尚硅谷Java高频面试闯关秘籍V12.1

(尚硅谷Java教研组)

**目录**

[**第一章 Java基础篇 6**](#_Toc24251)

[1、你是怎样理解OOP面向对象 难度系数：⭐ 6](#_Toc10742)

[2、 重载与重写区别 难度系数：⭐ 6](#_Toc6830)

[3、 接口与抽象类的区别 难度系数：⭐ 6](#_Toc2940)

[4、深拷贝与浅拷贝的理解 难度系数：⭐ 7](#_Toc416)

[5、sleep和wait区别 难度系数：⭐ 7](#_Toc20792)

[6、什么是自动拆装箱 int和Integer有什么区别 难度系数：⭐ 9](#_Toc26967)

[7、==和equals区别 难度系数：⭐ 10](#_Toc30066)

[8、String能被继承吗 为什么用final修饰 难度系数：⭐ 10](#_Toc22575)

[9、String buffer和String builder区别 难度系数：⭐ 10](#_Toc19841)

[10、final、finally、finalize 难度系数：⭐ 11](#_Toc15582)

[11、Object中有哪些方法 难度系数：⭐ 11](#_Toc18681)

[12、说一下集合体系 难度系数：⭐ 12](#_Toc32127)

[13、ArrarList和LinkedList区别 难度系数：⭐ 12](#_Toc24346)

[14、HashMap底层是 数组+链表+红黑树，为什么要用这几类结构 难度系数：⭐⭐ 13](#_Toc24500)

[15、HashMap和HashTable区别 难度系数：⭐ 13](#_Toc17619)

[16、线程的创建方式 难度系数：⭐ 14](#_Toc6947)

[17、线程的状态转换有什么（生命周期） 难度系数：⭐ 16](#_Toc6109)

[18、Java中有几种类型的流 难度系数：⭐ 18](#_Toc12446)

[19、请写出你最常见的5个RuntimeException 难度系数：⭐ 19](#_Toc20413)

[20、谈谈你对反射的理解 难度系数：⭐ 19](#_Toc13228)

[21、什么是 java 序列化，如何实现 java 序列化 难度系数：⭐ 20](#_Toc24922)

[22、Http 常见的状态码 难度系数：⭐ 21](#_Toc19597)

[23、GET 和POST 的区别 难度系数：⭐ 21](#_Toc30320)

[24、Cookie 和Session 的区别 难度系数：⭐ 22](#_Toc25859)

[**第二章 Java高级篇 23**](#_Toc29821)

[1、HashMap底层源码 难度系数：⭐⭐⭐ 23](#_Toc138)

[2、JVM内存分哪几个区，每个区的作用是什么 难度系数：⭐⭐ 28](#_Toc12851)

[3、Java中垃圾收集的方法有哪些 难度系数：⭐ 32](#_Toc2748)

[4、如何判断一个对象是否存活(或者GC对象的判定方法) 难度系数：⭐ 33](#_Toc11207)

[5、什么情况下会产生StackOverflowError（栈溢出）和OutOfMemoryError（堆溢出）怎么排查 难度系数：⭐⭐ 33](#_Toc12066)

[6、什么是线程池，线程池有哪些（创建） 难度系数：⭐ 35](#_Toc32334)

[7、为什么要使用线程池 难度系数：⭐ 37](#_Toc7938)

[8、线程池底层工作原理 难度系数：⭐ 38](#_Toc1138)

[9、ThreadPoolExecutor对象有哪些参数 怎么设定核心线程数和最大线程数 拒绝策略有哪些 难度系数：⭐ 38](#_Toc26351)

[10、常见线程安全的并发容器有哪些 难度系数：⭐ 41](#_Toc29281)

[11、Atomic原子类了解多少 原理是什么 难度系数：⭐ 41](#_Toc25451)

[12、了解ConcurrentHashMap吗 为什么性能比HashTable高，说下原理 难度系数：⭐⭐ 45](#_Toc11988)

[13、ConcurrentHashMap底层原理 难度系数：⭐⭐⭐ 45](#_Toc31703)

[14、是否了解volatile关键字 难度系数：⭐ 53](#_Toc27812)

[15、synchronized和volatile有什么区别 难度系数：⭐⭐ 53](#_Toc5232)

[16、Java类加载过程 难度系数：⭐ 53](#_Toc26981)

[17、什么是类加载器，类加载器有哪些 难度系数：⭐ 54](#_Toc5288)

[18、简述java内存分配与回收策略以及Minor GC和Major GC（full GC）](#_Toc9962)

[难度系数：⭐⭐ 55](#_Toc9962)

[栈区：栈分为java虚拟机栈和本地方法栈 55](#_Toc15540)

[19、如何查看java死锁 难度系数：⭐ 56](#_Toc15251)

[20、Java死锁如何避免 难度系数：⭐ 59](#_Toc19566)

[**第三章Java框架篇 60**](#_Toc18152)

[1、简单的谈一下SpringMVC的工作流程 难度系数：⭐ 60](#_Toc4103)

[2、说出Spring或者SpringMVC中常用的5个注解 难度系数：⭐ 60](#_Toc3062)

[3、简述SpringMVC中如何返回JSON数据 难度系数：⭐ 61](#_Toc20336)

[4、谈谈你对Spring的理解 难度系数：⭐ 61](#_Toc26524)

[5、Spring中常用的设计模式 难度系数：⭐ 61](#_Toc7676)

[6、Spring循环依赖问题 难度系数：⭐⭐ 常见问法 62](#_Toc18982)

[7、Spring bean 的生命周期、注入方式和作用域 难度系数：⭐ 68](#_Toc22253)

[8、请描述一下Spring 的事务管理 难度系数：⭐ 69](#_Toc2672)

[9、MyBatis中 #{}和${}的区别是什么 难度系数：⭐ 69](#_Toc8664)

[10、Mybatis 中一级缓存与二级缓存 难度系数：⭐ 70](#_Toc28923)

[11、MyBatis如何获取自动生成的(主)键值 难度系数：⭐ 70](#_Toc491)

[13、 简述Mybatis的动态SQL，列出常用的6个标签及作用 70](#_Toc22718)

[13、Mybatis 如何完成MySQL的批量操作 难度系数：⭐ 71](#_Toc26954)

[14、谈谈怎么理解SpringBoot框架 难度系数：⭐⭐ 71](#_Toc8904)

[15、Spring Boot 的核心注解是哪个 它主要由哪几个注解组成的 难度系数：⭐ 73](#_Toc30297)

[16、Spring Boot自动配置原理是什么 难度系数：⭐ 73](#_Toc30519)

[17、SpringBoot配置文件有哪些 怎么实现多环境配置 难度系数：⭐ 73](#_Toc20194)

[18、SpringBoot和SpringCloud是什么关系 难度系数：⭐ 74](#_Toc4675)

[19、SpringCloud都用过哪些组件 介绍一下作用 难度系数：⭐ 74](#_Toc7529)

[20、Nacos作用以及注册中心的原理 难度系数：⭐⭐ 75](#_Toc7884)

[21、Feign工作原理 难度系数：⭐⭐ 77](#_Toc2578)

[**第四章 Mysql数据库 78**](#_Toc12833)

[1、Select 语句完整的执行顺序 难度系数：⭐ 78](#_Toc5062)

[2、MySQL事务 难度系数：⭐⭐ 79](#_Toc23310)

[3、MyISAM和InnoDB的区别 难度系数：⭐ 81](#_Toc4990)

[4、悲观锁和乐观锁的怎么实现 难度系数：⭐⭐ 81](#_Toc1670)

[5、聚簇索引与非聚簇索引区别 难度系数：⭐⭐ 82](#_Toc17111)

[6、什么情况下mysql会索引失效 难度系数：⭐ 83](#_Toc20057)

[7、B+tree 与 B-tree区别 难度系数：⭐⭐ 84](#_Toc10337)

[8、以MySQL为例Linux下如何排查问题 难度系数：⭐⭐ 86](#_Toc10965)

[9、如何处理慢查询 难度系数：⭐⭐ 88](#_Toc1611)

[10、数据库分表操作 难度系数：⭐ 89](#_Toc7468)

[11、MySQL优化 难度系数：⭐ 90](#_Toc27079)

[12、SQL语句优化案例 难度系数：⭐ 90](#_Toc12464)

[13、你们公司有哪些数据库设计规范 难度系数： 91](#_Toc29819)

[14、有没有设计过数据表?你是如何设计的 难度系数：⭐ 95](#_Toc8225)

[15、常见面试SQL 难度系数：⭐ 96](#_Toc5892)

[**第五章 Redis技术篇 103**](#_Toc93)

[1、介绍下Redis Redis有哪些数据类型 难度系数：⭐ 103](#_Toc30334)

[2、Redis提供了哪几种持久化方式 难度系数：⭐ 104](#_Toc29421)

[可读的日志文本，通过操作AOF 105](#_Toc21212)

[4、Redis为什么是单线程的 难度系数：⭐ 105](#_Toc7751)

[5、Redis服务器的的内存是多大 难度系数：⭐ 106](#_Toc31124)

[6、为什么Redis的操作是原子性的，怎么保证原子性的 难度系数：⭐ 106](#_Toc29043)

[7、Redis有事务吗 难度系数：⭐ 106](#_Toc25906)

[8、Redis数据和MySQL数据库的一致性如何实现 难度系数：⭐⭐ 107](#_Toc19920)

[9、缓存击穿，缓存穿透，缓存雪崩的原因和解决方案(或者说使用缓存的过程中有没有遇到什么问题，怎么解决的） 难度系数：⭐ 109](#_Toc10074)

[10、哨兵模式是什么样的 难度系数：⭐⭐ 110](#_Toc2628)

[11、Redis为什么快 难度系数：⭐ 111](#_Toc23479)

[12、Redis常见性能问题和解决方案 难度系数：⭐ 112](#_Toc12597)

[13、MySQL里有大量数据，如何保证Redis中的数据都是热点数据 难度系数：⭐⭐ 112](#_Toc27114)

[14、Redis集群方案应该怎么做 都有哪些方案 难度系数：⭐⭐ 113](#_Toc30816)

[15、说说Redis哈希槽的概念 难度系数：⭐⭐ 113](#_Toc2622)

[16、Redis有哪些适合的场景 难度系数：⭐ 114](#_Toc8965)

[17、Redis在项目中的应用 难度系数：⭐ 115](#_Toc9340)

[**第六章 分布式技术篇 116**](#_Toc1221)

[1、消息队列的作用与使用场景 难度系数：⭐ 116](#_Toc7563)

[2、RabbitMQ如何保证消息的顺序性 难度系数：⭐⭐ 120](#_Toc22234)

[3、如何使用RabbitMQ解决分布式事务 难度系数：⭐⭐ 123](#_Toc14907)

[4、RabbitMQ如何确保消息可靠性 难度系数：⭐⭐ 124](#_Toc10381)

[5、如何防止RabbitMQ消息重复消费 难度系数：⭐ 125](#_Toc17252)

[6、如何解决消息队列的延时以及过期失效问题 消息队列满了之后该如何处理 有几百万的消息持续积压几小时,说说如何解决 难度系数：⭐⭐ 126](#_Toc28462)

[7、如果让你写一个消息队列,该如何进行架构设计 难度系数：⭐⭐ 128](#_Toc23380)

[8、什么是ElasticSearch 怎么解决中文分词 难度系数：⭐ 129](#_Toc23300)

[9、Elasticsearch中的倒排索引是什么 难度系数：⭐ 129](#_Toc6016)

[10、Dubbo执行流程是什么样的？各个组件作用 难度系数：⭐ 131](#_Toc22422)

[11、ZooKeeper是什么 难度系数：⭐ 133](#_Toc24071)

[12、Zookeeper Watcher 机制 -- 数据变更通知 难度系数：⭐ 134](#_Toc4347)

[13、ZooKeeper分布式锁的实现原理 难度系数：⭐ 136](#_Toc26949)

[14、Nginx原理 难度系数：⭐ 136](#_Toc32168)

[15、Nginx作用以及常见配置 难度系数：⭐ 140](#_Toc22183)

[**第七章 项目管理工具Git篇 144**](#_Toc18294)

[1、工作中git开发使用流程（命令版描述） 144](#_Toc20506)

[2、Reset 与Rebase,Pull 与 Fetch 的区别 145](#_Toc23686)

[3、git merge和git rebase的区别 144](#_Toc20506)

[4、git如何解决代码冲突 14](#_Toc23686)4

[5、项目开发时git分支情况 145](#_Toc20506)

[**第八章 Linux 146**](#_Toc31922)

[1、Linux常用命令 146](#_Toc25520)

[2、如何查看测试项目的日志 147](#_Toc8232)

[3、如何查看最近1000行日志 148](#_Toc6805)

[4、Linux中如何查看某个端口是否被占用 148](#_Toc20284)

[5、查看当前所有已经使用的端口情况 149](#_Toc27438)

[6、怎么用一条命令杀掉端口号为8080的进程 150](#_Toc14483)

[**第九章 电商项目篇 150**](#_Toc16356)

[1、介绍下最近做的项目 150](#_Toc8677)

[2、项目开发周期： 157](#_Toc183)

[3、项目参与人数： 158](#_Toc20401)

[4、公司开发相关各岗位职责： 158](#_Toc29350)

[5、项目开发流程: 160](#_Toc6112)

[6、项目版本控制： 165](#_Toc10561)

[7、一般项目服务器数量： 165](#_Toc26642)

[8、上线后QPS并发量，用户量、同时在线人数并发数等问题 166](#_Toc7532)

[9、你们项目的微服务是怎么拆分的，拆分了多少？ 167](#_Toc2472)

[10、如何解决并发问题的？ 167](#_Toc15112)

[11、如何保证接口的幂等性？ 167](#_Toc13282)

[12、你们项目中有没有用到什么设计模式？ 169](#_Toc3423)

[13、生产环境出问题，你们是怎么排查的？ 171](#_Toc372)

[14、你做完这个项目后有什么收获？ 171](#_Toc12602)

[15、在做这个项目的时候你碰到了哪些问题？你是怎么解决的？ 173](#_Toc6920)

[**第十章 数据结构和算法篇 174**](#_Toc23215)

[1、怎么理解时间复杂度和空间复杂度 174](#_Toc11061)

[2、数组和链表结构简单对比 174](#_Toc14116)

[3、怎么遍历一个树 175](#_Toc14292)

[4、冒泡排序（Bubble Sort） 175](#_Toc29131)

[5、快速排序（Quick Sort） 181](#_Toc5518)

[6、二分查找（Binary Search） 189](#_Toc16506)

[**第十一章 设计模式 204**](#_Toc23452)

[1、你所知道的设计模式有哪些 204](#_Toc14749)

[2、单例模式（Binary Search） 205](#_Toc20243)

[3、工厂设计模式（Factory） 230](#_Toc19580)

[4、代理模式（Proxy） 252](#_Toc11635)

[**第十二章 面试技巧篇 269**](#_Toc11623)

[1、面试过程最关键的是什么？ 269](#_Toc25450)

[2、面试时该怎么说？ 269](#_Toc16026)

[3、面试技巧 270](#_Toc11476)

第一章 Java基础篇

### 1、你是怎样理解OOP面向对象 难度系数：⭐

面向对象是利于语言对现实事物进行抽象。面向对象具有以下特征：

* 继承**：**继承是从已有类得到继承信息创建新类的过程
* 封装：封装是把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口
* 多态性：多态性是指允许不同子类型的对象对同一消息作出不同的响应

### 重载与重写区别 难度系数：⭐

* 重载发生在本类，重写发生在父类与子类之间
* 重载的方法名必须相同，重写的方法名相同且返回值类型必须相同
* 重载的参数列表不同，重写的参数列表必须相同
* 重写的访问权限不能比父类中被重写的方法的访问权限更低
* 构造方法不能被重写

### 接口与抽象类的区别 难度系数：⭐

* 抽象类要被子类继承，接口要被类实现
* 接口可多继承接口，但类只能单继承
* 抽象类可以有构造器、接口不能有构造器
* 抽象类：除了不能实例化抽象类之外，它和普通Java类没有任何区别
* 抽象类：抽象方法可以有public、protected和default这些修饰符、接口：只能是public
* 抽象类：可以有成员变量；接口：只能声明常量

### 4、深拷贝与浅拷贝的理解 难度系数：⭐

深拷贝和浅拷贝就是指对象的拷贝，一个对象中存在两种类型的属性，一种是基本数据类型，一种是实例对象的引用。

* 浅拷贝是指，只会拷贝基本数据类型的值，以及实例对象的引用地址，并不会复制一份引用地址所指向的对象，也就是浅拷贝出来的对象，内部的类属性指向的是同一个对象
* 深拷贝是指，既会拷贝基本数据类型的值，也会针对实例对象的引用地址所指向的对象进行复制，深拷贝出来的对象，内部的类执行指向的不是同一个对象

## 5、sleep和wait区别 难度系数：⭐

* sleep方法

属于Thread类中的方法

释放cpu给其它线程 不释放锁资源

sleep(1000) 等待超过1s被唤醒

* wait方法

属于Object类中的方法

释放cpu给其它线程，同时释放锁资源

wait(1000) 等待超过1s被唤醒

wait() 一直等待需要通过notify或者notifyAll进行唤醒

wait 方法必须配合 synchronized 一起使用，不然在运行时就会抛出IllegalMonitorStateException异常

#### 锁释放时机代码演示  
public static void main(String[] args) {  
 Object o = new Object();  
 Thread thread = new Thread(() -> {  
 synchronized (o) {  
 System.out.println("新线程获取锁时间：" + LocalDateTime.now() + " 新线程名称：" + Thread.currentThread().getName());  
 try {  
  
 //wait 释放cpu同时释放锁  
 o.wait(2000);  
  
 //sleep 释放cpu不释放锁  
 //Thread.sleep(2000);  
 System.out.println("新线程获取释放锁锁时间：" + LocalDateTime.now() + " 新线程名称：" + Thread.currentThread().getName());  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
 });  
  
 thread.start();  
  
 try {  
 Thread.sleep(100);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
  
 System.out.println("主线程获取锁时间：" + LocalDateTime.now() + " 主线程名称：" + Thread.currentThread().getName());  
  
 synchronized (o){  
 System.out.println("主线程获取释放锁锁时间：" + LocalDateTime.now() + " 主线程名称：" + Thread.currentThread().getName());  
 }  
}

Java

## 6、什么是自动拆装箱 int和Integer有什么区别 难度系数：⭐

基本数据类型，如int,float,double,boolean,char,byte,不具备对象的特征，不能调用方法。

* 装箱：将基本类型转换成包装类对象
* 拆箱：将包装类对象转换成基本类型的值

java为什么要引入自动装箱和拆箱的功能？主要是用于java集合中，List<Inteter> list=new ArrayList<Integer>();

list集合如果要放整数的话，只能放对象，不能放基本类型，因此需要将整数自动装箱成对象。

**实现原理：**javac编译器的语法糖，底层是通过Integer.valueOf()和Integer.intValue()方法实现。

**区别：**

* Integer是int的包装类，int则是java的一种基本数据类型
* Integer变量必须实例化后才能使用，而int变量不需要
* Integer实际是对象的引用，当new一个Integer时，实际上是生成一个指针指向此对象；而int则是直接存储数据值
* Integer的默认值是null，int的默认值是0

## 7、==和equals区别 难度系数：⭐

* ==

如果比较的是基本数据类型，那么比较的是变量的值

如果比较的是引用数据类型，那么比较的是地址值（两个对象是否指向同一块内存）

* equals

如果没重写equals方法比较的是两个对象的地址值

如果重写了equals方法后我们往往比较的是对象中的属性的内容

equals方法是从Object类中继承的，默认的实现就是使用==

## 8、String能被继承吗 为什么用final修饰 难度系数：⭐

* 不能被继承，因为String类有final修饰符，而final修饰的类是不能被继承的。
* String 类是最常用的类之一，为了效率，禁止被继承和重写。
* 为了安全。String 类中有native关键字修饰的调用系统级别的本地方法，调用了操作系统的 API，如果方法可以重写，可能被植入恶意代码，破坏程序。Java 的安全性也体现在这里。

## 9、String buffer和String builder区别 难度系数：⭐

* StringBuffer 与 StringBuilder 中的方法和功能完全是等价的，
* 只是StringBuffer 中的方法大都采用了 synchronized 关键字进行修饰，因此是线程安全的，而 StringBuilder 没有这个修饰，可以被认为是线程不安全的。
* 在单线程程序下，StringBuilder效率更快，因为它不需要加锁，不具备多线程安全而StringBuffer则每次都需要判断锁，效率相对更低

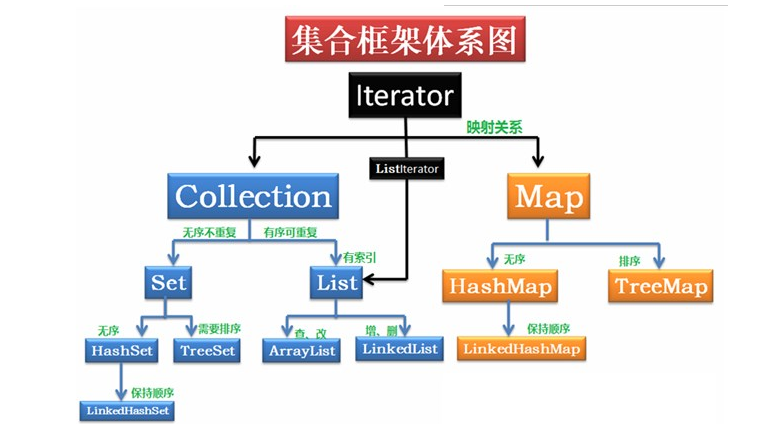
## 10、final、finally、finalize 难度系数：⭐

* **final：**修饰符（关键字）有三种用法：修饰类、变量和方法。修饰类时，意味着它不能再派生出新的子类，即不能被继承，因此它和abstract是反义词。修饰变量时，该变量使用中不被改变，必须在声明时给定初值，在引用中只能读取不可修改，即为常量。修饰方法时，也同样只能使用，不能在子类中被重写。
* **finally：**通常放在try…catch的后面构造最终执行代码块，这就意味着程序无论正常执行还是发生异常，这里的代码只要JVM不关闭都能执行，可以将释放外部资源的代码写在finally块中。
* **finalize：**Object类中定义的方法，Java中允许使用finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在销毁对象时调用的，通过重写finalize() 方法可以整理系统资源或者执行其他清理工作。

## 11、Object中有哪些方法 难度系数：⭐

* protected Object clone()--->创建并返回此对象的一个副本。
* boolean equals(Object obj)--->指示某个其他对象是否与此对象“相等
* protected void finalize()--->当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。
* Class<? extendsObject> getClass()--->返回一个对象的运行时类。
* int hashCode()--->返回该对象的哈希码值。
* void notify()--->唤醒在此对象监视器上等待的单个线程。
* void notifyAll()--->唤醒在此对象监视器上等待的所有线程。
* String toString()--->返回该对象的字符串表示。
* void wait()--->导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法。   
   void wait(long timeout)--->导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll()方法，或者超过指定的时间量。   
   void wait(long timeout, int nanos)--->导致当前的线程等待，直到其他线程调用此对象的 notify()

## 12、说一下集合体系 难度系数：⭐



## 13、ArrarList和LinkedList区别 难度系数：⭐

* ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。
* 对于随机访问get和set，ArrayList效率优于LinkedList，因为LinkedList要移动指针。
* 对于新增和删除操作add和remove，LinkedList比较占优势，因为ArrayList要移动数据。 这一点要看实际情况的。若只对单条数据插入或删除，ArrayList的速度反而优于LinkedList。但若是批量随机的插入删除数据，LinkedList的速度大大优于ArrayList. 因为ArrayList每插入一条数据，要移动插入点及之后的所有数据。

## 14、HashMap底层是 数组+链表+红黑树，为什么要用这几类结构 难度系数：⭐⭐

* 数组 Node<K,V>[] table ,哈希表，根据对象的key的hash值进行在数组里面是哪个节点
* 链表的作用是解决hash冲突，将hash值取模之后的对象存在一个链表放在hash值对应的槽位
* 红黑树 JDK8使用红黑树来替代超过8个节点的链表，主要是查询性能的提升，从原来的O(n)到O(logn),
* 通过hash碰撞，让HashMap不断产生碰撞，那么相同的key的位置的链表就会不断增长，当对这个Hashmap的相应位置进行查询的时候，就会循环遍历这个超级大的链表，性能就会下降，所以改用红黑树

## 15、HashMap和HashTable区别 难度系数：⭐

* **线程安全性不同**

HashMap是线程不安全的，HashTable是线程安全的，其中的方法是Synchronized，在多线程并发的情况下，可以直接使用HashTable，但是使用HashMap时必须自己增加同步处理。

* **是否提供contains方法**

HashMap只有containsValue和containsKey方法；HashTable有contains、containsKey和containsValue三个方法，其中contains和containsValue方法功能相同。

* **key和value是否允许null值**

Hashtable中，key和value都不允许出现null值。HashMap中，null可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为null。

* **数组初始化和扩容机制**

HashTable在不指定容量的情况下的默认容量为11，而HashMap为16，Hashtable不要求底层数组的容量一定要为2的整数次幂，而HashMap则要求一定为2的整数次幂。

Hashtable扩容时，将容量变为原来的2倍加1，而HashMap扩容时，将容量变为原来的2倍。

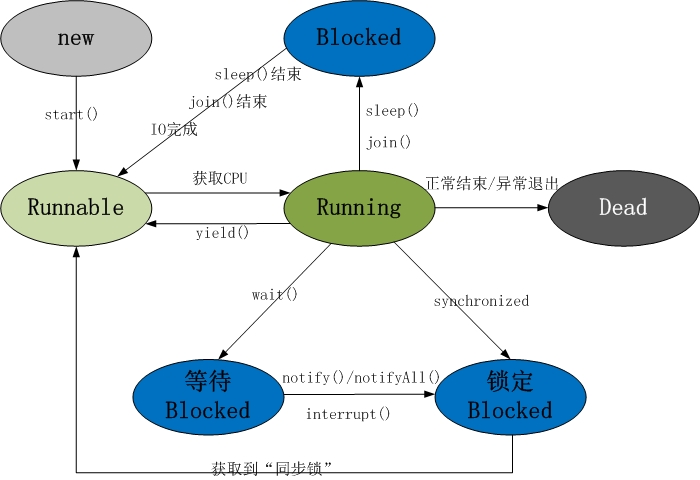
## 16、线程的创建方式 难度系数：⭐

* 继承Thread类创建线程
* 实现Runnable接口创建线程
* 使用Callable和Future创建线程 有返回值
* 使用线程池创建线程

#### 代码演示  
import java.util.concurrent.\*;  
public class threadTest{  
 public static void main(String[] args) throws ExecutionException, InterruptedException {  
 //继承thread  
 ThreadClass thread = new ThreadClass();  
 thread.start();  
 Thread.sleep(100);  
 System.out.println("#####################");  
  
 //实现runnable  
 RunnableClass runnable = new RunnableClass();  
 new Thread(runnable).start();  
 Thread.sleep(100);  
 System.out.println("#####################");  
  
 //实现callable  
 FutureTask futureTask = new FutureTask(new CallableClass());  
 futureTask.run();  
 System.out.println("callable返回值：" + futureTask.get());  
 Thread.sleep(100);  
 System.out.println("#####################");  
  
 //线程池  
 ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(1, 1, 2, TimeUnit.SECONDS, new ArrayBlockingQueue<>(10));  
 threadPoolExecutor.execute(thread);  
 threadPoolExecutor.shutdown();  
 Thread.sleep(100);  
 System.out.println("#####################");  
  
 //使用并发包Executors  
 ExecutorService executorService = Executors.newFixedThreadPool(5);  
 executorService.execute(thread);  
 executorService.shutdown();  
 }  
}  
  
class ThreadClass extends Thread{  
 @Override  
 public void run() {  
 System.out.println("我是继承thread形式：" + Thread.currentThread().getName());  
 }  
}  
  
class RunnableClass implements Runnable{  
 @Override  
 public void run(){  
 System.out.println("我是实现runnable接口：" + Thread.currentThread().getName());  
 }  
}  
  
class CallableClass implements Callable<String> {  
 @Override  
 public String call(){  
 System.out.println("我是实现callable接口：");  
 return "我是返回值，可以通过get方法获取";  
 }  
}

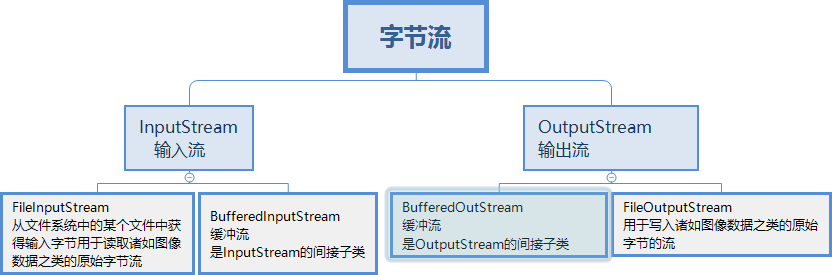
Java

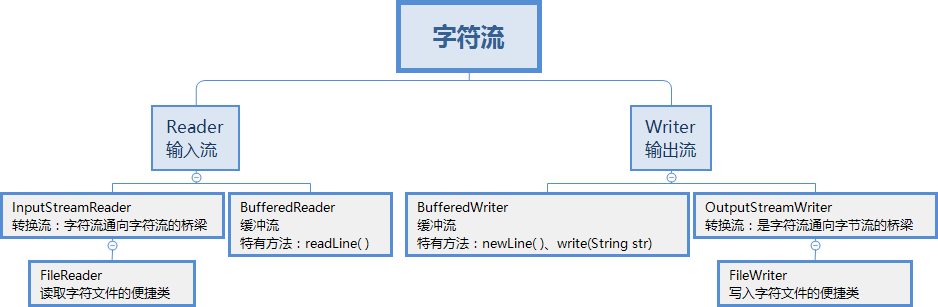
## 17、线程的状态转换有什么（生命周期） 难度系数：⭐



* 新建状态(New) ：线程对象被创建后，就进入了新建状态。例如，Thread thread = new Thread()。
* 就绪状态(Runnable): 也被称为“可执行状态”。线程对象被创建后，其它线程调用了该对象的start()方法，从而来启动该线程。例如，thread.start()。处于就绪状态的线程，随时可能被CPU调度执行。
* 运行状态(Running)：线程获取CPU权限进行执行。需要注意的是，线程只能从就绪状态进入到运行状态。
* 阻塞状态(Blocked)：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：
* 等待阻塞 -- 通过调用线程的wait()方法，让线程等待某工作的完成。
* 同步阻塞 -- 线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态。
* 其他阻塞 -- 通过调用线程的sleep()或join()或发出了I/O请求时，线程会进入到阻塞状态。当sleep()状态超时、join(）等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。
* 死亡状态(Dead)：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

## 18、Java中有几种类型的流 难度系数：⭐





## 19、请写出你最常见的5个RuntimeException 难度系数：⭐

* java.lang.NullPointerException
* 空指针异常；出现原因：调用了未经初始化的对象或者是不存在的对象。
* java.lang.ClassNotFoundException
* 指定的类找不到；出现原因：类的名称和路径加载错误；通常都是程序试图通过字符串来加载某个类时可能引发异常。
* java.lang.NumberFormatException
* 字符串转换为数字异常；出现原因：字符型数据中包含非数字型字符。
* java.lang.IndexOutOfBoundsException
* 数组角标越界异常，常见于操作数组对象时发生。
* java.lang.IllegalArgumentException
* 方法传递参数错误。
* java.lang.ClassCastException

数据类型转换异常。

## 20、谈谈你对反射的理解 难度系数：⭐

* **反射机制**

所谓的反射机制就是java语言在运行时拥有一项自观的能力。通过这种能力可以彻底了解自身的情况为下一步的动作做准备。

Java的反射机制的实现要借助于4个类：class，Constructor，Field，Method;其中class代表的时类对 象，Constructor－类的构造器对象，Field－类的属性对象，Method－类的方法对象。通过这四个对象我们可以粗略的看到一个类的各个组成部分。

* **Java反射的作用**

在Java运行时环境中，对于任意一个类，可以知道这个类有哪些属性和方法。对于任意一个对象，可以调用它的任意一个方法。这种动态获取类的信息以及动态调用对象的方法的功能来自于Java 语言的反射（Reflection）机制。

* **Java 反射机制提供功能**

在运行时判断任意一个对象所属的类。

在运行时构造任意一个类的对象。

在运行时判断任意一个类所具有的成员变量和方法。

在运行时调用任意一个对象的方法

## 21、什么是 java 序列化，如何实现 java 序列化 难度系数：⭐

* 序列化是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间。序列化是为了解决在对对象流进行读写操作时所引发的问题。
* 序 列 化 的 实 现 ： 将 需 要 被 序 列 化 的 类 实 现 Serializable 接 口 ， 该 接 口 没 有 需 要 实 现 的 方 法 ， implements Serializable 只是为了标注该对象是可被序列化的，然后使用一个输出流(如：FileOutputStream)来构造一个ObjectOutputStream(对象流)对象，接着，使用 ObjectOutputStream 对象的 writeObject(Object obj)方法就可以将参数为 obj 的对象写出(即保存其状态)，要恢复的话则用输入流。

## 22、Http 常见的状态码 难度系数：⭐

1. 200 OK //客户端请求成功
2. 301 Permanently Moved （永久移除)，请求的 URL 已移走。Response 中应该包含一个 Location URL, 说明资源现在所处的位置
3. 302 Temporarily Moved 临时重定向
4. 400 Bad Request //客户端请求有语法错误，不能被服务器所理解
5. 401 Unauthorized //请求未经授权，这个状态代码必须和 WWW-Authenticate 报头域一起使用
6. 403 Forbidden //服务器收到请求，但是拒绝提供服务
7. 404 Not Found //请求资源不存在，eg：输入了错误的 URL
8. 500 Internal Server Error //服务器发生不可预期的错误
9. 503 Server Unavailable //服务器当前不能处理客户端的请求，一段时间后可能恢复正常

## 23、GET 和POST 的区别 难度系数：⭐

* GET 请求的数据会附在URL 之后（就是把数据放置在 HTTP 协议头中），以?分割URL 和传输数据，参数之间以&相连，如：login.action?name=zhagnsan&password=123456。POST 把提交的数据则放置在是 HTTP 包的包体中。
* GET 方式提交的数据最多只能是 1024 字节，理论上POST 没有限制，可传较大量的数据。其实这样说是错误的，不准确的：“GET 方式提交的数据最多只能是 1024 字节"，因为 GET 是通过 URL 提交数据，那么 GET 可提交的数据量就跟URL 的长度有直接关系了。而实际上，URL 不存在参数上限的问题，HTTP 协议规范没有对 URL 长度进行限制。这个限制是特定的浏览器及服务器对它的限制。IE 对URL 长度的限制是2083 字节(2K+35)。对于其他浏览器，如Netscape、FireFox 等，理论上没有长度限制，其限制取决于操作系统的支持。
* POST 的安全性要比GET 的安全性高。注意：这里所说的安全性和上面 GET 提到的“安全”不是同个概念。上面“安全”的含义仅仅是不作数据修改，而这里安全的含义是真正的 Security 的含义，比如：通过 GET 提交数据，用户名和密码将明文出现在 URL 上，因为(1)登录页面有可能被浏览器缓存，(2)其他人查看浏览器的历史纪录，那么别人就可以拿到你的账号和密码了，除此之外，使用 GET 提交数据还可能会造成 Cross-site request forgery 攻击。
* Get 是向服务器发索取数据的一种请求，而 Post 是向服务器提交数据的一种请求，在 FORM（表单）中，Method
* 默认为"GET"，实质上，GET 和 POST 只是发送机制不同，并不是一个取一个发！

## 24、Cookie 和Session 的区别 难度系数：⭐

* Cookie 是 web 服务器发送给浏览器的一块信息，浏览器会在本地一个文件中给每个 web 服务器存储 cookie。以后浏览器再给特定的 web 服务器发送请求时，同时会发送所有为该服务器存储的 cookie
* Session 是存储在 web 服务器端的一块信息。session 对象存储特定用户会话所需的属性及配置信息。当用户在应用程序的 Web 页之间跳转时，存储在 Session 对象中的变量将不会丢失，而是在整个用户会话中一直存在下去
* Cookie 和session 的不同点

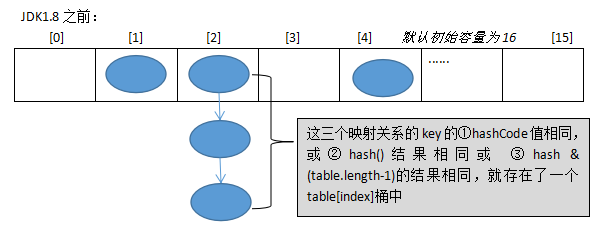
无论客户端做怎样的设置，session 都能够正常工作。当客户端禁用 cookie 时将无法使用 cookie

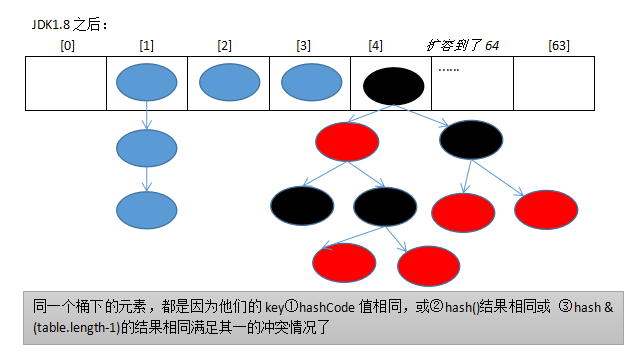
在存储的数据量方面：session 能够存储任意的java 对象，cookie 只能存储 String 类型的对象

第二章 Java高级篇

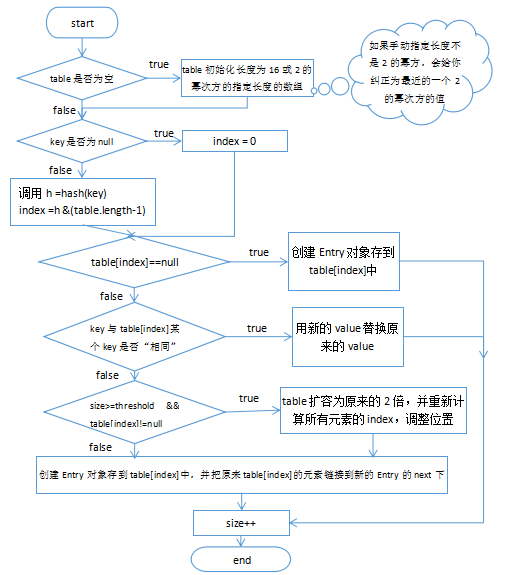
### 1、HashMap底层源码 难度系数：⭐⭐⭐

HashMap的底层结构在jdk1.7中由数组+链表实现，在jdk1.8中由数组+链表+红黑树实现，以数组+链表的结构为例。

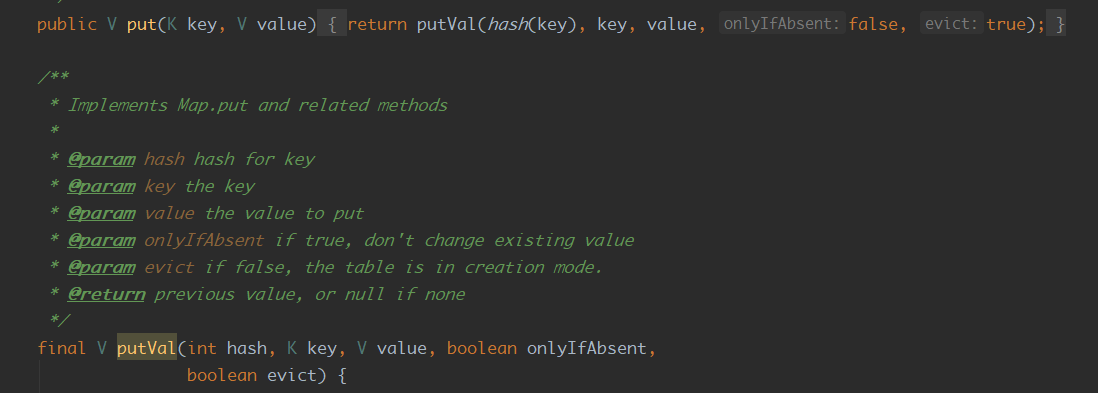




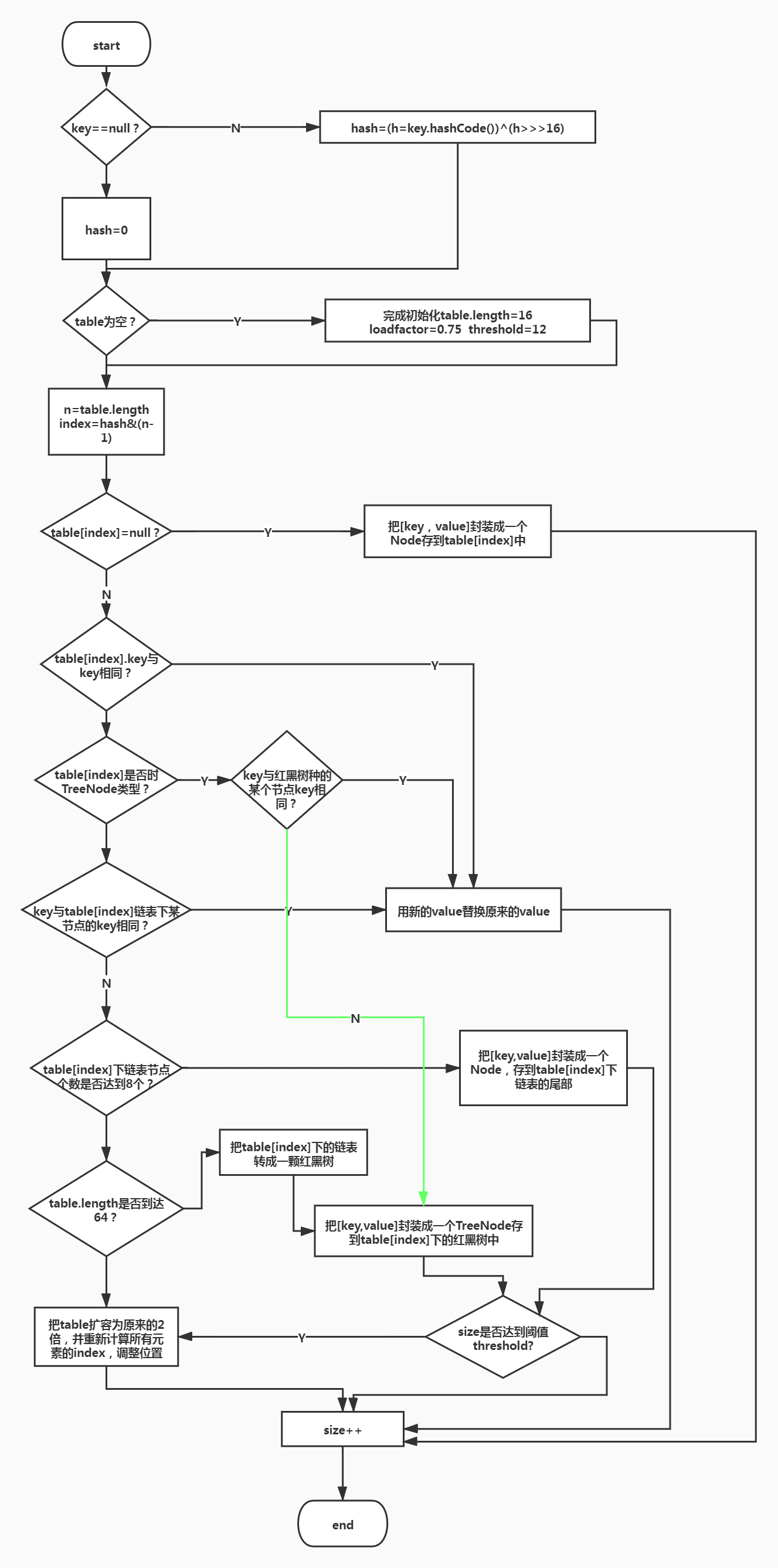
**JDK1.8之前Put方法：**



**JDK1.8之后Put方法：**







**HashMap基于哈希表的Map接口实现，是以key-value存储形式存在，即主要用来存放键值对**。HashMap 的实现不是同步的，这意味着它不是线程安全的。它的key、value都可以为null。此外，HashMap中的映射不是有序的。

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(两个对象调用的hashCode方法计算的哈希码值一致导致计算的数组索引值相同)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（或者红黑树的边界值，默认为 8）并且当前数组的长度大于64时，此时此索引位置上的所有数据改为使用红黑树存储。

**补充**：将链表转换成红黑树前会判断，即使阈值大于8，但是数组长度小于64，此时并不会将链表变为红黑树。而是选择进行数组扩容。

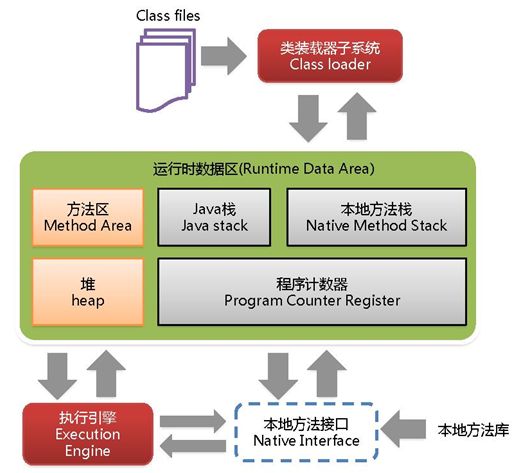
这样做的目的是因为数组比较小，尽量避开红黑树结构，这种情况下变为红黑树结构，反而会降低效率，因为红黑树需要进行左旋，右旋，变色这些操作来保持平衡 。同时数组长度小于64时，搜索时间相对要快些。所以综上所述为了提高性能和减少搜索时间，底层在阈值大于8并且数组长度大于64时，链表才转换为红黑树。具体可以参考 treeifyBin方法。

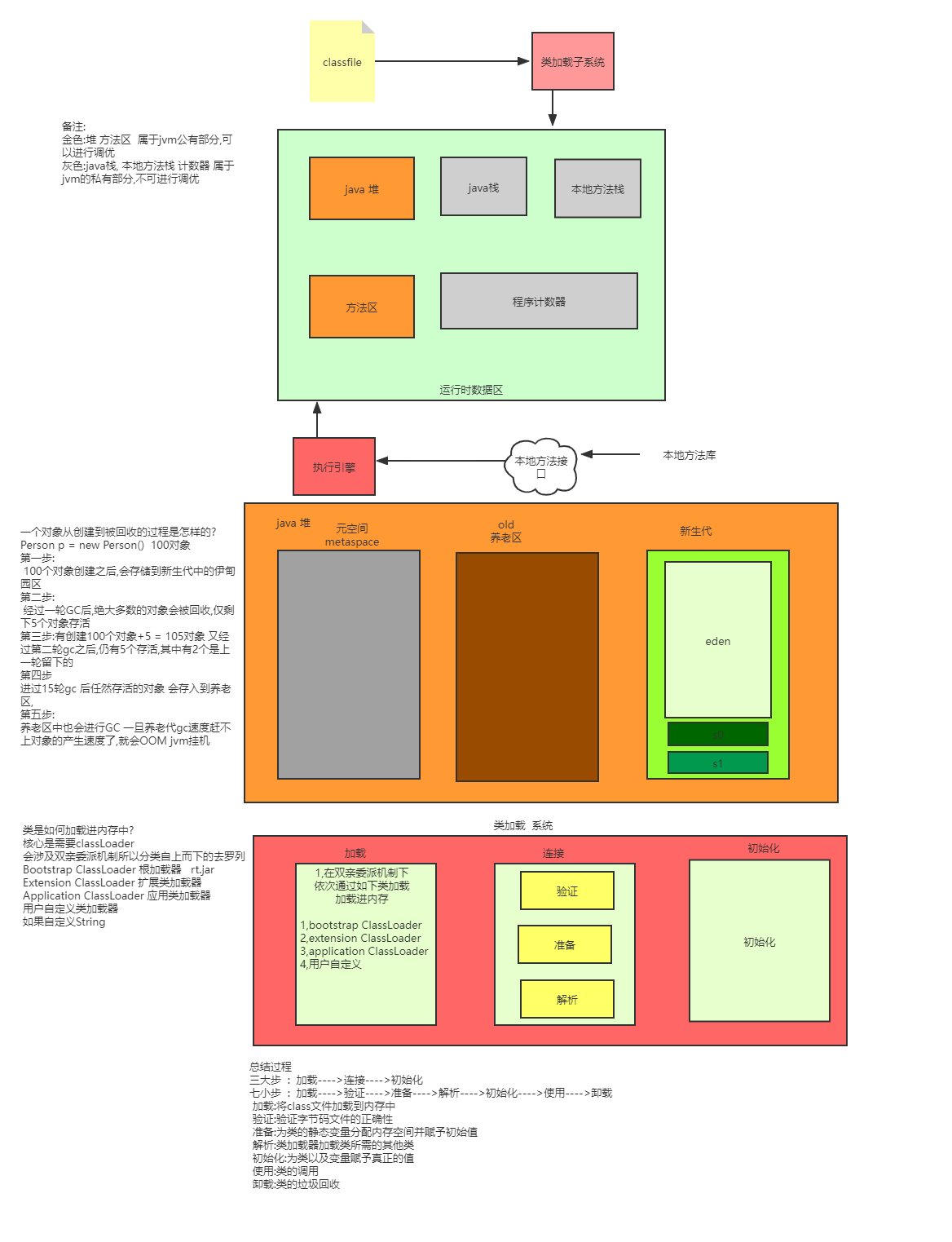
当然虽然增了红黑树作为底层数据结构，结构变得复杂了，但是阈值大于8并且数组长度大于64时，链表转换为红黑树时，效率也变的更高效。

**注意：可以结合百度hashmap源码解析进行更深入的了解。**

[**https://blog.csdn.net/v123411739/article/details/78996181**](https://blog.csdn.net/v123411739/article/details/78996181)

### 2、JVM内存分哪几个区，每个区的作用是什么 难度系数：⭐⭐





java虚拟机主要分为以下几个区

1. **方法区**

* 有时候也成为永久代，在该区内很少发生垃圾回收，但是并不代表不发生GC，在这里进行的GC主要是对方法区里的常量池和对类型的卸载
* 方法区主要用来存储已被虚拟机加载的类的信息、常量、静态变量和即时编译器编译后的代码等数据。
* 该区域是被线程共享的。
* 方法区里有一个运行时常量池，用于存放静态编译产生的字面量和符号引用。该常量池具有动态性，也就是说常量并不一定是编译时确定，运行时生成的常量也会存在这个常量池中。

(2)**虚拟机栈**

* 虚拟机栈也就是我们平常所称的栈内存,它为java方法服务，每个方法在执行的时候都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接和方法出口等信息。
* 虚拟机栈是线程私有的，它的生命周期与线程相同。
* 局部变量表里存储的是基本数据类型、returnAddress类型（指向一条字节码指令的地址）和对象引用，这个对象引用有可能是指向对象起始地址的一个指针，也有可能是代表对象的句柄或者与对象相关联的位置。局部变量所需的内存空间在编译器间确定
* 操作数栈的作用主要用来存储运算结果以及运算的操作数，它不同于局部变量表通过索引来访问，而是压栈和出栈的方式
* 每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接.动态链接就是将常量池中的符号引用在运行期转化为直接引用。

**(3)本地方法栈**

本地方法栈和虚拟机栈类似，只不过本地方法栈为Native方法服务。

* **堆**

java堆是所有线程所共享的一块内存，在虚拟机启动时创建，几乎所有的对象实例都在这里创建，因此该区域经常发生垃圾回收操作。

* **程序计数器：**

内存空间小，字节码解释器工作时通过改变这个计数值可以选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理和线程恢复等功能都需要依赖这个计数器完成。该内存区域是唯一一个java虚拟机规范没有规定任何OOM情况的区域。

<https://openjdk.org/jeps/122> 介绍 静态变量、字符串常量从永久代移动到堆中

### 3、Java中垃圾收集的方法有哪些 难度系数：⭐

**采用分区分代回收思想：**

* **复制算法** 年轻代中使用的是Minor GC，这种GC算法采用的是复制算法(Copying)

a) 效率高，缺点：需要内存容量大，比较耗内存

b) 使用在占空间比较小、刷新次数多的新生区

* **标记-清除** 老年代一般是由标记清除或者是标记清除与标记整理的混合实现

a) 效率比较低，会差生碎片。

* **标记-整理** 老年代一般是由标记清除或者是标记清除与标记整理的混合实现

a) 效率低速度慢，需要移动对象，但不会产生碎片。

## 4、如何判断一个对象是否存活(或者GC对象的判定方法) 难度系数：⭐

1. **引用计数法**

所谓引用计数法就是给每一个对象设置一个引用计数器，每当有一个地方引用这个对象时，就将计数器加一，引用失效时，计数器就减一。当一个对象的引用计数器为零时，说明此对象没有被引用，也就是“死对象”,将会被垃圾回收.

引用计数法有一个缺陷就是无法解决循环引用问题，也就是说当对象A引用对象B，对象B又引用者对象A，那么此时A,B对象的引用计数器都不为零，也就造成无法完成垃圾回收，所以主流的虚拟机都没有采用这种算法。

1. **可达性算法(引用链法)** 
   * 该算法的基本思路就是通过一些被称为引用链（GC Roots）的对象作为起点，从这些节点开始向下搜索，搜索走过的路径被称为（Reference Chain)，当一个对象到GC Roots没有任何引用链相连时（即从GC Roots节点到该节点不可达），则证明该对象是不可用的。
   * 在java中可以作为GC Roots的对象有以下几种：虚拟机栈中引用的对象、方法区类静态属性引用的对象、方法区常量池引用的对象、本地方法栈JNI引用的对象。

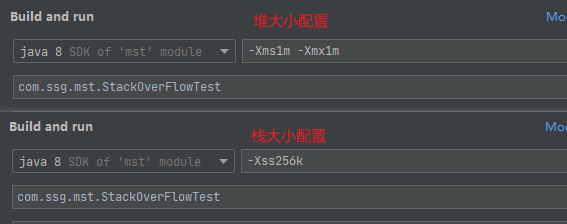
## 5、什么情况下会产生StackOverflowError（栈溢出）和OutOfMemoryError（堆溢出）怎么排查 难度系数：⭐⭐

1. 引发 StackOverFlowError 的常见原因有以下几种
   * 无限递归循环调用（最常见）
   * 执行了大量方法，导致线程栈空间耗尽
   * 方法内声明了海量的局部变量
   * native 代码有栈上分配的逻辑，并且要求的内存还不小，比如 java.net.SocketInputStream.read0 会在栈上要求分配一个 64KB 的缓存（64位 Linux）。
2. 引发 OutOfMemoryError的常见原因有以下几种
   * 内存中加载的数据量过于庞大，如一次从数据库取出过多数据
   * 集合类中有对对象的引用，使用完后未清空，使得JVM不能回收
   * 代码中存在死循环或循环产生过多重复的对象实体
   * 启动参数内存值设定的过小

* 排查：可以通过jvisualvm进行内存快照分析

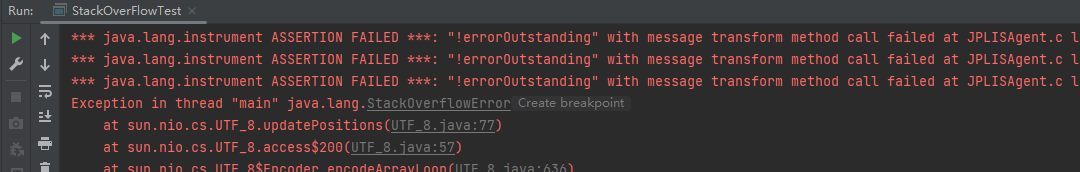
参考<https://www.cnblogs.com/boboooo/p/13164071.html>

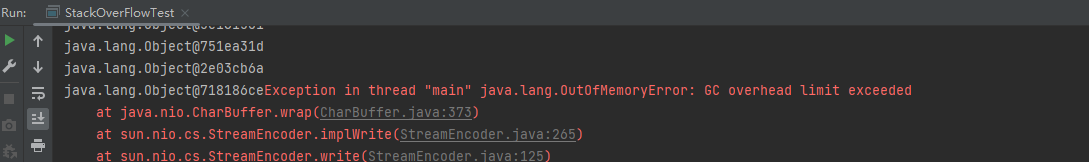
* 栈溢出、堆溢出案例演示



public class StackOverFlowTest {  
 private static int count = 1;  
 public static void main(String[] args) {  
 //模拟栈溢出  
 //getDieCircle();  
   
 //模拟堆溢出  
 getOutOfMem();  
 }  
  
 public static void getDieCircle(){  
 System.out.println(count++);  
 getDieCircle();  
 }  
  
 public static void getOutOfMem(){  
 while (true) {  
 Object o = new Object();  
 System.out.println(o);  
 }  
 }  
}

Java





### 6、什么是线程池，线程池有哪些（创建） 难度系数：⭐

**线程池就是事先将多个线程对象放到一个容器中，当使用的时候就不用 new 线程而是直接去池中拿线程即可，节省了开辟子线程的时间，提高的代码执行效率**

**在 JDK 的 java.util.concurrent.Executors 中提供了生成多种线程池的静态方法。**

ExecutorService newCachedThreadPool = Executors.newCachedThreadPool();

ExecutorService newFixedThreadPool = Executors.newFixedThreadPool(4);

ScheduledExecutorService newScheduledThreadPool = Executors.newScheduledThreadPool(4);

ExecutorService newSingleThreadExecutor = Executors.newSingleThreadExecutor();

然后调用他们的 execute 方法即可。

这4种线程池底层 全部是ThreadPoolExecutor对象的实现，阿里规范手册中规定线程池采用ThreadPoolExecutor自定义的，实际开发也是。

1. **newCachedThreadPool**

创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。这种类型的线程池特点是：

工作线程的创建数量几乎没有限制(其实也有限制的,数目为Interger. MAX\_VALUE), 这样可灵活的往线程池中添加线程。

如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间(默认为1分钟)，则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。

在使用CachedThreadPool时，一定要注意控制任务的数量，否则，由于大量线程同时运行，很有会造成系统瘫痪。

1. **newFixedThreadPool**

创建一个指定工作线程数量的线程池。每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。FixedThreadPool是一个典型且优秀的线程池，它具有线程池提高程序效率和节省创建线程时所耗的开销的优点。但是，在线程池空闲时，即线程池中没有可运行任务时，它不会释放工作线程，还会占用一定的系统资源。

1. **newSingleThreadExecutor**

创建一个单线程化的Executor，即只创建唯一的工作者线程来执行任务，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。如果这个线程异常结束，会有另一个取代它，保证顺序执行。单工作线程最大的特点是可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的。

1. **newScheduleThreadPool**

创建一个定长的线程池，而且支持定时的以及周期性的任务执行。例如延迟3秒执行。

### 7、为什么要使用线程池 难度系数：⭐

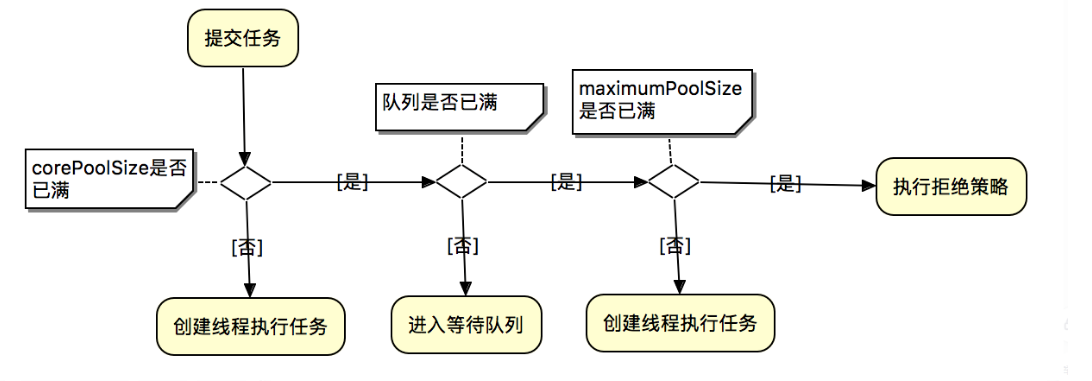
1. 线程池做的工作主要是控制运行的线程数量，处理过程中将任务放入队列，然后在线程创建后启动这些任务，如果线程数量超过了最 大数量，超出数量的线程排队等候，等其它线程执行完毕，再从队列中取出任务来执行。
2. 主要特点:线程复用;控制最大并发数:管理线程。

第一: 降低资源消耗。通过重复利用己创建的线程降低线程创建和销毁造成的消耗。

第二: 提高响应速度。当任务到达时，任务可以不需要的等到线程创建就能立即执行。

第三: 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

### 8、线程池底层工作原理 难度系数：⭐



1. 第一步：线程池刚创建的时候，里面没有任何线程，等到有任务过来的时候才会创建线程。当然也可以调用 prestartAllCoreThreads() 或者 prestartCoreThread() 方法预创建corePoolSize个线程
2. 第二步：调用execute()提交一个任务时，如果当前的工作线程数<corePoolSize，直接创建新的线程执行这个任务
3. 第三步：如果当时工作线程数量>=corePoolSize，会将任务放入任务队列中缓存
4. 第四步：如果队列已满，并且线程池中工作线程的数量<maximumPoolSize，还是会创建线程执行这个任务
5. 第五步：如果队列已满，并且线程池中的线程已达到maximumPoolSize，这个时候会执行拒绝策略，JAVA线程池默认的策略是AbortPolicy，即抛出RejectedExecutionException异常

## 9、ThreadPoolExecutor对象有哪些参数 怎么设定核心线程数和最大线程数 拒绝策略有哪些 难度系数：⭐

**参数与作用：**共7个参数

1. **corePoolSize：**核心线程数，

在ThreadPoolExecutor中有一个与它相关的配置：allowCoreThreadTimeOut（默认为false），当allowCoreThreadTimeOut为false时，核心线程会一直存活，哪怕是一直空闲着。而当allowCoreThreadTimeOut为true时核心线程空闲时间超过keepAliveTime时会被回收。

1. **maximumPoolSize：**最大线程数

线程池能容纳的最大线程数，当线程池中的线程达到最大时，此时添加任务将会采用拒绝策略，默认的拒绝策略是抛出一个运行时错误（RejectedExecutionException）。值得一提的是，当初始化时用的工作队列为LinkedBlockingDeque时，这个值将无效。

1. **keepAliveTime：**存活时间，

当非核心空闲超过这个时间将被回收，同时空闲核心线程是否回收受allowCoreThreadTimeOut影响。

1. **unit：**keepAliveTime的单位。
2. **workQueue：**任务队列

常用有三种队列，即SynchronousQueue,LinkedBlockingDeque（无界队列）,ArrayBlockingQueue（有界队列）。

1. **threadFactory：**线程工厂，

ThreadFactory是一个接口，用来创建worker。通过线程工厂可以对线程的一些属性进行定制。默认直接新建线程。

1. **RejectedExecutionHandler：**拒绝策略

也是一个接口，只有一个方法，当线程池中的资源已经全部使用，添加新线程被拒绝时，会调用RejectedExecutionHandler的rejectedExecution法。默认是抛出一个运行时异常。

**线程池大小设置：**

1. 需要分析线程池执行的任务的特性： CPU 密集型还是 IO 密集型
2. 每个任务执行的平均时长大概是多少，这个任务的执行时长可能还跟任务处理逻辑是否涉及到网络传输以及底层系统资源依赖有关系

如果是 CPU 密集型，主要是执行计算任务，响应时间很快，cpu 一直在运行，这种任务 cpu的利用率很高，那么线程数的配置应该根据 CPU 核心数来决定，CPU 核心数=最大同时执行线程数，加入 CPU 核心数为 4，那么服务器最多能同时执行 4 个线程。过多的线程会导致上下文切换反而使得效率降低。那线程池的最大线程数可以配置为 cpu 核心数+1 如果是 IO 密集型，主要是进行 IO 操作，执行 IO 操作的时间较长，这是 cpu 出于空闲状态，导致 cpu 的利用率不高，这种情况下可以增加线程池的大小。这种情况下可以结合线程的等待时长来做判断，等待时间越高，那么线程数也相对越多。一般可以配置 cpu 核心数的 2 倍。

一个公式：线程池设定最佳线程数目 = （（线程池设定的线程等待时间+线程 CPU 时间）/  
线程 CPU 时间 ）\* CPU 数目

这个公式的线程 cpu 时间是预估的程序单个线程在 cpu 上运行的时间（通常使用 loadrunner测试大量运行次数求出平均值）

**拒绝策略：**

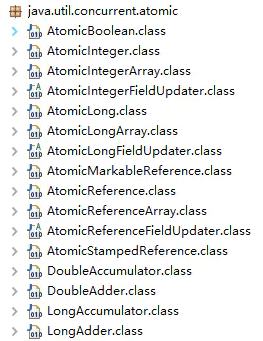
1. AbortPolicy：直接抛出异常，默认策略；
2. CallerRunsPolicy：用调用者所在的线程来执行任务；
3. DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；
4. DiscardPolicy：直接丢弃任务；当然也可以根据应用场景实现 RejectedExecutionHandler 接口，自定义饱和策略，如记录日志或持久化存储不能处理的任务

### 10、常见线程安全的并发容器有哪些 难度系数：⭐

* CopyOnWriteArrayList、CopyOnWriteArraySet、ConcurrentHashMap
* CopyOnWriteArrayList、CopyOnWriteArraySet采用写时复制实现线程安全
* ConcurrentHashMap采用分段锁的方式实现线程安全

### 11、Atomic原子类了解多少 原理是什么 难度系数：⭐

Java 的原子类都存放在并发包 java.util.concurrent.atomic下，如下图



**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

**引用类型**

* AtomicReference：引用类型原子类
* AtomicStampedReference：原子更新引用类型里的字段原子类
* AtomicMarkableReference ：原子更新带有标记位的引用类型
* 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
* 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，以及解决使用 CAS 进行原子更新时可能出现的 ABA 问题

1. AtomicInteger 类利用 CAS (Compare and Swap) + volatile + 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。
2. CAS 的原理，是拿期望值和原本的值作比较，如果相同，则更新成新的值。UnSafe 类的 objectFieldOffset() 方法是个本地方法，这个方法是用来拿“原值”的内存地址，返回值是 valueOffset；另外，value 是一个 volatile 变量，因此 JVM 总是可以保证任意时刻的任何线程总能拿到该变量的最新值。

## synchronized底层实现是什么 lock底层是什么 有什么区别

### 难度系数：⭐⭐⭐

**Synchronized原理：**

**方法级的同步是隐式，即无需通过字节码指令来控制的，它实现在方法调用和返回操作之中。**JVM可以从方法常量池中的方法表结构(method\_info Structure) 中的 ACC\_SYNCHRONIZED 访问标志区分一个方法是否同步方法。当方法调用时，调用指令将会 检查方法的 ACC\_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先持有monitor（虚拟机规范中用的是管程一词），然后再执行方法，最后再方法完成(无论是正常完成还是非正常完成)时释放monitor。

**代码块的同步是利用monitorenter和monitorexit这两个字节码指令。它们分别位于同步代码块的开始和结束位置**。当jvm执行到monitorenter指令时，当前线程试图获取monitor对象的所有权，如果未加锁或者已经被当前线程所持有，就把锁的计数器+1；当执行monitorexit指令时，锁计数器-1；当锁计数器为0时，该锁就被释放了。如果获取monitor对象失败，该线程则会进入阻塞状态，直到其他线程释放锁。

参考：<https://blog.csdn.net/ben040661/article/details/125697819>

**Lock原理：**

1. Lock的存储结构：一个int类型状态值（用于锁的状态变更），一个双向链表（用于存储等待中的线程）
2. Lock获取锁的过程：本质上是通过CAS来获取状态值修改，如果当场没获取到，会将该线程放在线程等待链表中。
3. Lock释放锁的过程：修改状态值，调整等待链表。
4. Lock大量使用CAS+自旋。因此根据CAS特性，lock建议使用在低锁冲突的情况下。

**Lock与synchronized的区别：**

1. Lock的加锁和解锁都是由java代码配合native方法（调用操作系统的相关方法）实现的，而synchronize的加锁和解锁的过程是由JVM管理的
2. 当一个线程使用synchronize获取锁时，若锁被其他线程占用着，那么当前只能被阻塞，直到成功获取锁。而Lock则提供超时锁和可中断等更加灵活的方式，在未能获取锁的 条件下提供一种退出的机制。
3. 一个锁内部可以有多个Condition实例，即有多路条件队列，而synchronize只有一路条件队列；同样Condition也提供灵活的阻塞方式，在未获得通知之前可以通过中断线程以 及设置等待时限等方式退出条件队列。
4. synchronize对线程的同步仅提供独占模式，而Lock即可以提供独占模式，也可以提供共享模式

|  |  |
| --- | --- |
| synchronized | Lock |
| 关键字 | 类 |
| 自动加锁和释放锁 | 需要手动调用unlock方法释放锁 |
| jvm层面的锁 | API层面的锁 |
| 非公平锁 | 可以选择公平或者非公平锁 |
| 锁是一个对象,并且锁的信息保存在了对象中 | 代码中通过int类型的state标识 |
| 有一个锁升级的过程 | 无 |

## 15、了解ConcurrentHashMap吗 为什么性能比HashTable高，说下原理 难度系数：⭐⭐

ConcurrentHashMap是线程安全的Map容器，JDK8之前，ConcurrentHashMap使用锁分段技术，将数据分成一段段存储，每个数据段配置一把锁，即segment类，这个类继承ReentrantLock来保证线程安全，JKD8的版本取消Segment这个分段锁数据结构，底层也是使用Node数组+链表+红黑树，从而实现对每一段数据就行加锁，也减少了并发冲突的概率。

hashtable类基本上所有的方法都是采用synchronized进行线程安全控制，高并发情况下效率就降低 ，ConcurrentHashMap是采用了分段锁的思想提高性能，锁粒度更细化

## 16、ConcurrentHashMap底层原理 难度系数：⭐⭐⭐

**Java7 中 ConcurrentHashMap 使用的分段锁，也就是每一个 Segment 上同时只有一个线程可以操作，每一个 Segment 都是一个类似 HashMap 数组的结构，它可以扩容，它的冲突会转化为链表。但是 Segment 的个数一但初始化就不能改变。**

public V put(K key, V value) {  
 Segment<K,V> s;  
 if (value == null)  
 throw new NullPointerException();  
 int hash = hash(key);  
 // hash 值无符号右移 28位（初始化时获得），然后与 segmentMask=15 做与运算  
 // 其实也就是把高4位与segmentMask（1111）做与运算  
  
 // this.segmentMask = ssize - 1;  
 //对hash值进行右移segmentShift位，计算元素对应segment中数组下表的位置  
 //把hash右移segmentShift，相当于只要hash值的高32-segmentShift位，右移的目的是保留了hash值的高位。然后和segmentMask与操作计算元素在segment数组中的下表  
 int j = (hash >>> segmentShift) & segmentMask;  
 //使用unsafe对象获取数组中第j个位置的值，后面加上的是偏移量  
 if ((s = (Segment<K,V>)UNSAFE.getObject // nonvolatile; recheck  
 (segments, (j << SSHIFT) + SBASE)) == null) // in ensureSegment  
 // 如果查找到的 Segment 为空，初始化  
 s = ensureSegment(j);  
 //插入segment对象  
 return s.put(key, hash, value, false);  
}  
  
/\*\*  
 \* Returns the segment for the given index, creating it and  
 \* recording in segment table (via CAS) if not already present.  
 \*  
 \* @param k the index  
 \* @return the segment  
 \*/  
@SuppressWarnings("unchecked")  
private Segment<K,V> ensureSegment(int k) {  
 final Segment<K,V>[] ss = this.segments;  
 long u = (k << SSHIFT) + SBASE; // raw offset  
 Segment<K,V> seg;  
 // 判断 u 位置的 Segment 是否为null  
 if ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u)) == null) {  
 Segment<K,V> proto = ss[0]; // use segment 0 as prototype  
 // 获取0号 segment 里的 HashEntry<K,V> 初始化长度  
 int cap = proto.table.length;  
 // 获取0号 segment 里的 hash 表里的扩容负载因子，所有的 segment 的 loadFactor 是相同的  
 float lf = proto.loadFactor;  
 // 计算扩容阀值  
 int threshold = (int)(cap \* lf);  
 // 创建一个 cap 容量的 HashEntry 数组  
 HashEntry<K,V>[] tab = (HashEntry<K,V>[])new HashEntry[cap];  
 if ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u)) == null) { // recheck  
 // 再次检查 u 位置的 Segment 是否为null，因为这时可能有其他线程进行了操作  
 Segment<K,V> s = new Segment<K,V>(lf, threshold, tab);  
 // 自旋检查 u 位置的 Segment 是否为null  
 while ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u))  
 == null) {  
 // 使用CAS 赋值，只会成功一次  
 if (UNSAFE.compareAndSwapObject(ss, u, null, seg = s))  
 break;  
 }  
 }  
 }  
 return seg;  
}  
  
final V put(K key, int hash, V value, boolean onlyIfAbsent) {  
 // 获取 ReentrantLock 独占锁，获取不到，scanAndLockForPut 获取。  
 HashEntry<K,V> node = tryLock() ? null : scanAndLockForPut(key, hash, value);  
 V oldValue;  
 try {  
 HashEntry<K,V>[] tab = table;  
 // 计算要put的数据位置  
 int index = (tab.length - 1) & hash;  
 // CAS 获取 index 坐标的值  
 HashEntry<K,V> first = entryAt(tab, index);  
 for (HashEntry<K,V> e = first;;) {  
 if (e != null) {  
 // 检查是否 key 已经存在，如果存在，则遍历链表寻找位置，找到后替换 value  
 K k;  
 if ((k = e.key) == key ||  
 (e.hash == hash && key.equals(k))) {  
 oldValue = e.value;  
 if (!onlyIfAbsent) {  
 e.value = value;  
 ++modCount;  
 }  
 break;  
 }  
 e = e.next;  
 }  
 else {  
 // first 有值没说明 index 位置已经有值了，有冲突，链表头插法。  
 if (node != null)  
 node.setNext(first);  
 else  
 node = new HashEntry<K,V>(hash, key, value, first);  
 int c = count + 1;  
 // 容量大于扩容阀值，小于最大容量，进行扩容  
 if (c > threshold && tab.length < MAXIMUM\_CAPACITY)  
 rehash(node);  
 else  
 // index 位置赋值 node，node 可能是一个元素，也可能是一个链表的表头  
 setEntryAt(tab, index, node);  
 ++modCount;  
 count = c;  
 oldValue = null;  
 break;  
 }  
 }  
 } finally {  
 unlock();  
 }  
 return oldValue;  
}

Java

1. Java8 中的 ConcurrentHashMap 使用的 Synchronized 锁加 CAS 的机制。结构Node 数组 + 链表 / 红黑树，Node 是类似于一个 HashEntry 的结构。它的冲突再达到一定大小时会转化成红黑树，在冲突小于一定数量时又退回链表。

public V put(K key, V value) {  
 return putVal(key, value, false);  
}  
  
/\*\* Implementation for put and putIfAbsent \*/  
final V putVal(K key, V value, boolean onlyIfAbsent) {  
 // key 和 value 不能为空  
 if (key == null || value == null) throw new NullPointerException();  
 int hash = spread(key.hashCode());  
 int binCount = 0;  
 for (Node<K,V>[] tab = table;;) {  
 // f = 目标位置元素  
 Node<K,V> f; int n, i, fh;// fh 后面存放目标位置的元素 hash 值  
 if (tab == null || (n = tab.length) == 0)  
 // 数组桶为空，初始化数组桶（自旋+CAS)  
 tab = initTable();  
 else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {  
 // 桶内为空，CAS 放入，不加锁，成功了就直接 break 跳出  
 if (casTabAt(tab, i, null,new Node<K,V>(hash, key, value, null)))  
 break; // no lock when adding to empty bin  
 }  
 else if ((fh = f.hash) == MOVED)  
 tab = helpTransfer(tab, f);  
 else {  
 V oldVal = null;  
 // 使用 synchronized 加锁加入节点  
 synchronized (f) {  
 if (tabAt(tab, i) == f) {  
 // 说明是链表  
 if (fh >= 0) {  
 binCount = 1;  
 // 循环加入新的或者覆盖节点  
 for (Node<K,V> e = f;; ++binCount) {  
 K ek;  
 if (e.hash == hash &&  
 ((ek = e.key) == key ||  
 (ek != null && key.equals(ek)))) {  
 oldVal = e.val;  
 if (!onlyIfAbsent)  
 e.val = value;  
 break;  
 }  
 Node<K,V> pred = e;  
 if ((e = e.next) == null) {  
 pred.next = new Node<K,V>(hash, key,  
 value, null);  
 break;  
 }  
 }  
 }  
 else if (f instanceof TreeBin) {  
 // 红黑树  
 Node<K,V> p;  
 binCount = 2;  
 if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,  
 value)) != null) {  
 oldVal = p.val;  
 if (!onlyIfAbsent)  
 p.val = value;  
 }  
 }  
 }  
 }  
 if (binCount != 0) {  
 if (binCount >= TREEIFY\_THRESHOLD)  
 treeifyBin(tab, i);  
 if (oldVal != null)  
 return oldVal;  
 break;  
 }  
 }  
 }  
 addCount(1L, binCount);  
 return null;  
}

Java

## 17、是否了解volatile关键字 难度系数：⭐

* volatile是Java提供的最轻量级的同步机制，保证了共享变量的可见性，被volatile关键字修饰的变量，如果值发生了变化，其他线程立刻可见，避免出现脏读现象。
* volatile禁止了指令重排，可以保证程序执行的有序性，但是由于禁止了指令重排，所以JVM相关的优化没了，效率会偏弱

## 18、synchronized和volatile有什么区别 难度系数：⭐⭐

* volatile本质是告诉JVM当前变量在寄存器中的值是不确定的，需要从主存中读取，synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。
* volatile仅能用在变量级别，而synchronized可以使用在变量、方法、类级别。
* volatile仅能实现变量的修改可见性，不能保证原子性；而synchronized则可以保证变量的修改可见性和原子性。
* volatile不会造成线程阻塞，synchronized可能会造成线程阻塞。
* volatile标记的变量不会被编译器优化，synchronized标记的变量可以被编译器优化。

## 19、Java类加载过程 难度系数：⭐

1. **加载** 加载时类加载的第一个过程，在这个阶段，将完成一下三件事情：

通过一个类的全限定名获取该类的二进制流。

将该二进制流中的静态存储结构转化为方法去运行时数据结构。

在内存中生成该类的Class对象，作为该类的数据访问入口。

**(2)验证** 验证的目的是为了确保Class文件的字节流中的信息不回危害到虚拟机.在该阶段主要完成以下四钟验证:

文件格式验证：验证字节流是否符合Class文件的规范，如主次版本号是否在当前虚拟机范围内，常量池中的常量是否有不被支持的类型.

元数据验证:对字节码描述的信息进行语义分析，如这个类是否有父类，是否集成了不被继承的类等。

字节码验证：是整个验证过程中最复杂的一个阶段，通过验证数据流和控制流的分析，确定程序语义是否正确，主要针对方法体的验证。如：方法中的类型转换是否正确，跳转指令是否正确等。

符号引用验证：这个动作在后面的解析过程中发生，主要是为了确保解析动作能正确执行。

* 准备

准备阶段是为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量的内存，实例变量将会在对象实例化时随着对象一起分配在Java堆中。

* 解析

该阶段主要完成符号引用到直接引用的转换动作。解析动作并不一定在初始化动作完成之前，也有可能在初始化之后。

* 初始化

初始化时类加载的最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的Java程序代码。

## 20、什么是类加载器，类加载器有哪些 难度系数：⭐

类加载器就是把类文件加载到虚拟机中，也就是说通过一个类的全限定名来获取描述该类的二进制字节流。

1. **主要有以下四种类加载器**

启动类加载器(Bootstrap ClassLoader)用来加载java核心类库，无法被java程序直接引用

扩展类加载器(extension class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类

系统类加载器（system class loader）也叫应用类加载器：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它

用户自定义类加载器，通过继承 java.lang.ClassLoader类的方式实现

1. **什么时候会使用到加载器？**

**Java中的加载器是按需加载，什么时候用到，什么时候加载**

* + new对象的时候
  + 访问某个类或者接口的静态变量，或者对该静态变量赋值时
  + 调用类的静态方法时
  + 反射
  + 初始化一个类的子类时，其父类首先会被加载
  + JVM启动时标明的启动类，也就是文件名和类名相同的那个类

## 21、简述java内存分配与回收策略以及Minor GC和Major GC（full GC） 难度系数：⭐⭐

* **内存分配**

**栈区**：栈分为java虚拟机栈和本地方法栈

**堆区**：堆被所有线程共享区域，在虚拟机启动时创建，唯一目的存放对象实例。堆区是gc的主要区域，通常情况下分为两个区块年轻代和年老代。更细一点年轻代又分为Eden区，主要放新创建对象，From survivor 和 To survivor 保存gc后幸存下的对象，默认情况下各自占比 8:1:1。

**方法区**：被所有线程共享区域，用于存放已被虚拟机加载的类信息，常量，静态变量等数据。被Java虚拟机描述为堆的一个逻辑部分。习惯是也叫它永久代（permanment generation）

**程序计数器**：当前线程所执行的行号指示器。通过改变计数器的值来确定下一条指令，比如循环，分支，跳转，异常处理，线程恢复等都是依赖计数器来完成。线程私有的。

* **回收策略以及Minor GC和Major GC**
  + 对象优先在堆的Eden区分配
  + 大对象直接进入老年代
  + 长期存活的对象将直接进入老年代

当Eden区没有足够的空间进行分配时，虚拟机会执行一次Minor GC.Minor GC通常发生在新生代的Eden区，在这个区的对象生存期短，往往发生GC的频率较高，回收速度比较快;Full Gc/Major GC 发生在老年代，一般情况下，触发老年代GC的时候不会触发Minor GC,但是通过配置，可以在Full GC之前进行一次Minor GC这样可以加快老年代的回收速度。

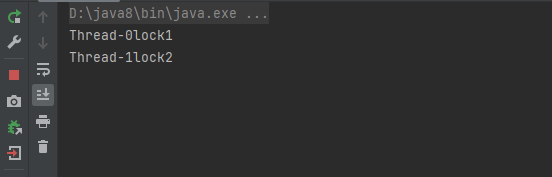
## 22、如何查看java死锁 难度系数：⭐

####演示死锁  
package com.ssg.mst;  
public class 死锁 {  
  
 private static final String lock1 = "lock1";  
 private static final String lock2 = "lock2";  
 public static void main(String[] args) {  
 Thread thread1 = new Thread(() -> {  
 while (true) {  
 synchronized (lock1) {  
 try {  
 System.out.println(Thread.currentThread().getName() + lock1);  
 Thread.sleep(1000);  
 synchronized (lock2){  
 System.out.println(Thread.currentThread().getName() + lock2);  
 }  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
 }  
 });  
  
 Thread thread2 = new Thread(() -> {  
 while (true) {  
 synchronized (lock2) {  
 try {  
 System.out.println(Thread.currentThread().getName() + lock2);  
 Thread.sleep(1000);  
 synchronized (lock1){  
 System.out.println(Thread.currentThread().getName() + lock1);  
 }  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
 }  
 });  
  
 thread1.start();  
 thread2.start();  
 }  
}  
 死锁代码演示

Java

死锁代码演示

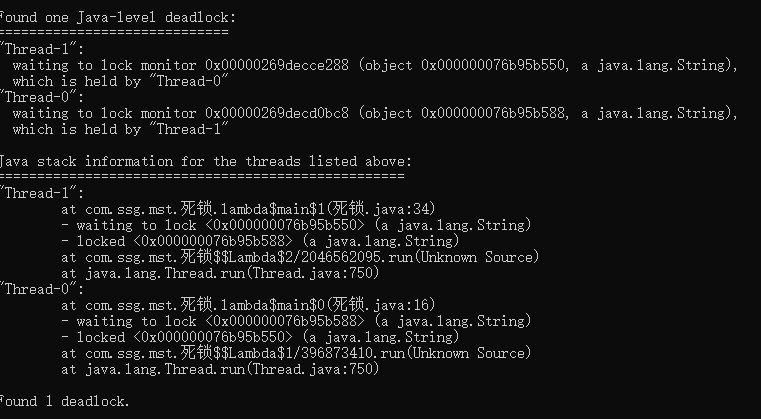
1. 程序运行，进程没有停止。



1. 通过jps查看java进程，找到没有停止的进程



1. 通过jstack 9060 查看进程具体执行信息



## 23、Java死锁如何避免 难度系数：⭐

造成死锁的几个原因

1. 一个资源每次只能被一个线程使用
2. 一个线程在阻塞等待某个资源时，不释放已占有资源
3. 一个线程已经获得的资源，在未使用完之前，不能被强行剥夺
4. 若干线程形成头尾相接的循环等待资源关系

这是造成死锁必须要达到的4个条件，如果要避免死锁，只需要不满足其中某一个条件即可。而其中前3个条件是作为锁要符合的条件，所以要避免死锁就需要打破第4个条件，不出现循环等待锁的关系。

**在开发过程中**

1.要注意加锁顺序，保证每个线程按同样的顺序进行加锁

2.要注意加锁时限，可以针对锁设置一个超时时间

3.要注意死锁检查，这是一种预防机制，确保在第一时间发现死锁并进行解决

第三章Java框架篇

## 1、简单的谈一下SpringMVC的工作流程 难度系数：⭐

* 用户发送请求至前端控制器DispatcherServlet
* DispatcherServlet收到请求调用HandlerMapping处理器映射器。
* 处理器映射器找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。
* DispatcherServlet调用HandlerAdapter处理器适配器
* HandlerAdapter经过适配调用具体的处理器(Controller，也叫后端控制器)。
* Controller执行完成返回ModelAndView
* HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet
* DispatcherServlet将ModelAndView传给ViewReslover视图解析器
* ViewReslover解析后返回具体View
* DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）。
* DispatcherServlet响应用户

## 2、说出Spring或者SpringMVC中常用的5个注解 难度系数：⭐

@Component 基本注解，标识一个受Spring管理的组件

@Controller 标识为一个表示层的组件

@Service 标识为一个业务层的组件

@Repository 标识为一个持久层的组件

@Autowired 自动装配

@Qualifier("") 具体指定要装配的组件的id值

@RequestMapping() 完成请求映射

@PathVariable 映射请求URL中占位符到请求处理方法的形参

只要说出几个注解并解释含义即可，如上答案只做参考

### 3、简述SpringMVC中如何返回JSON数据 难度系数：⭐

**Step1：**在项目中加入json转换的依赖，例如jackson，fastjson，gson等

**Step2：**在请求处理方法中将返回值改为具体返回的数据的类型， 例如数据的集合类List<Employee>等

**Step3：**在请求处理方法上使用@ResponseBody注解

### 4、谈谈你对Spring的理解 难度系数：⭐

Spring 是一个开源框架，为简化企业级应用开发而生。Spring 可以是使简单的JavaBean 实现以前只有EJB 才能实现的功能。Spring 是一个 IOC 和 AOP 容器框架。

**Spring 容器的主要核心是：**

**控制反转（IOC）**，传统的 java 开发模式中，当需要一个对象时，我们会自己使用 new 或者 getInstance 等直接或者间接调用构造方法创建一个对象。而在 spring 开发模式中，spring 容器使用了工厂模式为我们创建了所需要的对象，不需要我们自己创建了，直接调用spring 提供的对象就可以了，这是控制反转的思想。

**依赖注入（DI）**，spring 使用 javaBean 对象的 set 方法或者带参数的构造方法为我们在创建所需对象时将其属性自动设置所需要的值的过程，就是依赖注入的思想。

**面向切面编程（AOP）**，在面向对象编程（oop）思想中，我们将事物纵向抽成一个个的对象。而在面向切面编程中，我们将一个个的对象某些类似的方面横向抽成一个切面，对这个切面进行一些如权限控制、事物管理，记录日志等公用操作处理的过程就是面向切面编程的思想。AOP 底层是动态代理，如果是接口采用 JDK 动态代理，如果是类采用CGLIB 方式实现动态代理。

### 5、Spring中常用的设计模式 难度系数：⭐

1. 代理模式——spring 中两种代理方式，若目标对象实现了若干接口，spring 使用jdk 的java.lang.reflect.Proxy类代理。若目标兑现没有实现任何接口，spring 使用 CGLIB 库生成目标类的子类。
2. 单例模式——在 spring 的配置文件中设置 bean 默认为单例模式。
3. 模板方式模式——用来解决代码重复的问题。
4. 比如：RestTemplate、JmsTemplate、JpaTemplate
5. 工厂模式——在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用同一个接口来指向新创建的对象。Spring 中使用 beanFactory 来创建对象的实例。

### 6、Spring循环依赖问题 难度系数：⭐⭐ 常见问法

请解释一下spring中的三级缓存

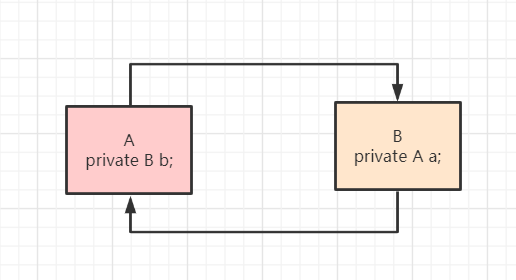
三级缓存分别是什么?三个Map有什么异同?

什么是循环依赖?请你谈谈?看过spring源码吗?

如何检测是否存在循环依赖?实际开发中见过循环依赖的异常吗?

多例的情况下,循环依赖问题为什么无法解决?

什么是循环依赖?



**两种注入方式对循环依赖的影响?**

**官方解释**

[https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-dependency-resolution](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html)

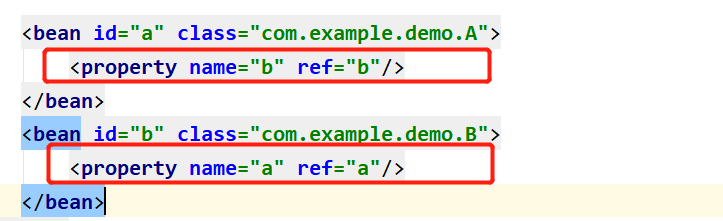
相关概念

实例化:堆内存中申请空间



初始化:对象属性赋值

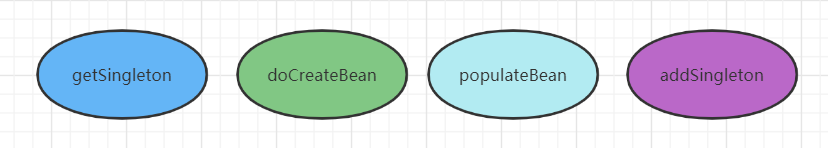




三级缓存

|  |  |  |
| --- | --- | --- |
| 名称 | 对象名 | 含义 |
| 一级 缓存 | singletonObjects | 存放已经经历了完整生命周期的Bean对象 |
| 二级 缓存 | earlySingletonObjects | 存放早期暴露出来的Bean对象，Bean的生命周期未结束（属性还未填充完) |
| 三级 缓存 | singletonFactories | 存放可以生成Bean的工厂 |

四个关键方法



package org.springframework.beans.factory.support;

public class DefaultSingletonBeanRegistry extends SimpleAliasRegistry implements SingletonBeanRegistry {

/\*\*

单例对象的缓存:bean名称—bean实例，即:所谓的单例池。

表示已经经历了完整生命周期的Bean对象

**第一级缓存**

\*/

private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);

/\*\*

早期的单例对象的高速缓存: bean名称—bean实例。

表示 Bean的生命周期还没走完（Bean的属性还未填充）就把这个 Bean存入该缓存中也就是实例化但未初始化的 bean放入该缓存里

**第二级缓存**

\*/

private final Map<String, Object> earlySingletonObjects = new HashMap<>(16);

/\*\*

单例工厂的高速缓存:bean名称—ObjectFactory

表示存放生成 bean的工厂

**第三级缓存**

\*/

private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<>(16);

}

debug源代码过程

需要22个断点(可选)

1，A创建过程中需要B，于是A将自己放到三级缓里面，去实例化B

2，B实例化的时候发现需要A，于是B先查一级缓存，没有，再查二级缓存，还是没有，再查三级缓存，找到了A然后把三级缓存里面的这个A放到二级缓存里面，并删除三级缓存里面的A

3，B顺利初始化完毕，将自己放到一级缓存里面(此时B里面的A依然是创建中状态)

然后回来接着创建A，此时B已经创建结束，直接从一级缓存里面拿到B，然后完成创建，并将A自己放到一级缓存里面。

**总结**

1，Spring创建 bean主要分为两个步骤，创建原始bean对象，接着去填充对象属性和初始化。

2，每次创建 bean之前，我们都会从缓存中查下有没有该bean，因为是单例，只能有一个。

3，当创建 A的原始对象后，并把它放到三级缓存中，接下来就该填充对象属性了，这时候发现依赖了B，接着就又去创建B，同样的流程，创建完B填充属性时又发现它依赖了A又是同样的流程，不同的是：这时候可以在三级缓存中查到刚放进去的原始对象A。

所以不需要继续创建，用它注入 B，完成 B的创建既然 B创建好了，所以 A就可以完成填充属性的步骤了，接着执行剩下的逻辑，闭环完成

Spring解决循环依赖依靠的是Bean的"中间态"这个概念，而这个中间态指的是已经实例化但还没初始化的状态—>半成品。实例化的过程又是通过构造器创建的，如果A还没创建好出来怎么可能提前曝光，所以构造器的循环依赖无法解决

其他衍生问题

问题1:为什么构造器注入属性无法解决循环依赖问题?

       由于spring中的bean的创建过程为先实例化 再初始化(在进行对象实例化的过程中不必赋值)将实例化好的对象暴露出去,供其他对象调用,然而使用构造器注入,必须要使用构造器完成对象的初始化的操作,就会陷入死循环的状态

问题2:一级缓存能不能解决循环依赖问题? 不能

       在三个级别的缓存中存储的对象是有区别的 一级缓存为完全实例化且初始化的对象 二级缓存实例化但未初始化对象 如果只有一级缓存,如果是并发操作下,就有可能取到实例化但未初始化的对象,就会出现问题

问题3:二级缓存能不能解决循环依赖问题?

      理论上二级缓存可以解决循环依赖问题,但是需要注意,为什么需要在三级缓存中存储匿名内部类(ObjectFactory),原因在于 需要创建代理对象  eg:现有A类,需要生成代理对象 A是否需要进行实例化(需要) 在三级缓存中存放的是生成具体对象的一个匿名内部类,该类可能是代理类也可能是普通的对象,而使用三级缓存可以保证无论是否需要是代理对象,都可以保证使用的是同一个对象,而不会出现,一会儿使用普通bean 一会儿使用代理类

### 7、Spring bean 的生命周期、注入方式和作用域 难度系数：⭐

**Bean的生命周期**

（1）默认情况下，IOC容器中bean的生命周期分为五个阶段:

* 调用构造器 或者是通过工厂的方式创建Bean对象
* 给bean对象的属性注入值
* 调用初始化方法，进行初始化， 初始化方法是通过init-method来指定的.
* 使用
* IOC容器关闭时， 销毁Bean对象.

（2）当加入了Bean的后置处理器后，IOC容器中bean的生命周期分为七个阶段:

* 调用构造器 或者是通过工厂的方式创建Bean对象
* 给bean对象的属性注入值
* 执行Bean后置处理器中的 postProcessBeforeInitialization
* 调用初始化方法，进行初始化， 初始化方法是通过init-method来指定的.x
* 执行Bean的后置处理器中 postProcessAfterInitialization
* 使用
* IOC容器关闭时， 销毁Bean对象

只需要回答出第一点即可，第二点也回答可适当 加分。

**注入方式：**

通过 setter 方法注入

通过构造方法注入

**Bean的作用域**

总共有四种作用域:

* Singleton 单例的
* Prototype 原型的
* Request
* Session

### 8、请描述一下Spring 的事务管理 难度系数：⭐

（1）声明式事务管理的定义：用在 Spring 配置文件中声明式的处理事务来代替代码式的处理事务。这样的好处是，事务管理不侵入开发的组件，具体来说，业务逻辑对象就不会意识到正在事务管理之中，事实上也应该如此，因为事务管理是属于系统层面的服务，而不是业务逻辑的一部分，如果想要改变事务管理策划的话，也只需要在定义文件中重新配置即可，这样维护起来极其方便。

基于 TransactionInterceptor 的声明式事务管理：两个次要的属性： transactionManager，用来指定一个事务治理器， 并将具体事务相关的操作请托给它； 其他一个是 Properties 类型的transactionAttributes 属性，该属性的每一个键值对中，键指定的是方法名，方法名可以行使通配符， 而值就是表现呼应方法的所运用的事务属性。

（2）基于 @Transactional 的声明式事务管理：Spring 2.x 还引入了基于 Annotation 的体式格式，具体次要触及@Transactional 标注。@Transactional 可以浸染于接口、接口方法、类和类方法上。算作用于类上时，该类的一切public 方法将都具有该类型的事务属性。

（3）编程式事物管理的定义：在代码中显式挪用 beginTransaction()、commit()、rollback()等事务治理相关的方法， 这就是编程式事务管理。Spring 对事物的编程式管理有基于底层 API 的编程式管理和基于 TransactionTemplate 的编程式事务管理两种方式。

### 9、MyBatis中 #{}和${}的区别是什么 难度系数：⭐

#{}是预编译处理，${}是字符串替换；

Mybatis在处理#{}时，会将sql中的#{}替换为?号，调用PreparedStatement的set方法来赋值；

Mybatis在处理${}时，就是把${}替换成变量的值；

使用#{}可以有效的防止SQL注入，提高系统安全性。

### 10、Mybatis 中一级缓存与二级缓存 难度系数：⭐

1. MyBatis的缓存分为一级缓存和 二级缓存。

**一级缓存**是SqlSession级别的缓存，默认开启。

**二级缓存**是NameSpace级别(Mapper)的缓存，多个SqlSession可以共享，使用时需要进行配置开启。

1. 缓存的查找顺序：二级缓存 => 一级缓存 => 数据库

### 11、MyBatis如何获取自动生成的(主)键值 难度系数：⭐

在<insert>标签中使用 useGeneratedKeys和keyProperty 两个属性来获取自动生成的主键值。

示例:

<insert id=”insertname” usegeneratedkeys=”true” keyproperty=”id”>  
 insert into names (name) values (#{name})   
</insert>

Java

## 简述Mybatis的动态SQL，列出常用的6个标签及作用

### 难度系数：⭐

动态SQL是MyBatis的强大特性之一 基于功能强大的OGNL表达式。

动态SQL主要是来解决查询条件不确定的情况，在程序运行期间，根据提交的条件动态的完成查询

常用的标签:

<if> : 进行条件的判断

<where>：在<if>判断后的SQL语句前面添加WHERE关键字，并处理SQL语句开始位置的AND 或者OR的问题

<trim>：可以在SQL语句前后进行添加指定字符 或者去掉指定字符.

<set>: 主要用于修改操作时出现的逗号问题

<choose> <when> <otherwise>：类似于java中的switch语句.在所有的条件中选择其一

<foreach>：迭代操作

### 13、Mybatis 如何完成MySQL的批量操作 难度系数：⭐

MyBatis完成MySQL的批量操作主要是通过<foreach>标签来拼装相应的SQL语句

例如:

<insert\*\* id="insertBatch" >  
 insert into tbl\_employee(last\_name,email,gender,d\_id) values   
 <foreach\*\* collection="emps" item="curr\_emp" separator=","\*\*>  
 (#{curr\_emp.lastName},#{curr\_emp.email},#{curr\_emp.gender},#{curr\_emp.dept.id})   
 </foreach>  
</insert>

Java

### 14、谈谈怎么理解SpringBoot框架 难度系数：⭐⭐

Spring Boot 是 Spring 开源组织下的子项目，是 Spring 组件一站式解决方案，主要是简化了使用 Spring 的难度，简省了繁重的配置，提供了各种启动器，开发者能快速上手。



**Spring Boot的优点**

* 独立运行

Spring Boot而且内嵌了各种servlet容器，Tomcat、Jetty等，现在不再需要打成war包部署到容器中，Spring Boot只要打成一个可执行的jar包就能独立运行，所有的依赖包都在一个jar包内。

* 简化配置

spring-boot-starter-web启动器自动依赖其他组件，简少了maven的配置。除此之外，还提供了各种启动器，开发者能快速上手。

* 自动配置

Spring Boot能根据当前类路径下的类、jar包来自动配置bean，如添加一个spring-boot-starter-web启动器就能拥有web的功能，无需其他配置。

* 无代码生成和XML配置

Spring Boot配置过程中无代码生成，也无需XML配置文件就能完成所有配置工作，这一切都是借助于条件注解完成的，这也是Spring4.x的核心功能之一。

* 应用监控

Spring Boot提供一系列端点可以监控服务及应用，做健康检测。

**Spring Boot缺点：**

Spring Boot虽然上手很容易，但如果你不了解其核心技术及流程，所以一旦遇到问题就很棘手，而且现在的解决方案也不是很多，需要一个完善的过程。

## 15、Spring Boot 的核心注解是哪个 它主要由哪几个注解组成的 难度系数：⭐

启动类上面的注解是@SpringBootApplication，它也是 Spring Boot 的核心注解，主要组合包含了以下 3 个注解：

* @SpringBootConfiguration：组合了 @Configuration 注解，实现配置文件的功能。
* @EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项，
  + 如关闭数据源自动配置功能： @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。
* @ComponentScan：Spring组件扫描。

### 16、Spring Boot自动配置原理是什么 难度系数：⭐

注解 @EnableAutoConfiguration, @Configuration, @ConditionalOnClass 就是自动配置的核心，

首先它得是一个配置文件，其次根据类路径下是否有这个类去自动配置。

@EnableAutoConfiguration是实现自动配置的注解

@Configuration表示这是一个配置文件

具体参考文档：

<https://mp.weixin.qq.com/s?__biz=MzI3ODcxMzQzMw==&mid=2247484365&idx=1&sn=a4ab1d977d6b03bf122b4d596d7ee1ab&scene=21>

### 17、SpringBoot配置文件有哪些 怎么实现多环境配置 难度系数：⭐

Spring Boot 的核心配置文件是 application 和 bootstrap 配置文件。

application 配置文件这个容易理解，主要用于 Spring Boot 项目的自动化配置。

**bootstrap配置文件的特性：**

* bootstrap 由父 ApplicationContext 加载，比 applicaton 优先加载
* bootstrap 里面的属性不能被覆盖

**bootstrap 配置文件有以下几个应用场景：**

* 使用 Spring Cloud Config 配置中心时，这时需要在 bootstrap 配置文件中添加连接到配置中心的配置属性来加载外部配置中心的配置信息；
* 一些固定的不能被覆盖的属性；
* 一些加密/解密的场景；

提供多套配置文件，如：

applcation.properties  
application-dev.properties  
application-test.properties  
application-prod.properties

运行时指定具体的配置文件，具体请看这篇文章《[Spring Boot Profile 不同环境配置](https://mp.weixin.qq.com/s?__biz=MzI3ODcxMzQzMw==&mid=2247484369&idx=1&sn=1155fccb4fef1db88cb76fd17b1756d7&scene=21)》。

### 18、SpringBoot和SpringCloud是什么关系 难度系数：⭐

Spring Boot 是 Spring 的一套快速配置脚手架，可以基于Spring Boot 快速开发单个微服务，Spring Cloud是一个基于Spring Boot实现的开发工具；Spring Boot专注于快速、方便集成的单个微服务个体，Spring Cloud关注全局的服务治理框架； Spring Boot使用了默认大于配置的理念，很多集成方案已经帮你选择好了，能不配置就不配置，Spring Cloud很大的一部分是基于Spring Boot来实现，必须基于Spring Boot开发。

可以单独使用Spring Boot开发项目，但是Spring Cloud离不开 Spring Boot。

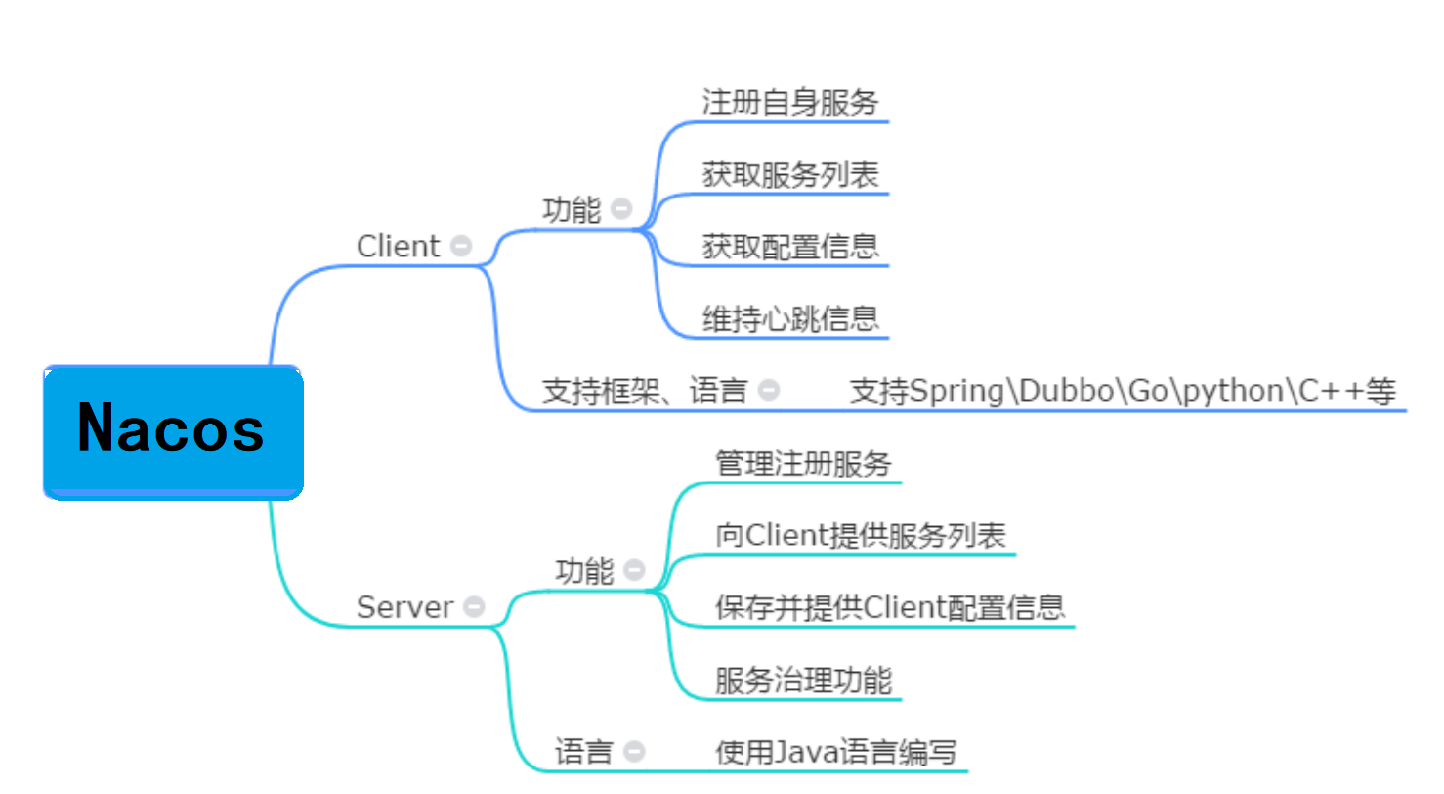
### 19、SpringCloud都用过哪些组件 介绍一下作用 难度系数：⭐

1. Nacos--作为注册中心和配置中心，实现服务注册发现和服务健康监测及配置信息统一管理
2. Gateway--作为网关，作为分布式系统统一的出入口，进行服务路由，统一鉴权等
3. OpenFeign--作为远程调用的客户端，实现服务之间的远程调用
4. Sentinel--实现系统的熔断限流
5. Sleuth--实现服务的链路追踪

### 20、Nacos作用以及注册中心的原理 难度系数：⭐⭐

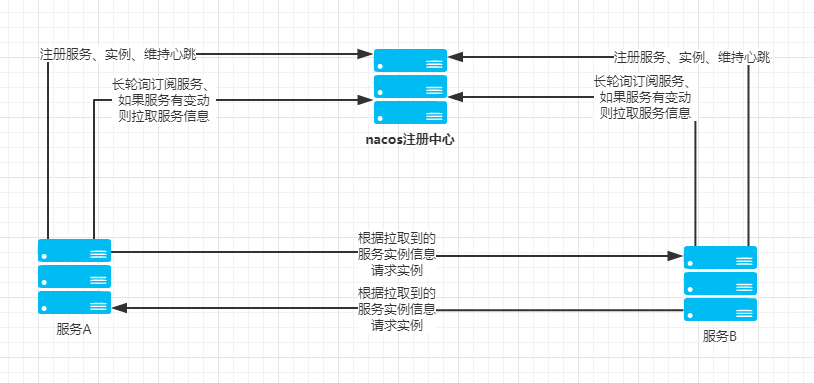
Nacos英文全称Dynamic Naming and Configuration Service，Na为naming/nameServer即注册中心,co为configuration即注册中心，service是指该注册/配置中心都是以服务为核心。

Nacos注册中心分为server与client，server采用Java编写，为client提供注册发现服务与配置服务。而client可以用多语言实现，client与微服务嵌套在一起，nacos提供sdk和openApi，如果没有sdk也可以根据openApi手动写服务注册与发现和配置拉取的逻辑。



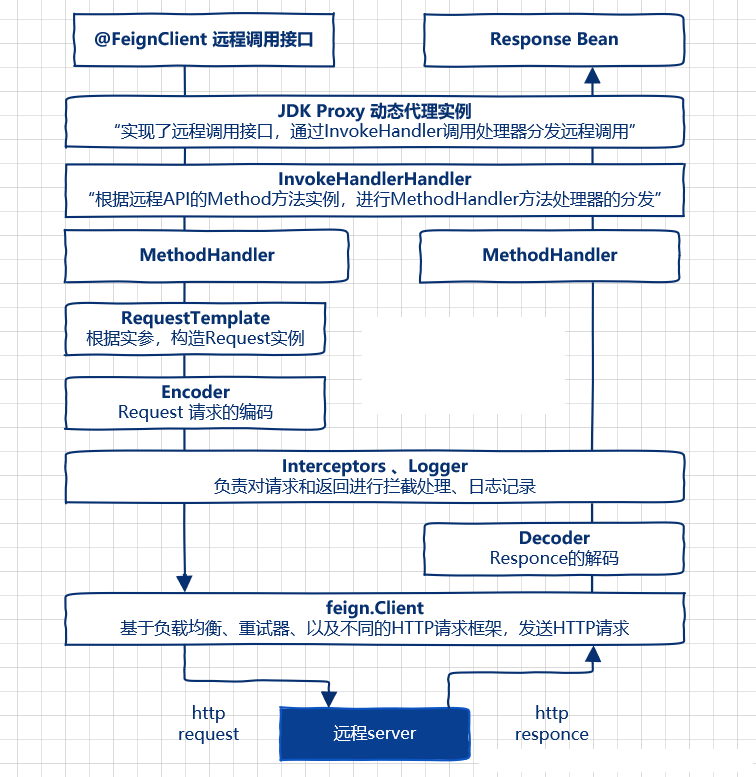
**服务注册原理**

服务注册方法：以Java nacos client v1.0.1 为例子，服务注册的策略的是每5秒向nacos server发送一次心跳，心跳带上了服务名，服务ip，服务端口等信息。同时 nacos server也会向client 主动发起健康检查，支持tcp/http检查。如果15秒内无心跳且健康检查失败则认为实例不健康，如果30秒内健康检查失败则剔除实例。



### 21、Feign工作原理 难度系数：⭐⭐

主程序入口添加了@EnableFeignClients注解开启对FeignClient扫描加载处理。根据Feign Client的开发规范，定义接口并加@FeignClient注解。当程序启动时，会进行包扫描，扫描所有@FeignClient的注解的类，并且讲这些信息注入Spring IOC容器中，当定义的的Feign接口中的方法被调用时，通过JDK的代理方式，来生成具体的RequestTemplate.当生成代理时，Feign会为每个接口方法创建一个RequestTemplate。当生成代理时，Feign会为每个接口方法创建一个RequestTemplate对象，该对象封装HTTP请求需要的全部信息，如请求参数名，请求方法等信息都是在这个过程中确定的。然后RequestTemplate生成Request,然后把Request交给Client去处理，这里指的时Client可以时JDK原生的URLConnection,Apache的HttpClient,也可以时OKhttp，最后Client被封装到LoadBalanceClient类，这个类结合Ribbon负载均衡发器服务之间的调用。



第四章 Mysql数据库

### 1、Select 语句完整的执行顺序 难度系数：⭐

SQL Select 语句完整的执行顺序：

（1）from 子句组装来自不同数据源的数据；

（2）where 子句基于指定的条件对记录行进行筛选；

（3）group by 子句将数据划分为多个分组；

（4）使用聚集函数进行计算；

（5）使用 having 子句筛选分组；

（6）计算所有的表达式；

（7）select 的字段；

（8）使用order by 对结果集进行排序。

### 2、MySQL事务 难度系数：⭐⭐

**事务的基本要素（ACID）**

* 原子性（Atomicity）：事务开始后所有操作，要么全部做完，要么全部不做，不可能停滞在中间环节。事务执行过程中出错，会回滚到事务开始前的状态，所有的操作就像没有发生一样。也就是说事务是一个不可分割的整体，就像化学中学过的原子，是物质构成的基本单位
* 一致性（Consistency）：事务开始前和结束后，数据库的完整性约束没有被破坏 。比如A向B转账，不可能A扣了钱，B却没收到。
* 隔离性（Isolation）：同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。比如A正在从一张银行卡中取钱，在A取钱的过程结束前，B不能向这张卡转账。
* 持久性（Durability）：事务完成后，事务对数据库的所有更新将被保存到数据库，不能回滚。

**MySQL事务隔离级别：**

|  |  |  |  |
| --- | --- | --- | --- |
| 事务隔离级别 | 脏读 | 不可重复读 | 幻读 |
| 读未提交（read-uncommitted） | 是 | 是 | 是 |
| 读提交（read-committed） | 否 | 是 | 是 |
| 可重复读（repeatable-read） | 否 | 否 | 是 |
| 串行化（serializable） | 否 | 否 | 否 |

**事务的并发问题**

* 脏读：事务A读取了事务B更新的数据，然后B回滚操作，那么A读取到的数据是脏数据
* 不可重复读：事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果 不一致
* 幻读：系统管理员A将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。

如何解决脏读、幻读、不可重复读

* 脏读： 隔离级别为 读提交、可重复读、串行化可以解决脏读
* 不可重复读：隔离级别为可重复读、串行化可以解决不可重复读
* 幻读：隔离级别为串行化可以解决幻读、通过MVCC + 区间锁可以解决幻读

**小结：**

不可重复读的和幻读很容易混淆，不可重复读侧重于修改，幻读侧重于新增或删除。解决不可重复读的问题只需锁住满足条件的行，解决幻读需要锁表

### 3、MyISAM和InnoDB的区别 难度系数：⭐

|  |  |  |
| --- | --- | --- |
|  | MyISAM | InnoDB |
| 事务 | 不支持 | 支持 |
| 锁 | 表锁 | 表锁、行锁 |
| 文件存储 | 3个 | 1个 |
| 外键 | 不支持 | 支持 |

### 4、悲观锁和乐观锁的怎么实现 难度系数：⭐⭐

**悲观锁：**select...for update是MySQL提供的实现悲观锁的方式。

例如：select price from item where id=100 for update

此时在items表中，id为100的那条数据就被我们锁定了，其它的要执行select price from items where id=100 for update的事务必须等本次事务提交之后才能执行。这样我们可以保证当前的数据不会被其它事务修改。MySQL有个问题是select...for update语句执行中所有扫描过的行都会被锁上，因此在MySQL中用悲观锁务必须确定走了索引，而不是全表扫描，否则将会将整个数据表锁住。

**乐观锁：**乐观锁相对悲观锁而言，它认为数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则让返回错误信息，让用户决定如何去做。

利用数据版本号（version）机制是乐观锁最常用的一种实现方式。一般通过为数据库表增加一个数字类型的 “version” 字段，当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值+1。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据，返回更新失败。

**举例：**

//1: 查询出商品信息

select (quantity,version) from items where id=100;

//2: 根据商品信息生成订单

insert into orders(id,item\_id) values(null,100);

//3: 修改商品的库存

update items set quantity=quantity-1,version=version+1 where id=100 and version=#{version};

### 5、聚簇索引与非聚簇索引区别 难度系数：⭐⭐

**都是B+树的数据结构**

**聚簇索引:** 将数据存储与索引放到了一块、并且是按照一定的顺序组织的，找到索引也就找到了数据，数据的物理存放顺序与索引顺序是一致的，即:只要索引是相邻的，那么对应的数据一定也是相邻地存放在磁盘上的

**非聚簇索引：** 叶子节点不存储数据、存储的是数据行地址，也就是说根据索引查找到数据行的位置再取磁盘查找数据，这个就有点类似一本书的目录，比如我们要找第三章第一节，那我们先在这个目录里面找，找到对应的页码后再去对应的页码看文章。

**优势:**

* 1、查询通过聚簇索引可以直接获取数据，相比非聚簇索引需要第二次查询(非覆盖索引的情况下)效率要高
* 2、聚簇索引对于范围查询的效率很高，因为其数据是按照大小排列的
* 3、聚簇索引适合用在排序的场合，非聚簇索引不适合

**劣势：**

* 维护索引很昂贵，特别是插入新行或者主键被更新导至要分页(pagesplit)的时候。建议在大量插入新行后，选在负载较低的时间段，通过OPTIMIZETABLE优化表，因为必须被移动的行数据可能造成碎片。使用独享表空间可以弱化碎片
* 表因为使用uuId(随机ID)作为主键，使数据存储稀疏，这就会出现聚簇索引有可能有比全表扫面更慢，所以建议使用int的auto\_increment作为主键
* 如果主键比较大的话，那辅助索引将会变的更大，因为辅助索引的叶子存储的是主键值，过长的主键值，会导致非叶子节点占用占用更多的物理空间

### 6、什么情况下mysql会索引失效 难度系数：⭐

**失效条件：**

* where 后面使用函数
* 使用or条件
* 模糊查询 %放在前边
* 类型转换
* 组合索引 （最佳左前缀匹配原则）

#查询条件用到了计算或者函数  
explain SELECT \* from test\_slow\_query where age = 20  
explain SELECT \* from test\_slow\_query where age +10 = 30  
  
#模糊查询  
EXPLAIN SELECT \* from test\_slow\_query where NAME like '%吕布'  
EXPLAIN SELECT \* from test\_slow\_query where NAME like '%吕布%'  
EXPLAIN SELECT \* from test\_slow\_query where NAME like '吕布&'  
  
#用到了or条件  
EXPLAIN SELECT \* from test\_slow\_query where NAME = '吕布' or name = "aaa"  
  
#类型不匹配查询  
explain SELECT \* from test\_slow\_query where NAME = 11  
explain SELECT \* from test\_slow\_query where NAME = '11'

SQL

### 7、B+tree 与 B-tree区别 难度系数：⭐⭐

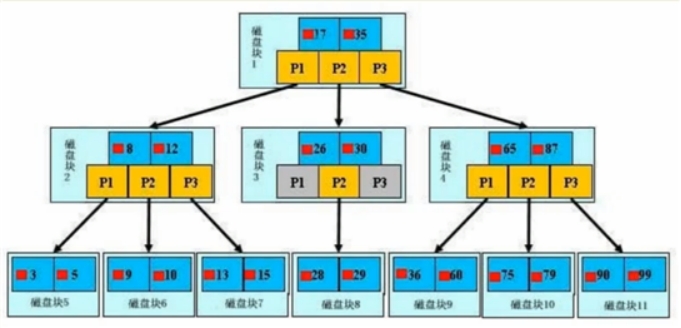
**原理:**分批次的将磁盘块加载进内存中进行检索,若查到数据,则直接返回,若查不到,则释放内存,并重新加载同等数据量的索引进内存,重新遍历

**结构:** 数据  向下的指针 指向数据的指针

**特点:**

        1，节点排序

        2 .一个节点了可以存多个元索，多个元索也排序了



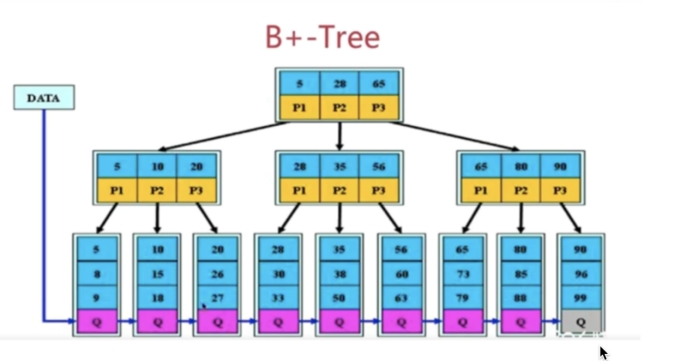
**结构:** 数据  向下的指针

**特点:**

    1.拥有B树的特点

    2.叶子节点之间有指针

    3.非叶子节点上的元素在叶子节点上都冗余了，也就是叶子节点中存储了所有的元素，并且排好顺序



从结构上看,B+Tree 相较于 B-Tree 而言  缺少了指向数据的指针 也就红色小方块;

Mysq|索引使用的是B+树，因为索引是用来加快查询的，而B+树通过对数据进行排序所以是可以提高查询速度的，然后通过一个节点中可以存储多个元素，从而可以使得B+树的高度不会太高，在Mysql中一个Innodb页就是一个B+树节点，一个Innodb页默认16kb，所以一般情况下一颗两层的B+树可以存2000万行左右的数据，然后通过利用B+树叶子节点存储了所有数据并且进行了排序，并且叶子节点之间有指针，可以很好的支持全表扫描，范围查找等SQL语句

文章推荐：<https://www.jianshu.com/p/b544d2e10726>

### 8、以MySQL为例Linux下如何排查问题 难度系数：⭐⭐

**类似提问方式: 如果线上环境出现问题比如网站卡顿重则瘫痪 如何是好?**

--->linux--->mysql/redis/nacos/sentinel/sluth--->可以从以上提到的技术点中选择一个自己熟悉单技术点进行分析

**以mysql为例**

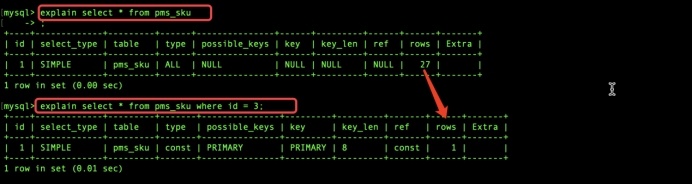
1,架构层面 是否使用主从

2,表结构层面 是否满足常规的表设计规范(大量冗余字段会导致查询会变得很复杂)

3,sql语句层面(⭐)

前提:由于慢查询日志记录默认是关闭的,所以开启数据库mysql的慢查询记录 的功能 从慢查询日志中去获取哪些sql语句时慢查询  默认10S ,从中获取到sql语句进行分析

*explain 分析一条sql*



Id:执行顺序 如果单表的话,无参考价值 如果是关联查询,会据此判断主表 从表

Select\_type:simple

Table:表

Type:  ALL 未创建索引 、const、 常量ref其他索引 、eq\_ref 主键索引、

Possible\_keys

Key  实际是到到索引到字段

Key\_len  索引字段数据结构所使用长度 与是否有默认值null 以及对应字段到数据类型有关，有一个理论值 有一个实际使用值也即key\_len的值

Rows  检索的行数 与查询返回的行数无关

Extra  常见的值：usingfilesort 使用磁盘排序算法进行排序，事关排序 分组 的字段是否使用索引的核心参考值

还可能这样去提问：sql语句中哪些位置适合建索引/索引建立在哪个位置

Select id,name,age from user where id=1 and name=”xxx” order by age

**总结:  查询字段  查询条件(最常用)   排序/分组字段**

**补充:如何判断是数据库的问题?可以借助于top命令**



### 9、如何处理慢查询 难度系数：⭐⭐

在业务系统中，除了使用主键进行的查询，其他的都会在测试库上测试其耗时，慢查询的统计主要由运维在做，会定期将业务中的慢查询反馈给我们。

慢查询的优化首先要搞明白慢的原因是什么?是查询条件没有命中索引?是加载了不需要的数据列?还是数据量太大?

所以优化也是针对这三个方向来的

首先分析语句，看看是否加载了额外的数据，可能是查询了多余的行并且抛弃掉了，可能是加载了许多结果中并不需要的列，对语句进行分析以及重写。

分析语句的执行计划，然后获得其使用索引的情况，之后修改语句或者修改索引，使得语句可以尽可能的命中索引。

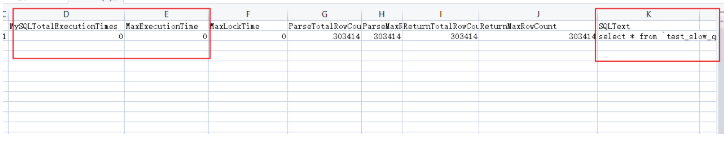
如果对语句的优化已经无法进行，可以考虑表中的数据量是否太大，如果是的话可以进行横向或者纵向的分表。

**具体处理流程 （阿里云RDS为例）**

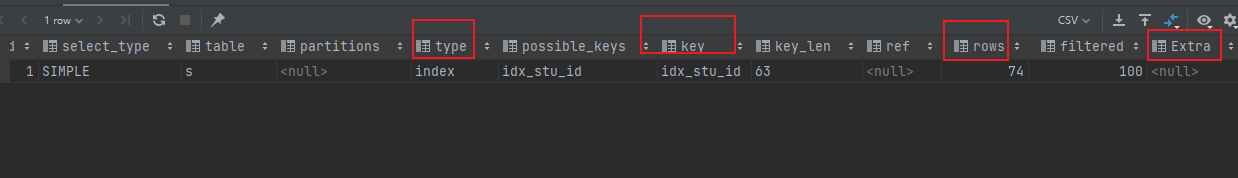
1. 开启慢查询设置



1. 日志管理导出慢查询文件



（3）测试环境通过explain执行sql，主要关心以下字段



type：连接类型

key： MYSQL使用的索引

rows：显示MYSQL执行查询的行数，简单且重要，数值越大越不好，说明没有用好索引

extra：该列包含MySQL解决查询的详细信息。

### 10、数据库分表操作 难度系数：⭐

**水平分表**

步长法：1000万一张表拆分

取模法：举例：根据用户id取模落入不能的表

**垂直分表：**大表拆小表。商品信息 spu\_info spu\_image ...

可以说使用Mycat或者ShardingSphere等中间件来做，具体怎么做就要结合具体的场景进行分析了。*可以参考：*<https://database.51cto.com/art/201809/583857.htm>

### 11、MySQL优化 难度系数：⭐

（1）尽量选择较小的列

（2）将where中用的比较频繁的字段建立索引

（3）select子句中避免使用‘\*’

（4）避免在索引列上使用计算、not in 和<>等操作

（5）当只需要一行数据的时候使用limit 1

（6）保证单表数据不超过200W，适时分割表。针对查询较慢的语句，可以使用explain 来分析该语句具体的执行情况。

（7）避免改变索引列的类型。

（8）选择最有效的表名顺序，from字句中写在最后的表是基础表，将被最先处理，在from子句中包含多个表的情况下，你必须选择记录条数最少的表作为基础表。

（9）避免在索引列上面进行计算。

（10）尽量缩小子查询的结果

### 12、SQL语句优化案例 难度系数：⭐

**例1：where 子句中可以对字段进行 null 值判断吗？**

可以，比如 select id from t where num is null 这样的 sql 也是可以的。但是最好不要给数据库留NULL，尽可能的使用 NOT NULL 填充数据库。不要以为 NULL 不需要空间，比如：char(100) 型，在字段建立时，空间就固定了， 不管是否插入值（NULL 也包含在内），都是占用 100 个字符的空间的，如果是 varchar 这样的变长字段，null 不占用空间。可以在 num 上设置默认值 0，确保表中 num 列没有 null 值，然后这样查询：select id from t where num= 0。

**例2：如何优化?下面的语句？**

select \* from admin left join log on admin.admin\_id = log.admin\_id where log.admin\_id>10

优化为：select \* from (select \* from admin where admin\_id>10) T1 lef join log on T1.admin\_id = log.admin\_id。

使用 JOIN 时候，应该用小的结果驱动大的结果（left join 左边表结果尽量小如果有条件应该放到左边先处理， right join 同理反向），同时尽量把牵涉到多表联合的查询拆分多个 query（多个连表查询效率低，容易到之后锁表和阻塞）。

**例3：limit 的基数比较大时使用 between**

例如：select \* from admin order by admin\_id limit 100000,10

优化为：select \* from admin where admin\_id between 100000 and 100010 order by admin\_id。

**例4：尽量避免在列上做运算，这样导致索引失效**

例如：select \* from admin where year(admin\_time)>2014

优化为： select \* from admin where admin\_time> '2014-01-01′

### 13、你们公司有哪些数据库设计规范 难度系数：

**（一）基础规范**

1、表存储引擎必须使用InnoD，表字符集默认使用utf8，必要时候使用utf8mb4

解读：

（1）通用，无乱码风险，汉字3字节，英文1字节

（2）utf8mb4是utf8的超集，有存储4字节例如表情符号时，使用它

2、禁止使用存储过程，视图，触发器，Event

解读：

（1）对数据库性能影响较大，互联网业务，能让站点层和服务层干的事情，不要交到数据库层

（2）调试，排错，迁移都比较困难，扩展性较差

3、禁止在数据库中存储大文件，例如照片，可以将大文件存储在对象存储系统，数据库中存储路径

4、禁止在线上环境做数据库压力测试

5、测试，开发，线上数据库环境必须隔离

**（二）命名规范**

1、库名，表名，列名必须用小写，采用下划线分隔

解读：abc，Abc，ABC都是给自己埋坑

2、库名，表名，列名必须见名知义，长度不要超过32字符

解读：tmp，wushan谁知道这些库是干嘛的

3、库备份必须以bak为前缀，以日期为后缀

4、从库必须以-s为后缀

5、备库必须以-ss为后缀

**（三）表设计规范**

1、单实例表个数必须控制在2000个以内

2、单表分表个数必须控制在1024个以内

3、表必须有主键，推荐使用UNSIGNED整数为主键

潜在坑：删除无主键的表，如果是row模式的主从架构，从库会挂住

4、禁止使用外键，如果要保证完整性，应由应用程式实现

解读：外键使得表之间相互耦合，影响update/delete等SQL性能，有可能造成死锁，高并发情况下容易成为数据库瓶颈

5、建议将大字段，访问频度低的字段拆分到单独的表中存储，分离冷热数据

**（四）列设计规范**

1、根据业务区分使用tinyint/int/bigint，分别会占用1/4/8字节

2、根据业务区分使用char/varchar

解读：

（1）字段长度固定，或者长度近似的业务场景，适合使用char，能够减少碎片，查询性能高

（2）字段长度相差较大，或者更新较少的业务场景，适合使用varchar，能够减少空间

3、根据业务区分使用datetime/timestamp

解读：前者占用5个字节，后者占用4个字节，存储年使用YEAR，存储日期使用DATE，存储时间使用datetime

4、必须把字段定义为NOT NULL并设默认值

解读：

（1）NULL的列使用索引，索引统计，值都更加复杂，MySQL更难优化

（2）NULL需要更多的存储空间

（3）NULL只能采用IS NULL或者IS NOT NULL，而在=/!=/in/not in时有大坑

5、使用INT UNSIGNED存储IPv4，不要用char(15)

6、使用varchar(20)存储手机号，不要使用整数

解读：

（1）牵扯到国家代号，可能出现+/-/()等字符，例如+86

（2）手机号不会用来做数学运算

（3）varchar可以模糊查询，例如like ‘138%’

7、使用TINYINT来代替ENUM

解读：ENUM增加新值要进行DDL操作

**（五）索引规范**

1、唯一索引使用uniq\_[字段名]来命名

2、非唯一索引使用idx\_[字段名]来命名

3、单张表索引数量建议控制在5个以内

解读：

（1）互联网高并发业务，太多索引会影响写性能

（2）生成执行计划时，如果索引太多，会降低性能，并可能导致MySQL选择不到最优索引

（3）异常复杂的查询需求，可以选择ES等更为适合的方式存储

4、组合索引字段数不建议超过5个

解读：如果5个字段还不能极大缩小row范围，八成是设计有问题

5、不建议在频繁更新的字段上建立索引

6、非必要不要进行JOIN查询，如果要进行JOIN查询，被JOIN的字段必须类型相同，并建立索引

解读：踩过因为JOIN字段类型不一致，而导致全表扫描的坑么？

7、理解组合索引最左前缀原则，避免重复建设索引，如果建立了(a,b,c)，相当于建立了(a), (a,b), (a,b,c)

**（六）SQL规范**

1、禁止使用select \*，只获取必要字段

解读：

（1）select \*会增加cpu/io/内存/带宽的消耗

（2）指定字段能有效利用索引覆盖

（3）指定字段查询，在表结构变更时，能保证对应用程序无影响

2、insert必须指定字段，禁止使用insert into T values()

解读：指定字段插入，在表结构变更时，能保证对应用程序无影响

3、隐式类型转换会使索引失效，导致全表扫描

4、禁止在where条件列使用函数或者表达式

解读：导致不能命中索引，全表扫描

5、禁止负向查询以及%开头的模糊查询

解读：导致不能命中索引，全表扫描

6、禁止大表JOIN和子查询

7、同一个字段上的OR必须改写问IN，IN的值必须少于50个

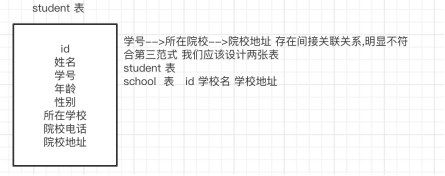
8、应用程序必须捕获SQL异常

解读：方便定位线上问题

说明：本规范适用于并发量大，数据量大的典型互联网业务，可直接参考。

### 14、有没有设计过数据表?你是如何设计的 难度系数：⭐

|  |  |
| --- | --- |
| 第一范式 | 每一列属性(字段)不可分割的,字段必须保证原子性 两列的属性值相近或者一样的,尽量合并到一列或者分表,确保数据不冗余 |
| 第二范式 | 每一行的数据只能与其中一行有关 即 主键  一行数据只能做一件事情或者表达一个意思, 只要数据出现重复,就要进行表的拆分 |
| 第三范式 | 数据不能存在传递关系,每个属性都跟主键有直接关联而不是间接关联 |



### 15、常见面试SQL 难度系数：⭐

**例1：**

用一条SQL语句查询出每门课都大于80分的学生姓名

name kecheng fenshu  
张三 语文 81  
张三 数学     75  
李四 语文     76  
李四 数学    90  
王五 语文     81  
王五 数学     100  
王五 英语     90  
**答1：**

**select** **distinct** **name** **from** **table** **where** **name** not in (**select** **distinct** **name** **from** **table** **where** fenshu<=80)    
**答2：**

**select** **name** **from** **table** **group** **by** **name** **having** **min**(fenshu)>80

**例2：**

学生表 如下:  
自动编号 学号 姓名 课程编号 课程名称 分数  
1      2005001  张三  0001    数学   69  
2      2005002  李四  0001    数学   89  
3      2005001  张三   0001    数学   69  
删除除了自动编号不同，其他都相同的学生冗余信息  
**答：**

**delete** tablename **where** 自动编号 not in(**select** **min**(自动编号) **from** tablename **group** **by**学号, 姓名, 课程编号, 课程名称, 分数)

**例3：**

一个叫team的表，里面只有一个字段name,一共有4条纪录，分别是a,b,c,d,对应四个球队，现在四个球队进行比赛，用一条sql语句显示所有可能的比赛组合.

**答：**

1. **select** a.**name**, b.**name**
2. **from** team a, team b
3. **where** a.**name** < b.**name**

**例4：**

怎么把这样一个表  
year month amount  
1991   1     1.1  
1991   2     1.2  
1991   3     1.3  
1991   4     1.4  
1992   1     2.1  
1992   2     2.2  
1992   3     2.3  
1992   4     2.4  
查成这样一个结果  
year m1 m2 m3 m4  
1991 1.1 1.2 1.3 1.4  
1992 2.1 2.2 2.3 2.4   
**答：**

1. **select** year,
2. (**select** amount **from** aaa m **where** month=1 and m.year=aaa.year) **as** m1,
3. (**select** amount **from** aaa m **where** month=2 and m.year=aaa.year) **as** m2,
4. (**select** amount **from** aaa m **where** month=3 and m.year=aaa.year) **as** m3,
5. (**select** amount **from**  aaa m **where** month=4 and m.year=aaa.year) **as** m4
6. **from** aaa **group** **by** year

**例5：**

说明：复制表(只复制结构,源表名：a新表名：b)

**答：**  
SQL:

**select** \* **into** b **from** a **where** 1<>1 (where1=1，拷贝表结构和数据内容)

ORACLE:

1. **create** **table** b
2. **As**
3. **Select** \* **from** a **where** 1=2

[<>（不等于）(SQL Server Compact)

比较两个表达式。 当使用此运算符比较非空表达式时，如果左操作数不等于右操作数，则结果为 TRUE。 否则，结果为 FALSE。]

**例6：**

原表：  
courseid coursename score  
1 java 70  
2 oracle 90  
3 xml 40  
4 jsp 30  
5 servlet 80

为了便于阅读,查询此表后的结果显式如下(及格分数为60):  
courseid coursename score mark  
1 java 70 pass  
2 oracle 90 pass  
3 xml 40 fail  
4 jsp 30 fail  
5 servlet 80 pass  
写出此查询语句

**答：**

**select** courseid, coursename ,score ,if(score>=60, "pass","fail")  **as** mark **from** course

**例7：**

表名：购物信息

购物人 商品名称 数量

A 甲 2

B 乙 4

C 丙 1

A 丁 2

B 丙 5

给出所有购入商品为两种或两种以上的购物人记录

**答：**

**select** \* **from** 购物信息 **where** 购物人 in (**select** 购物人 **from** 购物信息 **group** **by** 购物人 **having** count(\*) >= 2);

**例8：**

info 表

date result

2005-05-09 win

2005-05-09 lose

2005-05-09 lose

2005-05-09 lose

2005-05-10 win

2005-05-10 lose

2005-05-10 lose

**如果要生成下列结果, 该如何写sql语句?**

date win lose

2005-05-09 2 2

2005-05-10 1 2

**答1：**

**select** **date**, sum(case **when** result = "win" **then** 1 **else** 0 **end**) **as** "win", sum(case **when** result = "lose" **then** 1 **else** 0 **end**) **as** "lose" **from** info **group** **by** **date**;

**答2：**

1. **select** a.**date**, a.result **as** win, b.result **as** lose
2. **from**
3. (**select** **date**, count(result) **as** result **from** info **where** result = "win" **group** **by** **date**) **as** a
4. join
5. (**select** **date**, count(result) **as** result **from** info **where** result = "lose" **group** **by** **date**) **as** b
6. **on** a.**date** = b.**date**;

例9 mysql 创建了一个联合索引(a,b,c) 以下 索引生效 的是(1,2,4）

1、where a = 1 and b = 1 and c =1

2、where a = 1 and c = 1

3、where b = 1 and c = 1,

4、where b = 1 and a =1 and c = 1

第五章 Redis技术篇

### 1、介绍下Redis Redis有哪些数据类型 难度系数：⭐

**Redis全称（Remote Dictionary Server）**本质上是一个Key-Value类型的内存数据库，整个数据库统统加载在内存当中进行操作，定期通过异步操作把数据库数据flush到硬盘上进行保存。因为是纯内存操作，Redis的性能非常出色，每秒可以处理超过 10万次读写操作，是已知性能最快的Key-Value DB。 Redis的出色之处不仅仅是性能，Redis最大的魅力是支持保存多种数据结构，此外单个value的最大限制是1GB，不像 memcached只能保存1MB的数据，因此Redis可以用来实现很多有用的功能，比方说用他的List来做FIFO双向链表，实现一个轻量级的高性 能消息队列服务，用他的Set可以做高性能的tag系统等等。另外Redis也可以对存入的Key-Value设置expire时间，因此也可以被当作一 个功能加强版的memcached来用。 Redis的主要缺点是数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，因此Redis适合的场景主要局限在较小数据量的高性能操作和运算上。

**常用基本数据类型如下：**

|  |  |
| --- | --- |
| string | 字符串（一个字符串类型最大存储容量为512M） |
| list | 可以重复的集合 |
| set | 不可以重复的集合 |
| hash | 类似于Map<String,String> |
| zset(sorted set） | 带分数的set |

### 2、Redis提供了哪几种持久化方式 难度系数：⭐

RDB持久化方式能够在指定的时间间隔能对你的数据进行快照存储。

AOF持久化方式记录每次对服务器写的操作,当服务器重启的时候会重新执行这些命令来恢复原始的数据，AOF命令以redis协议追加保存每次写的操作到文件末尾.Redis还能对AOF文件进行后台重写,使得AOF文件的体积不至于过大。

如果你只希望你的数据在服务器运行的时候存在，你也可以不使用任何持久化方式。

你也可以同时开启两种持久化方式，在这种情况下，当redis重启的时候会优先载入AOF文件来恢复原始的数据,因为在通常情况下AOF文件保存的数据集要比RDB文件保存的数据集要完整。

**（1）RDB持久化：**

每隔一段时间，将内存中的数据集写到磁盘

Redis会单独创建（fork）一个子进程来进行持久化，会先将数据写入到个临时文件中，待持久化过程都结束了，再用这个临时文件替换上次持久化好的文件。整个过程中，主进程是不进行任何IO操作的，这就确保了极高的性能如果需要进行大规模数据的恢复，且对于数据恢复的完整性不是非常敏感，那RDB方式要比AOF方式更加的高效。

**保存策略：**

save 900 1 900 秒内如果至少有 1 个 key 的值变化，则保存

save 300 10 300 秒内如果至少有 10 个 key 的值变化，则保存

save 60 1 0000 60 秒内如果至10000 个 key 的值变化，则保存

**（2）AOF 持久化: 以日志形式记录每个更新（(总结、改）操作**

Redis重新启动时读取这个文件，重新执行新建、修改数据的命令恢复数据。

**保存策略：**

appendfsync always：每次产生一条新的修改数据的命令都执行保存操作；效率低，但是安全！

appendfsync everysec：每秒执行一次保存操作。如果在未保存当前秒内操作时发生了断电，仍然会导致一部分数据丢失（即1秒钟的数据）。

appendfsync no：从不保存，将数据交给操作系统来处理。更快，也更不安全的选择。

推荐（并且也是默认）的措施为每秒 fsync 一次， 这种 fsync 策略可以兼顾速度和安全性。

**缺点：**

1 比起RDB占用更多的磁盘空间

2 恢复备份速度要慢

3 每次读写都同步的话，有一定的性能压力

4 存在个别Bug，造成恢复不能

**（3）选择策略：**

可读的日志文本，通过操作AOF

官方推荐：

如果对数据不敏感，可以选单独用RDB；不建议单独用AOF，因为可能出现Bug;如果只是做纯内存缓存，可以都不用

### 3、Redis为什么是单线程的 难度系数：⭐

官方FAQ表示，因为Redis是基于内存的操作，CPU不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且CPU不会成为瓶颈，那就顺理成章地采用单线程的方案了Redis利用队列技术将并发访问变为串行访问

1）绝大部分请求是纯粹的内存操作

2）采用单线程,避免了不必要的上下文切换和竞争条件

### 4、Redis在项目中的应用 难度系数：⭐

**Redis一般来说在项目中有几方面的应用**

* 作为缓存，将热点数据进行缓存，减少和数据库的交互，提高系统的效率
* 作为分布式锁的解决方案，解决缓存击穿等问题
* 作为消息队列，使用Redis的发布订阅功能进行消息的发布和订阅

具体的使用场景要结合项目去说，比如说项目中有哪些场景用到Redis来作为缓存，以及分布式锁等等。

### 5、Redis服务器的的内存是多大 难度系数：⭐

配置文件中设置redis内存的参数：。

该参数如果不设置或者设置为0，则redis默认的内存大小为：

32位下默认是3G

64位下不受限制

一般推荐Redis设置内存为最大物理内存的四分之三，也就是0.75

命令行设置config set maxmemory <内存大小，单位字节>，服务器重启失效

config get maxmemory获取当前内存大小

永久则需要设置maxmemory参数，maxmemory是bytes字节类型，注意转换

### 6、为什么Redis的操作是原子性的，怎么保证原子性的 难度系数：⭐

对于Redis而言，命令的原子性指的是：一个操作的不可以再分，操作要么执行，要么不执行。

Redis的操作之所以是原子性的，是因为Redis是单线程的。

Redis本身提供的所有API都是原子操作，Redis中的事务其实是要保证批量操作的原子性。

多个命令在并发中也是原子性的吗？

不一定， 将get和set改成单命令操作，incr 。使用Redis的事务，或者使用Redis+Lua==的方式实现.

### 7、Redis有事务吗 难度系数：⭐

Redis是有事务的，redis中的事务是一组命令的集合，这组命令要么都执行，要不都不执行，

redis事务的实现，需要用到MULTI（事务的开始）和EXEC（事务的结束）命令 ;



当输入MULTI命令后，服务器返回OK表示事务开始成功，然后依次输入需要在本次事务中执行的所有命令，每次输入一个命令服务器并不会马上执行，而是返回”QUEUED”，这表示命令已经被服务器接受并且暂时保存起来，最后输入EXEC命令后，本次事务中的所有命令才会被依次执行，可以看到最后服务器一次性返回了两个OK，这里返回的结果与发送的命令是按顺序一一对应的，这说明这次事务中的命令全都执行成功了。

Redis的事务除了保证所有命令要不全部执行，要不全部不执行外，还能保证一个事务中的命令依次执行而不被其他命令插入。同时，redis的事务是不支持回滚操作的。

### 8、Redis数据和MySQL数据库的一致性如何实现 难度系数：⭐⭐

一、 延时双删策略

        在写库前后都进行redis.del(key)操作，并且设定合理的超时时间。具体步骤是：

        1）先删除缓存

        2）再写数据库

        3）休眠500毫秒（根据具体的业务时间来定）

        4）再次删除缓存。

        那么，这个500毫秒怎么确定的，具体该休眠多久呢？

        需要评估自己的项目的读数据业务逻辑的耗时。这么做的目的，就是确保读请求结束，写请求可以删除读请求造成的缓存脏数据。

        当然，这种策略还要考虑 redis 和数据库主从同步的耗时。最后的写数据的休眠时间：则在读数据业务逻辑的耗时的基础上，加上几百ms即可。比如：休眠1秒。

二、设置缓存的过期时间

        从理论上来说，给缓存设置过期时间，是保证最终一致性的解决方案。所有的写操作以数据库为准，只要到达缓存过期时间，则后面的读请求自然会从数据库中读取新值然后回填缓存

        结合双删策略+缓存超时设置，这样最差的情况就是在超时时间内数据存在不一致，而且又增加了写请求的耗时。

三、如何写完数据库后，再次删除缓存成功？

        上述的方案有一个缺点，那就是操作完数据库后，由于种种原因删除缓存失败，这时，可能就会出现数据不一致的情况。这里，我们需要提供一个保障重试的方案。

1、方案一具体流程

        （1）更新数据库数据；

        （2）缓存因为种种问题删除失败；

        （3）将需要删除的key发送至消息队列；

        （4）自己消费消息，获得需要删除的key；

        （5）继续重试删除操作，直到成功。

        然而，该方案有一个缺点，对业务线代码造成大量的侵入。于是有了方案二，在方案二中，启动一个订阅程序去订阅数据库的binlog，获得需要操作的数据。在应用程序中，另起一段程序，获得这个订阅程序传来的信息，进行删除缓存操作。

2、方案二具体流程

        （1）更新数据库数据；

        （2）数据库会将操作信息写入binlog日志当中；

        （3）订阅程序提取出所需要的数据以及key；

        （4）另起一段非业务代码，获得该信息；

        （5）尝试删除缓存操作，发现删除失败；

        （6）将这些信息发送至消息队列；

        （7）重新从消息队列中获得该数据，重试操作。

## 9、缓存击穿，缓存穿透，缓存雪崩的原因和解决方案(或者说使用缓存的过程中有没有遇到什么问题，怎么解决的） 难度系数：⭐

1. 缓存穿透：

是指查询一个不存在的数据，由于缓存无法命中，将去查询数据库，但是数据库也无此记录，并且出于容错考虑，我们没有将这次查询的null写入缓存，这将导致这个不存在的数据每次请求都要到存储层去查询，失去了缓存的意义。在流量大时，可能DB就挂掉了，要是有人利用不存在的key频繁攻击我们的应用，这就是漏洞。

解决方案：空结果也进行缓存，可以设置一个空对象，但它的过期时间会很短，最长不超过五分钟。 或者用布隆过滤器也可以解决，Redisson框架中有布隆过滤器。

（2）缓存雪崩：

是指在我们设置缓存时采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到DB，DB瞬时压力过重雪崩。

解决方案：原有的失效时间基础上增加一个随机值，比如1-5分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

（3） 缓存击穿

是指对于一些设置了过期时间的key，如果这些key可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：如果这个key在大量请求同时进来之前正好失效，那么所有对这个key的数据查询都落到DB，我们称为缓存击穿。

*解决方案：*在分布式的环境下，应使用分布式锁来解决，分布式锁的实现方案有多种，比如使用Redis的setnx、使用Zookeeper的临时顺序节点等来实现

### 10、哨兵模式是什么样的 难度系数：⭐⭐

如果Master异常，则会进行Master-Slave切换，将其中一Slae作为Master，将之前的Master作为Slave

**下线：**

①主观下线：Subjectively Down，简称 SDOWN，指的是当前 Sentinel 实例对某个redis服务器做出的下线判断。

②客观下线：Objectively Down， 简称 ODOWN，指的是多个 Sentinel 实例在对Master Server做出 SDOWN 判断，并且通过 SENTINEL is-master-down-by-addr 命令互相交流之后，得出的Master Server下线判断，然后开启failover.

**工作原理：**

（1）每个Sentinel以每秒钟一次的频率向它所知的Master，Slave以及其他 Sentinel 实例发送一个 PING 命令 ；

（2）如果一个实例（instance）距离最后一次有效回复 PING 命令的时间超过 down-after-milliseconds 选项所指定的值， 则这个实例会被 Sentinel 标记为主观下线；

（3）如果一个Master被标记为主观下线，则正在监视这个Master的所有 Sentinel 要以每秒一次的频率确认Master的确进入了主观下线状态；

（4）当有足够数量的 Sentinel（大于等于配置文件指定的值）在指定的时间范围内确认Master的确进入了主观下线状态， 则Master会被标记为客观下线 ；

（5）在一般情况下， 每个 Sentinel 会以每 10 秒一次的频率向它已知的所有Master，Slave发送 INFO 命令

（6）当Master被 Sentinel 标记为客观下线时，Sentinel 向下线的 Master 的所有 Slave 发送 INFO 命令的频率会从 10 秒一次改为每秒一次 ；

（7）若没有足够数量的 Sentinel 同意 Master 已经下线， Master 的客观下线状态就会被移除；

若 Master 重新向 Sentinel 的 PING 命令返回有效回复， Master 的主观下线状态就会被移除；

### 11、Redis为什么快 难度系数：⭐

1)完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)

2)数据结构简单，对数据操作也简单，Redis中的数据结构是专门进行设计的

3)采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗

4)使用多路I/O复用模型，非阻塞IO

5)使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求

### 12、Redis常见性能问题和解决方案 难度系数：⭐

(1) Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件

(2) 如果数据比较重要，某个Slave开启AOF备份数据，策略设置为每秒同步一次

(3) 为了主从复制的速度和连接的稳定性，Master和Slave最好在同一个局域网内

(4) 尽量避免在压力很大的主库上增加从库

(5) 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1 <- Slave2 <- Slave3...

这样的结构方便解决单点故障问题，实现Slave对Master的替换。如果Master挂了，可以立刻启用Slave1做Master，其他不变。

## 13、MySQL里有大量数据，如何保证Redis中的数据都是热点数据 难度系数：⭐⭐

Redis内存淘汰策略

redis内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。

**数据淘汰策略：**

noeviction:返回错误当内存限制达到并且客户端尝试执行会让更多内存被使用的命令（大部分的写入指令，但DEL和几个例外）

allkeys-lru: 尝试回收最少使用的键（LRU），使得新添加的数据有空间存放。

volatile-lru: 尝试回收最少使用的键（LRU），但仅限于在过期集合的键,使得新添加的数据有空间存放。

allkeys-random: 回收随机的键使得新添加的数据有空间存放。

volatile-random: 回收随机的键使得新添加的数据有空间存放，但仅限于在过期集合的键。

volatile-ttl: 回收在过期集合的键，并且优先回收存活时间（TTL）较短的键,使得新添加的数据有空间存放。

### 14、Redis集群方案应该怎么做 都有哪些方案 难度系数：⭐⭐

（1）twemproxy，大概概念是，它类似于一个代理方式，使用方法和普通redis无任何区别，设置好它下属的多个redis实例后，使用时在本需要连接redis的地方改为连接twemproxy，它会以一个代理的身份接收请求并使用一致性hash算法，将请求转接到具体redis，将结果再返回twemproxy。使用方式简便(相对redis只需修改连接端口)，对旧项目扩展的首选。 问题：twemproxy自身单端口实例的压力，使用一致性hash后，对redis节点数量改变时候的计算值的改变，数据无法自动移动到新的节点。

（2）codis，目前用的最多的集群方案，基本和twemproxy一致的效果，但它支持在 节点数量改变情况下，旧节点数据可恢复到新hash节点。

（3）redis cluster3.0自带的集群，特点在于他的分布式算法不是一致性hash，而是hash槽的概念，以及自身支持节点设置从节点。具体看官方文档介绍。

（4）在业务代码层实现，起几个毫无关联的redis实例，在代码层，对key 进行hash计算，然后去对应的redis实例操作数据。 这种方式对hash层代码要求比较高，考虑部分包括，节点失效后的替代算法方案，数据震荡后的自动脚本恢复，实例的监控，等等。

### 15、说说Redis哈希槽的概念 难度系数：⭐⭐

Redis集群没有使用一致性hash,而是引入了哈希槽的概念，Redis集群有16384个哈希槽，每个key通过CRC16校验后对16384取模来决定放置哪个槽，集群的每个节点负责一部分hash槽。

### 16、Redis有哪些适合的场景 难度系数：⭐

（1）会话缓存（Session Cache）

最常用的一种使用Redis的情景是会话缓存（session cache）。用Redis缓存会话比其他存储（如Memcached）的优势在于：Redis提供持久化。当维护一个不是严格要求一致性的缓存时，如果用户的购物车信息全部丢失，大部分人都会不高兴的，现在，他们还会这样吗？

幸运的是，随着 Redis 这些年的改进，很容易找到怎么恰当的使用Redis来缓存会话的文档。甚至广为人知的商业平台Magento也提供Redis的插件。

（2）全页缓存（FPC）

除基本的会话token之外，Redis还提供很简便的FPC平台。回到一致性问题，即使重启了Redis实例，因为有磁盘的持久化，用户也不会看到页面加载速度的下降，这是一个极大改进，类似PHP本地FPC。

再次以Magento为例，Magento提供一个插件来使用Redis作为全页缓存后端。

此外，对WordPress的用户来说，Pantheon有一个非常好的插件 wp-redis，这个插件能帮助你以最快速度加载你曾浏览过的页面。

（3）队列

Reids在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得Redis能作为一个很好的消息队列平台来使用。Redis作为队列使用的操作，就类似于本地程序语言（如Python）对 list 的 push/pop 操作。

如果你快速的在Google中搜索“Redis queues”，你马上就能找到大量的开源项目，这些项目的目的就是利用Redis创建非常好的后端工具，以满足各种队列需求。例如，Celery有一个后台就是使用Redis作为broker，你可以从这里去查看。

（4）排行榜/计数器

Redis在内存中对数字进行递增或递减的操作实现的非常好。集合（Set）和有序集合（Sorted Set）也使得我们在执行这些操作的时候变的非常简单，Redis只是正好提供了这两种数据结构。所以，我们要从排序集合中获取到排名最靠前的10个用户–我们称之为“user\_scores”，我们只需要像下面一样执行即可：

当然，这是假定你是根据你用户的分数做递增的排序。如果你想返回用户及用户的分数，你需要这样执行：

ZRANGE user\_scores 0 10 WITHSCORES

Agora Games就是一个很好的例子，用Ruby实现的，它的排行榜就是使用Redis来存储数据的，你可以在这里看到。

（5）发布/订阅

最后（但肯定不是最不重要的）是Redis的发布/订阅功能。发布/订阅的使用场景确实非常多。我已看见人们在社交网络连接中使用，还可作为基于发布/订阅的脚本触发器，甚至用Redis的发布/订阅功能来建立聊天系统！（不，这是真的，你可以去核实）。

第六章 分布式技术篇

### 1、消息队列的作用与使用场景 难度系数：⭐

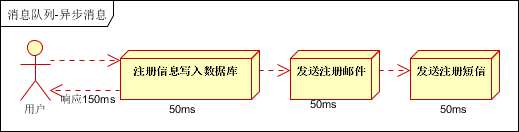
消息队列在实际应用中常用的使用场景。异步处理，应用解耦，流量削锋和消息通讯四个场景

**2.1异步处理**

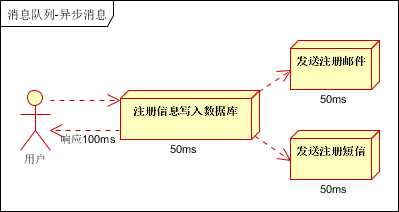
场景说明：用户注册后，需要发注册邮件和注册短信。

传统的做法有两种 1.串行的方式；2.并行方式

（1）串行方式：将注册信息写入[数据库](http://lib.csdn.net/base/mysql)成功后，发送注册邮件，再发送注册短信。以上三个任务全部完成后，返回给客户端



（2）并行方式：将注册信息写入数据库成功后，发送注册邮件的同时，发送注册短信。以上三个任务完成后，返回给客户端。与串行的差别是，并行的方式可以提高处理的时间

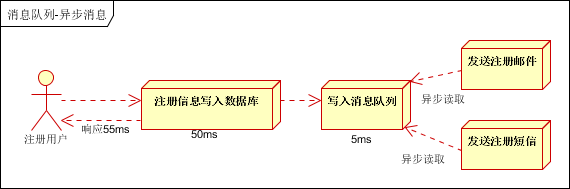


假设三个业务节点每个使用50毫秒钟，不考虑网络等其他开销，则串行方式的时间是150毫秒，并行的时间可能是100毫秒。

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

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

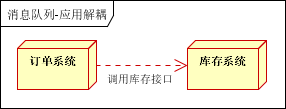
引入消息队列，将不是必须的业务逻辑，异步处理。改造后的架构如下：



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

**2.2应用解耦**

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

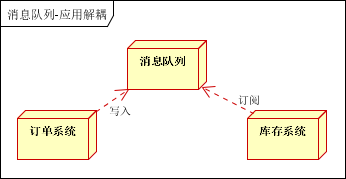


传统模式的缺点：

假如库存系统无法访问，则订单减库存将失败，从而导致订单失败

订单系统与库存系统耦合

如何解决以上问题呢？引入应用消息队列后的方案，如下图：



订单系统：用户下单后，订单系统完成持久化处理，将消息写入消息队列，返回用户订单下单成功

库存系统：订阅下单的消息，采用拉/推的方式，获取下单信息，库存系统根据下单信息，进行库存操作

假如：在下单时库存系统不能正常使用。也不影响正常下单，因为下单后，订单系统写入消息队列就不再关心其他的后续操作了。实现订单系统与库存系统的应用解耦

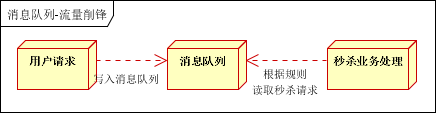
**2.3流量削锋**

流量削锋也是消息队列中的常用场景，一般在秒杀或团抢活动中使用广泛

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

可以控制活动的人数

可以缓解短时间内高流量压垮应用



用户的请求，服务器接收后，首先写入消息队列。假如消息队列长度超过最大数量，则直接抛弃用户请求或跳转到错误页面

秒杀业务根据消息队列中的请求信息，再做后续处理

**2.4日志处理**

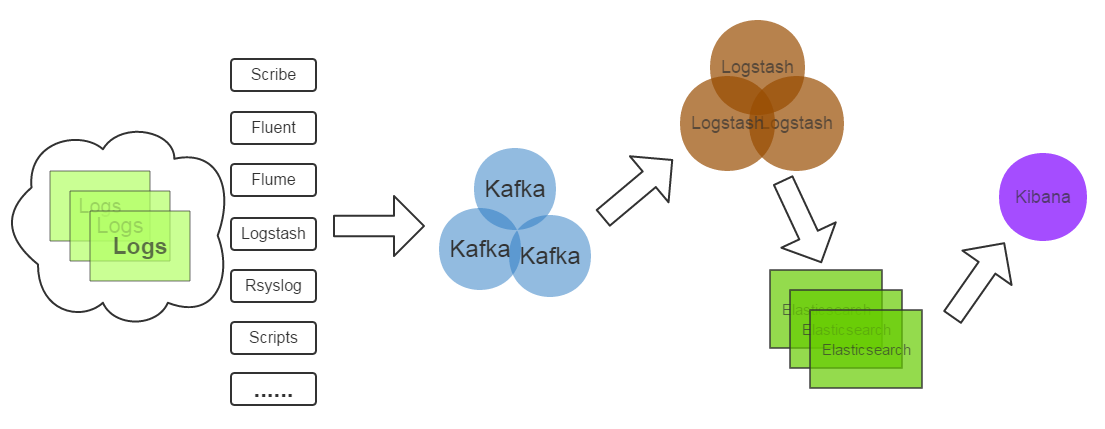
日志处理是指将消息队列用在日志处理中，比如Kafka的应用，解决大量日志传输的问题。

日志采集客户端，负责日志数据采集，定时写受写入Kafka队列

Kafka消息队列，负责日志数据的接收，存储和转发

日志处理应用：订阅并消费kafka队列中的日志数据

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



(1)Kafka：接收用户日志的消息队列

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

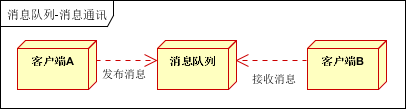
(3)Elasticsearch：实时日志分析服务的核心技术，一个schemaless，实时的数据存储服务，通过index组织数据，兼具强大的搜索和统计功能

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

2.5消息通讯

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

点对点通讯：



客户端A和客户端B使用同一队列，进行消息通讯。

聊天室通讯：

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

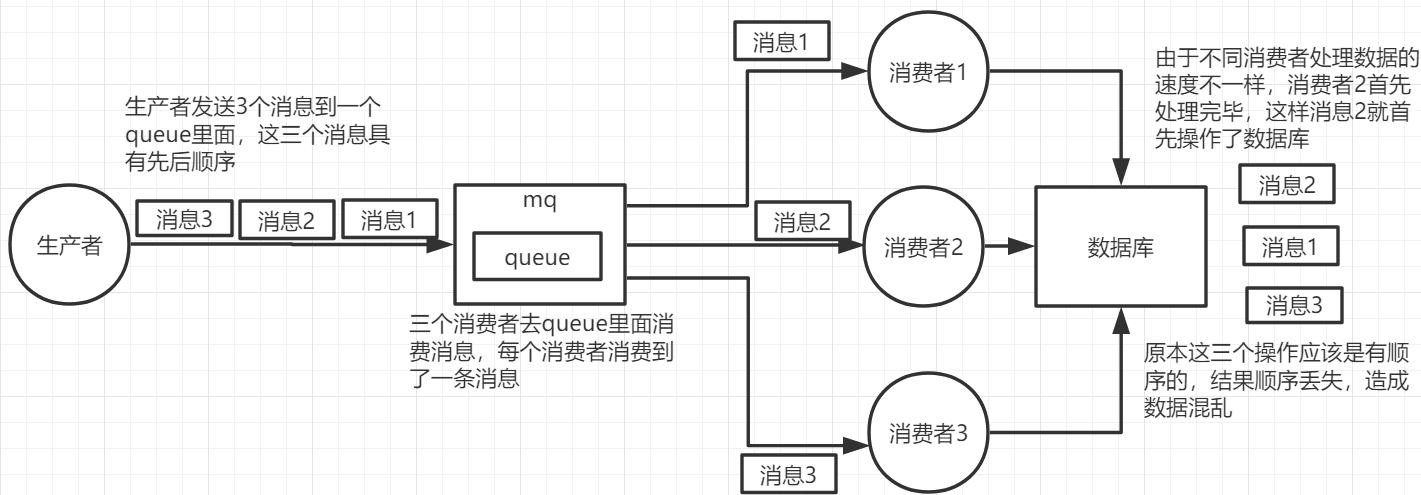
以上实际是消息队列的两种消息模式，点对点或发布订阅模式。模型为示意图，供参考。

### 2、RabbitMQ如何保证消息的顺序性 难度系数：⭐⭐

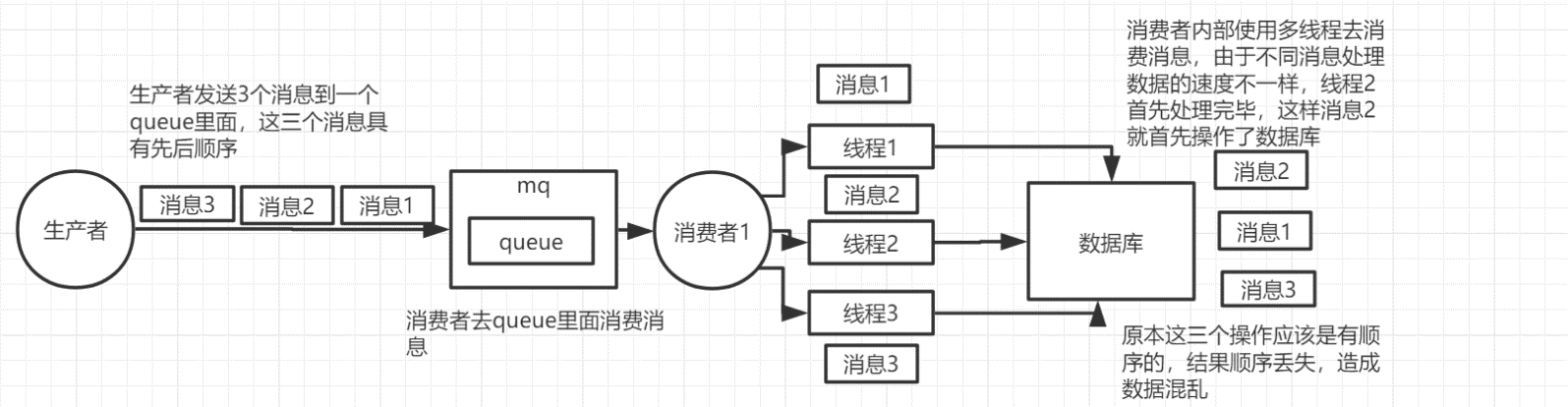
消息队列中的若干消息如果是对同一个数据进行操作，这些操作具有前后的关系，必须要按前后的顺序执行，否则就会造成数据异常。举例：  
比如通过mysql binlog进行两个数据库的数据同步，由于对数据库的数据操作是具有顺序性的，如果操作顺序搞反，就会造成不可估量的错误。比如数据库对一条数据依次进行了 插入->更新->删除操作，这个顺序必须是这样，如果在同步过程中，消息的顺序变成了 删除->插入->更新，那么原本应该被删除的数据，就没有被删除，造成数据的不一致问题。

举例场景：

RabbitMQ：①一个queue，有多个consumer去消费，这样就会造成顺序的错误，consumer从MQ里面读取数据是有序的，但是每个consumer的执行时间是不固定的，无法保证先读到消息的consumer一定先完成操作，这样就会出现消息并没有按照顺序执行，造成数据顺序错误。

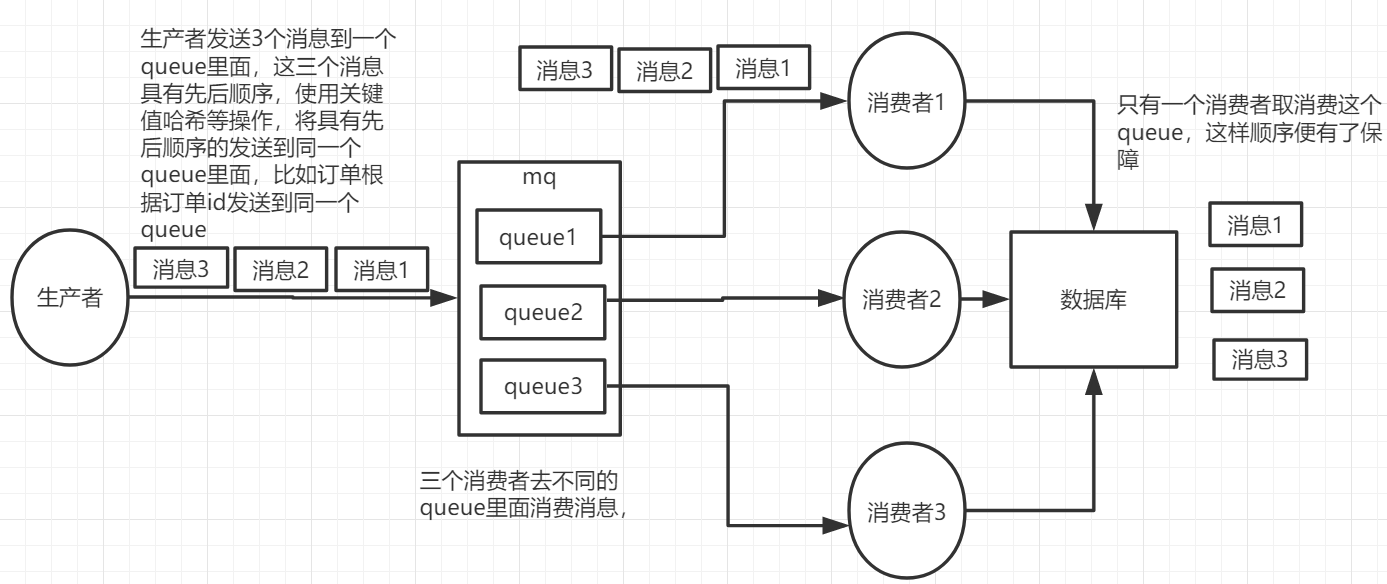


②一个queue对应一个consumer，但是consumer里面进行了多线程消费，这样也会造成消息消费顺序错误。



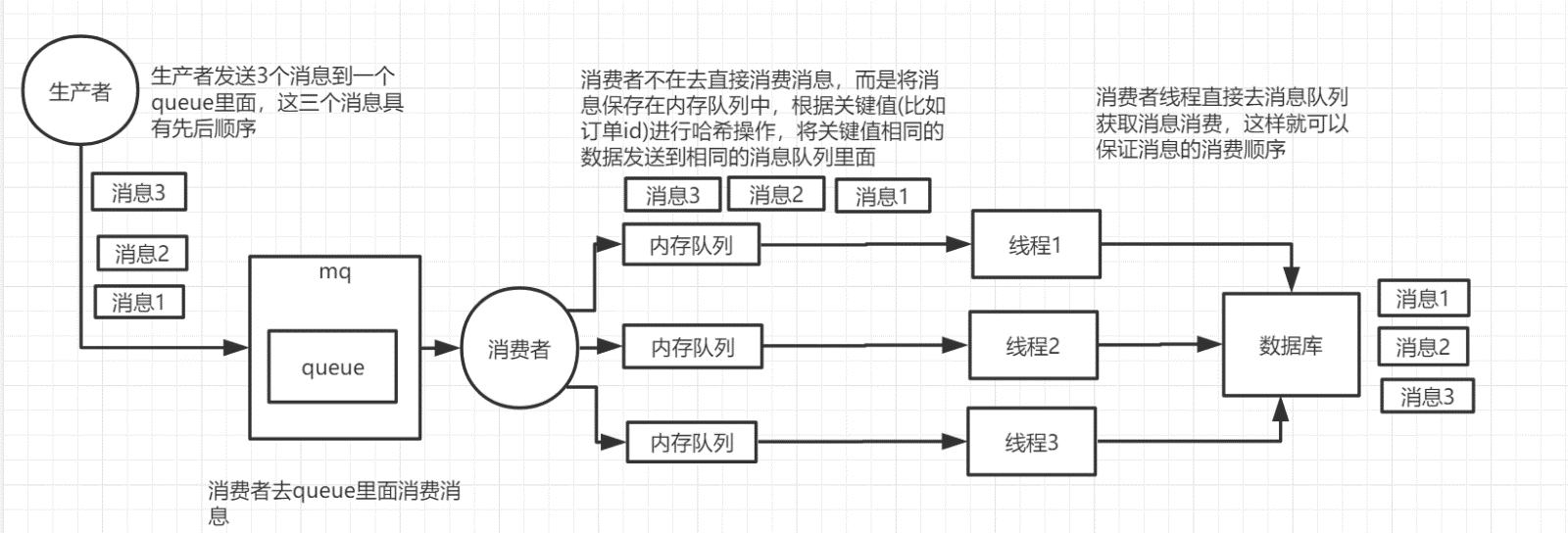
解决方案：

①拆分多个queue，每个queue一个consumer，就是多一些queue而已，确实是麻烦点；这样也会造成吞吐量下降，可以在消费者内部采用多线程的方式取消费。



一个queue对应一个consumer

②或者就一个queue但是对应一个consumer，然后这个consumer内部用内存队列做排队，然后分发给底层不同的worker来处理



一个queue对应一个consumer，采用多线程

### 3、如何使用RabbitMQ解决分布式事务 难度系数：⭐⭐

**分布式事务：**不同的服务操作不同的数据源（库或表），保证数据一致性的问题。

**解决：**采用RabbitMQ消息最终一致性的解决方案，解决分布式事务问题。

分布式事务场景：

1、电商项目中的商品库和ES库数据同步问题。

2、电商项目中：支付----🡪订单---🡪库存，一系列操作，进行状态更改等。

在互联网应用中，基本都会有用户注册的功能。在注册的同时，我们会做出如下操作：

收集用户录入信息，保存到数据库向用户的手机或邮箱发送验证码等等…

如果是传统的集中式架构，实现这个功能非常简单：开启一个本地事务，往本地数据库中插入一条用户数据，发送验证码，提交事物。

但是在分布式架构中，用户和发送验证码是两个独立的服务，它们都有各自的数据库，那么就不能通过本地事物保证操作的原子性。这时我们就需要用到 RabbitMQ（消息队列）来为我们实现这个需求。

在用户进行注册操作的时候，我们为该操作创建一条消息，当用户信息保存成功时，把这条消息发送到消息队列。验证码系统会监听消息，一旦接受到消息，就会给该用户发送验证码。

### 4、RabbitMQ如何确保消息可靠性 难度系数：⭐⭐

消息可靠性一般来说由3方面来保证：

1) 生产者

RabbitMQ提供transaction事务和confirm模式来确保生产者不丢消息；

Transaction事务机制就是说：发送消息前，开启事务（channel.txSelect()）,然后发送消息，如果发送过程中出现什么异常，事务就会回滚 （channel.txRollback()）,如果发送成功则提交事务（channel.txCommit()），然而，这种方式有个缺点：吞吐量下降。

confirm模式用的居多：一旦channel进入confirm模式，所有在该信道上发布的消息都将会被指派一个唯一的ID（从1开始），一旦消息被投递到所有匹配的队列之后；

rabbitMQ就会发送一个ACK给生产者（包含消息的唯一ID），这就使得生产者知道消息已经正确到达目的队列了；

如果rabbitMQ没能处理该消息，则会发送一个Nack消息给你，可以进行重试操作。

2) 消息队列本身

可以进行消息持久化, 即使rabbitMQ挂了，重启后也能恢复数据

如果要进行消息持久化，那么需要对以下3种实体均配置持久化

a) Exchange

声明exchange时设置持久化（durable = true）并且不自动删除(autoDelete = false)

b) Queue

声明queue时设置持久化（durable = true）并且不自动删除(autoDelete = false)

c) message

发送消息时通过设置deliveryMode=2持久化消息

3) 消费者

消费者丢数据一般是因为采用了自动确认消息模式，消费者在收到消息之后，处理消息之前，会自动回复RabbitMQ已收到消息；如果这时处理消息失败，就会丢失该消息；改为手动确认消息即可！手动确认模式下消费失败时，不将其重新放入队列（确认重试也不会成功的情形），打印错误信息后，通知相关人员，人工介入处理。

### 5、如何防止RabbitMQ消息重复消费 难度系数：⭐

保证消息幂等性。幂等性概念：一个请求，不管重复来多少次，结果是不会改变的。

RabbitMQ、RocketMQ、Kafka等任何队列不保证消息不重复，如果业务需要消息不重复消费，则需要消费端处理业务消息要保持幂等性

方式一：Redis的setNX() , 做消息id去重 java版本目前不支持设置过期时间

//Redis中操作，判断是否已经操作过 TODO

boolean flag = jedis.setNX(key);

if(flag){

//消费

}else{

//忽略，重复消费

}

方式二：redis的 Incr 原子操作：key自增，大于0 返回值大于0则说明消费过，(key可以是消息的md5取值, 或者如果消息id设计合理直接用id做key)

int num = jedis.incr(key);

if(num == 1){

//消费

}else{

//忽略，重复消费

}

方式三：数据库去重表

设计一个去重表，某个字段使用Message的key做唯一索引，因为存在唯一索引，所以重复消费会失败

CREATE TABLE `message\_record` ( `id` int(11) unsigned NOT NULL AUTO\_INCREMENT, `key` varchar(128) DEFAULT NULL, `create\_time` datetime DEFAULT NULL, PRIMARY KEY (`id`), UNIQUE KEY `key` (`key`) ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

## 6、如何解决消息队列的延时以及过期失效问题 消息队列满了之后该如何处理 有几百万的消息持续积压几小时,说说如何解决 难度系数：⭐⭐

方案分析

该问题,其本质针对的场景，都是说，可能你的消费端出了问题，不消费了，或者消费的极其极其慢。另外还有可能你的消息队列集群的磁盘都快写满了，都没人消费，这个时候怎么办？或者是整个这就积压了几个小时，你这个时候怎么办？或者是你积压的时间太长了，导致比如rabbitmq设置了消息过期时间后就没了怎么办？

所以这种问题线上常见的，一般不出，一出就是大问题，一般常见于，举个例子，消费端每次消费之后要写mysql，结果mysql挂了，消费端挂掉了。导致消费速度极其慢。

分析1+话术

这个是我们真实遇到过的一个场景，确实是线上故障了，这个时候要不然就是修复consumer的问题，让他恢复消费速度，然后傻傻的等待几个小时消费完毕。(可行,但是不建议 在面试的时候说)

一个消费者一秒是1000条，一秒3个消费者是3000条，一分钟是18万条，1000多万条

所以如果你积压了几百万到上千万的数据，即使消费者恢复了，也需要大概1小时的时间才能恢复过来

一般这个时候，只能操作临时紧急扩容了，具体操作步骤和思路如下：

1）先修复consumer的问题，确保其恢复消费速度，然后将现有cnosumer都停掉

2）新建一个topic，partition是原来的10倍，临时建立好原先10倍或者20倍的queue数量

3）然后写一个临时的分发数据的consumer程序，这个程序部署上去消费积压的数据，消费之后不做耗时的处理，直接均匀轮询写入临时建立好的10倍数量的queue

4）接着临时征用10倍的机器来部署consumer，每一批consumer消费一个临时queue的数据

5）这种做法相当于是临时将queue资源和consumer资源扩大10倍，以正常的10倍速度来消费数据

6）等快速消费完积压数据之后，得恢复原先部署架构，重新用原先的consumer机器来消费消息

分析2+话术

rabbitmq是可以设置过期时间的，就是TTL，如果消息在queue中积压超过一定的时间就会被rabbitmq给清理掉，这个数据就没了。那这就是第二个坑了。这就不是说数据会大量积压在mq里，而是大量的数据会直接搞丢。

这个情况下，就不是说要增加consumer消费积压的消息，因为实际上没啥积压，而是丢了大量的消息。我们可以采取一个方案，就是批量重导，这个我们之前线上也有类似的场景干过。就是大量积压的时候，我们当时就直接丢弃数据了，然后等过了高峰期以后，比如大家一起喝咖啡熬夜到晚上12点以后，用户都睡觉了。

这个时候我们就开始写程序，将丢失的那批数据，写个临时程序，一点一点的查出来，然后重新灌入mq里面去，把白天丢的数据给他补回来。也只能是这样了。

假设1万个订单积压在mq里面，没有处理，其中1000个订单都丢了，你只能手动写程序把那1000个订单给查出来，手动发到mq里去再补一次

分析3+话术

如果走的方式是消息积压在mq里，那么如果你很长时间都没处理掉，此时导致mq都快写满了，咋办？这个还有别的办法吗？没有，谁让你第一个方案执行的太慢了，你临时写程序，接入数据来消费，消费一个丢弃一个，都不要了，快速消费掉所有的消息。然后走第二个方案，到了晚上再补数据吧。

### 7、如果让你写一个消息队列,该如何进行架构设计 难度系数：⭐⭐

面试官心理分析

（1）你有没有对某一个消息队列做过较为深入的原理的了解，或者从整体了解把握住一个mq的架构原理

（2）看看你的设计能力，给你一个常见的系统，就是消息队列系统，看看你能不能从全局把握一下整体架构设计，给出一些关键点出来

类似问题

如果让你来设计一个spring框架你会怎么做？如果让你来设计一个dubbo框架你会怎么做？如果让你来设计一个mybatis框架你会怎么做？

回答思路

（1）首先这个mq得支持可伸缩性吧，就是需要的时候快速扩容，就可以增加吞吐量和容量，那怎么搞？设计个分布式的系统

（2）其次你得考虑一下这个mq的数据要不要落地磁盘吧？那肯定要了，落磁盘，才能保证别进程挂了数据就丢了。那落磁盘的时候怎么落啊？顺序写，这样就没有磁盘随机读写的寻址开销，磁盘顺序读写的性能是很高的。

（3）其次你考虑一下你的mq的可用性啊？

（4）能不能支持数据0丢失啊？

面试官问你这个问题，其实是个开放题，他就是看看你有没有从架构角度整体构思和设计的思维以及能力。确实这个问题可以刷掉一大批人，因为大部分人平时不思考这些东西。

### 8、什么是ElasticSearch 怎么解决中文分词 难度系数：⭐

Elasticsearch是一个基于Lucene的搜索引擎。它提供了具有HTTP Web界面和无架构JSON文档的分布式，多租户能力的全文搜索引擎。Elasticsearch是用Java开发的，根据Apache许可条款作为开源发布。

采用IK中文分词器插件进行中文分词处理。

### 9、Elasticsearch中的倒排索引是什么 难度系数：⭐

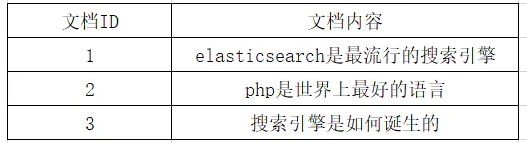
**正排索引：**

在说倒排索引之前我们先说说什么是正排索引。正排索引也称为"前向索引"，它是创建倒排索引的基础。

这种组织方法在建立索引的时候结构比较简单，建立比较方便且易于维护;因为索引是基于文档建立的，若是有新的文档加入，直接为该文档建立一个新的索引块，挂接在原来索引文件的后面。若是有文档删除，则直接找到该文档号文档对应的索引信息，将其直接删除。

他适合根据文档ID来查询对应的内容。但是在查询一个keyword在哪些文档里包含的时候需对所有的文档进行扫描以确保没有遗漏，这样就使得检索时间大大延长，检索效率低下。

比如有几个文档及里面的内容，他正排索引构建的结果如下图：



**优点：**工作原理非常的简单。  
**缺点：**检索效率太低，只能在一起简单的场景下使用。

**倒排索引：**

根据字面意思可以知道他和正序索引是反的。在搜索引擎中每个文件都对应一个文件ID，文件内容被表示为一系列关键词的集合（文档要除去一些无用的词，比如’的’这些，剩下的词就是关键词，每个关键词都有自己的ID）。例如“文档1”经过分词，提取了3个关键词，每个关键词都会记录它所在在文档中的出现频率及出现位置。

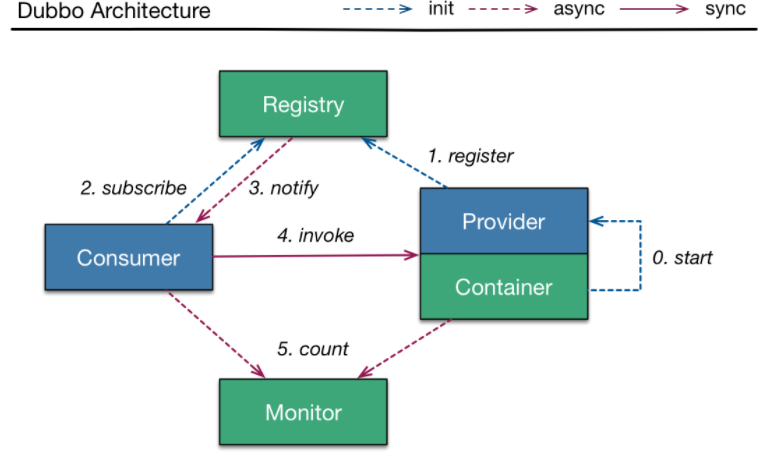
那么上面的文档及内容构建的倒排索引结果会如下图（注：这个图里没有记录展示该词在出现在哪个文档的具体位置）：



**如何来查询呢？**

比如我们要查询‘搜索引擎’这个关键词在哪些文档中出现过。首先我们通过倒排索引可以查询到该关键词出现的文档位置是在1和3中;然后再通过正排索引查询到文档1和3的内容并返回结果。

### 10、Dubbo执行流程是什么样的？各个组件作用 难度系数：⭐



它的架构主要有五个角色/核心组件，分为是Container（容器）、Provider（服务的提供方）、Registry（注册中心）、Consumer（服务的消费方）、Monitor（监控中心）。

容器主要负责启动、加载、运行服务提供者；

同时服务提供者在启动时，向注册中心注册自己提供的服务；

消费者向注册中心订阅自己的服务；

注册中心返回服务提供者列表给消费者，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；

对于服务消费者，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，再选另外一台调用；

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；

**注册中心：**

1. 注册中心只负责地址的注册和查找，相当于我们的目录服务，只有在容器启动时，服务提供者和调用者与注册中心交互，整个过程，注册中心不参与数据传输，不转发请求，压力较小 （“两不一小”）；
2. 注册中心宕机问题：（Dubbo挂了怎么办？）

注册中心会部署集群，

如果集群中的任意一台宕机之后，将自动切换到另外一台，不会影响已运行的提供者和消费者；

如果集群中所有注册中心全部宕机之后，服务提供者和服务消费者仍能通过本地缓存通讯；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；

**监控中心：**

1. 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
2. 监控中心宕机不影响使用，只是丢失部分的采样数据；
3. dubbo-admin可以通过监控中心的可视化界面，进行禁止服务和截止消费者（大量恶意访问的ip）；

### 11、ZooKeeper是什么 难度系数：⭐

ZooKeeper是一个开放源码的分布式协调服务，它是集群的管理者，监视着集群中各个节点的状态根据节点提交的反馈进行下一步合理操作。最终，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

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保证了如下分布式一致性特性：

* 顺序一致性
* 原子性
* 单一视图
* 可靠性
* 实时性（最终一致性）

客户端的读请求可以被集群中的任意一台机器处理，如果读请求在节点上注册了监听器，这个监听器也是由所连接的zookeeper机器来处理。对于写请求，这些请求会同时发给其他zookeeper机器并且达成一致后，请求才会返回成功。因此，随着zookeeper的集群机器增多，读请求的吞吐会提高但是写请求的吞吐会下降。

有序性是zookeeper中非常重要的一个特性，所有的更新都是全局有序的，每个更新都有一个唯一的时间戳，这个时间戳称为zxid（Zookeeper Transaction Id）。而读请求只会相对于更新有序，也就是读请求的返回结果中会带有这个zookeeper最新的zxid。

## 12、Zookeeper Watcher 机制 -- 数据变更通知 难度系数：⭐

Zookeeper允许客户端向服务端的某个Znode注册一个Watcher监听，当服务端的一些指定事件触发了这个Watcher，服务端会向指定客户端发送一个事件通知来实现分布式的通知功能，然后客户端根据Watcher通知状态和事件类型做出业务上的改变。

**工作机制：**

* 客户端注册watcher
* 服务端处理watcher
* 客户端回调watcher

**Watcher特性总结：**

（1）一次性  
无论是服务端还是客户端，一旦一个Watcher被触发，Zookeeper都会将其从相应的存储中移除。这样的设计有效的减轻了服务端的压力，不然对于更新非常频繁的节点，服务端会不断的向客户端发送事件通知，无论对于网络还是服务端的压力都非常大。

（2）客户端串行执行  
客户端Watcher回调的过程是一个串行同步的过程。

（3）轻量

Watcher通知非常简单，只会告诉客户端发生了事件，而不会说明事件的具体内容。

客户端向服务端注册Watcher的时候，并不会把客户端真实的Watcher对象实体传递到服务端，仅仅是在客户端请求中使用boolean类型属性进行了标记。

watcher event异步发送watcher的通知事件从server发送到client是异步的，这就存在一个问题，不同的客户端和服务器之间通过socket进行通信，由于网络延迟或其他因素导致客户端在不通的时刻监听到事件，由于Zookeeper本身提供了ordering guarantee，即客户端监听事件后，才会感知它所监视znode发生了变化。所以我们使用Zookeeper不能期望能够监控到节点每次的变化。Zookeeper只能保证最终的一致性，而无法保证强一致性。

注册watcher getData、exists、getChildren

触发watcher create、delete、setData

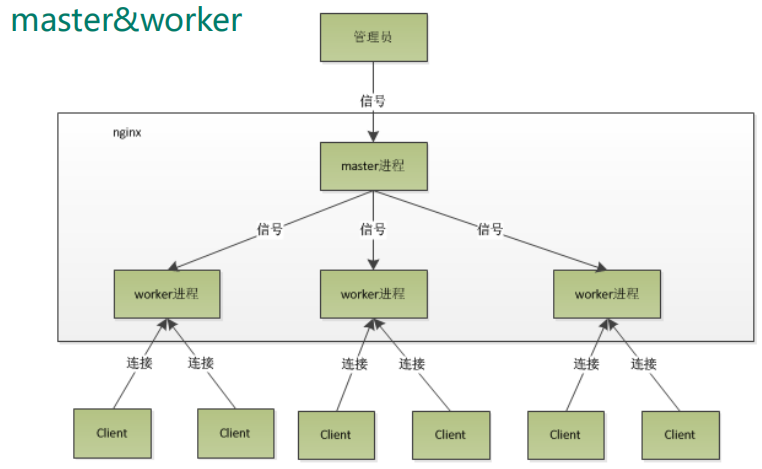
当一个客户端连接到一个新的服务器上时，watch将会被以任意会话事件触发。当与一个服务器失去连接的时候，是无法接收到watch的。而当client重新连接时，如果需要的话，所有先前注册过的watch，都会被重新注册。通常这是完全透明的。只有在一个特殊情况下，watch可能会丢失：对于一个未创建的znode的exist watch，如果在客户端断开连接期间被创建了，并且随后在客户端连接上之前又删除了，这种情况下，这个watch事件可能会被丢失。

### 13、ZooKeeper分布式锁的实现原理 难度系数：⭐

使用zookeeper创建临时序列节点来实现分布式锁，适用于顺序执行的程序，大体思路就是创建临时序列节点，找出最小的序列节点，获取分布式锁，程序执行完成之后此序列节点消失，通过watch来监控节点的变化，从剩下的节点的找到最小的序列节点，获取分布式锁，执行相应处理。

### 14、Nginx原理 难度系数：⭐

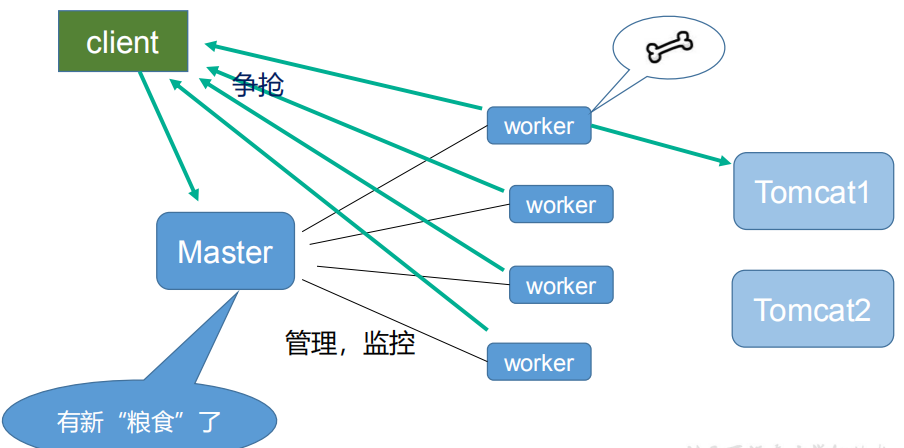
Nginx内部进程模型：



1.在nginx启动后，会有一个master进程和多个worker进程，master进程主要用来管理worker进程，包括：接受信号，将信号分发给worker进程，监听worker进程工作状态，当worker进程退出时(非正常)，启动新的worker进程。基本的网络事件会交给worker进程处理。多个worker进程之间是对等的，他们同等竞争来自客户端的请求，各进程互相之间是独立的 。一个请求，只可能在一个worker进程中处理，一个worker进程，不可能处理其它进程的请求。 worker进程的个数是可以设置的，一般我们会设置与机器cpu核数一致，这里面的原因与nginx的进程模型以及事件处理模型是分不开的 。

2.当master接收到重新加载的信号会怎么处理(./nginx -s reload)?，master会重新加载配置文件，然后启动新的进程，使用的新的worker进程来接受请求，并告诉老的worker进程他们可以退休了，老的worker进程将不会接受新的，老的worker进程处理完手中正在处理的请求就会退出。

3.worker进程是如何处理用户的请求呢？首先master会根据配置文件生成一个监听相应端口的socket，然后再faster出多个worker进程，这样每个worker就可以接受从socket过来的消息（其实这个时候应该是每一个worker都有一个socket，只是这些socket监听的地址是一样的）。当一个连接过来的时候，每一个worker都能接收到通知，但是只有一个worker能和这个连接建立关系，其他的worker都会连接失败，这就是所谓的惊群现在，为了解决这个问题，nginx提供一个共享锁accept\_mutex，有了这个共享锁后，就会只有一个worker去接收这个连接。当一个worker进程在accept这个连接之后，就开始读取请求，解析请求，处理请求，产生数据后，再返回给客户端，最后才断开连接，这样一个完整的请求就是这样的了。



**master-workers的机制的好处**

首先，对于每个worker进程来说，独立的进程，不需要加锁，所以省掉了锁带来的开销，同时在编程以及问题查找时，也会方便很多。

其次，采用独立的进程，可以让互相之间不会影响，一个进程退出后，其它进程还在工作，服务不会中断，master进程则很快启动新的worker进程。

当然，worker进程的异常退出，肯定是程序有bug了，异常退出，会导致当前worker上的所有请求失败，不过不会影响到所有请求，所以降低了风险。

**需要设置多少个worker**

Nginx 同redis类似都采用了io多路复用机制，每个worker都是一个独立的进程，但每个进程里只有一个主线程，通过异步非阻塞的方式来处理请求， 即使是千上万个请求也不在话下。每个worker的线程可以把一个cpu的性能发挥到极致。

所以worker数和服务器的cpu数相等是最为适宜的。设少了会浪费cpu，设多了会造成cpu频繁切换上下文带来的损耗。

**#设置worker数量**

worker\_processes 4

#work绑定cpu(4 work绑定4cpu)。

worker\_cpu\_affinity 0001 0010 0100 1000

#work绑定cpu (4 work绑定8cpu中的4个) 。

worker\_cpu\_affinity 00000001 00000010 00000100 00001000 00010000 00100000 01000000 10000000

**#连接数**

**worker\_connections 1024**

这个值是表示每个worker进程所能建立连接的最大值，所以，一个nginx能建立的最大连接数，应该是worker\_connections \* worker\_processes。当然，这里说的是最大连接数，对于HTTP请求本地资源来说，能够支持的最大并发数量是worker\_connections \* worker\_processes，如果是支持http1.1的浏览器每次访问要占两个连接，所以普通的静态访问最大并发数是： worker\_connections \* worker\_processes /2，而如果是HTTP作为反向代理来说，最大并发数量应该是worker\_connections \* worker\_processes/4。

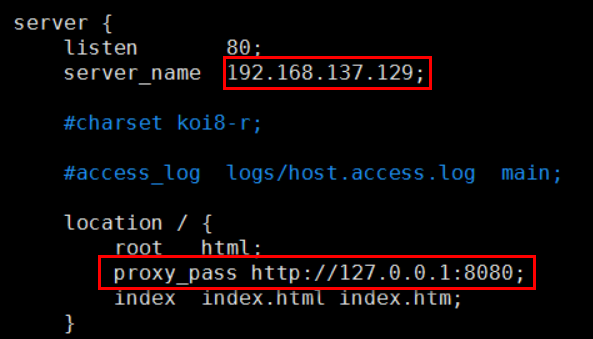
### 15、Nginx作用以及常见配置 难度系数：⭐

**作用：**反向代理、负载均衡、动静分离（静态资源服务器，可以进行图片等资源管理）

**具体配置实现：**

**反向代理：**

在 nginx.conf 配置文件中增加如下配置



**负载均衡：**

**常见策略**

**权重：**weight代表权,重默认为1,权重越高被分配的客户端越多

指定轮询几率，weight和访问比率成正比，用于后端服务器性能不均的情况。 例如：

upstream server\_pool{

server 192.168.5.21 weight=1;

server 192.168.5.22 weight=2;

server 192.168.5.23 weight=3;

}

**ip\_hash：**每个请求按访问ip的hash结果分配，这样每个访客固定访问一个后端服务器，可以解决session的问题。 例如：

upstream server\_pool{

ip\_hash;

server 192.168.5.21:80;

server 192.168.5.22:80;

}

**fair（第三方）：**按后端服务器的响应时间来分配请求，响应时间短的优先分配。

upstream server\_pool{

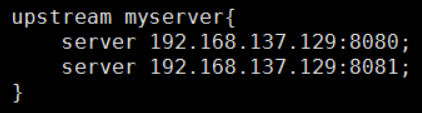
server 192.168.5.21:80;

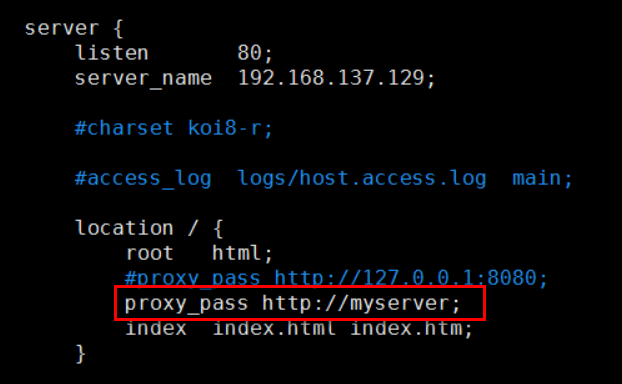
server 192.168.5.22:80;

fair;

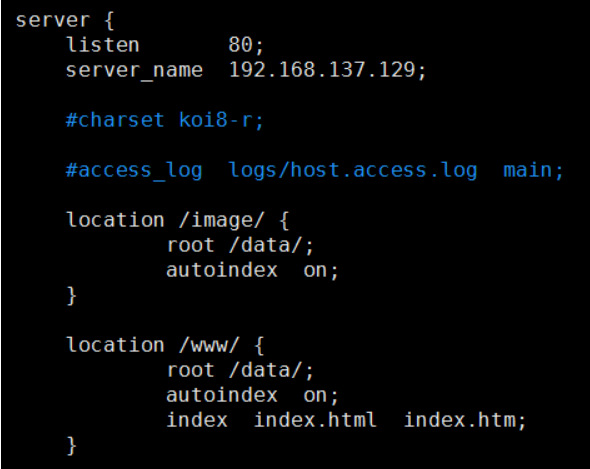
}

**轮询（默认）**





**静态资源（动静分离）：**



第七章 项目管理工具Git篇

### 1、工作中git开发使用流程（命令版描述）

开发一个新功能流程: （master线上分支，dev测试分支）

git clone 注释1

git checkout -b product 新建一个product 分支并且切换到product 分支

git add ./ 提交开发需求到暂存区域

git commit -m '开发商品模块'

git push origin pengyu

git co test //切换到test分支

git merge pengyu //带你开发的业务代码合并到test分支

git push origin test //带你开发的业务代码推送到远端的test分支

### 2、Reset 与Rebase,Pull 与 Fetch 的区别

git reset 不修改commit相关的东西，只会去修改.git目录下的东西。

git rebase 会试图修改你已经commit的东西，比如覆盖commit的历史等，但是不能使用rebase来修改已经push过的内容，容易出现兼容性问题。rebase还可以来解决内容的冲突，解决两个人修改了同一份内容，然后失败的问题。

git pull pull=fetch+merge,

使用git fetch是取回远端更新，不会对本地执行merge操作，不会去动你的本地的内容。 pull会更新你本地代码到服务器上对应分支的最新版本

## 3、git merge和git rebase的区别

git merge把本地代码和已经取得的远程仓库代码合并。

git rebase是复位基底的意思，gitmerge会生成一个新的节点，之前的提交会分开显示，而rebase操作不会生成新的操作，将两个分支融合成一个线性的提交。

## 4、git如何解决代码冲突

第一种：

git stash

git pull

git stash pop

这个操作就是把自己修改的代码隐藏，然后把远程仓库的代码拉下来，然后把自己隐藏的修改的代码释放出来，让gie自动合并。

如果要代码库的文件完全覆盖本地版本。

git reset –hard

git pull

第二种：通过开发工具 idea进行merge代码合并

### 5、项目开发时git分支情况

主干分支master：主要负责管理正在运行的生产环境代码。永远保持与正在运行的生产环境完全一致。

开发分支develop：主要负责管理正在开发过程中的代码。一般情况下应该是最新的代码。

bug修理分支hotfix：要负责管理生产环境下出现的紧急修复的代码。 从主干分支分出，修理完毕并测试上线后，并回主干分支。并回后，视情况可以删除该分支。

发布版本分支release：较大的版本上线前，会从开发分支中分出发布版本分支，进行最后阶段的集成测试。该版本上线后，会合并到主干分支。生产环境运行一段阶段较稳定后可以视情况删除。

功能分支feature：为了不影响较短周期的开发工作，一般把中长期开发模块，会从开发分支中独立出来。 开发完成后会合并到开发分支。

[注释1] 克隆远端仓库代码到本地

第八章 Linux

### 1、Linux常用命令

|  |  |  |
| --- | --- | --- |
| 序号 | 命令 | 命令解释 |
| 1 | top | 查看内存 |
| 2 | df -h | 查看磁盘存储情况 |
| 3 | iotop | 查看磁盘IO读写(yum install iotop安装） |
| 4 | iotop -o | 直接查看比较高的磁盘读写程序 |
| 5 | netstat -tunlp | grep 端口号 | 查看端口占用情况 |
| 6 | uptime | 查看报告系统运行时长及平均负载 |
| 7 | ps aux | 查看进程 |

### 2、如何查看测试项目的日志

一般测试的项目里面，有个logs的目录文件，会存放日志文件，有个xxx.out的文件，可以用tail -f 动态实时查看后端日志

先cd 到logs目录(里面有xx.out文件)

>tail -f xx.out

这时屏幕上会动态实时显示当前的日志，ctr+c停止

### 3、如何查看最近1000行日志

>tail -1000 xx.out

### 4、Linux中如何查看某个端口是否被占用

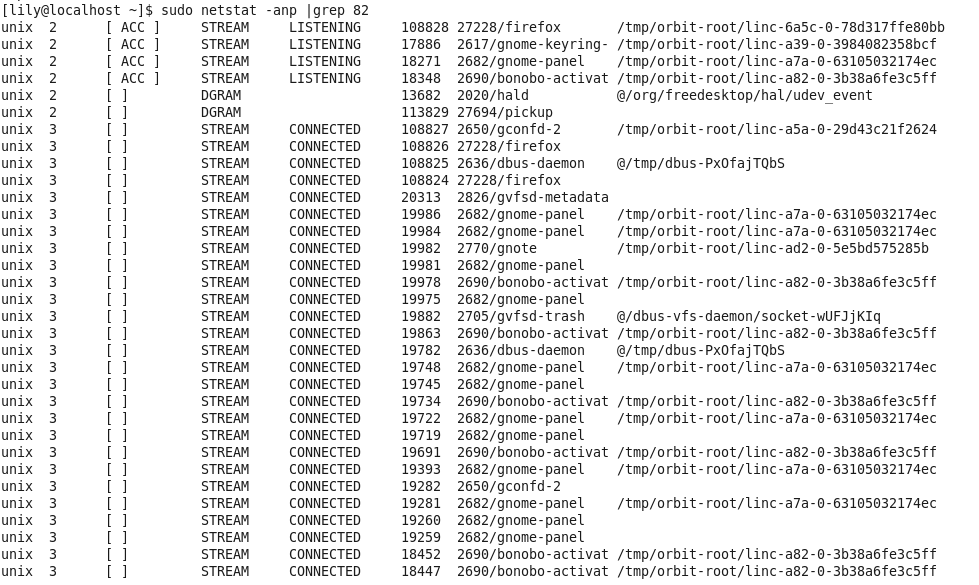
>netstat -anp | grep 端口号



图中主要看监控状态为LISTEN表示已经被占用，最后一列显示被服务mysqld占用，查看具体端口号，只要有如图这一行就表示被占用了

查看82端口的使用情况，如图

>netstat -anp |grep 82

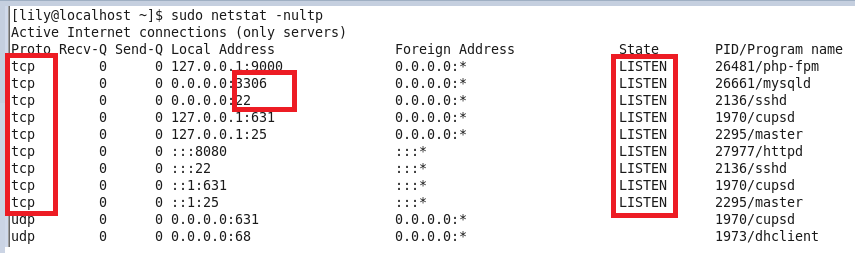


可以看出并没有LISTEN那一行，所以就表示没有被占用。此处注意，图中显示的LISTENING并不表示端口被占用，不要和LISTEN混淆哦，查看具体端口时候，必须要看到tcp，端口号，LISTEN那一行，才表示端口被占用了

### 5、查看当前所有已经使用的端口情况

如图：

netstat -nultp（此处不用加端口号）



### 6、怎么用一条命令杀掉端口号为8080的进程

第九章 电商项目篇

### 1、介绍下最近做的项目

可以从2个方向出发

介绍项目背景、项目功能和自己负责的功能模块

介绍项目背景、项目使用技术栈和自己负责的功能模块

#### 1.1 项目背景：

可以介绍项目是什么类型的（B2C、B2B2C、O2O这类），为什么要做这个项目，

有工作经验的找工作一般这样介绍：项目是自己公司开发，自己运营的，然后不断加功能进行迭代和维护；或者是项目定制的，给甲方客户开发的一个项目，上线后不负责维护和迭代，这样避免了很多后期问题，这两种都可以。

咱们可以借鉴以上的方式去介绍，刨除公司情况，以学习为主。

#### 1.2 项目功能：

结合项目，进行主要的功能模块阐述，可以结合电商项目的核心购物流程去说：后台管理系统（商品的管理）、商品详情、商品搜索、购物车、单点登录+社交登录、订单、支付、秒杀等等。

#### 1.3 技术栈：

使用springboot整合SpringCloud 以及MyBatis-Plus进行微服务构建，使用nacos作为注册中心和配置中心，使用feign进行服务远程调用，使用gateway网关进行请求负载、请求过滤、统一鉴权和限流，使用Sentinel进行服务的熔断和降级，使用Spring Cloud Sleuth进行链路追踪，针对于项目图片文件资源较多，采用FastDFS进行文件资源存储，使用redis数据库进行数据缓存以及分布式锁的实现，使用ElasticSearch进行商品的搜索业务实现…..（这块基础架构说完后，主要结合自己负责的功能模块去说技术点的应用）

#### 1.4 自己负责的功能模块：

已简历为主，简历上写了哪几个，就说那几个，一定要知道自己简历写的什么内容。

#### 1.5 项目介绍参考：

尚品汇商城是B2C模式的综合性在线销售平台。商城分为后台管理部分与用户前台使用部分。后台管理部分包括：商品管理模块（商品分类、品牌、平台属性、SPU与SKU以及销售属性、商品上下架和商品评论管理等）、内容广告模块、库存管理模块、订单管理模块、促销管理（秒杀等商品设置）、客户模块、统计报表模块和系统基础权限等模块。

用户前台使用部分：商城首页、商品搜索（可按条件查询展示）、商品详情信息展示、购物车、用户单点登录和社交登录（微信登录）、用户会员中心、订单的创建修改、展示以及在线支付（支付宝、微信）、物流模块、商品评论以及秒杀活动等功能。

#### 1.6 项目架构图：



#### 1.7 整体业务介绍：

|  |  |
| --- | --- |
| 首页 | 静态页面，包含了商品分类，搜索栏，商品广告位。 |
| 全文搜索 | 通过搜索栏填入的关键字进行搜索，并列表展示 |
| 分类查询 | 根据首页的商品类目进行查询 |
| 商品详情 | 商品的详细信息展示 |
| 购物车 | 将有购买意向的商品临时存放的地方 |
| 单点登录 | 用户统一登录的管理 |
| 结算 | 将购物车中勾选的商品初始化成要填写的订单 |
| 下单 | 填好的订单提交 |
| 支付服务 | 下单后，用户点击支付，负责对接第三方支付系统。 |
| 订单服务 | 负责确认订单是否付款成功，并对接仓储物流系统。 |
| 仓储物流 | 独立的管理系统，负责商品的库存。 |
| 后台管理 | 主要维护类目、商品、库存单元、广告位等信息。 |
| 秒杀 | 秒杀抢购完整方案 |

#### 1.8 后台管理系统功能：

电子商务网站整个系统的后端管理，按功能划分为九大模块，包括商品组织管理、订单处理、内容发布管理等模块。

##### 1.8.1 后台主页：

各类主要信息的概要统计，包括客户信息、 订单信息、商品信息、库存信息、评论和最近反馈等。

##### 1.8.2 商品模块：

1).商品管理：

商品SPU和SKU的添加、修改、 删除、复制、批处理、商品计划上下架、SEO、商品多媒体上传等，可以定义商品是实体还是虚拟，可以定义是否预订、是否缺货销售等。

2).商品分类管理：

树形的商品目录组织管理，并可以设置品类关联与商品推荐。

3).商品平台属性管理：

定义商品的属性类型，设置自定义属性项。

4).品牌管理：

添加、修改、删除、上传品牌 LOGO。

5).商品评论管理：

商品评论的搜索、条件查询列表展示、回复、删除等功能。

##### 1.8.3 销售模块：

1).促销秒杀管理：

设置秒杀商品、购物车促销和 优惠券促销三类，可以随意定义不同的促销规则，满足日常促销活动：购物折扣、购物赠送积分、购物赠送优惠券、购物免运输费、特价商品、特定会员购买特定商品、折上折、买二送一等。

2).礼券、积分管理：

比如添加、发送礼券和积分

3).关联/推荐管理：

基于规则引擎，可以支持多种推荐类型，可手工添加或者自动评估商品。

##### 1.8.4 订单模块：

1).订单管理：

可以编辑、解锁、取消订单、 拆分订单、添加商品、移除商品、确认可备货等，也可对因促销规则发生变化引起的价格变化进行调整。订单处理完可发起退货、换货流程。

2).支付：

常用于订单支付信息的查看和手工 支付两种功能。手工支付订单，常用于“款到发货”类型的订单，可理解为对款到发货这类订单的一种补登行为。

3).结算：

提供商家与第三方物流公司的结算 功能，通常是月结。同时，结算功能也是常用来对“货到付款”这一类型订单支付后的数据进行对帐

##### 1.8.5 库存模块：

1).库存管理：

引入库存的概念，不包括销售规则为永远可售的商品，一个SKU对应一个库存量。库存管理提供增加、减少等调整库存量的功能;另外，也可对具具体的SKU设置商品的保留数量、最小库存量、再进货数量。每条SKU商品的具体库存操作都会记录在库存明细记录里边。

2).查看库存明细记录。

3).备货/发货：

创建备货单、打印备货单、打印发货单、打印快递单、完成发货等一系列物流配送的操作。

4).退/换货：

对退/换货的订单进行收货流程的处理。

##### 1.8.6 内容模块：

1).内容管理：

包括内容管理以及内容目录管理。内容目录由树形结构组织管理。类似于商品目录的树形结构，可设置目录是否为链接目录。

2).广告管理：

添加、修改、删除、上传广告、 定义广告有效时限。

3).可自由设置商城导航栏目以及栏目内容、栏目链接。

##### 1.8.7 客户模块：

1).客户管理：

添加、删除、修改、重设密码、 发送邮件等。  

2).反馈管理：

删除、回复。

3).消息订阅管理：

添加、删除、修改消息组 和消息、分配消息组、查看订阅人。

4).会员资格：

添加、删除、修改。

##### 1.8.8 系统模块：

1).安全管理：

管理员、角色权限分配和安全日志

2).系统属性管理：

用于管理自定义属性。可关联模块包括商品管理、商品目录管理、内容管理、客户管理。

3).运输与区域：

运输公司、运输方式、运输 地区。

4).支付管理：

支付方式、支付历史。

5).包装管理：

添加、修改、删除。

6).数据导入管理：

商品目录导入、商品导入、 会员资料导入。

##### 1.8.9 报表模块：

   缺省数个统计报表，支持时间段过滤、支持按不同状态过滤、支持HTML、PDF和Excel格式的导出和打印。

1.用户注册统计   2.低库存汇总   3.缺货订单   4.订单汇总   5.退换货

### 2、项目开发周期：

**开发、维护和运营一体化的：**开发周期8个月左右，后期维护与迭代时间会更长

**项目定制：**前期架构+数据库设计+编码+开发+测试解bug共7个月左右进行项目交付

**培训学习项目:** 20天教程（咱们是学习20天课程）

### 3、项目参与人数：

**一般公司:**项目经理（PM）1人、产品（PD）2人、界面设计（UI）2人、前端 3人、Java后台（DE）6人，其中1人是开发组长、测试（QA）2人、运维（SRE）1人

**培训学习项目：**根据课程内容编写代码，自己实现部分功能

### 4、公司开发相关各岗位职责：

4.1 项目经理（PM）：

企业建立以项目经理责任制为核心，对项目实行质量、安全、进度、成本管理的责任保证体系和全面提高项目管理水平设立的重要管理岗位。职责：

1、负责软件项目管理及计划实施；

2、具备较强管理、协调及沟通能力，帮助开发人员解决开发过程中遇到的技术问题，做好日常的开发团队管理工作；

3、与各团队协同工作，确保开发工作正常顺利的开展；

4.2 产品（PD）：

企业中专门负责产品管理的职位，负责调查并根据用户的需求，确定开发何种产品，选择何种技术、商业模式等。并推动相应产品的开发组织，他还要根据产品的生命周期，协调研发、营销、运营等，确定和组织实施相应的产品策略，以及其他一系列相关的产品管理活动。**职责：**

1. 根据公司产品及用户需求，结合市场调研情况，进行产品规划；

2. 负责用户沟通、需求分析诊断；

3. 负责产品定位、用户体验流程定位及产品设计；

4. 推动、协调与控制产品策划及研发工作，保证产品需求的有效实现；

5. 负责产品持续升级，不断提升用户满意度及忠诚度；

6. 对行业及竞争产品的分析，跟踪最新发展趋势，并提交分析报告。

4.3 界面设计（UI）：

对软件的人机交互、操作逻辑、界面美观的整体设计。职责：

1、负责公司产品PC端和移动端的UI界面设计工作；

2、配合完成校样修改和界面调整；

3、深入了解负责的产品，并通过各种设计形式和视觉语言让用户感受到产品的优点和特性；

4、跟进设计的变化和需求，注重相关文档的整理、资料的收集；能独立完成界面设计工作。

4.4 开发组长（TL）：

其实就是个更小一点的项目经理。其职责：

1、 参与软件的设计负责系统需求的分析，进行系统设计和数据库设计；

2、 解决开发过程中技术问题和提供解决办法；

3、 能够带领小组负责模块的功能开发；

4、 负责项目组代码的审查工作，有效地控制项目的质量风险。

4.5 测试（QA）：

测试工程师，软件质量的把关者，工作起点高，发展空间大。职责： 1.理解、分析需求文档，挖掘、细化需求;

2.根据软件需求及设计文档编写测试用例，参与文档评审并维护相关文档;

3.准备测试数据，执行测试用例，记录测试结果，整理测试报告;

4.负责BUG的提交、跟踪、验证、关闭;

5.负责测试部门测试环境及BUG系统管理与维护。

6.对产品进行必要的功能，性能，安全，兼容性及其它方面的测试工作；

7.公司安排的其它工作。

4.6 运维（SRE）：

运维工程师最基本的职责都是负责服务的稳定性。

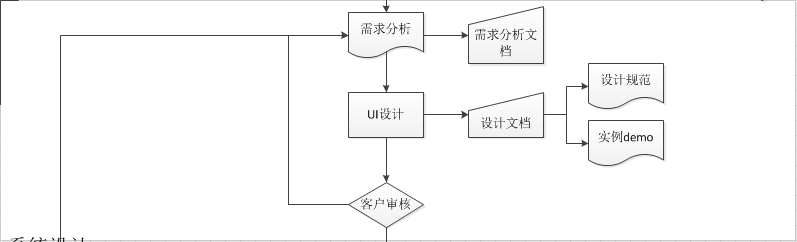
1. 产品发布前：负责参与并审核架构设计的合理性和可运维性，以确保在产品发布之后能高效稳定的运行。

2. 产品发布阶段：负责用自动化的技术或者平台确保产品可以高效的发布上线，之后可以快速稳定迭代。

3. 产品运行维护阶段：负责保障产品7\*24H稳定运行，在此期间对出现的各种问题可以快速定位并解决；在日常工作中不断优化系统架构和部署的合理性，以提升系统服务的稳定性。

### 5、项目开发流程:

5.1 需求分析



项目前期主要指的是项目业务需求调研、包括配合用户制定项目建设方案、技术规范书、配合市场人员进行售前技术交流等环节，此阶段应该组织由售前工程师、需求分析师以及系项目经理等组成一个临时小组，负责跟踪项目。这个小组根据项目的大小和客户的要求确定小组成员。

项目前期小组的工作是项目的开始，这个小组工作成绩的优劣、工作质量的高低，将直接影响项目的成败。因此，从管理层的角度，一定要重视这个环节。

项目前期小组需要完成的工作包括以下方面：

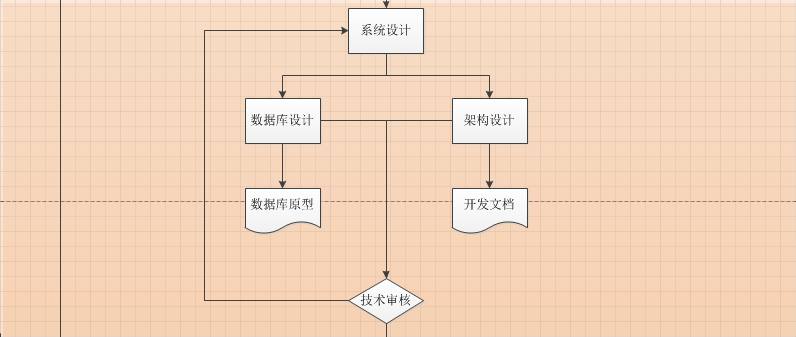
1、 客户的各种项目前期要求，如：方案介绍、业务需求编写等

2、 提交项目可行性分析报告，包括成本/效益分析

3、 提交项目建议方案

4、 提交业务需求说明书或需求分析说明书

5.2 系统设计



系统设计是决定项目或软件系统“怎样做”的过程，这个过程回答了系统应该如何实现的问题。从软件工程的角度，设计阶段大约是整个项目开发成本的25%，所以，设计团队以及该团队的工作成绩对于整个系统来说至关重要。

人员需求

设计团队一般由3—8名设计人员组成，从这个阶段起，项目需要一名项目经理，行使项目组的各种管理职能。设计团队的成员具体包括：

1名项目经理

包括1—2名项目前期成员

1名系统构架师

1名数据库设计人员

1名用户界面设计人员组成

设计团队需要完成的工作包括：

1、项目开发计划

2、确定系统软硬件配置最佳方案

3、确定系统开发平台以及开发工具

4、确定系统软件结构

5、确定系统功能模块以及各个模块之间的关系

6、确定系统测试方案

7、提交系统数据库设计方案

8、提交系统概要设计文档

由于应用软件需求经常变化，因此设计需要考虑系统可扩展性，并需要在设计过程中对于重要的环节和用户进行及时沟通。

5.3 编码开发



将用户的需求变成真正可用的软件系统，是通过编码和系统实现阶段来完成的。虽然软件的质量主要取决于系统设计的质量，但是编码的途径和实现的具体方法对程序的可靠性、可读性、可测试性和可维护性产生深远的影响。

人员需求

这个阶段要根据用户对项目进度的要求灵活组织开发团队。为了工作的连贯性，同时也为了解决在开发过程中用户需求有可能变化的因素，开发团队应该保留1—3名设计团队的成员。

人员分工

开发过程中，项目经理的角色非常重要，项目经理负责项目组开发人员的日常管理，控制项目的进度，负责和设计部门、市场部门以及客户之间进行必要的沟通。这个阶段通常是多个部门的人员共同组成一个项目组，因此，项目管理的一定要保证统一管理，理想状态是项目经理全权负责项目组人员的人员工作安排、业绩考核、工资奖金等，因为项目经理最了解项目组成员的工作态度和工作业绩。

一般在大型项目开发团队中，应该设立专门的技术经理岗位，负责对项目组的技术方案进行管控，技术经理最好是由设计团队中抽调出来。技术经理在项目开发过程中需要注意程序风格、编码规范等问题，并必须进行有效的代码管理（版本管理）。

开发过程还应该进行系统的单元测试工作，确保各个独立模块功能的正确性和性能满足需求说明书的要求。

开发团队应该完成的工作包括：

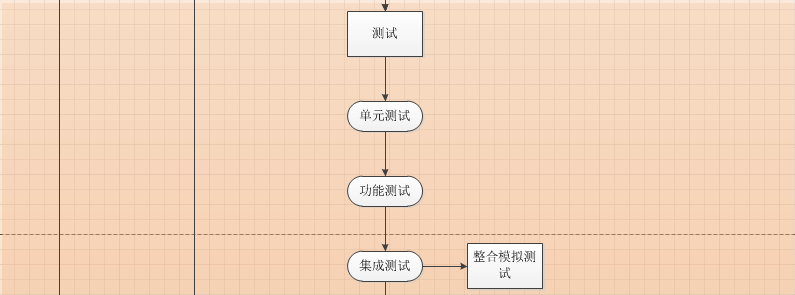
1、 系统的实现代码编写

2、 单元测试

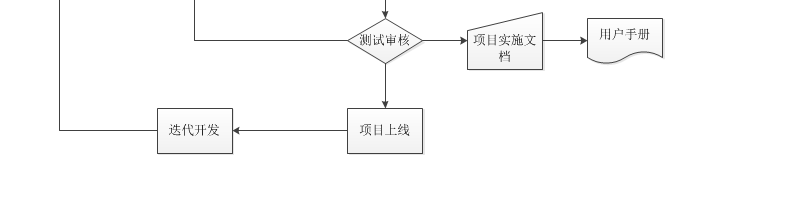
3、 提交源代码清单

4、 提交单元测试报告

5.4 系统测试



5.5 部署实施



由于从事的应用软件的开发，因此，在开发完成之后经常会有系统集成、软件的安装等工作。这个阶段还经常伴随着新的业务需求和本地化需求的产生，因此将会有一部分的开发工作需要在这个阶段完成。

### 6、项目版本控制：

使用git进行版本控制，剩下的主要是对git仓库的一些列操作，要记住操作命令，以及分支切换等等。

### 7、一般项目服务器数量：

开发测试阶段：

开发在自己电脑上开发，代码提交到git仓库（gitlab、gitee等）的开发分支，到测试时，公司有1-2台测试服务器，把开发分支代码合并到测试分支，使用jenkins进行测试环境代码部署，测试人员进行测试。

生产环境：

为了保证高可用，首先每个服务都要进行 集群部署，包括项目中依赖的第三方服务，这样的话，服务器的数量是很庞大的，以尚品汇商城所学功能实现为例，如下：

Nginx2台（主备）；Nacos 3台（官方推荐最少3台）；项目中有服务数量\*2（19\*2=38）共需要38台；redis无中心化集群6台（3主3从），如果是主从哨兵集群5台；mysql数据库（读写分离的情况下，1主2从 4\*3 =12）共12台；ES集群3台；rabbitmq集群2台；fastDFS集群（保证高可用情况下，traker2台，storage2组4台）共6台。

以上所述 服务器数量一共为72台，数量很多，以目前市场硬件服务器平均一台5万块钱来计算，72\*5=360万，投入成本和维护成本很大，在项目前期没有大用户量的情况下，不建议。

针对于这个问题，如果是项目定制（外包）的话，可以由甲方（客户）自己去部署，这种情况也是存在的。

还有一种情况是在项目前期没有那么大的用户量情况下，可以采用单机（主备）部署方案，把所有服务部署同一台服务器上，进行资源节省。

### 8、上线后QPS并发量，用户量、同时在线人数并发数等问题

**一般情况下,给客户定制的项目这些数据是拿不到的，项目给客户做的，甲方客户的运营数据是拿不到的。**

但是要了解以下数据关键词：

（1）QPS（TPS）：每秒钟request/事务 数量

（2）并发数： 系统同时处理的request/事务数

（3）响应时间： 一般取平均响应时间

QPS（TPS）= 并发数/平均响应时间或者并发数 = QPS\*平均响应时间 一个典型的上班签到系统，早上8点上班，7点半到8点的30分钟的时间里用户会登录签到系统进行签到。公司员工为1000人，平均每个员上登录签到系统的时长为5分钟。可以用下面的方法计算。 QPS = 1000/(30\*60) 事务/秒 平均响应时间为 = 5\*60 秒 并发数= QPS\*平均响应时间 = 1000/(30\*60) \*(5\*60)=166.7

说明：

一个系统吞吐量通常由QPS（TPS）、并发数两个因素决定，每套系统这两个值都有一个相对极限值，在应用场景访问压力下，只要某一项达到系统最高值，系统的吞吐量就上不去了，如果压力继续增大，系统的吞吐量反而会下降，原因是系统超负荷工作，上下文切换、内存等等其它消耗导致系统性能下降。

我们做项目要排计划，可以多人同时并发做多项任务，也可以一个人或者多个人串行工作，始终会有一条关键路径，这条路径就是项目的工期。系统一次调用的响应时间跟项目计划一样，也有一条关键路径，这个关键路径是就是系统影响时间； 关键路径是有CPU运算、IO、外部系统响应等等组成。

如果非要说，以下提供一套参数，但是仅供参考，可以根据自己设计适当调整：

用户总量 几万+，日活 3000+，月活 12W+，一个月PV 30W+，并发量 500+

### 9、你们项目的微服务是怎么拆分的，拆分了多少？

根据功能模块进行拆分，有多少功能模块就基本上拆出来多少个服务。

### 10、如何解决并发问题的？

尚品汇商城是微服务架构构建，集群部署，数据库的分库分表读写分离，Redis缓存，RabbitMQ消息异步解耦，页面静态化等这些都是解决并发的手段。

开发层面：微服务架构、缓存（Redis）、异步（MQ）、队排好（限流和削峰）

部署层面：集群（高可用）和负载均衡----Nginx、Gateway、Feign

硬件层面：CPU性能、硬盘（SSD）性能、内存大小

网络层面：增加网络带宽、网络加速器

### 11、如何保证接口的幂等性？

1. 根据状态机很多时候业务表是有状态的，比如订单表中有：1-下单、2-已支付、3-完成、4-撤销等状态。如果这些状态的值是有规律的，按照业务节点正好是从小到大，我们就能通过它来保证接口的幂等性。假如id=123的订单状态是已支付，现在要变成完成状态。update `order` set status=3 where id=123 and status=2;第一次请求时，该订单的状态是已支付，值是2，所以该update语句可以正常更新数据，sql执行结果的影响行数是1，订单状态变成了3。后面有相同的请求过来，再执行相同的sql时，由于订单状态变成了3，再用status=2作为条件，无法查询出需要更新的数据，所以最终sql执行结果的影响行数是0，即不会真正的更新数据。但为了保证接口幂等性，影响行数是0时，接口也可以直接返回成功。

具体步骤：

1 用户通过浏览器发起请求，服务端收集数据。

2 根据id和当前状态作为条件，更新成下一个状态

3 判断操作影响行数，如果影响了1行，说明当前操作成功，可以进行其他数据操作。

4 如果影响了0行，说明是重复请求，直接返回成功。

主要特别注意的是，该方案仅限于要更新的表有状态字段，并且刚好要更新状态字段的这种特殊情况，并非所有场景都适用。

2. 加分布式锁其实前面介绍过的加唯一索引或者加防重表，本质是使用了数据库的分布式锁，也属于分布式锁的一种。但由于数据库分布式锁的性能不太好，我们可以改用：redis或zookeeper。鉴于现在很多公司分布式配置中心改用apollo或nacos，已经很少用zookeeper了，我们以redis为例介绍分布式锁。目前主要有三种方式实现redis的分布式锁：

1 setNx命令

2 set命令

3 Redission框架

具体步骤：

1 用户通过浏览器发起请求，服务端会收集数据，并且生成订单号code作为唯一业务字段。

2 使用redis的set命令，将该订单code设置到redis中，同时设置超时时间。

3 判断是否设置成功，如果设置成功，说明是第一次请求，则进行数据操作。

4 如果设置失败，说明是重复请求，则直接返回成功。

3. 获取token

除了上述方案之外，还有最后一种使用token的方案。该方案跟之前的所有方案都有点不一样，需要两次请求才能完成一次业务操作。

第一次请求获取token

第二次请求带着这个token，完成业务操作。

具体步骤：

1 用户访问页面时，浏览器自动发起获取token请求。

2 服务端生成token，保存到redis中，然后返回给浏览器。

3 用户通过浏览器发起请求时，携带该token。

4 在redis中查询该token是否存在，如果不存在，说明是第一次请求，做则后续的数据操作。

5 如果存在，说明是重复请求，则直接返回成功。

6 在redis中token会在过期时间之后，被自动删除。

### 12、你们项目中有没有用到什么设计模式？

这个建议提前去了解几种常见的设计模式及其应用场景，将其套用到项目的某个场景中，以下提供几种常见的设计模式及其应用场景

1) 单例模式。

单例模式是一种常用的软件设计模式。

在它的核心结构中只包含一个被称为单例类的特殊类。通过单例模式可以保证系统中一个类只有一个实例而且该实例易于外界访问，从而方便对实例个数的控制并节约系统资源。

应用场景：如果希望在系统中某个类的对象只能存在一个，单例模式是最好的解决方案。

2) 工厂模式。

工厂模式主要是为创建对象提供了接口。

应用场景如下：

a、 在编码时不能预见需要创建哪种类的实例。

b、 系统不应依赖于产品类实例如何被创建、组合和表达的细节。

3) 策略模式。

策略模式：定义了算法族，分别封装起来，让它们之间可以互相替换。此模式让算法的变化独立于使用算法的客户。

应用场景如下。

a、 一件事情，有很多方案可以实现。

b、我可以在任何时候，决定采用哪一种实现。

c.、未来可能增加更多的方案。

d、 策略模式让方案的变化不会影响到使用方案的客户。

举例业务场景如下。

系统的操作都要有日志记录，通常会把日志记录在数据库里面，方便后续的管理，但是在记录日志到数据库的时候，可能会发生错误，比如暂时连不上数据库了，那就先记录在文件里面。日志写到数据库与文件中是两种算法，但调用方不关心，只负责写就是。

4) 观察者模式。

观察者模式又被称作发布/订阅模式，定义了对象间一对多依赖，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新。

应用场景如下：

a、对一个对象状态的更新，需要其他对象同步更新，而且其他对象的数量动态可变。

b、对象仅需要将自己的更新通知给其他对象而不需要知道其他对象的细节。

5) 迭代器模式。

迭代器模式提供一种方法顺序访问一个聚合对象中各个元素，而又不暴露该对象的内部表示。

应用场景如下：

当你需要访问一个聚集对象，而且不管这些对象是什么都需要遍 历的时候，就应该考虑用迭代器模式。其实stl容器就是很好的迭代器模式的例子。

6) 模板方法模式。

模板方法模式定义一个操作中的算法的骨架，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步骤。

应用场景如下：

对于一些功能，在不同的对象身上展示不同的作用，但是功能的框架是一样的。

### 13、生产环境出问题，你们是怎么排查的？

生产环境出问题一般由多方面引起，可以由多方面进行排查，可以借鉴以下链接文章进行参考，面试的时候只需提出大概的思路即可：

https://blog.csdn.net/GitChat/article/details/79019454

### 14、你做完这个项目后有什么收获？

首先，在数据库方面，我现在是真正地体会到数据库的设计真的是一个程序或软件设计的重要和根基。因为数据库怎么设计，直接影响到一个程序或软件的功能的实现方法、性能和维护。由于我做的模块是要对数据库的数据进行计算和操作的，所以我对数据库的设计对程序的影响是深有体会，就是因为我们的数据库设计得不好，搞得我在对数据库中的数据进行获取和计算利润、总金时，非常困难，而且运行效率低，时间和空间的复杂也高，而且维护起来很困难，过了不久，即使自己有注释，但是也要认真地看自己的代码才能明白自己当初的想法和做法。加上师兄的解说，让我对数据库的重要的认识更深一层，数据库的设计真的是重中之重。

其次，就是分工的问题。虽然这次的项目我们没有在四人选出一个组长，但是，由于我跟其他人都比较熟，也有他们的号码，然后我就像一个小组长一样，也是我对他们进行了分工。俗话也说，分工合作，分好了工，才能合作。但是这次项目，我们的分工却非常糟糕，我们在分工之前分好了模块，每个 责哪些模块。本以为我们的分工是明确的，后来才发现，我们的分工是那么的一踏糊涂，一些功能上紧密相连的模块分给了两个人来完成，使两个人都感到迷惘，不知道自己要做什么，因为两个人做的东西差不多。我做的，他也在做，那我是否要继续做下去？总是有这样的疑问。从而导致了重复工作，浪费时间和精力，并打击了队员的激情，因为自己辛辛苦苦写的代码，最后可能没有派上用场。我也知道，没有一点经验的我犯这样的错是在所难免，我也不过多地怪责自己，吸取这次的教训就好。分工也是一门学问。 再者，就是命名规范的问题。可能我们以前都是自己一个人在写代码，写的代码都是给自己看的，所以我们都没有注意到这个问题。就像师兄说的那样，我们的代码看上去很上难看很不舒服，也不知道我们的变量是什么类型的，也不知道是要来做什么的。但是我觉得我们这一组人的代码都写得比较好看，每个人的代码都有注释和分隔，就是没有一个统一的规范，每个人都人自己的一个命名规则和习惯，也不能见名知义。还有就是没有定义好一些公共的部分，使每个人都有一个自己的“公共部分”，从而在拼起来时，第一件事，就是改名字。而这些都应该是在项目一开始，还没开始写代码时应该做的。 然后，我自己在计算时，竟然太大意算错了利润，这不能只一句我不小心就敷衍过去，也是我的责任，而且这也是我们的项目的核心部分，以后在做完一个模块后，一定要测试多次，不能过于随便地用一个数据测试一下，能成功就算了，要用可能出现的所有情况去测试程序，让所有的代码都有运行过一次，确认无误。

最后，也是我比较喜欢的东西，就是大家一起为了一个问题去讨论和去交流。因为我觉得，无论是谁，他能想的东西都是有限的，别人总会想到一些自己想不到的地方。跟他人讨论和交流能知道别人的想法、了解别人是怎样想一个问题的，对于同样的问题自己又是怎样想的，是别人的想法好，还是自己的想法好，好在什么地方。因为我发现问题的能力比较欠缺，所以我也总是喜欢别人问我问题，也喜欢跟别人去讨论一个问题，因为他们帮我发现了我自己没有发现的问题。在这次项目中，我跟植荣的讨论就最多了，很多时候都是不可开交的那种，不过我觉得他总是能够想到很多我想不到的东西，他想的东西也比我深入很多，虽然很多时候我们好像闹得很僵，但是我们还是很要好的! 嘻嘻！而且在以后的学习和做项目的过程中，我们遇到的问题可能会多很多，复杂很多，我们一个人也不能解决，或者是没有想法，但是懂得与他人讨论与交流就不怕这个问题，总有人的想法会给我们带来一片新天地。相信我能做得更好。 还有就是做项目时要抓准客户的要求，不要自以为是，自己觉得这样好，那样好就把客户的需求改变，项目就是项目，就要根据客户的要求来完成。

### 15、在做这个项目的时候你碰到了哪些问题？你是怎么解决的？

（1）开发SpringBoot接口出现客户端和服务端不同步，导致接口无法测试，产生的原因沟通不畅。

（2）订单提交时由于本地bug或者意外故障导致用户钱支付了但是订单不成功，采用对账方式来解决。

（3）上线的时候一定要把支付的假接口换成真接口。

（4）项目中用到了曾经没有用过的技术，解决方式：用自己的私人时间主动学习

（5）在开发过程中与测试人员产生一些问题，本地环境ok但是测试环境有问题，环境的问题产生的，浏览器环境差异，服务器之间的差异

（6）系统运行环境问题，有些问题是在开发环境下OK，但是到了测试环境就问题，比如说系统文件路径问题、导出报表中的中文问题（报表采用POI），需要在系统jdk中添加相应的中文字体才能解决；

第十章 数据结构和算法篇

### 1、怎么理解时间复杂度和空间复杂度

时间复杂度和空间复杂度一般是针对算法而言，是衡量一个算法是否高效的重要标准。先纠正一个误区，时间复杂度并不是算法执行的时间，再纠正一个误区，算法不单单指冒泡排序之类的，一个循环甚至是一个判断都可以称之为算法。其实理解起来并不冲突，八大排序甚至更多的算法本质上也是通过各种循环判断来实现的。

**时间复杂度：**指算法语句的执行次数。O(1),O(n),O(logn),O(n2)

**空间复杂度：**就是一个算法在运行过程中临时占用的存储空间大小，换句话说就是被创建次数最多的变量，它被创建了多少次，那么这个算法的空间复杂度就是多少。有个规律，如果算法语句中就有创建对象，那么这个算法的时间复杂度和空间复杂度一般一致，很好理解，算法语句被执行了多少次就创建了多少对象。

### 2、数组和链表结构简单对比

**数组：**相同数据类型的元素按一定顺序排列的集合，就是把有限个类型相同的变量用一个名字命名，然后用编号区分他们的变量的集合，这个名字称为数组名，编号称为下标

**数组的特性：**

1.数组必须先定义固定长度，不能适应数据动态增减

2.当数据增加时，可能超出原先定义的元素个数，当数据减少时，造成内存浪费

3.数组查询比较方便，根据下标就可以直接找到元素，时间复杂度O(1)；增加和删除比较复杂，需要移动操作数所在位置后的所有数据，时间复杂度为O(N)

**链表：**是一种物理存储单元上非连续，非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。

**链表的特性：**

1.链表动态进行存储分配，可适应数据动态增减

2.插入、删除数据比较方便,时间复杂度O(1)；查询必须从头开始找起，十分麻烦，时间复杂度O(N)

常见的链表:

1.单链表：通常链表每一个元素都要保存一个指向下一个元素的指针

2.双链表：每个元素既要保存到下一个元素的指针，还要保存一个上一个元素的指针

3.循环链表：在最后一个元素中下一个元素指针指向首元素

链表和数组都是在堆里分配内存

**应用：**

如果需要快速访问数据，很少或不插入和删除元素，就应该用数组；相反， 如果需要经常插入和删除元素就需要用链表数据结构了

### 3、怎么遍历一个树

四种遍历概念

先序遍历：先访问根节点，再访问左子树，最后访问右子树。

后序遍历：先左子树，再右子树，最后根节点。

中序遍历：先左子树，再根节点，最后右子树。

层序遍历：每一层从左到右访问每一个节点。

每一个子树遍历时依然按照此时的遍历顺序。可以采用递归实现遍历。

### 4、冒泡排序（Bubble Sort）

**算法描述：**

* 比较相邻的元素。如果第一个比第二个大，就交换它们两个；
* 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；
* 针对所有的元素重复以上的步骤，除了最后一个；
* 重复步骤1~3，直到排序完成。



如果两个元素相等，不会再交换位置，所以冒泡排序是一种稳定排序算法。

**代码实现：**

package  com.atguigu.interview.chapter02;

Java

/\*\*

Java

 \* @author atguigu

Java

 \* @since 2019/7/22

Java

 \* 冒泡排序

Java

 \*/

Java

publc class BubbleSort {

Java

     /\*\*

Java

     \* @param data 被排序的数组

Java

     \*/

Java

     public   static   void  bubbleSort( int [] data) {

Java

         int  arrayLength = data.length;

Java

         for  ( int  i =  1 ; i < arrayLength; i++) { //第i次排序

Java

             for  ( int  j =  0 ; j < arrayLength - i; j++) { //从索引为j的数开始

Java

                 if  (data[j] > data[j +  1 ]) {  //相邻元素两两对比

Java

                     int  temp = data[j +  1 ];   // 元素交换

Java

                    data[j +  1 ] = data[j];

Java

                    data[j] = temp;

Java

                }

Java

            }

Java

            System.out.println( "第"  + i +  "次排序：\n"  + java.util.Arrays.toString(data));

Java

        }

Java

    }

Java

     public   static   void  main(String[] args) {

Java

         int [] data = { 3 ,  44 ,  38 ,  5 ,  47 ,  15 ,  36 ,  26 ,  27 ,  2 ,  46 ,  4 ,  19 ,  50 ,  48 };

Java

        System.out.println( "排序之前：\n"  + java.util.Arrays.toString(data));

Java

        bubbleSort(data);

Java

        System.out.println( "排序之后：\n"  + java.util.Arrays.toString(data));

Java

    }

Java

}

Java

### 5、快速排序（Quick Sort）

**算法描述：**

使用分治法来把一个串（list）分为两个子串（sub-lists）。具体算法描述如下：

* 从数列中挑出一个元素，称为 “基准”（pivot）；
* 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
* 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。



key值的选取可以有多种形式，例如中间数或者随机数，分别会对算法的复杂度产生不同的影响。

**代码实现：**

package  com.atguigu.interview.chapter02;

Java

/\*\*

Java

 \* @author atguigu

Java

 \* @since 2019/7/22

Java

 \* 快速排序

Java

 \*/

Java

public   class  QuickSort {

Java

     public   static   void  quickSort( int [] data,  int  low,  int  high) {

Java

         int  i, j, temp, t;

Java

         if  (low > high) {

Java

             return ;

Java

        }

Java

        i = low;

Java

        j = high;

Java

         //temp就是基准位

Java

        temp = data[low];

Java

        System.out.println( "基准位："  + temp);

Java

         while  (i < j) {

Java

             //先看右边，依次往左递减

Java

             while  (temp <= data[j] && i < j) {

Java

                j--;

Java

            }

Java

             //再看左边，依次往右递增

Java

             while  (temp >= data[i] && i < j) {

Java

                i++;

Java

            }

Java

             //如果满足条件则交换

Java

             if  (i < j) {

Java

                System.out.println( "交换："  + data[i] +  "和"  + data[j]);

Java

                t = data[j];

Java

                data[j] = data[i];

Java

                data[i] = t;

Java

                System.out.println(java.util.Arrays.toString(data));

Java

            }

Java

        }

Java

         //最后将基准位与i和j相等位置的数字交换

Java

        System.out.println( "基准位"  + temp +  "和i、j相遇的位置"  + data[i] +  "交换" );

Java

        data[low] = data[i];

Java

        data[i] = temp;

Java

        System.out.println(java.util.Arrays.toString(data));

Java

         //递归调用左半数组

Java

        quickSort(data, low, j -  1 );

Java

         //递归调用右半数组

Java

        quickSort(data, j +  1 , high);

Java

    }

Java

     public   static   void  main(String[] args) {

Java

         int [] data = { 3 ,  44 ,  38 ,  5 ,  47 ,  15 ,  36 ,  26 ,  27 ,  2 ,  46 ,  4 ,  19 ,  50 ,  48 };

Java

        System.out.println( "排序之前：\n"  + java.util.Arrays.toString(data));

Java

        quickSort(data,  0 , data.length -  1 );

Java

        System.out.println( "排序之后：\n"  + java.util.Arrays.toString(data));

Java

    }

Java

}

Java

### 6、二分查找（Binary Search）

**算法描述：**

* 二分查找也称折半查找，它是一种效率较高的查找方法，要求列表中的元素首先要进行有序排列。
* 首先，假设表中元素是按升序排列，将表中间位置记录的关键字与查找关键字比较，如果两者相等，则查找成功；
* 否则利用中间位置记录将表分成前、后两个子表，如果中间位置记录的关键字大于查找关键字，则进一步查找前一子表，否则进一步查找后一子表。
* 重复以上过程，直到找到满足条件的记录，使查找成功，或直到子表不存在为止，此时查找不成功。

**代码实现：**

package  com.atguigu.interview.chapter02;

Java

/\*\*

Java

 \* @author atguigu

Java

 \* @since 2019/7/22

Java

 \*/

Java

public   class  BinarySearch {

Java

     /\*\*

Java

     \* 二分查找 时间复杂度O(log2n);空间复杂度O(1)

Java

     \*

Java

     \* @param arr     被查找的数组

Java

     \* @param left

Java

     \* @param right

Java

     \* @param findVal

Java

     \* @return 返回元素的索引

Java

     \*/

Java

     public   static   int  binarySearch( int [] arr,  int  left,  int  right,  int  findVal) {

Java

         if  (left > right) { //递归退出条件，找不到，返回-1

Java

             return  - 1 ;

Java

        }

Java

         int  midIndex = (left + right) /  2 ;

Java

         if  (findVal < arr[midIndex]) { //向左递归查找

Java

             return  binarySearch(arr, left, midIndex, findVal);

Java

        }  else   if  (findVal > arr[midIndex]) { //向右递归查找

Java

             return  binarySearch(arr, midIndex, right, findVal);

Java

        }  else  {

Java

             return  midIndex;

Java

        }

Java

    }

Java

     public   static   void  main(String[] args){

Java

         //注意：需要对已排序的数组进行二分查找

Java

         int [] data = {- 49 , - 30 , - 16 ,  9 ,  21 ,  21 ,  23 ,  30 ,  30 };

Java

         int  i = binarySearch(data,  0 , data.length,  21 );

Java

        System.out.println(i);

Java

    }

Java

}

Java

**拓展需求：**

当一个有序数组中，有多个相同的数值时，如何将所有的数值都查找到。

**代码实现：**

package  com.atguigu.interview.chapter02;

Java

import  java.util.ArrayList;

Java

import  java.util.List;

Java

/\*\*

Java

 \* @author  atguigu

Java

 \* @since 2019/7/22

Java

 \*/

Java

public   class  BinarySearch2 {

Java

     /\*\*

Java

     \* {1, 8, 10, 89, 1000, 1000, 1234}

Java

     \* 一个有序数组中，有多个相同的数值，如何将所有的数值都查找到，比如这里的 1000.

Java

     \* 分析：

Java

     \* 1. 返回的结果是一个列表 list

Java

     \* 2. 在找到结果时，向左边扫描，向右边扫描 [条件]

Java

     \* 3. 找到结果后，就加入到ArrayBuffer

Java

     \*

Java

     \* @return

Java

     \*/

Java

     public   static  List<Integer> binarySearch2( int [] arr,  int  left,  int  right,  int  findVal) {

Java

         //找不到条件?

Java

        List<Integer> list =  new  ArrayList<>();

Java

         if  (left > right) { //递归退出条件，找不到，返回-1

Java

             return  list;

Java

        }

Java

         int  midIndex = (left + right) /  2 ;

Java

         int  midVal = arr[midIndex];

Java

         if  (findVal < midVal) { //向左递归查找

Java

             return  binarySearch2(arr, left, midIndex -  1 , findVal);

Java

        }  else   if  (findVal > midVal) {  //向右递归查找

Java

             return  binarySearch2(arr, midIndex +  1 , right, findVal);

Java

        }  else  {

Java

            System.out.println( "midIndex="  + midIndex);

Java

             //向左边扫描

Java

             int  temp = midIndex -  1 ;

Java

             while  ( true ) {

Java

                 if  (temp <  0  || arr[temp] != findVal) {

Java

                     break ;

Java

                }

Java

                 if  (arr[temp] == findVal) {

Java

                    list.add(temp);

Java

                }

Java

                temp -=  1 ;

Java

            }

Java

             //将中间这个索引加入

Java

            list.add(midIndex);

Java

             //向右边扫描

Java

            temp = midIndex +  1 ;

Java

             while  ( true ) {

Java

                 if  (temp > arr.length -  1  || arr[temp] != findVal) {

Java

                     break ;

Java

                }

Java

                 if  (arr[temp] == findVal) {

Java

                    list.add(temp);

Java

                }

Java

                temp +=  1 ;

Java

            }

Java

             return  list;

Java

        }

Java

    }

Java

     public   static   void  main(String[] args){

Java

         //注意：需要对已排序的数组进行二分查找

Java

         int [] data = { 1 ,  8 ,  10 ,  89 ,  1000 ,  1000 ,  1234 };

Java

        List<Integer> list = binarySearch2(data,  0 , data.length,  1000 );

Java

        System.out.println(list);

Java

    }

Java

}

Java

第十一章-设计模式

### 1、你所知道的设计模式有哪些

Java 中一般认为有 23 种设计模式，我们不需要所有的都会，但是其中常用的几种设计模式应该去掌握。下面列出了所有的设计模式。需要掌握的设计模式我单独列出来了，当然能掌握的越多越好。

总体来说设计模式分为三大类：

创建型模式，共5种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共7种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共11种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

### 2、单例模式（Binary Search）

#### 2.1 单例模式定义

单例模式确保某个类只有一个实例，而且自行实例化并向整个系统提供这个实例。在计算机系统中，线程池、缓存、日志对象、对话框、打印机、显卡的驱动程序对象常被设计成单例。这些应用都或多或少具有资源管理器的功能。每台计算机可以有若干个打印机，但只能有一个Printer Spooler，以避免两个打印作业同时输出到打印机中。每台计算机可以有若干通信端口，系统应当集中管理这些通信端口，以避免一个通信端口同时被两个请求同时调用。总之，选择单例模式就是为了避免不一致状态。

#### 2.2 单例模式的特点

* 单例类只能有一个实例。
* 单例类必须自己创建自己的唯一实例。
* 单例类必须给所有其他对象提供这一实例。

单例模式保证了全局对象的唯一性，比如系统启动读取配置文件就需要单例保证配置的一致性。

#### 2.3 单例的四大原则

* 构造私有
* 以静态方法或者枚举返回实例
* 确保实例只有一个，尤其是多线程环境
* 确保反序列换时不会重新构建对象

#### 2.4 实现单例模式的方式

##### 1）饿汉式（立即加载）：

饿汉式单例在类加载初始化时就创建好一个静态的对象供外部使用，除非系统重启，这个对象不会改变，所以本身就是线程安全的。

Singleton通过将构造方法限定为private避免了类在外部被实例化，在同一个虚拟机范围内，Singleton的唯一实例只能通过getInstance()方法访问。（事实上，通过Java反射机制是能够实例化构造方法为private的类的，会使Java单例实现失效）

package com.atguigu.interview.chapter02;

Java

/\*\*

Java

 \* @author atguigu

Java

 \*

Java

 \* 饿汉式（立即加载）

Java

 \*/

Java

public   class  Singleton {

Java

     /\*\*

Java

     \* 私有构造

Java

     \*/

Java

     private  Singleton() {

Java

        System.out.println( "构造函数Singleton1" );

Java

    }

Java

     /\*\*

Java

     \* 初始值为实例对象

Java

     \*/

Java

     private   static  Singleton single =  new  Singleton();

Java

Java

     /\*\*

Java

     \* 静态工厂方法

Java

     \* @return 单例对象

Java

     \*/

Java

     public   static  Singleton getInstance() {

Java

        System.out.println( "getInstance" );

Java

         return  single;

Java

    }

Java

     public   static   void  main(String[] args){

Java

        System.out.println( "初始化" );

Java

        Singleton instance = Singleton.getInstance();

Java

    }

Java

}

Java

##### 2）懒汉式（延迟加载）：

该示例虽然用延迟加载方式实现了懒汉式单例，但在多线程环境下会产生多个Singleton对象

package com.atguigu.interview.chapter02;

Java

/\*\*

Java

 \* @author atguigu

Java

 \*

Java

 \* 懒汉式（延迟加载）

Java

 \*/

Java

public   class  Singleton2 {

Java

     /\*\*

Java

     \* 私有构造

Java

     \*/

Java

     private  Singleton2() {

Java

        System.out.println( "构造函数Singleton2" );

Java

    }

Java

     /\*\*

Java

     \* 初始值为null

Java

     \*/

Java

     private   static  Singleton2 single = null;

Java

     /\*\*

Java

     \* 静态工厂方法

Java

     \* @return 单例对象

Java

     \*/

Java

     public   static  Singleton2 getInstance() {

Java

         if (single == null){

Java

            System.out.println( "getInstance" );

Java

            single =  new  Singleton2();

Java

        }

Java

         return  single;

Java

    }

Java

     public   static   void  main(String[] args){

Java

        System.out.println( "初始化" );

Java

        Singleton2 instance = Singleton2.getInstance();

Java

    }

Java

}

Java

##### 3）同步锁（解决线程安全问题）：

在方法上加synchronized同步锁或是用同步代码块对类加同步锁，此种方式虽然解决了多个实例对象问题，但是该方式运行效率却很低下，下一个线程想要获取对象，就必须等待上一个线程释放锁之后，才可以继续运行。

package com.atguigu.interview.chapter02;

Java

/\*\*

Java

 \* @author atguigu

Java

 \*

Java

 \* 同步锁（解决线程安全问题）

Java

 \*/

Java

public   class  Singleton3 {

Java

     /\*\*

Java

     \* 私有构造

Java

     \*/

Java

     private  Singleton3() {}

Java

     /\*\*

Java

     \* 初始值为null

Java

     \*/

Java

     Private static  Singleton3 single = null;

Java

     Public synchronized    static  Singleton3 getInstance() {

Java

             if (single == null){

Java

                single =  new  Singleton3();

Java

            }

Java

}

Java

         return  single;

Java

    }

Java

}

Java

##### 4）双重检查锁（提高同步锁的效率）：

使用双重检查锁进一步做了优化，可以避免整个方法被锁，只对需要锁的代码部分加锁，可以提高执行效率。

package com.atguigu.interview.chapter02;

Java

/\*\*

Java

 \* @author atguigu

Java

\* 双重检查锁（提高同步锁的效率）

Java

 \*/

Java

public   class  Singleton4 {

Java

     /\*\*

Java

     \* 私有构造

Java

     \*/

Java

     private  Singleton4() {}

Java

     /\*\*

Java

     \* 初始值为null

Java

\* 加volatile关键字是为了防止 创建对象时的指令重排问题，导致其他线程使用对象时造成空指针问题。

Java

     \*/

Java

     Private volatile   static  Singleton4 single = null;

Java

     /\*\*

Java

     \* 双重检查锁

Java

     \* @return 单例对象

Java

     \*/

Java

     public   static  Singleton4 getInstance() {

Java

         if  (single == null) {

Java

            synchronized (Singleton4. class ) {

Java

                 if  (single == null) {

Java

                    single =  new  Singleton4();

Java

                }

Java

            }

Java

        }

Java

         return  single;

Java

    }

Java

}

Java

##### 5） 静态内部类：

这种方式引入了一个内部静态类（static class），静态内部类只有在调用时才会加载，它保证了Singleton 实例的延迟初始化，又保证了实例的唯一性。它把singleton 的实例化操作放到一个静态内部类中，在第一次调用getInstance() 方法时，JVM才会去加载InnerObject类，同时初始化singleton 实例，所以能让getInstance() 方法线程安全。

特点是：即能延迟加载，也能保证线程安全。

静态内部类虽然保证了单例在多线程并发下的线程安全性，但是在遇到序列化对象时，默认的方式运行得到的结果就是多例的。

package com.atguigu.interview.chapter02;

Java

/\*\*

Java

 \* @author atguigu

Java

 \*

Java

 \* 静态内部类（延迟加载，线程安全）

Java

 \*/

Java

public   class  Singleton5 {

Java

     /\*\*

Java

     \* 私有构造

Java

     \*/

Java

     private  Singleton5() {}

Java

Java

     /\*\*

Java

     \* 静态内部类

Java

     \*/

Java

     private   static   class  InnerObject{

Java

         private   static  Singleton5 single =  new  Singleton5();

Java

    }

Java

     public   static  Singleton5 getInstance() {

Java

         return  InnerObject.single;

Java

    }

Java

}

Java

##### 6）内部枚举类实现（防止反射和反序列化攻击）：

事实上，通过Java反射机制是能够实例化构造方法为private的类的。这也就是我们现在需要引入的枚举单例模式。

package com.atguigu.interview.chapter02;

Java

/\*\*

Java

 \* @author atguigu

Java

 \*/

Java

public   class  SingletonFactory {

Java

     /\*\*

Java

     \* 内部枚举类

Java

     \*/

Java

     private   enum  EnumSingleton{

Java

        Singleton;

Java

         private  Singleton6 singleton;

Java

         //枚举类的构造方法在类加载是被实例化

Java

         private  EnumSingleton(){

Java

            singleton =  new  Singleton6();

Java

        }

Java

         public  Singleton6 getInstance(){

Java

             return  singleton;

Java

        }

Java

    }

Java

     public   static  Singleton6 getInstance() {

Java

         return  EnumSingleton.Singleton.getInstance();

Java

    }

Java

}

Java

class  Singleton6 {

Java

     public  Singleton6(){}

Java

}

Java

### 3、工厂设计模式（Factory）

#### 3.1 什么是工厂设计模式？

工厂设计模式，顾名思义，就是用来生产对象的，在java中，万物皆对象，这些对象都需要创建，如果创建的时候直接new该对象，就会对该对象耦合严重，假如我们要更换对象，所有new对象的地方都需要修改一遍，这显然违背了软件设计的开闭原则，如果我们使用工厂来生产对象，我们就只和工厂打交道就可以了，彻底和对象解耦，如果要更换对象，直接在工厂里更换该对象即可，达到了与对象解耦的目的；所以说，工厂模式最大的优点就是：解耦

#### 3.2 简单工厂（Simple Factory）

**定义：**

一个工厂方法，依据传入的参数，生成对应的产品对象；  
**角色：**  
1、抽象产品  
2、具体产品  
3、具体工厂  
4、产品使用者  
**使用说明：**

先将产品类抽象出来，比如，苹果和梨都属于水果，抽象出来一个水果类Fruit，苹果和梨就是具体的产品类，然后创建一个水果工厂，分别用来创建苹果和梨。代码如下：

**水果接口：**

public  interface Fruit {

Java

     void  whatIm();

Java

}

Java

**苹果类：**

public   class  Apple  implements  Fruit {

Java

     @Override

Java

     public   void  whatIm() {

Java

        System.out.println( "苹果" );

Java

    }

Java

}

Java

**梨类：**

public   class  Pear  implements  Fruit {

Java

    @Override

Java

     public   void  whatIm() {

Java

        System.out.println( "梨" );

Java

    }

Java

}

Java

**水果工厂：**

public   class  FruitFactory {

Java

     public  Fruit createFruit(String type) {

Java

         if  (type.equals( "apple" )) { //生产苹果

Java

             return   new  Apple();

Java

        }  else   if  (type.equals( "pear" )) { //生产梨

Java

             return   new  Pear();

Java

        }

Java

         return  null;

Java

    }

Java

}

Java

**使用工厂生产产品：**

public   class  FruitApp {

Java

     public   static   void  main(String[] args) {

Java

        FruitFactory mFactory =  new  FruitFactory();

Java

        Apple apple = (Apple) mFactory.createFruit( "apple" ); //获得苹果

Java

        Pear pear = (Pear) mFactory.createFruit( "pear" ); //获得梨

Java

        apple.whatIm();

Java

        pear.whatIm();

Java

    }

Java

}

Java

以上的这种方式，每当添加一种水果，就必然要修改工厂类，违反了开闭原则；

所以简单工厂只适合于产品对象较少，且产品固定的需求，对于产品变化无常的需求来说显然不合适。

#### 3.3 工厂方法（Factory Method）

**定义：**

将工厂提取成一个接口或抽象类，具体生产什么产品由子类决定；  
**角色：**  
1、抽象产品  
2、具体产品  
3、抽象工厂  
4、具体工厂  
**使用说明：**

和上例中一样，产品类抽象出来，这次我们把工厂类也抽象出来，生产什么样的产品由子类来决定。代码如下：  
**水果接口、苹果类和梨类：**

代码和上例一样

**抽象工厂接口：**

public  interface FruitFactory {

Java

    Fruit createFruit(); //生产水果

Java

}

Java

**苹果工厂：**

public   class  AppleFactory implements FruitFactory {

Java

    @Override

Java

     public  Apple createFruit() {

Java

         return   new  Apple();

Java

    }

Java

}

Java

**梨工厂：**

public   class  PearFactory implements FruitFactory {

Java

    @Override

Java

     public  Pear createFruit() {

Java

         return   new  Pear();

Java

    }

Java

}

Java

**使用工厂生产产品：**

public   class  FruitApp {

Java

     public   static   void  main(String[] args){

Java

        AppleFactory appleFactory =  new  AppleFactory();

Java

        PearFactory pearFactory =  new  PearFactory();

Java

        Apple apple = appleFactory.createFruit(); //获得苹果

Java

        Pear pear = pearFactory.createFruit(); //获得梨

Java

        apple.whatIm();

Java

        pear.whatIm();

Java

    }

Java

}

Java

以上这种方式，虽然解耦了，也遵循了开闭原则，但是如果我需要的产品很多的话，需要创建非常多的工厂，所以这种方式的缺点也很明显。

#### 3.4 抽象工厂（Abstract Factory）

**定义：**

为创建一组相关或者是相互依赖的对象提供的一个接口，而不需要指定它们的具体类。  
**角色：**

1. 抽象产品  
   2、具体产品  
   3、抽象工厂  
   4、具体工厂

**使用说明：**

抽象工厂和工厂方法的模式基本一样，区别在于，工厂方法是生产一个具体的产品，而抽象工厂可以用来生产一组相同，有相对关系的产品；重点在于一组，一批，一系列；举个例子，假如生产小米手机，小米手机有很多系列，小米note、红米note等；假如小米note生产需要的配件有825的处理器，6英寸屏幕，而红米只需要650的处理器和5寸的屏幕就可以了。用抽象工厂来实现：

**cpu接口和实现类：**

public  interface Cpu {

Java

     void  run();

Java

     class  Cpu650 implements Cpu {

Java

        @Override

Java

         public   void  run() {

Java

            System.out.println( "650 也厉害" );

Java

        }

Java

    }

Java

     class  Cpu825 implements Cpu {

Java

        @Override

Java

         public   void  run() {

Java

            System.out.println( "825 更强劲" );

Java

        }

Java

    }

Java

}

Java

**屏幕接口和实现类：**

public  interface Screen {

Java

     void  size();

Java

     class  Screen5 implements Screen {

Java

        @Override

Java

         public   void  size() {

Java

            System.out.println( ""  +   "5寸" );

Java

        }

Java

    }

Java

     class  Screen6 implements Screen {

Java

        @Override

Java

         public   void  size() {

Java

            System.out.println( "6寸" );

Java

        }

Java

    }

Java

}

Java

**抽象工厂接口：**

public  interface PhoneFactory {

Java

    Cpu getCpu(); //使用的cpu

Java

    Screen getScreen(); //使用的屏幕

Java

}

Java

**小米手机工厂：**

public   class  XiaoMiFactory implements PhoneFactory {

Java

    @Override

Java

     public  Cpu.Cpu825 getCpu() {

Java

         return   new  Cpu.Cpu825(); //高性能处理器

Java

    }

Java

    @Override

Java

     public  Screen.Screen6 getScreen() {

Java

         return   new  Screen.Screen6(); //6寸大屏

Java

    }

Java

}

Java

**红米手机工厂：**

public   class  HongMiFactory implements PhoneFactory {

Java

    @Override

Java

     public  Cpu.Cpu650 getCpu() {

Java

         return   new  Cpu.Cpu650(); //高效处理器

Java

    }

Java

    @Override

Java

     public  Screen.Screen5 getScreen() {

Java

         return   new  Screen.Screen5(); //小屏手机

Java

    }

Java

}

Java

**使用工厂生产产品：**

public   class  PhoneApp {

Java

     public   static   void  main(String[] args){

Java

        HongMiFactory hongMiFactory =  new  HongMiFactory();

Java

        XiaoMiFactory xiaoMiFactory =  new  XiaoMiFactory();

Java

        Cpu.Cpu650 cpu650 = hongMiFactory.getCpu();

Java

        Cpu.Cpu825 cpu825 = xiaoMiFactory.getCpu();

Java

        cpu650.run();

Java

        cpu825.run();

Java

        Screen.Screen5 screen5 = hongMiFactory.getScreen();

Java

        Screen.Screen6 screen6 = xiaoMiFactory.getScreen();

Java

        screen5.size();

Java

        screen6.size();

Java

    }

Java

}

Java

以上例子可以看出，抽象工厂可以解决一系列的产品生产的需求，对于大批量，多系列的产品，用抽象工厂可以更好的管理和扩展。

#### 3.5 三种工厂方式总结

1、对于简单工厂和工厂方法来说，两者的使用方式实际上是一样的，如果对于产品的分类和名称是确定的，数量是相对固定的，推荐使用简单工厂模式；

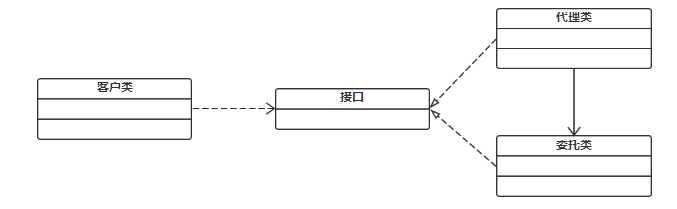
2、抽象工厂用来解决相对复杂的问题，适用于一系列、大批量的对象生产。

### 4、代理模式（Proxy）

#### 4.1 什么是代理模式？

代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。通俗的来讲代理模式就是我们生活中常见的中介。

举个例子来说明：假如说我现在想买一辆二手车，虽然我可以自己去找车源，做质量检测等一系列的车辆过户流程，但是这确实太浪费我得时间和精力了。我只是想买一辆车而已为什么我还要额外做这么多事呢？于是我就通过中介公司来买车，他们来给我找车源，帮我办理车辆过户流程，我只是负责选择自己喜欢的车，然后付钱就可以了。用图表示如下：



#### 4.2 为什么要用代理模式？

**中介隔离作用：**

在某些情况下，一个客户类不想或者不能直接引用一个委托对象，而代理类对象可以在客户类和委托对象之间起到中介的作用，其特征是代理类和委托类实现相同的接口。

**开闭原则，增加功能：**

代理类除了是客户类和委托类的中介之外，我们还可以通过给代理类增加额外的功能来扩展委托类的功能，这样做我们只需要修改代理类而不需要再修改委托类，符合代码设计的开闭原则。代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后对返回结果的处理等。代理类本身并不真正实现服务，而是同过调用委托类的相关方法，来提供特定的服务。真正的业务功能还是由委托类来实现，但是可以在业务功能执行的前后加入一些公共的服务。例如我们想给项目加入缓存、日志这些功能，我们就可以使用代理类来完成，而没必要修改已经封装好的委托类。

#### 4.3 有哪几种代理模式？

我们有多种不同的方式来实现代理。

如果按照代理创建的时期来进行分类的话，可以分为两种：静态代理、动态代理。

* 静态代理是由程序员创建或特定工具自动生成源代码，再对其编译。在程序员运行之前，代理类.class文件就已经被创建了。
* 动态代理是在程序运行时通过反射机制动态创建的。

#### 4.4 静态代理（Static Proxy）

**第一步：创建服务类接口**

public  interface BuyHouse {

Java

     void  buyHouse();

Java

}

Java

**第二步：实现服务接口**

public   class  BuyHouseImpl implements BuyHouse {

Java

    @Override

Java

     public   void  buyHouse() {

Java

        System.out.println( "我要买房" );

Java

    }

Java

}

Java

**第三步：创建代理类**

  public   class  BuyHouseProxy implements BuyHouse {

Java

     private  BuyHouse buyHouse;

Java

     public  BuyHouseProxy(final BuyHouse buyHouse) {

Java

         this .buyHouse = buyHouse;

Java

    }

Java

    @Override

Java

     public   void  buyHouse() {

Java

        System.out.println( "买房前准备" );

Java

        buyHouse.buyHouse();

Java

        System.out.println( "买房后装修" );

Java

    }

Java

}

Java

**第四步：编写测试类**

public   class  HouseApp {

Java

     public   static   void  main(String[] args) {

Java

        BuyHouse buyHouse =  new  BuyHouseImpl();

Java

        BuyHouseProxy buyHouseProxy =  new  BuyHouseProxy(buyHouse);

Java

        buyHouseProxy.buyHouse();

Java

    }

Java

}

Java

**静态代理总结：**

优点：可以做到在符合开闭原则的情况下对目标对象进行功能扩展。

缺点：我们得为每一个服务创建代理类，工作量太大，不易管理。同时接口一旦发生改变，代理类也得相应修改。

#### 4.5 JDK动态代理（Dynamic Proxy）

在动态代理中我们不再需要再手动的创建代理类，我们只需要编写一个动态处理器就可以了。真正的代理对象由JDK在运行时为我们动态的来创建。

**第一步：创建服务类接口**

代码和上例一样

**第二步：实现服务接口**

代码和上例一样

**第三步：编写动态处理器**

public   class  DynamicProxyHandler implements InvocationHandler {

Java

     private  Object object;

Java

     public  DynamicProxyHandler(final Object object) {

Java

         this .object = object;

Java

    }

Java

    @Override

Java

     public  Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

Java

        System.out.println( "买房前准备" );

Java

        Object result = method.invoke(object, args);

Java

        System.out.println( "买房后装修" );

Java

         return  result;

Java

    }

Java

}

Java

**第四步：编写测试类**

public   class  HouseApp {

Java

     public   static   void  main(String[] args) {

Java

        BuyHouse buyHouse =  new  BuyHouseImpl();

Java

        BuyHouse proxyBuyHouse = (BuyHouse) Proxy.newProxyInstance(

Java

                BuyHouse. class .getClassLoader(),

Java

                 new  Class[]{BuyHouse. class },

Java

                 new  DynamicProxyHandler(buyHouse));

Java

        proxyBuyHouse.buyHouse();

Java

    }

Java

}

Java

 Proxy是所有动态生成的代理的共同的父类，这个类有一个静态方法Proxy.newProxyInstance()，接收三个参数：

* ClassLoader loader：指定当前目标对象使用的类加载器,获取加载器的方法是固定的
* Class<?>[] interfaces：指定目标对象实现的接口的类型,使用泛型方式确认类型
* InvocationHandler：指定动态处理器，执行目标对象的方法时,会触发事件处理器的方法

**JDK动态代理总结：**

优点：相对于静态代理，动态代理大大减少了开发任务，同时减少了对业务接口的依赖，降低了耦合度。

缺点：Proxy是所有动态生成的代理的共同的父类，因此服务类必须是接口的形式，不能是普通类的形式，因为Java无法实现多继承。

#### 4.6 CGLib动态代理（CGLib Proxy）

JDK实现动态代理需要实现类通过接口定义业务方法，对于没有接口的类，如何实现动态代理呢，这就需要CGLib了。CGLib采用了底层的字节码技术，其原理是通过字节码技术为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。但因为采用的是继承，所以不能对final修饰的类进行代理。JDK动态代理与CGLib动态代理均是实现Spring AOP的基础。

**Cglib子类代理实现方法：**

（1）引入cglib的jar文件，asm的jar文件

（2）代理的类不能为final

（3）目标业务对象的方法如果为final/static，那么就不会被拦截，即不会执行目标对象额外的业务方法

**第一步：创建服务类**

public   class  BuyHouse2 {

Java

     public   void  buyHouse() {

Java

        System.out.println( "我要买房" );

Java

    }

Java

}

Java

**第二步：创建CGLIB代理类**

public   class  CglibProxy implements MethodInterceptor {

Java

     private  Object target;

Java

     public  CglibProxy(Object target) {

Java

         this .target = target;

Java

    }

Java

     /\*\*

Java

     \*  给目标对象创建一个代理对象

Java

     \* @return 代理对象

Java

     \*/

Java

     public  Object getProxyInstance() {

Java

         //1.工具类

Java

        Enhancer enhancer =  new  Enhancer();

Java

         //2.设置父类

Java

        enhancer.setSuperclass(target.getClass());

Java

         //3.设置回调函数

Java

        enhancer.setCallback( this );

Java

         //4.创建子类(代理对象)

Java

         return  enhancer.create();

Java

    }

Java

Java

     public  Object intercept(Object object, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {

Java

        System.out.println( "买房前准备" );

Java

         //执行目标对象的方法

Java

        Object result = method.invoke(target, args);

Java

        System.out.println( "买房后装修" );

Java

         return  result;

Java

    }

Java

}

Java

**第三步：创建测试类**

public   class  HouseApp {

Java

     public   static   void  main(String[] args) {

Java

        BuyHouse2 target =  new  BuyHouse2();

Java

        CglibProxy cglibProxy =  new  CglibProxy(target);

Java

        BuyHouse2 buyHouseCglibProxy = (BuyHouse2) cglibProxy.getProxyInstance();

Java

        buyHouseCglibProxy.buyHouse();

Java

    }

Java

}

Java

**CGLib代理总结：**

CGLib创建的动态代理对象比JDK创建的动态代理对象的性能更高，但是CGLIB创建代理对象时所花费的时间却比JDK多得多。所以对于单例的对象，因为无需频繁创建对象，用CGLIB合适，反之使用JDK方式要更为合适一些。同时由于CGLib由于是采用动态创建子类的方法，对于final修饰的方法无法进行代理。

#### 4.7 简述动态代理的原理， 常用的动态代理的实现方式

动态代理的原理: 使用一个代理将对象包装起来，然后用该代理对象取代原始对象。任何对原始对象的调用都要通过代理。

代理对象决定是否以及何时将方法调用转到原始对象上

动态代理的方式

基于接口实现动态代理： JDK动态代理

基于继承实现动态代理： Cglib、Javassist动态代理

第十二章 面试技巧篇

1、面试过程最关键的是什么？

1）不是你说了什么，而是你怎么说

2）大大方方的聊，放松，总结经验就是一点---淡定，越淡定越真实，回答的越好

2、面试时该怎么说？

1）语言表达清楚

（1）思维逻辑清晰，表达流畅

（2）一二三层次表达

2）所述内容不犯错

（1）不说前东家或者自己的坏话

（2）说自己擅长的方面，重点就是---遇到问题，往自己会的方向去引导，让面试官跟着你的思路走，不要让面试官随意问，如果面试官的问题是你擅长的点，一定要抓住机会，体现出你的技术深度。

（3）实质：内容听过，自我肯定；没听过，学习过程。

3、面试技巧

3.1 常见问题

1）你的优点是什么？

大胆的说出自己各个方面的优势和特长

2）你的缺点是什么？

不要谈为自己减分的问题；用“缺点”衬托自己的优点

3）你的离职原因是什么？

不说前东家坏话，哪怕被伤过

合情合理合法

不要说超过1个以上的原因

4）您对薪资的期望是多少？

非终面不深谈薪资

只说区间，不说具体数字

底线是不低于当前薪资

非要具体数字，区间取中间值，或者当前薪资的+20%

5）您还有什么想问的问题？

如果是技术面试官

Eg:项目进展 项目规模 项目选用技术 项目团队

如果是人事面试官

Eg:福利待遇等事关切身利益的问题

6）您最快多长时间能入职？

一周左右，如果公司需要，可以适当提前

7）为什么来北京（上海、深圳等某城市）

合理即可

Eg:有同学/朋友/家人 在上海 等 重要的不是问什么,而是当遇到此类问题自己是否能够自如应对

3.2 两个注意事项

1）职业化的语言

2）职业化的形象

3.3 自我介绍

1）个人基本信息

2）工作履历

时间、公司名称、任职岗位、主要工作内容、工作业绩、离职原因

3）深度沟通（也叫压力面试）

刨根问底下沉式追问（注意是下沉式，而不是发散式的）

基本技巧：往自己熟悉的方向说