第一部分线程API

第一章Thread和Runnable

有两种方式创建Runnable对象：

方法一：创建一个实现了Runnable接口的匿名类，如下：

//创建Runnable对象，方法一:创建一个实现了Runnable接口的匿名类  
Runnable r1=new Runnable() {  
 @Override  
 public void run() {  
 System.out.println("hello from thread");  
 }  
};

方法二：使用lambda表达式

//方法二：使用lambda表达式  
Runnable r2=()-> System.out.println("hello from threader as well");

使用Runnalbe对象创建Thread对象：

//使用Runnable对象构造Thread对象  
Thread t1=new Thread(r1);

少数Thread的构造方法不接受Runnable对象，这时你需要定义一个新的继承自Thread的类并重写run方法，然后使用这个新的类来创建线程对象，如：

class MyThread extends Thread{  
 @Override  
 public void run() {  
 System.out.println("hello from thread");  
 }  
}

一个Thread对象关联着一条线程的状态，这些状态包含：

1. 线程名称
2. 线程存活的标识
3. 线程的执行状态
4. 线程的优先级
5. 线程是否为守护线程

每个Thread会被赋予一个名称，这样有利于线程的调试，这个名称可以在构造函数中设置或者通过set设值的方法来设置，如下：

Runnable r1=()-> System.out.println("hello from thread");

//为线程设置名称

Thread t1=new Thread(r1,"thread-1");

System.out.println(t1.getName());

Thread t2=new Thread(t1);

t2.setName("thread-2");

System.out.println(t2.getName());

Thread的getId方法会返回一个长整型的唯一名称，这个数字值在线程的声明周期内不会改变。

获取线程的存活状态，使用线程对象的isAlive方法，如果线程是存活的返回true，一条线程的存活是从start方法启动，结束于从run方法离开，此时线程死亡。

Runnable r2=()->{

System.out.println(Thread.currentThread().isAlive());//true

System.out.println("hello from thread as well");

};

Thread t2=new Thread(r2);

t2.setName("thread-2");

System.out.println(t2.getName());

System.out.println(t2.getId());

t2.start();

线程的执行状态：

Runnable r2=()->{  
 System.out.println(Thread.currentThread().isAlive());//true  
 System.out.println(Thread.currentThread().getState());//RUNNABLE  
 System.out.println("hello from thread as well");  
};  
Thread t2=new Thread(r2);  
t2.setName("thread-2");  
t2.start();//启动线程  
Thread.sleep(1000);  
System.out.println(t2.getState());//可能是RUNNABLE,也可能是TERMINATED

获取和设置线程的优先级：

调用java.lang.Runtime类的availableProcessors返回处理器或者处理器的核心数量，可能在jvm运行时变化，即不一定准确。

操作系统使用调度器来决定什么时候一个线程得以执行，主要的调度器包括：

1. Linux2.6-2.6.23使用的是O(1)调度器
2. Linux2.6.23同时使用Completely Fair调度器，也作为标准调度器
3. Windows基于NT的操作系统使用多级反馈队列调度器

操作系统在进行线程调度时会考虑线程的优先级（这可以得出线程之间的相对重要性），通常会结合抢占式调度和轮转时间片调度。高优先级的线程优先得到更多的机会执行。可以通过线程对象的getPriority和setPriority获取和设置线程的优先级。传递给线程的优先级的值需要介于Thread.MIN\_PRIORITY（1）和Thread.MAX\_PRIORITY（10）之间。如果没有手动设置线程的优先级，则线程获得默认的优先级Thread.NORMA\_PRIORITY（5）。

注意：线程设置优先级可能影响程序跨操作系统的可移植性，不同的操作系统对线程的优先级的实现有些许的差别。

获取和设置线程的守护线程状态：

Java线程分为两类：守护线程和非守护线程。

一条守护线程扮演着非守护线程辅助者的角色，并且会在应用程序最后一条非守护线程消失之后自动死亡。

通过isDaemon和setDaemon来获取和设置线程的守护状态。

注意：主线程main为非守护线程，当非守护线程默认主线程（即main）结束后，应用程序还会等到所有后台的非守护线程终止之后才会终止。如果后台的线程本来就是守护线程，那么当默认的主线程终止时，应用程序会立刻终止。

Runnable r1=()->{  
 System.out.println("hello from thread");  
 for(int index=0;;) {  
 try {  
 System.out.println(++index);  
 Thread.sleep(1000);  
 } catch (InterruptedException ex) {  
 ex.printStackTrace();  
 }  
 }  
};  
Thread t1=new Thread(r1,"thread-1");  
t1.setDaemon(true);  
t1.setPriority(9);  
t1.start();  
Thread.sleep(5000);

启动线程

通过调用Thread类或者其子类对象的start方法来启动与该对象关联的线程。

线程不能重复启动：如果线程已经启动并且处于运行状态或者线程已经死亡，则调用start方法会抛出异常。

操作更高级的线程任务：中断其他线程、join以及线程睡眠

1. void interrupt()：中断调用此方法的Thread对象所关联的线程（即中断自己）；当一条线程由于调用了Thread的sleep或者join方法而被阻塞，该线程的中断状态就会被消除他，同时抛出InterruptedException，否则出了会设置中断状态，其他动作也会取决于当前线程行为相应的发生。
2. static Boolean interrupted()：验证当前线程是否已经中断，该线程的中断状态会被这个方法清除掉（即中断的线程恢复可运行状态）

public static void main(String[] args) throws Exception{  
 Runnable r1=new Runnable() {  
 private int count=0;  
 @Override  
 public void run() {  
 while(count++<10){  
 System.out.println("thread-1");  
 try{  
 Thread.sleep(100);  
 }catch(InterruptedException ex){  
 ex.printStackTrace();  
 }  
 }  
 }  
 };  
  
 //ExecutorService executorService= Executors.newFixedThreadPool(1);  
 Thread t1= new Thread(r1);  
 t1.start();  
 Thread.sleep(10);  
 t1.interrupt();  
 System.out.println(t1.isInterrupted());  
 //Thread.sleep(1000);  
 t1.join();  
 System.out.println(t1.isAlive());  
}

1. boolean isInterrupted:验证线程是否已经中断，该线程的中断状态不受此方法的影响。

等待线程

线程会偶尔启动另一个线程取操作单调计算、下载大文件或者操作一些其他的耗时任务，在结束他自己的任务后，这个启动工作线程的线程就准备着处理工作线程的结果，同时等待该工作线程的结束。

Thread类提供了3种join方法，允许调用线程等待执行此方法的线程对象所关联的线程执行完毕：

1. void join()

无限的等待其他指定的线程直至该线程死亡。当任意线程中断当前线程的时候，InterruptedException就会抛出，如果该异常被抛出，该线程的中断状态就会被清除。

1. void join(long millis)

该线程死亡之前最多等待millis毫秒，0表示无限期等待，与join等价。

1. void join(long mills,int nanos)

该线程死亡之前最多等待mills毫秒+nanos纳秒

public class JoinDemo {  
 private static final BigDecimal FOUR=BigDecimal.valueOf(4);  
 private static final int ROUNDING\_MODE=BigDecimal.ROUND\_HALF\_EVEN;  
 private static BigDecimal result;  
  
 public static void main(String[] args) {  
 Runnable r=()->{  
 result=computePi(50000);  
 };  
  
 Thread t=new Thread(r);  
 t.start();  
 try{  
 t.join();//主线程等待工作线程t的完成/死亡  
 }catch(InterruptedException ex){}  
  
 System.out.println(result);  
 }  
  
 private static BigDecimal computePi(int digits){  
 int scale=digits+5;  
 BigDecimal arctan1\_5=arctan(5,scale);  
 BigDecimal arctan1\_239=arctan(239,1);  
 BigDecimal pi=arctan1\_5.multiply(FOUR).subtract(arctan1\_239).multiply(FOUR);  
  
 return pi.setScale(digits,BigDecimal.ROUND\_HALF\_UP);  
 }  
  
 private static BigDecimal arctan(int inverseX,int scale){  
 BigDecimal result,number,term;  
 BigDecimal invX=BigDecimal.valueOf(inverseX);  
 BigDecimal invX2=BigDecimal.valueOf(inverseX\*inverseX);  
 number=BigDecimal.ONE.divide(invX,scale,ROUNDING\_MODE);  
 result =number;  
 int i=1;  
 do{  
 number=number.divide(invX2,scale,ROUNDING\_MODE);  
 int denom=2\*i+1;  
 term=number.divide(BigDecimal.valueOf(denom),scale,ROUNDING\_MODE);  
 if((i%2)!=0){  
 result=result.subtract(term);  
 }else{  
 result=result.add(term);  
 }  
  
 i++;  
 }while(term.compareTo(BigDecimal.ZERO)!=0);  
  
 return result;  
 }  
}

在上面的例子中，默认的主线程在Thread对象上调用了join方法等待工作线程死亡，工作线程死亡了，默认主线程才会继续执行（me:join是通过调用线程对象的wait来实现的）。

线程睡眠

Thread类声明了几个静态方法来使得线程睡眠（暂时的停止执行，如果线程持有锁/监听对象，不会释放）：

1. void sleep(long mills)

睡眠millis毫秒

1. void sleep(long mills,int nanos)

sleep方法相对于忙循环要好，因为它不会浪费处理器周期。

第二章 同步

线程之间通常是通过共享变量完成交互即线程通信，当线程之间没有共享变量的时候开发线程会非常的简单（此时不用考虑线程安全问题）。

使用多线程意味着增加了系统的复杂性。

线程的问题多和竞态条件、数据竞争和缓存变量有关。

线程中的问题：

1. 竞态条件：调度器控制的多线程出现交叉时，竞态条件就会发生

示例：

if(a==10.0)  
 b=a/2.0;

1. 在单线程条件下这段代码没有什么问题
2. 在多线程条件下，如果这两个变量都是局部变量那么也没有什么问题
3. 如果在多线程条件下，这两个变量是成员变量或者静态变量，并且多线程对这两个变量都有访问权限，那么就会出现问题。

竞态条件分为两类，上面的是一种，被称为check-then-act，还有一种被称为read-modify-write，典型代码如下：

public void incrCount(){  
 this.count++;  
}

表达式count++是三个独立的操作，且可以分割：读取count,+1,更新count的值（me:在线程的工作空间内进行）

1. 数据竞争

数据竞争指的是两条或者多条线程(在单个应用中)并发的访问同一块内存区域，同时其中至少一条是为了写，而且这些线程没有协调对那块内存区域的访问。这种情况下因为每次执行程序时，访问顺序是不确定的，结果也就不确定。

典型代码，懒汉式异步单例：

class Parser{  
 private static Parser instance;  
 private Parser(){}  
 public static Parser getInstance(){  
 if(instance==null){  
 instance=new Parser();  
 }  
  
 return instance;  
 }  
}

1. 缓存变量

为了提升性能，编译器java虚拟机jvm以及操作系统会协调在寄存器中或者处理器缓存中缓存变量，而不是依赖主存。每条线程都有自己的变量拷贝，但线程写入这个变量的时候，其实是写入自己的拷贝，其他线程不太可能看到这个共享的变量发生了改变。(典型的线程下可见性问题，由此出现了volatile关键字,me：volatile也解决了指令重排序问题)

同步能够解决之前的线程问题，它旨在保证两个或者多个并发的线程不会同时执行同一块临界区，临界区就是必须以串行的方式（一次一条线程）访问的一段代码块。

线程对临界区的访问都是互斥执行，所以锁称为互斥锁。

同步是通过监听器来实现的，监听器控制线程对临界区的访问控制，每一个java对象都和一个监听器相关联(即每个java对象都可以作为一个监听器，此监听器的方法写在了Object中)，这样线程就可以通过获取和释放监听器（一个标识）来上锁和解锁，只有获取到了锁才有操作临界区的权限。

Thread类的sleep方法被调用时不会释放同步锁。

只有一个线程可以持有监听器的锁，其他尝试锁住该监听器的线程都会一直阻塞，直到能够获取锁为止。当线程离开临界区它会释放锁解锁监听器。Thread的静态方法holdsLock可以判断线程是否持有某个对象的锁。

Runnable r1=new Runnable() {  
 @Override  
 public void run() {  
 synchronized (this){  
 try{  
 System.out.println("thread:"+Thread.currentThread().getName()+":"+Thread.currentThread().holdsLock(this));  
 Thread.sleep(5000);  
 }catch(InterruptedException ex){  
 ex.printStackTrace();  
 }  
  
 }  
 }  
};  
  
Thread th1=new Thread(r1);  
Thread th2=new Thread(r1);  
th1.start();  
th2.start();  
Thread.sleep(1000);  
System.out.println("main:th1:"+th1.holdsLock(r1));  
System.out.println("main:th2:"+th2.holdsLock(r1));

Synchronized关键字：

1. 当同步在实例方法上，锁会和调用该方法的实例对象相关联，所以在实例方法内部的同步代码块上使用synchronized(this)，
2. 当同步在一个类方法上，锁会和调用该类方法的类所对应的java.lang.Class对象相关联，所以在类的静态方法内部的同步代码块上使用synchronized(Demo.class)

使用同步块，将synchronized修饰在语句块上。

谨防活跃性问题：

1. 死锁：线程1等待线程2释放锁A，线程2等待线程1释放锁B
2. 活锁：线程获取锁总是失败，还总是尝试
3. 饿死：线程总是得不到调度器的调度导致线程总是得不到执行。饿死通常也称为无限延迟。

死锁实例：

class DeadLockDemo {  
 private final Object lock1=new Object();  
 private final Object lock2=new Object();  
  
 public void instanceMethod1(){  
 synchronized (lock1){  
 synchronized (lock2){  
 System.out.println("first thread in instanceMethod1");  
 }  
 }  
 }  
  
 public void instanceMethod2(){  
 synchronized (lock2){  
 synchronized (lock1){  
 System.out.println("second thread in instanceMethod2");  
 }  
 }  
 }  
  
 public static void main(String[] args) {  
 final DeadLockDemo dld=new DeadLockDemo();//同一个对象实例化的线程  
 Runnable r1=new Runnable() {  
 @Override  
 public void run() {  
 while (true) {  
 dld.instanceMethod1();  
 try {  
 Thread.sleep(50);  
 } catch (InterruptedException ex) {  
 ex.printStackTrace();  
 }  
 }  
 }  
 };  
  
 Thread thdA=new Thread(r1);  
 Runnable r2=new Runnable() {  
 @Override  
 public void run() {  
 while (true) {  
 dld.instanceMethod2();  
 try {  
 Thread.sleep(50);  
 } catch (InterruptedException ex) {  
 ex.printStackTrace();  
 }  
 }  
 }  
 };  
  
 Thread thdB=new Thread(r2);  
  
 thdA.start();  
 thdB.start();  
 }  
}

java语言和jvm都没有提供一种方式来避免死锁。需要程序员编写程序时避免此问题。

同步的两种属性：

1. 互斥性
2. 可见性

Volatile和final变量：

Synchronized与互斥性和可见性都有关，volatile只与可见性有关，使用volatile使得每个线程都从主存中获取该变量的值，而不是从自己的拷贝，并将操作的结果更新到缓存。

停止线程实例1：

class ThreadStopping {  
 public static void main(String[] args) {  
 class StoppableThread extends Thread{  
 private boolean stopped;  
  
 @Override  
 public void run() {  
 while(!this.stopped){  
 try{  
 Thread.sleep(10);  
 }catch(InterruptedException ex){  
 ex.printStackTrace();  
 }  
 System.out.println("running");  
 }  
 }  
  
 public void stopThread(){  
 this.stopped=true;  
 }  
 }  
  
 StoppableThread thd=new StoppableThread();  
 thd.start();  
 try{  
 Thread.sleep(1000);  
 }catch(InterruptedException ex){  
 ex.printStackTrace();  
 }  
  
 thd.stopThread();  
 }  
}

上述程序如果在单核/单处理器的机器上运行时，可能没有什么问题，能够观察到程序运行结束，但是如果放到多处理器/多核的机器上运行时，则可能不会观察到程序运行结束，因为每个处理器或者核心很可能有自己的一份stopped的拷贝，当一条线程修改了自己的拷贝，其他线程的拷贝并没有改变。

获取锁会存在性能开销。

使用volatile关键字：

private volatile boolean stopped;

其他代码同上，这时就能解决多线程的可见性问题。

只有在存在可见性问题时才应该使用volatile关键字。并且该关键字只能修饰属性，不能修饰局部变量。

当一个属性被声明为volatile时就不能声明为final的了，java可以让你安全的访问final的属性而不需要同步。

关于不可变对象java提供了一种特殊的线程安全保证，也并不存在变量缓存的问题，因为变量不可变。

不可变对象编程规则：

1. 不可变对象绝对不允许状态变更
2. 所有的不可变属性必须声明为final
3. 对象必须被恰当的构造出来以防this引用脱离构造函数

第三章等待和通知

线程间交互，可以使用共享变量和wait,notify

等待和通知API一览：java .lang.Object类提供了一套等待和通知的API，它们包含3个wait方法和1个notify方法和1个notifyAll方法：

1. Void wait():导致当前线程一直等待，直到其他的线程调用这个对象的（即锁对象）notify或者notifyAll方法。又或者一直等待其他的线程中断当前等待的线程。
2. Void wait(long timeout): 导致当前线程一直等待，直到其他的线程调用调用这个对象的（即锁对象）notify或者notifyAll方法。或者等待特定的毫秒数，又或者一直等待其他的线程中断当前等待的线程。
3. Void wait(long timeout,int nanos): 导致当前线程一直等待，直到其他的线程调用调用这个对象的（即锁对象）notify或者notifyAll方法。或者等待特定的毫秒数和纳秒数，又或者一直等待其他的线程中断当前等待的线程。
4. Void notify():唤醒正在等待该对象监听器的单条线程。如果有多条线程在该对象上等待，则随意的唤醒一条。被唤醒的线程加入到对对象锁的竞争。唤醒后的线程与普通的线程没有什么区别，在接下来锁住对象这件事上没有什么优势和劣势。
5. Void notifyAll()：唤醒正在等待该对象监听器的所有线程。被唤醒的线程加入到对对象锁的竞争。唤醒后的线程与普通的线程没有什么区别，在接下来锁住对象这件事上没有什么优势和劣势。

注意：wait会导致当前线程释放锁持有的同步锁。

如果当前线程开始或正在等待通知，任意线程中断了它，3个wait方法都会抛出异常，此时线程的中断状态会被清除。

这组API利用了一个对象的条件队列，该队列就是一个种数据结构，用于存储那些等待某个条件成立的线程。这些等待的线程称为等待集合。由于该条件队列和对象的锁紧紧捆绑在一起，以上5个方法必须在同步的上下文（当前线程必须是该对象监听器的所有者）被调用，否则会抛出IllegalMonitorStateException异常。

实例代码：

public class WaitInterrupt {  
 public static void main(String[] args) {  
 Runnable r=new Runnable() {  
 private final Object lock=new Object();  
 @Override  
 public void run() {  
 System.out.println("running:inner run");  
 try {  
 synchronized (lock){  
 System.out.println(Thread.currentThread().getName()+" notify");  
 lock.notify();  
  
 System.out.println(Thread.currentThread().getName()+" wait");  
 lock.wait();  
  
 System.out.println("hahahah");  
 }  
 } catch (InterruptedException ex) {  
 ex.printStackTrace();  
 }  
 }  
 };  
  
 Thread t1=new Thread(r);  
 Thread t2=new Thread(r);  
 t1.start();  
 t2.start();  
 }  
}

notify是从临界区调用的并且使用同样的监听对象。

锁-监听器/监听对象

生产者和消费者实例：

public class PC {  
 public static void main(String[] args) {  
 Shared s=new Shared();  
 new Producer(s).start();  
 new Consumer(s).start();  
 }  
}  
  
class Shared {  
 private char c;  
 private volatile boolean writable = true;  
  
 synchronized void setSharedChar(char c) {  
 while (!writable) {  
 try {  
 wait();  
 } catch (InterruptedException ex) {  
 }  
 }  
 this.c = c;  
 writable = false;  
 notify();  
 }  
  
 synchronized char getSharedChar() {  
 while (writable) {  
 try {  
 wait();  
 } catch (InterruptedException ex) {  
 }  
 }  
 writable = true;  
 notify();  
 return c;  
 }  
}  
  
class Producer extends Thread{  
 private final Shared s;  
 Producer(Shared s){  
 this.s=s;  
 }  
 @Override  
 public void run() {  
 for(char ch='A';ch<='Z';ch++){  
 synchronized (s){  
 s.setSharedChar(ch);  
 System.out.println(ch+" produced by producer.");  
 }  
 }  
 }  
}  
  
class Consumer extends Thread{  
 private final Shared s;  
 Consumer(Shared s){  
 this.s=s;  
 }  
  
 @Override  
 public void run() {  
 char ch;  
 do{  
 synchronized (s){//使用与生产者的共享对象，才能获取控制生产-消费的顺序  
 ch=s.getSharedChar();  
 System.out.println(ch+" consumed by consumer.");  
 }  
 }while(ch!='Z');  
 }  
}

第四章额外的线程能力

线程组：一个线程组代表了一组线程。除此之外，一个线程组也可以包含其他的线程组，这些线程组形成一棵树，其中除了初始线程，每个线程都有一个父线程组。

基于多种原因不能使用线程组类：ThreadGroup。

线程异常捕获和处理：

public class ExceptionThread {  
 public static void main(String[] args) {  
 Runnable r=new Runnable() {  
 @Override  
 public void run() {  
 int x=1/0;  
 }  
 };  
  
 Thread thd=new Thread(r);  
 Thread.UncaughtExceptionHandler uceh=new Thread.UncaughtExceptionHandler() {  
 @Override  
 public void uncaughtException(Thread t, Throwable e) {  
 System.out.println("Caguht throwable "+e+" for thread "+t);  
 }  
 };  
 thd.setUncaughtExceptionHandler(uceh);  
 thd.start();  
 }  
}

线程局部变量

每个java.lang.ThreadLocal的实例代表了一个线程局部变量。它为每一条访问线程提供了单独的存储槽，你可以把它想象成具有多个槽的变量，然后每条线程可以在同一个变量中存储不同的值，并且每条线程都只能看到自己的值，而不会意识到其他线程在这个变量中也有自己的值。

ThreadLocal类被声明为泛型类，方法如下：

1. ThreadLocal：构造方法，创建一个新的线程局部变量
2. T get()：返回调用线程的存储槽中的值。当这个线程调用此方法时，如果值不存在则get()会调用initialValue方法
   1. T initialValue：创建调用线程的存储槽，并存入一个初始值，默认的只是指为null，你必须继承类ThreadLocal并重写其protected方法以提供和一个更加合适的初始值。
3. Void remove：清空调用线程的存储槽
4. Void set(T value)：设置调用线程存储槽上的值。

示例：

public class ThreadLocalDemo {  
 private static volatile ThreadLocal<String> userId=new ThreadLocal<>();  
  
 public static void main(String[] args) {  
 Runnable r=new Runnable() {  
 @Override  
 public void run() {  
 String name=Thread.currentThread().getName();  
 if(name.equalsIgnoreCase("A")){  
 userId.set("Daniel");  
 }else{  
 userId.set("Mary");  
 }  
  
 System.out.println(name+" "+userId.get());  
 }  
 };  
  
 Thread t1=new Thread(r);  
 Thread t2=new Thread(r);  
 t1.setName("A");  
 t2.setName("B");  
  
 t1.start();  
 t2.start();  
 }  
}

使用volatile是因为该程序可能在多处理器/多核心的机器上运行。

存储在线程本地变量中的值都是不相关的，有时候你或许想要把值从父线程传给子线程，可以借助InheriableThreadLocal来完成这个任务。InheriableThreadLocal是ThreadLocal的子类，除了定义了一个InheriableThreadLocal构造方法，这个类还定义了下面的方法：

T childValue(T parentValue)：在子线程被创建出来的时候，用父线程的值（作为参数）计算出子线程的初始值，此方法会在子线程启动之前被父线程调用，若它没被重写就会返回参数parentValue，反之返回期望的值。

定时器任务

Timer TimerTask

示例：

一次执行

public class TimerDemo {  
 public static void main(String[] args) {  
 TimerTask task=new TimerTask() {  
 @Override  
 public void run() {  
 System.out.println("alarm going off");  
 System.exit(0);  
 }  
 };  
  
 Timer timer=new Timer();  
 timer.schedule(task,2000);//2秒后执行  
 }  
}

定时执行

public class TimerDemo2 {  
 public static void main(String[] args) {  
 TimerTask task=new TimerTask() {  
 @Override  
 public void run() {  
 System.out.println(new Date());  
 }  
 };  
  
 Timer timer=new Timer();  
 timer.schedule(task,1000,1000);//1秒中后每隔1秒执行一次task  
 }  
}

Timer适用于大规模并发调度定时任务（数以千计的任务不在话下）。

Timer声明了下列方法：

1. Timer():构造方法，创建一个新的定时器，其任务执行线程是非守护线程
2. Timer(Boolean isDaemon)：创建一个新的定时器，当传入true时，其任务执行线程会以守护线程的方式运行。

守护线程在定时器被用于调度重复的维护活动的场景下使用。只要应用程序处于运行状态这些维护活动就会一直被执行，但是它们不会延长应用的存在时长，因为守护线程会在所有非守护线程执行完毕后自动结束。

1. Timer(string name)：创建一个定时器，其任务执行线程拥有指定的名称且不会以守护线程的方式执行。
2. Timer(string name,Boolean isDaemon)
3. Void cancel():终止该定时器，丢弃所有当前调度的定时任务。执行线程优雅的终止。
4. Int purg()：从定时器队列中移除所有取消的定时任务并且返回被移除任务的数目。大部分应用程序不需要调用这个方法，该方法设计用于极少数需要取消大量定时器任务的应用程序。
5. Void schedule(TimerTask task,Date time):在某个时间点调度任务执行，如果time是过去时间任务会被立即执行。
6. Void schedule(TimerTask task,Date firstTime,long period):调度任务于firstTime开始，以固定时间间隔的方式重复执行。如果firstTime是过去时间任务会被立即执行。
7. Void schedule(TimerTask task,long delay):在delay毫秒数之后调度任务执行
8. Void schedule(TimerTask task,Date firstTime,long period):调度任务于firstTime开始，然后以固定的时间间隔的方式执行。
9. Void scheduleAtFixedRate(TimerTask task,Date firstTime,long period):调度任务于firstTime开始，以固定的速率重复执行。
10. Void scheduleAtFixedRate(TimerTask task,long delay,long period):调度任务于delay毫秒后，调度任务开始以固定的方式重复执行。

当应用程序想要快速的终止一个定时器执行线程，它应该调用Timer的cancel方法（me：否则定时器一旦运行起来就不会停止）。

TimerTask

定时任务都是抽象类TimerTask子类的实例，这些子类实现了Runnable接口，当子类化TimerTask的时候，需要重写其run方法，可以在该run方法中调用下列方法：

1. Boolean cancel()：取消这个定时任务
2. Long scheduledExcetionTime()：返回此定时任务最近被实际调度的执行时间

第二部分并发工具类

第五章并发工具类和Executor框架

Synchronized会带来性能问题。

并发工具可以被分为executor、同步器（synchronized）和锁框架等。

并发工具类框架包含：

1. Java.util.concurrent：经常在并发编程中使用的工具类，如executors
2. Java.util.concurrent.atomic：支持在单个变量上进行无锁且线程安全操作的工具类。
3. Java.util.concurrent.locks:在某些条件上获取锁和执行等待的工具类。

一个任务是这样的一种对象，它的类实现了java.lang.Runnable（一个可运行的任务）或者java.util.concurrent.Callbale接口（一个可被调用的任务）。

创建线程的几种方式：继承自Thread，实现Runnable和实现Callable接口，使用ThreadFactory。

Executor声明了一个单独的void execute(Runnable runnable)方法，该方法会在将来的某个时间点执行这个名为runnable的可执行任务。

Executor类有很多的缺陷，如执行线程没有返回值，不能有效的追踪正在运行的线程等，因此开发中实际使用的是java.util.concurrent.ExecutorService接口。这个接口扩展Executor接口并且其实现就是一个典型的线程池。它包含的方法如下：

1. Boolean awaitTermination(long timeout,TimeUnit unit):
2. List<Future<T>> invokeAll(<T> Collection<? Extends Callable<T>> tasks):执行任务集合中每一个callable任务并当所有的任务执行完成后（即任务要么正常结束，要么抛出异常），返回一个java.util.list 的java.util.concurrent.Future的实例，这些实例会持有任务的状态和结果。
3. List<Future<T>> invokeAll(<T> Collection<? Extends Callable<T>> tasks,long timeout,TimeUnit unit)
4. T invokeAny(<<T> Collection<? Extends Callable<T>> tasks)
5. T invokeAny(<T> Collection<? Extends Callable<T>> tasks,long timeout,TimeUnit unit)
6. boolean isShutdown():当executor已经终止时，返回true,否则返回false
7. boolean isTerminated()：若紧接着终止之后所有的任务都完成了，返回true，否则返回false
8. void shutdown():该方法不会等待之前提交的任务执行完毕，当需要等待时，使用awaitTermination方法。
9. List<Runnable> shutdown():尝试停止所有活跃的执行线程
10. Future<T> submit(Callable<T> task):提交一个callable任务来执行
11. Future<?> submit(Runnable task):提交一个runnable任务来执行同时返回一个代表此任务的Future实例，该Future的get方法在成功完成之后返回null。
12. Future<T> submit(Runnable task,T result)：提交一个runnable任务来执行，同时返回一个Future实例，其get方法在成功完成的情况下返回结果的值。

Callable的call方法有返回值，所以如果希望线程在执行之后有返回值需要实现Callable接口，并使用Executor来执行task。

Callable任务，不同于Runnable的void run方法既不能返回值也不能抛出受检异常，Callable的V call() 方法不仅可以返回值还可以抛出受检异常。因为它声明了一个throws Exception子句。

Future接口，它代表着一种异步计算的结果，这个结果之所以称为Future，是因为它通常需要到未来的某个时刻才有效，Future的泛型是Future<V>，它提供了取消任务、返回任务的结果以及判断任务是否已经结束的方法。

Future接口的方法：

1. Boolean cancel(boolean mayInterruptIfRunning):尝试取消此任务的执行，并且当该任务被取消时返回true，否则返回false（任务可能在cancel方法被调用之前已经正常结束）。
2. V get():返回此线程执行之后返回的结果，即call方法的返回值。
3. V get(long timeout,TimeUnit unit):最多等待timeout时间单元直到此任务完成执行，之后返回结果。
4. Boolean isCancelled():当此任务在正常完成之前被取消了返回true，否则返回false。
5. Boolean isDone():当此任务完成返回true，否则返回false。此方法在正常的中断、出现异常或者取消等场景下都会返回true。

接口java.util.concurrent.ScheduledExecutorService继承自接口ExecutorServcices，它代表了一种能够让你调度任务运行一次或者在指定延迟之后周期性执行的executor（executor是一个接口）。

Executors工具类声明了几个类方法用来返回多种ExecutorService和ScheduledExecutorService实现的实例，这个类的静态方法完成了下列任务：

1. 创建并返回一个ExecutorService实例，使用常用的配置项设置
2. 创建并返回一个ScheduledExecutorService实例，使用常用的配置项设置
3. 创建并返回一个包装过的ExecutorService或ScheduleExectorSevice的实例。这个实例让某些方法变得不可访问，从而禁止重新配置executor service。
4. 创建并返回一个ThreadFactory实例来创建新的线程对象
5. 创建并返回一个了闭包的Callable实例，它可以用在那些接收Callable参数的执行方法中。

例如，static ExecutorService newFixedThreadPool(int nThreads)方法创建了一个线程池复用固定数量的线程操作一个共享的无限队列，至多有nThreads个线程同时处理任务。如果额外的任务在所有线程都活跃时被提交了，它们会在队列中等待一条可用线程。这些线程池中的线程会一直存在直到被executor被显示的关闭。

线程池用来消除不得不为每条提交任务创建一条新线程的开销，线程创建并不廉价，必须创建大量线程可能会严重影响应用程序的性能。

示例：

public class CalculateE {  
 final static int LASTITER=17;  
  
 public static void main(String[] args) {  
 ExecutorService executor=Executors.newFixedThreadPool(2);  
 Callable<BigDecimal> callable1,callable2;  
 callable1=new Callable<BigDecimal>() {  
 @Override  
 public BigDecimal call() throws Exception {  
 MathContext mc=new MathContext(100, RoundingMode.HALF\_UP);  
 BigDecimal result=BigDecimal.ZERO;  
 for(int i=0;i<LASTITER;i++){  
 BigDecimal factorial=factorial(new BigDecimal(i));  
 BigDecimal res=BigDecimal.ONE.divide(factorial,mc);  
 System.out.println("\*\*\*\*\*"+Thread.currentThread().getName()+"\*\*\*\*\*");  
 result=result.add(res);  
 }  
  
 return result;  
 }  
  
 private BigDecimal factorial(BigDecimal n){  
 if(n.equals(BigDecimal.ZERO))  
 return BigDecimal.ONE;  
 else  
 return n.multiply(factorial(n.subtract(BigDecimal.ONE)));  
 }  
 };  
 callable2=new Callable<BigDecimal>() {  
 @Override  
 public BigDecimal call() throws Exception {  
 MathContext mc=new MathContext(100, RoundingMode.HALF\_UP);  
 BigDecimal result=BigDecimal.ZERO;  
 for(int i=0;i<LASTITER;i++){  
 BigDecimal factorial=factorial(new BigDecimal(i));  
 BigDecimal res=BigDecimal.ONE.divide(factorial,mc);  
 System.out.println("\*\*\*\*\*"+Thread.currentThread().getName()+"\*\*\*\*\*");  
 result=result.add(res);  
 }  
  
 return result;  
 }  
  
 private BigDecimal factorial(BigDecimal n){  
 if(n.equals(BigDecimal.ZERO))  
 return BigDecimal.ONE;  
 else  
 return n.multiply(factorial(n.subtract(BigDecimal.ONE)));  
 }  
 };  
  
 Future<BigDecimal> taskFuture1=executor.submit(callable1);  
 Future<BigDecimal> taskFuture2=executor.submit(callable2);  
 try{  
 while(!taskFuture2.isDone()){  
 System.out.println("waiting--2");  
 }  
 System.out.println(taskFuture2.get());  
  
 while(!taskFuture1.isDone()){  
 System.out.println("waiting--1");  
 }  
 System.out.println(taskFuture1.get());  
 }catch(ExecutionException ex){  
 System.err.println("task threw an exception");  
 System.err.println(ex);  
 }catch(InterruptedException ex){  
 System.err.println("interrpted while waiting");  
 }  
  
 executor.shutdownNow();  
 }  
}

在executor完成之后关闭是十分重要的，否则此应用程序可能不会终止。

Shutdown:会将已经提交的任务执行完，不会接收新的task。

newFixedThreadPool:

public static ExecutorService newFixedThreadPool(int nThreads) {  
 return new ThreadPoolExecutor(nThreads, nThreads,  
 0L, TimeUnit.MILLISECONDS,  
 new LinkedBlockingQueue<Runnable>());  
}

newSingleThreadExecutor:

public static ExecutorService newSingleThreadExecutor() {  
 return new FinalizableDelegatedExecutorService  
 (new ThreadPoolExecutor(1, 1,  
 0L, TimeUnit.MILLISECONDS,  
 new LinkedBlockingQueue<Runnable>()));  
}

newCachedThreadPool:

public static ExecutorService newCachedThreadPool() {  
 return new ThreadPoolExecutor(0, Integer.MAX\_VALUE,  
 60L, TimeUnit.SECONDS,  
 new SynchronousQueue<Runnable>());  
}

Initiates an orderly shutdown in which previously submitted  
\* tasks are executed, but no new tasks will be accepted.  
\* Invocation has no additional effect if already shut down.

ShutdownNow立即停止所有的正在执行的任务，挂起等待执行的任务，并返回这些挂起的任务。

Attempts to stop all actively executing tasks, halts the  
\* processing of waiting tasks, and returns a list of the tasks  
\* that were awaiting execution.

工具类包：

Java.util.concurrent

Java.util.concurrent.atomic

Java.util.concurrent.locks

Executor框架包含：

Executor

ExecutorService

ScheduledExecutorService

Executors会和Callable及Future相关联。

第六章 同步器

并发工具类提供了高级的同步器（控制通用同步方法的类），包括倒计时门闩、同步屏障、交换器、信号量和phaser同步器。

CountDownLatch:倒计时门闩，提供了下列方法：

1. CountDownLatch(int count):构造方法，初始化一个CountDownLatch实例
2. Void await():除非线程被中断，否则强制调用线程一直等到计数器倒数到0。
3. Boolean await(long timeout,TimeUnit unit)：除非线程被中断，否则强制调用线程一直等到计数倒数到0或者以unit作为时间单元的timeout超时。
4. Void countDown(): 递减计数，当计数将至0时释放所有等待线程。如果当该方法被调用时count已经是0则什么也不会发生。
5. long getCount()：返回当前的计数，该方法多用于测试和调试
6. string toString()：返回一条标识这个门闩即其状态的字符串。

经常使用倒计时门闩即CountDownLatch来保证多条线程几乎同时开始工作（me:控制线程的执行顺序）。

示例：

public class CountDownLatchDemo {  
 final static int NTHREADS=3;  
  
 public static void main(String[] args) {  
 final CountDownLatch startSignal=new CountDownLatch(1);  
 final CountDownLatch doneSignal=new CountDownLatch(1);  
 Runnable r=new Runnable() {  
 @Override  
 public void run() {  
 try{  
 report("entered run()");  
 startSignal.await();  
 report("doing work");  
 Thread.sleep((int)Math.random()\*1000);  
 doneSignal.countDown();  
 }catch(InterruptedException ex){  
 ex.printStackTrace();  
 }  
 }  
  
 void report(String s){  
 System.out.println(System.currentTimeMillis()+": "+Thread.currentThread()+": "+s);  
 }  
 };  
  
 ExecutorService executor= Executors.newFixedThreadPool(NTHREADS);  
 for(int i=0;i<NTHREADS;i++){  
 executor.execute(r);  
 }  
  
  
 try{  
 System.out.println("main thread doing something");  
 Thread.sleep(1000);  
 startSignal.countDown();  
 System.out.println("main thread doing something else");  
 doneSignal.await();  
 executor.shutdownNow();  
 }catch(InterruptedException ex) {  
 System.err.println(ex);  
 }  
 }  
}

默认主线程首先创建了一个倒计时门闩，这个startSignal门闩会在默认主线程就绪之前禁止任何工作线程执行，而doneSignal门闩会使得默认主线程等待所有的工作线程全部执行结束。

package com.test.thread\_study.chapter06;  
  
import java.util.concurrent.CountDownLatch;  
import java.util.concurrent.Executors;  
import java.util.concurrent.ThreadFactory;  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
/\*\*  
 \* Created by IntelliJ IDEA  
 \* Project: ThreadDemo  
 \* User: DaiYan  
 \* Date: 2017/10/24  
 \*/  
public class Demo2 {  
 static class R1 implements Runnable{  
 private final CountDownLatch countDownLatch=new CountDownLatch(3);  
 private Lock lock=new ReentrantLock();  
 @Override  
 public void run() {  
 System.out.println(Thread.currentThread().getName()+" enter thread-run");  
 lock.lock();  
 if(countDownLatch.getCount()>0) countDownLatch.countDown();  
 lock.unlock();  
 try{  
 if(countDownLatch.getCount()>0)  
 countDownLatch.await();  
 }catch(InterruptedException ex){  
 ex.printStackTrace();  
 }  
  
 System.out.println(Thread.currentThread().getName()+" running back");  
 }  
 }  
 public static void main(String[] args) {  
 Runnable r=new R1();  
 Executors.defaultThreadFactory().newThread(r).start();  
 Executors.defaultThreadFactory().newThread(r).start();  
 Executors.defaultThreadFactory().newThread(r).start();  
 Executors.defaultThreadFactory().newThread(r).start();  
 }  
}

//=============未完待续20171016

第七章 锁框架

Java.util.concurrent.locks包提供了一个包含多种接口和类的框架，它针对条件进行加锁和等待。包含锁框架的并发工具通过轮询锁、限时等待即其他方式。

Java支持同步以便线程能够安全的更新共享变量并且保证一条线程的更新对其他线程可见，可以给代码块标记上synchronized关键字以达到同步的目的，这样的代码序列被称为临界区。

Jvm通过jvm的指令monitors、monitorenter和monitorexit来实现同步的目的。

每一个java对象都和一个监听器相关联，监听器是一个互斥（每次只允许一条线程在临界区中执行）构造。它阻止多条线程同时在临界区中并发执行。在线程可以进入临界区（任何此锁控制的临界区）之前，他需要锁住监听器，如果这个监听器已经上锁，在监听器释放之前这条线程会一种阻塞（因为其他线程正在使用临界区）。

当线程在多核、多处理器的环境中锁住一个监听器，存储在主存中的共享变量的值会被读取到对应的拷贝中，然后存储在线程的工作内存（称为本地内存或者缓存）。这一动作能够保证该线程使用这些变量最近的值并且不会污染这些值，我们称为可见性。线程会持续使用这些共享变量的拷贝，当离开临界区，线程会释放监听器，这些共享变量的值就会被写回内存，以确保下一条线程进入临界区也能访问这些变量最近的值。Volatile关键字仅仅解决了可见性。

锁接口Lock:

1. void lock():获取锁，当锁不可用时调用线程会被强制一直等待直到锁可用。
2. void lockInterruptibly()：除非调用线程被中断，否则获得锁，当锁不可用时，调用线程被强制一直等待，直到锁可用或线程中断。
3. Condition newCondition()：返回一个新的绑定到当前锁实例之上的Condition实例，当Lock的实现类不支持条件（Condition）时，该方法抛出异常。
4. Boolean tryLock():在该方法被调用时，锁可用则获得锁。当获得锁之后，该方法返回true，反之返回false.
5. Boolean tryLock(long time,TimeUnit unit)：当在指定等待时间之内锁可用并且调用线程没有被中断，则获得锁。当锁不可用时，调用线程会被强制一直等待，直到指定时间内锁可用或者线程中断。当获得锁后返回true，否则返回false。
6. Void unlock():释放锁

获取到的锁必须释放，否则其他线程不能获取锁，一般代码格式：

lock.lock();  
try{  
 //do something  
}catch(InterruptedException ex){  
 //handle exception  
}finally{  
 lock.unlock();  
}

在同步方法和代码块（即synchronized）的上下文中，隐式的监听锁和每个对象相关联。所有的锁获取和释放都发生在块状结构中。当多个锁被获取，它们会以相反的顺序被释放，并且所有的锁都会在它们被获取的文本作用域中被释放掉（自动的）。

在接口Lock实现的上下文中，锁的获取和释放可以更加弹性。接口Lock的实现允许使用在不同的作用于中获取、释放锁遮掩的技术。同时以不同的顺序允许多个锁被获取和释放。

非多线程则是单线程，线程是最小执行单元，进程是分配资源的最小单元

块状锁的缺失（synchronized）移除了发生在同步方法和代码块上自动释放锁的功能，这时需要我们手动释放锁。

重入锁：ReentrantLock

描述了一个可重入的锁，这个锁和一个持有量相关联，当一条线程持有这个锁并且调用lock、lockUniterruptibly或者任意一个tryLock方法重新获取锁，这个持有量就递增1。当线程调用unlock方法，持有量就递减1，当持有量降为0，锁就会被释放掉。

重入锁提供了与通过同步方法或代码块得以访问的隐式监听锁（即synchronized）同样的并发即语义，即不需要手动释放，当抽有量为0时自动释放锁。

方法：

1. ReentrantLock():创建一个Reentrant的实例，该构造函数等价于ReentrantLock(false)
2. ReentrantLock(boolean fair):创建一个拥有指定公平策略的ReentrantLock的实例，当这个锁需要使用公平的排序策略时，给fair赋值为true，在争用的环境下，这个锁倾向于将访问权限分配给等待最久的线程。
3. Boolean isFair():返回公平策略
4. Boolean isHeldByCurrentThread()：方法在锁被当前线程持有的情况下，返回true。

public class LockDemo {  
 public static void main(String[] args) {  
 ExecutorService executorService= Executors.newFixedThreadPool(2);  
 final ReentrantLock lock=new ReentrantLock();  
  
 class Worker implements Runnable{  
 private String name;  
 public Worker(String name){  
 this.name=name;  
 }  
 @Override  
 public void run() {  
 lock.lock();  
 try{  
 if(lock.isHeldByCurrentThread()){  
 System.out.printf("Thread %s entered critical section.%n",name);  
 }  
  
 System.out.printf("Thread %s performing work.%n",name);  
 try{  
 Thread.sleep(2000);  
 }catch(InterruptedException ex){  
 ex.printStackTrace();  
 }  
 System.out.printf("Thread %s finished working.%n",name);  
 }finally{  
 lock.unlock();  
 }  
 }  
 }  
  
 executorService.execute(new Worker("ThreadA"));  
 executorService.execute(new Worker("ThreadB"));  
  
 try{  
 executorService.awaitTermination(5, TimeUnit.SECONDS);  
 }catch(InterruptedException ex){  
 ex.printStackTrace();  
 }  
  
 executorService.shutdownNow();  
 }  
}

当一条线程调用lock()方法而锁又不可用时，这个线程就会一直禁用（并且不能被调度），知道锁变为可用。

条件Condition

Lock取代了同步方法、代码块，Condition取代了Object的wait、notification方法。一个Condition的实例原本就会绑定到一个锁上，可以使用Lock的newCondition方法获取一个针对特定Lock实例的Condition实例。

Condition的方法：

1. Void await()：在接收到信号或者被中断之前，强制调用线程一直等待
2. Boolean await(long time,TimeUnit unit)：在接收到信号、被中断或者超过指定的等待时间之前，强制调用线程一直等待。
3. Long await(long nanosTimeout):
4. Void awaitUninterruptibly():在接收到信号之前，强制当前线程一直等待
5. Boolean awaitUntil(Date deadline):在接收到信号、被中断，或者超过指定的截止时间之前，强制当前线程一直等待。
6. Void signal():唤醒一个等待中的线程
7. Void signalAll()：唤醒所有等待中的线程

代码示例：

public class ConditionDemo {  
 public static void main(String[] args) {  
 Shared s=new Shared();  
 new Producer(s).start();  
 new Consumer(s).start();  
 }  
}  
  
class Shared{  
 private char c;  
 private volatile boolean available;  
 private final Lock lock;  
 private final Condition condition;  
 Shared(){  
 this.available=false;  
 lock=new ReentrantLock();  
 condition=lock.newCondition();  
 }  
  
 Lock getLock(){return this.lock;}  
 char getSharedChar(){  
 lock.lock();  
 try{  
 while(!available) {  
 try {  
 condition.await();  
 } catch (InterruptedException ex) {  
 ex.printStackTrace();  
 }  
 }  
 available = false;  
 condition.signal();  
 }finally {  
 lock.unlock();  
 return c;  
 }  
 }  
  
 void setSharedChar(char c){  
 lock.lock();  
 try{  
 while(available){  
 try{  
 condition.await();  
 }catch(InterruptedException ex){  
 ex.printStackTrace();  
 }  
 }  
  
 this.c=c;  
 available=true;  
 condition.signal();  
 }finally{  
 lock.unlock();  
 }  
 }  
}  
  
class Producer extends Thread{  
 private final Lock l;  
 private final Shared s;  
 Producer(Shared s){  
 this.s=s;  
 l=s.getLock();  
 }  
  
 @Override  
 public void run() {  
 for(char ch='A';ch<='Z';ch++){  
 l.lock();  
 s.setSharedChar(ch);  
 System.out.println(ch+" produced by producer");  
 l.unlock();  
 }  
 }  
}  
  
class Consumer extends Thread{  
 private final Lock l;  
 private final Shared s;  
 Consumer(Shared s){  
 this.s=s;  
 this.l=s.getLock();  
 }  
  
 @Override  
 public void run() {  
 char ch;  
 do{  
 l.lock();  
 ch=s.getSharedChar();  
 System.out.println(ch+" consumed by consumer.");  
 l.unlock();  
 }while(ch!='Z');  
 }  
}

final修饰的成员变量，jvm提供了可见性的机制，也可以使用volatile。

读写锁

读写锁适用于对数据结构频繁读而较少修改的场景，在读取时有更好的并发性，而写入时保证安全的互斥访问。这一机制基于接口ReadWriteLock

ReadWriteLock维护了一对锁，一个锁针对只读操作，一个锁针对写操作。在没有写操作的时候，读锁可能被多条线程同时持有，写入锁是互斥的，只有单个线程可以修改共享数据。

方法如下：

Lock readLock():返回用于读的锁

Lock writeLock():返回用于写的锁

重入读写锁

类ReentrantReadWriteLock实现了接口ReadWriteLock。构造函数有：

ReentrantReadWriteLock():该函数等价于ReentrantReadWriteLock(false)

ReentrantReadWriteLock(boolean fair):创建一个具有公平策略的ReentrantReadWriteLock的实例。当需要使用公平策略时将fair设置为true。基于公平的策略，若当前持有的锁被释放了，那要么是等待最久的单条写线程会被分配写锁，要么就是当一组读线程比所有等待中的写线程等待时间还长时，这组读线程会被分配锁。

方法：

ReentrantReadWriteLock.ReadLock readLock():返回用于读的锁

ReentrantReadWriteLock.WriteLock writeLock():返回用于写的锁

int getReadHoldCount():返回被调用线程在这个锁上持有的读锁的数量

int getWriteHoldCount():返回被调用线程在这个锁上持有的写锁的数量

匿名类访问的外部局部变量需要被定义成final的。

代码示例：

public class ReadWriteLockDemo {  
 public static void main(String[] args) {  
 final String[] words={"a","b","c","d","e"};  
 final String[] definitions={"aa","bb","cc","dd","ee"};  
 final Map<String,String> dictionary=new HashMap<String,String>();  
  
 ReadWriteLock rwl=new ReentrantReadWriteLock(true);  
 final Lock rlock=rwl.readLock();  
 final Lock wlock=rwl.writeLock();  
  
 Runnable writer=new Runnable() {  
 @Override  
 public void run() {  
 for (int i=0;i<words.length;i++){  
 wlock.lock();  
 try{  
 dictionary.put(words[i],definitions[i]);  
 System.out.println("writer storing "+words[i]+" entry");  
 }finally {  
 wlock.unlock();  
 }  
  
 try{  
 Thread.sleep(100);  
 }catch(InterruptedException ex){  
 System.out.println("writer interrupted");  
 }  
 }  
 }  
 };  
  
 ExecutorService es= Executors.newFixedThreadPool(1);  
 es.submit(writer);  
  
 Runnable reader=new Runnable() {  
 @Override  
 public void run() {  
 while(true){  
 rlock.lock();  
 try{  
 int i=(int)(Math.random()\*words.length);  
 System.out.println("reader accessing "+words[i]+" : "+dictionary.get(words[i])+" entry");  
 }finally {  
 rlock.unlock();  
 }  
  
 try{  
 Thread.sleep(1000);  
 }catch(InterruptedException ex){  
 System.out.println("reader interrupted");  
 }  
 }  
 }  
 };  
  
 es=Executors.newFixedThreadPool(1);  
 es.submit(reader);  
 }  
}

第八章 额外的开发工具类

开发集合

原子变量

Fork/join框架

Completion service