发件人：方堃 fangkun119@hotmail.com
主题：
日期：2021年3月4日 下午1:57
收件人：

CH21 多线程(B)：互斥机制、线程本地存储、终结线程任务

2016年9月16日 星期五
15:55

目录

1. 测试框架 【目录】

IntGenerator

- 线程使用IntGenerator接口，在IntGenerator的各种实现中添加不同的同步策略
- canceled是volatile boolean类型，因此具有原子性

int next();
void cancle();
boolean isCanceled()

```
public abstract class IntGenerator {
  private volatile boolean canceled = false;
  public abstract int next();
  // Allow this to be canceled:
  public void cancel() { canceled = true; }
  public boolean isCanceled() { return canceled; }
} ///:~
```

EvenChecker

- 线程：实现了Runnable接口，在main()/test()函数中交给ExecutorService，封装在线程中运行
- cancel()/isCanceled()：
  run()检查IntGenerator.isCanceled()，只要一个线程调用cancel()，其他线程就能感知
  一旦线程发现从next()取出来的数字不是偶数，就调用cancel()结束运行
- next()：多个线程通用同一个IntGenerator，调用next()的时候会产生并发访问d

```
//: concurrency/EvenChecker.java
import java.util.concurrent.*;

public class EvenChecker implements Runnable {
  private IntGenerator generator;
  private final int id;
  public EvenChecker(IntGenerator g, int ident) {
    generator = g;
    id = ident;
  }
  public void run() {
    while(!generator.isCanceled()) {
      int val = generator.next();
      if(val % 2 != 0) {
        System.out.println(val + " not even!");
        generator.cancel(); // Cancels all EvenCheckers
      }
    }
  }
  // Test any type of IntGenerator:
  public static void test(IntGenerator gp, int count) {
    System.out.println("Press Control-C to exit");
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < count; i++)
      exec.execute(new EvenChecker(gp, i));
    exec.shutdown();
  }
  // Default value for count:
  public static void test(IntGenerator gp) {
    test(gp, 10);
  }
} ///:~
```

```
//线程任务(runnable)
public class EvenCheckerTask implement Runnable {
 private IntGenerator evenGenerator;
 private final int id;
 //外部代码创建EvenChecker时，传入的都是同一个IntGenerator
 //这个IntGenerator都是用来生成偶数的
 public EvenChecker(IntGenerator gen, int runnableId) {
  evenGenerator = gen;
  id = runnableId;
 }
 public void run() {
  //多个线程共享同一个IntGenerator
  //这个Generator可以停止多个线程
  while (!evenGenerator.isCanceled()) {
   int value = evenGenerator.next();
   //如果synchronize能处理好，返回的都是偶数
   if (value % 2 != 0) {
    //检测到线程没处理好同步
    evenGenerator.cancel();
   }
  }
 }
 //test()启动多个线程，一边运行一边检查是否正确同步生成偶数
 …
```

```
//: concurrency/EvenGenerator.java
// When threads collide.

public class EvenGenerator extends IntGenerator {
  private int currentEvenValue = 0;
  public int next() {
    ++currentEvenValue; // Danger point here!
    ++currentEvenValue;
    return currentEvenValue;
  }
  public static void main(String[] args) {
    EvenChecker.test(new EvenGenerator());
  }
} /* Output: (Sample)
Press Control-C to exit
89476993 not even!
89476993 not even!
*///:~
```

```
不做同步处理的EvenGenerator
不能把两个++currentEvenValue封装在一个原子操作中
不能正确synchronize
```

2. 互斥机制： synchronize函数、显示锁、不该依赖隐式地原子操作或volatile、原子类、临界区 【目录】

synchronize函数

方法

synchronized

原理	• 所有synchronized方法、以及用synchronized(someObject)标记的代码段（临界区），共享同一把可重入的互斥锁 • 这把互斥锁是编辑器加在object中的（隐式对象锁）
要点	• 把需要synchronize的域声明为private • 所有访问这个域的函数（或代码段）都放在synchronized函数（或临界

<table>
<tr><td></td><td></td><td>
<table>
<tr><td></td><td>区）中
• synchronized方法或临界区可重入，同一线程可多次加锁，不会阻塞和死锁（不同线程加锁才会互斥）</td></tr>
<tr><td>synchronized static</td><td>synchronized static方法使用的互斥锁是编辑器加载.class对象中的，作用于整个类
Synchronized static可以在累的范围内防止对static域的并发访问</td></tr>
</table>
</td></tr>
<tr><td></td><td>例子</td><td>下面这个类放在测试框架中，可以正确处理线程同步的问题
<table>
<tr><td>

```
public class
SynchronizedEvenGenerator extends IntGenerator {
  private int currentEvenValue = 0;
  public synchronized int next() {
    ++currentEvenValue;
    Thread.yield(); // Cause failure faster
    ++currentEvenValue;
    return currentEvenValue;
  }
  public static void main(String[] args) {
    EvenChecker.test(new SynchronizedEvenGenerator());
  }
} ///:~
```

</td><td>

```
public class
SynchrobnizedEventGenerator extends IntGenerator {
 …
 //synchrnoizde保证两个currentEvenValue
 //被封装在一个原子操作中
 public synchronized int next() {
  ++currentEvenValue;
  Thread.yield(); //构造测试场景用
  ++currentEvenValue;
  return currentEvenValue;
 }
 …
}
```

</td></tr>
</table>
</td></tr>
<tr><td></td><td>备选</td><td>原子类一节给出另外一种基于原子类的实现，不需要使用互斥锁</td></tr>
<tr><td>显式Lock</td><td>包</td><td>
<table>
<tr><td>Java.util.concurrent.locks
Java SE5引入的技术，Lock必须备显式地创建、锁定、释放</td><td>Interface
Condition
Lock
ReadWriteLock</td><td>Class
AbstractOwnableSynchronizer
AbstractQueuedLongSynchronizer
AbstractQueuedSynchronizer
LockSupport
ReentrantLock
ReentrantReadWriteLock
ReentrantReadWriteLock.ReadLock
ReentrantReadWriteLock.WriteLock</td></tr>
</table>
</td></tr>
<tr><td></td><td>特点</td><td>
<table>
<tr><td>与内建在Object中的隐式互斥锁相比</td><td>（1）解决特定问题时更加灵活、强大
（2）同时代码也更加复杂、缺乏优雅性</td></tr>
</table>
</td></tr>
<tr><td></td><td>例子1</td><td>
<table>
<tr><td>

```
import java.util.concurrent.locks.*;

public class MutexEvenGenerator extends IntGenerator {
  private int currentEvenValue = 0;
  private Lock lock = new ReentrantLock();
  public int next() {
    lock.lock();
    try {
      ++currentEvenValue;
      Thread.yield(); // Cause failure faster
      ++currentEvenValue;
      return currentEvenValue;
    } finally {
      lock.unlock(); // 放在finally里面比较好
    }
  }
  public static void main(String[] args) {
    EvenChecker.test(new MutexEvenGenerator());
  }
} ///:~
```

</td><td>

```
public class MutexEvenGenerator extends IntGenerator {
 private int currentValue;
 //互斥锁
 private Lock reentrantLock = new ReentrantLock();
 public int next() {
  //加锁
  reentrantLock.lock();
  try {
   …t
   //先return再解锁
   return currentEvenValue;
  } finally {
   //解锁放在finally中保证其一定会执行
   reentrantLock.unlock();
  }
 }
 …
}
```

</td></tr>
</table>
</td></tr>
<tr><td></td><td>例子2</td><td>
<table>
<tr><td>

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class AttemptLocking {
  private ReentrantLock lock = new ReentrantLock();
  public void untimed() {
    boolean captured = lock.tryLock();
    try {
      System.out.println("tryLock(): " + captured);
    } finally {
      if(captured)
        lock.unlock();
    }
  }
```

</td><td>

```
private ReentrantLock reentrantLock
                = new ReentrantLock();
boolean isLocked = reentrantLock.tryLock();
try {
 ...
} finally {
 if (isLocked) {
  reentrantLock.unlock();
 }
}
```

</td></tr>
<tr><td>

```
 public void timed() {
   boolean captured = false;
   try {
     captured = lock.tryLock(2, TimeUnit.SECONDS);
   } catch(InterruptedException e) {
     throw new RuntimeException(e);
   }
   try {
     System.out.println("tryLock(2, TimeUnit.SECONDS): " +
       captured);
   } finally {
     if(captured)
       lock.unlock();
   }
 }
```

</td><td>

```
bool isLocked
     = reentrantLock.tryLock(2, TimeUnit.SECONDS);
try {
  ...
} finally {
 if (isLocked) {
  reentrantLock.unlock();
}
```

</td></tr>
<tr><td>

```
public static void main(String[] args) {
  final AttemptLocking al = new AttemptLocking();
  al.untimed(); // True -- lock is available
  al.timed();   // True -- lock is available
  // Now create a separate task to grab the lock:
  new Thread() {
    { setDaemon(true); }
    public void run() {
      al.lock.lock();
      System.out.println("acquired");
```

</td><td>显式地Lock可以使用tryLock接口
这一点synchronized做不到</td></tr>
</table>
</td></tr>
</table>

```
  }
}.start();
 Thread.yield(); // Give the 2nd task a chance
 al.untimed(); // False -- lock grabbed by task
 al.timed();   // False -- lock grabbed by task
 }
} /* Output:
tryLock(): true
tryLock(2, TimeUnit.SECONDS): true
acquired
tryLock(): false
tryLock(2, TimeUnit.SECONDS): false
*///:~
```

volatile与原子性没必然关系

内容

原子性	概念	不能被线程调度机制中断的操作 一旦操作开始，一定可以在可能发生“上下文切换”之前执行完毕
	基本类型	大部分是原子性的 但是long、double等会被某些JVM拆分成两个32位操作，加了volatile后才会具有原子性
	原则	依赖原子性棘手且非常危险，除非有并发专家（写JVM的那种）帮助、否则不建议依赖原子性 java.util.concurrent类库提供了更加精妙的构件，不需要依赖原子性
volatile	非volatile	多核处理器架构下,一个CPU对某个变量的修改，只是改在这个CPU的cache中，其他CPU是看不到的
	volatile	将变量声明为volatile，会强制变量的修改写到内存中，对所有CPU可见。
	原则	首选synchonized机制 仅当volatile是唯一需要同步的域,并且不依赖它之前的值时,才考虑 但仍不建议依赖volatile，P682例子，即使return_value()这样的函数，都无法保证线程安全

例1

用java -p 证明 a++; a+=2; 不是原子的

代码	java -p输出	说明

`//: concurrency/Atomicity.java`
`// (Exec: javap -c Atomicity)`
 	
`public class Atomicity {`
`  int i;`
`  void f1() { i++; }`
`  void f2() { i += 3; }`
`} /* Output: (Sample)`
`...`
	`void f1();`
	`  Code:`
	`  0:  aload_0`
	`  1:  dup`
	`  2:  getfield  #2; //Field i:I`
	`  5:  iconst_1`
	`  6:  iadd`
	`  7:  putfield  #2; //Field i:I`
	`  10:  return`
	
	`void f2();`
	`  Code:`
	`  0:  aload_0`
	`  1:  dup`
	`  2:  getfield  #2; //Field i:I`
	`  5:  iconst_3`
	`  6:  iadd`
	`  7:  putfield  #2; //Field i:I`
	`  10:  return`
	`*///:~`
		每条指令都会产生一条get，一条put，打破了原子性

例2

用实验证明不加锁的偶数生成函数 evenIncrement() 线程不安全

```
import java.util.concurrent.*;

public class AtomicityTest implements Runnable {
  private int i = 0;
  public int getValue() { return i; }
  private synchronized void evenIncrement() { i++; i++; }
  public void run() {
    while(true)
      evenIncrement();
  }
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    AtomicityTest at = new AtomicityTest();
    exec.execute(at);
    while(true) {
      int val = at.getValue();
      if(val % 2 != 0) {
        System.out.println(val);
        System.exit(0);
      }
    }
  }
} /* Output: (Sample)
```

```
191583767
*///:~
```

getValue()函数没加锁

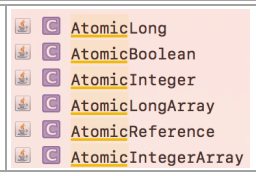
例子3

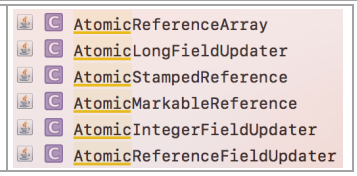
用实验证明 a++ 也是线程不安全的， 代码 P684a

原子类

内容

包名	java.util.concurrent.*
函数	andAndGet(...) compareAndSet(...)
说明	JavaSE5引入，为CAS操作(Compare And Set)提供原子保证，性能调优用





例子1

线程代码：用原子类替代synchronize

```
//: concurrency/AtomicIntegerTest.java
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;

public class AtomicIntegerTest implements Runnable {
  private AtomicInteger i = new AtomicInteger(0);
  public int getValue() { return i.get(); }
  private void evenIncrement() { i.addAndGet(2); }
  public void run() {
    while(true)
      evenIncrement();
  }
```

```
//工作线程
public class AtomicIntegerRunnable implements Runnable {
  private AtomicInteger atomInt = new AtomicInteger(0);
  public int getValue() {
    //用AtomicInteger包裹了需要同步的唯一数据
    return atomInt.get();
  }
  public void evenIncrement() {
    //用AtomicInteger包裹了需要同步的唯一数据
    atomInt.addAndGet(2);
  }
  public void run() {
    //一直运行直到程序exit()
    while(true)
      evenIncrement();
```

```
  }
}
```

```
public static void main(String[] args) {
  new Timer().schedule(new TimerTask() {
    public void run() {
      System.err.println("Aborting");
      System.exit(0);
    }
  }, 5000); // Terminate after 5 seconds
  ExecutorService exec = Executors.newCachedThreadPool();
  AtomicIntegerTest ait = new AtomicIntegerTest();
  exec.execute(ait);
  while(true) {
    int val = ait.getValue();
    if(val % 2 != 0) {
      System.out.println(val);
      System.exit(0);
    }
  }
} ///:~
```

```
//主线程
public static void main(String[] args) {
 //用Timer#schedule()函数设置定时任务(TimerTask)
 //5000ms后开始执行
 new Timer().schedule(new TimerTask() {
  public void run() {
    System.exit(0);
  }
 }, 5000);
 //启动整数+2线程
 ExecutorService execSvc
        = Executors.newCachedThreadPool();
 AtomicIntegerRunnable atomicIntegerRunnable
        = new AtomicIntegerRunnable()
 execSvc.execute(atomicIntegerRunnable);
 //检查,直到进程被TimerTask关闭,或发现错误自己退出
 while(true) {
  int value2check = atomicIntegerRunnable.getValue();
  if (val % 2 != 0) {
    System.out.println("error " + val + " not even");
    System.exit(0);
  }
 }
```

例子2

```
//: concurrency/AtomicEvenGenerator.java
// Atomic classes are occasionally useful in regular code.
// {RunByHand}
import java.util.concurrent.atomic.*;

public class AtomicEvenGenerator extends IntGenerator {
  private AtomicInteger currentEvenValue =
    new AtomicInteger(0);
  public int next() {
    return currentEvenValue.addAndGet(2);
  }
  public static void main(String[] args) {
    EvenChecker.test(new AtomicEvenGenerator());
  }
} ///:~
```

用原子类重写《synchronize函数》例子中的偶数生成器SynchronizedEvenGenerator，实现多线程同步

注意

应该强调的是，**Atomic类被设计用来构建java.util.concurrent中的类，因此只有在特殊情况下才在自己的代码中使用它们，即便使用了也需要确保不存在其他可能出现的问题。通常依赖于锁要更安全一些（要么是synchronized关键字，要么是显式的Lock对象）。**

临界区

概念

代码样式

```
synchronized(syncObject) {
  // This code can be accessed
  // by only one task at a time
}
```

用途

希望防止多线程同时访问代码中的某一段、而不是整个函数
相比synchronized方法，缩短了锁的粒度，提高了并发度、性能更好

原理

synchronized(objectToSync) {...}	进入临界区时,其实是隐式地获得objectToSync的对象锁
synchronized(this) {...}	获得当前对象的对象锁，可以与当前类的synchronized方法/临界区形成互斥
synchronized(otherObject) {...}	获得其他对象的对象锁，可以与其他类的synchronized方法/临界区形成互斥

例子1

内容

Pair的函数不是线程安全的,用PairManager(子类PairManager1, PairManager2)进行封装保护,做到线程安全
PairManager采用了模板方法设计模式，基类调用方法，子类提供increase()函数的两种不同实现(临界区及synchronized函数)

```
package concurrency;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;

class Pair { // Not thread-safe
  private int x, y;
  public Pair(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public Pair() { this(0, 0); }
  public int getX() { return x; }
  public int getY() { return y; }
  public void incrementX() { x++; }
  public void incrementY() { y++; }
  public String toString() {
    return "x: " + x + ", y: " + y;
  }
  public class PairValuesNotEqualException
  extends RuntimeException {
    public PairValuesNotEqualException() {
      super("Pair values not equal: " + Pair.this);
    }
  }
  // Arbitrary invariant -- both variables must be equal:
  public void checkState() {
    if(x != y)
      throw new PairValuesNotEqualException();
  }
}
```

类Pair线程不安全

getX(),incrementX()	不能互斥
getY(),incrementY()	不能互斥
incrementX(), incrementY()	不在一个原子操作中,要求x,y取值同步时会有问题

```
//测试用,假定使用场景要求x,y取值相等,不相等时通过抛异常来通知测试框架
public void checkState() {
  if (x != y) {
    throw new PairValuesNotEqualException();
  }
}
```

```
// Protect a Pair inside a thread-safe class:
abstract class PairManager {
  AtomicInteger checkCounter = new AtomicInteger(0);
  protected Pair p = new Pair();
```

```
//PairManager通过封装Pair, 来以线程安全的方式使用Pair
//这里是模板方法设计模式的基类,封装调用increment()的框架
//方法increment()的实现留给子类
```

```
  private List<Pair> storage =
    Collections.synchronizedList(new ArrayList<Pair>());
  public synchronized Pair getPair() {
    // Make a copy to keep the original safe:
    return new Pair(p.getX(), p.getY());
  }
  // Assume this is a time consuming operation
  protected void store(Pair p) {
    storage.add(p);
    try {
      TimeUnit.MILLISECONDS.sleep(50);
    } catch(InterruptedException ignore) {}
  }
  public abstract void increment();
}
```

```
abstract class PairManager {
 //测试用,记录当前是第几次check
 AtomicInteger atomicCheckCounter = new AtomicInteger(0);
 //Pair pair需要用同步机制进行保护
 protected Pair pair = new Pair();
 private List<Pair> storage
  = Collections.synchronizedList(new ArrayList<Pair>());
 //对Pair pair的读操作已经用synchronized保护
 //接下来要测试increment()对Pair的保护
 public synchronized Pair getPair() {
  return new Pair(pair.getX(), pair.gexY();
 }
 //store不需要synchronize保护,为了体现临界区对性能的提升,
 //使用sleep()将其实现为一个耗时操作
 protocted void store(Pair pair) {
  storage.add(pair);
  try {
   TimeUnit.MILLISECONDS.sleep(50);
  } catch (InterruptedException ignore) {}
 }
 //对Pair的写操作,留给子类实现
 public abstract void increment();
}
```

```
// Synchronize the entire method:
class PairManager1 extends PairManager {
  public synchronized void increment() {
    p.incrementX();
    p.incrementY();
    store(getPair());
  }
}
```

```
//模板方法的具体实现1,用synchronized来保护increment()
class PairManager1 extends PairManger {
 public synchronized void increment() {
  p.incrementX(); //x, y的自增被封装在一个原子操作中
  p.incrementY();
  store(getPair());//其实不需要同步保护,而且很耗时
 }
}
```

```
// Use a critical section:
class PairManager2 extends PairManager {
  public void increment() {
    Pair temp;
    synchronized(this) {
      p.incrementX();
      p.incrementY();
      temp = getPair();
    }
    store(temp);
  }
}
```

②sync

```
//模板方法的具体实现1,用synchronized来保护increment()
class PairManager2 extends PairManager {
 public void increment() {
  Pair temp;
  synchronized(this) { //只互斥必须保护的操作
   pair.incrementX();
   pair.incrementY();
   temp = getPair();
  }
  store(temp); //store的计算开销可以并发
 }
}
```

```
class PairManipulator implements Runnable {
  private PairManager pm;
  public PairManipulator(PairManager pm) {
    this.pm = pm;
  }
  public void run() {
    while(true)
      pm.increment();
  }
  public String toString() {
    return "Pair: " + pm.getPair() +
      " checkCounter = " + pm.checkCounter.get();
  }
}
```

```
//测试线程1: 操作传入的PairManager,不断调用其increment()直到进程结束
class PairManipulator implements Runnable {
 private PairManager pairManager;
 private PairManipulator(PairManager pairManager) {
  this.pairManager = pairManager;
 }
 public void run() v{
  while (true) {
   pairManager.increment(); //调用要求suynchronize的操作
  }
 }
 ...
}
```

```
class PairChecker implements Runnable {
  private PairManager pm;
  public PairChecker(PairManager pm) {
    this.pm = pm;
  }
  public void run() {
    while(true) {
      pm.checkCounter.incrementAndGet();
      pm.getPair().checkState();
    }
  }
}
```

```
//测试线程2: 操作传入的PairManager,不断调用其checkState看是否发生了x,y取值不相等的情况
class PairChecker implements Runnable {
 private PairManager pairManager;
 public PairChecker(PairManager pairManager) {
  this.pairManager = pairManager;
 }
 public void run() {
  while(true) {
   //更新计数器: 第n次检查
   pairManager.atomicCheckCounter.incrementAndGet();
   //检查x,y取值是否一致
   pair.getPair().checkState();
  }
 }
}
```

```
public class CriticalSection {
  // Test the two different approaches:
  static void
  testApproaches(PairManager pman1, PairManager pman2) {
    ExecutorService exec = Executors.newCachedThreadPool();
    PairManipulator
      pm1 = new PairManipulator(pman1),
      pm2 = new PairManipulator(pman2);
    PairChecker
      pcheck1 = new PairChecker(pman1),
      pcheck2 = new PairChecker(pman2);
    exec.execute(pm1);
    exec.execute(pm2);
    exec.execute(pcheck1);
    exec.execute(pcheck2);
    try {
      TimeUnit.MILLISECONDS.sleep(500);
    } catch(InterruptedException e) {
```

```
//两个PairManager, 前者用子类PairManager1的实例, 后者用子类PaiurManager2的实例
static void test(
 PairManager pairManager1, PairManager pairManager2){
  //线程池
  ExecutorService execSvc
   = Executors.newCachedThreadPool();
  //操作pairManager1的两个Runnable,它两之间应当互斥
  //检查PairManager1的syncronized increment()对Pair的保护
  PairManipulator pm1 = new PairManipulator(pairManager1);
  PairChecker pc1 = new PairChecker(pairManager1);
  //操作pairManager2的两个Runnable,它两之间应当互斥
  //检查PairManager2的increment()临界区对Pair的保护
```

```
    System.out.println("pm1: " + pm1 + "\npm2: " + pm2);
    System.exit(0);
  }
  public static void main(String[] args) {
    PairManager
      pman1 = new PairManager1(),
      pman2 = new PairManager2();
    testApproaches(pman1, pman2);
  }
} /* Output: (Sample)
pm1: Pair: x: 15, y: 15 checkCounter = 272565
pm2: Pair: x: 16, y: 16 checkCounter = 3956974
*///:~
```

```
  PairManipulator pm2 = new PairManipulator(pairManager2);
  PairChecker pc2 = new PairChecker(pairManager2);
  //启动4个线程
  execSvc.execute(pm1);
  execSvc.execute(pc1);
  execSvc.execute(pm2);
  execSvc.execute(pc2);
  ...
 }
 ...
}
```

例子2

用显式的Lock也能做到互斥，下面再提供两个PairManager的子类（ExplictPairManager1，ExplictPairManager2），提供与例子1中PairManager1，PairManager2相同的功能（其实有bug，因为加的锁与synchronized加的隐式锁不是同一把）

```
package concurrency;
import java.util.concurrent.locks.*;

// Synchronize the entire method:
class ExplicitPairManager1 extends PairManager {
  private Lock lock = new ReentrantLock();
  public synchronized void increment() {
    lock.lock();
    try {
      p.incrementX();
      p.incrementY();
      store(getPair());
    } finally {
      lock.unlock();
    }
  }
}
```

```
// Use a critical section:
class ExplicitPairManager2 extends PairManager {
  private Lock lock = new ReentrantLock();
  public void increment() {
    Pair temp;
    lock.lock();
    try {
      p.incrementX();
      p.incrementY();
      temp = getPair();
    } finally {
      lock.unlock();
    }
    store(temp);
  }
}
```

例子3

演示在同一个类中synchronized(otherObject)，与synchronized方法并不互斥

```
//: concurrency/SyncObject.java
// Synchronizing on another object.
import static net.mindview.util.Print.*;

class DualSynch {
  private Object syncObject = new Object();
  public synchronized void f() {
    for(int i = 0; i < 5; i++) {
      print("f()");
      Thread.yield();
    }
  }
```

public synchronize void f()
获得DualSynch类对象的隐式锁之后
连续打印5个f()

```
  public void g() {
    synchronized(syncObject) {
      for(int i = 0; i < 5; i++) {
        print("g()");
        Thread.yield();
      }
    }
  }
}
```

```
public void g() {
  //获得syncObject的隐式锁之后连续打印5个g()
  suynchronized(syncObject) {
    ...
  }
}
```

```
public class SyncObject {
  public static void main(String[] args) {
    final DualSynch ds = new DualSynch();
    new Thread() {
      public void run() {
        ds.f();
      }
    }.start();
    ds.g();
  }
} /* Output: (Sample)
```

```
g()
f()
g()
f()
g()
f()
g()
f()
g()
f()
*///:~
```

f()和g()并发运行,没有发生互斥

3.线程本地存储 【目录】

内容

写法

```
//虽然是static,但由于类型是ThreadLocal<T>因此每个线程指向的对象都不一样,是相互独立的
//用继承ThreadLocal<Integer>的内部类来实现,需要编写initialValue()函数
private static ThreadLocal<Integer> value = new ThreadLocal<Integer>() {
   //synchriobnized?????
   protected synchronized Integer initialValue(){
      return new Integer(1);
   }
}
```

类

java.lang.ThreadLocal

例子

```
public class ThreadLocalVariableHolder {
  private static ThreadLocal<Integer> value =
    new ThreadLocal<Integer>() {
      private Random rand = new Random(47);
      protected synchronized Integer initialValue() {
        return rand.nextInt(10000);
      }
    };
  public static void increment() {
    value.set(value.get() + 1);
  }
  public static int get() { return value.get(); }
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(new Accessor(i));
```

```
//封装线程本地变量封装
public class ThreadLocalVarHolder {
  //唯一变量,继承ThreadLocal<Integer>的内部类
  private static ThreadLocal<Integer> threadLocalInteger
    = new ThreadLocal<Integer>() {
      private Random rand = new Random(47);
      protected synchronized Integer initialValue() {
        return rand.nextInt(10000);
      }
    };
  //操作线程本地变量的函数
```

```
    TimeUnit.SECONDS.sleep(3);  // Run for a while
    exec.shutdownNow();         // All Accessors will quit
  }
} /* Output: (Sample)
```

```
  public static void increment() {
    threadLocalInteger.set(value.get() + 1);
  }
  ..
}
```

```
//: concurrency/ThreadLocalVariableHolder.java
// Automatically giving each thread its own storage.
import java.util.concurrent.*;
import java.util.*;

class Accessor implements Runnable {
  private final int id;
  public Accessor(int idn) { id = idn; }
  public void run() {
    while(!Thread.currentThread().isInterrupted()) {
      ThreadLocalVariableHolder.increment();
      System.out.println(this);
      Thread.yield();
    }
  }
  public String toString() {
    return "#" + id + ": " +
      ThreadLocalVariableHolder.get();
  }
}
```

```
//线程
class VisitThreadLocalRunnable implements Runnable {
  private final int threadId;
  public VisitThreadLocxalRunnable(int id) { threaID = id; }
  public void run() {
    //Thread.currentThread()获得当前线程
    //isInterrupted()是否被中断
    while (!Thread.currentThread().isInterrupted()) {
      //封装了线程本地变量,更新
      ThreadLocalVariableHolder.increment();
      //打印线程本地变量
      System.out.println(
        threadId + ":" ThreadLocalVariablerHolder.get());
      //尽量让线程交替运行
      Thread.yield();
    }
```

```
#0: 9259
#1: 556
#2: 6694
#3: 1862
#4: 962
#0: 9260
#1: 557
#2: 6695
#3: 1863
#4: 963
...
*///:~
```

#1 556，557，558,...
#2 6694, 6695, 6696,...
两个线程各打印各的,互不影响

4. 终止线程 【目录】

标志位方法	内容	标志位终止法	如上节中的例子，提供cancel()函数，调用时可设置一个所有线程都可见的变量 所有线程都可以通过isCancel()来访问这个变量，读到true时，处理完当前的请求，执行清理工作，优雅退出
		问题	某些情况下，例如线程被某个操作阻塞了，等待阻塞解除的时间太长，必须做更多的处理来让程序立即终止 下一节在阻塞时终结演示如何解决这个问题，让程序尽快走到检查isCancel()这一步

例子

计数器

```
//: concurrency/OrnamentalGarden.java
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Count {
  private int count = 0;
  private Random rand = new Random(47);
  // Remove the synchronized keyword to see counting fail:
  public synchronized int increment() {
    int temp = count;
    if(rand.nextBoolean()) // Yield half the time
      Thread.yield();
    return (count = ++temp);
  }
  public synchronized int value() { return count; }
}
```

```
//读操作用synchronized保护
pubic synchronized int increment() {
  int temp = count;
  if(rand.nextBoolean())
    Thread.yield(); //构造一个不同步的状态
  return (count = ++temp);
}
//写操作用synchron ized保护
public synchronized int value() {
  return count;
}
```

入口

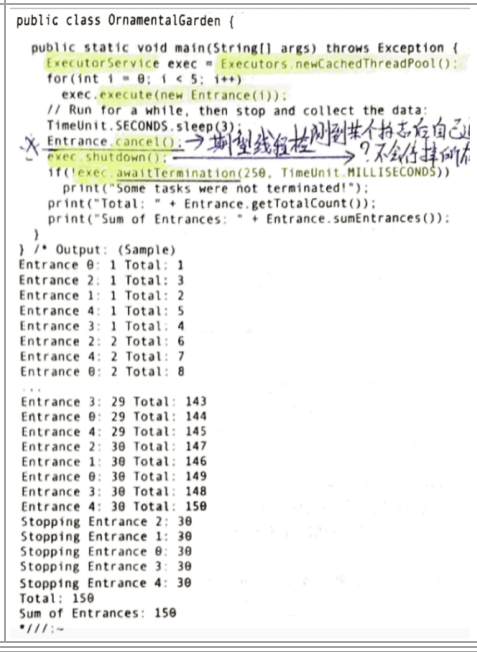
```
class Entrance implements Runnable {
  private static Count count = new Count();
  private static List<Entrance> entrances =
    new ArrayList<Entrance>();
  private int number = 0;
  // Doesn't need synchronization to read:
  private final int id;
  private static volatile boolean canceled = false;
  // Atomic operation on a volatile field:
  public static void cancel() { canceled = true; }
  public Entrance(int id) {
    this.id = id;
    // Keep this task in a list. Also prevents
    // garbage collection of dead tasks:
    entrances.add(this);
  }
  public void run() {
    while(!canceled) {
      synchronized(this) {
        ++number;
      }
      print(this + " Total: " + count.increment());
      try {
        TimeUnit.MILLISECONDS.sleep(100);
      } catch(InterruptedException e) {
        print("sleep interrupted");
      }
    }
    print("Stopping " + this);
  }
  public synchronized int getValue() { return number; }
  public String toString() {
    return "Entrance " + id + ": " + getValue();
  }
  public static int getTotalCount() {
    return count.value();
  }
  public static int sumEntrances() {
    int sum = 0;
    for(Entrance entrance : entrances)
```

```
//每个公园入口用一个线程来模拟
class Entrance implements Runnable {
  //全局所有线程总共循环了多少次
  private static Count count = new Count();
  //多个线程共享标志位
    //用volatile让标志位修改对所有CPU可见
  private static volatile boolean canceled = false;
  public static void cancel() {canceled = true; }
  //把所有的Entrance都集中在一个list中
  //防止先退出的线程被垃圾回收
  private static List<Entrance>
          entrances = new ArrayList<Entrance>();
  private int number = 0;
  private final int id;
  public Entrance(int id) {
    this.id = id;
    entrances.add(this);
  }
  //写number变量时需要synchronized
  public void run() {
    while (!canceled) {
      synchronized(this) {
        ++number;
      }
      //count自己有同步机制
      print(this + " Total: " + count.increment();
      try {
        //要阻塞比较长时间,会影响标志位的读取
        TimeUnit.MILLISECONDS.sleep(100);
      } catch (InterruptedException e) {
        print ("sleep interrupted");
```

```
    sum += entrance.getValue();
    return sum;
  }
}
```

```
  }
 }
}
//读number变量时也需要synchronized
public synchronized int getValue() {
  return number;
}
//getTotalCount()起校验用,给测试函数double check
public srtatic int getTotalCount() {...}
...
}
```

公园



```
public class OrnamentalGarden {
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(new Entrance(i));
    // Run for a while, then stop and collect the data:
    TimeUnit.SECONDS.sleep(3);
    Entrance.cancel();
    exec.shutdown();
    if(!exec.awaitTermination(250, TimeUnit.MILLISECONDS))
      print("Some tasks were not terminated!");
    print("Total: " + Entrance.getTotalCount());
    print("Sum of Entrances: " + Entrance.sumEntrances());
  }
} /* Output: (Sample)
Entrance 0: 1 Total: 1
Entrance 2: 1 Total: 3
Entrance 1: 1 Total: 2
Entrance 4: 1 Total: 5
Entrance 3: 1 Total: 4
Entrance 2: 2 Total: 6
Entrance 4: 2 Total: 7
Entrance 0: 2 Total: 8
...
Entrance 3: 29 Total: 143
Entrance 0: 29 Total: 144
Entrance 4: 29 Total: 145
Entrance 2: 30 Total: 147
Entrance 1: 30 Total: 146
Entrance 0: 30 Total: 149
Entrance 3: 30 Total: 148
Entrance 4: 30 Total: 150
Stopping Entrance 2: 30
Stopping Entrance 1: 30
Stopping Entrance 0: 30
Stopping Entrance 3: 30
Stopping Entrance 4: 30
Total: 150
Sum of Entrances: 150
*///:~
```

```
public static void main(String[] args) throws Exception {
 //创建线程, 并让线程运行一段时间
 ExecutorService exec = Executors.newCachedThreadPool();
 for (int i = 0; i < 5; ++i)
  exec.execute(new Entrance(i));
 TimeUnit.SECONDS.sleep(3);
 //设置线程结束标志位,标志位时static变量,所有线程都能读到
 Entrance.cancel();
 //不接受新的线程创建
 exec.shutdown();
 //等待所有线程都运行完毕
 if (!exec.awaitTermination(250, TimerUnit.MILLISECONDS))
  print("Some tasks were not erminated!");
 ...
}
```

在阻塞时终结

[标志位方法](#)的问题是，如果线程被阻塞，那么就不会去检查标志位，本节解决这问题

四种线程状态

新建	短暂处于该状态	1) 新建（new）：当线程被创建时，它只会短暂地处于这种状态。此时它已经分配了必需的系统资源，并执行了初始化。此刻线程已经有资格获得CPU时间了，之后调度器将把这个线程转变为可运行状态或阻塞状态。
就绪	获得CPU时间片就能运行	2) 就绪（Runnable）：在这种状态下，只要调度器把时间片分配给线程，线程就可以运行。也就是说，在任意时刻，线程可以运行也可以不运行。只要调度器能分配时间片给线程，它就可以运行，这不同于死亡和阻塞状态。
阻塞	等待某个条件满足后，才会能够获得CPU时间片	3) 阻塞（Blocked）：线程能够运行，但有某个条件阻止它的运行。当线程处于阻塞状态时，调度器将忽略线程，不会分配给线程任何CPU时间。直到线程重新进入了就绪状态，它才有可能执行操作。
死亡	通常是run()返回，不会再获得CPU时间片	4) 死亡（Dead）：处于死亡或终止状态的线程将不再是可调度的，并且再也不会得到CPU时间，它的任务已结束，或不再是可运行的。任务死亡的通常方式是从run()方法返回，但是任务的线程还可以被中断。你将要看到这一点。

如何阻塞

sleep	线程调用各种sleep()函数时，会在指定时间内被阻塞
wait	线程调用wait()会阻塞，直到得到了notify()/notifyAll()（或JavaSE5的signal()、signalAll()消息,来自java.util.concurrent包）
I/O	线程阻塞在某个I/O操作上
互斥	等待互斥锁时，线程被阻塞
[X]suspend	[X]suspend()可阻塞线程，直到被resume()唤醒 （这两个方法已废弃）
[X]stop	[X]stop()可阻塞线程（该方法也被废弃，因为不会释放已经获得的互斥锁）

注意，不是所有的阻塞类型都不能被中断，参考[哪些阻塞类型能中断](#)

中断线程风险

线程被中断时，可能还有资源需要清理，会比较棘手；而“[标志位方法](#)”优雅结束比较安全
在Java语言的设计中，线程中断更被当做一种异常（InterruptedException）来处理，以迫使更外层的开发者思考如何设计代码逻辑

如何中断

直接操作线程

方法	线程A在线程B阻塞时调用threadB.interrupt() ，threadB: 1.在run()函数中捕捉到InterruptedException同时结束阻塞(没阻塞时是否也会捕捉到异常?) 2.进入中断处理代码中调用interrupted()重置中断标志 3.结束当前循环进入下一轮，这样有机会去调用isCancel()来检测标志位并优雅结束（参考[标志位方法](#)）	
函数	interrupt()	someThread.interrupt(); //其他线程发送中断给someThread
	InterruptedException	线程执行的代码(如sleep()等)抛出Interrupted时,线程的run()函数可捕捉到该异常
	interrupted()	给被中断的线程(或中断处理代码)使用,在了解到是否被中断的同时,将中断标志复位(也就是线程被中断时,这个函数只有一次机会返回true,之后标记复位只能返回false)
	isInterrupted()	只检查是否被中断，不重置中断标记

用Executors托管线程

中断所有线程	myExecutorService.shutdownNow(); //不是shutdown()
中断制定线	1.线程任务需要通过implements Callable<RstType>来实现，而不是Runnable

	程	2.用submit()提交Callable<RstType>给myExecutorService，submit(...)返回Future<RstType> 因为不打算调用myFuture<RstType>.get()，因此把Future<RstType>赋值给Future<?>也没关系 3.调用myFuture.cancel()就可以中断线程
		Modifier and Type / **Method and Description** `boolean` — `cancel(boolean mayInterruptIfRunning)` Attempts to cancel execution of this task.

可中断/不可中断阻塞	判断方法	调用一个JavaSE API, 如果这个API声明了会抛出InterruptException，就说明可被中断；否则说明不可被中断	
		SleepBlock	因为sleep()被阻塞的线程，可以被中断
		IOBlock	因为IO被阻塞的线程，不可被中断（不会抛出Interrupetedexception）,NIO的支持好一些
		SynchronizedBlock	因为等待同步机制（如互斥锁）被阻塞的线程，不可被中断
		参考例1：SleepBlock阻塞可以被中断、IOBlock、SynchronizedBlock不可被中断	
	不可中断函数处理	笨拙但有效的方法	关闭引发阻塞的底层资源、从而解除阻塞。参考关闭底层资源解除线程的阻塞
		IOBlock	使用NIO可以让线程接收到中断信号，同时也能区分出是否是底层资源关闭引发，更好管理中断，参考用NIO管理线程中断
		互斥锁	1.同一个线程可以多次获得互斥锁而不被阻塞 2.ReentrantLock提供可被中断的互斥api
	中断状态检查	中断异常捕捉结合中断状态检查	中断信号仅在进程执行阻塞操作时才能触发异常，因此只使用异常检测并不靠谱，还需要结合异常状态检查

例子1

SleepBlock阻塞可以被中断、IOBlock、SynchronizedBlock不可被中断

SleepBlock

```
//: concurrency/Interrupting.java
// Interrupting a blocked thread.
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

class SleepBlocked implements Runnable {
  public void run() {
    try {
      TimeUnit.SECONDS.sleep(100);
    } catch(InterruptedException e) {
      print("InterruptedException");
    }
    print("Exiting SleepBlocked.run()");
  }
```

```
class SleepBloackedRunnable implements Runnable {
  public void run() {
   try {
    //Sleep阻塞
    TimeUnit.SECONDS.sleep(100);
   } catch (InterruptedException e) {
    //被主线程中断这个线程时,捕捉到异常,走到这里
    print ("InterruptedException");
   }
  }
}
```

IOBlock

```
class IOBlocked implements Runnable {
  private InputStream in;
  public IOBlocked(InputStream is) { in = is; }
  public void run() {
    try {
      print("Waiting for read():");
      in.read();
    } catch(IOException e) {
      if(Thread.currentThread().isInterrupted()) {
        print("Interrupted from blocked I/O");
      } else {
        throw new RuntimeException(e);
      }
    }
    print("Exiting IOBlocked.run()");
  }
}
```

```
class IOBlockedRunnable implements Runnable {
  ...
  public void run() {
   try {
    //阻塞在读操作,该函数不抛出InterruptedException
    inputStream.read();
   } catch(IOException e) {
    if (Thread.currentThread().isInterrupted()) {
     //主线程虽发出中断,但不抛InterruptedException
     //走不到这一句
     print("Interrupted from blocked I/O");
    } else {
     throw new RuntimeException(e);
    }
   }
  }
}
```

SynchronizedBlock

```
class SynchronizedBlocked implements Runnable {
  public synchronized void f() {
    while(true) // Never releases lock
      Thread.yield();
  }
  public SynchronizedBlocked() {
    new Thread() {
      public void run() {
        f(); // Lock acquired by this thread
      }
    }.start();
  }
  public void run() {
    print("Trying to call f()");
    f();
    print("Exiting SynchronizedBlocked.run()");
  }
}
```

构造函数启动一个线程，占用互斥锁
run()函数获取互斥锁时，锁已经被占有，阻塞在互斥锁上
此时虽然主线程发出了中断，但阻塞无法解除，"Existing Synchronized Blocked" 永远都没有打印

main

```
public class Interrupting {
  private static ExecutorService exec =
    Executors.newCachedThreadPool();
  static void test(Runnable r) throws InterruptedException{
    Future<?> f = exec.submit(r);
    TimeUnit.MILLISECONDS.sleep(100);
    print("Interrupting " + r.getClass().getName());
    f.cancel(true); // Interrupts if running
    print("Interrupt sent to " + r.getClass().getName());
  }
  public static void main(String[] args) throws Exception {
    test(new SleepBlocked());
    test(new IOBlocked(System.in));
    test(new SynchronizedBlocked());
    TimeUnit.SECONDS.sleep(3);
    print("Aborting with System.exit(0)");
    System.exit(0); // ... since last 2 interrupts failed
  }
} /* Output: (95% match)
 Interrupting SleepBlocked
 InterruptedException
 Exiting SleepBlocked.run()
 Interrupt sent to SleepBlocked
 Waiting for read():
 Interrupting IOBlocked
 Interrupt sent to IOBlocked
 Trying to call f()
 Interrupting SynchronizedBlocked
```

调用3次test(Runnable r):
* 每次test启动一个线程
  static void test(Runnable r) throws ... {
* 线程用 Future<?> f = exec.submit(r)启动
  启动后在主线程的3个test函数中：
  调用 f.cancel(true); 来发出中断信号
* 三个线程:
  SleepBlocked: 被 f.cancel(true) 中断、解除阻塞
  IOBlocked：线程被一直阻塞，直到进程退出
  SynchronizedBlocked：线程一直被阻塞，知道线程退出

```
Interrupt sent to SynchronizedBlocked
Aborting with System.exit(0)
*///:~
```

例子2

关闭底层资源解除线程的IO阻塞(对于IOBlock、SynchronizedBlock）（笨拙但有时确实有效）（P689 NIO会更用好些）

```
//: concurrency/CloseResource.java
// Interrupting a blocked task by
// closing the underlying resource.
// (RunByHand)
import java.net.*;
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

public class CloseResource {
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    ServerSocket server = new ServerSocket(8080);
    InputStream socketInput =
      new Socket("localhost", 8080).getInputStream();
    exec.execute(new IOBlocked(socketInput));
    exec.execute(new IOBlocked(System.in));
    TimeUnit.MILLISECONDS.sleep(100);
    print("Shutting down all threads");
    exec.shutdownNow();
    TimeUnit.SECONDS.sleep(1);
    print("Closing " + socketInput.getClass().getName());
    socketInput.close(); // Releases blocked thread
    TimeUnit.SECONDS.sleep(1);
    print("Closing " + System.in.getClass().getName());
    System.in.close(); // Releases blocked thread
  }
} /* Output: (85% match)
```

```
//InputStream from socket
ServerSocket server = new ServerSocket(8880);
InputStream socketInput
        = new Socket("localhost", 8080).getInputStream();
//两个线程，一个阻塞在Socket上，一个阻塞在STDIN上
ExecutorService exec = Executors.newCachedThreadPool();
exec.execute(new IOBlocked(socketInput));
exec.execute(new IOBlocked(System.in));
//主线程通过shutdownNow向线程发出中断
TimeUnit.MILLISECONDS.sleep(100);
exec.shutdownNow();
//等待一秒钟，两个线程并没有收到中断，仍然阻塞在IO操作
TimeUnit.SECONDS.sleep(1);
//关闭底层资源,两个线程的IO阻塞调用被终止,线程结束
socketInput.close();
TimeUnit.SECONDS.sleep(1);
System.in.close();
TimeUnit.SECONDS.sleep(1);
```

```
Waiting for read():
Waiting for read():
Shutting down all threads
Closing java.net.SocketInputStream
Interrupted from blocked I/O
Exiting IOBlocked.run()
Closing java.io.BufferedInputStream
Exiting IOBlocked.run()
*///:~
```

NIO管理线程中断

例子：使用NIO更好地管理中断，不仅是，也是

```
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

class NIOBlocked implements Runnable {
  private final SocketChannel sc;
  public NIOBlocked(SocketChannel sc) { this.sc = sc; }
  public void run() {
    try {
      print("Waiting for read() in " + this);
      sc.read(ByteBuffer.allocate(1));
    } catch(ClosedByInterruptException e) {
      print("ClosedByInterruptException");
    } catch(AsynchronousCloseException e) {
      print("AsynchronousCloseException");
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    print("Exiting NIOBlocked.run() " + this);
  }
}
```

```
class NIOBlocked implements Runnable {
  private final SocketChannel socketChannel;
  public NIOBlocked(SocketChannel sc) {
    this.socketChannel = sc;
  }
  public void run() {
    try {
      //阻塞在NIO read
      socketChannel.read(ByteBuffer.allocate(1));
    } catch (ClosedByInterruptException e) {
      //中断引发的异常
      ..
    } catch (AsynchronousCloseException e) {
      //底层资源关闭引发的Exception
      ..
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

```
public class NIOInterruption {
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    ServerSocket server = new ServerSocket(8080);
    InetSocketAddress isa =
      new InetSocketAddress("localhost", 8080);
    SocketChannel sc1 = SocketChannel.open(isa);
    SocketChannel sc2 = SocketChannel.open(isa);
    Future<?> f = exec.submit(new NIOBlocked(sc1));
    exec.execute(new NIOBlocked(sc2));
    exec.shutdown();
    TimeUnit.SECONDS.sleep(1);
    // Produce an interrupt via cancel:
    f.cancel(true);
    TimeUnit.SECONDS.sleep(1);
    // Release the block by closing the channel:
    sc2.close();
  }
} /* Output: (Sample)
Waiting for read() in NIOBlocked@7a84e4
Waiting for read() in NIOBlocked@15c7850
ClosedByInterruptException
Exiting NIOBlocked.run() NIOBlocked@15c7850
AsynchronousCloseException
Exiting NIOBlocked.run() NIOBlocked@7a84e4
*///:~
```

```
//两个SocketChannel
ServerSocket server = new ServerSocket(8080);
InetSocketSocketAddress sockAddr =
  new InetSocketSocketAddress("localhost", 8080);
SocketChannel sockChannel1 = SocketChannel.open(sockAddr);
SocketChannel sockChannel2 = SocketChannel.open(sockAddr);
//线程1,发出中断,NIO抛出ClosedByInterruptedException结束阻塞
ExecutorService exec = Executors.newCachedThreadPool();
Future<?> future = exec.submit(new NIOBlocked(sockChannel1));
TimeUnit.SECONDS.sleep(1);
future.cancel(true); //发出中断信号给线程1
//线程2,不中断,进程退出时因AsynchronousCloseException结束阻塞
exec.execute(new NIOBlocked(sockChannel2));
TimeUnit.SECONDS.sleep(1);
sc2.close();
```

同一个线程可以多次获得互斥锁而不被阻塞

代码

```
//: concurrency/MultiLock.java
// One thread can reacquire the same lock.
import static net.mindview.util.Print.*;

public class MultiLock {
  public synchronized void f1(int count) {
    if(count-- > 0) {
      print("f1() calling f2() with count " + count);
      f2(count);
    }
  }
  public synchronized void f2(int count) {
    if(count-- > 0) {
      print("f2() calling f1() with count " + count);
      f1(count);
```

main以及输出：f1()->f2()->f1()->f2()

```
f1() calling f2() with count 9
f2() calling f1() with count 8
f1() calling f2() with count 7
f2() calling f1() with count 6
f1() calling f2() with count 5
f2() calling f1() with count 4
f1() calling f2() with count 3
f2() calling f1() with count 2
f1() calling f2() with count 1
f2() calling f1() with count 0
*///:~
```

f1(),f2()都是synchronized方法，获取编译器加在Object中的隐式

<table>
<tr><td></td><td>

```
  }
  public static void main(String[] args) throws Exception {
    final MultiLock multiLock = new MultiLock();
    new Thread() {
      public void run() {
        multiLock.f1(10);
      }
    }.start();
  }
} /* Output:
```

</td><td>锁

只在同一个线程中调用f1()、f1()调用f2()。
因为隐式锁时可重入锁(reentrant lock)，同一个线程多次加锁不会死锁，因此代码没有被阻塞

备注：显式的ReentrantLock类提供lockInterruptly()容许通过线程中断来解锁，隐式锁没有此功能</td></tr>
<tr><td>ReentrantLock提供可被中断的互斥api</td><td colspan="2">调用ReentrantLock的lockInterruptibly()来获取互斥锁时，被阻塞期间如果收到InterruptedException，会解除阻塞
<table>
<tr><td>代码</td><td>启动一个线程，并中断它</td></tr>
<tr><td>

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import static net.mindview.util.Print.*;

class BlockedMutex {
  private Lock lock = new ReentrantLock();
  public BlockedMutex() {
    // Acquire it right away, to demonstrate interruption
    // of a task blocked on a ReentrantLock:
    lock.lock();
  }
  public void f() {
    try {
      // This will never be available to a second task
      lock.lockInterruptibly(); // Special call
      print("lock acquired in f()");
    } catch(InterruptedException e) {
      print("Interrupted from lock acquisition in f()");
    }
  }
}
```

</td><td>

```
class BlockedMutex {
 //可重入锁
 private Lock reentrantLock = new ReentrantLock();
 //构造函数在主线程中被调用,以不可中断方式加锁
 public BlockedMutex() {
  reentrantLock .lock();
 }
 //f()在工作线程中被调用，用可中断的方式加锁
 public void f() {
  try {
   //加锁时以为锁已被占用，阻塞
   reentrantLock.lockInterruptibly();
  } catch (InterruptedException e) {
   //lockIntrruptibly()可被中断，中断时走到这里
   ...
  }
 }
}
```

</td></tr>
<tr><td>

```
class Blocked2 implements Runnable {
  BlockedMutex blocked = new BlockedMutex();
  public void run() {
    print("Waiting for f() in BlockedMutex");
    blocked.f();
    print("Broken out of blocked call");
  }
}

public class Interrupting2 {
  public static void main(String[] args) throws Exception {
    Thread t = new Thread(new Blocked2());
    t.start();
    TimeUnit.SECONDS.sleep(1);
    System.out.println("Issuing t.interrupt()");
    t.interrupt();
  }
} /* Output:
Waiting for f() in BlockedMutex
Issuing t.interrupt()
Interrupted from lock acquisition in f()
Broken out of blocked call
*///:~
```

</td><td>

```
//工作线程
class BlockedRunnable2 implements Runnable {
 //在主线程中被调用并加锁
 BlockedMutex blockedMutex = new BlockedMutex();
 public void run() {
  //在工作线程中被调用,再次尝试加锁,被阻塞
  blockedMutex.f();
 }
}
//主线程
Thread thread = new Thread(new BlockedRunnable2());
thread.start();
TimeUnit.SECONDS.sleep(1);
thread.interrupt(); //中断工作线程
```

</td></tr>
</table></td></tr>
<tr><td>中断异常捕捉结合中断状态检查</td><td colspan="2"><table>
<tr><td>问题</td><td>

- catch(InterruptedException e/*或其他异常类*/)并不总是靠得住
- 因为收到中断信号时/之后，只有线程执行阻塞操作调用，异常才会被触发

</td></tr>
<tr><td>解决</td><td>

- 将中断状态检查（while(!Thread.interrupted())）与捕捉中断信号（catch）

</td></tr>
<tr><td>例子</td><td><table>
<tr><td>

```
class Blocked3 implements Runnable {
  private volatile double d = 0.0;
  public void run() {
    try {
      while(!Thread.interrupted()) {
        // point1
        NeedsCleanup n1 = new NeedsCleanup(1);
        // Start try-finally immediately after definition
        // of n1, to guarantee proper cleanup of n1:
        try {
          print("Sleeping");
          TimeUnit.SECONDS.sleep(1);
          // point2
          NeedsCleanup n2 = new NeedsCleanup(2);
          // Guarantee proper cleanup of n2:
          try {
            print("Calculating");
            // A time-consuming, non-blocking operation:
            for(int i = 1; i < 2500000; i++)
              d = d + (Math.PI + Math.E) / d;
            print("Finished time-consuming operation");
          } finally {
            n2.cleanup();
          }
        } finally {
          n1.cleanup();
        }
      }
      print("Exiting via while() test");
    } catch(InterruptedException e) {
      print("Exiting via InterruptedException");
    }
  }
}
```

</td><td>

```
class Blocked3 implements Runnable {
  private volatile double doubleVar = 0.0;
  public void run() {
    try {//try0
      //发生在sleep()之外的中断，不触发异常，正常执行到while处
      //正常释放resource
      while(!Thread.interrupted()) {
        //发生在point1 - point2之间的中断，
        //只有阻塞在sleep()时才会抛出异常，finally1块被执行
        //point1
        ResourceNeedCleanup
            resource1 = new ResourceNeedCleanup(1);
        try { //try1
          TimeUnit.SEONCDS.sleep(1);
          //point2
          ResourceNeedCleanup resource2
            = new ResourceNeedCleanup(2);
          try {//try2
            for (int i = 1; i < 2500000; ++i)
              doubleVar
              = doubleVar + (Math.PI + Math.E) / doubleVar;
          } finally {//finall2
            resource2.cleanup();
          }
        } finally {//finall1
          resource1.cleanup();
        }
      }
    } catch (InterruptedException e) {
      print ("exiting via InterruptException");
    }
  }
}
```

</td></tr>
<tr><td>

```
import java.util.concurrent.*;
import static net.mindview.util.Print.*;
```

</td><td>//主线程</td></tr>
</table></td></tr>
</table></td></tr>
</table>

```
class NeedsCleanup {
  private final int id;
  public NeedsCleanup(int ident) {
    id = ident;
    print("NeedsCleanup " + id);
  }
  public void cleanup() {
    print("Cleaning up " + id);
  }
}
public class InterruptingIdiom {
  public static void main(String[] args) throws Exception {
    if(args.length != 1) {
      print("usage: java InterruptingIdiom delay-in-mS");
      System.exit(1);
    }
    Thread t = new Thread(new Blocked3());
    t.start();
    TimeUnit.MILLISECONDS.sleep(new Integer(args[0]));
    t.interrupt();
  }
} /* Output: (Sample)
NeedsCleanup 1
Sleeping
NeedsCleanup 2
Calculating
Finished time-consuming operation
Cleaning up 2
Cleaning up 1
NeedsCleanup 1
Sleeping
Cleaning up 1
Exiting via InterruptedException
*///:~
```

Thread thread = new Thread(new BlockedRunnable3());
thread.start();
TimeUnit.MILLISECONDS.sleep(new Integer(args[0]));
thread.ingterrupt(); //中断单个线程