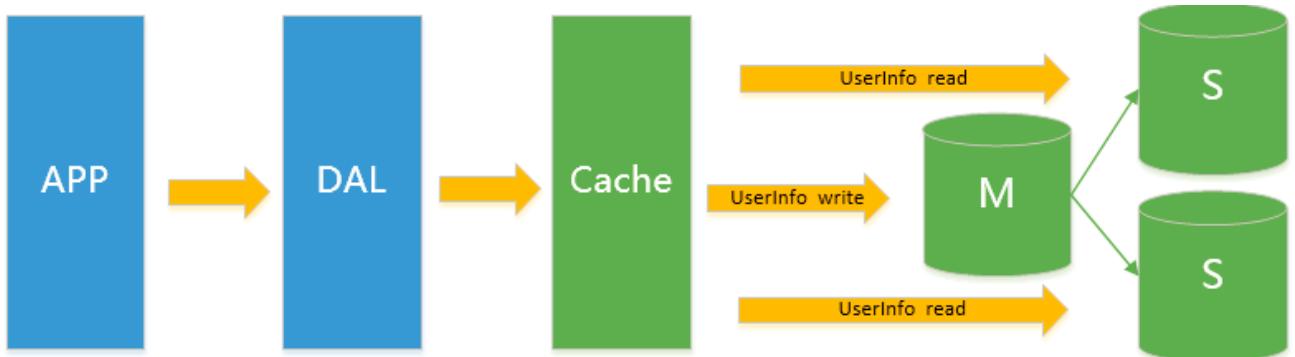
后来，随着访问量的上升，几乎大部分使用MySQL架构的网站在数据库上都开始出现了性能问题，web程序不再仅仅专注在功能上，同时也在追求性能。程序员们开始大量的使用缓存技术来缓解数据库的压力，优化数据库的结构和索引。开始比较流行的是通过文件缓存来缓解数据库压力，但是当访问量继续增大的时候，多台web机器通过文件缓存不能共享，大量的小文件缓存也带了了比较高的IO压力。在这个时候，Memcached就自然的成为一个非常时尚的技术产品。



Memcached作为一个独立的分布式的缓存服务器，为多个web服务器提供了一个共享的高性能缓存服务，在Memcached服务器上，又发展了根据hash算法来进行多台Memcached缓存服务的扩展，然后又出现了一致性hash来解决增加或减少缓存服务器导致重新hash带来的大量缓存失效的弊端

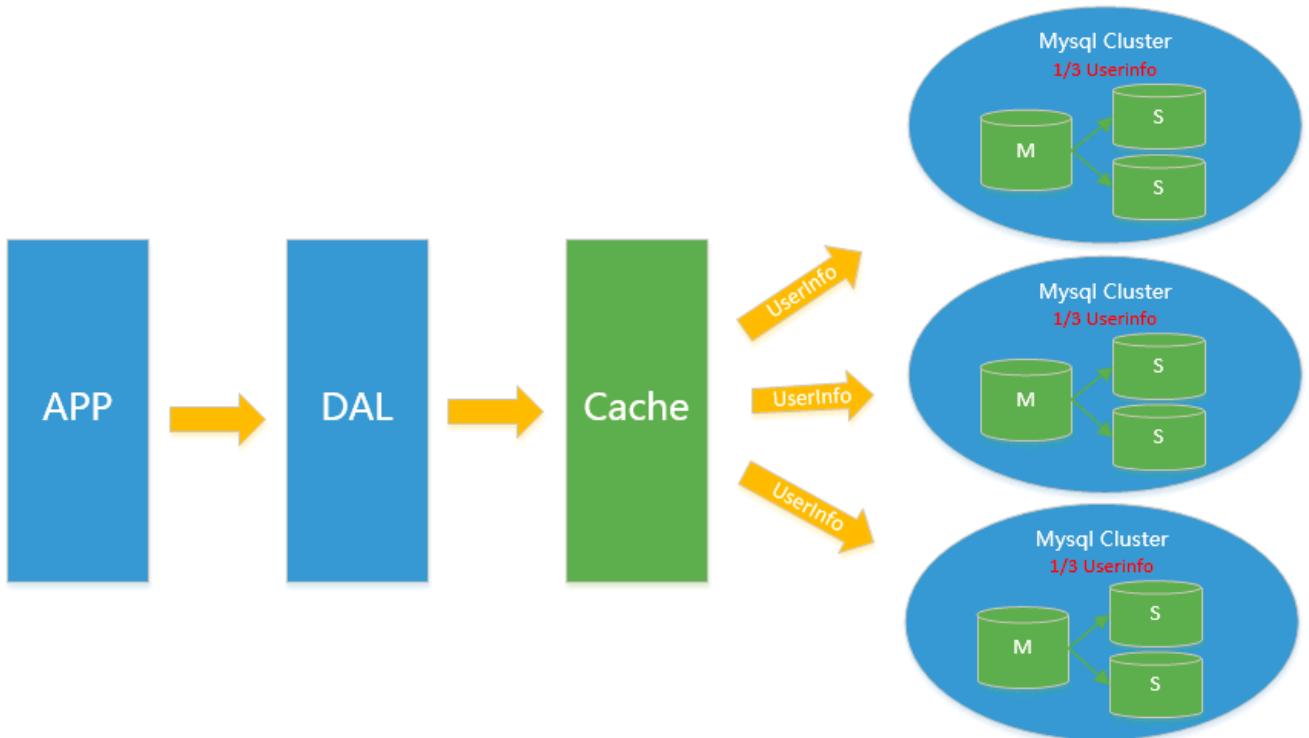
MySQL主从复制读写分离

由于数据库的写入压力增加，Memcached只能缓解数据库的读取压力。读写集中在一个数据库上让数据库不堪重负，大部分网站开始使用主从复制技术来达到读写分离，以提高读写性能和读库的可扩展性。Mysql的master-slave模式成为这个时候的网站标配了。



分表分库+水平拆分+mysql集群

在Memcached的高速缓存，MySQL的主从复制，读写分离的基础之上，这时MySQL主库的写压力开始出现瓶颈，而数据量的持续猛增，由于MyISAM使用表锁，在高并发下会出现严重的锁问题，大量的高并发MySQL应用开始使用InnoDB引擎代替MyISAM。



同时，开始流行使用分表分库来缓解写压力和数据增长的扩展问题。这个时候，分表分库成了一个热门技术，是面试的热门问题也是业界讨论的热门技术问题。也就在这个时候，MySQL推出了还不太稳定的表分区，这也给技术实力一般的公司带来了希望。虽然MySQL推出了MySQL Cluster集群，但性能也不能很好满足互联网的要求，只是在高可靠性上提供了非常大的保证。

三、MySQL的扩展性瓶颈

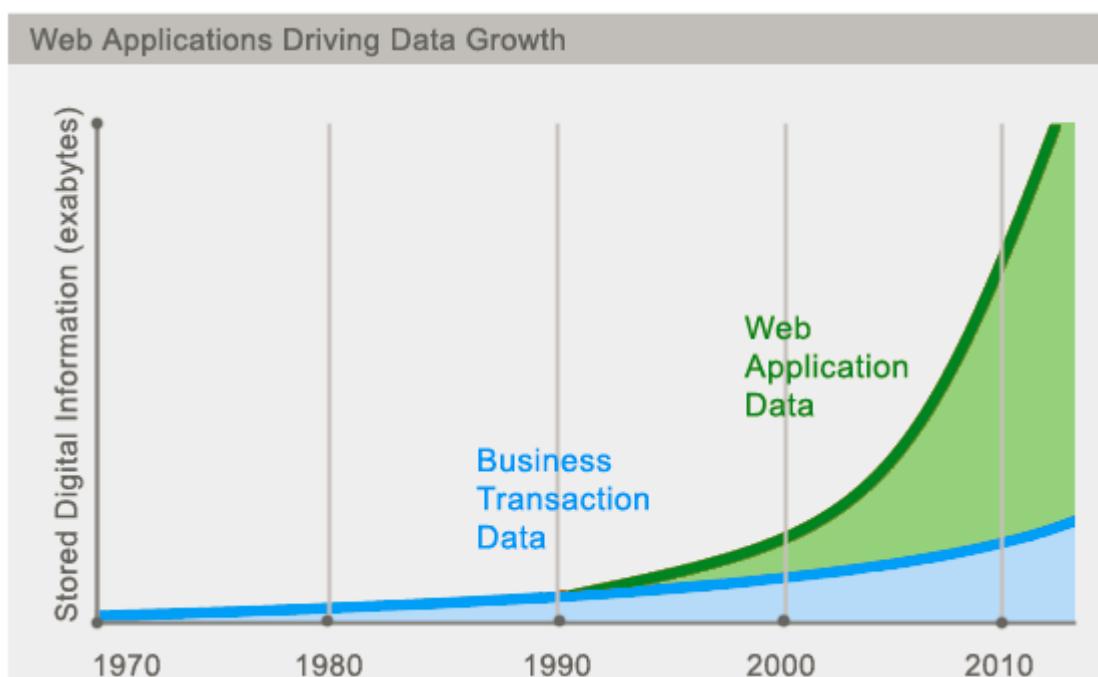
MySQL数据库也经常存储一些大文本字段，导致数据库表非常的大，在做数据库恢复的时候就导致非常的慢，不容易快速恢复数据库。比如1000万4KB大小的文本就接近40GB的大小，如果能把这些数据从MySQL省去，MySQL将变得非常的小。关系数据库很强大，但是它并不能很好的应付所有的应用场景。MySQL的扩展性差（需要复杂的技术来实现），大数据下IO压力大，**表结构更改困难**，正是当前使用MySQL的开发人员面临的问题。

alter table add Column

四、为什么要使用NOSQL NOT ONLY SQL

为什么使用NoSQL？

今天我们可以很容易的访问和抓取数据。用户的个人信息，社交网络，地理位置，用户生成的数据和用户操作日志已经成倍的增加。我们如果要对这些用户数据进行挖掘，那SQL数据库已经不适合这些应用了，NoSQL数据库的发展也却能很好的处理这些大的数据。



NoSQL(NoSQL = Not Only SQL)，意即“不仅仅是SQL”，泛指非关系型的数据库。随着互联网web2.0网站的兴起，传统的关系数据库在应付web2.0网站，特别是超大规模和高并发的SNS类型的web2.0纯动态网站已经显得力不从心，暴露了很多难以克服的问题，而非关系型的数据库则由于其本身的特点得到了非常迅速的发展。NoSQL数据库的产生就是为了解决大规模数据集合多重数据种类带来的挑战，尤其是大数据应用难题，包括超大规模数据的存储。

五、传统RDBMS VS NOSQL

RDBMS vs NoSQL

RDBMS SQL

- 高度组织化结构化数据
- 结构化查询语言 (SQL)
- 数据和关系都存储在单独的表中。
- 数据操纵语言，数据定义语言

- 严格的一致性
- 基础事务

NoSQL

- 代表着不仅仅是SQL
- 没有声明性查询语言
- 没有预定义的模式
-键 - 值对存储, 列存储, 文档存储, 图形数据库
- 最终一致性, 而非ACID属性
- 非结构化和不可预知的数据
- CAP定理
- 高性能, 高可用性和可伸缩性

六、NoSQL数据库的类型

分类	Examples举例	典型应用场景	数据模型	优点	缺点
键值 (key-value) [3]	Tokyo Cabinet/Tyrant, Redis, Voldemort, Oracle BDB	内容缓存, 主要用于处理大量数据的高访问负载, 也用于一些日志系统等等。 [3]	Key 指向 Value 的键值对, 通常用 hash table 来实现 [3]	查找速度快	数据无结构化, 通常只被当作字符串或者二进制数据 [3]
列存储数据库 [3]	Cassandra, HBase, Riak	分布式的文件系统	以列簇式存储, 将同一列数据存在一起	查找速度快, 可扩展性强, 更容易进行分布式扩展	功能相对局限
文档型数据库 [3]	CouchDB, MongoDB	Web 应用 (与 Key-Value 类似, Value 是结构化的, 不同的是数据库能够了解 Value 的内容)	Key-Value 对应的键值对, Value 为结构化数据	数据结构要求不严格, 表结构可变, 不需要像关系型数据库一样需要预先定义表结构	查询性能不高, 而且缺乏统一的查询语法。
图形 (Graph)数据库 [3]	Neo4J, InfoGrid, Infinite Graph	社交网络, 推荐系统等。专注于构建关系图谱	图结构	利用图结构相关算法。比如最短路径寻址, N 度关系查找等	很多时候需要对整个图做计算才能得出需要的信息, 而且这种结构不太好做分布式的集群方案。 [3]

七、阿里巴巴中文站商品信息如何存放

看看阿里巴巴中文网站首页 以女装/女包包为例

The screenshot shows the 1688.com homepage. At the top, there's a navigation bar with links like '我的阿里', '进货单 0', '收藏夹', '诚信通服务', '实力商家', '我是供应商', '客服中心', and '网站导航'. Below the navigation is a search bar with a QR code icon. The main content area features a large promotional banner for '2015 淘货源 试卖包销专区'. To the left is a sidebar with a '行业市场' section containing various industry categories such as女装/男装/内衣、鞋靴/箱包/配饰等. On the right, there's a user profile section with a welcome message 'Hi, 早上好 欢迎来到 1688.com', login and registration buttons, and a sidebar for announcements and statistics.

商品基本信息

名称、价格，出厂日期，生产厂商等

关系型数据库：mysql/oracle目前淘宝在去O化(也即拿掉Oracle)，注意，淘宝内部用的Mysql是里面的大牛自己改造过的

为什么去IOE

IBM小型机 廉价的PC机

oracle数据库 mysql

EMC存储

集中式----->分布式



2008年，王坚加盟阿里巴巴成为集团首席架构师，即现在的首席技术官。这位前微软亚洲研究院常务副院长被马云定位为：将帮助阿里巴巴集团建立世界级的技术团队，并负责集团技术架构以及基础技术平台搭建。

在加入阿里后，带着技术基因和学者风范的王坚就在阿里巴巴集团提出了被称为“去IOE”（在IT建设过程中，去除IBM小型机、Oracle数据库及EMC存储设备）的想法，并开始把云计算的本质，植入阿里IT基因。

王坚这样概括“去IOE”运动和阿里云之间的关系：“去IOE”彻底改变了阿里集团IT架构的基础，是阿里拥抱云计算，产出计算服务的基础。“去IOE”的本质是分布化，让随处可以买到的Commodity PC架构成为可能，使云计算能够落地的首要条件。

推荐阅读： [《阿里云这群疯子》](#)

飞天的内核

- 盘古：分布式存储
- 伏羲：任务调度
- 有巢：结构化存储与处理
- 夸父：网络通信
- 女娲：命名与协同
- 神农：监控
- 大禹：部署
- 钟馗：安全管理

商品描述、详情、评价信息(多文字类)

多文字信息描述类，IO读写性能变差

文档数据库MongoDB中

商品的图片

流媒体服务器

分布式的文件系统中

淘宝自己的TFS

Google的GFS

Hadoop的HDFS

商品的关键字

搜索引擎，淘宝内用

ISearch

扫地僧



多隆在阿里的层级是 P11，相当于副总裁。刚来阿里的时候，我以为专家组，一定是都是高 P 的大团队。哪知道进来发现，多隆下面包括我，仅有 3 个下属，其中一位师兄还长期在北京。每天中午一起吃饭，可以当团建，吃完饭一起散步，就算是 outing 了。

多隆不爱带团队，团队一般沟通成本高、水平参差不齐，而他一个人就能顶一个高效顶尖的团队（所以每次问他问题打断他，我都深深内疚，感觉拖了阿里的后腿）。作为淘宝最早的程序员之一，很多产品早期就是他一个人开发维护的，文件系统 tfs、key-value 系统 tair，cache、搜索、通讯框架等等，引用行颠对他的评价：

在内网的标签上，他被称为神，这不是恭维，在所有工程师眼中，他就是个神。多隆做事一个人能顶一个团队，比如说写一个文件系统，别人很可能是一个项目组，甚至一个公司在做，而他从头到尾都是一个人，在很短的时间内就完成了。从 03 年到 07 年，淘宝搜索引擎就是他一个人在写，一个人在维护，而且这还不是他全部的工作，另外他还做了其他很多事情。

商品的波段性的热点高频信息

内存数据库

Tair、Redis、Memcache

商品的交易、价格计算、积分累计

外部系统，外部第3方支付接口

支付宝

大型互联网应用(大数据、高并发、多样数据类型)的难点和解决方案

难点

数据类型多样性

数据源多样性和变化重构

数据源改造而数据服务平台不需要大面积重构

解决办法

阿里、淘宝干了什么？UDSL

是什么

数据层

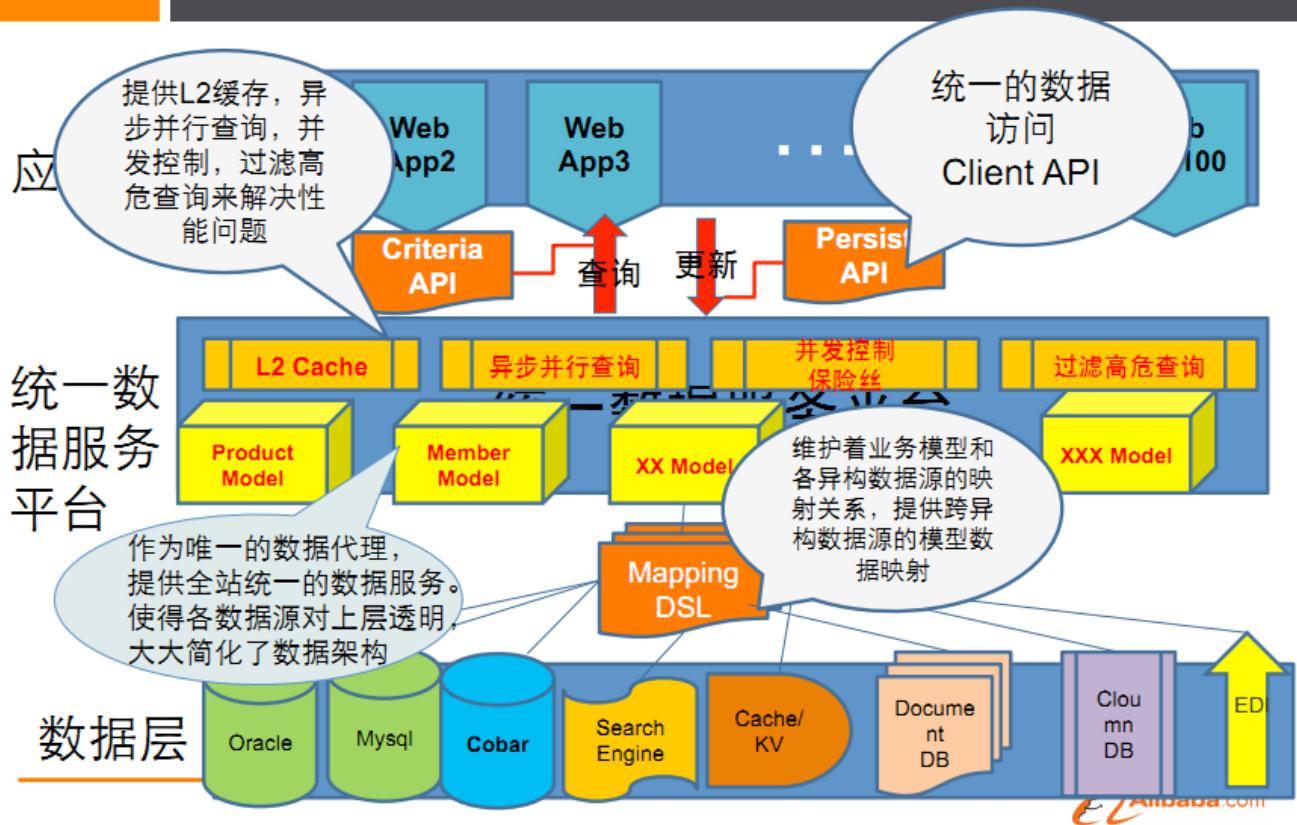
解决方案：统一数据服务层UDSL

网站应用层

- 在网站应用集群和底层数据源之间，构建一层代理，统一数据层
- 统一数据层的特性
 - 模型数据映射
 - 实现 业务模型 各属性 与 底层不同类型数据源的 模型数据映射
 - 目前支持关系数据库，iSearch, redis, mongodb
 - 统一的查询和更新API
 - 提供了基于业务模型的统一的查询和更新的API，简化网站应用跨不同数据源的开发模式。
 - 性能优化策略
 - 字段延迟加载,按需返回设置
 - 基于热点缓存平台的二级缓存。
 - 异步并行的查询数据:异步并行加载模型中来自不同数据源的字段
 - 并发保护：拒绝访问频率过高的主机IP或IP段
 - 过滤高危的查询：例如会导致数据库崩溃的全表扫描



什么样



- 问题

- 传统O/R mapping框架，不能实现非关系数据库和跨不同类型数据库的对象数据映射
 - 早期 O/R Mapping 框架，简化了对关系数据库的查询方式，采用面向对象的方式查询管理数据，但对非关系数据库类型，不能实现对象数据映射，更不能跨不同数据源映射
- UDSL提供除关系数据库以外的数据源的对象关系映射
 - 目前支持 关系数据库, iSearch, Mongodb, Redis
- UDSL支持同一业务模型的各字段映射到不同的数据
- **设计1：模型数据映射 DSL，定义模型字段和底层数据库的映射关系**
 - 我们设计了一套DSL，
 - 描述模型对应哪些类型的数据源，各个数据源的访问方式是什么
 - 模型有哪些字段，每个字段对应哪个数据源的哪个字段或哪个数据接口方法
 - 模型字段是否延迟加载
 - 模型字段的值如果需要对原始数据进行逻辑处理，用何种方式逻辑处理
 - UDSL阅读DSL定义的数据路由，访问不同数据源，组装模型
 - 通过这套DSL我们定义模型各字段和各数据源的数据路由, UDSL在查询数据时通过阅读DSL路由规则来聚合各数据源数据，组装模型。



API

- UDSL采用统一的查询/更新API，统一了不同数据源的查询方式

- **查询API (Criteria API)：** 提供类似JPA

- 基于模型表达式的查询方式：
 - 商品模型的行业属性是否等于“服装”
 - 支持按模型属性的结果排序
 - 支持限定结果返回条数和返回哪些字段
 - 和JPA非常不同的是：
 - 支持按需加载：可以指定只返回哪些字段，或者过滤哪些字段。

方法	说明
find	查找
orderBy	排序
limit	约束

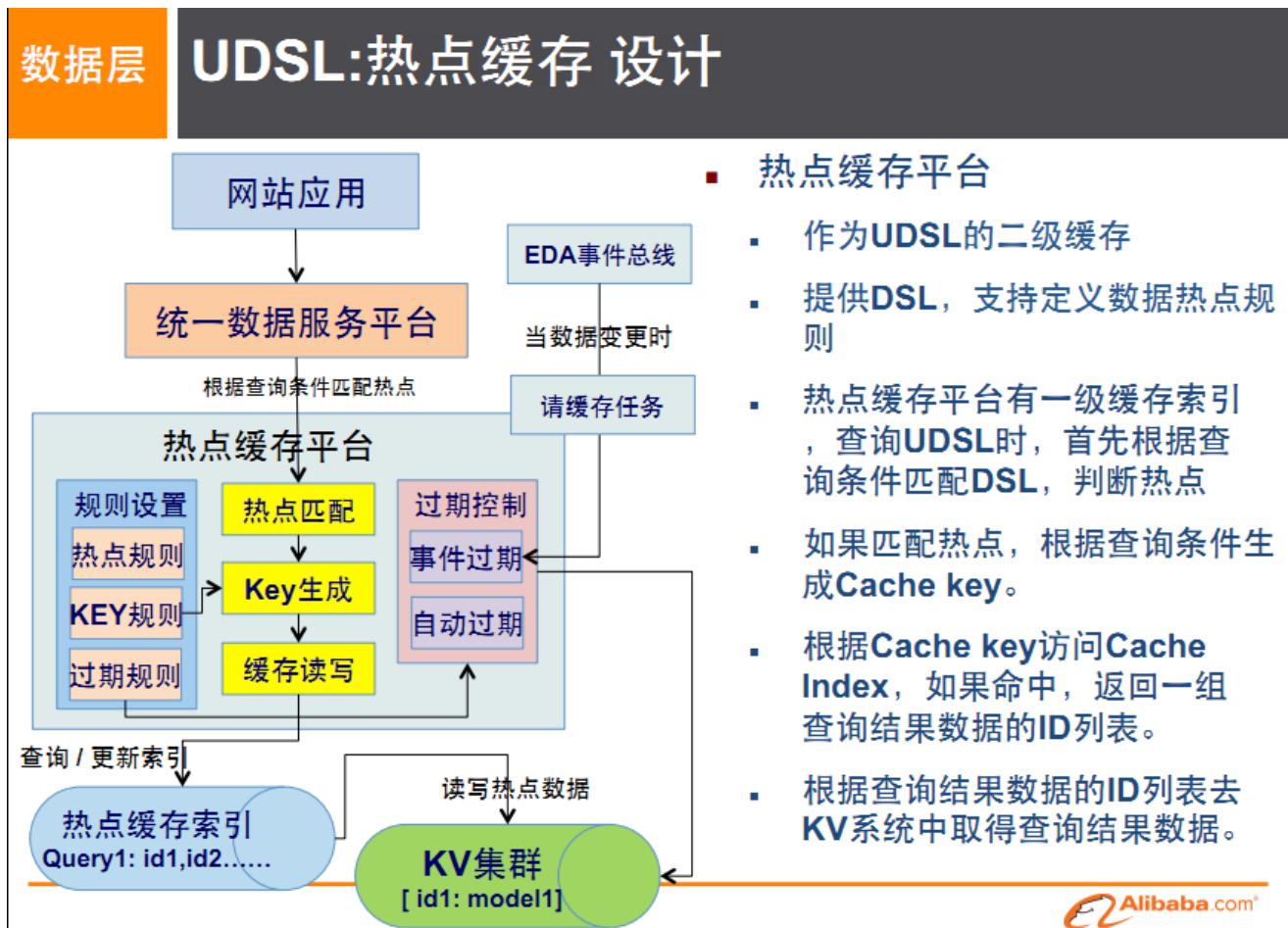
表达式	说明	示例
eq	等于	object1.field1.eq ("xxx")
gt	大于	object1.field2.gt (100)
desc	倒序	object1.field3.desc

- **持久化API (Persist API)：** 提供简单的接口

- 只有四个方法：`insert`, `update`, `delete`, `insertOrUpdate`，参数就是数据对象，非常简单
 - `Dml.insert(product)`, `Dml.update(product)`, `Dml.delete(product)`
 - `Dml.insertOrUpdate(product)`

- UDSL自动的分析查询/更新参数，并根据模型字段映射配置，转换成底层各数据源的native语句进行数据操作





八、数据的水平拆分和垂直拆分

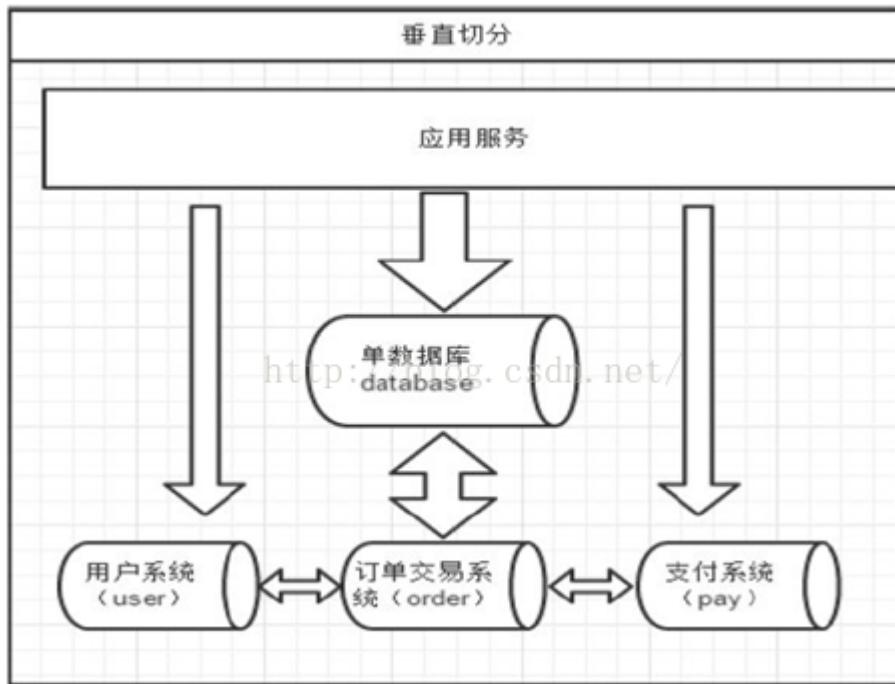
当我们使用读写分离、缓存后，数据库的压力还是很大的时候，这就需要使用到数据库拆分了。

数据库拆分简单来说，就是指通过某种特定的条件，按照某个维度，将我们存放在同一个数据库中的数据分散存放到底个数据库（主机）上面以达到分散单库（主机）负载的效果。

切分模式：垂直（纵向）拆分、水平拆分。

垂直拆分

一个数据库由很多表的构成，每个表对应着不同的业务，垂直切分是指按照业务将表进行分类，分布到不同的数据库上面，这样也就将数据或者说压力分担到不同的库上面，如下图：



优点:

1. 拆分后业务清晰，拆分规则明确。
2. 系统之间整合或扩展容易。
3. 数据维护简单。

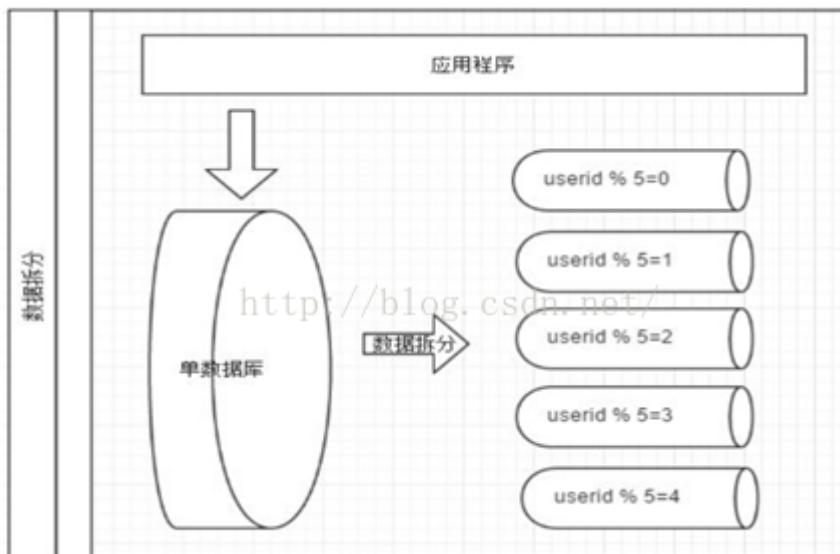
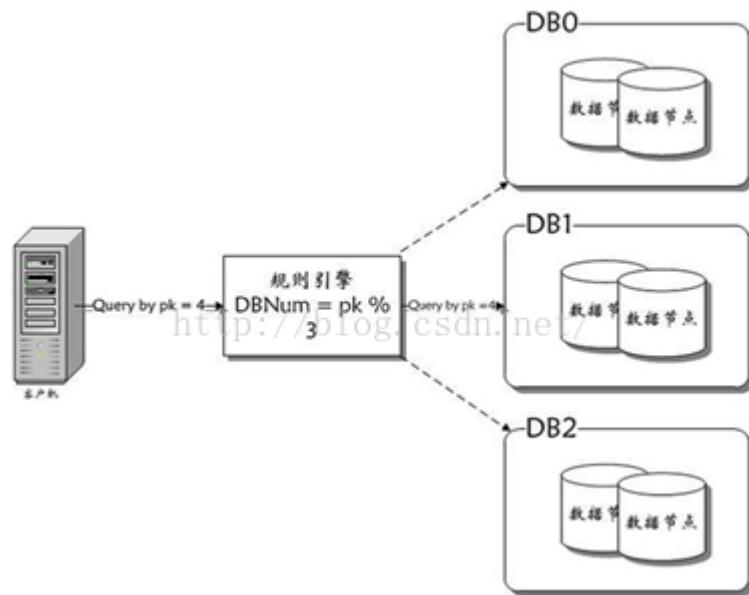
缺点:

1. 部分业务表无法join，只能通过接口方式解决，提高了系统复杂度。
2. 受每种业务不同的限制存在单库性能瓶颈，不易数据扩展跟性能提高。
3. 事务处理复杂。

水平拆分

垂直拆分后遇到单机瓶颈，可以使用水平拆分。相对于垂直拆分的区别是：垂直拆分是把不同的表拆到不同的数据库中，而水平拆分是把同一个表拆到不同的数据库中。

相对于垂直拆分，水平拆分不是将表的数据做分类，而是按照某个字段的某种规则来分散到多个库之中，每个表中包含一部分数据。简单来说，我们可以将数据的水平切分理解为是按照数据行的切分，就是将表中的某些行切分到一个数据库，而另外的某些行又切分到其他的数据库中，主要有分表、分库两种模式，如图：



优点:

1. 不存在单库大数据，高并发的性能瓶颈。
2. 对应用透明，应用端改造较少。
3. 按照合理拆分规则拆分，join操作基本避免跨库。
4. 提高了系统的稳定性跟负载能力。

缺点:

1. 拆分规则难以抽象。
2. 分片事务一致性难以解决。
3. 数据多次扩展难度跟维护量极大。
4. 跨库join性能较差。

拆分的处理难点

两张方式共同缺点

1. 引入分布式事务的问题。
2. 跨节点Join 的问题。
3. 跨节点合并排序分页问题。

针对数据源管理，目前主要有两种思路：

A. 客户端模式，在每个应用程序模块中配置管理自己需要的一个（或者多个）数据源，直接访问各个数据库，在模块内完成数据的整合。

优点：相对简单，无性能损耗。

缺点：不够通用，数据库连接的处理复杂，对业务不够透明，处理复杂。

B. 通过中间代理层来统一管理所有的数据源，后端数据库集群对前端应用程序透明；

优点：通用，对应用透明，改造少。

缺点：实现难度大，有二次转发性能损失。

拆分原则

1. 尽量不拆分，架构是进化而来，不是一蹴而就。(SOA)
2. 最大可能的找到最合适切分维度。
3. 由于数据库中间件对数据Join 实现的优劣难以把握，而且实现高性能难度极大，业务读取尽量少使用多表Join - 尽量通过数据冗余，分组避免数据跨库多表join。
4. 尽量避免分布式事务。
5. 单表拆分到数据1000万以内。

切分方

范围、枚举、时间、取模、哈希、指定等

案例分析

场景一

建立一个历史his系统，将公司的一些历史个人游戏数据保存到这个his系统中，主要是写入，还有部分查询，读写比约为1:4；由于是所有数据的历史存取，所以并发要求比较高；

分析：

历史数据

写多都少

越近日期查询越频繁？

什么业务数据？用户游戏数据

有没有大规模分析查询？

数据量多大？

保留多久?

机器资源有多少?

方案1: 按照日期每月一个分片

带来的问题: 1.数据热点问题 (压力不均匀)

方案2: 按照用户取模, --by Jerome 就这个比较合适了

带来的问题: 后续扩容困难

方案3: 按用户ID范围分片 (1-1000万=分片1, xxx)

带来的问题: 用户活跃度无法掌握, 可能存在热点问题

场景二

建立一个商城订单系统, 保存用户订单信息。

分析:

电商系统

一号店或京东类? 淘宝或天猫?

实时性要求高

存在瞬时压力

基本不存在大规模分析

数据规模?

机器资源有多少?

维度? 商品? 用户? 商户?

方案1: 按照用户取模,

带来的问题: 后续扩容困难

方案2: 按用户ID范围分片 (1-1000万=分片1, xxx)

带来的问题: 用户活跃度无法掌握, 可能存在热点问题

方案3: 按省份地区或者商户取模

数据分配不一定均匀

场景3

上海公积金, 养老金, 社保系统

分析:

社保系统

实时性要求不高

不存在瞬时压力

大规模分析?

数据规模大

数据重要不可丢失

偏于查询？

方案1：按照用户取模，

带来的问题：后续扩容困难

方案2：按用户ID范围分片 (1-1000万=分片1, xxx)

带来的问题：用户活跃度无法掌握，可能存在热点问题

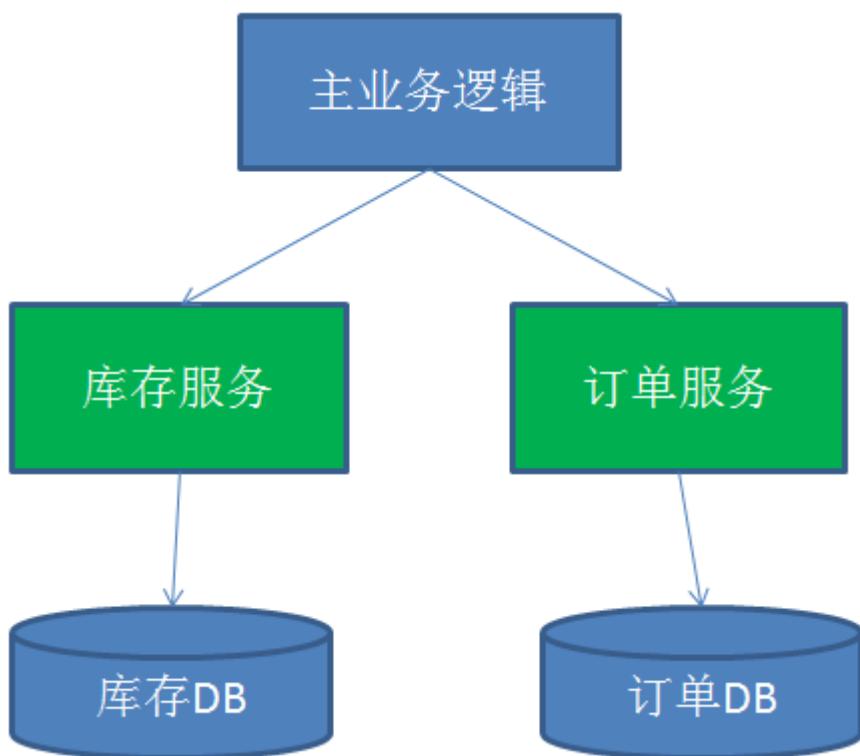
方案3：按省份区县地区枚举

数据分配不一定均匀

九、分布式事务

假如没有分布式事务

在一系列微服务系统当中，假如不存在分布式事务，会发生什么呢？让我们以互联网中常用的交易业务为例子：



上图中包含了库存和订单两个独立的微服务，每个微服务维护了自己的数据库。在交易系统的业务逻辑中，一个商品在下单之前需要先调用库存服务，进行扣除库存，再调用订单服务，创建订单记录。

正常情况下，两个数据库各自更新成功，两边数据维持着一致性。



A商品的总库存-1
(执行成功)



插入一条A商品的订单
(执行成功)

但是，在非正常情况下，有可能库存的扣减完成了，随后的订单记录却因为某些原因插入失败。这个时候，两边数据就失去了应有的一致性。



A商品的总库存-1
(执行成功)



插入一条A商品的订单
(执行失败)

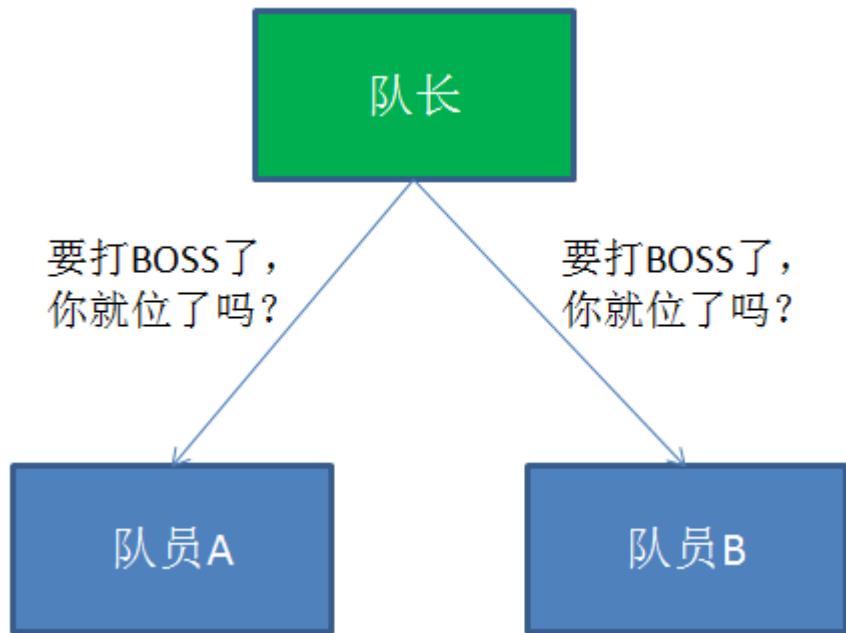
什么是分布式事务？

分布式事务用于在分布式系统中保证不同节点之间的数据一致性。分布式事务的实现有很多种，最具有代表性的是由Oracle Tuxedo系统提出的XA分布式事务协议。

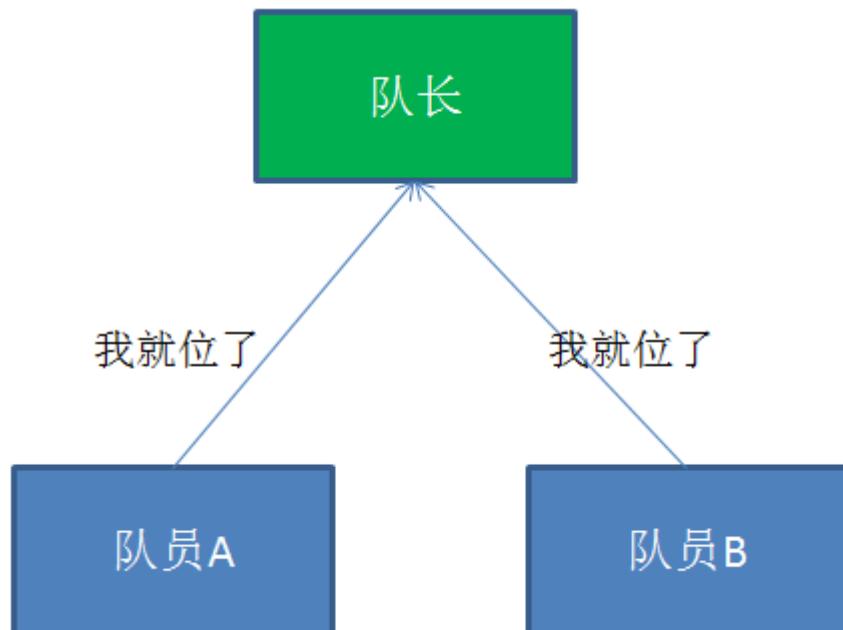
XA协议包含**两阶段提交（2PC）**和**三阶段提交（3PC）**两种实现，这里我们重点介绍两阶段提交的具体过程。

XA两阶段提交（2PC）

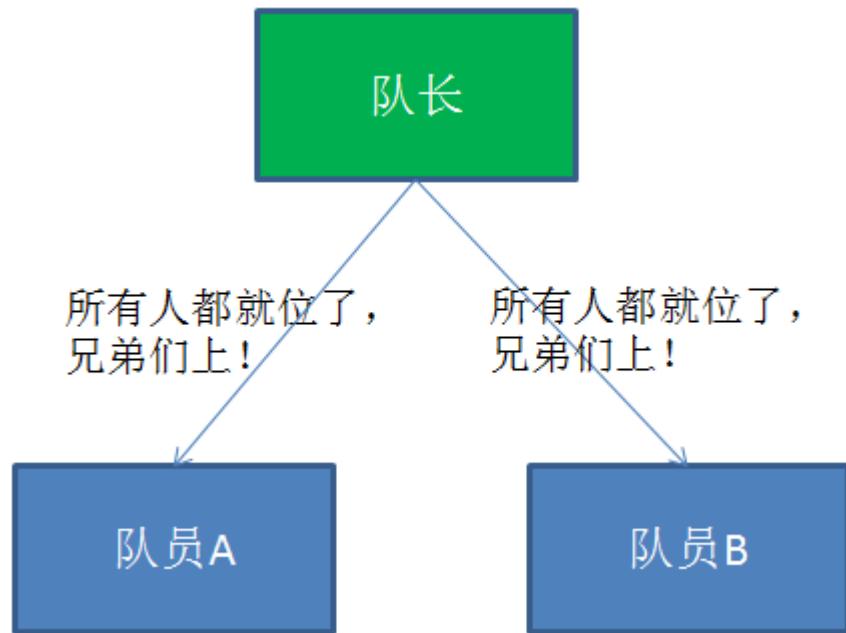
在魔兽世界这款游戏中，副本组团打BOSS的时候，为了更方便队长与队员们之间的协作，队长可以发起一个“就位确认”的操作：



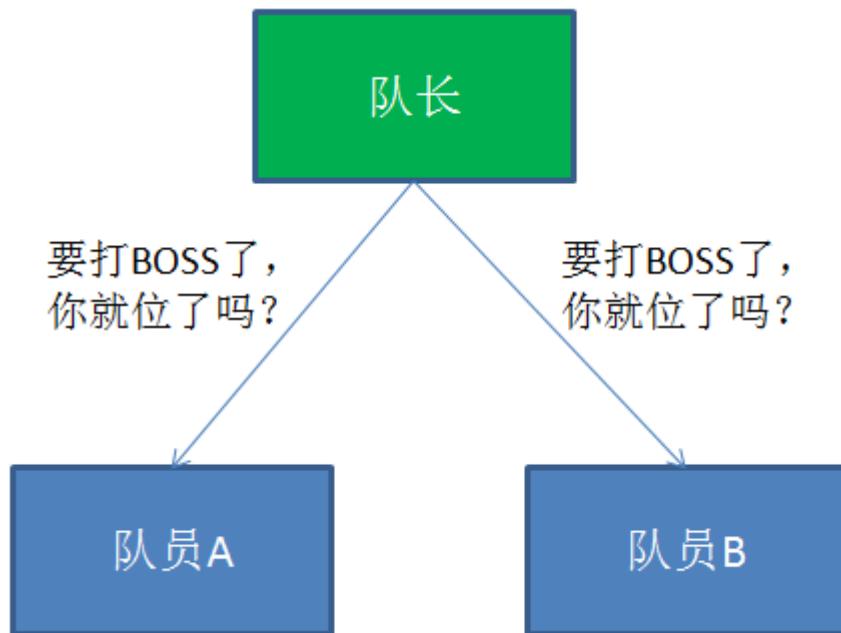
当队员收到就位确认提示后，如果已经就位，就选择“是”，如果还没就位，就选择“否”。

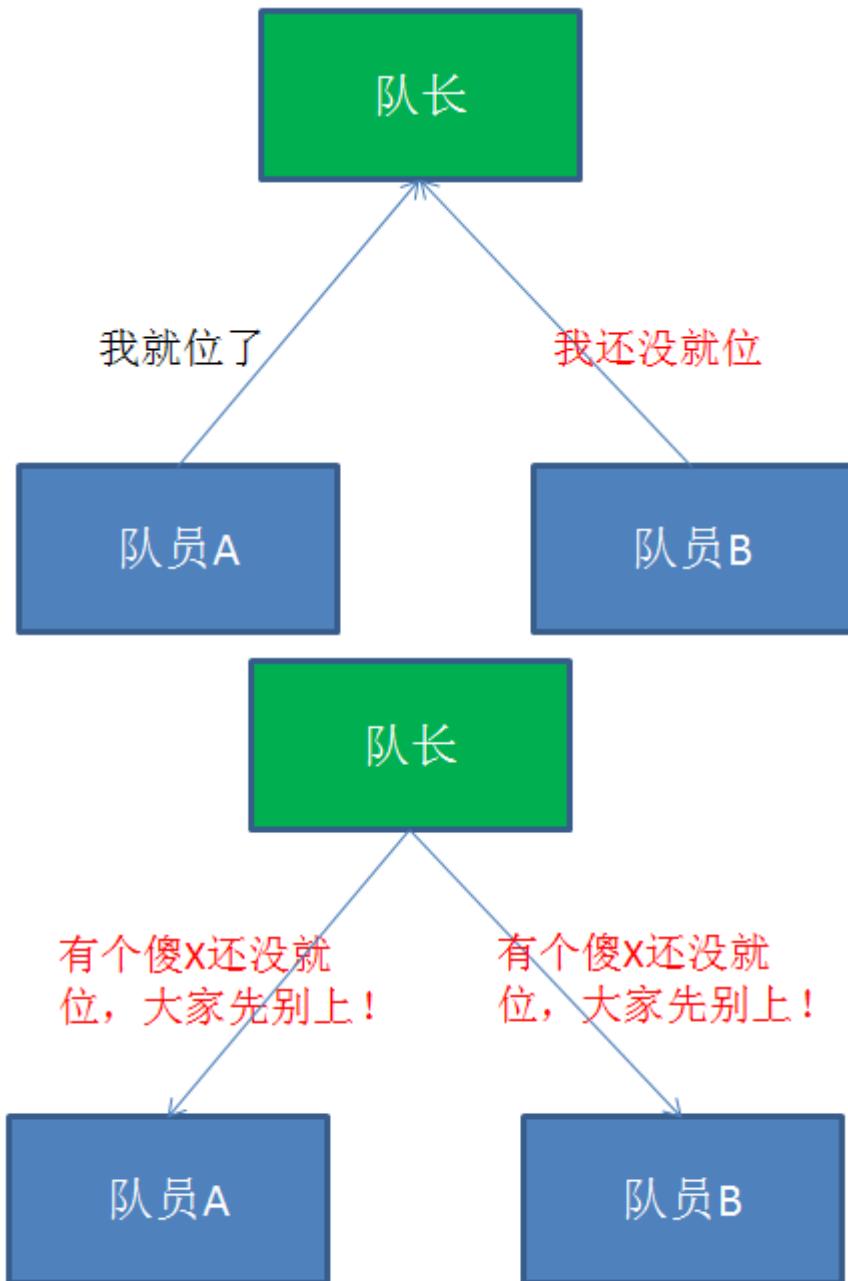


当队长收到了所有人的就位确认，就会向所有队员们发布消息，告诉他们开始打BOSS。



相应的，在队长发起就位确认的时候，有可能某些队员还并没有就位：

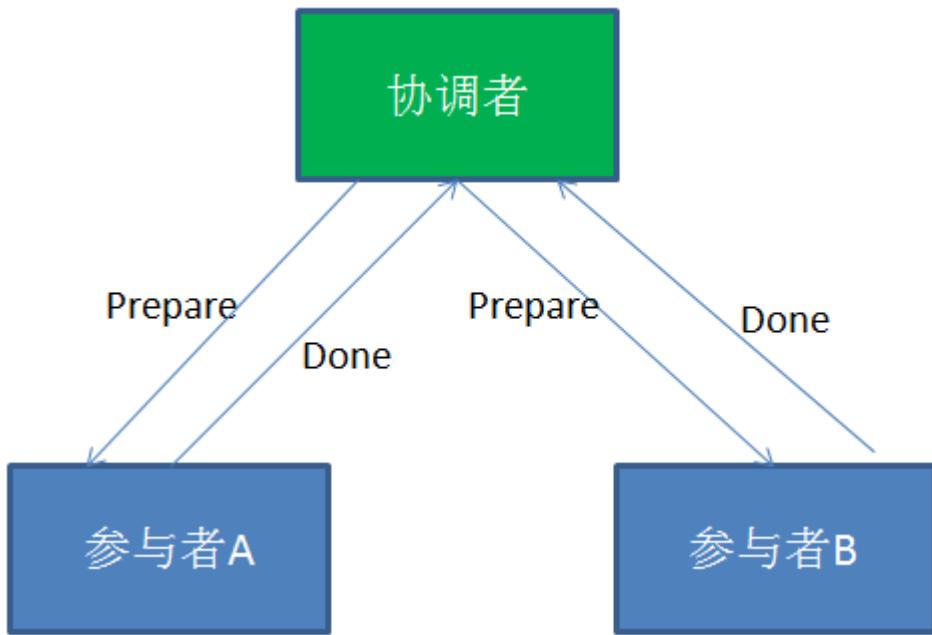




以上就是魔兽世界当中组团打BOSS的确认流程。这个流程和XA分布式事务协议的两阶段提交非常相似。

那么XA协议究竟是什么样子呢？在XA协议中包含着两个角色：**事务协调者**和**事务参与者**。让我们来看一看他们之间的交互流程：

第一阶段：

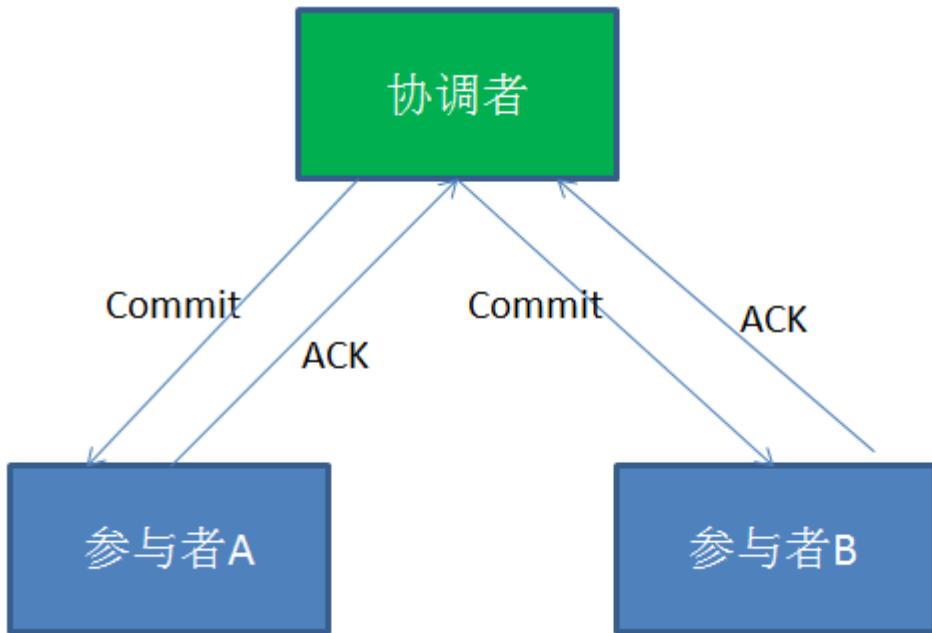


在XA分布式事务的第一阶段，作为事务协调者的节点会首先向所有的参与者节点发送Prepare请求。

在接到Prepare请求之后，每一个参与者节点会各自执行与事务有关的数据更新，写入Undo Log和Redo Log。如果参与者执行成功，暂时不提交事务，而是向事务协调节点返回“完成”消息。

当事务协调者接到了所有参与者的返回消息，整个分布式事务将会进入第二阶段。

第二阶段：



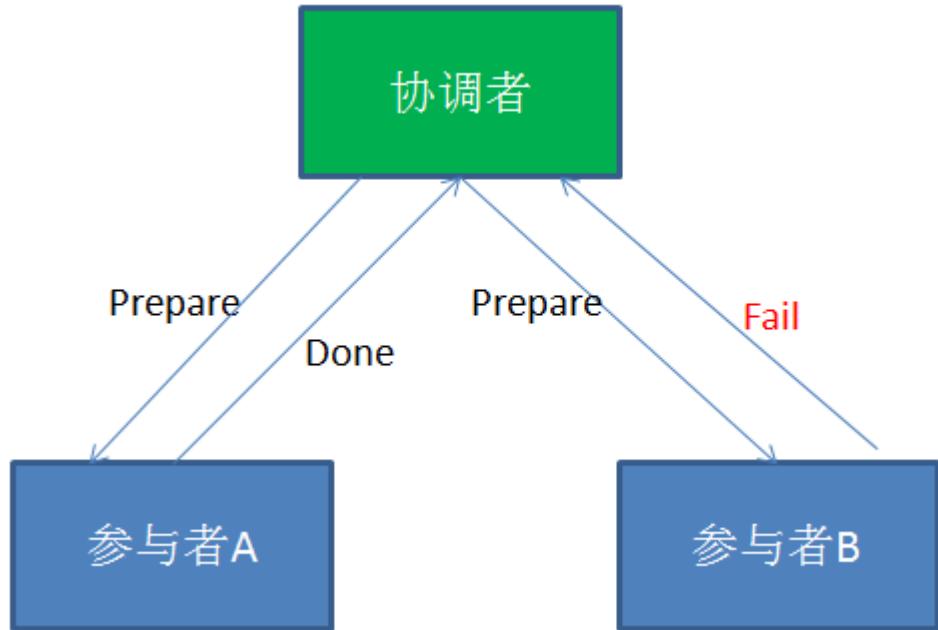
在XA分布式事务的第二阶段，如果事务协调节点在之前所收到都是正向返回，那么它将会向所有事务参与者发出Commit请求。

接到Commit请求之后，事务参与者节点会各自进行本地的事务提交，并释放锁资源。当本地事务完成提交后，将会向事务协调者返回“完成”消息。

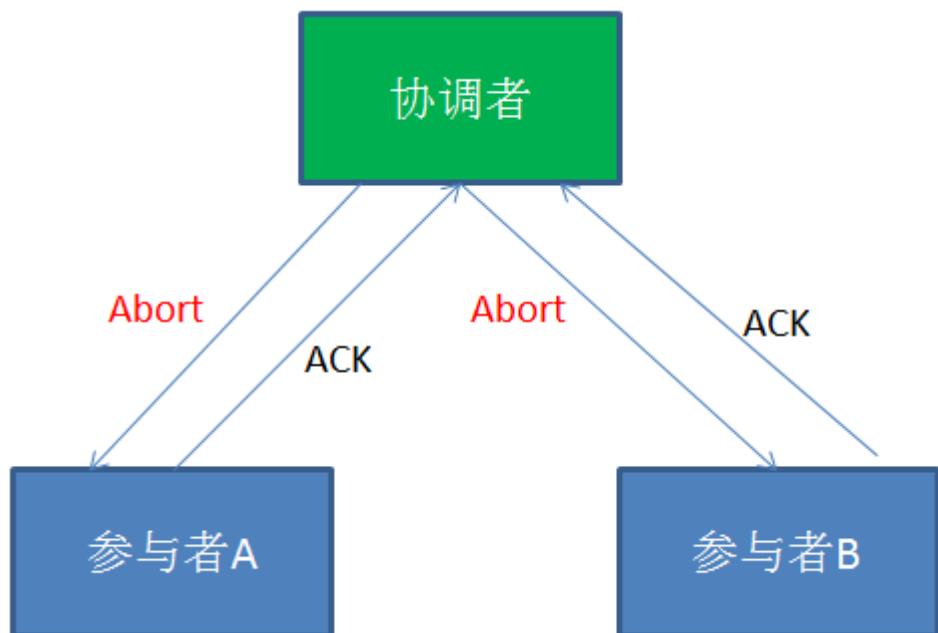
当事务协调者接收到所有事务参与者的“完成”反馈，整个分布式事务完成。

以上所描述的是XA两阶段提交的正向流程，接下来我们看一看失败情况的处理流程：

第一阶段：



第二阶段：



在XA的第一阶段，如果某个事务参与者反馈失败消息，说明该节点的本地事务执行不成功，必须回滚。

于是在第二阶段，事务协调节点向所有的事务参与者发送Abort请求。接收到Abort请求之后，各个事务参与者节点需要在本地进行事务的回滚操作，回滚操作依照Undo Log来进行。

以上就是XA两阶段提交协议的详细过程。

XA两阶段提交的不足

XA两阶段提交究竟有哪些不足呢？

1. 性能问题

XA协议遵循强一致性。在事务执行过程中，各个节点占用着数据库资源，只有当所有节点准备完毕，事务协调者才会通知提交，参与者提交后释放资源。这样的过程有着非常明显的性能问题。

2. 协调者单点故障问题

事务协调者是整个XA模型的核心，一旦事务协调者节点挂掉，参与者收不到提交或是回滚通知，参与者会一直处于中间状态无法完成事务。

3. 丢失消息导致的不一致问题。

在XA协议的第二个阶段，如果发生局部网络问题，一部分事务参与者收到了提交消息，另一部分事务参与者没收到提交消息，那么就导致了节点之间数据的不一致。

如果避免XA两阶段提交的种种问题呢？有许多其他的分布式事务方案可供选择：

XA三阶段提交 (3PC)

XA三阶段提交在两阶段提交的基础上增加了CanCommit阶段，并且引入了超时机制。一旦事物参与者迟迟没有接到协调者的commit请求，会自动进行本地commit。这样有效解决了协调者单点故障的问题。但是性能问题和不一致的问题仍然没有根本解决。

MQ事务

利用消息中间件来异步完成事务的后一半更新，实现系统的最终一致性。这个方式避免了像XA协议那样的性能问题。

TCC事务

TCC事务是Try、Commit、Cancel三种指令的缩写，其逻辑模式类似于XA两阶段提交，但是实现方式是在代码层面来人为实现。



十、BitMap

Bit-map的基本思想

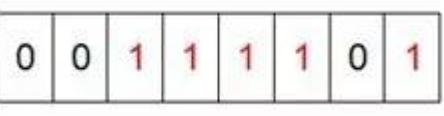
32位机器上，对于一个整型数，比如int a=1 在内存中占32bit位，这是为了方便计算机的运算。但是对于某些应用场景而言，这属于一种巨大的浪费，因为我们可以用对应的32bit位对应存储十进制的0-31个数，而这就是Bit-map的基本思想。Bit-map算法利用这种思想处理大量数据的排序、查询以及去重。 Bitmap在用户群做交集和并集运算的时候也有极大的便利。

Bit-map应用之快速排序

假设我们要对0-7内的5个元素(4,7,2,5,3)排序（这里假设这些元素没有重复），我们就可以采用Bit-map的方法来达到排序的目的。要表示8个数，我们就只需要8个Bit（1Bytes），首先我们开辟1Byte的空间，将这些空间的所有Bit位都置为0，



对应位设置为1：



遍历一遍Bit区域，将该位是一的位的编号输出 (2, 3, 4, 5,

7)，这样就达到了排序的目的，时间复杂度O(n)。 优点： 运算效率高，不需要进行比较和移位；

占用内存少，比如N=10000000；只需占用内存为N/8=1250000Byte=1.25M。 缺点： 所有的数据不能重复。即不可对重复的数据进行排序和查找。

Bit-map应用之快速去重

2.5亿个整数中找出不重复的整数的个数，内存空间不足以容纳这2.5亿个整数。 首先，根据“内存空间不足以容纳这2.5亿个整数”我们可以快速的联想到Bit-map。下边关键的问题就是怎么设计我们的Bit-map来表示这2.5亿个数字的状态了。其实这个问题很简单，一个数字的状态只有三种，分别为不存在，只有一个，有重复。因此，我们只需要2bits就可以对一个数字的状态进行存储了，假设我们设定一个数字不存在为00，存在一次01，存在两次及其以上为11。那我们大概需要存储空间几十兆左右。 接下来的任务就是遍历一次这2.5亿个数字，如果对应的状态位为00，则将其变为01；如果对应的状态位为01，则将其变为11；如果为11，对应的转态位保持不变。 最后，我们将状态位为01的进行统计，就得到了不重复的数字个数，时间复杂度为O(n)。

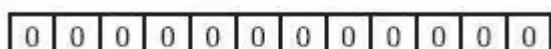
Bit-map应用之快速查询

同样，我们利用Bit-map也可以进行快速查询，这种情况下对于一个数字只需要一个bit位就可以了，0表示不存在，1表示存在。假设上述的题目改为，如何快速判断一个数字是否存在于上述的2.5亿个数字集合中。 同之前一样，首先我们先对所有的数字进行一次遍历，然后将相应的转态位改为1。遍历完以后就是查询，由于我们的Bit-map采取的是连续存储（整形数组形式，一个数组元素对应32bits），我们实际上是采用了一种分桶的思想。一个数组元素可以存储32个状态位，那将待查询的数字除以32，定位到对应的数组元素（桶），然后再求余（%32），就可以定位到相应的位置。如果为1，则代表改数字存在；否则，该数字不存在。

Bit-map扩展——Bloom Filter(布隆过滤器)

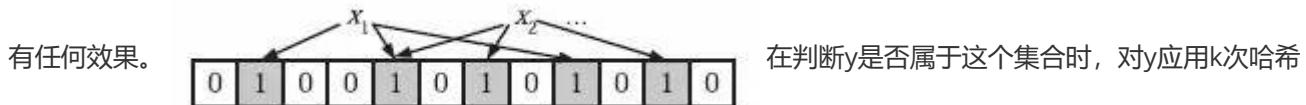
当一个元素被加入集合中时，通过k各散列函数将这个元素映射成一个位数组中的k个点，并将这k个点全部置为1。

有一定的误判率--在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误判为属于这个集合。因此，它不适合那些“零误判”的应用场合。在能容忍低误判的应用场景下，布隆过滤器通过极少的误判换取了存储空间的极大节省。



Bloom Filter使用k个相互独立的哈希函数

(Hash Function)，它们分别将集合中的每个元素映射到{1,...,m}的范围中。对任意一个元素x，第i个哈希函数映射的位置hi(x)就会被置为1 (1≤i≤k)。注：如果一个位置多次被置为1，那么只有第一次会起作用，后面几次将没



在判断y是否属于这个集合时，对y应用k次哈希函数，若所有 $h_i(y)$ 的位置都是1 ($1 \leq i \leq k$)，就认为y是集合中的元素，否则就认为y不是集合中的元素。



总结

使用Bit-map的思想，我们可以将存储空间进行压缩，而且可以对数字进行快速排序、去重和查询的操作。Bloom Filter是Bit-map思想的一种扩展，它可以在允许低错误率的场景下，大大地进行空间压缩，是一种拿错误率换取空间的数据结构。

应用

适用范围：可进行数据的快速查找，判重，删除，一般来说数据范围是int的10倍以下
基本原理及要点：
使用bit数组来表示某些元素是否存在，比如8位电话号码
扩展：bloom filter可以看做是对bit-map的扩展

问题实例：1、已知某个文件内包含一些电话号码，每个号码为8位数字，统计不同号码的个数。

8位最多99 999 999，大概需要99m个bit，大概10几M字节的内存即可。

2、在2.5亿个整数中找出不重复的整数，内存不足以容纳这2.5亿个整数。

方案1：采用2-Bitmap（每个数分配2bit，00表示不存在，01表示出现一次，10表示多次，11无意义）进行，共需内存 $232 * 2\text{bit} = 1\text{GB}$ 内存，还可以接受。然后扫描这2.5亿个整数，查看Bitmap中相对应位，如果是00变01，01变10，10保持不变。扫描完事后，查看bitmap，把对应位是01的整数输出即可。

十一、Bloom Filter

1. 把第一个URL按照三种Hash算法，分别生成三个不同的Hash值。



2. 把第二个URL也按照三种Hash算法，分别生成三个不同的Hash值。



3.依次比较每一个Hash结果，只有当全部结果都相等时，才判定两个URL相同。

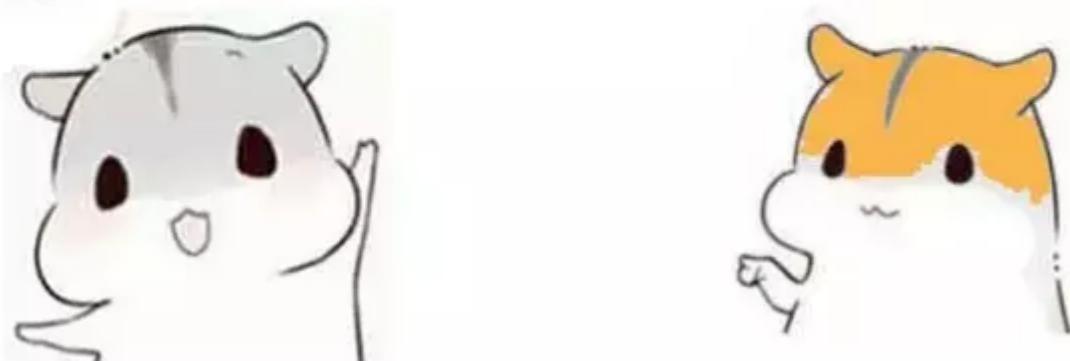
www.aaa.com

HashA	5	\neq	10
HashB	17	\neq	12
HashC	9	$= =$	9

↓

最终结果： URL不相同

假设标准HashCode 的重复几率是
0.01%，那么3个Hash结果同时重复的
几率就是 $0.01\% ^ 3 = 0.0000000001\%$ ！

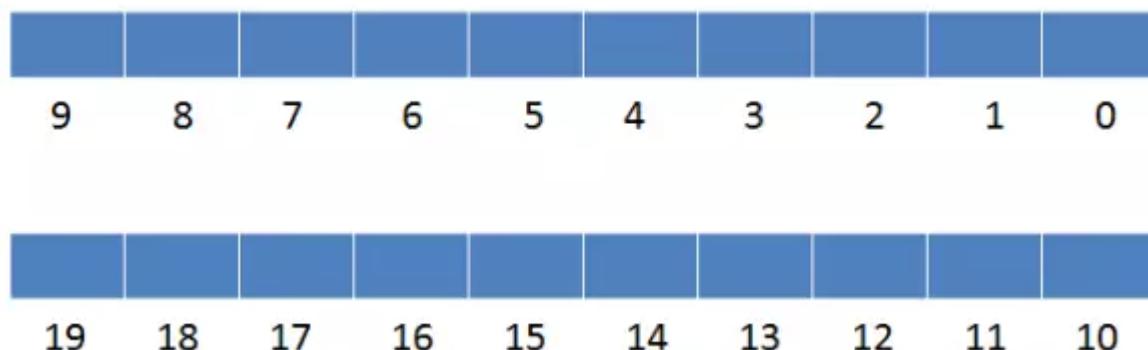


很好，你说的基本上就是布隆算法的核心思想。不过有一点不同，布隆算法会把每一个 Hash 结果都映射到同一个 Bitmap 上面。



具体怎样映射呢？流程如下：

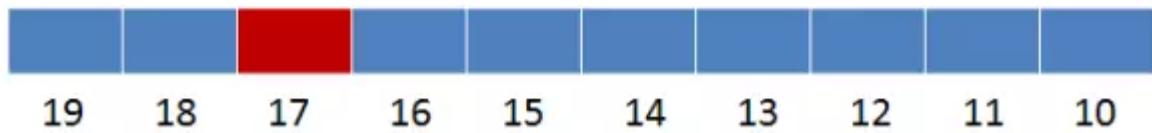
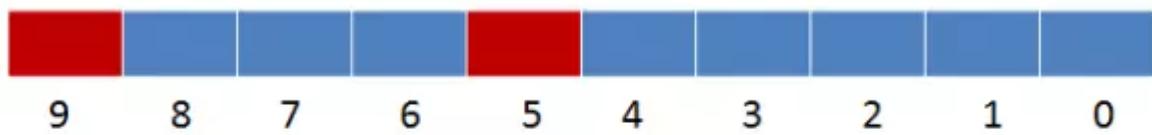
1. 创建一个空的Bitmap集合。



2. 把第一个URL按照三种Hash算法，分别生成三个不同的Hash值。



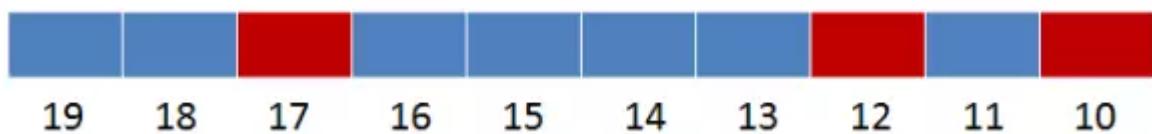
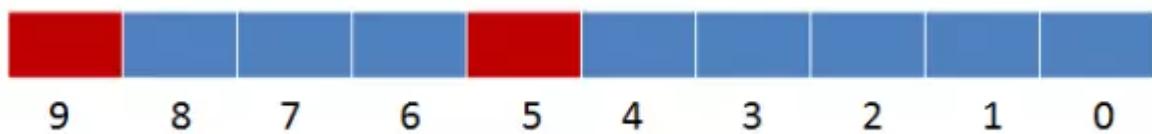
3. 分别判断5, 17, 9在Bitmap的对应位置是否为1，只要不同时为1，就认为该Url没有重复，于是把5, 17, 9的对应位置设置为1。



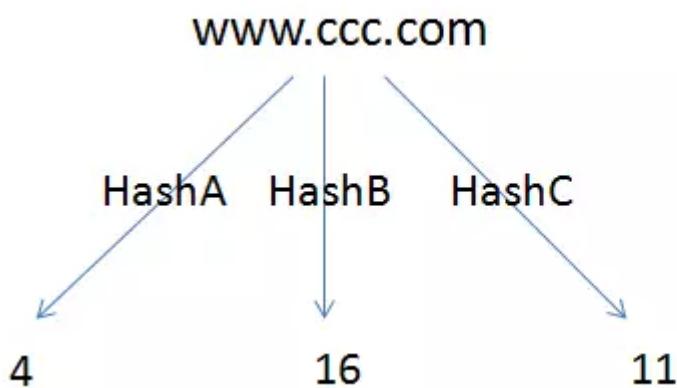
4. 把第二个URL按照三种Hash算法，分别生成三个不同的Hash值。



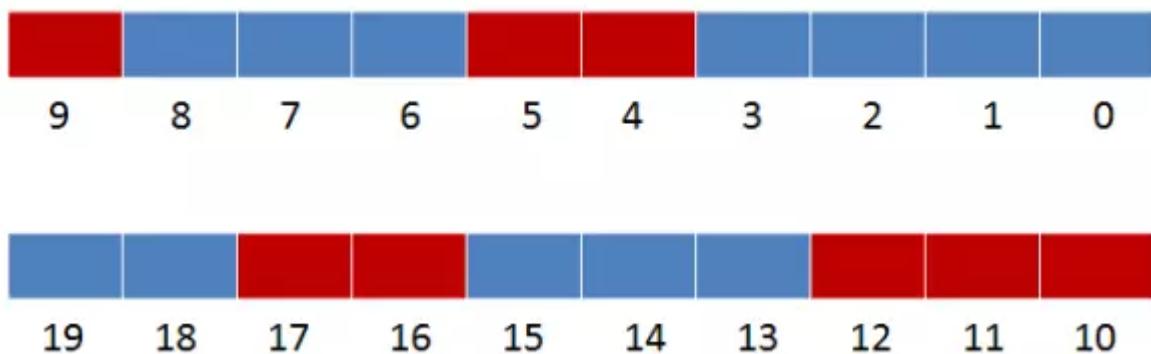
5. 分别判断10, 12, 9 在Bitmap的对应位置是否为1, 只要不同时为1, 就认为该Url没有重复, 于是把10, 12, 9 的对应位置设置为1。



6. 把第三个URL按照三种Hash算法，分别生成三个不同的Hash值。



7. 分别判断4, 16, 11在Bitmap的对应位置是否为1, 只要不同时为1, 就认为该Url没有重复, 于是把4, 16, 11的对应位置设置为1。



8. 把第四个URL按照三种Hash算法, 分别生成三个不同的Hash值。



9. 分别判断5, 17, 9在Bitmap的对应位置是否为1。判断的结果是5, 17, 9在Bitmap对应位置的值都是1, 所以判定该Url是一个重复的Url。

像上面这样, 就是布隆算法对于海量数据的过滤流程。



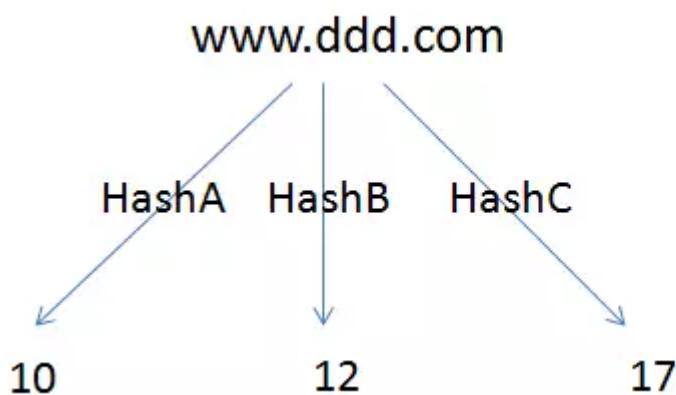
等等，我有个疑问，如果某一个新的 URL 经过三次 Hash 的结果是 10, 12, 17，那是否也会当成重复呢？



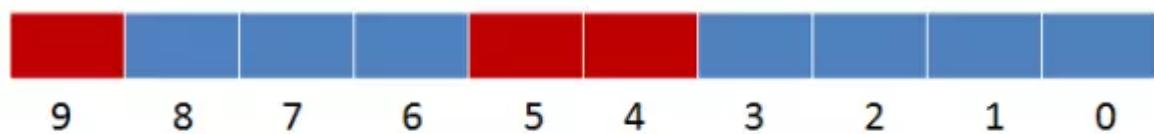
问得很好，假如这时候有一个新的 Url [www. ddd. com] 要插入进来，结果如下：



1. URL按照三个Hash算法得到三个结果。



2. 分别判断 10, 12, 17 在 Bitmap 的对应位置是否为 1。判断的结果是 10, 12, 17 在 Bitmap 对应位置的值都是 1，所以判定该Url是一个重复的Url。



这种情况叫做「误判」。由此可见，布隆算法虽然极力降低了 Hash 冲突的几率，但是仍然有一定的误判率。



为了减小误判的几率，可以让 Bitmap 的空间更大一些，单个 URL 所做的 Hash 更多一些（一般是 8 次），总之是在空间和准确率上做出取舍。



我还有一个问题，既然使用同一个 Bitmap
会出现误判，为什么不让每一种 Hash 算
法的结果对应一个独立的 Bitmap 呢？



那样的话，占用的空间也会相应增
加几倍，反而不如用 HashSet 了。



那么，如果我希望完全杜绝误判的
情况，应该怎么做？



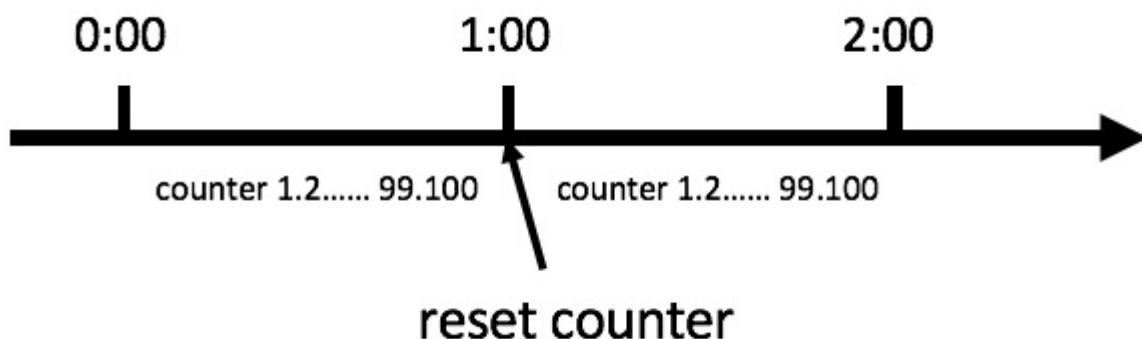
对于爬虫来说，可以容许极少量的 Url 误判为重复。如果是用于垃圾邮箱的过滤，可以考虑加上一个白名单，专门存储那些被误判的正常邮箱。



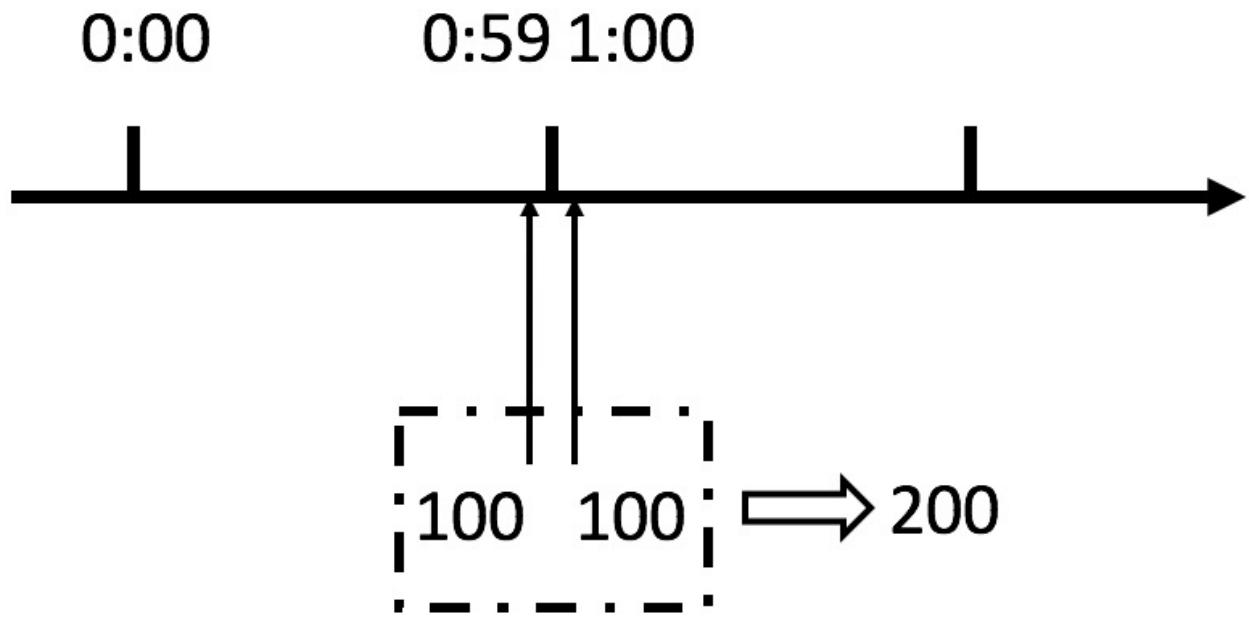
十二、常见的限流算法

计数器法

计数器法是限流算法里最简单也是最容易实现的一种算法。比如我们规定，对于A接口来说，我们1分钟的访问次数不能超过100个。那么我们可以这么做：在一开始的时候，我们可以设置一个计数器counter，每当一个请求过来的时候，counter就加1，如果counter的值大于100并且该请求与第一个请求的间隔时间还在1分钟之内，那么说明请求数过多；如果该请求与第一个请求的间隔时间大于1分钟，且counter的值还在限流范围内，那么就重置counter，具体算法的示意图如下：



这个算法虽然简单，但是有一个十分致命的问题，那就是临界问题，我们看下图：

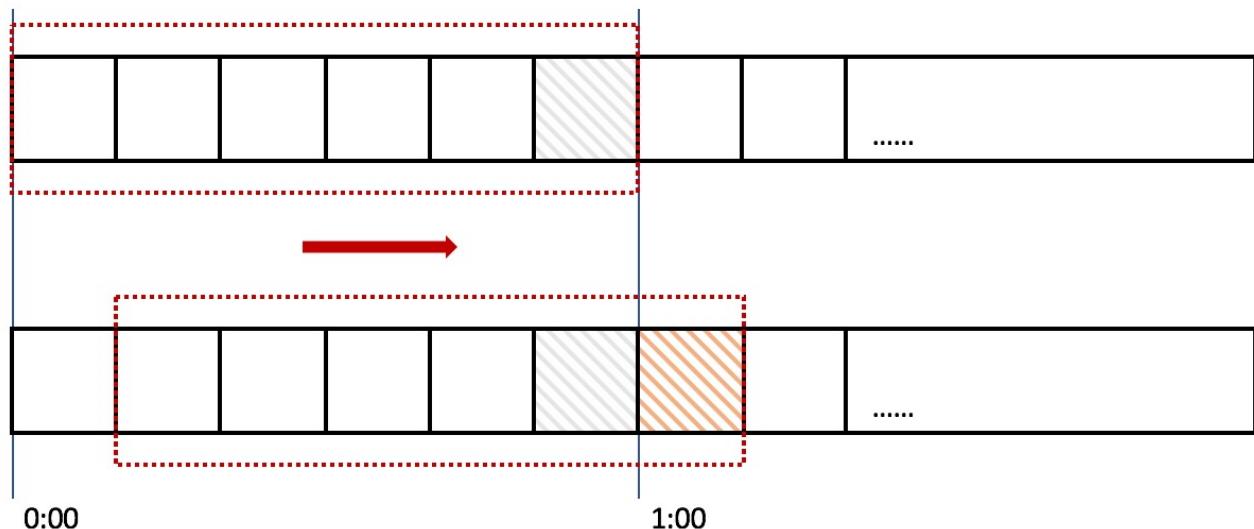


从上图中我们可以看到，假设有一个恶意用户，他在0:59时，瞬间发送了100个请求，并且1:00又瞬间发送了100个请求，那么其实这个用户在1秒里面，瞬间发送了200个请求。我们刚才规定的是1分钟最多100个请求，也就是每秒钟最多1.7个请求，用户通过在时间窗口的重置节点处突发请求，可以瞬间超过我们的速率限制。用户有可能通过算法的这个漏洞，瞬间压垮我们的应用。

聪明的朋友可能已经看出来了，刚才的问题其实是因为我们统计的精度太低。那么如何很好地处理这个问题呢？或者说，如何将临界问题的影响降低呢？我们可以看下面的滑动窗口算法。

滑动窗口

滑动窗口，又称rolling window。为了解决这个问题，我们引入了滑动窗口算法。如果学过TCP网络协议的话，那么一定对滑动窗口这个名词不会陌生。下面这张图，很好地解释了滑动窗口算法：



在上图中，整个红色的矩形框表示一个时间窗口，在我们的例子中，一个时间窗口就是一分钟。然后我们将时间窗口进行划分，比如图中，我们就将滑动窗口划成了6格，所以每格代表的是10秒钟。每过10秒钟，我们的时间窗口就会往右滑动一格。每一个格子都有自己独立的计数器counter，比如当一个请求在0:35秒的时候到达，那么0:30~0:39对应的counter就会加1。

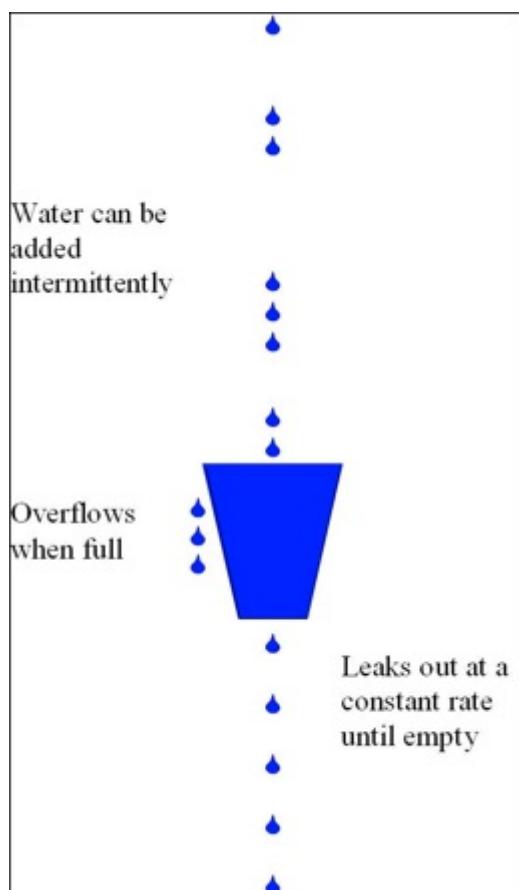
那么滑动窗口怎么解决刚才的临界问题的呢？我们可以看上图，0:59到达的100个请求会落在灰色的格子中，而1:00到达的请求会落在橘黄色的格子中。当时间到达1:00时，我们的窗口会往右移动一格，那么此时时间窗口内的总请求数量一共是200个，超过了限定的100个，所以此时能够检测出来触发了限流。

我再来看看刚才的计数器算法，我们可以发现，计数器算法其实就是滑动窗口算法。只是它没有对时间窗口做进一步地划分，所以只有1格。

由此可见，当滑动窗口的格子划分的越多，那么滑动窗口的滚动就越平滑，限流的统计就会越精确。

漏桶算法

漏桶算法，又称leaky bucket。为了理解漏桶算法，我们看一下维基百科上的对于该算法的示意图：

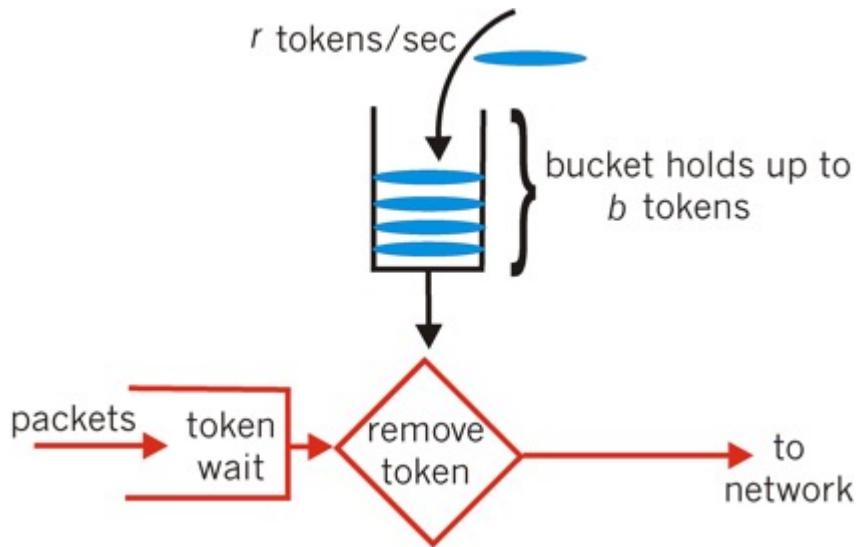


从图中我们可以看到，整个算法其实十分简单。首先，我们有一个固定容量的桶，有水流进来，也有水流出去。对于流进来的水来说，我们无法预计一共有多少水会流进来，也无法预计水流的速度。但是对于流出去的水来说，这个桶可以固定水流出的速率。而且，当桶满了之后，多余的水将会溢出。

我们将算法中的水换成实际应用中的请求，我们可以看到漏桶算法天生就限制了请求的速度。当使用了漏桶算法，我们可以保证接口会以一个常速速率来处理请求。所以漏桶算法天生不会出现临界问题。

令牌桶算法

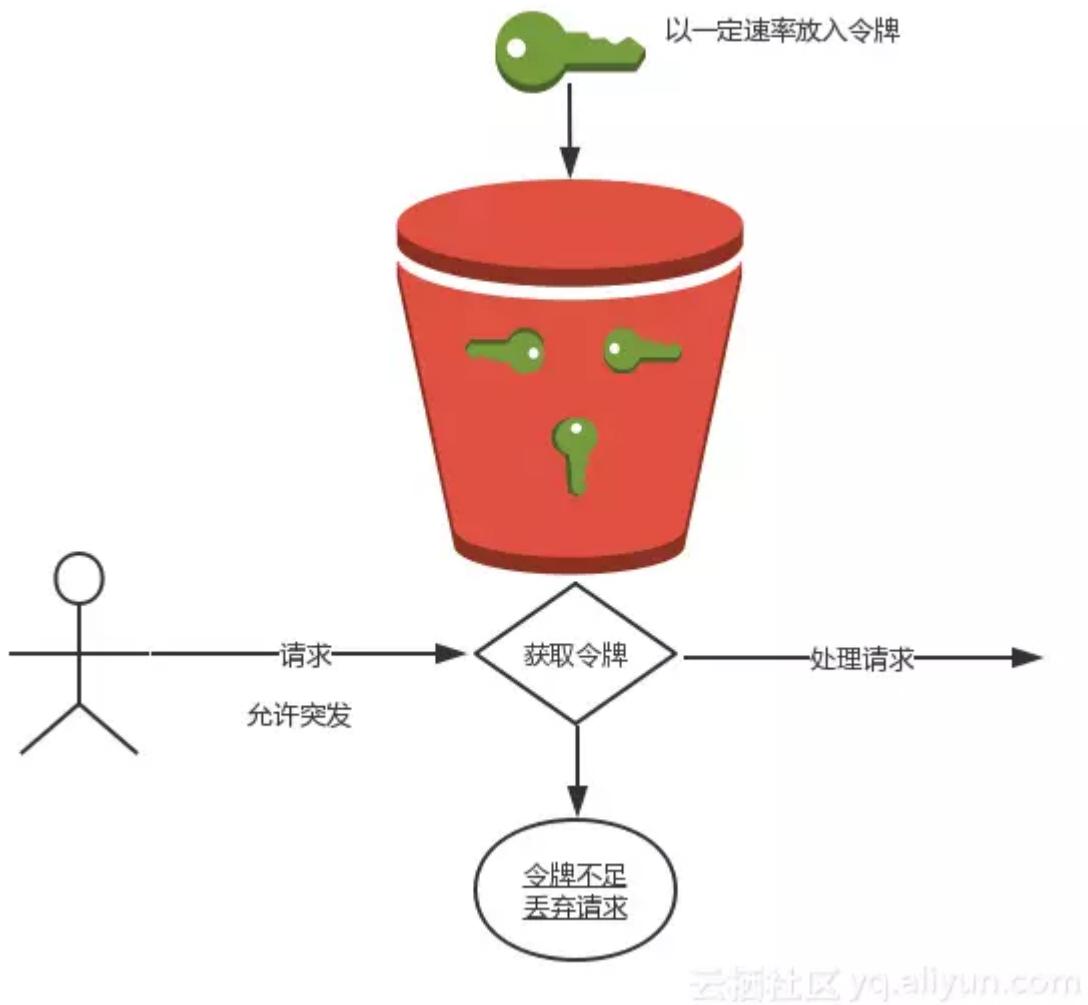
令牌桶算法，又称token bucket。为了理解该算法，我们再来看一下维基百科上对该算法的示意图：



从图中我们可以看到，令牌桶算法比漏桶算法稍显复杂。首先，我们有一个固定容量的桶，桶里存放着令牌（token）。桶一开始是空的，token以一个固定的速率 r 往桶里填充，直到达到桶的容量，多余的令牌将会被丢弃。每当一个请求过来时，就会尝试从桶里移除一个令牌，如果没有令牌的话，请求无法通过。

令牌桶

令牌桶算法是网络流量整形（Traffic Shaping）和速率限制（Rate Limiting）中最常使用的一种算法。典型情况下，令牌桶算法用来控制发送到网络上的数据的数目，并允许突发数据的发送(百科)。



云栖社区 yq.aliyun.com

在秒杀活动中，用户的请求速率是不固定的，这里我们假定为10r/s，令牌按照5个每秒的速率放入令牌桶，桶中最多存放20个令牌。仔细想想，是不是总有那么一部分请求被丢弃。

计数器 VS 滑动窗口

计数器算法是最简单的算法，可以看成是滑动窗口的低精度实现。滑动窗口由于需要存储多份的计数器（每一个格子存一份），所以滑动窗口在实现上需要更多的存储空间。也就是说，如果滑动窗口的精度越高，需要的存储空间就越大。

漏桶算法 VS 令牌桶算法

漏桶算法和令牌桶算法最明显的区别是令牌桶算法允许流量一定程度的突发。因为默认的令牌桶算法，取走token是不需要耗费时间的，也就是说，假设桶内有100个token时，那么可以瞬间允许100个请求通过。

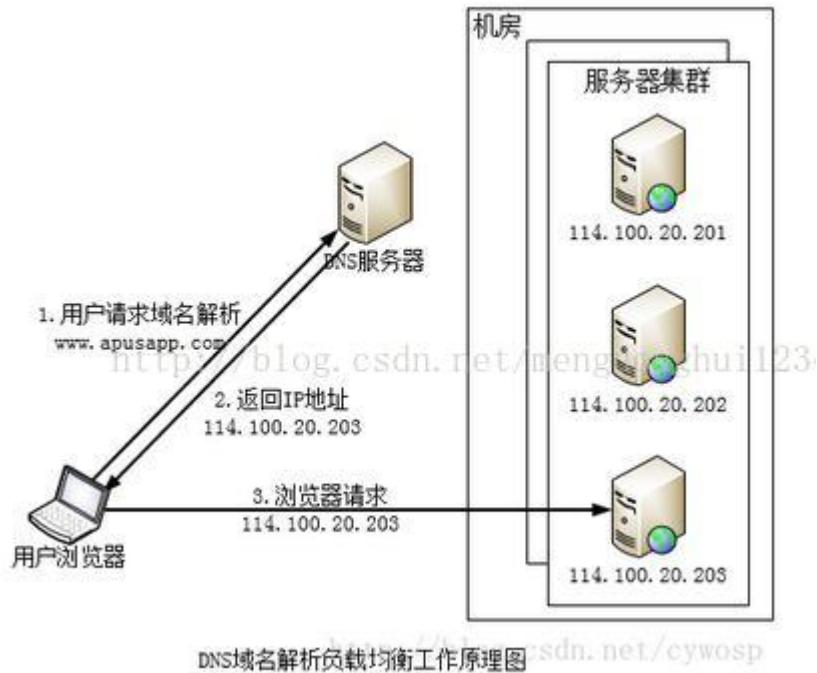
令牌桶算法由于实现简单，且允许某些流量的突发，对用户友好，所以被业界采用地较多。当然我们需要具体情况具体分析，只有最合适的算法，没有最优的算法。

十三、负载均衡

dns域名解析负载均衡

原理：在DNS服务器上配置多个域名对应IP的记录。例如一个域名www.baidu.com对应一组web服务器IP地址，域名解析时经过DNS服务器的算法将一个域名请求分配到合适的真实服务器上。

如图：



优点：将负载均衡的工作交给了DNS，省却了网站管理维护负载均衡服务器的麻烦，同时许多DNS还支持基于地理位置的域名解析，将域名解析成距离用户地理最近的一个服务器地址，加快访问速度吗，改善性能。

缺点：目前的DNS解析是多级解析，每一级DNS都可能缓存记录A，当某一服务器下线后，该服务器对应的DNS记录A可能仍然存在，导致分配到该服务器的用户访问失败。

DNS负载均衡的控制权在域名服务商手里，网站可能无法做出过多的改善和管理。

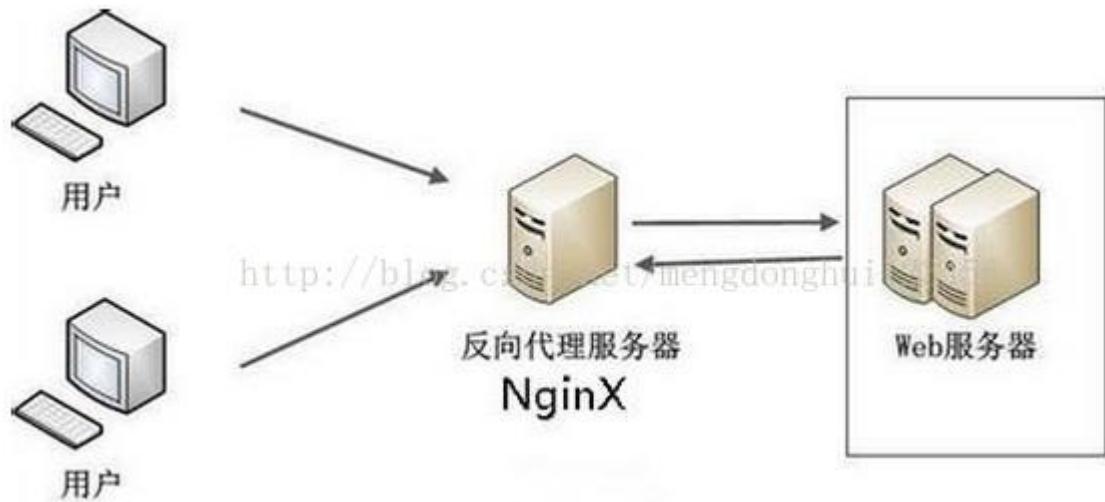
不能够按服务器的处理能力来分配负载。DNS负载均衡采用的是简单的轮询算法，不能区分服务器之间的差异，不能反映服务器当前运行状态，所以其的负载均衡效果并不是太好。

可能会造成额外的网络问题。为了使本DNS服务器和其他DNS服务器及时交互，保证DNS数据及时更新，使地址能随机分配，一般都要将DNS的刷新时间设置的较小，但太小将会使DNS流量大增造成额外的网络问题。

反向代理负载均衡

原理：反向代理处于web服务器这边，反向代理服务器提供负载均衡的功能，同时管理一组web服务器，它根据负载均衡算法将请求的浏览器访问转发到不同的web服务器处理，处理结果经过反向服务器返回给浏览器。

如图：



例如：浏览器访问请求的地址是反向代理服务器的地址114.100.80.10，反向代理服务器收到请求，经过负载均衡算法后得到一个真实物理地址10.0.03，并将请求结果发给真实无服务，真实服务器处理完后通过反向代理服务器返回给请求用户。

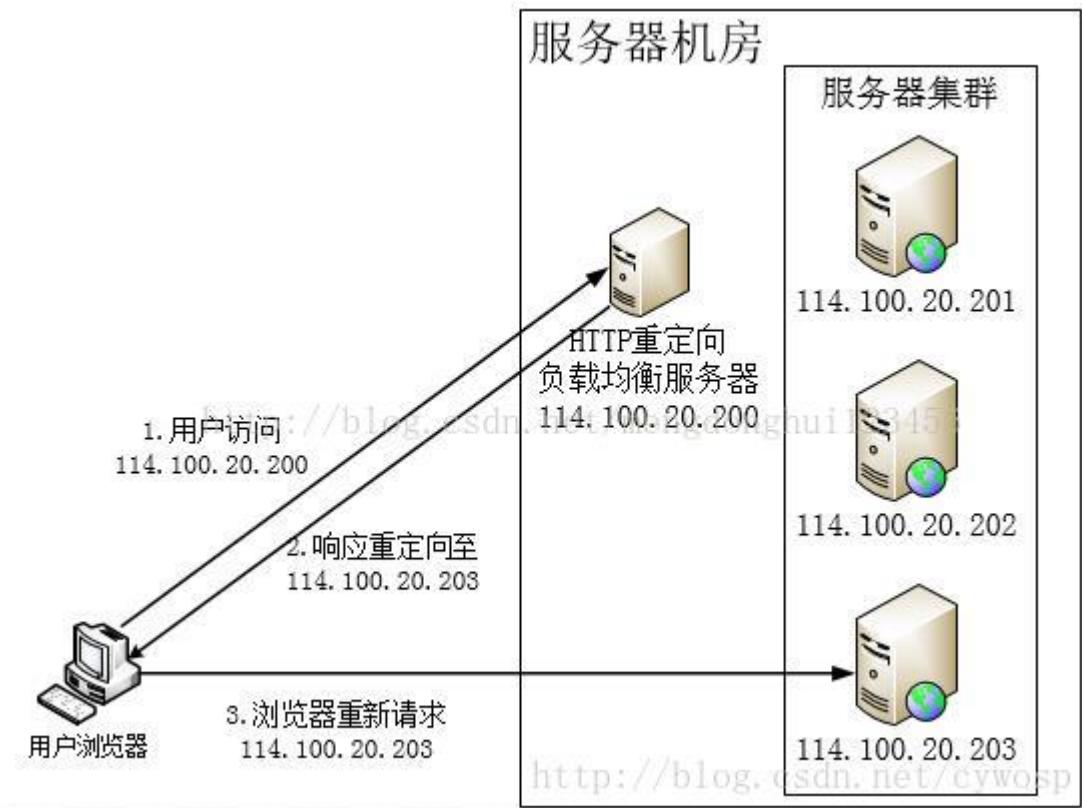
优点：部署简单，处于http协议层面。

缺点：使用了反向代理服务器后，web 服务器地址不能直接暴露在外，因此web服务器不需要使用外部IP地址，而反向代理服务作为沟通桥梁就需要配置双网卡、外部内部两套IP地址。

http重定向协议实现负载均衡

原理：根据用户的http请求计算出一个真实的web服务器地址，并将该web服务器地址写入http重定向响应中返回给浏览器，由浏览器重新进行访问。

如图：



优点：比较简单

缺点：浏览器需要两次次请求服务器才能完成一次访问，性能较差。

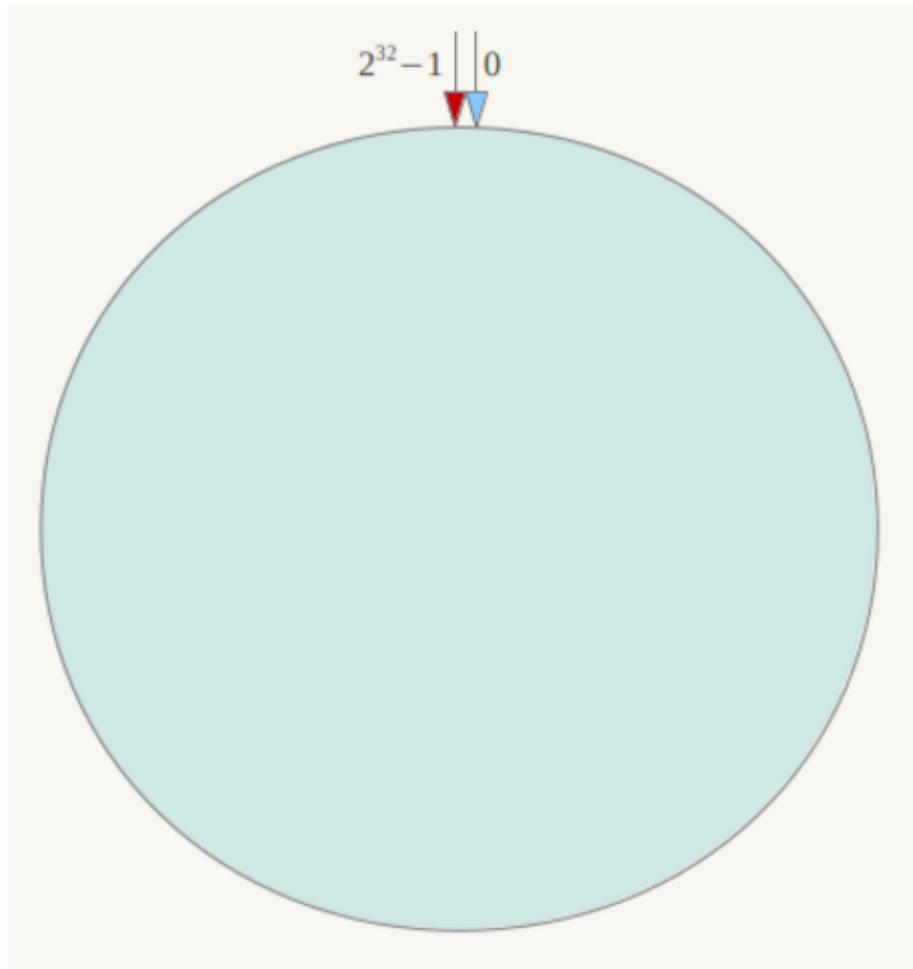
http重定向服务器自身的处理能力可能成为瓶颈。

使用http302响应重定向，有可能使搜索引擎判断为SEO作弊，降低搜索排名。

分层的负载均衡算法

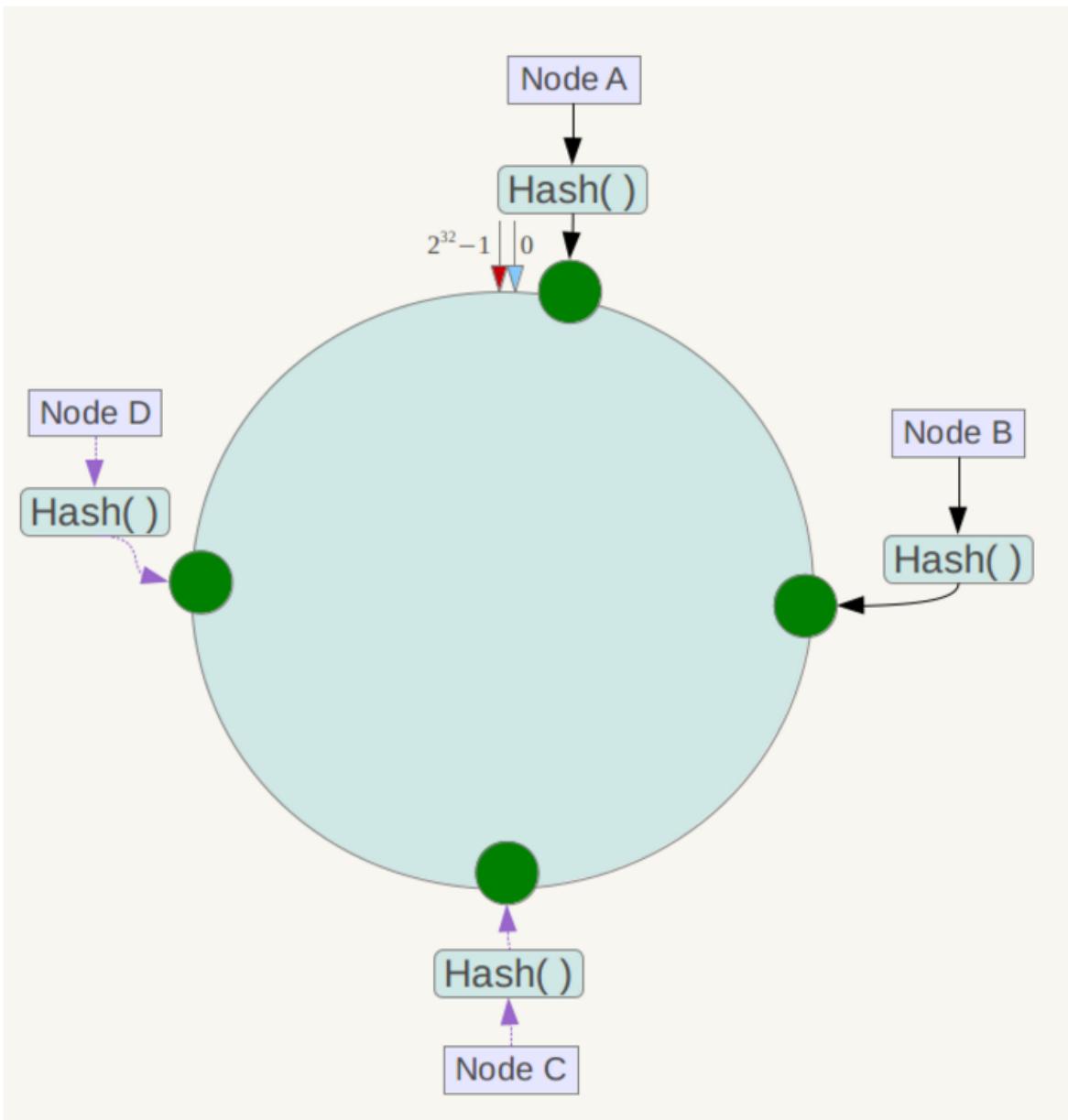
十四、一致性Hash算法

一致性哈希算法 (Consistent Hashing) 最早在论文《[Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web](#)》中被提出。简单来说，一致性哈希将整个哈希值空间组织成一个虚拟的圆环，如假设某哈希函数H的值空间为0-2^32-1 (即哈希值是一个32位无符号整形)，整个哈希空间环如下：



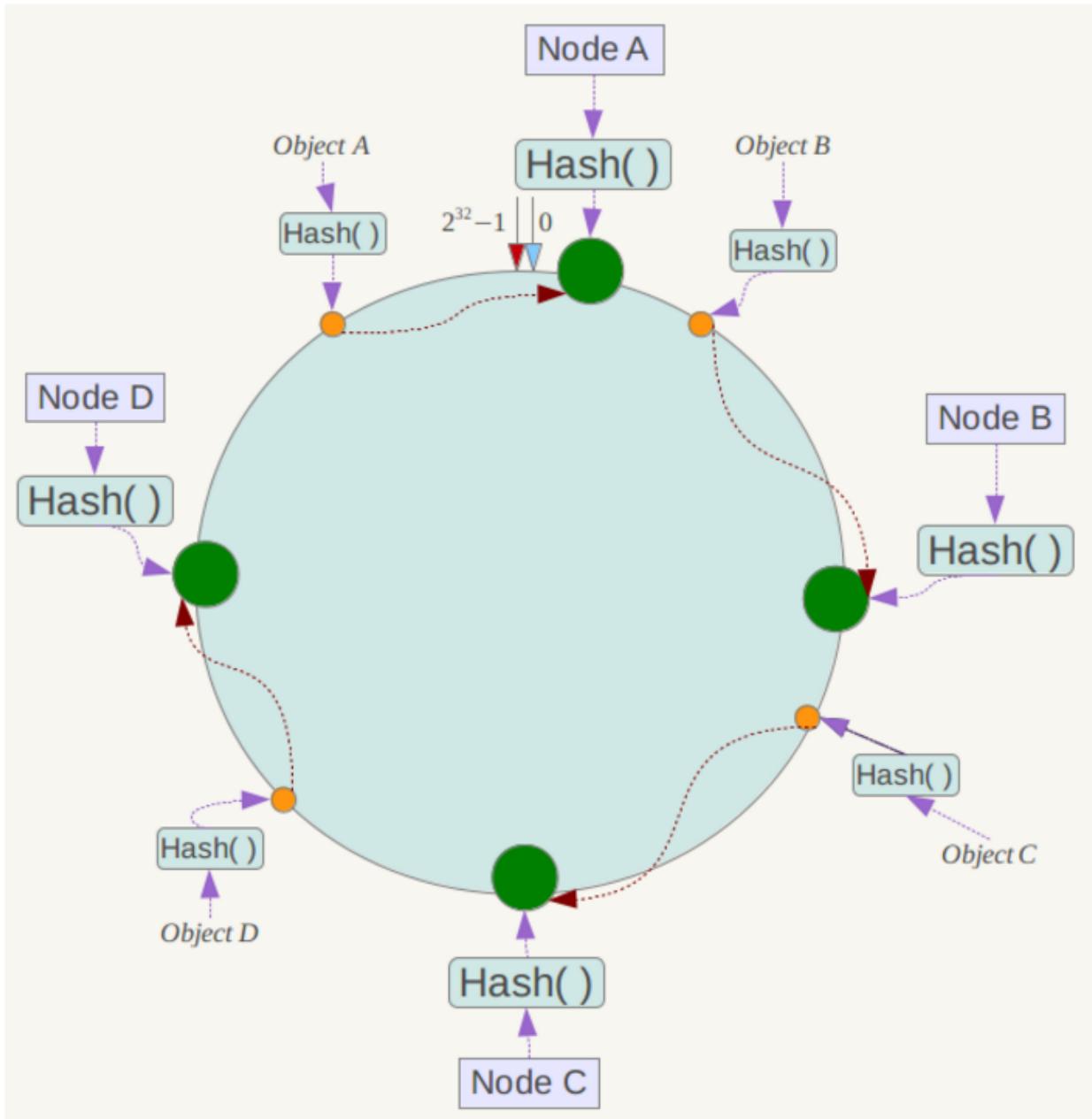
整个空间按顺时针方向组织。0和 $2^{32}-1$ 在零点中方向重合。

下一步将各个服务器使用Hash进行一个哈希，具体可以选择服务器的ip或主机名作为关键字进行哈希，这样每台机器就能确定其在哈希环上的位置，这里假设将上文中四台服务器使用ip地址哈希后在环空间的位置如下：



接下来使用如下算法定位数据访问到相应服务器：将数据key使用相同的函数Hash计算出哈希值，并确定此数据在环上的位置，从此位置沿环顺时针“行走”，第一台遇到的服务器就是其应该定位到的服务器。

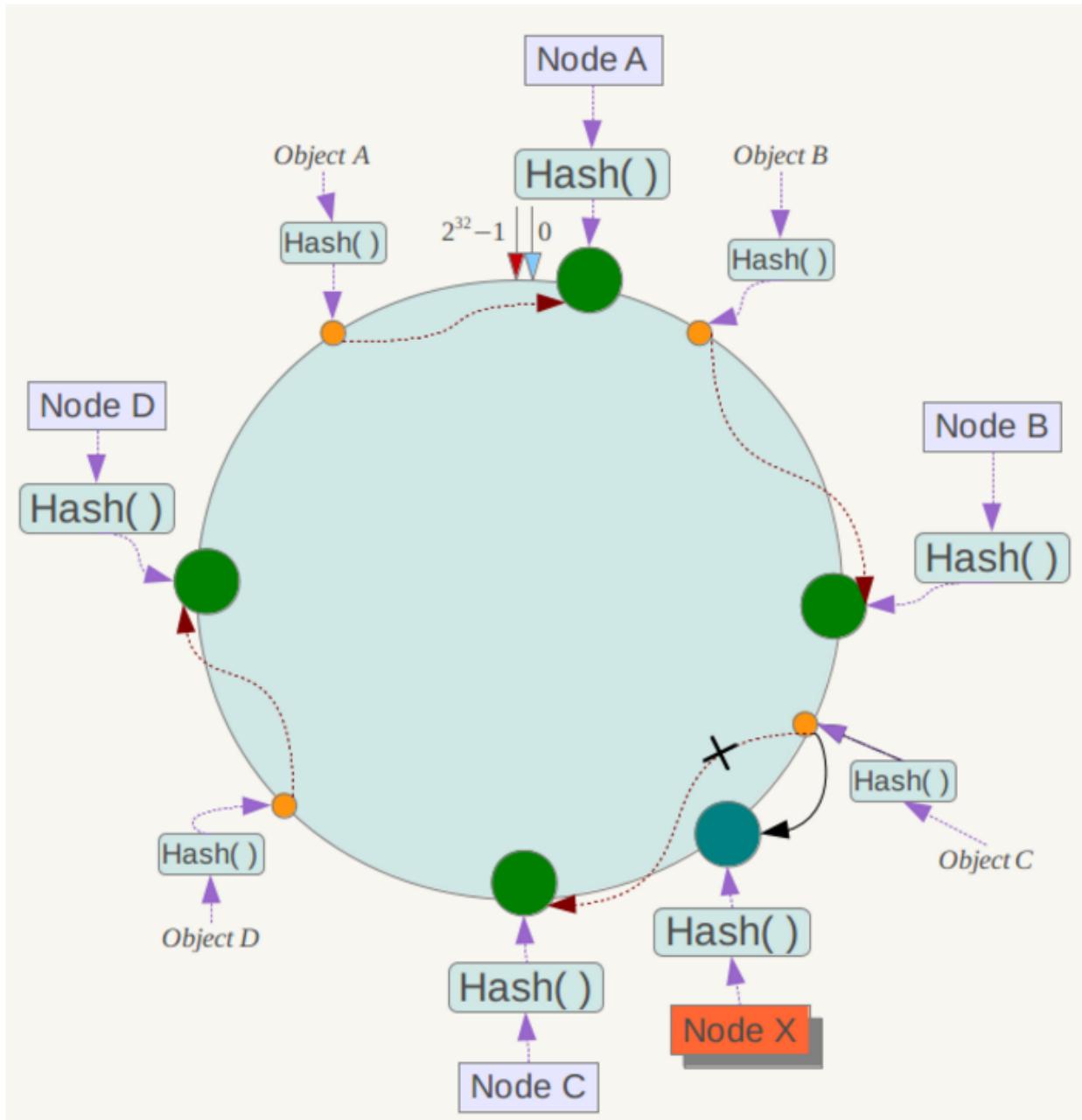
例如我们有Object A、Object B、Object C、Object D四个数据对象，经过哈希计算后，在环空间上的位置如下：



根据一致性哈希算法，数据A会被定为到Node A上，B被定为到Node B上，C被定为到Node C上，D被定为到Node D上。

下面分析一致性哈希算法的容错性和可扩展性。现假设Node C不幸宕机，可以看到此时对象A、B、D不会受到影响，只有C对象被重定位到Node D。一般的，在一致性哈希算法中，如果一台服务器不可用，则受影响的数据仅仅是此服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它不会受到影响。

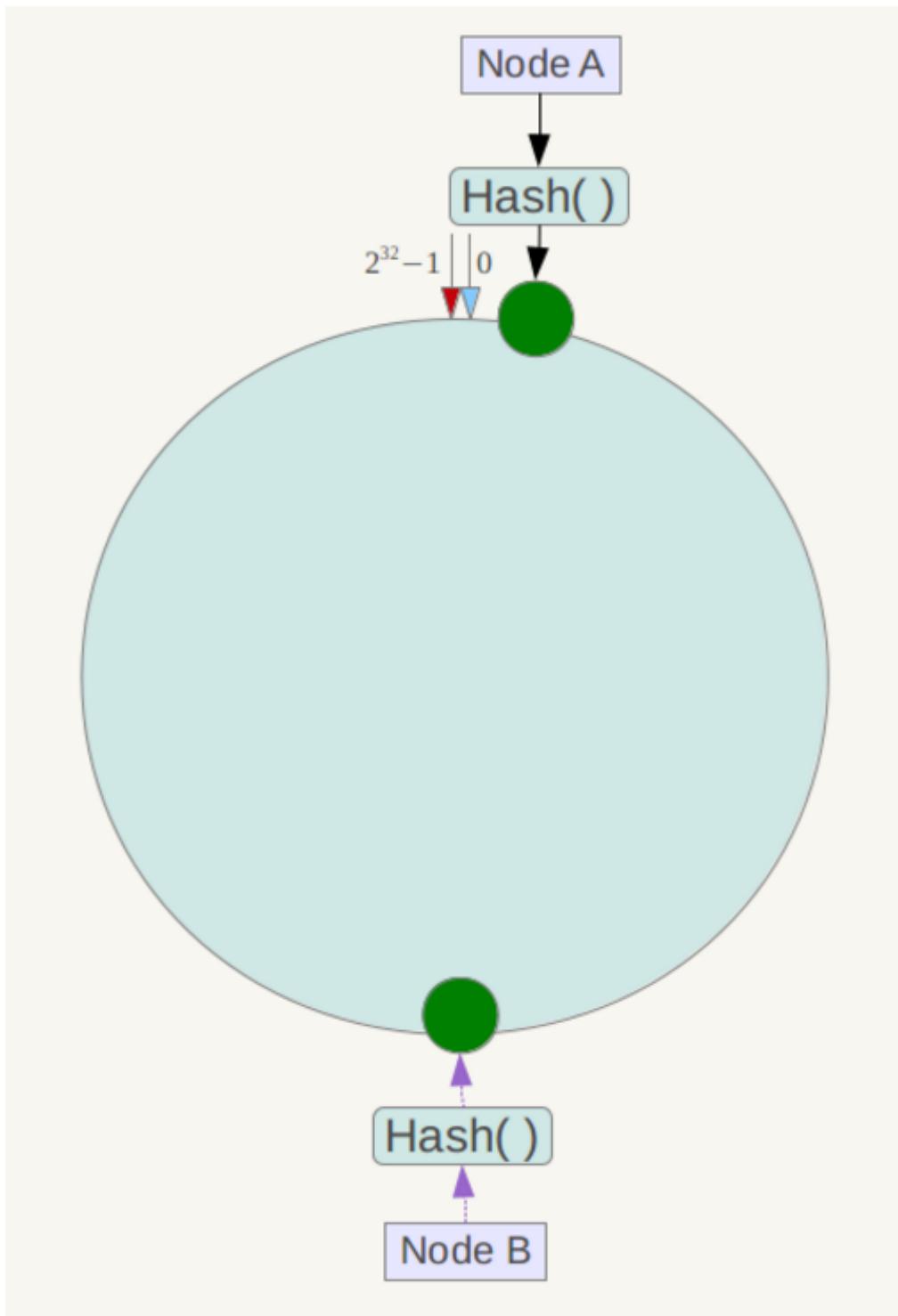
下面考虑另外一种情况，如果在系统中增加一台服务器Node X，如下图所示：



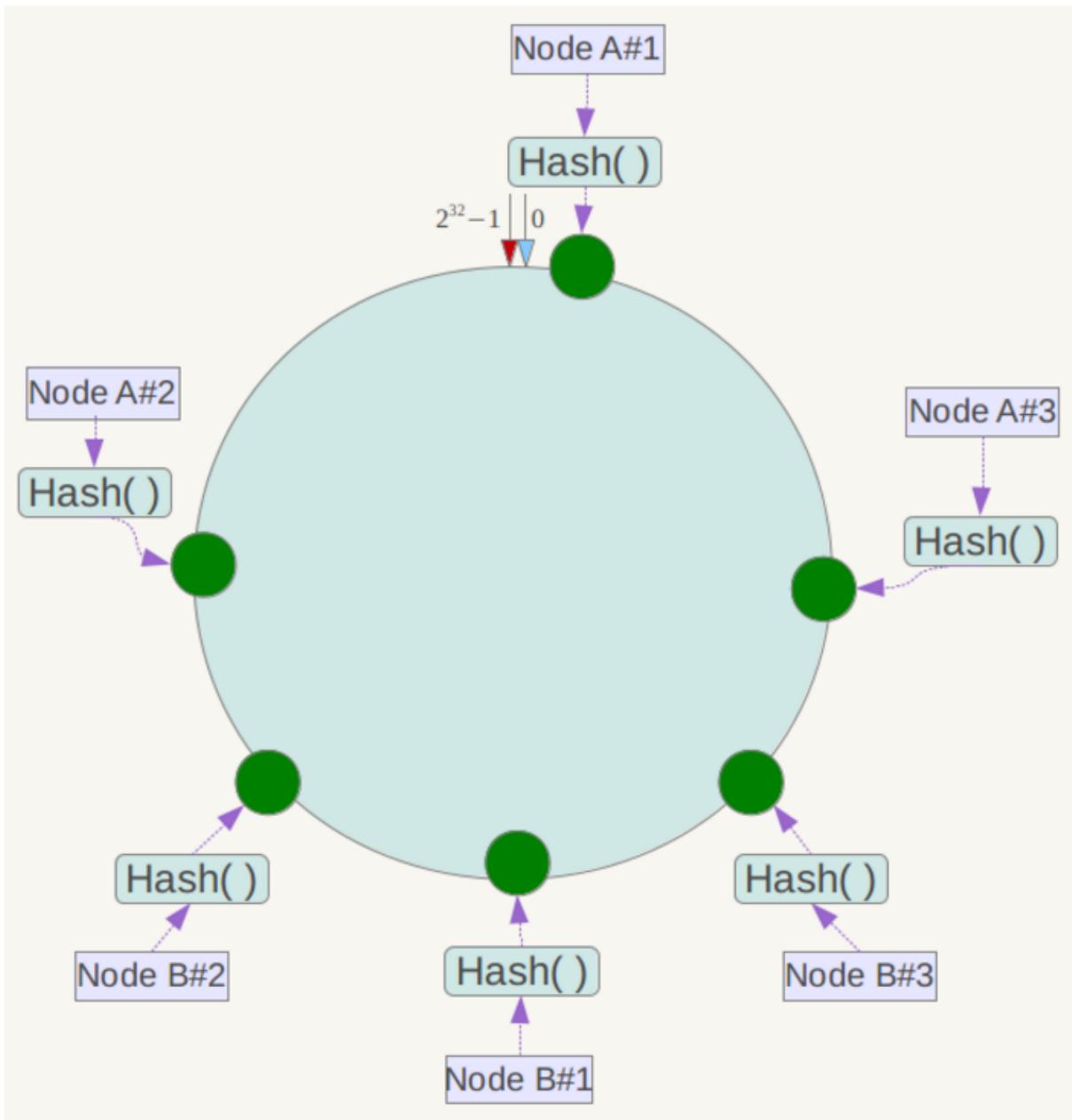
此时对象Object A、B、D不受影响，只有对象C需要重定位到新的Node X。一般的，在一致性哈希算法中，如果增加一台服务器，则受影响的数据仅仅是新服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它数据也不会受到影响。

综上所述，一致性哈希算法对于节点的增减都只需重定位环空间中的一小部分数据，具有较好的容错性和可扩展性。

另外，一致性哈希算法在服务节点太少时，容易因为节点分部不均匀而造成数据倾斜问题。例如系统中只有两台服务器，其环分布如下，



此时必然造成大量数据集中到Node A上，而只有极少量会定位到Node B上。为了解决这种数据倾斜问题，一致性哈希算法引入了虚拟节点机制，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个此服务节点，称为虚拟节点。具体做法可以在服务器ip或主机名的后面增加编号来实现。例如上面的情况，可以为每台服务器计算三个虚拟节点，于是可以分别计算“Node A#1”、“Node A#2”、“Node A#3”、“Node B#1”、“Node B#2”、“Node B#3”的哈希值，于是形成六个虚拟节点：



同时数据定位算法不变，只是多了一步虚拟节点到实际节点的映射，例如定位到“Node A#1”、“Node A#2”、“Node A#3”三个虚拟节点的数据均定位到Node A上。这样就解决了服务节点少时数据倾斜的问题。在实际应用中，通常将虚拟节点数设置为32甚至更大，因此即使很少的服务节点也能做到相对均匀的数据分布。

数据库

一、数据库范式

1NF(第一范式)

第一范式是指数据库表中的每一列都是不可分割的基本数据项，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。如果出现重复的属性，就可能需要定义一个新的实体，新的实体由重复的属性构成，新实体与原实体之间为一对多关系。第一范式的模式要求属性值不可再分裂成更小部分，即属性项不能是属性组合或是由一组属性构成。

简而言之，第一范式就是无重复的列。例如，由“职工号”“姓名”“电话号码”组成的表(一个人可能有一部办公电话和一部移动电话)，这时将其规范化为1NF可以将电话号码分为“办公电话”和“移动电话”两个属性，即职工(职工号，姓名，办公电话，移动电话)。

2NF(第二范式)

第二范式(2NF)是在第一范式(1NF)的基础上建立起来的，即满足第二范式(2NF)必须先满足第一范式(1NF)。第二范式(2NF)要求数据库表中的每个实例或行必须可以被唯一地区分。为实现区分通常需要为表加上一个列，以存储各个实例的唯一标识。

如果关系模型R为第一范式，并且R中的每一个非主属性完全函数依赖于R的某个候选键，则称R为第二范式模式(如果A是关系模式R的候选键的一个属性，则称A是R的主属性，否则称A是R的非主属性)。

例如，在选课关系表(学号，课程号，成绩，学分)，关键字为组合关键字(学号，课程号)，但由于非主属性学分仅依赖于课程号，对关键字(学号，课程号)只是部分依赖，而不是完全依赖，因此此种方式会导致数据冗余以及更新异常等问题，解决办法是将其分为两个关系模式：学生表(学号，课程号，分数)和课程表(课程号，学分)，新关系通过学生表中的外关键字课程号联系，在需要时进行连接。

3NF(第三范式)

如果关系模型R是第二范式，且每个非主属性都不传递依赖于R的候选键，则称R是第三范式的模式。

以学生表(学号，姓名，课程号，成绩)为例，其中学生姓名无重名，所以该表有两个候选码(学号，课程号)和(姓名，课程号)，故存在函数依赖：学号——>姓名，(学号，课程号)——>成绩，唯一的非主属性成绩对码不存在部分依赖，也不存在传递依赖，所以属性属于第三范式。

二、数据库开发规范

基础规范

- (1) 必须使用InnoDB存储引擎解读：支持事务、行级锁、并发性能更好、CPU及内存缓存页优化使得资源利用率更高
- (2) 必须使用UTF8字符集解读：万国码，无需转码，无乱码风险，节省空间
- (3) **数据表、数据字段必须加入中文注释解读：N年后谁知道这个r1,r2,r3字段是干嘛的**
- (4) 禁止使用存储过程、视图、触发器、Event解读：高并发大数据的互联网业务，架构设计思路是“**解放数据库CPU，将计算转移到服务层**”，并发量大的情况下，这些功能很可能将数据库拖死，业务逻辑放到服务层具备更好的扩展性，能够轻易实现“增机器就加性能”。**数据库擅长存储与索引，CPU计算还是上移吧**
- (5) 禁止存储大文件或者大照片解读：为何要让数据库做它不擅长的事情？大文件和照片存储在文件系统，数据库里存URI多好

命名规范

- (6) 只允许使用内网域名，而不是ip连接数据库
 - (7) 线上环境、开发环境、测试环境数据库内网域名遵循命名规范业务名称：xxx线上环境：
my10000m.mysql.jddb.com
开发环境：yf10000m.mysql.jddb.com
测试环境：test10000m.mysql.jddb.com
从库在名称后加-s标识，备库在名称后加-ss标识线上从库：my10000sa.mysql.jddb.com
-
- (8) 库名、表名、字段名：小写，**下划线风格**，不超过32个字符，必须见名知意，禁止拼音英文混用
 - (9) 表名t_xxx，非唯一索引名idx_xxx，唯一索引名uniq_xxx
 - (10) 单实例表数目必须小于500
 - (11) 单表列数目必须小于30
 - (12) 表必须有主键，例如自增主键解读：
 - a) 主键递增，数据行写入可以提高插入性能，可以避免page分裂，减少表碎片提升空间和内存的使用
 - b) **主键要选择较短的数据类型**，Innodb引擎普通索引都会保存主键的值，较短的数据类型可以有效的减少索引的磁盘空间，提高索引的缓存效率
 - c) 无主键的表删除，在row模式的主从架构，会导致备库夯住
 - (13) 禁止使用外键，如果有外键完整性约束，需要应用程序控制解读：外键会导致表与表之间耦合，update与delete操作都会涉及相关联的表，十分影响sql的性能，甚至会造成死锁。高并发情况下容易造成数据库性能，大数据高并发业务场景数据库使用以性能优先

字段设计规范

- (14) 必须把字段定义为**NOT NULL**并且提供默认值解读：
 - a) null的列使索引/索引统计/值比较都更加复杂，对MySQL来说**更难优化**
 - b) null这种类型MySQL内部需要进行特殊处理，**增加数据库处理记录的复杂性**；同等条件下，表中有较多空字段的时候，数据库的处理性能会降低很多
 - c) null值需要更多的存储空间，无论是表还是索引中每行中的null的列都需要额外的空间来标识
 - d) 对null的处理时候，只能采用is null或is not null，而不能采用=、in、<、<>、!=、not in这些操作符号。如：where name!='shenjian'，如果存在name为null值的记录，查询结果就不会包含name为null值的记录
- (15) 禁止使用TEXT、BLOB类型解读：会浪费更多的磁盘和内存空间，非必要的大量的大字段查询会淘汰掉热数据，导致内存命中率急剧降低，影响数据库性能
- (16) 禁止使用小数存储货币解读，小数容易导致钱对不上
- (17) 必须使用varchar(20)存储手机号解读：
 - a) 涉及到区号或者国家代号，可能出现+()-
 - b) 手机号会去做数学运算么？
 - c) varchar可以支持模糊查询，例如：like“138%”

- (18) 禁止使用ENUM，可使用TINYINT代替解读：a) 增加新的ENUM值要做DDL操作b) ENUM的内部实际存储就是整数，你以为自己定义的是字符串？
- (19) 单表索引建议控制在5个以内
- (20) 单索引字段数不允许超过5个解读：字段超过5个时，实际已经起不到有效过滤数据的作用了
- (21) 禁止在更新十分频繁、区分度不高的属性上建立索引解读：a) 更新会变更B+树，更新频繁的字段建立索引会大大降低数据库性能b) “性别”这种区分度不大的属性，建立索引是没有什么意义的，不能有效过滤数据，性能与全表扫描类似
- (22) 建立组合索引，必须把区分度高的字段放在前面解读：能够更加有效的过滤数据
- (23) **禁止使用SELECT *，只获取必要的字段，需要显示说明列属性解读：**
- ** a) 读取不需要的列会增加CPU、IO、NET消耗**
- ** b) 不能有效的利用覆盖索引**
- ** c) 使用SELECT *容易在增加或者删除字段后出现程序BUG**
- (24) 禁止使用INSERT INTO t_xxx VALUES(xxx)，必须显示指定插入的列属性解读：容易在增加或者删除字段后出现程序BUG
- (25) 禁止使用属性隐式转换解读：SELECT uid FROM t_user WHERE phone=13800000000 会导致全表扫描，而不能命中phone索引，猜猜为什么？（这个线上问题不止出现过一次）
- (26) 禁止在WHERE条件的属性上使用函数或者表达式解读：SELECT uid FROM t_user WHERE from_unixtime(day)>='2017-01-15' 会导致全表扫描正确的写法是：SELECT uid FROM t_user WHERE day>=unix_timestamp('2017-01-15 00:00:00')
- (27) 禁止负向查询，以及%开头的模糊查询解读：
- a) 负向查询条件：NOT、!=、<>、!<、!>、NOT IN、NOT LIKE等，会导致全表扫描
- b) %开头的模糊查询，会导致全表扫描
- (28) **禁止使用JOIN查询，禁止大表使用子查询解读：会产生临时表，消耗较多内存与CPU，极大影响数据库性能**
- (29) **禁止使用OR条件，必须改为IN查询解读：旧版本Mysql的OR查询是不能命中索引的，即使能命中索引，为何要让数据库耗费更多的CPU帮助实施查询优化呢？**
- (30) 应用程序必须捕获SQL异常，并有相应处理
- (31) 同表的增删字段、索引合并一条DDL语句执行，提高执行效率，减少与数据库的交互。

总结

大数据量高并发的互联网业务，极大影响数据库性能的都不让用，不让用哟。

三、数据库索引

Hash索引

B+索引

索引的作用是“排列好次序，使得查询时可以快速找到”。

唯一索引

唯一索引是在表上一个或者多个字段组合建立的索引，这个或者这些字段的值组合起来在表中不可以重复。如学生表中的‘学号’

非唯一索引

非唯一索引是在表上一个或者多个字段组合建立的索引，这个或者这些字段的值组合起来在表中可以重复，不要求唯一。如学生表中的‘成绩’

主键索引

主键索引（主索引）是唯一索引的特定类型。表中创建主键时自动创建的索引。一个表只能建立一个主索引。

聚集索引（聚簇索引）

聚集索引（聚簇索引），表中记录的物理顺序与键值的索引顺序相同。一个表只能有一个聚集索引。

扩展：聚集索引和非聚集索引的区别？分别在什么情况下使用？

聚集索引和非聚集索引的根本区别是表中记录的物理顺序和索引的排列顺序是否一致。

聚集索引的表中记录的物理顺序与索引的排列顺序一致。

优点是查询速度快，因为一旦具有第一个索引值的记录被找到，具有连续索引值的记录也一定物理的紧跟其后。

缺点是对表进行修改速度较慢，这是为了保持表中的记录的物理顺序与索引的顺序一致，而把记录插入到数据页的相应位置，必须在数据页中进行数据重排，降低了执行速度。在插入新记录时数据文件为了维持 B+Tree 的特性而频繁的分裂调整，十分低效。

建议使用聚集索引的场合为：

- 某列包含了小数目的不同值。
- 排序和范围查找。

非聚集索引的记录的物理顺序和索引的顺序不一致。

其他方面的区别：

1.聚集索引和非聚集索引都采用了 B+树的结构，但非聚集索引的叶子层并不与实际的数据页相重叠，而采用叶子层包含一个指向表中的记录在数据页中的指针的方式。聚集索引的叶节点就是数据节点，而非聚集索引的叶节点仍然是索引节点。

2.非聚集索引添加记录时，不会引起数据顺序的重组。看上去聚簇索引的效率明显要低于非聚簇索引，因为每次使用辅助索引检索都要经过两次 B+树查找，这不是多此一举吗？聚簇索引的优势在哪？

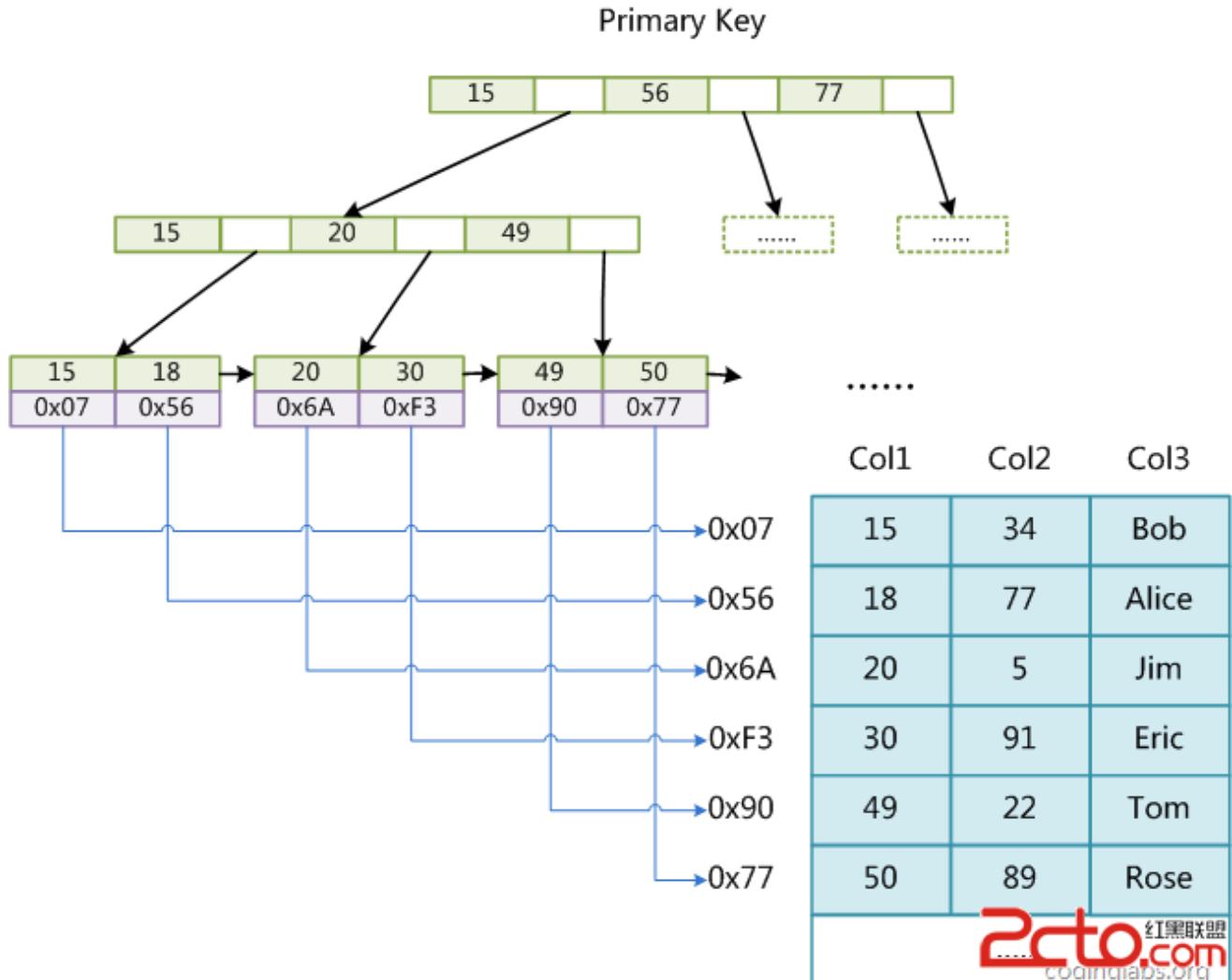
- 由于行数据和叶子节点存储在一起，这样主键和行数据是一起被载入内存的，找到叶子节点就可以立刻将行数据返回了，如果按照主键 Id 来组织数据，获得数据更快。
- 辅助索引使用主键作为“指针”，而不是使用地址值作为指针的好处是，减少了当出现行移动或者数据页分裂时，辅助索引的维护工作，InnoDB 在移动行时无须更新辅助索引中的这个“指针”。也就是说行的位置会随着数据库里数据的修改而发生变化，使用聚簇索引就可以保证不管这个主键 B+树的节点如何变化，辅助索引树都不受影响。

建议使用非聚集索引的场合为：

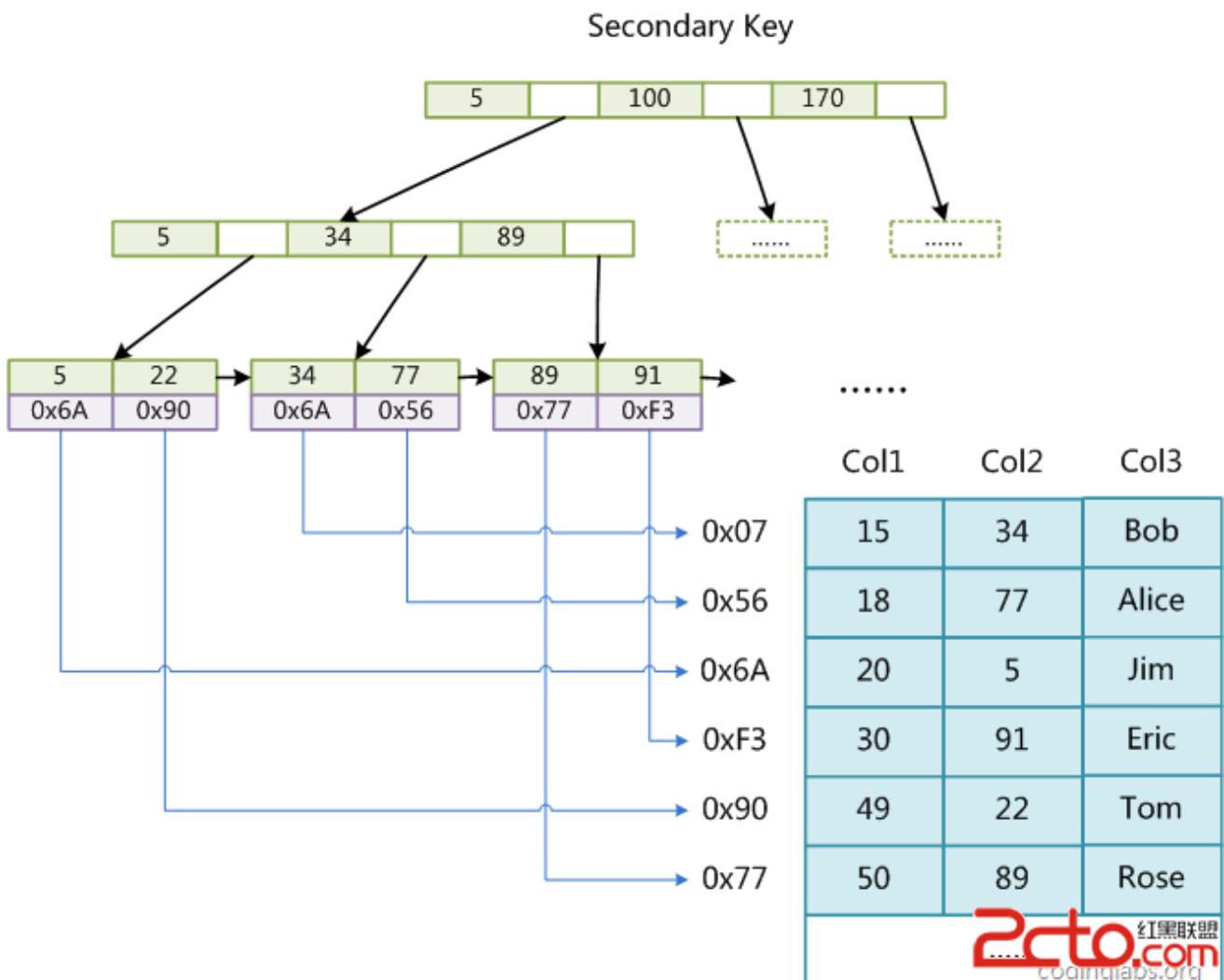
- 此列包含了大数目的不同值；
- 频繁更新的列

索引实现机制

MyISAM引擎使用B+Tree作为索引结构，叶节点的数据域存放的是数据记录的地址。下图是MyISAM索引的原理图：



这里设表一共有三列，假设我们以Col1为主键，则上图是一个MyISAM表的主索引（Primary key）示意。可以看出MyISAM的索引文件仅仅保存数据记录的地址。在MyISAM中，主索引和辅助索引（Secondary key）在结构上没有任何区别，只是主索引要求key是唯一的，而辅助索引的key可以重复。如果我们在Col2上建立一个辅助索引，则此索引的结构如下图所示：



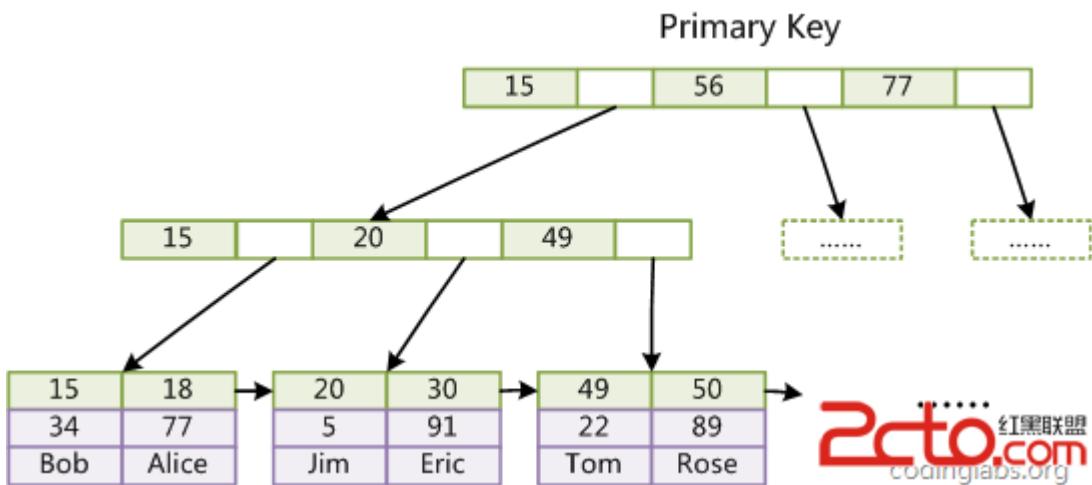
同样也是一颗B+Tree，data域保存数据记录的地址。因此，MyISAM中索引检索的算法为首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其data域的值，然后以data域的值为地址，读取相应数据记录。

MyISAM的索引方式也叫做“非聚集”的，之所以这么称呼是为了与InnoDB的聚集索引区分。

InnoDB索引实现

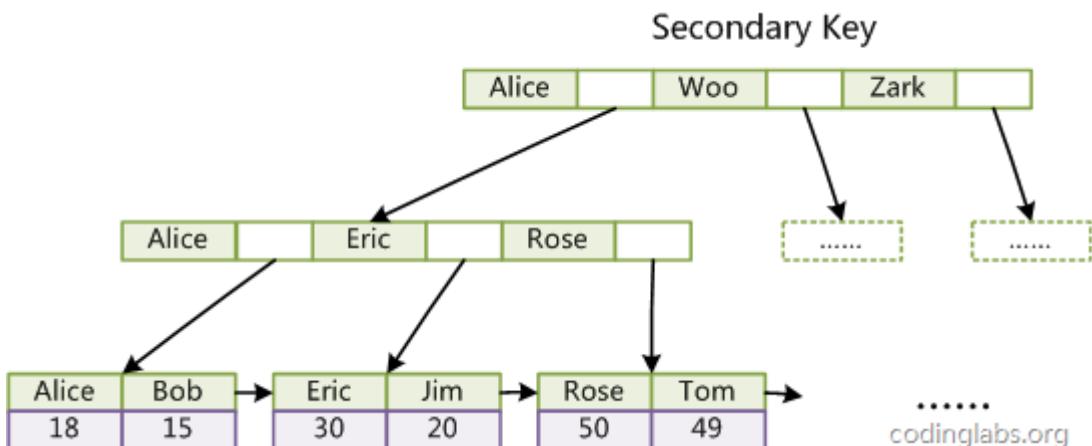
虽然InnoDB也使用B+Tree作为索引结构，但具体实现方式却与MyISAM截然不同。

第一个重大区别是InnoDB的数据文件本身就是索引文件。从上文知道，MyISAM索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。而在InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。



上图是InnoDB主索引（同时也是数据文件）的示意图，可以看到叶节点包含了完整的数据记录。这种索引叫做聚集索引。因为InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键（MyISAM可以没有），如果没有显式指定，则MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。

第二个与MyISAM索引的不同是InnoDB的辅助索引data域存储相应记录主键的值而不是地址。换句话说，InnoDB的所有辅助索引都引用主键作为data域。例如，下图为定义在Col3上的一个辅助索引：



这里以英文字符的ASCII码作为比较准则。聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录。

了解不同存储引擎的索引实现方式对于正确使用和优化索引都非常有帮助，例如知道了InnoDB的索引实现后，就很容易明白为什么不建议使用过长的字段作为主键，因为所有辅助索引都引用主索引，过长的主索引会令辅助索引变得过大。再例如，用非单调的字段作为主键在InnoDB中不是个好主意，因为InnoDB数据文件本身是一颗B+Tree，非单调的主键会造成在插入新记录时数据文件为了维持B+Tree的特性而频繁的分裂调整，十分低效，而使用自增字段作为主键则是一个很好的选择。

索引建立原则

(id, name) where id=1 and name='xxx'

1. **最左前缀匹配原则**，mysql 会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，范围查询会导致组合索引半生效。比如 a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，c 可以用到索引，d 是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d 的顺序可以任意调整。where 范围查询要放在最后（这不绝对，但可以利用一部分索引）。

2. 特别注意：and 之间的部分可以乱序，比如 a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任意顺序，mysql 的查询优化器会帮你优化成索引可以识别的形式。where 字句有 or 出现还是会遍历全表。
3. 尽量选择区分度高的字段作为索引，某字段的区分度的公式是 $\text{count}(\text{distinctcol})/\text{count}(*)$ ，表示字段不重复的比例，比例越大，我们扫描的记录数越少，查找匹配的时候可以过滤更多的行，唯一索引的区分度是 1，而一些状态、性别字段可能在大数据面前区分度就是 0。
4. 不在索引列做运算或者使用函数。
5. 尽量扩展索引，不要新建索引。比如表中已经有 a 的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可。
6. Where 子句中经常使用的字段应该创建索引，分组字段或者排序字段应该创建索引，两个表的连接字段应该创建索引。
7. like 模糊查询中，右模糊查询（321%）会使用索引，而%321 和%321%会放弃索引而使用全局扫描。

四、 MyISAM vs InnoDB

Mysql [数据库](#)中，最常用的两种引擎是 innodb 和 myisam。InnoDB 是 Mysql 的默认存储引擎。

1.事务处理上方面：

MyISAM 强调的是性能，查询的速度比 InnoDB 类型更快，但是不提供事务支持。InnoDB 提供事务支持事务。

2.外键： MyISAM 不支持外键， InnoDB 支持外键。

3.锁方面的介绍：

MyISAM 只支持表级锁，InnoDB 支持行级锁和表级锁，默认是行级锁，行锁大幅度提高了多用户并发操作的性能。innodb 比较适合于插入和更新操作比较多的情况，而 myisam 则适合用于频繁查询的情况。另外，InnoDB 表的行锁也不是绝对的，如果在执行一个 SQL 语句时，MySQL 不能确定要扫描的范围，InnoDB 表同样会锁全表，例如 update table set num=1 where name like "%aaa%"。

4.全文索引：

MyISAM 支全文索引， InnoDB 不支持全文索引。innodb 从 [mysql](#)5.6 版本开始提供对全文索引的支持。

5.表主键的区别：

MyISAM：允许没有主键的表存在。

InnoDB：如果没有设定主键，就会自动生成一个 6 字节的主键(用户不可见)。

6.表的具体行数问题：

MyISAM：select count() from table,MyISAM 只要简单的读出保存好的行数。因为MyISAM 内置了一个计数器，count()时它直接从计数器中读。

InnoDB：不保存表的具体行数，也就是说，执行 select count(*) from table 时，InnoDB要扫描一遍整个表来计算有多少行。

一张表，里面有 ID 自增主键，当 insert 了 17 条记录之后，删除了第 15,16,17 条记录，再把 Mysql 重启，再 insert 一条记录，这条记录的 ID 是 18 还是 15？

如果表的类型是 MyISAM， 那么是 18。因为 MyISAM 表会把自增主键的最大 ID 记录到数据文件里，重启MySQL 自增主键的最大 ID 也不会丢失。

如果表的类型是 InnoDB， 那么是 15。InnoDB 表只是把自增主键的最大 ID 记录到内存中，所以重启数据库会导致最大 ID 丢失。

五、并发事务带来的问题

丢失更新

如果两个事务都要更新数据库一个字段X, $x=100$

事务A	事务B
读取 $X = 100$	读取 $X = 100$
写入 $x = X+100$	写入 $x = X+200$
事务结束 $x= 200$	事务结束 $x=300$
	最后 $x=300$

两个不同事物同时获得相同数据，然后在各自事务中同时修改了该数据，那么先提交的事务更新会被后提交事务的更新给覆盖掉，这种情况事务A的更新就被覆盖掉了、丢失了。

脏读（未提交读）

防止一个事务读到另一个事务还没有提交的记录。如：

事务A	事务B
	写入 $x = X+100$ ($x=200$)
读取 $X = 200$ (读取了事务B未提交的数据)	
	事务回滚 $x=100$
	事务结束 $x=100$
事务结束	

事务读取了未提交的数据，事务B的回滚，导致了事务A的数据不一致，导致了事务A的脏读！

不可重复读

一个事务在自己没有更新数据库数据的情况下，同一个查询操作执行两次或多次的结果应该是一致的；如果不一致，就说明为不可重复读。还是用上面的例子

事务A	事务B
读取X = 100	读取X = 100
读取X = 100	写入x = X+100
	事务结束, x=200
读取X = 200 (此时, 在同一个事务A中, 读取的X值发生了变化!)	
事务结束	

这种情况事务A多次读取x的结果出现了不一致, 即为**不可重复读**。

幻读 (Phantom Read)

事务A读的时候读出了15条记录, 事务B在事务A执行的过程中 增加了1条, 事务A再读的时候就变成了 16 条, 这种情况就叫做幻影读。不可重复读说明了做数据库读操作的时候可能会出现的问题。

六、事务隔离级别及锁的实现机制

排他锁 写锁 被加锁的对象只能被持有锁的事务读取和修改, 其他事务无法在该对象上加其他锁, 也不能读取和修改该对象

共享锁 **读锁** 被加锁的对象可以被持锁事务读取, 但是不能被修改, 其他事务也可以在上面再加共享锁。

特别的, 对共享锁: 如果两个事务对同一个资源上了共享锁, 事务A 想更新该数据, 那么它必须等待 事务B 释放其共享锁。

在运用 排他锁 和 共享锁 对数据对象加锁时, 还需要约定一些规则, 例如何时申请 排他锁 或 共享锁、持锁时间、何时释放等。称这些规则为封锁协议 (Locking Protocol) 。对封锁方式规定不同的规则, 就形成了各种不同的封锁协议。

一级封锁协议 (对应 read uncommited)

一级封锁协议是: 事务 在对需要修改的数据上面 (就是在发生修改的瞬间) 对其加共享锁 (其他事务不能更改, 但是可以读取-导致“脏读”), 直到事务结束才释放。事务结束包括正常结束 (COMMIT) 和非正常结束 (ROLLBACK) 。

一级封锁协议不能避免 **丢失更新, 脏读, 不可重复读, 幻读!**

二级封锁协议 (对应read commited)

二级封锁协议是:

- 1) 事务 在对需要更新的数据 上 (就是发生更新的瞬间) 加 排他锁 (直到事务结束), 防止其他事务读取未提交的数据, 这样, 也就避免了 “脏读” 的情况。
- 2) 事务 对当前被读取的数据 上面加共享锁 (当读到时加上共享锁), 一旦读完该行, 立即 释放该该行的共享锁 - 上面只能防止不读脏数据
- 2) 事务 对当前被读取的数据 上面加共享锁 (当读到时加上共享锁), 直到事务结束才释放

可以防止不可重复读

从数据库的底层实现更深入的来理解，既是，数据库会对游标当前的数据上加**共享锁**，但是当游标离开当前行的时候，立即释放该行的共享锁。

二级封锁协议除防止了“脏读”数据，但是不能避免**丢失更新，不可重复读，幻读**。

但在二级封锁协议中，由于读完数据后**立即**释放共享锁，所以它不能避免**可重复读**，同时它也不能避免**丢失更新**，如果事务A、B同时获取资源X，然后事务A先发起更新记录X，那么事务B将等待事务A执行完成，然后获得记录X的排他锁，进行更改。这样事务A的更新将会被丢失。具体情况如下：

事务A	事务B
读取X=100 (同时上共享锁)	读取X=100 (同时上共享锁)
读取成功 (释放共享锁)	读取成功 (释放共享锁)
UPDATE X=X+100 (上排他锁)	
	UPDATING A (等待事务A释放对X的排他锁)
事务成功 (释放排他锁) X=200	
	UPDATE X=X+200 (成功上排他锁)
	事务成功 (释放排他锁) X=300

由此可以看到，事务A的提交被事务B覆盖了，所以不能防止**丢失更新**。

如果要避免**丢失更新，我们需要额外的操作**，对凡是读到的数据加**共享锁**和**排他锁**，这个往往需要程序员自己编程实现，比如在Oracle中，需要加SELECT FOR UPDATE语句，表明，凡是该事务读到的数据，额外的加上排他锁，防止其他数据同一时间获取相同数据，这样就防止了**丢失更新**！

三级封锁协议（对应repeatable read）

三级封锁协议是：二级封锁协议加上事务在读取数据的瞬间必须先对其加**共享锁，但是直到事务结束才释放**，这样保证了可重复读（既是其他的事务只能读取该数据，但是不能更新该数据）。

三级封锁协议除防止了“脏”数据和不可重复读。但是这种情况不能避免**幻读和丢失更新**的情况，在事务A没有完成之前，事务B可以新增数据，那么当事务A再次读取的时候，事务B新增的数据会被读取到，这样，在该封锁协议下，**幻读**就产生了。如果事务A和事务B同时读取了资源X=100，同样，如果事务A先对X进行更新X=X+100，等待事务A执行完成X=200，那么事务B获得X的排他锁，进行更新X=X+200，然后提交X=300，同样A的更新被B所覆盖！（如果要避免**丢失更新，我们需要额外的操作**，对凡是读到的数据加**共享锁**和**排他锁**，这个往往需要程序员自己编程实现，比如在Oracle中，需要加SELECT FOR UPDATE语句，表明，凡是读到的数据，我会加排他锁，防止其他数据同一时间获取相同数据）！

进阶：repeatable read 导致死锁的情况（即便是不同的资源在相同的顺序下获取）。比如事务1读取A，同时事务2也读取A，那么事务1和事务2同时对A上了共享锁，然后事务1要UPDATE A，而此时事务2也要UPDATE A，这个时候事务1等待事务2释放其在A上的共享锁，然后事务2要等待事务1释放其在A上的共享锁，这样，事务1和事务2相互等待，产生死锁！（SQL Server/DB2里面有UPDATE LOCK可以解决这种情况，具体的

思路是，在 repeatable read 的情况下，将读取的数据上的 UPDATE 锁，介于 共享锁 和 排他锁之间的一种锁，该锁的作用是当出现上面这种情况后，事务1 和 事务2 对 A 上的是 UPDATE 锁，那么谁先要修改 A，那么该事务就会将 UPDATE 锁可以顺利升级为 排他锁对该数据进行修改！）

最强封锁协议（对应Serialization）

四级封锁协议是对三级封锁协议的增强，其实现机制也最为简单，直接对事务中所读取或者更改的数据所在的表加表锁，也就是说，其他事务不能读写该表中的任何数据。这样所有的 脏读，不可重复读，幻读，都得以避免！

七、MVCC（多版本并发控制）

mysql的innodb采用的是行锁，而且采用了多版本并发控制来提高读操作的性能

MVCC只在REPEATABLE READ和READ COMMITTED两个隔离级别下工作，其它两个隔离级别下不存在MVCC

什么是多版本并发控制呢？其实就是在每一行记录的后面增加两个隐藏列，记录创建版本号和删除版本号，而每一个事务在启动的时候，都有一个唯一的递增的版本号。

1、在插入操作时：记录的创建版本号就是事务版本号。

比如我插入一条记录，事务id假设是1，那么记录如下：也就是说，创建版本号就是事务版本号。

id	name	create version	delete version
1	test	1	

2、在更新操作的时候，采用的是先标记旧的那行记录为已删除，并且删除版本号是事务版本号，然后插入一行新的记录的方式。

比如，针对上面那行记录，事务Id为2要把name字段更新

```
update table set name= 'new_value' where id=1;
```

id	name	create version	delete version
1	test	1	2
1	new_value	2	

3、删除操作的时候，就把事务版本号作为删除版本号。比如

```
delete from table where id=1;
```

id	name	create version	delete version
1	new_value	2	3

4、查询操作：

从上面的描述可以看到，在查询时要符合以下两个条件的记录才能被事务查询出来：

1) InnoDB只查找版本早于当前事务版本的数据行（也就是，行的系统版本号小于或等于事务的系统版本号），这样可以确保事务读取的行，只么是在事务开始前已经存在的，要么是事务自身插入或者修改过的。

2) 行的删除版本要么未定义，要么大于当前事务版本号。这可以确保事务读取到的行，在事务开始之前未被删除。

这样就保证了各个事务互不影响。从这里也可以体会到一种提高系统性能的思路，就是：

通过版本号来减少锁的争用。

另外，只有**read-committed**和**repeatable-read**两种事务隔离级别才能使用**MVCC**

read-uncommitted由于是读到未提交的，所以不存在版本的问题

而**Serializable**则会对所有读取的行加锁。

八、间隙锁与幻读

间隙锁 (Next-Key锁)

当我们用范围条件而不是相等条件检索数据，并请求共享或排他锁时，InnoDB会给符合条件的已有数据记录的索引项加锁；对于键值在条件范围内但并不存在的记录，叫做“间隙 (GAP)”，InnoDB也会对这个“间隙”加锁，这种锁机制就是所谓的间隙锁 (Next-Key锁)。

举例来说，假如emp表中只有101条记录，其empid的值分别是 1,2,...,100,101，下面的SQL：

```
Select * from emp where empid > 100 for update;
```

是一个范围条件的检索，InnoDB不仅会对符合条件的empid值为101的记录加锁，也会对empid大于101（这些记录并不存在）的“间隙”加锁。

InnoDB使用间隙锁的目的，一方面是为了防止幻读，以满足相关隔离级别的要求，对于上面的例子，要是不使用间隙锁，如果其他事务插入了empid大于100的任何记录，那么本事务如果再次执行上述语句，就会发生幻读；另外一方面，是为了满足其恢复和复制的需要。有关其恢复和复制对锁机制的影响，以及不同隔离级别下InnoDB使用间隙锁的情况，在后续的章节中会做进一步介绍。

很显然，在使用范围条件检索并锁定记录时，InnoDB这种加锁机制会阻塞符合条件范围内键值的并发插入，这往往会造成严重的锁等待。因此，在实际应用开发中，尤其是并发插入比较多的应用，我们要尽量优化业务逻辑，尽量使用相等条件来访问更新数据，避免使用范围条件。

RR级别下防止幻读

快照读：使用MVCC防止幻读

当前读：使用间隙锁防止幻读

设计模式与实践

一、OOP五大原则SOLID

S.O.L.I.D是面向对象设计和编程(OOD&OOP)中几个重要编码原则(Programming Principle)的首字母缩写。

SRP	The Single Responsibility Principle	单一责任原则
OCP	The Open Closed Principle	开放封闭原则
LSP	The Liskov Substitution Principle	里氏替换原则
DIP	The Dependency Inversion Principle	依赖倒置原则
ISP	The Interface Segregation Principle	接口分离原则

单一责任原则

当需要修改某个类的时候原因有且只有一个 (THERE SHOULD NEVER BE MORE THAN ONE REASON FOR A CLASS TO CHANGE)。换句话说就是让一个类只做一种类型责任，当这个类需要承当其他类型的责任的时候，就需要分解这个类。



SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

开放封闭原则

开闭原则的意思是：对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。简言之，是为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。



OPEN CLOSED PRINCIPLE

Open Chest Surgery Is Not Needed When Putting On A Coat

里氏替换原则

里氏代换原则是面向对象设计的基本原则之一。里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。LSP 是继承复用的基石，只有当派生类可以替换掉基类，且软件单位的功能不受到影响时，基类才能真正被复用，而派生类也能够在基类的基础上增加新的行为。里氏代换原则是对开闭原则的补充。实现开闭原则的关键步骤就是抽象化，而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规

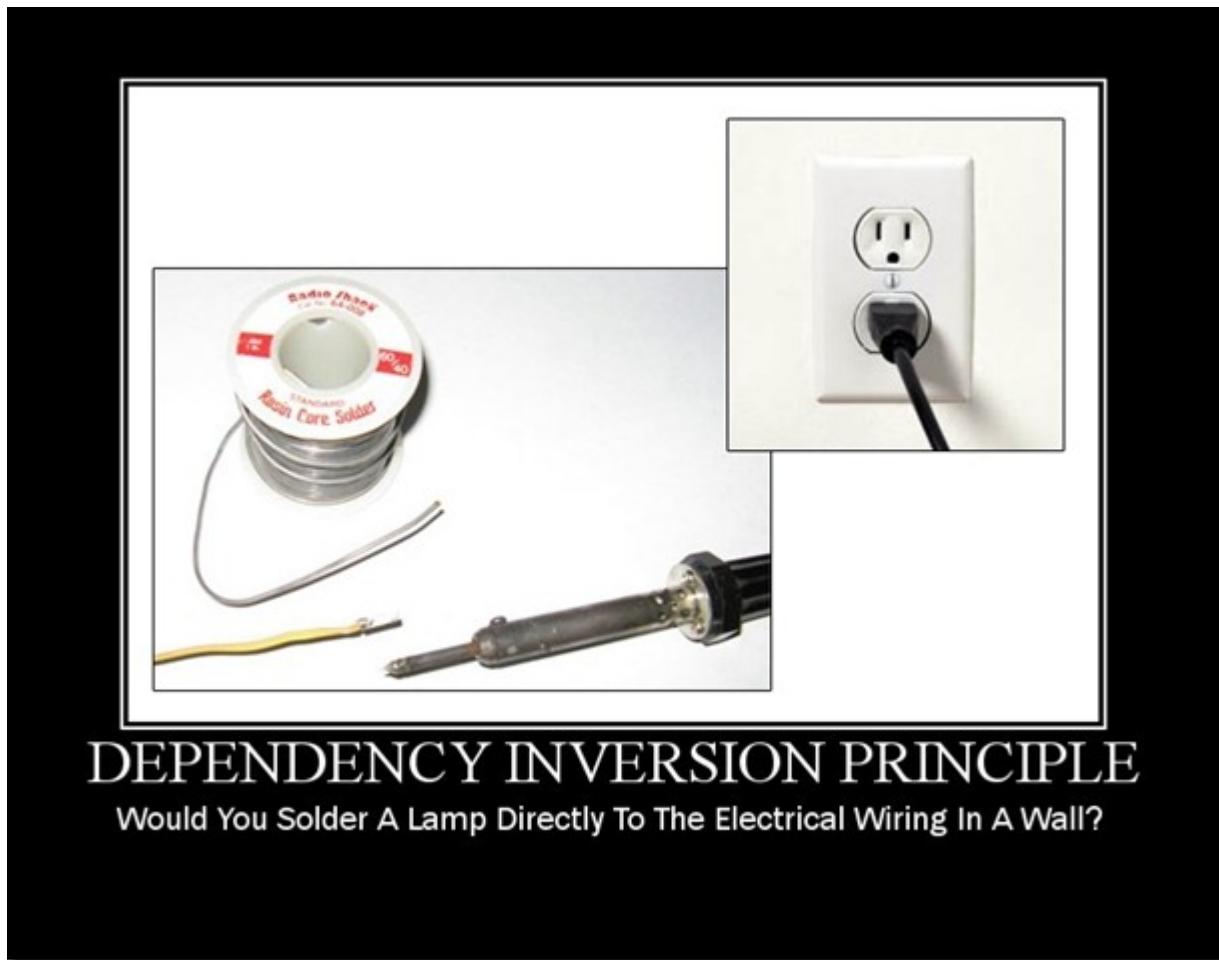


LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

依赖倒置原则

1. 高层模块不应该依赖于低层模块，二者都应该依赖于抽象
2. 抽象不应该依赖于细节，细节应该依赖于抽象



DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

接口分离原则

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。它还有另外一个意思是：降低类之间的耦合度。由此可见，其实设计模式就是从大型软件架构出发、便于升级和维护的软件设计思想，它强调降低依赖，降低耦合。



INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

这几条原则是非常基础而且重要的面向对象设计原则。正是由于这些原则的基础性，理解、融汇贯通这些原则需要不少的经验和知识的积累。上述的图片很好的注释了这几条原则。

二、设计模式

设计模式（Design pattern）代表了最佳的实践，通常被有经验的面向对象的软件开发人员所采用。设计模式是软件开发人员在软件开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。

设计模式是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了重用代码、让代码更容易被他人理解、保证代码可靠性。毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理地运用设计模式可以完美地解决很多问题，每种模式在现实中都有相应的原理来与之对应，每种模式都描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是设计模式能被广泛应用的原因。

序号	模式 & 描述	包括
1	创建型模式 这些设计模式提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用 new 运算符直接实例化对象。这使得程序在判断针对某个给定实例需要创建哪些对象时更加灵活。	工厂模式 (Factory Pattern) 抽象工厂模式 (Abstract Factory Pattern) 单例模式 (Singleton Pattern) 建造者模式 (Builder Pattern) 原型模式 (Prototype Pattern)
2	结构型模式 这些设计模式关注类和对象的组合。继承的概念被用来组合接口和定义组合对象获得新功能的方式。	适配器模式 (Adapter Pattern) 桥接模式 (Bridge Pattern) 过滤器模式 (Filter、Criteria Pattern) 组合模式 (Composite Pattern) 装饰器模式 (Decorator Pattern) 外观模式 (Facade Pattern) 享元模式 (Flyweight Pattern) 代理模式 (Proxy Pattern)
3	行为型模式 这些设计模式特别关注对象之间的通信。	责任链模式 (Chain of Responsibility Pattern) 命令模式 (Command Pattern) 解释器模式 (Interpreter Pattern) 迭代器模式 (Iterator Pattern) 中介者模式 (Mediator Pattern) 备忘录模式 (Memento Pattern) 观察者模式 (Observer Pattern) 状态模式 (State Pattern) 空对象模式 (Null Object Pattern) 策略模式 (Strategy Pattern) 模板模式 (Template Pattern) 访问者模式 (Visitor Pattern)

下面用一个图片来整体描述一下设计模式之间的关系：

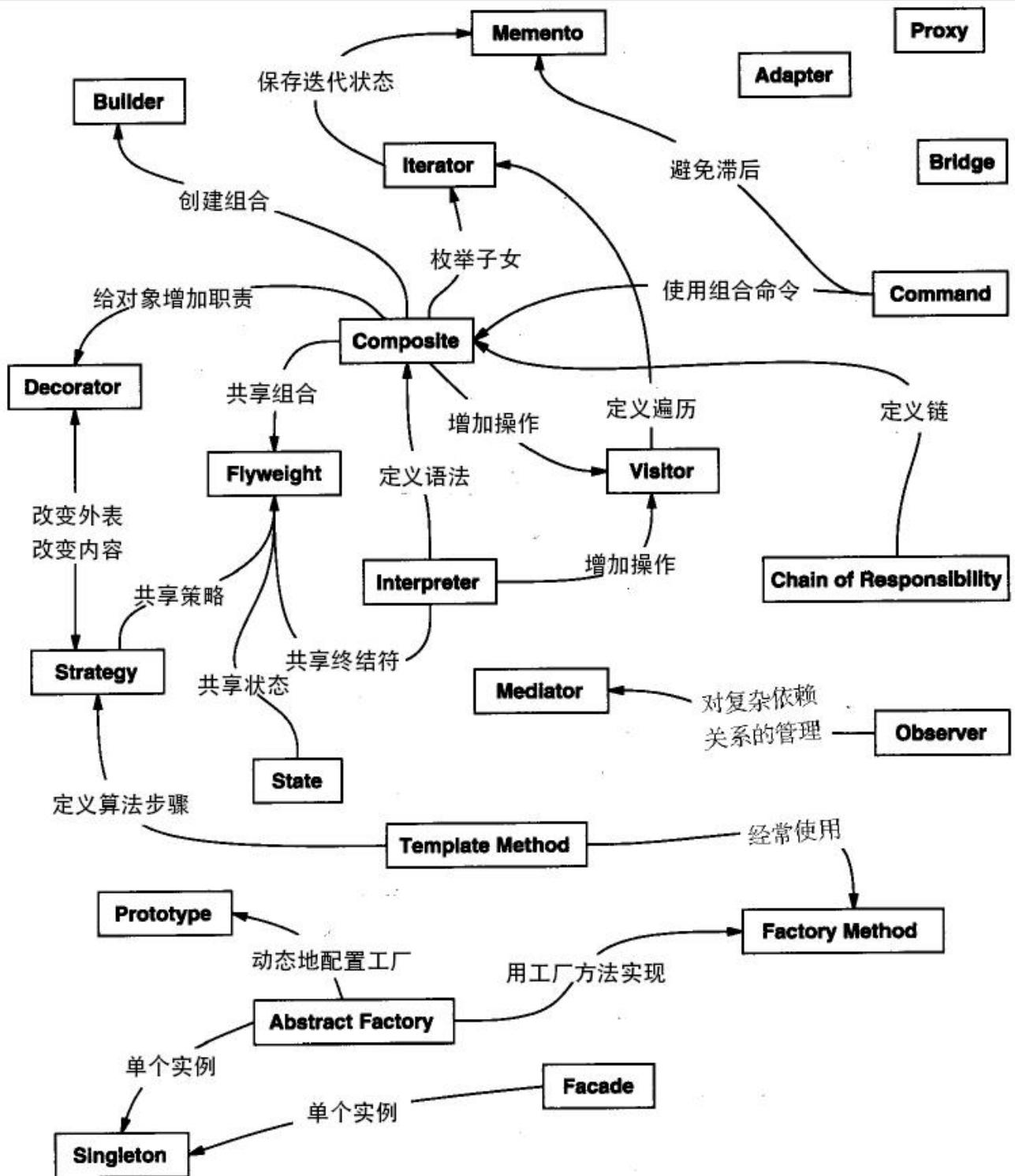


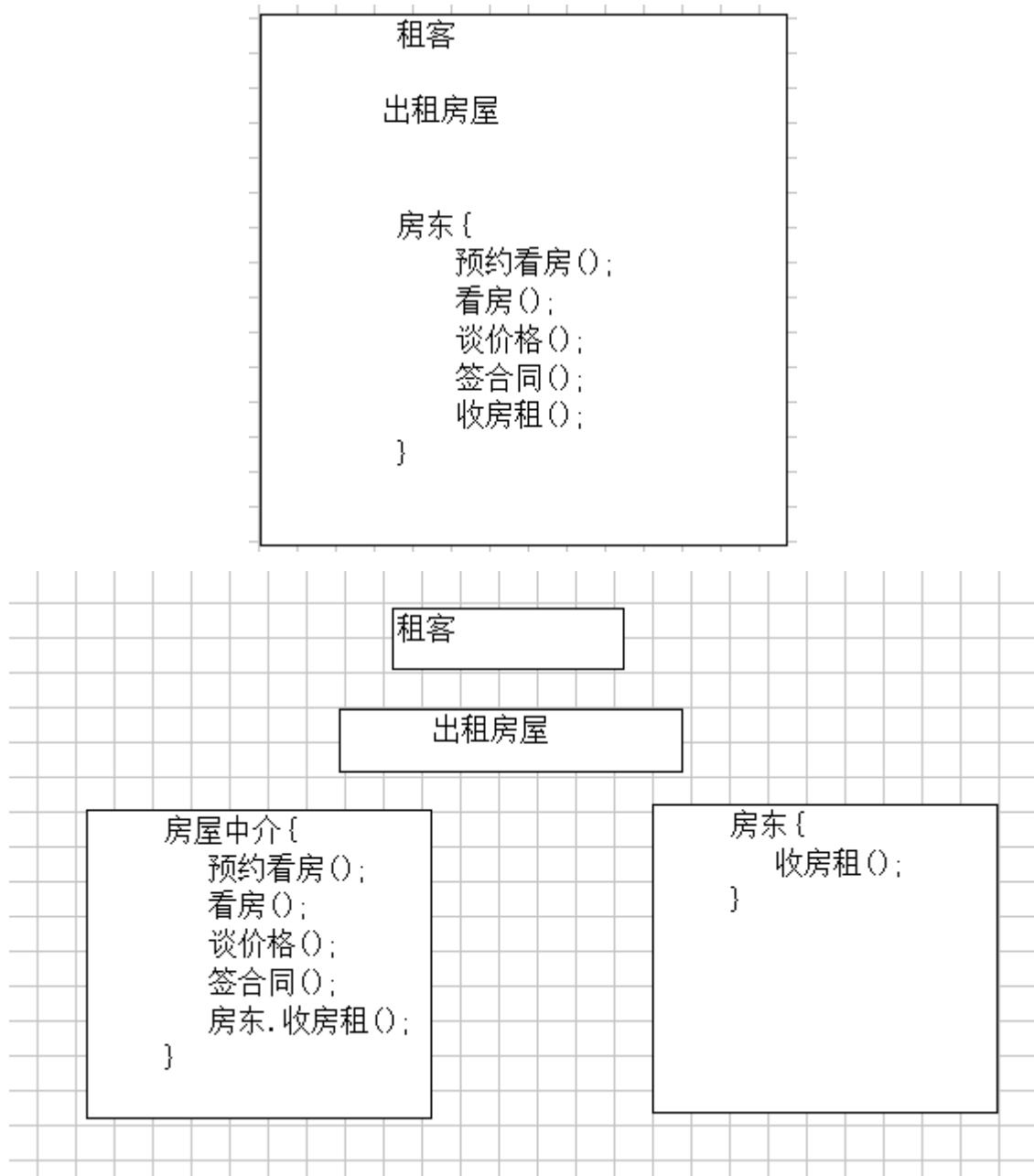
图 设计模式之间的关系

三、代理模式

定义与举例

为其他对象提供一种代理以控制对这个对象的访问。

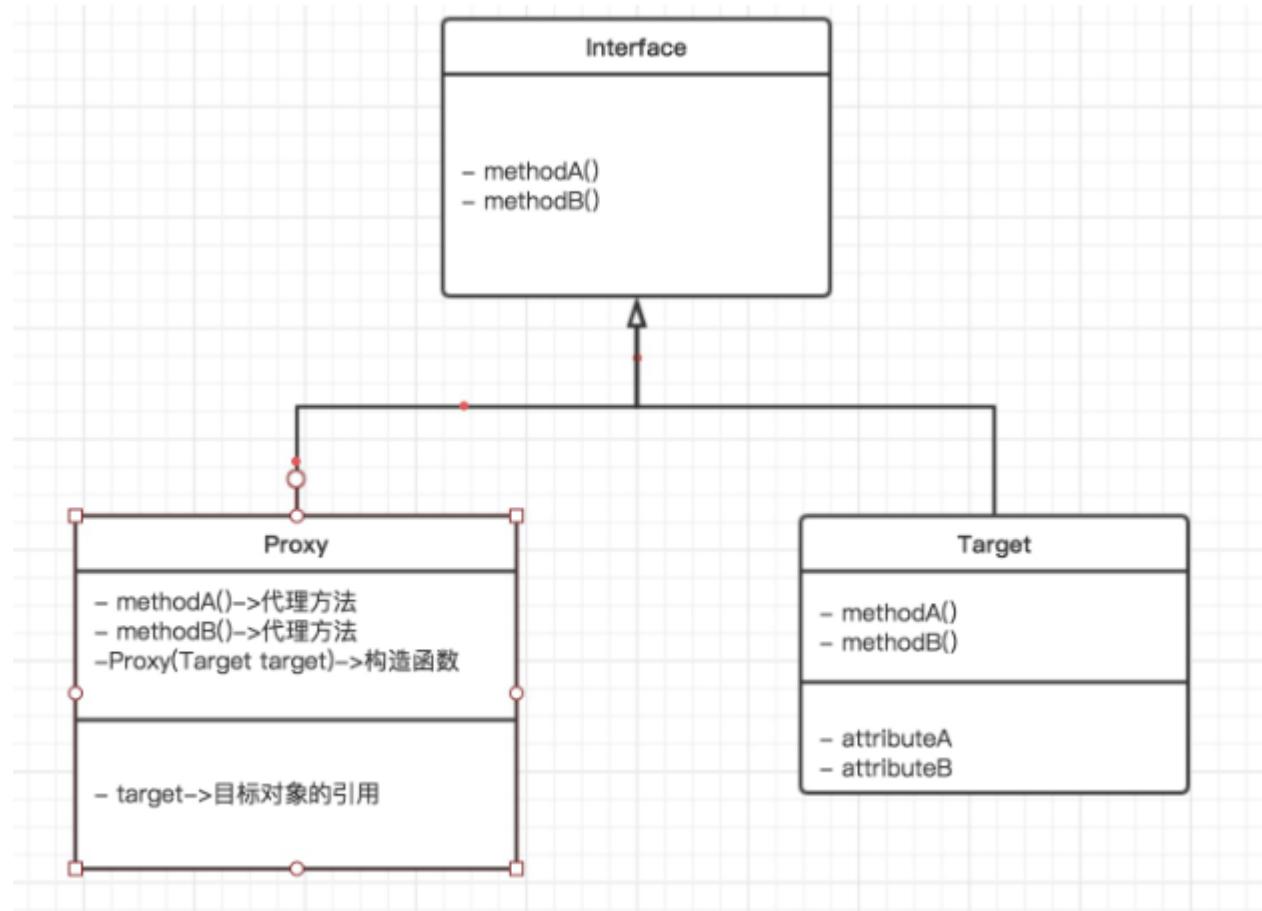
1, 其他对象：目标对象，想要访问的对象，常被称为被委托对象或被代理对象。 2, 提供一种代理：这里“一种”两个字比较重要，为什么不是提供一个呢？一种代表了某一类，即代理类和被代理类必须实现同一接口，这个接口下的所有实现类都能被代理访问到，其实只是单纯的为了实现代理访问功能，代理类不实现任何接口也能完成，不过针对于面向接口的编程，这种方式更易于维护和扩展，代理类实现接口，不管目标对象怎么改或改成谁，代理类不需要任何修改，而且任何目标对象能使用的地方都能用代理去替换。 3, 通过代理访问目标对象：代理类需要持有目标对象的引用，这样用户可以通过代理类访问目标对象，实现了用户与目标对象的解耦。 4, 访问：访问二字说明代理对象的目的是访问被代理类，业务逻辑的具体执行与其无关，由被代理对象完成。 5, 为什么要通过代理来访问：设计模式都是为了解决某一类的问题，可能目标对象不想让该用户访问或者是该用户无法访问到目标对象，这样就需要一个第三者来建立他们的联系，如代理服务器情景，被访问的服务器设置防火墙过滤掉某些地址的访问，这时用户就可以通过一个代理服务器来访问目标，使得目标服务器不用对外暴露细节，用户也能访问到想访问的数据。 5, 代理类功能增强：代理对象能直接访问到目标对象，这样它就能在调用目标对象的某个方法之前做一个预处理，在调用方法之后进行一些结尾工作，这样就对目标对象的方法进行了增强。但是我们不能说代理模式提供对象功能的增强，它的设计初衷是对代理对象施加控制，只是这种设计思路能达到功能增强的目的。



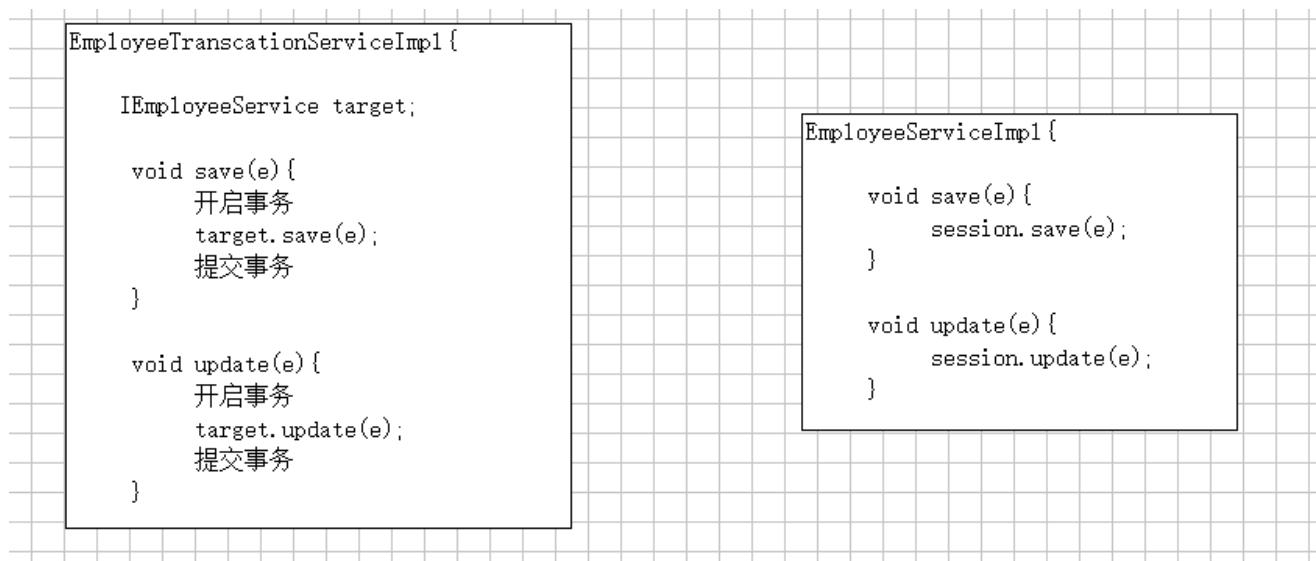
静态代理

静态代理模式就是如下图所示，构造三个类实现他们的关系。

静态代理由程序员创建或特定工具自动生成源代码，也就是在编译时就已经将接口，被代理类，代理类等确定下来。在程序运行之前，代理类的.class文件就已经生成。



静态代理举例，事务要开在service层上



静态代理分析:静态代理确实处理了代码污染的问题;

问题:

1,重复的代码仍然分散在了各个方法中;

2,需要为每一个真实对象写一个代理对象;

动态代理

代理类在程序运行时创建的代理方式被成为动态代理。我们上面静态代理的例子中，代理类(studentProxy)是自己定义好的，在程序运行之前就已经编译完成。然而动态代理，代理类并不是在Java代码中定义的，而是在运行时根据我们在Java代码中的“指示”动态生成的。相比于静态代理，动态代理的优势在于可以很方便的对代理类的函数进行统一的处理，而不用修改每个代理类中的方法。

也就是说，有了动态代理，我们只需要编写一份开始事务，提交事务的代码，然后再指明哪些方法需要事务就可以了，程序会自动地帮我们生成代理类

JDK动态代理

基于java的反射，适用于有接口的类

CGLIB动态代理

基于ASM字节码操纵技术，适用于没有接口的类

四、面向切面编程 (AOP)

基本思想

AOP 即 Aspect Oriented Program 面向切面编程

首先，在面向切面编程的思想里面，把功能分为核心业务功能，和周边功能。

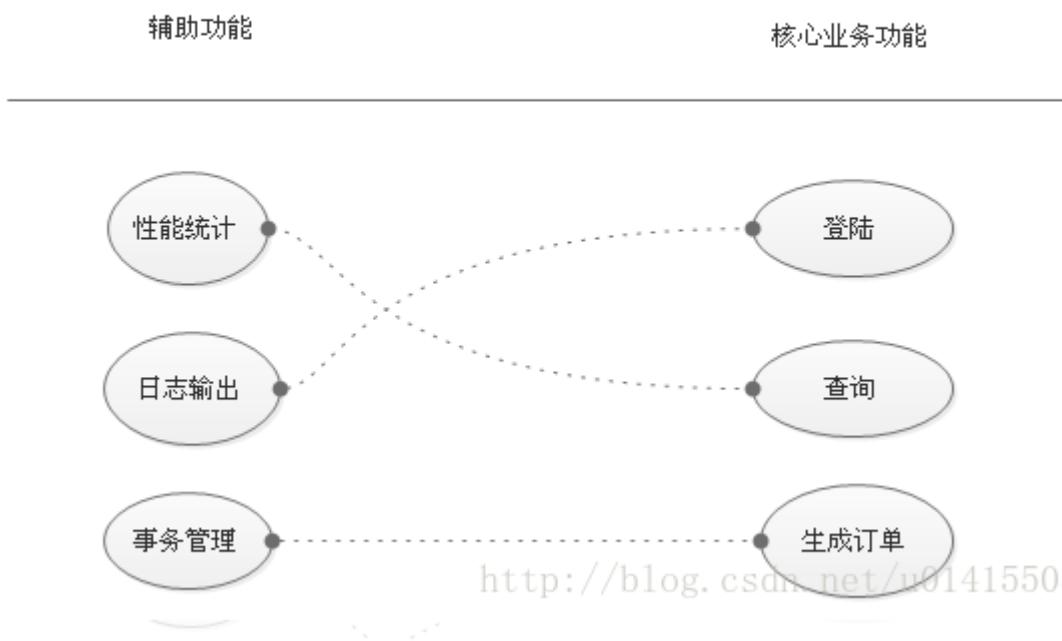
所谓的核心业务，比如登陆，增加数据，删除数据都叫核心业务

所谓的周边功能，比如性能统计，日志，事务管理等等

周边功能在Spring的面向切面编程AOP思想里，即被定义为切面

在面向切面编程AOP的思想里面，核心业务功能和切面功能分别独立进行开发

然后把切面功能和核心业务功能“编织”在一起，这就叫AOP



登录验证

将登录验证的代码写成一个方法，做成切面，织入到相关对象中，这样，访问资源之前会先验证是否登录，相关的代码我们只需要编写一份

基于RBAC的权限管理

类似于登录验证，AOP还可以做权限验证，思路与上述相同

角色访问控制 (RBAC)

简单理解为：谁扮演什么角色，被允许做什么操作用户对象：user：当前操作用户角色对象：role：表示权限操作许可权的集合权限对象：permission：资源操作许可权例子：张三（user）下载（permission）一个高清无码的种子（资源），需要VIP权限（role）张三--->普通用户--->授权---->VIP用户----->下载种子

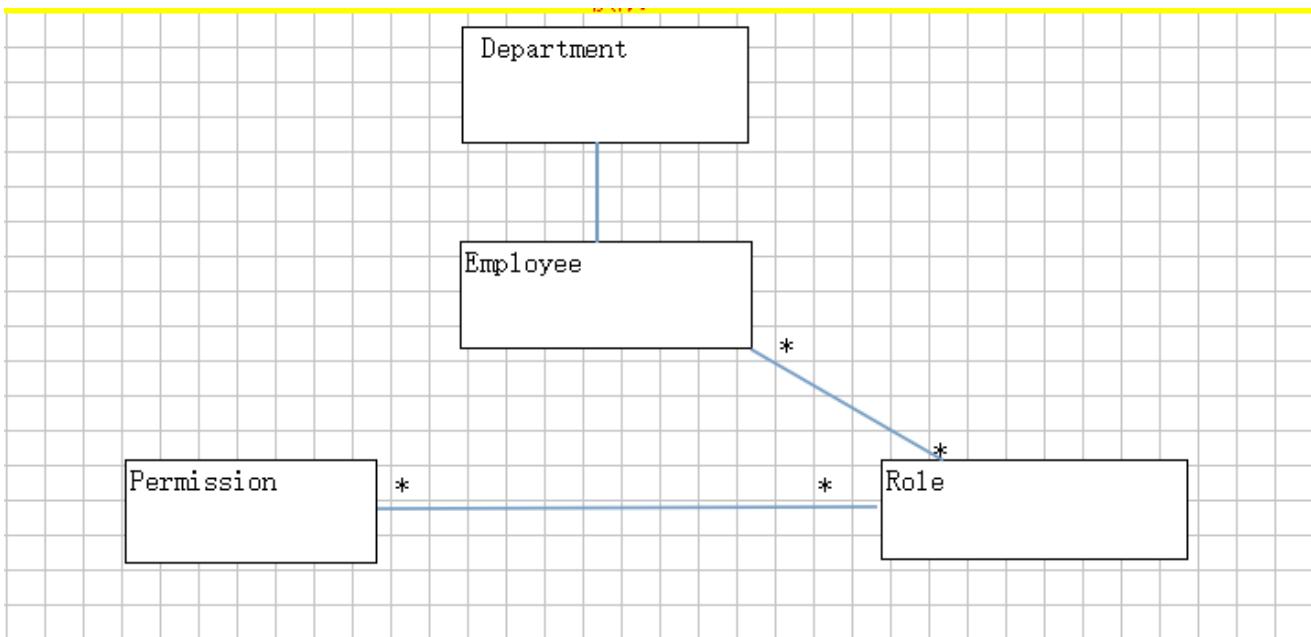
引入了Role的概念，目的是为了隔离User(即动作主体，Subject)与Privilege(权限，表示对Resource的一个操作，即Operation+Resource)。

Role作为一个用户(User)与权限(Privilege)的代理层，解耦了权限和用户的关系，所有的授权应该给予Role而不是直接给User或 Group。

Privilege是权限颗粒，由Operation和Resource组成，表示对Resource的一个Operation。例如，对于新闻的删除操作。Role-Privilege是many-to-many的关系，这就是权限的核心。

两大特征是

- 1.由于角色/权限之间的变化比角色/用户关系之间的变化相对要慢得多，减小了授权管理的复杂性，降低管理开销。
- 2.灵活地支持企业的安全策略，并对企业的变化有很大的伸缩性。



执行流程分析

Permission		
id	expression	
1	com.zqx.PermissionController:execute	name
2	com.zqx.PermissionController:edit	权限列表
3	com.zqx.PermissionController:delete	编辑权限
4	com.zqx.PermissionController:reload	删除权限
5	com.zqx.RoleController:execute	重新加载权限
6	com.zqx.RoleController:edit	角色列表
7	com.zqx.RoleController:delete	编辑角色
8	com.zqx.RoleController:save	删除角色
		添加角色

角色

id	name	List<Permission>
1	角色管理员	5,6,7,8
2	权限管理员	1,2,3,4

员工

id	name	List<Role>
1	小胖	1
2	小陈	2
3	班长	1,2

小胖这个用户登陆：

- 1, 检查用户名和密码；
- 2, 检查通过；
- 1, 得到小胖这个用户的对应的所有角色：R1
- 2, 根据所有的角色，得到小胖所有的权限信息：P5,P6,P7,P8
- 3, 把小胖所有的权限的expression放到一个set中；
- 4, 把小胖这个对象和他的权限列表放到session中；

添加微信获取更多Java架构视频：JaneS0307

小胖操作系统：

1, 点角色管理；

2, 请求被权限检查拦截器拦截到了(PermissionCheckInterceptor.intercepte);

 1, 得到当前请求的Controller和方法；

 2, 判断, 当前的这个方法是否是需要一个权限的；

 3, 如果当前方法不需要权限, 直接放行；

 4, 如果当前方法需要权限，

 1, 把当前请求的method变成com.zqx.RoleController:execute一个表达式；

 2, 在当前用户的permissionset中去看是否有这个表达式；

 3, 如果有, 放行；

 4, 如果没有, 直接导向到没有权限那个页面；

日志记录

当很多方法在开始、结束、抛异常时都需要记录，我们就可以采取上述AOP的思想来简化代码

日志记录最佳实践

1. **关键业务操作：**比如删除，更新等dml类型的操作的数据，尤其是要审计的日志一定要记录；推荐info级别：查询类的操作不推荐记录日志。
2. **异常日志：**如果是系统异常，比如网络不通，数据库连接失败等需要记录error日志。
3. **业务规则异常：**例如数据违反约束，这类推荐记录debug级别日志，不推荐info级别。这类信息正常情况下应该不需要关注，由程序返回值直接来实现，但是如果需要排查bug这部分信息还是很有价值。

1、在程序开始运行应该以INFO记录**程序开始运行**的消息。 2、在程序初始化过程中，如果影响程序主体正常运行错误出现，应该以FATAL记录出现错误的函数名、事件和错误号；如果只是一个不影响程序正常功能的模块出现错误，则应该以ERROR记录出现错误的**函数名、模块名、事件和错误号**。 3、在程序初始化完成后，应该以INFO记录程序初始化完成的消息。 4、在函数的入口，如果需要验证参数，则可以以DEBUG的形式**输出参数的信息**。如果重要参数不正确，则应该以ERROR输出。 5、在调用比较成熟的API时，如果失败，则以ERROR记录，并且有错误号记下错误号。 6、在调用没有经过严格测试的库时，即使返回成功，也要以DEBUG的形式记录下返回的结果。 7、以捕获异常时，以**ERROR记录下错误**。 8、在进行数据库操作时，以DEBUG的级别输出执行的SQL语句，对于取回的结果，最好是能打印出所有构造完成的对象的信息。 9、在与其它程序进行通信时，以DEBUG记录下通信过程中的重要信息。 10、对程序中的每个线程，它们的初始化完成和开始运行也要以INFO记录下来。 11、对程序中需要检查运行性能的地方，以DEBUG记录下运行耗时。 12、程序正常结束时，和初始化的记录方式相同，对各个模块的卸载采用和加载是一样的处理方式。当程序都卸载完成后以INFO记录**程序退出的消息**。

你应在适当级别上进行log

如果你遵循了上述第一点的做法，接下来你要对你程序中每一个log语句使用不同的log级别。其中最困难的一个任务是找出这个log应该是什么级别

以下是我的一些建议：

- TRACE level: 如果使用在生产环境中，这是一个代码异味(code smell)。它可用于开发过程中追踪bug，但不要提交到你的版本控制系统
- DEBUG level: 把一切东西都记录在这里。这在debug过程中最常用到。我主张在进入生产阶段前减少debug语句的数量，只留下最有意义的部分，在调试(troubleshooting)的时候激活。
- INFO level: 把用户行为(user-driven)和系统的特定行为(例如计划任务...)
- NOTICE level: 这是生产环境中使用的级别。把一切不认为是错误的，可以记录的事件都log起来
- WARN level: 记录在这个级别的事件都有可能成为一个error。例如，一次调用数据库使用的时间超过了预设时间，或者内存缓存即将到达容量上限。这可以让你适当地发出警报，或者在调试时更好地理解系统在failure之前做了些什么
- ERROR level: 把每一个错误条件都记录在这。例如API调用返回了错误，或是内部错误条件
- FATAL level: 末日来了。它极少被用到，在实际程序中也不应该出现多少。在这个级别上进行log意味着程序要结束了。例如一个网络守护进程无法bind到socket上，那么它唯一能做的就只有log到这里，然后退出运行。

记住，在你的程序中，默认的运行级别是高度可变的。例如我通常用INFO运行我的服务端代码，但是我的桌面程序用的是DEBUG。这是因为你很难在一台你没有接入权限的机器上进行调试，但你在做用户服务时，比起教他们怎么修改log level再把生成的log发给你，我的做法可以让你轻松得多。当然你可以有其他的做法：)

4. 你应该写有意义的log

这可能是最重要的建议了。没有什么比你深刻理解程序内部，却写出含糊的log更糟了。

在你写日志信息之前，总要提醒自己，有突发事件的时候，你唯一拥有的只有来自log文件，你必须从中明白发生了什么。这可能就是被开除和升职之间的微妙的差距。

当开发者写log的时候，它(log语句)是直接写在代码环境中的，在各种条件下我们应该写入基于当前环境的信息。不幸的是，在log文件中并没有这些环境，这可能导致这些信息无法被理解。

解决这个情况(在写warn和error level时尤为重要)的一个方法是，添加辅助信息到log信息中，如果做不到，那么改为把这个操作的作用写下。

还有，不要让一个log信息的内容基于上一个。这是因为前面的信息可能由于(与当前信息)处于不同的类别或者level而没被写入。更坏的情况是，它因多线程或异步操作，在另一个地方(或是以另一方式)出现。

日志信息应该用英语

这个建议可能有点奇怪，尤其是对法国佬(French guy)来说。我还是认为英语远比法语更简炼，更适应技术语言。如果一个信息里面包含超过50%的英语单词，你有什么理由去用法语写log呢

把英法之争丢一边，下面是这个建议背后的原因：

- 英语意味着你的log是用ASCII编码的。这非常重要，因为你不会真正知道log信息会发生什么，或是它被归档前经过何种软件层和介质。如果你的信息里面使用了特殊字符集，乃至UTF-8，它可能并不会被正确地显示(render)，更糟的是，它可能在传输过程中被损坏，变得不可读。不过这还有个问题，log用户输入时，可能有各种字符集或者编码。
- 如果你的程序被大多数人使用，而你又没有足够的资源做国际化，英语会成为你的不二之选。如果你有国际化，那么让界面与终端用户更亲近(closer)(这通常不会是你的log)
- 如果你国际化了你的log(例如所有的warning和error level信息)，给他们一个特定的有意义的错误码。这样，用户做与语言无关的搜索，找到相关信息。这种良好的模式已经在虚拟内存(VMS)操作系统中应用了很久，而我必须承认它非常有用。如果你曾经设计过这种模式，你还可以试试这种模式: APP-S-CODE 或者 APP-S-SUB-CODE，它们分别代表:
APP: 应用程序的3字缩写

S: 严重程度的1字缩写(例如D代表debug, I代表info)

SUB: 这个code所从属的应用程序的子部分

CODE: 一个数字代号，指定这个问题中的错误

你应该给log带上上下文

没有什么比这样的log信息更糟的了

```
Transaction failed
```

或是

```
User operation succeeds
```

又或是API异常时：

```
java.lang.IndexOutOfBoundsException
```

没有相应的上下文，这些信息不过是噪音，它们不会对调试过程中有意义的数值或是空间起作用(add value and consume space)。

带上上下文的信息要有价值得多，例如：

```
Transaction 234632 failed: cc number checksum incorrect
```

或是

```
User 54543 successfully registered e-mail<mailto:user@domain.com>user@domain.com</a>
```

又或是

```
IndexOutOfBoundsException: index 12 is greater than collection size 10
```

日志不宜太多或太少

这听着貌似很愚蠢。log的数量是有一个合适的平衡的。

太多的log会使从中获得有价值的东西变得困难。当人工地浏览这种十分混乱的log，尝试调试产品在早上3点的一个问题可不是一个好事。

太少的log，你可能无法调试问题：调试就像在拼一个困难的拼图，你需要得到足够的拼块。

不幸的是，这没有魔法般的规则去知道应该log些什么。所以需要严格地遵从第一第二点，程序可以变得很灵活，轻松地增减log的长度(verbosity)。

解决这个问题的一个方法是，在开发过程中尽可能多地进行log(不要被加入用于程序调试的log所迷惑)。当应用程序进入生产过程时，对生成的log进行一次分析，根据所发现的问题增减log语句。尤其是在调试时，在你需要的部分，你可以有更多的上下文或logging，确保在下一个版本中加入这些语句(可以的话，同时解决它来让这个问题在记忆中保持新鲜)。当然，这需要运维人员和开发者之间大量的交流。

这是一个复杂的任务，但是我推荐你重构logging语句，如你重构代码一样多。这样可以在产品的log和它的log语句的修改中有一个紧密的反馈循环。如果你的组织有一个连续的交付进程的话，它会十分有效，正如持续的重构。

Logging语句是与代码注释同级的代码元数据。保持logging语句与代码相同步是很重要的。没什么比调试时获得与所运行的代码毫无关系的信息更糟了。

你应该考虑阅读者

为什么要对应用程序做log

唯一的答案是，在某一天会有人去读它(或是它的意义)。更重要的是，猜猜谁会读它，这是很有趣的事。对于不同的“谁”，你将要写下的log信息的内容，上下文，类别和level会大不同。

这些“谁”包括：

- 一个尝试自己解决问题的终端用户(想象一个客户端或桌面程序)
- 一个在调试产品问题的系统管理员或者运维工程师
- 一个在开发中debug，或者在解决产品问题的开发者

开发者了解程序内部，所以给他的log信息可以比给终端用户的复杂得多。为你的目标阅读者调整你的表达方式，乃至为此加入额外的类别(dedicate separate catagories)。

你不应该只为调试而log

正如log会有不同的阅读者，它也有不同的使用理由。即便调试是最显而易见的阅读log的目的，你同样可以有效地把log用在：

- 审查：有时商业上会有需求。这可以获取与管理或者合法用户的有意义的事件。通常会有一些语句描述这个系统中的用户在做些什么(例如谁登录了，谁在编辑.....)
- 建档：log是打上了时间戳的(有时是微妙级的)，可以成为一个为程序各部分建档的好工具。例如记录一个操作的开始和结束，你可以自动化(通过解析log)或是在调试中，进行性能度量，而不需要把这些度量加到程序中。
- 统计：如果你每次对一个特定事件(例如特定的错误或事件)进行log，你可以对运行中的程序(或用户行为)进行有趣的统计。这可以添加(hook)到一个警报系统中去连续地发现大量error。

总结

我希望这可以帮助你生成更多有用的log。如果我忘记了一些必须的(对你而言)建议，请谅解。对了，如果你看了这篇博客之后并不能更好地进行log，我并不负责：)

事务处理

上面已经说过了

统一异常处理

在Web接口开发中，我们希望我们的接口函数运行过程中，即使抛出了异常，也能给客户端相应数据，告诉客户请求出错或者服务器出错，而不是直接宕机不去响应客户，并且我们还希望，我们抛异常之后相应数据格式要统一，此时便可以使用AOP思想做统一的异常处理

首先看我们添加一个girl的这段代码

```
@PostMapping(value = "/girl")
public Girl addGirl(@Valid Girl girl, BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        logger.error(bindingResult.getFieldError().getDefaultMessage());
        return null;
    }

    return girlRepository.save(girl);
}
```

有错误时返回null，而成功是返回girl。这边的问题是格式不统一。如果我们定义"code", "msg", 和"data"是server返回给client端的三个field，这样client端因为统一的格式，能够更容易的处理之后的操作。

```
{
    "code": -1,
    "msg": "some error message here",
    "data": null
}

{
    "code": 0
    "msg": "success",
    "data": {
        "id": 20,
        "cupSize": "B",
        "age": 25
    }
}
```

我们先来创建一个Result对象并包含这三个field

```
public class Result<T> {

    private int code;

    private String msg;

    private T data;

    public int getCode() {
        return code;
    }

    public String getMsg() {
        return msg;
    }
}
```

```

    }

    public T getData() {
        return data;
    }

    public void setCode(int code) {
        this.code = code;
    }

    public void setMsg(String msg) {
        this.msg = msg;
    }

    public void setData(T data) {
        this.data = data;
    }
}

```

然后重新refactor一下这段添加girl的代码

```

@PostMapping(value = "/girl")
public Result addGirl(@Valid Girl girl, BindingResult bindingResult) {
    Result result = new Result();
    if (bindingResult.hasErrors()) {
        result.setCode(-1);
        result.setMsg(bindingResult.getFieldError().getDefaultMessage());
        return result;
    }

    result.setCode(0);
    result.setMsg("success");
    result.setData(girlRepository.save(girl));
    return result;
}

```

我们看到每次都是做很多的set操作还是很繁琐，那就进一步refactor一下。建一个ResultUtil类

```

import com.zfu.domain.Result;

public class ResultUtil {

    public static Result generateSuccessResult(final Object object) {
        Result result = new Result();
        result.setCode(0);
        result.setMsg("success");
        result.setData(object);

        return result;
    }
}

```

```

public static Result generateErrorResult(final int errorCode, final String errorMsg) {
    Result result = new Result();
    result.setCode(errorCode);
    result.setMsg(errorMsg);

    return result;
}
}

```

有了这个类的帮助，我们再进一步改进一下添加一个girl这段代码

```

@PostMapping(value = "/girl")
public Result addGirl(@Valid Girl girl, BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        return ResultUtil.generateErrorResult(-1,
bindingResult.getFieldError().getDefaultMessage());
    }

    return ResultUtil.generateSuccessResult(girlRepository.save(girl));
}

```

这时我们有一个需求，将女生按年龄分类。假如她的年龄小于等于12，返回你还在读小学吧；假如他的年龄小于等于17，返回你还在读中学吧。为了满足这个需求，我们可以先在service类里写一个方法

```

/**
 * if age <= 12,
 *      返回你还在读小学吧
 * else if age <= 17,
 *      返回你还在读中学吧
 * else
 *      do something...
 */
public void classifyGirlsByAge(int id) throws Exception {
    Girl girl = girlRepository.findOne(id);
    int age = girl.getAge();

    if (age <= 12) {
        throw new Exception("你还在读小学吧");
    } else if (age <= 17) {
        throw new Exception("你还在读中学吧");
    }

    // do something..
}

```

然后在controller类里调用这个service的方法，来响应这个请求

```

@GetMapping(value = "girl/age/{id}")
public void classifyGirlsByAge(@PathVariable("id") int id) throws Exception {
    girlService.classifyGirlsByAge(id);
}

```

然后在controller这一层throw一个exception，会导致server挂掉。那我们怎么handle这个情况呢。先创建一个ResultEnum来枚举Exception类型的信息

```

public enum ResultEnum {
    UNKNOWN_ERROR(-1, "unknown error"),
    SUCCESS(0, "success"),
    PRIMARY SCHOOL(100, "你还在上小学吧"),
    MIDDLE SCHOOL(101, "你还在上中学吧");

    private int code;

    private String msg;

    ResultEnum(int code, String msg) {
        this.code = code;
        this.msg = msg;
    }

    public int getCode() {
        return code;
    }

    public String getMsg() {
        return msg;
    }
}

```

再写一个ExceptionHandler，带上`@ControllerAdvice`这个Annotation来handle那些runtime exception。

```

import com.zfu.domain.Result;
import com.zfu.domain.ResultEnum;
import com.zfu.exception.GirlException;
import com.zfu.utils.ResultUtil;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;

@ControllerAdvice
public class GirlExceptionHandler {

    private static final Logger logger = LoggerFactory.getLogger(GirlExceptionHandler.class);

    @ExceptionHandler(value = Exception.class)

```

```

@RequestBody
public Result handle(Exception e) {
    if (e instanceof GirlException) {
        GirlException girlException = (GirlException) e;
        return ResultUtil.generateErrorResult(girlException.getCode(),
                girlException.getMsg());
    }

    logger.error(e.getMessage());
    return ResultUtil.generateErrorResult(ResultEnum.UNKNOWN_ERROR.getCode(),
            ResultEnum.UNKNOWN_ERROR.getMsg());
}

```

五、工厂模式

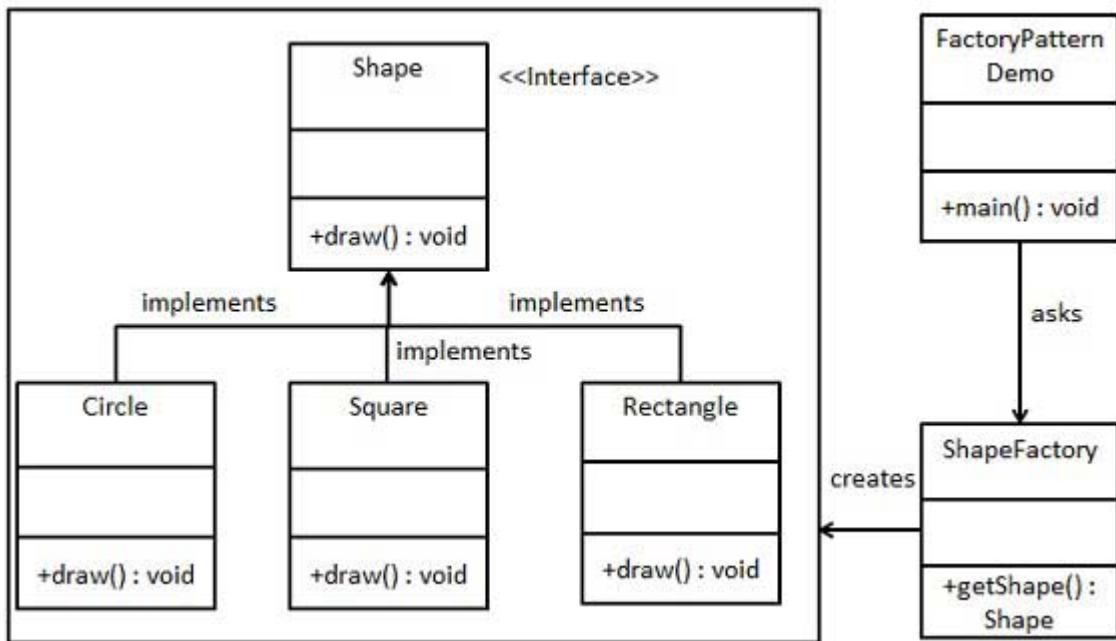
简单工厂

简单工厂模式不能说是一个设计模式，说它是一种编程习惯可能更恰当些。因为它至少不是Gof23种设计模式之一。但它在实际的编程中经常被用到，而且思想也非常简单，可以说是工厂方法模式的一个引导，所以我想有必要把它作为第一个讲一下。



一个简单的软件应用场景，一个软件系统可以提供多个外观不同的按钮（如圆形按钮、矩形按钮、菱形按钮等），这些按钮都源自同一个基类，不过在继承基类后不同的子类修改了部分属性从而使得它们可以呈现不同的外观，如果我们希望在使用这些按钮时，不需要知道这些具体按钮类的名字，只需要知道表示该按钮类的一个参数，并提供一个调用方便的方法，把该参数传入方法即可返回一个相应的按钮对象，此时，就可以使用简单工厂模式。

模式定义 简单工厂模式(Simple Factory Pattern): 又称为静态工厂方法(Static Factory Method)模式，它属于类创建型模式。在简单工厂模式中，可以根据参数的不同返回不同类的实例。简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。



步骤 1

创建一个接口:

```
Shape.java
public interface Shape {
    void draw();
}
```

步骤 2

创建实现接口的实体类。

Rectangle.java

```
public class Rectangle implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }
}
```

Square.java

```
public class Square implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Square::draw() method.");
    }
}
```

Circle.java

```
public class Circle implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Circle::draw() method.");
    }
}
```

步骤 3

创建一个工厂，生成基于给定信息的实体类的对象。

```
ShapeFactory.java
public class ShapeFactory {

    //使用 getShape 方法获取形状类型的对象
    public Shape getShape(String shapeType){
        if(shapeType == null){
            return null;
        }
        if(shapeType.equalsIgnoreCase("CIRCLE")){
            return new Circle();
        } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
            return new Rectangle();
        } else if(shapeType.equalsIgnoreCase("SQUARE")){
            return new Square();
        }
        return null;
    }
}
```

步骤 4

使用该工厂，通过传递类型信息来获取实体类的对象。

```
FactoryPatternDemo.java
public class FactoryPatternDemo {

    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();

        //获取 Circle 的对象，并调用它的 draw 方法
        Shape shape1 = shapeFactory.getShape("CIRCLE");

        //调用 Circle 的 draw 方法
        shape1.draw();

        //获取 Rectangle 的对象，并调用它的 draw 方法
        Shape shape2 = shapeFactory.getShape("RECTANGLE");

        //调用 Rectangle 的 draw 方法
        shape2.draw();

        //获取 Square 的对象，并调用它的 draw 方法
        Shape shape3 = shapeFactory.getShape("SQUARE");

        //调用 Square 的 draw 方法
        shape3.draw();
    }
}
```

工厂方法

抽象工厂

六、控制反转IOC

在传统的应用开发过程中，当我们需要一个A对象的时候，需要我们自己去new一个A对象，并且如果这个A对象在创建过程中还依赖了B对象，我们还得自己去new这个B对象，这样就导致了newA对象的程序员还必须得知道B对象的存在，增大了类之间的耦合性，违反了依赖导致原则，不利于后续的拓展

并且在传统的应用开发过程中，我们即使使用到了接口，但还是需要自己去new接口的实现类，因此并没有做到真正的解耦，并没有做到真正的面向接口编程

IoC (Inversion of Control, 控制反转)

思想就是利用工厂模式，设置一个对象的容器，将对象的创建，依赖的管理，生命周期的管理都交给容器来完成，当我们需要一个对象A的时候，只需要使用类似于 A a = factory.get("A")的方式，从容器中拿A对象就可以了，至于A对象在创建过程中还需要什么对象我们完全不用去考虑

使用接口的时候，我们只需要将接口的实现类放入IOC容器中，然后从容器中拿接口的实现就可以了，可以实现真正的解耦，真正的面向接口编程。

七、观察者模式

定义：当对象间存在一对多关系时，则使用观察者模式（Observer Pattern）。比如，当一个对象被修改时，则会自动通知它的依赖对象。观察者模式属于行为型模式。

意图：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

主要解决：一个对象状态改变给其他对象通知的问题，而且要考虑到易用和低耦合，保证高度的协作。

何时使用：一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知，进行广播通知。

```
/**
 * 被观察者，也就是微信公众号服务
 * 实现了Observable接口，对Observable接口的三个方法进行了具体实现
 * @author jstao
 *
 */
public class WechatServer implements Observable {

    //注意到这个List集合的泛型参数为Observer接口，设计原则：面向接口编程而不是面向实现编程
    private List<Observer> list;
    private String message;

    public WechatServer() {
        list = new ArrayList<Observer>();
    }

    @Override
    public void registerObserver(Observer o) {

        list.add(o);
    }
}
```

```
}

@Override
public void removeObserver(Observer o) {
    if(!list.isEmpty())
        list.remove(o);
}

//遍历
@Override
public void notifyObserver() {
    for(int i = 0; i < list.size(); i++) {
        Observer oserver = list.get(i);
        oserver.update(message);
    }
}

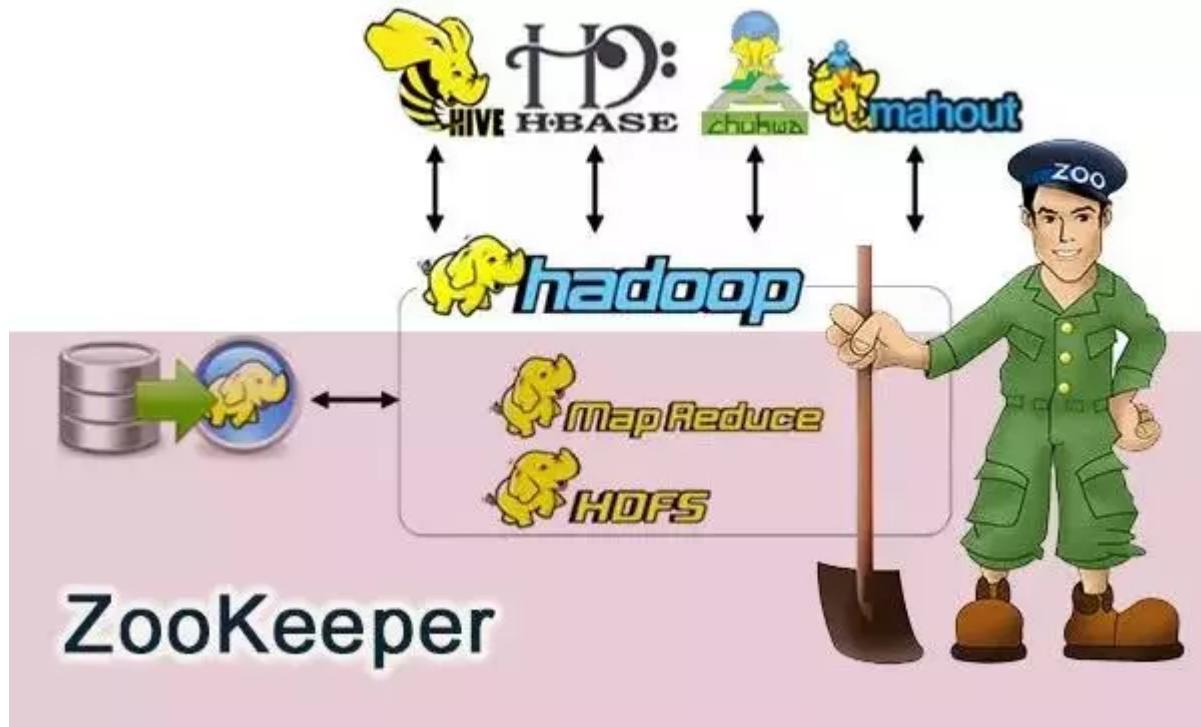
public void setInfomation(String s) {
    this.message = s;
    System.out.println("微信服务更新消息: " + s);
    //消息更新，通知所有观察者
    notifyObserver();
}

}
```

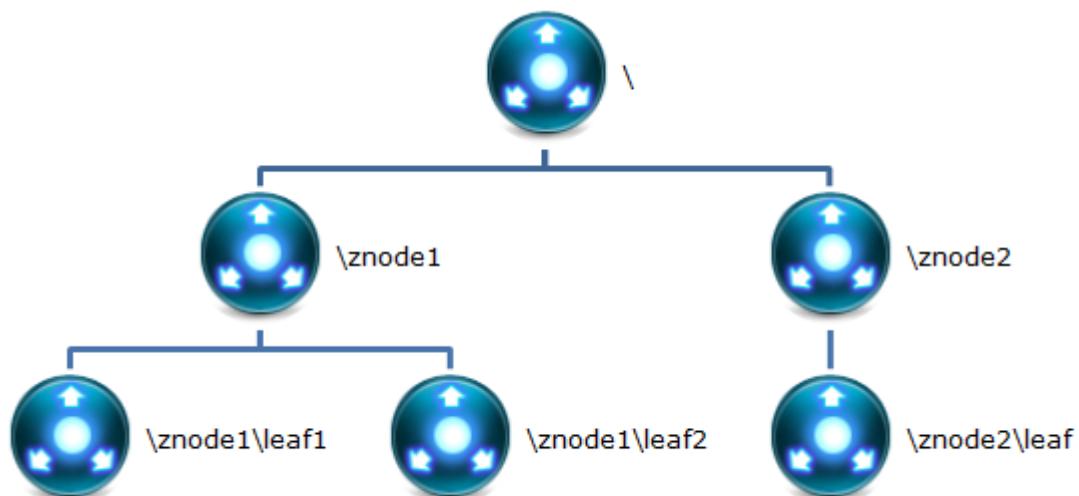
八、Zookeeper

ZK简述

Zookeeper从设计模式角度来理解：是一个基于观察者模式设计的分布式服务管理框架，它负责存储和管理大家都关心的数据，然后接受观察者的注册，一旦这些数据的状态发生变化，Zookeeper就将负责通知已经在Zookeeper上注册的那些观察者做出相应的反应，从而实现集群中类似Master/Slave管理模式



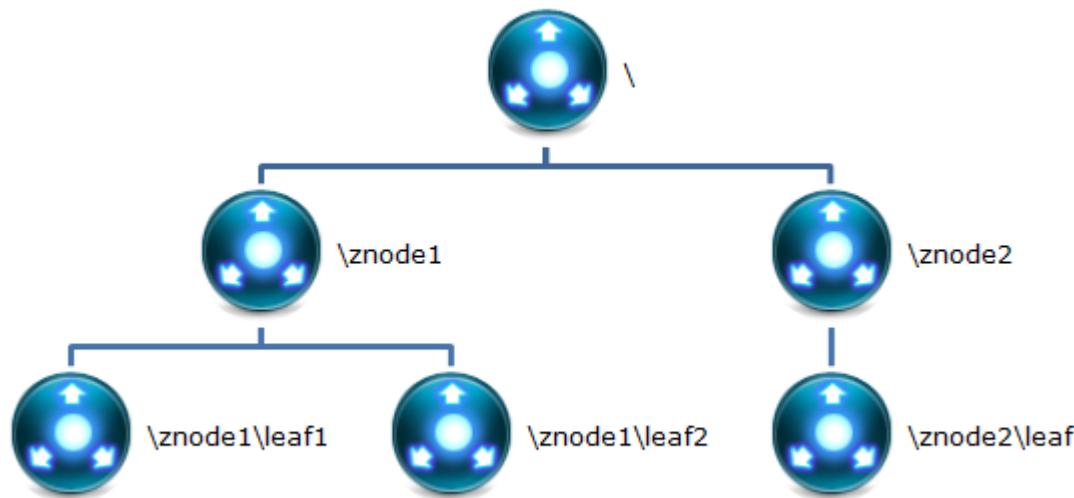
存储结构



zookeeper中的数据是按照“树”结构进行存储的。而且znode节点还分为4中不同的类型。

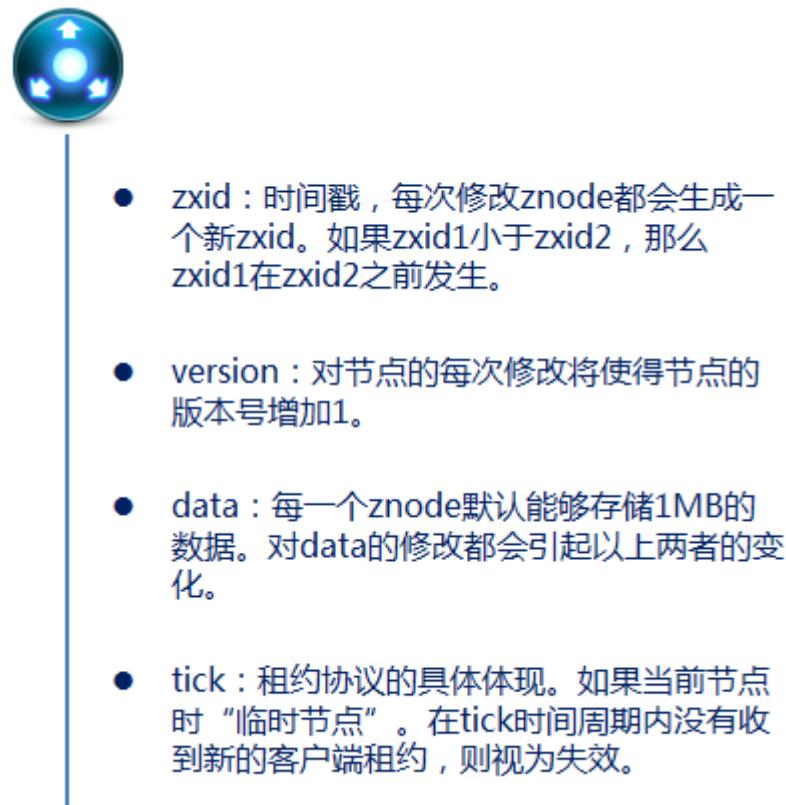
znode

根据本小结第一部分的描述，很显然zookeeper集群自身维护了一套数据结构。这个存储结构是一个树形结构，其上的每一个节点，我们称之为“znode”。如下如所示：



- 每一个znode默认能够存储1MB的数据（对于记录状态性质的数据来说，够了）
- 可以使用zkCli命令，登录到zookeeper上，并通过ls、create、delete、sync等命令操作这些znode节点
- znode除了名称、数据以外，还有一套属性：zxid。这套zxid与时间戳对应，记录zxid不同的状态（后续我们将用到）

那么每个znode结构又是什么样的呢？如下图所示：



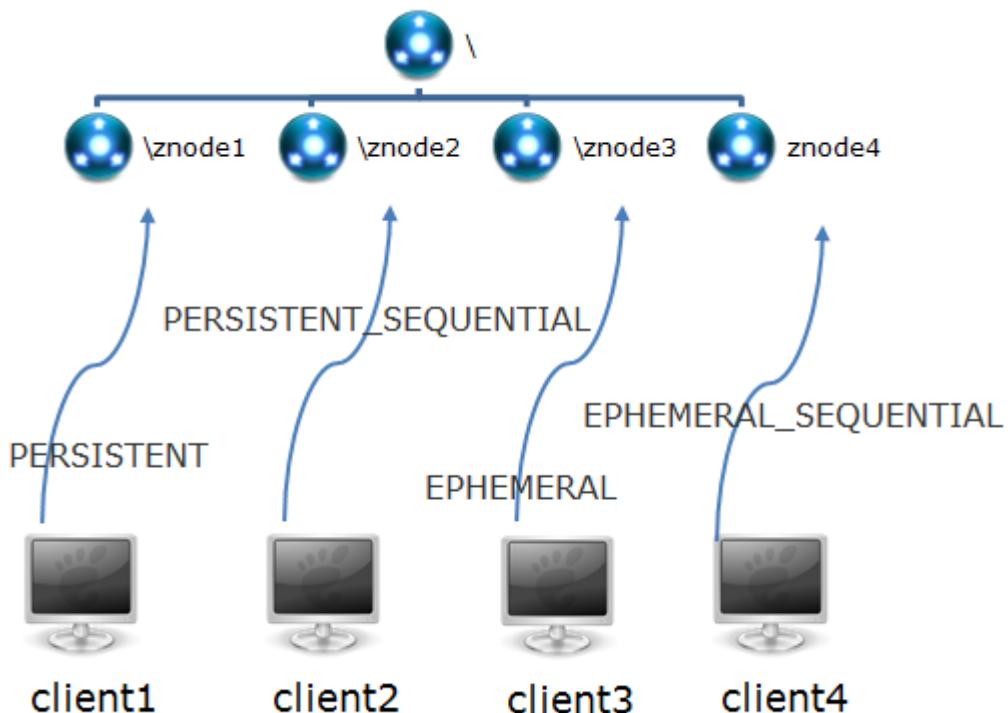
此外，znode还有操作权限。如果我们把以上几类属性细化，又可以得到以下属性的细节：

- czxid：创建节点的事务的zxid
- mzxid：对znode最近修改的zxid
- ctime：以距离时间原点(epoch)的毫秒数表示的znode创建时间
- mtime：以距离时间原点(epoch)的毫秒数表示的znode最近修改时间

- version: znode数据的修改次数
- cversion: znode子节点修改次数
- aversion: znode的ACL修改次数
- ephemeralOwner: 如果znode是临时节点，则指示节点所有者的会话ID；如果不是临时节点，则为零。
- dataLength: znode数据长度。
- numChildren: znode子节点个数。

znode中的存在类型

我们知道了zookeeper内部维护了一套数据结构：由znode构成的集合，znode的集合又是一个树形结构。每一个znode又有很多属性进行描述。并且znode的存在性还分为四类，如下如所示：



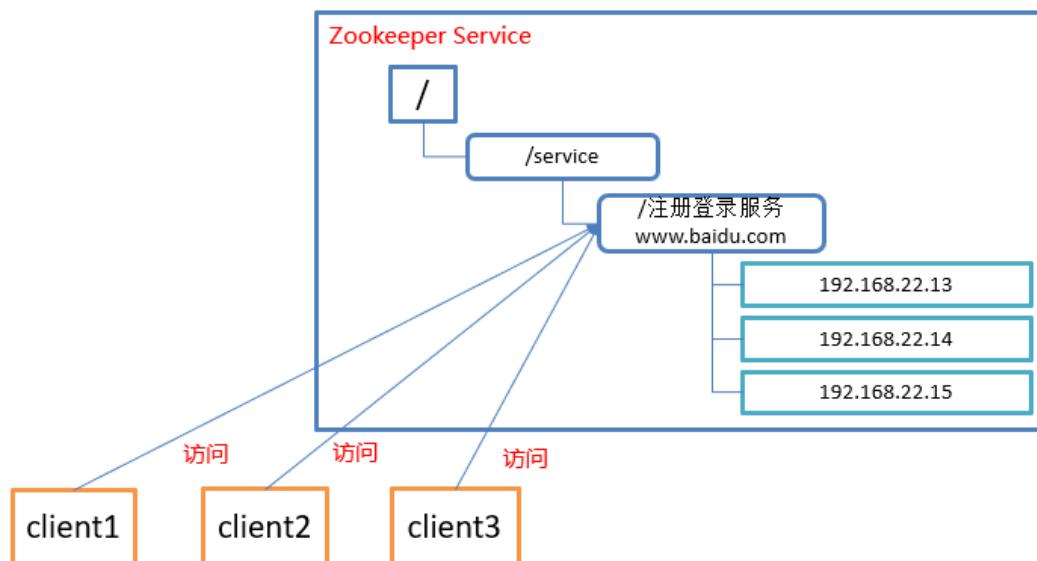
znode是由客户端创建的，它和创建它的客户端的内在联系，决定了它的存在性：

- PERSISTENT-持久化节点：创建这个节点的客户端在与zookeeper服务的连接断开后，这个节点也不会被删除（除非您使用API强制删除）。
- PERSISTENT_SEQUENTIAL-持久化顺序编号节点：当客户端请求创建这个节点A后，zookeeper会根据parent-znode的xid状态，为这个A节点编写一个全目录唯一的编号（这个编号只会一直增长）。当客户端与zookeeper服务的连接断开后，这个节点也不会被删除。
- EPHEMERAL-临时目录节点：创建这个节点的客户端在与zookeeper服务的连接断开后，这个节点（还有涉及到的子节点）就会被删除。
- EPHEMERAL_SEQUENTIAL-临时顺序编号目录节点：当客户端请求创建这个节点A后，zookeeper会根据parent-znode的xid状态，为这个A节点编写一个全目录唯一的编号（这个编号只会一直增长）。当创建这个节点的客户端与zookeeper服务的连接断开后，这个节点被删除。
- 另外，无论是EPHEMERAL还是EPHEMERAL_SEQUENTIAL节点类型，在zookeeper的client异常终止后，节点也会被删除。

应用场景

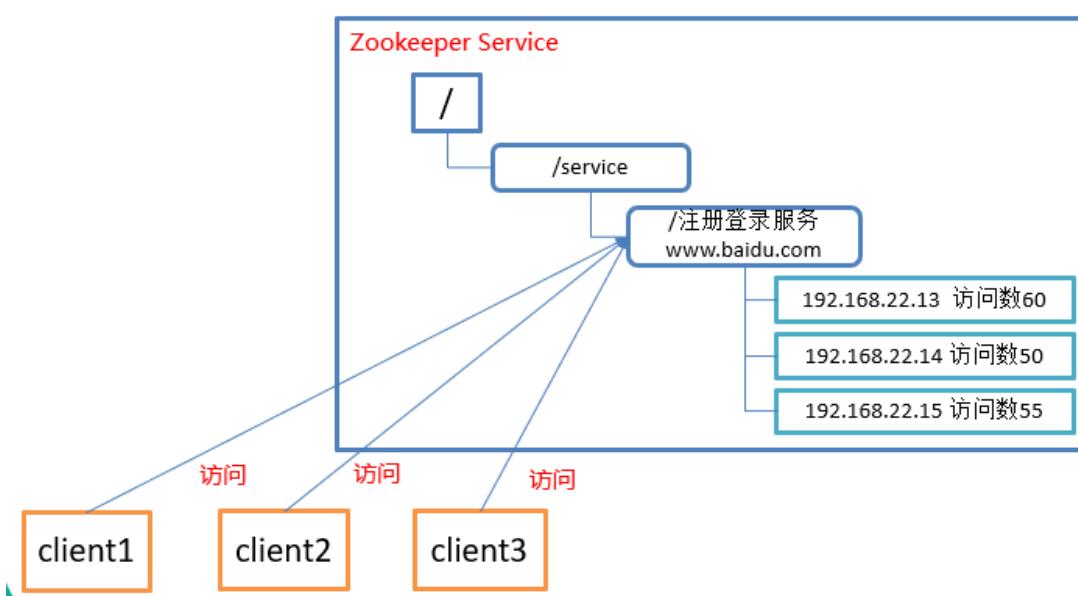
统一命名服务

统一命名服务

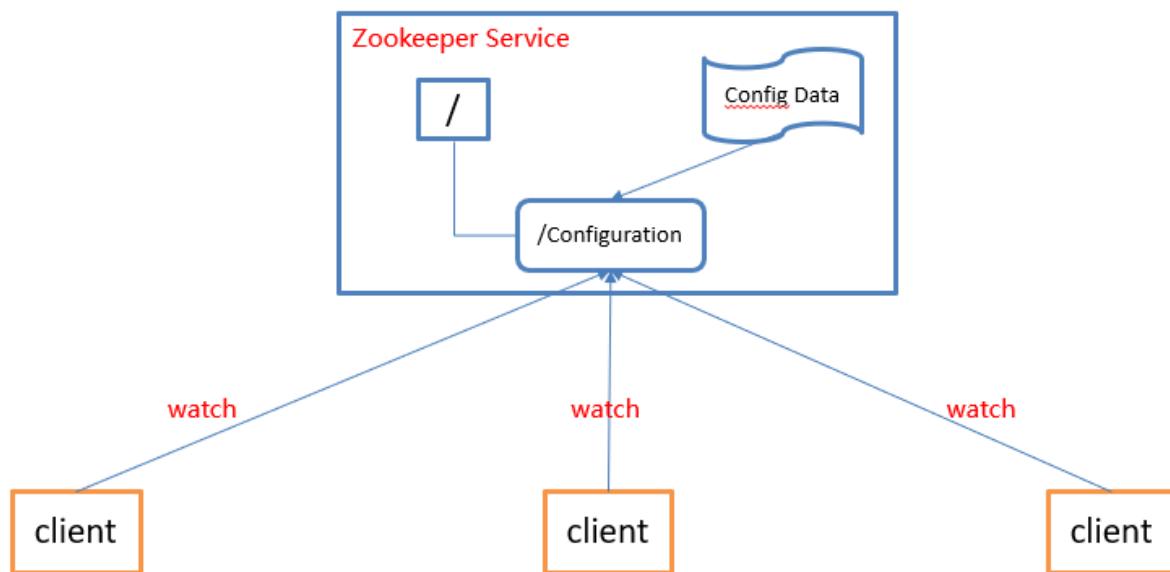
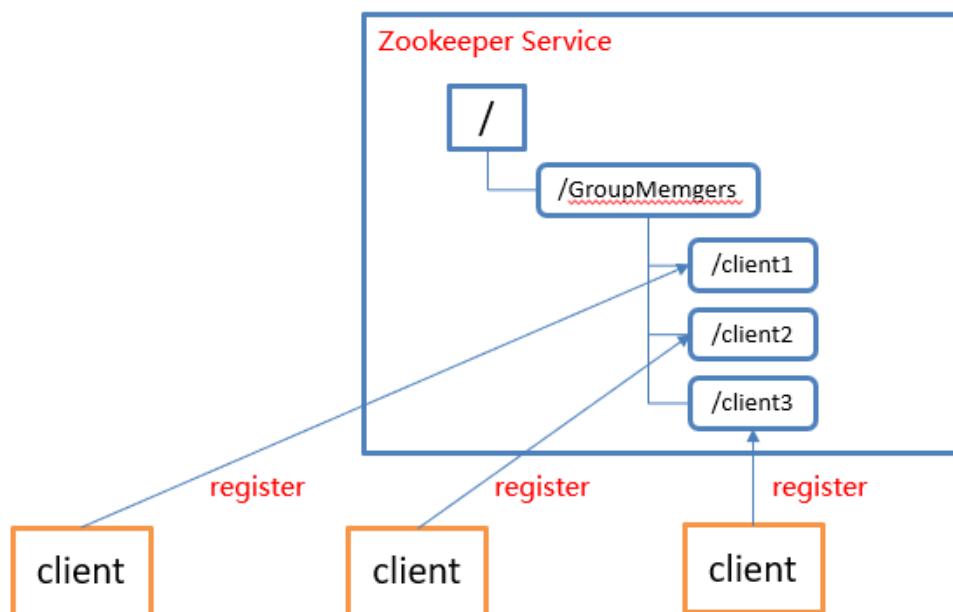


负载均衡

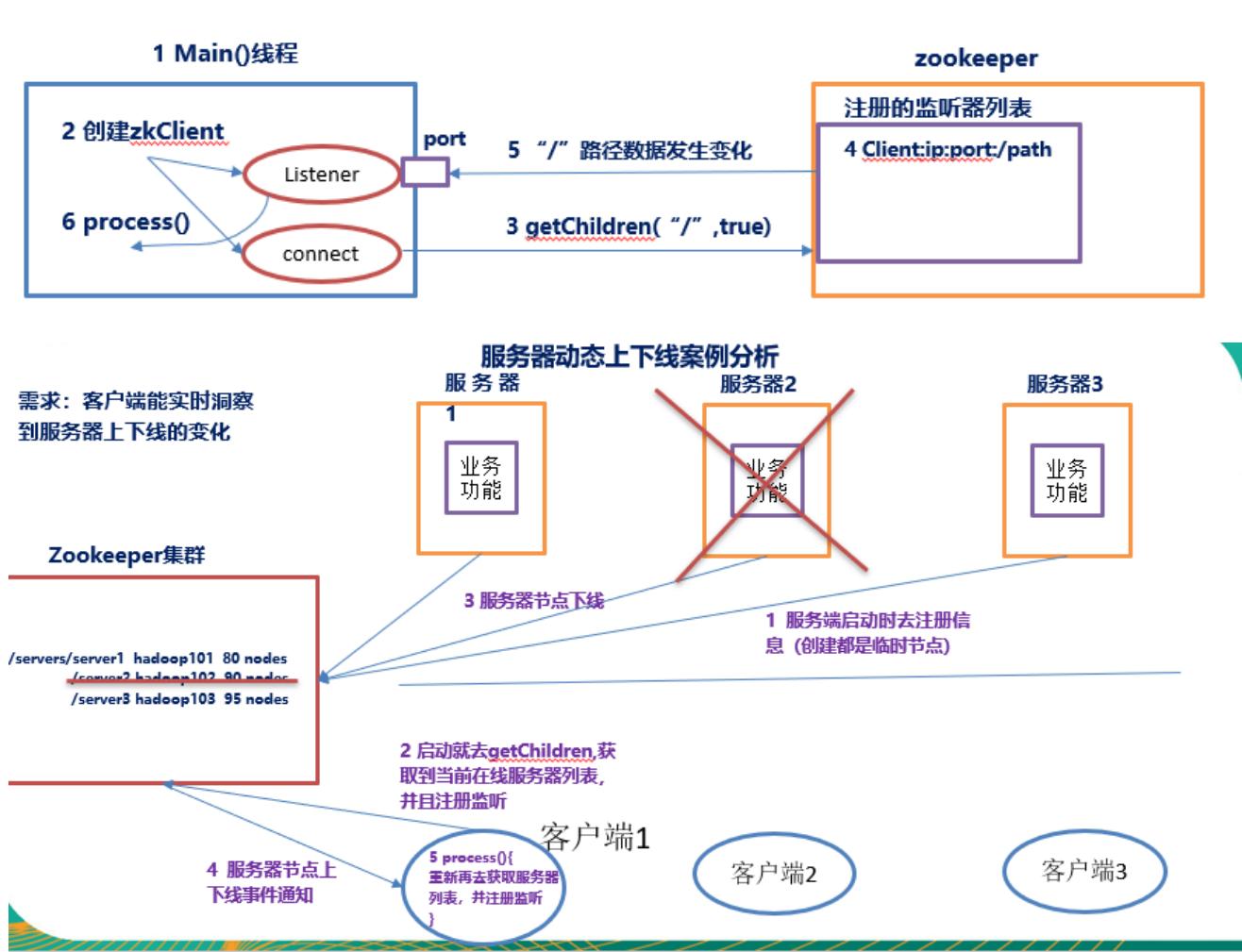
软负载均衡



统一配置管理

配置管理**集群管理****集群管理****服务器动态上下线**

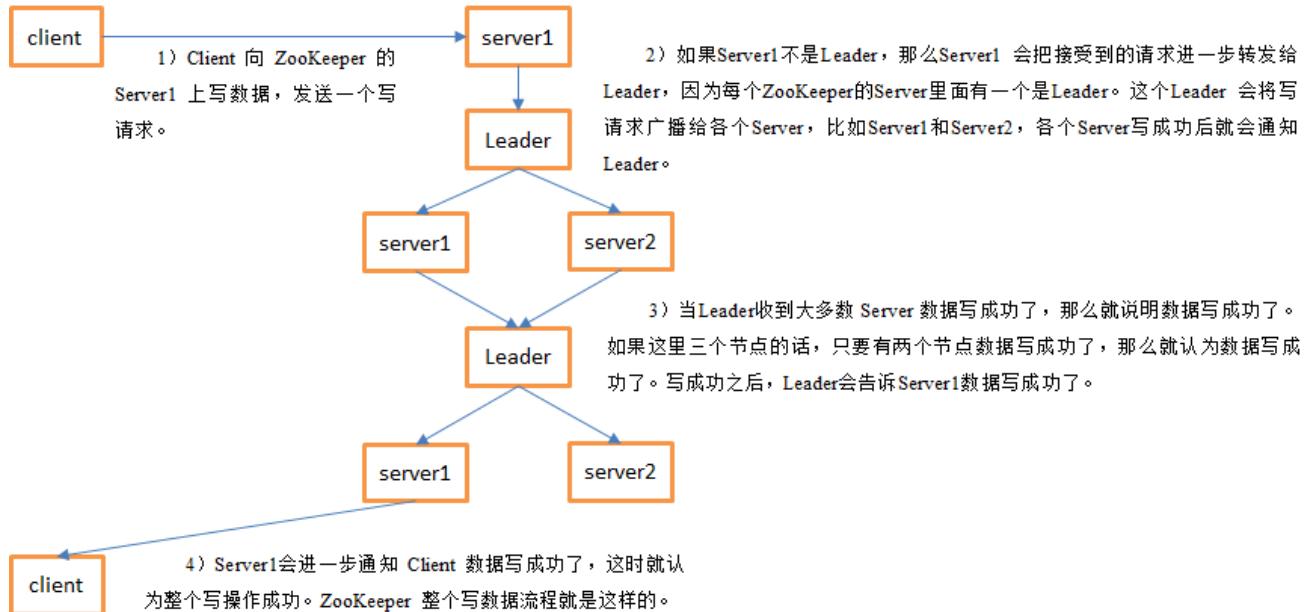
监听器原理



写数据流程

Zookeeper提供的是弱一致性，CAP限制，读的数据可能不是最新的，如果想读到最新的数据，应该手动调用sync方法从Leader同步数据

写数据流程

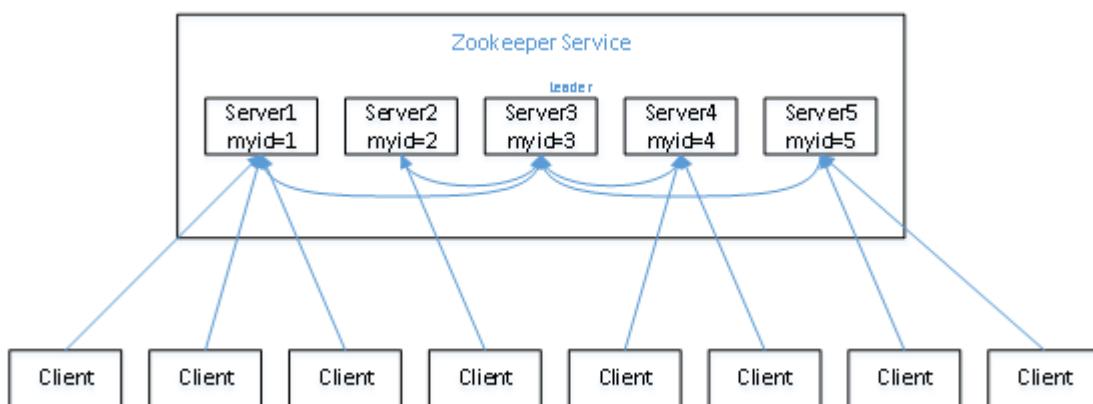


Leader选举

ZK的Leader负责同步数据，发起选举

- 1) 半数机制：集群中半数以上机器存活，集群可用。所以zookeeper适合装在奇数台机器上。
- 2) Zookeeper虽然在配置文件中并没有指定master和slave。但是，zookeeper工作时，是有一个节点为leader，其他则为follower，Leader是通过内部的选举机制临时产生的
- 3) 以一个简单的例子来说明整个选举的过程。

假设有五台服务器组成的zookeeper集群，它们的id从1-5，同时它们都是最新启动的，也就是没有历史数据，在存放数据量这一点上，都是一样的。假设这些服务器依序启动，来看看会发生什么。



(1) 服务器1启动，此时只有它一台服务器启动了，它发出去的报没有任何响应，所以它的选举状态一直是 LOOKING 状态。

(2) 服务器2启动，它与最开始启动的服务器1进行通信，互相交换自己的选举结果，由于两者都没有历史数据，所以id值较大的服务器2胜出，但是由于没有达到超过半数以上的服务器都同意选举它(这个例子中的半数以上是3)，所以服务器1、2还是继续保持LOOKING状态。

(3) 服务器3启动，根据前面的理论分析，服务器3成为服务器1、2、3中的老大，而与上面不同的是，此时有三台服务器选举了它，所以它成为了这次选举的leader。

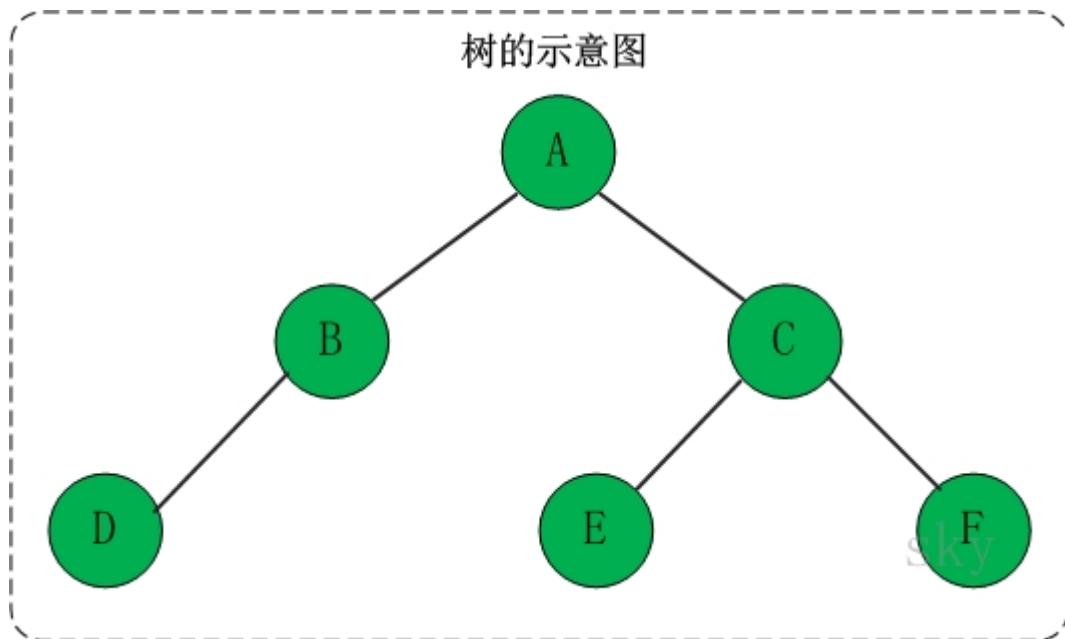
(4) 服务器4启动，根据前面的分析，理论上服务器4应该是服务器1、2、3、4中最大的，但是由于前面已经有半数以上的服务器选举了服务器3，所以它只能接收当小弟的命了。

(5) 服务器5启动，同4一样当小弟。

数据结构与算法

一、 树

树是一种数据结构，它是由 n ($n \geq 1$) 个有限节点组成一个具有层次关系的集合。

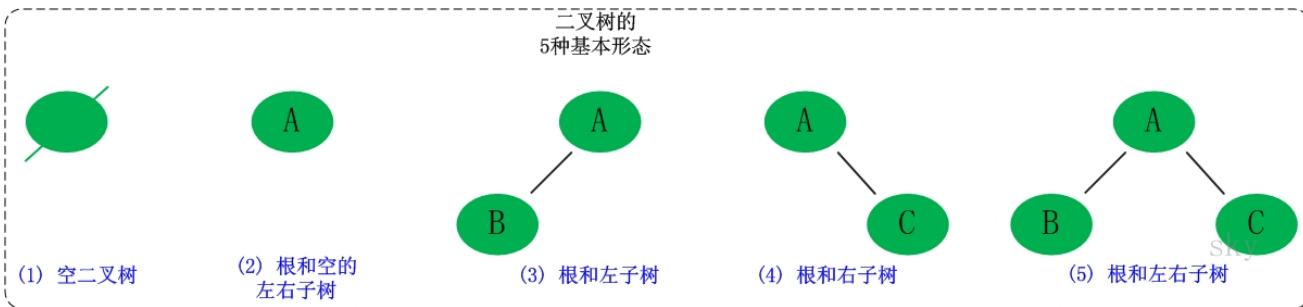


把它叫做“树”是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。它具有以下的特点：(01) 每个节点有零个或多个子节点；(02) 没有父节点的节点称为根节点；(03) 每一个非根节点有且只有一个父节点；(04) 除了根节点外，每个子节点可以分为多个不相交的子树。

二、 BST树

二叉树的定义

二叉树是每个节点最多有两个子树的树结构。它有五种基本形态：二叉树可以是空集；根可以有空的左子树或右子树；或者左、右子树皆为空。



2. 二叉树的性质

二叉树有以下几个性质：TODO(上标和下标) **性质1：**二叉树第*i*层上的结点数目最多为 2^{i-1} (*i*≥1)。 **性质2：**深度为k的二叉树至多有 2^k-1 个结点(*k*≥1)。 **性质3：**包含n个结点的二叉树的高度至少为 $\log_2(n+1)$ 。 **性质4：**在任意一棵二叉树中，若终端结点的个数为n0，度为2的结点数为n2，则 $n_0=n_2+1$ 。

2.1 性质1：二叉树第*i*层上的结点数目最多为 2^{i-1} (*i*≥1)

证明：下面用“数学归纳法”进行证明。(01) 当*i*=1时，第1层的节点数目为 $2^{i-1}=2^0=1$ 。因为第1层上只有一个根结点，所以命题成立。(02) 假设当*i*>1，第*i*层的节点数目为 2^{i-1} 。这个是根据(01)推断出来的！下面根据这个假设，推断出“第(i+1)层的节点数目为 2^i ”即可。由于二叉树的每个结点至多有两个孩子，故“第(i+1)层上的结点数目”最多是“第*i*层的结点数目的2倍”。即，第(i+1)层上的结点数目最大值= $2 \times 2^{i-1} = 2^i$ 。故假设成立，原命题得证！

2.2 性质2：深度为k的二叉树至多有 2^k-1 个结点(*k*≥1)

证明：在具有相同深度的二叉树中，当每一层都含有最大结点数时，其树中结点数最多。利用“性质1”可知，深度为k的二叉树的结点数至多为： $2^0+2^1+\dots+2^{k-1}=2^k-1$ 故原命题得证！

2.3 性质3：包含n个结点的二叉树的高度至少为 $\log_2(n+1)$

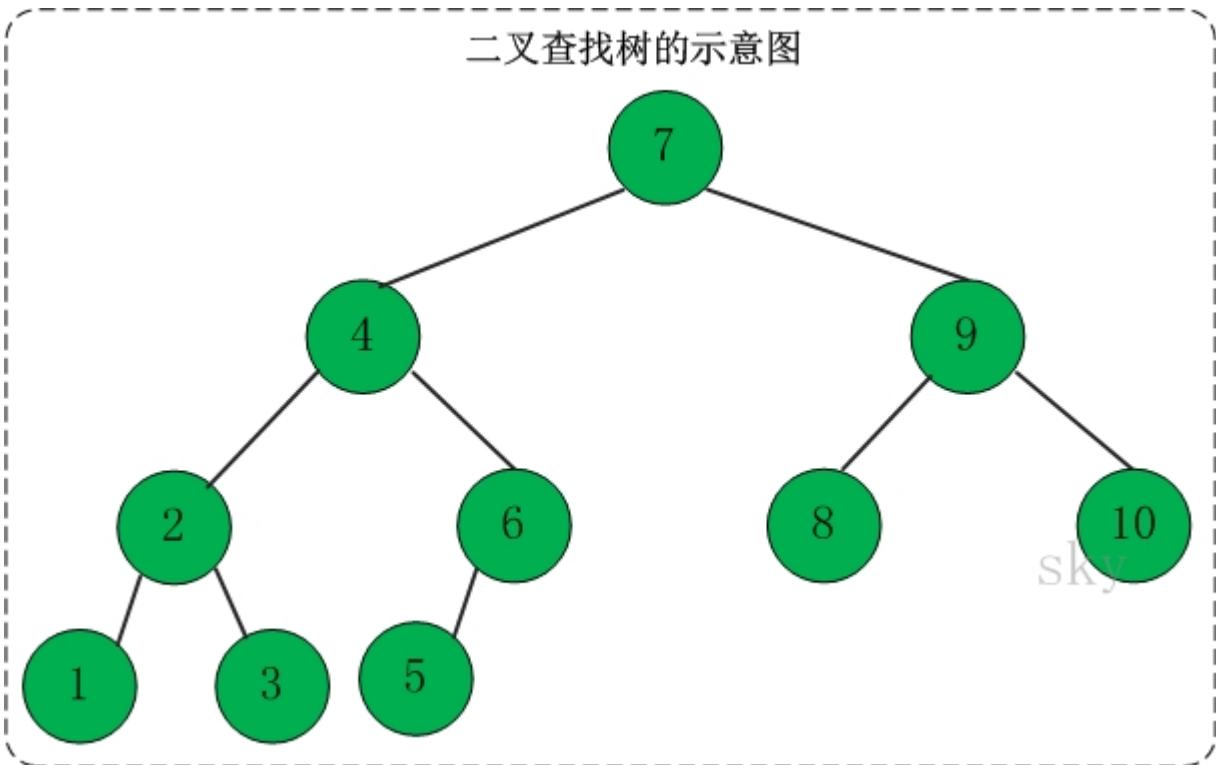
证明：根据“性质2”可知，高度为h的二叉树最多有 2^h-1 个结点。反之，对于包含n个节点的二叉树的高度至少为 $\log_2(n+1)$ 。

2.4 性质4：在任意一棵二叉树中，若终端结点的个数为n0，度为2的结点数为n2，则 $n_0=n_2+1$

证明：因为二叉树中所有结点的度数均不大于2，所以结点总数(记为n)=“0度结点数(n0)” + “1度结点数(n1)” + “2度结点数(n2)”。由此，得到等式一。(等式一) $n=n_0+n_1+n_2$ 另一方面，0度结点没有孩子，1度结点有一个孩子，2度结点有两个孩子，故二叉树中孩子结点总数是： n_1+2n_2 。此外，只有根不是任何结点的孩子。故二叉树中的结点总数又可表示为等式二。(等式二) $n=n_1+2n_2+1$ 由(等式一)和(等式二)计算得到： $n_0=n_2+1$ 。原命题得证！

三、BST树

定义：二叉查找树(Binary Search Tree)，又被称为二叉搜索树。设x为二叉查找树中的一个结点，x节点包含关键字key，节点x的key值记为key[x]。如果y是x的左子树中的一个结点，则 $key[y] \leq key[x]$ ；如果y是x的右子树的一个结点，则 $key[y] \geq key[x]$ 。



在二叉查找树中：(01) 若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；(02) 任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；(03) 任意节点的左、右子树也分别为二叉查找树。(04) 没有键值相等的节点 (no duplicate nodes)。

四、AVL树

AVL树是高度平衡的二叉搜索树，按照二叉搜索树（Binary Search Tree）的性质，AVL首先要满足：

- 若它的左子树不为空，则左子树上所有结点的值均小于它的根结点的值；
- 若它的右子树不为空，则右子树上所有结点的值均大于它的根结点的值；
- 它的左、右子树也分别为二叉搜索树。

AVL树的性质：左子树和右子树的高度之差的绝对值不超过1。树中的每个左子树和右子树都是AVL树。每个节点都有一个平衡因子(balance factor--bf)，任一节点的平衡因子是-1,0,1之一(每个节点的平衡因子bf 等于右子树的高度减去左子树的高度)

当插入或者删除节点之后，若AVL树的条件被破坏，则需要进行旋转操作来调整数据的结构以恢复AVL条件

旋转至少涉及三层节点，所以至少要向上回溯一层，才会发现非法的平衡因子并进行旋转向上回溯校验时，需要进行旋转的几种情况：

1. 当前节点的父节点的平衡因子等于2时，说明父节点的右树比左树高：这时如果当前节点的平衡因子等于1，那么当前节点的右树比左树高，形如“\”，需要进行左旋；如果当前节点的平衡因子等于-1，那么当前节点的右树比左树低，形如“>”，需要进行右左双旋！
2. 当前节点的父节点的平衡因子等于-2时，说明父节点的右树比左树低：这时如果当前节点的平衡因子等于-1，那么当前节点的右树比左树低，形如“/”，需要进行右旋；如果当前节点的平衡因子等于1，那么当前节点的右树比左树高，形如“<”，需要进行左右双旋！

五、红黑树

红黑树（Red Black Tree）是一种自平衡二叉查找树，满足以下条件：

1. 节点是红色或黑色。
2. 根节点是黑色。
3. 每个叶子节点都是黑色的空节点 (NIL节点)。
4. 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)
5. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

这些特性使得红黑树中从根节点到叶子节点的最长路径不会超过最短路径的两倍

红黑树通过变色、左旋和右旋来保持平衡，**任何不平衡都会在三次旋转之内解决**

首先红黑树是不符合AVL树的平衡条件的，即每个节点的左子树和右子树的高度最多差1的二叉查找树。但是提出了为节点增加颜色，红黑是用非严格的平衡来换取增删节点时候旋转次数的降低，任何不平衡都会在三次旋转之内解决，而AVL是严格平衡树，因此在增加或者删除节点的时候，根据不同情况，旋转的次数比红黑树要多。所以红黑树的插入效率更高！！！

六、B-树

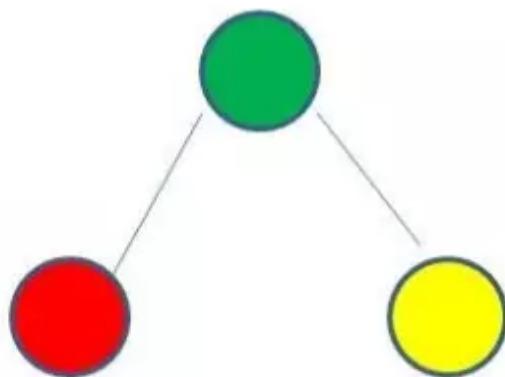
B-树就是B树，千万不要读B减树！！！！

从算法逻辑上来讲，二叉查找树的查找次数和比较次数都是最小的。但是，我们不得不考虑一个现实的问题：磁盘IO

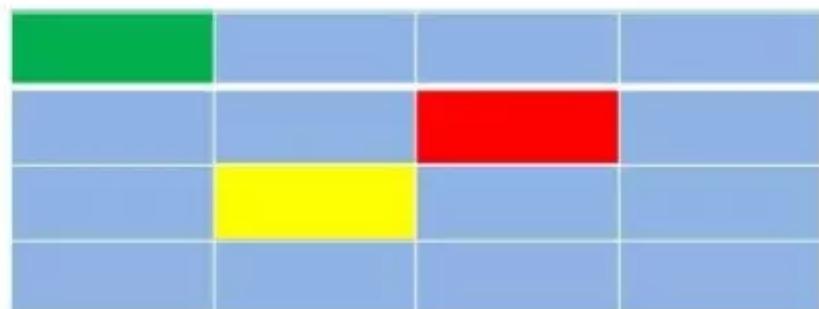
数据库索引是存储在磁盘上的，当数据量比较大的时候，索引的大小可能有几个G甚至更多

当我们利用索引查询的时候，能把整个索引文件全部加载到内存吗？显然不可能，能做的只有逐一加载每一个磁盘页，这里的磁盘页对应着索引树的节点

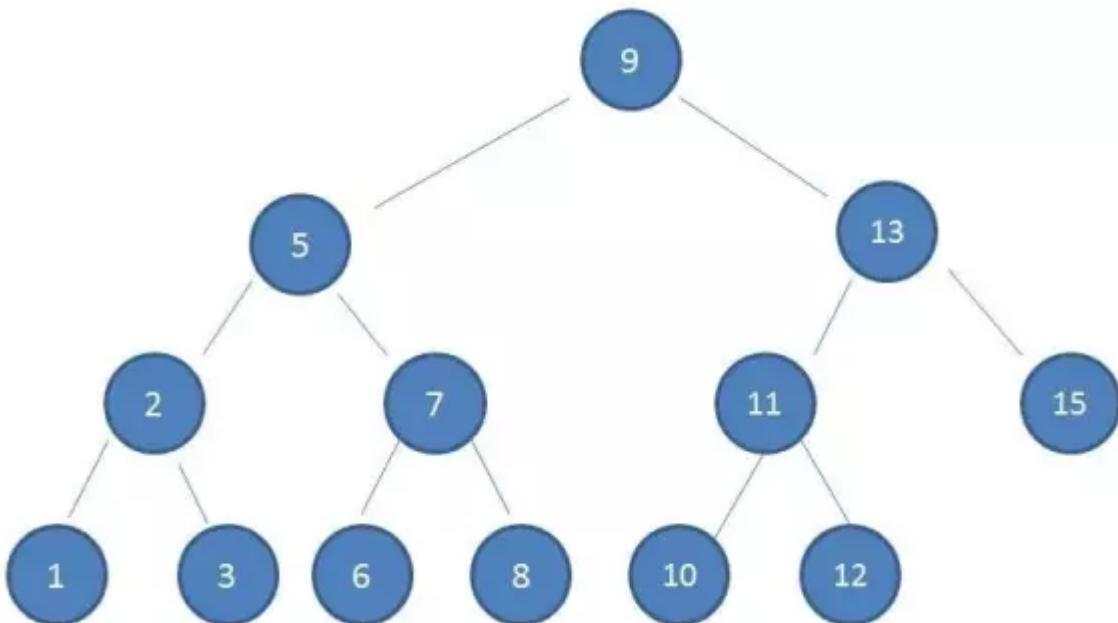
索引树：



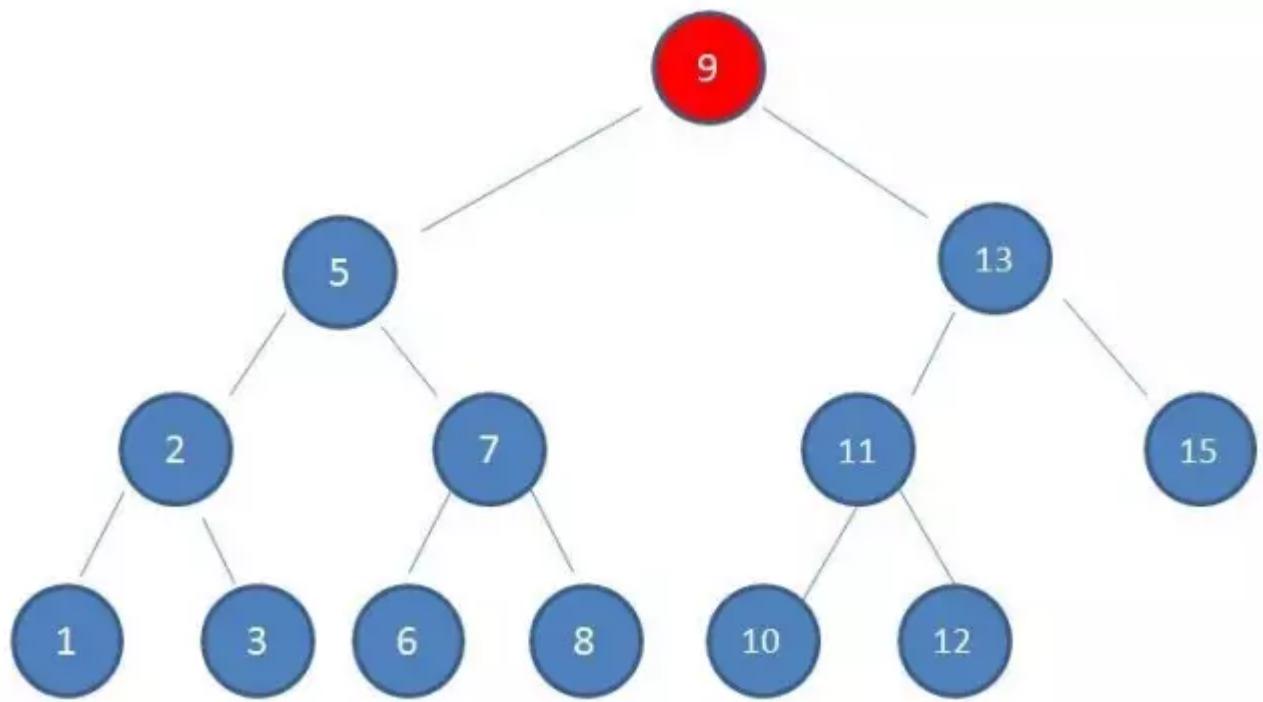
磁盘页：



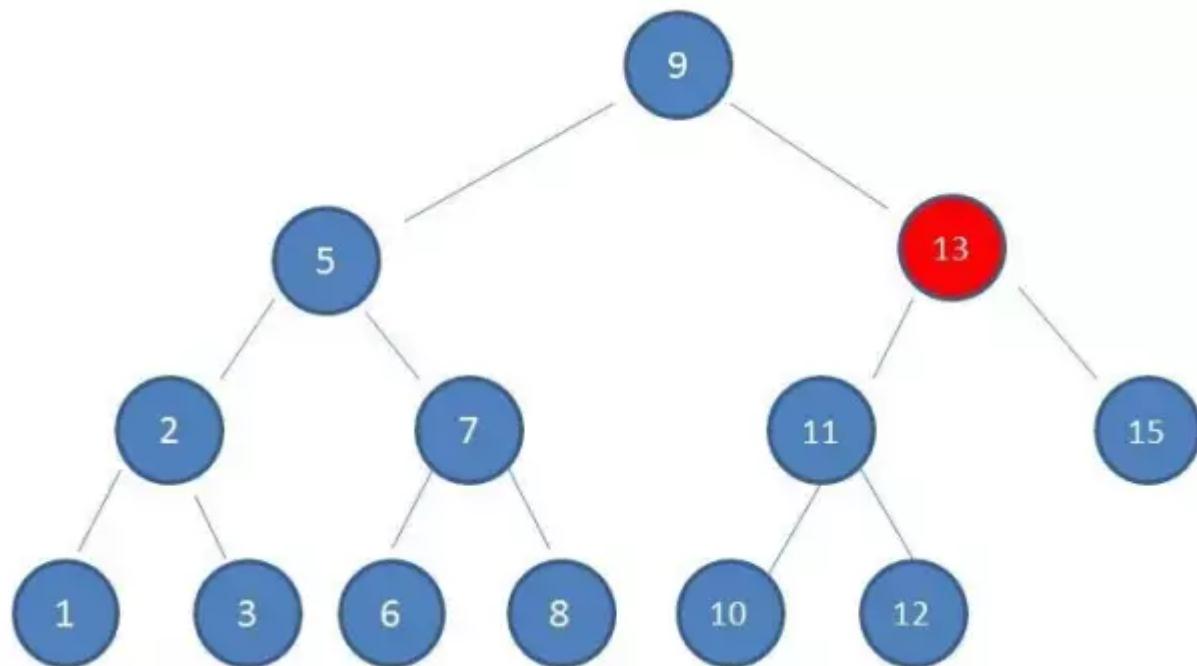
二叉查找树的结构：



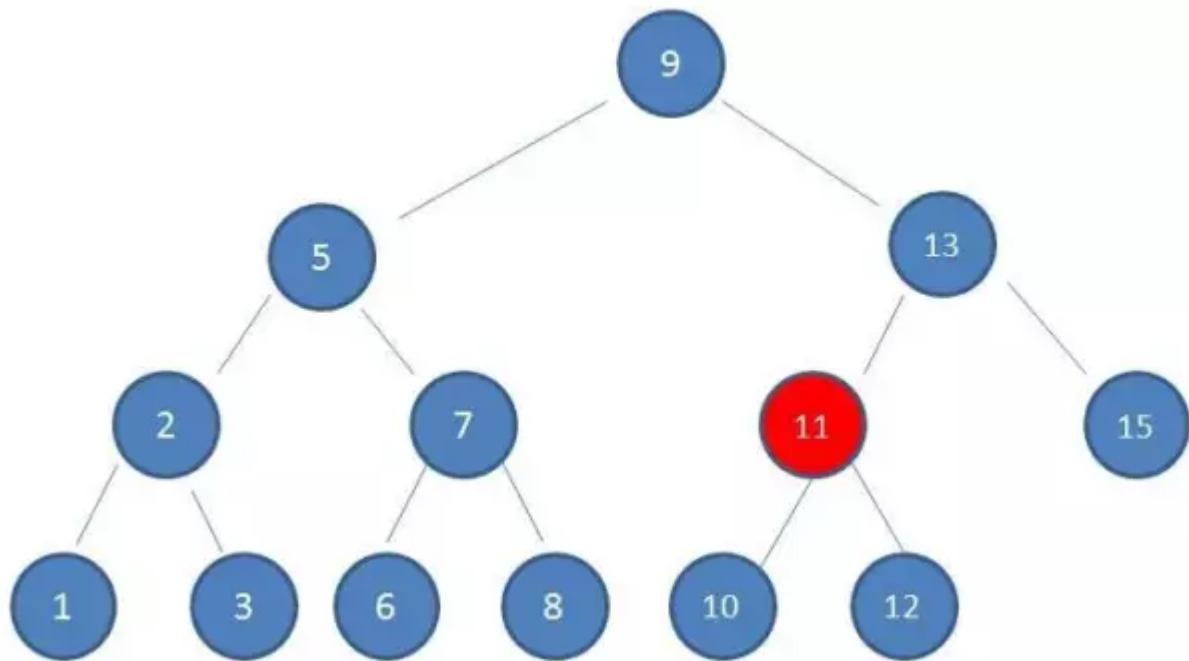
第1次磁盘IO：



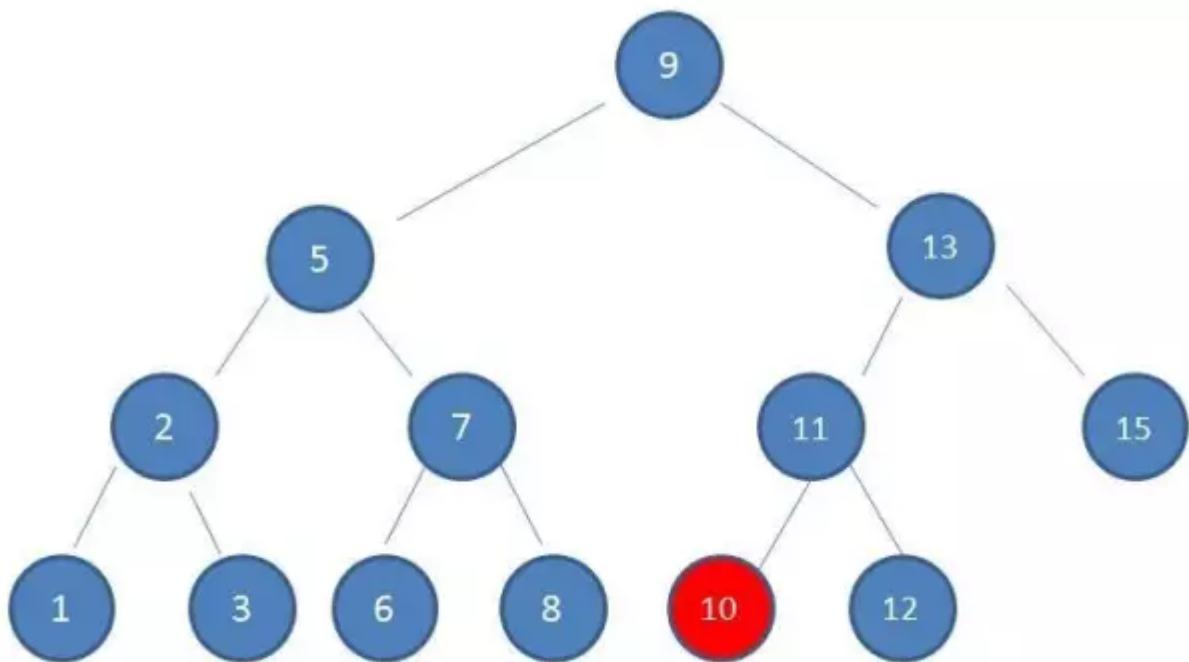
第2次磁盘IO:



第3次磁盘IO:



第4次磁盘IO:



在使用二叉查找树查询过程中，我们发现在最坏的情况下，磁盘IO次数等于索引树的高度

因此，为了减少磁盘IO次数，我们就需要把原本“瘦高”的树结构变得“矮胖”些。这就是B-树的特征之一

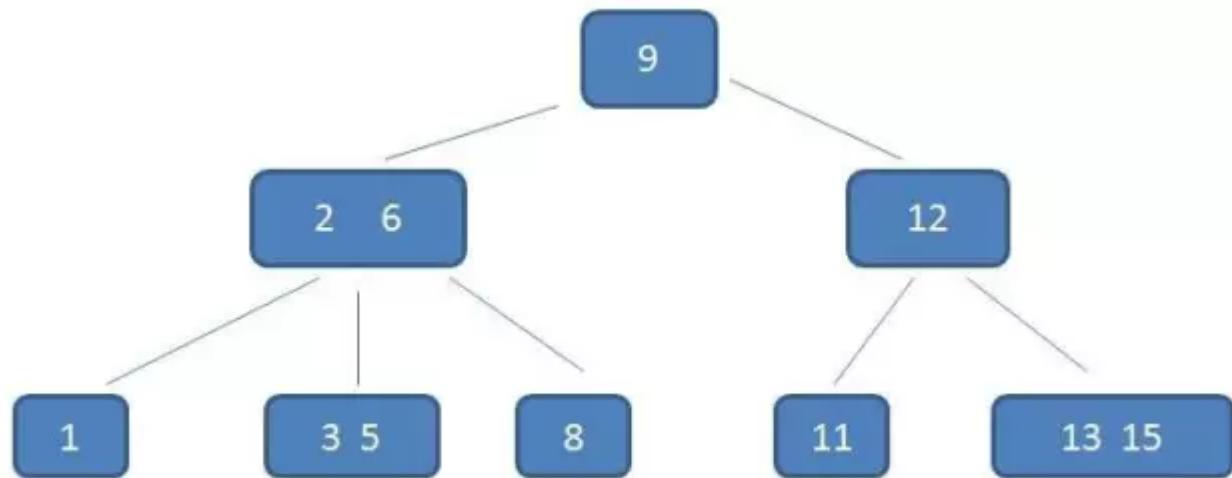
B树是一种多路平衡查找树，它的每一个节点最多包含K个孩子，K被称为B树的阶，K的大小取决于磁盘页的大小

一个m阶的B树具有如下几个特征：

1. 根结点至少有两个子女。
2. 每个中间节点都包含 $k-1$ 个元素和 k 个孩子，其中 $m/2 \leq k \leq m$
3. 每一个叶子节点都包含 $k-1$ 个元素，其中 $m/2 \leq k \leq m$

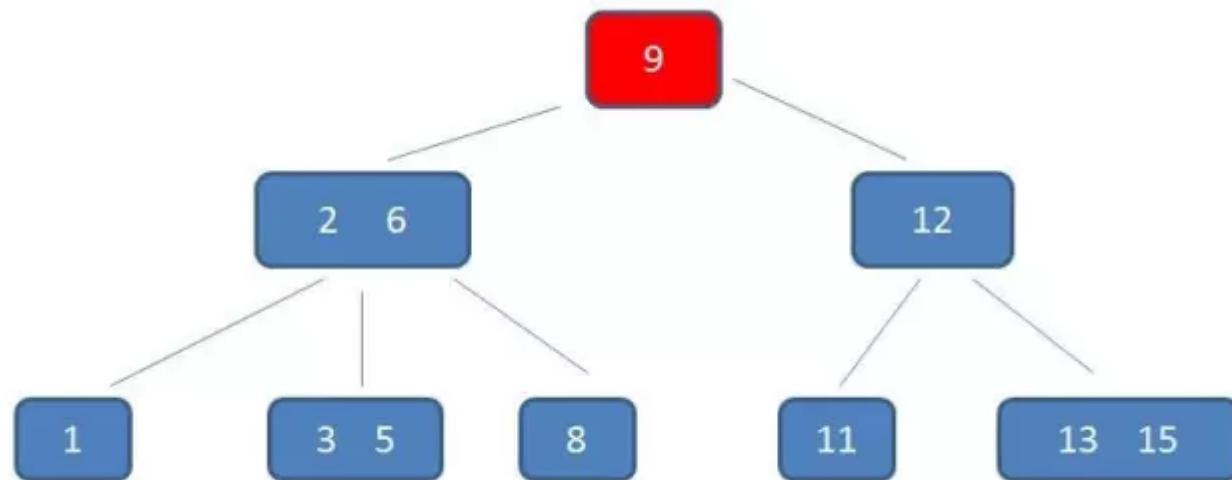
4. 所有的叶子结点都位于同一层。

5. 每个节点中的元素从小到大排列，节点当中k-1个元素正好是k个孩子包含的元素的值域分划。

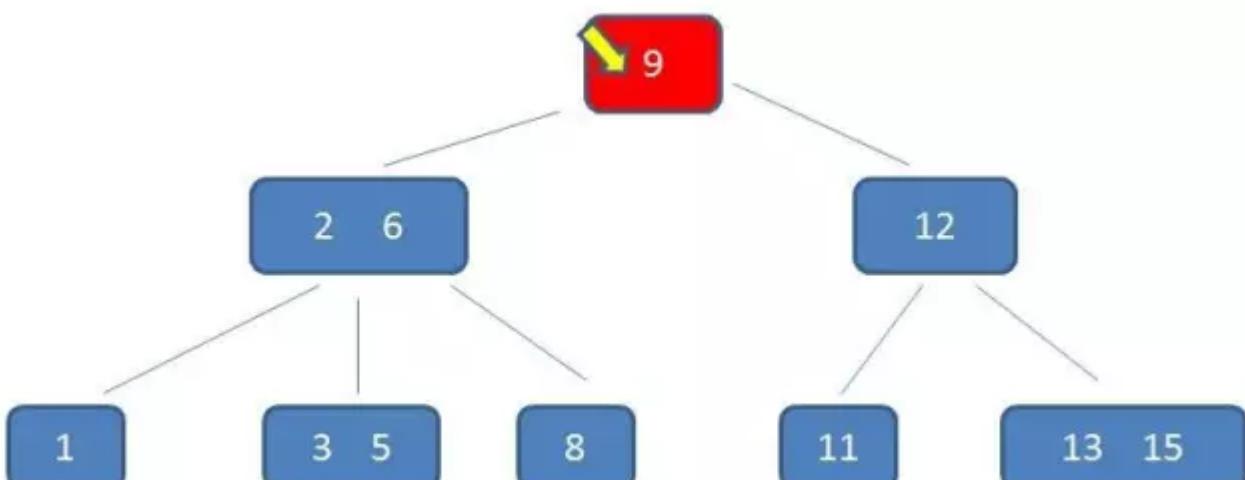


在这棵B树中，假设我们要查询的关键字为6，查询过程如下：

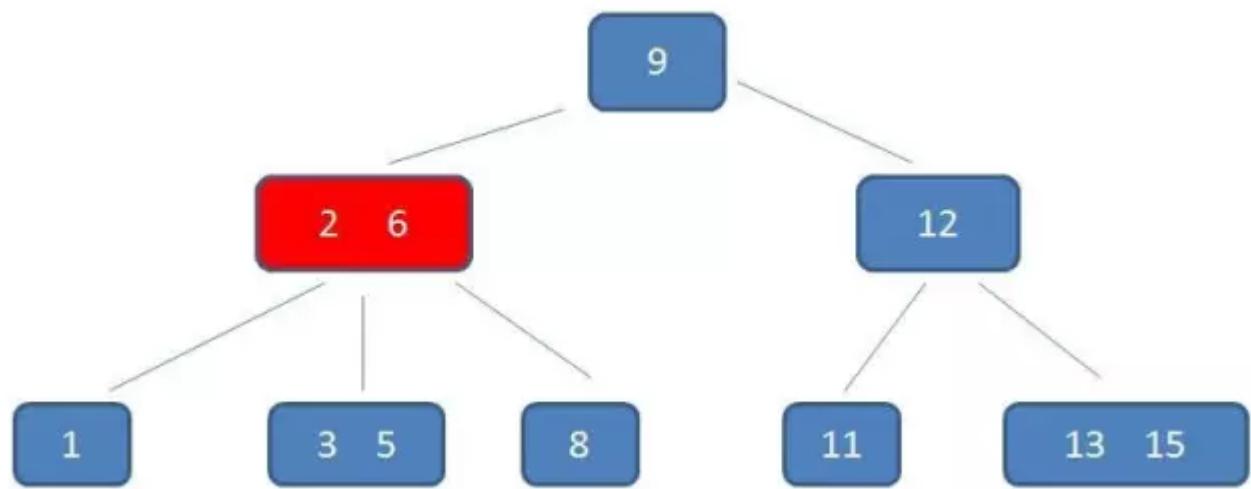
第1次磁盘IO：



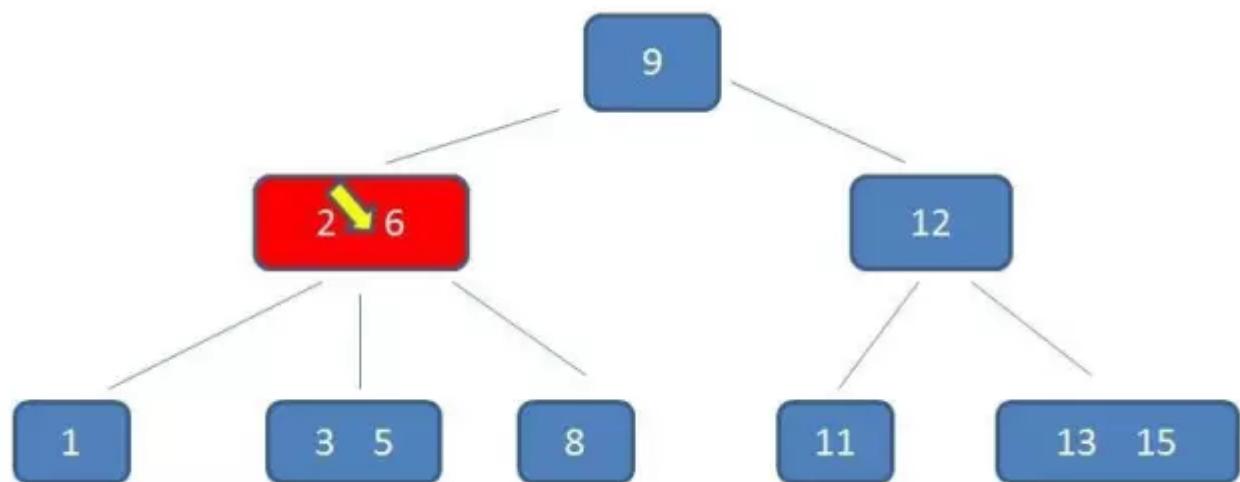
在内存中定位（和9比较）：



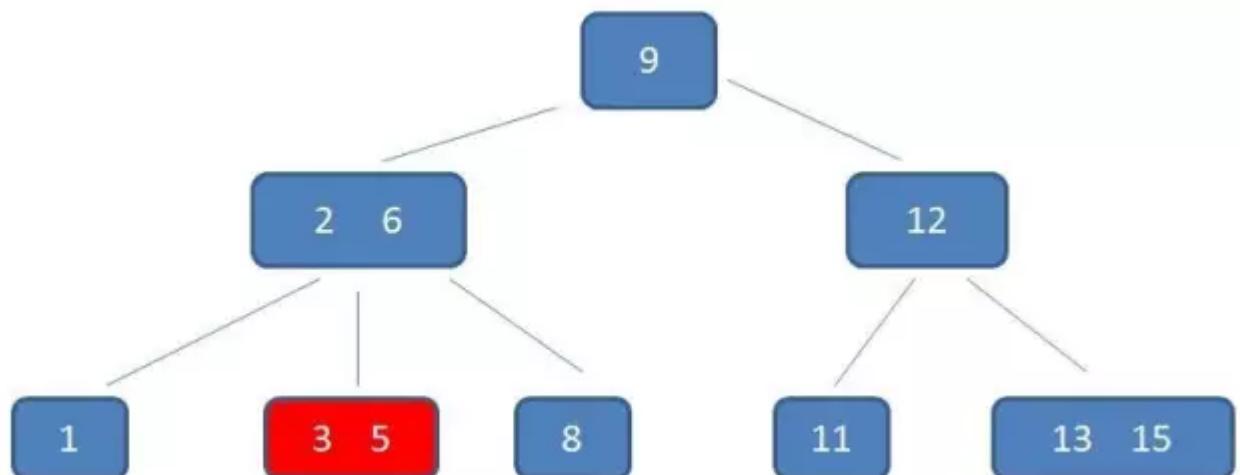
第2次磁盘IO:



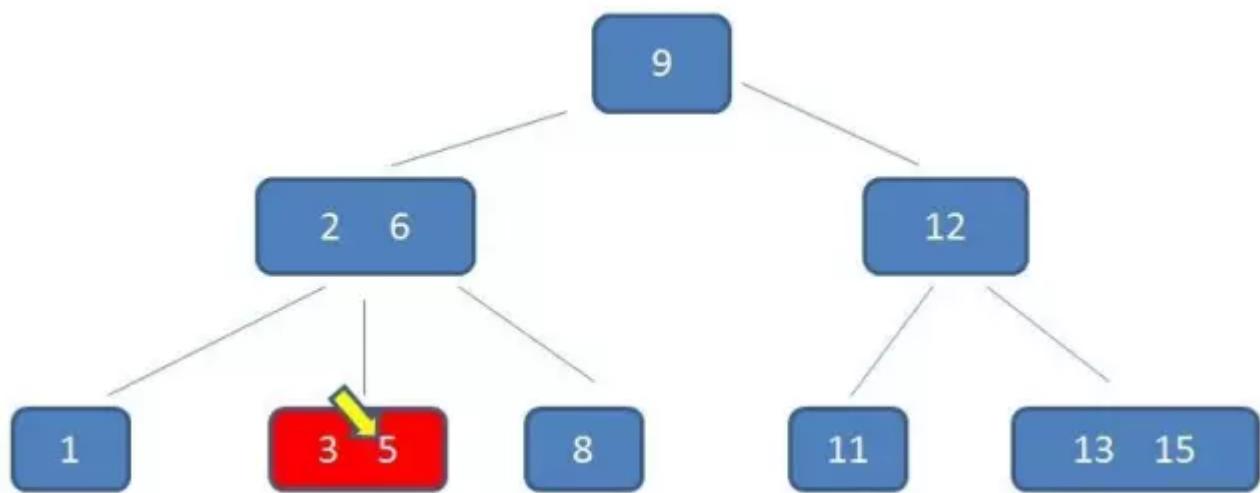
在内存中定位 (和2, 6比较) :



第3次磁盘IO:



在内存中定位 (和3, 5比较) :



通过整个流程我们可以看出，B树在查询中的比较次数其实不比二叉查找树少，尤其是单一节点中元素的数量很多时

可是相比磁盘IO的速度，内存中比较耗时几乎可以忽略。所以只要树的高度足够低，IO次数足够小，就可以提升查找性能

相比之下节点内部元素多一些也没有关系，仅仅是多了几次内存交互，只要不超过磁盘页的大小即可。这就是B树的优势之一

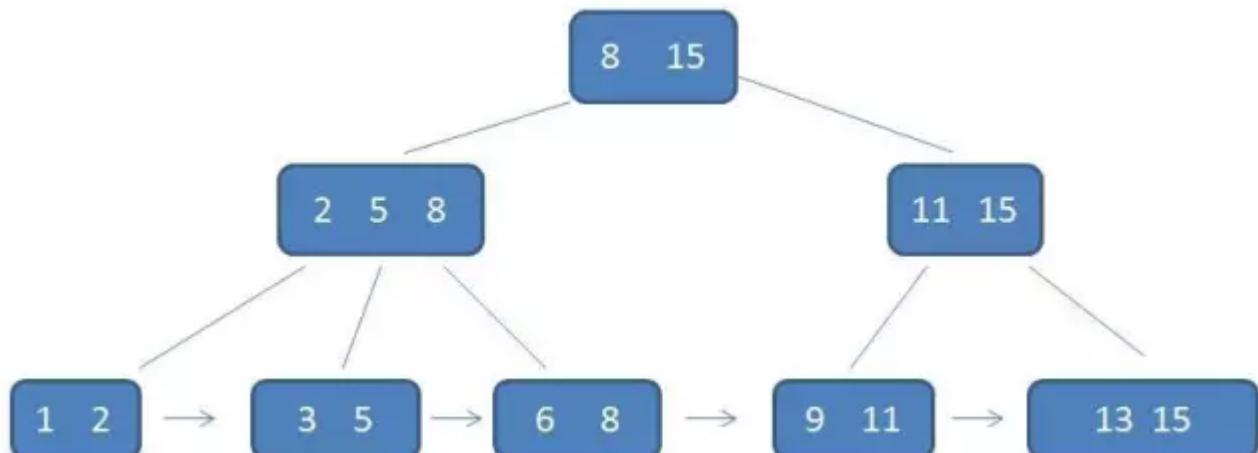
B树的插入删除操作.....

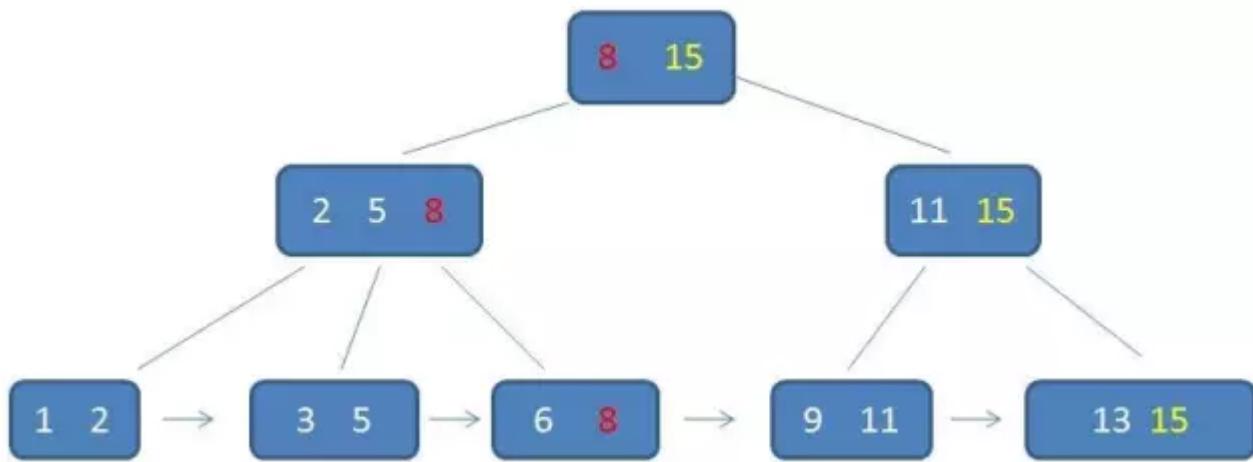
七、B+树

B+树是基于B-树的一种变体，有着比B-树更高的查询性能

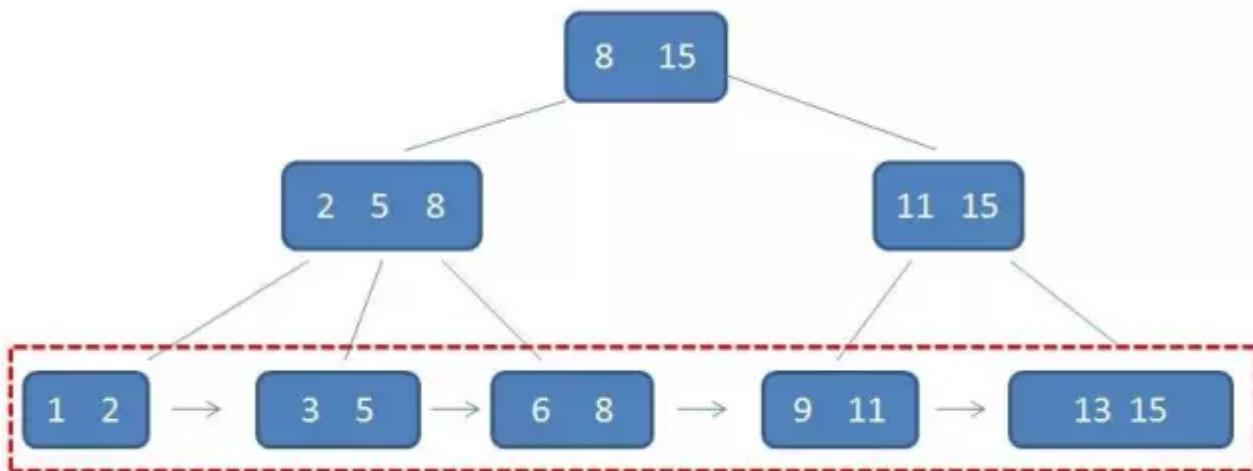
一个m阶的B+树具有如下几个特征：

1. 有k个子树的中间节点包含有k个元素（B树中是k-1个元素），每个元素不保存数据，只用来索引，所有数据都保存在叶子节点。
2. 所有的叶子结点中包含了全部元素的信息，及指向含这些元素记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。
3. 所有的中间节点元素都同时存在于子节点，在子节点元素中是最大（或最小）元素。





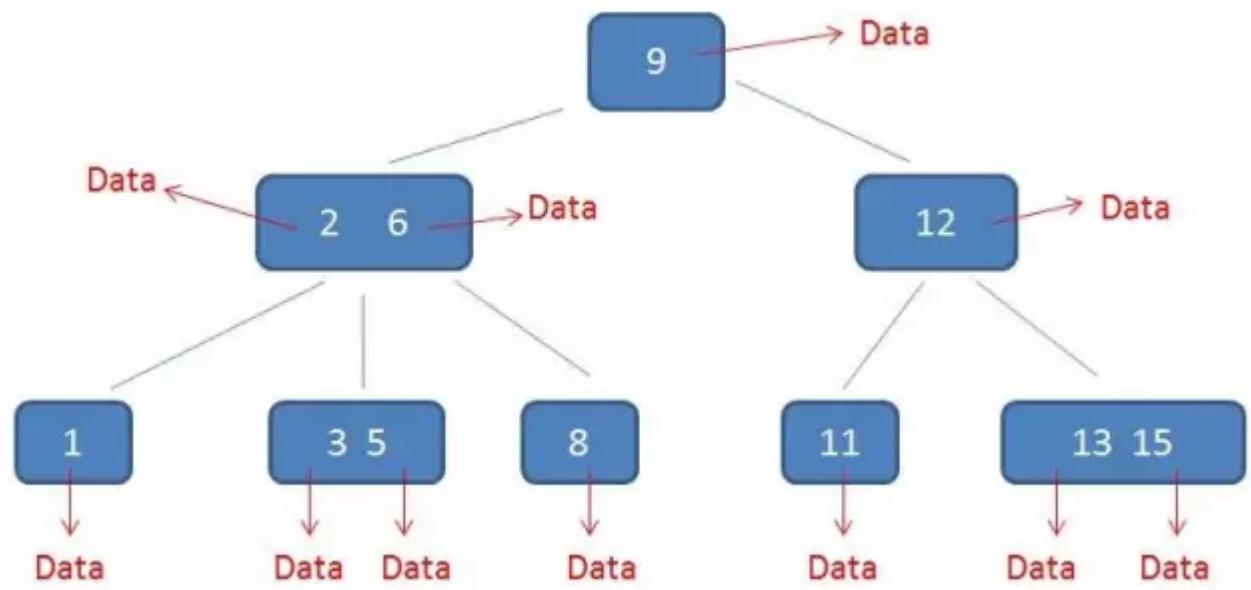
由于父节点的元素都出现在子节点中，因此所有的叶子节点包含了全量元素信息，并且每一个叶子节点都带有指向下一个节点的指针，形成了一个有序链表



B+树还有一个特点，这个特点是在索引之外，确是至关重要的特点，那就是【卫星数据】

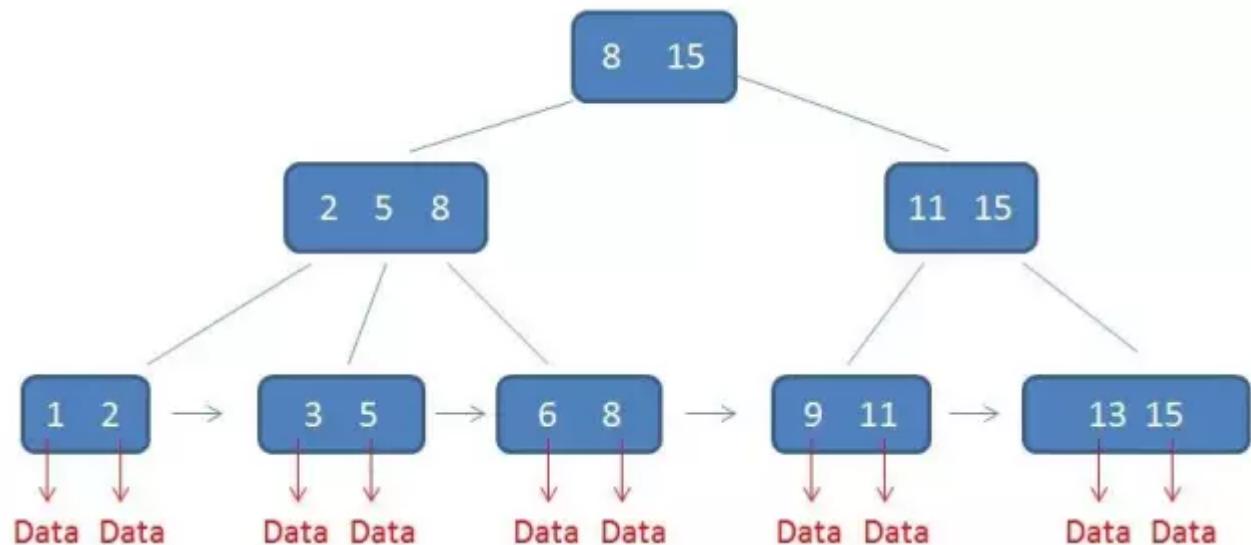
所谓卫星数据，指的是索引元素所指向的数据记录，比如数据库中的某一行。在B-树种，无论中间节点还是叶子节点都带有卫星数据

B-树中的卫星数据 (Satellite Information) :



而在B+树中，只有叶子节点带有卫星数据，其余中间节点仅仅是索引，没有人任何数据关联

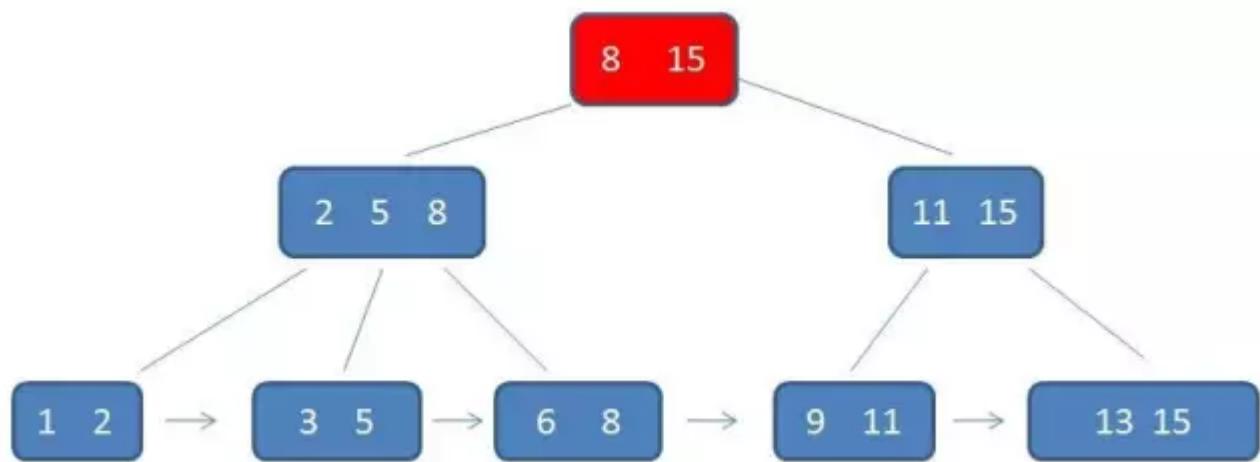
B+树中的卫星数据（Satellite Information）：



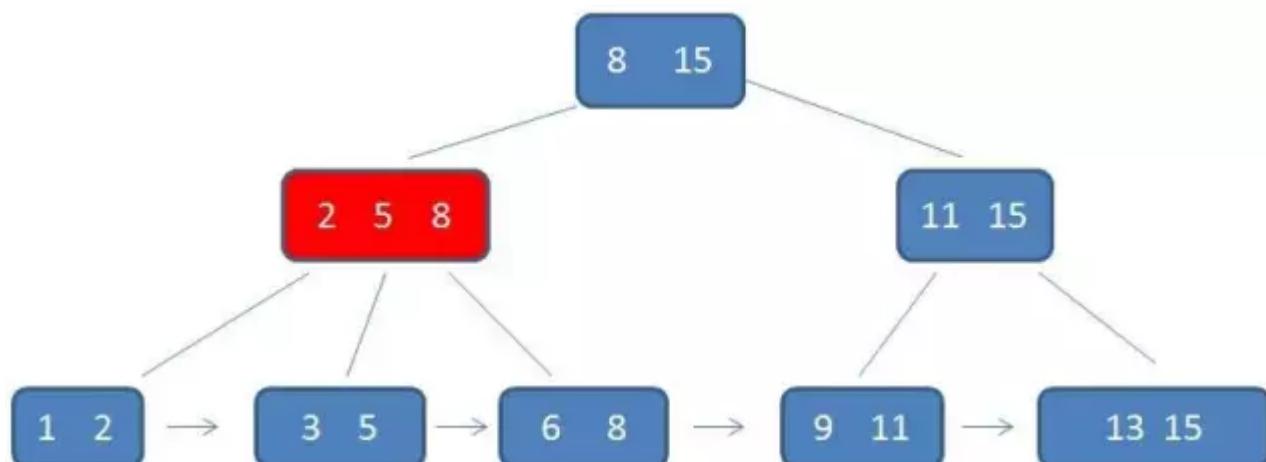
需要补充的是，在数据库的聚集索引（Clustered Index）中，叶子节点直接包含卫星数据。在非聚集索引（NonClustered Index）中，叶子节点带有指向卫星数据的指针。

在B+树中查找元素3，流程如下：

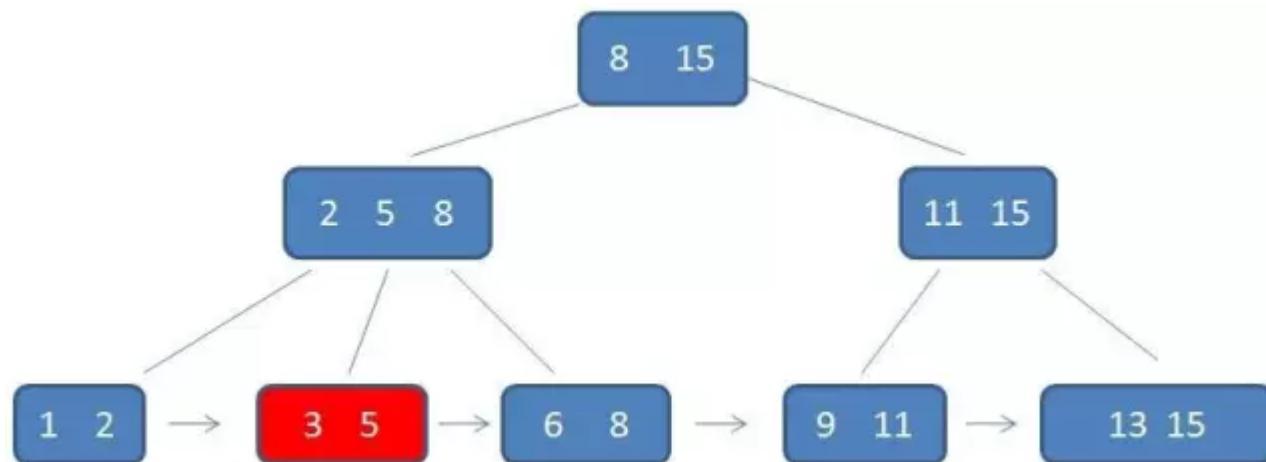
第一次磁盘IO：



第二次磁盘IO:



第三次磁盘IO:



与B-树不同的是，B+树中间节点没有卫星数据，所以同样大小的磁盘页可以容纳更多的节点元素，这意味着，在数据量相同的情况下，B+树的结构比B-树更加“矮胖”，因此查询时IO次数也更少。

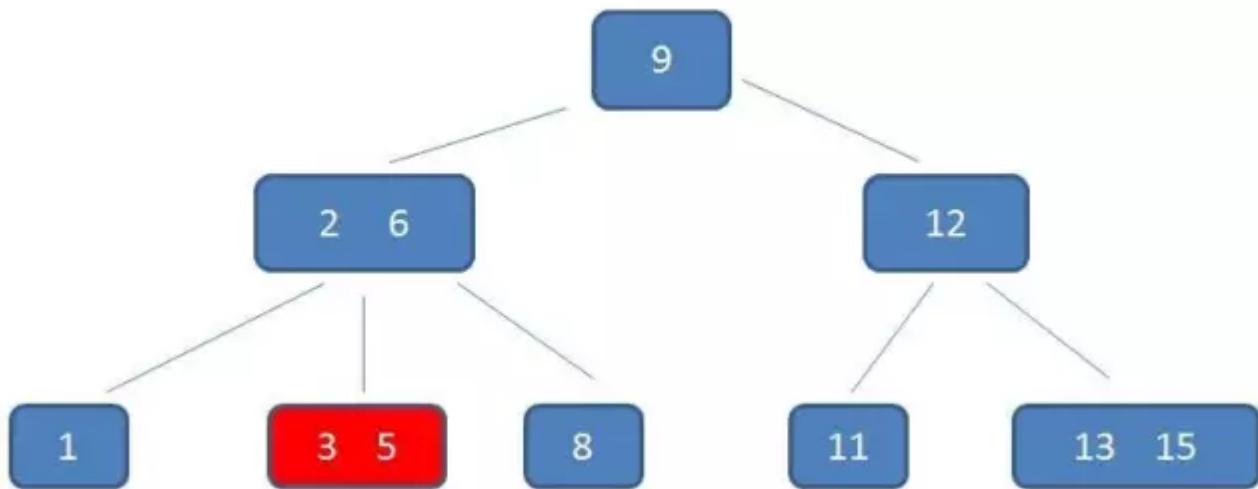
其次，B+树的查询必须最终查找到叶子节点，而B-树只要找到匹配元素即可，无论匹配元素处于中间节点还是叶子节点。

因此，B-树的查找性能并不稳定，最好的情况下直查根节点，最坏的情况下查找到叶子节点，而B+树的每一次查找都是稳定的。

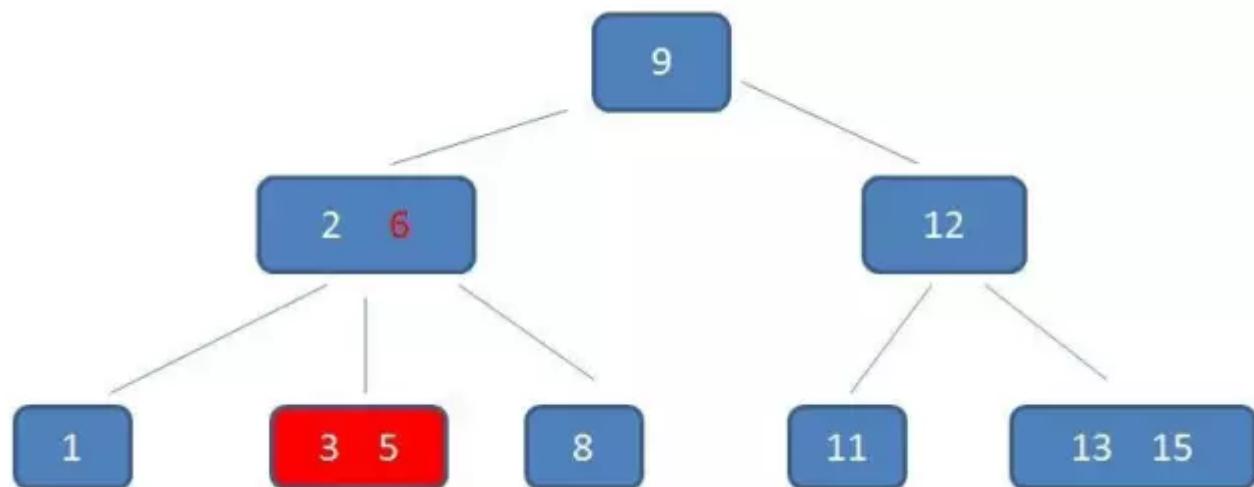
B+树比B-树更适合做范围查询：

B-树的范围查找过程

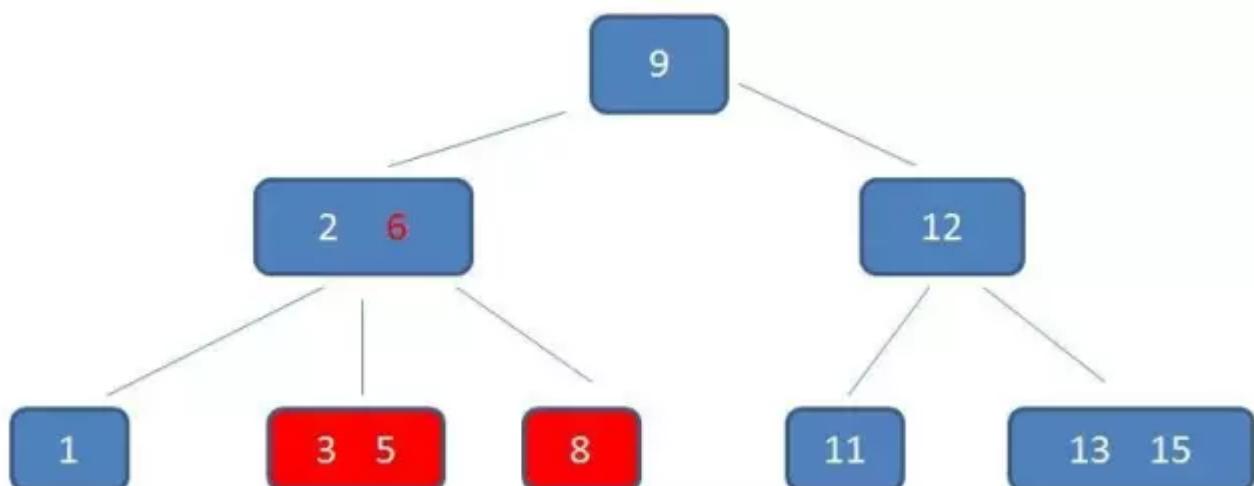
自顶向下，查找到范围的下限（3）：



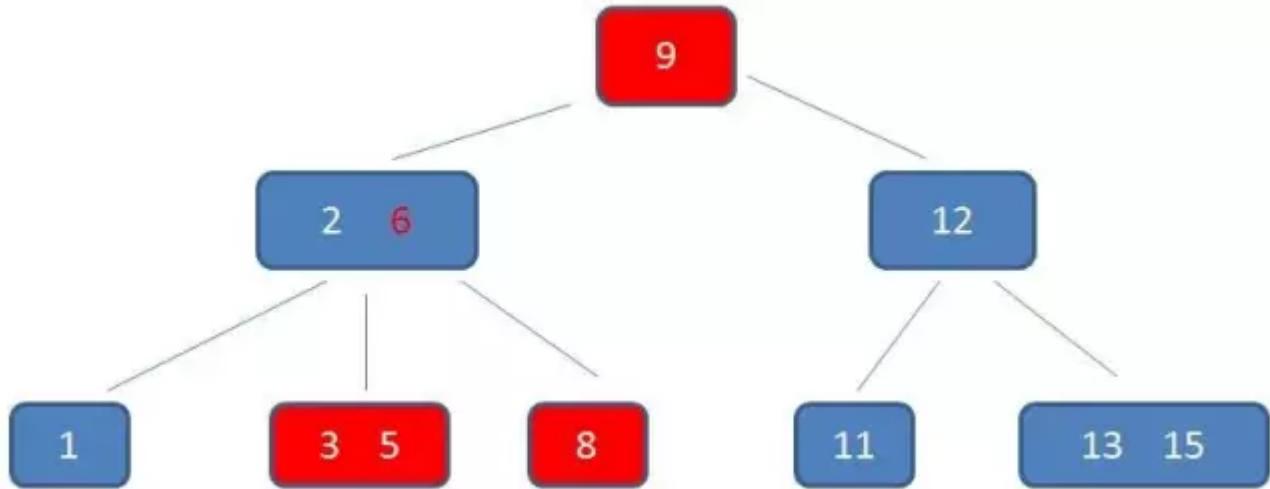
中序遍历到元素6:



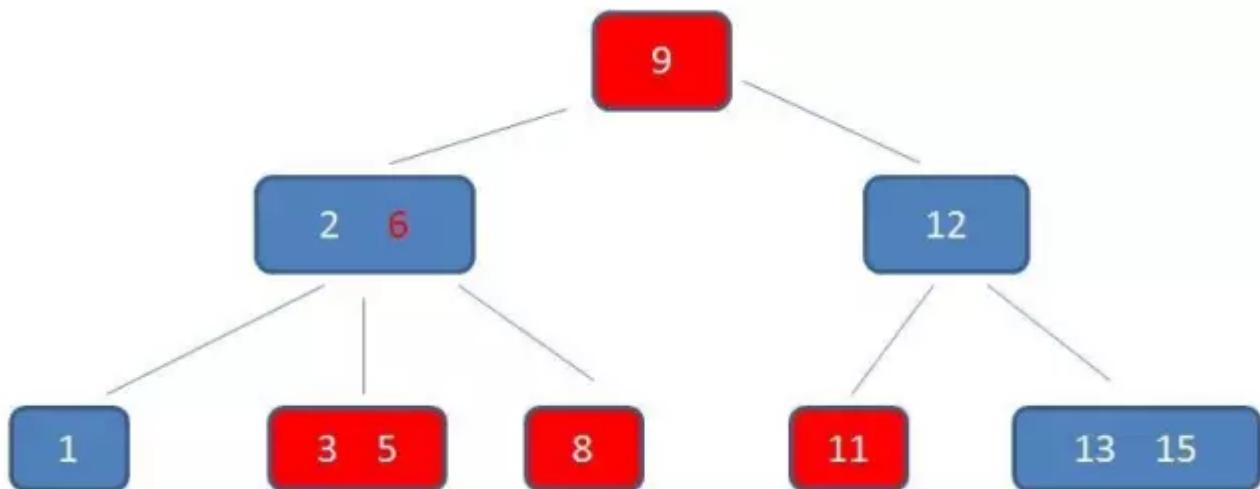
中序遍历到元素8:



中序遍历到元素9:

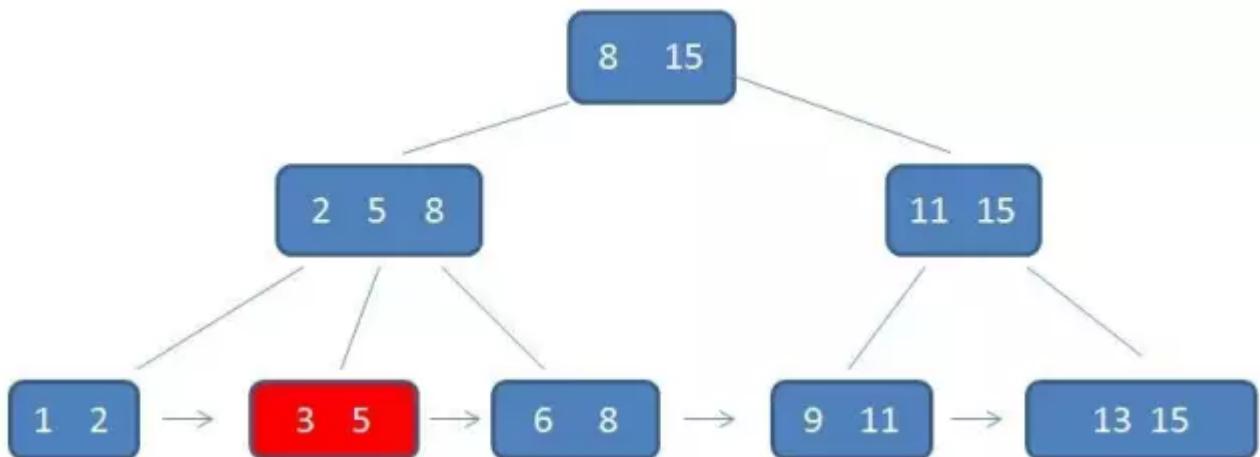


中序遍历到元素11，遍历结束:

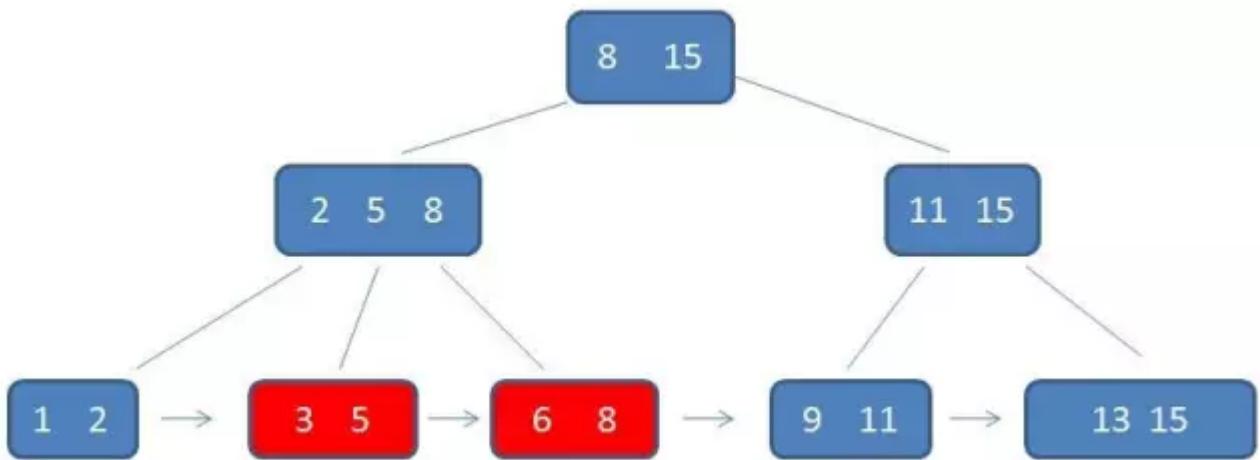


B+树的范围查找过程

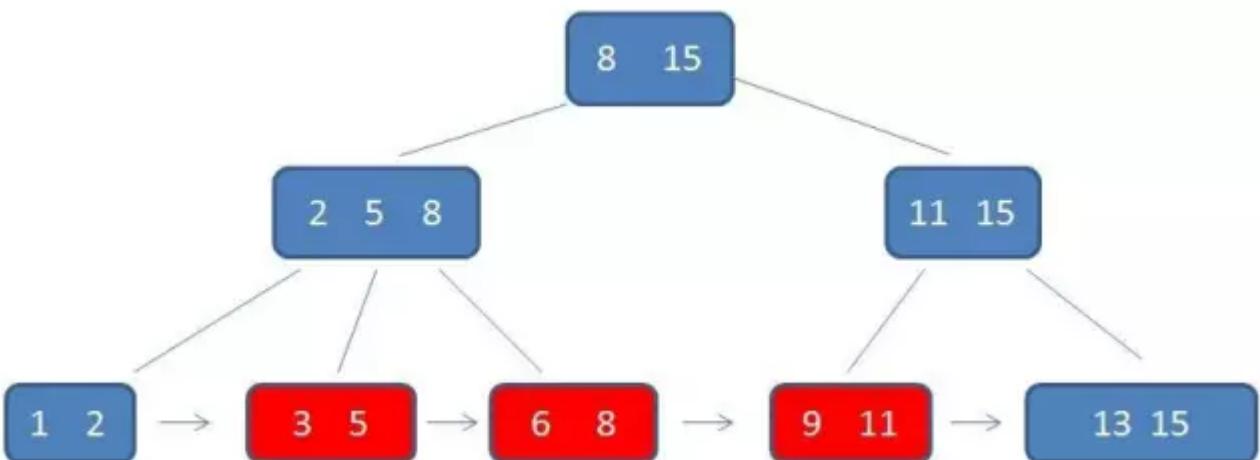
自顶向下，查找到范围的下限 (3) :



通过链表指针，遍历到元素6, 8:



通过链表指针，遍历到元素9, 11，遍历结束：



B+树的优势：

1. 单一节点存储更多的元素，使得查询的IO次数更少。
2. 所有查询都要查找到叶子节点，查询性能稳定。
3. 所有叶子节点形成有序链表，便于范围查询。

八、字典树

字典树

又称单词查找树，Trie树，是一种树形结构，是一种哈希树的变形。典型应用是用于统计，排序和保存大量的字符串，所以经常被搜索引擎系统用于文本词频统计。它的优点是利用最大公共前缀来减少查询时间，最大限度地减少无谓的字符串比较，查询效率比哈希表高。

1538144760624

性质

- 根节点不包含字符，除根节点以外的每一个节点都只包含一个字符；
- 从根节点到某一节点，路径上经过的字符串连接起来，为该节点对应的字符串；
- 每个节点的所有子节点包含的字符都不相同。

实现方法

搜索字典项目的方法：

- 从根节点开始一次搜索；
- 取得要查找关键词的第一个字母，并根据该字母选择对应的子树继续进行检索；
- 在相应的子树上，取得要查找关键词的第二个字母，并进一步选择对应的子树进行检索
- 迭代下去
- 在某个结点处，关键词的所在字母已被取出，则读取附在该结点上的信息，即完成查找。

应用

(1) 串的快速检索

给出N个单词组成的熟词表，以及一篇全用小写英文书写的文章，请你按最早出现的顺序写出所有不在熟词表中的生词。

方法1：可以将英文文章中的所有单词逐个与熟词表进行比较， $O(N)=O(n\text{avg}(\text{length}_1)\text{Avg}(\text{length}_2)) = O(nN)$

方法2：采用hash表，

方法3：采用字典树，将该熟词表构成字典树，然后通过字典树进行查找。建树的时间复杂度： $O(n) = O(N)$ ，查找的时间复杂度只和树的深度相关，而与熟词表中有多少个单词无关，树的深度又与单词的长度有关，而单词最长不过30个字符，因此 $O(N)=O(1)$ ；另外在空间复杂度上又优于其他的算法，由于公共前缀的存在，不需要大量存储重复的字符。

(2) 串的排序

给定N个互不相同的仅由一个单词构成的英文名，让你将他们按字典序从小到大输出。

用字典树进行排序，采用数组的方式创建字典树，因为树的每个结点的所有子结点很显然是按照其字母大小排序的，那么对待这棵树进行先序遍历即可。

(3) 最长公共前缀

对所有串建立字典树

九、跳表

为什么选择跳表

目前经常使用的平衡数据结构有：B树，红黑树，AVL树，Splay Tree, Treep等。

想象一下，给你一张草稿纸，一只笔，一个编辑器，你能立即实现一颗红黑树，或者AVL树出来吗？很难吧，这需要时间，要考虑很多细节，要参考一堆算法与数据结构之类的树，还要参考网上的代码，相当麻烦。

用跳表吧，跳表是一种随机化的数据结构，目前开源软件 Redis 和 LevelDB 都有用到它，它的效率和红黑树以及 AVL 树不相上下，但跳表的原理相当简单，只要你能熟练操作链表，就能轻松实现一个 SkipList。

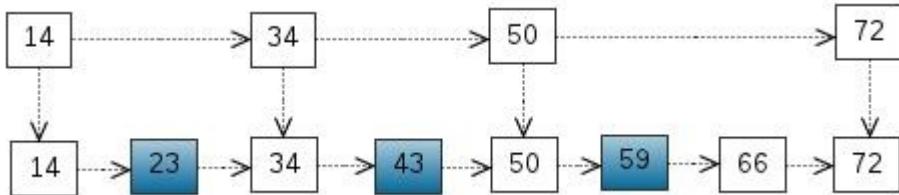
有序表的搜索

考虑一个有序表：



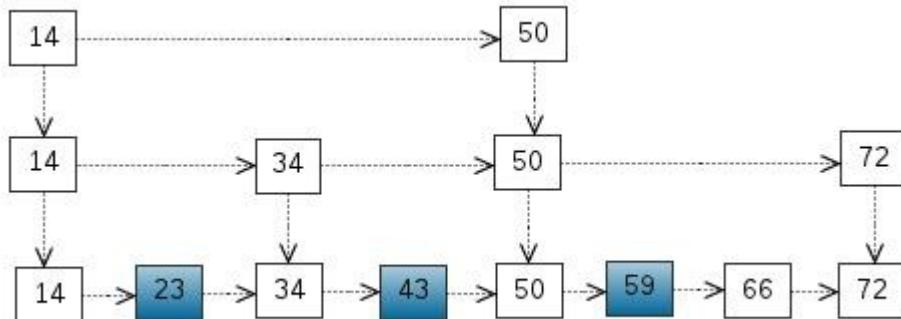
从该有序表中搜索元素 $< 23, 43, 59 >$ ，需要比较的次数分别为 $< 2, 4, 6 >$ ，总共比较的次数为 $2 + 4 + 6 = 12$ 次。有没有优化的算法吗？链表是有序的，但不能使用二分查找。类似二叉

搜索树，我们把一些节点提取出来，作为索引。得到如下结构：



这里我们把 $<14, 34, 50, 72>$ 提取出来作为一级索引，这样搜索的时候就可以减少比较次数了。

我们还可以再从一级索引提取一些元素出来，作为二级索引，变成如下结构：



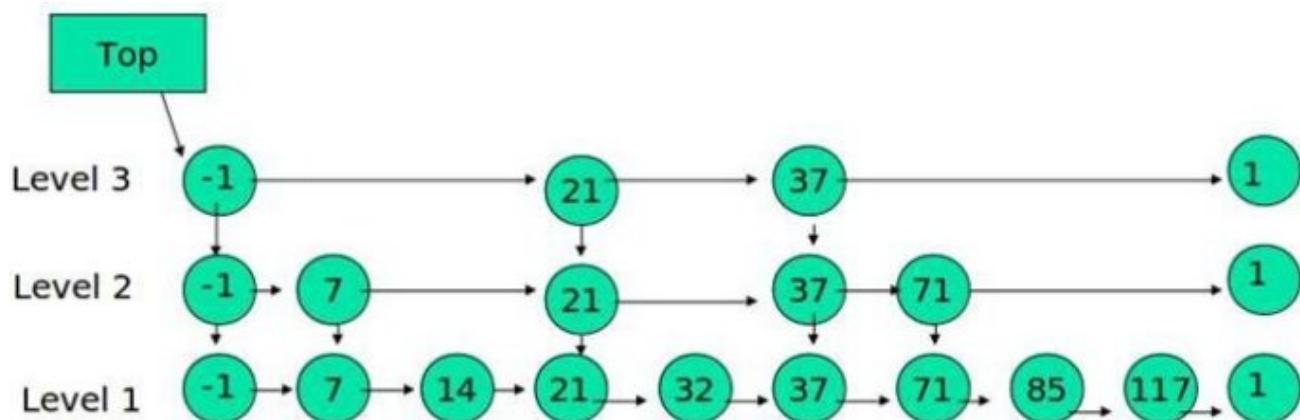
这里元素不多，体现不出优势，如果元素足够多，这种索引结构就能体现出优势来了。

这基本上就是跳表的核心思想，其实也是一种通过“空间来换取时间”的一个算法，通过在每个节点中增加了向前的指针，从而提升查找的效率。

跳表

下面的结构是跳表：

其中 -1 表示 INT_MIN，链表的最小值，1 表示 INT_MAX，链表的最大值。

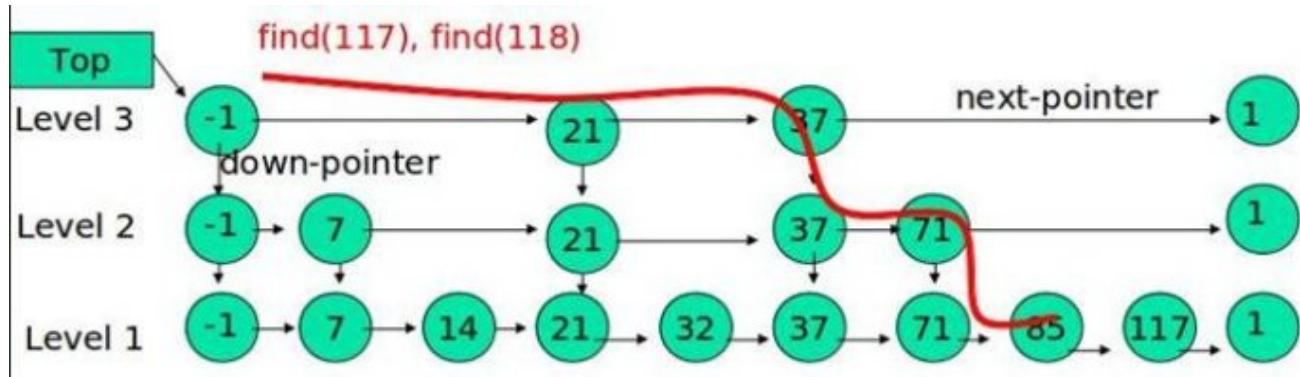


跳表具有如下性质：

- (1) 由很多层结构组成
- (2) 每一层都是一个有序的链表
- (3) 最底层(Level 1)的链表包含所有元素
- (4) 如果一个元素出现在 Level i 的链表中，则它在 Level i 之下的链表也都会出现。

(5) 每个节点包含两个指针，一个指向同一链表中的下一个元素，一个指向下面一层的元素。

跳表的搜索



例子：查找元素 117

- (1) 比较 21, 比 21 大, 往后面找
- (2) 比较 37, 比 37 大, 比链表最大值小, 从 37 的下面一层开始找
- (3) 比较 71, 比 71 大, 比链表最大值小, 从 71 的下面一层开始找
- (4) 比较 85, 比 85 大, 从后面找
- (5) 比较 117, 等于 117, 找到了节点。

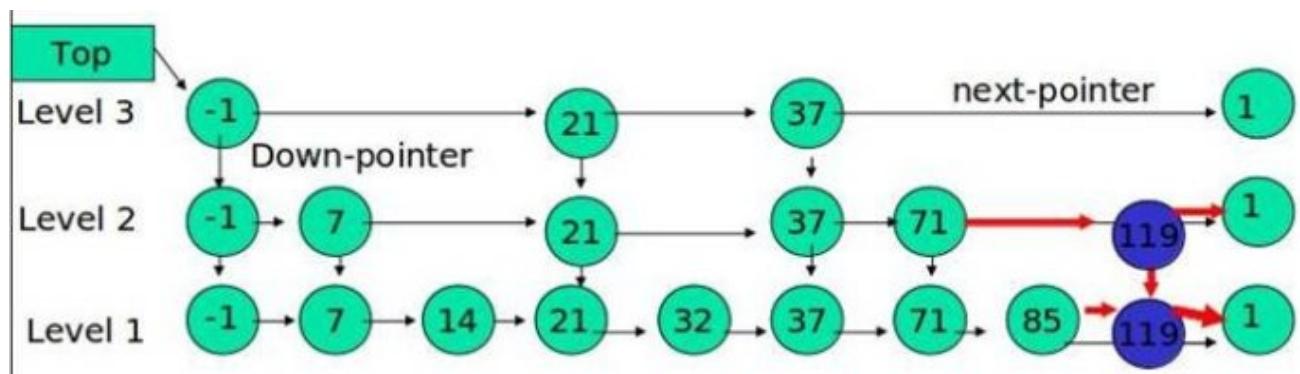
具体的搜索算法如下：

跳表的插入

先确定该元素要占据的层数 K (采用丢硬币的方式, 这完全是随机的)

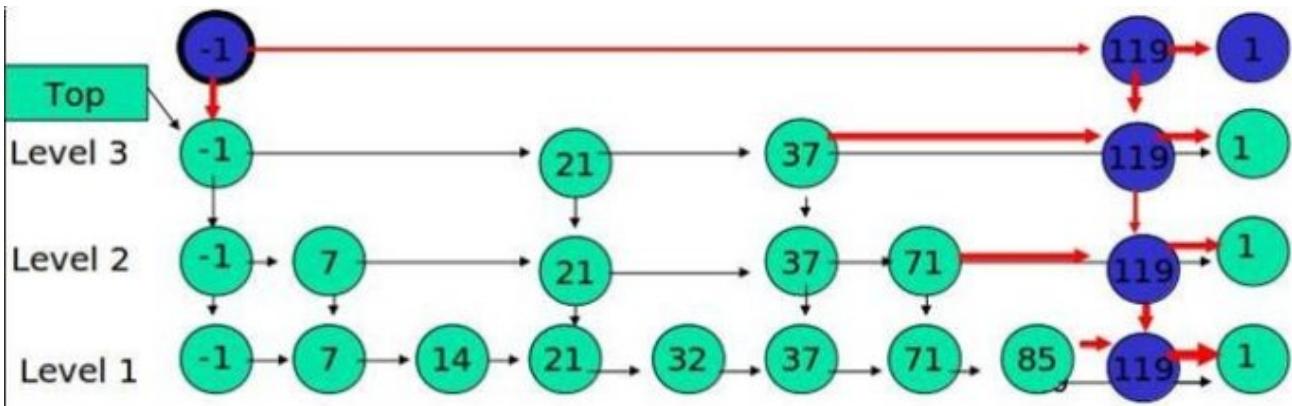
然后在 Level 1 ... Level K 各个层的链表都插入元素。

例子：插入 119, K = 2



如果 K 大于链表的层数，则要添加新的层。

例子：插入 119, K = 4



丢硬币决定 K

插入元素的时候，元素所占有的层数完全是随机的，通过随机算法产生：

相当与做一次丢硬币的实验，如果遇到正面，继续丢，遇到反面，则停止，

用实验中丢硬币的次数 K 作为元素占有的层数。显然随机变量 K 满足参数为 $p = 1/2$ 的几何分布，

K 的期望值 $E[K] = 1/p = 2$. 就是说，各个元素的层数，期望值是 2 层。

跳表的高度。

n 个元素的跳表，每个元素插入的时候都要做一次实验，用来决定元素占据的层数 K ，

跳表的高度等于这 n 次实验中产生的最大 K ，待续。。

跳表的空间复杂度分析

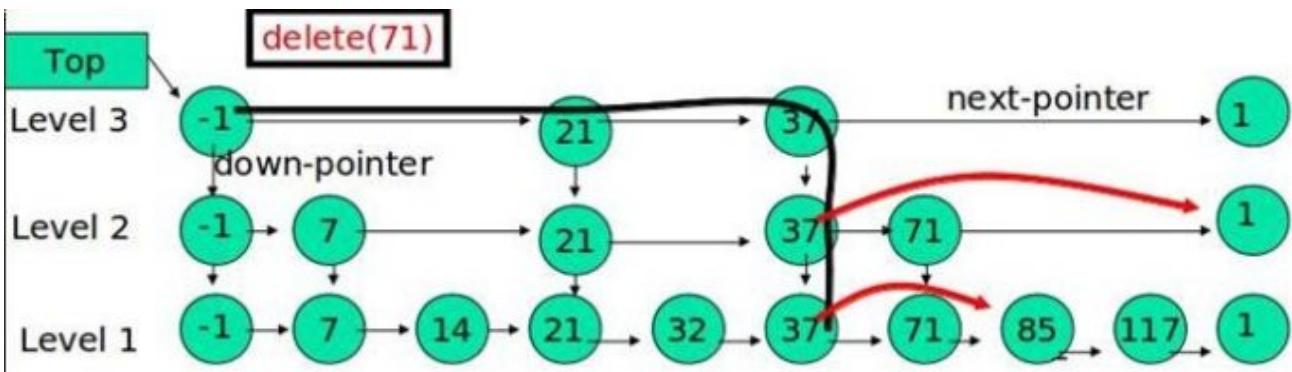
根据上面的分析，每个元素的期望高度为 2，一个大小为 n 的跳表，其节点数目的

期望值是 $2n$ 。

跳表的删除

在各个层中找到包含 x 的节点，使用标准的 delete from list 方法删除该节点。

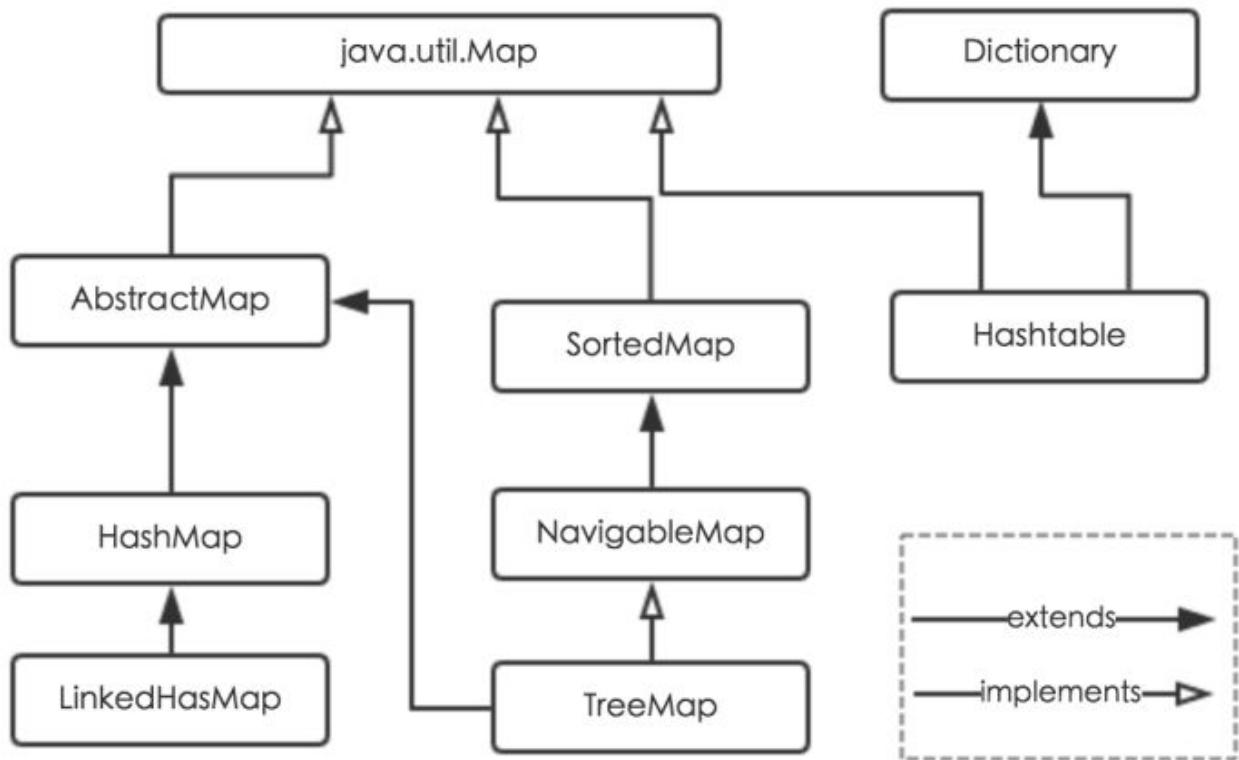
例子：删除 71



十、HashMap

简介

Java为数据结构中的映射定义了一个接口java.util.Map，此接口主要有四个常用的实现类，分别是HashMap、Hashtable、LinkedHashMap和TreeMap，类继承关系如下图所示：



下面针对各个实现类的特点做一些说明：

- (1) **HashMap**: 它根据键的hashCode值存储数据，大多数情况下可以直接定位到它的值，因而具有很快的访问速度，但遍历顺序却是不确定的。HashMap最多只允许一条记录的键为null，允许多条记录的值为null。HashMap非线程安全，即任一时刻可以有多个线程同时写HashMap，可能会导致数据的不一致。如果需要满足线程安全，可以用Collections的synchronizedMap方法使HashMap具有线程安全的能力，或者使用ConcurrentHashMap。
- (2) **Hashtable**: Hashtable是遗留类，很多映射的常用功能与HashMap类似，不同的是它承自Dictionary类，并且是线程安全的，任一时间只有一个线程能写Hashtable，并发性不如ConcurrentHashMap，因为ConcurrentHashMap引入了分段锁。Hashtable不建议在新代码中使用，不需要线程安全的场合可以用HashMap替换，需要线程安全的场合可以用ConcurrentHashMap替换。
- (3) **LinkedHashMap**: LinkedHashMap是HashMap的一个子类，保存了记录的插入顺序，在用Iterator遍历 LinkedHashMap时，先得到的记录肯定是先插入的，也可以在构造时带参数，按照访问次序排序。
- (4) **TreeMap**: TreeMap实现SortedMap接口，能够把它保存的记录根据键排序，默认是按键值的升序排序，也可以指定排序的比较器，当用Iterator遍历TreeMap时，得到的记录是排过序的。如果使用排序的映射，建议使用TreeMap。在使用TreeMap时，key必须实现Comparable接口或者在构造TreeMap传入自定义的Comparator，否则会在运行时抛出java.lang.ClassCastException类型的异常。

对于上述四种Map类型的类，要求映射中的key是不可变对象。不可变对象是该对象在创建后它的哈希值不会被改变。如果对象的哈希值发生变化，Map对象很可能就定位不到映射的位置了。

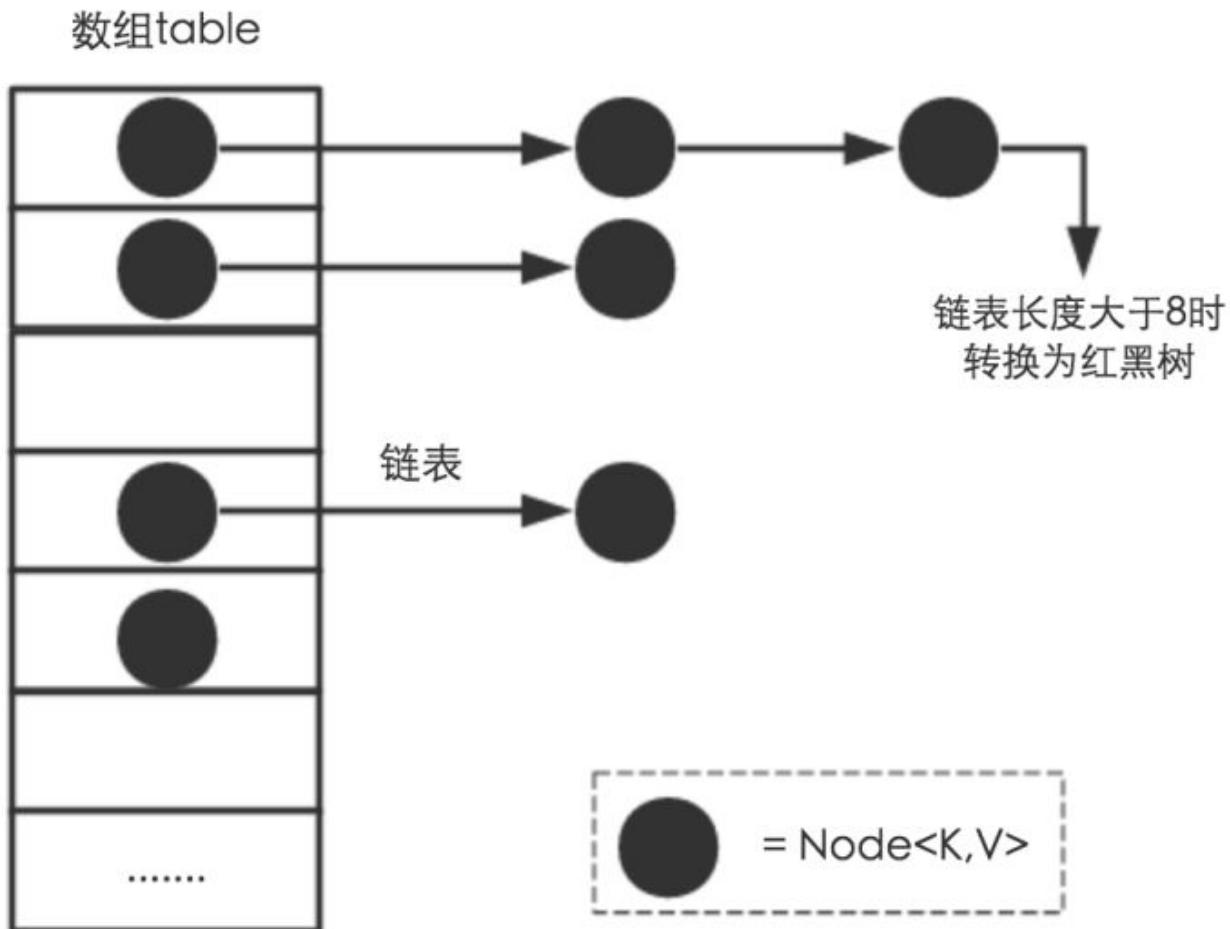
通过上面的比较，我们知道HashMap是Java的Map家族中一个普通成员，鉴于它可以满足大多数场景的使用条件，所以是使用频度最高的一个。下文我们主要结合源码，从存储结构、常用方法分析、扩容以及安全性等方面深入讲解HashMap的工作原理。

内部实现

搞清楚HashMap，首先需要知道HashMap是什么，即它的存储结构-字段；其次弄明白它能干什么，即它的功能实现-方法。下面我们针对这两个方面详细展开讲解。

存储结构-字段

从结构实现来讲，HashMap是数组+链表+红黑树（JDK1.8增加了红黑树部分）实现的，如下如所示。



这里需要讲明白两个问题：数据底层具体存储的是什么？这样的存储方式有什么优点呢？

(1) 从源码可知，HashMap类中有一个非常重要的字段，就是 Node[] table，即哈希桶数组，明显它是一个Node的数组。我们来看Node[JDK1.8]是何物。

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;      //用来定位数组索引位置
    final K key;
    V value;
    Node<K,V> next;    //链表的下一个node

    Node(int hash, K key, V value, Node<K,V> next) { ... }
    public final K getKey(){ ... }
    public final V getValue() { ... }
    public final String toString() { ... }
    public final int hashCode() { ... }
    public final V setValue(V newValue) { ... }

    public final boolean equals(Object o) { ... }
```

```
}
```

Node是HashMap的一个内部类，实现了Map.Entry接口，本质就是映射(键值对)。上图中的每个黑色圆点就是一个Node对象。

(2) HashMap就是使用哈希表来存储的。哈希表为解决冲突，可以采用开放地址法和链地址法等来解决问题，Java中HashMap采用了链地址法。链地址法，简单来说，就是数组加链表的结合。在每个数组元素上都一个链表结构，当数据被Hash后，得到数组下标，把数据放在对应下标元素的链表上。例如程序执行下面代码：

```
map.put("美团", "小美");
```

系统将调用"美团"这个key的hashCode()方法得到其hashCode值（该方法适用于每个Java对象），然后再通过Hash算法的后两步运算（高位运算和取模运算，下文有介绍）来定位该键值对的存储位置，有时两个key会定位到相同的位置，表示发生了Hash碰撞。当然Hash算法计算结果越分散均匀，Hash碰撞的概率就越小，map的存取效率就会越高。

如果哈希桶数组很大，即使较差的Hash算法也会比较分散，如果哈希桶数组数组很小，即使好的Hash算法也会出现较多碰撞，所以就需要在空间成本和时间成本之间权衡，其实就是在根据实际情况确定哈希桶数组的大小，并在此基础上设计好的hash算法减少Hash碰撞。那么通过什么方式来控制map使得Hash碰撞的概率又小，哈希桶数组(Node[] table) 占用空间又少呢？答案就是好的Hash算法和扩容机制。

在理解Hash和扩容流程之前，我们得先了解下HashMap的几个字段。从HashMap的默认构造函数源码可知，构造函数就是对下面几个字段进行初始化，源码如下：

```
int threshold;          // 所能容纳的key-value对极限
final float loadFactor; // 负载因子
int modCount;
int size;
```

首先，Node[] table的初始化长度length(默认值是16)，Load factor为负载因子(默认值是0.75)，threshold是HashMap所能容纳的最大数据量的Node(键值对)个数。threshold = length * Load factor。也就是说，在数组定义好长度之后，负载因子越大，所能容纳的键值对个数越多。

结合负载因子的定义公式可知，threshold就是在此Load factor和length(数组长度)对应下允许的最大元素数目，超过这个数目就重新resize(扩容)，扩容后的HashMap容量是之前容量的两倍。默认的负载因子0.75是对空间和时间效率的一个平衡选择，建议大家不要修改，除非在时间和空间比较特殊的情况下，如果内存空间很多而又对时间效率要求很高，可以降低负载因子Load factor的值；相反，如果内存空间紧张而对时间效率要求不高，可以增加负载因子loadFactor的值，这个值可以大于1。

size这个字段其实很好理解，就是HashMap中实际存在的键值对数量。注意和table的长度length、容纳最大键值对数量threshold的区别。而modCount字段主要用来记录HashMap内部结构发生变化的次数，主要用于迭代的快速失败。强调一点，内部结构发生变化指的是结构发生变化，例如put新键值对，但是某个key对应的value值被覆盖不等于结构变化。

在HashMap中，哈希桶数组table的长度length大小必须为2的n次方(一定是合数)，这是一种非常规的设计，常规的设计是把桶的大小设计为素数。相对来说素数导致冲突的概率要小于合数，具体证明可以参考http://blog.csdn.net/liuqiyao_01/article/details/14475159，Hashtable初始化桶大小为11，就是桶大小设计为素数的应用(Hashtable扩容后不能保证还是素数)。HashMap采用这种非常规设计，主要是为了在取模和扩容时做优化，同时为了减少冲突，HashMap定位哈希桶索引位置时，也加入了高位参与运算的过程。

这里存在一个问题，即使负载因子和Hash算法设计的再合理，也免不了会出现拉链过长的情况，一旦出现拉链过长，则会严重影响HashMap的性能。于是，在JDK1.8版本中，对数据结构做了进一步的优化，引入了红黑树。而当链表长度太长（默认超过8）时，链表就转换为红黑树，利用红黑树快速增删改查的特点提高HashMap的性能，其中会用到红黑树的插入、删除、查找等算法。本文不再对红黑树展开讨论，想了解更多红黑树数据结构的工作原理可以参考http://blog.csdn.net/v_july_v/article/details/6105630。

功能实现-方法

HashMap的内部功能实现很多，本文主要从根据key获取哈希桶数组索引位置、put方法的详细执行、扩容过程三个具有代表性的点深入展开讲解。

1. 确定哈希桶数组索引位置

不管增加、删除、查找键值对，定位到哈希桶数组的位置都是很关键的第一步。前面说过HashMap的数据结构是数组和链表的结合，所以我们当然希望这个HashMap里面的元素位置尽量分布均匀些，尽量使得每个位置上的元素数量只有一个，那么当我们用hash算法求得这个位置的时候，马上就可以知道对应位置的元素就是我们要的，不用遍历链表，大大优化了查询的效率。HashMap定位数组索引位置，直接决定了hash方法的离散性能。先看看源码的实现(方法一+方法二)：

方法一：

```
static final int hash(Object key) {    //jdk1.8 & jdk1.7
    int h;
    // h = key.hashCode() 为第一步 取hashCode值
    // h ^ (h >>> 16) 为第二步 高位参与运算
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

方法二：

```
static int indexFor(int h, int length) {    //jdk1.7的源码，jdk1.8没有这个方法，但是实现原理一样的
    return h & (length-1);    //第三步 取模运算
}
```

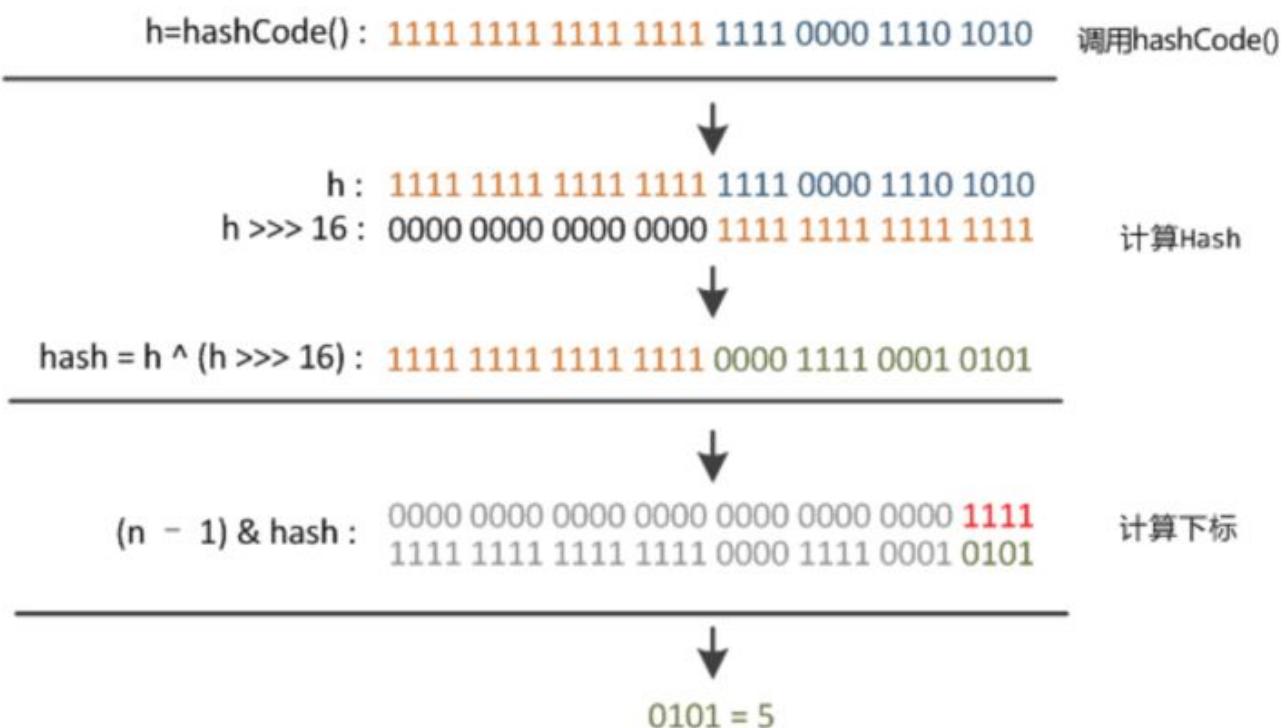
这里的Hash算法本质上就是三步：**取key的hashCode值、高位运算、取模运算**。

对于任意给定的对象，只要它的hashCode()返回值相同，那么程序调用方法一所计算得到的Hash码值总是相同的。我们首先想到的就是把hash值对数组长度取模运算，这样一来，元素的分布相对来说是比较均匀的。但是，模运算的消耗还是比较大的，在HashMap中是这样做的：调用方法二来计算该对象应该保存在table数组的哪个索引处。

这个方法非常巧妙，它通过 $h \& (table.length - 1)$ 来得到该对象的保存位，而HashMap底层数组的长度总是2的n次方，这是HashMap在速度上的优化。当length总是2的n次方时， $h \& (length - 1)$ 运算等价于对length取模，也就是 $h \% length$ ，但是&%比&&具有更高的效率。

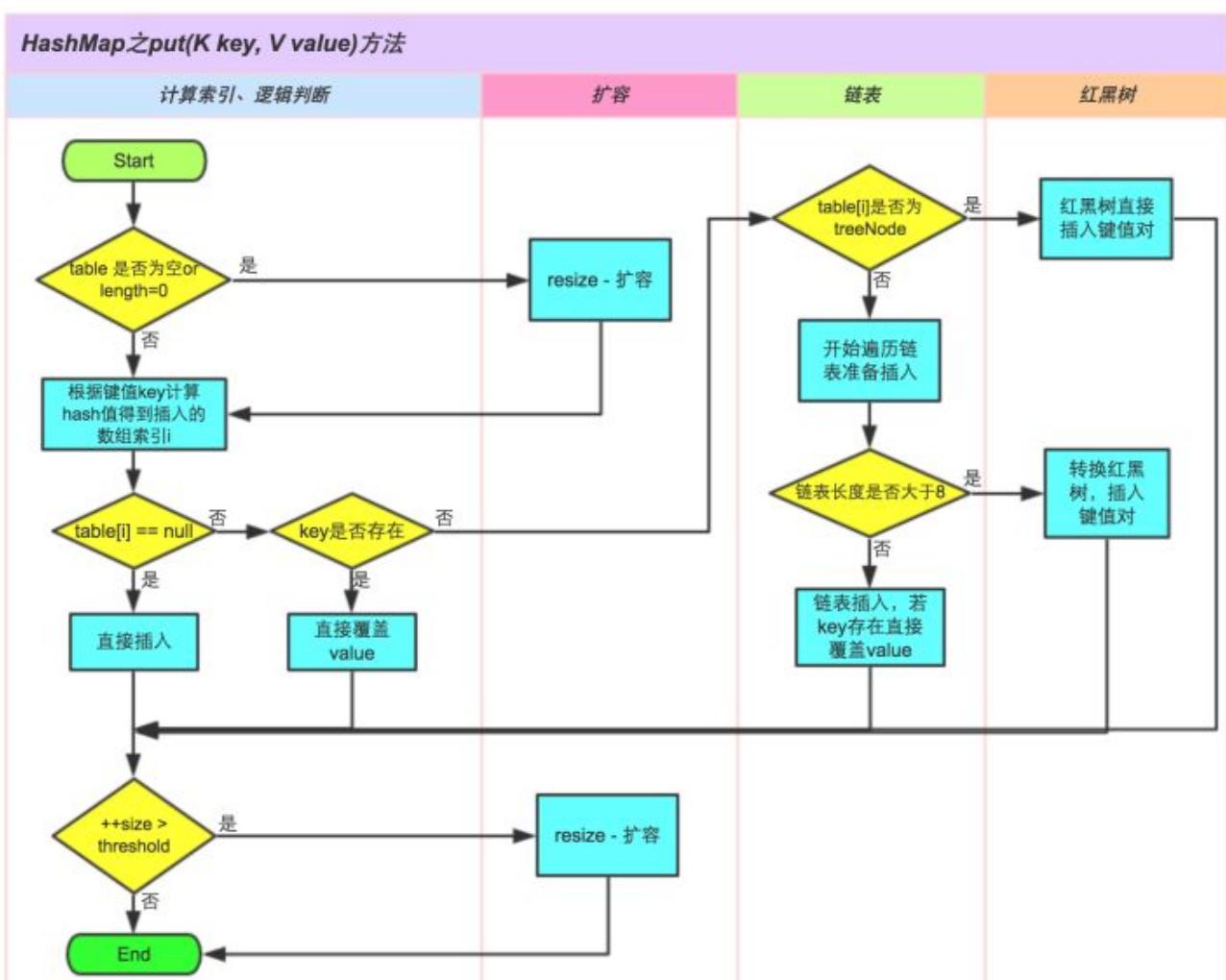
在JDK1.8的实现中，优化了高位运算的算法，通过hashCode()的高16位异或低16位实现的： $(h = k.hashCode()) ^ (h >>> 16)$ ，主要是从速度、功效、质量来考虑的，这么做可以在数组table的length比较小的时候，也能保证考虑到高低Bit都参与到Hash的计算中，同时不会有太大的开销。

下面举例说明下，n为table的长度。



2. 分析HashMap的put方法

HashMap的put方法执行过程可以通过下图来理解，自己有兴趣可以去对比源码更清楚地研究学习。



- ①.判断键值对数组table[i]是否为空或为null，否则执行resize()进行扩容；
- ②.根据键值key计算hash值得到插入的数组索引i，如果table[i]==null，直接新建节点添加，转向⑥，如果table[i]不为空，转向③；
- ③.判断table[i]的首个元素是否和key一样，如果相同直接覆盖value，否则转向④，这里的相同指的是hashCode以及equals；
- ④.判断table[i] 是否为treeNode，即table[i] 是否是红黑树，如果是红黑树，则直接在树中插入键值对，否则转向⑤；
- ⑤.遍历table[i]，判断链表长度是否大于8，大于8的话把链表转换为红黑树，在红黑树中执行插入操作，否则进行链表的插入操作；遍历过程中若发现key已经存在直接覆盖value即可；
- ⑥.插入成功后，判断实际存在的键值对数量size是否超多了最大容量threshold，如果超过，进行扩容。

JDK1.8HashMap的put方法源码如下：

```

1 public V put(K key, V value) {
2     // 对key的hashCode()做hash
3     return putVal(hash(key), key, value, false, true);
4 }
5
6 final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
7                 boolean evict) {
8     Node<K,V>[] tab; Node<K,V> p; int n, i;
9     // 步骤①: tab为空则创建
10    if ((tab = table) == null || (n = tab.length) == 0)
11        n = (tab = resize()).length;
12    // 步骤②: 计算index，并对null做处理
13    if ((p = tab[i = (n - 1) & hash]) == null)
14        tab[i] = newNode(hash, key, value, null);
15    else {
16        Node<K,V> e; K k;
17        // 步骤③: 节点key存在，直接覆盖value
18        if (p.hash == hash &&
19            ((k = p.key) == key || (key != null && key.equals(k))))
20            e = p;
21        // 步骤④: 判断该链为红黑树
22        else if (p instanceof TreeNode)
23            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
24        // 步骤⑤: 该链为链表
25        else {
26            for (int binCount = 0; ; ++binCount) {
27                if ((e = p.next) == null) {
28                    p.next = newNode(hash, key, value, null);
29                    // 链表长度大于8转换为红黑树进行处理
30                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
31                        treeifyBin(tab, hash);
32                    break;
33                }
34                // key已经存在直接覆盖value
35                if (e.hash == hash &&
36                    ((k = e.key) == key || (key != null && key.equals(k))))
```

```

        break;
36         p = e;
37     }
38 }
39
40     if (e != null) { // existing mapping for key
41         V oldValue = e.value;
42         if (!onlyIfAbsent || oldValue == null)
43             e.value = value;
44         afterNodeAccess(e);
45         return oldValue;
46     }
47 }

48     ++modCount;
49     // 步骤⑤：超过最大容量 就扩容
50     if (++size > threshold)
51         resize();
52     afterNodeInsertion(evict);
53     return null;
54 }

```

3. 扩容机制

扩容(resize)就是重新计算容量，向HashMap对象里不停的添加元素，而HashMap对象内部的数组无法装载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素。当然Java里的数组是无法自动扩容的，方法是使用一个新的数组代替已有的容量小的数组，就像我们用一个小桶装水，如果想装更多的水，就得换大水桶。

我们分析下resize的源码，鉴于JDK1.8融入了红黑树，较复杂，为了便于理解我们仍然使用JDK1.7的代码，好理解一些，本质上区别不大，具体区别后文再说。

```

1 void resize(int newCapacity) { //传入新的容量
2     Entry[] oldTable = table; //引用扩容前的Entry数组
3     int oldCapacity = oldTable.length;
4     if (oldCapacity == MAXIMUM_CAPACITY) { //扩容前的数组大小如果已经达到最大(2^30)了
5         threshold = Integer.MAX_VALUE; //修改阈值为int的最大值(2^31-1)，这样以后就不会扩容了
6         return;
7     }
8
9     Entry[] newTable = new Entry[newCapacity]; //初始化一个新的Entry数组
10    transfer(newTable); //将数据转移到新的Entry数组里
11    table = newTable; //HashMap的table属性引用新的Entry数组
12    threshold = (int)(newCapacity * loadFactor); //修改阈值
13 }

```

这里就是使用一个容量更大的数组来代替已有的容量小的数组，transfer()方法将原有Entry数组的元素拷贝到新的Entry数组里。

```

1 void transfer(Entry[] newTable) {
2     Entry[] src = table; //src引用了旧的Entry数组
3     int newCapacity = newTable.length;

```

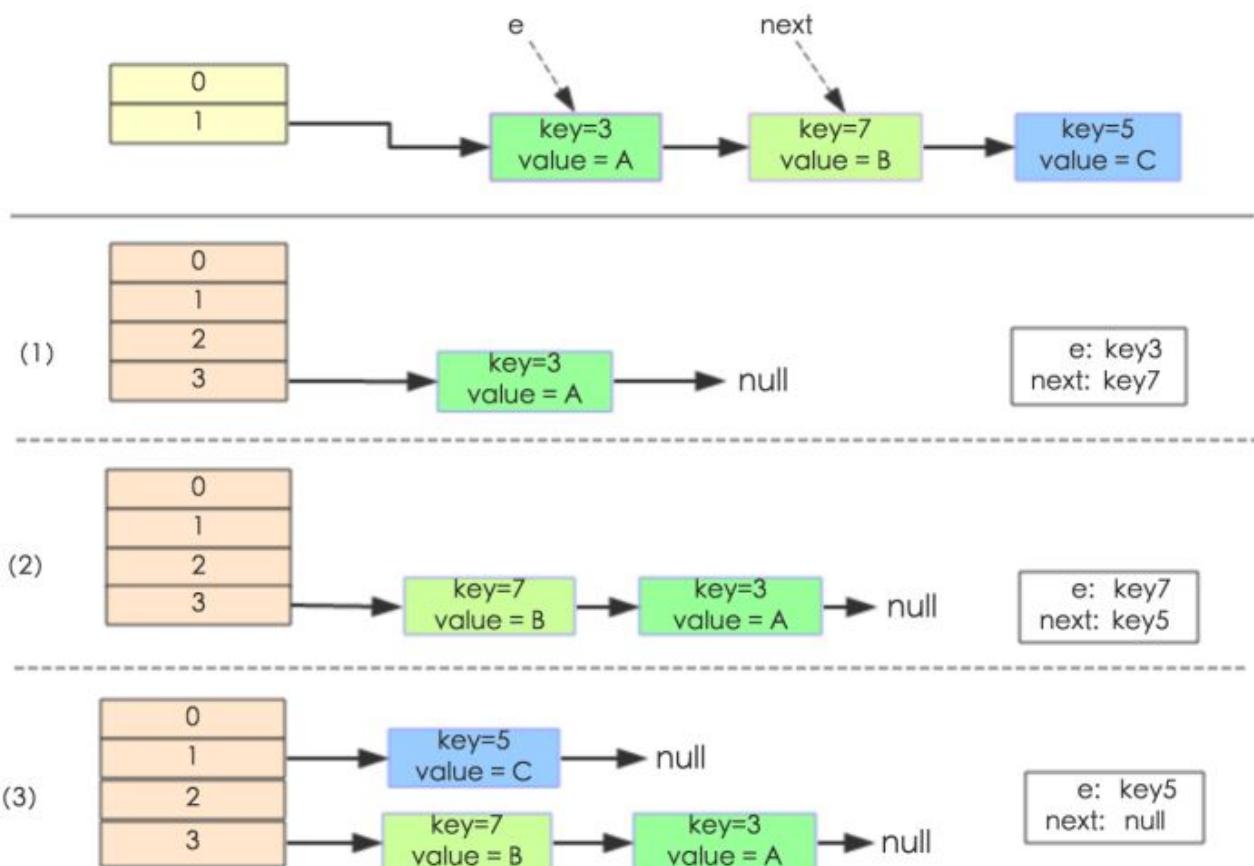
```

4     for (int j = 0; j < src.length; j++) { //遍历旧的Entry数组
5         Entry<K,V> e = src[j];           //取得旧Entry数组的每个元素
6         if (e != null) {
7             src[j] = null; //释放旧Entry数组的对象引用 (for循环后, 旧的Entry数组不再引用任何对象)
8             do {
9                 Entry<K,V> next = e.next;
10                int i = indexFor(e.hash, newCapacity); //! ! 重新计算每个元素在数组中的位置
11                e.next = newTable[i]; //标记[1]
12                newTable[i] = e;      //将元素放在数组上
13                e = next;          //访问下一个Entry链上的元素
14            } while (e != null);
15        }
16    }
17 }

```

newTable[i]的引用赋给了e.next，也就是使用了单链表的头插入方式，同一位置上新元素总会被放在链表的头部位置；这样先放在一个索引上的元素终会被放到Entry链的尾部(如果发生了hash冲突的话)，这一点和Jdk1.8有区别，下文详解。在旧数组中同一条Entry链上的元素，通过重新计算索引位置后，有可能被放到了新数组的不同位置上。

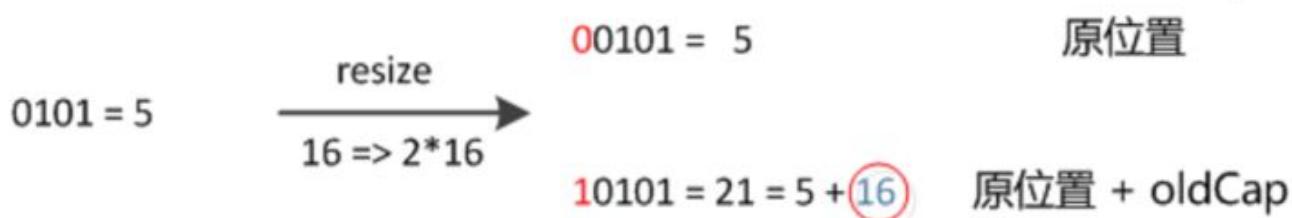
下面举个例子说明扩容过程。假设了我们的hash算法就是简单的用key mod 一下表的大小（也就是数组的长度）。其中的哈希桶数组table的size=2，所以key = 3、7、5，put顺序依次为 5、7、3。在mod 2以后都冲突在table[1]这里了。这里假设负载因子 loadFactor=1，即当键值对的实际大小size 大于 table的实际大小时进行扩容。接下来的三个步骤是哈希桶数组 resize成4，然后所有的Node重新rehash的过程。



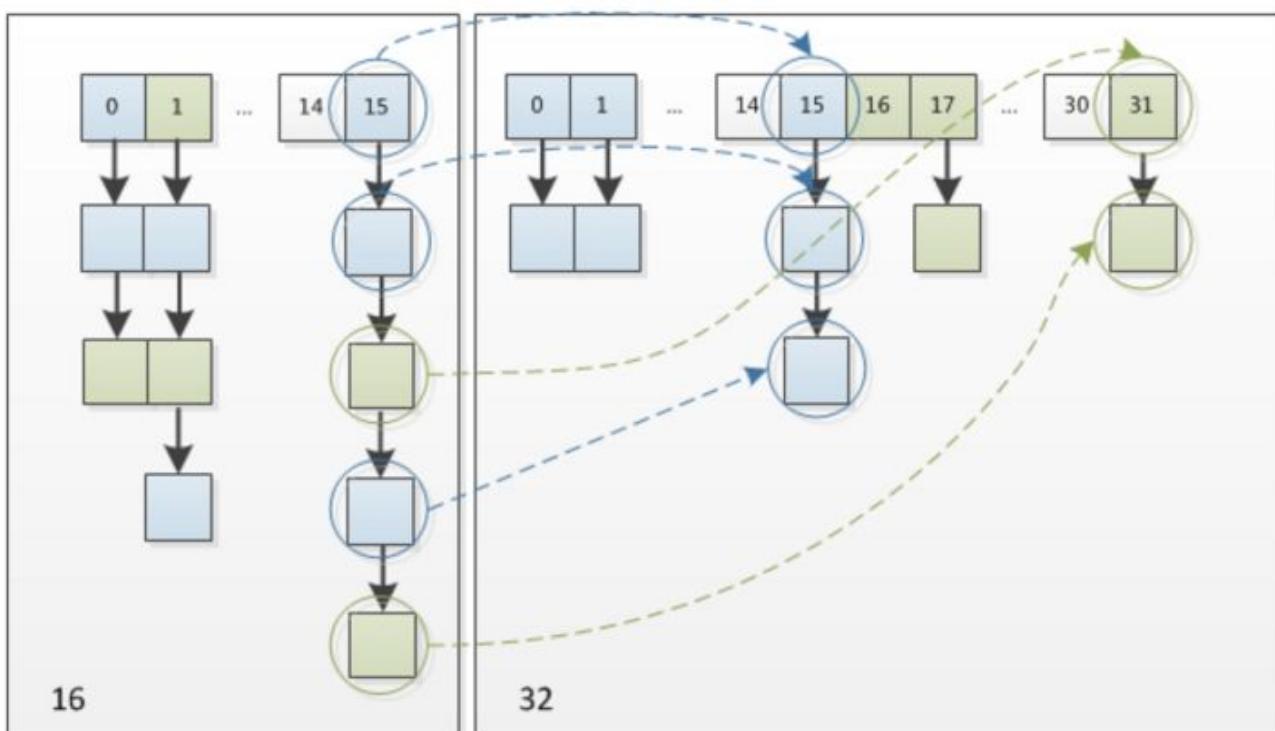
下面我们讲解下JDK1.8做了哪些优化。经过观测可以发现，我们使用的是2次幂的扩展(指长度扩为原来2倍)，所以，元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置。看下图可以明白这句话的意思， n 为table的长度，图(a)表示扩容前的key1和key2两种key确定索引位置的示例，图(b)表示扩容后key1和key2两种key确定索引位置的示例，其中hash1是key1对应的哈希与高位运算结果。

	$n - 1$	0000 0000 0000 0000 0000 0000 0000 1111	
(a)	key1(hash1)	1111 1111 1111 1111 0000 1111 0000 0101	→
	key2(hash2)	1111 1111 1111 1111 0000 1111 0001 0101	
			0000 0000 0000 0000 0000 0000 0000 0101
			0000 0000 0000 0000 0000 0000 0000 0101
(b)	key1(hash1)	0000 0000 0000 0000 0000 0000 0001 1111	→
	key2(hash2)	1111 1111 1111 1111 0000 1111 0000 0101	
			0000 0000 0000 0000 0000 0000 0000 0101
			0000 0000 0000 0000 0000 0000 0001 0101

元素在重新计算hash之后，因为 n 变为2倍，那么 $n-1$ 的mask范围在高位多1bit(红色)，因此新的index就会发生这样的变化：



因此，我们在扩充HashMap的时候，不需要像JDK1.7的实现那样重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”，可以看看下图为16扩充为32的resize示意图：



这个设计确实非常的巧妙，既省去了重新计算hash值的时间，而且同时，由于新增的1bit是0还是1可以认为是随机的，因此resize的过程，均匀的把之前的冲突的节点分散到新的bucket了。这一块就是JDK1.8新增的优化点。有一点注意区别，JDK1.7中rehash的时候，旧链表迁移新链表的时候，如果在新表的数组索引位置相同，则链表元素会倒置，但是从上图可以看出，JDK1.8不会倒置。有兴趣的同学可以研究下JDK1.8的resize源码，写的很赞，如下：

```
1 final Node<K,V>[] resize() {
2     Node<K,V>[] oldTab = table;
3     int oldCap = (oldTab == null) ? 0 : oldTab.length;
4     int oldThr = threshold;
5     int newCap, newThr = 0;
6     if (oldCap > 0) {
7         // 超过最大值就不再扩充了，就只好随你碰撞去吧
8         if (oldCap >= MAXIMUM_CAPACITY) {
9             threshold = Integer.MAX_VALUE;
10            return oldTab;
11        }
12        // 没超过最大值，就扩充为原来的2倍
13        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
14                  oldCap >= DEFAULT_INITIAL_CAPACITY)
15            newThr = oldThr << 1; // double threshold
16    }
17    else if (oldThr > 0) // initial capacity was placed in threshold
18        newCap = oldThr;
19    else {                // zero initial threshold signifies using defaults
20        newCap = DEFAULT_INITIAL_CAPACITY;
21        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
22    }
23    // 计算新的resize上限
24    if (newThr == 0) {
25
26        float ft = (float)newCap * loadFactor;
27        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
28                    (int)ft : Integer.MAX_VALUE);
29    }
30    threshold = newThr;
31    @SuppressWarnings({"rawtypes", "unchecked"})
32    Node<K,V>[] newTab = (Node<K,V>[])(new Node[newCap]);
33    table = newTab;
34    if (oldTab != null) {
35        // 把每个bucket都移动到新的buckets中
36        for (int j = 0; j < oldCap; ++j) {
37            Node<K,V> e;
38            if ((e = oldTab[j]) != null) {
39                oldTab[j] = null;
40                if (e.next == null)
41                    newTab[e.hash & (newCap - 1)] = e;
42                else if (e instanceof TreeNode)
43                    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
44                else { // 链表优化重hash的代码块
45                    Node<K,V> loHead = null, loTail = null;
46                    Node<K,V> hiHead = null, hiTail = null;
47                    Node<K,V> next;
48                    do {
```

```

49                     next = e.next;
50                     // 原索引
51                     if ((e.hash & oldCap) == 0) {
52                         if (loTail == null)
53                             loHead = e;
54                         else
55                             loTail.next = e;
56                         loTail = e;
57                     }
58                     // 原索引+oldCap
59                     else {
60                         if (hiTail == null)
61                             hiHead = e;
62                         else
63                             hiTail.next = e;
64                         hiTail = e;
65                     }
66                 } while ((e = next) != null);
67                 // 原索引放到bucket里
68                 if (loTail != null) {
69                     loTail.next = null;
70                     newTab[j] = loHead;
71                 }
72                 // 原索引+oldCap放到bucket里
73                 if (hiTail != null) {
74                     hiTail.next = null;
75                     newTab[j + oldCap] = hiHead;
76                 }
77             }
78         }
79     }
80 }
81 return newTab;
82 }

```

线程安全性

在多线程使用场景中，应该尽量避免使用线程不安全的HashMap，而使用线程安全的ConcurrentHashMap。那么为什么说HashMap是线程不安全的，下面举例子说明在并发的多线程使用场景中使用HashMap可能造成死循环。代码例子如下(便于理解，仍然使用JDK1.7的环境)：

```

public class HashMapInfiniteLoop {

    private static HashMap<Integer, String> map = new HashMap<Integer, String>(2, 0.75f);
    public static void main(String[] args) {
        map.put(5, "C");

        new Thread("Thread1") {
            public void run() {
                map.put(7, "B");
                System.out.println(map);
            };
        };
    }
}

```

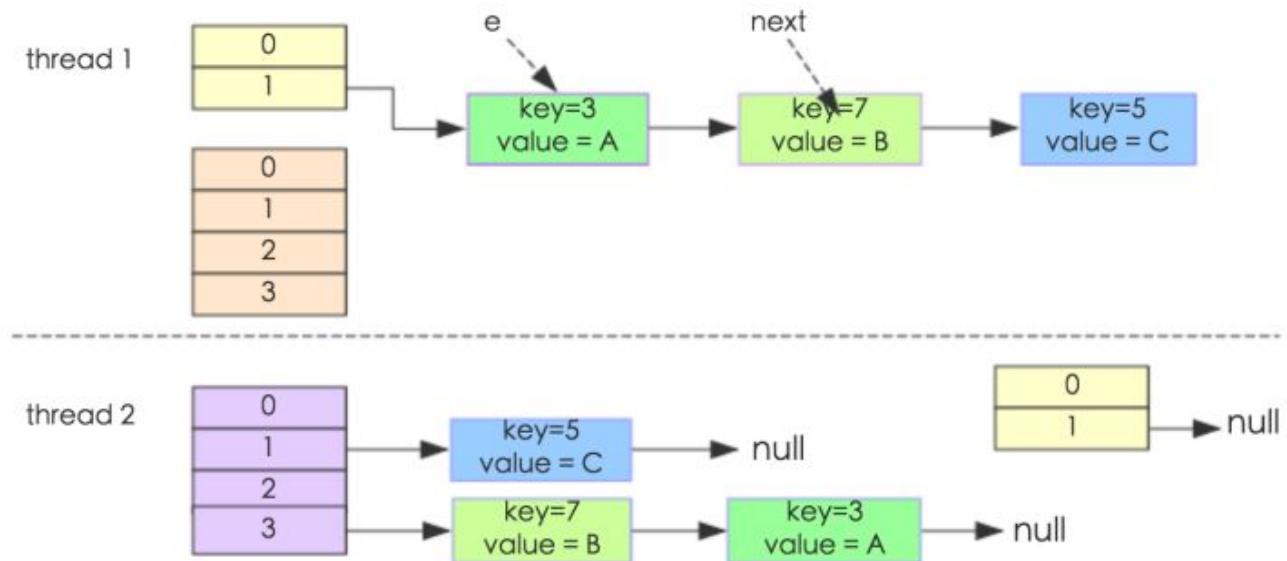
```

    }.start();
    new Thread("Thread2") {
        public void run() {
            map.put(3, "A");
            System.out.println(map);
        }
    }.start();
}
}

```

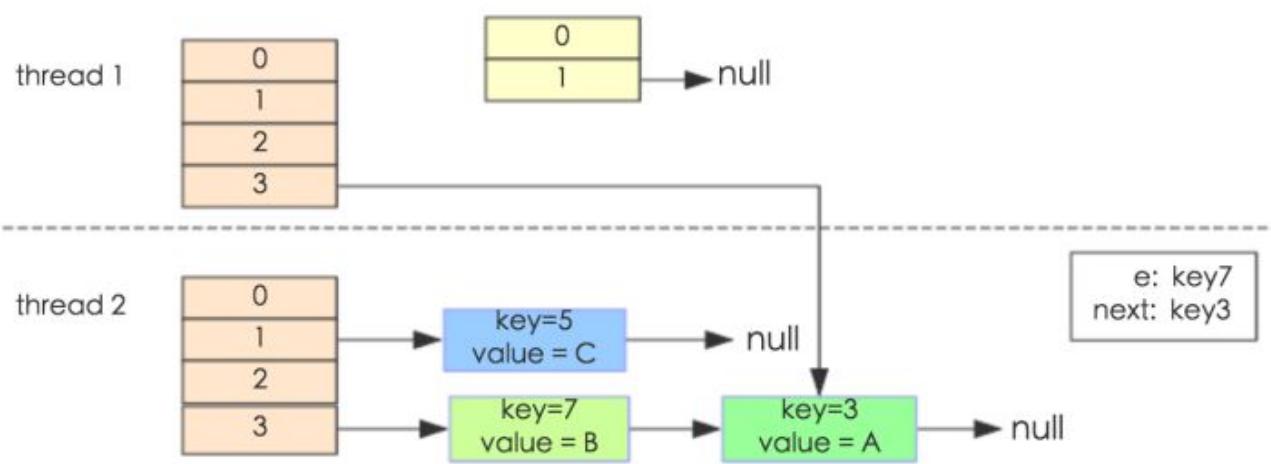
其中，map初始化为一个长度为2的数组，loadFactor=0.75，threshold=2*0.75=1，也就是说当put第二个key的时候，map就需要进行resize。

通过设置断点让线程1和线程2同时debug到transfer方法(3.3小节代码块)的首行。注意此时两个线程已经成功添加数据。放开thread1的断点至transfer方法的“Entry next = e.next;”这一行；然后放开线程2的的断点，让线程2进行resize。结果如下图。

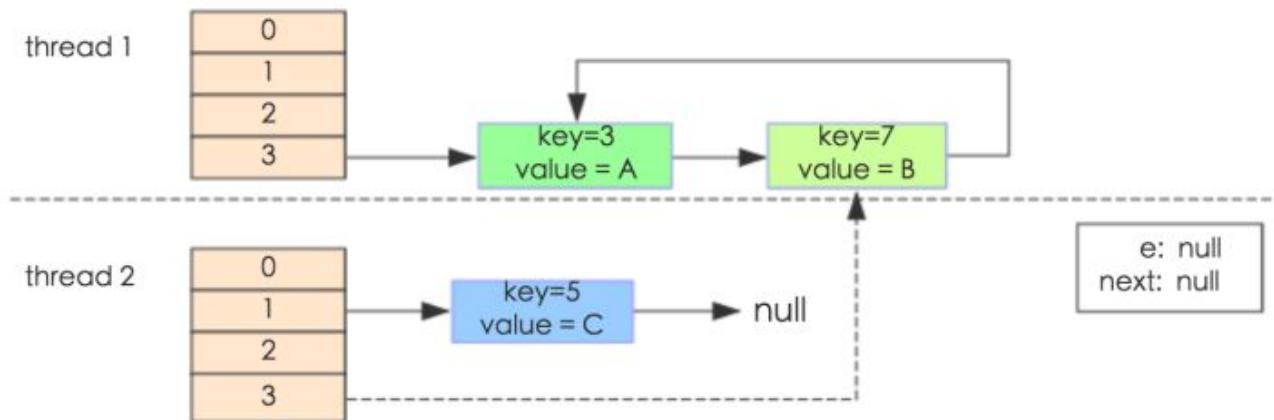


注意，Thread1的e指向了key(3)，而next指向了key(7)，其在线程二rehash后，指向了线程二重组后的链表。

线程一被调度回来执行，先是执行newTable[i] = e，然后是e = next，导致了e指向了key(7)，而下一次循环的next = e.next导致了next指向了key(3)。



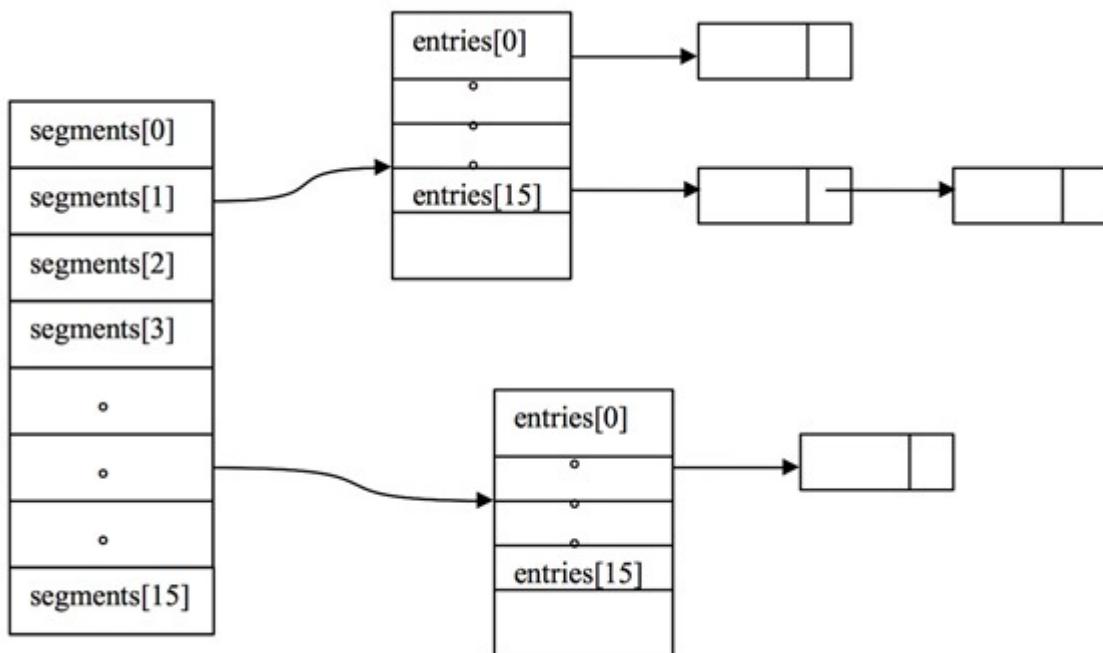
e.next = newTable[i] 导致 key(3).next 指向了 key(7)。注意：此时的key(7).next 已经指向了key(3)， 环形链表就这样出现了。



于是，当我们用线程一调用map.get(11)时，悲剧就出现了——Infinite Loop。

十一、 ConcurrentHashMap

锁分段技术



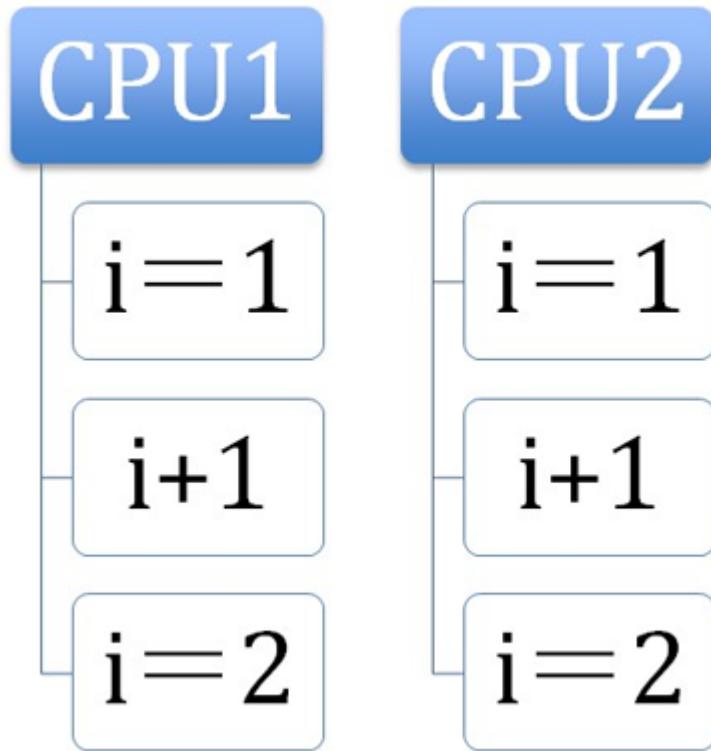
CAS无锁算法

实现方式

CAS:Compare and Swap, 翻译成比较并交换。

java.util.concurrent包中借助CAS实现了区别于synchronous同步锁的一种乐观锁。

CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。



```
public final boolean compareAndSet(int expect, int update) {
    return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
}
```

整体的过程就是这样子的，利用CPU的CAS指令，同时借助JNI来完成Java的非阻塞算法。其它原子操作都是利用类似的特性完成的。

其中

```
unsafe.compareAndSwapInt(this, valueOffset, expect, update);
```

类似：

```
if (this == expect) {

    this = update

    return true;

} else {

    return false;

}
```

那么问题就来了，成功过程中需要2个步骤：比较this == expect，替换this = update，compareAndSwapInt如何这两个步骤的原子性呢？

借助于字节码指令锁住缓存行或者锁住总线

存在的缺点

- 1. ABA问题。**因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A - B - A 就会变成1A-2B - 3A。

从Java1.5开始JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

关于ABA问题参考文档: <http://blog.hesey.net/2011/09/resolve-aba-by-atomicstampedreference.html>

- 2. 循环时间长开销大。**自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline），使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

- 3. 只能保证一个共享变量的原子操作。**当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁，或者有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如有两个共享变量i = 2, j = a，合并一下ij = 2a，然后用CAS来操作ij。从Java1.5开始JDK提供了AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行CAS操作。

十二、 ConcurrentLinkedQueue

延迟更新tail节点

1538048995966

延迟删除head节点

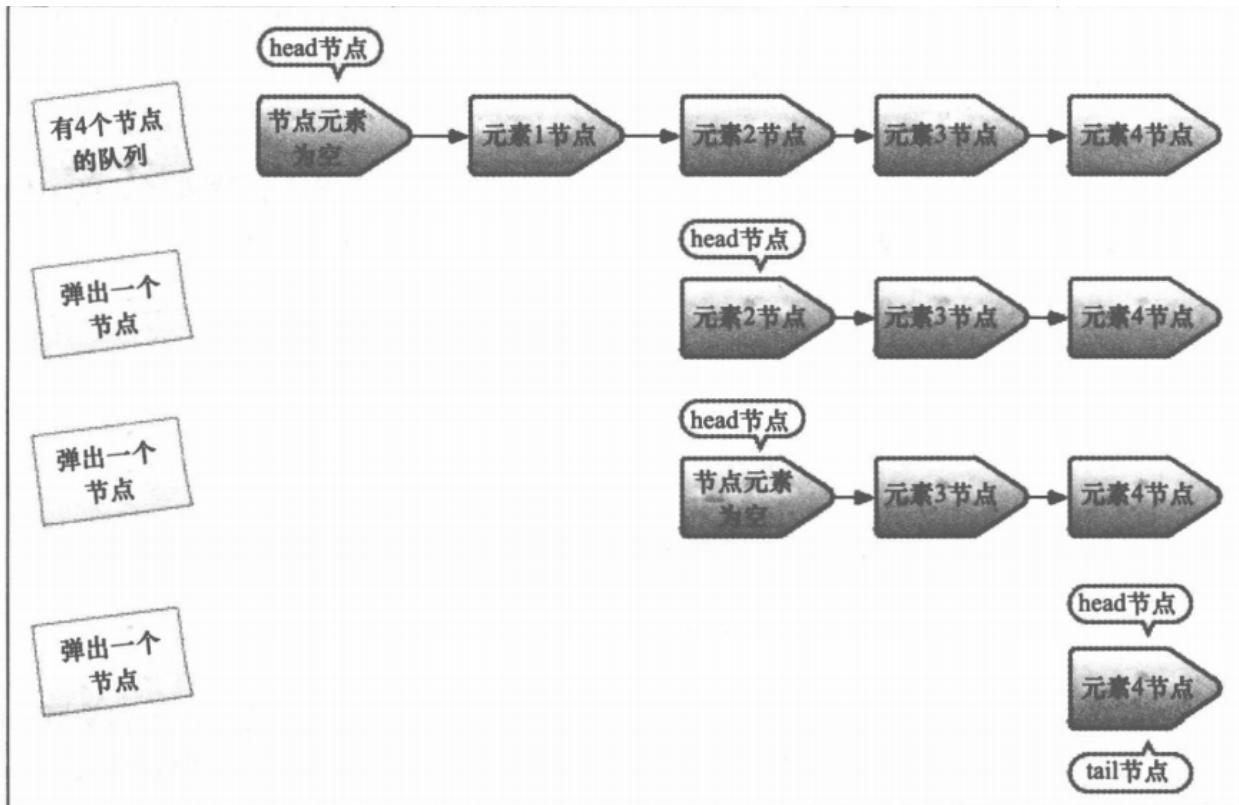


图 6-5 队列出节点快照图

十三、Topk问题

简述

在大规模数据处理中，经常会遇到的一类问题：在海量数据中找出出现频率最好的前k个数，或者从海量数据中找出最大的前k个数，这类问题通常被称为top K问题。例如，在搜索引擎中，统计搜索最热门的10个查询词；在歌曲库中统计下载最高的前10首歌等。

针对top K类问题，通常比较好的方案是分治+Trie树/hash+小顶堆（就是上面提到的最小堆），即先将数据集按照Hash方法分解成多个小数据集，然后使用Trie树活着Hash统计每个小数据集中的query词频，之后用小顶堆求出每个数据集中出现频率最高的前K个数，最后在所有top K中求出最终的top K。

eg：有1亿个浮点数，如果找出期中最大的10000个？

解决方案

最容易想到的方法是将数据全部排序，然后在排序后的集合中进行查找，最快的排序算法的时间复杂度一般为 $O(n \log n)$ ，如快速排序。但是在32位的机器上，每个float类型占4个字节，1亿个浮点数就要占用400MB的存储空间，对于一些可用内存小于400M的计算机而言，很显然是不能一次将全部数据读入内存进行排序的。其实即使内存能够满足要求（我机器内存都是8GB），该方法也并不高效，因为题目的目的是寻找出最大的10000个数即可，而排序却是将所有的元素都排序了，做了很多的无用功。

第二种方法为局部淘汰法，该方法与排序方法类似，用一个容器保存前10000个数，然后将剩余的所有数字一一与容器内的最小数字相比，如果所有后续的元素都比容器内的10000个数还小，那么容器内这个10000个数就是最大10000个数。如果某一后续元素比容器内最小数字大，则删掉容器内最小元素，并将该元素插入容器，最后遍历完这1亿个数，得到的结果容器中保存的数即为最终结果了。此时的时间复杂度为 $O(n+m^2)$ ，其中m为容器的大小，即10000。

第三种方法是分治法，将1亿个数据分成100份，每份100万个数据，找到每份数据中最大的10000个，最后在剩下的10010000个数据里面找出最大的10000个。如果100万数据选择足够理想，那么可以过滤掉1亿数据里面99%的数据。100万个数据里面查找最大的10000个数据的方法如下：用快速排序的方法，将数据分为2堆，如果大的那堆个数N大于10000个，继续对大堆快速排序一次分成2堆，如果大的那堆个数N大于10000个，继续对大堆快速排序一次分成2堆，如果大堆个数N小于10000个，就在小的那堆里面快速排序一次，找第10000-n大的数字；递归以上过程，就可以找到第1w大的数。参考上面的找出第1w大数字，就可以类似的方法找到前10000大数字了。此种方法需要每次的内存空间为 $10^{64}=4\text{MB}$ ，一共需要101次这样的比较。

第四种方法是Hash法。如果这1亿个数里面有很多重复的数，先通过Hash法，把这1亿个数字去重，这样如果重复率很高的话，会减少很大的内存用量，从而缩小运算空间，然后通过分治法或最小堆法查找最大的10000个数。

第五种方法采用最小堆。首先读入前10000个数来创建大小为10000的最小堆，建堆的时间复杂度为 $O(m\log m)$ (m 为数组的大小即为10000)，然后遍历后续的数字，并于堆顶（最小）数字进行比较。如果比最小的数小，则继续读取后续数字；如果比堆顶数字大，则替换堆顶元素并重新调整堆为最小堆。整个过程直至1亿个数全部遍历完为止。然后按照中序遍历的方式输出当前堆中的所有10000个数字。该算法的时间复杂度为 $O(nm\log m)$ ，空间复杂度是10000（常数）。

实际运行

实际上，最优的解决方案应该是最符合实际设计需求的方案，在时间应用中，可能有足够大的内存，那么直接将数据扔到内存中一次性处理即可，也可能机器有多个核，这样可以采用多线程处理整个数据集。

下面针对不同的应用场景，分析了适合相应应用场景的解决方案。

(1) 单机+单核+足够大内存

如果需要查找10亿个查询次（每个占8B）中出现频率最高的10个，考虑到每个查询词占8B，则10亿个查询次所需的内存大约是 $10^9 * 8\text{B} = 8\text{GB}$ 内存。如果有这么大内存，直接在内存中对查询次进行排序，顺序遍历找出10个出现频率最大的即可。这种方法简单快速，使用。然后，也可以先用HashMap求出每个词出现的频率，然后求出频率最大的10个词。

(2) 单机+多核+足够大内存

这时可以直接在内存总使用Hash方法将数据划分成n个partition，每个partition交给一个线程处理，线程的处理逻辑同（1）类似，最后一个线程将结果归并。

该方法存在一个瓶颈会明显影响效率，即数据倾斜。每个线程的处理速度可能不同，快的线程需要等待慢的线程，最终的处理速度取决于慢的线程。而针对此问题，解决的方法是，将数据划分成 $c \times n$ 个partition ($c > 1$)，每个线程处理完当前partition后主动取下一个partition继续处理，知道所有数据处理完毕，最后由一个线程进行归并。

(3) 单机+单核+受限内存

这种情况下，需要将原数据文件切割成一个一个小文件，如次啊用 $\text{hash}(x)\%M$ ，将原文件中的数据切割成M小文件，如果小文件仍大于内存大小，继续采用Hash的方法对数据文件进行分割，知道每个小文件小于内存大小，这样每个文件可放到内存中处理。采用（1）的方法依次处理每个小文件。

(4) 多机+受限内存

这种情况，为了合理利用多台机器的资源，可将数据分发到多台机器上，每台机器采用（3）中的策略解决本地的数据。可采用hash+socket方法进行数据分发。

从实际应用的角度考虑，(1) (2) (3) (4) 方案并不可行，因为在大规模数据处理环境下，作业效率并不是首要考虑的问题，算法的扩展性和容错性才是首要考虑的。算法应该具有良好的扩展性，以便数据量进一步加大（随着业务的发展，数据量加大是必然的）时，在不修改算法框架的前提下，可达到近似的线性比；算法应该具有容错性，即当前某个文件处理失败后，能自动将其交给另外一个线程继续处理，而不是从头开始处理。top K问题很适合采用MapReduce框架解决，用户只需编写一个Map函数和两个Reduce 函数，然后提交到Hadoop（采用Mapchain和Reducechain）上即可解决该问题。具体而言，就是首先根据数据值或者把数据hash(MD5)后的值按照范围划分到不同的机器上，最好可以让数据划分后一次读入内存，这样不同的机器负责处理不同的数值范围，实际上就是Map。得到结果后，各个机器只需拿出各自出现次数最多的前N个数据，然后汇总，选出所有的数据中出现次数最多的前N个数据，这实际上就是Reduce过程。对于Map函数，采用Hash算法，将Hash值相同的数据交给同一个Reduce task；对于第一个Reduce函数，采用HashMap统计出每个词出现的频率，对于第二个Reduce 函数，统计所有Reduce task，输出数据中的top K即可。直接将数据均分到不同的机器上进行处理是无法得到正确的结果的。因为一个数据可能被均分到不同的机器上，而另一个则可能完全聚集到一个机器上，同时还可能存在具有相同数目的数据。

经常被提及的该类问题

- (1) 有10000000个记录，这些查询串的重复度比较高，如果除去重复后，不超过3000000个。一个查询串的重复度越高，说明查询它的用户越多，也就是越热门。请统计最热门的10个查询串，要求使用的内存不能超过1GB。
- (2) 有10个文件，每个文件1GB，每个文件的每一行存放的都是用户的query，每个文件的query都可能重复。按照query的频度排序。
- (3) 有一个1GB大小的文件，里面的每一行是一个词，词的大小不超过16个字节，内存限制大小是1MB。返回频数最高的100个词。
- (4) 提取某日访问网站次数最多的那个IP。
- (5) 10亿个整数找出重复次数最多的100个整数。
- (6) 搜索的输入信息是一个字符串，统计300万条输入信息中最热门的前10条，每次输入的一个字符串为不超过255B，内存使用只有1GB。
- (7) 有1000万个身份证号以及他们对应的数据，身份证号可能重复，找出出现次数最多的身份证号。

重复问题

在海量数据中查找出重复出现的元素或者去除重复出现的元素也是常考的问题。针对此类问题，一般可以通过位图法实现。例如，已知某个文件内包含一些电话号码，每个号码为8位数字，统计不同号码的个数。本题最好的解决方法是通过使用位图法来实现。8位整数可以表示的最大十进制数值为99999999。如果每个数字对应于位图中一个bit位，那么存储8位整数大约需要99MB。因为 $1B=8bit$ ，所以99Mbit折合成内存为 $99/8=12.375MB$ 的内存，即可以只用12.375MB的内存表示所有的8位数电话号码的内容。

十四、资源池思想

作用

1. 降低资源消耗
2. 提高响应速度
3. 增强可管理性

线程池

与任务单元自己执行任务不同，线程池将任务单元和执行机制相分离，线程池中的线程为执行机制，提交给线程池的线程为任务单元，任务单元不自己执行任务，不创建线程，而是由线程池中的线程来执行任务

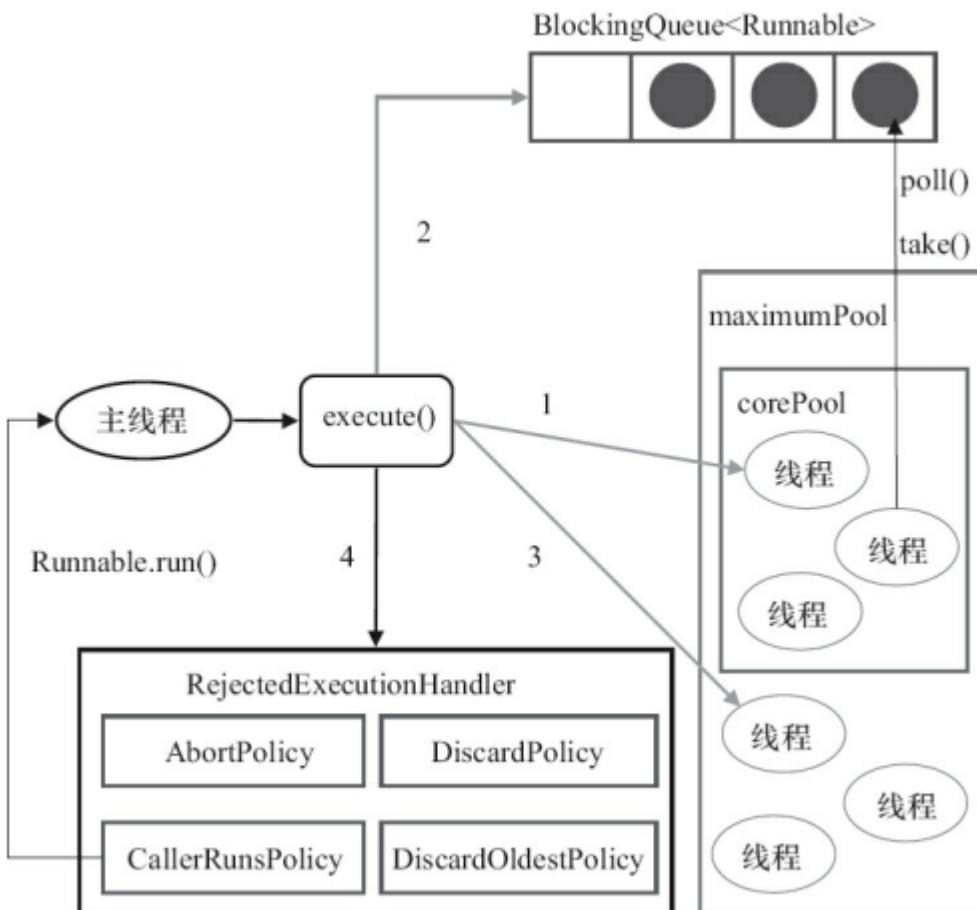


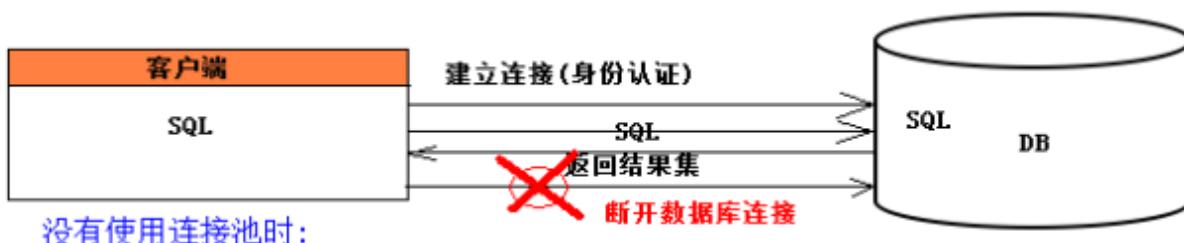
图9-2 ThreadPoolExecutor执行示意图

连接池

普通的JDBC数据库连接(Connection对象)使用 `DriverManager` 来获取，每次向数据库建立连接的时候都要将 `Connection` 加载到内存中，再验证用户名和密码(得花费0.05s ~ 1s的时间)，数据库的连接是比较昂贵的(创建的成本比较大)。

需要数据库连接的时候，就向数据库要求一个，执行完成后再断开连接。

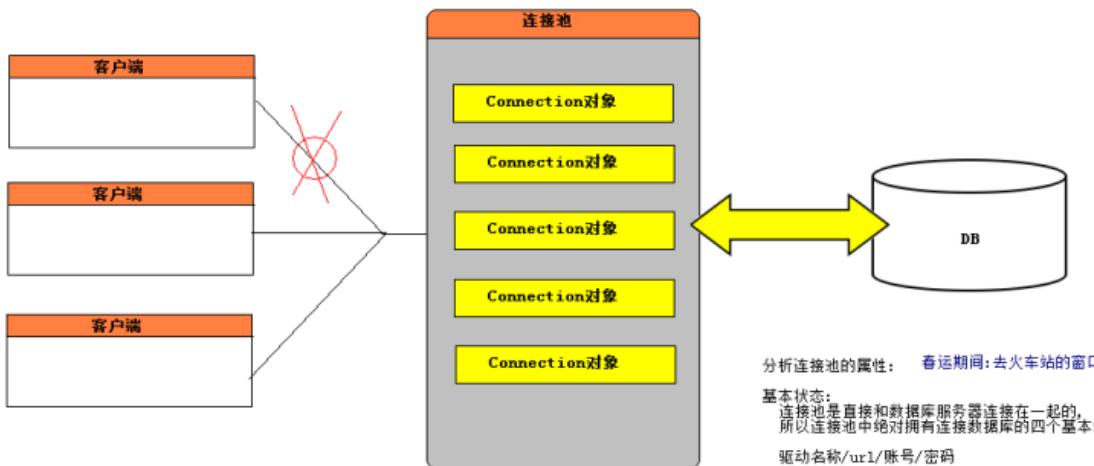
这样方式将会消耗大量的资源和时间。数据库的连接资源并没有得到很好的重复利用。若同时有几百人甚至几千人在线，频繁的进行数据库连接操作将占用很多的系统资源，严重的甚至会造成服务器的崩溃。



一个新浪首页：
会查询 10 个类别的新闻。---> 连接10次，发送10次。

$$\text{耗时} = 0.05\text{s} * 10 = 0.5\text{s};$$

考虑有100个人在新浪首页：
耗时 = $0.5\text{s} * 100 = 50\text{s};$



连接池的作用：

在于充分重复利用Connection对象.

分析连接池的属性： 春运期间：去火车站的窗口买票

基本状态：
连接池是直接和数据库服务器连接在一起的，
所以连接池中绝对拥有连接数据库的四个基本信息。

驱动名称/url/账号/密码

额外状态：

初始化连接数 = 5: 在连接池中实现先准备好5个Connection对象。

最大连接数 = 10: 在连接池中最多有10个Connection对象。
其他客户端，进入等待状态。

最小连接数 = 3: 在连接池中最少得存在3个Connection对象。

最大的等待时间 = 5min: 使用5分钟未申请获取Connection对象，如果时间到还没有申请到，则提示。自动放弃。

最大的连接超时时间 = 10min: 如果你在10分钟之内没有任何动作，则也认为是自动放弃Connection对象。

.....

十五、JVM内存管理算法

判断对象是否存活

引用计数法

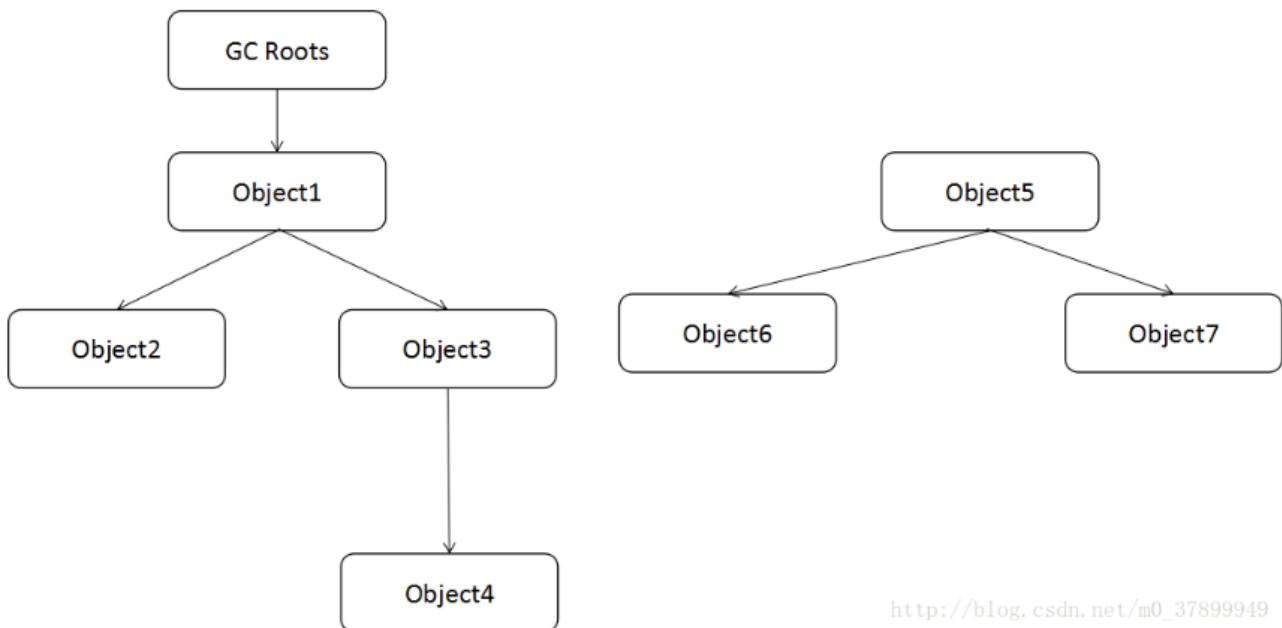
很多教科书判断对象是否存活的算法是这样：给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器为0的对象就是不可能再被使用的。

客观地说，引用计数算法 (Reference Counting)的实现简单，判定效率也很高，在大部分情况下它都是一个不错的算法。也有一些比较著名的应用案例，例如微软公司的COM(Component Object Model)技术、使用ActionScript 3的FlashPlayer、Python语言和在游戏脚本领域被广泛应用的Squirrel中都使用了引用计数算法进行内存管理。但是，至少主流的Java虚拟机里面没有选用引用计数算法来管理内存，其中最主要原因是它很难解决对象之间互循环引用的问题。



可达性分析算法

这个算法的基本思路就是通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到GC Roots没有任何引用链相连（用图论的话来说，就是从GC Roots到这个对象不可达）时，则证明此对象是不可用的。如下图所示，对象object 5、object 6、object 7虽然互相有关联，但是它们到GC Roots是不可达的，所以它们将被判定为是可回收的对象。



http://blog.csdn.net/m0_37899949

在Java中只有以下的对象才可以被作为GC Root.

- 1>虚拟机栈(栈帧中的本地方法表)中引用的对象。
- 2>方法区中类静态属性引用的对象。
- 3>方法区中常量引用的对象。
- 4>本地方法栈JNI (即一般说的Native方法) 的引用对象。

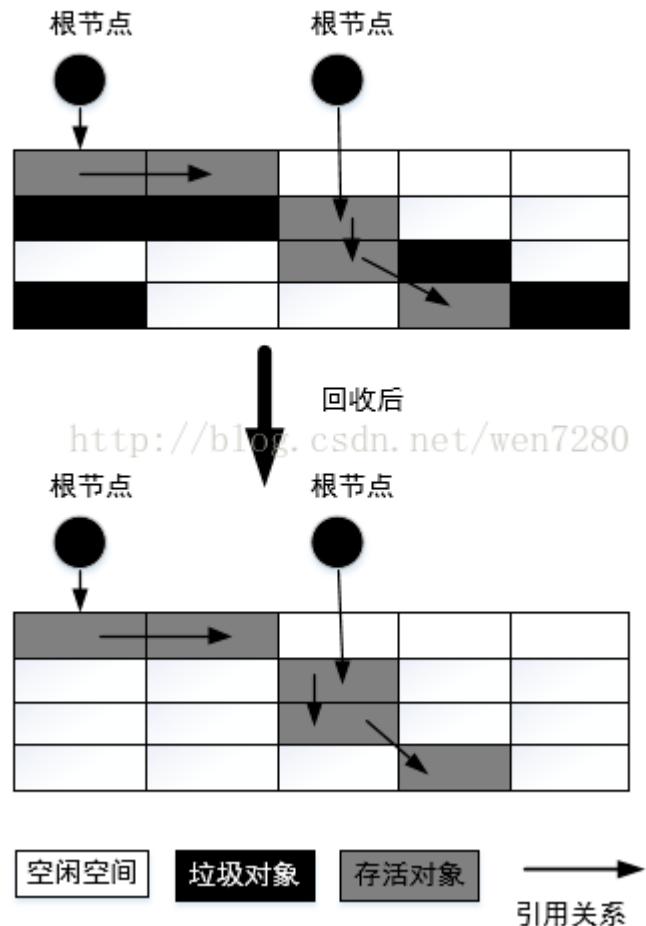
垃圾回收算法

JVM规范中并没有明确GC的运作方式，各个厂商可以采用不同的方式去实现垃圾回收器。这里讨论几种常见的GC算法。

标记-清除算法(Mark-Sweep)

最基础的垃圾回收算法，分为两个阶段，标注和清除。标记阶段标记出所有需要回收的对象，清除阶段回收被标记的对象所占用的空间。如图：

标记清除算法示意图

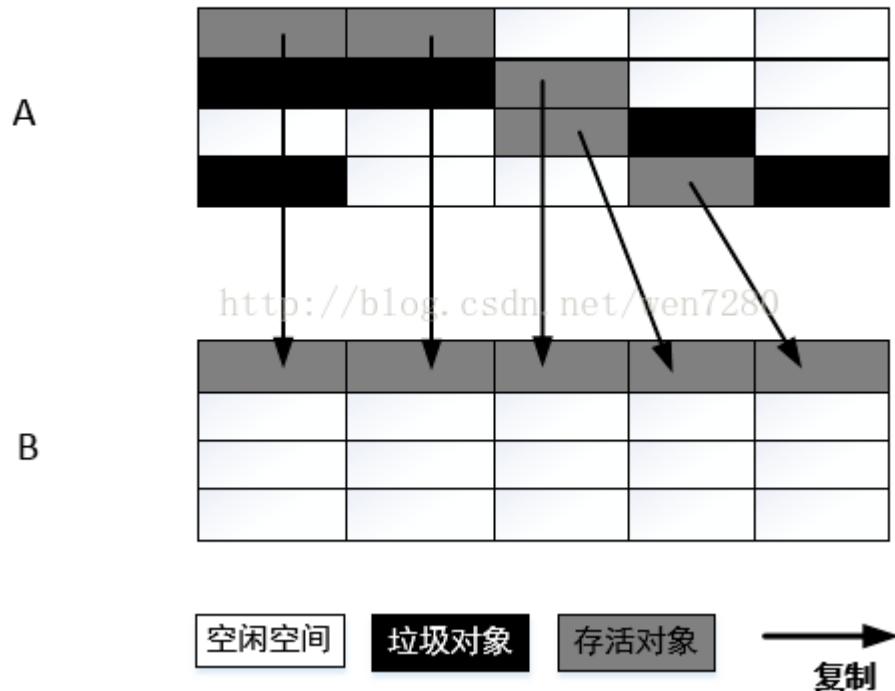


从图中我们就可以发现，该算法最大的问题是内存碎片化严重，后续可能发生大对象不能找到可利用空间的问题。

复制算法(Copying)

为了解决Mark-Sweep算法内存碎片化的缺陷而被提出的算法。按内存容量将内存划分为等大小的两块。每次只使用其中一块，当这一块内存满后将尚存活的对象复制到另一块上去，把已使用的内存清掉，如图：

复制算法工作示意图

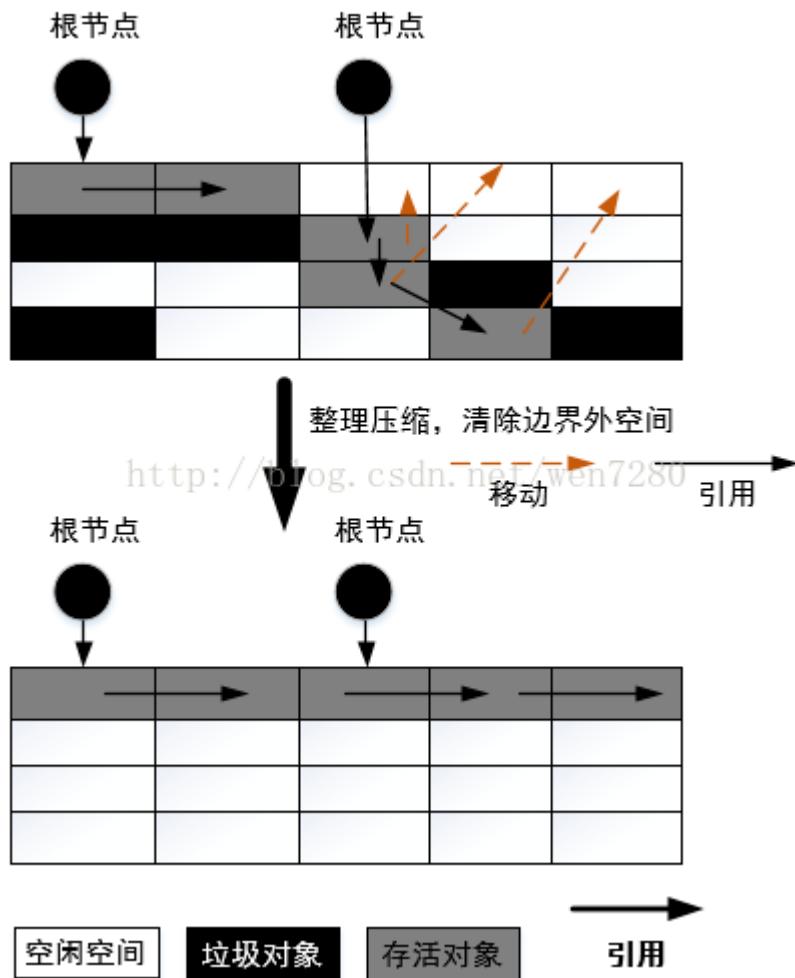


这种算法虽然实现简单，内存效率高，不易产生碎片，但是最大的问题是可用内存被压缩到了原本的一半。且存活对象增多的话，Copying算法的效率会大大降低。

标记-整理算法(Mark-Compact)

结合了以上两个算法，为了避免缺陷而提出。标记阶段和Mark-Sweep算法相同，标记后不是清理对象，而是将存活对象移向内存的一端。然后清除端边界外的对象。如图：

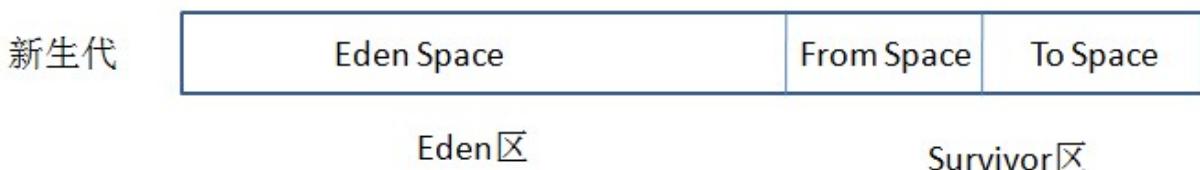
标记压缩算法示意图

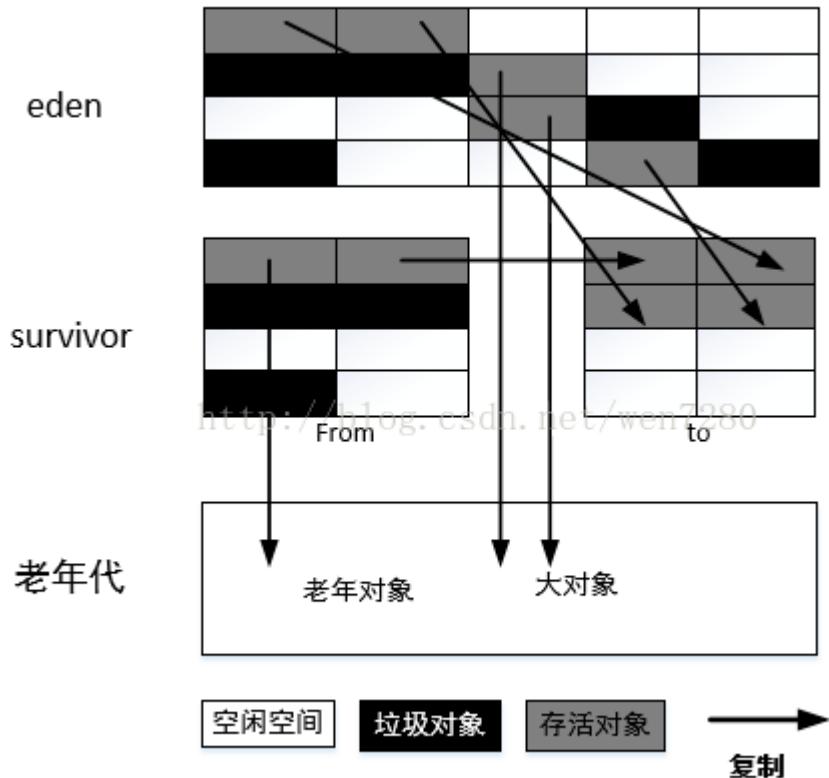


分代收集算法(Generational Collection)

分代收集法是目前大部分JVM所采用的方法，其核心思想是根据对象存活的不同生命周期将内存划分为不同的域，一般情况下将GC堆划分为老生代(Tenured/Old Generation)和新生代(Young Generation)。老生代的特点是每次垃圾回收时只有少量对象需要被回收，新生代的特点是每次垃圾回收时都有大量垃圾需要被回收，因此可以根据不同区域选择不同的算法。

目前大部分JVM的GC对于新生代都采取Copying算法，因为新生代中每次垃圾回收都要回收大部分对象，即要复制的操作比较少，但通常并不是按照1:1来划分新生代。一般将新生代划分为一块较大的Eden空间和两个较小的Survivor空间(From Space, To Space)，每次使用Eden空间和其中的一块Survivor空间，当进行回收时，将该两块空间中还存活的对象复制到另一块Survivor空间中。





而老生代因为每次只回收少量对象，因而采用Mark-Compact算法。

另外，不要忘记在[Java基础：Java虚拟机\(JVM\)](#)中提到过的处于方法区的永生代(Permanet Generation)。它用来存储class类，常量，方法描述等。对永生代的回收主要包括废弃常量和无用的类。

对象的内存分配主要在新生代的Eden Space和Survivor Space的From Space(Survivor目前存放对象的那一块)，少数情况会直接分配到老生代。当新生代的Eden Space和From Space空间不足时就会发生一次GC，进行GC后，Eden Space和From Space区的存活对象会被挪到To Space，然后将Eden Space和From Space进行清理。如果To Space无法足够存储某个对象，则将这个对象存储到老生代。在进行GC后，使用的便是Eden Space和To Space了，如此反复循环。当对象在Survivor区躲过一次GC后，其年龄就会+1。默认情况下年龄到达15的对象会被移到老生代中。

十六、容器虚拟化技术，Docker思想

为什么会有docker

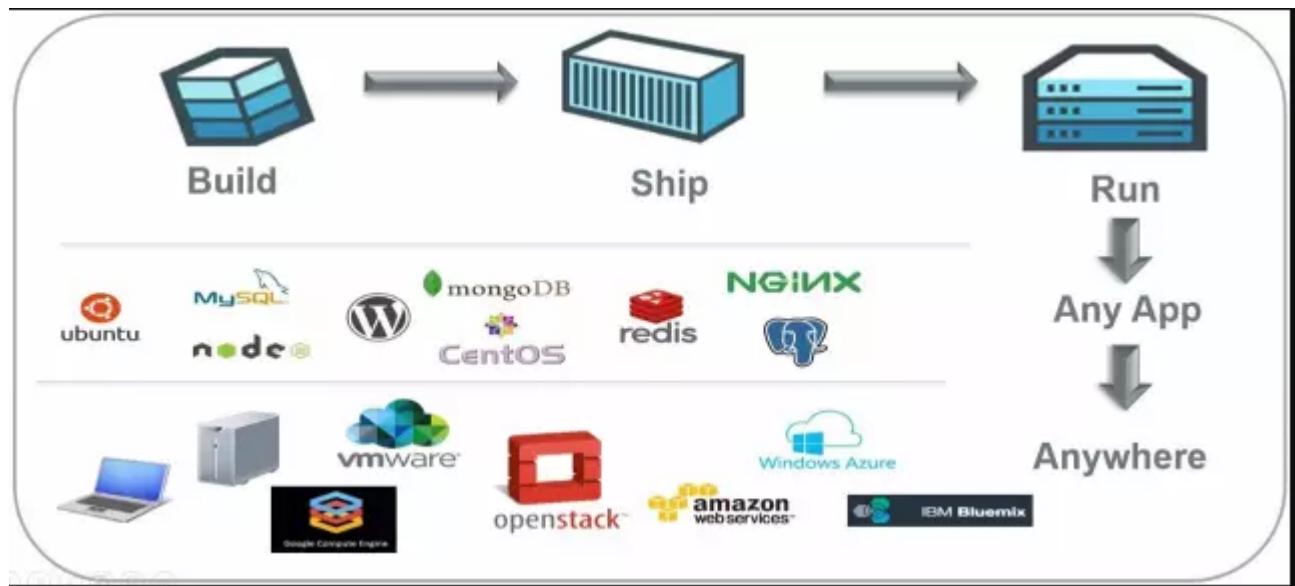
一款产品从开发到上线，从操作系统，到运行环境，再到应用配置。作为开发+运维之间的协作我们需要关心很多东西，这也是很多互联网公司都不得不面对的问题，特别是各种版本的迭代之后，不同版本环境的兼容，对运维人员都是考验 Docker之所以发展如此迅速，也是因为它对此给出了一个标准化的解决方案。环境配置如此麻烦，换一台机器，就要重来一次，费力费时。很多人想到，能不能从根本上解决问题，软件可以带环境安装？也就是说，安装的时候，把原始环境一模一样地复制过来。开发人员利用 Docker 可以消除协作编码时“在我的机器上可正常工作”的问题。

之前在服务器配置一个应用的运行环境，要安装各种软件，就拿尚硅谷电商项目的环境来说吧，Java/Tomcat/MySQL/JDBC驱动包等。安装和配置这些东西有多麻烦就不说了，它还不能跨平台。假如我们是在Windows 上安装的这些环境，到了 Linux 又得重新装。况且就算不跨操作系统，换另一台同样操作系统的服务器，要移植应用也是非常麻烦的。

传统上认为，软件编码开发/测试结束后，所产出的成果即是程序或是能够编译执行的二进制字节码等(java为例)。而为了让这些程序可以顺利执行，开发团队也得准备完整的部署文件，让运维团队得以部署应用程式，开发需要清楚的告诉运维部署团队，用的全部配置文件+所有软件环境。不过，即便如此，仍然常常发生部署失败的状况。Docker镜像的设计，使得Docker得以打破过去「程序即应用」的观念。透过镜像(images)将作业系统核心除外，运作应用程式所需要的系统环境，由下而上打包，达到应用程式跨平台间的无缝接轨运作。

docker理念

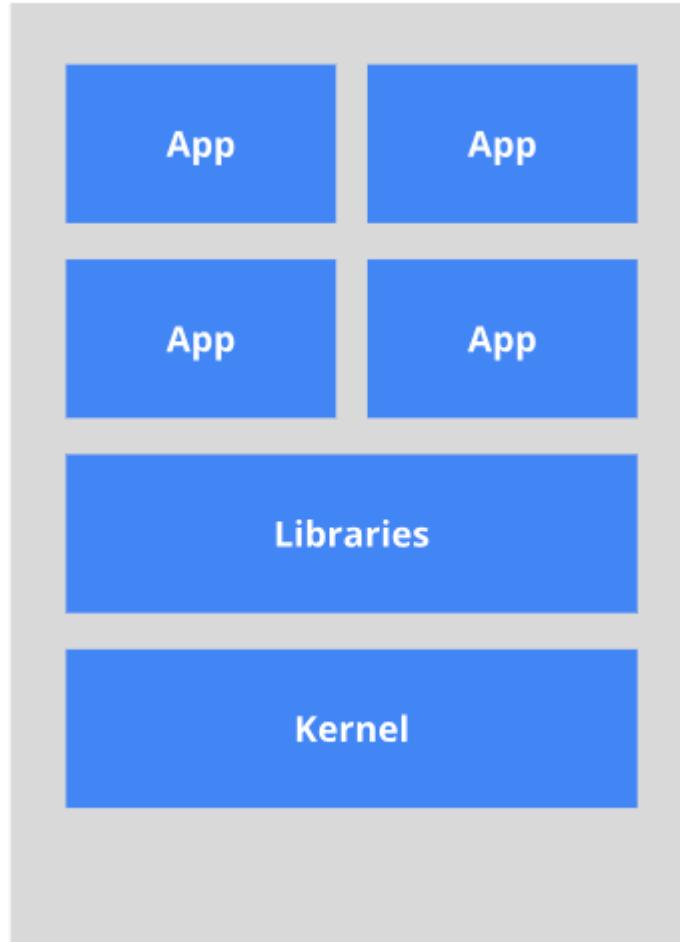
Docker是基于Go语言实现的云开源项目。 Docker的主要目标是“Build, Ship and Run Any App, Anywhere”，也就是通过对应用组件的封装、分发、部署、运行等生命周期的管理，使用户的APP（可以是一个WEB应用或数据库应用等等）及其运行环境能够做到“一次封装，到处运行”。



Linux 容器技术的出现就解决了这样一个问题，而 Docker 就是在它的基础上发展过来的。将应用运行在 Docker 容器上面，而 Docker 容器在任何操作系统上都是一致的，这就实现了跨平台、跨服务器。只需要一次配置好环境，换到别的机子上就可以一键部署好，大大简化了操作

实现方式

虚拟机 (virtual machine) 就是带环境安装的一种解决方案。它可以在一种操作系统里面运行另一种操作系统，比如在Windows 系统里面运行Linux 系统。应用程序对此毫无感知，因为虚拟机看上去跟真实系统一模一样，而对于底层系统来说，虚拟机就是一个普通文件，不需要了就删掉，对其他部分毫无影响。这类虚拟机完美的运行了另一套系统，能够使应用程序，操作系统和硬件三者之间的逻辑不变。



虚拟机的缺点： 1 资源占用多 2 冗余步骤多 3 启动慢

由于前面虚拟机存在这些缺点，Linux 发展出了另一种虚拟化技术：Linux 容器（Linux Containers，缩写为 LXC）。**Linux 容器不是模拟一个完整的操作系统**，而是对进程进行隔离。有了容器，就可以将软件运行所需的所有资源打包到一个隔离的容器中。容器与虚拟机不同，不需要捆绑一整套操作系统，只需要软件工作所需的库资源和设置。系统因此而变得高效轻量并保证部署在任何环境中的软件都能始终如一地运行。

比较了 Docker 和传统虚拟化方式的不同之处：

- 传统虚拟机技术是虚拟出一套硬件后，在其上运行一个完整操作系统，在该系统上再运行所需应用进程；
- 而容器内的应用进程直接运行于宿主的内核，容器内没有自己的内核，而且也没有进行硬件虚拟。因此容器要比传统虚拟机更为轻便。
- 每个容器之间互相隔离，每个容器有自己的文件系统，容器之间进程不会相互影响，能区分计算资源。

Why Docker

- 更轻量：基于容器的虚拟化，仅包含业务运行所需的runtime环境，CentOS/Ubuntu基础镜像仅170M；宿主机可部署100~1000个容器
- 更高效：无操作系统虚拟化开销
 - ◆ 计算：轻量，无额外开销
 - ◆ 存储：系统盘aufs/dm/overlayfs；数据盘volume
 - ◆ 网络：宿主机网络，NS隔离
- 更敏捷、更灵活：
 - ◆ 分层的存储和包管理，devops理念
 - ◆ 支持多种网络配置

美团云

docker的组成

镜像

Docker 镜像 (Image) 就是一个只读的模板。镜像可以用来创建 Docker 容器，一个镜像可以创建很多容器。

容器与镜像的关系类似于面向对象编程中的对象与类。

Docker	面向对象
容器	对象
镜像	类

容器

Docker 利用容器 (Container) 独立运行的一个或一组应用。容器是用镜像创建的运行实例。

它可以被启动、开始、停止、删除。每个容器都是相互隔离的、保证安全的平台。

可以把容器看做是一个简易版的 Linux 环境（包括root用户权限、进程空间、用户空间和网络空间等）和运行在其中的应用程序。

容器的定义和镜像几乎一模一样，也是一堆层的统一视角，唯一区别在于容器的最上面那一层是可读可写的。

仓库

仓库 (Repository) 是集中存放镜像文件的场所。仓库(Repository)和仓库注册服务器 (Registry) 是有区别的。仓库注册服务器上往往存放着多个仓库，每个仓库中又包含了多个镜像，每个镜像有不同的标签 (tag) 。

仓库分为公开仓库 (Public) 和私有仓库 (Private) 两种形式。最大的公开仓库是 Docker Hub(<https://hub.docker.com/>)，存放了数量庞大的镜像供用户下载。国内的公开仓库包括阿里云、网易云等

总结

需要正确的理解仓储/镜像/容器这几个概念：

Docker 本身是一个容器运行载体或称之为管理引擎。我们把应用程序和配置依赖打包好形成一个可交付的运行环境，这个打包好的运行环境就似乎 image 镜像文件。只有通过这个镜像文件才能生成 Docker 容器。image 文件可以看作是容器的模板。Docker 根据 image 文件生成容器的实例。同一个 image 文件，可以生成多个同时运行的容器实例。

- image 文件生成的容器实例，本身也是一个文件，称为镜像文件。
- 一个容器运行一种服务，当我们需要的时候，就可以通过 docker 客户端创建一个对应的运行实例，也就是我们的容器
- 至于仓储，就是放了一堆镜像的地方，我们可以把镜像发布到仓储中，需要的时候从仓储中拉下来就可以了。

十七、持续集成、持续发布，jenkins

持续集成

好处 1：降低风险 一天中进行多次的集成，并做了相应的测试，这样有利于检查缺陷，了解软件的健康状况，减少假定。

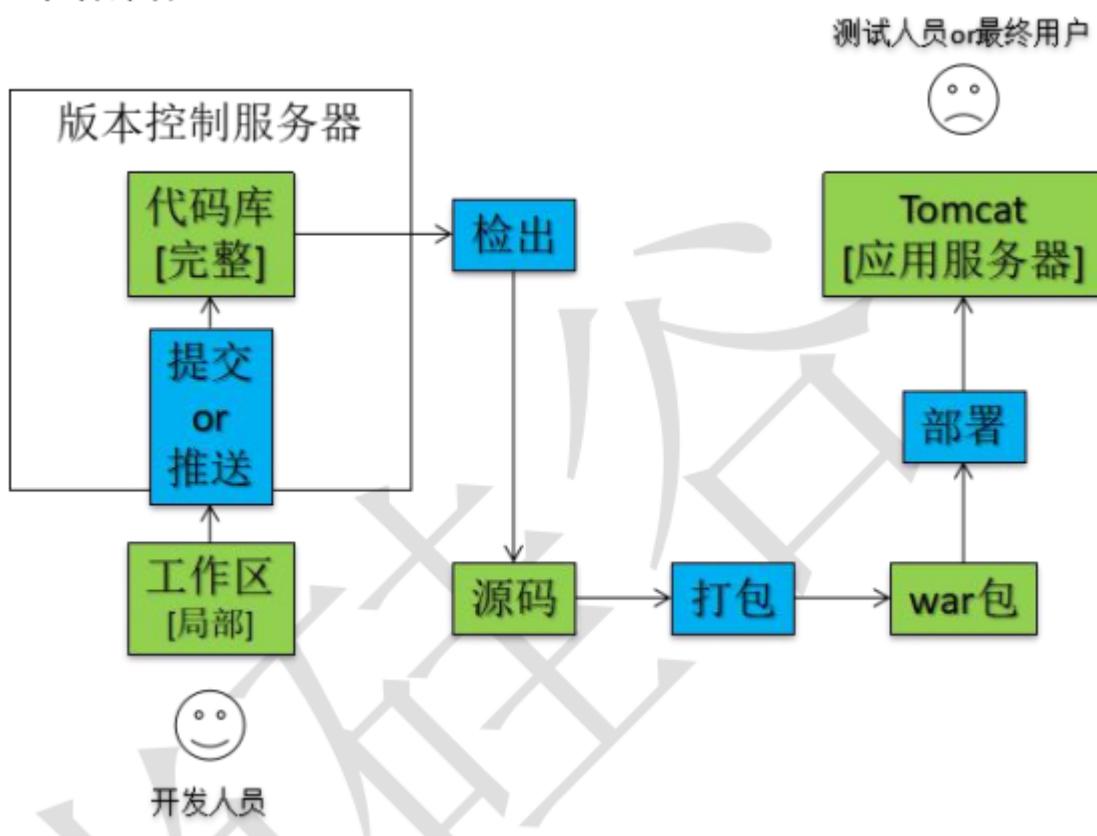
好处 2：减少重复过程 产生重复过程有两个方面的原因，一个是编译、测试、打包、部署等等固定操作都必须要做，无法省略任何一个环节；另一个是一个缺陷如果没有及时发现，有可能导致后续代码的开发方向是错误的，要修复问题需要重新编写受影响的所有代码。而使用 Jenkins 等持续集成工具既可以构建环节从手动完成转换为自动化完成，又可以通过增加集成频次尽早发现缺陷避免方向性错误。

好处 3：任何时间、任何地点生成可部署的软件 持续集成可以让您在任何时间发布可以部署的软件。从外界来看，这是持续集成最明显的好处，我们可以对改进软件品质和减少风险说起来滔滔不绝，但对于客户来说，可以部署的软件产品是最实际的资产。利用持续集成，您可以经常对源代码进行一些小改动，并将这些改动和其他的代码进行集成。如果出现问题，项目成员马上就会被通知到，问题会第一时间被修复。不采用持续集成的情况下，这些问题有可能到交付前的集成测试的时候才发现，有可能会导致延迟发布产品，而在急于修复这些缺陷的时候又有可能引入新的缺陷，最终可能导致项目失败。

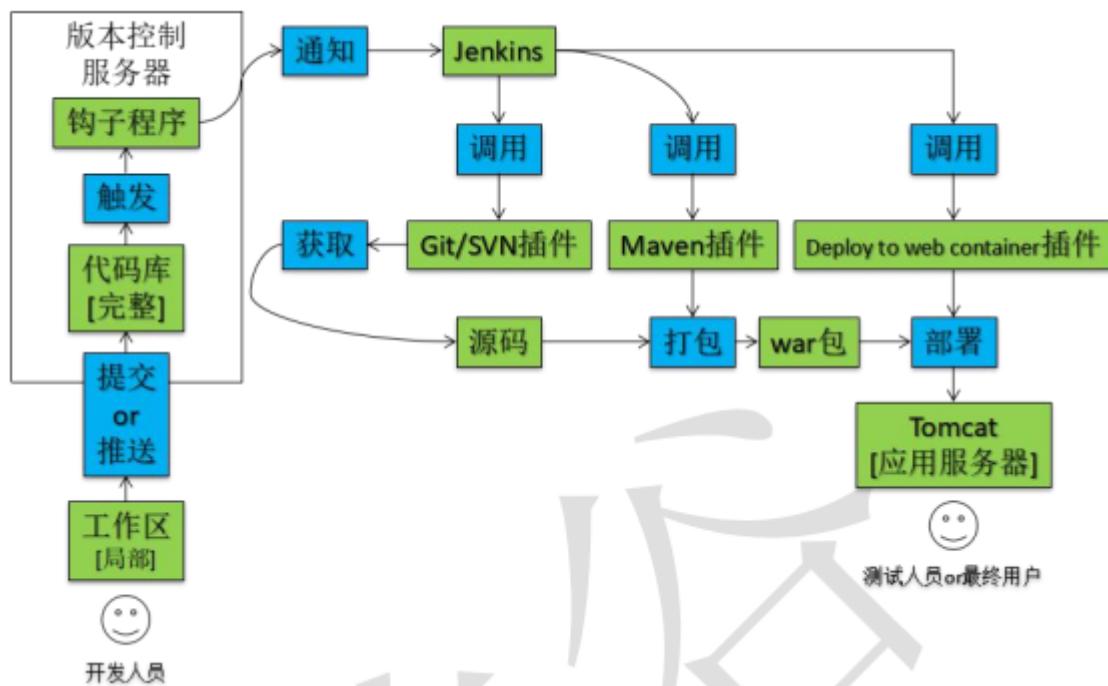
好处 4：增强项目的可见性 持续集成让我们能够注意到趋势并进行有效的决策。如果没有真实或最新的数据提供支持，项目就会遇到麻烦，每个人都会提出他最好的猜测。通常，项目成员通过手工收集这些信息，增加了负担，也很耗时。持续集成可以带来两点积极效果：(1)有效决策：持续集成系统为项目构建状态和品质指标提供了及时的信息，有些持续集成系统可以报告功能完成度和缺陷率。(2)注意到趋势：由于经常集成，我们可以看到一些趋势，如构建成功或失败、总体品质以及其它的项目信息。

好处 5：建立团队对开发产品的信心 持续集成可以建立开发团队对开发产品的信心，因为他们清楚的知道每一次构建的结果，他们知道他们对软件的改动造成了哪些影响，结果怎么样。

手动部署



自动部署



面试题举例

一、设计一个分布式环境下全局唯一的发号器

1、UUID

常见的方式。可以利用数据库也可以利用程序生成，一般来说全球唯一。

优点：

1. 简单，代码方便。
2. 生成ID性能非常好，基本不会有性能问题。
3. 全球唯一，在遇见数据迁移，系统数据合并，或者数据库变更等情况下，可以从容应对。

缺点：

1. 没有排序，无法保证趋势递增。
2. UUID往往是使用字符串存储，查询的效率比较低。
3. 存储空间比较大，如果是海量数据库，就需要考虑存储量的问题。
4. 传输数据量大
5. 不可读。

2、数据库自增长序列或字段

最常见的方式。利用数据库，全数据库唯一。

优点：

1. 简单，代码方便，性能可以接受。
2. 数字ID天然排序，对分页或者需要排序的结果很有帮助。

缺点：

1. 不同数据库语法和实现不同，数据库迁移的时候或多数据库版本支持的时候需要处理。
2. 在单个数据库或读写分离或一主多从的情况下，只有一个主库可以生成。有单点故障的风险。
3. 在性能达不到要求的情况下，比较难于扩展。
4. 如果遇见多个系统需要合并或者涉及到数据迁移会相当痛苦。
5. 分表分库的时候会有麻烦。

优化方案：

针对主库单点，如果有多个Master库，则每个Master库设置的起始数字不一样，步长一样，可以是Master的个数。比如：Master1生成的是1, 4, 7, 10, Master2生成的是2,5,8,11 Master3生成的是3,6,9,12。这样就可以有效生成集群中的唯一ID，也可以大大降低ID生成数据库操作的负载。

3、数据库sequence表以及乐观锁

我们可以单独设置一张表，来存储所有表的下一个主键的值，例如现在有A、B、C三个表，sequence表结构如下

表名 (name)	下一个主键(id)
A	10
B	100
C	500

然后，每当需要获取下一个主键值的时候，首先使用select语句获取主键，然后使用数据库的乐观锁机制去update这个sequence表，更新成功则说明获取主键成功，更新失败则说明存在并发，当前主键被别的机器抢走了，需要重新select出新的主键，再update。例如要获取表B的下一个主键，需要发送sql

```
select id from sequence where name=B  
//获得id=100,更新sequence表  
update sequence set id=id+1 where name=B and id=100
```

优点：

1. 操作简单，使用乐观锁可以提高性能
2. 生成的id有序递增，连续
3. 可适用于分布式环境，可以进行分库分表

缺点

1. 需要单独设置一张表，浪费存储空间
2. 数据库更新比较频繁，写压力太大

改进方案

可以将每次获取一个主键，改为每次获取500个或者更多，然后缓存再当前机器中，用完这500个后，再去请求数据库，做更新操作，可以减少数据库的读写压力，但是会造成主键的不连续

4、Redis生成ID

当使用数据库来生成ID性能不够要求的时候，我们可以尝试使用Redis来生成ID。这主要依赖于Redis是单线程的，所以也可以用生成全局唯一的ID。可以用Redis的原子操作INCR和INCRBY来实现。

可以使用Redis集群来获取更高的吞吐量。假如一个集群中有5台Redis。可以初始化每台Redis的值分别是1,2,3,4,5，然后步长都是5。各个Redis生成的ID为：

A: 1,6,11,16,21 B: 2,7,12,17,22 C: 3,8,13,18,23 D: 4,9,14,19,24 E: 5,10,15,20,25

这个，随便负载到哪个机确定好，未来很难做修改。但是3-5台服务器基本能够满足器上，都可以获得不同的ID。但是步长和初始值一定需要事先需要了。使用Redis集群也可以方式单点故障的问题。

另外，比较适合使用Redis来生成每天从0开始的流水号。比如订单号=日期+当日自增长号。可以在Redis中生成一个Key，使用INCR进行累加。

优点：

1. 不依赖于数据库，灵活方便，且性能优于数据库。
2. 数字ID天然排序，对分页或者需要排序的结果很有帮助。

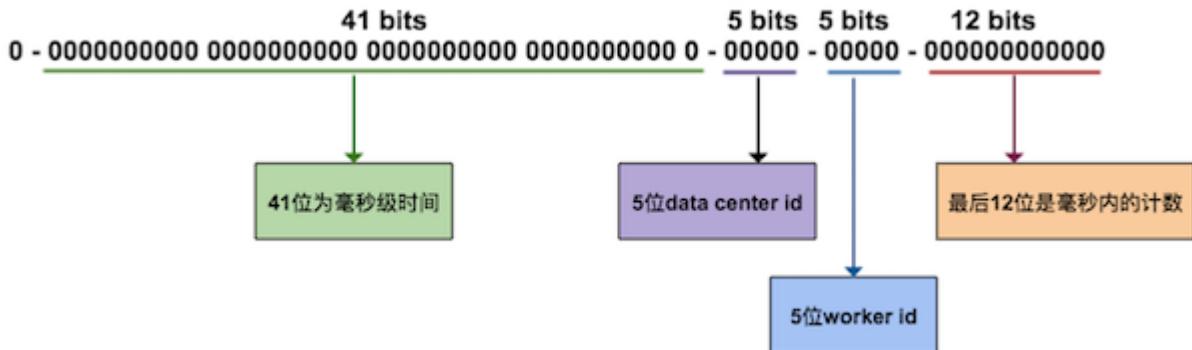
缺点：

1. 如果系统中没有Redis，还需要引入新的组件，增加系统复杂度。
2. 需要编码和配置的工作量比较大。

5、Twitter的snowflake算法

snowflake 是 twitter 开源的分布式ID生成算法，其核心思想为，一个long型的ID：

- 41 bit 作为毫秒数 - **41位的长度可以使用69年**
- 10 bit 作为机器编号 (5个bit是数据中心, 5个bit的机器ID) - **10位的长度最多支持部署1024个节点**
- 12 bit 作为毫秒内序列号 - **12位的计数顺序号支持每个节点每毫秒产生4096个ID序号**



Snowflake图示

算法单机每秒内理论上最多可以生成 $1000 * (2^{12})$, 也就是400W的ID, 完全能满足业务的需求。

snowflake算法可以根据自身项目的需要进行一定的修改。比如估算未来的数据中心个数, 每个数据中心的机器数以及统一毫秒可以能的并发数来调整在算法中所需要的bit数。

优点:

1. 不依赖于数据库, 灵活方便, 且性能优于数据库。
2. ID按照时间在单机上是递增的。

缺点:

在单机上是递增的, 但是由于涉及到分布式环境, 每台机器上的时钟不可能完全同步, 也许有时候也会出现不是全局递增的情况。

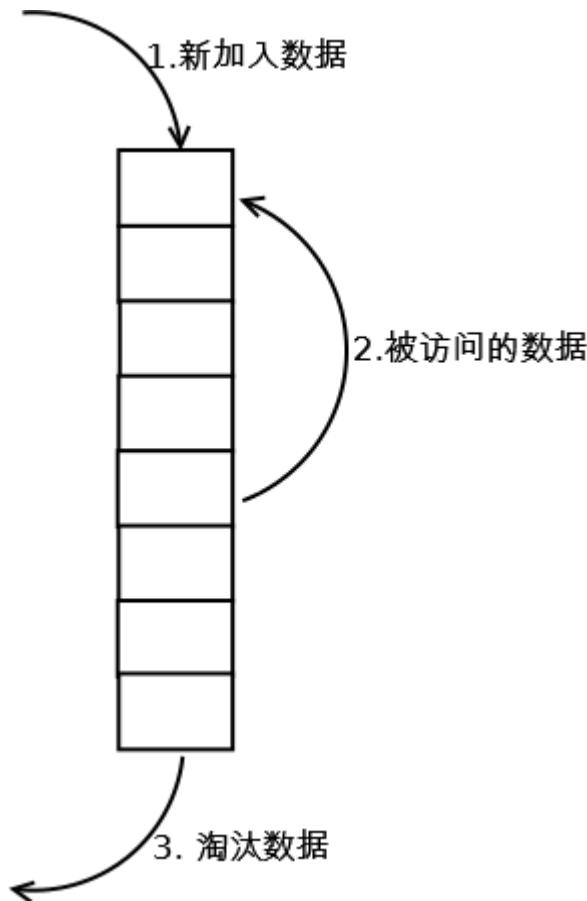
二、设计一个带有过期时间的LRU缓存

问题描述

如何设计实现LRU缓存? 且Set() 和 Get() 的复杂度为O(1)。

问题分析

LRU, 全称Least Recently Used, 最近最少使用缓存。



LRU算法的设计原则

如果一个数据在最近一段时间没有被访问到，那么在将来它被访问的可能性也很小。也就是说，当限定的空间已存满数据时，应当把最久没有被访问到的数据淘汰。

而用什么数据结构来实现LRU算法呢？

可能大多数人都会想到：用一个数组来存储数据，给每一个数据项标记一个访问时间戳，每次插入新数据项的时候，先把数组中存在的数据项的时间戳自增，并将新数据项的时间戳置为0并插入到数组中。每次访问数组中的数据项的时候，将被访问的数据项的时间戳置为0。当数组空间已满时，将时间戳最大的数据项淘汰。这种实现思路很简单，但是有什么缺陷呢？需要不停地维护数据项的访问时间戳，另外，在插入数据、删除数据以及访问数据时，时间复杂度都是 $O(n)$ ，数组的缺陷凸显无疑。

那么有没有更好的实现办法呢？

那就是利用链表和HashMap。当需要插入新数据项，在链表中

命中，则把该节点移到链表头部；不存在，则新建一个节点，放在链表头部。若缓存满，则把链表最后一个节点删除即可。在访问数据时，若数据项在链表中存在，则把该节点移到链表头部，否则返回-1。这样一来在链表尾部的节点就是最近最久未访问的数据项。

1) set(key,value)

若key在hashmap中存在，则先重置value，然后获取对应节点cur，将其从链表删除，并移到链表头；不存在，则新建一个节点，并将节点放到链表的头部。当Cache满，删除链表最后一个节点。

2) get(key)

若key在hashmap中存在，把对应的节点放到链表头，并返回对应value 若不存在，则返回-1 即保证基本的get/set同时，还要保证最近访问(get或put)的节点保持在限定容量的Cache中，如果超过容量则应该把LRU(近期最少使用)的节点删除掉。

当我们在get/set一个节点时都会把操作的这个节点移动到tail节点处，代表最新操作的节点，head节点永远指向最老的节点，当超过设定的容量时，我们就删除head节点指向的最老节点

就像是个LinkedHashMap，这样做的好处是，get/set在不冲突情况下可保证O(1)复杂度 也可通过双向链表保证LRU的删除/更新O(1)复杂度

当然可简化head和tail变成一个head节点，成环，这样head的next指向最旧的节点，prev指向最新的节点。

过期时间实现

维护一个线程

惰性删除

三、设计一个分布式锁

什么是分布式锁？

当在分布式模型下，数据只有一份（或有限制），此时需要利用锁的技术控制某一时刻修改数据的进程数。与单机模式下的锁不仅需要保证进程可见，还需要考虑进程与锁之间的网络问题。（我觉得分布式情况下之所以问题变得复杂，主要就是需要考虑到网络的延时和不可靠。。。一个大坑）分布式锁还是可以将标记存在内存，只是该内存不是某个进程分配的内存而是公共内存如 Redis、Memcache。至于利用数据库、文件等做锁与单机的实现是一样的，只要保证标记能互斥就行。

我们需要怎样的分布式锁？

- 可以保证在分布式部署的应用集群中，同一个方法在同一时间只能被一台机器上的一个线程执行。
- 这把锁要是一把可重入锁（避免死锁）
- 这把锁最好是一把阻塞锁（根据业务需求考虑要不要这条）
- 这把锁最好是一把公平锁（根据业务需求考虑要不要这条）
- 有高可用的获取锁和释放锁功能
- 获取锁和释放锁的性能要好

基于数据库做分布式锁

1、基于乐观锁

(1)、基于表主键唯一做分布式锁 利用主键唯一的特性，如果有多个请求同时提交到数据库的话，数据库会保证只有一个操作可以成功，那么我们就可以认为操作成功的那个线程获得了该方法的锁，当方法执行完毕之后，想要释放锁的话，删除这条数据库记录即可。上面这种简单的实现有以下几个问题：

- 这把锁强依赖数据库的可用性，数据库是一个单点，一旦数据库挂掉，会导致业务系统不可用。
- 这把锁没有失效时间，一旦解锁操作失败，就会导致锁记录一直在数据库中，其他线程无法再获得锁。
- 这把锁只能是非阻塞的，因为数据的insert操作，一旦插入失败就会直接报错。没有获得锁的线程并不会进入排队队列，要想再次获得锁就要再次触发获得锁操作。
- 这把锁是非重入的，同一个线程在没有释放锁之前无法再次获得该锁。因为数据中数据已经存在了。
- 这把锁是非公平锁，所有等待锁的线程凭运气去争夺锁。

- 在 MySQL 数据库中采用主键冲突防重，在大并发情况下有可能会造成锁表现象。
当然，我们也可以有其他方式解决上面的问题。
- 数据库是单点？搞两个数据库，数据之前双向同步，一旦挂掉快速切换到备库上。
- 没有失效时间？只要做一个定时任务，每隔一定时间把数据库中的超时数据清理一遍。
- 非阻塞的？搞一个 while 循环，直到 insert 成功再返回成功。
- 非重入的？在数据库表中加个字段，记录当前获得锁的机器的主机信息和线程信息，那么下次再获取锁的时候先查询数据库，如果当前机器的主机信息和线程信息在数据库可以查到的话，直接把锁分配给他就可以了。
- 非公平的？再建一张中间表，将等待锁的线程全记录下来，并根据创建时间排序，只有最先创建的允许获取锁。比较好的办法是在程序中生产主键进行防重。

(2)、基于表字段版本号做分布式锁

这个策略源于 mysql 的 mvcc 机制，使用这个策略其实本身没有什么问题，唯一的问题就是对数据表侵入较大，我们要为每个表设计一个版本号字段，然后写一条判断 sql 每次进行判断，增加了数据库操作的次数，在高并发的要求下，对数据库连接的开销也是无法忍受的。

2、基于悲观锁

(1) 、基于数据库排他锁做分布式锁 在查询语句后面增加for update，数据库会在查询过程中给数据库表增加排他锁 (注意： InnoDB 引擎在加锁的时候，只有通过索引进行检索的时候才会使用行级锁，否则会使用表级锁。这里我们希望使用行级锁，就要给要执行的方法字段名添加索引，值得注意的是，这个索引一定要创建成唯一索引，否则会出现多个重载方法之间无法同时被访问的问题。重载方法的话建议把参数类型也加上。)。当某条记录被加上排他锁之后，其他线程无法再在该行记录上增加排他锁。我们可以认为获得排他锁的线程即可获得分布式锁，当获取到锁之后，可以执行方法的业务逻辑，执行完方法之后，通过connection.commit()操作来释放锁。这种方法可以有效的解决上面提到的无法释放锁和阻塞锁的问题。阻塞锁 for update语句会在执行成功后立即返回，在执行失败时一直处于阻塞状态，直到成功。锁定之后服务宕机，无法释放？使用这种方式，服务宕机之后数据库会自己把锁释放掉。

但是还是无法直接解决数据库单点和可重入问题。这里还可能存在另外一个问题，虽然我们对方法字段名使用了唯一索引，并且显示使用 for update 来使用行级锁。但是，MySQL 会对查询进行优化，即便在条件中使用了索引字段，是否使用索引来检索数据是由 MySQL 通过判断不同执行计划的代价来决定的，如果 MySQL 认为全表扫效率更高，比如对一些很小的表，它就不会使用索引，这种情况下 InnoDB 将使用表锁，而不是行锁。如果发生这种情况就悲剧了。。。还有一个问题，就是我们要使用排他锁来进行分布式锁的 lock，那么一个排他锁长时间不提交，就会占用数据库连接。一旦类似的连接变得多了，就可能把数据库连接池撑爆。

优缺点 优点：简单，易于理解 缺点：会有各种各样的问题（操作数据库需要一定的开销，使用数据库的行级锁并不一定靠谱，性能不靠谱）

基于 Redis 做分布式锁

1、基于 redis 的 setnx()、expire() 方法做分布式锁

setnx() setnx 的含义就是 SET if Not Exists，其主要有两个参数 setnx(key, value)。该方法是原子的，如果 key 不存在，则设置当前 key 成功，返回 1；如果当前 key 已经存在，则设置当前 key 失败，返回 0。 **expire()** expire 设置过期时间，要注意的是 setnx 命令不能设置 key 的超时时间，只能通过 expire() 来对 key 设置。

使用步骤：

- setnx(lockkey, 1) 如果返回 0，则说明占位失败；如果返回 1，则说明占位成功
- expire() 命令对 lockkey 设置超时时间，为的是避免死锁问题。
- 执行完业务代码后，可以通过 delete 命令删除 key。

这个方案其实是可以解决日常工作中的需求的，但从技术方案的探讨上来说，可能还有一些可以完善的地方。比如，如果在第一步 setnx 执行成功后，在 expire() 命令执行成功前，发生了宕机的现象，那么就依然会出现死锁的问题，所以如果要对其进行完善的话，可以使用 redis 的 setnx()、get() 和 getset() 方法来实现分布式锁。

2、基于 redis 的 setnx()、get()、getset()方法做分布式锁

这个方案的背景主要是在 setnx() 和 expire() 的方案上针对可能存在的死锁问题，做了一些优化。getset() 这个命令主要有两个参数 getset(key, newValue)。该方法是原子的，对 key 设置 newValue 这个值，并且返回 key 原来的旧值。假设 key 原来是不存在的，那么多次执行这个命令，会出现下边的效果：getset(key, "value1") 返回 null 此时 key 的值会被设置为 value1 getset(key, "value2") 返回 value1 此时 key 的值会被设置为 value2 依次类推！

使用步骤：

- setnx(lockkey, 当前时间+过期超时时间)，如果返回 1，则获取锁成功；如果返回 0 则没有获取到锁，转向 2。
- get(lockkey) 获取值 oldExpireTime，并将这个 value 值与当前的系统时间进行比较，如果小于当前系统时间，则认为这个锁已经超时，可以允许别的请求重新获取，转向 3。
- 计算 newExpireTime = 当前时间+过期超时时间，然后 getset(lockkey, newExpireTime) 会返回当前 lockkey 的值 currentExpireTime。
- 判断 currentExpireTime 与 oldExpireTime 是否相等，如果相等，说明当前 getset 设置成功，获取到了锁。如果不相等，说明这个锁又被别的请求获取走了，那么当前请求可以直接返回失败，或者继续重试。
- 在获取到锁之后，当前线程可以开始自己的业务处理，当处理完毕后，比较自己的处理时间和对于锁设置的超时时间，如果小于锁设置的超时时间，则直接执行 delete 释放锁；如果大于锁设置的超时时间，则不需要再锁进行处理。

基于 ZooKeeper 做分布式锁

zookeeper 锁相关基础知识 zk 一般由多个节点构成（单数），采用 zab 一致性协议。因此可以将 zk 看成一个单点结构，对其修改数据其内部自动将所有节点数据进行修改而后才提供查询服务。zk 的数据以目录树的形式，每个目录称为 znode，znode 中可存储数据（一般不超过 1M），还可以在其中增加子节点。子节点有三种类型。序列化节点，每在该节点下增加一个节点自动给该节点的名称上自增。临时节点，一旦创建这个 znode 的客户端与服务器失去联系，这个 znode 也将自动删除。最后就是普通节点。Watch 机制，client 可以监控每个节点的变化，当产生变化会给 client 产生一个事件。

zk 基本锁 原理：利用临时节点与 watch 机制。每个锁占用一个普通节点 /lock，当需要获取锁时在 /lock 目录下创建一个临时节点，创建成功则表示获取锁成功，失败则 watch/lock 节点，有删除操作后再去争锁。临时节点好处在于当进程挂掉后能自动上锁的节点自动删除即取消锁。**缺点：**所有取锁失败的进程都监听父节点，很容易发生羊群效应，即当释放锁后所有等待进程一起来创建节点，并发量很大。

zk 锁优化 原理：上锁改为创建临时有序节点，每个上锁的节点均能创建节点成功，只是其序号不同。只有序号最小的可以拥有锁，如果这个节点序号不是最小的则 watch 序号比本身小的前一个节点（公平锁）。**步骤：**

- 在 /lock 节点下创建一个有序临时节点 (EPHEMERAL_SEQUENTIAL)。
- 判断创建的节点序号是否最小，如果是最小则获取锁成功。不是则取锁失败，然后 watch 序号比本身小的前一个节点。（避免很多线程watch同一个node，导致羊群效应）
- 当取锁失败，设置 watch 后则等待 watch 事件到来后，再次判断是否序号最小。
- 取锁成功则执行代码，最后释放锁（删除该节点）。

优缺点 优点：有效的解决单点问题，不可重入问题，非阻塞问题以及锁无法释放的问题。实现起来较为简单。缺点：性能上可能并没有缓存服务那么高，因为每次在创建锁和释放锁的过程中，都要动态创建、销毁临时节点来实现锁功能。ZK 中创建和删除节点只能通过 Leader 服务器来执行，然后将数据同步到所有的 Follower 机器上。还需要对 ZK 的原理有所了解。

使用分布式锁的注意事项

- 1、注意分布式锁的开销
- 2、注意加锁的粒度
- 3、加锁的方式

分布式可重入锁的设计

需记录机器线程id (MAC地址 + jvm进程ID + 线程ID) 和重入次数

四、设计一个分布式环境下的统一配置中心

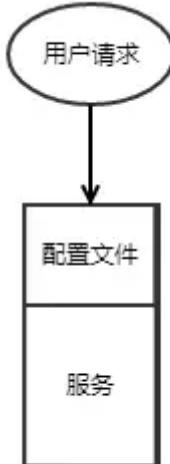
配置中心概述

对于配置文件，我们不陌生，它提供我们可以动态修改程序运行能力。引用别人的一句话就是：系统运行时 (runtime) 飞行姿态的动态调整。

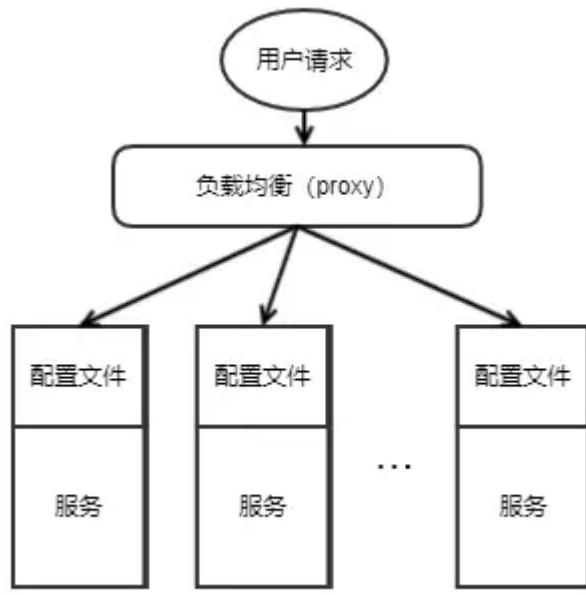
我可以把我们的工作称之为**在快速飞行的飞机上修理零件**。我们人类总是无法掌控和预知一切。对于我们系统来说，我们总是需要预留一些控制线条，以便在我们需要的时候做出调整，控制系统方向（如灰度控制、限流调整），这对于拥抱变化的互联网行业尤为重要。对于单机版，我们称之为**配置（文件）**，对于分布式集群系统，我们称之为**配置中心（系统）**；下面聊聊我们的配置中心。

演进中的配置

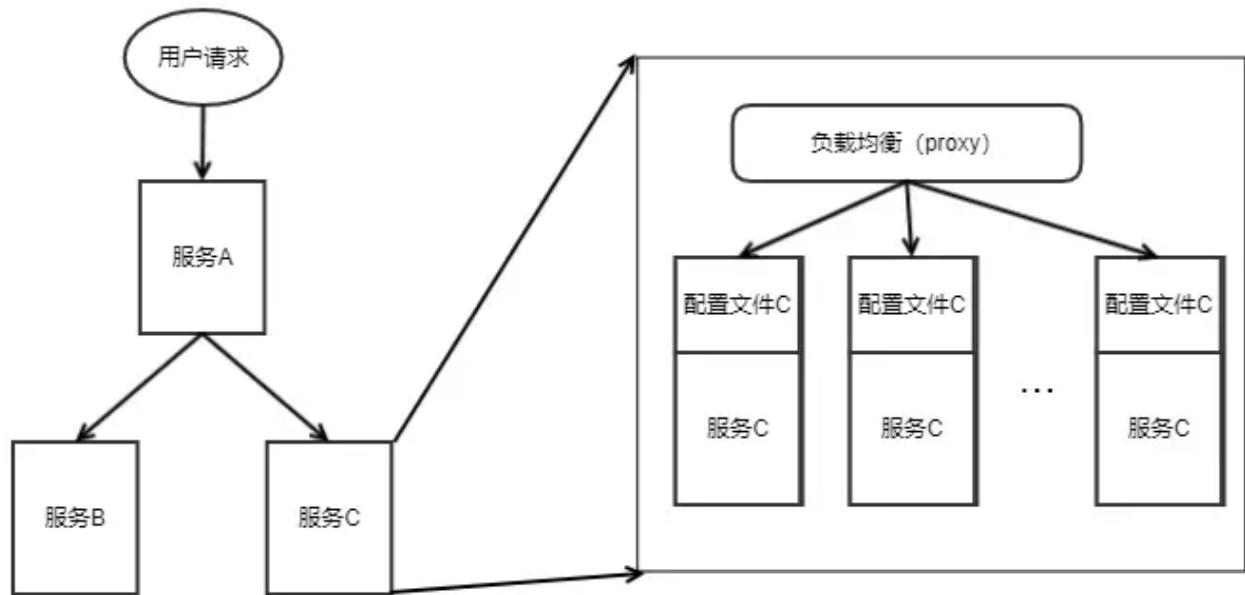
当我们是一个单机服务的是，我们的配置通常写在一个文件中的，代码发布的时候，把配置文件和程序推送到机器上去。



当随着业务的用户量增加，通常我们会把我们的服务进行多机器（集群）部署。这时候，配置的发布就变成了如下：



业务的急剧扩张，导致单机服务无法满足业务需求。这时候需要对单体大服务进行切开，服务走向SOA(微服务化)。



这种场景中，配置文件的部署可能如上图所示。这样去部署配置简直是一场噩梦，而且无法做到快速的动态的调整。失去了配置主要意义之一。这时候就需要今天说的统一配置中心。

配置中心之简版

首先来看下我们理想中的配置中心需要具备哪些特点。

配置的增删改查

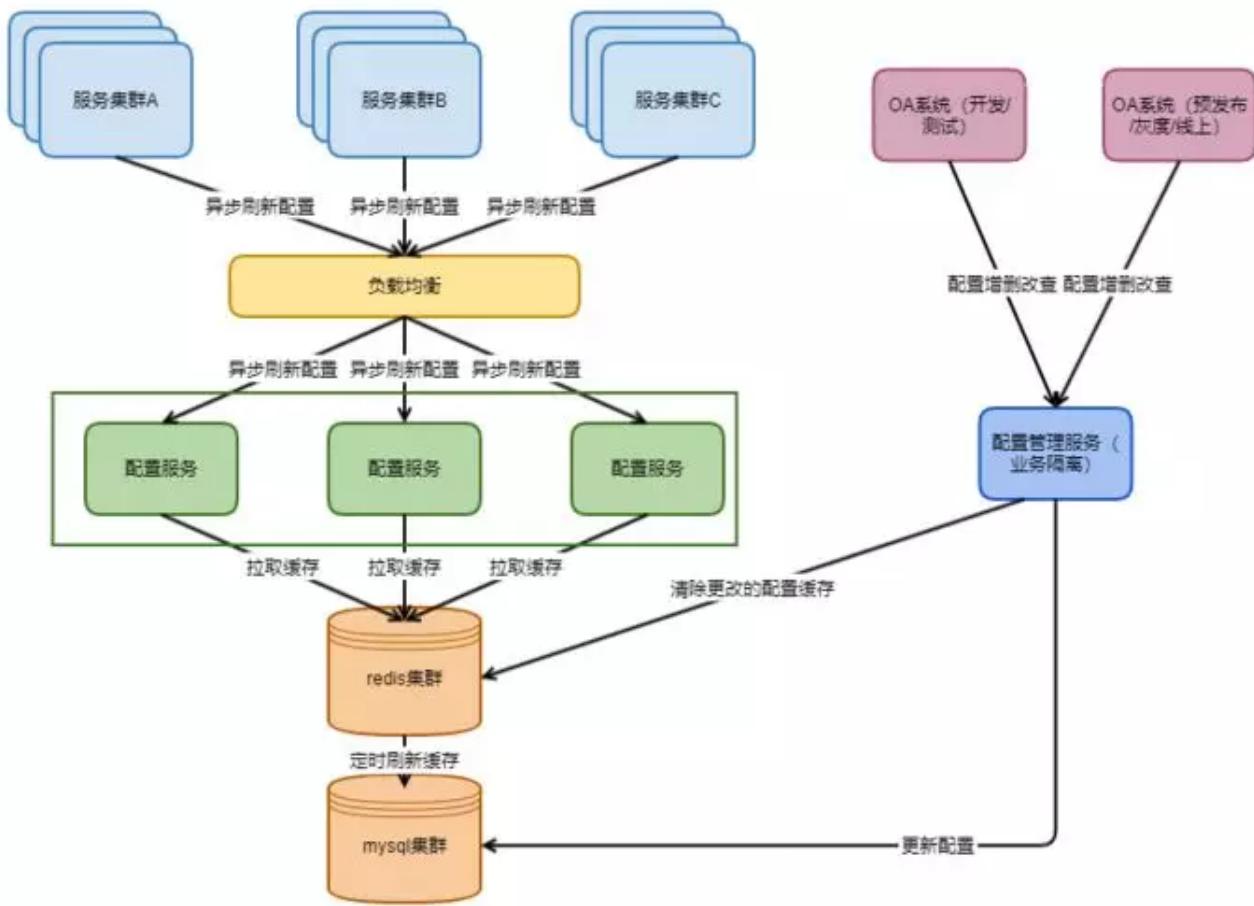
不同环境配置隔离（开发、测试、预发布、灰度/线上）

高性能、高可用性

请求量多、高并发

读多写少

我们可以设计出如下的简版配置中心



设计说明点：

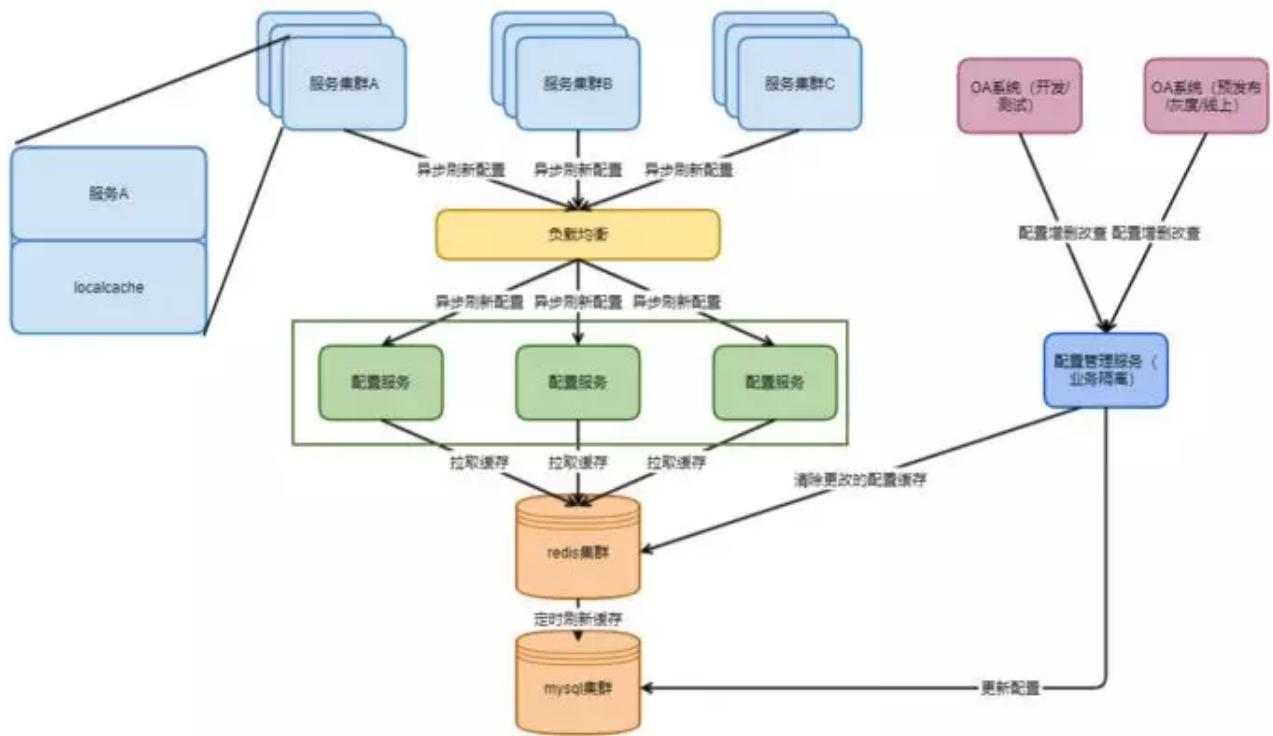
- 通过OA系统对每一条配置（每一个配置有唯一的配置ID）进行增删改查。
- 区分不同环境的配置，每个环境同一配置ID对应不同数据库记录。
- 配置最终以json格式（便于编辑和理解）储存在mysql数据库中。
- 引入redis集群，做配置的缓存（比如可以设置配置修改后1分钟后生效）
- 配置对外服务，多机器部署，满足性能需要。
- 如果有必要，可以引入配置历史修改记录。
-

很多时候，这样可以基本上满足我们对配置系统的基本需求。

这种设计，由于所有的配置都存放在集中式缓存中，这样集中式的缓存也会有他的性能瓶颈。而且，每次配置的访问都需要发起rpc请求(网络请求)，因此考虑在客户端引入[本地缓存的选择及其原理](#) (localCache, 例如 Ehcache)。

配置中心之性能改进

为了提高配置中心的可用性，减少网络请求等因素对性能带来的影响，我们考虑在客户端引入localcache，来解决系统的高可用，高性能、可伸缩性。



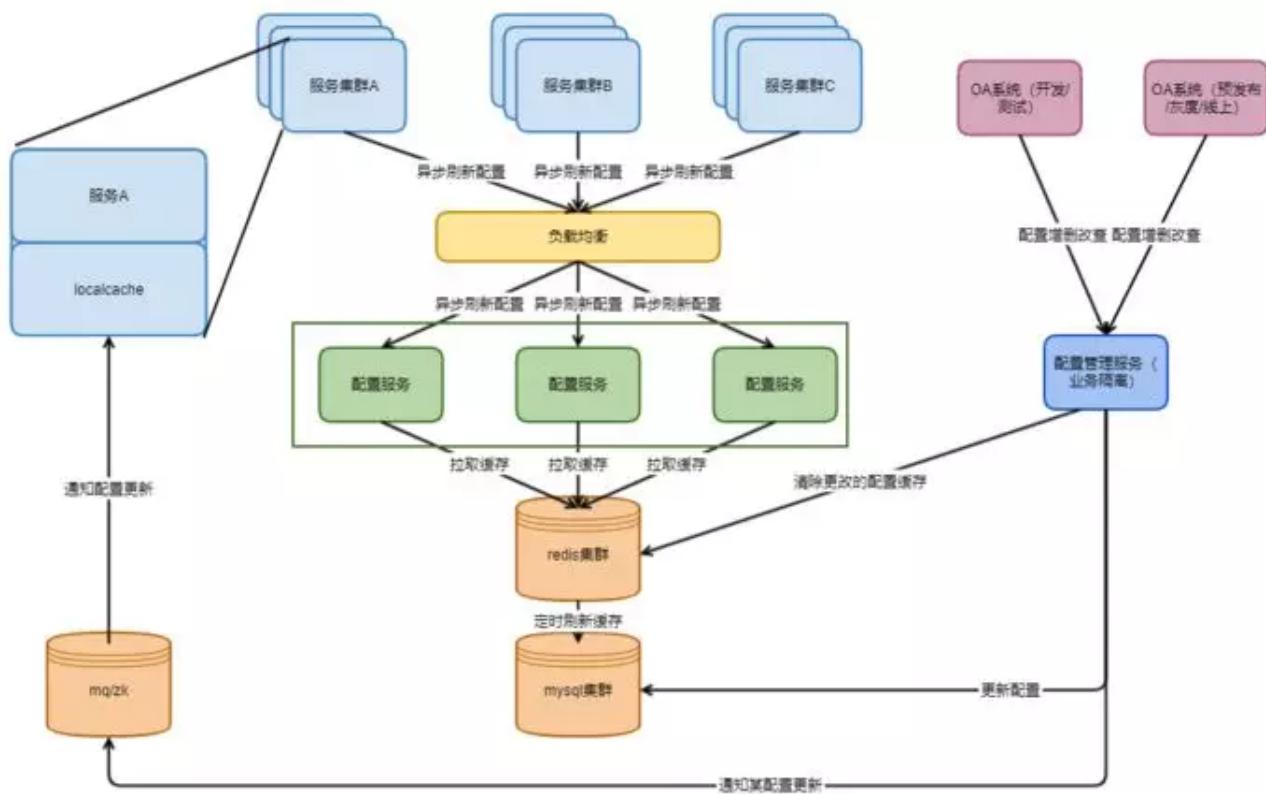
相对于第一版的改进点是，在客户端引入localcache。开启线程异步调用配置服务，更新本地配置。这样可以减少rpc调用。

这种方式较为简单，但是存在一个问题，就是一旦用户量大的时候，会增加很多无意义的轮询。因为配置中心的定位就表明了他的修改并不会很多，所以大多数情况下的轮询都是无意义的。会给缓存系统增加很多无谓的压力。

同时，由于各个客户端的拉取时间及网络延迟等都不尽相同，也会存在数据一致性的问题，

配置中心之可用性改进

还好，配置通常都只会有一个入口修改，因此可以考虑在配置修改后，通知应用服务清理本地缓存和分布式缓存。这里可以引入mq或ZooKeeper。



感兴趣的朋友可以了解下阿里巴巴的Diamond，他的工作原理就是这种通过推拉结合的方式，减少不必要的轮询，并且可以降低缓存系统的负载。

五、如何准备HR面试