一．中断

每一个线程都有一个boolean类型标志，用来表明当前线程是否请求中断，当一个线程调用interrupt()方法时，线程的中断标志将被设置为true。

我们可以通过调用Thread.currentThread().isInterrupted()或者Thread.interrupted()来检测线程的中断标志是否被置位。这两个方法的区别是

Thread.currentThread().isInterrupted()是线程对象的方法，调用它后不清除线程中断标志位；而Thread.interrupted()是一个静态方法，调用它会清除

线程中断标志位。

Thread.currentThread().isInterrupted()：对象方法不清除中断标志位

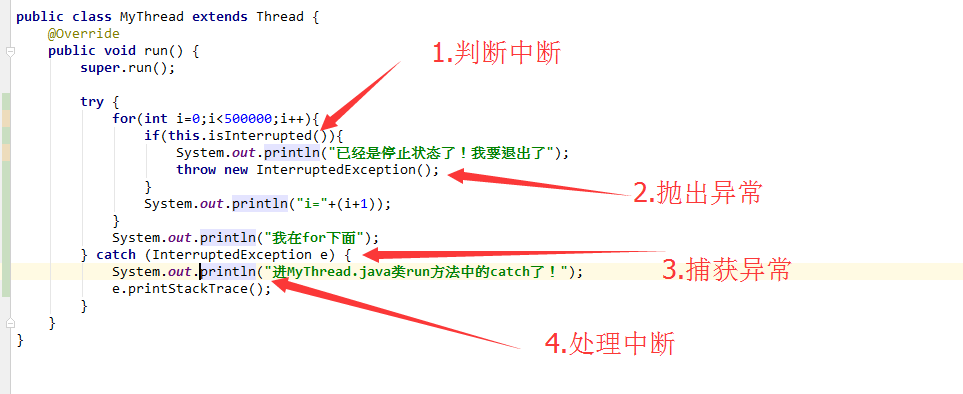
Thread.interrupted()：静态方法清除中断标志位(设置为false)

所以说调用线程的interrupt()方法不会中断一个正在运行的线程，这个机制只是设置了一个线程中断标志位，如果在程序中你不检测线程中断标志位，那么即使

设置了中断标志位为true，线程也一样照常运行。

1. 处理中断

处理方法是需要判断线程中断后再抛出异常，才能中断下面的代码（输出我在for下面），然后再catch这个异常，在catch里记录中断或者处理数据



1. 也可以用return处理中断（不执行后面的代码）

用return也能停止已经被中断的线程



不过还是建议使用“抛异常”的方法来实现线程的停止，因为在catch块中还可以将异常向上抛，使线程停止的事件得以传播。

1. 暂停与恢复线程

暂停线程暂停线程意味着此线程还可以恢复运行。在Java多线程中，可以使用suspend（）方法暂停线程，使用resume（）方法恢复线程的执行。

1. 优先级

高优先级的线程总是大部分先执行完，但不代表高优先级的线程全部先执行完。不要以为线程先被main线程所调用就会先执行完，出现这样的结果全是因为线程的优先级是最高值为10造成的。当线程优先级的等级差距很大时，谁先执行完和代码的调用顺序无关。

1. 多线程的随机性

结论，不要把线程的优先级与运行结果的顺序作为衡量的标准，优先级较高的线程并不一定每一次都先执行完run（）方法中的任务，也就是说，线程优先级与打印顺序无关，不要将这两者的关系相关联，它们的关系具有不确定性和随机性。

1. 守护线程

守护线程是一种特殊的线程，它的特性有“陪伴”的含义，当进程中不存在非守护线程了，则守护线程自动销毁。典型的守护线程就是垃圾回收线程，当进程中没有非守护线程了，则垃圾回收线程也就没有存在的必要了，自动销毁。用个比较通俗的比喻来解释一下“守护线程”：任何一个守护线程都是整个JVM中所有非守护线程的“保姆”，只要当前JVM实例中存在任何一个非守护线程没有结束，守护线程就在工作，只有当最后一个非守护线程结束时，守护线程才随着JVM一同结束工作。Daemon的作用是为其他线程的运行提供便利服务，守护线程最典型的应用就是GC（垃圾回收器），它就是一个很称职的守护者。

1. 变量的并发访问

方法中的变量不存在非线程安全问题，永远都是线程安全的。这是方法内部的变量是私有的特性造成的。

1. 方法前加上synchronized关键字

调用用关键字synchronized声明的方法一定是排队运行的。

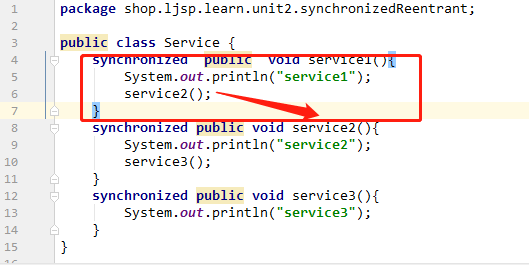
一个线程持有了某个对象的锁，但另一个对象完全可以异步调用非synchronized类型的方法。

总结一下：

1. 一个线程先持有一个对象的Lock锁，另一个线程可以以异步的方式调用对象中的非synchronized类型的方法。
2. 一个线程先持有一个对象的Lock锁，另一个线程如果在这时调用对象中的其它synchronized类型的方法则需等待，也就是同步。

十．可重入锁

“可重入锁”的概念是：自己可以再次获取自己的内部锁。比如有1条线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果不可锁重入的话，就会造成死锁。可重入锁也支持在父子类继承的环境中。当存在父子类继承关系时，子类是完全可以通过“可重入锁”调用父类的同步方法的。



Service1执行的时候再次调取Service的锁，如果没有可重入锁，那么service2将不能被调用，会一直锁死。

十一．出现异常，锁自动释放

当一个线程执行的代码出现异常时，其所持有的锁会自动释放。

1. 同步不可以继承

同步不能继承，所以还得在子类的方法中添加synchronized关键字，

十三．

　synchronized同步语句块用关键字synchronized声明方法在某些情况下是有弊端的，比如A线程调用同步方法执行一个长时间的任务，那么B线程则必须等待比较长时间。在这样的情况下可以使用synchronized同步语句块来解决。

当两个并发线程访问同一个对象object中的synchronized（this）同步代码块时，一段时间内只能有一个线程被执行，另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块。

当一个线程访问object的一个synchronized同步代码块时，另一个线程仍然可以访问该object对象中的非synchronized（this）同步代码块。

在使用同步synchronized（this）代码块时需要注意的是，当一个线程访问object的一个synchronized（this）同步代码块时，其他线程对同一个object中所有其他synchronized（this）同步代码块的访问将被阻塞，这说明synchronized使用的“对象监视器”是一个。

和synchronized方法一样，synchronized（this）代码块也是锁定当前对象的。

1. 总结

多个线程调用同一个对象中的不同名称的synchronized同步方法或synchronized（this）同步代码块时，调用的效果就是按顺序执行，也就是同步的，阻塞的。这说明synchronized同步方法或synchronized（this）同步代码块分别有两种作用。

## synchronized同步方法

1. 对其他synchronized同步方法或synchronized（this）同步代码块调用呈阻塞状态。

2）同一时间只有一个线程可以执行synchronized同步方法中的代码。

## （2）synchronized（this）同步代码块

1）对其他synchronized同步方法或synchronized（this）同步代码块调用呈阻塞状态。

2）同一时间只有一个线程可以执行synchronized（this）同步代码块中的代码。

其实Java还支持对“任意对象”作为“对象监视器”来实现同步的功能。这个“任意对象”大多数是实例变量及方法的参数

1）在多个线程持有“对象监视器”为同一个对象的前提下，同一时间只有一个线程可以执行synchronized（非this对象x）同步代码块中的代码。

2）当持有“对象监视器”为同一个对象的前提下，同一时间只有一个线程可以执行synchronized（非this对象x）同步代码块中的代码。

锁非this对象优点：

锁非this对象具有一定的优点：如果在一个类中有很多个synchronized方法，这时虽然能实现同步，但会受到阻塞，所以影响运行效率；但如果使用同步代码块锁非this对象，则synchronized（非this）代码块中的程序与同步方法是异步的，不与其他锁this同步方法争抢this锁，则可大大提高运行效率。

# 锁整个类

关键字synchronized还可以应用在static静态方法上，如果这样写，那是对当前的\*.java文件对应的Class类进行持锁

synchronized关键字加到static静态方法上是给Class类上锁，而synchronized关键字加到非static静态方法上是给对象上锁。这两个锁不是一个锁，会异步执行。

异步的原因是持有不同的锁，一个是对象锁，另外一个是Class锁，而Class锁可以对类的所有对象实例起作用。

Class中如果有静态方法：

synchronized（class）对class上锁后，其他线程只能以同步的方式调用class中的静态同步方法。

同步synchronized（class）代码块的作用其实和synchronizedstatic方法的作用一样。

例如：

packageservice;

publicclassService{

publicstaticvoidprintA(){

synchronized(Service.class){

｝

与

SynchronizedpublicstaticvoidprintA(){

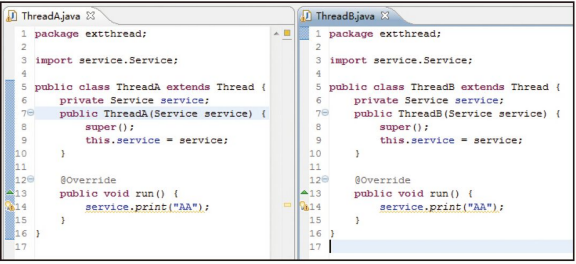
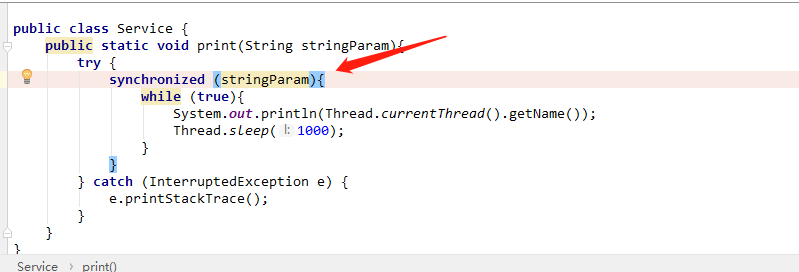
}

作用一样

在将任何数据类型作为同步锁时，需要注意的是，是否有多个线程同时持有锁对象，如果同时持有相同的锁对象，则这些线程之间就是同步的；如果分别获得锁对象，这些线程之间就是异步的。

1. 锁字符串（不推荐）

因为String的两个值都是AA，两个线程持有相同的锁，所以造成线程B不能执行。这就是String常量池所带来的问题。因此在大多数的情况下，同步synchronized代码块都不使用String作为锁对象，而改用其他，比如newObject（）实例化一个Object对象，但它并不放入缓存中。



十六．当一个变量被volatile修饰时，任何线程对它的写操作都会立即刷新到主内存中，并且会强制让缓存了该变量的线程中的数据清空，必须从主内存重新读取最新数据。

volatile修饰之后并不是让线程直接从主内存中获取数据，依然需要将变量拷贝到工作内存中。

关键字volatile的作用是强制从公共堆栈中取得变量的值，而不是从线程私有数据栈中取得变量的值。

使用volatile关键字增加了实例变量在多个线程之间的可见性。但volatile关键字最致命的缺点是**不支持原子性**。

十七．**关键字synchronized和volatile进行一下比较：**

1. 关键字volatile是线程同步的轻量级实现，所以volatile性能肯定比synchronized要好，并且volatile只能修饰于变量，而synchronized可以修饰方法，以及代码块。随着JDK新版本的发布，synchronized关键字在执行效率上得到很大提升，在开发中使用synchronized关键字的比率还是比较大的。
2. 多线程访问volatile不会发生阻塞，而synchronized会出现阻塞。
3. volatile能保证数据的可见性，但不能保证原子性；而synchronized可以保证原子性，也可以间接保证可见性，因为它会将私有内存和公共内存中的数据做同步。此知识点在后面有实验做论证。
4. 再次重申一下，关键字volatile解决的是变量在多个线程之间的可见性；而synchronized关键字解决的是多个线程之间访问资源的同步性。

关键字volatile虽然增加了实例变量在多个线程之间的可见性，但它却不具备同步性，那么也就不具备原子性。

线程安全包含原子性和可见性两个方面，Java的同步机制都是围绕这两个方面来确保线程安全的。

volatile本身并不处理数据的原子性，而是强制对数据的读写及时影响到主内存的。

1. 使用原子类解决volatile的线程不安全

除了在i++操作时使用synchronized关键字实现同步外，还可以使用AtomicInteger原子类进行实现。原子操作是不能分割的整体，没有其他线程能够中断或检查正在原子操作中的变量。一个原子（atomic）类型就是一个原子操作可用的类型，它可以在没有锁的情况下做到线程安全（thread-safe）。

PrivateAtomicIntegercount=newAtomicInteger(0);

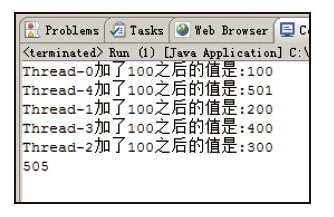
　原子类也并不完全安全原子类在具有有逻辑性的情况下输出结果也具有随机性。

publicvoidaddNum(){

System.out.println(Thread.currentThread().getName()+

"加了100之后的值是:"+aiRef.addAndGet(100));

aiRef.addAndGet(1);



需要给方法加上锁：

SynchronizedpublicvoidaddNum(){

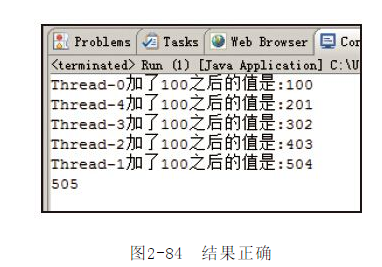
System.out.println(Thread.currentThread().getName()+

"加了100之后的值是:"+aiRef.addAndGet(100));

aiRef.addAndGet(1);

个人理解（需要求证）：

也就是说，一个方法用到两次原子类的操作时，存在第一个执行完后，其它线程在第二次执行前改变值，所以要给方法加上锁。



小技巧：

　synchronized代码块有volatile同步的功能关键字synchronized可以使多个线程访问同一个资源具有同步性，而且它还具有将线程工作内存中的私有变量与公共内存中的变量同步的功能，在此实验中进行验证。

关键字synchronized可以保证在同一时刻，只有一个线程可以执行某一个方法或某一个代码块。它包含两个特征：互斥性和可见性。同步synchronized不仅可以解决一个线程看到对象处于不一致的状态，还可以保证进入同步方法或者同步代码块的每个线程，都看到由同一个锁保护之前所有的修改效果。

十九.**第三章线程间通信**

## Wait()与notify()方法

方法**wait（）**的作用是使当前执行代码的线程进行等待，wait（）方法是Object类的方法，该方法用来将当前线程置入“预执行队列”中，并且在wait（）所在的代码行处停止执行，直到接到通知或被中断为止。在调用wait（）之前，线程必须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**。在执行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。如果调用wait（）时没有持有适当的锁，则抛出IllegalMonitorStateException，它是RuntimeException的一个子类，因此，不需要try-catch语句进行捕捉异常。

方法**notify（）也要在同步方法或同步块中调用**，即在调用前，线程也必须获得该对象的对象级别锁。如果调用notify（）时没有持有适当的锁，也会抛出IllegalMonitorStateException。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，**则由线程规划器随机挑选出其中一个呈wait状态的线程，对其发出通知notify**，并使它等待获取该对象的对象锁。需要说明的是，**在执行notify（）方法后，当前线程不会马上释放该对象锁，呈wait状态的线程也并不能马上获取该对象锁，要等到执行notify（）方法的线程将程序执行完，也就是退出synchronized代码块后，当前线程才会释放锁，而呈wait状态所在的线程才可以获取该对象锁。**当第一个获得了该对象锁的wait线程运行完毕以后，它会释放掉该对象锁，此时如果该对象没有再次使用notify语句，则即便该对象已经空闲，其他wait状态等待的线程由于没有得到该对象的通知，还会继续阻塞在wait状态，直到这个对象发出一个notify或notifyAll。

**重要：必须执行完notify（）方法所在的同步synchronized代码块后才释放锁。**

用一句话来总结一下wait和notify：wait使线程停止运行，而notify使停止的线程继续运行。

说明：

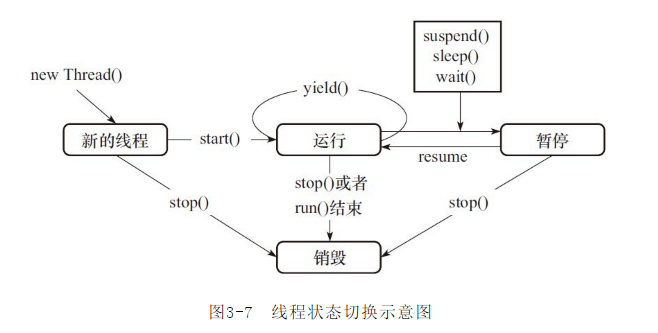
关键字synchronized可以将任何一个Object对象作为同步对象来看待，而Java为每个Object都实现了wait（）和notify（）方法，它们**必须用在被synchronized同步的Object的临界区内**。通过调用wait（）方法可以使处于临界区内的线程进入等待状态，同时释放被同步对象的锁。而notify操作可以唤醒一个因调用了wait操作而处于阻塞状态中的线程，使其进入就绪状态。被重新换醒的线程会试图重新获得临界区的控制权，也就是锁，并继续执行临界区内wait之后的代码。**如果发出notify操作时没有处于阻塞状态中的线程，那么该命令会被忽略。**

**wait（）**方法可以**使调用该方法的线程释放共享资源的锁，然后从运行状态退出，进入等待队列，直到被再次唤醒。**

**notify（）**方法**可以随机唤醒等待队列中等待同一共享资源的“一个”线程，并使该线程退出等待队列，进入可运行状态**，也就是notify（）方法仅通知“一个”线程。

**notifyAll（）**方法可以**使所有正在等待队列中等待同一共享资源的“全部”线程从等待状态退出，进入可运行状态。此时，优先级最高的那个线程最先执行，但也有可能是随机执行**，因为这要取决于JVM虚拟机的实现。

二十.线程生存周期说明



1. 新创建一个新的线程对象后，再调用它的start（）方法，系统会为此线程分配CPU资源，使其处于Runnable（可运行）状态，这是一个准备准备运行的阶段。如果线程抢占到CPU资源，此线程就处于Running（运行）状态。
2. Runnable状态和Running状态可相互切换，因为有可能线程运行一段时间后，有其他高优先级的线程抢占了CPU资源，这时此线程就从Running状态变成Runnable状态。线程进入Runnable状态大体分为如下5种情况：

·调用sleep（）方法后经过的时间超过了指定的休眠时间。

·线程调用的阻塞IO已经返回，阻塞方法执行完毕。

·线程成功地获得了试图同步的监视器。

·线程正在等待某个通知，其他线程发出了通知。

·处于挂起状态的线程调用了resume恢复方法。

1. Blocked是阻塞的意思，例如遇到了一个IO操作，此时CPU处于空空闲状态，可能会转而把CPU时间片分配给其他线程，这时也可以称为“暂停”状态。Blocked状态结束后，进入Runnable状态，等待系统重新分配资源。出现阻塞的情况大体分为如下5种：

·线程调用sleep方法，主动放弃占用的处理器资源。

·线程调用了阻塞式IO方法，在该方法返回前，该线程被阻塞。

·线程试图获得一个同步监视器，但该同步监视器正被其他线程所持有。

·线程等待某个通知。

·程序调用了suspend方法将该线程挂起。此方法容易导致死锁，尽量避免使用该方法。

4）run（）方法运行结束后进入销毁阶段，整个线程执行完毕。

**每个锁对象都有两个队列，一个是就绪队列，一个是阻塞队列。**就绪队列存储了将要获得锁的线程，阻塞队列存储了被阻塞的线程。**一个线程被唤醒后，才会进入就绪队列，等待CPU的调度；反之，一个线程被wait后，就会进入阻塞队列，等待下一次被唤醒。**

注意：

当线程呈wait（）状态时，调用线程对象的interrupt（）方法会出现InterruptedException异常。

二十一.总结：

通过上面的几个实验可以总结如下3点：

1. 执行完同步代码块就会释放对象的锁。
2. 在执行同步代码块的过程中，遇到异常而导致线程终止，锁也会被释放。
3. 在执行同步代码块的过程中，执行了锁所属对象的wait（）方法，这个线程会释放对象锁，而此线程对象会进入线程等待池中，等待被唤醒。

二十二.wait(long)超时自动唤醒

带一个参数的wait（long）方法的功能是等待某一时间内是否有线程对锁进行唤醒，如果超过这个时间则自动唤醒。

二十三.通过管道实现线程间通信

在Java语言中提供了各种各样的输入/输出流Stream，使我们能够很方便地对数据进行操作，其中管道流（pipeStream）是一种特殊的流，用于在不同线程间直接传送数据。一个线程发送数据到输出管道，另一个线程从输入管道中读数据。通过使用管道，实现不同线程间的通信，而无须借助于类似临时文件之类的东西。在Java的JDK中提供了4个类来使线程间可以进行通信：

1. PipedInputStream和PipedOutputStream

2）PipedReader和PipedWriter

二十四.线程的join方法

在很多情况下，主线程创建并启动子线程，如果子线程中要进行大量的耗时运算，主线程往往将早于子线程结束之前结束。这时，如果主线程想等待子线程执行完成之后再结束，比如子线程处理一个数据，主线程要取得这个数据中的值，就要用到join（）方法了。方法join（）的作用是等待线程对象销毁。

方法join的作用是使所属的线程对象x正常执行run（）方法中的任务，而使当前线程z进行无限期的阻塞，等待线程x销毁后再继续执行线程z后面的代码。方法join具有使线程排队运行的作用，有些类似同步的运行效果。join与synchronized的区别是：join在内部使用wait（）方法进行等待，而sychronized关键字使用的是“对象监视器”原理做为同步。

在join过程中，如果当前线程对象被中断，则当前线程出现异常。

二十五。Join(long)与sleep(long)区别

方法join（long）中的参数是设定等待的时间。

方法join（long）的功能在内部是使用wait（long）方法来实现的，所以join（long）方法具有释放锁的特点。

当执行wait（long）方法后，当前线程的锁被释放，那么其他线程就可以调用此线程中的同步方法了。

**而Thread.sleep（long）方法却不释放锁。**

**join（）或者join(long)方法释放锁的特点。**

二十六.Join总结

**在A线程中调用了B线程的join()方法时，表示只有当B线程执行完毕时，A线程才能继续执行。**

**join方法必须在线程start方法调用之后调用才有意义。这个也很容易理解：如果一个线程都没有start，那它也就无法同步了。**

**join方法的原理就是调用相应线程的wait方法进行等待操作的，例如A线程中调用了B线程的join方法，则相当于在A线程中调用了B线程的wait方法，当B线程执行完（或者到达等待时间），B线程会自动调用自身的notifyAll方法唤醒A线程，从而达到同步的目的。**

如果一个线程t1调用了Join方法的意思是使得调用t1的线程主动放弃cpu控制权，

也就是说，如果main线程中t1调用了join方法，main就放弃了控制权，等t1执行完再并行执行main线程与其它的线程，这一过程t1与main是串行执行的。

Join实现原理：

publicstaticvoidmain(String[]args)throwsInterruptedException{

ThreadJoinTestt1=newThreadJoinTest("小明");

ThreadJoinTestt2=newThreadJoinTest("小东");

t1.start();

t1.join();

t2.start();

/\*\*join的意思是使得放弃当前线程的执行，并返回对应的线程，例如下面代 码的意思就是：

程序在main线程中调用t1线程的join方法，则main线程放弃cpu控制权， 并返回t1线程继续执行直到线程t1执行完毕

所以结果是t1线程执行完后，才到主线程执行，相当于在main线程中同步 t1线程，t1执行完了，main线程才有执行的机会

\*/

}

二十七.线程私有数据

类ThreadLocal主要解决的就是每个线程绑定自己的值，可以将ThreadLocal类比喻成全局存放数据的盒子，盒子中可以存储每个线程的私有数据。

类Threadlocal解决的是变量在不同线程间的隔离性，也就是不同线程拥有自己的值，不同线程中的值是可以放入Threadlocal类中进行保存的。

使用类InheritableThreadLocal可以在子线程中取得父线程继承下来的值。

重写childValue（）方法可以在子线程中修改继承下来的值。

Package ext;

Import java.util.Date;

Public class InheritableThreadLocalExt extends Inheritable ThreadLocal{

@Override

protectedObjectinitialValue(){

Return new Date().getTime();

}

@Override

Protected Object child Value(Object parentValue){

Return parentValue+"我在子线程加的~!";

}

}

注意：如果子线程在取得值的同时，主线程将InheritableThreadLocal中的值进行更改，那么子线程取到的值还是旧值。

二十八.Lock的使用

Lock lock=new ReentrantLock();

调用lock.lock（）代码的线程就持有了“对象监视器”，其他线程只有等待锁被释放时再次争抢。效果和使用synchronized关键字一样，线程之间还是顺序执行的。

使用notify（）/notifyAll（）方法进行通知时，被通知的线程却是由JVM随机选择的。

使用ReentrantLock结合Condition类是可以实现“选择性通知”，这个功能是非常重要的，而且在Condition类中是默认提供的。

二十九.condition实现等待/通知

Object类中的wait（）方法相当于Condition类中的await（）方法。

Object类中的wait（long timeout）方法相当于Condition类中的await（longtime， Time Unitunit）方法。

Object类中的notify（）方法相当于Condition类中的signal（）方法。

Object类中的notifyAll（）方法相当于Condition类中的signalAll（）方法。