

一、Java

(一). 基础

1. Java 基本功

- 1.1. Java 入门 (基础概念与常识)
 - 1.1.1. Java 语言有哪些特点?
 - 1.1.2. 关于 JVM JDK 和 JRE 最详细通俗的解答
 - 1.1.2.1. JVM
 - 1.1.2.2. JDK 和 JRE
 - 1.1.3. Oracle JDK 和 OpenJDK 的对比
 - 1.1.4. Java 和 C++的区别?
 - 1.1.5. 什么是 Java 程序的主类 应用程序和小程序的主类有何不同?
 - 1.1.6. Java 应用程序与小程序之间有哪些差别?
 - 1.1.7. import java 和 javax 有什么区别?
 - 1.1.8. 为什么说 Java 语言“编译与解释并存”?

1.2. Java 语法

- 1.2.1. 字符型常量和字符串常量的区别?
- 1.2.2. 关于注释?
- 1.2.3. 标识符和关键字的区别是什么?
- 1.2.4. Java 中有哪些常见的关键字?
- 1.2.5. 自增自减运算符
- 1.2.6. continue、break、和return的区别是什么?
- 1.2.7. Java 泛型了解么? 什么是类型擦除? 介绍一下常用的通配符?
- 1.2.8. == 和 equals 的区别
- 1.2.9. hashCode() 与 equals()

1.3. 基本数据类型

- 1.3.1. Java 中的几种基本数据类型是什么? 对应的包装类型是什么? 各自占用多少字节呢?
- 1.3.2. 自动装箱与拆箱
- 1.3.3. 8 种基本类型的包装类和常量池

1.4. 方法 (函数)

- 1.4.1. 什么是方法的返回值? 返回值在类的方法里的作用是什么?
- 1.4.2. 为什么 Java 中只有值传递?
- 1.4.3. 重载和重写的区别
- 1.4.4. 深拷贝 vs 浅拷贝
- 1.4.5. 方法的四种类型

2. Java 面向对象

2.1. 类和对象

- 2.1.1. 面向对象和面向过程的区别
- 2.1.2. 构造器 Constructor 是否可被 override?
- 2.1.3. 在 Java 中定义一个不做事且没有参数的构造方法的作用
- 2.1.4. 成员变量与局部变量的区别有哪些?
- 2.1.5. 创建一个对象用什么运算符? 对象实体与对象引用有何不同?
- 2.1.6. 一个类的构造方法的作用是什么? 若一个类没有声明构造方法, 该程序能正确执行吗? 为什么?
- 2.1.7. 构造方法有哪些特性?
- 2.1.8. 在调用子类构造方法之前会先调用父类没有参数的构造方法, 其目的是?
- 2.1.9. 对象的相等与指向他们的引用相等, 两者有什么不同?

2.2. 面向对象三大特征

- 2.2.1. 封装
- 2.2.2. 继承
- 2.2.3. 多态

2.3. 修饰符

- 2.3.1. 在一个静态方法内调用一个非静态成员为什么是非法的?
- 2.3.2. 静态方法和实例方法有何不同

2.4. 接口和抽象类

- 2.4.1. 接口和抽象类的区别是什么?

2.5. 其它重要知识点

- 2.5.1. String StringBuffer 和 StringBuilder 的区别是什么? String 为什么是不可变的?
- 2.5.2. Object 类的常见方法总结
- 2.5.3. == 与 equals(重要)
- 2.5.4. hashCode 与 equals (重要)
 - 2.5.4.1. hashCode () 介绍
 - 2.5.4.2. 为什么要有 hashCode
 - 2.5.4.3. hashCode () 与 equals () 的相关规定
- 2.5.5. Java 序列化中如果有些字段不想进行序列化, 怎么办?
- 2.5.6. 获取用键盘输入常用的两种方法

3. Java 核心技术

3.1. 集合

- 3.1.1. Collections 工具类和 Arrays 工具类常见方法总结

3.2. 异常

- 3.2.1. Java 异常类层次结构图
- 3.2.2. Throwable 类常用方法
- 3.2.3. try-catch-finally
- 3.2.4. 使用 `try-with-resources` 来代替 `try-catch-finally`

3.3. 多线程

- 3.3.1. 简述线程、程序、进程的基本概念。以及他们之间关系是什么?
- 3.3.2. 线程有哪些基本状态?

3.4. 文件与 I/O 流

- 3.4.1. Java 中 IO 流分为几种?
 - 3.4.1.1. 既然有了字节流,为什么还要有字符流?
 - 3.4.1.2. BIO,NIO,AIO 有什么区别?

(二). 容器

1. ArrayList

- 1.1 ArrayList简介
- 1.2 ArrayList核心源码
- 1.3 ArrayList源码分析
 - 1.3.1 System.arraycopy()和Arrays.copyOf()方法
 - 1.3.2 两者联系与区别
 - 1.3.3 ArrayList 核心扩容技术
 - 1.3.4 内部类
- 1.4 ArrayList经典Demo

2. LinkedList

- 2.1 简介
- 2.2 内部结构分析
- 2.3 LinkedList源码分析
 - 2.3.1 构造方法
 - 2.3.2 add方法
 - 2.3.3 根据位置取数据的方法
 - 2.3.4 根据对象得到索引的方法
 - 2.3.5 检查链表是否包含某对象的方法:
 - 2.3.6 删除方法
- 2.4 LinkedList类常用方法测试

3. HashMap

- 3.1 HashMap 简介
- 3.2 底层数据结构分析
 - 3.2.1 JDK1.8之前
 - 3.2.2 JDK1.8之后
- 3.3 HashMap源码分析
 - 3.3.1 构造方法
 - 3.3.2 put方法
- 3.4 get方法
- 3.5 resize方法
- 3.6 HashMap常用方法测试

(三). 并发

- 1. 并发容器
 - 1.1 JDK 提供的并发容器总结
 - 1.2 ConcurrentHashMap
 - 1.3 CopyOnWriteArrayList
 - 1.3.1 CopyOnWriteArrayList 简介
 - 1.3.2 CopyOnWriteArrayList 是如何做到的?
 - 1.3.3 CopyOnWriteArrayList 读取和写入源码简单分析
 - 1.4 ConcurrentLinkedQueue
 - 1.5 BlockingQueue
 - 1.5.1 BlockingQueue 简单介绍
 - 1.5.2 ArrayBlockingQueue
 - 1.5.3 LinkedBlockingQueue
 - 1.5.4 PriorityBlockingQueue
 - 1.6 ConcurrentSkipListMap
- 2. 线程池
 - 2.1 使用线程池的好处
 - 2.2 Executor 框架
 - 2.2.1 简介
 - 2.2.2 Executor 框架结构(主要由三大部分组成)
 - 2.2.3 Executor 框架的使用示意图
 - 2.3 (重要)ThreadPoolExecutor 类简单介绍
 - 2.3.1 ThreadPoolExecutor 类分析
 - 2.3.2 推荐使用 `ThreadPoolExecutor` 构造函数创建线程池
 - 2.4 (重要)ThreadPoolExecutor 使用示例
 - 2.4.1 示例代码: `Runnable + ThreadPoolExecutor`
 - 2.4.2 线程池原理分析
 - 2.4.3 几个常见的对比
 - 2.4.4 加餐: `callable + ThreadPoolExecutor` 示例代码
 - 2.5 几种常见的线程池详解
 - 2.5.1 FixedThreadPool
 - 2.5.2 SingleThreadExecutor 详解
 - 2.5.3 CachedThreadPool 详解
 - 2.6 ScheduledThreadPoolExecutor 详解
 - 2.6.1 简介
 - 2.6.2 运行机制
 - 2.6.3 ScheduledThreadPoolExecutor 执行周期任务的步骤
 - 2.7 线程池大小确定
- 3. 乐观锁与悲观锁
 - 3.1 何谓悲观锁与乐观锁
 - 3.1.1 悲观锁
 - 3.1.2 乐观锁
 - 3.1.3 两种锁的使用场景
 - 3.2 乐观锁常见的两种实现方式
 - 3.2.1. 版本号机制
 - 3.2.2. CAS算法
 - 3.3 乐观锁的缺点
 - 3.3.1 ABA 问题
 - 3.3.2 循环时间长开销大
 - 3.3.3 只能保证一个共享变量的原子操作
 - 3.4 CAS与synchronized的使用情景
- 4. Atomic
 - 4.1 Atomic 原子类介绍
 - 4.2 基本类型原子类
 - 4.2.1 基本类型原子类介绍
 - 4.2.2 AtomicInteger 常见方法使用
 - 4.2.3 基本数据类型原子类的优势
 - 4.2.4 AtomicInteger 线程安全原理简单分析

- 4.3 数组类型原子类
 - 4.3.1 数组类型原子类介绍
 - 4.3.2 AtomicIntegerArray 常见方法使用
- 4.4 引用类型原子类
 - 4.4.1 引用类型原子类介绍
 - 4.4.2 AtomicReference 类使用示例
 - 4.4.3 AtomicStampedReference 类使用示例
 - 4.4.4 AtomicMarkableReference 类使用示例
- 4.5 对象的属性修改类型原子类
 - 4.5.1 对象的属性修改类型原子类介绍
 - 4.5.2 AtomicIntegerFieldUpdater 类使用示例
- 5. AQS
 - 5.1 AQS 简单介绍
 - 5.2 AQS 原理
 - 5.2.1 AQS 原理概览
 - 5.2.2 AQS 对资源的共享方式
 - 5.2.3 AQS 底层使用了模板方法模式
 - 5.3 Semaphore(信号量)-允许多个线程同时访问
 - 5.4 CountDownLatch (倒计时器)
 - 5.4.1 CountDownLatch 的两种典型用法
 - 5.4.2 CountDownLatch 的使用示例
 - 5.4.3 CountDownLatch 的不足
 - 5.4.4 CountDownLatch 相常见面试题
 - 5.5 CyclicBarrier(循环栅栏)
 - 5.5.1 CyclicBarrier 的应用场景
 - 5.5.2 CyclicBarrier 的使用示例
 - 5.5.3 CyclicBarrier 源码分析
 - 5.5.4 CyclicBarrier 和 CountDownLatch 的区别
 - 5.6 ReentrantLock 和 ReentrantReadWriteLock
- (四). JVM
 - 1. Java内存区域
 - 1.1 概述
 - 1.2 运行时数据区域
 - 1.2.1 程序计数器
 - 1.2.2 Java 虚拟机栈
 - 1.2.3 本地方法栈
 - 1.2.4 堆
 - 1.2.5 方法区
 - 1.2.5.1 方法区和永久代的关系
 - 1.2.5.2 常用参数
 - 1.2.5.3 为什么要将永久代 (PermGen) 替换为元空间 (MetaSpace) 呢?
 - 1.2.6 运行时常量池
 - 1.2.7 直接内存
 - 1.3 HotSpot 虚拟机对象探秘
 - 1.3.1 对象的创建
 - 1.3.2 对象的内存布局
 - 1.3.3 对象的访问定位
 - 1.4 重点补充内容
 - 1.4.1 String 类和常量池
 - 1.4.2 String s1 = new String("abc");这句话创建了几个字符串对象?
 - 1.4.3 8 种基本类型的包装类和常量池
 - 2. JVM垃圾回收
 - 2.1 揭开 JVM 内存分配与回收的神秘面纱
 - 2.1.1 对象优先在 eden 区分配
 - 2.1.2 大对象直接进入老年带
 - 2.1.3 长期存活的对象将进入老年带
 - 2.1.4 动态对象年龄判定
 - 2.2 对象已经死亡?

- 2.2.1 引用计数法
 - 2.2.2 可达性分析算法
 - 2.2.3 再谈引用
 - 2.2.4 不可达的对象并非“非死不可”
 - 2.2.5 如何判断一个常量是废弃常量
 - 2.2.6 如何判断一个类是无用的类
 - 2.3 垃圾收集算法
 - 2.3.1 标记-清除算法
 - 2.3.2 复制算法
 - 2.3.3 标记-整理算法
 - 2.3.4 分代收集算法
 - 2.4 垃圾收集器
 - 2.4.1 Serial 收集器
 - 2.4.2 ParNew 收集器
 - 2.4.3 Parallel Scavenge 收集器
 - 2.4.4 Serial Old 收集器
 - 2.4.5 Parallel Old 收集器
 - 2.4.6 CMS 收集器
 - 2.4.7 G1 收集器
3. JDK 监控和故障处理工具
- 3.1 JDK 命令行工具
 - 3.1.1 `jps` :查看所有 Java 进程
 - 3.1.2 `jstat` :监视虚拟机各种运行状态信息
 - 3.1.3 `jinfo` :实时地查看和调整虚拟机各项参数
 - 3.1.4 `jmap` :生成堆转储快照
 - 3.1.5 `jhat` :分析 heapdump 文件
 - 3.1.6 `jstack` :生成虚拟机当前时刻的线程快照
 - 3.2 JDK 可视化分析工具
 - 3.2.1 JConsole:Java 监视与管理控制台
 - 3.2.2 Visual VM:多合一故障处理工具
4. 类文件结构
- 4.1 概述
 - 4.2 Class 文件结构总结
 - 4.2.1 魔数
 - 4.2.2 Class 文件版本
 - 4.2.3 常量池
 - 4.2.4 访问标志
 - 4.2.5 当前类索引,父类索引与接口索引集合
 - 4.2.6 字段表集合
 - 4.2.7 方法表集合
 - 4.2.8 属性表集合
5. 类加载过程
- 5.1 类加载过程
 - 5.1.1 加载
 - 5.1.2 验证
 - 5.1.3 准备
 - 5.1.4 解析
 - 5.1.5 初始化
 - 5.2 卸载
6. 类加载器
- 6.1 回顾一下类加载过程
 - 6.2 类加载器总结
 - 6.3 双亲委派模型
 - 6.3.1 双亲委派模型介绍
 - 6.3.2 双亲委派模型实现源码分析
 - 6.3.3 双亲委派模型的好处
 - 6.3.4 如果我们不想用双亲委派模型怎么办?

6.4 自定义类加载器

二、网络

(一). 计算机网络知识

1. 计算机概述
 - 1.1 基本术语
 - 1.2 重要知识点总结
2. 物理层
 - 2.1 基本术语
 - 2.2 重要知识点总结
 - 2.3 最重要的知识点
 - 2.3.1 拓展:
 - 2.3.2 几种常用的信道复用技术
 - 2.3.3 几种常用的宽带接入技术,主要是ADSL和FTTx
3. 数据链路层
 - 3.1 基本术语
 - 3.2 重要知识点总结
 - 3.3 最重要的知识点
4. 网络层
 - 4.1 基本术语
 - 4.2 重要知识点总结
 - 4.3 最重要知识点
5. 运输层
 - 5.1 基本术语
 - 5.2 重要知识点总结
 - 5.3 最重要的知识点
6. 应用层
 - 6.1 基本术语
 - 6.2 重要知识点总结
 - 6.3 最重要知识点总结

(二). HTTPS中的TLS

1. SSL 与 TLS
2. 从网络协议的角度理解 HTTPS
3. 从密码学的角度理解 HTTPS
 - 3.1. TLS 工作流程
 - 3.2. 密码基础
 - 3.2.1. 伪随机数生成器
 - 3.2.2. 消息认证码
 - 3.2.3. 数字签名
 - 3.2.4. 公钥密码
 - 3.2.5. 证书
 - 3.2.6. 密码小结
 - 3.3. TLS 使用的密码技术
 - 3.4. TLS 总结
4. RSA 简单示例

三、Linux

(一). 从认识操作系统开始

- 1.1 操作系统简介
- 1.2 操作系统简单分类
- 1.3 操作系统的内核
- 1.4 操作系统的用户态与内核态
 - 1.4.1 为什么要有用户态与内核态?
 - 1.4.2 用户态切换到内核态的几种方式
 - 1.4.3 物理内存RAM(Random Access Memory 随机存储器)
 - 1.4.4 虚拟内存(Virtual Memory)
 - 1.4.5 Swap交换空间

(二). 初探Linux

- 2.1 Linux简介
- 2.2 Linux诞生简介

2.3 Linux的分类

(三) Linux文件系统概览

3.1 Linux文件系统简介

3.2 Inode

3.2.1 Inode是什么?有什么作用?

3.3 文件类型与目录结构

3.4 Linux目录树

(四) Linux基本命令

4.1 目录切换命令

4.2 目录的操作命令(增删改查)

4.3 文件的操作命令(增删改查)

4.4 压缩文件的操作命令

4.5 Linux的权限命令

4.6 Linux 用户管理

4.7 Linux系统用户组的管理

4.8 其他常用命令

四、数据结构与算法

(一). 数据结构(布隆过滤器)

1.什么是布隆过滤器?

2.布隆过滤器的原理介绍

3.布隆过滤器使用场景

4.通过 Java 编程手动实现布隆过滤器

5.利用Google开源的 Guava中自带的布隆过滤器

6.Redis 中的布隆过滤器

6.1 介绍

6.2 使用Docker安装

6.3常用命令一览

6.4实际使用

(二). 算法

五、数据库

(一). MySQL

1. 基本操作

2. 数据库操作

3. 表的操作

4. 数据操作

5. 字符集编码

6. 数据类型(列类型)

7. 列属性(列约束)

8. 建表规范

9. SELECT

10. UNION

11. 子查询

12. 连接查询(join)

13. TRUNCATE

14. 备份与还原

15. 视图

16. 事务(transaction)

17. 锁表

18. 触发器

19. SQL编程

20. 存储过程

21. 用户和权限管理

22. 表维护

23. 杂项

(二). Redis

1. 5种基本数据结构

1.1 Redis 简介

1.1.1 Redis 的优点

- 1.1.2 Redis 的安装
- 1.1.3 测试本地 Redis 性能
- 1.2 Redis 五种基本数据结构
 - 1.2.1 字符串 string
 - 1.2.2 列表 list
 - 1.2.3 字典 hash
 - 1.2.4 集合 set
 - 1.2.5 有序列表 zset
- 2. 跳跃表
 - 2.1 跳跃表简介
 - 2.1.1 为什么使用跳跃表
 - 2.1.2 本质是解决查找问题
 - 2.1.3 更进一步的跳跃表
 - 2.2 跳跃表的实现
 - 2.2.1 随机层数
 - 2.2.2 创建跳跃表
 - 2.2.3 插入节点实现
 - 2.2.4 节点删除实现
 - 2.2.5 节点更新实现
 - 2.2.6 元素排名的实现
- 3. 分布式锁深入探究
 - 3.1 分布式锁简介
 - 3.1.1 为何需要分布式锁
 - 3.1.2 Java 中实现的常见方式
 - 3.1.3 Redis 分布式锁的问题
 - 3.2 Redis 分布式锁的实现
 - 3.2.1 代码实现
- 4. Redlock分布式锁
 - 4.1 什么是 RedLock
 - 4.2 怎么在单节点上实现分布式锁
 - 4.3 Redlock 算法
 - 4.4 失败重试
 - 4.5 放锁
 - 4.6 性能、崩溃恢复和 fsync
- 5. 如何做可靠的分布式锁, Redlock真的可行么
 - 5.1 用锁保护资源
 - 5.2 使用 Fencing (栅栏) 使得锁变安全
 - 5.3 使用时间来解决一致性
 - 5.4 用不可靠的时间打破 Redlock
 - 5.5 Redlock 的同步性假设
 - 5.6 结论
- 6. 神奇的HyperLoglog解决统计问题
 - 6.1 HyperLogLog 简介
 - 6.1.1 关于基数统计
 - 6.1.2 基数统计的常用方法
 - 6.1.3 概率算法
 - 6.2 HyperLogLog 原理
 - 6.2.1 代码实验
 - 6.2.2 更近一步：分桶平均
 - 6.2.3 真实的 HyperLogLog
 - 6.3 Redis 中的 HyperLogLog 实现
 - 6.3.1 密集型存储结构
 - 6.3.2 稀疏存储结构
 - 6.3.3 对象头
 - 6.4 HyperLogLog 的使用
- 7. 亿级数据过滤和布隆过滤器
 - 7.1 布隆过滤器简介
 - 7.1.1 布隆过滤器是什么

- 7.1.2 布隆过滤器的使用场景
- 7.2 布隆过滤器原理解析
- 7.3 布隆过滤器的使用
 - 7.3.1 布隆过滤器的基本用法
- 7.4 布隆过滤器代码实现
 - 7.4.1 自己简单模拟实现
 - 7.4.2 手动实现参考
 - 7.4.3 使用 Google 开源的 Guava 中自带的布隆过滤器
- 8. GeoHash查找附近的人
 - 8.1 使用数据库实现查找附近的人
 - 8.2 GeoHash 算法简述
 - 8.3 在 Redis 中使用 Geo
 - 8.3.1 增加
 - 8.3.2 距离
 - 8.3.3 获取元素位置
 - 8.3.4 获取元素的 hash 值
 - 8.3.5 附近的公司
 - 8.3.6 注意事项
- 9. 持久化
 - 9.1 持久化简介
 - 9.1.1 持久化发生了什么 | 从内存到磁盘
 - 9.1.2 如何尽可能保证持久化的安全
 - 9.2 Redis 中的两种持久化方式
 - 9.2.1 方式一：快照
 - 9.2.2 方式二：AOF
 - 9.2.3 Redis 4.0 混合持久化
- 10. 发布订阅与Stream
 - 10.1 Redis 中的发布/订阅功能
 - 10.1.1 PubSub 简介
 - 10.1.2 快速体验
 - 10.1.3 实现原理
 - 10.1.4 订阅频道原理
 - 10.1.5 订阅模式原理
 - 10.1.6 PubSub 的缺点
 - 10.2 更为强大的 Stream | 持久化的发布/订阅系统
 - 10.2.1 消息 ID 和消息内容
 - 10.2.2 增删改查示例
 - 10.2.3 独立消费示例
 - 10.2.4 创建消费者示例
 - 10.2.5 组内消费示例
 - 10.2.6 QA 1: Stream 消息太多怎么办？ | Stream 的上限
 - 10.2.7 QA 2: PEL 是如何避免消息丢失的？
 - 10.2.8 Redis Stream Vs Kafka
- 11. [集群]入门实践教程
 - 11.1 Redis 集群概述
 - 11.1.1 Redis 主从复制
 - 11.1.2 Redis 哨兵
 - 11.1.3 Redis 集群化
 - 11.2 主从复制
 - 11.2.1 主从复制主要的作用
 - 11.2.2 快速体验
 - 11.2.3 实现原理简析
 - 11.3 Redis Sentinel 哨兵
 - 11.3.1 快速体验
 - 11.3.2 客户端访问哨兵系统代码演示
 - 11.3.3 新的主服务器是怎样被挑选出来的？
 - 11.4 Redis 集群
 - 11.4.1 基本原理

- 11.4.2 集群的主要作用
- 11.4.3 快速体验
- 11.4.5 数据分区方案简析
- 11.4.6 节点通信机制简析
- 11.4.7 数据结构简析
- 12. Redis数据类型、编码、底层数据结构
 - 12.1 Redis构建的类型系统
 - 12.1.1 redisObject对象
 - 12.1.2 命令的类型检查和多态
 - 12.2 5种数据类型对应的编码和数据结构
 - 12.2.1 string
 - 12.2.2 list`list`列表,它是简单的字符串列表,你可以添加一个元素到列表的头部,或者尾部。
 - 12.2.3 set
 - 12.2.4 zset
 - 12.2.5 hash
 - 12.3 内存回收和内存共享

六、系统设计

(一). RestFul API

- 1. 重要概念
- 2. REST 接口规范
 - 2.1 动作
 - 2.2 路径 (接口命名)
 - 2.3 过滤信息 (Filtering)
 - 2.4 状态码 (Status Codes)
- 3. HATEOAS

(二). 常用框架

1. Spring常见问题

- 1.1 什么是 Spring 框架?
- 1.2 列举一些重要的Spring模块?
- 1.3 @RestController vs @Controller
- 1.4 Spring IOC & AOP
 - 1.4.1 谈谈自己对于 Spring IoC 和 AOP 的理解
 - 1.4.2 Spring AOP 和 AspectJ AOP 有什么区别?
- 1.5 Spring bean
 - 1.5.1 Spring 中的 bean 的作用域有哪些?
 - 1.5.2 Spring 中的单例 bean 的线程安全问题了解吗?
 - 1.5.3 @Component 和 @Bean 的区别是什么?
 - 1.5.4 将一个类声明为Spring的 bean 的注解有哪些?
 - 1.5.5 ### 5.5 Spring 中的 bean 生命周期?

1.6 Spring MVC

- 1.6.1 说自己对于 Spring MVC 了解?
- 1.6.2 SpringMVC 工作原理了解吗?

1.7 Spring 框架中用到了哪些设计模式?

1.8 Spring 事务

- 1.8.1 Spring 管理事务的方式有几种?
- 1.8.2 Spring 事务中的隔离级别有哪几种?
- 1.8.3 Spring 事务中哪几种事务传播行为?
- 1.8.4 @Transactional(rollbackFor = Exception.class)注解了解吗?

1.9 JPA

- 1.9.1 如何使用JPA在数据库中非持久化一个字段?

2. Spring常用注解

2.1 `@SpringBootApplication`

2.2 Spring Bean 相关

2.2.1 `@Autowired`

2.2.2 `Component, @Repository, @Service, @Controller`

2.2.3 `@RestController`

2.2.4 `@Scope`

- 2.2.5 `Configuration`
 - 2.3 处理常见的 HTTP 请求类型
 - 2.3.1 GET 请求
 - 2.3.2 POST 请求
 - 2.3.3 PUT 请求
 - 2.3.4 DELETE 请求
 - 2.3.5 PATCH 请求
 - 2.4 前后端传值
 - 2.4.1 `@PathVariable` 和 `@RequestParam`
 - 2.4.2 `@RequestBody`
 - 2.5 读取配置信息
 - 2.5.1 `@Value` (常用)
 - 2.5.2 `@ConfigurationProperties` (常用)
 - 2.5.3 `PropertySource` (不常用)
 - 2.6 参数校验
 - 2.6.1 一些常用的字段验证的注解
 - 2.6.2 验证请求体(RequestBody)
 - 2.6.3 验证请求参数(Path Variables 和 Request Parameters)
 - 2.7 全局处理 Controller 层异常
 - 2.8 JPA 相关
 - 2.8.1 创建表
 - 2.8.2 创建主键
 - 2.8.3 设置字段类型
 - 2.8.4 指定不持久化特定字段
 - 2.8.5 声明大字段
 - 2.8.6 创建枚举类型的字段
 - 2.8.7 增加审计功能
 - 2.8.8 删除/修改数据
 - 2.8.9 关联关系
 - 2.9 事务 `@Transactional`
 - 2.10 json 数据处理
 - 2.10.1 过滤 json 数据
 - 2.10.2 格式化 json 数据
 - 2.10.3 扁平化对象
 - 2.11 测试相关
- 3. Spring事务
 - 3.1 什么是事务?
 - 3.2 事物的特性 (ACID) 了解么?
 - 3.3 详谈 Spring 对事务的支持
 - 3.3.1. Spring 支持两种方式的事务管理
 - 3.3.2 Spring 事务管理接口介绍
 - 3.3.3 事务属性详解
 - 3.3.4 `@Transactional` 注解使用详解
 - 4. Spring IOC和 AOP详解
 - 4.1 什么是 IOC
 - 4.1.1 为什么叫控制反转
 - 4.2 IOC 解决了什么问题
 - 4.3 IOC 和 DI 的区别
 - 4.4 什么是AOP
 - 4.5 AOP解决了什么问题
 - 4.6 AOP为什么叫面向切面编程
 - 5. Spring中 Bean 的作用域与生命周期
 - 5.1 前言
 - 5.2 bean的作用域
 - 5.2.1 singleton——唯一 bean 实例
 - 5.2.2 prototype——每次请求都会创建一个新的 bean 实例

- 5.2.3 request——每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP request内有效
- 5.2.4 session——每一次HTTP请求都会产生一个新的 bean，该bean仅在当前 HTTP session 内有效
- 5.2.5 globalSession
- 5.3 bean的生命周期
 - 5.3.1 initialization 和 destroy
 - 5.3.2 实现*Aware接口 在Bean中使用Spring框架的一些对象
 - 5.3.3 BeanPostProcessor
 - 5.3.4 总结
 - 5.3.5 单例管理的对象
 - 5.3.6 非单例管理的对象
- 6. SpringMVC 工作原理详解
 - 6.1 先来看一下什么是 MVC 模式
 - 6.2 SpringMVC 简单介绍
 - 6.3 SpringMVC 使用
 - 6.4 SpringMVC 工作原理（重要）
 - 6.5 SpringMVC 重要组件说明
 - 6.6 DispatcherServlet详细解析
- 7. Spring中都用到了那些设计模式?
 - 7.1 控制反转(IoC)和依赖注入(DI)
 - 7.2 工厂设计模式
 - 7.3 单例设计模式
 - 7.4 代理设计模式
 - 7.4.1 代理模式在 AOP 中的应用
 - 7.4.2 Spring AOP 和 AspectJ AOP 有什么区别?
 - 7.5 模板方法
 - 7.6 观察者模式
 - 7.6.1 Spring 事件驱动模型中的三种角色
 - 7.6.2 Spring 的事件流程总结
 - 7.7 适配器模式
 - 7.7.1 spring AOP中的适配器模式
 - 7.7.2 spring MVC中的适配器模式
 - 7.8 装饰者模式
 - 7.9 总结
- (三). 认证授权(JWT、SSO)
 - 1. JWT 身份认证优缺点分析以及常见问题解决方案
 - 1.1 Token 认证的优势
 - 1.1.1 无状态
 - 1.1.2 有效避免了CSRF 攻击
 - 1.1.3 适合移动端应用
 - 1.1.4 单点登录友好
 - 1.2 Token 认证常见问题以及解决办法
 - 1.2.1 注销登录等场景下 token 还有效
 - 1.2.2 token 的续签问题
 - 1.3 总结
 - 2. SSO 单点登录
 - 2.1 前言
 - 2.1.1 SSO说明
 - 2.1.2 单点登录系统的好处
 - 2.1.3 设计目标
 - 2.2 SSO设计与实现
 - 2.2.1 核心应用与依赖
 - 2.2.2 用户登录状态的存储与校验
 - 2.2.3 用户登录/登录校验
 - 2.2.4 用户登出
 - 2.2.5 跨域登录、登出
 - 2.3 备注

(四). 分布式

1. 分布式相关概念入门

- 1.1 分布式系统的经典基础理论
- 1.2 分布式事务
- 1.3 一致性协议/算法
- 1.4 分布式存储
- 1.5 分布式计算

2. Dubbo

2.1 重要的概念

- 2.1.1 什么是 Dubbo?
- 2.1.2 什么是 RPC?RPC原理是什么?
- 2.1.3 为什么要用 Dubbo?
- 2.1.4 什么是分布式?
- 2.1.5 为什么要分布式?

2.2 Dubbo 的架构

- 2.2.1 Dubbo 的架构图解
- 2.2.2 Dubbo 工作原理

2.3 Dubbo 的负载均衡策略

- 2.3.1 先来解释一下什么是负载均衡
- 2.3.2 再来看看 Dubbo 提供的负载均衡策略
- 2.3.3 配置方式

2.4 zookeeper宕机与dubbo直连的情况

3.消息队列其实很简单

3.1 什么是消息队列

3.2 为什么要用消息队列

- 3.2.1 通过异步处理提高系统性能（削峰、减少响应所需时间）
- 3.2.2 降低系统耦合性

3.3 使用消息队列带来的一些问题

3.4 JMS VS AMQP

- 3.4.1 JMS
- 3.4.2 AMQP
- 3.4.3 JMS vs AMQP

3.5 常见的消息队列对比

4. RabbitMQ

4.1 RabbitMQ 介绍

- 4.1.1 RabbitMQ 简介
- 4.1.2 RabbitMQ 核心概念

4.2 安装 RabbitMq

- 4.2.1 安装 erlang
- 4.2.2 安装 RabbitMQ

5. RocketMQ

5.1 消息队列扫盲

- 5.1.1 消息队列为什么会出现?
- 5.1.2 消息队列能用来干什么?

5.2 RocketMQ是什么?

5.3 队列模型和主题模型

- 5.3.1 队列模型
- 5.3.2 主题模型
- 5.3.3 RocketMQ中的消息模型

5.4 RocketMQ的架构图

5.5 如何解决 顺序消费、重复消费

- 5.5.1 顺序消费
- 5.5.2 重复消费

5.6 分布式事务

5.7 消息堆积问题

5.8 回溯消费

5.9 RocketMQ 的刷盘机制

- 5.9.1 同步刷盘和异步刷盘

- 5.9.2 同步复制和异步复制
- 5.9.3 存储机制
- 5.10 总结
- 6. Kafka
 - 6.1 Kafka 简介
 - 6.1.1 Kafka 创建背景
 - 6.1.2 Kafka 简介
 - 6.1.3 Kafka 基础概念
 - 6.2 Kafka 的设计与实现
 - 6.2.1 讨论一：Kafka 存储在文件系统上
 - 6.2.2 讨论二：Kafka 中的底层存储设计
 - 6.2.3 讨论三：生产者设计概要
 - 6.2.4 讨论四：消费者设计概要
 - 6.2.5 讨论五：Kafka 如何保证可靠性
 - 6.3 动手搭一个 Kafka
 - 6.3.1 第一步：下载 Kafka
 - 6.3.2 第二步：启动服务
 - 6.3.3 第三步：发送消息
- 7. API网关
 - 7.1 背景
 - 7.1.1 什么是API网关
 - 7.1.2 为什么需要API网关
 - 7.1.3 统一API网关
 - 7.2 统一网关的设计
 - 7.2.1 异步化请求
 - 7.2.2 链式处理
 - 7.2.3 业务隔离
 - 7.2.4 请求限流
 - 7.2.5 熔断降级
 - 7.2.6 泛化调用
 - 7.2.7 管理平台
 - 7.3 总结
- 8. 分布式ID
 - 8.1 数据库自增ID
 - 8.2 数据库多主模式
 - 8.3 号段模式
 - 8.4 雪花算法
 - 8.5 百度 (uid-generator)
 - 8.6 美团 (Leaf)
 - 8.7 总结
 - 8.8 Redis
- 9. 限流的算法有哪些?
 - 9.1 固定窗口计数器算法
 - 9.2 滑动窗口计数器算法
 - 9.3 漏桶算法
 - 9.4 令牌桶算法
- 10. Zookeeper
 - 10.1 前言
 - 10.2 什么是 ZooKeeper
 - 10.2.1 ZooKeeper 的由来
 - 10.2.2 ZooKeeper 概览
 - 10.2.3 结合个人使用情况的讲一下 ZooKeeper
 - 10.3 关于 ZooKeeper 的一些重要概念
 - 10.3.1 重要概念总结
 - 10.3.2 会话 (Session)
 - 10.3.3 Znode
 - 10.3.4 版本
 - 10.3.5 Watcher

10.3.6 ACL

10.4 ZooKeeper 特点

10.5 ZooKeeper 设计目标

10.5.1 简单的数据模型

10.5.2 可构建集群

10.5.3 顺序访问

10.5.4 高性能

10.6 ZooKeeper 集群角色介绍

10.7 ZooKeeper &ZAB 协议&Paxos算法

10.7.1 ZAB 协议&Paxos算法

10.7.2 ZAB 协议介绍

10.7.3 ZAB 协议两种基本的模式：崩溃恢复和消息广播

10.8 总结

(五). 大型网站架构

1. 如何设计一个高可用系统？要考虑哪些地方？

1.1 什么是高可用？可用性的判断标准是啥？

1.2 哪些情况会导致系统不可用？

1.3 有哪些提高系统可用性的方法？

1.3.1 注重代码质量，测试严格把关

1.3.2 使用集群，减少单点故障

1.3.3 限流

1.3.4 超时和重试机制设置

1.3.5 熔断机制

1.3.6 异步调用

1.3.7 使用缓存

1.3.8 其他

1.4 总结

(六). 微服务

1. Spring Cloud

1.1 什么是Spring cloud

1.2 Spring Cloud 的版本

1.3 Spring Cloud 的服务发现框架——Eureka

1.4 负载均衡之 Ribbon

1.4.1 什么是 `RestTemplate`？

1.4.2 为什么需要 Ribbon？

1.4.3 Nginx 和 Ribbon 的对比

1.4.4 Ribbon 的几种负载均衡算法

1.5 什么是 Open Feign

1.6 必不可少的 Hystrix

1.6.1 什么是 Hystrix之熔断和降级

1.6.2 什么是Hystrix之其他

1.7 微服务网关——Zuul

1.7.1 Zuul 的路由功能

1.7.2 Zuul 的过滤功能

1.7.3 关于 Zuul 的其他

1.7.4 为什么要使用进行配置管理？

1.7.5 Config 是什么

1.8 引出 Spring Cloud Bus

1.9 总结

七、必会工具

(一). Git

1. 版本控制

1.1 什么是版本控制

1.2 为什么要版本控制

1.3 本地版本控制系统

1.4 集中化的版本控制系统

1.5 分布式版本控制系统

2. 认识 Git

- 2.1 Git 简史
- 2.2 Git 与其他版本管理系统的主要区别
- 2.3 Git 的三种状态
- 3. Git 使用快速入门
 - 3.1 获取 Git 仓库
 - 3.2 记录每次更新到仓库
 - 3.3 一个好的 Git 提交消息
 - 3.4 推送改动到远程仓库
 - 3.5 远程仓库的移除与重命名
 - 3.6 查看提交历史
 - 3.7 撤销操作
 - 3.8 分支

(二). Docker

- 1. 认识容器
 - 1.1 什么是容器?
 - 1.1.1 先来看看容器较为官方的解释
 - 1.1.2 再来看看容器较为通俗的解释
 - 1.2 图解物理机,虚拟机与容器
- 2. 再来谈谈 Docker 的一些概念
 - 2.1 什么是 Docker?
 - 2.2 Docker 思想
 - 2.3 Docker 容器的特点
 - 2.4 为什么要用 Docker ?
- 3. 容器 VS 虚拟机
 - 3.1 两者对比图
 - 3.2 容器与虚拟机总结
 - 3.3 容器与虚拟机两者是可以共存的
- 4. Docker 基本概念
 - 4.1 镜像(Image):一个特殊的文件系统
 - 4.2 容器(Container):镜像运行时的实体
 - 4.3 仓库(Repository):集中存放镜像文件的地方
- 5. 常见命令
 - 5.1 基本命令
 - 5.2 拉取镜像
 - 5.3 删除镜像
- 6. Build Ship and Run
- 7. 简单了解一下 Docker 底层原理
 - 7.1 虚拟化技术
 - 7.2 Docker 基于 LXC 虚拟容器技术
- 8. 总结

八、面试指南

- (一) . 程序员简历该怎么写
 - 1. 为什么说简历很重要?
 - 1.1 先从面试前来说
 - 1.2 再从面试中来说
 - 2. 下面这几点你必须知道
 - 3. 必须了解的两大法则
 - 3.1 STAR法则 (Situation Task Action Result)
 - 3.2 FAB 法则 (Feature Advantage Benefit)
 - 4. 项目经历怎么写?
 - 5. 专业技能该怎么写?
 - 6. 排版注意事项
 - 7. 其他的一些小tips
- (二) . 如何准备面试
 - 1. 如何获取大厂面试机会?
 - 2. 面试前的准备
 - 2.1 准备自己的自我介绍
 - 2.2 搞清楚技术面可能会问哪些方向的问题

- 2.3 休闲着装即可
 - 2.4 随身带上自己的成绩单和简历
 - 2.5 如果需要笔试就提前刷一些笔试题
 - 2.6 花时间一些逻辑题
 - 2.7 准备好自己的项目介绍
 - 2.8 提前准备技术面试
 - 2.9 面试之前做好定向复习
- 3. 面试之后复盘
 - 4. 如何学习?学会各种框架有必要吗?
 - 4.1 我该如何学习?
 - 4.2 学会各种框架有必要吗?
- (三) .Java学习路线和方法推荐
- 1. Java 基础
 - 2. 操作系统与计算机网络
 - 3. 数据结构与算法
 - 4. 前端知识
 - 5. MySQL
 - 6. 常用工具
 - 7. 常用框架
 - 8. 多线程的简单使用
 - 9. 分布式
 - 10. 深入学习
 - 11. 微服务
 - 12. 总结

一、Java

(一). 基础

1. Java 基本功

1.1. Java 入门 (基础概念与常识)

1.1.1. Java 语言有哪些特点?

- 1. 简单易学;
- 2. 面向对象 (封装, 继承, 多态) ;
- 3. 平台无关性 (Java 虚拟机实现平台无关性) ;
- 4. 可靠性;
- 5. 安全性;
- 6. 支持多线程 (C++ 语言没有内置的多线程机制, 因此必须调用操作系统的多线程功能来进行多线程程序设计, 而 Java 语言却提供了多线程支持) ;
- 7. 支持网络编程并且很方便 (Java 语言诞生本身就是为简化网络编程设计的, 因此 Java 语言不仅支持网络编程而且很方便) ;
- 8. 编译与解释并存;

1.1.2. 关于 JVM JDK 和 JRE 最详细通俗的解答

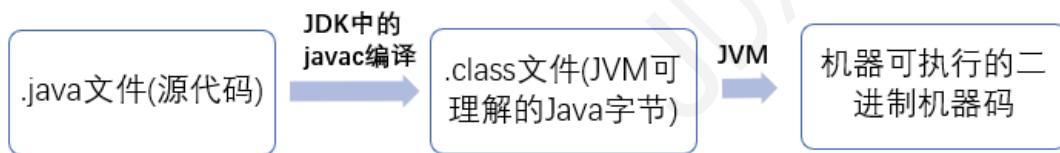
1.1.2.1. JVM

Java 虚拟机 (JVM) 是运行 Java 字节码的虚拟机。JVM 有针对不同系统的特定实现 (Windows, Linux, macOS) , 目的是使用相同的字节码, 它们都会给出相同的结果。

什么是字节码?采用字节码的好处是什么?

在 Java 中, JVM 可以理解的代码就叫做 **字节码** (即扩展名为 `.class` 的文件), 它不面向任何特定的处理器, 只面向虚拟机。Java 语言通过字节码的方式, 在一定程度上解决了传统解释型语言执行效率低的问题, 同时又保留了解释型语言可移植的特点。所以 Java 程序运行时比较高效, 而且, 由于字节码并不针对一种特定的机器, 因此, Java 程序无须重新编译便可在多种不同操作系统的计算机上运行。

Java 程序从源代码到运行一般有下面 3 步:



我们需要格外注意的是 `.class`->`机器码` 这一步。在这一步 JVM 类加载器首先加载字节码文件, 然后通过解释器逐行解释执行, 这种方式的执行速度会相对比较慢。而且, 有些方法和代码块是经常需要被调用的(也就是所谓的热点代码), 所以后面引进了 JIT 编译器, 而 JIT 属于运行时编译。当 JIT 编译器完成第一次编译后, 其会将字节码对应的机器码保存下来, 下次可以直接使用。而我们知道, 机器码的运行效率肯定是高于 Java 解释器的。这也解释了我们为什么经常会说 Java 是编译与解释共存的语言。

HotSpot 采用了惰性评估(Lazy Evaluation)的做法, 根据二八定律, 消耗大部分系统资源的只有那一小部分的代码(热点代码), 而这也正是 JIT 所需要编译的部分。JVM 会根据代码每次被执行的情况收集信息并相应地做出一些优化, 因此执行的次数越多, 它的速度就越快。JDK 9 引入了一种新的编译模式 AOT(Ahead of Time Compilation), 它是直接将字节码编译成机器码, 这样就避免了 JIT 预热等各方面的开销。JDK 支持分层编译和 AOT 协作使用。但是, AOT 编译器的编译质量是肯定比不上 JIT 编译器的。

总结:

Java 虚拟机 (JVM) 是运行 Java 字节码的虚拟机。JVM 有针对不同系统的特定实现 (Windows, Linux, macOS), 目的是使用相同的字节码, 它们都会给出相同的结果。字节码和不同系统的 JVM 实现是 Java 语言“一次编译, 随处可以运行”的关键所在。

1.1.2.2. JDK 和 JRE

JDK 是 Java Development Kit, 它是功能齐全的 Java SDK。它拥有 JRE 所拥有的一切, 还有编译器 (`javac`) 和工具 (如 `javadoc` 和 `jdb`)。它能够创建和编译程序。

JRE 是 Java 运行时环境。它是运行已编译 Java 程序所需的所有内容的集合, 包括 Java 虚拟机 (JVM), Java 类库, `java` 命令和其他的一些基础构件。但是, 它不能用于创建新程序。

如果你只是为了运行一下 Java 程序的话, 那么你只需要安装 JRE 就可以了。如果你需要进行一些 Java 编程方面的工作, 那么你就需要安装 JDK 了。但是, 这不是绝对的。有时, 即使您不打算在计算机上进行任何 Java 开发, 仍然需要安装 JDK。例如, 如果要使用 JSP 部署 Web 应用程序, 那么从技术上讲, 您只是在应用程序服务器中运行 Java 程序。那你为什么需要 JDK 呢? 因为应用程序服务器会将 JSP 转换为 Java servlet, 并且需要使用 JDK 来编译 servlet。

1.1.3. Oracle JDK 和 OpenJDK 的对比

可能在看这个问题之前很多人和我一样并没有接触和使用过 OpenJDK。那么 Oracle 和 OpenJDK 之间是否存在重大差异? 下面我通过收集到的一些资料, 为你解答这个被很多人忽视的问题。

对于 Java 7, 没什么关键的地方。OpenJDK 项目主要基于 Sun 捐赠的 HotSpot 源代码。此外, OpenJDK 被选为 Java 7 的参考实现, 由 Oracle 工程师维护。关于 JVM, JDK, JRE 和 OpenJDK 之间的区别, Oracle 博客帖子在 2012 年有一个更详细的答案:

问：OpenJDK 存储库中的源代码与用于构建 Oracle JDK 的代码之间有什么区别？

答：非常接近 - 我们的 Oracle JDK 版本构建过程基于 OpenJDK 7 构建，只添加了几个部分，例如部署代码，其中包括 Oracle 的 Java 插件和 Java WebStart 的实现，以及一些封闭的源代码派对组件，如图形光栅化器，一些开源的第三方组件，如 Rhino，以及一些零碎的东西，如附加文档或第三方字体。展望未来，我们的目的是开源 Oracle JDK 的所有部分，除了我们考虑商业功能的部分。

总结：

1. Oracle JDK 大概每 6 个月发一次主要版本，而 OpenJDK 版本大概每三个月发布一次。但这不是固定的，我觉得了解这个没啥用处。
2. OpenJDK 是一个参考模型并且是完全开源的，而 Oracle JDK 是 OpenJDK 的一个实现，并不是完全开源的；
3. Oracle JDK 比 OpenJDK 更稳定。OpenJDK 和 Oracle JDK 的代码几乎相同，但 Oracle JDK 有更多的类和一些错误修复。因此，如果您想开发企业/商业软件，我建议您选择 Oracle JDK，因为它经过了彻底的测试和稳定。某些情况下，有些人提到在使用 OpenJDK 可能会遇到了许多应用程序崩溃的问题，但是，只需切换到 Oracle JDK 就可以解决问题；
4. 在响应性和 JVM 性能方面，Oracle JDK 与 OpenJDK 相比提供了更好的性能；
5. Oracle JDK 不会为即将发布的版本提供长期支持，用户每次都必须通过更新到最新版本获得支持来获取最新版本；
6. Oracle JDK 根据二进制代码许可协议获得许可，而 OpenJDK 根据 GPL v2 许可获得许可。

1.1.4. Java 和 C++的区别？

我知道很多人没学过 C++，但是面试官就是没事喜欢拿咱们 Java 和 C++ 比呀！没办法！！！就算没学过 C++，也要记下来！

- 都是面向对象的语言，都支持封装、继承和多态
- Java 不提供指针来直接访问内存，程序内存更加安全
- Java 的类是单继承的，C++ 支持多重继承；虽然 Java 的类不可以多继承，但是接口可以多继承。
- Java 有自动内存管理机制，不需要程序员手动释放无用内存
- 在 C 语言中，字符串或字符数组最后都会有一个额外的字符'\0'来表示结束。但是，Java 语言中没有结束符这一概念。

1.1.5. 什么是 Java 程序的主类 应用程序和小程序的主类有何不同？

一个程序中可以有多个类，但只能有一个类是主类。在 Java 应用程序中，这个主类是指包含 main () 方法的类。而在 Java 小程序中，这个主类是一个继承自系统类 JApplet 或 Applet 的子类。应用程序的主类不一定要求是 public 类，但小程序的主类要求必须是 public 类。主类是 Java 程序执行的入口点。

1.1.6. Java 应用程序与小程序之间有哪些差别？

简单说应用程序是从主线程启动(也就是 `main()` 方法)。applet 小程序没有 `main()` 方法，主要是嵌在浏览器页面上运行(调用 `init()` 或者 `run()` 来启动)，嵌入浏览器这点跟 flash 的小游戏类似。

1.1.7. import java 和 javax 有什么区别？

刚开始的时候 JavaAPI 所必需的包是 `java` 开头的包，`javax` 当时只是扩展 API 包来使用。然而随着时间的推移，`javax` 逐渐地扩展成为 Java API 的组成部分。但是，将扩展从 `javax` 包移动到 `java` 包确实太麻烦了，最终会破坏一堆现有的代码。因此，最终决定 `javax` 包将成为标准 API 的一部分。

所以，实际上 `java` 和 `javax` 没有区别。这都是一个名字。

1.1.8. 为什么说 Java 语言“编译与解释并存”？

高级编程语言按照程序的执行方式分为编译型和解释型两种。简单来说，编译型语言是指编译器针对特定的操作系统将源代码一次性翻译成可被该平台执行的机器码；解释型语言是指解释器对源程序逐行解释成特定平台的机器码并立即执行。比如，你想阅读一本英文名著，你可以找一个英文翻译人员帮助你阅读，

有两种选择方式，你可以先等翻译人员将全本的英文名著（也就是源码）都翻译成汉语，再去阅读，也可以让翻译人员翻译一段，你在旁边阅读一段，慢慢把书读完。

Java 语言既具有编译型语言的特征，也具有解释型语言的特征，因为 Java 程序要经过先编译，后解释两个步骤，由 Java 编写的程序需要先经过编译步骤，生成字节码 (*.class 文件)，这种字节码必须由 Java 解释器来解释执行。因此，我们可以认为 Java 语言编译与解释并存。

1.2. Java 语法

1.2.1. 字符型常量和字符串常量的区别？

- 形式上：字符常量是单引号引起的一个字符；字符串常量是双引号引起的若干个字符
- 含义上：字符常量相当于一个整型值（ASCII 值），可以参加表达式运算；字符串常量代表一个地址值（该字符串在内存中存放位置）
- 占内存大小：字符常量只占 2 个字节；字符串常量占若干个字节（注意：char 在 Java 中占两个字节）

Java 要确定每种基本类型所占存储空间的大小。它们的大小并不像其他大多数语言那样随机器硬件架构的变化而变化。这种所占存储空间大小的不变性是 Java 程序比用其他大多数语言编写的程序更具可移植性的原因之一。

基本类型	大小	最小值	最大值	包装器类型
boolean	—	—	—	Boolean
char	16-bit	Unicode 0	Unicode $2^{16}-1$	Character
byte	8 bits	-128	+127	Byte
short	16 bits	-2^{15}	$+2^{15}-1$	Short
int	32 bits	-2^{31}	$+2^{31}-1$	Integer
long	64 bits	-2^{63}	$+2^{63}-1$	Long
float	32 bits	IEEE754	IEEE754	Float
double	64 bits	IEEE754	IEEE754	Double
void	—	—	—	Void

1.2.2. 关于注释？

Java 中的注释有三种：

- 单行注释
- 多行注释
- 文档注释。

在我们编写代码的时候，如果代码量比较少，我们自己或者团队其他成员还可以很轻易地看懂代码，但是当项目结构一旦复杂起来，我们就需要用到注释了。注释并不会执行，是我们程序员写给自己看的，注释是你的代码说明书，能够帮助看代码的人快速地理清代码之间的逻辑关系。因此，在写程序的时候随手加上注释是一个非常好的习惯。

《Clean Code》这本书明确指出：

代码的注释不是越详细越好。实际上好的代码本身就是注释，我们要尽量规范和美化自己的代码来减少不必要的注释。

若编程语言足够有表达力，就不需要注释，尽量通过代码来阐述。

举个例子：

去掉下面复杂的注释，只需要创建一个与注释所言同一事物的函数即可

```
// check to see if the employee is eligible for full benefits  
if ((employee.flags & HOURLY_FLAG) && (employee.age > 65))
```

应替换为

```
if (employee.isEligibleForFullBenefits())
```

1.2.3. 标识符和关键字的区别是什么？

在我们编写程序的时候，需要大量地为程序、类、变量、方法等取名字，于是就有了标识符，简单来说，标识符就是一个名字。但是有一些标识符，Java 语言已经赋予了其特殊的含义，只能用于特定的地方，这种特殊的标识符就是关键字。因此，关键字是被赋予特殊含义的标识符。比如，在我们的日常生活中，“警察局”这个名字已经被赋予了特殊的含义，所以如果你开一家店，店的名字不能叫“警察局”，“警察局”就是我们日常生活中的关键字。

1.2.4. Java中有哪些常见的关键字？

访问控制	private	protected	public					
类、方法和变量修饰符	abstract	class	extends	final	implements	interface	native	
	new	static	strictfp	synchronized	transient	volatile		
程序控制	break	continue	return	do	while	if	else	
	for	instanceof	switch	case	default			
错误处理	try	catch	throw	throws	finally			
包相关	import	package						
基本类型	boolean	byte	char	double	float	int	long	
	short	null	true	false				
变量引用	super	this	void					
保留字	goto	const						

1.2.5. 自增自减运算符

在写代码的过程中，常见的一种情况是需要某个整数类型变量增加 1 或减少 1，Java 提供了一种特殊的运算符，用于这种表达式，叫做自增运算符（++）和自减运算符（--）。

++和--运算符可以放在操作数之前，也可以放在操作数之后，当运算符放在操作数之前时，先自增/减，再赋值；当运算符放在操作数之后时，先赋值，再自增/减。例如，当“b=++a”时，先自增（自己增加 1），再赋值（赋值给 b）；当“b=a++”时，先赋值（赋值给 b），再自增（自己增加 1）。也就是，++a 输出的是 a+1 的值，a++输出的是 a 值。用一句口诀就是：“符号在前就先加/减，符号在后就后加/减”。

1.2.6. continue、break、和return的区别是什么？

在循环结构中，当循环条件不满足或者循环次数达到要求时，循环会正常结束。但是，有时候可能需要在循环的过程中，当发生了某种条件之后，提前终止循环，这就需要用到下面几个关键词：

1. continue：指跳出当前的这一次循环，继续下一次循环。
2. break：指跳出整个循环体，继续执行循环下面的语句。

return 用于跳出所在方法，结束该方法的运行。return 一般有两种用法：

1. return;：直接使用 return 结束方法执行，用于没有返回值函数的方法

2. `return value;` : return 一个特定值，用于有返回值函数的方法

1.2.7. Java泛型了解么？什么是类型擦除？介绍一下常用的通配符？

Java 泛型 (generics) 是 JDK 5 中引入的一个新特性, 泛型提供了编译时类型安全检测机制, 该机制允许程序员在编译时检测到非法的类型。泛型的本质是参数化类型, 也就是说所操作的数据类型被指定为一个参数。

Java的泛型是伪泛型，这是因为Java在编译期间，所有的泛型信息都会被擦掉，这也就是通常所说类型擦除。

```
List<Integer> list = new ArrayList<>();  
  
list.add(12);  
//这里直接添加会报错  
list.add("a");  
Class<? extends List> clazz = list.getClass();  
Method add = clazz.getDeclaredMethod("add", Object.class);  
//但是通过反射添加，是可以的  
add.invoke(list, "k1");  
  
System.out.println(list)
```

泛型一般有三种使用方式:泛型类、泛型接口、泛型方法。

1.泛型类：

```
//此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型  
//在实例化泛型类时，必须指定T的具体类型  
public class Generic<T>{  
  
    private T key;  
  
    public Generic(T key) {  
        this.key = key;  
    }  
  
    public T getKey(){  
        return key;  
    }  
}
```

如何实例化泛型类：

```
Generic<Integer> genericInteger = new Generic<Integer>(123456);
```

2.泛型接口：

```
public interface Generator<T> {  
    public T method();  
}
```

实现泛型接口，不指定类型：

```
class GeneratorImpl<T> implements Generator<T>{
    @Override
    public T method() {
        return null;
    }
}
```

实现泛型接口，指定类型：

```
class GeneratorImpl<T> implements Generator<String>{
    @Override
    public String method() {
        return "hello";
    }
}
```

3. 泛型方法：

```
public static < E > void printArray( E[] inputArray )
{
    for ( E element : inputArray ){
        System.out.printf( "%s ", element );
    }
    System.out.println();
}
```

使用：

```
// 创建不同类型数组： Integer, Double 和 Character
Integer[] intArray = { 1, 2, 3 };
String[] stringArray = { "Hello", "world" };
printArray( intArray );
printArray( stringArray );
```

常用的通配符为： T, E, K, V, ?

- ? 表示不确定的 java 类型
- T (type) 表示具体的一个java类型
- K V (key value) 分别代表java键值中的Key Value
- E (element) 代表Element

1.2.8. ==和equals的区别

== : 它的作用是判断两个对象的地址是不是相等。即判断两个对象是不是同一个对象。**(基本数据类型 ==比较的是值，引用数据类型==比较的是内存地址)**

因为 Java 只有值传递，所以，对于 == 来说，不管是比较基本数据类型，还是引用数据类型的变量，其本质比较的都是值，只是引用类型变量存的值是对象的地址。

equals() : 它的作用也是判断两个对象是否相等，它不能用于比较基本数据类型的变量。equals() 方法存在于 object 类中，而 object 类是所有类的直接或间接父类。

object 类 equals() 方法：

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

`equals()` 方法存在两种使用情况：

- 情况 1：类没有覆盖 `equals()` 方法。则通过 `equals()` 比较该类的两个对象时，等价于通过“`==`”比较这两个对象。使用的默认是 `Object` 类 `equals()` 方法。
- 情况 2：类覆盖了 `equals()` 方法。一般，我们都覆盖 `equals()` 方法来两个对象的内容相等；若它们的内容相等，则返回 `true`(即，认为这两个对象相等)。

举个例子：

```
public class test1 {  
    public static void main(String[] args) {  
        String a = new String("ab"); // a 为一个引用  
        String b = new String("ab"); // b 为另一个引用，对象的内容一样  
        String aa = "ab"; // 放在常量池中  
        String bb = "ab"; // 从常量池中查找  
        if (aa == bb) // true  
            System.out.println("aa==bb");  
        if (a == b) // false, 非同一对象  
            System.out.println("a==b");  
        if (a.equals(b)) // true  
            System.out.println("aEQb");  
        if (42 == 42.0) { // true  
            System.out.println("true");  
        }  
    }  
}
```

说明：

- `String` 中的 `equals` 方法是被重写过的，因为 `Object` 的 `equals` 方法是比较的对象的内存地址，而 `String` 的 `equals` 方法比较的是对象的值。
- 当创建 `String` 类型的对象时，虚拟机会在常量池中查找有没有已经存在的值和要创建的值相同的对象，如果有就把它赋给当前引用。如果没有就在常量池中重新创建一个 `String` 对象。

`String` 类 `equals()` 方法：

```
public boolean equals(Object anObject) {  
    if (this == anObject)  
        return true;  
    if (anObject instanceof String) {  
        String anotherString = (String)anObject;  
        int n = value.length;  
        if (n == anotherString.value.length) {  
            char v1[] = value;  
            char v2[] = anotherString.value;  
            int i = 0;  
            while (n-- != 0) {  
                if (v1[i] != v2[i])  
                    return false;  
                i++;  
            }  
        }  
    }  
}
```

```
        return true;
    }
}
return false;
}
```

1.2.9. hashCode()与 equals()

面试官可能会问你：“你重写过 hashCode 和 equals 么，为什么重写 equals 时必须重写 hashCode 方法？”

1)hashCode()介绍:

hashCode() 的作用是获取哈希码，也称为散列码；它实际上是返回一个 int 整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。 hashCode() 定义在 JDK 的 Object 类中，这就意味着 Java 中的任何类都包含有 hashCode() 函数。另外需要注意的是： Object 的 hashCode 方法是本地方法，也就是用 c 语言或 c++ 实现的，该方法通常用来将对象的 内存地址 转换为整数之后返回。

```
public native int hashCode();
```

散列表存储的是键值对(key-value)，它的特点是：能根据“键”快速的检索出对应的“值”。这其中就利用到了散列码！（可以快速找到所需要的对象）

2)为什么要有 hashCode?

我们以“ HashSet 如何检查重复”为例子来说明为什么要有 hashCode?

当你把对象加入 HashSet 时， HashSet 会先计算对象的 hashCode 值来判断对象加入的位置，同时也会与其他已经加入的对象的 hashCode 值作比较，如果没有相符的 hashCode， HashSet 会假设对象没有重复出现。但是如果发现有相同 hashCode 值的对象，这时会调用 equals () 方法来检查 hashCode 相等的对象是否真的相同。如果两者相同， HashSet 就不会让其加入操作成功。如果不同的话，就会重新散列到其他位置。（摘自我的 Java 启蒙书《Head fist java》第二版）。这样我们就大大减少了 equals 的次数，相应就大大提高了执行速度。

3)为什么重写 equals 时必须重写 hashCode 方法?

如果两个对象相等，则 hashCode 一定也是相同的。两个对象相等，对两个对象分别调用 equals 方法都返回 true。但是，两个对象有相同的 hashCode 值，它们也不一定是相等的。因此， equals 方法被覆盖，则 hashCode 方法也必须被覆盖。

hashCode() 的默认行为是对堆上的对象产生独特值。如果没有重写 hashCode()，则该 class 的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）

4)为什么两个对象有相同的 hashCode 值，它们也不一定是相等的?

在这里解释一位小伙伴的问题。以下内容摘自《Head Fisrt Java》。

因为 hashCode() 所使用的杂凑算法也许刚好会让多个对象传回相同的杂凑值。越糟糕的杂凑算法越容易碰撞，但这也与数据值域分布的特性有关（所谓碰撞也就是指的是不同的对象得到相同的 hashCode ）。

我们刚刚也提到了 HashSet ,如果 HashSet 在对比的时候，同样的 hashCode 有多个对象，它会使用 equals() 来判断是否真的相同。也就是说 hashCode 只是用来缩小查找成本。

1.3. 基本数据类型

1.3.1. Java中的几种基本数据类型是什么？对应的包装类型是什么？各自占用多少字节呢？

Java中有8种基本数据类型，分别为：

- 6种数字类型：byte、short、int、long、float、double
- 1种字符类型：char
- 1中布尔型：boolean。

这八种基本类型都有对应的包装类分别为：Byte、Short、Integer、Long、Float、Double、Character、Boolean

基本类型	位数	字节	默认值
int	32	4	0
short	16	2	0
long	64	8	0L
byte	8	1	0
char	16	2	'u0000'
float	32	4	0f
double	64	8	0d
boolean	1		false

对于boolean，官方文档未明确定义，它依赖于JVM厂商的具体实现。逻辑上理解是占用1位，但是实际中会考虑计算机高效存储因素。

注意：

- Java里使用long类型的数据一定要在数值后面加上L，否则将作为整型解析：
- char a = 'h' char：单引号，String a = "hello"：双引号

1.3.2. 自动装箱与拆箱

- 装箱：将基本类型用它们对应的引用类型包装起来；
- 拆箱：将包装类型转换为基本数据类型；

1.3.3. 8种基本类型的包装类和常量池

Java基本类型的包装类的大部分都实现了常量池技术，即Byte,Short,Integer,Long,Character[Boolean]；前面4种包装类默认创建了数值[-128, 127]的相应类型的缓存数据，Character创建了数值在[0,127]范围的缓存数据，Boolean直接返回True Or False。如果超出对应范围仍然会去创建新的对象。

```
public static Boolean valueof(boolean b) {
    return (b ? TRUE : FALSE);
}
```

```
private static class CharacterCache {
    private CharacterCache() {}

    static final Character cache[] = new Character[127 + 1];
    static {
        for (int i = 0; i < cache.length; i++)
            cache[i] = new Character((char)i);
    }
}
```

两种浮点数类型的包装类 Float,Double 并没有实现常量池技术。**

```
Integer i1 = 33;
Integer i2 = 33;
System.out.println(i1 == i2); // 输出 true
Integer i11 = 333;
Integer i22 = 333;
System.out.println(i11 == i22); // 输出 false
Double i3 = 1.2;
Double i4 = 1.2;
System.out.println(i3 == i4); // 输出 false
```

Integer 缓存源代码:

```
/**
 * 此方法将始终缓存 -128 到 127 (包括端点) 范围内的值，并可以缓存此范围之外的其他值。
 */
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

应用场景:

1. Integer i1=40; Java 在编译的时候会直接将代码封装成 Integer i1=Integer.valueOf(40);, 从而使用常量池中的对象。
2. Integer i1 = new Integer(40);这种情况下会创建新的对象。

```
Integer i1 = 40;
Integer i2 = new Integer(40);
System.out.println(i1==i2); //输出 false
```

Integer 比较更丰富的一个例子:

```
Integer i1 = 40;
Integer i2 = 40;
Integer i3 = 0;
Integer i4 = new Integer(40);
Integer i5 = new Integer(40);
Integer i6 = new Integer(0);

System.out.println("i1=i2 " + (i1 == i2));
System.out.println("i1=i2+i3 " + (i1 == i2 + i3));
System.out.println("i1=i4 " + (i1 == i4));
System.out.println("i4=i5 " + (i4 == i5));
System.out.println("i4=i5+i6 " + (i4 == i5 + i6));
System.out.println("40=i5+i6 " + (40 == i5 + i6));
```

结果：

```
i1=i2 true
i1=i2+i3 true
i1=i4 false
i4=i5 false
i4=i5+i6 true
40=i5+i6 true
```

解释：

语句 `i4 == i5 + i6`，因为`+`这个操作符不适用于 `Integer` 对象，首先 `i5` 和 `i6` 进行自动拆箱操作，进行数值相加，即 `i4 == 40`。然后 `Integer` 对象无法与数值进行直接比较，所以 `i4` 自动拆箱转为 `int` 值 `40`，最终这条语句转为 `40 == 40` 进行数值比较。

1.4. 方法（函数）

1.4.1. 什么是方法的返回值？返回值在类的方法里的作用是什么？

方法的返回值是指我们获取到的某个方法体中的代码执行后产生的结果！（前提是该方法可能产生结果）。返回值的作用是接收出结果，使得它可以用于其他的操作！

1.4.2. 为什么 Java 中只有值传递？

首先回顾一下在程序设计语言中有关将参数传递给方法（或函数）的一些专业术语。**按值调用(call by value)**表示方法接收的是调用者提供的值，而**按引用调用 (call by reference)**表示方法接收的是调用者提供的变量地址。一个方法可以修改传递引用所对应的变量值，而不能修改传递值调用所对应的变量值。它用来描述各种程序设计语言（不只是 Java）中方法参数传递方式。

Java 程序设计语言总是采用按值调用。也就是说，方法得到的是所有参数值的一个拷贝，也就是说，方法不能修改传递给它的任何参数变量的内容。

下面通过 3 个例子来给大家说明

example 1

```
public static void main(String[] args) {
    int num1 = 10;
    int num2 = 20;

    swap(num1, num2);

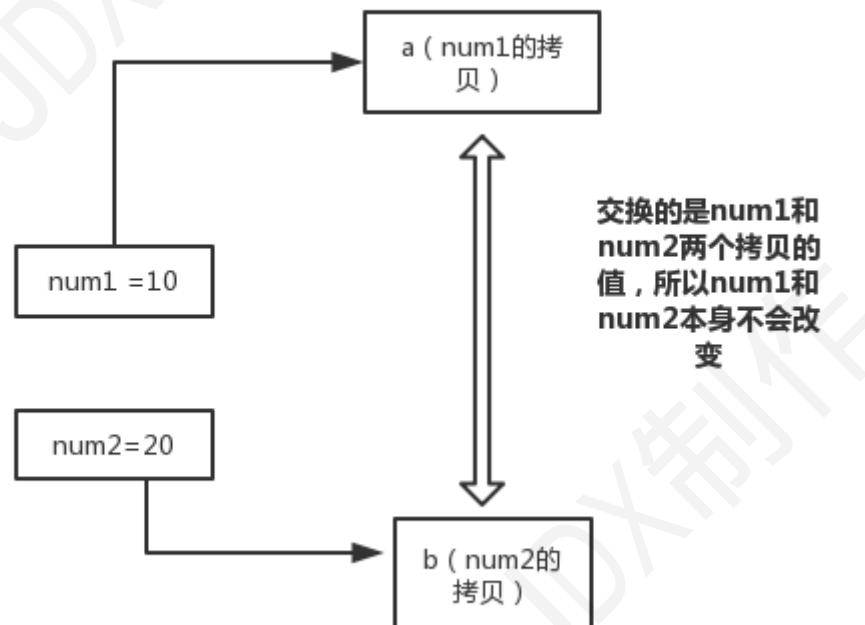
    System.out.println("num1 = " + num1);
```

```
        System.out.println("num2 = " + num2);  
    }  
  
    public static void swap(int a, int b) {  
        int temp = a;  
        a = b;  
        b = temp;  
  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
    }  
}
```

结果：

```
a = 20  
b = 10  
num1 = 10  
num2 = 20
```

解析：



在 swap 方法中，a、b 的值进行交换，并不会影响到 num1、num2。因为，a、b 中的值，只是从 num1、num2 的复制过来的。也就是说，a、b 相当于 num1、num2 的副本，副本的内容无论怎么修改，都不会影响到原件本身。

通过上面例子，我们已经知道了一个方法不能修改一个基本数据类型的参数，而对象引用作为参数就不一样，请看 example2。

| example 2

```

public static void main(String[] args) {
    int[] arr = { 1, 2, 3, 4, 5 };
    System.out.println(arr[0]);
    change(arr);
    System.out.println(arr[0]);
}

public static void change(int[] array) {
    // 将数组的第一个元素变为0
    array[0] = 0;
}

```

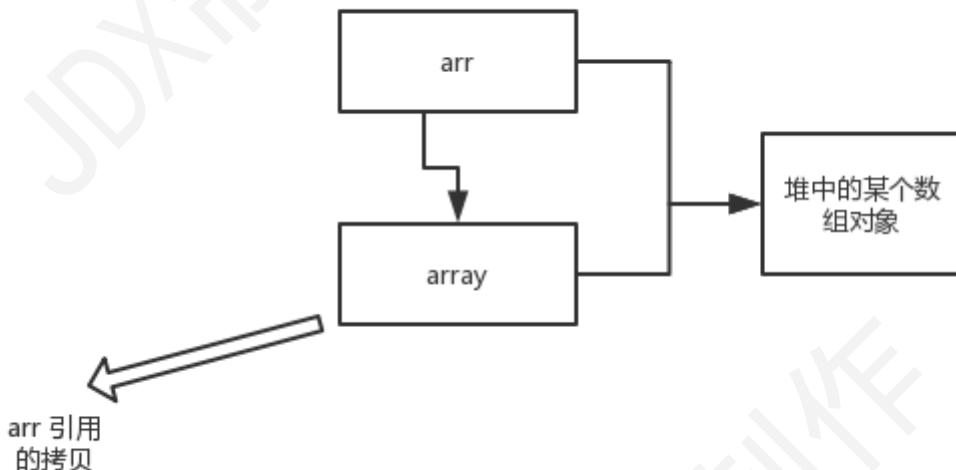
结果：

```

1
0

```

解析：



`array` 被初始化 `arr` 的拷贝也是一个对象的引用，也就是说 `array` 和 `arr` 指向的是同一个数组对象。因此，外部对引用对象的改变会反映到所对应的对象上。

通过 example2 我们已经看到，实现一个改变对象参数状态的方法并不是一件坏事。理由很简单，方法得到的是对象引用的拷贝，对象引用及其他的拷贝同时引用同一个对象。

很多程序设计语言（特别是，C++和 Pascal）提供了两种参数传递的方式：值调用和引用调用。有些程序员（甚至本书的作者）认为 Java 程序设计语言对对象采用的是引用调用，实际上，这种理解是不对的。由于这种误解具有一定的普遍性，所以下面给出一个反例来详细地阐述一下这个问题。

example 3

```

public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Student s1 = new Student("小张");
        Student s2 = new Student("小李");
        Test.swap(s1, s2);
        System.out.println("s1:" + s1.getName());
    }

    public void swap(Student s1, Student s2) {
        Student temp = s1;
        s1 = s2;
        s2 = temp;
    }
}

```

```
        System.out.println("s2:" + s2.getName());
    }

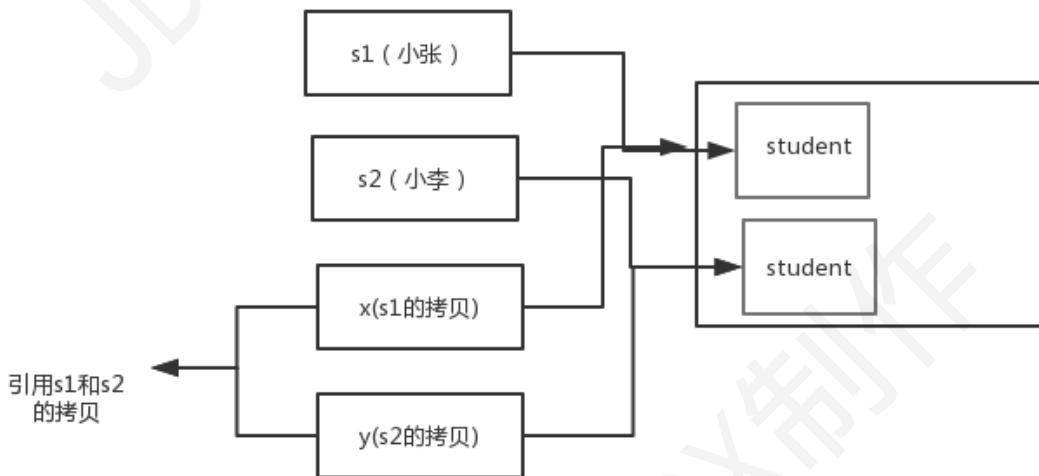
public static void swap(Student x, Student y) {
    Student temp = x;
    x = y;
    y = temp;
    System.out.println("x:" + x.getName());
    System.out.println("y:" + y.getName());
}
}
```

结果：

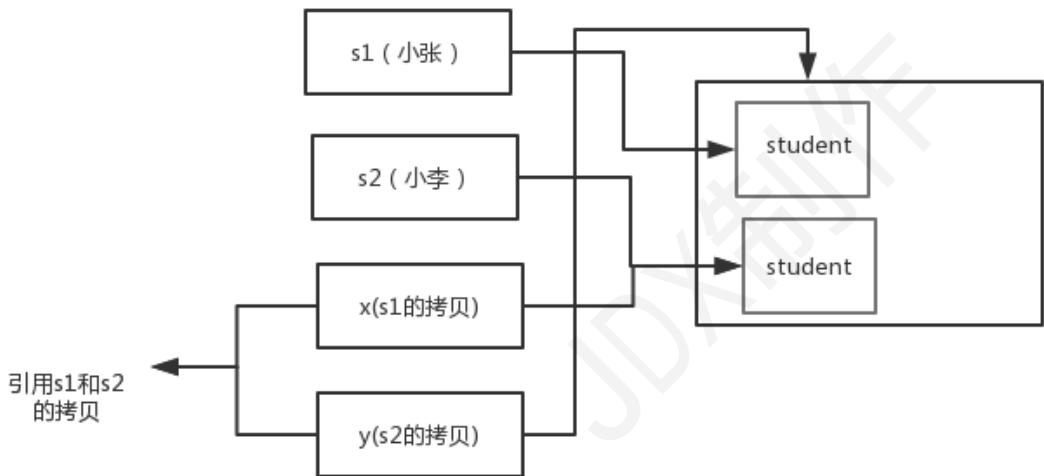
```
x:小李  
y:小张  
s1:小张  
s2:小李
```

解析：

交换之前：



交换之后：



通过上面两张图可以很清晰的看出：方法并没有改变存储在变量 `s1` 和 `s2` 中的对象引用。swap 方法的参数 `x` 和 `y` 被初始化为两个对象引用的拷贝，这个方法交换的是这两个拷贝

总结

Java 程序设计语言对对象采用的不是引用调用，实际上，对象引用是按值传递的。

下面再总结一下 Java 中方法参数的使用情况：

- 一个方法不能修改一个基本数据类型的参数（即数值型或布尔型）。
- 一个方法可以改变一个对象参数的状态。
- 一个方法不能让对象参数引用一个新的对象。

1.4.3. 重载和重写的区别

重载就是同样的一个方法能够根据输入数据的不同，做出不同的处理

重写就是当子类继承自父类的相同方法，输入数据一样，但要做出有别于父类的响应时，你就要覆盖父类方法

1.4.3.1. 重载

发生在同一个类中，方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同。

下面是《Java 核心技术》对重载这个概念的介绍：

4.6.1 重载

有些类有多个构造器。例如，可以如下构造一个空的 `StringBuilder` 对象：

```
StringBuilder messages = new StringBuilder();
```

或者，可以指定一个初始字符串：

```
StringBuilder todoList = new StringBuilder("To do:\n");
```

这种特征叫做**重载** (*overloading*)。如果多个方法（比如，`StringBuilder` 构造器方法）有相同的名字、不同的参数，便产生了**重载**。编译器必须挑选出具体执行哪个方法，它通过用各个方法给出的参数类型与特定方法调用所使用的值类型进行匹配来挑选出相应的方法。如果编译器找不到匹配的参数，就会产生编译时错误，因为根本不存在匹配，或者没有一个比其他的更好。（这个过程被称为**重载解析** (*overloading resolution*)。）

注释：Java 允许**重载**任何方法，而不只是构造器方法。因此，要完整地描述一个方法，需要指出方法名以及参数类型。这叫做方法的签名 (*signature*)。例如，`String` 类有 4 个称为 `indexOf` 的公有方法。它们的签名是

```
indexOf(int)  
indexOf(int, int)  
indexOf(String)  
indexOf(String, int)
```

返回类型不是方法签名的一部分。也就是说，不能有两个名字相同、参数类型也相同却返回不同类型值的方法。

综上：**重载就是同一个类中多个同名方法根据不同的传参来执行不同的逻辑处理。**

1.4.3.2. 重写

重写发生在运行期，是子类对父类的允许访问的方法的实现过程进行重新编写。

1. 返回值类型、方法名、参数列表必须相同，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类。
2. 如果父类方法访问修饰符为 `private/final/static` 则子类就不能重写该方法，但是被 `static` 修饰的方法能够被再次声明。
3. 构造方法无法被重写

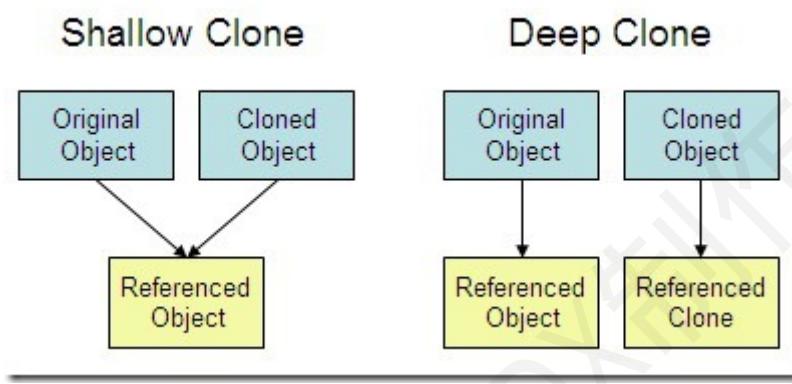
综上：**重写就是子类对父类方法的重新改造，外部样子不能改变，内部逻辑可以改变**

暖心的 Guide 哥最后再来个图标总结一下！

区别点	重载方法	重写方法
发生范围	同一个类	子类中
参数列表	必须修改	一定不能修改
返回类型	可修改	一定不能修改
异常	可修改	可以减少或删除，一定不能抛出新的或者更广的异常
访问修饰符	可修改	一定不能做更严格的限制（可以降低限制）
发生阶段	编译期	运行期

1.4.4. 深拷贝 vs 浅拷贝

1. **浅拷贝**：对基本数据类型进行值传递，对引用数据类型进行引用传递般的拷贝，此为浅拷贝。
2. **深拷贝**：对基本数据类型进行值传递，对引用数据类型，创建一个新的对象，并复制其内容，此为深拷贝。



1.4.5. 方法的四种类型

1、无参数无返回值的方法

```
// 无参数无返回值的方法(如果方法没有返回值, 不能不写, 必须写void, 表示没有返回值)
public void f1() {
    System.out.println("无参数无返回值的方法");
}
```

2、有参数无返回值的方法

```
/**
 * 有参数无返回值的方法
 * 参数列表由零组到多组“参数类型+形参名”组合而成, 多组参数之间以英文逗号 (,) 隔开, 形参类型和形参名之间以英文空格隔开
 */
public void f2(int a, String b, int c) {
    System.out.println(a + "-->" + b + "-->" + c);
}
```

3、有返回值无参数的方法

```
// 有返回值无参数的方法 (返回值可以是任意的类型, 在函数里面必须有return关键字返回对应的类型)
public int f3() {
    System.out.println("有返回值无参数的方法");
    return 2;
}
```

4、有返回值有参数的方法

```
// 有返回值有参数的方法
public int f4(int a, int b) {
    return a * b;
}
```

5、return 在无返回值方法的特殊使用

```
// return在无返回值方法的特殊使用
public void f5(int a) {
    if (a>10) {
        return;//表示结束所在方法（f5方法）的执行，下方的输出语句不会执行
    }
    System.out.println(a);
}
```

2. Java 面向对象

2.1. 类和对象

2.1.1. 面向对象和面向过程的区别

- **面向过程**：面向过程性能比面向对象高。因为类调用时需要实例化，开销比较大，比较消耗资源，所以当性能是最重要的考量因素的时候，比如单片机、嵌入式开发、Linux/Unix 等一般采用面向过程开发。但是，**面向过程没有面向对象易维护、易复用、易扩展**。
- **面向对象**：面向对象易维护、易复用、易扩展。因为面向对象有封装、继承、多态性的特性，所以可以设计出低耦合的系统，使系统更加灵活、更加易于维护。但是，**面向对象性能比面向过程低**。

这个并不是根本原因，面向过程也需要分配内存，计算内存偏移量，Java 性能差的主要原因并不是因为它是面向对象语言，而是 Java 是半编译语言，最终的执行代码并不是可以直接被 CPU 执行的二进制机械码。

而面向过程语言大多都是直接编译成机械码在电脑上执行，并且其它一些面向过程的脚本语言性能也并不一定比 Java 好。

2.1.2. 构造器 Constructor 是否可被 override？

Constructor 不能被 override（重写），但是可以 overload（重载），所以你可以看到一个类中有多个构造函数的情况。

2.1.3. 在 Java 中定义一个不做事且没有参数的构造方法的作用

Java 程序在执行子类的构造方法之前，如果没有用 super() 来调用父类特定的构造方法，则会调用父类中“没有参数的构造方法”。因此，如果父类中只定义了有参数的构造方法，而在子类的构造方法中又没有用 super() 来调用父类中特定的构造方法，则编译时将发生错误，因为 Java 程序在父类中找不到没有参数的构造方法可供执行。解决办法是在父类里加上一个不做事且没有参数的构造方法。

2.1.4. 成员变量与局部变量的区别有哪些？

1. 从语法形式上看：成员变量是属于类的，而局部变量是在方法中定义的变量或是方法的参数；成员变量可以被 public, private, static 等修饰符所修饰，而局部变量不能被访问控制修饰符及 static 所修饰；但是，成员变量和局部变量都能被 final 所修饰。
2. 从变量在内存中的存储方式来看：如果成员变量是使用 static 修饰的，那么这个成员变量是属于类的，如果没有使用 static 修饰，这个成员变量是属于实例的。而对象存在于堆内存，局部变量则存在于栈内存。
3. 从变量在内存中的生存时间上看：成员变量是对象的一部分，它随着对象的创建而存在，而局部变量随着方法的调用而自动消失。
4. 成员变量如果没有被赋初值：则会自动以类型的默认值而赋值（一种情况例外：被 final 修饰的成员变量也必须显式地赋值），而局部变量则不会自动赋值。

2.1.5. 创建一个对象用什么运算符？对象实体与对象引用有何不同？

`new` 运算符，`new` 创建对象实例（对象实例在堆内存中），对象引用指向对象实例（对象引用存放在栈内存中）。一个对象引用可以指向 0 个或 1 个对象（一根绳子可以不系气球，也可以系一个气球）；一个对象可以有 n 个引用指向它（可以用 n 条绳子系住一个气球）。

2.1.6. 一个类的构造方法的作用是什么？若一个类没有声明构造方法，该程序能正确执行吗？为什么？

主要作用是完成对类对象的初始化工作。可以执行。因为一个类即使没有声明构造方法也会有默认的不带参数的构造方法。如果我们自己添加了类的构造方法（无论是否有参），Java 就不会再添加默认的无参数的构造方法了，这时候，就不能直接 `new` 一个对象而不传递参数了，所以我们一直在不知不觉地使用构造方法，这也是为什么我们在创建对象的时候后面要加一个括号（因为要调用无参的构造方法）。如果我们重载了有参的构造方法，记得都要把无参的构造方法也写出来（无论是否用到），因为这可以帮助我们在创建对象的时候少踩坑。

2.1.7. 构造方法有哪些特性？

1. 名字与类名相同。
2. 没有返回值，但不能用 `void` 声明构造函数。
3. 生成类的对象时自动执行，无需调用。

2.1.8. 在调用子类构造方法之前会先调用父类没有参数的构造方法，其目的是？

帮助子类做初始化工作。

2.1.9. 对象的相等与指向他们的引用相等，两者有什么不同？

对象的相等，比的是内存中存放的内容是否相等。而引用相等，比较的是他们指向的内存地址是否相等。

2.2. 面向对象三大特征

2.2.1. 封装

封装是指把一个对象的状态信息（也就是属性）隐藏在对象内部，不允许外部对象直接访问对象的内部信息。但是可以提供一些可以被外界访问的方法来操作属性。就好像我们看不到挂在墙上的空调的内部的零件信息（也就是属性），但是可以通过遥控器（方法）来控制空调。如果属性不想被外界访问，我们大可不必提供方法给外界访问。但是如果一个类没有提供给外界访问的方法，那么这个类也没有什么意义了。就好像如果没有空调遥控器，那么我们就无法操控空调制冷，空调本身就没有意义了（当然现在很多其他方法，这里只是为了举例子）。

```
public class Student {  
    private int id; //id属性私有化  
    private String name; //name属性私有化  
  
    //获取id的方法  
    public int getId() {  
        return id;  
    }  
  
    //设置id的方法  
    public void setId(int id) {  
        this.id = id;  
    }  
  
    //获取name的方法  
    public String getName() {  
        return name;  
    }  
}
```

```
//设置name的方法  
public void setName(String name) {  
    this.name = name;  
}  
}
```

2.2.2. 继承

不同类型的对象，相互之间经常有一定数量的共同点。例如，小明同学、小红同学、小李同学，都共享学生的特性（班级、学号等）。同时，每一个对象还定义了额外的特性使得他们与众不同。例如小明的数学比较好，小红的性格惹人喜爱；小李的力气比较大。继承是使用已存在的类的定义作为基础建立新类的技术，新类的定义可以增加新的数据或新的功能，也可以用父类的功能，但不能选择性地继承父类。通过使用继承，可以快速地创建新的类，可以提高代码的重用，程序的可维护性，节省大量创建新类的时间，提高我们的开发效率。

关于继承如下 3 点请记住：

1. 子类拥有父类对象所有的属性和方法（包括私有属性和私有方法），但是父类中的私有属性和方法子类是无法访问，**只是拥有**。
2. 子类可以拥有自己属性和方法，即子类可以对父类进行扩展。
3. 子类可以用自己的方式实现父类的方法。（以后介绍）。

2.2.3. 多态

多态，顾名思义，表示一个对象具有多种的状态。具体表现为父类的引用指向子类的实例。

多态的特点：

- 对象类型和引用类型之间具有继承（类）/实现（接口）的关系；
- 对象类型不可变，引用类型可变；
- 方法具有多态性，属性不具有多态性；
- 引用类型变量发出的方法调用的到底是哪个类中的方法，必须在程序运行期间才能确定；
- 多态不能调用“只在子类存在但在父类不存在”的方法；
- 如果子类重写了父类的方法，真正执行的是子类覆盖的方法，如果子类没有覆盖父类的方法，执行的是父类的方法。

2.3. 修饰符

2.3.1. 在一个静态方法内调用一个非静态成员为什么是非法的？

由于静态方法可以不通过对象进行调用，因此在静态方法里，不能调用其他非静态变量，也不可以访问非静态变量成员。

2.3.2. 静态方法和实例方法有何不同

1. 在外部调用静态方法时，可以使用“类名.方法名”的方式，也可以使用“对象名.方法名”的方式。而实例方法只有后面这种方式。也就是说，调用静态方法可以无需创建对象。
2. 静态方法在访问本类的成员时，只允许访问静态成员（即静态成员变量和静态方法），而不允许访问实例成员变量和实例方法；实例方法则无此限制。

2.4. 接口和抽象类

2.4.1. 接口和抽象类的区别是什么？

1. 接口的方法默认是 public，所有方法在接口中不能有实现（Java 8 开始接口方法可以有默认实现），而抽象类可以有非抽象的方法。
2. 接口中除了 static、final 变量，不能有其他变量，而抽象类中则不一定。

3. 一个类可以实现多个接口，但只能实现一个抽象类。接口自己本身可以通过 extends 关键字扩展多个接口。
4. 接口方法默认修饰符是 public，抽象方法可以有 public、protected 和 default 这些修饰符（抽象方法就是为了被重写所以不能使用 private 关键字修饰！）。
5. 从设计层面来说，抽象是对类的抽象，是一种模板设计，而接口是对行为的抽象，是一种行为的规范。

备注：

1. 在 JDK8 中，接口也可以定义静态方法，可以直接用接口名调用。实现类和实现是不可以调用的。如果同时实现两个接口，接口中定义了一样的默认方法，则必须重写，不然会报错。
 2. jdk9 的接口被允许定义私有方法。
1. 在 jdk 7 或更早版本中，接口里面只能有常量变量和抽象方法。这些接口方法必须由选择实现接口的类实现。
 2. jdk8 的时候接口可以有默认方法和静态方法功能。
 3. Jdk 9 在接口中引入了私有方法和私有静态方法。

2.5. 其它重要知识点

2.5.1. String StringBuffer 和 StringBuilder 的区别是什么? String 为什么是不可变的?

简单的来说：`String` 类中使用 `final` 关键字修饰字符数组来保存字符串，`private final char value[]`，所以 `String` 对象是不可变的。

而 `StringBuilder` 与 `StringBuffer` 都继承自 `AbstractStringBuilder` 类，在 `AbstractStringBuilder` 中也是使用字符数组保存字符串 `char[] value` 但是没有用 `final` 关键字修饰，所以这两种对象都是可变的。

`StringBuilder` 与 `StringBuffer` 的构造方法都是调用父类构造方法也就是 `AbstractStringBuilder` 实现的，大家可以自行查阅源码。

`AbstractStringBuilder.java`

```
abstract class AbstractStringBuilder implements Appendable, CharSequence {  
    /**  
     * The value is used for character storage.  
     */  
    char[] value;  
  
    /**  
     * The count is the number of characters used.  
     */  
    int count;  
  
    AbstractStringBuilder(int capacity) {  
        value = new char[capacity];  
    }  
}
```

线程安全性

`String` 中的对象是不可变的，也就可以理解为常量，线程安全。`AbstractStringBuilder` 是 `StringBuilder` 与 `StringBuffer` 的公共父类，定义了一些字符串的基本操作，如 `expandCapacity`、`append`、`insert`、`indexOf` 等公共方法。`StringBuffer` 对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。`StringBuilder` 并没有对方法进行加同步锁，所以是非线程安全的。

性能

每次对 `String` 类型进行改变的时候，都会生成一个新的 `String` 对象，然后将指针指向新的 `String` 对象。`StringBuffer` 每次都会对 `StringBuffer` 对象本身进行操作，而不是生成新的对象并改变对象引用。相同情况下使用 `StringBuilder` 相比使用 `StringBuffer` 仅能获得 10%~15% 左右的性能提升，但却要冒多线程不安全的风险。

对于三者使用的总结：

1. 操作少量的数据：适用 `String`
2. 单线程操作字符串缓冲区下操作大量数据：适用 `StringBuilder`
3. 多线程操作字符串缓冲区下操作大量数据：适用 `StringBuffer`

2.5.2. Object 类的常见方法总结

`Object` 类是一个特殊的类，是所有类的父类。它主要提供了以下 11 个方法：

`public final native Class<?> getClass()`//native方法，用于返回当前运行时对象的Class对象，使用了final关键字修饰，故不允许子类重写。

`public native int hashCode()` //native方法，用于返回对象的哈希码，主要使用在哈希表中，比如JDK中的HashMap。

`public boolean equals(Object obj)`//用于比较2个对象的内存地址是否相等，`String`类对该方法进行了重写用户比较字符串的值是否相等。

`protected native Object clone()` throws `CloneNotSupportedException`//native方法，用于创建并返回当前对象的一份拷贝。一般情况下，对于任何对象 `x`，表达式 `x.clone() != x` 为true，`x.clone().getClass() == x.getClass()` 为true。`Object`本身没有实现`Cloneable`接口，所以不重写`clone`方法并且进行调用的话会发生`CloneNotSupportedException`异常。

`public String toString()`//返回类的名字@实例的哈希码的16进制的字符串。建议`Object`所有的子类都重写这个方法。

`public final native void notify()`//native方法，并且不能重写。唤醒一个在此对象监视器上等待的线程(监视器相当于就是锁的概念)。如果有多个线程在等待只会任意唤醒一个。

`public final native void notifyAll()`//native方法，并且不能重写。跟`notify`一样，唯一的区别就是会唤醒在此对象监视器上等待的所有线程，而不是一个线程。

`public final native void wait(long timeout)` throws `InterruptedException`//native方法，并且不能重写。暂停线程的执行。注意：`sleep`方法没有释放锁，而`wait`方法释放了锁。`timeout`是等待时间。

`public final void wait(long timeout, int nanos)` throws `InterruptedException`//多了`nanos`参数，这个参数表示额外时间（以毫微秒为单位，范围是 0-999999）。所以超时的时间还需要加上`nanos`毫秒。

`public final void wait()` throws `InterruptedException`//跟之前的2个`wait`方法一样，只不过该方法一直等待，没有超时时间这个概念

`protected void finalize()` throws `Throwable` {}//实例被垃圾回收器回收的时候触发的操作

2.5.3. == 与 equals(重要)

`==` : 它的作用是判断两个对象的地址是不是相等。即，判断两个对象是不是同一个对象(基本数据类型`==`比较的是值，引用数据类型`==`比较的是内存地址)。

`equals()` : 它的作用也是判断两个对象是否相等。但它一般有两种使用情况：

- 情况 1：类没有覆盖 `equals()` 方法。则通过 `equals()` 比较该类的两个对象时，等价于通过“`==`”比较这两个对象。
- 情况 2：类覆盖了 `equals()` 方法。一般，我们都覆盖 `equals()` 方法来比较两个对象的内容是否相等；若它们的内容相等，则返回 `true` (即，认为这两个对象相等)。

举个例子：

```
public class test1 {  
    public static void main(String[] args) {  
        String a = new String("ab"); // a 为一个引用  
        String b = new String("ab"); // b 为另一个引用，对象的内容一样  
        String aa = "ab"; // 放在常量池中  
        String bb = "ab"; // 从常量池中查找  
        if (aa == bb) // true  
            System.out.println("aa==bb");  
        if (a == b) // false, 非同一对象  
            System.out.println("a==b");  
        if (a.equals(b)) // true  
            System.out.println("aEQb");  
        if (42 == 42.0) { // true  
            System.out.println("true");  
        }  
    }  
}
```

说明：

- `String` 中的 `equals` 方法是被重写过的，因为 `Object` 的 `equals` 方法是比较的对象的内存地址，而 `String` 的 `equals` 方法比较的是对象的值。
- 当创建 `String` 类型的对象时，虚拟机会在常量池中查找有没有已经存在的值和要创建的值相同的对象，如果有就把它赋给当前引用。如果没有就在常量池中重新创建一个 `String` 对象。

2.5.4. `hashCode` 与 `equals` (重要)

面试官可能会问你：“你重写过 `hashcode` 和 `equals` 么，为什么重写 `equals` 时必须重写 `hashCode` 方法？”

2.5.4.1. `hashCode()` 介绍

`hashCode()` 的作用是获取哈希码，也称为散列码；它实际上是返回一个 `int` 整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。`hashCode()` 定义在 JDK 的 `Object.java` 中，这就意味着 Java 中的任何类都包含有 `hashCode()` 函数。

散列表存储的是键值对(key-value)，它的特点是：能根据“键”快速的检索出对应的“值”。这其中就利用到了散列码！（可以快速找到所需要的对象）

2.5.4.2. 为什么要有 `hashCode`

我们先以“`HashSet` 如何检查重复”为例子来说明为什么要有 `hashCode`：当你把对象加入 `HashSet` 时，`HashSet` 会先计算对象的 `hashcode` 值来判断对象加入的位置，同时也会与该位置其他已经加入的对象的 `hashcode` 值作比较，如果没有相符的 `hashcode`，`HashSet` 会假设对象没有重复出现。但是如果发现有相同 `hashcode` 值的对象，这时会调用 `equals()` 方法来检查 `hashcode` 相等的对象是否真的相同。如果两者相同，`HashSet` 就不会让其加入操作成功。如果不同的话，就会重新散列到其他位置。

(摘自我的 Java 启蒙书《Head first java》第二版)。这样我们就大大减少了 equals 的次数，相应就大大提高了执行速度。

通过我们可以看出：`hashCode()` 的作用就是**获取哈希码**，也称为散列码；它实际上是返回一个 int 整数。这个**哈希码的作用是确定该对象在哈希表中的索引位置**。**hashCode() 在散列表中才有用，在其它情况下没用**。在散列表中 `hashCode()` 的作用是获取对象的散列码，进而确定该对象在散列表中的位置。

2.5.4.3. `hashCode ()` 与 `equals ()` 的相关规定

1. 如果两个对象相等，则 `hashcode` 一定也是相同的
2. 两个对象相等，对两个对象分别调用 `equals` 方法都返回 true
3. 两个对象有相同的 `hashcode` 值，它们也不一定是相等的
4. **因此，`equals` 方法被覆盖过，则 `hashCode` 方法也必须被覆盖**
5. `hashCode()` 的默认行为是对堆上的对象产生独特值。如果没有重写 `hashCode()`，则该 class 的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）

2.5.5. Java 序列化中如果有些字段不想进行序列化，怎么办？

对于不想进行序列化的变量，使用 `transient` 关键字修饰。

`transient` 关键字的作用是：阻止实例中那些用此关键字修饰的的变量序列化；当对象被反序列化时，被 `transient` 修饰的变量值不会被持久化和恢复。`transient` 只能修饰变量，不能修饰类和方法。

2.5.6. 获取用键盘输入常用的两种方法

方法 1：通过 `Scanner`

```
Scanner input = new Scanner(System.in);
String s = input.nextLine();
input.close();
```

方法 2：通过 `BufferedReader`

```
BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
String s = input.readLine();
```

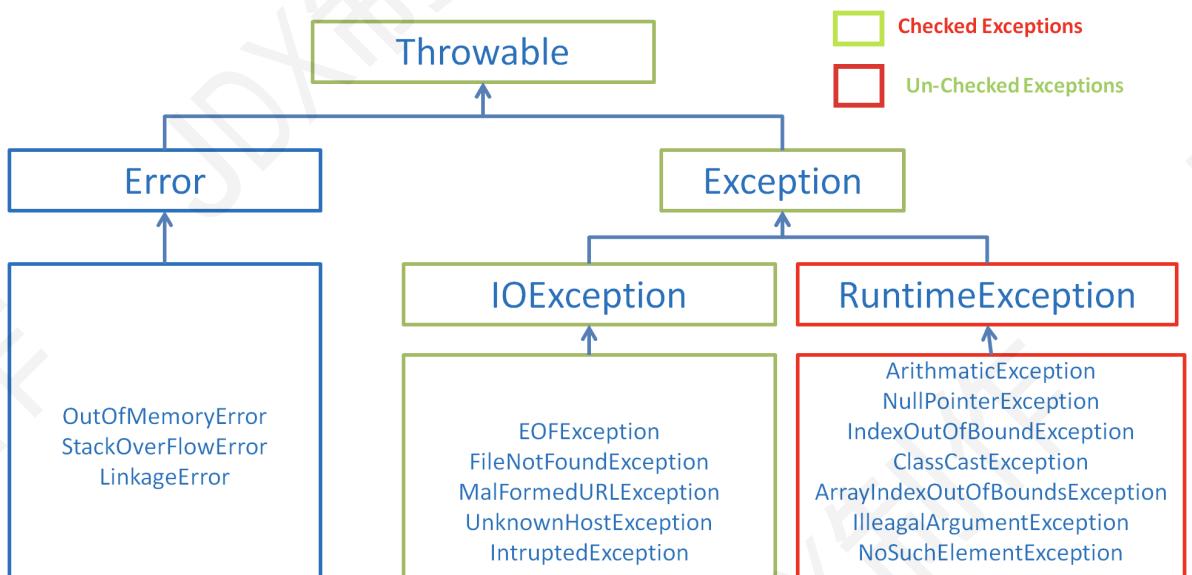
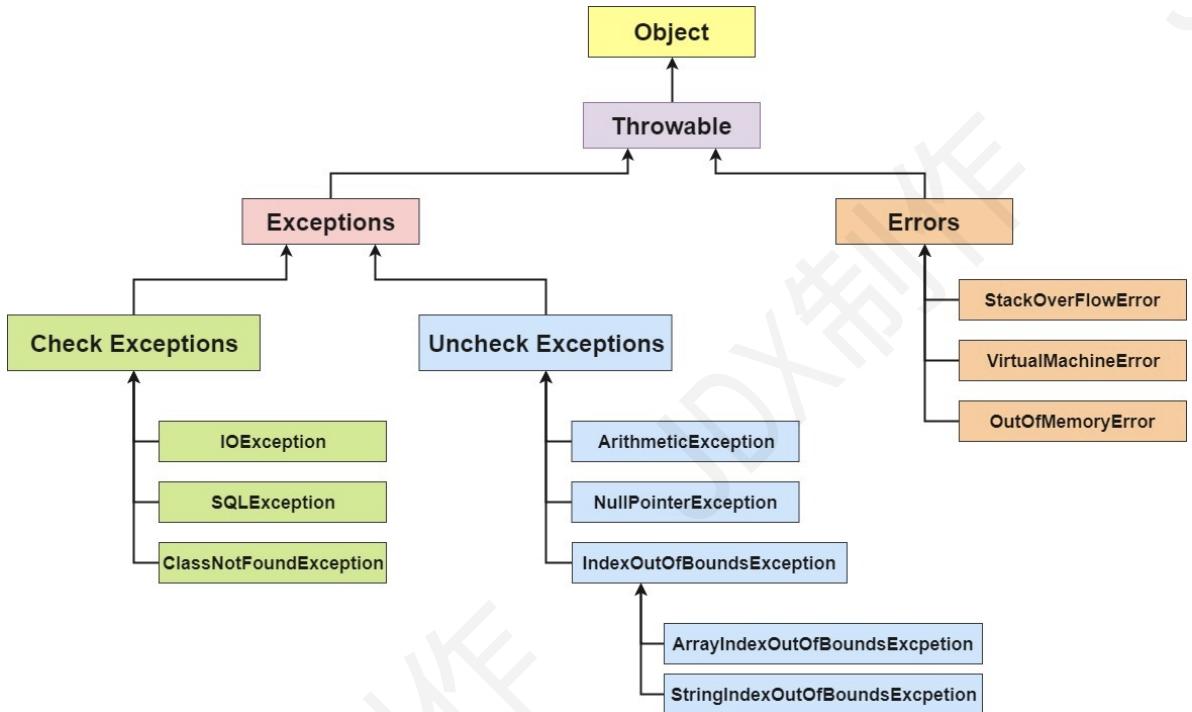
3. Java 核心技术

3.1. 集合

3.1.1. Collections 工具类和 Arrays 工具类常见方法总结

3.2. 异常

3.2.1. Java 异常类层次结构图



在 Java 中，所有的异常都有一个共同的祖先 `java.lang` 包中的 **Throwable** 类。 **Throwable**: 有两个重要的子类：**Exception**（异常）和 **Error**（错误），二者都是 Java 异常处理的重要子类，各自都包含大量子类。

Error（错误）：是程序无法处理的错误，表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。例如，Java 虚拟机运行错误（Virtual MachineError），当 JVM 不再有继续执行操作所需的内存资源时，将出现 **OutOfMemoryError**。这些异常发生时，Java 虚拟机（JVM）一般会选择线程终止。

这些错误表示故障发生于虚拟机自身、或者发生在虚拟机试图执行应用时，如 Java 虚拟机运行错误（Virtual MachineError）、类定义错误（`NoClassDefFoundError`）等。这些错误是不可查的，因为它们在应用程序的控制和处理能力之外，而且绝大多数是程序运行时不允许出现的状况。对于设计合理的应用程序来说，即使确实发生了错误，本质上也不应该试图去处理它所引起的异常状况。在 Java 中，错误通过 **Error** 的子类描述。

Exception (异常) :是程序本身可以处理的异常。Exception 类有一个重要的子类 **RuntimeException**。RuntimeException 异常由 Java 虚拟机抛出。**NullPointerException** (要访问的变量没有引用任何对象时, 抛出该异常)、**ArithmaticException** (算术运算异常, 一个整数除以 0 时, 抛出该异常) 和 **ArrayIndexOutOfBoundsException** (下标越界异常)。

注意：异常和错误的区别：异常能被程序本身处理，错误是无法处理。

3.2.2. Throwable 类常用方法

- `public string getMessage()` :返回异常发生时的简要描述
- `public string toString()` :返回异常发生时的详细信息
- `public string getLocalizedMessage()` :返回异常对象的本地化信息。使用 `Throwable` 的子类覆盖这个方法, 可以生成本地化信息。如果子类没有覆盖该方法, 则该方法返回的信息与 `getMessage()` 返回的结果相同
- `public void printStackTrace()` :在控制台上打印 `Throwable` 对象封装的异常信息

3.2.3. try-catch-finally

- **try 块**: 用于捕获异常。其后可接零个或多个 catch 块, 如果没有 catch 块, 则必须跟一个 finally 块。
- **catch 块**: 用于处理 try 捕获到的异常。
- **finally 块**: 无论是否捕获或处理异常, finally 块里的语句都会被执行。当在 try 块或 catch 块中遇到 return 语句时, finally 语句块将在方法返回之前被执行。

在以下 4 种特殊情况下, finally 块不会被执行:

1. 在 finally 语句块第一行发生了异常。因为在其他行, finally 块还是会得到执行
2. 在前面的代码中用了 `System.exit(int)` 已退出程序。exit 是带参函数 ; 若该语句在异常语句之后, finally 会执行
3. 程序所在的线程死亡。
4. 关闭 CPU。

注意：当 try 语句和 finally 语句中都有 return 语句时, 在方法返回之前, finally 语句的内容将被执行, 并且 finally 语句的返回值将会覆盖原始的返回值。如下:

```
public class Test {  
    public static int f(int value) {  
        try {  
            return value * value;  
        } finally {  
            if (value == 2) {  
                return 0;  
            }  
        }  
    }  
}
```

如果调用 `f(2)`, 返回值将是 0, 因为 finally 语句的返回值覆盖了 try 语句块的返回值。

3.2.4. 使用 try-with-resources 来代替 try-catch-finally

《Effecitve Java》中明确指出:

面对必须要关闭的资源, 我们总是应该优先使用try-with-resources而不是try-finally。随之产生的代码更简短, 更清晰, 产生的异常对我们也更有用。try-with-resources 语句让我们更容易编写必须要关闭的资源的代码, 若采用try-finally则几乎做不到这点。

Java 中类似于 `InputStream`、`OutputStream`、`Scanner`、`Printwriter` 等的资源都需要我们调用 `close()` 方法来手动关闭，一般情况下我们都是通过 `try-catch-finally` 语句来实现这个需求，如下：

```
//读取文本文件的内容
Scanner scanner = null;
try {
    scanner = new Scanner(new File("D://read.txt"));
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
} finally {
    if (scanner != null) {
        scanner.close();
    }
}
```

使用 Java 7 之后的 `try-with-resources` 语句改造上面的代码：

```
try (Scanner scanner = new Scanner(new File("test.txt"))) {
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (FileNotFoundException fnfe) {
    fnfe.printStackTrace();
}
```

当然多个资源需要关闭的时候，使用 `try-with-resources` 实现起来也非常简单，如果你还是用 `try-catch-finally` 可能会带来很多问题。

通过使用分号分隔，可以在 `try-with-resources` 块中声明多个资源：

3.3. 多线程

3.3.1. 简述线程、程序、进程的基本概念。以及他们之间关系是什么？

线程与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享同一块内存空间和一组系统资源，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。

程序是含有指令和数据的文件，被存储在磁盘或其他的数据存储设备中，也就是说程序是静态的代码。

进程是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。简单来说，一个进程就是一个执行中的程序，它在计算机中一个指令接着一个指令地执行着，同时，每个进程还占有某些系统资源如 CPU 时间，内存空间，文件，输入输出设备的使用权等等。换句话说，当程序在执行时，将被操作系统载入内存中。

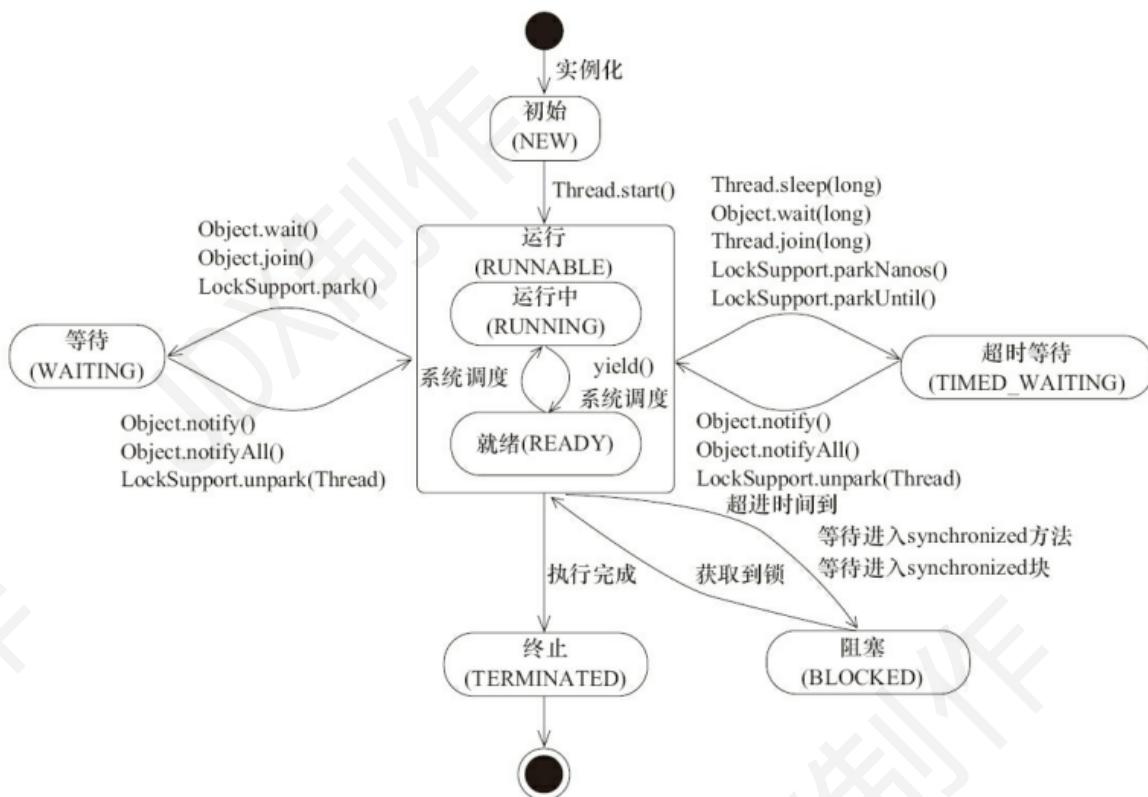
线程是进程划分成的更小的运行单位。线程和进程最大的不同在于基本上各进程是独立的，而各线程则不一定，因为同一进程中的线程极有可能会相互影响。从另一角度来说，进程属于操作系统的范畴，主要是同一段时间内，可以同时执行一个以上的程序，而线程则是在同一程序内几乎同时执行一个以上的程序段。

3.3.2. 线程有哪些基本状态？

Java 线程在运行的生命周期中的指定时刻只可能处于下面 6 种不同状态的其中一个状态（图源《Java 并发编程艺术》4.1.4 节）。

状态名称	说 明
NEW	初始状态，线程被构建，但是还没有调用 start() 方法
RUNNABLE	运行状态，Java 线程将操作系统中的就绪和运行两种状态笼统地称作“运行中”
BLOCKED	阻塞状态，表示线程阻塞于锁
WAITING	等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）
TIME_WAITING	超时等待状态，该状态不同于 WAITING，它是可以在指定的时间自行返回的
TERMINATED	终止状态，表示当前线程已经执行完毕

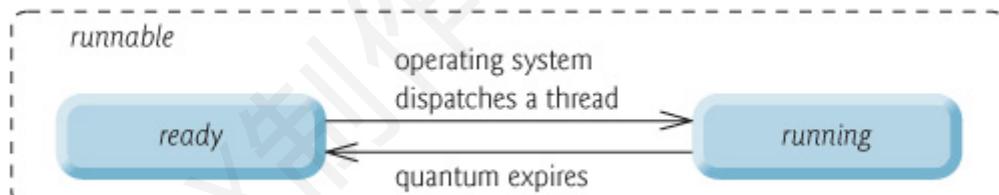
线程在生命周期中并不是固定处于某一个状态而是随着代码的执行在不同状态之间切换。Java 线程状态变迁如下图所示（图源《Java 并发编程艺术》4.1.4 节）：



由上图可以看出：

线程创建之后它将处于 **NEW (新建)** 状态，调用 `start()` 方法后开始运行，线程这时候处于 **READY (可运行)** 状态。可运行状态的线程获得了 cpu 时间片 (timeslice) 后就处于 **RUNNING (运行)** 状态。

操作系统隐藏 Java 虚拟机 (JVM) 中的 READY 和 RUNNING 状态，它只能看到 RUNNABLE 状态，所以 Java 系统一般将这两个状态统称为 **RUNNABLE (运行中)** 状态。



当线程执行 `wait()` 方法之后，线程进入 **WAITING (等待)** 状态。进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态，而 **TIME_WAITING(超时等待)** 状态相当于在等待状态的基础上增加了超时限制，比如通过 `sleep (long millis)` 方法或 `wait (long millis)` 方法可以将 Java 线程置于 TIMED_WAITING 状态。当超时时间到达后 Java 线程将会返回到 RUNNABLE 状态。当线程调用同

步方法时，在没有获取到锁的情况下，线程将会进入到 **BLOCKED (阻塞)** 状态。线程在执行 Runnable 的 run() 方法之后将会进入到 **TERMINATED (终止)** 状态。

3.4. 文件与 I/O 流

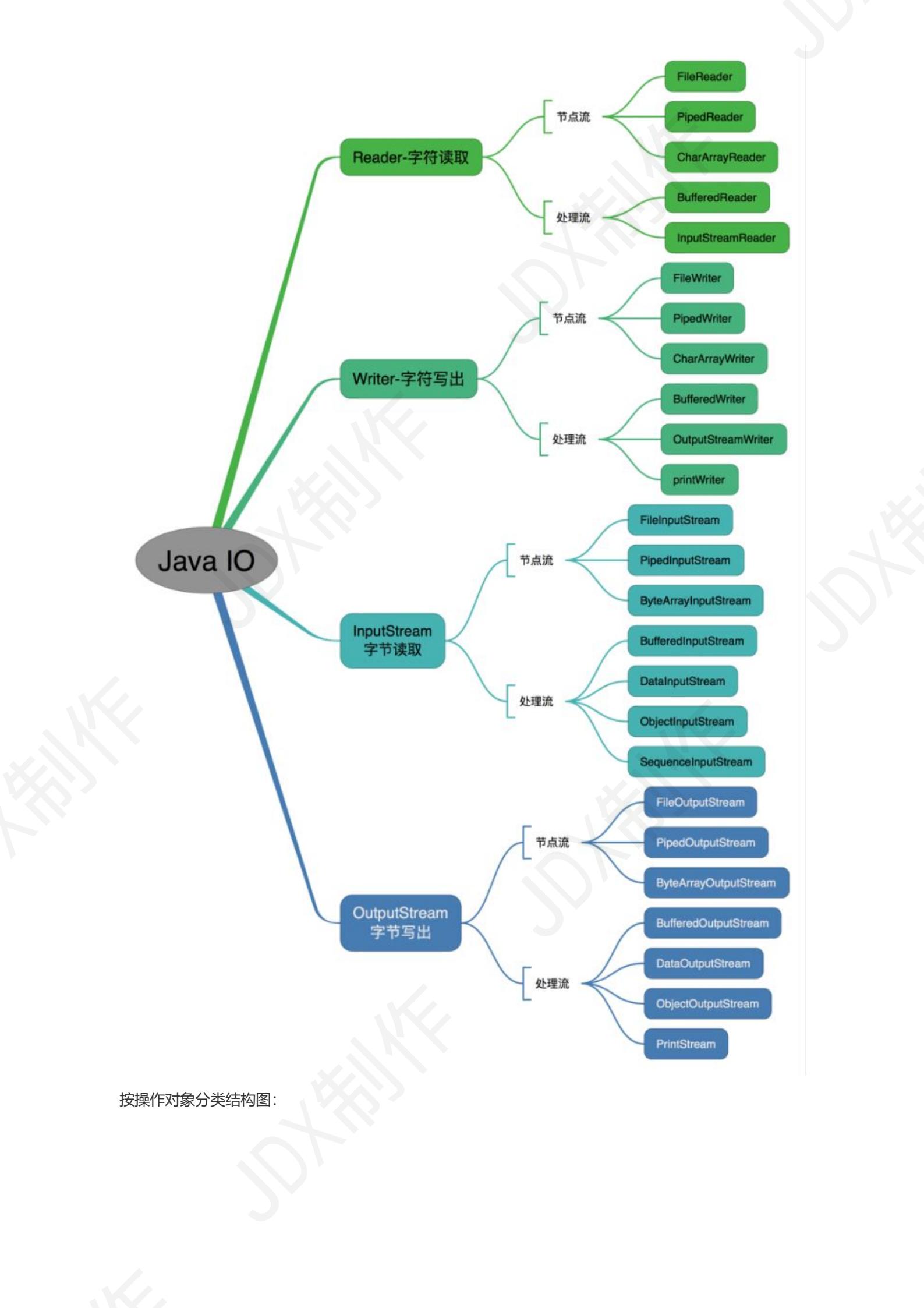
3.4.1. Java 中 IO 流分为几种？

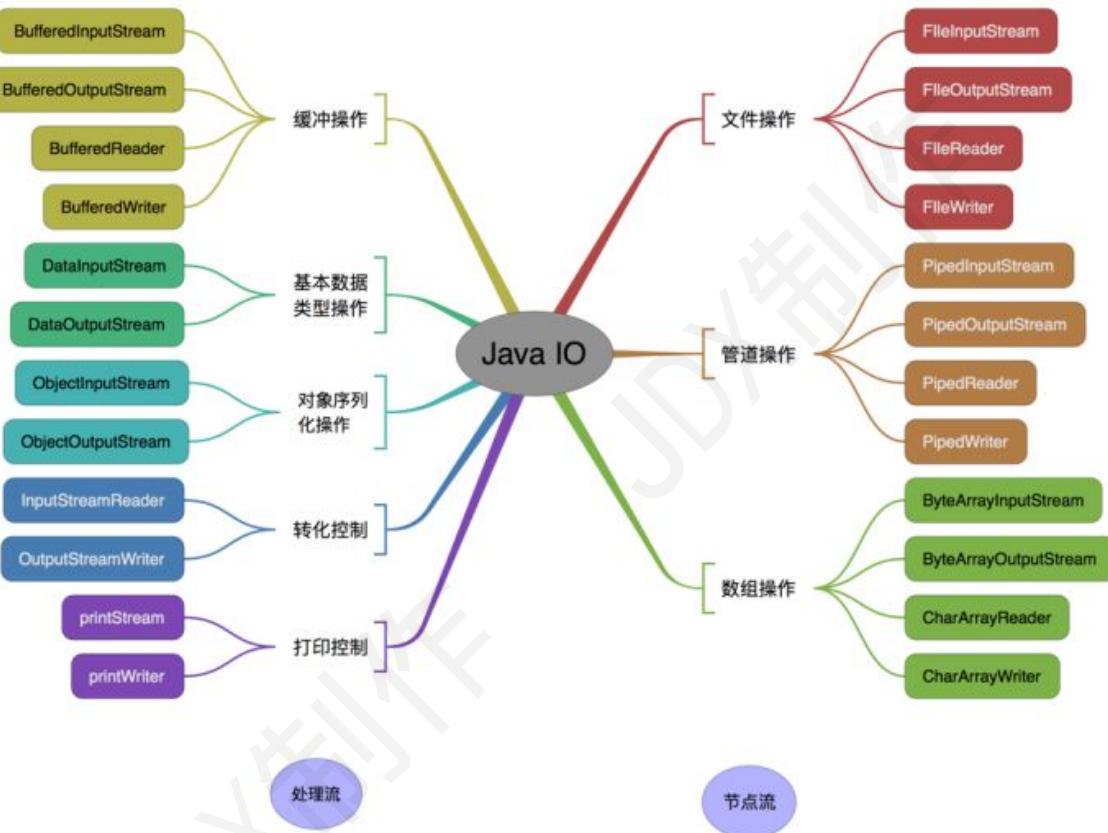
- 按照流的流向分，可以分为输入流和输出流；
- 按照操作单元划分，可以划分为字节流和字符流；
- 按照流的角色划分为节点流和处理流。

Java I/O 流共涉及 40 多个类，这些类看上去很杂乱，但实际上很有规则，而且彼此之间存在非常紧密的联系，Java I/O 流的 40 多个类都是从如下 4 个抽象类基类中派生出来的。

- InputStream/Reader: 所有的输入流的基类，前者是字节输入流，后者是字符输入流。
- OutputStream/Writer: 所有输出流的基类，前者是字节输出流，后者是字符输出流。

按操作方式分类结构图：





3.4.1.1. 既然有了字节流,为什么还要有字符流?

问题本质想问：不管是文件读写还是网络发送接收，信息的最小存储单元都是字节，那为什么 I/O 流操作要分为字节流操作和字符流操作呢？

回答：字符流是由 Java 虚拟机将字节转换得到的，问题就出在这个过程还算是非常耗时，并且，如果我们不知道编码类型就很容易出现乱码问题。所以，I/O 流就干脆提供了一个直接操作字符的接口，方便我们平时对字符进行流操作。如果音频文件、图片等媒体文件用字节流比较好，如果涉及到字符的话使用字符流比较好。

3.4.1.2. BIO,NIO,AIO 有什么区别?

- **BIO (Blocking I/O):** 同步阻塞 I/O 模式，数据的读取写入必须阻塞在一个线程内等待其完成。在活动连接数不是特别高（小于单机 1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。
- **NIO (Non-blocking/New I/O):** NIO 是一种同步非阻塞的 I/O 模型，在 Java 1.4 中引入了 NIO 框架，对应 java.nio 包，提供了 Channel, Selector, Buffer 等抽象。NIO 中的 N 可以理解为 Non-blocking，不单纯是 New。它支持面向缓冲的，基于通道的 I/O 操作方法。NIO 提供了与传统 BIO 模型中的 socket 和 ServerSocket 相对应的 SocketChannel 和 ServerSocketChannel 两种不同的套接字通道实现，两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。对于低负载、低并发的应用程序，可以使用同步阻塞 I/O 来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发
- **AIO (Asynchronous I/O):** AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2,它是异步非阻塞的 IO 模型。异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。AIO 是异步 IO 的缩写，虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线程自行进行 IO 操

作，IO 操作本身是同步的。查阅网上相关资料，我发现就目前来说 AIO 的应用还不是很广泛，Netty 之前也尝试使用过 AIO，不过又放弃了。

(二). 容器

1. ArrayList

1.1 ArrayList简介

ArrayList 的底层是数组队列，相当于动态数组。与 Java 中的数组相比，它的容量能动态增长。在添加大量元素前，应用程序可以使用 ensureCapacity 操作来增加 ArrayList 实例的容量。这可以减少递增式再分配的数量。

它继承于 **AbstractList**，实现了 **List, RandomAccess, Cloneable, java.io.Serializable** 这些接口。

在我们学数据结构的时候就知道了线性表的顺序存储，插入删除元素的时间复杂度为 **O (n)**，求表长以及增加元素，取第 *i* 元素的时间复杂度为 **O (1)**

ArrayList 继承了 AbstractList，实现了 List。它是一个数组队列，提供了相关的添加、删除、修改、遍历等功能。

ArrayList 实现了 **RandomAccess 接口**， RandomAccess 是一个标志接口，表明实现这个这个接口的 List 集合是支持快速随机访问的。在 ArrayList 中，我们即可以通过元素的序号快速获取元素对象，这就是快速随机访问。

ArrayList 实现了 **Cloneable 接口**，即覆盖了函数 clone()，能被克隆。

ArrayList 实现 **java.io.Serializable 接口**，这意味着 ArrayList 支持序列化，能通过序列化去传输。

和 Vector 不同， **ArrayList 中的操作不是线程安全的！** 所以，建议在单线程中才使用 ArrayList，而在多线程中可以选择 Vector 或者 CopyOnWriteArrayList。

1.2 ArrayList核心源码

```
package java.util;

import java.util.function.Consumer;
import java.util.function.Predicate;
import java.util.function.UnaryOperator;

public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    private static final long serialVersionUID = 8683452581122892189L;

    /**
     * 默认初始容量大小
     */
    private static final int DEFAULT_CAPACITY = 10;

    /**
     * 空数组（用于空实例）。
     */
    private static final Object[] EMPTY_ELEMENTDATA = {};

    //用于默认大小空实例的共享空数组实例。
```

```
//我们把它从EMPTY_ELEMENTDATA数组中区分出来，以知道在添加第一个元素时容量需要增加多少。
private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

/**
 * 保存ArrayList数据的数组
 */
transient Object[] elementData; // non-private to simplify nested class
access

/**
 * ArrayList 所包含的元素个数
 */
private int size;

/**
 * 带初始容量参数的构造函数。（用户自己指定容量）
 */
public ArrayList(int initialCapacity) {
    if (initialCapacity > 0) {
        //创建initialCapacity大小的数组
        this.elementData = new Object[initialCapacity];
    } else if (initialCapacity == 0) {
        //创建空数组
        this.elementData = EMPTY_ELEMENTDATA;
    } else {
        throw new IllegalArgumentException("Illegal Capacity: "+
                                           initialCapacity);
    }
}

/**
 *默认构造函数，DEFAULTCAPACITY_EMPTY_ELEMENTDATA 为0.初始化为10，也就是说初始其实是空数组 当添加第一个元素的时候数组容量才变成10
 */
public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}

/**
 * 构造一个包含指定集合的元素的列表，按照它们由集合的迭代器返回的顺序。
 */
public ArrayList(Collection<? extends E> c) {
    //
    elementData = c.toArray();
    //如果指定集合元素个数不为0
    if ((size = elementData.length) != 0) {
        // c.toArray 可能返回的不是Object类型的数组所以加上下面的语句用于判断,
        //这里用到了反射里面的getClass()方法
        if (elementData.getClass() != Object[].class)
            elementData = Arrays.copyOf(elementData, size, Object[].class);
    } else {
        // 用空数组代替
        this.elementData = EMPTY_ELEMENTDATA;
    }
}

/**
```

```
* 修改这个ArrayList实例的容量是列表的当前大小。 应用程序可以使用此操作来最小化
ArrayList实例的存储。
*/
public void trimToSize() {
    modCount++;
    if (size < elementData.length) {
        elementData = (size == 0)
            ? EMPTY_ELEMENTDATA
            : Arrays.copyOf(elementData, size);
    }
}

//下面是ArrayList的扩容机制
//ArrayList的扩容机制提高了性能，如果每次只扩充一个，
//那么频繁的插入会导致频繁的拷贝，降低性能，而ArrayList的扩容机制避免了这种情况。
/**
 * 如有必要，增加此ArrayList实例的容量，以确保它至少能容纳元素的数量
 * @param minCapacity 所需的最小容量
 */
public void ensureCapacity(int minCapacity) {
    int minExpand = (elementData != DEFAULTCAPACITY_EMPTY_ELEMENTDATA)
        // any size if not default element table
        ? 0
        // larger than default for default empty table. It's already
        // supposed to be at default size.
        : DEFAULT_CAPACITY;

    if (minCapacity > minExpand) {
        ensureExplicitCapacity(minCapacity);
    }
}

//得到最小扩容量
private void ensureCapacityInternal(int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        // 获取默认的容量和传入参数的较大值
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
    }

    ensureExplicitCapacity(minCapacity);
}

//判断是否需要扩容
private void ensureExplicitCapacity(int minCapacity) {
    modCount++;

    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        //调用grow方法进行扩容，调用此方法代表已经开始扩容了
        grow(minCapacity);
}

/**
 * 要分配的最大数组大小
 */
private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;

/**
 * ArrayList扩容的核心方法。
 */
private void grow(int minCapacity) {
```

```
// oldCapacity为旧容量, newCapacity为新容量
int oldCapacity = elementData.length;
//将oldCapacity 右移一位, 其效果相当于oldCapacity /2,
//我们知道位运算的速度远远快于整除运算, 整句运算式的结果就是将新容量更新为旧容量的1.5
倍,
int newCapacity = oldCapacity + (oldCapacity >> 1);
//然后检查新容量是否大于最小需要容量, 若还是小于最小需要容量, 那么就把最小需要容量当作
数组的新容量,
if (newCapacity - minCapacity < 0)
    newCapacity = minCapacity;
//再检查新容量是否超出了ArrayList所定义的最大容量,
//若超出了, 则调用hugeCapacity()来比较minCapacity和 MAX_ARRAY_SIZE,
//如果minCapacity大于MAX_ARRAY_SIZE, 则新容量则为Integer.MAX_VALUE, 否则, 新
容量大小则为 MAX_ARRAY_SIZE.
if (newCapacity - MAX_ARRAY_SIZE > 0)
    newCapacity = hugeCapacity(minCapacity);
// minCapacity is usually close to size, so this is a win:
elementData = Arrays.copyOf(elementData, newCapacity);
}

//比较minCapacity和 MAX_ARRAY_SIZE
private static int hugeCapacity(int minCapacity) {
    if (minCapacity < 0) // overflow
        throw new OutOfMemoryError();
    return (minCapacity > MAX_ARRAY_SIZE) ?
        Integer.MAX_VALUE :
        MAX_ARRAY_SIZE;
}

/**
 * 返回此列表中的元素数。
 */
public int size() {
    return size;
}

/**
 * 如果此列表不包含元素, 则返回 true 。
 */
public boolean isEmpty() {
    //注意=和==的区别
    return size == 0;
}

/**
 * 如果此列表包含指定的元素, 则返回true 。
 */
public boolean contains(Object o) {
    //indexOf()方法: 返回此列表中指定元素的首次出现的索引, 如果此列表不包含此元素, 则
    //为-1
    return indexOf(o) >= 0;
}

/**
 * 返回此列表中指定元素的首次出现的索引, 如果此列表不包含此元素, 则为-1
 */
public int indexOf(Object o) {
    if (o == null) {
        for (int i = 0; i < size; i++)
```

```
        if (elementData[i]==null)
            return i;
    } else {
        for (int i = 0; i < size; i++)
            //equals()方法比较
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}

/**
 * 返回此列表中指定元素的最后一次出现的索引，如果此列表不包含元素，则返回-1。.
 */
public int lastIndexOf(Object o) {
    if (o == null) {
        for (int i = size-1; i >= 0; i--)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = size-1; i >= 0; i--)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}

/**
 * 返回此ArrayList实例的浅拷贝。（元素本身不被复制。）
 */
public Object clone() {
    try {
        ArrayList<?> v = (ArrayList<?>) super.clone();
        //Arrays.copyOf功能是实现数组的复制，返回复制后的数组。参数是被复制的数组和复制
        的长度
        v.elementData = Arrays.copyOf(elementData, size);
        v.modCount = 0;
        return v;
    } catch (CloneNotSupportedException e) {
        // 这不应该发生，因为我们是可以克隆的
        throw new InternalError(e);
    }
}

/**
 *以正确的顺序（从第一个到最后一个元素）返回一个包含此列表中所有元素的数组。
 *返回的数组将是“安全的”，因为该列表不保留对它的引用。（换句话说，这个方法必须分配一个新的
数组）。
 *因此，调用者可以自由地修改返回的数组。此方法充当基于阵列和基于集合的API之间的桥梁。
 */
public Object[] toArray() {
    return Arrays.copyOf(elementData, size);
}

/**
 * 以正确的顺序返回一个包含此列表中所有元素的数组（从第一个到最后一个元素）；
 *返回的数组的运行时类型是指定数组的运行时类型。如果列表适合指定的数组，则返回其中。
 *否则，将为指定数组的运行时类型和此列表的大小分配一个新数组。

```

*如果列表适用于指定的数组，其余空间（即数组的列表数量多于此元素），则紧跟在集合结束后的数组中的元素设置为null。

```
* (这仅在调用者知道列表不包含任何空元素的情况下才能确定列表的长度。)
*/
@SuppressWarnings("unchecked")
public <T> T[] toArray(T[] a) {
    if (a.length < size)
        // 新建一个运行时类型的数组，但是ArrayList数组的内容
        return (T[]) Arrays.copyOf(elementData, size, a.getClass());
    // 调用System提供的arraycopy()方法实现数组之间的复制
    System.arraycopy(elementData, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}

// Positional Access Operations

@SuppressWarnings("unchecked")
E elementData(int index) {
    return (E) elementData[index];
}

/**
 * 返回此列表中指定位置的元素。
 */
public E get(int index) {
    rangeCheck(index);

    return elementData(index);
}

/**
 * 用指定的元素替换此列表中指定位置的元素。
 */
public E set(int index, E element) {
    // 对index进行界限检查
    rangeCheck(index);

    E oldValue = elementData(index);
    elementData[index] = element;
    // 返回原来在这个位置的元素
    return oldValue;
}

/**
 * 将指定的元素追加到此列表的末尾。
 */
public boolean add(E e) {
    ensureCapacityInternal(size + 1); // Increments modCount!!
    // 这里看到ArrayList添加元素的实质就相当于为数组赋值
    elementData[size++] = e;
    return true;
}

/**
 * 在此列表中的指定位置插入指定的元素。
*/
```

```
*先调用 rangeCheckForAdd 对index进行界限检查; 然后调用 ensureCapacityInternal 方法保证capacity足够大;
*再将从index开始之后的所有成员后移一个位置; 将element插入index位置; 最后size加1。
*/
public void add(int index, E element) {
    rangeCheckForAdd(index);

    ensureCapacityInternal(size + 1); // Increments modCount!!
    //arraycopy()这个实现数组之间复制的方法一定要看一下，下面就用到了arraycopy()方法实现数组自己复制自己
    System.arraycopy(elementData, index, elementData, index + 1,
                     size - index);
    elementData[index] = element;
    size++;
}

/**
 * 删除该列表中指定位置的元素。 将任何后续元素移动到左侧（从其索引中减去一个元素）。
 */
public E remove(int index) {
    rangeCheck(index);

    modCount++;
    E oldValue = elementData(index);

    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
                         numMoved);
    elementData[--size] = null; // clear to let GC do its work
    //从列表中删除的元素
    return oldValue;
}

/**
 * 从列表中删除指定元素的第一个出现（如果存在）。 如果列表不包含该元素，则它不会更改。
 *返回true，如果此列表包含指定的元素
 */
public boolean remove(Object o) {
    if (o == null) {
        for (int index = 0; index < size; index++)
            if (elementData[index] == null)
                fastRemove(index);
                return true;
    }
    else {
        for (int index = 0; index < size; index++)
            if (o.equals(elementData[index]))
                fastRemove(index);
                return true;
    }
    return false;
}

/*
 * Private remove method that skips bounds checking and does not
 * return the value removed.
*/
```

```
 */
private void fastRemove(int index) {
    modCount++;
    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
                         numMoved);
    elementData[--size] = null; // clear to let GC do its work
}

/**
 * 从列表中删除所有元素。
 */
public void clear() {
    modCount++;

    // 把数组中所有的元素的值设为null
    for (int i = 0; i < size; i++)
        elementData[i] = null;

    size = 0;
}

/**
 * 按指定集合的Iterator返回的顺序将指定集合中的所有元素追加到此列表的末尾。
 */
public boolean addAll(Collection<? extends E> c) {
    Object[] a = c.toArray();
    int numNew = a.length;
    ensureCapacityInternal(size + numNew); // Increments modCount
    System.arraycopy(a, 0, elementData, size, numNew);
    size += numNew;
    return numNew != 0;
}

/**
 * 将指定集合中的所有元素插入到此列表中，从指定的位置开始。
 */
public boolean addAll(int index, Collection<? extends E> c) {
    rangeCheckForAdd(index);

    Object[] a = c.toArray();
    int numNew = a.length;
    ensureCapacityInternal(size + numNew); // Increments modCount

    int numMoved = size - index;
    if (numMoved > 0)
        System.arraycopy(elementData, index, elementData, index + numNew,
                         numMoved);

    System.arraycopy(a, 0, elementData, index, numNew);
    size += numNew;
    return numNew != 0;
}

/**
 * 从此列表中删除所有索引为fromIndex（含）和toIndex之间的元素。
 * 将任何后续元素移动到左侧（减少其索引）。
 */
```

```
/*
protected void removeRange(int fromIndex, int toIndex) {
    modCount++;
    int numMoved = size - toIndex;
    System.arraycopy(elementData, toIndex, elementData, fromIndex,
                     numMoved);

    // clear to let GC do its work
    int newSize = size - (toIndex-fromIndex);
    for (int i = newSize; i < size; i++) {
        elementData[i] = null;
    }
    size = newSize;
}

/**
 * 检查给定的索引是否在范围内。
 */
private void rangeCheck(int index) {
    if (index >= size)
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}

/**
 * add和addAll使用的rangeCheck的一个版本
 */
private void rangeCheckForAdd(int index) {
    if (index > size || index < 0)
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}

/**
 * 返回IndexOutOfBoundsException细节信息
 */
private String outOfBoundsMsg(int index) {
    return "Index: "+index+", Size: "+size;
}

/**
 * 从此列表中删除指定集合中包含的所有元素。
 */
public boolean removeAll(Collection<?> c) {
    Objects.requireNonNull(c);
    //如果此列表被修改则返回true
    return batchRemove(c, false);
}

/**
 * 仅保留此列表中包含在指定集合中的元素。
 *换句话说，从此列表中删除其中不包含在指定集合中的所有元素。
 */
public boolean retainAll(Collection<?> c) {
    Objects.requireNonNull(c);
    return batchRemove(c, true);
}

/**
```

```

    * 从列表中的指定位置开始，返回列表中的元素（按正确顺序）的列表迭代器。
    * 指定的索引表示初始调用将返回的第一个元素为next。 初始调用previous将返回指定索引减1的
      元素。
    * 返回的列表迭代器是fail-fast。
    */
    public ListIterator<E> listIterator(int index) {
        if (index < 0 || index > size)
            throw new IndexOutOfBoundsException("Index: "+index);
        return new ListItr(index);
    }

    /**
     * 返回列表中的列表迭代器（按适当的顺序）。
     * 返回的列表迭代器是fail-fast。
     */
    public ListIterator<E> listIterator() {
        return new ListItr(0);
    }

    /**
     * 以正确的顺序返回该列表中的元素的迭代器。
     * 返回的迭代器是fail-fast。
     */
    public Iterator<E> iterator() {
        return new Itr();
    }

```

1.3 ArrayList源码分析

1.3.1 System.arraycopy()和Arrays.copyOf()方法

通过上面源码我们发现这两个实现数组复制的方法被广泛使用而且很多地方都特别巧妙。比如下面add(int index, E element)方法就很巧妙的用到了arraycopy()方法让数组自己复制自己实现让index开始之后的所有成员后移一个位置:

```

    /**
     * 在此列表中的指定位置插入指定的元素。
     * 先调用 rangeCheckForAdd 对index进行界限检查；然后调用 ensureCapacityInternal 方法
     * 保证capacity足够大；
     * 再将从index开始之后的所有成员后移一个位置；将element插入index位置；最后size加1。
     */
    public void add(int index, E element) {
        rangeCheckForAdd(index);

        ensureCapacityInternal(size + 1); // Increments modCount!!
        //arraycopy()方法实现数组自己复制自己
        //elementData:源数组;index:源数组中的起始位置;elementData: 目标数组; index +
        1: 目标数组中的起始位置; size - index: 要复制的数组元素的数量;
        System.arraycopy(elementData, index, elementData, index + 1, size -
        index);
        elementData[index] = element;
        size++;
    }

```

又如toArray()方法中用到了copyOf()方法

```

/**
 * 以正确的顺序（从第一个到最后一个元素）返回一个包含此列表中所有元素的数组。
 * 返回的数组将是“安全的”，因为该列表不保留对它的引用。（换句话说，这个方法必须分配一个新的
 * 数组）。
 * 因此，调用者可以自由地修改返回的数组。此方法充当基于阵列和基于集合的API之间的桥梁。
 */
public Object[] toArray() {
    //elementData: 要复制的数组; size: 要复制的长度
    return Arrays.copyOf(elementData, size);
}

```

1.3.2 两者联系与区别

联系：

看两者源代码可以发现 `copyOf()` 内部调用了 `System.arraycopy()` 方法

区别：

1. `arraycopy()` 需要目标数组，将原数组拷贝到你自己定义的数组里，而且可以选择拷贝的起点和长度以及放入新数组中的位置
2. `copyOf()` 是系统自动在内部新建一个数组，并返回该数组。

1.3.3 ArrayList 核心扩容技术

```

//下面是ArrayList的扩容机制
//ArrayList的扩容机制提高了性能，如果每次只扩充一个，
//那么频繁的插入会导致频繁的拷贝，降低性能，而ArrayList的扩容机制避免了这种情况。
/**
 * 如有必要，增加此ArrayList实例的容量，以确保它至少能容纳元素的数量
 * @param minCapacity 所需的最小容量
 */
public void ensureCapacity(int minCapacity) {
    int minExpand = (elementData != DEFAULTCAPACITY_EMPTY_ELEMENTDATA)
        // any size if not default element table
        ? 0
        // larger than default for default empty table. It's already
        // supposed to be at default size.
        : DEFAULT_CAPACITY;

    if (minCapacity > minExpand) {
        ensureExplicitCapacity(minCapacity);
    }
}

//得到最小扩容量
private void ensureCapacityInternal(int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        // 获取默认的容量和传入参数的较大值
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
    }

    ensureExplicitCapacity(minCapacity);
}

//判断是否需要扩容，上面两个方法都要调用
private void ensureExplicitCapacity(int minCapacity) {
    modCount++;
}

```

```

    // 如果说minCapacity也就是所需的最小容量大于保存ArrayList数据的数组的长度的话，就需要调用grow(minCapacity)方法扩容。
    //这个minCapacity到底为多少呢？举个例子在添加元素(add)方法中这个minCapacity的大小就为现在数组的长度加1
    if (minCapacity - elementData.length > 0)
        //调用grow方法进行扩容，调用此方法代表已经开始扩容了
        grow(minCapacity);
}

```

```

/**
 * ArrayList扩容的核心方法。
 */
private void grow(int minCapacity) {
    //elementData为保存ArrayList数据的数组
    //elementData.length求数组长度elementData.size是求数组中的元素个数
    // oldCapacity为旧容量, newCapacity为新容量
    int oldCapacity = elementData.length;
    //将oldCapacity 右移一位，其效果相当于oldCapacity /2,
    //我们知道位运算的速度远远快于整除运算，整句运算式的结果就是将新容量更新为旧容量的1.5倍,
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    //然后检查新容量是否大于最小需要容量，若还是小于最小需要容量，那么就把最小需要容量当作数组的新容量,
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    //再检查新容量是否超出了ArrayList所定义的最大容量,
    //若超出了，则调用hugeCapacity()来比较minCapacity和 MAX_ARRAY_SIZE,
    //如果minCapacity大于MAX_ARRAY_SIZE，则新容量则为Integer.MAX_VALUE, 否则，新容量大小则为 MAX_ARRAY_SIZE。
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}

```

扩容机制代码已经做了详细的解释。另外值得注意的是大家很容易忽略的一个运算符：**移位运算符**

简介：移位运算符就是在二进制的基础上对数字进行平移。按照平移的方向和填充数字的规则分为三种：
<<(左移)、>>(带符号右移)和>>>(无符号右移)。

作用：对于大数据的2进制运算,位移运算符比那些普通运算符的运算要快很多,因为程序仅仅移动一下而已,不去计算,这样提高了效率,节省了资源

比如这里：int newCapacity = oldCapacity + (oldCapacity >> 1);

右移一位相当于除2, 右移n位相当于除以2的n次方。这里 oldCapacity 明显右移了1位所以相当于 oldCapacity /2。

另外需要注意的是：

1. java 中的**length 属性**是针对数组说的,比如说你声明了一个数组,想知道这个数组的长度则用到了 length 这个属性.
2. java 中的**length()方法**是针对字符串String说的,如果想看这个字符串的长度则用到 length()这个方法.
3. java 中的**size()方法**是针对泛型集合说的,如果想看这个泛型有多少个元素,就调用此方法来查看!

1.3.4 内部类

```
(1)private class Itr implements Iterator<E>
(2)private class ListItr extends Itr implements ListIterator<E>
(3)private class SubList extends AbstractList<E> implements RandomAccess
(4)static final class ArrayListSpliterator<E> implements Spliterator<E>
```

ArrayList有四个内部类，其中的Itr是实现了Iterator接口，同时重写了里面的hasNext(), next(), remove() 等方法；其中的ListItr继承 Itr，实现了ListIterator接口，同时重写了hasPrevious(), nextIndex(), previousIndex(), previous(), set(E e), add(E e) 等方法，所以这也看出了Iterator和ListIterator的区别：ListIterator在Iterator的基础上增加了添加对象，修改对象，逆向遍历等方法，这些是Iterator不能实现的。

1.4 ArrayList经典Demo

```
package list;
import java.util.ArrayList;
import java.util.Iterator;

public class ArrayListDemo {

    public static void main(String[] args){
        ArrayList<Integer> arrayList = new ArrayList<Integer>();

        System.out.printf("Before add:arrayList.size() =
%d\n",arrayList.size());

        arrayList.add(1);
        arrayList.add(3);
        arrayList.add(5);
        arrayList.add(7);
        arrayList.add(9);
        System.out.printf("After add:arrayList.size() =
%d\n",arrayList.size());

        System.out.println("Printing elements of arrayList");
        // 三种遍历方式打印元素
        // 第一种：通过迭代器遍历
        System.out.print("通过迭代器遍历:");
        Iterator<Integer> it = arrayList.iterator();
        while(it.hasNext()){
            System.out.print(it.next() + " ");
        }
        System.out.println();

        // 第二种：通过索引值遍历
        System.out.print("通过索引值遍历:");
        for(int i = 0; i < arrayList.size(); i++){
            System.out.print(arrayList.get(i) + " ");
        }
        System.out.println();

        // 第三种：for循环遍历
        System.out.print("for循环遍历:");
        for(Integer number : arrayList){
            System.out.print(number + " ");
        }
    }
}
```

```
// toArray用法
// 第一种方式(最常用)
Integer[] integer = arrayList.toArray(new Integer[0]);

// 第二种方式(容易理解)
Integer[] integer1 = new Integer[arrayList.size()];
arrayList.toArray(integer1);

// 抛出异常, java不支持向下转型
//Integer[] integer2 = new Integer[arrayList.size()];
//integer2 = arrayList.toArray();
System.out.println();

// 在指定位置添加元素
arrayList.add(2, 2);
// 删除指定位置上的元素
arrayList.remove(2);
// 删除指定元素
arrayList.remove((Object)3);
// 判断arrayList是否包含5
System.out.println("ArrayList contains 5 is: " +
arrayList.contains(5));

// 清空ArrayList
arrayList.clear();
// 判断ArrayList是否为空
System.out.println("ArrayList is empty: " + arrayList.isEmpty());
}

}
```

2. LinkedList

2.1 简介

LinkedList是一个实现了List接口和Deque接口的双端链表。

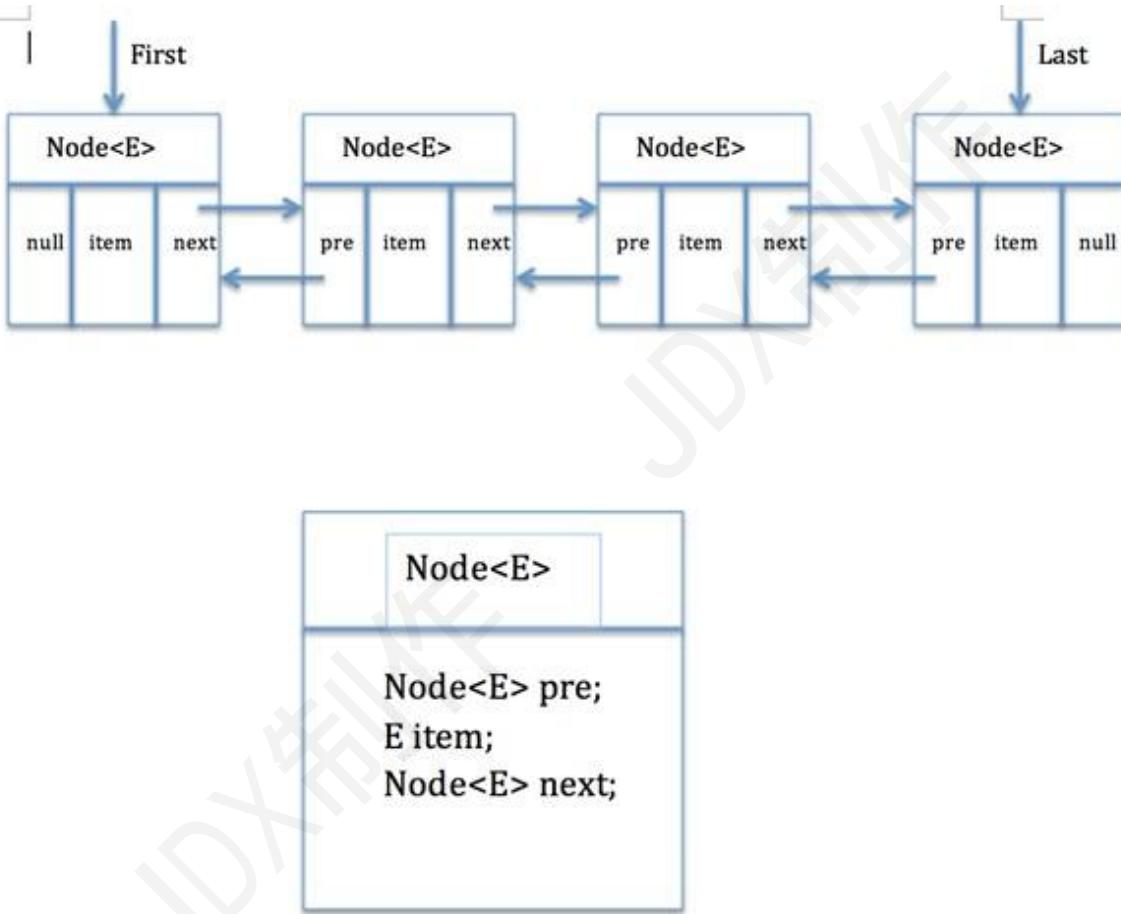
LinkedList底层的链表结构使它支持高效的插入和删除操作，另外它实现了Deque接口，使得LinkedList类也具有队列的特性；

LinkedList不是线程安全的，如果想使LinkedList变成线程安全的，可以调用静态类Collections类中的synchronizedList方法：

```
List list= Collections.synchronizedList(new LinkedList(...));
```

2.2 内部结构分析

如下图所示：



看完了图之后，我们再看LinkedList类中的一个**内部私有类Node**就很好理解了：

```
private static class Node<E> {
    E item;//节点值
    Node<E> next;//后继节点
    Node<E> prev;//前驱节点

    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

这个类就代表双端链表的节点Node。这个类有三个属性，分别是前驱节点，本节点的值，后继结点。

2.3 LinkedList源码分析

2.3.1 构造方法

空构造方法：

```
public LinkedList() { }
```

用已有的集合创建链表的构造方法：

```
public LinkedList(Collection<? extends E> c) {  
    this();  
    addAll(c);  
}
```

2.3.2 add方法

add(E e) 方法：将元素添加到链表尾部

```
public boolean add(E e) {  
    linkLast(e); //这里就只调用了这一个方法  
    return true;  
}
```

```
/**  
 * 链接使e作为最后一个元素。  
 */  
void linkLast(E e) {  
    final Node<E> l = last;  
    final Node<E> newNode = new Node<>(l, e, null);  
    last = newNode; //新建节点  
    if (l == null)  
        first = newNode;  
    else  
        l.next = newNode; //指向后继元素也就是指向下一个元素  
    size++;  
    modCount++;  
}
```

add(int index, E e): 在指定位置添加元素

```
public void add(int index, E element) {  
    checkPositionIndex(index); //检查索引是否处于[0-size]之间  
  
    if (index == size) //添加在链表尾部  
        linkLast(element);  
    else //添加在链表中间  
        linkBefore(element, node(index));  
}
```

linkBefore方法需要给定两个参数，一个插入节点的值，一个指定的node，所以我们又调用了Node(index)去找到index对应的node

addAll(Collection c): 将集合插入到链表尾部

```
public boolean addAll(Collection<? extends E> c) {  
    return addAll(size, c);  
}
```

addAll(int index, Collection c): 将集合从指定位置开始插入

```
public boolean addAll(int index, Collection<? extends E> c) {  
    //1: 检查index范围是否在size之内  
    checkPositionIndex(index);
```

```

//2:toArray()方法把集合的数据存到对象数组中
Object[] a = c.toArray();
int numNew = a.length;
if (numNew == 0)
    return false;

//3: 得到插入位置的前驱节点和后继节点
Node<E> pred, succ;
//如果插入位置为尾部, 前驱节点为last, 后继节点为null
if (index == size) {
    succ = null;
    pred = last;
}
//否则, 调用node()方法得到后继节点, 再得到前驱节点
else {
    succ = node(index);
    pred = succ.prev;
}

// 4: 遍历数据将数据插入
for (Object o : a) {
    @SuppressWarnings("unchecked") E e = (E) o;
    //创建新节点
    Node<E> newNode = new Node<>(pred, e, null);
    //如果插入位置在链表头部
    if (pred == null)
        first = newNode;
    else
        pred.next = newNode;
    pred = newNode;
}

//如果插入位置在尾部, 重置last节点
if (succ == null) {
    last = pred;
}
//否则, 将插入的链表与先前链表连接起来
else {
    pred.next = succ;
    succ.prev = pred;
}

size += numNew;
modCount++;
return true;
}

```

上面可以看出addAll方法通常包括下面四个步骤:

1. 检查index范围是否在size之内
2. toArray()方法把集合的数据存到对象数组中
3. 得到插入位置的前驱和后继节点
4. 遍历数据, 将数据插入到指定位置

addFirst(E e): 将元素添加到链表头部

```
public void addFirst(E e) {  
    linkFirst(e);  
}
```

```
private void linkFirst(E e) {  
    final Node<E> f = first;  
    final Node<E> newNode = new Node<E>(null, e, f); //新建节点，以头节点为后继节点  
    first = newNode;  
    //如果链表为空，last节点也指向该节点  
    if (f == null)  
        last = newNode;  
    //否则，将头节点的前驱指针指向新节点，也就是指向一个元素  
    else  
        f.prev = newNode;  
    size++;  
    modCount++;  
}
```

addLast(E e): 将元素添加到链表尾部，与 **add(E e)** 方法一样

```
public void addLast(E e) {  
    linkLast(e);  
}
```

2.3.3 根据位置取数据的方法

get(int index): 根据指定索引返回数据

```
public E get(int index) {  
    //检查index范围是否在size之内  
    checkElementIndex(index);  
    //调用Node(index)去找到index对应的node然后返回它的值  
    return node(index).item;  
}
```

获取头节点 (index=0) 数据方法:

```
public E getFirst() {  
    final Node<E> f = first;  
    if (f == null)  
        throw new NoSuchElementException();  
    return f.item;  
}  
public E element() {  
    return getFirst();  
}  
public E peek() {  
    final Node<E> f = first;  
    return (f == null) ? null : f.item;  
}  
public E peekFirst() {  
    final Node<E> f = first;  
    return (f == null) ? null : f.item;  
}
```

区别：

getFirst(),element(),peek(),peekFirst()

这四个获取头结点方法的区别在于对链表为空时的处理，是抛出异常还是返回null，其中**getFirst()** 和 **element()** 方法将会在链表为空时，抛出异常

element()方法的内部就是使用**getFirst()**实现的。它们会在链表为空时，抛出
NoSuchElementException

获取尾节点 (index=-1) 数据方法：

```
public E getLast() {  
    final Node<E> l = last;  
    if (l == null)  
        throw new NoSuchElementException();  
    return l.item;  
}  
  
public E peekLast() {  
    final Node<E> l = last;  
    return (l == null) ? null : l.item;  
}
```

两者区别：

getLast() 方法在链表为空时，会抛出**NoSuchElementException**，而**peekLast()** 则不会，只是会返回 **null**。

2.3.4 根据对象得到索引的方法

int indexOf(Object o): 从头遍历找

```
public int indexOf(Object o) {  
    int index = 0;  
    if (o == null) {  
        //从头遍历  
        for (Node<E> x = first; x != null; x = x.next) {  
            if (x.item == null)  
                return index;  
            index++;  
        }  
    } else {  
        //从头遍历  
        for (Node<E> x = first; x != null; x = x.next) {  
            if (o.equals(x.item))  
                return index;  
            index++;  
        }  
    }  
    return -1;  
}
```

int lastIndexOf(Object o): 从尾遍历找

```
public int lastIndexOf(Object o) {  
    int index = size;  
    if (o == null) {  
        //从尾遍历  
        for (Node<E> x = last; x != null; x = x.prev) {  
            if (o.equals(x.item))  
                return index;  
            index--;  
        }  
    }  
    return -1;  
}
```

```

        index--;
        if (x.item == null)
            return index;
    }
} else {
    //从尾遍历
    for (Node<E> x = last; x != null; x = x.prev) {
        index--;
        if (o.equals(x.item))
            return index;
    }
}
return -1;
}

```

2.3.5 检查链表是否包含某对象的方法：

contains(Object o): 检查对象o是否存在于链表中

```

public boolean contains(Object o) {
    return indexOf(o) != -1;
}

```

2.3.6 删除方法

remove() ,removeFirst(),pop(): 删除头节点

```

public E pop() {
    return removeFirst();
}
public E remove() {
    return removeFirst();
}
public E removeFirst() {
    final Node<E> f = first;
    if (f == null)
        throw new NoSuchElementException();
    return unlinkFirst(f);
}

```

removeLast(),pollLast(): 删除尾节点

```

public E removeLast() {
    final Node<E> l = last;
    if (l == null)
        throw new NoSuchElementException();
    return unlinkLast(l);
}
public E pollLast() {
    final Node<E> l = last;
    return (l == null) ? null : unlinkLast(l);
}

```

区别： removeLast()在链表为空时将抛出NoSuchElementException，而pollLast()方法返回null。

remove(Object o): 删除指定元素

```

public boolean remove(Object o) {
    //如果删除对象为null
    if (o == null) {
        //从头开始遍历
        for (Node<E> x = first; x != null; x = x.next) {
            //找到元素
            if (x.item == null) {
                //从链表中移除找到的元素
                unlink(x);
                return true;
            }
        }
    } else {
        //从头开始遍历
        for (Node<E> x = first; x != null; x = x.next) {
            //找到元素
            if (o.equals(x.item)) {
                //从链表中移除找到的元素
                unlink(x);
                return true;
            }
        }
    }
    return false;
}

```

当删除指定对象时，只需调用remove(Object o)即可，不过该方法一次只会删除一个匹配的对象，如果删除了匹配对象，返回true，否则false。

unlink(Node x) 方法：

```

E unlink(Node<E> x) {
    // assert x != null;
    final E element = x.item;
    final Node<E> next = x.next;//得到后继节点
    final Node<E> prev = x.prev;//得到前驱节点

    //删除前驱指针
    if (prev == null) {
        first = next;//如果删除的节点是头节点，令头节点指向该节点的后继节点
    } else {
        prev.next = next;//将前驱节点的后继节点指向后继节点
        x.prev = null;
    }

    //删除后继指针
    if (next == null) {
        last = prev;//如果删除的节点是尾节点，令尾节点指向该节点的前驱节点
    } else {
        next.prev = prev;
        x.next = null;
    }

    x.item = null;
    size--;
    modCount++;
    return element;
}

```

```
}
```

remove(int index): 删除指定位置的元素

```
public E remove(int index) {  
    //检查index范围  
    checkElementIndex(index);  
    //将节点删除  
    return unlink(node(index));  
}
```

2.4 LinkedList类常用方法测试

```
package list;  
  
import java.util.Iterator;  
import java.util.LinkedList;  
  
public class LinkedListDemo {  
    public static void main(String[] args) {  
        //创建存放int类型的LinkedList  
        LinkedList<Integer> linkedList = new LinkedList<>();  
        /***** LinkedList的基本操作 *****/  
        linkedList.addFirst(0); // 添加元素到列表开头  
        linkedList.add(1); // 在列表结尾添加元素  
        linkedList.add(2, 2); // 在指定位置添加元素  
        linkedList.addLast(3); // 添加元素到列表结尾  
  
        System.out.println("LinkedList (直接输出的) : " + linkedList);  
  
        System.out.println("getFirst() 获得第一个元素: " + linkedList.getFirst());  
        // 返回此列表的第一个元素  
        System.out.println("getLast() 获得最后一个元素: " + linkedList.getLast());  
        // 返回此列表的最后一个元素  
        System.out.println("removeFirst() 删除第一个元素并返回: " +  
        linkedList.removeFirst()); // 移除并返回此列表的第一个元素  
        System.out.println("removeLast() 删除最后一个元素并返回: " +  
        linkedList.removeLast()); // 移除并返回此列表的最后一个元素  
        System.out.println("After remove: " + linkedList);  
        System.out.println("contains() 方法判断列表是否包含1这个元素: " +  
        linkedList.contains(1)); // 判断此列表包含指定元素, 如果是, 则返回true  
        System.out.println("该LinkedList的大小 : " + linkedList.size()); // 返回此  
        // 列表的元素个数  
  
        /***** 位置访问操作 *****/  
        System.out.println("-----");  
        linkedList.set(1, 3); // 将此列表中指定位置的元素替换为指定的元素  
        System.out.println("After set(1, 3): " + linkedList);  
        System.out.println("get(1) 获得指定位置 (这里为1) 的元素: " +  
        linkedList.get(1)); // 返回此列表中指定位置处的元素  
  
        /***** Search操作 *****/  
        System.out.println("-----");  
        linkedList.add(3);  
        System.out.println("indexOf(3): " + linkedList.indexOf(3)); // 返回此列表  
        // 中首次出现的指定元素的索引
```

```
System.out.println("lastIndexOf(3): " + linkedList.lastIndexOf(3)); // 返回此列表中最后出现的指定元素的索引

头
    **** Queue操作 ****
    System.out.println("-----");
    System.out.println("peek(): " + linkedList.peek()); // 获取但不移除此列表的头
    System.out.println("element(): " + linkedList.element()); // 获取但不移除此列表的头
    linkedList.poll(); // 获取并移除此列表的头
    System.out.println("After poll(): " + linkedList);
    linkedList.remove();
    System.out.println("After remove(): " + linkedList); // 获取并移除此列表的头
    linkedList.offer(4);
    System.out.println("After offer(4): " + linkedList); // 将指定元素添加到此列表的末尾

素
    **** Deque操作 ****
    System.out.println("-----");
    linkedList.offerFirst(2); // 在此列表的开头插入指定的元素
    System.out.println("After offerFirst(2): " + linkedList);
    linkedList.offerLast(5); // 在此列表末尾插入指定的元素
    System.out.println("After offerLast(5): " + linkedList);
    System.out.println("peekFirst(): " + linkedList.peekFirst()); // 获取但不移除此列表的第一个元素
    System.out.println("peekLast(): " + linkedList.peekLast()); // 获取但不移除此列表的第一个元素
    linkedList.pollFirst(); // 获取并移除此列表的第一个元素
    System.out.println("After pollFirst(): " + linkedList);
    linkedList.pollLast(); // 获取并移除此列表的最后一个元素
    System.out.println("After pollLast(): " + linkedList);
    linkedList.push(2); // 将元素推入此列表所表示的堆栈（插入到列表的头）
    System.out.println("After push(2): " + linkedList);
    linkedList.pop(); // 从此列表所表示的堆栈处弹出一个元素（获取并移除列表第一个元素）
    System.out.println("After pop(): " + linkedList);
    linkedList.add(3);
    linkedList.removeFirstOccurrence(3); // 从此列表中移除第一次出现的指定元素（从头部到尾部遍历列表）
    System.out.println("After removeFirstOccurrence(3): " + linkedList);
    linkedList.removeLastOccurrence(3); // 从此列表中移除最后一次出现的指定元素（从尾部到头部遍历列表）
    System.out.println("After removeFirstOccurrence(3): " + linkedList);

    **** 遍历操作 ****
    System.out.println("-----");
    linkedList.clear();
    for (int i = 0; i < 100000; i++) {
        linkedList.add(i);
    }
    // 迭代器遍历
    long start = System.currentTimeMillis();
    Iterator<Integer> iterator = linkedList.iterator();
    while (iterator.hasNext()) {
        iterator.next();
    }
    long end = System.currentTimeMillis();
    System.out.println("Iterator: " + (end - start) + " ms");
```

```

// 顺序遍历(随机遍历)
start = System.currentTimeMillis();
for (int i = 0; i < linkedList.size(); i++) {
    linkedList.get(i);
}
end = System.currentTimeMillis();
System.out.println("for: " + (end - start) + " ms");

// 另一种for循环遍历
start = System.currentTimeMillis();
for (Integer i : linkedList)
    ;
end = System.currentTimeMillis();
System.out.println("for2: " + (end - start) + " ms");

// 通过pollFirst()或pollLast()来遍历LinkedList
LinkedList<Integer> temp1 = new LinkedList<>();
temp1.addAll(linkedList);
start = System.currentTimeMillis();
while (temp1.size() != 0) {
    temp1.pollFirst();
}
end = System.currentTimeMillis();
System.out.println("pollFirst()或pollLast(): " + (end - start) + " ms");

// 通过removeFirst()或removeLast()来遍历LinkedList
LinkedList<Integer> temp2 = new LinkedList<>();
temp2.addAll(linkedList);
start = System.currentTimeMillis();
while (temp2.size() != 0) {
    temp2.removeFirst();
}
end = System.currentTimeMillis();
System.out.println("removeFirst()或removeLast(): " + (end - start) + " ms");
}
}
}

```

3. HashMap

3.1 HashMap 简介

HashMap 主要用来存放键值对，它基于哈希表的Map接口实现，是常用的Java集合之一。

JDK1.8 之前 HashMap 由 数组+链表 组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的（“拉链法”解决冲突）。JDK1.8 以后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）时，将链表转化为红黑树（将链表转换成红黑树前会判断，如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树），以减少搜索时间，具体可以参考 `treeifyBin` 方法。

3.2 底层数据结构分析

3.2.1 JDK1.8之前

JDK1.8 之前 HashMap 底层是 **数组和链表** 结合在一起使用也就是 **链表散列**。HashMap 通过 key 的 hashCode 经过扰动函数处理过后得到 hash 值，然后通过 `(n - 1) & hash` 判断当前元素存放的位置（这里的 n 指的是数组的长度），如果当前位置存在元素的话，就判断该元素与要存入的元素的 hash 值以及 key 是否相同，如果相同的话，直接覆盖，不相同就通过拉链法解决冲突。

所谓扰动函数指的就是 HashMap 的 hash 方法。使用 hash 方法也就是扰动函数是为了防止一些实现比较差的 hashCode() 方法 换句话说使用扰动函数之后可以减少碰撞。

JDK 1.8 HashMap 的 hash 方法源码：

JDK 1.8 的 hash方法相比于 JDK 1.7 hash 方法更加简化，但是原理不变。

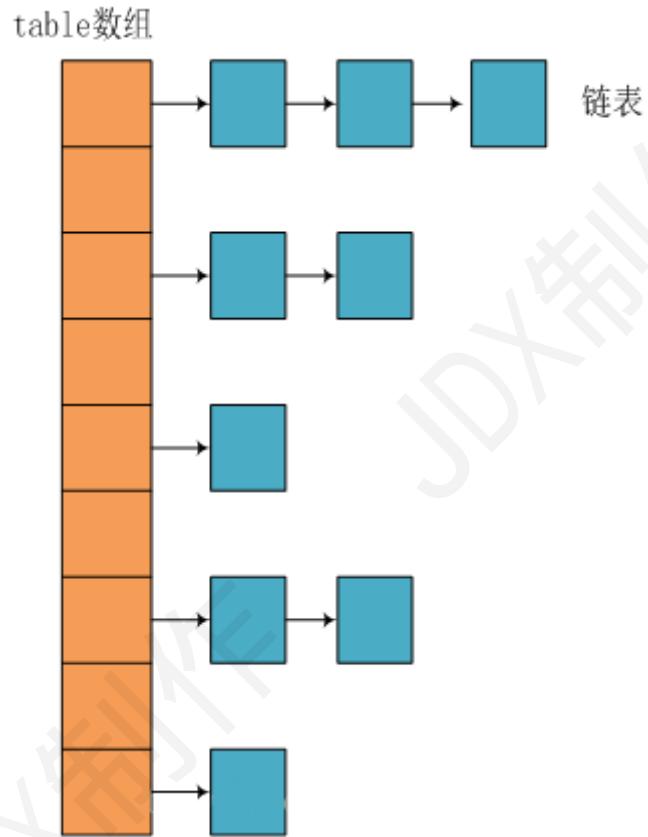
```
static final int hash(Object key) {  
    int h;  
    // key.hashCode(): 返回散列值也就是hashcode  
    // ^ : 按位异或  
    // >>>:无符号右移，忽略符号位，空位都以0补齐  
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
}
```

对比一下 JDK1.7 的 HashMap 的 hash 方法源码.

```
static int hash(int h) {  
    // This function ensures that hashCodes that differ only by  
    // constant multiples at each bit position have a bounded  
    // number of collisions (approximately 8 at default load factor).  
  
    h ^= (h >>> 20) ^ (h >>> 12);  
    return h ^ (h >>> 7) ^ (h >>> 4);  
}
```

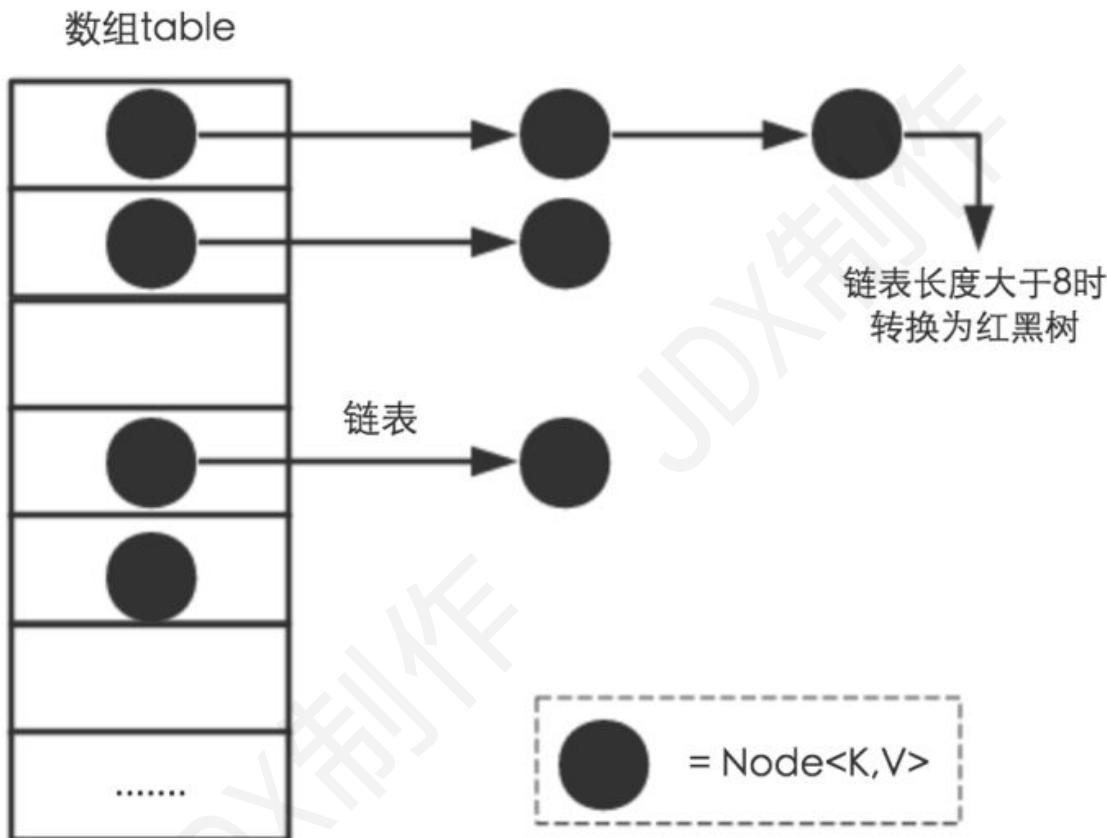
相比于 JDK1.8 的 hash 方法，JDK 1.7 的 hash 方法的性能会稍差一点点，因为毕竟扰动了 4 次。

所谓“**拉链法**”就是：将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可。



3.2.2 JDK1.8之后

相比于之前的版本，jdk1.8在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间。



类的属性：

```

public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>,
cloneable, Serializable {
    // 序列号
    private static final long serialVersionUID = 362498820763181265L;
    // 默认的初始容量是16
    static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;
    // 最大容量
    static final int MAXIMUM_CAPACITY = 1 << 30;
    // 默认的填充因子
    static final float DEFAULT_LOAD_FACTOR = 0.75f;
    // 当桶(bucket)上的结点数大于这个值时会转成红黑树
    static final int TREEIFY_THRESHOLD = 8;
    // 当桶(bucket)上的结点数小于这个值时树转链表
    static final int UNTREEIFY_THRESHOLD = 6;
    // 桶中结构转化为红黑树对应的table的最小大小
    static final int MIN_TREEIFY_CAPACITY = 64;
    // 存储元素的数组，总是2的幂次倍
    transient Node<k,v>[] table;
    // 存放具体元素的集
    transient Set<map.entry<k,v>> entrySet;
    // 存放元素的个数，注意这个不等于数组的长度。
    transient int size;
    // 每次扩容和更改map结构的计数器
    transient int modCount;
    // 临界值 当实际大小(容量*填充因子)超过临界值时，会进行扩容
    int threshold;
    // 加载因子
    final float loadFactor;
}

```

- **loadFactor加载因子**

loadFactor加载因子是控制数组存放数据的疏密程度，loadFactor越趋近于1，那么 数组中存放的数据(entry)也就越多，也就越密，也就是会让链表的长度增加，loadFactor越小，也就是趋近于0，数组中存放的数据(entry)也就越少，也就越稀疏。

loadFactor太大导致查找元素效率低，太小导致数组的利用率低，存放的数据会很分散。

loadFactor的默认值为0.75f是官方给出的一个比较好的临界值。

给定的默认容量为 16，负载因子为 0.75。Map 在使用过程中不断的往里面存放数据，当数量达到了 $16 * 0.75 = 12$ 就需要将当前 16 的容量进行扩容，而扩容这个过程涉及到 rehash、复制数据等操作，所以非常消耗性能。

- **threshold**

threshold = capacity * loadFactor，当Size>=threshold的时候，那么就要考虑对数组的扩增了，也就是说，这个的意思就是衡量数组是否需要扩增的一个标准。

Node节点类源码：

```

// 继承自 Map.Entry<K,V>
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;// 哈希值，存放元素到hashmap中时用来与其他元素hash值比较
    final K key;//键
    V value;//值
    // 指向下一个节点
    Node<K,V> next;
    Node(int hash, K key, V value, Node<K,V> next) {
        this.hash = hash;
    }
}

```

```

        this.key = key;
        this.value = value;
        this.next = next;
    }
    public final K getKey() { return key; }
    public final V getValue() { return value; }
    public final String toString() { return key + "=" + value; }
    // 重写hashCode()方法
    public final int hashCode() {
        return Objects.hashCode(key) ^ Objects.hashCode(value);
    }

    public final V setValue(V newValue) {
        V oldValue = value;
        value = newValue;
        return oldValue;
    }
    // 重写 equals() 方法
    public final boolean equals(Object o) {
        if (o == this)
            return true;
        if (o instanceof Map.Entry) {
            Map.Entry<?, ?> e = (Map.Entry<?, ?>)o;
            if (Objects.equals(key, e.getKey()) &&
                Objects.equals(value, e.getValue()))
                return true;
        }
        return false;
    }
}

```

树节点类源码:

```

static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
    TreeNode<K,V> parent; // 父
    TreeNode<K,V> left; // 左
    TreeNode<K,V> right; // 右
    TreeNode<K,V> prev; // needed to unlink next upon deletion
    boolean red; // 判断颜色
    TreeNode(int hash, K key, V val, Node<K,V> next) {
        super(hash, key, val, next);
    }
    // 返回根节点
    final TreeNode<K,V> root() {
        for (TreeNode<K,V> r = this, p;;) {
            if ((p = r.parent) == null)
                return r;
            r = p;
        }
    }
}

```

3.3 HashMap源码分析

3.3.1 构造方法

HashMap 中有四个构造方法，它们分别如下：

// 默认构造函数。

```

public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
}

// 包含另一个“Map”的构造函数
public HashMap(Map<? extends K, ? extends V> m) {
    this.loadFactor = DEFAULT_LOAD_FACTOR;
    putMapEntries(m, false); //下面会分析到这个方法
}

// 指定“容量大小”的构造函数
public HashMap(int initialCapacity) {
    this(initialCapacity, DEFAULT_LOAD_FACTOR);
}

// 指定“容量大小”和“加载因子”的构造函数
public HashMap(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal initial capacity: " +
initialCapacity);
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal load factor: " +
loadFactor);
    this.loadFactor = loadFactor;
    this.threshold = tableSizeFor(initialCapacity);
}

```

putMapEntries方法:

```

final void putMapEntries(Map<? extends K, ? extends V> m, boolean evict) {
    int s = m.size();
    if (s > 0) {
        // 判断table是否已经初始化
        if (table == null) { // pre-size
            // 未初始化, s为m的实际元素个数
            float ft = ((float)s / loadFactor) + 1.0F;
            int t = ((ft < (float)MAXIMUM_CAPACITY) ?
                (int)ft : MAXIMUM_CAPACITY);
            // 计算得到的t大于阈值, 则初始化阈值
            if (t > threshold)
                threshold = tableSizeFor(t);
        }
        // 已初始化, 并且m元素个数大于阈值, 进行扩容处理
        else if (s > threshold)
            resize();
        // 将m中的所有元素添加至HashMap中
        for (Map.Entry<? extends K, ? extends V> e : m.entrySet()) {
            K key = e.getKey();
            V value = e.getValue();
            putVal(hash(key), key, value, false, evict);
        }
    }
}

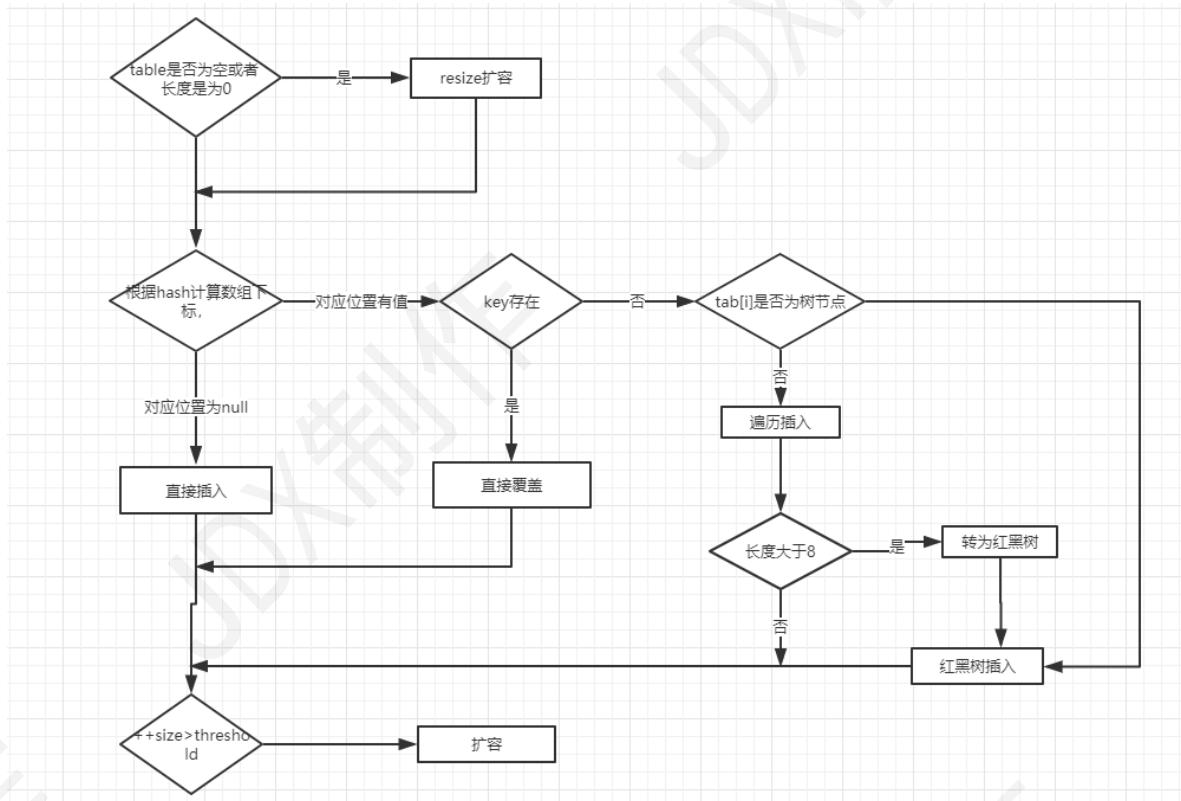
```

3.3.2 put方法

HashMap只提供了put用于添加元素，putVal方法只是给put方法调用的一个方法，并没有提供给用户使用。

对putVal方法添加元素的分析如下：

- ①如果定位到的数组位置没有元素就直接插入。
- ②如果定位到的数组位置有元素就和要插入的key比较，如果key相同就直接覆盖，如果key不相同，就判断p是否是一个树节点，如果是就调用 e = ((TreeNode<K,V>)p).putTreeval(this, tab, hash, key, value) 将元素添加进入。如果不是就遍历链表插入(插入的是链表尾部)。



```
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    // table未初始化或者长度为0, 进行扩容
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // (n - 1) & hash 确定元素存放在哪个桶中, 桶为空, 新生成结点放入桶中(此时, 这个结点是放在数组中)
    if (((p = tab[i = (n - 1) & hash]) == null))
        tab[i] = newNode(hash, key, value, null);
    // 桶中已经存在元素
    else {
        Node<K,V> e; K k;
        // 比较桶中第一个元素(数组中的结点)的hash值相等, key相等
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            // 将第一个元素赋值给e, 用e来记录
            e = p;
        // hash值不相等, 即key不相等; 为红黑树结点
        else if (p instanceof TreeNode)
```

```

        // 放入树中
        e = ((TreeNode<K,V>)p).putTreeval(this, tab, hash, key, value);
    // 为链表结点
    else {
        // 在链表最末插入结点
        for (int binCount = 0; ; ++binCount) {
            // 到达链表的尾部
            if ((e = p.next) == null) {
                // 在尾部插入新结点
                p.next = newNode(hash, key, value, null);
                // 结点数量达到阈值，转化为红黑树
                if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                    treeifyBin(tab, hash);
                // 跳出循环
                break;
            }
            // 判断链表中结点的key值与插入的元素的key值是否相等
            if (e.hash == hash &&
                ((k = e.key) == key || (key != null && key.equals(k))))
                // 相等，跳出循环
                break;
            // 用于遍历桶中的链表，与前面的e = p.next组合，可以遍历链表
            p = e;
        }
    }
    // 表示在桶中找到key值、hash值与插入元素相等的结点
    if (e != null) {
        // 记录e的value
        v.oldvalue = e.value;
        // onlyIfAbsent为false或者旧值为null
        if (!onlyIfAbsent || oldvalue == null)
            // 用newValue替换旧值
            e.value = value;
        // 访问后回调
        afterNodeAccess(e);
        // 返回旧值
        return oldvalue;
    }
}
// 结构性修改
++modCount;
// 实际大小大于阈值则扩容
if (++size > threshold)
    resize();
// 插入后回调
afterNodeInsertion(evict);
return null;
}

```

我们再来对比一下 JDK1.7 put方法的代码

对于put方法的分析如下：

- ①如果定位到的数组位置没有元素 就直接插入。
- ②如果定位到的数组位置有元素，遍历以这个元素为头结点的链表，依次和插入的key比较，如果key相同就直接覆盖，不同就采用头插法插入元素。

```

public V put(K key, V value) {
    if (table == EMPTY_TABLE) {
        inflateTable(threshold);
    }
    if (key == null)
        return putForNullKey(value);
    int hash = hash(key);
    int i = indexFor(hash, table.length);
    for (Entry<K,V> e = table[i]; e != null; e = e.next) { // 先遍历
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }
    modCount++;
    addEntry(hash, key, value, i); // 再插入
    return null;
}

```

3.4 get方法

```

public V get(Object key) {
    Node<K,V> e;
    return (e = getNode(hash(key), key)) == null ? null : e.value;
}

final Node<K,V> getNode(int hash, Object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & hash]) != null) {
        // 数组元素相等
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equals(k))))
            return first;
        // 桶中不止一个节点
        if ((e = first.next) != null) {
            // 在树中get
            if (first instanceof TreeNode)
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
            // 在链表中get
            do {
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    return e;
                } while ((e = e.next) != null);
        }
    }
    return null;
}

```

3.5 resize方法

进行扩容，会伴随着一次重新hash分配，并且会遍历hash表中所有的元素，是非常耗时的。在编写程序中，要尽量避免resize。

```
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {
        // 超过最大值就不再扩充了，就只好随你碰撞去吧
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        // 没超过最大值，就扩充为原来的2倍
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY && oldCap >=
        DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
    else {
        // signifies using defaults
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
    // 计算新的resize上限
    if (newThr == 0) {
        float ft = (float)newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
        (int)ft : Integer.MAX_VALUE);
    }
    threshold = newThr;
    @SuppressWarnings({"rawtypes","unchecked"})
    Node<K,V>[] newTab = (Node<K,V>[])(new Node[newCap]);
    table = newTab;
    if (oldTab != null) {
        // 把每个bucket都移动到新的buckets中
        for (int j = 0; j < oldCap; ++j) {
            Node<K,V> e;
            if ((e = oldTab[j]) != null) {
                oldTab[j] = null;
                if (e.next == null)
                    newTab[e.hash & (newCap - 1)] = e;
                else if (e instanceof TreeNode)
                    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                else {
                    Node<K,V> loHead = null, loTail = null;
                    Node<K,V> hiHead = null, hiTail = null;
                    Node<K,V> next;
                    do {
                        next = e.next;
                        // 原索引
                        if ((e.hash & oldCap) == 0) {
                            if (loTail == null)
                                loHead = e;
                            else
```

```

        loTail.next = e;
        loTail = e;
    }
    // 原索引+oldCap
    else {
        if (hiTail == null)
            hiHead = e;
        else
            hiTail.next = e;
        hiTail = e;
    }
} while ((e = next) != null);
// 原索引放到bucket里
if (loTail != null) {
    loTail.next = null;
    newTab[j] = loHead;
}
// 原索引+oldCap放到bucket里
if (hiTail != null) {
    hiTail.next = null;
    newTab[j + oldCap] = hiHead;
}
}
}
}
}
return newTab;
}
}

```

3.6 HashMap常用方法测试

```

package map;

import java.util.Collection;
import java.util.HashMap;
import java.util.Set;

public class HashMapDemo {

    public static void main(String[] args) {
        HashMap<String, String> map = new HashMap<String, String>();
        // 键不能重复，值可以重复
        map.put("san", "张三");
        map.put("si", "李四");
        map.put("wu", "王五");
        map.put("wang", "老王");
        map.put("wang", "老王2");// 老王被覆盖
        map.put("lao", "老王");
        System.out.println("-----直接输出 hashmap:-----");
        System.out.println(map);
        /**
         * 遍历HashMap
         */
        // 1. 获取Map中的所有键
        System.out.println("-----foreach获取Map中所有的键:-----");
        Set<String> keys = map.keySet();
        for (String key : keys) {

```

```

        System.out.print(key+"  ");
    }
    System.out.println();//换行
    // 2. 获取Map中所有值
    System.out.println("-----foreach获取Map中所有的值:-----");
    Collection<String> values = map.values();
    for (String value : values) {
        System.out.print(value+"  ");
    }
    System.out.println();//换行
    // 3. 得到key的值的同时得到key所对应的值
    System.out.println("-----得到key的值的同时得到key所对应的值:-----");
    Set<String> keys2 = map.keySet();
    for (String key : keys2) {
        System.out.print(key + ": " + map.get(key)+"  ");
    }
}

/**
 * 如果既要遍历key又要value，那么建议这种方式，应为如果先获取keyset然后再执行
map.get(key)，map内部会执行两次遍历。
 * 一次是在获取keyset的时候，一次是在遍历所有key的时候。
 */
// 当我调用put(key,value)方法的时候，首先会把key和value封装到
// Entry这个静态内部类对象中，把Entry对象再添加到数组中，所以我们想获取
// map中的所有键值对，我们只要获取数组中的所有Entry对象，接下来
// 调用Entry对象中的getKey()和getValue()方法就能获取键值对了
Set<java.util.Map.Entry<String, String>> entrys = map.entrySet();
for (java.util.Map.Entry<String, String> entry : entrys) {
    System.out.println(entry.getKey() + "--" + entry.getValue());
}

/**
 * HashMap其他常用方法
 */
System.out.println("after map.size(): "+map.size());
System.out.println("after map.isEmpty(): "+map.isEmpty());
System.out.println(map.remove("san"));
System.out.println("after map.remove(): "+map);
System.out.println("after map.get(si): "+map.get("si"));
System.out.println("after map.containsKey(si): "+map.containsKey("si"));
System.out.println("after containsValue(李四): "+map.containsValue("李
四"));
System.out.println(map.replace("si", "李四2"));
System.out.println("after map.replace(si, 李四2): "+map);
}
}

```

(三). 并发

1. 并发容器

1.1 JDK 提供的并发容器总结

JDK 提供的这些容器大部分在 `java.util.concurrent` 包中。

- **ConcurrentHashMap**: 线程安全的 HashMap
- **CopyOnWriteArrayList**: 线程安全的 List，在读多写少的场合性能非常好，远远好于 Vector。
- **ConcurrentLinkedQueue**: 高效的并发队列，使用链表实现。可以看做一个线程安全的 LinkedList，这是一个非阻塞队列。
- **BlockingQueue**: 这是一个接口，JDK 内部通过链表、数组等方式实现了这个接口。表示阻塞队列，非常适合用于作为数据共享的通道。
- **ConcurrentSkipListMap**: 跳表的实现。这是一个 Map，使用跳表的数据结构进行快速查找。

1.2 ConcurrentHashMap

我们知道 HashMap 不是线程安全的，在并发场景下如果要保证一种可行的方式是使用 `Collections.synchronizedMap()` 方法来包装我们的 HashMap。但这是通过使用一个全局的锁来同步不同线程间的并发访问，因此会带来不可忽视的性能问题。

所以就有了 HashMap 的线程安全版本—— ConcurrentHashMap 的诞生。在 ConcurrentHashMap 中，无论是读操作还是写操作都能保证很高的性能：在进行读操作时(几乎)不需要加锁，而在写操作时通过锁分段技术只对所操作的段加锁而不影响客户端对其它段的访问。

1.3 CopyOnWriteArrayList

1.3.1 CopyOnWriteArrayList 简介

```
public class CopyOnwriteArrayList<E>
extends Object
implements List<E>, RandomAccess, Cloneable, Serializable
```

在很多应用场景中，读操作可能会远远大于写操作。由于读操作根本不会修改原有的数据，因此对于每次读取都进行加锁其实是一种资源浪费。我们应该允许多个线程同时访问 List 的内部数据，毕竟读取操作是安全的。

这和我们之前在多线程章节讲过 `ReentrantReadWriteLock` 读写锁的思想非常类似，也就是读读共享、写写互斥、读写互斥、写读互斥。JDK 中提供了 `copyonwriteArrayList` 类比相比于在读写锁的思想又更进一步。为了将读取的性能发挥到极致，`CopyOnwriteArrayList` 读取是完全不用加锁的，并且更厉害的是：写入也不会阻塞读取操作。只有写入和写入之间需要进行同步等待。这样一来，读操作的性能就会大幅度提升。**那它是怎么做的呢？**

1.3.2 CopyOnWriteArrayList 是如何做到的？

`CopyOnwriteArrayList` 类的所有可变操作（add, set 等等）都是通过创建底层数组的新副本实现的。当 List 需要被修改的时候，我并不修改原有内容，而是对原有数据进行一次复制，将修改的内容写入副本。写完之后，再将修改完的副本替换原来的数据，这样就可以保证写操作不会影响读操作了。

从 `CopyOnwriteArrayList` 的名字就能看出 `CopyOnwriteArrayList` 是满足 `CopyOnwrite` 的 `ArrayList`，所谓 `CopyOnwrite` 也就是说：在计算机，如果你想要对一块内存进行修改时，我们不在原有内存块中进行写操作，而是将内存拷贝一份，在新的内存中进行写操作，写完之后呢，就将指向原来内存指针指向新的内存，原来的内存就可以被回收掉了。

1.3.3 CopyOnWriteArrayList 读取和写入源码简单分析

1.3.3.1 CopyOnWriteArrayList 读取操作的实现

读取操作没有任何同步控制和锁操作，理由就是内部数组 array 不会发生修改，只会被另外一个 array 替换，因此可以保证数据安全。

```

/** The array, accessed only via getArray/setArray. */
private transient volatile Object[] array;
public E get(int index) {
    return get(getArray(), index);
}
@SuppressWarnings("unchecked")
private E get(Object[] a, int index) {
    return (E) a[index];
}
final Object[] getArray() {
    return array;
}

```

1.3.3.2 CopyOnWriteArrayList 写入操作的实现

CopyOnWriteArrayList 写入操作 add() 方法在添加集合的时候加了锁，保证了同步，避免了多线程写的时候会 copy 出多个副本出来。

[获取更多资源可添加关注微信公众号：JAVA架构进阶之路；或添加微信：jiang10086a 免费获取面试真题教程，笔记等。](#)

```

* Appends the specified element to the end of this list.
*
* @param e element to be appended to this list
* @return {@code true} (as specified by {@link Collection#add})
*/
public boolean add(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock(); //加锁
    try {
        Object[] elements = getArray();
        int len = elements.length;
        Object[] newElements = Arrays.copyOf(elements, len + 1); //拷贝新数组
        newElements[len] = e;
        setArray(newElements);
        return true;
    } finally {
        lock.unlock(); //释放锁
    }
}

```

1.4 ConcurrentLinkedQueue

Java 提供的线程安全的 Queue 可以分为**阻塞队列**和**非阻塞队列**，其中阻塞队列的典型例子是 BlockingQueue，非阻塞队列的典型例子是 ConcurrentLinkedQueue，在实际应用中要根据实际需要选用阻塞队列或者非阻塞队列。**阻塞队列可以通过加锁来实现，非阻塞队列可以通过 CAS 操作实现。**

从名字可以看出，ConcurrentLinkedQueue 这个队列使用链表作为其数据结构。

ConcurrentLinkedQueue 应该算是在高并发环境中性能最好的队列了。它之所以能有很好的性能，是因为其内部复杂的实现。

ConcurrentLinkedQueue 内部代码我们就不分析了，大家知道 ConcurrentLinkedQueue 主要使用 CAS 非阻塞算法来实现线程安全就好了。

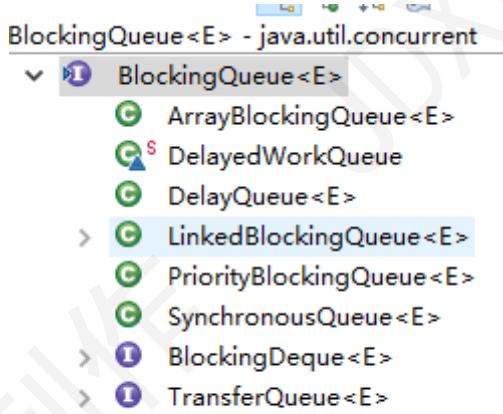
ConcurrentLinkedQueue 适合在对性能要求相对较高，同时对队列的读写存在多个线程同时进行的场景，即如果对队列加锁的成本较高则适合使用无锁的 ConcurrentLinkedQueue 来替代。

1.5 BlockingQueue

1.5.1 BlockingQueue 简单介绍

上面我们已经提到了 ConcurrentLinkedQueue 作为高性能的非阻塞队列。下面我们要讲到的是阻塞队列——BlockingQueue。阻塞队列（BlockingQueue）被广泛使用在“生产者-消费者”问题中，其原因是 BlockingQueue 提供了可阻塞的插入和移除的方法。当队列容器已满，生产者线程会被阻塞，直到队列未满；当队列容器为空时，消费者线程会被阻塞，直至队列非空时为止。

BlockingQueue 是一个接口，继承自 Queue，所以其实现类也可以作为 Queue 的实现来使用，而 Queue 又继承自 Collection 接口。下面是 BlockingQueue 的相关实现类：



下面主要介绍一下：ArrayBlockingQueue、LinkedBlockingQueue、PriorityBlockingQueue，这三个 BlockingQueue 的实现类。

1.5.2 ArrayBlockingQueue

ArrayBlockingQueue 是 BlockingQueue 接口的有界队列实现类，底层采用数组来实现。

ArrayBlockingQueue 一旦创建，容量不能改变。其并发控制采用可重入锁来控制，不管是插入操作还是读取操作，都需要获取到锁才能进行操作。当队列容量满时，尝试将元素放入队列将导致操作阻塞；尝试从一个空队列中取一个元素也会同样阻塞。

ArrayBlockingQueue 默认情况下不能保证线程访问队列的公平性，所谓公平性是指严格按照线程等待的绝对时间顺序，即最先等待的线程能够最先访问到 ArrayBlockingQueue。而非公平性则是指访问 ArrayBlockingQueue 的顺序不是遵守严格的时间顺序，有可能存在，当 ArrayBlockingQueue 可以被访问时，长时间阻塞的线程依然无法访问到 ArrayBlockingQueue。如果保证公平性，通常会降低吞吐量。如果需要获得公平性的 ArrayBlockingQueue，可采用如下代码：

```
private static ArrayBlockingQueue<Integer> blockingQueue = new  
ArrayBlockingQueue<Integer>(10, true);
```

1.5.3 LinkedBlockingQueue

LinkedBlockingQueue 底层基于单向链表实现的阻塞队列，可以当做无界队列也可以当做有界队列来使用，同样满足 FIFO 的特性，与 ArrayBlockingQueue 相比起来具有更高的吞吐量，为了防止 LinkedBlockingQueue 容量迅速增，损耗大量内存。通常在创建 LinkedBlockingQueue 对象时，会指定其大小，如果未指定，容量等于 Integer.MAX_VALUE。

相关构造方法：

```
/**  
 * 某种意义上的无界队列  
 * Creates a {@code LinkedBlockingQueue} with a capacity of  
 * {@link Integer#MAX_VALUE}.  
 */
```

```

/*
public LinkedBlockingQueue() {
    this(Integer.MAX_VALUE);
}

/**
 * 有界队列
 * Creates a {@code LinkedBlockingQueue} with the given (fixed) capacity.
 *
 * @param capacity the capacity of this queue
 * @throws IllegalArgumentException if {@code capacity} is not greater
 *         than zero
 */
public LinkedBlockingQueue(int capacity) {
    if (capacity <= 0) throw new IllegalArgumentException();
    this.capacity = capacity;
    last = head = new Node<E>(null);
}

```

1.5.4 PriorityBlockingQueue

PriorityBlockingQueue 是一个支持优先级的无界阻塞队列。默认情况下元素采用自然顺序进行排序，也可以通过自定义类实现 `compareTo()` 方法来指定元素排序规则，或者初始化时通过构造器参数 `Comparator` 来指定排序规则。

PriorityBlockingQueue 并发控制采用的是 **ReentrantLock**，队列为无界队列（`ArrayBlockingQueue` 是有界队列，`LinkedBlockingQueue` 也可以通过在构造函数中传入 `capacity` 指定队列最大的容量，但是 `PriorityBlockingQueue` 只能指定初始的队列大小，后面插入元素的时候，**如果空间不够的话会自动扩容**）。

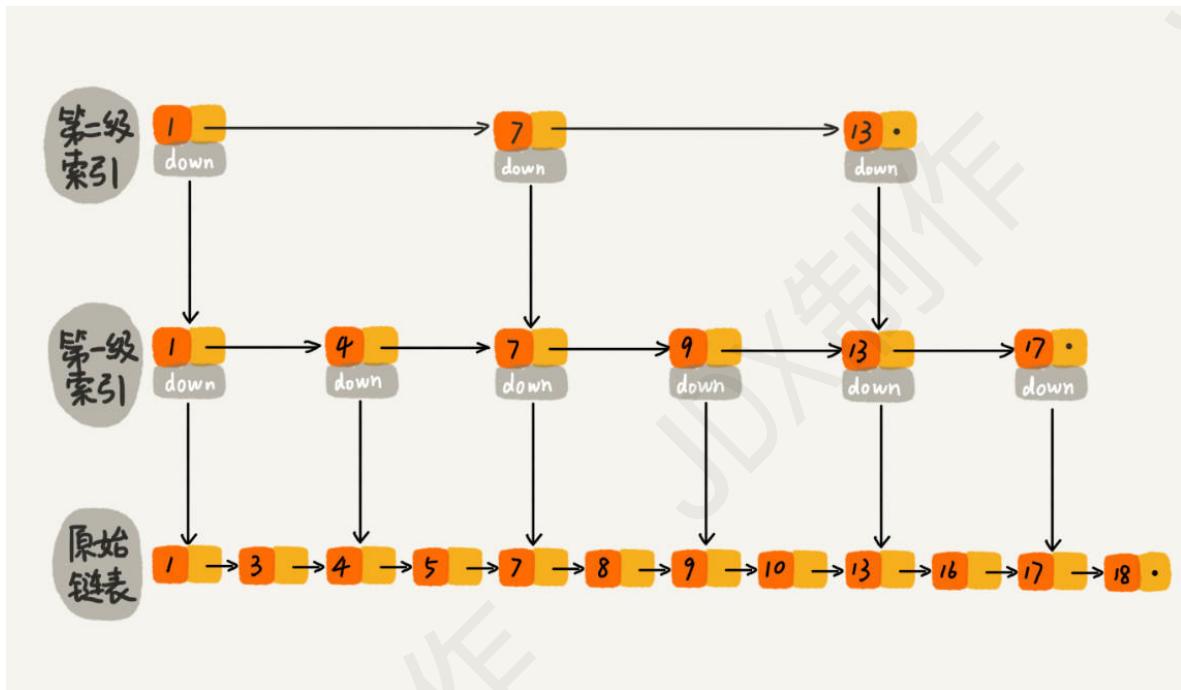
简单地说，它就是 `PriorityQueue` 的线程安全版本。不可以插入 `null` 值，同时，插入队列的对象必须是可比较大小的（`comparable`），否则报 `ClassCastException` 异常。它的插入操作 `put` 方法不会 `block`，因为它是无界队列（`take` 方法在队列为空的时候会阻塞）。

1.6 ConcurrentSkipListMap

为了引出 `ConcurrentSkipListMap`，先带着大家简单理解一下跳表。

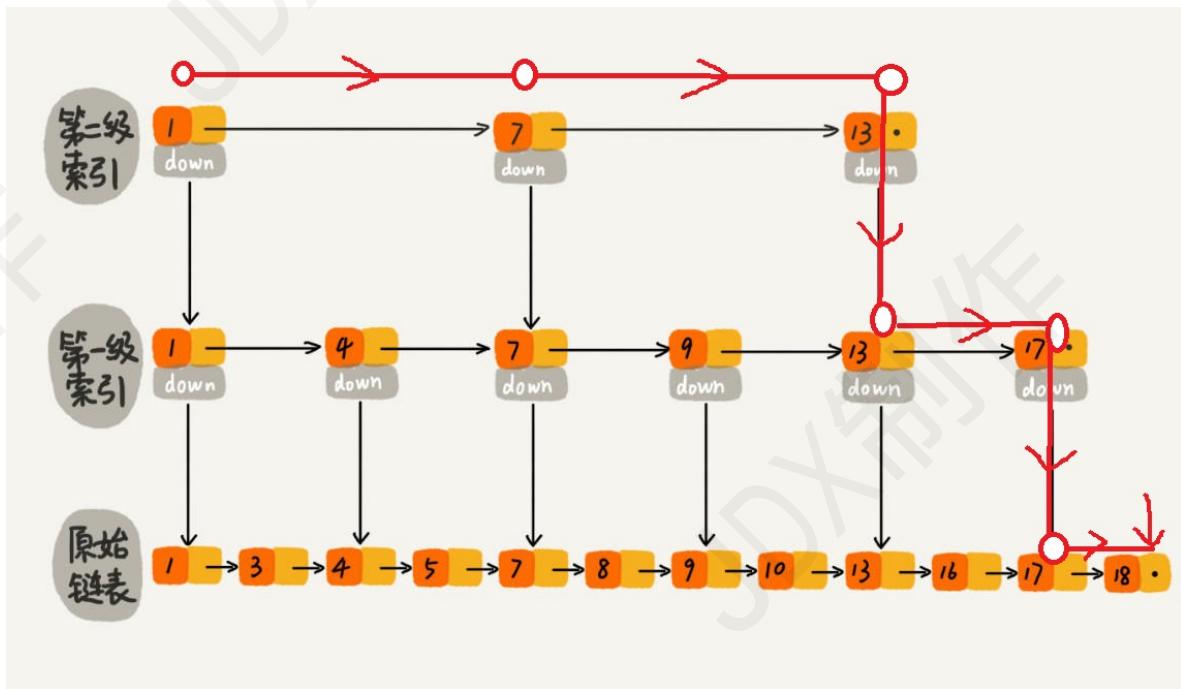
对于一个单链表，即使链表是有序的，如果我们想要在其中查找某个数据，也只能从头到尾遍历链表，这样效率自然就会很低，跳表就不一样了。跳表是一种可以用来快速查找的数据结构，有点类似于平衡树。它们都可以对元素进行快速的查找。但一个重要的区别是：对平衡树的插入和删除往往很可能导致平衡树进行一次全局的调整。而对跳表的插入和删除只需要对整个数据结构的局部进行操作即可。这样带来的好处是：在高并发的情况下，你会需要一个全局锁来保证整个平衡树的线程安全。而对于跳表，你只需要部分锁即可。这样，在高并发环境下，你就可以拥有更好的性能。而就查询的性能而言，跳表的时间复杂度也是 **O(logn)** 所以在并发数据结构中，JDK 使用跳表来实现一个 Map。

跳表的本质是同时维护了多个链表，并且链表是分层的，



最低层的链表维护了跳表内所有的元素，每上面一层链表都是下面一层的子集。

跳表内的所有链表的元素都是排序的。查找时，可以从顶级链表开始找。一旦发现被查找的元素大于当前链表中的取值，就会转入下一层链表继续找。这也就是说在查找过程中，搜索是跳跃式的。如上图所示，在跳表中查找元素 18。



查找 18 的时候原来需要遍历 18 次，现在只需要 7 次即可。针对链表长度比较大的时候，构建索引查找效率的提升就会非常明显。

从上面很容易看出，**跳表是一种利用空间换时间的算法。**

使用跳表实现 Map 和使用哈希算法实现 Map 的另外一个不同之处是：哈希并不会保存元素的顺序，而跳表内所有的元素都是排序的。因此在对跳表进行遍历时，你会得到一个有序的结果。所以，如果你的应用需要有序性，那么跳表就是你不二的选择。JDK 中实现这一数据结构的类是 ConcurrentSkipListMap。

2. 线程池

2.1 使用线程池的好处

| 池化技术相比大家已经屡见不鲜了，线程池、数据库连接池、Http 连接池等等都是对这个思想的应用。池化技术的思想主要是为了减少每次获取资源的消耗，提高对资源的利用率。

线程池提供了一种限制和管理资源（包括执行一个任务）。每个线程池还维护一些基本统计信息，例如已完成任务的数量。

这里借用《Java 并发编程的艺术》提到的来说一下使用线程池的好处：

- **降低资源消耗**。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- **提高响应速度**。当任务到达时，任务可以不需要等到线程创建就能立即执行。
- **提高线程的可管理性**。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

2.2 Executor 框架

2.2.1 简介

Executor 框架是 Java5 之后引进的，在 Java 5 之后，通过 Executor 来启动线程比使用 Thread 的 start 方法更好，除了更易管理，效率更好（用线程池实现，节约开销）外，还有关键的一点：有助于避免 this 逃逸问题。

| 补充：this 逃逸是指在构造函数返回之前其他线程就持有该对象的引用。调用尚未构造完全的对象的方法可能引发令人疑惑的错误。

Executor 框架不仅包括了线程池的管理，还提供了线程工厂、队列以及拒绝策略等，Executor 框架让并发编程变得更加简单。

2.2.2 Executor 框架结构(主要由三大部分组成)

1) 任务(Runnable / Callable)

执行任务需要实现的 `Runnable` 接口或 `Callable` 接口。`Runnable` 接口或 `Callable` 接口实现类都可以被 `ThreadPoolExecutor` 或 `ScheduledThreadPoolExecutor` 执行。

2) 任务的执行(Executor)

如下图所示，包括任务执行机制的核心接口 `Executor`，以及继承自 `Executor` 接口的 `ExecutorService` 接口。`ThreadPoolExecutor` 和 `ScheduledThreadPoolExecutor` 这两个关键类实现了 `ExecutorService` 接口。

这里提了很多底层的类关系，但是，实际上我们需要更多关注的是 `ThreadPoolExecutor` 这个类，这个类在我们实际使用线程池的过程中，使用频率还是非常高的。

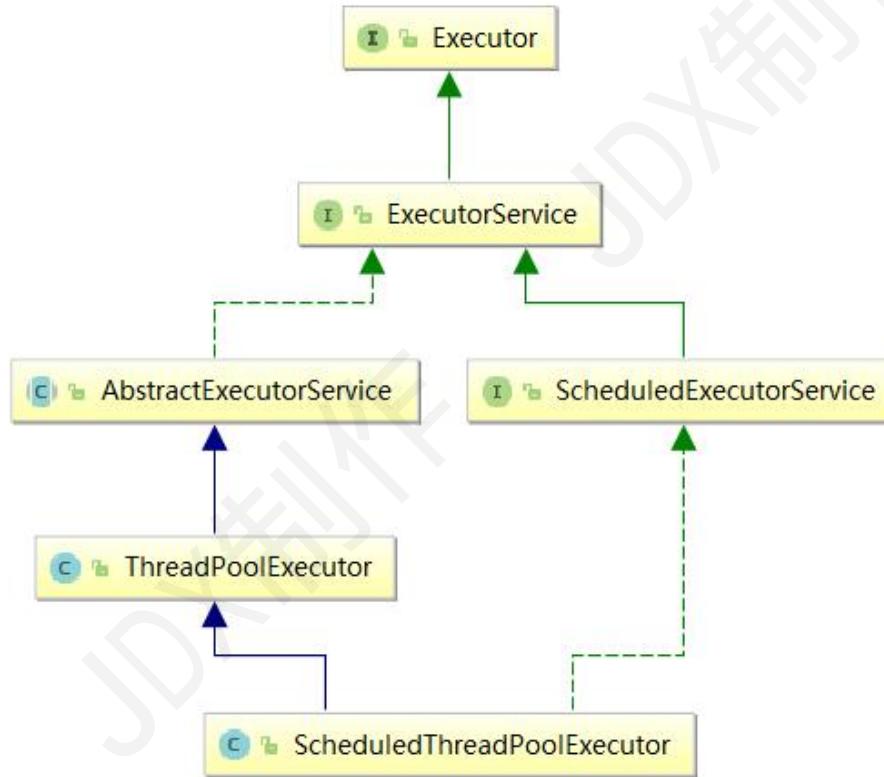
| 注意：通过查看 `ScheduledThreadPoolExecutor` 源代码我们发现 `ScheduledThreadPoolExecutor` 实际上是继承了 `ThreadPoolExecutor` 并实现了 `ScheduledExecutorService`，而 `ScheduledExecutorService` 又实现了 `ExecutorService`，正如我们下面给出的类关系图显示的一样。

`ThreadPoolExecutor` 类描述：

```
//AbstractExecutorService实现了ExecutorService接口
public class ThreadPoolExecutor extends AbstractExecutorService
```

`ScheduledThreadPoolExecutor` 类描述：

```
//ScheduledExecutorService实现了ExecutorService接口
public class ScheduledThreadPoolExecutor
    extends ThreadPoolExecutor
    implements ScheduledExecutorService
```

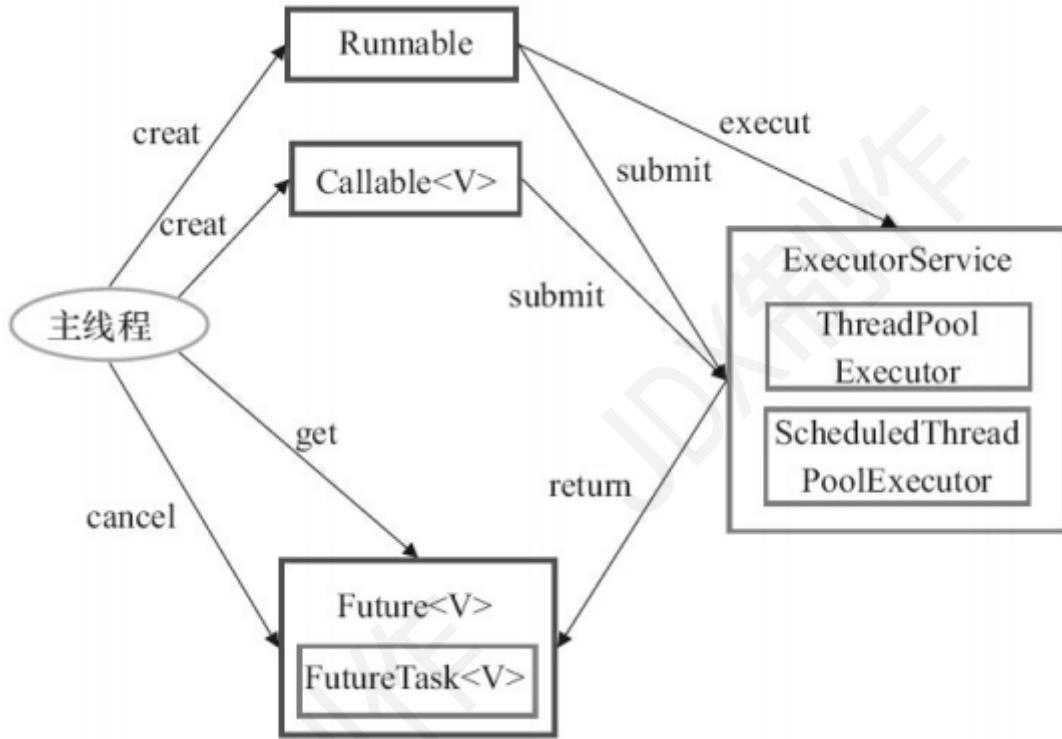


3) 异步计算的结果(Future)

`Future` 接口以及 `Future` 接口的实现类 `FutureTask` 类都可以代表异步计算的结果。

当我们把 `Runnable` 接口 或 `Callable` 接口 的实现类提交给 `ThreadPoolExecutor` 或 `ScheduledThreadPoolExecutor` 执行。 (调用 `submit()` 方法时会返回一个 `FutureTask` 对象)

2.2.3 Executor 框架的使用示意图



1. 主线程首先要创建实现 `Runnable` 或者 `Callable` 接口的任务对象。
2. 把创建完成的实现 `Runnable / Callable` 接口的对象直接交给 `ExecutorService` 执行:
`ExecutorService.execute (Runnable command)` 或者也可以把 `Runnable` 对象或 `Callable` 对象提交给 `ExecutorService` 执行 (`ExecutorService.submit (Runnable task)` 或 `ExecutorService.submit (Callable <T> task)`)。
3. 如果执行 `ExecutorService.submit (...)`，`ExecutorService` 将返回一个实现 `Future` 接口的对象（我们刚刚也提到过了执行 `execute()` 方法和 `submit()` 方法的区别，`submit()` 会返回一个 `FutureTask` 对象）。由于 `FutureTask` 实现了 `Runnable`，我们也可以创建 `FutureTask`，然后直接交给 `ExecutorService` 执行。
4. 最后，主线程可以执行 `FutureTask.get()` 方法来等待任务执行完成。主线程也可以执行 `FutureTask.cancel (boolean mayInterruptIfRunning)` 来取消此任务的执行。

2.3 (重要)ThreadPoolExecutor 类简单介绍

线程池实现类 `ThreadPoolExecutor` 是 `Executor` 框架最核心的类。

2.3.1 ThreadPoolExecutor 类分析

`ThreadPoolExecutor` 类中提供的四个构造方法。我们来看最长的那个，其余三个都是在这个构造方法的基础上产生（其他几个构造方法说白点都是给定某些默认参数的构造方法比如默认制定拒绝策略是什么），这里就不贴代码讲了，比较简单。

```

/**
 * 用给定的初始参数创建一个新的ThreadPoolExecutor。
 */
public ThreadPoolExecutor(int corePoolSize, //线程池的核心线程数量
                          int maximumPoolSize, //线程池的最大线程数
                          long keepAliveTime, //当线程数大于核心线程数时，多余的空
                          TimeUnit unit, //时间单位
                          BlockingQueue<Runnable> workQueue, //任务队列，用来储
                          ThreadFactory threadFactory, //线程工厂，用来创建线程,

```

闲线程存活的最长时间
存等待执行任务的队列
一般默认即可

RejectedExecutionHandler handler // 拒绝策略，当提交的任务过多而不能及时处理时，我们可以定制策略来处理任务

```
        ) {  
            if (corePoolSize < 0 ||  
                maximumPoolSize <= 0 ||  
                maximumPoolSize < corePoolSize ||  
                keepAliveTime < 0)  
                throw new IllegalArgumentException();  
            if (workQueue == null || threadFactory == null || handler == null)  
                throw new NullPointerException();  
            this.corePoolSize = corePoolSize;  
            this.maximumPoolSize = maximumPoolSize;  
            this.workQueue = workQueue;  
            this.keepAliveTime = unit.toNanos(keepAliveTime);  
            this.threadFactory = threadFactory;  
            this.handler = handler;  
        }  
    }
```

下面这些对创建非常重要，在后面使用线程池的过程中你一定会用到！所以，务必拿着小本本记清楚。

ThreadPoolExecutor 3 个最重要的参数：

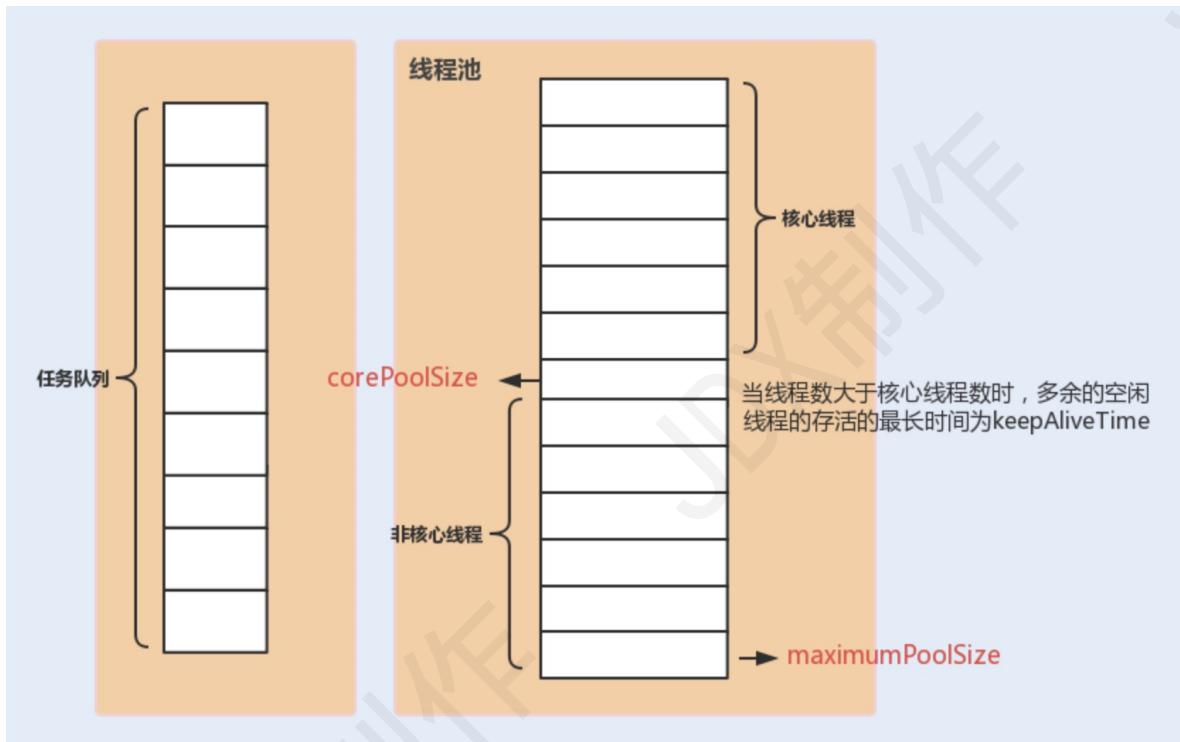
- `corePoolSize`：核心线程数线程数定义了最小可以同时运行的线程数量。
- `maximumPoolSize`：当队列中存放的任务达到队列容量的时候，当前可以同时运行的线程数量变为最大线程数。
- `workQueue`：当新任务来的时候会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，任务就会被存放在队列中。

ThreadPoolExecutor 其他常见参数：

1. `keepAliveTime`：当线程池中的线程数量大于 `corePoolSize` 的时候，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了 `keepAliveTime` 才会被回收销毁；
2. `unit`：`keepAliveTime` 参数的时间单位。
3. `threadFactory`：executor 创建新线程的时候会用到。
4. `handler`：饱和策略。关于饱和策略下面单独介绍一下。

下面这张图可以加深你对线程池中各个参数的相互关系的理解（图片来源：《Java 性能调优实战》）：





ThreadPoolExecutor 饱和策略定义:

如果当前同时运行的线程数量达到最大线程数量并且队列也已经被放满了任时，

ThreadPoolExecutor 定义一些策略：

- `ThreadPoolExecutor.AbortPolicy`：抛出 `RejectedExecutionException` 来拒绝新任务的处理。
- `ThreadPoolExecutor.CallerRunsPolicy`：调用执行自己的线程运行任务，也就是直接在调用 `execute` 方法的线程中运行(`run`)被拒绝的任务，如果执行程序已关闭，则会丢弃该任务。因此这种策略会降低对于新任务提交速度，影响程序的整体性能。另外，这个策略喜欢增加队列容量。如果您的应用程序可以承受此延迟并且你不能任务丢弃任何一个任务请求的话，你可以选择这个策略。
- `ThreadPoolExecutor.DiscardPolicy`：不处理新任务，直接丢弃掉。
- `ThreadPoolExecutor.DiscardOldestPolicy`：此策略将丢弃最早的未处理的任务请求。

举个例子：

Spring 通过 `ThreadPoolTaskExecutor` 或者我们直接通过 `ThreadPoolExecutor` 的构造函数创建线程池的时候，当我们不指定 `RejectedExecutionHandler` 饱和策略的话来配置线程池的时候默认使用的是 `ThreadPoolExecutor.AbortPolicy`。在默认情况下，`ThreadPoolExecutor` 将抛出 `RejectedExecutionException` 来拒绝新来的任务，这代表你将丢失对这个任务的处理。对于可伸缩的应用程序，建议使用 `ThreadPoolExecutor.CallerRunsPolicy`。当最大池被填满时，此策略为我们提供可伸缩队列。（这个直接查看 `ThreadPoolExecutor` 的构造函数源码就可以看出，比较简单的原因，这里就不贴代码了。）

2.3.2 推荐使用 `ThreadPoolExecutor` 构造函数创建线程池

在《阿里巴巴 Java 开发手册》“并发处理”这一章节，明确指出线程资源必须通过线程池提供，不允许在应用中自行显示创建线程。

为什么呢？

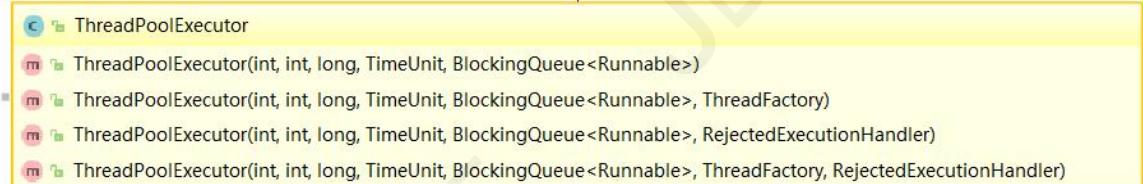
使用线程池的好处是减少在创建和销毁线程上所消耗的时间以及系统资源开销，解决资源不足的问题。如果不使用线程池，有可能会造成系统创建大量同类线程而导致消耗完内存或者“过度切换”的问题。

另外《阿里巴巴 Java 开发手册》中强制线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 构造函数的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险

Executors 返回线程池对象的弊端如下：

- `FixedThreadPool` 和 `SingleThreadExecutor`：允许请求的队列长度为 `Integer.MAX_VALUE`, 可能堆积大量的请求，从而导致 OOM。
- `CachedThreadPool` 和 `ScheduledThreadPool`：允许创建的线程数量为 `Integer.MAX_VALUE`，可能会创建大量线程，从而导致 OOM。

方式一：通过 ThreadPoolExecutor 构造函数实现（推荐）

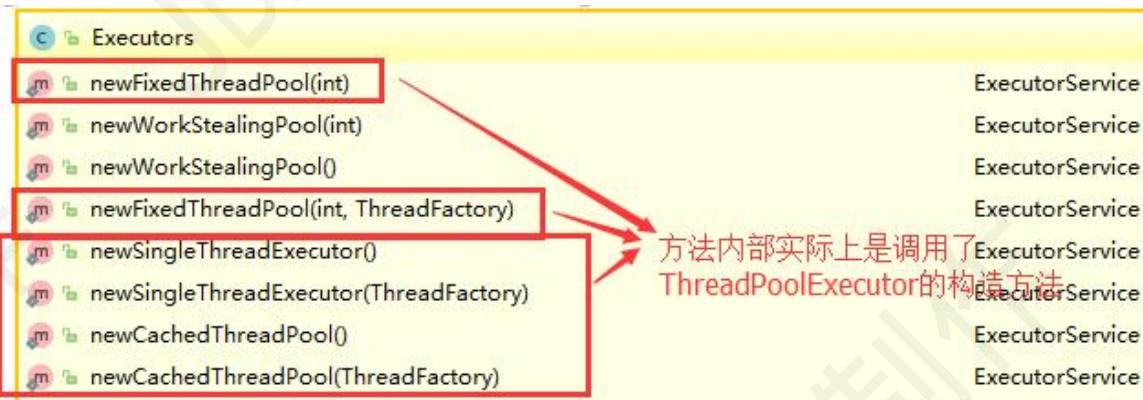


方式二：通过 Executor 框架的工具类 Executors 来实现

我们可以创建三种类型的 ThreadPoolExecutor：

- `FixedThreadPool`
- `SingleThreadExecutor`
- `CachedThreadPool`

对应 Executors 工具类中的方法如图所示：



2.4 (重要)ThreadPoolExecutor 使用示例

我们上面讲解了 `Executor` 框架以及 `ThreadPoolExecutor` 类，下面让我们实战一下，来通过写一个 `ThreadPoolExecutor` 的小 Demo 来回顾上面的内容。

2.4.1 示例代码: Runnable + ThreadPoolExecutor

首先创建一个 `Runnable` 接口的实现类（当然也可以是 `Callable` 接口，我们上面也说了两者的区别。）

`MyRunnable.java`

```
import java.util.Date;

/**
 * 这是一个简单的Runnable类，需要大约5秒钟来执行其任务。
 * @author shuang.kou
 */
public class MyRunnable implements Runnable {
```

```

private String command;

public MyRunnable(String s) {
    this.command = s;
}

@Override
public void run() {
    System.out.println(Thread.currentThread().getName() + " Start. Time = "
+ new Date());
    processCommand();
    System.out.println(Thread.currentThread().getName() + " End. Time = " +
new Date());
}

private void processCommand() {
    try {
        Thread.sleep(5000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

@Override
public String toString() {
    return this.command;
}
}

```

编写测试程序，我们这里以阿里巴巴推荐的使用 `ThreadPoolExecutor` 构造函数自定义参数的方式来创建线程池。

`ThreadPoolExecutorDemo.java`

```

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class ThreadPoolExecutorDemo {

    private static final int CORE_POOL_SIZE = 5;
    private static final int MAX_POOL_SIZE = 10;
    private static final int QUEUE_CAPACITY = 100;
    private static final Long KEEP_ALIVE_TIME = 1L;
    public static void main(String[] args) {

        //使用阿里巴巴推荐的创建线程池的方式
        //通过ThreadPoolExecutor构造函数自定义参数创建
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
            CORE_POOL_SIZE,
            MAX_POOL_SIZE,
            KEEP_ALIVE_TIME,
            TimeUnit.SECONDS,
            new ArrayBlockingQueue<>(QUEUE_CAPACITY),
            new ThreadPoolExecutor.CallerRunsPolicy());
    }
}

```

```

        for (int i = 0; i < 10; i++) {
            //创建workerThread对象 (workerThread类实现了Runnable 接口)
            Runnable worker = new MyRunnable("" + i);
            //执行Runnable
            executor.execute(worker);
        }
        //终止线程池
        executor.shutdown();
        while (!executor.isTerminated()) {
        }
        System.out.println("Finished all threads");
    }
}

```

可以看到我们上面的代码指定了：

1. `corePoolSize`: 核心线程数为 5。
2. `maximumPoolSize` : 最大线程数 10
3. `keepAliveTime` : 等待时间为 1L。
4. `unit`: 等待时间的单位为 `TimeUnit.SECONDS`。
5. `workQueue` : 任务队列为 `ArrayBlockingQueue` , 并且容量为 100;
6. `handler`: 饱和策略为 `CallerRunsPolicy`。

Output:

```

pool-1-thread-3 Start. Time = Sun Apr 12 11:14:37 CST 2020
pool-1-thread-5 Start. Time = Sun Apr 12 11:14:37 CST 2020
pool-1-thread-2 Start. Time = Sun Apr 12 11:14:37 CST 2020
pool-1-thread-1 Start. Time = Sun Apr 12 11:14:37 CST 2020
pool-1-thread-4 Start. Time = Sun Apr 12 11:14:37 CST 2020
pool-1-thread-3 End. Time = Sun Apr 12 11:14:42 CST 2020
pool-1-thread-4 End. Time = Sun Apr 12 11:14:42 CST 2020
pool-1-thread-1 End. Time = Sun Apr 12 11:14:42 CST 2020
pool-1-thread-5 End. Time = Sun Apr 12 11:14:42 CST 2020
pool-1-thread-1 Start. Time = Sun Apr 12 11:14:42 CST 2020
pool-1-thread-2 End. Time = Sun Apr 12 11:14:42 CST 2020
pool-1-thread-5 Start. Time = Sun Apr 12 11:14:42 CST 2020
pool-1-thread-4 Start. Time = Sun Apr 12 11:14:42 CST 2020
pool-1-thread-3 Start. Time = Sun Apr 12 11:14:42 CST 2020
pool-1-thread-2 Start. Time = Sun Apr 12 11:14:42 CST 2020
pool-1-thread-1 End. Time = Sun Apr 12 11:14:47 CST 2020
pool-1-thread-4 End. Time = Sun Apr 12 11:14:47 CST 2020
pool-1-thread-5 End. Time = Sun Apr 12 11:14:47 CST 2020
pool-1-thread-3 End. Time = Sun Apr 12 11:14:47 CST 2020
pool-1-thread-2 End. Time = Sun Apr 12 11:14:47 CST 2020

```

2.4.2 线程池原理分析

承接 4.1 节，我们通过代码输出结果可以看出：线程首先会先执行 5 个任务，然后这些任务有任务被执行完的话，就会去拿新的任务执行。大家可以先通过上面讲解的内容，分析一下到底是咋回事？（自己独立思考一会儿）

现在，我们就分析上面的输出内容来简单分析一下线程池原理。

为了搞懂线程池的原理，我们需要首先分析一下 `execute` 方法。在 4.1 节中的 Demo 中我们使用 `executor.execute(worker)` 来提交一个任务到线程池中去，这个方法非常重要，下面我们来看看它的源码：

```
// 存放线程池的运行状态 (runState) 和线程池内有效线程的数量 (workerCount)
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));

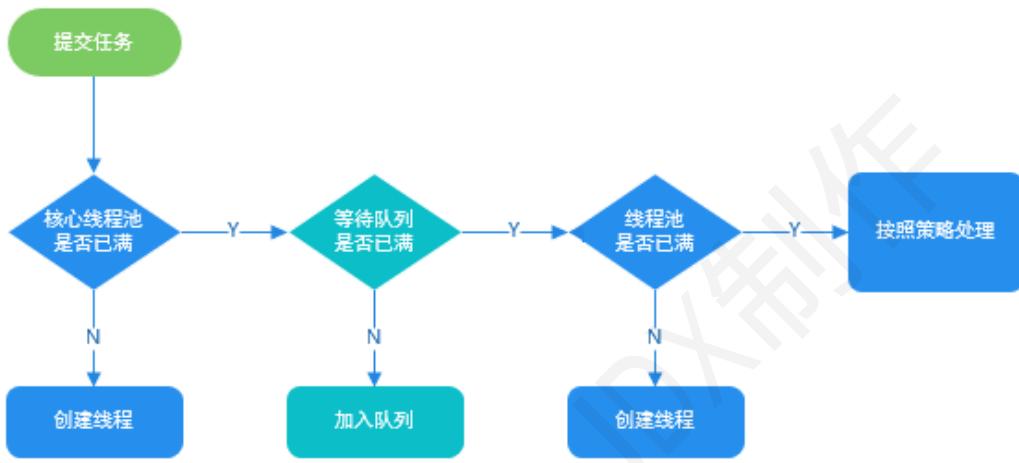
private static int workerCountOf(int c) {
    return c & CAPACITY;
}

//任务队列
private final BlockingQueue<Runnable> workQueue;

public void execute(Runnable command) {
    // 如果任务为null，则抛出异常。
    if (command == null)
        throw new NullPointerException();
    // ctl 中保存的线程池当前的一些状态信息
    int c = ctl.get();

    // 下面会涉及到 3 步 操作
    // 1.首先判断当前线程池中之行的任务数量是否小于 corePoolSize
    // 如果小于的话，通过addWorker(command, true)新建一个线程，并将任务(command)添加到该线程中；然后，启动该线程从而执行任务。
    if (workerCountOf(c) < corePoolSize) {
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    // 2.如果当前之行的任务数量大于等于 corePoolSize 的时候就会走到这里
    // 通过 isRunning 方法判断线程池状态，线程池处于 RUNNING 状态才会被并且队列可以加入任务，该任务才会被加入进去
    if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();
        // 再次获取线程池状态，如果线程池状态不是 RUNNING 状态就需要从任务队列中移除任务，并尝试判断线程是否全部执行完毕。同时执行拒绝策略。
        if (!isRunning(recheck) && remove(command))
            reject(command);
        // 如果当前线程池为空就新创建一个线程并执行。
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    //3. 通过addWorker(command, false)新建一个线程，并将任务(command)添加到该线程中；然后，启动该线程从而执行任务。
    //如果addWorker(command, false)执行失败，则通过reject()执行相应的拒绝策略的内容。
    else if (!addWorker(command, false))
        reject(command);
}
```

通过下图可以更好的对上面这 3 步做一个展示，下图是我为了省事直接从网上找到，原地址不明。



`addworker` 这个方法主要用来创建新的工作线程，如果返回true说明创建和启动工作线程成功，否则的话返回的就是false。

```

// 全局锁，并发操作必备
private final ReentrantLock mainLock = new ReentrantLock();
// 跟踪线程池的最大大小，只有在持有全局锁mainLock的前提下才能访问此集合
private int largestPoolSize;
// 工作线程集合，存放线程池中所有的（活跃的）工作线程，只有在持有全局锁mainLock的前提下才能访问此集合
private final HashSet<Worker> workers = new HashSet<>();
// 获取线程池状态
private static int runStateOf(int c) { return c & ~CAPACITY; }
// 判断线程池的状态是否为 Running
private static boolean isRunning(int c) {
    return c < SHUTDOWN;
}

/**
 * 添加新的工作线程到线程池
 * @param firstTask 要执行
 * @param core 参数为true的话表示使用线程池的基本大小，为false使用线程池最大大小
 * @return 添加成功就返回true否则返回false
 */
private boolean addWorker(Runnable firstTask, boolean core) {
    retry:
    for (;;) {
        //这两句用来获取线程池的状态
        int c = ctl.get();
        int rs = runStateOf(c);

        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN &&
            ! (rs == SHUTDOWN &&
                firstTask == null &&
                ! workQueue.isEmpty())))
            return false;

        for (;;) {
            // 获取线程池中线程的数量
            int wc = workerCountOf(c);

```

```
// core参数为true的话表明队列也满了，线程池大小变为 maximumPoolSize
if (wc >= CAPACITY ||
    wc >= (core ? corePoolSize : maximumPoolSize))
    return false;
//原子操作将workCount的数量加1
if (compareAndIncrementWorkerCount(c))
    break retry;
// 如果线程的状态改变了就再次执行上述操作
c = ctl.get();
if (runStateOf(c) != rs)
    continue retry;
// else CAS failed due to workerCount change; retry inner loop
}
}

// 标记工作线程是否启动成功
boolean workerStarted = false;
// 标记工作线程是否创建成功
boolean workerAdded = false;
Worker w = null;
try {

    w = new Worker(firstTask);
    final Thread t = w.thread;
    if (t != null) {
        // 加锁
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            //获取线程池状态
            int rs = runStateOf(ctl.get());
            //rs < SHUTDOWN 如果线程池状态依然为RUNNING，并且线程的状态是存活的
            //话，就会将工作线程添加到工作线程集合中
            //((rs=SHUTDOWN && firstTask == null)如果线程池状态小于STOP，也就是
            //RUNNING或者SHUTDOWN状态下，同时传入的任务实例firstTask为null，则需要添加到工作线程集合和启
            //动新的worker
            // firstTask == null证明只新建线程而不执行任务
            if (rs < SHUTDOWN ||
                (rs == SHUTDOWN && firstTask == null)) {
                if (t.isAlive()) // precheck that t is startable
                    throw new IllegalThreadStateException();
                workers.add(w);
                //更新当前工作线程的最大容量
                int s = workers.size();
                if (s > largestPoolSize)
                    largestPoolSize = s;
                // 工作线程是否启动成功
                workerAdded = true;
            }
        } finally {
            // 释放锁
            mainLock.unlock();
        }
    }
    ////////////////////////////////////////////////////////////////// 如果成功添加工作线程，则调用worker内部的线程实例t的Thread#start()方
    //法启动真实的线程实例
    if (workerAdded) {
        t.start();
        // 标记线程启动成功
        workerStarted = true;
    }
}
```

```

        }
    }
} finally {
    // 线程启动失败，需要从工作线程中移除对应的worker
    if (!workerStarted)
        addWorkerFailed(w);
}
return workerStarted;
}

```

现在，让我们回到 4.1 节我们写的 Demo，现在应该是很容易就可以搞懂它的原理了呢？

没搞懂的话，也没关系，可以看看我的分析：

我们在代码中模拟了 10 个任务，我们配置的核心线程数为 5、等待队列容量为 100，所以每次只可能存在 5 个任务同时执行，剩下的 5 个任务会被放到等待队列中去。当前的 5 个任务中如果有任务被执行完了，线程池就会去拿新的任务执行。

2.4.3 几个常见的对比

2.4.3.1 Runnable vs Callable

`Runnable` 自 Java 1.0 以来一直存在，但 `Callable` 仅在 Java 1.5 中引入，目的就是为了来处理 `Runnable` 不支持的用例。`Runnable` 接口不会返回结果或抛出检查异常，但是 `Callable` 接口可以。所以，如果任务不需要返回结果或抛出异常推荐使用 `Runnable` 接口，这样代码看起来会更加简洁。

工具类 `Executors` 可以实现 `Runnable` 对象和 `Callable` 对象之间的相互转换。

(`Executors.callable(Runnable task)` 或 `Executors.callable(Runnable task, Object result)`)。

Runnable.java

```

@FunctionalInterface
public interface Runnable {
    /**
     * 被线程执行，没有返回值也无法抛出异常
     */
    public abstract void run();
}

```

Callable.java

```

@FunctionalInterface
public interface Callable<V> {
    /**
     * 计算结果，或在无法这样做时抛出异常。
     * @return 计算得出的结果
     * @throws 如果无法计算结果，则抛出异常
     */
    V call() throws Exception;
}

```

2.4.3.2 execute() vs submit()

1. `execute()` 方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功与否；

2. `submit()` 方法用于提交需要返回值的任务。线程池会返回一个 `Future` 类型的对象，通过这个 `Future` 对象可以判断任务是否执行成功，并且可以通过 `Future` 的 `get()` 方法来获取返回值，`get()` 方法会阻塞当前线程直到任务完成，而使用 `get(long timeout, TimeUnit unit)` 方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

我们以 `AbstractExecutorService` 接口中的一个 `submit` 方法为例子来看看源代码：

```
public Future<?> submit(Runnable task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<Void> ftask = newTaskFor(task, null);
    execute(ftask);
    return ftask;
}
```

上面方法调用的 `newTaskFor` 方法返回了一个 `FutureTask` 对象。

```
protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T value) {
    return new FutureTask<T>(runnable, value);
}
```

我们再来看看 `execute()` 方法：

```
public void execute(Runnable command) {
    ...
}
```

2.4.3.3 `shutdown()` VS `shutdownNow()`

- `shutdown()` :关闭线程池，线程池的状态变为 `SHUTDOWN`。线程池不再接受新任务了，但是队列里的任务得执行完毕。
- `shutdownNow()` :关闭线程池，线程的状态变为 `STOP`。线程池会终止当前正在运行的任务，并停止处理排队的任务并返回正在等待执行的 List。

2.4.3.2 `isTerminated()` VS `isShutdown()`

- `isShutdown` 当调用 `shutdown()` 方法后返回为 true。
- `isTerminated` 当调用 `shutdown()` 方法后，并且所有提交的任务完成后返回为 true

2.4.4 加餐: `Callable + ThreadPoolExecutor` 示例代码

`MyCallable.java`

```
import java.util.concurrent.Callable;

public class Mycallable implements Callable<String> {
    @Override
    public String call() throws Exception {
        Thread.sleep(1000);
        //返回执行当前 Callable 的线程名字
        return Thread.currentThread().getName();
    }
}
```

`CallableDemo.java`

```

import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class CallableDemo {

    private static final int CORE_POOL_SIZE = 5;
    private static final int MAX_POOL_SIZE = 10;
    private static final int QUEUE_CAPACITY = 100;
    private static final Long KEEP_ALIVE_TIME = 1L;

    public static void main(String[] args) {

        //使用阿里巴巴推荐的创建线程池的方式
        //通过ThreadPoolExecutor构造函数自定义参数创建
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
            CORE_POOL_SIZE,
            MAX_POOL_SIZE,
            KEEP_ALIVE_TIME,
            TimeUnit.SECONDS,
            new ArrayBlockingQueue<>(QUEUE_CAPACITY),
            new ThreadPoolExecutor.CallerRunsPolicy());

        List<Future<String>> futureList = new ArrayList<>();
        Callable<String> callable = new MyCallable();
        for (int i = 0; i < 10; i++) {
            //提交任务到线程池
            Future<String> future = executor.submit(callable);
            //将返回值 future 添加到 list, 我们可以通过 future 获得 执行 callable 得到
            的返回值
            futureList.add(future);
        }
        for (Future<String> fut : futureList) {
            try {
                System.out.println(new Date() + ":" + fut.get());
            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
            }
        }
        //关闭线程池
        executor.shutdown();
    }
}

```

Output:

```
wed Nov 13 13:40:41 CST 2019::pool-1-thread-1
wed Nov 13 13:40:42 CST 2019::pool-1-thread-2
wed Nov 13 13:40:42 CST 2019::pool-1-thread-3
wed Nov 13 13:40:42 CST 2019::pool-1-thread-4
wed Nov 13 13:40:42 CST 2019::pool-1-thread-5
wed Nov 13 13:40:42 CST 2019::pool-1-thread-3
wed Nov 13 13:40:43 CST 2019::pool-1-thread-2
Wed Nov 13 13:40:43 CST 2019::pool-1-thread-1
wed Nov 13 13:40:43 CST 2019::pool-1-thread-4
wed Nov 13 13:40:43 CST 2019::pool-1-thread-5
```

2.5 几种常见的线程池详解

2.5.1 FixedThreadPool

2.5.1.1 介绍

`FixedThreadPool` 被称为可重用固定线程数的线程池。通过 `Executors` 类中的相关源代码来看一下相关实现：

```
/**
 * 创建一个可重用固定数量线程的线程池
 */
public static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory
threadFactory) {
    return new ThreadPoolExecutor(nThreads, nThreads,
        0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>(),
        threadFactory);
}
```

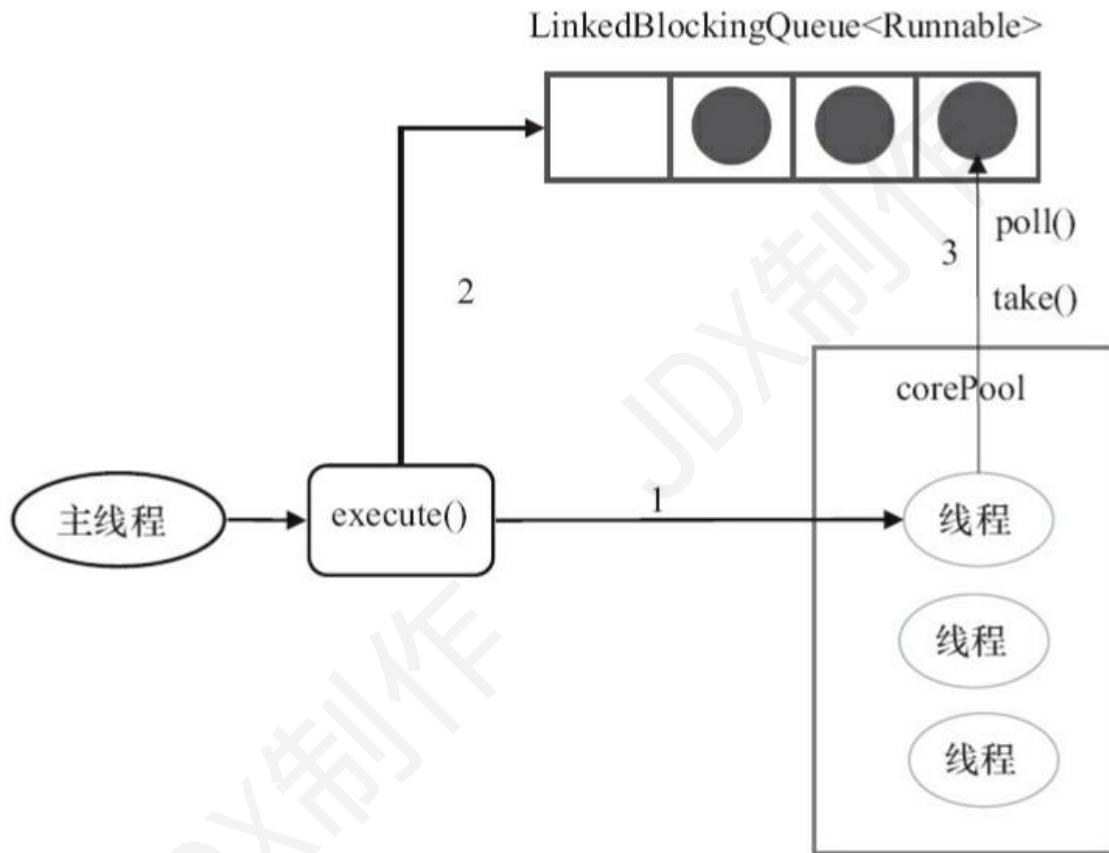
另外还有一个 `FixedThreadPool` 的实现方法，和上面的类似，所以这里不多做阐述：

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
        0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>());
}
```

从上面源代码可以看出新创建的 `FixedThreadPool` 的 `corePoolSize` 和 `maximumPoolsize` 都被设置为 `nThreads`，这个 `nThreads` 参数是我们使用的时候自己传递的。

2.5.1.2 执行任务过程介绍

`FixedThreadPool` 的 `execute()` 方法运行示意图（该图片来源：《Java 并发编程的艺术》）：



上图说明：

1. 如果当前运行的线程数小于 `corePoolSize`, 如果再来新任务的话, 就创建新的线程来执行任务;
2. 当前运行的线程数等于 `corePoolSize` 后, 如果再来新任务的话, 会将任务加入 `LinkedBlockingQueue`;
3. 线程池中的线程执行完手头的任务后, 会在循环中反复从 `LinkedBlockingQueue` 中获取任务来执行;

2.5.1.3 为什么不推荐使用 `FixedThreadPool` ?

`FixedThreadPool` 使用无界队列 `LinkedBlockingQueue` (队列的容量为 `Integer.MAX_VALUE`) 作为线程池的工作队列会对线程池带来如下影响：

1. 当线程池中的线程数达到 `corePoolSize` 后, 新任务将在无界队列中等待, 因此线程池中的线程数不会超过 `corePoolSize`;
2. 由于使用无界队列时 `maximumPoolSize` 将是一个无效参数, 因为不可能存在任务队列满的情况。所以, 通过创建 `FixedThreadPool` 的源码可以看出创建的 `FixedThreadPool` 的 `corePoolSize` 和 `maximumPoolSize` 被设置为同一个值。
3. 由于 1 和 2, 使用无界队列时 `keepAliveTime` 将是一个无效参数;
4. 运行中的 `FixedThreadPool` (未执行 `shutdown()` 或 `shutdownNow()`) 不会拒绝任务, 在任务比较多的时候会导致 OOM (内存溢出)。

2.5.2 SingleThreadExecutor 详解

2.5.2.1 介绍

`singleThreadExecutor` 是只有一个线程的线程池。下面看看 `SingleThreadExecutor` 的实现：

```

    /**
     * 返回只有一个线程的线程池
     */
    public static ExecutorService newSingleThreadExecutor(ThreadFactory
threadFactory) {
        return new FinalizableDelegatedExecutorService
            (new ThreadPoolExecutor(1, 1,
                0L, TimeUnit.MILLISECONDS,
                new LinkedBlockingQueue<Runnable>(),
                threadFactory));
    }
}

```

```

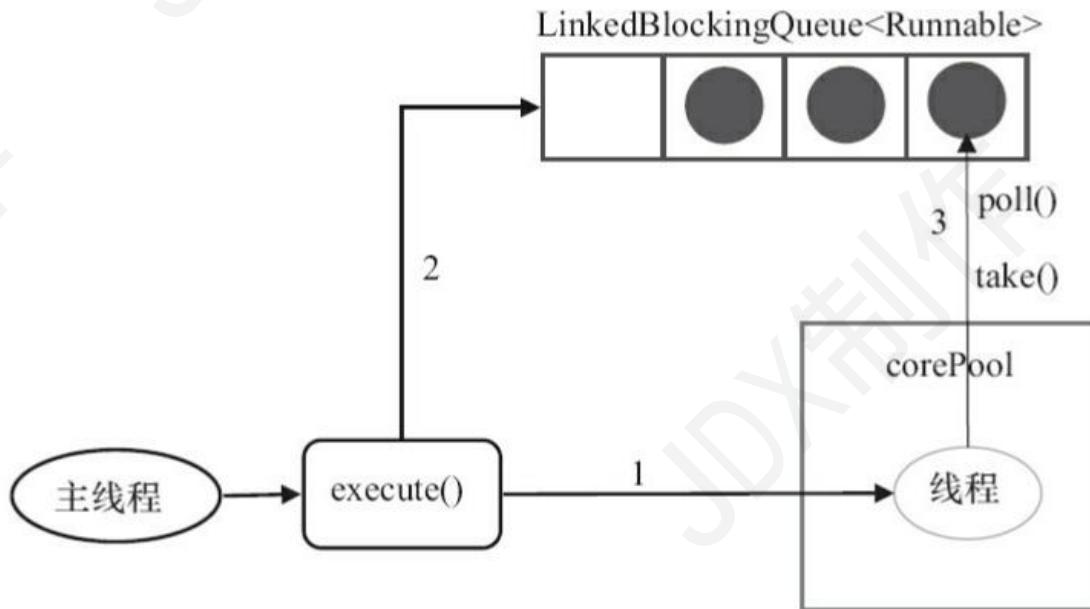
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
            0L, TimeUnit.MILLISECONDS,
            new LinkedBlockingQueue<Runnable>()));
}

```

从上面源代码可以看出新创建的 `SingleThreadExecutor` 的 `corePoolSize` 和 `maximumPoolSize` 都被设置为 1. 其他参数和 `FixedThreadPool` 相同。

2.5.2.2 执行任务过程介绍

`singleThreadExecutor` 的运行示意图 (该图片来源: 《Java 并发编程的艺术》) :



上图说明:

1. 如果当前运行的线程数少于 `corePoolSize`, 则创建一个新的线程执行任务;
2. 当前线程池中有一个运行的线程后, 将任务加入 `LinkedBlockingQueue`
3. 线程执行完当前的任务后, 会在循环中反复从 `LinkedBlockingQueue` 中获取任务来执行;

2.5.2.3 为什么不推荐使用 `SingleThreadExecutor`?

`SingleThreadExecutor` 使用无界队列 `LinkedBlockingQueue` 作为线程池的工作队列 (队列的容量为 `Integer.MAX_VALUE`)。 `SingleThreadExecutor` 使用无界队列作为线程池的工作队列会对线程池带来的影响与 `FixedThreadPool` 相同。说简单点就是可能会导致 OOM,

2.5.3 CachedThreadPool 详解

2.5.3.1 介绍

`CachedThreadPool` 是一个会根据需要创建新线程的线程池。下面通过源码来看看 `CachedThreadPool` 的实现：

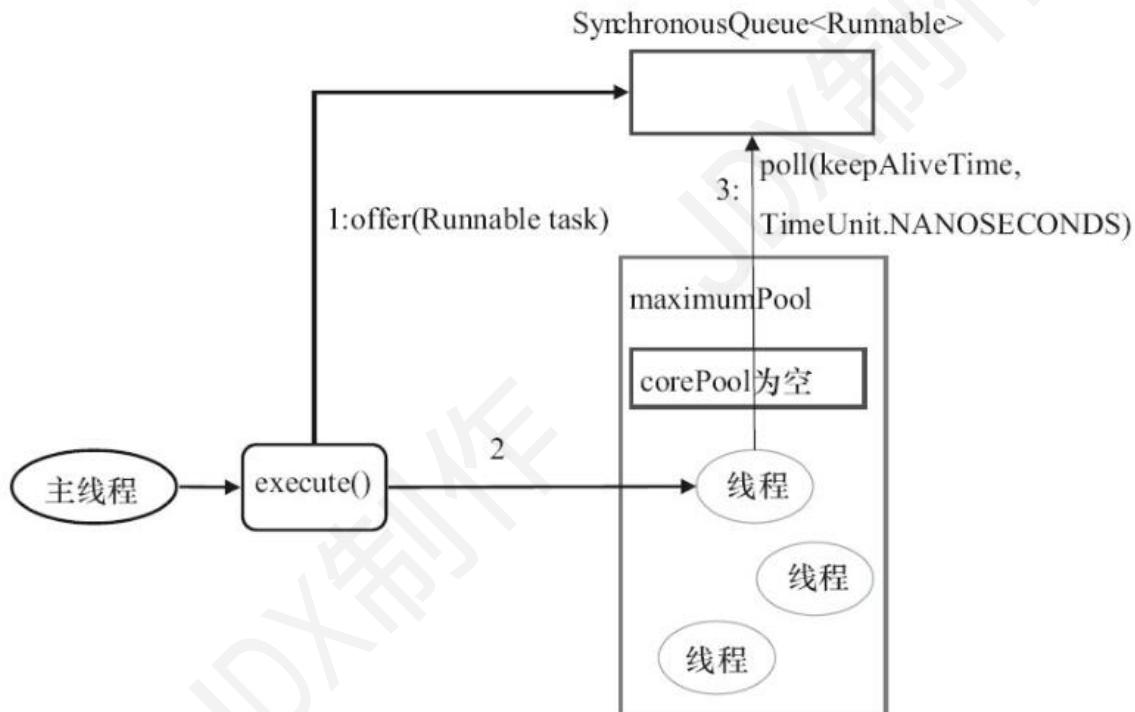
```
/*
 * 创建一个线程池，根据需要创建新线程，但会在先前构建的线程可用时重用它。
 */
public static ExecutorService newCachedThreadPool(ThreadFactory
threadFactory) {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>(),
                                  threadFactory);
}

public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>());
}
```

`CachedThreadPool` 的 `corePoolSize` 被设置为空 (0) , `maximumPoolSize` 被设置为 `Integer.MAX_VALUE`, 即它是无界的, 这也就意味着如果主线程提交任务的速度高于 `maximumPool` 中线程处理任务的速度时, `CachedThreadPool` 会不断创建新的线程。极端情况下, 这样会导致耗尽 cpu 和内存资源。

2.5.3.2 执行任务过程介绍

`CachedThreadPool` 的 `execute()`方法的执行示意图 (该图片来源：《Java 并发编程的艺术》) :



上图说明：

- 首先执行 `SynchronousQueue.offer(Runnable task)` 提交任务到任务队列。如果当前 `maximumPool` 中有闲线程正在执行 `SynchronousQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS)`，那么主线程执行 `offer` 操作与空闲线程执行的 `poll` 操作配对成功，主线程把任务交给空闲线程执行，`execute()` 方法执行完成，否则执行下面的步骤 2；
- 当初始 `maximumPool` 为空，或者 `maximumPool` 中没有空闲线程时，将没有线程执行 `SynchronousQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS)`。这种情况下，步骤 1 将失败，此时 `CachedThreadPool` 会创建新线程执行任务，`execute` 方法执行完成；

2.5.3.3 为什么不推荐使用 `CachedThreadPool`？

`CachedThreadPool` 允许创建的线程数量为 `Integer.MAX_VALUE`，可能会创建大量线程，从而导致 OOM。

2.6 ScheduledThreadPoolExecutor 详解

`ScheduledThreadPoolExecutor` 主要用来在给定的延迟后运行任务，或者定期执行任务。这个在实际项目中基本不会被用到，因为有其他方案选择比如 quartz。大家只需要简单了解一下它的思想。

2.6.1 简介

`ScheduledThreadPoolExecutor` 使用的任务队列 `DelayQueue` 封装了一个 `PriorityQueue`，`PriorityQueue` 会对队列中的任务进行排序，执行所需时间短的放在前面先被执行（`ScheduledFutureTask` 的 `time` 变量小的先执行），如果执行所需时间相同则先提交的任务将被先执行（`ScheduledFutureTask` 的 `squenceNumber` 变量小的先执行）。

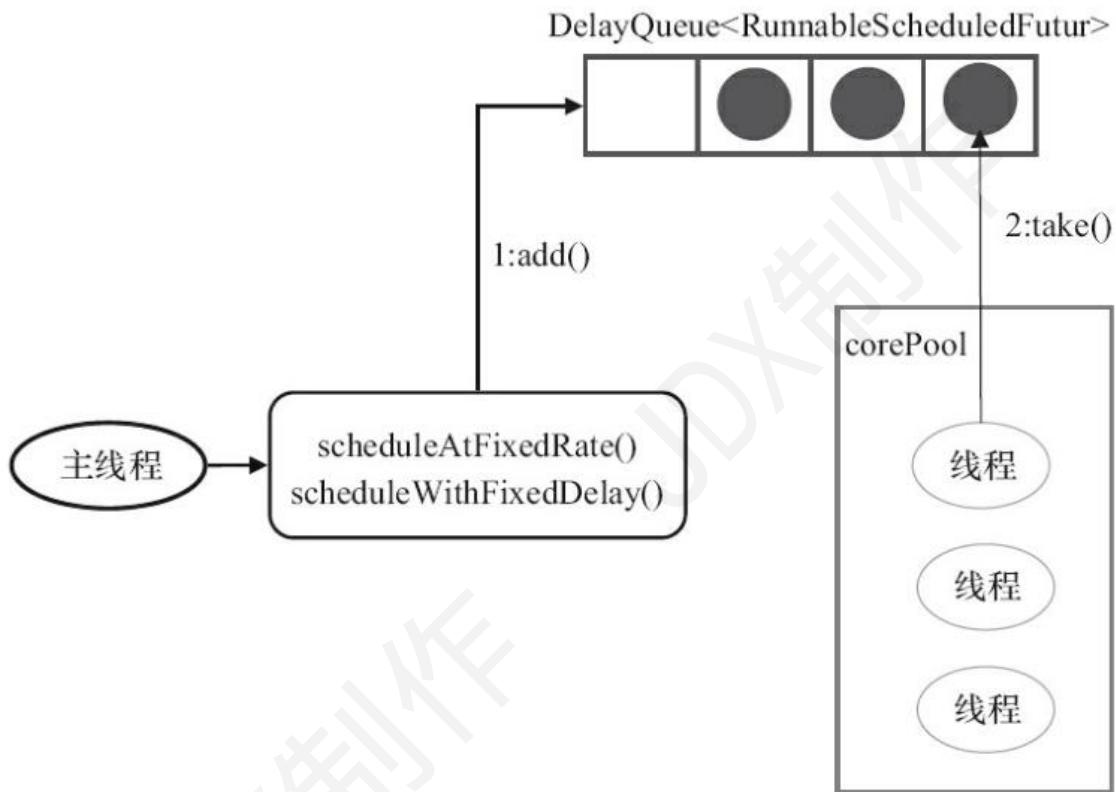
`ScheduledThreadPoolExecutor` 和 `Timer` 的比较：

- `Timer` 对系统时钟的变化敏感，`ScheduledThreadPoolExecutor` 不是；
- `Timer` 只有一个执行线程，因此长时间运行的任务可以延迟其他任务。
`ScheduledThreadPoolExecutor` 可以配置任意数量的线程。此外，如果你想（通过提供 `ThreadFactory`），你可以完全控制创建的线程；
- 在 `TimerTask` 中抛出的运行时异常会杀死一个线程，从而导致 `Timer` 死机：...即计划任务将不再运行。`ScheduledThreadPoolExecutor` 不仅捕获运行时异常，还允许您在需要时处理它们（通过重写 `afterExecute` 方法 `ThreadPoolExecutor`）。抛出异常的任务将被取消，但其他任务将继续运行。

综上，在 JDK1.5 之后，你没有理由再使用 `Timer` 进行任务调度了。

备注： Quartz 是一个由 java 编写的任务调度库，由 OpenSymphony 组织开源出来。在实际项目开发中使用 Quartz 的还是居多，比较推荐使用 Quartz。因为 Quartz 理论上能够同时对上万个任务进行调度，拥有丰富的功能特性，包括任务调度、任务持久化、可集群化、插件等等。

2.6.2 运行机制



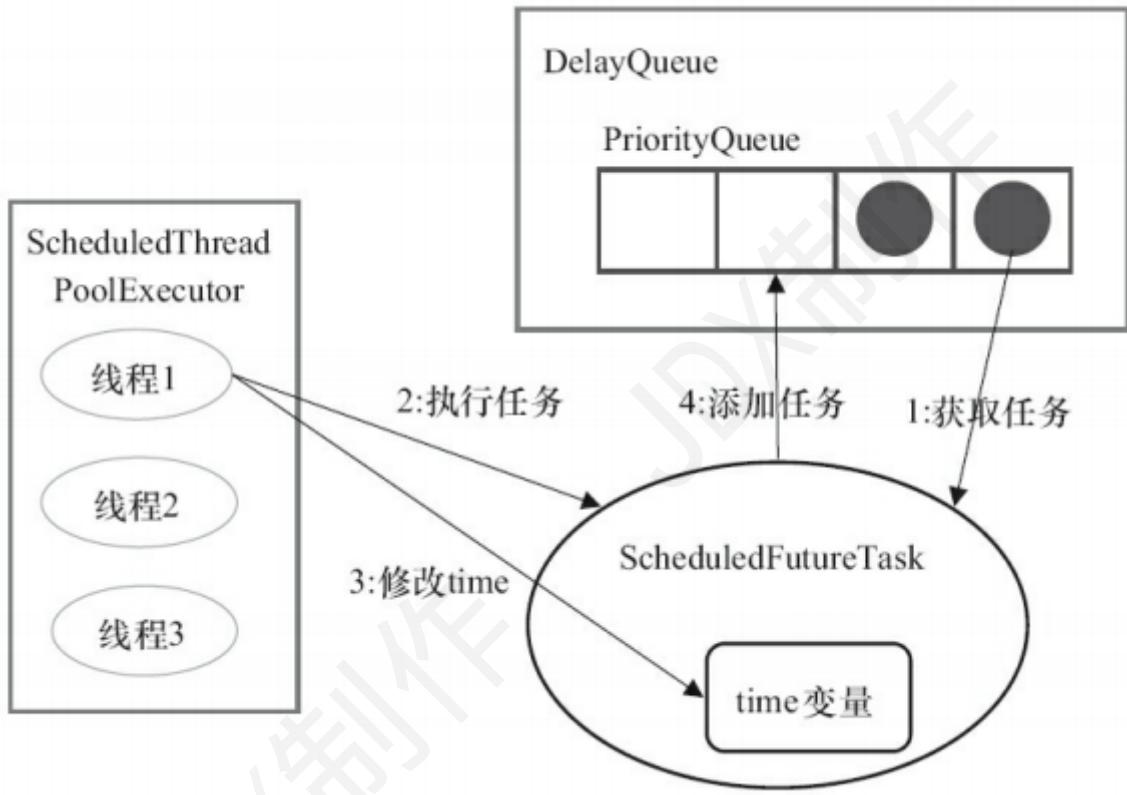
`ScheduledThreadPoolExecutor` 的执行主要分为两大部分：

1. 当调用 `ScheduledThreadPoolExecutor` 的 `scheduleAtFixedRate()` 方法或者 `scheduleWithFixedDelay()` 方法时，会向 `ScheduledThreadPoolExecutor` 的 `DelayQueue` 添加一个实现了 `RunnableScheduledFuture` 接口的 `ScheduledFutureTask`。
2. 线程池中的线程从 `DelayQueue` 中获取 `ScheduledFutureTask`，然后执行任务。

`ScheduledThreadPoolExecutor` 为了实现周期性的执行任务，对 `ThreadPoolExecutor` 做了如下修改：

- 使用 `DelayQueue` 作为任务队列；
- 获取任务的方式不同
- 执行周期任务后，增加了额外的处理

2.6.3 ScheduledThreadPoolExecutor 执行周期任务的步骤



1. 线程 1 从 `DelayQueue` 中获取已到期的 `ScheduledFutureTask (DelayQueue.take())`。到期任务是指 `ScheduledFutureTask` 的 time 大于等于当前系统的时间；
2. 线程 1 执行这个 `ScheduledFutureTask`；
3. 线程 1 修改 `ScheduledFutureTask` 的 time 变量为下次将要被执行的时间；
4. 线程 1 把这个修改 time 之后的 `ScheduledFutureTask` 放回 `DelayQueue` 中 (`DelayQueue.add()`)。

2.7 线程池大小确定

线程池数量的确定一直是困扰着程序员的一个难题，大部分程序员在设定线程池大小的时候就是随心而定。

很多人甚至可能都会觉得把线程池配置过大一点比较好！我觉得这明显是有问题的。就拿我们生活中非常常见的一例子来说：**并不是人多就能把事情做好，增加了沟通交流成本。你本来一件事情只需要 3 个人做，你硬是拉来了 6 个人，会提升做事效率嘛？我想并不会。** 线程数量过多的影响也是和我们分配多少人做事情一样，对于多线程这个场景来说主要是增加了**上下文切换成本**。不清楚什么是上下文切换的话，可以看我下面的介绍。

上下文切换：

多线程编程中一般线程的个数都大于 CPU 核心的个数，而一个 CPU 核心在任意时刻只能被一个线程使用，为了让这些线程都能得到有效执行，CPU 采取的策略是为每个线程分配时间片并轮转的形式。当一个线程的时间片用完的时候就会重新处于就绪状态让给其他线程使用，这个过程就属于一次上下文切换。概括来说就是：当前任务在执行完 CPU 时间片切换到另一个任务之前会先保存自己的状态，以便下次再切换回这个任务时，可以再加载这个任务的状态。**任务从保存到再加载的过程就是一次上下文切换。**

上下文切换通常是计算密集型的。也就是说，它需要相当可观的处理器时间，在每秒几十上百次的切换中，每次切换都需要纳秒量级的时间。所以，上下文切换对系统来说意味着消耗大量的 CPU 时间，事实上，可能是操作系统中时间消耗最大的操作。

Linux 相比与其他操作系统（包括其他类 Unix 系统）有很多的优点，其中有一项就是，其上下文切换和模式切换的时间消耗非常少。

类比于实现世界中的人类通过合作做某件事情，我们可以肯定的一点是线程池大小设置过大或者过小都会有问题，合适的才是最好。

如果我们设置的线程池数量太小的话，如果同一时间有大量任务/请求需要处理，可能会导致大量的请求/任务在任务队列中排队等待执行，甚至会出现任务队列满了之后任务/请求无法处理的情况，或者大量任务堆积在任务队列导致 OOM。这样很明显是有问题的！CPU 根本没有得到充分利用。

但是，如果我们设置线程数量太大，大量线程可能会同时在争取 CPU 资源，这样会导致大量的上下文切换，从而增加线程的执行时间，影响了整体执行效率。

有一个简单并且适用面比较广的公式：

- **CPU 密集型任务(N+1)**：这种任务消耗的主要是 CPU 资源，可以将线程数设置为 N (CPU 核心数) +1，比 CPU 核心数多出来的一个线程是为了防止线程偶发的缺页中断，或者其它原因导致的任务暂停而带来的影响。一旦任务暂停，CPU 就会处于空闲状态，而在这种情况下多出来的一个线程就可以充分利用 CPU 的空闲时间。
- **I/O 密集型任务(2N)**：这种任务应用起来，系统会用大部分的时间来处理 I/O 交互，而线程在处理 I/O 的时间段内不会占用 CPU 来处理，这时就可以将 CPU 交给其它线程使用。因此在 I/O 密集型任务的应用中，我们可以多配置一些线程，具体的计算方法是 2N。

如何判断是 CPU 密集任务还是 IO 密集任务？

CPU 密集型简单理解就是利用 CPU 计算能力的任务比如你在内存中对大量数据进行排序。单凡涉及到网络读取，文件读取这类都是 IO 密集型，这类任务的特点是 CPU 计算耗费时间相比于等待 IO 操作完成的时间来说很少，大部分时间都花在了等待 IO 操作完成上。

3. 乐观锁与悲观锁

3.1 何谓悲观锁与乐观锁

乐观锁对应于生活中乐观的人总是想着事情往好的方向发展，悲观锁对应于生活中悲观的人总是想着事情往坏的方向发展。这两种人各有优缺点，不能不以场景而定说一种人好于另外一种人。

3.1.1 悲观锁

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（**共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程**）。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。Java 中 `synchronized` 和 `ReentrantLock` 等独占锁就是悲观锁思想的实现。

3.1.2 乐观锁

总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和 CAS 算法实现。**乐观锁适用于多读的应用类型，这样可以提高吞吐量**，像数据库提供的类似于 `write_condition` 机制，其实都是提供的乐观锁。在 Java 中 `java.util.concurrent.atomic` 包下面的原子变量类就是使用了乐观锁的一种实现方式 **CAS** 实现的。

3.1.3 两种锁的使用场景

从上面对两种锁的介绍，我们知道两种锁各有优缺点，不可认为一种好于另一种，像**乐观锁适用于写比较少的情况下（多读场景）**，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行 `retry`，这样反倒是降低了性能，所以**一般多写的场景下用悲观锁就比较合适**。

3.2 乐观锁常见的两种实现方式

乐观锁一般会使用版本号机制或CAS算法实现。

3.2.1. 版本号机制

一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。

举一个简单的例子：

假设数据库中帐户信息表中有一个 version 字段，当前值为 1；而当前帐户余额字段（balance）为 \$100。

1. 操作员 A 此时将其读出（version=1），并从其帐户余额中扣除 \$50 (\$100-\$50)。
2. 在操作员 A 操作的过程中，操作员B 也读入此用户信息（version=1），并从其帐户余额中扣除 \$20 (\$100-\$20)。
3. 操作员 A 完成了修改工作，将数据版本号加一（version=2），连同帐户扣除后余额（balance=\$50），提交至数据库更新，此时由于提交数据版本大于数据库记录当前版本，数据被更新，数据库记录 version 更新为 2。
4. 操作员 B 完成了操作，也将版本号加一（version=2）试图向数据库提交数据（balance=\$80），但此时比对数据库记录版本时发现，操作员 B 提交的数据版本号为 2，数据库记录当前版本也为 2，不满足“提交版本必须大于记录当前版本才能执行更新”的乐观锁策略，因此，操作员 B 的提交被驳回。

这样，就避免了操作员 B 用基于 version=1 的旧数据修改的结果覆盖操作员A 的操作结果的可能。

3.2.2. CAS算法

即**compare and swap (比较与交换)**，是一种有名的**无锁算法**。无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）。**CAS算法**涉及到三个操作数

- 需要读写的内存值 V
- 进行比较的值 A
- 拟写入的新值 B

当且仅当 V 的值等于 A 时，CAS 通过原子方式用新值 B 来更新 V 的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个**自旋操作**，即**不断的重试**。

3.3 乐观锁的缺点

ABA问题是乐观锁一个常见的问题

3.3.1 ABA 问题

如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然是A值，那我们就能说明它的值没有被其他线程修改过了吗？很明显是不能的，因为在这段时间它的值可能被改为其他值，然后又改回A，那CAS操作就会误认为它从来没有被修改过。这个问题被称为CAS操作的 "**ABA**" 问题。

JDK 1.5 以后的 `AtomicStampedReference` 类就提供了此种能力，其中的 `compareAndSet` 方法就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。

3.3.2 循环时间长开销大

自旋CAS（也就是不成功就一直循环执行直到成功）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。

3.3.3 只能保证一个共享变量的原子操作

CAS只对单个共享变量有效，当操作涉及跨多个共享变量时CAS无效。但是从JDK 1.5开始，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。所以我们可以使用锁或者利用`AtomicReference`类把多个共享变量合并成一个共享变量来操作。

3.4 CAS与synchronized的使用情景

简单的来说CAS适用于写比较少的情况下（多读场景，冲突一般较少），synchronized适用于写比较多的情况下（多写场景，冲突一般较多）

- 对于资源竞争较少（线程冲突较轻）的情况，使用synchronized同步锁进行线程阻塞和唤醒切换以及用户态内核态间的切换操作额外浪费消耗cpu资源；而CAS基于硬件实现，不需要进入内核，不需要切换线程，操作自旋几率较少，因此可以获得更高的性能。
- 对于资源竞争严重（线程冲突严重）的情况，CAS自旋的概率会比较大，从而浪费更多的CPU资源，效率低于synchronized。

补充：Java并发编程这个领域中synchronized关键字一直都是元老级的角色，很久之前很多人都会称它为“重量级锁”。但是，在JavaSE 1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的**偏向锁**和**轻量级锁**以及其它**各种优化**之后变得在某些情况下并不是那么重了。synchronized的底层实现主要依靠**Lock-Free**的队列，基本思路是**自旋后阻塞，竞争切换后继续竞争锁，稍微牺牲了公平性，但获得了高吞吐量**。在线程冲突较少的情况下，可以获得和CAS类似性能；而线程冲突严重的情况下，性能远高于CAS。

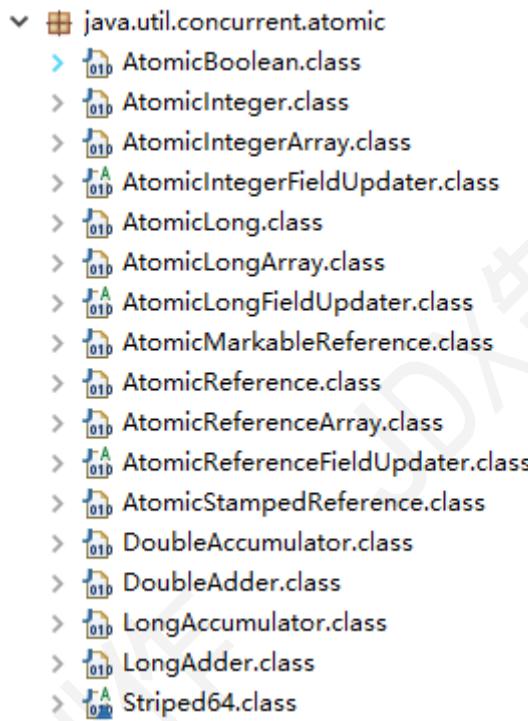
4. Atomic

4.1 Atomic 原子类介绍

Atomic翻译成中文是原子的意思。在化学上，我们知道原子是构成一般物质的最小单位，在化学反应中是不可分割的。在我们这里Atomic是指一个操作是不可中断的。即使是在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程干扰。

所以，所谓原子类说简单点就是具有原子/原子操作特征的类。

并发包`java.util.concurrent`的原子类都存放在`java.util.concurrent.atomic`下，如下图所示。



根据操作的数据类型，可以将JUC包中的原子类分为4类

基本类型

使用原子的方式更新基本类型

- AtomicInteger：整型原子类
- AtomicLong：长整型原子类
- AtomicBoolean：布尔型原子类

数组类型

使用原子的方式更新数组里的某个元素

- AtomicIntegerArray：整型数组原子类
- AtomicLongArray：长整型数组原子类
- AtomicReferenceArray：引用类型数组原子类

引用类型

- AtomicReference：引用类型原子类
- AtomicMarkableReference：原子更新带有标记的引用类型。该类将 boolean 标记与引用关联起来，也可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。
- AtomicStampedReference：原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于解决原子的更新数据和数据的版本号，可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。

对象的属性修改类型

- AtomicIntegerFieldUpdater：原子更新整型字段的更新器
- AtomicLongFieldUpdater：原子更新长整型字段的更新器
- AtomicReferenceFieldUpdater：原子更新引用类型里的字段

修正：AtomicMarkableReference 不能解决ABA问题

```
/**
```

`AtomicMarkableReference`是将一个`boolean`值作是否有更改的标记，本质就是它的版本号只有两个，`true`和`false`，

修改的时候在这两个版本号之间来回切换，这样做并不能解决ABA的问题，只是会降低ABA问题发生的几率而已

```
@author : mazh

@Date : 2020/1/17 14:41
 */

public class SolveABAByAtomicMarkableReference {

    private static AtomicMarkableReference atomicMarkableReference = new
AtomicMarkableReference(100, false);

    public static void main(String[] args) {

        Thread refT1 = new Thread(() -> {
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            atomicMarkableReference.compareAndSet(100, 101,
atomicMarkableReference.isMarked(), !atomicMarkableReference.isMarked());
            atomicMarkableReference.compareAndSet(101, 100,
atomicMarkableReference.isMarked(), !atomicMarkableReference.isMarked());
        });

        Thread refT2 = new Thread(() -> {
            boolean marked = atomicMarkableReference.isMarked();
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            boolean c3 = atomicMarkableReference.compareAndSet(100, 101,
marked, !marked);
            System.out.println(c3); // 返回true,实际应该返回false
        });

        refT1.start();
        refT2.start();
    }
}
```

CAS ABA 问题

- 描述: 第一个线程取到了变量 `x` 的值 `A`, 然后巴拉巴拉干别的事, 总之就是只拿到了变量 `x` 的值 `A`。这段时间内第二个线程也取到了变量 `x` 的值 `A`, 然后把变量 `x` 的值改为 `B`, 然后巴拉巴拉干别的事, 最后又把变量 `x` 的值变为 `A` (相当于还原了)。在这之后第一个线程终于进行了变量 `x` 的操作, 但是此时变量 `x` 的值还是 `A`, 所以 `compareAndSet` 操作是成功。
- 例子描述(可能不太合适, 但好理解): 年初, 现金为零, 然后通过正常劳动赚了三百万, 之后正常消费了(比如买房子)三百万。年末, 虽然现金零收入(可能变成其他形式了), 但是赚了钱是事实, 还是得交税的!
- 代码例子(以 `AtomicInteger` 为例)

```
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicIntegerDefectDemo {
    public static void main(String[] args) {
        defectofABA();
    }

    static void defectofABA() {
        final AtomicInteger atomicInteger = new AtomicInteger(1);

        Thread coreThread = new Thread(
            () -> {
                final int currentValue = atomicInteger.get();
                System.out.println(Thread.currentThread().getName() + " ----
-- currentValue=" + currentValue);

                // 这段目的：模拟处理其他业务花费的时间
                try {
                    Thread.sleep(300);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                boolean casResult = atomicInteger.compareAndSet(1, 2);
                System.out.println(Thread.currentThread().getName()
                    + " ----- currentValue=" + currentValue
                    + ", finalValue=" + atomicInteger.get()
                    + ", compareAndSet Result=" + casResult);
            }
        );
        coreThread.start();

        // 这段目的：为了让 coreThread 线程先跑起来
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        Thread amateurThread = new Thread(
            () -> {
                int currentValue = atomicInteger.get();
                boolean casResult = atomicInteger.compareAndSet(1, 2);
                System.out.println(Thread.currentThread().getName()
                    + " ----- currentValue=" + currentValue
                    + ", finalValue=" + atomicInteger.get()
                    + ", compareAndSet Result=" + casResult);

                currentValue = atomicInteger.get();
                casResult = atomicInteger.compareAndSet(2, 1);
                System.out.println(Thread.currentThread().getName()
                    + " ----- currentValue=" + currentValue
                    + ", finalValue=" + atomicInteger.get()
                    + ", compareAndSet Result=" + casResult);
            }
        );
        amateurThread.start();
    }
}
```

```
    }  
}
```

输出内容如下：

```
Thread-0 ----- currentValue=1  
Thread-1 ----- currentValue=1, finalValue=2, compareAndSet Result=true  
Thread-1 ----- currentValue=2, finalValue=1, compareAndSet Result=true  
Thread-0 ----- currentValue=1, finalValue=2, compareAndSet Result=true
```

下面我们来详细介绍一下这些原子类。

4.2 基本类型原子类

4.2.1 基本类型原子类介绍

使用原子的方式更新基本类型

- AtomicInteger: 整型原子类
- AtomicLong: 长整型原子类
- AtomicBoolean : 布尔型原子类

上面三个类提供的方法几乎相同，所以我们这里以 AtomicInteger 为例子来介绍。

AtomicInteger 类常用方法

```
public final int get() //获取当前的值  
public final int getAndSet(int newValue)//获取当前的值，并设置新的值  
public final int getAndIncrement()//获取当前的值，并自增  
public final int getAndDecrement() //获取当前的值，并自减  
public final int getAndAdd(int delta) //获取当前的值，并加上预期的值  
boolean compareAndSet(int expect, int update) //如果输入的数值等于预期值，则以原子方式  
将该值设置为输入值 (update)  
public final void lazySet(int newValue)//最终设置为newValue,使用 lazySet 设置之后可  
能导致其他线程在之后的一小段时间内还是可以读到旧的值。
```

4.2.2 AtomicInteger 常见方法使用

```
import java.util.concurrent.atomic.AtomicInteger;  
  
public class AtomicIntegerTest {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        int temvalue = 0;  
        AtomicInteger i = new AtomicInteger(0);  
        temvalue = i.getAndSet(3);  
        System.out.println("temvalue:" + temvalue + "; i:" + i); //temvalue:0;  
i:3  
        temvalue = i.getAndIncrement();  
        System.out.println("temvalue:" + temvalue + "; i:" + i); //temvalue:3;  
i:4  
        temvalue = i.getAndAdd(5);  
        System.out.println("temvalue:" + temvalue + "; i:" + i); //temvalue:4;  
i:9  
    }  
}
```

```
}
```

4.2.3 基本数据类型原子类的优势

通过一个简单例子带大家看一下基本数据类型原子类的优势

①多线程环境不使用原子类保证线程安全（基本数据类型）

```
class Test {  
    private volatile int count = 0;  
    //若要线程安全执行执行count++, 需要加锁  
    public synchronized void increment() {  
        count++;  
    }  
  
    public int getCount() {  
        return count;  
    }  
}
```

②多线程环境使用原子类保证线程安全（基本数据类型）

```
class Test2 {  
    private AtomicInteger count = new AtomicInteger();  
  
    public void increment() {  
        count.incrementAndGet();  
    }  
    //使用AtomicInteger之后，不需要加锁，也可以实现线程安全。  
    public int getCount() {  
        return count.get();  
    }  
}
```

4.2.4 AtomicInteger 线程安全原理简单分析

AtomicInteger 类的部分源码：

```
// setup to use Unsafe.compareAndSwapInt for updates (更新操作时提供“比较并替换”的作用)  
private static final Unsafe unsafe = Unsafe.getUnsafe();  
private static final long valueOffset;  
  
static {  
    try {  
        valueOffset = unsafe.objectFieldOffset  
            (AtomicInteger.class.getDeclaredField("value"));  
    } catch (Exception ex) { throw new Error(ex); }  
}  
  
private volatile int value;
```

AtomicInteger 类主要利用 CAS (compare and swap) + volatile 和 native 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升。

CAS的原理是拿期望的值和原本的一个值作比较，如果相同则更新成新的值。Unsafe类的objectFieldOffset()方法是一个本地方法，这个方法是用来拿到“原来的值”的内存地址。另外value是一个volatile变量，在内存中可见，因此JVM可以保证任何时刻任何线程总能拿到该变量的最新值。

4.3 数组类型原子类

4.3.1 数组类型原子类介绍

使用原子的方式更新数组里的某个元素

- AtomicIntegerArray: 整形数组原子类
- AtomicLongArray: 长整形数组原子类
- AtomicReferenceArray : 引用类型数组原子类

上面三个类提供的方法几乎相同，所以我们这里以 AtomicIntegerArray 为例子来介绍。

AtomicIntegerArray 类常用方法

```
public final int get(int i) //获取 index=i 位置元素的值
public final int getAndSet(int i, int newValue)//返回 index=i 位置的当前的值，并将其设置为newValue
public final int getAndIncrement(int i)//获取 index=i 位置元素的值，并让该位置的元素自增
public final int getAndDecrement(int i) //获取 index=i 位置元素的值，并让该位置的元素自减
public final int getAndAdd(int i, int delta) //获取 index=i 位置元素的值，并加上预期的值
boolean compareAndSet(int i, int expect, int update) //如果输入的数值等于预期值，则以原子方式将 index=i 位置的元素值设置为输入值 (update)
public final void lazySet(int i, int newValue)//最终 将index=i 位置的元素设置为 newValue，使用 lazySet 设置之后可能导致其他线程在之后的一小段时间内还是可以读到旧的值。
```

4.3.2 AtomicIntegerArray 常见方法使用

```
import java.util.concurrent.atomic.AtomicIntegerArray;

public class AtomicIntegerArrayTest {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int temvalue = 0;
        int[] nums = { 1, 2, 3, 4, 5, 6 };
        AtomicIntegerArray i = new AtomicIntegerArray(nums);
        for (int j = 0; j < nums.length; j++) {
            System.out.println(i.get(j));
        }
        temvalue = i.getAndSet(0, 2);
        System.out.println("temvalue:" + temvalue + "; i:" + i);
        temvalue = i.getAndIncrement(0);
        System.out.println("temvalue:" + temvalue + "; i:" + i);
        temvalue = i.getAndAdd(0, 5);
        System.out.println("temvalue:" + temvalue + "; i:" + i);
    }
}
```

4.4 引用类型原子类

4.4.1 引用类型原子类介绍

基本类型原子类只能更新一个变量，如果需要原子更新多个变量，需要使用引用类型原子类。

- AtomicReference：引用类型原子类
- AtomicStampedReference：原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于解决原子的更新数据和数据的版本号，可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。
- AtomicMarkableReference：原子更新带有标记的引用类型。该类将 boolean 标记与引用关联起来，也可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。

上面三个类提供的方法几乎相同，所以我们这里以 AtomicReference 为例子来介绍。

4.4.2 AtomicReference 类使用示例

```
import java.util.concurrent.atomic.AtomicReference;

public class AtomicReferenceTest {

    public static void main(String[] args) {
        AtomicReference<Person> ar = new AtomicReference<Person>();
        Person person = new Person("SnailClimb", 22);
        ar.set(person);
        Person updatePerson = new Person("Daisy", 20);
        ar.compareAndSet(person, updatePerson);

        System.out.println(ar.get().getName());
        System.out.println(ar.get().getAge());
    }
}

class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

```
}
```

上述代码首先创建了一个 Person 对象，然后把 Person 对象设置进 AtomicReference 对象中，然后调用 compareAndSet 方法，该方法就是通过 CAS 操作设置 ar。如果 ar 的值为 person 的话，则将其设置为 updatePerson。实现原理与 AtomicInteger 类中的 compareAndSet 方法相同。运行上面的代码后的输出结果如下：

```
Daisy  
20
```

4.4.3 AtomicStampedReference 类使用示例

```
import java.util.concurrent.atomic.AtomicStampedReference;  
  
public class AtomicStampedReferenceDemo {  
    public static void main(String[] args) {  
        // 实例化、取当前值和 stamp 值  
        final Integer initialRef = 0, initialStamp = 0;  
        final AtomicStampedReference<Integer> asr = new AtomicStampedReference<>(initialRef, initialStamp);  
        System.out.println("currentValue=" + asr.getReference() + ",  
currentStamp=" + asr.getStamp());  
  
        // compare and set  
        final Integer newReference = 666, newStamp = 999;  
        final boolean casResult = asr.compareAndSet(initialRef, newReference,  
initialStamp, newStamp);  
        System.out.println("currentValue=" + asr.getReference()  
+ ", currentStamp=" + asr.getStamp()  
+ ", casResult=" + casResult);  
  
        // 获取当前的值和当前的 stamp 值  
        int[] arr = new int[1];  
        final Integer currentValue = asr.get(arr);  
        final int currentStamp = arr[0];  
        System.out.println("currentValue=" + currentValue + ", currentStamp=" +  
currentStamp);  
  
        // 单独设置 stamp 值  
        final boolean attemptStampResult = asr.attemptStamp(newReference, 88);  
        System.out.println("currentValue=" + asr.getReference()  
+ ", currentStamp=" + asr.getStamp()  
+ ", attemptStampResult=" + attemptStampResult);  
  
        // 重新设置当前值和 stamp 值  
        asr.set(initialRef, initialStamp);  
        System.out.println("currentValue=" + asr.getReference() + ",  
currentStamp=" + asr.getStamp());  
  
        // [不推荐使用，除非搞清楚注释的意思了] weak compare and set  
        // 困惑！weakCompareAndSet 这个方法最终还是调用 compareAndSet 方法。[版本：  
jdk-8u191]  
        // 但是注释上写着 "May fail spuriously and does not provide ordering  
guarantees,  
        // so is only rarely an appropriate alternative to compareAndSet."
```

```

    // todo 感觉有可能是 jvm 通过方法名在 native 方法里面做了转发
    final boolean wCasResult = asr.weakCompareAndSet(initialRef,
newReference, initialStamp, newStamp);
    System.out.println("currentValue=" + asr.getReference()
+ ", currentStamp=" + asr.getStamp()
+ ", wCasResult=" + wCasResult);
}
}

```

输出结果如下：

```

currentValue=0, currentStamp=0
currentValue=666, currentStamp=999, casResult=true
currentValue=666, currentStamp=999
currentValue=666, currentStamp=88, attemptStampResult=true
currentValue=0, currentStamp=0
currentValue=666, currentStamp=999, wCasResult=true

```

4.4.4 AtomicMarkableReference 类使用示例

```

import java.util.concurrent.atomic.AtomicMarkableReference;

public class AtomicMarkableReferenceDemo {
    public static void main(String[] args) {
        // 实例化、取当前值和 mark 值
        final Boolean initialRef = null, initialMark = false;
        final AtomicMarkableReference<Boolean> amr = new
AtomicMarkableReference<>(initialRef, initialMark);
        System.out.println("currentValue=" + amr.getReference() + ", "
currentMark=" + amr.isMarked());

        // compare and set
        final Boolean newReference1 = true, newMark1 = true;
        final boolean casResult = amr.compareAndSet(initialRef, newReference1,
initialMark, newMark1);
        System.out.println("currentValue=" + amr.getReference()
+ ", currentMark=" + amr.isMarked()
+ ", casResult=" + casResult);

        // 获取当前的值和当前的 mark 值
        boolean[] arr = new boolean[1];
        final Boolean currentValue = amr.get(arr);
        final boolean currentMark = arr[0];
        System.out.println("currentValue=" + currentValue + ", currentMark=" +
currentMark);

        // 单独设置 mark 值
        final boolean attemptMarkResult = amr.attemptMark(newReference1, false);
        System.out.println("currentValue=" + amr.getReference()
+ ", currentMark=" + amr.isMarked()
+ ", attemptMarkResult=" + attemptMarkResult);

        // 重新设置当前值和 mark 值
        amr.set(initialRef, initialMark);
        System.out.println("currentValue=" + amr.getReference() + ", "
currentMark=" + amr.isMarked());
    }
}

```

```

    // [不推荐使用，除非搞清楚注释的意思了] weak compare and set
    // 困惑! weakCompareAndSet 这个方法最终还是调用 compareAndSet 方法。[版本:
    jdk-8u191]
        // 但是注释上写着 "May fail spuriously and does not provide ordering
        guarantees,
        // so is only rarely an appropriate alternative to compareAndSet."
        // todo 感觉有可能是 jvm 通过方法名在 native 方法里面做了转发
        final boolean wCasResult = amr.weakCompareAndSet(initialRef,
newReference1, initialMark, newMark1);
        System.out.println("currentValue=" + amr.getReference()
+ ", currentMark=" + amr.isMarked()
+ ", wCasResult=" + wCasResult);
    }
}

```

输出结果如下：

```

currentValue=null, currentMark=false
currentValue=true, currentMark=true, casResult=true
currentValue=true, currentMark=true
currentValue=true, currentMark=false, attemptMarkResult=true
currentValue=null, currentMark=false
currentValue=true, currentMark=true, wCasResult=true

```

4.5 对象的属性修改类型原子类

4.5.1 对象的属性修改类型原子类介绍

如果需要原子更新某个类里的某个字段时，需要用到对象的属性修改类型原子类。

- AtomicIntegerFieldUpdater: 原子更新整形字段的更新器
- AtomicLongFieldUpdater: 原子更新长整形字段的更新器
- AtomicReferenceFieldUpdater : 原子更新引用类型里的字段的更新器

要想原子地更新对象的属性需要两步。第一步，因为对象的属性修改类型原子类都是抽象类，所以每次使用都必须使用静态方法 newUpdater() 创建一个更新器，并且需要设置想要更新的类和属性。第二步，更新的对象属性必须使用 public volatile 修饰符。

上面三个类提供的方法几乎相同，所以我们这里以 `AtomicIntegerFieldUpdater` 为例子来介绍。

4.5.2 AtomicIntegerFieldUpdater 类使用示例

```

import java.util.concurrent.atomic.AtomicIntegerFieldUpdater;

public class AtomicIntegerFieldUpdaterTest {
    public static void main(String[] args) {
        AtomicIntegerFieldUpdater<User> a =
        AtomicIntegerFieldUpdater.newUpdater(User.class, "age");

        User user = new User("Java", 22);
        System.out.println(a.getAndIncrement(user)); // 22
        System.out.println(a.get(user)); // 23
    }
}

class User {

```

```
private String name;
public volatile int age;

public User(String name, int age) {
    super();
    this.name = name;
    this.age = age;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

}
```

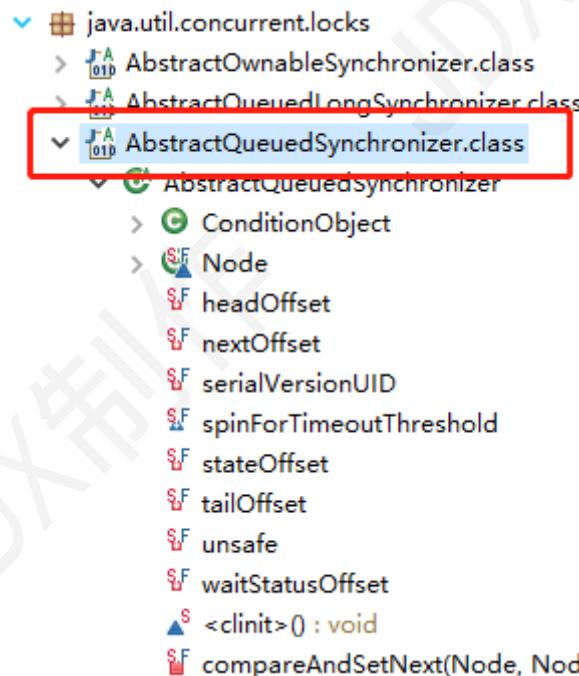
输出结果：

```
22  
23
```

5. AQS

5.1 AQS 简单介绍

AQS 的全称为 (AbstractQueuedSynchronizer) , 这个类在 java.util.concurrent.locks 包下面。



AQS 是一个用来构建锁和同步器的框架，使用 AQS 能简单且高效地构造出应用广泛的大量的同步器，比如我们提到的 ReentrantLock, Semaphore, 其他的诸如 ReentrantReadWriteLock, SynchronousQueue, FutureTask(jdk1.7) 等等皆是基于 AQS 的。当然，我们自己也能利用 AQS 非常轻松容易地构造出符合我们自己需求的同步器。

5.2 AQS 原理

在面试中被问到并发知识的时候，大多都会被问到“请你说一下自己对于 AQS 原理的理解”。下面给大家一个示例供大家参考，面试不是背题，大家一定要加入自己的思想，即使加入不了自己的思想也要保证自己能够通俗的讲出来而不是背出来。

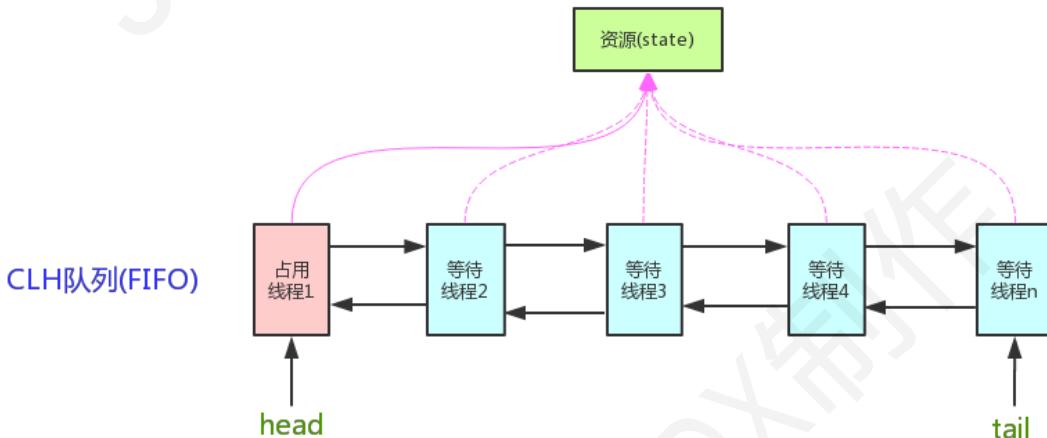
下面大部分内容其实在 AQS 类注释上已经给出了，不过是英语看着比较吃力一点，感兴趣的话可以看看源码。

5.2.1 AQS 原理概览

AQS 核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制 AQS 是用 CLH 队列锁实现的，即将暂时获取不到锁的线程加入到队列中。

CLH(Craig,Landin, and Hagersten)队列是一个虚拟的双向队列（虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系）。AQS 是将每条请求共享资源的线程封装成一个 CLH 锁队列的一个结点（Node）来实现锁的分配。

看个 AQS(AbstractQueuedSynchronizer)原理图：



AQS 使用一个 int 成员变量来表示同步状态，通过内置的 FIFO 队列来完成获取资源线程的排队工作。AQS 使用 CAS 对该同步状态进行原子操作实现对其值的修改。

```
private volatile int state; //共享变量，使用volatile修饰保证线程可见性
```

状态信息通过 protected 类型的 `getState`, `setState`, `compareAndSetState` 进行操作

```

//返回同步状态的当前值
protected final int getState() {
    return state;
}

// 设置同步状态的值
protected final void setState(int newState) {
    state = newState;
}

//原子地(CAS操作)将同步状态值设置为给定值update如果当前同步状态的值等于expect(期望值)
protected final boolean compareAndSetState(int expect, int update) {
    return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
}

```

5.2.2 AQS 对资源的共享方式

AQS 定义两种资源共享方式

1) Exclusive (独占)

只有一个线程能执行，如 ReentrantLock。又可分为公平锁和非公平锁，ReentrantLock 同时支持两种锁，下面以 ReentrantLock 对这两种锁的定义做介绍：

- 公平锁：按照线程在队列中的排队顺序，先到者先拿到锁
- 非公平锁：当线程要获取锁时，先通过两次 CAS 操作去抢锁，如果没抢到，当前线程再加入到队列中等待唤醒。

下面来看 ReentrantLock 中相关的源代码：

ReentrantLock 默认采用非公平锁，因为考虑获得更好的性能，通过 boolean 来决定是否用公平锁（传入 true 用公平锁）。

```

/** Synchronizer providing all implementation mechanics */
private final Sync sync;
public ReentrantLock() {
    // 默认非公平锁
    sync = new NonfairSync();
}
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}

```

ReentrantLock 中公平锁的 lock 方法

```

static final class FairSync extends Sync {
    final void lock() {
        acquire(1);
    }
    // AbstractQueuedSynchronizer.acquire(int arg)
    public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
    protected final boolean tryAcquire(int acquires) {
        final Thread current = Thread.currentThread();
        int c = getState();
        if (c == 0) {

```

```

        // 1. 和非公平锁相比，这里多了一个判断：是否有线程在等待
        if (!hasQueuedPredecessors() &&
            compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
}

```

非公平锁的 lock 方法：

```

static final class NonfairSync extends Sync {
    final void lock() {
        // 2. 和公平锁相比，这里会直接先进行一次CAS，成功就返回了
        if (compareAndSetState(0, 1))
            setExclusiveOwnerThread(Thread.currentThread());
        else
            acquire(1);
    }
    // AbstractQueuedSynchronizer.acquire(int arg)
    public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
    protected final boolean tryAcquire(int acquires) {
        return nonfairTryAcquire(acquires);
    }
}
/**
 * Performs non-fair tryLock. tryAcquire is implemented in
 * subclasses, but both need nonfair try for trylock method.
 */
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        // 这里没有对阻塞队列进行判断
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
    }
}

```

```
        return true;
    }
    return false;
}
```

总结：公平锁和非公平锁只有两处不同：

1. 非公平锁在调用 lock 后，首先就会调用 CAS 进行一次抢锁，如果这个时候恰巧锁没有被占用，那么直接就获取到锁返回了。
2. 非公平锁在 CAS 失败后，和公平锁一样都会进入到 tryAcquire 方法，在 tryAcquire 方法中，如果发现锁这个时候被释放了 (state == 0)，非公平锁会直接 CAS 抢锁，但是公平锁会判断等待队列是否有线程处于等待状态，如果有则不去抢锁，乖乖排到后面。

公平锁和非公平锁就这两点区别，如果这两次 CAS 都不成功，那么后面非公平锁和平公锁是一样的，都要进入到阻塞队列等待唤醒。

相对来说，非公平锁会有更好的性能，因为它的吞吐量比较大。当然，非公平锁让获取锁的时间变得更加不确定，可能会导致在阻塞队列中的线程长期处于饥饿状态。

2) Share (共享)

多个线程可同时执行，如 Semaphore/CountDownLatch。Semaphore、CountDownLatch、CyclicBarrier、ReadWriteLock 我们都会在后面讲到。

ReentrantReadWriteLock 可以看成是组合式，因为 ReentrantReadWriteLock 也就是读写锁允许多个线程同时对某一资源进行读。

不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源 state 的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS 已经在上层已经帮我们实现好了。

5.2.3 AQS 底层使用了模板方法模式

同步器的设计是基于模板方法模式的，如果需要自定义同步器一般的方式是这样（模板方法模式很经典的一个应用）：

1. 使用者继承 AbstractQueuedSynchronizer 并重写指定的方法。（这些重写方法很简单，无非是对于共享资源 state 的获取和释放）
2. 将 AQS 组合在自定义同步组件的实现中，并调用其模板方法，而这些模板方法会调用使用者重写的方法。

这和我们以往通过实现接口的方式有很大区别，这是模板方法模式很经典的一个运用，下面简单的给大家介绍一下模板方法模式，模板方法模式是一个很容易理解的设计模式之一。

模板方法模式是基于“继承”的，主要是为了在不改变模板结构的前提下在子类中重新定义模板中的内容以实现复用代码。举个很简单的例子假如我们要去一个地方的步骤是：购票 `buyTicket()` -> 安检 `securityCheck()` -> 乘坐某某工具回家 `ride()` -> 到达目的地 `arrive()`。我们可能乘坐不同的交通工具回家比如飞机或者火车，所以除了 `ride()` 方法，其他方法的实现几乎相同。我们可以定义一个包含了这些方法的抽象类，然后用户根据自己的需要继承该抽象类然后修改 `ride()` 方法。

AQS 使用了模板方法模式，自定义同步器时需要重写下面几个 AQS 提供的模板方法：

```
isHeldExclusively() //该线程是否正在独占资源。只有用到condition才需要去实现它。  
tryAcquire(int) //独占方式。尝试获取资源，成功则返回true，失败则返回false。  
tryRelease(int) //独占方式。尝试释放资源，成功则返回true，失败则返回false。  
tryAcquireShared(int) //共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；  
正数表示成功，且有剩余资源。  
tryReleaseShared(int) //共享方式。尝试释放资源，成功则返回true，失败则返回false。
```

默认情况下，每个方法都抛出 `UnsupportedOperationException`。这些方法的实现必须是内部线程安全的，并且通常应该简短而不是阻塞。AQS 类中的其他方法都是 final，所以无法被其他类使用，只有这几个方法可以被其他类使用。

以 `ReentrantLock` 为例，state 初始化为 0，表示未锁定状态。A 线程 `lock()` 时，会调用 `tryAcquire()` 独占该锁并将 state+1。此后，其他线程再 `tryAcquire()` 时就会失败，直到 A 线程 `unlock()` 到 state=0（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A 线程自己是可以重复获取此锁的（state 会累加），这就是可重入的概念。但要注意，获取多少次就要释放多少次，这样才能保证 state 是能回到零态的。

再以 `CountDownLatch` 以例，任务分为 N 个子线程去执行，state 也初始化为 N（注意 N 要与线程个数一致）。这 N 个子线程是并行执行的，每个子线程执行完后 `countDown()` 一次，state 会 CAS(Compare and Swap) 减 1。等到所有子线程都执行完后（即 state=0），会 `unpark()` 主调用线程，然后主调用线程就会从 `await()` 函数返回，继续后续动作。

一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现 `tryAcquire`、`tryRelease`、`tryAcquireShared`-`tryReleaseShared` 中的一种即可。但 AQS 也支持自定义同步器同时实现独占和共享两种方式，如 `ReentrantReadWriteLock`。

5.3 Semaphore(信号量)-允许多个线程同时访问

`synchronized` 和 `ReentrantLock` 都是一次只允许一个线程访问某个资源，`Semaphore(信号量)` 可以指定多个线程同时访问某个资源。

示例代码如下：

```
/**  
 *  
 * @author Snailclimb  
 * @date 2018年9月30日  
 * @Description: 需要一次性拿一个许可的情况  
 */  
public class SemaphoreExample1 {  
    // 请求的数量  
    private static final int threadCount = 550;  
  
    public static void main(String[] args) throws InterruptedException {  
        // 创建一个具有固定线程数量的线程池对象（如果这里线程池的线程数量给太少的话你会发现执行的很慢）  
        ExecutorService threadPool = Executors.newFixedThreadPool(300);  
        // 一次只能允许执行的线程数量。  
        final Semaphore semaphore = new Semaphore(20);  
  
        for (int i = 0; i < threadCount; i++) {  
            final int threadnum = i;  
            threadPool.execute(() -> { // Lambda 表达式的运用  
                try {  
                    semaphore.acquire(); // 获取一个许可，所以可运行线程数量为20/1=20
```

```

        test(threadnum);
        semaphore.release(); // 释放一个许可
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

}

threadPool.shutdown();
System.out.println("finish");
}

public static void test(int threadnum) throws InterruptedException {
    Thread.sleep(1000); // 模拟请求的耗时操作
    System.out.println("threadnum:" + threadnum);
    Thread.sleep(1000); // 模拟请求的耗时操作
}
}

```

执行 `acquire` 方法阻塞，直到有一个许可证可以获得然后拿走一个许可证；每个 `release` 方法增加一个许可证，这可能会释放一个阻塞的 `acquire` 方法。然而，其实并没有实际的许可证这个对象，Semaphore 只是维持了一个可获得许可证的数量。Semaphore 经常用于限制获取某种资源的线程数量。

当然一次也可以一次拿取和释放多个许可，不过一般没有必要这样做：

```

semaphore.acquire(5); // 获取5个许可，所以可运行线程数量为20/5=4
test(threadnum);
semaphore.release(5); // 获取5个许可，所以可运行线程数量为20/5=4

```

除了 `acquire` 方法之外，另一个比较常用的是与之对应的方法是 `tryAcquire` 方法，该方法如果获取不到许可就立即返回 `false`。

Semaphore 有两种模式，公平模式和非公平模式。

- **公平模式：** 调用 `acquire` 的顺序就是获取许可证的顺序，遵循 FIFO；
- **非公平模式：** 抢占式的。

Semaphore 对应的两个构造方法如下：

```

public Semaphore(int permits) {
    sync = new NonfairSync(permits);
}

public Semaphore(int permits, boolean fair) {
    sync = fair ? new FairSync(permits) : new NonfairSync(permits);
}

```

这两个构造方法，都必须提供许可的数量，第二个构造方法可以指定是公平模式还是非公平模式，默认非公平模式。

补充：Semaphore与CountDownLatch一样，也是共享锁的一种实现。它默认构造AQS的state为permits。当执行任务的线程数量超出permits,那么多余的线程将会被放入阻塞队列Park,并自旋判断state是否大于0。只有当state大于0的时候，阻塞的线程才能继续执行,此时先前执行任务的线程继续执行release方法，release方法使得state的变量会加1，那么自旋的线程便会判断成功。

如此，每次只有最多不超过permits数量的线程能自旋成功，便限制了执行任务线程的数量。

5.4 CountDownLatch (倒计时器)

CountDownLatch允许 count 个线程阻塞在一个地方，直至所有线程的任务都执行完毕。在 Java 并发中，countdownlatch 的概念是一个常见的面试题，所以一定要确保你很好的理解了它。

CountDownLatch是共享锁的一种实现，它默认构造 AQS 的 state 值为 count。当线程使用countDown方法时，其实使用了 tryReleaseshared 方法以CAS的操作来减少state，直至state为0就代表所有的线程都调用了countDown方法。当调用await方法的时候，如果state不为0，就代表仍然有线程没有调用countDown方法，那么就把已经调用过countDown的线程都放入阻塞队列Park，并自旋CAS判断state == 0，直至最后一个线程调用了countDown，使得state == 0，于是阻塞的线程便判断成功，全部往下执行。

5.4.1 CountDownLatch 的两种典型用法

- 某一线程在开始运行前等待 n 个线程执行完毕。将 CountDownLatch 的计数器初始化为 n：new CountDownLatch(n)，每当一个任务线程执行完毕，就将计数器减 1 countdownLatch.countDown()，当计数器的值变为 0 时，在 CountDownLatch 上 await() 的线程就会被唤醒。一个典型应用场景就是启动一个服务时，主线程需要等待多个组件加载完毕，之后再继续执行。
- 实现多个线程开始执行任务的最大并行性。注意是并行性，不是并发，强调的是多个线程在同一时刻同时开始执行。类似于赛跑，将多个线程放到起点，等待发令枪响，然后同时开跑。做法是初始化一个共享的 CountDownLatch 对象，将其计数器初始化为 1：new CountDownLatch(1)，多个线程在开始执行任务前首先 countdownLatch.await()，当主线程调用 countDown() 时，计数器变为 0，多个线程同时被唤醒。

5.4.2 CountDownLatch 的使用示例

```
/*
 *
 * @author Snailclimb
 * @date 2018年10月1日
 * @Description: CountDownLatch 使用方法示例
 */
public class CountDownLatchExample1 {
    // 请求的数量
    private static final int threadCount = 550;

    public static void main(String[] args) throws InterruptedException {
        // 创建一个具有固定线程数量的线程池对象（如果这里线程池的线程数量给太少的话你会发现执行的很慢）
        ExecutorService threadPool = Executors.newFixedThreadPool(300);
        final CountDownLatch countDownLatch = new CountDownLatch(threadCount);
        for (int i = 0; i < threadCount; i++) {
            final int threadnum = i;
            threadPool.execute(() -> { // Lambda 表达式的运用
                try {
                    test(threadnum);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                } finally {
                    countDownLatch.countDown(); // 表示一个请求已经被完成
                }
            });
        }
        countDownLatch.await(); // 等待所有线程执行完毕
        System.out.println("主线程执行完毕");
    }

    private static void test(int num) {
        System.out.println("线程 " + num + " 执行完毕");
    }
}
```

```

        });
    }
    countDownLatch.await();
    threadPool.shutdown();
    System.out.println("finish");
}

public static void test(int threadnum) throws InterruptedException {
    Thread.sleep(1000); // 模拟请求的耗时操作
    System.out.println("threadnum:" + threadnum);
    Thread.sleep(1000); // 模拟请求的耗时操作
}
}

```

上面的代码中，我们定义了请求的数量为 550，当这 550 个请求被处理完成之后，才会执行 `System.out.println("finish");`。

与 `CountDownLatch` 的第一次交互是主线程等待其他线程。主线程必须在启动其他线程后立即调用 `countDownLatch.await()` 方法。这样主线程的操作就会在这个方法上阻塞，直到其他线程完成各自的任务。

其他 N 个线程必须引用闭锁对象，因为他们需要通知 `CountDownLatch` 对象，他们已经完成了各自的任务。这种通知机制是通过 `CountDownLatch.countDown()` 方法来完成的；每调用一次这个方法，在构造函数中初始化的 `count` 值就减 1。所以当 N 个线程都调用了这个方法，`count` 的值等于 0，然后主线程就能通过 `await()` 方法，恢复执行自己的任务。

再插一嘴：`CountDownLatch` 的 `await()` 方法使用不当很容易产生死锁，比如我们上面代码中的 for 循环改为：

```

for (int i = 0; i < threadCount-1; i++) {
    .....
}

```

这样就导致 `count` 的值没办法等于 0，然后就会导致一直等待。

5.4.3 CountDownLatch 的不足

`CountDownLatch` 是一次性的，计数器的值只能在构造方法中初始化一次，之后没有任何机制再次对其设置值，当 `CountDownLatch` 使用完毕后，它不能再次被使用。

5.4.4 CountDownLatch 相常见面试题

解释一下 `CountDownLatch` 概念？

`CountDownLatch` 和 `CyclicBarrier` 的不同之处？

给出一些 `CountDownLatch` 使用的例子？

`CountDownLatch` 类中主要的方法？

5.5 CyclicBarrier(循环栅栏)

`CyclicBarrier` 和 `CountDownLatch` 非常类似，它也可以实现线程间的技术等待，但是它的功能比 `CountDownLatch` 更加复杂和强大。主要应用场景和 `CountDownLatch` 类似。

CountDownLatch的实现是基于AQS的，而CyclicBarrier是基于 ReentrantLock(ReentrantLock也属于AQS同步器)和 Condition 的。

CyclicBarrier 的字面意思是可循环使用 (Cyclic) 的屏障 (Barrier)。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。CyclicBarrier 默认的构造方法是 `CyclicBarrier(int parties)`，其参数表示屏障拦截的线程数量，每个线程调用 `await` 方法告诉 CyclicBarrier 我已经到达了屏障，然后当前线程被阻塞。

再来看一下它的构造函数：

```
public CyclicBarrier(int parties) {
    this(parties, null);
}

public CyclicBarrier(int parties, Runnable barrierAction) {
    if (parties <= 0) throw new IllegalArgumentException();
    this.parties = parties;
    this.count = parties;
    this.barrierCommand = barrierAction;
}
```

其中，`parties` 就代表了有拦截的线程的数量，当拦截的线程数量达到这个值的时候就打开栅栏，让所有线程通过。

5.5.1 CyclicBarrier 的应用场景

CyclicBarrier 可以用于多线程计算数据，最后合并计算结果的应用场景。比如我们用一个 Excel 保存了用户所有银行流水，每个 Sheet 保存一个帐户近一年的每笔银行流水，现在需要统计用户的日均银行流水，先用多线程处理每个 sheet 里的银行流水，都执行完之后，得到每个 sheet 的日均银行流水，最后，再用 `barrierAction` 用这些线程的计算结果，计算出整个 Excel 的日均银行流水。

5.5.2 CyclicBarrier 的使用示例

示例 1：

```
/**
 *
 * @author Snailclimb
 * @date 2018年10月1日
 * @Description: 测试 CyclicBarrier 类中带参数的 await() 方法
 */
public class CyclicBarrierExample2 {
    // 请求的数量
    private static final int threadCount = 550;
    // 需要同步的线程数量
    private static final CyclicBarrier cyclicBarrier = new CyclicBarrier(5);

    public static void main(String[] args) throws InterruptedException {
        // 创建线程池
        ExecutorService threadPool = Executors.newFixedThreadPool(10);

        for (int i = 0; i < threadCount; i++) {
            final int threadNum = i;
            Thread.sleep(1000);
            threadPool.execute(() -> {
                try {
                    test(threadNum);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                }
            });
        }
    }

    private static void test(int threadNum) {
        System.out.println("线程 " + threadNum + " 到达屏障");
        cyclicBarrier.await();
        System.out.println("线程 " + threadNum + " 穿过屏障");
    }
}
```

```
        e.printStackTrace();
    } catch (BrokenBarrierException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

threadPool.shutdown();
}

public static void test(int threadnum) throws InterruptedException,
BrokenBarrierException {
    System.out.println("threadnum:" + threadnum + "is ready");
    try {
        /**等待60秒，保证子线程完全执行结束*/
        cyclicBarrier.await(60, TimeUnit.SECONDS);
    } catch (Exception e) {
        System.out.println("----CyclicBarrierException----");
    }
    System.out.println("threadnum:" + threadnum + "is finish");
}
}
```

运行结果，如下：

```
threadnum:0is ready
threadnum:1is ready
threadnum:2is ready
threadnum:3is ready
threadnum:4is ready
threadnum:4is finish
threadnum:0is finish
threadnum:1is finish
threadnum:2is finish
threadnum:3is finish
threadnum:5is ready
threadnum:6is ready
threadnum:7is ready
threadnum:8is ready
threadnum:9is ready
threadnum:9is finish
threadnum:5is finish
threadnum:8is finish
threadnum:7is finish
threadnum:6is finish
....
```

可以看到当线程数量也就是请求数量达到我们定义的 5 个的时候，`await` 方法之后的方法才被执行。

另外，`CyclicBarrier` 还提供一个更高级的构造函数 `cyclicBarrier(int parties, Runnable barrierAction)`，用于在线程到达屏障时，优先执行 `barrierAction`，方便处理更复杂的业务场景。示例代码如下：

```
/***
 *
```

```

* @author snailclimb
* @date 2018年10月1日
* @Description: 新建 CyclicBarrier 的时候指定一个 Runnable
*/
public class CyclicBarrierExample3 {
    // 请求的数量
    private static final int threadCount = 550;
    // 需要同步的线程数量
    private static final CyclicBarrier cyclicBarrier = new CyclicBarrier(5, () ->
    {
        System.out.println("-----当线程数达到之后，优先执行-----");
    });

    public static void main(String[] args) throws InterruptedException {
        // 创建线程池
        ExecutorService threadPool = Executors.newFixedThreadPool(10);

        for (int i = 0; i < threadCount; i++) {
            final int threadNum = i;
            Thread.sleep(1000);
            threadPool.execute(() -> {
                try {
                    test(threadNum);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                } catch (BrokenBarrierException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            });
        }
        threadPool.shutdown();
    }

    public static void test(int threadnum) throws InterruptedException,
BrokenBarrierException {
        System.out.println("threadnum:" + threadnum + "is ready");
        cyclicBarrier.await();
        System.out.println("threadnum:" + threadnum + "is finish");
    }
}

```

运行结果，如下：

```

threadnum:0is ready
threadnum:1is ready
threadnum:2is ready
threadnum:3is ready
threadnum:4is ready
-----当线程数达到之后，优先执行-----
threadnum:4is finish
threadnum:0is finish
threadnum:2is finish
threadnum:1is finish
threadnum:3is finish

```

```
threadnum:5is ready
threadnum:6is ready
threadnum:7is ready
threadnum:8is ready
threadnum:9is ready
-----当线程数达到之后，优先执行-----
threadnum:9is finish
threadnum:5is finish
threadnum:6is finish
threadnum:8is finish
threadnum:7is finish
....
```

5.5.3 CyclicBarrier 源码分析

当调用 `cyclicBarrier` 对象调用 `await()` 方法时，实际上调用的是 `dowait(false, 0L)` 方法。`await()` 方法就像建立起一个栅栏的行为一样，将线程挡住了，当拦住的线程数量达到 `parties` 的值时，栅栏才会打开，线程才得以通过执行。

```
public int await() throws InterruptedException, BrokenBarrierException {
    try {
        return dowait(false, 0L);
    } catch (TimeoutException toe) {
        throw new Error(toe); // cannot happen
    }
}
```

`dowait(false, 0L)`:

```
// 当线程数量或者请求数量达到 count 时 await 之后的方法才会被执行。上面的示例中 count
// 的值就为 5。
private int count;
/**
 * Main barrier code, covering the various policies.
 */
private int dowait(boolean timed, long nanos)
    throws InterruptedException, BrokenBarrierException,
           TimeoutException {
    final ReentrantLock lock = this.lock;
    // 锁住
    lock.lock();
    try {
        final Generation g = generation;

        if (g.broken)
            throw new BrokenBarrierException();

        // 如果线程中断了，抛出异常
        if (Thread.interrupted()) {
            breakBarrier();
            throw new InterruptedException();
        }
        // cout减1
        int index = --count;
        // 当 count 数量减为 0 之后说明最后一个线程已经到达栅栏了，也就是达到了可以执行
        await 方法之后的条件
    }
```

```

        if (index == 0) { // tripped
            boolean ranAction = false;
            try {
                final Runnable command = barrierCommand;
                if (command != null)
                    command.run();
                ranAction = true;
                // 将 count 重置为 parties 属性的初始化值
                // 唤醒之前等待的线程
                // 下一波执行开始
                nextGeneration();
                return 0;
            } finally {
                if (!ranAction)
                    breakBarrier();
            }
        }

        // loop until tripped, broken, interrupted, or timed out
        for (;;) {
            try {
                if (!timed)
                    trip.await();
                else if (nanos > 0L)
                    nanos = trip.awaitNanos(nanos);
            } catch (InterruptedException ie) {
                if (g == generation && ! g.broken) {
                    breakBarrier();
                    throw ie;
                } else {
                    // We're about to finish waiting even if we had not
                    // been interrupted, so this interrupt is deemed to
                    // "belong" to subsequent execution.
                    Thread.currentThread().interrupt();
                }
            }
        }

        if (g.broken)
            throw new BrokenBarrierException();

        if (g != generation)
            return index;

        if (timed && nanos <= 0L) {
            breakBarrier();
            throw new TimeoutException();
        }
    }
} finally {
    lock.unlock();
}
}

```

总结：`cyclicBarrier` 内部通过一个 `count` 变量作为计数器，`count` 的初始值为 `parties` 属性的初始化值，每当一个线程到了栅栏这里了，那么就将计数器减一。如果 `count` 值为 0 了，表示这是这一代最后一个线程到达栅栏，就尝试执行我们构造方法中输入的任务。

5.5.4 CyclicBarrier 和 CountDownLatch 的区别

下面这个是国外一个大佬的回答：

CountDownLatch 是计数器，只能使用一次，而 CyclicBarrier 的计数器提供 reset 功能，可以多次使用。但是我不那么认为它们之间的区别仅仅就是这么简单的一点。我们来从 jdk 作者设计的目的来看，javadoc 是这么描述它们的：

CountDownLatch: A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.(CountDownLatch: 一个或者多个线程，等待其他多个线程完成某件事情之后才能执行；)

CyclicBarrier : A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point.(CyclicBarrier : 多个线程互相等待，直到到达同一个同步点，再继续一起执行。)

对于 CountDownLatch 来说，重点是“一个线程（多个线程）等待”，而其他的 N 个线程在完成“某件事情”之后，可以终止，也可以等待。而对于 CyclicBarrier，重点是多个线程，在任意一个线程没有完成，所有的线程都必须等待。

CountDownLatch 是计数器，线程完成一个记录一个，只不过计数不是递增而是递减，而 CyclicBarrier 更像是一个阀门，需要所有线程都到达，阀门才能打开，然后继续执行。

5.6 ReentrantLock 和 ReentrantReadWriteLock

ReentrantLock 和 synchronized 的区别在上面已经讲过了这里就不多做讲解。另外，需要注意的是：读写锁 ReentrantReadWriteLock 可以保证多个线程可以同时读，所以在读操作远大于写操作的时候，读写锁就非常有用了。

(四). JVM

1. Java内存区域

1.1 概述

对于 Java 程序员来说，在虚拟机自动内存管理机制下，不再需要像 C/C++ 程序开发程序员这样为每一个 new 操作去写对应的 delete/free 操作，不容易出现内存泄漏和内存溢出问题。正是因为 Java 程序员把内存控制权利交给 Java 虚拟机，一旦出现内存泄漏和溢出方面的问题，如果不了解虚拟机是怎样使用内存的，那么排查错误将会是一个非常艰巨的任务。

1.2 运行时数据区域

Java 虚拟机在执行 Java 程序的过程中会把它管理的内存划分成若干个不同的数据区域。JDK 1.8 和之前的版本略有不同，下面会介绍到。

JDK 1.8 之前：

小插曲：

更多阿里、腾讯、美团、京东等一线互联网大厂 **Java** 面试真题；包含：基础、并发、锁、**JVM**、设计模式、数据结构、反射/**IO**、数据库、**Redis**、**Spring**、消息队列、分布式、**Zookeeper**、**Dubbo**、**Mybatis**、**Maven**、面经等。

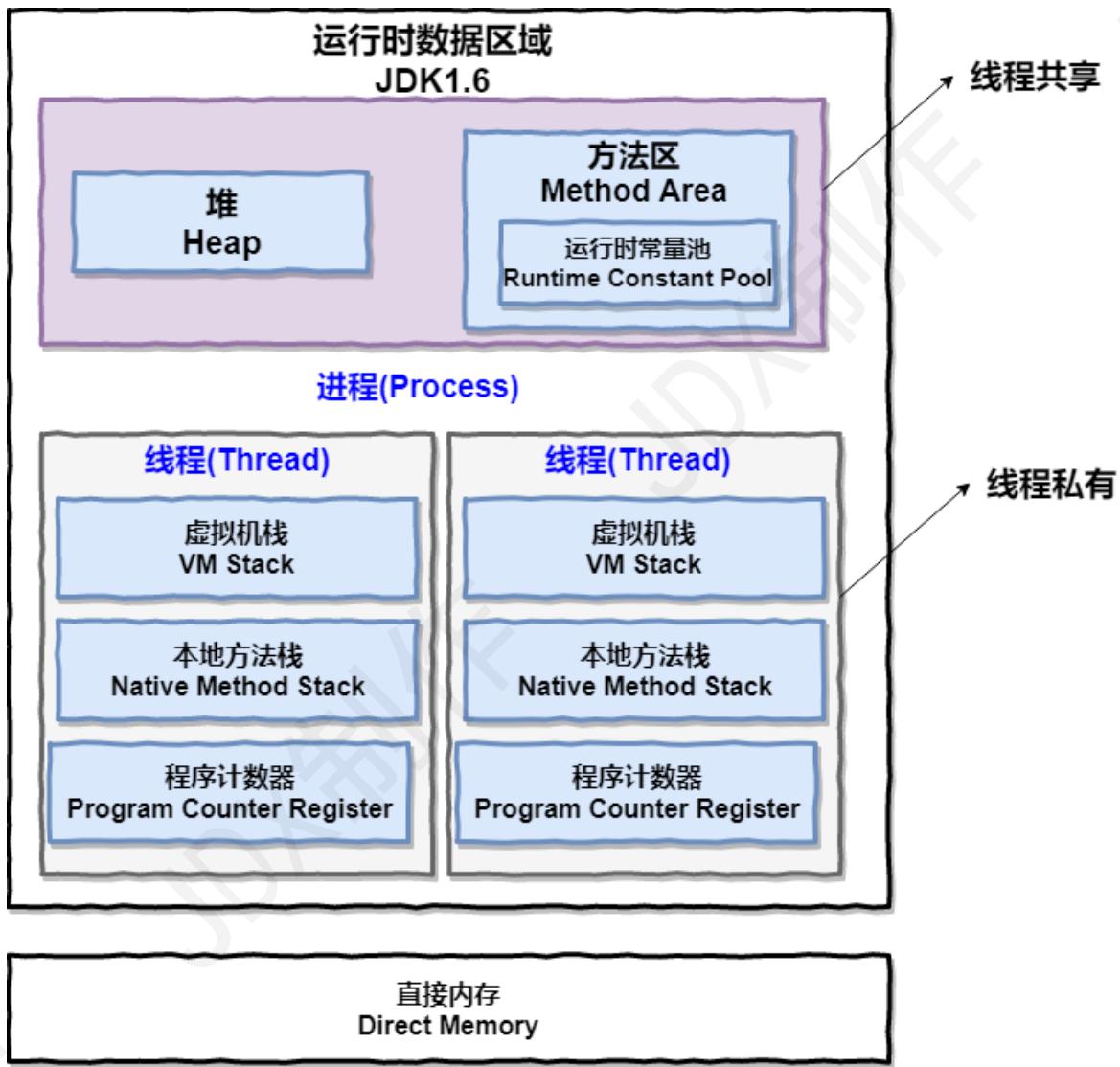
更多**Java** 程序员技术进阶小技巧；例如高效学习（如何学习和阅读代码、面对枯燥和量大的知识）高效沟通（沟通方式及技巧、沟通技术）

更多**Java** 大牛分享的一些职业生涯分享文档

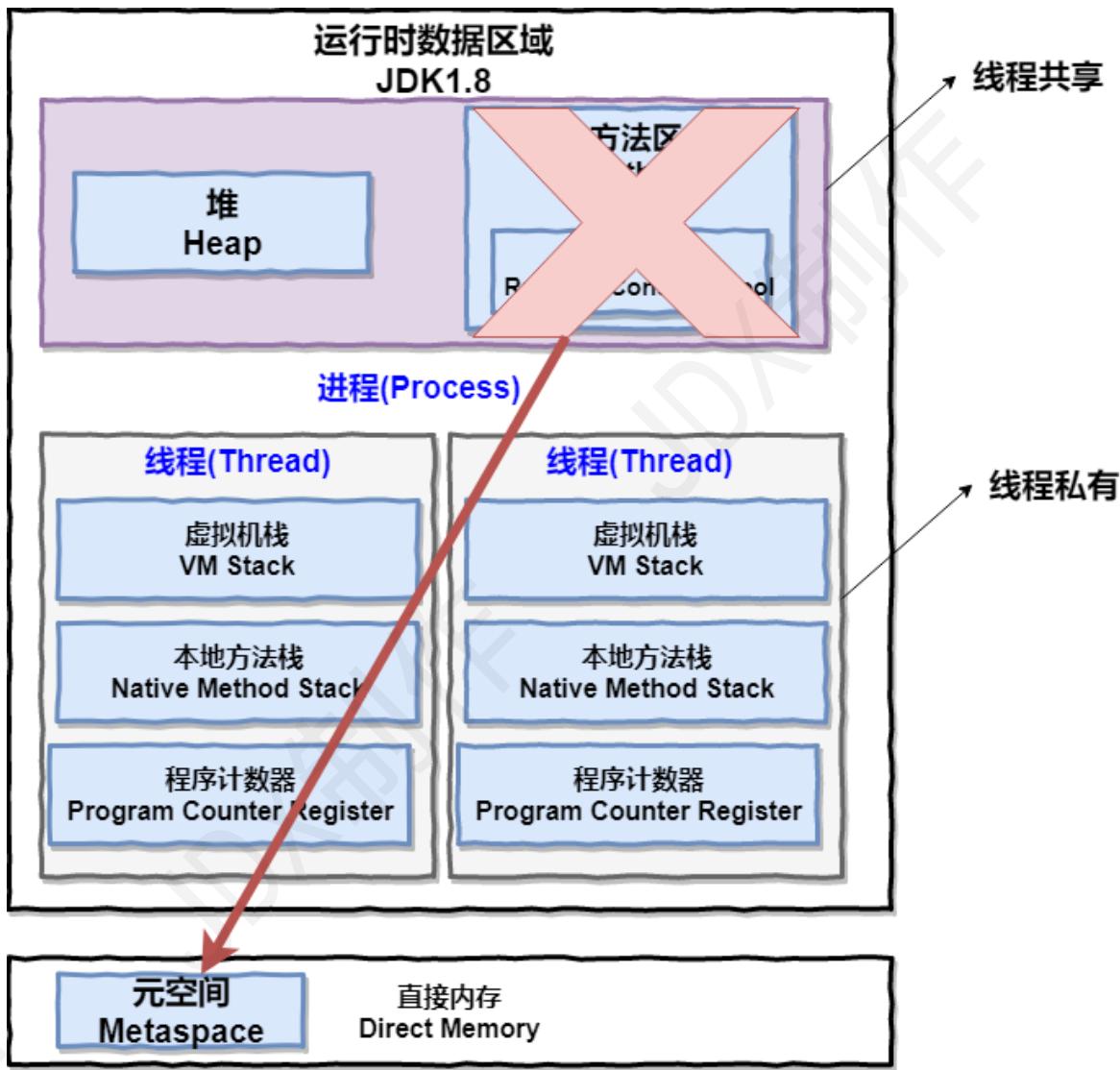
请添加微信：jiang10086a 免费获取

比你优秀的对手在学习，你的仇人在磨刀，你的闺蜜在减肥，隔壁老王在练腰，我们必须不断学习，否则我们将被学习者超越！

趁年轻，使劲拼，给未来的自己一个交代！



JDK 1.8 :



线程私有的：

- 程序计数器
- 虚拟机栈
- 本地方法栈

线程共享的：

- 堆
- 方法区
- 直接内存 (非运行时数据区的一部分)

1.2.1 程序计数器

程序计数器是一块较小的内存空间，可以看作是当前线程所执行的字节码的行号指示器。字节码解释器工作时通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等功能都需要依赖这个计数器来完成。

另外，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

从上面的介绍中我们知道程序计数器主要有两个作用：

1. 字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。
2. 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

注意：程序计数器是唯一一个不会出现 OutOfMemoryError 的内存区域，它的生命周期随着线程的创建而创建，随着线程的结束而死亡。

1.2.2 Java 虚拟机栈

与程序计数器一样，Java 虚拟机栈也是线程私有的，它的生命周期和线程相同，描述的是 Java 方法执行的内存模型，每次方法调用的数据都是通过栈传递的。

Java 内存可以粗略的区分为堆内存（Heap）和栈内存（Stack），其中栈就是现在说的虚拟机栈，或者说是虚拟机栈中局部变量表部分。（实际上，Java 虚拟机栈是由一个个栈帧组成，而每个栈帧中都拥有：局部变量表、操作数栈、动态链接、方法出口信息。）

局部变量表主要存放了编译器可知的各种数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference 类型，它不同于对象本身，可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或其他与此对象相关的位置）。

Java 虚拟机栈会出现两种错误：StackOverflowError 和 OutOfMemoryError。

- **StackOverflowError：**若 Java 虚拟机栈的内存大小不允许动态扩展，那么当线程请求栈的深度超过当前 Java 虚拟机栈的最大深度的时候，就抛出 StackOverflowError 错误。
- **OutOfMemoryError：**若 Java 虚拟机栈的内存大小允许动态扩展，且当线程请求栈时内存用完了，无法再动态扩展了，此时抛出 OutOfMemoryError 错误。

Java 虚拟机栈也是线程私有的，每个线程都有各自的 Java 虚拟机栈，而且随着线程的创建而创建，随着线程的死亡而死亡。

扩展：那么方法/函数如何调用？

Java 栈可用类比数据结构中栈，Java 栈中保存的主要内容是栈帧，每一次函数调用都会有一个对应的栈帧被压入 Java 栈，每一个函数调用结束后，都会有一个栈帧被弹出。

Java 方法有两种返回方式：

1. return 语句。
2. 抛出异常。

不管哪种返回方式都会导致栈帧被弹出。

1.2.3 本地方法栈

和虚拟机栈所发挥的作用非常相似，区别是：**虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。**在 HotSpot 虚拟机中和 Java 虚拟机栈合二为一。

本地方法被执行的时候，在本地方法栈也会创建一个栈帧，用于存放该本地方法的局部变量表、操作数栈、动态链接、出口信息。

方法执行完毕后相应的栈帧也会出栈并释放内存空间，也会出现 StackOverflowError 和 OutOfMemoryError 两种错误。

1.2.4 堆

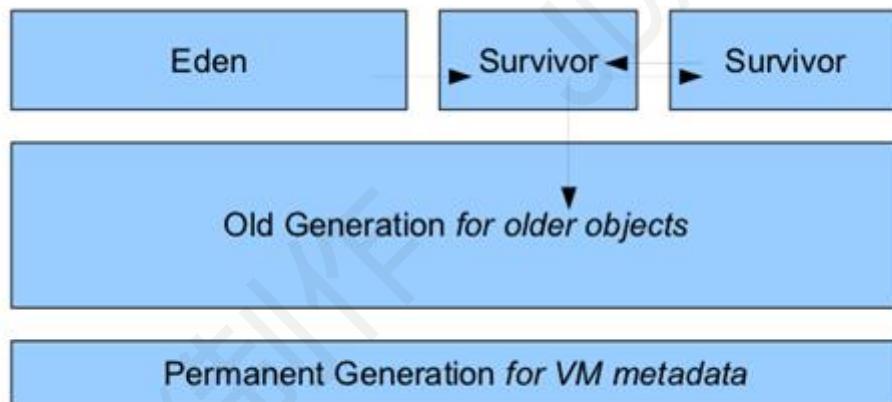
Java 虚拟机所管理的内存中最大的一块，Java 堆是所有线程共享的一块内存区域，在虚拟机启动时创建。**此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例以及数组都在这里分配内存。**

Java 世界中“几乎”所有的对象都在堆中分配，但是，随着 JIT 编译期的发展与逃逸分析技术逐渐成熟，栈上分配、标量替换优化技术将会导致一些微妙的变化，所有的对象都分配到堆上也渐渐变得不那么“绝对”了。从 jdk 1.7 开始已经默认开启逃逸分析，如果某些方法中的对象引用没有被返回或者未被外面使用（也就是未逃逸出去），那么对象可以直接在栈上分配内存。

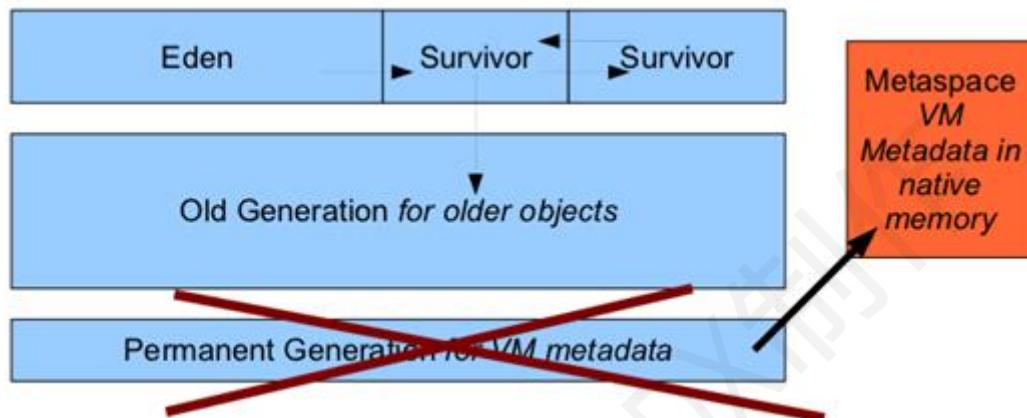
Java 堆是垃圾收集器管理的主要区域，因此也被称作**GC 堆 (Garbage Collected Heap)**。从垃圾回收的角度，由于现在收集器基本都采用分代垃圾收集算法，所以 Java 堆还可以细分为：新生代和老年代；再细致一点有：Eden 空间、From Survivor、To Survivor 空间等。**进一步划分的目的是更好地回收内存，或者更快地分配内存。**

在 JDK 7 版本及 JDK 7 版本之前，堆内存被通常被分为下面三部分：

1. 新生代内存(Young Generation)
2. 老年代(Old Generation)
3. 永生代(Permanent Generation)



JDK 8 版本之后方法区 (HotSpot 的永久代) 被彻底移除了 (JDK1.7 就已经开始了)，取而代之是元空间，元空间使用的是直接内存。



上图所示的 Eden 区、两个 Survivor 区都属于新生代（为了区分，这两个 Survivor 区域按照顺序被命名为 from 和 to），中间一层属于老年代。

大部分情况，对象都会首先在 Eden 区域分配，在一次新生代垃圾回收后，如果对象还存活，则会进入 s0 或者 s1，并且对象的年龄还会加 1(Eden 区->Survivor 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 `-XX:MaxTenuringThreshold` 来设置。

修正：“Hotspot 遍历所有对象时，按照年龄从小到大对之所占用的大小进行累积，当累积的某个年龄大小超过了 survivor 区的一半时，取这个年龄和 MaxTenuringThreshold 中更小的一个值，作为新的晋升年龄阈值”。

动态年龄计算的代码如下

```
uint ageTable::compute_tenuring_threshold(size_t survivor_capacity) {  
    //survivor_capacity是survivor空间的大小
```

```

size_t desired_survivor_size = (size_t)((double)
survivor_capacity)*TargetSurvivorRatio)/100);
size_t total = 0;
uint age = 1;
while (age < table_size) {
total += sizes[age];//sizes数组是每个年龄段对象大小
if (total > desired_survivor_size) break;
age++;
}
uint result = age < MaxTenuringThreshold ? age : MaxTenuringThreshold;
...
}

```

堆这里最容易出现的就是 OutOfMemoryError 错误，并且出现这种错误之后的表现形式还会有几种，比如：

1. `OutOfMemoryError: GC overhead limit exceeded`：当JVM花太多时间执行垃圾回收并且只能回收很少的堆空间时，就会发生此错误。
2. `java.lang.OutOfMemoryError: Java heap space`：假如在创建新的对象时，堆内存中的空间不足以存放新创建的对象，就会引发 `java.lang.OutOfMemoryError: Java heap space` 错误。
(和本机物理内存无关，和你配置的内存大小有关！)
3.

1.2.5 方法区

方法区与 Java 堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 **Non-Heap (非堆)**，目的应该是与 Java 堆区分开来。

方法区也被称为永久代。很多人都会分不清方法区和永久代的关系，为此我也查阅了文献。

1.2.5.1 方法区和永久代的关系

《Java 虚拟机规范》只是规定了有方法区这么个概念和它的作用，并没有规定如何去实现它。那么，在不同的 JVM 上方法区的实现肯定是不同的了。**方法区和永久代的关系很像 Java 中接口和类的关系，类实现了接口，而永久代就是 HotSpot 虚拟机对虚拟机规范中方法区的一种实现方式。**也就是说，永久代是 HotSpot 的概念，方法区是 Java 虚拟机规范中的定义，是一种规范，而永久代是一种实现，一个是标准一个是实现，其他的虚拟机实现并没有永久代这一说法。

1.2.5.2 常用参数

JDK 1.8 之前永久代还没被彻底移除的时候通常通过下面这些参数来调节方法区大小

```

-XX:PermSize=N //方法区（永久代）初始大小
-XX:MaxPermSize=N //方法区（永久代）最大大小，超过这个值将会抛出 OutOfMemoryError 异常：java.lang.OutOfMemoryError: PermGen

```

相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进入方法区后就“永远存在”了。

JDK 1.8 的时候，方法区（HotSpot 的永久代）被彻底移除了（JDK1.7 就已经开始了），取而代之是元空间，元空间使用的是直接内存。

下面是一些常用参数：

```
-XX:MetaspaceSize=N //设置 Metaspace 的初始 (和最小大小)  
-XX:MaxMetaspaceSize=N //设置 Metaspace 的最大大小
```

与永久代很大的不同就是，如果不指定大小的话，随着更多类的创建，虚拟机会耗尽所有可用的系统内存。

1.2.5.3 为什么要将永久代 (PermGen) 替换为元空间 (MetaSpace) 呢？

- 整个永久代有一个 JVM 本身设置固定大小上限，无法进行调整，而元空间使用的是直接内存，受本机可用内存的限制，虽然元空间仍旧可能溢出，但是比原来出现的几率会更小。

当你元空间溢出时会得到如下错误： `java.lang.OutOfMemoryError: MetaSpace`

你可以使用 `-XX: MaxMetaspaceSize` 标志设置最大元空间大小，默认值为 `unlimited`，这意味着它只受系统内存的限制。`-XX: MetaspaceSize` 调整标志定义元空间的初始大小如果未指定此标志，则 Metaspace 将根据运行时的应用程序需求动态地重新调整大小。

- 元空间里面存放的是类的元数据，这样加载多少类的元数据就不由 `MaxPermSize` 控制了，而由系统的实际可用空间来控制，这样能加载的类就更多了。
- 在 JDK8，合并 HotSpot 和 JRockit 的代码时，JRockit 从来没有一个叫永久代的东西，合并之后就没有必要额外的设置这么一个永久代的地方了。

1.2.6 运行时常量池

运行时常量池是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述信息外，还有常量池表（用于存放编译期生成的各种字面量和符号引用）

既然运行时常量池是方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时会抛出 `OutOfMemoryError` 错误。

~~JDK1.7 及之后版本的 JVM 已经将运行时常量池从方法区中移了出来，在 Java 堆 (Heap) 中开辟了一块区域存放运行时常量池。~~

修正：

- JDK1.7之前运行时常量池逻辑包含字符串常量池存放在方法区，此时hotspot虚拟机对方法区的实现为永久代
- JDK1.7 字符串常量池被从方法区拿到了堆中，这里没有提到运行时常量池，也就是说字符串常量池被单独拿到堆，运行时常量池剩下的东西还在方法区，也就是hotspot中的永久代。
- JDK1.8 hotspot移除了永久代用元空间(Metaspace)取而代之，这时候字符串常量池还在堆，运行时常量池还在方法区，只不过方法区的实现从永久代变成了元空间(Metaspace)

1.2.7 直接内存

直接内存并不是虚拟机运行时数据区的一部分，也不是虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用。而且也可能导致 `OutOfMemoryError` 错误出现。

JDK1.4 中新加入的 **NIO(New Input/Output)** 类，引入了一种基于通道（Channel）与缓存区（Buffer）的 I/O 方式，它可以直接使用 Native 函数库直接分配堆外内存，然后通过一个存储在 Java 堆中的 `DirectByteBuffer` 对象作为这块内存的引用进行操作。这样就能在一些场景中显著提高性能，因为避免了在 Java 堆和 Native 堆之间来回复制数据。

本机直接内存的分配不会受到 Java 堆的限制，但是，既然是内存就会受到本机总内存大小以及处理器寻址空间的限制。

1.3 HotSpot 虚拟机对象探秘

通过上面的介绍我们大概知道了虚拟机的内存情况，下面我们来详细的了解一下 HotSpot 虚拟机在 Java 堆中对象分配、布局和访问的全过程。

1.3.1 对象的创建

下图便是 Java 对象的创建过程，我建议最好是能默写出来，并且要掌握每一步在做什么。

Java 创建对象的过程



Step1:类加载检查

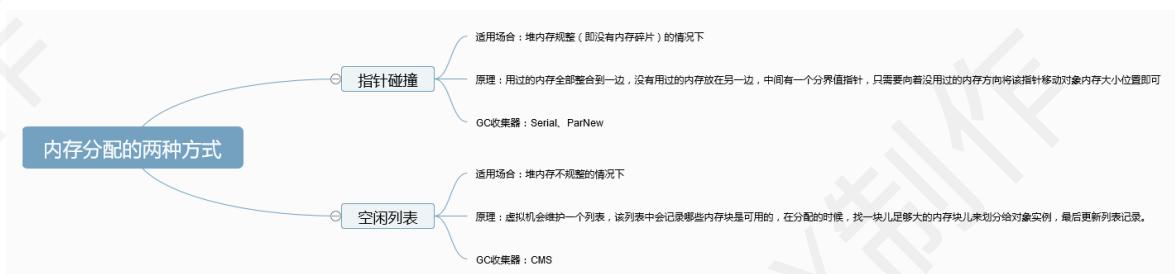
虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到这个类的符号引用，并且检查这个符号引用代表的类是否已被加载过、解析和初始化过。如果没有，那必须先执行相应的类加载过程。

Step2:分配内存

在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需的内存大小在类加载完成后便可确定，为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。分配方式有“指针碰撞”和“空闲列表”两种，选择那种分配方式由 Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

内存分配的两种方式：（补充内容，需要掌握）

选择以上两种方式中的哪一种，取决于 Java 堆内存是否规整。而 Java 堆内存是否规整，取决于 GC 收集器的算法是“标记-清除”，还是“标记-整理”（也称作“标记-压缩”），值得注意的是，复制算法内存也是规整的。



内存分配并发问题（补充内容，需要掌握）

在创建对象的时候有一个很重要的问题，就是线程安全，因为在实际开发过程中，创建对象是很频繁的事情，作为虚拟机来说，必须要保证线程是安全的，通常来讲，虚拟机采用两种方式来保证线程安全：

- **CAS+失败重试：** CAS 是乐观锁的一种实现方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。**虚拟机采用 CAS 配上失败重试的方式保证更新操作的原子性。**
- **TLAB：** 为每一个线程预先在 Eden 区分配一块儿内存，JVM 在给线程中的对象分配内存时，首先在 TLAB 分配，当对象大于 TLAB 中的剩余内存或 TLAB 的内存已用尽时，再采用上述的 CAS 进行内存分配

Step3:初始化零值

内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头），这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

Step4:设置对象头

初始化零值完成之后，**虚拟机要对对象进行必要的设置**，例如这个对象是那个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的 GC 分代年龄等信息。**这些信息存放在对象头中**。另外，根据虚拟机当前运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。

Step5: 执行 init 方法

在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从 Java 程序的视角来看，对象创建才刚刚开始，`<init>` 方法还没有执行，所有的字段都还为零。所以一般来说，执行 new 指令之后会接着执行 `<init>` 方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生出来。

1.3.2 对象的内存布局

在 Hotspot 虚拟机中，对象在内存中的布局可以分为 3 块区域：**对象头、实例数据和对齐填充**。

Hotspot 虚拟机的对象头包括两部分信息，第一部分用于存储对象自身的运行时数据（哈希码、GC 分代年龄、锁状态标志等等），**另一部分是类型指针**，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是那个类的实例。

实例数据部分是对象真正存储的有效信息，也是在程序中所定义的各种类型的字段内容。

对齐填充部分不是必然存在的，也没有什么特别的含义，仅仅起占位作用。因为 Hotspot 虚拟机的自动内存管理系统要求对象起始地址必须是 8 字节的整数倍，换句话说就是对象的大小必须是 8 字节的整数倍。而对象头部分正好是 8 字节的倍数（1 倍或 2 倍），因此，当对象实例数据部分没有对齐时，就需要通过对齐填充来补全。

1.3.3 对象的访问定位

建立对象就是为了使用对象，我们的 Java 程序通过栈上的 reference 数据来操作堆上的具体对象。对象的访问方式由虚拟机实现而定，目前主流的访问方式有①**使用句柄**和②**直接指针**两种：

1. **句柄**：如果使用句柄的话，那么 Java 堆中将会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息；

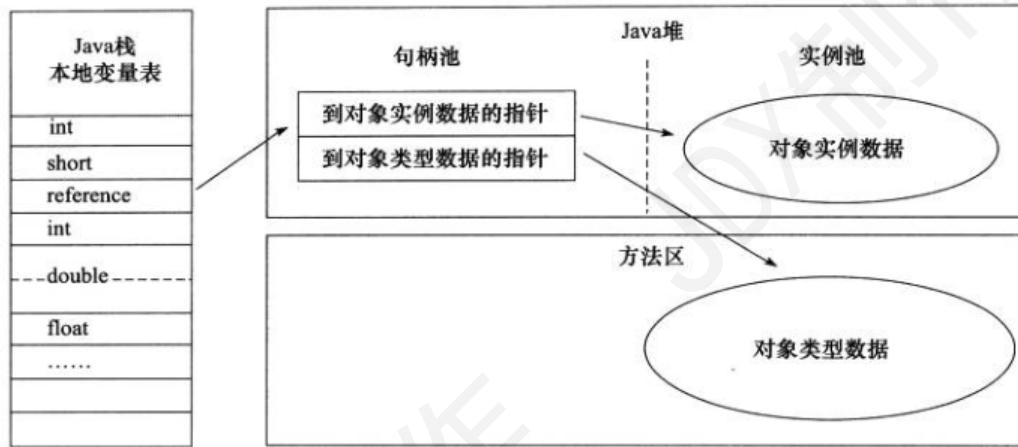


图 2-2 通过句柄访问对象

2. **直接指针**：如果使用直接指针访问，那么 Java 堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，而 reference 中存储的直接就是对象的地址。

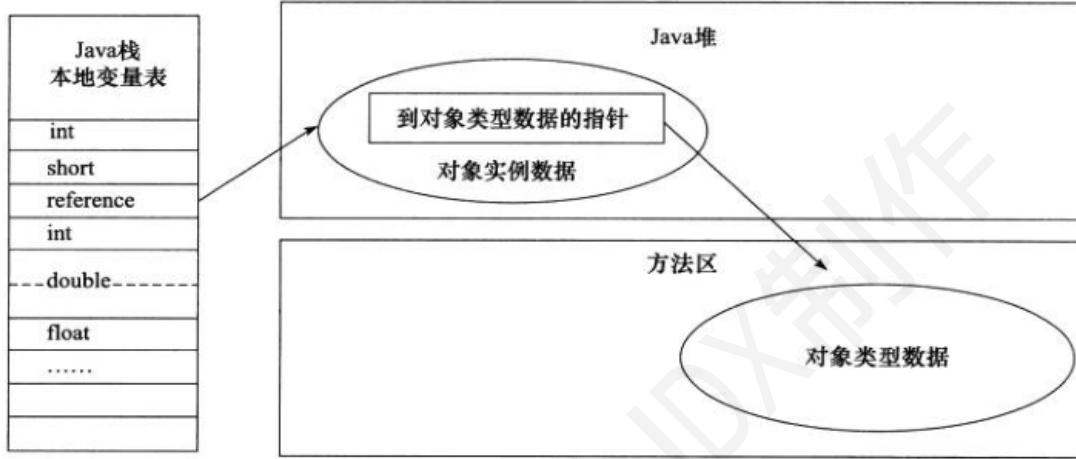


图 2-3 通过直接指针访问对象

这两种对象访问方式各有优势。使用句柄来访问的最大好处是 `reference` 中存储的是稳定的句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而 `reference` 本身不需要修改。使用直接指针访问方式最大的好处就是速度快，它节省了一次指针定位的时间开销。

1.4 重点补充内容

1.4.1 String 类和常量池

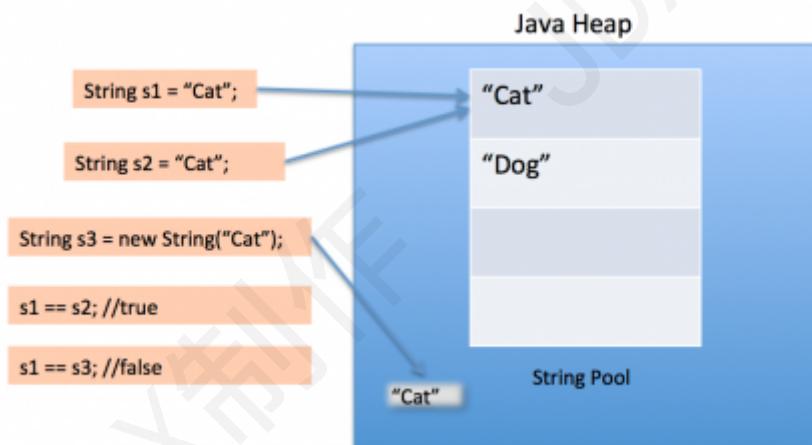
String 对象的两种创建方式：

```
String str1 = "abcd"; //先检查字符串常量池中有没有"abcd"，如果字符串常量池中没有，则创建一个，然后 str1 指向字符串常量池中的对象，如果有，则直接将 str1 指向"abcd"；  
String str2 = new String("abcd"); //堆中创建一个新的对象  
String str3 = new String("abcd"); //堆中创建一个新的对象  
System.out.println(str1==str2); //false  
System.out.println(str2==str3); //false
```

这两种不同的创建方法是有差别的。

- 第一种方式是在常量池中拿对象；
- 第二种方式是直接在堆内存空间创建一个新的对象。

记住一点：只要使用 `new` 方法，便需要创建新的对象。



String 类型的常量池比较特殊。它的主要使用方法有两种：

- 直接使用双引号声明出来的 String 对象会直接存储在常量池中。

- 如果不是用双引号声明的 String 对象，可以使用 String 提供的 intern 方法。String.intern() 是一个 Native 方法，它的作用是：如果运行时常量池中已经包含一个等于此 String 对象内容的字符串，则返回常量池中该字符串的引用；如果没有，JDK1.7之前（不包含1.7）的处理方式是在常量池中创建与此 String 内容相同的字符串，并返回常量池中创建的字符串的引用，JDK1.7以及之后的处理方式是在常量池中记录此字符串的引用，并返回该引用。

```

String s1 = new String("计算机");
String s2 = s1.intern();
String s3 = "计算机";
System.out.println(s2); //计算机
System.out.println(s1 == s2); //false, 因为一个是堆内存中的 String 对象一个
是常量池中的 String 对象,
System.out.println(s3 == s2); //true, 因为两个都是常量池中的 String 对象

```

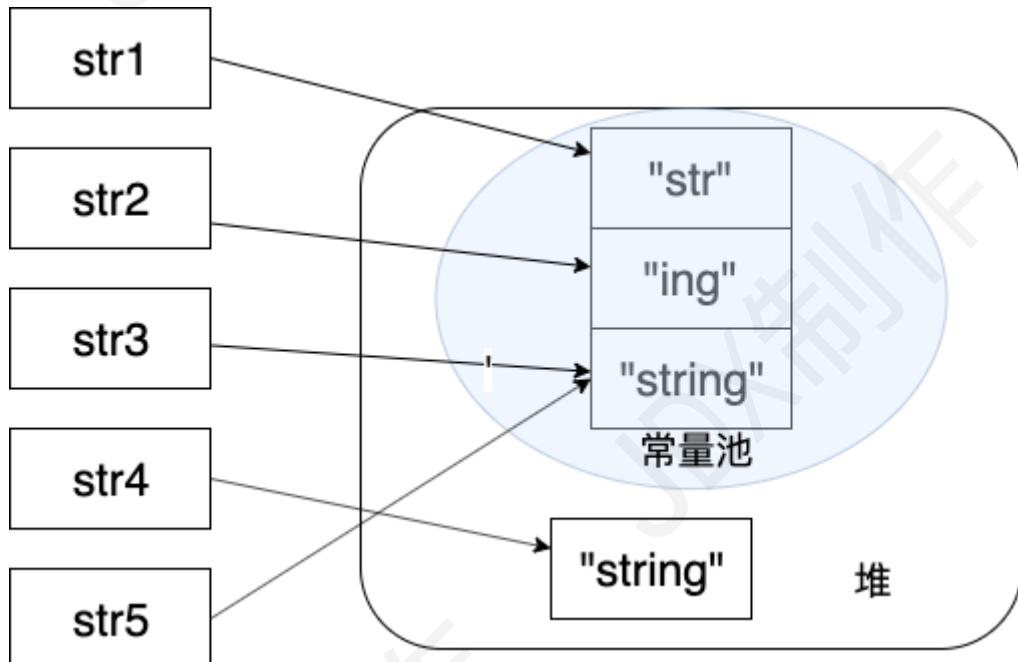
字符串拼接：

```

String str1 = "str";
String str2 = "ing";

String str3 = "str" + "ing"; //常量池中的对象
String str4 = str1 + str2; //在堆上创建的新的对象
String str5 = "string"; //常量池中的对象
System.out.println(str3 == str4); //false
System.out.println(str3 == str5); //true
System.out.println(str4 == str5); //false

```



尽量避免多个字符串拼接，因为这样会重新创建对象。如果需要改变字符串的话，可以使用 StringBuilder 或者 StringBuffer。

1.4.2 String s1 = new String("abc");这句话创建了几个字符串对象？

将创建 1 或 2 个字符串。如果池中已存在字符串常量“abc”，则只会在堆空间创建一个字符串常量“abc”。如果池中没有字符串常量“abc”，那么它将首先在池中创建，然后在堆空间中创建，因此将创建总共 2 个字符串对象。

验证：

```
String s1 = new String("abc");// 堆内存的地址值
String s2 = "abc";
System.out.println(s1 == s2);// 输出 false,因为一个是堆内存，一个是常量池的内
存，故两者是不同的。
System.out.println(s1.equals(s2));// 输出 true
```

结果：

```
false
true
```

1.4.3 8 种基本类型的包装类和常量池

Java 基本类型的包装类的大部分都实现了常量池技术，即
Byte,Short, Integer, Long, Character, Boolean；前面 4 种包装类默认创建了数值[-128, 127] 的相应类型的缓存数据，Character 创建了数值在[0,127]范围的缓存数据， Boolean 直接返回True Or False。如果超出对应范围仍然会去创建新的对象。

```
public static Boolean valueOf(boolean b) {
    return (b ? TRUE : FALSE);
}

private static class CharacterCache {
    private CharacterCache(){}
    static final Character cache[] = new Character[127 + 1];
    static {
        for (int i = 0; i < cache.length; i++)
            cache[i] = new Character((char)i);
    }
}
```

两种浮点数类型的包装类 Float, Double 并没有实现常量池技术。**

```
Integer i1 = 33;
Integer i2 = 33;
System.out.println(i1 == i2); // 输出 true
Integer i11 = 333;
Integer i22 = 333;
System.out.println(i11 == i22); // 输出 false
Double i3 = 1.2;
Double i4 = 1.2;
System.out.println(i3 == i4); // 输出 false
```

Integer 缓存源代码：

```

/**
 *此方法将始终缓存-128 到 127（包括端点）范围内的值，并可以缓存此范围之外的其他值。
 */
public static Integer valueof(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}

```

应用场景：

1. Integer i1=40; Java 在编译的时候会直接将代码封装成 Integer i1=Integer.valueOf(40);, 从而使常量池中的对象。
2. Integer i1 = new Integer(40);这种情况下会创建新的对象。

```

Integer i1 = 40;
Integer i2 = new Integer(40);
System.out.println(i1==i2); //输出 false

```

Integer 比较更丰富的一个例子：

```

Integer i1 = 40;
Integer i2 = 40;
Integer i3 = 0;
Integer i4 = new Integer(40);
Integer i5 = new Integer(40);
Integer i6 = new Integer(0);

System.out.println("i1=i2 " + (i1 == i2));
System.out.println("i1=i2+i3 " + (i1 == i2 + i3));
System.out.println("i1=i4 " + (i1 == i4));
System.out.println("i4=i5 " + (i4 == i5));
System.out.println("i4=i5+i6 " + (i4 == i5 + i6));
System.out.println("40=i5+i6 " + (40 == i5 + i6));

```

结果：

```

i1=i2 true
i1=i2+i3 true
i1=i4 false
i4=i5 false
i4=i5+i6 true
40=i5+i6 true

```

解释：

语句 $i4 == i5 + i6$, 因为+这个操作符不适用于 Integer 对象, 首先 i5 和 i6 进行自动拆箱操作, 进行数值相加, 即 $i4 == 40$ 。然后 Integer 对象无法与数值进行直接比较, 所以 i4 自动拆箱转为 int 值 40, 最终这条语句转为 $40 == 40$ 进行数值比较。

2. JVM垃圾回收

2.1 揭开 JVM 内存分配与回收的神秘面纱

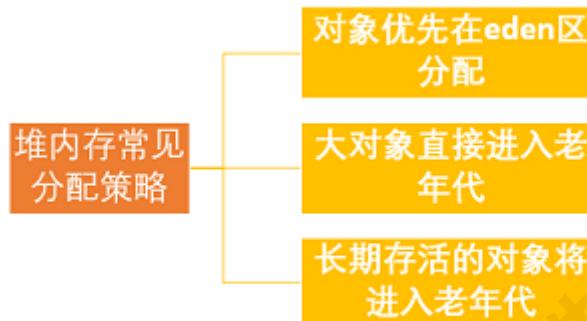
Java 的自动内存管理主要是针对对象内存的回收和对象内存的分配。同时，Java 自动内存管理最核心的功能是 **堆** 内存中对象的分配与回收。

Java 堆是垃圾收集器管理的主要区域，因此也被称作**GC 堆 (Garbage Collected Heap)** .从垃圾回收的角度，由于现在收集器基本都采用分代垃圾收集算法，所以 Java 堆还可以细分为：新生代和老年代：再细致一点有：Eden 空间、From Survivor、To Survivor 空间等。**进一步划分的目的是更好地回收内存，或者更快地分配内存。**

堆空间的基本结构：



上图所示的 eden 区、s0("From") 区、s1("To") 区都属于新生代，tentired 区属于老年代。大部分情况，对象都会首先在 Eden 区域分配，在一次新生代垃圾回收后，如果对象还存活，则会进入 s1("To")，并且对象的年龄还会加 1(Eden 区->Survivor 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 `-XX:MaxTenuringThreshold` 来设置。经过这次GC后，Eden区和"From"区已经被清空。这个时候，"From"和"To"会交换他们的角色，也就是新的"To"就是上次GC前的"From"，新的"From"就是上次GC前的"To"。不管怎样，都会保证名为To的Survivor区域是空的。Minor GC会一直重复这样的过程，直到"To"区被填满，"To"区被填满之后，会将所有对象移动到老年代中。



2.1.1 对象优先在 eden 区分配

目前主流的垃圾收集器都会采用分代回收算法，因此需要将堆内存分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

大多数情况下，对象在新生代中 eden 区分配。当 eden 区没有足够空间进行分配时，虚拟机将发起一次 Minor GC.下面我们来进行实际测试以下。

在测试之前我们先来看看 **Minor GC 和 Full GC 有什么不同呢？**

- **新生代 GC (Minor GC)** :指发生新生代的的垃圾收集动作，Minor GC 非常频繁，回收速度一般也比较快。
- **老年代 GC (Major GC/Full GC)** :指发生在老年代的 GC，出现了 Major GC 经常会伴随至少一次的 Minor GC (并非绝对) ， Major GC 的速度一般会比 Minor GC 的慢 10 倍以上。

测试：

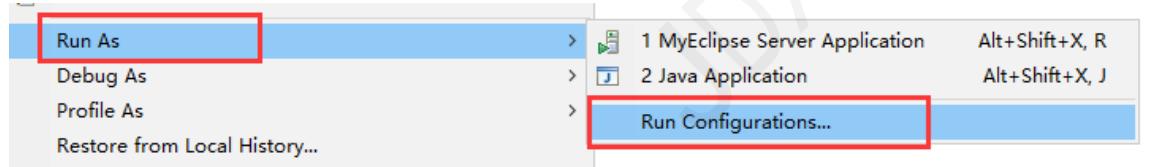
```

public class GCTest {

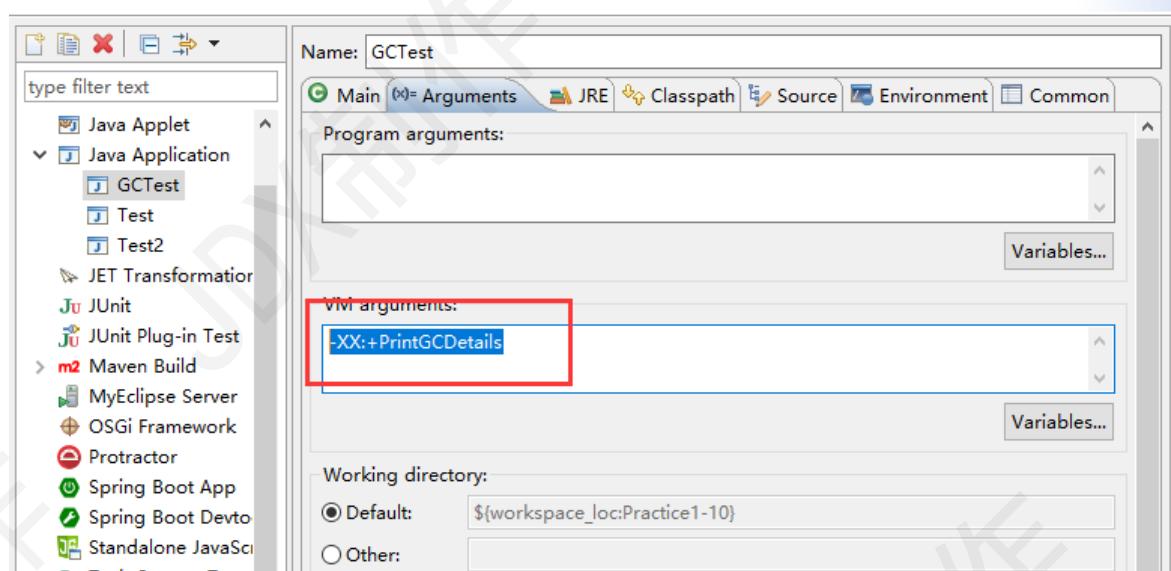
    public static void main(String[] args) {
        byte[] allocation1, allocation2;
        allocation1 = new byte[30900*1024];
        //allocation2 = new byte[900*1024];
    }
}

```

通过以下方式运行：



添加的参数：-XX:+PrintGCDetails



运行结果 (红色字体描述有误，应该是对于于 JDK1.7 的永久代)：

```

Heap 新生代
PSYoungGen total 38400K, used 33280K [0x00000000d5d00000, 0x00000000d8780000, 0x0000000100000000)
eden space 33280K, 100% used [0x00000000d5d00000, 0x00000000d7d80000, 0x00000000d7d80000)
from space 5120K, 0% used [0x00000000d8280000, 0x00000000d8280000, 0x00000000d8780000)
to space 5120K, 0% used [0x00000000d7d80000, 0x00000000d7d80000, 0x00000000d8280000)
ParOldGen 老年代
total 87552K, used 0K [0x0000000081600000, 0x0000000086b80000, 0x00000000d5d00000)
object space 87552K, 0% used [0x0000000081600000, 0x0000000086b80000, 0x00000000d5d00000)
Metaspace 元空间
used 2621K, capacity 4486K, committed 4864K, reserved 1056768K
class space used 283K, capacity 386K, committed 512K, reserved 1048576K
元空间对应于JDK1.8的永久代

```

从上图我们可以看出 eden 区内存几乎已经被分配完全（即使程序什么也不做，新生代也会使用 2000 多 k 内存）。假如我们再为 allocation2 分配内存会出现什么情况呢？

```

allocation2 = new byte[900*1024];

```

```

[[GC (Allocation Failure) [PSYoungGen: 32897K->768K(38400K)] 32897K->31676K(125952K), 0.0229658 secs] [Times: us
Heap
PSYoungGen total 38400K, used 2001K [0x00000000d5d00000, 0x00000000da800000, 0x0000000100000000)
eden space 33280K, 3% used [0x00000000d5d00000, 0x00000000d5e344b8, 0x00000000d7d80000)
from space 5120K, 15% used [0x00000000d7d80000, 0x00000000d7e40030, 0x00000000d8280000)
to space 5120K, 0% used [0x00000000da300000, 0x00000000da300000, 0x00000000da800000)
ParOldGen total 87552K, used 30908K [0x0000000081600000, 0x0000000086b80000, 0x00000000d5d00000)
object space 87552K, 35% used [0x0000000081600000, 0x000000008342f010, 0x0000000086b80000)
Metaspace used 2621K, capacity 4486K, committed 4864K, reserved 1056768K
class space used 283K, capacity 386K, committed 512K, reserved 1048576K

```

简单解释一下为什么会出现这种情况：因为给 allocation2 分配内存的时候 eden 区内存几乎已经被分配完了，我们刚刚讲了当 Eden 区没有足够空间进行分配时，虚拟机将发起一次 Minor GC.GC 期间虚拟机又发现 allocation1 无法存入 Survivor 空间，所以只好通过 **分配担保机制** 把新生代的对象提前转移到老年代中去，老年代上的空间足够存放 allocation1，所以不会出现 Full GC。执行 Minor GC 后，后面分配的对象如果能够存在 eden 区的话，还是会在 eden 区分配内存。可以执行如下代码验证：

```
public class GCTest {  
    public static void main(String[] args) {  
        byte[] allocation1, allocation2, allocation3, allocation4, allocation5;  
        allocation1 = new byte[32000*1024];  
        allocation2 = new byte[1000*1024];  
        allocation3 = new byte[1000*1024];  
        allocation4 = new byte[1000*1024];  
        allocation5 = new byte[1000*1024];  
    }  
}
```

2.1.2 大对象直接进入老年代

大对象就是需要大量连续内存空间的对象（比如：字符串、数组）。

为什么要这样呢？

为了避免为大对象分配内存时由于分配担保机制带来的复制而降低效率。

2.1.3 长期存活的对象将进入老年代

既然虚拟机采用了分代收集的思想来管理内存，那么内存回收时就必须能识别哪些对象应放在新生代，哪些对象应放在老年代中。为了做到这一点，虚拟机给每个对象一个对象年龄（Age）计数器。

如果对象在 Eden 出生并经过第一次 Minor GC 后仍然能够存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间中，并将对象年龄设为 1. 对象在 Survivor 中每熬过一次 MinorGC, 年龄就增加 1 岁，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 `-XX:MaxTenuringThreshold` 来设置。

2.1.4 动态对象年龄判定

大部分情况，对象都会首先在 Eden 区域分配，在一次新生代垃圾回收后，如果对象还存活，则会进入 s0 或者 s1，并且对象的年龄还会加 1(Eden 区->Survivor 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 `-XX:MaxTenuringThreshold` 来设置。

修正：“Hotspot 遍历所有对象时，按照年龄从小到大对对其所占用的大小进行累积，当累积的某个年龄大小超过了 survivor 区的一半时，取这个年龄和 MaxTenuringThreshold 中更小的一个值，作为新的晋升年龄阈值”。

动态年龄计算的代码如下

```
uint ageTable::compute_tenuring_threshold(size_t survivor_capacity) {  
    //survivor_capacity是survivor空间的大小  
    size_t desired_survivor_size = (size_t)((double)  
    survivor_capacity)*TargetSurvivorRatio)/100);  
    size_t total = 0;  
    uint age = 1;  
    while (age < table_size) {  
        total += sizes[age]; //sizes数组是每个年龄段对象大小
```

```

if (total > desired_survivor_size) break;
age++;
}
uint result = age < MaxTenuringThreshold ? age : MaxTenuringThreshold;
...
}

```

额外补充说明：关于默认的晋升年龄是15，这个说法的来源大部分都是《深入理解Java虚拟机》这本书。

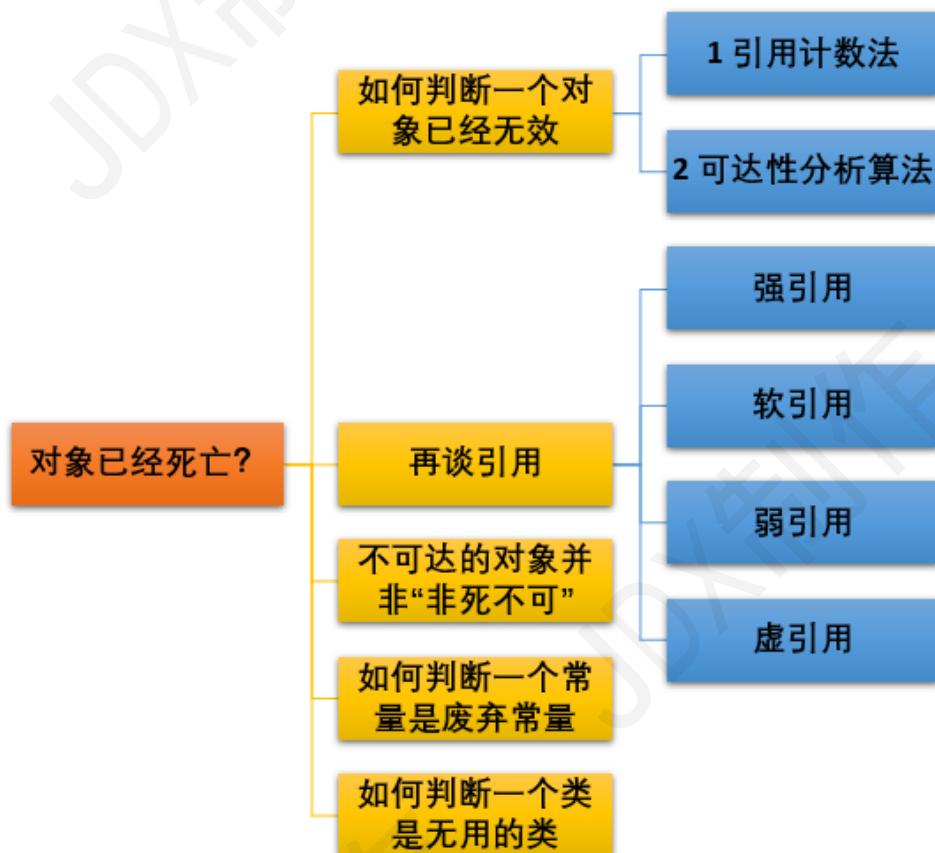
如果你去Oracle的官网阅读相关的虚拟机参数，你会发现

`XX:MaxTenuringThreshold=threshold`这里有个说明

Sets the maximum tenuring threshold for use in adaptive GC sizing. The largest value is 15. The default value is 15 for the parallel (throughput) collector, and 6 for the CMS collector. 默认晋升年龄并不都是15，这个是要区分垃圾收集器的，CMS就是6。

2.2 对象已经死亡？

堆中几乎放着所有的对象实例，对堆垃圾回收前的第一步就是要判断那些对象已经死亡（即不能再被任何途径使用的对象）。



2.2.1 引用计数法

给对象中添加一个引用计数器，每当有一个地方引用它，计数器就加1；当引用失效，计数器就减1；任何时候计数器为0的对象就是不可能再被使用的。

这个方法实现简单，效率高，但是目前主流的虚拟机中并没有选择这个算法来管理内存，其最主要的原因是它很难解决对象之间相互循环引用的问题。所谓对象之间的相互引用问题，如下面代码所示：除了对象 objA 和 objB 相互引用着对方之外，这两个对象之间再无任何引用。但是他们因为互相引用对方，导致它们的引用计数器都不为0，于是引用计数算法无法通知 GC 回收器回收他们。

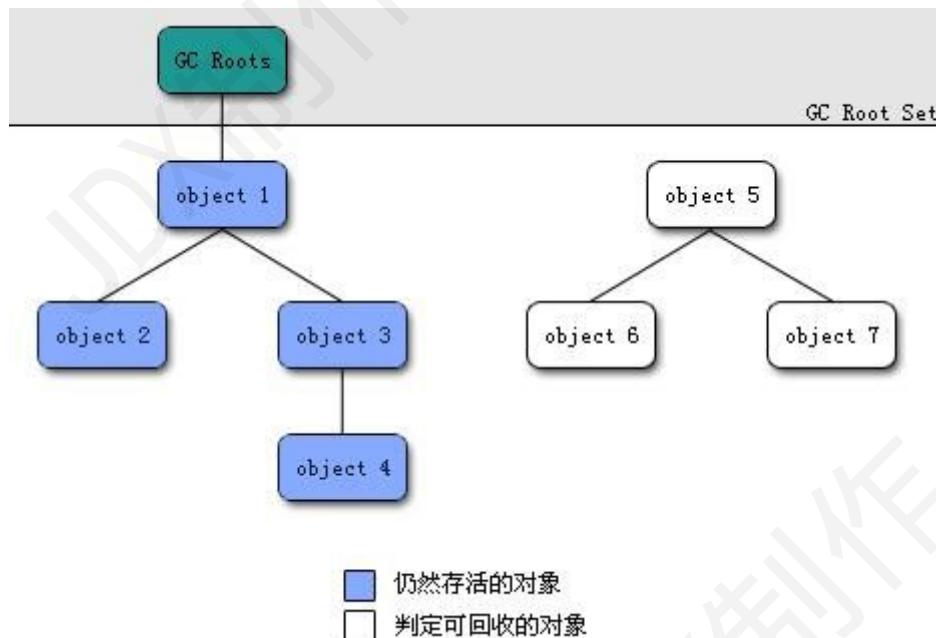
```

public class ReferenceCountingGc {
    Object instance = null;
    public static void main(String[] args) {
        ReferenceCountingGc objA = new ReferenceCountingGc();
        ReferenceCountingGc objB = new ReferenceCountingGc();
        objA.instance = objB;
        objB.instance = objA;
        objA = null;
        objB = null;
    }
}

```

2.2.2 可达性分析算法

这个算法的基本思想就是通过一系列的称为“**GC Roots**”的对象作为起点，从这些节点开始向下搜索，节点所走过的路径称为引用链，当一个对象到 GC Roots 没有任何引用链相连的话，则证明此对象是不可用的。



2.2.3 再谈引用

无论是通过引用计数法判断对象引用数量，还是通过可达性分析法判断对象的引用链是否可达，判定对象的存活都与“引用”有关。

JDK1.2 之前，Java 中引用的定义很传统：如果 reference 类型的数据存储的数值代表的是另一块内存的起始地址，就称这块内存代表一个引用。

JDK1.2 以后，Java 对引用的概念进行了扩充，将引用分为强引用、软引用、弱引用、虚引用四种（引用强度逐渐减弱）

1. 强引用 (StrongReference)

以前我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于**必不可少的生活用品**，垃圾回收器绝不会回收它。当内存空间不足，Java 虚拟机宁愿抛出 OutOfMemoryError 错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

2. 软引用 (SoftReference)

如果一个对象只具有软引用，那就类似于**可有可无的生活用品**。如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，JAVA 虚拟机就会把这个软引用加入到与之关联的引用队列中。

3. 弱引用 (WeakReference)

如果一个对象只具有弱引用，那就类似于**可有可无的生活用品**。弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。

4. 虚引用 (PhantomReference)

"虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。

虚引用主要用来跟踪对象被垃圾回收的活动。

虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列（ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

特别注意，在程序设计中一般很少使用弱引用与虚引用，使用软引用的情况较多，这是因为**软引用可以加速 JVM 对垃圾内存的回收速度，可以维护系统的运行安全，防止内存溢出 (OutOfMemory) 等问题的产生。**

2.2.4 不可达的对象并非“非死不可”

即使在可达性分析法中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓刑阶段”，要真正宣告一个对象死亡，至少要经历两次标记过程；可达性分析法中不可达的对象被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行 finalize 方法。当对象没有覆盖 finalize 方法，或 finalize 方法已经被虚拟机调用过时，虚拟机将这两种情况视为没有必要执行。

被判定为需要执行的对象将会被放在一个队列中进行第二次标记，除非这个对象与引用链上的任何一个对象建立关联，否则就会被真的回收。

2.2.5 如何判断一个常量是废弃常量

运行时常量池主要回收的是废弃的常量。那么，我们如何判断一个常量是废弃常量呢？

假如在常量池中存在字符串 "abc"，如果当前没有任何 String 对象引用该字符串常量的话，就说明常量 "abc" 就是废弃常量，如果这时发生内存回收的话而且有必要的话，"abc" 就会被系统清理出常量池。

2.2.6 如何判断一个类是无用的类

方法区主要回收的是无用的类，那么如何判断一个类是无用的类的呢？

判定一个常量是否是“废弃常量”比较简单，而要判定一个类是否是“无用的类”的条件则相对苛刻许多。

类需要同时满足下面 3 个条件才能算是“**无用的类**”：

- 该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。
- 加载该类的 ClassLoader 已经被回收。

- 该类对应的 `java.lang.Class` 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

虚拟机可以对满足上述 3 个条件的无用类进行回收，这里说的仅仅是“可以”，而并不是和对象一样不使用了就会必然被回收。

2.3 垃圾收集算法

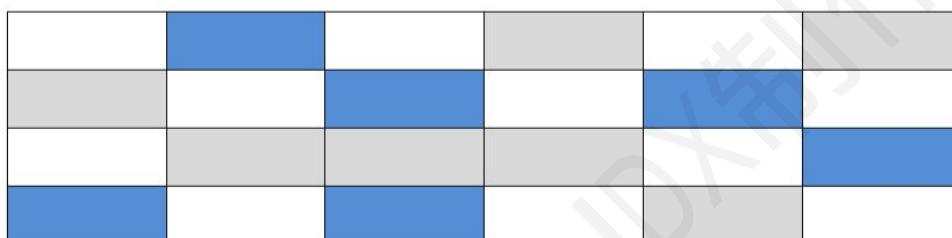


2.3.1 标记-清除算法

该算法分为“标记”和“清除”阶段：首先比较出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象。它是最基础的收集算法，后续的算法都是对其不足进行改进得到。这种垃圾收集算法会带来两个明显的问题：

1. 效率问题
2. 空间问题（标记清除后会产生大量不连续的碎片）

内存整理前



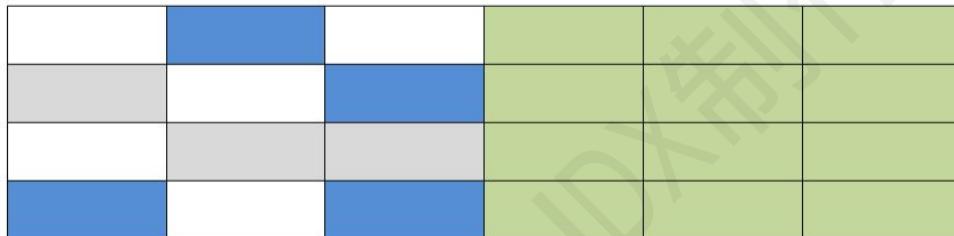
内存整理后



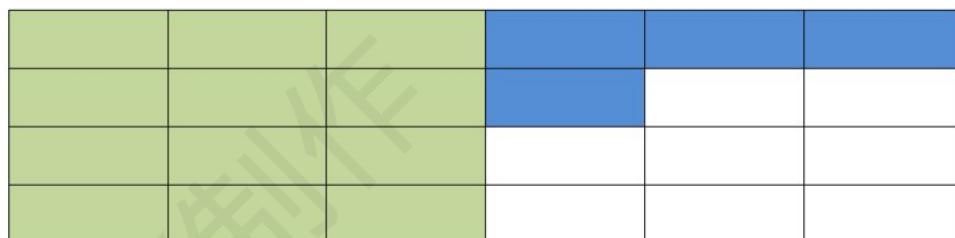
2.3.2 复制算法

为了解决效率问题，“复制”收集算法出现了。它可以将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。

内存整理前



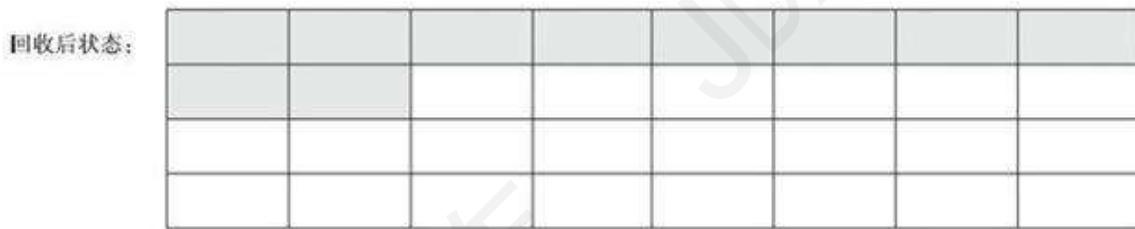
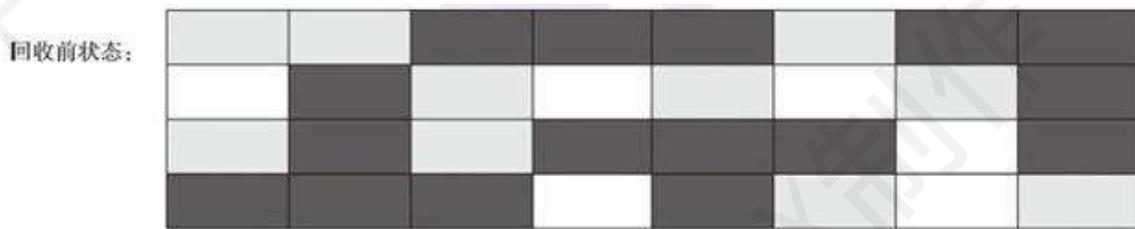
内存整理后



可用内存	可回收内存	存活对象	保留内存
------	-------	------	------

2.3.3 标记-整理算法

根据老年代的特点提出的一种标记算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象回收，而是让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存。



存活对象	可回收	未使用
------	-----	-----

2.3.4 分代收集算法

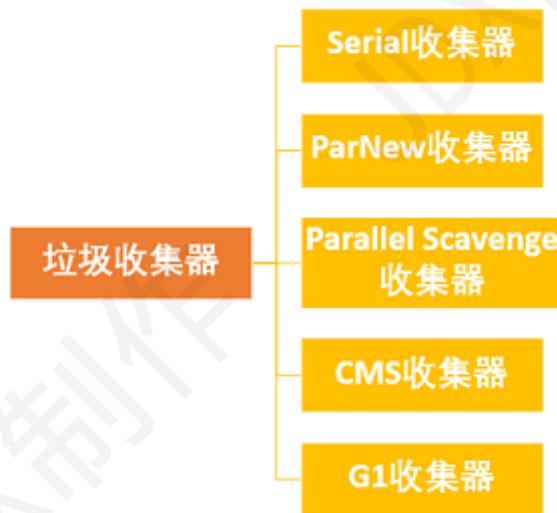
当前虚拟机的垃圾收集都采用分代收集算法，这种算法没有什么新的思想，只是根据对象存活周期的不同将内存分为几块。一般将 java 堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

比如在新生代中，每次收集都会有大量对象死去，所以可以选择复制算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集。

延伸面试问题：HotSpot 为什么要分为新生代和老年代？

根据上面的对分代收集算法的介绍回答。

2.4 垃圾收集器



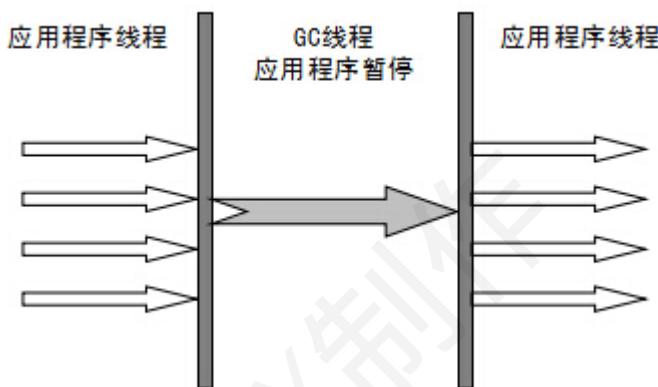
如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。

虽然我们对各个收集器进行比较，但并非要挑选出一个最好的收集器。因为直到现在为止还没有最好的垃圾收集器出现，更加没有万能的垃圾收集器，**我们能做的就是根据具体应用场景选择适合自己的垃圾收集器**。试想一下：如果有一种四海之内、任何场景下都适用的完美收集器存在，那么我们的 HotSpot 虚拟机就不会实现那么多不同的垃圾收集器了。

2.4.1 Serial 收集器

Serial（串行）收集器收集器是最基本、历史最悠久的垃圾收集器了。大家看名字就知道这个收集器是一个单线程收集器了。它的“**单线程**”的意义不仅仅意味着它只会使用一条垃圾收集线程去完成垃圾收集工作，更重要的是它在进行垃圾收集工作的时候必须暂停其他所有的工作线程（"Stop The World"），直到它收集结束。

新生代采用复制算法，老年代采用标记-整理算法。



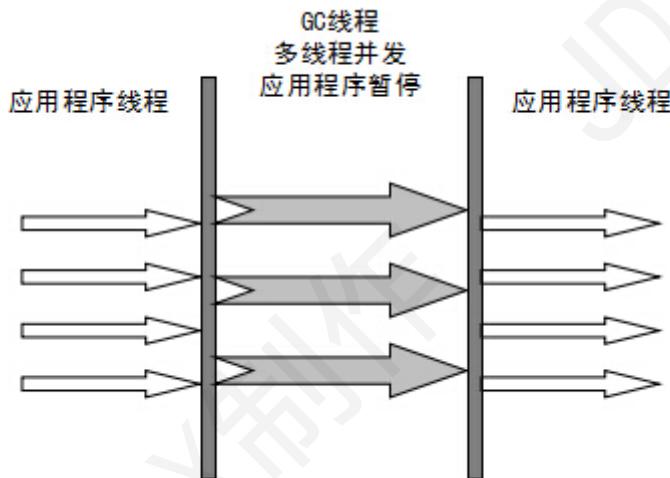
虚拟机的设计者们当然知道 Stop The World 带来的不良用户体验，所以在后续的垃圾收集器设计中停顿时间在不断缩短（仍然还有停顿，寻找最优秀的垃圾收集器的过程仍然在继续）。

但是 Serial 收集器有没有优于其他垃圾收集器的地方呢？当然有，它简单而高效（与其他收集器的单线程相比）。Serial 收集器由于没有线程交互的开销，自然可以获得很高的单线程收集效率。Serial 收集器对于运行在 Client 模式下的虚拟机来说是个不错的选择。

2.4.2 ParNew 收集器

ParNew 收集器其实就是 Serial 收集器的多线程版本，除了使用多线程进行垃圾收集外，其余行为（控制参数、收集算法、回收策略等等）和 Serial 收集器完全一样。

新生代采用复制算法，老年代采用标记-整理算法。



它是许多运行在 Server 模式下的虚拟机的首要选择，除了 Serial 收集器外，只有它能与 CMS 收集器（真正意义上的并发收集器，后面会介绍到）配合工作。

并行和并发概念补充：

- **并行 (Parallel)**：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。
- **并发 (Concurrent)**：指用户线程与垃圾收集线程同时执行（但不一定是并行，可能会交替执行），用户程序在继续运行，而垃圾收集器运行在另一个 CPU 上。

2.4.3 Parallel Scavenge 收集器

Parallel Scavenge 收集器也是使用复制算法的多线程收集器，它看上去几乎和 ParNew 都一样。那么它有什么特别之处呢？

`-XX:+UseParallelGC`

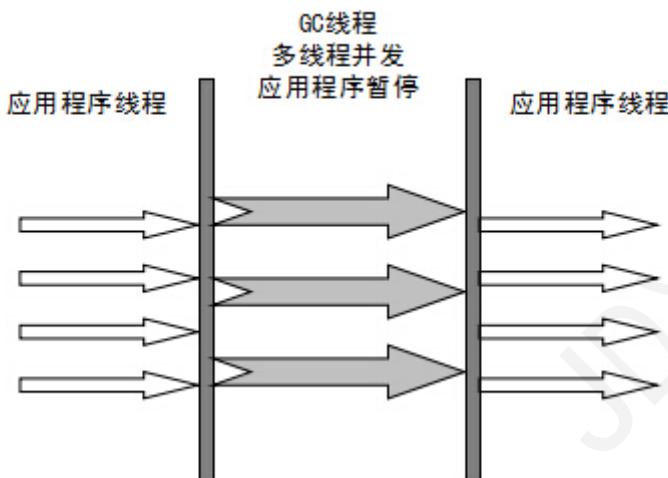
使用 Parallel 收集器+ 老年代串行

`-XX:+UseParallelOldGC`

使用 Parallel 收集器+ 老年代并行

Parallel Scavenge 收集器关注点是吞吐量（高效率的利用 CPU）。CMS 等垃圾收集器的关注点更多的是用户线程的停顿时间（提高用户体验）。所谓吞吐量就是 CPU 中用于运行用户代码的时间与 CPU 总消耗时间的比值。Parallel Scavenge 收集器提供了很多参数供用户找到最合适的停顿时间或最大吞吐量，如果对于收集器运作不太了解的话，手工优化存在困难的话可以选择把内存管理优化交给虚拟机去完成也是一个不错的选择。

新生代采用复制算法，老年代采用标记-整理算法。



2.4.4 Serial Old 收集器

Serial 收集器的老年代版本，它同样是一个单线程收集器。它主要有两大用途：一种用途是在 JDK1.5 以及以前的版本中与 Parallel Scavenge 收集器搭配使用，另一种用途是作为 CMS 收集器的后备方案。

2.4.5 Parallel Old 收集器

Parallel Scavenge 收集器的老年代版本。使用多线程和“标记-整理”算法。在注重吞吐量以及 CPU 资源的场合，都可以优先考虑 Parallel Scavenge 收集器和 Parallel Old 收集器。

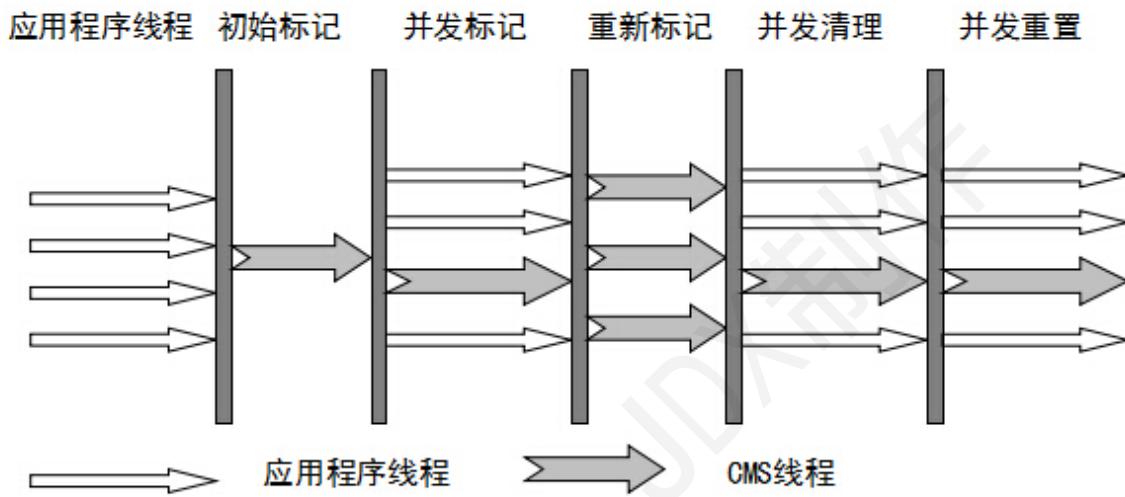
2.4.6 CMS 收集器

CMS (Concurrent Mark Sweep) 收集器是一种以获取最短回收停顿时间为 目标 的收集器。它非常符合在注重用户体验的应用上使用。

CMS (Concurrent Mark Sweep) 收集器是 HotSpot 虚拟机第一款真正意义上的并发收集器，它第一次实现了让垃圾收集线程与用户线程（基本上）同时工作。

从名字中的**Mark Sweep**这两个词可以看出，CMS 收集器是一种“**标记-清除**”算法实现的，它的运作过程相比于前面几种垃圾收集器来说更加复杂一些。整个过程分为四个步骤：

- **初始标记**：暂停所有的其他线程，并记录下直接与 root 相连的对象，速度很快；
- **并发标记**：同时开启 GC 和用户线程，用一个闭包结构去记录可达对象。但在这个阶段结束，这个闭包结构并不能保证包含当前所有的可达对象。因为用户线程可能会不断的更新引用域，所以 GC 线程无法保证可达性分析的实时性。所以这个算法里会跟踪记录这些发生引用更新的地方。
- **重新标记**：重新标记阶段就是为了修正并发标记期间因为用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段的时间稍长，远远比并发标记阶段时间短
- **并发清除**：开启用户线程，同时 GC 线程开始对未标记的区域做清扫。



从它的名字就可以看出它是一款优秀的垃圾收集器，主要优点：**并发收集、低停顿**。但是它有下面三个明显的缺点：

- 对 CPU 资源敏感；
- 无法处理浮动垃圾；
- 它使用的回收算法-“标记-清除”算法会导致收集结束时会有大量空间碎片产生。

2.4.7 G1 收集器

G1 (Garbage-First) 是一款面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存的机器. 以极高概率满足 GC 停顿时间要求的同时,还具备高吞吐量性能特征.

被视为 JDK1.7 中 HotSpot 虚拟机的一个重要进化特征。它具备一下特点：

- **并行与并发**: G1 能充分利用 CPU、多核环境下的硬件优势，使用多个 CPU (CPU 或者 CPU 核心) 来缩短 Stop-The-World 停顿时间。部分其他收集器原本需要停顿 Java 线程执行的 GC 动作，G1 收集器仍然可以通过并发的方式让 java 程序继续执行。
- **分代收集**: 虽然 G1 可以不需要其他收集器配合就能独立管理整个 GC 堆，但是还是保留了分代的概念。
- **空间整合**: 与 CMS 的“标记-清理”算法不同，G1 从整体来看是基于“标记整理”算法实现的收集器；从局部上来看是基于“复制”算法实现的。
- **可预测的停顿**: 这是 G1 相对于 CMS 的另一个大优势，降低停顿时间是 G1 和 CMS 共同的关注点，但 G1 除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为 M 毫秒的时间片段内。

G1 收集器的运作大致分为以下几个步骤：

- 初始标记
- 并发标记
- 最终标记
- 筛选回收

G1 收集器在后台维护了一个优先列表，每次根据允许的收集时间，优先选择回收价值最大的 Region(这也就是它的名字 Garbage-First 的由来)。这种使用 Region 划分内存空间以及有优先级的区域回收方式，保证了 G1 收集器在有限时间内可以尽可能高的收集效率 (把内存化整为零)。

3. JDK 监控和故障处理工具

3.1 JDK 命令行工具

这些命令在 JDK 安装目录下的 bin 目录下：

- `jps` (JVM Process Status) : 类似 UNIX 的 `ps` 命令。用户查看所有 Java 进程的启动类、传入参数和 Java 虚拟机参数等信息;
- `jstat` (JVM Statistics Monitoring Tool) : 用于收集 HotSpot 虚拟机各方面的运行数据;
- `jinfo` (Configuration Info for Java) : Configuration Info for Java, 显示虚拟机配置信息;
- `jmap` (Memory Map for Java) : 生成堆转储快照;
- `jhat` (JVM Heap Dump Browser) : 用于分析 heapdump 文件, 它会建立一个 HTTP/HTML 服务器, 让用户可以在浏览器上查看分析结果;
- `jstack` (Stack Trace for Java) : 生成虚拟机当前时刻的线程快照, 线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合。

3.1.1 `jps` : 查看所有 Java 进程

`jps` (JVM Process Status) 命令类似 UNIX 的 `ps` 命令。

`jps` : 显示虚拟机执行主类名称以及这些进程的本地虚拟机唯一 ID (Local Virtual Machine Identifier,LVMID)。`jps -q` : 只输出进程的本地虚拟机唯一 ID。

```
C:\Users\SnailClimb>jps
7360 NettyClient2
17396
7972 Launcher
16504 Jps
17340 NettyServer
```

`jps -l` : 输出主类的全名, 如果进程执行的是 Jar 包, 输出 Jar 路径。

```
C:\Users\SnailClimb>jps -l
7360 firstNettyDemo.NettyClient2
17396
7972 org.jetbrains.jps.cmdline.Launcher
16492 sun.tools.jps.Jps
17340 firstNettyDemo.NettyServer
```

`jps -v` : 输出虚拟机进程启动时 JVM 参数。

`jps -m` : 输出传递给 Java 进程 main() 函数的参数。

3.1.2 `jstat` : 监视虚拟机各种运行状态信息

`jstat` (JVM Statistics Monitoring Tool) 使用于监视虚拟机各种运行状态信息的命令行工具。它可以显示本地或者远程 (需要远程主机提供 RMI 支持) 虚拟机进程中的类信息、内存、垃圾收集、JIT 编译等运行数据, 在没有 GUI, 只提供了纯文本控制台环境的服务器上, 它将是运行期间定位虚拟机性能问题的首选工具。

`jstat` 命令使用格式:

```
jstat -<option> [<t> [-h<lines>]] <vmid> [<interval> [<count>]]
```

比如 `jstat -gc -h3 31736 1000 10` 表示分析进程 id 为 31736 的 gc 情况, 每隔 1000ms 打印一次记录, 打印 10 次后打印指标头部。

常见的 option 如下:

- `jstat -class vmid` : 显示 ClassLoader 的相关信息;
- `jstat -compiler vmid` : 显示 JIT 编译的相关信息;

- `jstat -gc vmid`：显示与 GC 相关的堆信息；
- `jstat -gccapacity vmid`：显示各个代的容量及使用情况；
- `jstat -gcnew vmid`：显示新生代信息；
- `jstat -gcnewcapacity vmid`：显示新生代大小与使用情况；
- `jstat -gcold vmid`：显示老年代和永久代的行为统计，从 jdk1.8 开始，该选项仅表示老年代，因为永久代被移除了；
- `jstat -gcoldcapacity vmid`：显示老年代的大小；
- `jstat -gcpermcapacity vmid`：显示永久代大小，从 jdk1.8 开始，该选项不存在了，因为永久代被移除了；
- `jstat -gcutil vmid`：显示垃圾收集信息；

另外，加上 `-t` 参数可以在输出信息上加一个 Timestamp 列，显示程序的运行时间。

3.1.3 jinfo：实时地查看和调整虚拟机各项参数

`jinfo vmid`：输出当前 jvm 进程的全部参数和系统属性（第一部分是系统的属性，第二部分是 JVM 的参数）。

`jinfo -flag name vmid`：输出对应名称的参数的具体值。比如输出 MaxHeapSize、查看当前 jvm 进程是否开启打印 GC 日志（`-XX:PrintGCDetails`：详细 GC 日志模式，这两个都是默认关闭的）。

```
C:\Users\SnailClimb>jinfo -flag MaxHeapSize 17340
-XX:MaxHeapSize=2124414976
C:\Users\SnailClimb>jinfo -flag PrintGC 17340
-XX:-PrintGC
```

使用 jinfo 可以在不重启虚拟机的情况下，可以动态的修改 jvm 的参数。尤其在线上的环境特别有用，请看下面的例子：

`jinfo -flag [+|-]name vmid` 开启或者关闭对应名称的参数。

```
C:\Users\SnailClimb>jinfo -flag PrintGC 17340
-XX:-PrintGC

C:\Users\SnailClimb>jinfo -flag +PrintGC 17340

C:\Users\SnailClimb>jinfo -flag PrintGC 17340
-XX:+PrintGC
```

3.1.4 jmap：生成堆转储快照

`jmap`（Memory Map for Java）命令用于生成堆转储快照。如果不使用 `jmap` 命令，要想获取 Java 堆转储，可以使用 “`-XX:+HeapDumpOnOutOfMemoryError`” 参数，可以让虚拟机在 OOM 异常出现之后自动生成 dump 文件，Linux 命令下可以通过 `kill -3` 发送进程退出信号也能拿到 dump 文件。

`jmap` 的作用并不仅仅是为了获取 dump 文件，它还可以查询 finalizer 执行队列、Java 堆和永久代的详细信息，如空间使用率、当前使用的是哪种收集器等。和 `jinfo` 一样，`jmap` 有不少功能在 Windows 平台下也是受限制的。

示例：将指定应用程序的堆快照输出到桌面。后面，可以通过 `jhat`、`Visual VM` 等工具分析该堆文件。

```
C:\Users\SnailClimb>jmap -
dump:format=b,file=C:\Users\SnailClimb\Desktop\heap.hprof 17340
Dumping heap to C:\Users\SnailClimb\Desktop\heap.hprof ...
Heap dump file created
```

3.1.5 jhat: 分析 heapdump 文件

jhat 用于分析 heapdump 文件，它会建立一个 HTTP/HTML 服务器，让用户可以在浏览器上查看分析结果。

```
C:\Users\snailclimb>jhat c:\Users\Snailclimb\Desktop\heap.hprof  
Reading from c:\Users\Snailclimb\Desktop\heap.hprof...  
Dump file created Sat May 04 12:30:31 CST 2019  
Snapshot read, resolving...  
Resolving 131419 objects...  
Chasing references, expect 26 dots.....  
Eliminating duplicate references.....  
Snapshot resolved.  
Started HTTP server on port 7000  
Server is ready.
```

访问 <http://localhost:7000/>

3.1.6 jstack :生成虚拟机当前时刻的线程快照

`jstack` (Stack Trace for Java) 命令用于生成虚拟机当前时刻的线程快照。线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合。

生成线程快照的目的是定位线程长时间出现停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等都是导致线程长时间停顿的原因。线程出现停顿的时候通过 jstack 来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做些什么事情，或者在等待些什么资源。

下面是一个线程死锁的代码。我们下面会通过 `jstack` 命令进行死锁检查，输出死锁信息，找到发生死锁的线程。

```
public class DeadLockDemo {  
    private static Object resource1 = new Object() //资源 1  
    private static Object resource2 = new Object() //资源 2  
  
    public static void main(String[] args) {  
        new Thread(() -> {  
            synchronized (resource1) {  
                System.out.println(Thread.currentThread() + "get resource1");  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
                System.out.println(Thread.currentThread() + "waiting get  
resource2");  
                synchronized (resource2) {  
                    System.out.println(Thread.currentThread() + "get  
resource2");  
                }  
            }  
        }, "线程 1").start();  
  
        new Thread(() -> {  
            synchronized (resource2) {  
                System.out.println(Thread.currentThread() + "get resource2");  
                try {  
                    Thread.sleep(1000);  
                }  
            }  
        }).start();  
    }  
}
```

```

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread() + "waiting get
resource1");
    synchronized (resource1) {
        System.out.println(Thread.currentThread() + "get
resource1");
    }
}
}, "线程 2").start();
}
}

```

Output

```

Thread[线程 1,5,main]get resource1
Thread[线程 2,5,main]get resource2
Thread[线程 1,5,main]waiting get resource2
Thread[线程 2,5,main]waiting get resource1

```

线程 A 通过 synchronized (resource1) 获得 resource1 的监视器锁，然后通过 Thread.sleep(1000); 让线程 A 休眠 1s 为的是让线程 B 得到执行然后获取到 resource2 的监视器锁。线程 A 和线程 B 休眠结束了都开始企图请求获取对方的资源，然后这两个线程就会陷入互相等待的状态，这也就产生了死锁。

通过 jstack 命令分析：

```

C:\Users\SnailClimb>jps
13792 KotlinCompileDaemon
7360 NettyClient2
17396
7972 Launcher
8932 Launcher
9256 DeadLockDemo
10764 Jps
17340 NettyServer

C:\Users\SnailClimb>jstack 9256

```

输出的部分内容如下：

```

Found one Java-level deadlock:
=====
"线程 2":
    waiting to lock monitor 0x00000000033e668 (object 0x0000000d5efe1c0, a
java.lang.Object),
    which is held by "线程 1"
"线程 1":
    waiting to lock monitor 0x00000000033be88 (object 0x0000000d5efe1d0, a
java.lang.Object),
    which is held by "线程 2"

Java stack information for the threads listed above:
=====

```

```

"线程 2":
  at DeadLockDemo.$lambda$main$1(DeadLockDemo.java:31)
  - waiting to lock <0x0000000d5efe1c0> (a java.lang.Object)
  - locked <0x0000000d5efe1d0> (a java.lang.Object)
  at DeadLockDemo$$Lambda$2/1078694789.run(Unknown Source)
  at java.lang.Thread.run(Thread.java:748)

"线程 1":
  at DeadLockDemo.$lambda$main$0(DeadLockDemo.java:16)
  - waiting to lock <0x0000000d5efe1d0> (a java.lang.Object)
  - locked <0x0000000d5efe1c0> (a java.lang.Object)
  at DeadLockDemo$$Lambda$1/1324119927.run(Unknown Source)
  at java.lang.Thread.run(Thread.java:748)

```

Found 1 deadlock.

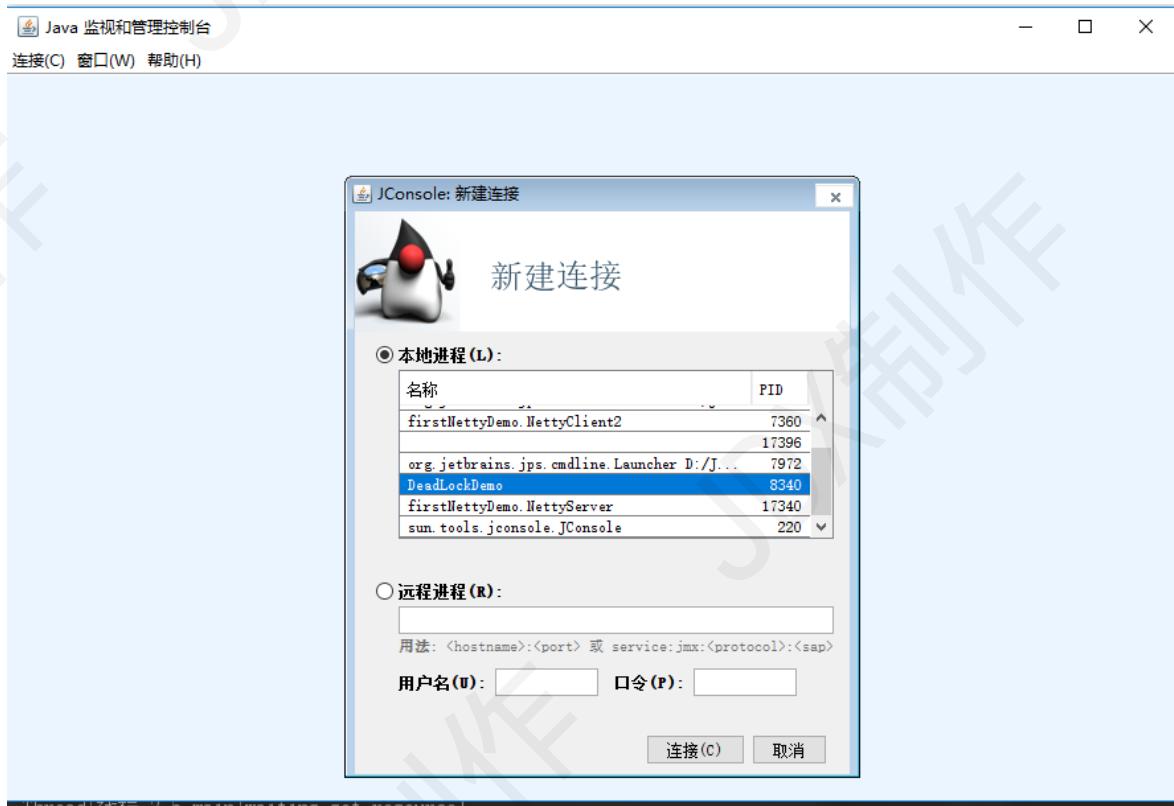
可以看到 `jstack` 命令已经帮我们找到发生死锁的线程的具体信息。

3.2 JDK 可视化分析工具

3.2.1 JConsole:Java 监视与管理控制台

JConsole 是基于 JMX 的可视化监视、管理工具。可以很方便的监视本地及远程服务器的 java 进程的内存使用情况。你可以在控制台输出 `console` 命令启动或者在 JDK 目录下的 bin 目录找到 `jconsole.exe` 然后双击启动。

连接 Jconsole



如果需要使用 JConsole 连接远程进程，可以在远程 Java 程序启动时加上下面这些参数：

```

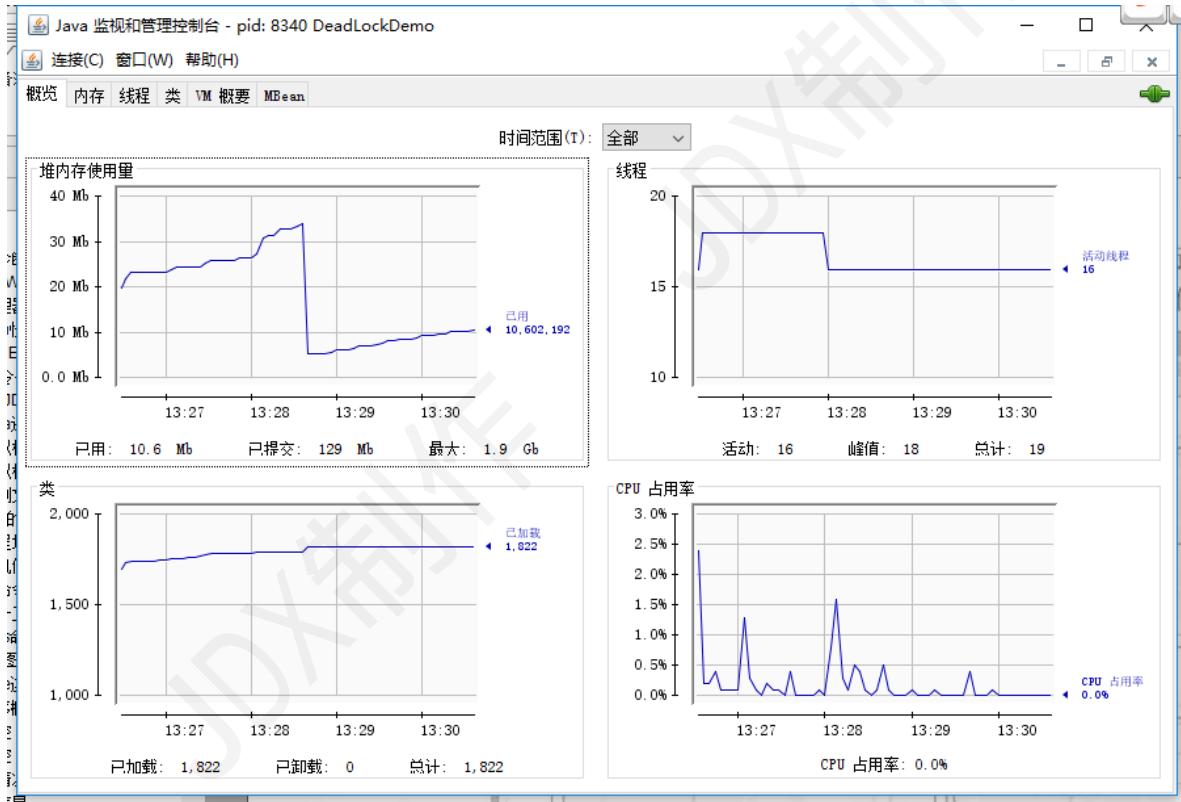
-Djava.rmi.server.hostname=外网访问 ip 地址
-Dcom.sun.management.jmxremote.port=60001 //监控的端口号
-Dcom.sun.management.jmxremote.authenticate=false //关闭认证
-Dcom.sun.management.jmxremote.ssl=false

```

在使用 JConsole 连接时，远程进程地址如下：

外网访问 ip 地址:60001

查看 Java 程序概况

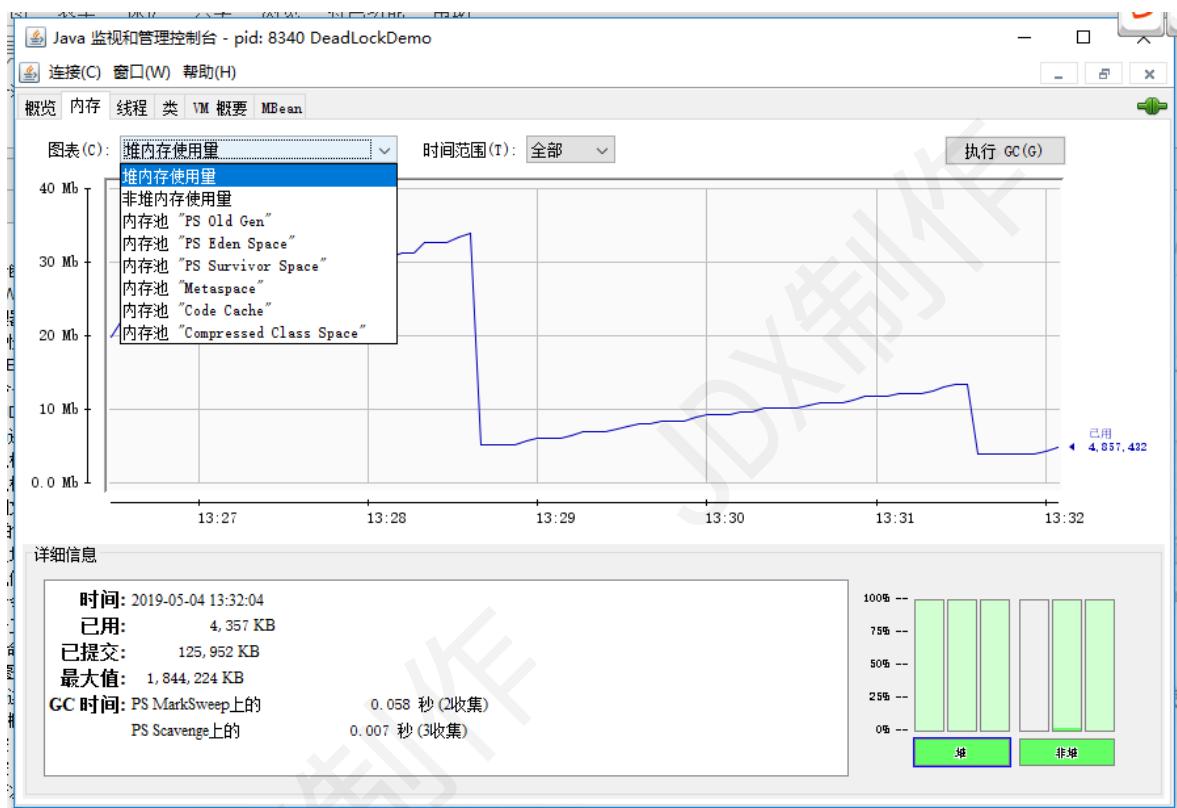


内存监控

JConsole 可以显示当前内存的详细信息。不仅包括堆内存/非堆内存的整体信息，还可以细化到 eden 区、survivor 区等的使用情况，如下图所示。

点击右边的“执行 GC(G)”按钮可以强制应用程序执行一个 Full GC。

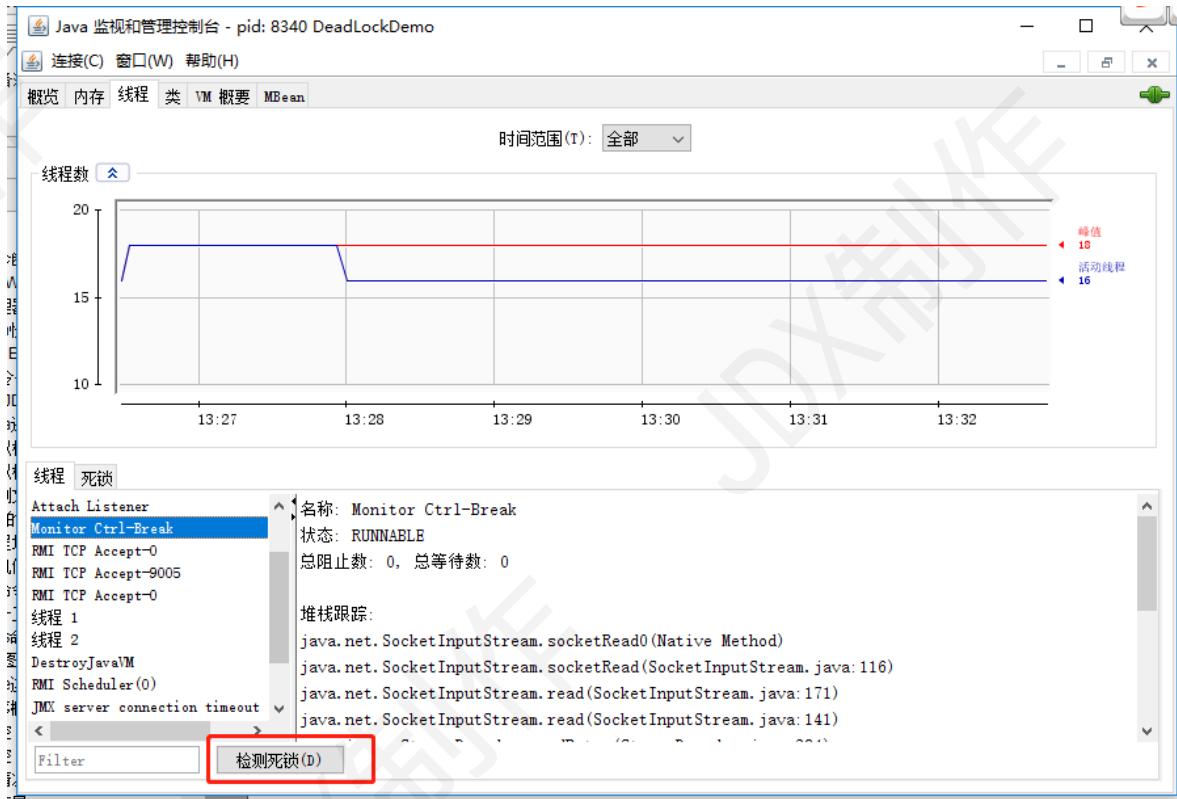
- **新生代 GC (Minor GC)** :指发生新生代的垃圾收集动作，Minor GC 非常频繁，回收速度一般也比较快。
- **老年代 GC (Major GC/Full GC)** :指发生在老年代的 GC，出现了 Major GC 经常会伴随着一次的 Minor GC (并非绝对)，Major GC 的速度一般会比 Minor GC 的慢 10 倍以上。



线程监控

类似我们前面讲的 `jstack` 命令，不过这个是可视化的。

最下面有一个“检测死锁(D)”按钮，点击这个按钮可以自动为你找到发生死锁的线程以及它们的详细信息。



3.2.2 Visual VM: 多合一故障处理工具

VisualVM 提供在 Java 虚拟机 (Java Virtual Machine, JVM) 上运行的 Java 应用程序的详细信息。在 VisualVM 的图形用户界面中，您可以方便、快捷地查看多个 Java 应用程序的相关信息。

下面这段话摘自《深入理解 Java 虚拟机》。

VisualVM (All-in-One Java Troubleshooting Tool) 是到目前为止随 JDK 发布的功能最强大的运行监视和故障处理程序，官方在 VisualVM 的软件说明中写上了“All-in-One”的描述字样，预示着他除了运行监视、故障处理外，还提供了很多其他方面的功能，如性能分析（Profiling）。VisualVM 的性能分析功能甚至比起 JProfiler、YourKit 等专业且收费的 Profiling 工具都不会逊色多少，而且 VisualVM 还有一个很大的优点：不需要被监视的程序基于特殊 Agent 运行，因此他对应用程序的实际性能的影响很小，使得他可以直接应用在生产环境中。这个优点是 JProfiler、YourKit 等工具无法与之媲美的。

VisualVM 基于 NetBeans 平台开发，因此他一开始就具备了插件扩展功能的特性，通过插件扩展支持，VisualVM 可以做到：

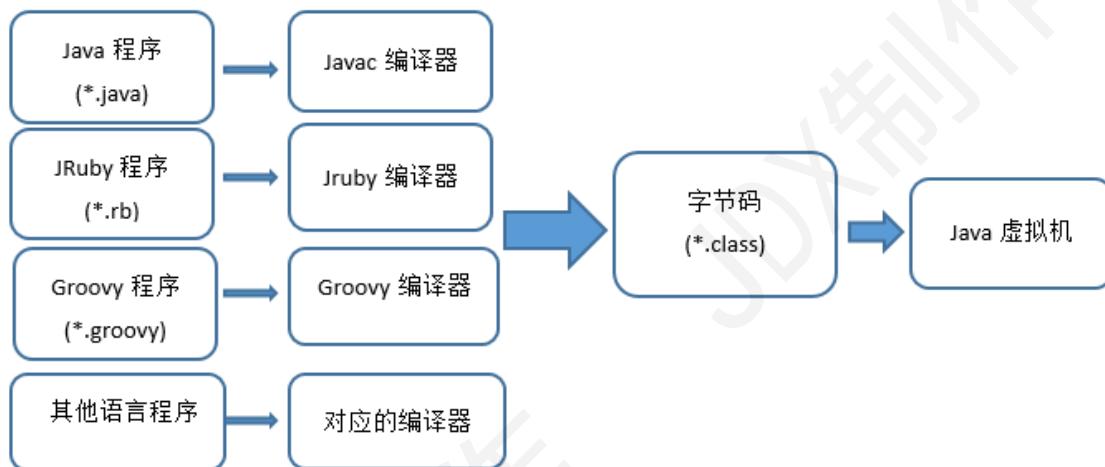
- 显示虚拟机进程以及进程的配置、环境信息 (jps, jinfo)。
- 监视应用程序的 CPU、GC、堆、方法区以及线程的信息 (jstat, jstack)。
- dump 以及分析堆转储快照 (jmap, jhat)。
- 方法级的程序运行性能分析，找到被调用最多、运行时间最长的方法。
- 离线程序快照：收集程序的运行时配置、线程 dump、内存 dump 等信息建立一个快照，可以将快照发送开发者处进行 Bug 反馈。
- 其他 plugins 的无限的可能性.....

4. 类文件结构

4.1 概述

在 Java 中，JVM 可以理解的代码就叫做 **字节码**（即扩展名为 `.class` 的文件），它不面向任何特定的处理器，只面向虚拟机。Java 语言通过字节码的方式，在一定程度上解决了传统解释型语言执行效率低的问题，同时又保留了解释型语言可移植的特点。所以 Java 程序运行时比较高效，而且，由于字节码并不针对一种特定的机器，因此，Java 程序无须重新编译便可在多种不同操作系统的计算机上运行。

Clojure (Lisp 语言的一种方言)、Groovy、Scala 等语言都是运行在 Java 虚拟机之上。下图展示了不同的语言被不同的编译器编译成 `.class` 文件最终运行在 Java 虚拟机之上。`.class` 文件的二进制格式可以使用 [WinHex] 查看。



可以说 `.class` 文件是不同的语言在 Java 虚拟机之间的重要桥梁，同时也是支持 Java 跨平台很重要的一个原因。

4.2 Class 文件结构总结

根据 Java 虚拟机规范，类文件由单个 ClassFile 结构组成：

```
classFile {  
    u4 magic; //Class 文件的标志
```

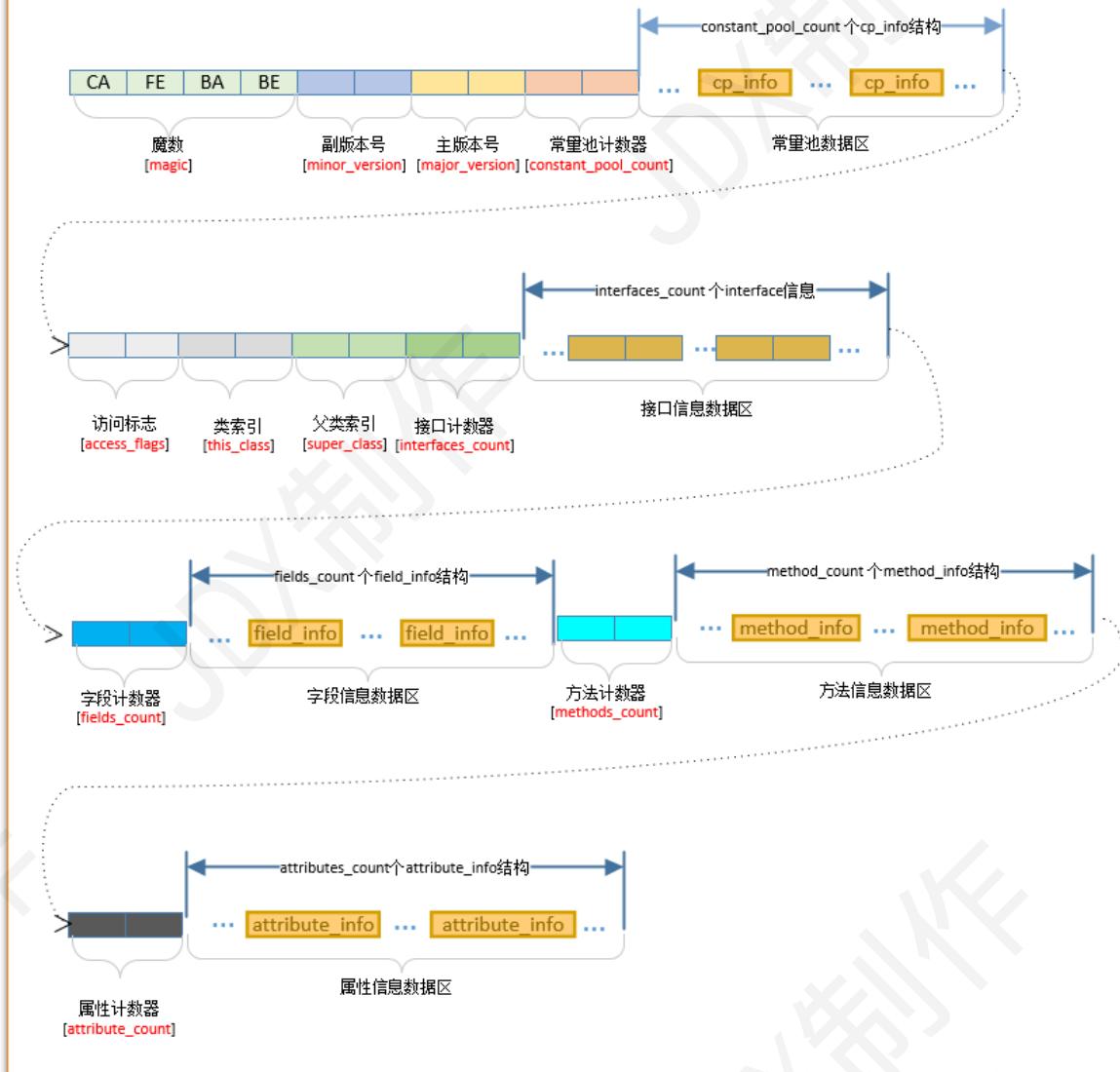
```
    u2          minor_version;//class 的小版本号
    u2          major_version;//class 的大版本号
    u2          constant_pool_count;//常量池的数量
    cp_info     constant_pool[constant_pool_count-1];//常量池
    u2          access_flags;//class 的访问标记
    u2          this_class;//当前类
    u2          super_class;//父类
    u2          interfaces_count;//接口
    u2          interfaces[interfaces_count];//一个类可以实现多个接口
    u2          fields_count;//class 文件的字段属性
    field_info   fields[fields_count];//一个类会可以有个字段
    u2          methods_count;//class 文件的方法数量
    method_info  methods[methods_count];//一个类可以有个多个方法
    u2          attributes_count;//此类的属性表中的属性数
    attribute_info attributes[attributes_count];//属性表集合
}
}
```

下面详细介绍一下 Class 文件结构涉及到的一些组件。

Class文件字节码结构组织示意图 (之前在网上保存的，非常不错，原出处不明) :

Class文件字节码结构组织示意图

注：被编译器编译成.class字节码文件的字节流以及其组织结构如下所示：



4.2.1 魔数

u4	magic; //Class 文件的标志
----	----------------------

每个 Class 文件的头四个字节称为魔数（Magic Number），它的唯一作用是确定这个文件是否为一个能被虚拟机接收的 Class 文件。

程序设计者很多时候都喜欢用一些特殊的数字表示固定的文件类型或者其它特殊的含义。

4.2.2 Class 文件版本

u2	minor_version; //Class 的小版本号
u2	major_version; //Class 的大版本号

紧接着魔数的四个字节存储的是 Class 文件的版本号：第五和第六是次版本号，第七和第八是主版本号。

高版本的 Java 虚拟机可以执行低版本编译器生成的 Class 文件，但是低版本的 Java 虚拟机不能执行高版本编译器生成的 Class 文件。所以，我们在实际开发的时候要确保开发的的 JDK 版本和生产环境的 JDK 版本保持一致。

4.2.3 常量池

```
u2 constant_pool_count; //常量池的数量  
cp_info constant_pool[constant_pool_count-1]; //常量池
```

紧接着主次版本号之后的是常量池，常量池的数量是 constant_pool_count-1（常量池计数器是从1开始计数的，将第0项常量空出来是有特殊考虑的，索引值为0代表“不引用任何一个常量池项”）。

常量池主要存放两大常量：字面量和符号引用。字面量比较接近于 Java 语言层面的的常量概念，如文本字符串、声明为 final 的常量值等。而符号引用则属于编译原理方面的概念。包括下面三类常量：

- 类和接口的全限定名
- 字段的名称和描述符
- 方法的名称和描述符

常量池中每一项常量都是一个表，这14种表有一个共同的特点：**开始的第一位是一个 u1 类型的标志位 -tag 来标识常量的类型，代表当前这个常量属于哪种常量类型。**

类型	标志 (tag)	描述
CONSTANT_utf8_info	1	UTF-8编码的字符串
CONSTANT_Integer_info	3	整形字面量
CONSTANT_Float_info	4	浮点型字面量
CONSTANT_Long_info	5	长整型字面量
CONSTANT_Double_info	6	双精度浮点型字面量
CONSTANT_Class_info	7	类或接口的符号引用
CONSTANT_String_info	8	字符串类型字面量
CONSTANT_Fieldref_info	9	字段的符号引用
CONSTANT_Methodref_info	10	类中方法的符号引用
CONSTANT_InterfaceMethodref_info	11	接口中方法的符号引用
CONSTANT_NameAndType_info	12	字段或方法的符号引用

类型	标志 (tag)	描述
CONSTANT_MethodType_info	16	标志方法类型
CONSTANT_MethodHandle_info	15	表示方法句柄
CONSTANT_InvokeDynamic_info	18	表示一个动态方法调用点

.class 文件可以通过 javap -v class类名 指令来看一下其常量池中的信息(javap -v class类名-> temp.txt : 将结果输出到 temp.txt 文件)。

4.2.4 访问标志

在常量池结束之后，紧接着的两个字节代表访问标志，这个标志用于识别一些类或者接口层次的访问信息，包括：这个 Class 是类还是接口，是否为 public 或者 abstract 类型，如果是类的话是否声明为 final 等等。

类访问和属性修饰符：

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared public; may be accessed from outside its package.
ACC_FINAL	0x0010	Declared final; no subclasses allowed.
ACC_SUPER	0x0020	Treat superclass methods specially when invoked by the invokespecial instruction.
ACC_INTERFACE	0x0200	Is an interface, not a class.
ACC_ABSTRACT	0x0400	Declared abstract; must not be instantiated.
ACC_SYNTHETIC	0x1000	Declared synthetic; not present in the source code.
ACC_ANNOTATION	0x2000	Declared as an annotation type.
ACC_ENUM	0x4000	Declared as an enum type.

我们定义了一个 Employee 类

```
package top.snailclimb.bean;
public class Employee {
    ...
}
```

通过 javap -v class类名 指令来看一下类的访问标志。

```
$ javap -v Employee
  ...
  Flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
#1 = Methodref      #17.#46          // java/lang/Object."<init>":()V
#2 = Fieldref        #16.#47          // top/snailclimb/bean/Employee.id:I
#3 = Fieldref        #16.#48          // top/snailclimb/bean/Employee.name:Ljava/lang/String;
;
#4 = Fieldref        #16.#49          // top/snailclimb/bean/Employee.age:I
#5 = Fieldref        #16.#50          // top/snailclimb/bean/Employee.gender:Ljava/lang/String;
```

访问标志

4.2.5 当前类索引,父类索引与接口索引集合

```
u2          this_class;//当前类
u2          super_class;//父类
u2          interfaces_count;//接口
u2          interfaces[interfaces_count];//一个类可以实现多个接口
```

类索引用于确定这个类的全限定名，父类索引用于确定这个类的父类的全限定名，由于 Java 语言的单继承，所以父类索引只有一个，除了 `java.lang.Object` 之外，所有的 java 类都有父类，因此除了 `java.lang.Object` 外，所有 Java 类的父类索引都不为 0。

接口索引集合用来描述这个类实现了那些接口，这些被实现的接口将按 `implements` (如果这个类本身是接口的话则是 `extends`) 后的接口顺序从左到右排列在接口索引集合中。

4.2.6 字段表集合

```
u2          fields_count;//class 文件的字段的个数
field_info  fields[fields_count];//一个类会可以有个字段
```

字段表 (field info) 用于描述接口或类中声明的变量。字段包括类级变量以及实例变量，但不包括在方法内部声明的局部变量。

`field info(字段表)` 的结构:

```
field_info {
    u2          access_flags;
    u2          name_index;
    u2          descriptor_index;
    u2          attributes_count;
    attribute_info attributes[attributes_count];
}
```

- **access_flags:** 字段的作用域 (`public`,`private`,`protected` 修饰符)，是实例变量还是类变量 (`static` 修饰符), 可否被序列化 (`transient` 修饰符), 可变性 (`final`), 可见性 (`volatile` 修饰符, 是否强制从主内存读写)。
- **name_index:** 对常量池的引用，表示的字段的名称；
- **descriptor_index:** 对常量池的引用，表示字段和方法的描述符；
- **attributes_count:** 一个字段还会拥有一些额外的属性，`attributes_count` 存放属性的个数；
- **attributes[attributes_count]:** 存放具体属性具体内容。

上述这些信息中，各个修饰符都是布尔值，要么有某个修饰符，要么没有，很适合使用标志位来表示。而字段叫什么名字、字段被定义为什么数据类型这些都是无法固定的，只能引用常量池中常量来描述。

`字段的 access_flags 的取值:`

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared <code>public</code> ; may be accessed from outside its package.
ACC_PRIVATE	0x0002	Declared <code>private</code> ; usable only within the defining class.
ACC_PROTECTED	0x0004	Declared <code>protected</code> ; may be accessed within subclasses.
ACC_STATIC	0x0008	Declared <code>static</code> .
ACC_FINAL	0x0010	Declared <code>final</code> ; never directly assigned to after object construction (JLS §17.5).
ACC_VOLATILE	0x0040	Declared <code>volatile</code> ; cannot be cached.
ACC_TRANSIENT	0x0080	Declared <code>transient</code> ; not written or read by a persistent object manager.
ACC_SYNTHETIC	0x1000	Declared synthetic; not present in the source code.
ACC_ENUM	0x4000	Declared as an element of an <code>enum</code> .

4.2.7 方法表集合

```

u2           methods_count; //Class 文件的方法的数量
method_info   methods[methods_count]; //一个类可以有个多个方法

```

methods_count 表示方法的数量，而 method_info 表示的方法表。

Class 文件存储格式中对方法的描述与对字段的描述几乎采用了完全一致的方式。方法表的结构如同字段表一样，依次包括了访问标志、名称索引、描述符索引、属性表集合几项。

method_info(方法表的) 结构：

```

method_info {
    u2           access_flags;
    u2           name_index;
    u2           descriptor_index;
    u2           attributes_count;
    attribute_info attributes[attributes_count];
}

```

方法表的 access_flag 取值：

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared <code>public</code> ; may be accessed from outside its package.
ACC_PRIVATE	0x0002	Declared <code>private</code> ; usable only within the defining class.
ACC_PROTECTED	0x0004	Declared <code>protected</code> ; may be accessed within subclasses.
ACC_STATIC	0x0008	Declared <code>static</code> .
ACC_FINAL	0x0010	Declared <code>final</code> ; never directly assigned to after object construction (JLS §17.5).
ACC_VOLATILE	0x0040	Declared <code>volatile</code> ; cannot be cached.
ACC_TRANSIENT	0x0080	Declared <code>transient</code> ; not written or read by a persistent object manager.
ACC_SYNTHETIC	0x1000	Declared synthetic; not present in the source code.
ACC_ENUM	0x4000	Declared as an element of an <code>enum</code> .

注意：因为 `volatile` 修饰符和 `transient` 修饰符不可以修饰方法，所以方法表的访问标志中没有这两个对应的标志，但是增加了 `synchronized`、`native`、`abstract` 等关键字修饰方法，所以也就多了这些关键字对应的标志。

4.2.8 属性表集合

```
u2 attributes_count; //此类的属性表中的属性数
attribute_info attributes[attributes_count]; //属性表集合
```

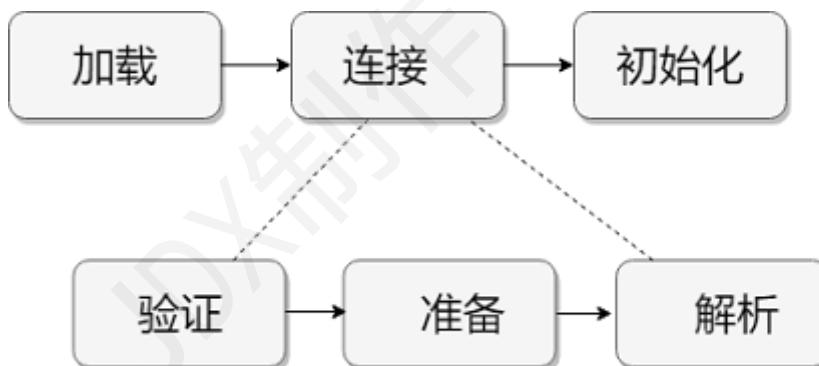
在 Class 文件，字段表，方法表中都可以携带自己的属性表集合，以用于描述某些场景专有的信息。与 Class 文件中其它的数据项目要求的顺序、长度和内容不同，属性表集合的限制稍微宽松一些，不再要求各个属性表具有严格的顺序，并且只要不与已有的属性名重复，任何人实现的编译器都可以向属性表中写入自己定义的属性信息，Java 虚拟机运行时会忽略掉它不认识的属性。

5. 类加载过程

5.1 类加载过程

Class 文件需要加载到虚拟机中之后才能运行和使用，那么虚拟机是如何加载这些 Class 文件呢？

系统加载 Class 类型的文件主要三步：**加载->连接->初始化**。连接过程又可分为三步：**验证->准备->解析**。



5.1.1 加载

类加载过程的第一步，主要完成下面3件事情：

1. 通过全类名获取定义此类的二进制字节流
2. 将字节流所代表的静态存储结构转换为方法区的运行时数据结构
3. 在内存中生成一个代表该类的 Class 对象,作为方法区这些数据的访问入口

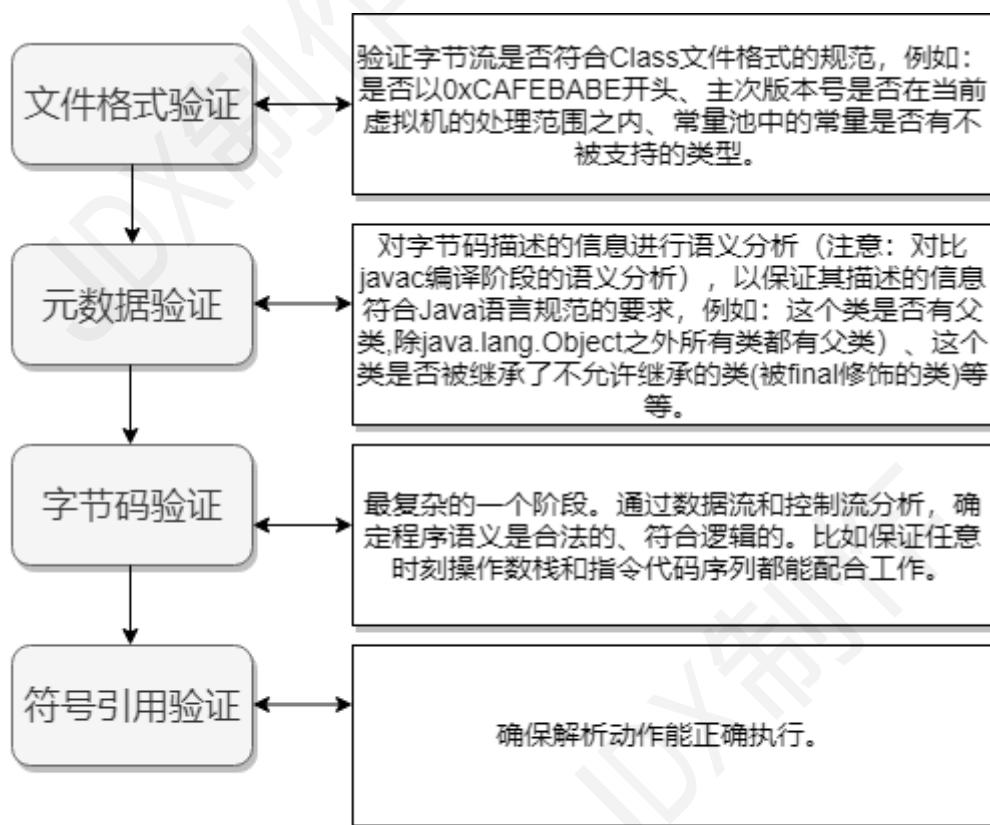
虚拟机规范多上面这3点并不具体，因此是非常灵活的。比如：“通过全类名获取定义此类的二进制字节流”并没有指明具体从哪里获取、怎样获取。比如：比较常见的就是从 ZIP 包中读取（日后出现的 JAR、EAR、WAR 格式的基础）、其他文件生成（典型应用就是 JSP）等等。

一个非数组类的加载阶段（加载阶段获取类的二进制字节流的动作）是可控性最强的阶段，这一步我们可以去完成还可以自定义类加载器去控制字节流的获取方式（重写一个类加载器的 `loadClass()` 方法）。数组类型不通过类加载器创建，它由 Java 虚拟机直接创建。

类加载器、双亲委派模型也是非常重要的知识点，这部分内容会在后面的文章中单独介绍到。

加载阶段和连接阶段的部分内容是交叉进行的，加载阶段尚未结束，连接阶段可能就已经开始了。

5.1.2 验证



5.1.3 准备

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中分配。对于该阶段有以下几点需要注意：

1. 这时候进行内存分配的仅包括类变量（static），而不包括实例变量，实例变量会在对象实例化时随着对象一块分配在 Java 堆中。
2. 这里所设置的初始值“通常情况”下是数据类型默认的零值（如0、0L、null、false等），比如我们定义了 `public static int value=111`，那么 `value` 变量在准备阶段的初始值就是 0 而不是 111（初始化阶段才会赋值）。特殊情况：比如给 `value` 变量加上了 `final` 关键字 `public static final int value=111`，那么准备阶段 `value` 的值就被赋值为 111。

基本数据类型的零值：

数据类型	零 值	数据类型	零 值
int	0	boolean	false
long	0L	float	0.0f
short	(short) 0	double	0.0d
char	'\u0000'	reference	null
byte	(byte) 0		

5.1.4 解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用限定符7类符号引用进行。

符号引用就是一组符号来描述目标，可以是任何字面量。**直接引用**就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。在程序实际运行时，只有符号引用是不够的，举个例子：在程序执行方法时，系统需要明确知道这个方法所在的位置。Java 虚拟机为每个类都准备了一张方法表来存放类中所有的方法。当需要调用一个类的方法的时候，只要知道这个方法在方法表中的偏移量就可以直接调用该方法了。通过解析操作符号引用就可以直接转变为方法在类中方法表的位置，从而使得方法可以被调用。

综上，解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程，也就是得到类或者字段、方法在内存中的指针或者偏移量。

5.1.5 初始化

初始化是类加载的最后一步，也是真正执行类中定义的 Java 程序代码(字节码)，初始化阶段是执行类构造器 `<clinit>()` 方法的过程。

对于 `<clinit>()` 方法的调用，虚拟机会自己确保其在多线程环境中的安全性。因为 `<clinit>()` 方法是带锁线程安全，所以在多线程环境下进行类初始化的话可能会引起死锁，并且这种死锁很难被发现。

对于初始化阶段，虚拟机严格规范了有且只有5种情况下，必须对类进行初始化(只有主动去使用类才会初始化类)：

1. 当遇到 new、getstatic、putstatic 或 invokestatic 这4条直接码指令时，比如 new 一个类，读取一个静态字段(未被 final 修饰)、或调用一个类的静态方法时。
 - 当jvm执行new指令时会初始化类。即当程序创建一个类的实例对象。
 - 当jvm执行getstatic指令时会初始化类。即程序访问类的静态变量(不是静态常量，常量会被加载到运行时常量池)。
 - 当jvm执行putstatic指令时会初始化类。即程序给类的静态变量赋值。
 - 当jvm执行invokestatic指令时会初始化类。即程序调用类的静态方法。
2. 使用 `java.lang.reflect` 包的方法对类进行反射调用时如 `Class.forName("...")`, `newInstance()` 等等。如果类没初始化，需要触发其初始化。
3. 初始化一个类，如果其父类还未初始化，则先触发该父类的初始化。
4. 当虚拟机启动时，用户需要定义一个要执行的主类(包含 main 方法的那个类)，虚拟机会先初始化这个类。
5. MethodHandle 和 VarHandle 可以看作是轻量级的反射调用机制，而要想使用这2个调用，就必须先使用 `findStaticVarHandle` 来初始化要调用的类。
6. 当一个接口中定义了JDK8新加入的默认方法(被 default 关键字修饰的接口方法)时，如果有这个接口的实现类发生了初始化，那该接口要在其之前被初始化。

5.2 卸载

卸载类即该类的 Class 对象被 GC。

卸载类需要满足3个要求:

1. 该类的所有的实例对象都已被GC, 也就是说堆不存在该类的实例对象。
2. 该类没有在其他任何地方被引用
3. 该类的类加载器的实例已被GC

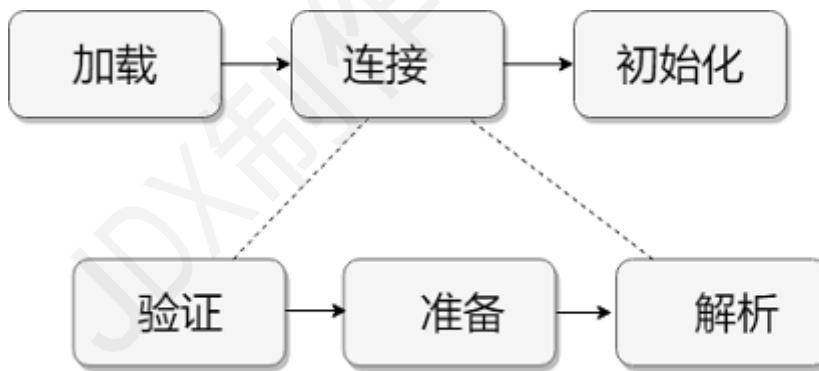
所以, 在JVM生命周期类, 由jvm自带的类加载器加载的类是不会被卸载的。但是由我们自定义的类加载器加载的类是可能被卸载的。

只要想通一点就好了, jdk自带的BootstrapClassLoader,PlatformClassLoader,AppClassLoader负责加载jdk提供的类, 所以它们(类加载器的实例)肯定不会被回收。而我们自定义的类加载器的实例是可以被回收的, 所以使用我们自定义加载器加载的类是可以被卸载掉的。

6. 类加载器

6.1 回顾一下类加载过程

类加载过程: 加载->连接->初始化。连接过程又可分为三步:验证->准备->解析。



一个非数组类的加载阶段 (加载阶段获取类的二进制字节流的动作) 是可控性最强的阶段, 这一步我们可以去完成还可以自定义类加载器去控制字节流的获取方式 (重写一个类加载器的 `loadClass()` 方法)。数组类型不通过类加载器创建, 它由 Java 虚拟机直接创建。

所有的类都由类加载器加载, 加载的作用就是将 .class 文件加载到内存。

6.2 类加载器总结

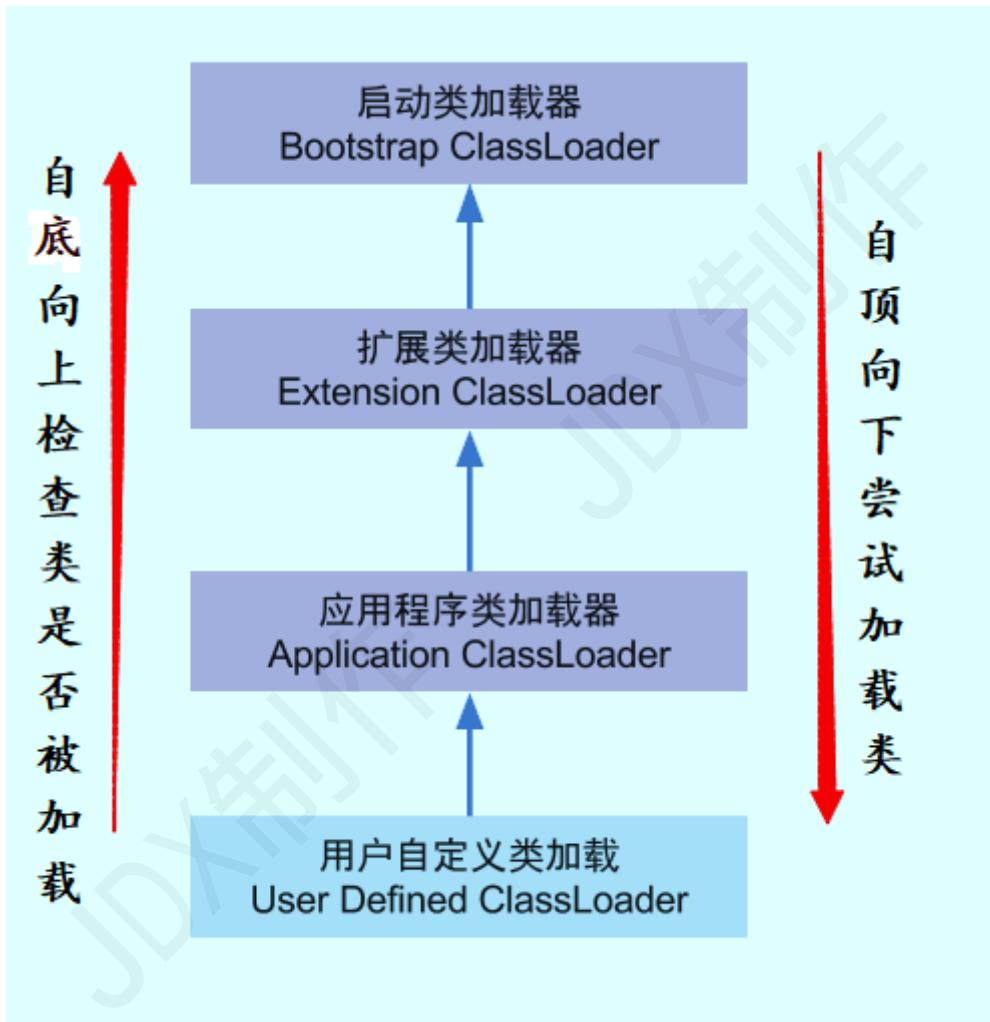
JVM 中内置了三个重要的 ClassLoader, 除了 BootstrapClassLoader 其他类加载器均由 Java 实现且全部继承自 `java.lang.ClassLoader`:

1. **BootstrapClassLoader(启动类加载器)**: 最顶层的加载类, 由C++实现, 负责加载 `%JAVA_HOME%/lib` 目录下的jar包和类或者或被 `-xbootclasspath` 参数指定的路径中的所有类。
2. **ExtensionClassLoader(扩展类加载器)**: 主要负责加载目录 `%JRE_HOME%/lib/ext` 目录下的jar包和类, 或被 `java.ext.dirs` 系统变量所指定的路径下的jar包。
3. **AppClassLoader(应用程序类加载器)**: 面向我们用户的加载器, 负责加载当前应用classpath下的所有jar包和类。

6.3 双亲委派模型

6.3.1 双亲委派模型介绍

每一个类都有一个对应它的类加载器。系统中的 ClassLoder 在协同工作的时候会默认使用 **双亲委派模型**。即在类加载的时候, 系统会首先判断当前类是否被加载过。已经被加载的类会直接返回, 否则才会尝试加载。加载的时候, 首先会把该请求委派该父类加载器的 `loadClass()` 处理, 因此所有的请求最终都应该传送到顶层的启动类加载器 `BootstrapClassLoader` 中。当父类加载器无法处理时, 才由自己来处理。当父类加载器为null时, 会使用启动类加载器 `BootstrapClassLoader` 作为父类加载器。



每个类加载都有一个父类加载器，我们通过下面的程序来验证。

```
public class ClassLoaderDemo {
    public static void main(String[] args) {
        System.out.println("ClassLodarDemo's ClassLoader is " +
classLoaderDemo.class.getClassLoader());
        System.out.println("The Parent of ClassLodarDemo's ClassLoader is " +
classLoaderDemo.class.getClassLoader().getParent());
        System.out.println("The GrandParent of ClassLodarDemo's ClassLoader is " +
+ classLoaderDemo.class.getClassLoader().getParent().getParent());
    }
}
```

Output

```
ClassLodarDemo's ClassLoader is sun.misc.Launcher$AppClassLoader@18b4aac2
The Parent of ClassLodarDemo's ClassLoader is
sun.misc.Launcher$ExtClassLoader@1b6d3586
The GrandParent of ClassLodarDemo's ClassLoader is null
```

AppClassLoader 的父类加载器为 ExtClassLoader

ExtClassLoader 的父类加载器为 null，null 并不代表 ExtClassLoader 没有父类加载器，而是 BootstrapClassLoader。

其实这个双亲翻译的容易让别人误解，我们一般理解的双亲都是父母，这里的双亲更多地表达的是“父母这一辈”的人而已，并不是说真的有一个 Mother ClassLoader 和一个 Father ClassLoader。另外，类加载器之间的“父子”关系也不是通过继承来体现的，是由“优先级”来决定。官方API文档对这部分的描述如下：

The Java platform uses a delegation model for loading classes. **The basic idea is that every class loader has a "parent" class loader.** When loading a class, a class loader first "delegates" the search for the class to its parent class loader before attempting to find the class itself.

6.3.2 双亲委派模型实现源码分析

双亲委派模型的实现代码非常简单，逻辑非常清晰，都集中在 `java.lang.ClassLoader` 的 `loadClass()` 中，相关代码如下所示。

```
private final ClassLoader parent;
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
    synchronized (getClassLoadingLock(name)) {
        // 首先，检查请求的类是否已经被加载过
        Class<?> c = findLoadedClass(name);
        if (c == null) {
            long t0 = System.nanoTime();
            try {
                if (parent != null) { //父加载器不为空，调用父加载器loadClass()方法处理
                    c = parent.loadClass(name, false);
                } else { //父加载器为空，使用启动类加载器 BootstrapClassLoader 加载
                    c = findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
                //抛出异常说明父类加载器无法完成加载请求
            }

            if (c == null) {
                long t1 = System.nanoTime();
                //自己尝试加载
                c = findClass(name);

                // this is the defining class loader; record the stats
                sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);

                sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
                sun.misc.PerfCounter.getFindClasses().increment();
            }
        }
        if (resolve) {
            resolveClass(c);
        }
        return c;
    }
}
```

6.3.3 双亲委派模型的好处

双亲委派模型保证了Java程序的稳定运行，可以避免类的重复加载（JVM 区分不同类型的方式不仅仅根据类名，相同的类文件被不同的类加载器加载产生的是两个不同的类），也保证了 Java 的核心 API 不被篡改。如果没有使用双亲委派模型，而是每个类加载器加载自己的话就会出现一些问题，比如我们编写一个称为 `java.lang.Object` 类的话，那么程序运行的时候，系统就会出现多个不同的 `Object` 类。

6.3.4 如果我们不想用双亲委派模型怎么办？

为了避免双亲委托机制，我们可以自己定义一个类加载器，然后重写 `loadClass()` 即可。

6.4 自定义类加载器

除了 `BootstrapClassLoader` 其他类加载器均由 Java 实现且全部继承自 `java.lang.ClassLoader`。如果我们要自定义自己的类加载器，很明显需要继承 `ClassLoader`。

二、网络

(一). 计算机网络知识

1. 计算机概述

1.1 基本术语

结点 (node)：网络中的结点可以是计算机，集线器，交换机或路由器等。

链路 (link)：从一个结点到另一个结点的一段物理线路。中间没有任何其他交点。

主机 (host)：连接在因特网上的计算机。

ISP (Internet Service Provider)：因特网服务提供者（提供商）

IXP (Internet eXchange Point)：互联网交换点IXP的主要作用就是允许两个网络直接相连并交换分组，而不需要再通过第三个网络来转发分组。

RFC(Request For Comments)：意思是“请求评议”，包含了关于Internet几乎所有的重要的文字资料。

广域网WAN (Wide Area Network)：任务是通过长距离运送主机发送的数据

城域网MAN (Metropolitan Area Network)：用来将多个局域网进行互连

局域网LAN (Local Area Network)：学校或企业大多拥有多个互连的局域网

个人区域网PAN (Personal Area Network)：在个人工作的地方把属于个人使用的电子设备用无线技术连接起来的网络

端系统 (end system)：处在因特网边缘的部分即是连接在因特网上的所有的主机

分组 (packet)：因特网中传送的数据单元。由首部header和数据段组成。分组又称为包，首部可称为包头。

存储转发 (store and forward)：路由器收到一个分组，先存储下来，再检查其首部，查找转发表，按照首部中的目的地址，找到合适的接口转发出去。

带宽 (bandwidth)：在计算机网络中，表示在单位时间内从网络中的某一点到另一点所能通过的“最高数据率”。常用来表示网络的通信线路所能传送数据的能力。单位是“比特每秒”，记为b/s。

吞吐量 (throughput)：表示在单位时间内通过某个网络（或信道、接口）的数据量。吞吐量更经常地用于对现实世界中的网络的一种测量，以便知道实际上到底有多少数据量能够通过网络。吞吐量受网络的带宽或网络的额定速率的限制。

1.2 重要知识点总结

1, 计算机网络 (简称网络) 把许多计算机连接在一起, 而互联网把许多网络连接在一起, 是网络的网络。

2, 小写字母i开头的internet (互联网) 是通用名词, 它泛指由多个计算机网络相互连接而成的网络。在这些网络之间的通信协议 (即通信规则) 可以是任意的。

大写字母I开头的Internet (互联网) 是专用名词, 它指全球最大的, 开放的, 由众多网络相互连接而成的特定的互联网, 并采用TCP/IP协议作为通信规则, 其前身为ARPANET。Internet的推荐译名为因特网, 现在一般流行称为互联网。

3, 路由器是实现分组交换的关键构件, 其任务是转发收到的分组, 这是网络核心部分最重要的功能。分组交换采用存储转发技术, 表示把一个报文 (要发送的整块数据) 分为几个分组后再进行传送。在发送报文之前, 先把较长的报文划分成为一个个更小的等长数据段。在每个数据端的前面加上一些由必要的控制信息组成的首部后, 就构成了一个分组。分组又称为包。分组是在互联网中传送的数据单元, 正是由于分组的头部包含了诸如目的地址和源地址等重要控制信息, 每一个分组才能在互联网中独立的选择传输路径, 并正确地交付到分组传输的终点。

4, 互联网按工作方式可划分为边缘部分和核心部分。主机在网络的边缘部分, 其作用是进行信息处理。由大量网络和连接这些网络的路由器组成核心部分, 其作用是提供连通性和交换。

5, 计算机通信是计算机中进程 (即运行着的程序) 之间的通信。计算机网络采用的通信方式是客户-服务器方式 (C/S方式) 和对等连接方式 (P2P方式) 。

6, 客户和服务器都是指通信中所涉及的应用进程。客户是服务请求方, 服务器是服务提供方。

7, 按照作用范围的不同, 计算机网络分为广域网WAN, 城域网MAN, 局域网LAN, 个人区域网PAN。

8, 计算机网络最常用的性能指标是: 速率, 带宽, 吞吐量, 时延 (发送时延, 处理时延, 排队时延), 时延带宽积, 往返时间和信道利用率。

9, 网络协议即协议, 是为进行网络中的数据交换而建立的规则。计算机网络的各层以及其协议集合, 称为网络的体系结构。

10, 五层体系结构由应用层, 运输层, 网络层 (网际层) , 数据链路层, 物理层组成。运输层最主要的协议是TCP和UDP协议, 网络层最重要的协议是IP协议。

2. 物理层

2.1 基本术语

数据 (data) : 运送消息的实体。

信号 (signal) : 数据的电气的或电磁的表现。或者说信号是适合在传输介质上传输的对象。

码元 (code) : 在使用时间域 (或简称为时域) 的波形来表示数字信号时, 代表不同离散数值的基本波形。

单工 (simplex) : 只能有一个方向的通信而没有反方向的交互。

半双工 (half duplex) : 通信的双方都可以发送信息, 但不能双方同时发送(当然也就不能同时接收)。

全双工 (full duplex) : 通信的双方可以同时发送和接收信息。

奈氏准则: 在任何信道中, 码元的传输的效率是有上限的, 传输速率超过此上限, 就会出现严重的码间串扰问题, 使接收端对码元的判决 (即识别) 成为不可能。

基带信号 (baseband signal) : 来自信源的信号。指没有经过调制的数字信号或模拟信号。

带通（频带）信号（bandpass signal）：把基带信号经过载波调制后，把信号的频率范围搬移到较高的频段以便在信道中传输（即仅在一段频率范围内能够通过信道），这里调制过后的信号就是带通信号。

调制（modulation）：对信号源的信息进行处理后加到载波信号上，使其变为适合在信道传输的形式的过程。

信噪比（signal-to-noise ratio）：指信号的平均功率和噪声的平均功率之比，记为S/N。信噪比 (dB) = $10 \times \log_{10} (S/N)$

信道复用（channel multiplexing）：指多个用户共享同一个信道。（并不一定是同时）

比特率（bit rate）：单位时间（每秒）内传送的比特数。

波特率（baud rate）：单位时间载波调制状态改变的次数。针对数据信号对载波的调制速率。

复用（multiplexing）：共享信道的方法

ADSL（Asymmetric Digital Subscriber Line）：非对称数字用户线。

光纤同轴混合网（HFC网）：在目前覆盖范围很广的有线电视网的基础上开发的一种居民宽带接入网

2.2 重要知识点总结

1，物理层的主要任务就是确定与传输媒体接口有关的一些特性，如机械特性，电气特性，功能特性，过程特性。

2，一个数据通信系统可划分为三大部分，即源系统，传输系统，目的系统。源系统包括源点（或源站，信源）和发送器，目的系统包括接收器和终点。

3，通信的目的是传送消息。如话音，文字，图像等都是消息，数据是运送消息的实体。信号则是数据的电器或电磁的表现。

4，根据信号中代表消息的参数的取值方式不同，信号可分为模拟信号（或连续信号）和数字信号（或离散信号）。在使用时间域（简称时域）的波形表示数字信号时，代表不同离散数值的基本波形称为码元。

5，根据双方信息交互的方式，通信可划分为单向通信（或单工通信），双向交替通信（或半双工通信），双向同时通信（全双工通信）。

6，来自信源的信号称为基带信号。信号要在信道上传输就要经过调制。调制有基带调制和带通调制之分。最基本的带通调制方法有调幅，调频和调相。还有更复杂的调制方法，如正交振幅调制。

7，要提高数据在信道上的传递速率，可以使用更好的传输媒体，或使用先进的调制技术。但数据传输速率不可能任意被提高。

8，传输媒体可分为两大类，即导引型传输媒体（双绞线，同轴电缆，光纤）和非导引型传输媒体（无线，红外，大气激光）。

9，为了有效利用光纤资源，在光纤干线和用户之间广泛使用无源光网络PON。无源光网络无需配备电源，其长期运营成本和管理成本都很低。最流行的无源光网络是以太网无源光网络EPON和吉比特无源光网络GPON。

2.3 最重要的知识点

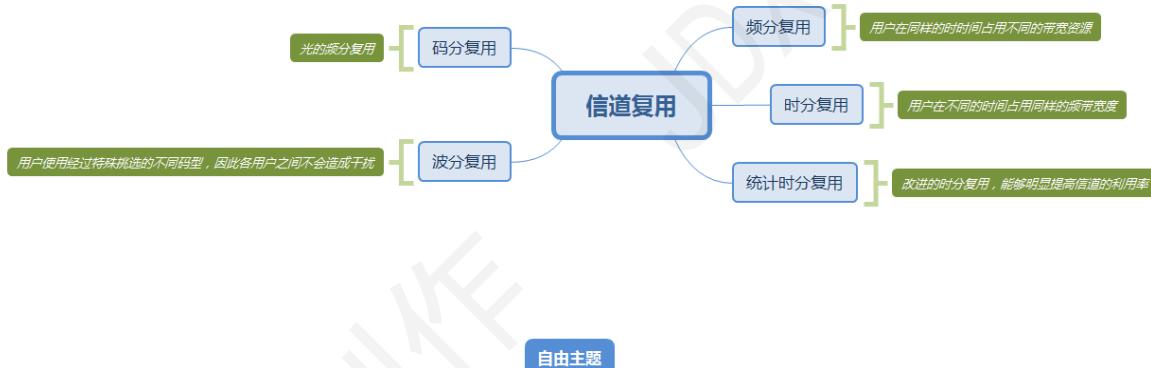
①，物理层的任务

透明地传送比特流。也可以将物理层的主要任务描述为确定与传输媒体的接口的一些特性，即：机械特性（接口所用接线器的一些物理属性如形状尺寸），电气特性（接口电缆的各条线上出现的电压的范围），功能特性（某条线上出现的某一电平的电压的意义），过程特性（对于不同功能的各种可能事件的出现顺序）。

2.3.1 拓展：

物理层考虑的是怎样才能在连接各种计算机的传输媒体上传输数据比特流，而不是指具体的传输媒体。现有的计算机网络中的硬件设备和传输媒体的种类非常繁多，而且通信手段也有许多不同的方式。物理层的作用正是尽可能地屏蔽掉这些传输媒体和通信手段的差异，使物理层上面的数据链路层感觉不到这些差异，这样就可以使数据链路层只考虑完成本层的协议和服务，而不必考虑网络的具体传输媒体和通信手段是什么。

2.3.2 几种常用的信道复用技术



2.3.3 几种常用的宽带接入技术，主要是ADSL和FTTx

用户到互联网的宽带接入方法有非对称数字用户线ADSL（用数字技术对现有的模拟电话线进行改造，而不需要重新布线。ADSL的快速版本是甚高速数字用户线VDSL。），光纤同轴混合网HFC（是在目前覆盖范围很广的有线电视网的基础上开发的一种居民宽带接入网）和FTTx（即光纤到……）

3. 数据链路层

3.1 基本术语

链路 (link)：一个结点到相邻结点的一段物理链路

数据链路 (data link)：把实现控制数据运输的协议的硬件和软件加到链路上就构成了数据链路

循环冗余检验CRC (Cyclic Redundancy Check)：为了保证数据传输的可靠性，CRC是数据链路层广泛使用的一种检错技术

帧 (frame)：一个数据链路层的传输单元，由一个数据链路层首部和其携带的封包所组成协议数据单元。

MTU (Maximum Transfer Unit)：最大传送单元。帧的数据部分的的长度上限。

误码率BER (Bit Error Rate)：在一段时间内，传输错误的比特占所传输比特总数的比率。

小插曲：

更多阿里、腾讯、美团、京东等一线互联网大厂Java面试真题；包含：基础、并发、锁、JVM、设计模式、数据结构、反射/IO、数据库、Redis、Spring、消息队列、分布式、Zookeeper、Dubbo、Mybatis、Maven、面经等。

更多Java程序员技术进阶小技巧；例如高效学习（如何学习和阅读代码、面对枯燥和量大的知识）高效沟通（沟通方式及技巧、沟通技术）

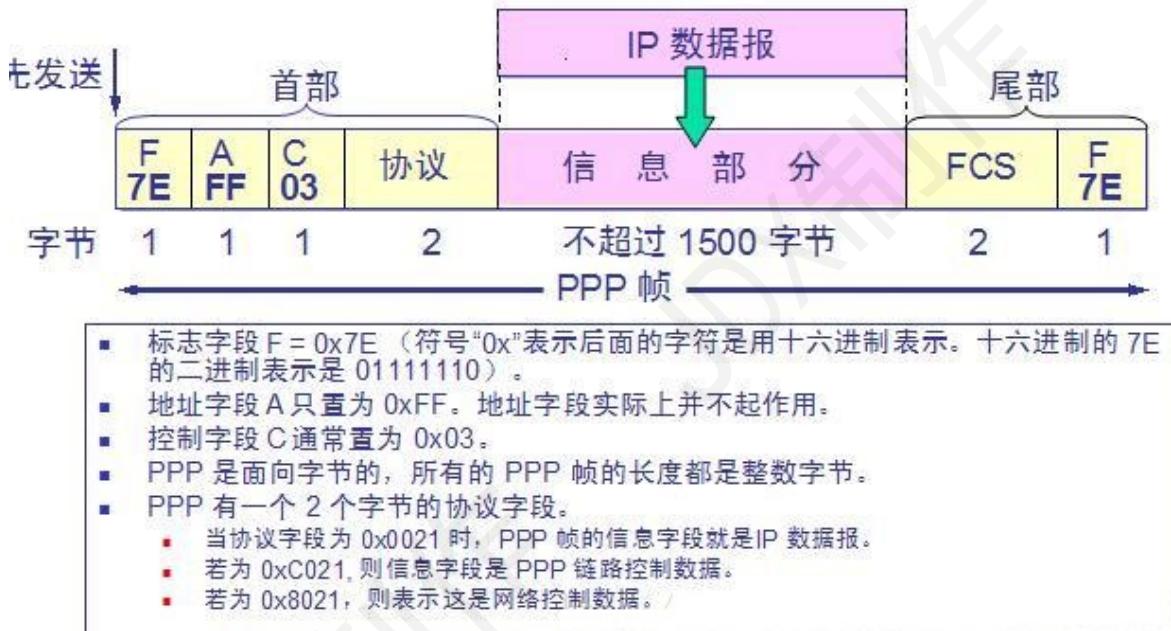
更多Java大牛分享的一些职业生涯分享文档

请添加微信：jiang10086a 免费获取

比你优秀的对手在学习，你的仇人在磨刀，你的闺蜜在减肥，隔壁老王在练腰，我们必须不断学习，否则我们将被学习者超越！

趁年轻，使劲拼，给未来的自己一个交代！

PPP (Point-to-Point Protocol) : 点对点协议。即用户计算机和ISP进行通信时所使用的数据链路层协议。以下是PPP帧的示意图:



MAC地址 (Media Access Control或者Medium Access Control) : 意译为媒体访问控制，或称为物理地址、硬件地址，用来定义网络设备的位置。在OSI模型中，第三层网络层负责IP地址，第二层数据链路层则负责MAC地址。因此一个主机会有一个MAC地址，而每个网络位置会有一个专属于它的IP地址。地址是识别某个系统的重要标识符，“名字指出我们所要寻找的资源，地址指出资源所在的地方，路由告诉我们如何到达该处”

网桥 (bridge) : 一种用于数据链路层实现中继，连接两个或多个局域网的网络互连设备。

交换机 (switch) : 广义的来说，交换机指的是一种通信系统中完成信息交换的设备。这里工作在数据链路层的交换机指的是交换式集线器，其实质是一个多接口的网桥

3.2 重要知识点总结

- 1, 链路是从一个结点到相邻节点的一段物理链路，数据链路则在链路的基础上增加了一些必要的硬件（如网络适配器）和软件（如协议的实现）
- 2, 数据链路层使用的主要有**点对点信道**和**广播信道**两种。
- 3, 数据链路层传输的协议数据单元是帧。数据链路层的三个基本问题是：**封装成帧**，**透明传输**和**差错检测**
- 4, **循环冗余检验CRC**是一种检错方法，而帧检验序列FCS是添加在数据后面的冗余码
- 5, **点对点协议PPP**是数据链路层使用最多的一种协议，它的特点是：简单，只检测差错而不去纠正差错，不使用序号，也不进行流量控制，可同时支持多种网络层协议
- 6, PPPoE是为宽带上网的主机使用的链路层协议
- 7, 局域网的优点是：具有广播功能，从一个站点可方便地访问全网；便于系统的扩展和逐渐演变；提高了系统的可靠性，可用性和生存性。
- 8, 共享媒体通信资源的方法有二：一是静态划分信道(各种复用技术)，而是动态媒体接入控制，又称为多点接入（随即接入或受控接入）
- 9, 计算机与外接局域网通信需要通过通信适配器（或网络适配器），它又称为网络接口卡或网卡。**计算机的硬件地址就在适配器的ROM中。**

10, 以太网采用的无连接的工作方式, 对发送的数据帧不进行编号, 也不要求对方发回确认。目的站收到有差错帧就把它丢掉, 其他什么也不做

11, 以太网采用的协议是具有冲突检测的**载波监听多点接入CSMA/CD**。协议的特点是: **发送前先监听, 边发送边监听, 一旦发现总线上出现了碰撞, 就立即停止发送。然后按照退避算法等待一段随机时间后再次发送。** 因此, 每一个站点在自己发送数据之后的一小段时间内, 存在这遭遇碰撞的可能性。以太网上的各站点平等的争用以太网信道

12, 以太网的适配器具有过滤功能, 它只接收单播帧, 广播帧和多播帧。

13, 使用集线器可以在物理层扩展以太网 (扩展后的以太网仍然是一个网络)

3.3 最重要的知识点

1. 数据链路层的点对点信道和广播信道的特点, 以及这两种信道所使用的协议 (PPP协议以及CSMA/CD协议) 的特点
2. 数据链路层的三个基本问题: **封装成帧, 透明传输, 差错检测**
3. 以太网的MAC层硬件地址
4. 适配器, 转发器, 集线器, 网桥, 以太网交换机的作用以及适用场合

4. 网络层

4.1 基本术语

虚电路 (Virtual Circuit) : 在两个终端设备的逻辑或物理端口之间, 通过建立的双向的透明传输通道。虚电路表示这只是逻辑上的连接, 分组都沿着这条逻辑连接按照存储转发方式传送, 而并不是真正建立了一条物理连接。

IP (Internet Protocol) : 网际协议 IP 是 TCP/IP体系中两个最主要的协议之一, 是TCP/IP体系结构网际层的核心。配套的有ARP, RARP, ICMP, IGMP。



ARP (Address Resolution Protocol) : 地址解析协议

ICMP (Internet Control Message Protocol) : 网际控制报文协议 (ICMP 允许主机或路由器报告差错情况和提供有关异常情况的报告。)

子网掩码 (subnet mask) : 它是一种用来指明一个IP地址的哪些位标识的是主机所在的子网以及哪些位标识的是主机的位掩码。子网掩码不能单独存在, 它必须结合IP地址一起使用。

CIDR (Classless Inter-Domain Routing) : 无分类域间路由选择 (特点是消除了传统的 A 类、B 类和 C 类地址以及划分子网的概念，并使用各种长度的“网络前缀”(network-prefix)来代替分类地址中的网络号和子网号)

默认路由 (default route) : 当在路由表中查不到能到达目的地址的路由时，路由器选择的路由。默认路由还可以减小路由表所占用的空间和搜索路由表所用的时间。

路由选择算法 (Virtual Circuit) : 路由选择协议的核心部分。因特网采用自适应的，分层次的路由选择协议。

4.2 重要知识点总结

1, TCP/IP协议中的网络层向上只提供简单灵活的，无连接的，尽最大努力交付的数据报服务。网络层不提供服务质量的承诺，不保证分组交付的时限所传送的分组可能出错，丢失，重复和失序。进程之间通信的可靠性由运输层负责

2, 在互联网的交付有两种，一是在本网络直接交付不用经过路由器，另一种是和其他网络的间接交付，至少经过一个路由器，但最后一次一定是直接交付

3, 分类的IP地址由网络号字段（指明网络）和主机号字段（指明主机）组成。网络号字段最前面的类别指明IP地址的类别。IP地址是一种分等级的地址结构。IP地址管理机构分配IP地址时只分配网络号，主机号由得到该网络号的单位自行分配。路由器根据目的主机所连接的网络号来转发分组。一个路由器至少连接到两个网络，所以一个路由器至少应当有两个不同的IP地址

4, IP数据报分为首部和数据两部分。首部的前一部分是固定长度，共20字节，是所有IP数据包必须具有的（源地址，目的地址，总长度等重要地段都固定在首部）。一些长度可变的可选字段固定在首部的后面。IP首部中的生存时间给出了IP数据报在互联网中所能经过的最大路由器数。可防止IP数据报在互联网中无限制的兜圈子。

5, 地址解析协议ARP把IP地址解析为硬件地址。ARP的高速缓存可以大大减少网络上的通信量。因为这样可以使主机下次再与同样地址的主机通信时，可以直接从高速缓存中找到所需要的硬件地址而不需要再去广播方式发送ARP请求分组

6, 无分类域间路由选择CIDR是解决目前IP地址紧缺的一个好办法。CIDR记法把IP地址后面加上斜线“/”，然后写上前缀所占的位数。前缀（或网络前缀用来指明网络），前缀后面的部分是后缀，用来指明主机。CIDR把前缀都相同的连续的IP地址组成一个“CIDR地址块”，IP地址分配都以CIDR地址块为单位。

7, 网际控制报文协议是IP层的协议.ICMP报文作为IP数据报的数据，加上首部后组成IP数据报发送出去。使用ICMP数据报并不是为了实现可靠传输。ICMP允许主机或路由器报告差错情况和提供有关异常情况的报告。ICMP报文的种类有两种 ICMP差错报告报文和ICMP询问报文。

8, 要解决IP地址耗尽的问题，最根本的办法是采用具有更大地址空间的新版本IP协议-IPv6。IPv6所带来的变化有①更大的地址空间（采用128位地址）②灵活的首部格式③改进的选项④支持即插即用⑤支持资源的预分配⑥IPv6的首部改为8字节对齐。另外IP数据报的目的地址可以是以下三种基本类型地址之一：单播，多播和任播

9, 虚拟专用网络VPN利用公用的互联网作为本机构专用网之间的通信载体。VPN内使用互联网的专用地址。一个VPN至少要有一个路由器具有合法的全球IP地址，这样才能和本系统的另一个VPN通过互联网进行通信。所有通过互联网传送的数据都需要加密

10, MPLS的特点是：①支持面向连接的服务质量②支持流量工程，平衡网络负载③有效的支持虚拟专用网VPN。MPLS在入口节点给每一个IP数据报打上固定长度的“标记”，然后根据标记在第二层（链路层）用硬件进行转发（在标记交换路由器中进行标记交换），因而转发速率大大加快。

4.3 最重要知识点

1. 虚拟互联网络的概念

2. IP地址和物理地址的关系
3. 传统的分类的IP地址（包括子网掩码）和无分类域间路由选择CIDR
4. 路由选择协议的工作原理

5. 运输层

5.1 基本术语

进程 (process) :

指计算机中正在运行的程序实体

应用进程互相通信:

一台主机的进程和另一台主机中的一个进程交换数据的过程（另外注意通信真正的端点不是主机而是主机中的进程，也就是说端到端的通信是应用进程之间的通信）

传输层的复用与分用:

复用指发送方不同的进程都可以通过统一个运输层协议传送数据。分用指接收方的运输层在剥去报文的首部后能把这些数据正确的交付到目的应用进程。

TCP (Transmission Control Protocol) :

传输控制协议

UDP (User Datagram Protocol) : 用户数据报协议

端口 (port) (link) : 端口的目的是为了确认对方机器是那个进程在于自己进行交互，比如MSN和QQ的端口不同，如果没有端口就可能出现QQ进程和MSN交互错误。端口又称协议端口号。

停止等待协议 (link) : 指发送方每发送完一个分组就停止发送，等待对方确认，在收到确认之后在发送下一个分组。

流量控制 (link) : 就是让发送方的发送速率不要太快，既要让接收方来得及接收，也不要使网络发生拥塞。

拥塞控制 (link) : 防止过多的数据注入到网络中，这样可以使网络中的路由器或链路不致过载。拥塞控制所要做的都有一个前提，就是网络能够承受现有的网络负荷。

5.2 重要知识点总结

1, 运输层提供应用进程之间的逻辑通信，也就是说，运输层之间的通信并不是真正在两个运输层之间直接传输数据。运输层向应用层屏蔽了下面网络的细节（如网络拓扑，所采用的路由选择协议等），它使应用进程之间看起来好像两个运输层实体之间有一条端到端的逻辑通信信道。

2, 网络层为主机提供逻辑通信，而运输层为应用进程之间提供端到端的逻辑通信。

3, 运输层的两个重要协议是用户数据报协议UDP和传输控制协议TCP。按照OSI的术语，两个对等运输实体在通信时传送的数据单位叫做运输协议数据单元TPDU (Transport Protocol Data Unit)。但在TCP/IP体系中，则根据所使用的协议是TCP或UDP，分别称之为TCP报文段或UDP用户数据报。

4, UDP在传送数据之前不需要先建立连接，远地主机在收到UDP报文后，不需要给出任何确认。虽然UDP不提供可靠交付，但在某些情况下UDP确是一种最有效的工作方式。TCP提供面向连接的服务。在传送数据之前必须先建立连接，数据传送结束后要释放连接。TCP不提供广播或多播服务。由于TCP要提供可靠的，面向连接的传输服务，这一难以避免增加了许多开销，如确认，流量控制，计时器以及连接管理等。这不仅使协议数据单元的首部增大很多，还要占用许多处理机资源。

5, 硬件端口是不同硬件设备进行交互的接口，而软件端口是应用层各种协议进程与运输实体进行层间交互的一种地址。UDP和TCP的首部格式中都有源端口和目的端口这两个重要字段。当运输层收到IP层交上来的运输层报文时，就能够根据其首部中的目的端口号把数据交付应用层的目的应用层。（两个进程之间进行通信不光要知道对方IP地址而且要知道对方的端口号(为了找到对方计算机中的应用进程)）

6, 运输层用一个16位端口号标志一个端口。端口号只有本地意义，它只是为了标志计算机应用层中的各个进程在和运输层交互时的层间接口。在互联网的不同计算机中，相同的端口号是没有关联的。协议端口号简称端口。虽然通信的终点是应用进程，但只要把所发送的报文交到目的主机的某个合适端口，剩下的工作（最后交付目的进程）就由TCP和UDP来完成。

7, 运输层的端口号分为服务器端使用的端口号（0~1023指派给熟知端口，1024~49151是登记端口号）和客户端暂时使用的端口号（49152~65535）

8, UDP的主要特点是①无连接②尽最大努力交付③面向报文④无拥塞控制⑤支持一对一，一对多，多对一和多对多的交互通信⑥首部开销小（只有四个字段：源端口，目的端口，长度和检验和）

9, TCP的主要特点是①面向连接②每一条TCP连接只能是一对一的③提供可靠交付④提供全双工通信⑤面向字节流

10, TCP用主机的IP地址加上主机上的端口号作为TCP连接的端点。这样的端点就叫做套接字（socket）或插口。套接字用（IP地址：端口号）来表示。每一条TCP连接唯一被通信两端的两个端点所确定。

11, 停止等待协议是为了实现可靠传输的，它的基本原理就是每发完一个分组就停止发送，等待对方确认。在收到确认后再发下一个分组。

12, 为了提高传输效率，发送方可以不使用低效率的停止等待协议，而是采用流水线传输。流水线传输就是发送方可连续发送多个分组，不必每发完一个分组就停下来等待对方确认。这样可使信道上一直有数据不间断的在传送。这种传输方式可以明显提高信道利用率。

13, 停止等待协议中超时重传是指只要超过一段时间仍然没有收到确认，就重传前面发送过的分组（认为刚才发送过的分组丢失了）。因此每发送完一个分组需要设置一个超时计时器，其重传时间应比数据在分组传输的平均往返时间更长一些。这种自动重传方式常称为自动重传请求ARQ。另外在停止等待协议中若收到重复分组，就丢弃该分组，但同时还要发送确认。连续ARQ协议可提高信道利用率。发送维持一个发送窗口，凡位于发送窗口内的分组可连续发送出去，而不需要等待对方确认。接收方一般采用累积确认，对按序到达的最后一个分组发送确认，表明到这个分组位置的所有分组都已经正确收到了。

14, TCP报文段的前20个字节是固定的，后面有 $4n$ 字节是根据需要增加的选项。因此，TCP首部的最小长度是20字节。

15, TCP使用滑动窗口机制。发送窗口里面的序号表示允许发送的序号。发送窗口后沿的后面部分表示已发送且已收到确认，而发送窗口前沿的前面部分表示不允许发送。发送窗口后沿的变化情况有两种可能，即不动（没有收到新的确认）和前移（收到了新的确认）。发送窗口的前沿通常是不断向前移动的。一般来说，我们总是希望数据传输更快一些。但如果发送方把数据发送的过快，接收方就可能来不及接收，这就会造成数据的丢失。所谓流量控制就是让发送方的发送速率不要太快，要让接收方来得及接收。

16, 在某段时间，若对网络中某一资源的需求超过了该资源所能提供的可用部分，网络的性能就要变坏。这种情况就叫拥塞。拥塞控制就是为了防止过多的数据注入到网络中，这样就可以使网络中的路由器或链路不致过载。拥塞控制所要做的都有一个前提，就是网络能够承受现有的网络负荷。拥塞控制是一个全局性的过程，涉及到所有的主机，所有的路由器，以及与降低网络传输性能有关的所有因素。相反，流量控制往往是点对点通信量的控制，是个端到端的问题。流量控制所要做的就是抑制发送端发送数据的速率，以便使接收端来得及接收。

17, 为了进行拥塞控制，TCP发送方要维持一个拥塞窗口cwnd的状态变量。拥塞控制窗口的大小取决于网络的拥塞程度，并且动态变化。发送方让自己的发送窗口取为拥塞窗口和接收方的接受窗口中较小的一个。

18, TCP的拥塞控制采用了四种算法，即慢开始，拥塞避免，快重传和快恢复。在网络层也可以使路由器采用适当的分组丢弃策略（如主动队列管理AQM），以减少网络拥塞的发生。

19, 运输连接的三个阶段，即：连接建立，数据传送和连接释放。

20，主动发起TCP连接建立的应用进程叫做客户，而被动等待连接建立的应用进程叫做服务器。TCP连接采用三报文握手机制。服务器要确认用户的连接请求，然后客户要对服务器的确认进行确认。

21，TCP的连接释放采用四报文握手机制。任何一方都可以在数据传送结束后发出连接释放的通知，待对方确认后进入半关闭状态。当另一方也没有数据再发送时，则发送连接释放通知，对方确认后就完全关闭了TCP连接

5.3 最重要的知识点

1. 端口和套接字的意义
2. 无连接UDP的特点
3. 面向连接TCP的特点
4. 在不可靠的网络上实现可靠传输的工作原理，停止等待协议和ARQ协议
5. TCP的滑动窗口，流量控制，拥塞控制和连接管理

6. 应用层

6.1 基本术语

域名系统（DNS）：

DNS (Domain Name System, 域名系统)，万维网上作为域名和IP地址相互映射的一个分布式数据库，能够使用户更方便的访问互联网，而不用去记住能够被机器直接读取的IP数串。

通过域名，最终得到该域名对应的IP地址的过程叫做域名解析（或主机名解析）。DNS协议运行在UDP协议之上，使用端口号53。在RFC文档中RFC 2181对DNS有规范说明，RFC 2136对DNS的动态更新进行说明，RFC 2308对DNS查询的反向缓存进行说明。

文件传输协议（FTP）：

FTP是File Transfer Protocol（文件传输协议）的英文简称，而中文简称为“文传协议”。用于Internet上的控制文件的双向传输。同时，它也是一个应用程序（Application）。

基于不同的操作系统有不同的FTP应用程序，而所有这些应用程序都遵守同一种协议以传输文件。在FTP的使用当中，用户经常遇到两个概念：“下载”（Download）和“上传”（Upload）。

“下载”文件就是从远程主机拷贝文件至自己的计算机上；“上传”文件就是将文件从自己的计算机中拷贝至远程主机上。用Internet语言来说，用户可通过客户机程序向（从）远程主机上传（下载）文件。

简单文件传输协议（TFTP）：

TFTP（Trivial File Transfer Protocol, 简单文件传输协议）是TCP/IP协议族中的一个用来在客户机与服务器之间进行简单文件传输的协议，提供不复杂、开销不大的文件传输服务。端口号为69。

远程终端协议（TELNET）：

Telnet协议是TCP/IP协议族中的一员，是Internet远程登陆服务的标准协议和主要方式。它为用户提供了在本地计算机上完成远程主机工作的能力。

在终端使用者的电脑上使用telnet程序，用它连接到服务器。终端使用者可以在telnet程序中输入命令，这些命令会在服务器上运行，就像直接在服务器的控制台上输入一样。

可以在本地就能控制服务器。要开始一个telnet会话，必须输入用户名和密码来登录服务器。Telnet是常用的远程控制Web服务器的方法。

万维网（WWW）：

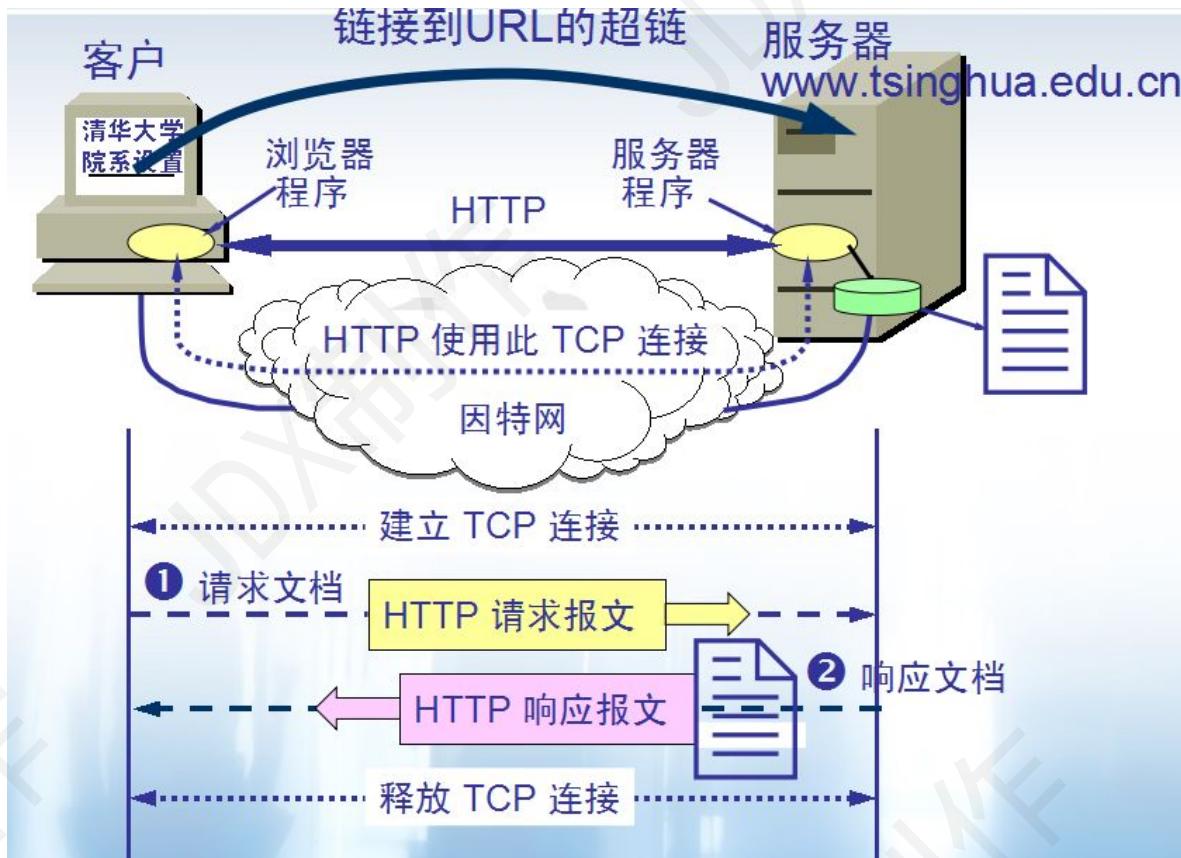
WWW是环球信息网的缩写，（亦作“Web”、“WWW”、“W3”，英文全称为“World Wide Web”），中文名字为“万维网”，“环球网”等，常简称为Web。分为Web客户端和Web服务器程序。

WWW可以让Web客户端（常用浏览器）访问浏览Web服务器上的页面。是一个由许多互相链接的超文本组成的系统，通过互联网访问。在这个系统中，每个有用的事物，称为一样“资源”；并且由一个全局“统一资源标识符”（URI）标识；这些资源通过超文本传输协议（Hypertext Transfer Protocol）传送给用户，而后者通过点击链接来获得资源。

万维网联盟（英语：World Wide Web Consortium，简称W3C），又称W3C理事会。1994年10月在麻省理工学院（MIT）计算机科学实验室成立。万维网联盟的创建者是万维网的发明者蒂姆·伯纳斯-李。

万维网并不等同互联网，万维网只是互联网所能提供的服务其中之一，是靠着互联网运行的一项服务。

万维网的大致工作流程：



统一资源定位符（URL）：

统一资源定位符是对可以从互联网上得到的资源的位置和访问方法的一种简洁的表示，是互联网上标准资源的地址。互联网上的每个文件都有一个唯一的URL，它包含的信息指出文件的位置以及浏览器应该怎么处理它。

超文本传输协议（HTTP）：

超文本传输协议（HTTP，HyperText Transfer Protocol）是互联网上应用最为广泛的一种网络协议。所有的WWW文件都必须遵守这个标准。

设计HTTP最初的目的为了提供一种发布和接收HTML页面的方法。1960年美国人Ted Nelson构思了一种通过计算机处理文本信息的方法，并称之为超文本（hypertext），这成为了HTTP超文本传输协议标准架构的发展根基。

代理服务器（Proxy Server）：

代理服务器（Proxy Server）是一种网络实体，它又称为万维网高速缓存。

代理服务器把最近的一些请求和响应暂存在本地磁盘中。当新请求到达时，若代理服务器发现这个请求与暂时存放的请求相同，就返回暂存的响应，而不需要按URL的地址再次去互联网访问该资源。

代理服务器可在客户端或服务器工作，也可以在中间系统工作。

http请求头:

http请求头，HTTP客户程序（例如浏览器），向服务器发送请求的时候必须指明请求类型（一般是GET或者POST）。如有必要，客户程序还可以选择发送其他的请求头。

- Accept: 浏览器可接受的MIME类型。
- Accept-Charset: 浏览器可接受的字符集。
- Accept-Encoding: 浏览器能够进行解码的数据编码方式，比如gzip。Servlet能够向支持gzip的浏览器返回经gzip编码的HTML页面。许多情形下这可以减少5到10倍的下载时间。
- Accept-Language: 浏览器所希望的语言种类，当服务器能够提供一种以上的语言版本时要用到。
- Authorization: 授权信息，通常出现在对服务器发送的WWW-Authenticate头的应答中。
- Connection: 表示是否需要持久连接。如果Servlet看到这里的值为“Keep-Alive”，或者看到请求使用的是HTTP 1.1（HTTP 1.1默认进行持久连接），它就可以利用持久连接的优点，当页面包含多个元素时（例如Applet，图片），显著地减少下载所需要的时间。要实现这一点，Servlet需要在应答中发送一个Content-Length头，最简单的实现方法是：先把内容写入ByteArrayOutputStream，然后在正式写出内容之前计算它的大小。
- Content-Length: 表示请求消息正文的长度。
- Cookie: 这是最重要的请求头信息之一
- From: 请求发送者的email地址，由一些特殊的Web客户程序使用，浏览器不会用到它。
- Host: 初始URL中的主机和端口。
- If-Modified-Since: 只有当所请求的内容在指定的日期之后又经过修改才返回它，否则返回304“Not Modified”应答。
- Pragma: 指定“no-cache”值表示服务器必须返回一个刷新后的文档，即使它是代理服务器而且已经有了页面的本地拷贝。
- Referer: 包含一个URL，用户从该URL代表的页面出发访问当前请求的页面。
- User-Agent: 浏览器类型，如果Servlet返回的内容与浏览器类型有关则该值非常有用。

简单邮件传输协议(SMTP):

SMTP (Simple Mail Transfer Protocol) 即简单邮件传输协议,它是一组用于由源地址到目的地址传送邮件的规则，由它来控制信件的中转方式。

SMTP协议属于TCP/IP协议簇，它帮助每台计算机在发送或中转信件时找到下一个目的地。

通过SMTP协议所指定的服务器,就可以把E-mail寄到收信人的服务器上了，整个过程只要几分钟。

SMTP服务器则是遵循SMTP协议的发送邮件服务器，用来发送或中转发出的电子邮件。

搜索引擎:

搜索引擎 (Search Engine) 是指根据一定的策略、运用特定的计算机程序从互联网上搜集信息，在对信息进行组织和处理后，为用户提供检索服务，将用户检索相关的信息展示给用户的系统。

搜索引擎包括全文索引、目录索引、元搜索引擎、垂直搜索引擎、集合式搜索引擎、门户搜索引擎与免费链接列表等。

全文索引:

全文索引技术是目前搜索引擎的关键技术。

试想在1M大小的文件中搜索一个词，可能需要几秒，在100M的文件中可能需要几十秒，如果在更大的文件中搜索那么就需要更大的系统开销，这样的开销是不现实的。

所以在这样的矛盾下出现了全文索引技术，有时候有人叫倒排文档技术。

目录索引:

目录索引 (search index/directory)，顾名思义就是将网站分门别类地存放在相应的目录中，因此用户在查询信息时，可选择关键词搜索，也可按分类目录逐层查找。

垂直搜索引擎：

垂直搜索引擎是针对某一个行业的专业搜索引擎，是搜索引擎的细分和延伸，是对网页库中的某类专门的信息进行一次整合，定向分字段抽取出需要的数据进行处理后再以某种形式返回给用户。

垂直搜索是相对通用搜索引擎的信息量大、查询不准确、深度不够等提出来的新的搜索引擎服务模式，通过针对某一特定领域、某一特定人群或某一特定需求提供的有一定价值的信息和相关服务。

其特点就是“专、精、深”，且具有行业色彩，相比较通用搜索引擎的海量信息无序化，垂直搜索引擎则显得更加专注、具体和深入。

6.2 重要知识点总结

1. 文件传输协议（FTP）使用TCP可靠的运输服务。FTP使用客户服务器方式。一个FTP服务器进程可以同时为多个用户提供服务。在进行文件传输时，FTP的客户和服务器之间要先建立两个并行的TCP连接：控制连接和数据连接。实际用于传输文件的是数据连接。
2. 万维网客户程序与服务器之间进行交互使用的协议是超文本传输协议HTTP。HTTP使用TCP连接进行可靠传输。但HTTP本身是无连接、无状态的。HTTP/1.1协议使用了持续连接（分为非流水线方式和流水线方式）
3. 电子邮件把邮件发送到收件人使用的邮件服务器，并放在其中的收件人邮箱中，收件人可随时上网到自己使用的邮件服务器读取，相当于电子邮箱。
4. 一个电子邮件系统有三个重要组成构件：用户代理、邮件服务器、邮件协议（包括邮件发送协议，如SMTP，和邮件读取协议，如POP3和IMAP）。用户代理和邮件服务器都要运行这些协议。

6.3 最重要知识点总结

1. 域名系统-从域名解析出IP地址
2. 访问一个网站大致的过程
3. 系统调用和应用编程接口概念

(二). HTTPS中的TLS

1. SSL 与 TLS

SSL：(Secure Socket Layer) 安全套接层，于1994年由网景公司设计，并于1995年发布了3.0版本

TLS：(Transport Layer Security) 传输层安全性协议，是IETF在SSL3.0的基础上设计的协议
以下全部使用TLS来表示

2. 从网络协议的角度理解 HTTPS

HTTP：HyperText Transfer Protocol 超文本传输协议

HTTPS：Hypertext Transfer Protocol Secure 超文本传输安全协议

TLS：位于HTTP和TCP之间的协议，其内部有TLS握手协议、TLS记录协议

HTTPS经由HTTP进行通信，但利用TLS来保证安全，即HTTPS = HTTP + TLS

3. 从密码学的角度理解 HTTPS

HTTPS使用TLS保证安全，这里的“安全”分两部分，一是传输内容加密、二是服务端的身份认证

3.1. TLS 工作流程

此为服务端单向认证，还有客户端/服务端双向认证，流程类似，只不过客户端也有自己的证书，并发给服务器进行验证

3.2. 密码基础

3.2.1. 伪随机数生成器

为什么叫伪随机数，因为没有真正意义上的随机数，具体可以参考 Random/ThreadLocalRandom
它的主要作用在于生成对称密码的秘钥、用于公钥密码生成秘钥对

3.2.2. 消息认证码

消息认证码主要用于验证消息的完整性与消息的认证，其中消息的认证指“消息来自正确的发送者”

| 消息认证码用于验证和认证，而不是加密

1. 发送者与接收者事先共享秘钥
2. 发送者根据发送消息计算 MAC 值
3. 发送者发送消息和 MAC 值
4. 接收者根据接收到的消息计算 MAC 值
5. 接收者根据自己计算的 MAC 值与收到的 MAC 对比
6. 如果对比成功，说明消息完整，并来自与正确的发送者

3.2.3. 数字签名

消息认证码的缺点在于**无法防止否认**，因为共享秘钥被 client、server 两端拥有，server 可以伪造 client 发送给自己的消息（自己给自己发送消息），为了解决这个问题，我们需要它们有各自的秘钥不被第二个知晓（这样也解决了共享秘钥的配送问题）

| 数字签名和消息认证码都不是为了加密

可以将单向散列函数获取散列值的过程理解为使用 md5 摘要算法获取摘要的过程

使用自己的私钥对自己所认可的消息生成一个该消息专属的签名，这就是数字签名，表明我承认该消息来自自己

注意：**私钥用于加签，公钥用于解签，每个人都可以解签，查看消息的归属人**

3.2.4. 公钥密码

公钥密码也叫非对称密码，由公钥和私钥组成，它是最开始是为了解决秘钥的配送传输安全问题，即，我们不配送私钥，只配送公钥，私钥由本人保管

它与数字签名相反，公钥密码的私钥用于解密、公钥用于加密，每个人都可以用别人的公钥加密，但只有对应的私钥才能解开密文

client：明文 + 公钥 = 密文

server：密文 + 私钥 = 明文

注意：**公钥用于加密，私钥用于解密，只有私钥的归属者，才能查看消息的真正内容**

3.2.5. 证书

证书：全称公钥证书（Public-Key Certificate, PKC），里面保存着归属者的基本信息，以及证书过期时间、归属者的公钥，并由认证机构（Certification Authority, CA）施加数字签名，表明，某个认证机构认定该公钥的确属于此人

| 想象这个场景：你想在支付宝页面交易，你需要支付宝的公钥进行加密通信，于是你从百度上搜索关键字“支付宝公钥”，你获得了支付宝的公钥，这个时候，支付宝通过中间人攻击，让你访问到了他们支付宝的页面，最后你在这个支付宝页面完美的使用了支付宝的公钥完成了与支付宝的交易

在上面的场景中，你可以理解支付宝证书就是由支付宝的公钥、和给支付宝颁发证书的企业的数字签名组成

任何人都可以给自己或别人的公钥添加自己的数字签名，表明：我拿我的尊严担保，我的公钥/别人的公钥是真的，至于信不信那是另一回事了

3.2.6. 密码小结

密码	作用	组成
消息认证码	确认消息的完整、并对消息的来源认证	共享秘钥+消息的散列值
数字签名	对消息的散列值签名	公钥+私钥+消息的散列值
公钥密码	解决秘钥的配送问题	公钥+私钥+消息
证书	解决公钥的归属问题	公钥密码中的公钥+数字签名

3.3. TLS 使用的密码技术

1. 伪随机数生成器：秘钥生成随机性，更难被猜测
2. 对称密码：对称密码使用的秘钥就是由伪随机数生成，相较于非对称密码，效率更高
3. 消息认证码：保证消息信息的完整性、以及验证消息信息的来源
4. 公钥密码：证书技术使用的就是公钥密码
5. 数字签名：验证证书的签名，确定由真实的某个 CA 颁发
6. 证书：解决公钥的真实归属问题，降低中间人攻击概率

3.4. TLS 总结

TLS 是一系列密码工具的框架，作为框架，它也是非常的灵活，体现在每个工具套件它都可以替换，即：客户端与服务端之间协商密码套件，从而更难的被攻破，例如使用不同方式的对称密码，或者公钥密码、数字签名生成方式、单向散列函数技术的替换等

4. RSA 简单示例

RSA 是一种公钥密码算法，我们简单的走一遍它的加密解密过程

加密算法：密文 = (明文^E) mod N，其中公钥为{E,N}，即“求明文的E次方的对 N 的余数”

解密算法：明文 = (密文^D) mod N，其中秘钥为{D,N}，即“求密文的D次方的对 N 的余数”

例：我们已知公钥为{5,323}，私钥为{29,323}，明文为300，请写出加密和解密的过程：

加密：密文 = $123^5 \bmod 323 = 225$

解密：明文 = $225^{29} \bmod 323 = [(225^5) \bmod 323] * [(225^4) \bmod 323] \bmod 323 = (4 * 4 * 4 * 4 * 290) \bmod 323 = 123$

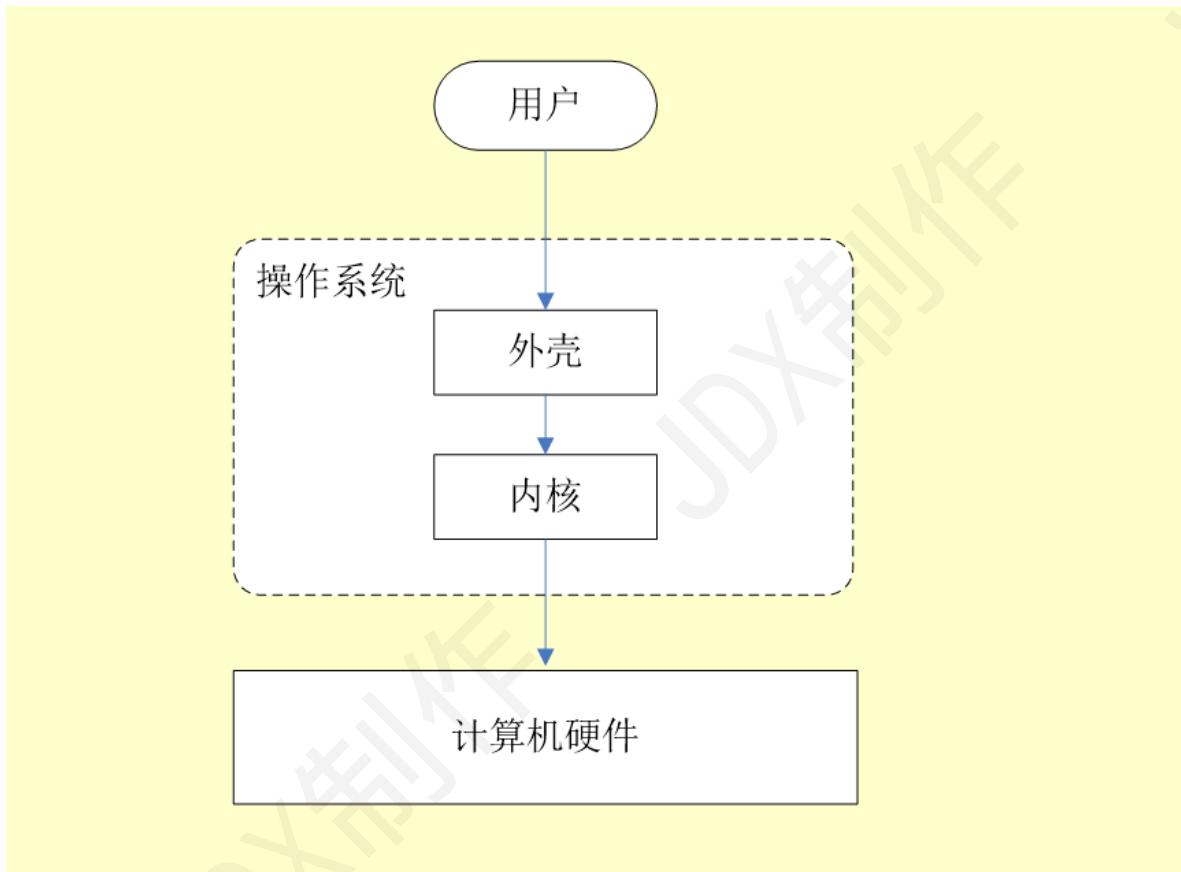
三、Linux

(一). 从认识操作系统开始

1.1 操作系统简介

我通过以下四点介绍什么是操作系统：

- 操作系统（Operation System，简称OS）是管理计算机硬件与软件资源的程序，是计算机系统的内核与基石；
- 操作系统本质上是运行在计算机上的软件程序；
- 为用户提供一个与系统交互的操作界面；
- 操作系统分内核与外壳（我们可以把外壳理解成围绕着内核的应用程序，而内核就是能操作硬件的程序）。

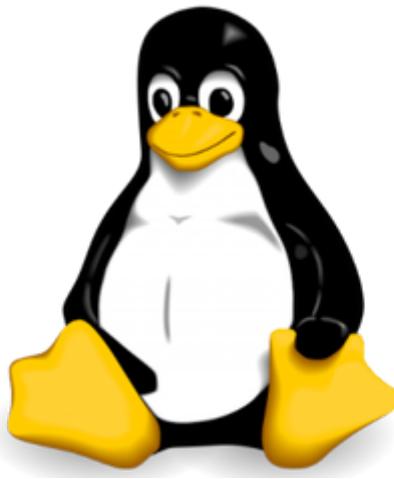


1.2 操作系统简单分类

1. **Windows:** 目前最流行的个人桌面操作系统，不做多的介绍，大家都清楚。
2. **Unix:** 最早的多用户、多任务操作系统。按照操作系统的分类，属于分时操作系统。Unix 大多被用在服务器、工作站，现在也有用在个人计算机上。它在创建互联网、计算机网络或客户端/服务器模型方面发挥着非常重要的作用。



3. **Linux:** Linux是一套免费使用和自由传播的类Unix操作系统。Linux存在着许多不同的Linux版本，但它们都使用了 **Linux内核**。Linux可安装在各种计算机硬件设备中，比如手机、平板电脑、路由器、视频游戏控制台、台式计算机、大型机和超级计算机。严格来讲，Linux这个词本身只表示Linux内核，但实际上人们已经习惯了用Linux来形容整个基于Linux内核，并且使用GNU工程各种工具和数据库的操作系统。



1.3 操作系统的内核

操作系统的内核是操作系统的核心部分。

它负责系统的内存管理，硬件设备的管理，文件系统的管理以及应用程序的管理。

我们常说的Linux，其实是指基于Linux内核开发的操作系统。

常见的Linux系统发行版有:Debian,RedHat,Ubuntu,Suse,Centeos等等。

1.4 操作系统的用户态与内核态

unix与linux的体系架构：分为用户态与内核态。

用户态与内核态与内核态是操作系统对执行权限进行分级后的不同的运行模式。

1.4.1 为什么要有用户态与内核态？

在cpu的所有指令中，有些指令是非常危险的，如果使用不当，将会造成系统崩溃等后果。

为了避免这种情况发生，cpu将指令划分为**特权级(内核态)指令**和**非特权级(用户态)指令**。

对于那些危险的指令只允许内核及其相关模块调用，对于那些不会造成危险的指令，就允许用户应用程序调用。

- 内核态(核心态,特权态): **内核态是操作系统内核运行的模式。**
内核态控制计算机的硬件资源，如硬件设备，文件系统等等，并为上层应用程序提供执行环境。
- 用户态: **用户态是用户应用程序运行的状态。**
应用程序必须依托于内核态运行,因此用户态的态的操作权限比内核态是要低的，
如磁盘，文件等，访问操作都是受限的。
- 系统调用: 系统调用是操作系统为应用程序提供能够访问到内核态的资源的接口。

1.4.2 用户态切换到内核态的几种方式

- 系统调用: 系统调用是用户态主动要求切换到内核态的一种方式，
用户应用程序通过操作系统调用内核为上层应用程序开放的接口来执行程序。
- 异常: 当cpu在执行用户态的应用程序时，发生了某些不可知的异常。
于是当前用户态的应用进程切换到处理此异常的内核的程序中去。
- 硬件设备的中断: 当硬件设备完成用户请求后，会向cpu发出相应的中断信号，
这时cpu会暂停执行下一条即将要执行的指令，转而去执行与中断信号对应的应用程序，
如果先前执行的指令是用户态下程序的指令，那么这个转换过程也是用户态到内核态的转换。

1.4.3 物理内存RAM(Random Access Memory 随机存储器)

物理内存是计算机的实际内存大小，它直接与CPU交换数据，也被称为主存。

1.4.4 虚拟内存(Virtual Memory)

虚拟内存是操作系统为了更高效率使用物理内存的一种概念，它是对物理内存的抽象。
windows上的虚拟内存和Linux上的swap交换空间都是虚拟内存的一种实现技术。

1.4.5 Swap交换空间

简单理解：当某个应用程序所需的内存空间不够了，
那么系统会判断当前物理内存是否还有足够的空闲可以分配给应用程序。
如果有，则应用程序直接进入内存运行；如果没有，系统就根据某种算法（如：LRU）挂起一个进程，
将挂起的进程交换到虚拟内存Swap中等待，并将应用程序调入内存执行。
虚拟内存是被虚拟出来的，可以使用硬盘（不仅仅是硬盘）来作为虚拟内存。

这就是为什么当我们运行一个所需内存比我们计算机内存还大的程序时，仍然可以正常运行，并感受不到内存的限制的原因。

（二）初探Linux

2.1 Linux简介

我们上面已经介绍到了Linux，我们这里只强调三点。

- **类Unix系统：** Linux是一种自由、开放源码的类似Unix的操作系统
- **Linux内核：** 严格来说，Linux这个词本身只表示Linux内核
- **Linux之父：** 一个编程领域的传奇式人物。他是Linux内核的最早作者，随后发起了这个开源项目，担任Linux内核的首要架构师与项目协调者，是当今世界最著名的电脑程序员、黑客之一。他还发起了Git这个开源项目，并为主要的开发者。



2.2 Linux诞生简介

- 1991年，芬兰的业余计算机爱好者Linus Torvalds编写了一款类似Minix的系统（基于微内核架构的类Unix操作系统）被ftp管理员命名为Linux加入到自由软件基金的GNU计划中；
- Linux以一只可爱的企鹅作为标志，象征着敢作敢为、热爱生活。

2.3 Linux的分类

Linux根据原生程度，分为两种：

- 内核版本：**Linux不是一个操作系统，严格来讲，Linux只是一个操作系统中的内核。内核是什么？内核建立了计算机软件与硬件之间通讯的平台，内核提供系统服务，比如文件管理、虚拟内存、设备I/O等；
- 发行版本：**一些组织或公司在内核版基础上进行二次开发而重新发行的版本。Linux发行版本有很多（ubuntu和CentOS用的都很多，初学建议选择CentOS），如下图所示：



(三) Linux文件系统概览

3.1 Linux文件系统简介

在Linux操作系统中，所有被操作系统管理的资源，例如网络接口卡、磁盘驱动器、打印机、输入输出设备、普通文件或是目录都被看作是一个文件。

也就是说在LINUX系统中有一个重要的概念：**一切都是文件**。其实这是UNIX哲学的一个体现，而Linux是重写UNIX而来，所以这个概念也就传承了下来。在UNIX系统中，把一切资源都看作是文件，包括硬件设备。UNIX系统把每个硬件都看成是一个文件，通常称为设备文件，这样用户就可以用读写文件的方式实现对硬件的访问。

3.2 Inode

inode是linux/unix文件系统和硬盘存储的基础，如果理解了inode，将会对我们学习如何将复杂的概念抽象成简单概念有重大帮助。

3.2.1 Inode是什么?有什么作用?

文件存储在硬盘上，硬盘的最小存储单位是扇区(Sector)，每个扇区存储512字节(0.5kb)。

操作系统读取硬盘的数据时，不会一个扇区一个扇区的读取，这样做效率较低，而是一次读取多个扇区，

即一次读取一个块(block)。块由多个扇区组成，是文件读取的最小单位，块的最常见的大小是4kb，约为8个连续的扇区组成。文件数据存储在块中，

但还需要一个空间来存储文件的元信息metadata，如文件拥有者，创建时间，权限，大小等。

这种存储文件元信息的区域就叫inode，译为索引节点。每个文件都有一个inode，存储文件的元信息。

使用 stat 命令可以查看文件的inode信息。每个inode都有一个号码，

Linux/Unix操作系统不使用文件名来区分文件，而是使用inode号码区分不同的文件。

inode也需要消耗硬盘空间，所以在格式化硬盘的时候，操作系统会将硬盘分为2个区域，一个区域存放文件数据，另一个区域存放inode所包含的信息，存放inode的区域被称为inode table。

3.3 文件类型与目录结构

**Linux支持很多文件类型，其中非常重要的文件类型有：

普通文件，目录文件，链接文件，设备文件，管道文件，Socket套接字文件等。

- 普通文件：普通文件是指txt,html,pdf等等的这样应用层面的文件类型，用户可以根据访问权限对普通文件进行访问，修改和删除。
- 目录文件：目录也是一种文件，打开目录实际上是打开目录文件。目录文件包含了它目录下的所有文件名以及指向这些文件的指针。
- 链接文件：链接文件分为符号链接(软链接)文件和硬链接文件
 - 硬链接(Hard Link):硬链接的文件拥有相同的inode，因为操作系统是靠inode来区分文件的，2个inode相同的文件，就代表它们是一个文件。删除一个文件并不会对其他拥有相同inode的文件产生影响，只有当inode相同的所有文件被删除了，这个文件才会被删除。换言之，你建立一个文件的硬链接，这个文件和硬链接它们的inode是相同的，无论你删除的是硬链接还是源文件，都不会对彼此造成影响，除非你把硬链接和源文件都删除，这个文件才被删除。
 - 符号链接(软链接)(Symbolic Link): 符号链接类似于Windows上的快捷方式，它保存了源文件的路径。当符号链接被删除时，并不会影响源文件。但是当源文件被删除时，符号链接就找不到源文件了。

软链接和硬链接：

- 设备文件：设备文件分为块设备文件和字符设备文件，设备文件一般存于/dev目录下。
 - 字符设备文件：字符设备是依照先后顺序被存取数据的设备，通常不支持随机存取，此类设备可以按字节/字符来读取数据，如键盘，串口等等。
 - 块设备文件：块设备是可以被随机存取数据的设备，应用程序可以访问块设备上任何一块位置。块设备以块的方式读取数据，在Windows下也称为簇，块设备不支持字符的方式寻址。如硬盘，软盘，光碟等等。

字符设备与块设备最根本的区别就是它们是否可以被随机访问。

如键盘，当我们在键盘上敲下一个单词：“word”的时候，

那么系统肯定是需要按照顺序来进行读取word的字节流(字符流)的，随机访问在此时是没有意义的。

- 管道文件：管道文件一般用于进程间通信，使用mkfifo命令可以创建一个管道文件。
- Socket套接字文件：套接字文件被用于网络进程之间的通信，既可以使2台不同的机器进行通信，也可以用于本机的Socket网络程序。

3.4 Linux目录树

所有可操作的计算机资源都存在于目录树这个结构中，对计算资源的访问，可以看做是对这棵目录树的访问。

Linux的目录结构如下：

小插曲：

更多阿里、腾讯、美团、京东等一线互联网大厂Java面试真题；包含：基础、并发、锁、JVM、设计模式、数据结构、反射/IO、数据库、Redis、Spring、消息队列、分布式、Zookeeper、Dubbo、Mybatis、Maven、面试经等。

更多Java程序员技术进阶小技巧；例如高效学习（如何学习和阅读代码、面对枯燥和量大的知识）高效沟通（沟通方式及技巧、沟通技术）

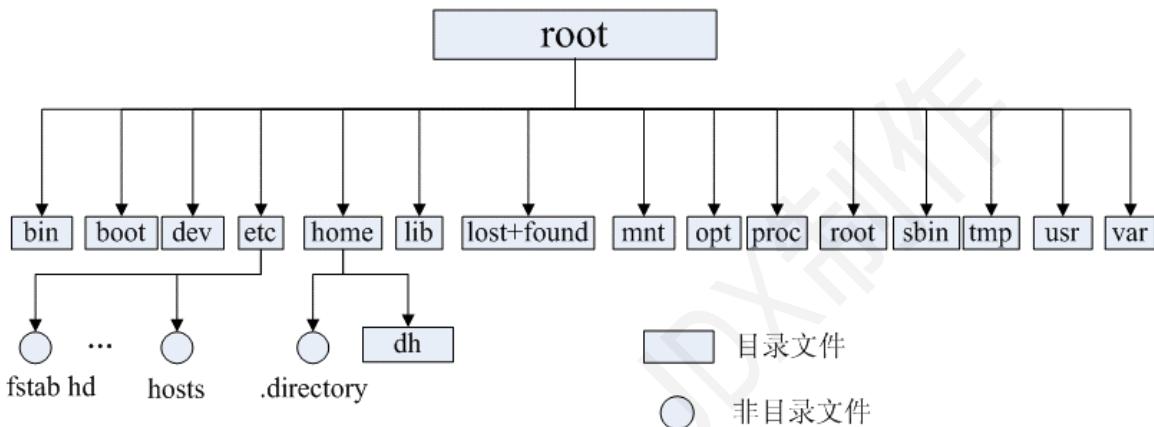
更多Java大牛分享的一些职业生涯分享文档

请添加微信：jiang10086a免费获取

比你优秀的对手在学习，你的仇人在磨刀，你的闺蜜在减肥，隔壁老王在练腰，我们必须不断学习，否则我们将被学习者超越！

趁年轻，使劲拼，给未来的自己一个交代！

Linux文件系统的结构层次鲜明，就像一棵倒立的树，最顶层是其根目录：



常见目录说明：

- **/bin:** 存放二进制可执行文件(ls、cat、mkdir等)，常用命令一般都在这里；
- **/etc:** 存放系统管理和配置文件；
- **/home:** 存放所有用户文件的根目录，是用户主目录的基点，比如用户user的主目录就是/home/user，可以用~user表示；
- **/usr :** 用于存放系统应用程序；
- **/opt:** 额外安装的可选应用程序包所放置的位置。一般情况下，我们可以把tomcat等都安装到这里面；
- **/proc:** 虚拟文件系统目录，是系统内存的映射。可直接访问这个目录来获取系统信息；
- **/root:** 超级用户（系统管理员）的主目录（特权阶级^o^）；
- **/sbin:** 存放二进制可执行文件，只有root才能访问。这里存放的是系统管理员使用的系统级别的管理命令和程序。如ifconfig等；
- **/dev:** 用于存放设备文件；
- **/mnt:** 系统管理员安装临时文件系统的安装点，系统提供这个目录是让用户临时挂载其他的文件系统；
- **/boot:** 存放用于系统引导时使用的各种文件；
- **/lib :** 存放着和系统运行相关的库文件；
- **/tmp:** 用于存放各种临时文件，是公用的临时文件存储点；
- **/var:** 用于存放运行时需要改变数据的文件，也是某些大文件的溢出区，比方说各种服务的日志文件（系统启动日志等。）等；
- **/lost+found:** 这个目录平时是空的，系统非正常关机而留下“无家可归”的文件（windows下叫什么.chk）就在这里。

(四) Linux基本命令

下面只是给出了一些比较常用的命令。推荐一个Linux命令快查网站，非常不错，大家如果遗忘某些命令或者对某些命令不理解都可以在这里得到解决。

4.1 目录切换命令

- `cd usr`： 切换到该目录下usr目录
- `cd ..`（或`cd.../`）： 切换到上一层目录
- `cd /`： 切换到系统根目录
- `cd ~`： 切换到用户主目录
- `cd -`： 切换到上一个操作所在目录

4.2 目录的操作命令(增删改查)

1. `mkdir` 目录名称： 增加目录

2. `ls`或者`ll` (`ll`是`ls -l`的别名, `ll`命令可以看到该目录下的所有目录和文件的详细信息) : 查看目录信息

3. `find 目录 参数`: 寻找目录(查)

示例:

- 列出当前目录及子目录下所有文件和文件夹: `find .`
- 在`/home`目录下查找以.txt结尾的文件名: `find /home -name "*.txt"`
- 同上, 但忽略大小写: `find /home -iname "*.txt"`
- 当前目录及子目录下查找所有以.txt和.pdf结尾的文件: `find . \(-name "*.txt" -o -name "*.pdf"\)` 或 `find . -name "*.txt" -o -name "*.pdf"`

4. `mv 目录名称 新目录名称`: 修改目录的名称(改)

注意: `mv`的语法不仅可以对目录进行重命名而且也可以对各种文件, 压缩包等进行重命名的操作。`mv`命令用来对文件或目录重新命名, 或者将文件从一个目录移到另一个目录中。后面会介绍到`mv`命令的另一个用法。

5. `mv 目录名称 目录的新位置`: 移动目录的位置--剪切(改)

注意: `mv`语法不仅可以对目录进行剪切操作, 对文件和压缩包等都可执行剪切操作。另外`mv`与`cp`的结果不同, `mv`好像文件“搬家”, 文件个数并未增加。而`cp`对文件进行复制, 文件个数增加了。

6. `cp -r 目录名称 目录拷贝的目标位置`: 拷贝目录(改), `-r`代表递归拷贝

注意: `cp`命令不仅可以拷贝目录还可以拷贝文件, 压缩包等, 拷贝文件和压缩包时不 用写`-r`递归

7. `rm [-rf] 目录`: 删除目录(删)

注意: `rm`不仅可以删除目录, 也可以删除其他文件或压缩包, 为了增强大家的记忆, 无论删除任何目录或文件, 都直接使用 `rm -rf` 目录/文件/压缩包

4.3 文件的操作命令(增删改查)

1. `touch 文件名称`: 文件的创建(增)

2. `cat/more/less/tail 文件名称`: 文件的查看(查)

- `cat`: 查看显示文件内容
- `more`: 可以显示百分比, 回车可以向下一行, 空格可以向下一页, `q`可以退出查看
- `less`: 可以使用键盘上的PgUp和PgDn向上 和向下翻页, `q`结束查看
- `tail -10`: 查看文件的后10行, `Ctrl+C`结束

注意: 命令 `tail -f` 文件 可以对某个文件进行动态监控, 例如tomcat的日志文件, 会随着程序的运行, 日志会变化, 可以使用`tail -f catalina-2016-11-11.log` 监控文件的变化

3. `vim 文件`: 修改文件的内容(改)

`vim`编辑器是Linux中的强大组件, 是`vi`编辑器的加强版, `vim`编辑器的命令和快捷方式有很多, 但此处不一一阐述, 大家也无需研究的很透彻, 使用`vim`编辑修改文件的方式基本会使用就可以了。

在实际开发中, 使用`vim`编辑器主要作用就是修改配置文件, 下面是一般步骤:

`vim` 文件----->进入文件----->命令模式----->按*i*进入编辑模式----->编辑文件 ----->按*Esc*进入底行模式----->输入: `wq/q!` (输入`wq`代表写入内容并退出, 即保存; 输入`q!`代表强制退出不保存。)

4. `rm -rf 文件`: 删除文件(删)

同目录删除: 熟记 `rm -rf` 文件即可

4.4 压缩文件的操作命令

1) 打包并压缩文件:

Linux中的打包文件一般是以.tar结尾的, 压缩的命令一般是以.gz结尾的。

而一般情况下打包和压缩是一起进行的，打包并压缩后的文件的后缀名一般.tar.gz。

命令：`tar -zcvf 打包压缩后的文件名 要打包压缩的文件`

其中：

z：调用gzip压缩命令进行压缩

c：打包文件

v：显示运行过程

f：指定文件名

比如：假如test目录下有三个文件分别是：aaa.txt bbb.txt ccc.txt，如果我们要打包test目录并指定压缩后的压缩包名称为test.tar.gz可以使用命令：`tar -zcvf test.tar.gz aaa.txt bbb.txt`

`ccc.txt 或：tar -zcvf test.tar.gz /test/`

2) 解压压缩包：

命令：`tar [-xvf]` 压缩文件

其中：x：代表解压

示例：

1 将/test下的test.tar.gz解压到当前目录下可以使用命令：`tar -xvf test.tar.gz`

2 将/test下的test.tar.gz解压到根目录/usr下：`tar -xvf test.tar.gz -C /usr` (- C代表指定解压的位置)

4.5 Linux的权限命令

操作系统中每个文件都拥有特定的权限、所属用户和所属组。权限是操作系统用来限制资源访问的机制，在Linux中权限一般分为读(readable)、写(writable)和执行(exutable)，分为三组。分别对应文件的属主(owner)，属组(group)和其他用户(other)，通过这样的机制来限制哪些用户、哪些组可以对特定的文件进行什么样的操作。通过`ls -l`命令我们可以查看某个目录下的文件或目录的权限

示例：在随意某个目录下`ls -l`

-rw-r--r--.	1	root	root	128	Jun 14	23:04	aaa.txt
-rw-r--r--.	1	root	root	0	Jun 14	23:03	bbb.txt
dr-xr-xr-x.	2	root	root	36864	Jun 14	18:08	bin
-rw-r--r--.	1	root	root	0	Jun 14	23:04	ccc.txt
drwxr-xr-x.	2	root	root	4096	Sep 23	2011	etc
drwxr-xr-x.	2	root	root	4096	Sep 23	2011	games
drwxr-xr-x.	35	root	root	4096	Jun 14	17:20	include
dr-xr-xr-x.	80	root	root	36864	Jun 14	18:08	lib
drwxr-xr-x.	17	root	root	4096	Jun 14	18:08	libexec
drwxr-xr-x.	11	root	root	4096	Jun 14	17:17	local
dr-xr-xr-x.	2	root	root	12288	Jun 14	18:08	sbin
drwxr-xr-x.	125	root	root	4096	Jun 14	17:21	share
drwxr-xr-x.	4	root	root	4096	Jun 14	17:17	src
lrwxrwxrwx.	1	root	root	10	Jun 14	17:17	tmp -> ../../var/tmp

第一列的内容的信息解释如下：



下面将详细讲解文件的类型、Linux中权限以及文件有所有者、所在组、其它组具体是什么？

文件的类型：

- d：代表目录
- -：代表文件
- l：代表软链接（可以认为是Windows中的快捷方式）

Linux中权限分为以下几种：

- r：代表权限是可读，r也可以用数字4表示
- w：代表权限是可写，w也可以用数字2表示
- x：代表权限是可执行，x也可以用数字1表示

文件和目录权限的区别：

对文件和目录而言，读写执行表示不同的意义。

对于文件：

权限名称	可执行操作
r	可以使用cat查看文件的内容
w	可以修改文件的内容
x	可以将其运行为二进制文件

对于目录：

权限名称	可执行操作
r	可以查看目录下列表
w	可以创建和删除目录下文件
x	可以使用cd进入目录

需要注意的是超级用户可以无视普通用户的权限，即使文件目录权限是000，依旧可以访问。

在Linux中的每个用户必须属于一个组，不能独立于组外。在Linux中每个文件有所有者、所在组、其它组的概念。

• 所有者

一般为文件的创建者，谁创建了该文件，就天然的成为该文件的所有者，用ls -ahl命令可以看到文件的所有者也可以使用chown 用户名 文件名来修改文件的所有者。

• 文件所在组

当某个用户创建了一个文件后，这个文件的所在组就是该用户所在的组。用ls -ahl命令可以看到文件的所有组，也可以使用chgrp 组名 文件名来修改文件所在的组。

- 其它组

除开文件的所有者和所在组的用户外，系统的其它用户都是文件的其它组。

我们再来看看如何修改文件/目录的权限。

修改文件/目录的权限的命令：chmod

示例：修改/test下的aaa.txt的权限为属主有全部权限，属主所在的组有读写权限，其他用户只有读的权限

```
chmod u=rwx,g=rw,o=r aaa.txt
```

```
chmod -R u=rwx,g=rwx,o=rwx ./log // 递归给log目录下的所有文件授权
```

```
[root@CentOS test]# ll
total 8
-rw-r--r--. 1 root root 2237 Jun 14 23:24 aaa.txt
-rw-r--r--. 1 root root     0 Jun 14 23:03 bbb.txt
-rw-r--r--. 1 root root     0 Jun 14 23:04 ccc.txt
-rw-r--r--. 1 root root  308 Jun 14 23:05 xxxx.tar.gz
[root@CentOS test]# chmod u=rwx,g=rw,o=r aaa.txt
[root@CentOS test]# ll
total 8
-rwxrw-r--. 1 root root 2237 Jun 14 23:24 aaa.txt
-rw-r--r--. 1 root root     0 Jun 14 23:03 bbb.txt
-rw-r--r--. 1 root root     0 Jun 14 23:04 ccc.txt
-rw-r--r--. 1 root root  308 Jun 14 23:05 xxxx.tar.gz
[root@CentOS test]# _
```

上述示例还可以使用数字表示：

```
chmod 764 aaa.txt
```

补充一个比较常用的东西：

假如我们装了一个zookeeper，我们每次开机到要求其自动启动该怎么办？

1. 新建一个脚本zookeeper
2. 为新建的脚本zookeeper添加可执行权限，命令是：`chmod +x zookeeper`
3. 把zookeeper这个脚本添加到开机启动项里面，命令是：`chkconfig --add zookeeper`
4. 如果想看看是否添加成功，命令是：`chkconfig --list`

4.6 Linux 用户管理

Linux系统是一个多用户多任务的分时操作系统，任何一个要使用系统资源的用户，都必须首先向系统管理员申请一个账号，然后以这个账号的身份进入系统。

用户的账号一方面可以帮助系统管理员对使用系统的用户进行跟踪，并控制他们对系统资源的访问；另一方面也可以帮助用户组织文件，并为用户提供安全性保护。

Linux用户管理相关命令：

- `useradd 选项 用户名` :添加用户账号
- `userdel 选项 用户名` :删除用户帐号
- `usermod 选项 用户名` :修改帐号
- `passwd 用户名` :更改或创建用户的密码
- `passwd -s 用户名` :显示用户账号密码信息

- `passwd -d 用户名`: 清除用户密码

useradd命令用于Linux中创建的新的系统用户。useradd可用来建立用户帐号。帐号建好之后，再用passwd设定帐号的密码。而可用userdel删除帐号。使用useradd指令所建立的帐号，实际上是保存在/etc/passwd文本文件中。

passwd命令用于设置用户的认证信息，包括用户密码、密码过期时间等。系统管理者则能用它管理系统用户的密码。只有管理者可以指定用户名称，一般用户只能变更自己的密码。

4.7 Linux系统用户组的管理

每个用户都有一个用户组，系统可以对一个用户组中的所有用户进行集中管理。不同Linux系统对用户组的规定有所不同，如Linux下的用户属于与它同名的用户组，这个用户组在创建用户时同时创建。

用户组的管理涉及用户组的添加、删除和修改。组的增加、删除和修改实际上就是对/etc/group文件的更新。

Linux系统用户组的管理相关命令：

- `groupadd 选项 用户组`: 增加一个新的用户组
- `groupdel 用户组`: 要删除一个已有的用户组
- `groupmod 选项 用户组`: 修改用户组的属性

4.8 其他常用命令

- `pwd`: 显示当前所在位置
- `sudo + 其他命令`: 以系统管理者的身份执行指令，也就是说，经由 sudo 所执行的指令就好像是 root 亲自执行。
- `grep 要搜索的字符串 要搜索的文件 --color`: 搜索命令，--color代表高亮显示
- `ps -ef / ps -aux`: 这两个命令都是查看当前系统正在运行进程，两者的区别是展示格式不同。如果想要查看特定的进程可以使用这样的格式：`ps aux|grep redis` (查看包括redis字符串的进程)，也可使用 `pgrep redis -a`。

注意：如果直接用ps ((Process Status)) 命令，会显示所有进程的状态，通常结合grep命令查看某进程的状态。

- `kill -9 进程的pid`: 杀死进程 (-9 表示强制终止。)

先用ps查找进程，然后用kill杀掉

- **网络通信命令：**

- 查看当前系统的网卡信息：ifconfig
- 查看与某台机器的连接情况：ping
- 查看当前系统的端口使用：netstat -an

- **net-tools 和 iproute2 :**

`net-tools` 起源于BSD的TCP/IP工具箱，后来成为老版本Linux内核中配置网络功能的工具。但自2001年起，Linux社区已经对其停止维护。同时，一些Linux发行版比如Arch Linux和CentOS/RHEL 7则已经完全抛弃了net-tools，只支持 iproute2。linux ip命令类似于ifconfig，但功能更强大，旨在替代它。

- `shutdown : shutdown -h now`: 指定现在立即关机；`shutdown +5 "System will shutdown after 5 minutes"`: 指定5分钟后关机，同时送出警告信息给登入用户。
- `reboot : reboot`: 重开机。`reboot -w`: 做个重开机的模拟（只有纪录并不会真的重开机）。

四、数据结构与算法

(一). 数据结构(布隆过滤器)

海量数据处理以及缓存穿透这两个场景让我认识了 布隆过滤器，我查阅了一些资料来了解它，但是很多现成资料并不满足我的需求，所以就决定自己总结一篇关于布隆过滤器的文章。希望通过这篇文章让更多人了解布隆过滤器，并且会实际去使用它！

下面我们将分为几个方面来介绍布隆过滤器：

1. 什么是布隆过滤器？
2. 布隆过滤器的原理介绍。
3. 布隆过滤器使用场景。
4. 通过 Java 编程手动实现布隆过滤器。
5. 利用 Google 开源的 Guava 中自带的布隆过滤器。
6. Redis 中的布隆过滤器。

1. 什么是布隆过滤器？

首先，我们需要了解布隆过滤器的概念。

布隆过滤器（Bloom Filter）是一个叫做 Bloom 的老哥于1970年提出的。我们可以把它看作由二进制向量（或者说位数组）和一系列随机映射函数（哈希函数）两部分组成的数据结构。相比于我们平时常用的 List、Map、Set 等数据结构，它占用空间更少并且效率更高，但是缺点是其返回的结果是概率性的，而不是非常准确的。理论情况下添加到集合中的元素越多，误报的可能性就越大。并且，存放在布隆过滤器的数据不容易删除。

bit数组

0	0	0	0	0	0
---	---	---	---	---	---

位数组中的每个元素都只占用 1 bit，并且每个元素只能是 0 或者 1。这样申请一个 100w 个元素的位数组只占用 $1000000 \text{ Bit} / 8 = 125000 \text{ Byte} = 125000 / 1024 \text{ kb} \approx 122 \text{ kb}$ 的空间。

总结：一个名叫 Bloom 的人提出了一种来检索元素是否在给定大集合中的数据结构，这种数据结构是高效且性能很好的，但缺点是具有一定的错误识别率和删除难度。并且，理论情况下，添加到集合中的元素越多，误报的可能性就越大。

2. 布隆过滤器的原理介绍

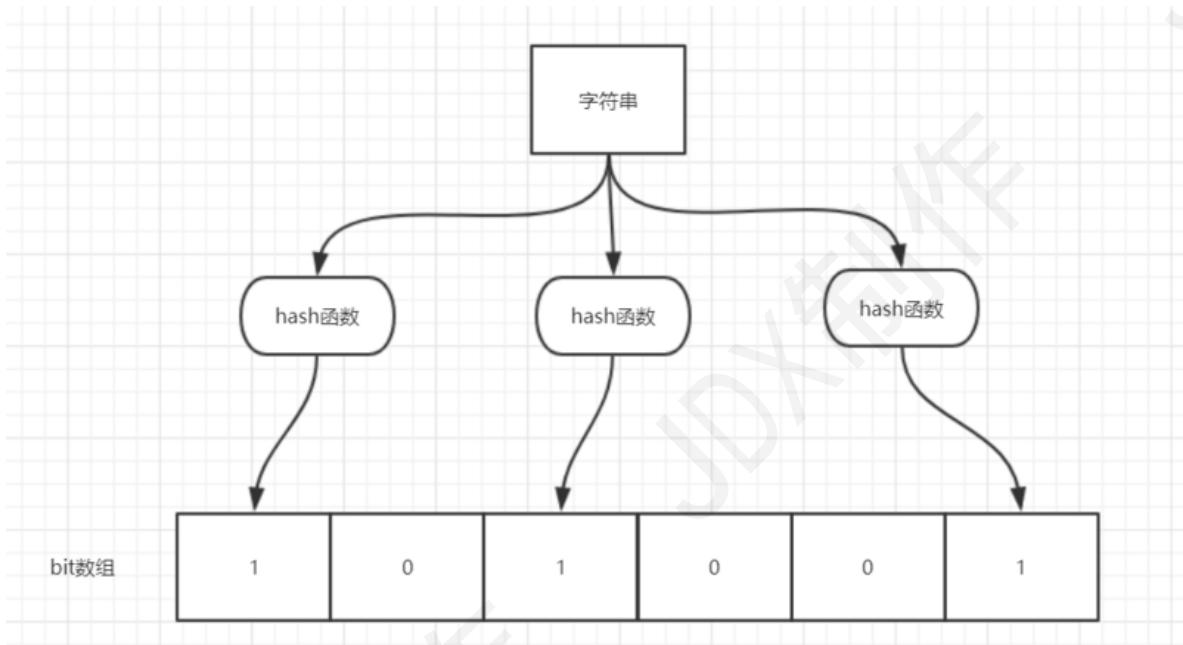
当一个元素加入布隆过滤器中的时候，会进行如下操作：

1. 使用布隆过滤器中的哈希函数对元素值进行计算，得到哈希值（有几个哈希函数得到几个哈希值）。
2. 根据得到的哈希值，在位数组中把对应下标的值置为 1。

当我们需要判断一个元素是否存在于布隆过滤器的时候，会进行如下操作：

1. 定元素再次进行相同的哈希计算；
2. 后判断位数组中的每个元素是否都为 1，如果值都为 1，那么说明这个值在布隆过滤器中，如果存在一个值不为 1，说明该元素不在布隆过滤器中。

举个简单的例子：



如图所示，当字符串存储要加入到布隆过滤器中时，该字符串首先由多个哈希函数生成不同的哈希值，然后在对应的位数组的下表的元素设置为 1（当位数组初始化时，所有位置均为 0）。当第二次存储同字符串时，因为先前的对应位置已设置为 1，所以很容易知道此值已经存在（去重非常方便）。

如果我们需要判断某个字符串是否在布隆过滤器中时，只需要对给定字符串再次进行相同的哈希计算，得到值之后判断位数组中的每个元素是否都为 1，如果值都为 1，那么说明这个值在布隆过滤器中，如果存在一个值不为 1，说明该元素不在布隆过滤器中。

不同的字符串可能哈希出来的位置相同，这种情况我们可以适当增加位数组大小或者调整我们的哈希函数。

综上，我们可以得出：**布隆过滤器说某个元素存在，小概率会误判。布隆过滤器说某个元素不在，那么这个元素一定不在。**

3. 布隆过滤器使用场景

1. 判断给定数据是否存在：比如判断一个数字是否在于包含大量数字的数字集中（数字集很大，5亿以上！）、防止缓存穿透（判断请求的数据是否有效避免直接绕过缓存请求数据库）等等、邮箱的垃圾邮件过滤、黑名单功能等等。
2. 去重：比如爬给定网址的时候对已经爬取过的 URL 去重。

4. 通过 Java 编程手动实现布隆过滤器

我们上面已经说了布隆过滤器的原理，知道了布隆过滤器的原理之后就可以自己手动实现一个了。

如果你想要手动实现一个的话，你需要：

1. 一个合适大小的位数组保存数据
2. 几个不同的哈希函数
3. 添加元素到位数组（布隆过滤器）的方法实现
4. 判断给定元素是否存在于位数组（布隆过滤器）的方法实现。

下面给出一个我觉得写的还算不错的代码（参考网上已有代码改进得到，对于所有类型对象皆适用）：

```

import java.util.BitSet;
public class MyBloomFilter {
    /**
     * 位数组的大小
     */
    
```

```
private static final int DEFAULT_SIZE = 2 << 24;
/**
 * 通过这个数组可以创建 6 个不同的哈希函数
 */
private static final int[] SEEDS = new int[]{3, 13, 46, 71, 91, 134};
/**
 * 位数组。数组中的元素只能是 0 或者 1
 */
private BitSet bits = new BitSet(DEFAULT_SIZE);
/**
 * 存放包含 hash 函数的类的数组
 */
private SimpleHash[] func = new SimpleHash[SEEDS.length];
/**
 * 初始化多个包含 hash 函数的类的数组，每个类中的 hash 函数都不一样
 */
public MyBloomFilter() {
    // 初始化多个不同的 Hash 函数
    for (int i = 0; i < SEEDS.length; i++) {
        func[i] = new SimpleHash(DEFAULT_SIZE, SEEDS[i]);
    }
}
/**
 * 添加元素到位数组
 */
public void add(Object value) {
    for (SimpleHash f : func) {
        bits.set(f.hash(value), true);
    }
}
/**
 * 判断指定元素是否存在于位数组
 */
public Boolean contains(Object value) {
    Boolean ret = true;
    for (SimpleHash f : func) {
        ret = ret && bits.get(f.hash(value));
    }
    return ret;
}
/**
 * 静态内部类。用于 hash 操作！
 */
public static class SimpleHash {
    private int cap;
    private int seed;
    public SimpleHash(int cap, int seed) {
        this.cap = cap;
        this.seed = seed;
    }
    /**
     * 计算 hash 值
     */
    public int hash(Object value) {
        int h;
        return (value == null) ? 0 : Math.abs(seed * (cap - 1) & ((h =
value.hashCode()) ^ (h >> 16)));
    }
}
```

```
    }  
}
```

测试：

```
String value1 = "https://javaguide.cn/";  
String value2 = "https://github.com/snailclimb";  
MyBloomFilter filter = new MyBloomFilter();  
System.out.println(filter.contains(value1));  
System.out.println(filter.contains(value2));  
filter.add(value1);  
filter.add(value2);  
System.out.println(filter.contains(value1));  
System.out.println(filter.contains(value2));
```

Output:

```
false  
false  
true  
true
```

测试：

```
Integer value1 = 13423;  
Integer value2 = 22131;  
MyBloomFilter filter = new MyBloomFilter();  
System.out.println(filter.contains(value1));  
System.out.println(filter.contains(value2));  
filter.add(value1);  
filter.add(value2);  
System.out.println(filter.contains(value1));  
System.out.println(filter.contains(value2));
```

Output:

```
false  
false  
true  
true
```

5.利用Google开源的 Guava中自带的布隆过滤器

自己实现的目的主要是为了让自己搞懂布隆过滤器的原理，Guava 中布隆过滤器的实现算是比较权威的，所以实际项目中我们不需要手动实现一个布隆过滤器。

首先我们需要在项目中引入 Guava 的依赖：

```
<dependency>  
  <groupId>com.google.guava</groupId>  
  <artifactId>guava</artifactId>  
  <version>28.0-jre</version>  
</dependency>
```

实际使用如下：

我们创建了一个最多存放 最多 1500个整数的布隆过滤器，并且我们可以容忍误判的概率为百分之 (0.01)

```
// 创建布隆过滤器对象
BloomFilter<Integer> filter = BloomFilter.create(
    Funnels.integerFunnel(),
    1500,
    0.01);

// 判断指定元素是否存在
System.out.println(filter.mightContain(1));
System.out.println(filter.mightContain(2));
// 将元素添加进布隆过滤器
filter.put(1);
filter.put(2);
System.out.println(filter.mightContain(1));
System.out.println(filter.mightContain(2));
```

在我们的示例中，当 `mightContain()` 方法返回 `true` 时，我们可以 99% 确定该元素在过滤器中，当过滤器返回 `false` 时，我们可以 100% 确定该元素不存在于过滤器中。

Guava 提供的布隆过滤器的实现还是很不错的（想要详细了解的可以看一下它的源码实现），但是它有一个重大的缺陷就是只能单机使用（另外，容量扩展也不容易），而现在互联网一般都是分布式的场景。为了解决这个问题，我们就需要用到 Redis 中的布隆过滤器了。

6.Redis 中的布隆过滤器

6.1 介绍

Redis v4.0 之后有了 Module（模块/插件）功能，Redis Modules 让 Redis 可以使用外部模块扩展其功能。布隆过滤器就是其中的 Module。

另外，官网推荐了一个 RedisBloom 作为 Redis 布隆过滤器的 Module，地址：<https://github.com/RedisBloom/RedisBloom>。其他还有：

- redis-lua-scaling-bloom-filter (lua 脚本实现) : <https://github.com/erikdubbelboer/redis-lua-scaling-bloom-filter>
- pyreBloom (Python中的快速Redis 布隆过滤器) : <https://github.com/seomoz/pyreBloom>
-

RedisBloom 提供了多种语言的客户端支持，包括：Python、Java、JavaScript 和 PHP。

6.2 使用Docker安装

具体操作如下：

```
→ ~ docker run -p 6379:6379 --name redis-redisbloom redislabs/redisbloom:latest
→ ~ docker exec -it redis-redisbloom bash
root@21396d02c252:/data# redis-cli
127.0.0.1:6379>
```

6.3 常用命令一览

注意： key:布隆过滤器的名称， item : 添加的元素。

1. **BF.ADD**: 将元素添加到布隆过滤器中，如果该过滤器尚不存在，则创建该过滤器。格式：
`BF.ADD {key} {item}`。
2. **BF.MADD**: 将一个或多个元素添加到“布隆过滤器”中，并创建一个尚不存在的过滤器。该命令的操作方式 **BF.ADD** 与之相同，只不过它允许多个输入并返回多个值。格式：
`BF.MADD {key} {item} [item ...]`。
3. **BF.EXISTS** : 确定元素是否在布隆过滤器中存在。格式：
`BF.EXISTS {key} {item}`。
4. **BF.MEXISTS** : 确定一个或者多个元素是否在布隆过滤器中存在格式：
`BF.MEXISTS {key} {item} [item ...]`。

另外，**BF.RESERVE** 命令需要单独介绍一下：

这个命令的格式如下：

```
BF.RESERVE {key} {error_rate} {capacity} [EXPANSION expansion]
```

下面简单介绍一下每个参数的具体含义：

1. key: 布隆过滤器的名称
2. error_rate :误报的期望概率。这应该是介于0到1之间的十进制值。例如，对于期望的误报率 0.1% (1000中为1)，error_rate应该设置为0.001。该数字越接近零，则每个项目的内存消耗越大，并且每个操作的CPU使用率越高。
3. capacity: 过滤器的容量。当实际存储的元素个数超过这个值之后，性能将开始下降。实际的降级将取决于超出限制的程度。随着过滤器元素数量呈指数增长，性能将线性下降。

可选参数：

- expansion: 如果创建了一个新的子过滤器，则其大小将是当前过滤器的大小乘以 expansion。默认扩展值为2。这意味着每个后续子过滤器将是前一个子过滤器的两倍。

6.4 实际使用

```
127.0.0.1:6379> BF.ADD myFilter java
(integer) 1
127.0.0.1:6379> BF.ADD myFilter javaguide
(integer) 1
127.0.0.1:6379> BF.EXISTS myFilter java
(integer) 1
127.0.0.1:6379> BF.EXISTS myFilter javaguide
(integer) 1
127.0.0.1:6379> BF.EXISTS myFilter github
(integer) 0
```

(二). 算法

五、数据库

(一). MySQL

1. 基本操作

获取更多资源可添加关注微信公众号：JAVA架构进阶之路；或添加微信：jiang10086a 免费获取面试真题，视频教程，笔记等。

```
/* Windows服务 */
-- 启动MySQL
    net start mysql
-- 创建Windows服务
    sc create mysql binPath= mysql\bin_path(注意：等号与值之间有空格)
/* 连接与断开服务器 */
mysql -h 地址 -P 端口 -u 用户名 -p 密码
SHOW PROCESSLIST -- 显示哪些线程正在运行
SHOW VARIABLES -- 显示系统变量信息
```

2. 数据库操作

```
/* 数据库操作 */
-- 查看当前数据库
    SELECT DATABASE();
-- 显示当前时间、用户名、数据库版本
    SELECT now(), user(), version();
-- 创建库
    CREATE DATABASE[ IF NOT EXISTS] 数据库名 数据库选项
    数据库选项：
        CHARACTER SET charset_name
        COLLATE collation_name
-- 查看已有库
    SHOW DATABASES[ LIKE 'PATTERN']
-- 查看当前库信息
    SHOW CREATE DATABASE 数据库名
-- 修改库的选项信息
    ALTER DATABASE 库名 选项信息
-- 删除库
    DROP DATABASE[ IF EXISTS] 数据库名
    同时删除该数据库相关的目录及其目录内容
```

3. 表的操作

```
-- 创建表
CREATE [TEMPORARY] TABLE[ IF NOT EXISTS] [库名.]表名 (表的结构定义)[ 表选项]
    每个字段必须有数据类型
    最后一个字段后不能有逗号
    TEMPORARY 临时表，会话结束时表自动消失
    对于字段的定义：
        字段名 数据类型 [NOT NULL | NULL] [DEFAULT default_value]
        [AUTO_INCREMENT] [UNIQUE [KEY] | [PRIMARY] KEY] [COMMENT 'string']
-- 表选项
-- 字符集
    CHARSET = charset_name
    如果表没有设定，则使用数据库字符集
-- 存储引擎
    ENGINE = engine_name
    表在管理数据时采用的不同的数据结构，结构不同会导致处理方式、提供的特性操作等不同
    常见的引擎：InnoDB MyISAM Memory/Heap BDB Merge Example CSV MaxDB Archive
    不同的引擎在保存表的结构和数据时采用不同的方式
    MyISAM表文件含义：.frm表定义，.MYD表数据，.MYI表索引
    InnoDB表文件含义：.frm表定义，表空间数据和日志文件
    SHOW ENGINES -- 显示存储引擎的状态信息
    SHOW ENGINE 引擎名 {LOGS|STATUS} -- 显示存储引擎的日志或状态信息
```

```
-- 自增起始数
    AUTO_INCREMENT = 行数
-- 数据文件目录
    DATA DIRECTORY = '目录'
-- 索引文件目录
    INDEX DIRECTORY = '目录'
-- 表注释
    COMMENT = 'string'
-- 分区选项
    PARTITION BY ... (详细见手册)
-- 查看所有表
    SHOW TABLES[ LIKE 'pattern']
    SHOW TABLES FROM 库名
-- 查看表结构
    SHOW CREATE TABLE 表名 (信息更详细)
    DESC 表名 / DESCRIBE 表名 / EXPLAIN 表名 / SHOW COLUMNS FROM 表名 [LIKE
'PATTERN']
    SHOW TABLE STATUS [FROM db_name] [LIKE 'pattern']
-- 修改表
-- 修改表本身的选项
    ALTER TABLE 表名 表的选项
    eg: ALTER TABLE 表名 ENGINE=MYISAM;
-- 对表进行重命名
    RENAME TABLE 原表名 TO 新表名
    RENAME TABLE 原表名 TO 库名.表名 (可将表移动到另一个数据库)
-- RENAME可以交换两个表名
-- 修改表的字段机构 (13.1.2. ALTER TABLE语法)
    ALTER TABLE 表名 操作名
-- 操作名
    ADD[ COLUMN] 字段定义      -- 增加字段
        AFTER 字段名          -- 表示增加在该字段名后面
        FIRST                  -- 表示增加在第一个
    ADD PRIMARY KEY(字段名)   -- 创建主键
    ADD UNIQUE [索引名] (字段名) -- 创建唯一索引
    ADD INDEX [索引名] (字段名) -- 创建普通索引
    DROP[ COLUMN] 字段名     -- 删除字段
    MODIFY[ COLUMN] 字段名 字段属性   -- 支持对字段属性进行修改, 不能修改字段名
(所有原有属性也需写上)
        CHANGE[ COLUMN] 原字段名 新字段名 字段属性      -- 支持对字段名修改
        DROP PRIMARY KEY       -- 删除主键(删除主键前需删除其AUTO_INCREMENT属性)
        DROP INDEX 索引名     -- 删除索引
        DROP FOREIGN KEY 外键  -- 删除外键
-- 删除表
    DROP TABLE[ IF EXISTS] 表名 ...
-- 清空表数据
    TRUNCATE [TABLE] 表名
-- 复制表结构
    CREATE TABLE 表名 LIKE 要复制的表名
-- 复制表结构和数据
    CREATE TABLE 表名 [AS] SELECT * FROM 要复制的表名
-- 检查表是否有错误
    CHECK TABLE tb1_name [, tb1_name] ... [option] ...
-- 优化表
    OPTIMIZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE tb1_name [, tb1_name] ...
-- 修复表
    REPAIR [LOCAL | NO_WRITE_TO_BINLOG] TABLE tb1_name [, tb1_name] ... [QUICK]
[EXTENDED] [USE_FRM]
-- 分析表
```

```
ANALYZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE tbl_name [, tbl_name] ...
```

4. 数据操作

```
/* 数据操作 */
-- 增
INSERT [INTO] 表名 [(字段列表)] VALUES (值列表)[, (值列表), ...]
-- 如果要插入的值列表包含所有字段并且顺序一致，则可以省略字段列表。
-- 可同时插入多条数据记录！
REPLACE 与 INSERT 完全一样，可互换。
INSERT [INTO] 表名 SET 字段名=值[, 字段名=值, ...]

-- 查
SELECT 字段列表 FROM 表名[ 其他子句]
-- 可来自多个表的多个字段
-- 其他子句可以不使用
-- 字段列表可以用*代替，表示所有字段

-- 删
DELETE FROM 表名[ 删除条件子句]
没有条件子句，则会删除全部

-- 改
UPDATE 表名 SET 字段名=新值[, 字段名=新值] [更新条件]
```

5. 字符集编码

```
/* 字符集编码 */
-- MySQL、数据库、表、字段均可设置编码
-- 数据编码与客户端编码不需一致
SHOW VARIABLES LIKE 'character_set_%' -- 查看所有字符集编码项
character_set_client          客户端向服务器发送数据时使用的编码
character_set_results         服务器端将结果返回给客户端所使用的编码
character_set_connection      连接层编码

SET 变量名 = 变量值
SET character_set_client = gbk;
SET character_set_results = gbk;
SET character_set_connection = gbk;
SET NAMES GBK; -- 相当于完成以上三个设置

-- 校对集
校对集用以排序
SHOW CHARACTER SET [LIKE 'pattern']/SHOW CHARSET [LIKE 'pattern']   查看所有字符集
SHOW COLLATION [LIKE 'pattern']           查看所有校对集
CHARSET 字符集编码       设置字符集编码
COLLATE 校对集编码       设置校对集编码
```

6. 数据类型(列类型)

```
/* 数据类型(列类型) */
1. 数值类型
-- a. 整型
    类型      字节      范围(有符号位)
    tinyint   1字节    -128 ~ 127     无符号位: 0 ~ 255
    smallint  2字节    -32768 ~ 32767
    mediumint 3字节   -8388608 ~ 8388607
    int       4字节
```

bigint 8字节
int(M) M表示总位数
- 默认存在符号位, **unsigned** 属性修改
- 显示宽度, 如果某个数不够定义字段时设置的位数, 则前面以0补填, **zerofill** 属性修改
 例: **int(5)** 插入一个数'123', 补填后为'00123'
- 在满足要求的情况下, 越小越好。
- 1表示bool值真, 0表示bool值假。MySQL没有布尔类型, 通过整型0和1表示。常用**tinyint(1)**表示布尔型。

-- b. 浮点型 -----

类型	字节	范围
float (单精度)	4字节	
double (双精度)	8字节	

浮点型既支持符号位 **unsigned** 属性, 也支持显示宽度 **zerofill** 属性。

不同于整型, 前后均会补填0。

定义浮点型时, 需指定总位数和小数位数。

float(M, D) **double(M, D)**

M表示总位数, D表示小数位数。

M和D的大小会决定浮点数的范围。不同于整型的固定范围。

M既表示总位数(不包括小数点和正负号), 也表示显示宽度(所有显示符号均包括)。

支持科学计数法表示。

浮点数表示近似值。

-- c. 定点数 -----

decimal -- 可变长度

decimal(M, D) M也表示总位数, D表示小数位数。

保存一个精确的数值, 不会发生数据的改变, 不同于浮点数的四舍五入。

将浮点数转换为字符串来保存, 每9位数字保存为4个字节。

2. 字符串类型

-- a. **char, varchar** -----

char 定长字符串, 速度快, 但浪费空间

varchar 变长字符串, 速度慢, 但节省空间

M表示能存储的最大长度, 此长度是字符数, 非字节数。

不同的编码, 所占用的空间不同。

char, 最多255个字符, 与编码无关。

varchar, 最多65535字符, 与编码有关。

一条有效记录最大不能超过65535个字节。

utf8 最大为21844个字符, **gbk** 最大为32766个字符, **latin1** 最大为65532个字符

varchar 是变长的, 需要利用存储空间保存 **varchar** 的长度, 如果数据小于255个字节, 则采用一个字节来保存长度, 反之需要两个字节来保存。

varchar 的最大有效长度由最大行大小和使用的字符集确定。

最大有效长度是65532字节, 因为在**varchar**存字符串时, 第一个字节是空的, 不存在任何数据, 然后还需两个字节来存放字符串的长度, 所以有效长度是 $65535 - 1 - 2 = 65532$ 字节。

例: 若一个表定义为 **CREATE TABLE tb(c1 int, c2 char(30), c3 varchar(N))**

charset=utf8; 问N的最大值是多少? 答: **(65535-1-2-4-30*3)/3**

-- b. **blob, text** -----

blob 二进制字符串(字节字符串)

tinyblob, blob, mediumblob, longblob

text 非二进制字符串(字符字符串)

tinytext, text, mediumtext, longtext

text 在定义时, 不需要定义长度, 也不会计算总长度。

text 类型在定义时, 不可给**default**值

-- c. **binary, varbinary** -----

类似于**char**和**varchar**, 用于保存二进制字符串, 也就是保存字节字符串而非字符字符串。

char, varchar, text 对应 **binary, varbinary, blob**.

3. 日期时间类型

一般用整型保存时间戳, 因为PHP可以很方便的将时间戳进行格式化。

datetime 8字节 日期及时间 1000-01-01 00:00:00 到 9999-12-31 23:59:59

date 3字节 日期 1000-01-01 到 9999-12-31

timestamp 4字节 时间戳 19700101000000 到 2038-01-19 03:14:07

<code>time</code>	3字节	时间	-838:59:59 到 838:59:59
<code>year</code>	1字节	年份	1901 - 2155
<code>datetime</code>	YYYY-MM-DD hh:mm:ss		
<code>timestamp</code>	YY-MM-DD hh:mm:ss		
	YYYYMMDDhhmmss		
	YYMMDDhhmmss		
	YYYYYMMDDhhmmss		
	YYMMDDhhmmss		
<code>date</code>	YYYY-MM-DD		
	YY-MM-DD		
	YYYYMMDD		
	YYMMDD		
	YYYYMMDD		
	YYMMDD		
<code>time</code>	hh:mm:ss		
	hhmmss		
	hhmmss		
<code>year</code>	YYYY		
	YY		
	YYYY		
	YY		

4. 枚举和集合

-- 枚举(enum) -----
`enum(val1, val2, val3...)`

在已知的值中进行单选。最大数量为65535。

枚举值在保存时，以2个字节的整型(smallint)保存。每个枚举值，按保存的位置顺序，从1开始逐一递增。

表现为字符串类型，存储却是整型。

`NULL`值的索引是NULL。

空字符串错误值的索引值是0。

-- 集合(set) -----
`set(val1, val2, val3...)`

`create table tab (gender set('男', '女', '无'));`
`insert into tab values ('男, 女');`

最多可以有64个不同的成员。以bigint存储，共8个字节。采取位运算的形式。

当创建表时，SET成员值的尾部空格将自动被删除。

7. 列属性(列约束)

/* 列属性(列约束) */ -----

1. PRIMARY 主键

- 能唯一标识记录的字段，可以作为主键。
- 一个表只能有一个主键。
- 主键具有唯一性。
- 声明字段时，用 `primary key` 标识。

也可以在字段列表之后声明

- 例：`create table tab (id int, stu varchar(10), primary key (id));`
- 主键字段的值不能为null。
 - 主键可以由多个字段共同组成。此时需要在字段列表后声明的方法。

例：`create table tab (id int, stu varchar(10), age int, primary key (stu, age));`

2. UNIQUE 唯一索引(唯一约束)

使得某字段的值也不能重复。

3. NULL 约束

`null`不是数据类型，是列的一个属性。

表示当前列是否可以为null，表示什么都没有。

`null`，允许为空。默认。

```
not null, 不允许为空。  
insert into tab values (null, 'val');  
-- 此时表示将第一个字段的值设为null, 取决于该字段是否允许为null  
4. DEFAULT 默认值属性  
当前字段的默认值。  
insert into tab values (default, 'val'); -- 此时表示强制使用默认值。  
create table tab ( add_time timestamp default current_timestamp );  
-- 表示将当前时间的时间戳设为默认值。  
current_date, current_time
```

5. AUTO_INCREMENT 自动增长约束
自动增长必须为索引（主键或unique）
只能存在一个字段为自动增长。
默认为1开始自动增长。可以通过表属性 `auto_increment = x` 进行设置，或 `alter table tbl auto_increment = x;`

6. COMMENT 注释
例: `create table tab (id int) comment '注释内容';`

7. FOREIGN KEY 外键约束
用于限制主表与从表数据完整性。
`alter table t1 add constraint `t1_t2_fk` foreign key (t1_id) references t2(id);`

-- 将表t1的t1_id外键关联到表t2的id字段。

-- 每个外键都有一个名字，可以通过 `constraint` 指定

存在外键的表，称之为从表（子表），外键指向的表，称之为父表。

作用：保持数据一致性，完整性，主要目的是控制存储在外键表（从表）中的数据。

MySQL中，可以对InnoDB引擎使用外键约束：

语法：

`foreign key` (外键字段) `references` 主表名 (关联字段) [主表记录删除时的动作] [主表记录更新时的动作]

此时需要检测一个从表的外键需要约束为主表的已存在的值。外键在没有关联的情况下，可以设置为 `null`。前提是该外键列，没有 `not null`。

可以不指定主表记录更改或更新时的动作，那么此时主表的操作被拒绝。

如果指定了 `on update` 或 `on delete`: 在删除或更新时，有如下几个操作可以选择：

1. `cascade`, 级联操作。主表数据被更新（主键值更新），从表也被更新（外键值更新）。主表记录被删除，从表相关记录也被删除。

2. `set null`, 设置为 `null`。主表数据被更新（主键值更新），从表的外键被设置为 `null`。主表记录被删除，从表相关记录外键被设置成 `null`。但注意，要求该外键列，没有 `not null` 属性约束。

3. `restrict`, 拒绝父表删除和更新。

注意，外键只被InnoDB存储引擎所支持。其他引擎是不支持的。

8. 建表规范

```
/* 建表规范 */ -----  
-- Normal Format, NF  
- 每个表保存一个实体信息  
- 每个具有一个ID字段作为主键  
- ID主键 + 原子表  
-- 1NF, 第一范式  
    字段不能再分，就满足第一范式。  
-- 2NF, 第二范式  
    满足第一范式的前提下，不能出现部分依赖。  
    消除复合主键就可以避免部分依赖。增加单列关键字。  
-- 3NF, 第三范式  
    满足第二范式的前提下，不能出现传递依赖。  
    某个字段依赖于主键，而有其他字段依赖于该字段。这就是传递依赖。  
    将一个实体信息的数据放在一个表内实现。
```

9. SELECT

```
/* SELECT */ -----
SELECT [ALL|DISTINCT] select_expr FROM -> WHERE -> GROUP BY [合计函数] -> HAVING
-> ORDER BY -> LIMIT
a. select_expr
    -- 可以用 * 表示所有字段。
    select * from tb;
    -- 可以使用表达式（计算公式、函数调用、字段也是个表达式）
    select stu, 29+25, now() from tb;
    -- 可以为每个列使用别名。适用于简化列标识，避免多个列标识符重复。
    - 使用 as 关键字，也可省略 as。
    select stu+10 as add10 from tb;
b. FROM 子句
    用于标识查询来源。
    -- 可以为表起别名。使用as关键字。
    SELECT * FROM tb1 AS tt, tb2 AS bb;
    -- from子句后，可以同时出现多个表。
    -- 多个表会横向叠加到一起，而数据会形成一个笛卡尔积。
    SELECT * FROM tb1, tb2;
    -- 向优化符提示如何选择索引
    USE INDEX、IGNORE INDEX、FORCE INDEX
    SELECT * FROM table1 USE INDEX (key1,key2) WHERE key1=1 AND key2=2 AND
key3=3;
    SELECT * FROM table1 IGNORE INDEX (key3) WHERE key1=1 AND key2=2 AND
key3=3;
c. WHERE 子句
    -- 从from获得的数据源中进行筛选。
    -- 整型1表示真，0表示假。
    -- 表达式由运算符和运算数组成。
        -- 运算数：变量（字段）、值、函数返回值
        -- 运算符：
            =, <=>, <>, !=, <=, <, >=, >, !, &&, ||,
            in (not) null, (not) like, (not) in, (not) between and, is (not),
            and, or, not, xor
            is/is not 加上ture/false/unknown, 检验某个值的真假
            <=>与<>功能相同, <=>可用于null比较
d. GROUP BY 子句, 分组子句
    GROUP BY 字段/别名 [排序方式]
    分组后会进行排序。升序: ASC, 降序: DESC
    以下[合计函数]需配合 GROUP BY 使用:
    count 返回不同的非NULL值数目  count(*)、count(字段)
    sum 求和
    max 求最大值
    min 求最小值
    avg 求平均值
    group_concat 返回带有来自一个组的连接的非NULL值的字符串结果。组内字符串连接。
e. HAVING 子句, 条件子句
    与 where 功能、用法相同，执行时机不同。
    where 在开始时执行检测数据，对原数据进行过滤。
    having 对筛选出的结果再次进行过滤。
    having 字段必须是查询出来的，where 字段必须是数据表存在的。
    where 不可以使用字段的别名，having 可以。因为执行WHERE代码时，可能尚未确定列值。
    where 不可以使用合计函数。一般需用合计函数才会用 having
    SQL标准要求HAVING必须引用GROUP BY子句中的列或用于合计函数中的列。
f. ORDER BY 子句, 排序子句
    order by 排序字段/别名 排序方式 [,排序字段/别名 排序方式]...
```

升序: ASC, 降序: DESC

支持多个字段的排序。

g. **LIMIT** 子句, 限制结果数量子句

仅对处理好的结果进行数量限制。将处理好的结果看作是一个集合, 按照记录出现的顺序, 索引从0开始。

limit 起始位置, 获取条数

省略第一个参数, 表示从索引0开始。**limit** 获取条数

h. **DISTINCT, ALL** 选项

distinct 去除重复记录

默认为 **all**, 全部记录

10. UNION

/* UNION */ -----

将多个**select**查询的结果组合成一个结果集合。

SELECT ... UNION [ALL|DISTINCT] SELECT ...

默认 **DISTINCT** 方式, 即所有返回的行都是唯一的

建议, 对每个**SELECT**查询加上小括号包裹。

ORDER BY 排序时, 需加上 **LIMIT** 进行结合。

需要各**select**查询的字段数量一样。

每个**select**查询的字段列表(数量、类型)应一致, 因为结果中的字段名以第一条**select**语句为准。

11. 子查询

/* 子查询 */ -----

- 子查询需用括号包裹。

-- from型

from后要求是一个表, 必须给子查询结果取个别名。

- 简化每个查询内的条件。

- **from**型需将结果生成一个临时表格, 可用以原表的锁定的释放。

- 子查询返回一个表, 表型子查询。

select * from (select * from tb where id>0) as subfrom where id>1;

-- where型

- 子查询返回一个值, 标量子查询。

- 不需要给子查询取别名。

- **where**子查询内的表, 不能直接用以更新。

select * from tb where money = (select max(money) from tb);

-- 列子查询

如果子查询结果返回的是一列。

使用 **in** 或 **not in** 完成查询

exists 和 **not exists** 条件

如果子查询返回数据, 则返回1或0。常用于判断条件。

select column1 from t1 where exists (select * from t2);

-- 行子查询

查询条件是一个行。

select * from t1 where (id, gender) in (select id, gender from t2);

行构造符: (col1, col2, ...) 或 **ROW**(col1, col2, ...)

行构造符通常用于与对能返回两个或两个以上列的子查询进行比较。

-- 特殊运算符

!= all() 相当于 **not in**

= some() 相当于 **in**。any 是 **some** 的别名

!= some() 不等同于 **not in**, 不等于其中某一个。

all, some 可以配合其他运算符一起使用。

12. 连接查询(join)

```

/* 连接查询(join) */
    将多个表的字段进行连接，可以指定连接条件。
-- 内连接(inner join)
    - 默认就是内连接，可省略inner。
    - 只有数据存在时才能发送连接。即连接结果不能出现空行。
    on 表示连接条件。其条件表达式与where类似。也可以省略条件（表示条件永远为真）
    也可用where表示连接条件。
    还有 using，但需字段名相同。 using(字段名)
-- 交叉连接 cross join
    即，没有条件的内连接。
    select * from tb1 cross join tb2;
-- 外连接(outer join)
    - 如果数据不存在，也会出现在连接结果中。
    -- 左外连接 left join
        如果数据不存在，左表记录会出现，而右表为null填充
    -- 右外连接 right join
        如果数据不存在，右表记录会出现，而左表为null填充
-- 自然连接(natural join)
    自动判断连接条件完成连接。
    相当于省略了using，会自动查找相同字段名。
    natural join
    natural left join
    natural right join
select info.id, info.name, info.stu_num, extra_info.hobby, extra_info.sex from
info, extra_info where info.stu_num = extra_info.stu_id;

```

13. TRUNCATE

```

/* TRUNCATE */
TRUNCATE [TABLE] tbl_name
清空数据
删除重建表
区别：
1. truncate 是删除表再创建，delete 是逐条删除
2. truncate 重置auto_increment的值。而delete不会
3. truncate 不知道删除了几条，而delete知道。
4. 当被用于带分区的表时，truncate 会保留分区

```

14. 备份与还原

```

/* 备份与还原 */
备份，将数据的结构与表内数据保存起来。
利用 mysqldump 指令完成。
-- 导出
mysqldump [options] db_name [tables]
mysqldump [options] --database DB1 [DB2 DB3...]
mysqldump [options] --all--database
1. 导出一张表
    mysqldump -u用户名 -p密码 库名 表名 > 文件名(D:/a.sql)
2. 导出多张表
    mysqldump -u用户名 -p密码 库名 表1 表2 表3 > 文件名(D:/a.sql)
3. 导出所有表
    mysqldump -u用户名 -p密码 库名 > 文件名(D:/a.sql)
4. 导出一个库
    mysqldump -u用户名 -p密码 --lock-all-tables --database 库名 > 文件名(D:/a.sql)

```

可以-w携带WHERE条件

-- 导入

1. 在登录mysql的情况下:

source 备份文件

2. 在不登录的情况下

mysql -u用户名 -p密码 库名 < 备份文件

15. 视图

什么是视图:

视图是一个虚拟表，其内容由查询定义。同真实的表一样，视图包含一系列带有名称的列和行数据。但是，视图并不在数据库中以存储的数据值集形式存在。行和列数据来自由定义视图的查询所引用的表，并且在引用视图时动态生成。

视图具有表结构文件，但不存在数据文件。

对其中所引用的基础表来说，视图的作用类似于筛选。定义视图的筛选可以来自当前或其它数据库的一个或多个表，或者其它视图。通过视图进行查询没有任何限制，通过它们进行数据修改时的限制也很少。

视图是存储在数据库中的查询的sql语句，它主要出于两种原因：安全原因，视图可以隐藏一些数据，如：社会保险基金表，可以用视图只显示姓名，地址，而不显示社会保险号和工资数等，另一原因是可使复杂的查询易于理解和使用。

-- 创建视图

`CREATE [OR REPLACE] [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}] [VIEW view_name [(column_list)] AS select_statement`

- 视图名必须唯一，同时不能与表重名。
- 视图可以使用select语句查询到的列名，也可以自己指定相应的列名。
- 可以指定视图执行的算法，通过ALGORITHM指定。
- column_list如果存在，则数目必须等于SELECT语句检索的列数

-- 查看结构

`SHOW CREATE VIEW view_name`

-- 删除视图

- 删除视图后，数据依然存在。
- 可同时删除多个视图。

`DROP VIEW [IF EXISTS] view_name ...`

-- 修改视图结构

- 一般不修改视图，因为不是所有的更新视图都会映射到表上。

`ALTER VIEW view_name [(column_list)] AS select_statement`

-- 视图作用

1. 简化业务逻辑
2. 对客户端隐藏真实的表结构

-- 视图算法(ALGORITHM)

`MERGE` 合并

将视图的查询语句，与外部查询需要先合并再执行！

`TEMPTABLE` 临时表

将视图执行完毕后，形成临时表，再做外层查询！

`UNDEFINED` 未定义(默认)，指的是MySQL自主去选择相应的算法。

16. 事务(transaction)

事务是指逻辑上的一组操作，组成这组操作的各个单元，要不全成功要不全失败。

- 支持连续SQL的集体成功或集体撤销。
- 事务是数据库在数据完整性方面的一个功能。
- 需要利用 InnoDB 或 BDB 存储引擎，对自动提交的特性支持完成。
- InnoDB 被称为事务安全型引擎。

-- 事务开启

`START TRANSACTION;` 或者 `BEGIN;`

开启事务后，所有被执行的SQL语句均被认作当前事务内的SQL语句。

-- 事务提交

```
COMMIT;
-- 事务回滚
ROLLBACK;
如果部分操作发生问题，映射到事务开启前。
```

-- 事务的特性

1. 原子性 (Atomicity)
事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。
2. 一致性 (Consistency)
事务前后数据的完整性必须保持一致。
 - 事务开始和结束时，外部数据一致
 - 在整个事务过程中，操作是连续的
3. 隔离性 (Isolation)
多个用户并发访问数据库时，一个用户的事务不能被其它用户的事物所干扰，多个并发事务之间的数据要相互隔离。
4. 持久性 (Durability)
一个事务一旦被提交，它对数据库中的数据改变就是永久性的。

-- 事务的实现

1. 要求是事务支持的表类型
2. 执行一组相关操作前开启事务
3. 整组操作完成后，都成功，则提交；如果存在失败，选择回滚，则会回到事务开始的备份点。

-- 事务的原理

利用InnoDB的自动提交(**autocommit**)特性完成。

普通的MySQL执行语句后，当前的数据提交操作均可被其他客户端可见。

而事务是暂时关闭“自动提交”机制，需要**commit**提交持久化数据操作。

-- 注意

1. 数据定义语言 (DDL) 语句不能被回滚，比如创建或取消数据库的语句，和创建、取消或更改表或存储的子程序的语句。
2. 事务不能被嵌套

-- 保存点

SAVEPOINT 保存点名称 -- 设置一个事务保存点
ROLLBACK TO SAVEPOINT 保存点名称 -- 回滚到保存点
RELEASE SAVEPOINT 保存点名称 -- 删除保存点

-- InnoDB自动提交特性设置

SET autocommit = 0|1; 0表示关闭自动提交，1表示开启自动提交。

- 如果关闭了，那普通操作的结果对其他客户端也不可见，需要**commit**提交后才能持久化数据操作。
- 也可以关闭自动提交来开启事务。但与**START TRANSACTION**不同的是，**SET autocommit**是永久改变服务器的设置，直到下次再次修改该设置。(针对当前连接)
而**START TRANSACTION**记录开启前的状态，而一旦事务提交或回滚后就需要再次开启事务。(针对当前事务)

17. 锁表

```
/* 锁表 */
表锁定只用于防止其它客户端进行不正当地读取和写入
MyISAM 支持表锁，InnoDB 支持行锁
-- 锁定
LOCK TABLES tbl_name [AS alias]
-- 解锁
UNLOCK TABLES
```

18. 触发器

```
/* 触发器 */ -----
触发程序是与表有关的命名数据库对象，当该表出现特定事件时，将激活该对象
监听：记录的增加、修改、删除。
```

```

-- 创建触发器
CREATE TRIGGER trigger_name trigger_time trigger_event ON tbl_name FOR EACH ROW
trigger_stmt

参数:
trigger_time是触发程序的动作时间。它可以是 before 或 after, 以指明触发程序是在激活它的语句之前或之后触发。
trigger_event指明了激活触发程序的语句的类型
    INSERT: 将新行插入表时激活触发程序
    UPDATE: 更改某一行时激活触发程序
    DELETE: 从表中删除某一行时激活触发程序
tbl_name: 监听的表, 必须是永久性的表, 不能将触发程序与TEMPORARY表或视图关联起来。
trigger_stmt: 当触发程序激活时执行的语句。执行多个语句, 可使用BEGIN...END复合语句结构

-- 删除
DROP TRIGGER [schema_name.]trigger_name

可以使用old和new代替旧的和新的数据
更新操作, 更新前是old, 更新后是new.
删除操作, 只有old.
增加操作, 只有new.

-- 注意
1. 对于具有相同触发程序动作时间和事件的给定表, 不能有两个触发程序。

-- 字符连接函数
concat(str1,str2,...)
concat_ws(separator,str1,str2,...)

-- 分支语句
if 条件 then
    执行语句
elseif 条件 then
    执行语句
else
    执行语句
end if;

-- 修改最外层语句结束符
delimiter 自定义结束符号
    SQL语句
自定义结束符号
delimiter ;      -- 修改回原来的分号

-- 语句块包裹
begin
    语句块
end

-- 特殊的执行
1. 只要添加记录, 就会触发程序。
2. Insert into on duplicate key update 语法会触发:
如果没有重复记录, 会触发 before insert, after insert;
如果有重复记录并更新, 会触发 before insert, before update, after update;
如果有重复记录但是没有发生更新, 则触发 before insert, before update
3. Replace 语法 如果有记录, 则执行 before insert, before delete, after delete, after
insert

```

19. SQL编程

```

/* SQL编程 */
--// 局部变量
-- 变量声明
declare var_name[,...] type [default value]
这个语句被用来声明局部变量。要给变量提供一个默认值, 请包含一个default子句。值可以被指定为一个表达式, 不需要为一个常数。如果没有default子句, 初始值为null。

```

```

-- 赋值
使用 set 和 select into 语句为变量赋值。
- 注意：在函数内是可以使用全局变量（用户自定义的变量）

--// 全局变量 -----
-- 定义、赋值
set 语句可以定义并为变量赋值。
set @var = value;
也可以使用select into语句为变量初始化并赋值。这样要求select语句只能返回一行，但是可以是多个字段，就意味着同时为多个变量进行赋值，变量的数量需要与查询的列数一致。
还可以把赋值语句看作一个表达式，通过select执行完成。此时为了避免=被当作关系运算符看待，使用:=代替。（set语句可以使用= 和 :=）。
select @var:=20;
select @v1:=id, @v2=name from t1 limit 1;
select * from tbl_name where @var:=30;
select into 可以将表中查询获得的数据赋给变量。
-| select max(height) into @max_height from tb;

-- 自定义变量名
为了避免select语句中，用户自定义的变量与系统标识符（通常是字段名）冲突，用户自定义变量在变量名前使用@作为开始符号。
@var=10;
- 变量被定义后，在整个会话周期都有效（登录到退出）

--// 控制结构 -----
-- if语句
if search_condition then
    statement_list
[elseif search_condition then
    statement_list]
...
[else
    statement_list]
end if;
-- case语句
CASE value WHEN [compare-value] THEN result
[WHEN [compare-value] THEN result ...]
[ELSE result]
END
-- while循环
[begin_label:] while search_condition do
    statement_list
end while [end_label];
- 如果需要在循环内提前终止 while循环，则需要使用标签；标签需要成对出现。

-- 退出循环
    退出整个循环 leave
    退出当前循环 iterate
    通过退出的标签决定退出哪个循环

--// 内置函数 -----
-- 数值函数
abs(x)          -- 绝对值 abs(-10.9) = 10
format(x, d)    -- 格式化千分位数值 format(1234567.456, 2) = 1,234,567.46
ceil(x)         -- 向上取整 ceil(10.1) = 11
floor(x)        -- 向下取整 floor (10.1) = 10
round(x)         -- 四舍五入去整
mod(m, n)       -- m%n m mod n 求余 10%3=1
pi()            -- 获得圆周率
pow(m, n)       -- m^n
sqrt(x)         -- 算术平方根
rand()          -- 随机数
truncate(x, d) -- 截取d位小数

```

```

-- 时间日期函数
now(), current_timestamp();          -- 当前日期时间
current_date();                      -- 当前日期
current_time();                      -- 当前时间
date('yyyy-mm-dd hh:ii:ss');        -- 获取日期部分
time('yyyy-mm-dd hh:ii:ss');        -- 获取时间部分
date_format('yyyy-mm-dd hh:ii:ss', '%d %y %a %d %m %b %j'); -- 格式化时间
unix_timestamp();                   -- 获得unix时间戳
from_unixtime();                    -- 从时间戳获得时间

-- 字符串函数
length(string)                     -- string长度, 字节
char_length(string)                -- string的字符个数
substring(str, position [,length]) -- 从str的position开始, 取length个字符
replace(str ,search_str ,replace_str) -- 在str中用replace_str替换search_str
instr(string ,substring)           -- 返回substring首次在string中出现的位置
concat(string [...])              -- 连接字串
charset(str)                      -- 返回字串字符集
lcase(string)                     -- 转换成小写
left(string, length)              -- 从string2中的左边起取length个字符
load_file(file_name)              -- 从文件读取内容
locate(substring, string [,start_position]) -- 同instr, 但可指定开始位置
lpad(string, length, pad)         -- 重复用pad加在string开头, 直到字串长度为length
ltrim(string)                     -- 去除前端空格
repeat(string, count)             -- 重复count次
rpad(string, length, pad)         -- 在str后用pad补充, 直到长度为length
rtrim(string)                     -- 去除后端空格
strcmp(string1 ,string2)          -- 逐字符比较两字串大小

-- 流程函数
case when [condition] then result [when [condition] then result ...] [else
result] end    多分支
if(expr1,expr2,expr3)  双分支。
-- 聚合函数
count()
sum();
max();
min();
avg();
group_concat()

-- 其他常用函数
md5();
default();

-- // 存储函数, 自定义函数 -----
-- 新建
CREATE FUNCTION function_name (参数列表) RETURNS 返回值类型
    函数体
    - 函数名, 应该合法的标识符, 并且不应该与已有的关键字冲突。
    - 一个函数应该属于某个数据库, 可以使用db_name.funciton_name的形式执行当前函数所属数据
库, 否则为当前数据库。
    - 参数部分, 由"参数名"和"参数类型"组成。多个参数用逗号隔开。
    - 函数体由多条可用的mysql语句, 流程控制, 变量声明等语句构成。
    - 多条语句应该使用 begin...end 语句块包含。
    - 一定要有 return 返回值语句。
-- 删除
DROP FUNCTION [IF EXISTS] function_name;
-- 查看
SHOW FUNCTION STATUS LIKE 'partten'
SHOW CREATE FUNCTION function_name;
-- 修改

```

```
ALTER FUNCTION function_name 函数选项
--// 存储过程, 自定义功能 -----
-- 定义
存储存储过程 是一段代码(过程), 存储在数据库中的sql组成。
一个存储过程通常用于完成一段业务逻辑, 例如报名, 交班费, 订单入库等。
而一个函数通常专注与某个功能, 视为其他程序服务的, 需要在其他语句中调用函数才可以, 而存储过程不能被其他调用, 是自己执行 通过call执行。
-- 创建
CREATE PROCEDURE sp_name (参数列表)
    过程体
参数列表: 不同于函数的参数列表, 需要指明参数类型
IN, 表示输入型
OUT, 表示输出型
INOUT, 表示混合型
注意, 没有返回值。
```

20. 存储过程

```
/* 存储过程 */
存储过程是一段可执行性代码的集合。相比函数, 更偏向于业务逻辑。
调用: CALL 过程名
-- 注意
- 没有返回值。
- 只能单独调用, 不可夹杂在其他语句中
-- 参数
IN|OUT|INOUT 参数名 数据类型
IN      输入: 在调用过程中, 将数据输入到过程体内部的参数
OUT     输出: 在调用过程中, 将过程体处理完的结果返回到客户端
INOUT   输入输出: 既可输入, 也可输出
-- 语法
CREATE PROCEDURE 过程名 (参数列表)
BEGIN
    过程体
END
```

21. 用户和权限管理

```
/* 用户和权限管理 */
-- root密码重置
1. 停止MySQL服务
2. [Linux] /usr/local/mysql/bin/safe_mysqld --skip-grant-tables &
[Windows] mysqld --skip-grant-tables
3. use mysql;
4. UPDATE `user` SET PASSWORD=PASSWORD("密码") WHERE `user` = "root";
5. FLUSH PRIVILEGES;
用户信息表: mysql.user
-- 刷新权限
FLUSH PRIVILEGES;
-- 增加用户
CREATE USER 用户名 IDENTIFIED BY [PASSWORD] 密码(字符串)
    - 必须拥有mysql数据库的全局CREATE USER权限, 或拥有INSERT权限。
    - 只能创建用户, 不能赋予权限。
    - 用户名, 注意引号: 如 'user_name'@'192.168.1.1'
    - 密码也需引号, 纯数字密码也要加引号
    - 要在纯文本中指定密码, 需忽略PASSWORD关键词。要把密码指定为由PASSWORD()函数返回的混编
值, 需包含关键字PASSWORD
```

```
-- 重命名用户
RENAME USER old_user TO new_user
-- 设置密码
SET PASSWORD = PASSWORD('密码') -- 为当前用户设置密码
SET PASSWORD FOR 用户名 = PASSWORD('密码') -- 为指定用户设置密码
-- 删除用户
DROP USER 用户名
-- 分配权限/添加用户
GRANT 权限列表 ON 表名 TO 用户名 [IDENTIFIED BY [PASSWORD] 'password']
  - all privileges 表示所有权限
  - *.* 表示所有库的所有表
  - 库名.表名 表示某库下面的某表
  GRANT ALL PRIVILEGES ON `pms`.* TO 'pms'@'%' IDENTIFIED BY 'pms0817';
-- 查看权限
SHOW GRANTS FOR 用户名
  -- 查看当前用户权限
  SHOW GRANTS; 或 SHOW GRANTS FOR CURRENT_USER; 或 SHOW GRANTS FOR CURRENT_USER();
-- 撤消权限
REVOKE 权限列表 ON 表名 FROM 用户名
REVOKE ALL PRIVILEGES, GRANT OPTION FROM 用户名 -- 撤销所有权限
-- 权限层级
-- 要使用GRANT或REVOKE，您必须拥有GRANT OPTION权限，并且您必须用于您正在授予或撤销的权限。
全局层级：全局权限适用于一个给定服务器中的所有数据库，mysql.user
  GRANT ALL ON *.*和 REVOKE ALL ON *.*只授予和撤销全局权限。
数据库层级：数据库权限适用于一个给定数据库中的所有目标，mysql.db, mysql.host
  GRANT ALL ON db_name.*和REVOKE ALL ON db_name.*只授予和撤销数据库权限。
表层级：表权限适用于一个给定表中的所有列，mysql.tables_priv
  GRANT ALL ON db_name.tbl_name和REVOKE ALL ON db_name.tbl_name只授予和撤销表权限。
列层级：列权限适用于一个给定表中的单一列，mysql.columns_priv
  当使用REVOKE时，您必须指定与被授权列相同的列。
-- 权限列表
ALL [PRIVILEGES] -- 设置除GRANT OPTION之外的所有简单权限
ALTER -- 允许使用ALTER TABLE
ALTER ROUTINE -- 更改或取消已存储的子程序
CREATE -- 允许使用CREATE TABLE
CREATE ROUTINE -- 创建已存储的子程序
CREATE TEMPORARY TABLES -- 允许使用CREATE TEMPORARY TABLE
CREATE USER -- 允许使用CREATE USER, DROP USER, RENAME USER和REVOKE ALL PRIVILEGES。
CREATE VIEW -- 允许使用CREATE VIEW
DELETE -- 允许使用DELETE
DROP -- 允许使用DROP TABLE
EXECUTE -- 允许用户运行已存储的子程序
FILE -- 允许使用SELECT...INTO OUTFILE和LOAD DATA INFILE
INDEX -- 允许使用CREATE INDEX和DROP INDEX
INSERT -- 允许使用INSERT
LOCK TABLES -- 允许对您拥有SELECT权限的表使用LOCK TABLES
PROCESS -- 允许使用SHOW FULL PROCESSLIST
REFERENCES -- 未被实施
RELOAD -- 允许使用FLUSH
REPLICATION CLIENT -- 允许用户询问从属服务器或主服务器的地址
REPLICATION SLAVE -- 用于复制型从属服务器（从主服务器中读取二进制日志事件）
SELECT -- 允许使用SELECT
SHOW DATABASES -- 显示所有数据库
SHOW VIEW -- 允许使用SHOW CREATE VIEW
SHUTDOWN -- 允许使用mysqladmin shutdown
```

```
SUPER    -- 允许使用CHANGE MASTER, KILL, PURGE MASTER LOGS和SET GLOBAL语句,  
mysqladmin debug命令; 允许您连接(一次), 即使已达到max_connections。  
UPDATE   -- 允许使用UPDATE  
USAGE    -- “无权限”的同义词  
GRANT OPTION    -- 允许授予权限
```

22. 表维护

```
/* 表维护 */  
-- 分析和存储表的关键字分布  
ANALYZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE 表名 ...  
-- 检查一个或多个表是否有错误  
CHECK TABLE tbl_name [,tbl_name] ... [option] ...  
option = {QUICK | FAST | MEDIUM | EXTENDED | CHANGED}  
-- 整理数据文件的碎片  
OPTIMIZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE tbl_name [,tbl_name] ...
```

23. 杂项

```
/* 杂项 */ -----  
1. 可用反引号(`)为标识符(库名、表名、字段名、索引、别名)包裹,以避免与关键字重名! 中文也可以作为标识符!  
2. 每个库目录存在一个保存当前数据库的选项文件db.opt。  
3. 注释:  
    单行注释 # 注释内容  
    多行注释 /* 注释内容 */  
    单行注释 -- 注释内容      (标准SQL注释风格, 要求双破折号后加一空格符(空格、TAB、换行等))  
4. 模式通配符:  
    _ 任意单个字符  
    % 任意多个字符, 甚至包括零字符  
    单引号需要进行转义 \'  
5. CMD命令行内的语句结束符可以为 ";", "\G", "\g", 仅影响显示结果。其他地方还是用分号结束。  
delimiter 可修改当前对话的语句结束符。  
6. SQL对大小写不敏感  
7. 清除已有语句: \c
```

(二). Redis

1. 5种基本数据结构

1.1 Redis 简介

"Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker." —— Redis是一个开放源代码(BSD许可)的内存中数据结构存储,用作数据库,缓存和消息代理。(摘自官网)

Redis是一个开源,高级的键值存储和一个适用的解决方案,用于构建高性能,可扩展的Web应用程序。Redis也被作者戏称为 数据结构服务器,这意味着使用者可以通过一些命令,基于带有TCP套接字的简单 服务器-客户端 协议来访问一组 可变数据结构。(在Redis中都采用键值对的方式,只不过对应的数据结构不一样罢了)

1.1.1 Redis 的优点

以下是Redis的一些优点:

- **异常快** - Redis 非常快，每秒可执行大约 110000 次的设置(SET)操作，每秒大约可执行 81000 次的读取/获取(GET)操作。
- **支持丰富的数据类型** - Redis 支持开发人员常用的大多数数据类型，例如列表，集合，排序集和散列等等。这使得 Redis 很容易被用来解决各种问题，因为我们知道哪些问题可以更好使用地哪些数据类型来处理解决。
- **操作具有原子性** - 所有 Redis 操作都是原子操作，这确保如果两个客户端并发访问，Redis 服务器能接收更新的值。
- **多实用工具** - Redis 是一个多实用工具，可用于多种用例，如：缓存，消息队列(Redis 本地支持发布/订阅)，应用程序中的任何短期数据，例如，web 应用程序中的会话，网页命中计数等。

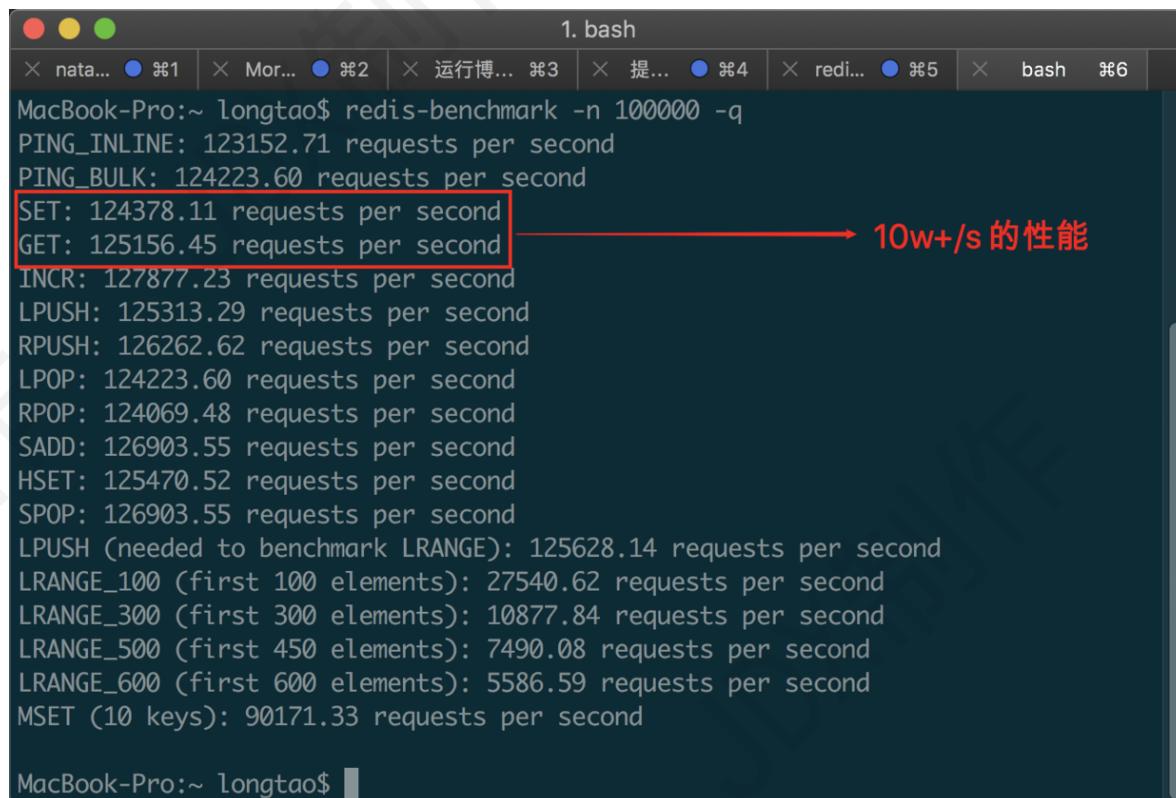
1.1.2 Redis 的安装

这一步比较简单，你可以在网上搜到许多满意的教程，这里就不再赘述。

给一个菜鸟教程的安装教程用作参考：<https://www.runoob.com/redis/redis-install.html>

1.1.3 测试本地 Redis 性能

当你安装完成之后，你可以先执行 `redis-server` 让 Redis 启动起来，然后运行命令 `redis-benchmark -n 100000 -q` 来检测本地同时执行 10 万个请求时的性能：



```

1. bash
MacBook-Pro:~ longtao$ redis-benchmark -n 100000 -q
PING_INLINE: 123152.71 requests per second
PING_BULK: 124223.60 requests per second
SET: 124378.11 requests per second
GET: 125156.45 requests per second → 10w+/s 的性能
INCR: 127877.23 requests per second
LPUSH: 125313.29 requests per second
RPUSH: 126262.62 requests per second
LPOP: 124223.60 requests per second
RPOP: 124069.48 requests per second
SADD: 126903.55 requests per second
HSET: 125470.52 requests per second
SPOP: 126903.55 requests per second
LPUSH (needed to benchmark LRANGE): 125628.14 requests per second
LRANGE_100 (first 100 elements): 27540.62 requests per second
LRANGE_300 (first 300 elements): 10877.84 requests per second
LRANGE_500 (first 450 elements): 7490.08 requests per second
LRANGE_600 (first 600 elements): 5586.59 requests per second
MSET (10 keys): 90171.33 requests per second

MacBook-Pro:~ longtao$ 

```

当然不同电脑之间由于各方面的原因会存在性能差距，这个测试您可以权当是一种「乐趣」就好。

1.2 Redis 五种基本数据结构

Redis 有 5 种基础数据结构，它们分别是：**string(字符串)**、**list(列表)**、**hash(字典)**、**set(集合)** 和 **zset(有序集合)**。这 5 种是 Redis 相关知识中最基础、最重要的部分，下面我们结合源码以及一些实践来给大家分别讲解一下。

注意：

每种数据结构都有自己底层的内部编码实现，而且是多种实现，这样 Redis 会在合适的场景选择合适的内部编码。

可以看到每种数据结构都有两种以上的内部编码实现，例如 string 数据结构就包含了 raw、int 和 embstr 三种内部编码。

同时，有些内部编码可以作为多种外部数据结构的内部实现，例如ziplist就是hash、list和zset共有的内部编码。

1.2.1 字符串 string

Redis 中的字符串是一种 **动态字符串**，这意味着使用者可以修改，它的底层实现有点类似于 Java 中的 **ArrayList**，有一个字符数组，从源码的 **sds.h/sdshdr** 文件中可以看到 Redis 底层对于字符串的定义 **SDS**，即 *Simple Dynamic String* 结构：

```
/* Note: sdshdr5 is never used, we just access the flags byte directly.
 * However is here to document the layout of type 5 SDS strings. */
struct __attribute__((__packed__)) sdshdr5 {
    unsigned char flags; /* 3 lsb of type, and 5 msb of string length */
    char buf[];
};

struct __attribute__((__packed__)) sdshdr8 {
    uint8_t len; /* used */
    uint8_t alloc; /* excluding the header and null terminator */
    unsigned char flags; /* 3 lsb of type, 5 unused bits */
    char buf[];
};

struct __attribute__((__packed__)) sdshdr16 {
    uint16_t len; /* used */
    uint16_t alloc; /* excluding the header and null terminator */
    unsigned char flags; /* 3 lsb of type, 5 unused bits */
    char buf[];
};

struct __attribute__((__packed__)) sdshdr32 {
    uint32_t len; /* used */
    uint32_t alloc; /* excluding the header and null terminator */
    unsigned char flags; /* 3 lsb of type, 5 unused bits */
    char buf[];
};

struct __attribute__((__packed__)) sdshdr64 {
    uint64_t len; /* used */
    uint64_t alloc; /* excluding the header and null terminator */
    unsigned char flags; /* 3 lsb of type, 5 unused bits */
    char buf[];
};
```

你会发现同样一组结构 Redis 使用泛型定义了好多次，**为什么不直接使用 int 类型呢？**

因为当字符串比较短的时候，len 和 alloc 可以使用 byte 和 short 来表示，**Redis 为了对内存做极致的优化，不同长度的字符串使用不同的结构体来表示。**

①、SDS 与 C 字符串的区别

为什么不考虑直接使用 C 语言的字符串呢？因为 C 语言这种简单的字符串表示方式 **不符合 Redis 对字符串在安全性、效率以及功能方面的要求**。我们知道，C 语言使用了一个长度为 N+1 的字符数组来表示长度为 N 的字符串，并且字符数组最后一个元素总是 '\0'。（下图就展示了 C 语言中值为 "Redis" 的一个字符数组）



这样简单的数据结构可能会造成以下一些问题：

- **获取字符串长度为 O(N) 级别的操作** → 因为 C 不保存数组的长度，每次都需要遍历一遍整个数组；
- 不能很好的杜绝 **缓冲区溢出/内存泄漏** 的问题 → 跟上述问题原因一样，如果执行拼接 or 缩短字符串的操作，如果操作不当就很容易造成上述问题；
- C 字符串 **只能保存文本数据** → 因为 C 语言中的字符串必须符合某种编码（比如 ASCII），例如中间出现的 `\0` 可能会被判定为提前结束的字符串而识别不了；

我们以追加字符串的操作举例，Redis 源码如下：

```
/* Append the specified binary-safe string pointed by 't' of 'len' bytes to the
 * end of the specified sds string 's'.
 *
 * After the call, the passed sds string is no longer valid and all the
 * references must be substituted with the new pointer returned by the call. */
sds sdscatlen(sds s, const void *t, size_t len) {
    // 获取原字符串的长度
    size_t curlen = sdslen(s);

    // 按需调整空间，如果容量不够容纳追加的内容，就会重新分配字节数组并复制原字符串的内容到新数组中
    s = sdsMakeRoomFor(s, len);
    if (s == NULL) return NULL; // 内存不足
    memcpy(s+curlen, t, len); // 追加目标字符串到字节数组中
    sdssetlen(s, curlen+len); // 设置追加后的长度
    s[curlen+len] = '\0'; // 让字符串以 \0 结尾，便于调试打印
    return s;
}
```

- **注：Redis 规定了字符串的长度不得超过 512 MB。**

②、对字符串的基本操作

安装好 Redis，我们可以使用 `redis-cli` 来对 Redis 进行命令行的操作，当然 Redis 官方也提供了在线的调试器，你也可以在里面敲入命令进行操作：<http://try.redis.io/#run>

③、设置和获取键值对

```
> SET key value
OK
> GET key
"value"
```

正如你看到的，我们通常使用 `SET` 和 `GET` 来设置和获取字符串值。

值可以是任何种类的字符串（包括二进制数据），例如你可以在一个键下保存一张 .jpeg 图片，只需要注意不要超过 512 MB 的最大限度就好了。

当 key 存在时，`SET` 命令会覆盖掉你上一次设置的值：

```
> SET key newValue
OK
> GET key
"newValue"
```

另外你还可以使用 `EXISTS` 和 `DEL` 关键字来查询是否存在和删除键值对：

```
> EXISTS key
(integer) 1
> DEL key
(integer) 1
> GET key
(nil)
```

④、批量设置键值对

```
> SET key1 value1
OK
> SET key2 value2
OK
> MGET key1 key2 key3      # 返回一个列表
1) "value1"
2) "value2"
3) (nil)
> MSET key1 value1 key2 value2
> MGET key1 key2
1) "value1"
2) "value2"
```

⑤、过期和 SET 命令扩展

可以对 key 设置过期时间，到时间会被自动删除，这个功能常用来控制缓存的失效时间。（过期可以是任意数据结构）

```
> SET key value1
> GET key
"value1"
> EXPIRE name 5      # 5s 后过期
...                  # 等待 5s
> GET key
(nil)
```

等价于 `SET` + `EXPIRE` 的 `SETEX` 命令：

```
> SETEX key 5 value1
...
# 等待 5s 后获取
> GET key
(nil)

> SETNX key value1 # 如果 key 不存在则 SET 成功
(integer) 1
> SETNX key value1 # 如果 key 存在则 SET 失败
(integer) 0
> GET key
"value"           # 没有改变
```

⑥、计数

如果 value 是一个整数，还可以对它使用 `INCR` 命令进行 **原子性** 的自增操作，这意味着及时多个客户端对同一个 key 进行操作，也决不会导致竞争的情况：

```
> SET counter 100
> INCR counter
(integer) 101
> INCRBY counter 50
(integer) 151
```

⑦、返回原值的 `GETSET` 命令

对字符串，还有一个 `GETSET` 比较让人觉得有意思，它的功能跟它名字一样：为 key 设置一个值并返回原值：

```
> SET key value
> GETSET key value1
"value"
```

这可以对于某一些需要隔一段时间就统计的 key 很方便的设置和查看，例如：系统每当由用户进入的时候你就是用 `INCR` 命令操作一个 key，当需要统计时候你就把这个 key 使用 `GETSET` 命令重新赋值为 0，这样就达到了统计的目的。

1.2.2 列表 list

Redis 的列表相当于 Java 语言中的 `LinkedList`，注意它是链表而不是数组。这意味着 list 的插入和删除操作非常快，时间复杂度为 $O(1)$ ，但是索引定位很慢，时间复杂度为 $O(n)$ 。

我们可以从源码的 `adlist.h/listNode` 来看到对其的定义：

```
! [7896890-8f569f06506845c1] (E:\工作007-MD\图片库\7896890-8f569f06506845c1.png) */
Node, List, and Iterator are the only data structures used currently. */

typedef struct listNode {
    struct listNode *prev;
    struct listNode *next;
    void *value;
} listNode;

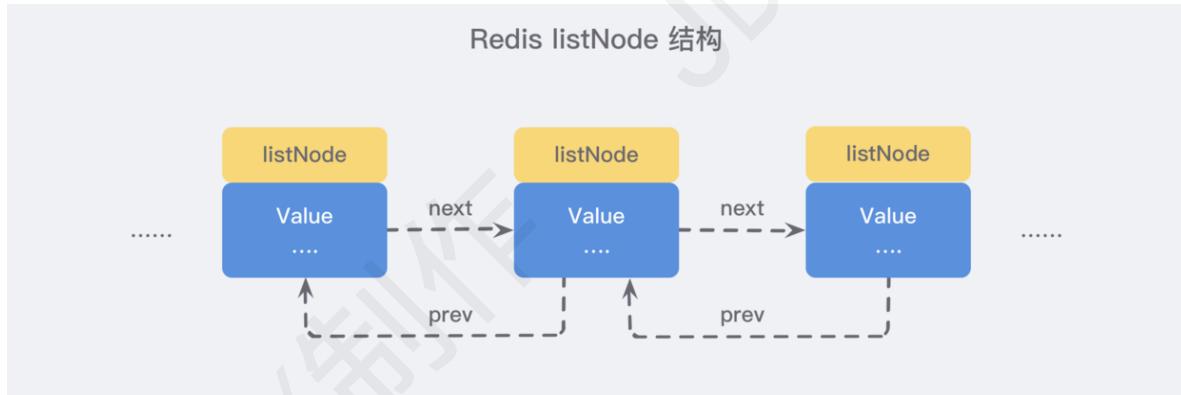
typedef struct listIter {
    listNode *next;
    int direction;
} listIter;
```

```

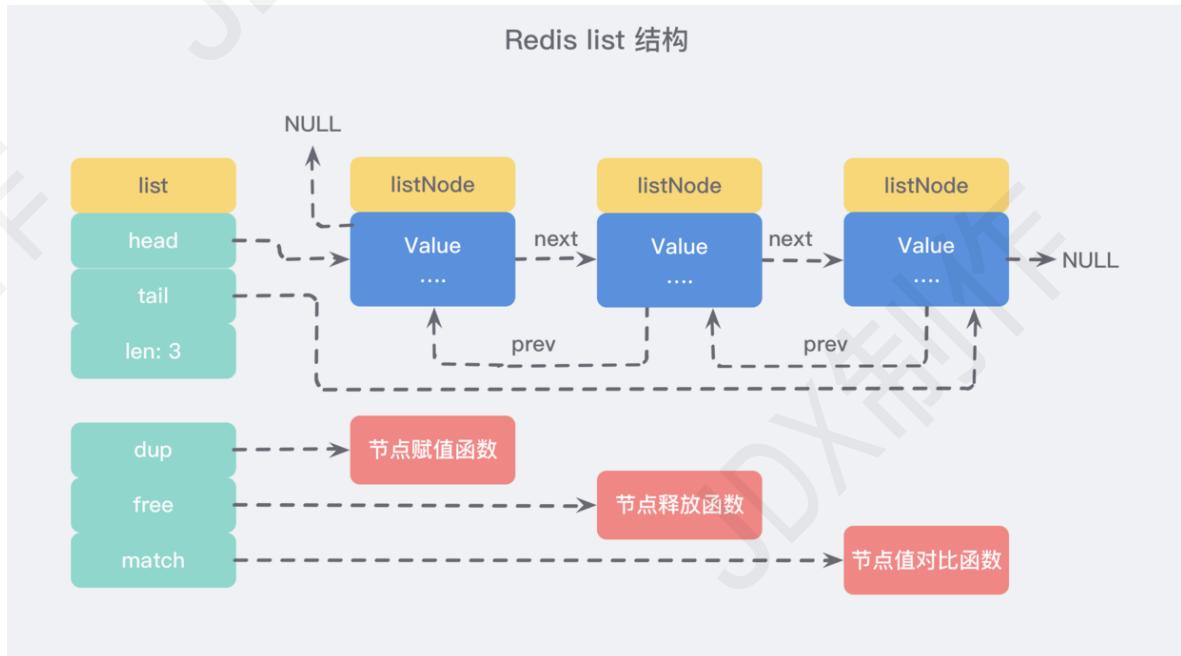
typedef struct list {
    listNode *head;
    listNode *tail;
    void *(*dup)(void *ptr);
    void (*free)(void *ptr);
    int (*match)(void *ptr, void *key);
    unsigned long len;
} list;

```

可以看到，多个 listNode 可以通过 `prev` 和 `next` 指针组成双向链表：



虽然仅仅使用多个 listNode 结构就可以组成链表，但是使用 `adlist.h/list` 结构来持有链表的话，操作起来会更加方便：



①、链表的基本操作

- `LPUSH` 和 `RPUSH` 分别可以向 list 的左边（头部）和右边（尾部）添加一个新元素；
- `LRANGE` 命令可以从 list 中取出一定范围的元素；
- `LINDEX` 命令可以从 list 中取出指定下标的元素，相当于 Java 链表操作中的 `get(int index)` 操作；

示范：

```
> rpush mylist A
(integer) 1
> rpush mylist B
(integer) 2
> lpush mylist first
(integer) 3
> lrange mylist 0 -1      # -1 表示倒数第一个元素，这里表示从第一个元素到最后一个元素，即所有
有
1) "first"
2) "A"
3) "B"
```

②、list 实现队列

队列是先进先出的数据结构，常用于消息排队和异步逻辑处理，它会确保元素的访问顺序：

```
> RPUSH books python java golang
(integer) 3
> LPOP books
"python"
> LPOP books
"java"
> LPOP books
"golang"
> LPOP books
(nil)
```

③、list 实现栈

栈是先进后出的数据结构，跟队列正好相反：

```
> RPUSH books python java golang
> RPOP books
"golang"
> RPOP books
"java"
> RPOP books
"python"
> RPOP books
(nil)
```

1.2.3 字典 hash

Redis 中的字典相当于 Java 中的 **HashMap**，内部实现也差不多类似，都是通过 “**数组 + 链表**” 的链地址法来解决部分 **哈希冲突**，同时这样的结构也吸收了两种不同数据结构的优点。源码定义如

`dict.h/dictht` 定义：

```
typedef struct dictht {
    // 哈希表数组
    dictEntry **table;
    // 哈希表大小
    unsigned long size;
    // 哈希表大小掩码，用于计算索引值，总是等于 size - 1
    unsigned long sizemask;
    // 该哈希表已有节点的数量
    unsigned long used;
```

```

} dictht;

typedef struct dict {
    dictType *type;
    void *privdata;
    // 内部有两个 dictht 结构
    dictht ht[2];
    long rehashidx; /* rehashing not in progress if rehashidx == -1 */
    unsigned long iterators; /* number of iterators currently running */
} dict;

```

`table` 属性是一个数组，数组中的每个元素都是一个指向 `dict.h/dictEntry` 结构的指针，而每个 `dictEntry` 结构保存着一个键值对：

```

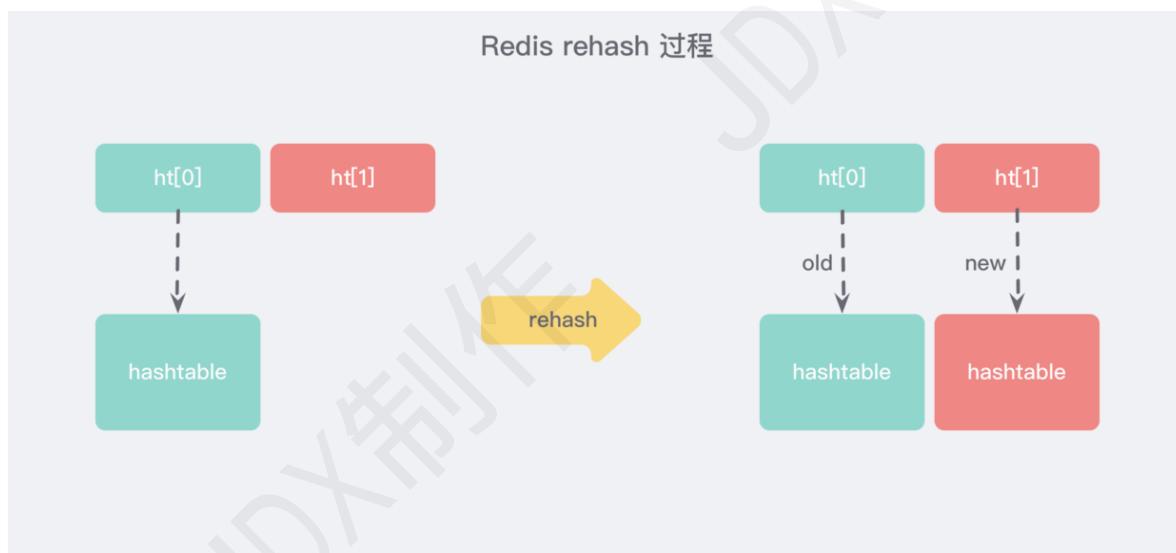
typedef struct dictEntry {
    // 键
    void *key;
    // 值
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
        double d;
    } v;
    // 指向下个哈希表节点，形成链表
    struct dictEntry *next;
} dictEntry;

```

可以从上面的源码中看到，**实际上字典结构的内部包含两个 hashtable**，通常情况下只有一个 hashtable 是有值的，但是在字典扩容缩容时，需要分配新的 hashtable，然后进行 **渐进式搬迁**（下面说原因）。

①、渐进式 `rehash`

大字典的扩容是比较耗时间的，需要重新申请新的数组，然后将旧字典所有链表中的元素重新挂接到新的数组下面，这是一个 O(n) 级别的操作，作为单线程的 Redis 很难承受这样耗时的过程，所以 Redis 使用 **渐进式 `rehash`** 小步搬迁：



渐进式 `rehash` 会在 `rehash` 的同时，保留新旧两个 hash 结构，如上图所示，查询时会同时查询两个 hash 结构，然后在后续的定时任务以及 hash 操作指令中，循序渐进的把旧字典的内容迁移到新字典中。当搬迁完成了，就会使用新的 hash 结构取而代之。

②、扩缩容的条件

正常情况下，当 hash 表中 **元素的个数等于第一维数组的长度时**，就会开始扩容，扩容的新数组是 **原数组大小的 2 倍**。不过如果 Redis 正在做 `bgsave`(持久化命令)，为了减少内存也得过多分离，Redis 尽量不去扩容，但是如果 hash 表非常满了，**达到了第一维数组长度的 5 倍了**，这个时候就会 **强制扩容**。

当 hash 表因为元素逐渐被删除变得越来越稀疏时，Redis 会对 hash 表进行缩容来减少 hash 表的第一维数组空间占用。所用的条件是 **元素个数低于数组长度的 10%**，缩容不会考虑 Redis 是否在做 `bgsave`。

③、字典的基本操作

hash 也有缺点，hash 结构的存储消耗要高于单个字符串，所以到底该使用 hash 还是字符串，需要根据实际情况再三权衡：

```
> HSET books java "think in java"      # 命令行的字符串如果包含空格则需要使用引号包裹
(integer) 1
> HSET books python "python cookbook"
(integer) 1
> HGETALL books      # key 和 value 间隔出现
1) "java"
2) "think in java"
3) "python"
4) "python cookbook"
> HGET books java
"think in java"
> HSET books java "head first java"
(integer) 0          # 因为是更新操作，所以返回 0
> HMSET books java "effetive java" python "learning python"      # 批量操作
OK
```

1.2.4 集合 set

Redis 的集合相当于 Java 语言中的 **HashSet**，它内部的键值对是无序、唯一的。它的内部实现相当于一个特殊的字典，字典中所有的 value 都是一个值 NULL。

①、集合 set 的基本使用

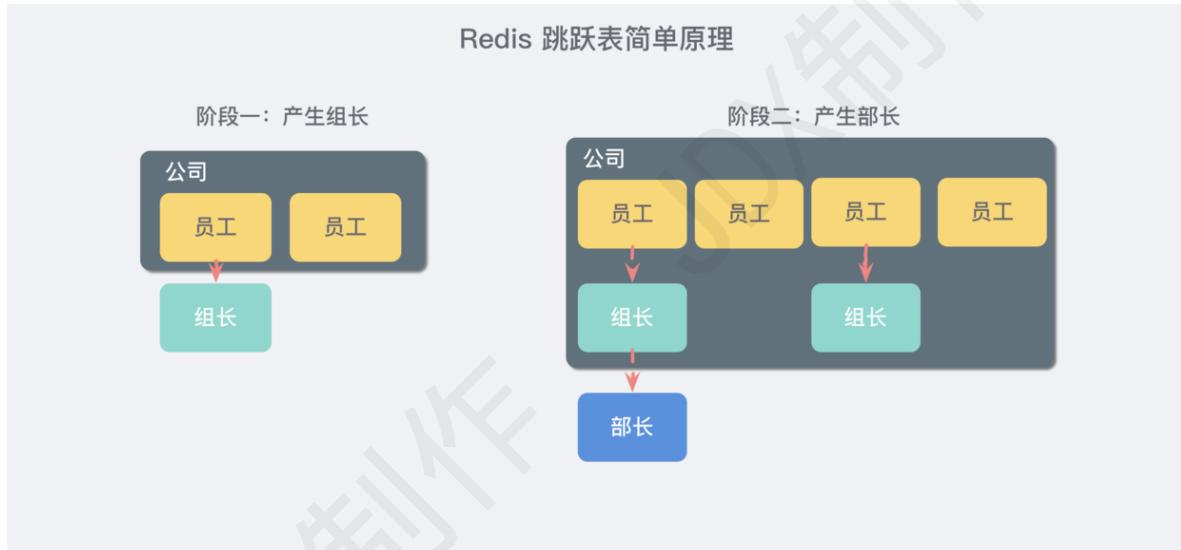
由于该结构比较简单，我们直接来看看是如何使用的：

```
> SADD books java
(integer) 1
> SADD books java      # 重复
(integer) 0
> SADD books python golang
(integer) 2
> SMEMBERS books      # 注意顺序，set 是无序的
1) "java"
2) "python"
3) "golang"
> SISMEMBER books java    # 查询某个 value 是否存在，相当于 contains
(integer) 1
> SCARD books      # 获取长度
(integer) 3
> SPOP books      # 弹出一个
"java"
```

1.2.5 有序列表 zset

这可能使 Redis 最具特色的一个数据结构了，它类似于 Java 中 **SortedSet** 和 **HashMap** 的结合体，一方面它是一个 set，保证了内部 value 的唯一性，另一方面它可以为每个 value 赋予一个 score 值，用来代表排序的权重。

它的内部实现用的是一种叫做「跳跃表」的数据结构，由于比较复杂，所以在这里简单提一下原理就好了：



想象你是一家创业公司的老板，刚开始只有几个人，大家都平起平坐。后来随着公司的发展，人数越来越多，团队沟通成本逐渐增加，渐渐地引入了组长制，对团队进行划分，于是有一些人**又是员工又有组长的身份**。

再后来，公司规模进一步扩大，公司需要再进入一个层级：部门。于是每个部门又会从组长中推举一位选出部长。

跳跃表就类似于这样的机制，最下面一层所有的元素都会串起来，都是员工，然后每隔几个元素就会挑选出一个代表，再把这几个代表使用另外一级指针串起来。然后再在这些代表里面挑出二级代表，再串起来。**最终形成了一个金字塔的结构**。

想一下你目前所在的地理位置：亚洲 > 中国 > 某省 > 某市 >，就是这样一个结构！

①、有序列表 zset 基础操作

```
> ZADD books 9.0 "think in java"
> ZADD books 8.9 "java concurrency"
> ZADD books 8.6 "java cookbook"

>ZRANGE books 0 -1      # 按 score 排序列出，参数区间为排名范围
1) "java cookbook"
2) "java concurrency"
3) "think in java"

> ZREVRANGE books 0 -1 # 按 score 逆序列出，参数区间为排名范围
1) "think in java"
2) "java concurrency"
3) "java cookbook"

> ZCARD books          # 相当于 count()
(integer) 3

> ZSCORE books "java concurrency"    # 获取指定 value 的 score
"8.9000000000000004"                 # 内部 score 使用 double 类型进行存储，所以存在小数点精度问题
```

```

> ZRANK books "java concurrency"      # 排名
(integer) 1

> ZRANGEBYSCORE books 0 8.91          # 根据分值区间遍历 zset
1) "java cookbook"
2) "java concurrency"

> ZRANGEBYSCORE books -inf 8.91 withscores # 根据分值区间 (-∞, 8.91] 遍历 zset, 同时返回分值。inf 代表 infinite, 无穷大的意思。
1) "java cookbook"
2) "8.599999999999996"
3) "java concurrency"
4) "8.900000000000004"

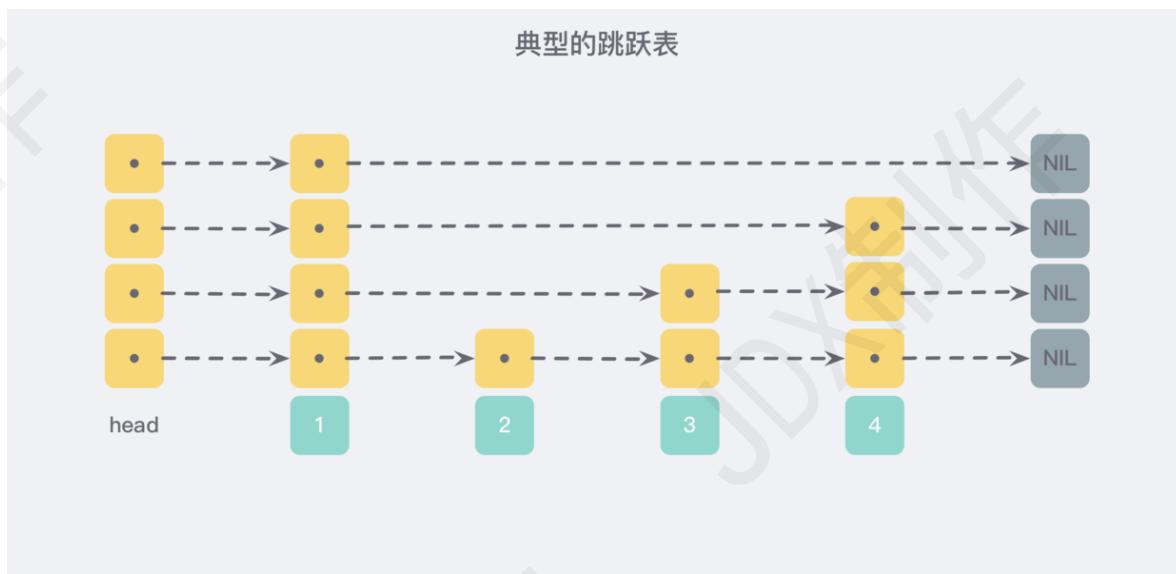
> ZREM books "java concurrency"        # 删除 value
(integer) 1
> ZRANGE books 0 -1
1) "java cookbook"
2) "think in java"

```

2. 跳跃表

2.1 跳跃表简介

跳跃表（skiplist）是一种随机化的数据结构，由 **William Pugh** 在论文《Skip lists: a probabilistic alternative to balanced trees》中提出，是一种可以于平衡树媲美的层次化链表结构——查找、删除、添加等操作都可以在对数期望时间下完成，以下是一个典型的跳跃表例子：



我们在上一篇中提到了 Redis 的五种基本结构中，有一个叫做 **有序列表 zset** 的数据结构，它类似于 Java 中的 **SortedSet** 和 **HashMap** 的结合体，一方面它是一个 set 保证了内部 value 的唯一性，另一方面又可以给每个 value 赋予一个排序的权重值 score，来达到 **排序** 的目的。

它的内部实现就依赖了一种叫做「**跳跃列表**」的数据结构。

2.1.1 为什么使用跳跃表

首先，因为 zset 要支持随机的插入和删除，所以它 **不宜使用数组来实现**，关于排序问题，我们也很容易就想到 **红黑树/ 平衡树** 这样的树形结构，为什么 Redis 不使用这样一些结构呢？

- 性能考虑：**在高并发的情况下，树形结构需要执行一些类似于 rebalance 这样的可能涉及整棵树的操作，相对来说跳跃表的变化只涉及局部（下面详细说）；
- 实现考虑：**在复杂度与红黑树相同的情况下，跳跃表实现起来更简单，看起来也更加直观；

基于以上的一些考虑，Redis 基于 William Pugh 的论文做出一些改进后采用了 跳跃表 这样的结构。

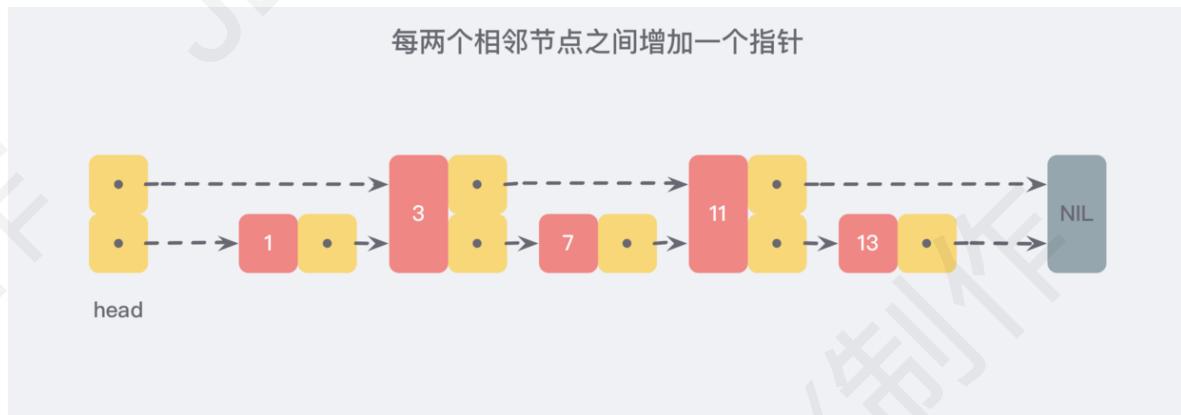
2.1.2 本质是解决查找问题

我们先来看一个普通的链表结构：



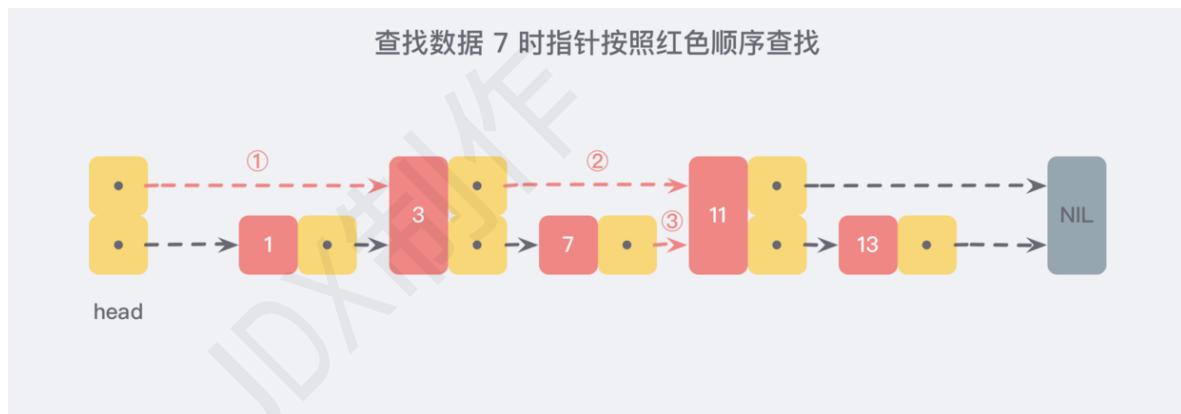
我们需要这个链表按照 score 值进行排序，这也就意味着，当我们需要添加新的元素时，我们需要定位到插入点，这样才可以继续保证链表是有序的，通常我们会使用 **二分查找法**，但二分查找是有序数组的，链表没办法进行位置定位，我们除了遍历整个找到第一个比给定数据大的节点为止（时间复杂度 $O(n)$ ）似乎没有更好的办法。

但假如我们每相邻两个节点之间就增加一个指针，让指针指向下一个节点，如下图：



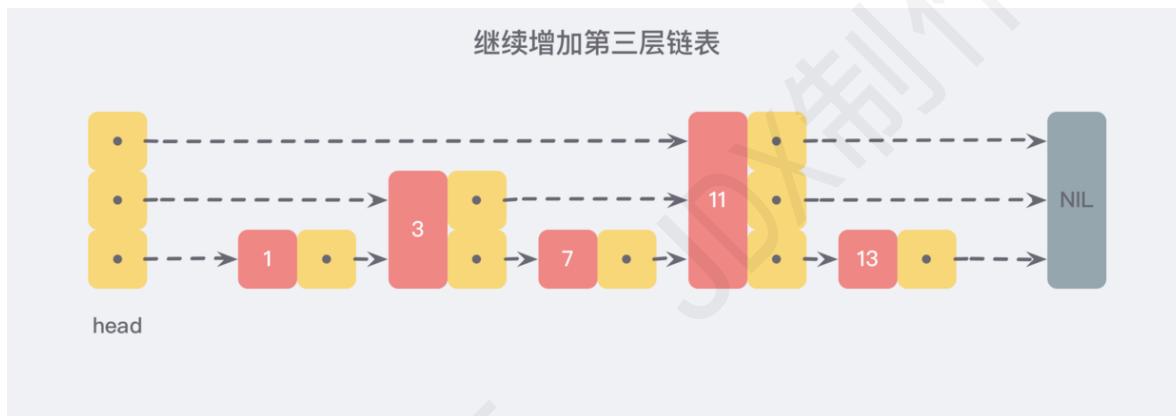
这样所有新增的指针连成了一个新的链表，但它包含的数据却只有原来的一半（图中的为 3, 11）。

现在假设我们想要查找数据时，可以根据这条新的链表查找，如果碰到比待查找数据大的节点时，再回到原来的链表中进行查找，比如，我们想要查找 7，查找的路径则是沿着下图中标注出的红色指针所指向的方向进行的：



这是一个略微极端的例子，但我们仍然可以看到，通过新增加的指针查找，我们不再需要与链表上的每一个节点逐一进行比较，这样改进之后需要比较的节点数大概只有原来的一半。

利用同样的方式，我们可以在新产生的链表上，继续为每两个相邻的节点增加一个指针，从而产生第三层链表：



在这个新的三层链表结构中，我们试着 **查找 13**，那么沿着最上层链表首先比较的是 11，发现 11 比 13 小，于是我们就知道只需要到 11 后面继续查找，从而一下子跳过了 11 前面的所有节点。

可以想象，当链表足够长，这样的多层链表结构可以帮助我们跳过很多下层节点，从而加快查找的效率。

2.1.3 更进一步的跳跃表

跳跃表 skiplist 就是受到这种多层次链表结构的启发而设计出来的。按照上面生成链表的方式，上面每一层链表的节点个数，是下面一层的节点个数的一半，这样查找过程就非常类似于一个二分查找，使得查找的时间复杂度可以降低到 $O(\log n)$ 。

但是，这种方法在插入数据的时候有很大的问题。新插入一个节点之后，就会打乱上下相邻两层链表上节点个数严格的 2:1 的对应关系。如果要维持这种对应关系，就必须把新插入的节点后面的所有节点（也包括新插入的节点）重新进行调整，这会让时间复杂度重新蜕化成 $O(n)$ 。删除数据也有同样的问题。

skiplist 为了避免这一问题，它不要求上下相邻两层链表之间的节点个数有严格的对应关系，而是 **为每个节点随机出一个层数(level)**。比如，一个节点随机出的层数是 3，那么就把它链入到第 1 层到第 3 层这三层链表中。为了表达清楚，下图展示了如何通过一步步的插入操作从而形成一个 skiplist 的过程：

小插曲：

更多阿里、腾讯、美团、京东等一线互联网大厂 Java 面试真题；包含：基础、并发、锁、JVM、设计模式、数据结构、反射/IO、数据库、Redis、Spring、消息队列、分布式、Zookeeper、Dubbo、Mybatis、Maven、面试等。
更多 Java 程序员技术进阶小技巧；例如高效学习（如何学习和阅读代码、面对枯燥和量大的知识）高效沟通（沟通方式及技巧、沟通技术）

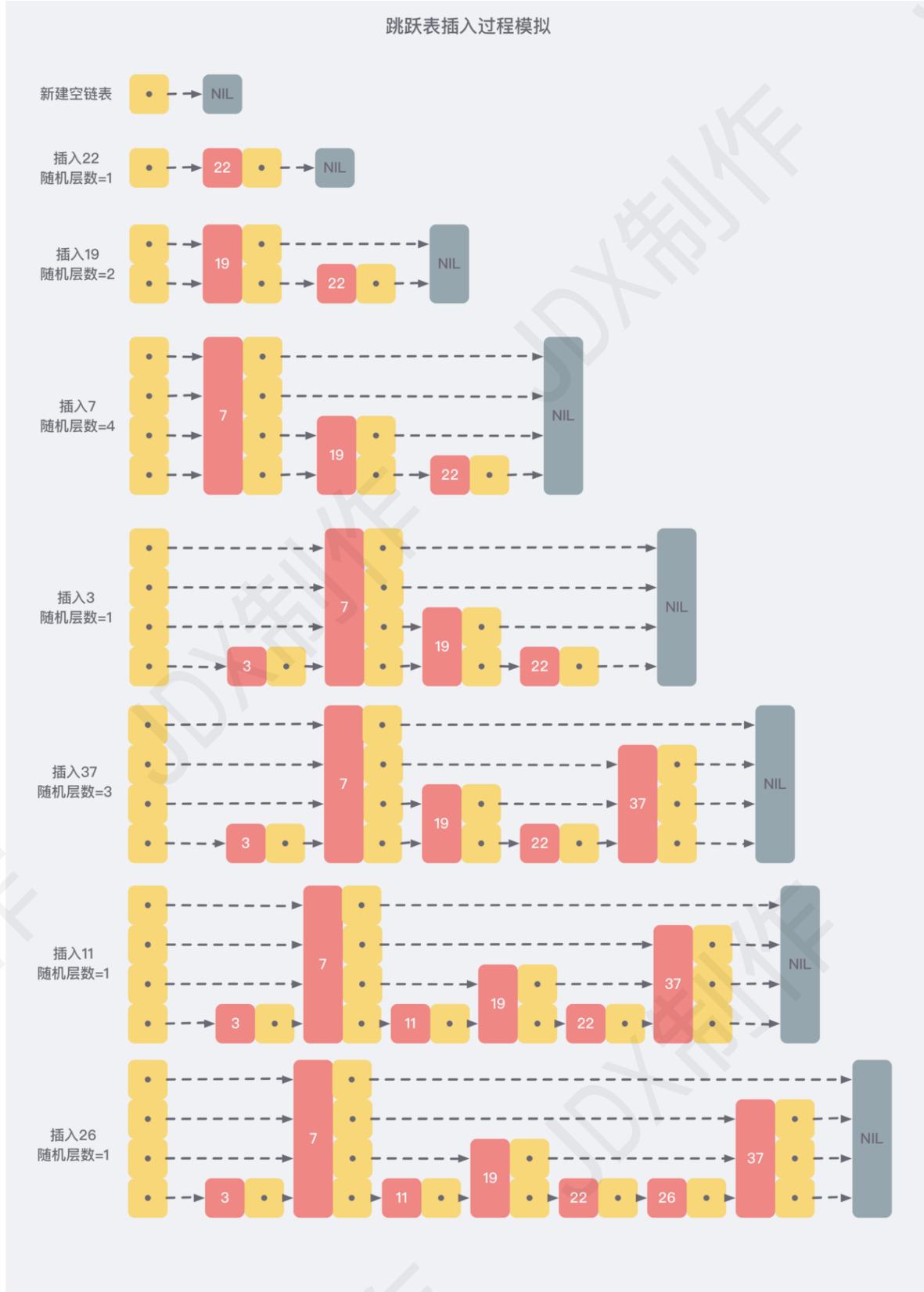
更多 Java 大牛分享的一些职业生涯分享文档

请添加微信：jiang10086a 免费获取

比你优秀的对手在学习，你的仇人在磨刀，你的闺蜜在减肥，隔壁老王在练腰，我们必须不断学习，否则我们将被学习者超越！

趁年轻，使劲拼，给未来的自己一个交代！

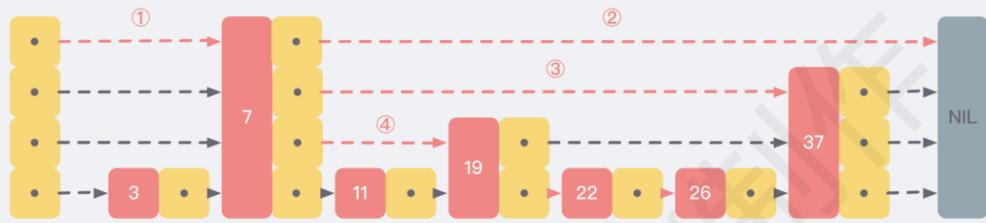
跳跃表插入过程模拟



从上面的创建和插入的过程中可以看出，每一个节点的层数（level）是随机出来的，而且新插入一个节点并不会影响到其他节点的层数，因此，插入操作只需要修改节点前后的指针，而不需要对多个节点都进行调整，这就降低了插入操作的复杂度。

现在我们假设从我们刚才创建的这个结构中查找 23 这个不存在的数，那么查找路径会如下图：

从刚才创建的结构中查找 23



2.2 跳跃表的实现

Redis 中的跳跃表由 `server.h/zskiplistNode` 和 `server.h/zskiplist` 两个结构定义，前者为跳跃表节点，后者则保存了跳跃节点的相关信息，同之前的 `list` 结构类似，其实只有 `zskiplistNode` 就可以实现了，但是引入后者是为了更加方便的操作：

```
/* ZSETS use a specialized version of skiplists */
typedef struct zskiplistNode {
    // value
    sds ele;
    // 分值
    double score;
    // 后退指针
    struct zskiplistNode *backward;
    // 层
    struct zskiplistLevel {
        // 前进指针
        struct zskiplistNode *forward;
        // 跨度
        unsigned long span;
    } level[];
} zskiplistNode;

typedef struct zskiplist {
    // 跳跃表头指针
    struct zskiplistNode *header, *tail;
    // 表中节点的数量
    unsigned long length;
    // 表中层数最大的节点的层数
    int level;
} zskiplist;
```

正如文章开头画出来的那张标准的跳跃表那样。

2.2.1 随机层数

对于每一个新插入的节点，都需要调用一个随机算法给它分配一个合理的层数，源码在 `t_zset.c/zslRandomLevel(void)` 中被定义：

```

int zslRandomLevel(void) {
    int level = 1;
    while ((random() & 0xFFFF) < (ZSKIPLIST_P * 0xFFFF))
        level += 1;
    return (level < ZSKIPLIST_MAXLEVEL) ? level : ZSKIPLIST_MAXLEVEL;
}

```

直观上期望的目标是 50% 的概率被分配到 Level 1，25% 的概率被分配到 Level 2，12.5% 的概率被分配到 Level 3，以此类推... 有 2^{-63} 的概率被分配到最顶层，因为这里每一层的晋升率都是 50%。

Redis 跳跃表默认允许最大的层数是 32，被源码中 `ZSKIPLIST_MAXLEVEL` 定义，当 `Level[0]` 有 2^{64} 个元素时，才能达到 32 层，所以定义 32 完全够用了。

2.2.2 创建跳跃表

这个过程比较简单，在源码中的 `t_zset.c/zslCreate` 中被定义：

```

zskiplist *zslCreate(void) {
    int j;
    zskiplist *zsl;

    // 申请内存空间
    zsl = zmalloc(sizeof(*zsl));
    // 初始化层数为 1
    zsl->level = 1;
    // 初始化长度为 0
    zsl->length = 0;
    // 创建一个层数为 32，分数为 0，没有 value 值的跳跃表头节点
    zsl->header = zslCreateNode(ZSKIPLIST_MAXLEVEL, 0, NULL);

    // 跳跃表头节点初始化
    for (j = 0; j < ZSKIPLIST_MAXLEVEL; j++) {
        // 将跳跃表头节点的所有前进指针 forward 设置为 NULL
        zsl->header->level[j].forward = NULL;
        // 将跳跃表头节点的所有跨度 span 设置为 0
        zsl->header->level[j].span = 0;
    }
    // 跳跃表头节点的后退指针 backward 置为 NULL
    zsl->header->backward = NULL;
    // 表头指向跳跃表尾节点的指针置为 NULL
    zsl->tail = NULL;
    return zsl;
}

```

即执行完之后创建了如下结构的初始化跳跃表：

小插曲：

更多阿里、腾讯、美团、京东等一线互联网大厂Java面试真题；包含：基础、并发、锁、JVM、设计模式、数据结构、反射/IO、数据库、Redis、Spring、消息队列、分布式、Zookeeper、Dubbo、Mybatis、Maven、面试等。

更多Java程序员技术进阶小技巧；例如高效学习（如何学习和阅读代码、面对枯燥和量大的知识）高效沟通（沟通方式及技巧、沟通技术）

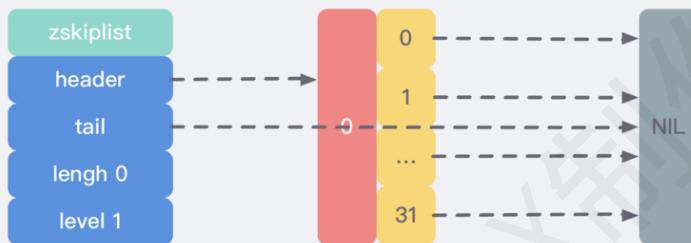
更多Java大牛分享的一些职业生涯分享文档

请添加微信：jiang10086a 免费获取

比你优秀的对手在学习，你的仇人在磨刀，你的闺蜜在减肥，隔壁老王在练腰，我们必须不断学习，否则我们将被学习者超越！

趁年轻，使劲拼，给未来的自己一个交代！

初始化跳跃表结构



2.2.3 插入节点实现

这几乎是最重要的一段代码了，但总体思路也比较清晰简单，如果理解了上面所说的跳跃表的原理，那么很容易理清楚插入节点时发生的几个动作（几乎跟链表类似）：

1. 找到当前我需要插入的位置（其中包括相同 score 时的处理）；
2. 创建新节点，调整前后的指针指向，完成插入；

为了方便阅读，我把源码 `t_zset.c/zsInsert` 定义的插入函数拆成了几个部分

第一部分：声明需要存储的变量

```
// 存储搜索路径  
zskiplistNode *update[ZSKIPLIST_MAXLEVEL], *x;  
// 存储经过的节点跨度  
unsigned int rank[ZSKIPLIST_MAXLEVEL];  
int i, level;
```

第二部分：搜索当前节点插入位置

```
serverAssert(!isnan(score));  
x = zsl->header;  
// 逐步降级寻找目标节点，得到 "搜索路径"  
for (i = zsl->level-1; i >= 0; i--) {  
    /* store rank that is crossed to reach the insert position */  
    rank[i] = i == (zsl->level-1) ? 0 : rank[i+1];  
    // 如果 score 相等，还需要比较 value 值  
    while (x->level[i].forward &&  
        (x->level[i].forward->score < score ||  
         (x->level[i].forward->score == score &&  
          sdscmp(x->level[i].forward->ele, ele) < 0)))  
    {  
        rank[i] += x->level[i].span;  
        x = x->level[i].forward;  
    }  
    // 记录 "搜索路径"  
    update[i] = x;  
}
```

讨论：有一种极端的情况，就是跳跃表中的所有 score 值都是一样，zset 的查找性能会不会退化为 $O(n)$ 呢？

从上面的源码中我们可以发现 zset 的排序元素不只是看 score 值，也会比较 value 值（字符串比较）

第三部分：生成插入节点

```
/* we assume the element is not already inside, since we allow duplicated
 * scores, reinserting the same element should never happen since the
 * caller of zslInsert() should test in the hash table if the element is
 * already inside or not. */
level = zslRandomLevel();
// 如果随机生成的 level 超过了当前最大 level 需要更新跳跃表的信息
if (level > zsl->level) {
    for (i = zsl->level; i < level; i++) {
        rank[i] = 0;
        update[i] = zsl->header;
        update[i]->level[i].span = zsl->length;
    }
    zsl->level = level;
}
// 创建新节点
x = zslCreateNode(level, score, ele);
```

第四部分：重排前向指针

```
for (i = 0; i < level; i++) {
    x->level[i].forward = update[i]->level[i].forward;
    update[i]->level[i].forward = x;

    /* update span covered by update[i] as x is inserted here */
    x->level[i].span = update[i]->level[i].span - (rank[0] - rank[i]);
    update[i]->level[i].span = (rank[0] - rank[i]) + 1;
}

/* increment span for untouched levels */
for (i = level; i < zsl->level; i++) {
    update[i]->level[i].span++;
}
```

第五部分：重排后向指针并返回

```
x->backward = (update[0] == zsl->header) ? NULL : update[0];
if (x->level[0].forward)
    x->level[0].forward->backward = x;
else
    zsl->tail = x;
zsl->length++;
return x;
```

2.2.4 节点删除实现

删除过程由源码中的 `t_zset.c/zslDeleteNode` 定义，和插入过程类似，都需要先把这个 "搜索路径" 找出来，然后对于每个层的相关节点重排一下前向后向指针，同时还要注意更新一下最高层数 `maxLevel`，直接放源码（如果理解了插入这里还是很容易理解的）：

```
/* Internal function used by zslDelete, zslDeleteByScore and zslDeleteByRank */
void zslDeleteNode(zskiplist *zsl, zskiplistNode *x, zskiplistNode **update) {
    int i;
    for (i = 0; i < zsl->level; i++) {
```

```

        if (update[i]->level[i].forward == x) {
            update[i]->level[i].span += x->level[i].span - 1;
            update[i]->level[i].forward = x->level[i].forward;
        } else {
            update[i]->level[i].span -= 1;
        }
    }

    if (x->level[0].forward) {
        x->level[0].forward->backward = x->backward;
    } else {
        zsl->tail = x->backward;
    }

    while(zsl->level > 1 && zsl->header->level[zsl->level-1].forward == NULL)
        zsl->level--;
    zsl->length--;
}

/* Delete an element with matching score/element from the skip list.
 * The function returns 1 if the node was found and deleted, otherwise
 * 0 is returned.
 *
 * If 'node' is NULL the deleted node is freed by zslFreeNode(), otherwise
 * it is not freed (but just unlinked) and *node is set to the node pointer,
 * so that it is possible for the caller to reuse the node (including the
 * referenced SDS string at node->ele). */
int zslDelete(zskiplist *zsl, double score, sds ele, zskiplistNode **node) {
    zskiplistNode *update[ZSKIPLIST_MAXLEVEL], *x;
    int i;

    x = zsl->header;
    for (i = zsl->level-1; i >= 0; i--) {
        while (x->level[i].forward &&
               (x->level[i].forward->score < score ||
                (x->level[i].forward->score == score &&
                 sdscmp(x->level[i].forward->ele,ele) < 0)))
        {
            x = x->level[i].forward;
        }
        update[i] = x;
    }

    /* We may have multiple elements with the same score, what we need
     * is to find the element with both the right score and object. */
    x = x->level[0].forward;
    if (x && score == x->score && sdscmp(x->ele,ele) == 0) {
        zslDeleteNode(zsl, x, update);
        if (!node)
            zslFreeNode(x);
        else
            *node = x;
        return 1;
    }
    return 0; /* not found */
}

```

2.2.5 节点更新实现

当我们调用 `ZADD` 方法时，如果对应的 value 不存在，那就是插入过程，如果这个 value 已经存在，只是调整一下 score 的值，那就需要走一个更新流程。

假设这个新的 score 值并不会带来排序上的变化，那么就不需要调整位置，直接修改元素的 score 值就可以了，但是如果排序位置改变了，那就需要调整位置，该如何调整呢？

从源码 `t_zset.c/zsetAdd` 函数 1350 行左右可以看到，Redis 采用了一个非常简单的策略：

```
/* Remove and re-insert when score changed. */
if (score != curscore) {
    zobj->ptr = zlDelete(zobj->ptr,eptr);
    zobj->ptr = zlInsert(zobj->ptr,ele,score);
    *flags |= ZADD_UPDATED;
}
```

把这个元素删除再插入这个，需要经过两次路径搜索，从这一点上来看，Redis 的 `ZADD` 代码似乎还有进一步优化的空间。

2.2.6 元素排名的实现

跳跃表本身是有序的，Redis 在 skipList 的 forward 指针上进行了优化，给每一个 forward 指针都增加了 `span` 属性，用来表示从前一个节点沿着当前层的 `forward` 指针跳到当前这个节点中间会跳过多少个节点。在上面的源码中我们也可以看到 Redis 在插入、删除操作时都会小心翼翼地更新 `span` 值的大小。

所以，沿着“**搜索路径**”，把所有经过节点的跨度 `span` 值进行累加就可以算出当前元素的最终 rank 值了：

```
/* Find the rank for an element by both score and key.
 * Returns 0 when the element cannot be found, rank otherwise.
 * Note that the rank is 1-based due to the span of zsl->header to the
 * first element. */
unsigned long zslGetRank(zskipList *zsl, double score, sds ele) {
    zskipListNode *x;
    unsigned long rank = 0;
    int i;

    x = zsl->header;
    for (i = zsl->level-1; i >= 0; i--) {
        while (x->level[i].forward &&
               (x->level[i].forward->score < score ||
                (x->level[i].forward->score == score &&
                 sdscmp(x->level[i].forward->ele,ele) <= 0))) {
            // span 累加
            rank += x->level[i].span;
            x = x->level[i].forward;
        }

        /* x might be equal to zsl->header, so test if obj is non-NULL */
        if (x->ele && sdscmp(x->ele,ele) == 0) {
            return rank;
        }
    }
    return 0;
}
```

3. 分布式锁深入探究

3.1 分布式锁简介

锁 是一种用来解决多个执行线程 访问共享资源 错误或数据不一致问题的工具。

如果 把一台服务器比作一个房子，那么 线程就好比里面的住户，当他们想要共同访问一个共享资源，例如厕所的时候，如果厕所门上没有锁...更甚者厕所没装门...这是会出原则性的问题的..

装上了锁，大家用起来就安心多了，本质也就是 同一时间只允许一个住户使用。

而随着互联网世界的发展，单体应用已经越来越无法满足复杂互联网的高并发需求，转而慢慢朝着分布式方向发展，慢慢进化成了 更大一些的住户。所以同样，我们需要引入分布式锁来解决分布式应用之间访问共享资源的并发问题。

3.1.1 为何需要分布式锁

一般情况下，我们使用分布式锁主要有两个场景：

1. 避免不同节点重复相同的工作：比如用户执行了某个操作有可能不同节点会发送多封邮件；
2. 避免破坏数据的正确性：如果两个节点在同一条数据上同时进行操作，可能会造成数据错误或不一致的情况出现；

3.1.2 Java 中实现的常见方式

上面我们用简单的比喻说明了锁的本质：同一时间只允许一个用户操作。所以理论上，能够满足这个需求的工具我们都能够使用（就是其他应用能帮我们加锁的）：

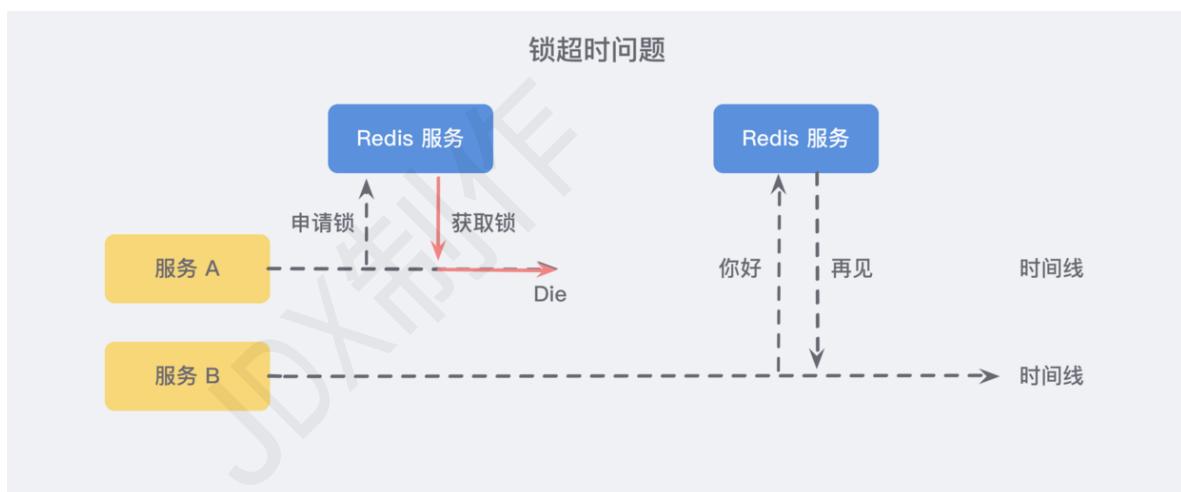
1. 基于 MySQL 中的锁：MySQL 本身有自带的悲观锁 `for update` 关键字，也可以自己实现悲观/乐观锁来达到目的；
2. 基于 Zookeeper 有序节点：Zookeeper 允许临时创建有序的子节点，这样客户端获取节点列表时，就能够当前子节点列表中的序号判断是否能够获得锁；
3. 基于 Redis 的单线程：由于 Redis 是单线程，所以命令会以串行的方式执行，并且本身提供了像 `SETNX(set if not exists)` 这样的指令，本身具有互斥性；

每个方案都有各自的优缺点，例如 MySQL 虽然直观理解容易，但是实现起来却需要额外考虑 锁超时、加事务 等，并且性能局限于数据库，诸如此类我们在此不作讨论，重点关注 Redis。

3.1.3 Redis 分布式锁的问题

①、锁超时

假设现在我们有两台平行的服务 A B，其中 A 服务在 获取锁之后 由于未知神秘力量突然 挂了，那么 B 服务就永远无法获取到锁了：



所以我们需要额外设置一个超时时间，来保证服务的可用性。

但是另一个问题随即而来：如果在加锁和释放锁之间的逻辑执行得太长，以至于超出了锁的超时限制，也会出现问题。因为这时候第一个线程持有锁过期了，而临界区的逻辑还没有执行完，与此同时第二个线程就提前拥有了这把锁，导致临界区的代码不能得到严格的串行执行。

为了避免这个问题，Redis 分布式锁不要用于较长时间的任务。如果真的偶尔出现了问题，造成的数据小错乱可能就需要人工的干预。

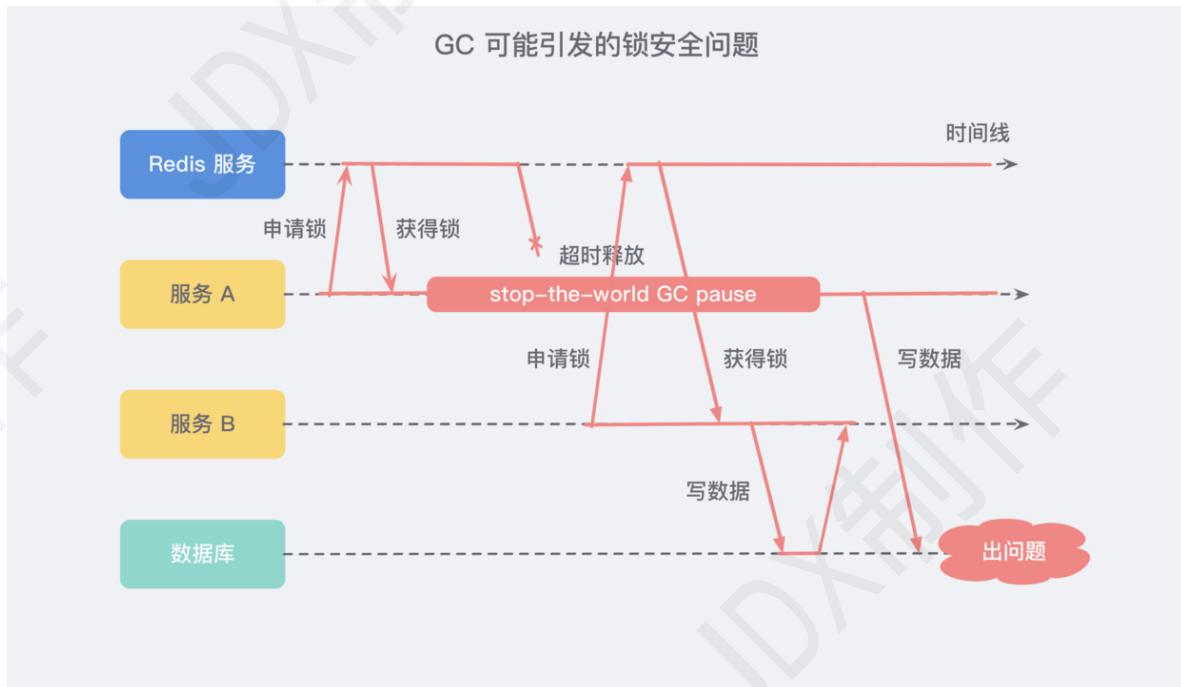
有一个稍微安全一点的方案是 将锁的 value 值设置为一个随机数，释放锁时先匹配随机数是否一致，然后再删除 key，这是为了 确保当前线程占有的锁不会被其他线程释放，除非这个锁是因为过期了而被服务器自动释放的。

但是匹配 value 和删除 key 在 Redis 中并不是一个原子性的操作，也没有类似保证原子性的指令，所以可能需要使用像 Lua 这样的脚本来处理了，因为 Lua 脚本可以 保证多个指令的原子性执行。

延伸的讨论：GC 可能引发的安全问题

Martin Kleppmann 曾与 Redis 之父 Antirez 就 Redis 实现分布式锁的安全性问题进行过深入的讨论，其中有一个问题就涉及到 GC。

熟悉 Java 的同学肯定对 GC 不陌生，在 GC 的时候会发生 STW(Stop-The-World)，这本身是为了保障垃圾回收器的正常执行，但可能会引发如下的问题：



服务 A 获取了锁并设置了超时时间，但是服务 A 出现了 STW 且时间较长，导致了分布式锁进行了超时释放，在这个期间服务 B 获取到了锁，待服务 A STW 结束之后又恢复了锁，这就导致了 服务 A 和服务 B 同时获取到了锁，这个时候分布式锁就不安全了。

不仅仅局限于 Redis，Zookeeper 和 MySQL 有同样的问题。

②、单点/多点问题

如果 Redis 采用单机部署模式，那就意味着当 Redis 故障了，就会导致整个服务不可用。

而如果采用主从模式部署，我们想象一个这样的场景：服务 A 申请到一把锁之后，如果作为主机的 Redis 宕机了，那么服务 B 在申请锁的时候就会从从机那里获取到这把锁，为了解决这个问题，Redis 作者提出了一种 RedLock 红锁的算法 (Redisson / Jedis)：

```

// 三个 Redis 集群
RLock lock1 = redissionInstance1.getLock("lock1");
RLock lock2 = redissionInstance2.getLock("lock2");
RLock lock3 = redissionInstance3.getLock("lock3");

RedissionRedLock lock = new RedissionLock(lock1, lock2, lock3);
lock.lock();
// do something....
lock.unlock();

```

3.2 Redis 分布式锁的实现

分布式锁类似于“占坑”，而 `SETNX(SET if Not exists)` 指令就是这样的一个操作，只允许被一个客户端占有，我们来看看 源码(`t_string.c/setGenericCommand`) 吧：

```

// SET/ SETEX/ SETTEX/ SETNX 最底层实现
void setGenericCommand(client *c, int flags, robj *key, robj *val, robj *expire,
int unit, robj *ok_reply, robj *abort_reply) {
    long long milliseconds = 0; /* initialized to avoid any harness warning */

    // 如果定义了 key 的过期时间则保存到上面定义的变量中
    // 如果过期时间设置错误则返回错误信息
    if (expire) {
        if (getLongLongFromObjectOrReply(c, expire, &milliseconds, NULL) !=
C_OK)
            return;
        if (milliseconds <= 0) {
            addReplyErrorFormat(c, "invalid expire time in %s", c->cmd->name);
            return;
        }
        if (unit == UNIT_SECONDS) milliseconds *= 1000;
    }

    // lookupKeywrite 函数是为执行写操作而取出 key 的值对象
    // 这里的判断条件是：
    // 1.如果设置了 NX(不存在)，并且在数据库中找到了 key 值
    // 2.或者设置了 XX(存在)，并且在数据库中没有找到该 key
    // => 那么回复 abort_reply 给客户端
    if ((flags & OBJ_SET_NX && lookupKeywrite(c->db, key) != NULL) ||
(flags & OBJ_SET_XX && lookupKeywrite(c->db, key) == NULL))
    {
        addReply(c, abort_reply ? abort_reply : shared.null[c->resp]);
        return;
    }

    // 在当前的数据库中设置键为 key 值为 value 的数据
    genericSetKey(c->db, key, val, flags & OBJ_SET_KEEPTTL);
    // 服务器每修改一个 key 后都会修改 dirty 值
    server.dirty++;
    if (expire) setExpire(c, c->db, key, mstime() + milliseconds);
    notifyKeyspaceEvent(NOTIFY_STRING, "set", key, c->db->id);
    if (expire) notifyKeyspaceEvent(NOTIFY_GENERIC,
        "expire", key, c->db->id);
    addReply(c, ok_reply ? ok_reply : shared.ok);
}

```

就像上面介绍的那样，其实在之前版本的 Redis 中，由于 `SETNX` 和 `EXPIRE` 并不是 **原子指令**，所以在一起执行会出现问题。

也许你会想到使用 Redis 事务来解决，但在这里不行，因为 `EXPIRE` 命令依赖于 `SETNX` 的执行结果，而事务中没有 `if-else` 的分支逻辑，如果 `SETNX` 没有抢到锁，`EXPIRE` 就不应该执行。

为了解决这个疑难问题，Redis 开源社区涌现了许多分布式锁的 library，为了治理这个乱象，后来在 Redis 2.8 的版本中，加入了 `SET` 指令的扩展参数，使得 `SETNX` 可以和 `EXPIRE` 指令一起执行了：

```
> SET lock:test true ex 5 nx
OK
... do something critical ...
> del lock:test
```

你只需要符合 `SET key value [EX seconds | PX milliseconds] [NX | XX] [KEEPTTL]` 这样的格式就好了。

另外，官方文档也在 `SETNX` 文档中提到了这样一种思路：把 `SETNX` 对应 key 的 value 设置为 `<current Unix time + lock timeout + 1>`，这样在其他客户端访问时就能够自己判断是否能够获取下一个 value 为上述格式的锁了。

3.2.1 代码实现

下面用 Jedis 来模拟实现以下，关键代码如下：

```
private static final String LOCK_SUCCESS = "OK";
private static final Long RELEASE_SUCCESS = 1L;
private static final String SET_IF_NOT_EXIST = "NX";
private static final String SET_WITH_EXPIRE_TIME = "PX";

@Override
public String acquire() {
    try {
        // 获取锁的超时时间，超过这个时间则放弃获取锁
        long end = System.currentTimeMillis() + acquireTimeout;
        // 随机生成一个 value
        String requireToken = UUID.randomUUID().toString();
        while (System.currentTimeMillis() < end) {
            String result = jedis
                .set(lockKey, requireToken, SET_IF_NOT_EXIST,
SET_WITH_EXPIRE_TIME, expireTime);
            if (LOCK_SUCCESS.equals(result)) {
                return requireToken;
            }
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    } catch (Exception e) {
        log.error("acquire lock due to error", e);
    }
    return null;
}
```

```

@Override
public boolean release(String identify) {
    if (identify == null) {
        return false;
    }

    String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return
redis.call('del', KEYS[1]) else return 0 end";
    Object result = new Object();
    try {
        result = jedis.eval(script, Collections.singletonList(lockKey),
            Collections.singletonList(identify));
        if (RELEASE_SUCCESS.equals(result)) {
            log.info("release lock success, requestToken:{}", identify);
            return true;
        }
    } catch (Exception e) {
        log.error("release lock due to error", e);
    } finally {
        if (jedis != null) {
            jedis.close();
        }
    }

    log.info("release lock failed, requestToken:{}, result:{}",
        identify, result);
    return false;
}

```

4. Redlock分布式锁

4.1 什么是 RedLock

Redis 官方站这篇文章提出了一种权威的基于 Redis 实现分布式锁的方式名叫 *Redlock*，此种方式比原先的单节点的方法更安全。它可以保证以下特性：

1. 安全特性：互斥访问，即永远只有一个 client 能拿到锁
2. 避免死锁：最终 client 都可能拿到锁，不会出现死锁的情况，即使原本锁住某资源的 client crash 了或者出现了网络分区
3. 容错性：只要大部分 Redis 节点存活就可以正常提供服务

4.2 怎么在单节点上实现分布式锁

```
| SET resource_name my_random_value NX PX 30000
```

主要依靠上述命令，该命令仅当 Key 不存在时（NX保证）set 值，并且设置过期时间 3000ms（PX保证），值 my_random_value 必须是所有 client 和所有锁请求发生期间唯一的，释放锁的逻辑是：

```

if redis.call("get",KEYS[1]) == ARGV[1] then
    return redis.call("del",KEYS[1])
else
    return 0
end

```

上述实现可以避免释放另一个client创建的锁，如果只有 del 命令的话，那么如果 client1 拿到 lock1 之后因为某些操作阻塞了很长时间，此时 Redis 端 lock1 已经过期了并且已经被重新分配给了 client2，那么 client1 此时再去释放这把锁就会造成 client2 原本获取到的锁被 client1 无故释放了，但现在为每个 client 分配一个 unique 的 string 值可以避免这个问题。至于如何去生成这个 unique string，方法很多随意选择一种就行了。

4.3 Redlock 算法

算法很易懂，起 5 个 master 节点，分布在不同的机房尽量保证可用性。为了获得锁，client 会进行如下操作：

1. 得到当前的时间，微秒单位
2. 尝试顺序地在 5 个实例上申请锁，当然需要使用相同的 key 和 random value，这里一个 client 需要合理设置与 master 节点沟通的 timeout 大小，避免长时间和一个 fail 了的节点浪费时间
3. 当 client 在大于等于 3 个 master 上成功申请到锁的时候，且它会计算申请锁消耗了多少时间，这部分消耗的时间采用获得锁的当下时间减去第一步获得的时间戳得到，如果锁的持续时长（lock validity time）比流逝的时间多的话，那么锁就真正获取到了。
4. 如果锁申请到了，那么锁真正的 lock validity time 应该是 origin (lock validity time) - 申请锁期间流逝的时间
5. 如果 client 申请锁失败了，那么它就会在少部分申请成功锁的 master 节点上执行释放锁的操作，重置状态

4.4 失败重试

如果一个 client 申请锁失败了，那么它需要稍等一会在重试避免多个 client 同时申请锁的情况，最好的情况是一个 client 需要几乎同时向 5 个 master 发起锁申请。另外就是如果 client 申请锁失败了它需要尽快在它曾经申请到锁的 master 上执行 unlock 操作，便于其他 client 获得这把锁，避免这些锁过期造成的时间浪费，当然如果这时候网络分区使得 client 无法联系上这些 master，那么这种浪费就是不得不付出的代价了。

4.5 放锁

放锁操作很简单，就是依次释放所有节点上的锁就行了

4.6 性能、崩溃恢复和 fsync

如果我们的节点没有持久化机制，client 从 5 个 master 中的 3 个处获得了锁，然后其中一个重启了，这是注意 整个环境中又出现了 3 个 master 可供另一个 client 申请同一把锁！ 违反了互斥性。如果我们开启了 AOF 持久化那么情况会稍微好转一些，因为 Redis 的过期机制是语义层面实现的，所以在 server 挂了的时候时间依旧在流逝，重启之后锁状态不会受到污染。但是考虑断电之后呢，AOF 部分命令没来得及刷回磁盘直接丢失了，除非我们配置刷回策略为 fsync = always，但这会损伤性能。解决这个问题的方法是，当一个节点重启之后，我们规定在 max TTL 期间它是不可用的，这样它就不会干扰原本已经申请到的锁，等到它 crash 前的那部分锁都过期了，环境不存在历史锁了，那么再把这个节点加进来正常工作。

5. 如何做可靠的分布式锁，Redlock真的可行么

如果你只是为了性能，那没必要用 Redlock，它成本高且复杂，你只用一个 Redis 实例也够了，最多加个从防止主挂了。当然，你使用单节点的 Redis 那么断电或者一些情况下，你会丢失锁，但是你的目的只是加速性能且断电这种事情不会经常发生，这并不是什么大问题。并且如果你使用了单节点 Redis，那么很显然你这个应用需要的锁粒度是很模糊粗糙的，也不会是什么重要的服务。

那么是否 Redlock 对于要求正确性的场景就合适呢？Martin 列举了若干场景证明 Redlock 这种算法是不可靠的。

5.1 用锁保护资源

这节里 Martin 先将 Redlock 放在了一边而是仅讨论总体上一个分布式锁是怎么工作的。在分布式环境下，锁比 mutex 这类复杂，因为涉及到不同节点、网络通信并且他们随时可能无征兆的 fail。

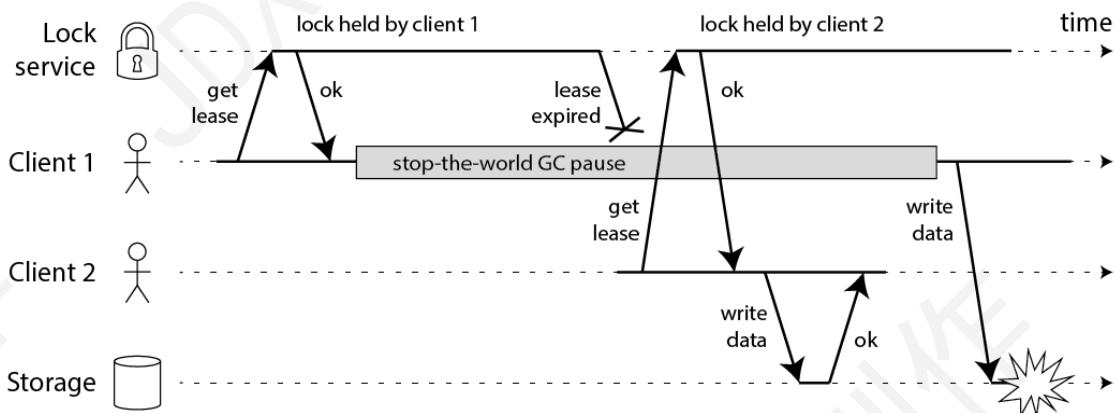
Martin 假设了一个场景，一个 client 要修改一个文件，它先申请得到锁，然后修改文件写回，放锁。

另一个 client 再申请锁 ... 代码流程如下：

```
// THIS CODE IS BROKEN
function writeData(filename, data) {
    var lock = lockService.acquireLock(filename);
    if (!lock) {
        throw 'Failed to acquire lock';
    }

    try {
        var file = storage.readFile(filename);
        var updated = updateContents(file, data);
        storage.writeFile(filename, updated);
    } finally {
        lock.release();
    }
}
```

可惜即使你的锁服务非常完美，上述代码还是可能跪，下面的流程图会告诉你为什么：



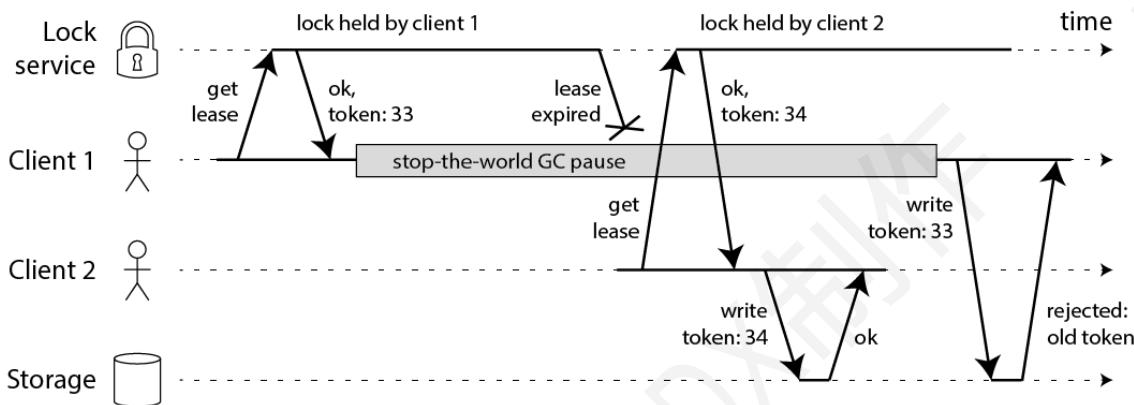
上述图中，得到锁的 client1 在持有锁的期间 pause 了一段时间，例如 GC 停顿。锁有过期时间（一般叫租约，为了防止某个 client 崩溃之后一直占有锁），但是如果 GC 停顿太长超过了锁租约时间，此时锁已经被另一个 client2 所得到，原先的 client1 还没有感知到锁过期，那么奇怪的结果就会发生，曾经 HBase 就发生过这种 Bug。即使你在 client1 写回之前检查一下锁是否过期也无助于解决这个问题，因为 GC 可能在任何时候发生，即使是你非常不便的时候（在最后的检查与写操作期间）。

如果你认为自己的程序不会有长时间的 GC 停顿，还有其他原因会导致你的进程 pause。例如进程可能读取尚未进入内存的数据，所以它得到一个 page fault 并且等待 page 被加载进缓存；还有可能你依赖于网络服务；或者其他进程占用 CPU；或者其他意外发生 SIGSTOP 等。

.... 这里 Martin 又增加了一节列举各种进程 pause 的例子，为了证明上面的代码是不安全的，无论你的锁服务多完美。

5.2 使用 Fencing (栅栏) 使得锁变安全

修复问题的方法也很简单：你需要在每次写操作时加入一个 fencing token。这个场景下，fencing token 可以是一个递增的数字（lock service 可以做到），每次有 client 申请锁就递增一次：



client1 申请锁同时拿到 token33，然后它进入长时间的停顿锁也过期了。client2 得到锁和 token34 写入数据，紧接着 client1 活过来之后尝试写入数据，自身 token33 比 34 小因此写入操作被拒绝。注意这需要存储层来检查 token，但这并不难实现。如果你使用 Zookeeper 作为 lock service 的话那么你可以使用 zxid 作为递增数字。

但是对于 Redlock 你要知道，没什么生成 fencing token 的方式，并且怎么修改 Redlock 算法使其能产生 fencing token 呢？好像并不那么显而易见。因为产生 token 需要单调递增，除非在单节点 Redis 上完成但是这又没有高可靠性，你好像需要引进一致性协议来让 Redlock 产生可靠的 fencing token。

5.3 使用时间来解决一致性

Redlock 无法产生 fencing token 早该成为在需求正确性的场景下弃用它的理由，但还有一些值得讨论的地方。

学术界有个说法，算法对时间不做假设：因为进程可能 pause 一段时间、数据包可能因为网络延迟延后到达、时钟可能根本就是错的。而可靠的算法依旧要在上述假设下做正确的事情。

对于 failure detector 来说，timeout 只能作为猜测某个节点 fail 的依据，因为网络延迟、本地时钟不正确等其他原因的限制。考虑到 Redis 使用 gettimeofday，而不是单调的时钟，会受到系统时间的影响，可能会突然前进或者后退一段时间，这会导致一个 key 更快或更慢地过期。

可见，Redlock 依赖于许多时间假设，它假设所有 Redis 节点都能对同一个 Key 在其过期前持有差不多的时间、跟过期时间相比网络延迟很小、跟过期时间相比进程 pause 很短。

5.4 用不可靠的时间打破 Redlock

这节 Martin 举了个因为时间问题，Redlock 不可靠的例子。

1. client1 从 ABC 三个节点处申请到锁，DE 由于网络原因请求没有到达
2. C 节点的时钟往前推了，导致 lock 过期
3. client2 在 CDE 处获得了锁，AB 由于网络原因请求未到达
4. 此时 client1 和 client2 都获得了锁

在 Redlock 官方文档中也提到了这个情况，不过是 C 崩溃的时候，Redlock 官方本身也是知道 Redlock 算法不是完全可靠的，官方为了解决这种问题建议使用延时启动。但是 Martin 这里分析得更加全面，指出延时启动不也是依赖于时钟的正确性的么？

接下来 Martin 又列举了进程 Pause 时而不是时钟不可靠时会发生的问题：

1. client1 从 ABCDE 处获得了锁
2. 当获得锁的 response 还没到达 client1 时 client1 进入 GC 停顿
3. 停顿期间锁已经过期了
4. client2 在 ABCDE 处获得了锁
5. client1 GC 完成收到了获得锁的 response，此时两个 client 又拿到了同一把锁

同时长时间的网络延迟也有可能导致同样的问题。

5.5 Redlock 的同步性假设

这些例子说明了，仅有在你假设了一个同步性系统模型的基础上，Redlock 才能正常工作，也就是系统能满足以下属性：

1. 网络延时边界，即假设数据包一定能在某个最大延时之内到达
2. 进程停顿边界，即进程停顿一定在某个最大时间之内
3. 时钟错误边界，即不会从一个坏的 NTP 服务器处取得时间

5.6 结论

Martin 认为 Redlock 实在不是一个好的选择，对于需求性能的分布式锁应用它太重了且成本高；对于需求正确性的应用来说它不够安全。因为它对高危的时钟或者说其他上述列举的情况进行了不可靠的假设，如果你的应用只需要高性能的分布式锁不要求多高的正确性，那么单节点 Redis 够了；如果你的应用想要保住正确性，那么不建议 Redlock，建议使用一个合适的一致性协调系统，例如 Zookeeper，且保证存在 fencing token。

6. 神奇的HyperLoglog解决统计问题

6.1 HyperLogLog 简介

HyperLogLog 是最早由Flajolet及其同事在 2007 年提出的一种 **估算基数的近似最优算法**。但跟原版论文不同的是，好像很多书包括 Redis 作者都把它称为一种 **新的数据结构(new datastruct)** (算法实现确实需要一种特定的数据结构来实现)。

6.1.1 关于基数统计

基数统计(Cardinality Counting) 通常是用来统计一个集合中不重复的元素个数。

思考这样的一个场景： 如果你负责开发维护一个大型的网站，有一天老板找产品经理要网站上每个网页的 **UV(独立访客，每个用户每天只记录一次)**，然后让你来开发这个统计模块，你会如何实现？

如果统计 **PV(浏览量，用户没点一次记录一次)**，那非常好办，给每个页面配置一个独立的 Redis 计数器就可以了，把这个计数器的 key 后缀加上当天的日期。这样每来一个请求，就执行 `INCRBY` 指令一次，最终就可以统计出所有的 PV 数据了。

但是 UV 不同，它要去重，**同一个用户一天之内的多次访问请求只能计数一次**。这就要求了每一个网页请求都需要带上用户的 ID，无论是登录用户还是未登录的用户，都需要一个唯一 ID 来标识。

你也许马上就想到了一个 **简单的解决方案**：那就是 **为每一个页面设置一个独立的 set 集合** 来存储所有当天访问过此页面的用户 ID。但这样的 **问题** 就是：

1. **存储空间巨大**：如果网站访问量一大，你需要用来存储的 set 集合就会非常大，如果页面再多.. 为了一个去重功能耗费的资源就可以直接让你 **老板打死你**；
2. **统计复杂**：这么多 set 集合如果要聚合统计一下，又是一个复杂的事情；

6.1.2 基数统计的常用方法

对于上述这样需要 **基数统计** 的事情，通常来说有两种比 set 集合更好的解决方案：

①、第一种：B 树

B 树最大的优势就是插入和查找效率很高，如果用 B 树存储要统计的数据，可以快速判断新来的数据是否存在，并快速将元素插入 B 树。要计算基础值，只需要计算 B 树的节点个数就行了。

不过将 B 树结构维护到内存中，能够解决统计和计算的问题，但是 **并没有节省内存**。

②、第二种：bitmap

bitmap 可以理解为通过一个 bit 数组来存储特定数据的一种数据结构，**每一个 bit 位都能独立包含信息**，bit 是数据的最小存储单位，因此能大量节省空间，也可以将整个 bit 数据一次性 load 到内存计算。如果定义一个很大的 bit 数组，基础统计中 **每一个元素对应到 bit 数组中的一位**，例如：



bitmap 还有一个明显的优势是 **可以轻松合并多个统计结果**，只需要对多个结果求异或就可以了，也可以大大减少存储内存。可以简单做一个计算，如果要统计 **1亿** 个数据的基数值，**大约需要的内存**：

`100_000_000 / 8 / 1024 / 1024 ≈ 12 M`，如果用 **32 bit** 的 int 代表 **每一个** 统计的数据，**大约需要内存**：`32 * 100_000_000 / 8 / 1024 / 1024 ≈ 381 M`

可以看到 bitmap 对于内存的节省显而易见，但仍然不够。统计一个对象的基数值就需要 **12 M**，如果统计 1 万个对象，就需要接近 **120 G**，对于大数据的场景仍然不适用。

6.1.3 概率算法

实际上目前还没有发现更好的在 **大数据场景** 中 **准确计算** 基数的高效算法，因此在不追求绝对精确的情况下，使用概率算法算是一个不错的解决方案。

概率算法 **不直接存储** 数据集合本身，通过一定的 **概率统计方法预估基数值**，这种方法可以大大节省内存，同时保证误差控制在一定范围内。目前用于基数计数的概率算法包括：

- **Linear Counting(LC)**: 早期的基数估计算法，LC 在空间复杂度方面并不算优秀，实际上 LC 的空间复杂度与上文中简单 bitmap 方法是一样的（但是有个常数项级别的降低），都是 $O(N_{\max})$
- **LogLog Counting(LLC)**: LogLog Counting 相比于 LC 更加节省内存，空间复杂度只有 $O(\log_2(\log_2(N_{\max})))$
- **HyperLogLog Counting(HLL)**: HyperLogLog Counting 是基于 LLC 的优化和改进，在同样空间复杂度情况下，能够比 LLC 的基数估计误差更小

其中，**HyperLogLog** 的表现是惊人的，上面我们简单计算过用 **bitmap** 存储 **1个亿** 统计数据大概需要 **12 M** 内存，而在 **HyperLogLog** 中，只需要不到 **1 K** 内存就能够做到！在 Redis 中实现的 **HyperLogLog** 也需要 **12 K** 内存，在 **标准误差 0.81%** 的前提下，**能够统计 2^{64} 个数据**！

这是怎么做到的？！ 下面赶紧来了解一下！

6.2 HyperLogLog 原理

我们来思考一个抛硬币的游戏：你连续掷 n 次硬币，然后说出其中**连续掷为正面的最大次数**，我来猜你**一共抛了多少次**。

这很容易理解吧，例如：你说你这一次 **最多连续出现了2次正面**，那么我就可以知道你这一次投掷的次数并不多，所以 **我可能会猜是5或者6**，但如果说你这一次 **最多连续出现了20次正面**，虽然我觉得不可能，但我仍然知道你花了特别多的时间，所以 **我说GUN...**。

这期间我可能会要求你重复实验，然后我得到了更多的数据之后就会估计得更准。**我们来把刚才的游戏换一种说法：**

掷硬币游戏 – 换一种说法



这张图的意思是，我们给定一系列的随机整数，记录下低位连续零位的最大长度 K，即为图中的 `maxbit`，通过这个 K 值我们就可以估算出随机数的数量 N。

6.2.1 代码实验

我们可以简单编写代码做一个实验，来探究一下 `K` 和 `N` 之间的关系：

```
public class PfTest {

    static class BitKeeper {

        private int maxbit;

        public void random() {
            long value = ThreadLocalRandom.current().nextLong(2L << 32);
            int bit = lowZeros(value);
            if (bit > this.maxbit) {
                this.maxbit = bit;
            }
        }

        private int lowZeros(long value) {
            int i = 0;
            for (; i < 32; i++) {
                if (value >> i << i != value) {
                    break;
                }
            }
            return i - 1;
        }
    }
}
```

```

static class Experiment {

    private int n;
    private BitKeeper keeper;

    public Experiment(int n) {
        this.n = n;
        this.keeper = new BitKeeper();
    }

    public void work() {
        for (int i = 0; i < n; i++) {
            this.keeper.random();
        }
    }

    public void debug() {
        System.out
            .printf("%d %.2f %d\n", this.n, Math.log(this.n) / Math.log(2),
this.keeper.maxbit);
    }
}

public static void main(String[] args) {
    for (int i = 1000; i < 100000; i += 100) {
        Experiment exp = new Experiment(i);
        exp.work();
        exp.debug();
    }
}
}

```

跟上图中的过程是一致的，话说为啥叫 `PfTest` 呢，包括 Redis 中的命令也一样带有一个 `PF` 前缀，还记得嘛，因为 `HyperLogLog` 的提出者上文提到过的，叫 `Philippe Flajolet`。

截取部分输出查看：

```

//n  n/log2 maxbit
34000 15.05 13
35000 15.10 13
36000 15.14 16
37000 15.18 17
38000 15.21 14
39000 15.25 16
40000 15.29 14
41000 15.32 16
42000 15.36 18

```

会发现 `K` 和 `N` 的对数之间存在显著的线性相关性：`N 约等于 2^K`

6.2.2 更进一步：分桶平均

如果 `N` 介于 2^K 和 2^{K+1} 之间，用这种方式估计的值都等于 2^K ，这明显是不合理的，所以我们可以使用多个 `BitKeeper` 进行加权估计，就可以得到一个比较准确的值了：

```

public class PfTest {

    static class BitKeeper {
        // 无变化，代码省略
    }

    static class Experiment {
        private int n;
        private int k;
        private BitKeeper[] keepers;

        public Experiment(int n) {
            this(n, 1024);
        }

        public Experiment(int n, int k) {
            this.n = n;
            this.k = k;
            this.keepers = new BitKeeper[k];
            for (int i = 0; i < k; i++) {
                this.keepers[i] = new BitKeeper();
            }
        }

        public void work() {
            for (int i = 0; i < this.n; i++) {
                long m = ThreadLocalRandom.current().nextLong(1L << 32);
                BitKeeper keeper = keepers[(int) (((m & 0xffff0000) >> 16) % keepers.length)];
                keeper.random();
            }
        }

        public double estimate() {
            double sumbitsInverse = 0.0;
            for (BitKeeper keeper : keepers) {
                sumbitsInverse += 1.0 / (float) keeper.maxbit;
            }
            double avgBits = (float) keepers.length / sumbitsInverse;
            return Math.pow(2, avgBits) * this.k;
        }
    }

    public static void main(String[] args) {
        for (int i = 100000; i < 1000000; i += 100000) {
            Experiment exp = new Experiment(i);
            exp.work();
            double est = exp.estimate();
            System.out.printf("%d %.2f %.2f\n", i, est, Math.abs(est - i) / i);
        }
    }
}

```

这个过程有点 **类似于选秀节目里面的打分**，一堆专业评委打分，但是有一些评委因为自己特别喜欢所以给高了，一些评委又打低了，所以一般都要 **屏蔽最高分和最低分**，然后 **再计算平均值**，这样的出来的分数就差不多是公平公正的了。

上述代码就有 1024 个 "评委"，并且在计算平均值的时候，采用了 **调和平均数**，也就是倒数的平均值，它能有效地平滑离群值的影响：

```
avg = (3 + 4 + 5 + 104) / 4 = 29  
avg = 4 / (1/3 + 1/4 + 1/5 + 1/104) = 5.044
```

观察脚本的输出，误差率百分比控制在个位数：

```
100000 94274.94 0.06  
200000 194092.62 0.03  
300000 277329.92 0.08  
400000 373281.66 0.07  
500000 501551.60 0.00  
600000 596078.40 0.01  
700000 687265.72 0.02  
800000 828778.96 0.04  
900000 944683.53 0.05
```

真实的 HyperLogLog 要比上面的示例代码更加复杂一些，也更加精确一些。上面这个算法在随机次数很少的情况下会出现除零错误，因为 `maxbit = 0` 是不可以求倒数的。

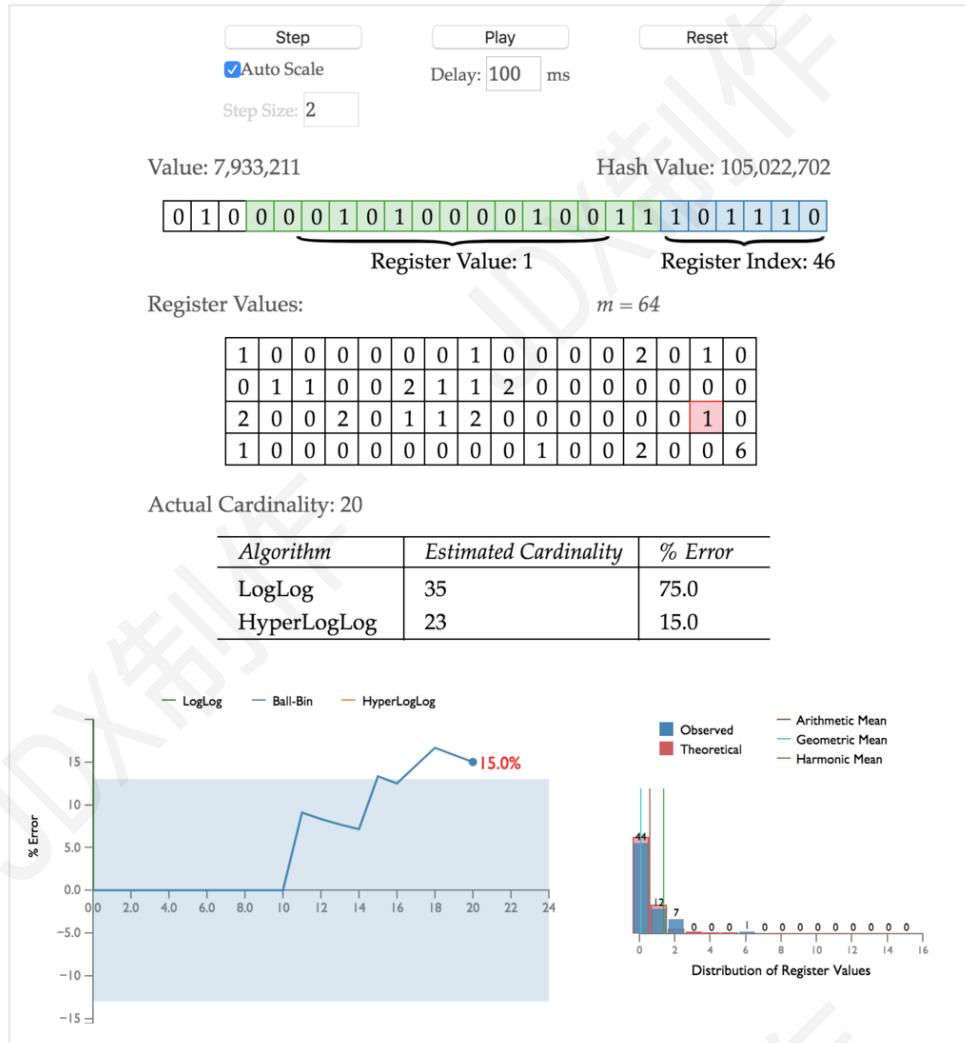
6.2.3 真实的 HyperLogLog

有一个神奇的网站，可以动态地让你观察到 HyperLogLog 的算法到底是怎么执行的：

<http://content.research.neustar.biz/blog/hll.html>

Sketch of the Day: HyperLogLog — Cornerstone of a Big Data Infrastructure

OCTOBER 25, 2012 BY MATT CURCIO LEAVE A COMMENT



其中的一些概念这里稍微解释一下，您就可以自行去点击 `step` 来观察了：

- **m 表示分桶个数：**从图中可以看到，这里分成了 64 个桶；
- **蓝色的 bit 表示在桶中的位置：**例如图中的 101110 实则表示二进制的 46，所以该元素被统计在中间大表格 `Register Values` 中标红的第 46 个桶之中；
- **绿色的 bit 表示第一个 1 出现的位置：**从图中可以看到标绿的 bit 中，从右往左数，第一位就是 1，所以在 `Register Values` 第 46 个桶中写入 1；
- **红色 bit 表示绿色 bit 的值的累加：**下一个出现在第 46 个桶的元素值会被累加；

①、为什么要统计 Hash 值中第一个 1 出现的位置？

因为第一个 1 出现的位置可以同我们抛硬币的游戏中第一次抛到正面的抛掷次数对应起来，根据上面抛硬币实验的结论，记录每个数据的第一个出现的位置 K ，就可以通过其中最大值 K_{\max} 来推导出数据集合中的基数： $N = 2^{K_{\max}}$

②、PF 的内存占用为什么是 12 KB？

我们上面的算法中使用了 1024 个桶，网站演示也只有 64 个桶，不过在 Redis 的 HyperLogLog 实现中，用的是 16384 个桶，即： 2^{14} ，也就是说，就像上面网站中间那个 `Register Values` 大表格有 16384 格。

而 Redis 最大能够统计的数据量是 2^{64} ，即每个桶的 `maxbit` 需要 6 个 bit 来存储，最大可以表示 `maxbit = 63`，于是总共占用内存就是： $(2^{14}) \times 6 / 8$ (每个桶 6 bit，而这么多桶本身要占用 16384 bit，再除以 8 转换成 KB)，算出来的结果就是 12 KB。

6.3 Redis 中的 HyperLogLog 实现

从上面我们算是对 **HyperLogLog** 的算法和思想有了一定的了解，并且知道了一个 **HyperLogLog** 实际占用的空间大约是 12 KB，但 Redis 对于内存的优化非常变态，当 **计数比较小** 的时候，大多数桶的计数值都是零，这个时候 Redis 就会适当节约空间，转换成另外一种 **稀疏存储方式**，与之相对的，正常的存储模式叫做 **密集存储**，这种方式会恒定地占用 12 KB。

6.3.1 密集型存储结构

密集型的存储结构非常简单，就是 16384 个 6 bit 连续串成的字符串位图：



我们都知道，一个字节是由 8 个 bit 组成的，这样 6 bit 排列的结构就会导致，有一些桶会 **跨越字节边界**，我们需要 **对这一个或者两个字节进行适当的移位拼接** 才可以得到具体的计数值。

假设桶的编号为 `index`，这个 6 bit 计数值的起始字节偏移用 `offset_bytes` 表示，它在这个字节的其实比特位置偏移用 `offset_bits` 表示，于是我们有：

```
offset_bytes = (index * 6) / 8  
offset_bits = (index * 6) % 8
```

前者是商，后者是余数。比如 `bucket 2` 的字节偏移是 1，也就是第 2 个字节。它的位偏移是 4，也就是第 2 个字节的第 5 个位开始是 `bucket 2` 的计数值。需要注意的是 **字节位序是左边低位右边高位**，而通常我们使用的字节都是左边高位右边低位。

这里就涉及到两种情况，如果 `offset_bits` 小于等于 2，说明这 6 bit 在一个字节的内部，可以直接使用下面的表达式得到计数值 `val`：

```
val = buffer[offset_bytes] >> offset_bits # 向右移位
```

如果 `offset_bits` 大于 2，那么就会涉及到 **跨越字节边界**，我们需要拼接两个字节的位片段：

```
# 低位值  
low_val = buffer[offset_bytes] >> offset_bits  
# 低位个数  
low_bits = 8 - offset_bits  
# 拼接，保留低6位  
val = (high_val << low_bits | low_val) & 0b111111
```

不过下面 Redis 的源码要晦涩一点，看形式它似乎只考虑了跨越字节边界的情况。这是因为如果 6 bit 在单个字节内，上面代码中的 `high_val` 的值是零，所以这一份代码可以同时照顾单字节和双字节：

```
// 获取指定桶的计数值  
#define HLL_DENSE_GET_REGISTER(target,p,regnum) do { \  
    uint8_t *_p = (uint8_t*) p; \  
    unsigned long _byte = regnum*HLL_BITS/8; \  
    if (_byte < 8) { \  
        target = *_p >> (7 - _byte); \  
    } else { \  
        target = ((*_p >> (8 - _byte)) << 8) | (*(_p+1) >> _byte); \  
    } \  
}
```

```

unsigned long _fb = regnum*HLL_BITS&7; \
unsigned long _fb8 = 8 - _fb; \
unsigned long b0 = _p[_byte]; \
unsigned long b1 = _p[_byte+1]; \
target = ((b0 >> _fb) | (b1 << _fb8)) & HLL_REGISTER_MAX; \
} while(0)

// 设置指定桶的计数值
#define HLL_DENSE_SET_REGISTER(p,regnum,val) do { \
    uint8_t *_p = (uint8_t*) p; \
    unsigned long _byte = regnum*HLL_BITS/8; \
    unsigned long _fb = regnum*HLL_BITS&7; \
    unsigned long _fb8 = 8 - _fb; \
    unsigned long _v = val; \
    _p[_byte] &= ~(HLL_REGISTER_MAX << _fb); \
    _p[_byte] |= _v << _fb; \
    _p[_byte+1] &= ~(HLL_REGISTER_MAX >> _fb8); \
    _p[_byte+1] |= _v >> _fb8; \
} while(0)

```

6.3.2 稀疏存储结构

稀疏存储适用于很多计数值都是零的情况。下图表示了一般稀疏存储计数值的状态：



当 **多个连续桶的计数值都是零** 时，Redis 提供了几种不同的表达形式：

- **00xxxxxx**：前缀两个零表示接下来的 6bit 整数值加 1 就是零值计数器的数量，注意这里要加 1 是因为数量如果为零是没有意义的。比如 **00010101** 表示连续 **22** 个零值计数器。
- **01xxxxxxxx yyyy-yyyy**：6bit 最多只能表示连续 **64** 个零值计数器，这样扩展出的 14bit 可以表示最多连续 **16384** 个零值计数器。这意味着 HyperLogLog 数据结构中 **16384** 个桶的初始状态，所有的计数器都是零值，可以直接使用 2 个字节来表示。
- **1vvvvvxx**：中间 5bit 表示计数值，尾部 2bit 表示连续几个桶。它的意思是连续 **(xx +1)** 个计数值都是 **(vvvv + 1)**。比如 **10101011** 表示连续 **4** 个计数值都是 **11**。

注意 上面第三种方式的计数值最大只能表示到 **32**，而 HyperLogLog 的密集存储单个计数值用 6bit 表示，最大可以表示到 **63**。当稀疏存储的某个计数值需要调整到大于 **32** 时，Redis 就会立即转换 HyperLogLog 的存储结构，将稀疏存储转换成密集存储。

6.3.3 对象头

HyperLogLog 除了需要存储 16384 个桶的计数值之外，它还有一些附加的字段需要存储，比如总计数缓存、存储类型。所以它使用了一个额外的对象头来表示：

```
struct hllhdr {
    char magic[4];      /* 魔术字符串"HYLL" */
    uint8_t encoding;   /* 存储类型 HLL_DENSE or HLL_SPARSE. */
    uint8_t notused[3]; /* 保留三个字节未来可能会使用 */
    uint8_t card[8];    /* 总计数缓存 */
    uint8_t registers[]; /* 所有桶的计数器 */
};
```

所以 **HyperLogLog** 整体的内部结构就是 **HLL 对象头** 加上 16384 个桶的计数值位图。它在 Redis 的内部结构表现就是一个字符串位图。你可以把 **HyperLogLog** 对象当成普通的字符串来进行处理：

```
> PFADD codehole python java golang
(integer) 1
> GET codehole
"HYLL\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x80C\x03\x84MK\x80P\xb8\x80^\\xf3"
```

但是 **不可以** 使用 **HyperLogLog** 指令来 **操纵普通的字符串**，因为它需要检查对象头魔术字符串是否是 "HYLL"。

6.4 HyperLogLog 的使用

HyperLogLog 提供了两个指令 **PFADD** 和 **PFCOUNT**，字面意思就是一个是增加，另一个是获取计数。**PFADD** 和 **set** 集合的 **SADD** 的用法是一样的，来一个用户 ID，就将用户 ID 塞进去就是，**PFCOUNT** 和 **SCARD** 的用法是一致的，直接获取计数值：

```
> PFADD codehole user1
(integer) 1
> PFCOUNT codehole
(integer) 1
> PFADD codehole user2
(integer) 1
> PFCOUNT codehole
(integer) 2
> PFADD codehole user3
(integer) 1
> PFCOUNT codehole
(integer) 3
> PFADD codehole user4 user 5
(integer) 1
> PFCOUNT codehole
(integer) 5
```

我们可以用 Java 编写一个脚本来试试 **HyperLogLog** 的准确性到底有多少：

```
public class JedisTest {  
    public static void main(String[] args) {  
        for (int i = 0; i < 100000; i++) {  
            jedis.pfadd("codehole", "user" + i);  
        }  
        long total = jedis.pfcnt("codehole");  
        System.out.printf("%d %d\n", 100000, total);  
        jedis.close();  
    }  
}
```

结果输出如下：

```
100000 99723
```

发现 10 万条数据只差了 277，按照百分比误差率是 0.277%，对于巨量的 UV 需求来说，这个误差率真的不算高。

当然，除了上面的 `PFADD` 和 `PFCOUNT` 之外，还提供了第三个 `PFMERGE` 指令，用于将多个计数值累加在一起形成一个新的 `pf` 值：

```
> PFADD nosql "Redis" "MongoDB" "Memcached"  
(integer) 1  
  
> PFADD RDBMS "MySQL" "MSSQL" "PostgreSQL"  
(integer) 1  
  
> PFMERGE databases nosql RDBMS  
OK  
  
> PFCOUNT databases  
(integer) 6
```

7. 亿级数据过滤和布隆过滤器

7.1 布隆过滤器简介

上一次我们学会了使用 `HyperLogLog` 来对大数据进行一个估算，它非常有价值，可以解决很多精确度不高的统计需求。但是如果我们想知道某一个值是不是已经在 `HyperLogLog` 结构里面了，它就无能为力了，它只提供了 `pfadd` 和 `pfcnt` 方法，没有提供类似于 `contains` 的这种方法。

就举一个场景吧，比如你 **刷抖音**：

你有 **刷到过重复的推荐内容** 吗？这么多的推荐内容要推荐给这么多的用户，它是怎么保证每个用户在看推荐内容时，保证不会出现之前已经看过的推荐视频呢？也就是说，抖音是如何实现 **推送去重** 的呢？

你会想到服务器 **记录** 了用户看过的 **所有历史记录**，当推荐系统推荐短视频时会从每个用户的历史记录里进行 **筛选**，过滤掉那些已经存在的记录。问题是当 **用户量很大**，每个用户看过的短视频又很多的情况下，这种方式，推荐系统的去重工作 **在性能上跟的上么？**

实际上，如果历史记录存储在关系数据库里，去重就需要频繁地对数据库进行 `exists` 查询，当系统并发量很高时，数据库是很难抗拒压力的。

你可能又想到了 **缓存**，但是这么多用户这么多的历史记录，如果全部缓存起来，那得需要 **浪费多大的空间** 啊.. (可能老板看一眼账单，看一眼你..) 并且这个存储空间会随着时间呈线性增长，就算你用缓存撑得住一个月，但是又能继续撑多久呢？不缓存性能又跟不上，咋办呢？



如上图所示，**布隆过滤器(Bloom Filter)** 就是这样一种专门用来解决去重问题的高级数据结构。但是跟 **HyperLogLog** 一样，它也一样有那么一点点不精确，也存在一定的误判概率，但它能在解决去重的同时，在 **空间上能节省 90% 以上**，也是非常值得的。

7.1.1 布隆过滤器是什么

布隆过滤器 (Bloom Filter) 是 1970 年由布隆提出的。它 **实际上** 是一个很长的二进制向量和一系列随机映射函数 (下面详细说)，实际上你也可以把它 **简单理解** 为一个不怎么精确的 **set** 结构，当你使用它的 `contains` 方法判断某个对象是否存在时，它可能会误判。但是布隆过滤器也不是特别不精确，只要参数设置的合理，它的精确度可以控制的相对足够精确，只会有小小的误判概率。

当布隆过滤器说某个值存在时，这个值 **可能存在**；当它说不存在时，那么 **一定不存在**。打个比方，当它说不认识你时，那就是真的不认识，但是当它说认识你的时候，可能是因为你长得像它认识的另外一个朋友 (脸长得有些相似)，所以误判认识你。

7.1.2 布隆过滤器的使用场景

基于上述的功能，我们大致可以把布隆过滤器用于以下的场景之中：

- **大数据判断是否存在**：这就可以实现上述的去重功能，如果你的服务器内存足够大的话，那么使用 `HashMap` 可能是一个不错的解决方案，理论上时间复杂度可以达到 $O(1)$ 的级别，但是当数据量起来之后，还是只能考虑布隆过滤器。
- **解决缓存穿透**：我们经常会把一些热点数据放在 `Redis` 中当作缓存，例如产品详情。通常一个请求过来之后我们会先查询缓存，而不用直接读取数据库，这是提升性能最简单也是最普遍的做法，但是 **如果一直请求一个不存在的缓存**，那么此时一定不存在缓存，那就会有 **大量请求直接打到数据库上**，造成 **缓存穿透**，布隆过滤器也可以用来解决此类问题。
- **爬虫/邮箱等系统的过滤**：平时不知道你有没有注意到有一些正常的邮件也会被放进垃圾邮件目录中，这就是使用布隆过滤器 **误判** 导致的。

7.2 布隆过滤器原理解析

布隆过滤器 **本质上** 是由长度为 m 的位向量或位列表（仅包含 `0` 或 `1` 位值的列表）组成，最初所有的值均设置为 `0`，所以我们先来创建一个稍微长一些的位向量用作展示：

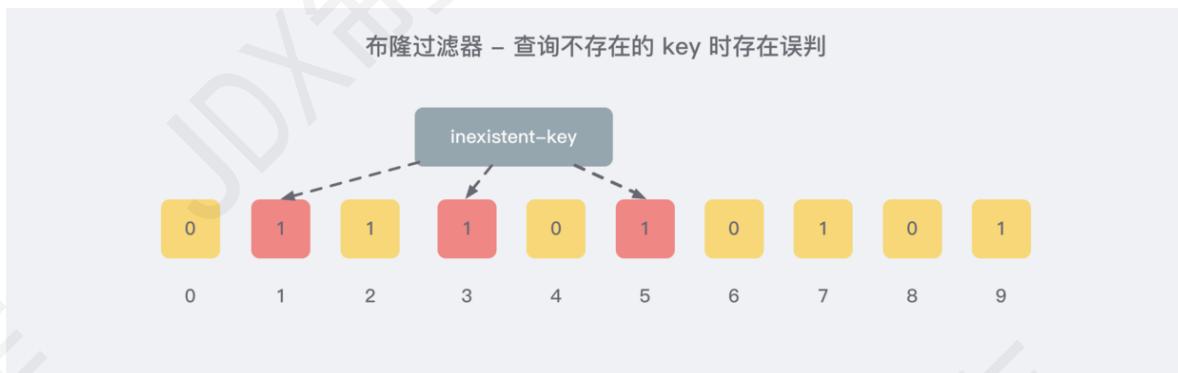


当我们向布隆过滤器中添加数据时，会使用 **多个** `hash` 函数对 `key` 进行运算，算得一个证书索引值，然后对位数组长度进行取模运算得到一个位置，每个 `hash` 函数都会算得一个不同的位置。再把位数组的这几个位置都置为 `1` 就完成了 `add` 操作，例如，我们添加一个 `wmyskxz`：



向布隆过滤器查查询 `key` 是否存在时，跟 `add` 操作一样，会把这个 `key` 通过相同的多个 `hash` 函数进行运算，查看 **对应的位置是否都为 1**，只要有一个位为 `0`，那么说明布隆过滤器中这个 `key` 不存在。如果这几个位置都是 `1`，并不能说明这个 `key` 一定存在，只能说极有可能存在，因为这些位置的 `1` 可能是因为其他的 `key` 存在导致的。

就比如我们在 `add` 了一定的数据之后，查询一个 **不存在** 的 `key`：



很明显，`1/3/5` 这几个位置的 `1` 是因为上面第一次添加的 `wmyskxz` 而导致的，所以这里就存在 **误判**。幸运的是，布隆过滤器有一个可以预判误判率的公式，比较复杂，感兴趣的朋友可以自行去阅读，比较烧脑.. 只需要记住以下几点就好了：

- 使用时 **不要让实际元素数量远大于初始化数量**；
- 当实际元素数量超过初始化数量时，应该对布隆过滤器进行 **重建**，重新分配一个 `size` 更大的过滤器，再将所有的历史元素批量 `add` 进行；

7.3 布隆过滤器的使用

Redis 官方 提供的布隆过滤器到了 Redis 4.0 提供了插件功能之后才正式登场。布隆过滤器作为一个插件加载到 Redis Server 中，给 Redis 提供了强大的布隆去重功能。下面我们来体验一下 Redis 4.0 的布隆过滤器，为了省去繁琐安装过程，我们直接用 Docker 吧。

```
> docker pull redislabs/rebloom # 拉取镜像
> docker run -p6379:6379 redislabs/rebloom # 运行容器
> redis-cli # 连接容器中的 redis 服务
```

如果上面三条指令执行没有问题，下面就可以体验布隆过滤器了。

- 当然，如果你不想使用 Docker，也可以在检查本机 Redis 版本合格之后自行安装插件。

7.3.1 布隆过滤器的基本用法

布隆过滤器有两个基本指令，`bf.add` 添加元素，`bf.exists` 查询元素是否存在，它的用法和 set 集合的 `sadd` 和 `sismember` 差不多。注意 `bf.add` 只能一次添加一个元素，如果想要一次添加多个，就需要用到 `bf.madd` 指令。同样如果需要一次查询多个元素是否存在，就需要用到 `bf.mexists` 指令。

```
127.0.0.1:6379> bf.add codehole user1
(integer) 1
127.0.0.1:6379> bf.add codehole user2
(integer) 1
127.0.0.1:6379> bf.add codehole user3
(integer) 1
127.0.0.1:6379> bf.exists codehole user1
(integer) 1
127.0.0.1:6379> bf.exists codehole user2
(integer) 1
127.0.0.1:6379> bf.exists codehole user3
(integer) 1
127.0.0.1:6379> bf.exists codehole user4
(integer) 0
127.0.0.1:6379> bf.madd codehole user4 user5 user6
1) (integer) 1
2) (integer) 1
3) (integer) 1
127.0.0.1:6379> bf.mexists codehole user4 user5 user6 user7
1) (integer) 1
2) (integer) 1
3) (integer) 1
4) (integer) 0
```

上面使用的布隆过滤器只是默认参数的布隆过滤器，它在我们第一次 `add` 的时候自动创建。Redis 也提供了可以自定义参数的布隆过滤器，只需要在 `add` 之前使用 `bf.reserve` 指令显式创建就好了。如果对应的 `key` 已经存在，`bf.reserve` 会报错。

`bf.reserve` 有三个参数，分别是 `key`、`error_rate` (错误率) 和 `initial_size`:

- `error_rate` 越低，需要的空间越大，对于不需要过于精确的场合，设置稍大一些也没有关系，比如上面说的推送系统，只会让一小部分的内容被过滤掉，整体的观看体验还是不会受到很大影响的；
- `initial_size` 表示预计放入的元素数量，当实际数量超过这个值时，误判率就会提升，所以需要提前设置一个较大的数值避免超出导致误判率升高；

如果不适用 `bf.reserve`，默认的 `error_rate` 是 `0.01`，默认的 `initial_size` 是 `100`。

7.4 布隆过滤器代码实现

7.4.1 自己简单模拟实现

根据上面的基础理论，我们很容易就可以自己实现一个用于 [简单模拟] 的布隆过滤器数据结构：

```
public static class BloomFilter {  
  
    private byte[] data;  
  
    public BloomFilter(int initSize) {  
        this.data = new byte[initSize * 2]; // 默认创建大小 * 2 的空间  
    }
```

```

public void add(int key) {
    int location1 = Math.abs(hash1(key) % data.length);
    int location2 = Math.abs(hash2(key) % data.length);
    int location3 = Math.abs(hash3(key) % data.length);

    data[location1] = data[location2] = data[location3] = 1;
}

public boolean contains(int key) {
    int location1 = Math.abs(hash1(key) % data.length);
    int location2 = Math.abs(hash2(key) % data.length);
    int location3 = Math.abs(hash3(key) % data.length);

    return data[location1] * data[location2] * data[location3] == 1;
}

private int hash1(Integer key) {
    return key.hashCode();
}

private int hash2(Integer key) {
    int hashCode = key.hashCode();
    return hashCode ^ (hashCode >>> 3);
}

private int hash3(Integer key) {
    int hashCode = key.hashCode();
    return hashCode ^ (hashCode >>> 16);
}
}

```

这里很简单，内部仅维护了一个 `byte` 类型的 `data` 数组，实际上 `byte` 仍然占有一个字节之多，可以优化成 `bit` 来代替，这里也仅仅是用于方便模拟。另外我也创建了三个不同的 `hash` 函数，其实也就是借鉴 `HashMap` 哈希抖动的办法，分别使用自身的 `hash` 和右移不同位数相异或的结果。并且提供了基础的 `add` 和 `contains` 方法。

下面我们来简单测试一下这个布隆过滤器的效果如何：

```

public static void main(String[] args) {
    Random random = new Random();
    // 假设我们的数据有 1 百万
    int size = 1_000_000;
    // 用一个数据结构保存一下所有实际存在的值
    LinkedList<Integer> existentNumbers = new LinkedList<>();
    BloomFilter bloomFilter = new BloomFilter(size);

    for (int i = 0; i < size; i++) {
        int randomKey = random.nextInt();
        existentNumbers.add(randomKey);
        bloomFilter.add(randomKey);
    }

    // 验证已存在的数是否都存在
    AtomicInteger count = new AtomicInteger();
    AtomicInteger finalCount = count;
    existentNumbers.forEach(number -> {
        if (bloomFilter.contains(number)) {

```

```

        finalCount.incrementAndGet();
    }
});

System.out.printf("实际的数据量: %d, 判断存在的数据量: %d \n", size,
count.get());

// 验证10个不存在的数
count = new AtomicInteger();
while (count.get() < 10) {
    int key = random.nextInt();
    if (existentNumbers.contains(key)) {
        continue;
    } else {
        // 这里一定是不存在的数
        System.out.println(bloomFilter.contains(key));
        count.incrementAndGet();
    }
}
}
}

```

输出如下：

```

实际的数据量: 1000000, 判断存在的数据量: 1000000
false
true
false
true
true
true
false
false
true
false

```

这就是前面说到的，当布隆过滤器说某个值 **存在时**，这个值 **可能不存在**，当它说某个值 **不存在时**，那就 **肯定不存在**，并且还有一定的误判率...

7.4.2 手动实现参考

当然上面的版本特别 low，不过主体思想是不差的，这里也给出一个好一些的版本用作自己实现测试的参考：

```

import java.util.BitSet;

public class MyBloomFilter {

    /**
     * 位数组的大小
     */
    private static final int DEFAULT_SIZE = 2 << 24;
    /**
     * 通过这个数组可以创建 6 个不同的哈希函数
     */
    private static final int[] SEEDS = new int[]{3, 13, 46, 71, 91, 134};

    /**
     * 位数组。数组中的元素只能是 0 或者 1
     */
}

```

```
/*
private BitSet bits = new BitSet(DEFAULT_SIZE);

/**
 * 存放包含 hash 函数的类的数组
 */
private SimpleHash[] func = new SimpleHash[SEEDS.length];

/**
 * 初始化多个包含 hash 函数的类的数组，每个类中的 hash 函数都不一样
 */
public MyBloomFilter() {
    // 初始化多个不同的 Hash 函数
    for (int i = 0; i < SEEDS.length; i++) {
        func[i] = new SimpleHash(DEFAULT_SIZE, SEEDS[i]);
    }
}

/**
 * 添加元素到位数组
 */
public void add(Object value) {
    for (SimpleHash f : func) {
        bits.set(f.hash(value), true);
    }
}

/**
 * 判断指定元素是否存在于位数组
 */
public boolean contains(Object value) {
    boolean ret = true;
    for (SimpleHash f : func) {
        ret = ret && bits.get(f.hash(value));
    }
    return ret;
}

/**
 * 静态内部类。用于 hash 操作！
 */
public static class SimpleHash {
    private int cap;
    private int seed;

    public SimpleHash(int cap, int seed) {
        this.cap = cap;
        this.seed = seed;
    }

    /**
     * 计算 hash 值
     */
    public int hash(Object value) {
        int h;
        return (value == null) ? 0 : Math.abs(seed * (cap - 1) & ((h =
value.hashCode()) ^ (h >> 16)));
    }
}
```

```
    }  
  
}  
}
```

7.4.3 使用 Google 开源的 Guava 中自带的布隆过滤器

自己实现的目的主要是为了让自己搞懂布隆过滤器的原理，Guava 中布隆过滤器的实现算是比较权威的，所以实际项目中我们不需要手动实现一个布隆过滤器。

首先我们需要在项目中引入 Guava 的依赖：

```
<dependency>  
    <groupId>com.google.guava</groupId>  
    <artifactId>guava</artifactId>  
    <version>28.0-jre</version>  
</dependency>
```

实际使用如下：

我们创建了一个最多存放 最多 1500 个整数的布隆过滤器，并且我们可以容忍误判的概率为百分之 (0.01)

```
// 创建布隆过滤器对象  
BloomFilter<Integer> filter = BloomFilter.create(  
    Funnels.integerFunnel(),  
    1500,  
    0.01);  
  
// 判断指定元素是否存在  
System.out.println(filter.mightContain(1));  
System.out.println(filter.mightContain(2));  
  
// 将元素添加进布隆过滤器  
filter.put(1);  
filter.put(2);  
System.out.println(filter.mightContain(1));  
System.out.println(filter.mightContain(2));
```

在我们的示例中，当 `mightContain()` 方法返回 `true` 时，我们可以 **99%** 确定该元素在过滤器中，当过滤器返回 `false` 时，我们可以 **100%** 确定该元素不存在于过滤器中。

Guava 提供的布隆过滤器的实现还是很不错的（想要详细了解的可以看一下它的源码实现），但是它有一个重大的缺陷就是只能单机使用（另外，容量扩展也不容易），而现在互联网一般都是分布式的场景。为了解决这个问题，我们就需要用到 Redis 中的布隆过滤器了。

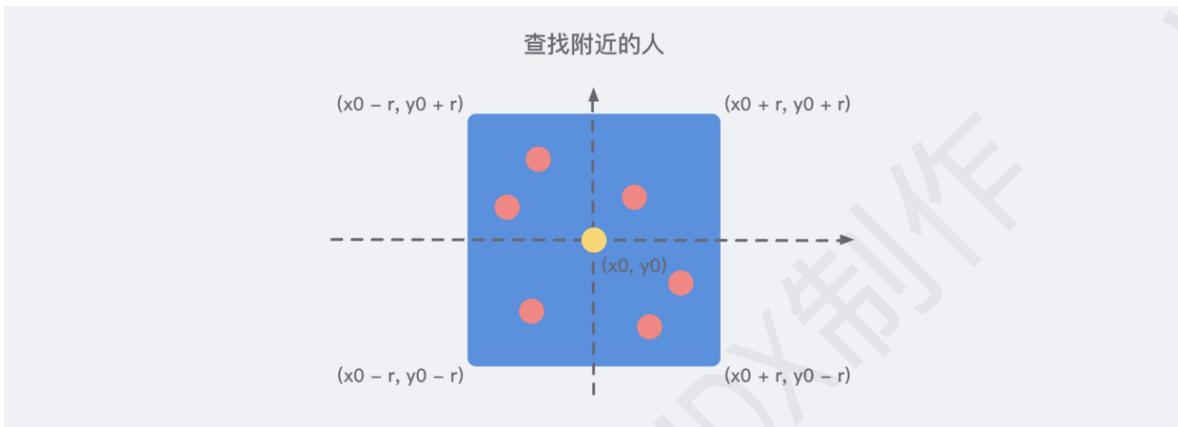
8. GeoHash 查找附近的人

像微信 "附近的人"，美团 "附近的餐厅"，支付宝共享单车 "附近的车" 是怎么设计实现的呢？

8.1 使用数据库实现查找附近的人

我们都知道，地球上的任何一个位置都可以使用二维的 **经纬度** 来表示，经度范围 $[-180, 180]$ ，纬度范围 $[-90, 90]$ ，纬度正负以赤道为界，北正南负，经度正负以本初子午线（英国格林尼治天文台）为界，东正西负。比如说，北京人民英雄纪念碑的经纬度坐标就是 $(39.904610, 116.397724)$ ，都是正数，因为中国位于东北半球。

所以，当我们使用数据库存储了所有人的 **经纬度** 信息之后，我们就可以基于当前的坐标节点，来划分出一个矩形的范围，来得知附近的人，如下图：



所以，我们很容易写出下列的伪 SQL 语句：

```
SELECT id FROM positions WHERE x0 - r < x < x0 + r AND y0 - r < y < y0 + r
```

如果我们还想进一步地知道与每个坐标元素的距离并排序的话，就需要一定的计算。

当两个坐标元素的距离不是很远的时候，我们就可以简单利用 **勾股定理** 就能够得出他们之间的 **距离**。不过需要注意的是，地球不是一个标准的球体，**经纬度的密度是不一样的**，所以我们使用勾股定理计算平方之后再求和时，需要按照一定的系数 **加权** 再进行求和。当然，如果不准求精确的话，加权也不必了。

参考下方 [参考资料2](#) 我们能够差不多能写出如下优化之后的 SQL 语句来：(仅供参考)

```
SELECT
  *
FROM
  users_location
WHERE
  latitude > '.$lat.' - 1
  AND latitude < '.$lat.' + 1 AND longitude > '.$lon.' - 1
  AND longitude < '.$lon.' + 1
ORDER BY
  ACOS(
    SIN( ('.$lat.' * 3.1415) / 180 ) * SIN( (latitude * 3.1415) / 180 )
    + COS( ('.$lat.' * 3.1415) / 180 ) * COS( (latitude * 3.1415) / 180 ) * COS(
      ('.$lon.' * 3.1415) / 180 - (longitude * 3.1415) / 180 )
  ) * 6380 ASC
LIMIT 10 ;
```

为了满足高性能的矩形区域算法，数据表也需要把经纬度坐标加上 **双向复合索引(x, y)**，这样可以满足最大优化查询性能。

8.2 GeoHash 算法简述

这是业界比较通用的，用于 **地理位置距离排序** 的一个算法，Redis 也采用了这样的算法。GeoHash 算法将 **二维的经纬度** 数据映射到 **一维** 的整数，这样所有的元素都将在挂载到一条线上，距离靠近的二维坐标映射到一维后的点之间距离也会很接近。当我们想要计算「附近的人时」，首先将目标位置映射到这条线上，然后在这个一维的线上获取附近的点就行了。

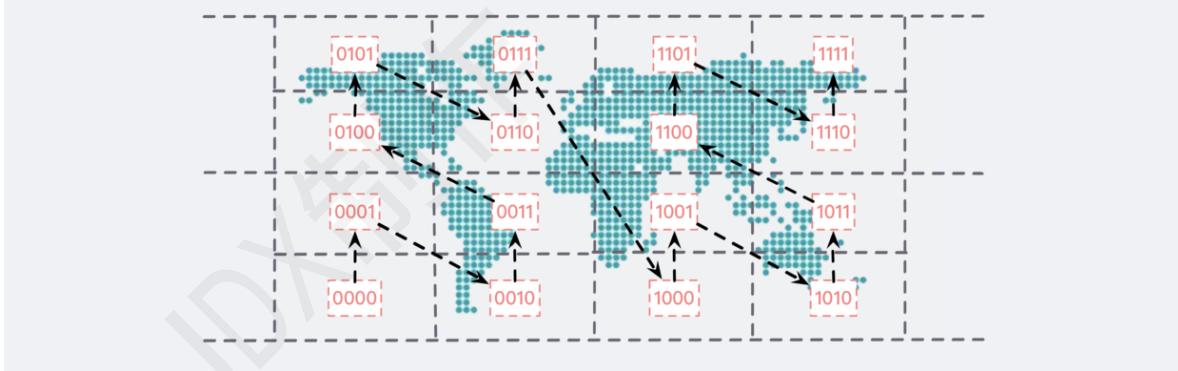
它的核心思想就是把整个地球看成是一个 **二维的平面**，然后把这个平面不断地等分成一个一个方格，**每一个** 坐标元素都位于其中的 **唯一一个方格** 中，等分之后的 **方格越小**，那么坐标也就 **越精确**，类似下图：

GeoHash 算法核心思想 1 – 给地球划小方格



经过划分的地球，我们需要对其进行编码：

GeoHash 算法核心思想 2 – 区域编码



经过这样顺序的编码之后，如果你仔细观察一会儿，你就会发现一些规律：

- 横着的所有编码中，**第 2 位和第 4 位都是一样的**，例如第一排第一个 0101 和第二个 0111，他们的第 2 位和第 4 位都是 1；
- 竖着的所有编码中，**第 1 位和第 3 位是递增的**，例如第一排第一个 0101，如果单独把第 1 位和第 3 位拎出来的话，那就是 00，同理看第一排第二个 0111，同样的方法第 1 位和第 3 位拎出来是 01，刚好是 00 递增一个；

通过这样的规律我们就把每一个小方块儿进行了一定顺序的编码，这样做的 **好处** 是显而易见的：每一个元素坐标既能够被 **唯一标识** 在这张被编码的地图上，也不至于 **暴露特别的具体的位置**，因为区域是共享的，我可以告诉你我就在公园附近，但是在具体的哪个地方你就无从得知了。

总之，我们通过上面的思想，能够把任意坐标变成一串二进制的编码了，类似于 11010010110001000100 这样（注意经度和维度是交替出现的哦.），通过这个整数我们就可以还原出元素的坐标，整数越长，还原出来的坐标值的损失程序就越小。对于 “附近的人” 这个功能来说，损失的一点经度可以忽略不计。

最后就是一个 Base32 (0~9, a~z, 去掉a/i/l/o 四个字母) 的编码操作，让它变成一个字符串，例如上面那一串儿就变成了 wx4g0ec1。

在 Redis 中，经纬度使用 52 位的整数进行编码，放进了 zset 里面，zset 的 value 是元素的 key，score 是 GeoHash 的 52 位整数值。zset 的 score 虽然是浮点数，但是对于 52 位的整数值来说，它可以无损存储。

8.3 在 Redis 中使用 Geo

| 下方内容引自 参考资料 1 - 《Redis 深度历险》

在使用 Redis 进行 Geo 查询 时，我们要时刻想到它的内部结构实际上只是一个 zset(skiplist)。通过 zset 的 score 排序就可以得到坐标附近的其他元素（实际情况要复杂一些，不过这样理解足够了），通过将 score 还原成坐标值就可以得到元素的原始坐标了。

Redis 提供的 Geo 指令只有 6 个，很容易就可以掌握。

8.3.1 增加

`geoadd` 指令携带集合名称以及多个经纬度名称三元组，注意这里可以加入多个三元组。

```
127.0.0.1:6379> geoadd company 116.48105 39.996794 juejin
(integer) 1
127.0.0.1:6379> geoadd company 116.514203 39.905409 ireader
(integer) 1
127.0.0.1:6379> geoadd company 116.489033 40.007669 meituan
(integer) 1
127.0.0.1:6379> geoadd company 116.562108 39.787602 jd 116.334255 40.027400
xiaomi
(integer) 2
```

不过很奇怪.. Redis 没有直接提供 Geo 的删除指令，但是我们可以通过 zset 相关的指令来操作 Geo 数据，所以元素删除可以使用 `zrem` 指令即可。

8.3.2 距离

`geodist` 指令可以用来计算两个元素之间的距离，携带集合名称、2 个名称和距离单位。

```
127.0.0.1:6379> geodist company juejin ireader km
"10.5501"
127.0.0.1:6379> geodist company juejin meituan km
"1.3878"
127.0.0.1:6379> geodist company juejin jd km
"24.2739"
127.0.0.1:6379> geodist company juejin xiaomi km
"12.9606"
127.0.0.1:6379> geodist company juejin juejin km
"0.0000"
```

我们可以看到掘金离美团最近，因为它们都在望京。距离单位可以是 `m`、`km`、`mi`、`ft`，分别代表米、千米、英里和尺。

8.3.3 获取元素位置

`geopos` 指令可以获取集合中任意元素的经纬度坐标，可以一次获取多个。

```
127.0.0.1:6379> geopos company juejin
1) 1) "116.48104995489120483"
   2) "39.99679348858259686"
127.0.0.1:6379> geopos company ireader
1) 1) "116.5142020583152771"
   2) "39.90540918662494363"
127.0.0.1:6379> geopos company juejin ireader
1) 1) "116.48104995489120483"
   2) "39.99679348858259686"
   2) 1) "116.5142020583152771"
   2) "39.90540918662494363"
```

我们观察到获取的经纬度坐标和 `geoadd` 进去的坐标有轻微的误差，原因是 **Geohash** 对二维坐标进行的一维映射是有损的，通过映射再还原回来的值会出现较小的差别。对于「附近的人」这种功能来说，这点误差根本不是事。

8.3.4 获取元素的 hash 值

`geohash` 可以获取元素的经纬度编码字符串，上面已经提到，它是 `base32` 编码。你可以使用这个编码值去 [http://geohash.org/\\${hash}](http://geohash.org/${hash}) 中进行直接定位，它是 **Geohash** 的标准编码值。

```
127.0.0.1:6379> geohash company ireader
1) "wx4g52e1ce0"
127.0.0.1:6379> geohash company juejin
1) "wx4gd94yjn0"
```

让我们打开地址 <http://geohash.org/wx4g52e1ce0>，观察地图指向的位置是否正确：



[Tips & Tricks](#) — This service isn't affiliated with any of the linked sites — geohash@geohash.org

很好，就是这个位置，非常准确。

8.3.5 附近的公司

`georadiusbymember` 指令是最为关键的指令，它可以用来查询指定元素附近的其它元素，它的参数非常复杂。

```
# 范围 20 公里以内最多 3 个元素按距离正排，它不会排除自身
127.0.0.1:6379> georadiusbymember company ireader 20 km count 3 asc
1) "ireader"
2) "juejin"
3) "meituan"
# 范围 20 公里以内最多 3 个元素按距离倒排
```

```
127.0.0.1:6379> georadiusbymember company ireader 20 km count 3 desc
1) "jd"
2) "meituan"
3) "juejin"
# 三个可选参数 withcoord withdist withhash 用来携带附加参数
# withdist 很有用，它可以用来显示距离
127.0.0.1:6379> georadiusbymember company ireader 20 km withcoord withdist
withhash count 3 asc
1) 1) "ireader"
2) "0.0000"
3) (integer) 4069886008361398
4) 1) "116.5142020583152771"
2) "39.90540918662494363"
2) 1) "juejin"
2) "10.5501"
3) (integer) 4069887154388167
4) 1) "116.48104995489120483"
2) "39.99679348858259686"
3) 1) "meituan"
2) "11.5748"
3) (integer) 4069887179083478
4) 1) "116.48903220891952515"
2) "40.00766997707732031"
```

除了 `georadiusbymember` 指令根据元素查询附近的元素，Redis 还提供了根据坐标值来查询附近的元素，这个指令更加有用，它可以根据用户的定位来计算「附近的车」，「附近的餐馆」等。它的参数和 `georadiusbymember` 基本一致，除了将目标元素改成经纬度坐标值：

```
127.0.0.1:6379> georadius company 116.514202 39.905409 20 km withdist count 3
asc
1) 1) "ireader"
2) "0.0000"
2) 1) "juejin"
2) "10.5501"
3) 1) "meituan"
2) "11.5748"
```

8.3.6 注意事项

在一个地图应用中，车的数据、餐馆的数据、人的数据可能会有百万千万条，如果使用 Redis 的 Geo 数据结构，它们将 **全部放在一个 zset 集合中**。在 Redis 的集群环境中，集合可能会从一个节点迁移到另一个节点，如果单个 key 的数据过大，会对集群的迁移工作造成较大的影响，在集群环境中单个 key 对应的数据量不宜超过 1M，否则会导致集群迁移出现卡顿现象，影响线上服务的正常运行。

所以，这里建议 Geo 的数据使用 **单独的 Redis 实例部署**，不使用集群环境。

如果数据量过亿甚至更大，就需要对 Geo 数据进行拆分，按国家拆分、按省拆分，按市拆分，在人口特大城市甚至可以按区拆分。这样就可以显著降低单个 zset 集合的大小。

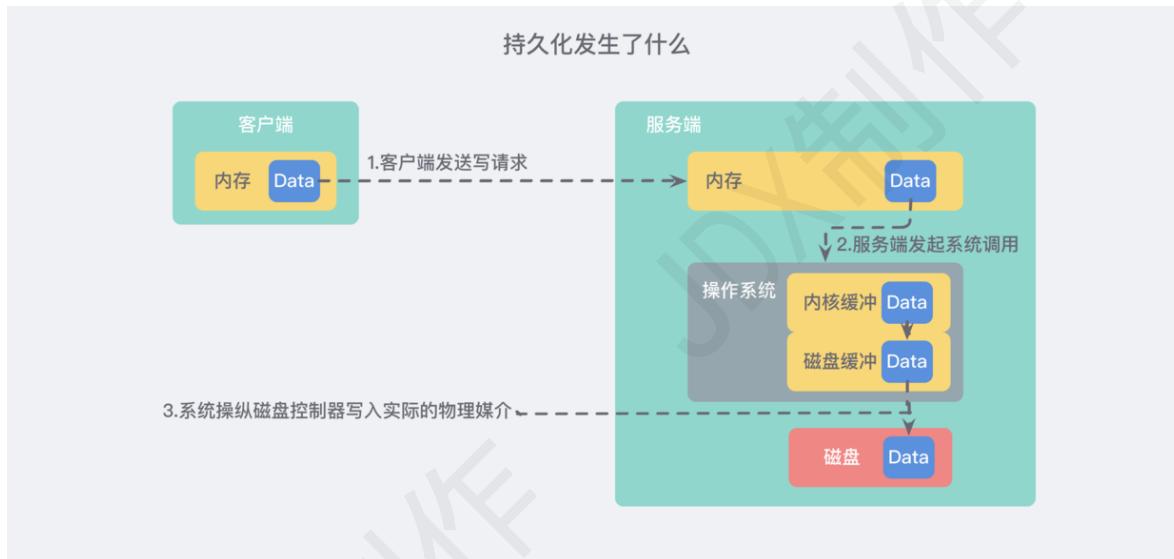
9. 持久化

9.1 持久化简介

Redis 的数据 **全部存储在内存中**，如果 **突然宕机**，数据就会全部丢失，因此必须有一套机制来保证 Redis 的数据不会因为故障而丢失，这种机制就是 Redis 的 **持久化机制**，它会将内存中的数据库状态 **保存到磁盘中**。

9.1.1 持久化发生了什么 | 从内存到磁盘

我们来稍微考虑一下 Redis 作为一个 "内存数据库" 要做的关于持久化的事情。通常来说，从客户端发起请求开始，到服务器真实地写入磁盘，需要发生如下几件事情：



详细版的文字描述大概就是下面这样：

1. 客户端向数据库 **发送写命令** (数据在客户端的内存中)
2. 数据库 **接收** 到客户端的 **写请求** (数据在服务器的内存中)
3. 数据库 **调用系统 API** 将数据写入磁盘 (数据在内核缓冲区中)
4. 操作系统将 **写缓冲区** 传输到 **磁盘控制器** (数据在磁盘缓存中)
5. 操作系统的磁盘控制器将数据 **写入实际的物理媒介** 中 (数据在磁盘中)

注意：上面的过程其实是 **极度精简** 的，在实际的操作系统中，**缓存** 和 **缓冲区** 会比这 **多得多**...

9.1.2 如何尽可能保证持久化的安全

如果我们故障仅仅涉及到 **软件层面** (该进程被管理员终止或程序崩溃) 并且没有接触到内核，那么在 上述步骤3 成功返回之后，我们就认为成功了。即使进程崩溃，操作系统仍然会帮助我们把数据正确地写入磁盘。

如果我们考虑 **停电/火灾** 等 **更具灾难性** 的事情，那么只有在完成了第 5 步之后，才是安全的。

所以我们可以总结得出数据安全最重要的阶段是：**步骤三、四、五**，即：

- 数据库软件调用写操作将用户空间的缓冲区转移到内核缓冲区的频率是多少？
- 内核多久从缓冲区取数据刷新到磁盘控制器？
- 磁盘控制器多久把数据写入物理媒介一次？
- **注意：**如果真的发生灾难性的事件，我们可以从上图的过程中看到，任何一步都可能被意外打断丢失，所以只能 **尽可能地保证** 数据的安全，这对于所有数据库来说都是一样的。

我们从 **第三步** 开始。Linux 系统提供了清晰、易用的用于操作文件的 `POSIX file API`，20 多年过去，仍然还有很多人对于这一套 `API` 的设计津津乐道，我想其中一个原因就是因为你光从 `API` 的命名就能够很清晰地知道这一套 API 的用途：

```
int open(const char *path, int oflag, .../*, mode_t mode */);
int close (int filedes); int remove( const char *fname );
ssize_t write(int filedes, const void *buf, size_t nbytes);
ssize_t read(int filedes, void *buf, size_t nbytes);
```

所以，我们有很好的可用的 `API` 来完成 **第三步**，但是对于成功返回之前，我们对系统调用花费的时间没有太多的控制权。

然后我们来说说 **第四步**。我们知道，除了早期对电脑特别了解那帮人（操作系统就这帮人搞的），实际的物理硬件都不是我们能够 **直接操作** 的，都是通过 **操作系统调用** 来达到目的的。为了防止过慢的 I/O 操作拖慢整个系统的运行，操作系统层面做了很多的努力，譬如说 **上述第四步** 提到的 **写缓冲区**，并不是所有的写操作都会被立即写入磁盘，而是要先经过一个缓冲区，默认情况下，Linux 将在 **30 秒** 后实际提交写入。

但是很明显，**30 秒** 并不是 Redis 能够承受的，这意味着，如果发生故障，那么最近 30 秒内写入的所有数据都可能会丢失。幸好 **PROSIX API** 提供了另一个解决方案：**fsync**，该命令会 **强制** 内核将 **缓冲区写入磁盘**，但这是一个非常消耗性能的操作，每次调用都会 **阻塞等待** 直到设备报告 IO 完成，所以一般在生产环境的服务器中，Redis 通常是每隔 1s 左右执行一次 **fsync** 操作。

到目前为止，我们了解到如何控制 **第三步** 和 **第四步**，但是对于 **第五步**，我们 **完全无法控制**。也许一些内核实现将试图告诉驱动实际提交物理介质上的数据，或者控制器可能会为了提高速度而重新排序写操作，不会尽快将数据真正写到磁盘上，而是会等待几个毫秒。这完全是我们无法控制的。

9.2 Redis 中的两种持久化方式

9.2.1 方式一：快照

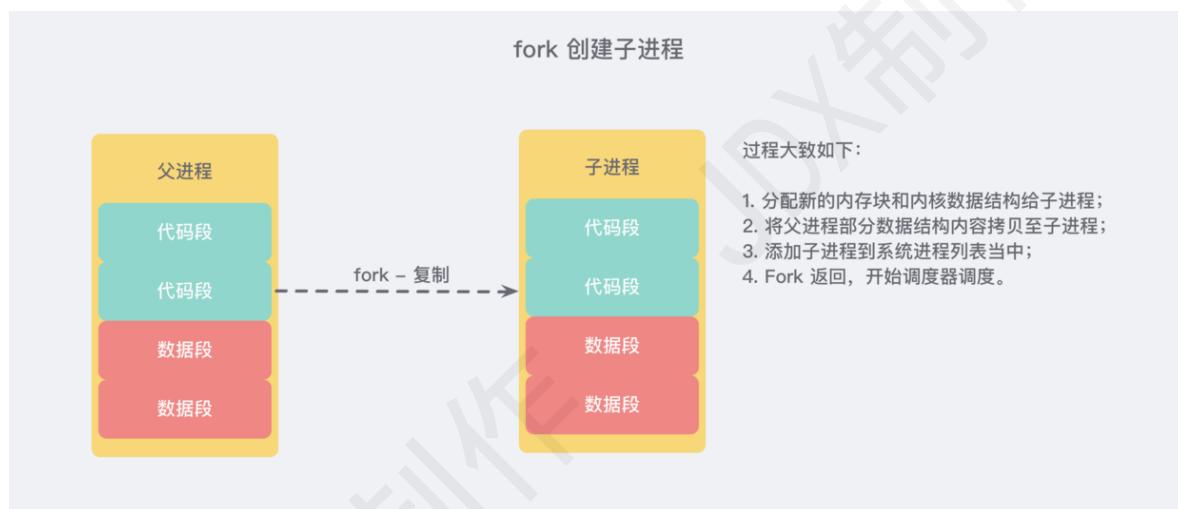
Redis 快照 是最简单的 Redis 持久性模式。当满足特定条件时，它将生成数据集的时间点快照，例如，如果先前的快照是在 2 分钟前创建的，并且现在已经至少有 100 次新写入，则将创建一个新的快照。此条件可以由用户配置 Redis 实例来控制，也可以在运行时修改而无需重新启动服务器。快照作为包含整个数据集的单个 **.rdb** 文件生成。

但我们知道，Redis 是一个 **单线程** 的程序，这意味着，我们不仅要响应用户的请求，还需要进行内存快照。而后者要求 Redis 必须进行 IO 操作，这会严重拖累服务器的性能。

还有一个重要的问题是，我们在 **持久化的同时**，**内存数据结构** 还可能在 **变化**，比如一个大型的 hash 字典正在持久化，结果一个请求过来把它删除了，可是这才刚持久化结束，咋办？

①、使用系统多进程 COW(Copy On Write) 机制 | fork 函数

操作系统多进程 **COW(Copy On Write)** 机制 拯救了我们。Redis 在持久化时会调用 **glibc** 的函数 **fork** 产生一个子进程，简单理解也就是基于当前进程 **复制** 了一个进程，主进程和子进程会共享内存里面的代码块和数据段：



这里多说一点，为什么 **fork** 成功调用后会有两个返回值呢？因为子进程在复制时复制了父进程的堆栈段，所以两个进程都停留在了 **fork** 函数中（都在同一个地方往下继续“同时”执行，等待返回，所以一次在父进程中返回子进程的 pid，另一次在子进程中返回零，系统资源不够时返回负数。（伪代码如下）

```
pid = os.fork()
if pid > 0:
    handle_client_request() # 父进程继续处理客户端请求
if pid == 0:
    handle_snapshot_write() # 子进程处理快照写磁盘
if pid < 0:
    # fork error
```

所以 **快照持久化** 可以完全交给 **子进程** 来处理，**父进程** 则继续 **处理客户端请求**。**子进程** 做数据持久化，它 **不会修改现有的内存数据结构**，它只是对数据结构进行遍历读取，然后序列化写到磁盘中。但是 **父进程** 不一样，它必须持续服务客户端请求，然后对 **内存数据结构进行不间断的修改**。

这个时候就会使用操作系统的 COW 机制来进行 **数据段页面** 的分离。数据段是由很多操作系统的页面组合而成，当父进程对其中一个页面的数据进行修改时，会将被共享的页面复制一份分离出来，然后 **对这个复制的页面进行修改**。这时 **子进程** 相应的页面是 **没有变化的**，还是进程产生时那一瞬间的数据。

子进程因为数据没有变化，它能看到的内存里的数据在进程产生的一瞬间就凝固了，再也不会改变，这也是为什么 Redis 的持久化叫「快照」的原因。接下来子进程就可以非常安心的遍历数据了进行序列化写磁盘了。

9.2.2 方式二：AOF

快照不是很持久。如果运行 Redis 的计算机停止运行，电源线出现故障或者您 `kill -9` 的实例意外发生，则写入 Redis 的最新数据将丢失。尽管这对于某些应用程序可能不是什么大问题，但有些使用案例具有充分的耐用性，在这些情况下，快照并不是可行的选择。

AOF(Append Only File - 仅追加文件) 它的工作方式非常简单：每次执行 **修改内存** 中数据集的写操作时，都会 **记录** 该操作。假设 AOF 日志记录了自 Redis 实例创建以来 **所有的修改性指令序列**，那么就可以通过对一个空的 Redis 实例 **顺序执行所有的指令**，也就是「重放」，来恢复 Redis 当前实例的内存数据结构的状态。

为了展示 AOF 在实际中的工作方式，我们来做一个简单的实验：

```
./redis-server --appendonly yes # 设置一个新实例为 AOF 模式
```

然后我们执行一些写操作：

```
redis 127.0.0.1:6379> set key1 Hello
OK
redis 127.0.0.1:6379> append key1 " World!"
(integer) 12
redis 127.0.0.1:6379> del key1
(integer) 1
redis 127.0.0.1:6379> del non_existing_key
(integer) 0
```

前三个操作实际上修改了数据集，第四个操作没有修改，因为没有指定名称的键。这是 AOF 日志保存的文本：

```
$ cat appendonly.aof
*2
$6
SELECT
$1
```

```
0
*3
$3
set
$4
key1
$5
Hello
*3
$6
append
$4
key1
$7
World!
*2
$3
del
$4
key1
```

如您所见，最后的那一条 `DEL` 指令不见了，因为它没有对数据集进行任何修改。

就这么简单。当 Redis 收到客户端修改指令后，会先进行参数校验、逻辑处理，如果没问题，就 **立即** 将该指令文本 **存储** 到 AOF 日志中，也就是说，**先执行指令再将日志存盘**。这一点不同于 MySQL、LevelDB、HBase 等存储引擎，如果我们先存储日志再做逻辑处理，这样就可以保证即使宕机了，我们仍然可以通过之前保存的日志恢复到之前的数据状态，但是 **Redis 为什么没有这么做呢？**

Emmm... 没找到特别满意的答案，引用一条来自知乎上的回答吧：

- **@缘于专注** - 我甚至觉得没有什么特别的原因。仅仅是因为，由于AOF文件会比较大，为了避免写入无效指令（错误指令），必须先做指令检查？如何检查，只能先执行了。因为语法级别检查并不能保证指令的有效性，比如删除一个不存在的key。而MySQL这种是因为它本身就维护了所有的表的信息，所以可以语法检查后过滤掉大部分无效指令直接记录日志，然后再执行。

①、AOF 重写

Redis 在长期运行的过程中，AOF 的日志会越变越长。如果实例宕机重启，重放整个 AOF 日志会非常耗时，导致长时间 Redis 无法对外提供服务。所以需要对 **AOF 日志 "瘦身"**。

Redis 提供了 `bgrewriteaof` 指令用于对 AOF 日志进行瘦身。其 **原理** 就是 **开辟一个子进程** 对内存进行 **遍历** 转换成一系列 Redis 的操作指令，**序列化到一个新的 AOF 日志文件** 中。序列化完毕后再将操作期间发生的 **增量 AOF 日志** 追加到这个新的 AOF 日志文件中，追加完毕后就立即替代旧的 AOF 日志文件了，瘦身工作就完成了。

②、fsync

AOF 日志是以文件的形式存在的，当程序对 AOF 日志文件进行写操作时，实际上是将内容写到了内核为文件描述符分配的一个内存缓存中，然后内核会异步将脏数据刷回到磁盘的。

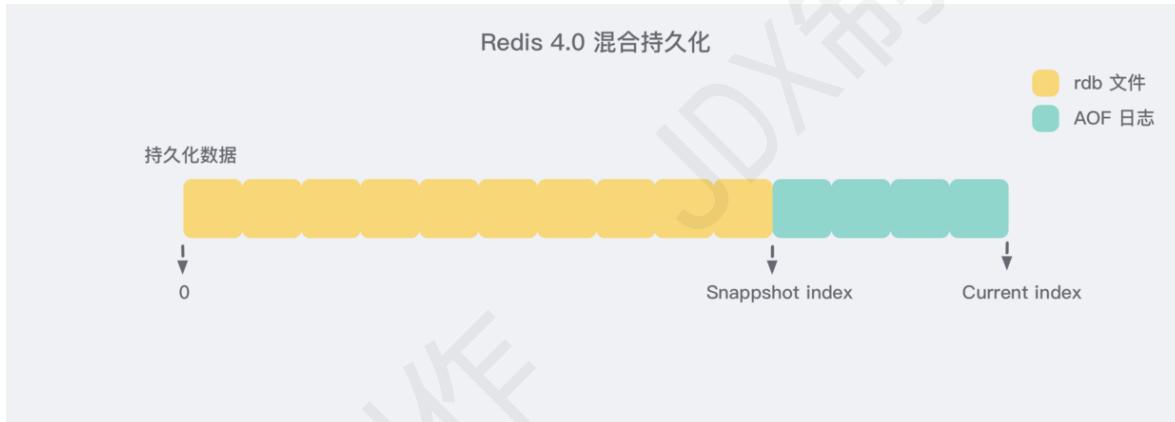
就像我们 上方第四步 描述的那样，我们需要借助 `glibc` 提供的 `fsync(int fd)` 函数来讲指定的文件内容 **强制从内核缓存刷到磁盘**。但 "**强制开车**" 仍然是一个很消耗资源的一个过程，需要 "**节制**"！通常来说，生产环境的服务器，Redis 每隔 1s 左右执行一次 `fsync` 操作就可以了。

Redis 同样也提供了另外两种策略，一个是 **永不 fsync**，来让操作系统来决定合适同步磁盘，很不安全，另一个是 **来一个指令就 fsync 一次**，非常慢。但是在生产环境基本不会使用，了解一下即可。

9.2.3 Redis 4.0 混合持久化

重启 Redis 时，我们很少使用 `rdb` 来恢复内存状态，因为会丢失大量数据。我们通常使用 AOF 日志重放，但是重放 AOF 日志性能相对 `rdb` 来说要慢很多，这样在 Redis 实例很大的情况下，启动需要花费很长的时间。

Redis 4.0 为了解决这个问题，带来了一个新的持久化选项——**混合持久化**。将 `rdb` 文件的内容和增量的 AOF 日志文件存在一起。这里的 AOF 日志不再是全量的日志，而是 **自持久化开始到持久化结束** 的这段时间发生的增量 AOF 日志，通常这部分 AOF 日志很小：



于是在 Redis 重启的时候，可以先加载 `rdb` 的内容，然后再重放增量 AOF 日志就可以完全替代之前的 AOF 全量文件重放，重启效率因此大幅得到提升。

10. 发布订阅与Stream

10.1 Redis 中的发布/订阅功能

发布/订阅系统 是 Web 系统中比较常用的一个功能。简单点说就是 **发布者发布消息，订阅者接受消息**，这有点类似于我们的报纸/杂志社之类的：(借用前边的一张图)



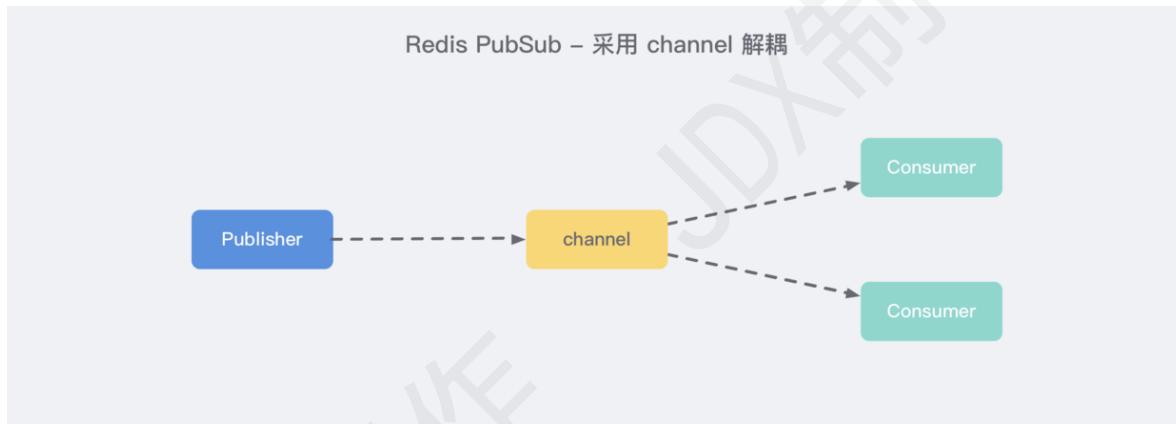
从我们 [前面\(下方相关阅读\)](#) 学习的知识来看，我们虽然可以使用一个 `list` 列表结构结合 `lpush` 和 `rpop` 来实现消息队列的功能，但是似乎很难实现实现 **消息多播** 的功能：



为了支持消息多播，Redis 不能再依赖于那 5 种基础的数据结构了，它单独使用了一个模块来支持消息多播，这个模块就是 **PubSub**，也就是 **PublisherSubscriber**（发布者/ 订阅者模式）。

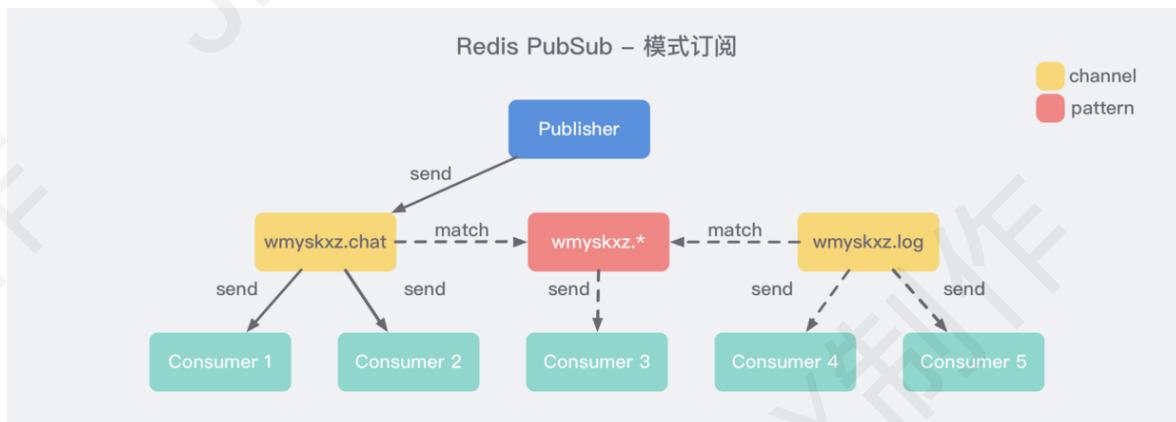
10.1.1 PubSub 简介

我们从 上面的图中可以看到，基于 `list` 结构的消息队列，是一种 `Publisher` 与 `Consumer` 点对点的强关联关系，Redis 为了消除这样的强关联，引入了另一种概念：频道（channel）：



当 `Publisher` 往 `channel` 中发布消息时，关注了指定 `channel` 的 `Consumer` 就能够同时受到消息。但这里的 **问题是**，消费者订阅一个频道是必须 **明确指定频道名称** 的，这意味着，如果我们想要 **订阅多个频道**，那么就必须 **显式地关注多个名称**。

为了简化订阅的繁琐操作，Redis 提供了 **模式订阅** 的功能 `Pattern Subscribe`，这样就可以 **一次性关注多个频道** 了，即使生产者新增了同模式的频道，消费者也可以立即受到消息：



例如上图中，**所有** 位于图片下方的 `Consumer` 都能够受到消息。

`Publisher` 往 `wmyskxz.chat` 这个 `channel` 中发送了一条消息，不仅仅关注了这个频道的 `Consumer 1` 和 `Consumer 2` 能够受到消息，图片中的两个 `channel` 都和模式 `wmyskxz.*` 匹配，所以 Redis 此时会同样发送消息给订阅了 `wmyskxz.*` 这个模式的 `Consumer 3` 和关注了在这个模式下的另一个频道 `wmyskxz.log` 下的 `Consumer 4` 和 `Consumer 5`。

另一方面，如果接收消息的频道是 `wmyskxz.chat`，那么 `Consumer 3` 也会受到消息。

10.1.2 快速体验

在 Redis 中，**PubSub** 模块的使用非常简单，常用的命令也就下面这么几条：

```

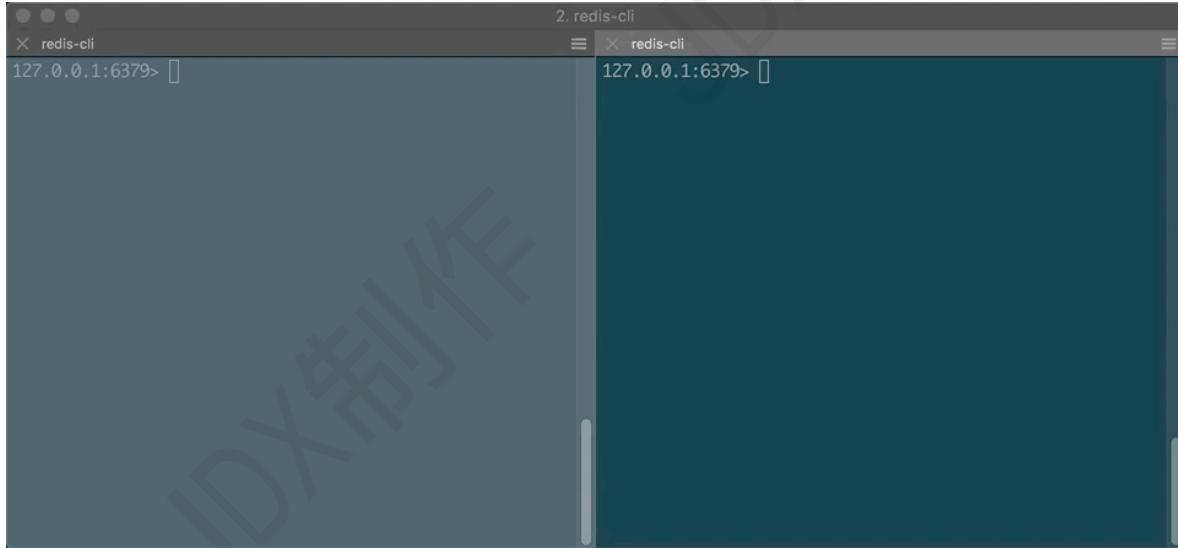
# 订阅频道:
SUBSCRIBE channel [channel ....]    # 订阅给定的一个或多个频道的信息
PUBLISH pattern [pattern ....]    # 订阅一个或多个符合给定模式的频道

# 发布频道:
PUBLISH channel message  # 将消息发送到指定的频道

# 退订频道:
UNSUBSCRIBE [channel [channel ....]]    # 退订指定的频道
PUNSUBSCRIBE [pattern [pattern ....]]    # 退订所有给定模式的频道

```

我们可以在本地快速地来体验一下 PubSub:



具体步骤如下：

1. 开启本地 Redis 服务，新建两个控制台窗口；
2. 在其中一个窗口输入 `SUBSCRIBE wmyskxz.chat` 关注 `wmyskxz.chat` 频道，让这个窗口成为 **消费者**。
3. 在另一个窗口输入 `PUBLISH wmyskxz.chat 'message'` 往这个频道发送消息，这个时候就会看到 **另一个窗口实时地出现了发送的测试消息**。

10.1.3 实现原理

可以看到，我们通过很简单的两条命令，几乎就可以简单使用这样的一个 **发布/订阅系统** 了，但是具体是怎么样实现的呢？

每个 Redis 服务器进程 维持着一个标识服务器状态的 `redis.h/redissServer` 结构，其中就 **保存着有订阅的频道以及订阅模式** 的信息：

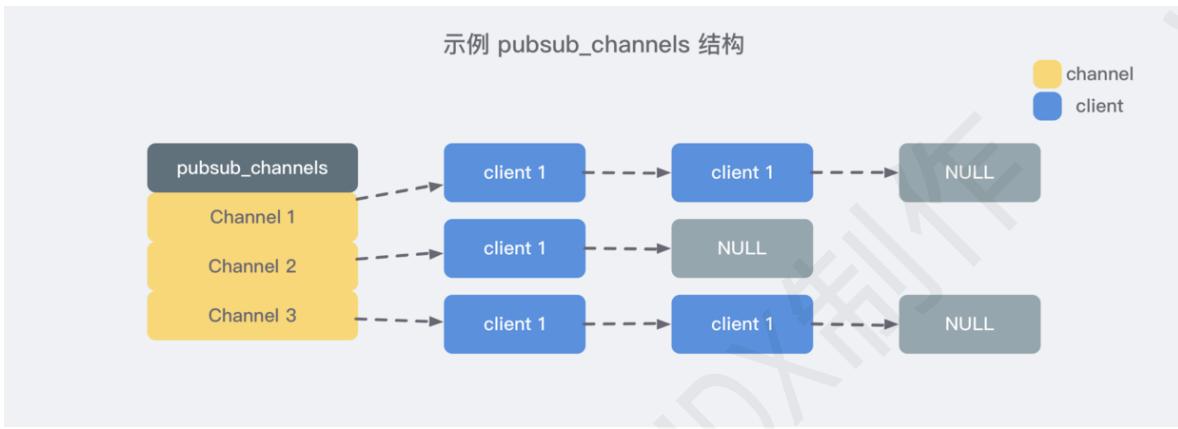
```

struct redissServer {
    // ...
    dict *pubsub_channels; // 订阅频道
    list *pubsub_patterns; // 订阅模式
    // ...
};

```

10.1.4 订阅频道原理

当客户端订阅某一个频道之后，Redis 就会往 `pubsub_channels` 这个字典中新添加一条数据，实际上这个 `dict` 字典维护的是一张链表，比如，下图展示的 `pubsub_channels` 示例中，`client 1`、`client 2` 就订阅了 `channel 1`，而其他频道也分别被其他客户端订阅：



①、SUBSCRIBE 命令

`SUBSCRIBE` 命令的行为可以用下列的伪代码表示：

```
def SUBSCRIBE(client, channels):
    # 遍历所有输入频道
    for channel in channels:
        # 将客户端添加到链表的末尾
        redisServer.pubsub_channels[channel].append(client)
```

通过 `pubsub_channels` 字典，程序只要检查某个频道是否为字典的键，就可以知道该频道是否正在被客户端订阅；只要取出某个键的值，就可以得到所有订阅该频道的客户端的信息。

②、PUBLISH 命令

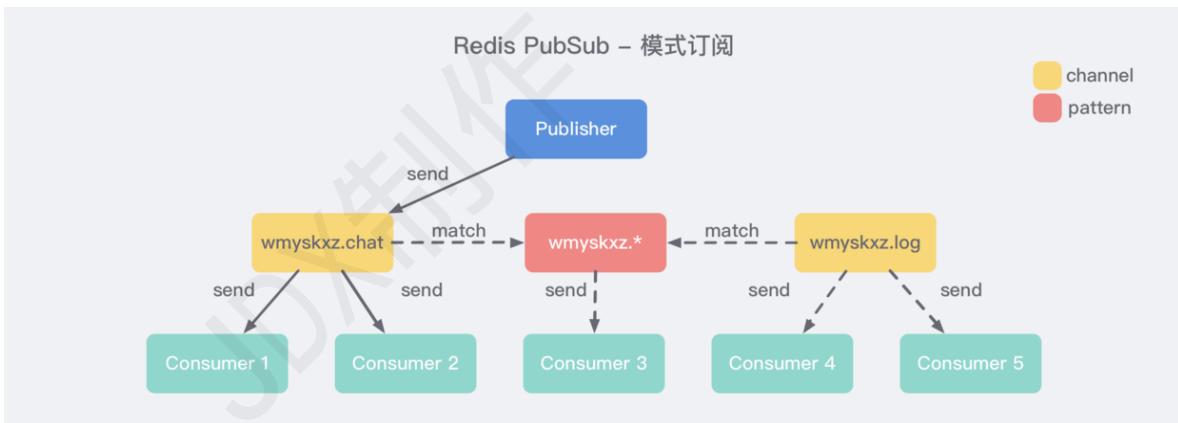
了解 `SUBSCRIBE`，那么 `PUBLISH` 命令的实现也变得十分简单了，只需要通过上述字典定位到具体的客户端，再把消息发送给它们就好了：(伪代码实现如下)

```
def PUBLISH(channel, message):
    # 遍历所有订阅频道 channel 的客户端
    for client in server.pubsub_channels[channel]:
        # 将信息发送给它们
        send_message(client, message)
```

③、UNSUBSCRIBE 命令

使用 `UNSUBSCRIBE` 命令可以退订指定的频道，这个命令执行的是订阅的反操作：它从 `pubsub_channels` 字典的给定频道（键）中，删除关于当前客户端的信息，这样被退订频道的信息就不会再发送给这个客户端。

10.1.5 订阅模式原理

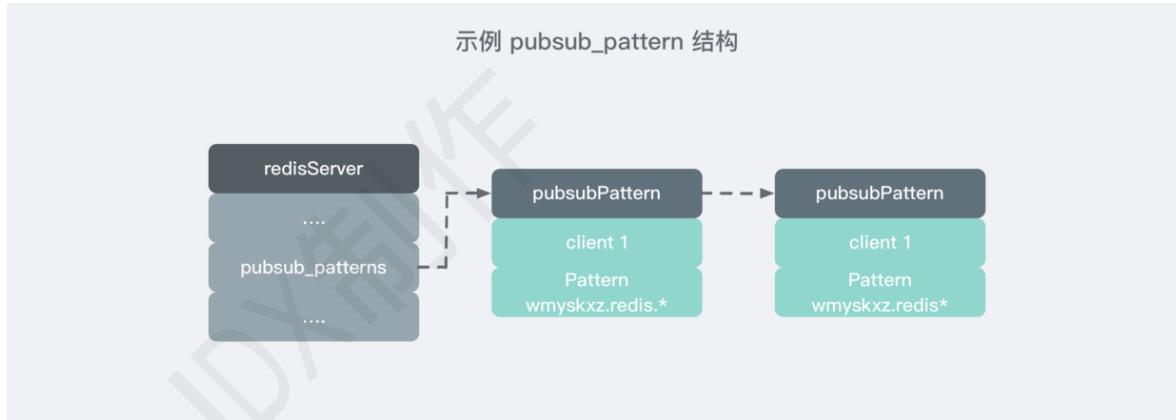


正如我们上面说到了，当发送一条消息到 `wmySkxz.chat` 这个频道时，Redis 不仅仅会发送到当前的频道，还会发送到匹配于当前模式的所有频道，实际上，`pubsub_patterns` 背后还维护了一个 `redis.h/pubsubPattern` 结构：

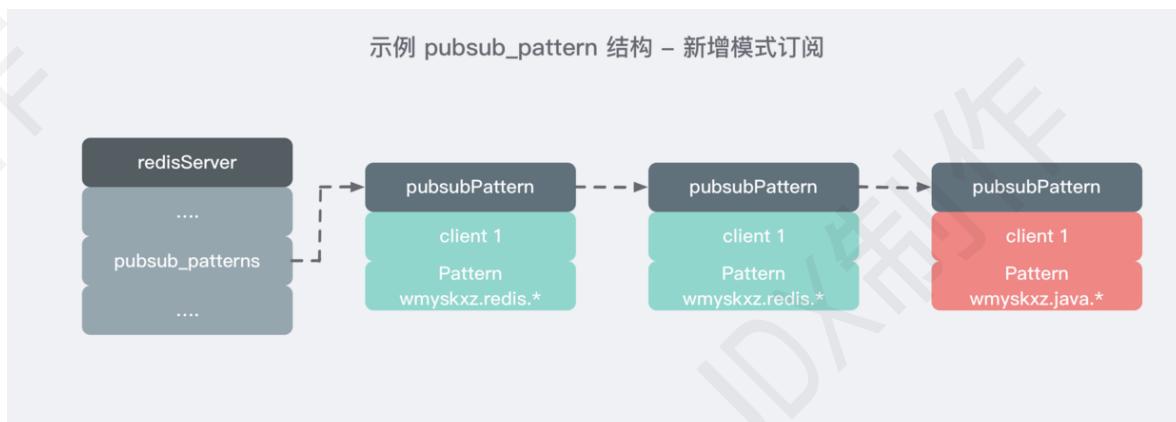
```
typedef struct pubsubPattern {
    redisClient *client; // 订阅模式的客户端
    robj *pattern; // 订阅的模式
} pubsubPattern;
```

每当调用 `PSUBSCRIBE` 命令订阅一个模式时，程序就创建一个包含客户端信息和被订阅模式的 `pubsubPattern` 结构，并将该结构添加到 `redisServer.pubsub_patterns` 链表中。

我们来看一个 `pubsub_patterns` 链表的示例：



这个时候客户端 `client 3` 执行 `PSUBSCRIBE wmySkxz.java.*`，那么 `pubsub_patterns` 链表就会被更新成这样：



通过遍历整个 `pubsub_patterns` 链表，程序可以检查所有正在被订阅的模式，以及订阅这些模式的客户端。

①、PUBLISH 命令

上面给出的伪代码并没有完整描述 `PUBLISH` 命令的行为，因为 `PUBLISH` 除了将 `message` 发送到所有订阅 `channel` 的客户端之外，它还会将 `channel` 和 `pubsub_patterns` 中的 模式 进行对比，如果 `channel` 和某个模式匹配的话，那么也将 `message` 发送到 订阅那个模式的客户端。

完整描述 `PUBLISH` 功能的伪代码定于如下：

```

def PUBLISH(channel, message):
    # 遍历所有订阅频道 channel 的客户端
    for client in server.pubsub_channels[channel]:
        # 将信息发送给它们
        send_message(client, message)
    # 取出所有模式，以及订阅模式的客户端
    for pattern, client in server.pubsub_patterns:
        # 如果 channel 和模式匹配
        if match(channel, pattern):
            # 那么也将信息发给订阅这个模式的客户端
            send_message(client, message)

```

②、PUNSUBSCRIBE 命令

使用 `PUNSUBSCRIBE` 命令可以退订指定的模式，这个命令执行的是订阅模式的反操作：序会删除 `redisServer.pubsub_patterns` 链表中，所有和被退订模式相关联的 `pubsubPattern` 结构，这样客户端就不会再收到和模式相匹配的频道发来的信息。

10.1.6 PubSub 的缺点

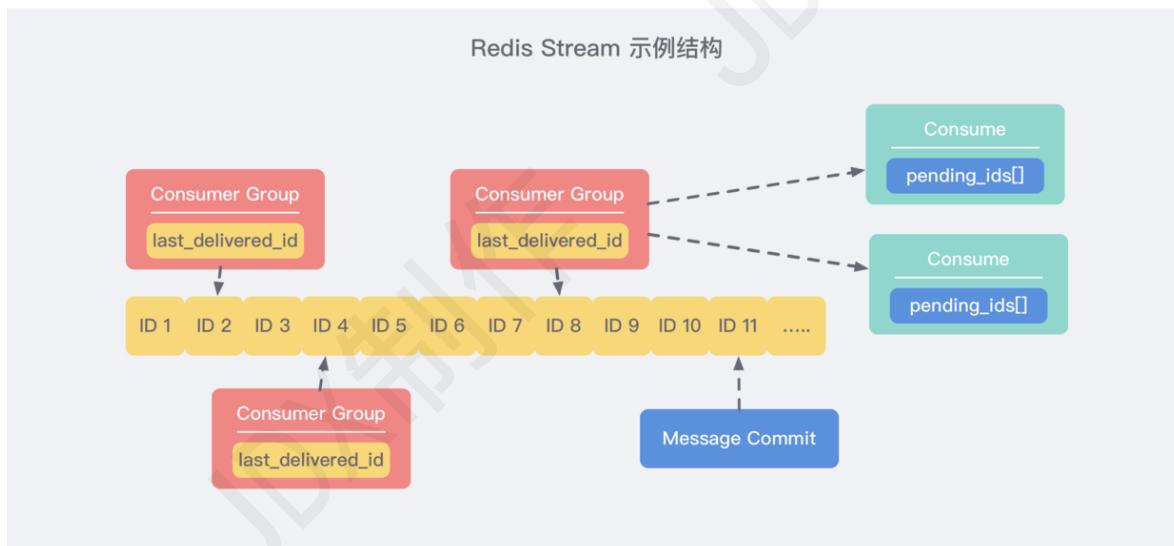
尽管 Redis 实现了 PubSub 模式来达到了 多播消息队列 的目的，但在实际的消息队列的领域，几乎 找不到特别合适的场景，因为它的缺点十分明显：

- **没有 Ack 机制，也不保证数据的连续：** PubSub 的生产者传递过来一个消息，Redis 会直接找到相应的消费者传递过去。如果没有一个消费者，那么消息会被直接丢弃。如果开始有三个消费者，其中一个突然挂掉了，过了一会儿等它再重连时，那么重连期间的消息对于这个消费者来说就彻底丢失了。
- **不持久化消息：** 如果 Redis 停机重启，PubSub 的消息是不会持久化的，毕竟 Redis 容机就相当于一个消费者都没有，所有的消息都会被直接丢弃。

基于上述缺点，Redis 的作者甚至单独开启了一个 Disque 的项目来专门用来做多播消息队列，不过该项目目前好像都没有成熟。不过后来在 2018 年 6 月，Redis 5.0 新增了 `Stream` 数据结构，这个功能给 Redis 带来了 **持久化消息队列**，从此 PubSub 作为消息队列的功能可以说是就消失了..

10.2 更为强大的 Stream | 持久化的发布/订阅系统

Redis Stream 从概念上来说，就像是一个 **仅追加内容** 的 **消息链表**，把所有加入的消息都一个一个串起来，每个消息都有一个唯一的 ID 和内容，这很简单，让它复杂的是从 Kafka 借鉴的另一种概念：**消费者组(Consumer Group)** (思路一致，实现不同)：



上图就展示了一个典型的 **Stream** 结构。每个 Stream 都有唯一的名称，它就是 Redis 的 **key**，在我们首次使用 `xadd` 指令追加消息时自动创建。我们对图中的一些概念做一下解释：

- **Consumer Group**: 消费者组，可以简单看成记录流状态的一种数据结构。消费者既可以选择使用 `XREAD` 命令进行 **独立消费**，也可以多个消费者同时加入一个消费者组进行 **组内消费**。同一个消费者组内的消费者共享所有的 Stream 信息，**同一条消息只会有一个消费者消费到**，这样就可以应用在分布式的应用场景中来保证消息的唯一性。
- **last_delivered_id**: 用来表示消费者组消费在 Stream 上 **消费位置** 的游标信息。每个消费者组都有一个 Stream 内 **唯一的名称**，消费者组不会自动创建，需要使用 `XGROUP CREATE` 指令来显式创建，并且需要指定从哪一个消息 ID 开始消费，用来初始化 `last_delivered_id` 这个变量。
- **pending_ids**: 每个消费者内部都有的一个状态变量，用来表示 **已经** 被客户端 **获取**，但是 **还没有 ack** 的消息。记录的目的是为了 **保证客户端至少消费了消息一次**，而不会在网络传输的中途丢失而没有对消息进行处理。如果客户端没有 ack，那么这个变量里面的消息 ID 就会越来越多，一旦某个消息被 ack，它就会对应开始减少。这个变量也被 Redis 官方称为 **PEL (Pending Entries List)**。

10.2.1 消息 ID 和消息内容

①、消息 ID

消息 ID 如果是由 `XADD` 命令返回自动创建的话，那么它的格式会像这样：`timestampInMillis-sequence` (毫秒时间戳-序列号)，例如 `1527846880585-5`，它表示当前的消息是在毫秒时间戳 `1527846880585` 时产生的，并且是该毫秒内产生的第 5 条消息。

这些 ID 的格式看起来有一些奇怪，**为什么要使用时间来当做 ID 的一部分呢？** 一方面，我们要 **满足 ID 自增** 的属性，另一方面，也是为了 **支持范围查找** 的功能。由于 ID 和生成消息的时间有关，这样就使得在根据时间范围内查找时基本上是没有额外损耗的。

当然消息 ID 也可以由客户端自定义，但是形式必须是 "**整数-整数**"，而且后面加入的消息的 ID 必须要大于前面的消息 ID。

②、消息内容

消息内容就是普通的键值对，形如 hash 结构的键值对。

10.2.2 增删改查示例

增删改查命令很简单，详情如下：

1. `xadd`: 追加消息
2. `xdel`: 删除消息，这里的删除仅仅是设置了标志位，不影响消息总长度
3. `xrange`: 获取消息列表，会自动过滤已经删除的消息
4. `xlen`: 消息长度
5. `del`: 删除Stream

使用示例：

```
# *号表示服务器自动生成ID，后面顺序跟着一堆key/value
127.0.0.1:6379> xadd codehole * name laoqian age 30 # 名字叫laoqian，年龄30岁
1527849609889-0 # 生成的消息ID
127.0.0.1:6379> xadd codehole * name xiaoyu age 29
1527849629172-0
127.0.0.1:6379> xadd codehole * name xiaoqian age 1
1527849637634-0
127.0.0.1:6379> xlen codehole
(integer) 3
```

```
127.0.0.1:6379> xrange codehole - + # -表示最小值, +表示最大值
1) 1) 1527849609889-0
   2) 1) "name"
      2) "laoqian"
      3) "age"
      4) "30"
2) 1) 1527849629172-0
   2) 1) "name"
      2) "xiaoyu"
      3) "age"
      4) "29"
3) 1) 1527849637634-0
   2) 1) "name"
      2) "xiaoqian"
      3) "age"
      4) "1"
127.0.0.1:6379> xrange codehole 1527849629172-0 + # 指定最小消息ID的列表
1) 1) 1527849629172-0
   2) 1) "name"
      2) "xiaoyu"
      3) "age"
      4) "29"
2) 1) 1527849637634-0
   2) 1) "name"
      2) "xiaoqian"
      3) "age"
      4) "1"
127.0.0.1:6379> xrange codehole - 1527849629172-0 # 指定最大消息ID的列表
1) 1) 1527849609889-0
   2) 1) "name"
      2) "laoqian"
      3) "age"
      4) "30"
2) 1) 1527849629172-0
   2) 1) "name"
      2) "xiaoyu"
      3) "age"
      4) "29"
127.0.0.1:6379> xdel codehole 1527849609889-0
(integer) 1
127.0.0.1:6379> xlen codehole # 长度不受影响
(integer) 3
127.0.0.1:6379> xrange codehole - + # 被删除的消息没了
1) 1) 1527849629172-0
   2) 1) "name"
      2) "xiaoyu"
      3) "age"
      4) "29"
2) 1) 1527849637634-0
   2) 1) "name"
      2) "xiaoqian"
      3) "age"
      4) "1"
127.0.0.1:6379> del codehole # 删除整个stream
(integer) 1
```

10.2.3 独立消费示例

我们可以在不定义消费组的情况下进行 Stream 消息的 **独立消费**，当 Stream 没有新消息时，甚至可以阻塞等待。Redis 设计了一个单独的消费指令 `xread`，可以将 Stream 当成普通的消息队列(list)来使用。使用 `xread` 时，我们可以完全忽略 **消费组(Consumer Group)** 的存在，就好比 Stream 就是一个普通的列表(list)：

```
# 从Stream头部读取两条消息
127.0.0.1:6379> xread count 2 streams codehole 0-0
1) 1) "codehole"
   2) 1) 1) 1527851486781-0
      2) 1) "name"
         2) "laoqian"
         3) "age"
         4) "30"
   2) 1) 1527851493405-0
      2) 1) "name"
         2) "yurui"
         3) "age"
         4) "29"
# 从Stream尾部读取一条消息，毫无疑问，这里不会返回任何消息
127.0.0.1:6379> xread count 1 streams codehole $
(nil)
# 从尾部阻塞等待新消息到来，下面的指令会堵住，直到新消息到来
127.0.0.1:6379> xread block 0 count 1 streams codehole $
# 我们从新打开一个窗口，在这个窗口往Stream里塞消息
127.0.0.1:6379> xadd codehole * name youming age 60
1527852774092-0
# 再切换到前面的窗口，我们可以看到阻塞解除了，返回了新的消息内容
# 而且还显示了一个等待时间，这里我们等待了93s
127.0.0.1:6379> xread block 0 count 1 streams codehole $
1) 1) "codehole"
   2) 1) 1) 1527852774092-0
      2) 1) "name"
         2) "youming"
         3) "age"
         4) "60"
(93.11s)
```

客户端如果想要使用 `xread` 进行 **顺序消费**，一定要 **记住当前消费** 到哪里了，也就是返回的消息 ID。下次继续调用 `xread` 时，将上次返回的最后一个消息 ID 作为参数传递进去，就可以继续消费后续的消息。

`block 0` 表示永远阻塞，直到消息到来，`block 1000` 表示阻塞 `1s`，如果 `1s` 内没有任何消息到来，就返回 `nil`：

```
127.0.0.1:6379> xread block 1000 count 1 streams codehole $
(nil)
(1.07s)
```

10.2.4 创建消费者示例

Stream 通过 `xgroup create` 指令创建消费组(Consumer Group)，需要传递起始消息 ID 参数用来初始化 `last_delivered_id` 变量：

```
127.0.0.1:6379> xgroup create codehole cg1 0-0 # 表示从头开始消费
OK
```

```

# $表示从尾部开始消费，只接受新消息，当前Stream消息会全部忽略
127.0.0.1:6379> xgroup create codehole cg2 $
OK
127.0.0.1:6379> xinfo codehole # 获取Stream信息
1) length
2) (integer) 3 # 共3个消息
3) radix-tree-keys
4) (integer) 1
5) radix-tree-nodes
6) (integer) 2
7) groups
8) (integer) 2 # 两个消费组
9) first-entry # 第一个消息
10) 1) 1527851486781-0
     2) 1) "name"
        2) "laoqian"
        3) "age"
        4) "30"
11) last-entry # 最后一个消息
12) 1) 1527851498956-0
     2) 1) "name"
        2) "xiaoqian"
        3) "age"
        4) "1"
127.0.0.1:6379> xinfo groups codehole # 获取Stream的消费组信息
1) 1) name
2) "cg1"
3) consumers
4) (integer) 0 # 该消费组还没有消费者
5) pending
6) (integer) 0 # 该消费组没有正在处理的消息
2) 1) name
2) "cg2"
3) consumers # 该消费组还没有消费者
4) (integer) 0
5) pending
6) (integer) 0 # 该消费组没有正在处理的消息

```

10.2.5 组内消费示例

Stream 提供了 `xreadgroup` 指令可以进行消费组的组内消费，需要提供 **消费组名称、消费者名称和起始消息 ID**。它同 `xread` 一样，也可以阻塞等待新消息。读到新消息后，对应的消息 ID 就会进入消费者的 **PEL (正在处理的消息)** 结构里，客户端处理完毕后使用 `xack` 指令 **通知服务器**，本条消息已经处理完毕，该消息 ID 就会从 **PEL** 中移除，下面是示例：

```

# >号表示从当前消费组的last_delivered_id后面开始读
# 每当消费者读取一条消息，last_delivered_id变量就会前进
127.0.0.1:6379> xreadgroup GROUP cg1 c1 count 1 streams codehole >
1) 1) "codehole"
2) 1) 1) 1527851486781-0
     2) 1) "name"
        2) "laoqian"
        3) "age"
        4) "30"
127.0.0.1:6379> xreadgroup GROUP cg1 c1 count 1 streams codehole >
1) 1) "codehole"
2) 1) 1) 1527851493405-0

```

```
2) 1) "name"
    2) "yurui"
    3) "age"
    4) "29"
127.0.0.1:6379> xreadgroup GROUP cg1 c1 count 2 streams codehole >
1) 1) "codehole"
    2) 1) 1) 1527851498956-0
        2) 1) "name"
        2) "xiaoqian"
        3) "age"
        4) "1"
    2) 1) 1527852774092-0
        2) 1) "name"
        2) "youming"
        3) "age"
        4) "60"
# 再继续读取，就没有新消息了
127.0.0.1:6379> xreadgroup GROUP cg1 c1 count 1 streams codehole >
(nil)
# 那就阻塞等待吧
127.0.0.1:6379> xreadgroup GROUP cg1 c1 block 0 count 1 streams codehole >
# 开启另一个窗口，往里塞消息
127.0.0.1:6379> xadd codehole * name lanying age 61
1527854062442-0
# 回到前一个窗口，发现阻塞解除，收到新消息了
127.0.0.1:6379> xreadgroup GROUP cg1 c1 block 0 count 1 streams codehole >
1) 1) "codehole"
    2) 1) 1) 1527854062442-0
        2) 1) "name"
        2) "lanying"
        3) "age"
        4) "61"
(36.54s)
127.0.0.1:6379> xinfo groups codehole # 观察消费组信息
1) 1) name
    2) "cg1"
    3) consumers
    4) (integer) 1 # 一个消费者
    5) pending
    6) (integer) 5 # 共5条正在处理的信息还有没有ack
2) 1) name
    2) "cg2"
    3) consumers
    4) (integer) 0 # 消费组cg2没有任何变化，因为前面我们一直在操纵cg1
    5) pending
    6) (integer) 0
# 如果同一个消费组有多个消费者，我们可以通过xinfo consumers指令观察每个消费者的状态
127.0.0.1:6379> xinfo consumers codehole cg1 # 目前还有1个消费者
1) 1) name
    2) "c1"
    3) pending
    4) (integer) 5 # 共5条待处理消息
    5) idle
    6) (integer) 418715 # 空闲了多长时间ms没有读取消息了
# 接下来我们ack一条消息
127.0.0.1:6379> xack codehole cg1 1527851486781-0
(integer) 1
127.0.0.1:6379> xinfo consumers codehole cg1
```

```
1) 1) name
2) "c1"
3) pending
4) (integer) 4 # 变成了5条
5) idle
6) (integer) 668504
# 下面ack所有消息
127.0.0.1:6379> xack codehole cg1 1527851493405-0 1527851498956-0 1527852774092-0 1527854062442-0
(integer) 4
127.0.0.1:6379> xinfo consumers codehole cg1
1) 1) name
2) "c1"
3) pending
4) (integer) 0 # pel空了
5) idle
6) (integer) 745505
```

10.2.6 QA 1: Stream 消息太多怎么办? | Stream 的上限

很容易想到，要是消息积累太多，Stream 的链表岂不是很长，内容会不会爆掉就是个问题了。`xdel` 指令又不会删除消息，它只是给消息做了个标志位。

Redis 自然考虑到了这一点，所以它提供了一个定长 Stream 功能。在 `xadd` 的指令提供一个定长长度 `maxlen`，就可以将老的消息干掉，确保最多不超过指定长度，使用起来也很简单：

```
> XADD mystream MAXLEN 2 * value 1
1526654998691-0
> XADD mystream MAXLEN 2 * value 2
1526654999635-0
> XADD mystream MAXLEN 2 * value 3
1526655000369-0
> XLEN mystream
(integer) 2
> XRANGE mystream - +
1) 1) 1526654999635-0
   2) 1) "value"
      2) "2"
2) 1) 1526655000369-0
   2) 1) "value"
      2) "3"
```

如果使用 `MAXLEN` 选项，当 Stream 的达到指定长度后，老的消息会自动被淘汰掉，因此 Stream 的大小是恒定的。目前还没有选项让 Stream 只保留给定数量的条目，因为为了保证一致性地运行，这样的命令必须在很长一段时间内阻塞以淘汰消息。(例如在添加数据的高峰期间，你不得不长暂停来淘汰旧消息和添加新的消息)

另外使用 `MAXLEN` 选项的花销是很大的，Stream 为了节省内存空间，采用了一种特殊的结构表示，而这种结构的调整是需要额外的花销的。所以我们可以使用一种带有 `~` 的特殊命令：

```
XADD mystream MAXLEN ~ 1000 * ... entry fields here ...
```

它会基于当前的结构合理地对节点执行裁剪，来保证至少会有 `1000` 条数据，可能是 `1010` 也可能是 `1030`。

10.2.7 QA 2: PEL 是如何避免消息丢失的?

在客户端消费者读取 Stream 消息时，Redis 服务器将消息回复给客户端的过程中，客户端突然断开了连接，消息就丢失了。但是 PEL 里已经保存了发出去的消息 ID，待客户端重新连上之后，可以再次收到 PEL 中的消息 ID 列表。不过此时 `xreadgroup` 的起始消息 ID 不能为参数 `>`，而必须是任意有效的消息 ID，一般将参数设为 `0-0`，表示读取所有的 PEL 消息以及自 `last_delivered_id` 之后的新消息。

10.2.8 Redis Stream Vs Kafka

Redis 基于内存存储，这意味着它会比基于磁盘的 Kafka 快上一些，也意味着使用 Redis 我们 **不能长时间存储大量数据**。不过如果您想以 **最小延迟** 实时处理消息的话，您可以考虑 Redis，但是如果 **消息很大并且应该重用数据** 的话，则应该首先考虑使用 Kafka。

另外从某些角度来说，`Redis Stream` 也更适用于小型、廉价的应用程序，因为 `Kafka` 相对来说更难配置一些。

11. [集群]入门实践教程

11.1 Redis 集群概述

11.1.1 Redis 主从复制

到目前为止，我们所学习的 Redis 都是 **单机版** 的，这也就意味着一旦我们所依赖的 Redis 服务宕机了，我们的主流程也会受到一定的影响，这当然是我们不能够接受的。

所以一开始我们的想法是：搞一台备用机。这样我们就可以在一台服务器出现问题的时候切换动态地到另一台去：



幸运的是，两个节点数据的同步我们可以使用 Redis 的 **主从同步** 功能帮助到我们，这样一来，有个备份，心里就踏实多了。

11.1.2 Redis 哨兵

后来因为某种神秘力量，Redis 老会在莫名其妙的时间点出问题（比如半夜 2 点），我总不能 24 小时时刻守在电脑旁边切换节点吧，于是另一个想法又开始了：给所有的节点找一个“管家”，自动帮我监听照顾节点的状态并切换：



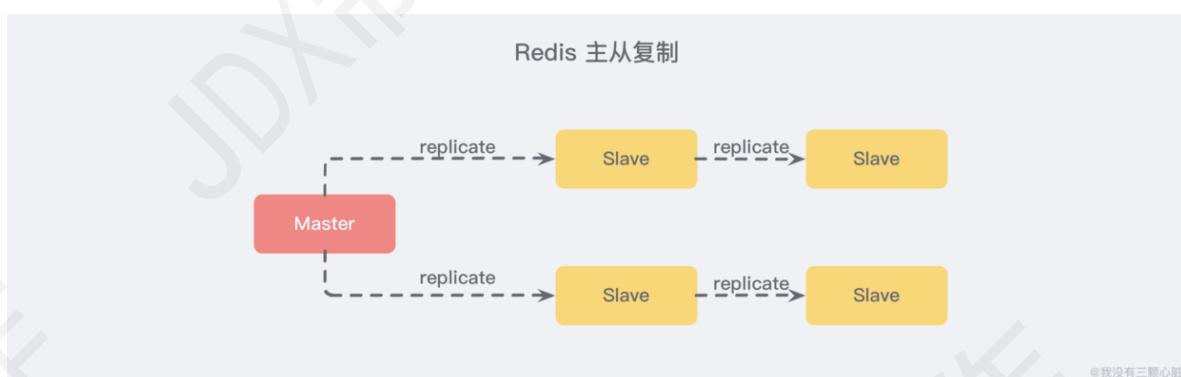
这大概就是 Redis 哨兵 (Sentinel) 的简单理解啦。什么？管家宕机了怎么办？相较于有大量请求的 Redis 服务来说，管家宕机的概率就要小得多啦.. 如果真的宕机了，我们也可以直接切换成当前可用的节点保证可用..

11.1.3 Redis 集群化

好了，通过上面的一些解决方案我们对 Redis 的 **稳定性** 稍微有了一些底气了，但单台节点的计算能力始终有限，所谓人多力量大，如果我们把 **多个节点组合成一个可用的工作节点**，那就大大增加了 Redis 的 **高可用、可扩展、分布式、容错** 等特性：



11.2 主从复制



主从复制，是指将一台 Redis 服务器的数据，复制到其他的 Redis 服务器。前者称为 **主节点 (master)**，后者称为 **从节点(slave)**。且数据的复制是 **单向** 的，只能由主节点到从节点。Redis 主从复制支持 **主从同步** 和 **从从同步** 两种，后者是 Redis 后续版本新增的功能，以减轻主节点的同步负担。

11.2.1 主从复制主要的作用

- **数据冗余**：主从复制实现了数据的热备份，是持久化之外的一种数据冗余方式。
- **故障恢复**：当主节点出现问题时，可以由从节点提供服务，实现快速的故障恢复 (实际上是一种服务的冗余)。
- **负载均衡**：在主从复制的基础上，配合读写分离，可以由主节点提供写服务，由从节点提供读服务 (即写 Redis 数据时应用连接主节点，读 Redis 数据时应用连接从节点)，分担服务器负载。尤其是在写少读多的场景下，通过多个从节点分担读负载，可以大大提高 Redis 服务器的并发量。
- **高可用基石**：除了上述作用以外，主从复制还是哨兵和集群能够实施的 **基础**，因此说主从复制是 Redis 高可用的基础。

11.2.2 快速体验

在 Redis 中，用户可以通过执行 `SLAVEOF` 命令或者设置 `slaveof` 选项，让一个服务器去复制另一个服务器，以下三种方式是 **完全等效** 的：

- **配置文件**：在从服务器的配置文件中加入：`slaveof <masterip> <masterport>`
- **启动命令**：`redis-server` 启动命令后加入 `--slaveof <masterip> <masterport>`

- **客户端命令**: Redis 服务器启动后, 直接通过客户端执行命令: `slaveof <masterip> <masterport>`, 让该 Redis 实例成为从节点。

需要注意的是: **主从复制的开启, 完全是在从节点发起的, 不需要我们在主节点做任何事情。**

①、第一步: 本地启动两个节点

在正确安装好 Redis 之后, 我们可以使用 `redis-server --port <port>` 的方式指定创建两个不同端口的 Redis 实例, 例如, 下方我分别创建了一个 6379 和 6380 的两个 Redis 实例:

```
# 创建一个端口为 6379 的 Redis 实例  
redis-server --port 6379  
# 创建一个端口为 6380 的 Redis 实例  
redis-server --port 6380
```

此时两个 Redis 节点启动后, 都默认为 **主节点**。

②、第二步: 建立复制

我们在 6380 端口的节点中执行 `slaveof` 命令, 使之变为从节点:

```
# 在 6380 端口的 Redis 实例中使用控制台  
redis-cli -p 6380  
# 成为本地 6379 端口实例的从节点  
127.0.0.1:6380> SLAVEOF 127.0.0.1 6379  
OK
```

③、第三步: 观察效果

下面我们来验证一下, 主节点的数据是否会复制到从节点之中:

- 先在 **从节点** 中查询一个 **不存在** 的 key:

```
127.0.0.1:6380> GET mykey  
(nil)
```

- 再在 **主节点** 中添加这个 key:

```
127.0.0.1:6379> SET mykey myvalue  
OK
```

- 此时再从 **从节点** 中查询, 会发现已经从 **主节点** 同步到 **从节点**:

```
127.0.0.1:6380> GET mykey  
"myvalue"
```

④、第四步: 断开复制

通过 `slaveof <masterip> <masterport>` 命令建立主从复制关系以后, 可以通过 `slaveof no one` 断开。需要注意的是, 从节点断开复制后, **不会删除已有的数据**, 只是不再接受主节点新的数据变化。

从节点执行 `slaveof no one` 之后, 从节点和主节点分别打印日志如下: 、

```

# 从节点打印日志
61496:M 17 Mar 2020 08:10:22.749 # Connection with master lost.
61496:M 17 Mar 2020 08:10:22.749 * Caching the disconnected master state.
61496:M 17 Mar 2020 08:10:22.749 * Discarding previously cached master state.
61496:M 17 Mar 2020 08:10:22.749 * MASTER MODE enabled (user request from 'id=4
addr=127.0.0.1:55096 fd=8 name= age=1664 idle=0 flags=N db=0 sub=0 psub=0
multi=-1 qbuf=34 qbuf-free=32734 obl=0 oll=0 omem=0 events=r cmd=slaveof')

# 主节点打印日志
61467:M 17 Mar 2020 08:10:22.749 # Connection with replica 127.0.0.1:6380 lost.

```

11.2.3 实现原理简析



为了节省篇幅，我把主要的步骤都 **浓缩** 在了上图中，其实也可以 **简化成三个阶段：准备阶段-数据同步阶段-命令传播阶段**。下面我们来进行一些必要的说明。

①、身份验证 | 主从复制安全问题

在上面的 **快速体验** 过程中，你会发现 `slaveof` 这个命令居然不需要验证？这意味着只要知道了 ip 和端口就可以随意拷贝服务器上的数据了？

那当然不能够了，我们可以通过在 **主节点** 配置 `requirepass` 来设置密码，这样就必须在 **从节点** 中对应配置好 `masterauth` 参数（与主节点 `requirepass` 保持一致）才能够进行正常复制了。

②、SYNC 命令是一个非常耗费资源的操作

每次执行 `SYNC` 命令，主从服务器需要执行如下动作：

- 1. 主服务器** 需要执行 `BGSAVE` 命令来生成 RDB 文件，这个生成操作会 **消耗** 主服务器大量的 **CPU、内存和磁盘 I/O 的资源**；
- 2. 主服务器** 需要将自己生成的 RDB 文件 发送给从服务器，这个发送操作会 **消耗** 主服务器 **大量的网络资源**（带宽和流量），并对主服务器响应命令请求的时间产生影响；
- 3. 接收到 RDB 文件的 从服务器** 需要载入主服务器发来的 RDB 文件，并且在载入期间，从服务器 **会因为阻塞而没办法处理命令请求**；

特别是当出现 **断线重复制** 的情况是时，为了让从服务器补足断线时确实的那一小部分数据，却要执行一次如此耗资源的 `SYNC` 命令，显然是不合理的。

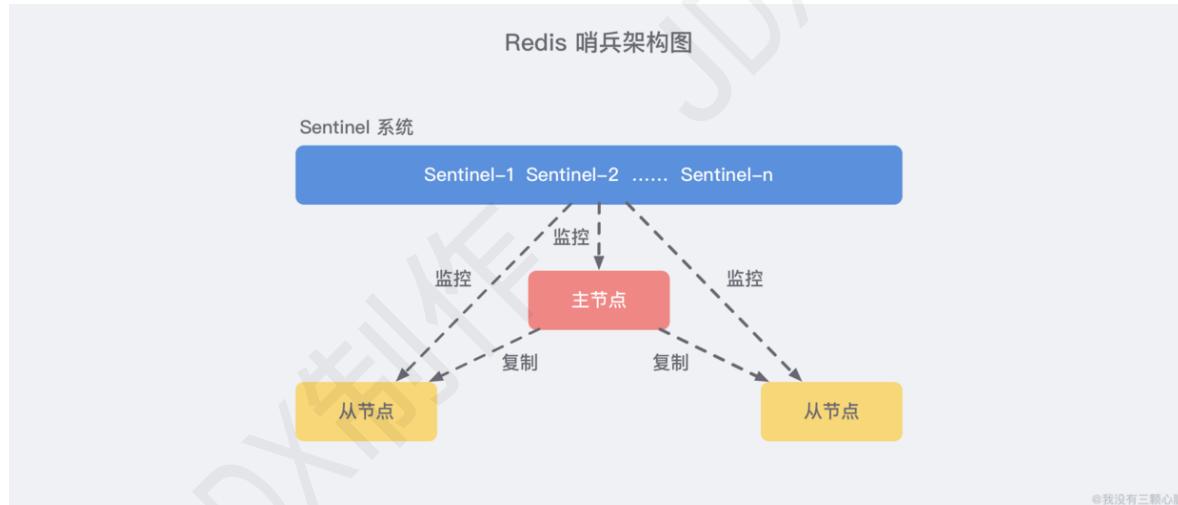
③、PSYNC 命令的引入

所以在 **Redis 2.8** 中引入了 `PSYNC` 命令来代替 `SYNC`，它具有两种模式：

- 全量复制**: 用于初次复制或其他无法进行部分复制的情况，将主节点中的所有数据都发送给从节点，是一个非常重型的操作；
- 部分复制**: 用于网络中断等情况后的复制，只将 **中断期间主节点执行的写命令** 发送给从节点，与全量复制相比更加高效。需要注意的是，如果网络中断时间过长，导致主节点没有能够完整地保存中断期间执行的写命令，则无法进行部分复制，仍使用全量复制；

部分复制的原理主要是靠主从节点分别维护一个 **复制偏移量**，有了这个偏移量之后断线重连之后一比较，之后就可以仅仅把从服务器断线之后确实的数据给补回来了。

11.3 Redis Sentinel 哨兵



上图展示了一个典型的哨兵架构图，它由两部分组成，哨兵节点和数据节点：

- 哨兵节点**: 哨兵系统由一个或多个哨兵节点组成，哨兵节点是特殊的 Redis 节点，不存储数据；
- 数据节点**: 主节点和从节点都是数据节点；

在复制的基础上，哨兵实现了 **自动化的故障恢复** 功能，下方是官方对于哨兵功能的描述：

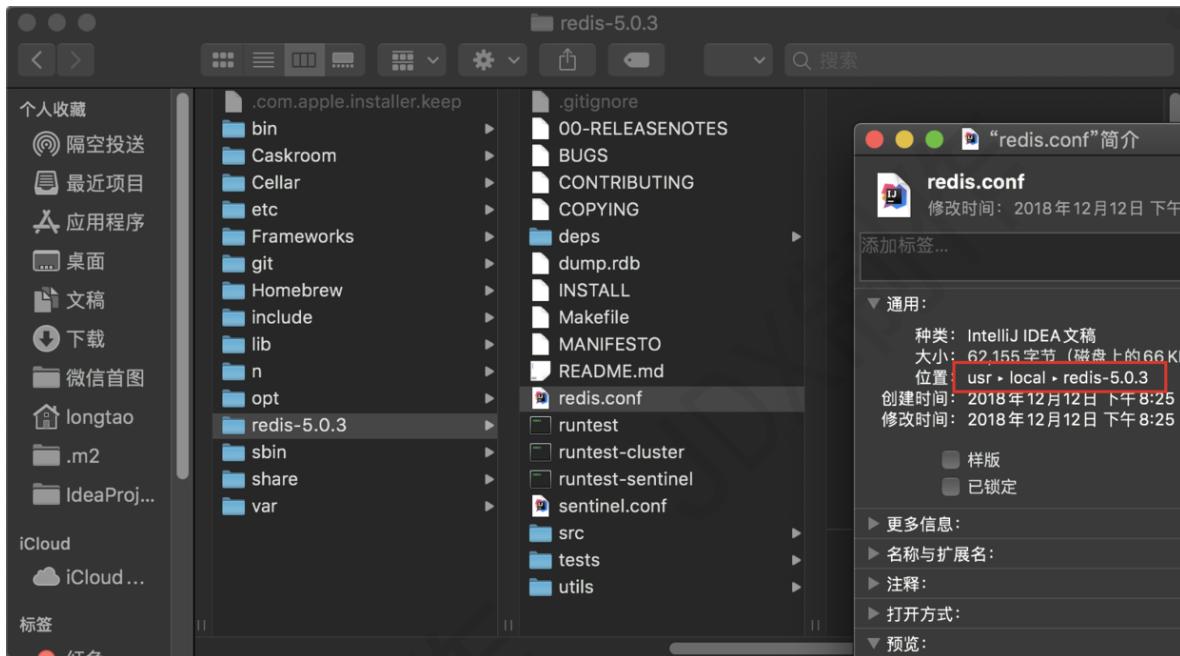
- 监控 (Monitoring)** : 哨兵会不断地检查主节点和从节点是否运作正常。
- 自动故障转移 (Automatic failover)** : 当 **主节点** 不能正常工作时，哨兵会开始 **自动故障转移** 操作，它会将失效主节点的其中一个 **从节点升级为新的主节点**，并让其他从节点改为复制新的主节点。
- 配置提供者 (Configuration provider)** : 客户端在初始化时，通过连接哨兵来获得当前 Redis 服务的主节点地址。
- 通知 (Notification)** : 哨兵可以将故障转移的结果发送给客户端。

其中，监控和自动故障转移功能，使得哨兵可以及时发现主节点故障并完成转移。而配置提供者和通知功能，则需要在与客户端的交互中才能体现。

11.3.1 快速体验

①、第一步：创建主从节点配置文件并启动

正确安装好 Redis 之后，我们去到 Redis 的安装目录 (mac 默认在 `/usr/local/`)，找到 `redis.conf` 文件复制三份分别命名为 `redis-master.conf`/`redis-slave1.conf`/`redis-slave2.conf`，分别作为 ① 个主节点和 ② 个从节点的配置文件 (下图演示了我本机的 `redis.conf` 文件的位置)



打开可以看到这个 `.conf` 后缀的文件里面有很多说明的内容，全部删除然后分别改成下面的样子：

```
#redis-master.conf
port 6379
daemonize yes
logfile "6379.log"
dbfilename "dump-6379.rdb"

#redis-slave1.conf
port 6380
daemonize yes
logfile "6380.log"
dbfilename "dump-6380.rdb"
slaveof 127.0.0.1 6379

#redis-slave2.conf
port 6381
daemonize yes
logfile "6381.log"
dbfilename "dump-6381.rdb"
slaveof 127.0.0.1 6379
```

然后我们可以执行 `redis-server <config file path>` 来根据配置文件启动不同的 Redis 实例，依次启动主从节点：

```
redis-server /usr/local/redis-5.0.3/redis-master.conf
redis-server /usr/local/redis-5.0.3/redis-slave1.conf
redis-server /usr/local/redis-5.0.3/redis-slave2.conf
```

节点启动后，我们执行 `redis-cli` 默认连接到我们端口为 6379 的主节点执行 `info Replication` 检查一下主从状态是否正常：(可以看到下方正确地显示了两个从节点)

②、第二步：创建哨兵节点配置文件并启动

按照上面同样的方法，我们给哨兵节点也创建三个配置文件。(哨兵节点本质上是特殊的 Redis 节点，所以配置几乎没什么差别，只是在端口上做区分就好)

```
# redis-sentinel-1.conf
port 26379
daemonize yes
logfile "26379.log"
sentinel monitor mymaster 127.0.0.1 6379 2

# redis-sentinel-2.conf
port 26380
daemonize yes
logfile "26380.log"
sentinel monitor mymaster 127.0.0.1 6379 2

# redis-sentinel-3.conf
port 26381
daemonize yes
logfile "26381.log"
sentinel monitor mymaster 127.0.0.1 6379 2
```

其中，`sentinel monitor mymaster 127.0.0.1 6379 2` 配置的含义是：该哨兵节点监控 `127.0.0.1:6379` 这个主节点，该主节点的名称是 `mymaster`，最后的 `2` 的含义与主节点的故障判定有关：至少需要 `2` 个哨兵节点同意，才能判定主节点故障并进行故障转移。

执行下方命令将哨兵节点启动起来：

```
redis-server /usr/local/redis-5.0.3/redis-sentinel-1.conf --sentinel  
redis-server /usr/local/redis-5.0.3/redis-sentinel-2.conf --sentinel  
redis-server /usr/local/redis-5.0.3/redis-sentinel-3.conf --sentinel
```

使用 `redis-cli` 工具连接哨兵节点，并执行 `info sentinel` 命令来查看是否已经在监视主节点了：

```
# 连接端口为 26379 的 Redis 节点  
→ ~ redis-cli -p 26379  
127.0.0.1:26379> info sentinel  
# Sentinel  
sentinel_masters:1  
sentinel_tilt:0  
sentinel_running_scripts:0  
sentinel_scripts_queue_length:0  
sentinel_simulate_failure_flags:0  
master0:name=mymaster,status=ok,address=127.0.0.1:6379,slaves=2,sentinels=3
```

此时你打开刚才写好的哨兵配置文件，你还会发现出现了一些变化：

③、第三步：演示故障转移

首先，我们使用 `kill -9` 命令来杀掉主节点，同时在哨兵节点中执行 `info sentinel` 命令来观察故障节点的过程：

```
→ ~ ps aux | grep 6379  
longtao      74529  0.3  0.0  4346936   2132 ?? Ss 10:30上午 0:03.09  
redis-server *:26379 [sentinel]  
longtao      73541  0.2  0.0  4348072   2292 ?? Ss 10:18上午 0:04.79  
redis-server *:6379  
longtao      75521  0.0  0.0  4286728    728 S008 S+ 10:39上午 0:00.00  
grep --color=auto --exclude-dir=.bzr --exclude-dir=CVS --exclude-dir=.git --  
--exclude-dir=.hg --exclude-dir=.svn 6379  
longtao      74836  0.0  0.0  4289844    944 S006 S+ 10:32上午 0:00.01  
redis-cli -p 26379  
→ ~ kill -9 73541
```

如果 **刚杀掉瞬间** 在哨兵节点中执行 `info` 命令来查看，会发现主节点还没有切换过来，因为哨兵发现主节点故障并转移需要一段时间：

```
# 第一时间查看哨兵节点发现并未转移，还在 6379 端口  
127.0.0.1:26379> info sentinel  
# Sentinel  
sentinel_masters:1  
sentinel_tilt:0  
sentinel_running_scripts:0  
sentinel_scripts_queue_length:0  
sentinel_simulate_failure_flags:0  
master0:name=mymaster,status=ok,address=127.0.0.1:6379,slaves=2,sentinels=3
```

一段时间之后你再执行 `info` 命令，查看，你就会发现主节点已经切换成了 `6381` 端口的从节点：

```

# 过一段时间之后在执行，发现已经切换了 6381 端口
127.0.0.1:26379> info Sentinel
# Sentinel
sentinel_masters:1
sentinel_tilt:0
sentinel_running_scripts:0
sentinel_scripts_queue_length:0
sentinel_simulate_failure_flags:0
master0:name=mymaster,status=ok,address=127.0.0.1:6381,slaves=2,sentinels=3

```

但同时还可以发现，哨兵节点认为新的主节点仍然有两个从节点（上方 `slaves=2`），这是因为哨兵在将 `6381` 切换成主节点的同时，将 `6379` 节点置为其从节点。虽然 `6379` 从节点已经挂掉，但是由于 **哨兵并不会对从节点进行客观下线**，因此认为该从节点一直存在。当 `6379` 节点重新启动后，会自动变成 `6381` 节点的从节点。

另外，在故障转移的阶段，哨兵和主从节点的配置文件都会被改写：

- **对于主从节点：**主要是 `slaveof` 配置的变化，新的主节点没有了 `slaveof` 配置，其从节点则 `slaveof` 新的主节点。
- **对于哨兵节点：**除了主从节点信息的变化，纪元(epoch)（记录当前集群状态的参数）也会变化，纪元相关的参数都 +1 了。

11.3.2 客户端访问哨兵系统代码演示

上面我们在 快速体验 中主要感受到了服务端自己对于当前主从节点的自动化治理，下面我们以 Java 代码为例，来演示一下客户端如何访问我们的哨兵系统：

```

public static void testSentinel() throws Exception {
    String masterName = "mymaster";
    Set<String> sentinels = new HashSet<>();
    sentinels.add("127.0.0.1:26379");
    sentinels.add("127.0.0.1:26380");
    sentinels.add("127.0.0.1:26381");

    // 初始化过程做了很多工作
    JedisSentinelPool pool = new JedisSentinelPool(masterName, sentinels);
    Jedis jedis = pool.getResource();
    jedis.set("key1", "value1");
    pool.close();
}

```

①、客户端原理

Jedis 客户端对哨兵提供了很好的支持。如上述代码所示，我们只需要向 Jedis 提供哨兵节点集合和 `masterName`，构造 `JedisSentinelPool` 对象，然后便可以像使用普通 Redis 连接池一样来使用了：通过 `pool.getResource()` 获取连接，执行具体的命令。

在整个过程中，我们的代码不需要显式的指定主节点的地址，就可以连接到主节点；代码中对故障转移没有任何体现，就可以在哨兵完成故障转移后自动的切换主节点。之所以可以做到这一点，是因为在 `JedisSentinelPool` 的构造器中，进行了相关的工作；主要包括以下两点：

- 1. 遍历哨兵节点，获取主节点信息：**遍历哨兵节点，通过其中一个哨兵节点 + `masterName` 获得主节点的信息；该功能是通过调用哨兵节点的 `sentinel get-master-addr-by-name` 命令实现；
- 2. 增加对哨兵的监听：**这样当发生故障转移时，客户端便可以收到哨兵的通知，从而完成主节点的切换。具体做法是：利用 Redis 提供的 **发布订阅** 功能，为每一个哨兵节点开启一个单独的线程，订阅哨兵节点的 `+switch-master` 频道，当收到消息时，重新初始化连接池。

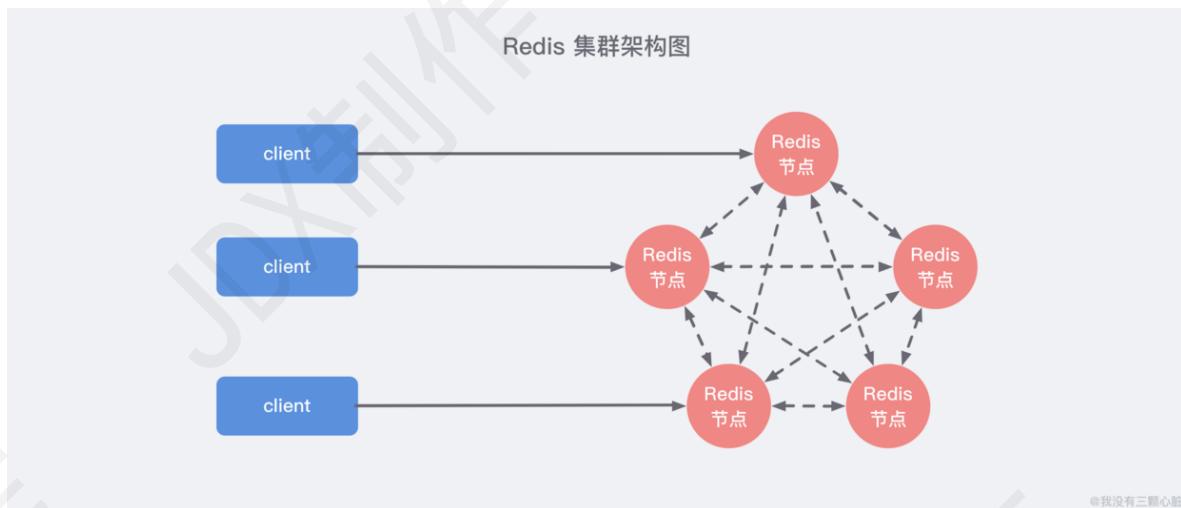
11.3.3 新的主服务器是怎样被挑选出来的？

故障转移操作的第一步 要做的就是在已下线主服务器属下的所有从服务器中，挑选出一个状态良好、数据完整的从服务器，然后向这个从服务器发送 `slaveof no one` 命令，将这个从服务器转换为主服务器。但是这个从服务器是怎么样被挑选出来的呢？

简单来说 Sentinel 使用以下规则来选择新的主服务器：

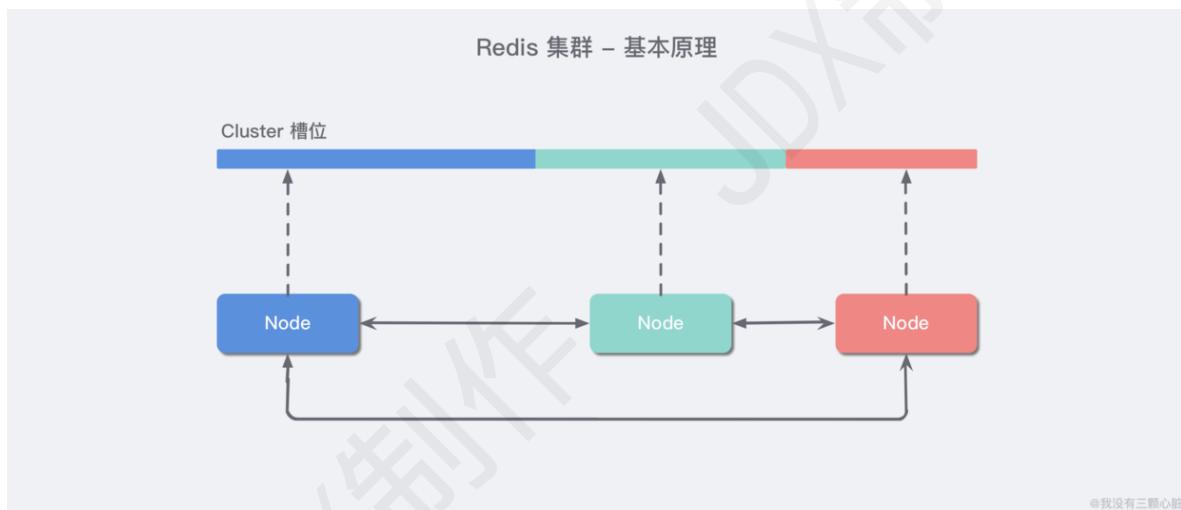
1. 在失效主服务器属下的从服务器当中，那些被标记为主观下线、已断线、或者最后一次回复 PING 命令的时间大于五秒钟的从服务器都会被 **淘汰**。
2. 在失效主服务器属下的从服务器当中，那些与失效主服务器连接断开的时长超过 `down-after` 选项指定的时长十倍的从服务器都会被 **淘汰**。
3. 在 **经历了以上两轮淘汰之后** 剩下来的从服务器中，我们选出 **复制偏移量 (replication offset)** **最大** 的那个 **从服务器** 作为新的主服务器；如果复制偏移量不可用，或者从服务器的复制偏移量相同，那么 **带有最小运行 ID** 的那个从服务器成为新的主服务器。

11.4 Redis 集群



上图展示了 Redis Cluster 典型的架构图，集群中的每一个 Redis 节点都 **互相两两相连**，客户端任意直连到集群中的 **任意一台**，就可以对其他 Redis 节点进行 **读写** 的操作。

11.4.1 基本原理



Redis 集群中内置了 16384 个哈希槽。当客户端连接到 Redis 集群之后，会同时得到一份关于这个 **集群的配置信息**，当客户端具体对某一个 `key` 值进行操作时，会计算出它的一个 Hash 值，然后把结果对 16384 **求余数**，这样每个 `key` 都会对应一个编号在 0-16383 之间的哈希槽，Redis 会根据节点数量 **大致均等** 的将哈希槽映射到不同的节点。

再结合集群的配置信息就能够知道这个 `key` 值应该存储在哪个具体的 Redis 节点中，如果不属于自己管，那么就会使用一个特殊的 `MOVED` 命令来进行一个跳转，告诉客户端去连接这个节点以获取数据：

```
GET X  
-MOVED 3999 127.0.0.1:6381
```

`MOVED` 指令第一个参数 `3999` 是 `key` 对应的槽位编号，后面是目标节点地址，`MOVED` 命令前面有一个减号，表示这是一个错误的消息。客户端在收到 `MOVED` 指令后，就立即纠正本地的 **槽位映射表**，那么下一次再访问 `key` 时就能够到正确的地方去获取了。

11.4.2 集群的主要作用

- 1. 数据分区：** 数据分区（或称数据分片）是集群最核心的功能。集群将数据分散到多个节点，**一方面**突破了 Redis 单机内存大小的限制，**存储容量大大增加**；**另一方面**每个主节点都可以对外提供读服务和写服务，**大大提高了集群的响应能力**。Redis 单机内存大小受限问题，在介绍持久化和主从复制时都有提及，例如，如果单机内存太大，`bgsave` 和 `bgrewriteaof` 的 `fork` 操作可能导致主进程阻塞，主从环境下主机切换时可能导致从节点长时间无法提供服务，全量复制阶段主节点的复制缓冲区可能溢出……
- 2. 高可用：** 集群支持主从复制和主节点的 **自动故障转移**（与哨兵类似），当任一节点发生故障时，集群仍然可以对外提供服务。

11.4.3 快速体验

①、第一步：创建集群节点配置文件

首先我们找一个地方创建一个名为 `redis-cluster` 的目录：

```
mkdir -p ~/Desktop/redis-cluster
```

然后按照上面的方法，创建六个配置文件，分别命名为：

`redis_7000.conf`/`redis_7001.conf`.....`redis_7005.conf`，然后根据不同的端口号修改对应的端口号值就好了：

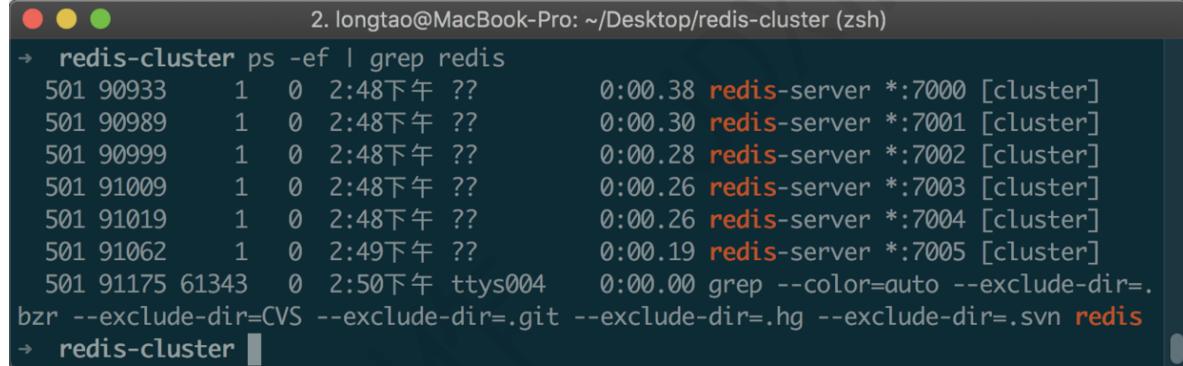
```
# 后台执行  
daemonize yes  
# 端口号  
port 7000  
# 为每一个集群节点指定一个 pid_file  
pidfile ~/Desktop/redis-cluster/redis_7000.pid  
# 启动集群模式  
cluster-enabled yes  
# 每一个集群节点都有一个配置文件，这个文件是不能手动编辑的。确保每一个集群节点的配置文件不通  
cluster-config-file nodes-7000.conf  
# 集群节点的超时时间，单位：ms，超时后集群会认为该节点失败  
cluster-node-timeout 5000  
# 最后将 appendonly 改成 yes(AOF 持久化)  
appendonly yes
```

记得把对应上述配置文件中根端口对应的配置都修改掉 (`port/ pidfile/ cluster-config-file`)。

②、第二步：分别启动 6 个 Redis 实例

```
redis-server ~/Desktop/redis-cluster/redis_7000.conf  
redis-server ~/Desktop/redis-cluster/redis_7001.conf  
redis-server ~/Desktop/redis-cluster/redis_7002.conf  
redis-server ~/Desktop/redis-cluster/redis_7003.conf  
redis-server ~/Desktop/redis-cluster/redis_7004.conf  
redis-server ~/Desktop/redis-cluster/redis_7005.conf
```

然后执行 `ps -ef | grep redis` 查看是否启动成功：



A terminal window titled "2. longtao@MacBook-Pro: ~/Desktop/redis-cluster (zsh)". It shows the command `redis-cluster ps -ef | grep redis` being run. The output lists six Redis processes (501, 50933, 50989, 50999, 51009, 51019) each running on a different port (7000, 7001, 7002, 7003, 7004, 7005). The command `bzr --exclude-dir=CVS --exclude-dir=.git --exclude-dir=.hg --exclude-dir=.svn redis` is also visible at the bottom.

```
redis-cluster ps -ef | grep redis  
501 90933 1 0 2:48下午 ?? 0:00.38 redis-server *:7000 [cluster]  
501 90989 1 0 2:48下午 ?? 0:00.30 redis-server *:7001 [cluster]  
501 90999 1 0 2:48下午 ?? 0:00.28 redis-server *:7002 [cluster]  
501 91009 1 0 2:48下午 ?? 0:00.26 redis-server *:7003 [cluster]  
501 91019 1 0 2:48下午 ?? 0:00.26 redis-server *:7004 [cluster]  
501 91062 1 0 2:49下午 ?? 0:00.19 redis-server *:7005 [cluster]  
501 91175 61343 0 2:50下午 ttys004 0:00.00 grep --color=auto --exclude-dir=.bzr --exclude-dir=CVS --exclude-dir=.git --exclude-dir=.hg --exclude-dir=.svn redis  
redis-cluster
```

可以看到 6 个 Redis 节点都以集群的方式成功启动了，但是现在每个节点还处于独立的状态，也就是说它们每一个都各自成了一个集群，还没有互相联系起来，我们需要手动地把他们之间建立起联系。

③、第三步：建立集群

执行下列命令：

```
redis-cli --cluster create --cluster-replicas 1 127.0.0.1:7000 127.0.0.1:7001  
127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005
```

- 这里稍微解释一下这个 `--replicas 1` 的意思是：我们希望为集群中的每个主节点创建一个从节点。

观察控制台输出：

```
2. longtao@MacBook-Pro: ~/Desktop/redis-cluster (zsh)
→ redis-cluster redis-cli --cluster create --cluster-replicas 1 127.0.0.1:7000 127.0.0.1:7001 127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
Adding replica 127.0.0.1:7004 to 127.0.0.1:7000
Adding replica 127.0.0.1:7005 to 127.0.0.1:7001
Adding replica 127.0.0.1:7003 to 127.0.0.1:7002
>>> Trying to optimize slaves allocation for anti-affinity
[WARNING] Some slaves are in the same host as their master
M: 4ec8c022e9d546c9b51deb9d85f6cf867bf73db6 127.0.0.1:7000
  slots:[0-5460] (5461 slots) master
M: 236cefaa9cdc295bc60a5bd1aed6a7152d4f384d 127.0.0.1:7001
  slots:[5461-10922] (5462 slots) master
M: a3406db9ae7144d17eb7df5bffe8b70bb5dd06b8 127.0.0.1:7002
  slots:[10923-16383] (5461 slots) master
S: 56a04742f36c6e84968cae871cd438935081e86f 127.0.0.1:7003
  replicates 4ec8c022e9d546c9b51deb9d85f6cf867bf73db6
S: d31cd1f423ab1e1849cac01ae927e4b6950f55d9 127.0.0.1:7004
  replicates 236cefaa9cdc295bc60a5bd1aed6a7152d4f384d
S: e2539c4398b8258d3f9ffa714bd778da107cb2cd 127.0.0.1:7005
  replicates a3406db9ae7144d17eb7df5bffe8b70bb5dd06b8
Can I set the above configuration? (type 'yes' to accept): yes 这里要输一下 yes
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join
...
>>> Performing Cluster Check (using node 127.0.0.1:7000)
M: 4ec8c022e9d546c9b51deb9d85f6cf867bf73db6 127.0.0.1:7000
  slots:[0-5460] (5461 slots) master
  1 additional replica(s)
M: a3406db9ae7144d17eb7df5bffe8b70bb5dd06b8 127.0.0.1:7002
  slots:[10923-16383] (5461 slots) master
  1 additional replica(s)
S: 56a04742f36c6e84968cae871cd438935081e86f 127.0.0.1:7003
  slots: (0 slots) slave
  replicates 4ec8c022e9d546c9b51deb9d85f6cf867bf73db6
S: e2539c4398b8258d3f9ffa714bd778da107cb2cd 127.0.0.1:7005
  slots: (0 slots) slave
  replicates a3406db9ae7144d17eb7df5bffe8b70bb5dd06b8
M: 236cefaa9cdc295bc60a5bd1aed6a7152d4f384d 127.0.0.1:7001
  slots:[5461-10922] (5462 slots) master
  1 additional replica(s)
S: d31cd1f423ab1e1849cac01ae927e4b6950f55d9 127.0.0.1:7004
  slots: (0 slots) slave
  replicates 236cefaa9cdc295bc60a5bd1aed6a7152d4f384d
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered. 16384 个槽点均已覆盖
→ redis-cluster ]
```

看到 [OK] 的信息之后，就表示集群已经搭建成功了，可以看到，这里我们正确地创建了三主三从的集群。

④、第四步：验证集群

我们先使用 `redis-cli` 任意连接一个节点：

```
redis-cli -c -h 127.0.0.1 -p 7000  
127.0.0.1:7000>
```

- `-c` 表示集群模式；`-h` 指定 ip 地址；`-p` 指定端口。

然后随便 `set` 一些值观察控制台输入：

```
127.0.0.1:7000> SET name wmyskxz  
-> Redirected to slot [5798] located at 127.0.0.1:7001  
OK  
127.0.0.1:7001>
```

可以看到这里 Redis 自动帮我们进行了 `Redirected` 操作跳转到了 `7001` 这个实例上。

我们再使用 `cluster info` (查看集群信息) 和 `cluster nodes` (查看节点列表) 来分别看看：(任意节点输入均可)

```
127.0.0.1:7001> CLUSTER INFO  
cluster_state:ok  
cluster_slots_assigned:16384  
cluster_slots_ok:16384  
cluster_slots_pfail:0  
cluster_slots_fail:0  
cluster_known_nodes:6  
cluster_size:3  
cluster_current_epoch:6  
cluster_my_epoch:2  
cluster_stats_messages_ping_sent:1365  
cluster_stats_messages_pong_sent:1358  
cluster_stats_messages_meet_sent:4  
cluster_stats_messages_sent:2727  
cluster_stats_messages_ping_received:1357  
cluster_stats_messages_pong_received:1369  
cluster_stats_messages_meet_received:1  
cluster_stats_messages_received:2727  
  
127.0.0.1:7001> CLUSTER NODES  
56a04742f36c6e84968cae871cd438935081e86f 127.0.0.1:7003@17003 slave  
4ec8c022e9d546c9b51deb9d85f6cf867bf73db6 0 1584428884000 4 connected  
4ec8c022e9d546c9b51deb9d85f6cf867bf73db6 127.0.0.1:7000@17000 master - 0  
1584428884000 1 connected 0-5460  
e2539c4398b8258d3f9ffa714bd778da107cb2cd 127.0.0.1:7005@17005 slave  
a3406db9ae7144d17eb7df5bffe8b70bb5dd06b8 0 1584428885222 6 connected  
d31cd1f423ab1e1849cac01ae927e4b6950f55d9 127.0.0.1:7004@17004 slave  
236cef9aa9cdc295bc60a5bd1aed6a7152d4f384d 0 1584428884209 5 connected  
236cef9aa9cdc295bc60a5bd1aed6a7152d4f384d 127.0.0.1:7001@17001 myself,master - 0  
1584428882000 2 connected 5461-10922  
a3406db9ae7144d17eb7df5bffe8b70bb5dd06b8 127.0.0.1:7002@17002 master - 0  
1584428884000 3 connected 10923-16383  
127.0.0.1:7001>
```

11.4.5 数据分区方案简析

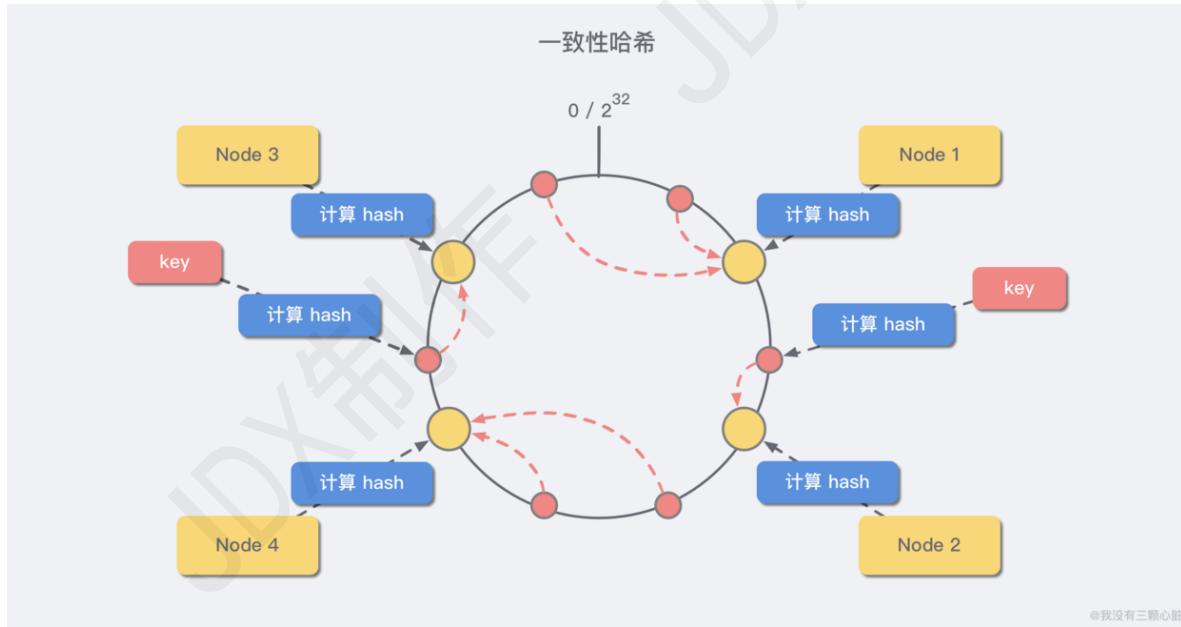
①、方案一：哈希值 % 节点数

哈希取余分区思路非常简单：计算 `key` 的 hash 值，然后对节点数量进行取余，从而决定数据映射到哪个节点上。

不过该方案最大的问题是，当新增或删减节点时，节点数量发生变化，系统中所有的数据都需要重新计算映射关系，引发大规模数据迁移。

②、方案二：一致性哈希分区

一致性哈希算法将 整个哈希值空间 组织成一个虚拟的圆环，范围是 $[0, 2^{32}-1]$ ，对于每一个数据，根据 `key` 计算 hash 值，确定数据在环上的位置，然后从此位置沿顺时针行走，找到的第一台服务器就是其应该映射到的服务器：



与哈希取余分区相比，一致性哈希分区将 增减节点的影响限制在相邻节点。以上图为例，如果在 `node1` 和 `node2` 之间增加 `node5`，则只有 `node2` 中的一部分数据会迁移到 `node5`；如果去掉 `node2`，则原 `node2` 中的数据只会迁移到 `node4` 中，只有 `node4` 会受影响。

一致性哈希分区的主要问题在于，当 节点数量较少 时，增加或删减节点， 对单个节点的影响可能很大，造成数据的严重不平衡。还是以上图为例，如果去掉 `node2`，`node4` 中的数据由总数据的 `1/4` 左右变为 `1/2` 左右，与其他节点相比负载过高。

③、方案三：带有虚拟节点的一致性哈希分区

该方案在 一致性哈希分区 的基础上，引入了 虚拟节点 的概念。Redis 集群使用的便是该方案，其中的虚拟节点称为 槽 (slot) 。槽是介于数据和实际节点之间的虚拟概念，每个实际节点包含一定数量的槽，每个槽包含哈希值在一定范围内的数据。

在使用了槽的一致性哈希分区中， 槽是数据管理和迁移的基本单位。槽解耦了 数据和实际节点 之间的关系，增加或删除节点对系统的影响很小。仍以上图为例，系统中有 4 个实际节点，假设为其分配 16 个槽(0-15)；

- 槽 0-3 位于 `node1`；4-7 位于 `node2`；以此类推....

如果此时删除 `node2`，只需要将槽 4-7 重新分配即可，例如槽 4-5 分配给 `node1`，槽 6 分配给 `node3`，槽 7 分配给 `node4`；可以看出删除 `node2` 后，数据在其他节点的分布仍然较为均衡。

11.4.6 节点通信机制简析

集群的建立离不开节点之间的通信，例如我们上节在 快速体验 中刚启动六个集群节点之后通过 `redis-cli` 命令帮助我们搭建起来了集群，实际上背后每个集群之间的两两连接是通过了 `CLUSTER MEET <ip> <port>` 命令发送 `MEET` 消息完成的，下面我们展开详细说说。

①、两个端口

在 **哨兵系统** 中，节点分为 **数据节点** 和 **哨兵节点**：前者存储数据，后者实现额外的控制功能。在 **集群** 中，没有数据节点与非数据节点之分：**所有的节点都存储数据，也都参与集群状态的维护**。为此，集群中的每个节点，都提供了两个 TCP 端口：

- **普通端口**：即我们在前面指定的端口 (7000等)。普通端口主要用于为客户端提供服务（与单机节点类似）；但在节点间数据迁移时也会使用。
- **集群端口**：端口号是普通端口 + 10000（10000是固定值，无法改变），如 7000 节点的集群端口为 17000。**集群端口只用于节点之间的通信**，如搭建集群、增减节点、故障转移等操作时节点间的通信；不要使用客户端连接集群接口。为了保证集群可以正常工作，在配置防火墙时，要同时开启普通端口和集群端口。

②Gossip 协议

节点间通信，按照通信协议可以分为几种类型：单对单、广播、Gossip 协议等。重点是广播和 Gossip 的对比。

- 广播是指向集群内所有节点发送消息。**优点** 是集群的收敛速度快(集群收敛是指集群内所有节点获得的集群信息是一致的)，**缺点** 是每条消息都要发送给所有节点，CPU、带宽等消耗较大。
- Gossip 协议的特点是：在节点数量有限的网络中，**每个节点都“随机”的与部分节点通信**（并不是真正的随机，而是根据特定的规则选择通信的节点），经过一番杂乱无章的通信，每个节点的状态很快会达到一致。Gossip 协议的**优点** 有负载(比广播)低、去中心化、容错性高(因为通信有冗余)等；**缺点** 主要是集群的收敛速度慢。

③、消息类型

集群中的节点采用 **固定频率（每秒10次）** 的 **定时任务** 进行通信相关的工作：判断是否需要发送消息及消息类型、确定接收节点、发送消息等。如果集群状态发生了变化，如增减节点、槽状态变更，通过节点间的通信，所有节点会很快得知整个集群的状态，使集群收敛。

节点间发送的消息主要分为 5 种：`meet` 消息、`ping` 消息、`pong` 消息、`fail` 消息、`publish` 消息。不同的消息类型，通信协议、发送的频率和时机、接收节点的选择等是不同的：

- **MEET 消息**：在节点握手阶段，当节点收到客户端的 `CLUSTER MEET` 命令时，会向新加入的节点发送 `MEET` 消息，请求新节点加入到当前集群；新节点收到 `MEET` 消息后会回复一个 `PONG` 消息。
- **PING 消息**：集群里每个节点每秒钟会选择部分节点发送 `PING` 消息，接收者收到消息后会回复一个 `PONG` 消息。**PING 消息的内容是自身节点和部分其他节点的状态信息**，作用是彼此交换信息，以及检测节点是否在线。`PING` 消息使用 Gossip 协议发送，接收节点的选择兼顾了收敛速度和带宽成本，**具体规则如下**：(1)随机找 5 个节点，在其中选择最久没有通信的 1 个节点；(2)扫描节点列表，选择最近一次收到 `PONG` 消息时间大于 `cluster_node_timeout / 2` 的所有节点，防止这些节点长时间未更新。
- **PONG消息**：`PONG` 消息封装了自身状态数据。可以分为两种：**第一种** 是在接到 `MEET/PING` 消息后回复的 `PONG` 消息；**第二种** 是指节点向集群广播 `PONG` 消息，这样其他节点可以获知该节点的最新信息，例如故障恢复后新的主节点会广播 `PONG` 消息。
- **FAIL 消息**：当一个主节点判断另一个主节点进入 `FAIL` 状态时，会向集群广播这一 `FAIL` 消息；接收节点会将这一 `FAIL` 消息保存起来，便于后续的判断。
- **PUBLISH 消息**：节点收到 `PUBLISH` 命令后，会先执行该命令，然后向集群广播这一消息，接收节点也会执行该 `PUBLISH` 命令。

11.4.7 数据结构简析

节点需要专门的数据结构来存储集群的状态。所谓集群的状态，是一个比较大的概念，包括：集群是否处于上线状态、集群中有哪些节点、节点是否可达、节点的主从状态、槽的分布……

节点为了存储集群状态而提供的数据结构中，最关键的是 `clusterNode` 和 `clusterState` 结构：前者记录了一个节点的状态，后者记录了集群作为一个整体的状态。

①、`clusterNode` 结构

`clusterNode` 结构保存了 **一个节点的当前状态**，包括创建时间、节点 id、ip 和端口号等。每个节点都会用一个 `clusterNode` 结构记录自己的状态，并为集群内所有其他节点都创建一个 `clusterNode` 结构来记录节点状态。

下面列举了 `clusterNode` 的部分字段，并说明了字段的含义和作用：

```
typedef struct clusterNode {
    //节点创建时间
    mstime_t ctime;
    //节点id
    char name[REDIS_CLUSTER_NAMELEN];
    //节点的ip和端口号
    char ip[REDIS_IP_STR_LEN];
    int port;
    //节点标识：整型，每个bit都代表了不同状态，如节点的主从状态、是否在线、是否在握手等
    int flags;
    //配置纪元：故障转移时起作用，类似于哨兵的配置纪元
    uint64_t configEpoch;
    //槽在该节点中的分布：占用16384/8个字节，16384个比特；每个比特对应一个槽：比特值为1，则该比特对应的槽在节点中；比特值为0，则该比特对应的槽不在节点中
    unsigned char slots[16384/8];
    //节点中槽的数量
    int numslots;
    .....
} clusterNode;
```

除了上述字段，`clusterNode` 还包含节点连接、主从复制、故障发现和转移需要的信息等。

②、`clusterState` 结构

`clusterState` 结构保存了在当前节点视角下，集群所处的状态。主要字段包括：

```
typedef struct clusterState {
    //自身节点
    clusterNode *myself;
    //配置纪元
    uint64_t currentEpoch;
    //集群状态：在线还是下线
    int state;
    //集群中至少包含一个槽的节点数量
    int size;
    //哈希表，节点名称->clusterNode节点指针
    dict *nodes;
    //槽分布信息：数组的每个元素都是一个指向clusterNode结构的指针；如果槽还没有分配给任何节点，则为NULL
    clusterNode *slots[16384];
    .....
} clusterState;
```

除此之外，`clusterState` 还包括故障转移、槽迁移等需要的信息。

12. Redis数据类型、编码、底层数据结构

12.1 Redis构建的类型系统

Redis构建了自己的类型系统，主要包括

- redisObject对象
- 基于redisObject对象的类型检查
- 基于redisObject对象的显示多态函数
- 对redisObject进行分配、共享和销毁的机制

C语言不是面向对象语言，这里将redisObject称呼为对象是为了讲述方便，让里面的内容更容易被理解，redisObject其实是一个结构体。

12.1.1 redisObject对象

Redis内部使用一个redisObject对象来表示所有的key和value，每次在Redis数据块中创建一个键值对时，一个是键对象，一个是值对象，而Redis中的每个对象都是由redisObject结构来表示。

在Redis中，键总是一个字符串对象，而值可以是字符串、列表、集合等对象，所以我们通常说键为字符串键，表示这个键对应的值为字符串对象，我们说一个键为集合键时，表示这个键对应的值为集合对象

redisobject最主要的信息：

```
redisobject源码
typedef struct redisObject{
    //类型
    unsigned type:4;
    //编码
    unsigned encoding:4;
    //指向底层数据结构的指针
    void *ptr;
    //引用计数
    int refcount;
    //记录最后一次被程序访问的时间
    unsigned lru:22;
}robj
```

- type代表一个value对象具体是何种数据类型
 - type key：判断对象的数据类型
- encoding属性和*ptr指针
 - ptr指针指向对象底层的数据结构，而数据结构由encoding属性来决定

编码常量	编码所对应的底层数据结构
REDIS_ENCODING_INT	long类型的整数
REDIS_ENCODING_EMBSTR	embstr编码的简单动态字符串
REDIS_ENCODING_RAW	简单动态字符串
REDIS_ENCODING_HT	字典
REDIS_ENCODING_LINKEDLIST	双端链表
REDIS_ENCODING_ZIPLIST	压缩列表
REDIS_ENCODING_INTSET	整数集合
REDIS_ENCODING_SKIPLIST	跳跃表和字典

- 每种类型的对象至少使用了两种不同的编码，而这些编码对用户是完全透明的。

类型	编码	对象
REDIS_STRING	REDIS_ENCODING_INT	使用整数值实现的字符串对象
REDIS_STRING	REDIS_ENCODING_EMBSTR	使用embstr编码的简单动态字符串实现的字符串对象
REDIS_STRING	REDIS_ENCODING_RAW	使用简单动态字符串实现的字符串对象
REDIS_LIST	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的列表对象
REDIS_LIST	REDIS_ENCODING_LINKEDLIST	使用双端链表实现的列表对象
REDIS_HASH	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的哈希对象
REDIS_HASH	REDIS_ENCODING_HT	使用字典实现的哈希对象
REDIS_SET	REDIS_ENCODING_INTSET	使用整数集合实现的集合对象
REDIS_SET	REDIS_ENCODING_HT	使用字典实现的集合对象
REDIS_ZSET	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的有序集合对象
REDIS_ZSET	REDIS_ENCODING_SKIPLIST	使用跳跃表和字典实现的有序集合对象

- object encoding key命令可以查看值对象的编码

12.1.2 命令的类型检查和多态

①、Redis命令分类

- 一种是只能用于对应数据类型的命令，例如LPUSH和LLEN只能用于列表键，SADD 和 SRANDMEMBER只能用于集合键。
- 另一种是可以用于任何类型键的命令。比如TTL。

当执行一个处理数据类型的命令时，Redis执行以下步骤：

- 根据给定 key，在数据库字典中查找和它相对应的 redisobject，如果没找到，就返回 NULL。
- 检查 redisobject 的 type 属性和执行命令所需的类型是否相符，如果不相符，返回类型错误。
- 根据 redisObject 的 encoding 属性所指定的编码，选择合适的操作函数来处理底层的数据结构。
- 返回数据结构的操作结果作为命令的返回值。

12.2 5种数据类型对应的编码和数据结构

12.2.1 string

string是最常用的一种数据类型，普通的key/value存储都可以归结为string类型，value不仅是string，也可以是数字。其他几种数据类型的构成元素也都是字符串，注意Redis规定字符串的长度不能超过512M

编码字符串对象的编码可以是int raw embstr

- int编码
 - 保存的是可以用long类型表示的整数值
- raw编码
 - 保存长度大于44字节的字符串
- embstr编码
 - 保存长度小于44字节的字符串

int用来保存整数值，raw用来保存长字符串，embstr用来保存短字符串。embstr编码是用来专门保存短字符串的一种优化编码。

Redis中对于浮点型也是作为字符串保存的，在需要时再将其转换成浮点数类型

编码的转换

- 当 int 编码保存的值不再是整数，或大小超过了long的范围时，自动转化为raw

- 对于 embstr 编码，由于 Redis 没有对其编写任何的修改程序（embstr 是只读的），在对embstr 对象进行修改时，都会先转化为raw再进行修改，因此，只要是修改embstr对象，修改后的对象一定是raw的，无论是否达到了44个字节。

常用命令

- set/get
 - set:设置key对应的值为string类型的value（多次set name会覆盖）
 - get:获取key对应的值
- mset /mget
 - mset 批量设置多个key的值，如果成功表示所有值都被设置，否则返回0表示没有任何值被设置
 - mget批量获取多个key的值，如果不存在则返回null

```
127.0.0.1:6379> mset user1:name redis user1:age 22
OK
127.0.0.1:6379> mget user1:name user1:age
1) "redis"
2) "22"
```

应用场景

- 类似于哈希操作，存储对象

incr && incrby<原子操作>

- incr对key对应的值进行加加操作，并返回新的值，incrby加指定的值

decr && decrby<原子操作>

- decr对key对应的值进行减减操作，并返回新的值，decrby减指定的值

setnx <小小体验一把分布式锁，真香>

- 设置Key对应的值为string类型的值，如果已经存在则返回0

setex

- 设置key对应的值为string类型的value，并设定有效期

setrange/getrange

- setrange从指定位置替换字符串
- getrange获取key对应value子字符串

其他命令

- msetnx 同mset，不存在就设置，不会覆盖已有的key
- getset 设置key的值，并返回key旧的值
- append 给指定的key的value追加字符串，并返回新字符串的长度
- strlen 返回key对应的value字符串的长度

应用场景

- 因为string类型是二进制安全的，可以用来存放图片，视频等内容。
- 由于redis的高性能的读写功能，而string类型的value也可以是数字，可以用做计数器（使用INCR，DECR指令）。比如分布式环境中统计系统的在线人数，秒杀等。
- 除了上面提到的，还有用于SpringSession实现分布式session
- 分布式系统全局序列号

12.2.2 listlist列表,它是简单的字符串列表,你可以添加一个元素到列表的头部,或者尾部。

编码

- 列表对象的编码可以是ziplist (压缩列表) 和linkedlist (双端链表) 。
- 编码转换
 - 同时满足下面两个条件时使用压缩列表:
 - 列表保存元素个数小于512个
 - 每个元素长度小于64字节
 - 不能满足上面两个条件使用linkedlist (双端列表) 编码
- 常用命令
 - lpush: 从头部加入元素

```
127.0.0.1:6379> lpush list1 hello
(integer) 1
127.0.0.1:6379> lpush list1 world
(integer) 2
127.0.0.1:6379> lrange list1 0 -1
1) "world"
2) "hello"
```

- rpush: 从尾部加入元素

```
127.0.0.1:6379> rpush list2 world
(integer) 1
127.0.0.1:6379> rpush list2 hello
(integer) 2
127.0.0.1:6379> lrange list2 0 -1
1) "world"
2) "hello"
```

- lpop: 从list的头部删除元素,并返回删除的元素

```
127.0.0.1:6379> lrange list1 0 -1
1) "world"
2) "hello"
127.0.0.1:6379> lpop list1
"world"
127.0.0.1:6379> lrange list1 0 -1
1) "hello"
```

- rpop: 从list的尾部删除元素,并返回删除的元素

```
127.0.0.1:6379> lrange list2 0 -1
1) "hello"
2) "world"
127.0.0.1:6379> rpop list2
"world"
127.0.0.1:6379> lrange list2 0 -1
1) "hello"
```

- rpoplpush: 第一步从尾部删除元素,第二步从首部插入元素 结合着使用
- linsert :插入方法 linsert listname before [集合的元素] [插入的元素]

```
127.0.0.1:6379> lpush list3 hello
(integer) 1
127.0.0.1:6379> lpush list3 world
(integer) 2
127.0.0.1:6379> linsert list3 before hello start
(integer) 3
127.0.0.1:6379> lrange list3 0 -1
1) "world"
2) "start"
3) "hello"
```

- lset : 替换指定下标的元素

```
127.0.0.1:6379> lrange list1 0 -1
1) "a"
2) "b"
127.0.0.1:6379> lset list1 0 v
OK
127.0.0.1:6379> lrange list1 0 -1
1) "v"
2) "b"
```

- lrm : 删除元素, 返回删除的个数

```
127.0.0.1:6379> lrange list1 0 -1
1) "b"
2) "b"
3) "a"
4) "b"
127.0.0.1:6379> lrange list1 0 -1
1) "a"
2) "b"
```

- lindex: 返回list中指定位置的元素
- llen: 返回list中的元素的个数

实现数据结构

- Stack (栈)
 - LPUSH+LPOP
- Queue (队列)
 - LPUSH + RPOP
- Blocking MQ (阻塞队列)
 - LPUSH+BRPOP

应用场景

- 实现简单的消息队列
- 利用LRANGE命令, 实现基于Redis的分页功能

12.2.3 set

集合对象set是string类型 (整数也会转成string类型进行存储) 的无序集合。注意集合和列表的区别: 集合中的元素是无序的, 因此不能通过索引来操作元素; 集合中的元素不能有重复。

编码

- 集合对象的编码可以是intset或者hashtable
 - intset编码的集合对象使用整数集合作为底层实现，集合对象包含的所有元素都被保存在整数集合中。
 - hashtable编码的集合对象使用字典作为底层实现，字典的每个键都是一个字符串对象，这里的每个字符串对象就是一个集合中的元素，而字典的值全部设置为null。当使用HT编码时，Redis中的集合SET相当于Java中的HashSet，内部的键值对是无序的，唯一的。内部实现相当于一个特殊的字典，字典中所有value都是NULL。
- 编码转换
 - 当集合满足下列两个条件时，使用intset编码：
 - 集合对象中的所有元素都是整数
 - 集合对象所有元素数量不超过512

常用命令

- sadd：向集合中添加元素（set不允许元素重复）
- smembers：查看集合中的元素

```
127.0.0.1:6379> sadd set1 aaa
(integer) 1
127.0.0.1:6379> sadd set1 bbb
(integer) 1
127.0.0.1:6379> sadd set1 ccc
(integer) 1
127.0.0.1:6379> smembers set1
1) "aaa"
2) "ccc"
3) "bbb"
```

- srem：删除集合元素
- spop：随机返回删除的key
- sdiff：返回两个集合的不同元素（哪个集合在前就以哪个集合为标准）

```
127.0.0.1:6379> smembers set1
1) "ccc"
2) "bbb"
127.0.0.1:6379> smembers set2
1) "fff"
2) "rrr"
3) "bbb"
127.0.0.1:6379> sdiff set1 set2
1) "ccc"
127.0.0.1:6379> sdiff set2 set1
1) "fff"
2) "rrr"
```

- sinter：返回两个集合的交集
- sinterstore：返回交集结果，存入目标集合

```
127.0.0.1:6379> sinterstore set3 set1 set2
(integer) 1
127.0.0.1:6379> smembers set3
1) "bbb"
```

- sunion: 取两个集合的并集
- sunionstore: 取两个集合的并集，并存入目标集合
- smove: 将一个集合中的元素移动到另一个集合中
- scard: 返回集合中的元素个数
- sismember: 判断某元素是否存在某集合中，0代表否 1代表是
- srandmember: 随机返回一个元素

```
127.0.0.1:6379> srandmember set1 1
1) "bbb"
127.0.0.1:6379> srandmember set1 2
1) "ccc"
2) "bbb"
```

应用场景

- 对于 set 数据类型，由于底层是字典实现的，查找元素特别快，另外set 数据类型不允许重复，利用这两个特性我们可以进行全局去重，比如在用户注册模块，判断用户名是否注册；微信点赞，微信抽奖小程序
- 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，可能认识的人等功能。

12.2.4 zset

和集合对象相比，有序集合对象是有序的。与列表使用索引下表作为排序依据不同，有序集合为每一个元素设置一个分数 (score) 作为排序依据。

编码

- 有序集合的编码可以使ziplist或者skipool
 - ziplist编码的有序集合对象使用压缩列表作为底层实现，每个集合元素使用两个紧挨在一起的压缩列表节点来保存，第一个节点保存元素的成员，第二个节点保存元素的分值。并且压缩列表内的集合元素按分值从小到大的顺序进行排列，小的放置在靠近表头的位置，大的放置在靠近表尾的位置。
 - skipool编码的有序集合对象使用zset结构作为底层实现，一个zset结构同时包含一个字典和一个跳跃表

```
typedef struct zset{
    //跳跃表
    zskiplist *zsl;
    //字典
    dict *dice;
}
```

字典的键保存元素的值，字典的值保存元素的分值，跳跃表节点的object属性保存元素的成员，跳跃表节点的score属性保存元素的分值。这两种数据结构会通过指针来共享相同元素的成员和分值，所以不会产生重复成员和分值，造成内存的浪费。

- 编码转换
 - 当有序集合对象同时满足以下两个条件时，对象使用ziplist编码，否则使用skipool编码
 - 保存的元素数量小于128
 - 保存的所有元素长度都小于64字节

常用命令

- zrem: 删除集合中名称为key的元素member
- zincrby: 以指定值去自动递增
- zcard: 查看元素集合的个数
- zcount: 返回score在给定区间中的数量

```
127.0.0.1:6379> zrange zset 0 -1
1) "one"
2) "three"
3) "two"
4) "four"
5) "five"
6) "six"
127.0.0.1:6379> zcard zset
(integer) 6
127.0.0.1:6379> zcount zset 1 4
(integer) 4
```

- zrangebyscore: 找到指定区间范围的数据进行返回

```
127.0.0.1:6379> zrangebyscore zset 0 4 withscores
1) "one"
2) "1"
3) "three"
4) "2"
5) "two"
6) "2"
7) "four"
8) "4"
```

- zremrangebyrank zset from to: 删除索引

```
127.0.0.1:6379> zrange zset 0 -1
1) "one"
2) "three"
3) "two"
4) "four"
5) "five"
6) "six"
127.0.0.1:6379> zremrangebyrank zset 1 3
(integer) 3
127.0.0.1:6379> zrange zset 0 -1
1) "one"
2) "five"
3) "six"
```

- zremrangebyscore zset from to: 删除指定序号

```
127.0.0.1:6379> zrange zset 0 -1 withscores
1) "one"
2) "1"
3) "five"
4) "5"
5) "six"
6) "6"
127.0.0.1:6379> zremrangebyscore zset 3 6
(integer) 2
127.0.0.1:6379> zrange zset 0 -1 withscores
1) "one"
2) "1"
```

- zrank: 返回排序索引 (升序之后再找索引)
- zrevrank: 返回排序索引 (降序之后再找索引)

应用场景

- 对于 zset 数据类型，有序的集合，可以做范围查找，排行榜应用，取 TOP N 操作等。

12.2.5 hash

hash对象的键是一个字符串类型，值是一个键值对集合

编码

- hash对象的编码可以是ziplist或者hashtable
 - 当使用ziplist，也就是压缩列表作为底层实现时，新增的键值是保存到压缩列表的表尾。
 - hashtable 编码的hash表对象底层使用字典数据结构，哈希对象中的每个键值对都使用一个字典键值对。Redis中的字典相当于Java里面的HashMap，内部实现也差不多类似，都是通过“数组+链表”的链地址法来解决哈希冲突的，这样的结构吸收了两种不同数据结构的优点。
- 编码转换
 - 当同时满足下面两个条件使用ziplist编码，否则使用hashtable编码
 - 列表保存元素个数小于512个
 - 每个元素长度小于64字节
- hash是一个String类型的field和value之间的映射表
- Hash特别适合存储对象
- 所存储的成员较少时数据存储为zipmap,当成员数量增大时会自动转成真正的HashMap，此时encoding为ht
- Hash命令详解
 - hset/hget
 - hset hashname hashkey hashvalue
 - hget hashname hashkey

```
127.0.0.1:6379> hset user id 1
(integer) 1
127.0.0.1:6379> hset user name z3
(integer) 1
127.0.0.1:6379> hset user add shanxi
(integer) 1
127.0.0.1:6379> hget user id
"1"
127.0.0.1:6379> hget user name
"z3"
127.0.0.1:6379> hget user add
"shanxi"
```

- hmset/hmget
 - hmset hashname hashkey1hashvalue1 hashkey2 hashvalue2 hashkey3 hashvalue3
 - hget hashname hashkey1 hashkey2 hashkey3

```
127.0.0.1:6379> hmset user id 1 name z3 add shanxi
OK
127.0.0.1:6379> hmget user id name add
1) "1"
2) "z3"
3) "shanxi"
```

- hsetnx/hgetnx
- hincrby/hdecrby

```
127.0.0.1:6379> hincrby user2 id 3
(integer) 6
127.0.0.1:6379> hget user2 id
"6"
```

- hexist 判断是否存在key, 不存在返回0

```
127.0.0.1:6379> hget user2 id
"6"
```

- hlen 返回hash集合里所有的键值数

```
127.0.0.1:6379> hmset user3 id 3 name w5
OK
127.0.0.1:6379> hlen user3
(integer) 2
```

- hdel :删除指定的hash的key
- hkeys 返回hash里所有的字段
- hvals 返回hash里所有的value
- hgetall: 返回hash集合里所有的key和value

```
127.0.0.1:6379> hgetall user3
1) "id"
2) "3"
3) "name"
4) "w3"
5) "add"
6) "beijing"
```

优点

- 同类数据归类整合存储，方便数据管理，比如单个用户的所有商品都放在一个hash表里面。
- 相比string操作消耗内存cpu更小

缺点

- hash结构的存储消耗要高于单个字符串
- 过期功能不能使用在field上，只能用在key上
- redis集群架构不适合大规模使用

应用场景

- 对于 hash 数据类型，value 存放的是键值对，比如可以做单点登录存放用户信息。
- 存放商品信息，实现购物车

12.3 内存回收和内存共享

```
typedef struct redisObject{
    //类型
    unsigned type:4;
    //编码
    unsigned encoding:4;
    //指向底层数据结构的指针
    void *ptr;
    //引用计数
    int refcount;
    //记录最后一次被程序访问的时间
    unsigned lru:22;
}
robj
```

内存回收：因为c语言不具备自动内存回收功能，当将redisObject对象作为数据库的键或值而不是作为参数存储时其生命周期是非常长的，为了解决这个问题，Redis自己构建了一个内存回收机制，通过redisobject结构中的refcount实现。这个属性会随着对象的使用状态而不断变化。

1. 创建一个新对象，属性初始化为1
2. 对象被一个新程序使用，属性refcount加1
3. 对象不再被一个程序使用，属性refcount减1
4. 当对象的引用计数值变为0时，对象所占用的内存就会被释放

内存共享：refcount属性除了能实现内存回收以外，还能实现内存共享

1. 将数据块的键的值指针指向一个现有值的对象
2. 将被共享的值对象引用refcount加1 Redis的共享对象目前只支持整数值的字符串对象。之所以如此，实际上是对内存和CPU（时间）的平衡：共享对象虽然会降低内存消耗，但是判断两个对象是否相等却需要消耗额外的时间。对于整数值，判断操作复杂度为o(1)，对于普通字符串，判断复杂度为o(n)；而对于哈希、列表、集合和有序集合，判断的复杂度为o(n^2)。虽然共享的对象只能是整数值的字符串对象，但是5种类型都可能使用共享对象。

六、系统设计

(一). Restful API

RESTful API 是每个程序员都应该了解并掌握的基本知识，我们在开发过程中设计 API 的时候也应该至少要满足 RESTful API 的最基本的要求（比如接口中尽量使用名词，使用 POST 请求创建资源，DELETE 请求删除资源等等，示例：`GET /notes/id`：获取某个指定 id 的笔记的信息）。

如果你看 RESTful API 相关的文章的话一般都比较晦涩难懂，包括我下面的文章也会提到一些概念性的东西。但是，实际上我们平时开发用到的 RESTful API 的知识非常简单也很容易概括！举个例子，如果我给你下面两个 url 你是不是立马能知道它们是干什么的！这就是 RESTful API 的强大之处！

RESTful API 可以让你看到 url + http method 就知道这个 url 是干什么的，让你看到了 http 状态码 (status code) 就知道请求结果如何。

```
GET  /classes: 列出所有班级  
POST /classes: 新建一个班级
```

下面的内容只是介绍了我觉得关于 RESTful API 比较重要的一些东西，欢迎补充。

1. 重要概念

REST,即 **R**epresentational **S**tate **T**ransfer 的缩写。这个词组的翻译过来就是"表现层状态转化"。这样理解起来甚是晦涩，实际上 REST 的全称是 **R**esource **R**epresentational **S**tate **T**ransfer，直白地翻译过来就是“**资源**在**网络传输**中以某种“**表现形式**”进行“**状态转移**”。如果还是不能继续理解，请继续往下看，相信下面的讲解一定能让你理解到底啥是 REST。

我们分别对上面涉及到的概念进行解读，以便加深理解，不过实际上你不需要搞懂下面这些概念，也能看懂我下一部分要介绍到的内容。不过，为了更好地能跟别人扯扯“RESTful API”我建议你还是要好好理解一下！

- **资源 (Resource)**：我们可以把真实的对象数据称为资源。一个资源既可以是一个集合，也可以是单个个体。比如我们的班级 classes 是代表一个集合形式的资源，而特定的 class 代表单个个体资源。每一种资源都有特定的 URI (统一资源定位符) 与之对应，如果我们需要获取这个资源，访问这个 URI 就可以了，比如获取特定的班级：`/class/12`。另外，资源也可以包含子资源，比如 `/classes/classId/teachers`：列出某个指定班级的所有老师的信息。
- **表现形式 (Representational)**：“资源”是一种信息实体，它可以有多种外在表现形式。我们把“资源”具体呈现出来的形式比如 json, xml, image, txt 等等叫做它的“表现层/表现形式”。
- **状态转移 (State Transfer)**：大家第一眼看到这个词语一定会很懵逼？内心 BB：这尼玛是啥啊？大白话来说 REST 中的状态转移更多地描述的服务器端资源的状态，比如你通过增删改查（通过 HTTP 动词实现）引起资源状态的改变。ps:互联网通信协议 HTTP 协议，是一个无状态协议，所有的资源状态都保存在服务器端。

综合上面的解释，我们总结一下什么是 RESTful 架构：

1. 每一个 URI 代表一种资源；
2. 客户端和服务器之间，传递这种资源的某种表现形式比如 json, xml, image, txt 等等；
3. 客户端通过特定的 HTTP 动词，对服务器端资源进行操作，实现“表现层状态转化”。

2. REST 接口规范

2.1 动作

- GET：请求从服务器获取特定资源。举个例子：`GET /classes`（获取所有班级）

- POST：在服务器上创建一个新的资源。举个例子：`POST /classes`（创建班级）
- PUT：更新服务器上的资源（客户端提供更新后的整个资源）。举个例子：`PUT /classes/12`（更新编号为 12 的班级）
- DELETE：从服务器删除特定的资源。举个例子：`DELETE /classes/12`（删除编号为 12 的班级）
- PATCH：更新服务器上的资源（客户端提供更改的属性，可以看做是部分更新），使用的比较少，这里就不举例子了。

2.2 路径 (接口命名)

路径又称"终点" (endpoint)，表示 API 的具体网址。实际开发中常见的规范如下：

1. 网址中不能有动词，只能有名词，API 中的名词也应该使用复数。因为 REST 中的资源往往和数据库中的表对应，而数据库中的表都是同种记录的"集合" (collection)。如果 API 调用并不涉及资源 (如计算，翻译等操作) 的话，可以用动词。比如：`GET /calculate?param1=11¶m2=33`
2. 不用大写字母，建议用中杠 - 不用下杠_ 比如邀请码写成 `invitation-code` 而不是 `invitation_code`

Talk is cheap！来举个实际的例子来说明一下吧！现在有这样一个 API 提供班级 (class) 的信息，还包括班级中的学生和教师的信息，则它的路径应该设计成下面这样。

接口尽量使用名词，禁止使用动词。下面是一些例子：

```

GET /classes: 列出所有班级
POST /classes: 新建一个班级
GET /classes/classId: 获取某个指定班级的信息
PUT /classes/classId: 更新某个指定班级的信息（一般倾向整体更新）
PATCH /classes/classId: 更新某个指定班级的信息（一般倾向部分更新）
DELETE /classes/classId: 删除某个班级
GET /classes/classId/teachers: 列出某个指定班级的所有老师的信息
GET /classes/classId/students: 列出某个指定班级的所有学生的信息
DELETE classes/classId/teachers/ID: 删除某个指定班级下的指定的老师的信息
  
```

反例：

```

/getAllclasses
/createNewclass
/deleteAllActiveclasses
  
```

理清资源的层次结构，比如业务针对的范围是学校，那么学校会是一级资源：`/schools`，老师：`/schools/teachers`，学生：`/schools/students` 就是二级资源。

2.3 过滤信息 (Filtering)

如果我们在查询的时候需要添加特定条件的话，建议使用 url 参数的形式。比如我们要查询 state 状态为 active 并且 name 为 guidegege 的班级：

```
GET /classes?state=active&name=guidegege
```

比如我们要实现分页查询：

```
GET /classes?page=1&size=10 //指定第1页，每页10个数据
```

2.4 状态码 (Status Codes)

状态码范围：

2xx: 成功	3xx: 重定向	4xx: 客户端错误	5xx: 服务器错误
200 成功	301 永久重定向	400 错误请求	500 服务器错误
201 创建	304 资源未修改	401 未授权	502 网关错误
		403 禁止访问	504 网关超时
		404 未找到	
		405 请求方法不对	

3. HATEOAS

RESTful 的极致是 hateoas，但是这个基本不会在实际项目中用到。

上面是 RESTful API 最基本的东西，也是我们平时开发过程中最容易实践到的。实际上，RESTful API 最好做到 Hypermedia，即返回结果中提供链接，连向其他 API 方法，使得用户不查文档，也知道下一步应该做什么。

比如，当用户向 `api.example.com` 的根目录发出请求，会得到这样一个文档。

```
{"link": {  
    "rel": "collection https://www.example.com/classes",  
    "href": "https://api.example.com/classes",  
    "title": "List of classes",  
    "type": "application/vnd.yourformat+json"  
}}
```

上面代码表示，文档中有一个 `link` 属性，用户读取这个属性就知道下一步该调用什么 API 了。`rel` 表示这个 API 与当前网址的关系（collection 关系，并给出该 collection 的网址），`href` 表示 API 的路径，`title` 表示 API 的标题，`type` 表示返回类型 Hypermedia API 的设计被称为 HATEOAS。

在 Spring 中有一个叫做 HATEOAS 的 API 库，通过它我们可以更轻松的创建除符合 HATEOAS 设计的 API。

(二). 常用框架

1. Spring常见问题

1.1 什么是 Spring 框架？

Spring 是一种轻量级开发框架，旨在提高开发人员的开发效率以及系统的可维护性。

我们一般说 Spring 框架指的都是 Spring Framework，它是很多模块的集合，使用这些模块可以很方便地协助我们进行开发。这些模块是：核心容器、数据访问/集成、Web、AOP（面向切面编程）、工具、消息和测试模块。比如：Core Container 中的 Core 组件是 Spring 所有组件的核心，Beans 组件和 Context 组件是实现 IOC 和 依赖注入的基础，AOP 组件用来实现面向切面编程。

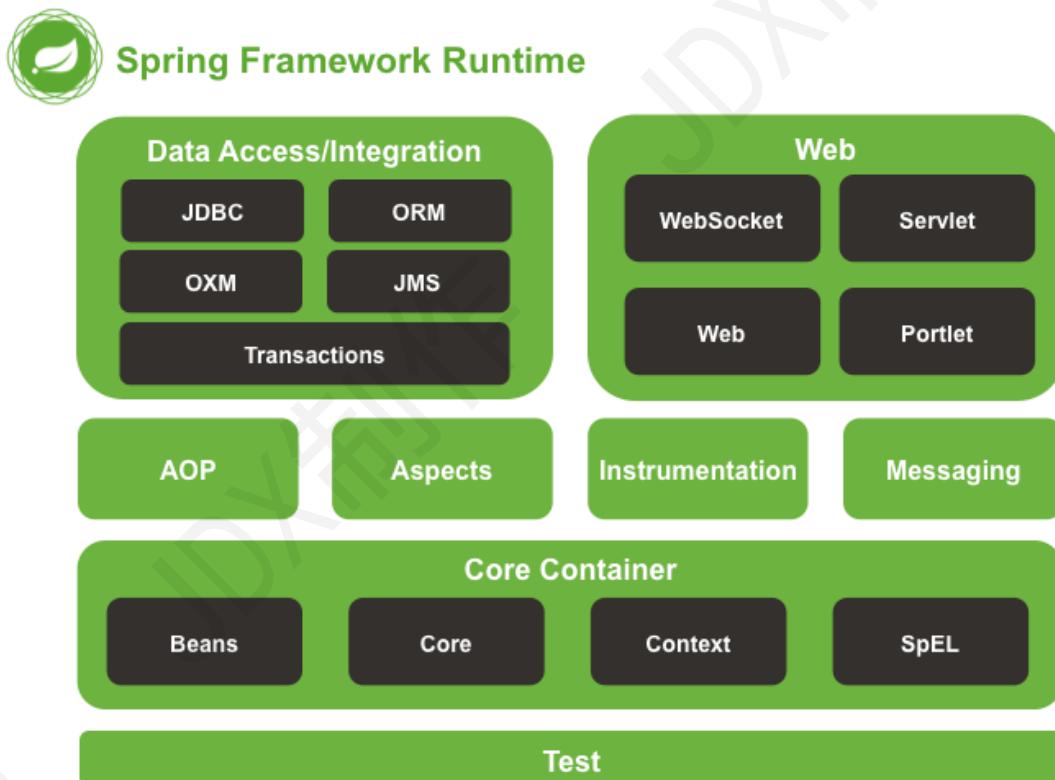
Spring 官网列出的 Spring 的 6 个特征：

- **核心技术**：依赖注入(DI), AOP, 事件(events), 资源, i18n, 验证, 数据绑定, 类型转换, SpEL。
- **测试**：模拟对象, TestContext 框架, Spring MVC 测试, WebTestClient。
- **数据访问**：事务, DAO 支持, JDBC, ORM, 编组 XML。

- **Web支持** : Spring MVC和Spring WebFlux Web框架。
- **集成** : 远程处理, JMS, JCA, JMX, 电子邮件, 任务, 调度, 缓存。
- **语言** : Kotlin, Groovy, 动态语言。

1.2 列举一些重要的Spring模块?

下图对应的是 Spring4.x 版本。目前最新的5.x版本中 Web 模块的 Portlet 组件已经被废弃掉，同时增加了用于异步响应式处理的 WebFlux 组件。

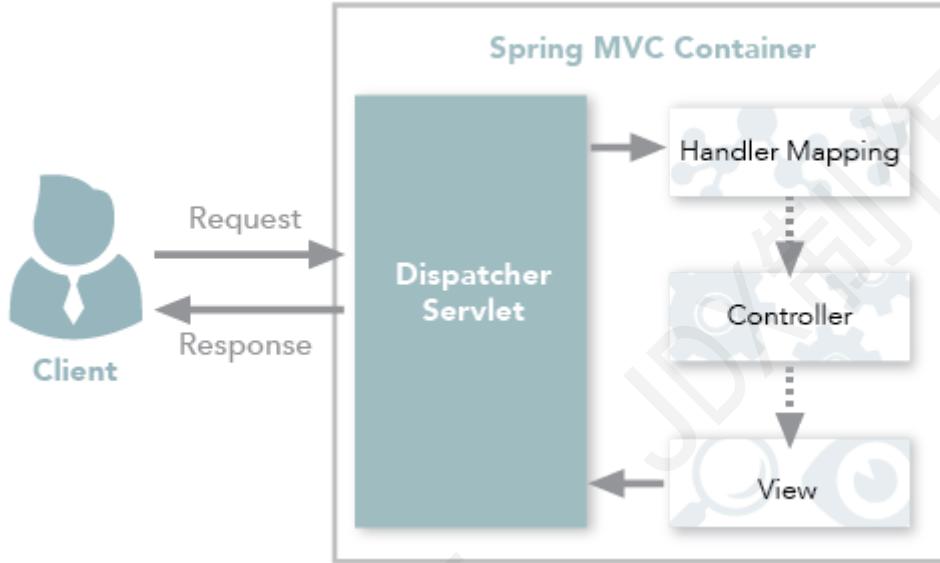


- **Spring Core**: 基础,可以说 Spring 其他所有的功能都需要依赖于该类库。主要提供 IoC 依赖注入功能。
- **Spring Aspects** : 该模块为与AspectJ的集成提供支持。
- **Spring AOP** : 提供了面向切面的编程实现。
- **Spring JDBC** : Java数据库连接。
- **Spring JMS** : Java消息服务。
- **Spring ORM** : 用于支持Hibernate等ORM工具。
- **Spring Web** : 为创建Web应用程序提供支持。
- **Spring Test** : 提供了对 JUnit 和 TestNG 测试的支持。

1.3 @RestController vs @Controller

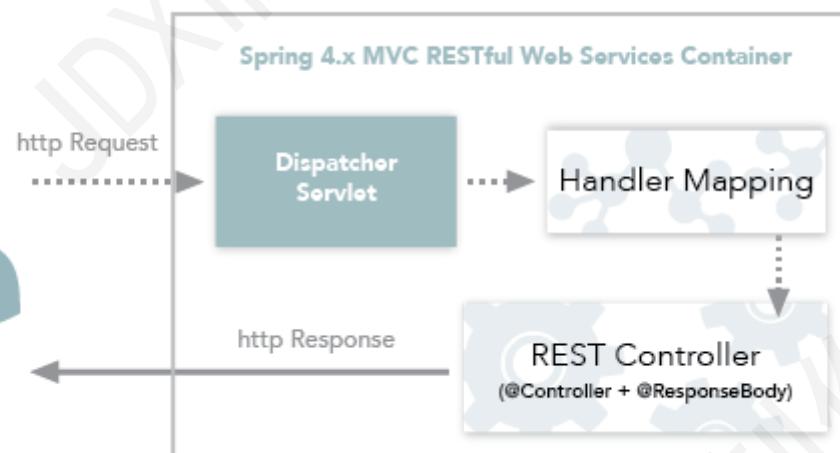
`Controller` 返回一个页面

单独使用 `@Controller` 不加 `@ResponseBody` 的话一般使用在要返回一个视图的情况，这种情况属于比较传统的Spring MVC 的应用，对于前后端不分离的情况。



`@RestController` 返回JSON或XML形式数据

但`@RestController`只返回对象，对象数据直接以JSON或XML形式写入HTTP响应(Response)中，这种情况属于RESTful Web服务，这也是目前日常开发所接触的最常用的情况（前端分离）。



`@Controller +@ResponseBody` 返回JSON或XML形式数据

如果你需要在Spring4之前开发RESTful Web服务的话，你需要使用`@Controller`并结合`@ResponseBody`注解，也就是说`@Controller + @ResponseBody = @RestController`（Spring 4之后新加的注解）。

`@ResponseBody`注解的作用是将`Controller`的方法返回的对象通过适当的转换器转换为指定的格式之后，写入到HTTP响应(Response)对象的body中，通常用来返回JSON或者XML数据，返回JSON数据的情况比较多。

小插曲：

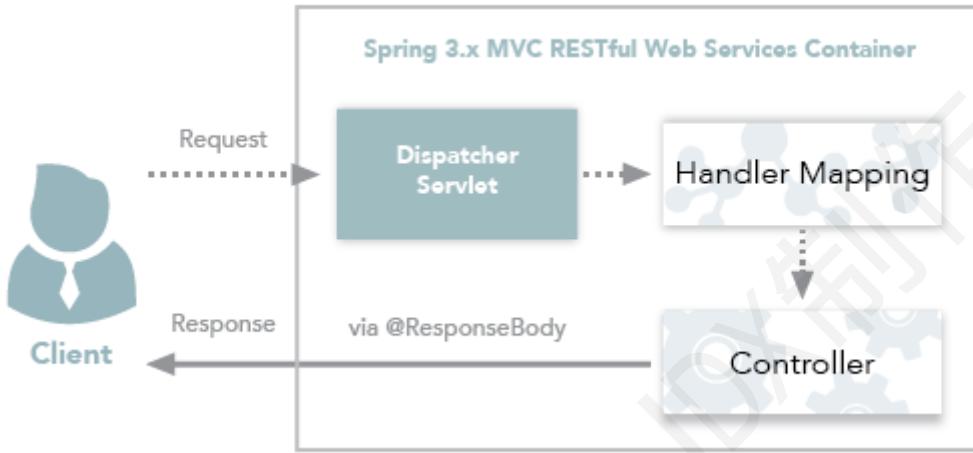
更多阿里、腾讯、美团、京东等一线互联网大厂Java面试真题；包含：基础、并发、锁、JVM、设计模式、数据结构、反射/IO、数据库、Redis、Spring、消息队列、分布式、Zookeeper、Dubbo、Mybatis、Maven、面经等。更多Java程序员技术进阶小技巧；例如高效学习（如何学习和阅读代码、面对枯燥和量大的知识）高效沟通（沟通方式及技巧、沟通技术）

更多Java大牛分享的一些职业生涯分享文档

请添加微信：jiang10086a 免费获取

比你优秀的对手在学习，你的仇人在磨刀，你的闺蜜在减肥，隔壁老王在练腰，我们必须不断学习，否则我们将被学习者超越！

趁年轻，使劲拼，给未来的自己一个交代！



1.4 Spring IOC & AOP

1.4.1 谈谈自己对于 Spring IoC 和 AOP 的理解

①、IoC

IoC (Inverse of Control:控制反转) 是一种设计思想，就是 将原本在程序中手动创建对象的控制权，交由Spring框架来管理。 IoC 在其他语言中也有应用，并非 Spring 特有。 IoC 容器是 Spring 用来实现 IoC 的载体， IoC 容器实际上就是个 Map (key, value) ,Map 中存放的是各种对象。

将对象之间的相互依赖关系交给 IoC 容器来管理，并由 IoC 容器完成对象的注入。这样可以很大程度上简化应用的开发，把应用从复杂的依赖关系中解放出来。 IoC 容器就像是一个工厂一样，当我们需要创建一个对象的时候，只需要配置好配置文件/注解即可，完全不用考虑对象是如何被创建出来的。在实际项目中一个 Service 类可能有几百甚至上千个类作为它的底层，假如我们需要实例化这个 Service，你可能要每次都要搞清这个 Service 所有底层类的构造函数，这可能会把人逼疯。如果利用 IoC 的话，你只需要配置好，然后在需要的地方引用就行了，这大大增加了项目的可维护性且降低了开发难度。

Spring 时代我们一般通过 XML 文件来配置 Bean，后来开发人员觉得 XML 文件来配置不太好，于是 SpringBoot 注解配置就慢慢开始流行起来。

Spring IoC的初始化过程：

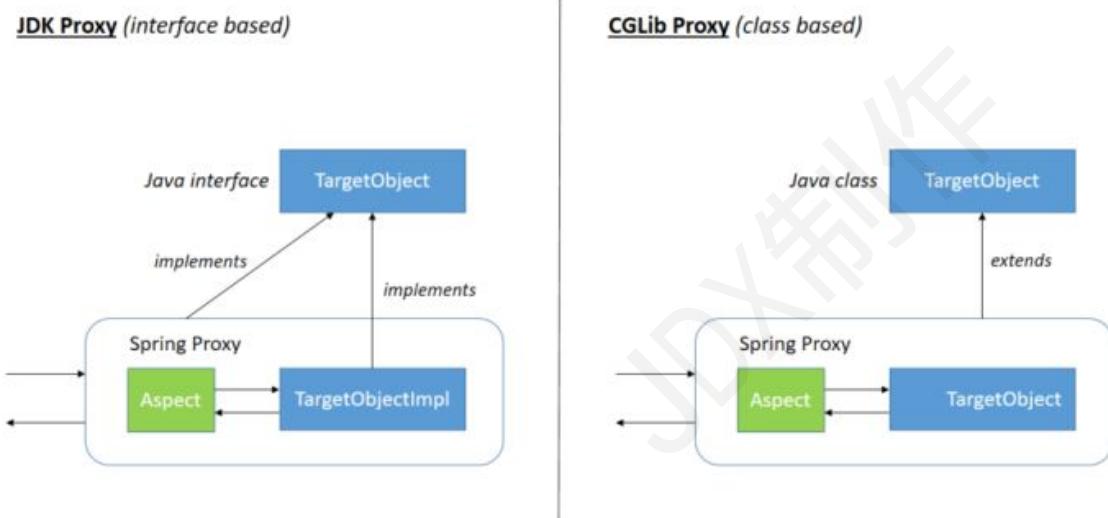


②、AOP

AOP(Aspect-Oriented Programming:面向切面编程)能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可拓展性和可维护性。

Spring AOP就是基于动态代理的，如果要代理的对象，实现了某个接口，那么Spring AOP会使用JDK Proxy，去创建代理对象，而对于没有实现接口的对象，就无法使用 JDK Proxy 去进行代理了，这时候 Spring AOP会使用Cglib，这时候Spring AOP会使用 Cglib 生成一个被代理对象的子类来作为代理，如下图所示：

Spring AOP Process



当然你也可以使用 AspectJ ,Spring AOP 已经集成了AspectJ , AspectJ 应该算的上是 Java 生态系统中最完整的 AOP 框架了。

使用 AOP 之后我们可以把一些通用功能抽象出来，在需要用到的地方直接使用即可，这样大大简化了代码量。我们需要增加新功能时也方便，这样也提高了系统扩展性。日志功能、事务管理等等场景都用到了 AOP 。

1.4.2 Spring AOP 和 AspectJ AOP 有什么区别？

Spring AOP 属于运行时增强，而 AspectJ 是编译时增强。 Spring AOP 基于代理(Proxying)，而 AspectJ 基于字节码操作(Bytecode Manipulation)。

Spring AOP 已经集成了 AspectJ , AspectJ 应该算的上是 Java 生态系统中最完整的 AOP 框架了。AspectJ 相比于 Spring AOP 功能更加强大，但是 Spring AOP 相对来说更简单，

如果我们的切面比较少，那么两者性能差异不大。但是，当切面太多的话，最好选择 AspectJ ，它比 Spring AOP 快很多。

1.5 Spring bean

1.5.1 Spring 中的 bean 的作用域有哪些？

- singleton : 唯一 bean 实例，Spring 中的 bean 默认都是单例的。
- prototype : 每次请求都会创建一个新的 bean 实例。
- request : 每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP request内有效。
- session : 每一次HTTP请求都会产生一个新的 bean，该bean仅在当前 HTTP session 内有效。
- global-session：全局session作用域，仅仅在基于portlet的web应用中才有意义，Spring5已经没有了。Portlet是能够生成语义代码(例如：HTML)片段的小型Java Web插件。它们基于portlet容器，可以像servlet一样处理HTTP请求。但是，与 servlet 不同，每个 portlet 都有不同的会话

1.5.2 Spring 中的单例 bean 的线程安全问题了解吗？

大部分时候我们并没有在系统中使用多线程，所以很少有人会关注这个问题。单例 bean 存在线程问题，主要是因为当多个线程操作同一个对象的时候，对这个对象的非静态成员变量的写操作会存在线程安全问题。

常见的有两种解决办法：

1. 在Bean对象中尽量避免定义可变的成员变量（不太现实）。
2. 在类中定义一个ThreadLocal成员变量，将需要的可变成员变量保存在 ThreadLocal 中（推荐的一种方式）。

1.5.3 @Component 和 @Bean 的区别是什么？

1. 作用对象不同: `@Component` 注解作用于类, 而 `@Bean` 注解作用于方法。
2. `@Component` 通常是通过类路径扫描来自动侦测以及自动装配到Spring容器中(我们可以使用`@ComponentScan`注解定义要扫描的路径从中找出标识了需要装配的类自动装配到 Spring 的 bean 容器中)。`@Bean` 注解通常是在标有该注解的方法中定义产生这个 bean,`@Bean`告诉了Spring这是某个类的示例, 当我需要用它的时候还给我。
3. `@Bean` 注解比 `Component` 注解的自定义性更强, 而且很多地方我们只能通过 `@Bean` 注解来注册 bean。比如当我们引用第三方库中的类需要装配到 `Spring` 容器时, 则只能通过 `@Bean` 来实现。

`@Bean` 注解使用示例:

```
@Configuration  
public class AppConfig {  
    @Bean  
    public TransferService transferService() {  
        return new TransferServiceImpl();  
    }  
}
```

上面的代码相当于下面的 xml 配置

```
<beans>  
    <bean id="transferService" class="com.acme.TransferServiceImpl"/>  
</beans>
```

下面这个例子是通过 `@Component` 无法实现的。

```
@Bean  
public OneService getService(status) {  
    case (status) {  
        when 1:  
            return new serviceImpl1();  
        when 2:  
            return new serviceImpl2();  
        when 3:  
            return new serviceImpl3();  
    }  
}
```

1.5.4 将一个类声明为Spring的 bean 的注解有哪些?

我们一般使用 `@Autowired` 注解自动装配 bean, 要想把类标识成可用于 `@Autowired` 注解自动装配的 bean 的类, 采用以下注解可实现:

- `@Component` : 通用的注解, 可标注任意类为 `Spring` 组件。如果一个 Bean 不知道属于哪个层, 可以使用 `@Component` 注解标注。
- `@Repository` : 对应持久层即 Dao 层, 主要用于数据库相关操作。
- `@Service` : 对应服务层, 主要涉及一些复杂的逻辑, 需要用到 Dao 层。
- `@Controller` : 对应 Spring MVC 控制层, 主要用户接受用户请求并调用 Service 层返回数据给前端页面。

1.5.5 5.5 Spring 中的 bean 生命周期?

- Bean 容器找到配置文件中 Spring Bean 的定义。
- Bean 容器利用 Java Reflection API 创建一个Bean的实例。
- 如果涉及到一些属性值利用 `set()` 方法设置一些属性值。
- 如果 Bean 实现了 `BeanNameAware` 接口，调用 `setBeanName()` 方法，传入Bean的名字。
- 如果 Bean 实现了 `BeanClassLoaderAware` 接口，调用 `setBeanClassLoader()` 方法，传入 `ClassLoader` 对象的实例。
- 与上面的类似，如果实现了其他 `*.Aware` 接口，就调用相应的方法。
- 如果有和加载这个 Bean 的 Spring 容器相关的 `BeanPostProcessor` 对象，执行 `postProcessBeforeInitialization()` 方法
- 如果Bean实现了 `InitializingBean` 接口，执行 `afterPropertiesSet()` 方法。
- 如果 Bean 在配置文件中的定义包含 `init-method` 属性，执行指定的方法。
- 如果有和加载这个 Bean 的 Spring 容器相关的 `BeanPostProcessor` 对象，执行 `postProcessAfterInitialization()` 方法
- 当要销毁 Bean 的时候，如果 Bean 实现了 `DisposableBean` 接口，执行 `destroy()` 方法。
- 当要销毁 Bean 的时候，如果 Bean 在配置文件中的定义包含 `destroy-method` 属性，执行指定的方法。

图示：

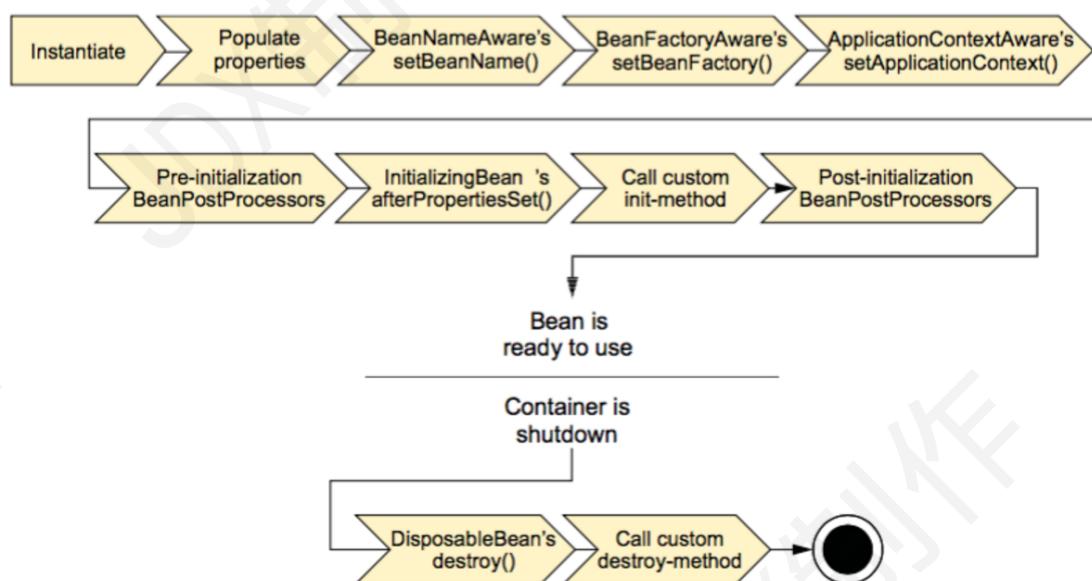
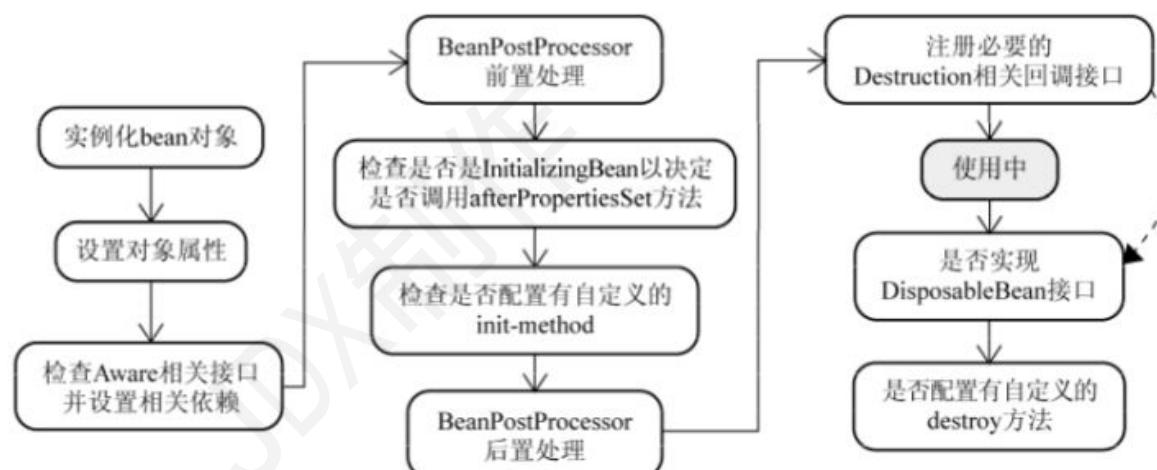


Figure 1.5 A bean goes through several steps between creation and destruction in the Spring container. Each step is an opportunity to customize how the bean is managed in Spring.

与之比较类似的中文版本：



1.6 Spring MVC

1.6.1 说说自己对于 Spring MVC 了解?

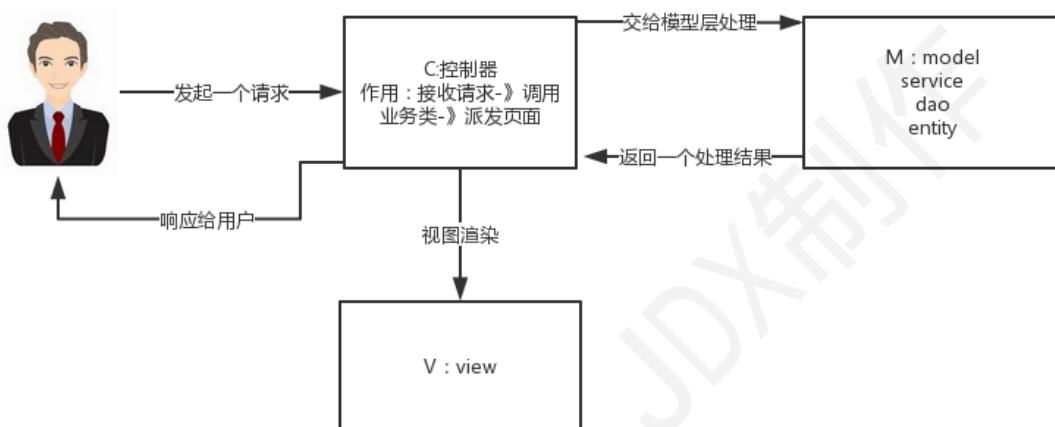
谈到这个问题，我们不得不提之前 Model1 和 Model2 这两个没有 Spring MVC 的时代。

- **Model1 时代**：很多学 Java 后端比较晚的朋友可能并没有接触过 Model1 模式下的 JavaWeb 应用开发。在 Model1 模式下，整个 Web 应用几乎全部用 JSP 页面组成，只用少量的 JavaBean 来处理数据库连接、访问等操作。这个模式下 JSP 即是控制层又是表现层。显而易见，这种模式存在很多问题。比如①将控制逻辑和表现逻辑混杂在一起，导致代码重用率极低；②前端和后端相互依赖，难以进行测试并且开发效率极低；
- **Model2 时代**：学过 Servlet 并做过相关 Demo 的朋友应该了解“Java Bean(Model)+JSP (View,) +Servlet (Controller) ”这种开发模式，这就是早期的 JavaWeb MVC 开发模式。
Model:系统涉及的数据，也就是 dao 和 bean。View：展示模型中的数据，只是用来展示。
Controller：处理用户请求都发送给，返回数据给 JSP 并展示给用户。

Model2 模式下还存在很多问题，Model2 的抽象和封装程度还远远不够，使用 Model2 进行开发时不可避免地会重复造轮子，这就大大降低了程序的可维护性和复用性。于是很多 JavaWeb 开发相关的 MVC 框架应运而生比如 Struts2，但是 Struts2 比较笨重。随着 Spring 轻量级开发框架的流行，Spring 生态圈出现了 Spring MVC 框架，Spring MVC 是当前最优秀的 MVC 框架。相比于 Struts2，Spring MVC 使用更加简单和方便，开发效率更高，并且 Spring MVC 运行速度更快。

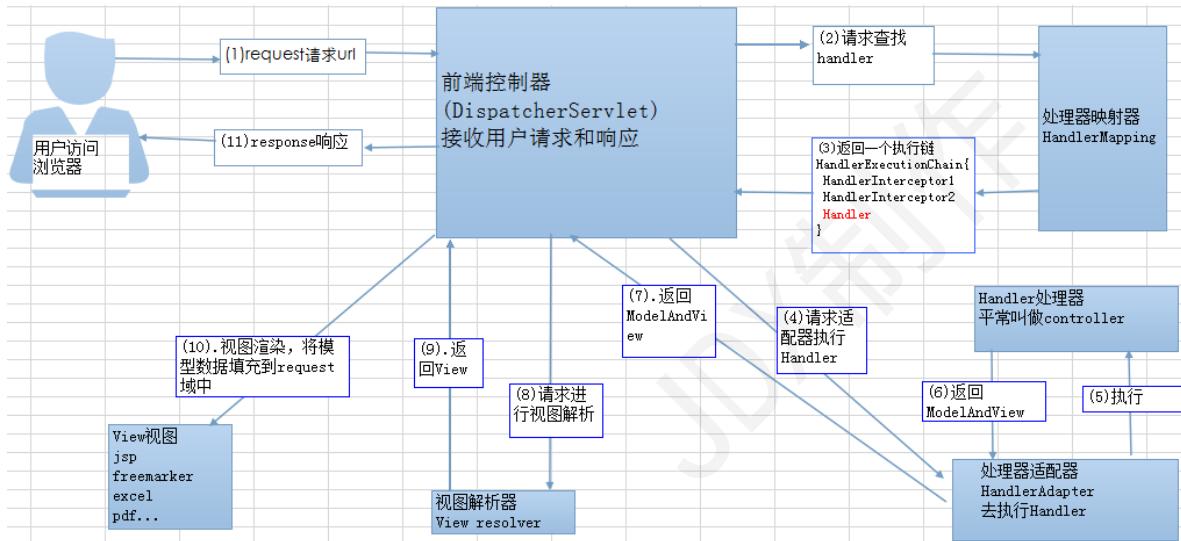
MVC 是一种设计模式，Spring MVC 是一款很优秀的 MVC 框架。Spring MVC 可以帮助我们进行更简洁的 Web 层的开发，并且它天生与 Spring 框架集成。Spring MVC 下我们一般把后端项目分为 Service 层（处理业务）、Dao 层（数据库操作）、Entity 层（实体类）、Controller 层（控制层，返回数据给前台页面）。

Spring MVC 的简单原理图如下：



1.6.2 SpringMVC 工作原理了解吗？

原理如下图所示：



上图的一个笔误的小问题：Spring MVC 的入口函数也就是前端控制器 DispatcherServlet 的作用是接收请求，响应结果。

流程说明（重要）：

1. 客户端（浏览器）发送请求，直接请求到 DispatcherServlet。
2. DispatcherServlet 根据请求信息调用 HandlerMapping，解析请求对应的 Handler。
3. 解析到对应的 Handler（也就是我们平常说的 Controller 控制器）后，开始由 HandlerAdapter 适配器处理。
4. HandlerAdapter 会根据 Handler 来调用真正的处理器来处理请求，并处理相应的业务逻辑。
5. 处理器处理完业务后，会返回一个 ModelAndView 对象，Model 是返回的数据对象，View 是个逻辑上的 view。
6. ViewResolver 会根据逻辑 View 查找实际的 View。
7. DispatcherServlet 把返回的 Model 传给 view（视图渲染）。
8. 把 view 返回给请求者（浏览器）

1.7 Spring 框架中用到了哪些设计模式？

- **工厂设计模式**：Spring 使用工厂模式通过 BeanFactory、ApplicationContext 创建 bean 对象。
- **代理设计模式**：Spring AOP 功能的实现。
- **单例设计模式**：Spring 中的 Bean 默认都是单例的。
- **模板方法模式**：Spring 中 jdbcTemplate、hibernateTemplate 等以 Template 结尾的对数据库操作的类，它们就使用到了模板模式。
- **包装器设计模式**：我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要会去访问不同的数据库。这种模式让我们可以根据客户的需求能够动态切换不同的数据源。
- **观察者模式**：Spring 事件驱动模型就是观察者模式很经典的一个应用。
- **适配器模式**：Spring AOP 的增强或通知(Advice)使用到了适配器模式、spring MVC 中也是用到了适配器模式适配 Controller。
-

1.8 Spring 事务

1.8.1 Spring 管理事务的方式有几种？

1. 编程式事务，在代码中硬编码。（不推荐使用）
2. 声明式事务，在配置文件中配置（推荐使用）

声明式事务又分为两种：

1. 基于XML的声明式事务
2. 基于注解的声明式事务

1.8.2 Spring 事务中的隔离级别有哪几种？

TransactionDefinition 接口中定义了五个表示隔离级别的常量：

- **TransactionDefinition.ISOLATION_DEFAULT:** 使用后端数据库默认的隔离级别，Mysql 默认采用的 REPEATABLE_READ 隔离级别 Oracle 默认采用的 READ_COMMITTED 隔离级别。
- **TransactionDefinition.ISOLATION_READ_UNCOMMITTED:** 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读。
- **TransactionDefinition.ISOLATION_READ_COMMITTED:** 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生。
- **TransactionDefinition.ISOLATION_REPEATABLE_READ:** 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。
- **TransactionDefinition.ISOLATION_SERIALIZABLE:** 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

1.8.3 Spring 事务中哪几种事务传播行为？

支持当前事务的情况：

- **TransactionDefinition.PROPAGATION_REQUIRED:** 如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。
- **TransactionDefinition.PROPAGATION_SUPPORTS:** 如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。
- **TransactionDefinition.PROPAGATION_MANDATORY:** 如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。（mandatory：强制性）

不支持当前事务的情况：

- **TransactionDefinition.PROPAGATIONQUIRES_NEW:** 创建一个新的事务，如果当前存在事务，则把当前事务挂起。
- **TransactionDefinition.PROPAGATION_NOT_SUPPORTED:** 以非事务方式运行，如果当前存在事务，则把当前事务挂起。
- **TransactionDefinition.PROPAGATION_NEVER:** 以非事务方式运行，如果当前存在事务，则抛出异常。

其他情况：

- **TransactionDefinition.PROPAGATION_NESTED:** 如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于 TransactionDefinition.PROPAGATION_REQUIRED。

1.8.4 @Transactional(rollbackFor = Exception.class)注解了解吗？

我们知道：Exception分为运行时异常RuntimeException和非运行时异常。事务管理对于企业应用来说是至关重要的，即使出现异常情况，它也可以保证数据的一致性。

当 @Transactional 注解作用于类上时，该类的所有 public 方法将都具有该类型的事务属性，同时，我们也可以在方法级别使用该标注来覆盖类级别的定义。如果类或者方法加了这个注解，那么这个类里面的方法抛出异常，就会回滚，数据库里面的数据也会回滚。

在 @Transactional 注解中如果不配置 rollbackFor 属性，那么事物只会在遇到 RuntimeException 的时候才会回滚，加上 rollbackFor=Exception.class，可以让事物在遇到非运行时异常时也回滚。

1.9 JPA

1.9.1 如何使用JPA在数据库中非持久化一个字段?

假如我们有下面一个类:

```
Entity(name="USER")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "ID")
    private Long id;

    @Column(name="USER_NAME")
    private String userName;

    @Column(name="PASSWORD")
    private String password;

    private String secrect;

}
```

如果我们想让 `secrect` 这个字段不被持久化, 也就是不被数据库存储怎么办? 我们可以采用下面几种方法:

```
static String transient1; // not persistent because of static
final String transient2 = "Satish"; // not persistent because of final
transient String transient3; // not persistent because of transient
@Transient
String transient4; // not persistent because of @Transient
```

一般使用后面两种方式比较多, 我个人使用注解的方式比较多。

2. Spring常用注解

2.1 @SpringBootApplication

这里先单独拎出 `@SpringBootApplication` 注解说一下, 虽然我们一般不会主动去使用它。

Guide 哥: 这个注解是 Spring Boot 项目的基石, 创建 SpringBoot 项目之后会默认在主类加上。

```
@SpringBootApplication
public class SpringSecurityJwtGuideApplication {
    public static void main(java.lang.String[] args) {
        SpringApplication.run(SpringSecurityJwtGuideApplication.class, args);
    }
}
```

我们可以把 `@SpringBootApplication` 看作是 `@Configuration`、`@EnableAutoConfiguration`、`@ComponentScan` 注解的集合。

```
package org.springframework.boot.autoconfigure;
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
```

```

@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
    @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
    @Filter(type = FilterType.CUSTOM, classes =
AutoConfigurationExcludeFilter.class)
})
public @interface SpringBootApplication {
    ...
}

package org.springframework.boot;
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {
}

```

根据 SpringBoot 官网，这三个注解的作用分别是：

- `@EnableAutoConfiguration`：启用 SpringBoot 的自动配置机制
- `@ComponentScan`：扫描被 `@Component` (`@Service`,`@Controller`)注解的 bean，注解默认会扫描该类所在的包下所有的类。
- `@Configuration`：允许在 Spring 上下文中注册额外的 bean 或导入其他配置类

2.2 Spring Bean 相关

2.2.1 `@Autowired`

自动导入对象到类中，被注入进的类同样要被 Spring 容器管理比如：Service 类注入到 Controller 类中。

```

@Service
public class UserService {
    ...
}

@RestController
@RequestMapping("/users")
public class UserController {
    @Autowired
    private UserService userService;
    ...
}

```

2.2.2 `Component`,`@Repository`,`@Service`,`@Controller`

我们一般使用 `@Autowired` 注解让 Spring 容器帮我们自动装配 bean。要想把类标识成可用于 `@Autowired` 注解自动装配的 bean 的类，可以采用以下注解实现：

- `@Component`：通用的注解，可标注任意类为 Spring 组件。如果一个 Bean 不知道属于哪个层，可以使用 `@Component` 注解标注。
- `@Repository`：对应持久层即 Dao 层，主要用于数据库相关操作。
- `@Service`：对应服务层，主要涉及一些复杂的逻辑，需要用到 Dao 层。

- `@Controller` : 对应 Spring MVC 控制层，主要用户接受用户请求并调用 Service 层返回数据给前端页面。

2.2.3 `@RestController`

`@RestController` 注解是 `@Controller` 和 `@ResponseBody` 的合集，表示这是个控制器 bean，并且是将函数的返回值直接填入 HTTP 响应体中，是 REST 风格的控制器。

Guide 哥：现在都是前后端分离，说实话我已经很久没有用过 `@Controller`。如果你的项目太老了的话，就当我没说。

单独使用 `@Controller` 不加 `@ResponseBody` 的话一般使用在要返回一个视图的情况，这种情况属于比较传统的 Spring MVC 的应用，对应于前后端不分离的情况。`@Controller + @ResponseBody` 返回 JSON 或 XML 形式数据

2.2.4 `@Scope`

声明 Spring Bean 的作用域，使用方法：

```
@Bean  
@Scope("singleton")  
public Person personSingleton() {  
    return new Person();  
}
```

四种常见的 Spring Bean 的作用域：

- `singleton` : 唯一 bean 实例，Spring 中的 bean 默认都是单例的。
- `prototype` : 每次请求都会创建一个新的 bean 实例。
- `request` : 每一次 HTTP 请求都会产生一个新的 bean，该 bean 仅在当前 HTTP request 内有效。
- `session` : 每一次 HTTP 请求都会产生一个新的 bean，该 bean 仅在当前 HTTP session 内有效。

2.2.5 `Configuration`

一般用来声明配置类，可以使用 `@Component` 注解替代，不过使用 `Configuration` 注解声明配置类更加语义化。

```
@Configuration  
public class AppConfig {  
    @Bean  
    public TransferService transferService() {  
        return new TransferServiceImpl();  
    }  
}
```

2.3 处理常见的 HTTP 请求类型

5 种常见的请求类型：

- **GET** : 请求从服务器获取特定资源。举个例子：`GET /users` (获取所有学生)
- **POST** : 在服务器上创建一个新的资源。举个例子：`POST /users` (创建学生)
- **PUT** : 更新服务器上的资源 (客户端提供更新后的整个资源)。举个例子：`PUT /users/12` (更新编号为 12 的学生)
- **DELETE** : 从服务器删除特定的资源。举个例子：`DELETE /users/12` (删除编号为 12 的学生)

- **PATCH**：更新服务器上的资源（客户端提供更改的属性，可以看做是部分更新），使用的比较少，这里就不举例子了。

2.3.1 GET 请求

`@GetMapping("users")` 等价于 `@RequestMapping(value="/users",method=RequestMethod.GET)`

```
@GetMapping("/users")
public ResponseEntity<List<User>> getAllUsers() {
    return userRepository.findAll();
}
```

2.3.2 POST 请求

`@PostMapping("users")` 等价于

`@RequestMapping(value="/users",method=RequestMethod.POST)`

关于 `@RequestBody` 注解的使用，在下面的“前后端传值”这块会讲到。

```
@PostMapping("/users")
public ResponseEntity<User> createUser(@Valid @RequestBody UserCreateRequest
userCreateRequest) {
    return userRepository.save(user);
}
```

2.3.3 PUT 请求

`@PutMapping("/users/{userId}")` 等价于

`@RequestMapping(value="/users/{userId}",method=RequestMethod.PUT)`

```
@PutMapping("/users/{userId}")
public ResponseEntity<User> updateUser(@PathVariable(value = "userId") Long
userId,
    @Valid @RequestBody UserUpdateRequest userUpdateRequest) {
    .....
}
```

2.3.4 DELETE 请求

`@DeleteMapping("/users/{userId}")` 等价于

`@RequestMapping(value="/users/{userId}",method=RequestMethod.DELETE)`

```
@DeleteMapping("/users/{userId}")
public ResponseEntity deleteUser(@PathVariable(value = "userId") Long userId){
    .....
}
```

2.3.5 PATCH 请求

一般实际项目中，我们都是 PUT 不够用了之后才用 PATCH 请求去更新数据。

```
@PatchMapping("/profile")
public ResponseEntity updateStudent(@RequestBody StudentUpdateRequest
studentUpdateRequest) {
    studentRepository.updateDetail(studentUpdateRequest);
    return ResponseEntity.ok().build();
}
```

2.4 前后端传值

掌握前后端传值的正确姿势，是你开始 CRUD 的第一步！

2.4.1 @PathVariable 和 @RequestParam

`@PathVariable` 用于获取路径参数，`@RequestParam` 用于获取查询参数。

举个简单的例子：

```
@GetMapping("/klasses/{klassId}/teachers")
public List<Teacher> getKlassRelatedTeachers(
    @PathVariable("klassId") Long klassId,
    @RequestParam(value = "type", required = false) String type) {
    ...
}
```

如果我们请求的 url 是： `/klasses/{123456}/teachers?type=web`

那么我们服务获取到的数据就是：`klassId=123456, type=web`。

2.4.2 @RequestBody

用于读取 Request 请求（可能是 POST,PUT,DELETE,GET 请求）的 body 部分并且 Content-Type 为 `application/json` 格式的数据，接收到数据之后会自动将数据绑定到 Java 对象上去。系统会使用 `HttpMessageConverter` 或者自定义的 `HttpMessageConverter` 将请求的 body 中的 json 字符串转换为 java 对象。

我用一个简单的例子来给演示一下基本使用！

我们有一个注册的接口：

```
@PostMapping("/sign-up")
public ResponseEntity signUp(@RequestBody @Valid UserRegisterRequest
userRegisterRequest) {
    userService.save(userRegisterRequest);
    return ResponseEntity.ok().build();
}
```

`UserRegisterRequest` 对象：

```
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
public class UserRegisterRequest {  
    @NotBlank  
    private String userName;  
    @NotBlank  
    private String password;  
    @FullName  
    @NotBlank  
    private String fullName;  
}
```

我们发送 post 请求到这个接口，并且 body 携带 JSON 数据：

```
{"userName": "coder", "fullName": "shuangkou", "password": "123456"}
```

这样我们的后端就可以直接把 json 格式的数据映射到我们的 `UserRegisterRequest` 类上。



需要注意的是：一个请求方法只可以有一个 `@RequestBody`，但是可以有多个 `@RequestParam` 和 `@PathVariable`。如果你的方法必须要用两个 `@RequestBody` 来接受数据的话，大概率是你的数据库设计或者系统设计出问题了！

2.5 读取配置信息

很多时候我们需要将一些常用的配置信息比如阿里云 oss、发送短信、微信认证的相关配置信息等等放到配置文件中。

下面我们来看一下 Spring 为我们提供了哪些方式帮助我们从配置文件中读取这些配置信息。

我们的数据源 `application.yml` 内容如下：

```
wuhan2020: 2020年初武汉爆发了新型冠状病毒，疫情严重，但是，我相信一切都会过去！武汉加油！中国加油！

my-profile:  
  name: Guide哥  
  email: koushuangbwcx@163.com

library:  
  location: 湖北武汉加油中国加油  
  books:  
    - name: 天才基本法  
      description: 二十二岁的林朝夕在父亲确诊阿尔茨海默病这天，得知自己暗恋多年的校园男神裴之即将出国深造的消息——对方考取的学校，恰是父亲当年为她放弃的那所。  
    - name: 时间的秩序
```

description: 为什么我们记得过去，而非未来？时间“流逝”意味着什么？是我们存在于时间之内，还是时间存在于我们之中？卡洛·罗韦利用诗意的文字，邀请我们思考这一亘古难题——时间的本质。

- **name:** 了不起的我

description: 如何养成一个新习惯？如何让心智变得更成熟？如何拥有高质量的关系？如何走出人生的艰难时刻？

2.5.1 @value(常用)

使用 `@Value("${property}")` 读取比较简单的配置信息：

```
@Value("${wuhan2020}")
String wuhan2020;
```

2.5.2 @ConfigurationProperties(常用)

通过 `@ConfigurationProperties` 读取配置信息并与 bean 绑定。

```
@Component
@ConfigurationProperties(prefix = "library")
class LibraryProperties {
    @NotEmpty
    private String location;
    private List<Book> books;

    @Setter
    @Getter
    @ToString
    static class Book {
        String name;
        String description;
    }
    省略getter/setter
    .....
}
```

你可以像使用普通的 Spring bean 一样，将其注入到类中使用。

2.5.3 PropertySource (不常用)

`@PropertySource` 读取指定 properties 文件

```
@Component
@PropertySource("classpath:website.properties")

class Website {
    @Value("${url}")
    private String url;

    省略getter/setter
    .....
}
```

2.6 参数校验

数据的校验的重要性就不用说了，即使在前端对数据进行校验的情况下，我们还是要对传入后端的数据再进行一遍校验，避免用户绕过浏览器直接通过一些 HTTP 工具直接向后端请求一些违法数据。

JSR(Java Specification Requests) 是一套 JavaBean 参数校验的标准，它定义了很多常用的校验注解，我们可以直接将这些注解加在我们 JavaBean 的属性上面，这样就可以在需要校验的时候进行校验了，非常方便！

校验的时候我们实际用的是 **Hibernate Validator** 框架。Hibernate Validator 是 Hibernate 团队最初的数据校验框架，Hibernate Validator 4.x 是 Bean Validation 1.0 (JSR 303) 的参考实现，Hibernate Validator 5.x 是 Bean Validation 1.1 (JSR 349) 的参考实现，目前最新版的 Hibernate Validator 6.x 是 Bean Validation 2.0 (JSR 380) 的参考实现。

SpringBoot 项目的 spring-boot-starter-web 依赖中已经有 hibernate-validator 包，不需要引用相关依赖。如下图所示（通过 idea 插件—Maven Helper 生成）：

```
▼ spring-boot-starter-web : 2.1.8.RELEASE [compile]
  ▼ hibernate-validator : 6.0.17.Final [compile]
    classmate : 1.4.0 [compile]
    jboss-logging : 3.3.3.Final [compile]
    validation-api : 2.0.1.Final [compile]
    spring-boot-starter : 2.1.8.RELEASE [compile]
  ▼ spring-boot-starter-json : 2.1.8.RELEASE [compile]
    jackson-databind : 2.9.9.3 [compile]
  ▼ jackson-datatype-jdk8 : 2.9.9 [compile]
    jackson-core : 2.9.9 [compile]
    jackson-databind : 2.9.9.3 [compile]
  ▼ jackson-datatype-jsr310 : 2.9.9 [compile]
    jackson-annotations : 2.9.0 [compile]
    jackson-core : 2.9.9 [compile]
    jackson-databind : 2.9.9.3 [compile]
  ▼ jackson-module-parameter-names : 2.9.9 [compile]
    jackson-core : 2.9.9 [compile]
    jackson-databind : 2.9.9.3 [compile]
    spring-boot-starter : 2.1.8.RELEASE [compile]
    spring-web : 5.1.9.RELEASE [compile]
  ▼ spring-boot-starter-tomcat : 2.1.8.RELEASE [compile]
    javax.annotation-api : 1.3.2 [compile]
    tomcat-embed-core : 9.0.24 [compile]
```

需要注意的是：所有的注解，推荐使用 JSR 注解，即 `javax.validation.constraints`，而不是 `org.hibernate.validator.constraints`

2.6.1 一些常用的字段验证的注解

- `@NotEmpty` 被注释的字符串的不能为 null 也不能为空
- `@NotBlank` 被注释的字符串非 null，并且必须包含一个非空白字符
- `@Null` 被注释的元素必须为 null
- `@NotNull` 被注释的元素必须不为 null
- `@AssertTrue` 被注释的元素必须为 true
- `@AssertFalse` 被注释的元素必须为 false
- `@Pattern(regex=, flag=)` 被注释的元素必须符合指定的正则表达式
- `@Email` 被注释的元素必须是 Email 格式。
- `@Min(value)` 被注释的元素必须是一个数字，其值必须大于等于指定的最小值
- `@Max(value)` 被注释的元素必须是一个数字，其值必须小于等于指定的最大值
- `@DecimalMin(value)` 被注释的元素必须是一个数字，其值必须大于等于指定的最小值

- `@DecimalMax(value)` 被注释的元素必须是一个数字，其值必须小于等于指定的最大值
- `@Size(max=, min=)` 被注释的元素的大小必须在指定的范围内
- `@Digits(integer, fraction)` 被注释的元素必须是一个数字，其值必须在可接受的范围内
- `@Past` 被注释的元素必须是一个过去的日期
- `@Future` 被注释的元素必须是一个将来的日期
-

2.6.2 验证请求体(RequestBody)

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Person {

    @NotNull(message = "classId 不能为空")
    private String classId;

    @Size(max = 33)
    @NotNull(message = "name 不能为空")
    private String name;

    @Pattern(regexp = "((^Man$|^Woman$|^UGM$))", message = "sex 值不在可选范围")
    @NotNull(message = "sex 不能为空")
    private String sex;

    @Email(message = "email 格式不正确")
    @NotNull(message = "email 不能为空")
    private String email;

}
```

我们在需要验证的参数上加上了`@Valid`注解，如果验证失败，它将抛出

`MethodArgumentNotValidException`。

```
@RestController
@RequestMapping("/api")
public class PersonController {

    @PostMapping("/person")
    public ResponseEntity<Person> getPerson(@RequestBody @Valid Person person) {
        return ResponseEntity.ok().body(person);
    }
}
```

2.6.3 验证请求参数(Path Variables 和 Request Parameters)

一定一定不要忘记在类上加上`validated`注解了，这个参数可以告诉Spring去校验方法参数。

```
@RestController
@RequestMapping("/api")
@Validated
public class PersonController {

    @GetMapping("/person/{id}")
    public ResponseEntity<Integer> getPersonByID(@Valid @PathVariable("id")
        @Max(value = 5,message = "超过 id 的范围了") Integer id) {
        return ResponseEntity.ok().body(id);
    }
}
```

2.7 全局处理 Controller 层异常

介绍一下我们 Spring 项目必备的全局处理 Controller 层异常。

相关注解：

1. `@ControllerAdvice` :注解定义全局异常处理类
2. `@ExceptionHandler` :注解声明异常处理方法

如何使用呢？拿我们在第 5 节参数校验这块来举例子。如果方法参数不对的话就会抛出 `MethodArgumentNotValidException`，我们来处理这个异常。

```
@ControllerAdvice
@ResponseBody
public class GlobalExceptionHandler {

    /**
     * 请求参数异常处理
     */
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<?>
    handleMethodArgumentNotValidException(MethodArgumentNotValidException ex,
    HttpServletRequest request) {
        .....
    }
}
```

2.8 JPA 相关

2.8.1 创建表

`@Entity` 声明一个类对应一个数据库实体。

`@Table` 设置表明

```
@Entity
@Table(name = "role")
public class Role {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String description;
    省略getter/setter.....
}
```

2.8.2 创建主键

`@Id` : 声明一个字段为主键。

使用`@Id`声明之后，我们还需要定义主键的生成策略。我们可以使用`@GeneratedValue`指定主键生成策略。

1.通过`@GeneratedValue`直接使用JPA内置提供的四种主键生成策略来指定主键生成策略。

```
@Id  
@GeneratedValue(strategy = GenerationType.IDENTITY)  
private Long id;
```

JPA 使用枚举定义了 4 中常见的主键生成策略，如下：

Guide 哥：枚举替代常量的一种用法

```
public enum GenerationType {  
  
    /**  
     * 使用一个特定的数据库表格来保存主键  
     * 持久化引擎通过关系数据库的一张特定的表格来生成主键，  
     */  
    TABLE,  
  
    /**  
     * 在某些数据库中，不支持主键自增长，比如oracle、PostgreSQL其提供了一种叫做"序列  
     * (sequence)"的机制生成主键  
     */  
    SEQUENCE,  
  
    /**  
     * 主键自增长  
     */  
    IDENTITY,  
  
    /**  
     * 把主键生成策略交给持久化引擎(persistence engine)，  
     * 持久化引擎会根据数据库在以上三种主键生成 策略中选择其中一种  
     */  
    AUTO  
}
```

`@GeneratedValue`注解默认使用的策略是`GenerationType.AUTO`

```
public @interface GeneratedValue {  
  
    GenerationType strategy() default AUTO;  
    String generator() default "";  
}
```

一般使用MySQL数据库的话，使用`GenerationType.IDENTITY`策略比较普遍一点（分布式系统的话需要另外考虑使用分布式ID）。

2.通过`@GenericGenerator`声明一个主键策略，然后`@GeneratedValue`使用这个策略

```
@Id  
@GeneratedValue(generator = "IdentityIdGenerator")  
@GenericGenerator(name = "IdentityIdGenerator", strategy = "identity")  
private Long id;
```

等价于：

```
@Id  
@GeneratedValue(strategy = GenerationType.IDENTITY)  
private Long id;
```

jpa 提供的主键生成策略有如下几种：

```
public class DefaultIdentifierGeneratorFactory  
    implements MutableIdentifierGeneratorFactory, Serializable,  
ServiceRegistryAwareService {  
  
    @SuppressWarnings("deprecation")  
    public DefaultIdentifierGeneratorFactory() {  
        register( "uuid2", UUIDGenerator.class );  
        register( "guid", GUIDGenerator.class );  
        // can be done with  
        UUIDGenerator + strategy  
        register( "uuid", UUIDHexGenerator.class );  
        // "deprecated" for  
        new use  
        register( "uuid.hex", UUIDHexGenerator.class );  
        // uuid.hex is  
        deprecated  
        register( "assigned", Assigned.class );  
        register( "identity", IdentityGenerator.class );  
        register( "select", SelectGenerator.class );  
        register( "sequence", SequenceStyleGenerator.class );  
        register( "seqhilo", SequenceHiLoGenerator.class );  
        register( "increment", IncrementGenerator.class );  
        register( "foreign", ForeignGenerator.class );  
        register( "sequence-identity", SequenceIdentityGenerator.class );  
        register( "enhanced-sequence", SequenceStyleGenerator.class );  
        register( "enhanced-table", TableGenerator.class );  
    }  
  
    public void register(String strategy, Class generatorClass) {  
        LOG.debugf( "Registering IdentifierGenerator strategy [%s] -> [%s]",  
        strategy, generatorClass.getName() );  
        final Class previous = generatorStrategyToClassNameMap.put( strategy,  
        generatorClass );  
        if ( previous != null ) {  
            LOG.debugf( "    - overriding [%s]", previous.getName() );  
        }  
    }  
}
```

2.8.3 设置字段类型

`@Column` 声明字段。

示例：

设置属性 `userName` 对应的数据库字段名为 `user_name`, 长度为 32, 非空

```
@Column(name = "user_name", nullable = false, length=32)  
private String userName;
```

设置字段类型并且加默认值, 这个还是挺常用的。

```
Column(columnDefinition = "tinyint(1) default 1")  
private Boolean enabled;
```

2.8.4 指定不持久化特定字段

`@Transient` : 声明不需要与数据库映射的字段, 在保存的时候不需要保存进数据库。

如果我们想让 `secrect` 这个字段不被持久化, 可以使用 `@Transient` 关键字声明。

```
Entity(name="USER")  
public class User {  
  
    ....  
    @Transient  
    private String secrect; // not persistent because of @Transient  
  
}
```

除了 `@Transient` 关键字声明, 还可以采用下面几种方法:

```
static String secrect; // not persistent because of static  
final String secrect = "Satish"; // not persistent because of final  
transient String secrect; // not persistent because of transient
```

一般使用注解的方式比较多。

2.8.5 声明大字段

`@Lob`: 声明某个字段为大字段。

```
@Lob  
private String content;
```

更详细的声明:

```
@Lob  
//指定 Lob 类型数据的获取策略, FetchType.EAGER 表示非延迟 加载, 而 FetchType.LAZY 表示  
//延迟加载 ;  
@Basic(fetch = FetchType.EAGER)  
//columnDefinition 属性指定数据表对应的 Lob 字段类型  
@Column(name = "content", columnDefinition = "LONGTEXT NOT NULL")  
private String content;
```

2.8.6 创建枚举类型的字段

可以使用枚举类型的字段, 不过枚举字段要用 `@Enumerated` 注解修饰。

```
public enum Gender {  
    MALE("男性"),  
    FEMALE("女性");  
  
    private String value;  
    Gender(String str){  
        value=str;  
    }  
}
```

```
@Entity  
@Table(name = "role")  
public class Role {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String name;  
    private String description;  
    @Enumerated(EnumType.STRING)  
    private Gender gender;  
    省略getter/setter.....  
}
```

数据库里面对应存储的是 MAIL/FEMAIL。

2.8.7 增加审计功能

只要继承了 `AbstractAuditBase` 的类都会默认加上下面四个字段。

```
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
@MappedSuperclass  
@EntityListeners(value = AuditingEntityListener.class)  
public abstract class AbstractAuditBase {  
  
    @CreatedDate  
    @Column(updatable = false)  
    @JsonIgnore  
    private Instant createdAt;  
  
    @LastModifiedDate  
    @JsonIgnore  
    private Instant updatedAt;  
  
    @CreatedBy  
    @Column(updatable = false)  
    @JsonIgnore  
    private String createdBy;  
  
    @LastModifiedBy  
    @JsonIgnore  
    private String updatedBy;  
}
```

我们对应的审计功能对应地配置类可能是下面这样的（Spring Security 项目）：

```
@Configuration
@EnableJpaAuditing
public class AuditSecurityConfiguration {
    @Bean
    AuditorAware<String> auditorAware() {
        return () -> Optional.ofNullable(SecurityContextHolder.getContext())
            .map(SecurityContext::getAuthentication)
            .filter(Authentication::isAuthenticated)
            .map(Authentication::getName);
    }
}
```

简单介绍一下上面设计到的一些注解：

1. `@CreatedDate`：表示该字段为创建时间时间字段，在这个实体被 insert 的时候，会设置值
2. `@CreatedBy`：表示该字段为创建人，在这个实体被 insert 的时候，会设置值
`@LastModifiedDate`、`@LastModifiedBy` 同理。

`@EnableJpaAuditing`：开启 JPA 审计功能。

2.8.8 删除/修改数据

`@Modifying` 注解提示 JPA 该操作是修改操作，注意还要配合 `@Transactional` 注解使用。

```
@Repository
public interface UserRepository extends JpaRepository<User, Integer> {

    @Modifying
    @Transactional(rollbackFor = Exception.class)
    void deleteByUserName(String userName);
}
```

2.8.9 关联关系

- `@OneToOne` 声明一对关系
- `@OneToMany` 声明一对多关系
- `@ManyToOne` 声明多对一关系
- `MangToMany` 声明多对多关系

2.9 事务 `@Transactional`

在要开启事务的方法上使用 `@Transactional` 注解即可！

```
@Transactional(rollbackFor = Exception.class)
public void save() {
    .....
}
```

我们知道 `Exception` 分为运行时异常 `RuntimeException` 和非运行时异常。在 `@Transactional` 注解中如果不配置 `rollbackFor` 属性，那么事物只会在遇到 `RuntimeException` 的时候才会回滚，加上 `rollbackFor=Exception.class`，可以让事物在遇到非运行时异常时也回滚。

`@Transactional` 注解一般用在可以作用在类或者方法上。

- 作用于类**: 当把`@Transactional`注解放在类上时，表示所有该类的public方法都配置相同的事务属性信息。
- 作用于方法**: 当类配置了`@Transactional`，方法也配置了`@Transactional`，方法的事务会覆盖类的事务配置信息。

2.10 json 数据处理

2.10.1 过滤 json 数据

`@JsonIgnoreProperties` 作用在类上用于过滤掉特定字段不返回或者不解析。

```
//生成json时将userRoles属性过滤
@JsonIgnoreProperties({"userRoles"})
public class User {

    private String userName;
    private String fullName;
    private String password;
    @JsonIgnore
    private List<UserRole> userRoles = new ArrayList<>();
}
```

`@JsonIgnore`一般用于类的属性上，作用和上面的`@JsonIgnoreProperties`一样。

```
public class User {

    private String userName;
    private String fullName;
    private String password;
    //生成json时将userRoles属性过滤
    @JsonIgnore
    private List<UserRole> userRoles = new ArrayList<>();
}
```

2.10.2 格式化 json 数据

`@JsonFormat`一般用来格式化 json 数据。：

比如：

```
@JsonFormat(shape=JsonFormat.Shape.STRING, pattern="yyyy-MM-
dd'T'HH:mm:ss.SSS'Z'", timezone="GMT")
private Date date;
```

2.10.3 扁平化对象

```
@Getter
@Setter
@ToString
public class Account {
    @JsonUnwrapped
    private Location location;
    @JsonUnwrapped
```

```
private PersonInfo personInfo;

@Getter
@Setter
@ToString
public static class Location {
    private String provinceName;
    private String countyName;
}

@Getter
@Setter
@ToString
public static class PersonInfo {
    private String userName;
    private String fullName;
}

}
```

未扁平化之前：

```
{
    "location": {
        "provinceName": "湖北",
        "countyName": "武汉"
    },
    "personInfo": {
        "userName": "coder1234",
        "fullName": "shaungkou"
    }
}
```

使用 `@JsonUnwrapped` 扁平对象之后：

```
@Getter
@Setter
@ToString
public class Account {
    @JsonUnwrapped
    private Location location;
    @JsonUnwrapped
    private PersonInfo personInfo;
    .....
}
```

```
{
    "provinceName": "湖北",
    "countyName": "武汉",
    "userName": "coder1234",
    "fullName": "shaungkou"
}
```

2.11 测试相关

`@ActiveProfiles` 一般作用于测试类上，用于声明生效的 Spring 配置文件。

```
@SpringBootTest(webEnvironment = RANDOM_PORT)
@ActiveProfiles("test")
@Slf4j
public abstract class TestBase {
    .....
}
```

`@Test` 声明一个方法为测试方法

`@Transactional` 被声明的测试方法的数据会回滚，避免污染测试数据。

`@withMockUser` Spring Security 提供的，用来模拟一个真实用户，并且可以赋予权限。

```
@Test
@Transactional
@withMockUser(username = "user-id-18163138155", authorities =
"ROLE_TEACHER")
void should_import_student_success() throws Exception {
    .....
}
```

3. Spring事务

3.1 什么是事务？

事务是逻辑上的一组操作，要么都执行，要么都不执行。

我们系统的每个业务方法可能包括了多个原子性的数据库操作，比如下面的 `savePerson()` 方法中就有两个原子性的数据库操作。这些原子性的数据库操作是有依赖的，它们要么都执行，要不就都不执行。

```
public void savePerson() {
    personDao.save(person);
    personDetailDao.save(personDetail);
}
```

另外，需要格外注意的是：事务能否生效数据库引擎是否支持事务是关键。比如常用的 MySQL 数据库默认使用支持事务的 `innodb` 引擎。但是，如果把数据库引擎变为 `myisam`，那么程序也就不再支持事务了！

事务最经典也经常被拿出来说例子就是转账了。假如小明要给小红转账 1000 元，这个转账会涉及到两个关键操作就是：

1. 将小明的余额减少 1000 元
2. 将小红的余额增加 1000 元。

万一在这两个操作之间突然出现错误比如银行系统崩溃或者网络故障，导致小明余额减少而小红的余额没有增加，这样就不对了。事务就是保证这两个关键操作要么都成功，要么都要失败。

```
public class OrdersService {
    private AccountDao accountDao;

    public void setOrdersDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }
}
```

```

@Transactional(propagation = Propagation.REQUIRED,
             isolation = Isolation.DEFAULT, readOnly = false, timeout = -1)
public void accountMoney() {
    //小红账户多1000
    accountDao.addMoney(1000, xiaohong);
    //模拟突然出现的异常，比如银行中可能为突然停电等等
    //如果没有配置事务管理的话会造成，小红账户多了1000而小明账户没有少钱
    int i = 10 / 0;
    //小王账户少1000
    accountDao.reduceMoney(1000, xiaoming);
}
}

```

另外，数据库事务的 ACID 四大特性是事务的基础，下面简单来了解一下。

3.2 事物的特性（ACID）了解么？



- **原子性**: 事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；
- **一致性**: 执行事务前后，数据保持一致；
- **隔离性**: 并发访问数据库时，一个用户的事物不被其他事务所干扰也就是说多个事务并发执行时，一个事务的执行不应影响其他事务的执行；
- **持久性**: 一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。

3.3 详谈 Spring 对事务的支持

再提醒一次：你的程序是否支持事务首先取决于数据库，比如使用 MySQL 的话，如果你选择的是 innodb 引擎，那么恭喜你，是可以支持事务的。但是，如果你的 MySQL 数据库使用的是 myisam 引擎的话，那不好意思，从根上就是不支持事务的。

这里再多提一下一个非常重要的知识点：MySQL 怎么保证原子性的？

我们知道如果想要保证事务的原子性，就需要在异常发生时，对已经执行的操作进行回滚，在 MySQL 中，恢复机制是通过 **回滚日志（undo log）** 实现的，所有事务进行的修改都会先记录到这个回滚日志中，然后再执行相关的操作。如果执行过程中遇到异常的话，我们直接利用 **回滚日志** 中的信息将数据回滚到修改之前的样子即可！并且，回滚日志会先于数据持久化到磁盘上。这样就保证了即使遇到数据库突然宕机等情况，当用户再次启动数据库的时候，数据库还能够通过查询回滚日志来回滚将之前未完成的事务。

3.3.1. Spring 支持两种方式的事务管理

1) 编程式事务管理

通过 `TransactionTemplate` 或者 `TransactionManager` 手动管理事务，实际应用中很少使用，但是对于你理解 Spring 事务管理原理有帮助。

使用 `TransactionTemplate` 进行编程式事务管理的示例代码如下：

```
@Autowired  
private TransactionTemplate transactionTemplate;  
public void testTransaction() {  
  
    transactionTemplate.execute(new TransactionCallbackWithoutResult() {  
        @Override  
        protected void doInTransactionWithoutResult(TransactionStatus transactionStatus) {  
  
            try {  
  
                // .... 业务代码  
            } catch (Exception e){  
                //回滚  
                transactionStatus.setRollbackOnly();  
            }  
        }  
    });  
}
```

使用 `TransactionManager` 进行编程式事务管理的示例代码如下：

```
@Autowired  
private PlatformTransactionManager transactionManager;  
  
public void testTransaction() {  
  
    TransactionStatus status = transactionManager.getTransaction(new DefaultTransactionDefinition());  
    try {  
        // .... 业务代码  
        transactionManager.commit(status);  
    } catch (Exception e) {  
        transactionManager.rollback(status);  
    }  
}
```

2) 声明式事务管理

推荐使用（代码侵入性最小），实际是通过 AOP 实现（基于 `@Transactional` 的全注解方式使用最多）。

使用 `@Transactional` 注解进行事务管理的示例代码如下：

```

@Transactional(propagation=propagation.PROPAGATION_REQUIRED)
public void aMethod {
    //do something
    B b = new B();
    C c = new C();
    b.bMethod();
    c.cMethod();
}

```

3.3.2 Spring 事务管理接口介绍

Spring 框架中，事务管理相关最重要的 3 个接口如下：

- `PlatformTransactionManager`：（平台）事务管理器，Spring 事务策略的核心。
- `TransactionDefinition`：事务定义信息(事务隔离级别、传播行为、超时、只读、回滚规则)。
- `TransactionStatus`：事务运行状态。

我们可以把 `PlatformTransactionManager` 接口可以被看作是事务上层的管理者，而 `TransactionDefinition` 和 `TransactionStatus` 这两个接口可以看作是事物的描述。

`PlatformTransactionManager` 会根据 `TransactionDefinition` 的定义比如事务超时时间、隔离级别、传播行为等来进行事务管理，而 `TransactionStatus` 接口则提供了一些方法来获取事务相应状态比如是否新事务、是否可以回滚等等。

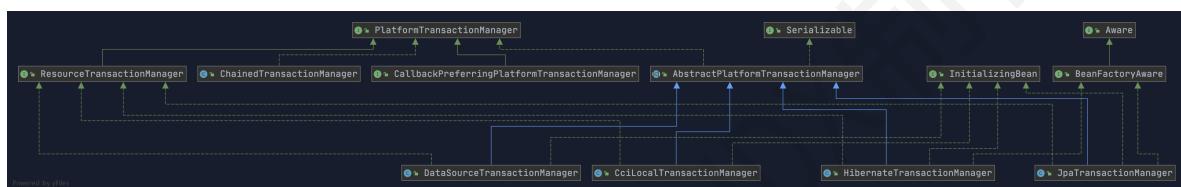
1) PlatformTransactionManager: 事务管理接口

Spring 并不直接管理事务，而是提供了多种事务管理器。Spring 事务管理器的接口是：

`PlatformTransactionManager`。

通过这个接口，Spring 为各个平台如 JDBC(`DataSourceTransactionManager`)、Hibernate(`HibernateTransactionManager`)、JPA(`JpaTransactionManager`)等都提供了对应的事务管理器，但是具体的实现就是各个平台自己的事情了。

`PlatformTransactionManager` 接口的具体实现如下：



`PlatformTransactionManager` 接口中定义了三个方法：

```

package org.springframework.transaction;

import org.springframework.lang.Nullable;

public interface PlatformTransactionManager {
    // 获得事务
    TransactionStatus getTransaction(@Nullable TransactionDefinition var1)
    throws TransactionException;
    // 提交事务
    void commit(TransactionStatus var1) throws TransactionException;
    // 回滚事务
    void rollback(TransactionStatus var1) throws TransactionException;
}

```

这里多插一嘴。为什么要定义或者说抽象出来 `PlatformTransactionManager` 这个接口呢？

主要是因为要将事务管理行为抽象出来，然后不同的平台去实现它，这样我们可以保证提供给外部的行为不变，方便我们扩展。我前段时间分享过：“**为什么我们要用接口？**”

此刻在火车上，简单分享一下自己对于为啥要用接口的看法吧！接口很多人估计每天都是迷迷糊糊的用，不太明白具体原因。

《设计模式》（GOF那本）这本书在很多年前都提到过说要基于接口而非实现编程，你真的知道为什么要基于接口编程么？

纵观开源框架和项目的源码，接口是它们不可或缺的重要组成部分。要理解为什么要用接口，首先要搞懂接口提供了什么功能。我们可以把接口理解为提供了一系列功能列表的约定，接口本身不提供功能，它只定义行为。但是谁要用，就要先实现我，遵守我的约定，然后再自己去实现我定义的要实现的功能。

举个例子，我上个项目有发送短信的需求，为此，我们定了一个接口，接口只有两个方法：

- 1.发送短信
- 2.处理发送结果的方法。

刚开始我们用的是阿里云短信服务，然后我们实现这个接口完成了一个阿里云短信的服务。后来，我们突然又换到了别的短信服务平台，我们这个时候只需要再实现这个接口即可。这样保证了我们提供给外部的行为不变。几乎不需要改变什么代码，我们就轻松完成了需求的转变，提高了代码的灵活性和可扩展性。

什么时候用接口？

当你要实现的功能模块设计抽象行为的时候，比如发送短信的服务，图床的存储服务等等。

2) TransactionDefinition:事务属性

事务管理器接口 `PlatformTransactionManager` 通过 `getTransaction(TransactionDefinition definition)` 方法来得到一个事务，这个方法里面的参数是 `TransactionDefinition` 类，这个类就定义了一些基本的事务属性。

那么什么是 **事务属性** 呢？

事务属性可以理解成事务的一些基本配置，描述了事务策略如何应用到方法上。

事务属性包含了 5 个方面：



`TransactionDefinition` 接口中定义了 5 个方法以及一些表示事务属性的常量比如隔离级别、传播行为等等。

```

package org.springframework.transaction;

import org.springframework.lang.Nullable;

public interface TransactionDefinition {
    int PROPAGATION_REQUIRED = 0;
    int PROPAGATION_SUPPORTS = 1;
    int PROPAGATION_MANDATORY = 2;
    int PROPAGATION_REQUIRES_NEW = 3;
    int PROPAGATION_NOT_SUPPORTED = 4;
    int PROPAGATION_NEVER = 5;
    int PROPAGATION_NESTED = 6;
    int ISOLATION_DEFAULT = -1;
    int ISOLATION_READ_UNCOMMITTED = 1;
    int ISOLATION_READ_COMMITTED = 2;
    int ISOLATION_REPEATABLE_READ = 4;
    int ISOLATION_SERIALIZABLE = 8;
    int TIMEOUT_DEFAULT = -1;
    // 返回事务的传播行为, 默认值为 REQUIRED。
    int getPropagationBehavior();
    // 返回事务的隔离级别, 默认值是 DEFAULT
    int getIsolationLevel();
    // 返回事务的超时时间, 默认值为 -1。如果超过该时间限制但事务还没有完成, 则自动回滚事务。
    int getTimeout();
    // 返回是否为只读事务, 默认值为 false
    boolean isReadOnly();

    @Nullable
    String getName();
}

```

3) `TransactionStatus`: 事务状态

`TransactionStatus` 接口用来记录事务的状态 该接口定义了一组方法, 用来获取或判断事务的相应状态信息。

`PlatformTransactionManager.getTransaction()` 方法返回一个 `Transactionstatus` 对象。

`TransactionStatus` 接口接口内容如下：

```
public interface TransactionStatus{  
    boolean isNewTransaction(); // 是否是新的事物  
    boolean hasSavepoint(); // 是否有恢复点  
    void setRollbackOnly(); // 设置为只回滚  
    boolean isRollbackOnly(); // 是否为只回滚  
    boolean isCompleted(); // 是否已完成  
}
```

3.3.3 事务属性详解

实际业务开发中，大家一般都是使用 `@Transactional` 注解来开启事务，很多人并不清楚这个参数里面的参数是什么意思，有什么用。为了更好的在项目中使用事务管理，强烈推荐好好阅读一下下面的内容。

1) 事务传播行为

事务传播行为是为了解决业务层方法之间互相调用的事务问题。

当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。

举个例子！

我们在 A 类的 `aMethod()` 方法中调用了 B 类的 `bMethod()` 方法。这个时候就涉及到业务层方法之间互相调用的事务问题。如果我们的 `bMethod()` 如果发生异常需要回滚，如何配置事务传播行为才能让 `aMethod()` 也跟着回滚呢？这个时候就需要事务传播行为的知识了，如果你不知道的话一定要好好看一下。

```
Class A {  
    @Transactional(propagation=propagation.xxx)  
    public void aMethod {  
        //do something  
        B b = new B();  
        b.bMethod();  
    }  
}  
  
Class B {  
    @Transactional(propagation=propagation.xxx)  
    public void bMethod {  
        //do something  
    }  
}
```

在 `TransactionDefinition` 定义中包括了如下几个表示传播行为的常量：

```
public interface TransactionDefinition {  
    int PROPAGATION_REQUIRED = 0;  
    int PROPAGATION_SUPPORTS = 1;  
    int PROPAGATION_MANDATORY = 2;  
    int PROPAGATION_REQUIRE_NEW = 3;  
    int PROPAGATION_NOT_SUPPORTED = 4;  
    int PROPAGATION_NEVER = 5;  
    int PROPAGATION_NESTED = 6;  
    ....  
}
```

不过如此，为了方便使用，Spring 会相应地定义了一个枚举类：`Propagation`

```
package org.springframework.transaction.annotation;  
  
import org.springframework.transaction.TransactionDefinition;  
  
public enum Propagation {  
  
    REQUIRED(TransactionDefinition.PROPAGATION_REQUIRED),  
  
    SUPPORTS(TransactionDefinition.PROPAGATION_SUPPORTS),  
  
    MANDATORY(TransactionDefinition.PROPAGATION_MANDATORY),  
  
    REQUIRE_NEW(TransactionDefinition.PROPAGATION_REQUIRE_NEW),  
  
    NOT_SUPPORTED(TransactionDefinition.PROPAGATION_NOT_SUPPORTED),  
  
    NEVER(TransactionDefinition.PROPAGATION_NEVER),  
  
    NESTED(TransactionDefinition.PROPAGATION_NESTED);  
  
    private final int value;  
  
    Propagation(int value) {  
        this.value = value;  
    }  
  
    public int value() {  
        return this.value;  
    }  
}
```

正确的事务传播行为可能的值如下：

1. `TransactionDefinition.PROPAGATION_REQUIRED`

使用的最多的一个事务传播行为，我们平时经常使用的`@Transactional`注解默认使用就是这个事务传播行为。如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。也就是说：

1. 如果外部方法没有开启事务的话，`Propagation.REQUIRED`修饰的内部方法会新开启自己的事务，且开启的事务相互独立，互不干扰。

2. 如果外部方法开启事务并且被 `Propagation.REQUIRED` 的话，所有 `Propagation.REQUIRED` 修饰的内部方法和外部方法均属于同一事务，只要一个方法回滚，整个事务均回滚。

举个例子：如果我们上面的 `aMethod()` 和 `bMethod()` 使用的都是 `PROPAGATION_REQUIRED` 传播行为的话，两者使用的就是同一个事务，只要其中一个方法回滚，整个事务均回滚。

```
class A {  
    @Transactional(propagation=propagation.PROPAGATION_REQUIRED)  
    public void aMethod {  
        //do something  
        B b = new B();  
        b.bMethod();  
    }  
}  
  
class B {  
    @Transactional(propagation=propagation.PROPAGATION_REQUIRED)  
    public void bMethod {  
        //do something  
    }  
}
```

2. `TransactionDefinition.PROPAGATION_REQUIRE_NEW`

创建一个新的事务，如果当前存在事务，则把当前事务挂起。也就是说不管外部方法是否开启事务，`Propagation.REQUIRES_NEW` 修饰的内部方法会新开启自己的事务，且开启的事务相互独立，互不干扰。

举个例子：如果我们上面的 `bMethod()` 使用 `PROPAGATION_REQUIRE_NEW` 事务传播行为修饰，`aMethod` 还是用 `PROPAGATION_REQUIRED` 修饰的话。如果 `aMethod()` 发生异常回滚，`bMethod()` 不会跟着回滚，因为 `bMethod()` 开启了独立的事务。但是，如果 `bMethod()` 抛出了未被捕获的异常并且这个异常满足事务回滚规则的话，`aMethod()` 同样也会回滚，因为这个异常被 `aMethod()` 的事务管理机制检测到了。

```
class A {  
    @Transactional(propagation=propagation.PROPAGATION_REQUIRED)  
    public void aMethod {  
        //do something  
        B b = new B();  
        b.bMethod();  
    }  
}  
  
class B {  
    @Transactional(propagation=propagation.REQUIRES_NEW)  
    public void bMethod {  
        //do something  
    }  
}
```

3. `TransactionDefinition.PROPAGATION_NESTED` :

如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于 `TransactionDefinition.PROPAGATION_REQUIRED`。也就是说：

1. 在外部方法未开启事务的情况下 `Propagation.NESTED` 和 `Propagation.REQUIRED` 作用相同，修饰的内部方法都会新开启自己的事务，且开启的事务相互独立，互不干扰。
2. 如果外部方法开启事务的话，`Propagation.NESTED` 修饰的内部方法属于外部事务的子事务，外部主事务回滚的话，子事务也会回滚，而内部子事务可以单独回滚而不影响外部主事务和其他子事务。

这里还是简单举个例子：

如果 `aMethod()` 回滚的话，`bMethod()` 和 `bMethod2()` 都要回滚，而 `bMethod()` 回滚的话，并不会造成 `aMethod()` 和 `bMethod()` 回滚。

```
Class A {  
    @Transactional(propagation=propagation.PROpagation_REQUIRED)  
    public void aMethod {  
        //do something  
        B b = new B();  
        b.bMethod();  
        b.bMethod2();  
    }  
}  
  
Class B {  
    @Transactional(propagation=propagation.PROpagation_NESTED)  
    public void bMethod {  
        //do something  
    }  
    @Transactional(propagation=propagation.PROpagation_NESTED)  
    public void bMethod2 {  
        //do something  
    }  
}
```

4. `TransactionDefinition.PROpagation_MANDATORY`

如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。（mandatory：强制性）

这个使用的很少，就不举例子来说了。

若是错误的配置以下 3 种事务传播行为，事务将不会发生回滚，这里不对照案例讲解了，使用的很少。

- `TransactionDefinition.PROpagation_SUPPORTS`：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。
- `TransactionDefinition.PROpagation_NOT_SUPPORTED`：以非事务方式运行，如果当前存在事务，则把当前事务挂起。
- `TransactionDefinition.PROpagation_NEVER`：以非事务方式运行，如果当前存在事务，则抛出异常。

2) 事务隔离级别

`TransactionDefinition` 接口中定义了五个表示隔离级别的常量：

```
public interface TransactionDefinition {  
    ....  
    int ISOLATION_DEFAULT = -1;  
    int ISOLATION_READ_UNCOMMITTED = 1;  
    int ISOLATION_READ_COMMITTED = 2;  
    int ISOLATION_REPEATABLE_READ = 4;  
    int ISOLATION_SERIALIZABLE = 8;  
    ....  
}
```

和事务传播行为这块一样，为了方便使用，Spring 也相应地定义了一个枚举类：`Isolation`

```
public enum Isolation {  
  
    DEFAULT(TransactionDefinition.ISOLATION_DEFAULT),  
  
    READ_UNCOMMITTED(TransactionDefinition.ISOLATION_READ_UNCOMMITTED),  
  
    READ_COMMITTED(TransactionDefinition.ISOLATION_READ_COMMITTED),  
  
    REPEATABLE_READ(TransactionDefinition.ISOLATION_REPEATABLE_READ),  
  
    SERIALIZABLE(TransactionDefinition.ISOLATION_SERIALIZABLE);  
  
    private final int value;  
  
    Isolation(int value) {  
        this.value = value;  
    }  
  
    public int value() {  
        return this.value;  
    }  
}
```

下面我依次对每一种事务隔离级别进行介绍：

- `TransactionDefinition.ISOLATION_DEFAULT`：使用后端数据库默认的隔离级别，MySQL 默认采用的 `REPEATABLE_READ` 隔离级别 Oracle 默认采用的 `READ_COMMITTED` 隔离级别。
- `TransactionDefinition.ISOLATION_READ_UNCOMMITTED`：最低的隔离级别，使用这个隔离级别很少，因为它允许读取尚未提交的数据变更，**可能会导致脏读、幻读或不可重复读**
- `TransactionDefinition.ISOLATION_READ_COMMITTED`：允许读取并发事务已经提交的数据，**可以阻止脏读，但是幻读或不可重复读仍有可能发生**
- `TransactionDefinition.ISOLATION_REPEATABLE_READ`：对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，**可以阻止脏读和不可重复读，但幻读仍有可能发生**。
- `TransactionDefinition.ISOLATION_SERIALIZABLE`：最高的隔离级别，完全服从 ACID 的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，**该级别可以防止脏读、不可重复读以及幻读**。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

因为平时使用 MySQL 数据库比较多，这里再多提一嘴！

MySQL InnoDB 存储引擎的默认支持的隔离级别是 `REPEATABLE-READ`（可重读）。我们可以通过 `SELECT @@tx_isolation;` 命令来查看，如下：

```
mysql> SELECT @@tx_isolation;
+-----+
| @@tx_isolation |
+-----+
| REPEATABLE-READ |
+-----+
```

这里需要注意的是：与 SQL 标准不同的地方在于 InnoDB 存储引擎在 **REPEATABLE-READ**（可重读）事务隔离级别下使用的是 Next-Key Lock 锁算法，因此可以避免幻读的产生，这与其他数据库系统（如 SQL Server）是不同的。所以说 InnoDB 存储引擎的默认支持的隔离级别是 **REPEATABLE-READ**（可重读）已经可以完全保证事务的隔离性要求，即达到了 SQL 标准的 **SERIALIZABLE**（可串行化）隔离级别。

因为隔离级别越低，事务请求的锁越少，所以大部分数据库系统的隔离级别都是 **READ-COMMITTED**（读取提交内容），但是你要知道的是 InnoDB 存储引擎默认使用 **REPEATABLE-READ**（可重读）并不会有什么性能上的损失。

3) 事务超时属性

所谓事务超时，就是指一个事务所允许执行的最长时间，如果超过该时间限制但事务还没有完成，则自动回滚事务。在 **TransactionDefinition** 中以 int 的值来表示超时时间，其单位是秒，默认值为 -1。

4) 事务只读属性

```
package org.springframework.transaction;

import org.springframework.lang.Nullable;

public interface TransactionDefinition {
    ...
    // 返回是否为只读事务，默认值为 false
    boolean isReadOnly();
}
```

对于只有读取数据查询的事务，可以指定事务类型为 `readonly`，即只读事务。只读事务不涉及数据的修改，数据库会提供一些优化手段，适合用在有多条数据库查询操作的方法中。

很多人就会疑问了，为什么我一个数据查询操作还要启用事务支持呢？

MySQL 默认对每一个新建立的连接都启用了 `autocommit` 模式。在该模式下，每一个发送到 MySQL 服务器的 `sql` 语句都会在一个单独的事务中进行处理，执行结束后会自动提交事务，并开启一个新的事务。

但是，如果你给方法加上了 `Transactional` 注解的话，这个方法执行的所有 `sql` 会被放在一个事务中。如果声明了只读事务的话，数据库就会去优化它的执行，并不会带来其他的什么收益。

如果不加 `Transactional`，每条 `sql` 会开启一个单独的事务，中间被其它事务改了数据，都会实时读取到最新值。

分享一下关于事务只读属性，其他人的解答：

1. 如果你一次执行单条查询语句，则没有必要启用事务支持，数据库默认支持 SQL 执行期间的读一致性；
2. 如果你一次执行多条查询语句，例如统计查询，报表查询，在这种场景下，多条查询 SQL 必须保证整体的读一致性，否则，在前条 SQL 查询之后，后条 SQL 查询之前，数据被其他用户改变，则该次整体的统计查询将会出现读数据不一致的状态，此时，应该启用事务支持

5) 事务回滚规则

这些规则定义了哪些异常会导致事务回滚而哪些不会。默认情况下，事务只有遇到运行期异常（`RuntimeException` 的子类）时才会回滚，`Error` 也会导致事务回滚，但是，在遇到检查型（`Checked`）异常时不会回滚。

```
/*
 * Defines zero (0) or more exception {@link Class classes}, which must be
 * subclasses of {@link Throwable}, indicating which exception types must cause
 * a transaction rollback.
 * <p>By default, a transaction will be rolling back on {@link RuntimeException}
 * and {@link Error} but not on checked exceptions (business exceptions). See
 * {@link org.springframework.transaction.interceptor.DefaultTransactionAttribute#rollbackOn(Throwable)}
 * for a detailed explanation.
 * <p>This is the preferred way to construct a rollback rule (in contrast to
 * {@link #rollbackForClassName}), matching the exception class and its subclasses.
 * <p>Similar to {@link org.springframework.transaction.interceptor.RollbackRuleAttribute#RollbackRuleAttribute#RollbackRuleAttribute}
 * @see #rollbackForClassName
 * @see org.springframework.transaction.interceptor.DefaultTransactionAttribute#rollbackOn(Throwable)
 */
Class<? extends Throwable>[] rollbackFor() default {};
```

默认遇到运行时异常和 Error 回滚
遇到检查型异常不会滚

如果你想要回滚你定义的特定的异常类型的话，可以这样：

```
@Transactional(rollbackFor= MyException.class)
```

3.3.4 `@Transactional` 注解使用详解

1) `@Transactional` 的作用范围

- 方法**：推荐将注解使用于方法上，不过需要注意的是：该注解只能应用到 `public` 方法上，否则不生效。
- 类**：如果这个注解使用在类上的话，表明该注解对该类中所有的 `public` 方法都生效。
- 接口**：不推荐在接口上使用。

2) `@Transactional` 的常用配置参数

`@Transactional` 注解源码如下，里面包含了基本事务属性的配置：

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface Transactional {

    @AliasFor("transactionManager")
    String value() default "";

    @AliasFor("value")
    String transactionManager() default "";

    Propagation propagation() default Propagation.REQUIRED;

    Isolation isolation() default Isolation.DEFAULT;

    int timeout() default TransactionDefinition.TIMEOUT_DEFAULT;

    boolean readOnly() default false;
```

```

    Class<? extends Throwable>[] rollbackFor() default {};
    String[] rollbackForClassName() default {};
    Class<? extends Throwable>[] noRollbackFor() default {};
    String[] noRollbackForClassName() default {};
}

```

`@Transactional` 的常用配置参数总结 (只列巨额 5 个我平时比较常用的) :

属性名	说明
propagation	事务的传播行为, 默认值为 REQUIRED, 可选的值在上面介绍过
isolation	事务的隔离级别, 默认值采用 DEFAULT, 可选的值在上面介绍过
timeout	事务的超时时间, 默认值为-1 (不会超时)。如果超过该时间限制但事务还没有完成, 则自动回滚事务。
readOnly	指定事务是否为只读事务, 默认值为 false。
rollbackFor	用于指定能够触发事务回滚的异常类型, 并且可以指定多个异常类型。

3) `@Transactional` 事务注解原理

面试中在问 AOP 的时候可能会被问到的一个问题。简单说下吧!

我们知道, `@Transactional` 的工作机制是基于 AOP 实现的, AOP 又是使用动态代理实现的。如果目标对象实现了接口, 默认情况下会采用 JDK 的动态代理, 如果目标对象没有实现了接口, 会使用 CGLIB 动态代理。

多提一嘴: `createAopProxy()` 方法决定了是使用 JDK 还是 Cglib 来做动态代理, 源码如下:

```

public class DefaultAopProxyFactory implements AopProxyFactory, Serializable {

    @Override
    public AopProxy createAopProxy(AdvisedSupport config) throws
AopConfigException {
        if (config.isOptimize() || config.isProxyTargetClass() ||
hasNoUserSuppliedProxyInterfaces(config)) {
            Class<?> targetClass = config.getTargetClass();
            if (targetClass == null) {
                throw new AopConfigException("TargetSource cannot determine
target class: " +
                    "Either an interface or a target is required for proxy
creation.");
            }
            if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
                return new JdkDynamicAopProxy(config);
            }
            return new ObjenesisCglibAopProxy(config);
        }
        else {
            return new JdkDynamicAopProxy(config);
        }
    }
}

```

```
    }  
    ....  
}
```

如果一个类或者一个类中的 public 方法上被标注 `@Transactional` 注解的话，Spring 容器就会在启动的时候为其创建一个代理类，在调用被 `@Transactional` 注解的 public 方法的时候，实际调用的是 `TransactionInterceptor` 类中的 `invoke()` 方法。这个方法的作用就是在目标方法之前开启事务，方法执行过程中如果遇到异常的时候回滚事务，方法调用完成之后提交事务。

`TransactionInterceptor` 类中的 `invoke()` 方法内部实际调用的是 `TransactionAspectSupport` 类的 `invokewithinTransaction()` 方法。由于新版本的 Spring 对这部分重写很大，而且用到了很多响应式编程的知识，这里就不列源码了。

4) Spring AOP 自调用问题

若同一类中的其他没有 `@Transactional` 注解的方法内部调用有 `@Transactional` 注解的方法，有 `@Transactional` 注解的方法的事务会失效。

这是由于 Spring AOP 代理的原因造成的，因为只有当 `@Transactional` 注解的方法在类以外被调用的时候，Spring 事务管理才生效。

`MyService` 类中的 `method1()` 调用 `method2()` 就会导致 `method2()` 的事务失效。

```
@Service  
public class MyService {  
  
    private void method1() {  
        method2();  
        //.....  
    }  
    @Transactional  
    public void method2() {  
        //.....  
    }  
}
```

解决办法就是避免同一类中自调用或者使用 AspectJ 取代 Spring AOP 代理。

`@Transactional` 的使用注意事项总结

1. `@Transactional` 注解只有作用到 public 方法上事务才生效，不推荐在接口上使用；
2. 避免同一个类中调用 `@Transactional` 注解的方法，这样会导致事务失效；
3. 正确的设置 `@Transactional` 的 `rollbackFor` 和 `propagation` 属性，否则事务可能会回滚失败
4.

4. Spring IOC和AOP详解

下面从以下几个问题展开对IOC & AOP的解释

- 什么是IOC？
- IOC解决了什么问题？
- IOC 和 DI 的区别？
- 什么是AOP？
- AOP解决了什么问题？
- AOP为什么叫做切面变成？

首先声明：IOC & AOP不是Spring提出来的，它们在Spring之前其实已经存在了，只不过当时更加偏向于理论。Spring在技术层次将这两个思想进行了很好的实现。

4.1 什么是 IOC

IOC (Inversion of control) 控制反转/反转控制。它是一种思想不是一个技术实现。描述的是：Java 开发领域对象的创建以及管理的问题。

例如：现有类A依赖于类B

传统的开发方式：往往是在类A中手动通过new关键字来 new 一个B的对象出来

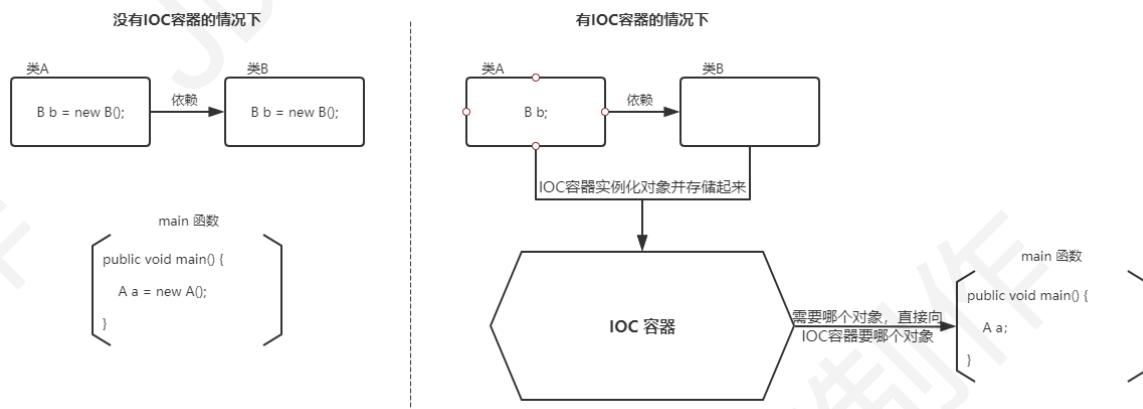
使用IOC思想的开发方式：不通过new关键字来创建对象，而是通过IOC容器(Spring 框架) 来帮助我们实例化对象。我们需要哪个对象，直接从IOC容器里面过去即可。

从以上两种开发方式的对比来看：我们“丧失了一个权力”(创建、管理对象的权力)，从而也得到了一个好处 (不用再考虑对象的创建、管理等一系列的事情)

4.1.1 为什么叫控制反转

控制：指的是对象创建 (实例化、管理) 的权力

反转：控制权交给外部环境 (Spring框架、IOC容器)

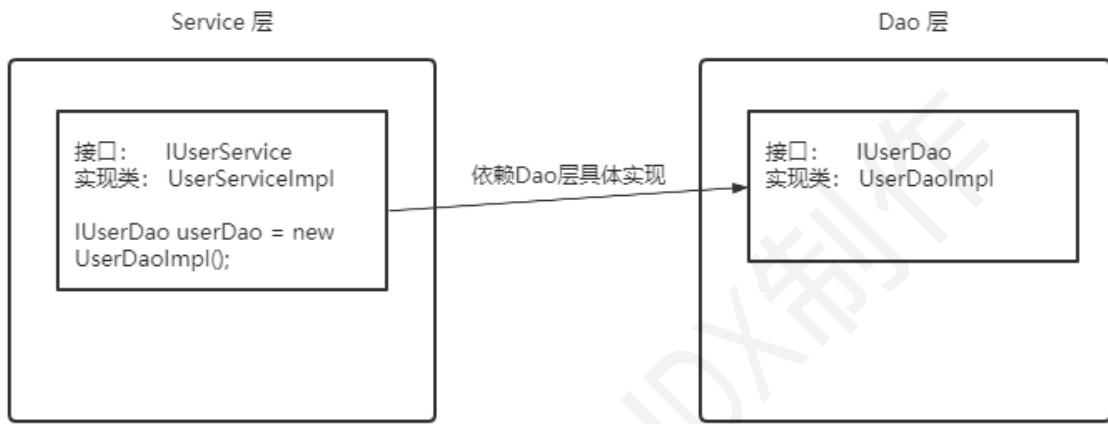


4.2 IOC 解决了什么问题

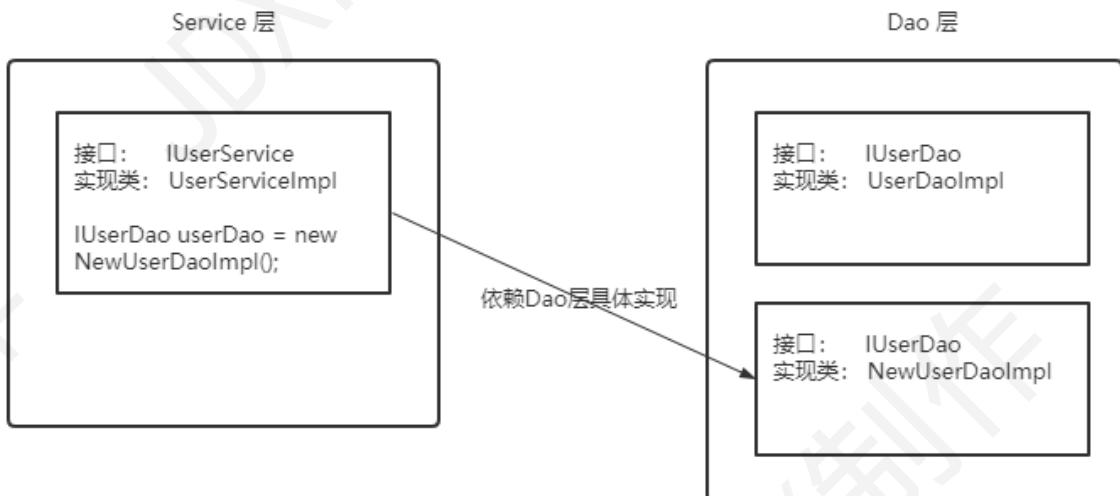
IOC 主要解决的是对象之间的耦合问题。

例如：现有一个针对User的操作，利用 Service 和 Dao 两层结构进行开发

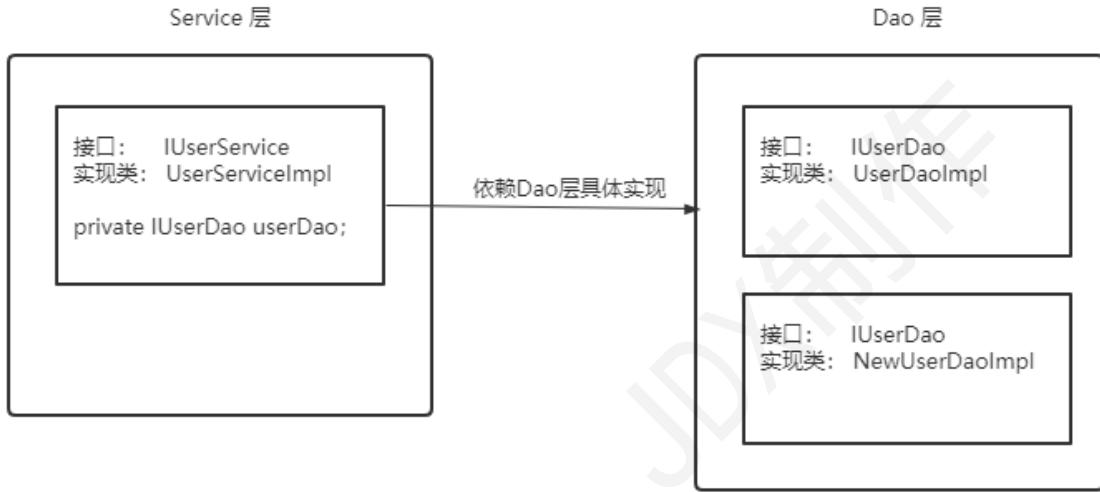
在没有使用IOC思想的情况下，Service 层想要使用 Dao层的具体实现的话，需要通过new关键字在 UserServiceImpl 中手动 new出 IUserDao 的具体实现类 UserDaoImpl (不能直接new接口类)。



很完美，这种方式也是可以实现的，但是我们想象一下如下场景：开发过程中突然接到一个新的需求，针对对IUserDao 接口开发出另一个具体实现类。因为Service层依赖了IUserDao的具体实现，所以我们需要修改UserServiceImpl中new的对象。如果只有一个类引用了IUserDao的具体实现，可能觉得还好，修改起来也不是很费力气，但是如果有许多许多的地方都引用了IUserDao的具体实现的话，一旦需要更换IUserDao的实现方式，那修改起来将会非常的头疼。



使用IOC的思想，我们将对象的控制权（创建、管理）交给IOC容器去管理，我们在使用的时候直接向IOC容器“要”就可以了



4.3 IOC 和 DI 的区别

IOC 和 DI 描述的是同一件事情（对象实例化以及依赖关系的维护），只不过角度不同。

IOC (Inversion of control) 控制反转/反转控制。是站在对象的角度，对象实例化以及管理的权限（反转）交给了容器。

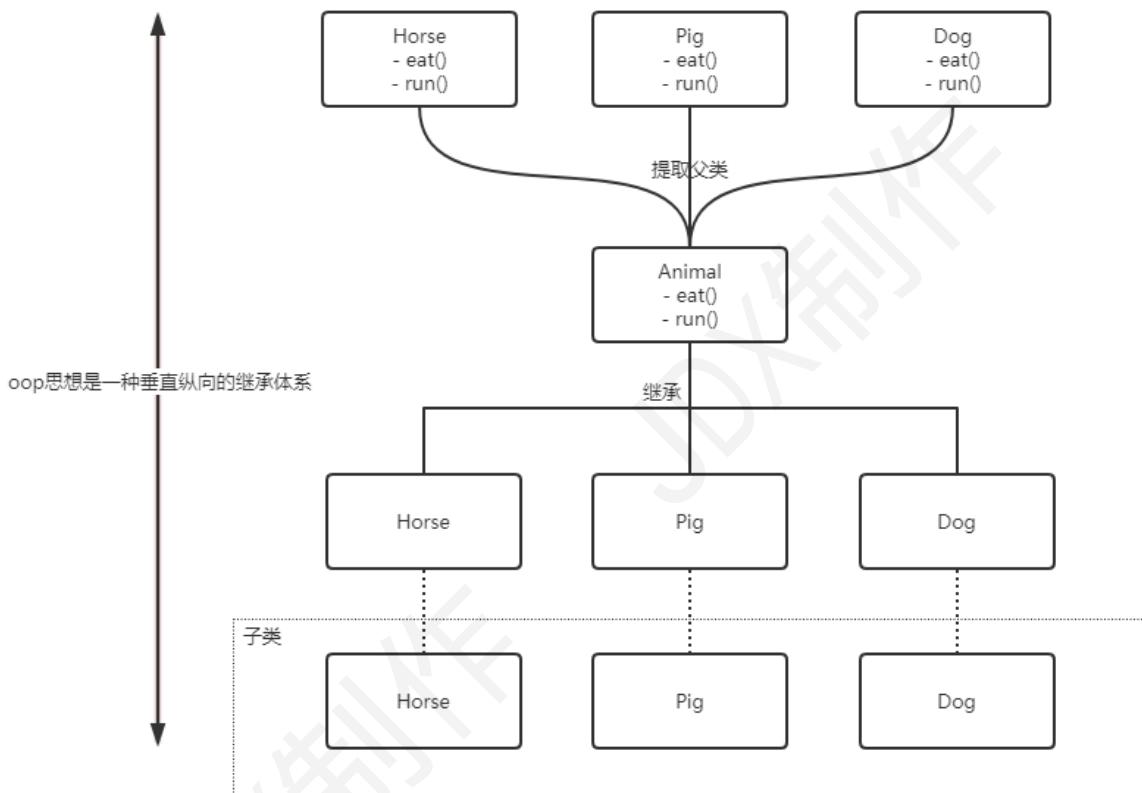
DI (Dependancy Injection) 依赖注入。是站在容器的角度，容器会把对象依赖的其他对象注入（送进去）。例如：对象A 实例化过程中因为声明了一个B类型的属性，那么就需要容器把B对象注入到A中。

4.4 什么是AOP

AOP: Aspect oriented programming 面向切面编程，AOP是 OOP（面向对象编程）的一种延续，下面我们将先看一个OOP的例子。

例如：现有三个类，Horse、Pig、Dog，这三个类中都有 eat 和 run 两个方法。

通过OOP思想中的继承，我们可以提取出一个 Animal 的父类，然后将 eat 和 run 方法放入父类中，Horse、Pig、Dog通过继承Animal类即可自动获得 eat 和 run 方法。这样将会少些很多重复的代码。



OOP编程思想可以解决大部分的代码重复问题。但是有一些问题是处理不了的。比如在父类Animal 中的多个方法的相同位置出现了重复的代码，OOP就解决不了。

```

COPY/**
 * 动物父类
 */
public class Animal {

    /** 身高 */
    private String height;

    /** 体重 */
    private double weight;

    public void eat() {
        // 性能监控代码
        long start = System.currentTimeMillis();

        // 业务逻辑代码
        System.out.println("I can eat...");

        // 性能监控代码
        System.out.println("执行时长: " + (System.currentTimeMillis() - start)/1000f + "s");
    }

    public void run() {
        // 性能监控代码
        long start = System.currentTimeMillis();

        // 业务逻辑代码
        System.out.println("I can run...");
    }
}

```

```
// 性能监控代码  
System.out.println("执行时长: " + (System.currentTimeMillis() -  
start)/1000f + "s");  
}  
}
```

这部分重复的代码，一般统称为 **横切逻辑代码**。

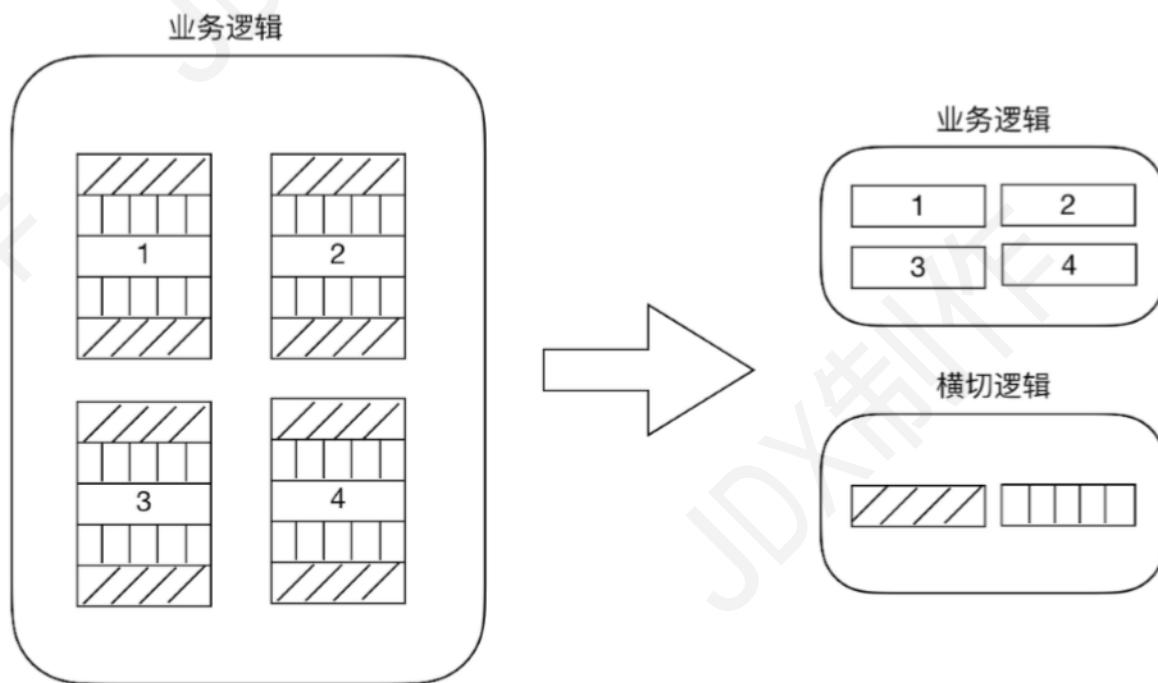
The requested content cannot be loaded.
Please try again later.

横切逻辑代码存在的问题：

- 代码重复问题
- 横切逻辑代码和业务代码混杂在一起，代码臃肿，不变维护

AOP 就是用来解决这些问题的

AOP 另辟蹊径，提出横向抽取机制，将横切逻辑代码和业务逻辑代码分离



代码拆分比较容易，难的是如何在不改变原有业务逻辑的情况下，悄无声息的将横向逻辑代码应用到原有的业务逻辑中，达到和原来一样的效果。

4.5 AOP解决了什么问题

通过上面的分析可以发现，AOP主要用来解决：在不改变原有业务逻辑的情况下，增强横切逻辑代码，根本上解耦合，避免横切逻辑代码重复。

4.6 AOP为什么叫面向切面编程

切：指的是横切逻辑，原有业务逻辑代码不动，只能操作横切逻辑代码，所以面向横切逻辑

面：横切逻辑代码往往要影响的是很多个方法，每个方法如同一个点，多个点构成一个面。这里有一个面的概念

5. Spring中 Bean 的作用域与生命周期

5.1 前言

在 Spring 中，那些组成应用程序的主体及由 Spring IOC 容器所管理的对象，被称之为 bean。简单地讲，bean 就是由 IOC 容器初始化、装配及管理的对象，除此之外，bean 就与应用程序中的其他对象没有什么区别了。而 bean 的定义以及 bean 相互间的依赖关系将通过配置元数据来描述。

Spring中的bean默认都是单例的，这些单例Bean在多线程程序下如何保证线程安全呢？ 例如对于 Web 应用来说，Web 容器对于每个用户请求都创建一个单独的 Sevlet 线程来处理请求，引入 Spring 框架之后，每个 Action 都是单例的，那么对于 Spring 托管的单例 Service Bean，如何保证其安全呢？**Spring 的单例是基于 BeanFactory 也就是 Spring 容器的，单例 Bean 在此容器内只有一个，Java 的单例是基于 JVM，每个 JVM 内只有一个实例。**

在大多数情况下。单例 bean 是很理想的方案。不过，有时候你可能会发现你所使用的类是易变的，它们会保持一些状态，因此重用是不安全的。在这种情况下，将 class 声明为单例的就不是那么明智了。因为对象会被污染，稍后重用的时候会出现意想不到的问题。所以 Spring 定义了多种作用域的 bean。

5.2 bean的作用域

创建一个 bean 定义，其实质是用该 bean 定义对应的类来创建真正实例的“配方”。把 bean 定义看成一个配方很有意义，它与 class 很类似，只根据一张“处方”就可以创建多个实例。不仅可以控制注入到对象中的各种依赖和配置值，还可以控制该对象的作用域。这样可以灵活选择所建对象的作用域，而不必在 Java Class 级定义作用域。Spring Framework 支持五种作用域，分别阐述如下表。

类别	说明
singleton	在 Spring IoC 容器中仅存在一个 Bean 实例，Bean 以单例方式存在，默认值
prototype	每次从容器中调用 Bean 时，都返回一个新的实例，即每次调用 getBean() 时，相当于执行 new XxxBean()
request	每次 HTTP 请求都会创建一个新的 Bean，该作用域仅适用于 WebApplicationContext 环境
session	同一个 HTTP Session 共享一个 Bean，不同 Session 使用不同 Bean，仅适用于 WebApplicationContext 环境
globalSession	一般用于 Portlet 应用环境，该作用域仅适用于 WebApplicationContext 环境

五种作用域中，request、session 和 global session 三种作用域仅在基于 web 的应用中使用（不必关心你所采用的是什么 web 应用框架），只能用在基于 web 的 Spring ApplicationContext 环境。

5.2.1 singleton——唯一 bean 实例

当一个 bean 的作用域为 singleton，那么 Spring IoC 容器中只会存在一个共享的 bean 实例，并且所有对 bean 的请求，只要 id 与该 bean 定义相匹配，则只会返回 bean 的同一实例。 singleton 是单例类型（对应于单例模式），就是在创建起容器时就同时自动创建了一个 bean 的对象，不管你是否使用，但我们可以指定 Bean 节点的 `lazy-init="true"` 来延迟初始化 bean，这时候，只有在第一次获取 bean

时才会初始化bean，即第一次请求该bean时才初始化。每次获取到的对象都是同一个对象。注意，singleton 作用域是Spring中的缺省作用域。要在XML中将 bean 定义成 singleton，可以这样配置：

```
<bean id="ServiceImpl" class="cn.csdn.service.ServiceImpl" scope="singleton">
```

也可以通过 @scope 注解（它可以显示指定bean的作用范围。）的方式

```
@Service  
@Scope("singleton")  
public class ServiceImpl {  
}
```

5.2.2 prototype——每次请求都会创建一个新的 bean 实例

当一个bean的作用域为 prototype，表示一个 bean 定义对应多个对象实例。prototype 作用域的 bean 会导致在每次对该 bean 请求（将其注入到另一个 bean 中，或者以程序的方式调用容器的 getBean() 方法）时都会创建一个新的 bean 实例。prototype 是原型类型，它在我们创建容器的时候并没有实例化，而是当我们获取bean的时候才会去创建一个对象，而且我们每次获取到的对象都不是同一个对象。根据经验，对有状态的 bean 应该使用 prototype 作用域，而对无状态的 bean 则应该使用 singleton 作用域。在 XML 中将 bean 定义成 prototype，可以这样配置：

```
<bean id="account" class="com.foo.DefaultAccount" scope="prototype"/>  
或者  
<bean id="account" class="com.foo.DefaultAccount" singleton="false"/>
```

通过 @scope 注解的方式实现就不做演示了。

5.2.3 request——每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP request内有效

request只适用于Web程序，每一次 HTTP 请求都会产生一个新的bean，同时该bean仅在当前HTTP request内有效，当请求结束后，该对象的生命周期即告结束。在 XML 中将 bean 定义成 request，可以这样配置：

```
<bean id="loginAction" class="cn.csdn.LoginAction" scope="request"/>
```

5.2.4 session——每一次HTTP请求都会产生一个新的 bean，该bean仅在当前 HTTP session 内有效

session只适用于Web程序，session 作用域表示该针对每一次 HTTP 请求都会产生一个新的 bean，同时该 bean 仅在当前 HTTP session 内有效。与request作用域一样，可以根据需要放心的更改所创建实例的内部状态，而别的 HTTP session 中根据 userPreferences 创建的实例，将不会看到这些特定于某个 HTTP session 的状态变化。当HTTP session最终被废弃的时候，在该HTTP session作用域内的bean也会被废弃掉。

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
```

5.2.5 globalSession

global session 作用域类似于标准的 HTTP session 作用域，不过仅仅在基于 portlet 的 web 应用中才有意义。Portlet 规范定义了全局 Session 的概念，它被所有构成某个 portlet web 应用的各种不同的 portlet 所共享。在global session 作用域中定义的 bean 被限定于全局portlet Session 的生命周期范围内。

```
<bean id="user" class="com.foo.Preferences" scope="globalSession"/>
```

5.3 bean的生命周期

Spring Bean是Spring应用中最最重要的部分了。所以来看看Spring容器在初始化一个bean的时候会做那些事情，顺序是怎样的，在容器关闭的时候，又会做哪些事情。

spring版本：4.2.3.RELEASE

鉴于Spring源码是用gradle构建的，我也决定舍弃我大maven，尝试下洪菊推荐过的gradle。运行beanLifeCycle模块下的junit test即可在控制台看到如下输出，可以清楚了解Spring容器在创建，初始化和销毁Bean的时候依次做了那些事情。

Spring容器初始化

=====

调用GiraffeService无参构造函数

GiraffeService中利用set方法设置属性值

调用setBeanName:: Bean Name defined in context=giraffeService

调用setBeanClassLoader,ClassLoader Name = sun.misc.Launcher\$AppClassLoader

调用setBeanFactory, setBeanFactory:: giraffe bean singleton=true

调用setEnvironment

调用setResourceLoader:: Resource File Name=spring-beans.xml

调用setApplicationEventPublisher

调用setApplicationContext:: Bean Definition Names=[giraffeService,
org.springframework.context.annotation.CommonAnnotationBeanPostProcessor#0,
com.giraffe.spring.service.GiraffeServicePostProcessor#0]

执行BeanPostProcessor的postProcessBeforeInitialization方法,beanName=giraffeService

调用PostConstruct注解标注的方法

执行InitializingBean接口的afterPropertiesSet方法

执行配置的init-method

执行BeanPostProcessor的postProcessAfterInitialization方法,beanName=giraffeService

Spring容器初始化完毕

=====

从容器中获取Bean

giraffe Name=李光洙

=====

调用preDestroy注解标注的方法

执行DisposableBean接口的destroy方法

执行配置的destroy-method

Spring容器关闭

先来看看，Spring在Bean从创建到销毁的生命周期中可能做得事情。

5.3.1 initialization 和 destroy

有时我们需要在Bean属性值set好之后和Bean销毁之前做一些事情，比如检查Bean中某个属性是否被正常的设置好值了。Spring框架提供了多种方法让我们可以在Spring Bean的生命周期中执行initialization和pre-destroy方法。

1.实现 InitializingBean 和 DisposableBean 接口

这两个接口都只包含一个方法。通过实现InitializingBean接口的afterPropertiesSet()方法可以在Bean属性值设置好之后做一些操作，实现DisposableBean接口的destroy()方法可以在销毁Bean之前做一些操作。

例子如下：

```
public class Giraffeservice implements InitializingBean,DisposableBean {  
    @Override  
    public void afterPropertiesSet() throws Exception {  
        System.out.println("执行InitializingBean接口的afterPropertiesSet方法");  
    }  
    @Override  
    public void destroy() throws Exception {  
        System.out.println("执行DisposableBean接口的destroy方法");  
    }  
}
```

这种方法比较简单，但是不建议使用。因为这样会将Bean的实现和Spring框架耦合在一起。

2.在bean的配置文件中指定init-method和destroy-method方法

Spring允许我们创建自己的 init 方法和 destroy 方法，只要在 Bean 的配置文件中指定 init-method 和 destroy-method 的值就可以在 Bean 初始化时和销毁之前执行一些操作。

例子如下：

```
public class Giraffeservice {  
    //通过<bean>的destroy-method属性指定的销毁方法  
    public void destroyMethod() throws Exception {  
        System.out.println("执行配置的destroy-method");  
    }  
    //通过<bean>的init-method属性指定的初始化方法  
    public void initMethod() throws Exception {  
        System.out.println("执行配置的init-method");  
    }  
}
```

配置文件中的配置：

```
<bean name="giraffeservice" class="com.giraffe.spring.service.Giraffeservice"  
init-method="initMethod" destroy-method="destroyMethod">  
</bean>
```

需要注意的是自定义的init-method和post-method方法可以抛异常但是不能有参数。

这种方式比较推荐，因为可以自己创建方法，无需将Bean的实现直接依赖于spring的框架。

3.使用@PostConstruct和@PreDestroy注解

除了xml配置的方式，Spring 也支持用 `@PostConstruct` 和 `@PreDestroy` 注解来指定 `init` 和 `destroy` 方法。这两个注解均在 `javax.annotation` 包中。为了注解可以生效，需要在配置文件中定义 `org.springframework.context.annotation.CommonAnnotationBeanPostProcessor` 或 `context:annotation-config`

例子如下：

```

public class Giraffeservice {
    @PostConstruct
    public void initPostConstruct(){
        System.out.println("执行PostConstruct注解标注的方法");
    }
    @PreDestroy
    public void preDestroy(){
        System.out.println("执行preDestroy注解标注的方法");
    }
}

```

配置文件：

```

<bean
    class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"
/>

```

5.3.2 实现*Aware接口 在Bean中使用Spring框架的一些对象

有些时候我们需要在 Bean 的初始化中使用 Spring 框架自身的一些对象来执行一些操作，比如获取 ServletContext 的一些参数，获取 ApplicationContext 中的 BeanDefinition 的名字，获取 Bean 在容器中的名字等等。为了让 Bean 可以获取到框架自身的一些对象，Spring 提供了一组名为*Aware的接口。

这些接口均继承于 `org.springframework.beans.factory.Aware` 标记接口，并提供一个将由 Bean 实现的 `set*` 方法，Spring 通过基于 setter 的依赖注入方式使相应的对象可以被 Bean 使用。

网上说，这些接口是利用观察者模式实现的，类似于 servlet listeners，目前还不明白，不过这也不在本文的讨论范围内。

介绍一些重要的 Aware 接口：

- **ApplicationContextAware**: 获得 ApplicationContext 对象，可以用来获取所有 Bean definition 的名字。
- **BeanFactoryAware**: 获得 BeanFactory 对象，可以用来检测 Bean 的作用域。
- **BeanNameAware**: 获得 Bean 在配置文件中定义的名字。
- **ResourceLoaderAware**: 获得 ResourceLoader 对象，可以获得 classpath 中某个文件。
- **ServletContextAware**: 在一个 MVC 应用中可以获取 ServletContext 对象，可以读取 context 中的参数。
- **ServletConfigAware**: 在一个 MVC 应用中可以获取 ServletConfig 对象，可以读取 config 中的参数。

```

public class Giraffeservice implements ApplicationContextAware,
    ApplicationEventPublisherAware, BeanClassLoaderAware, BeanFactoryAware,
    BeanNameAware, EnvironmentAware, ImportAware, ResourceLoaderAware{
    @Override
    public void setBeanClassLoader(ClassLoader classLoader) {
        System.out.println("执行setBeanClassLoader, ClassLoader Name = " +
classLoader.getClass().getName());
    }
    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        System.out.println("执行setBeanFactory, setBeanFactory:: giraffe bean
singleton=" + beanFactory.isSingleton("giraffeservice"));
    }
}

```

```
@Override
public void setBeanName(String s) {
    System.out.println("执行setBeanName:: Bean Name defined in context="
        + s);
}
@Override
public void setApplicationContext(ApplicationContext applicationContext)
throws BeansException {
    System.out.println("执行setApplicationContext:: Bean Definition Names="
        + Arrays.toString(applicationContext.getBeanDefinitionNames()));
}
@Override
public void setApplicationEventPublisher(ApplicationEventPublisher
applicationEventPublisher) {
    System.out.println("执行setApplicationEventPublisher");
}
@Override
public void setEnvironment(Environment environment) {
    System.out.println("执行setEnvironment");
}
@Override
public void setResourceLoader(ResourceLoader resourceLoader) {
    Resource resource = resourceLoader.getResource("classpath:spring-
beans.xml");
    System.out.println("执行setResourceLoader:: Resource File Name="
        + resource.getFilename());
}
@Override
public void setImportMetadata(AnnotationMetadata annotationMetadata) {
    System.out.println("执行setImportMetadata");
}
}
```

5.3.3 BeanPostProcessor

上面的*Aware接口是针对某个实现这些接口的Bean定制初始化的过程，Spring同样可以针对容器中的所有Bean，或者某些Bean定制初始化过程，只需提供一个实现BeanPostProcessor接口的类即可。该接口中包含两个方法，postProcessBeforeInitialization和postProcessAfterInitialization。postProcessBeforeInitialization方法会在容器中的Bean初始化之前执行，postProcessAfterInitialization方法在容器中的Bean初始化之后执行。

例子如下：

```

public class CustomerBeanPostProcessor implements BeanPostProcessor {
    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException {
        System.out.println("执行BeanPostProcessor的
postProcessBeforeInitialization方法,beanName=" + beanName);
        return bean;
    }
    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException {
        System.out.println("执行BeanPostProcessor的postProcessAfterInitialization
方法,beanName=" + beanName);
        return bean;
    }
}

```

要将BeanPostProcessor的Bean像其他Bean一样定义在配置文件中

```
<bean class="com.giraffe.spring.service.CustomerBeanPostProcessor"/>
```

5.3.4 总结

所以。。。结合第一节控制台输出的内容，Spring Bean的生命周期是这样纸的：

- Bean容器找到配置文件中 Spring Bean 的定义。
- Bean容器利用Java Reflection API创建一个Bean的实例。
- 如果涉及到一些属性值 利用set方法设置一些属性值。
- 如果Bean实现了BeanNameAware接口，调用setBeanName()方法，传入Bean的名字。
- 如果Bean实现了BeanClassLoaderAware接口，调用setBeanClassLoader()方法，传入 ClassLoader对象的实例。
- 如果Bean实现了BeanFactoryAware接口，调用setBeanFactory()方法，传入BeanFactory对象的实例。
- 与上面的类似，如果实现了其他*Aware接口，就调用相应的方法。
- 如果有和加载这个Bean的Spring容器相关的BeanPostProcessor对象，执行 postProcessBeforeInitialization()方法
- 如果Bean实现了InitializingBean接口，执行afterPropertiesSet()方法。
- 如果Bean在配置文件中的定义包含init-method属性，执行指定的方法。
- 如果有和加载这个Bean的Spring容器相关的BeanPostProcessor对象，执行 postProcessAfterInitialization()方法
- 当要销毁Bean的时候，如果Bean实现了DisposableBean接口，执行destroy()方法。
- 当要销毁Bean的时候，如果Bean在配置文件中的定义包含destroy-method属性，执行指定的方法。

小插曲：

更多阿里、腾讯、美团、京东等一线互联网大厂Java面试真题；包含：基础、并发、锁、JVM、设计模式、数据结构、反射/IO、数据库、Redis、Spring、消息队列、分布式、Zookeeper、Dubbo、Mybatis、Maven、面经等。

更多Java程序员技术进阶小技巧；例如高效学习（如何学习和阅读代码、面对枯燥和量大的知识）高效沟通（沟通方式及技巧、沟通技术）

更多Java大牛分享的一些职业生涯分享文档

请添加微信：jiang10086a 免费获取

比你优秀的对手在学习，你的仇人在磨刀，你的闺蜜在减肥，隔壁老王在练腰，我们必须不断学习，否则我们将被学习者超越！

趁年轻，使劲拼，给未来的自己一个交代！

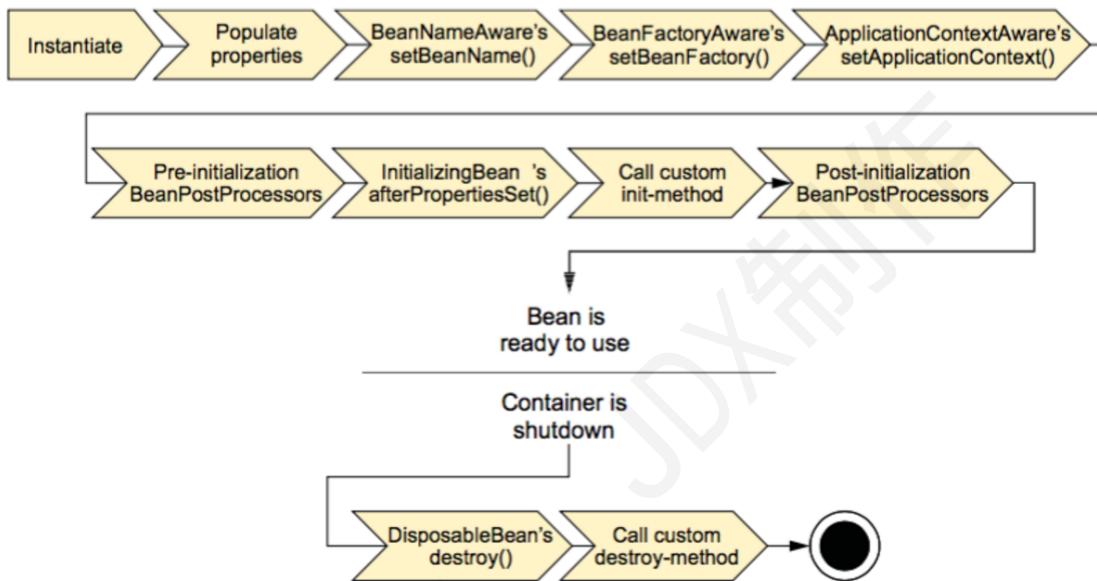
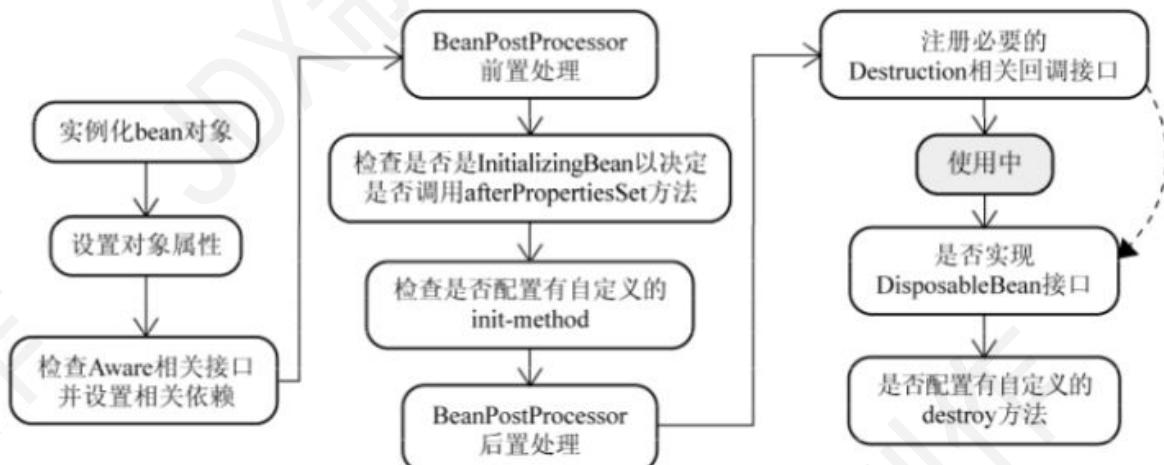


Figure 1.5 A bean goes through several steps between creation and destruction in the Spring container. Each step is an opportunity to customize how the bean is managed in Spring.

与之比较类似的中文版本：



其实很多时候我们并不会真的去实现上面说描述的那些接口，那么下面我们就除去那些接口，针对bean的单例和非单例来描述下bean的生命周期：

5.3.5 单例管理的对象

当scope="singleton"，即默认情况下，会在启动容器时（即实例化容器时）时实例化。但我们可以指定Bean节点的lazy-init="true"来延迟初始化bean，这时候，只有在第一次获取bean时才会初始化bean，即第一次请求该bean时才初始化。如下配置：

```
<bean id="ServiceImpl" class="cn.csdn.service.ServiceImpl" lazy-init="true"/>
```

如果想对所有的默认单例bean都应用延迟初始化，可以在根节点beans设置default-lazy-init属性为true，如下所示：

```
<beans default-lazy-init="true" ...>
```

默认情况下，Spring 在读取 xml 文件的时候，就会创建对象。在创建对象的时候先调用构造器，然后调用 init-method 属性值中所指定的方法。对象在被销毁的时候，会调用 destroy-method 属性值中所指定的方法（例如调用Container.destroy()方法的时候）。写一个测试类，代码如下：

```
public class LifeBean {
    private String name;

    public LifeBean(){
        System.out.println("LifeBean()构造函数");
    }
    public String getName(){
        return name;
    }

    public void setName(String name) {
        System.out.println("setName()");
        this.name = name;
    }

    public void init(){
        System.out.println("this is init of lifeBean");
    }

    public void destory(){
        System.out.println("this is destory of lifeBean " + this);
    }
}
```

life.xml配置如下：

```
<bean id="life_singleton" class="com.bean.LifeBean" scope="singleton"
      init-method="init" destroy-method="destory" lazy-init="true"/>
```

测试代码：

```
public class LifeTest {
    @Test
    public void test(){
        AbstractApplicationContext container =
            new ClassPathXmlApplicationContext("life.xml");
        LifeBean life1 = (LifeBean)container.getBean("life");
        System.out.println(life1);
        container.close();
    }
}
```

运行结果：

```
LifeBean()构造函数
this is init of lifeBean
com.bean.LifeBean@573f2bb1
.....
this is destory of lifeBean com.bean.LifeBean@573f2bb1
```

5.3.6 非单例管理的对象

当 scope="prototype" 时，容器也会延迟初始化 bean，Spring 读取xml 文件的时候，并不会立刻创建对象，而是在第一次请求该 bean 时才初始化（如调用getBean方法时）。在第一次请求每一个 prototype 的bean 时，Spring容器都会调用其构造器创建这个对象，然后调用 init-method 属性值中所指定的方法。对象销毁的时候，Spring 容器不会帮我们调用任何方法，因为是非单例，这个类型的对象有很多个，Spring容器一旦把这个对象交给你之后，就不再管理这个对象了。

为了测试prototype bean的生命周期life.xml配置如下：

```
<bean id="life_prototype" class="com.bean.LifeBean" scope="prototype" init-method="init" destroy-method="destory"/>
```

测试程序：

```
public class LifeTest {  
    @Test  
    public void test() {  
        AbstractApplicationContext container = new  
        ClassPathXmlApplicationContext("life.xml");  
        LifeBean life1 = (LifeBean)container.getBean("life_singleton");  
        System.out.println(life1);  
  
        LifeBean life3 = (LifeBean)container.getBean("life_prototype");  
        System.out.println(life3);  
        container.close();  
    }  
}
```

运行结果：

```
LifeBean()构造函数  
this is init of lifeBean  
com.bean.LifeBean@573f2bb1  
LifeBean()构造函数  
this is init of lifeBean  
com.bean.LifeBean@5ae9a829  
.....  
this is destory of lifeBean com.bean.LifeBean@573f2bb1
```

可以发现，对于作用域为 prototype 的 bean，其 destroy 方法并没有被调用。如果 bean 的 scope 设为prototype时，当容器关闭时， destroy 方法不会被调用。对于 prototype 作用域的 bean，有一点非常重要，那就是 Spring不能对一个 prototype bean 的整个生命周期负责：容器在初始化、配置、装饰或者是装配完一个prototype实例后，将它交给客户端，随后就对该prototype实例不闻不问了。不管何种作用域，容器都会调用所有对象的初始化生命周期回调方法。但对prototype而言，任何配置好的析构生命周期回调方法都将不会被调用。清除prototype作用域的对象并释放任何prototype bean所持有的昂贵资源，都是客户端代码的职责（让Spring容器释放被prototype作用域bean占用资源的一种可行方式是，通过使用bean的后置处理器，该处理器持有要被清除的bean的引用）。谈及 prototype作用域的bean时，在某些方面你可以将Spring容器的角色看作是Java new操作的替代者，任何迟于该时间点的生命周期事宜都得交由客户端来处理。

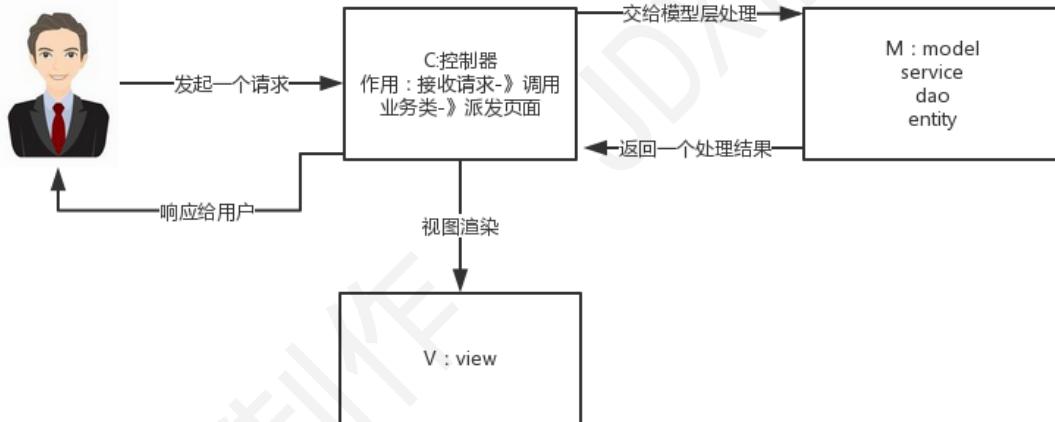
Spring 容器可以管理 singleton 作用域下 bean 的生命周期，在此作用域下，Spring 能够精确地知道 bean何时被创建，何时初始化完成，以及何时被销毁。而对于 prototype 作用域的bean，Spring只负责创建，当容器创建了 bean 的实例后，bean 的实例就交给了客户端的代码管理，Spring容器将不再跟踪其生命周期，并且不会管理那些被配置成prototype作用域的bean的生命周期。

6. SpringMVC 工作原理详解

6.1 先来看一下什么是 MVC 模式

MVC 是一种设计模式.

MVC 的原理图如下:



6.2 SpringMVC 简单介绍

SpringMVC 框架是以请求为驱动，围绕 Servlet 设计，将请求发给控制器，然后通过模型对象，分派器来展示请求结果视图。其中核心类是 DispatcherServlet，它是一个 Servlet，顶层是实现的Servlet接口。

6.3 SpringMVC 使用

需要在 web.xml 中配置 DispatcherServlet。并且需要配置 Spring 监听器ContextLoaderListener

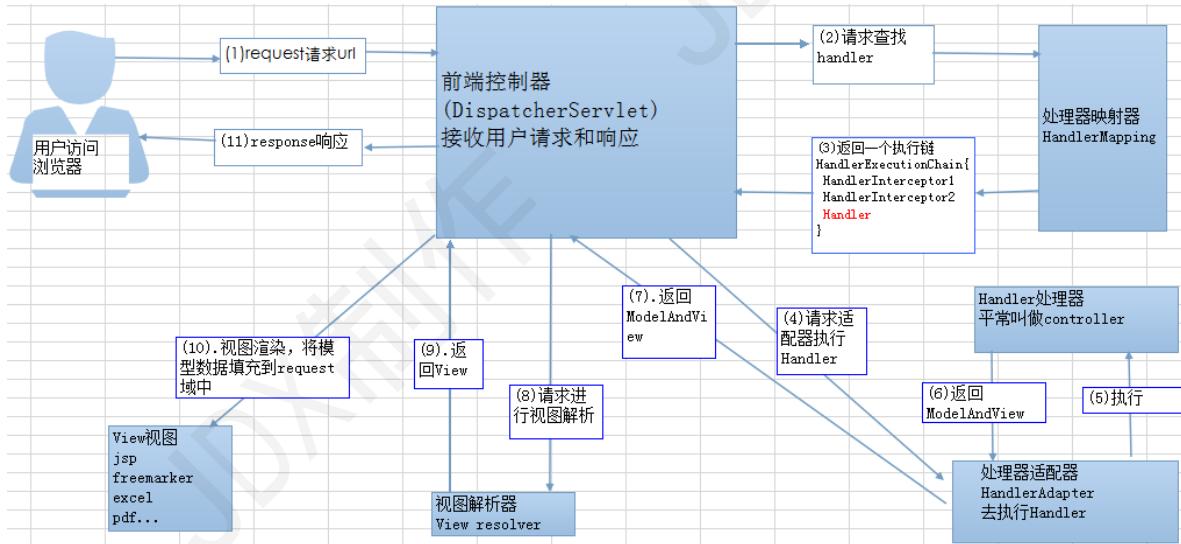
```
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
<servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <!-- 如果不设置init-param标签，则必须在/WEB-INF/下创建xxx-servlet.xml文件，其中xxx是servlet-name中配置的名称。 -->
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring/springmvc-servlet.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

6.4 SpringMVC 工作原理 (重要)

简单来说：

客户端发送请求-> 前端控制器 DispatcherServlet 接受客户端请求 -> 找到处理器映射 HandlerMapping 解析请求对应的 Handler-> HandlerAdapter 会根据 Handler 来调用真正的处理器来处理请求，并处理相应的业务逻辑 -> 处理器返回一个模型视图 ModelAndView -> 视图解析器进行解析 -> 返回一个视图对象->前端控制器 DispatcherServlet 渲染数据 (Model) -> 将得到视图对象返回给用户

如下图所示：



上图的一个笔误的小问题：Spring MVC 的入口函数也就是前端控制器 DispatcherServlet 的作用是接收请求，响应结果。

流程说明（重要）：

- (1) 客户端（浏览器）发送请求，直接请求到 DispatcherServlet。
- (2) DispatcherServlet 根据请求信息调用 HandlerMapping，解析请求对应的 Handler。
- (3) 解析到对应的 Handler（也就是我们平常说的 Controller 控制器）后，开始由 HandlerAdapter 适配器处理。
- (4) HandlerAdapter 会根据 Handler 来调用真正的处理器来处理请求，并处理相应的业务逻辑。
- (5) 处理器处理完业务后，会返回一个 ModelAndView 对象，Model 是返回的数据对象，View 是个逻辑上的 View。
- (6) ViewResolver 会根据逻辑 View 查找实际的 View。
- (7) DispatcherServlet 把返回的 Model 传给 View（视图渲染）。
- (8) 把 View 返回给请求者（浏览器）

6.5 SpringMVC 重要组件说明

1、前端控制器DispatcherServlet (不需要工程师开发), 由框架提供 (重要)

作用：Spring MVC 的入口函数。接收请求，响应结果，相当于转发器，中央处理器。有了 DispatcherServlet 减少了其它组件之间的耦合度。用户请求到达前端控制器，它就相当于mvc模式中的c，DispatcherServlet是整个流程控制的中心，由它调用其它组件处理用户的请求，DispatcherServlet的存在降低了组件之间的耦合性。

2、处理器映射器HandlerMapping(不需要工程师开发),由框架提供

作用：根据请求的url查找Handler。HandlerMapping负责根据用户请求找到Handler即处理器（Controller），SpringMVC提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

3、处理器适配器HandlerAdapter

作用：按照特定规则（HandlerAdapter要求的规则）去执行Handler
通过HandlerAdapter对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。

4、处理器Handler(需要工程师开发)

注意：编写Handler时按照HandlerAdapter的要求去做，这样适配器才可以去正确执行Handler
Handler 是继DispatcherServlet前端控制器的后端控制器，在DispatcherServlet的控制下Handler对具体的用户请求进行处理。
由于Handler涉及到具体的用户业务请求，所以一般情况需要工程师根据业务需求开发Handler。

5、视图解析器View resolver(不需要工程师开发),由框架提供

作用：进行视图解析，根据逻辑视图名解析成真正的视图（view）
View Resolver负责将处理结果生成View视图，View Resolver首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成View视图对象，最后对View进行渲染将处理结果通过页面展示给用户。
springmvc框架提供了很多的View视图类型，包括：jstlView、freemarkerView、pdfView等。
一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由工程师根据业务需求开发具体的页面。

6、视图View(需要工程师开发)

View是一个接口，实现类支持不同的View类型（jsp、freemarker、pdf...）

注意：处理器Handler（也就是我们平常说的Controller控制器）以及视图层view都是需要我们自己手动开发的。其他的一些组件比如：前端控制器DispatcherServlet、处理器映射器HandlerMapping、处理器适配器HandlerAdapter等等都是框架提供给我们的，不需要自己手动开发。

6.6 DispatcherServlet详细解析

首先看下源码：

```
package org.springframework.web.servlet;

@SuppressWarnings("serial")
public class DispatcherServlet extends FrameworkServlet {

    public static final String MULTIPART_RESOLVER_BEAN_NAME =
    "multipartResolver";
    public static final String LOCALE_RESOLVER_BEAN_NAME = "localeResolver";
    public static final String THEME_RESOLVER_BEAN_NAME = "themeResolver";
    public static final String HANDLER_MAPPING_BEAN_NAME = "handlerMapping";
    public static final String HANDLER_ADAPTER_BEAN_NAME = "handlerAdapter";
    public static final String HANDLER_EXCEPTION_RESOLVER_BEAN_NAME =
    "handlerExceptionResolver";
```

```
public static final String REQUEST_TO_VIEW_NAME_TRANSLATOR_BEAN_NAME =
"viewNameTranslator";
public static final String VIEW_RESOLVER_BEAN_NAME = "viewResolver";
public static final String FLASH_MAP_MANAGER_BEAN_NAME = "flashMapManager";
public static final String WEB_APPLICATION_CONTEXT_ATTRIBUTE =
DispatcherServlet.class.getName() + ".CONTEXT";
public static final String LOCALE_RESOLVER_ATTRIBUTE =
DispatcherServlet.class.getName() + ".LOCALE_RESOLVER";
public static final String THEME_RESOLVER_ATTRIBUTE =
DispatcherServlet.class.getName() + ".THEME_RESOLVER";
public static final String THEME_SOURCE_ATTRIBUTE =
DispatcherServlet.class.getName() + ".THEME_SOURCE";
public static final String INPUT_FLASH_MAP_ATTRIBUTE =
DispatcherServlet.class.getName() + ".INPUT_FLASH_MAP";
public static final String OUTPUT_FLASH_MAP_ATTRIBUTE =
DispatcherServlet.class.getName() + ".OUTPUT_FLASH_MAP";
public static final String FLASH_MAP_MANAGER_ATTRIBUTE =
DispatcherServlet.class.getName() + ".FLASH_MAP_MANAGER";
public static final String EXCEPTION_ATTRIBUTE =
DispatcherServlet.class.getName() + ".EXCEPTION";
public static final String PAGE_NOT_FOUND_LOG_CATEGORY =
"org.springframework.web.servlet.PageNotFound";
private static final String DEFAULT_STRATEGIES_PATH =
"DispatcherServlet.properties";
protected static final Log pageNotFoundLogger =
LogFactory.getLog(PAGE_NOT_FOUND_LOG_CATEGORY);
private static final Properties defaultStrategies;
static {
    try {
        ClassPathResource resource = new
ClassPathResource(DEFAULT_STRATEGIES_PATH, DispatcherServlet.class);
        defaultStrategies = PropertiesLoaderUtils.loadProperties(resource);
    }
    catch (IOException ex) {
        throw new IllegalStateException("Could not load
'DispatcherServlet.properties': " + ex.getMessage());
    }
}

/** Detect all HandlerMappings or just expect "handlerMapping" bean? */
private boolean detectAllHandlerMappings = true;

/** Detect all HandlerAdapters or just expect "handlerAdapter" bean? */
private boolean detectAllHandlerAdapters = true;

/** Detect all HandlerExceptionResolvers or just expect
"handlerExceptionResolver" bean? */
private boolean detectAllHandlerExceptionResolvers = true;

/** Detect all ViewResolvers or just expect "viewResolver" bean? */
private boolean detectAllViewResolvers = true;

/** Throw a NoHandlerFoundException if no Handler was found to process this
request? */
private boolean throwExceptionIfNoHandlerFound = false;

/** Perform cleanup of request attributes after include request? */
private boolean cleanupAfterInclude = true;
```

```

/** MultipartResolver used by this servlet */
private MultipartResolver multipartResolver;

/** LocaleResolver used by this servlet */
private LocaleResolver localeResolver;

/** ThemeResolver used by this servlet */
private ThemeResolver themeResolver;

/** List of HandlerMappings used by this servlet */
private List<HandlerMapping> handlerMappings;

/** List of HandlerAdapters used by this servlet */
private List<HandlerAdapter> handlerAdapters;

/** List of HandlerExceptionResolvers used by this servlet */
private List<HandlerExceptionResolver> handlerExceptionResolvers;

/** RequestToViewNameTranslator used by this servlet */
private RequestToViewNameTranslator viewNameTranslator;

private FlashMapManager flashMapManager;

/** List of ViewResolvers used by this servlet */
private List<ViewResolver> viewResolvers;

public DispatcherServlet() {
    super();
}

public DispatcherServlet(ApplicationContext webApplicationContext) {
    super(webApplicationContext);
}

@Override
protected void onRefresh(ApplicationContext context) {
    initStrategies(context);
}

protected void initStrategies(ApplicationContext context) {
    initMultipartResolver(context);
    initLocaleResolver(context);
    initThemeResolver(context);
    initHandlerMappings(context);
    initHandlerAdapters(context);
    initHandlerExceptionResolvers(context);
    initRequestToViewNameTranslator(context);
    initViewResolvers(context);
    initFlashMapManager(context);
}
}

```

DispatcherServlet类中的属性beans:

- HandlerMapping: 用于handlers映射请求和一系列的对于拦截器的前处理和后处理, 大部分用@Controller注解。

- HandlerAdapter：帮助DispatcherServlet处理映射请求处理器的适配器，而不用考虑实际调用的是哪个处理器。---
- ViewResolver：根据实际配置解析实际的View类型。
- ThemeResolver：解决Web应用程序可以使用的主题，例如提供个性化布局。
- MultipartResolver：解析多部分请求，以支持从HTML表单上传文件。-
- FlashMapManager：存储并检索可用于将一个请求属性传递到另一个请求的input和output的FlashMap，通常用于重定向。

在Web MVC框架中，每个DispatcherServlet都拥自己的WebApplicationContext，它继承了ApplicationContext。WebApplicationContext包含了其上下文和Servlet实例之间共享的所有基础框架beans。

HandlerMapping

Type hierarchy of 'org.springframework.web.servlet.HandlerMapping':

```

    ▼ ⓘ HandlerMapping - org.springframework.web.servlet
        ▼ ⓘ AbstractHandlerMapping - org.springframework.web.servlet.handler
            > ⓘ AbstractHandlerMethodMapping<T> - org.springframework.web.servlet.handler
            ▼ ⓘ AbstractUrlHandlerMapping - org.springframework.web.servlet.handler
                ▼ ⓘ AbstractDetectingUrlHandlerMapping - org.springframework.web.servlet.handler
                    ▼ ⓘ AbstractControllerUrlHandlerMapping - org.springframework.web.servlet.mvc
                        ⓘ ControllerBeanNameHandlerMapping - org.springframework.web.servlet.mvc
                        ⓘ ControllerClassNameHandlerMapping - org.springframework.web.servlet.mvc
                        ⓘ BeanNameUrlHandlerMapping - org.springframework.web.servlet.handler
                        ⓘ DefaultAnnotationHandlerMapping - org.springframework.web.servlet.mvc.annotation
                    ▼ ⓘ SimpleUrlHandlerMapping - org.springframework.web.servlet.handler
            ⓘ EmptyHandlerMapping - org.springframework.web.config.annotation.WebM

```

<https://blog.csdn.net/yanweihp>

HandlerMapping接口处理请求的映射HandlerMapping接口的实现类：

- SimpleUrlHandlerMapping类通过配置文件把URL映射到Controller类。
- DefaultAnnotationHandlerMapping类通过注解把URL映射到Controller类。

HandlerAdapter

```

    ▼ ⓘ HandlerAdapter - org.springframework.web.servlet
        ▼ ⓘ AbstractHandlerMethodAdapter - org.springframework.web.servlet.mvc.method
            ⓘ RequestMappingHandlerAdapter - org.springframework.web.servlet.mvc.method.annotation
            ⓘ AnnotationMethodHandlerAdapter - org.springframework.web.servlet.mvc.annotation
            ⓘ HttpRequestHandlerAdapter - org.springframework.web.servlet.mvc
            ⓘ SimpleControllerHandlerAdapter - org.springframework.web.servlet.mvc
            ⓘ SimpleServletHandlerAdapter - org.springframework.web.servlet.handler

```

<https://blog.csdn.net/yanweihp>

HandlerAdapter接口-处理请求映射

AnnotationMethodHandlerAdapter：通过注解，把请求URL映射到Controller类的方法上。

HandlerExceptionResolver

Type hierarchy of 'org.springframework.web.servlet.HandlerExceptionResolver':

- HandlerExceptionResolver - org.springframework.web.servlet
- AbstractHandlerExceptionResolver - org.springframework.web.servlet.handler
- AbstractHandlerMethodExceptionResolver - org.springframework.web.servlet.handler
- ExceptionHandlerExceptionResolver - org.springframework.web.servlet.mvc.method
- AnnotationMethodHandlerExceptionResolver - org.springframework.web.servlet.mvc
- DefaultHandlerExceptionResolver - org.springframework.web.servlet.mvc.support
- ResponseStatusExceptionResolver - org.springframework.web.servlet.mvc.annotation
- SimpleMappingExceptionResolver - org.springframework.web.servlet.handler
- HandlerExceptionResolverComposite - org.springframework.web.servlet.handler
- MyExceptionHandler - com.flight.manager.merchant.handler
- MyExceptionHandler - com.flight.manager.handler

<https://blog.csdn.net/yanweihu>

HandlerExceptionResolver接口-异常处理接口

- SimpleMappingExceptionResolver通过配置文件进行异常处理。
- AnnotationMethodHandlerExceptionResolver：通过注解进行异常处理。

ViewResolver

- ViewResolver - org.springframework.web.servlet
- AbstractCachingViewResolver - org.springframework.web.servlet.view
- ResourceBundleViewResolver - org.springframework.web.servlet.view
- UrlBasedViewResolver - org.springframework.web.servlet.view
- AbstractTemplateViewResolver - org.springframework.web.servlet.view
- FreeMarkerViewResolver - org.springframework.web.servlet.view.freemarker
- GroovyMarkupViewResolver - org.springframework.web.servlet.view.groovy
- VelocityViewResolver - org.springframework.web.servlet.view.velocity
- VelocityLayoutViewResolver - org.springframework.web.servlet.view.velocity
- InternalResourceViewResolver - org.springframework.web.servlet.view
- JasperReportsViewResolver - org.springframework.web.servlet.view.jasperreports
- ScriptTemplateViewResolver - org.springframework.web.servlet.view.script
- TilesViewResolver - org.springframework.web.servlet.view.tiles3
- TilesViewResolver - org.springframework.web.servlet.view.tiles2
- XsltViewResolver - org.springframework.web.servlet.view.xslt
- XmlViewResolver - org.springframework.web.servlet.view
- BeanNameViewResolver - org.springframework.web.servlet.view
- ContentNegotiatingViewResolver - org.springframework.web.servlet.view
- StaticViewResolver - org.springframework.test.web.servlet.setup.StandaloneMockMvcBuilder
- ViewResolverComposite - org.springframework.web.servlet.view

<https://blog.csdn.net/yanweihu>

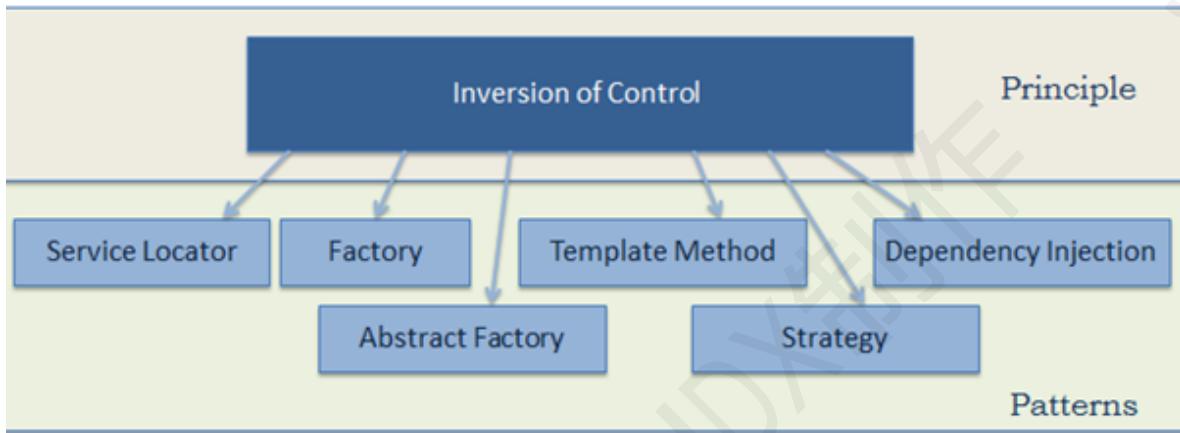
ViewResolver接口解析View视图。

UrlBasedViewResolver类 通过配置文件，把一个视图名交给到一个View来处理。

7. Spring中都用到了那些设计模式？

7.1 控制反转(IoC)和依赖注入(DI)

IoC(Inversion of Control,控制翻转) 是Spring 中一个非常非常重要的概念，它不是什么技术，而是一种解耦的设计思想。它的主要目的是借助于“第三方”(Spring 中的 IOC 容器) 实现具有依赖关系的对象之间的解耦(IOC容易管理对象，你只管使用即可)，从而降低代码之间的耦合度。**IOC是一个原则，而不是一个模式，以下模式（但不限于）实现了IoC原则。**



Spring IOC 容器就像是一个工厂一样，当我们需要创建一个对象的时候，只需要配置好配置文件/注解即可，完全不用考虑对象是如何被创建出来的。IOC 容器负责创建对象，将对象连接在一起，配置这些对象，并从创建中处理这些对象的整个生命周期，直到它们被完全销毁。

在实际项目中一个 Service 类如果有几百甚至上千个类作为它的底层，我们需要实例化这个 Service，你可能要每次都要搞清这个 Service 所有底层类的构造函数，这可能会把人逼疯。如果利用 IOC 的话，你只需要配置好，然后在需要的地方引用就行了，这大大增加了项目的可维护性且降低了开发难度。

控制翻转怎么理解呢？举个例子：“对象a 依赖了对象 b，当对象 a 需要使用 对象 b 的时候必须自己去创建。但是当系统引入了 IOC 容器后，对象a 和对象 b 之前就失去了直接的联系。这个时候，当对象 a 需要使用 对象 b 的时候，我们可以指定 IOC 容器去创建一个对象b注入到对象 a 中”。对象 a 获得依赖对象 b 的过程，由主动行为变为了被动行为，控制权翻转，这就是控制反转名字的由来。

DI(Dependency Inject, 依赖注入)是实现控制反转的一种设计模式，依赖注入就是将实例变量传入到一个对象中去。

7.2 工厂设计模式

Spring 使用工厂模式可以通过 `BeanFactory` 或 `ApplicationContext` 创建 bean 对象。

两者对比：

- `BeanFactory`：延迟注入(使用到某个 bean 的时候才会注入)，相比于 `ApplicationContext` 来说会占用更少的内存，程序启动速度更快。
- `ApplicationContext`：容器启动的时候，不管你用没用到，一次性创建所有 bean。
`BeanFactory` 仅提供了最基本的依赖注入支持，`ApplicationContext` 扩展了 `BeanFactory`，除了有 `BeanFactory` 的功能还有额外更多功能，所以一般开发人员使用 `ApplicationContext` 会更多。

`ApplicationContext` 的三个实现类：

1. `ClassPathXmlApplication`：把上下文文件当成类路径资源。
2. `FileSystemXmlApplication`：从文件系统中的 XML 文件载入上下文定义信息。
3. `XmlWebApplicationContext`：从 Web 系统中的 XML 文件载入上下文定义信息。

Example:

```

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class App {
    public static void main(String[] args) {
        ApplicationContext context = new FileSystemXmlApplicationContext(
            "C:/work/IOC
Containers/springframework.applicationcontext/src/main/resources/bean-factory-
config.xml");
        HelloApplicationContext obj = (HelloApplicationContext)
context.getBean("helloApplicationContext");
        obj.getMsg();
    }
}

```

7.3 单例设计模式

在我们的系统中，有一些对象其实我们只需要一个，比如说：线程池、缓存、对话框、注册表、日志对象、充当打印机、显卡等设备驱动程序的对象。事实上，这一类对象只能有一个实例，如果制造出多个实例就可能会导致一些问题的产生，比如：程序的行为异常、资源使用过量、或者不一致性的结果。

使用单例模式的好处：

- 对于频繁使用的对象，可以省略创建对象所花费的时间，这对于那些重量级对象而言，是非常可观的一笔系统开销；
- 由于 new 操作的次数减少，因而对系统内存的使用频率也会降低，这将减轻 GC 压力，缩短 GC 停顿时间。

Spring 中 bean 的默认作用域就是 singleton(单例)的。除了 singleton 作用域，Spring 中 bean 还有下面几种作用域：

- prototype : 每次请求都会创建一个新的 bean 实例。
- request : 每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP request内有效。
- session : 每一次HTTP请求都会产生一个新的 bean，该bean仅在当前 HTTP session 内有效。
- global-session：全局session作用域，仅仅在基于portlet的web应用中才有意义，Spring5已经没有了。Portlet是能够生成语义代码(例如：HTML)片段的小型Java Web插件。它们基于portlet容器，可以像servlet一样处理HTTP请求。但是，与 servlet 不同，每个 portlet 都有不同的会话

Spring 实现单例的方式：

- xml: <bean id="userService" class="top.snailclimb.UserService" scope="singleton"/>
- 注解： @Scope(value = "singleton")

Spring 通过 ConcurrentHashMap 实现单例注册表的特殊方式实现单例模式。Spring 实现单例的核心代码如下

```

// 通过 ConcurrentHashMap (线程安全) 实现单例注册表
private final Map<String, Object> singletonObjects = new
ConcurrentHashMap<String, Object>(64);

public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {
    Assert.notNull(beanName, "'beanName' must not be null");
    synchronized (this.singletonObjects) {
        // 检查缓存中是否存在实例
        Object singletonObject = this.singletonObjects.get(beanName);

```

```

        if (singletonObject == null) {
            //...省略了很多代码
            try {
                singletonObject = singletonFactory.getObject();
            }
            //...省略了很多代码
            // 如果实例对象不存在，我们注册到单例注册表中。
            addSingleton(beanName, singletonObject);
        }
        return (singletonObject != NULL_OBJECT ? singletonObject : null);
    }
}

//将对象添加到单例注册表
protected void addSingleton(String beanName, Object singletonObject) {
    synchronized (this.singletonObjects) {
        this.singletonObjects.put(beanName, (singletonObject != null ?
singletonObject : NULL_OBJECT));
    }
}
}

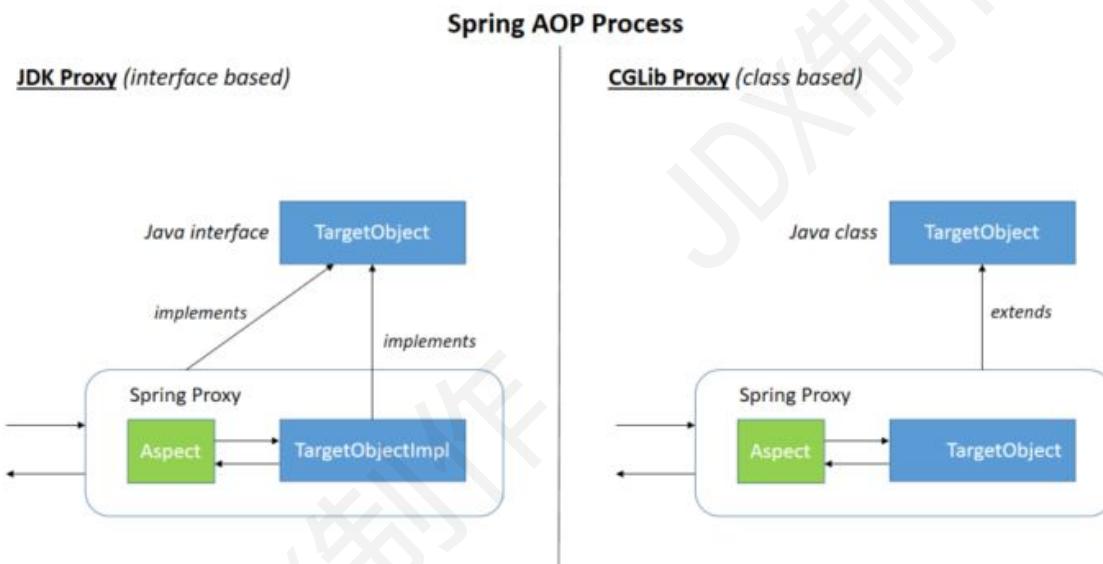
```

7.4 代理设计模式

7.4.1 代理模式在 AOP 中的应用

AOP(Aspect-Oriented Programming:面向切面编程)能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可拓展性和可维护性。

Spring AOP 就是基于动态代理的，如果要代理的对象，实现了某个接口，那么Spring AOP会使用JDK Proxy，去创建代理对象，而对于没有实现接口的对象，就无法使用 JDK Proxy 去进行代理了，这时候 Spring AOP会使用Cglib，这时候Spring AOP会使用 Cglib 生成一个被代理对象的子类来作为代理，如下图所示：



当然你也可以使用 AspectJ ,Spring AOP 已经集成了AspectJ ， AspectJ 应该算的上是 Java 生态系统中最完整的 AOP 框架了。

使用 AOP 之后我们可以把一些通用功能抽象出来，在需要用到的地方直接使用即可，这样大大简化了代码量。我们需要增加新功能时也方便，这样也提高了系统扩展性。日志功能、事务管理等等场景都用到了 AOP 。

7.4.2 Spring AOP 和 AspectJ AOP 有什么区别?

Spring AOP 属于运行时增强，而 AspectJ 是编译时增强。Spring AOP 基于代理(Proxying)，而 AspectJ 基于字节码操作(Bytecode Manipulation)。

Spring AOP 已经集成了 AspectJ，AspectJ 应该算是 Java 生态系统中最完整的 AOP 框架了。AspectJ 相比于 Spring AOP 功能更加强大，但是 Spring AOP 相对来说更简单，

如果我们的切面比较少，那么两者性能差异不大。但是，当切面太多的话，最好选择 AspectJ，它比 Spring AOP 快很多。

7.5 模板方法

模板方法模式是一种行为设计模式，它定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤的实现方式。

```
public abstract class Template {  
    //这是我们的模板方法  
    public final void TemplateMethod(){  
        PrimitiveOperation1();  
        PrimitiveOperation2();  
        PrimitiveOperation3();  
    }  
  
    protected void PrimitiveOperation1(){  
        //当前类实现  
    }  
  
    //被子类实现的方法  
    protected abstract void PrimitiveOperation2();  
    protected abstract void PrimitiveOperation3();  
}  
  
public class TemplateImpl extends Template {  
  
    @Override  
    public void PrimitiveOperation2() {  
        //当前类实现  
    }  
  
    @Override  
    public void PrimitiveOperation3() {  
        //当前类实现  
    }  
}
```

Spring 中 `jdbcTemplate`、`hibernateTemplate` 等以 `Template` 结尾的对数据库操作的类，它们就使用到了模板模式。一般情况下，我们都是使用继承的方式来实现模板模式，但是 Spring 并没有使用这种方式，而是使用 `Callback` 模式与模板方法模式配合，既达到了代码复用的效果，同时增加了灵活性。

7.6 观察者模式

观察者模式是一种对象行为型模式。它表示的是一种对象与对象之间具有依赖关系，当一个对象发生改变的时候，这个对象所依赖的对象也会做出反应。Spring 事件驱动模型就是观察者模式很经典的一个应用。Spring 事件驱动模型非常有用，在很多场景都可以解耦我们的代码。比如我们每次添加商品的时候都需要重新更新商品索引，这个时候就可以利用观察者模式来解决这个问题。

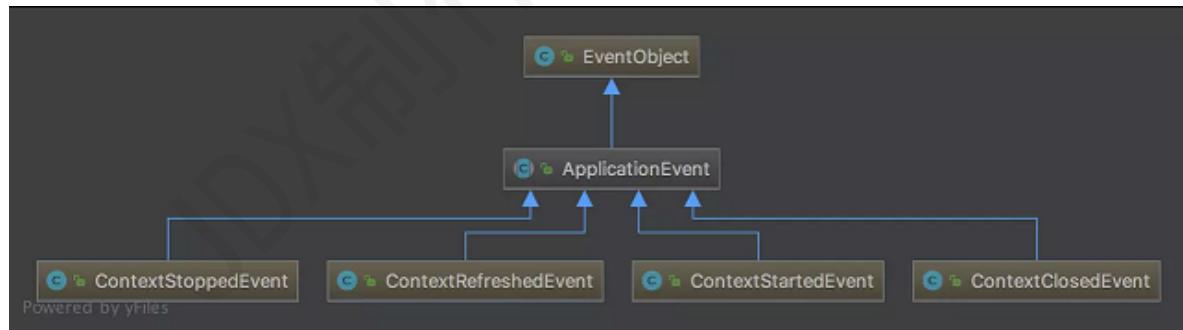
7.6.1 Spring 事件驱动模型中的三种角色

1)事件角色

`ApplicationEvent` (`org.springframework.context` 包下)充当事件的角色,这是一个抽象类，它继承了 `java.util.EventObject` 并实现了 `java.io.Serializable` 接口。

Spring 中默认存在以下事件，他们都是对 `ApplicationContextEvent` 的实现(继承自 `ApplicationEvent`)：

- `ContextStartedEvent`: `ApplicationContext` 启动后触发的事件;
- `ContextStoppedEvent`: `ApplicationContext` 停止后触发的事件;
- `ContextRefreshedEvent`: `ApplicationContext` 初始化或刷新完成后触发的事件;
- `ContextClosedEvent`: `ApplicationContext` 关闭后触发的事件。



2)事件监听者角色

`ApplicationListener` 充当了事件监听者角色，它是一个接口，里面只定义了一个 `onApplicationEvent()` 方法来处理 `ApplicationEvent`。`ApplicationListener` 接口类源码如下，可以看出接口定义看出接口中的事件只要实现了 `ApplicationEvent` 就可以了。所以，在 Spring 中我们只要实现 `ApplicationListener` 接口实现 `onApplicationEvent()` 方法即可完成监听事件

```
package org.springframework.context;
import java.util.EventListener;
@FunctionalInterface
public interface ApplicationListener<E extends ApplicationEvent> extends
EventListener {
    void onApplicationEvent(E var1);
}
```

3)事件发布者角色

`ApplicationEventPublisher` 充当了事件的发布者，它也是一个接口。

```

@FunctionalInterface
public interface ApplicationEventPublisher {
    default void publishEvent(ApplicationEvent event) {
        this.publishEvent((Object)event);
    }

    void publishEvent(Object var1);
}

```

`ApplicationEventPublisher` 接口的 `publishEvent()` 这个方法在 `AbstractApplicationContext` 类中被实现，阅读这个方法的实现，你会发现实际上事件真正是通过 `ApplicationEventMulticaster` 来广播出去的。具体内容过多，就不在这里分析了，后面可能会单独写一篇文章提到。

7.6.2 Spring 的事件流程总结

1. 定义一个事件：实现一个继承自 `ApplicationEvent`，并且写相应的构造函数；
2. 定义一个事件监听者：实现 `ApplicationListener` 接口，重写 `onApplicationEvent()` 方法；
3. 使用事件发布者发布消息：可以通过 `ApplicationEventPublisher` 的 `publishEvent()` 方法发布消息。

Example:

```

// 定义一个事件,继承自ApplicationEvent并且写相应的构造函数
public class DemoEvent extends ApplicationEvent{
    private static final long serialVersionUID = 1L;

    private String message;

    public DemoEvent(Object source, String message){
        super(source);
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

// 定义一个事件监听者,实现ApplicationListener接口, 重写 onApplicationEvent() 方法;
@Component
public class DemoListener implements ApplicationListener<DemoEvent>{

    //使用onApplicationEvent接收消息
    @Override
    public void onApplicationEvent(DemoEvent event) {
        String msg = event.getMessage();
        System.out.println("接收到的信息是: "+msg);
    }
}

// 发布事件, 可以通过ApplicationEventPublisher 的 publishEvent() 方法发布消息。
@Component
public class DemoPublisher {

```

```

    @Autowired
    ApplicationContext applicationContext;

    public void publish(String message){
        //发布事件
        applicationContext.publishEvent(new DemoEvent(this, message));
    }
}

```

当调用 `DemoPublisher` 的 `publish()` 方法的时候，比如 `demoPublisher.publish("你好")`，控制台就会打印出：接收到的信息是：你好。

7.7 适配器模式

适配器模式(Adapter Pattern) 将一个接口转换成客户希望的另一个接口，适配器模式使接口不兼容的那些类可以一起工作，其别名为包装器(Wrapper)。

7.7.1 spring AOP中的适配器模式

我们知道 Spring AOP 的实现是基于代理模式，但是 Spring AOP 的增强或通知(Advice)使用到了适配器模式，与之相关的接口是 `AdvisorAdapter`。Advice 常用的类型有：`BeforeAdvice`（目标方法调用前,前置通知）、`AfterAdvice`（目标方法调用后,后置通知）、`AfterReturningAdvice`（目标方法执行结束后, return之前）等等。每个类型Advice（通知）都有对应的拦截器：`MethodBeforeAdviceInterceptor`、`AfterReturningAdviceAdapter`、`AfterReturningAdviceInterceptor`。Spring预定义的通知要通过对称的适配器，适配成 `MethodInterceptor` 接口(方法拦截器)类型的对象（如：`MethodBeforeAdviceInterceptor` 负责适配 `MethodBeforeAdvice`）。

7.7.2 spring MVC中的适配器模式

在Spring MVC中，`DispatcherServlet` 根据请求信息调用 `HandlerMapping`，解析请求对应的 `Handler`。解析到对应的 `Handler`（也就是我们平常说的 `Controller` 控制器）后，开始由 `HandlerAdapter` 适配器处理。`HandlerAdapter` 作为期望接口，具体的适配器实现类用于对目标类进行适配，`Controller` 作为需要适配的类。

为什么要在 Spring MVC 中使用适配器模式？ Spring MVC 中的 `Controller` 种类众多，不同类型的 `Controller` 通过不同的方法来对请求进行处理。如果不利用适配器模式的话，`DispatcherServlet` 直接获取对应类型的 `Controller`，需要的自行来判断，像下面这段代码一样：

```

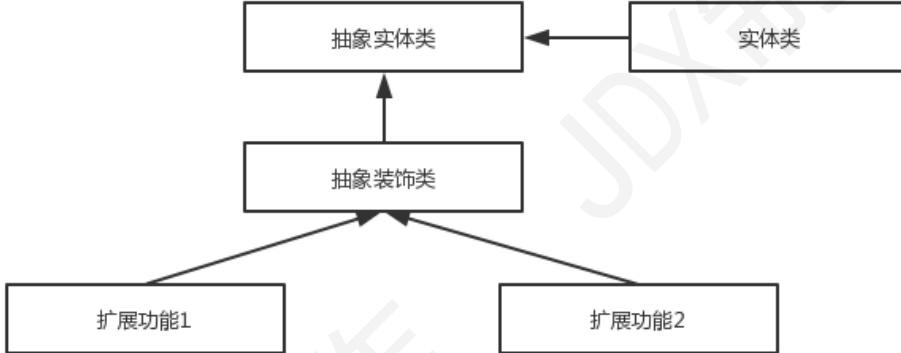
if(mappedHandler.getHandler() instanceof MultiActionController){
    ((MultiActionController)mappedHandler.getHandler()).xxx
} else if(mappedHandler.getHandler() instanceof XXX){
    ...
} else if(...){
    ...
}

```

假如我们再增加一个 `Controller` 类型就要在上面代码中再加入一行 判断语句，这种形式就使得程序难以维护，也违反了设计模式中的开闭原则 - 对扩展开放，对修改关闭。

7.8 装饰者模式

装饰者模式可以动态地给对象添加一些额外的属性或行为。相比于使用继承，装饰者模式更加灵活。简单点儿说就是当我们需要修改原有的功能，但我们又不愿直接去修改原有的代码时，设计一个 `Decorator` 套在原有代码外面。其实在 JDK 中就有很多地方用到了装饰者模式，比如 `InputStream` 家族，`InputStream` 类下有 `FileInputStream` (读取文件)、`BufferedInputStream` (增加缓存,使读取文件速度大大提升) 等子类都在不修改 `InputStream` 代码的情况下扩展了它的功能。



Spring 中配置 `DataSource` 的时候，`DataSource` 可能是不同的数据库和数据源。我们能否根据客户的需求在少修改原有类的代码下动态切换不同的数据源？这个时候就要用到装饰者模式(这一点我自己还没太理解具体原理)。Spring 中用到的包装器模式在类名上含有 `wrapper` 或者 `Decorator`。这些类基本上都是动态地给一个对象添加一些额外的职责

7.9 总结

Spring 框架中用到了哪些设计模式？

- **工厂设计模式** : Spring 使用工厂模式通过 `BeanFactory`、`ApplicationContext` 创建 bean 对象。
- **代理设计模式** : Spring AOP 功能的实现。
- **单例设计模式** : Spring 中的 Bean 默认都是单例的。
- **模板方法模式** : Spring 中 `jdbcTemplate`、`hibernateTemplate` 等以 `Template` 结尾的对数据库操作的类，它们就使用到了模板模式。
- **包装器设计模式** : 我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要会去访问不同的数据库。这种模式让我们可以根据客户的需求能够动态切换不同的数据源。
- **观察者模式**: Spring 事件驱动模型就是观察者模式很经典的一个应用。
- **适配器模式** : Spring AOP 的增强或通知(Advice)使用到了适配器模式、spring MVC 中也是用到了适配器模式适配 `Controller`。
-

(三). 认证授权(JWT、SSO)

1. JWT 身份认证优缺点分析以及常见问题解决方案

1.1 Token 认证的优势

相比于 Session 认证的方式来说，使用 token 进行身份认证主要有下面三个优势：

1.1.1 无状态

token 自身包含了身份验证所需要的所有信息，使得我们的服务器不需要存储 Session 信息，这显然增加了系统的可用性和伸缩性，大大减轻了服务端的压力。但是，也正是由于 token 的无状态，也导致了它最大的缺点：当后端在 token 有效期内废弃一个 token 或者更改它的权限的话，不会立即生效，一般需要等到有效期过后才可以。另外，当用户 Logout 的话，token 也还有效。除非，我们在后端增加额外的处理逻辑。

1.1.2 有效避免了CSRF 攻击

CSRF (Cross Site Request Forgery) 一般被翻译为 跨站请求伪造，属于网络攻击领域范围。相比于 SQL 脚本注入、XSS 等等安全攻击方式，CSRF 的知名度并没有它们高。但是，它的确是每个系统都要考虑的安全隐患，就连技术帝国 Google 的 Gmail 在早些年也被曝出过存在 CSRF 漏洞，这给 Gmail 的用户造成了很大的损失。

那么究竟什么是 跨站请求伪造 呢？说简单用你的身份去发送一些对你不友好的请求。举个简单的例子：

小壮登录了某网上银行，他来到了网上银行的帖子区，看到一个帖子下面有一个链接写着“科学理财，年盈利率过万”，小壮好奇的点开了这个链接，结果发现自己的账户少了10000元。这是怎么回事呢？原来黑客在链接中藏了一个请求，这个请求直接利用小壮的身份给银行发送了一个转账请求，也就是通过你的 Cookie 向银行发出请求。

```
<a src="http://www.mybank.com/Transfer?bankId=11&money=10000">科学理财，年盈利率过万</a>
```

导致这个问题很大的原因就是：Session 认证中 Cookie 中的 session_id 是由浏览器发送到服务端的，借助这个特性，攻击者就可以通过让用户误点攻击链接，达到攻击效果。

那为什么 token 不会存在这种问题呢？

我是这样理解的：一般情况下我们使用 JWT 的话，在我们登录成功获得 token 之后，一般会选择存放在 local storage 中。然后我们在前端通过某些方式会给每个发到后端的请求加上这个 token，这样就不会出现 CSRF 漏洞的问题。因为，即使有个你点击了非法链接发送了请求到服务端，这个非法请求是不会携带 token 的，所以这个请求将是非法的。

但是这样会存在 XSS 攻击中被盗的风险，为了避免 XSS 攻击，你可以选择将 token 存储在标记为 `httpOnly` 的 cookie 中。但是，这样又导致了你必须自己提供 CSRF 保护。

具体采用上面哪两种方式存储 token 呢，大部分情况下存放在 local storage 下都是最好的选择，某些情况下可能需要存放在标记为 `httpOnly` 的 cookie 中会更好。

1.1.3 适合移动端应用

使用 Session 进行身份认证的话，需要保存一份信息在服务器端，而且这种方式会依赖到 Cookie（需要 Cookie 保存 SessionId），所以不适合移动端。

但是，使用 token 进行身份认证就不会存在这种问题，因为只要 token 可以被客户端存储就能够使用，而且 token 还可以跨语言使用。

1.1.4 单点登录友好

使用 Session 进行身份认证的话，实现单点登录，需要我们把用户的 Session 信息保存在一台电脑上，并且还会遇到常见的 Cookie 跨域的问题。但是，使用 token 进行认证的话，token 被保存在客户端，不会存在这些问题。

1.2 Token 认证常见问题以及解决办法

1.2.1 注销登录等场景下 token 还有效

与之类似的具体相关场景有：

1. 退出登录；
2. 修改密码；
3. 服务端修改了某个用户具有的权限或者角色；
4. 用户的帐户被删除/暂停。
5. 用户由管理员注销；

这个问题不存在于 Session 认证方式中，因为在 Session 认证方式中，遇到这种情况的话服务端删除对应的 Session 记录即可。但是，使用 token 认证的方式就不好解决了。我们也说过了，token 一旦派发出去，如果后端不增加其他逻辑的话，它在失效之前都是有效的。那么，我们如何解决这个问题呢？查阅了很多资料，总结了下面几种方案：

- **将 token 存入内存数据库**：将 token 存入 DB 中，redis 内存数据库在这里是不错的选择。如果需要让某个 token 失效就直接从 redis 中删除这个 token 即可。但是，这样会导致每次使用 token 发送请求都要先从 DB 中查询 token 是否存在的步骤，而且违背了 JWT 的无状态原则。
- **黑名单机制**：和上面的方式类似，使用内存数据库比如 redis 维护一个黑名单，如果想让某个 token 失效的话就直接将这个 token 加入到 **黑名单** 即可。然后，每次使用 token 进行请求的话都会先判断这个 token 是否存在于黑名单中。
- **修改密钥 (Secret)**：我们为每个用户都创建一个专属密钥，如果我们想让某个 token 失效，我们直接修改对应用户的密钥即可。但是，这样相比于前两种引入内存数据库带来了危害更大，比如：
 - 1□如果服务是分布式的，则每次发出新的 token 时都必须在多台机器同步密钥。为此，你需要将必须将机密存储在数据库或其他外部服务中，这样和 Session 认证就没太大区别了。
 - 2□如果用户同时在两个浏览器打开系统，或者在手机端也打开了系统，如果它从一个地方将账号退出，那么其他地方都要重新进行登录，这是不可取的。
- **保持令牌的有效期限短并经常轮换**：很简单的一种方式。但是，会导致用户登录状态不会被持久记录，而且需要用户经常登录。

对于修改密码后 token 还有效问题的解决还是比较容易的，说一种我觉得比较好的方式：**使用用户的密码的哈希值对 token 进行签名。因此，如果密码更改，则任何先前的令牌将自动无法验证。**

1.2.2 token 的续签问题

token 有效期一般都建议设置的不太长，那么 token 过期后如何认证，如何实现动态刷新 token，避免用户经常需要重新登录？

我们先来看看在 Session 认证中一般的做法：**假如 session 的有效期30分钟，如果 30 分钟内用户有访问，就把 session 有效期被延长30分钟。**

1. **类似于 Session 认证中的做法**：这种方案满足于大部分场景。假设服务端给的 token 有效期设置为30分钟，服务端每次进行校验时，如果发现 token 的有效期马上快过期了，服务端就重新生成 token 给客户端。客户端每次请求都检查新旧token，如果不一致，则更新本地的token。这种做法的问题是仅仅在快过期的时候请求才会更新 token，对客户端不是很友好。
2. **每次请求都返回新 token**：这种方案的思路很简单，但是，很明显，开销会比较大。
3. **token 有效期设置到半夜**：这种方案是一种折衷的方案，保证了大部分用户白天可以正常登录，适用于对安全性要求不高的系统。
4. **用户登录返回两个 token**：第一个是 accessToken，它的过期时间 token 本身的过期时间比如半个小时，另外一个是 refreshToken 它的过期时间更长一点比如为1天。客户端登录后，将 accessToken 和 refreshToken 保存在本地，每次访问将 accessToken 传给服务端。服务端校验 accessToken 的有效性，如果过期的话，就将 refreshToken 传给服务端。如果有效，服务端就生成新的 accessToken 给客户端。否则，客户端就重新登录即可。该方案的不足是：
 - 1□需要客户端来配合；
 - 2□用户注销的时候需要同时保证两个 token 都无效；
 - 3□重新请求获取 token 的过程中会有短暂 token 不可用的情况（可以通过在客户端设置定时器，当accessToken 快过期的时候，提前去通过 refreshToken 获取新的accessToken）。

1.3 总结

JWT 最适合的场景是不需要服务端保存用户状态的场景，比如如果考虑到 token 注销和 token 续签的场景话，没有特别好的解决方案，大部分解决方案都给 token 加上了状态，这就有点类似 Session 认证了。

2. SSO 单点登录

2.1 前言

2.1.1 SSO说明

SSO英文全称Single Sign On，单点登录。SSO是在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统。<https://baike.baidu.com/item/SSO/3451380>

例如访问在网易账号中心（<http://reg.163.com/>）登录之后
访问以下站点都是登录状态

- 网易直播 <http://v.163.com>
- 网易博客 <http://blog.163.com>
- 网易花田 <http://love.163.com>
- 网易考拉 <https://www.kaola.com>
- 网易Lofter <http://www.lofter.com>

2.1.2 单点登录系统的好处

1. **用户角度** : 用户能够做到一次登录多次使用，无需记录多套用户名和密码，省心。
2. **系统管理员角度** : 管理员只需维护好一个统一的账号中心就可以了，方便。
3. **新系统开发角度**: 新系统开发时只需直接对接统一的账号中心即可，简化开发流程，省时。

2.1.3 设计目标

本篇文章也主要是为了探讨如何设计&实现一个SSO系统

以下为需要实现的核心功能：

- 单点登录
- 单点登出
- 支持跨域单点登录
- 支持跨域单点登出

2.2 SSO设计与实现

2.2.1 核心应用与依赖

小插曲：

更多阿里、腾讯、美团、京东等一线互联网大厂 Java面试真题；包含：基础、并发、锁、JVM、设计模式、数据结构、反射/IO、数据库、Redis、Spring、消息队列、分布式、Zookeeper、Dubbo、Mybatis、Maven、面经等。

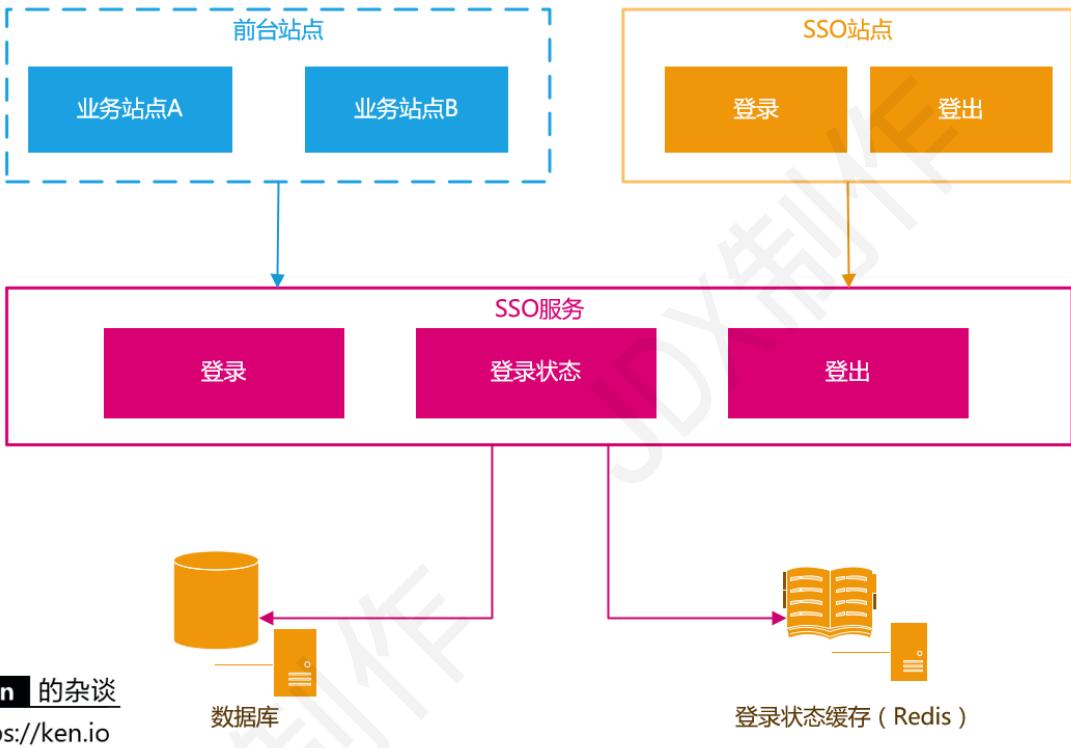
更多Java程序员技术进阶小技巧；例如高效学习（如何学习和阅读代码、面对枯燥和量大的知识）高效沟通（沟通方式及技巧、沟通技术）

更多Java大牛分享的一些职业生涯分享文档

请添加微信：jiang10086a 免费获取

比你优秀的对手在学习，你的仇人在磨刀，你的闺蜜在减肥，隔壁老王在练腰，我们必须不断学习，否则我们将被学习者超越！

趁年轻，使劲拼，给未来的自己一个交代！



应用/模块/对象	说明
前台站点	需要登录的站点
SSO站点-登录	提供登录的页面
SSO站点-登出	提供注销登录的入口
SSO服务-登录	提供登录服务
SSO服务-登录状态	提供登录状态校验/登录信息查询的服务
SSO服务-登出	提供用户注销登录的服务
数据库	存储用户账户信息
缓存	存储用户的登录信息，通常使用Redis

2.2.2 用户登录状态的存储与校验

常见的Web框架对于Session的实现都是生成一个SessionId存储在浏览器Cookie中。然后将Session内存存储在服务器端内存中，这个 ken.io 在之前Session工作原理中也提到过。整体也是借鉴这个思路。用户登录成功之后，生成AuthToken交给客户端保存。如果是浏览器，就保存在Cookie中。如果是手机App就保存在App本地缓存中。本篇主要探讨基于Web站点的SSO。

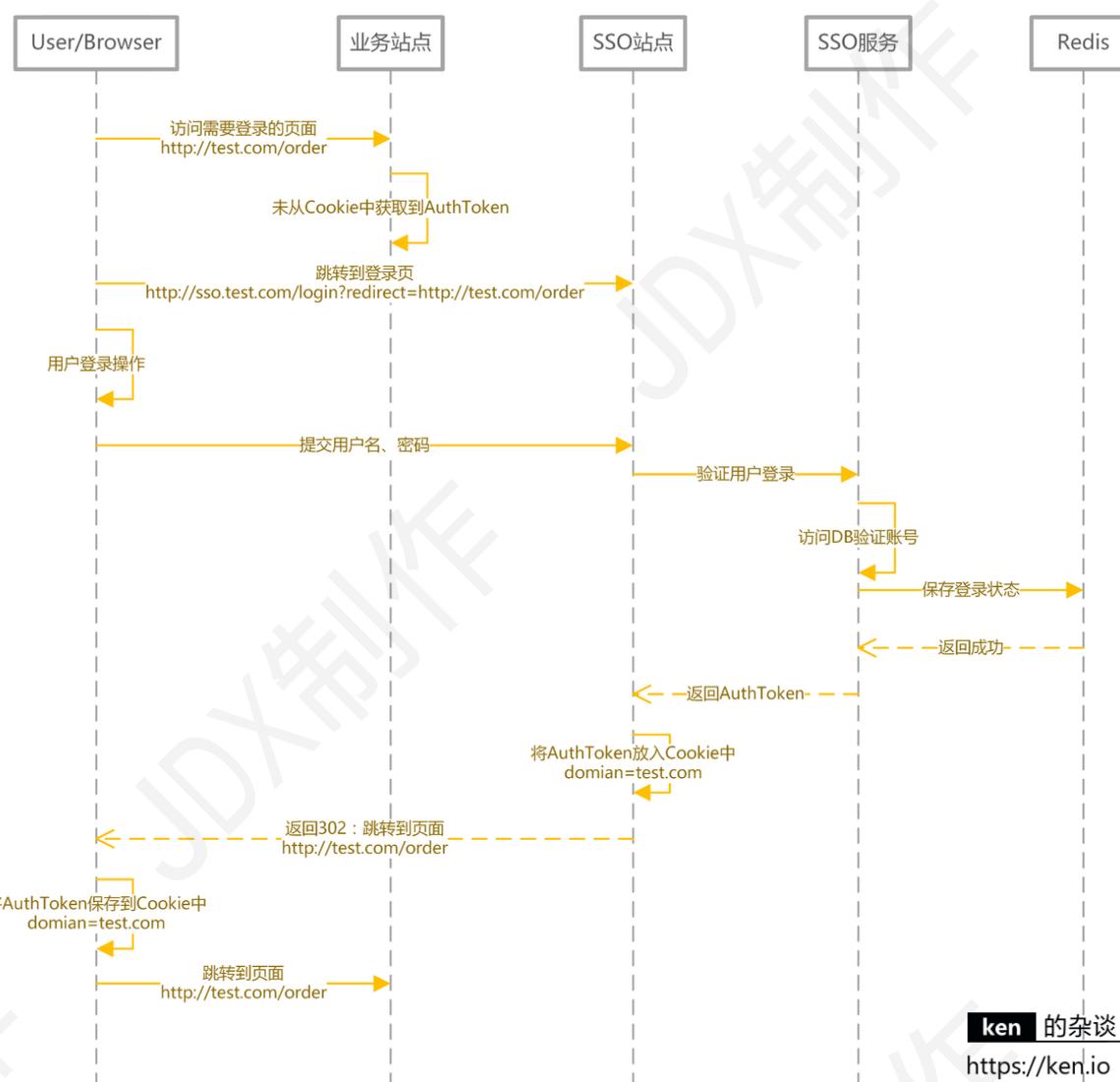
用户在浏览需要登录的页面时，客户端将AuthToken提交给SSO服务校验登录状态/获取用户登录信息

对于登录信息的存储，建议采用Redis，使用Redis集群来存储登录信息，既可以保证高可用，又可以线性扩充。同时也可以让SSO服务满足负载均衡/可伸缩的需求。

对象	说明
AuthToken	直接使用UUID/GUID即可，如果有验证AuthToken合法性需求，可以将UserName+时间戳加密生成，服务端解密之后验证合法性
登录信息	通常是将UserId, UserName缓存起来

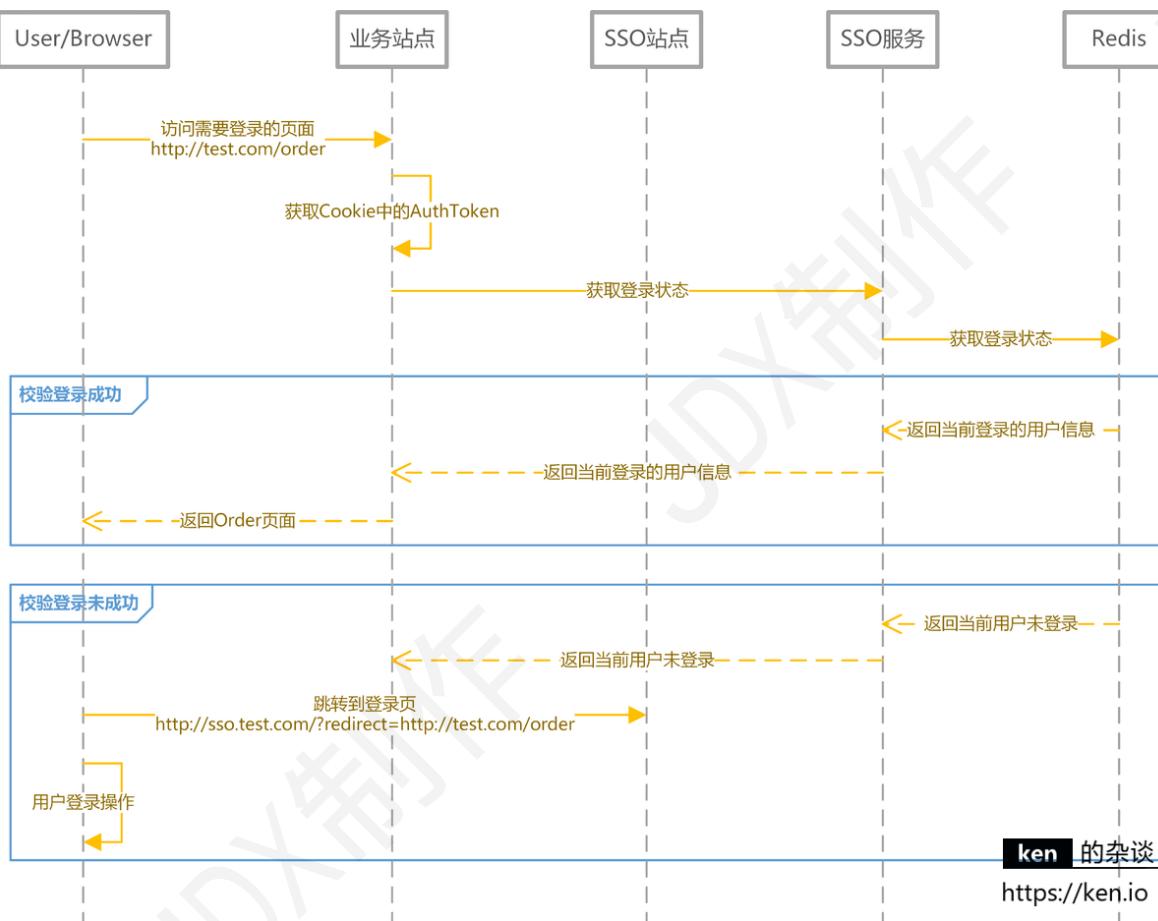
2.2.3 用户登录/登录校验

- 登录时序图



按照上图，用户登录后Auth token保存在Cookie中。 domain= test. com
浏览器会将domain设置成 .test.com，
这样访问所有*.test.com的web站点，都会将Auth token携带到服务器端。
然后通过SSO服务，完成对用户状态的校验/用户登录信息的获取

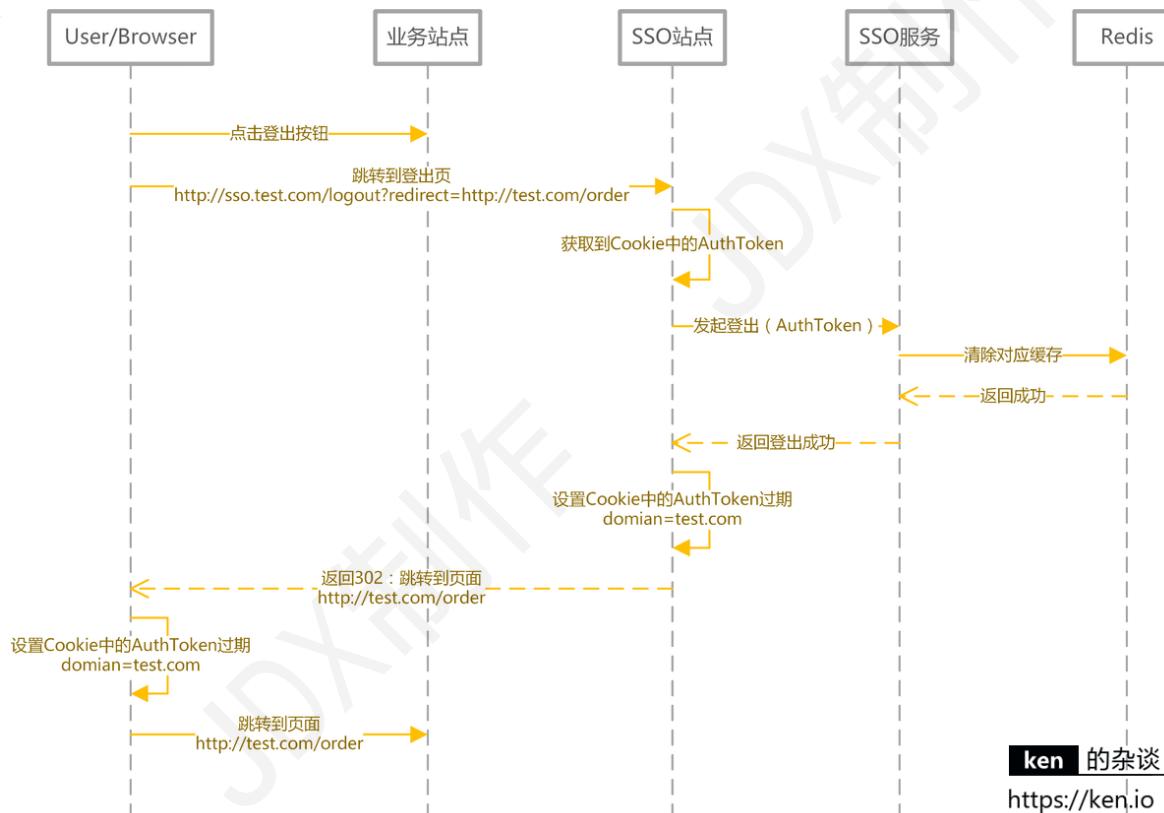
- 登录信息获取/登录状态校验



2.2.4 用户登出

用户登出时要做的事情很简单：

1. 服务端清除缓存（Redis）中的登录状态
 2. 客户端清除存储的AuthToken
- 登出时序图



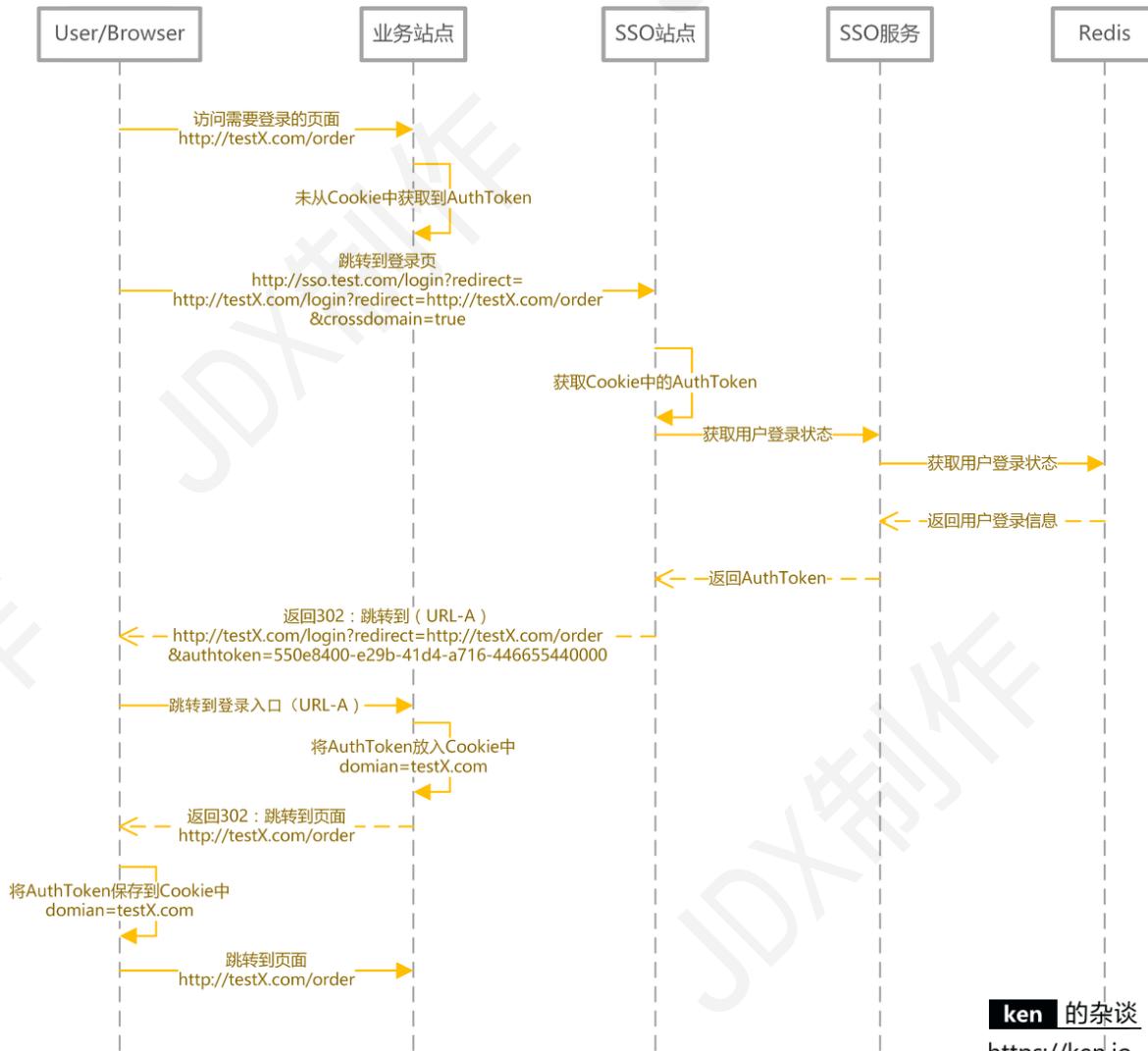
2.2.5 跨域登录、登出

前面提到过，核心思路是客户端存储AuthToken，服务器端通过Redis存储登录信息。由于客户端是将AuthToken存储在Cookie中的。所以跨域要解决的问题，就是如何解决Cookie的跨域读写问题。

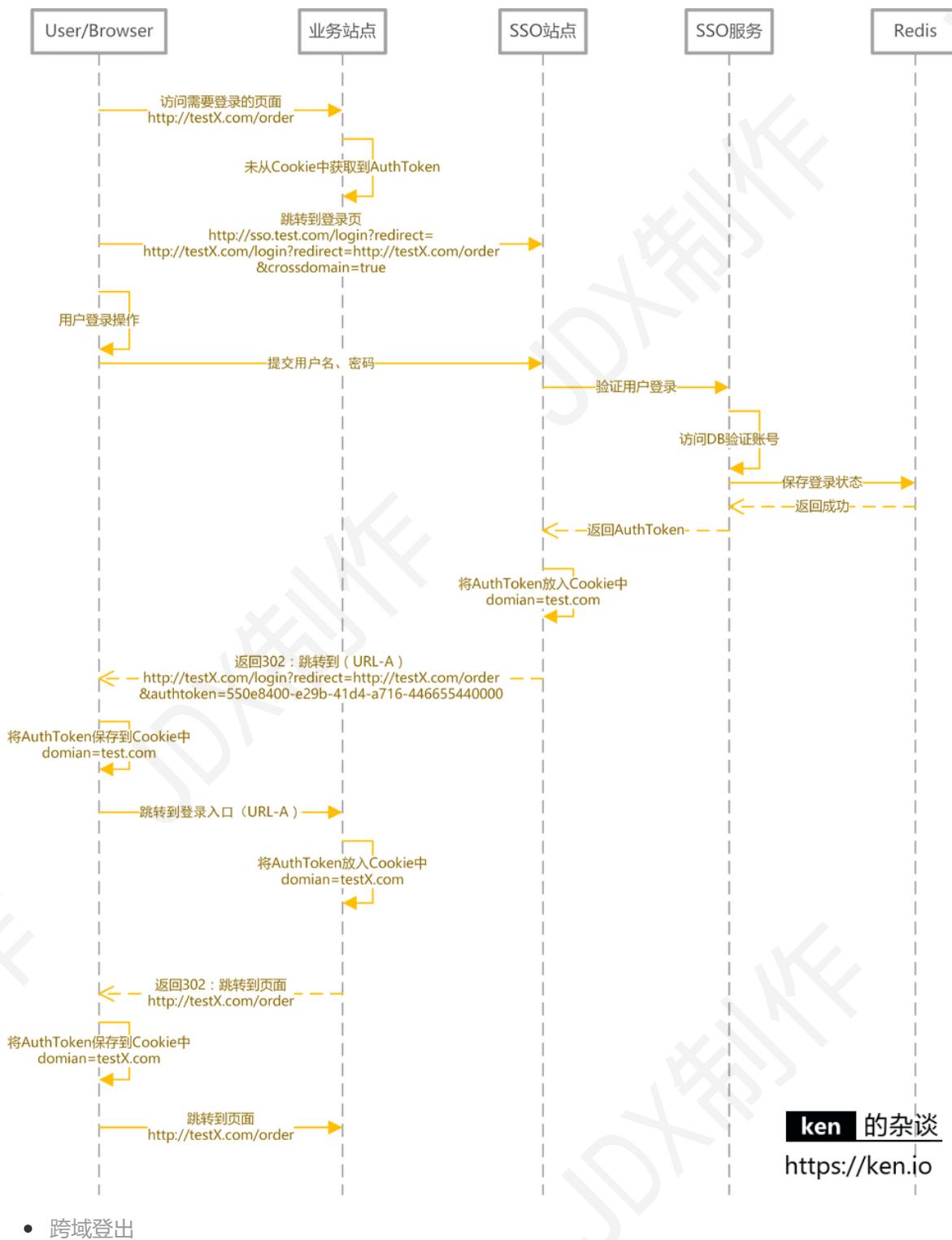
Cookie是不能跨域的，比如我一个

解决跨域的核心思路就是：

- 登录完成之后通过回调的方式，将AuthToken传递给主域名之外的站点，该站点自行将AuthToken保存在当前域下的Cookie中。
- 登出完成之后通过回调的方式，调用非主域名站点的登出页面，完成设置Cookie中的AuthToken过期的操作。
- 跨域登录（主域名已登录）

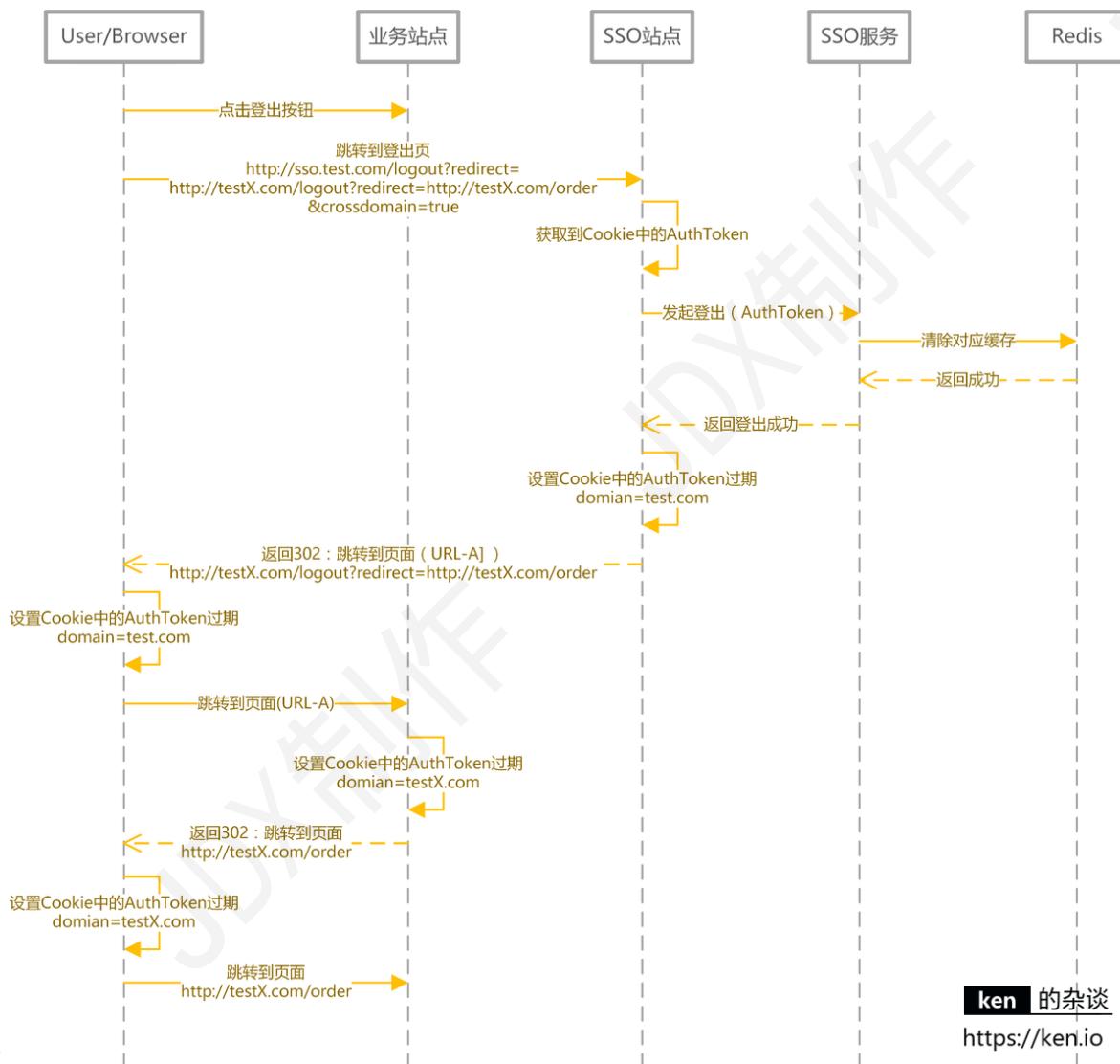


- 跨域登录（主域名未登录）



ken 的杂谈
<https://ken.io>

- 跨域登出



ken 的杂谈
<https://ken.io>

2.3 备注

- 关于方案

这次设计方案更多是提供实现思路。如果涉及到APP用户登录等情况，在访问SSO服务时，增加对APP的签名验证就好了。当然，如果有无线网关，验证签名不是问题。

- 关于时序图

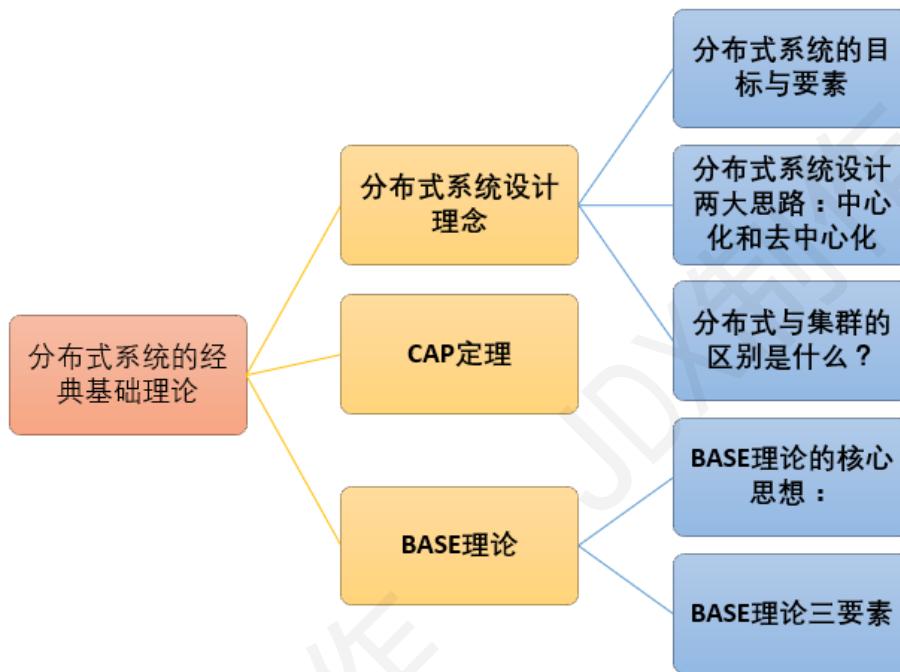
时序图中并没有包含所有场景，ken.io只列举了核心/主要场景，另外对于一些不影响理解思路的消息能省就省了。

(四). 分布式

1. 分布式相关概念入门

1.1 分布式系统的经典基础理论

本文主要是简单的介绍了三个常见的概念：分布式系统设计理念、CAP定理、BASE理论，关于分布式的还有很多很多东西。



1.2 分布式事务

分布式事务就是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。以上是百度百科的解释，简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。

1.3 一致性协议/算法

早在1900年就诞生了著名的 Paxos经典算法（Zookeeper就采用了Paxos算法的近亲兄弟Zab算法），但由于Paxos算法非常难以理解、实现、排错。所以不断有人尝试简化这一算法，直到2013年才有了重大突破：斯坦福的Diego Ongaro、John Ousterhout以易懂性为目标设计了新的一致性算法——Raft算法，并发布了对应的论文《In Search of an Understandable Consensus Algorithm》，到现在有十多种语言实现的Raft算法框架，较为出名的有以Go语言实现的Etcd，它的功能类似于Zookeeper，但采用了更为主流的Rest接口。

1.4 分布式存储

分布式存储系统将数据分散存储在多台独立的设备上。传统的网络存储系统采用集中的存储服务器存放所有数据，存储服务器成为系统性能的瓶颈，也是可靠性和安全性的焦点，不能满足大规模存储应用的需要。分布式网络存储系统采用可扩展的系统结构，利用多台存储服务器分担存储负荷，利用位置服务器定位存储信息，它不但提高了系统的可靠性、可用性和存取效率，还易于扩展。

1.5 分布式计算

所谓分布式计算是一门计算机科学，它研究如何把一个需要非常巨大的计算能力才能解决的问题分成许多小的部分，然后把这些部分分配给许多计算机进行处理，最后把这些计算结果综合起来得到最终的结果。

分布式网络存储技术是将数据分散的存储于多台独立的机器设备上。分布式网络存储系统采用可扩展的系统结构，利用多台存储服务器分担存储负荷，利用位置服务器定位存储信息，不但解决了传统集中式存储系统中单存储服务器的瓶颈问题，还提高了系统的可靠性、可用性和扩展性。

2. Dubbo

2.1 重要的概念

2.1.1 什么是 Dubbo?

Apache Dubbo (incubating) | 'dʌbəʊ | 是一款高性能、轻量级的开源Java RPC 框架，它提供了三大核心能力：面向接口的远程方法调用，智能容错和负载均衡，以及服务自动注册和发现。简单来说 Dubbo 是一个分布式服务框架，致力于提供高性能和透明化的RPC远程服务调用方案，以及SOA服务治理方案。

另外，在开源中国举行的2018年度最受欢迎中国开源软件这个活动的评选中，Dubbo 更是凭借其超高人气仅次于 vue.js 和 ECharts 获得第三名的好成绩。

Dubbo 是由阿里开源，后来加入了 Apache 。正式由于 Dubbo 的出现，才使得越来越多的公司开始使用以及接受分布式架构。

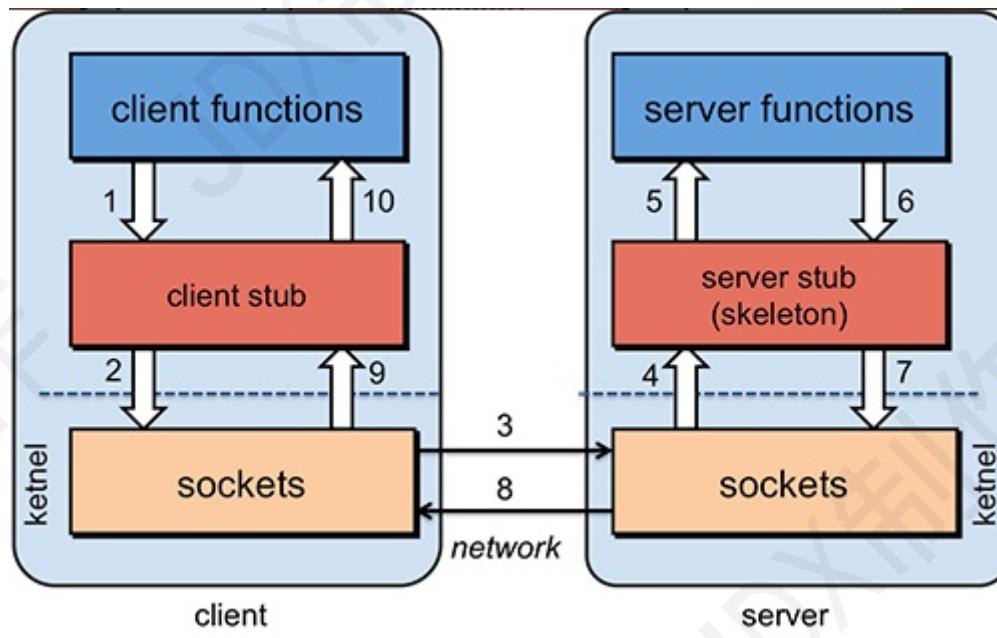
我们上面说了 Dubbo 实际上是 RPC 框架，那么什么是 RPC呢？

2.1.2 什么是 RPC?RPC原理是什么?

什么是 RPC?

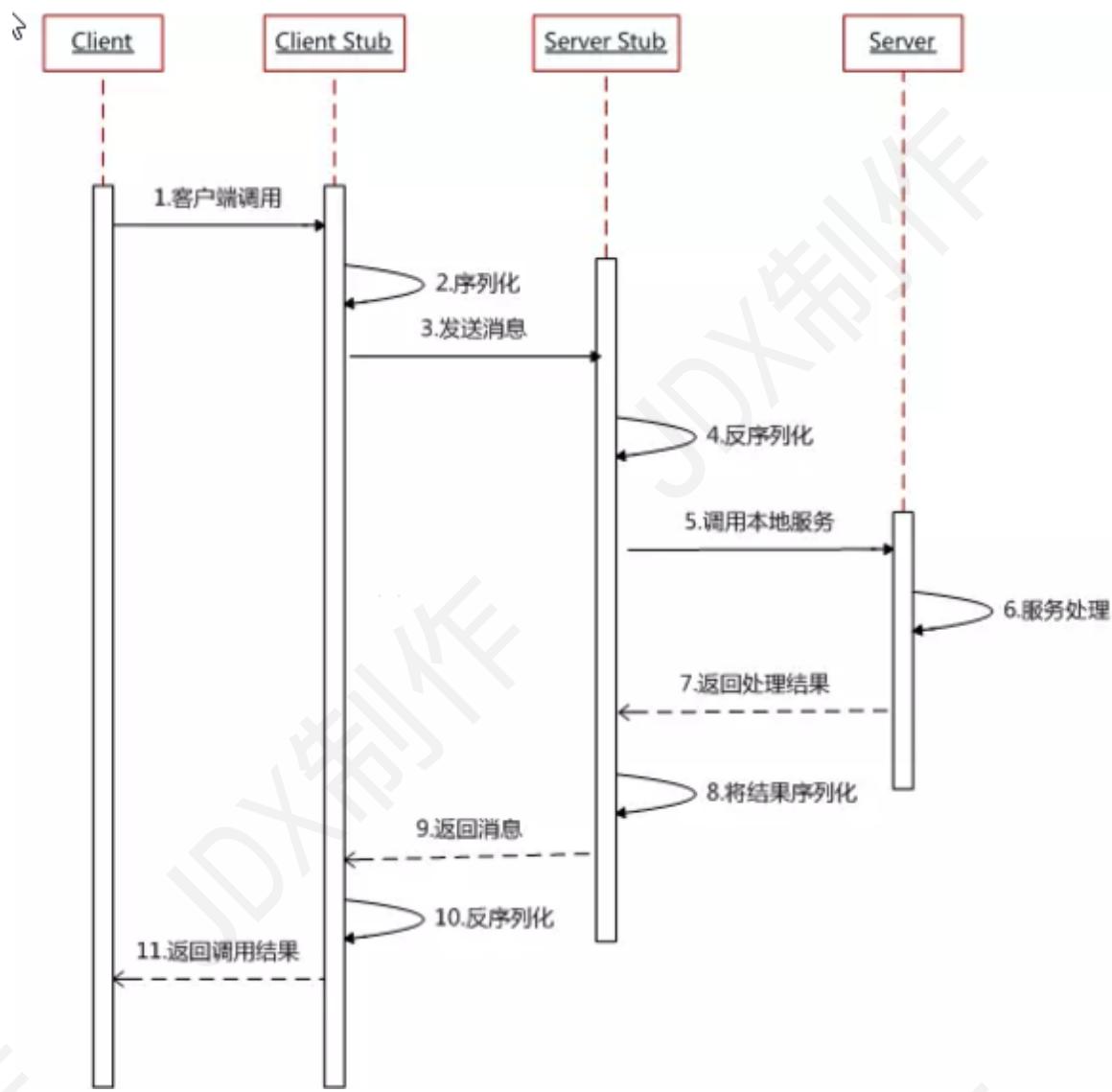
RPC (Remote Procedure Call) —远程过程调用，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。比如两个不同的服务 A、B 部署在两台不同的机器上，那么服务 A 如果想要调用服务 B 中的某个方法该怎么办呢？使用 HTTP请求当然可以，但是可能会比较麻烦。RPC 的出现就是为了让你调用远程方法像调用本地方法一样简单。

RPC原理是什么？



1. 服务消费方（client）调用以本地调用方式调用服务；
2. client stub接收到调用后负责将方法、参数等组装成能够进行网络传输的消息体；
3. client stub找到服务地址，并将消息发送到服务端；
4. server stub收到消息后进行解码；
5. server stub根据解码结果调用本地的服务；
6. 本地服务执行并将结果返回给server stub；
7. server stub将返回结果打包成消息并发送至消费方；
8. client stub接收到消息，并进行解码；
9. 服务消费方得到最终结果。

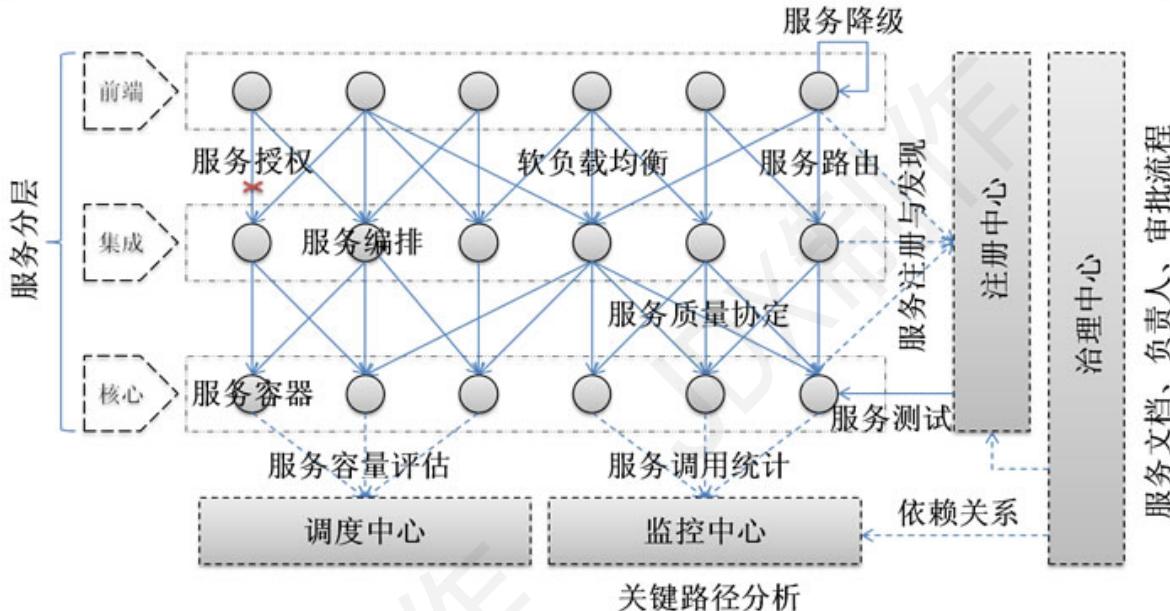
下面再贴一个网上的时序图：



说了这么多，我们为什么要用 Dubbo 呢？

2.1.3 为什么要用 Dubbo？

Dubbo 的诞生和 SOA 分布式架构的流行有着莫大的关系。SOA 面向服务的架构（Service Oriented Architecture），也就是把工程按照业务逻辑拆分成服务层、表现层两个工程。服务层中包含业务逻辑，只需要对外提供服务即可。表现层只需要处理和页面的交互，业务逻辑都是调用服务层的服务来实现。SOA 架构中有两个主要角色：服务提供者（Provider）和服务使用者（Consumer）。



如果你要开发分布式程序，你也可以直接基于 HTTP 接口进行通信，但是为什么要用 Dubbo呢？

我觉得主要可以从 Dubbo 提供的下面四点特性来说为什么要用 Dubbo：

1. **负载均衡**——同一个服务部署在不同的机器时该调用那一台机器上的服务。
2. **服务调用链路生成**——随着系统的发展，服务越来越多，服务间依赖关系变得错综复杂，甚至分不清哪个应用要在哪个应用之前启动，架构师都不能完整的描述应用的架构关系。Dubbo 可以为我们解决服务之间互相是如何调用的。
3. **服务访问压力以及时长统计、资源调度和治理**——基于访问压力实时管理集群容量，提高集群利用率。
4. **服务降级**——某个服务挂掉之后调用备用服务。

另外，Dubbo 除了能够应用在分布式系统中，也可以应用在现在比较火的微服务系统中。不过，由于 Spring Cloud 在微服务中应用更加广泛，所以，我觉得一般我们提 Dubbo 的话，大部分是分布式系统的情况。

我们刚刚提到了分布式这个概念，下面再给大家介绍一下什么是分布式？为什么要分布式？

2.1.4 什么是分布式？

分布式或者说 SOA 分布式重要的就是面向服务，说简单的分布式就是我们把整个系统拆分成不同的服务然后将这些服务放在不同的服务器上减轻单体服务的压力提高并发量和性能。比如电商系统可以简单地拆分成订单系统、商品系统、登录系统等等，拆分之后的每个服务可以部署在不同的机器上，如果某一个服务的访问量比较大的话也可以将这个服务同时部署在多台机器上。

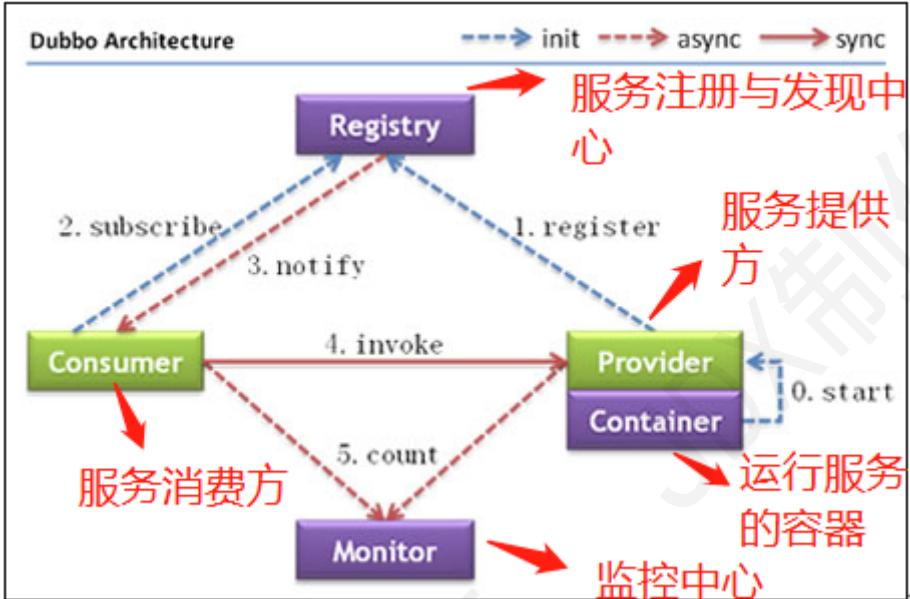
2.1.5 为什么要分布式？

从开发角度来讲单体应用的代码都集中在一起，而分布式系统的代码根据业务被拆分。所以，每个团队可以负责一个服务的开发，这样提升了开发效率。另外，代码根据业务拆分之后更加便于维护和扩展。

另外，我觉得将系统拆分分布式之后不光便于系统扩展和维护，更能提高整个系统的性能。你想一想嘛？把整个系统拆分成不同的服务/系统，然后每个服务/系统 单独部署在一台服务器上，是不是很大程度上提高了系统性能呢？

2.2 Dubbo 的架构

2.2.1 Dubbo 的架构图解



上述节点简单说明：

- **Provider**: 暴露服务的服务提供方
- **Consumer**: 调用远程服务的服务消费方
- **Registry**: 服务注册与发现的注册中心
- **Monitor**: 统计服务的调用次数和调用时间的监控中心
- **Container**: 服务运行容器

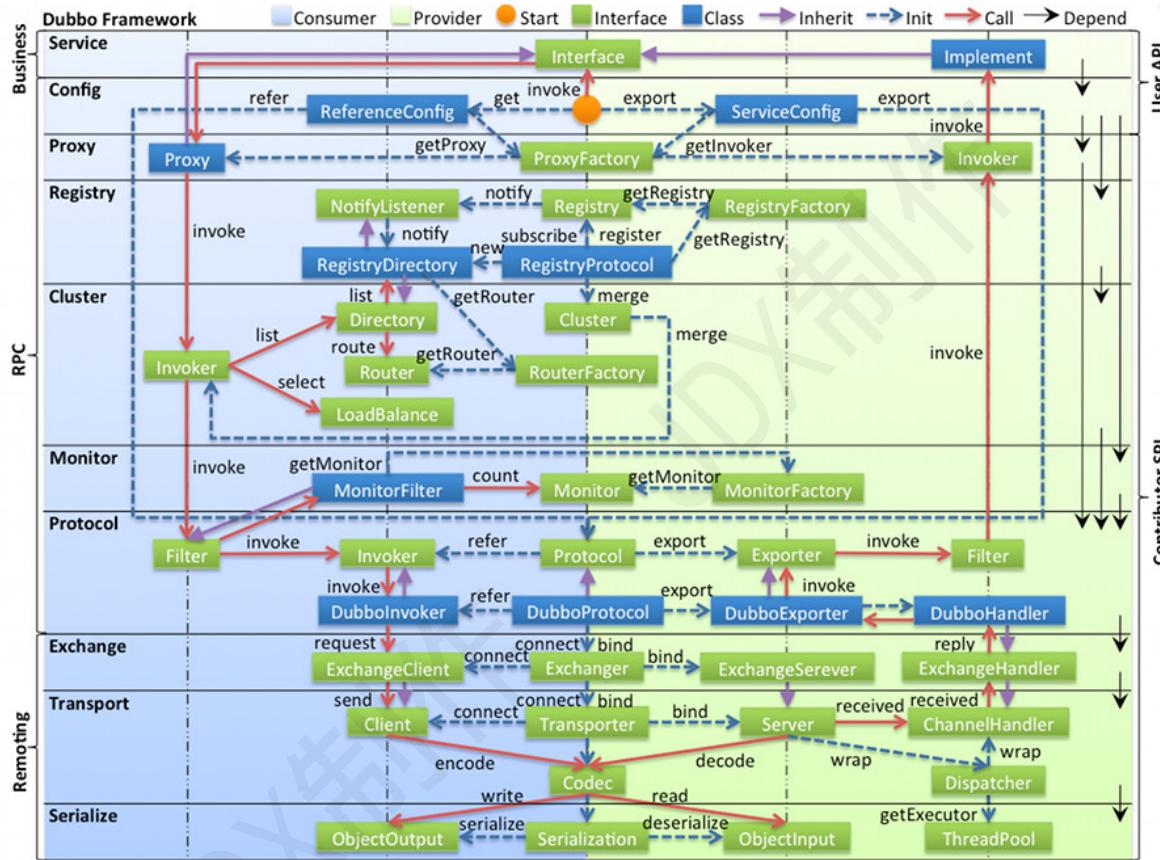
调用关系说明：

1. 服务容器负责启动，加载，运行服务提供者。
2. 服务提供者在启动时，向注册中心注册自己提供的服务。
3. 服务消费者在启动时，向注册中心订阅自己所需的服务。
4. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。
5. 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。
6. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

重要知识点总结：

- 注册中心负责服务地址的注册与查找，相当于目录服务，服务提供者和消费者只在启动时与注册中心交互，注册中心不转发请求，压力较小
- 监控中心负责统计各服务调用次数，调用时间等，统计先在内存汇总后每分钟一次发送到监控中心服务器，并以报表展示
- 注册中心，服务提供者，服务消费者三者之间均为长连接，监控中心除外
- 注册中心通过长连接感知服务提供者的存在，服务提供者宕机，注册中心将立即推送事件通知消费者
- 注册中心和监控中心全部宕机，不影响已运行的提供者和消费者，消费者在本地缓存了提供者列表
- 注册中心和监控中心都是可选的，服务消费者可以直连服务提供者
- 服务提供者无状态，任意一台宕掉后，不影响使用
- 服务提供者全部宕掉后，服务消费者应用将无法使用，并无限次重连等待服务提供者恢复

2.2.2 Dubbo 工作原理



图中从下至上分为十层，各层均为单向依赖，右边的黑色箭头代表层之间的依赖关系，每一层都可以剥离上层被复用，其中，Service 和 Config 层为 API，其它各层均为 SPI。

各层说明：

- 第一层： **service层**，接口层，给服务提供者和消费者来实现的
- 第二层： **config层**，配置层，主要是对dubbo进行各种配置的
- 第三层： **proxy层**，服务接口透明代理，生成服务的客户端 Stub 和服务器端 Skeleton
- 第四层： **registry层**，服务注册层，负责服务的注册与发现
- 第五层： **cluster层**，集群层，封装多个服务提供者的路由以及负载均衡，将多个实例组合成一个服务
- 第六层： **monitor层**，监控层，对rpc接口的调用次数和调用时间进行监控
- 第七层： **protocol层**，远程调用层，封装rpc调用
- 第八层： **exchange层**，信息交换层，封装请求响应模式，同步转异步
- 第九层： **transport层**，网络传输层，抽象mina和netty为统一接口
- 第十层： **serialize层**，数据序列化层，网络传输需要

2.3 Dubbo 的负载均衡策略

2.3.1 先来解释一下什么是负载均衡

先来个官方的解释。

维基百科对负载均衡的定义：负载均衡改善了跨多个计算资源（例如计算机，计算机集群，网络链接，中央处理单元或磁盘驱动的工作负载分布。负载平衡旨在优化资源使用，最大化吞吐量，最小化响应时间，并避免任何单个资源的过载。使用具有负载平衡而不是单个组件的多个组件可以通过冗余提高可靠性和可用性。负载平衡通常涉及专用软件或硬件。

上面讲的大家可能不太好理解，再用通俗的话给大家说一下。

比如我们的系统中的某个服务的访问量特别大，我们将这个服务部署在了多台服务器上，当客户端发起请求的时候，多台服务器都可以处理这个请求。那么，如何正确选择处理该请求的服务器就很关键。假如，你就要一台服务器来处理该服务的请求，那该服务部署在多台服务器的意义就不复存在了。负载均衡就是为了避免单个服务器响应同一请求，容易造成服务器宕机、崩溃等问题，我们从负载均衡的这四个字就能明显感受到它的意义。

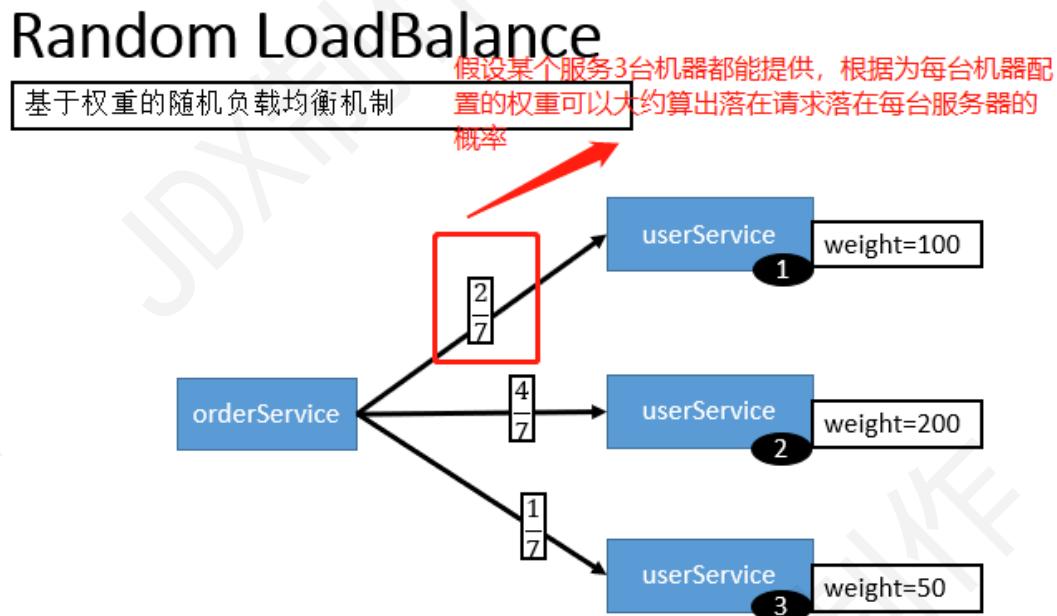
2.3.2 再来看看 Dubbo 提供的负载均衡策略

在集群负载均衡时，Dubbo 提供了多种均衡策略，默认为 `random` 随机调用。可以自行扩展负载均衡策略。

备注:下面的图片来自于：尚硅谷2018Dubbo 视频。

1) Random LoadBalance(默认，基于权重的随机负载均衡机制)

- 随机，按权重设置随机概率。
- 在一个截面上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比较均匀，有利于动态调整提供者权重。

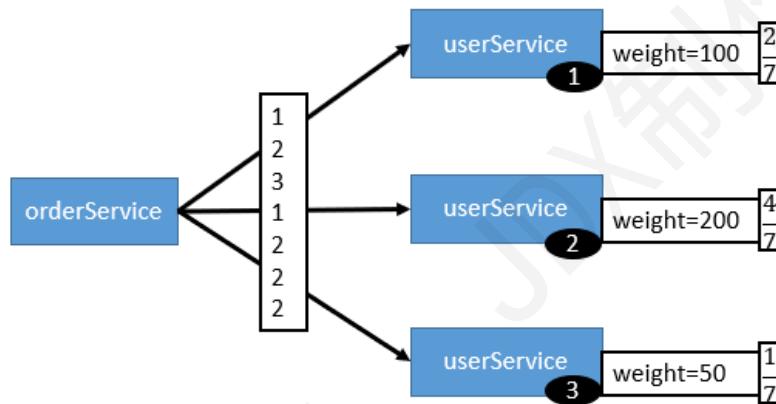


2) RoundRobin LoadBalance(不推荐，基于权重的轮询负载均衡机制)

- 轮循，按公约后的权重设置轮循比率。
- 存在慢的提供者累积请求的问题，比如：第二台机器很慢，但没挂，当请求调到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上。

RoundRobin LoadBalance

基于权重的轮询负载均衡机制



3) LeastActive LoadBalance

- 最少活跃调用数，相同活跃数的随机，活跃数指调用前后计数差。
- 使慢的提供者收到更少请求，因为越慢的提供者的调用前后计数差会越大。

4) ConsistentHash LoadBalance

- 一致性 Hash，相同参数的请求总是发到同一提供者。**(如果你需要的不是随机负载均衡，是要一类请求都到一个节点，那就走这个一致性hash策略。)**
- 当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。
- 缺省只对第一个参数 Hash，如果要修改，请配置 `<dubbo:parameter key="hash.arguments" value="0,1" />`
- 缺省用 160 份虚拟节点，如果要修改，请配置 `<dubbo:parameter key="hash.nodes" value="320" />`

2.3.3 配置方式

xml 配置方式

服务端服务级别

```
<dubbo:service interface="..." loadbalance="roundrobin" />
```

客户端服务级别

```
<dubbo:reference interface="..." loadbalance="roundrobin" />
```

服务端方法级别

```
<dubbo:service interface="...">
    <dubbo:method name="..." loadbalance="roundrobin"/>
</dubbo:service>
```

客户端方法级别

```
<dubbo:reference interface="...">
    <dubbo:method name="..." loadbalance="roundrobin"/>
</dubbo:reference>
```

注解配置方式：

消费方基于基于注解的服务级别配置方式：

```
@Reference(loadbalance = "roundrobin")
HelloService helloService;
```

2.4 zookeeper宕机与dubbo直连的情况

zookeeper宕机与dubbo直连的情况在面试中可能会被经常问到，所以要引起重视。

在实际生产中，假如zookeeper注册中心宕掉，一段时间内服务消费方还是能够调用提供方的服务的，实际上它使用的本地缓存进行通讯，这只是dubbo健壮性的一种体现。

dubbo的健壮性表现：

1. 监控中心宕掉不影响使用，只是丢失部分采样数据
2. 数据库宕掉后，注册中心仍能通过缓存提供服务列表查询，但不能注册新服务
3. 注册中心对等集群，任意一台宕掉后，将自动切换到另一台
4. 注册中心全部宕掉后，服务提供者和服务消费者仍能通过本地缓存通讯
5. 服务提供者无状态，任意一台宕掉后，不影响使用
6. 服务提供者全部宕掉后，服务消费者应用将无法使用，并无限次重连等待服务提供者恢复

我们前面提到过：注册中心负责服务地址的注册与查找，相当于目录服务，服务提供者和消费者只在启动时与注册中心交互，注册中心不转发请求，压力较小。所以，我们可以完全可以绕过注册中心——采用 **dubbo 直连**，即在服务消费方配置服务提供方的位置信息。

xml配置方式：

```
<dubbo:reference id="userService" interface="com.zang.gmall.service.UserService"
url="dubbo://localhost:20880" />
```

注解方式：

```
@Reference(url = "127.0.0.1:20880")
HelloService helloService;
```

3.消息队列其实很简单

“RabbitMQ？”“Kafka？”“RocketMQ？”...在日常学习与开发过程中，我们常常听到消息队列这个关键词。我也在我的多篇文章中提到了这个概念。可能你是熟练使用消息队列的老手，又或者你是不懂消息队列的新手，不论你了不了解消息队列，本文都将带你搞懂消息队列的一些基本理论。如果你是老手，你可能从本文学到你之前不曾注意的一些关于消息队列的重要概念，如果你是新手，相信本文将是打开消息队列大门的一块砖。

3.1 什么是消息队列

我们可以把消息队列比作是一个存放消息的容器，当我们需要使用消息的时候可以取出消息供自己使用。消息队列是分布式系统中重要的组件，使用消息队列主要是为了通过异步处理提高系统性能和削峰、降低系统耦合性。目前使用较多的消息队列有ActiveMQ，RabbitMQ，Kafka，RocketMQ，我们后面会一一对比这些消息队列。

另外，我们知道队列 Queue 是一种先进先出的数据结构，所以消费消息时也是按照顺序来消费的。比如生产者发送消息1,2,3...对于消费者就会按照1,2,3...的顺序来消费。但是偶尔也会出现消息被消费的顺序不对的情况，比如某个消息消费失败又或者一个 queue 多个consumer 也会导致消息被消费的顺序不对，我们一定要保证消息被消费的顺序正确。

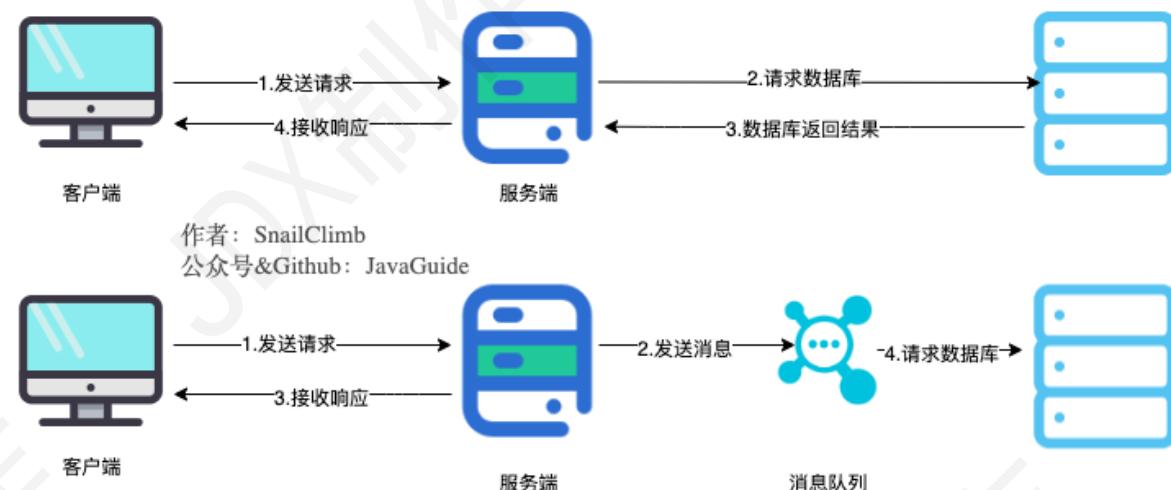
除了上面说的消息消费顺序的问题，使用消息队列，我们还要考虑如何保证消息不被重复消费？如何保证消息的可靠性传输（如何处理消息丢失的问题）？……等等问题。所以说使用消息队列也不是十全十美的，使用它也会让系统可用性降低、复杂度提高，另外需要我们保障一致性等问题。

3.2 为什么要用消息队列

我觉得使用消息队列主要有两点好处：1.通过异步处理提高系统性能（削峰、减少响应所需时间）；2.降低系统耦合性。如果在面试的时候你被面试官问到这个问题的话，一般情况是你在你的简历上涉及到消息队列这方面的内容，这个时候推荐你结合你自己的项目来回答。

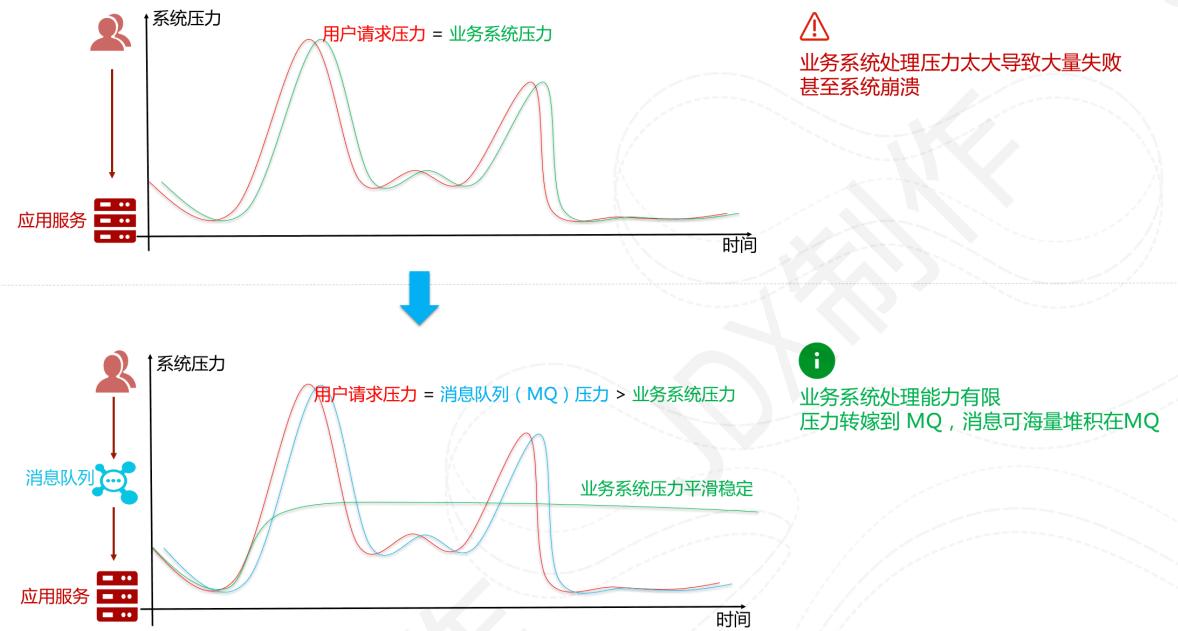
《大型网站技术架构》第四章和第七章均有提到消息队列对应用性能及扩展性的提升。

3.2.1 通过异步处理提高系统性能（削峰、减少响应所需时间）



如上图，在不使用消息队列服务器的时候，用户的请求数据直接写入数据库，在高并发的情况下数据库压力剧增，使得响应速度变慢。但是在使用消息队列之后，用户的请求数据发送给消息队列之后立即返回，再由消息队列的消费者进程从消息队列中获取数据，异步写入数据库。由于消息队列服务器处理速度快于数据库（消息队列也比数据库有更好的伸缩性），因此响应速度得到大幅改善。

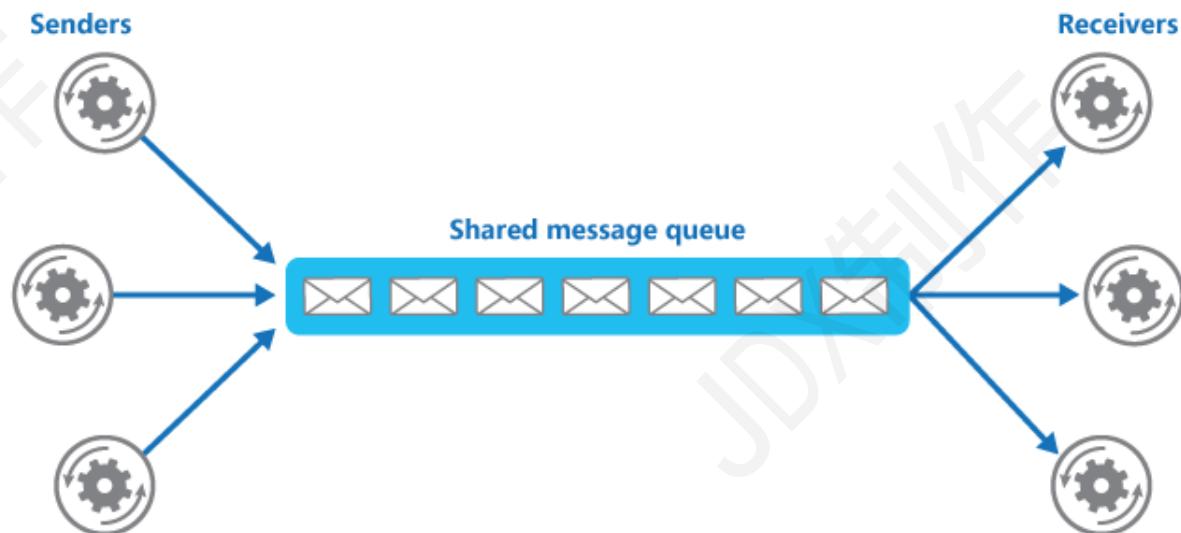
通过以上分析我们可以得出消息队列具有很好的削峰作用的功能——即通过异步处理，将短时间高并发产生的事务消息存储在消息队列中，从而削平高峰期的并发事务。举例：在电子商务一些秒杀、促销活动中，合理使用消息队列可以有效抵御促销活动刚开始大量订单涌入对系统的冲击。如下图所示：



因为用户请求数据写入消息队列之后就立即返回给用户了，但是请求数据在后续的业务校验、写数据库等操作中可能失败。因此使用消息队列进行异步处理之后，需要适当修改业务流程进行配合，比如用户在提交订单之后，订单数据写入消息队列，不能立即返回用户订单提交成功，需要在消息队列的订单消费者进程真正处理完该订单之后，甚至出库后，再通过电子邮件或短信通知用户订单成功，以免交易纠纷。这就类似我们平时手机订火车票和电影票。

3.2.2 降低系统耦合性

使用消息队列还可以降低系统耦合性。我们知道如果模块之间不存在直接调用，那么新增模块或者修改模块就对其他模块影响较小，这样系统的可扩展性无疑更好一些。还是直接上图吧：



生产者（客户端）发送消息到消息队列中去，接受者（服务端）处理消息，需要消费的系统直接去消息队列取消息进行消费即可而不需要和其他系统有耦合，这显然也提高了系统的扩展性。

消息队列使利用发布-订阅模式工作，消息发送者（生产者）发布消息，一个或多个消息接受者（消费者）订阅消息。从上图可以看到消息发送者（生产者）和消息接受者（消费者）之间没有直接耦合，消息发送者将消息发送至分布式消息队列即结束对消息的处理，消息接受者从分布式消息队列获取该消息后进行后续处理，并不需要知道该消息从何而来。对新增业务，只要对该类消息感兴趣，即可订阅该消息，对原有系统和业务没有任何影响，从而实现网站业务的可扩展性设计。

消息接受者对消息进行过滤、处理、包装后，构造成一个新的消息类型，将消息继续发送出去，等待其他消息接受者订阅该消息。因此基于事件（消息对象）驱动的业务架构可以是一系列流程。

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

备注：不要认为消息队列只能利用发布-订阅模式工作，只不过在解耦这个特定业务环境下是使用发布-订阅模式的。除了发布-订阅模式，还有点对点订阅模式（一个消息只有一个消费者），我们比较常用的是发布-订阅模式。另外，这两种消息模型是JMS提供的，AMQP协议还提供了5种消息模型。

3.3 使用消息队列带来的一些问题

- **系统可用性降低：**系统可用性在某种程度上降低，为什么这样说呢？在加入MQ之前，你不用考虑消息丢失或者说MQ挂掉等等的情况，但是，引入MQ之后你就需要去考虑了！
- **系统复杂性提高：**加入MQ之后，你需要保证消息没有被重复消费、处理消息丢失的情况、保证消息传递的顺序性等等问题！
- **一致性问题：**我上面讲了消息队列可以实现异步，消息队列带来的异步确实可以提高系统响应速度。但是，万一消息的真正消费者并没有正确消费消息怎么办？这样就会导致数据不一致的情况了！

3.4 JMS VS AMQP

3.4.1 JMS

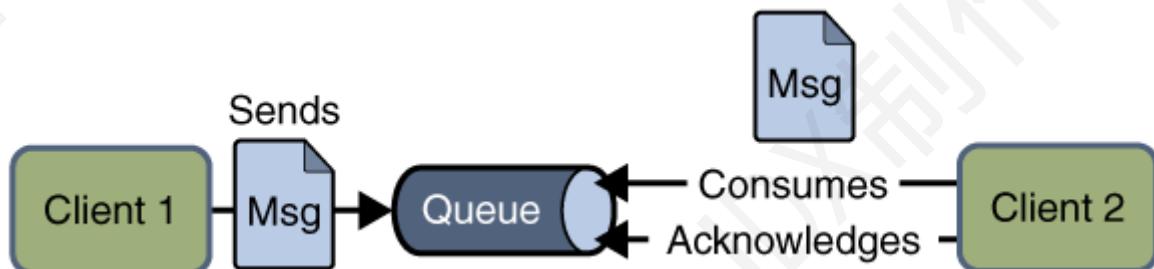
1) JMS 简介

JMS (JAVA Message Service, Java消息服务) 是Java的消息服务，JMS的客户端之间可以通过JMS服务进行异步的消息传输。**JMS (JAVA Message Service, Java消息服务) API是一个消息服务的标准或者说是规范**，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。

ActiveMQ 就是基于 JMS 规范实现的。

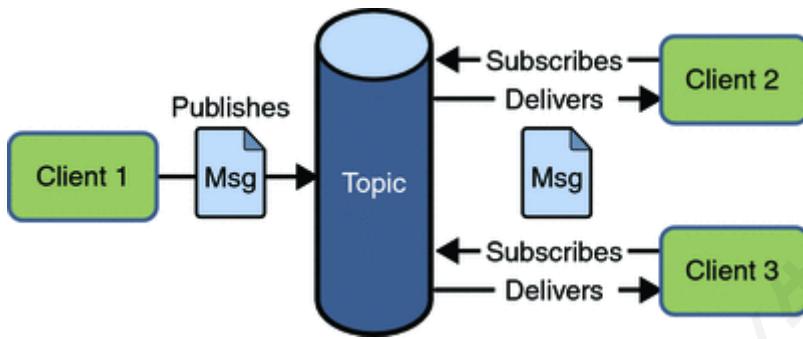
2) JMS两种消息模型

①点到点 (P2P) 模型



使用队列 (Queue) 作为消息通信载体；满足**生产者与消费者模式**，一条消息只能被一个消费者使用，未被消费的消息在队列中保留直到被消费或超时。比如：我们生产者发送100条消息的话，两个消费者来消费一般情况下两个消费者会按照消息发送的顺序各自消费一半（也就是你一个我一个的消费。）

② 发布/订阅 (Pub/Sub) 模型



发布订阅模型（Pub/Sub）使用**主题（Topic）**作为消息通信载体，类似于广播模式；发布者发布一条消息，该消息通过主题传递给所有的订阅者，在一条消息广播之后才订阅的用户则是收不到该条消息的。

3) JMS 五种不同的消息正文格式

JMS定义了五种不同的消息正文格式，以及调用的消息类型，允许你发送并接收以一些不同形式的数据，提供现有消息格式的一些级别的兼容性。

- StreamMessage -- Java原始值的数据流
- MapMessage---一套名称-值对
- TextMessage--一个字符串对象
- ObjectMessage--一个序列化的Java对象
- BytesMessage--一个字节的数据流

3.4.2 AMQP

AMQP，即Advanced Message Queuing Protocol，一个提供统一消息服务的应用层标准**高级消息队列协议**（二进制应用层协议），是应用层协议的一个开放标准，为面向消息的中间件设计，兼容JMS。基于此协议的客户端与消息中间件可传递消息，并不受客户端/中间件同产品，不同的开发语言等条件的限制。

RabbitMQ 就是基于 AMQP 协议实现的。

3.4.3 JMS vs AMQP

对比方向	JMS	AMQP
定义	Java API	协议
跨语言	否	是
跨平台	否	是
支持消息类型	提供两种消息模型：①Peer-to-Peer; ②Pub/sub	提供了五种消息模型：①direct exchange；②fanout exchange；③topic exchange；④headers exchange；⑤system exchange。本质来讲，后四种和JMS的pub/sub模型没有太大差别，仅是在路由机制上做了更详细的划分；
支持消息类型	支持多种消息类型，我们在上面提到过	byte[] (二进制)

总结：

- AMQP 为消息定义了线路层 (wire-level protocol) 的协议，而 JMS 所定义的是 API 规范。在 Java 体系中，多个 client 均可以通过 JMS 进行交互，不需要应用修改代码，但是其对跨平台的支持较差。而 AMQP 天然具有跨平台、跨语言特性。
- JMS 支持 TextMessage、MapMessage 等复杂的消息类型；而 AMQP 仅支持 byte[] 消息类型（复杂的类型可序列化后发送）。
- 由于 Exchange 提供的路由算法，AMQP 可以提供多样化的路由方式来传递消息到消息队列，而 JMS 仅支持 队列 和 主题/订阅 方式两种。

3.5 常见的消息队列对比

对比方向	概要
吞吐量	万级的 ActiveMQ 和 RabbitMQ 的吞吐量 (ActiveMQ 的性能最差) 要比十万级甚至是百万级的 RocketMQ 和 Kafka 低一个数量级。
可用性	都可以实现高可用。ActiveMQ 和 RabbitMQ 都是基于主从架构实现高可用性。RocketMQ 基于分布式架构。kafka 也是分布式的，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用
时效性	RabbitMQ 基于erlang开发，所以并发能力很强，性能极其好，延时很低，达到微秒级。其他三个都是 ms 级。
功能支持	除了 Kafka，其他三个功能都较为完备。Kafka 功能较为简单，主要支持简单的MQ功能，在大数据领域的实时计算以及日志采集被大规模使用，是事实上的标准
消息丢失	ActiveMQ 和 RabbitMQ 丢失的可能性非常低，RocketMQ 和 Kafka 理论上不会丢失。

总结：

- ActiveMQ 的社区算是比较成熟，但是较目前来说，ActiveMQ 的性能比较差，而且版本迭代很慢，不推荐使用。
- RabbitMQ 在吞吐量方面虽然稍逊于 Kafka 和 RocketMQ，但是由于它基于 erlang 开发，所以并发能力很强，性能极其好，延时很低，达到微秒级。但是也因为 RabbitMQ 基于 erlang 开发，所以国内很少有公司有实力做erlang源码级别的研究和定制。如果业务场景对并发量要求不是太高（十万级、百万级），那这四种消息队列中，RabbitMQ 一定是你的首选。如果是大数据领域的实时计算、日志采集等场景，用 Kafka 是业内标准的，绝对没问题，社区活跃度很高，绝对不会黄，何况几乎是全世界这个领域的事实性规范。
- RocketMQ 阿里出品，Java 系开源项目，源代码我们可以直接阅读，然后可以定制自己公司的 MQ，并且 RocketMQ 有阿里巴巴的实际业务场景的实战考验。RocketMQ 社区活跃度相对一般，不过也还可以，文档相对来说简单一些，然后接口这块不是按照标准 JMS 规范走的有些系统要迁移需要修改大量代码。还有就是阿里出台的技术，你得做好这个技术万一被抛弃，社区黄掉的风险，那如果你们公司有技术实力我觉得用RocketMQ 挺好的
- kafka 的特点其实很明显，就是仅仅提供较少的核心功能，但是提供超高的吞吐量，ms 级的延迟，极高的可用性以及可靠性，而且分布式可以任意扩展。同时 kafka 最好是支撑较少的 topic 数量即可，保证其超高吞吐量。kafka 唯一的一点劣势是有可能消息重复消费，那么对数据准确性会造成极其轻微的影响，在大数据领域中以及日志采集中，这点轻微影响可以忽略这个特性天然适合大数据实时计算以及日志收集。

4. RabbitMQ

4.1 RabbitMQ 介绍

4.1.1 RabbitMQ 简介

RabbitMQ 是采用 Erlang 语言实现 AMQP(Advanced Message Queuing Protocol, 高级消息队列协议) 的消息中间件，它最初起源于金融系统，用于在分布式系统中存储转发消息。

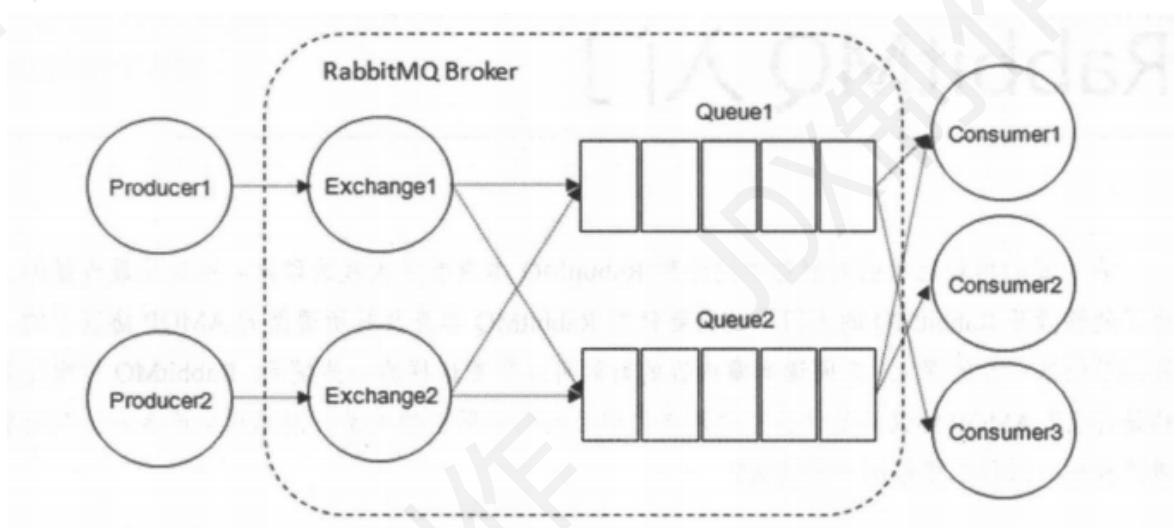
RabbitMQ 发展到今天，被越来越多的人认可，这和它在易用性、扩展性、可靠性和高可用性等方面的表现是分不开的。RabbitMQ 的具体特点可以概括为以下几点：

- **可靠性**: RabbitMQ 使用一些机制来保证消息的可靠性，如持久化、传输确认及发布确认等。
- **灵活的路由**: 在消息进入队列之前，通过交换器来路由消息。对于典型的路由功能，RabbitMQ 已经提供了一些内置的交换器来实现。针对更复杂的路由功能，可以将多个交换器绑定在一起，也可以通过插件机制来实现自己的交换器。这个后面会在我们将 RabbitMQ 核心概念的时候详细介绍到。
- **扩展性**: 多个 RabbitMQ 节点可以组成一个集群，也可以根据实际业务情况动态地扩展集群中节点。
- **高可用性**: 队列可以在集群中的机器上设置镜像，使得在部分节点出现问题的情况下队列仍然可用。
- **支持多种协议**: RabbitMQ 除了原生支持 AMQP 协议，还支持 STOMP、MQTT 等多种消息中间件协议。
- **多语言客户端**: RabbitMQ 几乎支持所有常用语言，比如 Java、Python、Ruby、PHP、C#、JavaScript 等。
- **易用的管理界面**: RabbitMQ 提供了一个易用的用户界面，使得用户可以监控和管理消息、集群中的节点等。在安装 RabbitMQ 的时候会介绍到，安装好 RabbitMQ 就自带管理界面。
- **插件机制**: RabbitMQ 提供了许多插件，以实现从多方面进行扩展，当然也可以编写自己的插件。感觉这个有点类似 Dubbo 的 SPI 机制。

4.1.2 RabbitMQ 核心概念

RabbitMQ 整体上是一个生产者与消费者模型，主要负责接收、存储和转发消息。可以把消息传递的过程想象成：当你将一个包裹送到邮局，邮局会暂存并最终将邮件通过邮递员送到收件人的手上，RabbitMQ 就好比由邮局、邮箱和邮递员组成的一个系统。从计算机术语层面来说，RabbitMQ 模型更像是一种交换机模型。

下面再来看看图1—— RabbitMQ 的整体模型架构。



下面我会一一介绍上图中的一些概念。

1) Producer(生产者) 和 Consumer(消费者)

- **Producer(生产者)**: 生产消息的一方 (邮件投递者)
- **Consumer(消费者)**: 消费消息的一方 (邮件收件人)

消息一般由 2 部分组成：**消息头**（或者说是标签 Label）和**消息体**。消息体也可以称为 payload，消息体是不透明的，而消息头则由一系列的可选属性组成，这些属性包括 routing-key（路由键）、priority（相对于其他消息的优先权）、delivery-mode（指出该消息可能需要持久性存储）等。生产者把消息交由 RabbitMQ 后，RabbitMQ 会根据消息头把消息发送给感兴趣的 Consumer（消费者）。

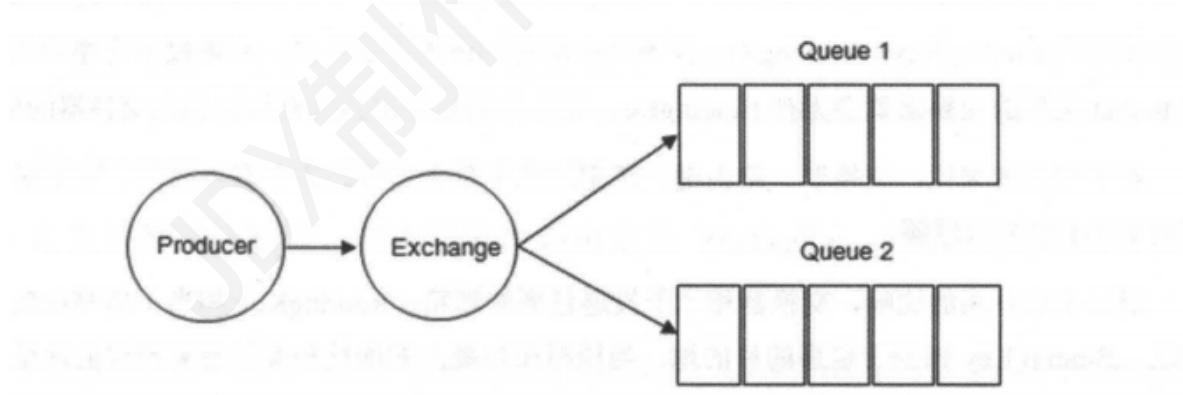
2) Exchange(交换器)

在 RabbitMQ 中，消息并不是直接被投递到 Queue（消息队列）中的，中间还必须经过 Exchange（交换器）这一层，Exchange（交换器）会把我们的消息分配到对应的 Queue（消息队列）中。

Exchange（交换器）用来接收生产者发送的消息并将这些消息路由给服务器中的队列中，如果路由不到，或许会返回给 Producer（生产者），或许会被直接丢弃掉。这里可以将 RabbitMQ 中的交换器看作一个简单的实体。

RabbitMQ 的 Exchange（交换器）有 4 种类型，不同的类型对应着不同的路由策略：direct（默认）、fanout、topic 和 headers，不同类型的 Exchange 转发消息的策略有所区别。这个会在介绍 Exchange Types（交换器类型）的时候介绍到。

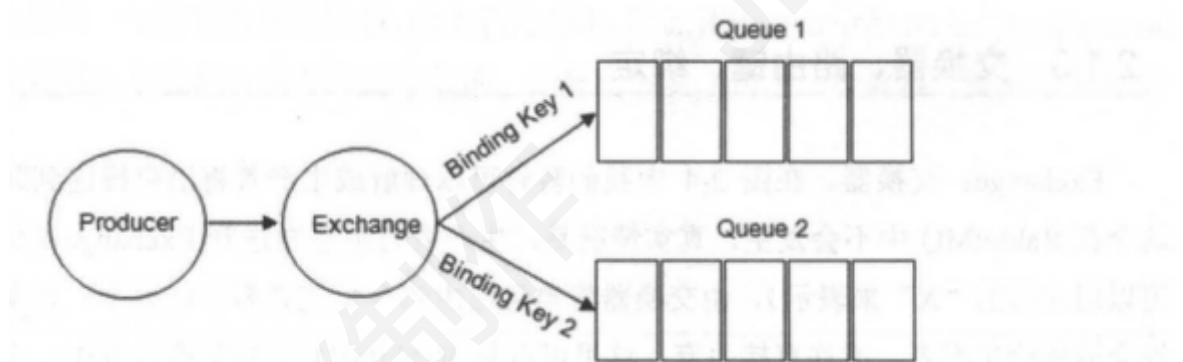
Exchange（交换器）示意图如下：



生产者将消息发给交换器的时候，一般会指定一个 RoutingKey（路由键），用来指定这个消息的路由规则，而这个 RoutingKey 需要与交换器类型和绑定键(BindingKey)联合使用才能最终生效。

RabbitMQ 中通过 Binding（绑定）将 Exchange（交换器）与 Queue（消息队列）关联起来，在绑定的时候一般会指定一个 BindingKey（绑定键），这样 RabbitMQ 就知道如何正确将消息路由到队列了，如下图所示。一个绑定就是基于路由键将交换器和消息队列连接起来的路由规则，所以可以将交换器理解成一个由绑定构成的路由表。Exchange 和 Queue 的绑定可以是多对多的关系。

Binding（绑定）示意图：



生产者将消息发送给交换器时，需要一个 RoutingKey，当 BindingKey 和 RoutingKey 相匹配时，消息会被路由到对应的队列中。在绑定多个队列到同一个交换器的时候，这些绑定允许使用相同的 BindingKey。BindingKey 并不是在所有的情况下都生效，它依赖于交换器类型，比如 fanout 类型的交换器就会无视，而是将消息路由到所有绑定到该交换器的队列中。

3) Queue(消息队列)

Queue(消息队列) 用来保存消息直到发送给消费者。它是消息的容器，也是消息的终点。一个消息可投入一个或多个队列。消息一直在队列里面，等待消费者连接到这个队列将其取走。

RabbitMQ 中消息只能存储在 **队列** 中，这一点和 **Kafka** 这种消息中间件相反。Kafka 将消息存储在 **topic (主题)** 这个逻辑层面，而相对应的队列逻辑只是topic实际存储文件中的位移标识。RabbitMQ 的生产者生产消息并最终投递到队列中，消费者可以从队列中获取消息并消费。

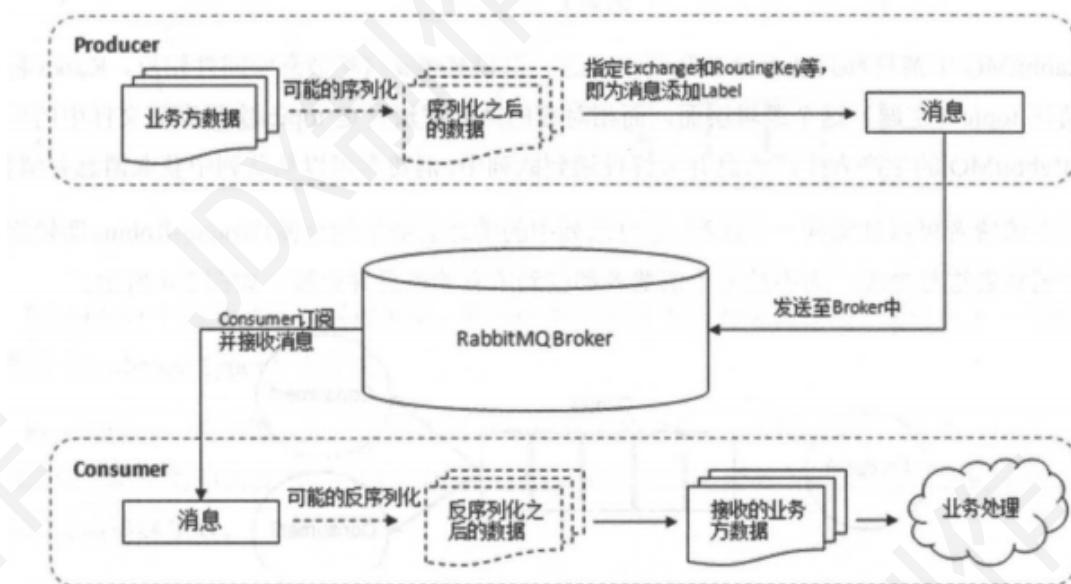
多个消费者可以订阅同一个队列，这时队列中的消息会被平均分摊 (Round-Robin, 即轮询) 给多个消费者进行处理，而不是每个消费者都收到所有的消息并处理，这样避免的消息被重复消费。

RabbitMQ 不支持队列层面的广播消费，如果有广播消费的需求，需要在其上进行二次开发，这样会很麻烦，不建议这样做。

4)Broker (消息中间件的服务节点)

对于 RabbitMQ 来说，一个 RabbitMQ Broker 可以简单地看作一个 RabbitMQ 服务节点，或者 RabbitMQ 服务实例。大多数情况下也可以将一个 RabbitMQ Broker 看作一台 RabbitMQ 服务器。

下图展示了生产者将消息存入 RabbitMQ Broker，以及消费者从 Broker 中消费数据的整个流程。



这样图1中的一些关于 RabbitMQ 的基本概念我们就介绍完毕了，下面再来介绍一下 **Exchange Types(交换器类型)**。

5)Exchange Types(交换器类型)

RabbitMQ 常用的 Exchange Type 有 **fanout**、**direct**、**topic**、**headers** 这四种 (AMQP规范里还提到两种 Exchange Type，分别为 **system** 与 **自定义**，这里不予以描述)。

① fanout

fanout 类型的 Exchange 路由规则非常简单，它会把所有发送到该 Exchange 的消息路由到所有与它绑定的 Queue 中，不需要做任何判断操作，所以 fanout 类型是所有的交换机类型里面速度最快的。fanout 类型常用来广播消息。

② direct

direct 类型的 Exchange 路由规则也很简单，它会把消息路由到那些 Bindingkey 与 RoutingKey 完全匹配的 Queue 中。

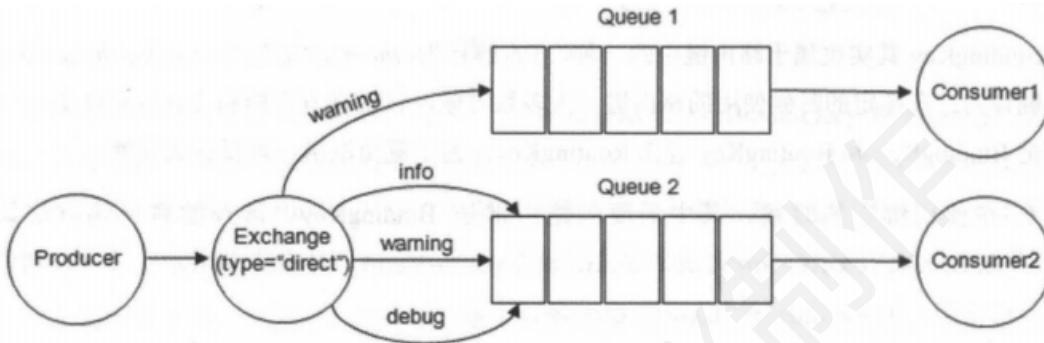


图 2-7 direct 类型的交换器

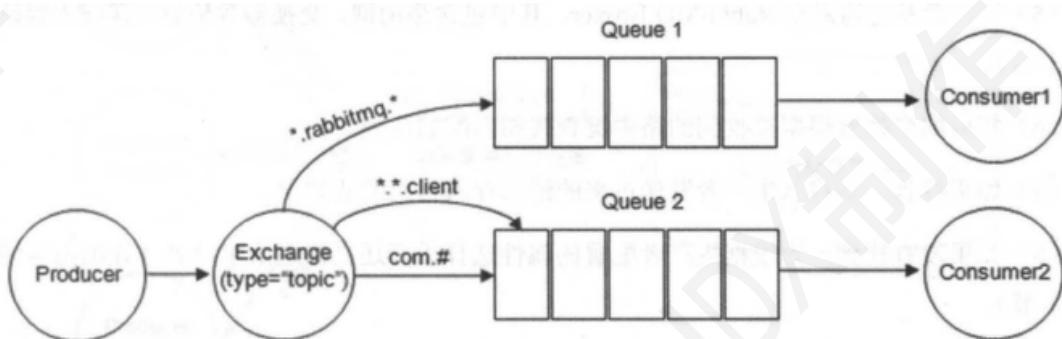
以上图为例，如果发送消息的时候设置路由键为“warning”，那么消息会路由到 Queue1 和 Queue2。如果在发送消息的时候设置路由键为“Info”或者“debug”，消息只会路由到Queue2。如果以其他的路由键发送消息，则消息不会路由到这两个队列中。

direct 类型常用在处理有优先级的任务，根据任务的优先级把消息发送到对应的队列，这样可以指派更多的资源去处理高优先级的队列。

③ topic

前面讲到direct类型的交换器路由规则是完全匹配 BindingKey 和 RoutingKey，但是这种严格的匹配方式在很多情况下不能满足实际业务的需求。topic类型的交换器在匹配规则上进行了扩展，它与 direct 类型的交换器相似，也是将消息路由到 BindingKey 和 RoutingKey 相匹配的队列中，但这里的匹配规则有些不同，它约定：

- RoutingKey 为一个点号“.”分隔的字符串（被点号“.”分隔开的每一段独立的字符串称为一个单词），如 “com.rabbitmq.client”、“java.util.concurrent”、“com.hidden.client”；
- BindingKey 和 RoutingKey 一样也是点号“.”分隔的字符串；
- BindingKey 中可以存在两种特殊字符串“*”和“#”，用于做模糊匹配，其中“*”用于匹配一个单词，“#”用于匹配多个单词(可以是零个)。



以上图为例：

- 路由键为 “com.rabbitmq.client” 的消息会同时路由到 Queue1 和 Queue2；
- 路由键为 “com.hidden.client” 的消息只会路由到 Queue2 中；
- 路由键为 “com.hidden.demo” 的消息只会路由到 Queue2 中；
- 路由键为 “java.rabbitmq.demo” 的消息只会路由到 Queue1 中；
- 路由键为 “java.util.concurrent” 的消息将会被丢弃或者返回给生产者（需要设置 mandatory 参数），因为它没有匹配任何路由键。

④ headers(不推荐)

headers 类型的交换器不依赖于路由键的匹配规则来路由消息，而是根据发送的消息内容中的 headers 属性进行匹配。在绑定队列和交换器时制定一组键值对，当发送消息到交换器时，RabbitMQ会获取到该消息的 headers (也是一个键值对的形式)对比其中的键值对是否完全匹配队列和交换器绑定时指定的键值对，如果完全匹配则消息会路由到该队列，否则不会路由到该队列。headers 类型的交换器性能会很差，而且也不实用，基本上不会看到它的存在。

4.2 安装 RabbitMq

通过 Docker 安装非常方便，只需要几条命令就好了，我这里是只说一下常规安装方法。

前面提到了 RabbitMQ 是由 Erlang 语言编写的，也正因如此，在安装 RabbitMQ 之前需要安装 Erlang。

注意：在安装 RabbitMQ 的时候需要注意 RabbitMQ 和 Erlang 的版本关系，如果不注意的话会导致出错，两者对应关系如下：

在安装rabbitmq的时候需要注意rabbitmq和erlang版本的关系，如果两者关系不一一对应的话，那么会导致出错的。rabbitmq版本和erlang的关系
[Compatibility Matrix](#)

RabbitMQ and Erlang/OTP Compatibility Matrix			
RabbitMQ version	Minimum required Erlang/OTP	Maximum supported Erlang/OTP	Notes
• 3.7.12 • 3.7.11 • 3.7.10 • 3.7.9 • 3.7.8 • 3.7.7	• 20.3.x	• 21.x	<ul style="list-style-type: none">Erlang/OTP 19.3.x support was discontinued as of Jan 1st, 2019For the best TLS support, the latest version of Erlang/OTP 21.x is recommendedOn Windows, Erlang/OTP 20.2 changed default cookie file location
• 3.7.6 • 3.7.5 • 3.7.4 • 3.7.3 • 3.7.2 • 3.7.1 • 3.7.0	• 19.3	• 20.3.x	<ul style="list-style-type: none">For the best 20.3.x is recommendedErlang vers 430_ERL-44 accepting connections and stoppingVersions prior to 19.3.6.4 are vulnerable to the ROBOT attack (CVE-2017-1000385)On Windows, Erlang/OTP 20.2 changed default cookie file location

As a rule of thumb, most recent patch versions of each supported Erlang/OTP series is recommended.

4.2.1 安装 erlang

1 下载 erlang 安装包

在官网下载然后上传到 Linux 上或者直接使用下面的命令下载对应的版本。

```
[root@Snailclimb local]#wget http://erlang.org/download/otp_src_19.3.tar.gz
```

2 解压 erlang 安装包

```
[root@Snailclimb local]#tar -xvzf otp_src_19.3.tar.gz
```

3 删除 erlang 安装包

```
[root@Snailclimb local]#rm -rf otp_src_19.3.tar.gz
```

4 安装 erlang 的依赖工具

```
[root@Snailclimb local]#yum -y install make gcc gcc-c++ kernel-devel m4 ncurses-devel openssl-devel unixODBC-devel
```

5 进入erlang 安装包解压文件对 erlang 进行安装环境的配置

新建一个文件夹

```
[root@snailclimb local]# mkdir erlang
```

对 erlang 进行安装环境的配置

```
[root@SnailClimb otp_src_19.3]#  
./configure --prefix=/usr/local/erlang --without-javac
```

6 编译安装

```
[root@SnailClimb otp_src_19.3]#  
make && make install
```

7 验证一下 erlang 是否安装成功了

```
[root@SnailClimb otp_src_19.3]# ./bin/erl
```

运行下面的语句输出“hello world”

```
io:format("hello world~n", []).
```

```
[root@SnailClimb otp_src_19.3]# ./bin/erl  
Erlang/OTP 19 [erts-8.3] [source] [64-bit] [async-threads:10] [hipe] [kernel-poll:false]  
Eshell V8.3 (abort with ^G)  
1> io:format("hello world~n", []).  
hello world  
ok  
2> █
```

大功告成，我们的 erlang 已经安装完成。

8 配置 erlang 环境变量

```
[root@SnailClimb etc]# vim profile
```

追加下列环境变量到文件末尾

```
#erlang  
ERL_HOME=/usr/local/erlang  
PATH=$ERL_HOME/bin:$PATH  
export ERL_HOME PATH
```

运行下列命令使配置文件 profile 生效

```
[root@SnailClimb etc]# source /etc/profile
```

输入 erl 查看 erlang 环境变量是否配置正确

```
[root@SnailClimb etc]# erl
```

```
[root@SnailClimb etc]# erl  
Erlang/OTP 19 [erts-8.3] [source] [64-bit] [async-threads:10] [hipe] [kernel-poll:false]  
Eshell V8.3 (abort with ^G)  
1> █
```

4.2.2 安装 RabbitMQ

1. 下载rpm

```
wget https://www.rabbitmq.com/releases/rabbitmq-server/v3.6.8/rabbitmq-server-3.6.8-1.el7.noarch.rpm
```

2. 安装rpm

```
rpm --import https://www.rabbitmq.com/rabbitmq-release-signing-key.asc
```

紧接着执行：

```
yum install rabbitmq-server-3.6.8-1.el7.noarch.rpm
```

中途需要你输入"y"才能继续安装。

3 开启 web 管理插件

```
rabbitmq-plugins enable rabbitmq_management
```

4 设置开机启动

```
chkconfig rabbitmq-server on
```

4. 启动服务

```
service rabbitmq-server start
```

5. 查看服务状态

```
service rabbitmq-server status
```

6. 访问 RabbitMQ 控制台

浏览器访问：http://你的ip地址:15672/

默认用户名和密码： guest/guest;但是需要注意的是：guest用户只是被容许从localhost访问。官网文档描述如下：

“guest” user can only connect via localhost

解决远程访问 RabbitMQ 远程访问密码错误

新建用户并授权

```
[root@SnailClimb rabbitmq]# rabbitmqctl add_user root root
Creating user "root" ...
[root@SnailClimb rabbitmq]# rabbitmqctl set_user_tags root administrator
Setting tags for user "root" to [administrator] ...
[root@SnailClimb rabbitmq]#
[root@SnailClimb rabbitmq]# rabbitmqctl set_permissions -p / root ".*" ".*" ".*"
Setting permissions for user "root" in vhost "/" ...

```

再次访问: http://你的ip地址:15672/ , 输入用户名和密码: root root

User: root Cluster: rabbit@SnailClimb (change) Log out
RabbitMQ 3.6.8, Erlang R16B03-1

Overview

Totals

- Queued messages (chart: last minute) (?)
- Currently idle
- Message rates (chart: last minute) (?)
- Currently idle
- Global counts (?)

Connections: 0 Channels: 0 Exchanges: 8 Queues: 0 Consumers: 0

Node

Node: rabbit@SnailClimb ([More about this node](#))

File descriptors (?)	Socket descriptors (?)	Erlang processes	Memory	Disk space	Rates mode	Info	Reset stats DB	+/-
52 1024 available	0 829 available	322 1048576 available	53MB 735MB high watermark	31GB 46MB low watermark	basic	Disc 1	Reset	

[Reset stats on all nodes](#)

5. RocketMQ

5.1 消息队列扫盲

消息队列顾名思义就是存放消息的队列，队列我就不解释了，别告诉我你连队列都不知道似啥吧？

所以问题并不是消息队列是什么，而是 **消息队列为什么会出现？消息队列能用来干什么？用它来干这些事会带来什么好处？消息队列会带来副作用吗？**

5.1.1 消息队列为什么会出现？

消息队列算是作为后端程序员的一个必备技能吧，因为**分布式应用必定涉及到各个系统之间的通信问题**，这个时候消息队列也应运而生了。可以说分布式的产生是消息队列的基础，而分布式怕是一个很古老的概念了吧，所以消息队列也是一个很古老的中间件了。

5.1.2 消息队列能用来干什么？

1) 异步

你可能会反驳我，应用之间的通信又不是只能由消息队列解决，好好的通信为什么中间非要插一个消息队列呢？我不能直接进行通信吗？

很好，你又提出了一个概念，**同步通信**。就比如现在业界使用比较多的 **Dubbo** 就是一个适用于各个系统之间同步通信的 **RPC** 框架。

我来举个栗子吧，比如我们有一个购票系统，需求是用户在购买完之后能接收到购买完成的短信。



我们省略中间的网络通信时间消耗，假如购票系统处理需要 150ms，短信系统处理需要 200ms，那么整个处理流程的时间消耗就是 $150\text{ms} + 200\text{ms} = 350\text{ms}$ 。

当然，乍看没什么问题。可是仔细一想你就感觉有点问题，我用户购票在购票系统的时候其实就已经完成了购买，而我现在通过同步调用非要让整个请求拉长时间，而短息系统这玩意又不是很有必要，它仅仅是一个辅助功能增强用户体验感而已。我现在整个调用流程就有点 **头重脚轻** 的感觉了，购票是一个不太耗时的流程，而我现在因为同步调用，非要等待发送短信这个比较耗时的操作才返回结果。那我如果再加一个发送邮件呢？



这样整个系统的调用链又变长了，整个时间就变成了 550ms。

当我们在学生时代需要在食堂排队的时候，我们和食堂大妈就是一个同步的模型。

我们需要告诉食堂大妈：“姐姐，给我加个鸡腿，再加个酸辣土豆丝，帮我浇点汁上去，多打点饭哦” 嘿~~~ 为了多吃点，真恶心。

然后大妈帮我们打饭配菜，我们看着大妈那颤抖的手和掉落的土豆丝不禁咽了咽口水。

最终我们从大妈手中接过饭菜然后去寻找座位了...

回想一下，我们在给大妈发送需要的信息之后我们是 **同步等待大妈给我配好饭菜** 的，上面我们只是加了鸡腿和土豆丝，万一我再加一个番茄牛腩，韭菜鸡蛋，这样是不是大妈打饭配菜的流程就会变长，我们等待的时间也会相应的变长。

那后来，我们工作赚钱了有钱去饭店吃饭了，我们告诉服务员来一碗牛肉面加个荷包蛋 (**传达一个消息**)，然后我们就可以在饭桌上安心的玩手机了 (**干自己其他事情**)，等到我们的牛肉面上了我们就可以吃了。这其中我们也就传达了一个消息，然后我们又转过头干其他事情了。这其中虽然做面的时间没有变短，但是我们只需要传达一个消息就可以看其他事情了，这是一个 **异步** 的概念。

所以，为了解决这一个问题，聪明的程序员在中间也加了个类似于服务员的中间件——消息队列。这个时候我们就可以把模型给改造了。



这样，我们在将消息存入消息队列之后我们就可以直接返回了(我们告诉服务员我们要吃什么然后玩手机)，所以整个耗时只是 $150\text{ms} + 10\text{ms} = 160\text{ms}$ 。

但是你需要注意的是，整个流程的时长是没变的，就像你仅仅告诉服务员要吃什么是不会影响到做面的速度的。

2) 解耦

回到最初同步调用的过程，我们写个伪代码简单概括一下。

```
public void purchaseTicket (Request request) {  
    // 校验  
    validate(request);  
    // 购票  
    Result result = purchase(request);  
    // 发送短信  
    sendMessage(result);  
}
```

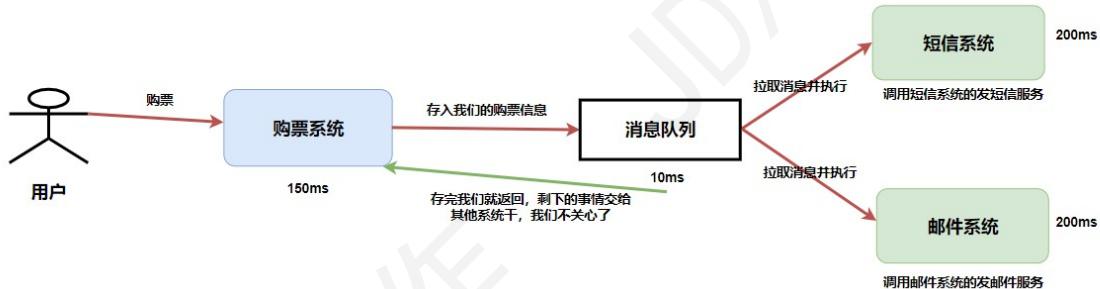
那么第二步，我们又添加了一个发送邮件，我们就得重新去修改代码，如果我们又加一个需求：用户购买完还需要给他加积分，这个时候我们是不是又得改代码？

```
public void purchaseTicket (Request request) {  
    // 校验  
    validate(request);  
    // 购票  
    Result result = purchase(request);  
    // 发送短信  
    sendMessage(result);  
    // 发送邮件  
    sendEmail(result);  
    // 添加积分  
    addPoint(result);  
    // ... 让暴风雨来的更猛烈些吧  
    // 要么一次性加完，一会加一个一会加一个  
    // 滚蛋。  
}
```

如果你觉得还行，那么我这个时候不要发邮件这个服务了呢，我是不是又得改代码，又得重启应用？

这样改来改去是不是很麻烦，那么 **此时我们就用一个消息队列在中间进行解耦**。你需要注意的是，我们后面的发送短信、发送邮件、添加积分等一些操作都依赖于上面的 `result`，这东西抽象出来就是购票的处理结果呀，比如订单号，用户账号等等，也就是说我们后面的一系列服务都是需要同样的消息来进行处理。既然这样，我们是不是可以通过“**广播消息**”来实现。

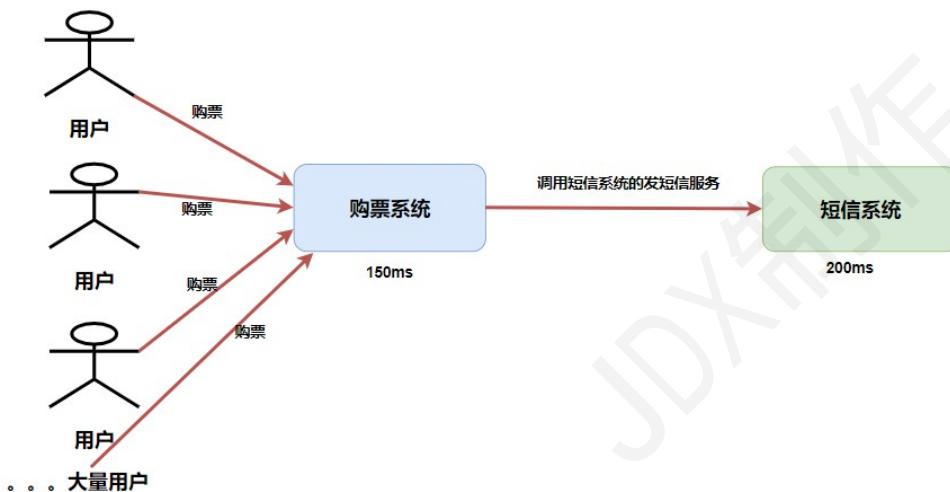
我上面所讲的“广播”并不是真正的广播，而是接下来的系统作为消费者去 **订阅** 特定的主题。比如我们这里的主题就可以叫做 `订票`，我们购买系统作为一个生产者去生产这条消息放入消息队列，然后消费者订阅了这个主题，会从消息队列中拉取消息并消费。就比如我们刚刚画的那张图，你会发现，在生产者这边我们只需要关注 **生产消息到指定主题中**，而 **消费者只需要关注从指定主题中拉取消息就行了**。



如果没有消息队列，每当一个新的业务接入，我们都要在主系统调用新接口、或者当我们取消某些业务，我们也得在主系统删除某些接口调用。有了消息队列，我们只需要关心消息是否送达了队列，至于谁希望订阅，接下来收到消息如何处理，是下游的事情，无疑极大地减少了开发和联调的工作量。

3) 削峰

我们再次回到一开始我们使用同步调用系统的情况，并且思考一下，如果此时有大量用户请求购票整个系统会变成什么样？



如果，此时有一万的请求进入购票系统，我们知道运行我们主营业务的服务器配置一般会比较好，所以这里我们假设购票系统能承受这一万的用户请求，那么也就意味着我们同时也会出现一万调用发短信服务的请求。而对于短信系统来说并不是我们的主要业务，所以我们配备的硬件资源并不会太高，那么你觉得现在这个短信系统能承受这一万的峰值么，且不说能不能承受，系统会不会 **直接崩溃** 了？

短信业务又不是我们的主营业务，我们能不能 **折中处理** 呢？如果我们把购买完成的信息发送到消息队列中，而短信系统 **尽自己所能地去消息队列中取消息和消费消息**，即使处理速度慢一点也无所谓，只要我们的系统没有崩溃就行了。

留得江山在，还怕没柴烧？你敢说每次发送验证码的时候是一发你就收到了的么？

4) 消息队列能带来什么好处？

其实上面我已经说了。**异步、解耦、削峰**。哪怕你上面的都没看懂也千万要记住这六个字，因为他不仅是消息队列的精华，更是编程和架构的精华。

5) 消息队列会带来副作用吗？

没有哪一门技术是“银弹”，消息队列也有它的副作用。

比如，本来好好的两个系统之间的调用，我中间加了个消息队列，如果消息队列挂了怎么办呢？是不是降低了系统的可用性？

那这样是不是要保证HA(高可用)？是不是要搞集群？那么我整个系统的复杂度是不是上升了？

抛开上面的问题不讲，万一我发送方发送失败了，然后执行重试，这样就可能产生重复的消息。

或者我消费端处理失败了，请求重发，这样也会产生重复的消息。

对于一些微服务来说，消费重复消息会带来更大的麻烦，比如增加积分，这个时候我加了多次是不是对其他用户不公平？

那么，又如何解决重复消费消息的问题呢？

如果我们此时的消息需要保证严格的顺序性怎么办呢？比如生产者生产了一系列的有序消息(对一个id为1的记录进行删除增加修改)，但是我们知道在发布订阅模型中，对于主题是无顺序的，那么这个时候就会导致对于消费者消费消息的时候没有按照生产者的发送顺序消费，比如这个时候我们消费的顺序为修改删除增加，如果该记录涉及到金额的话是不是会出大事情？

那么，又如何解决消息的顺序消费问题呢？

就拿我们上面所讲的分布式系统来说，用户购票完成之后是不是需要增加账户积分？在同一个系统中我们一般会使用事务来进行解决，如果用 Spring 的话我们在上面的代码中加入 `@Transactional` 注解就好了。但是在不同系统中如何保证事务呢？总不能这个系统我扣钱成功了你那积分系统积分没加吧？或者说我的扣钱明明失败了，你那积分系统给我加了积分。

那么，又如何解决分布式事务问题呢？

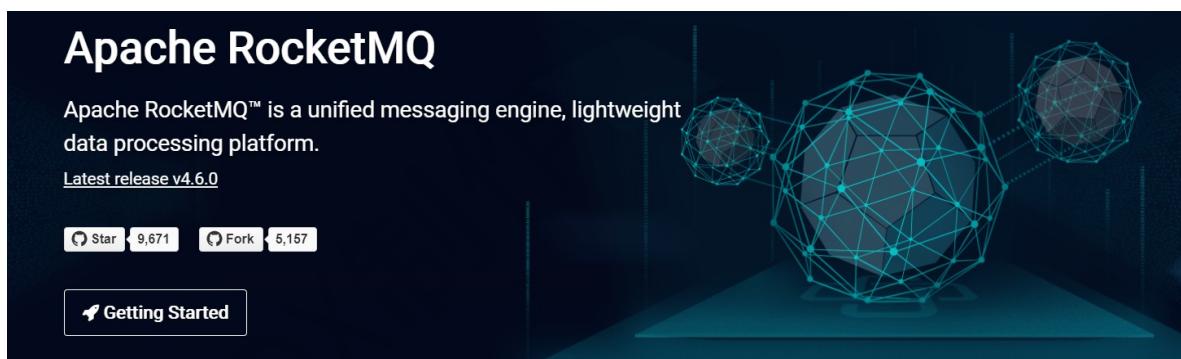
我们刚刚说了，消息队列可以进行削峰操作，那如果我的消费者如果消费很慢或者生产者生产消息很快，这样是不是会将消息堆积在消息队列中？

那么，又如何解决消息堆积的问题呢？

可用性降低，复杂度上升，又带来一系列的重复消费，顺序消费，分布式事务，消息堆积的问题，这消息队列还怎么用啊？

别急，办法总是有的。

5.2 RocketMQ是什么？



哇，你个混蛋！上面给我抛出那么多问题，你现在又讲 RocketMQ，还让不让人活了？！

别急别急，话说你现在清楚 MQ 的构造吗，我还没讲呢，我们先搞明白 MQ 的内部构造，再来看看如何解决上面的一系列问题吧，不过你最好带着问题去阅读和了解喔。

`RocketMQ` 是一个 **队列模型** 的消息中间件，具有高性能、高可靠、高实时、分布式的特点。它是一个采用 `Java` 语言开发的分布式的消息系统，由阿里巴巴团队开发，在2016年底贡献给 `Apache`，成为了 `Apache` 的一个顶级项目。在阿里内部，`RocketMQ` 很好地服务了集团大大小小上千个应用，在每年的双十一当天，更有不可思议的万亿级消息通过 `RocketMQ` 流转。

废话不多说，想要了解 `RocketMQ` 历史的同学可以自己去搜寻资料。听完上面的介绍，你只要知道 `RocketMQ` 很快、很牛、而且经历过双十一的实践就行了！

5.3 队列模型和主题模型

在谈 `RocketMQ` 的技术架构之前，我们先来了解一下两个名词概念——**队列模型** 和 **主题模型**。

首先我问一个问题，消息队列为什么要叫消息队列？

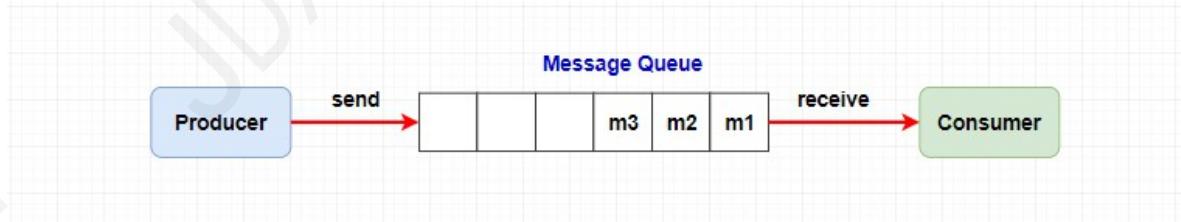
你可能觉得很弱智，这玩意不就是存放消息的队列嘛？不叫消息队列叫什么？

的确，早期的消息中间件是通过 **队列** 这一模型来实现的，可能是历史原因，我们都习惯把消息中间件成为消息队列。

但是，如今例如 `RocketMQ`、`Kafka` 这些优秀的消息中间件不仅仅是通过一个 **队列** 来实现消息存储的。

5.3.1 队列模型

就像我们理解队列一样，消息中间件的队列模型就真的只是一个队列。。。我画一张图给大家理解。



在一开始我跟你提到了一个“广播”的概念，也就是说如果我们此时我们需要将一个消息发送给多个消费者(比如此时我需要将信息发送给短信系统和邮件系统)，这个时候单个队列即不能满足需求了。

当然你可以让 `Producer` 生产消息放入多个队列中，然后每个队列去对应每一个消费者。问题是不可以解决，创建多个队列并且复制多份消息是会很影响资源和性能的。而且，这样子就会导致生产者需要知道具体消费者个数然后去复制对应数量的消息队列，这就违背我们消息中间件的 **解耦** 这一原则。

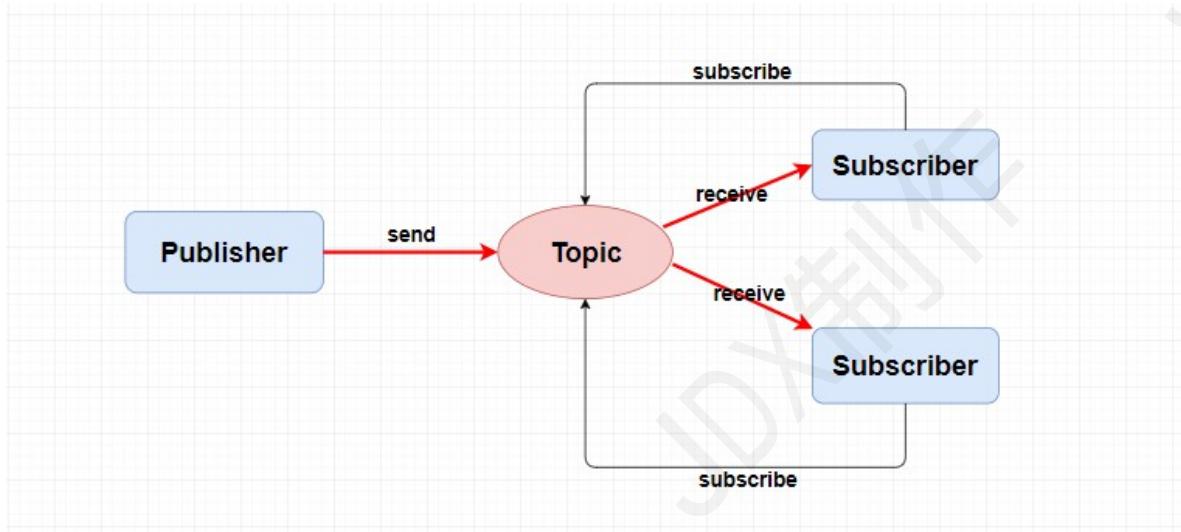
5.3.2 主题模型

那么有没有好的方法去解决这一个问题呢？有，那就是 **主题模型** 或者可以称为 **发布订阅模型**。

感兴趣的同學可以去了解一下设计模式里面的观察者模式并且手动实现一下，我相信你会有所收获的。

在主题模型中，消息的生产者称为 **发布者(Publisher)**，消息的消费者称为 **订阅者(Subscriber)**，存放消息的容器称为 **主题(Topic)**。

其中，发布者将消息发送到指定主题中，订阅者需要 **提前订阅主题** 才能接受特定主题的消息。

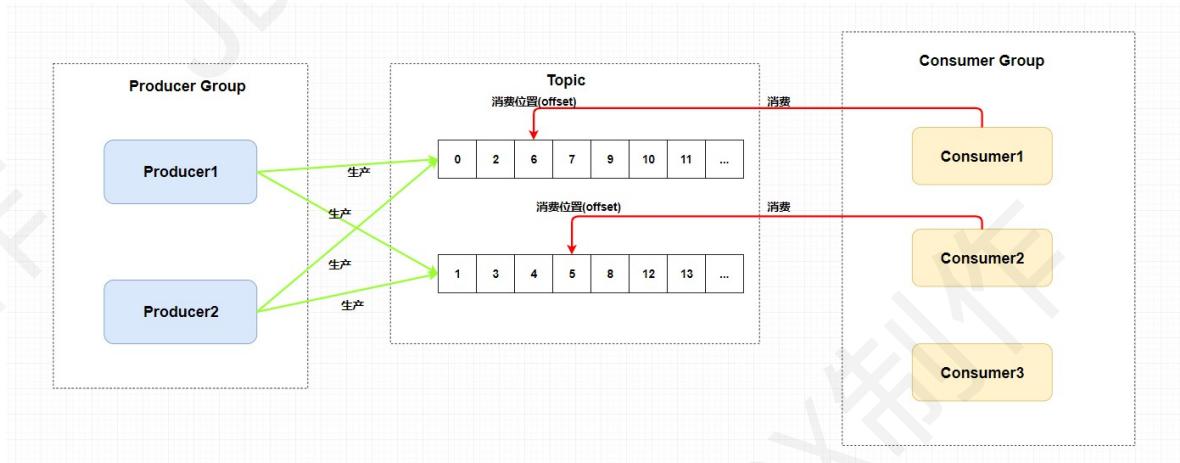


5.3.3 RocketMQ中的消息模型

`RocketMQ` 中的消息模型就是按照 **主题模型** 所实现的。你可能会好奇这个 **主题** 到底是怎么实现的呢？你上面也没有讲到呀！

其实对于主题模型的实现来说每个消息中间件的底层设计都是不一样的，就比如 `Kafka` 中的 **分区**，`RocketMQ` 中的 **队列**，`RabbitMQ` 中的 `Exchange`。我们可以理解为 **主题模型/发布订阅模型** 就是一个标准，那些中间件只不过照着这个标准去实现而已。

所以，`RocketMQ` 中的 **主题模型** 到底是如何实现的呢？首先我画一张图，大家尝试着去理解一下。



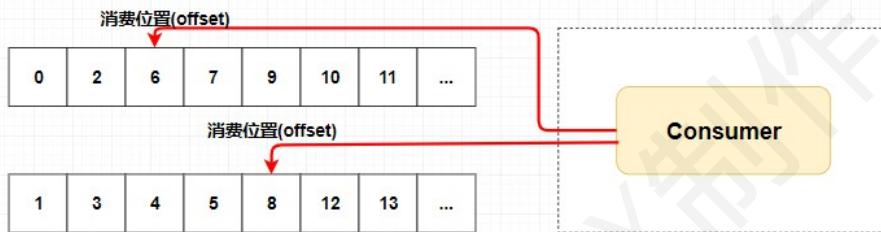
我们可以看到在整个图中有 `Producer Group`、`Topic`、`Consumer Group` 三个角色，我来分别介绍一下他们。

- `Producer Group` 生产者组：代表某一类的生产者，比如我们有多个秒杀系统作为生产者，这多个合在一起就是一个 `Producer Group` 生产者组，它们一般生产相同的消息。
- `Consumer Group` 消费者组：代表某一类的消费者，比如我们有多个短信系统作为消费者，这多个合在一起就是一个 `Consumer Group` 消费者组，它们一般消费相同的消息。
- `Topic` 主题：代表一类消息，比如订单消息，物流消息等等。

你可以看到图中生产者组中的生产者会向主题发送消息，而 **主题中存在多个队列**，生产者每次生产消息之后是指定主题中的某个队列发送消息的。

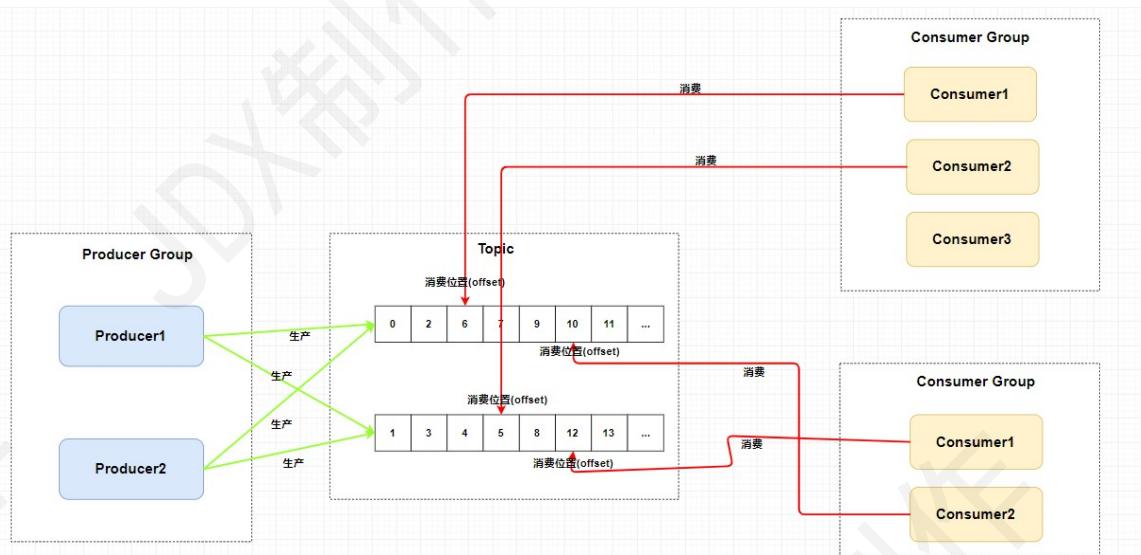
每个主题中都有多个队列(这里还不涉及到 `Broker`)，集群消费模式下，一个消费者集群多台机器共同消费一个 `topic` 的多个队列，**一个队列只会被一个消费者消费**。如果某个消费者挂掉，分组内其它消费者会接替挂掉的消费者继续消费。就像上图中 `consumer1` 和 `consumer2` 分别对应着两个队列，而 `Consumer3` 是没有队列对应的，所以一般来讲要控制 **消费者组中的消费者个数和主题中队列个数相同**。

当然也可以消费者个数小于队列个数，只不过不太建议。如下图。



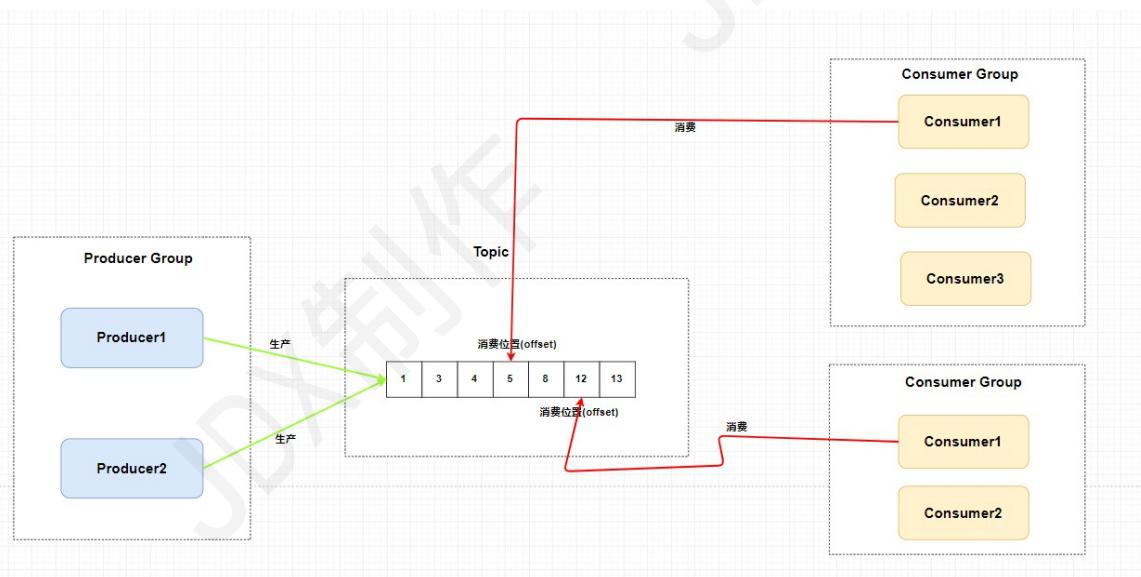
每个消费组在每个队列上维护一个消费位置，为什么呢？

因为我们刚刚画的仅仅是一个消费者组，我们知道在发布订阅模式中一般会涉及到多个消费者组，而每个消费者组在每个队列中的消费位置都是不同的。如果此时有多个消费者组，那么消息被一个消费者组消费完之后是不会删除的(因为其它消费者组也需要呀)，它仅仅是为每个消费者组维护一个 **消费位移 (offset)**，每次消费者组消费完会返回一个成功的响应，然后队列再把维护的消费位移加一，这样就不会出现刚刚消费过的消息再一次被消费了。



可能你还有一个问题，为什么一个主题中需要维护多个队列？

答案是 **提高并发能力**。的确，每个主题中只存在一个队列也是可行的。你想一下，如果每个主题中只存在一个队列，这个队列中也维护着每个消费者组的消费位置，这样也可以做到 **发布订阅模式**。如下图。



但是，这样我生产者是不是只能向一个队列发送消息？又因为需要维护消费位置所以一个队列只能对应一个消费者组中的消费者，这样是不是其他的 `Consumer` 就没有用武之地了？从这两个角度来讲，并发度一下子就小了很多。

所以总结来说，`RocketMQ` 通过使用在一个 `Topic` 中配置多个队列并且每个队列维护每个消费者组的消费位置 实现了 **主题模式/发布订阅模式**。

5.4 RocketMQ的架构图

讲完了消息模型，我们理解起 `RocketMQ` 的技术架构起来就容易多了。

`RocketMQ` 技术架构中有四大角色 `NameServer`、`Broker`、`Producer`、`Consumer`。我来向大家分别解释一下这四个角色是干啥的。

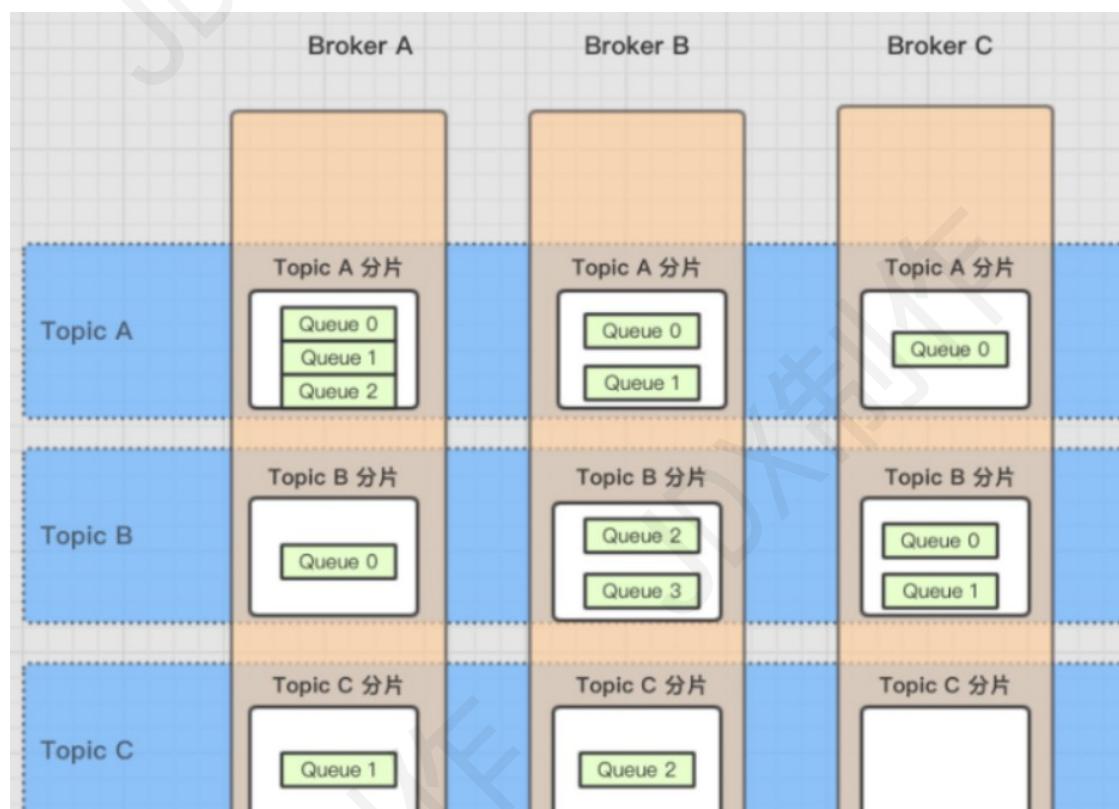
- `Broker`：主要负责消息的存储、投递和查询以及服务高可用保证。说白了就是消息队列服务器嘛，生产者生产消息到 `Broker`，消费者从 `Broker` 拉取消息并消费。

这里，我还得普及一下关于 `Broker`、`Topic` 和队列的关系。上面我讲解了 `Topic` 和队列的关系——一个 `Topic` 中存在多个队列，那么这个 `Topic` 和队列存放在哪呢？

一个 `Topic` 分布在多个 `Broker` 上，一个 `Broker` 可以配置多个 `Topic`，它们是多对多的关系。

如果某个 `Topic` 消息量很大，应该给它多配置几个队列(上文中提到了提高并发能力)，并且 **尽量多分布在不同 `Broker` 上，以减轻某个 `Broker` 的压力**。

`Topic` 消息量都比较均匀的情况下，如果某个 `Broker` 上的队列越多，则该 `Broker` 压力越大。

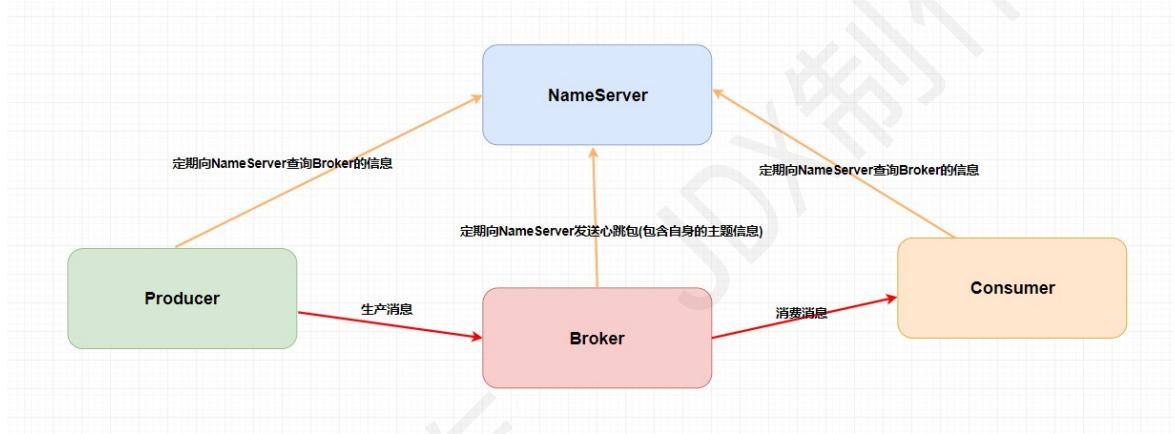


所以说我们需要配置多个Broker。

- `NameServer`：不知道你们有没有接触过 `zooKeeper` 和 `Spring cloud` 中的 `Eureka`，它其实也是一个 **注册中心**，主要提供两个功能：**Broker管理** 和 **路由信息管理**。说白了就是 `Broker` 会将自己的信息注册到 `NameServer` 中，此时 `NameServer` 就存放了很多 `Broker` 的信息(`Broker` 的路由表)，消费者和生产者就从 `NameServer` 中获取路由表然后照着路由表的信息和对应的 `Broker` 进行通信(生产者和消费者定期会向 `NameServer` 去查询相关的 `Broker` 的信息)。
- `Producer`：消息发布的角色，支持分布式集群方式部署。说白了就是生产者。

- `Consumer`: 消息消费的角色，支持分布式集群方式部署。支持以push推，pull拉两种模式对消息进行消费。同时也支持集群方式和广播方式的消费，它提供实时消息订阅机制。说白了就是消费者。

听完了上面的解释你可能会觉得，这玩意好简单。不就是这样的么？



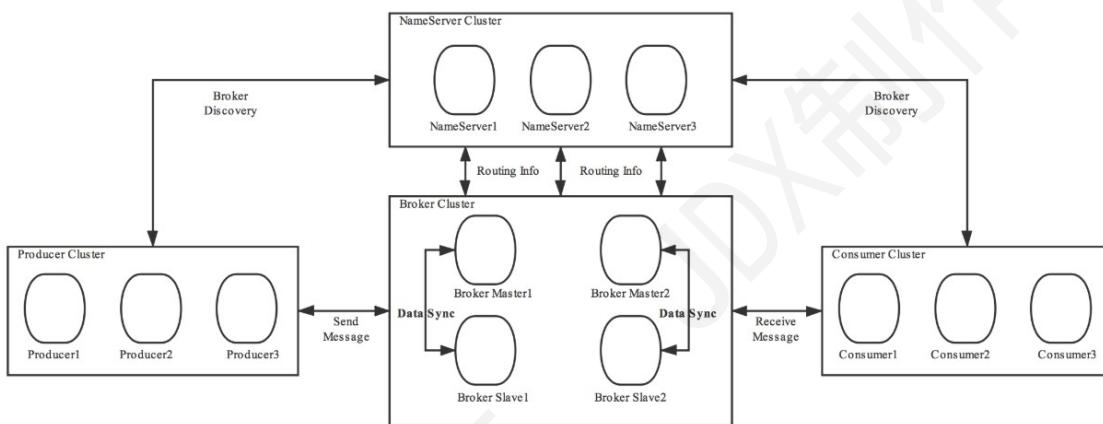
嗯？你可能会发现一个问题，这老家伙 `NameServer` 干啥用的，这多余吗？直接 `Producer`、`Consumer` 和 `Broker` 直接进行生产消息，消费消息不就好了么？

但是，我们上文提到过 `Broker` 是需要保证高可用的，如果整个系统仅仅靠着一个 `Broker` 来维持的话，那么这个 `Broker` 的压力会不会很大？所以我们需要使用多个 `Broker` 来保证 **负载均衡**。

如果说，我们的消费者和生产者直接和多个 `Broker` 相连，那么当 `Broker` 修改的时候必定会牵连着每个生产者和消费者，这样就会产生耦合问题，而 `NameServer` 注册中心就是用来解决这个问题的。

| 如果还不是很理解的话，可以去看我介绍 `spring cloud` 的那篇文章，其中介绍了 `Eureka` 注册中心。

当然，`RocketMQ` 中的技术架构肯定不止前面那么简单，因为上面图中的四个角色都是需要做集群的。我给出一张官网的架构图，大家尝试理解一下。



其实和我们最开始画的那张乞丐版的架构图也没什么区别，主要是一些细节上的差别。听我细细道来。

第一、我们的 `Broker` 做了**集群并且还进行了主从部署**，由于消息分布在各个 `Broker` 上，一旦某个 `Broker` 宕机，则该 `Broker` 上的消息读写都会受到影响。所以 `Rocketmq` 提供了 `master/slave` 的结构，`slave` 定时从 `master` 同步数据(同步刷盘或者异步刷盘)，如果 `master` 宕机，则 `slave` 提供**消费服务，但是不能写入消息**(后面我还会提到哦)。

第二、为了保证 HA，我们的 NameServer 也做了集群部署，但是请注意它是去中心化的。也就意味着它没有主节点，你可以很明显地看出 NameServer 的所有节点是没有进行 Info Replicate 的，在 RocketMQ 中是通过 单个Broker和所有NameServer保持长连接，并且在每隔30秒 Broker 会向所有 Nameserver 发送心跳，心跳包含了自身的 Topic 配置信息，这个步骤就对应这上面的 Routing Info。

第三、在生产者需要向 Broker 发送消息的时候，需要先从 NameServer 获取关于 Broker 的路由信息，然后通过 轮询 的方法去向每个队列中生产数据以达到 负载均衡 的效果。

第四、消费者通过 NameServer 获取所有 Broker 的路由信息后，向 Broker 发送 Pull 请求来获取消息数据。Consumer 可以以两种模式启动——广播（Broadcast）和集群（Cluster）。广播模式下，一条消息会发送给 同一个消费组中的所有消费者，集群模式下消息只会发送给一个消费者。

5.5 如何解决 顺序消费、重复消费

其实，这些东西都是我在介绍消息队列带来的一些副作用的时候提到的，也就是说，这些问题不仅仅挂钩于 RocketMQ，而是应该每个消息中间件都需要去解决的。

在上面我介绍 RocketMQ 的技术架构的时候我已经向你展示了 它是如何保证高可用的，这里不涉及运维方面的搭建，如果你感兴趣可以自己去官网上照着例子搭建属于你自己的 RocketMQ 集群。

其实 Kafka 的架构基本和 RocketMQ 类似，只是它注册中心使用了 zookeeper、它的 分区 就相当于 RocketMQ 中的 队列。还有一些小细节不同会在后面提到。

5.5.1 顺序消费

在上面的技术架构介绍中，我们已经知道了 RocketMQ 在主题上是无序的、它只有在队列层面才是保证有序 的。

这又扯到两个概念——普通顺序 和 严格顺序。

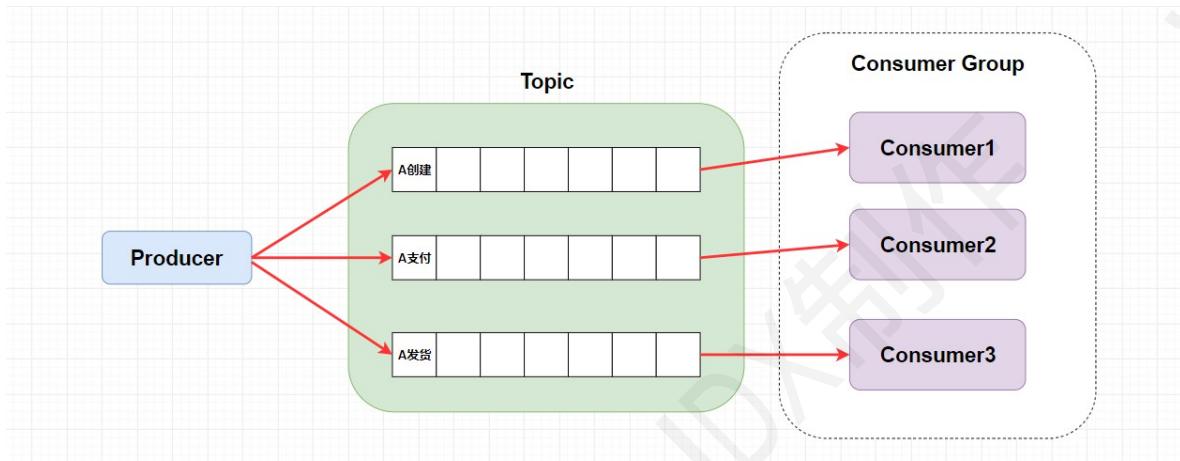
所谓普通顺序是指 消费者通过 同一个消费队列收到的消息是有顺序的，不同消息队列收到的消息则可能是无顺序的。普通顺序消息在 Broker 重启情况下不会保证消息顺序性(短暂时)。

所谓严格顺序是指 消费者收到的 所有消息 均是有顺序的。严格顺序消息 即使在异常情况下也会保证消息的顺序性。

但是，严格顺序看起来虽好，实现它可会付出巨大的代价。如果你使用严格顺序模式，Broker 集群中只要有一台机器不可用，则整个集群都不可用。你还用啥？现在主要场景也就在 binlog 同步。

一般而言，我们的 MQ 都是能容忍短暂的乱序，所以推荐使用普通顺序模式。

那么，我们现在使用了 普通顺序模式，我们从上面学习知道了在 Producer 生产消息的时候会进行轮询(取决于你的负载均衡策略)来向同一主题的不同消息队列发送消息。那么如果此时我有几个消息分别是同一个订单的创建、支付、发货，在轮询的策略下这 三个消息会被发送到不同队列，因为在不同的队列此时就无法使用 RocketMQ 带来的队列有序特性来保证消息有序性了。



那么，怎么解决呢？

其实很简单，我们需要处理的仅仅是将同一语义下的消息放入同一个队列(比如这里是同一个订单)，那我们就可以使用 **Hash取模法** 来保证同一个订单在同一个队列中就行了。

5.5.2 重复消费

emmm，就两个字——**幂等**。在编程中一个幂等操作的特点是其任意多次执行所产生的影响均与一次执行的影响相同。比如说，这个时候我们有一个订单的处理积分的系统，每当来一个消息的时候它就负责为创建这个订单的用户的积分加上相应的数值。可是有一次，消息队列发送给订单系统 FrancisQ 的订单信息，其要求是给 FrancisQ 的积分加上 500。但是积分系统在收到 FrancisQ 的订单信息处理完成之后返回给消息队列处理成功的信息的时候出现了网络波动(当然还有很多种情况，比如Broker意外重启等等)，这条回应没有发送成功。

那么，消息队列没收到积分系统的回应会不会尝试重发这个消息？问题就来了，我再发这个消息，万一它又给 FrancisQ 的账户加上 500 积分怎么办呢？

所以我们需要给我们的消费者实现 **幂等**，也就是对同一个消息的处理结果，执行多少次都不变。

那么如何给业务实现幂等呢？这个还是需要结合具体的业务的。你可以使用 **写入 Redis** 来保证，因为 **Redis** 的 **key** 和 **value** 就是天然支持幂等的。当然还有使用 **数据库插入法**，基于数据库的唯一键来保证重复数据不会被插入多条。

不过最主要的还是需要 **根据特定场景使用特定的解决方案**，你要知道你的消息消费是否是完全不可重复消费还是可以忍受重复消费的，然后再选择强校验和弱校验的方式。毕竟在 CS 领域还是很少有技术银弹的说法。

而在整个互联网领域，幂等不仅仅适用于消息队列的重复消费问题，这些实现幂等的方法，也同样适用于，在其他场景中来解决重复请求或者重复调用的问题。比如将 HTTP 服务设计成幂等的，解决前端或者 APP 重复提交表单数据的问题，也可以将一个微服务设计成幂等的，解决 **RPC** 框架自动重试导致的重复调用问题。

5.6 分布式事务

如何解释分布式事务呢？事务大家都知道吧？**要么都执行要么都不执行**。在同一个系统中我们可以轻松地实现事务，但是在分布式架构中，我们有很多服务是部署在不同系统之间的，而不同服务之间又需要进行调用。比如此时我下订单然后增加积分，如果保证不了分布式事务的话，就会出现 A 系统下了订单，但是 B 系统增加积分失败或者 A 系统没有下订单，B 系统却增加了积分。前者对用户不友好，后者对运营商不利，这是我们都不愿意见到的。

那么，如何去解决这个问题呢？

如今比较常见的分布式事务实现有 2PC、TCC 和事务消息(half 半消息机制)。每一种实现都有其特定的使用场景，但是也有各自的问题，**都不是完美的解决方案**。

在RocketMQ中使用的是**事务消息加上事务反查机制**来解决分布式事务问题的。我画了张图，大家可以对照着图进行理解。



在第一步发送的 half 消息，它的意思是 在事务提交之前，对于消费者来说，这个消息是不可见的。

那么，如何做到写入消息但是对用户不可见呢？RocketMQ事务消息的做法是：如果消息是half消息，将备份原消息的主题与消息消费队列，然后**改变主题**为RMQ_SYS_TRANS_HALF_TOPIC。由于消费组未订阅该主题，故消费端无法消费half类型的消息，**然后RocketMQ会开启一个定时任务，从Topic为RMQ_SYS_TRANS_HALF_TOPIC中拉取消息进行消费**，根据生产者组获取一个服务提供者发送回查事务状态请求，根据事务状态来决定是提交或回滚消息。

你可以试想一下，如果没有从第5步开始的**事务反查机制**，如果出现网路波动第4步没有发送成功，这样就会产生MQ不知道是不是需要给消费者消费的问题，他就像一个无头苍蝇一样。在RocketMQ中就是使用的上述的事务反查来解决的，而在Kafka中通常是直接抛出一个异常让用户来自行解决。

你还需要注意的是，在MQ Server指向系统B的操作已经和系统A不相关了，也就是说在消息队列中的分布式事务是——**本地事务和存储消息到消息队列才是同一个事务**。这样也就产生了事务的**最终一致性**，因为整个过程是异步的，**每个系统只要保证它自己那一部分的事务就行了**。

5.7 消息堆积问题

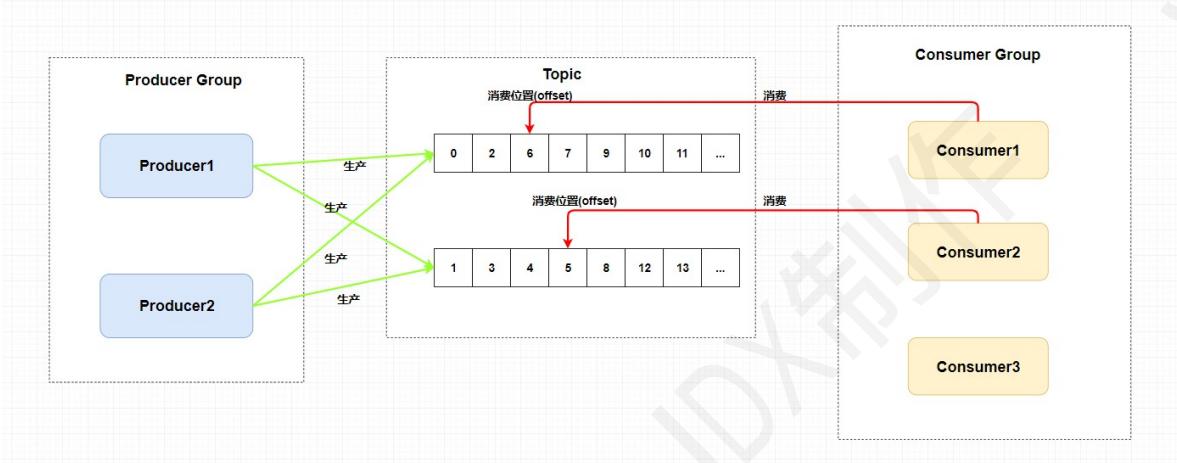
在上面我们提到了消息队列一个很重要的功能——**削峰**。那么如果这个峰值太大了导致消息堆积在队列中怎么办呢？

其实这个问题可以将它广义化，因为产生消息堆积的根源其实就只有两个——生产者生产太快或者消费者消费太慢。

我们可以从多个角度去思考解决这个问题，当流量到峰值的时候是因为生产者生产太快，我们可以使用一些**限流降级**的方法，当然你也可以增加多个消费者实例去水平扩展增加消费能力来匹配生产的激增。如果消费者消费过慢的话，我们可以先检查**是否是消费者出现了大量的消费错误**，或者打印一下日志查看是否是哪一个线程卡死，出现了锁资源不释放等等的问题。

当然，最快速解决消息堆积问题的方法还是增加消费者实例，不过**同时你还需要增加每个主题的队列数量**。

别忘了在RocketMQ中，**一个队列只会被一个消费者消费**，如果你仅仅是增加消费者实例就会出现我一开始给你画架构图的那种情况。



5.8 回溯消费

回溯消费是指 `Consumer` 已经消费成功的消息，由于业务上需求需要重新消费，在 `RocketMQ` 中，`Broker` 在向 `Consumer` 投递成功消息后，**消息仍然需要保留**。并且重新消费一般是按照时间维度，例如由于 `Consumer` 系统故障，恢复后需要重新消费1小时前的数据，那么 `Broker` 要提供一种机制，可以按照时间维度来回退消费进度。`RocketMQ` 支持按照时间回溯消费，时间维度精确到毫秒。

5.9 RocketMQ 的刷盘机制

上面我讲了那么多的 `RocketMQ` 的架构和设计原理，你有没有好奇

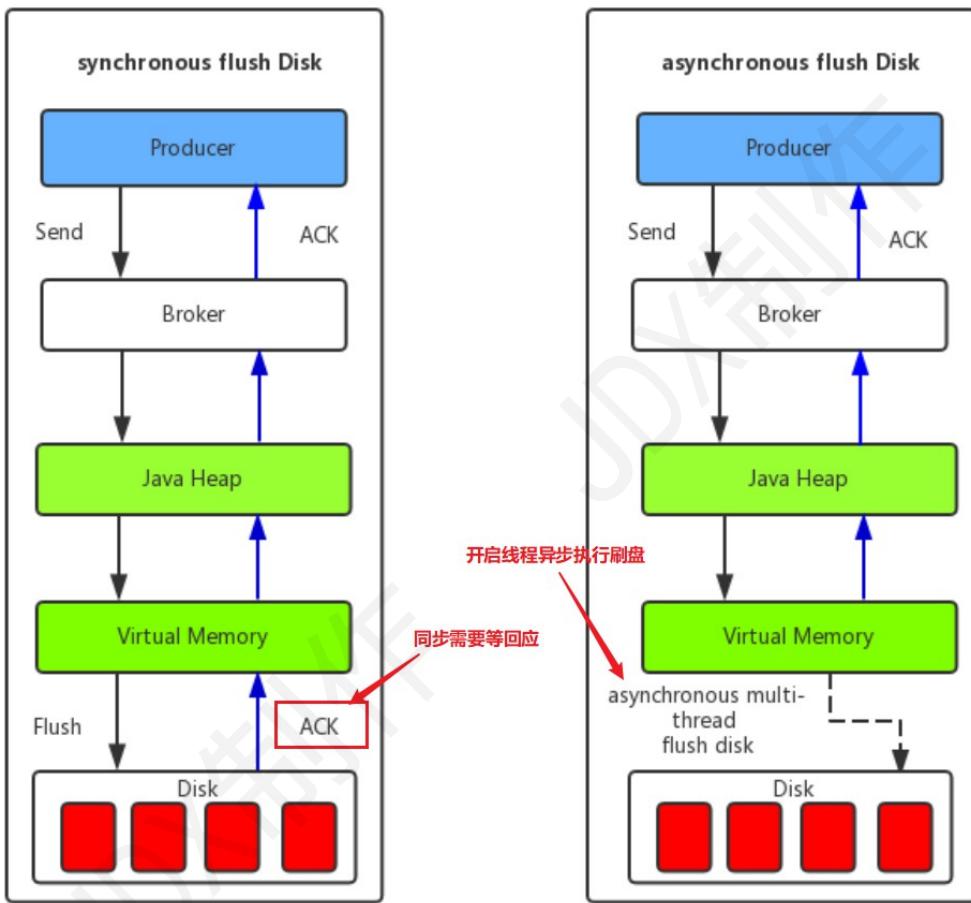
在 `Topic` 中的 队列是以什么样的形式存在的？

队列中的消息又是如何进行存储持久化的呢？

我在上文中提到的 同步刷盘 和 异步刷盘 又是什么呢？它们会给持久化带来什么样的影响呢？

下面我将给你们一一解释。

5.9.1 同步刷盘和异步刷盘



如上图所示，在同步刷盘中需要等待一个刷盘成功的 **ACK**，同步刷盘对 **MQ** 消息可靠性来说是一种不错的保障，但是 **性能上会有较大影响**，一般地适用于金融等特定业务场景。

而异步刷盘往往是开启一个线程去异步地执行刷盘操作。消息刷盘采用后台异步线程提交的方式进行，降低了读写延迟，提高了 **MQ** 的性能和吞吐量，一般适用于如发验证码等对于消息保证要求不太高的业务场景。

一般地，**异步刷盘只有在 Broker 意外宕机的时候会丢失部分数据**，你可以设置 **Broker** 的参数 `FlushDiskType` 来调整你的刷盘策略(ASYNC_FLUSH 或者 SYNC_FLUSH)。

5.9.2 同步复制和异步复制

上面的同步刷盘和异步刷盘是在单个结点层面的，而同步复制和异步复制主要是指的 **Broker** 主从模式下，主节点返回消息给客户端的时候是否需要同步从节点。

- **同步复制：**也叫“同步双写”，也就是说，**只有消息同步双写到主从结点上时才返回写入成功**。
- **异步复制：****消息写入主节点之后就直接返回写入成功**。

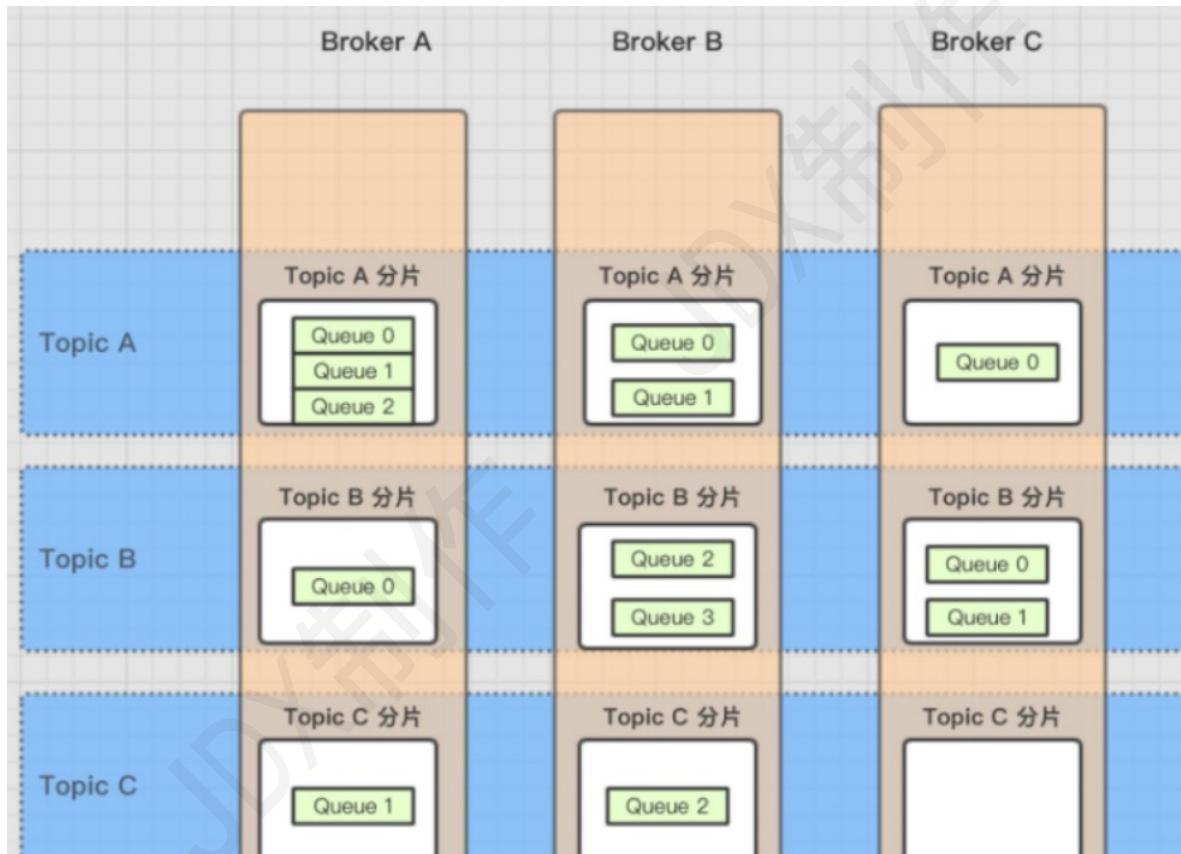
然而，很多事情是没有完美的方案的，就比如我们进行消息写入的节点越多就更能保证消息的可靠性，但是随之的性能也会下降，所以需要程序员根据特定业务场景去选择适应的主从复制方案。

那么，**异步复制会不会也像异步刷盘那样影响消息的可靠性呢？**

答案是不会的，因为两者就是不同的概念，对于消息可靠性是通过不同的刷盘策略保证的，而像异步同步复制策略仅仅是影响到了 **可用性**。为什么呢？其主要原因是 **RocketMQ 是不支持自动主从切换的，当主节点挂掉之后，生产者就不能再给这个主节点生产消息了**。

比如这个时候采用异步复制的方式，在主节点还未发送完需要同步的消息的时候主节点挂掉了，这个时候从节点就少了一部分消息。但是此时生产者无法再给主节点生产消息了，**消费者可以自动切换到从节点进行消费**(仅仅是消费)，所以在主节点挂掉的时间只会产生主从结点短暂的消息不一致的情况，降低了可用性，而当主节点重启之后，从节点那部分未来得及复制的消息还会继续复制。

在单主从架构中，如果一个主节点挂掉了，那么也就意味着整个系统不能再生产了。那么这个可用性的问题能否解决呢？**一个主从不行那就多个主从的呗**，别忘了在我们最初的架构图中，每个 Topic 是分布在不同 Broker 中的。



但是这种复制方式同样也会带来一个问题，那就是无法保证 **严格顺序**。在上文中我们提到了如何保证的消息顺序性是通过将一个语义的消息发送在同一个队列中，使用 Topic 下的队列来保证顺序性的。如果此时我们主节点A负责的是订单A的一系列语义消息，然后它挂了，这样其他节点是无法代替主节点A的，如果我们任意节点都可以存入任何消息，那就没有顺序性可言了。

而在 RocketMQ 中采用了 Dledger 解决这个问题。他要求在写入消息的时候，要求**至少消息复制到半数以上的节点之后**，才给客户端返回写入成功，并且它是支持通过选举来动态切换主节点的。这里就不展开说明了，读者可以自己去了解。

也不是说 Dledger 是个完美的方案，至少在 Dledger 选举过程中是无法提供服务的，而且他必须要使用三个节点或以上，如果多数节点同时挂掉他也是无法保证可用性的，而且要求消息复制板书以上节点的效率和直接异步复制还是有一定的差距的。

5.9.3 存储机制

还记得上面我们一开始的三个问题吗？到这里第三个问题已经解决了。

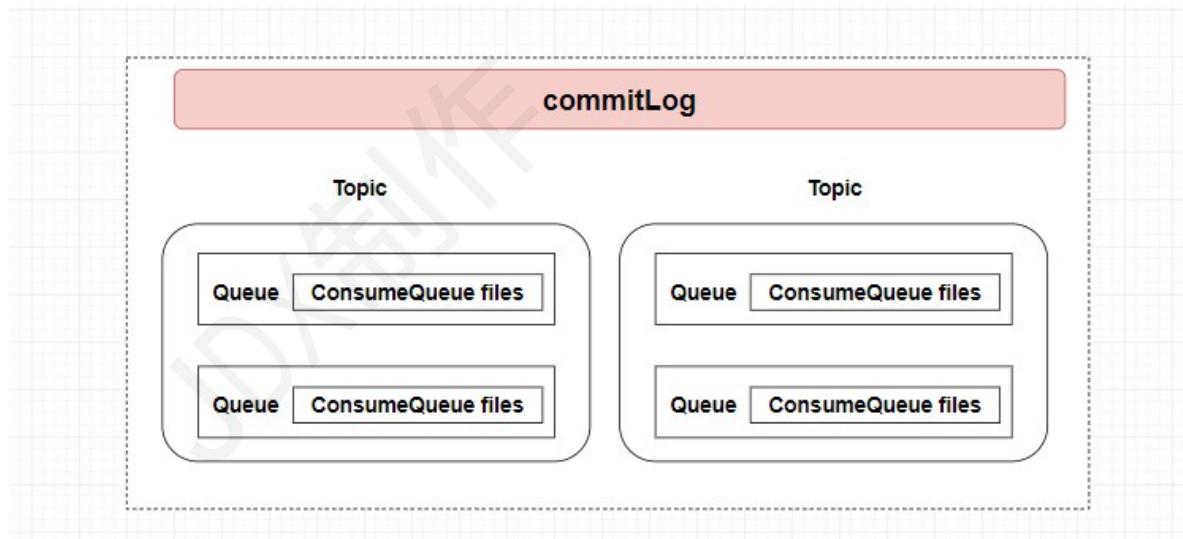
但是，在 Topic 中的 队列是以什么样的形式存在的？队列中的消息又是如何进行存储持久化的呢？还未解决，其实这里涉及到了 RocketMQ 是如何设计它的存储结构了。我首先想大家介绍 RocketMQ 消息存储架构中的三大角色—— CommitLog 、 ConsumeQueue 和 IndexFile 。

- CommitLog：消息主体以及元数据的存储主体，存储 Producer 端写入的消息主体内容，消息内容不是定长的。单个文件大小默认1G，文件名长度为20位，左边补零，剩余为起始偏移量，比如 00000000000000000000 代表了第一个文件，起始偏移量为0，文件大小为1G=1073741824；当第一个文件写满了，第二个文件为 00000000001073741824，起始偏移量为 1073741824，以此类推。消息主要是**顺序写入日志文件**，当文件满了，写入下一个文件。
- ConsumeQueue：消息消费队列，引入的目的主要是提高消息消费的性能(我们再前面也讲了)，由于 RocketMQ 是基于主题 Topic 的订阅模式，消息消费是针对主题进行的，如果要遍历 commitlog 文件中根据 Topic 检索消息是非常低效的。 Consumer 即可根据 ConsumeQueue 来

查找待消费的消息。其中，`ConsumeQueue`（逻辑消费队列）作为消费消息的索引，保存了指定 `Topic` 下的队列消息在 `CommitLog` 中的起始物理偏移量 `offset`，消息大小 `size` 和消息 `Tag` 的 `HashCode` 值。`consumequeue` 文件可以看成是基于 `topic` 的 `commitlog` 索引文件，故 `consumequeue` 文件夹的组织方式如下：topic/queue/file三层组织结构，具体存储路径为：`$HOME/store/consumequeue/{topic}/{queueId}/{fileName}`。同样 `consumequeue` 文件采取定长设计，每一个条目共20个字节，分别为8字节的 `commitlog` 物理偏移量、4字节的消息长度、8字节tag `hashcode`，单个文件由30W个条目组成，可以像数组一样随机访问每一个条目，每个 `ConsumeQueue` 文件大小约5.72M；

- `IndexFile`：`IndexFile`（索引文件）提供了一种可以通过key或时间区间来查询消息的方法。这里只做科普不做详细介绍。

总结来说，整个消息存储的结构，最主要的就是 `CommitLog` 和 `ConsumeQueue`。而 `ConsumeQueue` 你可以大概理解为 `Topic` 中的队列。

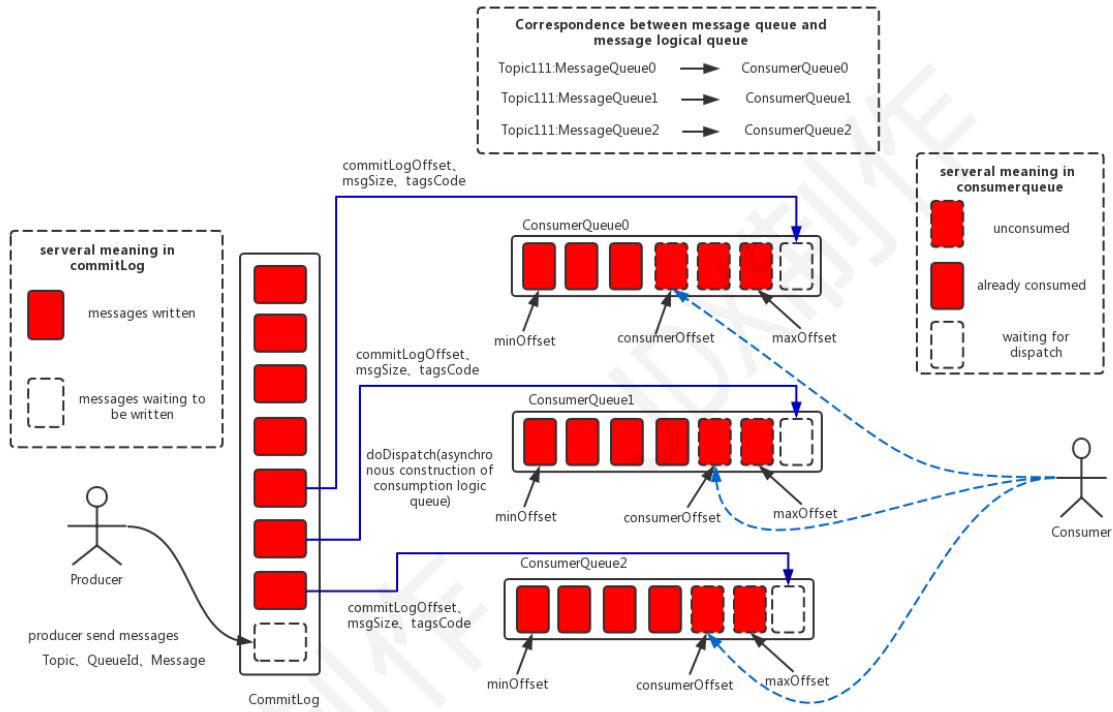


RocketMQ 采用的是 **混合型的存储结构**，即为 `Broker` 单个实例下所有的队列共用一个日志数据文件来存储消息。有意思的是在同样高并发的 `Kafka` 中会为每个 `Topic` 分配一个存储文件。这就有点类似于我们有一大堆书需要装上书架，RocketMQ 是不分书的种类直接成批的塞上去的，而 Kafka 是将书本放入指定的分类区域的。

而 RocketMQ 为什么要这么做呢？原因是 **提高数据的写入效率**，不分 `Topic` 意味着我们有更大的几率获取 **成批** 的消息进行数据写入，但也会带来一个麻烦就是读取消息的时候需要遍历整个大文件，这是非常耗时的。

所以，在 RocketMQ 中又使用了 `ConsumeQueue` 作为每个队列的索引文件来 **提升读取消息的效率**。我们可以直接根据队列的消息序号，计算出索引的全局位置（索引序号*索引固定长度20），然后直接读取这条索引，再根据索引中记录的消息的全局位置，找到消息。

讲到这里，你可能对 RocketMQ 的存储架构还有些模糊，没事，我们结合着图来理解一下。



emmm，是不是有一点复杂，看英文图片和英文文档的时候就不要怂，硬着头皮往下看就行。

| 如果上面没看懂的读者一定要认真看下面的流程分析！

首先，在最上面的那一块就是我刚刚讲的你现在可以直接把 `ConsumerQueue` 理解为 `Queue`。

在图中最左边说明了 **红色方块** 代表被写入的消息，虚线方块代表等待被写入的。左边的生产者发送消息会指定 `Topic`、`QueueId` 和具体消息内容，而在 `Broker` 中管你是哪门子消息，他直接 **全部顺序存储到了 CommitLog**。而根据生产者指定的 `Topic` 和 `QueueId` 将这条消息本身在 `CommitLog` 的偏移(offset)，消息本身大小，和tag的hash值存入对应的 `ConsumeQueue` 索引文件中。而在每个队列中都保存了 `ConsumeOffset` 即每个消费者组的消费位置(我在架构那里提到了，忘了的同学可以回去看一下)，而消费者拉取消息进行消费的时候只需要根据 `ConsumeOffset` 获取下一个未被消费的消息就行了。

上述就是我对于整个消息存储架构的大概理解(这里不涉及到一些细节讨论，比如稀疏索引等等问题)，希望对你有帮助。

因为有一个知识点因为写嗨了忘讲了，想想在哪里加也不好，所以我留给大家去思考一下吧。

为什么 `CommitLog` 文件要设计成固定大小的长度呢？提醒：**内存映射机制**。

5.10 总结

这篇文章中我主要想大家介绍了

1. 消息队列出现的原因
2. 消息队列的作用(异步，解耦，削峰)
3. 消息队列带来的一系列问题(消息堆积、重复消费、顺序消费、分布式事务等等)
4. 消息队列的两种消息模型——队列和主题模式
5. 分析了 `RocketMQ` 的技术架构(`NameServer`、`Broker`、`Producer`、`Consumer`)
6. 结合 `RocketMQ` 回答了消息队列副作用的解决方案
7. 介绍了 `RocketMQ` 的存储机制和刷盘策略。

6. Kafka

6.1 Kafka 简介

6.1.1 Kafka 创建背景

Kafka 是一个消息系统，原本开发自 LinkedIn，用作 LinkedIn 的活动流（Activity Stream）和运营数据处理管道（Pipeline）的基础。现在它已被多家不同类型的公司作为多种类型的数据管道和消息系统使用。

活动流数据是几乎所有站点在其网站使用情况做报表时都要用到的数据中最常规的部分。活动数据包括页面访问量（Page View）、被查看内容方面的信息以及搜索情况等内容。这种数据通常的处理方式是先把各种活动以日志的形式写入某种文件，然后周期性地对这些文件进行统计分析。**运营数据**指的是服务器的性能数据（CPU、IO 使用率、请求时间、服务日志等等数据）。运营数据的统计方法种类繁多。

近年来，活动和运营数据处理已经成为了网站软件产品特性中一个至关重要的组成部分，这就需要一套稍微更加复杂的基础设施对其提供支持。

6.1.2 Kafka 简介

Kafka 是一种分布式的，基于发布 / 订阅的消息系统。

主要设计目标如下：

- 以时间复杂度为 O(1) 的方式提供消息持久化能力，即使对 TB 级以上数据也能保证常数时间复杂度的访问性能。
- 高吞吐率。即使在非常廉价的商用机器上也能做到单机支持每秒 100K 条以上消息的传输。
- 支持 Kafka Server 间的消息分区，及分布式消费，同时保证每个 Partition 内的消息顺序传输。
- 同时支持离线数据处理和实时数据处理。
- Scale out：支持在线水平扩展。

6.1.3 Kafka 基础概念

概念一：生产者与消费者



对于 Kafka 来说客户端有两种基本类型：

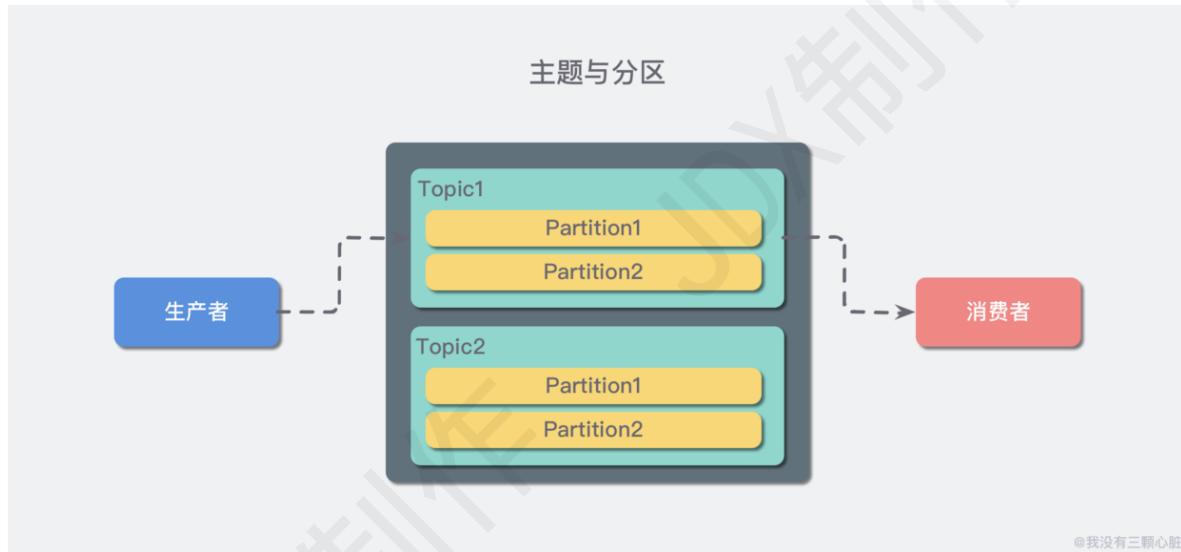
1. 生产者（Producer）
2. 消费者（Consumer）。

©我没有三颗心脏

除此之外，还有用来做数据集成的 Kafka Connect API 和流式处理的 Kafka Streams 等高阶客户端，但这些高阶客户端底层仍然是生产者和消费者API，它们只不过是在上层做了封装。

这很容易理解，生产者（也称为发布者）创建消息，而消费者（也称为订阅者）负责消费or读取消息。

概念二：主题 (Topic) 与分区 (Partition)



©我没有三颗心脏

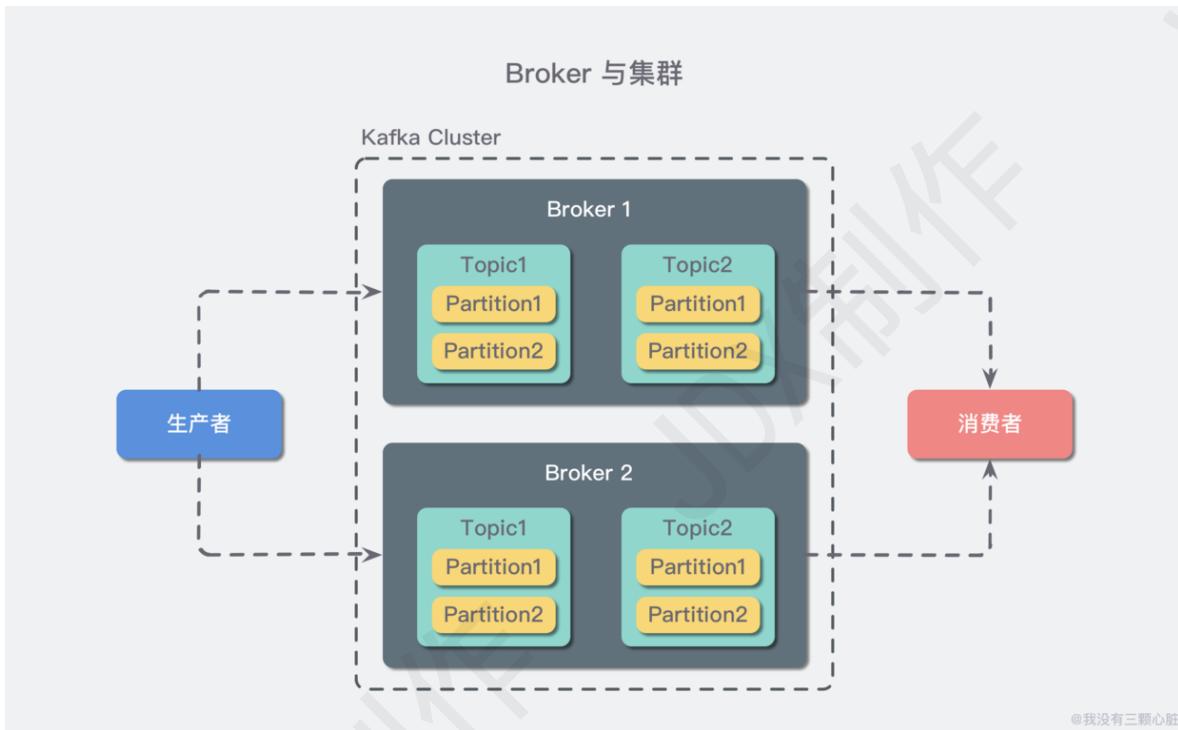
在 Kafka 中，消息以**主题 (Topic)** 来分类，每一个主题都对应一个「消息队列」，这有点儿类似于数据库中的表。但是如果我们把所有同类的消息都塞入到一个“中心”队列中，势必缺少可伸缩性，无论是生产者/消费者数目的增加，还是消息数量的增加，都可能耗尽系统的性能或存储。

我们使用一个生活中的例子来说明：现在 A 城市生产的某商品需要运输到 B 城市，走的是公路，那么单通道的高速公路不论是在「A 城市商品增多」还是「现在 C 城市也要往 B 城市运输东西」这样的情况下都会出现「吞吐量不足」的问题。所以我们现在引入**分区 (Partition)** 的概念，类似“允许多修几条道”的方式对我们的主题完成了水平扩展。

概念三：Broker 和集群 (Cluster)

一个 Kafka 服务器也称为 Broker，它接受生产者发送的消息并存入磁盘；Broker 同时服务消费者拉取分区消息的请求，返回目前已经提交的消息。使用特定的机器硬件，一个 Broker 每秒可以处理成千上万的分区和百万量级的消息。（现在动不动就百万量级..我特地去查了一把，好像确实集群的情况下吞吐量挺高的..嗯..）

若干个 Broker 组成一个集群 (Cluster)，其中集群内某个 Broker 会成为集群控制器 (Cluster Controller)，它负责管理集群，包括分配分区到 Broker、监控 Broker 故障等。在集群内，一个分区由一个 Broker 负责，这个 Broker 也称为这个分区的 Leader；当然一个分区可以被复制到多个 Broker 上来实现冗余，这样当存在 Broker 故障时可以将其分区重新分配到其他 Broker 来负责。下图是一个样例：



Kafka 的一个关键性质是日志保留 (retention)，我们可以配置主题的消息保留策略，譬如只保留一段时间的日志或者只保留特定大小的日志。当超过这些限制时，老的消息会被删除。我们也可以针对某个主题单独设置消息过期策略，这样对于不同应用可以实现个性化。

概念四：多集群

随着业务发展，我们往往需要多集群，通常处于下面几个原因：

- 基于数据的隔离；
- 基于安全的隔离；
- 多数据中心（容灾）

当构建多个数据中心时，往往需要实现消息互通。举个例子，假如用户修改了个人资料，那么后续的请求无论被哪个数据中心处理，这个更新需要反映出来。又或者，多个数据中心的数据需要汇总到一个总控中心来做数据分析。

上面说的分区复制冗余机制只适用于同一个 Kafka 集群内部，对于多个 Kafka 集群消息同步可以使用 Kafka 提供的 MirrorMaker 工具。本质上来说，MirrorMaker 只是一个 Kafka 消费者和生产者，并使用一个队列连接起来而已。它从一个集群中消费消息，然后往另一个集群生产消息。

6.2 Kafka 的设计与实现

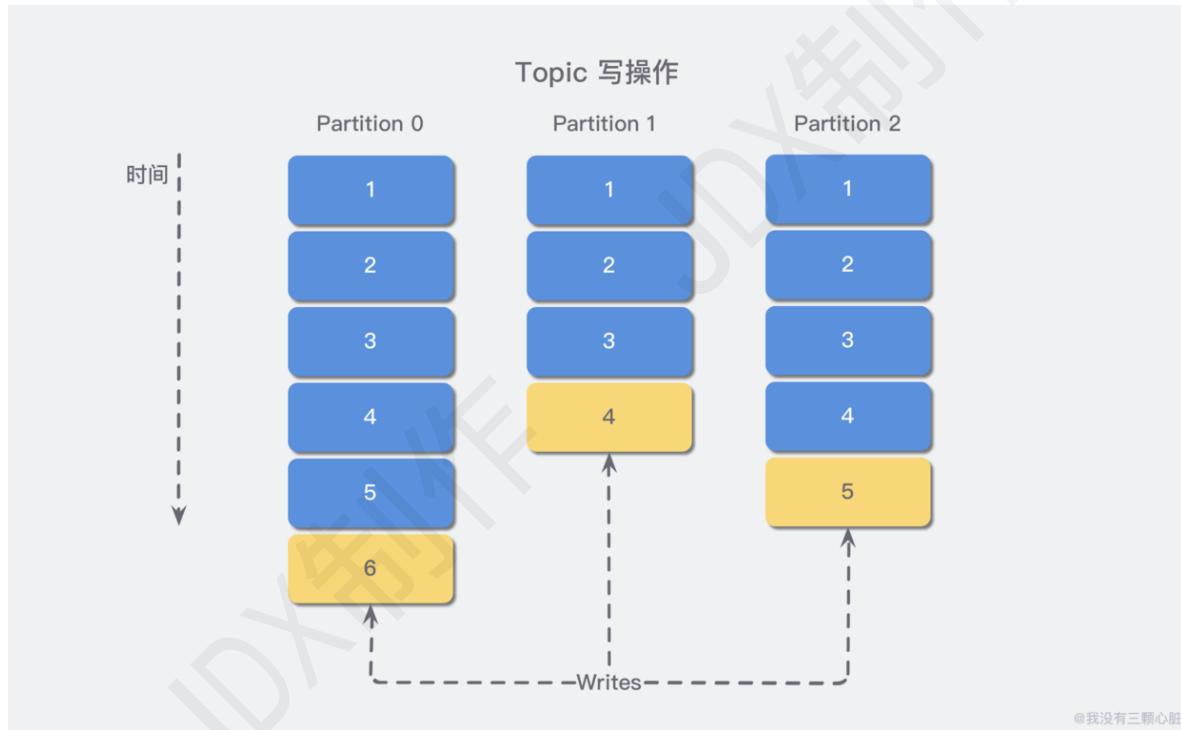
上面我们知道了 Kafka 中的一些基本概念，但作为一个成熟的「消息队列」中间件，其中有许多有意思的设计值得我们思考，下面我们简单列举一些。

6.2.1 讨论一：Kafka 存储在文件系统上

是的，您首先应该知道 Kafka 的消息是存在于文件系统之上的。Kafka 高度依赖文件系统来存储和缓存消息，一般的人认为“磁盘是缓慢的”，所以对这样的设计持有怀疑态度。实际上，磁盘比人们预想的快很多也慢很多，这取决于它们如何被使用；一个好的磁盘结构设计可以使之跟网络速度一样快。

现代的操作系统针对磁盘的读写已经做了一些优化方案来加快磁盘的访问速度。比如，**预读**会提前将一个比较大的磁盘快读入内存。**后写**会将很多小的逻辑写操作合并起来组合成一个大的物理写操作。并且，操作系统还会将主内存剩余的所有空闲内存空间都用作**磁盘缓存**，所有的磁盘读写操作都会经过统一的磁盘缓存（除了直接 I/O 会绕过磁盘缓存）。综合这几点优化特点，**如果是针对磁盘的顺序访问，某些情况下它可能比随机的内存访问都要快，甚至可以和网络的速度相差无几。**

上述的 Topic 其实是逻辑上的概念，面对消费者和生产者，物理上存储的其实是 Partition，每一个 Partition 最终对应一个目录，里面存储所有的消息和索引文件。默认情况下，每一个 Topic 在创建时如果不指定 Partition 数量时只会创建 1 个 Partition。比如，我创建了一个 Topic 名字为 test，没有指定 Partition 的数量，那么会默认创建一个 test-0 的文件夹，这里的命名规则是：<topic_name>-<partition_id>。



任何发布到 Partition 的消息都会被追加到 Partition 数据文件的尾部，这样的顺序写磁盘操作让 Kafka 的效率非常高（经验证，顺序写磁盘效率比随机写内存还要高，这是 Kafka 高吞吐率的一个很重要的保证）。

每一条消息被发送到 Broker 中，会根据 Partition 规则选择被存储到哪一个 Partition。如果 Partition 规则设置的合理，所有消息可以均匀分布到不同的 Partition 中。

6.2.2 讨论二：Kafka 中的底层存储设计

假设我们现在 Kafka 集群只有一个 Broker，我们创建 2 个 Topic 名称分别为：「topic1」和「topic2」，Partition 数量分别为 1、2，那么我们的根目录下就会创建如下三个文件夹：

```
| --topic1-0  
| --topic2-0  
| --topic2-1
```

在 Kafka 的文件存储中，同一个 Topic 下有多个不同的 Partition，每个 Partition 都为一个目录，而每一个目录又被平均分配成多个大小相等的 **Segment File** 中，Segment File 又由 index file 和 data file 组成，他们总是成对出现，后缀 “.index” 和 “.log” 分别表示 Segment 索引文件和数据文件。

现在假设我们设置每个 Segment 大小为 500 MB，并启动生产者向 topic1 中写入大量数据，topic1-0 文件夹中就会产生类似如下的一些文件：

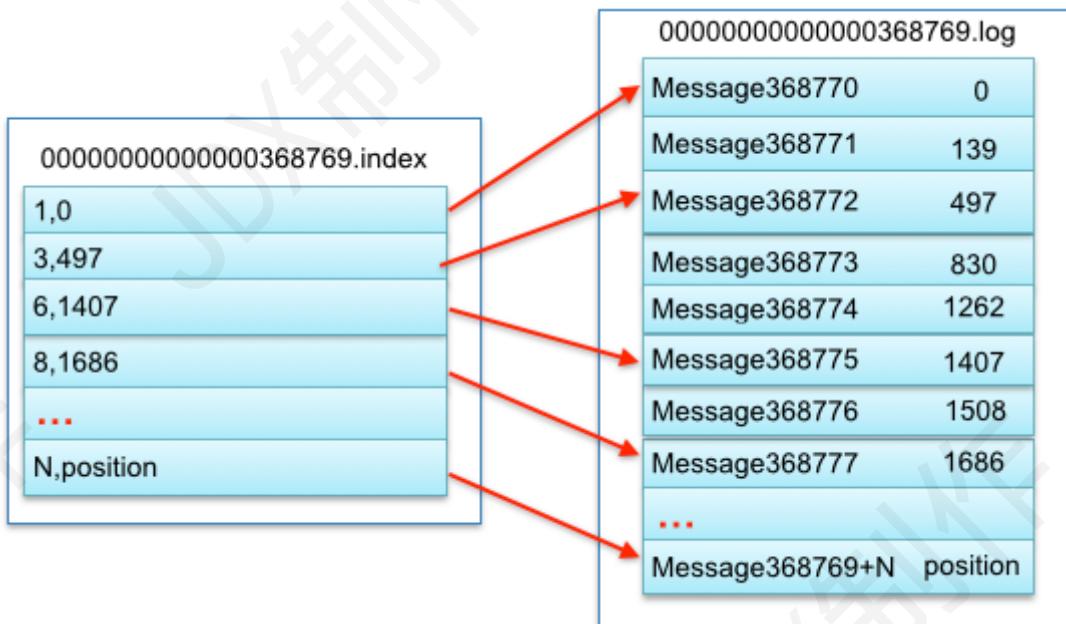
```

| --topic1-0
|   |--00000000000000000000000000000000.index
|   |--00000000000000000000000000000000.log
|   |--00000000000000000000000000000000368769.index
|   |--00000000000000000000000000000000368769.log
|   |--00000000000000000000000000000000737337.index
|   |--00000000000000000000000000000000737337.log
|   |--000000000000000000000000000000001105814.index | --000000000000000000000000000000001105814.log
| --topic2-0
| --topic2-1

```

Segment 是 Kafka 文件存储的最小单位。 Segment 文件命名规则：Partition 全局的第一个 Segment 从 0 开始，后续每个 Segment 文件名为上一个 Segment 文件最后一条消息的 offset 值。数值最大为 64 位 long 大小，19 位数字字符长度，没有数字用0填充。如 00000000000000000000000000000000368769.index 和 00000000000000000000000000000000368769.log。

以上面的一对 Segment File 为例，说明一下索引文件和数据文件对应关系：



其中以索引文件中元数据 `<3, 497>` 为例，依次在数据文件中表示第 3 个 message（在全局 Partition 表示第 $368769 + 3 = 368772$ 个 message）以及该消息的物理偏移地址为 497。

注意该 index 文件并不是从0开始，也不是每次递增1的，这是因为 Kafka 采取稀疏索引存储的方式，每隔一定字节的数据建立一条索引，它减少了索引文件大小，使得能够把 index 映射到内存，降低了查询时的磁盘 IO 开销，同时也并没有给查询带来太多的时间消耗。

因为其文件名为上一个 Segment 最后一条消息的 offset，所以当需要查找一个指定 offset 的 message 时，通过在所有 segment 的文件名中进行二分查找就能找到它归属的 segment，再在其 index 文件中找到其对应到文件上的物理位置，就能拿出该 message。

由于消息在 Partition 的 Segment 数据文件中是顺序读写的，且消息消费后不会删除（删除策略是针对过期的 Segment 文件），这种顺序磁盘 IO 存储设计是 Kafka 高性能很重要的原因。

Kafka 是如何准确的知道 message 的偏移的呢？这是因为在 Kafka 定义了标准的数据存储结构，在 Partition 中的每一条 message 都包含了以下三个属性：

- offset：表示 message 在当前 Partition 中的偏移量，是一个逻辑上的值，唯一确定了 Partition 中的一条 message，可以简单的认为是一个 id；
- MessageSize：表示 message 内容 data 的大小；

- data: message 的具体内容

6.2.3 讨论三：生产者设计概要

当我们发送消息之前，先问几个问题：每条消息都是很关键且不能容忍丢失么？偶尔重复消息可以么？我们关注的是消息延迟还是写入消息的吞吐量？

举个例子，有一个信用卡交易处理系统，当交易发生时会发送一条消息到 Kafka，另一个服务来读取消息并根据规则引擎来检查交易是否通过，将结果通过 Kafka 返回。对于这样的业务，消息既不能丢失也不能重复，由于交易量大因此吞吐量需要尽可能大，延迟可以稍微高一点。

再举个例子，假如我们需要收集用户在网页上的点击数据，对于这样的场景，少量消息丢失或者重复是可以容忍的，延迟多大都不重要只要不影响用户体验，吞吐则根据实时用户数来决定。

不同的业务需要使用不同的写入方式和配置。具体的方式我们在这里不做讨论，现在先看下生产者写消息的基本流程：

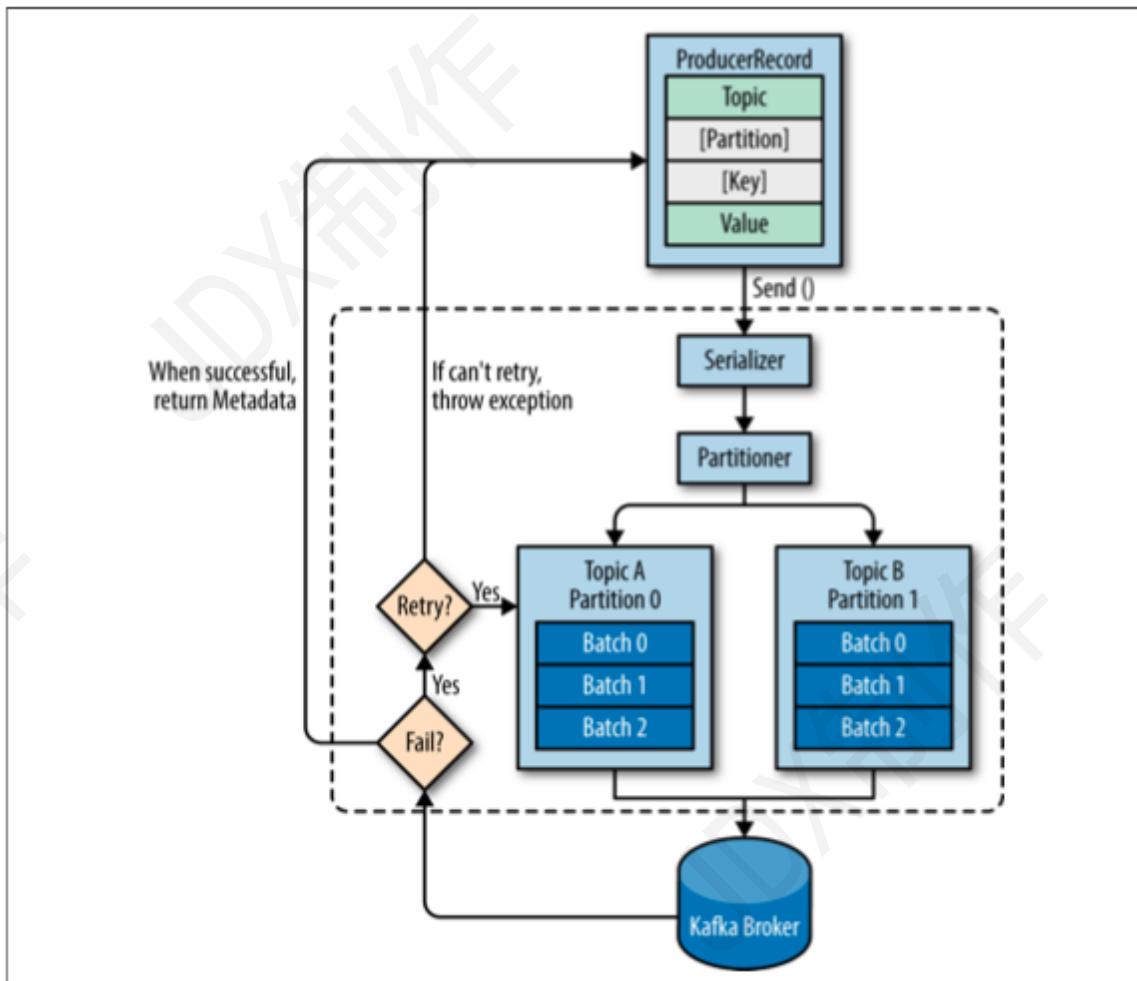


Figure 3-1. High-level overview of Kafka producer components

流程如下：

1. 首先，我们需要创建一个ProducerRecord，这个对象需要包含消息的主题（topic）和值（value），可以选择性指定一个键值（key）或者分区（partition）。
2. 发送消息时，生产者会对键值和值序列化成字节数组，然后发送到分配器（partitioner）。
3. 如果我们指定了分区，那么分配器返回该分区即可；否则，分配器将会基于键值来选择一个分区并返回。
4. 选择完分区后，生产者知道了消息所属的主题和分区，它将这条记录添加到相同主题和分区的批量消息中，另一个线程负责发送这些批量消息到对应的Kafka broker。
5. 当broker接收到消息后，如果成功写入则返回一个包含消息的主题、分区及位移的RecordMetadata对象，否则返回异常。

6. 生产者接收到结果后，对于异常可能会进行重试。

6.2.4 讨论四：消费者设计概要

**1) 消费者与消费组)

假设这么个场景：我们从Kafka中读取消息，并且进行检查，最后产生结果数据。我们可以创建一个消费者实例去做这件事情，但如果生产者写入消息的速度比消费者读取的速度快怎么办呢？这样随着时间增长，消息堆积越来越严重。对于这种场景，我们需要增加多个消费者来进行水平扩展。

Kafka消费者是**消费组**的一部分，当多个消费者形成一个消费组来消费主题时，每个消费者会收到不同分区的消息。假设有一个T1主题，该主题有4个分区；同时我们有一个消费组G1，这个消费组只有一个消费者C1。那么消费者C1将会收到这4个分区的消息，如下所示：

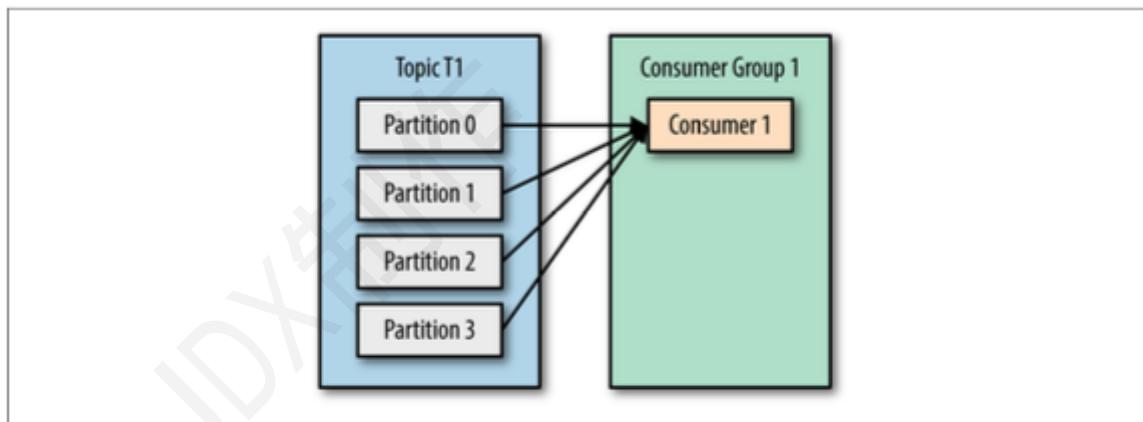


Figure 4-1. One Consumer group with four partitions

如果我们增加新的消费者C2到消费组G1，那么每个消费者将会分别收到两个分区的消息，如下所示：

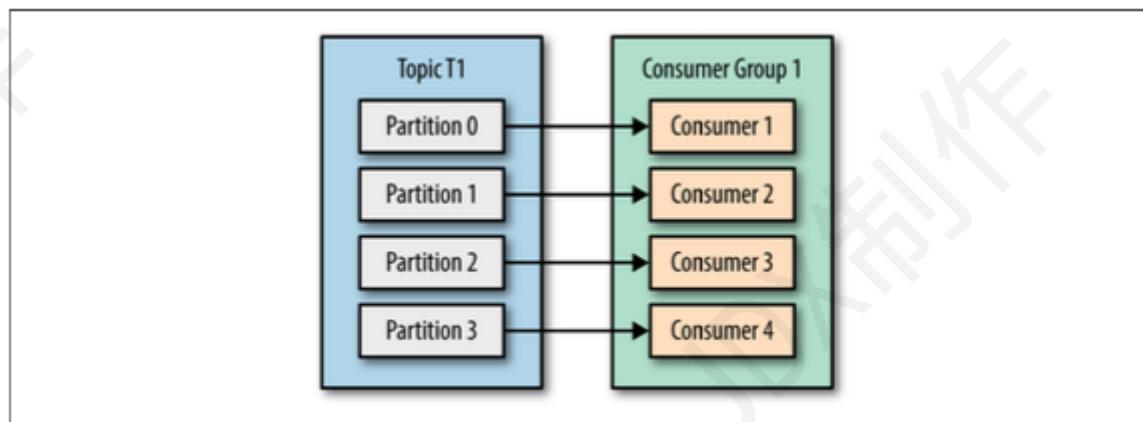


Figure 4-3. Four consumer groups to one partition each

如果增加到4个消费者，那么每个消费者将会分别收到一个分区的消息，如下所示：



但如果我们继续增加消费者到这个消费组，剩余的消费者将会空闲，不会收到任何消息：

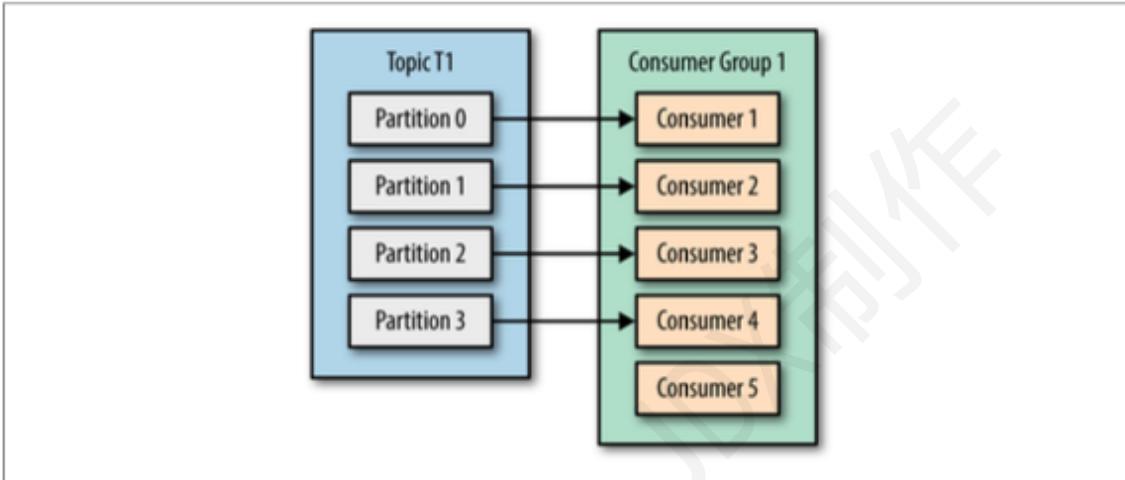


Figure 4-4. More consumer groups than partitions means missed messages

总而言之，我们可以通过增加消费组的消费者来进行水平扩展提升消费能力。这也是为什么建议创建主题时使用比较多的分区数，这样可以在消费负载高的情况下增加消费者来提升性能。另外，消费者的数量不应该比分区数多，因为多出来的消费者是空闲的，没有任何帮助。

Kafka一个很重要的特性就是，只需写入一次消息，可以支持任意多的应用读取这个消息。换句话说，每个应用都可以读到全量的消息。为了使得每个应用都能读到全量消息，应用需要有不同的消费组。对于上面的例子，假如我们新增了一个新的消费组G2，而这个消费组有两个消费者，那么会是这样的：

生产者设计概要

在这个场景中，消费组G1和消费组G2都能收到T1主题的全量消息，在逻辑意义上来说它们属于不同的应用。

最后，总结起来就是：如果应用需要读取全量消息，那么请为该应用设置一个消费组；如果该应用消费能力不足，那么可以考虑在这个消费组里增加消费者。

2) 消费组与分区重平衡

可以看到，当新的消费者加入消费组，它会消费一个或多个分区，而这些分区之前是由其他消费者负责的；另外，当消费者离开消费组（比如重启、宕机等）时，它所消费的分区会分配给其他分区。这种现象称为重平衡（rebalance）。重平衡是 Kafka 一个很重要的性质，这个性质保证了高可用和水平扩展。不过也需要注意到，在重平衡期间，所有消费者都不能消费消息，因此会造成整个消费组短暂的不可用。而且，将分区进行重平衡也会导致原来的消费者状态过期，从而导致消费者需要重新更新状态，这段期间也会降低消费性能。后面我们会讨论如何安全的进行重平衡以及如何尽可能避免。

消费者通过定期发送心跳（heartbeat）到一个作为组协调者（group coordinator）的 broker 来保持在消费组内存活。这个 broker 不是固定的，每个消费组都可能不同。当消费者拉取消息或者提交时，便会发送心跳。

如果消费者超过一定时间没有发送心跳，那么它的会话（session）就会过期，组协调者会认为该消费者已经宕机，然后触发重平衡。可以看到，从消费者宕机到会话过期是有一定时间的，这段时间内该消费者的分区都不能进行消息消费；通常情况下，我们可以进行优雅关闭，这样消费者会发送离开的消息到组协调者，这样组协调者可以立即进行重平衡而不需要等待会话过期。

在 0.10.1 版本，Kafka 对心跳机制进行了修改，将发送心跳与拉取消息进行分离，这样使得发送心跳的频率不受拉取的频率影响。另外更高版本的 Kafka 支持配置一个消费者多长时间不拉取消息但仍然保持存活，这个配置可以避免活锁（livelock）。活锁，是指应用没有故障但是由于某些原因不能进一步消费。

**3) Partition 与消费模型

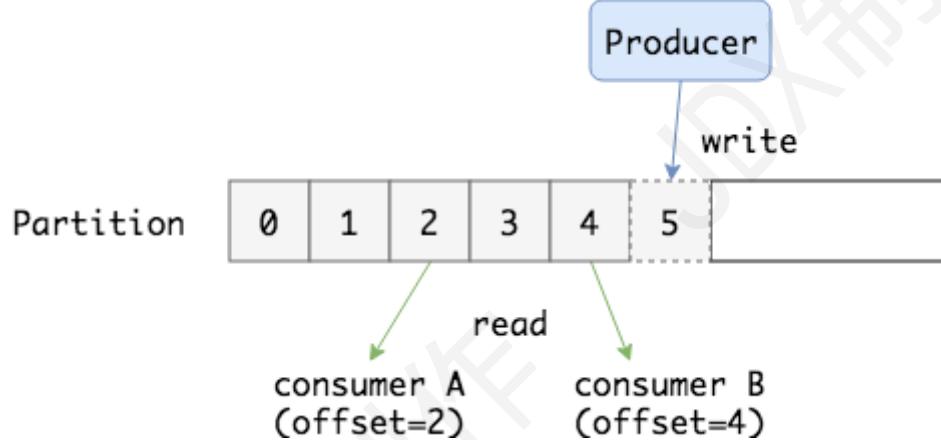
上面提到，Kafka 中一个 topic 中的消息是被打散分配在多个 Partition(分区) 中存储的，Consumer Group 在消费时需要从不同的 Partition 获取消息，那最终如何重建出 Topic 中消息的顺序呢？

答案是：没有办法。Kafka 只会保证在 Partition 内消息是有序的，而不管全局的情况。

下一个问题是：Partition 中的消息可以被（不同的 Consumer Group）多次消费，那 Partition 中被消费的消息是何时删除的？Partition 又是如何知道一个 Consumer Group 当前消费的位置呢？

无论消息是否被消费，除非消息到期 Partition 从不删除消息。例如设置保留时间为 2 天，则消息发布 2 天内任何 Group 都可以消费，2 天后，消息自动被删除。

Partition 会为每个 Consumer Group 保存一个偏移量，记录 Group 消费到的位置。如下图：



4) 为什么 Kafka 是 pull 模型

消费者应该向 Broker 要数据 (pull) 还是 Broker 向消费者推送数据 (push) ? 作为一个消息系统，Kafka 遵循了传统的方式，选择由 Producer 向 broker push 消息并由 Consumer 从 broker pull 消息。一些 logging-centric system，比如 Facebook 的 Scribe 和 Cloudera 的 Flume，采用 push 模式。事实上，push 模式和 pull 模式各有优劣。

push 模式很难适应消费速率不同的消费者，因为消息发送速率是由 broker 决定的。 push 模式的目标是尽可能以最快速度传递消息，但是这样很容易造成 Consumer 来不及处理消息，典型的表现就是拒绝服务以及网络拥塞。而 pull 模式则可以根据 Consumer 的消费能力以适当的速率消费消息。

对于 Kafka 而言，pull 模式更合适。 pull 模式可简化 broker 的设计，Consumer 可自主控制消费消息的速率，同时 Consumer 可以自己控制消费方式——即可批量消费也可逐条消费，同时还能选择不同的提交方式从而实现不同的传输语义。

6.2.5 讨论五：Kafka 如何保证可靠性

当我们讨论可靠性的时侯，我们总会提到“保证”这个词语。可靠性保证是基础，我们基于这些基础之上构建我们的应用。比如关系型数据库的可靠性保证是 ACID，也就是原子性 (Atomicity)、一致性 (Consistency)、隔离性 (Isolation) 和持久性 (Durability)。

Kafka 中的可靠性保证有如下四点：

- 对于一个分区来说，它的消息是有序的。如果一个生产者向一个分区先写入消息A，然后写入消息B，那么消费者会先读取消息A再读取消息B。
- 当消息写入所有in-sync状态的副本后，消息才会认为已提交（committed）。这里的写入有可能只是写入到文件系统的缓存，不一定刷新到磁盘。生产者可以等待不同时机的确认，比如等待分区主副本写入即返回，后者等待所有in-sync状态副本写入才返回。
- 一旦消息已提交，那么只要有一个副本存活，数据不会丢失。
- 消费者只能读取到已提交的消息。

使用这些基础保证，我们构建一个可靠的系统，这时候需要考虑一个问题：究竟我们的应用需要多大程度的可靠性？可靠性不是无偿的，它与系统可用性、吞吐量、延迟和硬件价格息息相关，得此失彼。因此，我们往往需要做权衡，一味的追求可靠性并不实际。

6.3 动手搭一个 Kafka

通过上面的描述，我们已经大致了解到了「Kafka」是何方神圣了，现在我们开始尝试自己动手本地搭一个来实际体验一把。

6.3.1 第一步：下载 Kafka

这里以 Mac OS 为例，在安装了 Homebrew 的情况下执行下列代码：

```
brew install kafka
```

由于 Kafka 依赖了 Zookeeper，所以在下载的时候会自动下载。

6.3.2 第二步：启动服务

我们在启动之前首先需要修改 Kafka 的监听地址和端口为 [localhost:9092]：

```
vi /usr/local/etc/kafka/server.properties
```

然后修改成下图的样子：

```
2. deploy@stage: /var/www/readio_server/current (vim)
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# see kafka.server.KafkaConfig for additional details and defaults

#####
# The id of the broker. This must be set to a unique integer for each broker.
broker.id=0

#####
# The address the socket server listens on. It will get the value returned from
# java.net.InetAddress.getCanonicalHostName() if not configured.
# FORMAT:
#   listeners = listener_name://host_name:port
# EXAMPLE:
#   listeners = PLAINTEXT://your.host.name:9092
listeners=PLAINTEXT://localhost:9092

# Hostname and port the broker will advertise to producers and consumers. If not
set,
:wq
```

依次启动 Zookeeper 和 Kafka：

```
brew services start zookeeper
brew services start kafka
```

然后执行下列语句来创建一个名字为 “test” 的 Topic：

```
kafka-topics --create --zookeeper localhost:2181 --replication-factor 1 --
partitions 1 --topic test
```

我们可以通过下列的命令查看我们的 Topic 列表：

```
kafka-topics --list --zookeeper localhost:2181
```

6.3.3 第三步：发送消息

然后我们新建一个控制台，运行下列命令创建一个消费者关注刚才创建的 Topic：

```
kafka-console-consumer --bootstrap-server localhost:9092 --topic test --from-beginning
```

用控制台往刚才创建的 Topic 中添加消息，并观察刚才创建的消费者窗口：

```
kafka-console-producer --broker-list localhost:9092 --topic test
```

能通过消费者窗口观察到正确的消息：

The image shows two terminal windows side-by-side. The left window, labeled '生产者窗口' (Producer Window), contains the command 'kafka-console-producer --broker-list localhost:9092 --topic test' followed by the input '>send message!'. A red arrow points from this input to the right window. The right window, labeled '消费者窗口' (Consumer Window), contains the command 'kafka-console-consumer --bootstrap-server localhost:9092 --topic test --from-beginning' followed by the output 'send message!'. This visualizes the flow of data from the producer to the consumer.

7. API网关

7.1 背景

7.1.1 什么是API网关

API网关可以看做系统与外界联通的入口，我们可以在网关进行处理一些非业务逻辑的逻辑，比如权限验证，监控，缓存，请求路由等等。

7.1.2 为什么需要API网关

- RPC协议转成HTTP。

由于在内部开发中我们都是以RPC协议(thrift or dubbo)去做开发，暴露给内部服务，当外部服务需要使用这个接口的时候往往需要将RPC协议转换成HTTP协议。

- 请求路由

在我们的系统中由于同一个接口新老两套系统都在使用，我们需要根据请求上下文将请求路由到对应的接口。

- 统一鉴权

对于鉴权操作不涉及到业务逻辑，那么可以在网关层进行处理，不用下层到业务逻辑。

- 统一监控

由于网关是外部服务的入口，所以我们可以在这里监控我们想要的数据，比如入参出参，链路时间。

- 流量控制，熔断降级

对于流量控制，熔断降级非业务逻辑可以统一放到网关层。

有很多业务都会自己去实现一层网关层，用来接入自己的服务，但是对于整个公司来说这还不够。

7.1.3 统一API网关

统一的API网关不仅有API网关的所有的特点，还有下面几个好处：

- 统一技术组件升级

在公司中如果有某个技术组件需要升级，那么是需要和每个业务线沟通，通常几个月都搞不定。举个例子如果对于入口的安全鉴权有重大安全隐患需要升级，如果速度还是这么慢肯定是不行，那么有了统一的网关升级是很快的。

- 统一服务接入

对于某个服务的接入也比较困难，比如公司已经研发出了比较稳定的服务组件，正在公司大力推广，这个周期肯定也特别漫长，由于有了统一网关，那么只需要统一网关统一接入。

- 节约资源

不同业务不同部门如果按照我们上面的做法应该会都自己搞一个网关层，用来做这个事，可以想象如果一个公司有100个这种业务，每个业务配备4台机器，那么就需要400台机器。并且每个业务的开发RD都需要去开发这个网关层，去随时去维护，增加人力。如果有了统一网关层，那么也许只需要50台机器就可以做这100个业务的网关层的事，并且业务RD不需要随时关注开发，上线的步骤。

7.2 统一网关的设计

7.2.1 异步化请求

对于我们自己实现的网关层，由于只有我们自己使用，对于吞吐量的要求并不高所以，我们一般同步请求调用即可。

对于我们统一的网关层，如何用少量的机器接入更多的服务，这就需要我们的异步，用来提高更多的吞吐量。对于异步化一般有下面两种策略：

- Tomcat/Jetty+NIO+servlet3

这种策略使用的比较普遍，京东，有赞，Zuul，都选取的是这个策略，这种策略比较适合HTTP。在Servlet3中可以开启异步。

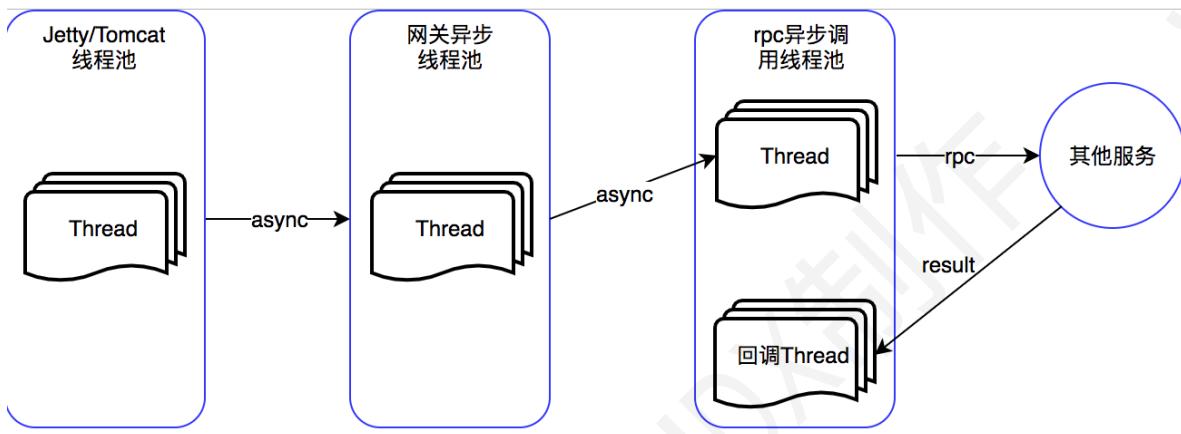
- Netty+NIO

Netty为高并发而生，目前唯品会的网关使用这个策略，在唯品会的技术文章中在相同的情况下Netty是每秒30w+的吞吐量，Tomcat是13w+,可以看出是有一定的差距的，但是Netty需要自己处理HTTP协议，这一块比较麻烦。

对于网关是HTTP请求场景比较多的情况可以采用Servlet，毕竟有更加成熟的处理HTTP协议。如果更加重视吞吐量那么可以采用Netty。

1) 全链路异步

对于来的请求我们已经使用异步了，为了达到全链路异步所以我们需要对去的请求也进行异步处理，对于去的请求我们可以利用我们rpc的异步支持进行异步请求所以基本可以达到下图：

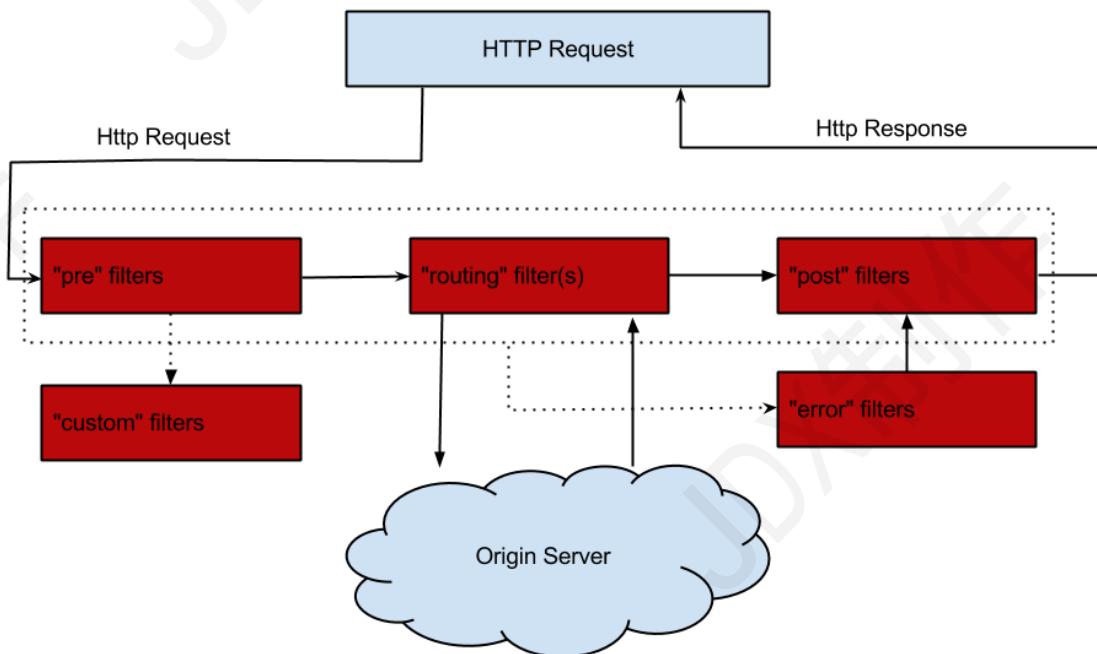


由在web容器中开启servlet异步，然后进入到网关的业务线程池中进行业务处理，然后进行rpc的异步调用并注册需要回调的业务，最后在回调线程池中进行回调处理。

7.2.2 链式处理

在设计模式中有一个模式叫责任链模式，他的作用是避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止。通过这种模式将请求的发送者和请求的处理器解耦了。在我们的各个框架中对此模式都有实现，比如servlet里面的filter，springmvc里面的Interceptor。

在Netflix Zuul中也应用了这种模式，如下图所示：



这种模式在网关的设计中我们可以借鉴到自己的网关设计：

- **preFilters**: 前置过滤器，用来处理一些公共的业务，比如统一鉴权，统一限流，熔断降级，缓存处理等，并且提供业务方扩展。
- **routingFilters**: 用来处理一些泛化调用，主要是做协议的转换，请求的路由工作。
- **postFilters**: 后置过滤器，主要用来做结果的处理，日志打点，记录时间等等。
- **errorFilters**: 错误过滤器，用来处理调用异常的情况。

这种设计在有赞的网关也有应用。

7.2.3 业务隔离

上面在全链路异步的情况下不同业务之间的影响很小，但是如果在提供的自定义Filter中进行了某些同步调用，一旦超时频繁那么就会对其他业务产生影响。所以我们需要采用隔离之术，降低业务之间的互相影响。

1) 信号量隔离

信号量隔离只是限制了总的并发数，服务还是主线程进行同步调用。这个隔离如果远程调用超时依然会影响主线程，从而会影响其他业务。因此，如果只是想限制某个服务的总并发调用量或者调用的服务不涉及远程调用的话，可以使用轻量级的信号量来实现。有赞的网关由于没有自定义filter所以选取的是信号量隔离。

2) 线程池隔离

最简单的就是不同业务之间通过不同的线程池进行隔离，就算业务接口出现了问题由于线程池已经进行了隔离那么也不会影响其他业务。在京东的网关实现之中就是采用的线程池隔离，比较重要的业务比如商品或者订单都是单独的通过线程池去处理。但是由于是统一网关平台，如果业务线众多，大家都觉得自己的业务比较重要需要单独的线程池隔离，如果使用的是Java语言开发的话那么，在Java中线程是比较重的资源比较受限，如果需要隔离的线程池过多不是很适用。如果使用一些其他语言比如Golang进行开发网关的话，线程是比较轻的资源，所以比较适合使用线程池隔离。

3) 集群隔离

如果有某些业务就需要使用隔离但是统一网关又没有线程池隔离那么应该怎么办呢？那么可以使用集群隔离，如果你的某些业务真的很重要那么可以为这一系列业务单独申请一个集群或者多个集群，通过机器之间进行隔离。

7.2.4 请求限流

流量控制可以采用很多开源的实现，比如阿里最近开源的Sentinel和比较成熟的Hystrix。

一般限流分为集群限流和单机限流：

- 利用统一存储保存当前流量的情况，一般可以采用Redis，这个一般会有一些性能损耗。
- 单机限流：限流每台机器我们可以直接利用Guava的令牌桶去做，由于没有远程调用性能消耗较小。

7.2.5 熔断降级

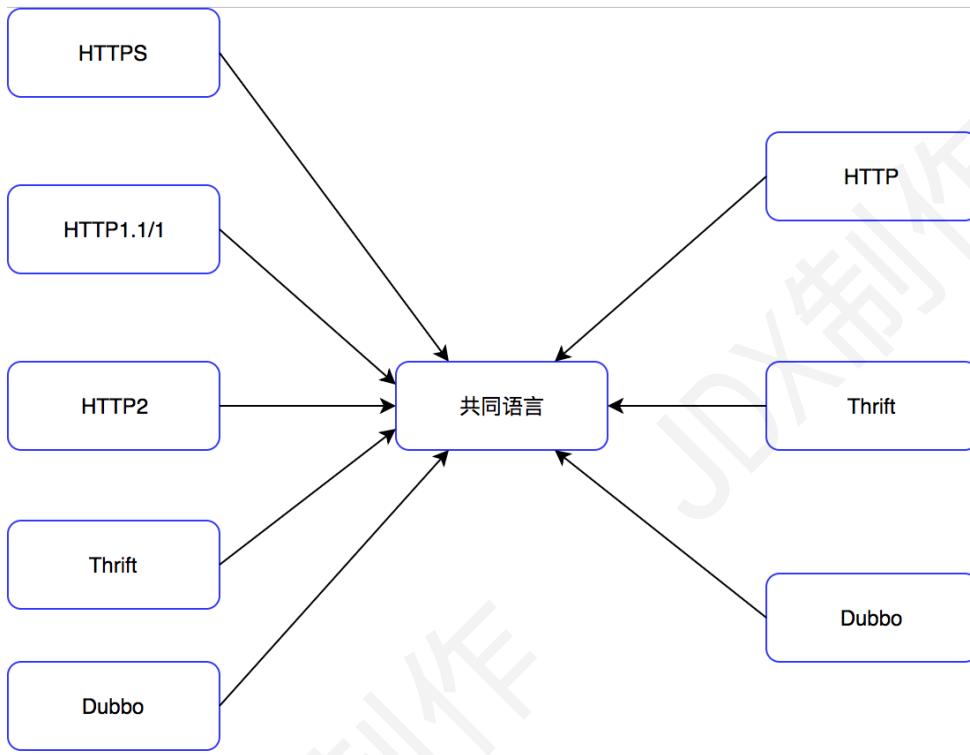
这一块也可以参照开源的实现Sentinel和Hystrix，这里不是重点就不多提了。

7.2.6 泛化调用

泛化调用指的是一些通信协议的转换，比如将HTTP转换成Thrift。在一些开源的网关中比如Zuul是没有实现的，因为各个公司的内部服务通信协议都不同。比如在唯品会中支持HTTP1,HTTP2,以及二进制的协议，然后转化成内部的协议，淘宝的支持HTTPS,HTTP1,HTTP2这些协议都可以转换成，HTTP,HSF,Dubbo等协议。

1) 泛化调用

如何去实现泛化调用呢？由于协议很难自动转换，那么其实每个协议对应的接口需要提供一种映射。简单来说就是把两个协议都能转换成共同语言，从而互相转换。



一般来说共同语言有三种方式指定:

- json: json数据格式比较简单,解析速度快,较轻量级。在Dubbo的生态中有一个HTTP转Dubbo的项目是用JsonRpc做的,将HTTP转化成JsonRpc再转化成Dubbo。

比如可以将一个 www.baidu.com/id = 1 GET 可以映射为json:

代码块

```
{
  "method": "getBaidu"
  "param" : {
    "id" : 1
  }
}
```

- xml:xml数据比较重,解析比较困难,这里不过多讨论。
- 自定义描述语言:一般来说这个成本比较高需要自己定义语言来进行描述并进行解析,但是其扩展性,自定义个性化性都是最高。例:spring自定义了一套自己的SPEL表达式语言

对于泛化调用如果要自己设计的话JSON基本可以满足,如果对于个性化的需要特别多的话倒是可以自己定义一套语言。

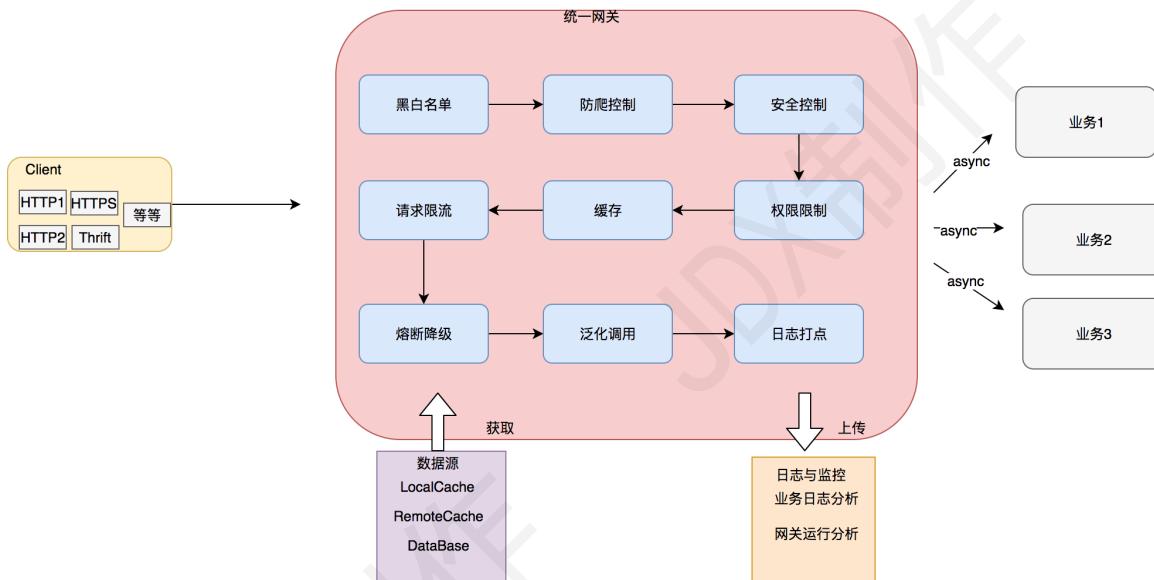
7.2.7 管理平台

上面介绍的都是如何实现一个网关的技术关键。这里需要介绍网关的一个业务关键。有了网关之后,需要一个管理平台如何去对我们上面所描述的技术关键进行配置,包括但不限于下面这些配置:

- 限流
- 熔断
- 缓存
- 日志
- 自定义filter
- 泛化调用

7.3 总结

最后一个合理的标准网关应该按照如下去实现:



8. 分布式ID

8.1 数据库自增ID

第一种方案仍然还是基于数据库的自增ID，需要单独使用一个数据库实例，在这个实例中新建一个单独的表：

表结构如下：

```
CREATE DATABASE `SEQID`;

CREATE TABLE SEQID.SEQUENCE_ID (
    id bigint(20) unsigned NOT NULL auto_increment,
    stub char(10) NOT NULL default '',
    PRIMARY KEY (id),
    UNIQUE KEY stub (stub)
) ENGINE=MyISAM;
```

可以使用下面的语句生成并获取到一个自增ID

```
begin;
replace into SEQUENCE_ID (stub) VALUES ('anyword');
select last_insert_id();
commit;
```

stub字段在这里并没有什么特殊的意义，只是为了方便的去插入数据，只有能插入数据才能产生自增id。而对于插入我们用的是replace，replace会先看是否存在stub指定值一样的数据，如果存在则先delete再insert，如果不存在则直接insert。

这种生成分布式ID的机制，需要一个单独的Mysql实例，虽然可行，但是基于性能与可靠性来考虑的话都不够，**业务系统每次需要一个ID时，都需要请求数据库获取，性能低，并且如果此数据库实例下线了，那么将影响所有的业务系统。**

为了解决数据库可靠性问题，我们可以使用第二种分布式ID生成方案。

8.2 数据库多主模式

如果我们两个数据库组成一个**主从模式集群**，正常情况下可以解决数据库可靠性问题，但是如果主库挂掉后，数据没有及时同步到从库，这个时候会出现ID重复的现象。我们可以使用**双主模式集群**，也就是两个Mysql实例都能单独的生产自增ID，这样能够提高效率，但是如果不过其他改造的话，这两个Mysql实例很可能会生成同样的ID。需要单独给每个Mysql实例配置不同的起始值和自增步长。

第一台Mysql实例配置：

```
set @@auto_increment_offset = 1;      -- 起始值  
set @@auto_increment_increment = 2;   -- 步长
```

第二台Mysql实例配置：

```
set @@auto_increment_offset = 2;      -- 起始值  
set @@auto_increment_increment = 2;   -- 步长
```

经过上面的配置后，这两个Mysql实例生成的id序列如下：mysql1,起始值为1,步长为2, ID生成的序列为：1,3,5,7,9,... mysql2,起始值为2,步长为2, ID生成的序列为：2,4,6,8,10,...

对于这种生成分布式ID的方案，需要单独新增一个生成分布式ID应用，比如DistributIdService，该应用提供一个接口供业务应用获取ID，业务应用需要一个ID时，通过rpc的方式请求DistributIdService，DistributIdService随机去上面的两个Mysql实例中去获取ID。

实行这种方案后，就算其中某一台Mysql实例下线了，也不会影响DistributIdService，DistributIdService仍然可以利用另外一台Mysql来生成ID。

但是这种方案的扩展性不太好，如果两台Mysql实例不够用，需要新增Mysql实例来提高性能时，这时就会比较麻烦。

现在如果要新增一个实例mysql3，要怎么操作呢？第一，mysql1、mysql2的步长肯定都要修改为3，而且只能是人工去修改，这是需要时间的。第二，因为mysql1和mysql2是不停在自增的，对于mysql3的起始值我们可能要定得大一点，以给充分的时间去修改mysql1, mysql2的步长。第三，在修改步长的时候很可能会出现重复ID，要解决这个问题，可能需要停机才行。

为了解决上面的问题，以及能够进一步提高DistributIdService的性能，如果使用第三种生成分布式ID机制。

8.3 号段模式

我们可以使用号段的方式来获取自增ID，号段可以理解成批量获取，比如DistributIdService从数据库获取ID时，如果能批量获取多个ID并缓存在本地的话，那样将大大提供业务应用获取ID的效率。

比如DistributIdService每次从数据库获取ID时，就获取一个号段，比如(1,1000]，这个范围表示了1000个ID，业务应用在请求DistributIdService提供ID时，DistributIdService只需要在本地从1开始自增并返回即可，而不需要每次都请求数库，一直到本地自增到1000时，也就是当前号段已经被用完时，才去数据库重新获取下一号段。

所以，我们需要对数据库表进行改动，如下：

```
CREATE TABLE id_generator (  
    id int(10) NOT NULL,  
    current_max_id bigint(20) NOT NULL COMMENT '当前最大id',  
    increment_step int(10) NOT NULL COMMENT '号段的长度',  
    PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

这个数据库表用来记录自增步长以及当前自增ID的最大值（也就是当前已经被申请的号段的最后一个值），因为自增逻辑被移到DistributIdService中去了，所以数据库不需要这部分逻辑了。

这种方案不再强依赖数据库，就算数据库不可用，那么DistributIdService也能继续支撑一段时间。但是如果DistributIdService重启，会丢失一段ID，导致ID空洞。

为了提高DistributIdService的高可用，需要做一个集群，业务在请求DistributIdService集群获取ID时，会随机的选择某一个DistributIdService节点进行获取，对每一个DistributIdService节点来说，数据库连接的是同一个数据库，那么可能会产生多个DistributIdService节点同时请求数据库获取号段，那么这个时候需要利用乐观锁来进行控制，比如在数据库表中增加一个version字段，在获取号段时使用如下SQL：

```
update id_generator set current_max_id=#{newMaxId}, version=version+1 where  
version = #{version}
```

因为newMaxId是DistributIdService中根据oldMaxId+步长算出来的，只要上面的update更新成功了就表示号段获取成功了。

为了提供数据库层的高可用，需要对数据库使用多主模式进行部署，对于每个数据库来说要保证生成的号段不重复，这就需要利用最开始的思路，再在刚刚的数据库表中增加起始值和步长，比如如果现在是两台Mysql，那么 mysql1将生成号段 (1,1001]，自增的时候序列为1, 3, 4, 5, 7... mysql2将生成号段 (2,1002]，自增的时候序列为2, 4, 6, 8, 10...

在TinyId中还增加了一步来提高效率，在上面的实现中，ID自增的逻辑是在DistributIdService中实现的，而实际上可以把自增的逻辑转移到业务应用本地，这样对于业务应用来说只需要获取号段，每次自增时不再需要请求调用DistributIdService了。

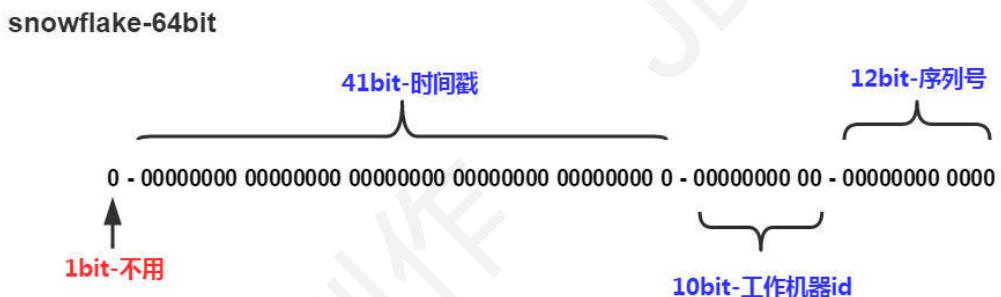
8.4 雪花算法

上面的三种方法总的来说是基于自增思想的，而接下来就介绍比较著名的雪花算法-snowflake。

我们可以换个角度来对分布式ID进行思考，只要能让负责生成分布式ID的每台机器在每毫秒内生成不一样的ID就行了。

snowflake是twitter开源的分布式ID生成算法，是一种算法，所以它和上面的三种生成分布式ID机制不太一样，它不依赖数据库。

核心思想是：分布式ID固定是一个long型的数字，一个long型占8个字节，也就是64个bit，原始snowflake算法中对于bit的分配如下图：



<https://blog.csdn.net/u011919808>

- 第一个bit位是标识部分，在java中由于long的最高位是符号位，正数是0，负数是1，一般生成的ID为正数，所以固定为0。
- 时间戳部分占41bit，这个是毫秒级的时间，一般实现上不会存储当前的时间戳，而是时间戳的差值（当前时间-固定的开始时间），这样可以使产生的ID从更小值开始；41位的时间戳可以使用69年， $(1L << 41) / (1000L * 60 * 60 * 24 * 365) = 69$ 年

- 工作机器id占10bit，这里比较灵活，比如，可以使用前5位作为数据中心机房标识，后5位作为单机房机器标识，可以部署1024个节点。
- 序列号部分占12bit，支持同一毫秒内同一个节点可以生成4096个ID

根据这个算法的逻辑，只需要将这个算法用Java语言实现出来，封装为一个工具方法，那么各个业务应用可以直接使用该工具方法来获取分布式ID，只需保证每个业务应用有自己的工作机器id即可，而不需要单独去搭建一个获取分布式ID的应用。

在大厂里，其实并没有直接使用snowflake，而是进行了改造，因为snowflake算法中最难实践的就是工作机器id，原始的snowflake算法需要人工去为每台机器去指定一个机器id，并配置在某个地方从而让snowflake从此处获取机器id。

但是在大厂里，机器是很多的，人力成本太大且容易出错，所以大厂对snowflake进行了改造。

8.5 百度 (uid-generator)

uid-generator使用的就是snowflake，只是在生产机器id，也叫做workId时有所不同。

uid-generator中的workId是由uid-generator自动生成的，并且考虑到了应用部署在docker上的情况，在uid-generator中用户可以自己去定义workId的生成策略，默认提供的策略是：应用启动时由数据库分配。说的简单一点就是：应用在启动时会往数据库表(uid-generator)需要新增一个WORKER_NODE表中去插入一条数据，数据插入成功后返回的该数据对应的自增唯一id就是该机器的workId，而数据由host, port组成。

对于uid-generator中的workId，占用了22个bit位，时间占用了28个bit位，序列化占用了13个bit位，需要注意的是，和原始的snowflake不太一样，时间的单位是秒，而不是毫秒，workId也不一样，同一个应用每重启一次就会消费一个workId。

8.6 美团 (Leaf)

美团的Leaf也是一个分布式ID生成框架。它非常全面，即支持号段模式，也支持snowflake模式。号段模式这里就不介绍了，和上面的分析类似。

Leaf中的snowflake模式和原始snowflake算法的不同点，也主要在workId的生成，Leaf中workId是基于ZooKeeper的顺序Id来生成的，每个应用在使用Leaf-snowflake时，在启动时都会都在Zookeeper中生成一个顺序Id，相当于一台机器对应一个顺序节点，也就是一个workId。

8.7 总结

总得来说，上面两种都是自动生成workId，以让系统更加稳定以及减少人工输入。

8.8 Redis

这里额外再介绍一下使用Redis来生成分布式ID，其实和利用Mysql自增ID类似，可以利用Redis中的incr命令来实现原子性的自增与返回，比如：

```
127.0.0.1:6379> set seq_id 1      // 初始化自增ID为1
OK
127.0.0.1:6379> incr seq_id       // 增加1，并返回
(integer) 2
127.0.0.1:6379> incr seq_id       // 增加1，并返回
(integer) 3
```

使用redis的效率是非常高的，但是要考虑持久化的问题。Redis支持RDB和AOF两种持久化的方式。

RDB持久化相当于定时打一个快照进行持久化，如果打完快照后，连续自增了几次，还没来得及做下一次快照持久化，这个时候Redis挂掉了，重启Redis后会出现ID重复。

AOF持久化相当于对每条写命令进行持久化，如果Redis挂掉了，不会出现ID重复的现象，但是会由于incr命令过得，导致重启恢复数据时间过长。

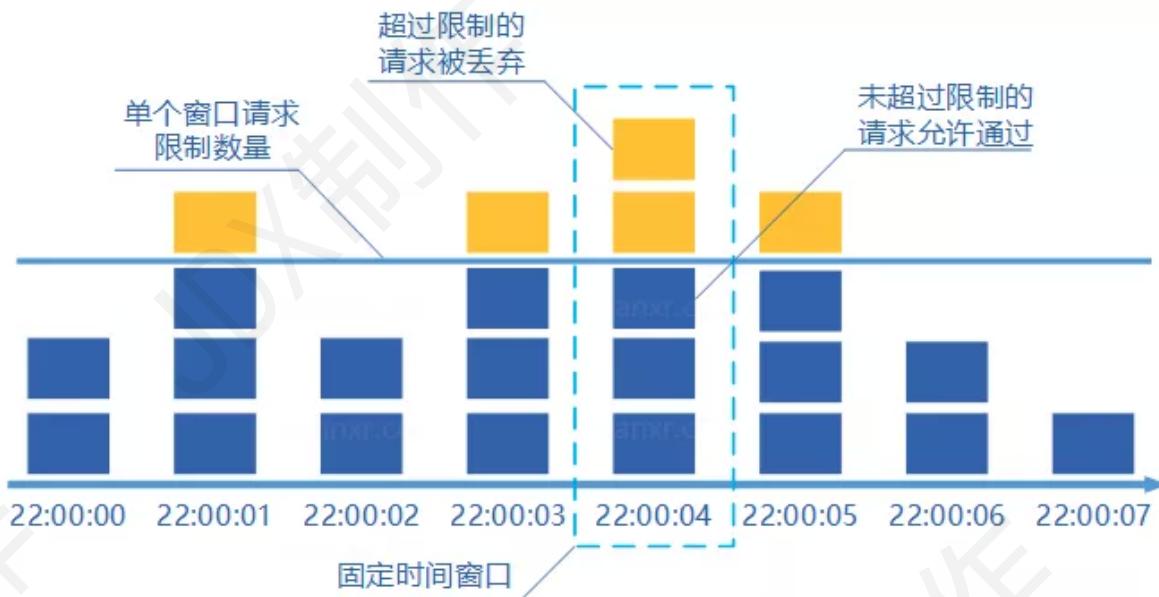
9. 限流的算法有哪些？

简单介绍4种非常好理解并且容易实现的限流算法！

9.1 固定窗口计数器算法

规定我们单位时间处理的请求数量。比如我们规定我们的一个接口一分钟只能访问10次的话。使用固定窗口计数器算法的话可以这样实现：给定一个变量counter来记录处理的请求数量，当1分钟之内处理一个请求之后counter+1，1分钟之内的如果counter=100的话，后续的请求就会被全部拒绝。等到1分钟结束后，将counter回归成0，重新开始计数（ps：只要过了一个周期就将counter回归成0）。

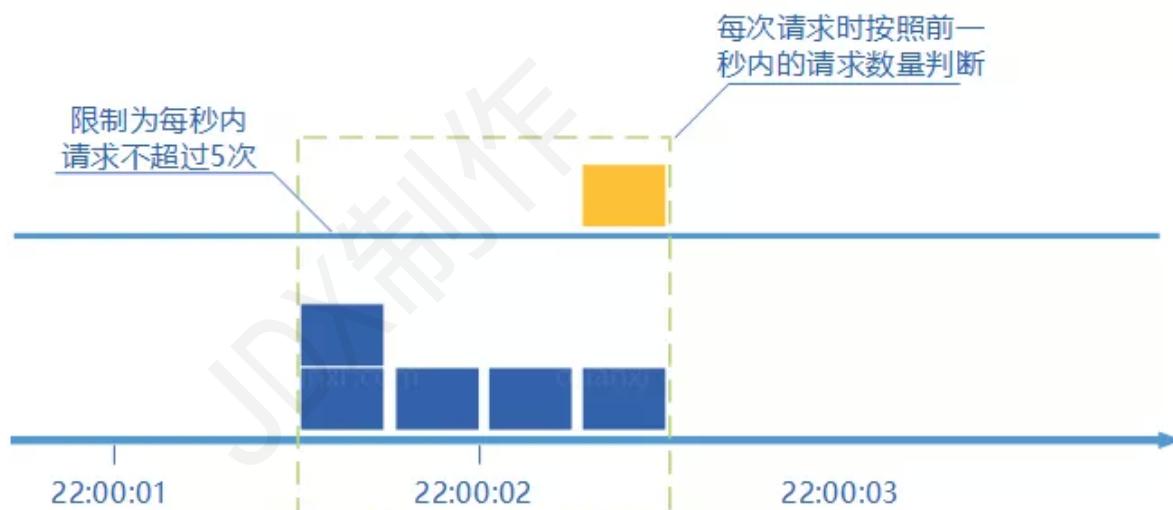
这种限流算法无法保证限流速率，因而无法保证突然激增的流量。比如我们限制一个接口一分钟只能访问10次的话，前半分钟一个请求没有接收，后半分钟接收了10个请求。



9.2 滑动窗口计数器算法

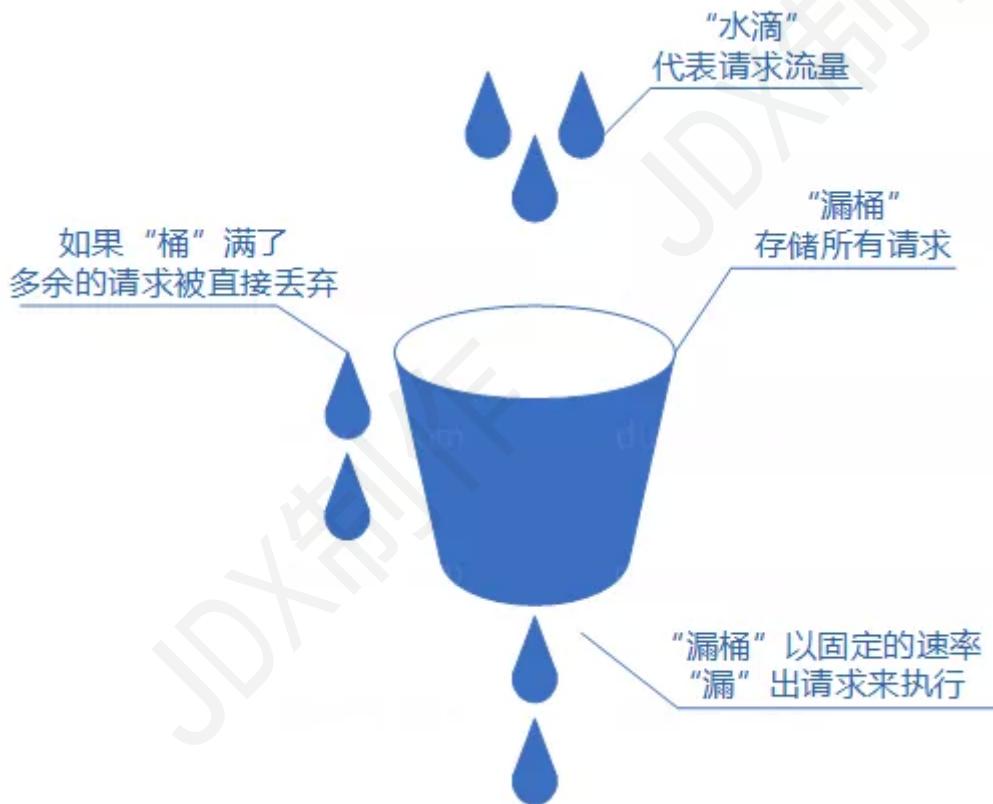
算的上是固定窗口计数器算法的升级版。滑动窗口计数器算法相比于固定窗口计数器算法的优化在于：它把时间以一定比例分片。例如我们的借口限流每分钟处理60个请求，我们可以把1分钟分为60个窗口。每隔1秒移动一次，每个窗口一秒只能处理不大于 $60(\text{请求数})/60(\text{窗口数})$ 的请求，如果当前窗口的请求计数总和超过了限制的数量的话就不再处理其他请求。

很显然：当滑动窗口的格子划分的越多，滑动窗口的滚动就越平滑，限流的统计就会越精确。



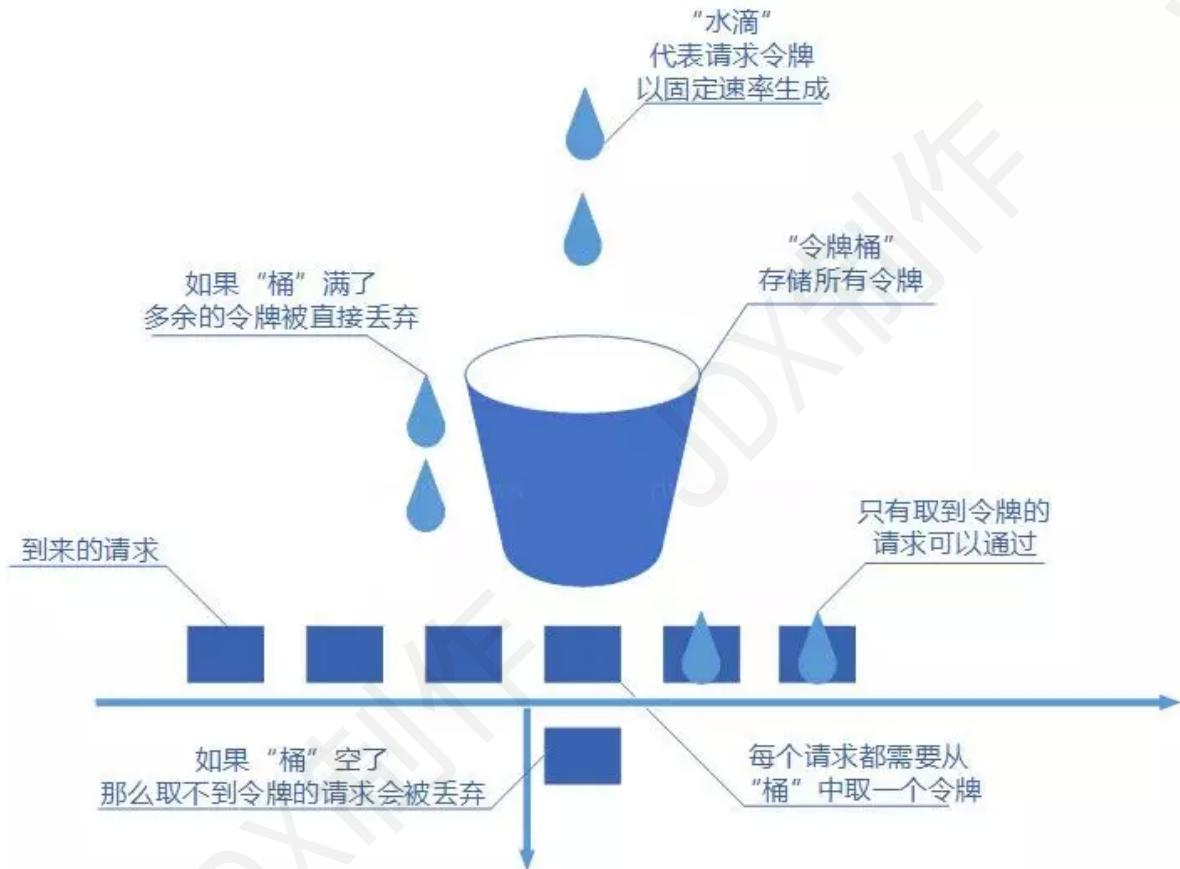
9.3 漏桶算法

我们可以把发请求的动作比作成注水到桶中，我们处理请求的过程可以比喻为漏桶漏水。我们往桶中以任意速率流入水，以一定速率流出水。当水超过桶容量时则丢弃，因为桶容量是不变的，保证了整体的速率。如果想要实现这个算法的话也很简单，准备一个队列用来保存请求，然后我们定期从队列中拿请求来执行就好了。



9.4 令牌桶算法

令牌桶算法也比较简单。和漏桶算法一样，我们的主角还是桶（这限流算法和桶过不去啊）。不过现在桶里装的是令牌了，请求在被处理之前需要拿到一个令牌，请求处理完毕之后将这个令牌丢弃（删除）。我们根据限流大小，按照一定的速率往桶里添加令牌。



10. Zookeeper

10.1 前言

相信大家对 ZooKeeper 应该不算陌生。但是你真的了解 ZooKeeper 是个什么东西吗？如果别人/面试官让你给他讲讲 ZooKeeper 是个什么东西，你能回答到什么地步呢？

获取更多资源可添加关注微信公众号：JAVA架构进阶之路；或添加微信：jiang10086a 免费获取面试真题，视频教程，笔记等。

我本人曾经使用过 ZooKeeper 作为 Dubbo 的注册中心，另外在搭建 solr 集群的时候，我使用到了 ZooKeeper 作为 solr 集群的管理工具。前几天，总结项目经验的时候，我突然问自己 ZooKeeper 到底是个什么东西？想了半天，脑海中只是简单的能浮现出几句话：“①Zookeeper 可以被用作注册中心。②Zookeeper 是 Hadoop 生态系统的一员；③构建 Zookeeper 集群的时候，使用的服务器最好是奇数台。”可见，我对于 Zookeeper 的理解仅仅是停留在了表面。

所以，通过本文，希望带大家稍微详细的了解一下 ZooKeeper。如果没有学过 ZooKeeper，那么本文将会是你进入 ZooKeeper 大门的垫脚砖。如果你已经接触过 ZooKeeper，那么本文将带你回顾一下 ZooKeeper 的一些基础概念。

10.2 什么是 ZooKeeper 一些概念，并不涉及 ZooKeeper 的使用以及 ZooKeeper 集群的搭建。
10.2.1 ZooKeeper 的由来
网上有介绍 ZooKeeper 的使用以及搭建 ZooKeeper 集群的文章，大家有需要可以自行查阅。

下面这段内容摘自《从 Paxos 到 Zookeeper》第四章第一节的某段内容，推荐大家阅读以下：

Zookeeper 最早起源于雅虎研究院的一个研究小组。在当时，研究人员发现，在雅虎内部很多大型系统基本都需要依赖一个类似的系统来进行分布式协调，但是这些系统往往都存在分布式单点问题。所以，雅虎的开发人员就试图开发一个通用的无单点问题的分布式协调框架，以便让开发人员将精力集中在处理业务逻辑上。

关于“ZooKeeper”这个项目的名字，其实也有一段趣闻。在立项初期，考虑到之前内部很多项目都是使用动物的名字来命名的（例如著名的Pig项目），雅虎的工程师希望给这个项目也取一个动物的名字。时任研究院的首席科学家RaghuRamakrishnan开玩笑地说：“在这样下去，我们这儿就变成动物园了！”此话一出，大家纷纷表示就叫动物园管理员吧——因为各个以动物命名的分布式组件放在一起，雅虎的整个分布式系统看上去就像一个大型的动物园了，而Zookeeper正好要用来进行分布式环境的协调——于是，Zookeeper的名字也就由此诞生了。

10.2.2 ZooKeeper 概览

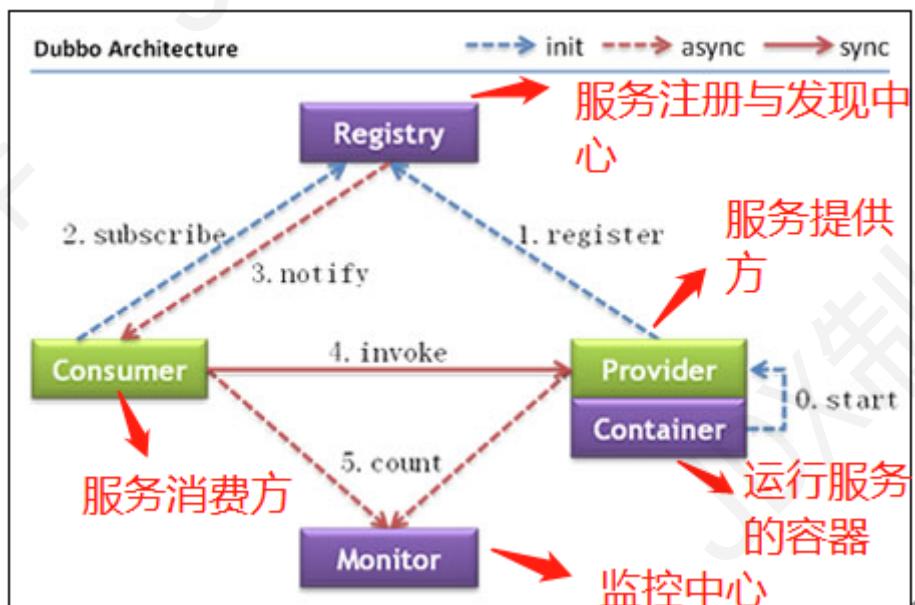
ZooKeeper 是一个开源的分布式协调服务，ZooKeeper 框架最初是在“Yahoo!”上构建的，用于以简单而稳健的方式访问他们的应用程序。后来，Apache ZooKeeper 成为 Hadoop，HBase 和其他分布式框架使用的有组织服务的标准。例如，Apache HBase 使用 ZooKeeper 跟踪分布式数据的状态。

ZooKeeper 的设计目标是将那些复杂且容易出错的分布式一致性服务封装起来，构成一个高效可靠的原语集，并以一系列简单易用的接口提供给用户使用。

原语：操作系统或计算机网络用语范畴。是由若干条指令组成的，用于完成一定功能的一个过程。具有不可分割性·即原语的执行必须是连续的，在执行过程中不允许被中断。

ZooKeeper 是一个典型的分布式数据一致性解决方案，分布式应用程序可以基于 ZooKeeper 实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能。

Zookeeper 一个最常用的使用场景就是用于担任服务生产者和服务消费者的注册中心(提供发布订阅服务)。服务生产者将自己提供的服务注册到Zookeeper中心，服务的消费者在进行服务调用的时候先到Zookeeper中查找服务，获取到服务生产者的详细信息之后，再去调用服务生产者的内容与数据。如下图所示，在 Dubbo 架构中 Zookeeper 就担任了注册中心这一角色。



10.2.3 结合个人使用情况的讲一下 ZooKeeper

在我自己做过的项目中，主要使用到了 ZooKeeper 作为 Dubbo 的注册中心(Dubbo 官方推荐使用 ZooKeeper 注册中心)。另外在搭建 solr 集群的时候，我使用 ZooKeeper 作为 solr 集群的管理工具。这时，ZooKeeper 主要提供下面几个功能：1、集群管理：容错、负载均衡。2、配置文件的集中管理 3、集群的入口。

我个人觉得在使用 ZooKeeper 的时候，最好是使用 集群版的 ZooKeeper 而不是单机版的。官网给出的架构图就描述的是一个集群版的 ZooKeeper 。通常 3 台服务器就可以构成一个 ZooKeeper 集群了。

为什么最好使用奇数台服务器构成 ZooKeeper 集群？

所谓的zookeeper容错是指，当宕掉几个zookeeper服务器之后，剩下的个数必须大于宕掉的个数的话整个zookeeper才依然可用。假如我们的集群中有n台zookeeper服务器，那么也就是剩下的服务数必须大于 $n/2$ 。先说一下结论， $2n$ 和 $2n-1$ 的容忍度是一样的，都是 $n-1$ ，大家可以先自己仔细想一想，这应该是一个很简单的数学问题了。

比如假如我们有3台，那么最大允许宕掉1台zookeeper服务器，如果我们有4台的时候也同样只允许宕掉1台。

假如我们有5台，那么最大允许宕掉2台zookeeper服务器，如果我们有6台的时候也同样只允许宕掉2台。

综上，何必增加那一个不必要的zookeeper呢？

10.3 关于 ZooKeeper 的一些重要概念

10.3.1 重要概念总结

- ZooKeeper 本身就是一个分布式程序（只要半数以上节点存活，ZooKeeper 就能正常服务）。
- 为了保证高可用，最好是以集群形态来部署 ZooKeeper，这样只要集群中大部分机器是可用的（能够容忍一定的机器故障），那么 ZooKeeper 本身仍然是可用的。
- ZooKeeper 将数据保存在内存中，这也就保证了 高吞吐量和低延迟（但是内存限制了能够存储的容量不太大，此限制也是保持znode中存储的数据量较小的进一步原因）。
- ZooKeeper 是高性能的。在“读”多于“写”的应用程序中尤其地高性能，因为“写”会导致所有的服务器间同步状态。（“读”多于“写”是协调服务的典型场景。）
- ZooKeeper有临时节点的概念。当创建临时节点的客户端会话一直保持活动，瞬时节点就一直存在。而当会话终结时，瞬时节点被删除。持久节点是指一旦这个ZNode被创建了，除非主动进行ZNode的移除操作，否则这个ZNode将一直保存在Zookeeper上。
- ZooKeeper 底层其实只提供了两个功能：①管理（存储、读取）用户程序提交的数据；②为用户程序提供数据节点监听服务。

下面关于会话（Session）、Znode、版本、Watcher、ACL概念的总结都在《从Paxos到Zookeeper》第四章第一节以及第七章第八节有提到，感兴趣的可以看看！

10.3.2 会话（Session）

Session 指的是 ZooKeeper 服务器与客户端会话。在 ZooKeeper 中，一个客户端连接是指客户端和服务器之间的一个 TCP 长连接。客户端启动的时候，首先会与服务器建立一个 TCP 连接，从第一次连接建立开始，客户端会话的生命周期也开始了。通过这个连接，客户端能够通过心跳检测与服务器保持有效的会话，也能够向Zookeeper服务器发送请求并接受响应，同时还能够通过该连接接收来自服务器的Watch事件通知。Session的 sessionTimeout 值用来设置一个客户端会话的超时时间。当由于服务器压力过大、网络故障或是客户端主动断开连接等各种原因导致客户端连接断开时，只要在 sessionTimeout 规定的时间内能够重新连接上集群中任意一台服务器，那么之前创建的会话仍然有效。

在为客户端创建会话之前，服务端首先会为每个客户端都分配一个sessionID。由于 sessionID 是 Zookeeper 会话的一个重要标识，许多与会话相关的运行机制都是基于这个 sessionID 的，因此，无论是哪台服务器为客户端分配的 sessionID，都务必保证全局唯一。

10.3.3 Znode

在谈到分布式的时候，我们通常说的“节点”是指组成集群的每一台机器。然而，在Zookeeper中，“节点”分为两类，第一类同样是指构成集群的机器，我们称之为机器节点；第二类则是指数据模型中的数据单元，我们称之为数据节点——ZNode。

Zookeeper将所有数据存储在内存中，数据模型是一棵树（Znode Tree），由斜杠（/）的进行分割的路径，就是一个Znode，例如/foo/path1。每个上都会保存自己的数据内容，同时还会保存一系列属性信息。

在Zookeeper中，node可以分为持久节点和临时节点两类。所谓持久节点是指一旦这个ZNode被创建了，除非主动进行ZNode的移除操作，否则这个ZNode将一直保存在Zookeeper上。而临时节点就不一样了，它的生命周期和客户端会话绑定，一旦客户端会话失效，那么这个客户端创建的所有临时节点都会被移除。另外，ZooKeeper还允许用户为每个节点添加一个特殊的属性：**SEQUENTIAL**。一旦节点被标记上这个属性，那么在这个节点被创建的时候，Zookeeper会自动在其节点名后面追加上一个整型数字，这个整型数字是一个由父节点维护的自增数字。

10.3.4 版本

在前面我们已经提到，Zookeeper 的每个 ZNode 上都会存储数据，对于每个ZNode，Zookeeper 都会为其维护一个叫作 **Stat** 的数据结构，Stat 中记录了这个 ZNode 的三个数据版本，分别是 **version**（当前ZNode的版本）、**cversion**（当前ZNode子节点的版本）和 **aversion**（当前ZNode的 ACL版本）。

10.3.5 Watcher

Watcher（事件监听器），是Zookeeper中的一个很重要的特性。Zookeeper允许用户在指定节点上注册一些Watcher，并且在一些特定事件触发的时候，ZooKeeper服务端会将事件通知到感兴趣的客户端上去，该机制是Zookeeper实现分布式协调服务的重要特性。

10.3.6 ACL

Zookeeper采用ACL（AccessControlLists）策略来进行权限控制，类似于 UNIX 文件系统的权限控制。Zookeeper 定义了如下5种权限。

- **CREATE**: 创建子节点的权限。
- **READ**: 获取节点数据和子节点列表的权限。
- **WRITE**: 更新节点数据的权限。
- **DELETE**: 删除子节点的权限。
- **ADMIN**: 设置节点 ACL 的权限。

其中尤其需要注意的是，CREATE和DELETE这两种权限都是针对子节点的权限控制。

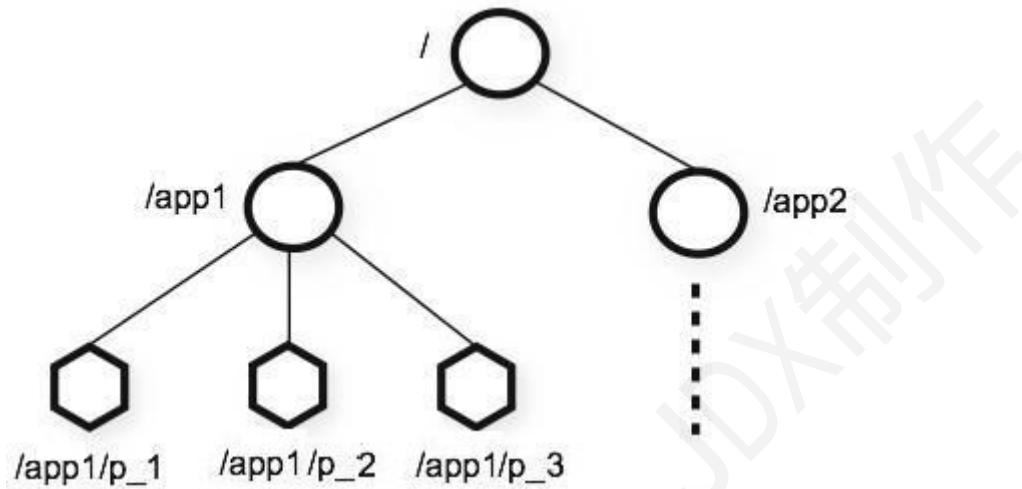
10.4 ZooKeeper 特点

- **顺序一致性**: 从同一客户端发起的事务请求，最终将会严格地按照顺序被应用到 ZooKeeper 中去。
- **原子性**: 所有事务请求的处理结果在整个集群中所有机器上的应用情况是一致的，也就是说，要么整个集群中所有的机器都成功应用了某一个事务，要么都没有应用。
- **单一系统映像**: 无论客户端连到哪一个 ZooKeeper 服务器上，其看到的服务端数据模型都是一致的。
- **可靠性**: 一旦一次更改请求被应用，更改的结果就会被持久化，直到被下一次更改覆盖。

10.5 ZooKeeper 设计目标

10.5.1 简单的数据模型

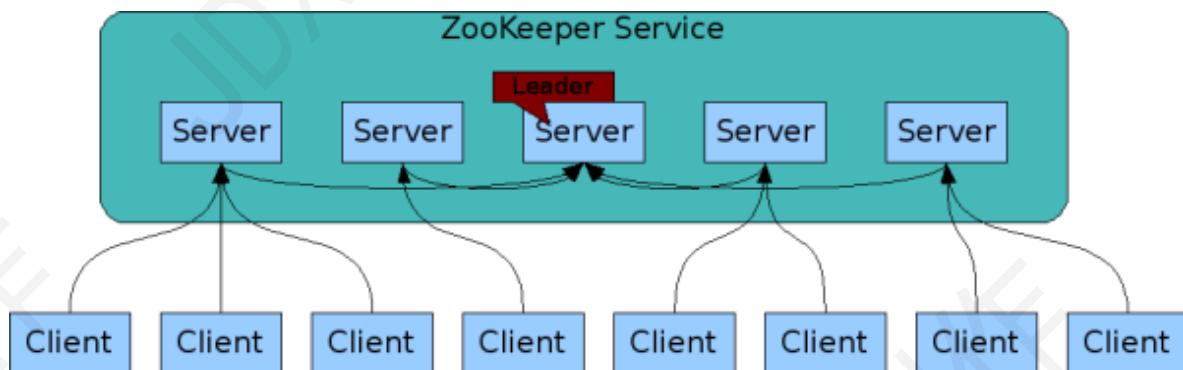
ZooKeeper 允许分布式进程通过共享的层次结构命名空间进行相互协调，这与标准文件系统类似。名称空间由 ZooKeeper 中的数据寄存器组成 - 称为znode，这些类似于文件和目录。与为存储设计的典型文件系统不同，ZooKeeper数据保存在内存中，这意味着ZooKeeper可以实现高吞吐量和低延迟。



10.5.2 可构建集群

为了保证高可用，最好是以集群形态来部署 ZooKeeper，这样只要集群中大部分机器是可用的（能够容忍一定的机器故障），那么 zookeeper 本身仍然是可用的。客户端在使用 ZooKeeper 时，需要知道集群机器列表，通过与集群中的某一台机器建立 TCP 连接来使用服务，客户端使用这个 TCP 链接来发送请求、获取结果、获取监听事件以及发送心跳包。如果这个连接异常断开了，客户端可以连接到另外的机器上。

ZooKeeper 官方提供的架构图：



上图中每一个 Server 代表一个安装 Zookeeper 服务的服务器。组成 ZooKeeper 服务的服务器都会在内存中维护当前的服务器状态，并且每台服务器之间都互相保持着通信。集群间通过 Zab 协议 (Zookeeper Atomic Broadcast) 来保持数据的一致性。

10.5.3 顺序访问

对于来自客户端的每个更新请求，ZooKeeper 都会分配一个全局唯一的递增编号，这个编号反映了所有事务操作的先后顺序，应用程序可以使用 ZooKeeper 这个特性来实现更高层次的同步原语。这个编号也叫做时间戳——**zxid** (Zookeeper Transaction Id)

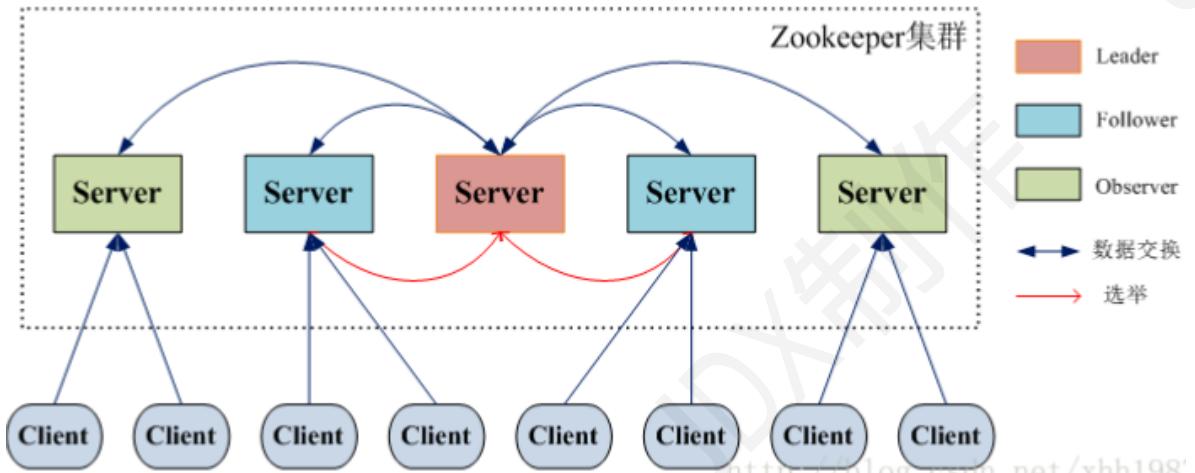
10.5.4 高性能

ZooKeeper 是高性能的。在“读”多于“写”的应用程序中尤其地高性能，因为“写”会导致所有的服务器间同步状态。（“读”多于“写”是协调服务的典型场景。）

10.6 ZooKeeper 集群角色介绍

最典型集群模式：Master/Slave 模式（主备模式）。在这种模式中，通常 Master 服务器作为主服务器提供写服务，其他的 Slave 服务器从服务器通过异步复制的方式获取 Master 服务器最新的数据提供读服务。

但是，在 ZooKeeper 中没有选择传统的 Master/Slave 概念，而是引入了 Leader、Follower 和 Observer 三种角色。如下图所示



ZooKeeper 集群中的所有机器通过一个 Leader 选举过程来选定一台称为“Leader”的机器，Leader 既可以为客户端提供写服务又能提供读服务。除了 Leader 外，Follower 和 Observer 都只能提供读服务。Follower 和 Observer 唯一的区别在于 Observer 机器不参与 Leader 的选举过程，也不参与写操作的“过半写成功”策略，因此 Observer 机器可以在不影响写性能的情况下提升集群的读性能。

角色		主要工作描述
领导者		1. 事务请求的唯一调度和处理者，保证集群事务处理的顺序性； 2. 集群内部各服务器的调度者
学习者 (Learner)	跟随者 (Follower)	1. 处理客户端非事务请求，转发事务请求给Leader服务器 2. 参与事务请求Proposal的投票 3. 参与Leader选举的投票
	观察者 (Observer)	Follower 和 Observer 唯一的区别在于 Observer 机器不参与 Leader 的选举过程，也不参与写操作的“过半写成功”策略，因此 Observer 机器可以在不影响写性能的情况下提升集群的读性能。
客户端 (Client)		请求发起方

当 Leader 服务器出现网络中断、崩溃退出与重启等异常情况时，ZAB 协议就会进入恢复模式并选举产生新的Leader服务器。这个过程大致是这样的：

1. Leader election (选举阶段)：节点在一开始都处于选举阶段，只要有一个节点得到超半数节点的票数，它就可以当选准 leader。
2. Discovery (发现阶段)：在这个阶段，followers 跟准 leader 进行通信，同步 followers 最近接收的事务提议。
3. Synchronization (同步阶段)：同步阶段主要是利用 leader 前一阶段获得的最新提议历史，同步集群中所有的副本。同步完成之后准 leader 才会成为真正的 leader。
4. Broadcast (广播阶段)：到了这个阶段，Zookeeper 集群才能正式对外提供事务服务，并且 leader 可以进行消息广播。同时如果有新的节点加入，还需要对新节点进行同步。

10.7 ZooKeeper &ZAB 协议&Paxos算法

10.7.1 ZAB 协议&Paxos算法

Paxos 算法应该可以说是 ZooKeeper 的灵魂了。但是，ZooKeeper 并没有完全采用 Paxos 算法，而是使用 ZAB 协议作为其保证数据一致性的核心算法。另外，在 ZooKeeper 的官方文档中也指出，ZAB 协议并不像 Paxos 算法那样，是一种通用的分布式一致性算法，它是一种特别为 Zookeeper 设计的崩溃可恢复的原子消息广播算法。

10.7.2 ZAB 协议介绍

ZAB (ZooKeeper Atomic Broadcast 原子广播) 协议是为分布式协调服务 ZooKeeper 专门设计的一种支持崩溃恢复的原子广播协议。在 ZooKeeper 中，主要依赖 ZAB 协议来实现分布式数据一致性，基于该协议，ZooKeeper 实现了一种主备模式的系统架构来保持集群中各个副本之间的数据一致性。

10.7.3 ZAB 协议两种基本的模式：崩溃恢复和消息广播

ZAB协议包括两种基本的模式，分别是 **崩溃恢复** 和 **消息广播**。当整个服务框架在启动过程中，或是当 Leader 服务器出现网络中断、崩溃退出与重启等异常情况时，ZAB 协议就会进入恢复模式并选举产生新的Leader服务器。当选举产生了新的 Leader 服务器，同时集群中已经有过半的机器与该Leader服务器完成了状态同步之后，ZAB协议就会退出恢复模式。其中，**所谓状态同步是指数据同步，用来保证集群中存在过半的机器能够和Leader服务器的数据状态保持一致**。

当集群中已经有过半的Follower服务器完成了和Leader服务器的状态同步，那么整个服务框架就可以进入消息广播模式了。当一台同样遵守ZAB协议的服务器启动后加入到集群中时，如果此时集群中已经存在一个Leader服务器在负责进行消息广播，那么新加人的服务器就会自觉地进入数据恢复模式：找到 Leader 所在的服务器，并与其进行数据同步，然后一起参与到消息广播流程中去。正如上文介绍中所说的，ZooKeeper设计成只允许唯一的一个Leader服务器来进行事务请求的处理。Leader服务器在接收到客户端的事务请求后，会生成对应的事务提案并发起一轮广播协议；而如果集群中的其他机器接收到客户端的事务请求，那么这些非Leader服务器会首先将这个事务请求转发给Leader服务器。

关于 **ZAB 协议&Paxos算法** 需要讲和理解的东西太多了，说实话，笔者到现在不太清楚这俩兄弟的具体原理和实现过程。

10.8 总结

通过阅读本文，想必大家已从 ①**ZooKeeper的由来**。-> ②**ZooKeeper 到底是什么**。-> ③**ZooKeeper 的一些重要概念**（会话（Session）、Znode、版本、Watcher、ACL）-> ④**ZooKeeper 的特点**。-> ⑤**ZooKeeper 的设计目标**。-> ⑥**ZooKeeper 集群角色介绍**（Leader、Follower 和 Observer 三种角色）-> ⑦**ZooKeeper &ZAB 协议&Paxos算法**。这七点了解了 ZooKeeper 。

(五). 大型网站架构

1. 如何设计一个高可用系统？要考虑哪些地方？

一篇短小的文章，面试经常遇到的这个问题。本文主要包括下面这些内容：

1. 高可用的定义
2. 哪些情况可能会导致系统不可用？
3. 有些提高系统可用性的方法？只是简单的提一嘴，更具体内容在后续的文章中介绍，就拿限流来说，你需要搞懂：何为限流？如何限流？为什么要限流？如何做呢？说一下原理？。

1.1 什么是高可用？可用性的判断标准是啥？

高可用描述的是一个系统在大部分时间都是可用的，可以为我们提供服务的。高可用代表系统即使在发生硬件故障或者系统升级的时候，服务仍然是可用的。

一般情况下，我们使用多少个 9 来评判一个系统的可用性，比如 99.9999% 就是代表该系统在所有的运行时间中只有 0.0001% 的时间是不可用的，这样的系统就是非常非常高可用的了！当然，也会有系统如果可用性不太好的话，可能连 9 都上不了。

除此之外，系统的可用性还可以用某功能的失败次数与总的请求次数之比来衡量，比如对网站请求 1000 次，其中有 10 次请求失败，那么可用性就是 99%。

1.2 哪些情况会导致系统不可用？

1. 黑客攻击；
2. 硬件故障，比如服务器坏掉。
3. 并发量/用户请求量激增导致整个服务宕掉或者部分服务不可用。
4. 代码中的坏味道导致内存泄漏或者其他问题导致程序挂掉。
5. 网站架构某个重要的角色比如 Nginx 或者数据库突然不可用。
6. 自然灾害或者人为破坏。
7.

1.3 有哪些提高系统可用性的方法？

1.3.1 注重代码质量，测试严格把关

我觉得这个是最最最重要的，代码质量有问题比如比较常见的内存泄漏、循环依赖都是对系统可用性极大的损害。大家都喜欢谈限流、降级、熔断，但是我觉得从代码质量这个源头把关是首先要做好的一件很重要的事情。如何提高代码质量？比较实际可用的就是 CodeReview，不要在乎每天多花的那 1 个小时左右的时间，作用可大着呢！

另外，安利这个对提高代码质量有实际效果的宝贝：

1. sonarqube：保证你写出更安全更干净的代码！（ps: 目前所在的项目基本都会用到这个插件）。
2. Alibaba 开源的 Java 诊断工具 Arthas 也是很不错的选项。
3. IDEA 自带的代码分析等工具进行代码扫描也是非常非常棒的。

1.3.2 使用集群，减少单点故障

先拿常用的 Redis 举个例子！我们如何保证我们的 Redis 缓存高可用呢？答案就是使用集群，避免单点故障。当我们使用一个 Redis 实例作为缓存的时候，这个 Redis 实例挂了之后，整个缓存服务可能就挂了。使用了集群之后，即使一台 Redis 实例，不到一秒就会有另外一台 Redis 实例顶上。

1.3.3 限流

流量控制（flow control），其原理是监控应用流量的 QPS 或并发线程数等指标，当达到指定的阈值时对流量进行控制，以避免被瞬时的流量高峰冲垮，从而保障应用的高可用性。——来自 alibaba-Sentinel 的 wiki。

1.3.4 超时和重试机制设置

一旦用户请求超过某个时间的得不到响应，就抛出异常。这个是非常重要的，很多线上系统故障都是因为没有进行超时设置或者超时设置的方式不对导致的。我们在读取第三方服务的时候，尤其适合设置超时和重试机制。一般我们使用一些 RPC 框架的时候，这些框架都自带的超时重试的配置。如果不进行超时设置可能会导致请求响应速度慢，甚至导致请求堆积进而让系统无法处理请求。重试的次数一般设为 3 次，再多次的重试没有好处，反而会加重服务器压力（部分场景使用失败重试机制会不太适合）。

1.3.5 熔断机制

超时和重试机制设置之外，熔断机制也是很重要的。熔断机制说的是系统自动收集所依赖服务的资源使用情况和性能指标，当所依赖的服务恶化或者调用失败次数达到某个阈值的时候就迅速失败，让当前系统立即切换依赖其他备用服务。比较常用的是流量控制和熔断降级框架是 Netflix 的 Hystrix 和 alibaba 的 Sentinel。

1.3.6 异步调用

异步调用的话我们不需要关心最后的结果，这样我们就可以在用户请求完成之后就立即返回结果，具体处理我们可以后续再做，秒杀场景用这个还是蛮多的。但是，使用异步之后我们可能需要 **适当修改业务流程** 进行配合，比如 **用户在提交订单之后，不能立即返回用户订单提交成功，需要在消息队列的订单消费者进程真正处理完该订单之后，甚至出库后，再通过电子邮件或短信通知用户订单成功**。除了可以在程

序中实现异步之外，我们常常还使用消息队列，消息队列可以通过异步处理提高系统性能（削峰、减少响应所需时间）并且可以降低系统耦合性。

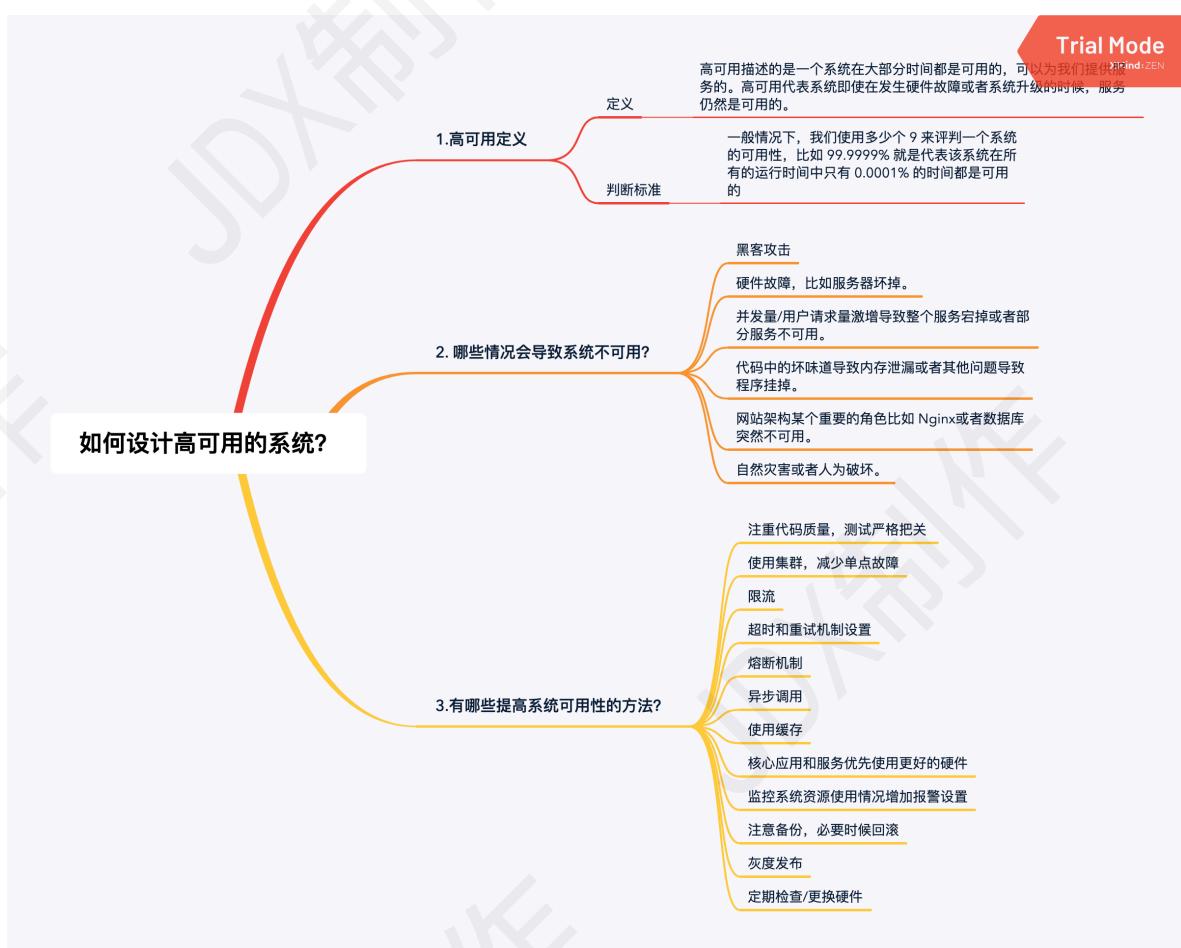
1.3.7 使用缓存

如果我们的系统属于并发量比较高的话，如果我们单纯使用数据库的话，当大量请求直接落到数据库可能数据库就会直接挂掉。使用缓存存储热点数据，因为缓存存储在内存中，所以速度相当地快！

1.3.8 其他

1. 核心应用和服务优先使用更好的硬件
2. 监控系统资源使用情况增加报警设置。
3. 注意备份，必要时候回滚。
4. 灰度发布：将服务器集群分成若干部分，每天只发布一部分机器，观察运行稳定没有故障，第二天继续发布一部分机器，持续几天才把整个集群全部发布完毕，期间如果发现问题，只需要回滚已发布的一部分服务器即可
5. 定期检查/更换硬件：如果不是购买的云服务的话，定期还是需要对硬件进行一波检查的，对于一些需要更换或者升级的硬件，要及时更换或者升级。

1.4 总结



(六). 微服务

1. Spring Cloud

1.1 什么是Spring cloud

构建分布式系统不需要复杂和容易出错。Spring Cloud 为最常见的分布式系统模式提供了一种简单且易于接受的编程模型，帮助开发人员构建有弹性的、可靠的、协调的应用程序。Spring Cloud 构建于 Spring Boot 之上，使得开发者很容易入手并快速应用于生产中。

获取更多资源可添加关注微信公众号：[JAVA架构进阶之路](#)；或添加微信：jiang10086a 免费获取面试真题，视频教程，笔记等。

官方果然官方，介绍都这么有板有眼的。

我所理解的 `Spring Cloud` 就是微服务系统架构的一站式解决方案，在平时我们构建微服务的过程中需要做如 **服务发现注册**、**配置中心**、**消息总线**、**负载均衡**、**断路器**、**数据监控** 等操作，而 Spring Cloud 为我们提供了一套简易的编程模型，使我们能在 Spring Boot 的基础上轻松地实现微服务项目的构建。

1.2 Spring Cloud 的版本

当然这个只是个题外话。

`Spring Cloud` 的版本号并不是我们通常常见的数字版本号，而是一些很奇怪的单词。这些单词均为英国伦敦地铁站的站名。同时根据字母表的顺序来对应版本时间顺序，比如：最早的 `Release` 版本 `Angel`，第二个 `Release` 版本 `Brixton`（英国地名），然后是 `Camden`、`Dalston`、`Edgware`、`Finchley`、`Greenwich`、`Hoxton`。

1.3 Spring Cloud 的服务发现框架——Eureka

`Eureka` 是基于 `REST`（代表性状态转移）的服务，主要在 `AWS` 云中用于定位服务，以实现负载均衡和中间层服务器的故障转移。我们称此服务为 `Eureka` 服务器。`Eureka` 还带有一个基于 `Java` 的客户端组件 `Eureka Client`，它使与服务的交互变得更加容易。客户端还具有一个内置的负载平衡器，可以执行基本的循环负载平衡。在 `Netflix`，更复杂的负载均衡器将 `Eureka` 包装起来，以基于流量，资源使用，错误条件等多种因素提供加权负载均衡，以提供出色的弹性。

总的来说，`Eureka` 就是一个服务发现框架。何为服务，何又为发现呢？

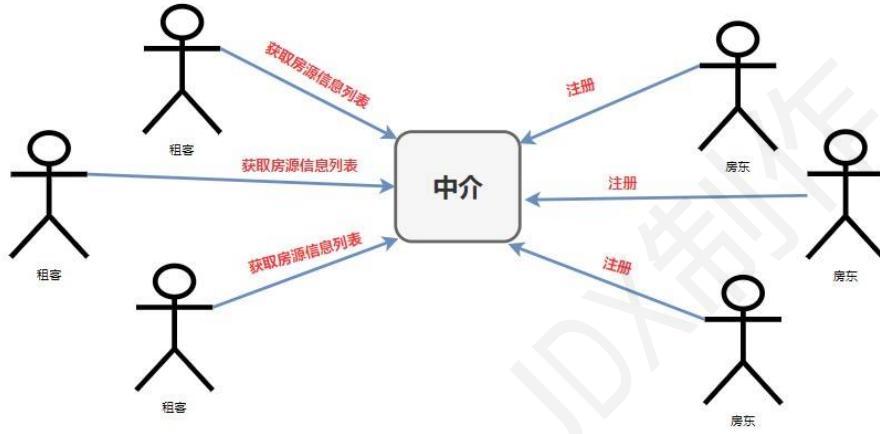
举一个生活中的例子，就比如我们平时租房子找中介的事情。

在没有中介的时候我们需要一个一个去寻找是否有房屋要出租的房东，这显然会非常的费力，一你找凭一个人的能力是找不到很多房源供你选择，再者你也懒得这么找下去(找了这么久，没有合适的只能将就)。这里的我们就相当于微服务中的 `Consumer`，而那些房东就相当于微服务中的 `Provider`。消费者 `Consumer` 需要调用提供者 `Provider` 提供的一些服务，就像我们现在需要租他们的房子一样。

但是如果只是租客和房东之间进行寻找的话，他们的效率是很低的，房东找不到租客赚不到钱，租客找不到房东住不了房。所以，后来房东肯定就想到了广播自己的房源信息(比如在街边贴贴小广告)，这样对于房东来说已经完成他的任务(将房源公布出去)，但是有两个问题就出现了。第一、其他不是租客的都能收到这种租房消息，这在现实世界没什么，但是在计算机的世界中就会出现 **资源消耗** 的问题了。第二、租客这样还是很难找到你，试想一下我需要租房，我还需要东一个西一个地去找街边小广告，麻烦不麻烦？

那怎么办呢？我们当然不会那么傻乎乎的，第一时间就是去找 **中介** 呀，它为我们提供了统一房源的地方，我们消费者只需要跑到它那里去找就行了。而对于房东来说，他们也只需要把房源在中介那里发布就行了。

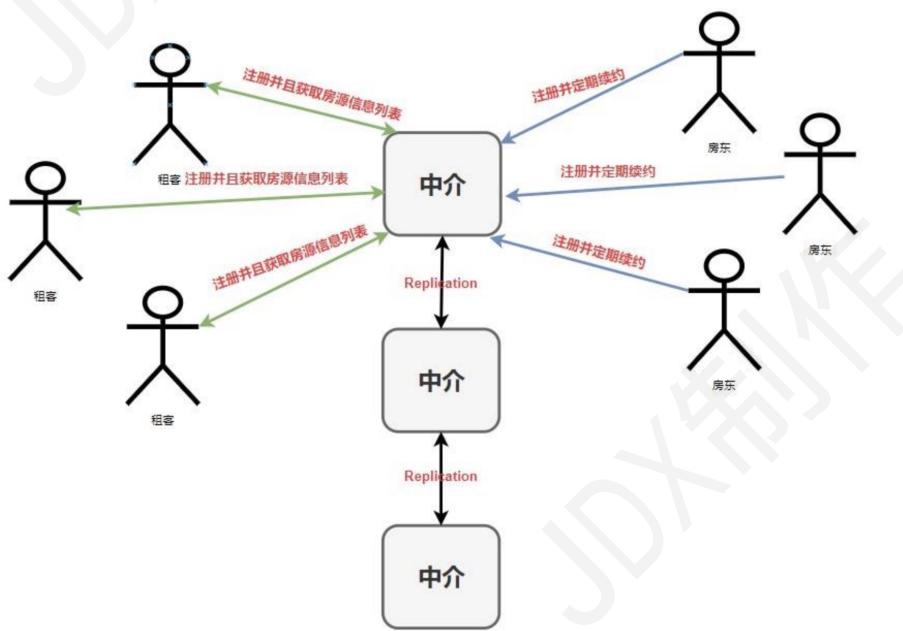
那么现在，我们的模式就是这样的了。



但是，这个时候还会出现一些问题。

1. 房东注册之后如果不想卖房子了怎么办？我们是不是需要让房东 **定期续约**？如果房东不进行续约是不是要将他们从中介那里的注册列表中 **移除**。
2. 租客是不是也要进行 **注册** 呢？不然合同乙方怎么来呢？
3. 中介可不可以做 **连锁店** 呢？如果这一个店因为某些不可抗力因素而无法使用，那么我们是否可以换一个连锁店呢？

针对上面的问题我们来重新构建一下上面的模式图



好了，举完这个栗子我们就可以来看关于 **Eureka** 的一些基础概念了，你会发现这东西理解起来怎么这么简单。

服务发现：其实就是一个“中介”，整个过程中有三个角色：**服务提供者(出租房子的)**、**服务消费者(租客)**、**服务中介(房屋中介)**。

服务提供者：就是提供一些自己能够执行的一些服务给外界。

服务消费者：就是需要使用一些服务的“用户”。

服务中介：其实就是服务提供者和服务消费者之间的“桥梁”，服务提供者可以把自己注册到服务中介那里，而服务消费者如需要消费一些服务(使用一些功能)就可以在服务中介中寻找注册在服务中介的服务提供者。

服务注册 Register：

官方解释：当 Eureka 客户端向 Eureka Server 注册时，它提供自身的元数据，比如IP地址、端口，运行状况指示符URL，主页等。

结合中介理解：房东(提供者 Eureka Client Provider)在中介(服务器 Eureka Server)那里登记房屋的信息，比如面积，价格，地段等等(元数据 metaData)。

服务续约 Renew:

官方解释：Eureka 客户会每隔30秒(默认情况下)发送一次心跳来续约。通过续约来告知 Eureka Server 该 Eureka 客户仍然存在，没有出现问题。正常情况下，如果 Eureka Server 在90秒没有收到 Eureka 客户的续约，它会将实例从其注册表中删除。

结合中介理解：房东(提供者 Eureka Client Provider)定期告诉中介(服务器 Eureka Server)我的房子还租(续约)，中介(服务器 Eureka Server)收到之后继续保留房屋的信息。

获取注册列表信息 Fetch Registries:

官方解释：Eureka 客户端从服务器获取注册表信息，并将其缓存在本地。客户端会使用该信息查找其他服务，从而进行远程调用。该注册列表信息定期(每30秒钟)更新一次。每次返回注册列表信息可能与 Eureka 客户端的缓存信息不同，Eureka 客户端自动处理。如果由于某种原因导致注册列表信息不能及时匹配，Eureka 客户端则会重新获取整个注册表信息。Eureka 服务器缓存注册列表信息，整个注册表以及每个应用程序的信息进行了压缩，压缩内容和没有压缩的内容完全相同。Eureka 客户端和 Eureka 服务器可以使用JSON / XML格式进行通讯。在默认的情况下 Eureka 客户端使用压缩 JSON 格式来获取注册列表的信息。

结合中介理解：租客(消费者 Eureka Client Consumer)去中介(服务器 Eureka Server)那里获取所有的房屋信息列表(客户端列表 Eureka Client List)，而且租客为了获取最新的信息会定期向中介(服务器 Eureka Server)那里获取并更新本地列表。

服务下线 Cancel:

官方解释：Eureka客户端在程序关闭时向Eureka服务器发送取消请求。发送请求后，该客户端实例信息将从服务器的实例注册表中删除。该下线请求不会自动完成，它需要调用以下内容：

```
DiscoveryManager.getInstance().shutdownComponent();
```

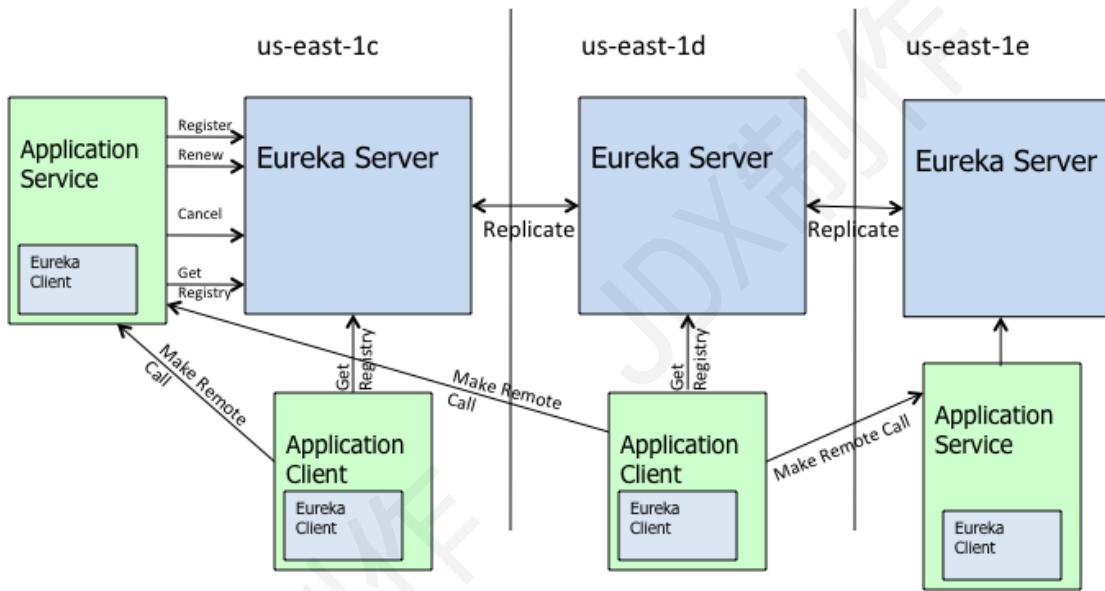
结合中介理解：房东(提供者 Eureka Client Provider)告诉中介(服务器 Eureka Server)我的房子不租了，中介之后就将注册的房屋信息从列表中剔除。

服务剔除 Eviction:

官方解释：在默认的情况下，当Eureka客户端连续90秒(3个续约周期)没有向Eureka服务器发送服务续约，即心跳，Eureka服务器会将该服务实例从服务注册列表删除，即服务剔除。

结合中介理解：房东(提供者 Eureka Client Provider)会定期联系中介(服务器 Eureka Server)告诉他我的房子还租(续约)，如果中介(服务器 Eureka Server)长时间没收到提供者的信息，那么中介会将他的房屋信息给下架(服务剔除)。

下面就是 Netflix 官方给出的 Eureka 架构图，你会发现和我们前面画的中介图别无二致。



当然，可以充当服务发现的组件有很多：`Zookeeper`，`Consul`，`Eureka` 等。

1.4 负载均衡之 Ribbon

1.4.1 什么是 `RestTemplate`？

不是讲 `Ribbon` 么？怎么扯到了 `RestTemplate` 了？你先别急，听我慢慢道来。

我不听我不听我不听。

我就说一句！`RestTemplate` 是 `Spring` 提供的一个访问Http服务的客户端类，怎么说呢？就是微服务之间的调用是使用的 `RestTemplate`。比如这个时候我们 消费者B 需要调用 提供者A 所提供的服务我们就需要这么写。如我下面的伪代码。

```

@.Autowired
private RestTemplate restTemplate;
// 这里是提供者A的ip地址，但是如果使用了 Eureka 那么就应该是提供者A的名称
private static final String SERVICE_PROVIDER_A = "http://localhost:8081";

@PostMapping("/judge")
public boolean judge(@RequestBody Request request) {
    String url = SERVICE_PROVIDER_A + "/service1";
    return restTemplate.postForObject(url, request, Boolean.class);
}

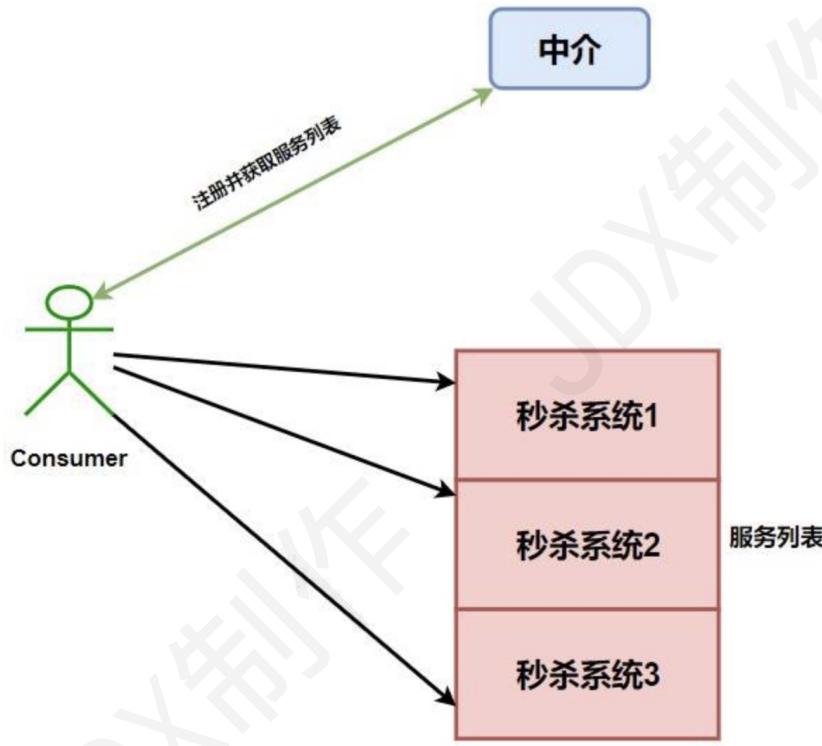
```

如果你对源码感兴趣的话，你会发现上面我们所讲的 `Eureka` 框架中的 **注册、续约** 等，底层都是使用的 `RestTemplate`。

1.4.2 为什么需要 Ribbon？

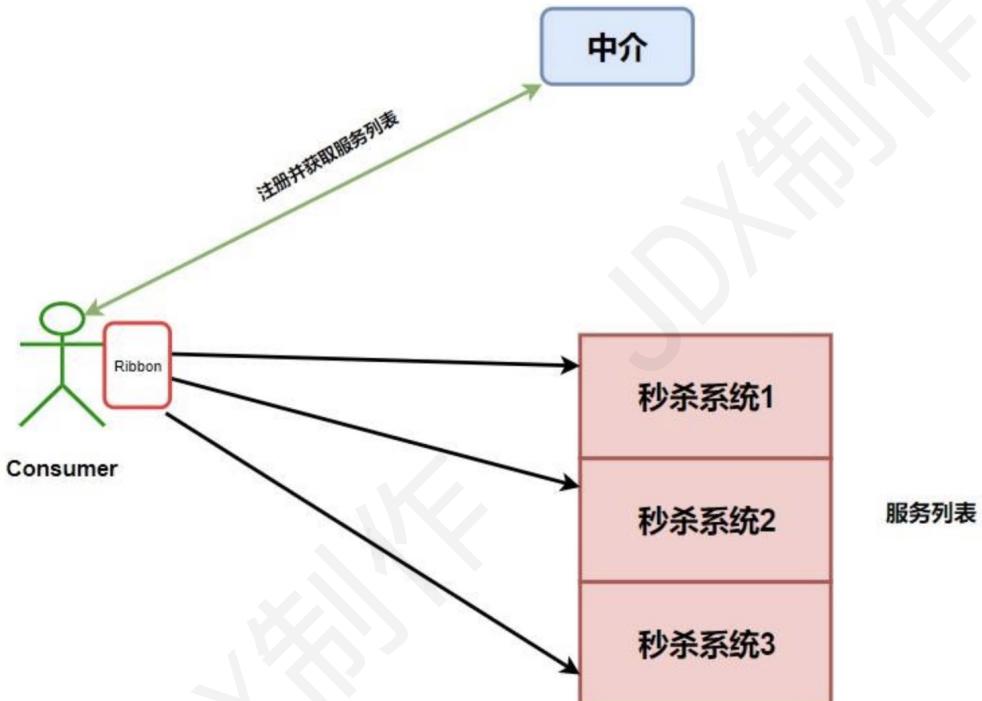
`Ribbon` 是 `Netflix` 公司的一个开源的负载均衡项目，是一个客户端/进程内负载均衡器，运行在消费者端。

我们再举个栗子，比如我们设计了一个秒杀系统，但是为了整个系统的 **高可用**，我们需要将这个系统做一个集群，而这个时候我们消费者就可以拥有多个秒杀系统的调用途径了，如下图。



如果这个时候我们没有进行一些 **均衡操作**，如果我们对 **秒杀系统1** 进行大量的调用，而另外两个基本不请求，就会导致 **秒杀系统1** 崩溃，而另外两个就变成了傀儡，那么我们为什么还要做集群，我们高可用体现的意义又在哪呢？

所以 **Ribbon** 出现了，注意我们上面加粗的几个字——**运行在消费者端**。指的是，**Ribbon** 是运行在消费者端的负载均衡器，如下图。



其工作原理就是 **Consumer** 端获取到了所有的服务列表之后，在其内部使用**负载均衡算法**，进行对多个系统的调用。

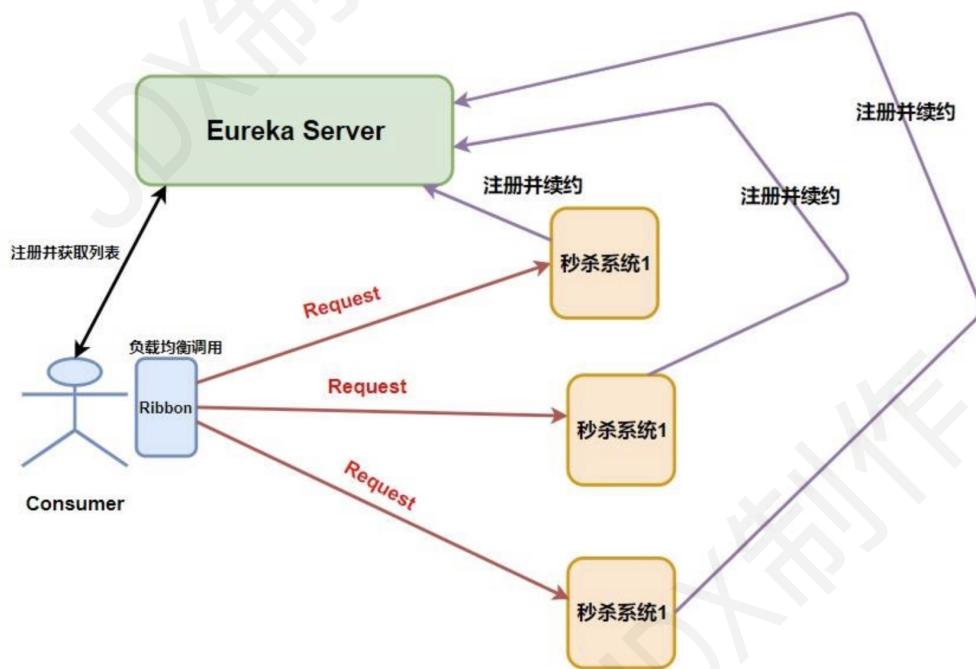
1.4.3 Nginx 和 Ribbon 的对比

提到 **负载均衡** 就不得不提到大名鼎鼎的 **Nginx** 了，而和 **Ribbon** 不同的是，它是一种**集中式**的负载均衡器。

何为集中式呢？简单理解就是 **将所有请求都集中起来，然后再进行负载均衡**。如下图。



我们可以看到 **Nginx** 是接收了所有的请求进行负载均衡的，而对于 **Ribbon** 来说它是在消费者端进行的负载均衡。如下图。



请注意 **Request** 的位置，在 **Nginx** 中请求是先进入负载均衡器，而在 **Ribbon** 中是先在客户端进行负载均衡才进行请求的。

1.4.4 Ribbon 的几种负载均衡算法

负载均衡，不管 **Nginx** 还是 **Ribbon** 都需要其算法的支持，如果我没记错的话 **Nginx** 使用的是轮询和加权轮询算法。而在 **Ribbon** 中有更多的负载均衡调度算法，其默认是使用的 **RoundRobinRule** 轮询策略。

- **RoundRobinRule**: 轮询策略。**Ribbon** 默认采用的策略。若经过一轮轮询没有找到可用的 **provider**，其最多轮询 10 轮。若最终还没有找到，则返回 **null**。
- **RandomRule**: 随机策略，从所有可用的 **provider** 中随机选择一个。
- **RetryRule**: 重试策略。先按照 **RoundRobinRule** 策略获取 **provider**，若获取失败，则在指定的时限内重试。默认的时限为 500 毫秒。

还有很多，这里不一一举了栗子，你最需要知道的是默认轮询算法，并且可以更换默认的负载均衡算法，只需要在配置文件中做出修改就行。

```
providerName:  
  ribbon:  
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule
```

当然，在 `Ribbon` 中你还可以自定义负载均衡算法，你只需要实现 `IRule` 接口，然后修改配置文件或者自定义 `Java Config` 类。

1.5 什么是 Open Feign

有了 `Eureka`，`RestTemplate`，`Ribbon`，我们就可以愉快地进行服务间的调用了，但是使用 `RestTemplate` 还是不方便，我们每次都要进行这样的调用。

```
@Autowired  
private RestTemplate restTemplate;  
// 这里是提供者A的ip地址，但是如果使用了 Eureka 那么就应该是提供者A的名称  
private static final String SERVICE_PROVIDER_A = "http://localhost:8081";  
  
@PostMapping("/judge")  
public boolean judge(@RequestBody Request request) {  
    String url = SERVICE_PROVIDER_A + "/service1";  
    // 是不是太麻烦了？？？每次都要 url、请求、返回类型的  
    return restTemplate.postForObject(url, request, Boolean.class);  
}
```

这样每次都调用 `RestTemplate` 的 API 是否太麻烦，我能不能像调用原来代码一样进行各个服务间的调用呢？

聪明的小朋友肯定想到了，那就用 映射 呀，就像域名和IP地址的映射。我们可以将被调用的服务代码映射到消费者端，这样我们就可以“无缝开发”啦。

`OpenFeign` 也是运行在消费者端的，使用 `Ribbon` 进行负载均衡，所以 `OpenFeign` 直接内置了 `Ribbon`。

在导入了 `open Feign` 之后我们就可以进行愉快编写 `Consumer` 端代码了。

```
// 使用 @FeignClient 注解来指定提供者的名字  
@FeignClient(value = "eureka-client-provider")  
public interface TestClient {  
    // 这里一定要注意需要使用的是提供者那端的请求相对路径，这里就相当于映射了  
    @RequestMapping(value = "/provider/xxx",  
    method = RequestMethod.POST)  
    CommonResponse<List<Plan>> getPlans(@RequestBody PlanGetRequest request);  
}
```

然后我们在 `Controller` 就可以像原来调用 `Service` 层代码一样调用它了。

```
@RestController
public class TestController {
    // 这里就相当于原来自动注入的 service
    @Autowired
    private Testclient testClient;
    // controller 调用 service 层代码
    @RequestMapping(value = "/test", method = RequestMethod.POST)
    public CommonResponse<List<Plan>> get(@RequestBody PlanGetRequest request) {
        return testClient.getPlans(request);
    }
}
```

1.6 必不可少的 Hystrix

1.6.1 什么是 Hystrix之熔断和降级

在分布式环境中，不可避免地会有许多服务依赖项中的某些失败。Hystrix是一个库，可通过添加等待时间容限和容错逻辑来帮助您控制这些分布式服务之间的交互。Hystrix通过隔离服务之间的访问点，停止服务之间的级联故障并提供后备选项来实现此目的，所有这些都可以提高系统的整体弹性。

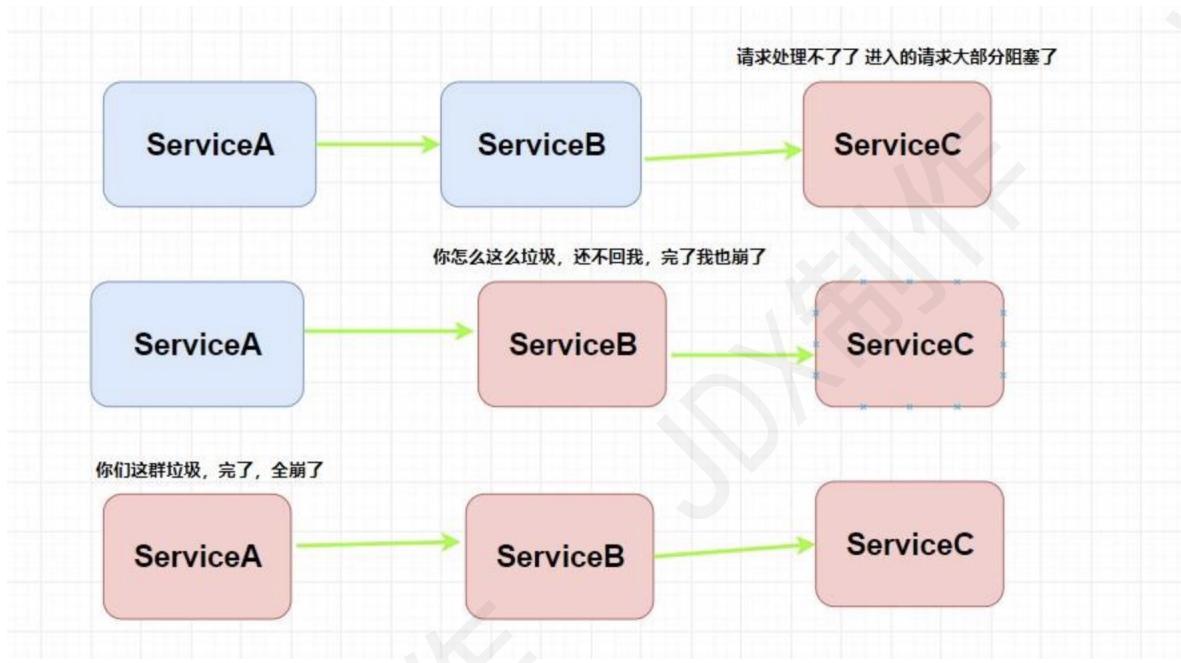
总体来说 Hystrix 就是一个能进行 **熔断** 和 **降级** 的库，通过使用它能提高整个系统的弹性。

那么什么是 熔断和降级 呢？再举个栗子，此时我们整个微服务系统是这样的。服务A调用了服务B，服务B再调用了服务C，但是因为某些原因，服务C顶不住了，这个时候大量请求会在服务C阻塞。



服务C阻塞了还好，毕竟只是一个系统崩溃了。但是请注意这个时候因为服务C不能返回响应，那么服务B调用服务C的请求就会阻塞，同理服务B阻塞了，那么服务A也会阻塞崩溃。

请注意，为什么阻塞会崩溃。因为这些请求会消耗占用系统的线程、IO 等资源，消耗完你这个系统服务器不就崩了么。



这就叫 **服务雪崩**。妈耶，上面两个 **熔断** 和 **降级** 你都没给我解释清楚，你现在又给我扯什么 **服务雪崩**？

别急，听我慢慢道来。

不听我也得讲下去！

所谓 **熔断** 就是服务雪崩的一种有效解决方案。当指定时间窗内的请求失败率达到设定阈值时，系统将通过 **断路器** 直接将此请求链路断开。

也就是我们上面服务B调用服务C在指定时间窗内，调用的失败率到达了一定的值，那么 **Hystrix** 则会自动将服务B与C之间的请求都断了，以免导致服务雪崩现象。

其实这里所讲的 **熔断** 就是指的 **Hystrix** 中的 **断路器模式**，你可以使用简单的 `@HystrixCommand` 注解来标注某个方法，这样 **Hystrix** 就会使用 **断路器** 来“包装”这个方法，每当调用时间超过指定时间时（默认为1000ms），断路器将会中断对这个方法的调用。

当然你可以对这个注解的很多属性进行设置，比如设置超时时间，像这样。

```
@HystrixCommand(
    commandProperties = {@HystrixProperty(name =
"execution.isolation.thread.timeoutInMilliseconds", value = "1200")}
)
public List<Xxx> getXxx() {
    // ...省略代码逻辑
}
```

但是，我查阅了一些博客，发现他们都将 **熔断** 和 **降级** 的概念混淆了，以我的理解，**降级是为了更好的用户体验，当一个方法调用异常时，通过执行另一种代码逻辑来给用户友好的回复**。这也就对应着 **Hystrix** 的 **后备处理** 模式。你可以通过设置 `fallbackMethod` 来给一个方法设置备用的代码逻辑。比如这个时候有一个热点新闻出现了，我们会推荐给用户查看详情，然后用户会通过id去查询新闻的详情，但是因为这条新闻太火了（比如最近什么*易对吧），大量用户同时访问可能会导致系统崩溃，那么我们就进行 **服务降级**，一些请求会做一些降级处理比如当前人数太多请稍后查看等等。

```
// 指定了后备方法调用
@HystrixCommand(fallbackMethod = "getHystrixNews")
@GetMapping("/get/news")
public News getNews(@PathVariable("id") int id) {
    // 调用新闻系统的获取新闻api 代码逻辑省略
}

// 
public News getHystrixNews(@PathVariable("id") int id) {
    // 做服务降级
    // 返回当前人数太多, 请稍后查看
}
```

1.6.2 什么是Hystrix之其他

我在阅读《Spring微服务实战》这本书的时候还接触到了一个 **舱壁模式** 的概念。在不使用舱壁模式的情况下，服务A调用服务B，这种调用默认的是 **使用同一批线程来执行的**，而在一个服务出现性能问题的时候，就会出现所有线程被刷爆并等待处理工作，同时阻塞新请求，最终导致程序崩溃。而舱壁模式会将远程资源调用隔离在他们自己的线程池中，以便可以控制单个表现不佳的服务，而不会使该程序崩溃。

具体其原理我推荐大家自己去了解一下，本篇文章中对 **舱壁模式** 不做过多解释。当然还有 **Hystrix仪表盘**，它是**用来实时监控 Hystrix 的各项指标信息的**，这里我将这个问题也抛出去，希望有不了解的可以自己去搜索一下。

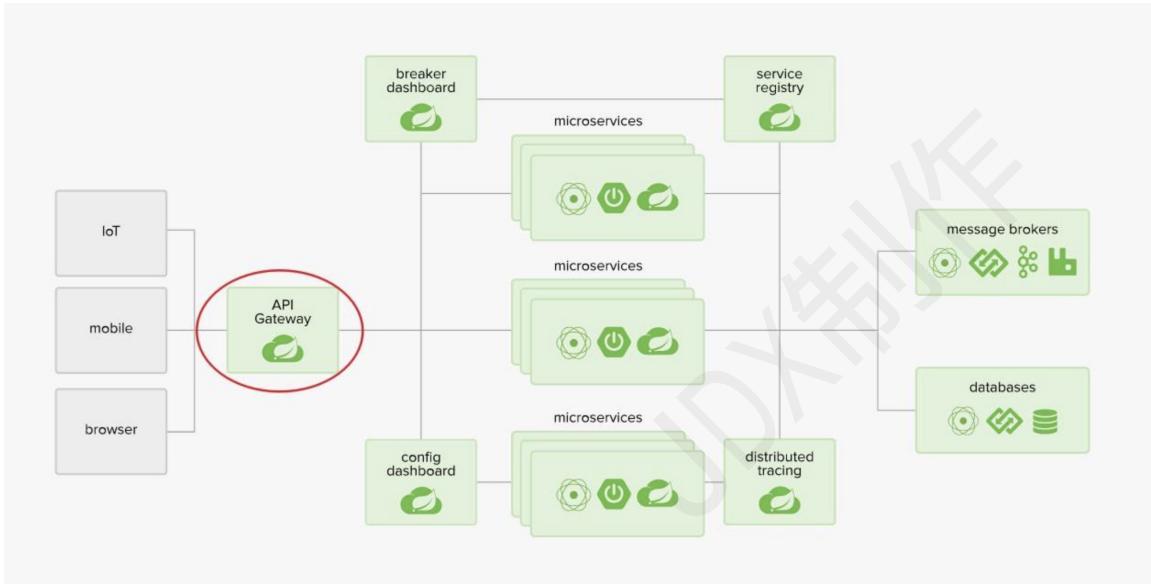
1.7 微服务网关——Zuul

ZUUL 是从设备和 web 站点到 Netflix 流应用后端的所有请求的前门。作为边界服务应用，ZUUL 是为了实现动态路由、监视、弹性和安全性而构建的。它还具有根据情况将请求路由到多个 Amazon Auto Scaling Groups (亚马逊自动缩放组，亚马逊的一种云计算方式) 的能力

在上面我们学习了 **Eureka** 之后我们就知道了 **服务提供者** 是 **消费者** 通过 **Eureka Server** 进行访问的，即 **Eureka Server** 是 **服务提供者** 的统一入口。那么整个应用中存在那么多 **消费者** 需要用户进行调用，这个时候用户该怎样访问这些 **消费者工程** 呢？当然可以像之前那样直接访问这些工程。但这种方式没有统一的消费者工程调用入口，不便于访问与管理，而 Zuul 就是这样的一个对于 **消费者的统一入口**。

如果学过前端的肯定都知道 Router 吧，比如 Flutter 中的路由，Vue，React中的路由，用了 Zuul 你会发现在路由功能方面和前端配置路由基本是一个理。我偶尔撸撸 Flutter。

大家对网关应该很熟吧，简单来讲网关是系统唯一对外的入口，介于客户端与服务器端之间，用于对请求进行**鉴权、限流、路由、监控**等功能。



没错，网关有的功能，`Zuul` 基本都有。而 `Zuul` 中最关键的就是 **路由和过滤器** 了，在官方文档中 `Zuul` 的标题就是

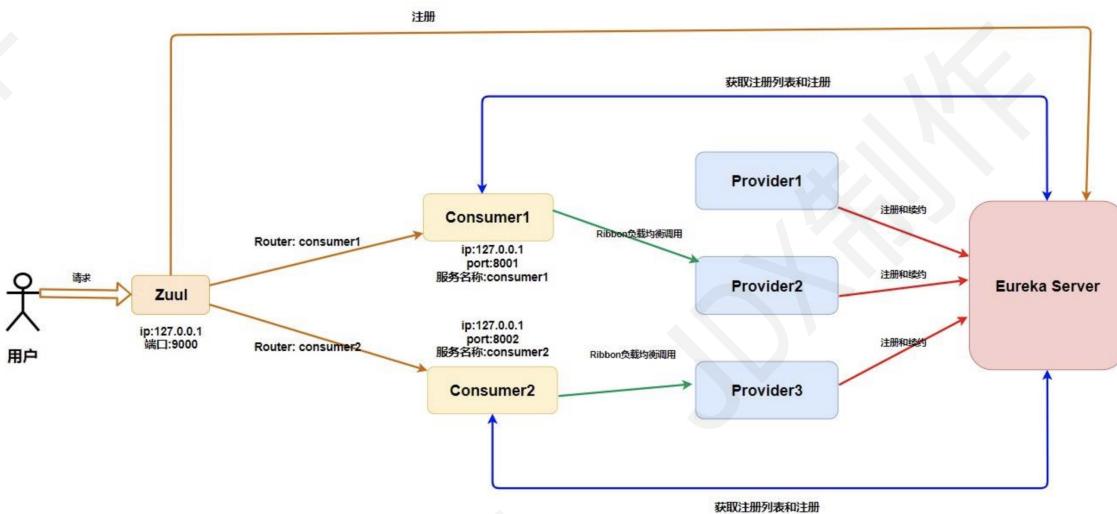
Router and Filter : Zuul

1.7.1 Zuul 的路由功能

1) 简单配置

本来想给你们复制一些代码，但是想了想，因为各个代码配置比较零散，看起来也比较零散，我决定还是给你们画个图来解释吧。

比如这个时候我们已经向 `Eureka Server` 注册了两个 `Consumer`、三个 `Provider`，这个时候我们再加个 `Zuul` 网关应该变成这样子了。



emmm，信息量有点大，我来解释一下。关于前面的知识我就不解释了。

首先，`Zuul` 需要向 `Eureka` 进行注册，注册有啥好处呢？

你傻呀，`Consumer` 都向 `Eureka Server` 进行注册了，我网关是不是只要注册就能拿到所有 `Consumer` 的信息了？

拿到信息有什么好处呢？

我拿到信息我是不是可以获取所有的 `Consumer` 的元数据(名称, ip, 端口)？

拿到这些元数据有什么好处呢？拿到了我们是不是直接可以做**路由映射**？比如原来用户调用 `Consumer1` 的接口 `localhost:8001/studentInfo/update` 这个请求，我们是不是可以这样进行调用了呢？`localhost:9000/consumer1/studentInfo/update` 呢？你这样是不是恍然大悟了？

这里的url为了让更多人看懂所以没有使用 restful 风格。

上面的你理解了，那么就能理解关于 `zuul` 最基本的配置了，看下面。

```
server:  
  port: 9000  
eureka:  
  client:  
    service-url:  
      # 这里只要注册 Eureka 就行了  
      defaultZone: http://localhost:9997/eureka
```

然后在启动类上加入 `@EnableZuulProxy` 注解就行了。没错，就是那么简单。

2) 统一前缀

这个很简单，就是我们可以在前面加一个统一的前缀，比如我们刚刚调用的是 `localhost:9000/consumer1/studentInfo/update`，这个时候我们在 `yaml` 配置文件中添加如下。

```
zuul:  
  prefix: /zuul
```

这样我们就需要通过 `localhost:9000/zuul/consumer1/studentInfo/update` 来进行访问了。

3) 路由策略配置

你会发现前面的访问方式(直接使用服务名)，需要将微服务名称暴露给用户，会存在安全性问题。所以，可以自定义路径来替代微服务名称，即自定义路由策略。

```
zuul:  
  routes:  
    consumer1: /FrancisQ1/**  
    consumer2: /FrancisQ2/**
```

这个时候你就可以使用 `localhost:9000/zuul/FrancisQ1/studentInfo/update` 进行访问了。`

4) 服务名屏蔽

这个时候你别以为你好了，你可以试试，在你配置完路由策略之后使用微服务名称还是可以访问的，这个时候你需要将服务名屏蔽。

```
zuul:  
  ignore-services: "*"
```

5) 路径屏蔽

`zuul` 还可以指定屏蔽掉的路径 URI，即只要用户请求中包含指定的 URI 路径，那么该请求将无法访问到指定的服务。通过该方式可以限制用户的权限。

```
zuul:  
  ignore-patterns: **/auto/**
```

这样关于 auto 的请求我们就可以过滤掉了。

** 代表匹配多级任意路径

* 代表匹配一级任意路径

6) 敏感请求头屏蔽

默认情况下，像 `Cookie`、`Set-Cookie` 等敏感请求头信息会被 `zuul` 屏蔽掉，我们可以将这些默认屏蔽去掉，当然，也可以添加要屏蔽的请求头。

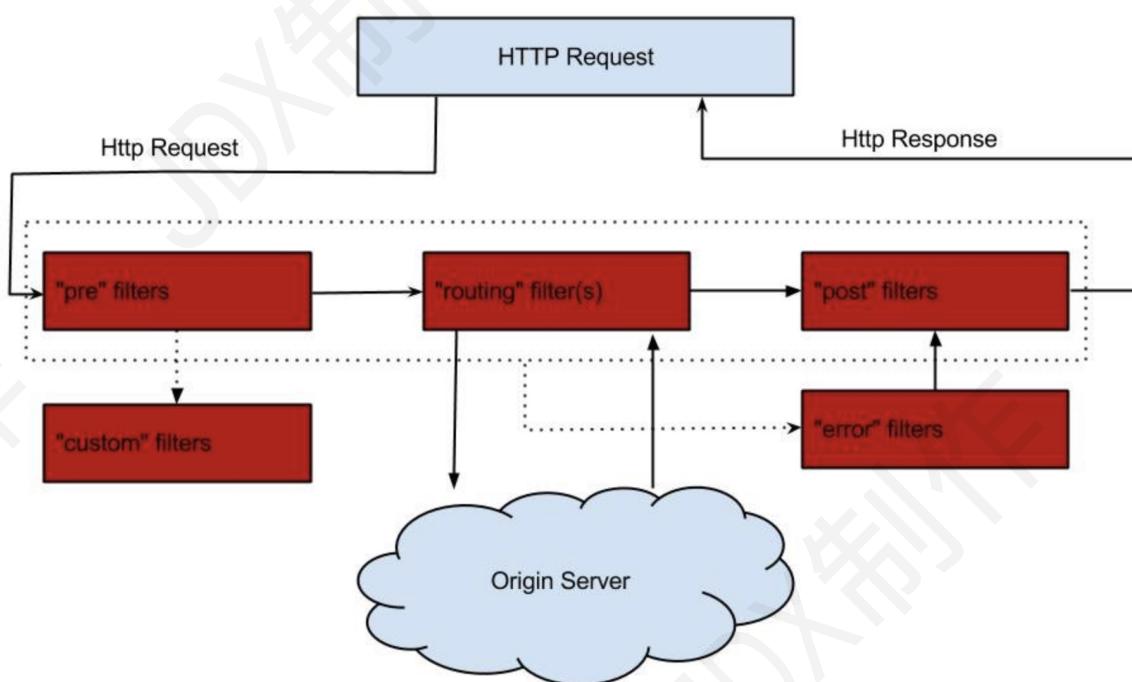
1.7.2 Zuul 的过滤功能

如果说，路由功能是 `zuul` 的基操的话，那么过滤器就是 `zuul` 的利器了。毕竟所有请求都经过网关 (`Zuul`)，那么我们可以进行各种过滤，这样我们就能实现 **限流**，**灰度发布**，**权限控制** 等等。

1) 简单实现一个请求时间日志打印

要实现自己定义的 `Filter` 我们只需要继承 `zuulFilter` 然后将这个过滤器类以 `@Component` 注解加入 Spring 容器中就行了。

在给你们看代码之前我先给你们解释一下关于过滤器的一些注意点。



过滤器类型：`Pre`、`Routing`、`Post`。前置 `Pre` 就是在请求之前进行过滤，`Routing` 路由过滤器就是我们上面所讲的路由策略，而 `Post` 后置过滤器就是在 `Response` 之前进行过滤的过滤器。你可以观察上图结合着理解，并且下面我会给出相应的注释。

```
// 加入Spring容器
@Component
public class PreRequestFilter extends zuulFilter {
    // 返回过滤器类型 这里是前置过滤器
    @Override
    public String filterType() {
        return FilterConstants.PRE_TYPE;
    }
    // 指定过滤顺序 越小越先执行，这里第一个执行
    // 当然不是只真正第一个 在Zuul内置中有其他过滤器会先执行
    // 那是写死的 比如 SERVLET_DETECTION_FILTER_ORDER = -3
    @Override
```

```

public int filterOrder() {
    return 0;
}
// 什么时候该进行过滤
// 这里我们可以进行一些判断，这样我们就可以过滤掉一些不符合规定的请求等等
@Override
public boolean shouldFilter() {
    return true;
}
// 如果过滤器允许通过则怎么进行处理
@Override
public Object run() throws ZuulException {
    // 这里我设置了全局的RequestContext并记录了请求开始时间
    RequestContext ctx = RequestContext.getCurrentContext();
    ctx.set("startTime", System.currentTimeMillis());
    return null;
}
}

```

```

// Lombok的日志
@Slf4j
// 加入 Spring 容器
@Component
public class AccessLogFilter extends ZuulFilter {
    // 指定该过滤器的过滤类型
    // 此时是后置过滤器
    @Override
    public String filterType() {
        return FilterConstants.POST_TYPE;
    }
    // SEND_RESPONSE_FILTER_ORDER 是最后一个过滤器
    // 我们此过滤器在它之前执行
    @Override
    public int filterOrder() {
        return FilterConstants.SEND_RESPONSE_FILTER_ORDER - 1;
    }
    @Override
    public boolean shouldFilter() {
        return true;
    }
    // 过滤时执行的策略
    @Override
    public Object run() throws ZuulException {
        RequestContext context = RequestContext.getCurrentContext();
        HttpServletRequest request = context.getRequest();
        // 从RequestContext获取原先的开始时间 并通过它计算整个时间间隔
        Long startTime = (Long) context.get("startTime");
        // 这里我可以获取HttpServletRequest来获取URI并且打印出来
        String uri = request.getRequestURI();
        long duration = System.currentTimeMillis() - startTime;
        log.info("uri: " + uri + ", duration: " + duration / 100 + "ms");
        return null;
    }
}

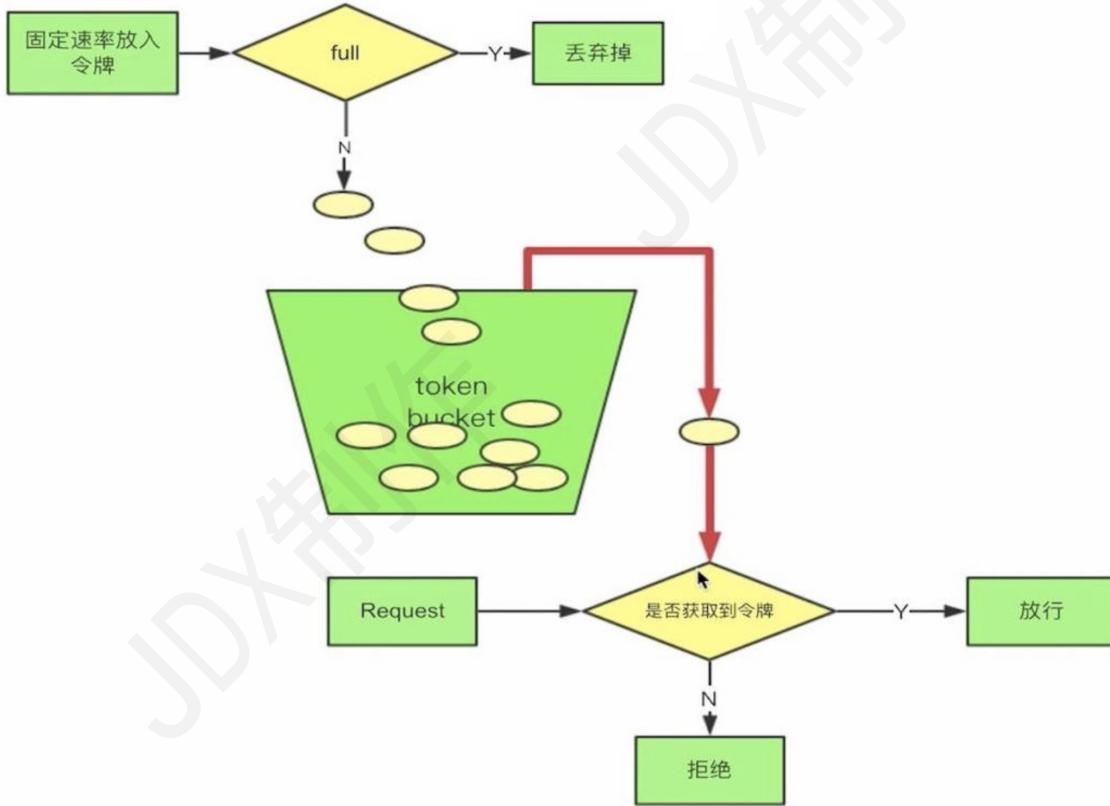
```

上面就简单实现了请求时间日志打印功能，你有没有感受到 zuul 过滤功能的强大了呢？

没有？好的、那我们再来。

2) 令牌桶限流

当然不仅仅是令牌桶限流方式，zuul 只要是限流的活它都能干，这里我只是简单举个栗子。



我先来解释一下什么是 令牌桶限流 吧。

首先我们会有个桶，如果里面没有满那么就会以一定 **固定的速率** 会往里面放令牌，一个请求过来首先要从桶中获取令牌，如果没有获取到，那么这个请求就拒绝，如果获取到那么就放行。很简单吧，啊哈哈。

下面我们就通过 zuul 的前置过滤器来实现一下令牌桶限流。

```
package com.lqq.zuul.filter;

import com.google.common.util.concurrent.RateLimiter;
import com.netflix.zuul.ZuulFilter;
import com.netflix.zuul.context.RequestContext;
import com.netflix.zuul.exception.ZuulException;
import lombok.extern.slf4j.Slf4j;
import org.springframework.cloud.netflix.zuul.filters.support.FilterConstants;
import org.springframework.stereotype.Component;

@Component
@Slf4j
public class RouteFilter extends ZuulFilter {
    // 定义一个令牌桶，每秒产生2个令牌，即每秒最多处理2个请求
    private static final RateLimiter RATE_LIMITER = RateLimiter.create(2);
    @Override
    public String filterType() {
        return FilterConstants.PRE_TYPE;
    }
}
```

```

@Override
public int filterOrder() {
    return -5;
}

@Override
public Object run() throws ZuulException {
    log.info("放行");
    return null;
}

@Override
public boolean shouldFilter() {
    RequestContext context = RequestContext.getCurrentContext();
    if(!RATE_LIMITER.tryAcquire()) {
        log.warn("访问量超载");
        // 指定当前请求未通过过滤
        context.setSendZuulResponse(false);
        // 向客户端返回响应码429, 请求数量过多
        context.setResponseStatusCode(429);
        return false;
    }
    return true;
}
}

```

这样我们就能将请求数量控制在一秒两个，有没有觉得很酷？

1.7.3 关于 Zuul 的其他

Zuul 的过滤器的功能肯定不止上面我所实现的两种，它还可以实现 **权限校验**，包括我上面提到的 **灰度发布** 等等。

当然，Zuul 作为网关肯定也存在 **单点问题**，如果我们要保证 Zuul 的高可用，我们就需要进行 Zuul 的集群配置，这个时候可以借助额外的一些负载均衡器比如 Nginx。

##Spring Cloud配置管理——Config

1.7.4 为什么要使用进行配置管理？

当我们的微服务系统开始慢慢地庞大起来，那么多 Consumer、Provider、Eureka Server、Zuul 系统都会持有自己的配置，这个时候我们在项目运行的时候可能需要更改某些应用的配置，如果我们不进行配置的统一管理，我们只能去每个应用下一个一个寻找配置文件然后修改配置文件再重启应用。

首先对于分布式系统而言我们就不应该去每个应用下去分别修改配置文件，再者对于重启应用来说，服务无法访问所以直接抛弃了可用性，这是我们更不愿见到的。

那么有没有一种方法既能对配置文件统一地进行管理，又能在项目运行时动态修改配置文件呢？

那就是我今天所要介绍的 Spring cloud Config。

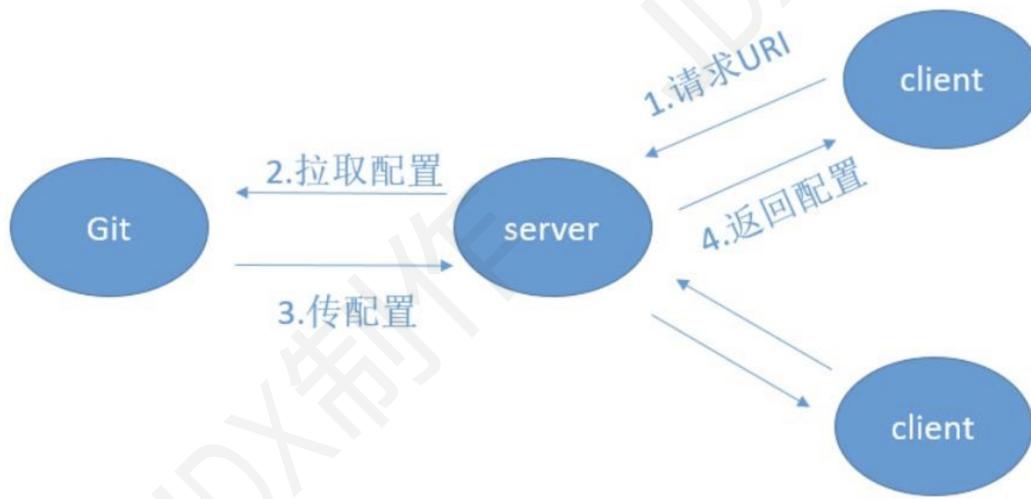
能进行配置管理的框架不止 Spring cloud Config 一种，大家可以根据需求自己选择 (disconf, 阿波罗等等)。而且对于 Config 来说有些地方实现的不是那么尽人意。

1.7.5 Config 是什么

| `Spring Cloud Config` 为分布式系统中的外部化配置提供服务器和客户端支持。使用 `Config` 服务器，可以在中心位置管理所有环境中应用程序的外部属性。

简单来说，`Spring Cloud Config` 就是能将各个应用/系统/模块的配置文件存放到统一的地方然后进行管理(Git 或者 SVN)。

你想一下，我们的应用是不是只有启动的时候才会进行配置文件的加载，那么我们的 `Spring Cloud Config` 就暴露出一个接口给启动应用来获取它所想要的配置文件，应用获取到配置文件然后再进行它的初始化工作。就如下图。



当然这里你肯定还会有一个疑问，如果我在应用运行时去更改远程配置仓库(Git)中的对应配置文件，那么依赖于这个配置文件的已启动的应用会不会进行其相应配置的更改呢？

答案是不会的。

什么？那怎么进行动态修改配置文件呢？这不是出现了 **配置漂移** 吗？你个渣男，你又骗我！

别急嘛，你可以使用 `webhooks`，这是 `github` 提供的功能，它能确保远端库的配置文件更新后客户端中的配置信息也得到更新。

噢噢，这还差不多。我去查查怎么用。

慢着，听我说完，`webhooks` 虽然能解决，但是你了解一下会发现它根本不适合用于生产环境，所以基本不会使用它的。

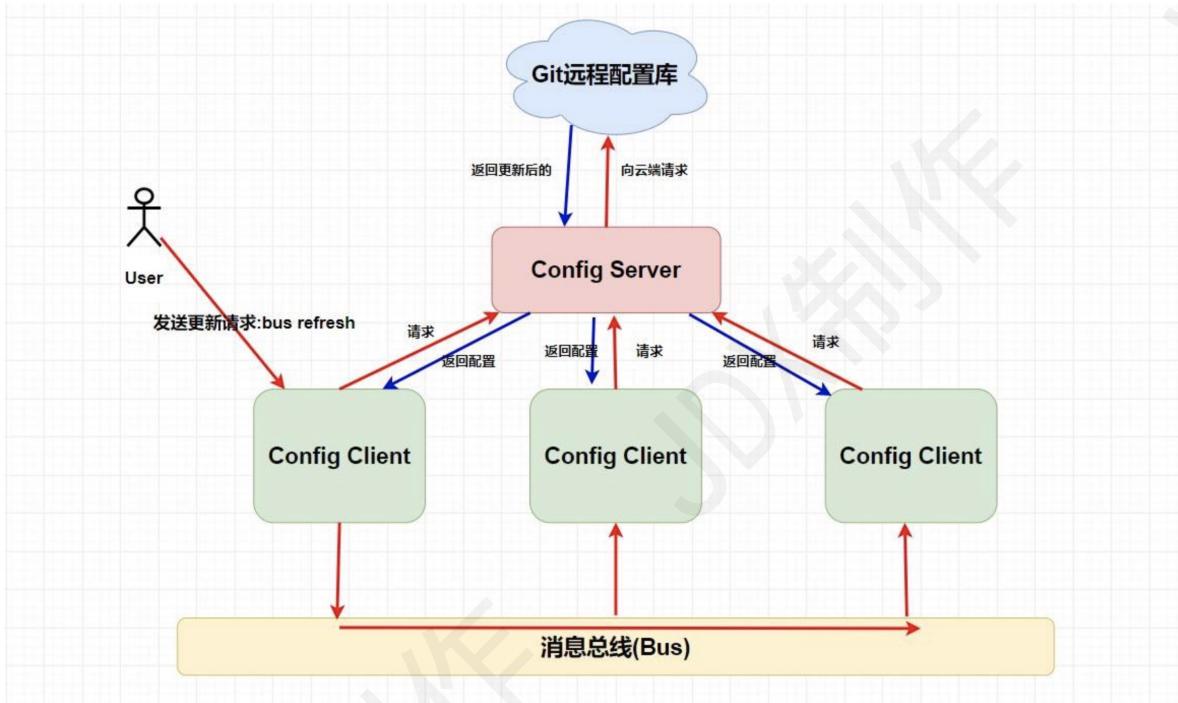
而一般我们会使用 `Bus` 消息总线 + `Spring Cloud Config` 进行配置的动态刷新。

1.8 引出 Spring Cloud Bus

| 用于将服务和服务实例与分布式消息系统链接在一起的事件总线。在集群中传播状态更改很有用（例如配置更改事件）。

你可以简单理解为 `Spring Cloud Bus` 的作用就是**管理和广播分布式系统中的消息**，也就是消息引擎系统中的广播模式。当然作为**消息总线**的 `Spring Cloud Bus` 可以做很多事而不仅仅是客户端的配置刷新功能。

而拥有了 `Spring Cloud Bus` 之后，我们只需要创建一个简单的请求，并且加上 `@RefreshScope` 注解就能进行配置的动态修改了，下面我画了张图供你理解。



1.9 总结

这篇文章中我带大家初步了解了 `spring cloud` 的各个组件，他们有

- `Eureka` 服务发现框架
- `Ribbon` 进程内负载均衡器
- `Open Feign` 服务调用映射
- `Hystrix` 服务降级熔断器
- `zuul` 微服务网关
- `Config` 微服务统一配置中心
- `Bus` 消息总线

如果你能这个时候能看懂文首那张图，也就说明了你已经对 `spring cloud` 微服务有了一定的架构认识。

七、必会工具

(一). Git

1. 版本控制

1.1 什么是版本控制

版本控制是一种记录一个或若干文件内容变化，以便将来查阅特定版本修订情况的系统。除了项目源代码，你可以对任何类型的文件进行版本控制。

1.2 为什么要版本控制

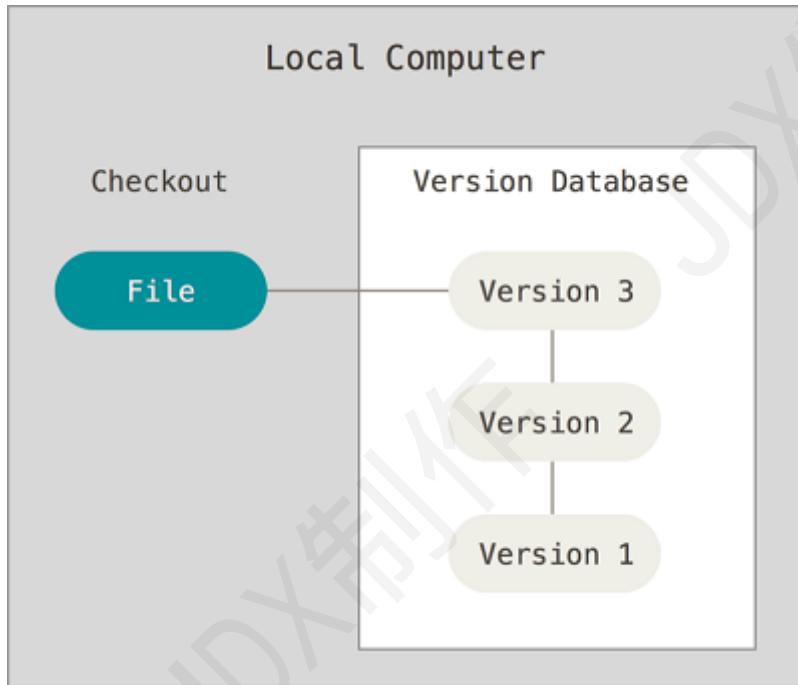
有了它你就可以将某个文件回溯到之前的状态，甚至将整个项目都回退到过去某个时间点的状态，你可以比较文件的变化细节，查出最后是谁修改了哪个地方，从而找出导致怪异问题出现的原因，又是谁在何时报告了某个功能缺陷等等。

1.3 本地版本控制系统

获取更多资源可添加关注微信公众号：JAVA架构进阶之路；或添加微信：jiang10086a 免费获取面试真题，视频教程，笔记等。

许多人习惯用复制整个项目目录的方式来保存不同的版本，或许还会改名加上备份时间以示区别。这么做唯一的好处就是简单，但是特别容易犯错。有时候会混淆所在的工作目录，一不小心会写错文件或者覆盖意想外的文件。

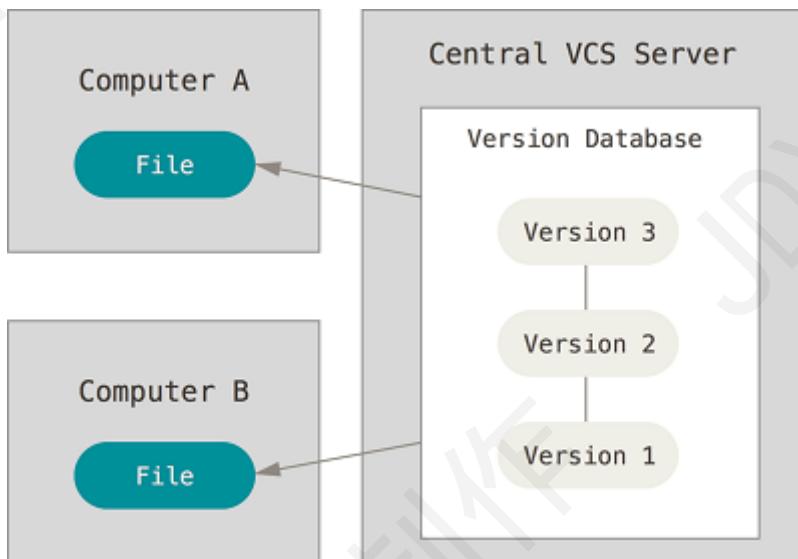
为了解决这个问题，人们很久以前就开发了许多种本地版本控制系统，大多都是采用某种简单的数据库来记录文件的历次更新差异。



1.4 集中化的版本控制系统

接下来人们又遇到一个问题，如何让在不同系统上的开发者协同工作？于是，集中化的版本控制系统（Centralized Version Control Systems, 简称 CVCS）应运而生。

集中化的版本控制系统都有一个单一的集中管理的服务器，保存所有文件的修订版本，而协同工作的人们都通过客户端连到这台服务器，取出最新的文件或者提交更新。



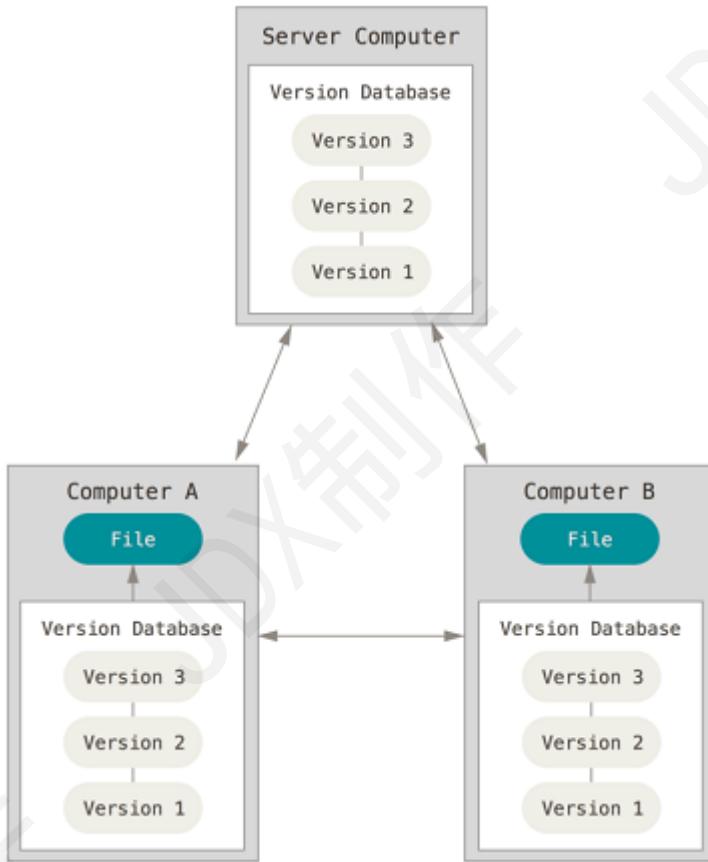
这么做虽然解决了本地版本控制系统无法让在不同系统上的开发者协同工作的诟病，但也还是存在下面的问题：

- **单点故障：**中央服务器宕机，则其他人无法使用；如果中心数据库磁盘损坏有没有进行备份，你将丢失所有数据。本地版本控制系统也存在类似问题，只要整个项目的历史记录被保存在单一位置，就有丢失所有历史更新记录的风险。
- **必须联网才能工作：**受网络状况、带宽影响。

1.5 分布式版本控制系统

于是分布式版本控制系统（Distributed Version Control System，简称 DVCS）面世了。Git 就是一个典型的分布式版本控制系统。

这类系统，客户端并不只提取最新版本的文件快照，而是把代码仓库完整地镜像下来。这么一来，任何一处协同工作用的服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复。因为每一次的克隆操作，实际上都是一次对代码仓库的完整备份。



分布式版本控制系统可以不用联网就可以工作，因为每个人的电脑上都是完整的版本库，当你修改了某个文件后，你只需要将自己的修改推送给别人就可以了。但是，在实际使用分布式版本控制系统的時候，很少会直接进行推送修改，而是使用一台充当“中央服务器”的东西。这个服务器的作用仅仅是用来方便“交换”大家的修改，没有它大家也一样干活，只是交换修改不方便而已。

分布式版本控制系统的优点不单是不必联网这么简单，后面我们还会看到 Git 极其强大的分支管理等功能。

2. 认识 Git

2.1 Git 简史

Linux 内核项目组当时使用分布式版本控制系统 BitKeeper 来管理和维护代码。但是，后来开发 BitKeeper 的商业公司同 Linux 内核开源社区的合作关系结束，他们收回了 Linux 内核社区免费使用 BitKeeper 的权力。Linux 开源社区（特别是 Linux 的缔造者 Linus Torvalds）基于使用 BitKeeper 时的经验教训，开发出自己的版本系统，而且对新的版本控制系统做了很多改进。

2.2 Git 与其他版本管理系统的区别

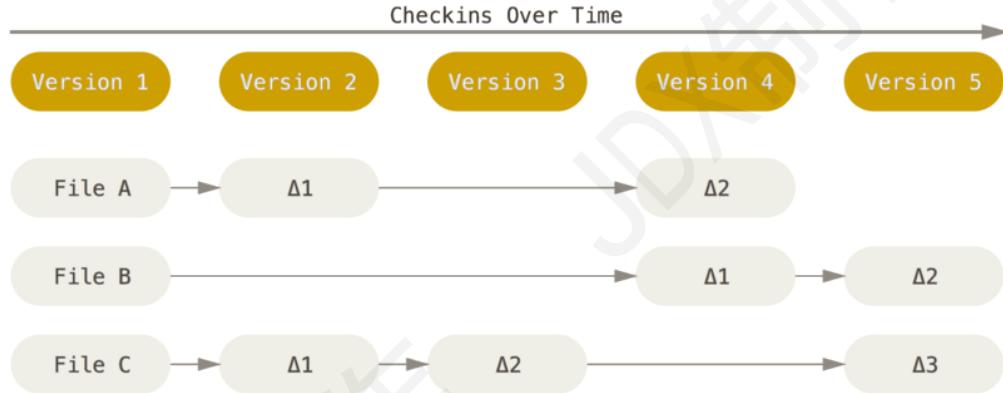
Git 在保存和对待各种信息的时候与其它版本控制系统有很大差异，尽管操作起来的命令形式非常相近，理解这些差异将有助于防止你使用中的困惑。

下面我们主要说一个关于 Git 其他版本管理系统的区别：对待数据的方式。

Git采用的是直接记录快照的方式，而非差异比较。我后面会详细介绍这两种方式的差别。

大部分版本控制系统（CVS、Subversion、Perforce、Bazaar 等等）都是以文件变更列表的方式存储信息，这类系统将它们保存的信息看作是一组基本文件和每个文件随时间逐步累积的差异。

具体原理如下图所示，理解起来其实很简单，每个我们对提交更新一个文件之后，系统记录都会记录这个文件做了哪些更新，以增量符号 Δ (Delta)表示。



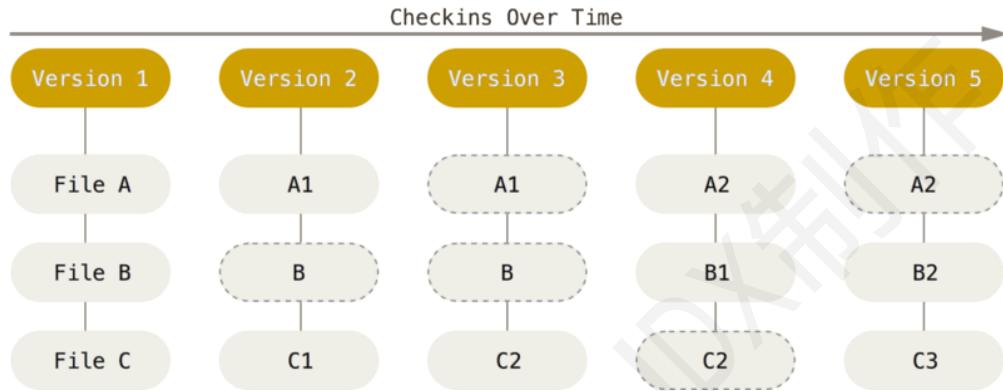
我们怎样才能得到一个文件的最终版本呢？

很简单，高中数学的基本知识，我们只需要将这些原文件和这些增加进行相加就行了。

这种方式有什么问题呢？

比如我们的增量特别特别多的话，如果我们要得到最终的文件是不是会耗费时间和性能。

Git 不按照以上方式对待或保存数据。反之，Git 更像是把数据看作是对小型文件系统的一组快照。每次你提交更新，或在 Git 中保存项目状态时，它主要对当时的全部文件制作一个快照并保存这个快照的索引。为了高效，如果文件没有修改，Git 不再重新存储该文件，而是只保留一个链接指向之前存储的文件。Git 对待数据更像是一个 **快照流**。

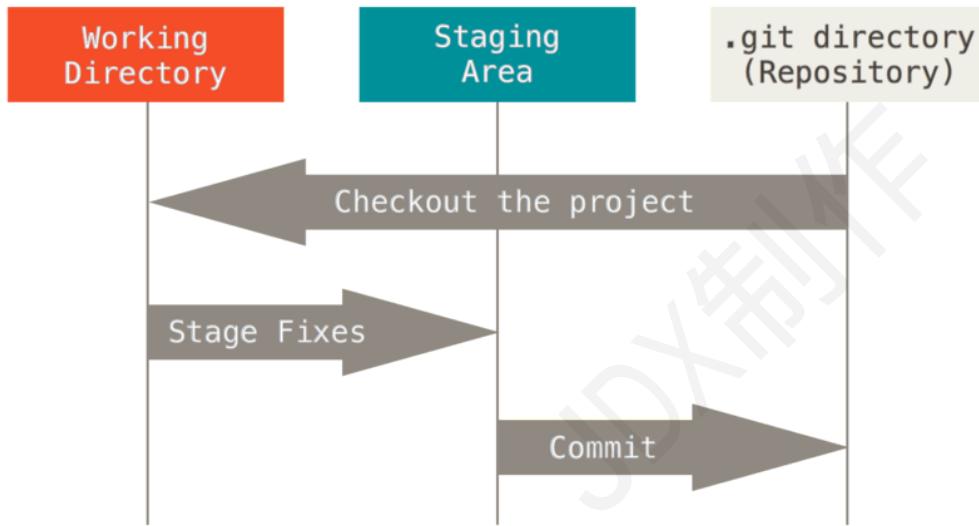


2.3 Git 的三种状态

Git 有三种状态，你的文件可能处于其中之一：

1. 已提交 (committed)：数据已经安全的保存在本地数据库中。
2. 已修改 (modified)：已修改表示修改了文件，但还没保存到数据库中。
3. 已暂存 (staged)：表示对一个已修改文件的当前版本做了标记，使之包含在下次提交的快照中。

由此引入 Git 项目的三个工作区域的概念：**Git 仓库(.git directory)**、**工作目录(Working Directory)**以及 **暂存区域(Staging Area)**。



基本的 Git 工作流程如下：

1. 在工作目录中修改文件。
2. 暂存文件，将文件的快照放入暂存区域。
3. 提交更新，找到暂存区域的文件，将快照永久性存储到 Git 仓库目录。

3. Git 使用快速入门

3.1 获取 Git 仓库

有两种取得 Git 项目仓库的方法。

1. 在现有目录中初始化仓库：进入项目目录运行 `git init` 命令，该命令将创建一个名为 `.git` 的子目录。
2. 从一个服务器克隆一个现有的 Git 仓库：`git clone [url]` 自定义本地仓库的名字：`git clone [url] directoryname`

3.2 记录每次更新到仓库

1. 检测当前文件状态：`git status`
2. 提出更改（把它们添加到暂存区）：`git add filename` (针对特定文件)、`git add *` (所有文件)、`git add *.txt` (支持通配符，所有 .txt 文件)
3. 忽略文件：`.gitignore` 文件
4. 提交更新：`git commit -m "代码提交信息"` (每次准备提交前，先用 `git status` 看下，是不是都已暂存起来了，然后再运行提交命令 `git commit`)
5. 跳过使用暂存区域更新的方式：`git commit -a -m "代码提交信息"`。`git commit` 加上 `-a` 选项，Git 就会自动把所有已经跟踪过的文件暂存起来一并提交，从而跳过 `git add` 步骤。
6. 移除文件：`git rm filename` (从暂存区域移除，然后提交。)
7. 对文件重命名：`git mv README.md README` (这个命令相当于 `mv README.md README`、`git rm README.md`、`git add README` 这三条命令的集合)

3.3 一个好的 Git 提交消息

一个好的 Git 提交消息如下：

标题行：用这一行来描述和解释你的这次提交

主体部分可以是很少的几行，来加入更多的细节来解释提交，最好是能给出一些相关的背景或者解释这个提交能修复和解决什么问题。

主体部分当然也可以有几段，但是一定要注意换行和句子不要太长。因为这样在使用 "git log" 的时候会有缩进比较好看。

提交的标题行描述应该尽量的清晰和尽量的一句话概括。这样就方便相关的 Git 日志查看工具显示和其他人的阅读。

3.4 推送改动到远程仓库

- 如果你还没有克隆现有仓库，并欲将你的仓库连接到某个远程服务器，你可以使用如下命令添加：
`git remote add origin <server>`, 比如我们要让本地的一个仓库和 Github 上创建的一个仓库关联可以这样 `git remote add origin https://github.com/snailclimb/test.git`
- 将这些改动提交到远端仓库：`git push origin master` (可以把 *master*换成你想要推送的任何分支)

如此你就能够将你的改动推送到所添加的服务器上去了。

3.5 远程仓库的移除与重命名

- 将 *test* 重命名位 *test1*: `git remote rename test test1`
- 移除远程仓库 *test1*: `git remote rm test1`

3.6 查看提交历史

在提交了若干更新，又或者克隆了某个项目之后，你也许想回顾下提交历史。完成这个任务最简单而又有效的工具是 `git log` 命令。`git log` 会按提交时间列出所有的更新，最近的更新排在最上面。

可以添加一些参数来查看自己希望看到的内容：

只看某个人的提交记录：

```
git log --author=bob
```

3.7 撤销操作

有时候我们提交完了才发现漏掉了几个文件没有添加，或者提交信息写错了。此时，可以运行带有 `--amend` 选项的提交命令尝试重新提交：

```
git commit --amend
```

取消暂存的文件

```
git reset filename
```

撤消对文件的修改：

```
git checkout -- filename
```

假如你想丢弃你在本地的所有改动与提交，可以到服务器上获取最新的版本历史，并将你本地主分支指向它：

```
git fetch origin  
git reset --hard origin/master
```

3.8 分支

分支是用来将特性开发绝缘开来的。在你创建仓库的时候，*master* 是“默认的”分支。在其他分支上进行开发，完成后再将它们合并到主分支上。

我们通常在开发新功能、修复一个紧急 bug 等等时候会选择创建分支。单分支开发好还是多分支开发好，还是要看具体场景来说。

创建一个名字叫做 test 的分支

```
git branch test
```

切换当前分支到 test (当你切换分支的时候，Git 会重置你的工作目录，使其看起来像回到了你在那个分支上最后一次提交的样子。Git 会自动添加、删除、修改文件以确保此时你的工作目录和这个分支最后一次提交时的样子一模一样)

```
git checkout test
```

```
SnailClimb@DESKTOP-QEM6F9G MINGW64 /g/GitHub/TWHomework/GitHomework (master)
$ git checkout test
Switched to a new branch 'test'
Branch 'test' set up to track remote branch 'test' from 'origin'.
SnailClimb@DESKTOP-QEM6F9G MINGW64 /g/GitHub/TWHomework/GitHomework (test)
```

你也可以直接这样创建分支并切换过去(上面两条命令的合写)

```
git checkout -b feature_x
```

切换到主分支

```
git checkout master
```

合并分支(可能会有冲突)

```
git merge test
```

把新建的分支删掉

```
git branch -d feature_x
```

将分支推送到远端仓库 (推送成功后其他人可见) :

```
git push origin
```

(二). Docker

本文只是对 Docker 的概念做了较为详细的介绍，并不涉及一些像 Docker 环境的安装以及 Docker 的一些常见操作和命令。

1. 认识容器

Docker 是世界领先的软件容器平台，所以想要搞懂 Docker 的概念我们必须先从容器开始说起。

1.1 什么是容器?

1.1.1 先来看看容器较为官方的解释

一句话概括容器：容器就是将软件打包成标准化单元，以用于开发、交付和部署。

- 容器镜像是轻量的、可执行的独立软件包，包含软件运行所需的所有内容：代码、运行时环境、系统工具、系统库和设置。
- 容器化软件适用于基于 Linux 和 Windows 的应用，在任何环境中都能够始终如一地运行。
- 容器赋予了软件独立性，使其免受外在环境差异（例如，开发和预演环境的差异）的影响，从而有助于减少团队间在相同基础设施上运行不同软件时的冲突。

1.1.2 再来看看容器较为通俗的解释

如果需要通俗的描述容器的话，我觉得容器就是一个存放东西的地方，就像书包可以装各种文具、衣柜可以放各种衣服、鞋架可以放各种鞋子一样。我们现在所说的容器存放的东西可能更偏向于应用比如网站、程序甚至是系统环境。

获取更多资源可添加关注微信公众号：JAVA架构进阶之路；或添加微信：jiang10086a 免费获取面试真题，视频教程，笔记等。



1.2 图解物理机,虚拟机与容器

关于虚拟机与容器的对比在后面会详细介绍到，这里只是通过网上的图片加深大家对于物理机、虚拟机与容器这三者的理解(下面的图片来源与网络)。

物理机



一栋楼一户人家，
独立地基，独立花园

虚拟机：



一栋楼包含多套房，
一套房一户人家，
共享地基，共享花园，独立卫生间、厨房和宽带

容器：

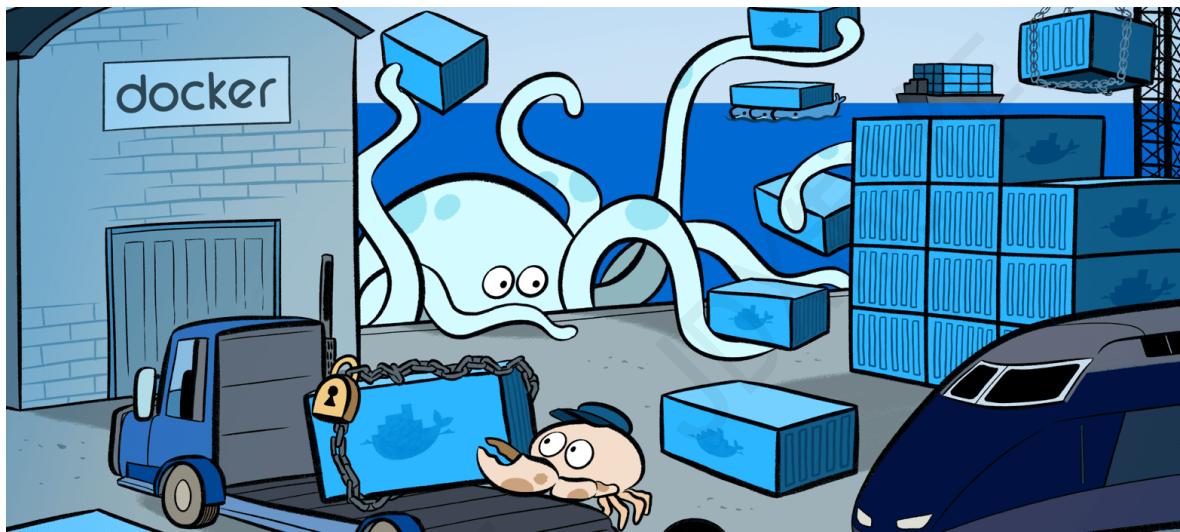


一套房隔成多个小隔间
(胶囊式公寓)，每个胶囊住一位租户，共享地基，共享花园，还共享卫生间、厨房和宽带

通过上面这三张抽象图，我们可以大概可以通过类比概括出：容器虚拟化的是操作系统而不是硬件，容器之间是共享同一套操作系统资源的。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。因此容器的隔离级别会稍低一些。

相信通过上面的解释大家对于容器这个既陌生又熟悉的概念有了一个初步的认识，下面我们就来谈谈 Docker 的一些概念。

2. 再来谈谈 Docker 的一些概念

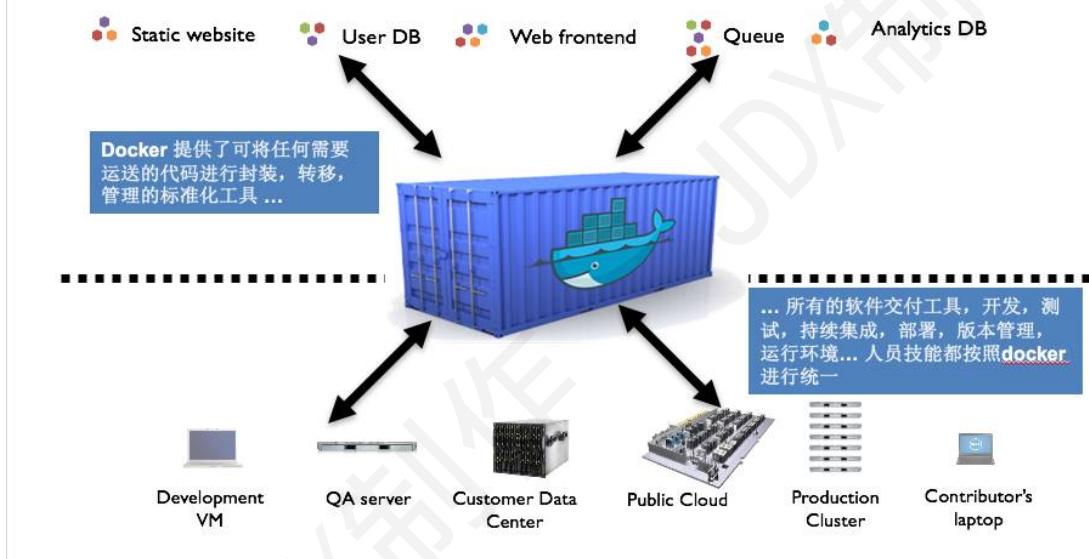


2.1 什么是 Docker?

说实话关于 Docker 是什么并不太好说，下面我通过四点向你说明 Docker 到底是个什么东西。

- Docker 是世界领先的软件容器平台。
- Docker 使用 Google 公司推出的 Go 语言进行开发实现，基于 Linux 内核 提供的 CGroup 功能和 name space 来实现的，以及 AUFS 类的 UnionFS 等技术，对进程进行封装隔离，属于操作系统层面的虚拟化技术。由于隔离的进程独立于宿主和其它的隔离的进程，因此也称其为容器。
- Docker 能够自动执行重复性任务，例如搭建和配置开发环境，从而解放了开发人员以便他们专注在真正重要的事情上：构建杰出的软件。
- 用户可以方便地创建和使用容器，把自己的应用放入容器。容器还可以进行版本管理、复制、分享、修改，就像管理普通的代码一样。

Docker : 代码集装箱装卸工



2.2 Docker 思想

- 集装箱
- 标准化：
 - ① 运输方式
 - ② 存储方式
 - ③ API 接口
- 隔离

2.3 Docker 容器的特点

- 轻量

在一台机器上运行的多个 Docker 容器可以共享这台机器的操作系统内核；它们能够迅速启动，只需占用很少的计算和内存资源。镜像是通过文件系统层进行构造的，并共享一些公共文件。这样就能尽量降低磁盘用量，并能更快地下载镜像。

- 标准

Docker 容器基于开放式标准，能够在所有主流 Linux 版本、Microsoft Windows 以及包括 VM、裸机服务器和云在内的任何基础设施上运行。

- 安全

Docker 赋予应用的隔离性不仅限于彼此隔离，还独立于底层的基础设施。Docker 默认提供最强的隔离，因此应用出现问题，也只是单个容器的问题，而不会波及到整台机器。

2.4 为什么要用 Docker？

- Docker 的镜像提供了除内核外完整的运行时环境，确保了应用运行环境一致性，从而不会再出现“这段代码在我机器上没问题啊”这类问题；——一致的运行环境
- 可以做到秒级、甚至毫秒级的启动时间。大大的节约了开发、测试、部署的时间。——更快速的启动时间
- 避免公用的服务器，资源会容易受到其他用户的影响。——隔离性
- 善于处理集中爆发的服务器使用压力；——弹性伸缩，快速扩展
- 可以很轻易的将在一个平台上运行的应用，迁移到另一个平台上，而不用担心运行环境的变化导致应用无法正常运行的情况。——迁移方便
- 使用 Docker 可以通过定制应用镜像来实现持续集成、持续交付、部署。——持续交付和部署

3. 容器 VS 虚拟机

每当说起容器，我们不得不将其与虚拟机做一个比较。就我而言，对于两者无所谓谁会取代谁，而是两者可以和谐共存。

简单来说：容器和虚拟机具有相似的资源隔离和分配优势，但功能有所不同，因为容器虚拟化的是操作系统，而不是硬件，因此容器更容易移植，效率也更高。

3.1 两者对比图

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。



3.2 容器与虚拟机总结

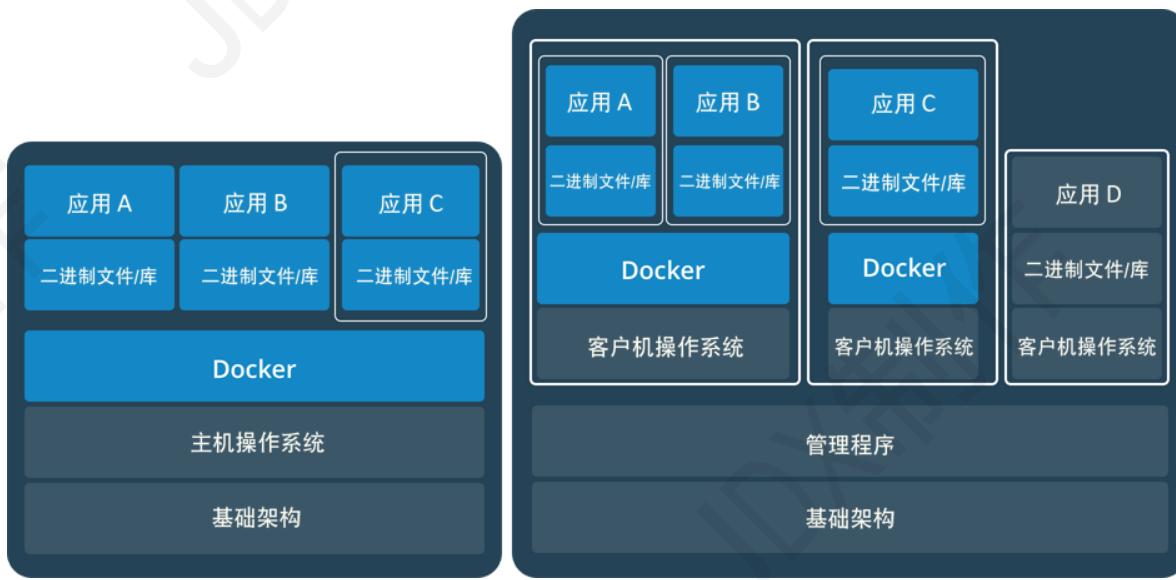
特性	容器	虚拟机
启动	秒级	分钟级
硬盘使用	一般为 MB	一般为 GB
性能	接近原生	弱于
系统支持量	单机支持上千个容器	一般几十个

- 容器是一个应用层抽象，用于将代码和依赖资源打包在一起。多个容器可以在同一台机器上运行，共享操作系统内核，但各自作为独立的进程在用户空间中运行。与虚拟机相比，容器占用的空间较少（容器镜像大小通常只有几十兆），瞬间就能完成启动。
- 虚拟机 (VM) 是一个物理硬件层抽象，用于将一台服务器变成多台服务器。管理程序允许多个 VM 在一台机器上运行。每个 VM 都包含一整套操作系统、一个或多个应用、必要的二进制文件和库资源，因此 占用大量空间。而且 VM 启动也十分缓慢。

通过 Docker 官网，我们知道了这么多 Docker 的优势，但是大家也没有必要完全否定虚拟机技术，因为两者有不同的使用场景。虚拟机更擅长于彻底隔离整个运行环境。例如，云服务提供商通常采用虚拟机技术隔离不同的用户。而 Docker 通常用于隔离不同的应用，例如前端，后端以及数据库。

3.3 容器与虚拟机两者是可以共存的

就我而言，对于两者无所谓谁会取代谁，而是两者可以和谐共存。

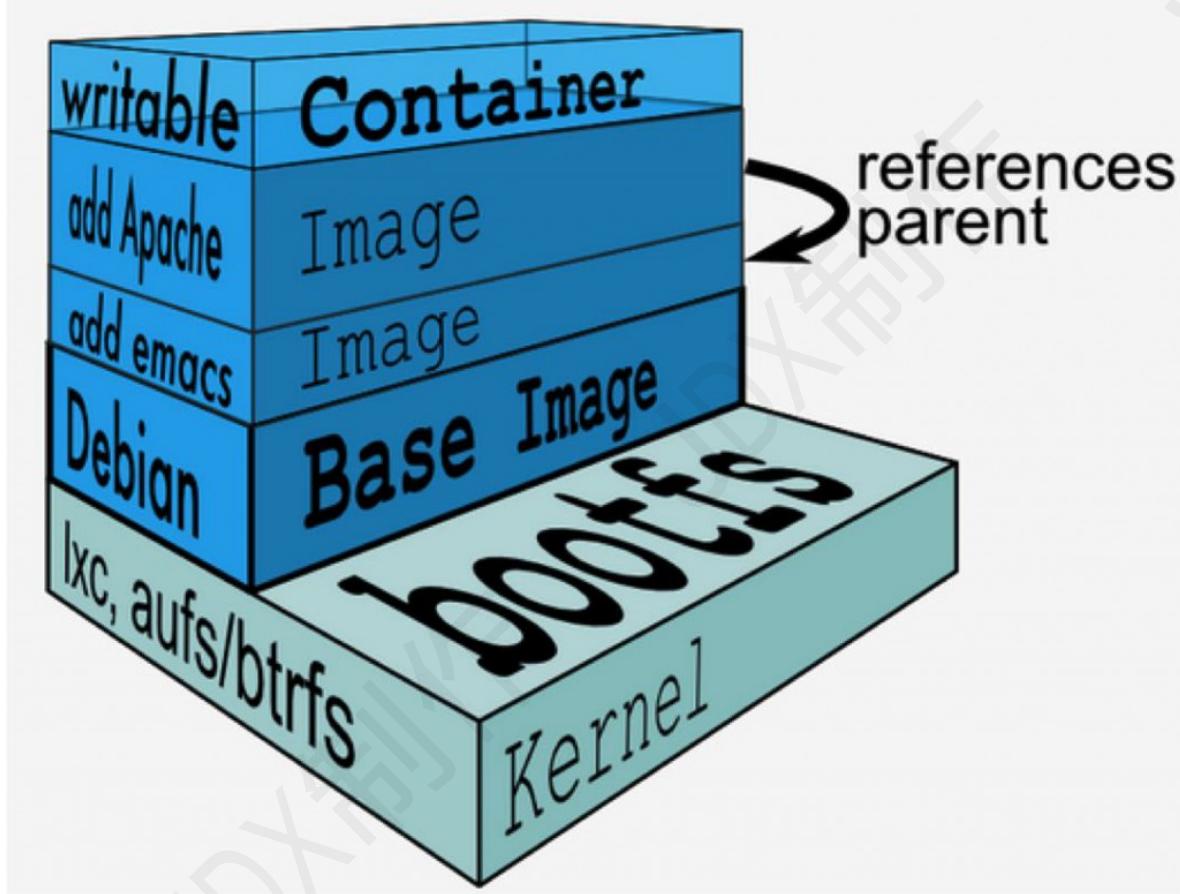


4. Docker 基本概念

Docker 中有非常重要的三个基本概念，理解了这三个概念，就理解了 Docker 的整个生命周期。

- 镜像 (Image)
- 容器 (Container)
- 仓库 (Repository)

理解了这三个概念，就理解了 Docker 的整个生命周期



4.1 镜像(Image):一个特殊的文件系统

操作系统分为内核和用户空间。对于 Linux 而言，内核启动后，会挂载 root 文件系统为其提供用户空间支持。而 Docker 镜像 (Image)，就相当于是一个 root 文件系统。

Docker 镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的一些配置参数（如匿名卷、环境变量、用户等）。镜像不包含任何动态数据，其内容在构建之后也不会被改变。

Docker 设计时，就充分利用 Union FS 的技术，将其设计为 分层存储的架构 。镜像实际是由多层文件系统联合组成。

镜像构建时，会一层层构建，前一层是后一层的基础。每一层构建完就不会再发生改变，后一层上的任何改变只发生在自己这一层。 比如，删除前一层文件的操作，实际不是真的删除前一层的文件，而是仅在当前层标记为该文件已删除。在最终容器运行的时候，虽然不会看到这个文件，但是实际上该文件会一直跟随镜像。因此，在构建镜像的时候，需要额外小心，每一层尽量只包含该层需要添加的东西，任何额外的东西应该在该层构建结束前清理掉。

分层存储的特征还使得镜像的复用、定制变的更容易。甚至可以用之前构建好的镜像作为基础层，然后进一步添加新的层，以定制自己所需的内容，构建新的镜像。

4.2 容器(Container):镜像运行时的实体

镜像 (Image) 和容器 (Container) 的关系，就像是面向对象程序设计中的 类 和 实例 一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等。

容器的实质是进程，但与直接在宿主执行的进程不同，容器进程运行于属于自己的独立的 命名空间。前面讲过镜像使用的是分层存储，容器也是如此。

容器存储层的生存周期和容器一样，容器消亡时，容器存储层也随之消亡。因此，任何保存于容器存储层的信息都会随容器删除而丢失。

按照 Docker 最佳实践的要求，容器不应该向其存储层内写入任何数据，容器存储层要保持无状态化。所有的文件写入操作，都应该使用数据卷（Volume）、或者绑定宿主目录，在这些位置的读写会跳过容器存储层，直接对宿主（或网络存储）发生读写，其性能和稳定性更高。数据卷的生存周期独立于容器，容器消亡，数据卷不会消亡。因此，使用数据卷后，容器可以随意删除、重新 run，数据却不会丢失。

4.3 仓库（Repository）：集中存放镜像文件的地方

镜像构建完成后，可以很容易的在当前宿主上运行，但是，如果需要在其它服务器上使用这个镜像，我们就需要一个集中的存储、分发镜像的服务，Docker Registry 就是这样的服务。

一个 Docker Registry 中可以包含多个仓库（Repository）；每个仓库可以包含多个标签（Tag）；每个标签对应一个镜像。所以说：镜像仓库是 Docker 用来集中存放镜像文件的地方类似于我们之前常用的代码仓库。

通常，一个仓库会包含同一个软件不同版本的镜像，而标签就常用于对应该软件的各个版本。我们可以通过 `<仓库名>:<标签>` 的格式来指定具体是这个软件哪个版本的镜像。如果不给出标签，将以 `latest` 作为默认标签。

这里补充一下 Docker Registry 公开服务和私有 Docker Registry 的概念：

Docker Registry 公开服务 是开放给用户使用、允许用户管理镜像的 Registry 服务。一般这类公开服务允许用户免费上传、下载公开的镜像，并可能提供收费服务供用户管理私有镜像。

最常使用的 Registry 公开服务是官方的 **Docker Hub**，这也是默认的 Registry，并拥有大量的高质量的官方镜像，网址为：<https://hub.docker.com/>。官方是这样介绍 Docker Hub 的：

Docker Hub 是 Docker 官方提供的一项服务，用于与您的团队查找和共享容器镜像。

比如我们想要搜索自己想要的镜像：

The screenshot shows the Docker Hub search interface. At the top, there's a search bar with 'Connected to Xian...hub' and a search term 'mysql'. Below the search bar are navigation links: Explore, Repositories, Organizations, Get Help, and a user profile icon. On the left, there are several filter categories: Docker Certified (with 'Docker Certified' checked), Images (with 'Verified Publisher' and 'Official Images' options), Categories (including Analytics, Application Frameworks, etc.), and Base Images (including MariaDB, MySQL, etc.). The main search results area displays three entries for 'mysql': 1) 'mysql' by MySQL, updated an hour ago, described as a widely used, open-source relational database management system (RDBMS). It has 10M+ downloads and 8.8K stars. 2) 'MySQL Server Enterprise Edition' by Oracle, updated a year ago, described as the world's most popular open source database system. It has 10M+ downloads and 3.1K stars. 3) 'mariadb' by MariaDB, updated an hour ago, described as a community-developed fork of MySQL intended to remain free under the GNU GPL. It has 10M+ downloads and 3.1K stars.

在 Docker Hub 的搜索结果中，有几项关键的信息有助于我们选择合适的镜像：

- **OFFICIAL Image**：代表镜像为 Docker 官方提供和维护，相对来说稳定性和安全性较高。
- **Stars**：和点赞差不多的意思，类似 GitHub 的 Star。
- **Downloads**：代表镜像被拉取的次数，基本上能够表示镜像被使用的频度。

当然，除了直接通过 Docker Hub 网站搜索镜像这种方式外，我们还可以通过 `docker search` 这个命令搜索 Docker Hub 中的镜像，搜索的结果是一致的。

→ ~ docker search mysql		
NAME	STARS	OFFICIAL
mysql	8763	[OK]
mariadb	3073	[OK]
mysql/mysql-server	650	Optimized MySQL Server Docker images. Create... [OK]

在国内访问**Docker Hub** 可能会比较慢国内也有一些云服务商提供类似于 Docker Hub 的公开服务。

除了使用公开服务外，用户还可以在 **本地搭建私有 Docker Registry**。Docker 官方提供了 Docker Registry 镜像，可以直接使用做为私有 Registry 服务。开源的 Docker Registry 镜像只提供了 Docker Registry API 的服务端实现，足以支持 docker 命令，不影响使用。但不包含图形界面，以及镜像维护、用户管理、访问控制等高级功能。

5. 常见命令

5.1 基本命令

```
docker version # 查看docker版本  
docker images # 查看所有已下载镜像, 等价于: docker image ls 命令  
docker container ls # 查看所有容器  
docker ps # 查看正在运行的容器  
docker image prune # 清理临时的、没有被使用的镜像文件。-a, --all: 删除所有没有用的镜像, 而不仅仅是临时文件;
```

5.2 拉取镜像

```
docker search mysql # 查看mysql相关镜像  
docker pull mysql:5.7 # 拉取mysql镜像  
docker image ls # 查看所有已下载镜像
```

5.3 删除镜像

比如我们要删除我们下载的 mysql 镜像。

通过 `docker rmi [image]` (等价于 `docker image rm [image]`) 删除镜像之前首先要确保这个镜像没有被容器引用 (可以通过标签名称或者镜像 ID 删除)。通过我们前面讲的 `docker ps` 命令即可查看。

→ ~ docker ps			
CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS		NAMES
c4cd691d9f80	mysql:5.7	"docker-entrypoint.s..."	7 weeks ago
Up 12 days	0.0.0.0:3306->3306/tcp, 33060/tcp		mysql

可以看到 mysql 正在被 id 为 c4cd691d9f80 的容器引用，我们需要首先通过 `docker stop c4cd691d9f80` 或者 `docker stop mysql` 暂停这个容器。

然后查看 mysql 镜像的 id

REPOSITORY	TAG	IMAGE ID	CREATED
mysql	5.7	f6509bac4980	3 months ago

通过 IMAGE ID 或者 REPOSITORY 名字即可删除

```
docker rmi f6509bac4980 # 或者 docker rm -f mysql
```

6. Build Ship and Run

Docker 的概念以及常见命令基本上已经讲完，我们再来谈谈：Build, Ship, and Run。

如果你搜索 Docker 官网，会发现如下的字样：“Docker - Build, Ship, and Run Any App, Anywhere”。那么 Build, Ship, and Run 到底是在干什么呢？



- **Build (构建镜像)**：镜像就像是集装箱包括文件以及运行环境等等资源。
- **Ship (运输镜像)**：主机和仓库间运输，这里的仓库就像是超级码头一样。
- **Run (运行镜像)**：运行的镜像就是一个容器，容器就是运行程序的地方。

Docker 运行过程也就是去仓库把镜像拉到本地，然后用一条命令把镜像运行起来变成容器。所以，我们也常常将 Docker 称为码头工人或码头装卸工，这和 Docker 的中文翻译搬运工人如出一辙。

7. 简单了解一下 Docker 底层原理

7.1 虚拟化技术

首先，Docker 容器虚拟化技术为基础的软件，那么什么是虚拟化技术呢？

简单点来说，虚拟化技术可以这样定义：

虚拟化技术是一种资源管理技术，是将计算机的各种实体资源（CPU、内存、磁盘空间、网络适配器等），予以抽象、转换后呈现出来并可供分割、组合为一个或多个电脑配置环境。由此，打破实体结构间的不可切割的障碍，使用户可以比原本的配置更好的方式来应用这些电脑硬件资源。这些资源的新虚拟部分是不受现有资源的架设方式，地域或物理配置所限制。一般所指的虚拟化资源包括计算能力和数据存储。

7.2 Docker 基于 LXC 虚拟容器技术

Docker 技术是基于 LXC (Linux container- Linux 容器) 虚拟容器技术的。

LXC，其名称来自 Linux 软件容器（Linux Containers）的缩写，一种操作系统层虚拟化（Operating system-level virtualization）技术，为 Linux 内核容器功能的一个用户空间接口。它将应用软件系统打包成一个软件容器（Container），内含应用软件本身的代码，以及所需要的的操作系统核心和库。通过统一的名字空间和共用 API 来分配不同软件容器的可用硬件资源，创造出应用程序的独立沙箱运行环境，使得 Linux 用户可以容易的创建和管理系统或应用容器。

LXC 技术主要是借助 Linux 内核中提供的 CGroup 功能和 name space 来实现的，通过 LXC 可以为软件提供一个独立的操作系统运行环境。

cgroup 和 namespace 介绍：

- **namespace 是 Linux 内核用来隔离内核资源的方式。** 通过 namespace 可以让一些进程只能看到与自己相关的一部分资源，而另外一些进程也只能看到与它们自己相关的资源，这两拨进程根本就感觉不到对方的存在。具体的实现方式是把一个或多个进程的相关资源指定在同一个 namespace 中。Linux namespaces 是对全局系统资源的一种封装隔离，使得处于不同 namespace 的进程拥有独立的全局系统资源，改变一个 namespace 中的系统资源只会影响当前 namespace 里的进程，对其他 namespace 中的进程没有影响。
- **CGroup 是 Control Groups 的缩写，是 Linux 内核提供的一种可以限制、记录、隔离进程组 (process groups) 所使用的物力资源 (如 cpu memory i/o 等等) 的机制。**

cgroup 和 namespace 两者对比：

两者都是将进程进行分组，但是两者的作用还是有本质区别。namespace 是为了隔离进程组之间的资源，而 cgroup 是为了对一组进程进行统一的资源监控和限制。

8. 总结

本文主要把 Docker 中的一些常见概念做了详细的阐述，但是并不涉及 Docker 的安装、镜像的使用、容器的操作等内容。这部分东西，希望读者自己可以通过阅读书籍与官方文档的形式掌握。如果觉得官方文档阅读起来很费力的话，这里推荐一本书籍《Docker 技术入门与实战第二版》。

八、面试指南

（一）. 程序员简历该怎么写

本篇文章除了教大家用Markdown如何写一份程序员专属的简历，后面还会给大家推荐一些不错的用来写Markdown简历的软件或者网站，以及如何优雅的将Markdown格式转变为PDF格式或者其他格式。

推荐大家使用Markdown语法写简历，然后再将Markdown格式转换为PDF格式后进行简历投递。

如果你对Markdown语法不太了解的话，可以花半个小时简单看一下Markdown语法说明: <http://www.markdown.cn>。

1. 为什么说简历很重要？

一份好的简历可以在整个申请面试以及面试过程中起到非常好的作用。在不夸大自己能力的情况下，写出一份好的简历也是一项很棒的能力。为什么说简历很重要呢？

1.1 先从面试前来说

- 假如你是网申，你的简历必然会经过HR的筛选，一张简历HR可能也就花费10秒钟看一下，然后HR就会决定你这一关是Fail还是Pass。
- 假如你是内推，如果你的简历没有什么优势的话，就算是内推你的人再用心，也无能为力。

另外，就算你通过了筛选，后面的面试中，面试官也会根据你的简历来判断你究竟是否值得他花费很多时间去面试。

获取更多资源可添加关注微信公众号：JAVA架构进阶之路；或添加微信：jiang10086a 免费获取面试真题，视频教程，笔记等。

所以，简历就像是我们的一个门面一样，它在很大程度上决定了你能否进入到下一轮的面试中。

1.2 再从面试中来说

我发现大家比较喜欢看面经，这点无可厚非，但是大部分面经都没告诉你很多问题都是在特定条件下才问的。举个简单的例子：一般情况下你的简历上注明你会的东西才会被问到（Java、数据结构、网络、算法这些基础是每个人必问的），比如写了你会 redis，那面试官就很大概率会问你 redis 的一些问题。比如：redis 的常见数据类型及应用场景、redis 是单线程为什么还这么快、redis 和 memcached 的区别、redis 内存淘汰机制等等。

所以，首先，你要明确的一点是：**你不会的东西就不要写在简历上**。另外，**你要考虑你该如何才能让你的亮点在简历中凸显出来**，比如：你在某某项目做了什么事情解决了什么问题（只要有项目就一定有要解决的问题）、你的某一个项目里使用了什么技术后整体性能和并发量提升了很多等等。

面试和工作是两回事，聪明的人会把面试官往自己擅长的领域领，其他人则被面试官牵着鼻子走。虽说面试和工作是两回事，但是你要想要获得自己满意的 offer，你自身的实力必须要强。

2. 下面这几点你必须知道

1. 大部分公司的HR都说我们不看重学历（骗你的！），但是如果你的学校不出众的话，很难在一堆简历中脱颖而出，除非你的简历上有特别的亮点，比如：某某大厂的实习经历、获得了某某大赛的奖等等。
2. **大部分应届生找工作的硬伤是没有工作经验或实习经历，所以如果你是应届生就不要错过秋招和春招。一旦错过，你后面就极大可能会面临社招，这个时候没有工作经验的你可能就会面临各种碰壁，导致找不到一个好的工作**
3. **写在简历上的东西一定要慎重，这是面试官大量提问的地方；**
4. **将自己的项目经历完美的展示出来非常重要。**

3. 必须了解的两大法则

3.1 STAR法则 (Situation Task Action Result)

- **Situation:** 事情是在什么情况下发生；
- **Task:** 你是如何明确你的任务的；
- **Action:** 针对这样的情况分析，你采用了什么行动方式；
- **Result:** 结果怎样，在这样的情况下你学习到了什么。

简而言之，STAR法则，就是一种讲述自己故事的方式，或者说，是一个清晰、条理的作文模板。不管是什 么，合理熟练运用此法则，可以轻松的对面试官描述事物的逻辑方式，表现出自己分析阐述问题的清晰性、条理性和逻辑性。

3.2 FAB 法则 (Feature Advantage Benefit)

- **Feature:** 是什么；
- **Advantage:** 比别人好在哪些地方；
- **Benefit:** 如果雇佣你，招聘方会得到什么好处。

简单来说，这个法则主要是让你的面试官知道你的优势、招了你之后对公司有什么帮助。

4. 项目经历怎么写？

简历上有一两个项目经历很正常，但是真正能把项目经历很好的展示给面试官的非常少。对于项目经历大家可以考虑从如下几点来写：

1. 对项目整体设计的一个感受
2. 在这个项目中你负责了什么、做了什么、担任了什么角色
3. 从这个项目中你学会了那些东西，使用到了那些技术，学会了那些新技术的使用

4. 另外项目描述中，最好可以体现自己的综合素质，比如你是如何协调项目组成员协同开发的或者在遇到某一个棘手的问题的时候你是如何解决的又或者说你在这个项目用了什么技术实现了什么功能比如用redis做缓存提高访问速度和并发量、使用消息队列削峰和降流等等。

5. 专业技能该怎么写？

先问一下你自己会什么，然后看看你意向的公司需要什么。一般HR可能并不太懂技术，所以他在筛选简历的时候可能就盯着你专业技能的关键词来看。对于公司有要求而你不会的技能，你可以花几天时间学习一下，然后在简历上可以写上自己了解这个技能。比如你可以这样写(下面这部分内容摘自我的简历，大家可以根据自己的情况做一些修改和完善)：

- 计算机网络、数据结构、算法、操作系统等课内基础知识：掌握
- Java 基础知识：掌握
- JVM 虚拟机 (Java内存区域、虚拟机垃圾算法、虚拟垃圾收集器、JVM内存管理)：掌握
- 高并发、高可用、高性能系统开发：掌握
- Struts2、Spring、Hibernate、Ajax、Mybatis、JQuery：掌握
- SSH 整合、SSM 整合、SOA 架构：掌握
- Dubbo：掌握
- Zookeeper：掌握
- 常见消息队列：掌握
- Linux：掌握
- MySQL常见优化手段：掌握
- Spring Boot +Spring Cloud +Docker：了解
- Hadoop 生态相关技术中的 HDFS、Storm、MapReduce、Hive、Hbase：了解
- Python 基础、一些常见第三方库比如OpenCV、wxpy、wordcloud、matplotlib：熟悉

6. 排版注意事项

1. 尽量简洁，不要太花里胡哨；
2. 一些技术名词不要弄错了大小写比如MySQL不要写成mysql，Java不要写成java。这个在我看来还是比较忌讳的，所以一定要注意这个细节；
3. 中文和数字英文之间加上空格的话看起来会舒服一点；

7. 其他的一些小tips

1. 尽量避免主观表述，少一点语义模糊的形容词，尽量要简洁明了，逻辑结构清晰。
2. 如果自己有博客或者个人技术栈点的话，写上去会为你加分很多。
3. 如果自己的Github比较活跃的话，写上去也会为你加分很多。
4. 注意简历真实性，一定不要写自己不会的东西，或者带有欺骗性的内容
5. 项目经历建议以时间倒序排序，另外项目经历不在于多，而在于有亮点。
6. 如果内容过多的话，不需要非把内容压缩到一页，保持排版干净整洁就可以了。
7. 简历最后最好能加上：“感谢您花时间阅读我的简历，期待能有机会和您共事。”这句话，显的你会很有礼貌。

(二) . 如何准备面试

不论是笔试还是面试都是有章可循的，但是，一定要不要想着如何去应付面试，糊弄面试官，这样做终究是欺骗自己。这篇文章的目的也主要想让大家知道自己应该从哪些方向去准备面试，有哪些可以提高的方向。

网上已经有很多面经了，但是我认为网上的各种面经仅仅只能作为参考，你的实际面试与之还是有一些区别的。另外如果要在网上看别人的面经的话，建议即要看别人成功的案例也要适当看看别人失败的案例。**看面经没问题，不论是你要找工作还是平时学习，这都是一种比较好地检验自己水平的一种方式。但是，一定不要过分寄希望于各种面经，试着去提高自己的综合能力。**

“80%的offer掌握在20%的人手”中这句话也不是不无道理的。决定你面试能否成功的因素中实力固然占有很大一部分比例，但是如果你的心态或者说运气不好的话，依然无法拿到满意的offer。

运气暂且不谈，就拿心态来说，千万不要因为面试失败而气馁或者说怀疑自己的能力，面试失败之后多总结一下失败的原因，后面你就会发现自己会越来越强大。

另外，笔者只是在这里分享一下自己对于“如何备战大厂面试”的一个看法，以下大部分理论/言辞都经过过反复推敲验证，如果有不对的地方或者和你想法不同的地方，请您敬请雅正、不舍赐教。

1. 如何获取大厂面试机会？

在讲如何获取大厂面试机会之前，先来给大家科普/对比一下两个校招非常常见的概念——春招和秋招。

1. 招聘人数：秋招多于春招；
2. 招聘时间：秋招一般7月左右开始，大概一直持续到10月底。**但是大厂（如BAT）都会早开始早结束，所以一定要把握好时间。**春招最佳时间为3月，次佳时间为4月，进入5月基本就不会再有春招了（金三银四）。
3. 应聘难度：秋招略大于春招；
4. 招聘公司：秋招数量多，而春招数量较少，一般为秋招的补充。

综上，一般来说，秋招的含金量明显是高于春招的。

下面我就说一下我自己知道的一些方法，不过应该也涵盖了大部分获取面试机会的方法。

1. 关注大厂官网，随时投递简历（走流程的网申）；
2. 线下参加宣讲会，直接投递简历；
3. 找到师兄师姐/认识的人，帮忙内推（能够让你避开网申简历筛选，笔试筛选，还是挺不错的，不过也还是需要你的简历够棒）；
4. 博客发文被看中/Github优秀开源项目作者，大厂内部人员邀请你面试；
5. 求职类网站投递简历（不是太推荐，适合海投）；

除了这些方法，我也遇到过这样的经历：有些大公司的一些部门可能暂时没招够人，然后如果你的亲戚或者朋友刚好在这个公司，而你正好又在寻求offer，那么面试机会基本上是有了，而且这种面试的难度好像一般还普遍比其他正规面试低很多。

2. 面试前的准备

2.1 准备自己的自我介绍

自我介绍一般是你和面试官的第一次面对面正式交流，换位思考一下，假如你是面试官的话，你想听到被你面试的人如何介绍自己呢？一定不是客套地说说自己喜欢编程、平时花了很多时间来学习、自己的兴趣爱好是打球吧？

我觉得一个好的自我介绍应该包含这几点要素：

1. 用简单的话说清楚自己主要的技术栈于擅长的领域；
2. 把重点放在自己在行的地方以及自己的优势之处；
3. 重点突出自己的能力比如自己的定位的bug的能力特别厉害；

从社招和校招两个角度来举例子吧！我下面的两个例子仅供参考，自我介绍并不需要死记硬背，记住要说的要点，面试的时候根据公司的情况临场发挥也是没问题的。另外，网上一般建议的是准备好两份自我介绍：一份对hr说的，主要讲能突出自己的经历，会的编程技术一语带过；另一份对技术面试官说的，主要讲自己会的技术细节和项目经验。

社招：

面试官，您好！我叫独秀儿。我目前有1年半的工作经验，熟练使用Spring、MyBatis等框架、了解Java底层原理比如JVM调优并且有着丰富的分布式开发经验。离开上一家公司是因为我想在技术上得到更多的锻炼。在上一个公司我参与了一个分布式电子交易系统的开发，负责搭建了整个项目的基础架构并且通过分库分表解决了原始数据库以及一些相关表过于庞大的问题，目前这个网站最高支持10万人同时访问。工作之余，我利用自己的业余时间写了一个简单的RPC框架，这个框架用到了Netty进行网络通信，目前我已经将这个项目开源，在Github上收获了2k的Star！说到业余爱好的话，我比较喜欢通过博客整理分享自己所学知识，现在已经是多个博客平台的认证作者。生活中我是一个比较积极乐观的人，一般会通过运动打球的方式来放松。我一直都非常想加入贵公司，我觉得贵公司的文化和技术氛围我都非常喜欢，期待能与你共事！

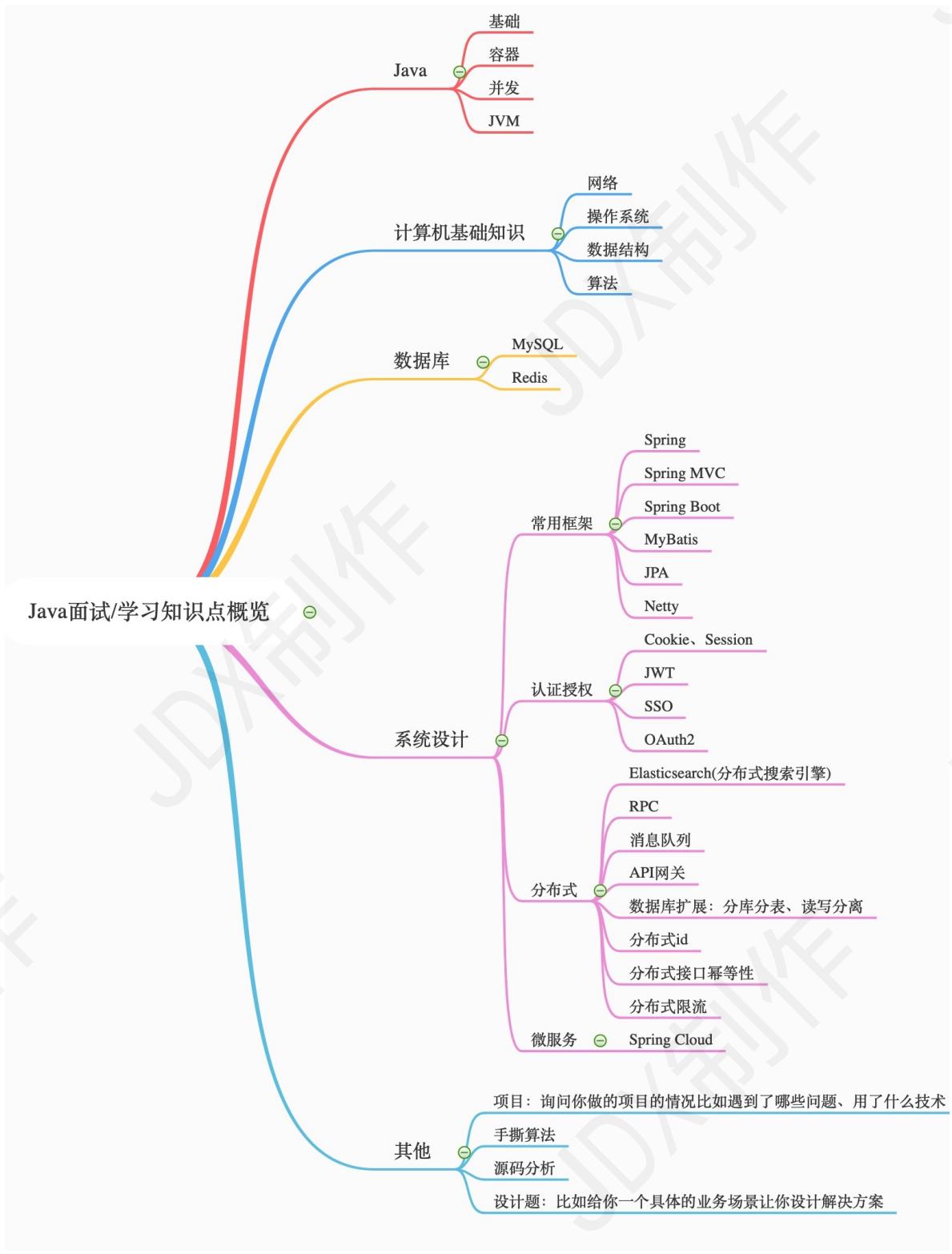
校招：

面试官，您好！我叫秀儿。大学时间我主要利用课外时间学习了Java以及Spring、MyBatis等框架。在校期间参与过一个考试系统的开发，这个系统的主要用了Spring、MyBatis和shiro这三种框架。我在其中主要担任后端开发，主要负责了权限管理功能模块的搭建。另外，我在大学的时候参加过一次软件编程大赛，我和我的团队做的在线订餐系统成功获得了第二名的成绩。我还利用自己的业余时间写了一个简单的RPC框架，这个框架用到了Netty进行网络通信，目前我已经将这个项目开源，在Github上收获了2k的Star！说到业余爱好的话，我比较喜欢通过博客整理分享自己所学知识，现在已经是多个博客平台的认证作者。生活中我是一个比较积极乐观的人，一般会通过运动打球的方式来放松。我一直都非常想加入贵公司，我觉得贵公司的文化和技术氛围我都非常喜欢，期待能与你共事！

2.2 搞清楚技术面可能会问哪些方向的问题

你准备面试的话首先要搞清技术面可能会被问哪些方向的问题吧！

我直接用思维导图的形式展示出来吧！这样更加直观形象一点，细化到某个知识点的话这张图没有介绍到，留个悬念，下篇文章会详细介绍。



上面思维导图大概涵盖了技术面试可能会设计的技术，但是你不需要把上面的每一个知识点都搞得很熟悉，要分清主次，对于自己不熟悉的技不要写在简历上，对于自己简单了解的技术不要说自己熟练掌握！

2.3 休闲着装即可

穿西装、打领带、小皮鞋？NO! NO! NO! 这是互联网公司面试又不是去走红毯，所以你只需要穿的简单大方就好，不需要太正式。

2.4 随身带上自己的成绩单和简历

有的公司在面试前都会让你交一份成绩单和简历当做面试中的参考。

2.5 如果需要笔试就提前刷一些笔试题

平时空闲时间多的可以刷一下笔试题目（牛客网上有很多）。但是不要只刷面试题，不动手code，程序员不是为了考试而存在的。

2.6 花时间一些逻辑题

面试中发现有些公司都有逻辑题测试环节，并且都把逻辑笔试成绩作为很重要的一个参考。

2.7 准备好自己的项目介绍

如果有项目的话，技术面试第一步，面试官一般都是让你自己介绍一下你的项目。你可以从以下几个方向来考虑：

1. 对项目整体设计的一个感受（面试官可能会让你画系统的架构图）
2. 在这个项目中你负责了什么、做了什么、担任了什么角色
3. 从这个项目中你学会了那些东西，使用到了那些技术，学会了那些新技术的使用
4. 另外项目描述中，最好可以体现自己的综合素质，比如你是如何协调项目组成员协同开发的或者在遇到某一个棘手的问题的时候你是如何解决的又或者说你在这个项目用了什么技术实现了什么功能比如：用redis做缓存提高访问速度和并发量、使用消息队列削峰和降流等等。

2.8 提前准备技术面试

搞清楚自己面试中可能涉及哪些知识点、哪些知识点是重点。面试中哪些问题会被经常问到、自己该如何回答。（强烈不推荐背题，第一：通过背这种方式你能记住多少？能记住多久？第二：背题的方式的学习很难坚持下去！）

2.9 面试之前做好定向复习

所谓定向复习就是专门针对你要面试的公司来复习。比如你在面试之前可以在网上找找有没有你要面试的公司的面经。

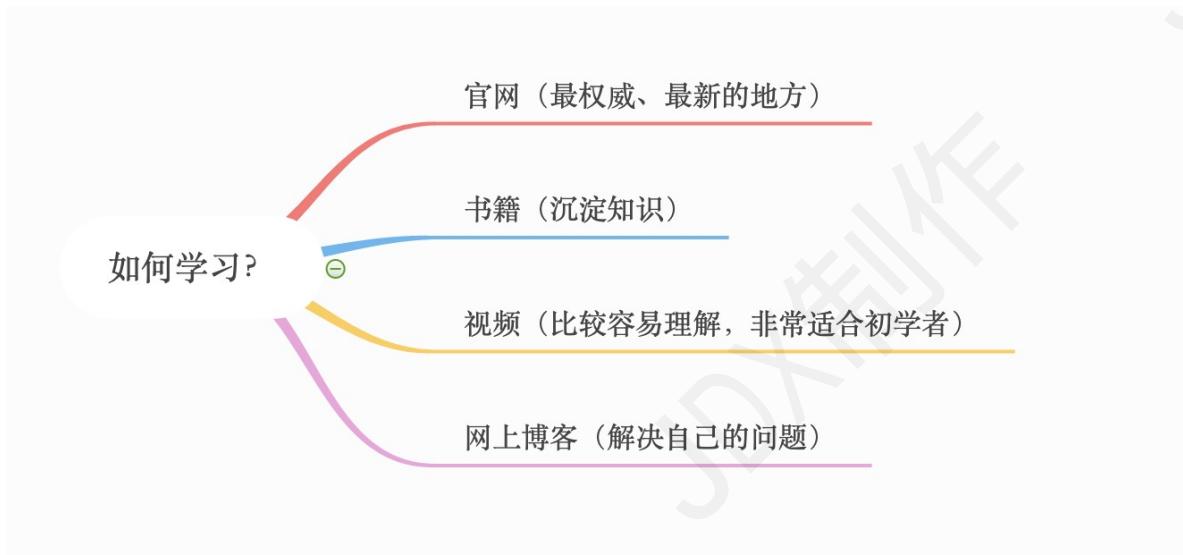
举个栗子：在我面试 ThoughtWorks 的前几天我就在网上找了一些关于 ThoughtWorks 的技术面的一些文章。然后知道了 ThoughtWorks 的技术面会让我们在之前做的作业的基础上增加一个或两个功能，所以我提前一天就把我之前做的程序重新重构了一下。然后在技术面的时候，简单的改了几行代码之后写个测试就完事了。如果没有提前准备，我觉得 20 分钟我很大几率会完不成这项任务。

3. 面试之后复盘

如果失败，不要灰心；如果通过，切勿狂喜。面试和工作实际上是两回事，可能很多面试未通过的人，工作能力比你强的多，反之亦然。我个人觉得面试也像是一场全新的征程，失败和胜利都是平常之事。所以，劝各位不要因为面试失败而灰心、丧失斗志。也不要因为面试通过而沾沾自喜，等待你的将是更美好的未来，继续加油！

4. 如何学习？学会各种框架有必要吗？

4.1 我该如何学习？



最最最关键也是对自己最最最重要的就是学习！看看别人分享的面经，看看我写的这篇文章估计你只需要10分钟不到。但这些东西终究是空洞的理论，最主要的还是自己平时的学习！

如何去学呢？我觉得学习每个知识点可以考虑这样去入手：

1. 官网（大概率是英文，不推荐初学者看）。
2. 书籍（知识更加系统完全，推荐）。
3. 视频（比较容易理解，推荐，特别是初学的时候。慕课网和哔哩哔哩上面有挺多学习视频可以看，只直接在上面搜索关键词就可以了）。
4. 网上博客（解决某一知识点的问题的时候可以看看）。

这里给各位一个建议，看视频的过程中最好跟着一起练，要做笔记！！！

最好可以边看视频边找一本书籍看，看视频没弄懂的知识点一定要尽快解决，如何解决？

首先百度/Google，通过搜索引擎解决不了的话就找身边的朋友或者认识的一些人。

4.2 学会各种框架有必要吗？

一定要学会分配自己时间，要学的东西很多，真的很多，搞清楚哪些东西是重点，哪些东西仅仅了解就够了。一定不要把精力都花在了学各种框架上，算法、数据结构还有计算机网络真的很重要！

(三) . Java学习路线和方法推荐

下面的学习路线以及方法是笔主根据个人学习经历总结改进后得出，我相信照着这条学习路线来你的学习效率会非常高。

另外，很重要的一点：建议使用 IntelliJ IDEA 进行编码，可以单独抽点时间学习 IntelliJ IDEA 的使用。

先说一个初学者很容易犯的错误：上来就通过项目学习。

很多初学者上来就像通过做项目学习，特别是在公司，我觉得这个是不太可取的。如果的 Java基础或者 Spring Boot 基础不好的话，建议自己先提前学习一下之后再开始看视频或者通过其他方式做项目。还有点事，我不知道为什么大家都会说边跟着项目边学习做的话效果最好，我觉得这个要加一个前提是对你这门技术有基本的了解或者说你对编程有了一定的了解。

关于如何学习且听我从一个电商系统网站的创建来说起。假如我们要创建一个基于Java的分布式/微服务电商系统的话，我们可以按照下面的学习路线来做：

首选第一步我们肯定是要从 Java 基础来学习的（如果你没有计算机基础知识的话推荐看一下《计算机导论》这类入门书籍）。

获取更多资源可添加关注微信公众号：JAVA架构进阶之路；或添加微信：jiang10086a 免费获取面试真题，视频教程，笔记等。

1. Java 基础

《Java 核心技术卷 1/2》和《Head First Java》这两本书在我看来都是入门 Java 的很不错的书籍（《Java 核心技术卷 1/2》知识点更全，我更推荐这本书），我倒是觉得《Java 编程思想》有点属于新手劝退书的意思，慎看，建议有点基础后再看。你也可以边看视频边看书学习（黑马、尚硅谷、慕课网的视频都还行）。对于 Java8 新特性的东西，我建议你基础学好之后可以看一下，暂时看不太明白也没关系，后面抽时间再回过头来看。

看完之后，你可以用自己学的东西实现一个简单的 Java 程序，也可以尝试用 Java 解决一些编程问题，以此来将自己学到的东西付诸于实践。

不太建议学习 Java 基础的时候通过做游戏来巩固。为什么培训班喜欢通过这种方式呢？说白点就是为了找到你的 G 点（不好意思开车了哈）。新手学习完 Java 基础后做游戏一般是不太现实的，还不如找一些简单的程序问题解决一下比如简单的算法题。

记得多总结！打好基础！把自己重要的东西都记录下来。 API 文档放在自己可以看到的地方，以备自己可以随时查阅。为了能让自己写出更优秀的代码，《Effective Java》、《重构》这两本书没事也可以看看。

另外，学习完之后可以看一下下面这几篇文章，检查一下自己的学习情况。这几篇文章不是我吹，可能是全网最具价值的 Java 基础知识总结，毕竟是在我的 JavaGuide 开源的，经过了各路大佬以及我的不断完善。

2. 操作系统与计算机网络

操作系统这方面我觉得掌握操作系统的基础知识和 Linux 的常用命令就行以及一些重要概念就行了。

关于操作系统的话，我没有什么操作系统方面的书籍可以推荐，因为我自己也没认真看过几本。因为操作系统比较枯燥的原因，我建议这部分看先看视频学可能会比较好一点。

另外，对于 Linux 我们要掌握基本的使用就需要对一些常用命令非常熟悉比如：目录切换命令、目录操作命令、文件的操作命令、压缩或者解压文件的命令等等。

计算机网络方面的学习，我觉得掌握基本的知识就行了，不要太深究，一般面试对这方面要求也不高，毕竟不是专门做网络的。推荐《网络是怎样连接的》、《图解 HTTP》这两本书来看，这两本书都属于比较有趣易懂的类型，也适合没有基础的人来看。

我们写程序的都知道一个公式叫做“**程序设计 = 算法 + 数据结构**”。我们想让我们的网站的地盘更加牢固的话，我觉得数据结构与算法还是很有必要学习的。所以第三步，我推荐可以适当花时间看一下 **数据结构与算法** 但是，同样不做强求！你抽时间一定要补上就行！

3. 数据结构与算法

如果你想进入大厂的话，我推荐你在学习完 Java 基础之后，就开始每天抽出一点时间来学习算法和数据结构。为了提高自己的编程能力，你也可以坚持刷 Leetcode。就目前国内外的大厂面试来说，刷 Leetcode 可以说已经成了不得不走的一条路。

对于想要入门算法和数据结构的朋友，建议看这两本书《算法图解》和《大话数据结构》，这两本书虽然算不上很经典的书籍，但是比较有趣，对于刚入门算法和数据结构的朋友非常友好。《算法导论》非常经典，但是对于刚入门的就不那么友好了。

另外，还有一本非常赞的算法书推荐给各位，这本书的名字就叫《算法》，书中的代码都是用 Java 语言编写。这本书的优点太多太多比如它的讲解基础而全面、对阅读者比较友好等等。我觉得这本书唯一的缺点就是太厚了（小声 BB，可能和作者讲解某些知识点的时候有点啰嗦有关）。除了这本书之外，《剑指 offer》、《编程珠玑》、《编程之美》这三本书都被很多大佬推荐过了，对于算法面试非常有帮助。《算法之美》这本书也非常不错，非常适合闲暇的时候看。

我们网站的页面搭建需要前端的知识，我们前端也后端的交互也需要前端的知识。所以第四步，我推荐你可以了解一下前端知识，不过不需要学的太精通。自己对与前端知识有了基本的了解之后通过

4. 前端知识

这一步主要是学习前端基础 (HTML、CSS、JavaScript),当然 BootStrap、Layui 等等比较简单的前端框架你也可以了解一下。网上有很多这方面资源。

另外，没记错的话 Spring Boot官方推荐的是模板引擎是 thymeleaf，这东西和HTML很像，了解了基本语法之后很容易上手。结合layui,booystrap这些框架的话也能做成比较美观的页面。开发一些简单的页面比如一个后端项目就是为了做个简单的前端页面做某些操作的话直接用thymeleaf就好。

现在都是前后端分离，就目前来看大部分项目都优先选择 React、Angular、Vue 这些厉害的框架来开发，这些框架的上手要求要高一些。如果你想往全栈方向发展的话（笔主目前的方向，我用 React 在公司做过两个小型项目），建议先把 JS 基础打好，然后再选择 React、Angular、Vue 其中的一个来认真学习一下。国内使用 Vue 比较多一点，国外一般用的是 React 和 Angular。

如何和后端交互呢？一般使用在 React、Vue这些框架的时候使用Axios比较多。

我们网站的数据比如用户信息、订单信息都需要存储，所以，下一步我推荐你学习 MySQL这个被广泛运用于各大网站的数据库。不光要学会如何写 sql 语句，更好的是还要搞清诸如索引这类重要的概念。

5. MySQL

学习 MySQL 的基本使用，基本的增删改查，SQL 命令，索引、存储过程这些都学一下吧！推荐书籍《SQL 基础教程 (第 2 版) 》（入门级）、《高性能 MySQL：第 3 版》(进阶)、《MySQL 必知必会》。

6. 常用工具

非常重要！非常重要！特别是 Git和 Docker。

1. **IDEA**：熟悉基本操作以及常用快捷。
2. **Maven**：建议学习常用框架之前可以提前花半天时间学习一下**Maven**的使用。（到处找 Jar 包，下载 Jar 包是真的麻烦费事，使用 Maven 可以为你省很多事情）。
3. **Git**：基本的 Git 技能也是必备的，试着在学习的过程中将自己的代码托管在 Github 上。
4. **Docker**：学着用 Docker 安装学习中需要用到的软件比如 MySQL,这样方便很多，可以为你节省不少时间。

利用常用框架可以极大程度简化我们的开发工作。学习完了常用工具之后，我们就可以开始常用框架的学习啦！

7. 常用框架

学习 Struts2(可不用学)、Spring、SpringMVC、Hibernate、Mybatis、shiro 等框架的使用，(可选)熟悉 Spring 原理（大厂面试必备），然后很有必要学习一下 SpringBoot，学好 SpringBoot 真的很重要。很多公司对于应届生都是直接上手 SpringBoot，不过如果时间允许的话，我还是推荐你把 Spring、SpringMVC 提前学一下。

Spring 真的很重要！一定要搞懂 AOP 和 IOC 这两个概念。Spring 中 bean 的作用域与生命周期、SpringMVC 工作原理详解等等知识点都是非常重要的，一定要搞懂。

8. 多线程的简单使用

多线程这部分内容可能会比较难以理解和上手，前期可以先简单地了解一下基础，到了后面有精力和能力后再回来仔细看。

学习完多线程之后可以通过下面这些问题检测自己是否掌握。

Java 多线程知识基础:

1. 什么是线程和进程?
2. 请简要描述线程与进程的关系,区别及优缺点?
3. 说说并发与并行的区别?
4. 为什么要使用多线程呢?
5. 使用多线程可能带来什么问题?
6. 说说线程的生命周期和状态?
7. 什么是上下文切换?
8. 什么是线程死锁?如何避免死锁?
9. 说说 sleep() 方法和 wait() 方法区别和共同点?
10. 为什么我们调用 start() 方法时会执行 run() 方法, 为什么我们不能直接调用 run() 方法?

Java 多线程知识进阶:

1. synchronized 关键字:① 说一说自己对于 synchronized 关键字的了解; ② 说说自己是怎么使用 synchronized 关键字, 在项目中用到了吗;③ 讲一下 synchronized 关键字的底层原理; ④ 说说 JDK1.6 之后的 synchronized 关键字底层做了哪些优化, 可以详细介绍一下这些优化吗; ⑤ 谈谈 synchronized 和 ReentrantLock 的区别。
2. volatile 关键字: ① 讲一下 Java 内存模型; ② 说说 synchronized 关键字和 volatile 关键字的区别。
3. ThreadLocal: ① 简介; ② 原理; ③ 内存泄露问题。
4. 线程池: ① 为什么要用线程池? ; ② 实现 Runnable 接口和 Callable 接口的区别; ③ 执行 execute() 方法和 submit() 方法的区别是什么呢? ; ④ 如何创建线程池。
5. Atomic 原子类: ① 介绍一下 Atomic 原子类; ② JUC 包中的原子类是哪 4 类?; ③ 讲讲 AtomicInteger 的使用; ④ 能不能给我简单介绍一下 AtomicInteger 类的原理。
6. AQS : ① 简介; ② 原理; ③ AQS 常用组件。

9. 分布式

1. 学习 Dubbo、Zookeeper 来实现简单的分布式服务
2. 学习 Redis 来提高访问速度, 减少对 MySQL 数据库的依赖;
3. 学习 Elasticsearch 的使用, 来为我们的网站增加搜索功能
4. 学习常见的消息队列 (比如 RabbitMQ、Kafka) 来解耦我们的服务(ActiveMq 不要学了, 已经淘汰);
5.

到了这一步你应该是有基础的一个 Java 程序员了, 我推荐你可以通过一个分布式项目来学习。觉得应该是掌握这些知识点比较好的一种方式了, 另外, 推荐边看视频边自己做, 遇到不懂的知识点要及时查阅网上博客和相关书籍, 这样学习效果更好。

一定要学会拓展知识, 养成自主学习的意识。 黑马项目对这些知识点的介绍都比较蜻蜓点水。

| 继续深入学习的话, 我们要了解 Netty、JVM 这些东西。

10. 深入学习

可以再回来看一下多线程方面的知识, 还可以利用业余时间学习一下 NIO 和 Netty, 这样简历上也可以多点东西。如果想去大厂, JVM 的一些知识也是必学的 (Java 内存区域、虚拟机垃圾算法、虚拟垃圾收集器、JVM 内存管理) 推荐《深入理解 Java 虚拟机: JVM 高级特性与最佳实践 (最新第二版)》和《实战 Java 虚拟机》, 如果嫌看书麻烦的话, 你也可以看我整理的文档。

另外，现在微服务特别火，很多公司在面试也明确要求需要微服务方面的知识。如果有精力的话可以去学一下 SpringCloud 生态系统微服务方面的东西。

微服务的概念庞大，技术种类也很多，但是目前大型互联网公司广泛采用的，实话实话这些东西我不在行，自己没有真实做过微服务的项目。不过下面是我自己总结的一些关于微服务比价重要的知识，选学。

11. 微服务

这部分太多了，选择性学习。

相关技术：

1. 网关 :kong,soul;
2. 分布式调用链： SkyWalking、 Zipkin
3. 日志系统： Kibana
4.

Spring Cloud 相关：

1. Eureka：服务注册与发现；
2. Ribbon：负载均衡；
3. Hytrix：熔断；
4. Zuul：网关；
5. Spring Cloud Config：配置中心；

Spring Cloud Alibaba也是很值得学习的：

1. **Sentinel**：A lightweight powerful flow control component enabling reliability and monitoring for microservices. (轻量级的流量控制、熔断降级 Java 库)。
2. **dubbo**：Apache Dubbo 是一个基于 Java 的高性能开源 RPC 框架。
3. **nacos**：Nacos 致力于帮助您发现、配置和管理微服务。Nacos 提供了一组简单易用的特性集，帮助您快速实现动态服务发现、服务配置、服务元数据及流量管理。Nacos 可以作为 Dubbo 的注册中心来使用。
4. **seata**：Seata 是一种易于使用，高性能，基于 Java 的开源分布式事务解决方案。
5. **RocketMQ**：阿里巴巴开源的一款高性能、高吞吐量的分布式消息中间件。

12. 总结

我上面主要概括一下每一步要学习的内容，对学习规划有一个建议。知道要学什么之后，如何去学呢？我觉得学习每个知识点可以考虑这样去入手：

1. 官网（大概率是英文，不推荐初学者看）。
2. 书籍（知识更加系统完全，推荐）。
3. 视频（比较容易理解，推荐，特别是初学的时候）。
4. 网上博客（解决某一知识点的问题的时候可以看看）。

这里给各位一个建议，看视频的过程中最好跟着一起练，要做笔记！！！

最好可以边看视频边找一本书籍看，看视频没弄懂的知识点一定要尽快解决，如何解决？

首先百度/Google，通过搜索引擎解决不了的话就找身边的朋友或者认识的一些人。另外，一定要进行项目实战！很多人这时候就会问没有实际项目让我做怎么办？我觉得可以通过下面几种方式：

1. 在网上找一个符合自己能力与找工作需求的实战项目视频或者博客跟着老师一起做。做的过程中，你要有自己的思考，不要浅尝辄止，对于很多知识点，别人的讲解可能只是满足项目就够了，你自己想多点知识的话，对于重要的知识点就要自己学会去往深处学。

2. Github 或者码云上面有很多实战类别项目，你可以选择一个来研究，为了让自己对这个项目更加理解，在理解原有代码的基础上，你可以对原有项目进行改进或者增加功能。
3. 自己动手去做一个自己想完成的东西，遇到不会的东西就临时去学，现学现卖(这种方式比较难，初学不推荐用这种方式，因为你脑海中没有基本的概念，写出来的代码一般会很难或者根本就做不出来一个像样的东西)。
4.

做项目不光要做，还要改进，改善。另外，如果你的老师有相关 Java 后台项目的话，你也可以主动申请参与进来。

一定要学会分配自己时间，要学的东西很多，真的很多，搞清楚哪些东西是重点，哪些东西仅仅了解就够了。一定不要把精力都花在了学各种框架上，算法和数据结构真的很重要！