K:首先解释一下为什么fina可以保证对象正确的初始化，首先要知道原因。因为：

A a = new A() 这个语句需要两步才能执行完毕。就是这两步是有可能发生重排序。而final在可以达到不重排序的目的。更何况远远不止两步。（内存分配，初始化对象，设置实例指向内存空间，初次访问对象）

Q：回想一下，static是如何实现正确初始化的？或者说类初始化方案是如何成立的

A：解决思路就是jvm级别保证的类初始化，同时也叫作基于类初始化的静态方案

基于类的初始化方案：

第一种非延迟初始化方案，直接使用静态域并new，触发类的初始化，jvm会保证初始化正确，但是这种问题就是如果有些域不用就会浪费，一般我们会选择延迟初始化。如果仅仅是在静态方法里new，

|  |
| --- |
| public class SingleTest {  private static CaptureImgBean *captureImgBean*;  private CaptureImgBean getInstance(){  *captureImgBean* = new CaptureImgBean();  return *captureImgBean*;  } }  延迟初始化可以做到，但是每次调用都会new一个对象，如果想要单例可以这样改进：  public class SingleTest {  private static CaptureImgBean *captureImgBean*;  private static CaptureImgBean getInstance(){  if(*captureImgBean* == null) {  *captureImgBean* = new CaptureImgBean();  }  return *captureImgBean*;  } }  这种有什么问题呢？当你判断为空后，这个时候可能另一个线程正好初始化完，不管加多少if(*captureImgBean* == null)都有可能这种情况  继续改进：  public class SingleTest {  private static CaptureImgBean *captureImgBean*;  private static CaptureImgBean getInstance(){  if(*captureImgBean* == null) {  synchronized (SingleTest.class) {  *captureImgBean* = new CaptureImgBean();  }  }  return *captureImgBean*;  } }  这个在进入锁后还是有可能已经不为空  public class SingleTest {  private static CaptureImgBean *captureImgBean*;  private static CaptureImgBean getInstance(){  if(*captureImgBean* == null) {  synchronized (SingleTest.class) {  if(*captureImgBean* == null) {  *captureImgBean* = new CaptureImgBean();  }  }  }  return *captureImgBean*;  } }  双重检查，但是还有问题，由于重排序，可能虽然不为空，但是并没有正常的初始化  这一点我不理解，因为既然静态变量，那么必然是结果类初始化的，为什么会出现未初始化完全的对象呢？除非*captureImgBean不是一个静态变量，这样我可以理解，但是这里不是，我认为作者说的不对。而且这样我可以推翻基于类类初始化创建对象的方案。如果他是对的，有一种解释，延迟加载场景下，并没有类初始化一说，而且你也看到了，基于类初始化的方案也是用到了静态类。*  *目前的解释：静态属性在方法中进行new，并不会受到jvm锁的保证。只有这样的解释才是行得通。*  *为了保证其能够正确初始化：*  public class SingleTest {  private static volatile CaptureImgBean *captureImgBean*;  public static CaptureImgBean getInstance(){  System.*err*.println("getInstance");  if(*captureImgBean* == null) {  synchronized (SingleTest.class) {  if(*captureImgBean* == null) {  *captureImgBean* = new CaptureImgBean();  }  }  }  return *captureImgBean*;  } }  有人会问问什么不用final，因为final在这里编译不会通过，final会要求声明时就进行初始化。  然后再说下基于类初始化的方案：  public class SingleTest {  static class InstanceHolder{  static CaptureImgBean *captureImgBean* = new CaptureImgBean();  }  public static CaptureImgBean getInstance(){  return InstanceHolder.*captureImgBean*;  } } |

Q:到底应该锁什么？还是锁什么都可以？

A：第一锁的一定要有层级关系，有了层级关系，接下来的要考虑的就是锁的粒度。一个类里有其他类或其它对象，name这个类就是最高层级，如果在这个类上加锁，粒度就是比较粗的。

Q:中断的理解，中断对于线程来说是个标志位，如果有些api中没有判断中断标志，那么就不会响应中断。而有一些，比如sleep（）方法

public static native void sleep(long millis) throws InterruptedException;

虽然调用的native方法，但是抛出InterruptedException，说明是可中断的。

中断标志位false的情况，第一种线程已经结束，第二种，线程抛出异常，如果说一种也可以，一种是正常结束，一种是非正常结束。

Q：Lock是怎么实现的，是syn的封装么？

A:观察源码你会发现，根本就没有出现syn关键词。通过队列的设计模式，采用cas操作实现。

Q：锁的位置

A：第一不要在try内部，据说会导致锁无法正常释放，第二，不要作为局部变量，否则每个线程都是新的AQS，加锁的意义不在

Q:为什么不能在try内部

A:应该说的是锁对象不应该在try内部，因为final捕获不到，而且异常并不会主动释放锁。关键点在于锁获取后不会因为异常的发生而主动释放锁。

Q: 共享锁，独占锁，公平锁，非公平锁有什么区别？

A:在源代码里我看到只有公平锁和不公平锁，不公平锁就是独占锁，不公平锁就是共享锁。

Q:什么是独占，什么是共享

A:独占是读写永远都只会有一个线程，而共享锁，允许有多个线程同时读，但不允许有多个线程同时写。

K：要重点关注acquire方法，以及头结点的处理

Q:公平锁与非公平锁

A：公平锁会保证FIFO，比如会判断当前节点是否有前驱节点（prev和head别搞混了）

看下公平锁的源代码：

final void lock() {  
 acquire(1);  
}

非公平锁的源代码：

final void lock() {  
 if (compareAndSetState(0, 1))  
 setExclusiveOwnerThread(Thread.*currentThread*());  
 else  
 acquire(1);  
}

你会发现非公平锁会先尝试获取锁，获取不到才会去调用acquire，走队列流程，这样说的话，其实也是存在巧合的，抢到就是你的，取不到乖乖等着。

Q：锁的实现

A：重入锁ReentrantLock，ReadWriteLock

Q：读写锁跟共享锁的区别

A:共享锁允许多个线程同时读，但是写的时候就会出现幻读，但是读写锁，会在有线程写的时候全部锁住共享读

Q:使用AQS为什么不考虑使用LinkedList，难道LinkedList不可以进行原子操作还是怎样？

Q:HashMap并发执行put操作会引起死循环？

A:问题是rehash，rehash有个特点是节点是反着来的，导致遍历会存在next不为空的情况而产生死循环。

关键代码在这里：

<https://blog.csdn.net/bnmb888/article/details/77164485>

值得一提的是，这里1.8跟1.7有所不同，1.7会产生倒序

else { // preserve order  
 Node<K,V> loHead = null, loTail = null;//留在原位的元素  
 Node<K,V> hiHead = null, hiTail = null;//移向其它位的元素  
 Node<K,V> next;  
 do {  
 next = e.next;  
 if ((e.hash & oldCap) == 0) {  
 if (loTail == null)  
 loHead = e;  
 else  
 loTail.next = e;  
 loTail = e;  
 }  
 else {  
 if (hiTail == null)  
 hiHead = e;  
 else  
 hiTail.next = e;  
 hiTail = e;  
 }  
 } while ((e = next) != null);  
 if (loTail != null) {  
 loTail.next = null;  
 newTab[j] = loHead;  
 }  
 if (hiTail != null) {  
 hiTail.next = null;  
 newTab[j + oldCap] = hiHead;  
 }  
}

理解e.hash & oldCap

|  |
| --- |
| JDK1.7中，resize时，index取得时，全部采用重新hash的方式进行了。JDK1.8对这个进行了改善。  以前要确定index的时候用的是(e.hash & oldCap-1)，是取模取余，而这里用到的是(e.hash & oldCap)，它有两种结果，一个是0，一个是oldCap，  比如oldCap=8,hash是3，11，19，27时，(e.hash & oldCap)的结果是0，8，0，8，这样3，19组成新的链表，index为3；而11，27组成新的链表，新分配的index为3+8；  JDK1.7中重写hash是(e.hash & newCap-1)，也就是3，11，19，27对16取余，也是3，11，3，11，和上面的结果一样，但是index为3的链表是19，3，index为3+8的链表是  27，11，也就是说1.7中经过resize后数据的顺序变成了倒叙，而1.8没有改变顺序。 |

这个地方也是要研究的

Q:hashmap的循环问题？JDK8还会产生循环问题么

参考资料：<https://www.cnblogs.com/Xieyang-blog/p/8886921.html>