

# 第1章 Java多线程技能

2020年10月22日 17:46

[https://blog.csdn.net/weixin\\_39754631/article/details/88844707](https://blog.csdn.net/weixin_39754631/article/details/88844707)

## 进程和线程

### 1. 进程

一个可并发执行的具有独立功能的程序关于某个数据集合的一次执行过程，也是操作系统进行资源分配和保护的基本单位。简单的说，进程就是一个程序的一次执行过程。

### 2. 引入线程的动机和思路

操作系统采用进程机制使得多任务能够并发执行，提高了资源使用和系统效率。在早期操作系统中，进程是系统进行资源分配的基本单位，也是处理器调度的基本单位，进程在任一时刻只有一个执行控制流，这种结构称为单线程进程。单线程进程调度时存在进程时空开销大、进程通信代价大、进程并发粒度粗、不适合于并发计算等问题，操作系统引入线程机制来解决这些问题。线程机制的基本思路是，把进程的两项功能——独立分配资源和被调度分派执行分离开来，后一项任务交给线程实体完成。这样，进程作为系统资源分配与保护的独立单位，不需要频繁切换；线程作为系统调度和分派的基本单位会被频繁的调度和切换。

### 3. 线程定义

线程是操作系统进程中能够独立执行的实体，是处理器调度和分派的基本单位。线程是进程的组成部分，每个进程内允许包含多个并发执行的线程。同一个进程中所有的线程共享进程的主存空间和资源，但是不拥有资源。

**线程就是进程中的一个负责程序执行的一个控制单元（执行路径）。一个进程中可以有多个执行路径，称之为多线程。**

### 4. 进程和线程的区别

- 定义方面：进程是程序在某个数据集合上的一次执行过程；线程是进程中的一个执行路径。
- 角色方面：在支持线程机制的系统中，进程是系统资源分配的单位，线程是系统调度的单位。
- 资源共享方面：进程之间不能共享资源，而线程共享所在进程的地址空间和其它资源。同时线程还有自己的栈和栈指针，程序计数器等寄存器。

- 独立性方面：进程有自己独立的地址空间，而线程没有，线程必须依赖于进程而存在。

## 多线程的实现

在Windows中启动一个JVM虚拟机相当于创建一个进程。

`Thread.currentThread().getName()` 获取当前进程的名字。

实现多线程编程的方式主要有两种，一种是继承Thread类，另一种是实现Runnable接口

```
public class Thread implements Runnable
```

执行main () 方法的为main线程

从jdk源码可以发现，Thread类实现了Runnable接口，他们之间是多态关系。其实，使用继承Thread类的方式创建新线程时，最大的局限就是不支持多继承，因为java语言的特性就是单根继承，所以为了多继承，完全可以实现Runnable接口的方式，一边实现一边继承。但是这两种方式创建的线程在工作时的性质是一样的，没有本质区别。

### 1.继承Thread类

- (1) 定义一个类继承Thread类。
- (2) 覆盖Thread类中的run方法。（方法run称为线程体）
- (3) **直接创建Thread类**的子类对象创建线程。
- (4) 调用start方法，开启线程并调用线程的任务run方法执行。

注意：run()方法和start()方法的区别。start()方法来启动线程（start方法通知“线程规划器”此线程已经准备就绪），run()方法当作普通方法的方式调用,程序还是顺序执行。

### 2.实现Runnable接口

- (1) 定义类实现Runnable接口
- (2) 覆盖接口中的run方法，将线程的任务代码封装到run方法中。
- (3) 通过**Thread类**创建线程对象，并将Runnable接口的子类对象作为Thread构造函数的参数进行传递。线程的任务都封装在Runnable接口子类对象的run方法中。所以要在线程对象创建时就必须明确要运行的任务。
- (4) 调用线程对象的start方法开启线程。

使用Runnable接口实现多线程的优点：

因为java是单根继承，不支持多继承。所以public class Bserver extends Aserver,Thread{}就会报错，而这时就有使用Runnable接口的必要了。public class Bserver extends Aserver **implements Runnable**{}而Thread.java类也实现了Runnable接口，所以不进可以传入Runnable接口的对象也可以传入Thread类的对象。

## 线程的状态

在Java当中，线程通常都有五种状态，创建、就绪、运行、阻塞和死亡。

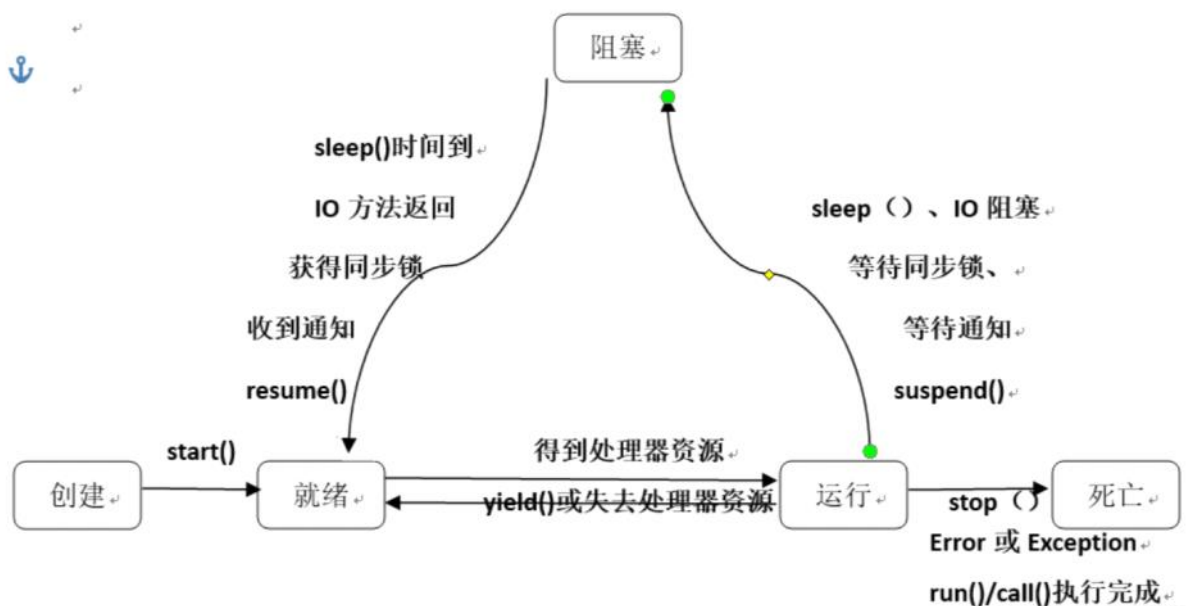
第一是创建状态。在生成线程对象，并没有调用该对象的start方法，这是线程处于创建状态。

第二是就绪状态。当调用了线程对象的start方法之后，该线程就进入了就绪状态，但是此时线程调度程序还没有把该线程设置为当前线程，此时处于就绪状态。在线程运行之后，从等待或者睡眠中回来之后，也会处于就绪状态。

第三是运行状态。线程调度程序将处于就绪状态的线程设置为当前线程，此时线程就进入了运行状态，开始运行run函数当中的代码。

第四是阻塞状态。线程正在运行的时候，被暂停，通常是为了等待某个时间的发生(比如说某项资源就绪)之后再继续运行。sleep,suspend,wait等方法都可以导致线程阻塞。

第五是死亡状态。如果一个线程的run方法执行结束或者调用stop方法后，该线程就会死亡。对于已经死亡的线程，无法再使用start方法令其进入就绪。



线程从阻塞状态只能进入就绪状态，无法直接进入运行状态。而就绪状态和运行状态之间的转换不受程序控制，而是由系统调度所决定，但是有一个方法例外，调用yield（）方法可以让**运行状态的线程转入就绪状态**。【参考自《Java疯狂讲义》】

解决非线程安全问题可以使用synchronized关键字

执行run()和start()是有一些区别的：

1. my.run(): 立即执行run方法，不启动新的线程
2. my.start(): 执行run方法时机不确定，启动新的线程。

常用API

1、currentThread()方法：返回代码段正在被哪个线程调用。在书中强调了Thread.currentThread()和this的差异，可以参考博客Thread.currentThread()和this的差异。

this.getName()代表类对象的name名称。始终保持一致。

Thread.currentThread().getName()返回的是当前代码段被哪个线程所调用。

```
public class currentThread extends Thread{
    public currentThread(){
        System.out.println("构造方法的打印: "+Thread.currentThread().getName());
    }

    @Override
    public void run() {
        System.out.println("run方法的打印: "+Thread.currentThread().getName());
    }

    public static void main(String[] args) {
        currentThread currentThread = new currentThread();
        currentThread.start();
    }
}
```

构造方法的打印: main

run方法的打印: Thread-0

结果可以发现，类的构造函数是被main线程调用的，而类中的run方法是被Thread-0调用的。



```

public class currentThread2 extends Thread {
    currentThread2(){
        System.out.println("构造函数开始");
        System.out.println("Thread.currentThread().getName()="+Thread.currentThread().getName());
        System.out.println("this.getName()="+ this.getName());
        System.out.println("构造函数结束");
    }

    @Override
    public void run() {
        System.out.println("run函数开始");
        System.out.println("Thread.currentThread().getName()="+Thread.currentThread().getName());
        System.out.println("this.getName()="+ this.getName());
        System.out.println("run函数结束");
    }

    public static void main(String[] args) {
        currentThread2 currentThread2 = new currentThread2();
        Thread t1 = new Thread(currentThread2);
        t1.setName("A");
        t1.start();
    }
}

```

```

构造函数开始
Thread.currentThread().getName()=main
this.getName()=Thread-0
构造函数结束
run函数开始
Thread.currentThread().getName()=A
this.getName()=Thread-0
run函数结束

```

t1.setName改变的是线程的名字，但是this.getName()获取的是类对象的name名称，名称一直未改，默认为Thread-0

2、isAlive()方法：判断当前线程是否处于活动状态。活动状态就是线程已经**启动**且尚未终止。线程处于正在运行或准备开始运行的状态。

3、sleep(long millis)方法：在指定的毫秒数内让当前“正在执行的线程”休眠（暂停执行）。这个正在执行的线程是指this.currentThread()返回的线程。而与当前代码段中创建的其他进程无关。还有sleep(long millis,int nanos)

不管调用顺序，只要interrupt方法和sleep方法碰到一起就会出现异常。

(1) 在sleep状态执行interrupt () 方法会出现异常；

(2) 调用interrupt()方法给线程打了中断的标记，再执行sleep方法也会发生异常。

4、getId()方法：取得线程的唯一标识。

5、yield()方法：放弃当前的CPU资源，将它让给其他的任务去占用CPU执

行时间。但是放弃时间不确定，有可能刚刚放弃，马上有获得CPU时间片。

```
public class YieldMethod extends Thread{
    @Override
    public void run() {
        long beginTime = System.currentTimeMillis();
        int count = 0;
        for(int i = 0; i < 50000000; i++){
            Thread.yield();
            count = count + (i+1);
        }

        long endTime = System.currentTimeMillis();
        System.out.println("用时" + (endTime - beginTime) + "毫秒");
    }

    public static void main(String[] args) {
        YieldMethod yieldMethod = new YieldMethod();
        yieldMethod.start();
    }
}
```

用时30372毫秒

如果注释掉Thread.yield()则

用时17毫秒

6、StackTraceElement[] getStackTrace()方法的作用返回一个表示**该线程堆栈跟踪元素数组**。如果该线程尚未启动或者已经终止，则返回零长度数组。如果返回的不是零长度的，则第一个元素表示堆栈顶，是数组中最新的方法的调用。最后一个元素表示栈底，是该数组中最旧的方法调用。

## 停止线程

在java中有以下三种方法可以终止正在运行的线程。

- (1) 使用退出标记，使线程正常退出，也就是当run方法完成之后线程终止。
- (2) 使用stop方法**强行终止线程**，但是不推荐使用这种方法，因为stop和suspend及resume一样，都是作废过期的方法，使用它们可能产生不可预料的结果。
- (3) 使用interrupt方法中断线程。需要注意的是interrupt方法仅仅是在当前线程中打了一个停止的标记，并不是真正的停止线程，需要与标记一起使用来停止线程。

在介绍如何停止线程之前，先看看Thread中提供的两种用于判断线程的状态

是不是停止的两种方法interrupted和isInterrupted方法。

#jdk1.8源码

```
public static boolean interrupted() {  
    return currentThread().isInterrupted(true);  
}
```

```
public boolean isInterrupted() {  
    return isInterrupted(false); //其中方法里面的变量是ClearnInterrupted, 即是否清除中  
断标志  
}
```

interrupted(): 测试当前线程是否已经是中断状态, 执行之后具有将状态标志置清除为false的功能。

isInterrupted(): 测试线程Thread对象是否已经是中断状态, 但不清除标志。

具体实现停止线程的操作可参考博客[停止线程的三种方法](#), 里面对使用interrupt方法中断线程做了详细说明。

两个过期的方法: suspend()和resume(): 非常容易造成公共同步对象被占用。也容易出现线程暂停, 进而数据不完整的情况。

想要对线程进行暂停和恢复的处理, 可使用wait(), notify () 或notifyAll()方法。

## 线程的优先级

在操作系统中线程可以划分优先级, 优先级较高的线程得到CPU资源较多, 其实就是让高优先级的线程获得更多CPU时间片。

setPriority(int priority)

线程优先级的继承特性: A线程启动B线程, 则B线程的优先级与A线程是一样的。

```

public static void main(String[] args) {
    for(int i = 0;i<5;i++){
        MyThread1 thread1 = new MyThread1();
        thread1.setPriority(10);
        thread1.start();

        MyThread2 thread2 = new MyThread2();
        thread2.setPriority(1);
        thread2.start();
    }
}

```

```

*****thread1 use time2251
*****thread1 use time2300
*****thread1 use time2928
*****thread1 use time3135
*****thread1 use time3165
@@@@@@@@ thread2 use time3722
@@@@@@@@ thread2 use time3822
@@@@@@@@ thread2 use time4015
@@@@@@@@ thread2 use time4016
@@@@@@@@ thread2 use time4028

```

高优先级的线程总是大部分先执行完，但不代表高优先级的线程全部先执行完。

当线程优先级的等级差距很大时，谁先执行完和代码调用顺序无关。

优先级较高的线程不一定每次都先执行完，因为优先级还具有“随机性”。（比如分别设置为5,6）优先级较高的不一定每一次都想执行完run()方法中的任务，也就是线程优先级与输出顺序无关，这两者没有依赖关系，具有不确定性和随机性。

优先级对线程运行速度的影响：优先级高的运行得快。

## 守护线程

在java中有两种线程，一种是用户线程，另一种是守护线程。

守护线程是一种特殊的线程，当进程中不存在非守护线程则守护线程自动销毁。典型的守护线程就是垃圾回收线程，当线程中没有非守护线程了，则垃圾回收线程也就没有存在的必要了，自动销毁。

该方法必须在启动线程前调用。守护线程和其他的线程城在开始和运行都是一样的，轮流抢占cpu的执行权，结束时不同。正常线程都需要手动结束，对于后台线程，如果所有的前台线程都结束了，后台线程无论处于什么状态都自动结束。



凡是调用setDaemon(true)代码并且传入true的值的线程才是守护线程。

```
public class Saemon extends Thread {  
  
    private int i = 0;  
  
    @Override  
    public void run() {  
        try{  
            while(true){  
                i++;  
                System.out.println("i="+i);  
                Thread.sleep( millis: 1000);  
            }  
        }catch (InterruptedException e){  
            e.printStackTrace();  
        }  
    }  
  
    public static void main(String[] args) {  
        try{  
            Saemon saemon = new Saemon();  
            saemon.setDaemon(true); // 设置为守护线程且必须在start之前设置  
            saemon.start();  
            Thread.sleep( millis: 5000);  
            System.out.println("我离开thread对象也不再打印了，停止了");  
        }catch (InterruptedException e){  
            e.printStackTrace();  
        }  
    }  
}
```

```
i=1  
i=2  
i=3  
i=4  
i=5  
我离开thread对象也不再打印了，停止了
```

## 第2章 对象及变量的并发访问

2020年12月16日 14:09

本章主要介绍Java多线程中的同步，如何使用**synchronized**关键字实现同步，以及**volatile**关键字的主要作用，**volatile**与**synchronized**的区别及使用情况的说明。下面是对这部分内容的理解、总结及一些扩展。

### 2.1 synchronized同步方法

#### 线程安全与非线程安全

关键词**synchronized**用来保障原子性，可见性和有序性。

非线程安全会在多个线程对同一个对象中的实例变量进行并发访问时发生，产生的后果就是脏读，也就是取到的数据其实是被更改过的。而“线程安全”就是以获得的实例变量的值是经过同步处理，不会出现脏读的现象。

非线程安全问题存在于实例变量，对于方法内部的私有变量，则不存在非线程安全问题。结果是线程安全的。这是因为**方法内部的变量具有私有特性**。

线程安全问题产生的原因：1. 多个线程在操作共享的数据。2. 操作共享数据的线程代码有多条。当一个线程在执行操作共享数据的多条代码过程中，其他线程参与了运算。就会导致线程安全问题的产生。

解决思路就是将多条操作共享数据的线程代码封装起来，当有线程在执行这些代码的时候，其他线程不可以参与运算。必须要当前线程把这些代码都执行完毕之后，其他线程才可以参与运算。

在java中同步代码块就可以解决这个问题。同步代码块的格式：

```
1 synchronized(对象){
2     需要被同步的代码;
3 }
```

象如同锁即（监视器），持有锁的线程可以在同步中执行。没有持有锁的线程即使获得cpu的执行权，也进不去，因为没有获取锁。

以去银行存钱的例子来说明这个问题：

```
/*
需求:两个储户，每个都到银行存钱每次存100，，共存三次。
*/

class Bank {
    private int sum;
    public void add(int num) {
        sum = sum + num;
        try{Thread.sleep(10);}catch(InterruptedException e){}
        System.out.println("sum="+sum);
    }
}
```

```

class Cus implements Runnable {
    private Bank b = new Bank();
    public void run() {
        for(int x=0; x<3; x++) {
            b.add(100);
        }
    }
}

public class BankDemo{
    public static void main(String[] args) {
        Cus c = new Cus();
        Thread t1 = new Thread(c);
        Thread t2 = new Thread(c);
        t1.start();
        t2.start();
    }
}

```

#输出结果为:

```

sum=200
sum=200
sum=400
sum=400
sum=600
sum=600

```

出现这个结果的原因就是多个线程对共享变量sum进行操作的原因，可以对代码进行修改实现同步

#同步代码块

```

class Bank {
    private int sum;
    Object object=new Object();
    public void add(int num) {
        synchronized (object){
            sum = sum + num;
            try{Thread.sleep(10);}catch(InterruptedException e){}
            System.out.println("sum="+sum);
        }
    }
}

```

#输出为

```

sum=100
sum=200
sum=300
sum=400
sum=500
sum=600

```

方法体也是封装代码块，也可以在方法上添加synchronized关键字来实现同步操作，即使用同步方法来实现：

#同步方法

```

class Bank {
    private int sum;
    public synchronized void add(int num) {
        sum = sum + num;
    }
}

```

```

        try{Thread.sleep(10);}catch(InterruptedException e){}
        System.out.println("sum="+sum);
    }
}

```

#输出为

```

sum=100
sum=200
sum=300
sum=400
sum=500
sum=600

```

结论：两个线程同时访问同一个对象中的同步方法时一定是线程安全的。不管哪个线程先运行，这个线程进入用synchronized声明的方法时就上锁，方法执行完后自动解锁，之后下一个线程才会进入用synchronized声明的方法里。

同步synchronized在字节码指令中的原理：

在方法中使用synchronized关键字实现同步的原因是使用了flag标记**ACC\_SYNCHRONIZED**，在调用方法时，调用指令会检查方法的ACC\_SYNCHRONIZED访问标志是否设置了，如果设置了，执行线程先持有同步锁，然后执行方法，最后在方法完成时释放锁。

这里需要思考一个问题：同步代码块的监视器（锁）可以是任意对象，那同步方法的监视器是什么？

同步方法和同步代码块

同步函数使用的锁是this，可通过一下代码来验证。

```

class Ticket implements Runnable {
    private int num = 100;
    boolean flag = true;
    public void run() {
        if(flag)
            while(true) {
                synchronized(this) {
                    if(num>0) {
                        try{Thread.sleep(10);}catch (InterruptedException e){}
                        System.out.println(Thread.currentThread().getName()+".....obj...."+num--);
                    }
                }
            }
        else
            while(true)
                this.show();
    }
    public synchronized void show() {
        if(num>0) {
            try{Thread.sleep(10);}catch (InterruptedException e){}
            System.out.println(Thread.currentThread().getName()+".....function...."+num--);
        }
    }
}

```

```

class SynFunctionLockDemo {
    public static void main(String[] args) {

```

```

Ticket t = new Ticket();
Thread t1 = new Thread(t);
Thread t2 = new Thread(t);
t1.start();
try{Thread.sleep(10);}catch(InterruptedException e){}
t.flag = false;
t2.start();
}
}

```

代码通过两个线程来卖一百张票，设置flag标志来切换同步代码块和同步函数来卖票，最终实现卖票的数据同步，说明同步函数的锁就是this。如果卖票出现数据不同步问题就说明同步代码块和同步函数不是同一个监视器。

### 同步函数与同步代码块的区别：

同步函数的锁是固定的this，同步代码块的锁是任意的对象。建议使用同步代码块。同步函数是同步代码块的简写形式，简写需要满足一定的条件：当同步代码块的锁是this的时候可以简写为同步函数，但是锁不是this的时候不能简写。

### 多个对象多个锁

```

public class Run {
    public static void main(String[] args) {
        hasSelfPrivateNum numRef1 = new hasSelfPrivateNum();//对象1
        hasSelfPrivateNum numRef2 = new hasSelfPrivateNum();//对象2

        ThreadA threadA = new ThreadA(numRef1);//线程1
        threadA.start();
        ThreadB threadB = new ThreadB(numRef2);//线程2 两个线程用的两个不同的对象，不存在争抢关系。
        threadB.start();
    }
}

```

```

public class hasSelfPrivateNum {
    private int num = 0;
    synchronized public void addI(String username){
        try {
            if (username.equals("a")) {
                num = 100;
                System.out.println("a set over!");
                Thread.sleep(1000);
            } else {
                num = 200;
                System.out.println("b set over");
            }
            System.out.println(username+" num= "+num);
        } catch (InterruptedException e){
            e.printStackTrace();
        }
    }
}

```



```

public class ThreadA extends Thread{
    private hasSelfPrivateNum numRef;

    public ThreadA(hasSelfPrivateNum numRef){
        super();
        this.numRef = numRef;
    }

    @Override
    public void run() {
        super.run();
        numRef.addI( username: "a");
    }
}

```

```

public class ThreadB extends Thread{
    private hasSelfPrivateNum numRef;

    public ThreadB(hasSelfPrivateNum numRef){
        super();
        this.numRef = numRef;
    }

    @Override
    public void run() {
        super.run();
        numRef.addI( username: "b");
    }
}

```

打印结果显示：并不是同步执行，而是异步。否则应该是a set over! a num = 100 b set over! b num = 200

本示例创建了两个业务对象，在系统中产生**两个锁**，每个线程执行**自己所属业务对象中的同步方法**，不存在争抢关系，所以运行结果是异步的，另外，在这种情况下，synchronized可以不需要，因为不会出现非线程安全问题。

关键词synchronized取得的锁都是对象锁，而不是把一段代码或者方法当做锁。所以在上面的示例中，哪个线程先执行带synchronized关键字的方法，哪个线程就持有该方法所属对象的锁Lock。其他线程只能处于等待状态。前提是多个线程访问的是同一个对象。当如果**多个线程访问多个对象**，也就是每个线程访问自己所属的业务对象（如上面的例子），则JVM会创建多个锁，不存在争抢锁的情况。

```

a set over!
b set over
b num= 200
a num= 100

```

将对象作为锁

```

public class Run {
    public static void main(String[] args) {
        MyObject object = new MyObject();
        ThreadA threadA = new ThreadA(object);
        threadA.setName("A");
        ThreadB threadB = new ThreadB(object);
        threadB.setName("B");

        threadA.start();
        threadB.start();
    }
}

```

```

public class MyObject {
    public void methodA(){
        try{
            System.out.println("begin methodA threadName = "+Thread.currentThread().getName());
            Thread.sleep(5000);
            System.out.println("end");
        }catch (InterruptedException e){
            e.printStackTrace();
        }
    }
}

```

```

public class ThreadA extends Thread{
    private MyObject object;

    public ThreadA(MyObject object){
        super();
        this.object = object;
    }

    @Override
    public void run() {
        super.run();
        object.methodA();
    }
}

```

```

public class ThreadB extends Thread{
    private MyObject object;

    public ThreadB(MyObject object){
        super();
        this.object = object;
    }

    @Override
    public void run() {
        super.run();
        object.methodA();
    }
}

```

因为methodA没有同步化。所以输出是

```

begin methodA threadName = B
begin methodA threadName = A
end
end

```

如果加上synchronized关键字,

```
begin methodA threadName = A
end
begin methodA threadName = B
end
```

调用关键字synchronized声明的方法一定是排队进行运行的。另外，需要牢牢记住“共享”这两个字，只有共享资源的读写访问才需要同步化，如果不是共享资源就没有同步的必要。

#### 演示两个线程访问同一个对象的两个同步方法

```
public class Run {

    public static void main(String[] args) {
        MyObject object = new MyObject();
        ThreadA threadA = new ThreadA(object);
        threadA.setName("A");
        ThreadB threadB = new ThreadB(object);
        threadB.setName("B");
        threadA.start();
        threadB.start();
    }
}
```

```
public class MyObject {

    synchronized public void methodA(){
        try{
            System.out.println("begin methodA threadName = "+Thread.currentThread().getName());
            Thread.sleep( millis: 5000);
            System.out.println("endTime=" + System.currentTimeMillis());
        }catch (InterruptedException e){
            e.printStackTrace();
        }
    }

    synchronized public void methodB(){
        try{
            System.out.println("begin methodB threadName = "+Thread.currentThread().getName()+
            " begin time =" + System.currentTimeMillis());
            Thread.sleep( millis: 5000);
            System.out.println("endTime=" + System.currentTimeMillis());
        }catch (InterruptedException e){
            e.printStackTrace();
        }
    }
}
```

```
public class ThreadA extends Thread{
    private MyObject object;

    public ThreadA(MyObject object){
        super();
        this.object = object;
    }

    @Override
    public void run() {
        super.run();
        object.methodA();
    }
}
```

```

public class ThreadB extends Thread{
    private MyObject object;

    public ThreadB(MyObject object){
        super();
        this.object = object;
    }

    @Override
    public void run() {
        super.run();
        object.methodB();
    }
}

```

结论如下:

1. A 线程先持有object对象的Lock锁, B线程可以以异步的方式调用object对象中的非synchronized类型的方法。
2. A线程先持有object对象的Lock锁, B线程如果在这时调用object对象中的synchronized类型的方法, 则需要等待, 也就是同步。
3. 在方法声明处添加synchronized并不是锁方法, 而是锁当前类的对象。
4. 在java中只有“将对象作为锁”这种说法, 没有“锁方法”这种说法。
5. 在java中, “锁”就是“对象”, “对象”可以映射“锁”, 哪个线程拿到这把锁, 哪个线程就可以执行这个对象中的synchronized同步方法。
6. 如果在X对象中使用了synchronized关键字声明非静态方法, 则X对象就被当成锁。

## 脏读

在多个线程调用同一个方法时为了避免数据出现交叉的情况, 使用synchronized关键词进行同步。在赋值的时候进行了同步, 在取值的时候也有可能出状况——脏读。原因是读取实例变量时, 此值已经被其他线程更改过了

出现脏读的原因是public void getValue()方法并不是同步的, 所以可以在任意时刻进行调用, 解决办法是加上同步synchronized关键字。

```

synchronized public void getValue(){
    System.out.println("getValue method thread name="
        + Thread.currentThread().getName()
        + " username= " + username + " password= " + password);
}

```

脏读是可以通过synchronized关键字解决的。

当A调用synchronized修饰的X方法时, A获得了X方法所在对象的锁。B线程可以随意

调用非synchronized同步方法。但如果调用synchronized关键字的非X方法，必须等A线程将X方法执行完，也就是将对象锁释放后才可以调用。

脏读前一定会出现不同线程一起去写实例变量的情况，这就是不同线程“争抢”实例变量的结果。

## synchronized锁重入

重入锁的功能，即在使用synchronized时，当一个线程得到一个对象锁后，再次请求此对象锁时是可以得到该对象锁的，这也证明在一个synchronized方法/块的内部调用本类的其他synchronized方法/块时，是永远可以得到的。

```
public class Service {  
    synchronized public void service1(){  
        System.out.println("service1");  
        service2();  
    }  
  
    synchronized public void service2() {  
        System.out.println("service2");  
        service3();  
    }  
  
    synchronized public void service3() {  
        System.out.println("service3");  
    }  
}
```

```
public class MyThread extends Thread {  
    @Override  
    public void run() {  
        Service service = new Service();  
        service.service1();  
    }  
}
```

```
public class Run {  
    public static void main(String[] args) {  
        MyThread myThread = new MyThread();  
        myThread.start();  
    }  
}
```

“可重入锁”是指自己可以再次获得自己的内部锁。一个线程获取了某个对象锁，此时这个对象锁还没有释放，当其再次想要获取这个对象锁时还可以获取，否则service2 () 和service () 不会被调用。

## 锁重入支持继承的环境

当存在父子类继承关系时，子类是完全可以通过锁重入调用父类的同步方法的。



## 出现异常，锁自动释放

当一个线程执行的代码出现异常的时候，其所持有的锁都会自动释放。当线程A出现异常并释放锁，线程B就可以正常进入方法。

要注意：suspend()方法和sleep(millis)方法被调用后并不释放锁。

## 重写方法不使用synchronized

重写方法不使用synchronized就不是同步的，加上synchronized就是同步的。（不会重写父类的方法后自动带synchronized功能）

## public static boolean holdsLock(Object object)方法的使用

作用是当currentThread在指定的对象上**保持锁定时**，才返回true。

```
public class Test1 {
    public static void main(String[] args) {
        System.out.println("A "+Thread.currentThread().holdsLock(Test1.class));//false
        synchronized (Test1.class){
            System.out.println("B "+Thread.currentThread().holdsLock(Test1.class));//true
        }
        System.out.println("C "+Thread.currentThread().holdsLock(Test1.class));//false
    }
}
```

## 2.2 synchronized同步语句块

### synchronized方法的弊端

A线程调用同步方法执行一个长时间的任务，B等待时间较长，可以用synchronized同步块语句来解决，以提高运行效率。

synchronized方法是将**当前对象**作为锁，而synchronized代码块是将任意对象作为锁。可以将锁看成一个标识，哪个线程持有这个标识，就可以执行同步方法。

### 一半异步，一半同步

当一个线程访问object的一个synchronized同步代码块时，另一个线程仍然可以访问该object对象中的非synchronized(this)同步代码块。加快运行效率。

不在synchronized块中就是异步执行，在synchronized块中就是同步执行。

### synchronized代码块间的同步性

当一个线程访问object的一个synchronized (this) 同步代码块时，其他线程对同一个object中所有其他synchronized(this)同步代码块的访问将被阻塞，这说明

synchronized使用的对象监视器是同一个，即使用的锁是同一个。

### **println()方法也是同步的**

PrintStream.java 类中的println()重载方法如下：

```
public void println(String x){
    synchronized(this){
        print(x);
        newLine();
    }
    public void println(Object s){
        synchronized(this){
            print(s);
            newLine();
        }
    }
```

说明执行的方式是顺序同步执行的，这样输出的数据是完整的，不会出现信息交叉混乱的情况。

### **synchronized(this)代码块是锁定当前对象的**

和synchronized方法一样。

### **将任意对象作为锁**

多个线程调用同一个对象中的不同名称的synchronized同步方法或synchronized(this)同步代码块时，调用的效果是按顺序执行，即同步。

java还支持将“任意对象”作为锁来实现同步的功能，这个“任意对象”大多数是实例变量及方法的参数。使用格式为synchronized(非this对象)，同样地，同一时间只有一个线程可以执行。

举例：

```
public class Run {
    public static void main(String[] args) {
        Service service = new Service();
        ThreadA threadA = new ThreadA(service);
        threadA.setName("A");
        ThreadB threadB = new ThreadB(service);
        threadB.setName("B");

        threadA.start();
        threadB.start();
    }
}
```

```

public class Service {
    private String username;
    private String password;

    private String anything = new String();
    public void setUsernamePassword(String username,String password){
        try{
            synchronized (anything){
                System.out.println("线程名称为 "+Thread.currentThread().getName()
                    +" 在 "+System.currentTimeMillis()+" 进入同步块");
                this.username = username;
                Thread.sleep( millis: 3000);
                this.password = password;
                System.out.println("线程名称为 "+Thread.currentThread().getName()
                    +" 在 "+System.currentTimeMillis()+" 离开同步块");
            }
        }catch (InterruptedException e){
            e.printStackTrace();
        }
    }
}

```

```

public class ThreadA extends Thread{
    private Service service;

    ThreadA(Service service){
        super();
        this.service = service;
    }

    public void run(){
        service.setUsernamePassword( username: "a", password: "aa");
    }
}

```

```

public class ThreadB extends Thread{
    private Service service;

    ThreadB(Service service){
        super();
        this.service = service;
    }

    public void run(){
        service.setUsernamePassword( username: "b", password: "bb");
    }
}

```

结果:

```

线程名称为 B 在 1608111655852 进入同步块
线程名称为 B 在 1608111658853 离开同步块
线程名称为 A 在 1608111658853 进入同步块
线程名称为 A 在 1608111661854 离开同步块

```

锁非this对象具有一定的优点：如果一个类中有很多个synchronized方法，则这时虽然能实现同步，但影响运行效率。如果使用同步代码块锁非this对象，则synchronized(非 this)代码块中的程序与同步方法是异步的，因为有两把锁，不与其他锁this同步方法争抢this锁，可以提高运行效率。

### 多个锁就是异步执行

把上面的Service.java类修改如下，anything此时是一个局部变量。

```

public void setUsernamePassword(String username,String password){
    try{
        String anything = new String();
        synchronized (anything){
            System.out.println("线程名称为 "+Thread.currentThread().getName()
                +" 在 "+System.currentTimeMillis()+" 进入同步块");
            this.username = username;
            Thread.sleep( millis: 3000);
            this.password = password;
            System.out.println("线程名称为 "+Thread.currentThread().getName()
                +" 在 "+System.currentTimeMillis()+" 离开同步块");
        }
    }catch (InterruptedException e){
        e.printStackTrace();
    }
}

```

运行结果是：

```

线程名称为 A 在 1608112226460 进入同步块
线程名称为 B 在 1608112226460 进入同步块
线程名称为 B 在 1608112229461 离开同步块
线程名称为 A 在 1608112229461 离开同步块

```

synchronized (非this对象x) 同步代码块格式进行同步操作时，锁必须是同一个，如果不是同一个锁，则运行结果就是异步条用，交叉运行。

### 方法调用是随机的

同步代码块放在非同步synchronized方法中进行声明，并不能保证调用方法的线程的执行同步（顺序性）也就是线程调用方法的顺序是无序的，虽然在同步块中执行的顺序是同步的。

### 不同步导致的逻辑错误

```

public class Run {
    public static void main(String[] args) throws InterruptedException {
        MyOneList list = new MyOneList();
        MyThread1 thread1 = new MyThread1(list);
        thread1.setName("A");
        thread1.start();

        MyThread2 thread2 = new MyThread2(list);
        thread2.setName("B");
        thread2.start();

        Thread.sleep( millis: 6000);

        System.out.println("ListSize= "+ list.getSize());
    }
}

```

```

public class MyThread2 extends Thread {
    private MyOneList list;

    public MyThread2(MyOneList list){
        super();
        this.list = list;
    }

    @Override
    public void run() {
        MyService myService = new MyService();
        myService.addServiceMethod(list, data: "B");
    }
}

```

```

public class MyThread1 extends Thread {
    private MyOneList list;

    public MyThread1(MyOneList list){
        super();
        this.list = list;
    }

    @Override
    public void run() {
        MyService myService = new MyService();
        myService.addServiceMethod(list, data: "A");
    }
}

```

```

public class MyService {
    public MyOneList addServiceMethod(MyOneList list,String data){
        try{
            if(list.getSize() < 1){ //小于1才会添加数据，但是结果list中有两个元素
                Thread.sleep( millis: 3000); //模拟从远程端花费3s取回数据
                list.add(data);
            }
        }catch (InterruptedException e){
            e.printStackTrace();
        }
        return list;
    }
}

```

```

public class MyOneList {
    private List list = new ArrayList();
    synchronized public void add(String data){
        list.add(data);
    };

    synchronized public int getSize(){
        return list.size();
    }
}

```

最后打印的list的大小为2，但是代码中要求只有list为空的时候才插入数据。出错的原因是两个线程以异步的方式返回list参数的size () 大小。

同步化：

```

public class MyService {
    public MyOneList addServiceMethod(MyOneList list,String data){
        try{
            synchronized (list){
                if(list.getSize() < 1){ //当对list加锁后，list中最多只能有一个元素
                    Thread.sleep( millis: 3000); //模拟从远程端花费3s取回数据
                    list.add(data);
                }
            }
        }catch (InterruptedException e){
            e.printStackTrace();
        }
        return list;
    }
}

```

由于list参数对象在项目中是一份实例，是单例的，也正需要对list参数的getSize()方法做同步的调用，所以就对list参数进行同步处理。



## 细化三个结论

1. 当多个线程同时执行synchronized(x){}同步代码块时呈同步效果。
2. 当其他线程执行x对象中synchronized同步方法时呈同步效果。
3. 当其他线程执行x对象方法里面的synchronized(this)代码块时呈现同步效果。

如果其他线程调用不加synchronized关键字的方法，则还是异步调用。

## 静态函数的锁

静态函数的锁绝对不是this，经过验证可以知道静态函数的锁是字节码文件对象。静态的同步函数使用的锁是该函数所属的字节码文件对象 可以用getClass方法获取（这里要注意getClass方法是非静态的，使用要谨慎），也可以用当前类名.class（class是静态属性）形式表示。

## 类Class的单例性

每一个\*.java 文件对应Class类的实例都是一个，在内存中是单例的。Class类用于描述类的基本信息，包括多少个字段，多少个构造方法，有多少个普通方法，为了减少对内存的高占用率，在内存中只需要存在一份Class类对象就可以了。

## 静态同步synchronized方法与synchronized(class)代码块

关键字synchronized还可以应用在static静态方法上，如果这样写，那就是对当前\*.java文件对应的Class类对象进行持锁。Class类的对象是单例的，在静态static方法上使用synchronized关键字声明同步方法时，使用当前静态方法所在类对应Class类的单例对象作为锁。

验证synchronized关键字加到非static方法和static方法是两种不同的锁：

```
public class Run {
    public static void main(String[] args) {
        Service service = new Service();
        ThreadA a = new ThreadA(service);
        a.setName("A");
        a.start();

        ThreadB b = new ThreadB(service);
        b.setName("B");
        b.start();

        ThreadC c = new ThreadC(service);
        c.setName("C");
        c.start();
    }
}
```

```

public class Service {

    synchronized public static void printA(){
        try{
            System.out.println("线程名称是 "+Thread.currentThread().getName()
            +" 在 "+System.currentTimeMillis()+" 进入printA");
            Thread.sleep( millis: 3000);
            System.out.println("线程名称是 "+Thread.currentThread().getName()
            +" 在 "+System.currentTimeMillis()+" 离开printA");
        }catch (InterruptedException e){
            e.printStackTrace();
        }
    }

    synchronized public static void printB(){
        System.out.println("线程名称是 "+Thread.currentThread().getName()
        +" 在 "+System.currentTimeMillis()+" 进入printB");
        System.out.println("线程名称是 "+Thread.currentThread().getName()
        +" 在 "+System.currentTimeMillis()+" 离开printB");
    }

    synchronized public void printC(){
        System.out.println("线程名称是 "+Thread.currentThread().getName()
        +" 在 "+System.currentTimeMillis()+" 进入printC");
        System.out.println("线程名称是 "+Thread.currentThread().getName()
        +" 在 "+System.currentTimeMillis()+" 离开printC");
    }
}

```

```

public class ThreadA extends Thread{
    private Service service;
    public ThreadA(Service service){
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.printA();
    }
}

```

```

public class ThreadB extends Thread{
    private Service service;
    public ThreadB(Service service){
        super();
        this.service = service;
    }

    @Override
    public void run() { service.printB(); }
}

```

```

public class ThreadC extends Thread{
    private Service service;
    public ThreadC(Service service){
        super();
        this.service = service;
    }

    @Override
    public void run() { service.printC(); }
}

```

异步运行的原因是因为持有不同的锁，一个是将类Service的对象作为锁，一个是将

Service类对应Class类的对象作为锁，A、B 和C是异步关系，A和B是同步关系。

### 同步syn static方法可以对类的所有对象实例起作用

Class锁可以对类的所有对象实例起作用。虽然是不同的对象，但是静态的同步方法还是同步执行。

### 同步syn(class)代码块可以对类的所有对象实例起作用

同步synchronized (class) 代码块的作用其实和synchronized static方法的作用一样。

```
public void printA(){
    synchronized (Service.class){
        try{
            System.out.println("线程名称是 "+Thread.currentThread().getName()
                +" 在 "+System.currentTimeMillis()+" 进入printA");
            Thread.sleep(3000);
            System.out.println("线程名称是 "+Thread.currentThread().getName()
                +" 在 "+System.currentTimeMillis()+" 离开printA");
        }catch (InterruptedException e){
            e.printStackTrace();
        }
    }
}
```

### String常量池特性与同步相关的问题与解决方案

JVM具有String常量池的功能。

```
public static void main(String[] args){

    String a = "a";

    String b = "a";

    System.out.println(a==b);//true
}
```

当将synchronized (string) 同步块与String联合使用时，要注意常量池会带来一些意外。

```

public class Run {
    public static void main(String[] args) {
        Service service = new Service();
        ThreadA a = new ThreadA(service);
        a.setName("A");
        a.start();
        ThreadB b = new ThreadB(service);
        b.setName("B");
        b.start();
    }
}

```

```

public class Service {
    public static void print(String stringParam){
        try{
            synchronized (stringParam){
                while(true){
                    System.out.println(Thread.currentThread().getName());
                    Thread.sleep(1000);
                }
            }
        }catch (InterruptedException e){
            e.printStackTrace();
        }
    }
}

```

```

public class ThreadA extends Thread {
    private Service service;
    public ThreadA(Service service){
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.print("AA");
    }
}

```

```

public class ThreadB extends Thread{
    private Service service;
    public ThreadB(Service service){
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.print("AA");
    }
}

```

最后结果

```

C:\Program File
A
A
A

```

因为两个String的值都是“AA”，两个线程持有相同的锁，造成B线程不能执行。这就

是String常量池带来的问题。所以大多情况，同步synchronized代码块**不使用String作为锁对象**，而改用为其他。例如，new Object()实例化一个新的Object对象，它并不放入缓冲池中，或者执行new String()创建不同的字符串对象，形成不同的锁。如下：

```
public class ThreadA extends Thread {
    private Service service;
    public ThreadA(Service service){
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.print(new String( original: "AA"));
    }
}
```

### 同步synchronized方法无限等待问题与解决方案

使用同步方法会导致所资源被长期占用，别的线程得不到运行机会。此时引入不同的锁，用同步代码块来解决这个问题。

### 多线程的死锁

java线程死锁是一个经典的多线程问题，因为不同的线程都在等待根本不可能释放的锁，从而导致所有的任务都无法完成。造成线程“假死”。

```
public class Run {

    public static void main(String[] args) throws InterruptedException {
        DealThread dealThread = new DealThread();
        dealThread.setFlag("a");
        Thread thread1 = new Thread(dealThread);
        thread1.start();

        Thread.sleep( millis: 100);
        dealThread.setFlag("b");
        Thread thread2 = new Thread(dealThread);
        thread2.start();
    }
}
```

```
public class DealThread implements Runnable {
```

```
    public String username;
    public Object lock1 = new Object();
    public Object lock2 = new Object();

    public void setFlag(String username){
        this.username = username;
    }
    @Override
    public void run() {
        if(username.equals("a")){
            synchronized (lock1){
                try{
                    System.out.println("username = "+username);
                    Thread.sleep(3000);
                }catch (InterruptedException e){
                    e.printStackTrace();
                }
            }
            synchronized (lock2){
```



```

        System.out.println("按lock1->lock2代码顺序执行了");
    }
}
}
if(username.equals("b")){
    synchronized (lock2){
        try{
            System.out.println("username = "+username);
            Thread.sleep(3000);
        }catch (InterruptedException e){
            e.printStackTrace();
        }
        synchronized (lock1){
            System.out.println("按lock2->lock1代码顺序执行了");
        }
    }
}
}
}
}
}

```

两个线程在不放弃自己持有的锁的同时，等待另一个锁，相互等待，死锁。

用jps命令和jstack命令可以检测死锁现象。

## 内置类和静态内置类

### 简单内置类

```

public class PublicClass {
    private String username;
    private String password;
    class PrivateClass{
        private String age;
        private String address;

        public String getAge() {
            return age;
        }

        public void setAge(String age) {
            this.age = age;
        }

        public String getAddress() {
            return address;
        }

        public void setAddress(String address) {
            this.address = address;
        }
        public void printPublicProperty(){
            System.out.println(username+" "+password);
        }
    }

    public String getUsername() {
        return username;
    }
}

```

```

}

public void setUsername(String username) {
    this.username = username;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}
}

```

```

public class Run {
    public static void main(String[] args) {
        PublicClass publicClass = new PublicClass();
        publicClass.setPassword("password");
        publicClass.setUsername("username");

        System.out.println(publicClass.getUsername()+" "+publicClass.getPassword());

        PrivateClass privateClass = publicClass.new PrivateClass();
        privateClass.setAddress("address");
        privateClass.setAge("12");
        privateClass.printPublicProperty();
        System.out.println(privateClass.getAddress()+" "+privateClass.getAge());
    }
}

```

```

username password
username password
address 12

```

如果PublicClass.java类和Run.java类不再同一个包中，则需要将PrivateClass内置类声明成public的。

要实例化内置类必须：`PrivateClass privateClass = publicClass.new PrivateClass();`

### 静态内置类

需要将username和password声明称静态的，否则静态内置类无法使用非静态变量。

```

public class PublicClass {
    static private String username;
    static private String password;

    static class PrivateClass{
        private String age;
        private String address;

        public String getAge() {
            return age;
        }
    }
}

```

```

    }

    public void setAge(String age) {
        this.age = age;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public void printPublicProperty(){
        System.out.println(username+" "+password);
    }
}

public String getUsername() {
    return username;
}

public void setUsername(String username) {
    this.username = username;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}
}

```

在创建内置类对象不需要依赖外部类，直接创建即可。但是上面和下面都需要将内置类import进来。

```

public class Run {
    public static void main(String[] args) {
        PublicClass publicClass = new PublicClass();
        publicClass.setPassword("password");
        publicClass.setUsername("username");

        System.out.println(publicClass.getUsername()+"
"+publicClass.getPassword());
        //需要import Chapter2.innerStaticClass.PublicClass.PrivateClass;引入
        PrivateClass privateClass = new PrivateClass();
        privateClass.setAddress("address");
        privateClass.setAge("12");
        privateClass.printPublicProperty();
        System.out.println(privateClass.getAddress()+" "+privateClass.getAge());
    }
}

```

## 内置类与同步

在内置类中有两个同步的方法，但使用的是不同的锁，输出的结果是异步的。

同步代码块synchronized(lock)对lock上锁后，其他线程只能以同步的方式调用lock中的同步方法。

OutClass中有两个静态内部类，InnerClass1中的method1是对InnerClass2类对象上锁，InnerClass1中的method2是对InnerClass1类上锁，所以两者并不存在竞争关系，InnerClass2中的method1是同步方法，是对InnerClass2s的对象持锁（不是静态方法所以不是Class对象），所以和InnerClass1中的method1存在竞争关系。

```
public class OutClass {
    static class InnerClass1{
        public void method1(InnerClass2 class2){
            String threadName = Thread.currentThread().getName();
            synchronized (class2){
                System.out.println(threadName+"进入InnerClass1类中的method1方法");
                for(int i = 0;i<3;i++){
                    System.out.println("i="+i);
                    try{
                        Thread.sleep(100);
                    }catch (InterruptedException e){
                        e.printStackTrace();
                    }
                }
                System.out.println(threadName + "离开InnerClass1类中的method1方法");
            }
        }

        public synchronized void method2(){
            String threadName = Thread.currentThread().getName();
            System.out.println(threadName+"进入InnerClass1的method2方法");
            for(int j = 0;j<3;j++){
                System.out.println("j="+j);
                try{
                    Thread.sleep(100);
                }catch (InterruptedException e){
                    e.printStackTrace();
                }
            }
            System.out.println(threadName + "离开InnerClass1类中的method2方法");
        }
    }

    static class InnerClass2 {
        public synchronized void method1(){
            String threadName = Thread.currentThread().getName();
            System.out.println(threadName+"进入InnerClass2的method1方法");
            for(int k = 0;k<3;k++){
                System.out.println("j="+k);
                try{
                    Thread.sleep(100);
```

```

        }catch (InterruptedException e){
            e.printStackTrace();
        }
    }
    System.out.println(threadName + "离开InnerClass2类中的method1方
法");
}
}
}

```

```

public class Run {
    public static void main(String[] args) {
        final OutClass.InnerClass1 in1 = new OutClass.InnerClass1();
        final OutClass.InnerClass2 in2 = new OutClass.InnerClass2();
        Thread thread1 = new Thread(new Runnable() {
            @Override
            public void run() {
                in1.method1(in2);
            }
        }, "T1");
        Thread thread2 = new Thread(new Runnable() {
            @Override
            public void run() {
                in1.method2();
            }
        }, "T2");
        Thread thread3 = new Thread(new Runnable() {
            @Override
            public void run() {
                in2.method1();
            }
        }, "T3");

        thread1.start();
        thread2.start();
        thread3.start();
    }
}

```

```

T1进入InnerClass1类中的method1方法
T2进入InnerClass1的method2方法
j=0
i=0
j=1
i=1
j=2
i=2
T2离开InnerClass1类中的method2方法
T1离开InnerClass1类中的method1方法
T3进入InnerClass2的method1方法
j=0
j=1
j=2
T3离开InnerClass2类中的method1方法

```



如果去掉T2，效果更明显：

```
T1进入InnerClass1类中的method1方法
i=0
i=1
i=2
T1离开InnerClass1类中的method1方法
T3进入InnerClass2的method1方法
j=0
j=1
j=2
T3离开InnerClass2类中的method1方法
```

### 锁对象改变导致异步执行

通常情况下，一旦持有锁就不在对锁对象进行更改，因为一旦更改就有可能出现一些错误。

```
public class MyService {
    private String lock = "123";
    public void testMethod(){
        try{
            synchronized (lock){
                System.out.println(Thread.currentThread().getName()+" begin "+System.currentTimeMillis());
                lock = "456";//锁改变
                Thread.sleep( millis: 2000);
                System.out.println(Thread.currentThread().getName()+" end "+System.currentTimeMillis());
            }
        }catch (InterruptedException e){
            e.printStackTrace();
        }
    }
}
```

```
public class Run {
    public static void main(String[] args) throws InterruptedException {
        MyService myService = new MyService();
        ThreadA threadA = new ThreadA(myService);
        threadA.setName("A");

        ThreadB threadB = new ThreadB(myService);
        threadB.setName("B");

        threadA.start();
        Thread.sleep( millis: 50); //存在50ms
        threadB.start();
    }
}
```

运行结果：

```
A begin 1608171021488
B begin 1608171021538
A end 1608171023489
B end 1608171023539
```

这说明，50ms后，B线程获得的锁是“456”

如果去掉Thread.sleep(50)

```
A begin 1608171520993
A end 1608171522994
B begin 1608171522994
B end 1608171524995
```

说明A线程和B线程检测到锁对象的值是“123”，“123”存储到A线程的内存空间中，虽然将锁改成了“456”，“456”存储到B线程内存空间中，但结果还是同步的，因为A线程和B线程共同争抢的锁是A线程中的“123”，不是B线程中的“456”

### 锁对象不改变依然同步运行

当所的对象是UserInfo

```
synchronized (userinfo) {
    try{
        System.out.println(Thread.currentThread().getName());
        userinfo.setUsername("abcabdf");//改变对象内部属性
        Thread.sleep(3000);
        System.out.println(Thread.currentThread().getName());
    }catch(InterruptedException e){
        e.printStackTrace();
    }
}
```

只要对象不变就是同步效果，因为A线程和B线程持有的锁对象永远为同一个，仅仅对象属性改变，对象并未发生改变。

### 同步写法案例比较

```
public class MyService{

    synchronized public static void testMethod1(){

    }

    public void testMethod2(){ synchronized(MyService.class){ } }

    synchronized public void testMethod3(){

    }

    public void testMethod4(){ synchronized(this){ } }

    public void testMethod5(){ synchronized("abc") { } }

}
```

三种类型锁对象：

1. testMethod1 和testMethod2持有的锁是同一个，即MyService.java对应的

## Class类的对象

2. testMethod3和testMethod4持有的锁是同一个，即MyService.java类的对象。
3. testMethod5持有的锁是字符串abc

## volatile关键字

volatile关键字的主要作用是使变量在多个线程间可见。具体的示例可以参考书上的代码。这里需要了解java的内存模型，可以参考博客[JAVA并发编程：volatile关键字](#)以及[内存模型以及volatile关键字解析](#)。

使用volatile关键字增加了实例变量在多个线程之间的可见性。但是volatile关键字不支持原子性（在上面的博客中也有说明）。下面将关键字synchronized和volatile进行比较：

- （1）关键字volatile是线程同步的轻量级实现，所以volatile性能肯定比synchronized要好，并且volatile只能修饰于变量，而synchronized可以修饰方法，以及代码块。随着JDK新版本的发布，synchronized关键字在执行效率上得到很大提升，在开发中使用synchronized关键字的比率还是比较大的。
- （2）多线程访问volatile不会发生阻塞，而synchronized会出现阻塞。
- （3）volatile能保证数据的可见性，但不能保证原子性；而synchronized可以保证原子性，也可以间接保证可见性，因为它会将私有内存和公共内存中的数据做同步
- （4）关键字volatile解决的是变量在多个线程之间的可见性；而synchronized关键字解决的是多个线程之间访问资源的同步性。

## 第3章 线程间通信

2020年12月17日 10:54

本章的重点是使用wait/notify实现线程间的通信，生产者/消费者模式的实现，方法join的使用以及ThreadLocal类的使用。

### wait/notify机制

如果不适用，则只能用while来循环一个条件，会很浪费CPU资源，且如果轮询时间间隔大，则可能取不到想要的数。据。

### 等待/通知机制的原理

拥有相同锁的进程才可以实现wait/notify机制。

方法wait的作用是使当前执行代码的线程进行等待，wait方法是Object类的方法。该方法用来将当前线程置入“预执行队列”中，并且在wait所在的代码处停止执行，直到接到通知或被中断为止。在调用wait方法之前，线程必须获得该**对象的对象级别锁**。即只能在同步方法或同步代码块中调用wait方法。在执行wait方法后，当前线程**释放锁**。在从wait返回前，线程与其他线程竞争重新获得锁。如果调用wait时**没有持有适当的锁**，则抛出IllegalMonitorStateException，它是RuntimeException的一个子类，因此，不需要try-catch语句进行捕获异常。

方法notify也要在同步方法或同步块中调用，即在**调用前，线程也必须获得该对象的对象级别锁**。如果调用notify时没有持有适当的锁，则抛出IllegalMonitorStateException。该方法用来**通知**那些可能等待该对象的对象锁的其他线程，如果有多个线程等待，则由**线程规划器随机挑选出其中一个呈wait状态的线程**，对其发出通知notify，并使它**等待获取该对象的对象锁**。需要说明的是，在执行notify方法后，当前线程不会马上释放该对象锁，呈wait状态的线程也并不能马上获取该对象锁，**要等到执行notify方法的线程将程序执行完，也就是退出synchronized代码块之后，当前线程才会释放锁**，而呈wait状态所在的线程才可以获取该对象锁。

### 实现wait/notify机制

```

public class MyThread1 extends Thread {
    private Object lock;

    public MyThread1(Object lock){
        super();
        this.lock = lock;
    }

    @Override
    public void run() {
        try{
            synchronized (lock){
                System.out.println("开始wait "+System.currentTimeMillis());
                lock.wait();
                System.out.println("结束wait "+System.currentTimeMillis());
            }
        }catch (InterruptedException e){
            e.printStackTrace();
        }
    }
}

```

```

public class MyThread2 extends Thread {
    private Object lock;

    public MyThread2(Object lock){
        super();
        this.lock = lock;
    }

    @Override
    public void run() {
        synchronized (lock){
            System.out.println("结束notify "+ System.currentTimeMillis());
            lock.notify();
            System.out.println("结束notify " + System.currentTimeMillis());
        }
    }
}

```

```

public class Test {

    public static void main(String[] args) throws InterruptedException {
        Object lock = new Object();
        MyThread1 thread1 = new MyThread1(lock);
        thread1.start();
        Thread.sleep( millis: 2000);
        MyThread2 thread2 = new MyThread2(lock);
        thread2.start();
    }
}

```

关键字synchronized可以将任何一个Object对象作为锁来看待，而java为每个Object都实现了wait()和notify()方法，**它们必须在被synchronized同步的Object的临界区内使用**。通过调用wait方法可以使处于临界区内的线程进入等待状态，同时释放被同步对象的锁，而notify操作可以唤醒一个因调用了wait操作而处于wait状态的线程，被重新唤醒的线程会试图重新获得临界区的控制权，也就是锁。如果发出的notify操作时没有处于wait状态的线程，则命令被忽略。

一般业务代码会放在Service层

InterruptedException 线程堵塞异常

主要的抛出该异常的方法有



- 1、java.lang.Object的wait():会进入等待区等待
- 2、java.lang.Thread的sleep():会睡眠执行参数内所设置的时间。
- 3、java.lang.Thread.join():会等待到指定的线程结束为止。

因为当这些方法执行结束后，该线程会重新启动，就的可能会出现线程堵塞。

所有这些方法需要处理InterruptedException异常

### 使用wait/notify机制实现list.size()等于5时的通知

```
public class MyList {  
    volatile private List<Integer> list = new ArrayList();  
  
    public void add(int data){  
        list.add(data);  
    }  
    public int size(){  
        return list.size();  
    }  
}
```

```
public class Service {  
    private Object lock = new Object();  
    private MyList list = new MyList();  
  
    public void waitMethod(){  
        try{  
            synchronized (lock){  
                if(list.size() != 5){  
                    System.out.println("开始wait");  
                    lock.wait();  
                    System.out.println("结束wait");  
                }  
            }  
        } catch (InterruptedException e){  
            e.printStackTrace();  
        }  
    }  
  
    public void notifyMethod(){  
  
        synchronized (lock){  
            for(int i = 0;i<10;i++){  
                list.add(i);  
                if(list.size() == 5){  
                    lock.notify();  
                }  
            }  
        }  
        System.out.println("list的大小: "+list.size());  
        System.out.println(" 如果是10说明notify之后并没有立刻释放资源，等同步块完  
成");  
  
    }  
}
```

```

    }
}

public class MyThreadA extends Thread{
    private Service service;

    public MyThreadA(Service service){
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.waitMethod();
    }
}

```

```

public class MyThreadB extends Thread{
    private Service service;

    public MyThreadB(Service service){
        super();
        this.service = service;
    }

    @Override
    public void run() { service.notifyMethod(); }
}

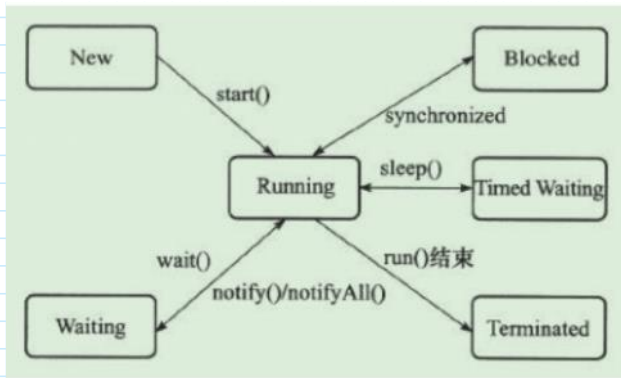
```

```

public class Run {
    public static void main(String[] args) {
        Service service = new Service();
        MyThreadA thread1 = new MyThreadA(service);
        thread1.start();
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        MyThreadB thread2 = new MyThreadB(service);
        thread2.start();
    }
}

```

## 线程状态的切换



1. 创建新的线程对象后，调用start方法，会分配CPU资源，此时线程处于runnable(可运行)状态，这是状态准备阶段，如果线程抢占到CPU资源，则处于running状态。
2. runnable和running状态可以相互转换，可能运行一段时间后，其他高优先级的线程抢占了CPU资源。
  - (1) 调用sleep()后超过了指定的休息时间
  - (2) 线程成功的获得了试图同步的监视器
  - (3) 线程正在等待某个通知，其他线程发出了通知
  - (4) 处于挂起状态的线程调用了resume恢复方法
3. block状态：是阻塞的意思，如果遇到I/O操作，此时当前线程由runnable运行状态转成blocked阻塞状态，等待I/O操作结果，此时操作系统会把宝贵的cpu时间片分配给其他线程。I/O操作结束后，再进入runnable状态。

出现阻塞的情况：

  - (1) 调用sleep方法，主动放弃占用的处理器资源
  - (2) 调用了阻塞式I/O方法
  - (3) 试图获得一个同步监视器，但该同步监视器被其他线程所持有。
  - (4) 线程等待某个通知 (notify)
  - (5)线程调用suspend方法将该线程挂起。
4. run方法运行结束后进入销毁阶段，整个线程执行完毕。

## 方法理解

wait方法：立即释放锁

sleep方法：不释放锁

notify方法：不立即释放锁

interrupt方法遇到wait方法方法：当调用wait方法后再执行interrupt方法会出现InterruptedException异常。

notify方法：只通知一个线程进行唤醒，唤醒的顺序与执行的wait顺序一致。

notifyAll方法：会按照执行wait方法的倒序一次对其他线程进行唤醒。

wait(long)方法：等待某一时间内是否有线程对锁进行notify通知唤醒，如果超过这段时间自动唤醒，能继续向下运行的前提是再次持有锁。如果没有锁也只能一直等待。

如果通知过早，会打乱程序正常运行逻辑，notify在前，wait在后，则有线程一直阻塞。

### 使用while的必要性

```
public class Service {
    public List<Integer> list = new ArrayList<>();

    public void add(){
        synchronized (list){
            list.add(new Random().nextInt());
            list.notifyAll();
        }
    }

    public void subtract(){
        try{
            synchronized (list){
                if(list.size() == 0){
                    System.out.println("进入wait");
                    list.wait();
                    System.out.println("结束wait");
                }
                list.remove(0);
            }
        }catch (InterruptedException e){
            e.printStackTrace();
        }
    }
}
```

```

public class MyThreadA extends Thread{
    private Service service;

    public MyThreadA(Service service){
        super();
        this.service = service;
    }

    @Override
    public void run() { service.add(); }
}

```

```

public class MyThreadB extends Thread{
    private Service service;

    public MyThreadB(Service service){
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.subtract();
    }
}

```

```

public class Run {

    public static void main(String[] args) throws InterruptedException {
        Service service = new Service();
        MyThreadA t1 = new MyThreadA(service);//add
        MyThreadB t3 = new MyThreadB(service);//subtract
        MyThreadB t4 = new MyThreadB(service);//subtract

        t3.start();
        t4.start();
        Thread.sleep(1000);
        t1.start();
    }
}

```

先启动了两个减法线程，一个遇到wait然后阻塞，另一个减法进程进入也阻塞，释放锁，这次加法线程得到锁，在list中加入一个值，然后notifyAll，其中一个减法进程先获得锁，输出结束wait,然后离开删除一个队列元素，运行完成释放锁，另一个减法进程获得锁，输出完成wait,但是再删除就报错。

```

进入wait
进入wait
结束wait
Exception in thread "Thread-2" java.lang.IndexOutOfBoundsException: Index: 0, Size: 0
结束wait
    at java.util.ArrayList.rangeCheck(ArrayList.java:653)
    at java.util.ArrayList.remove(ArrayList.java:492)
    at Chapter3.wait0ld.Service.subtract(Service.java:31)
    at Chapter3.wait0ld.MyThreadB.run(MyThreadB.java:20)

```

```

public void subtract(){
    try{
        synchronized (list){
            while(list.size() == 0){

```



```

        System.out.println("进入wait");
        list.wait();
        System.out.println("结束wait");
    }
    list.remove(0);
}
} catch (InterruptedException e){
    e.printStackTrace();
}
}

```

结果:

```

进入wait
进入wait
结束wait
结束wait
进入wait

```

两个减法进程都进入阻塞状态，等加法进程运行完通知所有减法进程，此时一个减法进程获得锁并离开while循环删除元素，释放锁。另一个减法进程获得锁后，此时list中依然为空，所以只能进入下一次循环，再次输出“进入wait”然后在wait方法出阻塞，这次没有加法进程，所以就一直处于阻塞状态。

## 一生产者—消费者

```

public class Service {
    private String str = "";
    private Object lock = new Object();
    public void setValue(){
        try{
            synchronized (lock){
                if(!str.equals("")){
                    lock.wait();
                }
                str = "set new string";
                System.out.println("set方法 string = " + str);
                lock.notify();
            }
        } catch (InterruptedException e){
            e.printStackTrace();
        }
    }

    public void getValue(){
        try{
            synchronized (lock){
                if(str.equals("")){
                    lock.wait();
                }
                str = "";
                System.out.println("get方法 string = " + str);
                lock.notify();
            }
        }
    }
}

```

```

        }
    }catch (InterruptedException e){
        e.printStackTrace();
    }
}
}
public class PThread extends Thread{
    private Service service;

    PThread(Service service){
        super();
        this.service = service;
    }
    @Override
    public void run() {
        while(true){
            service.setValue();
        }
    }
}
public class CThread extends Thread{
    private Service service;

    CThread(Service service){
        super();
        this.service = service;
    }
    @Override
    public void run() {
        while(true){
            service.getValue();
        }
    }
}
public class Run {
    public static void main(String[] args) {
        Service service = new Service();
        PThread p = new PThread(service);
        CThread c = new CThread(service);
        p.start();
        c.start();
    }
}

```

运行结果:

```

get方法 string =
set方法 string = set new string
get方法 string =
set方法 string = set new string
get方法 string =
set方法 string = set new string
get方法 string =
set方法 string = set new string
get方法 string =
set方法 string = set new string

```

### 多生产多消费（假死）

当多生产多消费的时候，判断临界条件不再使用if，而是改成while。

```

public class Run {
    public static void main(String[] args) {
        Service service = new Service();
        PThread[] p = new PThread[5];
        CThread[] c = new CThread[5];
        for(int i = 0;i<5;i++){
            p[i] = new PThread(service);
            p[i].start();

            c[i] = new CThread(service);
            c[i].start();
        }
    }
}

```

结果：假死了

```

set方法 string = set new string
get方法 string =
set方法 string = set new string
get方法 string =
set方法 string = set new string
get方法 string =
set方法 string = set new string
get方法 string =
set方法 string = set new string
get方法 string =
set方法 string = set new string
get方法 string =

```

因为notify唤醒的不保证是异类，也可能是同类。生产者唤醒生产者，消费者唤醒消费者，那么积少成多，就会导致所有线程运行不下去。

应该将notify改为notifyAll方法，所有阻塞线程一起唤醒。

### ★ 允许连续生产多个；允许连续消费多个

生产者和消费者都需要多一个线程来检测是否超过边界（list是否超过50，是否小于0）

### 通过管道进行线程间通信——字节流

管道流是一种特殊的流，用于在不同线程间直接传送数据。一个线程发送一个线程接收。JDK提供了4个类进行线程间的通信：PipedInputStream和

PipedOutputStream,PipedReader,PipedWriter

Java在它的jdk文档中提到不要在一个线程中同时使用PipedInputStream和PipedOutputStream，这会造成死锁

java.io.PipedInputStream.read()的声明：public int read(byte[] b, int off, int len)

b数组存储从管道中读取的数据。

off是b的偏移开始。

len是读取的最大长度。

如果是read()没有参数则是读取下一个数据字节。

将最多 len 个数据字节从此管道输入流读入 byte 数组。如果已到达数据流的末尾，或者 len 超出管道缓冲区大小，则读取的字节数将少于 len。如果 len 为 0，则不读取任何字节并返回 0；否则，在至少 1 个输入字节可用、检测到流末尾、抛出异常前，该方法将一直阻塞。

读写业务代码

```
public class ReaderAndWriter {  
  
    public void readData(PipedInputStream input){  
        System.out.println("read: ");  
        byte[] byteArray = new byte[20];  
        try {  
            int readLength = input.read(byteArray);//读取字节到byteArray  
            while(readLength != -1){  
                String newData = new String(byteArray,0,readLength);  
                System.out.println(newData);  
                readLength = input.read(byteArray);//读取字节到byteArray
```

```

        }
        System.out.println();
        input.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public void writeData(PipedOutputStream out){
    try{
        System.out.println("write: ");
        for(int i = 0;i<300;i++){
            String outData = ""+(i+1);
            out.write(outData.getBytes());
            System.out.print(outData);
        }
        System.out.println();
        out.close();
    }catch (IOException e){
        e.printStackTrace();
    }
}
}

```

#### 读线程

```

public class ReadThread extends Thread {
    private ReaderAndWriter rw;
    private PipedInputStream input;

    ReadThread(ReaderAndWriter rw, PipedInputStream input){
        this.rw = rw;
        this.input = input;
    }

    @Override
    public void run() {
        rw.readData(input);
    }
}

```

#### 写线程

```

public class WriteThread extends Thread {
    private ReaderAndWriter rw;
    private PipedOutputStream out;

    WriteThread(ReaderAndWriter rw, PipedOutputStream out){
        this.rw = rw;
        this.out = out;
    }

    @Override
    public void run() {
        rw.writeData(out);
    }
}

```



```

public class Run {
    public static void main(String[] args) {

        ReaderAndWriter rw = new ReaderAndWriter();

        PipedOutputStream out = new PipedOutputStream();
        PipedInputStream in = new PipedInputStream();
        try {
            out.connect(in);
            //in.connect(out); 一次连接就够了

            ReadThread readThread = new ReadThread(rw,in);
            readThread.start();
            Thread.sleep(1000);

            WriteThread writeThread = new WriteThread(rw,out);
            writeThread.start();

        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

```

read:
write:
12345678910111213141516171819202122232425262728293031323334353637
12345678910111213141516171819202122232425262728293031323334353637

```

首先是读线程启动，由于当时没有数据被写入，所以线程阻塞在  
`int readLength=in.read(byteArray)`,知道有数据被写入，才继续向下运行。

字符流PipedReader和PipedWriter使用方式类似，也是read方法和write方法。只是  
 存储读取的数据数组改成字符数组而不是字节数组 `char charArray = new char[20]`

### 实现wait/notify的交叉备份

```

public class DBTools {
    volatile private boolean prevIsA = false;

    synchronized public void backupA(){
        try{
            while(prevIsA == true){
                wait();
            }
            System.out.println("*****");
            prevIsA = true;
        }
    }
}

```

```

        notifyAll();
    }catch (InterruptedException e){
        e.printStackTrace();
    }
}

synchronized public void backupB(){
    try{
        while(prevIsA == false){
            wait();
        }
        System.out.println("$$$$$$");
        prevIsA = false;
        notifyAll();
    }catch (InterruptedException e){
        e.printStackTrace();
    }
}
}

public class BackupB extends Thread {

    private DBTools dbTools;
    public BackupB(DBTools dbTools){
        this.dbTools = dbTools;
    }
    @Override
    public void run() {
        dbTools.backupB();
    }
}

public class BackupA extends Thread {

    private DBTools dbTools;
    public BackupA(DBTools dbTools){
        this.dbTools = dbTools;
    }
    @Override
    public void run() {dbTools.backupA(); }
}

public class Run {

    public static void main(String[] args) {
        DBTools dbTools = new DBTools();
        for(int i=0; i<10; i++){
            BackupA backupA = new BackupA(dbTools);
            BackupB backupB = new BackupB(dbTools);
            backupA.start();
            backupB.start();
        }
    }
}

```

能够实现交替输出的关键是有prevIsA作为标记。 volatile private boolean

prevIsA = false;

## join方法的使用

主线程创建并启动子线程，如果子线程要进行大量的耗时运算，主线程在子线程结束之前结束。如果主线程想要子线程执行完成之后再结束，就要用join方法。join方法的作用是等待线程对象销毁。

使用：MyThread testThread = new MyThread(); testThread.start();  
testThread.join()

join方法的作用是使**所属的线程对象 (testThread) x** 正常执行run方法中的任务，而使**当前线程(main线程) z** 进行无限期的阻塞，等待线程x销毁后再执行z后面的代码，具有串联执行的效果。

join方法是在内部使用wait方法进行等待，synchronized关键字使用锁作为同步。

## join方法和interrupt方法出现异常

如果彼此遇到，则出现异常，不管先后顺序。

## join(long)方法的使用

x.join(long)方法中的参数用于设定等待的时间，不管x是否执行完毕，时间到了且重新获得锁，则当前方法会继续向后运行，如果咩有重新获得锁，则一直在尝试，知道获得锁位置。（因为join内部是用wait阻塞了当前进程，获得锁就可以继续执行）

## join(long)和sleep(long)的区别

join(long)方法的功能在内部是使用wait(long)来实现的，所以join(long)具有**释放锁**的特点。

而Thread.sleep(long)不释放锁。

## 类ThreadLocal的使用

变量值的共享可以使用public static变量的形式实现，所有的线程都使用同一个public static变量。如何实现自己的变量：jdk提供的ThreadLocal用于解决这样的问题。

类ThreadLocal的主要作用是将数据放如**当前线程对象**的Map中，这个Map是Thread类的实例变量。类ThreadLocal自己不管理，不存储任何数据，只是一个数据和Map

之间的桥梁。数据->ThreadLocal->currentThread()->Map

执行后每个线程中的Map都存有自己的数据，Map中的key存储的是ThreadLocal对象，value存储的是值。每个Thread中的Map值只对当前线程可见，其他线程不可以访问当前线程中的Map的值。当前线程销毁，map也销毁。map中的数据如果没有被应用，没有被使用，则被GC收回。

```
public class ThreadA extends Thread{

    private ThreadLocal threadLocal;
    ThreadA(ThreadLocal threadLocal){
        super();
        this.threadLocal = threadLocal;
    }
    @Override
    public void run() {
        try{
            for(int i = 0;i<5;i++){
                threadLocal.set("A" +(i+1));
                System.out.println("A get "+threadLocal.get());
                Thread.sleep(1000);
            }
        }catch (InterruptedException e){
            e.printStackTrace();
        }
    }
}

public class ThreadB extends Thread{

    private ThreadLocal threadLocal;
    ThreadB(ThreadLocal threadLocal){
        super();
        this.threadLocal = threadLocal;
    }
    @Override
    public void run() {
        try{
            for(int i = 0;i<5;i++){
                threadLocal.set("B" +(i+1));
                System.out.println("B get "+threadLocal.get());
                Thread.sleep(1000);
            }
        }catch (InterruptedException e){
            e.printStackTrace();
        }
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        ThreadLocal threadLocal = new ThreadLocal();
        ThreadA threadA = new ThreadA(threadLocal);
        ThreadB threadB = new ThreadB(threadLocal);
    }
}
```

```

        threadA.start();
        threadB.start();
        for(int i=0;i<5;i++){
            threadLocal.set("main" + (i+1));
            System.out.println("main get " + threadLocal.get());
            Thread.sleep(1000);
        }
    }
}

```

每个线程都可以通过ThreadLocal向每一个线程存储自己的私有数据。3个线程都往threadLocal对象中set数据值，每个线程还是能取出自己的数据，不能取出别的线程的。

### 解决get方法返回null的问题

第一次调用get方法返回null，如何具有默认值的效果呢。

继承ThreadLocal类产生ThreadLocalExt类，重写ThreadLocal的initialValue()方法。

书中是单写了一个ThreadLocalExt来完成重写，但是可以用匿名内部类来完成重写。

```

public class Main {
    public static void main(String[] args) throws InterruptedException {
        ThreadLocal threadLocal = new ThreadLocal(){
            @Override
            protected Object initialValue() {
                return "我是默认值";
            }
        };
        ThreadA threadA = new ThreadA(threadLocal);
        ThreadB threadB = new ThreadB(threadLocal);
        threadA.start();
        threadB.start();
        for(int i=0;i<5;i++){
            System.out.println("main get " + threadLocal.get());

            threadLocal.set("main" + (i+1));
            Thread.sleep(1000);
        }
    }
}

```

```

C:\Program Files\Java
main get 我是默认值
B get 我是默认值
A get 我是默认值
A get A1
main get main1
B get B1

```



## 类InheritableThreadLocal的使用

使用类InheritableThreadLocal可使子线程继承父线程的值。

main线程创建了ThreadA线程，所以main方法是ThreadA的父线程，但是main线程中的值并没有继承为ThreadA，所以ThreadLocal并不具有值继承特性，需要使用InheritableThreadLocal。

需要注意的是：要在set值之后创建的线程才能继承main的值。如果在set之前创建就获取不到（null）

```
public class Main {  
    public static void main(String[] args) throws InterruptedException {  
        InheritableThreadLocal threadLocal = new InheritableThreadLocal();  
  
        for(int i=0;i<5;i++){  
            threadLocal.set("main线程放入的值"+(i+1));  
            System.out.println("main get "+threadLocal.get());  
            Thread.sleep(1000);  
        }  
        ThreadA threadA = new ThreadA(threadLocal);  
        ThreadB threadB = new ThreadB(threadLocal);  
        threadA.start();  
        threadB.start();  
    }  
}
```

```
main get main线程放入的值1  
main get main线程放入的值2  
main get main线程放入的值3  
main get main线程放入的值4  
main get main线程放入的值5  
A get main线程放入的值5  
B get main线程放入的值5
```

**父线程有最新的值，子线程仍是旧值 子线程有最新的值，父线程仍是旧值**

```
public class Main {  
    public static void main(String[] args) throws InterruptedException {  
        InheritableThreadLocal threadLocal = new InheritableThreadLocal();  
  
        for(int i=0;i<5;i++){  
            threadLocal.set("main线程放入的值"+(i+1));  
            System.out.println("main get "+threadLocal.get());  
            Thread.sleep(1000);  
        }  
    }  
}
```

```

        ThreadA threadA = new ThreadA(threadLocal);
        ThreadB threadB = new ThreadB(threadLocal);
        threadA.start();
        threadB.start();
        threadLocal.set("main线程创建线程后放入的值");
        System.out.println("main get "+threadLocal.get());
    }
}

```

通过构造方法的完整源代码算法可以发现，子线程将父线程中的table对象以复制的方式赋值给子线程的table数组，这个过程是在创建Thread类对象时发生，也就说明当子线程创建完毕后，子线程中的数据就是主线程中**旧的数据**，因为主线程和子线程用两个Entry[]对象数组各自存储自己的值。

### 子线程可以感应对象属性值的变化

前面示例都是在主、子线程中使用String数据类型做继承特性的演示。当子线程从父线程继承可变对象数据类型时，子线程可以取到最新对象中的值。

```

public class ThreadA extends Thread{

    private InheritableThreadLocal<User> threadLocal;
    ThreadA(InheritableThreadLocal threadLocal){
        super();
        this.threadLocal = threadLocal;
    }

    @Override
    public void run() {
        try{
            for(int i = 0;i<5;i++){
                User user = threadLocal.get();
                System.out.println("ThreadA中的值 "+user.getUsername());
                Thread.sleep(1000);
            }
        }catch (InterruptedException e){
            e.printStackTrace();
        }
    }
}

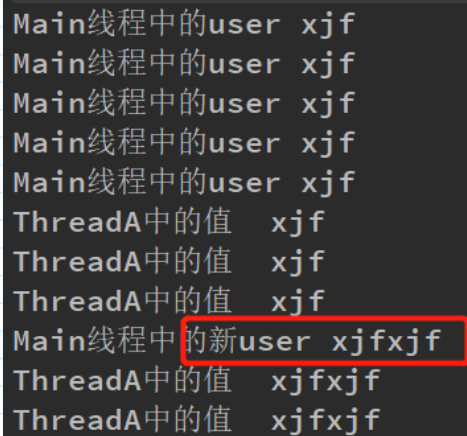
public class Main {
    public static void main(String[] args) throws InterruptedException {
        InheritableThreadLocal<User> threadLocal = new
        InheritableThreadLocal();
        User user = new User();
        user.setUsername("xjf");
        for(int i=0;i<5;i++){
            threadLocal.set(user);
            System.out.println("Main线程中的user "+ user.getUsername());
        }
    }
}

```

```

    }
    ThreadA threadA = new ThreadA(threadLocal);
    threadA.start();
    Thread.sleep(2000);
    user.setUsername("xjfxjf");
    System.out.println("Main线程中的新用户 "+user.getUsername());
}
}

```



```

Main线程中的user xjf
Main线程中的user xjf
Main线程中的user xjf
Main线程中的user xjf
Main线程中的user xjf
ThreadA中的值 xjf
ThreadA中的值 xjf
ThreadA中的值 xjf
Main线程中的新用户 xjfxjf
ThreadA中的值 xjfxjf
ThreadA中的值 xjfxjf

```

如果重新set进main函数的是一个新的对象，那依然是父线程拥有新的值，子进程是旧的值。

### 重写InheritableThreadLocal中的childValue方法实现对继承的值进行加工

子线程可以对父线程继承的值进行加工。重写childValue方法实现子线程具有最新的值是只有在创建子线程时才会发生，仅是一次。

```

public class Main {
    public static void main(String[] args) throws InterruptedException {
        InheritableThreadLocal threadLocal = new InheritableThreadLocal(){
            @Override
            protected Object childValue(Object parentValue) {
                return parentValue+"子线程后缀";
            }
        };
        for(int i=0;i<5;i++){
            threadLocal.set("xjf");
            System.out.println("Main线程中 "+ threadLocal.get());
        }
        ThreadA threadA = new ThreadA(threadLocal);
        threadA.start();
        Thread.sleep(2000);
    }
}

```

等待唤醒机制涉及的方法总结：

1.wait();让线程处于阻塞状态，这时线程会释放执行资格和执行权，被wait的线程会

被存储到线程池中2.notify();唤醒线程池中的一个线程（任意）

3.notifyAll();唤醒线程池中的所有线程这些方法都必须定义在同步中

因为这些方法是用于操作线程状态的方法，必须要明确到底操作的是哪个锁上的线程。例如在a锁里wait必须在该锁里notify。由于锁可以是任意对象，故这些方法定义在Object类中。等待和唤醒必须是同一个锁。

## 第4章 Lock对象的使用

2020年12月20日 13:17

### 使用ReentrantLock类

synchronized用来实现线程同步。jdk1.5新增的ReentrantLock也可以达到同样效果，同时还具有嗅探锁定，多路分支通知等功能。

### 使用ReentrantLock实现同步

```
public class Run {  
    public static void main(String[] args) {  
        MyService service = new MyService();  
        for(int i=0;i<5;i++){  
            MyThread myThread = new MyThread(service);  
            myThread.start();  
        }  
    }  
}
```

创建MyThread类继承Thread类，调用MyService的testMethod方法，不复制在这里了。

```
public class MyService {  
    private ReentrantLock lock = new ReentrantLock();  
  
    public void testMethod(){  
        lock.lock();  
        for(int i = 0;i<5;i++){  
            System.out.println(Thread.currentThread().getName() + " " + (i+1));  
        }  
        lock.unlock();  
    }  
}
```



```
Thread-0 1
Thread-0 2
Thread-0 3
Thread-0 4
Thread-0 5
Thread-3 1
Thread-3 2
Thread-3 3
Thread-3 4
Thread-3 5
Thread-1 1
Thread-1 2
Thread-1 3
```

**ReentrantLock**的lock方法获取锁，unlock方法释放锁，这两个方法成对使用。

只有当当前线程输出完毕后将锁释放，其他线程才可以继续抢锁并输出，每个线程内输出的数据是有序的。线程之间输出的顺序是随机的，谁抢到锁，谁输出。

## await方法的使用

synchronized与wait方法，notify，notifyAll方法相结合实现wait/notify模式。ReentrantLock类也可以实现同样功能，但需要借助Condition对象。**Condition**类是JDK5的技术，具有更好灵活性。例如可以实现多路通知功能，也就是在一个Lock对象中可以创建多个Condition实例，线程对象注册在指定的Condition中，从而可以有选择性的进行线程通知，在线程调度上更加灵活。

notify方法进行通知时，被通知的线程由JVM选择。notifyAll方法会通知所有waiting线程，没有选择权，会出现相当大的效率问题，但使用ReentrantLock结合Condition实现选择性通知。

## 使用await方法和signal方法实现wait/notify机制

condition.await()方法调用之前需要调用lock.lock()获得锁。

```
public class MyService {
    private Lock lock = new ReentrantLock();
    private Condition condition = lock.newCondition();

    public void await(){
```

```

    try{
        lock.lock();
        System.out.println("wait begin");
        condition.await();
        System.out.println("wait end");
        lock.unlock();
    }catch (InterruptedException e){
        e.printStackTrace();
    }
}
}

public void signal(){
    try{
        lock.lock();
        System.out.println("signal begin");
        condition.signal();
        System.out.println("signal end");
    }finally {
        lock.unlock();
    }
}
}

public class Run {
    public static void main(String[] args) {
        MyService myService = new MyService();
        ThreadA threadA = new ThreadA(myService);
        ThreadB threadB = new ThreadB(myService);
        threadA.start();
        threadB.start();
    }
}

```

```

wait begin
signal begin
signal end
wait end

```

Object类中的wait方法 == Condition类中的await方法

```

wait(long timeout) == await(long time, TimeUnit unit)
notify()           == signal()
notifyAll()        == signalAll()

```

### await方法暂停线程运行的原理

并发包源代码内部执行了Unsafe类中的public native void park(boolean

isAbsolute,long time), 让当前线程呈暂停状态, 方法参数isAbsolute代表是否为绝对时间, 方法参数time代表时间值。如果对参数isAbsolute传入true, 第二个参数time时间单位为毫秒, 如果是false, 则第2个参数时间单位为纳秒。

```
public static void park(Object blocker){
    Thread t = Thread.currentThread();
    setBlocker(t,blocker);
    UNSAFE.park(false,0L);
    setBlocker(t, null);
}
```

## 通知部分线程

如果想单独唤醒部分线程, 有必要使用多个Condition对象, 有助于提示程序的运行效率, 可以对线程进行分组, 然后唤醒指定组中的线程。

```
public class MyService {
    Lock lock = new ReentrantLock();
    Condition conditionA = lock.newCondition();
    Condition conditionB = lock.newCondition();

    public void awaitA(){
        try{
            lock.lock();
            System.out.println("awaitA begin " +
Thread.currentThread().getName());
            conditionA.await();
            System.out.println("awaitA end " + Thread.currentThread().getName());
        }catch (InterruptedException e){
            e.printStackTrace();
        }finally {
            lock.unlock();
        }
    }

    public void awaitB(){
        try{
            lock.lock();
            System.out.println("awaitB begin
"+Thread.currentThread().getName());
            conditionB.await();
            System.out.println("waitB end " + Thread.currentThread().getName());
        }catch (InterruptedException e){
            e.printStackTrace();
        }finally {
            lock.unlock();
        }
    }
}
```

```

    }
}

public void signalAll_A(){
    try{
        lock.lock();
        System.out.println("signalAll_A " + Thread.currentThread().getName());
        conditionA.signalAll();
    }finally {
        lock.unlock();
    }
}

public void signalAll_B(){
    try{
        lock.lock();
        System.out.println("signalAll_B " + Thread.currentThread().getName());
        conditionB.signalAll();
    }finally {
        lock.unlock();
    }
}
}

public class ThreadA extends Thread {
    private MyService service;

    ThreadA(MyService service){
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.awaitA();
    }
}

public class ThreadB extends Thread {
    private MyService service;

    ThreadB(MyService service){
        super();
        this.service = service;
    }

    @Override
    public void run() {
        service.awaitB();
    }
}
}

```

```

public class Run {
    public static void main(String[] args) throws InterruptedException {
        MyService service = new MyService();
        ThreadA threadA = new ThreadA(service);
        ThreadB threadB = new ThreadB(service);
        threadA.start();
        threadB.start();

        Thread.sleep(3000);
        System.out.println("三秒后signalAll_A");
        service.signalAll_A();
    }
}

```

```

awaitA begin Thread-0
awaitB begin Thread-1
三秒后signalAll_A
signalAll_A main
awaitA end Thread-0

```

3s后只有线程A被唤醒。

使用Condition对象可以唤醒指定种类的线程，这是控制部分线程行为的方便方式。

并发包Condition是Doug Lea发布的基于java语言的Condition并发工具包，提供了丰富的功能，是的java语言在并发编程上较其他语言更有优势。

## 实现生产者/消费者一对一交替输出

```

public class MyService {

    private Lock lock = new ReentrantLock();
    private Condition condition = lock.newCondition();
    private boolean hasValue = false;

    public void setValue(){
        try{
            lock.lock();
            while (hasValue == true){
                condition.await();
            }
            hasValue = true;
            System.out.println("set value success");
            condition.signal();
        }catch (InterruptedException e){

```



```

        e.printStackTrace();
    }finally {
        lock.unlock();
    }
}

public void getValue(){
    try{
        lock.lock();
        while(hasValue == false){
            condition.await();
        }
        hasValue = false;
        System.out.println("get value success");
        condition.signal();
    }catch (InterruptedException e){
        e.printStackTrace();
    }finally {
        lock.unlock();
    }
}

}

public class ThreadA extends Thread {
    private MyService service;

    ThreadA(MyService service){
        super();
        this.service = service;
    }

    @Override
    public void run() {
        while(true)
            service.setValue();
    }
}

public class ThreadB extends Thread {
    private MyService service;

    ThreadB(MyService service){
        super();
        this.service = service;
    }

    @Override
    public void run() {
        while(true)
            service.getValue();
    }
}

```

```

    }
}
public class Run {
    public static void main(String[] args) throws InterruptedException {
        MyService service = new MyService();
        ThreadA threadA = new ThreadA(service);
        ThreadB threadB = new ThreadB(service);
        threadA.start();
        threadB.start();

    }
}

```

```

set value success
get value success
set value success
get value success
set value success
get value success
set value success
get value success
进程完成，退出码 1

```

上面是一一对一，所以setValue和getValue中的可以不是while而是if，因为最多一个线程调用该方法。但是当多个生产线程和消费线程的时候。就需要用while。

## 实现生产者/消费者模式多对多交替输出

主函数创建多个线程即可。

```

public class Run {
    public static void main(String[] args) throws InterruptedException {
        MyService service = new MyService();
        ThreadA[] threadA = new ThreadA[10];
        ThreadB[] threadB = new ThreadB[10];

        for(int i = 0;i<5;i++){
            threadA[i] = new ThreadA(service);
            threadB[i] = new ThreadB(service);
            threadA[i].start();
            threadB[i].start();
        }

    }
}

```

```
Run (2) Run (2) Run (3) Run (4)
get value success
set value success
get value success
set value success
get value success
set value success
get value success
set value success
get value success
```

会造成假死现象。

改用signalAll()来解决假死现象。可能会出现连续的get或者set，因为signalAll可能唤醒的是同类线程。

## 公平锁和非公平锁

公平锁：采用先到先得的策略，每次获取锁之前都会先检查队列里有没有排队等待的线程，没有才会尝试获取锁，如果有就将当前线程追加到队列中。

非公平锁：采用“有机会插队”的策略，一个线程获取锁之前要先去尝试获取锁而不是在队列中等待。如果获取锁成功，则说明线程虽然是后启动的，但是先获得了锁。这就是“作弊插队”的效果。如果获取锁没有成功，再将自身追加到队列中进行等待。

公平锁示例：

```
public class MyService {
    private Lock lock;

    public MyService(boolean fair){
        lock = new ReentrantLock(fair);
    }

    public void testMethod(){
        try{
            lock.lock();
            System.out.println(Thread.currentThread().getName());
            Thread.sleep(500);//这里的暂停是配合main方法中的500ms让array2有机
```

会在不公平的情况下抢到锁。

```
        lock.unlock();
    } catch (InterruptedException e){
        e.printStackTrace();
    }
}
}
public class Run {
    public static void main(String[] args) throws InterruptedException {
        MyService service = new MyService(true);//公平锁
        MyThread[] arrayA = new MyThread[10];
        MyThread[] arrayB = new MyThread[10];

        for(int i=0;i<10;i++){
            arrayA[i] = new MyThread(service);
            arrayA[i].setName("arrayA "+(i+1));
        }
        for(int i=0;i<10;i++){
            arrayA[i].start();
        }
        for(int i=0;i<10;i++){
            arrayB[i] = new MyThread(service);
            arrayB[i].setName("arrayB "+(i+1));
        }
        Thread.sleep(500);

        for(int i=0;i<10;i++){
            arrayB[i].start();
        }
    }
}
```

```
arrayA 1  
arrayA 2  
arrayA 3  
arrayA 4  
arrayA 5  
arrayA 6  
arrayA 10  
arrayA 8  
arrayA 9  
arrayA 7  
arrayB 2  
arrayB 1  
arrayB 4  
arrayB 3  
arrayB 5  
arrayB 6  
arrayB 7  
arrayB 8  
arrayB 9  
arrayB 10
```

运行结果arrayA在前，arrayB在后，说明arrayB没有任何机会抢到锁。这就是公平锁的特点。

```
public static void main(String[] args) throws InterruptedException {  
    MyService service = new MyService(false); //公平锁  
    MyThread[] arrayA = new MyThread[10];  
    MyThread[] arrayB = new MyThread[10];  
  
    for(int i=0;i<10;i++){  
        arrayA[i] = new MyThread(service);  
        arrayA[i].setName("arrayA " + (i+1));  
    }  
    for(int i=0;i<10;i++){  
        arrayA[i].start();  
    }  
    for(int i=0;i<10;i++){  
        arrayB[i] = new MyThread(service);  
        arrayB[i].setName("arrayB " + (i+1));  
    }  
    // Thread.sleep(500);  
  
    for(int i=0;i<10;i++){  
        arrayB[i].start();  
    }  
}
```

```
}
```

多次执行后，使用非公平锁时有可能提前输出array2，说明启动的线程先抢到了锁，这就是非公平锁的特点。

A线程持有锁后，B线程不能执行的原理是在内部执行了unsafe.park(false,0L)代码；A线程释放锁后B线程可以运行的原理是当A线程执行unlock方法时在内部执行了unsafe.unpark(Bthread),B线程得以运行。

### **public int getHoldCount()方法的使用**

getHoldCount()方法的作用是查询“**当前线程**”保持此锁定的个数，即调用lock方法的次数。

### **public final int getQueueLength()方法的使用**

public final int getQueueLength()方法的作用是返回正在等待获取此锁的线程估计书。调用getQueueLength()方法后返回值是4，说明有4个线程在同时等待该锁的释放。

### **public int getWaitQueueLength(Condition condition)方法的使用**

返回等待与次所相关的给定条件Condition的线程估计数。例如，这里有5个线程，每个线程都执行了同一个Condition对象的await方法，则调用getWaitQueueLength(Condition condition)方法，返回int值5.7

### **public final boolean hasQueuedThread(Thread thread)方法的使用**

查询指定的线程是否正在等待获取此锁，也就是判断参数中的线程是否在等待队列中。

```
lock.hasQueuedThread(threadA)
```

### **public final boolean hasQueuedThreads()方法的使用**

查询是否有线程正在等待获取此锁。

```
lock.hasQueuedThreads()
```

### **public boolean hasWriters(Condition condition)**

查询**是否有**线程正在等待与此锁有关的condition条件，也就是是否有线程执行了condition对象中的await方法而呈等待状态。public int

getWaitQueueLength(Condition condition) 方法的作用是返回有多少个线程执行



了condition对象中的await方法而呈等待状态。

### **public final boolean isFair()方法的使用**

判断是不是公平锁。

### **public boolean isHeldByCurrentThread()**

查询当前线程是否保持此锁。lock.isHeldByCurrentThread

### **public boolean isLocked()方法的使用**

查询此锁是否由任意线程保持。并没有释放。

### **public void lockInterruptibly()方法的使用**

当某个线程尝试获得锁并且阻塞在lockInterruptibly方法时，该线程可以被中断。

### **public boolean tryLock()方法的使用**

嗅探拿锁，如果当前发现锁被其他线程持有，则返回false，继续执行后面的代码，而不是呈阻塞等待锁的状态。

### **public boolean tryLock(long timeout,TimeUnit unit)方法使用**

嗅探拿锁，如果当前线程发现锁被其他线程持有了，则返回false，继续执行后面代码，而不是呈阻塞等待状态。如果当前线程在指定的timeout时间内持有了锁，返回值是true，超过时间则返回false。timeout代表当前线程抢锁时间。

### **public boolean await (long time,TimeUnit unit)方法使用**

和public final native void wait(long timeout)方法一样，都具有自动唤醒线程的功能。

### **public long awaitNanos(long nanosTimeout)方法的使用**

自动唤醒 纳秒ns

1000ns等于1 $\mu$ s 1000 $\mu$ s等于1ms,1000ms等于1s

### **public boolean awaitUntil(Date deadline)方法使用**

在指定的Date结束等待。在等待时间到达前，可以被其他线程提前唤醒。

### **public void awaitUninterruptibly()方法的使用**

实现线程在等待的过程中，不允许被中断。也即使执行threadA.interrupt ()并不会出现异常。

## 实现线程按顺序执行业务

```
public class MyService {
    private Lock lock = new ReentrantLock();
    private Condition condition = lock.newCondition();

    volatile private int nextWhoPrint = 1;

    public void testMethod(){
        try{
            lock.lock();
            while(nextWhoPrint != 1){
                condition.await();
            }
            System.out.println("AAA");
            nextWhoPrint = 2;
            condition.signalAll();
            lock.unlock();
        }catch (InterruptedException e){
            e.printStackTrace();
        }
    }

    public void testMethod2(){
        try{
            lock.lock();
            while(nextWhoPrint != 2){
                condition.await();
            }
            System.out.println("  BBB");
            nextWhoPrint = 3;
            condition.signalAll();
            lock.unlock();
        }catch (InterruptedException e){
            e.printStackTrace();
        }
    }

    public void testMethod3(){
        try{
            lock.lock();
            while(nextWhoPrint != 3){
                condition.await();
            }
            System.out.println("    CCC");
            nextWhoPrint = 1;
            condition.signalAll();
        }
    }
}
```

```

        lock.unlock();
    }catch (InterruptedException e){
        e.printStackTrace();
    }
}
}

public class Run {
    public static void main(String[] args) {
        MyService service = new MyService();
        for(int i = 0;i<5;i++){
            ThreadA a = new ThreadA(service);
            a.start();
            ThreadB b = new ThreadB(service);
            b.start();
            ThreadC c = new ThreadC(service);
            c.start();
        }
    }
}

```



```

AAA
  BBB
    CCC
AAA
  BBB
    CCC
AAA
  BBB
    CCC
AAA
  BBB
    CCC
AAA
  BBB
    CCC

```

## 使用ReentrantReadWriteLock类

ReentrantLock类具有完全互斥排他的效果，同一时间只有一个线程在执行ReentrantLock.lock方法后面的任务，这样虽然保证了同时写实例变量的线程安全性，但效率低下。JDK提供了ReentrantReadWriteLock类，再进行**读操作**时不需要同步执行。

读写锁有两个锁：一个是读操作相关的锁，也称共享锁；另一个是写操作有关的锁，也称排他锁。

读锁之间不互斥，读锁和写锁互斥，写锁与写锁互斥，因此只要出现写锁，就会出现

互斥同步的效果。

读操作是指读取实例变量的值，写操作是指向实例变量写入值。

```
public class MyService {
    private Lock lock = new ReentrantLock();
    private String username = "abc";

    public void testMethod1(){
        try{
            lock.lock();
            System.out.println("begin "+System.currentTimeMillis());
            System.out.println("print service " + username);
            Thread.sleep(4000);
            System.out.println("end "+System.currentTimeMillis());
            lock.unlock();
        } catch (InterruptedException e){
            e.printStackTrace();
        }
    }
}

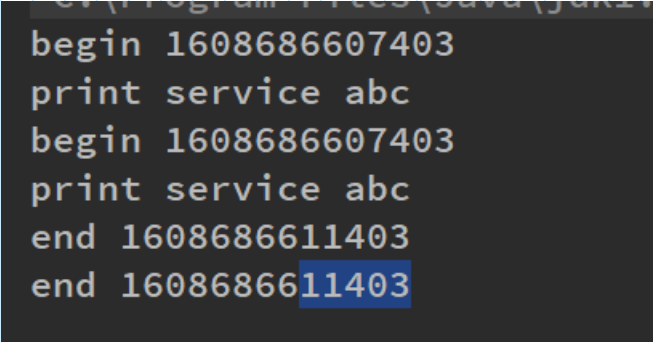
public class Run {
    public static void main(String[] args) {
        MyService service = new MyService();
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                service.testMethod1();
            }
        };
        Thread threadA = new Thread(runnable);
        threadA.start();
        Thread threadB = new Thread(runnable);
        threadB.start();
    }
}
```

```
begin 1608686355691
print service abc
end 1608686359692
begin 1608686359692
print service abc
end 1608686363692
```

从运行的时间来看，两个线程读取实例变量共耗时8s，每个线程占用4s，非常浪费时间。

### 利用ReentrantReadWriteLock类的使用——读读共享

```
public class MyService {  
    private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();  
    private String username = "abc";  
  
    public void testMethod1(){  
        try{  
            lock.readLock().lock();  
            System.out.println("begin "+System.currentTimeMillis());  
            System.out.println("print service " + username);  
            Thread.sleep(4000);  
            System.out.println("end "+System.currentTimeMillis());  
            lock.readLock().lock();  
        }catch (InterruptedException e){  
            e.printStackTrace();  
        }  
    }  
}
```



```
begin 1608686607403  
print service abc  
begin 1608686607403  
print service abc  
end 1608686611403  
end 1608686611403
```

从输出来看，两个线程几乎同时进入lock方法后面的代码，一共耗时4s,说明lock.readLock读锁可以提高程序运行效率，允许多个线程同时执行lock()方法后面的代码。

在此实例中，完全不使用锁也可以实现异步。但是有可能存在第三个线程在执行写操作，再执行写操作的时候就不能和这两个读操作同时运行了。

### 利用ReentrantReadWriteLock类的使用——写写共享

```
public class MyService {  
    private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();  
    private String username = "abc";  
  
    public void testMethod1(){  
        try{  
            lock.writeLock().lock();  
            System.out.println("begin "+System.currentTimeMillis());  
            System.out.println("print service " + username);  
            Thread.sleep(4000);  
            System.out.println("end "+System.currentTimeMillis());  
            lock.writeLock().lock();  
        }catch (InterruptedException e){  
            e.printStackTrace();  
        }  
    }  
}
```

lock.writeLock()的效果是同一时间只允许一个线程执行lock后面的代码。

## 利用ReentrantReadWriteLock类的使用——读写共享

就是把上面两个MyService放到一起。

总结：只要出现“写操作”，就是互斥的。



# 第5章 定时器Timer

2020年12月23日 10:45

## 定时器Timer的使用

Timer类主要作用是设置计划任务，即在指定时间开始执行某一个任务。

TimerTask类的主要作用是封装任务。

执行计划任务的代码要放在TimerTask的子类中，因为它是个抽象类。

### **schedule (TimerTask task,Date time)方法的测试**

该方法的作用是在指定日期执行一次某一任务。

```
public class Run {  
    public static void main(String[] args) throws InterruptedException {  
        long nowTime = System.currentTimeMillis();  
        System.out.println("当前时间是: "+nowTime);  
  
        long scheduleTime = nowTime + 10000;  
        System.out.println("计划时间为: "+scheduleTime);  
  
        MyTask myTask = new MyTask();  
        Timer timer = new Timer();  
        Thread.sleep(1000);  
        timer.schedule(myTask,new Date(scheduleTime));  
        System.out.println("main线程结束");  
    }  
}  
  
public class MyTask extends TimerTask {  
    @Override  
    public void run() {  
        System.out.println("任务执行时间:"+System.currentTimeMillis());  
    }  
}
```



```
当前时间是: 1608692511719  
计划时间为: 1608692521719  
main线程结束  
任务执行时间:1608692521720
```

任务执行完发现还有进程未销毁，按钮呈红色显示。

原因是创建Timer对象时启动了一个新的非守护线程，用这个新启动的线程去执行计划任务

新启动的线程并不是守护线程，而是一直再运行，原因是新线程内部有一个死循环。

### 使用public void cancel()方法实现线程TimerThread销毁

cancel方法是终止此计时器，丢弃所有当前已安排的任务。这不会干扰当前正在执行的任务。在此计时器调用的计时器任务的run方法内调用此方法，可以确保正在执行的任务是此计时器所执行的最后一个任务。

```
public class Run2 {  
    public static void main(String[] args) throws InterruptedException {  
        long nowTime = System.currentTimeMillis();  
        System.out.println("当前时间是: "+nowTime);  
  
        long scheduleTime = nowTime + 10000;  
        System.out.println("计划时间为: "+scheduleTime);  
  
        MyTask myTask = new MyTask();  
        Timer timer = new Timer();  
        timer.schedule(myTask,new Date(scheduleTime));  
        Thread.sleep(10000);  
        timer.cancel();  
        System.out.println("main线程结束");  
    }  
}
```

此时TimerThread线程销毁了。

### 如果计划时间早于当前时间——立即运行的效果

如果执行任务的时间早于当前时间，则立即执行task任务。

### 延时执行TimerTask的测试

TimerTask以队列的方式逐一按顺序执行，所以执行的时间有可能和预期的不一样，如果前面任务耗时较长，那么后面的任务运行时间可能被延迟。

### schedule (TimerTask task,Date firstTime,long period)

在指定日期之后按指定的间隔周期无限循环的执行一个任务

```
Timer timer = new Timer();  
timer.schedule(task, new Date(scheduleTime),4000)
```

如果计划时间早于当前时间，立即执行。

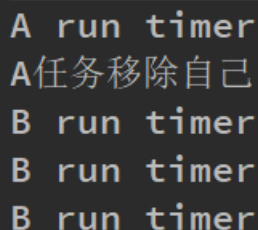
```
long scheduleTime = (nowTime-1000)
Time timer = new Timer();
timer.schedule(task,new Date(scheduleTime),4000)
```

### TimerTask类中的cancel方法

作用是将**自身**从任务队列中清除。

```
public class MyTaskA extends TimerTask {
    @Override
    public void run() {
        System.out.println("A run timer");
        this.cancel();
        System.out.println("A任务移除自己");
    }
}
public class MyTaskB extends TimerTask {
    @Override
    public void run() {
        System.out.println("B run timer");
    }
}
public class Test {
    public static void main(String[] args) {
        MyTaskA myTaskA = new MyTaskA();
        MyTaskB myTaskB = new MyTaskB();

        Timer timer = new Timer();
        timer.schedule(myTaskA,new Date(System.currentTimeMillis()),4000);
        timer.schedule(myTaskB,new Date(System.currentTimeMillis()),4000);
    }
}
```



```
A run timer
A任务移除自己
B run timer
B run timer
B run timer
```

### Timer类中的cancel方法

```
public class Test {
    public static void main(String[] args) throws InterruptedException {
        MyTaskA myTaskA = new MyTaskA();
        MyTaskB myTaskB = new MyTaskB();
```

```

Timer timer = new Timer();
timer.schedule(myTaskA,new Date(System.currentTimeMillis()),4000);
timer.schedule(myTaskB,new Date(System.currentTimeMillis()),4000);
Thread.sleep(5000);

```

**timer.cancel();//全部任务取消**

```

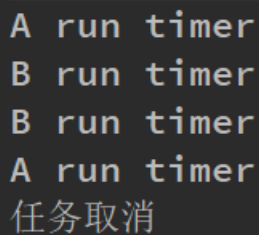
System.out.println("任务取消");

```

```

}
}

```



```

A run timer
B run timer
B run timer
A run timer
任务取消

```

## 间隔执行Task任务的算法

当队列中有3个任务ABC,每一次将最后一个任务放到队头，再执行队列头中的Task的run方法。

(1) ABC

(2) CAB

(3) BCA

```

public class Test {
    public static void main(String[] args) throws InterruptedException {
        MyTaskA myTaskA = new MyTaskA();
        MyTaskB myTaskB = new MyTaskB();
        MyTaskC myTaskC = new MyTaskC();
        Timer timer = new Timer();
        timer.schedule(myTaskA,new Date(System.currentTimeMillis()),4000);
        timer.schedule(myTaskB,new Date(System.currentTimeMillis()),4000);
        timer.schedule(myTaskC,new Date(System.currentTimeMillis()),4000);

        Thread.sleep(Integer.MAX_VALUE);
    }
}

public class MyTaskC extends TimerTask {
    @Override
    public void run() {
        System.out.println("C run timer");
    }
}

```

### **schedule(TimerTask task,long delay)方法的测试**

该方法的作用是以执行该方法的当前时间为参考时间，往后延迟指定的毫秒数后执行TimerTask任务。

### **schedule(TimerTask task,long delay,long period)方法的测试**

该方法的作用是以执行该方法的当前时间为参考时间，往后延迟指定的毫秒数再以某一间隔时间无限次数的执行TimerTask任务。

### **scheduleAtFixeRate(TimerTask task,Date firstTime,long period)**

schedule和scheduleAtFixedRate方法的区别在于有没有追赶特性。

### **schedule方法任务不延时——Date类型**

```
public class Test{
    static class MyTask extends TimerTask {
        @Override
        public void run() {
            try{
                System.out.println("begin timer = "+System.currentTimeMillis());
                Thread.sleep(1000);
                System.out.println("end timer = "+System.currentTimeMillis());
            }catch (InterruptedException e){
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        MyTask myTask = new MyTask();
        long nowTime = System.currentTimeMillis();
        Timer timer = new Timer();
        timer.scheduleAtFixedRate(myTask,new Date(nowTime),3000);
    }
}
```

```
begin timer = 1608704615881
end timer = 1608704616881
begin timer = 1608704618881
end timer = 1608704619881
begin timer = 1608704621881
end timer = 1608704622881
```

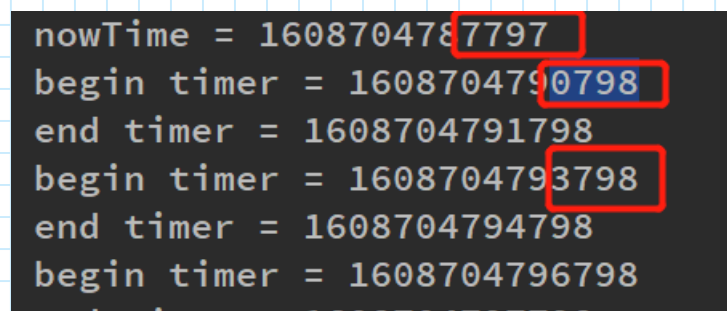
在不延时的情况下，如果任务没有被延时执行，则下一次执行任务的开始时间是上一次任务的开始时间加上period时间。

不延时是指执行任务的时间小于period间隔时间。

### schedule方法任务不延时——long类型

```
public class Test2 {
    static class MyTask extends TimerTask {
        @Override
        public void run() {
            try{
                System.out.println("begin timer = "+System.currentTimeMillis());
                Thread.sleep(1000);
                System.out.println("end timer = "+System.currentTimeMillis());
            }catch (InterruptedException e){
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        MyTask myTask = new MyTask();
        long nowTime = System.currentTimeMillis();
        System.out.println("nowTime = "+System.currentTimeMillis());
        Timer timer = new Timer();
        timer.scheduleAtFixedRate(myTask,3000,3000);
    }
}
```



The screenshot shows the output of the Java program. It displays a series of timestamps for 'begin timer' and 'end timer' events. The first 'begin timer' is at 1608704787797. The first 'end timer' is at 1608704790798. The second 'begin timer' is at 1608704793798. The second 'end timer' is at 1608704794798. The third 'begin timer' is at 1608704796798. The intervals between the 'begin' and 'end' events are approximately 3000 milliseconds, demonstrating the 'not delayed' behavior of the `scheduleAtFixedRate` method.

在不延时的情况下，第一次执行任务的时间是任务开始时间加delay时间，接下来执行任务的时间是上一次任务的开始时间加period时间。

### schedule方法任务延迟——Date类型

### schedule方法任务延迟——long类型

在延迟的情况下，任务被延迟执行，那么下一次任务的执行时间参考的是上一次任务



“结束”时的时间来开始的。

**scheduleAtFixeRate方法四种情况和上面一致。**那两个方法有什么区别呢，就是是否具有追赶执行性。

schedule方法不具有追赶执行性

```
public class Test9 {
    static class MyTask extends TimerTask {
        @Override
        public void run() {
            System.out.println("begin timer = "+System.currentTimeMillis());
            System.out.println("end timer = "+System.currentTimeMillis());
        }
    }

    public static void main(String[] args) {
        MyTask myTask = new MyTask();
        long nowTime = System.currentTimeMillis();
        System.out.println("nowTime = "+System.currentTimeMillis());
        long runTime = nowTime - 10000;
        System.out.println("runTime = "+runTime);
        Timer timer = new Timer();
        timer.schedule(myTask,new Date(runTime),2000);
    }
}
```

在nowTime和runTime之间的Task任务被取消，不被执行，说明Task任务不具有追赶性。

```
nowTime = 1608706197855
runTime = 1608706187855
begin timer = 1608706197857
end timer = 1608706197857
begin timer = 1608706199858
end timer = 1608706199858
begin timer = 1608706201858
end timer = 1608706201858
begin timer = 1608706203859
end timer = 1608706203859
```

scheduleAtFixedRate方法具有追赶执行性

```
public class Test10 {
```

```

static class MyTask extends TimerTask {
    @Override
    public void run() {
        System.out.println("begin timer = "+System.currentTimeMillis());
        System.out.println("end timer = "+System.currentTimeMillis());
    }
}

public static void main(String[] args) {
    MyTask myTask = new MyTask();
    long nowTime = System.currentTimeMillis();
    System.out.println("nowTime = "+System.currentTimeMillis());
    long runTime = nowTime - 10000;
    System.out.println("runTime = "+runTime);
    Timer timer = new Timer();
    timer.scheduleAtFixedRate(myTask,new Date(runTime),2000);
}
}

```

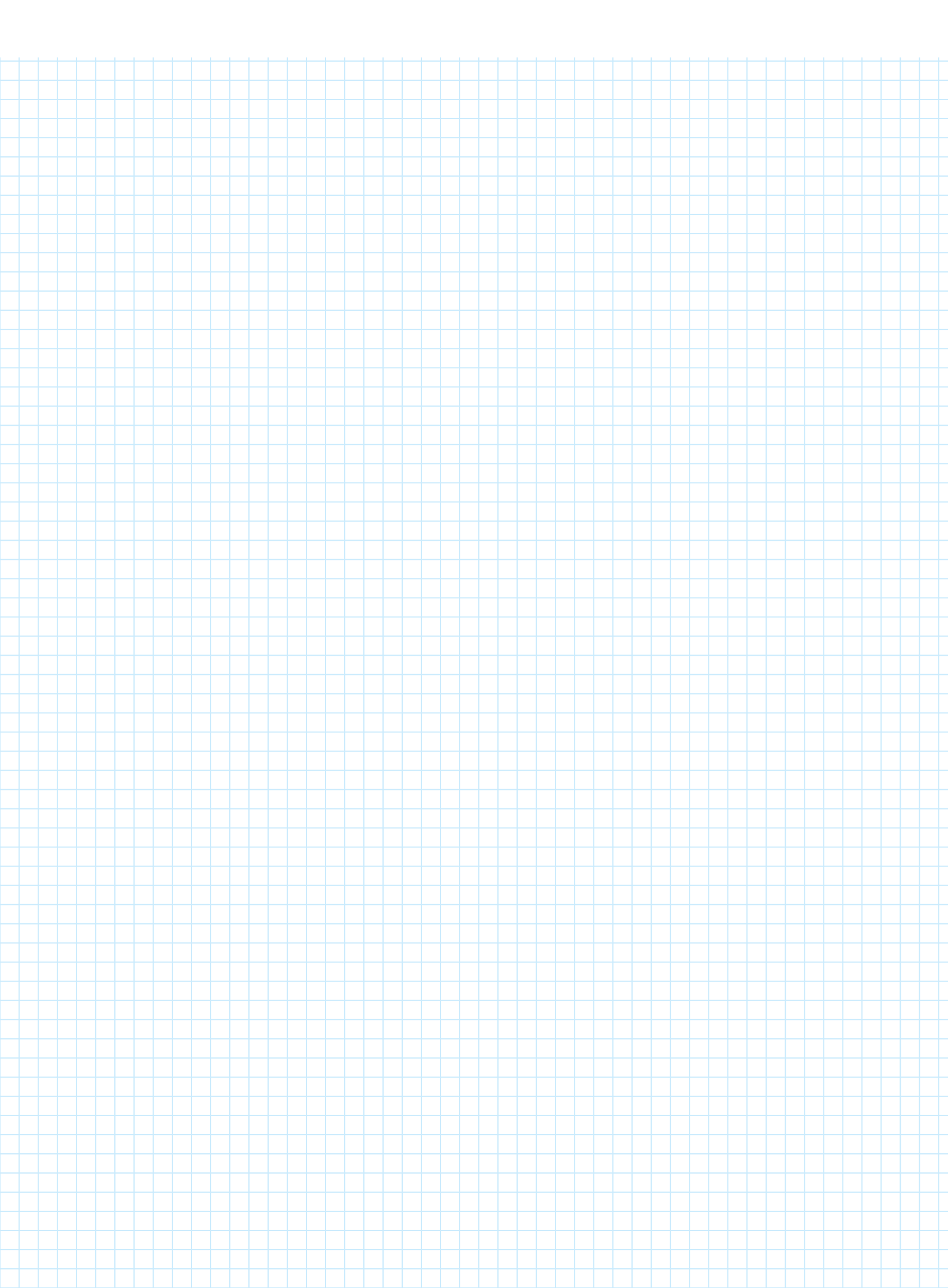
```

nowTime = 1608706257908
runTime = 1608706247907
begin timer = 1608706257910
end timer = 1608706257910
begin timer = 1608706257910
end timer = 1608706257910
begin timer = 1608706257910
end timer = 1608706257910
begin timer = 1608706257910
end timer = 1608706257910
begin timer = 1608706257910
end timer = 1608706257910
begin timer = 1608706257910
end timer = 1608706257910
begin timer = 1608706259908
end timer = 1608706259908

```

会将之前没有执行的任务追加执行，将10s之内的都输出完，后续再每隔2s执行一次任务。

将两个时间段内的时间所对应的Task任务被“弥补”执行，也就是在指定时间段内的运行次数比较运行完整。这就是追赶性。



## 第6章 单例模式和多线程

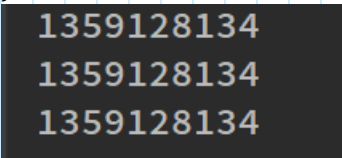
2020年12月23日 14:53

### 立即加载/饿汉模式

指使用类的时候已经将对象创建完毕，常见办法就是直接new实例对象。

在这个模式中，调用方法之前，实例已经被工厂创建了。

```
public class MyObject {  
    private static MyObject myObject = new MyObject();  
    private MyObject(){  
  
    }  
  
    public static MyObject getInstance(){  
        return myObject;  
    }  
}  
  
public class MyThread extends Thread {  
    @Override  
    public void run() {  
        System.out.println(MyObject.getInstance().hashCode());  
    }  
}  
  
public class Run {  
    public static void main(String[] args) {  
        MyThread t1 = new MyThread();  
        MyThread t2 = new MyThread();  
        MyThread t3 = new MyThread();  
        t1.start();  
        t2.start();  
        t3.start();  
  
    }  
}
```



```
1359128134  
1359128134  
1359128134
```

hashCode是同一个值，说明对象是同一个。实现了立即加载单例模式。

**以下各种方式都会有一个私有的构造器防止外部类以new的方式**

## 创建对象。

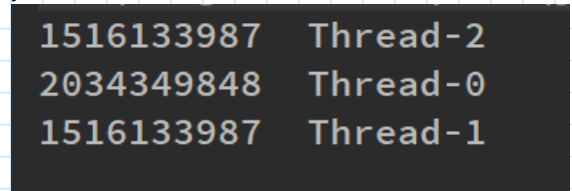
### 延迟加载/懒汉模式

延迟加载是指调用get方法时实例才被工厂创建，常见的实现方法是在get方法中进行new实例化。

```
public class MyObject {
    private static MyObject myObject;
    private MyObject(){

    }

    public static MyObject getInstance(){
        if (myObject == null) myObject = new MyObject();
        return myObject;
    }
}
```



```
1516133987 Thread-2
2034349848 Thread-0
1516133987 Thread-1
```

在多线程环境下会出现取出多个实例的情况，不符合单例模式。

这是由于多个线程进入getInstance方法创建实例导致的。

解决方法：

1. 对getInstance方法加synchronized关键字，可以得到相同的实例。但是运行效率很低。
2. 改成同步代码块synchronized (MyObject.class) {}，可以得到相同实例，但是运行效率也非常低，和上面的方法是一样的。

### 使用DCL机制

DCL double-check locking 双锁机制来实现多线程环境中的延迟加载单例模式。

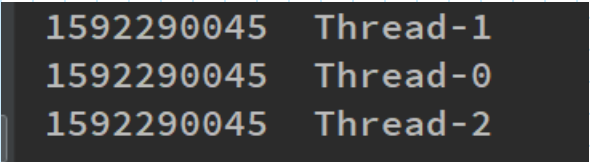
```
public static Singleton getInstance()
{
    if (instance == null)
    {
        synchronized(Singleton.class) { //1
            if (instance == null) //2
                instance = new Singleton(); //3
        }
    }
    return instance;
}
```

双重检查锁定背后的理论是：在 //2 处的第二次检查使（如清单 3 中那样）创建两个不同的 Singleton 对象成为不可能。

```
public class MyObject {
    private volatile static MyObject myObject;

    public static MyObject getInstance(){
        try{
            if(myObject != null){

            }else{
                Thread.sleep(3000);
                synchronized (MyObject.class){
                    if(myObject == null){
                        myObject = new MyObject();
                    }
                }
            }
        }catch (InterruptedException e){
            e.printStackTrace();
        }
        return myObject;
    }
}
```



```
1592290045 Thread-1
1592290045 Thread-0
1592290045 Thread-2
```

### DCL使用volatile的必要性

使用DCL机制成功解决了懒汉模式约到多线程的问题。也是大多数多线程结合单例模式使用的解决方案。

使用volatile修改变量myObject使该变量在多个线程间达到可见性，另外也禁止了myObject = new myObject()代码重排序。

myObject = new myObject()在内部 分为三个步骤

- (1) memory=allocate();//分配对象的内存空间
- (2) ctorInstance(memory);//初始化对象
- (3) myObject = memory;//设置instance指向刚分配的内存地址。

JIT编译器有可能将这个三个步骤重排序为：

- (1) memory=allocate();//分配对象的内存空间
- (2) myObject = memory;//设置instance指向刚分配的内存地址。



(3) ctorInstance(memory);//初始化对象

这时会出现，虽然构造方法还带有执行，但myObject对象具有了内存地址，值不是null。当访问myObject对象中的变量时还是数据类型的默认值。

Java的concurrent包里面的CountDownLatch其实可以把它看作一个计数器，只不过这个计数器的操作是原子操作，同时只能有一个线程去操作这个计数器，也就是同时只能有一个线程去减这个计数器里面的值。用来控制线程执行的顺序。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。

1、CountDownLatch end = new CountDownLatch(N); //构造对象时候 需要传入参数N

2、end.await() 能够阻塞线程 直到调用N次end.countDown() 方法才释放线程

3、end.countDown() 可以在多个线程中调用 计算调用次数是所有线程调用次数的总和

```
public class Test {
    public static void main(String[] args) throws InterruptedException {
        for (;) {
            CountDownLatch latch = new CountDownLatch(1);
            CountDownLatch end = new CountDownLatch(100);
            for (int i = 0; i < 100; i++) {
                Thread t1 = new Thread() {
                    @Override
                    public void run() {
                        try {
                            latch.await();

                            OneInstanceService one = OneInstanceService.getTest();
                            if (one.i_am_has_state == 0) { //等于0说明没有经过构造器
                                System.out.println("one.i_am_has_state == 0进程结束");
                                System.exit(0);
                            }
                            end.countDown();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                };
            }
        }
    }
};
t1.start();
```

```

    }
    latch.countDown();
    end.await();
    OneInstanceService.reset();
}
}
}
one.i_am_has_state == 0进程结束

```

添加VM参数-server更容易获得预期结果。说明确实发生了重排序。

加上volatile后不会再输出任何信息，说明禁止重排序后，i\_am\_has\_state永远不是0了。

### 使用静态内置类实现单例模式

```

public class MyObject {
    //静态内部类
    private static class MyObjectHandler{
        private static MyObject myObject = new MyObject();
    }

    public static MyObject getInstance(){
        return MyObjectHandler.myObject;
    }
}

```

```

754568537 Thread-0
754568537 Thread-1
754568537 Thread-2

```

### 序列化和反序列化

当将单例的对象进行序列化时，使用默认反序列行为取出的对象是多例的。

```

public class MyObject implements Serializable {
    private static final long serialVersionUID = 888L;

    public static Userinfo userinfo = new Userinfo();
    private static MyObject myObject = new MyObject();
}

```

```

private MyObject(){
}
public static MyObject getInstance(){
    return myObject;
}
}

public class SaveAndRead {
    public static void main(String[] args) throws IOException,
    ClassNotFoundException {
        MyObject myObject = MyObject.getInstance();
        System.out.println("序列化-myobject= " + myObject.hashCode() + "
userinfo " + MyObject.userinfo.hashCode());
        FileOutputStream fosRef = new FileOutputStream(new File("mytest.txt"));
        ObjectOutputStream oosRef = new ObjectOutputStream(fosRef);
        oosRef.writeObject(myObject);
        oosRef.close();
        fosRef.close();

        FileInputStream fisRef = new FileInputStream(new File("mytest.txt"));
        ObjectInputStream iosRef = new ObjectInputStream(fisRef);
        MyObject myObject1 = (MyObject)iosRef.readObject();
        iosRef.close();
        fisRef.close();
        System.out.println("序列化-myobject= " + myObject1.hashCode() + "
userinfo " + myObject1.userinfo.hashCode());

    }
}

```

```

序列化-myobject= 1163157884 userinfo 1956725890
序列化-myobject= 1828972342 userinfo 1956725890

```

在反序列化时创建了新的MyObject对象，内存中产生了两个MyObject对象。但userinfo对象得到复用。

解决办法就是在反序列化时使用readResolve方法，对原有的MyObject对象进行复用。

当JVM从内存中反序列化地"组装"一个新对象时，就会**自动**调用这个 readResolve方法来返回我们指定好的对象了，单例规则也就得到了保证。

```

public class MyObject implements Serializable {
    private static final long serialVersionUID = 888L;

    public static Userinfo userinfo = new Userinfo();
    private static MyObject myObject = new MyObject();

    private MyObject(){

    }
    public static MyObject getInstance(){
        return myObject;
    }

    protected Object readResolve(){
        System.out.println("调用了readResolve方法");
        return MyObject.myObject;
    }
}

```

如果将序列化和反序列化操作分别放入两个class中，则反序列化时会产生新的对象。放在两个class类中分别执行相当于创建了两个JVM虚拟机，每个虚拟机里面的确是有一个MyObject对象，我们想要的是在一个JVM虚拟机中进行序列化和反序列化时保持MyObject单例性的效果，而不是创建两个JVM虚拟机。

### 使用static代码块实现单例模式

静态代码块中的代码在使用类的时候就已经执行。

```

public class MyObject {
    private static MyObject instance = null;
    private MyObject(){

    }

    static {
        instance = new MyObject();
    }
    public static MyObject getInstance(){
        return instance;
    }
}

public class Run {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
    }
}

```

```
MyThread t3 = new MyThread();  
t1.start();  
t2.start();  
t3.start();  
  
}  
}
```

```
1871638897  
1871638897  
1871638897  
1871638897  
1871638897  
1871638897  
1871638897
```

### 使用enum枚举数据类型实现单例模式

enum枚举数据类型的特性和静态代码块相似，在使用枚举类时，构造方法会自动被调用，可以应用这个特性实现单例模式。

# 第7章

2020年12月23日 21:42

## 线程状态

NEW：刚new完至今尚未启动的线程

RUNNABLE：正在java虚拟机中执行的线程

BLOCKED：受阻塞并等待某个监视器锁的线程处于这种状态。

WAITING：无期限的等待另一个线程来执行某一特定操作的线程处于这种状态。执行了Object.wait()后所处的状态。

TIMED\_WAITING：等待另一个线程来执行取决于指定等待时间的操作的线程处于这种状态 代表线程执行了Thread.sleep(1000)方法。

TERMINATED：已退出的。

## 线程组

线程对象关联线程组：一级关联

一级关联就是父对象中有子对象，但不创建子孙对象。

```
public class Run {
    public static void main(String[] args) {
        ThreadA arunnable = new ThreadA();
        ThreadB brunnable = new ThreadB();
        ThreadGroup threadGroup = new ThreadGroup("xjf");
        Thread thread1 = new Thread(threadGroup,arunnable);
        Thread thread2 = new Thread(threadGroup,brunnable);
        thread1.start();
        thread2.start();

        System.out.println("活动的线程数: "+threadGroup.activeCount());

        System.out.println("线程组的名称: "+threadGroup.getName());
    }
}

public class ThreadA extends Thread {
    @Override
    public void run() {
        try{
            while(!Thread.currentThread().isInterrupted()){
                System.out.println("ThreadName= "+Thread.currentThread().getName());
                Thread.sleep(3000);
            }
        }catch (InterruptedException e){
            e.printStackTrace();
        }
    }
}
```



```
}
```

线程对象关联线程组：多级关联  
略

### 线程组自动归属特性

就是没有指定所属线程组，则线程组自动归到当前线程对象所属的线程组中。就是以隐式的方式在一个线程组中添加了一个子线程组。

### 获取根线程组

```
Thread.currentThread().getThreadGroup().getParent()
```

main线程所在的线程组名为main

main线程所在的线程组的父线程是system。

JVM的根线程组是system，再取父线程组则空异常。

### 线程组中加线程组

```
ThreadGroup newGroup = new  
ThreadGroup(Thread.currentThread().getThreadGroup(),"newgroup");
```

### 组内的线程批量停止

调用ThreadGroup的interrupt方法可以中断该组所有正在运行的线程。

### Thread.activeCount()方法

返回**当前线程**的线程组中活动线程是数目。

### Thread.enumerate(Thread tarray[])方法

将**当前线程**的线程组及其子组中的每一个活动线程复制到**指定的数组**中。

enumerate(Thread[] list)方法实际上就是enumerate(Thread[] list,true)的方法。事实上，true的情况是将所有子group中的active线程都递归到Thread数组中，而false的情况则是将ThreadGroup(父线程组)中的active线程全部复制到Thread数组中。**我们通常称true的情况是递归方法，而false的方法是递归方法。**

### 线程出现异常的处理

当有多个线程需要处理异常时，对每一个线程的run方法都要用catch处理，代码冗余。可以使用setDefaultUncaughtExceptionHandler方法和setUncaughtExceptionHandler方法来几种处理线程的异常。UncaughtExceptionHandler接口的主要作用是当线程出现异常时，JVM捕获到自动调用UncaughtExceptionHandler接口中的void uncaughtException(Thread t, Throwable e)来处理异常。

### 使用setUncaughtExceptionHandler()方法进行异常处理

```
public class Main2 {  
    public static void main(String[] args) {  
        MyThread myThread = new MyThread();  
        myThread.setName("t1");  
        myThread.setUncaughtExceptionHandler(new  
Thread.UncaughtExceptionHandler() {  
            @Override  
            public void uncaughtException(Thread t, Throwable e) {  
                System.out.println("线程" + t.getName() + "出现异常");  
                e.printStackTrace();  
            }  
        });  
        myThread.start();  
    }  
}
```

setUncaughtExceptionHandler()的作用是对**指定的线程对象**设置默认的异常处理器。

### 使用setDefaultUncaughtExceptionHandler()方法进行异常处理

是对指定线程类的**所有线程对象**设置默认的异常处理器。

```
public class Main3 {  
    public static void main(String[] args) {  
        MyThread.setDefaultUncaughtExceptionHandler(new  
Thread.UncaughtExceptionHandler() {  
            @Override  
            public void uncaughtException(Thread t, Throwable e) {  
                System.out.println("线程 " + t.getName() + " 出现异常 ");  
                e.printStackTrace();  
            }  
        });  
        MyThread t1 = new MyThread();  
        MyThread t2 = new MyThread();  
        t1.start();  
        t2.start();  
    }  
}
```

```
}
```

### 线程异常处理的优先性

如果调用过`setUncaughtExceptionHandler`方法，则此异常处理器优先处理，其他异常处理器不再处理。