# ======线程的基本概念========

## 线程和进程的概念 √

1. 简单回答：

**进程**就是一个程序运行起来的状态，**线程**是一个进程中的不同的执行路径，进程的最小执行单元。

（你硬盘上一个QQ.exe是一个程序，双击运行，就叫一个进程，相对于程序来说进程是动态的概念。）

1. 专业回答：

进程是OS分配资源的基本单位，线程是执行调度的基本单位。

分配资源最重要的是：独立的内存空间，线程调度执行（线程共享进程的内存空间，没有自己独立的内存空间）

windows、linux、unix都是多进程、多线程的，dos是只支持单进程的，同一个时间点只能执行一个进程执行。

**纤程：**

用户态的线程，线程中的线程，切换和调度不需要经过OS

优势：1：占有资源很少 OS : 线程1M Fiber：4K 2：切换比较简单 3：启动很多个10W+

目前2020 3 22支持内置纤程的语言：Kotlin Scala Go Python(lib)... Java? （open jdk : loom）

Java中对于纤程的支持：没有内置，盼望内置

纤程 vs 线程池：很短的计算任务，不需要和内核打交道，并发量高！

## 创建线程的方式及实现 √

创建线程的三种方式：

* **new** Thread().start();
* **new** Thread(Runnable).start();
* 吹毛求疵：

线程池也是用的也是用的这两种之一

|  |
| --- |
| ExecutorService service = Executors.*newCachedThreadPool*();  service.execute(()->{  System.***out***.println("KKKKK");  });  service.shutdown();  ////// FutureTask+Callable ///////  **new** Thread(**new** FutureTask<Integer>(**new** MyCall())).start(); |

* JDK8: **new** Thread(()->{ }).start();

## 线程常见方法 √

### Thread.sleep();

线程睡眠指定的毫秒数，睡完后进入就绪状态

### Join

合并某个线程，等待某个线程执行结束，再恢复当前线程的运行。

在t1线程调用t2.join ，t1等待t2线程执行结束，t1再继续执行。

**问： 启动三个线程怎么让他们按顺序执行？**

在t1里调用t2.join，在t2里调t3.join



### Yield()

高风亮节，当前线程让出一下CPU，进到等待队列，给其他线程执行的机会，你能不能抢到cpu我就不管了。

### Priority

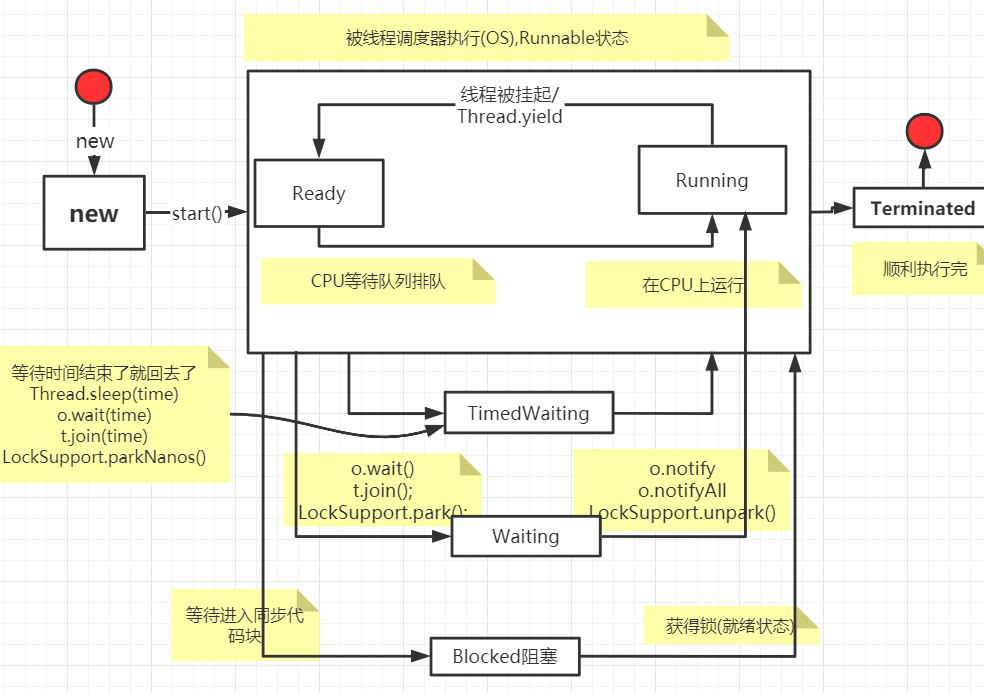
线程优先级

## 线程的生命周期，状态是如何转移的 √

六种状态

Thread.State 枚举类有6个状态：

***NEW，RUNNABLE，BLOCKED，WAITING，TIMED\_WAITING，TERMINATED***



* **（1）新建**

New一个线程，还没有调用start(); 时处于新建状态。

线程对象调用start();方法时，它会被线程调度器来执行（也就是交给操作系统执行），操作系统执行的时候，这整个状态叫**（2）Runnable状态**，Runnable内部又有两个状态：

* 1. **Ready就绪状态**

扔到CPU的等待队列里去等待CPU运行

* 1. **Running状态**

等真正扔到CPU上去运行的时候叫Running.（调用yield方法，从Running🡪Ready，线程调度器选中执行的时候 Ready🡪Running）

* **（3）Terminaled结束状态**

Runnable还有一些其他状态：

线程顺利执行完了进入Terminaled结束状态（不可以再回到new状态调用start，这就算结束了）

* **（4）TimedWaiting等待**

按照时间等待，等待时间结束了自己就到了Running状态。

Thread.sleep(time); 、o.wait(time)；、t.join(time)；、LockSupport.parkNanos()都是关于时间等待的方法

* **（5）Waiting等待**

在运行过程中，如果调用了o.wait()、t.join(); 、LockSupport.park();进入waiting状态；

调用o.notify();、o.notifyAll(); 、LockSupport.unpark();又回到Running状态

* **（6）Block阻塞**

同步代码块没有获得锁就会阻塞状态，获得锁就是就绪状态

### 追问：这些状态，哪些是JVM管理的，哪些是操作系统管理的？

这些状态全是由JVM管理的，因为JVM管理的时候也要通过操作系统，所以，哪个是JVM哪个是操作系统，他俩分不开。JVM是跑在操作系统上的一个普通程序。

### 追问：线程什么时候会被挂起？挂起是否也是一个状态

Running的时候，在一个CPU上会跑很多个线程，CPU会隔一段时间执行这个线程一下，在隔一段时间执行那个线程一下，这是CPU内部的一个调度，把这个状态线程扔出去，从Running扔出去，就叫线程被挂起，CPU控制它。

## 说说线程安全问题，什么是线程安全，如何保证线程安全

线程安全：当多个线程访问某一个类（对象或方法）时，这个类始终都能表现出正确的行为，那么这个类（对象或方法）就是线程安全的。

# =========Synchronized============

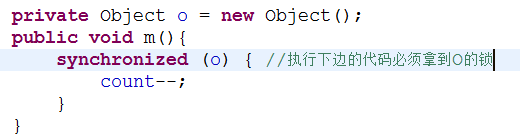
## Synchronized 基本概念√

### 概念：

可以在任意对象及方法上加锁，而加锁的这段代码成为“互斥区” 或 “临界区”

### 对象锁：

锁定的并不是count数字，可以任意指定，想锁谁就锁谁。



也可以不new对象锁定，可以**synchronized** (**this**)锁定当前对象。

### 类锁：

\*\*静态方法static是没有this对象的，static方法加synchronized就是synchronized(T.class);锁的是T类的对象

### 代码示例：

多个线程一把锁 。会有锁竞争问题，消耗CPU资源

|  |
| --- |
| **public** **class** MyThread **extends** Thread{    **private** **int** count = 5 ;    //加锁  @Override  **public** **synchronized** **void** run(){  count--;  System.***out***.println(**this**.*currentThread*().getName() + " count = "+ count);  }    **public** **static** **void** main(String[] args) {  /\*\*  \* 分析：当多个线程访问myThread的run方法时，以排队的方式进行处理（这里排对是按照CPU分配的先后顺序而定的），  \* 一个线程想要执行synchronized修饰的方法里的代码：  \* 1 尝试获得锁  \* 2 如果拿到锁，执行synchronized代码体内容；拿不到锁，这个线程就会不断的尝试获得这把锁，直到拿到为止，  \* 而且是多个线程同时去竞争这把锁。（也就是会有锁竞争的问题）  \*/  MyThread myThread = **new** MyThread();  Thread t1 = **new** Thread(myThread,"t1");  Thread t2 = **new** Thread(myThread,"t2");  Thread t3 = **new** Thread(myThread,"t3");  Thread t4 = **new** Thread(myThread,"t4");  Thread t5 = **new** Thread(myThread,"t5");  t1.start();  t2.start();  t3.start();  t4.start();  t5.start();  }  }  打印：和心里预期一样，就是线程安全的 |

扩展：如果是多个线程，每个线程锁定自己的对象，（多个线程多把锁）就不存在锁竞争问题，每个线程正常执行。

## synchronized方法和非synchronized方法能否同时被多个调用？√

有一个类T，有一个synchronized方法m1，一个普通方法m2，new T()；一个对象，多个线程能否同时调用synchronized方法和非synchronized方法？ 用大腿想想，肯定可以的。调m1的时候需要加锁，调m2的时候不需要加锁。

## 解释/为什么说Synchronized是可重入锁 \*\*\*\*\* √

Synchronized的一个重要属性：可重入。

一个对象里，一个同步方法可以调用另外一个同步方法，一个线程已经拥有某个对象的锁，再次申请的时候仍然会得到该对象的锁。也就是说synchronized获得锁是可重入的。

一个方法m1是synchronized的，另一个方法m2也是synchronized的，m1里调用m2，m1开始的时候获得了这把锁，然后m1里调m2，这时候m2发现是同一个线程，因为你m2也需要申请这把锁，他发现是同一个线程申请的这把锁，允许，这就叫可重入锁。

**原理：**

Synchronized一个对象的时候，在对象头上有2位标识是否加锁，一个线程锁定这个对象，这个线程又申请这把锁，发现当前持有这把锁的线程就是自己，就继续往下执行了。

## synchronized 出现异常，异常锁 问：sync内抛异常会怎样，死锁吗？还是释放掉？ √

程序出现异常，锁将被释放，别的准备拿到这把锁的线程乱冲进来，程序乱入。

所以在并发处理的过程中，有异常要多加小心，不然可能发生不一致的情况。

**发生异常得处理：**

* 情况一：

所有任务不是一个整体，一个有问题不影响其他，只把出错的记录日志，下次执行,此时catch应该捕捉Exception。

* 情况二：

所有任务是一个整体，一个有问题影响其他，此时可以通过捕捉InterruptedException、catch里抛出RuntimeException();来终止线程继续执行。

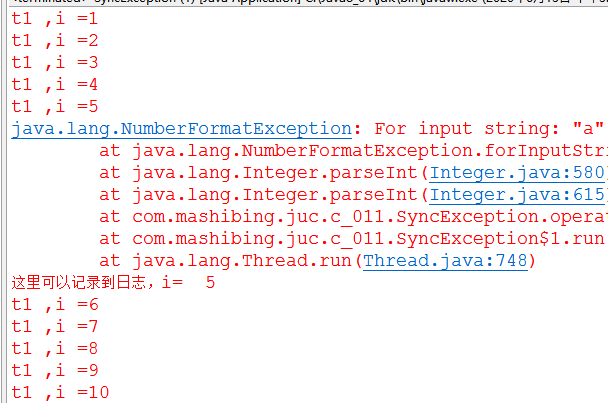
例子程序：

|  |
| --- |
| **import** java.util.concurrent.TimeUnit;  **public** **class** T {  **int** count = 0;  **synchronized** **void** m() {  System.***out***.println(Thread.*currentThread*().getName() + " start");  **while**(**true**) {  count ++;  System.***out***.println(Thread.*currentThread*().getName() + " count = " + count);  **try** {  TimeUnit.***SECONDS***.sleep(1);    } **catch** (InterruptedException e) {  e.printStackTrace();  }  **if**(count % 5 == 0) {  **int** i = 1/0; //此处抛出异常，锁将被释放，要想不被释放，可以在这里进行catch，然后让循环继续  System.***out***.println(i);  }  }  }  **public** **static** **void** main(String[] args) {  T t = **new** T();  Runnable r = **new** Runnable() {  @Override  **public** **void** run() {  t.m();  }  };  **new** Thread(r, "t1").start();    **try** {  TimeUnit.***SECONDS***.sleep(3);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **new** Thread(r, "t3").start();  **new** Thread(r, "t2").start();  **new** Thread(r, "t4").start();  **new** Thread(r, "t5").start();  **new** Thread(r, "t6").start();  **new** Thread(r, "t7").start();  }  } |



例子程序2：

|  |
| --- |
| **package** com.mashibing.juc.c\_011;  /\*\*  \* 模拟synchronized加锁的方法，发生异常时，怎么处理  \* **@author** dev  \*  \*/  **public** **class** SyncException {  **private** **int** i = 0;  **boolean** flag = **true**;    //传过来多个任务，taskSize模拟任务个数  **public** **synchronized** **void** operation(**int** taskSize){    **while**(flag){  **try** {  i++;  Thread.*sleep*(200);  System.***err***.println(Thread.*currentThread*().getName()+" ,i =" +i);  **if**(i == (taskSize/2)){//模拟任务执行了一半  Integer.*parseInt*("a"); //发生异常  //throw new RuntimeException();  }  //任务执行完了停止  **if**(i == taskSize){  shutdown();  }  } **catch** (Exception e) { //catch到InterruptedException，不再往下执行  e.printStackTrace();  //情况一：所有任务不是一个整体，一个有问题不影响其他，只把出错的记录日志，下次执行,此时catch应该捕捉Exception。/continue;  System.***err***.println("这里可以记录到日志，i= "+i);  //情况二：所有任务是一个整体，一个有问题影响其他，此时可以通过捕捉InterruptedException、catch里抛出RuntimeException();来终止线程继续执行  //throw new RuntimeException();  //continue;  }  }  }  //终止  **public** **void** shutdown(){  **this**.flag = **false**;  }    **public** **static** **void** main(String[] args) {  **final** SyncException se = **new** SyncException();  Thread t1 = **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  se.operation(10);  }  },"t1");  t1.start();    }  } |



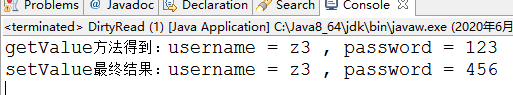
## synchronized 解决 脏读问题 √

对于对象的同步和异步的方法，我们在设计自己的程序的时候，一定要考虑问题的整体，不然就会出现数据不一致的错误，很经典的错误就是脏读。

例子程序

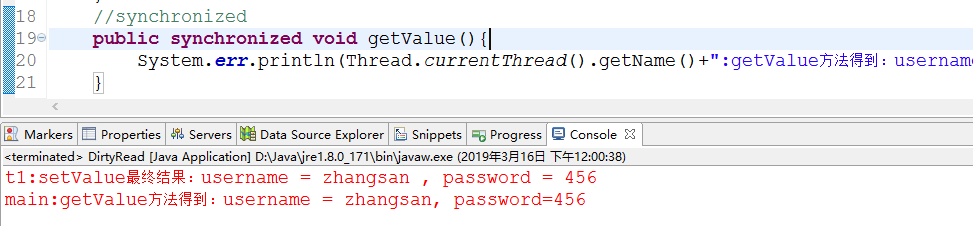
|  |
| --- |
| /\*\*  \* 业务整体需要使用完整的synchronized，保持业务的原子性。  \* **@author** alienware  \*  \*/  **public** **class** DirtyRead {  **private** String username = "bjsxt";  **private** String password = "123";    **public** **synchronized** **void** setValue(String username, String password){  **this**.username = username;  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **this**.password = password;  System.***out***.println("setValue最终结果：username = " + username + " , password = " + password);  }    **public** **void** getValue(){  System.***out***.println("getValue方法得到：username = " + **this**.username + " , password = " + **this**.password);  }      **public** **static** **void** main(String[] args) **throws** Exception{    **final** DirtyRead dr = **new** DirtyRead();  Thread t1 = **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  dr.setValue("z3", "456");  }  });  t1.start();  Thread.*sleep*(1000);    dr.getValue();  }  } |

执行结果：



t1线程先对username设置值，将初始值 lhy 修改为 zhangsan ，然后睡眠1秒，此时 password还是初始值 123，所以主线程在1秒的时候读取到的username是修改后的zhangsan，password是初始值123，等到2秒时，t1线程将password修改为456，t1线程执行结束。打印username=zhangsan 、password=456。

要想保证读写业务的一致性，getValue也应该加上synchornized关键字。加上之后打印的结果就是我们预期的：





## synchronized 底层实现 \*\*\*\*\*\*\*\*\*五星 （饿了么被问过）√

## synchronized关键字锁住的是什么东西？在字节码中是怎么表示的？在内存中的对象上表现为什么？

锁住的是对象头上的两位，字节码、内存里的表示往下看

### 一般回答：

早期JDK的时候，synchronized的底层实现是**重量级的**，要**去操作系统申请锁，**这就会造成synchronized效率非常低，java后来越来越处理高并发程序的时候，很多程序员不满意，说synchronized用的太重了，我没办法，就要开发新的框架，不用你原生的了。

后来JDK进行了改进，有了**锁升级**的概念：

当使用synchronize的时候，Hotspot的实现是这样的：上来之后第一个去访问某把锁的线程（比如sync(Object)） 来了之后，只是先在Object的对象头上的Markword记录这个线程的ID（这叫**偏向锁）**，实际并没有给这个Object加锁。

偏向锁如果有线程竞争的话，就升级为**自旋锁:** 概念就是：

有一个哥们在蹲马桶，另外来了一个哥们，他就在旁边等着，他不会跑到CPI的就绪队列离去，而就在这等着占用CPU，用一个while循环在这转圈，转很多圈之后不行的话就再进行一次升级。

自旋锁转圈十次之后，升级为**重量级锁，**重量级锁就是要去操作系统那里去申请资源。这是一个锁升级的过程。

#### 重量级锁IMG_256 Or 自旋锁IMG_256

**自旋锁：**占CPU，但不访问OS，在用户态解决，不经过内核态，加锁解锁效率要比经过内核态高。

**OS锁：**不占CPU，竞争的线程是进等待队列里等着，CPU什么时候叫你再执行。

执行时间段（加锁代码），线程数少，用自旋

执行时间长，线程数多，用系统锁

### 详细回答：

JVM对synchronized底层实现没有要求，下边是Hotspot的实现

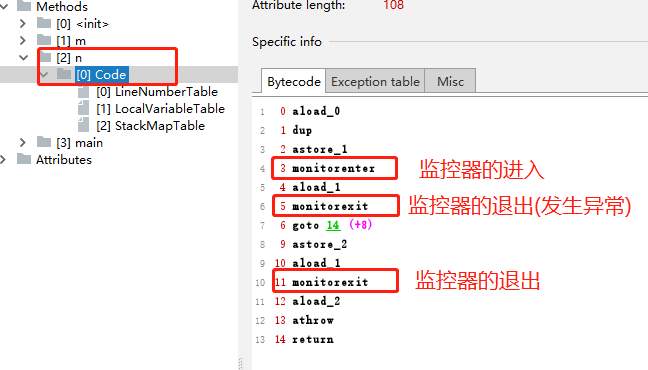
|  |
| --- |
| **package** com.mashibing.jvm.c3\_jmm; **public class** TestSync {  *//同步方法，字节码只是加了个ACC\_SYNCHRONIZED* **synchronized void** m(){  }  **void** n(){  *//同步代码块* **synchronized** (**this**){  }  }  **public static void** main(String[] args) {  *//main执行一遍确保编译* } } |

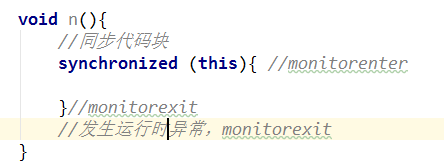
* **字节码层面：**

同步方法： 直接加了个 Access flag：ACC\_SYNCHRONIZED 修饰符

同步代码块： monitorenter指令 、monitorexit指令，告诉虚拟机，这里需要加锁处理

1. monitorenter：监控器的进入指令,
2. monitorexit：监控器的退出指令1，发生异常自动退出
3. monitorexit：监控器的退出指令2





* **JVM层面**
  1. C C++调用了操作系统提供的同步机制（依赖于硬件，依赖于CPU）
* **硬件层面**

X86: lock指令====> lock cmpxchg XXX

说明：

cmpxchg ：对内存某个区域修改

lock是用来锁定其他指令

lock cmpxchg XXX : 当修改某块内存的时候，是锁定这块内存，只有这条指定能改，其他指令不能改。

号外：

|  |
| --- |
| 普通对象（跟数组有区别）内存布局：    **Markword，一共8个字节， 64位：** |

## 为什么加锁的对象一般都加final？ √

锁定某个对象O，如果o的属性发生了改变，不影响锁的使用，

但是如果o变成了另外一个对象，则锁定的对象发生改变。

应该避免将锁定对象的引用变成另外的对象。

原因：锁是在对象头上两位作为代表的，本来多个线程，大家都去访问这个对的头上的这两位了，结果你突然把这把锁变成了别的对象，去访问别的对象头的两位了，这俩之间就没有任何关系了。因此，以对象作为锁的时候，不让他发生改变，加final

## 淘宝面试题：线程之间的通信 （synchronized +wait+notify牢记\*\*）√

实现一个容器，提供两个方法，add、size，写两个线程：

线程1，添加10个元素到容器中

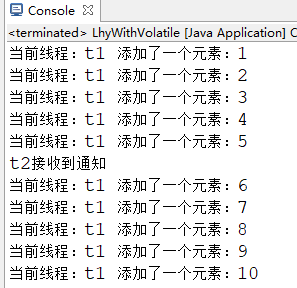
线程2，实时监控元素个数，当个数到5个时，线程2给出提示并结束

synchronized +wait+notify实现 ，注意：

Wait + notify +synchronized实现线程之间的通信，wait方法释放锁，notify方法不释放锁。

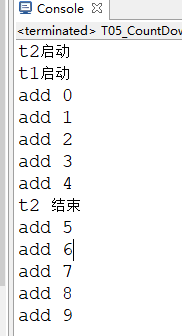
|  |
| --- |
| package com.mashibing.juc.c\_020\_01\_Interview;  import java.util.ArrayList;  import java.util.List;  import java.util.concurrent.TimeUnit;  /\*\*  \* 曾经的面试题：（淘宝？）  \* 实现一个容器，提供两个方法，add，size  \* 写两个线程，线程1添加10个元素到容器中，线程2实现监控元素的个数，当个数到5个时，线程2给出提示并结束  \*  \* 给lists添加volatile之后，t2能够接到通知，但是，t2线程的死循环很浪费cpu，如果不用死循环，该怎么做呢？  \*  \* 这里使用wait和notify做到，wait会释放锁，而notify不会释放锁  \* 需要注意的是，运用这种方法，必须要保证t2先执行，也就是首先让t2监听才可以  \*  \* 阅读下面的程序，并分析输出结果  \* 可以读到输出结果并不是size=5时t2退出，而是t1结束时t2才接收到通知而退出  \* 想想这是为什么？  \*  \* notify之后，t1必须释放锁，t2退出后，也必须notify，通知t1继续执行  \* 整个通信过程比较繁琐  \* \*/  public class LhyWithVolatile {  //添加volatile，使t2能够得到通知  private volatile List<Object> list = new ArrayList<>();    public void add(Object o){  list.add(o);  }  public int size(){  return list.size();  }    public static void main(String[] args) {    LhyWithVolatile myContiner = new LhyWithVolatile();    final Object lock = new Object();    Thread t2 = new Thread(()->{  synchronized (lock) {  if(myContiner.size() != 5){  try {  lock.wait();  } catch (Exception e) {  e.printStackTrace();  }  }  System.out.println("t2接收到通知");  //通知t1继续执行  lock.notify();  }    },"t2");    Thread t1 = new Thread(()->{  synchronized (lock) {  for(int i=1;i<=10;i++){  myContiner.add(i);  System.out.println("当前线程："+ Thread.currentThread().getName()+" 添加了一个元素："+i);  if(i == 5){  lock.notify();//notify不释放锁    try {  lock.wait();//t1等待，释放锁，让t2得以执行  } catch (Exception e) {  e.printStackTrace();  }  }  try {  TimeUnit.SECONDS.sleep(1);  } catch (Exception e) {  e.printStackTrace();  }  }  }    },"t1");  //注意：先启动t2,再启动t1  t2.start();  t1.start();      }  } |

结果：



还可以用CountDownLatch门闩解决：

|  |
| --- |
| /\*\*  \* 曾经的面试题：（淘宝？）  \* 实现一个容器，提供两个方法，add，size  \* 写两个线程，线程1添加10个元素到容器中，线程2实现监控元素的个数，当个数到5个时，线程2给出提示并结束  \*  \* 给lists添加volatile之后，t2能够接到通知，但是，t2线程的死循环很浪费cpu，如果不用死循环，该怎么做呢？  \*  \* 这里使用wait和notify做到，wait会释放锁，而notify不会释放锁  \* 需要注意的是，运用这种方法，必须要保证t2先执行，也就是首先让t2监听才可以  \*  \* 阅读下面的程序，并分析输出结果  \* 可以读到输出结果并不是size=5时t2退出，而是t1结束时t2才接收到通知而退出  \* 想想这是为什么？  \*  \* notify之后，t1必须释放锁，t2退出后，也必须notify，通知t1继续执行  \* 整个通信过程比较繁琐  \*  \* 使用Latch（门闩）替代wait notify来进行通知  \* 好处是通信方式简单，同时也可以指定等待时间  \* 使用await和countdown方法替代wait和notify  \* CountDownLatch不涉及锁定，当count的值为零时当前线程继续运行  \* 当不涉及同步，只是涉及线程通信的时候，用synchronized + wait/notify就显得太重了  \* 这时应该考虑countdownlatch/cyclicbarrier/semaphore  \* @author mashibing  \*/  package com.mashibing.juc.c\_020\_01\_Interview;  import java.util.ArrayList;  import java.util.List;  import java.util.concurrent.CountDownLatch;  import java.util.concurrent.TimeUnit;  public class T05\_CountDownLatch {  // 添加volatile，使t2能够得到通知  volatile List lists = new ArrayList();  public void add(Object o) {  lists.add(o);  }  public int size() {  return lists.size();  }  public static void main(String[] args) {  T05\_CountDownLatch c = new T05\_CountDownLatch();  //门闩，计数器为1  CountDownLatch latch = new CountDownLatch(1);  new Thread(() -> {  System.out.println("t2启动");  if (c.size() != 5) {  try {  latch.await();//启动就等着  } catch (InterruptedException e) {  e.printStackTrace();  }  }  System.out.println("t2 结束");  }, "t2").start();  //保证t2运行  try {  TimeUnit.SECONDS.sleep(1);  } catch (InterruptedException e1) {  e1.printStackTrace();  }  new Thread(() -> {  System.out.println("t1启动");  for (int i = 0; i < 10; i++) {  c.add(new Object());  System.out.println("add " + i);  if (c.size() == 5) {  // 打开门闩，让t2得以执行  latch.countDown();  }  try {  TimeUnit.SECONDS.sleep(1);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }, "t1").start();  }  } |



# ========= Volatile========

|  |
| --- |
|  |

## volatile基本概念 √

volatile 关键字，使一个变量在多个线程间可见

A B线程都用到一个变量，java默认是A线程中保留一份copy，这样如果B线程修改了该变量，则A线程未必知道

使用volatile关键字，会让所有线程都会读到变量的修改值

使用volatile，将会强制所有线程都去堆内存中读取running的值

* 保证线程可见性，
* 禁止指令重重拍序
* 并不保证原子性volatile不能替代synchronized

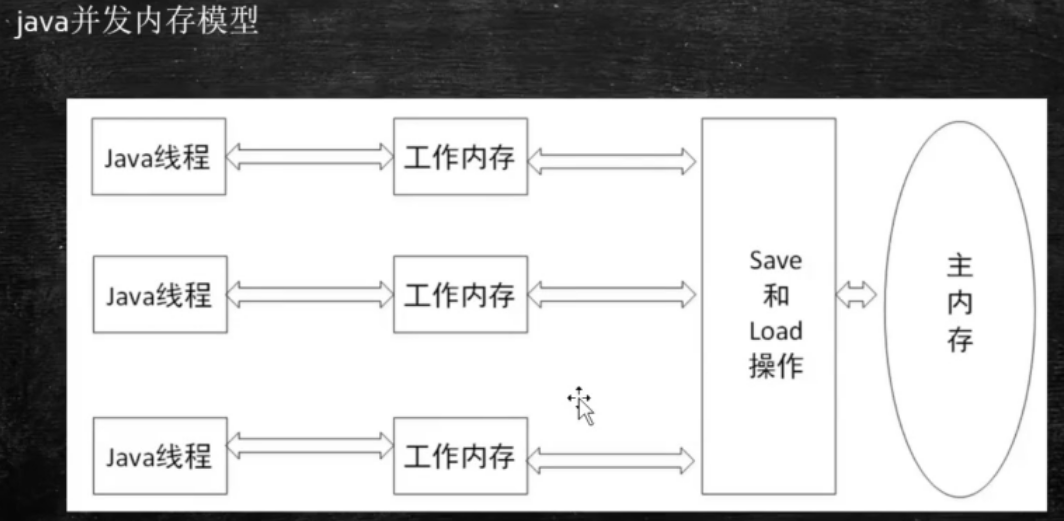
**例子程序：**

在下面的代码中，running是存在于堆内存的t对象中

当线程t1开始运行的时候，会把running值从内存中读到t1线程的工作区，在运行过程中直接使用这个copy，并不会每次都去读取堆内存，这样，当主线程修改running的值之后，t1线程感知不到，所以不会停止运行

使用volatile，将会强制所有线程都去堆内存中读取running的值

|  |
| --- |
| **public** **class** T01\_HelloVolatile {  /\*volatile\*/ **boolean** running = **true**; //对比一下有无volatile的情况下，整个程序运行结果的区别  **void** m() {  System.***out***.println("m start");  **while**(running) {  }  System.***out***.println("m end!");  }  **public** **static** **void** main(String[] args) {  T01\_HelloVolatile t = **new** T01\_HelloVolatile();    **new** Thread(t::m, "t1").start();  **try** {  TimeUnit.***SECONDS***.sleep(1);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  t.running = **false**;  }  }  ///////输出////////  不加volatile，只输出  m start  加volatile，输出  m start  m end! |



## Volatile的作用\*\*\*\* 五颗星 必须记着 √

### 保证线程可见性

#### 线程可见概念：

堆内存是所有线程共享的内存，除了共享内存之外，每个线程都有自己的专属区域，有自己的工作内存。

如果在共享内存存一个值，当某个线程去访问这个值的时候，会将这个值Copy一份到自己的工作空间里，然后对这个值的任何改变，首先在自己的空间里进行改变，什么时候写回去？是改完之后马上写回主内存。什么时候去检查有没有新值？不好说，别的线程改了你也不知道。

在这个线程里面发生的改变，并没有及时的反应到另外一个线程里面，这就是线程之间的不可见。

对这个值加了volatile后，就能够保证一个线程的改变，另外一个线程马上就能看到。

本质上是用来MESI ，缓存一致性协议

|  |
| --- |
| 现代CPU的数据一致性实现 = 缓存锁（MESI...）+总线锁  在MESI协议中，每个Cache line有4个状态，可用2个bit表示，它们分别是：    M(Modified)和E(Exclusive)状态的Cache line，数据是独有的，不同点在于M状态的数据是dirty的(和内存的不一致)，E状态的数据是clean的(和内存的一致)。 (Shared)状态的Cache line，数据和其他Core的Cache共享。只有clean的数据才能被多个Cache共享。 I(Invalid)表示这个Cache line无效。 |

### 禁止指令重排序（CPU）

Cpu执行指令本来是一条条顺序执行的，CPU为了提高指令执行效率，它会把指令并发执行，会打乱多条指令的执行顺序。【（比如去内存读数据（慢100倍）），去同时执行另一条指令（前提是两条指令没有依赖关系）】

那么就要求编译器吧源码编译完的指令进行一个指令的重新排序。

【volatile防止指令重拍序，就是防止这里的指令重排】

DCL（Double Check Lock）单例

内存屏障

Volatile保证线程的可见性，并不能替代synchronized，保证不了原子性，解决办法：加synchronized

## Volatile的实现细节 √

Volatile保证有序性，volatile就是一个内存屏障，在不同的CPU也是不一样的。

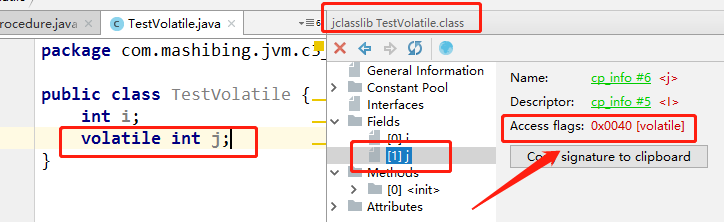
volatile在不同的层级有不同的实现，JVM、字节码、操作系统OS硬件HD

测试源码：

|  |
| --- |
| **public class** TestVolatile {  **int i**;  **volatile int j**; } |

### 字节码层面

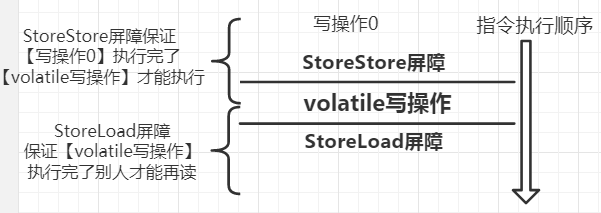
用diea 插件看Class文件，成员变量j，只是有一个volatile 的Access flags描述符：



### JVM层面 （面试题）

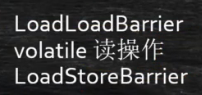
Volatile内存区的读写，都加屏障。

##### Volatile写操作



##### Volatile读操作

Volatile读操作前加 LoadLoad 屏障：保证前面的读操作先执行完，后面加LoadStore屏障：保证volatile读操作执行完，后续才能写。

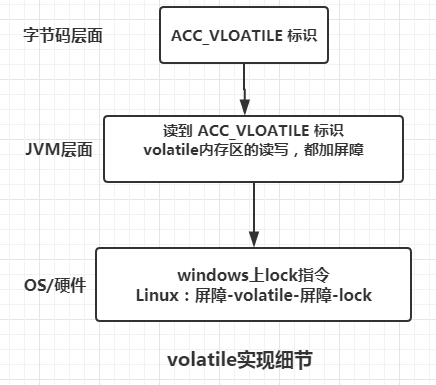


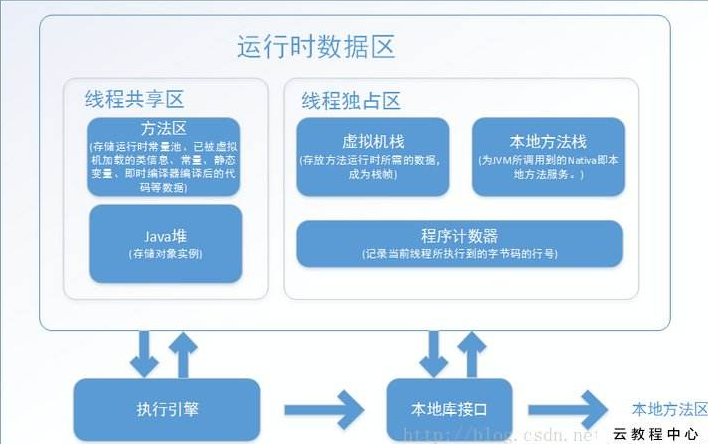
|  |
| --- |
| 插曲  现在很多人有一个变成趋势，不用volatile ，直接用synchornized，synchornized经过优化后效率也很高，并不比volatile差多少，所以现在volatile变成面试中存在。 |

### 操作系统及硬件层面

Hsdis -- Hotspot Dis Assembler 去观察汇编码（Hotspot的反汇编，虚拟机的反汇编，将虚拟机编译好的字节码，再进行反汇编。虚拟机编译好的字节码，在CPU级别到底是用怎么的指令来完成的）

不同的操作系统的实现可能是不同的，Windows上使用lock指令实现的，lock指令xxx执行xxx指令的时候，保证对内存区域加锁，并没有翻译成为屏障





## 双重检查单例，要不要加volatile？√

要，这个很难实验出出错的情况，JD淘宝天猫那种高并发场景才会出问题。

不加volatile问题就出在指令重排序上。

第一个线程*INSTANCE* = **new** Mgr06(); 经过编译器编译之后的指令分成三步：

1. 给指令申请内存
2. 给成员变量初始化
3. 把这块内存的内容赋值给INSTANCE

既然有这个值了，你在另外一个线程里头，上来先去检查，发现这个值已经有了，就不会进入锁的那部分代码。（线程一半初始化，线程二直接用了）

加了volatile后，指令重排序就不允许存在了，所以在这个时候一定是保证你初始化完了之后才会赋值给你这个变量。

|  |
| --- |
| **public** **class** Mgr06 {    **private** **static** **volatile** Mgr06 *INSTANCE* ;  **private** Mgr06(){}  **public** **static** Mgr06 getInstance(){  **if**(*INSTANCE* == **null**){  **synchronized** (Mgr06.**class**) {//类锁  **if**(*INSTANCE* == **null**){  *INSTANCE* = **new** Mgr06();  }  }  }  **return** *INSTANCE*;  }  } |
| 号外：   1. 给指令申请内存 2. 给成员变量初始化 3. 把这块内存的内容赋值给INSTANCE   这三个步骤顺序有严格规定么？在JVM里规定了八种原则，happen-before原则，除此之外的其他指令都可以重排序，保证原子性只是保证这些操作必须都完成了之后，其他才能访问，但是保证了原子性和保证重排序是两回事。 |

## Volatile 为什么不支持原子性？举个例子 √

|  |
| --- |
| /\*\*  \* volatile并不能保证多个线程共同修改running变量时所带来的不一致问题，  \* 也就是说volatile不能替代synchronized  \* 运行下面的程序，并分析结果  \* **@author** mashibing  \*/  **package** com.mashibing.juc.c\_012\_Volatile;  **import** java.util.ArrayList;  **import** java.util.List;  **public** **class** T04\_VolatileNotSync {  **volatile** **int** count = 0;  **synchronized** **void** m() {  **for**(**int** i=0; i<10000; i++) count++;  }    **public** **static** **void** main(String[] args) {  //初始化 count=0  T04\_VolatileNotSync t = **new** T04\_VolatileNotSync();    //10个线程装list  List<Thread> threads = **new** ArrayList<Thread>();  **for**(**int** i=0; i<10; i++) {  threads.add(**new** Thread(t::m, "thread-"+i));  }  //启动10个线程，调了了m方法对count进行++，每个线程加到10000  threads.forEach((o)->o.start());    threads.forEach((o)->{  **try** {  o.join();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  });    System.***out***.println(t.count);    }    } |

# ==========CAS与Atomic类============

## CAS无锁的概念、乐观锁和悲观锁 √

## 解释一下自旋 （CAS） /乐观锁 /无锁优化 √

Cas，compare and set 比较且设定。CAS号称是无锁优化，或者叫自旋。 （厕所蹲马桶例子）

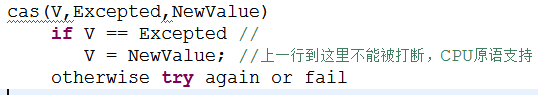
比较并且设定：我原来想改变某一个值0，我想把它变成1 ，但是其中我想做到线程安全，就只能加synchronized，不然线程就不安全。现在我可以用另外一种操作来替代这把锁。就是cas操作，你可以把它想象成一个方法，这个方法有三个参数：

cas（V，Excepted，NewValue）

V：要改的值

Excepted：期望当前的这个值会是几

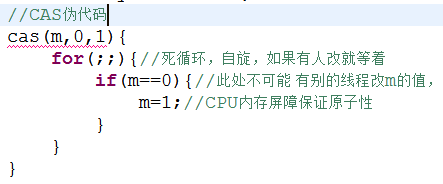
NewValue：要设定的新值



比如我原来有个值是3，我这个线程改这个值的时候一定期望你现在是3，如果你在我改的过程中变成了4，那你跟我期望值就对不上了，说明有另外一个线程改了这个值了，那我这个cas就重新再试一下，再试的时候我希望你这个值是4，在修改的时候期望值是4，没有其他线程修改这个值，那好，我给你改成5。这就是cas操作。

Excepted如果对得上期望值，NewValue才会去对其修改，进行新值设定的时候，这个过程中如果来了一个线程把你的值改变了怎么办，我就可以再试一遍，或者失败，这个就是cas操作。

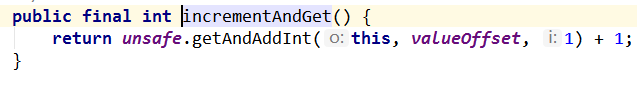
当你判断的时候，发现是我期望的值，还没有进行新值设定的时候发生了改变怎么办，cas是cpu的原语支持，cas操作是cpu指令级别上的支持，中间不能被打断。

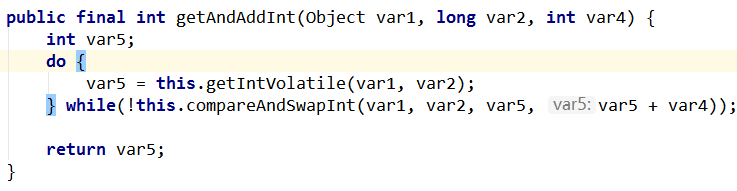


例子：

JDK里以Atomic开头的都是用CAS这种操作来保证现场安全的。

比如AtomicInteger类的getAndIncrement方法，调用了Unsafe类的getAndAddInt方法。这是一个CAS操作。



getAndAddInt

|  |
| --- |
| **扩展：Unsafe 类**  **Unsafe类 = C C++的指针**   1. 直接操作内存   allocateMemory putXXX freeMemory pageSize  看源码方法都是native的，直接操作c++   1. 直接生成类实例   **public native** Object allocateInstance(Class<?> var1)   1. 直接操作类或实例变量   **public native long** objectFieldOffset(Field var1);  **public native int** getInt(Object var1, **long** var2);  **public native** Object getObject(Object var1, **long** var2);   1. CAS相关操作   compareAndSwapInt Object Long   1. -> malloc free   C++ --> new delete  CAS不需要加锁怎么做到的呢？原因就是用了Unsafe这个类，这个类除了用反射使用之外，其他不能直接使用，不能使用的原因和ClassLoader有关系。所有Atomic操作内部都是CompareAndSet这样的操作。 |

## CAS机制会出现什么问题； √

ABA问题

## 什么是ABA问题，出现ABA问题JDK是如何解决的 \*\*\*3星 √

使用CAS操作会有ABA问题。

ABA问题是这样的：假如说你有一个值，我拿到这个值是1，想把它变成2，我拿到1用cas操作，期望值是1，准备变成2，在这个过程中，没有一个线程改过我肯定是可以更改的，但是如果有一个线程先把这个1变成2后来又变回1，中间值改过，他不会影响我这个cas下面的操作，这就是ABA问题。

如果是int等基础类型，最终值是你期望的，也没关系。可以不用管，如果你确实想管这个问题可以加版本号，做任何一个值的修改，修改完之后加一，后面检查的时候连带版本号一起检查。

结论：

加版本号解决，CAS操作时加Version。

A 1.0

B 2.0

A 3.0

JDK的解决方案：AtomicStampedReference atomic类里带有版本号的类

如果是基础类型：无所谓，不影响结果值。

如果是引用类型：就像你的女朋友和你分手之后又复合，中间经历了别的男人。

## （自命题）分段锁 是什么？跟Atomic原子类有什么区别

多线程对一个数进行递增，目前学了两种方法：

第一种是long类型的数，递增的时候加锁

第二种，使用AtomicLong

还有第三种：LongAdder

下面的小程序，用三种方式，模拟1000个线程对一个long进行递增操作，每个线程对count进行10w递增。分别对其三种方式测试其性能

|  |
| --- |
| **import** java.util.concurrent.TimeUnit;  **import** java.util.concurrent.atomic.AtomicLong;  **import** java.util.concurrent.atomic.LongAdder;  **public** **class** T02\_AtomicVsSyncVsLongAdder {  //测试synchronized  **static** **long** *count2* = 0L;  //测试Atomic  **static** AtomicLong *count1* = **new** AtomicLong(0L);  //测试LongAdder  **static** LongAdder *count3* = **new** LongAdder();  **public** **static** **void** main(String[] args) **throws** Exception {  //1000个线程  Thread[] threads = **new** Thread[1000];    **for**(**int** i=0; i<threads.length; i++) {  threads[i] =  **new** Thread(()-> {//每个线程将count1递增10w  **for**(**int** k=0; k<100000; k++) *count1*.incrementAndGet();  });  }  **long** start = System.*currentTimeMillis*();  //启动线程  **for**(Thread t : threads ) t.start();  //等待线程执行完  **for** (Thread t : threads) t.join();  **long** end = System.*currentTimeMillis*();  //TimeUnit.SECONDS.sleep(10);  System.***out***.println("Atomic: " + *count1*.get() + " time " + (end-start));  //-----------------------------------------------------------100000000  Object lock = **new** Object();    //1000线程，每个线程对count2 地递增10w  **for**(**int** i=0; i<threads.length; i++) {  threads[i] =  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  **for** (**int** k = 0; k < 100000; k++)  **synchronized** (lock) {  *count2*++;  }  }  });  }  start = System.*currentTimeMillis*();  **for**(Thread t : threads ) t.start();  **for** (Thread t : threads) t.join();  end = System.*currentTimeMillis*();  System.***out***.println("Sync: " + *count2* + " time " + (end-start));  //LongAdder-------------------------  **for**(**int** i=0; i<threads.length; i++) {  threads[i] =  **new** Thread(()-> {  **for**(**int** k=0; k<100000; k++) *count3*.increment();  });  }  start = System.*currentTimeMillis*();  **for**(Thread t : threads ) t.start();  **for** (Thread t : threads) t.join();  end = System.*currentTimeMillis*();  //TimeUnit.SECONDS.sleep(10);  System.***out***.println("LongAdder: " + *count1*.longValue() + " time " + (end-start));  }  **static** **void** microSleep(**int** m) {  **try** {  TimeUnit.***MICROSECONDS***.sleep(m);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  } |

运行结果：

Atomic: 100000000 time 2214

Sync: 100000000 time 3810

LongAdder: 100000000 time 1959

实验结果说明，LongAdder效率是最高的。但是把线程数变小了LongAdder未必有优势，循环数量减少了LongAdder也未必有优势。所以实际当中用哪种需要考虑并发有多高。

**Atomic为什么要比Synchronized快？**

因为Atomic不加锁，Synchronized要加锁，有可能去操作系统申请重量级锁，所以synchronized效率偏低。

**LongAdder为什么比Atomic效率高？**

**老马：因为LongAdder内部做了一个分段锁，会把一个值放到一个数组里，比如说数组长度是4，最开始是0,1000个线程，250个线程锁在第一个数组元素里，以此类推，每一个都往上地址算出来结果再加到一起。**

网上：LongAdder在AtomicLong的基础上将单点的更新压力分散到各个节点，在低并发的时候通过对base的直接更新可以很好的保障和AtomicLong的性能基本保持一致，而在高并发的时候通过分散提高了性能。

# =======基于CAS新类型的锁IMG_256======

## 讲下ReentrantLock可重入锁？什么是可重入锁？为什么要设计可重入锁？

### 可重入锁:

一个对象里，一个同步方法可以调用另外一个同步方法，一个线程已经拥有某个对象的锁，再次申请的时候仍然会得到该对象的锁（注意是一个线程，两个线程就没有重入的事）

ReentrantLock 可重入锁，synchronized本身就是可重入锁的一种。

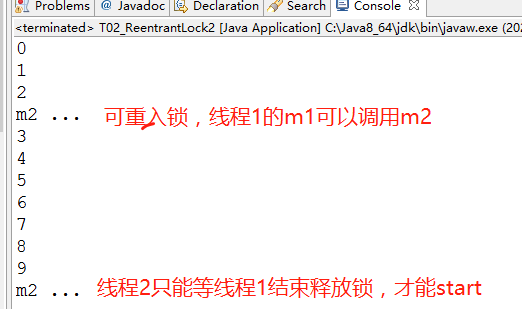
### 例子程序：

使用ReentrantLock 替代synchronized

原来写synchronized的地方，换成lock.lock(); 加完锁之后需要注意的是记得lock.unlock();解锁。

Synchronized是自动解锁的，大括号执行完了就结束了。Lock不行，lock必须手动解锁。写在try finally 里保证一定解锁，否则上锁之后中间执行的过程中有问题了，死在那了，别人就永远也拿不到这把锁了。

|  |
| --- |
| /\*\*  \* reentrantlock用于替代synchronized  \* 由于m1锁定this,只有m1执行完毕的时候,m2才能执行  \* 这里是复习synchronized最原始的语义  \*  \* 使用reentrantlock可以完成同样的功能  \* 需要注意的是，必须要必须要必须要手动释放锁（重要的事情说三遍）  \* 使用syn锁定的话如果遇到异常，jvm会自动释放锁，但是lock必须手动释放锁，因此经常在finally中进行锁的释放  \* **@author** mashibing  \*/  **package** com.mashibing.juc.c\_020;  **import** java.util.concurrent.TimeUnit;  **import** java.util.concurrent.locks.Lock;  **import** java.util.concurrent.locks.ReentrantLock;  **public** **class** T02\_ReentrantLock2 {  Lock lock = **new** ReentrantLock();  **void** m1() {  **try** {  lock.lock(); //synchronized(this)  **for** (**int** i = 0; i < 10; i++) {  TimeUnit.***SECONDS***.sleep(1);  System.***out***.println(i);  }  } **catch** (InterruptedException e) {  e.printStackTrace();  } **finally** {  lock.unlock();  }  }  **void** m2() {  **try** {  lock.lock();  System.***out***.println("m2 ...");  } **finally** {  lock.unlock();  }  }  **public** **static** **void** main(String[] args) {  T02\_ReentrantLock2 rl = **new** T02\_ReentrantLock2();  **new** Thread(()->{  rl.m1();  }).start();    **try** {  TimeUnit.***SECONDS***.sleep(1);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **new** Thread(rl::m2).start();  }  } |



### 为什么要设计可重入锁？

如果是想问ReentrantLock ，有了synchronized还为啥要设计ReentrantLock ？看下一问回答。 next

|  |
| --- |
| (这个问题我觉得问的有问题, 如果是想问为什么要设计可重入锁？Synchronized锁定一个对象的时候，锁定的是对象头上的两位，标识是否加锁。一个线程锁定这个对象，这个线程又过来申请这把锁，发现当前持有这把锁的线程就是我自己，那好就继续执行就行。可以说父子类的例子：子类重写父类的synchronized方法，子类调用super.m();的时候，必须是可重入的，否则就会出问题（调父类是同一把锁），所谓的重入锁就是你拿到这把锁之后不停的加锁加锁，加好几道，但锁定的还是同一个对象，去一道就减个一，就是这么个概念。  /\*\*  \* 一个同步方法可以调用另外一个同步方法，一个线程已经拥有某个对象的锁，再次申请的时候仍然会得到该对象的锁.  \* 也就是说synchronized获得的锁是可重入的  \* 这里是继承中有可能发生的情形，子类调用父类的同步方法  \* **@author** mashibing  \*/  **package** com.mashibing.juc.c\_010;  **import** java.util.concurrent.TimeUnit;  **public** **class** T {  **synchronized** **void** m() {  System.***out***.println("m start");  **try** {  TimeUnit.***SECONDS***.sleep(1);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println("m end");  }    **public** **static** **void** main(String[] args) {  **new** TT().m();  }    }  **class** TT **extends** T {  @Override  **synchronized** **void** m() {  System.***out***.println("child m start");  **super**.m();  System.***out***.println("child m end");  }  } |

## Synchronized 和 ReecentrantLock 有什么区别？这两个有没有深入了解源码 √

区别：

Synchronized 系统自动加锁，自动解锁，不可以出现多个不同的等待队列，默认进行无锁、偏向锁、轻量级锁（自旋锁），重量级锁（系统锁）4种锁状态升级

ReecentrantLock：需要手动加锁、解锁，可以出现多个不同的等待队列。

------------------------------------------------------------------------------------------------

RecentrantLock和Synchronized 既然差不多的话，那要他何用？

ReentrantLock有一些功能还是比synchronized强大的：

ReentrantLock更灵活。

* tryLock尝试锁定
* 可以被打断的加锁
* 可以指定为公平锁

### tryLock尝试锁定

#### 概念：

可以使用tryLock 进行尝试锁定，不管锁定与否，方法都将继续执行，synchronized如果搞不定的话他就阻塞了，但是用ReentrantLock你自己就可以决定你要不要wait。

#### 例子程序：

线程1 调用T03\_ReentrantLock3对象的m1方法，进行lock.lock进行锁定当前对象，for循环3次，每次1秒，所以锁定了3秒。

线程2调用T03\_ReentrantLock3对象的m2方法，调用lock.tryLock(5, TimeUnit.***SECONDS***);尝试在5秒内获得锁，进行后续操作（拿到拿不到锁，后续是否wait等待，自己决定）

|  |
| --- |
| /\*\*  \* reentrantlock用于替代synchronized  \* 由于m1锁定this,只有m1执行完毕的时候,m2才能执行  \* 这里是复习synchronized最原始的语义  \*  \* 使用reentrantlock可以完成同样的功能  \* 需要注意的是，必须要必须要必须要手动释放锁（重要的事情说三遍）  \* 使用syn锁定的话如果遇到异常，jvm会自动释放锁，但是lock必须手动释放锁，因此经常在finally中进行锁的释放  \*  \* 使用reentrantlock可以进行“尝试锁定”tryLock，这样无法锁定，或者在指定时间内无法锁定，线程可以决定是否继续等待  \* **@author** mashibing  \*/  **package** com.mashibing.juc.c\_020;  **import** java.util.concurrent.TimeUnit;  **import** java.util.concurrent.locks.Lock;  **import** java.util.concurrent.locks.ReentrantLock;  **public** **class** T03\_ReentrantLock3 {  Lock lock = **new** ReentrantLock();  **void** m1() {  **try** {  lock.lock();  **for** (**int** i = 0; i < 3; i++) {  TimeUnit.***SECONDS***.sleep(1);  System.***out***.println(i);  }  } **catch** (InterruptedException e) {  e.printStackTrace();  } **finally** {  lock.unlock();  }  }  /\*\*  \* 使用tryLock进行尝试锁定，不管锁定与否，方法都将继续执行  \* 可以根据tryLock的返回值来判定是否锁定  \* 也可以指定tryLock的时间，由于tryLock(time)抛出异常，所以要注意unclock的处理，必须放到finally中  \*/  **void** m2() {  /\*  boolean locked = lock.tryLock();  System.out.println("m2 ..." + locked);  if(locked) lock.unlock();  \*/    **boolean** locked = **false**;    **try** {  locked = lock.tryLock(5, TimeUnit.***SECONDS***);  System.***out***.println("m2 ..." + locked);  } **catch** (InterruptedException e) {  e.printStackTrace();  } **finally** {  **if**(locked) lock.unlock();  }    }  **public** **static** **void** main(String[] args) {  T03\_ReentrantLock3 rl = **new** T03\_ReentrantLock3();  **new** Thread(rl::m1).start();  **try** {  TimeUnit.***SECONDS***.sleep(1);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **new** Thread(rl::m2).start();  }  }  ///////输出//////////  线程1调用对象的m1方法，当第3秒输出2时，线程1释放锁。线程2获得锁，tryLock5秒为true，继续往下执行 |

### 可以被打断的加锁

#### 概念：

ReentrantLock可以用lock.lockInterruptibly();进行加锁，这个方法对interrupt()方法做出响应，可以被打断的加锁，如果以这种方式加锁的话，可以调用一个t2.interrupt(); 打断某个线程的等待。

#### 例子程序：

如果以这种方式加锁的话，我们可以调用一个t2.interrupt();打断线程2的等待。线程1上来之后加锁，枷锁之后开始睡，睡的没完没了的，被线程1拿到这把锁的话，线程2如果说再想拿到这把锁是不可能的，拿不到锁他就会在那等着，如果我们使用原来的lock.lock是打断不了他的，那么就可以用另外一种方式lock.lockInterruptibly();这个方法是可以被打断的，当你想要停止线程2就可以interrupt();

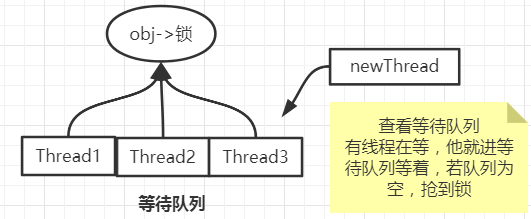
|  |
| --- |
| /\*\*  \* reentrantlock用于替代synchronized  \* 由于m1锁定this,只有m1执行完毕的时候,m2才能执行  \* 这里是复习synchronized最原始的语义  \*  \* 使用reentrantlock可以完成同样的功能  \* 需要注意的是，必须要必须要必须要手动释放锁（重要的事情说三遍）  \* 使用syn锁定的话如果遇到异常，jvm会自动释放锁，但是lock必须手动释放锁，因此经常在finally中进行锁的释放  \*  \* 使用reentrantlock可以进行“尝试锁定”tryLock，这样无法锁定，或者在指定时间内无法锁定，线程可以决定是否继续等待  \*  \* 使用ReentrantLock还可以调用lockInterruptibly方法，可以对线程interrupt方法做出响应，  \* 在一个线程等待锁的过程中，可以被打断  \*  \* **@author** mashibing  \*/  **package** com.mashibing.juc.c\_020;  **import** java.util.concurrent.TimeUnit;  **import** java.util.concurrent.locks.Lock;  **import** java.util.concurrent.locks.ReentrantLock;  **import** java.util.function.Function;  **public** **class** T04\_ReentrantLock4 {    **public** **static** **void** main(String[] args) {  Lock lock = **new** ReentrantLock();      Thread t1 = **new** Thread(()->{  **try** {  lock.lock();  System.***out***.println("t1 start");  TimeUnit.***SECONDS***.sleep(10);  System.***out***.println("t1 end");  } **catch** (InterruptedException e) {  System.***out***.println("interrupted!");  } **finally** {  lock.unlock();  }  });  t1.start();    Thread t2 = **new** Thread(()->{  **try** {  //lock.lock();  lock.lockInterruptibly(); //可以对interrupt()方法做出响应  System.***out***.println("t2 start");  TimeUnit.***SECONDS***.sleep(5);  System.***out***.println("t2 end");  } **catch** (InterruptedException e) {  System.***out***.println("interrupted!");  } **finally** {  lock.unlock();  }  });  t2.start();    **try** {  TimeUnit.***SECONDS***.sleep(1);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  t2.interrupt(); //打断线程2的等待    }  } |

### ReentrantLock可以指定为公平锁

#### 概念：

公平锁意思就是当我们new一个ReentrantLock的时候你可以传一个参数为true，表示公平锁。

公平锁的意思是谁等在前面就让谁先执行，而不是说谁后来了之后就马上让谁执行。如果这个锁不公平，来了一个线程上来就抢，他是有可能抢得到的，如果说这个锁是公平锁，这个线程上来会先检查队列里有没有原来等待着的，如果有的话，他就先进队列里等着别人先运行。



#### 例子程序

|  |
| --- |
| /\*\*  \* ReentrantLock还可以指定为公平锁  \* **@author** mashibing  \*/  **package** com.mashibing.juc.c\_020;  **import** java.util.concurrent.locks.ReentrantLock;  **public** **class** T05\_ReentrantLock5 **extends** Thread {    **private** **static** ReentrantLock *lock*=**new** ReentrantLock(**true**); //参数为true表示为公平锁，请对比输出结果  **public** **void** run() {  **for**(**int** i=0; i<100; i++) {  *lock*.lock();  **try**{  System.***out***.println(Thread.*currentThread*().getName()+"获得锁");  }**finally**{  *lock*.unlock();  }  }  }  **public** **static** **void** main(String[] args) {  T05\_ReentrantLock5 rl=**new** T05\_ReentrantLock5();  Thread th1=**new** Thread(rl);  Thread th2=**new** Thread(rl);  th1.start();  th2.start();  }  }  //////////运行结果/////////// |

### ReentrantLock VS Synchronized 总结：

ReentrantLock可以替代Synchronized是没问题的，他也是可重入的，可以锁定的。本身底层是CAS

TryLock：自己来控制，我锁不住该怎么办

lockInterruptibly ：中间你还可以被打断

ReentrantLock可以指定公平锁和非公平锁

现在除了synchronized之外，多数内部都是cas。

## 说说 CountDownLatch、CyclicBarrier 原理和区别 √

### CountdownLatch:

countDown：倒数，Latch，门闩。倒数的一个门闩。构造方法new CountDownLatch(2);指定count数值，调用countdown() 计数器减一，5，4，3，2，1,0数到了，这个门闩就开了。

countDownLatch这个类使一个线程等待其他线程各自执行完毕后再执行

是通过一个计数器来实现的，计数器的初始值是线程的数量。每当一个线程执行完毕后，计数器的值就-1，当计数器的值为0时，表示所有线程都执行完毕，然后在闭锁上等待的线程就可以恢复工作了。

用join也可以实现，但是join不好控制。Join必须是你线程执行完了才能控制。

### 源码

**构造器：**

public CountDownLatch(int count) { }; //参数count为计数值

**三个方法是最重要：**

* await() ;

//调用await()方法的线程会被挂起，它会等待直到count值为0才继续执行

* await(long timeout, TimeUnit unit)

//和await()类似，只不过等待一定的时间后count值还没变为0的话就会继续执行

* countDown()

//将count值减1

### 例子程序：

下面的例子说了，在主线程调用latch.await(); 插好门别动，等到计数器减到0，你就给我打开。

T1 t2线程执行完，就调一次latch.countDown();

|  |
| --- |
| package com.mashibing.juc.c\_020;  import java.util.concurrent.CountDownLatch;  import java.util.concurrent.TimeUnit;  /\*\*  \* 下面的例子说了，在主线程调用latch.await(); 插好门别动，等到计数器减到0，你就给我打开。  \* T1 t2线程执行完，就调一次latch.countDown();  \* @author EDZ  \*  \*/  public class MyCountdownLatch {    public static void main(String[] args){    CountDownLatch latch = new CountDownLatch(2);    //子线程1  new Thread(()->{  System.out.println("进入t1 ");  try {  TimeUnit.SECONDS.sleep(2);  } catch (Exception e) {  e.printStackTrace();  }  latch.countDown();  System.out.println("t1执行完，count:"+latch.getCount());  },"t1").start();    //子线程3  new Thread(()->{  System.out.println("进入t2 ");  try {  TimeUnit.SECONDS.sleep(3);  } catch (Exception e) {  e.printStackTrace();  }  latch.countDown();  System.out.println("t2执行完，count:"+latch.getCount());  },"t2").start();    System.out.println("主线程开始运行，等待其他线程执行完");  try {  latch.await();//等待  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println("其他线程执行完,主线程继续运行，count:"+latch.getCount());  }  } |

|  |
| --- |
| ///////运行结果///////// |

### CyclicBarrier

循环栅栏。一个羊圈，这里有一个栅栏门，门口就这么大地，要放羊出去，羊一只一只的过来，我告诉每个羊，过来你给我等着，等满了5只，我把栅栏打开，你们一块出去。出去之后栅栏又重新围起来，再来羊，满了5只，放开栅栏又继续。

CyclicBarrier就是这么个概念,每只羊就是一个线程，告诉羊等着，就是调用**barrier.await()方法。**

### 方法：

* **构造方法：**

**public** CyclicBarrier(**int** parties, Runnable barrierAction)

第一个参数是barrierAction执行之前要达到的线程数量，达到这个数量后，就执行第二个参数的动作，是个runnable对象。

* **barrier.await()**

使当前线程等待直到锁存器倒计时到零（其实内部是倒计时）

### 例子程序

|  |
| --- |
| **package** com.mashibing.juc.c\_020;  **import** java.util.concurrent.BrokenBarrierException;  **import** java.util.concurrent.CyclicBarrier;  **import** java.util.concurrent.TimeUnit;  **public** **class** T07\_TestCyclicBarrier {  **public** **static** **void** main(String[] args) {    //构造方法第二个参数不传，满了之后不做任何事情。  //CyclicBarrier barrier = new CyclicBarrier(20);  //CyclicBarrier barrier = new CyclicBarrier(20, () -> System.out.println("满人"));  //满20人 （线程），发一次车  CyclicBarrier barrier = **new** CyclicBarrier(20, **new** Runnable() {  @Override  **public** **void** run() {  System.***out***.println("满人，发车");  }  });  //100个线程，一共发5次车  **for**(**int** i=0; i<100; i++) {  **new** Thread(()->{  **try** {  barrier.await(); //使当前线程等待直到 到达设定的满员阈值  } **catch** (InterruptedException e) {  e.printStackTrace();  } **catch** (BrokenBarrierException e) {  e.printStackTrace();  }  }).start();  }  }  } |

## **CountDownLatch 和CyclicBarrier区别** 1.countDownLatch是一个计数器，线程完成一个记录一个，计数器递减，只能只用一次 2.CyclicBarrier的计数器更像一个阀门，需要所有线程都到达，然后继续执行，计数器递增，提供reset功能，可以多次使用

## ReadWriteLock √

ReadWriteLock管理一组锁，一个是只读的锁，一个是写锁。

**ReadWriteLock是读写锁**。读写锁的概念就是共享锁和排他锁。读锁就是共享锁，写锁就是排他锁。

一个获得了读锁的线程必须能看到前一个释放的写锁所更新的内容，读写锁之间为互斥。

ReetrantReadWriteLock实现了ReadWriteLock接口并添加了**可重入**的特性。

构造方方法可以指定true/false是否是公平锁，默认非公平（谁等在前面就让谁先执行，而不是说谁后来了之后就马上让谁执行）

ReentrantReadWriteLock又分出两把锁：

//读写锁

**static** ReadWriteLock *readWriteLock* = **new** ReentrantReadWriteLock();

//读锁

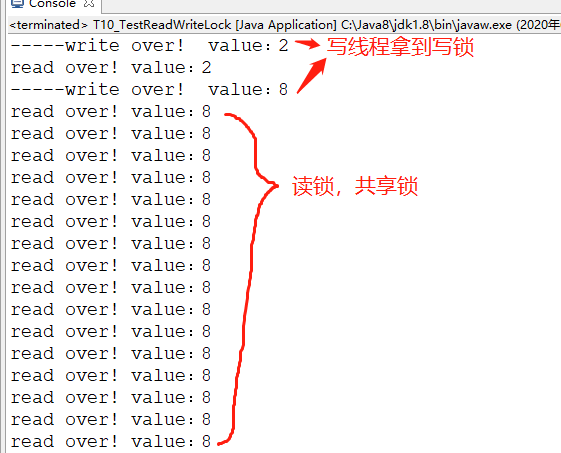
**static** Lock *readLock* = *readWriteLock*.readLock();

//写锁

**static** Lock *writeLock* = *readWriteLock*.writeLock();

例子程序：

|  |
| --- |
| **package** com.mashibing.juc.c\_020;  **import** java.util.Random;  **import** java.util.concurrent.locks.Lock;  **import** java.util.concurrent.locks.ReadWriteLock;  **import** java.util.concurrent.locks.ReentrantLock;  **import** java.util.concurrent.locks.ReentrantReadWriteLock;  **public** **class** T10\_TestReadWriteLock {      **static** Lock *lock* = **new** ReentrantLock(); //不是读写锁  **private** **static** **int** *value*;    //读写锁  **static** ReadWriteLock *readWriteLock* = **new** ReentrantReadWriteLock(**true**); //是否公平锁，默认false  //读锁  **static** Lock *readLock* = *readWriteLock*.readLock();  //写锁  **static** Lock *writeLock* = *readWriteLock*.writeLock();  **public** **static** **void** read(Lock lock) {  **try** {  lock.lock();  Thread.*sleep*(1000);  System.***out***.println("read over! value："+*value*);  //模拟读取操作  } **catch** (InterruptedException e) {  e.printStackTrace();  } **finally** {  lock.unlock();  }  }  **public** **static** **void** write(Lock lock, **int** v) {  **try** {  lock.lock();  Thread.*sleep*(1000);  *value* = v;  System.***out***.println("-----write over! value："+*value*);  //模拟写操作  } **catch** (InterruptedException e) {  e.printStackTrace();  } **finally** {  lock.unlock();  }  }  **public** **static** **void** main(String[] args) {    //2个写线程，写放前边让他可能先执行  **for**(**int** i=0; i<2; i++){  **new** Thread(()->{  *write*(*writeLock*, **new** Random().nextInt(10)); //writeLock  }).start();  }  //18个读线程  **for**(**int** i=0; i<18; i++){  **new** Thread(()->{  *read*(*readLock*); //readLock  }).start();  }  }  } |



写，排他锁，读，共享锁，读线程可以一起读。

## 说说 Semaphore 原理 √

### Semaphore 信号量，

Semaphore用于限制可以访问某些资源的线程数目,他维护了一个许可证集合，假如这里有N个资源，那就对应于N个许可证，同一时刻也只能有N个线程访问。一个线程获取许可证就调用acquire方法，用完了释放资源就调用release方法。

Semaphore s = **new** Semaphore(1); //初始化有1个许可证

**semaphore.acquire()** ，获得锁，阻塞方法， s.acquire();一下就变成0个许可，变成0后别人是acquire不到的，需要你线程结束后，**s.release();**许可证又还回去。又变成1

tryacquire（time，时间单位）尝试在时间内获取锁，若获取不到抛异常

### 例子程序：

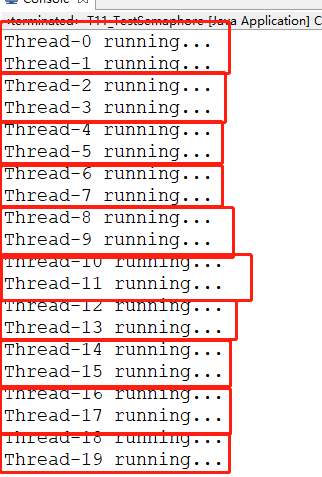
初始化2个信号量，数组初始化20个线程，每个线程里s.acquire();然后s.release();

，启动20个线程，观察输出

|  |
| --- |
| **package** com.mashibing.juc.c\_020;  **import** java.util.concurrent.Semaphore;  **import** java.util.concurrent.TimeUnit;  **public** **class** T11\_TestSemaphore {  **public** **static** **void** main(String[] args) {  Semaphore s = **new** Semaphore(2);    //信号量，只允许 3个线程同时访问 ，通过 Semaphore 限流。  //Semaphore s = new Semaphore(2, true);  //允许一个线程同时执行  //Semaphore s = new Semaphore(1);  Thread[] threads = **new** Thread[20];    **for**(**int** i=0;i<20;i++){    threads[i] = **new** Thread(()->{  **try** {  s.acquire();  System.***out***.println(Thread.*currentThread*().getName() + " running...");  TimeUnit.***SECONDS***.sleep(1);  s.release();  } **catch** (Exception e) {  e.printStackTrace();  }  });  }  **for**(Thread t : threads){  t.start();  }  }  } |

执行结果：

2个线程输出一次，限流了，最多一次只能有2个线程一起执行。



## 说说 Exchanger 原理 √

Exchanger：交换器，两个线程之间交换数据用的。只能是两个线程

线程间的通信方式有很多种，这只是其中的一种。

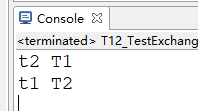
注意：

第一个线程执行到*exchanger*.exchange(s);的时候，会阻塞，等着第二个线程也执行*exchanger*.exchange(s);

例子程序

|  |
| --- |
| **package** com.mashibing.juc.c\_020;  **import** java.util.concurrent.Exchanger;  **import** java.util.concurrent.TimeUnit;  **public** **class** T12\_TestExchanger {  **static** Exchanger<String> *exchanger* = **new** Exchanger<>();  **public** **static** **void** main(String[] args) {  **new** Thread(()->{  System.***out***.println("进入t1,执行exchange");  String s = "T1";  **try** {  /\*\*  \* 把s传给exchanger，然后s接收exchange后的值  \* 在这里s已经被交换成了【t2】传过来的值  \*/  s = *exchanger*.exchange(s);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName() + " " + s);  }, "t1").start();  **new** Thread(()->{  System.***out***.println("进入t2,先睡2秒，再执行exchange");  String s = "T2";  **try** {  /\*\*  \* 把s传给exchanger，然后s接收exchange后的值  \* 在这里s已经被交换成了【t1】传过来的值  \*/  TimeUnit.***SECONDS***.sleep(2); //t2睡的时候，t1会阻塞。  s = *exchanger*.exchange(s);//偷梁换柱  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName() + " " + s);  }, "t2").start();  }  } |

结果：



## LockSupport √

### 概念

以前要阻塞和唤醒某一个具体的线程有很多限制，比如，用wait和notify：

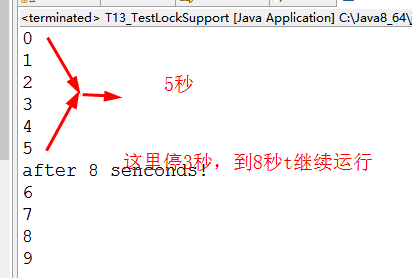
1. wait()方法需要释放锁，Notify(); 必须在synchronized中使用
2. Synchronized wait() ； notify()； 对象必须一致，一个synchronized代码块中只能有一个线程调用wait()和notify

种种限制表现出很多不足，所以LockSupport的好处就体现了：

在JDK1.6中的java.util.concurrent的子包locks中引入了LockSupport这个API，LockSupport是一个比较底层的工具类，用来创建锁和其他同步工具类的基本线程阻塞原语。Java锁和同步容器框架的核心：AQS：AbstractQueueSynchronizer ，就是通过调用LockSupport.park(); 和LockSupport.unpark(); 方法，来实现线程的阻塞和唤醒的。

### 例子程序

|  |
| --- |
| package com.mashibing.juc.c\_020;  import java.util.concurrent.TimeUnit;  import java.util.concurrent.locks.LockSupport;  public class T13\_TestLockSupport {  public static void main(String[] args) {  Thread t = new Thread(()->{  for (int i = 0; i < 10; i++) {  System.out.println(i);  if(i == 5) {  LockSupport.park(); //t线程--停车  }  try {  TimeUnit.SECONDS.sleep(1);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  });  t.start();  //LockSupport.unpark(t);  try {  TimeUnit.SECONDS.sleep(8);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println("after 8 senconds!");  LockSupport.unpark(t);//t线程--开车，唤醒t线程  }  } |



1. LockSupport 不需要synchronized加锁就可以实现线程的阻塞和唤醒
2. LockSupport.unpark(); 可以优先于LockSupport.park(); 执行，并且线程不会阻塞。
3. 如果一个线程处于等待状态，连续调用了两次park();方法，就会使该线程永远无法被唤醒。

### 原理

LockSupport.*park*(); LockSupport.*unpark*(t);实现原理

*park*();*unpark*(t);方法实现是由Unsafe类提供的，而Unsafe类是由C和C++语言完成的，方法都是native的，他的原理主要是通过一个标识，变量值在0、1之间来回切换，当这个变量大于0的时候获得了“令牌”，从这点可以看出，其实*park*();*unpark*(t);方法就是在改变这个变量的值，来达到线程的阻塞和唤醒的。

## 基于CAS锁的总结：

ReentrantLock 和synchronized的区别，ReentrantLock 更灵活，更方便

CountDownLatch：倒计时，什么时候计数完了，门闩打开，程序继续往下执行

CyclicBarrier：一个栅栏，循环使用，什么时候人满了，栅栏放倒，大家冲出去

ReadWriteLock：读写锁 \*\*\*\*

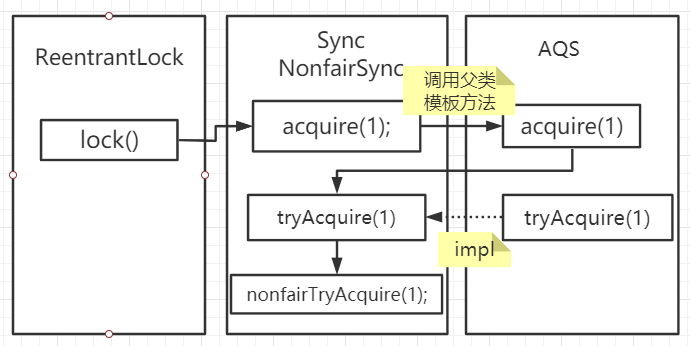
Semaphore：限流用的

Exchanger：两个线程交换数据

-----------都是基于AQS实现的---------------

## AQS原理，ReentranLock源码，设计原理，整体过程。 √

### 以ReentranLock源码为例，解读AQS



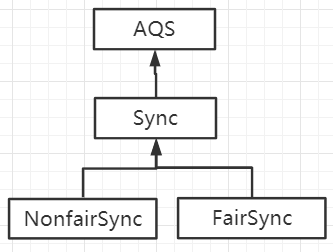
从图中可以看出：

1. ReentranLock类里有个静态内部类Sync，Sync有个子类NofairSync，调用ReentranLock的lock()方法，它会调用NofairSync的acquire(1)方法。

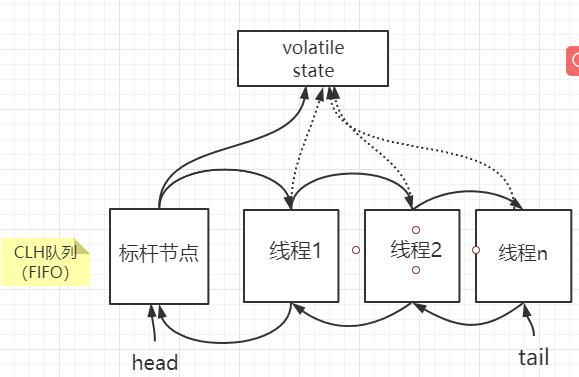
### 示意代码

注：可以是公平锁，也可以是非公平说，跟构造方法传true/false，new NonfairSync/new FairSync

|  |
| --- |
| **public** **class** ReentrantLock **implements** Lock, java.io.Serializable {    **private** **final** Sync sync;  **public** ReentrantLock() {  sync = **new** NonfairSync();  }  **public** **void** lock() {  sync.lock();  }  **abstract** **static** **class** Sync **extends** AbstractQueuedSynchronizer {  }  **static** **final** **class** NonfairSync **extends** Sync {  }  } |



### 解释



AQS队列，又称CLH队列，AQS的核心就是这个state状态。这个state所代表的意思随你定，随子类来定。

在ReentrantLock里，拿state来记录这个线程被重入了多少次。

state刚开始是0，当你获得了之后，它变成了1，表示当前线程获得了这把锁。这个线程又重入了一次，state就变成2了，又重入一次，就变成3。

什么时候释放，state从3变成2变成1变成0，说明当前线程释放了这把锁，state的 0、1就表示了加锁和解锁。这个就是AQS核心的东西。

谁拿到了state，表示谁就得到了这把锁。AQS的核心就是一个共享的数据，一堆互相争抢的线程，这就是AQS。

State根据子类的不同实现取不同的意义，这个state值下面跟着一个队列。

AQS的核心是一个state，以及监控这个state的双向链表：

这个队列是AQS自己内部维护的队列，在AQS内部的一个内部类Node，队列里面维护的是一个个node节点。在这个Node里面最重要的是它里面保留了一个Thread，线程。所以这个队列是线程队列，还有pre和next分别指向前面的节点和后面的节点，是个双向链表。

哪个线程得到了state这把锁，哪个线程要等待，都要进入到这个队列里面。当其中的一个node得到了state这把锁，就说明node线程持有这把锁。

当调用acquire(1)之后，看到state值是0，就直接拿到state这把锁，这是非公平的，上来就抢，抢不着，就进队列里acquireQueued(); 怎么抢得到呢？先得到当前线程，然后获取state的值，如果state的值等于0，用compareAndSetState(0,acquire);方法尝试把state改为1，假如改成功了，setExclusiveOwnerThread(current)把当前线程设置为独占state这把锁的状态，说明我已经得到这把锁，二而且这把锁是互斥的，我得到了别人是得不到的。因为别人再来的时候这个state的值已经变成1了，如果说当前线程已经是独占state这把锁了，就往后加1表示重入了。（重入方法执行完一个就减1，直到减到0）

。。。。面试要紧，未完待续（2020-06-17）

### 结论：

AQS（AbstractQueuedSynchronizer）的核心就是用CAS（CompareAndSet）去操作head和tail，就是说CAS操作代替了锁整条双向链表的操作。

### AQS为什么这么高效？

以AQS设置链表尾巴来理解AQS为什么效率这么高。

假如你要往一个链表上添加尾巴，尤其是好多线程都要往链表上添加尾巴，

普通的方法，多线程要保证线程安全，要加锁，一般情况下要锁定整个链表。（锁的太多太大了）

AQS用的并不是锁定整个链表，而是用CAS compareAndSetTail(oldTail,node);锁定尾巴，不需要把原来整个链表上锁。所以效率高。

### 为什么是双向链表？

因为你要添加一个线程节点的时候，需要看一下前面这个节点的状态，如果前面的节点是持有锁的过程中，你就得在后面等着，如果前面的节点已经取消了，那么你就应该越过这个节点，不去考虑他的状态，所以你需要看前面节点状态的时候，就必须是双向的、

IMG_256

# =========ThreadLocal =========

## ThreadLocal 原理分析，ThreadLocal为什么会出现OOM，出现的深层次原理

### 含义：

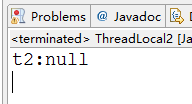
ThreadLocal，Thread，线程，Local，本地，线程本地。ThreadLocal线程局部变量

### 例子程序

t1线程在1秒的时候往ThreadLocal里设置了一个Person对象，第2秒的时候，t2线程从threadLocal里取值。但是是拿不到的，因为用ThreadLocal的时候，里面设置的值是线程独有的，哪个线程用到ThreadLocal的时候，只有自己往里面去设置值，设置的是只有自己的线程才能访问到的值，

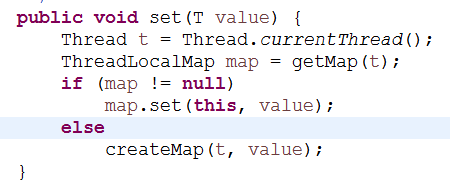
|  |
| --- |
| /\*\*  \* ThreadLocal线程局部变量  \*  \* ThreadLocal是使用空间换时间，synchronized是使用时间换空间  \* 比如在hibernate中session就存在与ThreadLocal中，避免synchronized的使用  \*  \* 运行下面的程序，理解ThreadLocal  \*  \* **@author** 马士兵  \*/  **package** com.mashibing.juc.c\_022\_RefTypeAndThreadLocal;  **import** java.util.concurrent.TimeUnit;  **public** **class** ThreadLocal2 {    **static** ThreadLocal<Person> *threadLocal* = **new** ThreadLocal<>();    **public** **static** **void** main(String[] args) {    **new** Thread(()->{  **try** {  TimeUnit.***SECONDS***.sleep(2);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  //t2 从 threadLocal取值  System.***out***.println("t2:"+*threadLocal*.get());  },"t2").start();    **new** Thread(()->{  **try** {  TimeUnit.***SECONDS***.sleep(1);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  //t1往 threadLocal设置值  *threadLocal*.set(**new** Person());  },"t1").start();  }    **static** **class** Person {  String name = "zhangsan";  }  } |

运行结果：



### 源码：

Set方法源码，ThreadLocal设置值的时候，首先拿到当前线程，然后是ThreadLocalMap，这个容器是一个key、Value的map。Key是this，指的值当前对象ThreadLocal，value就是Person类.



### 为什么要用ThreadLocal：

Spring使用ThreadLocal解决线程安全问题，Spring的声名式事务，就用到了ThreadLocal，事务一般是通过数据库控制的，Spring结合mybatis，可以吧整个事务写在配置文件中，它管理了一系列的方法：方法1，方法2，方法3，而这些方法可能都写了去配置文件拿数据库连接Connection，而Connection是线程不安全的，声名式事务可以把这几个方法结合在一起，视为一个完整的事务，如果这些方法拿的不同的Connection，这些东西是不能形成一个完整的事务的。把这个Connection放到这个线程的本地对象ThreadLocal里，以后再拿的时候，就从ThreadLocal里拿。

注意：

使用ThreadLocal里面的对象不用了，务必要remove掉，不然还会有内存泄露。

## 强引用、软引用、弱引用、虚引用

Java有4种引用：强、软、弱、虚

什么是引用？ Object o = new Object(); 这就是一个引用，一个变量指向一个new出来的对象，这个就叫引用。

### 强引用

Object o = new Object(); 这就叫强引用。

特点：只要有一个引用指向这个对象，那么垃圾回收器一定不会回收它。

M m = new M();

m=null;

System.gc();

这样m就会被回收，因为m=null; 就是说不会再有引用指向这个M对象了。m被回收。

### 软引用

当一个对象（字节数组）被一个软引用所指向的时候，只有系统内存不够的时候，才会回收它（字节数组）。

特点：垃圾回收不一定会回收，只有内存不足才回收。生存周期较长，做缓存用。

例子程序

|  |
| --- |
| /\*\*  \* 软引用  \* 软引用是用来描述一些还有用但并非必须的对象。  \* 对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围进行第二次回收。  \* 如果这次回收还没有足够的内存，才会抛出内存溢出异常。  \* 启动程序设置堆内存-Xmx20M  \*/  **package** com.mashibing.juc.c\_022\_RefTypeAndThreadLocal;  **import** java.lang.ref.SoftReference;  **public** **class** T02\_SoftReference {  **public** **static** **void** main(String[] args) {  SoftReference<**byte**[]> m = **new** SoftReference<>(**new** **byte**[1024\*1024\*10]);  //m = null;  System.***out***.println(m.get());  System.*gc*();  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(m.get());  //再分配一个数组，heap将装不下，这时候系统会垃圾回收，先回收一次，如果不够，会把软引用干掉  **byte**[] b = **new** **byte**[1024\*1024\*15];  System.***out***.println(m.get());  }  }  //软引用非常适合缓存使用 |

### 弱引用

弱引用意思是，只要遭遇GC就会回收。

只要垃圾回收看到这个引用是一个特别弱的引用指向的时候，就直接把它干掉。

一般用在容器里，WeakHashMap

弱引用典型应用：ThreadLocal，ThreadLocal里的Map的Entry类，继承了WeakReference

为什么Entry要使用弱引用？

若是强引用，即使tl = null; 但key的引用依然是指向ThreadLocal对象，所以会有内存泄露，使用了弱引用则不会、

但是还可能会有内存泄露的存在，ThreadLocal被回收，key的值变成null，则导致整个value也无法被访问到，因此依然存在内存泄露。

使用了ThreadLocal里面的对象，务必remove 掉，不然还会有内存泄露。

例子程序

|  |
| --- |
| /\*\*  \* 弱引用遭到gc就会回收  \*  \*/  package com.mashibing.juc.c\_022\_RefTypeAndThreadLocal;  import java.lang.ref.WeakReference;  public class T03\_WeakReference {  public static void main(String[] args) {  WeakReference<M> m = new WeakReference<>(new M());  System.out.println(m.get());  System.gc();  System.out.println(m.get());  ThreadLocal<M> tl = new ThreadLocal<>();  tl.set(new M());  tl.remove();  }  } |

### 虚引用

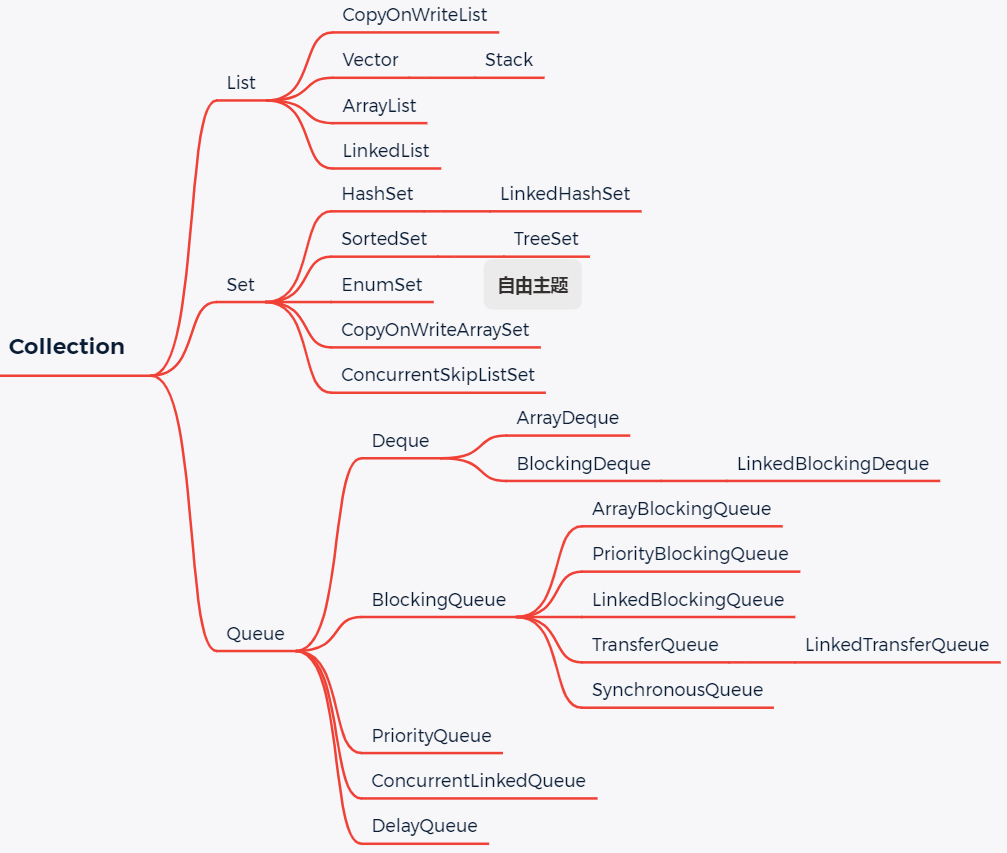
虚引用就干一件事，管理对外内存的。他是给写JVM虚拟机的人用的。

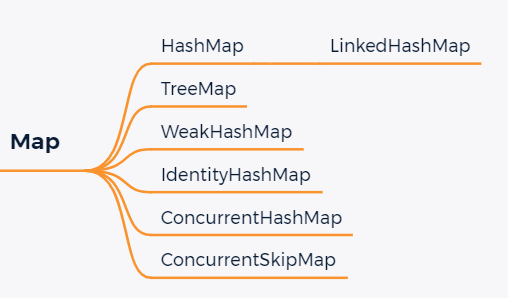
# =========并发容器=======

为线程池做准备。

## Java容器结构 √

容器分为Collection 和 Map





### 容器发展史

Java1.0时候容器只有两个：

* Vector: 实现List，一个个单独往里扔
* Hashtable：实现Map，一对对往里扔

在1.0设计有点问题，所有方法默认加了synchronized。因此最开始性能比较差。后来sun意识到了这一点，在Hashtable之后添加了HashMap，HashMap完全没加锁，两者代码上也有些区别。

一个完全加锁一个完全没加锁，能不能有个像HashMap这么好用的又加锁的呢？sun又设计了Connections容器工具类，里面有synchronizedMap 把它变成了加锁版本。

结论：Vector 和 Hashtable自带锁，但是基本不用

### **Collection：**

不管你这个容器是什么结构，你可以一个一个元素往里扔。

#### List

#### Set

#### Queue：

实现了队列逻辑，专为高并发而生。实现了很多多线程的访问方法（比如put阻塞式的放，take阻塞式的取）。

队列最主要的原因就是为了实现任务的装载的这种取和装，这里面最重要的就是阻塞队列，实现的初衷就是为了线程池、高并发做准备的。Deque双端队列，

### Map

一对一对往里扔，装的是Entry对象。

严格来说数组也属于容器，从数据结构上来讲，在物理上的 这种存储的数据结构其实只有两种：

* 连续存储的数组
* 非连续存储的链表

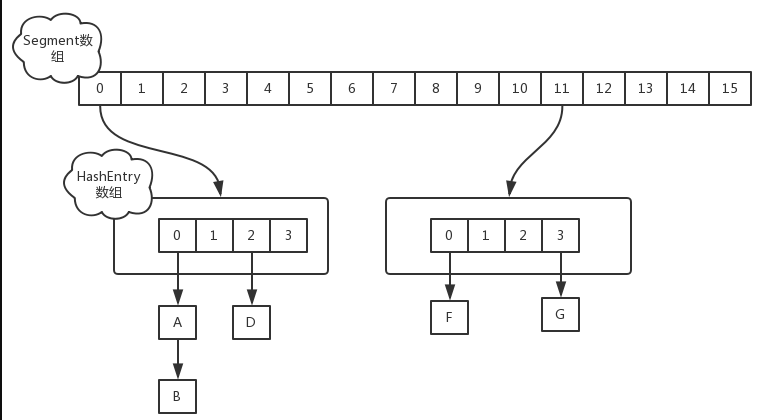
逻辑结构就非常多了

## 能把 ConcurrentHashMap 里面的实现详细的讲下吗？ √

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

### **JDK1.7的实现**

ConcurrentHashMap的数据结构是由一个Segment数组和多个HashEntry组成



Segment是一种可重入锁ReentrantLock，在ConcurrentHashMap里扮演锁的角色，HashEntry则用于存储键值对数据。一个ConcurrentHashMap里包含一个Segment数组，Segment的结构和HashMap类似，是一种数组和链表结构， 一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素， 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。

Segment数组的意义就是将一个大的table分割成多个小的table来进行加锁，也就是上面的提到的锁分离技术，而每一个Segment元素存储的是HashEntry数组+链表，这个和HashMap的数据存储结构一样

#### **put操作**

对于ConcurrentHashMap的数据插入，这里要进行两次Hash去定位数据的存储位置

从上Segment的继承体系可以看出，Segment实现了ReentrantLock,也就带有锁的功能，当执行put操作时，

**第一次hash：**

会进行第一次key的hash来定位Segment的位置，如果该Segment还没有初始化，即通过CAS操作进行赋值，

**第二次hash：**

然后进行第二次hash操作，找到相应的HashEntry的位置，这里会利用继承过来的锁的特性，在将数据插入指定的HashEntry位置时（链表的尾端），会通过继承ReentrantLock的tryLock（）方法尝试去获取锁，如果获取成功就直接插入相应的位置，如果已经有线程获取该Segment的锁，那当前线程会以自旋的方式去继续的调用tryLock（）方法去获取锁，超过指定次数就挂起，等待唤醒

#### **get操作**

ConcurrentHashMap的get操作跟HashMap类似，只是ConcurrentHashMap第一次需要经过一次hash定位到Segment的位置，然后再hash定位到指定的HashEntry，遍历该HashEntry下的链表进行对比，成功就返回，不成功就返回null

#### **size操作**

计算ConcurrentHashMap的元素大小是一个有趣的问题，因为他是并发操作的，就是在你计算size的时候，他还在并发的插入数据，可能会导致你计算出来的size和你实际的size有相差（在你return size的时候，插入了多个数据），要解决这个问题，JDK1.7版本用两种方案

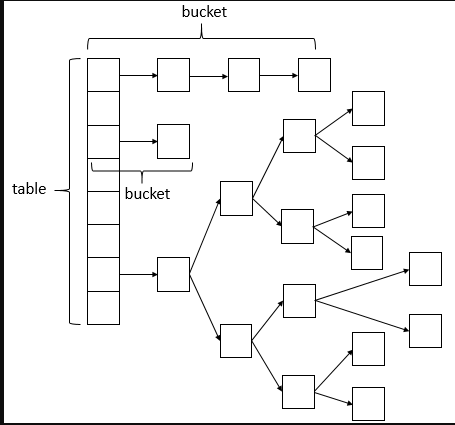
· 第一种方案他会使用不加锁的模式去尝试多次计算ConcurrentHashMap的size，最多三次，比较前后两次计算的结果，结果一致就认为当前没有元素加入，计算的结果是准确的

· 第二种方案是如果第一种方案不符合，他就会给每个Segment加上锁，然后计算ConcurrentHashMap的size返回

### JDK1.8的实现

而是直接用Node数组+链表+红黑树的数据结构来实现，并发控制使用Synchronized和CAS来操作，整个看起来就像是优化过且线程安全的HashMap，虽然在JDK1.8中还能看到Segment的数据结构，但是已经简化了属性，只是为了兼容旧版本

桶中的结构可能是链表，也可能是红黑树，红黑树是为了提高查找效率。



一些内部的一些结构组成:

#### **Node**

Node是ConcurrentHashMap存储结构的基本单元，继承于HashMap中的Entry，用于存储数据，

Node数据结构很简单，就是一个链表，但是只允许对数据进行查找，不允许进行修改

#### **TreeNode**

TreeNode继承与Node，但是数据结构换成了二叉树结构，它是红黑树的数据的存储结构，用于红黑树中存储数据，当链表的节点数大于8时会转换成红黑树的结构，他就是通过TreeNode作为存储结构代替Node来转换成黑红树

#### **Put**

这个put的过程很清晰，对当前的table进行无条件自循环直到put成功，可以分成以下六步流程来概述

1. 如果没有初始化就先调用initTable（）方法来进行初始化过程
2. 如果没有hash冲突就直接CAS插入
3. 如果还在进行扩容操作就先进行扩容
4. 如果存在hash冲突，就加锁来保证线程安全，这里有两种情况，一种是链表形式就直接遍历到尾端插入，一种是红黑树就按照红黑树结构插入，
5. 最后一个如果Hash冲突时会形成Node链表，在链表长度超过8，Node数组超过64时会将链表结构转换为红黑树的结构，break再一次进入循环
6. 如果添加成功就调用addCount（）方法统计size，并且检查是否需要扩容  
   总结与思考

其实可以看出JDK1.8版本的ConcurrentHashMap的数据结构已经接近HashMap，相对而言，ConcurrentHashMap只是增加了同步的操作来控制并发，从JDK1.7版本的ReentrantLock+Segment+HashEntry，到JDK1.8版本中synchronized+CAS+HashEntry+红黑树,相对而言，总结如下思考

1. JDK1.8的实现降低锁的粒度，JDK1.7版本锁的粒度是基于Segment的，包含多个HashEntry，而JDK1.8锁的粒度就是HashEntry（首节点）
2. JDK1.8版本的数据结构变得更加简单，使得操作也更加清晰流畅，因为已经使用synchronized来进行同步，所以不需要分段锁的概念，也就不需要Segment这种数据结构了，由于粒度的降低，实现的复杂度也增加了
3. JDK1.8使用红黑树来优化链表，基于长度很长的链表的遍历是一个很漫长的过程，而红黑树的遍历效率是很快的，代替一定阈值的链表，这样形成一个最佳拍档
4. JDK1.8为什么使用内置锁synchronized来代替重入锁ReentrantLock，我觉得有以下几点
   1. 因为粒度降低了，在相对而言的低粒度加锁方式，synchronized并不比ReentrantLock差，在粗粒度加锁中ReentrantLock可能通过Condition来控制各个低粒度的边界，更加的灵活，而在低粒度中，Condition的优势就没有了
   2. JVM的开发团队从来都没有放弃synchronized，而且基于JVM的synchronized优化空间更大，使用内嵌的关键字比使用API更加自然
   3. 在大量的数据操作下，对于JVM的内存压力，基于API的ReentrantLock会开销更多的内存，虽然不是瓶颈，但是也是一个选择依据

* 读的效率

比SynchronizedMap、Hashtable高多了。

SynchronizedMap和Hashtable虽然说读的效率会稍微低一些，但是插入的时候检查的东西特别少，就加个锁然后往里一插。

* 插入的效率

还不比synchronizedMap、Hashtable。

插入的时候内部做了各种各样的判断，本来是链表，到8之后又变成了红黑树，里面又做了各种各样的cas判断所以插入效率低。

ConcurrentHashMap jdk7使用分段锁，1.8以后使用cas

## ConcurrentHashMap 和 HashTable 中线程安全的区别？为啥建议用 ConcurrentHashMap ？ √

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。

## 把所有认识熟用的JUC( java.util.concurrent(简称JUC)包)下的类写出来，讲下使用，然后讲下原生的线程操作;

AtomicXX类

LockSupport

ReadWriteLock

ReentrantLock

## ConcurrentSkipListMap √

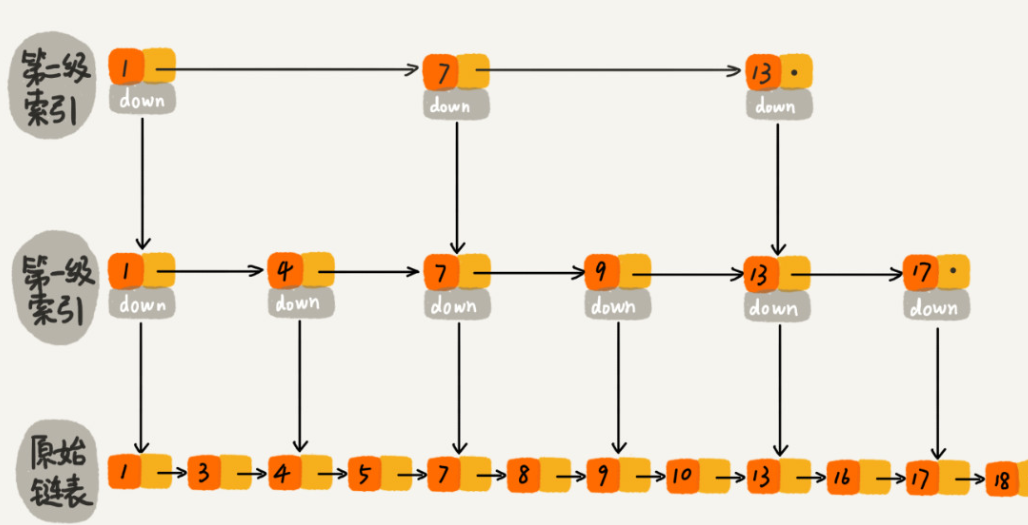
ConcurrentHashMap是无序的，需要用到排好序的Map，就用ConcurrentSkipListMap ，跳表结构，有序。是基于链表实现的一种类似“二分”的算法，

底层本身存储的元素的一个链表，是排好序的。时间复杂度是O(n)

在底层链表基础上，每隔一些节点就提取出来一个元素到上一层，把这一层称作索引。

再加多几级索引的话，效率将会进一步提升。****这种链表加多级索引的结构，就叫做跳表****。

跳表的查询时间复杂度可以达到O(logn)



## CopyOnWrite √

写时复制，有CopyOnWriteArrayList 、CopyOnWriteArraySet，两个类。

当写的的时候，需要复制，当需要往里面加元素的时候，吧元素复制出来。

### 适用场景：

写的特别少，读的特别多。写的效率比较低因为每次都要复制。

而且在迭代的时候数据也是不及时的，数据量少还好说，数据太多的时候，实时性可能就差距很大了。在多读取，少添加的时候，他的效果还是不错的（数据量大无所谓，只要你不添加，他都是好用的）**为什么提高效率：**

读的时候不加锁，写的时候用cas操作，会在原来基础上copy一份，copy的时候扩展出一个元素。然后把新添加的元素扔到这个位置上。与此同时，把指向老的容器的引用指向新的。

**源码：**

底层是一个volatile的数组：

private transient volatile Object[] array;保证了读取的那一刻读取的是最新的数据。

**Get方法很普通：**

public E get(int index) {

return get(getArray(), index);

}

**Add方法：**

reentrantlock加锁的，接下来就是复制数据和添加数据的过程，在setArray的过程中，把新的数组赋值给成员变量array（这里是引用的指向，java保证赋值的过程是一个原子操作）。

关于迭代，他采取的是获取传递给迭代器的数组值进行迭代，中间就算加入新的值也迭代不到。

## BlockingQueue

阻塞队列， BlockingQueue----->Queue-->Collection，队列的特点是：先进先出（FIFO）

比queue接口扩充的方法：

put(E e) :如果队列慢了，线程就会阻塞

E take():如果队列空了，线程阻塞

#### BlockingQueue的方法

面试题：Queue和list的区别



四组不同的行为方式解释:

* **1（异常）**

如果试图的操作无法立即执行，抛一个异常。

* **2（特定值）**

如果试图的操作无法立即执行，返回一个特定的值(常常是 true / false)。

* **3(阻塞)**

如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行。

* **4（超时）**

如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行，但等待时间不会超过给定值。返回一个特定值以告知该操作是否成功(典型的是 true / false)。

不能向BlockingQueue插入一个空对象，否则会抛出NullPointerException

#### 特点：

* ****不接受**null**元素。****

**试图**add**、**put**或**offer**一个**null**元素时，某些实现会抛出**NullPointerException**。**null**被用作指示**poll**操作失败的警戒值。**

* ****可以是限定容量的。****

它在任意给定时间都可以有一个 remainingCapacity，超出此容量，便无法无阻塞地 put 附加元素。没有任何内部容量约束的 BlockingQueue 总是报告Integer.MAX\_VALUE 的剩余容量。

* **实现主要用于生产者-使用者队列，还支持 Collection 接口**

因此，举例来说，使用 remove(x) 从队列中移除任意一个元素是有可能的。然而，这种操作通常不 会有效执行，只能有计划地偶尔使用，比如在取消排队信息时。

* ****实现是线程安全的。****

所有排队方法都可以使用内部锁或其他形式的并发控制来自动达到它们的目的。然而，大量的 Collection 操作（addAll、containsAll、retainAll 和removeAll）没有 必要自动执行，除非在实现中特别说明。因此，举例来说，在只添加了 c 中的一些元素后，addAll(c) 有可能失败（抛出一个异常）。

#### 具体实现：

### ArrayBlockingQueue

### 数组组成的有界队列，可以指定大小，往里扔的时候，一旦满了，put方法就会阻塞住。

### LinkedBlockingQueue

链表组成的无界队列，内部以 FIFO(先进先出)的顺序对元素进行存储

### DelayQueue

### synchronousQueue：

### TransferQueue

# =======线程池======

## 讲讲线程池的实现原理

## 多线程是解决什么问题的？线程池解决什么问题？

## 线程池的四种实现方式，重要参数及原理？任务拒接策略有哪几种？

## 线程池，如何设计的，里面的参数有多少种，里面的工作队列和线程队列是怎样的结构，如果给你，怎样设计线程池？

## Java线程池的核心属性以及处理流程；

## Java线程池的几个参数的意义和实现机制；

## Java线程池使用无界任务队列和有界任务队列的优劣对比；

## 线程池，如何根据CPU的核数来设计线程大小，如果是计算机密集型的呢，如果是IO密集型的呢？

## 线程池内的线程如果全部忙，提交⼀个新的任务，会发⽣什么？队列全部塞满了之后，还是忙，再提交会发⽣什么？

## ExecutorService你一般是怎么⽤的？是每个Service放一个还是个项目放一个？有什么好处？

## 从源码详细说下 Java 里面的线程池吧，使用线程池有什么要注意的地方？你们公司有规范吗？

线程池：主要工作是控制工作线程的数量，处理过程中将任务放入队列，然后在线程中创建任务，如果线程超过了最大线程数量，则排队等候，等其他线程执行完再从队列中取出任务来执行

主要特点：线程复用，控制最大线程数，管理线程

优点：1降低资源消耗2提高响应速度3提高线程的可管理性

获取线程的方式1、继承Thread类2、实现Runable接口3、实现Callable接口4、线程池，newFixedThreadPoll固定大小 newsingleThreadExcute单线程 newCacheTreadPoll不固定大小

public static ExecutorService newFixedThreadPool(int var0) {

return new ThreadPoolExecutor(var0, var0, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue());

}

public static ExecutorService newSingleThreadExecutor() {

return new Executors.FinalizableDelegatedExecutorService(new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue()));

}

public static ExecutorService newCachedThreadPool() {

return new ThreadPoolExecutor(0, 2147483647, 60L, TimeUnit.SECONDS, new SynchronousQueue());

}

线程池7大参数：

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue) {

this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,

Executors.defaultThreadFactory(), defaultHandler);

ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler)

corePoolSize：线程池中常驻线程数量，线程池中不会被注销的线程

maximumPoolSize：线程池中能够容纳同时执行的最大线程数量，必须大于等于1

keepAliveTime：多余空闲线程存活时间，当前线程数量超过corePollSize时当空闲时间达到keepAliveTime时，多余空闲线程会被注销，直到剩下corepoolsize个线程为止

unit：keepalivetime的时间单位

workQueue：任务队列，被提交但是未被执行的任务

threadFactory：生产线程池中工作线程的线程工厂，用于创建线程，一般使用默认

handler：拒绝策略，当任务队列workqueue满了并且工作线程数大于等于线程池的最大线程数maximumpoolsize时的拒绝方式

1、abortpolicy，默认方式，直接抛出RejectedExecutorException异常

2、callerRunsPolicy:调节模式，既不会抛出异常，也不会抛弃任务，而是将任务回退到任务调用者，从而降低新的任务流量

3、DiscardOldestPolicy:抛弃队列中等待最久的任务，然后把当前任务加入队列中尝试再次提交任务

4、DiscardPolicy：直接抛弃任务，不予处理也不抛出异常，如果允许丢弃任务，是最佳选择

线程池大小配置原则：1、io密集型 由于io密集型线程不是一直执行任务，则应配置尽量大些，一般可设置为cpu核数\*2；cpu核数/（1-阻塞系数） 阻塞系数一般是0.8-0.9

2、cpu密集型 由于cpu密集型一直都处于高速的运算，而没有阻塞，一般公式 cpu核数+1

# 三、Synchronized

## synchronized 与 lock 的区别

## synchronized与ReentraLock哪个是公平锁；

## 怎么排查死锁？死锁会怎样？有没有什么更好的替代方案？

## wait/notify/notifyAll⽅法需不需要被包含在synchronized块中？这是为什么？

死锁：两个及以上个线程争执行过程中夺资源而造成的相互等待的现象，若无其他外力干涉将会一直僵持下去

原因：系统资源不足；进程运行推进的顺序不合适；资源分配不当

死锁分析:linux ps-ef|grep ；windows jps -l获取进程号； jstack 进程号

# ======常见的原子操作类=======

1. AtomicInteger底层实现原理；

Atomic 怎么设计的？有用过里面的类吗？

# =======用过并发包下边的哪些类======

# =====一些高级概念=====

## 重入锁的概念，重入锁为什么可以防止死锁

## 产生死锁的四个条件（互斥、请求与保持、不剥夺、循环等待）

## 如何检查死锁（通过jConsole检查死锁）

## Java 8并法包下常见的并发类

## 偏向锁、轻量级锁、重量级锁、自旋锁的概念

## wait方法能不能被重写，wait能不能被中断；

## 常用的避免死锁方法；

## 写出一个必然会产生死锁的伪代码；

## Java中有哪些同步方案（重量级锁、显式锁、并发容器、并发同步器、CAS、volatile、AQS等）

## 多个线程同时读写，读线程的数量远远⼤于写线程，你认为应该如何解决并发的问题？你会选择加什么样的锁？

## 乐观悲观锁的设计，如何保证原子性，解决的问题；

## 乐观锁的业务场景及实现方式

## 怎么认为一个类是线程安全？线程安全的定义是什么？Java有多少个关键字进行同步？为什么这样设计？（聊了一大堆，一堆为什么）；

## 两个线程设计题。记得一个是：t1,t2,t3，让t1，t2执行完才执行t3，原生实现。

## 说下多线程，我们什么时候需要分析线程数，怎么分析，分析什么因素;

## 解释一下信号量

## 保证线程安全的还有其他的方式吗？

## 线程安全类和线程不安全的类，项目使用的时候你会怎么选择？怎么判断项目代码哪里会有线程不安全问题？

## cas 比较交换 自旋锁 unsafe

缺点:do while 循环开销大；只能保证一个共享数据；引发出ABA问题（版本号）

读写锁/互斥锁（独占锁：写锁；共享锁：读锁） reentrantLock和synchronized都是独占锁，只能同时被一个线程共享；

|  |
| --- |
|  |

futureTask实现了runable接口，构造方法中需要传入callAble接口实现类，多个线程共用一个futureTask，只有一个线程执行runable

synchronized与lock的区别

1、原始构成

synchronized是关键字属于jvm层次，底层通过monitor对象来完成，wait和notify等方法也依赖monotor对象在同步块或方法中才能被调用monitorenter，monitorexist；

lock属于api层次的调用

2、使用方法

synchronized不需要手动释放锁，不会造成死锁

lock需手动释放，lock和unclock需配对使用

3、等待是否可中断

synchronized锁不可中断，只能正常结束或异常退出

renntranLock可设置超时退出tryLock(time,unit);lockInterruptibly()方法代码块中调用interrup()方法可中断

4、加锁是否公平

synchronized是非公平锁

lock可通过构造方法，true公平锁，false非公平锁，默认非公平锁

5、绑定多个条件condition

synchronized不支持，只能随机唤醒一个或所有

reentrantLock可分组唤醒，精确唤醒某个线程

非静态方法的锁默认是this，静态方法的锁是对应的Class实例

某一个时刻内，只能有一个线程持有锁，无论几个方法

三个线程循环输出ABC

## 并行和并发的概念

## 进程间通信的方式

## 一个线程连着调用start两次会出现什么情况？

1. （由于状态只有就绪、阻塞、执行，状态是无法由执行转化为执行的，所以会报不合法的状态！）
2. 如何调试多线程的程序；