- 1.java基础-java特性
  - 1-1：java特点
- 2.java基础-面对对象-综述
  - 2-1：面对对象三大特征(特点)
  - 2-2：面对对象的五大原则
  - 2-3：Java创建对象方式
  - 2-4：new abc是在堆里面呢?
  - 2-4：面向过程性能比面向对象高
- 3.java基础-面对对象-多态
  - 3-1：多态的必要条件（实现方式）
  - 3-2：多态机制
  - 3-3：多态的好处
  - 3-4：多态的例子
- 4.java基础-面对对象-重载与重写
  - 4-1：重载与重写
  - 4-2：Java 中是否可以覆盖(override)一个 private 或者是 static 的方法?
- 5.java基础-面对对象-接口与抽象类
  - 5-1：接口和抽象类的区别是什么?
  - 5-2：Java 抽象类可以有构造函数吗？作用是什么
  - 5-3: Java 抽象类可以实现接口吗？ 它们需要实现所有的方法吗?
  - 5-4：Java 抽象类可以是 final 的吗?
  - 5-5：Java 抽象类可以有 static 方法吗?
  - 5-6：可以创建抽象类的实例吗?
  - 5-7：抽象类必须有抽象方法吗?
  - 5-8：何时选用抽象类而不是接口?
  - 5-9：Java中的抽象方法是什么?
  - 5-10：Java抽象类中可以包含main方法吗?
  - 5-11：创建一个对象用什么运算符?对象实体与对象引用有何不同?
  - 5-12：interface实现方法
- 6.java基础-值传递与引用传递
- 7.java基础-深拷贝与浅拷贝
  - 7-1：深拷贝与浅拷贝
  - 7-2：浅拷贝方法
  - 7-3：深拷贝方法
- 8.java基础-面对对象-构造器
  - 8-1：一个类的构造方法的作用是什么? 若一个类没有声明构造方法，该程序能正确执行吗? 为什么?
  - 8-2：构造方法有哪些特性?
- 9.java基础-面对对象-静态与非静态
  - 9-1：静态方法和实例方法有何不同
  - 9-2：静态变量和实例变量的区别?
- 10.java基础-运算符-相等问题
  - 10-1： == 与 equals
  - 10-2：为什么要重写hashcode与equals
  - 10-3：重写equals不重写hashcode会出现什么问题
  - 10-4：为什么两个对象有相同的hashcode值，它们也不一定是相等的?

    - 10-5：hashcode和equals源码写一下
    - 10-5：说说&和&&的区别。
    - 10-6：用最有效率的方法算出 2 乘以 8 等于几?
    - 10-7：i++和++i的区别，及其线程安全问题
- 11.java基础-数据类型-8种常见类型
    - 11-1：八种数据类型是什么？-中兴
    - 11-2：数据类型的范围
    - 11-3：为什么byte类型是-128~+127
    - 11-4：java为什么除了基本数据类型还要有引用数据类型
    - 11-5：String为什么不是基本数据类型-字节
- 12.java基础-数据类型-自动拆装箱
    - 12-1：为什么要有自动拆装箱
    - 12-2：Integer缓存机制
    - 12-3：自动拆装箱的原理
    - 12-4：自动拆装箱使用场景
    - 12-5：自动拆装箱带来的问题
    - 12-6：String转出int型， 判断能不能转？ 如何转?
    - 12-7：short s1 = 1; s1 = s1 + 1;有什么错? short s1 = 1; s1+=1;有什么错?
    - 12-8：int与Integer区别
- 13.java基础-关键字-访问权限关键字
    - 13-1：访问控制关键字级别
    - 13-2: 通过反射访问private成员和方法，既然能访问为什么要private?
- 14.java基础-关键字-static
    - 14-1：static使用场景
- 15.java基础-关键字-final关键字
    - 15-1：final关键字使用特点
    - 15-2：final, finally, finalize 的区别。
    - 15-3：使用 final 关键字修饰一个变量时，是引用不能变，还是引用的对象不能变?
- 16.java基础-关键字-this关键字和super关键字
- 17.java基础-关键字-transient
- 18.java基础-集合map-hashmap的数据结构
    - 18-1：hashmap的数据结构
    - 18-1：扩容死循环问题
    - 18-2；为什么JDK8时候引入了红黑树?
    - 18-3：为什么不把链表全部换为红黑树
    - 18-4：为什么是使用红黑树而不是AVL树?
    - 18-5：为什么在JDK1.7的时候是先进行扩容后进行插入，而在JDK1.8的时候则是先插入后进行扩容的呢?
    - 18-6：为什么在JDK1.8中进行对HashMap优化的时候，把链表转化为红黑树的阈值是8,而不是7或者不是20呢（面试蘑菇街问过）
    - 18-7：扰动函数以及作用
    - 18-8：哈希冲突的解决方法
    - 18-9：HashMap的put操作
    - 18-10：HashMap的get操作
    - 18-11：hashmap的get和put操作的时间复杂度
    - 18-12：hashmap的String类型如何计算hashcode的

    - 18-13：reHash过程
    - 18-14：hash函数以及常用方法
    - 18-15：HashMap为什么要树化?
    - 18-16：hashmap树化门槛及作用
    - 18-17：hashmap的特性
    - 18-18：HashMap为什么可以插入空值?
    - 18-19：JDK的hashmap与Redis的hashmap的区别
    - 18-20：为什么要两次hash
- 19.java基础-集合map-hashmap源码数值分析
    - 19-1：HashMap中(tab.length - 1) & hash作用
    - 19-2：请解释一下HashMap的参数loadFactor，它的作用是什么?
    - 19-3：HashMap的扩容因子为什么是0.75
    - 19-4：为什么默认初始化桶数组大小为16
    - 19-5：hashmap为什么是2的次幂
- 20.java基础-集合map-hashmap线程问题
    - 20-1：hashMap是否线程安全
    - 20-2：为什么hashmap中String、integer包装类适合作为key
    - 20-3：线程安全的Map
    - 20-4：设计线程安全的map
- 21.java基础-集合map-ConcurrentHashMap
    - 21-1：ConcurrentHashMap的底层实现
    - 21-2：为何会出现ConcurrenHashMap?
    - 21-3：为什么ConcurrentHashMap（hashtable）为何不支持null键和null值
    - 21-4：ConcurrentHashMap的put操作
    - 21-5：分段锁原理
    - 21-6：hashmap与ConcurrentHashMap中put的区别
    - 21-7：扩容机制
    - 21-8：什么时候会发生扩容机制
- 22.java基础-集合map-TreeMap
    - 22-1：TreeMap底层原理：
    - 22-2：使用场景
- 23.java基础-集合map-LinkedHashmap
    - 23-1：linkedhashmap的底层原理
- 24.java基础-集合map-HashTable
    - 24-1：HashTable的底层原理
- 25.java基础-集合list-ArrayList
    - 25-1：数组(Array)和列表(ArrayList)有什么区别？ 什么时候应该使用 Array 而不是ArrayList?
    - 25-2：扩容机制
    - 25-3：ArrayList的add操作
    - 25-4：Arraylist初始大小以及扩容大小
    - 25-5：那如何解决ArrayList线程不安全问题呢?
- 26.java基础-集合list-vector
    - 26-1：Vector是保证线程安全的
- 27.java基础-集合list-linkedlist
- 28.java基础-集合set-HashSet
    - 28-1：hashset原理

    - 28-2：hashSet的内存泄漏
    - 28-3：为什么HashSet不安全
    - 28-4：如何保证线程安全
- 29.java基础-集合set-TreeSet
    - 29-1：TreeSet原理
- 30.java基础-集合set-LinkedSet
    - 30-1：LinkedSet原理
- 31.java基础-集合-集合大比较（区别和使用场景）
    - 31-1：set和list、 map的区别
    - 31-2：arraylist、linkedlist区别和适用场景
    - 31-3：vector、Arraylist区别和适用场景
    - 31-4：HashMap、Treemap、linkedHashMap区别和适用场景
    - 31-5：HashTable、Hashmap区别和适用场景
    - 31-6： ConcurrentHashMap、Hashmap区别和适用场景
    - 31-7： Hashset、Hashmap区别和适用场景
    - 31-8：treeset、hashset区别和适用场景
    - 31-9：JAVA集合类
    - 31-10：并发集合
    - 31-11：并发集合出现的原因
    - 31-11：collection与collections的区别
    - 31-12：Collections有哪些静态方法
    - 31-13：Comparable和Comparator区别
- 32.java基础-设计类问题
    - 32-1：如果想要一个key对应多个Value的话，怎么设计Map
    - 32-2：插入一万个元素之后会不会扩容，扩容扩多少
    - 32-3：创建一个对象HashMap<Integer,Integer> map=new HashMap<>先put(10),然后get(new Long(10))结果是多少?
    - 32-4：两个线程同时操作一个集合，一个线程读，一个线程写。有可能会产生并发问题吗?
- 33.java基础-IO-各种流
    - 33-1：为何还要有字符流
    - 33-2：字节流和字符流区别
    - 33-3：同步、异步与阻塞、非阻塞
    - 33-4：BIO\NIO\AIO区别
    - 33-5：linux的5种IO模型
    - 33-6：IO多路复用
    - 33-7：三种常用的实现方式-select
    - 33-8：三种常用的实现方式-poll
    - 33-9：三种常用的实现方式-epoll
    - 33-10：三种常用的实现方式区别
- 34.java基础-反射
    - 34-1：什么是反射
    - 34-2：反射会导致性能问题呢?
    - 34-3：如果避免反射导致的性能问题?
    - 34-2：获取class对象方法
    - 34-3：Class.forName和classloader.loadClass的区别
    - 34-4：哪些类不能反射

    - 34-5：反射优缺点
    - 34-6：反射的应用场景
- 35.java基础-注解
    - 35-1：元注解以及分类
    - 35-2：Java常用注解
- 36.java基础-泛型
    - 36-1：什么是泛型
    - 36-2：编译器如何处理泛型
    - 36-3：为什么Java要用这种编译器
    - 36-4: 什么是类型擦除
    - 36-5：类型擦除过程
    - 36-6：泛型带来的问题
    - 36-7：List泛型和原始类型List之间的区别?
    - 36-8：List泛型和原始类型List泛型之间的区别?
    - 36-9:子类继承父类的public可以写成private吗
    - 36-10：多态时是否会出现类型擦除
- 37-java基础-异常
    - 37-1：异常的分类
    - 37-2：Java 中的两种异常类型是什么？ 他们有什么区别?
    - 37-3：异常类型
    - 37-4：什么是OOM？常见有哪些OOM?
    - 37-3：异常链
- 38.java基础-常用类-String
    - 38-1：String为什么是final的?
    - 38-2：拼接方式
    - 38-3: String、StringBuffer和StringBuilder区别
    - 38-4：StringBuffer如何实现线程安全
    - 38-5：String 和 char[] 数组谁更适合存密码
    - 38-6：String str = new String("abc");创建了几个对象-百度，京东
    - 38-7：处理数据量较大的字符串用string还是stringbuilder，为什么
    - 38-8：为什么StringBuffer和StringBuilder比String更快（不变性）
    - 38-9：如何把一段逗号分割的字符串转换成一个数组?
    - 38-10：String的内部属性
    - 38-11：String的常用方法
- 39.java基础-常用类-枚举
    - 39-1：enum线程安全
    - 39-2: switch 是否可用于String类型的判断，Java哪个版本之后有此功能的
- 40.Java基础-常用类-时间类
    - 40-1：SimpDateFormat是线程不安全的类，不要定义为static变量，如果定义，必须加锁或工具类
- 41.java基础-常用类-Object类
    - 41-1：Object类有哪些方法
    - 41-2：为什么操作线程方法会在Object对象中
- 42.java基础-序列化
    - 42-1：什么是序列化和饭序列化
    - 42-2：序列化的作用
    - 42-3：java对象如何实现序列化

    - 42-4：java对象如何实现反序列化
    - 42-5：哪些不会被序列化
    - 42-6：序列化协议有哪些
    - 42-7：该接口并没有方法和字段，为什么只有实现了该接口的类的对象才能被序列化呢?
    - 42-8: 什么是serialVersionUID
- ----------
- 1.线程与进程
    - 1-1：什么是进程
    - 1-2：何为线程?
    - 1-3：线程与进程的区别
    - 1-4：进程的通信方式
    - 1-5：进程调度6任务过程
    - 1-5：进程的调度算法
    - 1-5：并发级别
    - 1-6：happen-before原则是什么
    - 单cpu上多线程效率和单线程比如何
- 2.什么是上下文切换?
- 3.线程死锁
    - 3-1：什么是线程死锁
    - 3-2：产生死锁的条件
    - 3-3：如何解决线程死锁问题
- 4.synchronized关键字
    - 4-1：synchronized关键字理解
    - 4-2：JDK1.6优化有哪些?
    - 4-3：底层原理
    - 4-4：谈谈 synchronized和ReentrantLock 的区别
    - 4-5：Lock和synchronized的区别
    - 4-6：synchronized的优势
    - 4-7：synchronized锁的膨胀过程（升级过程）
    - 4-8：那如何判断共享数据不会被线程竞争?
- 5.volatile关键字
    - 5-1：为什么要是用volatile关键字
    - 5-2：为什么其他线程能感知到变量更新
    - 5-3：volatile为什么不保证原子性吗?
    - 5-4：怎么保证输出结果是20000呢?
    - 5-5：为什么要重排
    - 5-6： 有哪几种重排
    - 5-7：举例说一下指令重排
    - 5-8： volatile怎么实现禁止指令重排?
    - 5-9：volatile都不保证原子性，为啥我们还要用它?
    - 5-10：synchronized 关键字和 volatile 关键字的区别
- 6.并发基础
    - 6-1：并发特性
    - 6-3：快速失败与安全失败
    - 6-4：说说并发与并行的区别?
    - 6-5：为什么要使用多线程呢?

    - 6-6：使用多线程可能带来什么问题?
    - 6-7：线程的生命周期和状态
    - 6-8：一般线程和守护线程的区别
    - 6-9：多线程公共用一个数据注意什么
    - 6-10：如何确保 N 个线程可以访问 N 个资源同时又不导致死锁?
- 7.创建线程方式
    - 7-2：创建线程的对比
- 8.线程基本操作与线程协作
    - 8-1：说说 sleep() 方法和 wait() 方法区别和共同点?
    - 8-2：yield join notify notifyAll
    - 8-3：为什么我们调用 start() 方法时会执行 run() 方法，为什么我们不能直接调用run() 方法?
    - 8-4：中断线程方法
    - 8-5：
- 9.线程同步
    - 9-2：线程间同步
- 10.线程池
    - 10-1：ThreadLocal内存泄露问题
    - 10-2：使用线程池的好处
    - 10-3：常规实现线程池方法
    - 10-4：线程池增长策略
    - 10-5：线程池拒绝策略
    - 10-6：BlockingQueue
    - 10-7：实现Runnable接口和Callable接口的区别
    - 10-8：实现 Runnable 接口比继承 Thread 类所具有的优势
    - 10-9：执行execute()方法和submit()方法的区别是什么呢?
- 11.Atomic原子类
    - 11-1：什么是原子类
    - 11-2：原子类的作用?
    - 11-3：i++自增操作不是原子性的，如何决绝原子性问题
    - 11-4：CAS
    - 11-5：CAS的ABA问题
    - 11-6：基本数据类型原子类的优势
- 12.AQS
    - 12-1：对AQS原理分析
    - 12-2：AQS 对资源的共享方式
    - 12-3：AQS 组件
- 13.锁
    - 13-1：锁
    - 13-2：乐观锁与悲观锁
    - 13-3：两种锁的使用场景
    - 13-4：乐观锁常见的两种实现方式
    - 13-5：乐观锁的缺点
    - 15-6：自旋锁
    - 13-7：自旋锁的优缺点
    - 13-8：自旋锁的升级——自适应自旋
    - 13-9：自旋锁使用场景

    - 13-10：可重入锁（递归锁）
    - 13-11：可重入锁使用场景
    - 13-12：可重入锁如果加了两把，但是只释放了一把会出现什么问题?
    - 13-13：如果只加了一把锁，释放两次会出现什么问题?
    - 13-14：读写锁
    - 13-15：公平锁
    - 13-16：非公平锁
    - 13-17：公平锁与非公平锁优缺点
    - 13-18：公平锁与非公平锁使用场景
    - 13-19：共享锁
    - 13-20：共享锁使用场景
    - 13-21：独占锁
    - 13-22：独占锁使用场景
    - 13-23：重量级锁
    - 13-24：重量级锁使用场景
    - 13-25：轻量级锁
    - 13-26：轻量级锁优缺点
    - 13-27：偏向锁
    - 13-28：偏向锁优缺点
    - 13-29：分段锁
    - 13-30：互斥锁
    - 13-31：同步锁
    - 13-32：死锁
    - 13-33：锁粗化
    - 13-34：锁消除
- 15.并发容器
    - 15-1：JDK 提供的并发容器总结
    - 15-2：CopyOnWriteArrayList 是如何做到的?
- ----------
- 1.类加载
    - 1.1：类的生命周期
    - 1.2：类的加载过程
    - 1.3：类加载机制
    - 1.4：知道哪些类加载器?
- 2.双亲委派模型
    - 2-1：双亲委派模型流程
    - 2-2：双亲委派模型带来了什么好处呢?
    - 2-3：如果我们不想用双亲委派模型怎么办?
    - 2-4：自己写一个类能不能被加载?
- 3.垃圾回收
    - 3-1：如何判断对象已经死亡?
    - 3-2：不可达的对象是否非死不可
    - 3-2：强、软、弱、虚引用
    - 3-3：如何减少 GC 的次数
    - 3-4：GC 是什么?为什么要有 GC?
    - 3-5：垃圾回收的优点

    - 3-6：垃圾回收器的基本原理是什么?
    - 3-7：什么样的对象需要回收
    - 3-8：可作为GC Roots的对象?
    - 3-9：垃圾回收器可以马上回收内存吗?
    - 3-10：有什么办法主动通知虚拟机进行垃圾回收?
    - 3-11：如何判断一个类是无用的类
    - 3-12：如何判断一个常量是废弃常量?
    - 3-11：垃圾回收算法
    - 3-12：Minor Gc和Full GC 有什么不同呢?
    - 3-13：何时发生full gc
- 4.常见的垃圾回收器有那些?
    - 4-1：Serial收集器
    - 4-2：ParNew收集器
    - 4-3：Parallel Scavenge收集器
    - 4-4：Serial Old收集器
    - 4-5：Parallel Old收集器
    - 4-6：CMS收集器
    - 4-7：G1收集器
    - 4-8：吞吐有限和响应有限的垃圾收集器如何选择
- 5.Java内存结构（JMM）
    - 5-0：简述（理解即可）
    - 5-1：为什么需要Java内存模型?
    - 5-2：什么是JMM
    - 5-3：什么是Java内存模型?
    - 5-4：Java内存模型的两大内存是啥?
    - 5-5：内存如何工作
    - 5-6：Java内存模型三大特性
    - 5-7：jvm内存结构
    - 5-8：程序计数器为什么是私有的?
    - 5-9：虚拟机栈和本地方法栈为什么是私有的?
    - 5-10：堆和栈的区别是什么?
    - 5-11：Java中的数组是存储在堆上还是栈上的?
    - 5-12：Java 8的metaspace (元空间)
    - 5-13：为什么要进行元空间代替持久代呢?
    - 5-14：Java中的对象一定在堆上分配内存吗?
    - 5-15：怎么如何获取堆和栈的dump文件?
    - 5-16：不同的虚拟机在实现运行时内存的时候有什么区别?
- 6.新生代， 老年代， 持久代
    - 6-1：新生代， 老年代， 持久代? 各存了什么?
- 7. HotSpot虚拟机对象
    - 7-1：说一下Java对象的创建过程
    - 7-2：内存分配的两种方式选择
    - 7-3：虚拟机如何保证线程安全
    - 7-4：对象的访问定位有哪两种方式?
    - 7-5：访问定位两种方式的优缺点
    - 7-6：对象分配规则

- 8.内存泄露与内存溢出
  - 8-1：什么是内存泄漏
  - 8-2：什么是内存溢出
  - 8-3：内存溢出， 内存泄漏区别?
  - 8-4：如何避免内存泄露、 溢出?
  - 8-5：如何检测内存泄露?
  - 8-6：java中会存在内存泄露呢？什么时候发生
- 9.调优工具
  - 9-1：调优工具有哪些?
- 10.JVM进程有哪些线程启动? (拼多多)
- 11.jvm启动模式之client 与server
- 12.简述JVM中静态分派和动态分派(引申:重载和重写)。
- ----------
- 1.计算机网络-HTTP-get与post
  - 1-1：get与post的区别
- 2.计算机网络-HTTP-报文结构与状态码
  - 2-1：状态码
  - 2-2：HTTP请求组成
- 3.计算机网络-HTTP-HTTP各个型号
  - 3-1：HTTP1.0优缺点
  - 3-2：HTTP/1.1相对于HTTP1.0改善
  - 3-3：HTTP1.1缺点
  - 3-4：HTTP长连接,短连接(也是TCP连接,短连接)
  - 3-5：HTTP/2 做了什么优化?
  - 3-6：HTTP/2有哪些缺陷?
  - 3-7：HTTP/3做了哪些优化?
- 4.计算机网络-HTTP-HTTPs
  - 4-1：HTTP与HTTPS区别
  - 4-2：HTTPS 解决了 HTTP 的哪些问题?
  - 4-3：HTTPS 是如何建立连接的？其间交互了什么?
  - 4-4：SSL/TLS握手
  - 4-5：HTTPS的加密过程
  - 4-6：加密
- 5.计算机网络-HTTP-Cookie与Session
  - 5-1：Cookie 和 Session 的区别
  - 5-2：Cookie作用
  - 5-3：Session用户登录状态过程
  - 5-4：token的验证流程
  - 5-5：token和cookie实现的区别
  - 5-6：HTTP是不保存状态的协议,如何保存用户状态?
  - 5-7：如何保存session
  - 5-8：如何实现 Session 跟踪呢?
  - 5-9：Cookie 被禁用怎么办?
  - 5-10：URI和URL的区别是什么?
- 6.计算机网络-综合应用-输入网址
  - 6-1：输入网址过程

    - 6-2：为什么域名要分级设计
    - 6-3：重定向原因
- 7.各层协议
    - 7-1：OSI与TCP/IP各层的结构与功能,都有哪些协议?
    - 7-2：网络层与数据链路层有什么关系呢?
- 8.TCP的三次握手
    - 8-1：TCP三次握手流程
    - 8-2：TCP为什么要三次握手
    - 8-3：TCP为什么SYN
    - 8-4：TCP除了SYN，为什么还要 ACK
    - 8-5：什么是 SYN 攻击？如何避免 SYN 攻击?
    - 8-5：如何对三次握手进行性能优化
    - 8-6：如何绕过三次握手发送数据
    - 8-7：TCP Fast Open的过程
    - 8-8：为什么需要 TCP 协议?
    - 8-9：什么是 TCP 连接?
    - 8-10：如何唯一确定一个 TCP 连接呢?
    - 8-11：有一个 IP 的服务器监听了一个端口，它的 TCP 的最大连接数是多少?
    - 8-12：服务端最大并发 TCP 连接数远不能达到理论上限
    - 8-15：为什么客户端和服务端的初始序列号 ISN 是不相同的?
    - 8-16：什么是Mss
    - 8-17：既然 IP 层会分片，为什么 TCP 层还需要 MSS 呢?
- 9.四次挥手
    - 9-1：TCP四次挥手流程
    - 9-2：TCP为什么要四次挥手
    - 9-3：如何对四次挥手进行优化
    - 9-4：为什么TIME_WAIT 等待的时间是 2MSL?
    - 9-5：为什么需要TIME_WAIT状态？（已经主动关闭连接了为啥还要保持资源一段时间呢？）
    - 9-6：TIME_WAIT 过多有什么危害?
    - 9-7：如何优化 TIME_WAIT?
    - 9-8：如果已经建立了连接，但是客户端突然出现故障了怎么办?
- 10.TCP传输数据优化方案
    - 10-1：TCP传输数据优化
    - 10-2： TCP 应该如何 Socket 编程?
- 11.TCP与UDP
    - 11-1：TCP与UDP区别
    - 11-2：TCP 和 UDP 应用场景
    - 11-3：TCP的作用
    - 11-4：TCP 数据包的大小
    - 11-5：TCP 数据包的编号（SEQ）
    - 11-5：TCP 数据包的组装
    - 11-6：什么是TCP粘包？怎么解决这个问题
    - 11-7：粘包的原因
    - 11-8：粘包解决方案
    - 11-9：UDP会不会产生粘包问题呢?
    - 11-2：TCP协议如何保证可靠传输方式

    - 11-3：UDP如何做可靠传输
- 12.重传机制
    - 12-1：常见的重传机制
    - 12-2：超时重传
    - 12-3：什么时候会发生超时重传
    - 12-4：超时重传存在的问题
    - 12-5：快速重传
    - 12-6：快速重传的问题
    - 12-7：SACK方法
    - 12-8：D-SACK
    - 12-9：D-SACK好处
- 13.滑动窗口与流量控制
    - 13-1：引入窗口概念的原因
    - 13-2：什么是窗口
    - 13-3：窗口大小由哪一方决定?
    - 13-4：发送方的窗口
    - 13-5：流量控制
    - 13-6：流量控制的过程
    - 13-7：
    - 13-8：
    - 13-9：TCP 是如何解决窗口关闭时，潜在的死锁现象呢?
- 14.拥塞控制
    - 14-1：为什么要有拥塞控制呀，不是有流量控制了吗?
    - 14-2：什么是拥塞控制
    - 14-3：什么是拥塞窗口？和发送窗口有什么关系呢?
    - 14-4：那么怎么知道当前网络是否出现了拥塞呢?
    - 14-5：拥塞控制算法
    - 14-6：那慢启动涨到什么时候是个头呢?
    - 14-7：重传机制何时结束
- 15.ARQ协议
    - 15-1：什么是ARQ协议
    - 15-2：什么是停止等待ARQ协议
    - 15-3: 什么是连续ARQ协议
- ----------
- 1.索引
    - 1-1：索引是什么
    - 1-2：为什么要用索引（优点）
    - 1-3：索引这么多优点，为什么不对表中的每一个列创建一个索引呢？（缺点）
    - 1-4：索引的主要原理，常用算法
    - 1-5：创建索引原则（使用场景）：
    - 1-6：创建索引的注意事项
    - 1-7：为什么索引能够提高查询速度
    - 1-8：创建索引的三种方式
    - 1-9：最左前缀原则
    - 1-10：最左匹配原则
- 2.索引的分类

    - 2-1：索引的分类
    - 2-2：各种索引定义
- 3.索引的结构
    - 3-1：mysql索引的结构
    - 3-2：B+树比B树的优势
    - 3-3：B+树与红黑树比较
    - 3-4：B+树与hash索引比较
    - 3-5：聚簇索引与非聚簇索引概念
    - 3-6：聚簇索引的优缺点
    - 3-7：非聚簇索引的优缺点
- 4.数据库优化
    - 4-1：为什么要优化
    - 4-2：索引优化
    - 4-3：查询优化
    - 4-4：当MySQL单表记录数过大时，数据库的CRUD性能会明显下降，如何解决
    - 4-5：垂直分表
    - 4-6：水平分表
    - 4-7：为什么要分库分表?
    - 4-8：分库分表，id如何处理
    - 4-9：为什么 MySQL 不推荐使用 uuid 或者雪花 id 作为主键?
    - 4-10：使用自增id的缺点
    - 4-11：SQL的生命周期
    - 4-12：mysql分页，超大分页怎么处理?
    - 4-13：慢查询优化
    - 4-14：为什么要尽量设定一个主键?
    - 4-15：主键使用自增ID还是UUID?
    - 4-16：字段为什么要求定义为not null?
- 5.事务
    - 5-1：什么是事务
    - 5-2：数据库事务特性
    - 5-3：为什么保证原子性和持久化
    - 5-3-1：什么是undo log
    - 5-4：四大隔离级别
    - 5-5：隔离级别的原理
    - 5-6：MySQL 中RC和RR隔离级别的区别
    - 5-7：隔离级别用来做什么
    - 5-8：并发事务带来什么问题
    - 5-9：数据库崩溃时事务的恢复机制
    - 5-10：Undo Log缺陷如何解决?
- 6.数据库基础知识
    - 6-1：为什么要使用数据库
    - 6-2: 什么是SQL?
    - 6-3：什么是MySQL?
    - 6-4：数据库三大范式是什么
    - 6-5：mysql有关权限的表都有哪几个
    - 6-6：什么是binlog

    - 6-6：binlog长什么样?
    - 6-6：binlog一般用来做什么
    - 6-8：redo log
    - 6-9：bin log和redo log
    - 6-10：我写其中的某一个log，失败了，那会怎么办?
    - 6-11：MySQL如何保证redo log和binlog的数据是一致的
    - 6-10：如果整个数据库的数据都被删除了，那我可以用redo log的记录来恢复吗?
    - 6-6：MySQL的binlog有有几种录入格式？分别有什么区别?
    - 6-7：mysql的数据类型
    - 6-8：
- 7.mysql引擎
    - 7-1：MySQL存储引擎MyISAM与InnoDB区别
    - 7-2：InnoDB引擎的4大特性
    - 7-3：InnoDB存储引擎的锁的算法
    - 7-4：MyISAM和InnoDB存储引擎使用的锁
    - 7-5：表级锁和行级锁对比：
    - 7-6：Innodb如何加锁
    - 7-7：Innodb解决死锁
- 8.锁
    - 8-1：隔离级别与锁的关系
    - 8-2：封锁粒度
    - 8-3：锁的类型
    - 8-4：三级封锁协议
    - 8-6：排它锁
    - 8-7：排他锁和共享锁的兼容性
- 9.阻塞
    - 9-1：什么是阻塞
    - 9-2：阻塞的优化
- 10.什么是慢查询
- 11.应用
    - 11.1：一条SQL语句执行得很慢的原因有哪些
    - 11.2：为什么数据库会选错了索引
- 12.常用sql的语句
    - 12-1：SQL语句主要分为哪几类
    - 12-2：超键、候选键、主键、外键分别是什么?
    - 12-3：为什么用自增列作为主键
    - 12-4：truncate、 delete区别
    - 12-5：什么是临时表，临时表什么时候删除?
    - 12-6：非关系型数据库和关系型数据库区别，优势比较?
    - 12-7：什么是 内连接、外连接、交叉连接、笛卡尔积等?
    - 12-8：varchar与char的区别
    - 12-9：count(*)、count(1)、count(column)的区别
    - 12-10：SQL 约束有哪几种?
    - 12-11：六种关联查询
    - 12-12：什么是子查询
    - 12-13：子查询的三种情况

    - 12-14：mysql中 in 和 exists 区别
    - 12-15：varchar(50)中50的涵义
    - 12-16：int(20)中20的涵义
    - 12-17：FLOAT和DOUBLE的区别是什么?
    - 12-18：UNION与UNION ALL的区别?
    - 12-19：数据库热备份与冷备份
- 13.主从复制
    - 13-1：什么是主从复制：
    - 13-2：主从复制的作用（好处，或者说为什么要做主从）：
    - 13-3：主从复制的原理（重中之重，面试必问）
    - 13-4：主从复制的几种方式
- 14.视图
    - 14-1：为什么要使用视图？什么是视图？那些特点?
- 15.存储过程与函数
    - 15-1：什么是存储过程?
    - 15-2：有哪些优缺点?
- 16.触发器
    - 16-1：什么是触发器?
    - 16-2：触发器的使用场景有哪些?
    - 16-3：MySQL中都有哪些触发器?
- 17.JDBC
    - 17-1：JDBC数据库连接步骤
    - 17-2：JDBC中的Statement 和PreparedStatement的区别?
    - 17-3：JDBC中大数据量的分页解决方法?
    - 17-4：说说数据库连接池工作原理和实现方案?
    - 17-5：Java中如何进行事务的处理?
    - 17-6：execute，executeQuery，executeUpdate的区别是什么?
    - 17-7：PreparedStatement的缺点是什么，怎么解决这个问题?
    - 17-8：JDBC的DriverManager是用来做什么的?
    - 17-9：JDBC的ResultSet是什么?
    - 17-10：有哪些不同的ResultSet?
    - 17-11：JDBC的DataSource是什么，有什么好处
    - 17-12：如何通过JDBC的DataSource和Apache Tomcat的JNDI来创建连接池?
    - 17-13：常见的JDBC异常有哪些?
    - 17-14：JDBC中存在哪些不同类型的锁?
    - 17-15：java.util.Date和java.sql.Date有什么区别?
    - 17-16：SQLWarning是什么，在程序中如何获取SQLWarning?
    - 17-17：如果java.sql.SQLException: No suitable driver found该怎么办?
    - 17-18：JDBC的RowSet是什么，有哪些不同的RowSet?
    - 17-19：什么是JDBC的最佳实践?
- ----------
- 1.缓存
    - 1-1：缓存思想
    - 1-2：使用缓存为系统带来了什么问题
    - 1-3：本地缓存解决方案
    - 1-4：为什么要有分布式缓存?/为什么不直接用本地缓存?

    - 1-5：缓存读写模式/更新策略
    - 1-6：为什么要用 redis/为什么要用缓存
    - 1-7：为什么要用 redis 而不用 map/guava 做缓存?
    - 1-8：缓存数据的处理流程是怎样的?
- 2.分布式缓存技术
    - 2-1：说一下 Redis 和 Memcached 的区别和共同点
    - 2-2：为什么说Redis快
    - 2-3：Redis应用场景
- 3.Redis五大数据类型与编码方式
    - 3-1：五大数据类型（理解就好）
    - 3-2：动态字符串与C语言自带字符串的区别
    - 3-3：Redis字典底层如何解决冲突
    - 3-4：hash如何扩容
    - 3-5：什么是渐进式
    - 3-6：什么时候使用ziplist而不是hash
    - 3-7：什么时候使用ziplist什么时候使用skiplist?
    - 3-8：skiplist原理
    - 3-9：ziplist原理
    - 3-10：为什么选择跳表而不是平衡树或者哈希表
    - 3-11：redis 数据存储过程
    - 3-12：数据淘汰过程
- 4.Redis的单线程
    - 4-1：为什么Redis是单线程
    - 4-2：既然是单线程，那怎么监听大量的客户端连接呢?
    - 4-3：Redis为什么又采用了多线程
- 5.过期策略
    - 5-1：Redis 给缓存数据设置过期时间有啥用?
    - 5-2：Redis是如何判断数据是否过期的呢?
    - 5-3：过期策略分类
    - 5-4：缓存淘汰机制
    - 5-5：LRU算法原理
    - 5-6：如何实现LRU
- 6.持久化机制
    - 6-1：什么是Redis持久化?
    - 6-2:各自的优缺点?
    - 6-3：Redis持久化数据和缓存怎么做扩容?
    - 6-4：持久化期间工作流程
    - 6-5：持久化机制
- 7.集群主从复制
- 8.缓存雪崩
    - 8-1：什么是缓存穿透
    - 8-2：缓存穿透解决方案
    - 8-3：什么是缓存雪崩
    - 8-4：有哪些解决办法?
- 9.如何保证缓存和数据库数据的一致性?
- 10.事务

    - 10-1：Redis事务的概念
    - 10-2：Redis事务的三个阶段
    - 10-3：Redis事务相关命令
    - 10-4：事务管理（ACID）概述
    - 10-5：Redis事务支持隔离性吗
    - 10-6：Redis事务保证原子性吗，支持回滚吗
- 11.Redis应用
    - 11-1：为什么Redis 变慢了
- ----------
- 1.Spring概述
    - 1-1：什么是spring?
    - 1-2：Spring用到的设计模式
    - 1-3：什么是 Spring 的循环依赖
    - 1-4：什么是三级缓存
    - 1-5：什么是早期暴露的对象
    - 1-6：如何解决循环依赖
    - Spring两大特性
    - Spring和Springboot的区别
- 2.Spring控制反转
    - 2-1：IOC原理
    - 2-2：IOC容器种类
    - 2-3：BeanFactory与ApplicationContext区别
    - 2-4：什么是依赖注入
    - 2-5：有哪些不同类型的IOC（依赖注入）方式?
    - 2-5：IOC与DI的的区别:
    - 2-6：
- 3.Spring面向切面编程(AOP)
    - 3-1：什么是aop
    - 3-2：AOP的相关概念：
- 3.Spring Bean
    - 3-1：Bean的五种作用域
    - 3-2：Spring 中的单例 bean 的线程安全问题了解吗?
    - 3-2：Bean的生命周期
    - 3-3：
- 4.Spring注解
- 5.Spring数据访问
- 6.
- 7.SpringMVC
    - 7-1：SpringMVC流程
- 8.Spring事务
    - 8-1：spring事务种类
    - 8-2：Spring两种事务区别
    - 8-3：事务管理接口
    - 8-3：事务传播行为
    - 8-4：事务隔离级别
    - 8-5：Spring的事务和数据库的事务隔离是一个概念么?

    - 8-6：事务属性详解
    - 8-7：事务传播行为
    - @Transactional 注解使用详解
    - 注解常用属性
    - 注解原理
- 9.MyBatis
    - 9-1：MyBatis是什么
    - 9-2：ORM是什么
    - 9-3：JPA
    - 9-4：MyBatis优点
    - 9-5：为什么说Mybatis是半自动ORM映射工具？它与全自动的区别在哪里？
    - 9-6：传统JDBC开发存在的问题，如何解决的
    - 9-7：Hibernate 和 MyBatis 的区别
    - 9-8：MyBatis的解析和运行原理
    - 9-9：MyBatis编程步骤是什么样的？
    - 9-10：MyBatis的工作原理
    - 2.5 为什么需要预编译
    - MyBatis是如何做到SQL预编译的呢？
    - 如何防止SQL注入
    - Mybatis都有哪些Executor执行器？它们之间的区别是什么？
    - 2.8 Mybatis是否支持延迟加载？如果支持，它的实现原理是什么？
- 10.映射器
    - 10-1：#{}和${}的区别是什么？
    - 10-2：Xml 映射文件中，除了常见的 select|insert|updae|delete 标签之外，还有哪些标签？
    - 最佳实践中，通常一个Xml映射文件，都会写一个Dao接口与之对应，请问，这个Dao接口的工作原理是什么？Dao接口里的方法，参数不同时，方法能重载吗
    - 3.11 Mybatis的Xml映射文件中，不同的Xml映射文件，id是否可以重复？
    - 简述Mybatis的Xml映射文件和Mybatis内部数据结构之间的映射关系？
    - Mybatis映射文件中，如果A标签通过include引用了B标签的内容，请问，B标签能否定义在A标签的后面，3.16 还是说必须定义在A标签的前面？
- 11.高级查询
    - 4.1 MyBatis实现一对一，一对多有几种方式，怎么操作的？
    - 4.2 Mybatis是否可以映射Enum枚举类？
- 12.动态SQL
    - 12-1：Mybatis动态sql是做什么的？都有哪些动态sql？能简述一下动态sql的执行原理不？
    - 12-2：Mybatis 是如何将 sql 执行结果封装为目标对象并返回的？都有哪些映射形式？
- 13.插件模块
    - 6.1 Mybatis是如何进行分页的？分页插件的原理是什么？
    - 6.2 简述Mybatis的插件运行原理，以及如何编写一个插件。
- 14.缓存
    - 14-1：Mybatis的一级、二级缓存
- 15.servlet
    - 15-1：Servlet生命周期?
    - forward和redirect的区别
    - tomcat容器是如何创建servlet类实例？用到了什么原理
    - Servlet安全性问题

    - servlet写就行了，为什么要有springMVC这个东西呢
    - 两者区别
    - servlet加载顺序
    - filter和Interceptor的区别
    - jsp和servlet区别
    - Servlet是线程安全的吗?
- 15.SpringBoot
    - 什么是 Spring Boot?
    - SpringBoot自动配置原理
- ----------
- 1.单例模式
    - 1.1：什么是单例模式
    - 1.2：为什么要用单例模式呢?
    - 1.3：简单来说使用单例模式可以带来下面几个好处:
    - 1.3：什么可以破坏单例模式
    - 1-4：单例有哪几种实现方式
    - 1-5：饿汉方式(线程安全)
    - 1-6：饿汉式（枚举方式）
    - 1-6：懒汉式（非线程安全和synchronized关键字线程安全版本）
    - 1-7：懒汉式(双重检查加锁版本)
    - 1-8：懒汉式（登记式/静态内部类方式）
    - 单例模式里面为什么写了volatile
- 2.工厂模式
    - 2-1：工厂模式的定义
    - 2-2：工厂模式的分类：
    - 2-3：在开源框架中的使用
    - 2-4：为什么要用工厂模式
    - 2-5：简单工厂例子
    - 2-6：使用反射机制改善简单工厂
    - 2-7：工厂方法模式
    - 2-8：抽象工厂模式
- 3.建造者模式
    - 3-1：什么是建造者模式
- 4.原型模式
- 5.代理模式

1.java基础-java特性

1-1：java特点

1. 简单易学；
2. 面向对象（封装，继承，多态）；
3. 平台无关性（Java 虚拟机实现平台无关性）；
4. 可靠性；
5. 安全性；

6. 支持多线程；
7. 支持网络编程并且很方便；
8. 编译与解释并存；

2.java基础-面对对象-综述

2-1：面对对象三大特征(特点)

1. 封装
2. 继承
3. 多态

2-2：面对对象的五大原则

1. 单一职责原则
    - 一个类，最好只做一件事，只有一个引起它的变化
2. 开放封闭原则
    - 对抽象编程，而不对具体编程
3. 里式替换原则
    - 子类必须能够替换其基类（这个是继承的关键）
4. 依赖倒置原则
    - 抽象不依赖于具体，具体依赖于抽象
5. 接口隔离原则
    - 使用多个小的专门的接口，而不要使用一个大的总接口

2-3：Java创建对象方式

1. 使用new关键字
2. 使用反射的机制创建对象
    - 使用Class类的newInstance方法
    - 使用Constructor类的newInstance方法
3. 使用clone方法
    - 需要已经有一个分配了内存的源对象，创建新对象时，首先应该分配一个和源对象一样大的内存空间。
4. 反序列化
    - 调用ObjectInputStream类的readObject（）方法

注：使用构造器的三种(new和反射的两种newInstance)，没用构造器的两种(clone和反序列化)

2-4：new abc是在堆里面呢?

放在堆里，栈里面放着引用

2-4：面向过程性能比面向对象高

Java 性能差的主要原因并不是因为它是面向对象语言，而是 Java 是半编译语言，最终的执行代码并不是可 以直接被 CPU 执行的二进制机械码。

3.java基础-面对对象-多态

3-1：多态的必要条件（实现方式）

1. 有类继承或者接口实现
2. 子类要重写父类的方法
3. 父类的引用指向子类的对象

3-2：多态机制

靠的是父类或接口定义的引用变量可以指向子类或具体实现类的实例对象

3-3：多态的好处

1. 应用程序不必为每一个派生类编写功能调用，只需要对抽象基类进行处理即可。大大提高程序的可复用性。
2. 派生类的功能可以被基类的方法或引用变量所调用，可以提高可扩充性和可维护性。

3-4：多态的例子

4.java基础-面对对象-重载与重写

4-1：重载与重写

1. 重载——函数或者方法有同样的名称，但是参数列表不相同的情形
2. 重写——Java的子类与父类中有两个名称、参数列表都相同的方法的情况。由于他们具有相同的方法签名，所以子类中的新方法将覆盖父类中原有的方法

4-2：Java 中是否可以覆盖(override)一个 private 或者是 static 的方法?

1. Java中static方法不能被覆盖， 因为方法覆盖是基于运行时动态绑定的， 而static方法是编译时静态 绑定的。

2. Java中也不可以覆盖private的方法， 因为private修饰的变量和方法只能在当前类中使用，如果是其他 的类继承当前类是不能访问到 private 变量或方法的， 当然也不能覆盖。

5.java基础-面对对象-接口与抽象类

5-1：接口和抽象类的区别是什么?

1. 所有方法在接口中不能有实现，而抽象类可以有非抽象的方法。

2. 接口中除了static final变量，不能有其他变量，而抽象类中则不一定。

3. 一个类可以实现多个接口，但只能实现一个抽象类。

4. 从设计层面来说，抽象类作为很多子类的父类，是一种模板式设计，接口是一种行为规范

5-2：Java 抽象类可以有构造函数吗？作用是什么

可以有， 抽象类可以声明并定义构造函数。

它可以用来初始化抽象类内部声明的通用变量，并被各种实现使用。

5-3: Java 抽象类可以实现接口吗？ 它们需要实现所有的方法吗?

可以， 抽象类可以通过使用关键字implements来实现接口。

5-4：Java 抽象类可以是 final 的吗?

不可以， Java 抽象类不能是 final 的。 将它们声明为final的将会阻止它们被继承， 而这正是使用抽象类唯一的方法。

5-5：Java 抽象类可以有 static 方法吗?

可以， 抽象类可以声明并定义 static 方法， 没什么阻止这样做。

5-6：可以创建抽象类的实例吗?

不可以， 当一段代码尝试实例化一个抽象类时 Java 编译器会抛错误。

5-7：抽象类必须有抽象方法吗?

不需要， 抽象类有抽象方法不是强制性的。 但是一般在抽象类中设置抽象方法

5-8：何时选用抽象类而不是接口?

1. 当关心升级时， 因为不可能在一个发布的接口中添加一个新方法， 用抽象类会更好。
2. 如果你的接口中有很多方法， 你对它们的实现感到很头疼， 考虑提供一个抽象类作为默认实现。

5-9：Java中的抽象方法是什么?

1. 抽象方法是一个没有方法体的方法。 你仅需要声明一个方法，
2. 不需要定义它并使用关键字abstract 声明。

5-10：Java抽象类中可以包含main方法吗?

是的， 抽象类可以包含 main 方法， 它只是一个静态方法， 你可以使用 main 方法执行抽象类， 但不可以创建任何实例。

5-11：创建一个对象用什么运算符?对象实体与对象引用有何不同?

new 运算符，new创建对象实例，对象引用指向对象实例。一个对象引用可以指向0个或1个对象;一个对象可以有n个引用指向它。

5-12：interface实现方法

6.java基础-值传递与引用传递

1. 值传递是对基本型变量而言的,传递的是该变量的一个副本,改变副本的值不影响原变量的值
2. 引用传递一般是对于引用类型变量而言的,传递的是该对象地址的一个副本，是一个地址。
    - 如果说改变了原地址的值（注意是 值 ），那么会影响
    - 如果改变了副本地址，如new 一个原地址不会改变

一般认为,java传递都是值传递.

7.java基础-深拷贝与浅拷贝

7-1：深拷贝与浅拷贝

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

7-2：浅拷贝方法

1. 通过拷贝构造方法实现浅拷贝：
    - 拷贝构造方法指的是该类的构造方法参数为该类的对象。
2. 通过重写clone()方法进行浅拷贝
    - 使用clone方法的类必须实现Cloneable接口

7-3：深拷贝方法

参考文献

1. 序列化
    - 序列化为数据流，在反序列化回来，就可以得到这个对象
2. 利用Kryo框架，这是一个快速高效的Java序列化框架
3. 利用json转化方式
    - 对象转化为JSON，再序列化为对象
4. 人工构建对象

8.java基础-面对对象-构造器

8-1：一个类的构造方法的作用是什么? 若一个类没有声明构造方法，该程序能正确执行吗? 为什么?

主要作用是完成对类对象的初始化工作。可以执行。因为一个类即使没有声明构造方法也会有默认的不带参数的构造方法。

8-2：构造方法有哪些特性?

1. 名字与类名相同。
2. 没有返回值，但不能用 void 声明构造函数。

3. 生成类的对象时自动执行，无需调用。

9.java基础-面对对象-静态与非静态

9-1：静态方法和实例方法有何不同

1. 调用静态方法可以无需创建对象。而实例方法需要用类名.方法名访问
2. 静态方法在访问本类的成员时，只允许访问静态成员，而不允许访问实例成员变量和实例方法；实例方法则无此限制。

9-2：静态变量和实例变量的区别?

1. 语法区别： 静态变量前要加 static 关键字，而实例变量前则不加。
2. 程序运行的区别：实例变量必须创建对象后才可以通过这个对象来使用，静态变量则可以直接使用类名来引用。

10.java基础-运算符-相等问题

10-1： == 与 equals

1. == : 它的作用是判断两个对象的地址是不是相等。 基本数据类型比较的是值，引用数据类型比较的是内存地址
2. equals() : 它的作用也是判断两个对象是否相等。但它一般有两种使用情况：
    - 情况 1：类没有覆盖equals()方法。则通过equals()比较该类的两个对象时，等价于通过"=="比 较这两个对象。
    - 情况 2：类覆盖了equals()方法。一般，我们都覆盖 equals() 方法来比较两个对象的内容是否相等；若它们的内容相等，则返回true。

10-2：为什么要重写hashcode与equals

如果只重写了equals方法而没有重写hashCode方法的话，那么就会违背，相等的对象必须具有相等的散列码（hashCode）。

同时对于HashSet和HashMap这些基于散列值（hash）实现的类。

HashMap的底层处理机制是以数组的方法保存放入的数据的，关键就是数组下标的处理。

数组的下标是根据传入的元素hashCode方法的返回值再和特定的值异或决定的。

如果该数组位置上已经有放入的值了，且传入的键值相等则不处理，

若不相等则覆盖原来的值，如果数组位置没有条目，则插入，并加入到相应的链表中。

检查键是否存在也是根据hashCode值来确定的。所以如果不重写hashCode的话，

可能导致HashSet、HashMap不能正常的运作、

如果我们将某个自定义对象存到HashMap或者HashSet及其类似实现类中的时候，

如果该对象的属性参与了hashCode的计算，那么就不能修改该对象参数hashCode计算的属性了。有可能会移除不了元素，导致内存泄漏。

10-3：重写equals不重写hashcode会出现什么问题

在存储散列集合时 ,如果原对象equals(新对象) ,但没有对hashCode重写,则在集合中将会存储两个值相同的对象,从而导致混淆。

10-4：为什么两个对象有相同的hashcode值，它们也不一定是相等的?

hashCode方法实际上返回的就是对象存储的物理地址，也就是说 hashcode 只是用来缩小查找成本。

10-5：hashcode和equals源码写一下

equals()方法在object类中定义如下：

```
public boolean equals(Object obj) {
    return (this == obj);  //用来比较其他对象是否等于此对象
}
```

比如说在String类中重写

```
public boolean equals(Object anObject) {
        //使用==操作符检查“参数是否为这个对象的引用”(比较对象地址)
        if (this == anObject) {
            return true;
        }
        //用instanceof检查“参数是否为正确的类型(是否为String的实例)”
        if (anObject instanceof String) {
            String anotherString = (String)anObject;
            int n = value.length;
            //判断两个字符串的长度是否相同
            if (n == anotherString.value.length) {
                char v1[] = value;
                char v2[] = anotherString.value;
                int i = 0;
                //一个字符一个字符的进行比较
                while (n-- != 0) {
                    if (v1[i] != v2[i])
                        return false;
                    i++;
                }
                return true;
            }
        }
        return false;
    }
```

```
public int hashCode() {
      int h = hash;
      if (h == 0 && value.length > 0) {
          char val[] = value;

          for (int i = 0; i < value.length; i++) {
              h = 31 * h + val[i];
          }
          hash = h;
      }
      return h;
 }
```

10-5：说说&和&&的区别。

1. &和&&都可以表示逻辑与，当运算符两边的表达式的结果都为 true 时，整个运算结果才为 true，否则，只要有一方为 false，则结果为 false。
2. &&还具有短路的功能，即如果第一个表达式为 false，则不再计算第二个表达式，所以不会出现 NullPointerException
3. &还可以用作位运算符，当&操作符两边的表达式不是 boolean 类型时， &表示按位与操作

10-6：用最有效率的方法算出 2 乘以 8 等于几?

用移位运算符

2<<3

- 因为将一个数左移 n 位，就相当于乘以了2的 n 次方

10-7：i++和++i的区别，及其线程安全问题

i++：先赋值再自加。

++i：先自加再赋值。

1. 如果i是局部变量（在方法里定义的），那么是线程安全的。因为局部变量是线程私有的，别的线程访问不 到

2. 如果i是全局变量（类的成员变量），那么是线程不安全的。因为如果是全局变量的话，同一进程中的不同 线程都有可能访问到。

如果有大量线程同时执行i++操作，i变量的副本拷贝到每个线程的线程栈，当同时有两个线程栈以上的线程读取线程变量，假如此时是1的话，那么同时执行i++操作，再写入到全局变量，最后两个线程执行完，i会等于3而不会是2，所以，出现不安全性。

11.java基础-数据类型-8种常见类型

11-1：八种数据类型是什么？-中兴

1. 字符型 char
2. 布尔型 boolean
3. 数值型
    1. 整型：byte、short、int、long
    2. 浮点型：float、double

<font?color=red>特别注意：String不是基本数据类型，是引用类型。

11-2：数据类型的范围

byte：-2^7 ~ 2^7-1，

short：-2^15 ~ 2^15-1

int：-2^31 ~ 2^31-1

long：-2^63 ~ 2^63-1，

浮点型：

float：4字节。

double：8字节。

字符型：

char：2字节。

11-3：为什么byte类型是-128~+127

使用原码或反码表示的范围为[-127, +127], -0并没有反码补码表示，而使用补码表示的范围为[-128, 127]

11-4：java为什么除了基本数据类型还要有引用数据类型

引用类型在堆里，基本类型在栈里。 栈空间小且连续，往往会被放在缓存。引用类型cache miss（缓存未命中）率高且要多一次解引用。对象还要再多储存一个对象头，对基本数据类型来说空间浪费率太高

11-5：String为什么不是基本数据类型-字节

1. 基本类型仅表示简单的数据类型，引用类型可以表示复杂的数据类型，还可以操作这种数据类型的行为
2. java虚拟机处理基础类型与引用类型的方式是不一样的，对于基本类型，java虚拟机会为其分配数据类型实际占用的内存空间，而对于引用类型变量，他仅仅是一个指向堆区中某个实例的指针。

12.java基础-数据类型-自动拆装箱

12-1：为什么要有自动拆装箱

比如说集合类中，要求元素必须是Object类，而int、double等基本数据类型无法使用，那么我们就把基本数据类型包装起来，使其具有对象的特征，并让他有了属性和方法

12-2：Integer缓存机制

Integer对小数据（-128~=127）具有缓存机制，当jvm在初始化的时候，如果数据是小数据，那么就会把数据存储在本地内存当中，当下次使用的时候该数据的时候，那么就可以直接从本地内存进行调用，就不需要再次创建对象来解决

1. 其中会有一个valueof函数，用来判断内存中是否有着这个数值，
    - 如果说有，那么直接从内存进行读取
    - 如果说没有，那么就需要创建一个对象
2. 在jvm初始化的时候，低值是不能改变的，但是高值是可以改变的，可以通过jvm进行参数设置，但是只有Integer源码可以对高值、低值进行改变。

12-3：自动拆装箱的原理

```
* 自动装箱：调用valueOf () 方法将原始类型值转换成对象
* 自动拆箱：调用intValue()方法，其他的（xxxValue())这类的方法将对象转换成原始类型值。
```

12-4：自动拆装箱使用场景

1. 场景一、将基本数据类型放入集合类
2. 场景二、包装类型和基本类型的大小比较
3. 场景三、包装类型的运算
4. 场景四、三目运算符的使用如果i是包装类，j是null就会NPE报错
5. 场景五、函数参数与返回值

12-5：自动拆装箱带来的问题

1. 包装对象的数值比较，不能简单的使用==，~~虽然-128到127之间的数字可以，但是这个范围之外还是需要使用equals比较~~。（笔试选择较为频繁，跟谁学考过）

2. 如果包装类对象为null，那么自动拆箱时就有可能抛出NPE（NullPointException）。

3. 如果一个for循环中有大量拆装箱操作，会浪费很多资源。

12-6：String转出int型， 判断能不能转？ 如何转?

可以转， 得处理异常 Integer.parseInt(s) 主要为 NumberFormatException：

12-7：short s1 = 1; s1 = s1 + 1;有什么错? short s1 = 1; s1+=1;有什么错?

对于 short s1 = 1; s1 = s1 + 1;由于 s1+1运算时会自动提升表达式的类型，所以结果是 int型，再赋值给 short 类型 s1时， 编译器将报告需要强制转换类型的错误。

对于 short s1 = 1; s1 += 1;由于 +=是 java 语言规定的运算符， java 编译器会对它进行特殊处理，因此可以正确编译

12-8：int与Integer区别

1. Integer是int的包装类， int则是java的一种基本数据类型
2. Integer变量必须实例化后才能使用，而int变量不需要
3. Integer的默认值是null, int的默认值是0

注：

1. 非new生成的Integer变量和new Integer生成的变量比较时， 结果为false。（因为非new生成的Integer变量指向的是java常量池中的对象，而new Integer0生成的变量指向堆中新建的对象，两者在内存中的地址不同)

```
Integer i = new Integer(100);
Integer j = 100;
Syatem.out.print(i == J); //false
```

2. Integer变量和int变量比较时，只要两个变量的值是相等的，则结果为true

13.java基础-关键字-访问权限关键字

13-1：访问控制关键字级别

13-2: 通过反射访问private成员和方法，既然能访问为什么要private?

1. private并不是解决安全问题的，如果想让解决代码的安全问题，请用别的办法。
2. private的意义是OOP（面向对象编程）的封装概念。

14.java基础-关键字-static

14-1：static使用场景

1. 修饰成员变量和成员方法

2. 静态代码块:

3. 静态内部类（static修饰类的话只能修饰内部类）：

4. 静态导包:

15.java基础-关键字-final关键字

15-1：final关键字使用特点

1. final修饰的类不能被继承

2. final修饰的方法不能被重写;

3. final修饰的变量是常量，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能让其指向另一个对象。

4. 想通过使用final提升程序性能

    - 因为编译器能从final中获取额外的信息，因此可以对类或者方法调用进行额外的优化处理。但这中优化对程序性能的提升极其 有限。

15-2：final, finally, finalize 的区别。

1. final 用于声明属性，方法和类，分别表示属性不可变，方法不可覆盖，类不可继承。

2. finally 是异常处理语句结构的一部分，表示总是执行。

3. finalize 是 Object 类的一个方法，在垃圾收集器执行的时候会调用被回收对象的此方法，可 以覆盖此方法提供垃圾收集时的其他资源回收，例如关闭文件等。

15-3：使用 final 关键字修饰一个变量时，是引用不能变，还是引用的对象不能变?

使用 final 关键字修饰一个变量时，是指引用变量不能变，引用变量所指向的对象中的内容还是可以改变的。

16.java基础-关键字-this关键字和super关键字

1. this程序

```
class Manager {
    Employees[] employees;

    void manageEmployees() {
        int totalEmp = this.employees.length;
        System.out.println("Total employees: " + totalEmp);
        this.report();
    }

    void report() { }
}
```

主要是访问本类（自己）的成员变量和方法（可写可不写） super主要是通过子类去访问父类的成员变量和方法，必须写 2. super程序

```
public class Super {
    protected int number;

    protected showNumber() {
        System.out.println("number = " + number);
    }
}
```

```
public class Sub extends Super {
    void bar() {
        super.number = 10;
        super.showNumber();
    }
}
```

1. 在构造器中使用 super（） 调用父类中的其他构造方法时，该语句必须处于构造器的首行，否则编译器会报错。
2. this 调用本类中的其他构造方法时，也要放在首行。
3. this、super不能用在static方法中。

17.java基础-关键字-transient

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

18.java基础-集合map-hashmap的数据结构

18-1：hashmap的数据结构

1. JDK1.7用的是头插法，而JDK1.8及之后使用的都是尾插法，JDK1.7采用头插法虽然能够提高插入的效率，但是为了安全,防止环 化，因为resize的赋值方式，也就是使用了单链表的头插入方式，同一位置上新元素总会被放在链表的头部位置，在旧数组中同 一条Entry链上的元素，通过重新计算索引位置后，有可能被放到了新数组的不同位置上。使用头插会改变链表的上的顺序，但是 如果使用尾插，在扩容时会保持链表元素原本的顺序，保持之前节点的引用关系，就不会出现逆序且链表死循环的问题

2. （扩容机制）扩容后数据存储位置的计算方式也不一样：

    1. 在JDK1.7的时候是直接用hash值和需要扩容的二进制数进行&
    2. 而在JDK1.8的时候直接用了JDK1.7的时候计算的规律，也就是扩容前的原始位置+扩容的大小值=JDK1.8的计算方式，但是这种方式就相当于只需要判断Hash值的新增参与运算的位是0还是1就直接迅速计算出了扩容后的存储方式。

3. （插入元素）JDK1.7的时候使用的是数组+ 单链表的数据结构。HashMap通过key的hashCode经过扰动函数处理过后得到hash 值，然后通过(n-1)&hash判断当前元素存放的位置，如果当前位置存在元素的话，就判断该元素与要存入的元素的hash值以及 key是否相同，如果相同的话，直接覆盖，不相同就通过拉链法解决冲突。但是在JDK1.8及之后时，使用的是数组+链表+红黑树的 数据结构,当链表的深度达到8的时候，也就是默认阈值，就会自动扩容把链表转成红黑树的数据结构，以减少搜索时间。

18-1：扩容死循环问题

比如说若当前线程在扩容并发的时候，此时获得ertry节点，但是被线程中断无法继续执行，此时线程二进入transfer 函数，并把函数顺利执行，

此时新表中的某个位置有了节点，之后线程一获得执行权继续执行，因为并发 transfer，所以两者都是扩容的同一个链表，

当线程一执行到new table[i]的时候，由于线程二之前数据迁移的原因导致此时new table[i]上就有ertry存在，

所以线程一执行的时候，会将next节点，设置为自己，导致自己互相使用next引用对方，因此产生链表，导致死循环。

但是在JDK 8用head 和 tail 来保证链表的顺序和之前一样。

18-2；为什么JDK8时候引入了红黑树?

因为当数组中每个元素，都是一个Entry，每一个Entry是一个单链表。

当链表长度过长的时候，查询链表中的一个元素就比较耗时，这时就引入了红黑树。

首先红黑树是一棵二叉树，而且属于二叉树中比较特殊的二叉搜索树。红黑树有一条特性就是从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。这一特 性，确保没有一条路径会比其他路径长出两倍，因而，红黑树是接近平衡的二叉树。这就使得红黑树的时间复杂度大大降低。

所以，用红黑树替代单链表会降低集合中元素的访问速度。

18-3：为什么不把链表全部换为红黑树

1. 链表的结构比红黑树简单，构造红黑树要比构造链表复杂，所以在链表的节点不多的情况下，从整体的性能看来，如果把链表全部换为红黑树，效率反而更低。
2. HashMap频繁的resize（扩容），扩容的时候需要重新计算节点的索引位置，也就是会将红黑树进行拆分和重组，其实这是很复杂的，这里涉及到红黑树的着色和旋转，所以为链表树化设置一个阀值是非常有必要的。

18-4：为什么是使用红黑树而不是AVL树?

在CurrentHashMap中是加锁了的，实际上是读写锁，如果写冲突就会等待，如果插入时间过长必然等待时间更长。同时因为AVL树需要更高的旋转次数才能在修改时正确地重新平衡数据结构，所以红黑树相对AVL树他的插入更快!

18-5：为什么在JDK1.7的时候是先进行扩容后进行插入，而在JDK1.8的时候则是先插入后进行扩容的呢?

在JDK1.7中的话，是先进行扩容后进行插入的，就是当你发现你插入的桶是不是为空，说明存在值就发生了hash冲突，那么就必须得扩容，但是如果不发生Hash冲突的话，说明当前桶是空的（后面并没有挂有链表），那就等到下一次发生Hash冲突的时候在进行扩容，但是当如果以后都没有发生hash冲突产生，那么就不会进行扩容了，减少了一次无用扩容，也减少了内存的使用

18-6：为什么在JDK1.8中进行对HashMap优化的时候，把链表转化为红黑树的阈值是8,而不是7或者不是20呢（面试蘑菇街问过）

1. 中间有个差值7可以有效防止链表和树频繁转换，降低效率

2. 由于treenodes的大小大约是常规节点的两倍，因此我们仅在容器包含足够的节点以保证使用时才使用它们，当它们变得太小（由于移除或调整大小）时，它们会被转换回普通的node节点，容器中节点分布在

hash桶中的频率遵循泊松分布，桶的长度超过8的概率非常非常小。

18-7：扰动函数以及作用

HashMap的hash方法。

为了防止一些实现比较差的hashCode()方法，使用扰动函数之后可以减少碰撞。

18-8：哈希冲突的解决方法

1. 拉链法

创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可。

2. 开发地址法

所有输入的元素全部存放在哈希表里，先通过哈希函数进行判断，若是发生哈希冲突，就以当前地址为基准，根据再寻址的方法（探查序列），去寻找下一个地址，若发生冲突再去寻找，直至找到一个为空的地址为止。

18-9：HashMap的put操作

HashMap通过key的hashCode经过扰动函数处理过后得到hash值，然后通过计算判断当前元素存放的位置

1. 如果数组的这个位置是空的，把key放进去，put操作就完成了。
2. 如果数组位置不为空，就判断该元素与要存入的元素的hash值以及key是否相同，如果相同的话，直接覆盖
3. 若果不相等，这个元素必然是个链表。遍历链表逐一比对value，如果value在链表中不存在，就把新建节点，将value放进去，put操作完成。
4. 如果链表中value存在，则替换原节点的value，put操作完成。
5. 如果链表节点数已经达到8个，首先判断当前hashMap的长度，如果不足64，只进行resize，扩容table，如果达到64就将冲突的链表为红黑树。

18-10：HashMap的get操作

1. 查找位置。
2. 如果访问的节点是bucket里的第一个节点，则直接命中；
3. 如果有冲突，则通过key.equals(k)去树或链表中查找对应的entry。

18-11：hashmap的get和put操作的时间复杂度

如果说一个entry数组下标最多只对应了一个entry，此时get方法的时间复杂度可以达到O(1)。

但是如果所有的hash都一样，那么退化为线性查找，变成了O（n）

18-12：hashmap的String类型如何计算hashcode的

就是以31为权，每一位为字符的ASCII值进行运算，用自然溢出来等效取模。

选择值31是因为它是素数。如果是偶数并且乘法运算溢出，则信息将丢失，因为乘以2等于移位。

31的一个不错的特性是乘法可以用移位和减法来代替，以获得更好的性能

哈希分布比较均匀。偶数的冲突率很高，只有少数例外。小乘数（1-20）的冲突率也很高

18-13：reHash过程

1. 首先创建一个比现有哈希表更大的新哈希表（expand）
2. 然后将旧哈希表的所有元素都迁移到新哈希表去（rehash）

18-14：hash函数以及常用方法

1. 直接定址法：直接以key或者key上加上某个常数作为哈希地址
2. 数字分析法：提取key中取值比较均匀的数字作为哈希地址
3. 除留余数法：用key除以某个不大于哈希表长度m的数，将所得余数作为哈希地址
4. 分段叠加法：按照哈希表地址位数将关键字分成了位数相等的几部分，其中最后一部分可以比较短，然后将这几部分相加，舍弃最 高位仅为后的结果就是该关键字的哈希地址
5. 平方取中法：如果key的各部分分部都不均匀，可以先求出他的平方值，然后按照需求求取中间的几位作为哈希地址
6. 伪随机数法：采用一个伪随机数作为哈希函数

18-15：HashMap为什么要树化?

安全问题。因为在元素放置过程中，如果一个对象哈希冲突，都被放置到同一个桶中，则会形成一个链表。而链表查询时线性的，会 严重影响存取的性能。

18-16：hashmap树化门槛及作用

- 链表长度大于8
- 数组长度大于64

作用： 这个本质上，是一个安全问题。因为在元素放置过程中，如果一个对象哈希冲突，都被放置到同一个桶中，则会形成一个链表。而链 表查询是线性的，会严重影响存取的性能。

18-17：hashmap的特性

1. 允许空键和空值（但空键只有一个，且放在第一位）
2. 元素是无序的，而且顺序会不定时改变
3. key不允许重复。

18-18：HashMap为什么可以插入空值?

HashMap在put的时候会调用hash()方法来计算key的hashcode值，可以从hash算法中看出当key==null时返回的值为0。因此key为 null时，hash算法返回值为0，不会调用key的hashcode方法。

18-19：JDK的hashmap与Redis的hashmap的区别

1. HashMap由于对链表size超过8采用二叉树结构，使得get操作随着激烈冲突导致变成一个类二叉树，时间复杂度为O(log(n))较 redis的字典表O(n)，性能提升明显。

2. Redis的rehash由于采用渐进式的方式，对于大数据量下的rehash操作性能提升明显。这也是由于HashMap大部分用于临时且数 据量不是特别大的数据，redis的hash用于存储避免大数据情况导致异常，双方的侧重点不一样。
3. Redis的单链表在冲突的情况下是从表头插入，时间复杂度为O(1)，而HashMap则为O(n)。

18-20：为什么要两次hash

两个不同的键值，在对数组长度进行按位与操作后得到的结果相同，就会发生冲突

19.java基础-集合map-hashmap源码数值分析

19-1：HashMap中(tab.length - 1) & hash作用

1. 保证不会发生数组越界
2. 保证元素尽可能的均匀分布

19-2：请解释一下HashMap的参数loadFactor，它的作用是什么?

loadFactor表示HashMap的拥挤程度

作用：影响hash操作到同一个数组位置的概率。

19-3：HashMap的扩容因子为什么是0.75

1. 如果设置过大，如0.85，桶中键值对碰撞的几率就会越大，同一个桶位置可能会存放好几个value值，这样就会增加搜索的时间， 性能下降。
2. 如果设置过小，如0.1，那么10个桶，threshold为1，你放两个键值对就要扩容，太浪费空间了。

19-4：为什么默认初始化桶数组大小为16

如果桶初始化桶数组设置太大，就会浪费内存空间，16是一个折中的大小，既不会像1，2，3那样放几个元素就扩容，也不会像几千几 万那样可以只会利用一点点空间从而造成大量的浪费。

19-5：hashmap为什么是2的次幂

取模运算可以变成位与运算，效率显著提高！但是要浪费一些空间。

20.java基础-集合map-hashmap线程问题

20-1：hashMap是否线程安全

在JDK1.7的时候没有加入同步锁保护，同时由于JDK1.7在并发执行put造作导致扩容行为从而导致环形链表，在获取数据遍历链表形 成死循环，同时hashmap迭代器的fail-fast策略，一旦在使用地带器过程中出现并发操作，就会跑出异常。

那么JDK1.8虽然解决了死循环问题，但是还是没有同步锁保护机制，所以依然线程不安全

所以多线程情况下，首选线程安全的ConcurrentHashMap

20-2：为什么hashmap中String、integer包装类适合作为key

1. 包装类重写了equals\hashcode方法，不容易出现hash值计算错误
2. 由于String类型是final的，保证了key的不可更改性

20-3：线程安全的Map

- Hashtable
- ConcurrentHashMap
- SynchronizedMap

1. Hashtable、SynchronizedMap源码中是使用synchronized来保证线程安全的

2. ConcurrentHashMap沿用了与它同时期的HashMap版本的思想，底层依然由“数组”+链表+红黑树的方式思想，但是 ConcurrentHashMap没有对整个hash表进行锁定，而是采用了分离锁（segment）的方式进行局部锁定。具体体现在，它在代码 中维护着一个segment数组。

20-4：设计线程安全的map

1. 使用synchronized来进行约束：

2. 使用JDK1.5版本所提供的lock机制

3. 使用JDK提供的读写锁

4. 使用JDK1.5提供的ConcurrentHashMap,该类将Map的存储空间分为若干块,每块拥有自己的锁,减少了多个线程争夺同一个锁的情 况

21.java基础-集合map-ConcurrentHashMap

21-1：ConcurrentHashMap的底层实现

从JDK1.7版本的数组+Segment+分段锁的方式实现，分段锁Segment，它类似于HashMap的结构，内部拥有一个Entry数组，数组中的 每个元素又是一个链表,同时又是一个ReentrantLock。ConcurrentHashMap内部使用分段锁技术，将数据分成一段一段的存储，然 后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问，能够实现真正的并发 访问。虽然在写操作的时候可以只对元素所在的Segment进行加锁即可，不会影响到其他的Segment这一种结构，但是带来的副作用是 Hash的过程要比普通的HashMap要长

到JDK1.8版本中synchronized+CAS+HashEntry+红黑树。数据结构上取消了Segment分段锁的数据结构，取而代之的是数组+链表 +红黑树的结构。为了保证线程安全，JDK1.8采用CAS+Synchronized保证线程安全。
JDK1.8现调整为对每个数组元素加锁。由于定 位结点的hash算法简化会带来弊端,Hash冲突加剧,因此在链表节点数量大于8时，会将链表转化为红黑树进行存储。这样查询时间复 杂度会得到改善

21-2：为何会出现ConcurrenHashMap?

1. 线程安全，读写还快，以空间换时间

2. 改善了hashmap迭代器出现的ConcurrentModificationException

- 由于ConcurrentHashMap对于会产生并发操作的node都会有加锁同步处理,且迭代器获取tab[index]开头node时都会从主存来获

  得,保证获取的数据是最新的，从而保证了迭代器在迭代过程中即使有put , remove 等操作同时发生也可以保证迭代的安全性,不会

  出现ConcurrentModificationException

21-3：为什么ConcurrentHashMap（hashtable）为何不支持null键和null值

ConcurrentHashmap和Hashtable都是支持并发的，这样会有一个问题，当你通过get(k)获取对应的value时，如果获取到的是null 时，你无法判断，它是put（k,v）的时候value为null，还是这个key从来没有做过映射。HashMap是非并发的，可以通过contains (key)来做这个判断。而支持并发的Map在调用m.contains（key）和m.get(key),m可能已经不同了。

21-4：ConcurrentHashMap的put操作

1. 首先判断是否初始化，如果没有初始化则进入initTable()方法进行初始化工作
2. 如果已经初始化了，进入无限循环，判断key对应的数组下标是否有值了
3. 如果key对应的下标没有值，通过CAS原理插入，插入成功则退出循环，插入失败则继续循环
4. 如果key对应的下标已经存在值,判断此时hash==MOVED(-1),则进入帮助扩容。
5. 如果key对应的下标已经存在值，但是hash!=MOVED,则需要对数组的这个下标进行加锁了，以保证线程的安全。
6. 如果数组的这个下标是一个链表，则对操作链表（判断链表用hash>=0）
7. 如果数组的这个下标是一个红黑树，则操作红黑树。
8. 插入成功后，如果链表的长度已经达到了红黑树的阀门8，则首先判断此时数组的长度是否大于64，如果小于64则进行扩容，如果 大于等于64则链表变成红黑树
9. 判断容器是否扩容

21-5：分段锁原理

21-6：hashmap与ConcurrentHashMap中put的区别

21-7：扩容机制

1. 通过计算 CPU 核心数和 Map 数组的长度得到每个线程要帮助处理多少个桶，并且这里每个线程处理都是平均的。默认每 个线程处理 16 个桶。因此，如果长度是 16 的时候，扩容的时候只会有一个线程扩容。

2. 初始化临时变量,将其在原有基础上扩容两倍。

3.死循环开始转移。多线程并发转移就是在这个死循环中，根据一个 finishing 变量来判断，该变量为 true 表示扩容结束，否则 继续扩容。 3.1 进入一个 while 循环，分配数组中一个桶的区间给线程，默认是 16. 从大到小进行分配。当拿到分配值后，进行 i-- 递减。 这个 i 就是数组下标。 （其中有一个 bound 参数，这个参数指的是该线程此次可以处理的区间的最小下标，超过这个下标，就需要 重新领取区间或者结束扩容，还有一个 advance 参数，该参数指的是是否继续递减转移下一个桶，如果为 true，表示可以继续向后 推进，反之，说明还没有处理好当前桶，不能推进) 3.2 出 while 循环，进 if 判断，判断扩容是否结束，如果扩容结束，清空临死

变量，更新 table 变量，更新库容阈值。如果没 完成，但已经无法领取区间（没了），该线程退出该方法，并将 sizeCtl 减一，表示扩容的线程少一个了。如果减完这个数以后， sizeCtl 回归了初始状态，表示没有线程再扩容了，该方法所有的线程扩容结束了。（这里主要是判断扩容任务是否结束，如果结束 了就让线程退出该方法，并更新相关变量）。然后检查所有的桶，防止遗漏。 3.3 如果没有完成任务，且 i 对应的槽位是空，尝试 CAS 插入占位符，让 putVal 方法的线程感知。 3.4 如果 i 对应的槽位不是空，且有了占位符，那么该线程跳过这个槽位，处理下一个槽位。 3.5 如果以上都是不是，说明这个槽位有一个实际的值。开始同步处理这个桶。 3.6 到这里，都还没有对桶内数据进行转移，只是计算了下标和处理区间，然后一些完成状态判断。同时，如果对应下标内没有数据 或已经被占位了，就跳过了。

处理每个桶的行为都是同步的。防止 putVal 的时候向链表插入数据。 4.1 如果这个桶是链表，那么就将这个链表根据 length 取于拆成两份，取于结果是 0 的放在新表的低位，取于结果是 1 放在新 表的高位。 4.2 如果这个桶是红黑数，那么也拆成 2 份，方式和链表的方式一样，然后，判断拆分过的树的节点数量，如果数量小于等于 6， 改造成链表。反之，继续使用红黑树结构。 4.3 到这里，就完成了一个桶从旧表转移到新表的过程。

21-8：什么时候会发生扩容机制

1. put操作（插入键值对）
2. putAll操作（批量插入键值对）
3. remove操作（移除元素，底层实现是用null空值代替原位元素）
4. replace操作（对已存在的键值对替换值）
5. computeIfAbsent操作（若key对应的value为空，会将第二个参数的返回值存入并返回）

22.java基础-集合map-TreeMap

22-1：TreeMap底层原理：

TreeMap是桶+红黑树的实现方式.TreeMap的底层结构就是一个数组,数组中每一个元素又是一个红黑树.当添加一个元素 (key-value)的时候,根据key的hash值来确定插入到哪一个桶中(确定插入数组中的位置),当桶中有多个元素时,使用红黑树进行保 存;当一个桶中存放的数据过多,那么根据key查找的效率就会降低

22-2：使用场景

1. 需要基于排序的统计功能：
2. 需要快速增删改查的存储功能：
3. 需要快速增删改查而且需要保证遍历和插入顺序一致的存储功能：

23.java基础-集合map-LinkedHashmap

23-1：linkedhashmap的底层原理

linkedhashmap继承HashMap，他比hashmap多维护了一个双向链表

24.java基础-集合map-HashTable

24-1：HashTable的底层原理

HashTable的与HashMap中相似，有一点重大区别就是所有的操作都是通过synchronized锁保护的。只有获得了对应的锁，才能进行后续的读写等操作。

25.java基础-集合list-ArrayList

25-1：数组(Array)和列表(ArrayList)有什么区别？ 什么时候应该使用 Array 而不是ArrayList?

1. 定义上： Array 可以包含基本类型和对象类型， ArrayList 只能包含对象类型。
2. 容量上： Array 大小固定， ArrayList 的大小是动态变化的。
3. 操作上： ArrayList 提供更多的方法和特性，

使用基本数据类型或者知道数据元素数量的时候可以考虑 Array;ArrayList 处理固定数量的基本类型数据类型时会自动装箱来减少 编码工作量，但是相对较慢。

25-2：扩容机制

1. 当前数组是由默认构造方法生成的空数组并且第一次添加数据。此时minCapacity等于默认的容量（10）那么根据下面逻辑可以看 到最后数组的容量会从0扩容成10。而后的数组扩容才是按照当前容量的1.5倍进行扩容；
2. 当前数组是由自定义初始容量构造方法创建并且指定初始容量为0。此时minCapacity等于1那么根据下面逻辑可以看到最后数组的 容量会从0变成1。这边可以看到一个严重的问题，一旦我们执行了初始容量为0，那么根据下面的算法前四次扩容每次都 +1，在 第5次添加数据进行扩容的时候才是按照当前容量的1.5倍进行扩容。
3. 当扩容量（newCapacity）大于ArrayList数组定义的最大值后会调用hugeCapacity来进行判断。如果minCapacity已经大于 Integer的最大值（溢出为负数）那么抛出OutOfMemoryError（内存溢出）否则的话根据与MAX_ARRAY_SIZE的比较情况确定是 返回Integer最大值还是MAX_ARRAY_SIZE。这边也可以看到ArrayList允许的最大容量就是Integer的最大值（-2的31次方~2的 31次方减1）。

25-3：ArrayList的add操作

不是原子操作，原因主要是elementData[size++] = e可以继续进行拆分

25-4：Arraylist初始大小以及扩容大小

ArrayList添加第一个元素时，数组的容量设置为10

4.当ArrayList数组超过当前容量时，扩容至1.5倍（遇到计算结果为小数的，向下取整），第一次扩容后，容量为15，第二次扩容至22

25-5：那如何解决ArrayList线程不安全问题呢?

1. 用Vector代替ArrayList
2. 用Collections.synchronized(new ArrayList<>())

- 因为Collections.synchronizedList封装后的list，list的所有操作方法都是带synchronized关键字的，相当于所有操作 都会进行加锁，所以使用它是线程安全的但是除迭代数组之外

3. CopyOnWriteArrayList
   - 写操作：添加元素时，不直接往当前容器添加，而是先拷贝一份数组，在新的数组中添加元素后，在将原容器的引用指向新的容 器。因为数组时用volatile关键字修饰的，所以当array重新赋值后，其他线程可以立即知道（volatile的可见性）
   - 读操作：读取数组时，读老的数组，不需要加锁。
   - 读写分离：写操作是copy了一份新的数组进行写，读操作是读老的数组，所以是读写分离。

26.java基础-集合list-vector

由于vector中Add方法加了synchronized，来保证add操作是线程安全的

26-1：Vector是保证线程安全的

27.java基础-集合list-linkedlist

28.java基础-集合set-HashSet

28-1：hashset原理

HashSet会先计算对象的hashcode值来判断对象加入的位置，同时也会与其他加入的对象的hashcode值作比较，如果没有相符 的hashcode，HashSet会假设对象没有重复出现。但是如果发现有相同hashcode值的对象，这时会调用equals()方法来检查 hashcode相等的对象是否真的相同。如果两者相同，HashSet就不会让加入操作成功。

28-2：hashSet的内存泄漏

当一个对象被存储进HashSet集合中以后，就不能修改该对象的参与计算哈希值的属性值了，否则对象修改后的哈希值与最初存储进

HashSet集合中时的哈希值就不同了，在这种情况下，即使在contains方法使用该对象的当前引用作为参数去HashSet集合中检索对 象，也将返回找不到对象的结果，这也会导致无法从HashSet集合中删除当前对象，造成内存泄露。

28-3：为什么HashSet不安全

底层add操作不保证可见性、原子性。所以不是线程安全的

28-4：如何保证线程安全

1. 使用Collections.synchronizedSet

2. 使用CopyOnWriteArraySet

29.java基础-集合set-TreeSet

29-1：TreeSet原理

30.java基础-集合set-LinkedSet

30-1：LinkedSet原理

31.java基础-集合-集合大比较（区别和使用场景）

主要从一下几方面分析

1. 线程
2. 底层
3. 时间复杂度
4. 内存
5. 其他

31-1：set和list、map的区别

1. List(对付顺序的好帮手)：List接口存储一组不唯一（可以有多个元素引用相同的对象），有序的对象
2. Set(注重独一无二的性质):不允许重复的集合。不会有多个元素引用相同的对象。
3. Map(用Key来搜索的专家):使用键值对存储。Map会维护与Key有关联的值。两个Key可以引用相同的对象，但Key不能重复，典型的Key是String类型，但也可以是任何对象。

31-2：arraylist、linkedlist区别和适用场景

1. 是否保证线程安全： ArrayList在单线程下是线程安全的，多线程下由于多个线程不断抢夺资源，所以会出现不安全 -------和 LinkedList 都是不同步的，也就是不保证线程安全；
2. 底层数据结构： Arraylist 底层使用的是 Object 数组；LinkedList 底层使用的是 双向链表 数据结构
3. 插入和删除是否受元素位置的影响： ① ArrayList 采用数组存储，所以插入和删除元素的时间复杂度受元素位置的影响。 ② LinkedList 采用链表存储，插入，删除元素时间复杂度不受元素位置的影响，如果是要在指定位置i插入和删除元素的话需要先移动到指定位置再插入。
4. 是否支持快速随机访问： LinkedList 不支持高效的随机元素访问，而 ArrayList 支持。快速随机访问就是通过元素的序号快速获取元素对象(对应于get(int index) 方法)。
5. 内存空间占用： ArrayList的空间浪费主要体现在在list列表的结尾会预留一定的容量空间，而LinkedList的空间花费则体现在它的每一个元素都需要消耗比ArrayList更多的空间（因为要存放直接后继和直接前驱以及数据）

使用场景：

当需要对数据进行对此访问的情况下选用ArrayList，当需要对数据进行多次增加删除修改时采用LinkedList。

31-3：vector、Arraylist区别和适用场景

1. 线程：Vector是多线程安全的，
2. 底层：两个都是数组实现，

3. 时间复杂度：Vector类中的方法很多有synchronized进行修饰，这样就导致了Vector在效率上无法与ArrayList相比
4. 内存：但是当空间不足的时候，两个类的增加方式是不同。vector增长率为目前数组长度的100%,而arraylist增长率为目前数组长度的41%
5. 其他：Vector可以设置增长因子，而ArrayList不可以

使用场景：

1. 安全因素
2. 在集合中使用数据量比较大的数据

31-4：HashMap、Treemap、linkedHashMap区别和适用场景

1. 线程安全：都不是线程安全的
2. 底层：TreeMap的底层是红黑树，能够按照键值进行升序排列，而HashMap与linkedHashMap是基于哈希表实现，
3. 时间复杂度：Treemap由于是红黑树，hashmap要更快一些，
4. 内存，由于Treemap使用的是红黑树，内存要大于另外两个， 又因为linkedhashmap多维护了一个双向链表，也要大约hashmap
5. 其他：hashmap排序是无序的。另外两种排序有序

使用场景：

31-5：HashTable、Hashmap区别和适用场景

1. 线程安全，hashtable更加安全
2. 底层，hashtable底层加入了锁保护
3. 时间复杂度，由于加入了锁保护，hashtable时间复杂度要低于hashmap
4. 内存，
5. 其他

使用场景：

1. 若在单线程中，我们往往会选择HashMap；
2. 而在多线程中，则会选择Hashtable。(02)，
3. 若不能插入null元素，则选择Hashtable；否则，可以选择HashMap。

31-6： ConcurrentHashMap、Hashmap区别和适用场景

1. ConcurrentHashMap对桶数组进行了分段，而HashMap并没有。
2. ConcurrentHashMap在每一个分段上都用锁进行了保护。HashMap没有锁机制。所以，前者线程安全的，后者不是线程安全的。

使用场景：

1.安全因素

31-7： Hashset、Hashmap区别和适用场景

待定

1. 接口：实现了Map接口 实现Set接口
2. 存储：存储键值对 仅存储对象
3. 添加元素：调用 put（）向map中添加元素 调用 add（） 方法向Set中添加元素
4. 计算：HashMap使用键（Key）计算Hashcode HashSet使用成员对象来计算hashcode值，对于两个对象来说hashcode可能相同，所以equals()方法用来判断对象的相等性，

使用场景：

31-8：treeset、hashset区别和适用场景

1. TreeSet 是二差树实现的,Treeset中的数据是自动排好序的，不允许放入null值 HashSet 是哈希表实现的,HashSet中的数据是无序的，可以放入null，但只能放入一个null，两者中的值都不能重复，就如数据库中唯一约束

2. HashSet要求放入的对象必须实现HashCode()方法，放入的对象，是以hashcode码作为标识的，而具有相同内容的String对象，hashcode是一样，所以放入的内容不能重复。但是同一个类的对象可以放入不同的实例

使用场景：

在我们需要排序的功能时，我们才使用TreeSet。

31-9：JAVA集合类

集合框架有Map和Collection两大类

1. Collection
    1. List
        - Arraylist： Object数组
        - Vector： Object数组
        - LinkedList： 双向链表(JDK1.6之前为循环链表， JDK1.7取消了循环)
    2. Set
        - HashSet（无序，唯一）：基于 HashMap 实现的，底层采用 HashMap 来保存元素
        - LinkedHashSet： LinkedHashSet 继承于 HashSet，并且其内部是通过 LinkedHashMap 来实现的。有点类似于我们之前说的LinkedHashMap 其内部是基于 HashMap 实现一样，不过还是有一点点区别的
        - TreeSet（有序，唯一）： 红黑树(自平衡的排序二叉树)
    3. Queue
2. Map
    1. HashMap： JDK1.8之前HashMap由数组+链表组成的，数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的（“拉链法”解决冲突）。 JDK1.8以后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间
    2. LinkedHashMap： LinkedHashMap 继承自 HashMap，所以它的底层仍然是基于拉链式散列结构即由数组和链表或红黑树组成。另外， LinkedHashMap 在上面结构的基础上，增加了一条双向链表，使得上面的结构可以保持键值对的插入顺序。同时通过对链表进行相应的操作，实现了访问顺序相关逻辑。详细可以查看：

3. Hashtable： 数组+链表组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的
4. TreeMap： 红黑树（自平衡的排序二叉树）

31-10：并发集合

1. Queue

- ConcurrentLinkedQueue
- BlockingQueue
    - ArrayBlockingQueue：基于数组、先进先出、线程安全，可实现指定时间的阻塞读写，并且容量可以限制
    - LinkedBlockingQueue：基于链表实现，读写各用一把锁，在高并发读写操作都多的情况下，性能优于ArrayBlockingQueue
- Deque

2. CopyOnWriteArrayList：线程安全且在读操作时无锁的ArrayList
3. CopyOnWriteArraySet：基于CopyOnWriteArrayList，不添加重复元素
4. ConcurrentMap：线程安全的HashMap的实现
    - ConcurrentHashMap
    - ConcurrentNavigableMap

31-11：并发集合出现的原因

书本p41 ---1

31-11：collection与collections的区别

java.util.Collection 是一个集合接口 Collections则是集合类的一个工具类/帮助类，其中提供了一系列静态方法

31-12：Collections有哪些静态方法

1. 排序(Sort)
2. 混排（Shuffling）
3. 反转(Reverse)
4. 替换所有的元素(Fill)
5. 拷贝(Copy)
6. 返回Collections中最小元素(min)
7. 返回Collections中最小元素(max)

31-13：Comparable和Comparator区别

1. 实现Comparable的类，该类就具有自身比较的功能；Comparator的实现，是一个外部比较工具器

32.java基础-设计类问题

32-1：如果想要一个key对应多个Value的话，怎么设计Map

https://blog.csdn.net/yanzhenjie1003/article/details/42541264?utm_medium=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.nonecase&depth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.nonecase

32-2：插入一万个元素之后会不会扩容，扩容扩多少

HashMap 是否扩容，由 threshold 决定，而 threshold 又由初始容量和 loadFactor 决定。

1. HashMap 构造方法传递的 initialCapacity，它实际表示 table 的容量。
    - 只是代表了 table 数组容量为 1000
2. 构造方法传递的initialCapacity，最终会被tableSizeFor()方法动态调整为2的N次幂，以方便在扩容的时候，计算数据在 newTable中的位置。

- 虽然你传入了10000，但是实际传入的是10000/loadFactor，但是呢会调整为最接近的2 的 N 次幂
    - 如：实际传入了10000/0.75=13333，最接近的是2^13=16384，那么就采用16384

3. 如果设置了table的初始容量，会在初始化 table 时，将扩容阈值 threshold 重新调整为 table.size * loadFactor。

- 那么可以储存的最大容量就是：16384*0.75=12288

32-3：创建一个对象HashMap<Integer,Integer> map=new HashMap<> 先put(10),然后get(new Long(10))结果是多少?

为空，原因是

1. hashmap在存入的时候，先对key做一遍hash，以hash值作为数组下标，如果发现下标已有值，判断存的key跟传入的key是不是 相同，如果相同覆盖，显然Integer 和 Long 肯定不是一个类型，所以 Long 123 和 Integer 123 hashmap会认为是 hash冲 突
2. hashmap 在 get的时候，也是先做hash处理，根据hash值查找对应的数组下标查找,虽然存入Integer 123 根据 Long 123 来获取返回的 是 NULL

32-4：两个线程同时操作一个集合，一个线程读，一个线程写。有可能会产生并发问题吗?

多线程编程的时候往往用到互斥锁与信号量使得线程同步，如果不按此方法进行安全编程，很有可能使得线程对境界资源访问的时候 出现竞态。注意:我们需要达到的是线 程同步，需要避免的是竞态

33.java基础-IO-各种流

33-1：为何还要有字符流

因为我们不知道编码类型很容易出现乱码，所以IO就提供了一个直接操作字符的接口

33-2：字节流和字符流区别

1. 字节流不会用到缓冲区(内存)的，而字符流在操作的时候是使用到缓冲区的
2. 字节流在操作文件时，即使不关闭资源，文件也能输出，但是如果字符流不使用close方法的话，则不会输出任何内容，只有在使 用flush方法强制进行刷新缓冲区，这时才能在不close的情况下输出内容
3. 字节流:处理字节和字节数组或二进制对象;字符流:处理字符、字符数组或字符串。

33-3：同步、异步与阻塞、非阻塞

同步：执行一个操作之后， 等待结果，然后执行其他后续的操作

异步：执行一个操作后，可以去执行其他的操作，然后等待通知再回来执行刚才没有执行完的操作

阻塞：进程给CPU传达一个任务后，一直等待CPU处理完成，然后执行后面的操作

非阻塞：进程给CPU传达一个任务后，继续处理其他的操作，隔段时间来询问之前的操作是否完成

33-4：BIO\NIO\AIO区别

BIO (Blocking I/O): 同步阻塞 I/O 模式，数据的读取写入必须阻塞在一个线程内等待其完成。

（NIO与IO区别）NIO (Non-blocking/New I/O): NIO 是一种同步非阻塞的I/O模型，

1. IO流是阻塞的，NIO流是不阻塞的。比如说，单线程中从通道读取数据到buffer，同时可以继续做别的事情，当数据读取到 buffer中后，线程再继续处理数据。Java IO的各种流是阻塞的。这意味着，当一个线程调用 read() 或 write() 时，该线程 被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了

2. IO面向流(Stream oriented)，而NIO面向缓冲区(Buffer oriented)。在面向流的I/O中,可以将数据直接写入或者将数据直接 读到Stream对象中。在从流读到缓冲区，因为Buffer是一个对象，它包含一些要写入或者要读出的数据。NIO是直接读到Buffer 中进行操作。

3. NIO通过Channel（通道）进行读写。通道是双向的，可读也可写，而流的读写是单向的。无论读写，通道只能和Buffer交互。因 为 Buffer，通道可以异步地读写。

4. NIO有选择器，而IO没有。线程之间的切换对于操作系统来说是昂贵的，因此选择器用于使用单个线程处理多个通道提高系统效率 选择器是有用的。

AIO: AIO 也就是NIO2。在引入了NIO的改进版,它是异步非阻塞的IO模型。异步IO是基于事件和回调机制实现的，也就是应用操作之 后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。

33-5：linux的5种IO模型

1. 阻塞式IO模型

2. 非阻塞IO模型

3. IO复用模型

4. 信号驱动IO模型

5. 异步IO模型

33-6：IO多路复用

如果有一百万个I/O流进来，那我们就需要开启一百万个进程一一对应处理这些I/O流，这样会造成CPU占有率会多高，这个实现方式 及其的不合理。

所以人们提出了I/O多路复用这个模型，一个线程，通过记录I/O流的状态来同时管理多个I/O，可以提高服务器的吞吐能力

33-7：三种常用的实现方式-select

a. 从用户空间将fd_set拷贝到内核空间 b. 注册回调函数 c. 调用其对应的poll方法 d. poll方法会返回一个描述读写是否就绪的mask掩码，根据这个mask掩码给fd_set赋值。 e. 如果遍历完所有的fd都没有返回一个可读写的mask掩码，就会让select的进程进入休眠模式，直到发现可读写的资源后，重 新唤醒等待队列上休眠的进程。如果在规定时间内都没有唤醒休眠进程，那么进程会被唤醒重新获得CPU，再去遍历一次fd。

f. 将fd_set从内核空间拷贝到用户空间

select函数优缺点 缺点：两次拷贝耗时、轮询所有fd耗时，支持的文件描述符太小 优点：跨平台支持

33-8：三种常用的实现方式-poll

poll函数的调用过程（与select完全一致）

优点：连接数（也就是���件描述符）没有限制（链表存储） 缺点：大量拷贝，水平触发（当报告了fd没有被处理，会重复报告，很耗性能）

33-9：三种常用的实现方式-epoll

epoll的优点

没有最大并发连接的限制 只有活跃可用的fd才会调用callback函数 内存拷贝是利用mmap()文件映射内存的方式加速与内核空间的消息传递，减少复制开销。（内核与用户空间共享一块内存） 只有存在大量的空闲连接和不活跃的连接的时候，使用epoll的效率才会比select/poll高

33-10：三种常用的实现方式区别

(1)select==>时间复杂度O(n)

只是知道有I/O事件发生了，却并不知道是哪那几个流，我们只能无差别轮询所有流，找出能读出数据，同时处理的流越多，无差别轮 询时间就越长。

(2)poll==>时间复杂度O(n)

poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态， 但是它没有最大连接数的 限制，原因是它是基于链表来存储的.

(3)epoll==>时间复杂度O(1)

epoll可以理解为event poll，不同于忙轮询和无差别轮询，epoll会把哪个流发生了怎样的I/O事件通知我们。所以我们说epoll实 际上是事件驱动（每个事件关联上fd）的，此时我们对这些流的操作都是有意义的。

例子：

1. select大妈 每一个女生下楼, select大妈都不知道这个是不是你的女神, 她需要一个一个询问, 并且select大妈能力还有限, 最多一次帮你监视1024个妹子
2. poll大妈不限制盯着女生的数量, 只要是经过宿舍楼门口的女生, 都会帮你去问是不是你女神
3. epoll大妈不限制盯着女生的数量, 并且也不需要一个一个去问. 那么如何做呢? epoll大妈会为每个进宿舍楼的女生脸上贴上一 个大字条,上面写上女生自己的名字, 只要女生下楼了, epoll大妈就知道这个是不是你女神了, 然后大妈再通知你.

34.java基础-反射

34-1：什么是反射

JAVA 反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一 个方法和属性

34-2：反射会导致性能问题呢?

反射会造成性能问题

由于反射的时候调用了native方法，可能暂时无法准确判断

可能造成的原因也是可能是

在程序运行时操作class有关，比如需要判断是否安全？是否允许这样操作？入参是否正确?

是否能够在虚拟机中找到需要反射的类？主要是这一系列判断条件导致了反射耗时；

也有可能是因为调用natvie方法，需要使用JNI接口，导致了性能问题

在四种访问方式当中，直接访问实例的方式效率最高；

其次是直接调用方法的方式；

接着是通过反射访问实例的方式；

最慢的是通过反射访问方法的方式

34-3：如果避免反射导致的性能问题?

不要过于频繁地使用反射，大量地使用反射会带来性能问题；

通过反射直接访问实例会比访问方法快很多，所以应该优先采用访问实例的方式。

34-2：获取class对象方法

1. Object类的getClass();
2. 任何数据类型（包括基本数据类型）都有一个“静态”的class属性
3. 通过Class类的静态方法：forName(常用)

34-3：Class.forName和classloader.loadClass的区别

1. 初始化不同:
    - Class.forName()会对类初始化，而loadClass()只会装载或链接。
    - foranme在类加载的时候会执行静态代码块，loadclass只有在调用newInstance方法的时候才会执行静态代码块
2. 类加载器不同:
    - Class.forName(String)方法(只有一个参数)，哪个调用了forname就用那个类加载器
    - ClassLoader.loadClass()方法是一个实例方法，调用时需要自己指定类加载器

34-4：哪些类不能反射

枚举，因为枚举类类的修饰abstract，所以没法实例化，反射也无能为力

34-5：反射优缺点

优点： 动态编译可以最大程度地支持多态，而多态最大的意义在于降低类的耦合性，因此反射的优点就很明显了:解耦以及提高代码 的 灵活性。

缺点： 1、性能瓶颈：反射相当于一系列解释操作，通知 JVM 要做的事情，性能比直接的 java 代码要慢很多。

> 2、安全问题，让我们可以动态操作改变类的属性同时也增加了类的安全隐患。

34-6：反射的应用场景

1. 使用 JDBC 连接数据库时使用 Class.forName()通过反射加载数据库的驱动程序；
2. Spring 框架的 IOC（动态加载管理 Bean）创建对象以及 AOP（动态代理）功能都和反射有联系；
3. 动态配置实例的属性；

35.java基础-注解

35-1：元注解以及分类

定义其他注解的注解 ， 共四个

1. @Target（表示该注解可以用于什么地方）
2. @Retention（表示再什么级别保存该注解信息）
3. @Documented（将此注解包含再javadoc中）
4. @Inherited（允许子类继承父类中的注解）

35-2：Java常用注解

1. @Override 表示当前方法覆盖了父类的方法
2. @Deprecation 表示方法已经过时,方法上有横线，使用时会有警告。
3. @SuppressWarnings 表示关闭一些警告信息(通知java编译器忽略特定的编译警告)

4. SafeVarargs (jdk1.7更新) 表示：专门为抑制“堆污染”警告提供的。

5. @FunctionalInterface (jdk1.8更新) 表示：用来指定某个接口必须是函数式接口，否则就会编译出错。

扩展Spring常用注解

36.java基础-泛型

36-1：什么是泛型

1. 允许在定义类和接口的时候使用类型

2. 泛型可以提高代码的复用性

36-2：编译器如何处理泛型

1. Code specialization：在实例化一个泛型类或泛型方法时都产生一份新的字节码or二进制代码。

2. Code sharing：对每个泛型类只生成唯一的一份目标代码；该泛型类的所有实例都映射到这份目标代码上，在需要的时候执行类 型检查和类型转换。

36-3：为什么Java要用这种编译器

1. C++和C#是使用Code specialization的处理机制，他有几个缺点:

    - 导致代码膨胀。
    - 在引用类型系统中，浪费空间

2. Java编译器通过Code sharing方式为每个泛型类型创建唯一的字节码表示，并且将该泛型类型的实例都映射到这个唯一的字节码 表示上。将多种泛型类形实例映射到唯一的字节码表示是通过类型擦除（type erasue）实现的。

36-4: 什么是类型擦除

Java的泛型基本上都是在编译器这个层次上实现的，在生成的字节码中是不包含泛型中的类型信息的，使用泛型的时候加上类型参 数，在编译器编译的时候会去掉，这个过程成为类型擦除。

36-5：类型擦除过程

1. 将所有的泛型参数用最顶级的父类型进行替换。

2. 移除所有的类型参数

36-6：泛型带来的问题

1. 虚拟机中没有泛型，只有普通类和普通方法,所有泛型类的类型参数在编译时都会被擦除,泛型类并没有自己独有的Class类对象。 比如并不存在List.class或是List.class，而只有List.class。

2. 创建泛型对象时需要指明类型，让编译器尽早的做参数检查

3. 不要忽略编译器的警告信息，那意味着潜在的ClassCastException等着你。

4. 静态变量是被泛型类的所有实例所共享的。

5. 泛型的类型参数不能用在Java异常处理的catch语句中。

36-7：List泛型和原始类型List之间的区别?

List