[线程8大核心基础 3](#_Toc34853037)

[一、实现多线程的方法 3](#_Toc34853038)

[1、实现Runnable接口 3](#_Toc34853039)

[2、继承Thread类 4](#_Toc34853040)

[两种方法对比，实现Runnable接口更好 4](#_Toc34853041)

[二、启动线程正确和错误的方法 8](#_Toc34853042)

[strat和run比较 8](#_Toc34853043)

[Start方法的含义 8](#_Toc34853044)

[启动线程 8](#_Toc34853045)

[准备工作 9](#_Toc34853046)

[不能重复调用start 9](#_Toc34853047)

[start源码解析 9](#_Toc34853048)

[三、停止线程的正确方法 10](#_Toc34853049)

[通常线程会在什么情况下停止 10](#_Toc34853050)

[正确的停止方法：interrupt 10](#_Toc34853051)

[通常线程会在什么情况下停止。 10](#_Toc34853052)

[线程可能被阻塞。 11](#_Toc34853053)

[如果线程在每次迭代后都阻塞。 12](#_Toc34853054)

[while内try/catch的问题 13](#_Toc34853055)

[实际开发中的两种最佳实践 14](#_Toc34853056)

[有限选择：传递中断。 14](#_Toc34853057)

[不想或无法传递：恢复中断。 15](#_Toc34853058)

[响应中断的方法总结列表 16](#_Toc34853059)

[正确停止的好处 17](#_Toc34853060)

[错误的停止方法 17](#_Toc34853061)

[被弃用的stop，suspend和resume方法 17](#_Toc34853062)

[用volatile设置boolean标记位 18](#_Toc34853063)

[错误原因 21](#_Toc34853064)

[修改方法 21](#_Toc34853065)

[停止线程相关函数解析 23](#_Toc34853066)

[Interrupt方法 23](#_Toc34853067)

[四、线程的生命周期 24](#_Toc34853068)

[六种状态 24](#_Toc34853069)

[New 24](#_Toc34853070)

[Runnable 25](#_Toc34853071)

[Blocked 25](#_Toc34853072)

[Waiting 25](#_Toc34853073)

[TimedWaiting 25](#_Toc34853074)

[Terminated 25](#_Toc34853075)

[线程各个状态转换代码 25](#_Toc34853076)

[阻塞状态 27](#_Toc34853077)

[五、Thread和Object类中和线程相关的重要方法 28](#_Toc34853078)

[方法概览 28](#_Toc34853079)

[wait，notify，notifyAll方法详解 28](#_Toc34853080)

[作用、用法：阻塞阶段、唤醒阶段、遇到中断 28](#_Toc34853081)

[wait，notify，notifyAll特点、性质 32](#_Toc34853082)

[生产者消费者设计模式 33](#_Toc34853083)

[使用wait和notify来实现 33](#_Toc34853084)

[sleep方法详解 35](#_Toc34853085)

[不释放锁（包括synchronized和lock） 35](#_Toc34853086)

[sleep方法响应中断 37](#_Toc34853087)

[join方法 38](#_Toc34853088)

[普通用法 38](#_Toc34853089)

[遇到中断 39](#_Toc34853090)

[在join期间，线程是什么状态？ Waiting 41](#_Toc34853091)

[源码 41](#_Toc34853092)

[join代替写法 42](#_Toc34853093)

[yield方法 43](#_Toc34853094)

[获取当前执行线程的引用：Thread.currentthread()方法 43](#_Toc34853095)

[六、线程各属性 44](#_Toc34853096)

[线程个属性纵览 44](#_Toc34853097)

[线程id 44](#_Toc34853098)

[线程名称 45](#_Toc34853099)

[守护线程 45](#_Toc34853100)

[线程优先级 45](#_Toc34853101)

[各属性总结 46](#_Toc34853102)

[七、线程的未捕获异常UncaughtException应该如何处理 46](#_Toc34853103)

[为什么需要UncaughtExceptionHandler？ 46](#_Toc34853104)

[两种解决方案 46](#_Toc34853105)

[八、多线程会导致的问题 47](#_Toc34853106)

[什么是线程安全 47](#_Toc34853107)

[什么情况下会出现线程安全问题，怎么避免 47](#_Toc34853108)

[1、 运行结果错误a++多线程下出现消失的请求现象 47](#_Toc34853109)

[2、 活跃性问题：死锁、活锁、饥饿 49](#_Toc34853110)

[3、 对象发布和初始化的时候的安全问题 50](#_Toc34853111)

[什么是发布 50](#_Toc34853112)

[什么是逸出 50](#_Toc34853113)

[如何解决逸出 52](#_Toc34853114)

[九、常见面试题 53](#_Toc34853115)

[1、实现多线程有几种方式 53](#_Toc34853116)

[2、一个线程两次调用start方法会出现什么情况 53](#_Toc34853117)

[3、既然start方法会调用run方法，为什么要调用start方法，不直接调用run 53](#_Toc34853118)

[4、如何停止线程 53](#_Toc34853119)

[5、如何处理不可中断的阻塞 53](#_Toc34853120)

[6、线程有哪几种状态？生命周期是什么？ 54](#_Toc34853121)

[7、用程序实现两个线程交替打印0-100的奇偶数？ 54](#_Toc34853122)

[8、手写生产者消费者模式 57](#_Toc34853123)

[9、为什么wait()需要在同步代码块内使用，而sleep()不需要 59](#_Toc34853124)

[10、为什么线程通信的方法wait()，notify()和notifyAll()被定义在Object类里？而sleep定义在Thread里 59](#_Toc34853125)

[11、wait方法是属于Object对象的，那么调用Thread.wait会怎么样？ 59](#_Toc34853126)

[12、如何选择用notify和notifyAll 60](#_Toc34853127)

[13、notifyAll之后所有线程都会再次抢夺锁，如果某线程抢夺失败怎么办？ 60](#_Toc34853128)

[14、用suspend和resume来阻塞线程可以吗？ 60](#_Toc34853129)

[15、JavaSe，javaEE，JavaME是什么？ 60](#_Toc34853130)

[16、JRE，JDK，JVM是什么关系？ 60](#_Toc34853131)

[17、wait/notify、sleep异同（方法属于哪个对象？线程状态怎么切换） 60](#_Toc34853132)

[18、Join期间，线程处于那种线程状态 61](#_Toc34853133)

[19、是否需要给线程设置为守护线程 61](#_Toc34853134)

[Java内存模型——底层原理 62](#_Toc34853135)

[JVM内存结构 java内存模型 java对象模型 62](#_Toc34853136)

[JVM内存结构和java虚拟机的运行时区域有关 62](#_Toc34853137)

[Java内存模型和java的并发编程有关 62](#_Toc34853138)

[Java对象模型，和java对象在虚拟机中的变现形式有关 62](#_Toc34853139)

[JMM是什么 63](#_Toc34853140)

[为什么需要JMM？ 63](#_Toc34853141)

[JMM是工具类和关键字的原理 63](#_Toc34853142)

[重排序 63](#_Toc34853143)

[可见性 64](#_Toc34853144)

[原子性 64](#_Toc34853145)

# 线程8大核心基础

## 一、实现多线程的方法

### 1、实现Runnable接口

package demo;  
*/\*\*  
 \** ***@ClassName*** *RunnableDemo  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/2/29 21:48  
 \** ***@Version*** *1.0  
 \*\*/*public class RunnableDemo implements Runnable{  
 public static void main(String[] args) {  
 new Thread(new RunnableDemo()).start();  
  
 }  
 @Override  
 public void run() {  
 System.*out*.println("使用runnable");  
 }  
}

### 2、继承Thread类

package demo;  
*/\*\*  
 \** ***@ClassName*** *ThreadDemo  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/2/29 21:46  
 \** ***@Version*** *1.0  
 \*\*/*public class ThreadDemo extends Thread{  
 public static void main(String[] args) {  
 new ThreadDemo().start();  
 }  
 @Override  
 public void run() {  
 System.*out*.println("使用Thread类");  
 }  
}

### 两种方法对比，实现Runnable接口更好

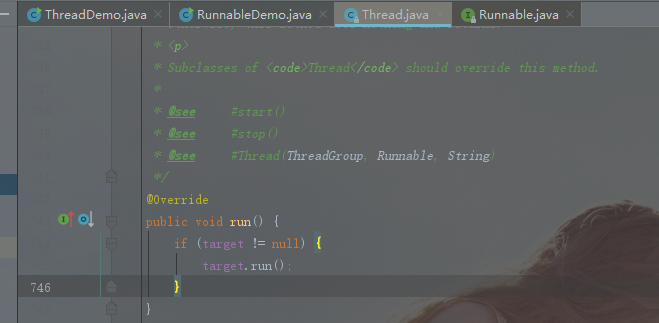
1、解耦。具体执行的任务（run方法）应该和线程的创建Thread类解耦

2、资源节约。每次新建任务，只能新建一个独立的线程，损耗比较大。实现Runnable接口

后续使用线程池可以节约资源。

3、java不支持多继承，大大减少的扩展性。

方法1是传入了一个target对象，target对象不为空，就会执行target对象的run方法



方法2继承Thread。重写了Thread的run方法。最终执行的是被重写的run方法。

如果使用两种方式共同实现多线程？

package demo;  
  
*/\*\*  
 \** ***@ClassName*** *RunnableAndThread  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/2/29 22:33  
 \** ***@Version*** *1.0  
 \* 同时使用Runnable和Thread两种实现线程的方式  
 \*\*/*public class RunnableAndThread {  
 public static void main(String[] args) {  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 System.*out*.println("runnable");  
 }  
 }){  
 @Override  
 public void run() {  
 System.*out*.println("thread");  
 }  
 }.start();  
 }  
}

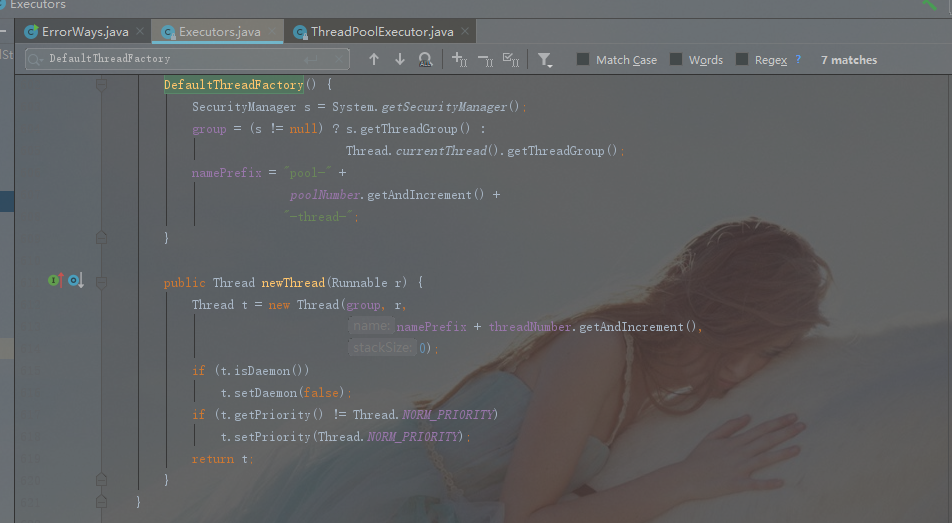
输出的结果为thread。方法被重写了。

准确的将，创建线程只有一种方式那就是构造Thread类，而实现线程的执行单元有两种方式

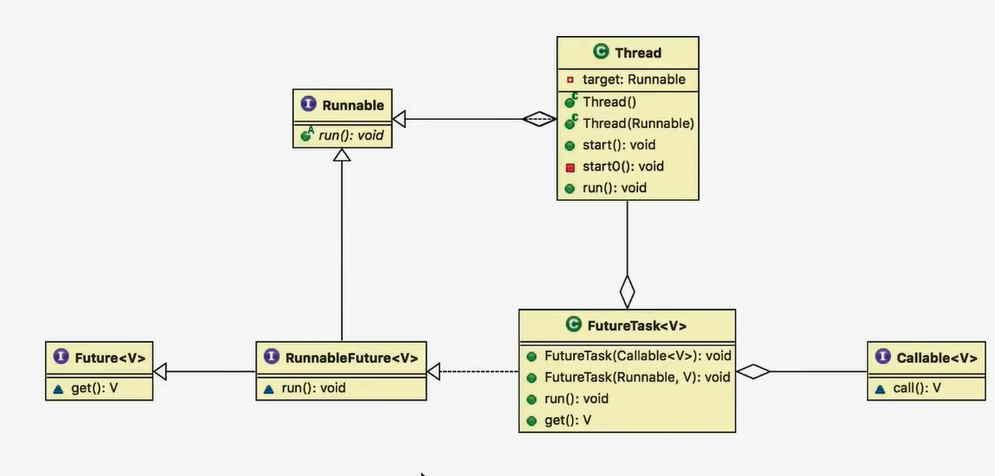
1. 实现Runnable接口的run方法，并把runnable实例传给Thread类
2. 重写Thread的run方法（继承Thread类）

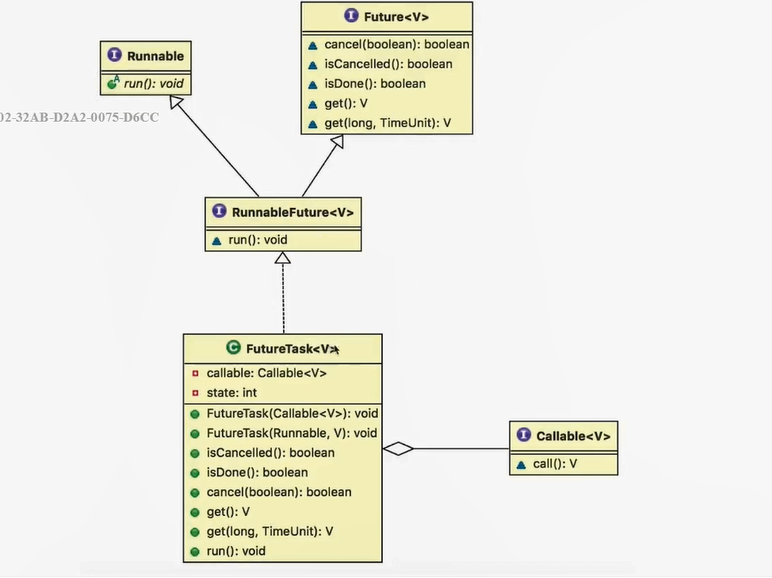
错误

1、线程池创建线程也算是一种新建线程的方式？通过源码可以看到，其实线程是也是通过new Thread类的方式创建线程的。



2、通过Callable和FutureTask创建线程也算是一种新的创建线程的方式





1. 无返回值是实现runnable接口，有返回值是实现callable接口，所以callable是新的实现线程的方式
2. 定时器

public static void main(String[] args) {  
 Timer timer =new Timer();  
 timer.scheduleAtFixedRate(new TimerTask() {  
 @Override  
 public void run() {  
 System.*out*.println(Thread.*currentThread*().getName());  
 }  
 },1000,1000);  
}

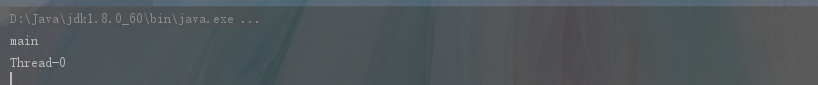
1. 匿名内部类
2. public static void main(String[] args) {  
    new Thread(){  
    @Override  
    public void run() {  
    System.*out*.println(Thread.*currentThread*().getName());  
    }  
    }.start();  
     
    new Thread(new Runnable() {  
    @Override  
    public void run() {  
    System.*out*.println(Thread.*currentThread*().getName());  
    }  
    }).start();  
     
     
   }
3. Lambda表达式

public static void main(String[] args) {  
 new Thread(() -> System.*out*.println(Thread.*currentThread*().getName())).start();  
}

## 二、启动线程正确和错误的方法

### strat和run比较

public class StartAndRunMethod {  
 public static void main(String[] args) {  
 Runnable runnable = () ->{  
 System.*out*.println(Thread.*currentThread*().getName());  
 };  
 runnable.run();  
 new Thread(runnable).start();  
 }  
}



### Start方法的含义

#### 启动线程

调用了.start()这行代码是有父线程或者主线执行了之后才创建了子线程（当前线程）；

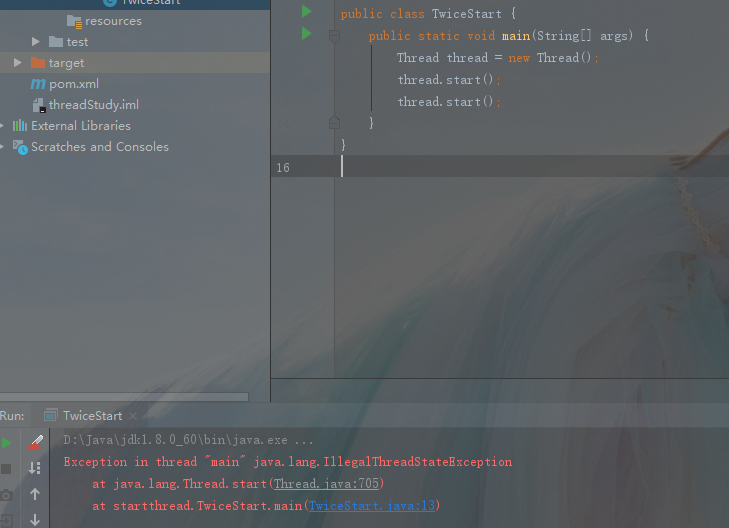
调用之后也不一定立即执行，只是告知JVM以合适的时候启动

#### 准备工作

子线程处于就绪状态（获取到除CPU以外的其他资源），被JVM调度到执行状态，获取cpu资源后变为运行状态

#### 不能重复调用start

一旦线程执行完毕，就会变为终止状态，不能再返回回去。



### start源码解析

1. 启动新线程检查线程状态

2、加入线程组

3、调用start0()



Run方法源码

判断target是否为null

## 三、停止线程的正确方法

使用interrupt来通知，而不是强制。

### 通常线程会在什么情况下停止

Run方法的代码都运行完毕，有异常，方法中没有捕获。

### 正确的停止方法：interrupt

#### 通常线程会在什么情况下停止。

package stopthread;  
  
*/\*\*  
 \** ***@ClassName*** *RightWayStopThreadWithoutSleep  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/1 21:06  
 \** ***@Version*** *1.0  
 \* run方法内没有sleep或wait方法时，停止线程  
 \*\*/*public class RightWayStopThreadWithoutSleep implements Runnable{  
  
 @Override  
 public void run() {  
 int num = 0;  
 while(!Thread.*currentThread*().isInterrupted()&&num<=Integer.*MAX\_VALUE*/2){  
 if(num%10000==0){  
 System.*out*.println(num+"是10000的倍数");  
 }  
 num++;  
 }  
 System.*out*.println("任务运行结束");  
 }  
  
 public static void main(String[] args) throws InterruptedException {  
 Thread thread=new Thread(new RightWayStopThreadWithoutSleep());  
 thread.start();  
 thread.*sleep*(1000);  
 thread.interrupt();  
 }  
}

#### 线程可能被阻塞。

package stopthread;  
  
*/\*\*  
 \** ***@ClassName*** *RightWayStopThreadWithSleep  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/1 21:16  
 \** ***@Version*** *1.0  
 \* 带有sleep的中断线程的写法  
 \*\*/*public class RightWayStopThreadWithSleep {  
 public static void main(String[] args) throws InterruptedException {  
 Runnable runnable = () -> {  
 int num = 0;  
 while (num <= 300 && Thread.*currentThread*().isInterrupted()) {  
 if (num % 100 == 0){  
 System.*out*.println(num+"是100的倍数");  
 }  
 num++;  
 }  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 };  
 Thread thread = new Thread(runnable);  
 thread.start();  
 //线程休眠0.5秒钟后，提出终止线程的信号，此时上面的线程还在休眠中（上面的线程休眠每次打印休眠1秒）；  
 thread.*sleep*(500);  
 thread.interrupt();  
 }  
}

#### 如果线程在每次迭代后都阻塞。

package stopthread;  
  
*/\*\*  
 \** ***@ClassName*** *RightWayStopThreadWithSleepEveryLoop  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/1 21:32  
 \** ***@Version*** *1.0  
 \*\*/  
  
/\*\*  
 \* 描述： 如果在执行过程中，每次循环都会调用sleep或wait等方法，那么不需要每次迭代都检查是否已中断  
 \*/*public class RightWayStopThreadWithSleepEveryLoop {  
 //打印依次休眠10毫秒 5秒的时候主线程休眠  
 public static void main(String[] args) throws InterruptedException {  
 Runnable runnable = () -> {  
 int num = 0;  
 try {  
 while (num <= 10000) {  
 if (num % 100 == 0) {  
 System.*out*.println(num + "是100的倍数");  
 }  
 num++;  
 Thread.*sleep*(10);  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 };  
 Thread thread = new Thread(runnable);  
 thread.start();  
 Thread.*sleep*(5000);  
 thread.interrupt();  
 }  
}

#### while内try/catch的问题

package stopthread;  
  
*/\*\*  
 \** ***@ClassName*** *CantInterrupt  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/1 21:55  
 \** ***@Version*** *1.0  
 \* 如果while里面放try/catch会导致中断失效  
 \*\*/*public class CantInterrupt {  
 public static void main(String[] args) throws InterruptedException {  
 Runnable runnable = () -> {  
 int num = 0;  
 //当5秒的时候告知要终止，此时异常被捕获了，但是循环没有到达结束条件，还会往下运行。  
 //即使加上了Thread.currentThread().isInterrupted()判断也不行。因为sleep过程中如果有线程中断了当前线程.抛出此异常时，当前线程的中断状态将被清除。  
 // if any thread has interrupted the current thread. The<i>interrupted status</i> of the current thread is cleared when this exception is thrown.  
 //如果有线程中断了当前线程。抛出此异常时，当前线程的中断状态将被清除。  
 while (num <= 10000 && !Thread.*currentThread*().isInterrupted()) {  
 if (num % 100 == 0){  
 System.*out*.println(num + "是100的倍数");  
 }  
 num++;  
 try {  
 Thread.*sleep*(10);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 };  
 Thread thread = new Thread(runnable);  
 thread.start();  
 Thread.*sleep*(5000);  
 thread.interrupt();  
 }  
  
}

#### 实际开发中的两种最佳实践

##### 有限选择：传递中断。

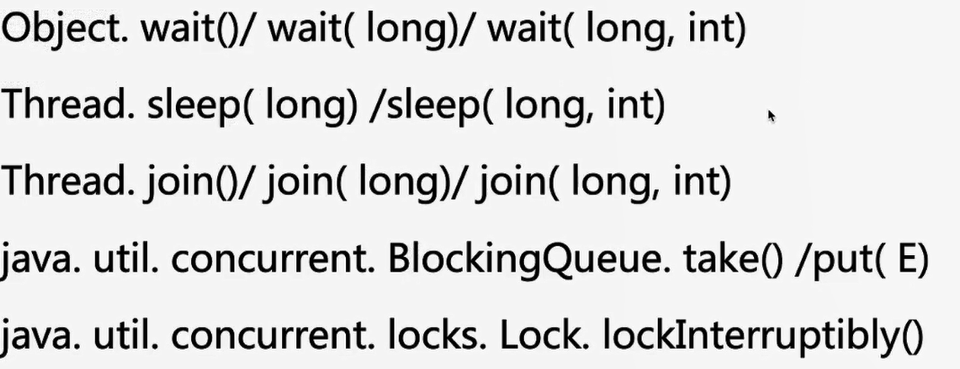
package stopthread;  
  
*/\*\*  
 \** ***@ClassName*** *RightWayStopThreadInProd  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/1 22:16  
 \** ***@Version*** *1.0  
 \* 最佳实践：catch了InterruptedException之后优先选择：在方法签名中抛出异常  
 \* 那么在run方法中就会强制try/catch  
 \*  
 \* 理解：因为sleep过程中如果有线程中断了当前线程.抛出此异常时，当前线程的中断状态将被清除。  
 \* 所以当下层方法如果把异常处理了，那么我们在run方法中就无法得到这个异常中断请求。所以底层方法应该讲异常抛出。这时候我们在run方法中调用  
 \* 的时候，我们就必须使用try/catch捕获，就可以收到这个异常中断请求。这时候我们可以根据实际情况作出相应的处理（打印日志或是停止等等）。  
 \*\*/*public class RightWayStopThreadInProd implements Runnable{  
 @Override  
 public void run() {  
 while(true){  
 System.*out*.println("aaaaa");  
 try {  
 ThrowInMethod();  
 } catch (InterruptedException e) {  
 //保存日志，停止程序。  
 e.printStackTrace();  
 }  
 }  
 }  
  
 private void ThrowInMethod() throws InterruptedException {  
 //直接try/catch是把异常处理在很低级的层次。而不是run方法。正确方法应该是throws InterruptedException  
 /\* try {  
 Thread.sleep(2000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }\*/  
 Thread.*sleep*(2000);  
 }  
  
 public static void main(String[] args) throws InterruptedException {  
 Thread thread = new Thread( new RightWayStopThreadInProd());  
 thread.start();  
 thread.*sleep*(500);  
 thread.interrupt();  
 }  
}

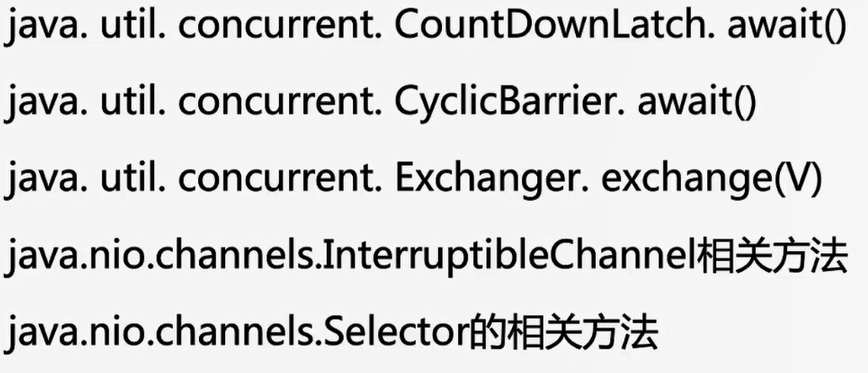
##### 不想或无法传递：恢复中断。

package stopthread;  
  
*/\*\*  
 \** ***@ClassName*** *RightWayStopThreadInProd  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/1 22:16  
 \** ***@Version*** *1.0  
 \* 最佳实践：在catch语句中调用Thread.currentThread().interrupt();来恢复设置中断状态，  
 \* 以便于在后续的执行中，依然能够检查到刚才发生的中断让他跳出  
 \*\*/*public class RightWayStopThreadInProd2 implements Runnable {  
 @Override  
 public void run() {  
 while (true) {  
 if(Thread.*currentThread*().isInterrupted()){  
 System.*out*.println("Interrputed,程序运行结束");  
 break;  
 }  
 reInMethod();  
  
 }  
 }  
  
 private void reInMethod() {  
 try {  
 Thread.*sleep*(2000);  
 } catch (InterruptedException e) {  
 //Thread.currentThread().interrupt();  
 e.printStackTrace();  
 }  
 }  
  
 public static void main(String[] args) throws InterruptedException {  
 Thread thread = new Thread(new RightWayStopThreadInProd2());  
 thread.start();  
 thread.*sleep*(500);  
 thread.interrupt();  
 }

}

##### 响应中断的方法总结列表





### 正确停止的好处

保证数据安全，把主动权交给被终止的线程

### 错误的停止方法

#### 被弃用的stop，suspend和resume方法

Suspend和resume带来的问题。

Suspend会让线程挂起，在恢复之前锁不会释放，很容易造成死锁（如果其他线程不及时唤醒，或者需要此线程的锁）

package stopthread;  
  
*/\*\*  
 \** ***@ClassName*** *StopThread  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/1 23:01  
 \** ***@Version*** *1.0  
 \* 错误的停止方法：用stop（）来停止线程，会导致线程运行到一半突然停止，没办法完成一个基本单位的操作，会造成脏数据  
 \*\*/*public class StopThread implements Runnable {  
  
 @Override  
 public void run() {  
 //模拟指挥军队：一共5个连队，每个连队10人，以连队为单位发放武器弹药，叫到号的士兵领取  
 for (int i = 0; i < 5; i++) {  
 System.*out*.println("连队"+i+"开始领取");  
 for(int j =0;j<10;j++){  
 System.*out*.println(j);  
 try {  
 Thread.*sleep*(10);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.println("连队"+i+"领取完毕");  
 }  
 }  
  
 public static void main(String[] args) {  
 Thread thread = new Thread(new StopThread());  
 thread.start();  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 thread.stop();  
 }  
}

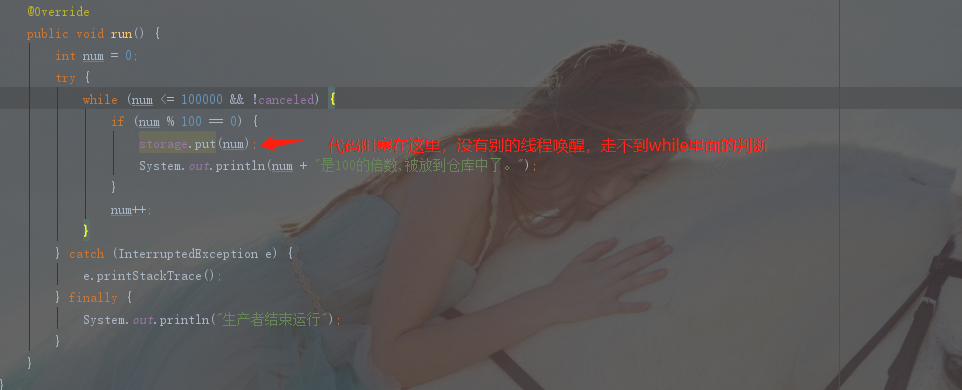
#### 用volatile设置boolean标记位

package stopthread.volatiledemo;  
  
import java.util.concurrent.ArrayBlockingQueue;  
import java.util.concurrent.BlockingQueue;  
  
*/\*\*  
 \** ***@ClassName*** *WrongWayVolatileCantStop  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/1 23:28  
 \** ***@Version*** *1.0  
 \* 演示用volatile的局限  
 \* 当陷入阻塞是，volatile是无法停止线程了  
 \* <p>  
 \* 此例中生产者的生产速度很快，消费者消费速度慢。就会出现阻塞队列满了以后，生产者会阻塞，等待消费者进一步消费  
 \*\*/*public class WrongWayVolatileCantStop {  
 public static void main(String[] args) throws InterruptedException {  
 ArrayBlockingQueue storage = new ArrayBlockingQueue(10);  
  
 Producer producer = new Producer(storage);  
 Thread producerThread = new Thread(producer);  
 producerThread.start();  
 Thread.*sleep*(1000);  
  
 Consumer consumer = new Consumer(storage);  
 while (consumer.needMoreNums()) {  
 System.*out*.println(consumer.storage.take()+"被消费了");  
 Thread.*sleep*(100);  
 }  
 System.*out*.println("消费者不需要更多数据了。");  
  
 //一旦消费不需要更多数据了，我们应该让生产者也停下来，但是实际情况  
 producer.canceled=true;  
 System.*out*.println(producer.canceled);  
 }  
}  
  
class Producer implements Runnable {  
  
 public volatile boolean canceled = false;  
  
 BlockingQueue storage;  
  
 public Producer(BlockingQueue storage) {  
 this.storage = storage;  
 }  
  
  
 @Override  
 public void run() {  
 int num = 0;  
 try {  
 while (num <= 100000 && !canceled) {  
 if (num % 100 == 0) {  
 storage.put(num);  
 System.*out*.println(num + "是100的倍数,被放到仓库中了。");  
 }  
 num++;  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 System.*out*.println("生产者结束运行");  
 }  
 }  
}  
  
class Consumer {  
  
 BlockingQueue storage;  
  
 public Consumer(BlockingQueue storage) {  
 this.storage = storage;  
 }  
  
 public boolean needMoreNums() {  
 if (Math.*random*() > 0.95) {  
 return false;  
 }  
 return true;  
 }  
}



可以看到当输出消费者不需要更多数据了，volatile变量也被设置为true但是没有输出生产者停止运行，程序还在运行。

##### 错误原因



##### 修改方法

使用interrupt

package stopthread.volatiledemo;  
  
import java.util.concurrent.ArrayBlockingQueue;  
import java.util.concurrent.BlockingQueue;  
  
*/\*\*  
 \** ***@ClassName*** *WrongWayVolatileFixed  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/2 0:08  
 \** ***@Version*** *1.0  
 \* 用中断来修复无尽等待的问题  
 \*\*/*public class WrongWayVolatileFixed {  
 public static void main(String[] args) throws InterruptedException {  
 WrongWayVolatileFixed body = new WrongWayVolatileFixed();  
 ArrayBlockingQueue storage = new ArrayBlockingQueue(10);  
  
 Producer producer = body.new Producer(storage);  
 Thread producerThread = new Thread(producer);  
 producerThread.start();  
 Thread.*sleep*(1000);  
  
 Consumer consumer = body.new Consumer(storage);  
 while (consumer.needMoreNums()) {  
 System.*out*.println(consumer.storage.take()+"被消费了");  
 Thread.*sleep*(100);  
 }  
 System.*out*.println("消费者不需要更多数据了。");  
  
 //一旦消费不需要更多数据了，我们应该让生产者也停下来，但是实际情况  
 producerThread.interrupt();  
  
 }  
  
 class Consumer {  
  
 BlockingQueue storage;  
  
 public Consumer(BlockingQueue storage) {  
 this.storage = storage;  
 }  
  
 public boolean needMoreNums() {  
 if (Math.*random*() > 0.95) {  
 return false;  
 }  
 return true;  
 }  
 }  
 class Producer implements Runnable {  
  
  
  
 BlockingQueue storage;  
  
 public Producer(BlockingQueue storage) {  
 this.storage = storage;  
 }  
  
  
 @Override  
 public void run() {  
 int num = 0;  
 try {  
 while (num <= 100000 && !Thread.*currentThread*().isInterrupted()) {  
 if (num % 100 == 0) {  
 storage.put(num);  
 System.*out*.println(num + "是100的倍数,被放到仓库中了。");  
 }  
 num++;  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 System.*out*.println("生产者结束运行");  
 }  
 }  
 }  
  
}

### 停止线程相关函数解析

#### Interrupt方法

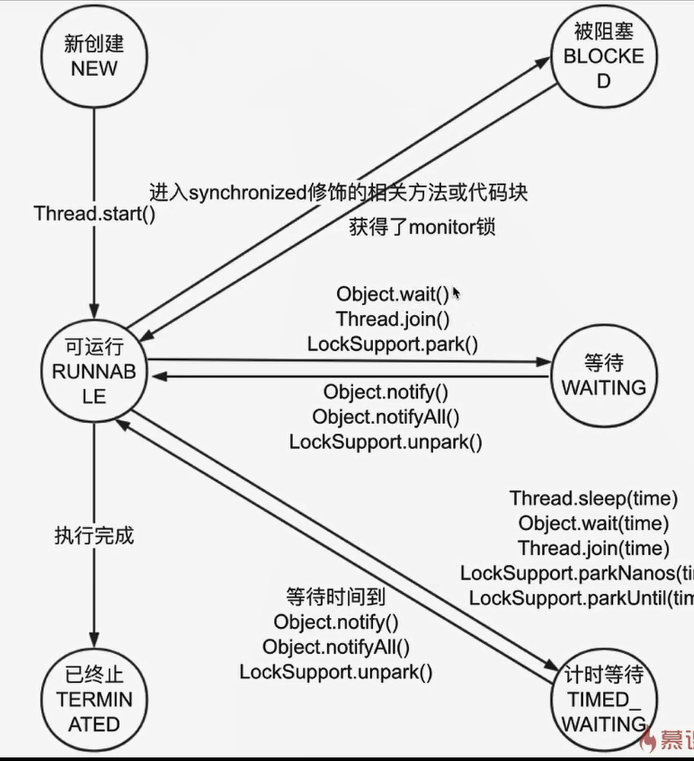
判断是否被中断相关方法

Static Boolean interrupted()

Boolean isInterrupted()

ThreadInterrupted()的目的对象

## 四、线程的生命周期



### 六种状态

#### New

已创建但还没有启动的线程（new了一个Thread但是没有调用start方法）

#### Runnable

调用了start方法之后就会进入runnable（可运行）状态。对应了操作系统中的两种状态（ready、running）。因为调用了start方法并不代表立刻会执行。

#### Blocked

当一个线程进入synchronized修饰的相关方法或代码块，并且该锁已经被其他线程拿走了。

#### Waiting

没有设置Timeout参数的Object.wait()方法

没有设置Timeout参数的Thread.join()方法

LockSupport.park()方法

#### TimedWaiting

Thread.sleep()方法

设置了Timeout参数的Object.wait()方法

设置了Timeout参数的Thread.join()方法

LocakSupport.parkNanos()方法

LockSupport.parkUntil()方法

#### Terminated

Run方法正常运行结束。

出现了没有捕获的异常导致结束。

#### 线程各个状态转换代码

package sixstates;  
  
*/\*\*  
 \** ***@ClassName*** *NewRunnableTerminated  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/2 12:55  
 \** ***@Version*** *1.0  
 \* 展示线程的new、runnable、terminated状态。即使是正在运行，也是runnable状态，而不是Running  
 \*\*/*public class NewRunnableTerminated implements Runnable{  
  
 public static void main(String[] args) {  
 Thread thread = new Thread(new NewRunnableTerminated());  
 //打印出NEW的状态  
 System.*out*.println(thread.getState());  
 thread.start();  
 System.*out*.println(thread.getState());  
 try {  
 Thread.*sleep*(10);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 //打印出RUNNABLE的状态，即使是正在运行，也是RUNNABLE，而不是RUNNING  
 System.*out*.println(thread.getState());  
 try {  
 Thread.*sleep*(100);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 //打印出TERMINATED状态  
 System.*out*.println(thread.getState());  
 }  
  
 @Override  
 public void run() {  
 for (int i = 0; i < 1000; i++) {  
 System.*out*.println(i);  
 }  
 }  
}

package sixstates;  
  
*/\*\*  
 \** ***@ClassName*** *BlockedWaitingTimedWaiting  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/2 14:43  
 \** ***@Version*** *1.0  
 \* 展示Blocked，Waiting，TimedWaiting  
 \*\*/*public class BlockedWaitingTimedWaiting implements Runnable{  
 public static void main(String[] args) {  
 BlockedWaitingTimedWaiting runnable = new BlockedWaitingTimedWaiting();  
 Thread thread1 = new Thread(runnable);  
 thread1.start();  
 Thread thread2 = new Thread(runnable);  
 thread2.start();  
 try {  
 Thread.*sleep*(5);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 //打印出Timed\_Waiting状态，因为正在执行Thread.sleep(1000);  
 System.*out*.println(thread1.getState());  
 //打印出BLOCKED状态，因为thread2想拿得到sync()的锁却拿不到  
 System.*out*.println(thread2.getState());  
 try {  
 Thread.*sleep*(1300);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 //打印出WAITING状态，因为执行了wait()  
 System.*out*.println(thread1.getState());  
  
 }  
  
 @Override  
 public void run() {  
 syn();  
 }  
  
 private synchronized void syn() {  
 try {  
 Thread.*sleep*(1000);  
 wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

### 阻塞状态

一般而言，把Blocked（被阻塞）、Waiting（等待）、Timed\_waiting（计时等待）都称为阻塞状态

## 五、Thread和Object类中和线程相关的重要方法

### 方法概览



### wait，notify，notifyAll方法详解

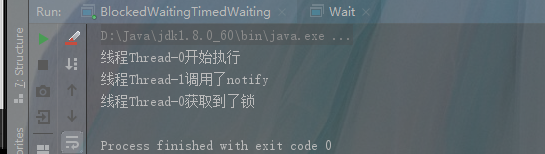
#### 作用、用法：阻塞阶段、唤醒阶段、遇到中断

调用wait方法后会进入阻塞状态，直到下面四种情况之一发生时，才会被唤醒

1. 另一个线程调用这个对象的notify（）方法且刚好被唤醒的是本线程
2. 另外一个线程调用这个对象的notifyAll()方法
3. 过了wait（long timeout）规定的超时时间，如果传入0就是永久等待
4. 线程自身调用interrupt()

Wait用法

package threadobjectclasscommonmethods;  
  
*/\*\*  
 \** ***@ClassName*** *Wait  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/2 17:05  
 \** ***@Version*** *1.0  
 \* 展示wait和notify的基本用法  
 \* 1、研究代码执行顺序  
 \* 2、证明wait释放锁  
 \*\*/*public class Wait {  
 public static Object *object* = new Object();  
 static class Thread1 extends Thread{  
 @Override  
 public void run() {  
 synchronized (*object*){  
 System.*out*.println("线程"+Thread.*currentThread*().getName()+"开始执行");  
 try {  
 *object*.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println("线程"+Thread.*currentThread*().getName()+"获取到了锁");  
 }  
 }  
 }  
  
 static class Thread2 extends Thread{  
 @Override  
 public void run() {  
 synchronized (*object*){  
 *object*.notify();  
 System.*out*.println("线程"+Thread.*currentThread*().getName()+"调用了notify");  
 }  
 }  
 }  
  
 public static void main(String[] args) {  
 Thread1 thread1 = new Thread1();  
 Thread2 thread2 = new Thread2();  
 thread1.start();  
 try {  
 Thread.*sleep*(200);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 thread2.start();  
 }  
}



Notify和notifuAll

package threadobjectclasscommonmethods;  
  
*/\*\*  
 \** ***@ClassName*** *WaitNotifyAll  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/2 17:17  
 \** ***@Version*** *1.0  
 \* 三个线程 线程1和线程2首先被阻塞，线程3唤醒 notify notifyAll  
 \* start先执行不代表线程先启动  
 \*\*/*public class WaitNotifyAll implements Runnable{  
  
 private static final Object *resourceA* = new Object();  
  
 @Override  
 public void run() {  
 synchronized (*resourceA*){  
 System.*out*.println(Thread.*currentThread*().getName()+"获取到锁");  
 try {  
 System.*out*.println(Thread.*currentThread*().getName()+"wait to start");  
 *resourceA*.wait();  
 System.*out*.println(Thread.*currentThread*().getName()+"is wating to end");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 public static void main(String[] args) {  
 Runnable r = new WaitNotifyAll();  
 Thread threadA = new Thread(r);  
 Thread threadB = new Thread(r);  
 Thread threadC = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 synchronized (*resourceA*){  
 *resourceA*.notifyAll();  
 //resourceA.notify();  
 System.*out*.println("ThreaC notify");  
 }  
 }  
 });  
 threadA.start();  
 threadB.start();  
 try {  
 Thread.*sleep*(200);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 threadC.start();  
 }  
}

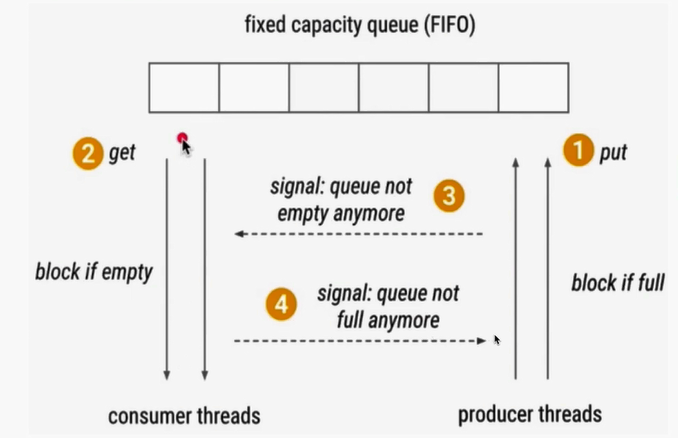
只释放当前锁

package threadobjectclasscommonmethods;  
  
*/\*\*  
 \** ***@ClassName*** *WaitNotifyReleaseOwnMonitor  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/2 17:29  
 \** ***@Version*** *1.0  
 \* 证明wait只释放当前的那把锁  
 \*\*/*public class WaitNotifyReleaseOwnMonitor {  
 private static volatile Object *resourceA* = new Object();  
 private static volatile Object *resourceB* = new Object();  
  
 public static void main(String[] args) {  
 Thread thread1 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 synchronized (*resourceA*) {  
 System.*out*.println("ThreadA got resourceA lock");  
 synchronized (*resourceB*) {  
 System.*out*.println("ThreadA got resourceB lock");  
 try {  
 System.*out*.println("ThreadA releases resourceA lock");  
 *resourceA*.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 }  
 }  
 }  
 });  
 Thread thread2 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 synchronized (*resourceA*) {  
 System.*out*.println("ThreadB got resourceA");  
 System.*out*.println("ThreadB try to resourceB");  
 synchronized (*resourceB*) {  
 System.*out*.println("ThreadB got resourceB");  
 }  
 }  
 }  
 });  
 thread1.start();  
 thread2.start();  
 }  
}

#### wait，notify，notifyAll特点、性质

1. 必须首先拥有monitor
2. 使用notify只能唤醒其中一个
3. 属于object类
4. 同时持有多个锁要注意锁之间是独立的。

#### 生产者消费者设计模式



##### 使用wait和notify来实现

package threadobjectclasscommonmethods;  
  
import java.util.Date;  
import java.util.LinkedList;  
  
*/\*\*  
 \** ***@ClassName*** *ProducerConsumerModel  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/2 17:46  
 \** ***@Version*** *1.0  
 \* 用wait/notify来实现  
 \*\*/*public class ProducerConsumerModel {  
 public static void main(String[] args) {  
 EvenStorage evenStorage = new EvenStorage();  
 Producer producer = new Producer(evenStorage);  
 Consumer consumer = new Consumer(evenStorage);  
 new Thread(producer).start();  
 new Thread(consumer).start();  
 }  
}  
 class Producer implements Runnable{  
 private EvenStorage storage;  
  
 public Producer(EvenStorage storage) {  
 this.storage = storage;  
 }  
  
 @Override  
 public void run() {  
 for(int i=0;i<100;i++){  
 storage.put();  
 }  
 }  
 }  
 //消费者  
 class Consumer implements Runnable{  
 private EvenStorage storage;  
  
 public Consumer(EvenStorage storage) {  
 this.storage = storage;  
 }  
  
 @Override  
 public void run() {  
 for(int i=0;i<100;i++){  
 storage.take();  
 }  
 }  
 }  
  
  
 class EvenStorage{  
 private int maxSize;  
 //作为存储商品的容器  
 private LinkedList<Date> storage;  
  
 public EvenStorage() {  
 maxSize = 10;  
 storage = new LinkedList<>();  
 }  
 //生产  
 public synchronized void put(){  
 //当仓库中商品已满，则等待  
 while(storage.size()==maxSize){  
 try {  
 wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 storage.add(new Date());  
 System.*out*.println("仓库中有"+storage.size()+"个产品");  
 notify();  
 }  
   
 //消费  
 public synchronized void take(){  
 //当仓库中没有商品时，则等待  
 while(storage.size()==0){  
 try {  
 wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.println("拿到了"+storage.poll()+",现在仓库还剩下："+storage.size());  
 notify();  
 }  
 }

### sleep方法详解

作用：指向让线程在预期的时间执行，其他时候不要占用CPU资源

#### 不释放锁（包括synchronized和lock）

和wait不同

package threadobjectclasscommonmethods;  
  
*/\*\*  
 \** ***@ClassName*** *SleepDontReleaseMonitor  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/2 19:01  
 \** ***@Version*** *1.0  
 \* 展示下承诺恒sleep的时候不是放synch的monitor，等sleep时间到了以后，正常结束后才会释放锁  
 \*\*/*public class SleepDontReleaseMonitor implements Runnable{  
 public static void main(String[] args) {  
 SleepDontReleaseMonitor sleepDontReleaseMonitor = new SleepDontReleaseMonitor();  
 new Thread(sleepDontReleaseMonitor).start();  
 new Thread(sleepDontReleaseMonitor).start();  
 }  
 @Override  
 public void run() {  
 syn();  
 }  
 private synchronized void syn(){  
 System.*out*.println("线程："+Thread.*currentThread*().getName()+"获取到monitor");  
 try {  
 Thread.*sleep*(5000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println("线程："+Thread.*currentThread*().getName()+"退出同步代码块");  
 }  
}

package threadobjectclasscommonmethods;  
  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
*/\*\*  
 \** ***@ClassName*** *SleepDontReleaseLock  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/2 19:29  
 \** ***@Version*** *1.0  
 \* 演示sleep不释放lock（需要手动释放）  
 \*\*/*public class SleepDontReleaseLock implements Runnable{  
 private static final Lock *lock* = new ReentrantLock();  
 @Override  
 public void run() {  
 *lock*.lock();  
 System.*out*.println("线程"+Thread.*currentThread*().getName()+"获取到锁");  
 try {  
 Thread.*sleep*(5000);  
 System.*out*.println("线程"+Thread.*currentThread*().getName()+"已经苏醒");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }finally {  
 *lock*.unlock();  
 }  
 }  
  
 public static void main(String[] args) {  
 SleepDontReleaseLock sleepDontReleaseLock = new SleepDontReleaseLock();  
 new Thread(sleepDontReleaseLock).start();  
 new Thread(sleepDontReleaseLock).start();  
 }  
}

#### sleep方法响应中断

sleep方法可以让线程进入Waiting状态，并且不占用CPU资源，但是不释放锁，直到规定时间后在执行，休眠期间如果被中断，会抛出异常并清除中断状态。

1. 抛出InterruptedException
2. 清除中断状态

使用TimeUnit进行休眠更加优雅

package threadobjectclasscommonmethods;  
  
import java.sql.Time;  
import java.util.Date;  
import java.util.concurrent.TimeUnit;  
  
*/\*\*  
 \** ***@ClassName*** *SleepInterrupted  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/2 20:38  
 \** ***@Version*** *1.0  
 \* TODO 每隔1秒钟输出当前事件，被中断  
 \* Thread.sleep()  
 \* TimeYnit.SECONDS.sleep()  
 \*\*/*public class SleepInterrupted implements Runnable {  
 @Override  
 public void run() {  
 for (int i = 0; i < 10; i++) {  
 System.*out*.println(new Date());  
 try {

//休眠3小时20分1秒  
 TimeUnit.*HOURS*.sleep(3);  
 TimeUnit.*MINUTES*.sleep(20);  
 TimeUnit.*SECONDS*.sleep(1);  
 } catch (InterruptedException e) {  
 System.*out*.println("被中断了");  
 e.printStackTrace();  
 }  
 }  
 }  
  
 public static void main(String[] args) throws InterruptedException {  
 Thread thread = new Thread(new SleepInterrupted());  
 thread.start();  
   
 thread.*sleep*(6500);  
  
 thread.interrupt();  
 }  
}

### join方法

作用：因为新的线程加入了，所以需要等待新加入的线程执行完在执行

用法：main等待thread1执行完毕

场景：需要等待其他资源加载完毕，其他线程.join（），就会等到资源加载完毕后主线程在执行

#### 普通用法

package threadobjectclasscommonmethods;  
  
*/\*\*  
 \** ***@ClassName*** *Join  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/2 20:54  
 \** ***@Version*** *1.0  
 \* TODO join 注意语句输出顺序，会变化  
 \*\*/*public class Join {  
 public static void main(String[] args) throws InterruptedException {  
 Thread thread1=new Thread(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println(Thread.*currentThread*().getName()+"执行完毕");  
 }  
 });  
 Thread thread2=new Thread(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println(Thread.*currentThread*().getName()+"执行完毕");  
 }  
 });  
 thread1.start();  
 thread2.start();  
 System.*out*.println("开始等待子线程执行完毕");  
 thread1.join();  
 thread2.join();  
 System.*out*.println("所有子线程执行完毕");  
 }  
}

#### 遇到中断

package threadobjectclasscommonmethods;  
  
*/\*\*  
 \** ***@ClassName*** *JoinInterrupt  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/2 20:58  
 \** ***@Version*** *1.0  
 \* TODO 演示join期间被中断的效果  
 \*\*/*public class JoinInterrupt {  
 public static void main(String[] args) {  
 Thread mainThread = Thread.*currentThread*();  
 Thread thread1 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 //主线程中断，这是当子线程join就会抛出中断异常  
 mainThread.interrupt();  
 Thread.*sleep*(5000);  
 System.*out*.println("Thread1 finished.");  
 } catch (InterruptedException e) {  
 System.*out*.println("子线程中断");  
 }  
 }  
 });  
 thread1.start();  
 System.*out*.println("等待子线程运行完毕");  
 try {  
 //子线程加入是会抛出异常，是因为子线程加入，我们等待子线程。等待过程中被中断，所以是主线程被中断，主线程抛出异常  
 thread1.join();  
 } catch (InterruptedException e) {  
 //输出main主线程中断了  
 System.*out*.println(Thread.*currentThread*().getName() + "主线程中断了");  
 //将中断传递给子线程  
 thread1.interrupt();  
 }  
 System.*out*.println("子线程已运行完毕");  
 }  
  
}

#### 在join期间，线程是什么状态？ Waiting

package threadobjectclasscommonmethods;  
  
*/\*\*  
 \** ***@ClassName*** *JoinThreadState  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/2 21:11  
 \** ***@Version*** *1.0  
 \* TODO 先join在mainThread.getState();通过debugger看线程join前后状态对比  
 \*\*/*public class JoinThreadState {  
 public static void main(String[] args) throws InterruptedException {  
 Thread mainThread = Thread.*currentThread*();  
 Thread thread = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 Thread.*sleep*(3000);  
 System.*out*.println(mainThread.getState());  
 System.*out*.println("Thread0 运行结束");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 });  
 thread.start();  
 System.*out*.println("等待子线程执行完毕");  
 thread.join();  
 System.*out*.println("等待子线程运行完毕");  
 }  
}

#### 源码

Join底层也是调用wait的方式。但是我们发现没有notify，那么是谁执行了唤醒操作（Threa在run方法执行完毕后会自动执行notify）

public final synchronized void join(long millis)  
throws InterruptedException {  
 long base = System.*currentTimeMillis*();  
 long now = 0;  
  
 if (millis < 0) {  
 throw new IllegalArgumentException("timeout value is negative");  
 }  
  
 if (millis == 0) {  
 while (isAlive()) {  
 wait(0);  
 }  
 } else {  
 while (isAlive()) {  
 long delay = millis - now;  
 if (delay <= 0) {  
 break;  
 }  
 wait(delay);  
 now = System.*currentTimeMillis*() - base;  
 }  
 }  
}

#### join代替写法

package threadobjectclasscommonmethods;  
  
*/\*\*  
 \** ***@ClassName*** *JoinPrinciple  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/2 21:32  
 \** ***@Version*** *1.0  
 \* TODO join的代替写法  
 \*\*/*public class JoinPrinciple {  
 public static void main(String[] args) throws InterruptedException {  
 Thread thread1=new Thread(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println(Thread.*currentThread*().getName()+"执行完毕");  
 }  
 });  
  
 thread1.start();  
 System.*out*.println("开始等待子线程执行完毕");  
 //thread1.join();  
 //等价于thread1.join();  
 synchronized (thread1){  
 thread1.wait();  
 }  
 System.*out*.println("所有子线程执行完毕");  
 }  
}

### yield方法

作用：释放CPU时间片

此时的状态还是runnable因为虽然释放了CPU时间片，但是不会释放锁，也不会陷入阻塞，下次CPU调度还是有可能被调度到。

和sleep区别：

Sleep期间线程调度器认为他已经阻塞了，不会调度

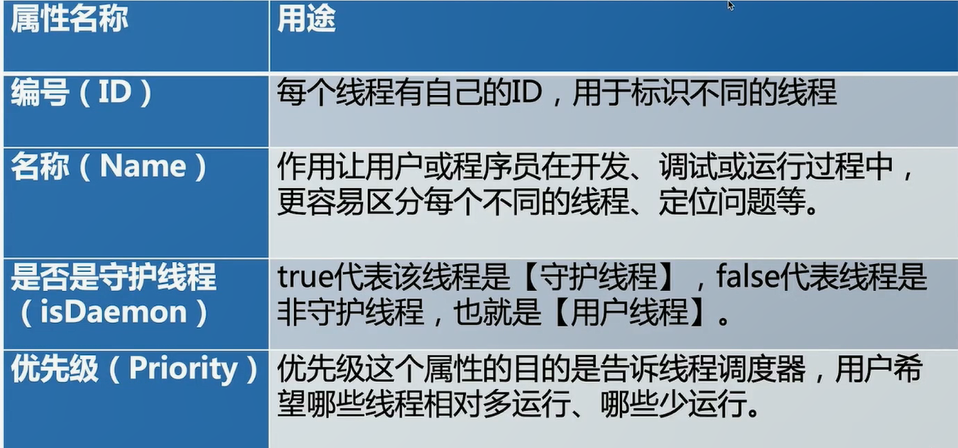
Yield只是暂时将把调度去让出去，但是又可以立刻竞争资源

### 获取当前执行线程的引用：Thread.currentthread()方法

package threadobjectclasscommonmethods;  
  
*/\*\*  
 \** ***@ClassName*** *CurrentThread  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/2 21:45  
 \** ***@Version*** *1.0  
 \* TODO 演示打印main thread0 thread1  
 \*\*/*public class CurrentThread implements Runnable{  
 public static void main(String[] args) {  
 new CurrentThread().run();  
 new Thread(new CurrentThread()).start();  
 new Thread(new CurrentThread()).start();  
 }  
 @Override  
 public void run() {  
 System.*out*.println(Thread.*currentThread*().getName());  
 }  
}

## 六、线程各属性

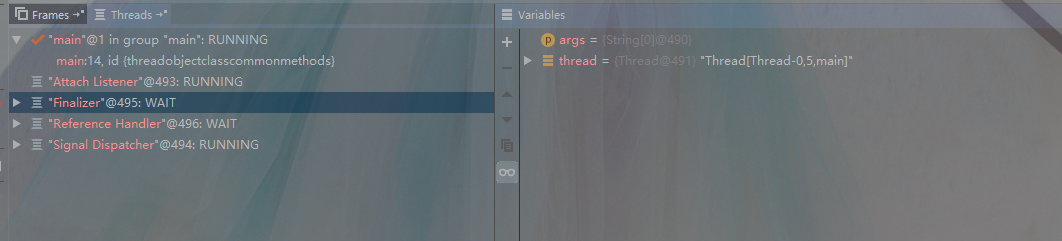
### 线程个属性纵览



### 线程id

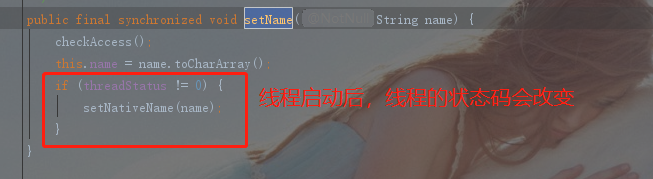
package threadobjectclasscommonmethods;  
  
*/\*\*  
 \** ***@ClassName*** *id  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/2 22:41  
 \** ***@Version*** *1.0  
 \* TODO ID从1开始，JVM运行起来后，我们会自己创建线程的id早已经不是2  
 \*\*/*public class id {  
 public static void main(String[] args) {  
 Thread thread = new Thread();  
 System.*out*.println("主线程ID："+Thread.*currentThread*().getId());  
 System.*out*.println("子线程id："+thread.getId());  
 }  
}

原因：当代码启动，JVM已经做了很多操作



### 线程名称

一旦线程启动了，java中线程的名称是可以改变的，但是native层面的名称是无法改变的



### 守护线程

作用：给用户线程提供服务

3个特性：

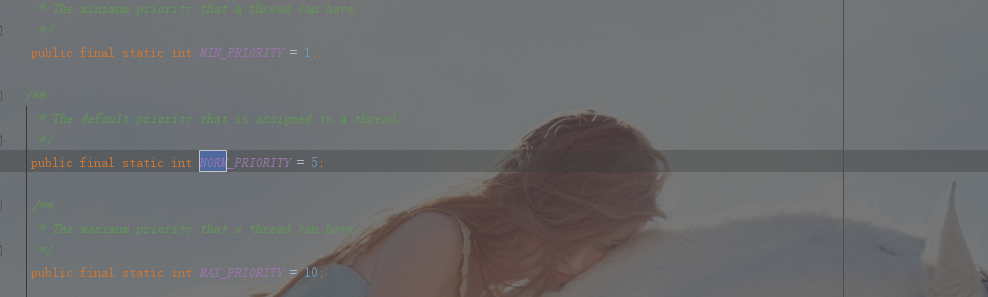
线程类型默认继承自父线程

被谁启动（通常守护线程都是由JVM启动）

不影响JVM退出（JVM退出的时候只会关注有没有用户线程，只要没有用户线程就会退出）

### 线程优先级

10个优先级，默认5



程序设计不应该依赖于优先级。因为不同操作系统对于优先级的处理是不同，而且优先级会被操作系统改变。

### 各属性总结

## 七、线程的未捕获异常UncaughtException应该如何处理

### 为什么需要UncaughtExceptionHandler？

主线程可以轻松发现异常，子线程却不行。

子线程异常无法用传统方法捕获

不能直接捕获的后果、提高健壮性。

### 两种解决方案

方案一（不推荐）：手动在每一个run方法中添加try/catch

利用UncaughtExceptionHandler

void uncaughtException(Thread t, Throwable e)

public void uncaughtException(Thread t, Throwable e) {

//默认情况下parent是null  
 if (parent != null) {  
 parent.uncaughtException(t, e);  
 } else {

//调用Thread.setDefaultUncaughExceptionHandler

//方法设置的全局handler进行处理  
 Thread.UncaughtExceptionHandler ueh =  
 Thread.*getDefaultUncaughtExceptionHandler*();  
 if (ueh != null) {  
 ueh.uncaughtException(t, e);  
 } else if (!(e instanceof ThreadDeath)) {

//全局handler也不存在就输出异常栈  
 System.*err*.print("Exception in thread \""  
 + t.getName() + "\" ");  
 e.printStackTrace(System.*err*);  
 }  
 }  
}

## 八、多线程会导致的问题

### 什么是线程安全

不管业务中遇到怎么的多个线程访问某个对象或某方的情况，而在编程这个业务逻辑的时候，都不需要做任何额外的处理，程序也可以正常运行（不会因为多线程而出错）就可以成为线程安全

### 什么情况下会出现线程安全问题，怎么避免

#### 运行结果错误a++多线程下出现消失的请求现象

package threaderr;  
  
import java.util.concurrent.BrokenBarrierException;  
import java.util.concurrent.CyclicBarrier;  
import java.util.concurrent.atomic.AtomicInteger;  
  
*/\*\*  
 \** ***@ClassName*** *MultiThreadsError  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/9 22:42  
 \** ***@Version*** *1.0  
 \* 运行结果出错  
 \* 演示计数不准确（减少），找出具体出错的位置  
 \*\*/*public class MultiThreadsError implements Runnable{  
 static volatile CyclicBarrier *cyclicBarrier1* =new CyclicBarrier(2);  
 static volatile CyclicBarrier *cyclicBarrier2* =new CyclicBarrier(2);  
 static MultiThreadsError *multiThreadsError*=new MultiThreadsError();  
 int index = 0;  
 final boolean[] marked =new boolean[100000];  
 static AtomicInteger *realIndex* = new AtomicInteger();  
 static AtomicInteger *wrongCount* = new AtomicInteger();  
 @Override  
 public void run() {  
 marked[0]=true;  
 for (int i = 0; i < 10000; i++) {  
 try {  
 *cyclicBarrier2*.reset();  
 *cyclicBarrier1*.await();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } catch (BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 index++;  
 try {  
 *cyclicBarrier1*.reset();  
 *cyclicBarrier2*.await();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } catch (BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 *realIndex*.incrementAndGet();  
 synchronized (*multiThreadsError*){  
 if(marked[index]&&marked[index-1]){  
 System.*out*.println("发生错误"+index);  
 *wrongCount*.incrementAndGet();  
 }  
 marked[index] =true;  
 }  
 }  
 }  
  
 public static void main(String[] args) throws InterruptedException {  
  
 Thread thread =new Thread(*multiThreadsError*);  
 Thread thread2 =new Thread(*multiThreadsError*);  
 thread.start();  
 thread2.start();  
 thread.join();  
 thread2.join();  
 System.*out*.println(*multiThreadsError*.index);  
 System.*out*.println("真正运行的次数"+*realIndex*.get());  
 System.*out*.println("错误的次数"+*wrongCount*.get());  
 }  
}

#### 活跃性问题：死锁、活锁、饥饿

package threaderr;  
  
*/\*\*  
 \** ***@ClassName*** *MultiThreadError  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/10 20:57  
 \** ***@Version*** *1.0  
 \* 演示死锁  
 \*\*/*public class MultiThreadError implements Runnable{  
 int flag=0;  
 static Object *o1*=new Object();  
 static Object *o2*=new Object();  
  
 public static void main(String[] args) {  
 MultiThreadError r1 = new MultiThreadError();  
 MultiThreadError r2 = new MultiThreadError();  
 r1.flag=1;  
 r2.flag=0;  
 new Thread(r1).start();  
 new Thread(r2).start();  
 }  
 @Override  
 public void run() {  
 System.*out*.println("flag:"+flag);  
 if(flag==1){  
 synchronized (*o1*){  
 try {  
 Thread.*sleep*(500);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 synchronized (*o2*){  
 System.*out*.println("1");  
 }  
 }  
 }  
 if(flag==0){  
 synchronized (*o2*){  
 try {  
 Thread.*sleep*(500);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 synchronized (*o1*){  
 System.*out*.println("0");  
 }  
 }  
 }  
 }  
}

#### 对象发布和初始化的时候的安全问题

##### 什么是发布

把对象让超过这个类的对象去用（比如public）

##### 什么是逸出

1. 方法返回一个private对象（private的本意是不让外部访问）

package threaderr;  
  
import java.util.HashMap;  
import java.util.Map;  
  
*/\*\*  
 \** ***@ClassName*** *MultiThreadEoor3  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/10 21:09  
 \** ***@Version*** *1.0  
 \* 发布逸出  
 \*\*/*public class MultiThreadEoor3 {  
 private Map<String,String> states;  
  
 public MultiThreadEoor3() {  
 this.states = new HashMap<>();  
 states.put("1","周2");  
 states.put("2","周3");  
 states.put("3","周4");  
 states.put("4","周5");  
 }  
 public Map<String,String> getStates(){  
 return states;  
 }  
  
 public static void main(String[] args) {  
 MultiThreadEoor3 multiThreadEoor3 = new MultiThreadEoor3();  
 Map<String, String> states = multiThreadEoor3.getStates();  
 System.*out*.println(states.get("1"));  
 states.remove("1");  
 System.*out*.println(states.get("1"));  
 }  
}

1. 还为完成初始化（构造函数没有完全执行完毕）就把对象提供给外界，

比如：在构造函数中为初始化完毕就this赋值；

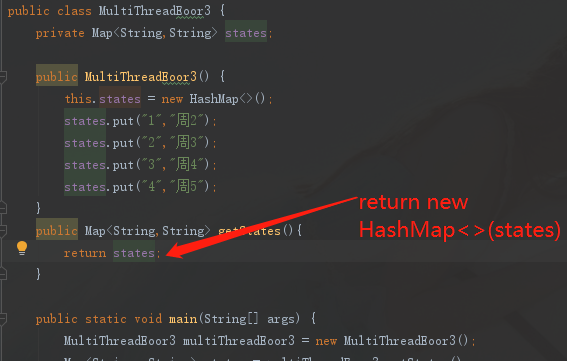
package threaderr;  
  
*/\*\*  
 \** ***@ClassName*** *MultiThreadEoor4  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/10 21:14  
 \** ***@Version*** *1.0  
 \* 初始化未完毕，就this赋值  
 \*\*/*public class MultiThreadEoor4 {  
 static Point *point*;  
  
 public static void main(String[] args) throws InterruptedException {  
 new PointMaker().start();  
 Thread.*sleep*(100);  
 if(*point*!=null){  
 System.*out*.println(*point*);  
 }  
 }  
}  
class Point{  
 private final int x,y;  
 public Point(int x,int y) throws InterruptedException {  
 this.x=x;  
 MultiThreadEoor4.*point*=this;  
 Thread.*sleep*(1000);  
 this.y=y;  
 }  
  
 @Override  
 public String toString() {  
 return "Point{" +  
 "x=" + x +  
 ", y=" + y +  
 '}';  
 }  
}  
class PointMaker extends Thread{  
 @Override  
 public void run() {  
 try {  
 new Point(1,1);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

隐式逸出——注册监听事件；

构造函数中运行线程

##### 如何解决逸出

返回副本



## 九、常见面试题

### 1、实现多线程有几种方式

准确的讲，创建线程只有一种方式那就是构造Thread类，而实现线程的执行单元有两种方式

实现Runnable接口的run方法，并把runnable实例传给Thread类

重写Thread的run方法（继承Thread类）

线程池，callable等表面上是一种新的创建线程的方式，实则都是通过以上两种方式来进行了。

### 2、一个线程两次调用start方法会出现什么情况

会抛出java.lang.IllegalThreadStateException。因为在调用了start之后会对线程的状态进行一个检查。

### 3、既然start方法会调用run方法，为什么要调用start方法，不直接调用run

调用了start方法才是真正意义上启动了线程，才会经历各个生命周期，如果只调用run方法，则这个线程不会被创建

### 4、如何停止线程

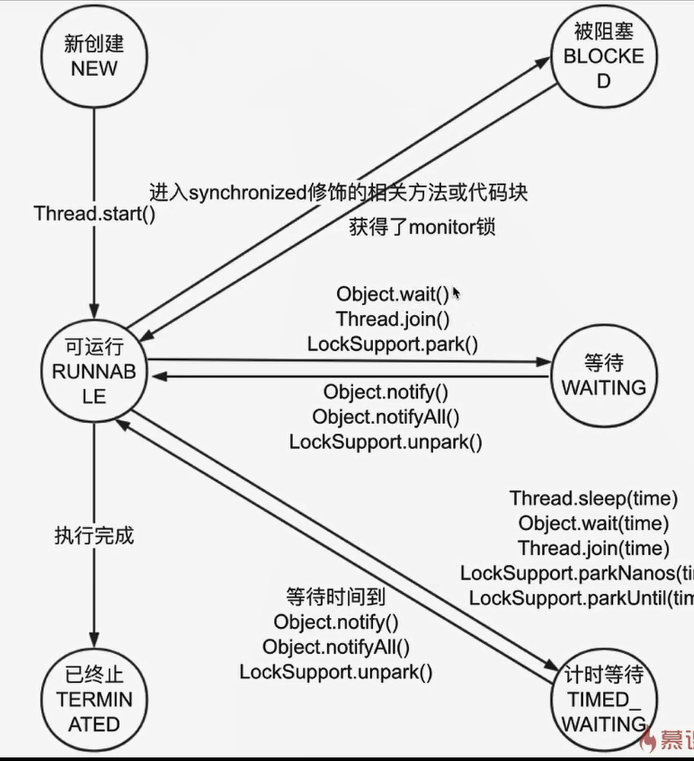
原理：

1. 用interrupt保证数据安全，把主动权交给被终止的线程
2. 想停止线程，要请求放、被停止方、子方法被调用方相互配合；请求放发出信号，被停止方在每次循环或适当的时候检查中断信号，在可能抛出InterrputException的地方处理这个信号。字方法优先抛出exception便于调用者做处理，或者捕获是收到的中信号后，再次设置为中断状态。
3. Stop/suspend以废弃，Suspend会让线程挂起，在恢复之前锁不会释放很容易造成死锁（如果其他线程不及时唤醒，或者需要此线程的锁）
4. volatile的boolean无法处理长时间阻塞的情况。

### 5、如何处理不可中断的阻塞

没有通用的解决方法，要依据相应的场景进行分析。比如ReentrantLock ，使用它的Lock方法，并且在Lock过程中把它阻塞了，是没有办法让他及时响应的。但是这个类提供了一个lockInterruptibly方法，这个方法是可以响应中断的。

### 6、线程有哪几种状态？生命周期是什么？



### 7、用程序实现两个线程交替打印0-100的奇偶数？

使用synchronized

package threadobjectclasscommonmethods;  
  
*/\*\*  
 \** ***@ClassName*** *WaitNotifyPrintOddEvenSyn  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/2 18:03  
 \** ***@Version*** *1.0  
 \* 两个线程交替打印0~100的奇偶数，用synchronized关键字实现  
 \*\*/*public class WaitNotifyPrintOddEvenSyn {  
 private static final Object *printResource* = new Object();  
 */\*\*  
 \* 1、新建两个线程1只处理偶数 2只处理奇数（用为运算）  
 \* 2、用synchronized作为通信机制  
 \*/* private static int *count*;  
  
 public static void main(String[] args) {  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 while (*count* < 100) {  
 synchronized (*printResource*) {  
 if ((*count* & 1) == 0){  
 System.*out*.println(Thread.*currentThread*().getName()+":"+*count*++);  
 }  
 }  
 }  
  
 }  
 },"偶数").start();  
  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 while (*count* < 100) {  
 synchronized (*printResource*) {  
 if ((*count* & 1) == 1){  
 System.*out*.println(Thread.*currentThread*().getName()+":"+*count*++);  
 }  
  
 }  
 }  
  
 }  
 },"奇数").start();  
  
  
 }  
}

更好的是应该使用wait和notify

package threadobjectclasscommonmethods;  
  
*/\*\*  
 \** ***@ClassName*** *WaitNotifyPrintOddEveWait  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/2 18:18  
 \** ***@Version*** *1.0  
 \* 两个线程交替打印0~100的奇偶数，用wait和notify  
 \*\*/*public class WaitNotifyPrintOddEveWait {  
 private static int *count* = 0;  
 private static final Object *resource* = new Object();  
 public static void main(String[] args) {  
 */\*\*  
 \* 1、拿到锁，就打印  
 \* 2、打印完，唤醒其他线程，自己就休眠  
 \*/* new Thread(new TurningRunner(),"偶数").start();  
 try {  
 Thread.*sleep*(100);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 new Thread(new TurningRunner(),"奇数").start();  
  
 }  
 static class TurningRunner implements Runnable{  
  
 @Override  
 public void run() {  
 while(*count*<=100){  
 synchronized (*resource*){  
 System.*out*.println(Thread.*currentThread*().getName()+";"+*count*++);  
 *resource*.notify();  
 if(*count*<=100){  
 try {  
 //如果任务还没结束就让出当前线程，自己休眠  
 *resource*.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
 }  
 }  
}

用synchronized会造成资源浪费，因为假设线程1在抢占锁是一致会抢占到，但是if条件缺一直不会成立，之后当线程2抢占到之后才会继续执行

### 8、手写生产者消费者模式

package threadobjectclasscommonmethods;  
  
import java.util.Date;  
import java.util.LinkedList;  
  
*/\*\*  
 \** ***@ClassName*** *ProducerConsumerModel  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/2 17:46  
 \** ***@Version*** *1.0  
 \* 用wait/notify来实现  
 \*\*/*public class ProducerConsumerModel {  
 public static void main(String[] args) {  
 EvenStorage evenStorage = new EvenStorage();  
 Producer producer = new Producer(evenStorage);  
 Consumer consumer = new Consumer(evenStorage);  
 new Thread(producer).start();  
 new Thread(consumer).start();  
 }  
}  
 class Producer implements Runnable{  
 private EvenStorage storage;  
  
 public Producer(EvenStorage storage) {  
 this.storage = storage;  
 }  
  
 @Override  
 public void run() {  
 for(int i=0;i<100;i++){  
 storage.put();  
 }  
 }  
 }  
 //消费者  
 class Consumer implements Runnable{  
 private EvenStorage storage;  
  
 public Consumer(EvenStorage storage) {  
 this.storage = storage;  
 }  
  
 @Override  
 public void run() {  
 for(int i=0;i<100;i++){  
 storage.take();  
 }  
 }  
 }  
  
  
 class EvenStorage{  
 private int maxSize;  
 //作为存储商品的容器  
 private LinkedList<Date> storage;  
  
 public EvenStorage() {  
 maxSize = 10;  
 storage = new LinkedList<>();  
 }  
 //生产  
 public synchronized void put(){  
 //当仓库中商品已满，则等待  
 while(storage.size()==maxSize){  
 try {  
 wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 storage.add(new Date());  
 System.*out*.println("仓库中有"+storage.size()+"个产品");  
 notify();  
 }  
  
 //消费  
 public synchronized void take(){  
 //当仓库中没有商品时，则等待  
 while(storage.size()==0){  
 try {  
 wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.println("拿到了"+storage.poll()+",现在仓库还剩下："+storage.size());  
 notify();  
 }  
 }

### 9、为什么wait()需要在同步代码块内使用，而sleep()不需要

为了使通信变得可靠，防止死锁的发生。加入有两个线程1和2。我们线程1wait之后想通过线程2唤醒（notify（）），但是如果没有synchronized的保护，可能没有执行到wait的时候就会切换到线程2。此时线程2如果把相关的操作执行了（notify）此时线程1在执行wait就与预期的效果发生分歧。

### 10、为什么线程通信的方法wait()，notify()和notifyAll()被定义在Object类里？而sleep定义在Thread里

在java中wait、notify、notifyAll是一个锁级别的操作，而锁是属于某一个对象的（锁是与对象绑定的）。现实中一个线程可能拥有多把锁，当我们把这个操作定义在Thread中，我们就无法让一个线程拥有多把锁。

### 11、wait方法是属于Object对象的，那么调用Thread.wait会怎么样？

可以把一个Thread当做锁，但是Thread比较特殊，线程退出的时候会自动执行notify。会使我们设计的流程受到影响。

### 12、如何选择用notify和notifyAll

主要取决于唤醒一个线程还是多个线程

### 13、notifyAll之后所有线程都会再次抢夺锁，如果某线程抢夺失败怎么办？

竞争失败会处于等待，等待这把锁的持有者释放，再去抢占。

### 14、用suspend和resume来阻塞线程可以吗？

suspend以废弃，Suspend会让线程挂起，在恢复之前锁不会释放很容易造成死锁（如果其他线程不及时唤醒，或者需要此线程的锁）

### 15、JavaSe，javaEE，JavaME是什么？

Se标准版（常用的），EE企业版，ME移动版适用于容量特别小的。

### 16、JRE，JDK，JVM是什么关系？

JRE是一个环境，要运行java代码就要有JRE。JDK（包含jre）是开发工具包，是程序员开发用的。JVM java虚拟机 JRE包含JVM

### 17、wait/notify、sleep异同（方法属于哪个对象？线程状态怎么切换）

相同：都会让线程阻塞；响应中断（即使在休眠期间也会响应中断抛出InterruptedException异常）

不同：

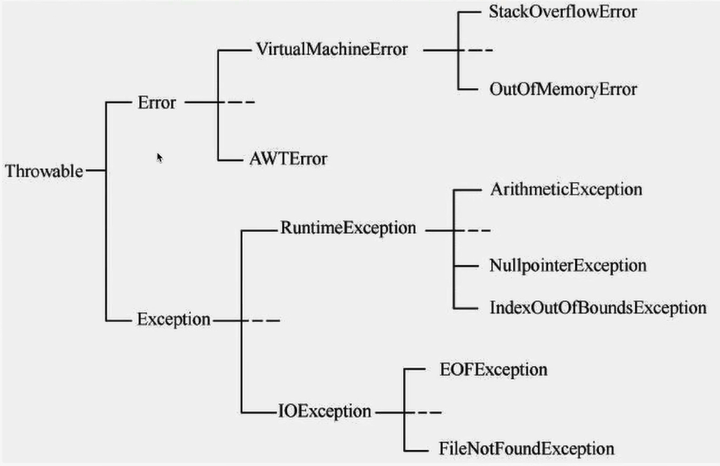
1. wait/notify必须在同步方法中，sleep不需要
2. wait会释放锁，sleep不释放
3. 指定时间的参数（wait可以指定也可以不指定，但是sleep必须指定）
4. Wait/notify属于Object；sleep属于Thread

### 18、Join期间，线程处于那种线程状态

Waiting

### 19、是否需要给线程设置为守护线程

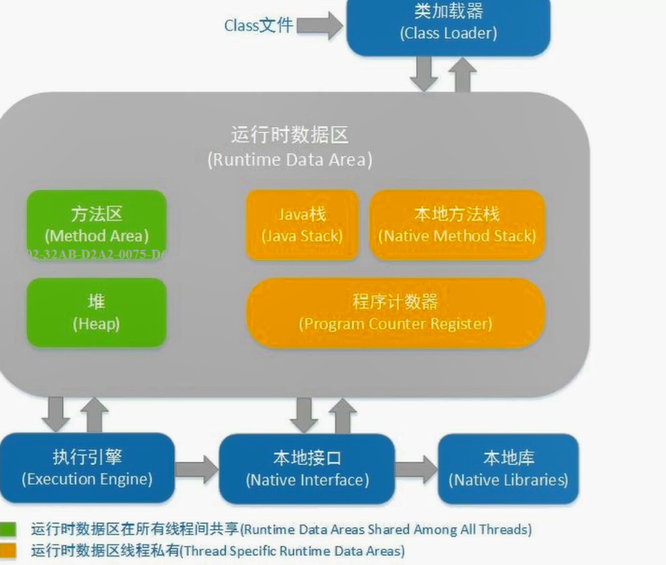
不需要，因为一旦我们把自己的线程设置为守护线程，当JVM发现已经只有守护线程了就会停止。



# Java内存模型——底层原理

## JVM内存结构 java内存模型 java对象模型

### JVM内存结构和java虚拟机的运行时区域有关



### Java内存模型和java的并发编程有关

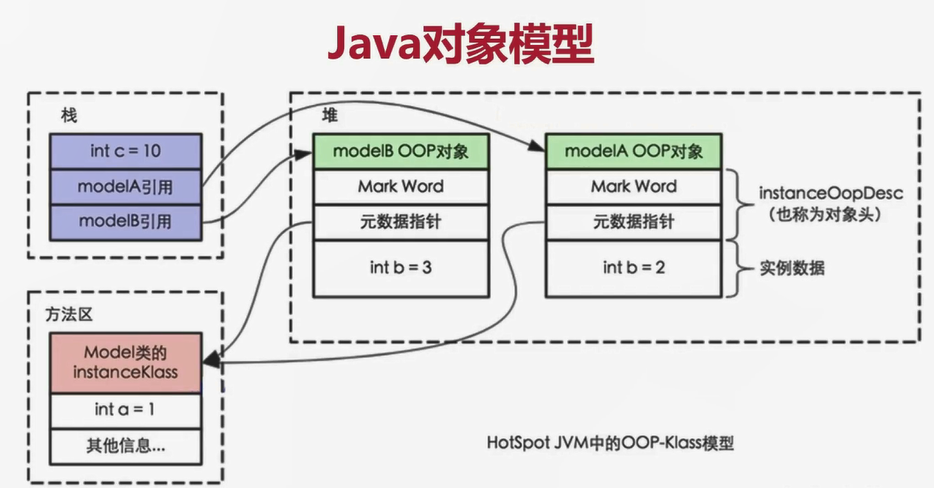
是一种规范

### Java对象模型，和java对象在虚拟机中的变现形式有关

Java对象自身的存储模型

JVM会给这个类创建一个instanceKlass，保存在方法区，用来在JVM层表示该java类

当我们在java代码中，使用new创建一个对象的时候，JVM会创建一个instanceOopDesc对象，这个对象中包含了对象头以及实例数据



## JMM是什么

Java Memory Model。

JMM是一种规范。是一组规范，需要各个JVM的实现来遵守JMM规范，以便于开发者可以利用这些规范，更方便地开发多线程程序

如果没有这样一个JMM内存模型来规范，那么很可能经过了不同JVM的不同规则的重排序之后，导致不同的虚拟机上运行的结果不一样。

### 为什么需要JMM？

依赖处理器，不同处理器的结果不一样，

无法保证并发安全

需要一个标准，让多线程运行的结果可预期

### JMM是工具类和关键字的原理

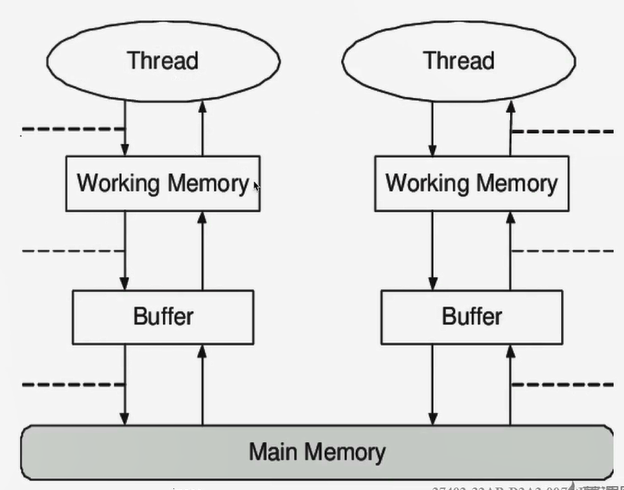
Volatile、synchronized、Lock等的原理都是JMM

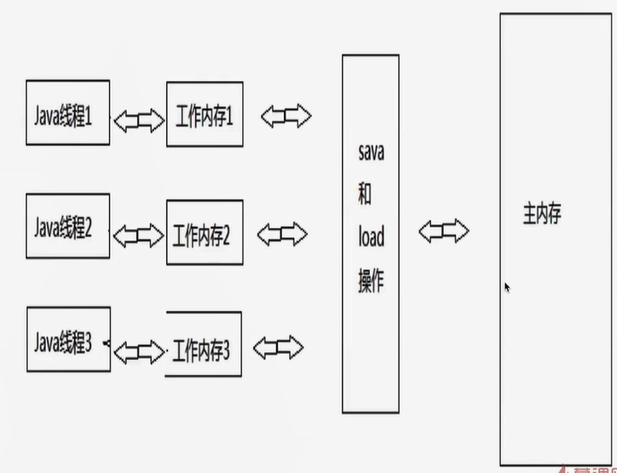
如果没有JMM，那就需要我们自己指定什么时候用内存栅栏等，有了JMM只需要用同步工具和关键字就可以开发并发程序

### 什么是主存和本地内存

Java作为高级语言，屏蔽了这些底层细节，用JMM定义了一套读写内存数据的规范，虽然我们不需要关心一级缓存和二级缓存的问题，但是，JMM抽象了主内存和本地内存的概念

这里的本地内存并不是真的是一块给每个线程分配的内存，而是JMM的一个抽象，是对于寄存器、一级缓存、二级缓存等的抽象





JMM有以下规定

1. 所有的变量都存储在主内存中同时每个线程也有自己独立的工作内存，工作内存中的变量内容是主内存中的拷贝
2. 线程不能直接读写主内存中的变量，而是只能操作自己工作内存中的变量，然后再同步到主内存中
3. 主内存是多个线程共享的，但线程间不共享工作内存，如果线程间需要通信，必须借助主内存中转来完成

所有的共享变量存在于主内存中，每个线程有自己的本地内存，而且线程读写共享数据也是通过本地内存交换的，所以导致了可见性问题。

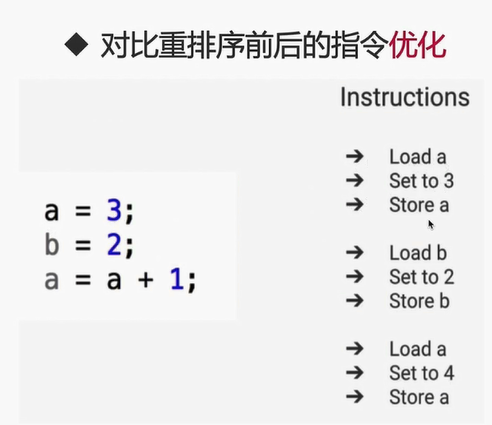
## 重排序

package jmm;  
  
import java.util.concurrent.CountDownLatch;  
  
*/\*\*  
 \** ***@ClassName*** *OutOfOrderExecution  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/10 23:02  
 \** ***@Version*** *1.0  
 \* 演示重排序  
 \* 直到达到某个条件才停止 测试小概率事件  
 \*  
 \*\*/*public class OutOfOrderExecution {  
 private static int *x* = 0, *y* = 0;  
 private static int *a* = 0, *b* = 0;  
  
 public static void main(String[] args) throws InterruptedException {  
 int i = 0;  
 for (; ; ) {  
 i++;  
 *x* = 0;  
 *y* = 0;  
 *a* = 0;  
 *b* = 0;  
 CountDownLatch latch = new CountDownLatch(3);  
 */\*\*  
 \* 三种情况  
 \* 1、a=1;x=b;b=1;y=a(1)最终结果 x=0 y=1  
 \* 2、b=1;y=a(0);a=1;x=b(1)最终结果 x=1 y=0  
 \* 3、b=1;a=1;x=b(1);y=a(1),最终结果 x=1 y=1  
 \* 会出现x=0 y=0是因为发生了重排序 4行代码的执行顺序为 y=a a=1 x=b b=1  
 \*/* Thread one = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 latch.countDown();  
 latch.await();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 *a* = 1;  
 *x* = *b*;  
 }  
 });  
 Thread two = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 latch.countDown();  
 latch.await();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 *b* = 1;  
 *y* = *a*;  
 }  
 });  
 two.start();  
 one.start();  
 latch.countDown();  
 one.join();  
 two.join();  
  
 String result = "第" + i + "次（" + *x* + "," + *y* + ")";  
 if (*x* == 0 && *y* == 0) {  
 System.*out*.println(result);  
 break;  
 } else {  
 System.*out*.println(result);  
 }  
 }  
 }  
}

### 什么是重排序

在线程1内部的两行代码的实际执行顺序和代码在java文件中的顺序不一致，代码指令并不是严格按照代码语句顺序执行的，他们的顺序被改变了，就是重排序。

### 重排序的好处：提高处理速度





### 重排序的三种情况：编译器优化、CPU指令重排、内存的重排序

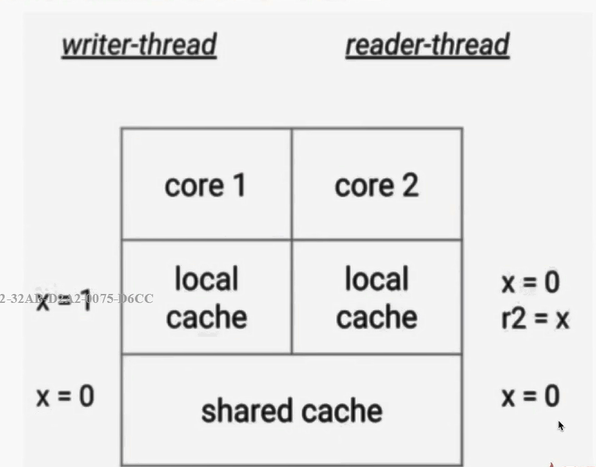
编译器：包括JVM，JIT编译器等

CPU指令重排：就算编译器不发生重排，CPU也可能对指令进行重排

内存的重排序：线程A的修改线程B却看不到，引出可见性问题

## 可见性

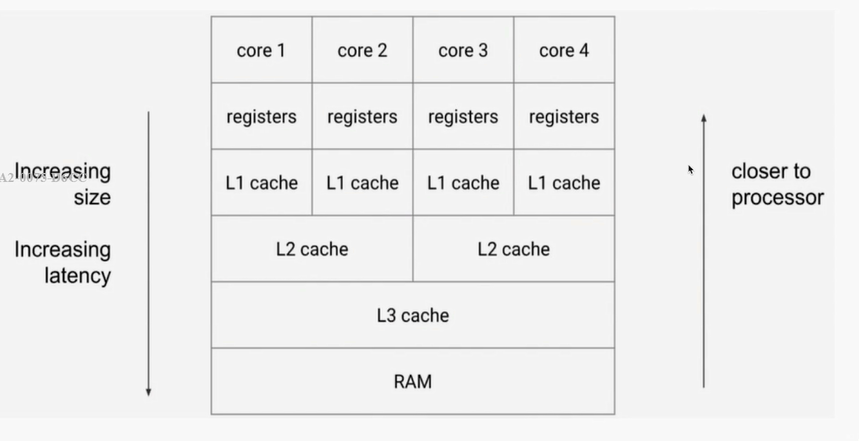
package jmm;  
  
*/\*\*  
 \* 演示可见性带来的问题  
 \** ***@ClassName*** *FieldVisibility  
 \** ***@Author*** *chenchen  
 \** ***@Date*** *2020/3/11 21:25  
 \** ***@Version*** *1.0  
 \*\*/*public class FieldVisibility {  
  
 /\*  
 //volatile解决可见性的问题  
 volatile int a = 1;  
 volatile int b = 2;\*/  
 int a =1;  
 int b=2;  
 private void change() {  
 a = 3;  
 b = a;  
 }  
  
  
 */\*\*  
 \* a=3 b=2第一个线程执行完a=3 线程2开始打印  
 \* a=1 b=2 没有经过任何修改直接打印  
 \* a=3 b=3 修改完成在打印  
 \* b=3 a=1 线程1已经做了change操作但是对于打印线程而言看到了b真实的值3 ，但是a还有没同步 所以找到a的原始值1  
 \*/* private void print() {  
 System.*out*.println("b=" + b + ";a=" + a);  
 }  
  
 public static void main(String[] args) {  
 while (true) {  
 FieldVisibility test = new FieldVisibility();  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 Thread.*sleep*(1);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 test.change();  
 }  
 }).start();  
  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 Thread.*sleep*(1);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 test.print();  
 }  
 }).start();  
 }  
  
 }  
  
  
}



使用volatile关键字会把本地内存刷新到共享内存



### 为什么会有可见性的问题



CPU有多级缓存，导致读的数据过期

高速缓存的容量比主内存小，但是速度仅次于寄存器，所以在CPU和主内存之间就多了Cache层

线程间的对于共享变量的可见性问题不是直接由多核引起的，而是由多个缓存引起的

如果所有核心都只用一个缓存，那就不存在内存可见性问题。

每个核心都会将自己需要的数据读到独占缓存中，数据修改后也是写入到缓存中，然后等待刷入到主存中。所以会导致有些核心读取的值是一个过期的值。

## 原子性

## Happens-before规则

### 单线程原则

在一个线程内 后面的语句一定能看到前面语句操作了什么。

### 锁操作（Synchronized和Lock）

如果一个线程对锁解锁了，另外一个线程对锁加锁了，加锁的线程一定能看到解锁之前的所有操作

### Volatile变量

只要变量被 volatile修饰 写线程只要完成操作，读线程就一定会读到最新的数据

### 线程启动

子线程所执行的所有语句都能看到主线程之前语句的结果

### 线程join

主线程等待join线程执行完成之后在执行。

### 传递性

如果hb（A,B）而且hb（B,C），那么可以推出hb（A，C）

### 中断

一个线程被其他线程interrupt，那么监测中断（isInterrupted）或者抛出InterruptedException一定能看到

### 构造方法

对象构造方法的最后一行指定happens-before于finalize()方法的第一行指令