**第1章 Java多线程技能(Thread API)**

1.1 进程和多线程的概念及线程的优点

* **进程:**



一个正在操作系统上执行的exe程序。

* **线程**

进程中独立运行的子任务。

**单任务:**排队执行，即同步。CPU利用率大幅降低。

**1.2 使用多线程**

JVM创建调用main()的线程

1.2.1 继承Thread类

* Public class Thread implements Runnable {}
* CPU以不确定的方式(随机时间)调用线程中run方法。

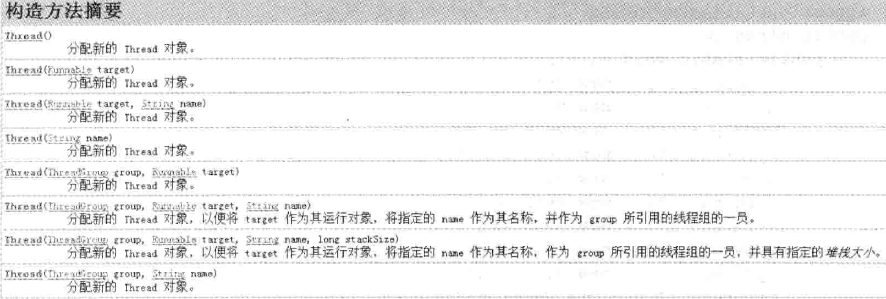
如多次调用start(),会出现Exception in thread “main” java.lang. IllegalThreadStateException

* start()通知”线程规划器”此线程已准备就绪，等待调用线程对象的run()方法，就是让系统安排时间来调用Thread中的run()；线程启动顺序与start()执行顺序无关。
* 如用thread.run()，是同步，此线程对象不交给”线程规划器”处理，而是由main线程调用run()

1.2.2 实现Runnable接口

如欲创建的线程类已有父类，就不能再继承Thread类。

* Thread构造函数：



* 例子:
* Thread(Runnable target, String name)
* Thread(Runnable target)：因Thread implements Runnable，因此也可传入Thread类对象

|  |
| --- |
| package com.thread.myrunable;  public class MyRunable implements Runnable {  @Override  public void run() {  System.out.println("MyRunable运行中!");  }  } |

|  |
| --- |
| package com.thread.myrunable;  public class Run {  public static void main(String[] args) {  MyRunable myRunable = new MyRunable();  Thread thread = new Thread(myRunable);//Thread(Runnable target)  thread.start();  System.out.println("main()运行结束！");  }  } |

结果：

|  |
| --- |
| main()运行结束！  MyRunable运行中! |

1.2.3 【补】实例变量与线程安全

1.2.4 【补】留意i——与System.out.println（）的异常

**1.3** 【补】**currentThread（）方法**

**1.4**【补】 **isAlive（）方法**

**1.5** 【补】**sleep（）方法**

**1.6** 【补】**getId（）方法**

**1.7 停止线程**

* 终止正在运行的线程:

1. run方法完成后线程正常终止
2. Thread.stop()：不安全，且已被废弃，最好不用。
3. Thread.interrupted()

1.7.1 停止不了的线程：interrupt ()

Thread.interrupt ()仅是在当前线程中打了一个停止标记，并不是真正停止线程。

1.7.8 使用return(+interrupt)停止线程

* 例子:

|  |
| --- |
| package com.thread.sleepinterupt;  public class MyThrrad extends Thread{  @Override  public void run() {  while(true) {  if(this.isInterrupted()) {  System.out.println("停止了！");  return;  }  System.out.println("timer=" + System.currentTimeMillis());  }  }  } |

|  |
| --- |
| package com.thread.sleepinterupt;  public class Run2 {  public static void main(String[] args) {  MyThrrad myThrrad = new MyThrrad();  myThrrad.start();    try {  Thread.sleep(300);  } catch (InterruptedException e) {  e.printStackTrace();  }    myThrrad.interrupt();  }  } |

结果:

|  |
| --- |
| timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  停止了！ |

* 总结:

但还是建议用”抛异常”的方法停止线程，因在catch中还可将异常往上抛，使线程停止事件得到传播。

1.7.2 判断线程是否是停止状态

* Thread.interrupted()：测试当前线程是否已中断

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

调用后，线程的中断状态由该方法清除。

* thread.isInterrupted()：测试线程Thread对象是否已中断

|  |
| --- |
| public boolean isInterrupted() {  return isInterrupted(false);  } |

调用后，未清除线程的中断状态。

1.7.3 【补】能停止的线程——异常法

1.7.4 sleep()状态下停止线程

* 例子

|  |
| --- |
| package com.thread.sleepinterupt;  public class MyThrrad extends Thread{  @Override  public void run() {  super.run();  try {  System.out.println("run begin");  Thread.sleep(200000);  System.out.println("run end");  } catch (InterruptedException e) {  System.out.println("在沉睡中停止！进入catch:" + this.isInterrupted());  e.printStackTrace();  }  }  } |

|  |
| --- |
| package com.thread.sleepinterupt;  public class Run2 {  public static void main(String[] args) {  try {  MyThrrad myThrrad = new MyThrrad();  myThrrad.start();  Thread.sleep(200);  myThrrad.interrupt();  } catch (InterruptedException e) {  System.out.println("main catch");  e.printStackTrace();  }  System.out.println("main end!");  }  } |

结果：

|  |
| --- |
| run begin  main end!  在沉睡中停止！进入catch:false  java.lang.InterruptedException: sleep interrupted  at java.lang.Thread.sleep(Native Method)  at com.thread.sleepinterupt.MyThrrad.run(MyThrrad.java:11) |

* 总结:

在sleep状态下停止某一线程，会抛异常，进入catch语句，且清除停止状态，变为false

1.7.5 【补】能停止的线程——暴力停止(stop())

1.7.6 stop()与java.lang.ThreadDeath异常

* 例子:

|  |
| --- |
| package com.thread.sleepinterupt;  public class MyThrrad extends Thread{  @Override  public void run() {  try {  this.~~stop();~~  } catch (ThreadDeath e) {  System.out.println("进入了catch方法！");  e.printStackTrace();  }  }  } |

调用代码(略),结果：

|  |
| --- |
| 进入了catch方法！  java.lang.ThreadDeath  at java.lang.Thread.stop(Thread.java:836)  at com.thread.sleepinterupt.MyThrrad.run(MyThrrad.java:8) |

* 总结:
* 调用stop()会抛出java.lang.ThreadDeath异常，但通常不用显式捕捉。

//public class ThreadDeath extends Error

* stop()已被废弃，因强制停止：

可能使一些清理性工作得不到完成。

另一情况是对锁定对象”解锁”,导致数据得不到同步处理，导致数据不一样问题。

1.7.7【补】 释放锁的不良后果

**1.8 暂停线程（**suspend- resume**）**

1.8.1 【补】suspend与resume方法的使用

suspend(停止)[已废弃] –resume(恢复)

1.8.2 【补】suspend与resume方法的缺点—独占

使用不当（如获得锁对象后suspend），易造成公共同步对象的独占，其它线程无法访问公共同步对象。

1.8.3 suspend与resume方法的缺点—不同步

**1.9 yield方法**

作用:放弃当前CPU资源，让给其它任务区去占用CPU执行时间。

但放弃的时间不确定，可能刚放弃，马上又获得。

**1.10线程的优先级：**setPriority()

public final void setPriority(int newPriority)

优先级较高的线程得到CPU资源较多。

newPriority：1-10, 不在此范围抛出异常:throw new IllegalArgumentException()

|  |
| --- |
| myThrrad.setPriority(Thread.MIN\_PRIORITY);  3个常量:  public final static int MIN\_PRIORITY = 1;  public final static int NORM\_PRIORITY = 5;  public final static int MAX\_PRIORITY = 10; |

1.10.1 【代码补】线程优先级的继承特性

如A线程启动B线程，B的线程优先级同A

1.10.2 优先级具有规则性

高优先级的线程总是大部分先执行完。

当线程优先级差距很大，低优先级先调用，却不一定先执行完

即:线程优先级与代码执行顺序无关，CPU尽量将执行资源让给优先级较高的线程。

1.10.3 优先级具有随机性和不确定性

即优先级较高的线程不一定每次都先执行完。

1.10.4 看谁运行得快

**1.11 守护线程**

|  |
| --- |
| 用户线程  守护线程 |

守护线程：

仅当进程中不存在非守护线程，则守护线程自动销毁。

作用：为其它线程的运行提供便利服务

典型:垃圾回收线程

|  |
| --- |
| MyThrrad myThrrad = new MyThrrad();  myThrrad.setDaemon(true); |

**第2章 对象及变量的并发访问**

**2.1 synchronized同步方法**

“线程安全”:获得的实例变量的值经过了同步处理，不会出现脏读。

2.1.1 方法内的变量为线程安全

“非线程安全”存在于实例变量中，方法内部的私有变量，不存在该问题。

2.1.2 【可写代码】实例变量非线程安全

如多个线程共同访问1个对象中的实例变量，可能出现非线程安全问题

用线程访问的对象中如有多个实例变量，结果可能交叉。

如对象只有一个实例变量，可能覆盖

2.1.3 多个对象多个锁

Synchronized取得的是对象锁，不是把一段代码或方法(函数)当作锁。

如多个线程访问多个对象，JVM会创建多个锁。

2.1.4 synchronized方法与锁对象

1个对象一个锁

* 结论:

1. A线程先持有object对象的Lock锁，B线程可异步方式调用object对象中的非

Synchronized类型方法。

1. A线程先持有object对象的Lock锁，B线程若在此时调用object对象的Synchronized方法则需等待，即同步

2.1.5 脏读

2.1.4中已实现多个线程调用同一方法时，用Synchronized进行同步。虽在赋值时同步，取值时可能出意外，即脏读。

脏读：在读取实例变量时，此值已被其它线程更改了。

解决方法：在取值(getValue())方法前加上synchronized



2.1.6 synchronized锁重入(代码P65)

当一个线程得到一个对象锁，再次请求此对象锁是可得到的。

* 锁重入:自己可再次获得自己的内部锁。

如不可锁重入，会造成死锁。

* 锁重入也支持在父子类继承的环境中(代码P67)。

即子类(的同步方法)可通过“可重入锁”调用父类的同步方法。

2.1.7 出现异常，锁自动释放

当一个线程执行代码出现异常，其持有的锁会自动释放。

2.1.8 同步不可继承(看代码P69)

父类的方法A是synchronized的，子类重写该方法，不能继承synchronized，需在子类方法中添加synchronized

**2.2 synchronized同步语句块**

2.2.1 synchronized方法的弊端(看代码P74)

弊端:运行时间长

2.2.2 synchronized同步代码块的使用(synchronized(this))

2.2.3 用同步代码块解决同步方法的弊端(看代码P76)

缩短时间，加快运行效率。

2.2.4 一半异步，一半同步

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

2.2.5 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，其它线程对同一个object中所有其它synchronized(this)同步代码块的访问将被阻塞，说明synchronized使用的”对象监视器”是同一个。

2.2.6 synchronized（this）同步代码块锁定当前对象(看代码P82)

* 多个线程调用同一对象中不同名称的synchronized方法或synchronized同步代码块，调用效果是按顺序执行，是同步的，阻塞的。即锁定当前对象的。
* synchronized方法或synchronized(this):

1. 对其它同步方法或synchronized(this)同步代码块呈阻塞状态
2. 同一时间只有一个线程可执行同步方法/同步代码块

2.2.7 将任意对象作为对象监视器

2.2.8 细化验证3个结论

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

2.2.10 数据类型String的常量池特性

2.2.11 同步synchronized方法无限等待与解决

2.2.12 多线程的死锁

2.2.13 内置类与静态内置类

2.2.14 内置类与同步：实验1

2.2.15 内置类与同步：实验2

2.2.16 锁对象的改变

**2.3 volatile关键字**

2.3.1 关键字volatile与死循环

2.3.2 解决同步死循环

2.3.3 解决异步死循环

2.3.4 volatile非原子的特性

2.3.5 使用原子类进行i++操作

2.3.6 原子类也并不完全安全

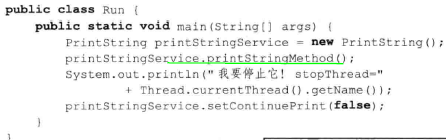
2.3.7 synchronized代码块有volatile同步的功能

**2.3 volatile关键字**

作用:使变量在多个线程间可见。

**2.3.1死循环**

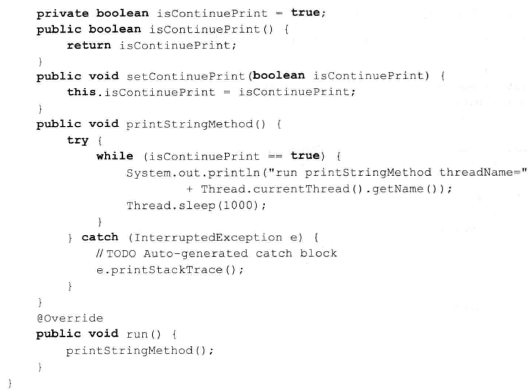




程序陷入死循环，原因:main线程一直在处理while()循环,程序不能执行后续代码。

**2.3.2解决同步死循环(但会出现异步死循环)**







运行在-server服务器模式中64bit的JVM上时，会出现死循环.

**2.3.3 解决异步死循环**

**2.4 本章总结**

**第3章 线程间通信**

**3.1 等待/通知机制**

3.1.1 不使用等待/通知机制实现线程间通信

3.1.2 什么是等待/通知机制

3.1.3 等待/通知机制的实现

3.1.4 方法wait（）锁释放与notify（）锁不释放

3.1.5 当interrupt方法遇到wait方法

3.1.6 只通知一个线程

3.1.7 唤醒所有线程

3.1.8 方法wait（long）的使用

3.1.9 通知过早

3.1.10 等待wait的条件发生变化

3.1.11 生产者/消费者模式实现

3.1.12 通过管道进行线程间通信：字节流

3.1.13 通过管道进行线程间通信：字符流

3.1.14 实战：等待/通知之交叉备份

**3.2 方法join的使用**

3.2.1 学习方法join前的铺垫

3.2.2 用join（）方法来解决

3.2.3 方法join与异常

3.2.4 方法join（long）的使用

3.2.5 方法join（long）与sleep（long）的区别

3.2.6 方法join（）后面的代码提前运行：出现意外

3.2.7 方法join（）后面的代码提前运行：解释意外

**3.3 类ThreadLocal的使用**

3.3.1 方法get（）与null

3.3.2 验证线程变量的隔离性

3.3.3 解决get（）返回null问题

3.3.4 再次验证线程变量的隔离性

**3.4 类InheritableThreadLocal的使用**

3.4.1 值继承

3.4.2 值继承再修改

**3.5 本章总结**

**第4章 Lock的使用**

4.1 使用ReentrantLock类

4.1.1 使用ReentrantLock实现同步：测试1

4.1.2 使用ReentrantLock实现同步：测试2

4.1.3 使用Condition实现等待/通知错误用法与解决

4.1.4 正确使用Condition实现等待/通知

4.1.5 使用多个Condition实现通知部分线程：错误用法

4.1.6 使用多个Condition实现通知部分线程：正确用法

4.1.7 实现生产者/消费者模式：一对一交替打印

4.1.8 实现生产者/消费者模式：多对多交替打印

4.1.9 公平锁与非公平锁

4.1.10 方法getHoldCount（）、getQueueLength（）和getWaitQueueLength（）的测试

4.1.11 方法hasQueuedThread（）、hasQueuedThreads（）和hasWaiters（）的测试

4.1.12 方法isFair（）、isHeldByCurrentThread（）和isLocked（）的测试

4.1.13 方法lockInterruptibly（）、tryLock（）和tryLock（long timeout，TimeUnit unit）的测试

4.1.14 方法awaitUninterruptibly（）的使用

4.1.15 方法awaitUntil（）的使用

4.1.16 使用Condition实现顺序执行

4.2 使用ReentrantReadWriteLock类

4.2.1 类ReentrantReadWriteLock的使用：读读共享

4.2.2 类ReentrantReadWriteLock的使用：写写互斥

4.2.3 类ReentrantReadWriteLock的使用：读写互斥

4.2.4 类ReentrantReadWriteLock的使用：写读互斥

4.3 本章总结

**第5章 定时器Timer**

5.1 定时器Timer的使用

5.1.1 方法schedule（TimerTask task， Date time）的测试

5.1.2 方法schedule（TimerTask task， Date firstTime， long period）的测试

5.1.3 方法schedule（TimerTask task， long delay）的测试

5.1.4 方法schedule（TimerTask task， long delay， long period）的测试

5.1.5 方法scheduleAtFixedRate（TimerTask task， Date firstTime， long period）的测试

5.2 本章总结

**第6章 单例模式与多线程**

6.1 立即加载/"饿汉模式"

6.2 延迟加载/"懒汉模式"

6.3 使用静态内置类实现单例模式

6.4 序列化与反序列化的单例模式实现

6.5 使用static代码块实现单例模式

6.6 使用enum枚举数据类型实现单例模式

6.7 完善使用enum枚举实现单例模式

6.8 本章总结

**第7章 拾遗增补**

7.1 线程的状态

7.1.1 验证NEW、RUNNABLE和TERMINATED

7.1.2 验证TIMED\_WAITING

7.1.3 验证BLOCKED

7.1.4 验证WAITING

7.2 线程组

7.2.1 线程对象关联线程组：1级关联

7.2.2 线程对象关联线程组：多级关联

7.2.3 线程组自动归属特性

7.2.4 获取根线程组

7.2.5 线程组里加线程组

7.2.6 组内的线程批量停止

7.2.7 递归与非递归取得组内对象

7.3 使线程具有有序性

7.4 SimpleDateFormat非线程安全

7.4.1 出现异常

7.4.2 解决异常方法1

7.4.3 解决异常方法2

7.5 线程中出现异常的处理

7.6 线程组内处理异常

7.7 线程异常处理的传递

7.8 本章总结

第一章



**1.2.3 实例变量与线程安全**

**1.2.3.1 不共享数据**

(1)

|  |
| --- |
| package com.thread.www;  public class MyThread3 extends Thread{  private int count = 5;  public MyThread3(String name) {  super();  this.setName(name);//设置线程名称  }  @Override  public void run() {  super.run();  while(count > 0) {  count--;  System.out.println("由" + this.currentThread().getName() + "计算，count=" +  count );  }  }  } |

(2)

|  |
| --- |
| package test;  import com.thread.www.MyThread3;  public class RunMyThread3 {  public static void main(String[] args) {  MyThread3 a = new MyThread3("A");  MyThread3 b = new MyThread3("B");  MyThread3 c = new MyThread3("C");  a.start();  b.start();  c.start();  }  } |

(3)结果:一共创建了3线程，每个线程都有各自的count变量，自己减少自己的count变量。变量不共享

|  |
| --- |
| 由A计算，count=3  由A计算，count=2  由A计算，count=1  由A计算，count=0  由C计算，count=4  由B计算，count=4  由B计算，count=3  由B计算，count=2  由B计算，count=1  由B计算，count=0  由C计算，count=3  由C计算，count=2  由C计算，count=1  由C计算，count=0 |

**1.2.3.2 共享数据**

1.2.3.2.1 例1:出现非线程安全问题

(1) MyThread4.java

|  |
| --- |
| package com.thread.www;  public class MyThread4 extends Thread {  private int count = 5;  @Override  public void run() {  super.run();  count--;  //此示例不要用for语句，因为使用同步后，其它线程就得不到线程运行的机会了，  //会一直由一个线程做减法运算  System.out.println("由" + this.currentThread().getName() + "计算count=" + count);  }  } |

(2) TestMyThread4.java

|  |
| --- |
| package test;  import com.thread.www.MyThread4;  public class TestMyThread4 {  public static void main(String[] args) {  MyThread4 myThread4 = new MyThread4();  Thread a = new Thread(myThread4, "a");  Thread b = new Thread(myThread4, "b");  Thread c = new Thread(myThread4, "c");  Thread d = new Thread(myThread4, "d");  Thread e = new Thread(myThread4, "e");  a.start();  b.start();  c.start();  d.start();  e.start();  }  } |

(3)结果: c和b线程同时对count进行处理，产生”非线程安全” 问题

|  |
| --- |
| 由a计算count=4  由c计算count=2  由b计算count=2  由e计算count=1  由d计算count=0 |

1.2.3.2.2 例2:

需要使多个线程间同步，即按顺序排队的方式进行减1操作

(1) 在例1基础上，更改 MyThread4.java代码如下：

|  |
| --- |
| package com.thread.www;  public class MyThread4 extends Thread {  private int count = 5;  @Override  synchronized public void run() {  super.run();  count--;  //此示例不要用for语句，因为使用同步后，其它线程就得不到线程运行的机会了，  //会一直由一个线程做减法运算  System.out.println("由" + this.currentThread().getName() + "计算count=" + count);  }  } |

(2)结果:

|  |
| --- |
| 由a计算count=4  由b计算count=3  由d计算count=2  由c计算count=1  由e计算count=0 |

synchronized使多个线程在执行run方法时，以排队方式处理。当一个线程调用run前，先判断run方法有没被上锁，若上锁，说明有其它线程在调用run方法，必须等其它线程对run方法调用结束后才可执行run方法。

Synchronized可在任意对象及方法前加锁，被加锁的代码称为”互斥区”或“临界区”。

如果不能拿到锁，该线程会不断的去尝试拿这把锁，直到能够拿到为止，且多个线程会同时争抢这把锁。

非线程安全:多个线程对同一个对象中的同一个实例变量进行操作时，会出现值更改、值不同步的情况，进而影响程序执行流程。

**1.2.4 留意i—与System.out.println()的异常**

1. MyThread5.java

|  |
| --- |
| package com.thread.www;  public class MyThread5 extends Thread {  private int i = 5;  @Override  public void run() {  System.out.println("i=" + (i--) + "threadname=" + Thread.currentThread().getName());  }  } |

2. TestMyThread5.java

|  |
| --- |
| package test;  import com.thread.www.MyThread5;  public class TestMyThread5 {  public static void main(String[] args) {  MyThread5 myThread5 = new MyThread5();  Thread a = new Thread(myThread5);  Thread b = new Thread(myThread5);  Thread c = new Thread(myThread5);  Thread d = new Thread(myThread5);  Thread e = new Thread(myThread5);  a.start();  b.start();  c.start();  d.start();  e.start();  }  } |

3.结果

|  |
| --- |
| i=5threadname=Thread-1  i=3threadname=Thread-4  i=4threadname=Thread-2  i=2threadname=Thread-3  i=1threadname=Thread-5 |

虽然println()方法在内部是同步的，但i--操作是在println之前发生的，所以有非线程同步的问题的概率。

|  |
| --- |
| public void println(String x) {  synchronized (this) {  print(x);  newLine();  }  } |

**1.3 currentThread()**

**3.3 ThreadLocal**

<http://blog.csdn.net/lufeng20/article/details/24314381>