TDOKE官网4-5年java面试葵花宝典

[前言 5](#_Toc505763510)

[Java基础部分 5](#_Toc505763511)

[String类为什么是final的？ 5](#_Toc505763512)

[你看过HashMap的源码吗，简单介绍一下实现原理，底层结构？ 6](#_Toc505763513)

[说说你知道的几个Java集合类 9](#_Toc505763514)

[Java中的队列都有哪些，有什么区别。 12](#_Toc505763515)

[反射中，Class.forName和classloader的区别 14](#_Toc505763516)

[Java7、Java8的新特性 14](#_Toc505763517)

[Java静态代理和动态代理的区别，以及原理 14](#_Toc505763518)

[Java如何自定义注解 14](#_Toc505763519)

[StringBuffer和StringBuilder的源码区别 14](#_Toc505763520)

[java多线程部分 15](#_Toc505763521)

[Java创建线程之后，直接调用start()方法和run()的区别 15](#_Toc505763522)

[常用的线程池模式以及不同线程池的使用场景 15](#_Toc505763523)

[如何控制多线程的执行顺序 15](#_Toc505763524)

[newFixedThreadPool此种线程池如果线程数达到最大值后会怎么办，底层原理。 15](#_Toc505763525)

[多线程之间通信的同步问题，synchronized锁的是对象，衍伸出和synchronized相关很多的具体问题，例如同一个类不同方法都有synchronized锁，一个对象是否可以同时访问。或者一个类的static构造方法加上synchronized之后的锁的影响。 15](#_Toc505763526)

[了解可重入锁的含义，以及ReentrantLock 和synchronized的区别 15](#_Toc505763527)

[同步的数据结构，例如concurrentHashMap的源码理解以及内部实现原理，为什么他是同步的且效率高 15](#_Toc505763528)

[atomicinteger和Volatile等线程安全操作的关键字的理解和使用 15](#_Toc505763529)

[线程间通信，wait和notify 15](#_Toc505763530)

[定时线程的使用 15](#_Toc505763531)

[场景：在一个主线程中，要求有大量(很多很多)子线程执行完之后，主线程才执行完成。多种方式，考虑效率。 16](#_Toc505763532)

[线程的几种状态 16](#_Toc505763533)

[J.U.C下的常见类的使用。 ThreadPool的深入考察； BlockingQueue的使用。（take，poll的区别，put，offer的区别）；原子类的实现。 16](#_Toc505763534)

[简单介绍下多线程的情况，从建立一个线程开始。然后怎么控制同步过程，多线程常用的方法和结构 16](#_Toc505763535)

[深入底层研究volatile和synchronized的区别 16](#_Toc505763536)

[深入底层Lock接口和synchronized的区别 16](#_Toc505763537)

[分布式锁的实现原理 16](#_Toc505763538)

[java IO流部分 16](#_Toc505763539)

[讲讲IO里面的常见类，字节流、字符流、接口、实现类、方法阻塞。 16](#_Toc505763540)

[讲讲NIO。 16](#_Toc505763541)

[String 编码UTF-8 和GBK的区别? 17](#_Toc505763542)

[什么时候使用字节流、什么时候使用字符流? 17](#_Toc505763543)

[递归读取文件夹下的文件，代码怎么实现 17](#_Toc505763544)

[图解阻塞IO和非阻塞IO及多路复用机制 17](#_Toc505763545)

[设计模式部分 17](#_Toc505763546)

[性能调优部分 17](#_Toc505763547)

[JVM调优 17](#_Toc505763548)

[Java的内存模型以及GC算法 17](#_Toc505763549)

[jvm性能调优都做了什么 17](#_Toc505763550)

[介绍JVM中7个区域，然后把每个区域可能造成内存的溢出的情况说明 17](#_Toc505763551)

[介绍GC 和GC Root不正常引用 17](#_Toc505763552)

[自己从classload 加载方式，加载机制说开去，从程序运行时数据区，讲到内存分配，讲到String常量池，讲到JVM垃圾回收机制，算法，hotspot。反正就是各种扩展 17](#_Toc505763553)

[jvm 如何分配直接内存， new 对象如何不分配在堆而是栈上，常量池解析 18](#_Toc505763554)

[数组多大放在 JVM 老年代（不只是设置 PretenureSizeThreshold ，问通常多大，没做过一问便知） 18](#_Toc505763555)

[老年代中数组的访问方式 18](#_Toc505763556)

[GC 算法，永久代对象如何 GC ， GC 有环怎么处理 18](#_Toc505763557)

[谁会被 GC ，什么时候 GC 18](#_Toc505763558)

[如果想不被 GC 怎么办 18](#_Toc505763559)

[如果想在 GC 中生存 1 次怎么办 18](#_Toc505763560)

[内存溢出和内存泄漏怎么解决 18](#_Toc505763561)

[如何dump java内存数据 18](#_Toc505763562)

[Tomcat调优 18](#_Toc505763563)

[Nginx调优 18](#_Toc505763564)

[TCP IP部分 18](#_Toc505763565)

[SSL实现安全的通信，公钥私钥非对称加密 18](#_Toc505763566)

[HTTP 协议与 TCP/IP 协议的关系 19](#_Toc505763567)

[开源框架部分 19](#_Toc505763568)

[hibernate和ibatis的区别 19](#_Toc505763569)

[讲讲mybatis的连接池。 19](#_Toc505763570)

[spring框架中需要引用哪些jar包，以及这些jar包的用途 19](#_Toc505763571)

[springMVC的原理 19](#_Toc505763572)

[springMVC注解的意思 19](#_Toc505763573)

[spring中beanFactory和ApplicationContext的联系和区别 19](#_Toc505763574)

[spring注入的几种方式（循环注入） 19](#_Toc505763575)

[spring如何实现事物管理的 19](#_Toc505763576)

[spring IOC的实现 19](#_Toc505763577)

[Spring AOP的底层实现原理，动态代理如何实现 19](#_Toc505763578)

[LINUX部分 20](#_Toc505763579)

# 前言

最近在准备一些java面试资料的时候，在网上寻找到的都是一些基础的面试，在找寻无果后，决定自己编排一篇java4-5年面试的宝典，帮助大家梳理一下知识点，以更好的提高自己。相对两三年的面试题，四五年的面试更偏重如何实现，也就是底层原理，一方面是为了考验你在这四五年里是否真实的扎实的学习，另一方面是因为这些技能是高级开发者所必须掌握的，所以此版本更多是对四到五年的面试宝典的补充，在这里祝大家能早日找到心仪的公司和工作。

# Java基础部分

## String类为什么是final的？

答：首先你要理解final的用途，在分析String为什么要用final修饰，final可以修饰类，方法和变量，并且被修饰的类或方法，被final修饰的类不能被继承，即它不能拥有自己的子类，被final修饰的方法不能被重写， final修饰的变量，无论是类属性、对象属性、形参还是局部变量，都需要进行初始化操作。

在了解final的用途后，在看String为什么要被final修饰：主要是为了”安全性“和”效率“的缘故。

1-1只有当字符串是不可变的，字符串池才有可能实现。字符串池的实现可以在运行时节约很多heap空间，因为不同的字符串变量都指向池中的同一个字符串。但如果字符串是可变的，那么String interning将不能实现(译者注：[String interning](http://en.wikipedia.org/wiki/String_interning)是指对不同的字符串仅仅只保存一个，即不会保存多个相同的字符串。)，因为这样的话，如果变量改变了它的值，那么其它指向这个值的变量的值也会一起改变。

1-2如果字符串是可变的，那么会引起很严重的安全问题。譬如，数据库的用户名、密码都是以字符串的形式传入来获得数据库的连接，或者在socket编程中，主机名和端口都是以字符串的形式传入。因为字符串是不可变的，所以它的值是不可改变的，否则黑客们可以钻到空子，改变字符串指向的对象的值，造成安全漏洞。

1-3因为字符串是不可变的，所以是多线程安全的，同一个字符串实例可以被多个线程共享。这样便不用因为线程安全问题而使用同步。字符串自己便是线程安全的。

1-4因为字符串是不可变的，所以是多线程安全的，同一个字符串实例可以被多个线程共享。这样便不用因为线程安全问题而使用同步。字符串自己便是线程安全的。

1-5因为字符串是不可变的，所以在它创建的时候hashcode就被缓存了，不需要重新计算。这就使得字符串很适合作为Map中的键，字符串的处理速度要快过其它的键对象。这就是HashMap中的键往往都使用字符串。

## 你看过HashMap的源码吗，简单介绍一下实现原理，底层结构？

答：如果面试官问你这题，他是在考验你对源码的阅读能力和对HashMap的理解，如果你答出HashMap的链表结构，说明你对HashMap有基本的了解，如果你能答出jdk1.7和jdk1.8HashMap的区别，说明你比较钻研源码，如果你能说出如何优化，让自己写的HashMap更快，那就会让面试官比较满意。

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

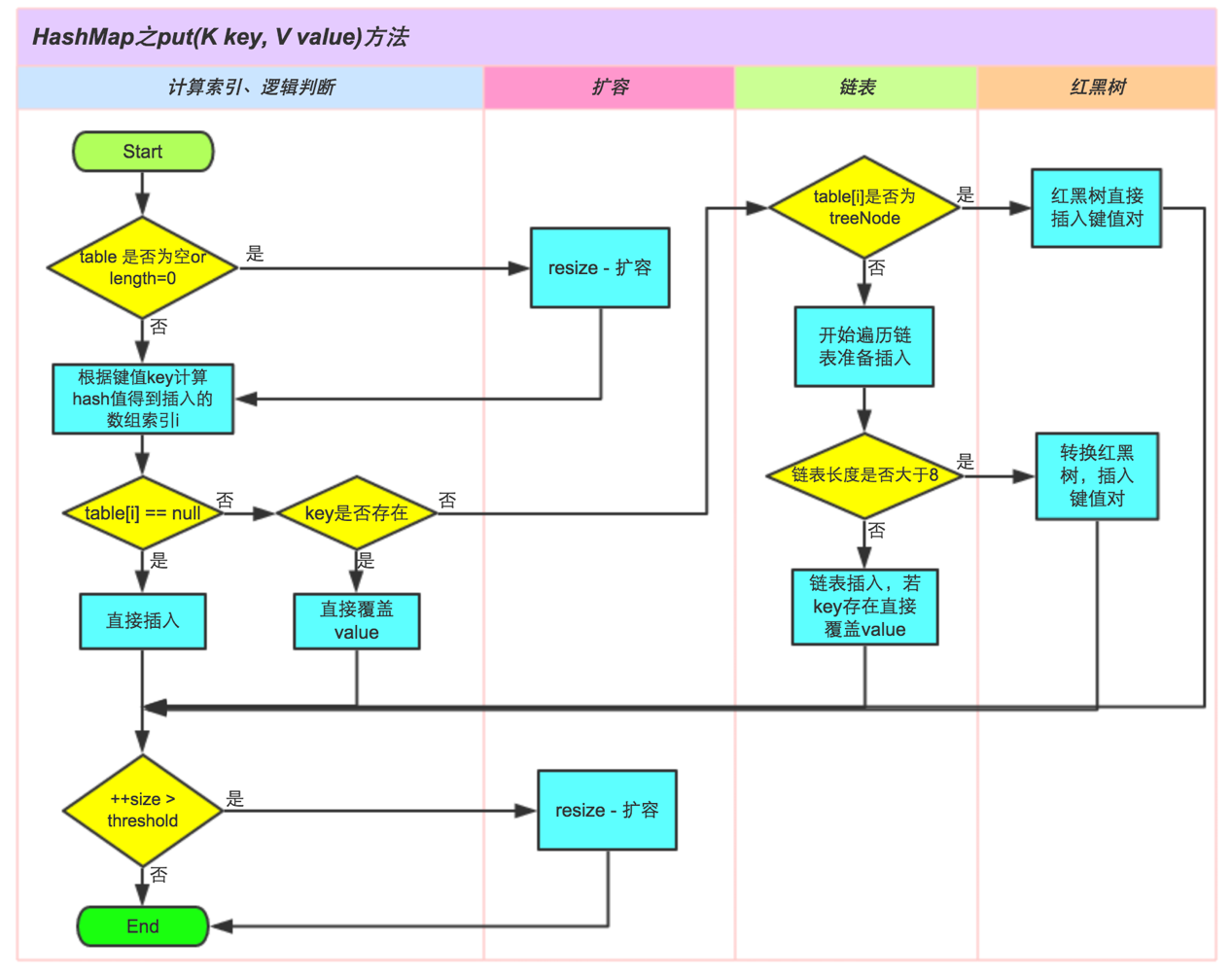
static final int hash(Object key) {  
 int h;  
 return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
}

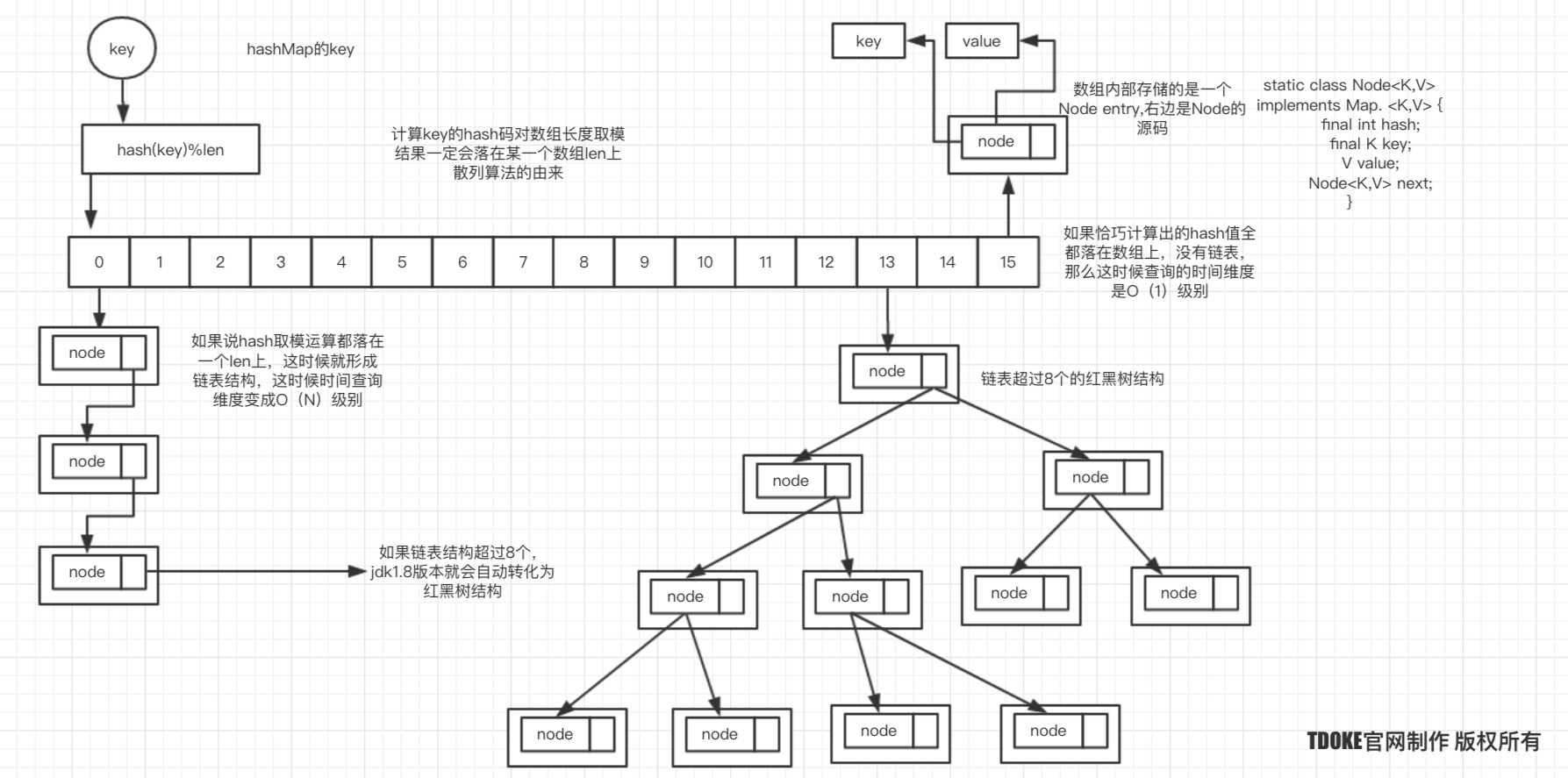
public V put(K key, V value) {  
 return putVal(hash(key), key, value, false, true);  
}

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,  
 boolean evict) {  
 Node<K,V>[] tab; Node<K,V> p; int n, i;  
 if ((tab = table) == null || (n = tab.length) == 0)  
 n = (tab = resize()).length;  
 if ((p = tab[i = (n - 1) & hash]) == null)  
 tab[i] = newNode(hash, key, value, null);  
 else {  
 Node<K,V> e; K k;  
 if (p.hash == hash &&  
 ((k = p.key) == key || (key != null && key.equals(k))))  
 e = p;  
 else if (p instanceof TreeNode)  
 e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);  
 else {  
 for (int binCount = 0; ; ++binCount) {  
 if ((e = p.next) == null) {  
 p.next = newNode(hash, key, value, null);  
 if (binCount >= TREEIFY\_THRESHOLD - 1) // -1 for 1st  
 treeifyBin(tab, hash);  
 break;  
 }  
 if (e.hash == hash &&  
 ((k = e.key) == key || (key != null && key.equals(k))))  
 break;  
 p = e;  
 }  
 }  
 if (e != null) { // existing mapping for key  
 V oldValue = e.value;  
 if (!onlyIfAbsent || oldValue == null)  
 e.value = value;  
 afterNodeAccess(e);  
 return oldValue;  
 }  
 }  
 ++modCount;  
 if (++size > threshold)  
 resize();  
 afterNodeInsertion(evict);  
 return null;  
}

上面是jdk1.8版本的HashMap的get源码，其中hash（）方法是整个HashMap的算法精髓，这就涉及到HashMap的原理。注意：1.7中HashMap采用链表数组实现，1.8之后改为前一部分使用链表数组，但是达到存放阈值后改为红黑树实现。

**HashMap的数据结构**：HashMap的底层主要是基于数组和链表来实现的，它之所以有相当快的查询速度主要是因为它是通过计算散列码来决定存储的位置。HashMap中主要是通过key的hashCode来计算hash值的，只要hashCode相同，计算出来的hash值就一样。如果存储的对象对多了，就有可能不同的对象所算出来的hash值是相同的，这就出现了所谓的hash碰撞。





如图所示，hashMap的原理基本都在图中复述，

hashmap结构：哈希表是由**数组+链表**组成的，数组的默认长度为16（可以自动变长。在构造HashMap的时候也可以指定一个长度），数组里每个元素存储的是一个链表的头结点。而组成链表的结点其实就是hashmap内部定义的一个类：Entity。Entity包含三个元素：key，value和指向下一个Entity的next。

HashMap的存取：这些元素是按照什么样的规则存储到数组中呢。一般情况是通过hash(key)%（len-1）获得，也就是元素的key的哈希值对数组长度取模得到。比如12%16=12,28%16=12,108%16=12,140%16=12。所以12、28、108以及140都存储在数组下标为12的位置。

如果两个不同对象的hashcode相同，就称为冲突。冲突会导致操作哈希表的时间开销增大，所以尽量定义好的hashCode()方法，能加快哈希表的操作。覆盖了equals方法之后一定要覆盖hashCode方法，原因很简单，比如，String a = new String(“abc”);String b = new String(“abc”);如果不覆盖hashCode的话，那么a和b的hashCode就会不同，把这两个类当做key存到HashMap中的话就 会出现问题，就会和key的唯一性相矛盾。

总结：

(1) 扩容是一个特别耗性能的操作，所以当程序员在使用HashMap的时候，估算map的大小，初始化的时候给一个大致的数值，避免map进行频繁的扩容。

(2) 负载因子是可以修改的，也可以大于1，但是建议不要轻易修改，除非情况非常特殊。

(3) HashMap是线程不安全的，不要在并发的环境中同时操作HashMap，建议使用ConcurrentHashMap。

(4) JDK1.8引入红黑树大程度优化了HashMap的性能。

## 说说你知道的几个Java集合类

答：此题的解题思路在于能够先回答出集合的UML图，从根接口到实现类，能说出这么多，面试官会感觉你已经了解集合类了，然后你再具体说每个集合类的优劣和对比，这是面试官就会计较满意了，如果你在加上设计模式和源码来讲，面试官会感觉你已经ok了，他应该也就不会过问了。



看**UNL图，看到**Collection接口是集合类的根接口，Java中没有提供这个接口的直接的实现类。但是却让其被继承产生了两个接口，就是Set，List，Queue。

**Set**接口通常表示一个集合，其中的元素不允许重复（通过hashcode和equals函数保证），常用实现类有HashSet和TreeSet，HashSet是通过Map中的HashMap实现的，而TreeSet是通过Map中的TreeMap实现的。另外，TreeSet还实现了SortedSet接口，因此是有序的集合（集合中的元素要实现Comparable接口，并覆写Compartor函数才行）。

**List**接口通常表示一个列表（数组、队列、链表、栈等），其中的元素可以重复，常用实现类为ArrayList和LinkedList，另外还有不常用的Vector。另外，LinkedList还是实现了Queue接口，因此也可以作为队列使用。

我们看到，抽象类AbstractCollection、AbstractList和AbstractSet分别实现了Collection、List和Set接口，这就是在Java集合框架中用的很多的适配器设计模式，用这些抽象类去实现接口，在抽象类中实现接口中的若干或全部方法，这样下面的一些类只需直接继承该抽象类，并实现自己需要的方法即可，而不用实现接口中的全部抽象方法。

**Map**是Java.util包中的另一个接口，它和Collection接口没有关系，是相互独立的，但是都属于集合类的一部分。Map包含了key-value对。Map不能包含重复的key，但是可以包含相同的value，同样抽象类AbstractMap通过适配器模式实现了Map接口中的大部分函数，TreeMap、HashMap、WeakHashMap等实现类都通过继承AbstractMap来实现，另外，不常用的HashTable直接实现了Map接口，它和Vector都是JDK1.0就引入的集合类。

Iterator，所有的集合类，都实现了Iterator接口，这是一个用于遍历集合中元素的接口，主要包含以下三种方法：  
1.hasNext()是否还有下一个元素。  
2.next()返回下一个元素。  
3.remove()删除当前元素。

回答到这里的时候，基本就把集合类的UML说全了，有的面试官觉得你能说全，基本就不会过问，但是有的面试官会默认不说话，这时候你可以具体深入去讲集合的实现类的作用和对比了，这样也比较深入，自己就把集合类说全了。

ArrayList以数组实现。节约空间，但数组有容量限制。超出限制时会增加50%容量，用System.arraycopy()复制到新的数组，因此最好能给出数组大小的预估值。默认第一次插入元素时创建大小为10的数组。

按数组下标访问元素–get(i)/set(i,e) 的性能很高，这是数组的基本优势。

直接在数组末尾加入元素–add(e)的性能也高，但如果按下标插入、删除元素–add(i,e), remove(i), remove(e)，则要用System.arraycopy()来移动部分受影响的元素，性能就变差了，这是基本劣势。

**LinkedList**以双向链表实现，既是List，也是Queue。链表无容量限制，但双向链表本身使用了更多空间，也需要额外的链表指针操作。

按下标访问元素–get(i)/set(i,e) 要悲剧的遍历链表将指针移动到位(如果i>数组大小的一半，会从末尾移起)。

插入、删除元素时修改前后节点的指针即可，但还是要遍历部分链表的指针才能移动到下标所指的位置，只有在链表两头的操作–add(), addFirst(),removeLast()或用iterator()上的remove()能省掉指针的移动。它是唯一一个允许放入null的Queue。，

ArryList查询快，LinkedList增删快。

Vector是线程安全的，其实有个口诀来记住集合类中哪些是线程安全的。

线程同步：喂，SHE

喂（Vector）

S（Stack）

H（hashtable）

E（enumeration）

**HashMap**

以Entry[]数组实现的哈希桶数组，用Key的哈希值取模桶数组的大小可得到数组下标。

插入元素时，如果两条Key落在同一个桶(比如哈希值1和17取模16后都属于第一个哈希桶)，Entry用一个next属性实现多个Entry以单向链表存放，后入桶的Entry将next指向桶当前的Entry。

查找哈希值为17的key时，先定位到第一个哈希桶，然后以链表遍历桶里所有元素，逐个比较其key值。

当Entry数量达到桶数量的75%时(很多文章说使用的桶数量达到了75%，但看代码不是)，会成倍扩容桶数组，并重新分配所有原来的Entry，所以这里也最好有个预估值。

取模用位运算(hash & (arrayLength-1))会比较快，所以数组的大小永远是2的N次方， 你随便给一个初始值比如17会转为32。默认第一次放入元素时的初始值是16。

iterator()时顺着哈希桶数组来遍历，看起来是个乱序。

在JDK8里，新增默认为8的閥值，当一个桶里的Entry超过閥值，就不以单向链表而以红黑树来存放以加快Key的查找速度。

**LinkedHashMap**

扩展HashMap增加双向链表的实现，号称是最占内存的数据结构。支持iterator()时按Entry的插入顺序来排序(但是更新不算， 如果设置accessOrder属性为true，则所有读写访问都算)。

实现上是在Entry上再增加属性before/after指针，插入时把自己加到Header Entry的前面去。如果所有读写访问都要排序，还要把前后Entry的before/after拼接起来以在链表中删除掉自己。

TreeMap

以红黑树实现，支持iterator()时按Key值排序，可按实现了Comparable接口的Key的升序排序，或由传入的Comparator控制。可想象的，在树上插入/删除元素的代价一定比HashMap的大。

支持SortedMap接口，如firstKey()，lastKey()取得最大最小的key，或sub(fromKey, toKey), tailMap(fromKey)剪取Map的某一段。

Set几乎都是内部用一个Map来实现, 因为Map里的KeySet就是一个Set，而value是假值，全部使用同一个Object。Set的特征也继承了那些内部Map实现的特征。

**HashSet**：内部是HashMap。

**LinkedHashSet**：内部是LinkedHashMap。

**TreeSet**：内部是TreeMap的SortedSet。

**ConcurrentSkipListSet**：内部是ConcurrentSkipListMap的并发优化的SortedSet。

**CopyOnWriteArraySet**：内部是CopyOnWriteArrayList的并发优化的Set，利用其addIfAbsent()方法实现元素去重，如前所述该方法的性能很一般。

## Java中的队列都有哪些，有什么区别。

这题主要考察工作里对队列的了解程度，首先队列是一种数据结构，他有什么好处？用途是什么，队列的分类有哪些，基本答出这些，面试官一般会放过你，觉你的知识还比较广泛，如果他深究底层代码，那你需要好好看源码，有备无患。

队列顾名思义，像一个在景点排队的队伍，先排队的人，先进入景点，也就是你有一个程序在产生内容然后入队（生产者） 另一个程序读取内容，内容出队（消费者），像ActiveMQ，RabbitMQ，ZeroMQ，Kafka，MetaMQ,RocketMQ都是队列实现的。

为什么要使用队列，通常来讲都有一个场景：当你不需要立即获得结果，但是并发量又不能无限大的时候，差不多就是你需要使用消息队列的时候（类邮件发布平台）。

队列是一种数据结构，它有两个基本操作：在队列尾部加入一个元素，和从队列头部移除一个元素，队列以一种先进先出的方式管理数据，如果你试图向一个 已经满了的阻塞队列中添加一个元素或者是从一个空的阻塞队列中移除一个元索，将导致线程阻塞，在多线程进行合作时，阻塞队列是很有用的工具，工作者线程可以定期地把中间结果存到阻塞队列中而其他工作者线程把中间结果取出并在将来修改它们。队列会自动平衡负载。如果第一个线程集运行得比第二个慢，则第二个 线程集在等待结果时就会阻塞。如果第一个线程集运行得快，那么它将等待第二个线程集赶上来。

通俗来讲，队列的好处就是它可以把不影响用户执行结果又比较耗时的任务（比如消息发送通知管理员）异步的扔给服务端去做，服务端能够多线程排队响应高并发的请求，并保证请求不丢失。

那么队列又分阻塞队列（线程安全）：

按 FIFO（先进先出）排序元素。队列的头部 是在队列中时间最长的元素。队列的尾部 是在队列中时间最短的元素。新元素插入到队列的尾部，并且队列检索操作会获得位于队列头部的元素。链接队列的吞吐量通常要高于基于数组的队列，但是在大多数并发应用程序中，其可预知的性能要低。

非阻塞队列（线程安全）：基于链接节点的、无界的、线程安全。此队列按照 FIFO（先进先出）原则对元素进行排序。队列的头部 是队列中时间最长的元素。队列的尾部 是队列中时间最短的元素。新的元素插入到队列的尾部，队列检索操作从队列头部获得元素。当许多线程共享访问一个公共 collection 时，ConcurrentLinkedQueue 是一个恰当的选择。此队列不允许 null 元素。

阻塞队列与普通队列的区别在于，当队列是空的时，从队列中获取元素的操作将会被阻塞，或者当队列是满时，往队列里添加元素的操作会被阻塞。试图从空的阻塞队列中获取元素的线程将会被阻塞，直到其他的线程往空的队列插入新的元素。同样，试图往已满的阻塞队列中添加新元素的线程同样也会被阻塞，直到其他的线程使队列重新变得空闲起来，如从队列中移除一个或者多个元素，或者完全清空队列.

Java 里面的队列：

1.ArrayDeque, （数组双端队列）   
2.PriorityQueue, （优先级队列）   
3.ConcurrentLinkedQueue, （基于链表的并发队列）   
4.DelayQueue, （延期阻塞队列）（阻塞队列实现了BlockingQueue接口）   
5.ArrayBlockingQueue, （基于数组的并发阻塞队列）   
6.LinkedBlockingQueue, （基于链表的FIFO阻塞队列）   
7.LinkedBlockingDeque, （基于链表的FIFO双端阻塞队列）   
8.PriorityBlockingQueue, （带优先级的无界阻塞队列）   
9.SynchronousQueue （并发同步阻塞队列）

阻塞队列（Blocking queue）提供了可阻塞的put和take方法，它们与可定时的offer和poll是等价的。如果Queue已经满了，put方法会被阻塞直到有空间可用；如果Queue是空的，那么take方法会被阻塞，直到有元素可用。Queue的长度可以有限，也可以无限；无限的Queue永远不会充满，所以它的put方法永远不会阻塞。

生产者-消费者设计是围绕阻塞队列展开的，生产者把数据放入队列，并使数据可用，当消费者为适当的行为做准备时会从队列中获取数据。生产者不需要知道消费者的身份或者数量，甚至根本没有消费者—它们只负责把数据放入队列。类似地，消费者也不需要知道生产者是谁，以及是谁给它们安排的工作。BlockingQueue可以使用任意数量的生产者和消费者，从而简化了生产者-消费者设计的实现。最常见的生产者-消费者设计是将线程池与工作队列相结合。

在并发编程中，一般推荐使用阻塞队列，这样实现可以尽量地避免程序出现意外的错误。阻塞队列使用最经典的场景就是socket客户端数据的读取和解析，读取数据的线程不断将数据放入队列，然后解析线程不断从队列取数据解析。还有其他类似的场景，只要符合生产者-消费者模型的都可以使用阻塞队列。

使用非阻塞队列，虽然能即时返回结果（消费结果），但必须自行编码解决返回为空的情况处理（以及消费重试等问题）。

另外他们都是线程安全的，不用考虑线程同步问题。

## 反射中，Class.forName和classloader的区别

这题主要考察java jvm的类加载机制，如果你对jvm特别熟，或者对类加载比较熟，应该能轻松答出这题。

如图所示，这是 java JVM 类装载过程，Java是如何加载到jvm里面然后被识别的呢，其实分为三步：

装载：（loading）找到class对应的字节码文件。

连接：（linking）将对应的字节码文件读入到JVM中。

初始化：（initializing）对class做相应的初始化动作。

Java中两种加载class到JVM中的方式  
Class.forName("className");其实这种方法调运的是：Class.forName(className, true,ClassLoader.getCallerClassLoader())方法  
        参数一：className，需要加载的类的名称。  
        参数二：true，是否对class进行初始化（需要initialize）  
        参数三：classLoader，对应的类加载器

ClassLoader.laodClass("className"); 其实这种方法调运的是ClassLoader.loadClass(name, false)方法  
        参数一：name,需要加载的类的名称  
        参数二：false，这个类加载以后是否需要去连接（不需要linking）

两种方式的区别：

Class.forName除了将类的.class文件加载到jvm中之外，还会对类进行解释，执行类中的static块。

而classloader只干一件事情，就是将.class文件加载到jvm中，不会执行static中的内容，只有在newInstance才会去执行static块。  
 forName("")得到的class是已经初始化完成的

 loadClass("")得到的class是还没有连接的  
  一般情况下，这两个方法效果一样，都能装载Class。但如果程序依赖于Class是否被初始化，就必须用Class.forName(name)了。最重要的区别是 forName 会初始化Class，而 loadClass 不会。因此如果要求加载时类的静态变量被初始化或静态块里的代码被执行就只能用 forName，而用 loadClass 只有等创建类实例时才会进行这些初始化。

## Java7、Java8的新特性

Java7新特性：

1. switch中可以使用字串了

2. 运用List<String> tempList = new ArrayList<>(); 即泛型实例化类型自动推断

3. 语法上支持集合，而不一定是数组；final List<Integer> piDigits = [ 1,2,3,4,5,8 ];

4. 新增一些取环境信息的工具方法

File System.getJavaIoTempDir() // IO临时文件夹

File System.getJavaHomeDir() // JRE的安装目录

File System.getUserHomeDir() // 当前用户目录

File System.getUserDir() // 启动java进程时所在的目录5

5. Boolean类型反转，空指针安全,参与位运算

Boolean Booleans.negate(Boolean booleanObj)

True => False , False => True, Null => Null

boolean Booleans.and(boolean[] array)

boolean Booleans.or(boolean[] array)

boolean Booleans.xor(boolean[] array)

boolean Booleans.and(Boolean[] array)

boolean Booleans.or(Boolean[] array)

boolean Booleans.xor(Boolean[] array)

6. .两个char间的equals   
boolean Character.equalsIgnoreCase(char ch1, char ch2)

7. map集合支持并发请求，且可以写成 Map map = {name:"xxx",age:18};  
java8新特性：

1.接口添加一个非抽象的方法实现，只需要使用 default关键字即可，这个特征又叫做扩展方法，示例如下：

interface Formula {

    double calculate(int a);

     default double sqrt(int a) {

        return Math.sqrt(a);

    }

}

2. Formula接口在拥有calculate方法之外同时还定义了sqrt方法，实现了Formula接口的子类只需要实现一个calculate方法，默认方法sqrt将在子类上可以直接使用。

Formula formula = new Formula() {

    @Override

    public double calculate(int a) {

        return sqrt(a \* 100);

    }

};

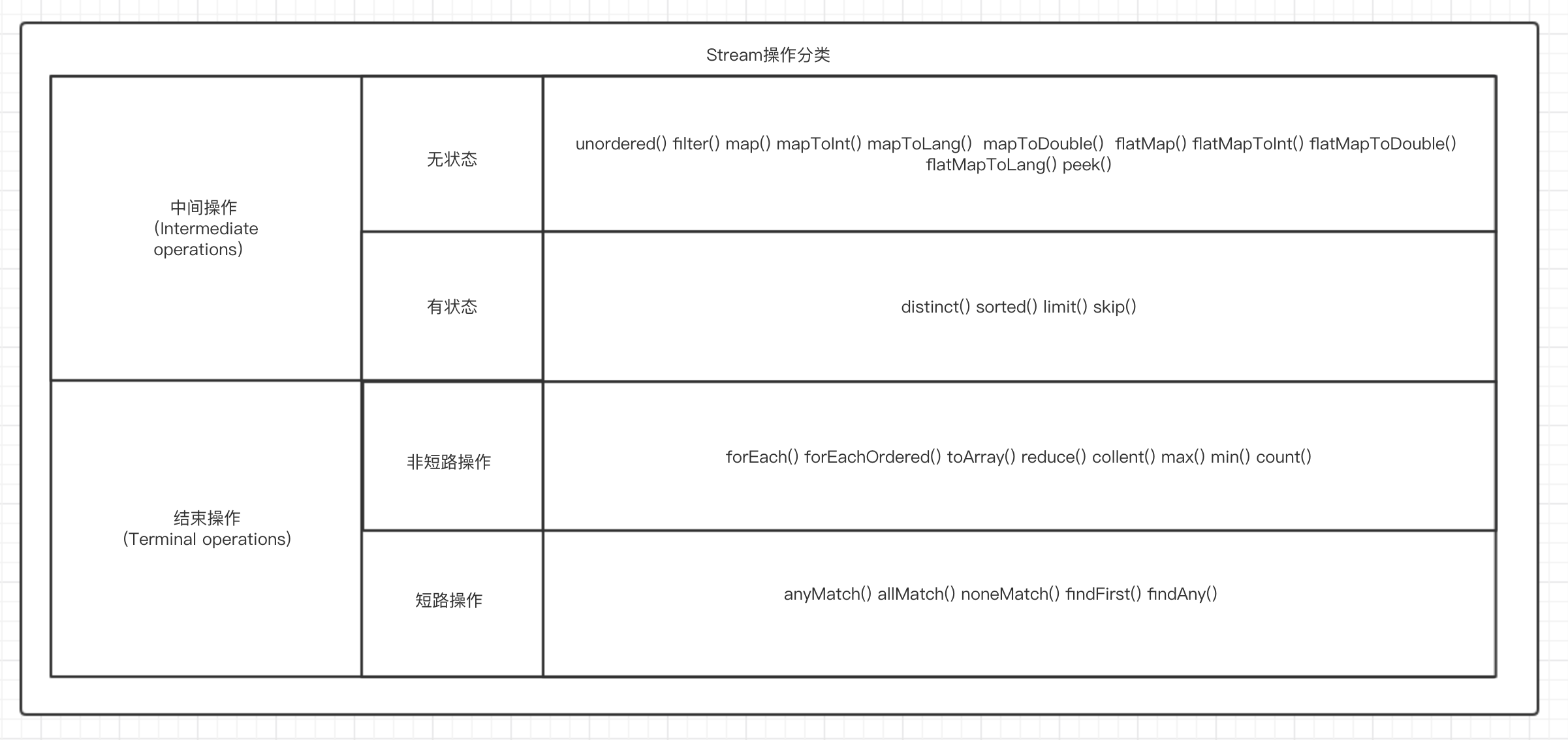
 formula.calculate(100);     // 100.0

formula.sqrt(16);

3. ambda表达式：

Collections.sort(names, (String a, String b) -> b.compareTo(a));

Stream：



java.util.Stream 表示能应用在一组元素上一次执行的操作序列。Stream 操作分为中间操作或者最终操作两种，最终操作返回一特定类型的计算结果，而中间操作返回Stream本身，这样你就可以将多个操作依次串起来。Stream 的创建需要指定一个数据源，比如 java.util.Collection的子类，List或者Set， Map不支持。Stream的操作可以串行执行或者并行执行。

Filter 过滤  
过滤通过一个predicate接口来过滤并只保留符合条件的元素，该操作属于中间操作，所以我们可以在过滤后的结果来应用其他Stream操作（比如forEach）。forEach需要一个函数来对过滤后的元素依次执行。forEach是一个最终操作，所以我们不能在forEach之后来执行其他Stream操作。

stringCollection

    .stream()

    .filter((s) -> s.startsWith("a"))

    .forEach(System.out::println);

 // "aaa2", "aaa1"

Sort 排序  
排序是一个中间操作，返回的是排序好后的Stream。如果你不指定一个自定义的Comparator则会使用默认排序。

stringCollection

    .stream()

    .sorted()

    .filter((s) -> s.startsWith("a"))

    .forEach(System.out::println);

 // "aaa1", "aaa2"

## Java静态代理和动态代理的区别，以及原理

答：讲到代理，就要知道什么是代理模式，而代理模式下，又有静态代理和动态代理，实质上这两种模式在类加载上是有本质区别的。



代理是一种常用的设计模式，其目的就是为其他对象提供一个代理以控制对某个对象的访问。代理类负责为委托类预处理消息，过滤消息并转发消息，以及进行消息被委托类执行后的后续处理。

静态代理：是由程序员创建或由特定工具自动生成源代码，再对其编译。在程序运行前，代理类的.[class文件](https://www.baidu.com/s?wd=class%E6%96%87%E4%BB%B6&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dWmWc3rHI-PAfzuAF-nyfs0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnW64PHDLPH6z)就已经存在了。

动态代理：在程序运行时，运用反射机制动态创建而成。java动态代理是利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。

 通常情况下， 静态代理中的代理类和委托类会实现同一接口或是派生自相同的父类。相比于静态代理， 动态代理的优势在于可以很方便的对代理类的函数进行统一的处理，而不用修改每个代理类的函数。

## Java如何自定义注解

## StringBuffer和StringBuilder的源码区别

# java多线程部分

## Java创建线程之后，直接调用start()方法和run()的区别

## 常用的线程池模式以及不同线程池的使用场景

## 如何控制多线程的执行顺序

## newFixedThreadPool此种线程池如果线程数达到最大值后会怎么办，底层原理。

## 多线程之间通信的同步问题，synchronized锁的是对象，衍伸出和synchronized相关很多的具体问题，例如同一个类不同方法都有synchronized锁，一个对象是否可以同时访问。或者一个类的static构造方法加上synchronized之后的锁的影响。

## 了解可重入锁的含义，以及ReentrantLock 和synchronized的区别

## 同步的数据结构，例如concurrentHashMap的源码理解以及内部实现原理，为什么他是同步的且效率高

## atomicinteger和Volatile等线程安全操作的关键字的理解和使用

## 线程间通信，wait和notify

## 定时线程的使用

## 场景：在一个主线程中，要求有大量(很多很多)子线程执行完之后，主线程才执行完成。多种方式，考虑效率。

## 线程的几种状态

## J.U.C下的常见类的使用。 ThreadPool的深入考察； BlockingQueue的使用。（take，poll的区别，put，offer的区别）；原子类的实现。

## 简单介绍下多线程的情况，从建立一个线程开始。然后怎么控制同步过程，多线程常用的方法和结构

## 深入底层研究volatile和synchronized的区别

# 深入底层Lock接口和synchronized的区别

## 分布式锁的实现原理

# java IO流部分

## 讲讲IO里面的常见类，字节流、字符流、接口、实现类、方法阻塞。

## 讲讲NIO。

## String 编码UTF-8 和GBK的区别?

## 什么时候使用字节流、什么时候使用字符流?

## 递归读取文件夹下的文件，代码怎么实现

## 图解阻塞IO和非阻塞IO及多路复用机制

## 设计模式部分

# 性能调优部分

## JVM调优

### Java的内存模型以及GC算法

### jvm性能调优都做了什么

### 介绍JVM中7个区域，然后把每个区域可能造成内存的溢出的情况说明

### 介绍GC 和GC Root不正常引用

### 自己从classload 加载方式，加载机制说开去，从程序运行时数据区，讲到内存分配，讲到String常量池，讲到JVM垃圾回收机制，算法，hotspot。反正就是各种扩展

### jvm 如何分配直接内存， new 对象如何不分配在堆而是栈上，常量池解析

### 数组多大放在 JVM 老年代（不只是设置 PretenureSizeThreshold ，问通常多大，没做过一问便知）

### 老年代中数组的访问方式

### GC 算法，永久代对象如何 GC ， GC 有环怎么处理

### 谁会被 GC ，什么时候 GC

### 如果想不被 GC 怎么办

### 如果想在 GC 中生存 1 次怎么办

### 内存溢出和内存泄漏怎么解决

### 如何dump java内存数据

## Tomcat调优

## Nginx调优

# TCP IP部分

## SSL实现安全的通信，公钥私钥非对称加密

## HTTP 协议与 TCP/IP 协议的关系

# 开源框架部分

## hibernate和ibatis的区别

## 讲讲mybatis的连接池。

## spring框架中需要引用哪些jar包，以及这些jar包的用途

## springMVC的原理

## springMVC注解的意思

## spring中beanFactory和ApplicationContext的联系和区别

## spring注入的几种方式（循环注入）

## spring如何实现事物管理的

## spring IOC的实现

## Spring AOP的底层实现原理，动态代理如何实现

# LINUX部分