# 创建线程的两种方式

Java中实现多线程有两种方法：继承Thread类、实现Runnable接口。

## 继承Thread

定义一个Thread的子类并重写其run方法，如：

**public** **class** TestThread **extends** Thread{

**public** **void** run(){

**for** (**int** i=0;i<10;i++) {

System.*out*.println("i="+i);

}

}

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

TestThread myThread = **new** TestThread();

myThread.start();//启动线程

//myThread.run(); //方法调用

**for** (**int** i = 0; i < 10; i++) {

System.*out*.println("main i="+i);

}

}

}

myThread.start()与myThread.run()都是启动线程？

## 实现Runnable接口

定义线程类继承Runnable接口，如：

**public** **class** TestRunnable **implements** Runnable{

@Override

**public** **void** run() {

**for** (**int** i=0;i<10;i++) {

System.*out*.println("i="+i);

}

}

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

Thread myThread = **new** Thread(**new** TestRunnable());

myThread.start();//启动线程

//myThread.run(); //方法调用

**for** (**int** i = 0; i < 10; i++) {

System.*out*.println("main i="+i);

}

}

}

## Runnable优势

1. 可以避免由于Java的单继承特性而带来的局限

2. 能够被多个线程共享

3. 适合多线程处理同一资源的情况

# 线程睡眠

## 线程睡眠

线程睡眠指让正在执行的线程暂停一段时间，并进入阻塞状态。

使线程睡眠，可使用Thread的sleep方法。如：

public class TestSleep implements Runnable {

@Override

public void run() {

for (int i=0;i<10;i++) {

System.out.println("i="+i);

}

}

public static void main(String[] args) {

TestSleep testSleep = new TestSleep();

Thread myThread = new Thread(testSleep);

myThread.start();

try {

//myThread.sleep(1000);

Thread.sleep(1000);

System.out.println("main over=========");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

myThread.sleep(1000) 是让myThread线程睡眠？

Thread.sleep(1000) 是让当前线程睡眠？

## 注意事项

1. sleep是静态方法，最好不要用Thread的实例对象调用它，因为它睡眠的始终是当前正在运行的线程，而不是调用它的线程对象，它只对正在运行状态的线程对象有效。
2. sleep 1秒，线程1秒后会立即进入运行吗？

# 线程中断

当一个线程运行时，另一个线程可以调用对应的Thread对象的interrupt（）方法来中断它，该方法只是在目标线程中设置一个标志，表示它已经被中断，并立即返回

## Interrupt方法的使用

interrupt()方法用于中断线程，如下:

public class TestInterrupt implements Runnable {

@Override

public void run() {

int i=0;

/\*boolean flag = Thread.currentThread().isInterrupted();

while(!flag){

System.out.println(i++);

}\*/

while(true){

System.out.println(i++);

}

}

public static void main(String[] args) {

TestInterrupt testInterrupt = new TestInterrupt();

Thread myThread = new Thread(testInterrupt);

myThread.start();

myThread.interrupt();

}

}

运行程序发现，线程并没有被中断。其实interrupt()方法并不是中断线程的执行，而是为调用该方法的线程对象打上一个标记，设置其中断状态为true，通过isInterrupted()方法可以得到这个线程状态。

## 休眠线程的中断

代码如下：

public class SleepInterrupt extends Object implements Runnable{

public void run(){

try{

System.out.println("in run() - sleeping start ");

Thread.sleep(20000);

System.out.println("in run() - sleep over ");

}catch(InterruptedException e){

System.out.println("in run() - interrupted while sleeping");

//处理完中断异常后，返回到run（）方法人口，

//如果没有return，线程不会实际被中断，它会继续打印下面的信息

return;

}

System.out.println("in run() - leaving normally");

}

public static void main(String[] args) {

SleepInterrupt si = new SleepInterrupt();

Thread t = new Thread(si);

t.start();

//主线程休眠2秒，从而确保刚才启动的线程有机会执行一段时间

try {

Thread.sleep(2000);

}catch(InterruptedException e){

e.printStackTrace();

}

System.out.println("in main() - interrupting other thread");

//中断线程t

t.interrupt();

System.out.println("in main() - leaving");

}

}

休眠线程被打断，会抛出InterruptedException异常，线程stop()方式中断线程，无异常抛出

## 待决中断

如果线程在调用sleep（）方法前被中断，那么该中断称为待决中断，它会在刚调用sleep（）方法时，立即抛出InterruptedException异常，如下:

public class PendingInterrupt implements Runnable{

public void run(){

try{

System.out.println("in run() - sleeping start ");

Thread.sleep(20000);

System.out.println("in run() - sleep over ");

}catch(InterruptedException e){

System.out.println("in run() - interrupted while sleeping");

//处理完中断异常后，返回到run（）方法人口，

//如果没有return，线程不会实际被中断，它会继续打印下面的信息

return;

}

System.out.println("in run() - leaving normally");

}

public static void main(String[] args) {

SleepInterrupt si = new SleepInterrupt();

Thread t = new Thread(si);

t.start();

//中断线程t

t.interrupt();

}

}

## 判断线程中断状态

IsInterrupted：

可以在Thread对象上调用isInterrupted（）方法来检查任何线程的中断状态。这里需要注意：线程一旦被中断，isInterrupted（）方法便会返回true，而一旦sleep（）方法抛出异常，它将清空中断标志，此时isInterrupted（）方法将返回false

Interrupted：

可以使用Thread.interrupted（）方法来检查当前线程的中断状态（并隐式重置为false）。又由于它是静态方法，因此不能在特定的线程上使用，而只能报告调用它的线程的中断状态，如果线程被中断，而且中断状态尚不清楚，那么，这个方法返回true。与isInterrupted（）不同，它将自动重置中断状态为false，第二次调用Thread.interrupted（）方法，总是返回false，除非中断了线程

# Join

合并某个线程，按顺序执行

public class TestJoin implements Runnable{

@Override

public void run() {

for (int i=0;i<10;i++) {

System.out.println("i="+i);

}

}

public static void main(String[] args) {

TestJoin testJoin = new TestJoin();

Thread myThread = new Thread(testJoin);

myThread.start();

try {

myThread.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

for (int i=0;i<10;i++) {

System.out.println("main i="+i);

}

}

}

# Yield

让出线程，进入线程就绪状态，等待cpu分配资源执行

public class TestYield implements Runnable{

@Override

public void run() {

for (int i=0;i<1000;i++) {

/\*if(i%10==0){

Thread.currentThread().yield();

}\*/

System.out.println("TestYield 1="+i);

}

}

public static void main(String[] args) {

TestYield testYield = new TestYield();

Thread myThread1 = new Thread(testYield);

//myThread1.setPriority(Thread.NORM\_PRIORITY+3);

TestYield2 testYield2 = new TestYield2();

Thread myThread2 = new Thread(testYield2);

//myThread2.setPriority(Thread.NORM\_PRIORITY-3);

myThread1.start();

myThread2.start();

}

}

class TestYield2 implements Runnable{

@Override

public void run() {

for (int i=0;i<1000;i++) {

System.out.println("TestYield 2="+i);

}

}

}

# Synchronized

在并编程中，当多个线程同时访问对象并要求操作相同资源时，分 割了原子操作就有可能出现数据的不一致或数据不完整的情况，为避免这种情况的发生，我们会采取同步机制，以确保在某一时刻，方法内只允许有一个线程。

采用synchronized修饰符实现的同步机制叫做互斥锁机制，它所获得的锁叫做互斥锁。每个对象都有一个monitor(锁标记)，当线程拥有这个锁标记时才能访问这个资源，没有锁标记便进入锁池。任何一个对象系统都会为其创建一个互斥锁，这个锁是为了分配给线程的，防止打断原子操作。每个对象的锁只能分配给一个线程，因此叫做互斥锁。

## 互斥锁

public class TestSynchronized {

public synchronized void m1(){

for (int i=0;i<50;i++) {

System.out.println("m1="+i);

}

}

public void m2(){

for (int i=0;i<50;i++) {

System.out.println("m2="+i);

}

}

//public static synchronized void m3()

public synchronized void m3(){

for (int i=0;i<50;i++) {

System.out.println("m3="+i);

}

}

public static void main(String[] args) {

TestSynchronized testSynchronized = new TestSynchronized();

Thread1 thread = new Thread1(testSynchronized);

thread.start();//m1

testSynchronized.m2();

testSynchronized.m3();

}

}

class Thread1 extends Thread{

private TestSynchronized testSynchronized;

public Thread1(TestSynchronized testSynchronized){

this.testSynchronized=testSynchronized;

}

public void run(){

testSynchronized.m1();

}

}

1. 如果同一个方法内同时有两个或更多线程，则每个线程有自己的局部变量拷贝
2. 类的每个实例都有自己的对象级别锁。当一个线程访问实例对象中的synchronized同步代码块或同步方法时，该线程便获取了该实例的对象级别锁，其他线程这时如果要访问synchronized同步代码块或同步方法，便需要阻塞等待，直到前面的线程从同步代码块或方法中退出，释放掉了该对象级别锁。
3. 访问同一个类的不同实例对象中的同步代码块，不存在阻塞等待获取对象锁的问题，因为它们获取的是各自实例的对象级别锁，相互之间没有影响。
4. 持有一个对象级别锁不会阻止该线程被交换出来，也不会阻塞其他线程访问同一示例对象中的非synchronized代码。当一个线程A持有一个对象级别锁（即进入了synchronized修饰的代码块或方法中）时，线程也有可能被交换出去，此时线程B有可能获取执行该对象中代码的时间，但它只能执行非同步代码（没有用synchronized修饰），当执行到同步代码时，便会被阻塞，此时可能线程规划器又让A线程运行，A线程继续持有对象级别锁，当A线程退出同步代码时（即释放了对象级别锁），如果B线程此时再运行，便会获得该对象级别锁，从而执行synchronized中的代码。
5. 持有对象级别锁的线程会让其他线程阻塞在所有的synchronized代码外。
6. 类级别锁被特定类的所有示例共享，它用于控制对static成员变量以及static方法的并发访问。
7. 互斥是实现同步的一种手段，临界区、互斥量和信号量都是主要的互斥实现方式。synchronized关键字经过编译后，会在同步块的前后分别形成monitorenter和monitorexit这两个字节码指令。根据虚拟机规范的要求，在执行monitorenter指令时，首先要尝试获取对象的锁，如果获得了锁，把锁的计数器加1，相应地，在执行monitorexit指令时会将锁计数器减1，当计数器为0时，锁便被释放了。由于synchronized同步块对同一个线程是可重入的，因此一个线程可以多次获得同一个对象的互斥锁，同样，要释放相应次数的该互斥锁，才能最终释放掉该锁。

## 内存可见性

加锁（synchronized同步）的功能不仅仅局限于互斥行为，同时还存在另外一个重要的方面：内存可见性

线程的同步实现了 当一个线程修改了对象状态后，其他线程能够看到该变化。

# Volatile

Java 语言中的 volatile 变量可以被看作是一种 “程度较轻的 synchronized”；与 synchronized 块相比，volatile 变量所需的编码较少，并且运行时开销也较少，但是它所能实现的功能也仅是 synchronized 的一部分

Volatile 变量具有 synchronized 的可见性特性，但是不具备原子特性。这就是说线程能够自动发现 volatile 变量的最新值。Volatile 变量可用于提供线程安全，但是只能应用于非常有限的一组用例：多个变量之间或者某个变量的当前值与修改后值之间没有约束。因此，单独使用 volatile 还不足以实现计数器、互斥锁或任何具有与多个变量相关的不变式（Invariants）的类（例如 “start <=end”）。

出于简易性或可伸缩性的考虑，您可能倾向于使用 volatile 变量而不是锁。当使用 volatile 变量而非锁时，某些习惯用法（idiom）更加易于编码和阅读。此外，volatile 变量不会像锁那样造成线程阻塞，因此也很少造成可伸缩性问题。在某些情况下，如果读操作远远大于写操作，volatile 变量还可以提供优于锁的性能优势。

## 正确使用volatile变量的条件

只能在有限的一些情形下使用 volatile 变量替代锁。要使 volatile 变量提供理想的线程安全，必须同时满足下面两个条件：

1. 对变量的写操作不依赖于当前值。

例如 volatile 变量不能用作线程安全计数器。虽然增量操作（x++）看上去类似一个单独操作，实际上它是一个由读取－修改－写入操作序列组成的组合操作，必须以原子方式执行，而 volatile 不能提供必须的原子特性。实现正确的操作需要使 x 的值在操作期间保持不变，而 volatile 变量无法实现这点。

1. 该变量没有包含在具有其他变量的不变式中。

非线程安全的数值范围类。它包含了一个不变式 —— 下界总是小于或等于上界

public class NumberRange {

private int lower, upper;

public int getLower() { return lower; }

public int getUpper() { return upper; }

public void setLower(int value) {

if (value > upper)

throw new IllegalArgumentException(...);

lower = value;

}

public void setUpper(int value) {

if (value < lower)

throw new IllegalArgumentException(...);

upper = value;

}

}

这种方式限制了范围的状态变量，因此将 lower 和 upper 字段定义为 volatile 类型不能够充分实现类的线程安全；从而仍然需要使用同步。否则，如果凑巧两个线程在同一时间使用不一致的值执行 setLower 和 setUpper 的话，则会使范围处于不一致的状态

## 正确使用volatile的模式

* 1. 状态标志

用于指示发生了一个重要的一次性事件，例如完成初始化或请求停机

volatile boolean shutdownRequested;

public void shutdown() { shutdownRequested = true; }

public void doWork() {

while (!shutdownRequested) {

// do stuff

}

}

* 1. **一次性安全发布**

public class BackgroundFloobleLoader {

public volatile Flooble theFlooble;

public void initInBackground() {

// do lots of stuff

theFlooble = new Flooble(); // this is the only write to theFlooble

}

}

public class SomeOtherClass {

public void doWork() {

while (true) {

// do some stuff...

// use the Flooble, but only if it is ready

if (floobleLoader.theFlooble != null)

doSomething(floobleLoader.theFlooble);

}

}

}

* 1. **独立观察**

定期 “发布” 观察结果供程序内部使用。例如，假设有一种环境传感器能够感觉环境温度。一个后台线程可能会每隔几秒读取一次该传感器，并更新包含当前文档的 volatile 变量。然后，其他线程可以读取这个变量，从而随时能够看到最新的温度值。

以下代码 展示了身份验证机制如何记忆最近一次登录的用户的名字。将反复使用 lastUser 引用来发布值，以供程序的其他部分使用。

public class UserManager {

public volatile String lastUser;

public boolean authenticate(String user, String password) {

boolean valid = passwordIsValid(user, password);

if (valid) {

User u = new User();

activeUsers.add(u);

lastUser = user;

}

return valid;

}

}

* 1. **volatile bean**

在 volatile bean 模式中，JavaBean 的所有数据成员都是 volatile 类型的，并且 getter 和 setter 方法必须非常普通 —— 除了获取或设置相应的属性外，不能包含任何逻辑。此外，对于对象引用的数据成员，引用的对象必须是有效不可变的。（这将禁止具有数组值的属性，因为当数组引用被声明为 volatile 时，只有引用而不是数组本身具有 volatile 语义）。对于任何 volatile 变量，不变式或约束都不能包含 JavaBean 属性

public class Person {

private volatile String firstName;

private volatile String lastName;

private volatile int age;

public String getFirstName() { return firstName; }

public String getLastName() { return lastName; }

public int getAge() { return age; }

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public void setAge(int age) {

this.age = age;

}

}

# 实现内存可见性两种方式的比较

1. volatile变量是一种稍弱的同步机制在访问volatile变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此volatile变量是一种比synchronized关键字更轻量级的同步机制。
2. 从内存可见性的角度看，写入volatile变量相当于退出同步代码块，而读取volatile变量相当于进入同步代码块。
3. 在代码中如果过度依赖volatile变量来控制状态的可见性，通常会比使用锁的代码更脆弱，也更难以理解。仅当volatile变量能简化代码的实现以及对同步策略的验证时，才应该使用它。一般来说，用同步机制会更安全些。
4. 加锁机制（即同步机制）既可以确保可见性又可以确保原子性，而volatile变量只能确保可见性，原因是声明为volatile的简单变量如果当前值与该变量以前的值相关，那么volatile关键字不起作用，也就是说如下的表达式都不是原子操作：“count++”、“count = count+1”。

# 线程间通信

在Java中，可以通过配合调用Object对象的wait（）方法和notify（）方法或notifyAll（）方法来实现线程间的通信。在线程中调用wait（）方法，将阻塞等待其他线程的通知（其他线程调用notify（）方法或notifyAll（）方法），在线程中调用notify（）方法或notifyAll（）方法，将通知其他线程从wait（）方法处返回。

      Object是所有类的超类，它有5个方法组成了等待/通知机制的核心：notify（）、notifyAll（）、wait（）、wait（long）和wait（long，int）。在Java中，所有的类都从Object继承而来，因此，所有的类都拥有这些共有方法可供使用。而且，由于他们都被声明为final，因此在子类中不能覆写任何一个方法

## wait（）

该方法用来将当前线程置入休眠状态，直到接到通知或被中断为止。在调用wait（）之前，线程必须要获得该对象的对象级别锁，即只能在同步方法或同步块中调用wait（）方法。进入wait（）方法后，当前线程释放锁。在从wait（）返回前，线程与其他线程竞争重新获得锁

## notify()

该方法用来通知那些可能等待该对象的对象锁的其他线程。如果有多个线程等待，则线程规划器任意挑选出其中一个wait（）状态的线程来发出通知，并使它等待获取该对象的对象锁（notify后，当前线程不会马上释放该对象锁，wait所在的线程并不能马上获取该对象锁，要等到程序退出synchronized代码块后，当前线程才会释放锁，wait所在的线程也才可以获取该对象锁）

## notifyAll（）

notifyAll使所有原来在该对象上wait的线程统统退出wait的状态

## wait（long）和wait（long,int）

 显然，这两个方法是设置等待超时时间的, 如果在等待线程接到通知或被中断之前，已经超过了指定的毫秒数，则它通过竞争重新获得锁，并从wait（long）返回.

样例代码:

public class TestWait {

public static void main(String[] args) {

Object obj = new Object();

TestWait1 wait1 = new TestWait1(obj);

TestWait2 wait2 = new TestWait2(obj);

new Thread(wait1).start();

/\*try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}\*/

new Thread(wait2).start();

}

}

class TestWait1 implements Runnable{

private Object obj;

public TestWait1 (Object obj){

this.obj=obj;

}

@Override

public void run() {

synchronized (obj) {

System.out.println("TestWait1 wait pre ");

try {

obj.wait();

System.out.println("TestWait1 wait ofter");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

class TestWait2 implements Runnable{

private Object obj;

public TestWait2 (Object obj){

this.obj=obj;

}

@Override

public void run() {

synchronized (obj) {

System.out.println("TestWait2 notify pre");

obj.notify();

System.out.println("TestWait2 notify ofter");

}

}

}

## notify通知的遗漏

notify通知的遗漏很容易理解，即threadA还没开始wait的时候，threadB已经notify了，这样，threadB通知是没有任何响应的，当threadB退出synchronized代码块后，threadA再开始wait，便会一直阻塞等待，直到被别的线程打断

总结：在使用线程的等待/通知机制时，一般都要配合一个boolean变量值（或者其他能够判断真假的条件），在notify之前改变该boolean变量的值，让wait返回后能够退出while循环（一般都要在wait方法外围加一层while循环，以防止早期通知），或在通知被遗漏后，不会被阻塞在wait方法处。这样便保证了程序的正确性。

# [并发新特性—Executor框架与线程池](http://blog.csdn.net/ns_code/article/details/17465497)

在Java 5之后，并发编程引入了一堆新的启动、调度和管理线程的API。Executor框架便是Java 5中引入的，其内部使用了线程池机制，它在java.util.cocurrent 包下，通过该框架来控制线程的启动、执行和关闭，可以简化并发编程的操作。因此，在Java 5之后，通过Executor来启动线程比使用Thread的start方法更好，更易管理，效率更好（用线程池实现，节约开销）.

Executor框架包括： Executor，Executors，ExecutorService，CompletionService，Future，Callable等。

Executor接口中之定义了一个方法execute（Runnable command），该方法接收一个Runable实例，用来执行一个任务。ExecutorService接口继承自Executor接口，它提供了更丰富的实现多线程的方法，比如，ExecutorService提供了关闭自己的方法，以及可为跟踪一个或多个异步任务执行状况而生成 Future 的方法。 可以调用ExecutorService的shutdown（）方法来平滑地关闭 ExecutorService，调用该方法后，将导致ExecutorService停止接受任何新的任务且等待已经提交的任务执行完成,当所有已经提交的任务执行完毕后将会关闭ExecutorService。因此我们一般用该接口来实现和管理多线程。

ExecutorService的生命周期包括三种状态：运行、关闭、终止。创建后便进入运行状态，当调用了shutdown（）方法时，便进入关闭状态，此时意味着ExecutorService不再接受新的任务，但它还在执行已经提交了的任务，当所有已经提交了的任务执行完后，便到达终止状态。如果不调用shutdown（）方法，ExecutorService会一直处在运行状态，不断接收新的任务，执行新的任务，服务器端一般不需要关闭它，保持一直运行即可。

Executors提供了一系列工厂方法用于创建线程池，返回的线程池都实现了ExecutorService接口。

public static ExecutorService newFixedThreadPool(int nThreads)

    创建固定数目线程的线程池。

    public static ExecutorService newCachedThreadPool()

    创建一个可缓存的线程池，调用execute将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。能重用的线程，必须是timeout IDLE内的池中线程，缺省timeout是60s,超过这个IDLE时长，线程实例将被终止及移出池。通常用于执行一些生存期很短的异步型任务。

    public static ExecutorService newSingleThreadExecutor()

    创建一个单线程化的Executor。

    public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)

    创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代Timer类。

## Executor执行Runnable任务

public class TestCachedThreadPool{

public static void main(String[] args){

ExecutorService executorService = Executors.newCachedThreadPool();

// ExecutorService executorService = Executors.newFixedThreadPool(5);

// ExecutorService executorService = Executors.newSingleThreadExecutor();

for (int i = 1; i < 6; i++){

executorService.execute(new TestRunnable());

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\* a" + i + " \*\*\*\*\*\*\*\*\*\*\*\*\*");

}

executorService.shutdown();

}

}

class TestRunnable implements Runnable{

public void run(){

System.out.println(Thread.currentThread().getName() + "线程被调用了。");

}

}

## Executor执行Callable任务

在Java 5之后，任务分两类：一类是实现了Runnable接口的类，一类是实现了Callable接口的类。两者都可以被ExecutorService执行，但是Runnable任务没有返回值，而Callable任务有返回值。并且Callable的call()方法只能通过ExecutorService的submit(Callable<T> task) 方法来执行，并且返回一个 <T>Future<T>，是表示任务等待完成的 Future

public class TestCallable{

public static void main(String[] args){

ExecutorService executorService = Executors.newCachedThreadPool();

List<Future<String>> resultList = new ArrayList<Future<String>>();

//创建10个任务并执行

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

//使用ExecutorService执行Callable类型的任务，并将结果保存在future变量中

Future<String> future = executorService.submit(new TaskWithResult(i));

//将任务执行结果存储到List中

resultList.add(future);

}

//遍历任务的结果

for (Future<String> fs : resultList){

try{

//while(!fs.isDone());//Future返回如果没有完成，则一直循环等待，直到Future返回完成

System.out.println(fs.get()); //打印各个线程（任务）执行的结果

}catch(InterruptedException e){

e.printStackTrace();

}catch(ExecutionException e){

e.printStackTrace();

}finally{

//启动一次顺序关闭，执行以前提交的任务，但不接受新任务

executorService.shutdown();

}

}

}

}

class TaskWithResult implements Callable<String>{

private int id;

public TaskWithResult(int id){

this.id = id;

}

/\*\*

\* 任务的具体过程，一旦任务传给ExecutorService的submit方法，

\* 则该方法自动在一个线程上执行

\*/

public String call() throws Exception {

System.out.println("call()方法被自动调用！！！ " + Thread.currentThread().getName());

//该返回结果将被Future的get方法得到

return "call()方法被自动调用，任务返回的结果是：" + id + " " + Thread.currentThread().getName();

}

}

## 自定义线程池

public class TestCustomThreadPool{

public static void main(String[] args){

//创建等待队列

BlockingQueue<Runnable> bqueue = new ArrayBlockingQueue<Runnable>(20);

//创建线程池，池中保存的线程数为3，允许的最大线程数为5

ThreadPoolExecutor pool = new ThreadPoolExecutor(3,5,50,TimeUnit.MILLISECONDS,bqueue);

//创建任务

for (int i=1;i<=20;i++) {

Runnable t = new MyThread();

pool.execute(t);

}

//关闭线程池

pool.shutdown();

}

}

class MyThread implements Runnable{

@Override

public void run(){

System.out.println(Thread.currentThread().getName() + "正在执行。。。");

try{

Thread.sleep(100);

}catch(InterruptedException e){

e.printStackTrace();

}

}

}

**自定义线程池各参数含义**ThreadPoolExecutor **(int corePoolSize, int maximumPoolSize, long         keepAliveTime,**[**TimeUnit**](file:///D:\adt-bundle-windows-x86\sdk\docs\reference\java\util\concurrent\TimeUnit.html)**unit,**[**BlockingQueue**](file:///D:\adt-bundle-windows-x86\sdk\docs\reference\java\util\concurrent\BlockingQueue.html)**<**[**Runnable**](file:///D:\adt-bundle-windows-x86\sdk\docs\reference\java\lang\Runnable.html)**> workQueue)**

corePoolSize：线程池中所保存的线程数。

maximumPoolSize：池中允许的最大线程数。

keepAliveTime：线程池中的空闲线程所能持续的最长时间。

unit：持续时间的单位。

workQueue：任务执行前保存任务的队列，仅保存由execute方法提交的Runnable任务。

当试图通过excute方法将一个Runnable任务添加到线程池中时，按照如下顺序来处理：

    1、如果线程池中的线程数量少于corePoolSize，即使线程池中有空闲线程，也会创建一个新的线程来执行新添加的任务；

    2、如果线程池中的线程数量大于等于corePoolSize，但缓冲队列workQueue未满，则将新添加的任务放到workQueue中，按照FIFO的原则依次等待执行（线程池中有线程空闲出来后依次将缓冲队列中的任务交付给空闲的线程执行）；

    3、如果线程池中的线程数量大于等于corePoolSize，且缓冲队列workQueue已满，但线程池中的线程数量小于maximumPoolSize，则会创建新的线程来处理被添加的任务；

4、如果线程池中的线程数量等于了maximumPoolSize，有4种才处理方式

**RejectedExecutionHandler的四种拒绝策略**

ThreadPoolExecutor.AbortPolicy当线程池中的数量等于最大线程数时抛出java.util.concurrent.RejectedExecutionException异常. 涉及到该异常的任务也不会被执行.

ThreadPoolExecutor.CallerRunsPolicy()当线程池中的数量等于最大线程数时,重试添加当前的任务;它会自动重复调用execute()方法.

ThreadPoolExecutor.DiscardOldestPolicy()当线程池中的数量等于最大线程数时,抛弃线程池中工作队列头部的任务(即等待时间最久Oldest的任务),并执行新传入的任务

ThreadPoolExecutor.DiscardPolicy()当线程池中的数量等于最大线程数时,丢弃不能执行的新加任务

**下面说说几种排队的策略：**

    1、直接提交。缓冲队列采用 SynchronousQueue，它将任务直接交给线程处理而不保持它们。如果不存在可用于立即运行任务的线程（即线程池中的线程都在工作），则试图把任务加入缓冲队列将会失败，因此会构造一个新的线程来处理新添加的任务，并将其加入到线程池中。直接提交通常要求无界 maximumPoolSizes（Integer.MAX\_VALUE） 以避免拒绝新提交的任务。newCachedThreadPool采用的便是这种策略。

    2、无界队列。使用无界队列（典型的便是采用预定义容量的 LinkedBlockingQueue，理论上是该缓冲队列可以对无限多的任务排队）将导致在所有 corePoolSize 线程都工作的情况下将新任务加入到缓冲队列中。这样，创建的线程就不会超过 corePoolSize，也因此，maximumPoolSize 的值也就无效了。当每个任务完全独立于其他任务，即任务执行互不影响时，适合于使用无界队列。newFixedThreadPool采用的便是这种策略。

    3、有界队列。当使用有限的 maximumPoolSizes 时，有界队列（一般缓冲队列使用ArrayBlockingQueue，并制定队列的最大长度）有助于防止资源耗尽，但是可能较难调整和控制，队列大小和最大池大小需要相互折衷，需要设定合理的参数。

# [并发新特性—Lock锁与条件变量](http://blog.csdn.net/ns_code/article/details/17487337)

Java 5中引入了新的锁机制——java.util.concurrent.locks中的显式的互斥锁：Lock接口，它提供了比synchronized更加广泛的锁定操作. Lock接口有3个实现它的类：ReentrantLock、ReetrantReadWriteLock.ReadLock和ReetrantReadWriteLock.WriteLock，即重入锁、读锁和写锁，为了可以使用更多的功能，一般用ReentrantLock为其实例化。为了保证锁最终一定会被释放（可能会有异常发生），要把互斥区放在try语句块内，并在finally语句块中释放锁，尤其当有return语句时，return语句必须放在try字句中，以确保unlock（）不会过早发生，从而将数据暴露给第二个任务。因此，采用lock加锁和释放锁的一般形式

如下:

Lock lock = new ReentrantLock();//默认使用非公平锁，如果要使用公平锁，需要传入参数true

lock.lock();

try {

//更新对象的状态

//捕获异常，必要时恢复到原来的不变约束

//如果有return语句，放在这里

} finally {

lock.unlock(); //锁必须在finally块中释放

}

## ReetrankLock与synchronized比较

### 性能比较

在JDK1.5中，synchronized是性能低效的。因为这是一个重量级操作，它对性能最大的影响是阻塞的是实现，挂起线程和恢复线程的操作都需要转入内核态中完成，这些操作给系统的并发性带来了很大的压力。相比之下使用Java提供的Lock对象，性能更高一些。Brian Goetz对这两种锁在JDK1.5、单核处理器及双Xeon处理器环境下做了一组吞吐量对比的实验，发现多线程环境下，synchronized的吞吐量下降的非常严重，而ReentrankLock则能基本保持在同一个比较稳定的水平上。但与其说ReetrantLock性能好，倒不如说synchronized还有非常大的优化余地，于是到了JDK1.6，发生了变化，对synchronize加入了很多优化措施，有自适应自旋，锁消除，锁粗化，轻量级锁，偏向锁等等。导致在JDK1.6上synchronize的性能并不比Lock差。官方也表示，他们也更支持synchronize，在未来的版本中还有优化余地，所以还是提倡在synchronized能实现需求的情况下，优先考虑使用synchronized来进行同步

**两种锁机制的底层的实现策略**

1. 互斥同步它属于一种悲观的并发策略，即线程获得的是独占锁。独占锁意味着其他线程只能依靠阻塞来等待线程释放锁。而在CPU转换线程阻塞时会引起线程上下文切换，当有很多线程竞争锁的时候，会引起CPU频繁的上下文切换导致效率很低。synchronized采用的便是这种并发策略。

2. 随着指令集的发展，我们有了另一种选择：基于冲突检测的乐观并发策略，通俗地讲就是先进性操作，如果没有其他线程争用共享数据，那操作就成功了，如果共享数据被争用，产生了冲突，那就再进行其他的补偿措施（最常见的补偿措施就是不断地重拾，直到试成功为止），这种乐观的并发策略的许多实现都不需要把线程挂起，因此这种同步被称为非阻塞同步。ReetrantLock采用的便是这种并发策略

悲观者与乐观者的做事方式完全不一样，悲观者的人生观是一件事情我必须要百分之百完全控制才会去做，否则就认为这件事情一定会出问题；而乐观者的人生观则相反，凡事不管最终结果如何，他都会先尝试去做，大不了最后不成功。这就是悲观锁与乐观锁的区别，悲观锁会把整个对象加锁占为自有后才去做操作，乐观锁不获取锁直接做操作，然后通过一定检测手段决定是否更新数据.

乐观锁的核心算法是CAS（Compareand Swap，比较并交换），它涉及到三个操作数：内存值、预期值、新值。当且仅当预期值和内存值相等时才将内存值修改为新值。这样处理的逻辑是，首先检查某块内存的值是否跟之前我读取时的一样，如不一样则表示期间此内存值已经被别的线程更改过，舍弃本次操作，否则说明期间没有其他线程对此内存值操作，可以把新值设置给此块内存。如图2-5-4-1，有两个线程可能会差不多同时对某内存操作，线程二先读取某内存值作为预期值，执行到某处时线程二决定将新值设置到内存块中，如果线程一在此期间修改了内存块，则通过CAS即可以检测出来，假如检测没问题则线程二将新值赋予内存块。



### 用途比较

基本语法上，ReentrantLock与synchronized很相似，它们都具备一样的线程重入特性，只是代码写法上有点区别而已，一个表现为API层面的互斥锁（Lock），一个表现为原生语法层面的互斥锁（synchronized）。ReentrantLock相对synchronized而言还是增加了一些高级功能，主要有以下三项：

    1、等待可中断：当持有锁的线程长期不释放锁时，正在等待的线程可以选择放弃等待，改为处理其他事情，它对处理执行时间非常长的同步块很有帮助。而在等待由synchronized产生的互斥锁时，会一直阻塞，是不能被中断的。

    2、可实现公平锁：多个线程在等待同一个锁时，必须按照申请锁的时间顺序排队等待，而非公平锁则不保证这点，在锁释放时，任何一个等待锁的线程都有机会获得锁。synchronized中的锁时非公平锁，ReentrantLock默认情况下也是非公平锁，但可以通过构造方法ReentrantLock（ture）来要求使用公平锁。

    3、锁可以绑定多个条件：ReentrantLock对象可以同时绑定多个Condition对象（条件变量），而在synchronized中，锁对象的wait（）和notify（）或notifyAll（）方法可以实现一个隐含条件，但如果要和多于一个的条件关联的时候，就不得不额外地添加一个锁，而ReentrantLock则无需这么做，只需要多次调用newCondition（）方法即可。而且我们还可以通过绑定Condition对象来判断当前线程通知的是哪些线程（即与Condition对象绑定在一起的其他线程）。

## 可中断锁

ReetrantLock有两种锁：忽略中断锁和响应中断锁。忽略中断锁与synchronized实现的互斥锁一样，不能响应中断，而响应中断锁可以响应中断。

    如果某一线程A正在执行锁中的代码，另一线程B正在等待获取该锁，可能由于等待时间过长，线程B不想等待了，想先处理其他事情，我们可以让它中断自己或者在别的线程中中断它，如果此时ReetrantLock提供的是忽略中断锁，则它不会去理会该中断，而是让线程B继续等待，而如果此时ReetrantLock提供的是响应中断锁，那么它便会处理中断，让线程B放弃等待，转而去处理其他事情。

## 条件变量实现线程间协作

Java 5之后，我们可以用Reentrantlock锁配合Condition对象上的await（）和signal（）或signalAll（）方法来实现线程间协作。在ReentrantLock对象上newCondition（）可以得到一个Condition对象，可以通过在Condition上调用await（）方法来挂起一个任务（线程），通过在Condition上调用signal（）来通知任务，从而唤醒一个任务，或者调用signalAll（）来唤醒所有在这个Condition上被其自身挂起的任务。另外，如果使用了公平锁，signalAll（）的与Condition关联的所有任务将以FIFO队列的形式获取锁，如果没有使用公平锁，则获取锁的任务是随机的，这样我们便可以更好地控制处在await状态的任务获取锁的顺序。与notifyAll（）相比，signalAll（）是更安全的方式。另外，它可以指定唤醒与自身Condition对象绑定在一起的任务。

# [阻塞队列](http://blog.csdn.net/ns_code/article/details/17465497)

阻塞队列是Java 5并发新特性中的内容，阻塞队列的接口是java.util.concurrent.BlockingQueue，它有多个实现类：ArrayBlockingQueue、DelayQueue、LinkedBlockingQueue、PriorityBlockingQueue、SynchronousQueue等。

1.BlockingQueue定义的常用方法如下:

        add(anObject):把anObject加到BlockingQueue里,即如果BlockingQueue可以容纳,则返回true,否则抛出异常

        offer(anObject):表示如果可能的话,将anObject加到BlockingQueue里,即如果BlockingQueue可以容纳,则返回true,否则返回false.

        put(anObject):把anObject加到BlockingQueue里,如果BlockQueue没有空间,则调用此方法的线程被阻断直到BlockingQueue里面有空间再继续.

        poll(time):取走BlockingQueue里排在首位的对象,若不能立即取出,则可以等time参数规定的时间,取不到时返回null

        take():取走BlockingQueue里排在首位的对象,若BlockingQueue为空,阻断进入等待状态直到Blocking有新的对象被加入为止

2.BlockingQueue有五个具体的实现类,根据不同需求,选择不同的实现类

ArrayBlockingQueue:规定大小的BlockingQueue,其构造函数必须带一个int参数来指明其大小.其所含的对象是以FIFO(先入先出)顺序排序的.

        LinkedBlockingQueue:大小不定的BlockingQueue,若其构造函数带一个规定大小的参数,生成的BlockingQueue有大小限制,若不带大小参数,所生成的BlockingQueue的大小由Integer.MAX\_VALUE来决定.其所含的对象是以FIFO(先入先出)顺序排序的

        PriorityBlockingQueue:类似于LinkedBlockQueue,但其所含对象的排序不是FIFO,而是依据对象的自然排序顺序或者是构造函数的Comparator决定的顺序.

        SynchronousQueue:特殊的BlockingQueue,对其的操作必须是放和取交替完成的.

DelayQueue：是一个无界的BlockingQueue，用于放置实现了Delayed接口的对象，其中的对象只能在其到期时才能从队列中取走。

# [障碍器](http://blog.csdn.net/ns_code/article/details/17465497)

CyclicBarrier（又叫障碍器）同样是Java 5中加入的新特性，使用时需要导入java.util.concurrent.CylicBarrier。它适用于这样一种情况：你希望创建一组任务，它们并发地执行工作，另外的一个任务在这一组任务并发执行结束前一直阻塞等待，直到该组任务全部执行结束，这个任务才得以执行。这非常像CountDownLatch，只是CountDownLatch是只触发一次的事件，而CyclicBarrier可以多次重用

# 信号量

Java并发包中的信号量Semaphore实际上是一个功能完毕的计数信号量，从概念上讲，它维护了一个许可集合，对控制一定资源的消费与回收有着很重要的意义。Semaphore可以控制某个资源被同时访问的任务数，它通过acquire（）获取一个许可，release（）释放一个许可。如果被同时访问的任务数已满，则其他acquire的任务进入等待状态，直到有一个任务被release掉，它才能得到许可

Droolls