

Java 多线程编程

深入详解

(Thread Programming)

Version1.0

作者	汪文君
版本	Version1.0
时间	2012/3/1
QQ	532500648

更改履历

类型	操作人	时间	内容
创建	汪文君	2012/3/1	编写 Java 多线程的详细描述文档，主要将自己工作中和学习中对多线程的理解加以总结和汇总

目录

目录.....	3
感谢.....	6
题记.....	7
本书大纲.....	8
参考资料.....	8
如何阅读本书.....	8
第一章 多进程多线程概述.....	9
第一节 什么是多进程多线程.....	9
1.1.1 什么是进程.....	9
1.1.2 什么是多进程.....	9
1.1.3 什么是线程.....	9
1.1.4 什么是多进程.....	9
1.1.5 咬文嚼字.....	10
第二节 Java 对多线程的支持.....	10
第三节 第一个多线程程序.....	11
1.3.1 没有真正意义上的多线程.....	12
1.3.2 纠结的思考.....	12
1.3.3 线程初探总结.....	12
第二章 多线程详解.....	13
第一节 继承 Thread 创建线程.....	13
2.1.1 你的疑惑曾经是我的疑惑.....	14
2.1.2 父类实现算法，子类实现细节.....	14
2.1.3 Thread 中的 Template Desgin.....	16
2.1.4 多线程编程.....	16
第二节 线程的状态.....	18
2.2.1 线程的初始化.....	18
2.2.2 线程的运行状态.....	19
2.2.3 线程的冻结状态.....	19
2.2.4 线程的死亡状态.....	19
2.2.5 深入探讨.....	19
第三节 通过实现 Runnable 接口创建线程.....	20
2.3.1 银行排队叫号程序第一版.....	20
2.3.2 银行排队叫号程序第二版.....	21
2.3.3 通过 Runnable 接口实现第三版.....	23
2.3.4 Runnable 和 Thread 的区别.....	24
2.3.5 线程中的策略模式.....	24
2.3.6 Runnable 接口的用法总结.....	27
2.3.7 查缺补漏.....	27
第三章 线程的同步.....	28

第一节	同步代码块.....	30
3.1.1	给共享数据加锁.....	31
3.1.2	同步代码块的有效范围.....	32
3.1.3	同步代码块的详解.....	32
3.1.4	该如何定义一个锁.....	33
第二节	同步方法.....	33
3.2.1	同步重构方法.....	34
3.2.2	同步 run 方法.....	35
3.2.3	同步总结.....	35
第三节	this 锁与 static 锁.....	36
3.3.1	this 锁.....	36
3.3.2	static 锁.....	40
第四节	线程的休眠.....	42
第五节	单例模式的详解.....	42
3.5.1	饿汉式单例模式.....	43
3.5.2	懒汉式单例模式.....	44
第六节	死锁.....	46
3.6.1	什么是死锁.....	47
3.6.2	死锁程序的模拟.....	47
3.6.3	如何避免死锁.....	49
第四章	线程间的通讯.....	49
第一节	生产者消费者.....	49
4.1.1	简易版生产者消费者.....	49
4.1.2	改进版生产者消费者.....	51
4.1.3	细说生产者消费者.....	54
第二节	多线程下的生产者消费者.....	55
4.2.1	多线程下的生产者消费者遇到的问题.....	55
4.2.2	多线程下的生产者消费者改进版.....	60
第三节	Object 类 wait, notify, notifyAll 详解.....	64
4.3.1	wait 详解.....	64
4.3.2	notify 详解.....	64
4.3.3	notifyAll 详解.....	64
4.3.4	实现秒表程序.....	65
4.3.5	小节.....	71
第五章	守护线程与线程的优先级.....	71
第一节	守护线程.....	71
第二节	线程的 yield.....	72
第三节	线程的停止.....	73
第四节	线程的优先级.....	73
第五节	线程 Join.....	74
第六节	线程的 interrupt.....	75
第六章	线程池的实现.....	75
第一节	线程组.....	75

6.1.1	什么是线程组.....	75
1、	线程组创建方式一.....	75
2、	线程组创建方式二.....	77
6.1.2	线程组的 API 详解.....	77
1、	获取线程组中活跃的线程.....	77
2、	将线程组的线程拷贝到线程数组中.....	78
3、	其他 API.....	80
第二节	线程池雏形.....	83
第三节	最大最小属性.....	86
6.3.1	最小线程数.....	87
6.3.2	最大线程数.....	87
6.3.3	最大活跃线程数.....	87
6.3.4	属性之间的关系.....	87
6.3.5	加入属性之后的线程池实现.....	88
第四节	任务队列属性.....	97
第七章	线程状态的监控.....	99
第一节	线程状态的监控.....	99
7.1.1	线程状态监控接口.....	99
7.1.2	Runnable 的封装.....	100
7.1.3	测试代码.....	102
7.1.4	详解设计思路.....	103
第二节	线程出现异常捕获.....	103
第八章	总结.....	104
第九章	近期推出.....	104
Programming	系列丛书附录.....	104

感谢

在编写该书的过程中,我得到了很多人的帮助,有些是我认识的,有些则是素未蒙面的,本书的初稿形成之后我将该书发给我之前的导师,现任 Neusoft-mid 部门架构师,他给了我很多的建议和思路;

坦白的讲学习每一门语言,我总觉得线程或者进程部分总是最复杂的,也是最最能够考量一个人编程功底,思路,见识的技术,所以,总结这本书时我本身就怀着诚惶诚恐的心情去完成,自始至终都不敢自认完全可以驾驭和掌握多线程编程,因为他的确确是很复杂,也许 Java 中提供的 API 并没有多少,但是,他真的是一门很复杂的技术领域。

除此之外,我还应该很庆幸我读到了一本 Java 线程最好的书籍,那就是来自 orally 的一本经典书籍《Java Thread》,除了对作者驾驭 Thread 的能力敬仰之外,更加人外作者能够抓住读者需要什么,由浅入深的去讲解;

感谢国内外的关于 Thread 的论坛,文章,程序代码,感谢五年程序员生涯我所接触过的有关 Thread 的编程,让我有了体会,有了理解,有了可以和别人交流的资本;

题记

转眼间，这已经是我第七本电子书编写，从工作第二年开始，我已经慢慢的养成了这样一个习惯，总结自己感觉掌握的差不多的技术，然后通过自己的方式作为记录，并且尽量用平实的语言将它写出来，既能起到总结自己的目的，又能做到引导初学者，还能起到以文交友的目的，希望通过图书的编写，不断的认识圈内的好学者，然后不断的交流也对我自己是一种提高。

这本电子书是我写的最长的一次，每次都要设计最能说明问题的例子然后慢慢的阐述其中所涉及的知识点，这个过程的确是一个比较熬人，但是每次通过代码的 debug 和结果的不断验证，推断自己的预期，这种小小的喜悦也是编写这本书给我带来的最大享受，希望本书能有幸被你读到，如果不介意，可以添加本书中留下的联系方式，和我交流，我愿意接受任何批评；

本书大纲

通过本书您可以了解到如下的知识点：

- ✚ 如何创建线程
- ✚ 线程的几个状态
- ✚ 资源锁
- ✚ 线程间的通信
- ✚ 线程组的知识
- ✚ 线程池的知识
- ✚ 单例模式的深入详解
- ✚ 生产者消费者模式

参考资料

- ✚ 《Java Thread》
- ✚ 《Java 并发编程》
- ✚ 《Core Java》
- ✚ 《Thinking Java》

如何阅读本书

- ✚ 本文中涉及的代码会以浅蓝色为背景色的表格
- ✚ 本文中涉及的输出会以黑色背景绿色字体的表格
- ✚ 本文中需要重点说明的地方会以**粗体字**的方式进行显示

第一章 多进程多线程概述

第一节 什么是多进程多线程

多进程多线程的具体概念到底是如何，我也基本上告别操作系统那本书很久了，所以也不会有官方的答案，但是比较山寨通俗化的理解还是有的，至于官方正式的描述读者可以自行查阅相关资料

1.1.1 什么是进程

进程就是 CPU 的执行路径，呵呵，这个也许也有些抽象，那我们就来一个更为通俗的说法，进程就是系统运行中的程序，比如说我们打开浏览器，打开 word，打开魔兽世界，他们就是程序，他们就是所谓的 CPU 执行路径，当然他们也就是进程，如果还不明白，那你就打开任务管理器，看看进程那一栏就明白了；

1.1.2 什么是多进程

多进程就是系统中可以同时运行多个程序，比如我们在浏览网页的时候可以打开 office，将网页中的内容粘贴到 word 之中，我们同时运行了浏览器和 word，这就是所谓的多进程，其实没有真正意义上的多进程，我们在后文中在进行讲解；先可以这么理解和认为；当然这是不够严谨的哦！

1.1.3 什么是线程

线程就是运行在进程当中的运行单元，比如说我们打开迅雷（它是一个运行程序，因此它是一个进程），我们下载的某个任务他就是一个线程；

1.1.4 什么是多进程

同样的道理，每个进程里面有多个独立的或者相互有协作关系的运行单元我们就称之为多线程，比如说我们可以通过迅雷同时下载多个文件，并且还可以上传某个文件，当然也没

有严格意义的多线程（多核 CPU 就另当别论了）

1.1.5 咬文嚼字

进程英文单词 **process**，有运行的意思，顾名思义，他必须是运行着的才能称之为进程；

线程英文单词 **thread**，有丝线的意思，就是颗粒很细，力度很小，因此他要依附于进程，所以我们可以姑且这样认为，没有进程肯定谈不上有线程；

第二节 Java 对多线程的支持

我们都知道，在任何一门语言中，程序的执行顺序都是顺序执行的，也就是说执行完 A 部分的代码片段之后才能执行 B 代码片段，当然多线程的情况除外，请看下面的代码；

```
package com.wenhuisoft.chapter;

public class ThreadSupport
{
    public static void main(String[] args)
    {
        int i = 100;
        while(i>0)
        {
            System.out.println("current i value is:"+i--);
        }

        System.out.println("=====");
        while(i<100)
        {
            System.out.println("current i value is :"+i++);
        }
    }
}
```

只有在结束了第一个循环之后才能进入第二个循环，两者并不能同时执行，如果我们引入多线程的概念，该问题将迎刃而解，Java 对多线程的支持和封装是目前所有语言中做的最好的，它让开发人员远离了 C 程序员那些信号量等繁琐的编程方式，只关注共享的数据，关注业务逻辑单元，不去关注细节；

Java 中多线程的实现方式有两种，这里只做一个简单的介绍，在后文中将会详细的讲解

这两种的具体实现方式和优缺点，以及工作中如何是使用线程；

- ✚ 继承 Thread 父类；
- ✚ 实现 Runnable 接口；

其实 JDK 的编码过程中涉及到了很多个设计模式，其中 Thread 就用到了 template design pattern 和 strategy design pattern 在后文中我会一一详细讲解，并且教大家如何使用这两个设计模式，这两个模式也是我本人非常喜欢和常用的设计思想；

第三节 第一个多线程程序

接着上面的示例,我们快速实现一个多线程的实现，来看看我们的程序如何交替执行从 1~100 然后从 100~1 的输出，其中的一些技术细节我们会慢慢的讨论，如果看不懂不要着急，我都会逐一进行讲解；

```
package com.wenhuisoft.chapter;

public class FirstThread
{
    public static void main(String[] args)
    {
        new Thread(new Runnable()
        {
            public void run()
            {
                int i = 0;
                while(i<100)
                {

                    System.out.println(Thread.currentThread().getName()+":"+i++);

                }

            }
        }).start();

        int i = 100;
        while(i>0)
        {
            System.out.println(Thread.currentThread().getName()+":"+i--);

        }

    }
}
```

```
}
```

读者可以运行上面的例子，发现程序实现了交替打印的功能，但是我们的目的不止于此，我们需要对其进行深究；

1.3.1 没有真正意义上的多线程

了解 CPU（单核）的人都知道，CPU 在同一个时刻只能给一个程序分配资源，也就是赋予一个程序运行权，那么我们看到一次能运行好几个程序其实是 CPU 来回切换执行权，所以让别人以为是并发运行，只是切换的速度很快（取决于 CPU 的主频）所以没有真正意义上的并发；

1.3.2 纠结的思考

其实刚才的程序我们看到的已经有两个线程的存在了，我之前说过，一个进程至少有一个执行单元，可以理解为至少有一个线程在运行，那么 main 函数应该就是一个线程，因为它是程序的入口，然后我们又写了一个匿名类它显示的实现了 Runnable 接口，并且构造了一个 Thread 类，因此它也是一个线程，因此有两个线程在同时运行

问题往往没有那么简单，这也是本小节名字“纠结的思考”的来源，他真的是两个线程么？难道 JVM 不做些什么吗？最起码我们应该联想到它应该有一个后台线程负责管理堆栈信息管理垃圾的清理工作啊，因此上述的代码远远不止于一个线程，但是往往这样的钻牛角尖会让我们学习 Thread API 产生很多顾虑，因此我们可以暂且不用去管 JVM 在有多少个线程在支撑着我们的程序，但是我们最起码应该有这样的意识，这样也不至于在学习的路上浅尝辄止。

1.3.3 线程初探总结

- 🌈 Main 函数本身其实就是一个线程，我们称他为主线程；
- 🌈 实现多线程我们可以继承 Thread 类，也可以继承 Runnable 接口；
- 🌈 没有严格意义上的并发；
- 🌈 JVM 自身有很多后台线程在运行；

第二章 多线程详解

本章的内容应该是本书中最为重要的部分，因为我会循序渐进的引出多线程的创建，线程的生命周期，并且以一个多窗口出票程序来说明多线程如何工作，还有会有我在本书之前承诺大家的两个设计模式（Template，strategy）

第一节 继承 Thread 创建线程

通过翻阅 JDK API 我们发现文档中重点描述到，实现一个线程的一种方式成为 Thread 的子类，也就是继承 Thread，然后重写 run 方法，其中 run 方法是线程的执行代码片段；综合这段话，我们可以总结创建并运行一个线程有三个步骤：

- ✚ 继承 Thread 类；
- ✚ 重写 run 方法；
- ✚ 调用线程的 start 方法（启动线程，调用 run 方法）

其实这段代码我们之前已经写过了，但是为了说明问题，我们还是在写一个简单的线程来说明上述的三个观点

```
package com.wenhuisoft.chapter;

public class MyThread extends Thread
{
    @Override
    public void run()
    {
        int i = 0;
        while(i<60)
        {
            System.out.println("i:"+i++);
        }
    }

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

```
}
```

首先我们继承了 `Thread` 类，我们也重写了 `run` 方法，我们也在 `main` 方法中构造了一个 `Thread` 类然后调用了 `start` 方法；因此线程被创建并且被运行了

2.1.1 你的疑惑曾经是我的疑惑

为什么我重写的是 `run` 方法，却要调用 `start` 方法来启动它，我们如果直接调用线程实例的 `run` 方法不行么？可以，当然可以因为它是成员函数，调用当然是无可厚非的事情了，但是为什么他不代表启动了线程呢？也许看到这里你有这样的疑惑，我希望我通过我的认识和理解不仅能说清楚 `start` 做了什么，并且能告诉你一种设计方法；

2.1.2 父类实现算法，子类实现细节

在程序的设计中我们经常会将算法进行抽象，因为它有很多种运算的可能，所以我们为了更好地扩展，我们将算法进行了抽象，并且统一交给父类进行实现，子类只需要知道某个单元模块的功能即可，具体是如何穿插起来的子类不用去关心，我们实现一个输出图形的算法，然后有两种不同的实现，为了明显期间，我将所有的类写在一个文件中

```
package com.wenhuisoft.chapter;

abstract class Diagram
{
    protected char c ;

    public Diagram(char c)
    {
        this.c = c;
    }

    abstract protected void print(int size);

    abstract protected void printContent(String msg);

    public final void display(String msg)
    {
        int len = msg.getBytes().length;
        print(len);
    }
}
```

```
        printContent(msg);
        print(len);
    }
}

class StarDiagram extends Diagram{

    public StarDiagram(char c)
    {
        super(c);
    }

    @Override
    protected void print(int size)
    {
        for(int i = 0;i<size+2;i++)
        {
            System.out.print(c);
        }
        System.out.println();
    }

    @Override
    protected void printContent(String msg)
    {
        System.out.print("*");
        System.out.print(msg);
        System.out.println("*");
    }
}

public class TemplateTest
{
    public static void main(String[] args)
    {
        Diagram d1 = new StarDiagram('*');
        d1.display("wangwenjun");
    }
}
```

我们可以看到父类 `Diagram` 中的 `display` 方法规范了算法，也就是将打印上线边线和输出内容的部分进行了算法约束，我们不用关心他的算法逻辑，我们只需要实现他所抽象的两

个方法 `print` 和 `printContent` 方法，奇怪我们明明实现的是这两个方法，但是为什么运行的时候需要调用 `display` 方法呢？这样的问题是否似曾相识呢？是否在前面也同样有过这样的疑问呢？为什么我实现了 `Thread` 的 `run` 方法却要调用 `start` 方法才能启动它呢？也许看到这里你明白我想要说什么了，不过你可以假装暂时不知道，因为我后面还要进一步说明哦

上面的代码是一个不折不扣，如假包换的模板模式也就是 `template` 模式，是最常用也是最简单的一种设计模式，将程序的运行逻辑以及算法逻辑交由父类进行管理，子类只需要实现其中功能模块即可，但是上面有很多设计上的小技巧，需要作为重点的掌握和体会

🚦 为什么实现模块的抽象方法都是 `protected` 的呢？

因为我们不想让调用者关注到我们实现的细节，这也是面向对象思想封装的一个体现；

🚦 为什么 `display` 方法是不可继承的呢？

因为算法一旦确定就不允许更改，更改也只允许算法的所有者也就是他的主人更改，如果调用者都可通过继承进行修改，那么算法将没有严谨性可言；

2.1.3 Thread 中的 Template Desgin

已经说到这里了，想必各位已经很清楚为什么调用 `start` 才算是对线程的真正启动，才算是对 `run` 方法的线程级别调用？先别急着下结论，我们以事实说话，打开 `JDK` 源码我们一探究竟，看看真的是不是我们所说的那样呢？

打开代码发现 `start` 代码中调用了 `JNI` 函数 `start0`，他就用到了模板模式；读者可以自己查看源码看看；

2.1.4 多线程编程

其实多线程的代码我们在前面已经演示过了，我们创建了一个线程，并且在 `main` 方法中进行了运行，其实 `main` 方法是一个线程，创建的线程也是一个线程，因此这就是多线程，也不要想得太复杂，我们现在的目的就是两个代码逻辑块想要独立运行，交替执行；为了能更加的说明问题，我们重复上面的代码只不过这次比较具体一些

代码片段如下：

我们的目的就是想要做到程序中有两个独立的运行单元同时运行，体现的执行结果是交替输出


```
package com.wenhuisoft.chapter;

class ThreadTest extends Thread
{
    private final static int DEFAULT_VALUE = 100;

    private int maxValue = 0;

    private String threadName = "";

    public ThreadTest(String threadName)
    {
        this(threadName, DEFAULT_VALUE);
    }

    public ThreadTest(String threadName, int defaultValue)
    {
        this.maxValue = defaultValue;
        this.threadName = threadName;
    }

    @Override
    public void run()
    {
        int i = 0;
        while(i<maxValue)
        {
            i++;
            System.out.println("Thread:"+threadName+": "+i);
        }
    }
}

public class MultThreadDemo
{
    public static void main(String[] args)
    {
        ThreadTest t1 = new ThreadTest("t1");
        ThreadTest t2 = new ThreadTest("t2", 200);
        t1.start();
        t2.start();
    }
}
```

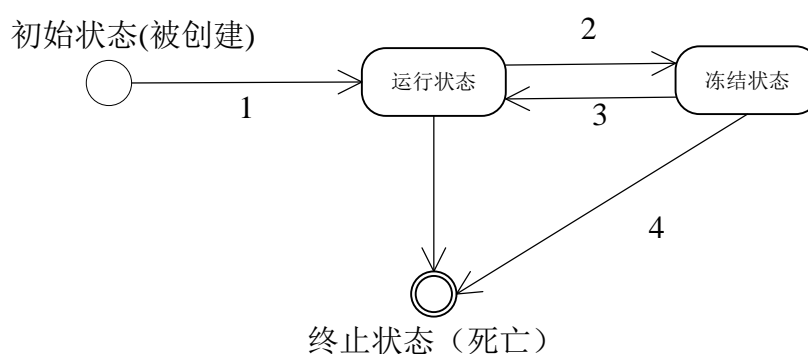
为了节省篇幅，我将出现交替运行的地方粘贴出来即可

```
Thread:t2:75
Thread:t1:76
Thread:t2:76
Thread:t1:77
Thread:t2:77
```

上面的程序就是多线程程序，上面的线程数有多少个呢？答案显而易见是三个，因为我们还有一个主线程在运行；

第二节 线程的状态

线程有他自己的状态，也就是我们所说的生命周期，简言之就是现在线程是个什么情况，其实粗粒度的划分线程大致上有四个基本状态，当然还可以进行进一步的细化，我们在本章中将会重点讨论这个问题，并且会以图表的方式进行讲述；



通过上面的图示我们可以大致看到，线程有四个基本的状态，那就是：

- ✚ 初始化（被创建）
- ✚ 运行状态
- ✚ 冻结状态
- ✚ 终止状态（死亡）

本节我们的主要任务就是说清楚这四种状态是如何切换，以及他们中间的一些细节知识

2.2.1 线程的初始化

线程的初始化状态就是我们所说的创建了一个线程，也就是说实例化了一个 `Thread` 的子类，就等着被 `start`，初始化状态应该是很容易理解的状态；

2.2.2 线程的运行状态

线程的运行状态就是我们当创建完线程之后，显式的调用了 `start` 方法，此时线程就处于运行状态，可是实际是这样的么？这就要看 CPU 的脸色了，因此我刚才的说法只能说对了一半，但是不够严谨，线程被 `start` 之后并不一定会马上运行，因此还有一个中间状态叫做临时状态我之所以没有在上图中画，是因为我觉得这个状态可以不用太多的关注，所谓临时状态就是指，在 CPU 的执行队列当中，等待 CPU 轮询进行执行，说白了就是在等待获取执行权；

2.2.3 线程的冻结状态

所谓线程的冻结状态就是，线程被调用了 `sleep` 方法或者调用了 `wait` 方法之后，放弃了 CPU 的执行权，根据上图的箭头可以看到这个时候的线程能够继续回到运行状态，也就是说重新获取了 CPU 的执行权，当然它也可以直接到死亡状态，比如被中断，或者出现异常；

2.2.4 线程的死亡状态

线程在什么情况下能够到死亡状态呢？第一种是出现了致命的异常导致线程被死亡，另外一种是在线程的执行逻辑执行完毕，线程也就正常死亡；死亡后的线程不可能再回到任何一个状态；

2.2.5 深入探讨

🌈 线程被 `start` 了为什么不能严格认为是运行状态呢？

因为 CPU 有一个执行权的问题，也就是说线程被 `start` 之后只具备运行资格，但未必获取到了执行权，因此不能严格认定他为运行状态；

🌈 线程冻结之后为什么还能够回到运行状态呢？

因为线程冻结之后其实他并没有死亡，他只是放弃了运行权，并且他已经没有运行资格了，只有在解冻之后他才有可能获取运行资格，然后获取执行权；

🌈 线程的这几种状态是如何切换的呢？

- 1、初始化状态只能到运行状态；
- 2、运行状态能到冻结状态也能到死亡状态；
- 3、冻结状态能到运行状态也能到死亡状态；
- 4、死亡状态只能接受死亡的事实；

第三节 通过实现 Runnable 接口创建线程

为了能更好的引出为什么要继承 Runnable 接口来创建线程，我们将通过一个实例穿插其中，然后不断的推演，最终将 Runnable 接口引出来，并且还要说明其中的一个设计模式那就是策略模式；

需求描述：

我们假设银行有好几个柜台，每个柜台前面都有一个叫号机，从 0 号一直开始叫号，去过银行或者移动营业厅的朋友们肯定都有这样的经历，那么我们将通过程序来实现一个，然后将我们要说的重点引申出来；

2.3.1 银行排队叫号程序第一版

为了能看清输出的内容，我们假设只能叫号到 50，并且我们只开启 3 个窗口，其余的窗口都暂时休息，看了上面的知识点，您可能会想到我们开辟三个线程同时做这一件事情就可以了，如果能想到这一点，说明上面那么多的文字描述起到了作用，好了，我就揣摩着初学者的心里开始形成第一个版本的售票程序；

```
package com.wenhuisoft.chapter2;

class TicketWindow extends Thread
{

    int max_value = 0; //最大的号码

    @Override
    public void run()
    {
        while(true)
        {
            if(max_value>50)
```

```
        {  
            break;  
        }  
  
        System.out.println(currentThread().getName()+"."+max_value++);  
    }  
}  
}  
  
public class Bank  
{  
    public static void main(String[] args)  
    {  
        TicketWindow t1 = new TicketWindow();  
        TicketWindow t2 = new TicketWindow();  
        TicketWindow t3 = new TicketWindow();  
  
        t1.start();  
        t2.start();  
        t3.start();  
    }  
}
```

好了，程序快速的写完了，但是当我们运行的时候发现，好几个线程叫到的号码是相同的，如果这种事情真的发生在银行营业厅，很容易引起混乱，输出信息如下所示：

```
Thread-0:4  
Thread-2:4  
Thread-0:5  
Thread-2:5
```

其中第一个窗口和第三个窗口都叫了 4 号和 5 号，这样肯定是不行的；虽然我们实现了多线程并发工作，但是这样的程序是相当危险的；

2.3.2 银行排队叫号程序第二版

上面的代码虽然实现了我们所谓的同时叫号程序，但是牵扯出来了一个问题，那就是多个窗口有可能同时叫道一个号码，带着这个问题我们分析一下，然后相处解决方案，因为我们实例化了四个 `TicketWindow` 线程，他的成员变量 `max_value` 都会被分别创建，也就是各自执行各自的逻辑单元，互不相干，那么我们就想到了我们为何不有一个独一份的数据让

他们几个线程去分享操作呢？因此我们想到了静态，于是我们将代码稍作改动

```
Static int max_value = 0;
```

执行后发现的确我们的程序再也没有出现过叫同一个号码的情况，运行后的结果如下所示：

```
Thread-0:45
Thread-1:46
Thread-2:47
Thread-0:48
Thread-1:49
Thread-2:50
```

上述代码虽然实现了我们预期的功能，但是 `static` 不是推荐使用的解决方案，因为他的生命周期实在太长了（伴随着 JVM 的销毁而结束）

好的，既然我们想到了他们需要共享同样一份业务逻辑，那么我们只需要实例化一个线程类然后启动三次不就可以了嘛，我们再次尝试一下！接着修改代码，去掉 `max_value` 之前的 `static` 描述，并且只实例化一个 `TicketWindow`，然后 `start` 三次不就可以了么？但是我们运行之后我们不幸的发现，只有一个线程在运行，并且叫完了所有的号码，并且抛出了一个异常：

```
Exception in thread "main" java.lang.IllegalThreadStateException
    at java.lang.Thread.start(Thread.java:571)
    at com.wenhuisoft.chapter2.Bank.main(Bank.java:32)
```

打开 JDK 源码我们不难找到刚才的那个异常为什么会发生，但是我们要说清楚原因，为什么会出现那样的异常呢？好了，我们一个一个的来分析为什么会出现这样的事情；

🌈 为什么会出现异常呢？这个异常代表什么意思？

线程只能被启动一次，你可以理解为规定，但是我们可以很形象的来描述它，假如你的老师叫你起立（代表启动了你这个线程），这个时候你已经处于运行状态或者说起立状态，然后你的老师又再次对你说起立起立，正常人的思维，肯定是不合理的；

🌈 那么为什么只能有一个线程运行呢？

因为当执行第二次 `start` 的时候主线程已经出现了异常，导致销毁或者死亡，只有正常启动的那个线程保持了正常的状态，因此它要等到将业务逻辑全部执行结束之后才行；

注>通过实验和 JDK 源码的查看，我们总结出来一个线程实例不能被 `start` 两次；

2.3.3 通过 Runnable 接口实现第三版

上面所作一切都是为了引入我们本节的主人公,那就是通过另一种方式来创建线程或者启动线程,其实 Runnable 只是一个任务的接口,他并不是一个线程,他的出现是为了将线程和业务执行逻辑分离,这是我的理解,并且我认为这样的理解是对的,呵呵;

好了,言归正传,我们通过 Runnable 接口的实现完成我们的第三版程序

```
package com.wenhuisoft.chapter2;

class TicketWindow2 implements Runnable
{
    private int max_value = 0;

    public void run()
    {
        while(true)
        {
            if(max_value>50)
                break;

            System.out.println(Thread.currentThread().getName()+":"+max_value++);
        }
    }
}

public class Bank2
{
    public static void main(String[] args)
    {
        TicketWindow2 tw2 = new TicketWindow2();//1
        Thread t1 = new Thread(tw2);//2
        Thread t2 = new Thread(tw2);//3
        Thread t3 = new Thread(tw2);//4

        t1.start();//5
        t2.start();//6
        t3.start();//7
    }
}
```

```
}
```

通过这样的编写，我们在没有用到静态的情况下实现了我们的需求功能！读者可以自行运行；

代码详解，我们通过前面的讨论，如果想要实现不出现重复输出号码的现象，我们必须保证业务逻辑的代码只有一份，因此我们只实例化了一个 `TicketWindow2`（注释 1）

注释 2，3，4 分别实例化了 3 个线程通过构造函数 `Thread(Runnable runnable)`;

注释 5，6，7 分别启动了三个线程；

因此我们的出了一个结论，创建线程的两种方式有如下：

✚ 成为线程的子类，也就是继承 `Thread` 类；

✚ 继承 `Runnable`，成为一个可执行任务；

2.3.4 Runnable 和 Thread 的区别

如果没有真的理解 `Runnable` 那么我觉得将它们两者放在一起讨论是有必要的，如果真的理解了他们压根就没有可比性，我们本小结的内容其实都是废话，请看下面的详细说明；

✚ `Runnable` 就是一个可执行任务的标识而已，仅此而已，而 `Thread` 才是线程所有 API 的体现；

这是因为在Java中存在单继承的问题，但是在接口的实现上却没有这个限制

✚ 继承了 `Thread` 父类就没有办法去继承其他类，而实现了 `Runnable` 接口也可以继承其他类并且实现其他接口，这个区别也是很多书中千篇一律提到的，其实 `Java` 中的对象即使继承了其他类，也可以通过再构造一个父类的方式继承很多个类，或者通过内部类的方式继承很多个类，因此这个区别个人觉得不痛不痒；

✚ 将任务执行单元和线程的执行控制区分开来，这才是引入 `Runnable` 最主要的目的，`Thread` 你就是一个线程的操作者，或者独裁者，你有 `Thread` 的所有方法，而 `Runnable` 只是一个任务的标识，只有实现了它才能称之为一个任务，这也符合面向对象接口的逻辑，接口其实就是行为的规范和标识；

2.3.5 线程中的策略模式

如果认真看了 2.3.4 中第三个区别的描述，其实应该能看出本小结的企图，我们可以姑且认为 `Thread` 是骨架，是提供功能的，而 `Runnable` 只是其中某个业务逻辑的一种实现罢了，

为什么说只是一种实现呢？因为业务逻辑会是很复杂，也会是千变万化的，因此我们需要对它进行高度的抽象，这样才能将具体业务逻辑与抽象分离，程序的可扩展性才能够强，该模式也是本人在编码的时候非常喜欢的一种设计思想；

下面我们就通过实例，来让读者切身体会一下策略模式都有哪些好处！

```
package com.wenhuisoft.chapter2;

/**
 * 策略接口，主要是规范或者让结构程序知道如何进行调用
 */
interface CalcStrategy
{
    int calc(int x,int y);
}

/**
 * 程序的结构，里面约束了整个程序的框架和执行的大概流程，但并未涉及到业务层面的东西
 * 只是将一个数据如何流入如何流出做了规范，只是提供了一个默认的逻辑实现
 * @author Administrator
 */
class Calculator
{
    private int x = 0;

    private int y = 0;

    private CalcStrategy strategy = null;

    public Calculator(int x,int y)
    {
        this.x = x;
        this.y = y;
    }

    public Calculator(int x,int y,CalcStrategy strategy)
    {
        this(x,y);
        this.strategy = strategy;
    }

    public int calc(int x,int y)
    {

```

```
        return x+y;
    }

    /**
     * 只需关注接口，并且将接口用到的入参传递进去即可，并不关心到底具体是要如何进行业务封装
     * @return
     */
    public int result()
    {
        if(null!=strategy)
        {
            return strategy.calc(x, y);
        }

        return calc(x, y);
    }
}

class AddStrategy implements CalcStrategy
{
    public int calc(int x, int y)
    {
        return x+y;
    }
}

class SubStrategy implements CalcStrategy
{
    public int calc(int x, int y)
    {
        return x-y;
    }
}

public class StrategyTest
{
    public static void main(String[] args)
    {
        //没有任何策略时的结果
        Calculator c = new Calculator(30, 24);
```

```
System.out.println(c.result());

//传入减法策略的结果
Calculator c1 = new Calculator(10, 30, new SubStrategy());
System.out.println(c1.result());

//看到这里就可以看到策略模式强大了，算法可以随意设置，系统的结构并不会发生任何变化
Calculator c2 = new Calculator(30, 40, new CalcStrategy())
{
    public int calc(int x, int y)
    {
        return ((x+10) - (y*3)) / 2;
    }
});
System.out.println(c2.result());
}
```

上述代码注释的部分已经很翔实了，我就不用再多说些什么了，通过上面代码的演示，相信大家就会明白为什么需要有 `Runnable` 的出现，也能体会到我为什么说将 `Thread` 和 `Runnable` 来进行比较本身就是一个不合适的提议，因为他们关注的东西就不是一个事情，一个负责线程本身的功能，另外一个则专注于业务逻辑的实现，说白了 `Runnable` 中的 `run` 方法，即使不再 `Thread` 中使用，他在其他地方照样也能使用，并不能说他就是一个线程；

2.3.6 Runnable 接口的用法总结

通过 `Runnable` 接口构造线程，其实翻阅 `JDK` 文档就非常清楚了，这里我只简单的说一下即可

`Thread(Runnable runnable)` 当然还有其他的重载构造函数，其实目的都是一样的，需要将 `Runnable` 传递给 `Thread` 才能被执行到逻辑运算单元；

2.3.7 查缺补漏

🌈 线程名字的默认编号

线程的名字默认是这样命名的 `thread-n`(其中 `n` 是从 0 开始的数字)当然你也可以通过显式的方式进行设定，比如他有 `setName` 方法，并且有 `Thread(String name)` 这样的构造函数

数传递名字，并且有 `getName()` 方法获取名字等

线程名字的获取方式

如何获取当前运行的线程名字呢？我们知道 `main` 函数并没有继承 `Thread` 也就是说我们不能通过 `getName` 这样的 API 获取名字，那么我们应该如何获取呢，其实 `Thread` 类提供了一个静态方法 `Thread.currentThread()` 就可以获取当前运行的线程，如果获取了线程那么获取他的名字应该是易如反掌的事情了；

第三章 线程的同步

我们在第二章中的那个叫号小程序，如果您多运行几次，或者将程序中的 `max_value` 修改大之后，您可能会发现有这样的问题，为什么有些号码没有显示出来，相反有些号码则被显示了几次，这到底是真么回事呢？

为了明显期间，我们让线程稍作休眠状态，这个时候我们再来看看到底发生了怎样的事情，然后慢慢来分析

修改后的代码

```
package com.wenhuisoft.chapter2;

class TicketWindow2 implements Runnable
{

    private int max_value = 0;

    public void run()
    {
        while(true)
        {
            if(max_value>500)
                break;
            try {
                Thread.sleep(5);
            } catch (InterruptedException e) {
            }

            System.out.println(Thread.currentThread().getName()+":"+max_value++);
        }
    }
}
```

```

    }
}

public class Bank2
{
    public static void main(String[] args)
    {
        TicketWindow2 tw2 = new TicketWindow2();//1
        Thread t1 = new Thread(tw2);//2
        Thread t2 = new Thread(tw2);//3
        Thread t3 = new Thread(tw2);//4

        t1.start();//5
        t2.start();//6
        t3.start();//7
    }
}

```

执行后的结果抓取我们发现有如下的现象

```

Thread-1:33
Thread-0:33
Thread-2:501
Thread-0:502

```

可以看到不仅有重复的号码出现，并且有超过 500 的情况出现，这到底是由于什么原因引起的呢？接下来我们将满满的来分析

```

private int max_value = 0;

public void run()
{
    while(true)
    {
        // (1)
        if(max_value > 500)
            break;
        try {
            Thread.sleep(5);
        } catch (InterruptedException e) {
        }
        // (2)
        System.out.println(Thread.currentThread().getName()+" "+max_value++);
    }
}

```

假设三个线程现在同时执行到了 (1) 这个位置，判断条件都满足也就是说都小于 500，好的我们假定一个数字此时刚好是 499，接着往下分析，假设此时 1 号线程执行到了 (2)

这个位置，此时 CPU 恰恰将它的执行权切换到了 2 号线程，那么由于 2 号线程在 1 这个位置上进行过判断条件满足，因此它直接输出 500，这个应该不难理解，2 号线程执行完毕之后继续回到了（1）的位置，但是条件不成立，它自己退出了，因此 CPU 又将执行权转到了 1 号线程，一号线程起来之后就执行输出语句，因此变成了 501，回去（1）位置之后判断发现条件不符合退出，现在只剩下 3 号线程，3 号线程也不用再进行判断了，直接到（2）号位置执行输出语句，因此输出了 502，至于重复输出的根据这样的逻辑也是能够解释通过的，读者可以自己进行解释；

为什么会出现这样的问题呢？因为我们需要访问的数据没有被保护，这就是多线程最最令人头疼的地方，线程安全，多线程之间的数据共享引发的安全问题，因此本章将针对这个问题进行全面的解释，读者阅读完本章之后将会了解到锁的概念和线程安全的概念，以及如何解决线程的安全问题；并且我们会模拟一个死锁的现象，让大家了解如何在以后的工作开发中规避死锁的出现；

第一节 同步代码块

首先我们来看一下如何让之前的程序不再出现错号的情况，我们需要对共享的数据进行保护，然后我们来讲如何进行保护

```
package com.wenhuisoft.chapter3;

class TicketWindow2 implements Runnable
{

    private int max_value = 0;

    private Object lock = new Object();

    public void run()
    {
        while(true)
        {
            synchronized(lock)
            {
                if(max_value>50)
                    break;

                try
```

```
{  
    Thread.sleep(10);  
} catch (InterruptedException e){}  
  
System.out.println(Thread.currentThread().getName()+":"+max_value++);  
}  
}  
}  
  
}  
  
public class Bank2  
{  
    public static void main(String[] args)  
    {  
        TicketWindow2 tw2 = new TicketWindow2();  
        Thread t1 = new Thread(tw2);  
        Thread t2 = new Thread(tw2);  
        Thread t3 = new Thread(tw2);  
  
        t1.start();  
        t2.start();  
        t3.start();  
    }  
}
```

代码经过这样的修改之后，再次执行之前的那种现象基本上不会再出现，并且不管你启动多少个线程，错号的问题都会得到彻底的解决，那么好让我们看看到底刚才的代码发生哪些事情？

3.1.1 给共享数据加锁

可以看到我们的代码中增加了一个 **synchronized** 这样的关键字，他的意思就是线程的同步，立即的通俗来讲就是给代码或者业务逻辑加锁，何为加锁呢？就是我们将部分数据保护起来，每次只能有一个线程进行访问，举个最简单的例子，假设有一个单行道，不管后面的人是老老实实排队的，还是从中间翻越过来的，还吃打架打赢挤过来的，但是单行道的出口总是只能有一个人才能通过！通过单行道的那个门口就是我们所说的锁，那程序加了锁之后是如何运行的呢？我们在 **3.1.3** 详解中会有深入的探讨：

由于个人能力有限，书中难免会有偏颇之处，希望读到这本书的朋友能够给予我指导和批评，欢迎你们的指正

Java 中同步代码块的语法如下所示，其中 `obj` 是任何一个对象；

```
.....  
synchronized(obj)  
{  
    .....//your business code  
}  
.....
```

3.1.2 同步代码块的有效范围

看到这里有些人就要问了，既然加了锁，我们的代码就只能被一个线程调用，这样岂不是降低了效率，在同步代码的部分并没有多线程并发的情况出现呀？如果你能想到这一点，就说明你对锁的机制了解的差不多了，的确，情况的确如此，因为我们要尽量的缩小同步锁的范围，有什么原则么？其实如果程序写多了，就会想到我们同步代码块最小的粒度应该放在共享数据的上下文，或者说共享数据被操作的上下文中，通过上面的代码示例，不难发现，我们的同步代码块的确做到了最小化的同步了共享数据 `max_value`

3.1.3 同步代码块的详解

```
private Object lock = new Object();  
  
public void run()  
{  
    while(true)  
    {  
        synchronized(lock)  
        {  
            if(max_value>50)  
                break;  
            try  
            {  
                Thread.sleep(10);  
            } catch (InterruptedException e){}  
            System.out.println(Thread.currentThread().getName()+"  
        }  
    }  
}
```

通过上图对代码的图示，我们可以看出，假设此时三个线程同时到达了（1）这个位置，由于锁的缘故，他们三个只能有一个获取到该锁，在（1）和（2）这个位置中间其实代码是单线程的，也就是说只有一个线程在运行，其他线程都在等待着获取锁的线程 X 将锁释放掉，

因此 `max_value` 只能被一个线程所操作，因此他就是安全的，直到 `x` 线程运行到（2）这个位置，其他线程才有可能获取到 `x` 所释放掉的锁，重新执行代码逻辑；

3.1.4 该如何定义一个锁

笔者曾经面试过一个有工作经验的员工，他所设计的锁是放在 `run` 方法里面，虽然定义了一个锁，但是这个锁并没有对其他线程起到作用，只会加重 `run` 方法效率的低下，这是为什么呢？其实锁的定义或者叫声明还是有一定规则的，在这里我将我理解的几个规则写出来，希望能与大家共勉

🚦 所谓加锁，就是为了防止多个线程同时操作一份数据，如果多个线程操作的数据都是各自的，那么就没有加锁的必要

🚦 共享数据的锁对于访问他们的线程来说必须是同一份，否则锁只能私有的锁，各锁个的，起不到保护共享数据的目的，试想一下将 `Object lock` 的定义放到 `run` 方法里面，每次都会实例化一个 `lock`，每个线程获取的锁都是不一样的，也就没有争抢可言，说的在通俗一点甲楼有一个门上了锁，`A` 要进门，乙楼有一个门上了锁 `B` 要进门，`A` 和 `B` 抢的不是一个门，因此不存在数据保护或者共享；

🚦 锁的定义可以是任意的一个对象，该对象可以不参与任何运算，只要保证在访问的多个线程看来他是唯一的即可；

在本小节中，我们通过同步代码块的方式解决了错票的问题，并且讲解了同步代码块是如何做到这样的事情的，并且向读者介绍了同步代码块的语法格式，还有需要注意的问题；

第二节 同步方法

其实方法的同步和代码块的公布大相径庭就是在方法名前面加上 `synchronized` 关键字，具体的格式如何呢？我这里简单的写一下

```
Private|default|protected|public    [static]    synchronized    void|return    type  
methodName(Parameters)
```

3.2.1 同步重构方法

好了，为了能更加立体的体现出来同步方法的使用方法，我们来进行一下演示

```
package com.wenhuisoft.chapter3;

class TicketWindow2 implements Runnable
{

    private int max_value = 0;

    public void run()
    {
        while (true)
        {
            if(ticket())
                break;
        }
    }

    /**
     *
     */
    private synchronized boolean ticket()
    {
        if (max_value > 500)
        {
            return true;
        }
        try
        {
            Thread.sleep(10);
        } catch (InterruptedException e)
        {
        }
        System.out.println(Thread.currentThread().getName() + ":" +
max_value++);
        return false;
    }

}
```

```
public class Bank2
{
    public static void main(String[] args)
    {
        TicketWindow2 tw2 = new TicketWindow2();
        Thread t1 = new Thread(tw2);
        Thread t2 = new Thread(tw2);
        Thread t3 = new Thread(tw2);

        t1.start();
        t2.start();
        t3.start();
    }
}
```

通过运行该代码，我们发现，我们的程序也可以像第一节中那样完成相应的功能，但是这个时候有人又要问了，为什么不能将 `synchronized` 关键字放在 `run` 方法上，好了请看下一个小节

3.2.2 同步 `run` 方法

`Run` 方法是否可以加 `synchronized` 关键字，当然是可以的，这个在任何时候都符合语法规则，但是为什么不能将 `run` 方法同步？如果您仔细阅读了我们对 `CPU` 执行权那部分的分析之后，这个问题也许您自己都已经找到答案了，当第一个线程获取到了 `CPU` 的执行权之后，进入 `run` 方法，一定是执行完毕所有的逻辑才会退出，因为 `run` 方法加了锁，其他线程只有等待的份，地一个线程执行完毕退出，其他线程获取到了锁，想要执行，一看判断已经不符合则自动退出；因此 `run` 方法加锁，真实情况是会有多个线程运行，但是只有一个线程执行业务逻辑，其他线程都等于阻塞状态，如果不相信，可以尝试一下，打印出来的信息一定只是一个线程相关的；我在这里就不做演示了。

3.2.3 同步总结

不管是同步代码块或者同步方法，我们需要事先确定的是：“当同一份的数据被多个线程操作的时候才考虑同步”，否则将会产生效率的问题，针对我用双引号引起来的文字重点说明一下：

🚦 同一份数据

如果不同的线程访问的不是同一份数据，就没有必要加锁保持同步，就像我之前举例 A，B 两个人进门的一样

🚦 多个线程访问

多个线程访问的时候采取考虑同步，如果一份数据只是被一个线程访问，就没有必要进行同步；

🚦 多个线程同步的代码块必须是同一个锁

这块应该不难理解了，我也就不再赘述了；

第三节 this 锁与 static 锁

我们看到之前的代码中，我们同步代码块，是定义一个对象，让他作为一个锁，但是同步函数，我们的锁是什么东西呢？我们在本节中将讨论两个最主要的锁；

3.3.1 this 锁

同步函数其实用到的锁就是 **this** 锁，为什么他用到的是 **this** 锁呢？为了证实这个结论我们本节将会有两个实验性的程序来作为支撑，说服自己和读者证明同步函数用到的就是 **this** 锁好了，请看下第一个程序

🚦 需求：

我们定义一个类，其中有两个方法，均加了同步锁，假设函数的同步不是 **this** 锁，我们如果启动一个线程调用方法 A，另外一个线程调用方法 B，A 方法和 B 方法里均是死循环，按照同步函数不是 **this** 锁的逻辑，两个函数中的逻辑将会被同时执行，但是情况是否是这样，我们需要通过代码进行实验

🚦 程序代码：

```
package com.wenhuisoft.chapter3;

class ClassA
{

    public synchronized void A()
    {
```

```
System.out.println("AAAAAAAAAAAAAAAA");
while (true)
{
    // do nothing
}

public synchronized void B()
{
    System.out.println("BBBBBBBBBBBBBBBBBB");
    while (true)
    {
        // do nothing
    }
}

public class MethodSynchronizedTest
{
    public static void main(String[] args)
    {
        final ClassA clazz = new ClassA();

        //启动一个线程
        new Thread(new Runnable()
        {
            public void run()
            {
                clazz.A();//调用A方法
            }
        }).start();

        //启动另一个线程
        new Thread(new Runnable()
        {
            public void run()
            {
                clazz.B();//调用B方法
            }
        }).start();
    }
}
```

✚ 程序输出:

```
AAAAAAAAAAAAAAAAAAAA
```

✚ 文字说明:

分别启动了两个线程，分别用来执行 `ClassA` 中的两个方法 `A` 和 `B`，两个方法都是加了锁的，也就是说某个线程进到方法 `A` 中其他线程就不能进入 `A`，但是另一个线程应该能进入 `B`，但是我们等了半天方法 `B` 仍然没有输出，因此我们得出一个结论，他们的锁是同一个，至于是哪一個鎖呢？答案就是 `this` 锁；但是这个例子似乎还不过瘾，那么我们继续修正叫号的程序，让读者有一个更加直观的理解；

✚ 需求:

我们增加一个方法，也用来进行叫号，在 `run` 方法中也进行叫号，第一个线程调用 `run` 方法中的逻辑，第二个方法调用函数中的逻辑；

✚ 程序:

```
package com.wenhuisoft.chapter3;

class TicketWindow3 implements Runnable
{
    private int max_value = 0;

    private Object lock = new Object();

    private boolean flag = true;

    public void run()
    {
        if (flag)
        {
            while (true)
            {
                synchronized (lock)
                {
                    if (max_value > 500)
                    {
                        break;
                    }

                    try
```

```
        {
            Thread.sleep(10);
        } catch (InterruptedException e)
        {
        }
        System.out.println(Thread.currentThread().getName()
            + ":lock..." + max_value++);
    }
}
} else
{
    while (true)
        if (ticket())
            break;
}
}

private synchronized boolean ticket()
{
    if (max_value > 500)
    {
        return true;
    }
    try
    {
        Thread.sleep(10);
    } catch (InterruptedException e)
    {
    }
    System.out.println(Thread.currentThread().getName() + ": method.."
        + max_value++);
    return false;
}

public void change() throws InterruptedException
{
    Thread.sleep(30); //读者可以自行思考为什么要sleep
    this.flag = false;
}
}

public class Bank3
```

```
{  
  
    public static void main(String[] args) throws InterruptedException  
    {  
  
        TicketWindow3 tw3 = new TicketWindow3();  
        Thread t1 = new Thread(tw3);  
        Thread t2 = new Thread(tw3);  
        t1.start();  
        tw3.change();  
        t2.start();  
  
    }  
  
}
```

结果输出：

```
Thread-0:lock...500  
Thread-1: method...501
```

文字说明：

可能到输出性信息，其中会有 501 这样的信息输出，为什么会这样呢？因为上述的代码两处业务逻辑同步锁是两把锁，如果您将 lock 换成 this，这个现象就不会出现，读者可以自己进行测试；

3.3.2 static 锁

如果我们的方法 ticket 是一个静态方法，再次测试一下您会发现，还是会出现 501 这样的输出信息，根据之前的描述读者可能会第一时间想到他们两个用到的锁不是同一把锁，因此我们将代码在次做了修改

```
package com.wenhuisoft.chapter3;  
  
class TicketWindow3 implements Runnable  
{  
  
    private static int max_value = 0;  
  
    private boolean flag = true;  
  
    public void run()  
    {  
        if (flag)  
        {
```



```
        while (true)
        {
            synchronized (TicketWindow3.class)
            {
                if (max_value > 500)
                {
                    break;
                }

                try
                {
                    Thread.sleep(10);
                } catch (InterruptedException e)
                {
                }

                System.out.println(Thread.currentThread().getName()
                    + ":lock..." + max_value++);
            }
        }
    } else
    {
        while (true)
        {
            if (ticket())
            {
                break;
            }
        }
    }

    private synchronized static boolean ticket()
    {
        if (max_value > 500)
        {
            return true;
        }

        try
        {
            Thread.sleep(10);
        } catch (InterruptedException e)
        {
        }

        System.out.println(Thread.currentThread().getName() + ": method.."
            + max_value++);

        return false;
    }
}
```

```
public void change() throws InterruptedException
{
    Thread.sleep(30); //读者可以自行思考为什么要sleep
    this.flag = false;
}

}

public class Bank3
{
    public static void main(String[] args) throws InterruptedException
    {
        TicketWindow3 tw3 = new TicketWindow3();
        Thread t1 = new Thread(tw3);
        Thread t2 = new Thread(tw3);
        t1.start();
        tw3.change();
        t2.start();
    }
}
```

静态锁，锁是类的字节码信息，因此如果一个类的函数为静态方法，那么我们需要通过该类的 class 信息进行加锁；

第四节 线程的休眠

看了这么多的代码，读者会发现进场会有一个 sleep 函数出现，那么 sleep 到底是干什么的呢？它是让当前的运行线程进入休眠状态，也就是主动放弃 CPU 执行权，其实 JDK 提供了很多个 sleep 的重构函数，请看下面的 JDK 文档

static void	sleep (long millis) 在指定的毫秒数内让当前正在执行的线程休眠（暂停执行）。
static void	sleep (long millis, int nanos) 在指定的毫秒数加指定的纳秒数内让当前正在执行的线程休眠（暂停执行）。

第五节 单例模式的详解

了解单例设计模式的人都知道，单例中涉及的类他在内存之中始终是独一份存在的，如

由于个人能力有限，书中难免会有偏颇之处，希望读到这本书的朋友能够给予我指导和批评，欢迎你们的指正

果存在两份则将出现问题，并且单例模式有两种相对比较有特点的形式，那就是饿汉式与懒汉式单例模式，在本节中我们将会详细讲解单例设计模式的两种形式，并且我们将会讲解如何在多线程的情况下使用单例设计模式；

3.5.1 饿汉式单例模式

所谓饿汉式单例设计模式，就是将类的静态实例作为该类的一个成员变量，也就是说在JVM加载它的时候就已经创建了该类的实例，因此它不会存在多线程的安全问题，详细代码请看如下：

```
package com.wenhuisoft.chapter3;

public class SingleTest
{
    private final static SingleTest instance = new SingleTest();

    private SingleTest()
    {
    }

    public static SingleTest newInstance()
    {
        return instance;
    }
}
```

可以看到上述代码中的单例不存在线程安全的问题，但是他有一个性能上面的问题，那就是提前对实例进行了初始化或者说构造，假设构造该类需要很多的性能消耗，如果代码写成这个样子将会提前完成构造，又假设我们在系统运行过程中压根就没有对该实例进行使用，那岂不是浪费系统的资源呢？因此单例设计模式其实还有下一个小节的版本；

3.5.2 懒汉式单例模式

所谓懒汉式单例模式的意思就是，实例虽然作为该类的一个实例变量，但是他不主动进行创建，如果你不使用它那么他将会永远不被创建，只有你在第一次使用它的时候才会被创建，并且得到保持；请看下一段代码

```
/**
 *
 */
package com.wenhuisoft.chapter3;

public class SingleTest
{
    private static SingleTest instance =null;

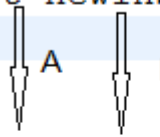
    private SingleTest()
    {
    }

    public static SingleTest newInstance()
    {
        if(null==instance)
        {
            instance = new SingleTest();
        }
        return instance;
    }
}
```

上述的代码就是我们所说的懒汉式单例模式，但是根据上文中的关于线程安全问题的分析我们不难发出现，`instance` 有可能会被创建两次，至于原因为什么呢？让我们根据之前的

知识点来慢慢分析，然后总结出来一个最优的懒汉式单例模式

```
public static SingleTest newInstance()
{
    if (null == instance)
    {
        instance = new SingleTest();
    }
    return instance;
}
```



根据我们之前的知识点，运行中的程序很有可能会出现上图所出现的情况。也就是 A 线程和 B 线程有可能同时执行到了 `null==instance` 的部分他们判断到了 `instance` 没有被创建，因此分别实例化了一个以上的 `instance`，这样的单例类将是非常危险的，那么我们应该如何避免多线程引起的问题呢，看到这里您可能想到了用 `synchronized` 这个关键字来解决问题，于是有了如下的代码

```
public synchronized static SingleTest newInstance()
{
    if (null == instance)
    {
        instance = new SingleTest();
    }
    return instance;
}
```

但是该方法的效率将是相当低下的，因为每一次调用都要获取锁，判断锁的状态，因此就会出现解决了安全问题，带来了效率问题，当然安全性和效率问题本来就是两个很不可调和的矛盾，但是我们也不应该就此妥协，需要尽我们的智慧既解决了安全问题又带来了最小的效率影响；我们将程序写成如下的样子

```
public static SingleTest newInstance()
{
    if (null == instance)
    {
        synchronized (SingleTest.class)
        {
            if (null == instance)
            {
                instance = new SingleTest();
            }
        }
    }
}
```

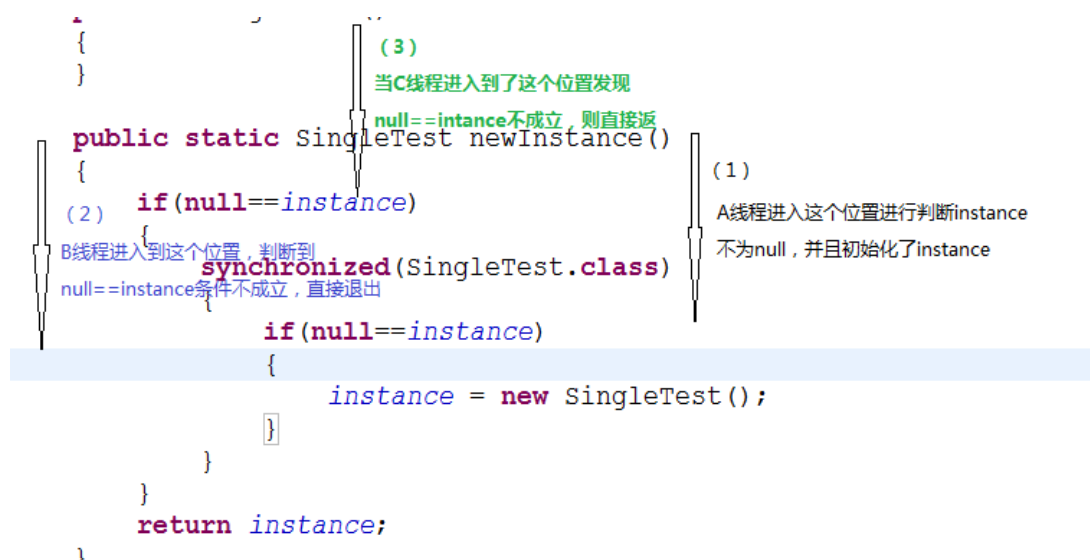
```

    }
}

return instance;
}

```

好的，让我们继续以图示的方式来原因一下，上述代码带来了哪些改变又如何将效率的损耗降到了最低



通过上述代码的分析，我们不难发现，锁的等待或者争抢最多发生两次，也就是同步代码块中的代码最多被执行两次，如此一来，安全问题解决了，效率问题也被解决掉了。

第六节 死锁

多线程同步锁总是以牺牲系统性能为代价的，但是比牺牲性能代价更加严重的将是死锁，程序一旦出现死锁的状况，将会挂死而并不是退出，有时候死锁的问题是很难排查的，尤其是在较大的项目中多人协作的项目中，死锁是一个很头疼的问题，所以我们应该在编写程序的时候规避掉死锁，为了规避死锁，我们首先需要写出一个死锁程序，这样会很清楚什么是死锁，然后又如何避免死锁；

学过操作系统一本书的人应该知道一个很经典的例子《科学家吃面》，读者可以找到操作系统那本书或者上网查找一下

3.6.1 什么是死锁

假设有两个线程 A 和 B，其中 A 持有 B 想要的锁，而 B 持有 A 想要的锁，两个都在等待各自释放所需要的锁，这样的情况很容易引起死锁现象的发生，我们将在下一小节中用程序来实现一个死锁的案例；

3.6.2 死锁程序的模拟

```
package com.wenhuisoft.chapter3;

class Dead
{
    private Object lock = new Object(); // 自定义的一个锁

    private int x = 0;

    public void methodA()
    {
        synchronized (lock) // 先用lock锁住程序
        {
            synchronized (this)
            {
                System.out.println("method a .." + (x++));
            }
        }
    }

    public void methodB()
    {
        synchronized (this)
        {
            synchronized (lock)
            {
                System.out.println("method b .." + (x++));
            }
        }
    }
}

public class DeadLock
```

```
{  
  
    public static void main(String[] args)  
    {  
  
        final Dead dead = new Dead();  
        new Thread(new Runnable()  
        {  
            public void run()  
            {  
                while (true)  
                {  
                    dead.methodA();  
                    try  
                    {  
                        Thread.sleep(10);  
                    } catch (InterruptedException e)  
                    {  
                        e.printStackTrace();  
                    }  
                }  
            }  
        }).start();  
  
        new Thread(new Runnable()  
        {  
            public void run()  
            {  
                while (true)  
                {  
                    dead.methodB();  
                    try  
                    {  
                        Thread.sleep(10);  
                    } catch (InterruptedException e)  
                    {  
                        e.printStackTrace();  
                    }  
                }  
            }  
        }).start();  
    }  
}
```

运行出现了如下的情况


```
method a ..0
method b ..1
method a ..2
method b ..3
method a ..4
method b ..5
method b ..6
method a ..7
method b ..8
method a ..9
method b ..10
method a ..11
method b ..12
method a ..13
```

可以看到程序运行到一定地步就阻塞住了，然后没有任何输出，但是程序并没有停止，而是出现了挂死；

3.6.3 如何避免死锁

从刚才的代码中可以看出我们实现死锁的方式是同步代码块中的同步，因此在日常的开发过程中应该避免使用这样的情况，如果有这样的情况出现也要认真的推演，反复地琢磨，万不得已的请看下才考虑同步代码块中有同步代码块；

第四章 线程间的通讯

通过前三章的学习，我们已经掌握了线程的基础知识，以及如何创建线程，还有多线程操作一份数据时可能引发的安全问题，在本章中我们将会着重阐述多个线程如何协同工作，这也就是多线程之间的通讯；

第一节 生产者消费者

4.1.1 简易版生产者消费者

```
package com.wenhuisoft.chapter4;
```

```
class NumberFactory
{
    private int i = 0;

    private Object lock = new Object();

    public void create()
    {
        synchronized (lock)
        {
            System.out.println("create...i-" + i++);
        }
    }

    public void consume()
    {
        synchronized (lock)
        {
            System.out.println("consume...i-" + i);
        }
    }
}

public class ProducerAndCustomer
{
    final static NumberFactory numberFactory = new NumberFactory();

    public static void main(String[] args)
    {
        new Thread(new Runnable()
        {
            public void run()
            {
                while(true)
                {
                    numberFactory.create();
                }
            }
        }).start();

        new Thread(new Runnable()
        {
            public void run()
            {
                while(true)
                {
                    numberFactory.consume();
                }
            }
        }).start();
    }
}
```

```
{  
    while(true)  
    {  
        numberFactory.consume();  
    }  
}  
}).start();  
}
```

上面的代码意图很简单，一个线程实现让 $x++$ （模拟我们在创建 x 值）而另外一个线程则是不断的消费 x 值（也就是上述代码中打印而已），请看看程序的输出片段

```
consume...i-33113  
create...i-33113  
consume...i-33114  
create...i-33114
```

可以看到生产一个消费一个，但是为什么消费在生产前面呢？这是因为我的笔记本是双核的缘故，一个 CPU 已经生产成功，但是还没有来得及进行输出被另外一个 CPU 最先执行了，这个问题不大，但是下面的输出片段将显得很诡异

```
create...i-35607  
create...i-35608  
create...i-35609  
create...i-35610  
create...i-35611  
create...i-35612  
create...i-35613  
create...i-35614  
consume...i-35615  
create...i-35615
```

看到成产了好多次，但是只被消费了一次，这是怎么回事呢？下节我们将会有一个改进版的让读者看看真正意义上的 Producer-Customer

4.1.2 改进版生产者消费者

真正意义上的生产者消费者应该是这样的，生产一个消费一个，如果没有生产那就没有消费，没有被消费完毕就不应该进行生产，因此我们将推出两个比较重要的方法，那就是 `wait` 和 `notify`，但是我们的目的不是讲解这两个方法，在后文中会有比较详细的讲解；

由于个人能力有限，书中难免会有偏颇之处，希望读到这本书的朋友能够给予我指导和批评，欢迎你们的指正

```
package com.wenhuisoft.chapter4;

class NumberFactory
{
    private int i = 0;

    private Object lock = new Object();

    private boolean created = false;

    public void create()
    {
        synchronized (lock)
        {
            if(!created)
            {
                i++;
                System.out.println("create:"+i);
                lock.notify();
                created = true;
            }else
            {
                try
                {
                    lock.wait();
                } catch (InterruptedException e)
                {
                    e.printStackTrace();
                }
            }
        }
    }

    public void consume()
    {
        synchronized (lock)
        {
            if(created)
            {
                System.out.println("consume:"+i);
                created = false;
                lock.notify();
            }
        }
    }
}
```

```
        else
        {
            try
            {
                lock.wait();
            } catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}

public class ProducerAndCustomer
{
    final static NumberFactory numberFactory = new NumberFactory();

    public static void main(String[] args)
    {
        new Thread(new Runnable()
        {
            public void run()
            {
                while(true)
                {
                    numberFactory.create();
                }
            }
        }).start();

        new Thread(new Runnable()
        {
            public void run()
            {
                while(true)
                {
                    numberFactory.consume();
                }
            }
        }).start();
    }
}
```

```
}
```

程序输出如下所示

```
create:59109
consume:59109
create:59110
consume:59110
create:59111
consume:59111
create:59112
consume:59112
create:59113
consume:59113
create:59114
consume:59114
```

可以看到非常规律的进行了生产和消费的切换,接下来我们针对上述的代码进行一下详细的讲解,讲解其中的每一个细节;

4.1.3 细说生产者消费者

细说生产

```
public void create()
{
    synchronized (lock)
    {
        if(!created)
        {
            i++;
            System.out.println("create:"+i);
            lock.notify();//(1)
            created = true;//(2)
        }else
        {
            try
            {
                lock.wait();//(3)
            } catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

上面的代码我故意标注了(1)(2)(3)然后将着重讲解这几部分的代码,首先我们定

义了一个 `created` 代表数据创建的状态，`created=false` 的时候表示需要进行创建数据，其中

(1) 通知消费者消费生产的数据，(2) 代表数据创建完毕标识，(3) 代表如果已经创建了，等待着消费再次创建数据；

细说消费

```
public void consume()
{
    synchronized (lock)
    {
        if(created)
        {
            System.out.println("consume:"+i);
            created = false;//(1)
            lock.notify();//(2)
        }
        else
        {
            try
            {
                lock.wait();//(3)
            } catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

同样我也对 (1) (2) (3) 做了标注，其中 (1) 表示已经消费完毕，(2) 表示通知生产者进行生产 (3) 代表当前正在生产暂且等待；

第二节 多线程下的生产者消费者

4.2.1 多线程下的生产者消费者遇到的问题

在第一节中我们实现了一个“准生产者消费者模式”为什么要说他是“准”呢？是因为现实的日常开发中不会是那样子的情况，也就是绝对不会是只有一个线程进行成产，只有一个线程进行消费，因此我们将该程序做成多线程生产，为了能够说明问题，我们将生产者消费者重新实现一个版本，代码如下：

```
package com.wenhuisoft.chapter4;

/**
 * 封装了数据生产工厂，该工厂中提供了生产和消费方法
```

```
*
* @author <a href='wangwenjun62@gmail.com'>wangwenjun</a>
*/
class Factory
{
    private int i = 0;

    private boolean created = false;

    public void create()
    {
        synchronized (this)
        {
            if (created)
            {
                try
                {
                    wait();//(1)
                } catch (InterruptedException e)
                {
                }
            } else
            {
                i = i + 1;
                System.out.println(Thread.currentThread().getName()
                    + "-create-" + i);
                notify();//(2)
                this.created = true;
            }
        }
    }

    public void consume()
    {
        synchronized (this)
        {
            if (created)//(3)
            {
                System.out.println(Thread.currentThread().getName()
                    + "-consume-" + i);
                notify();
                this.created = false;
            } else
```



```
        {
            try
            {
                wait();//(4)
            } catch (InterruptedException e)
            {
            }
        }
    }
}

/**
 * 生产者与消费者的基类
 *
 * @author <a href='wangwenjun62@gmail.com'>wangwenjun</a>
 */
abstract class AbsFactory implements Runnable
{
    protected Factory factory = null;

    public AbsFactory(Factory factory)
    {
        this.factory = factory;
    }

    abstract protected void execute();

    public void run()
    {
        while (true)
        {
            execute();
            try
            {
                Thread.sleep(10);
            } catch (InterruptedException e)
            {
            }
        }
    }
}
```

```
class Producer extends AbsFactory
{

    public Producer(Factory factory)
    {
        super(factory);
    }

    @Override
    protected void execute()
    {
        factory.create();
    }
}

class Consumer extends AbsFactory
{

    public Consumer(Factory factory)
    {
        super(factory);
    }

    @Override
    protected void execute()
    {
        factory.consume();
    }
}

public class ProducerCustomer
{
    public static void main(String[] args)
    {
        if (args.length == 0)
        {
            System.out.println("Usage:java
com.wenhuisoft.chapter4.ProducerCustomer number");
            System.out.println("Please restart again...");
            System.exit(0);
        }

        int count = 0;
```

```
try
{
    count = Integer.parseInt(args[0]);
} catch (Throwable t)
{
    System.out.println("Please enter a integer type number...");
    System.exit(0);
}

final Factory factory = new Factory();
for (int i = 0; i < count; i++)
{
    new Thread(new Producer(factory)).start();
    new Thread(new Consumer(factory)).start();
}
}
```

当我们执行 `java com.wenhuisoft.chapter4.ProducerConsumer 1` 也就是只启动了一个生产线程和一个消费线程；系统运行并没有什么异样，但是当我们执行

`java com.wenhuisoft.chapter4.ProducerConsumer 2` 则会出现死锁的问题或者生产多次消费一次，或者生产一次消费多次，为什么会出现上述现象呢？我们在本节中将会进行详细的分析（以死锁的情况为例进行分析），分析的过程依赖于上述代码

- ✚ 假设创建线程 1 进入到（1）号位置：因为 `created` 为 `false` 所以他完成了创建，再次循环到条件判断之后发现 `created` 为 `true` 则放弃了 CPU 执行权因为执行了 `wait` 方法；
- ✚ 假设创建线程 2 进入到（1）号位置，判断到 `created` 为 `true` 则也放弃了 CPU 执行权；
- ✚ 此时消费线程 3 进入到了（3）号位置：因为 `created` 为 `true`，所以他直接消费掉了并且唤醒了此时，再次循环到判断语句发现 `created` 为 `false`，他也进入到了阻塞状态；
- ✚ 消费线程 4 进入到（3）号位置，判断到 `created` 为 `false`，直接进入了阻塞；
- ✚ 其中 2 号创建线程被 3 号消费线程唤醒，因此它可以进行创建，当他创建完毕之后将 `created` 设置为 `true`，然后唤醒了“某个线程”并且自己进入了阻塞状态；

好了，分析到了这里我们现在可以肯定的是 2 号线程是阻塞的，1，3，4 有可能被 2 号线程唤醒，假设 2 号线程唤醒的是 1 号线程，1 号线程运行起来之后发现 `created` 为 `true` 则有进入了阻塞状态，此时 1，2，3，4 全部都被阻塞，没有一个处于运行状态的线程，因此系统将会不再有任何输出，至于会有一次生产多次消费或者多次生产一次消费的情况请读者

自行进行分析

4.2.2 多线程下的生产者消费者改进版

出现死锁的问题我们已经进行了分析，在本节中我们将对其进行完善，形成最终一个比较可靠的版本，先来看代码，然后看输出，最后看分析

程序代码输出

```
package com.wenhuisoft.chapter4;

/**
 * 封装了数据生产工厂，该工厂中提供了生产和消费方法
 *
 * @author <a href='wangwenjun62@gmail.com'>wangwenjun</a>
 */
class Factory
{
    private int i = 0;

    private boolean created = false;

    public synchronized void create()
    {
        while (created)
        {
            try
            {
                wait();
            } catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
        i = i + 1;
        System.out.println(Thread.currentThread().getName() + "-create-" + i);
        this.created = true;
        notifyAll();
    }

    public synchronized void consume()
    {

```

由于个人能力有限，书中难免会有偏颇之处，希望读到这本书的朋友能够给予我指导和批评，欢迎你们的指正

```
        while (created)
        {
            System.out.println(Thread.currentThread().getName() + "-consume-"
                                + i);
            this.created = false;
            notifyAll();
        }

        try
        {
            wait();
        } catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}

/**
 * 生产者与消费者的基类
 *
 * @author <a href='wangwenjun62@gmail.com'>wangwenjun</a>
 */
abstract class AbsFactory implements Runnable
{
    protected Factory factory = null;

    public AbsFactory(Factory factory)
    {
        this.factory = factory;
    }

    abstract protected void execute();

    public void run()
    {
        while (true)
        {
            execute();
            // try
            // {
            // Thread.sleep(1);
            // } catch (InterruptedException e)
            // {
```

```
        // }

    }

}

class Producer extends AbsFactory
{

    public Producer(Factory factory)
    {
        super(factory);
    }

    @Override
    protected void execute()
    {
        factory.create();
    }
}

class Consumer extends AbsFactory
{

    public Consumer(Factory factory)
    {
        super(factory);
    }

    @Override
    protected void execute()
    {
        factory.consume();
    }
}

public class ProducerCustomer
{
    public static void main(String[] args)
    {
        if (args.length == 0)
        {
            System.out.println("Usage:java ProducerCustomer number");
            System.out.println("Please restart again...");
        }
    }
}
```

```
        System.exit(0);
    }

    int count = 0;
    try
    {
        count = Integer.parseInt(args[0]);
    } catch (Throwable t)
    {
        System.out.println("Please enter a integer type number...");
        System.exit(0);
    }

    Factory factory = new Factory();

    for (int i = 0; i < count; i++)
    {
        new Thread(new Producer(factory)).start();
        new Thread(new Consumer(factory)).start();
    }
}
```

系统输出信息:

```
Thread-0-create-96545
Thread-3-consume-96545
Thread-2-create-96546
Thread-3-consume-96546
Thread-2-create-96547
Thread-1-consume-96547
Thread-0-create-96548
Thread-3-consume-96548
```

分析说明:

其实根据以前的知识点,现在完全可以解释上一个程序为什么解决了并发的生产者消费者模型,读者可以自行一步一步的分析,其实生产者消费者在日常的工作中经常会被使用到,比如做通讯项目,接受线程接收到信息存放到信息队列,然后有一个消费进行队列中数据的消费,原始的方式就是队列的轮询,这种方式比较占用 CPU 资源,效率也比较低下,如果采用生产者消费者的方式则将是一个非常不错的选择;

第三节 Object 类 wait, notify, notifyAll 详解

看了本章中的大家可以看到有多处 wait, notify, notifyAll 函数的使用，在本节中我们将对这三个函数进行详细的解释和说明

4.3.1 wait 详解

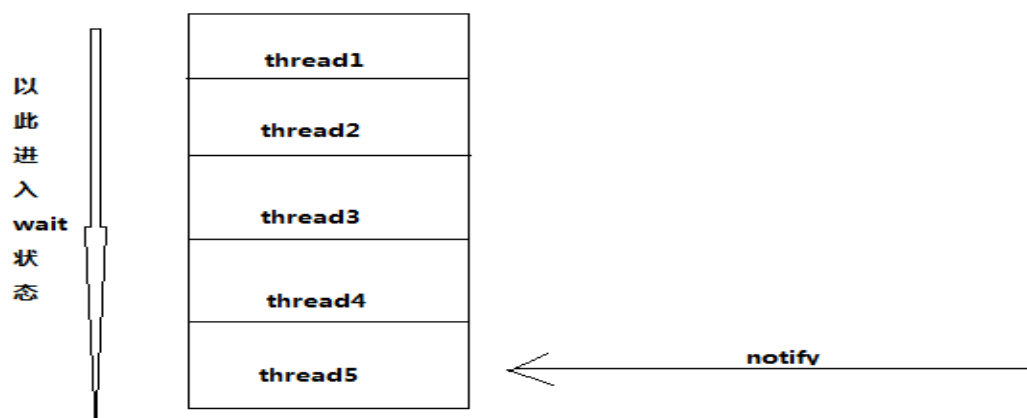
wait 方法和之前的 sleep 一样就是放弃 CPU 执行权，但是他和 sleep 不一样的地方是需要等待另外一个持有相同锁的线程对其进行唤醒操作，并且 wait 方法必须有一个同步锁，否则会抛出一个异常 `java.lang.IllegalMonitorStateException: current thread not owner`

4.3.2 notify 详解

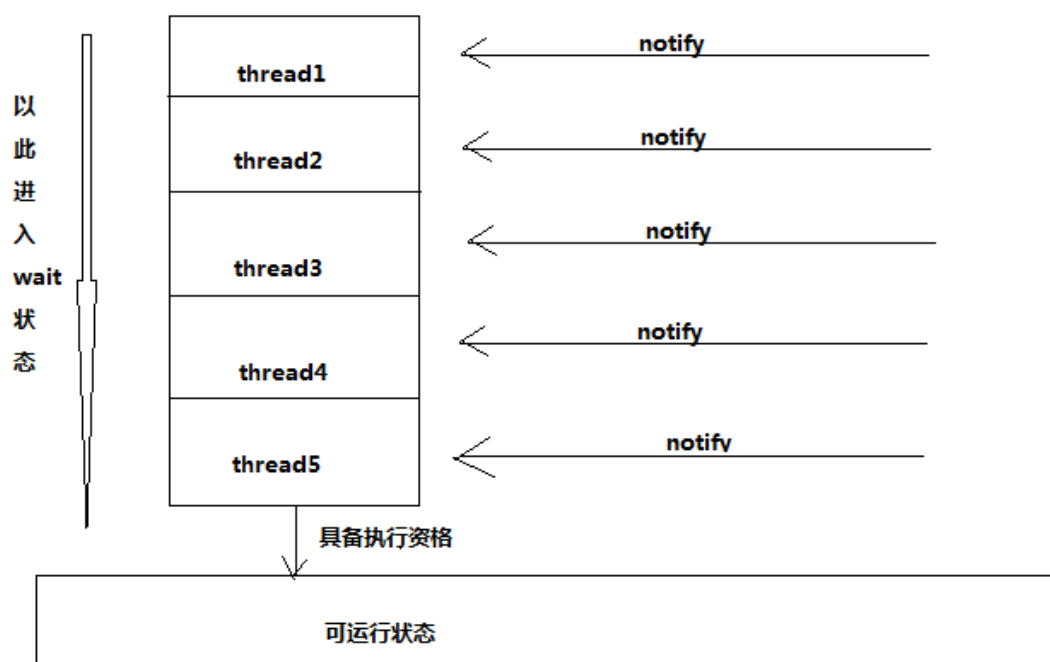
notify 方法就是将之前处在临时状态的线程唤醒，并且获取执行权，等待 CPU 的再次调度，但是有一点需要注意的是必须和之前的 wait 方法用到的锁是同一个；

4.3.3 notifyAll 详解

在上一小节中我们看到 notify 方法是唤醒一个正处在阻塞状态的线程，那他到底唤醒的是谁呢？其实在 JVM 中也存在一个线程队列或者线程池的概念，我们看看下图中的表示，关于 wait 和 notify 中的线程两者均使用的是一把锁，否则将没有可以探讨的必要；



从该图中可以看出，`notify` 方法将严格按照 FIFO（先进先出）的方式唤醒在线程队列中的与自己持有同样一把锁的线程；通过上图读者应该很清楚 `notify` 的作用，那么 `notifyAll` 的作用是什么呢？请看下图



通过上图的描述，我们可以看出 `notifyAll` 方法是将所有 `wait` 中的线程都进行唤醒，当然前提就是唤醒的线程持有和自己一样的锁，否则将不能被唤醒；

4.3.4 实现秒表程序

好了在本节中我们用之前学习的 `wait`，`notify` 等技术来实现一个秒表程序，让读者看一下如何在日常的开发中使用这些 API

程序实现后的效果为：



程序代码：

```
package com.wenhuisoft.chapter4;

import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

/**
 * 秒表程序演示
 *
 * @author wangwenjun(Alex) 主要是为了加深对wait.notify方法的理解
 */
public class Stopwatch extends JFrame
{
    private static final long serialVersionUID =
-5212710477644044656L;

    private final JLabel label = new JLabel("00:00:00");

    private JButton button1 = new JButton("Start");

    private JButton button2 = new JButton("Suspend");

    private JButton button3 = new JButton("Clear");

    private JPanel panel1 = new JPanel();

    private JPanel panel2 = new JPanel();

    private CommandThread command = null;

    private final ClockDisplay clockDisplay = new ClockDisplay();

    public Stopwatch()
    {
        super("秒表小程序");
    }
}
```

```
public void init()
{
    setLayout(new BorderLayout());
    setSize(250, 120);
    panel1.setLayout(new FlowLayout(FlowLayout.CENTER));
    panel2.setLayout(new FlowLayout(FlowLayout.CENTER));

    panel1.add(label);
    command = new CommandThread(clockDisplay, label);
    panel2.add(button1);
    panel2.add(button2);
    panel2.add(button3);

    button1.addActionListener(command);
    button2.addActionListener(command);
    button3.addActionListener(command);
    add(BorderLayout.NORTH, panel1);
    add(BorderLayout.CENTER, panel2);
    setVisible(true);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
}

public static void main(String[] args)
{
    new Stopwatch().init();
}

class CommandThread implements Runnable, ActionListener
{
    private boolean flag = true;

    private ClockDisplay clockDisplay = null;

    private JLabel label = null;

    private Thread t = null;

    private boolean hasStart = false;

    private boolean start = false;

    public CommandThread(ClockDisplay display, JLabel label)
```

```
{
    this.clockDisplay = display;
    this.label = label;
    t = new Thread(this);
}

public void run()
{
    while (flag)
    {
        try
        {
            start();
            Thread.sleep(1);
        } catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}

private void start()
{
    synchronized (clockDisplay)
    {
        if(start)
        {
            label.setText(clockDisplay.refresh());
        }
        else
        {
            try
            {
                clockDisplay.wait();
            } catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}

public void close()
```

```
{
    this.flag = false;
}

private void command(String type)
{
    synchronized (clockDisplay)
    {
        if (type.equalsIgnoreCase("start")) // 启动
        {
            start = true;
            if(!hasStart)
            {
                t = new Thread(this);
                hasStart = true;
                t.start();
            }
            clockDisplay.notify();
        } else if (type.equalsIgnoreCase("clear")) // 暂停
        {
            start = false;
            clockDisplay.clear();
            label.setText(clockDisplay.toString());
        } else
        {
            start = false;
        }
    }
}

public void actionPerformed(ActionEvent e)
{
    command(e.getActionCommand());
}
}

class ClockDisplay
{
    private int minute = 0;

    private int second = 0;
```

```
private int millisecond = 0;

public ClockDisplay()
{
}

public void clear()
{
    this.minute = 0;
    this.second = 0;
    this.millisecond = 0;
}

public String refresh()
{
    if (millisecond >= 999)
    {
        second++;
        millisecond = 0;
    } else
    {
        millisecond++;
    }

    if (second >= 59)
    {
        minute++;
        second = 0;
    }

    return toString();
}

public String toString()
{
    return String.format("%02d", minute) + ":"
        + String.format("%02d", second) + ":"
        + String.format("%03d", millisecond);
}
}
```

如果使用已经过时的 API [resume\(\)](#) 和 [suspend\(\)](#) 比较容易实现, 但是该 API 已经被标注

过时了，所以我们就是用 wait 和 notify 进行；

4.3.5 小节

其实上述三个方法在线程通讯的过程中非常常用，但是他们有一个共同的特点，那就是，他们必须都进行代码的同步或者方法级别的同步，否则将会出现 `current thread not owner`

第五章 守护线程与线程的优先级

本章中的内容其实在平时的工作开发中也会经常用到，但是重要性远不止前面总结的内容，因此需要知道有这样的特性和方法，在实际的开发过程中可以继续深造；

第一节 守护线程

什么叫做守护线程，你可以简单的将其理解为一个后台线程，他的特点主要是，主线程一旦运行结束，它就会随之结束，不管运行没运行完毕，都会随之结束，好了，我们直接写一段代码进行演示，然后重点进行拆分和分析；

```
package com.wenhuisoft.chapter5;

public class DaemonThreadTest
{
    public static void main(String[] args)
    {
        Thread t1=new Thread(){
            public void run()
            {
                int i = 0;
                while(i++<=1000)
                {
                    System.out.println("jjjjjjjjjjjjjjj:"+i);
                }
            }
        };
    }
};
```

```
t1.start();

int i = 0;
while(i++<=10)
{
    System.out.println("iiiiiiiiii:"+i);
}
}
```

上述的代码我们再也熟悉不过了，创建了一个线程 **t1** 并且执行之，在上述代码中有两个线程一个是 **main** 函数另外一个为 **t1**，**main** 函数执行完毕之后，程序还没有退出，而是当 **t1** 执行完毕之后，整个程序才算运行结束；

我们稍作改动，请看下面的代码，并且运行之

```
Thread t1=new Thread() {
    public void run()
    {
        int i = 0;
        while(i++<=1000)
        {
            System.out.println("jjjjjjjjjjjjjjjj:"+i);
        }
    }
};

t1.setDaemon(true);
t1.start();
```

运行之后我们发现，当主线程退出的时候，不管 **t1** 运行到什么时候都必须无条件的退出，这就是我们所说的守护线程，守护线程的设置也是相当简单，只需要将线程的 **Daemon** 设置为 **true** 即可

第二节 线程的 yield

线程的 **yield** 方法就是短暂放弃 CPU 执行权，但是它刹那点就和其他线程争抢 CPU 执行权；

```
Thread t1=new Thread() {
    public void run()
    {
```



```
        int i = 0;
        while(i++<=2000)
        {
            System.out.println("iiiiiiiiiiiiiiii:" + i);
        }
    };

    t1.start();
    t1.yield();
    Thread t2=new Thread() {
        public void run()
        {
            int i = 0;
            while(i++<=1000)
            {
                System.out.println("jjjjjjjjjjjjjjjj:" + i);
            }
        }
    };

    t2.start();
```

第三节 线程的停止

线程的停止，之前调用 `stop` 方法可以停止，但是该方法目前也被过期掉了，因为它存在线程安全问题，但是我们也有自己的方法让线程停止，我们一般在线程的 `run` 方法中会是一个死循环，因此线程的停止一般有两种方式

✚ Run 方法中的业务逻辑执行完毕；

✚ 死循环退出；

第四节 线程的优先级

线程的优先级别为 5，没有一个线程但是我们可以通过设置提高或者降低线程的优先级别，所谓线程的优先级别高就是获得 CPU 执行权的几率高，但是企图通过线程优先级设置来进行业务的控制这个是不可行的

```
void setPriority(int newPriority)
```

第五节 线程 Join

通过字面意思就可以理解到，线程的 `Join` 方法就是临时加入一个线程，等到该线程执行结束之后才能运行主线程，好了我们进行一下代码测试

```
package com.wenhuisoft.chapter5;

public class JoinTest
{
    public static void main(String[] args) throws
InterruptedException
    {
        Thread t1 = new Thread()
        {
            public void run()
            {
                int i = 0;
                while(i++<1000)
                {
                    try
                    {
                        Thread.sleep(100);
                    } catch (InterruptedException e)
                    {
                        e.printStackTrace();
                    }
                    System.out.println("t1:"+i);
                }
            }
        };
        t1.start();
        t1.join();
        for(int i = 0;i<1000;i++)
        {
            Thread.sleep(100);
            System.out.println("main:"+i);
        }
    }
}
```

原本是 `main` 和 `tf` 交替进行输出，但是由于 `t1` 是 `join` 进来的，因此需要等到 `join` 完全执行完毕之后才能执行 `main` 方法中的信息，如果我们在写一个线程 `t2`，如果 `t1.join()` 之后，

由于个人能力有限，书中难免会有偏颇之处，希望读到这本书的朋友能够给予我指导和批评，欢迎你们的指正

t2 是否也需要等待 t1 执行完毕呢，答案是否定的，这也是我进行测试之后得出的结论，读者也可以自行进行测试；

第六节 线程的 interrupt

Interrupt 方法的作用就是将处在阻塞中的线程打断，也就是线程将从阻塞状态转换到临时状态或者其他状态，执行该方法会抛出一个异常，也就是 wait 方法或者 sleep 方法中我们经常需要捕获的异常 InterruptedException

第六章 线程池的实现

熟练多线程编程的人都应该知道，线程的创建和销毁是比较消耗系统性能的，所以如何将已创建线程再次复用就可以避免线程创建和销毁带来的消耗，因此本章中我们一起来讨论线程池这一机制，并且我们将逐一的完善一个我们自己的线程池，让读者慢慢的对线程池这一技术有一个比较深入和理性的认识；

第一节 线程组

在学习线程池之前我们先来了解一下线程组的概念，以及如何使用线程组；

6.1.1 什么是线程组

线程组顾名思义就是一组线程的意思，将一组线程存放在一个组里面，方便管理，方便监控，相比 Thread，ThreadGroup 的使用并不是那么频繁，说实话我在日常的工作中也几乎很少用到 ThreadGroup，但是既然说到线程组，借着这个机会我也好好学习一把；

1、线程组创建方式一

```
public static void main(String[] args)
{
    ThreadGroup tg = new ThreadGroup("tg1");
    Thread t1 = new Thread(tg, "线程1")
```

```
{
    @Override
    public void run()
    {
        while(true)
        {

System.out.println("t1...."+getThreadGroup().getName());
            try
            {
                Thread.sleep(1000);
            } catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
};

Thread t2 = new Thread(tg, "线程2"){
    @Override
    public void run()
    {
        while(true)
        {
            System.out.println("t2....");
            try
            {
                Thread.sleep(100);
            } catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
};

Thread t3 = new Thread(tg, "线程3"){
    @Override
    public void run()
    {
        while(true)
        {
            System.out.println("t3....");
            try
```

```
        {
            Thread.sleep(100);
        } catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}

};

t1.start();

System.out.println("Active Count:"+tg.activeCount());

t2.start();
System.out.println("Active Count:"+tg.activeCount());
t3.start();
System.out.println("Active Count:"+tg.activeCount());
tg.list();
}
```

可以看到,我们创建了一个线程组 **tg** 并且创建了 3 个线程将其增加到了 **tg** 中,这样 **t1**, **t2**, **t3** 就同属于一个组,可以通过 **tg** 的观察和管理来进行 **t1**, **t2**, **t3**; 当然这只是创建线程组的第一种方式,通过 API 的学习,我们来发现还有另外一种方式来进行线程组的创建;

2、线程组创建方式二

```
ThreadGroup tg2 = new ThreadGroup(tg,"tg2");
```

可以看到线程组的创建可以将另外一个线程组作为参数传递进去,该线程组就称之为父线程组,在该线程组中可以通过 **getParent()** 方法获取父线程组,当然也可以查看到父线程组中线程的状态等;

6.1.2 线程组的 API 详解

1、获取线程组中活跃的线程

int	activeCount()
-----	-------------------------------

由于个人能力有限,书中难免会有偏颇之处,希望读到这本书的朋友能够给予我指导和批评,欢迎你们的指正

	返回此线程组中活动线程的估计数。
int	activeGroupCount () 返回此线程组中活动线程组的估计数。

2、将线程组的线程拷贝到线程数组中

int	enumerate (Thread[] list) 把此线程组及其子组中的所有活动线程复制到指定数组中。
int	enumerate (Thread[] list, boolean recurse) 把此线程组中的所有活动线程复制到指定数组中。
int	enumerate (ThreadGroup[] list) 把对此线程组中的所有活动子组的引用复制到指定数组中。
int	enumerate (ThreadGroup[] list, boolean recurse) 把对此线程组中的所有活动子组的引用复制到指定数组中。

可以看到有四个方法能够实现这样的需求，我们着重讨论一下第一个就可以了，其他的读者可以自行进行测试，在测试之前我比较好奇的一点是，拷贝后的线程数组是单纯的拷贝还是克隆呢？带着这个疑问我们一起进行一下测试

```
package com.wenhuisoft.chapter6;

public class ThreadGroupCopy
{
    public static void main(String[] args)
    {
        //local inner class extends Thread
        class MyThread extends Thread
        {
            public MyThread(ThreadGroup tg,String name)
            {
                super(tg,name);
            }

            private boolean flag = true;
            @Override
            public void run()
            {
                while(flag)
                {
                    try
```

由于个人能力有限，书中难免会有偏颇之处，希望读到这本书的朋友能够给予我指导和批评，欢迎你们的指正

```
        {  
  
        System.out.println(currentThread().getName());  
            Thread.sleep(1000);  
        } catch (InterruptedException e)  
        {  
            e.printStackTrace();  
        }  
    }  
}  
  
    public void close()  
    {  
        this.flag = false;  
    }  
}  
  
    ThreadGroup tg = new ThreadGroup("tg");  
    MyThread t1 = new MyThread(tg, "t1");  
    t1.start();  
    MyThread t2 = new MyThread(tg, "t2");  
    t2.start();  
    MyThread t3 = new MyThread(tg, "t3");  
    t3.start();  
  
    //copy  
  
    Thread[] ts = new Thread[tg.activeCount()];  
    System.out.println("current active  
Count:"+tg.activeCount());  
    tg.enumerate(ts);  
    for(Thread t:ts)  
    {  
        ((MyThread)t).close();  
        break;  
    }  
    while(true)  
    {  
        System.out.println("current active  
Count:"+tg.activeCount());  
    }  
}  
}
```

通过上述代码的演示我们不经看到了如何进行线程的拷贝，又深入的体会到了 `enumerate` 方法其实就是线程引用的拷贝，并不是深入克隆，因此在以后的开发中需要注意到；

3、其他 API

static int	<code>activeCount()</code> 返回当前线程的线程组中活动线程的数目。
void	<code>checkAccess()</code> 判定当前运行的线程是否有权修改该线程。
int	<code>countStackFrames()</code> 已过时。 该调用的定义依赖于 <code>suspend()</code> ，但它遭到了反对。此外，该调用的结果从来都不是意义明确的。
static Thread	<code>currentThread()</code> 返回对当前正在执行的线程对象的引用。
void	<code>destroy()</code> 已过时。 该方法最初用于破坏该线程，但不作任何清除。它所保持的任何监视器都会保持锁定状态。不过，该方法决不会被实现。即使要实现，它也极有可能以 <code>suspend()</code> 方式被死锁。如果目标线程被破坏时保持一个保护关键系统资源的锁，则任何线程在任何时候都无法再次访问该资源。如果另一个线程曾试图锁定该资源，则会出现死锁。这类死锁通常会证明它们自己是“冻结”的进程。有关更多信息，请参阅 为何 Thread.stop、Thread.suspend 和 Thread.resume 遭到反对？ 。
static void	<code>dumpStack()</code> 打印当前线程的堆栈跟踪。
static int	<code>enumerate(Thread[] tarray)</code> 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。
static Map<Thread, StackTraceElement[]>	<code>getAllStackTraces()</code> 返回所有活动线程的堆栈跟踪的一个映射。
ClassLoader	<code>getContextClassLoader()</code> 返回该线程的上下文 ClassLoader。
static Thread.UncaughtExceptionHandler	<code>getDefaultUncaughtExceptionHandler()</code> 返回线程由于未捕获到异常而突然终止时调用的默认处理程序。

long	<code>getId()</code> 返回该线程的标识符。
<code>String</code>	<code>getName()</code> 返回该线程的名称。
int	<code>getPriority()</code> 返回线程的优先级。
<code>StackTraceElement[]</code>	<code>getStackTrace()</code> 返回一个表示该线程堆栈转储的堆栈跟踪元素数组。
<code>Thread.State</code>	<code>getState()</code> 返回该线程的状态。
<code>ThreadGroup</code>	<code>getThreadGroup()</code> 返回该线程所属的线程组。
<code>Thread.UncaughtExceptionHandler</code>	<code>getUncaughtExceptionHandler()</code> 返回该线程由于未捕获到异常而突然终止时调用的处理程序。
static boolean	<code>holdsLock(Object obj)</code> 当且仅当当前线程在指定的对象上保持监视器锁时，才返回 true。
void	<code>interrupt()</code> 中断线程。
static boolean	<code>interrupted()</code> 测试当前线程是否已经中断。
boolean	<code>isAlive()</code> 测试线程是否处于活动状态。
boolean	<code>isDaemon()</code> 测试该线程是否为守护线程。
boolean	<code>isInterrupted()</code> 测试线程是否已经中断。
void	<code>join()</code> 等待该线程终止。
void	<code>join(long millis)</code> 等待该线程终止的时间最长为 millis 毫秒。
void	<code>join(long millis, int nanos)</code> 等待该线程终止的时间最长为 millis 毫秒 + nanos 纳秒。
void	<code>resume()</code> 已过时。 该方法只与 <code>suspend()</code> 一起使用，但 <code>suspend()</code> 已经遭到反对，因为它具有死锁倾向。有关更多信息，请参阅 为何 Thread.stop、

	Thread.suspend 和 Thread.resume 遭到反对? 。
void	run() 如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对象的 run 方法；否则，该方法不执行任何操作并返回。
void	setContextClassLoader(ClassLoader cl) 设置该线程的上下文 ClassLoader。
void	setDaemon(boolean on) 将该线程标记为守护线程或用户线程。
static void	setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh) 设置当线程由于未捕获到异常而突然终止，并且没有为该线程定义其他处理程序时所调用的默认处理程序。
void	setName(String name) 改变线程名称，使之与参数 name 相同。
void	setPriority(int newPriority) 更改线程的优先级。
void	setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh) 设置该线程由于未捕获到异常而突然终止时调用的处理程序。
static void	sleep(long millis) 在指定的毫秒数内让当前正在执行的线程休眠（暂停执行）。
static void	sleep(long millis, int nanos) 在指定的毫秒数加指定的纳秒数内让当前正在执行的线程休眠（暂停执行）。
void	start() 使该线程开始执行；Java 虚拟机调用该线程的 run 方法。
void	stop() 已过时。 该方法具有固有的不安全性。用 <code>Thread.stop</code> 来终止线程将释放它已经锁定的所有监视器（作为沿堆栈向上传播的未检查 <code>ThreadDeath</code> 异常的一个自然后果）。如果以前受这些监视器保护的任何对象都处于一种不一致的状态，则损坏的对象将对其他线程可见，这有可能导致任意的行为。 <code>stop</code> 的许多使用都应由只修改某些变量以指示目标线程应该停止运行的代码来取代。目标线程应定期检查该变量，并且如果该变量指示它要停止运行，则从其运行方法依次返回。如果目标线程等待很长时间（例如基于一个条件变量），则应使用 <code>interrupt</code> 方法来中断该等待。有关更多信息，请参阅 《为何不赞成使用 Thread.stop、Thread.suspend 和 Thread.resume?》 。
void	stop(Throwable obj) 已过时。 该方法具有固有的不安全性。请参阅 stop() 以获得详细

	<p>信息。该方法的附加危险是它可用于生成目标线程未准备处理的异常（包括若没有该方法该线程不太可能抛出的已检查的异常）。有关更多信息，请参阅为何 Thread.stop、Thread.suspend 和 Thread.resume 遭到反对？。</p>
void	<p>suspend()</p> <p>已过时。该方法已经遭到反对，因为它具有固有的死锁倾向。如果目标线程挂起时在保护关键系统资源的监视器上保持有锁，则在目标线程重新开始以前任何线程都不能访问该资源。如果重新开始目标线程的线程想在调用 resume 之前锁定该监视器，则会发生死锁。这类死锁通常会证明自己是“冻结”的进程。有关更多信息，请参阅为何 Thread.stop、Thread.suspend 和 Thread.resume 遭到反对？。</p>
String	<p>toString()</p> <p>返回该线程的字符串表示形式，包括线程名称、优先级和线程组。</p>
static void	<p>yield()</p> <p>暂停当前正在执行的线程对象，并执行其他线程。</p>

第二节 线程池雏形

线程池应该最起码具备的就是如下几个特点，在接下来的文字中我们将会围绕着这几点然后一一进行实现

- ✚ 任务队列；
- ✚ 线程管理者；
- ✚ 最大线程活跃数；
- ✚ 线程最小数；
- ✚ 线程最大数；

```
package com.wenhuisoft.chapter6.threadPool;

import java.util.LinkedList;
import java.util.List;

public class ThreadPoolManager
{
    static interface RunnableTask
    {
        public void run();
    }

    private int max_thread_size = 0;
```

由于个人能力有限，书中难免会有偏颇之处，希望读到这本书的朋友能够给予我指导和批评，欢迎你们的指正

```
private int min_thread_size = 0;

private int active thread size = 0;

private List<RunnableTask> runnableList = null; //任务队列

private ThreadTask[] tasks = null;

public ThreadPoolManager()
{
    this(2, 8, 6);
}

public ThreadPoolManager(int minSize, int maxSize, int
activeSize)
{
    this.min_thread_size = minSize;
    this.max_thread_size = maxSize;
    this.active_thread_size = activeSize;
    init();
}

private void init()
{
    runnableList = new
LinkedList<ThreadPoolManager.RunnableTask>();
    tasks = new ThreadTask[min_thread_size];
    for(int i = 0; i < min_thread_size; i++)
    {
        tasks[i] = new ThreadTask("-"+i, runnableList);
        tasks[i].start();
    }
}

public void execute(RunnableTask task)
{
    synchronized (runnableList)
    {
        ((LinkedList<RunnableTask>) runnableList).addLast(task);
        runnableList.notify();
    }
}
```

```
}

private class ThreadTask extends Thread
{

    private String name = "";

    private List<RunnableTask> runnableList = null;

    public ThreadTask(String name, List<RunnableTask>
runnableList)
    {
        super(name);
        this.name = name;
        this.runnableList = runnableList;
    }

    public void run()
    {
        while(true)
        {
            RunnableTask task = null;
            synchronized (runnableList)
            {
                while(runnableList.isEmpty())
                {
                    try
                    {
                        runnableList.wait();
                    } catch (InterruptedException e)
                    {
                        e.printStackTrace();
                    }
                }
            }

            task=((LinkedList<RunnableTask>)runnableList).removeFirst()
;

            task.run();
        }
    }
}
```

```
}
```

测试代码:

```
package com.wenhuisoft.chapter6.threadPool;

import
com.wenhuisoft.chapter6.threadPool.ThreadPoolManager.RunnableT
ask;

public class ThreadPoolTest1
{
    public static void main(String[] args)
    {
        final RunnableTask task = new RunnableTask()
        {
            public void run()
            {
                System.out.println(Thread.currentThread().getName()
                    + "...is execute...");
            }
        };

        ThreadPoolManager poolManager = new ThreadPoolManager();
        for (int i = 0; i < 10; i++)
        {
            poolManager.execute(task);
        }
    }
}
```

上述的代码我们实现了一个简易的线程池，并且通过了测试，如果我们现在就认定她是一个完成的线程池还为时尚早，因为线程池所具备一些基本特性还没有完全做到；

第三节 最大最小属性

虽然上述代码中我们定义了最小线程数，最大线程数，最大活跃线程数，但是基本上没有使用到，好了在本节中我们将来讨论如何使用这几个属性，在使用之前我们先来了解一下这几个属性所代表的意思是什么？

6.3.1 最小线程数

既然是线程池，里面的线程应该不止一个，因此它有若干个，也就是线程池初始化的时候需要创建的线程最小数

6.3.2 最大线程数

虽然线程中有很多个线程，但是也是有个极限的吧，因此最大线程就是限制线程池中最大的线程数，那么当线程池中的线程已经不能满足任务时，这个时候需要采取哪些策略呢？当然方式有很多种，等待，抛出异常告知调用者，放入任务队列中等

6.3.3 最大活跃线程数

其实前面的两个概念比较容易理解，但是这个就稍微有些抽象了，那么到底什么是最大活跃线程数呢，这个严格的来说也没有什么规范，我就根据自己的理解来解析这一个概念，所谓最大活跃数是这样的一个属性，当线程池的线程需要超过最小线程数时，他需要增加到一个不超过最大线程数的值，这个时候他就重新动态的开辟一个线程，当线程的需求量不是太大的时候，线程池就有义务负责销毁线程释放 CPU，内存等资源，那么应该如何释放这些线程呢？那就释放到活跃线程数的这个值；

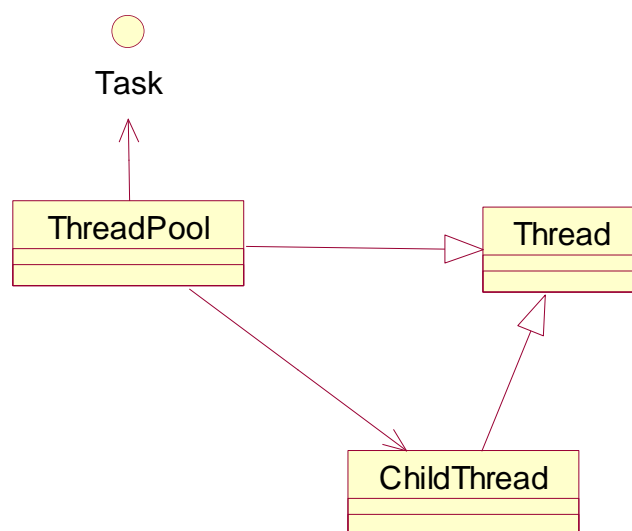
6.3.4 属性之间的关系

通过上述文字的描述，想必大家已经清楚了这三个值之间的关系了，没错，那就是下面的不等关系

最小线程数 \leq 最大活跃线程数 \leq 最大线程数

当然一个线程池中的属性可能还会有一些其他的，但是我们暂且就实现这三个，其他的大家可以参阅 [JDK 线程池源码](#)

6.3.5 加入属性之后的线程池实现



线程池结构图

实现代码如下：

```
package com.wenhuisoft.chapter6.threadPool2;

import java.util.LinkedList;
import java.util.List;

/**
 * thread pool itself is a thread(it not represent other programming
 * have same
 * realize)
 *
 * @author wangwenjun
 * @date 2012-03-23
 */
public class ThreadPool extends Thread
{
    private final static int DEFAULT_MIN_THREAD_SIZE = 5; /*
                                                    * minimum default
                                                    * thread size value
                                                    */

    private final static int DEFAULT_ACTIVE_THREAD_SIZE = 8; /*
                                                    * maximum
                                                    default

```



```

* thread active
* size value
*/

    private final static int DEFAULT_MAX_THREAD_SIZE = 10; /*
default                                     * maximum
value                                     * thread size
                                           */

    private int min_thread_size = 0;

    private int active_thread_size = 0;

    private int max_thread_size = 0;

    private List<Task> taskQuene = null;

    private List<ChildThread> pools = null;

    private boolean destory = false;

    public ThreadPool()
    {
        this(DEFAULT_MIN_THREAD_SIZE, DEFAULT_ACTIVE_THREAD_SIZE,
            DEFAULT_MAX_THREAD_SIZE);
    }

    public ThreadPool(int min_thread_size, int active_thread_size,
        int max_thread_size)
    {
        this.min_thread_size = min_thread_size;
        this.active_thread_size = active_thread_size;
        this.max_thread_size = max_thread_size;
        createPool();
    }

    private void createPool()
    {
        taskQuene = new LinkedList<Task>();
        pools = new LinkedList<ChildThread>();
        for (int i = 0; i < min_thread_size; i++)

```

```
{
    ChildThread t = new ChildThread(taskQuene);
    ((LinkedList<ChildThread>) pools).add(t);
    t.start();
}
this.start();
}

private Object lock = new Object();

public void run()
{
    while (!destory)
    {
        if (getFreeThreadSize() >= active_thread_size)
        {
            ((LinkedList<ChildThread>) pools).removeLast();
            System.out.println("destory one thread");
            synchronized(lock)
            {
                try
                {
                    lock.wait(1000);
                } catch (InterruptedException e)
                {
                    e.printStackTrace();
                }
            }
        } else
        {
            System.out.println("task number less than active
size");
            synchronized(lock)
            {
                try
                {
                    lock.wait(3000);
                } catch (InterruptedException e)
                {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

```
    }
}

/**
 * destory thread pool and close all child thread in thread pool,
close
 * every thread for loop
 */
public void destory()
{
    this.destory = false; // close thread pool thread
    synchronized (taskQuene)
    {
        for (ChildThread t : pools)
        {
            t.close(); // close child thread one by one
            t.interrupt();
        }
    }
}

/**
 * <pre>
 *      &nbsp;execute method is add a task to queue in fact
 *      sometimes task number more than maximum thread number in
pool, we have some way to deal with it
 *      for example:
 *      1、 we can throw an exception notify invoker thread pool is
full!
 *      2、 we can store task in a list and wait free thread hand
it
 *      3、 we also can make invoker wait some seconds and try again!
 *      4、 others solutions
 *      in our thread pool we use method 2 to realize it;
 * </pre>
 *
 * @param task
 *          Task type
 * @version 1.0.0.0
 * @since 1.0.0.0
 */
public void execute(Task task)
{
```

```
        int threadSize = pools.size();
        if (threadSize >= max_thread_size)
        {
            // because we use way 2 to deal with task number more
            // thread pool limit size,so we can ignore it and add it
            // queue
            System.out.println("more than thread pool size....");
        }

        if (threadSize >= min_thread_size && threadSize
        < max_thread_size)
        {
            // create new thread to process extra task
            createNewThread();
        }

        synchronized (taskQuene)
        {
            ((LinkedList<Task>) taskQuene).addFirst(task);
            taskQuene.notify();
        }
    }

    private void createNewThread()
    {
        System.out.println("begin create new thread!");
        ChildThread t = new ChildThread(taskQuene);
        ((LinkedList<ChildThread>) pools).add(t);
        t.start();
        System.out.println("new thread have creat success!");
    }

    /**
     * get thread pool current running thread size
     *
     * @return running thread size
     */
    public int getRunningThreadSize()
    {
        int count = 0;
        synchronized (taskQuene)
```

```
{
    for (ChildThread c : pools)
    {
        if (c.getCurrentThreadState())
        {
            count++;
        }
    }
}
System.out.println("running count:" + count);
return count;
}

public int getFreeThreadSize()
{
    int count = 0;
    synchronized (taskQuene)
    {
        for (ChildThread c : pools)
        {
            if (!c.getCurrentThreadState())
            {
                count++;
            }
        }
    }
    System.out.println("free count:" + count);
    return count;
}

/**
 * <p>
 * every thread must implement this interface
 * </p>
 *
 * @author wangwenjun
 * @date 2012-03-23
 */
public static interface Task
{
    public void run();
}
```

```
/**
 * current runner state thread
 *
 * @author wangwenjun
 * @date 2012-03-23
 */
private class ChildThread extends Thread
{
    private boolean state = false;

    private boolean closed = false;

    private List<Task> pools = null;

    public ChildThread(List<Task> pools)
    {
        System.out.println("Thread Name:"+getName());
        this.pools = pools;
    }

    public void run()
    {
        while (!closed)
        {
            Task task = null;
            synchronized (pools)
            {
                if (pools.isEmpty())
                {
                    try
                    {
                        pools.wait();
                    } catch (InterruptedException e)
                    {
                    }
                } else
                {
                    task = ((LinkedList<Task>)
pools).removeLast();
                }
            }

            state = true; // identify this thread is running state
            if (null != task)
```

```
        task.run();
        state = false; /*
                           * set this thread have finished task but
no
                           * stop
                           */
    }
}

/**
 * close thread by set boolean variable
 */
public void close()
{
    closed = true;
}

/**
 * get worker thread current state
 *
 * @return
 */
public boolean getCurrentThreadState()
{
    return state;
}
}
```

测试代码如下

```
package com.wenhuisoft.chapter6.threadPool2;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class ThreadPoolTest
{

    private ThreadPool pool = null;

    @Before
```

```
public void init()
{
    pool = new ThreadPool(10,15,30);    //default value
}

@After
public void release()
{
    try
    {
        Thread.sleep(5000);
    } catch (InterruptedException e)
    {
        e.printStackTrace();
    }
    pool.destory();
    pool = null;
}

@Test
public void testnormal()
{
    for(int i = 0;i<50;i++)
    {
        TestTask tt = new TestTask();
        pool.execute(tt);
    }
}

private static class TestTask implements ThreadPool.Task
{
    public void run()
    {
        try
        {
            Thread.sleep(30);
        } catch (InterruptedException e)
        {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName());
    }
}
```



```
}  
}
```

第四节 任务队列属性

其实任务队列的属性我们在前面已经有所接触，我们的任务都是存放在队列之中，但是严格来说，一个任务队列应该还有如下的一些最基本特点：

- ✚ 任务队列最大任务数
- ✚ 超过最大任务数该如何处理；
- ✚ 任务队列中的任务状态监控；

我们按照上面的内容对任务队列进行简单的修改，加入以上的特点，我们依然使用 `LinkedList` 作为我们的队列方案，只是少做了一些简单的封装，其实读者可以做得很好，我只是做了一点点的改动而已

```
package com.wenhuisoft.chapter6.threadPool2;  
  
import java.util.LinkedList;  
import java.util.List;  
  
import com.wenhuisoft.chapter6.threadPool2.ThreadPool.Task;  
  
public class TaskQueue  
{  
  
    private int max_value = 0;  
  
    private int index = 0;  
  
    private List<ThreadPool.Task> queue = new  
LinkedList<ThreadPool.Task>();  
  
    public TaskQueue(int max_value)  
    {  
        if (max_value <= 0)  
            throw new IllegalArgumentException(  
                "task queue length should not less than zero!");  
  
        this.max_value = max_value;  
    }  
}
```

```
public void addFirst(Task task) throws ExceedMaxException
{
    if (index >= max_value)
        throw new ExceedMaxException("task queue current size:
"
                                     + index + " exceed maxium value: " + max_value);
    index++;
    ((LinkedList<Task>)queue).addFirst(task);
}

public Task removeLast()
{
    index--;
    return ((LinkedList<Task>)queue).removeLast();
}

class ExceedMaxException extends Exception
{
    private static final long serialVersionUID =
2194487581753097550L;

    private String message;

    private Throwable t;

    public ExceedMaxException()
    {
        this(null, null);
    }

    public ExceedMaxException(String message)
    {
        this(message, null);
    }

    public ExceedMaxException(String message, Throwable t)
    {
        super(message, t);
        this.message = message;
        this.t = t;
    }
}
```

```
public String getMessage()
{
    return message;
}

public void setMessage(String message)
{
    this.message = message;
}

public Throwable getT()
{
    return t;
}

public void setT(Throwable t)
{
    this.t = t;
}
}
```

第七章 线程状态的监控

所谓线程的状态监控就是指通过回调或者监听的手段，得知当前运行线程运行的状况，启动，运行中，正常结束，异常结束等状况，那么我们将在本章中重点来讨论这几个问题，并且我们也会通过实例来演示如何监控线程的状态

第一节 线程状态的监控

7.1.1 线程状态监控接口

监控状态监听器接口代码如下

```
package com.wenhuisoft.chapter7;

public interface ThreadListener
{
```

```
/**
 * when thread business start this method will invoke
 * @param args
 * @return
 */
public Object threadStart(Object[] args);

/**
 * running
 * @param args
 * @return
 */
public Object threadRunning(Object[] args);

/**
 * normal finish
 * @param args
 * @return
 */
public Object threadFinish(Object[] args);

/**
 * exception
 * @param args
 * @return
 */
public Object threadException(Object[] args);
}
```

读者可以在此基础之上进行追加或者重构，考虑到可能会传递参数和返回值，我们专门在每个方法上都设定了入参和出参；

7.1.2 Runnable 的封装

对线程任务的包装类，代码如下

```
package com.wenhuisoft.chapter7;

abstract public class RunnableWarper implements Runnable
{
```

```
private ThreadListener threadListener = null;

private final static ThreadListener DEFAULT_LISTENER = new
ThreadListener() {

    private String name = Thread.currentThread().getName();

    public Object threadStart(Object[] args)
    {
        System.out.println(name+" start...");
        return null;
    }

    public Object threadRunning(Object[] args)
    {
        System.out.println(name+" running...");
        return null;
    }

    public Object threadFinish(Object[] args)
    {
        System.out.println(name+" finish...");
        return null;
    }

    public Object threadException(Object[] args)
    {
        System.out.println(name+" exception...");
        return null;
    }

};

public RunnableWarper()
{
    this(DEFAULT_LISTENER);
}

public RunnableWarper(ThreadListener listener)
{
    this.threadListener = listener;
}
```

```
public void run()
{
    try
    {
        threadListener.threadStart(null);
        handler();
        threadListener.threadFinish(null);
    } catch (Exception e)
    {
        threadListener.threadException(null);
    }
}

abstract public void handler();
}
```

为了让使用者必须使用线程的监控，专门设定了一个模板，对 `Runnable` 接口进行了简单的封装，这样就可以将整个线程的状态监控点进行植入；

7.1.3 测试代码

```
package com.wenhuisoft.chapter7;

public class Test extends RunnableWarper
{
    @Override
    public void handler()
    {
        System.out.println("business handler");
    }

    public static void main(String[] args)
    {
        Thread t = new Thread(new Test());
        t.start();

        Thread t1 = new Thread(new Test());
        t1.start();
    }
}
```

7.1.4 详解设计思路

- ✚ 设计一个接口提供了监听者被回调的动作（ThreadListener）；
- ✚ 设计一个 Warpper 封装 Runnable 接口，强制注入线程状态监控的代码(RunnableWarper)；

第二节 线程出现异常捕获

当线程出现异常的时候，也就是面临死亡时，他会告知 JVM 自己要挂掉了，我们同样可以对其尽心监控，获取到线程死亡的信息，并且触发相应的动作；我们在本节中进行一下演示，其实这个思路 JDK 在设计的时候已经考虑到了，并且帮助我们进行了很好的设计；

```
package com.wenhuisoft.chapter7;

public class FetalException
{

    static class MyRunnable implements Runnable
    {
        public void run()
        {
            throw new Error();
        }
    }

    public static void main(String[] args)
    {
        Thread t = new Thread(new MyRunnable());
        t.setUncaughtExceptionHandler(new
Thread.UncaughtExceptionHandler()
        {
            public void uncaughtException(Thread t, Throwable e)
            {
                System.out.println(t.getName());
                System.out.println(e);
            }
        });

        t.start();
    }
}
```



可以看到，扮演这个比较重要角色的接口是 `Thread.UncaughtExceptionHandler`，运行之后我们发现，线程意外死亡被我们很好的不会到了

第八章 总结

一个月的时间，终于完成了这本书的编写，由于工作的原因，业余时间不够多，所以每天都要拿出仅有的一点时间来完善该电子书，在该书中，几乎涉及了 `Java` 中线程方面的全部知识，包括：

- 1、线程的创建，启动，停止等；
- 2、线程间的通讯；
- 3、锁机制（`this` 锁，静态锁）
- 4、生产者消费者，多线程情况下的生产者消费者，电子秒表等；
- 5、线程池的逐步演化；
- 6、线程状态的监控；

第九章 近期推出

-  最近打算乘热打铁，继续线程的下一本电子书，其中涵盖的内容将更加丰富，包括线程的常用设计思想，线程更进一步的探讨，`JDK5.0` 提供的并发包探讨和学习；
-  `Junit3.0` 源码剖析，很早之前读完了 `Junit` 源码，除了崇拜就是崇拜，代码之优美，模式的灵活使用都让人折服，因此有必要重温一下也和大家共同探讨交流一下；

Programming 系列丛书附录

书名	创建时间	内容简介
《Java Annotation》	2009 年	讲解 <code>Java</code> 中的 <code>Annotation</code> 技术
《Java JDBC》	2010 年	全面讲解几乎所有的 <code>JDBC API</code>
《IO 编程》	2011 年	讲解并且深入分析 <code>IO</code> 中的各个技术细节
《Ant 入门》	2009 年	讲解 <code>Ant</code> 的基本用法
《Java 与反射》	2009 年	讲解 <code>Java</code> 中的反射 <code>API</code> ，并且对如何使用进行了

由于个人能力有限，书中难免会有偏颇之处，希望读到这本书的朋友能够给予我指导和批评，欢迎你们的指正

		详细的分析
《Java 与 enumeration》	2012 年	全面细致的讲解枚举中的所有技术细节
《Java 多线程编程深入详解》	2012 年	全面的讲解 Java 中的多线程编程，结合作者工作和学习多年的知识，该书中涵盖了大量的技术细节