**1. 九种基本数据类型的大小，以及他们的封装类。**

（1）九种基本数据类型和封装类

|  |  |  |  |
| --- | --- | --- | --- |
| 基本类型 | 大小(字节) | 默认值 | 封装类 |
| byte | 1 | (byte)0 | Byte |
| short | 2 | (short)0 | Short |
| int | 4 | 0 | Integer |
| long | 8 | 0L | Long |
| float | 4 | 0.0f | Float |
| double | 8 | 0.0d | Double |
| boolean | - | false | Boolean |
| char | 2 | \u0000(null) | Character |
| void | - | - | Void |

（2）自动装箱和自动拆箱

**什么是自动装箱拆箱**

基本数据类型的自动装箱(autoboxing)、拆箱(unboxing)是自J2SE 5.0开始提供的功能。

一般我们要创建一个类的对象实例的时候，我们会这样：

 Class a = new Class(parameter);

 当我们创建一个Integer对象时，却可以这样：

 Integer i = 100; (注意：不是 int i = 100; )

实际上，执行上面那句代码的时候，系统为我们执行了：Integer i = Integer.valueOf(100);

此即基本数据类型的自动装箱功能。

**基本数据类型与对象的差别**

基本数据类型不是对象，也就是使用int、double、boolean等定义的变量、常量。

基本数据类型没有可调用的方法。

eg：  int t = 1；     t.  后面是没有方法滴。

 Integer t =1； t.  后面就有很多方法可让你调用了。

**什么时候自动装箱**

例如：Integer i = 100;

相当于编译器自动为您作以下的语法编译：Integer i = Integer.valueOf(100);

**什么时候自动拆箱**

自动拆箱(unboxing)，也就是将对象中的基本数据从对象中自动取出。如下可实现自动拆箱：

1 Integer i = 10; //装箱   
2  int t = i; //拆箱，实际上执行了 int t = i.intValue();

　　在进行运算时，也可以进行拆箱。

1 Integer i = 10;   
2 System.out.println(i++);

**Integer的自动装箱**

//在-128~127 之外的数  
 Integer i1 =200;   
 Integer i2 =200;   
 System.out.println("i1==i2: "+(i1==i2));   
 // 在-128~127 之内的数  
 Integer i3 =100;   
 Integer i4 =100;   
 System.out.println("i3==i4: "+(i3==i4));

**输出的结果是：**

i1==i2: false  
 i3==i4: true

说明：

equals() 比较的是两个对象的值（内容）是否相同。

"==" 比较的是两个对象的引用（内存地址）是否相同，也用来比较两个基本数据类型的变量值是否相等。

前面说过，int 的自动装箱，是系统执行了 Integer.valueOf(int i)，先看看Integer.java的源码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public static Integer valueOf(int i) {      if(i >= -128 && i <= IntegerCache.high)　　// 没有设置的话，IngegerCache.high 默认是127          return IntegerCache.cache[i + 128];      else          return new Integer(i);  } |

对于–128到127（默认是127）之间的值，Integer.valueOf(int i) 返回的是**缓存的Integer对象！！！**（并不是新建对象）

**所以范例中，i3 与 i4实际上是指向同一个对象。**

而其他值，执行Integer.valueOf(int i) 返回的是一个新建的 Integer对象，所以范例中，i1与i2 指向的是不同的对象。

当然，当不使用自动装箱功能的时候，情况与普通类对象一样，请看下例：

1 Integer i3 =new Integer(100);   
2 Integer i4 =new Integer(100);   
3 System.out.println("i3==i4: "+(i3==i4));//显示false

**2. Switch能否用string做参数？**

答案：在 [Java](https://www.baidu.com/s?wd=Java&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YzmWm3Pjn3nHf3PynLm1RY0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnW0dPjD3rjbLrjT3n1TkPjb3Ps) 7之前，switch 只能支持 byte、short、char、int或者其对应的封装类以及 Enum 类型。在 [Java](https://www.baidu.com/s?wd=Java&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YzmWm3Pjn3nHf3PynLm1RY0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnW0dPjD3rjbLrjT3n1TkPjb3Ps) 7中，String支持被加上了。

**3. equals与==的区别。**

答案：equals是逻辑相等，==完全是对象是否是同一个（地址相等）。

**4. Object有哪些公用方法？**

**（1）clone**

保护方法，实现对象的浅复制，只有实现了Cloneable接口才可以调用该方法，否则抛出CloneNotSupportedException异常

**（2）equals**

在Object中与==是一样的，子类一般需要重写该方法

**（3）hashCode**

该方法用于哈希查找，重写了equals方法一般都要重写hashCode方法。这个方法在一些具有哈希功能的Collection中用到

**（4）getClass**

final方法，获得运行时类型

**（5）wait**

使当前线程等待该对象的锁，当前线程必须是该对象的拥有者，也就是具有该对象的锁。wait()方法一直等待，直到获得锁或者被中断。wait(long timeout)设定一个超时间隔，如果在规定时间内没有获得锁就返回。

调用该方法后当前线程进入睡眠状态，直到以下事件发生：   
1. 其他线程调用了该对象的notify方法   
2. 其他线程调用了该对象的notifyAll方法   
3. 其他线程调用了interrupt中断该线程   
4. 时间间隔到了   
此时该线程就可以被调度了，如果是被中断的话就抛出一个InterruptedException异常

**（6）notify**

唤醒在该对象上等待的某个线程

**（7）notifyAll**

唤醒在该对象上等待的所有线程

**（8）toString**

转换成字符串，一般子类都有重写，否则打印句柄

**5. Java的四种引用，强弱软虚，用到的场景。**

**（1）强引用（StrongReference）**  
强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。

如下：

|  |  |
| --- | --- |
| 1 | Object o=new Object();   //  强引用 |

当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。如果不使用时，要通过如下方式来弱化引用，如下：

|  |  |
| --- | --- |
| 1 | o=null;     // 帮助垃圾收集器回收此对象 |

  显式地设置o为null，或超出对象的生命周期范围，则gc认为该对象不存在引用，这时就可以回收这个对象。具体什么时候收集这要取决于gc的算法。

举例：

|  |  |
| --- | --- |
| 1  2  3  4 | public void test(){      Object o=new Object();      // 省略其他操作  } |

在一个方法的内部有一个强引用，这个引用保存在栈中，而真正的引用内容（Object）保存在堆中。当这个方法运行完成后就会退出方法栈，则引用内容的引用不存在，这个Object会被回收。

但是如果这个o是全局的变量时，就需要在不用这个对象时赋值为null，因为强引用不会被垃圾回收。

强引用在实际中有非常重要的用处，举个ArrayList的实现源代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | private transient Object[] elementData;  public void clear() {          modCount++;          // Let gc do its work          for (int i = 0; i < size; i++)              elementData[i] = null;          size = 0;  } |

在ArrayList类中定义了一个私有的变量elementData数组，在调用方法清空数组时可以看到为每个数组内容赋值为null。不同于elementData=null，强引用仍然存在，避免在后续调用 add()等方法添加元素时进行重新的内存分配。**使用如clear()方法中释放内存的方法对数组中存放的引用类型特别适用，这样就可以及时释放内存。**

**（2）软引用（SoftReference）**

如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

|  |  |
| --- | --- |
| 1  2 | String str=new String("abc");                                     // 强引用   SoftReference<String> softRef=new SoftReference<String>(str);     // 软引用 |

  当内存不足时，等价于：

|  |  |
| --- | --- |
| 1  2  3  4 | If(JVM.内存不足()) {     str = null;  // 转换为软引用     System.gc(); // 垃圾回收器进行回收  } |

软引用在实际中有重要的应用，例如浏览器的后退按钮。按后退时，这个后退时显示的网页内容是重新进行请求还是从缓存中取出呢？这就要看具体的实现策略了。

（1）如果一个网页在浏览结束时就进行内容的回收，则按后退查看前面浏览过的页面时，需要重新构建

（2）如果将浏览过的网页存储到内存中会造成内存的大量浪费，甚至会造成内存溢出

这时候就可以使用软引用

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Browser prev = new Browser();               // 获取页面进行浏览  SoftReference sr = new SoftReference(prev); // 浏览完毕后置为软引用  if(sr.get()!=null){      rev = (Browser) sr.get();           // 还没有被回收器回收，直接获取  }else{      prev = new Browser();               // 由于内存吃紧，所以对软引用的对象回收了      sr = new SoftReference(prev);       // 重新构建  } |

这样就很好的解决了实际的问题。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。

**3、弱引用**

如果一个对象只具有弱引用，那就类似于可有可无的生活用品。弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个**优先级很低的线程**， 因此不一定会很快发现那些只具有弱引用的对象。弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。**当你想引用一个对象，但是这个对象有自己的生命周期，你不想介入这个对象的生命周期，这时候你就是用弱引用。这个引用不会在对象的垃圾回收判断中产生任何附加的影响。比如说Thread中保存的ThreadLocal的全局映射，因为我们的Thread不想在ThreadLocal生命周期结束后还对其造成影响，所以应该使用弱引用，这个和缓存没有关系，只是为了防止内存泄漏所做的特殊操作。**

**4、幽灵引用(虚引用)**

**虚引用主要用来跟踪对象被垃圾回收器回收的活动。**虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列 （ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存后，把这个虚引用加入到与之关联的引用队列中。**程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否被垃圾回收。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存回收后采取必要的行动。**由于Object.finalize()方法的不安全性、低效性，常常**使用虚引用完成对象回收后的资源释放工作**。当你创建一个虚引用时要传入一个引用队列，如果引用队列中出现了你的虚引用，说明它已经被回收，那么你可以在其中做一些相关操作，主要是实现细粒度的内存控制。比如**监视缓存，当缓存被回收后才申请新的缓存区。**

**6. hashcode的作用。**

答案：hashCode用于返回对象的散列值，用于在散列函数中确定放置的桶的位置。

1、hashCode的存在主要是用于查找的快捷性，如Hashtable，HashMap等，hashCode是用来在散列存储结构中确定对象的存储地址的；

2、如果两个对象相同，就是适用于equals(java.lang.Object) 方法，那么这两个对象的hashCode一定要相同；

3、如果对象的equals方法被重写，那么对象的hashCode也尽量重写，并且产生hashCode使用的对象，一定要和equals方法中使用的一致，否则就会违反上面提到的第2点；

4、两个对象的hashCode相同，并不一定表示两个对象就相同，也就是不一定适用于equals(java.lang.Object) 方法，只能够说明这两个对象在散列存储结构中，如Hashtable，他们**“存放在同一个篮子里”**。

**7. ArrayList、LinkedList、Vector的区别。**

Arraylist和Vector是采用数组方式存储数据，此数组元素数大于实际存储的数据以便增加插入元素，都允许直接序号索引元素，但是插入数据要涉及到数组元素移动等内存操作，所以插入数据慢，查找有下标，所以查询数据快，**Vector由于使用了synchronized方法-线程安全**，所以性能上比ArrayList要差，LinkedList使用双向链表实现存储，按序号索引数据需要进行向前或向后遍历，但是插入数据时只需要记录本项前后项即可，插入数据较快。

**8. String、StringBuffer与StringBuilder的区别。**

答案:

**String 字符串常量  
StringBuffer 字符串变量（线程安全）  
StringBuilder 字符串变量（非线程安全）**

简要的说， String 类型和 StringBuffer 类型的主要性能区别其实在于 String 是不可变的对象, 因此在每次对 String 类型进行改变的时候其实都等同于生成了一个新的 String 对象，然后将指针指向新的 String 对象，所以经常改变内容的字符串最好不要用 String ，因为每次生成对象都会对系统性能产生影响，特别当内存中无引用对象多了以后， JVM 的 GC 就会开始工作，那速度是一定会相当慢的。

而如果是使用 StringBuffer 类则结果就不一样了，每次结果都会对 StringBuffer 对象本身进行操作，而不是生成新的对象，再改变对象引用。所以在一般情况下我们推荐使用 StringBuffer ，特别是字符串对象经常改变的情况下。

1. String s1 = "aaaaa";
2. String s2 = "bbbbb";
3. String r = **null**;
4. **int** i = 3694;
5. r = s1 + i + s2;

通过查看字节码可以发现，JVM内部使用的是创建了一个StringBuilder完成拼接工作

**StringBuffer**  
Java.lang.StringBuffer线程安全的可变字符序列。一个类似于 String 的字符串缓冲区，但不能修改。虽然在任意时间点上它都包含某种特定的字符序列，但通过某些方法调用可以改变该序列的长度和内容。  
可将字符串缓冲区安全地用于多个线程。可以在必要时对这些方法进行同步，因此任意特定实例上的所有操作就好像是以串行顺序发生的，该顺序与所涉及的每个线程进行的方法调用顺序一致。  
StringBuffer 上的主要操作是 append 和 insert 方法，可重载这些方法，以接受任意类型的数据。每个方法都能有效地将给定的数据转换成字符串，然后将该字符串的字符追加或插入到字符串缓冲区中。append 方法始终将这些字符添加到缓冲区的末端；而 insert 方法则在指定的点添加字符。  
例如，如果 z 引用一个当前内容是“start”的字符串缓冲区对象，则此方法调用 z.append("le") 会使字符串缓冲区包含“startle”，而 z.insert(4, "le") 将更改字符串缓冲区，使之包含“starlet”。  
在大部分情况下 StringBuilder > StringBuffer

**java.lang.StringBuilder**  
java.lang.StringBuilder一个可变的字符序列是5.0新增的。此类提供一个与 StringBuffer 兼容的 API，但不保证同步。该类被设计用作 StringBuffer 的一个简易替换，用在字符串缓冲区被单个线程使用的时候（这种情况很普遍）。如果可能，建议优先采用该类，因为在大多数实现中，它比 StringBuffer 要快。两者的方法基本相同。

9. Map、Set、List、Queue、Stack的特点与用法。

**10. HashMap和HashTable的区别。**

答案：Hashtable和HashMap类有三个重要的不同之处。

1、第一个不同主要是历史原因。**Hashtable是基于陈旧的Dictionary类的，HashMap是Java 1.2引进的**[**Map接口**](https://www.baidu.com/s?wd=Map%E6%8E%A5%E5%8F%A3&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y1mH6YnHRLujRsnjuhmHcv0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1RsPjDsPjRv)**的一个实现。**

2、也许最重要的不同是Hashtable的方法是同步的，**但是这也是HashTable效率低下的原因，任何方法都是同步的**，而HashMap的方法不是。这就意味着，虽然你可以不用采取任何特殊的行为就可以在一个多线程的应用程序中用一个Hashtable，但你必须同样地为一个HashMap提供外同步。一个方便的方法就是利用Collections类的静态的synchronizedMap()方法，它创建一个[线程安全](https://www.baidu.com/s?wd=%E7%BA%BF%E7%A8%8B%E5%AE%89%E5%85%A8&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y1mH6YnHRLujRsnjuhmHcv0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1RsPjDsPjRv)的Map对象，并把它作为一个封装的对象来返回。这个对象的方法可以让你同步访问潜在的HashMap。这么做的结果就是当你不需要同步时，你不能切断Hashtable中的同步（比如在一个单线程的应用程序中），而且同步增加了很多处理费用。

3、第三点不同是，**只有HashMap可以让你将空值作为一个表的条目的key或value。**HashMap中只有一条记录可以是一个空的key，但任意数量的条目可以是空的value。这就是说，如果在表中没有发现搜索键，或者如果发现了搜索键，但它是一个空的值，那么get()将返回null。如果有必要，用containKey()方法来区别这两种情况。

一些资料建议，当需要同步时，用Hashtable，反之用HashMap。但是，因为在需要时，HashMap可以被同步，HashMap的功能比Hashtable的功能更多，而且它不是基于一个陈旧的类的，所以有人认为，在各种情况下，HashMap都优先于Hashtable。

**11. HashMap和ConcurrentHashMap的区别，HashMap的底层源码。**

**（1）HashMap的源码：**

1、HashMap中的hash函数实现：

详见：https://www.zhihu.com/question/20733617

2、HashMap源码解读

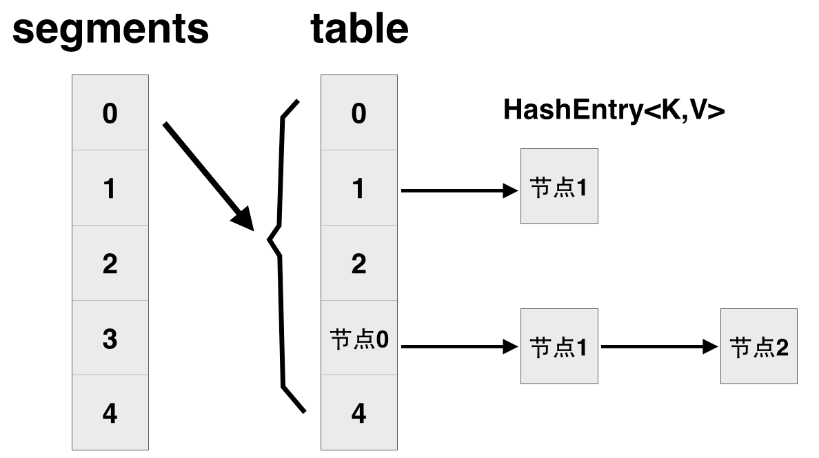
详见：http://blog.csdn.net/ll530304349/article/details/53056346

**（2）ConcurrentHashMap的源码：**

**1、JDK1.7版本的实现**

ConcurrentHashMap的**锁分段技术**：假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术。首先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。

ConcurrentHashMap不允许Key或者Value的值为NULL

[](https://github.com/FrankLee96/interview/blob/master/java/ConcurrentHashMap.png)

**第一：Segment类**

**Put**

将一个HashEntry放入到该Segment中，**使用自旋机制，减少了加锁的可能性。**

final V put(K key, int hash, V value, boolean onlyIfAbsent) {

HashEntry<K,V> node = tryLock() ? null :

scanAndLockForPut(key, hash, value); //如果加锁失败，则调用该方法

V oldValue;

try {

HashEntry<K,V>[] tab = table;

int index = (tab.length - 1) & hash; //同hashMap相同的哈希定位方式

HashEntry<K,V> first = entryAt(tab, index);

for (HashEntry<K,V> e = first;;) {

if (e != null) {

//若不为null，则持续查找，知道找到key和hash值相同的节点，将其value更新

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 { //若头结点为null

if (node != null) //在遍历key对应节点链时没有找到相应的节点

node.setNext(first);

//当前修改并不需要让其他线程知道，在锁退出时修改自然会

//更新到内存中,可提升性能

else

node = new HashEntry<K,V>(hash, key, value, first);

int c = count + 1;

if (c > threshold && tab.length < MAXIMUM\_CAPACITY)

rehash(node); //如果超过阈值，则进行rehash操作

else

setEntryAt(tab, index, node);

++modCount;

count = c;

oldValue = null;

break;

}

}

} finally {

unlock();

}

return oldValue;

}

**scanAndLockForPut**

该操作持续查找key对应的节点链中是否已存在该节点，如果没有找到已存在的节点，则预创建一个新节点，**并且尝试n次，直到尝试次数超出限制**，才真正进入等待状态，即所谓的 **自旋等待**。

private HashEntry<K,V> scanAndLockForPut(K key, int hash, V value) {

//根据hash值找到segment中的HashEntry节点

HashEntry<K,V> first = entryForHash(this, hash); //首先获取头结点

HashEntry<K,V> e = first;

HashEntry<K,V> node = null;

int retries = -1; // negative while locating node

while (!tryLock()) { //持续遍历该哈希链

HashEntry<K,V> f; // to recheck first below

if (retries < 0) {

if (e == null) {

if (node == null) //若不存在要插入的节点，则创建一个新的节点

node = new HashEntry<K,V>(hash, key, value, null);

retries = 0;

}

else if (key.equals(e.key))

retries = 0;

else

e = e.next;

}

else if (++retries > MAX\_SCAN\_RETRIES) {

//尝试次数超出限制，则进行自旋等待

lock();

break;

}

/\*当在自旋过程中发现节点链的链头发生了变化，则更新节点链的链头，

并重置retries值为－1，重新为尝试获取锁而自旋遍历\*/

else if ((retries & 1) == 0 &&

(f = entryForHash(this, hash)) != first) {

e = first = f; // re-traverse if entry changed

retries = -1;

}

}

return node;

}

**remove**

用于移除某个节点，返回移除的节点值。

final V remove(Object key, int hash, Object value) {

if (!tryLock())

scanAndLock(key, hash);

V oldValue = null;

try {

HashEntry<K,V>[] tab = table;

int index = (tab.length - 1) & hash;

//根据这种哈希定位方式来定位对应的HashEntry

HashEntry<K,V> e = entryAt(tab, index);

HashEntry<K,V> pred = null;

while (e != null) {

K k;

HashEntry<K,V> next = e.next;

if ((k = e.key) == key ||

(e.hash == hash && key.equals(k))) {

V v = e.value;

if (value == null || value == v || value.equals(v)) {

if (pred == null)

setEntryAt(tab, index, next);

else

pred.setNext(next);

++modCount;

--count;

oldValue = v;

}

break;

}

pred = e;

e = next;

}

} finally {

unlock();

}

return oldValue;

}

**Clear**

要首先对整个segment加锁，然后将每一个HashEntry都设置为null。

final void clear() {

lock();

try {

HashEntry<K,V>[] tab = table;

for (int i = 0; i < tab.length ; i++)

setEntryAt(tab, i, null);

++modCount;

count = 0;

} finally {

unlock();

}

}

**Put**

public V put(K key, V value) {

Segment<K,V> s;

if (value == null)

throw new NullPointerException();

int hash = hash(key); //求出key的hash值

int j = (hash >>> segmentShift) & segmentMask;

//求出key在segments数组中的哪一个segment中

if ((s = (Segment<K,V>)UNSAFE.getObject

(segments, (j << SSHIFT) + SBASE)) == null)

s = ensureSegment(j); //使用unsafe操作取出该segment

return s.put(key, hash, value, false); //向segment中put元素

}

**Get**

public V get(Object key) {

Segment<K,V> s;

HashEntry<K,V>[] tab;

int h = hash(key); //找出对应的segment的位置

long u = (((h >>> segmentShift) & segmentMask) << SSHIFT) + SBASE;

if ((s = (Segment<K,V>)UNSAFE.getObjectVolatile(segments, u)) != null &&

(tab = s.table) != null) { //使用Unsafe获取对应的Segmen

for (HashEntry<K,V> e = (HashEntry<K,V>) UNSAFE.getObjectVolatile

(tab, ((long)(((tab.length - 1) & h)) << TSHIFT) + TBASE);

e != null; e = e.next) { //找出对应的HashEntry，从头开始遍历

K k;

if ((k = e.key) == key || (e.hash == h && key.equals(k)))

return e.value;

}

}

return null;

}

**Size**

求出所有的HashEntry的数目，**先尝试的遍历查找、计算2遍**，如果两遍遍历过程中整个Map没有发生修改（即两次所有Segment实例中modCount值的和一致），则可以认为整个查找、计算过程中Map没有发生改变。否则,需要对所有segment实例进行加锁、计算、解锁，然后返回。

public int size() {

final Segment<K,V>[] segments = this.segments;

int size;

boolean overflow; // true if size overflows 32 bits

long sum; // sum of modCounts

long last = 0L; // previous sum

int retries = -1; // first iteration isn't retry

try {

for (;;) {

if (retries++ == RETRIES\_BEFORE\_LOCK) {

for (int j = 0; j < segments.length; ++j)

ensureSegment(j).lock(); // force creation

}

sum = 0L;

size = 0;

overflow = false;

for (int j = 0; j < segments.length; ++j) {

Segment<K,V> seg = segmentAt(segments, j);

if (seg != null) {

sum += seg.modCount;

int c = seg.count;

if (c < 0 || (size += c) < 0)

overflow = true;

}

}

if (sum == last)

break;

last = sum;

}

} finally {

if (retries > RETRIES\_BEFORE\_LOCK) {

for (int j = 0; j < segments.length; ++j)

segmentAt(segments, j).unlock();

}

}

return overflow ? Integer.MAX\_VALUE : size;

}

**（2）JDK1.8实现**

在JDK1.8中对ConcurrentHashmap做了两个改进：

* 取消segments字段，直接采用transient volatile HashEntry<K,V>[] table保存数据，**采用table数组元素作为锁，从而实现了对每一行数据进行加锁，进一步减少并发冲突的概率**。
* 将原先 **table数组＋单向链表** 的数据结构，变更为 **table数组＋单向链表＋红黑树** 的结构。对于hash表来说，最核心的能力在于将key hash之后能均匀的分布在数组中。如果hash之后散列的很均匀，那么table数组中的每个队列长度主要为0或者1。但实际情况并非总是如此理想，虽然ConcurrentHashMap类默认的加载因子为0.75，但是在数据量过大或者运气不佳的情况下，还是会存在一些队列长度过长的情况，如果还是采用单向列表方式，**那么查询某个节点的时间复杂度为O(n)；因此，对于个数超过8(默认值)的列表，jdk1.8中采用了红黑树的结构，那么查询的时间复杂度可以降低到O(logN)，**可以改进性能。

**12. TreeMap、HashMap、LinkedHashMap的区别。**

Map主要用于存储健值对，根据键得到值，因此不允许键重复(重复了覆盖了),但允许值

重复。Hashmap 是一个最常用的Map,它根据键的HashCode 值存储数据,根据键可以直接获取它的值，具有很快的访问速度，遍历时，取得数据的顺序是完全随机的。HashMap最多只允许一条记录的键为Null;允许多条记录的值为 Null;HashMap不支持线程的同步，即任一时刻可以有多个线程同时写HashMap;可能会导致数据的不一致。如果需要同步，可以用 Collections的synchronizedMap方法使HashMap具有同步的能力，或者使用ConcurrentHashMap。

* HashMap：线程不同步。根据key的hashcode进行存储，内部使用静态内部类Node的数组进行存储，默认初始大小为16，每次扩大一倍。当发生Hash冲突时，采用拉链法（链表）。**可以接受为null的键值(key)和值(value)**。JDK 1.8中：当单个桶中元素个数大于等于8时，链表实现改为红黑树实现；当元素个数小于6时，变回链表实现。由此来防止hashCode攻击。
* LinkedHashMap：**保存了记录的插入顺序**，在用Iterator遍历LinkedHashMap时，先得到的记录肯定是先插入的. 也可以在构造时用带参数，按照应用次数排序。在遍历的时候会比HashMap慢，不过有种情况例外，当HashMap容量很大，实际数据较少时，遍历起来可能会比LinkedHashMap慢，因为LinkedHashMap的遍历速度只和实际数据有关，和容量无关，而HashMap的遍历速度和他的容量有关。**LinkedHashMap采用的hash算法和HashMap相同，但是它重新定义了数组中保存的元素Entry，该Entry除了保存当前对象的引用外，还保存了其上一个元素before和下一个元素after的引用，从而在哈希表的基础上又构成了双向链接列表。**
* TreeMap：线程不同步，基于 **红黑树** （Red-Black tree）的NavigableMap 实现，**能够把它保存的记录根据键排序,默认是按键值的升序排序，也可以指定排序的比较器，当用Iterator 遍历TreeMap时，得到的记录是排过序的。**
* HashTable：线程安全，HashMap的迭代器(Iterator)是fail-fast迭代器。**HashTable不能存储NULL的key和value。**

**13. Collection包结构，与Collections的区别。**

**（1）Collection  是单列集合**

**List**元素是有序的、可重复

有序的 collection，可以对列表中每个元素的插入位置进行精确地控制。

可以根据元素的整数索引（在列表中的位置）访问元素，并搜索列表中的元素。

可存放重复元素，元素存取是有序的。

List接口中常用类

**l Vector**：线程安全，但速度慢，已被ArrayList替代。

底层数据结构是数组结构

**l ArrayList**：线程不安全，查询速度快。

            底层数据结构是数组结构

**l LinkedList**：线程不安全。增删速度快。

             底层数据结构是列表结构

**Set**(集)元素无序的、不可重复。

取出元素的方法只有迭代器。不可以存放重复元素，元素存取是无序的。

Set接口中常用的类

**l HashSet**：线程不安全，存取速度快。

        它是如何保证元素唯一性的呢？依赖的是元素的hashCode方法和euqals方法。

**l TreeSet**：线程不安全，可以对Set集合中的元素进行排序。

 它的排序是如何进行的呢？通过compareTo或者compare方法中的来保证元素的唯一性。元素是以二叉树的形式存放的。

**（2）Map**是一个双列集合

|--**Hashtable**:线程安全，速度快。底层是哈希表数据结构。是**同步**的。

不允许null作为键，null作为值。

      |--**Properties**:用于配置文件的定义和操作，使用频率非常高，同时键和值都是字符串。

是集合中可以和IO技术相结合的对象。(到了IO在学习它的特有和io相关的功能。)

|--**HashMap**:线程不安全，速度慢。底层也是哈希表数据结构。是**不同步**的。

允许null作为键，null作为值。替代了Hashtable.

   |--**LinkedHashMap**: 可以保证HashMap集合有序。存入的顺序和取出的顺序一致。

|--**TreeMap**：可以用来对Map集合中的**键**进行排序.

**（3）Collection 和 Collections的区别**

**Collection**是集合类的上级接口，子接口主要有Set 和List、Map。

**Collections**是针对集合类的一个帮助类，提供了操作集合的工具方法：一系列静态方法实现对各种集合的搜索、排序、线程安全化等操作。

**14. try catch finally，try里有return，finally还执行么？**

答案：执行，并且finally的执行早于try里面的return

结论：

1、不管有木有出现异常，finally块中代码都会执行；

2、当try和catch中有return时，finally仍然会执行；

3、finally是在return后面的表达式运算后执行的（此时并没有返回运算后的值，而是先把要返回的值保存起来，管finally中的代码怎么样，返回的值都不会改变，任然是之前保存的值），所以函数返回值是在finally执行前确定的；

4、finally中最好不要包含return，否则程序会提前退出，返回值不是try或catch中保存的返回值。

举例：

**情况1**：try{} catch(){}finally{} return;

显然程序按顺序执行。

**情况2**:try{ return; }catch(){} finally{} return;

程序执行try块中return之前（包括return语句中的表达式运算）代码；

再执行finally块，最后执行try中return;

finally块之后的语句return，因为程序在try中已经return所以不再执行。

**情况3**:try{ } catch(){return;} finally{} return;

程序先执行try，如果遇到异常执行catch块，

有异常：则执行catch中return之前（包括return语句中的表达式运算）代码，再执行finally语句中全部代码，

最后执行catch块中return. finally之后也就是4处的代码不再执行。

无异常：执行完try再finally再return.

**情况4**:try{ return; }catch(){} finally{return;}

程序执行try块中return之前（包括return语句中的表达式运算）代码；

再执行finally块，因为finally块中有return所以提前退出。

**情况5**:try{} catch(){return;}finally{return;}

程序执行catch块中return之前（包括return语句中的表达式运算）代码；

再执行finally块，因为finally块中有return所以提前退出。

**情况6**:try{ return;}catch(){return;} finally{return;}

程序执行try块中return之前（包括return语句中的表达式运算）代码；

有异常：执行catch块中return之前（包括return语句中的表达式运算）代码；

则再执行finally块，因为finally块中有return所以提前退出。

无异常：则再执行finally块，因为finally块中有return所以提前退出。

**最终结论：**任何执行try 或者catch中的return语句之前，都会先执行finally语句，如果finally存在的话。

如果finally中有return语句，那么程序就return了，所以finally中的return是一定会被return的，编译器把finally中的return实现为一个warning。

**15. Excption与Error包结构。OOM你遇到过哪些情况，SOF你遇到过哪些情况。**

**（1）Java异常**

Java中有Error和Exception，它们都是继承自Throwable类。

[](https://github.com/FrankLee96/interview/blob/master/java/error.png)

[](https://github.com/FrankLee96/interview/blob/master/java/exception.png)

**二者的不同之处**

Exception：

* 可以是可被控制(checked) 或不可控制的(unchecked)。
* 表示一个由程序员导致的错误。
* 应该在应用程序级被处理。

Error：

* 总是不可控制的(unchecked)。
* 经常用来用于表示系统错误或低层资源的错误。
* 如何可能的话，应该在系统级被捕捉。

**异常的分类**

* **Checked exception**: 这类异常都是Exception的子类。异常的向上抛出机制进行处理，假如子类可能产生A异常，那么在父类中也必须throws A异常。可能导致的问题：代码效率低，耦合度过高。
* **Unchecked exception**: **这类异常都是RuntimeException的子类，虽然RuntimeException同样也是Exception的子类，但是它们是非凡的，它们不能通过client code来试图解决**，所以称为Unchecked exception 。

**16. Java面向对象的三个特征与含义。**

答案：继承、多态、封装

**17. Override和Overload的含义与区别。**

答案：见JVM静态分派和动态分派

**18. Interface与abstract类的区别。**

答案：  
1.abstract class 在Java中表示的是一种继承关系，一个类只能使用一次继承关系。但是，一个类却可以实现多个interface。  
2.在abstract class 中可以有自己的数据成员，也可以有非abstarct的方法，而在interface中，只能够有静态的不能被修改的数据成员（也就是必须是static final的，不过在 interface中一般不定义数据成员），所有的方法都是public abstract的。  
3.抽象类中的变量默认是 friendly 型，其值可以在子类中重新定义，也可以重新赋值。接口中定义的变量默认是public static final 型，且必须给其赋初值，所以实现类中不能重新定义，也不能改变其值。  
4.abstract class和interface所反映出的设计理念不同。其实abstract class表示的是"is-a"关系，interface表示的是"like-a"关系，门和报警的关系。   
5.实现抽象类和接口的类必须实现其中的所有方法。抽象类中可以有非抽象方法。接口中则不能有实现方法。

abstract class 和 interface 是 Java语言中的两种定义抽象类的方式，它们之间有很大的相似性。但是对于它们的选择却又往往反映出对于问题领域中的概念本质的理解、对于设计意图的反映是否正确、合理，因为它们表现了概念间的不同的关系。

**19. Static class 与non static class的区别。**

答案：

**比对如下：**

                                         静态对象                                                        非静态对象       
拥有属性：                是类共同拥有的                                             是类各对象独立拥有的  
内存分配：               内存空间上是固定的                                      空间在各个附属类里面分配   
分配顺序：               先分配静态对象的空间                    继而再对非静态对象分配空间,也就是初始化顺序是先静态再非静态.  
**Java静态对象到底有什么好处？**

A，静态对象的数据在全局是唯一的，一改都改。如果你想要处理的东西是整个程序中唯一的，弄成静态是个好方法。 非静态的东西你修改以后只是修改了他自己的数据，但是不会影响其他同类对象的数据。   
B，引用方便。**直接用 类名.静态方法名  或者  类名.静态变量名**就可引用并且直接可以修改其属性值，不用get和set方法。  
C，保持数据的唯一性。此数据全局都是唯一的，修改他的任何一处地方，在程序所有使用到的地方都将会体现到这些数据的修改。有效减少多余的浪费。

D，static final用来修饰成员变量和成员方法，可简单理解为“全局常量”。对于变量，表示一旦给值就不可修改；对于方法，表示不可覆盖。

**20. java多态的实现原理。**

答案：见JVM动态分派

21. 实现多线程的两种方法：Thread与Runable。

**22. volatile原理与线程同步的方法：sychronized、lock、reentrantLock等。**

**（1）**[**Volatile原理**](http://www.cnblogs.com/dolphin0520/p/3920373.html)

**（一）计算机内存模型**

计算机在执行程序时，每条指令都是在CPU中执行的，而执行指令过程中，势必涉及到数据的读取和写入。由于程序运行过程中的临时数据是存放在主存（物理内存）当中的，这时就存在一个问题，由于CPU执行速度很快，而从内存读取数据和向内存写入数据的过程跟CPU执行指令的速度比起来要慢的多，因此如果任何时候对数据的操作都要通过和内存的交互来进行，会大大降低指令执行的速度。因此在CPU里面就有了高速缓存。**当程序在运行过程中，会将运算需要的数据从主存复制一份到CPU的高速缓存当中，那么CPU进行计算时就可以直接从它的高速缓存读取数据和向其中写入数据，当运算结束之后，再将高速缓存中的数据刷新到主存当中**。举个简单的例子，比如下面的这段代码：

i = i + 1;

当线程执行这个语句时，会先从主存当中读取i的值，然后复制一份到高速缓存当中，然后 CPU 执行指令对i进行加1操作，然后将数据写入高速缓存，最后将高速缓存中i最新的值刷新到主存当中。

这个代码在单线程中运行是没有任何问题的，但是在多线程中运行就会有问题了。在多核 CPU 中，每条线程可能运行于不同的 CPU 中，因此 **每个线程运行时有自己的高速缓存**（对单核CPU来说，其实也会出现这种问题，只不过是以线程调度的形式来分别执行的）。比如同时有两个线程执行这段代码，假如初始时i的值为0，那么我们希望两个线程执行完之后i的值变为2。但是事实会是这样吗？

可能出现这种情况：初始时，**两个线程分别读取i的值存入各自所在的 CPU 的高速缓存当中，然后 线程1 进行加1操作，然后把i的最新值1写入到内存。此时线程2的高速缓存当中i的值还是0，进行加1操作之后，i的值为1，然后线程2把i的值写入内存。最终结果i的值是1，而不是2。这就是著名的缓存一致性问题**。通常称这种被多个线程访问的变量为共享变量。

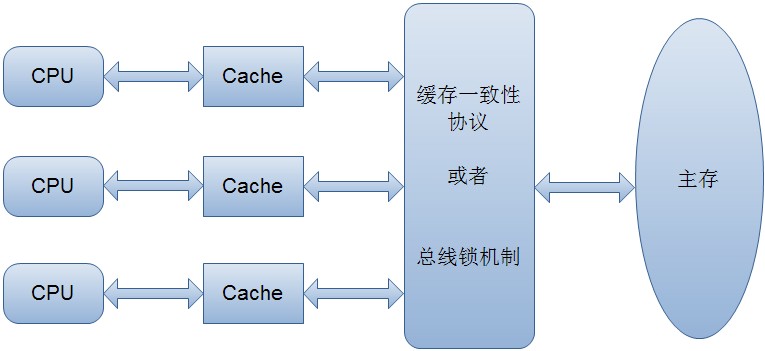
为了解决缓存不一致性问题，通常来说有以下两种解决方法：

* 通过在总线加LOCK#锁的方式
* 通过 **缓存一致性协议**

这两种方式都是硬件层面上提供的方式。

在早期的 CPU 当中，是通过在总线上加LOCK#锁的形式来解决缓存不一致的问题。因为 CPU 和其他部件进行通信都是通过总线来进行的，如果对总线加LOCK#锁的话，也就是说阻塞了其他 CPU 对其他部件访问（如内存），从而使得只能有一个 CPU 能使用这个变量的内存。比如上面例子中 如果一个线程在执行 i = i +1，如果在执行这段代码的过程中，在总线上发出了LCOK#锁的信号，那么只有等待这段代码完全执行完毕之后，其他CPU才能从变量i所在的内存读取变量，然后进行相应的操作。这样就解决了缓存不一致的问题。但是上面的方式会有一个问题，**由于在锁住总线期间，其他CPU无法访问内存，导致效率低下**。

所以就出现了缓存一致性协议。最出名的就是 Intel 的MESI协议，MESI协议保证了每个缓存中使用的共享变量的副本是一致的。它核心的思想是：**当CPU写数据时，如果发现操作的变量是共享变量，即在其他CPU中也存在该变量的副本，会发出信号通知其他CPU将该变量的缓存行置为无效状态，因此当其他CPU需要读取这个变量时，发现自己缓存中缓存该变量的缓存行是无效的，那么它就会从内存重新读取**。

[](https://github.com/FrankLee96/interview/blob/master/java/volatile_1.jpg)

**（二）Java内存模型**

在Java虚拟机规范中试图定义一种Java内存模型（Java Memory Model，JMM）来屏蔽各个硬件平台和操作系统的内存访问差异，以实现让Java程序在各种平台下都能达到一致的内存访问效果。那么Java内存模型规定了程序中变量的访问规则，往大一点说是定义了程序执行的次序。**注意，为了获得较好的执行性能，Java内存模型并没有限制执行引擎使用处理器的寄存器或者高速缓存来提升指令执行速度，也没有限制编译器对指令进行重排序。也就是说，在java内存模型中，也会存在缓存一致性问题和指令重排序的问题**。

**Java内存模型规定所有的变量都是存在主存当中（类似于前面说的物理内存），每个线程都有自己的工作内存（类似于前面的高速缓存）。线程对变量的所有操作都必须在工作内存中进行，而不能直接对主存进行操作。并且每个线程不能访问其他线程的工作内存**。

在Java中，执行下面这个语句：

i = 10;

执行线程必须先在自己的工作线程中对变量i所在的缓存行进行赋值操作，然后再写入主存当中。而不是直接将数值10写入主存当中。那么Java语言本身对 原子性、可见性以及有序性提供了哪些保证呢？

**原子性**

即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

**在Java中，对基本数据类型的变量的读取和赋值操作是原子性操作，即这些操作是不可被中断的，要么执行，要么不执行**。上面一句话虽然看起来简单，但是理解起来并不是那么容易。看下面一个例子i：请分析以下哪些操作是原子性操作：

x = 10; //语句1

y = x; //语句2

x++; //语句3

x = x + 1; //语句4

咋一看，有些朋友可能会说上面的4个语句中的操作都是原子性操作。**其实只有语句1是原子性操作，其他三个语句都不是原子性操作**。

* 语句1是直接将数值10赋值给x，也就是说线程执行这个语句的会直接将数值10写入到工作内存中。
* 语句2实际上包含2个操作，它先要去读取x的值，再将x的值写入工作内存，虽然读取x的值以及 将x的值写入工作内存 这2个操作都是原子性操作，但是合起来就不是原子性操作了。
* 同样的，x++和 x = x+1包括3个操作：读取x的值，进行加1操作，写入新的值。

也就是说，**只有简单的读取、赋值（而且必须是将数字赋值给某个变量，变量之间的相互赋值不是原子操作）才是原子操作**。不过这里有一点需要注意：**在32位平台下，对64位数据的读取和赋值是需要通过两个操作来完成的，不能保证其原子性。但是好像在最新的JDK中，JVM已经保证对64位数据的读取和赋值也是原子性操作了**。

从上面可以看出，Java内存模型只保证了基本读取和赋值是原子性操作，如果要实现更大范围操作的原子性，可以通过synchronized和Lock来实现。由于synchronized和Lock能够保证任一时刻只有一个线程执行该代码块，那么自然就不存在原子性问题了，从而保证了原子性。

**可见性**

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

对于可见性，Java提供了volatile关键字来保证可见性。**当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值**。而普通的共享变量不能保证可见性，因为普通共享变量被修改之后，什么时候被写入主存是不确定的，当其他线程去读取时，此时内存中可能还是原来的旧值，因此无法保证可见性。

另外，通过synchronized和Lock也能够保证可见性，synchronized和Lock能保证同一时刻只有一个线程获取锁然后执行同步代码，并且在释放锁之前会将对变量的修改刷新到主存当中。因此可以保证可见性。

**有序性**

即程序执行的顺序按照代码的先后顺序执行。

指令重排序，一般来说，处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。

**处理器在进行重排序时是会考虑指令之间的数据依赖性，如果一个指令Instruction 2必须用到Instruction 1的结果，那么处理器会保证Instruction 1会在Instruction 2之前执行**。

在Java内存模型中，允许编译器和处理器对指令进行重排序，但是重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。

在Java里面，可以通过volatile关键字来保证一定的“有序性”（具体原理在下一节讲述）。另外可以通过synchronized和Lock来保证有序性，很显然，synchronized和Lock保证每个时刻是有一个线程执行同步代码，相当于是让线程顺序执行同步代码，自然就保证了有序性。

另外，Java内存模型具备一些先天的“有序性”，即不需要通过任何手段就能够得到保证的有序性，这个通常也称为 happens-before 原则。如果两个操作的执行次序无法从happens-before原则推导出来，那么它们就不能保证它们的有序性，虚拟机可以随意地对它们进行重排序。

下面就来具体介绍下happens-before原则（先行发生原则）：

* **程序次序规则**：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作
* **锁定规则**：一个unLock操作先行发生于后面对同一个锁额lock操作
* **volatile变量规则**：对一个变量的写操作先行发生于后面对这个变量的读操作
* **传递规则**：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C
* **线程启动规则**：Thread对象的start()方法先行发生于此线程的每个一个动作
* **线程中断规则**：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生
* **线程终结规则**：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行
* **对象终结规则**：一个对象的初始化完成先行发生于他的finalize()方法的开始

对于程序次序规则来说，我的理解就是一段程序代码的执行在单个线程中看起来是有序的。注意，虽然这条规则中提到“书写在前面的操作先行发生于书写在后面的操作”，这个应该是程序看起来执行的顺序是按照代码顺序执行的，因为虚拟机可能会对程序代码进行指令重排序。虽然进行重排序，但是最终执行的结果是与程序顺序执行的结果一致的，它只会对不存在数据依赖性的指令进行重排序。因此，在单个线程中，程序执行看起来是有序执行的，这一点要注意理解。事实上，这个规则是用来保证程序在单线程中执行结果的正确性，但无法保证程序在多线程中执行的正确性。

第二条规则也比较容易理解，也就是说无论在单线程中还是多线程中，同一个锁如果出于被锁定的状态，那么必须先对锁进行了释放操作，后面才能继续进行lock操作。

第三条规则是一条比较重要的规则，也是后文将要重点讲述的内容。**直观地解释就是，如果一个线程先去写一个变量，然后一个线程去进行读取，那么写入操作肯定会先行发生于读操作**。

第四条规则实际上就是体现happens-before原则具备传递性。

**（三）深入剖析Volatile关键字**

**Volatile的语义**

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

* 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。
* 禁止进行指令重排序。

先看一段代码，假如线程1先执行，线程2后执行：

//线程1

boolean stop = false;

while(!stop){

doSomething();

}

//线程2

stop = true;

这段代码是很典型的一段代码，很多人在中断线程时可能都会采用这种标记办法。但是事实上，这段代码会完全运行正确么？即一定会将线程中断么？不一定，也许在大多数时候，这个代码能够把线程中断，但是也有可能会导致无法中断线程（虽然这个可能性很小，但是只要一旦发生这种情况就会造成死循环了）。

下面解释一下这段代码为何有可能导致无法中断线程。在前面已经解释过，每个线程在运行过程中都有自己的工作内存，那么线程1在运行的时候，会将stop变量的值拷贝一份放在自己的工作内存当中。

那么当线程2更改了stop变量的值之后，但是还没来得及写入主存当中，线程2转去做其他事情了，那么线程1由于不知道线程2对stop变量的更改，因此还会一直循环下去。

但是用volatile修饰之后就变得不一样了：

　- 使用volatile关键字会强制将修改的值立即写入主存；

* 使用volatile关键字的话，当线程2进行修改时，会导致线程1的工作内存中缓存变量stop的缓存行无效（*反映到硬件层的话，就是CPU的L1或者L2缓存中对应的缓存行无效*）；
* 由于线程1的工作内存中缓存变量stop的缓存行无效，所以线程1再次读取变量stop的值时会去**主存**读取。
* **那么在线程2修改stop值时（当然这里包括2个操作，修改线程2工作内存中的值，然后将修改后的值写入内存），会使得线程1的工作内存中缓存变量stop的缓存行无效，然后线程1读取时，发现自己的缓存行无效！！！！！！，它会等待缓存行对应的主存地址被更新之后，然后去对应的主存读取最新的值。**

那么线程1读取到的就是最新的正确的值。

**Volatile与原子性**

从上面知道volatile关键字保证了操作的可见性，但是volatile能保证对变量的操作是原子性吗？

下面看一个例子：

public class Test {

public volatile int inc = 0;

public void increase() {

inc++;

}

public static void main(String[] args) {

final Test test = new Test();

for(int i=0;i<10;i++){

new Thread(){

public void run() {

for(int j=0;j<1000;j++)

test.increase();

};

}.start();

}

while(Thread.activeCount()>1) //保证前面的线程都执行完

Thread.yield();

System.out.println(test.inc);

}

}

大家想一下这段程序的输出结果是多少？**也许有些朋友认为是10000。但是事实上运行它会发现每次运行结果都不一致，都是一个小于10000的数字**。可能有的朋友就会有疑问，不对啊，上面是对变量inc进行自增操作，由于volatile保证了可见性，那么在每个线程中对inc自增完之后，在其他线程中都能看到修改后的值啊，所以有10个线程分别进行了1000次操作，那么最终inc的值应该是1000\*10=10000。

**这里面就有一个误区了，volatile关键字能保证可见性没有错，但是上面的程序错在没能保证原子性。可见性只能保证每次读取的是最新的值，但是volatile没办法保证对变量的操作的原子性**。

在前面已经提到过，**自增操作是不具备原子性的**，它包括读取变量的原始值、进行加1操作、写入工作内存。那么就是说自增操作的三个子操作可能会分割开执行，就有可能导致下面这种情况出现：

假如某个时刻变量inc的值为10，**线程1**对变量进行自增操作，线程1先读取了变量inc的原始值，然后线程1被阻塞了；

然后**线程2**对变量进行自增操作，线程2也去读取变量inc的原始值，由于线程1只是对变量inc进行读取操作，而没有对变量进行修改操作，所以不会导致线程2的工作内存中缓存变量inc的缓存行无效，所以线程2会直接去主存读取inc的值，发现inc的值时10，然后进行加1操作，并把11写入工作内存，最后写入主存。

然后**线程1**接着进行加1操作，由于已经读取了inc的值，注意此时在线程1的工作内存中inc的值仍然为10，所以线程1对inc进行加1操作后inc的值为11，然后将11写入工作内存，最后写入主存。

那么两个线程分别进行了一次自增操作后，inc只增加了1。解释到这里，可能有朋友会有疑问，不对啊，前面不是保证一个变量在修改volatile变量时，会让缓存行无效吗？然后其他线程去读就会读到新的值，对，这个没错。这个就是上面的happens-before规则中的volatile变量规则，但是要注意，**线程1对变量进行读取操作之后，被阻塞了的话，并没有对inc值进行修改。然后虽然volatile能保证线程2对变量inc的值读取是从内存中读取的，但是线程1没有进行修改，所以线程2根本就不会看到修改的值**。

**根源就在这里，自增操作不是原子性操作，而且volatile也无法保证对变量的任何操作都是原子性的**。解决的方法也就是对提供原子性的自增操作即可。

在Java 1.5的java.util.concurrent.atomic包下提供了一些原子操作类，即对基本数据类型的 自增（加1操作），自减（减1操作）、以及加法操作（加一个数），减法操作（减一个数）进行了封装，保证这些操作是原子性操作。atomic是利用**CAS**来实现原子性操作的（Compare And Swap），CAS实际上是利用处理器提供的CMPXCHG指令实现的，而处理器执行CMPXCHG指令是一个原子性操作。

**Volatile与有序性**

在前面提到volatile关键字能禁止指令重排序，所以volatile能在一定程度上保证有序性。volatile关键字禁止指令重排序有两层意思：

* 当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见，在其后面的操作肯定还没有进行；
* **在进行指令优化时，不能将在对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行**。

可能上面说的比较绕，举个简单的例子：

//x、y为非volatile变量

//flag为volatile变量

x = 2; //语句1

y = 0; //语句2

flag = true; //语句3

x = 4; //语句4

y = -1; //语句5

由于flag变量为volatile变量，那么在进行指令重排序的过程的时候，不会将语句3放到语句1、语句2前面，也不会讲语句3放到语句4、语句5后面。但是要注意语句1和语句2的顺序、语句4和语句5的顺序是不作任何保证的。

并且volatile关键字能保证，执行到语句3时，语句1和语句2必定是执行完毕了的，且语句1和语句2的执行结果对语句3、语句4、语句5是可见的。

**Volatile的原理和实现机制**

前面讲述了源于volatile关键字的一些使用，下面我们来探讨一下volatile到底如何保证可见性和禁止指令重排序的。下面这段话摘自《深入理解Java虚拟机》：

观察加入volatile关键字和没有加入volatile关键字时所生成的汇编代码发现，加入volatile关键字时，会多出一个lock前缀指令

lock前缀指令实际上相当于一个 **内存屏障**（也成内存栅栏），内存屏障会提供3个功能：

* 它 **确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面**；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；
* 它会 **强制将对缓存的修改操作立即写入主存**；
* **如果是写操作，它会导致其他CPU中对应的缓存行无效**。

**23. 锁的等级：方法锁、对象锁、类锁。**

**（1）基础**

Java中的每一个对象都可以作为锁。

* 对于同步方法，**锁是当前实例对象**。
* 对于静态同步方法，**锁是当前对象的Class对象**。
* 对于同步方法块，**锁是Synchonized括号里配置的对象**。

当一个线程试图访问同步代码块时，它首先必须得到锁，退出或抛出异常时必须释放锁。那么锁存在哪里呢？锁里面会存储什么信息呢？

**（2）同步的原理**

JVM规范规定JVM基于进入和退出 Monitor 对象来实现方法同步和代码块同步，但两者的实现细节不一样。代码块同步是使用monitorenter和monitorexit指令实现，而方法同步是使用另外一种方式实现的，细节在JVM规范里并没有详细说明，但是方法的同步同样可以使用这两个指令来实现。

monitorenter指令是在编译后插入到同步代码块的开始位置，而monitorexit是插入到方法结束处和异常处，JVM要保证每个monitorenter必须有对应的monitorexit与之配对。任何对象都有一个monitor与之关联，当且一个monitor被持有后，它将处于锁定状态。线程执行到 monitorenter 指令时，将会尝试获取对象所对应的 monitor 的所有权，即尝试获得对象的锁。

**2.1、Java对象头**

锁存在Java对象头里。如果对象是数组类型，则虚拟机用3个Word（字宽）存储对象头，如果对象是非数组类型，则用2字宽存储对象头。在32位虚拟机中，一字宽等于四字节，即32bit。

| 长度 | 内容 | 说明 |

| :------------- | :------------- |

|32/64bit| Mark Word |存储对象的hashCode或锁信息等|

|32/64bit| Class Metadata Address |存储到对象类型数据的指针|

|32/64bit| Array length |数组的长度（如果当前对象是数组）|

Java对象头里的Mark Word里默认存储对象的HashCode，分代年龄和锁标记位。32位JVM的Mark Word的默认存储结构如下：

| | 25 bit |4bit|1bit是否是偏向锁|2bit锁标志位| | :------------- | :------------- | |无锁状态| 对象的hashCode| 对象分代年龄| 0| 01|

在运行期间Mark Word里存储的数据会随着锁标志位的变化而变化。Mark Word可能变化为存储以下4种数据：

[](https://github.com/FrankLee96/interview/blob/master/java/synchronized_1.png)

**2.2、锁的升级**

Java SE1.6为了减少获得锁和释放锁所带来的性能消耗，引入了“偏向锁”和“轻量级锁”，所以在Java SE1.6里锁一共有四种状态，无锁状态，偏向锁状态，轻量级锁状态和重量级锁状态，它会随着竞争情况逐渐升级。**锁可以升级但不能降级，意味着偏向锁升级成轻量级锁后不能降级成偏向锁。这种锁升级却不能降级的策略，目的是为了提高获得锁和释放锁的效率**。

24. 写出生产者消费者模式。

**25. ThreadLocal的设计理念与作用。**

**（1）ThreadLocal的介绍**

ThreadLocal是一个线程的内部存储类，可以在每个线程的内部存储数据，当某个数据的作用域应该对应线程的时候就应该使用它；而是当某个很复杂的逻辑下的对象传递，需要在线程这个作用域内贯穿其中，用ThreadLocal可以避免这个创建多个静态类。当你创建一个ThreadLocal对象后，注意，是一个对象，你在不同线程中去访问它的值是可以不一样的，并且是和线程相关联的。它的实现原理其实比较简单，每个线程中都会维护一个ThreadLocalMap，当在某个线程中访问时，会取出这个线程自己的Map并且用当前ThreadLocal对象做索引来取出相对应的Value值，从而达到不同线程不同值的效果。

**（2）实现原理**

　1、每个Thread对象内部都维护了一个ThreadLocalMap这样一个ThreadLocal的Map，可以存放若干个ThreadLocal。

|  |  |
| --- | --- |
| 1  2  3 | /\* ThreadLocal values pertaining to this thread. This map is maintained   \* by the ThreadLocal class. \*/  ThreadLocal.ThreadLocalMap threadLocals = null; |

　2、当我们在调用get()方法的时候，先获取当前线程，然后获取到当前线程的ThreadLocalMap对象，如果非空，那么取出ThreadLocal的value，否则进行初始化，初始化就是将initialValue的值set到ThreadLocal中。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public T get() {  **Thread t = Thread.currentThread();**  **ThreadLocalMap map = getMap(t);**      if (map != null) {          ThreadLocalMap.Entry e = map.getEntry(this);          if (e != null)              return (T)e.value;      }      return setInitialValue();  } |

　3、当我们调用set()方法的时候，很常规，就是将值设置进ThreadLocal中。

4、ThreadLocalMap里面存储的Entry对象本质上是一个WeakReference<ThreadLocal>。也就是说，ThreadLocalMap里面存储的对象本质是一个对ThreadLocal对象的弱引用，该ThreadLocal随时可能会被回收！即导致ThreadLocalMap里面对应的Value的Key是null。我们需要把这样的Entry给清除掉，**不要让它们占坑，避免内存泄漏。**

那为什么需要弱引用呢？因为线程的声明周期是长于ThreadLocal对象的，当此对象不再需要的时候如果线程中**还持有它的引用势必也会产生内存泄漏的问题，所以自然应该是用弱引用来进行key的保存。**

26. ThreadPool用法与优势。

27. Concurrent包里的其他东西：ArrayBlockingQueue、CountDownLatch等等。

**28. wait()和sleep()的区别。**

答案：

① 这两个方法来自不同的类分别是，sleep来自Thread类，和wait来自Object类。

sleep是Thread的静态类方法，**谁调用的谁去睡觉，即使在a线程里调用b的sleep方法，实际上还是a去睡觉，要让b线程睡觉要在b的代码中调用sleep。**

② 锁: 最主要是sleep方法没有释放锁，而wait方法释放了锁，使得其他线程可以使用同步控制块或者方法。

sleep不出让系统资源；wait是进入线程等待池等待，出让系统资源，其他线程可以占用CPU。一般wait不会加时间限制，因为如果wait线程的运行资源不够，再出来也没用，要等待其他线程调用notify/notifyAll唤醒等待池中的所有线程，才会进入就绪队列等待OS分配系统资源。sleep(milliseconds)可以用时间指定使它自动唤醒过来，如果时间不到只能调用interrupt()强行打断。

Thread.sleep(0)的作用是“触发操作系统立刻重新进行一次CPU竞争”。

③ 使用范围：wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用。

   synchronized(x){   
      x.notify()   
     //或者wait()   
   }

**29. foreach与正常for循环效率对比。**

答案：

可以看出，循环ArrayList时，普通for循环比foreach循环花费的时间要少一点；循环LinkList时，普通for循环比foreach循环花费的时间要多很多。

当我将循环次数提升到一百万次的时候，循环ArrayList，普通for循环还是比foreach要快一点；但是普通for循环在循环LinkList时，程序直接卡死。

**结论：**需要**循环数组结构**的数据时，建议**使用普通for循环**，因为for循环采用下标访问，对于数组结构的数据来说，采用下标访问比较好。

需要**循环链表结构**的数据时，**一定不要使用普通for循环**，这种做法很糟糕，数据量大的时候有可能会导致系统崩溃。

**原因：foreach使用的是迭代器**

30. Java IO与NIO。

**31. 反射的作用与原理。**

（1）概念

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；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

（2）功能

反射机制主要提供了以下功能：

* 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；
* 生成动态代理。

**32. 泛型常用特点，List<String>能否转为List<Object>。**

**（1）Java泛型**

开发人员在使用泛型的时候，很容易根据自己的直觉而犯一些错误。比如一个方法如果接收List<Object>作为形式参数，那么如果尝试将一个List<String>的对象作为实际参数传进去，却发现无法通过编译。虽然从直觉上来说，Object是String的父类，这种类型转换应该是合理的。**但是实际上这会产生隐含的类型转换问题，因此编译器直接就禁止这样的行为**。

**（2）实现方式：类型擦除**

Java中的泛型基本上都是在编译器这个层次来实现的，**在生成的Java字节代码中是不包含泛型中的类型信息的。使用泛型的时候加上的类型参数，会被编译器在编译的时候去掉，这个过程就称为类型擦除**。如在代码中定义的List<Object>和List<String>等类型，在编译之后都会变成List。**JVM看到的只是List，而由泛型附加的类型信息对JVM来说是不可见的**。Java编译器会在编译时尽可能的发现可能出错的地方，但是仍然无法避免在运行时刻出现类型转换异常的情况。

很多泛型的奇怪特性都与这个类型擦除的存在有关，包括：

* **泛型类并没有自己独有的Class类对象**。比如并不存在List<String>.class或是List<Integer>.class，而只有List.class。
* **静态变量是被泛型类的所有实例所共享的**。对于声明为MyClass<T>的类，访问其中的静态变量的方法仍然是 MyClass.myStaticVar。不管是通过new MyClass<String>还是new MyClass<Integer>创建的对象，都是共享一个静态变量。
* **泛型的类型参数不能用在Java异常处理的catch语句中**。因为异常处理是由JVM在**运行时刻**来进行的。由于类型信息被擦除，JVM是无法区分两个异常类型MyException<String>和MyException<Integer>的。对于JVM来说，它们都是 MyException类型的。也就无法执行与异常对应的catch语句。

**类型擦除**的基本过程也比较简单，首先是找到用来替换类型参数的具体类。这个具体类一般是Object。如果指定了类型参数的上界的话，则使用这个上界。把代码中的类型参数都替换成具体的类。同时去掉出现的类型声明，即去掉<>的内容。比如T get()方法声明就变成了Object get()；List<String>就变成了List。接下来就可能需要生成一些桥接方法（bridge method）。这是由于擦除了类型之后的类可能缺少某些必须的方法。比如考虑下面的代码：

class MyString implements Comparable<String> {

public int compareTo(String str) {

return 0;

}

}

当类型信息被擦除之后，上述类的声明变成了class MyString implements Comparable。但是这样的话，类MyString就会有编译错误，因为没有实现接口Comparable声明的int compareTo(Object)方法。这个时候就由编译器来动态生成这个方法。

**（3）泛型的检测：不符合泛型T的检测**

在进行编译之前就对所有泛型进行检测，加入类型检测和转换的指令，比如返回泛型的结果实际上返回的是擦出后的类型，而虚拟机会多加一个类型转换的指令。

**（4）需要泛型之间的类型转换怎么做？通配符？与PECS**

1. 如果你是想遍历collection，并对每一项元素操作时，此时这个集合时生产者（生产元素），应该使用 Collection<? extends E>，因为相对于你泛型E，能放进E中的（把Collection中的元素放入E类型中） ，应该是E的子类型。
2. 如果你是想添加元素到collection中去，那么此时集合时消费者（消费元素）应该使用Collection<? super E>，同理，你要将E类的元素放入Collection中去，那么Collection应该存放的是E的父类型。

（5）泛型与多态的冲突

33. 解析XML的几种方式的原理与特点：DOM、SAX、PULL。

34. Java与C++对比。

**35. Java1.7与1.8新特性。**

**（1）1.8**

**一、接口的默认方法与静态方法。**也就是接口中可以有实现方法；并且接口也可以有静态方法，工具类也可以使用接口来实现；

**二、Lambda 表达式。**简化了代码，实际上是函数式接口的简化

函数式接口（functional interface 也叫功能性接口，其实是同一个东西）。简单来说，函数式接口是只包含一个方法的接口。比如Java标准库中的java.lang.Runnable和 java.util.Comparator都是典型的函数式接口。java 8提供 @FunctionalInterface作为注解,这个注解是非必须的，只要接口符合函数式接口的标准（即只包含一个方法的接口），虚拟机会自动判断，但最好在接口上使用注解@FunctionalInterface进行声明，以免团队的其他人员错误地往接口中添加新的方法。

使用了函数式接口的匿名类可以使用Lambda来优化代码。

**三、方法引用**

在学习[lambda表达式](http://www.cnblogs.com/JohnTsai/p/5584905.html)之后，我们通常使用lambda表达式来创建匿名方法。然而，有时候我们仅仅是调用了一个已存在的方法。如下：

Arrays.sort(stringsArray,(s1,s2)->s1.compareToIgnoreCase(s2));

在Java8中，我们可以直接通过方法引用来**简写**lambda表达式中已经存在的方法。这种特性就叫方法引用：

Arrays.sort(stringsArray, String::compareToIgnoreCase);

**（2）1.7**

**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.安全的加减乘除**  
int Math.safeToInt(long value)

int Math.safeNegate(int value)

long Math.safeSubtract(long value1, int value2)

long Math.safeSubtract(long value1, long value2)

int Math.safeMultiply(int value1, int value2)

long Math.safeMultiply(long value1, int value2)

long Math.safeMultiply(long value1, long value2)

long Math.safeNegate(long value)

int Math.safeAdd(int value1, int value2)

long Math.safeAdd(long value1, int value2)

long Math.safeAdd(long value1, long value2)

int Math.safeSubtract(int value1, int value2)

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

36. 设计模式：单例、工厂、适配器、责任链、观察者等等。

37. JNI的使用。

**38. 匿名内部类使用的参数为什么要是final的**

答案：因为你得到的参数实际上是拷贝而来的

（1）具体：<https://www.zhihu.com/question/21395848/answer/91184967>

（2）精简：两点

1.形参为什么要拷贝一份：

不拷贝的话，匿名内部类中使用的引用与外部的引用是同一份引用，但是这个引用的生命周期与函数相同，若匿名内部类的生命周期超过这个函数的话（**比如新开了一个线程**），再去访问这个引用会有空指针等问题。  
2.形参为什么是final

从我们的角度来看，是看不到这个拷贝的过程。我们看到匿名内部类直接使用了这个引用，会造成一种错觉：当在匿名内部类中修改这个引用的时候（指向别的实例），我们同时修改的是这个外面的引用，因为看起来他们是同一个嘛。**然而并不是这样的。为了防止这种错觉，java索性将其设为final，即这个引用是无法修改的。**

**39. 单例模式的实现**

**最佳：静态内部类**

1. **public** **class** SingletonE {
3. **private** **static** **class** SingletonHolder {
4. /\*\*
5. \* 单例对象实例
6. \*/
7. **static** **final** SingletonE instance = **new** SingletonE();
8. }
10. **public** **static** SingletonE getInstance() {
11. **return** SingletonHolder.instance;
12. }
13. }