# Java 多线程编程

Java 给多线程编程提供了内置的支持。 一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

多线程是多任务的一种特别的形式，但多线程使用了更小的资源开销。

这里定义和线程相关的另一个术语 - 进程：一个进程包括由操作系统分配的内存空间，包含一个或多个线程。一个线程不能独立的存在，它必须是进程的一部分。一个进程一直运行，直到所有的非守护线程都结束运行后才能结束。

多线程能满足程序员编写高效率的程序来达到充分利用 CPU 的目的。

## 一个线程的生命周期

线程是一个动态执行的过程，它也有一个从产生到死亡的过程。

下图显示了一个线程完整的生命周期。



* **新建状态:**

使用 **new** 关键字和 **Thread** 类或其子类建立一个线程对象后，该线程对象就处于新建状态。它保持这个状态直到程序**start()** 这个线程。

* **就绪状态:**

当线程对象调用了start()方法之后，该线程就进入就绪状态。就绪状态的线程处于就绪队列中，要等待JVM里线程调度器的调度。

* **运行状态:**

如果就绪状态的线程获取 CPU 资源，就可以执行 **run()**，此时线程便处于运行状态。处于运行状态的线程最为复杂，它可以变为阻塞状态、就绪状态和死亡状态。

* **阻塞状态:**

如果一个线程执行了sleep（睡眠）、suspend（挂起）等方法，失去所占用资源之后，该线程就从运行状态进入阻塞状态。在睡眠时间已到或获得设备资源后可以重新进入就绪状态。可以分为三种：

* + 等待阻塞：运行状态中的线程执行 wait() 方法，使线程进入到等待阻塞状态。
  + 同步阻塞：线程在获取 synchronized 同步锁失败(因为同步锁被其他线程占用)。
  + 其他阻塞：通过调用线程的 sleep() 或 join() 发出了 I/O 请求时，线程就会进入到阻塞状态。当sleep() 状态超时，join() 等待线程终止或超时，或者 I/O 处理完毕，线程重新转入就绪状态。
* **死亡状态:**

一个运行状态的线程完成任务或者其他终止条件发生时，该线程就切换到终止状态。

## 线程的优先级

每一个 Java 线程都有一个优先级，这样有助于操作系统确定线程的调度顺序。

Java 线程的优先级是一个整数，其取值范围是 1 （Thread.MIN\_PRIORITY ） - 10 （Thread.MAX\_PRIORITY ）。

默认情况下，每一个线程都会分配一个优先级 NORM\_PRIORITY（5）。

具有较高优先级的线程对程序更重要，并且应该在低优先级的线程之前分配处理器资源。但是，线程优先级不能保证线程执行的顺序，而且非常依赖于平台。

## 创建一个线程

Java 提供了三种创建线程的方法：

* 通过实现 Runnable 接口；
* 通过继承 Thread 类本身；
* 通过 Callable 和 Future 创建线程。

## 通过实现 Runnable 接口来创建线程

创建一个线程，最简单的方法是创建一个实现 Runnable 接口的类。

为了实现 Runnable，一个类只需要执行一个方法调用 run()，声明如下：

public void run()

你可以重写该方法，重要的是理解的 run() 可以调用其他方法，使用其他类，并声明变量，就像主线程一样。

在创建一个实现 Runnable 接口的类之后，你可以在类中实例化一个线程对象。

Thread 定义了几个构造方法，下面的这个是我们经常使用的：

Thread(Runnable threadOb,String threadName);

这里，threadOb 是一个实现 Runnable 接口的类的实例，并且 threadName 指定新线程的名字。

新线程创建之后，你调用它的 start() 方法它才会运行。

void start();

下面是一个创建线程并开始让它执行的实例：

## 实例

class RunnableDemo implements Runnable {

private Thread t;

private String threadName;

RunnableDemo( String name) {

threadName = name;

System.out.println("Creating " + threadName );

}

public void run() {

System.out.println("Running " + threadName );

try {

for(int i = 4; i > 0; i--) {

System.out.println("Thread: " + threadName + ", " + i); // 让线程睡眠一会

Thread.sleep(50);

}

}catch (InterruptedException e) {

System.out.println("Thread " + threadName + " interrupted.");

}

System.out.println("Thread " + threadName + " exiting.");

}

public void start () {

System.out.println("Starting " + threadName );

if (t == null) {

t = new Thread (this, threadName);

t.start ();

}

}

}

public class TestThread {

public static void main(String args[]) {

RunnableDemo R1 = new RunnableDemo( "Thread-1");

R1.start();

RunnableDemo R2 = new RunnableDemo( "Thread-2"); R2.start();

}

}

编译以上程序运行结果如下：

Creating Thread-1

Starting Thread-1

Creating Thread-2

Starting Thread-2

Running Thread-1

Thread: Thread-1, 4

Running Thread-2

Thread: Thread-2, 4

Thread: Thread-1, 3

Thread: Thread-2, 3

Thread: Thread-1, 2

Thread: Thread-2, 2

Thread: Thread-1, 1

Thread: Thread-2, 1

Thread Thread-1 exiting.

Thread Thread-2 exiting.

## 通过继承Thread来创建线程

创建一个线程的第二种方法是创建一个新的类，该类继承 Thread 类，然后创建一个该类的实例。

继承类必须重写 run() 方法，该方法是新线程的入口点。它也必须调用 start() 方法才能执行。

该方法尽管被列为一种多线程实现方式，但是本质上也是实现了 Runnable 接口的一个实例。

## 实例

class ThreadDemo extends Thread {

private Thread t;

private String threadName;

ThreadDemo( String name) {

threadName = name;

System.out.println("Creating " + threadName );

}

public void run() {

System.out.println("Running " + threadName );

try {

for(int i = 4; i > 0; i--) {

System.out.println("Thread: " + threadName + ", " + i); // 让线程睡眠一会

Thread.sleep(50);

}

}catch (InterruptedException e) {

System.out.println("Thread " + threadName + " interrupted.");

}

System.out.println("Thread " + threadName + " exiting.");

}

public void start () {

System.out.println("Starting " + threadName );

if (t == null) {

t = new Thread (this, threadName);

t.start ();

}

}

}

public class TestThread {

public static void main(String args[]) {

ThreadDemo T1 = new ThreadDemo( "Thread-1");

T1.start();

ThreadDemo T2 = new ThreadDemo( "Thread-2");

T2.start();

}

}

编译以上程序运行结果如下：

Creating Thread-1

Starting Thread-1

Creating Thread-2

Starting Thread-2

Running Thread-1

Thread: Thread-1, 4

Running Thread-2

Thread: Thread-2, 4

Thread: Thread-1, 3

Thread: Thread-2, 3

Thread: Thread-1, 2

Thread: Thread-2, 2

Thread: Thread-1, 1

Thread: Thread-2, 1

Thread Thread-1 exiting.

Thread Thread-2 exiting.

## Thread 方法

下表列出了Thread类的一些重要方法：

|  |  |
| --- | --- |
| **序号** | **方法描述** |
| 1 | **public void start()** 使该线程开始执行；**Java** 虚拟机调用该线程的 run 方法。 |
| 2 | **public 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。 |
| 3 | **public final void setName(String name)** 改变线程名称，使之与参数 name 相同。 |
| 4 | **public final void setPriority(int priority)**  更改线程的优先级。 |
| 5 | **public final void setDaemon(boolean on)** 将该线程标记为守护线程或用户线程。 |
| 6 | **public final void join(long millisec)** 等待该线程终止的时间最长为 millis 毫秒。 |
| 7 | **public void interrupt()** 中断线程。 |
| 8 | **public final boolean isAlive()** 测试线程是否处于活动状态。 |

测试线程是否处于活动状态。 上述方法是被Thread对象调用的。下面的方法是Thread类的静态方法。

|  |  |
| --- | --- |
| **序号** | **方法描述** |
| 1 | **public static void yield()** 暂停当前正在执行的线程对象，并执行其他线程。 |
| 2 | **public static void sleep(long millisec)** 在指定的毫秒数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。 |
| 3 | **public static boolean holdsLock(Object x)** 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。 |
| 4 | **public static Thread currentThread()** 返回对当前正在执行的线程对象的引用。 |
| 5 | **public static void dumpStack()** 将当前线程的堆栈跟踪打印至标准错误流。 |

### 实例

如下的ThreadClassDemo 程序演示了Thread类的一些方法：

DisplayMessage.java 文件代码：

// 文件名 : DisplayMessage.java

// 通过实现 Runnable 接口创建线程

public class DisplayMessage implements Runnable {

private String message;

public DisplayMessage(String message) {

this.message = message;

}

public void run() {

while(true) {

System.out.println(message);

}

}

}

GuessANumber.java 文件代码：

// 文件名 : GuessANumber.java

// 通过继承 Thread 类创建线程

public class GuessANumber extends Thread {

private int number;

public GuessANumber(int number) {

this.number = number;

}

public void run() {

int counter = 0;

int guess = 0;

do {

guess = (int) (Math.random() \* 100 + 1);

System.out.println(this.getName() + " guesses " + guess);

counter++;

} while(guess != number);

System.out.println("\*\* Correct!" + this.getName() + "in" + counter + "guesses.\*\*");

}

}

ThreadClassDemo.java 文件代码：

// 文件名 : ThreadClassDemo.java

public class ThreadClassDemo {

public static void main(String [] args) {

Runnable hello = new DisplayMessage("Hello");

Thread thread1 = new Thread(hello);

thread1.setDaemon(true);

thread1.setName("hello");

System.out.println("Starting hello thread...");

thread1.start();

Runnable bye = new DisplayMessage("Goodbye");

Thread thread2 = new Thread(bye);

thread2.setPriority(Thread.MIN\_PRIORITY);

thread2.setDaemon(true);

System.out.println("Starting goodbye thread...");

thread2.start();

System.out.println("Starting thread3...");

Thread thread3 = new GuessANumber(27);

thread3.start();

try {

thread3.join();

}catch(InterruptedException e) {

System.out.println("Thread interrupted.");

}

System.out.println("Starting thread4...");

Thread thread4 = new GuessANumber(75);

thread4.start();

System.out.println("main() is ending...");

}

}

运行结果如下，每一次运行的结果都不一样。

Starting hello thread...

Starting goodbye thread...

Hello

Hello

Hello

Hello

Hello

Hello

Goodbye

Goodbye

Goodbye

Goodbye

Goodbye

.......

## 通过 Callable 和 Future 创建线程

* 1. 创建 Callable 接口的实现类，并实现 call() 方法，该 call() 方法将作为线程执行体，并且有返回值。
* 2. 创建 Callable 实现类的实例，使用 FutureTask 类来包装 Callable 对象，该 FutureTask 对象封装了该 Callable 对象的 call() 方法的返回值。
* 3. 使用 FutureTask 对象作为 Thread 对象的 target 创建并启动新线程。
* 4. 调用 FutureTask 对象的 get() 方法来获得子线程执行结束后的返回值。

## 实例

public class CallableThreadTest implements Callable<Integer> {

public static void main(String[] args) {

CallableThreadTest ctt = new CallableThreadTest();

FutureTask<Integer> ft = new FutureTask<>(ctt);

for(int i = 0;i < 100;i++) {

System.out.println(Thread.currentThread().getName()+" 的循环变量i的值"+i);

if(i==20) {

new Thread(ft,"有返回值的线程").start();

}

}

try {

System.out.println("子线程的返回值："+ft.get());

} catch (InterruptedException e) {

e.printStackTrace();

} catch (ExecutionException e) {

e.printStackTrace();

}

}

@Override

public Integer call() throws Exception {

int i = 0;

for(;i<100;i++) {

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

} return i;

}

}

## 创建线程的三种方式的对比

* 1. 采用实现 Runnable、Callable 接口的方式创建多线程时，线程类只是实现了 Runnable 接口或 Callable 接口，还可以继承其他类。
* 2. 使用继承 Thread 类的方式创建多线程时，编写简单，如果需要访问当前线程，则无需使用 Thread.currentThread() 方法，直接使用 this 即可获得当前线程。

## 线程的几个主要概念

在多线程编程时，你需要了解以下几个概念：

* 线程同步
* 线程间通信
* 线程死锁
* 线程控制：挂起、停止和恢复

## 多线程的使用

有效利用多线程的关键是理解程序是并发执行而不是串行执行的。例如：程序中有两个子系统需要并发执行，这时候就需要利用多线程编程。

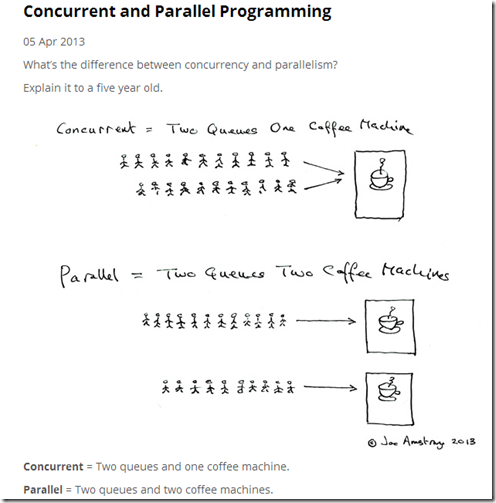
通过对多线程的使用，可以编写出非常高效的程序。不过请注意，如果你创建太多的线程，程序执行的效率实际上是降低了，而不是提升了。

请记住，上下文的切换开销也很重要，如果你创建了太多的线程，CPU 花费在上下文的切换的时间将多于执行程序的时间！

用多线程只有一个目的，那就是更好的利用cpu的资源，因为所有的多线程代码都可以用单线程来实现。说这个话其实只有一半对，因为反应“多角色”的程序代码，最起码每个角色要给他一个线程吧，否则连实际场景都无法模拟，当然也没法说能用单线程来实现：比如最常见的“生产者，消费者模型”。

很多人都对其中的一些概念不够明确，如同步、并发等等，让我们先建立一个数据字典，以免产生误会。

* 多线程：指的是这个程序（一个进程）运行时产生了不止一个线程
* 并行与并发：
  + 并行：多个cpu实例或者多台机器同时执行一段处理逻辑，是真正的同时。
  + 并发：通过cpu调度算法，让用户看上去同时执行，实际上从cpu操作层面不是真正的同时。并发往往在场景中有公用的资源，那么针对这个公用的资源往往产生瓶颈，我们会用TPS或者QPS来反应这个系统的处理能力。



并发与并行

* 线程安全：经常用来描绘一段代码。指在并发的情况之下，该代码经过多线程使用，线程的调度顺序不影响任何结果。这个时候使用多线程，我们只需要关注系统的内存，cpu是不是够用即可。反过来，线程不安全就意味着线程的调度顺序会影响最终结果，如不加事务的转账代码：
* void transferMoney(User from, User to, float amount){
* to.setMoney(to.getBalance() + amount);
* from.setMoney(from.getBalance() - amount);

}

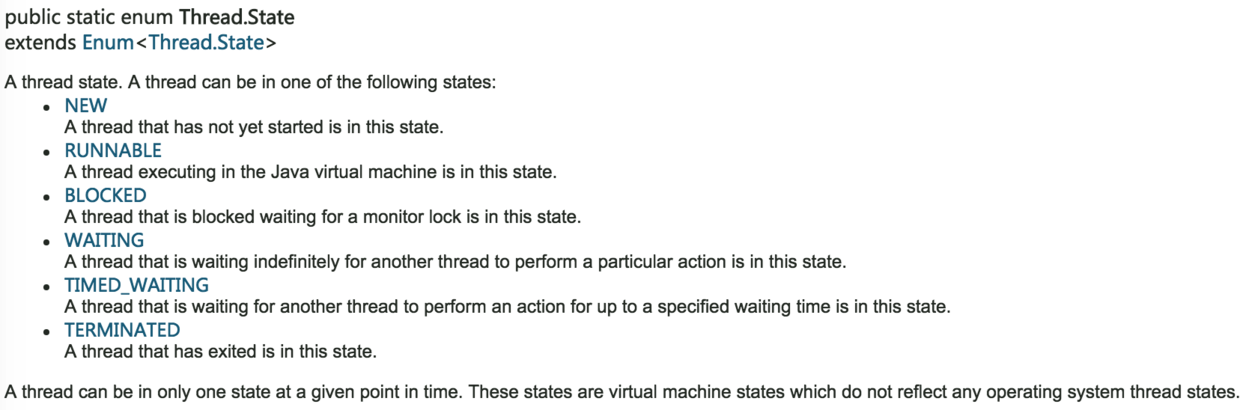
* 同步：Java中的同步指的是通过人为的控制和调度，保证共享资源的多线程访问成为线程安全，来保证结果的准确。如上面的代码简单加入@synchronized关键字。在保证结果准确的同时，提高性能，才是优秀的程序。线程安全的优先级高于性能。

好了，让我们开始吧。我准备分成几部分来总结涉及到多线程的内容：

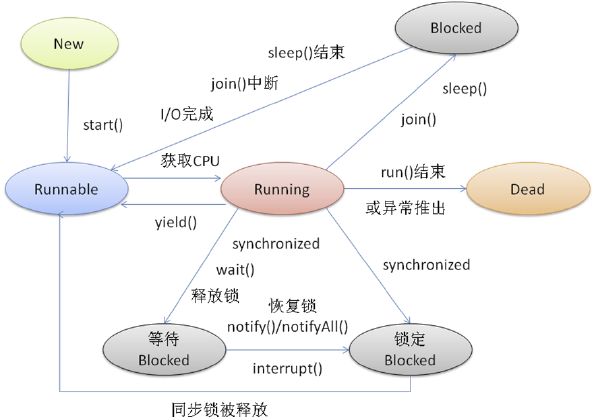
1. 扎好马步：线程的状态
2. 内功心法：每个对象都有的方法（机制）
3. 太祖长拳：基本线程类
4. 九阴真经：高级多线程控制类

**扎好马步：线程的状态**

先来两张图：



线程状态



线程状态转换

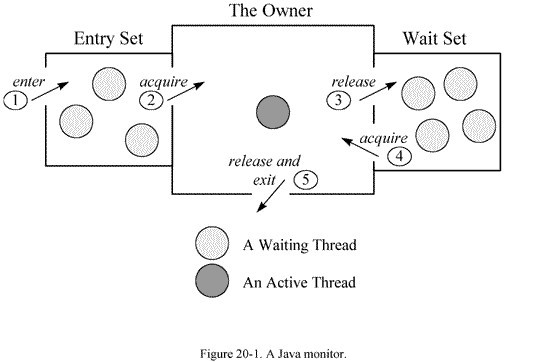
各种状态一目了然，值得一提的是"blocked"这个状态：  
线程在Running的过程中可能会遇到阻塞(Blocked)情况

1. 调用join()和sleep()方法，sleep()时间结束或被打断，join()中断,IO完成都会回到Runnable状态，等待JVM的调度。
2. 调用wait()，使该线程处于等待池(wait blocked pool),直到notify()/notifyAll()，线程被唤醒被放到锁定池(lock blocked pool )，释放同步锁使线程回到可运行状态（Runnable）
3. 对Running状态的线程加同步锁(Synchronized)使其进入(lock blocked pool ),同步锁被释放进入可运行状态(Runnable)。

此外，在runnable状态的线程是处于被调度的线程，此时的调度顺序是不一定的。Thread类中的yield方法可以让一个running状态的线程转入runnable。

**内功心法：每个对象都有的方法（机制）**

synchronized, wait, notify 是任何对象都具有的同步工具。让我们先来了解他们



monitor

他们是应用于同步问题的人工线程调度工具。讲其本质，首先就要明确monitor的概念，Java中的每个对象都有一个监视器，来监测并发代码的重入。在非多线程编码时该监视器不发挥作用，反之如果在synchronized 范围内，监视器发挥作用。

wait/notify必须存在于synchronized块中。并且，这三个关键字针对的是同一个监视器（某对象的监视器）。这意味着wait之后，其他线程可以进入同步块执行。

当某代码并不持有监视器的使用权时（如图中5的状态，即脱离同步块）去wait或notify，会抛出java.lang.IllegalMonitorStateException。也包括在synchronized块中去调用另一个对象的wait/notify，因为不同对象的监视器不同，同样会抛出此异常。

再讲用法：

* synchronized单独使用：
  + 代码块：如下，在多线程环境下，synchronized块中的方法获取了lock实例的monitor，如果实例相同，那么只有一个线程能执行该块内容

[复制代码](javascript:void(0);)

public class Thread1 implements Runnable {

Object lock;

public void run() {

synchronized(lock){

..do something

}

}

}

[复制代码](javascript:void(0);)

* + 直接用于方法： 相当于上面代码中用lock来锁定的效果，实际获取的是Thread1类的monitor。更进一步，如果修饰的是static方法，则锁定该类所有实例。
  + public class Thread1 implements Runnable {
  + public synchronized void run() {
  + ..do something
  + }

}

* synchronized, wait, notify结合:典型场景生产者消费者问题

[复制代码](javascript:void(0);)

/\*\*

\* 生产者生产出来的产品交给店员

\*/

public synchronized void produce()

{

if(this.product >= MAX\_PRODUCT)

{

try

{

wait();

System.out.println("产品已满,请稍候再生产");

}

catch(InterruptedException e)

{

e.printStackTrace();

}

return;

}

this.product++;

System.out.println("生产者生产第" + this.product + "个产品.");

notifyAll(); //通知等待区的消费者可以取出产品了

}

/\*\*

\* 消费者从店员取产品

\*/

public synchronized void consume()

{

if(this.product <= MIN\_PRODUCT)

{

try

{

wait();

System.out.println("缺货,稍候再取");

}

catch (InterruptedException e)

{

e.printStackTrace();

}

return;

}

System.out.println("消费者取走了第" + this.product + "个产品.");

this.product--;

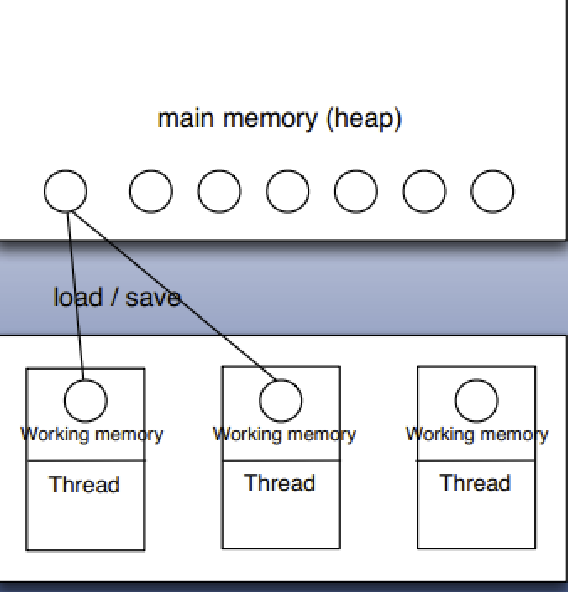
notifyAll(); //通知等待去的生产者可以生产产品了

}

[复制代码](javascript:void(0);)

**volatile**

多线程的内存模型：main memory（主存）、working memory（线程栈），在处理数据时，线程会把值从主存load到本地栈，完成操作后再save回去(volatile关键词的作用：每次针对该变量的操作都激发一次load and save)。



volatile

针对多线程使用的变量如果不是volatile或者final修饰的，很有可能产生不可预知的结果（另一个线程修改了这个值，但是之后在某线程看到的是修改之前的值）。其实道理上讲同一实例的同一属性本身只有一个副本。但是多线程是会缓存值的，本质上，volatile就是不去缓存，直接取值。在线程安全的情况下加volatile会牺牲性能。

**太祖长拳：基本线程类**

基本线程类指的是Thread类，Runnable接口，Callable接口  
Thread 类实现了Runnable接口，启动一个线程的方法：

　MyThread my = new MyThread();

　　my.start();

**Thread类相关方法：**

[复制代码](javascript:void(0);)

//当前线程可转让cpu控制权，让别的就绪状态线程运行（切换）

public static Thread.yield()

//暂停一段时间

public static Thread.sleep()

//在一个线程中调用other.join(),将等待other执行完后才继续本线程。

public join()

//后两个函数皆可以被打断

public interrupte()

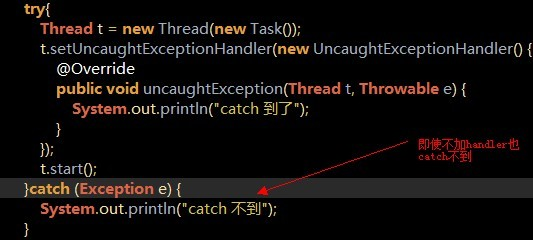
[复制代码](javascript:void(0);)

**关于中断**：它并不像stop方法那样会中断一个正在运行的线程。线程会不时地检测中断标识位，以判断线程是否应该被中断（中断标识值是否为true）。终端只会影响到wait状态、sleep状态和join状态。被打断的线程会抛出InterruptedException。  
Thread.interrupted()检查当前线程是否发生中断，返回boolean  
synchronized在获锁的过程中是不能被中断的。

中断是一个状态！interrupt()方法只是将这个状态置为true而已。所以说正常运行的程序不去检测状态，就不会终止，而wait等阻塞方法会去检查并抛出异常。如果在正常运行的程序中添加while(!Thread.interrupted()) ，则同样可以在中断后离开代码体

**Thread类最佳实践**：  
写的时候最好要设置线程名称 Thread.name，并设置线程组 ThreadGroup，目的是方便管理。在出现问题的时候，打印线程栈 (jstack -pid) 一眼就可以看出是哪个线程出的问题，这个线程是干什么的。

**如何获取线程中的异常**



不能用try,catch来获取线程中的异常

**Runnable**

与Thread类似

**Callable**

future模式：并发模式的一种，可以有两种形式，即无阻塞和阻塞，分别是isDone和get。其中Future对象用来存放该线程的返回值以及状态

ExecutorService e = Executors.newFixedThreadPool(3);

//submit方法有多重参数版本，及支持callable也能够支持runnable接口类型.

Future future = e.submit(new myCallable());

future.isDone() //return true,false 无阻塞

future.get() // return 返回值，阻塞直到该线程运行结束

**九阴真经：高级多线程控制类**

以上都属于内功心法，接下来是实际项目中常用到的工具了，Java1.5提供了一个非常高效实用的多线程包:*java.util.concurrent*, 提供了大量高级工具,可以帮助开发者编写高效、易维护、结构清晰的Java多线程程序。

**1.ThreadLocal类**

用处：保存线程的独立变量。对一个线程类（继承自Thread)  
当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。常用于用户登录控制，如记录session信息。

实现：每个Thread都持有一个TreadLocalMap类型的变量（该类是一个轻量级的Map，功能与map一样，区别是桶里放的是entry而不是entry的链表。功能还是一个map。）以本身为key，以目标为value。  
主要方法是get()和set(T a)，set之后在map里维护一个threadLocal -> a，get时将a返回。ThreadLocal是一个特殊的容器。

**2.原子类（AtomicInteger、AtomicBoolean……）**

如果使用atomic wrapper class如atomicInteger，或者使用自己保证原子的操作，则等同于synchronized

//返回值为boolean

AtomicInteger.compareAndSet(int expect,int update)

该方法可用于实现乐观锁，考虑文中最初提到的如下场景：a给b付款10元，a扣了10元，b要加10元。此时c给b2元，但是b的加十元代码约为：

if(b.value.compareAndSet(old, value)){

return ;

}else{

//try again

// if that fails, rollback and log

}

**AtomicReference**  
对于AtomicReference 来讲，也许对象会出现，属性丢失的情况，即oldObject == current，但是oldObject.getPropertyA != current.getPropertyA。  
这时候，AtomicStampedReference就派上用场了。这也是一个很常用的思路，即加上版本号

**3.Lock类**

lock: 在java.util.concurrent包内。共有三个实现：

ReentrantLock

ReentrantReadWriteLock.ReadLock

ReentrantReadWriteLock.WriteLock

主要目的是和synchronized一样， 两者都是为了解决同步问题，处理资源争端而产生的技术。功能类似但有一些区别。

区别如下：

lock更灵活，可以自由定义多把锁的枷锁解锁顺序（synchronized要按照先加的后解顺序）

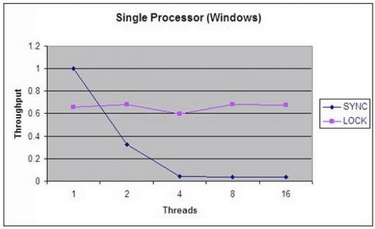
提供多种加锁方案，lock 阻塞式, trylock 无阻塞式, lockInterruptily 可打断式， 还有trylock的带超时时间版本。

本质上和监视器锁（即synchronized是一样的）

能力越大，责任越大，必须控制好加锁和解锁，否则会导致灾难。

和Condition类的结合。

性能更高，对比如下图：



synchronized和Lock性能对比

**ReentrantLock**　　　　  
可重入的意义在于持有锁的线程可以继续持有，并且要释放对等的次数后才真正释放该锁。  
使用方法是：

1.先new一个实例

static ReentrantLock r=new ReentrantLock();

2.加锁

r.lock()或r.lockInterruptibly();

此处也是个不同，后者可被打断。当a线程lock后，b线程阻塞，此时如果是lockInterruptibly，那么在调用b.interrupt()之后，b线程退出阻塞，并放弃对资源的争抢，进入catch块。（如果使用后者，必须throw interruptable exception 或catch）

3.释放锁

r.unlock()

必须做！何为必须做呢，要放在finally里面。以防止异常跳出了正常流程，导致灾难。这里补充一个小知识点，finally是可以信任的：经过测试，哪怕是发生了OutofMemoryError，finally块中的语句执行也能够得到保证。

**ReentrantReadWriteLock**

可重入读写锁（读写锁的一个实现）

　ReentrantReadWriteLock lock = new ReentrantReadWriteLock()

　　ReadLock r = lock.readLock();

　　WriteLock w = lock.writeLock();

两者都有lock,unlock方法。写写，写读互斥；读读不互斥。可以实现并发读的高效线程安全代码

**4.容器类**

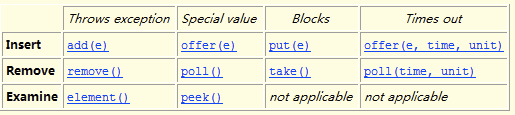
这里就讨论比较常用的两个：

BlockingQueue

ConcurrentHashMap

**BlockingQueue**  
阻塞队列。该类是java.util.concurrent包下的重要类，通过对Queue的学习可以得知，这个queue是单向队列，可以在队列头添加元素和在队尾删除或取出元素。类似于一个管　　道，特别适用于先进先出策略的一些应用场景。普通的queue接口主要实现有PriorityQueue（优先队列），有兴趣可以研究

BlockingQueue在队列的基础上添加了多线程协作的功能：



BlockingQueue

除了传统的queue功能（表格左边的两列）之外，还提供了阻塞接口put和take，带超时功能的阻塞接口offer和poll。put会在队列满的时候阻塞，直到有空间时被唤醒；take在队　列空的时候阻塞，直到有东西拿的时候才被唤醒。用于生产者-消费者模型尤其好用，堪称神器。

常见的阻塞队列有：

ArrayListBlockingQueue

LinkedListBlockingQueue

DelayQueue

SynchronousQueue

**ConcurrentHashMap**  
高效的线程安全哈希map。请对比hashTable , concurrentHashMap, HashMap

**5.管理类**

管理类的概念比较泛，用于管理线程，本身不是多线程的，但提供了一些机制来利用上述的工具做一些封装。  
了解到的值得一提的管理类：ThreadPoolExecutor和 JMX框架下的系统级管理类 ThreadMXBean  
**ThreadPoolExecutor**  
如果不了解这个类，应该了解前面提到的ExecutorService，开一个自己的线程池非常方便：

[复制代码](javascript:void(0);)

ExecutorService e = Executors.newCachedThreadPool();

ExecutorService e = Executors.newSingleThreadExecutor();

ExecutorService e = Executors.newFixedThreadPool(3);

// 第一种是可变大小线程池，按照任务数来分配线程，

// 第二种是单线程池，相当于FixedThreadPool(1)

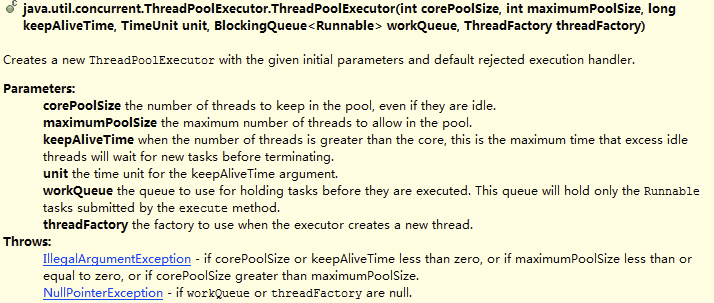
// 第三种是固定大小线程池。

// 然后运行

e.execute(new MyRunnableImpl());

[复制代码](javascript:void(0);)

该类内部是通过ThreadPoolExecutor实现的，掌握该类有助于理解线程池的管理，本质上，他们都是ThreadPoolExecutor类的各种实现版本。请参见javadoc：



ThreadPoolExecutor参数解释

翻译一下：

[复制代码](javascript:void(0);)

corePoolSize:池内线程初始值与最小值，就算是空闲状态，也会保持该数量线程。

maximumPoolSize:线程最大值，线程的增长始终不会超过该值。

keepAliveTime：当池内线程数高于corePoolSize时，经过多少时间多余的空闲线程才会被回收。回收前处于wait状态

unit：

时间单位，可以使用TimeUnit的实例，如TimeUnit.MILLISECONDS

workQueue:待入任务（Runnable）的等待场所，该参数主要影响调度策略，如公平与否，是否产生饿死(starving)

threadFactory:线程工厂类，有默认实现，如果有自定义的需要则需要自己实现ThreadFactory接口并作为参数传入。

[复制代码](javascript:void(0);)