目录

Java 面试

Java 基础

集合

1、List 和 Set 的区别

Java 中的集合包括三大类，它们是 Set（集）、List（列表）和 Map（映射），它们都处于 java.util 包中，Set、List 和 Map 都是接口，它们有各自的实现类。Set 的实现类主要有 HashSet 和 TreeSet，List 的实现类主要有 ArrayList

Collection 是最基本的集合接口，声明了适用于 JAVA 集合的通用方法，list 和 set 都继承自 collection 接口。

Collection 接口的方法：

boolean add(Object o) ：向集合中加入一个对象的引用

void clear()：删除集合中所有的对象，即不再持有这些对象的引用

boolean isEmpty() ：判断集合是否为空

boolean contains(Object o) ： 判断集合中是否持有特定对象的引用

Iterartor iterator() ：返回一个 Iterator 对象，可以用来遍历集合中的元素

boolean remove(Object o) ：从集合中删除一个对象的引用

int size() ：返回集合中元素的数目

Object[] toArray() ： 返回一个数组，该数组中包括集合中的所有元素

关于：Iterator() 和 toArray() 方法都用于集合的所有的元素，前者返回一个 Iterator 对象，后者返回一个包含集合中所有元素的数组。

Collection 没有 get()方法来取得某个元素。只能通过 iterator()遍历元素。

List 的功能方法：

实际上有两种 List：一种是基本的 ArrayList,其优点在于随机访问元素，另一种是更强大的 LinkedList,它并不是为快速随机访问设计的，而是具有一套更通用的方法。

List：次序是 List 最重要的特点：它保证维护元素特定的顺序。List 为 Collection 添加了许多方法，使得能够向 List 中间插入与移除元素(这只推 荐 LinkedList 使用。)一个 List 可以生成 ListIterator,使用它可以从两个方向遍历 List,也可以从 List 中间插入和移除元素。

ArrayList：由数组实现的 List。允许对元素进行快速随机访问，但是向 List 中间插入与移除元素的速度很慢。ListIterator 只应该用来由后向前遍历 ArrayList,而不是用来插入和移除元素。因为那比 LinkedList 开销要大很多。

LinkedList ：对顺序访问进行了优化，向 List 中间插入与删除的开销并不大。随机访问则相对较慢。(使用 ArrayList 代替。)还具有下列方 法：addFirst(), addLast(), getFirst(), getLast(), removeFirst() 和 removeLast(), 这些方法 (没有在任何接口或基类中定义过)使得 LinkedList 可以当作堆栈、队列和双向队列使用。

Set 的功能方法：

Set 具有与 Collection 完全一样的接口，因此没有任何额外的功能。实际上 Set 就是 Collection,只 是行为不同。这是继承与多态思想的典型应用：表现不同的行为。Set 不保存重复的元素(至于如何判断元素相同则较为负责)

Set：存入 Set 的每个元素都必须是唯一的，因为 Set 不保存重复元素。加入 Set 的元素必须定义 equals()方法以确保对象的唯一性。Set 与 Collection 有完全一样的接口。Set 接口不保证维护元素的次序。

HashSet：为快速查找设计的 Set。存入 HashSet 的对象必须定义 hashCode()。

TreeSet： 保存次序的 Set, 底层为树结构。使用它可以从 Set 中提取有序的序列。

LinkedHashSet：具有 HashSet 的查询速度，且内部使用链表维护元素的顺序(插入的次序)。于是在使用迭代器遍历 Set 时，结果会按元素插入的次序显示。

list 与 Set 区别：

1、List,Set 都是继承自 Collection 接口

2、List 特点：元素有放入顺序，元素可重复 ，Set 特点：元素无放入顺序，元素不可重复，重复元素会覆盖掉，（元素虽然无放入顺序，但是元素在 set 中的位置是有该元素的 HashCode 决定的，其位置其实是固定的，加入 Set 的 Object 必须定义 equals()方法 ，另外 list 支持 for 循环，也就是通过下标来遍历，也可以用迭代器，但是 set 只能用迭代，因为他无序，无法用下标来取得想要的值。）

3.Set 和 List 对比：

Set：检索元素效率低下，删除和插入效率高，插入和删除不会引起元素位置改变。

List：和数组类似，List 可以动态增长，查找元素效率高，插入删除元素效率低，因为会引起其他元素位置改变。

2、HashSet 是如何保证不重复的

弄清怎么个逻辑达到元素不重复的，源码先上

HashSet 类中的 add()方法：

```
public boolean add(E e) {
      return map.put(e, PRESENT)==null;
   }
```

类中 map 和 PARENT 的定义：

```
private transient HashMap<E,Object> map;
 // Dummy value to associate with an Object in the backing Map 用来匹配 Map 中后面的对象的一个虚拟值
private static final Object PRESENT = new Object();
```

实际上 HashSet 如何保证不重复就是 HashMap 如何保证不重复。

HashMap 的 put 方法：

```
public V put(K key, V value) {
        if (key == null)
            return putForNullKey(value);
        int hash = hash(key.hashCode());
        int i = indexFor(hash, table.length);
        for (Entry<K,V> e = table[i]; e != null; e = e.next) {
            Object k;
            if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
                V oldValue = e.value;
                e.value = value;
                e.recordAccess(this);
                return oldValue;
```

```
                }
            }
            modCount++;
            addEntry(hash, key, value, i);
            return null;
        }
```

最重要的就是： if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

可以看到 for 循环中，遍历 table 中的元素，

1，如果 hash 码值不相同，说明是一个新元素，存；

如果没有元素和传入对象（也就是 add 的元素）的 hash 值相等，那么就认为这个元素在 table 中不存在，将其添加进 table；

2（1），如果 hash 码值相同，且 equles 判断相等，说明元素已经存在，不存；

2（2），如果 hash 码值相同，且 equles 判断不相等，说明元素不存在，存；

如果有元素和传入对象的 hash 值相等，那么，继续进行 equles()判断，如果仍然相等，那么就认为传入元素已经存在，不再添加，结束，否则仍然添加。

Hashcode 和 equals 重写的方法：

```
public int hashCode() {
            final int prime = 31;
            int result = 1;
            result = prime * result + age;
            result = prime * result + ((name == null) ? 0 : name.hashCode());
            return result;
        }
public boolean equals(Object obj) {
            if (this == obj)
                return true;
            if (obj == null)
                return false;
            if (getClass() != obj.getClass())
                return false;
            Person other = (Person) obj;
            if (age != other.age)
                return false;
            if (name == null) {
                if (other.name != null)
                    return false;
            } else if (!name.equals(other.name))
                return false;
            return true;
        }
```

3、HashMap 是线程安全的吗，为什么不是线程安全的

一直以来都知道 HashMap 是线程不安全的，但是到底为什么线程不安全，在多线程操作情况下什么时候线程不安全？

让我们先来了解一下 HashMap 的底层存储结构，HashMap 底层是一个 Entry 数组，一旦发生 Hash 冲突的的时候，HashMap 采用拉链法解决碰撞冲突，Entry 内部的变量：

```
final Object key;
Object value;
Entry next;
int hash;
```

通过 Entry 内部的 next 变量可以知道使用的是链表，这时候我们可以知道，如果多个线程，在某一时刻同时操作 HashMap 并执行 put 操作，而有大于两个 key 的 hash 值相同，如图中 a1、a2，这个时候需要解决碰撞冲突，而解决冲突的办法上面已经说过，对于链表的结构在这里不再赘述，暂且不讨论是从链表头部插入还是从尾部初入，这个时候两个线程如果恰好都取到了对应位置的头结点 e1，而最终的结果可想而知，a1、a2 两个数据中势必会有一个会丢失。

再来看下 put 方法：

```
public Object put(Object obj, Object obj1)
    {
        if(table == EMPTY_TABLE)
            inflateTable(threshold);
        if(obj == null)
            return putForNullKey(obj1);
        int i = hash(obj);
        int j = indexFor(i, table.length);
        for(Entry entry = table[j]; entry != null; entry = entry.next)
        {
            Object obj2;
            if(entry.hash == i && ((obj2 = entry.key) == obj ||
obj.equals(obj2)))
            {
                Object obj3 = entry.value;
                entry.value = obj1;
                entry.recordAccess(this);
                return obj3;
            }
        }
```

```
            modCount++;
            addEntry(i, obj, obj1, j);
            return null;
        }
```

put 方法不是同步的，同时调用了 addEntry 方法：

```
    void addEntry(int i, Object obj, Object obj1, int j)
    {
        if(size >= threshold && null != table[j])
        {
            resize(2 * table.length);
            i = null == obj ? 0 : hash(obj);
            j = indexFor(i, table.length);
        }
        createEntry(i, obj, obj1, j);
    }
```

addEntry 方法依然不是同步的，所以导致了线程不安全出现伤处问题，其他类似操作不再说明，源码一看便知，下面主要说一下另一个非常重要的知识点，同样也是 HashMap 非线程安全的原因，我们知道在 HashMap 存在扩容的情况，对应的方法为 HashMap 中的 resize 方法：

```
    void resize(int i)
        {
            Entry aentry[] = table;
            int j = aentry.length;
            if(j == 1073741824)
            {
                threshold = 2147483647;
                return;
            } else
            {
                Entry aentry1[] = new Entry[i];
                transfer(aentry1, initHashSeedAsNeeded(i));
```

```
                table = aentry1;
                threshold = (int)Math.min((float)i * loadFactor,
1.073742E+009F);
                return;
            }
        }
```

可以看到扩容方法也不是同步的，通过代码我们知道在扩容过程中，会新生成一个新的容量的数组，然后对原数组的所有键值对重新进行计算和写入新的数组，之后指向新生成的数组。

当多个线程同时检测到总数量超过门限值的时候就会同时调用 resize 操作，各自生成新的数组并 rehash 后赋给该 map 底层的数组 table，结果最终只有最后一个线程生成的新数组被赋给 table 变量，其他线程的均会丢失。而且当某些线程已经完成赋值而其他线程刚开始的时候，就会用已经被赋值的 table 作为原始数组，这样也会有问题。

4、HashMap 的扩容过程

5、HashMap 1.7 与 1.8 的 区别，说明 1.8 做了哪些优化，如何优化的？

（一）Hashmap 的结构，1.7 和 1.8 有哪些区别

不同点：

（1）JDK1.7 用的是头插法，而 JDK1.8 及之后使用的都是尾插法，那么他们为什么要这样做呢？因为 JDK1.7 是用单链表进行的纵向延伸，当采用头插法时会容易出现逆序且环形链表死循环问题。但是在 JDK1.8 之后是因为加入了红黑树使用尾插法，能够避免出现逆序且链表死循环的问题。

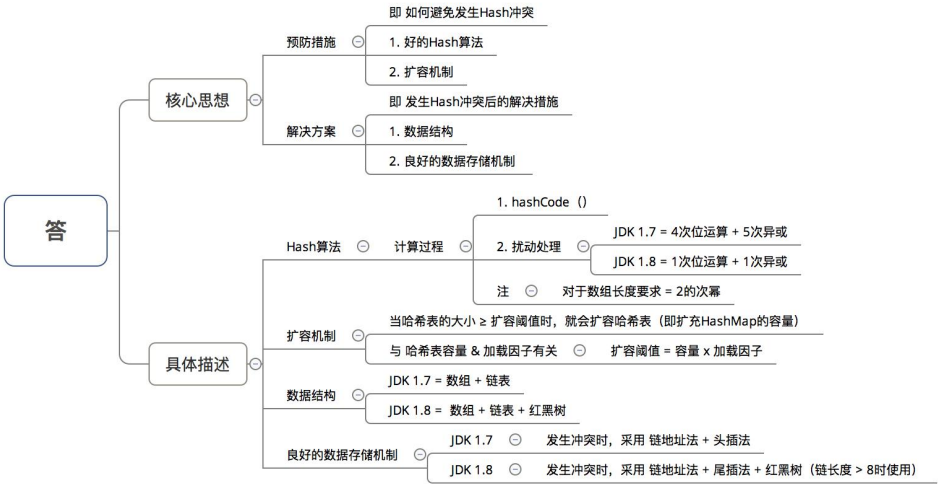
（2）扩容后数据存储位置的计算方式也不一样：

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&（这里就是为什么扩容的时候为啥一定必须是 2 的多少次幂的原因所在，因为如果只有 2 的 n 次幂的情况时最后一位二进制数才一定是 1，这样能最大程度减少 hash 碰撞）（hash 值 & length-1）

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，而不再是 JDK1.7 的那种异或的方法。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储存方式。

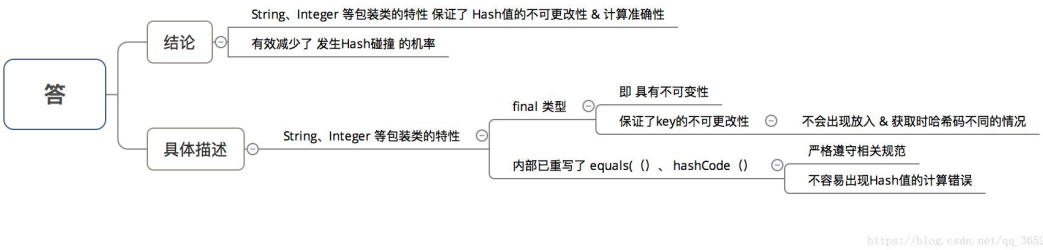
（3）JDK1.7 的时候使用的是数组+ 单链表的数据结构。但是在 JDK1.8 及之后时，使用的是数组+链表+红黑树的数据结构（当链表的深度达到 8 的时候，也就是默认阈值，就会自动扩容把链表转成红黑树的数据结构来把时间复杂度从 O（n）变成 O（logN）提高了效率）

（二）哈希表如何解决 Hash 冲突？

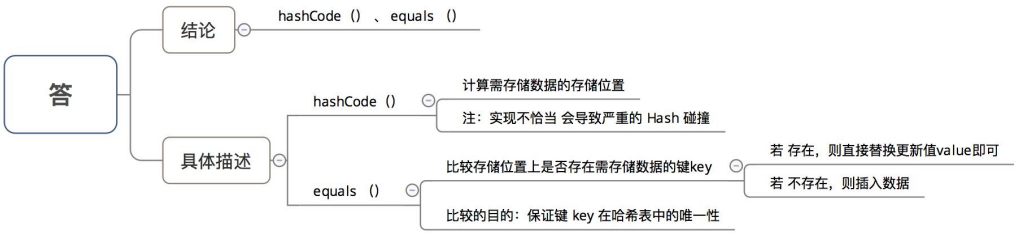


（三）为什么 HashMap 具备下述特点：键-值（key-value）都允许为空、线程不安全、不保证有序、存储位置随时间变化

（四）为什么 HashMap 中 String、Integer 这样的包装类适合作为 key 键



（五）HashMap 中的 key 若 Object 类型， 则需实现哪些方法？



6、final finally finalize

1.简单区别：

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

finally 是异常处理语句结构的一部分，表示总是执行。

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，供垃圾收集时的其他资源回收，例如关闭文件等。

2.中等区别：

虽然这个单词在 Java 中都存在，但是并没太多关联：

final：java 中的关键字，修饰符。

A).如果一个类被声明为 final，就意味着它不能再派生出新的子类，不能作为父类被继承。因此，一个类不能同时被声明为 abstract 抽象类的和 final 的类。

B).如果将变量或者方法声明为 final，可以保证它们在使用中不被改变.

1)被声明为 final 的变量必须在声明时给定初值，而在以后的引用中只能读取，不可修改。

2)被声明 final 的方法只能使用，不能重载。

finally：java 的一种异常处理机制。

finally 是对 Java 异常处理模型的最佳补充。finally 结构使代码总会执行，而不管无异常发生。使用 finally 可以维护对象的内部状态，并可以清理非内存资源。特别是在关闭数据库连接这方面，如果程序员把数据库连接的 close()方法放到 finally 中，就会大大降低程序出错的几率。

finalize：Java 中的一个方法名。

Java 技术使用 finalize()方法在垃圾收集器将对象从内存中清除出去前，做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没被引用时对这个对象调用的。它是在 Object 类中定义的，因此所的类都继承了它。子类覆盖 finalize()方法以整理系统资源或者执行其他清理工作。finalize()方法是在垃圾收集器删除对象之前对这个对象调用的。

3.详细区别：

这是一道再经典不过的面试题了，我们在各个公司的面试题中几乎都能看到它的身影。final、finally 和 finalize 虽然长得像孪生兄弟一样，但是它们的含义和用法却是大相径庭。

final 关键字我们首先来说说 final。它可以用于以下四个地方:

1).定义变量，包括静态的和非静态的。

2).定义方法的参数。

3).定义方法。

4).定义类。

定义变量，包括静态的和非静态的。定义方法的参数

第一种情况：

如果 final 修饰的是一个基本类型，就表示这个变量被赋予的值是不可变的，即它是个常量；

如果 final 修饰的是一个对象，就表示这个变量被赋予的引用是不可变的

这里需要提醒大家注意的是，不可改变的只是这个变量所保存的引用，并不是这个引用所指向的对象。

第二种情况：final 的含义与第一种情况相同。

实际上对于前两种情况，一种更贴切的表述 final 的含义的描述，那就是，如果一个变量或方法参数被 final 修饰，就表示它只能被赋值一次，但是 JAVA 虚拟机为变量设定的默认值不记作一次赋值。被 final 修饰的变量必须被初始化。初始化的方式以下几种：

1.在定义的时候初始化。

2.final 变量可以在初始化块中初始化，不可以在静态初始化块中初始化。

3.静态 final 变量可以在定义时初始化，也可以在静态初始化块中初始化，不可以在初始化块中初始化。

4.final 变量还可以在类的构造器中初始化，但是静态 final 变量不可以。

我们运行上面的代码之后出了可以发现 final 变量（常量和静态 final 变量（静态常量被初始化时，编译会报错。

用 final 修饰的变量（常量比非 final 的变量（普通变量拥更高的效率，因此我们在际编程中应该尽可能多的用常量来代替普通变量。

定义方法

当 final 用来定义一个方法时，它表示这个方法不可以被子类重写，但是并不影响它被子类继承。

这里需要特殊说明的是，具有 private 访问权限的方法也可以增加 final 修饰，但是由于子类无法继承 private 方法，因此也无法重写它。编译器在处理 private 方法时，是照 final 方来对待的，这样可以提高该方法被调用时的效率。不过子类仍然可以定义同父类中 private 方法具同样结构的方法，但是这并不会产生重写的效果，而且它们之间也不存在必然联系。

定义类

最后我们再来回顾一下 final 用于类的情况。这个大家应该也很熟悉了，因为我们最常用的 String 类就是 final 的。由于 final 类不允许被继承，编译器在处理时把它的所方法都当作 final 的，因此 final 类比普通类拥更高的效率。而由关键字 abstract 定义的抽象类含必须由继承自它的子类重载实现的抽象方法，因此无法同时用 final 和 abstract 来修饰同一个类。同样的道理，

final 也不能用来修饰接口。 final 的类的所方法都不能被重写，但这并不表示 final 的类的属性（变量值也是不可改变的，要想做到 final 类的属性值不可改变，必须给它增加 final 修饰。

finally 语句

接下来我们一起回顾一下 finally 的用法。finally 只能用在 try/catch 语句中并且附带着一个语句块，表示这段语句最终总是被执行。

运行结果说明了 finally 的作用：

1.程序抛出了异常

2.执行了 finally 语句块请大家注意，捕获程序抛出的异常之后，既不加处理，也不继续向上抛出异常，并不是良好的编程习惯，它掩盖了程序执行中发生的错误，这里只是方便演示，请不要学习。

那么，没一种情况使 finally 语句块得不到执行呢？

return、continue、break 这个可以打乱代码顺序执行语句的规律。那我们就来试试看，这个语句是否能影响 finally 语句块的执行：

很明显，return、continue 和 break 都没能阻止 finally 语句块的执行。从输出的结果来看，return 语句似乎在 finally 语句块之前执行了，事实真的如此吗？我们来想想看，return 语句的作用是什么呢？是退出当前的方法，并将值或对象返回。如果 finally 语句块是在 return 语句之后执行的，那么 return 语句被执行后就已经退出当前方法了，finally 语句块又如何能被执行呢？因此，正确的执行顺序应该是这样的：编译器在编译 return new ReturnClass();时，将它分成了两个步骤，new ReturnClass()和 return，前一个创建对象的语句是在 finally 语句块之前被执行的，而后一个 return 语句是在 finally 语句块之后执行的，也就是说 finally 语句块是在程序退出方法之前被执行的。同样，finally 语句块是在循环被跳过（continue 和中断（break 之前被执行的

finalize 方法

最后，我们再来看看 finalize，它是一个方法，属于 java.lang.Object 类，它的定义如下：protected void finalize()throws Throwable{}众所周知，finalize()方法是 GC（garbagecollector 运行机制的一部分,在此我们只说说 finalize()方法的作用是什么呢？finalize()方法是在 GC 清理它所从属的对象时被调用的，如果执行它的过程中抛出了无法捕获的异常（uncaughtexception，GC 将终止对改对象的清理，并且该异常会被忽略；直到下一次 GC 开始清理这个对象时，它的 finalize()会被再次调用。

序调用了 java.lang.System 类的 gc()方法，引起 GC 的执行，GC 在清理 ft 对象时调用了它的 finalize()方法，因此才了上面的输出结果。调用 System.gc()等同于调用下面这行代码：Runtime.getRuntime().gc();调用它们的作用只是建议垃圾收集器（GC 启动，清理无用的对象释放内存空间，但是 GC 的启动并不是一定的，这由 JAVA 虚拟机来决定。直到 JAVA 虚拟机停止运行，些对象的 finalize()可能都没被运行过，那么怎样保证所对象的这个方法在 JAVA 虚拟机停止运行之前一定被调用呢？

给这个方法传入 true 就可以保证对象的 finalize()方法在 JAVA 虚拟机停止运行前一定被运行了，不过遗憾的是这个方法是不安全的，它会导致有用的对象 finalize()被误调用，因此已不被赞成使用了。由于 finalize()属于 Object 类，因此所类都这个方法，Object 的任意子类都可以重写（override 该方法，在其中释放系统资源或者做其它的清理工作，如关闭输入输出流。通过以上知识的回顾，我想大家对于 final、finally、finalize 的用法区别已经很清楚了。

7、强引用 、软引用、 弱引用、虚引用

Java 四种引用包括强引用，软引用，弱引用，虚引用。

强引用：

只要引用存在，垃圾回收器永远不会回收

```
Object obj = new Object();
```

//可直接通过 obj 取得对应的对象 如 obj.equels(new Object());

而这样 obj 对象对后面 new Object 的一个强引用，只有当 obj 这个引用被释放之后，对象才会被释放掉，这也是我们经常所用到的编码形式。

软引用：

非必须引用，内存溢出之前进行回收，可以通过以下代码实现

```
Object obj = new Object();
SoftReference<Object> sf = new SoftReference<Object>(obj);
obj = null;
```

sf.get();//有时候会返回 null

这时候 sf 是对 obj 的一个软引用，通过 sf.get()方法可以取到这个对象，当然，当这个对象被标记为需要回收的对象时，则返回 null；

软引用主要用户实现类似缓存的功能，在内存足够的情况下直接通过软引用取值，无需从繁忙的真实来源查询数据，提升速度；当内存不足时，自动删除这部分缓存数据，从真正的来源查询这些数据。

弱引用：

第二次垃圾回收时回收，可以通过如下代码实现

```
Object obj = new Object();
WeakReference<Object> wf = new WeakReference<Object>(obj);
obj = null;
wf.get();//有时候会返回 null
wf.isEnQueued();//返回是否被垃圾回收器标记为即将回收的垃圾
```

弱引用是在第二次垃圾回收时回收，短时间内通过弱引用取对应的数据，可以取到，当执行过第二次垃圾回收时，将返回 null。

弱引用主要用于监控对象是否已经被垃圾回收器标记为即将回收的垃圾，可以通过弱引用的 isEnQueued 方法返回对象是否被垃圾回收器标记。

虚引用：

垃圾回收时回收，无法通过引用取到对象值，可以通过如下代码实现

```
Object obj = new Object();
PhantomReference<Object> pf = new PhantomReference<Object>(obj);
obj=null;
pf.get();//永远返回 null
pf.isEnQueued();//返回是否从内存中已经删除
```

虚引用是每次垃圾回收的时候都会被回收，通过虚引用的 get 方法永远获取到的数据为 null，因此也被成为幽灵引用。

虚引用主要用于检测对象是否已经从内存中删除。

对象的强、软、弱和虚引用

在 JDK 1.2 以前的版本中，若一个对象不被任何变量引用，那么程序就无法再使用这个对象。也就是说，只有对象处于可触及（reachable）状态，程序才能使用它。从 JDK 1.2 版本开始，把对象的引用分为 4 种级别，从而使程序能更加灵活地控制对象的生命周期。这 4 种级别由高到低依次为：强引用、软引用、弱引用和虚引用。

⑴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。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。 ps：强引用其实也就是我们平时 A a = new A()这个意思。

⑵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（下文给出示例）。

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。

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。

```
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：

```
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。

⑶弱引用（WeakReference）

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。

如果这个对象是偶尔的使用，并且希望在使用时随时就能获取到，但又不想影响此对象的垃圾收集，那么你应该用 Weak Reference 来记住此对象。

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。

⑷虚引用（PhantomReference）

“虚引用”顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。

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之前，把这个虚引用加入到与之关联的引用队列中。

```
ReferenceQueue queue = new ReferenceQueue ();
PhantomReference pr = new PhantomReference (object, queue);
```

程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。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被回收之前采取必要的行动。

***：

使用 ReferenceQueue 清除失去了软引用对象的 SoftReference

作为一个 Java 对象，SoftReference 对象除了具有保存软引用的特殊性之外，也具有 Java 对象的一般性。所以，当软可及对象被回收之后，虽然这个 SoftReference 对象的 get()方法返回 null,但这个 SoftReference 对象已经不再具有存在的价值，需要一个适当的清除机制，避免大量 SoftReference 对象带来的内存泄漏。在 java.lang.ref 包里还提供了 ReferenceQueue。如果在创建 SoftReference 对象的时候，使用了一个 ReferenceQueue 对象作为参数提供给 SoftReference 的构造方法，如:

```
ReferenceQueue queue = new
ReferenceQueue();
SoftReference
ref=new
SoftReference(aMyObject, queue);
```

那么当这个 SoftReference 所软引用的 aMyOhject 被垃圾收集器回收的同时，ref 所强引用的 SoftReference 对象被列入 ReferenceQueue。也就是说，ReferenceQueue 中保存的对象是 Reference 对象，而且是已经失去了它所软引用的对象的 Reference 对象。另外从 ReferenceQueue 这个名字也可以看出，它是一个队列，当我们调用它的 poll()方法的时候，如果这个队列中不是空队列，那么将返回队列前面的那个 Reference 对象。

在任何时候，我们都可以调用 ReferenceQueue 的 poll()方法来检查是否有它所关心的非强可及对象被回收。如果队列为空，将返回一个 null,否则该方法返回队列中前面的一个 Reference 对象。利用这个方法，我们可以检查哪个 SoftReference 所软引用的对象已经被回收。于是我们可以把这些失去所软引用的对象的 SoftReference 对象清除掉。常用的方式为:

```
SoftReference ref = null;
while ((ref = (EmployeeRef) q.poll()) != null) {
// 清除 ref
}
```

8、Java 反射

1、什么是 Java 类中的反射？

当程序运行时，允许改变程序结构或变量类型，这种语言称为动态语言。我们认为 Java 并不是动态语言，但是它却又一个非常突出的动态相关的机制，俗称：反射。

Reflection 是 Java 程序开发语言的特征之一，它允许运行中的 Java 程序获取自身的信息，并且可以操作类和对象的内部属性。

通过反射，我们可以在运行时获得程序或程序集中每一个类型成员和成员变量的信息。

程序中一般的对象类型都是在编译期就确定下来的，而 Java 反射机制可以动态的创建对象并调用其属性，这样对象的类型在编译期是未知的。所以我们可以通过反射机制直接创建对象即使这个对象在编译期是未知的，

反射的核心：是 JVM 在运行时 才动态加载的类或调用方法或属性，他不需要事先（写代码的时候或编译期）知道运行对象是谁。

一、Java 反射框架主要提供以下功能：

1.在运行时判断任意一个对象所属的类；

2.在运行时构造任意一个类的对象；

3.在运行时判断任意一个类所具有的成员变量和方法（通过反射甚至可以调用 private 方法）；

4.在运行时调用任意一个对象的方法

二、主要用途 ：

1、反射最重要的用途就是开发各种通用框架。

很多框架（比如 Spring）都是配置化的（比如通过 XML 文件配置 JavaBean，Action 之类的），为了保证框架的通用性，他们可能根据配置文件加载不同的对象或类，调用不同的方法，这个时候就必须用到反射——运行时动态加载需要加载的对象。

三、基本反射功能的实现(反射相关的类一般都在 java.lang.relfect 包里)：

1、获得 Class 对象

使用 Class 类的 forName 静态方法

```
Class.forName(driver)
```

直接获取某一个对象的 class

```
Class<?> klass = int.class;
Class<?> classInt = Integer.TYPE;
```

调用某个对象的 getClass()方法

```
StringBuilder str = new StringBuilder("123");
Class<?> klass = str.getClass();
```

2、判断是否为某个类的实例

```
public native boolean isInstance(Object obj);
```

用 instanceof 关键字来判断是否为某个类的实例

3、创建实例

使用 Class 对象的 newInstance()方法来创建 Class 对象对应类的实例。

```
Class<?> c    = String.calss;
Object str = c.getInstance();
```

先通过 Class 对象获取指定的 Constructor 对象，再调用 Constructor 对象的 newInstance()方法来创建实例。

```
//获取 String 所对应的 Class 对象
    Class<?> c = String.class;
  //获取 String 类带一个 String 参数的构造器
  Constructor constructor = c.getConstructor(String.class);
  //根据构造器创建实例
  Object obj = constructor.newInstance("23333");
  System.out.println(obj);
```

4、获取方法

getDeclaredMethods()

5、获取构造器信息

getDeclaredMethods()

getMethods()

getMethod()

6、获取类的成员变量（字段）信息

getFiled: 访问公有的成员变量

getDeclaredField：所有已声明的成员变量。但不能得到其父类的成员变量

getFileds 和 getDeclaredFields 用法

7、调用方法

invoke()

8、利用反射创建数组

反射缺点:

由于反射会额外消耗一定的系统资源，因此如果不需要动态地创建一个对象，那么就不需要用反射。

另外，反射调用方法时可以忽略权限检查，因此可能会破坏封装性而导致安全问题。

9、List 和 map 的区别

List 是继承自 Collection 接口，Map 是 Map 接口。

1、List 是存储单列数据的集合，map 是存储键和值这样的双列数据的集合，List 中存储的数据是有顺序，并且允许重复；

2、Map 中存储的数据是没有顺序的，其键是不能重复的，它的值是可以有重复

List 特点：元素有放入顺序，元素可重复;

Map 特点：元素按键值对存储，无放入顺序 ;

List 接口有三个实现类：LinkedList，ArrayList，Vector;

LinkedList：底层基于链表实现，链表内存是散乱的，每一个元素存储本身内存地址的同时还存储下一个元素的地址。链表增删快，查找慢;

Map 接口有三个实现类：HashMap，HashTable，LinkeHashMap

Map 相当于和 Collection 一个级别的；Map 该集合存储键值对，且要求保持键的唯一性。

10、Arraylist 与 LinkedList 区别

1) 因为 Array 是基于索引(index)的数据结构，它使用索引在数组中搜索和读取数据是很快的。Array 获取数据的时间复杂度是 O(1),但是要删除数据却是开销很大的，因为这需要重排数组中的所有数据。

2) 相对于 ArrayList，LinkedList 插入是更快的。因为 LinkedList 不像 ArrayList 一样，不需要改变数组的大小，也不需要在数组装满的时候要将所有的数据重新装入一个新的数组，这是 ArrayList 最坏的一种情况，时间复杂度是 O(n)，而 LinkedList 中插入或删除的时间复杂度仅为 O(1)。ArrayList 在插入数据时还需要更新索引（除了插入数组的尾部）。

3) 类似于插入数据，删除数据时，LinkedList 也优于 ArrayList。

4) LinkedList 需要更多的内存，因为 ArrayList 的每个索引的位置是实际的数据，而 LinkedList 中的每个节点中存储的是实际的数据和前后节点的位置。

5) 你的应用不会随机访问数据。因为如果你需要 LinkedList 中的第 n 个元素的时候，你需要从第一个元素顺序数到第 n 个数据，然后读取数据。

6) 你的应用更多的插入和删除元素，更少的读取数据。因为插入和删除元素不涉及重排数据，所以它要比 ArrayList 要快。

11、 ArrayList 与 Vector 区别

1） 同步性:Vector 是线程安全的，也就是说是同步的 ，而 ArrayList 是线程序不安全的，不是同步的 数 2。

2）数据增长:当需要增长时,Vector 默认增长为原来一倍 ，而 ArrayList 却是原来的 50% ，这样,ArrayList 就有利于节约内存空间。

如果涉及到堆栈，队列等操作，应该考虑用 Vector，如果需要快速随机访问元素，应该使用 ArrayList 。

12、HashMap 和 Hashtable 的区别

1)HashMap 几乎可以等价于 Hashtable，除了 HashMap 是非 synchronized 的，并可以接受 null(HashMap 可以接受为 null 的键值(key)和值(value)，而 Hashtable 则不行)。

2) HashMap 是非 synchronized，而 Hashtable 是 synchronized，这意味着 Hashtable 是线程安全的，多个线程可以共享一个 Hashtable；而如果没有正确的同步的话，多个线程是不能共享 HashMap 的。Java 5 提供了 ConcurrentHashMap，它是 HashTable 的替代，比 HashTable 的扩展性更好。

3) 另一个区别是 HashMap 的迭代器(Iterator)是 fail-fast 迭代器，而 Hashtable 的 enumerator 迭代器不是 fail-fast 的。所以当有其它线程改变了 HashMap 的结构（增加或者移除元素），将会抛出 ConcurrentModificationException，但迭代器本身的 remove()方法移除元素则不会抛出 ConcurrentModificationException 异常。但这并不是一个一定发生的行为，要看 JVM。这条同样也是 Enumeration 和 Iterator 的区别。

4) 由于 Hashtable 是线程安全的也是 synchronized，所以在单线程环境下它比 HashMap 要慢。如果你不需要同步，只需要单一线程，那么使用 HashMap 性能要好过 Hashtable。

5) HashMap 不能保证随着时间的推移 Map 中的元素次序是不变的。

13、HashSet 和 HashMap 区别

HashSet：

HashSet 是对 HashMap 的简单包装，对 HashSet 的函数调用都会转换成合适的 HashMap 方法。

HashSet 实现了 Set 接口，它不允许集合中出现重复元素。当我们提到 HashSet 时，第一件事就是在将对象存储在 HashSet 之前，要确保重写 hashCode（）方法和 equals（）方法，这样才能比较对象的值是否相等，确保集合中没有储存相同的对象。如果不重写上述两个方法，那么将使用下面方法默认实现：public boolean add(Object obj)方法用在 Set 添加元素时，如果元素值重复时返回 "false"，如果添加成功则返回"true"

HashMap：

HashMap 实现了 Map 接口，Map 接口对键值对进行映射。Map 中不允许出现重复的键（Key）。Map 接口有两个基本的实现 TreeMap 和 HashMap。TreeMap 保存了对象的排列次序，而 HashMap 不能。HashMap 可以有空的键值对（Key（null）-Value（null））。

HashMap 是非线程安全的（非 Synchronize），要想实现线程安全，那么需要调用 collections 类的静态方法 synchronizeMap（）实现。public Object put(Object Key,Object value)方法用来将元素添加到 map 中。

14、HashMap 和 ConcurrentHashMap 的区别

（1）放入 HashMap 的元素是 key-value 对。

（2）底层说白了就是以前数据结构课程讲过的散列结构。

（3）要将元素放入到 hashmap 中，那么 key 的类型必须要实现实现 hashcode 方法，默认这个方法是根据对象的地址来计算的，具体我也记不太清楚了，接着还必须覆盖对象的 equal 方法。

（4）ConcurrentHashMap 对整个桶数组进行了分段，而 HashMap 则没有

（5）ConcurrentHashMap 在每一个分段上都用锁进行保护，从而让锁的粒度更精细一些，并发性能更好，而 HashMap 没有锁机制，不是线程安全的。。。

15、HashMap 的工作原理及代码实现

HashMap 基于 hashing 原理，我们通过 put()和 get()方法储存和获取对象。当我们将键值对传递给 put()方法时，它调用键对象的 hashCode()方法来计算 hashcode，让后找到 bucket 位置来储存值对象。当获取对象时，通过键对象的 equals()方法找到正确的键值对，然后返回值对象。HashMap 使用链表来解决碰撞问题，当发生碰撞了，对象将会储存在链表的下一个节点中。 HashMap 在每个链表节点中储存键值对对象。

1.HashMap 介绍

HashMap 为 Map 接口的一个实现类，实现了所有 Map 的操作。HashMap 除了允许 key 和 value 保存 null 值和非线程安全外，其他实现几乎和 HashTable 一致。

HashMap 使用散列存储的方式保存 kay-value 键值对，因此其不支持数据保存的顺序。如果想要使用有序容器可以使用 LinkedHashMap。

在性能上当 HashMap 中保存的 key 的哈希算法能够均匀的分布在每个 bucket 中的是时候，HashMap 在基本的 get 和 set 操作的的时间复杂度都是 O(n)。

在遍历 HashMap 的时候，其遍历节点的个数为 bucket 的个数+HashMap 中保存的节点个数。因此当遍历操作比较频繁的时候需要注意 HashMap 的初始化容量不应该太大。 这一点其实比较好理解：当保存的节点个数一致的时候，bucket 越少，遍历次数越少。

另外 HashMap 在 resize 的时候会有很大的性能消耗，因此当需要在保存 HashMap 中保存大量数据的时候，传入适当的默认容量以避免 resize 可以很大的提高性能。 具体的 resize 操作请参考下面对此方法的分析

HashMap 是非线程安全的类，当作为共享可变资源使用的时候会出现线程安全问题。需要使用线程安全容器：

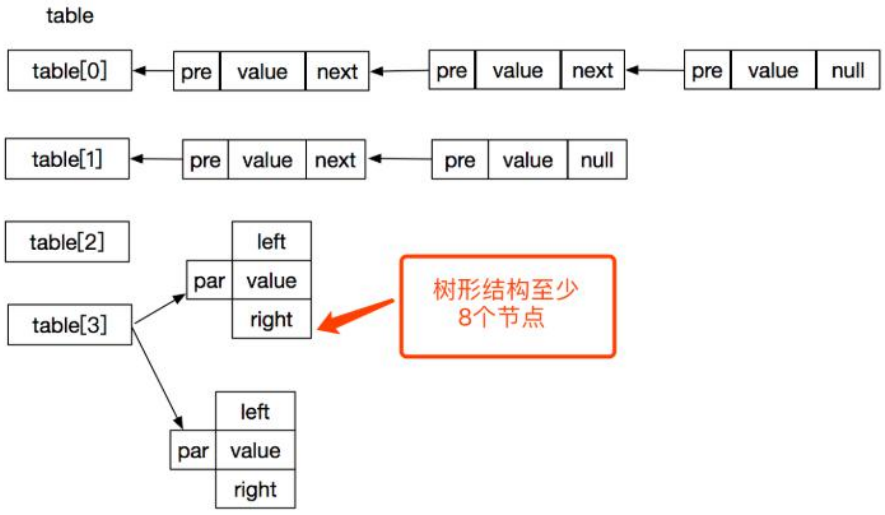
Map m = new ConcurrentHashMap();或者

Map m = Collections.synchronizedMap(new HashMap());

具体的 HashMap 会出现的线程安全问题分析请参考 9 中的分析。

2.数据结构介绍

HashMap 使用数组+链表+树形结构的数据结构。其结构图如下所示。



3.HashMap 源码分析（基于 JDK1.8）

3.1 关键属性分析

transient Node<K,V>[] table; //Node 类型的数组，记我们常说的 bucket 数组，其中每个元素为链表或者树形结构

transient int size;//HashMap 中保存的数据个数

int threshold;//HashMap 需要 resize 操作的阈值

final float loadFactor;//负载因子，用于计算 threshold。计算公式为：threshold = loadFactor * capacity

其中还有一些默认值得属性，有默认容量 2^4，默认负载因子 0.75 等.用于构造函数没有指定数值情况下的默认值。

3.2 构造函数分析

HashMap 提供了三个不同的构造函数，主要区别为是否传入初始化容量和负载因子。分别文以下三个。

```
//此构造函数创建一个空的 HashMap，其中负载因子为默认值 0.75
public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
}
//传入默认的容量大小，创造一个指定容量大小和默认负载因子为 0.75 的 HashMap
 public HashMap(int initialCapacity) {
    this(initialCapacity, DEFAULT_LOAD_FACTOR);
}
//创建一个指定容量和指定负载因为 HashMap，以下代码删除了入参检查
public HashMap(int initialCapacity, float loadFactor) {
    this.loadFactor = loadFactor;
    this.threshold = tableSizeFor(initialCapacity);
}
```

注意：此处的 initialCapacity 为数组 table 的大小，即 bucket 的个数。

其中在指定初始化容量的时候，会根据传入的参数来确定 HashMap 的容量大小。

初始化 this.threshold 的值为入参 initialCapacity 距离最近的一个 2 的 n 次方的值。取值方法如下：

case initialCapacity = 0:

this.threshold = 1；

case initialCapacity 为非 0 且不为 2 的 n 次方:

this.threshold = 大于 initialCapacity 中第一个 2 的 n 次方的数。

case initialCapacity = 2^n：

this.threshold = initialCapacity

具体的计算方法为 tableSizeFor(int cap)函数。计算方法是将入参的最高位下面的所有位都设置为 1，然后加 1

下面以入参为 134217729 为例分析计算过程。

首先将 int 转换为二进制如下：

cap = 0000 1000 0000 0000 0000 0000 0000 0001

另外此处赋值为 this.threshold，是因为构造函数的时候并不会创建 table，只有实际插入数据的时候才会创建。目的应该是为了节省内存空间吧。

在第一次插入数据的时候，会将 table 的 capacity 设置为 threshold，同时将 threshold 更新为 loadFactor * capacity

3.3 关键函数源码分析

3.3.1 第一次插入数据的操作

HashMap 在插入数据的时候传入 key-value 键值对。使用 hash 寻址确定保存数据的

bucket。当第一次插入数据的时候会进行 HashMap 中容器的初始化。具体操作如下：

```
Node<K,V>[] tab;
        int n, i;
        if ((tab = table) == null || (n = tab.length) == 0)
            n = (tab = resize()).length;
```

其中 resize 函数的源码如下，主要操作为根据 cap 和 loadFactory 创建初始化 table

```
Node<K, V>[] oldTab = table;
    int oldThr = threshold;   //oldThr 根据传入的初始化 cap 决定 2 的 n 次方
    int newCap, newThr = 0;
    if (oldThr > 0) // 当构造函数中传入了 capacity 的时候
        newCap = oldThr;   //newCap = threshold   2 的 n 次方，即构造函数的时候的初始化容量
     else {                        // zero initial threshold signifies using defaults
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
    float ft = (float)newCap * loadFactor;   // 2 的 n 次方  * loadFactory
    newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
            (int)ft : Integer.MAX_VALUE);
    threshold = newThr; //新的 threshold== newCap * loadFactory
    Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap]; //长度为 2 的 n 次方的数组
    table = newTab;
```

在初始化 table 之后，将数据插入到指定位置，其中 bucket 的确定方法为：

```
i = (n - 1) & hash // 此处 n-1 必定为 0000 1111 1111....的格式，取&操作之后的值一定在数组的容量范围内。
```

其中 hash 的取值方式为：

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

具体操作如下，创建 Node 并将 node 放到 table 的第 i 个元素中

```
if ((p = tab[i = (n - 1) & hash]) == null)
    tab[i] = new Node(hash, key, value, null);
```

3.3.2 非第一次插入数据的操作源码分析

当 HashMap 中已有数据的时候，再次插入数据，会多出来在链表或者树中寻址的操作，和当 size 到达阈值时候的 resize 操作。多出来的步骤如下：

另外，在 resize 操作中也和第一次插入数据的操作不同，当 HashMap 不为空的时候 resize 操作需要将之前的数据节点复制到新的 table 中。操作如下：

3.4Cloneable 和 Serializable 分析

在 HashMap 的定义中实现了 Cloneable 接口，Cloneable 是一个标识接口，主要用来标识 Object.clone()的合法性，在没有实现此接口的实例中调用 Object.clone()方法会抛出 CloneNotSupportedException 异常。可以看到 HashMap 中重写了 clone 方法。

HashMap 实现 Serializable 接口主要用于支持序列化。同样的 Serializable 也是一个标识接口，本身没有定义任何方法和属性。另外 HashMap 自定义了

private void writeObject(java.io.ObjectOutputStream s) throws IOException
private void readObject(java.io.ObjectInputStream s) throws IOException,
ClassNotFoundException
两个方法实现了自定义序列化操作。
注意：支持序列化的类必须有无参构造函数。这点不难理解，反序列化的过程中需要通过反射创建对象。
4.HashMap 的遍历
以下讨论两种遍历方式，测试代码如下：
方法一：
通过 map.keySet()获取 key 的集合，然后通过遍历 key 的集合来遍历 map
方法二：
通过 map.entrySet()方法获取 map 中节点集合，然后遍历此集合遍历 map
测试代码如下：

```
public static void main(String[] args) throws Exception {
        Map<String, Object> map = new HashMap<>();
        map.put("name", "test");
        map.put("age", "25");
        map.put("address", "HZ");
        Set<String> keySet = map.keySet();
        for (String key : keySet) {
            System.out.println(map.get(key));
        }
        Set<Map.Entry<String, Object>> set = map.entrySet();
        for (Map.Entry<String, Object> entry : set) {
            System.out.println("key is : " + entry.getKey() + ".    value is " + entry.getValue());
        }
    }
```

16、ConcurrentHashMap 的工作原理及代码实现

ConcurrentHashMap 采用了非常精妙的"分段锁"策略，ConcurrentHashMap 的主干是个 Segment 数组。Segment 继承了 ReentrantLock，所以它就是一种可重入锁（ReentrantLock)。在 ConcurrentHashMap，一个 Segment 就是一个子哈希表，Segment 里维护了一个 HashEntry 数组，并发环境下，对于不同 Segment 的数据进行操作是不用考虑锁竞争的。

ConcurrentHashMap 是 Java1.5 中引用的一个线程安全的支持高并发的 HashMap 集合类。

1、线程不安全的 HashMap

因为多线程环境下，使用 Hashmap 进行 put 操作会引起死循环，导致 CPU 利用率接近 100%，所以在并发情况下不能使用 HashMap。

2、效率低下的 HashTable

HashTable 容器使用 synchronized 来保证线程安全，但在线程竞争激烈的情况下 HashTable 的效率非常低下。

因为当一个线程访问 HashTable 的同步方法时，其他线程访问 HashTable 的同步方法时，可能会进入阻塞或轮询状态。

如线程 1 使用 put 进行添加元素，线程 2 不但不能使用 put 方法添加元素，并且也不能使用 get 方法来获取元素，所以竞争越激烈效率越低。

3、锁分段技术

HashTable 容器在竞争激烈的并发环境下表现出效率低下的原因，是因为所有访问 HashTable 的线程都必须竞争同一把锁，

那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。

有些方法需要跨段，比如 size()和 containsValue()，它们可能需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。

这里“按顺序”是很重要的，否则极有可能出现死锁，在 ConcurrentHashMap 内部，段数组是 final 的，并且其成员变量实际上也是 final 的，

但是，仅仅是将数组声明为 final 的并不保证数组成员也是 final 的，这需要实现上的保证。这可以确保不会出现死锁，因为获得锁的顺序是固定的。

oncurrentHashMap 类中包含两个静态内部类 HashEntry 和 Segment。

HashEntry 用来封装映射表的键 / 值对;Segment 用来充当锁的角色，每个 Segment 对象守护整个散列映射表的若干个桶。

每个桶是由若干个 HashEntry 对象链接起来的链表。一个 ConcurrentHashMap 实例中包含由若干个 Segment 对象组成的数组。

每个 Segment 守护者一个 HashEntry 数组里的元素,当对 HashEntry 数组的数据进行修改时，必须首先获得它对应的 Segment 锁。

5、HashEntry 类

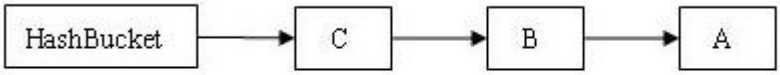
```
static final class HashEntry<K,V> {
        final K key;                        // 声明 key 为 final 型
        final int hash;                     // 声明 hash 值为 final 型
        volatile V value;                   // 声明 value 为 volatile 型
        final HashEntry<K,V> next;   // 声明 next 为 final 型


        HashEntry(K key, int hash, HashEntry<K,V> next, V value)   {
            this.key = key;
            this.hash = hash;
            this.next = next;
            this.value = value;
        }
 }
```

每个 HashEntry 代表 Hash 表中的一个节点，在其定义的结构中可以看到，除了 value 值没有定义 final，其余的都定义为 final 类型，我们知道 Java 中关键词 final 修饰的域成为最终域。

用关键词 final 修饰的变量一旦赋值，就不能改变，也称为修饰的标识为常量。这就意味着我们删除或者增加一个节点的时候，就必须从头开始重新建立 Hash 链，因为 next 引用值需要改变。

由于 HashEntry 的 next 域为 final 型，所以新节点只能在链表的表头处插入。 例如将 A,B,C 插入空桶中，插入后的结构为：



注意：由于只能在表头插入，所以链表中节点的顺序和插入的顺序相反。

6、segment 类

```
static final class Segment<K,V> extends ReentrantLock implements Serializable {
  private static final long serialVersionUID = 2249069246763182397L;
        /**
         * 在本 segment 范围内，包含的 HashEntry 元素的个数
         * 该变量被声明为 volatile 型,保证每次读取到最新的数据
         */
        transient volatile int count;


        /**
         *table 被更新的次数
         */
        transient int modCount;


        /**
         * 当 table 中包含的 HashEntry 元素的个数超过本变量值时，触发 table 的再散列
         */
        transient int threshold;


        /**
         * table 是由 HashEntry 对象组成的数组
         * 如果散列时发生碰撞，碰撞的 HashEntry 对象就以链表的形式链接成一个链表
         * table 数组的数组成员代表散列映射表的一个桶
         * 每个 table 守护整个 ConcurrentHashMap 包含桶总数的一部分
         * 如果并发级别为 16，table 则守护 ConcurrentHashMap 包含的桶总数的 1/16
         */
        transient volatile HashEntry<K,V>[] table;


        /**
         * 装载因子
         */
```

```
        final float loadFactor;
    }
```

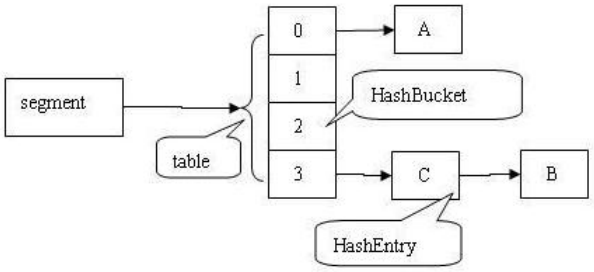
Segment 类继承于 ReentrantLock 类，从而使得 Segment 对象能充当锁的角色。每个 Segment 对象用来守护其（成员对象 table 中）包含的若干个桶。

table 是一个由 HashEntry 对象组成的数组。table 数组的每一个数组成员就是散列映射表的一个桶。

每一个 Segment 对象都有一个 count 对象来表示本 Segment 中包含的 HashEntry 对象的总数。

之所以在每个 Segment 对象中包含一个计数器，而不是在 ConcurrentHashMap 中使用全局的计数器，是为了避免出现“热点域”而影响 ConcurrentHashMap 的并发性。

下图是依次插入 ABC 三个 HashEntry 节点后，Segment 的结构示意图。



7、ConcurrentHashMap 类

默认的情况下，每个 ConcurrentHashMap 类会创建 16 个并发的 segment，每个 segment 里面包含多个 Hash 表，每个 Hash 链都是有 HashEntry 节点组成的。

如果键能均匀散列，每个 Segment 大约守护整个散列表中桶总数的 1/16。

```
    public class ConcurrentHashMap<K, V> extends AbstractMap<K, V>
            implements ConcurrentMap<K, V>, Serializable {


         /**
         * 散列映射表的默认初始容量为 16，即初始默认为 16 个桶
         * 在构造函数中没有指定这个参数时，使用本参数
         */
        static final        int DEFAULT_INITIAL_CAPACITY= 16;

        /**
         * 散列映射表的默认装载因子为 0.75，该值是 table 中包含的 HashEntry 元素的个数与
    * table 数组长度的比值
         * 当 table 中包含的 HashEntry 元素的个数超过了 table 数组的长度与装载因子的乘积时，
    * 将触发 再散列
         * 在构造函数中没有指定这个参数时，使用本参数
         */
        static final float DEFAULT_LOAD_FACTOR= 0.75f;
```

```
    /**
     * 散列表的默认并发级别为 16。该值表示当前更新线程的估计数
     * 在构造函数中没有指定这个参数时，使用本参数
     */
    static final int DEFAULT_CONCURRENCY_LEVEL= 16;

    /**
     * segments 的掩码值
     * key 的散列码的高位用来选择具体的 segment
     */
    final int segmentMask;

    /**
     * 偏移量
     */
    final int segmentShift;

    /**
     * 由 Segment 对象组成的数组
     */
    final Segment<K,V>[] segments;

    /**
     * 创建一个带有指定初始容量、加载因子和并发级别的新的空映射。
     */
    public ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel) {
        if(!(loadFactor > 0) || initialCapacity < 0 ||
concurrencyLevel <= 0)
            throw new IllegalArgumentException();

        if(concurrencyLevel > MAX_SEGMENTS)
            concurrencyLevel = MAX_SEGMENTS;

        // 寻找最佳匹配参数（不小于给定参数的最接近的 2 次幂）
        int sshift = 0;
        int ssize = 1;
        while(ssize < concurrencyLevel) {
            ++sshift;
            ssize <<= 1;
        }
        segmentShift = 32 - sshift;          // 偏移量值
        segmentMask = ssize - 1;             // 掩码值
        this.segments = Segment.newArray(ssize);    // 创建数组
```

```
            if (initialCapacity > MAXIMUM_CAPACITY)
                initialCapacity = MAXIMUM_CAPACITY;
            int c = initialCapacity / ssize;
            if(c * ssize < initialCapacity)
                ++c;
            int cap = 1;
            while(cap < c)
                cap <<= 1;

            // 依次遍历每个数组元素
            for(int i = 0; i < this.segments.length; ++i)
                // 初始化每个数组元素引用的 Segment 对象
    this.segments[i] = new Segment<K,V>(cap, loadFactor);
        }

        /**
         * 创建一个带有默认初始容量 (16)、默认加载因子 (0.75) 和 默认并发级别 (16)
     * 的空散列映射表。
         */
        public ConcurrentHashMap() {
            // 使用三个默认参数，调用上面重载的构造函数来创建空散列映射表
    this(DEFAULT_INITIAL_CAPACITY,                    DEFAULT_LOAD_FACTOR,
DEFAULT_CONCURRENCY_LEVEL);
    }
```

8、 用分离锁实现多个线程间的并发写操作

插入数据后的 ConcurrentHashMap 的存储形式

（1）Put 方法的实现

首先，根据 key 计算出对应的 hash 值：

```
public V put(K key, V value) {
    if (value == null)              //ConcurrentHashMap 中不允许用 null 作为映射值
        throw new NullPointerException();
    int hash = hash(key.hashCode());          // 计算键对应的散列码
    // 根据散列码找到对应的 Segment
    return segmentFor(hash).put(key, hash, value, false);
}
```

根据 hash 值找到对应的 Segment：

```
/**
* 使用 key 的散列码来得到 segments 数组中对应的 Segment
*/
final Segment<K,V> segmentFor(int hash) {
// 将散列值右移 segmentShift 个位，并在高位填充 0
// 然后把得到的值与 segmentMask 相“与”
// 从而得到 hash 值对应的 segments 数组的下标值
// 最后根据下标值返回散列码对应的 Segment 对象
```

```
        return segments[(hash >>> segmentShift) & segmentMask];
    }
```

在这个 Segment 中执行具体的 put 操作：

```
        V put(K key, int hash, V value, boolean onlyIfAbsent) {
            lock();  // 加锁，这里是锁定某个 Segment 对象而非整个 ConcurrentHashMap
            try {
                int c = count;

                if (c++ > threshold)        // 如果超过再散列的阈值
                    rehash();                    // 执行再散列，table 数组的长度将扩充一倍

                HashEntry<K,V>[] tab = table;
                // 把散列码值与 table 数组的长度减 1 的值相“与”
                // 得到该散列码对应的 table 数组的下标值
                int index = hash & (tab.length - 1);
                // 找到散列码对应的具体的那个桶
                HashEntry<K,V> first = tab[index];

                HashEntry<K,V> e = first;
                while (e != null && (e.hash != hash || !key.equals(e.key)))
                    e = e.next;

                V oldValue;
                if (e != null) {                    // 如果键 / 值对以经存在
                    oldValue = e.value;
                    if (!onlyIfAbsent)
                        e.value = value;       // 设置 value 值
                }
                else {                                  // 键 / 值对不存在
                    oldValue = null;
                    ++modCount;                // 要添加新节点到链表中，所以 modCont 要
加 1
                    // 创建新节点，并添加到链表的头部
                    tab[index] = new HashEntry<K,V>(key, hash, first, value);
                    count = c;                       // 写 count 变量
                }
                return oldValue;
            } finally {
                unlock();                                // 解锁
            }
        `}
```

这里的加锁操作是针对（键的 hash 值对应的）某个具体的 Segment，锁定的是该 Segment 而不是整个 ConcurrentHashMap。

因为插入键 / 值对操作只是在这个 Segment 包含的某个桶中完成，不需要锁定整个

ConcurrentHashMap。
此时，其他写线程对另外 15 个 Segment 的加锁并不会因为当前线程对这个 Segment 的加锁而阻塞。
同时，所有读线程几乎不会因本线程的加锁而阻塞（除非读线程刚好读到这个 Segment 中某个 HashEntry 的 value 域的值为 null，此时需要加锁后重新读取该值）。
（2）Get 方法的实现

```
V get(Object key, int hash) {
    if(count != 0) {            // 首先读 count 变量
        HashEntry<K,V> e = getFirst(hash);
        while(e != null) {
            if(e.hash == hash && key.equals(e.key)) {
                V v = e.value;
                if(v != null)
                    return v;
                // 如果读到 value 域为 null，说明发生了重排序，加锁后重新读取
                return readValueUnderLock(e);
            }
            e = e.next;
        }
    }
    return null;
}
 V readValueUnderLock(HashEntry<K,V> e) {
     lock();
     try {
         return e.value;
     } finally {
         unlock();
     }
 }
```

ConcurrentHashMap 中的读方法不需要加锁，所有的修改操作在进行结构修改时都会在最后一步写 count 变量，通过这种机制保证 get 操作能够得到几乎最新的结构更新。
（3）Remove 方法的实现

```
V remove(Object key, int hash, Object value) {
    lock(); //加锁
    try{
        int c = count - 1;
        HashEntry<K,V>[] tab = table;
        //根据散列码找到 table 的下标值
        int index = hash & (tab.length - 1);
        //找到散列码对应的那个桶
        HashEntry<K,V> first = tab[index];
        HashEntry<K,V> e = first;
        while(e != null&& (e.hash != hash || !key.equals(e.key)))
```

```
                e = e.next;


            V oldValue = null;
            if(e != null) {
                V v = e.value;
                if(value == null|| value.equals(v)) { //找到要删除的节点
                    oldValue = v;
                    ++modCount;
                    //所有处于待删除节点之后的节点原样保留在链表中
                    //所有处于待删除节点之前的节点被克隆到新链表中
                    HashEntry<K,V> newFirst = e.next;// 待删节点的后继结点
                    for(HashEntry<K,V> p = first; p != e; p = p.next)
                        newFirst = new HashEntry<K,V>(p.key, p.hash,
                                                      newFirst, p.value);
                    //把桶链接到新的头结点
                    //新的头结点是原链表中，删除节点之前的那个节点
                    tab[index] = newFirst;
                    count = c;          //写 count 变量
                }
            }
            return oldValue;
        } finally{
            unlock(); //解锁
        }
    }
```

整个操作是在持有段锁的情况下执行的，空白行之前的行主要是定位到要删除的节点 e。

如果不存在这个节点就直接返回 null，否则就要将 e 前面的结点复制一遍，尾结点指向 e 的下一个结点。

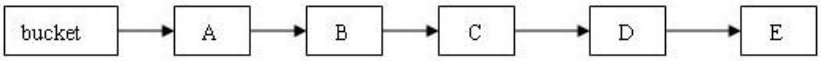
e 后面的结点不需要复制，它们可以重用。

中间那个 for 循环是做什么用的呢？从代码来看，就是将定位之后的所有 entry 克隆并拼回前面去，但有必要吗？

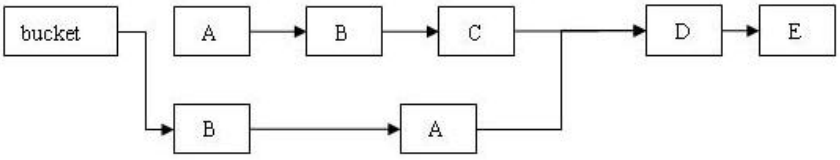
每次删除一个元素就要将那之前的元素克隆一遍？这点其实是由 entry 的不变性来决定的，仔细观察 entry 定义，发现除了 value，其他所有属性都是用 final 来修饰的，

这意味着在第一次设置了 next 域之后便不能再改变它，取而代之的是将它之前的节点全都克隆一次。至于 entry 为什么要设置为不变性，这跟不变性的访问不需要同步从而节省时间有关。

执行删除之前的原链表：



执行删除之后的新链表



注意：新链表在clone的时候。顺序发生反转，A->B变为B->A。

（4）containsKey 方法的实现，它不需要读取值。

```
boolean containsKey(Object key, int hash) {
        if (count != 0) { // read-volatile
            HashEntry<K,V> e = getFirst(hash);
            while (e != null) {
                if (e.hash == hash && key.equals(e.key))
                    return true;
                e = e.next;
            }
        }
        return false;
    }
```

（5）size()

我们要统计整个 ConcurrentHashMap 里元素的大小，就必须统计所有 Segment 里元素的大小后求和。

Segment 里的全局变量 count 是一个 volatile 变量，那么在多线程场景下，我们是不是直接把所有 Segment 的 count 相加就可以得到整个 ConcurrentHashMap 大小了呢？

不是的，虽然相加时可以获取每个 Segment 的 count 的最新值，但是拿到之后可能累加前使用的 count 发生了变化，那么统计结果就不准了。

所以最安全的做法，是在统计 size 的时候把所有 Segment 的 put，remove 和 clean 方法全部锁住，但是这种做法显然非常低效。

因为在累加 count 操作过程中，之前累加过的 count 发生变化的几率非常小，所以 ConcurrentHashMap 的做法是先尝试 2 次通过不锁住 Segment 的方式来统计各个 Segment 大小，如果统计的过程中，容器的 count 发生了变化，则再采用加锁的方式来统计所有 Segment 的大小。

那么 ConcurrentHashMap 是如何判断在统计的时候容器是否发生了变化呢？使用 modCount 变量，在 put , remove 和 clean 方法里操作元素前都会将变量 modCount 进行加 1，那么在统计 size 前后比较 modCount 是否发生变化，从而得知容器的大小是否发生变化。

9、总结

1.在使用锁来协调多线程间并发访问的模式下，减小对锁的竞争可以有效提高并发性。

有两种方式可以减小对锁的竞争：

减小请求同一个锁的频率。

减少持有锁的时间。

2.ConcurrentHashMap 的高并发性主要来自于三个方面：

用分离锁实现多个线程间的更深层次的共享访问。

用 HashEntery 对象的不变性来降低执行读操作的线程在遍历链表期间对加锁的需求。

通过对同一个 Volatile 变量的写 / 读访问，协调不同线程间读 / 写操作的内存可见性。

使用分离锁，减小了请求同一个锁的频率。

17、为什么 Hashtable ConcurrentHashmap 不支持 key 或者 value 为 null

ConcurrentHashmap HashMap 和 Hashtable 都是 key-value 存储结构，但他们有一个不同点是 ConcurrentHashmap、Hashtable 不支持 key 或者 value 为 null，而 HashMap 是支持的。为什么会有这个区别？在设计上的目的是什么？

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。

HashMap.class:

// 此处计算 key 的 hash 值时，会判断是否为 null，如果是，则返回 0，即 key 为 null 的键值对

// 的 hash 为 0。因此一个 hashmap 对象只会存储一个 key 为 null 的键值对，因为它们的 hash 值都相同。

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
// 将键值对放入 table 中时，不会校验 value 是否为 null。因此一个 hashmap 对象可以存储
// 多个 value 为 null 的键值对
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;
        }
Hashtable.class:
public synchronized V put(K key, V value) {
        // 确保 value 不为空。这句代码过滤掉了所有 value 为 null 的键值对。因此 Hashtable
不能
        // 存储 value 为 null 的键值对
        if (value == null) {
            throw new NullPointerException();
        }
        // 确保 key 在 table 数组中尚未存在。
        Entry<?,?> tab[] = table;
        int hash = key.hashCode(); //在此处计算 key 的 hash 值，如果此处 key 为 null，则直
```

接抛出空指针异常。

```
        int index = (hash & 0x7FFFFFFF) % tab.length;
        @SuppressWarnings("unchecked")
        Entry<K,V> entry = (Entry<K,V>)tab[index];
        for(; entry != null ; entry = entry.next) {
            if ((entry.hash == hash) && entry.key.equals(key)) {
                V old = entry.value;
                entry.value = value;
                return old;
            }
        }
        addEntry(hash, key, value, index);
        return null;
    }
```

基本功

18、面向对象的特征

面向对象的三个基本特征是：封装、继承、多态。

封装

封装最好理解了。封装是面向对象的特征之一，是对象和类概念的主要特性。

封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。

继承

面向对象编程 (OOP) 语言的一个主要功能就是“继承”。继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。

多态

多态性（polymorphisn）是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。简单的说，就是一句话：允许将子类类型的指针赋值给父类类型的指针。

实现多态，有二种方式，覆盖，重载。

19、int 和 Integer 有什么区别

int 是 java 提供的 8 种原始数据类型之一。Java 为每个原始类型提供了封装类，Integer 是 java 为 int 提供的封装类。

int 的默认值为 0，而 Integer 的默认值为 null，是引用类型，即 Integer 可以区分出未赋值和值为 0 的区别，int 则无法表达出未赋值的情况，

Java 中 int 和 Integer 关系是比较微妙的。关系如下：

1、int 是基本的数据类型；

2、Integer 是 int 的封装类；

3、int 和 Integer 都可以表示某一个数值；

4、int 和 Integer 不能够互用，因为他们两种不同的数据类型；

20、重载和重写的区别

重载 Overload 表示同一个类中可以有多个名称相同的方法，但这些方法的参数列表各不相同（即参数个数或类型不同）。

重写 Override 表示子类中的方法可以与父类中的某个方法的名称和参数完全相同，通过子类创建的实例对象调用这个方法时，将调用子类中的定义方法，这相当于把父类中定义的那个完全相同的方法给覆盖了，这也是面向对象编程的多态性的一种表现。子类覆盖父类的方法时，只能比父类抛出更少的异常，或者是抛出父类抛出的异常的子异常，因为子类可以解决父类的一些问题，不能比父类有更多的问题。子类方法的访问权限只能比父类的更大，不能更小。如果父类的方法是 private 类型，那么，子类则不存在覆盖的限制，相当于子类中增加了一个全新的方法。

21、抽象类和接口有什么区别

抽象类是用来捕捉子类的通用特性的。它不能被实例化，只能被用作子类的超类。抽象类是被用来创建继承层级里的子类的模板。

接口是抽象方法的集合，如果一个类实现类某个接口，那么他就继承了这个接口的抽象方法。这就像契约模式。如果实现了这个接口，那么就必须确保使用这些方法。接口只是一种形式，接口自身不能做任何事情。

参数	抽象类	接口
默认的方法实现	它可以有默认的方法实现	接口完全是抽象的。它根本不存在方法的实现
实现	子类使用 **extends** 关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽象类中所有声明的方法的实现。	子类使用关键字 **implements** 来实现接口。它需要提供接口中所有声明的方法的实现
构造器	抽象类可以有构造器	接口不能有构造器
与正常 Java 类的区别	除了你不能实例化抽象类之外，它和普通 Java 类没有任何区别	接口是完全不同的类型
访问修饰符	抽象方法可以有 **public**、**protected** 和 **default** 这些修饰符	接口方法默认修饰符是 **public**。你不可以使用其它修饰符。
main 方法	抽象方法可以有 main 方法并且我们可以运行它	接口没有 main 方法，因此我们不能运行它。
多继承	抽象方法可以继承一个类和实现多个接口	接口只可以继承一个或多个其它接口
速度	它比接口速度要快	接口是稍微有点慢的，因为它需要时间去寻找在类中实现的方法。
添加新方法	如果你往抽象类中添加新的方法，你可以给它提供默认的实现。因此你不需要改变你现在的代码。	如果你往接口中添加方法，那么你必须改变实现该接口的类。

什么时候使用抽象类和接口

1、如果你拥有一些方法想让他们中的一些默认实现，那么使用抽象类。

2、如果你想实现多重继承，那么你必须使用接口。由于 java 不支多继承，子类不能够继承多个类，但可以实现多个接口

3、如果基本功能在不断改变，那么就需要使用抽象类。如果不断改变基本功能并且使用接口 ，那么就需要改变所有实现了该接口的类。

JDK 8 中的默认方法

向接口中引入了默认方法和静态方法，以此来减少抽象类和接口之间的差异。现在我们可以为接口提供默认实现的方法来，并且不用强制来实现它。

22、说说自定义注解的场景及实现

登陆、权限拦截、日志处理，以及各种 Java 框架，如 Spring，Hibernate，JUnit 提到注解就不能不说反射，Java 自定义注解是通过运行时靠反射获取注解。实际开发中，例如我们要获取某个方法的调用日志，可以通过 AOP（动态代理机制）给方法添加切面，通过反射来获取方法包含的注解，如果包含日志注解，就进行日志记录。反射的实现在 Java 应用层面上讲，是通过对 Class 对象的操作实现的，Class 对象为我们提供了一系列方法对类进行操作。在 Jvm 这个角度来说，Class 文件是一组以 8 位字节为基础单位的二进制流，各个数据项目按严格的顺序紧凑的排列在 Class 文件中，里面包含了类、方法、字段等等相关数据。通过对 Claas 数据流的处理我们即可得到字段、方法等数据。

23、HTTP 请求的 GET 与 POST 方式的区别

GET 方法

请注意，查询字符串（名称/值对）是在 GET 请求的 URL 中发送的：

/test/demo_form.asp?name1=value1&name2=value2

请求可被缓存

请求保留在浏览器历史记录中

请求可被收藏为书签

请求不应在处理敏感数据时使用

请求有长度限制

请求只应当用于取回数据

POST 方法

请注意，查询字符串（名称/值对）是在 POST 请求的 HTTP 消息主体中发送的：

POST /test/demo_form.asp HTTP/1.1

Host: w3schools.com

name1=value1&name2=value2

方法	GET	POST
缓存	能被缓存	不能缓存
编码类型	application/x-www-form-urlencoded	application/x-www-form-urlencoded 或 multipart/form-data。为二进制数据使用多重编码。
对数据长度的限制	是的。当发送数据时，GET 方法向 URL 添加数据；URL 的长度是受限制的（URL 的最大长度是 2048 个字符）	无限制。
对数据类型的限制	只允许 ASCII 字符	没有限制。也允许二进制数据。
安全性	与 POST 相比，GET 的安全性较差，因为所发送的数据是 URL 的一部分。在发送密码或其他敏感信息时绝不要使用 GET	POST 比 GET 更安全，因为参数不会被保存在浏览器历或 web 服务器日志中。
可见性	数据在 URL 中对所有人都是可见的。	数据不会显示在 URL 中。

比较 GET 与 POST

其他 HTTP 请求方法
HEAD 与 GET 相同，但只返回 HTTP 报头，不返回文档主体。
PUT 上传指定的 URI 表示。
DELETE 删除指定资源。
OPTIONS 返回服务器支持的 HTTP 方法
CONNECT 把请求连接转换到透明的 TCP/IP 通道。

24、session 与 cookie 区别

cookie 数据存放在客户的浏览器上，session 数据放在服务器上。

cookie 不是很安全，别人可以分析存放在本地的 COOKIE 并进行 COOKIE 欺骗考虑到安全应当使用 session。

session 会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能考虑到减轻服务器性能方面，应当使用 COOKIE。

单个 cookie 保存的数据不能超过 4K，很多浏览器都限制一个站点最多保存 20 个 cookie。

所以个人建议：

将登陆信息等重要信息存放为 SESSION

其他信息如果需要保留，可以放在 COOKIE 中

25、session 分布式处理

第一种：粘性 session

粘性 Session 是指将用户锁定到某一个服务器上，比如上面说的例子，用户第一次请求时，负载均衡器将用户的请求转发到了 A 服务器上，如果负载均衡器设置了粘性 Session 的话，那么用户以后的每次请求都会转发到 A 服务器上，相当于把用户和 A 服务器粘到了一块，这就是粘性 Session 机制

第二种：服务器 session 复制

原理：任何一个服务器上的 session 发生改变（增删改），该节点会把这个 session 的所有内容序列化，然后广播给所有其它节点，不管其他服务器需不需要 session，以此来保证 Session 同步。

第三种：session 共享机制

使用分布式缓存方案比如 memcached、Redis，但是要求 Memcached 或 Redis 必须是集群。

原理：不同的 tomcat 指定访问不同的主 memcached。多个 Memcached 之间信息是同步的，能主从备份和高可用。用户访问时首先在 tomcat 中创建 session，然后将 session 复制一份放到它对应的 memcahed 上

第四种：session 持久化到数据库

原理：就不用多说了吧，拿出一个数据库，专门用来存储 session 信息。保证 session 的持久化。优点：服务器出现问题，session 不会丢失 缺点：如果网站的访问量很大，把 session 存储到数据库中，会对数据库造成很大压力，还需要增加额外的开销维护数据库。

第五种 terracotta 实现 session 复制

原理：就不用多说了吧，拿出一个数据库，专门用来存储 session 信息。保证 session 的持久化。优点：服务器出现问题，session 不会丢失 缺点：如果网站的访问量很大，把 session 存储到数据库中，会对数据库造成很大压力，还需要增加额外的开销维护数据库

26、JDBC 流程

（1）向 DriverManager 类注册驱动数据库驱动程序

```
Class.forName( "com.somejdbcvendor.TheirJdbcDriver" );
```

（2）调用 DriverManager.getConnection 方法， 通过 JDBC URL，用户名，密码取得数据库连接的 Connection 对象。

```
Connection conn = DriverManager.getConnection(
        "jdbc:somejdbcvendor:other data needed by some jdbc vendor", //URL
         "myLogin", // 用户名
        "myPassword" ); // 密码
```

（3）获取 Connection 后， 便可以通过 createStatement 创建 Statement 用以执行 SQL 语句。

```
Statement stmt = conn.createStatement();
 stmt.executeUpdate( "INSERT INTO MyTable( name ) VALUES ( 'my name' ) " );
```

要执行 SQL 语句，必须获得 java.sql.Statement 实例，Statement 实例分为以下 3 种类型：

1、执行静态 SQL 语句。通常通过 Statement 实例实现。

2、执行动态 SQL 语句。通常通过 PreparedStatement 实例实现。

3、执行数据库存储过程。通常通过 CallableStatement 实例实现。
具体的实现方式：
Statement stmt = con.createStatement() ; PreparedStatement pstmt = con.prepareStatement(sql) ; CallableStatement cstmt = con.prepareCall("{CALL demoSp(? , ?)}") ;
（4）有时候会得到查询结果，比如 select，得到查询结果，查询（SELECT）的结果存放于结果集（ResultSet）中。
ResultSet rs = stmt.executeQuery("SELECT * FROM MyTable");
Statement 接口提供了三种执行 SQL 语句的方法：executeQuery 、executeUpdate 和 execute
1、ResultSet executeQuery(String sqlString)：执行查询数据库的 SQL 语句 ，返回一个结果集（ResultSet）对象。
2、int executeUpdate(String sqlString)：用于执行 INSERT、UPDATE 或 DELETE 语句以及 SQL DDL 语句，如：CREATE TABLE 和 DROP TABLE 等
3、execute(sqlString):用于执行返回多个结果集、多个更新计数或二者组合的 语句。 具体实现的代码：
ResultSet rs = stmt.executeQuery(“ SELECT * FROM … ”) ; int rows = stmt.executeUpdate(“INSERT INTO …”) ; boolean flag = stmt.execute(String sql) ;
获取结果为两种情况：
1、执行更新返回的是本次操作影响到的记录数。
2、执行查询返回的结果是一个 ResultSet 对象。
• ResultSet 包含符合 SQL 语句中条件的所有行，并且它通过一套 get 方法提供了对这些 行中数据的访问。
• 使用结果集（ResultSet）对象的访问方法获取数据：
```
while(rs.next()){
String name = rs.getString(“name”) ;
String pass = rs.getString(1) ; // 此方法比较高效
}
```
（列是从左到右编号的，并且从列 1 开始）
（5）关闭数据库语句，关闭数据库连接。
```
rs.close();
stmt.close();
```

27、equals 与 == 的区别

==与 equals 的主要区别是：==常用于比较原生类型，而 equals()方法用于检查对象的相等性。另一个不同的点是：如果==和 equals()用于比较对象，当两个引用地址相同，==返回 true。而 equals()可以返回 true 或者 false 主要取决于重写实现。最常见的一个例子，字符串的比较，不同情况==和 equals()返回不同的结果。

使用==比较原生类型如：boolean、int、char 等等，使用 equals()比较对象。

==返回 true 如果两个引用指向相同的对象，equals()的返回结果依赖于具体业务实现

字符串的对比使用 equals()代替==操作符

使用==比较原生类型如：boolean、int、char 等等，使用 equals()比较对象。

==返回 true 如果两个引用指向相同的对象，equals()的返回结果依赖于具体业务实现

字符串的对比使用 equals()代替==操作符

其主要的不同是一个是操作符一个是方法，==用于对比原生类型而 equals()方法比较对象的相等性。

28、MVC 设计思想

1：什么是 MVC

MVC(Model View Controller)是一种软件设计的框架模式，它采用模型(Model)-视图(View)-控制器(controller)的方法把业务逻辑、数据与界面显示分离。把众多的业务逻辑聚集到一个部件里面，当然这种比较官方的解释是不能让我们足够清晰的理解什么是 MVC 的。用通俗的话来讲，MVC 的理念就是把数据处理、数据展示(界面)和程序/用户的交互三者分离开的一种编程模式。

注意！MVC 不是设计模式！

MVC 框架模式是一种复合模式，MVC 的三个核心部件分别是

1：Model(模型)：所有的用户数据、状态以及程序逻辑，独立于视图和控制器

2：View(视图)：呈现模型，类似于 Web 程序中的界面，视图会从模型中拿到需要展现的状态以及数据，对于相同的数据可以有多种不同的显示形式(视图)

3：Controller(控制器)：负责获取用户的输入信息，进行解析并反馈给模型，通常情况下一个视图具有一个控制器

1.2：为什么要使用 MVC

程序通过将 M(Model)和 V(View)的代码分离，实现了前后端代码的分离，会带来几个好处

1：可以使同一个程序使用不同的表现形式，如果控制器反馈给模型的数据发生了变化，那么模型将及时通知有关的视图，视图会对应的刷新自己所展现的内容

2：因为模型是独立于视图的，所以模型可复用，模型可以独立的移植到别的地方继续使用

3：前后端的代码分离，使项目开发的分工更加明确，程序的测试更加简便，提高开发效率。

其实控制器的功能类似于一个中转站，会决定调用那个模型去处理用户请求以及调用哪个视图去呈现给用户

1.3：JavaWeb 中 MVC 模式的应用

在 JavaWeb 程序中，MVC 框架模式是经常用到的，举一个 Web 程序的结构可以更好的理解 MVC 的理念

V：View 视图，Web 程序中指用户可以看到的并可以与之进行数据交互的界面，比如一个 Html 网页界面，或者某些客户端的界面，在前面讲过，MVC 可以为程序处理很多不同的视图，用户在视图中进行输出数据以及一系列操作，注意：视图中不会发生数据的处理操作。

M：Model 模型：进行所有数据的处理工作，模型返回的数据是中立的，和数据格式无关，一个模型可以为多个视图来提供数据，所以模型的代码重复性比较低

C：Controller 控制器：负责接受用户的输入，并且调用模型和视图去完成用户的需求，控制器不会输出也不会做出任何处理，只会接受请求并调用模型构件去处理用户的请求，然后在确定用哪个视图去显示返回的数据

1.4：Web 程序中 MVC 模式的优点

耦合性低：视图(页面)和业务层(数据处理)分离，一个应用的业务流程或者业务规则的改变只需要改动 MVC 中的模型即可，不会影响到控制器与视图

部署快，成本低：MVC 使开发和维护用户接口的技术含量降低。使用 MVC 模式使开发时间得到相当大的缩减，它使程序员（Java 开发人员）集中精力于业务逻辑，界面程序员（HTML 和 JSP 开发人员）集中精力于表现形式上

可维护性高：分离视图层和业务逻辑层也使得 WEB 应用更易于维护和修改

1.5：Web 程序中 MVC 模式的缺点

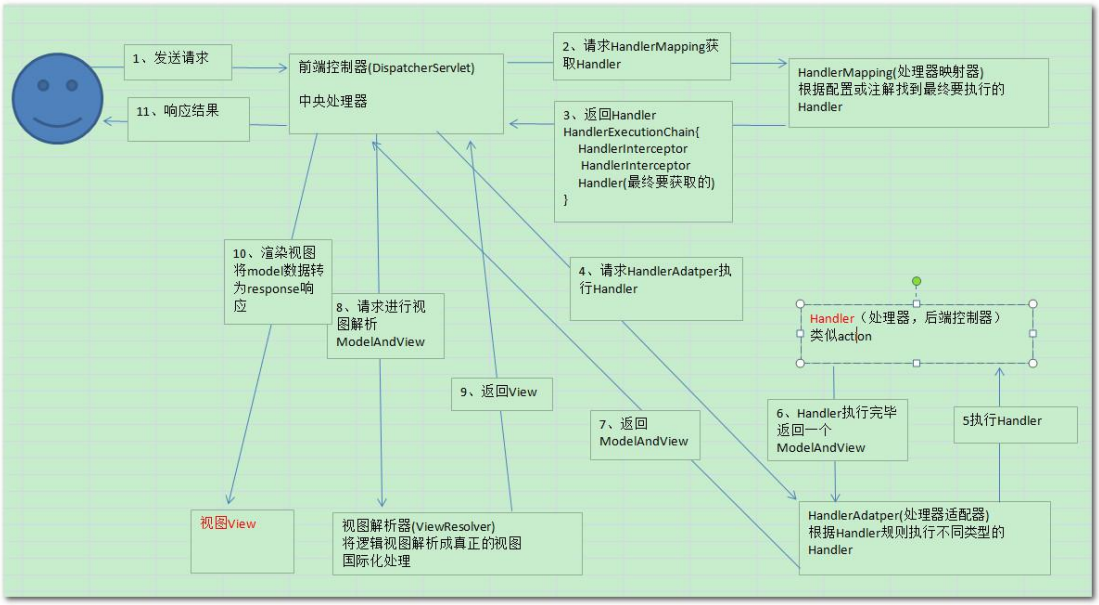
调试困难：因为模型和视图要严格的分离，这样也给调试应用程序带来了一定的困难，每个构件在使用之前都需要经过彻底的测试

不适合小型，中等规模的应用程序：在一个中小型的应用程序中，强制性的使用 MVC 进行开发，往往会花费大量时间，并且不能体现 MVC 的优势，同时会使开发变得繁琐

增加系统结构和实现的复杂性：对于简单的界面，严格遵循 MVC，使模型、视图与控制器分离，会增加结构的复杂性，并可能产生过多的更新操作，降低运行效率

视图与控制器间的过于紧密的连接并且降低了视图对模型数据的访问：视图与控制器是相互分离，但却是联系紧密的部件，视图没有控制器的存在，其应用是很有限的，反之亦然，这样就妨碍了他们的独立重用。依据模型操作接口的不同，视图可能需要多次调用才能获得足够的显示数据。对未变化数据的不必要的频繁访问，也将损害操作性能。

29、springmvc 工作原理图



SpringMVC 流程

1、 用户发送请求至前端控制器 DispatcherServlet。

2、 DispatcherServlet 收到请求调用 HandlerMapping 处理器映射器。

3、 处理器映射器找到具体的处理器(可以根据 xml 配置、注解进行查找)，生成处理器对象及处理器拦截器(如果有则生成)一并返回给 DispatcherServlet。

4、 DispatcherServlet 调用 HandlerAdapter 处理器适配器。

5、 HandlerAdapter 经过适配调用具体的处理器(Controller，也叫后端控制器)。

6、 Controller 执行完成返回 ModelAndView。

7、 HandlerAdapter 将 controller 执行结果 ModelAndView 返回给 DispatcherServlet。

8、 DispatcherServlet 将 ModelAndView 传给 ViewReslover 视图解析器。

9、 ViewReslover 解析后返回具体 View。

10、DispatcherServlet 根据 View 进行渲染视图（即将模型数据填充至视图中）。

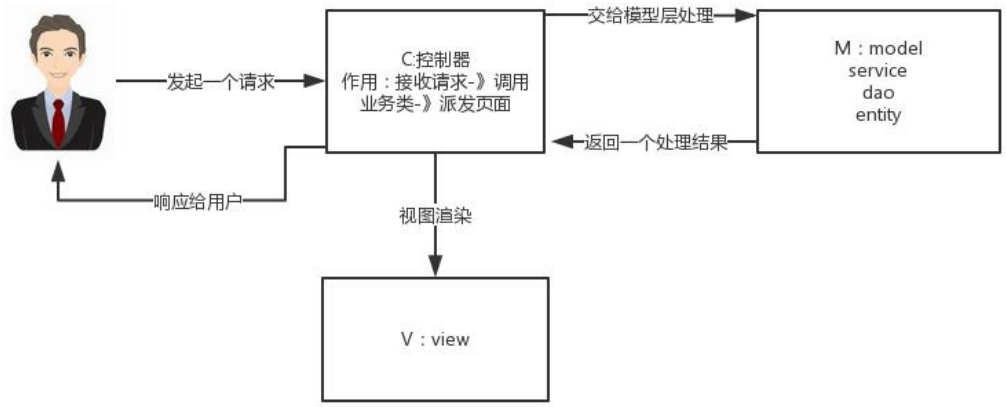
11、 DispatcherServlet 响应用户。

组件说明：

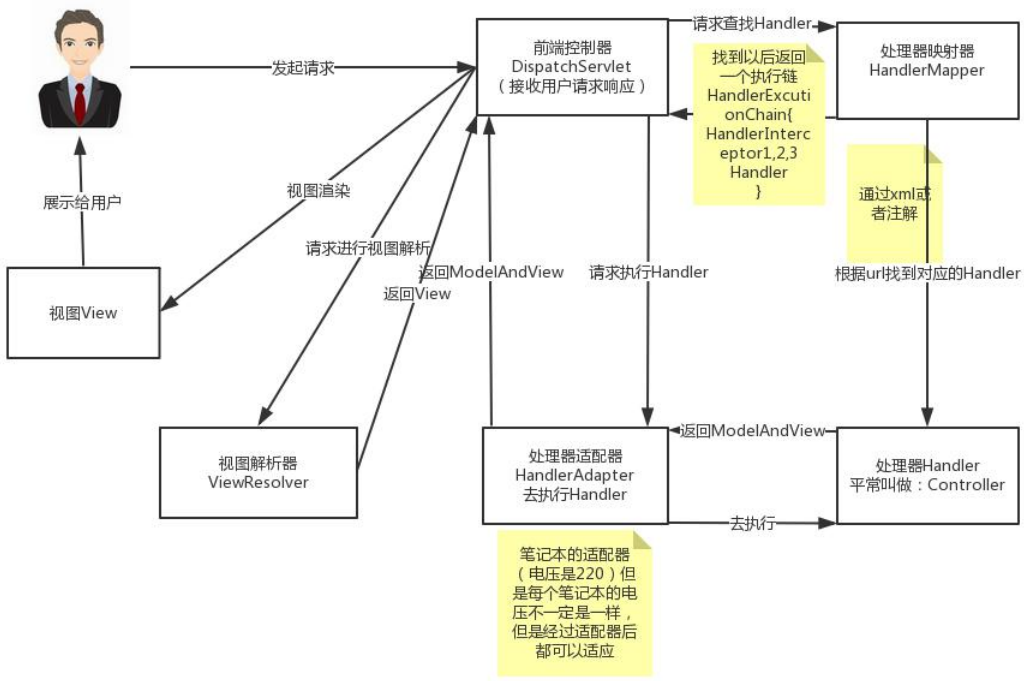
以下组件通常使用框架提供实现：

DispatcherServlet：作为前端控制器，整个流程控制的中心，控制其它组件执行，统一调度，降低组件之间的耦合性，提高每个组件的扩展性。
HandlerMapping：通过扩展处理器映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。
HandlAdapter：通过扩展处理器适配器，支持更多类型的处理器。
ViewResolver：通过扩展视图解析器，支持更多类型的视图解析，例如：jsp、freemarker、pdf、excel 等。
组件：
1、前端控制器 DispatcherServlet（不需要工程师开发）,由框架提供
作用：接收请求，响应结果，相当于转发器，中央处理器。有了 dispatcherServlet 减少了其它组件之间的耦合度。
用户请求到达前端控制器，它就相当于 mvc 模式中的 c，dispatcherServlet 是整个流程控制的中心，由它调用其它组件处理用户的请求，dispatcherServlet 的存在降低了组件之间的耦合性。
2、处理器映射器 HandlerMapping(不需要工程师开发),由框架提供
作用：根据请求的 url 查找 Handler
HandlerMapping 负责根据用户请求找到 Handler 即处理器，springmvc 提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。
3、处理器适配器 HandlerAdapter
作用：按照特定规则（HandlerAdapter 要求的规则）去执行 Handler
通过 HandlerAdapter 对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。
4、处理器 Handler(需要工程师开发)
注意：编写 Handler 时按照 HandlerAdapter 的要求去做，这样适配器才可以去正确执行 Handler
Handler 是继 DispatcherServlet 前端控制器的后端控制器，在 DispatcherServlet 的控制下 Handler 对具体的用户请求进行处理。
由于 Handler 涉及到具体的用户业务请求，所以一般情况需要工程师根据业务需求开发 Handler。
5、视图解析器 View resolver(不需要工程师开发),由框架提供
作用：进行视图解析，根据逻辑视图名解析成真正的视图（view）
View Resolver 负责将处理结果生成 View 视图，View Resolver 首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成 View 视图对象，最后对 View 进行渲染将处理结果通过页面展示给用户。 springmvc 框架提供了很多的 View 视图类型，包括：jstlView、freemarkerView、pdfView 等。
一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由工程师根据业务需求开发具体的页面。
6、视图 View(需要工程师开发 jsp...)
View 是一个接口，实现类支持不同的 View 类型（jsp、freemarker、pdf...）
核心架构的具体流程步骤如下：
1、首先用户发送请求——>DispatcherServlet，前端控制器收到请求后自己不进行处理，而是委托给其他的解析器进行处理，作为统一访问点，进行全局的流程控制；
2 、 DispatcherServlet —— >HandlerMapping ， HandlerMapping 将会把请求映射为 HandlerExecutionChain 对象（包含一个 Handler 处理器（页面控制器）对象、多个

HandlerInterceptor 拦截器）对象，通过这种策略模式，很容易添加新的映射策略；
3、DispatcherServlet——>HandlerAdapter，HandlerAdapter 将会把处理器包装为适配器，从而支持多种类型的处理器，即适配器设计模式的应用，从而很容易支持很多类型的处理器；
4、HandlerAdapter——>处理器功能处理方法的调用，HandlerAdapter 将会根据适配的结果调用真正的处理器的功能处理方法，完成功能处理；并返回一个 ModelAndView 对象（包含模型数据、逻辑视图名）；
5、ModelAndView 的逻辑视图名——> ViewResolver， ViewResolver 将把逻辑视图名解析为具体的 View，通过这种策略模式，很容易更换其他视图技术；
6、View——>渲染，View 会根据传进来的 Model 模型数据进行渲染，此处的 Model 实际是一个 Map 数据结构，因此很容易支持其他视图技术；
7、返回控制权给 DispatcherServlet，由 DispatcherServlet 返回响应给用户，到此一个流程结束。
下边两个组件通常情况下需要开发：
Handler：处理器，即后端控制器用 controller 表示。
View：视图，即展示给用户的界面，视图中通常需要标签语言展示模型数据。
在将 SpringMVC 之前我们先来看一下什么是 MVC 模式
MVC：MVC 是一种设计模式
MVC 的原理图：



分析：
M-Model 模型（完成业务逻辑：有 javaBean 构成，service+dao+entity）
V-View 视图（做界面的展示 jsp，html……）
C-Controller 控制器（接收请求—>调用模型—>根据结果派发页面）
springMVC 是什么：
springMVC 是一个 MVC 的开源框架，springMVC=struts2+spring，springMVC 就相当于是 Struts2 加上 sring 的整合，但是这里有一个疑惑就是，springMVC 和 spring 是什么样的关系呢？这个在百度百科上有一个很好的解释：意思是说，springMVC 是 spring 的一个后续产品，其实就是 spring 在原有基础上，又提供了 web 应用的 MVC 模块，可以简单的把 springMVC 理解为是 spring 的一个模块（类似 AOP，IOC 这样的模块），网络上经常会说 springMVC 和 spring 无缝集成，其实 springMVC 就是 spring 的一个子模块，所以根本不需要同 spring 进行整合。
SpringMVC 的原理图：



看到这个图大家可能会有很多的疑惑，现在我们来看一下这个图的步骤：（可以对比 MVC 的原理图进行理解）

第一步:用户发起请求到前端控制器（DispatcherServlet）

第二步：前端控制器请求处理器映射器（HandlerMappering）去查找处理器（Handle）：通过 xml 配置或者注解进行查找

第三步：找到以后处理器映射器（HandlerMappering）像前端控制器返回执行链（HandlerExecutionChain）

第四步：前端控制器（DispatcherServlet）调用处理器适配器（HandlerAdapter）去执行处理器（Handler）

第五步：处理器适配器去执行 Handler

第六步：Handler 执行完给处理器适配器返回 ModelAndView

第七步：处理器适配器向前端控制器返回 ModelAndView

第八步：前端控制器请求视图解析器（ViewResolver）去进行视图解析

第九步：视图解析器像前端控制器返回 View

第十步：前端控制器对视图进行渲染

第十一步：前端控制器向用户响应结果

看到这些步骤我相信大家很感觉非常的乱，这是正常的，但是这里主要是要大家理解 springMVC 中的几个组件：

前端控制器（DispatcherServlet）：接收请求，响应结果，相当于电脑的 CPU。

处理器映射器（HandlerMapping）：根据 URL 去查找处理器

处理器（Handler）：（需要程序员去写代码处理逻辑的）

处理器适配器（HandlerAdapter）：会把处理器包装成适配器，这样就可以支持多种类型的处理器，类比笔记本的适配器（适配器模式的应用）

视图解析器（ViewResovler）：进行视图解析，多返回的字符串，进行处理，可以解析成对应的页面。

线程

30、创建线程的方式及实现

线程被称为轻量级进程，是程序执行的最小单位，它是指在程序执行过程中，能够执行代码的一个执行单位。每个程序程序都至少有一个线程，也即是程序本身。

线程状态：

（1）新建（New）：创建后尚未启动的线程处于这种状态

（2）运行（Runable）：Runable 包括了操作系统线程状态的 Running 和 Ready，也就是处于此状态的线程有可能正在执行，也有可能正在等待着 CPU 为它分配执行时间。

（3）等待（Wating）：处于这种状态的线程不会被分配 CPU 执行时间。等待状态又分为无限期等待和有限期等待，处于无限期等待的线程需要被其他线程显示地唤醒，没有设置 Timeout 参数的 Object.wait()、没有设置 Timeout 参数的 Thread.join()方法都会使线程进入无限期等待状态；有限期等待状态无须等待被其他线程显示地唤醒，在一定时间之后它们会由系统自动唤醒，Thread.sleep()、设置了 Timeout 参数的 Object.wait()、设置了 Timeout 参数的 Thread.join()方法都会使线程进入有限期等待状态。

（4）阻塞（Blocked）：线程被阻塞了，“阻塞状态”与”等待状态“的区别是：”阻塞状态“在等待着获取到一个排他锁，这个时间将在另外一个线程放弃这个锁的时候发生；而”等待状态“则是在等待一段时间或者唤醒动作的发生。在程序等待进入同步区域的时候，线程将进入这种状态。

（5）结束（Terminated）：已终止线程的线程状态，线程已经结束执行。

线程同步方法

线程有 4 中同步方法，分别为 wait()、sleep()、notify()和 notifyAll()。

wait()：使线程处于一种等待状态，释放所持有的对象锁。

sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用它时要捕获 InterruptedException 异常，不释放对象锁。

notify()：唤醒一个正在等待状态的线程。注意调用此方法时，并不能确切知道唤醒的是哪一个等待状态的线程，是由 JVM 来决定唤醒哪个线程，不是由线程优先级决定的。

notifyAll()：唤醒所有等待状态的线程，注意并不是给所有唤醒线程一个对象锁，而是让它们竞争。

创建线程呢个的 4 种方式：

（1）继承 Thread 类

```
//继承 Thread 类来创建线程
public class ThreadTest {
    public static void main(String[] args) {
        //设置线程名字
        Thread.currentThread().setName("main thread");
        MyThread myThread = new MyThread();
        myThread.setName("子线程:");
        //开启线程
        myThread.start();
        for(int i = 0;i<5;i++){
            System.out.println(Thread.currentThread().getName() + i);
        }
```

```
    }
}

class MyThread extends Thread{
    //重写 run()方法
    public void run(){
        for(int i = 0;i < 10; i++){
            System.out.println(Thread.currentThread().getName() + i);
        }
    }
}
```

（2）实现 Runnable 接口

```
//实现 Runnable 接口
public class RunnableTest {
    public static void main(String[] args) {
        //设置线程名字
        Thread.currentThread().setName("main thread:");
        Thread thread = new Thread(new MyRunnable());
        thread.setName("子线程:");
        //开启线程
        thread.start();
        for(int i = 0; i <5;i++){
            System.out.println(Thread.currentThread().getName() + i);
        }
    }
}

class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread().getName() + i);
        }
    }
}
```

（3）实现 Callable 接口

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;
//实现 Callable 接口
public class CallableTest {

    public static void main(String[] args) {
        //执行 Callable 方式，需要 FutureTask 实现实现，用于接收运算结果
```

```
        FutureTask<Integer> futureTask = new FutureTask<Integer>(new MyCallable());
        new Thread(futureTask).start();
        //接收线程运算后的结果
        try {
            Integer sum = futureTask.get();
            System.out.println(sum);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}

class MyCallable implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        int sum = 0;
        for (int i = 0; i < 100; i++) {
            sum += i;
        }
        return sum;
    }
}
```

（4）线程池

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
//线程池实现
public class ThreadPoolExecutorTest {
    public static void main(String[] args) {
        //创建线程池
        ExecutorService executorService = Executors.newFixedThreadPool(10);
        ThreadPool threadPool = new ThreadPool();
        for(int i =0;i<5;i++){
            //为线程池分配任务
            executorService.submit(threadPool);
        }
        //关闭线程池
        executorService.shutdown();
    }
}

class ThreadPool implements Runnable {
    @Override
```

```
    public void run() {
        for(int i = 0 ;i<10;i++){
            System.out.println(Thread.currentThread().getName() + ":" + i);
        }
    }
}
```

Executors 提供了一系列工厂方法用于创先线程池，返回的线程池都实现了 ExecutorService 接口。

public static ExecutorService newFixedThreadPool(int nThreads)

创建固定数目线程的线程池。

public static ExecutorService newCachedThreadPool()

创建一个可缓存的线程池，调用 execute 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线　程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。

public static ExecutorService newSingleThreadExecutor()

创建一个单线程化的 Executor。

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)

创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代 Timer 类。

31、sleep() 、join（）、yield（）有什么区别

sleep()

sleep()方法需要指定等待的时间，它可以让当前正在执行的线程在指定的时间内暂停执行，进入阻塞状态，该方法既可以让其他同优先级或者高优先级的线程得到执行的机会，也可以让低优先级的线程得到执行机会。但是 sleep()方法不会释放“锁标志”，也就是说如果有 synchronized 同步块，其他线程仍然不能访问共享数据。

wait()

wait()方法需要和 notify()及 notifyAll()两个方法一起介绍，这三个方法用于协调多个线程对共享数据的存取，所以必须在 synchronized 语句块内使用，也就是说，调用 wait()，notify()和 notifyAll()的任务在调用这些方法前必须拥有对象的锁。注意，它们都是 Object 类的方法，而不是 Thread 类的方法。

wait()方法与 sleep()方法的不同之处在于，wait()方法会释放对象的“锁标志”。当调用某一对象的 wait()方法后，会使当前线程暂停执行，并将当前线程放入对象等待池中，直到调用了 notify()方法后，将从对象等待池中移出任意一个线程并放入锁标志等待池中，只有锁标志等待池中的线程可以获取锁标志，它们随时准备争夺锁的拥有权。当调用了某个对象的 notifyAll()方法，会将对象等待池中的所有线程都移动到该对象的锁标志等待池。

除了使用 notify()和 notifyAll()方法，还可以使用带毫秒参数的 wait(long timeout)方法，效果是在延迟 timeout 毫秒后，被暂停的线程将被恢复到锁标志等待池。

此外，wait()，notify()及 notifyAll()只能在 synchronized 语句中使用，但是如果使用的是 ReenTrantLock 实现同步，该如何达到这三个方法的效果呢？解决方法是使用 ReenTrantLock.newCondition()获取一个 Condition 类对象，然后 Condition 的 await()，signal()以及 signalAll()分别对应上面的三个方法。

yield()

yield()方法和 sleep()方法类似，也不会释放“锁标志”，区别在于，它没有参数，即 yield()方法只是使当前线程重新回到可执行状态，所以执行 yield()的线程有可能在进入到可执行状

态后马上又被执行，另外 yield()方法只能使同优先级或者高优先级的线程得到执行机会，这也和 sleep()方法不同。

join()

join()方法会使当前线程等待调用 join()方法的线程结束后才能继续执行，例如：

```
package concurrent;
public class TestJoin {
    public static void main(String[] args) {
        Thread thread = new Thread(new JoinDemo());
        thread.start();

        for (int i = 0; i < 20; i++) {
            System.out.println("主线程第" + i + "次执行！");
            if (i >= 2)
                try {
                    // t1 线程合并到主线程中，主线程停止执行过程，转而执行 t1 线程，直到 t1 执行完毕后继续。
                    thread.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
        }
    }
}
class JoinDemo implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("线程 1 第" + i + "次执行！");
        }
    }
}
```

32、说说 CountDownLatch 原理

CountDownLatch CountDownLatch 用过很多次了，突然有点好奇，它是如何实现阻塞线程的，猜想是否跟 LockSupport 有关。今天浏览了一下它的源码，发现其实现是十分简单的，只是简单的继承了 AbstractQueuedSynchronizer，便实现了其功能。

实现原理：让需要的暂时阻塞的线程，进入一个死循环里面，得到某个条件后再退出循环，以此实现阻塞当前线程的效果。

下面从构造方法开始，一步步解释实现的原理：

一、构造方法

下面是实现的源码，非常简短，主要是创建了一个 Sync 对象。

```
public CountDownLatch(int count) {
        if (count < 0) throw new IllegalArgumentException("count < 0");
        this.sync = new Sync(count);
```

```
    }
```

二、Sync 对象

```
private static final class Sync extends AbstractQueuedSynchronizer {
        private static final long serialVersionUID = 4982264981922014374L;
        Sync(int count) {
            setState(count);
        }
        int getCount() {
            return getState();
        }
        protected int tryAcquireShared(int acquires) {
            return (getState() == 0) ? 1 : -1;
        }
        protected boolean tryReleaseShared(int releases) {
            // Decrement count; signal when transition to zero
            for (;;) {
                int c = getState();
                if (c == 0)
                    return false;
                int nextc = c-1;
                if (compareAndSetState(c, nextc))
                    return nextc == 0;
            }
        }
    }
```

假设我们是这样创建的：new CountDownLatch(5)。其实也就相当于 new Sync(5)，相当于 setState(5)。setState 我们可以暂时理解为设置一个计数器，当前计数器初始值为 5。

tryAcquireShared 方法其实就是判断一下当前计数器的值，是否为 0 了，如果为 0 的话返回 1（返回 1 的时候，就表明当前线程可以继续往下走了，不再停留在调用 countDownLatch.await(）这个方法的地方）。

tryReleaseShared 方法就是利用 CAS 的方式，对计数器进行减一的操作，而我们实际上每次调用 countDownLatch.countDown()方法的时候，最终都会调到这个方法，对计数器进行减一操作，一直减到 0 为止。

稍微跑偏了一点，我们看看调用 countDownLatch.await()的时候，做了些什么。

三、countDownLatch.await()

```
public void await() throws InterruptedException {
        sync.acquireSharedInterruptibly(1);
    }
```

代码很简单，就一句话（注意 acquireSharedInterruptibly（）方法是抽象类：AbstractQueuedSynchronizer 的一个方法，我们上面提到的 Sync 继承了它），我们跟踪源码，继续往下看：

四、acquireSharedInterruptibly(int arg)

```
public final void acquireSharedInterruptibly(int arg)
```

```
            throws InterruptedException {
        if (Thread.interrupted())
            throw new InterruptedException();
        if (tryAcquireShared(arg) < 0)
            doAcquireSharedInterruptibly(arg);
    }
```

源码也是非常简单的，首先判断了一下，当前线程是否有被中断，如果没有的话，就调用 tryAcquireShared(int acquires)方法，判断一下当前线程是否还需要“阻塞”。其实这里调用的 tryAcquireShared 方法，就是我们上面提到的 java.util.concurrent.CountDownLatch.Sync.tryAcquireShared(int)这个方法。

当然，在一开始我们没有调用过 countDownLatch.countDown() 方法时，这里 tryAcquireShared 方法肯定是会返回 1 的，因为会进入到 doAcquireSharedInterruptibly 方法。

五、doAcquireSharedInterruptibly(int arg)

```
private void doAcquireSharedInterruptibly(int arg)
    throws InterruptedException {
    final Node node = addWaiter(Node.SHARED);
    boolean failed = true;
    try {
        for (;;) {
            final Node p = node.predecessor();
            if (p == head) {
                int r = tryAcquireShared(arg);
                if (r >= 0) {
                    setHeadAndPropagate(node, r);
                    p.next = null; // help GC
                    failed = false;
                    return;
                }
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                throw new InterruptedException();
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```


这个时候，我们应该对于 countDownLatch.await()方法是怎么“阻塞”当前线程的，已经非常明白了。其实说白了，就是当你调用了 countDownLatch.await()方法后，你当前线程就会进入了一个死循环当中，在这个死循环里面，会不断的进行判断，通过调用 tryAcquireShared 方法，不断判断我们上面说的那个计数器，看看它的值是否为 0 了（为 0 的时候，其实就是我们调用了足够多次数的 countDownLatch.countDown（）方法的时候），如果是为 0 的话，tryAcquireShared 就会返回 1，代码也会进入到图中的红框部分，然后跳出了循环，也就不再“阻塞”当前线程了。需要注意的是，说是在不停的循环，其实也并非在不停的执行 for 循环里面的内容，因为在后面调用 parkAndCheckInterrupt（）方法时，在这个方法里面是会调用 LockSupport.park(this);，来禁用当前线程的。

五、关于 AbstractQueuedSynchronizer

看到这里，如果各位对 AbstractQueuedSynchronizer 没有了解过的话，可能代码还是看得有点迷糊，这是一篇我觉得解释得非常好的文章，大家可以看一下：http://ifeve.com/introduce-abstractqueuedsynchronizer/

33、说说 CyclicBarrier 原理

CyclicBarrier 的字面意思是可循环（Cyclic）使用的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。线程进入屏障通过 CyclicBarrier 的 await() 方法。

CyclicBarrier 默认的构造方法是 CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，每个线程调用 await 方法告诉 CyclicBarrier 我已经到达了屏障，然后当前线程被阻塞。

CyclicBarrier 还提供一个更高级的构造函数 CyclicBarrier(int parties, Runnable barrierAction)，用于在线程到达屏障时，优先执行 barrierAction 这个 Runnable 对象，方便处理更复杂的业务场景。

构造函数

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

实现原理：在 CyclicBarrier 的内部定义了一个 Lock 对象，每当一个线程调用 CyclicBarrier 的 await 方法时，将剩余拦截的线程数减 1，然后判断剩余拦截数是否为 0，如果不是，进入 Lock 对象的条件队列等待。如果是，执行 barrierAction 对象的 Runnable 方法，然后将锁的条件队列中的所有线程放入锁等待队列中，这些线程会依次的获取锁、释放锁，接着先从 await 方法返回，再从 CyclicBarrier 的 await 方法中返回。

实现原理：在 CyclicBarrier 的内部定义了一个 Lock 对象，每当一个线程调用 CyclicBarrier 的 await 方法时，将剩余拦截的线程数减 1，然后判断剩余拦截数是否为 0，如果不是，进入 Lock 对象的条件队列等待。如果是，执行 barrierAction 对象的 Runnable 方法，然后将锁的条件队列中的所有线程放入锁等待队列中，这些线程会依次的获取锁、释放锁，接着先从 await 方法返回，再从 CyclicBarrier 的 await 方法中返回。

await 源码：

```
public int await() throws InterruptedException, BrokenBarrierException {
    try {
        return dowait(false, 0L);
    } catch (TimeoutException toe) {
        throw new Error(toe); // cannot happen
    }
}
```

dowait 源码：

```
private int dowait(boolean timed, long nanos)
    throws InterruptedException, BrokenBarrierException,
           TimeoutException {
    final ReentrantLock lock = this.lock;
```

```
lock.lock();
try {
    final Generation g = generation;
    if (g.broken)
        throw new BrokenBarrierException();
    if (Thread.interrupted()) {
        breakBarrier();
        throw new InterruptedException();
    }
    int index = --count;
    if (index == 0) {    // tripped
        boolean ranAction = false;
        try {
            final Runnable command = barrierCommand;
            if (command != null)
                command.run();
            ranAction = true;
            nextGeneration();
            return 0;
        } finally {
            if (!ranAction)
                breakBarrier();
        }
    }
    // loop until tripped, broken, interrupted, or timed out
    for (;;) {
        try {
            if (!timed)
                trip.await();
            else if (nanos > 0L)
                nanos = trip.awaitNanos(nanos);
        } catch (InterruptedException ie) {
            if (g == generation && ! g.broken) {
                breakBarrier();
                throw ie;
            } else {
                // We're about to finish waiting even if we had not
                // been interrupted, so this interrupt is deemed to
                // "belong" to subsequent execution.
                Thread.currentThread().interrupt();
            }
        }
        if (g.broken)
            throw new BrokenBarrierException();
```

```
                if (g != generation)
                    return index;

                if (timed && nanos <= 0L) {
                    breakBarrier();
                    throw new TimeoutException();
                }
            }
        } finally {
            lock.unlock();
        }
    }
```

当最后一个线程到达屏障点，也就是执行 dowait 方法时，会在 return 0 返回之前调用 finally 块中的 breakBarrier 方法。

breakBarrier 源代码：

```
private void breakBarrier() {
    generation.broken = true;
    count = parties;
    trip.signalAll();
}
```

CyclicBarrier 主要用于一组线程之间的相互等待，而 CountDownLatch 一般用于一组线程等待另一组些线程。实际上可以通过 CountDownLatch 的 countDown()和 await()来实现 CyclicBarrier 的功能。即 CountDownLatch 中的 countDown()+await() = CyclicBarrier 中的 await()。注意：在一个线程中先调用 countDown()，然后调用 await()。

其它方法：CycliBarrier 对象可以重复使用，重用之前应当调用 CyclicBarrier 对象的 reset 方法。

reset 源码：

```
public void reset() {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        breakBarrier();     // break the current generation
        nextGeneration(); // start a new generation
    } finally {
        lock.unlock();
    }
}
import java.util.Random;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class CyclicBarrierDemo {
```

```
    private CyclicBarrier cb = new CyclicBarrier(4);
    private Random rnd = new Random();
    class TaskDemo implements Runnable{
        private String id;
        TaskDemo(String id){
            this.id = id;
        }
        @Override
        public void run(){
            try {
                Thread.sleep(rnd.nextInt(1000));
                System.out.println("Thread " + id + " will wait");
                cb.await();
                System.out.println("-------Thread " + id + " is over");
            } catch (InterruptedException e) {
            } catch (BrokenBarrierException e) {
            }
        }
    }
    public static void main(String[] args){
        CyclicBarrierDemo cbd = new CyclicBarrierDemo();
        ExecutorService es = Executors.newCachedThreadPool();
        es.submit(cbd.new TaskDemo("a"));
        es.submit(cbd.new TaskDemo("b"));
        es.submit(cbd.new TaskDemo("c"));
        es.submit(cbd.new TaskDemo("d"));
        es.shutdown();
    }
}
```

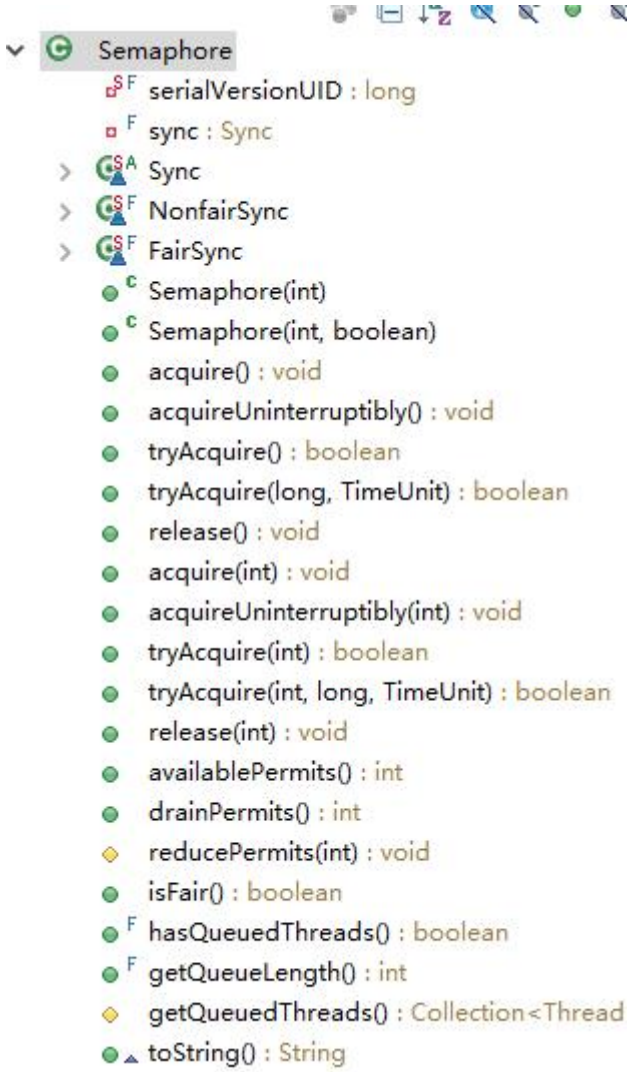
34、说说 Semaphore 原理

Semaphore 直译为信号。实际上 Semaphore 可以看做是一个信号的集合。不同的线程能够从 Semaphore 中获取若干个信号量。当 Semaphore 对象持有的信号量不足时，尝试从 Semaphore 中获取信号的线程将会阻塞。直到其他线程将信号量释放以后，阻塞的线程会被唤醒，重新尝试获取信号量。

Semaphore 使用：

Semaphore 实现了信号量，概念上讲，一个信号量相当于持有一些许可（permits），线程可以调用 Semaphore 对象的 acquire()方法获取一个许可，调用 release()来归还一个许可。

信号量一般用来限制访问资源的线程数量。



1 构造方法：

Semaphore 有两个构造方法 Semaphore(int)、Semaphore(int,boolean)，参数中的 int 表示该信号量拥有的许可数量，boolean 表示获取许可的时候是否是公平的，如果是公平的那么，当有多个线程要获取许可时，会按照线程来的先后顺序分配许可，否则，线程获得许可的顺序是不定的。

2 获取许可

可以使用 acquire()、acquire(int)、tryAcquire()等去获取许可，其中 int 参数表示一次性要获取几个许可，默认为 1 个，acquire 方法在没有许可的情况下，要获取许可的线程会阻塞，而 tryAcquire()方法在没有许可的情况下会立即返回 false，要获取许可的线程不会阻塞。

3 释放许可

线程可调用 release()、release(int)来释放（归还）许可，注意一个线程调用 release()之前并不要求一定要调用了 acquire

35、说说 Exchanger 原理

当一个线程到达 exchange 调用点时，如果它的伙伴线程此前已经调用了此方法，那么它的伙伴会被调度唤醒并与之进行对象交换，然后各自返回。如果它的伙伴还没到达交换点，那么当前线程将会被挂起，直至伙伴线程到达——完成交换正常返回；或者当前线程被中断——抛出中断异常；又或者是等候超时——抛出超时异常。

1.实现原理

Exchanger（交换者）是一个用于线程间协作的工具类。Exchanger 用于进行线程间的数据交换。它提供一个同步点，在这个同步点两个线程可以交换彼此的数据。这两个线程通过 exchange 方法交换数据， 如果第一个线程先执行 exchange 方法，它会一直等待第二个线程也执行 exchange，当两个线程都到达同步点时，这两个线程就可以交换数据，将本线程生产出来的数据传递给对方。因此使用 Exchanger 的重点是成对的线程使用 exchange()方法，当有一对线程达到了同步点，就会进行交换数据。因此该工具类的线程对象是成对的。

Exchanger 类提供了两个方法，String exchange(V x):用于交换，启动交换并等待另一个线程调用 exchange；String exchange(V x,long timeout,TimeUnit unit)：用于交换，启动交换并等待另一个线程调用 exchange，并且设置最大等待时间，当等待时间超过 timeout 便停止等待。

2.实例讲解

通过以上的原理，可以知道使用 Exchanger 类的核心便是 exchange()方法的使用，接下来通过一个例子来使的该工具类的用途更加清晰。该例子主要讲解的是前段时间 NBA 交易截止日的交易。

```
public class ExchangerDemo {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();
        final Exchanger exchanger = new Exchanger();
        executor.execute(new Runnable() {
            String data1 = "克拉克森，小拉里南斯";
            @Override
            public void run() {
                nbaTrade(data1, exchanger);
            }
        });
        executor.execute(new Runnable() {
            String data1 = "格里芬";
            @Override
            public void run() {
                nbaTrade(data1, exchanger);
            }
        });
        executor.execute(new Runnable() {
            String data1 = "哈里斯";
            @Override
            public void run() {
                nbaTrade(data1, exchanger);
            }
        });
        executor.execute(new Runnable() {
            String data1 = "以赛亚托马斯，弗莱";
            @Override
```

```
                public void run() {
                    nbaTrade(data1, exchanger);
                }
            });
            executor.shutdown();
        }
        private static void nbaTrade(String data1, Exchanger exchanger) {
            try {
                System.out.println(Thread.currentThread().getName() + "在交易截止之前把
" + data1 + "  交易出去");
                Thread.sleep((long) (Math.random() * 1000));
                String data2 = (String) exchanger.exchange(data1);
                System.out.println(Thread.currentThread().getName() + "交易得到" + data2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

当两个线程之间出现数据交换的情况，可以使用 Exchanger 工具类实现数据交换。注意 exchange 方法的含义，以及触发数据交换的条件。

36、说说 CountDownLatch 与 CyclicBarrier 区别

(01) CountDownLatch 的作用是允许 1 或 N 个线程等待其他线程完成执行;而 CyclicBarrier 则是允许 N 个线程相互等待。

(02) CountDownLatch 的计数器无法被重置;CyclicBarrier 的计数器可以被重置后使用,因此它被称为是循环的 barrier。

CountDownLatch	CyclicBarrier
减计数方式	加计数方式
计算为0时释放所有等待的线程	计数达到指定值时释放所有等待线程
计数为0时，无法重置	计数达到指定值时，计数置为0重新开始
调用countDown()方法计数减一，调用await()方法只进行阻塞，对计数没任何影响	调用await()方法计数加1，若加1后的值不等于构造方法的值，则线程阻塞
不可重复利用	可重复利用

CountDownLatch 是一个同步的辅助类，允许一个或多个线程，等待其他一组线程完成操作，再继续执行。

CyclicBarrier 是一个同步的辅助类，允许一组线程相互之间等待，达到一个共同点，再继续执行。

他们都是:Synchronization aid，我把它翻译成同步辅助器，既然是辅助工具，怎么使用啊？哪些场景使用啊？

个人理解：CountDownLatch:我把他理解成倒计时锁

场景还原：一年级期末考试要开始了，监考老师发下去试卷，然后坐在讲台旁边玩着手机等待着学生答题，有的学生提前交了试卷，并约起打球了，等到最后一个学生交卷了，老

师开始整理试卷，贴封条，下班，陪老婆孩子去了。

个人理解：CyclicBarrier:可看成是个障碍，所有的线程必须到齐后才能一起通过这个障碍.

场景还原：以前公司组织户外拓展活动，帮助团队建设，其中最重要一个项目就是全体员工（包括女同事，BOSS）在完成其他项目时，到达一个高达四米的高墙没有任何抓点，要求所有人，一个不能少的越过高墙，才能继续进行其他项目。

37、ThreadLocal 原理分析

ThreadLocal 提供了线程本地变量，它可以保证访问到的变量属于当前线程，每个线程都保存有一个变量副本，每个线程的变量都不同。ThreadLocal 相当于提供了一种线程隔离，将变量与线程相绑定。

ThreadLocal 原理分析:

首先，在每个线程 Thread 内部有一个 ThreadLocal.ThreadLocalMap 类型的成员变量 threadLocals，这个 threadLocals 就是用来存储实际的变量副本的，键值为当前 ThreadLocal 变量，value 为变量副本（即 T 类型的变量）。

初始时，在 Thread 里面，threadLocals 为空，当通过 ThreadLocal 变量调用 get()方法或者 set()方法，就会对 Thread 类中的 threadLocals 进行初始化，并且以当前 ThreadLocal 变量为键值，以 ThreadLocal 要保存的副本变量为 value，存到 threadLocals。 然后在当前线程里面，如果要使用副本变量，就可以通过 get 方法在 threadLocals 里面查找。

1）实际的通过 ThreadLocal 创建的副本是存储在每个线程自己的 threadLocals 中的；

2）为何 threadLocals 的类型 ThreadLocalMap 的键值为 ThreadLocal 对象，因为每个线程中可有多个 threadLocal 变量，就像上面代码中的 longLocal 和 stringLocal；

3）在进行 get 之前，必须先 set，否则会报空指针异常；

如果想在 get 之前不需要调用 set 就能正常访问的话，必须重写 initialValue()方法。

因为在上面的代码分析过程中，我们发现如果没有先 set 的话，即在 map 中查找不到对应的存储，则会通过调用 setInitialValue 方法

返回 i，而在 setInitialValue 方法中，有一个语句是 T value = initialValue()， 而默认情况下，initialValue 方法返回的是 null。

ThreadLocal 的应用场景：

最常见的 ThreadLocal 使用场景为 用来解决 数据库连接、Session 管理等。

```
public class Document {
static ThreadLocal<Long> longLocal = new ThreadLocal<Long>();
    static ThreadLocal<String> stringLocal = new ThreadLocal<String>();
    public static void set() {
        longLocal.set(Thread.currentThread().getId());
        stringLocal.set(Thread.currentThread().getName());
    }
    public static long getLong() {
        return longLocal.get();
    }
    public static String getString() {
        return stringLocal.get();
    }
    public static void main(String[] args) throws InterruptedException {
```

```
        final Document test = new Document();
        Document.set();
        System.out.println("--------------------------1");
        System.out.println(test.getLong());
        System.out.println(test.getString());
        Thread thread1 = new Thread(() -> {
            test.set();
            System.out.println("--------------------------2");
            System.out.println(test.getLong());
            System.out.println(test.getString());
        });
        thread1.start();
        thread1.join();//线程 thread1 加进来，先执行 2，执行完后再执行 3
        System.out.println("--------------------------3");
        System.out.println(test.getLong());
        System.out.println(test.getString());
        /**
         最后打印结果为：
         -------------------------1
         1
         main
         -------------------------2
         11
         Thread-0
         -------------------------3
         1
         main
```

在 main 线程中和 thread1 线程中，longLocal 保存的副本值和 stringLocal 保存的副本值都不一样。

最后一次在 main 线程再次打印副本值是为了证明在 main 线程中和 thread1 线程中的副本值确实是不同的。

```
        */
    }
}
```

38、 讲讲线程池的实现原理

当提交一个新任务到线程池时，线程池的处理流程如下。

1）线程池判断核心线程池里的线程是否都在执行任务。如果不是，则创建一个新的工作线程来执行任务。如果核心线程池里的线程都在执行任务，则进入下个流程。

2）线程池判断工作队列是否已经满。如果工作队列没有满，则将新提交的任务存储在这个工作队列里。如果工作队列满了，则进入下个流程。

3）线程池判断线程池的线程是否都处于工作状态。如果没有，则创建一个新的工作线程来执行任务。如果已经满了，则交给饱和策略来处理这个任务。

39、线程池的几种方式

在 Executors 类里面提供了一些静态工厂，生成一些常用的线程池。

1、newFixedThreadPool：创建固定大小的线程池。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

2、newCachedThreadPool：创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60 秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说 JVM）能够创建的最大线程大小。

3、newSingleThreadExecutor：创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

4、newScheduledThreadPool：创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

5、newSingleThreadScheduledExecutor：创建一个单线程的线程池。此线程池支持定时以及周期性执行任务的需求。

40、线程的生命周期

1.线程的生命周期

线程是一个动态执行的过程，它也有一个从产生到死亡的过程。

(1)生命周期的五种状态

新建（new Thread）

当创建 Thread 类的一个实例（对象）时，此线程进入新建状态（未被启动）。

例如：Thread　t1=new Thread();

就绪（runnable）

线程已经被启动，正在等待被分配给 CPU 时间片，也就是说此时线程正在就绪队列中排队等候得到 CPU 资源。例如：t1.start();

运行（running）

线程获得 CPU 资源正在执行任务（run()方法），此时除非此线程自动放弃 CPU 资源或者有优先级更高的线程进入，线程将一直运行到结束。

死亡（dead）

当线程执行完毕或被其它线程杀死，线程就进入死亡状态，这时线程不可能再进入就绪状态等待执行。

自然终止：正常运行 run()方法后终止

异常终止：调用 stop()方法让一个线程终止运行

堵塞（blocked）

由于某种原因导致正在运行的线程让出 CPU 并暂停自己的执行，即进入堵塞状态。

正在睡眠：用 sleep(long t) 方法可使线程进入睡眠方式。一个睡眠着的线程在指定的时间过去可进入就绪状态。

正在等待：调用 wait()方法。（调用 motify()方法回到就绪状态）

被另一个线程所阻塞：调用 suspend()方法。（调用 resume()方法恢复）

2.常用方法

void run()　创建该类的子类时必须实现的方法

void start() 开启线程的方法

static void sleep(long t) 释放 CPU 的执行权，不释放锁
static void sleep(long millis,int nanos)
final void wait()释放 CPU 的执行权，释放锁
final void notify()
static void yied()可以对当前线程进行临时暂停（让线程将资源释放出来）
3.（1）结束线程原理：就是让 run 方法结束。而 run 方法中通常会定义循环结构，所以只要控制住循环即可
(2)方法----可以 boolean 标记的形式完成，只要在某一情况下将标记改变，让循环停止即可让线程结束
（3）public final void join()//让线程加入执行，执行某一线程 join 方法的线程会被冻结，等待某一线程执行结束，该线程才会恢复到可运行状态
4. 临界资源：多个线程间共享的数据称为临界资源
（1）互斥锁
a.每个对象都对应于一个可称为“互斥锁”的标记，这个标记用来保证在任一时刻，只能有一个线程访问该对象。
b.Java 对象默认是可以被多个线程共用的，只是在需要时才启动“互斥锁”机制，成为专用对象。
c.关键字 synchronized 用来与对象的互斥锁联系
d.当某个对象用 synchronized 修饰时，表明该对象已启动“互斥锁”机制，在任一时刻只能由一个线程访问，即使该线程出现堵塞，该对象的被锁定状态也不会解除，其他线程任不能访问该对象。

锁机制

41、 说说线程安全问题

线程安全是多线程领域的问题，线程安全可以简单理解为一个方法或者一个实例可以在多线程环境中使用而不会出现问题。

在 Java 多线程编程当中，提供了多种实现 Java 线程安全的方式：

最简单的方式，使用 Synchronization 关键字:Java Synchronization 介绍

使用 java.util.concurrent.atomic 包中的原子类，例如 AtomicInteger

使用 java.util.concurrent.locks 包中的锁

使用线程安全的集合 ConcurrentHashMap

使用 volatile 关键字，保证变量可见性（直接从内存读，而不是从线程 cache 读）

42、volatile 实现原理

在 JVM 底层 volatile 是采用“内存屏障”来实现的。

缓存一致性协议（MESI 协议）它确保每个缓存中使用的共享变量的副本是一致的。其核心思想如下：当某个 CPU 在写数据时，如果发现操作的变量是共享变量，则会通知其他 CPU 告知该变量的缓存行是无效的，因此其他 CPU 在读取该变量时，发现其无效会重新从主存中加载数据。

并发编程中的三个概念：原子性问题，可见性问题，有序性问题。

Java 内存模型：原子性，可见性，有序性。

1. volatile 关键字的两层语义

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：

1）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。

2）禁止进行指令重排序

2. 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？、

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。

3. 保证原子性：

```
public class Test {
    public    int inc = 0;
    （1）
    public synchronized void increase() {
        inc++;
    }
    （2）
        Lock lock = new ReentrantLock();
        public    void increase() {
            lock.lock();
            try {
                inc++;
            } finally{
                lock.unlock();
            }
        }
    （3）public    AtomicInteger inc = new AtomicInteger();
        public    void increase() {
            inc.getAndIncrement();
        }

    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);
    }
}
```

4.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：

1）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；

2）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。

5.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：

1）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；

2）它会强制将对缓存的修改操作立即写入主存；

3）如果是写操作，它会导致其他 CPU 中对应的缓存行无效。

使用 volatile 关键字的场景

6.synchronized 关键字是防止多个线程同时执行一段代码，那么就会很影响程序执行效率，而 volatile 关键字在某些情况下性能要优于 synchronized，但是要注意 volatile 关键字是无法替代 synchronized 关键字的，因为 volatile 关键字无法保证操作的原子性。通常来说，使用 volatile 必须具备以下 2 个条件：

1）对变量的写操作不依赖于当前值

2）该变量没有包含在具有其他变量的不变式中

实际上，这些条件表明，可以被写入 volatile 变量的这些有效值独立于任何程序的状态，包括变量的当前状态。

事实上，我的理解就是上面的 2 个条件需要保证操作是原子性操作，才能保证使用 volatile 关键字的程序在并发时能够正确执行。

下面列举几个 Java 中使用 volatile 的几个场景。

状态标志量：

```
volatile boolean flag = false;
while(!flag){
    doSomething();
}
public void setFlag() {
    flag = true;
}
```

double check：

```
class Singleton{
```

```
    private volatile static Singleton instance = null;
    private Singleton() {
    }
    public static Singleton getInstance() {
        if(instance==null) {
            synchronized (Singleton.class) {
                if(instance==null)
                    instance = new Singleton();
            }
        }
        return instance;
    }
}
```

43、synchronize 实现原理

同步代码块是使用 monitorenter 和 monitorexit 指令实现的，同步方法（在这看不出来需要看 JVM 底层实现）依靠的是方法修饰符上的 ACC_SYNCHRONIZED 实现。

44、synchronized 与 lock 的区别

一、synchronized 和 lock 的用法区别

（1）synchronized(隐式锁)：在需要同步的对象中加入此控制，synchronized 可以加在方法上，也可以加在特定代码块中，括号中表示需要锁的对象。

（2）lock（显示锁）：需要显示指定起始位置和终止位置。一般使用 ReentrantLock 类做为锁，多个线程中必须要使用一个 ReentrantLock 类做为对 象才能保证锁的生效。且在加锁和解锁处需要通过 lock()和 unlock()显示指出。所以一般会在 finally 块中写 unlock()以防死锁。

二、synchronized 和 lock 性能区别

synchronized 是托管给 JVM 执行的，而 lock 是 java 写的控制锁的代码。在 Java1.5 中，synchronize 是性能低效的。因为 这是一个重量级操作，需要调用操作接口，导致有可能加锁消耗的系统时间比加锁以外的操作还多。相比之下使用 Java 提供的 Lock 对象，性能更高一些。但 是到了 Java1.6，发生了变化。synchronize 在语义上很清晰，可以进行很多优化，有适应自旋，锁消除，锁粗化，轻量级锁，偏向锁等等。导致 在 Java1.6 上 synchronize 的性能并不比 Lock 差。

三、synchronized 和 lock 机制区别

（1）synchronized 原始采用的是 CPU 悲观锁机制，即线程获得的是独占锁。独占锁意味着其 他线程只能依靠阻塞来等待线程释放锁。

（2）Lock 用的是乐观锁方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。乐观锁实现的机制就 是 CAS 操作（Compare and Swap）。

1、ReentrantLock 拥有 Synchronized 相同的并发性和内存语义，此外还多了 锁投票，定时锁等候和中断锁等候

线程 A 和 B 都要获取对象 O 的锁定，假设 A 获取了对象 O 锁，B 将等待 A 释放对 O 的锁定，

如果使用 synchronized ，如果 A 不释放，B 将一直等下去，不能被中断

如果 使用 ReentrantLock，如果 A 不释放，可以使 B 在等待了足够长的时间以后，中断等待，而干别的事情

ReentrantLock 获取锁定与三种方式：

a) lock(), 如果获取了锁立即返回，如果别的线程持有锁，当前线程则一直处于休眠状态，直到获取锁

b) tryLock(), 如果获取了锁立即返回 true，如果别的线程正持有锁，立即返回 false；

c)tryLock(long timeout,TimeUnit unit)， 如果获取了锁定立即返回 true，如果别的线程正持有锁，会等待参数给定的时间，在等待的过程中，如果获取了锁定，就返回 true，如果等待超时，返回 false；

d) lockInterruptibly:如果获取了锁定立即返回，如果没有获取锁定，当前线程处于休眠状态，直到或者锁定，或者当前线程被别的线程中断

2、synchronized 是在 JVM 层面上实现的，不但可以通过一些监控工具监控 synchronized 的锁定，而且在代码执行时出现异常，JVM 会自动释放锁定，但是使用 Lock 则不行，lock 是通过代码实现的，要保证锁定一定会被释放，就必须将 unLock()放到 finally{}中

3、在资源竞争不是很激烈的情况下，Synchronized 的性能要优于 ReetrantLock，但是在资源竞争很激烈的情况下，Synchronized 的性能会下降几十倍，但是 ReetrantLock 的性能能维持常态；

5.0 的多线程任务包对于同步的性能方面有了很大的改进，在原有 synchronized 关键字的基础上，又增加了 ReentrantLock，以及各种 Atomic 类。了解其性能的优劣程度，有助与我们在特定的情形下做出正确的选择。

总体的结论先摆出来：

synchronized：

在资源竞争不是很激烈的情况下，偶尔会有同步的情形下，synchronized 是很合适的。原因在于，编译程序通常会尽可能的进行优化 synchronize，另外可读性非常好，不管用没用过 5.0 多线程包的程序员都能理解。

ReentrantLock:

ReentrantLock 提供了多样化的同步，比如有时间限制的同步，可以被 Interrupt 的同步（synchronized 的同步是不能 Interrupt 的）等。在资源竞争不激烈的情形下，性能稍微比 synchronized 差点点。但是当同步非常激烈的时候，synchronized 的性能一下子能下降好几十倍。而 ReentrantLock 确还能维持常态。

Atomic:

和上面的类似，不激烈情况下，性能比 synchronized 略逊，而激烈的时候，也能维持常态。激烈的时候，Atomic 的性能会优于 ReentrantLock 一倍左右。但是其有一个缺点，就是只能同步一个值，一段代码中只能出现一个 Atomic 的变量，多于一个同步无效。因为他不能在多个 Atomic 之间同步。

45、CAS 乐观锁

所谓原子操作类，指的是 java.util.concurrent.atomic 包下，一系列以 Atomic 开头的包装类。例如 AtomicBoolean，AtomicInteger，AtomicLong。它们分别用于 Boolean，Integer，Long 类型的原子性操作。

CAS 是项乐观锁技术，当多个线程尝试使用 CAS 同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争

中失败，并可以再次尝试。

CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值。否则，处理器不做任何操作。无论哪种情况，它都会在 CAS 指令之前返回该位置的值。（在 CAS 的一些特殊情况下将仅返回 CAS 是否成功，而不提取当前值。）CAS 有效地说明了“我认为位置 V 应该包含值 A；如果包含该值，则将 B 放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。”这其实和乐观锁的冲突检查+数据更新的原理是一样的。

CAS 的缺点：

1.CPU 开销较大

在并发量比较高的情况下，如果许多线程反复尝试更新某一个变量，却又一直更新不成功，循环往复，会给 CPU 带来很大的压力。

2.不能保证代码块的原子性

CAS 机制所保证的只是一个变量的原子性操作，而不能保证整个代码块的原子性。比如需要保证 3 个变量共同进行原子性的更新，就不得不使用 Synchronized 了。

46、ABA 问题

CAS 会导致“ABA 问题”。

CAS 算法实现一个重要前提需要取出内存中某时刻的数据，而在下时刻比较并替换，那么在这个时间差类会导致数据的变化。

比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但是不代表这个过程就是没有问题的。

部分乐观锁的实现是通过版本号（version）的方式来解决 ABA 问题，乐观锁每次在执行数据的修改操作时，都会带上一个版本号，一旦版本号和数据的版本号一致就可以执行修改操作并对版本号执行+1 操作，否则就执行失败。因为每次操作的版本号都会随之增加，所以不会出现 ABA 问题，因为版本号只会增加不会减少。

47、乐观锁的业务场景及实现方式

乐观锁（Optimistic Lock）：

每次获取数据的时候，都不会担心数据被修改，所以每次获取数据的时候都不会进行加锁，但是在更新数据的时候需要判断该数据是否被别人修改过。如果数据被其他线程修改，则不进行数据更新，如果数据没有被其他线程修改，则进行数据更新。由于数据没有进行加锁，期间该数据可以被其他线程进行读写操作。

比较适合读取操作比较频繁的场景，如果出现大量的写入操作，数据发生冲突的可能性就会增大，为了保证数据的一致性，应用层需要不断的重新获取数据，这样会增加大量的查询操作，降低了系统的吞吐量。