第30章 多线程和并行程序设计

目 录

contents

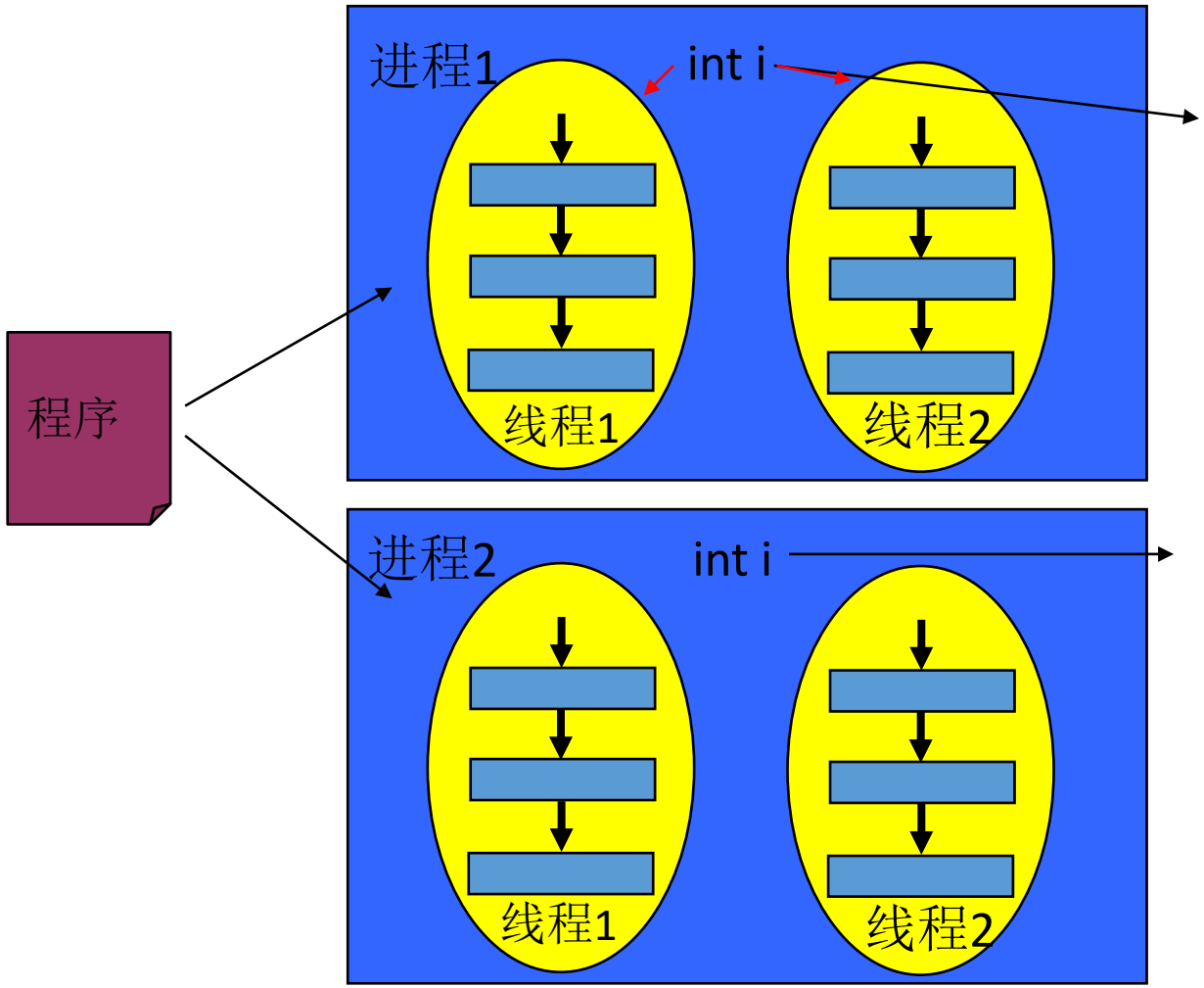
30.1 线程的概念

程序、进程和线程

1. “程序”代表一个静态的对象，是内含指令和数据的文件，存储在磁盘或其他存储设备中
2. “进程”代表一个动态的对象，是程序的一个执行过程，存在于系统的内存中。一个进程对应于一个程序
3. “线程”是运行于某个进程中，用于完成某个具体任务的顺序控制流程，有时被称为轻型进程。

30.1 线程的概念

程序、进程和线程



但是一个进程里的线程切换开销小的多，因为它们位于同一内存空间里。线程1、2线程位于同一内存空间使得线程之间数据交换非常容易。变量i可以被线程1、2访问（但要考虑同步）。因此线程又叫轻量级进程

不同进程的内存空间是隔离的，因此进程1中的变量i与进程2中的变量i属于不同的内存空间。进程切换和进程间通信开销大。进程间交换数据只能通过：共享内存、管道、消息队列、Socket通信等机制

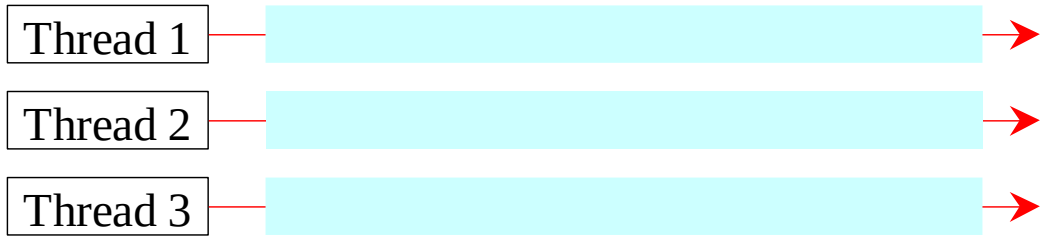
30.1 线程的概念

程序、进程和线程

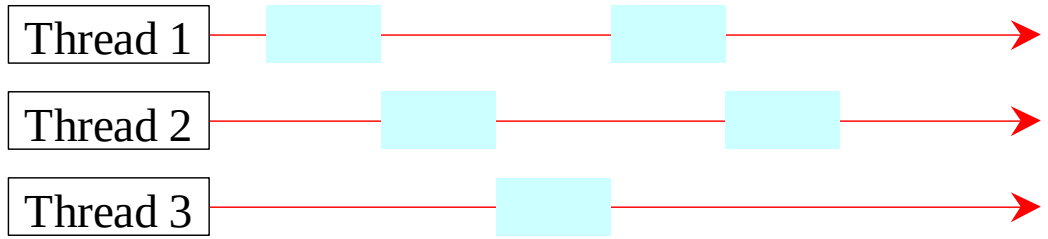
当一个进程被创建，自动地创建了一个主线程。因此，一个进程至少有一个主线程。

线程：程序中完成一个任务的有始有终的执行流，都有一个执行的起点，经过一系列指令后到达终点。

多CPU上运行的多线程



多线程共享一个CPU：某时刻，只能有一个线程在使用CPU



现代OS都将线程作为最小调度单位，进程作为资源分配的最小单位。分配给进程的资源（如文件，外设）可以被进程里的线程使用

30.1 线程的概念

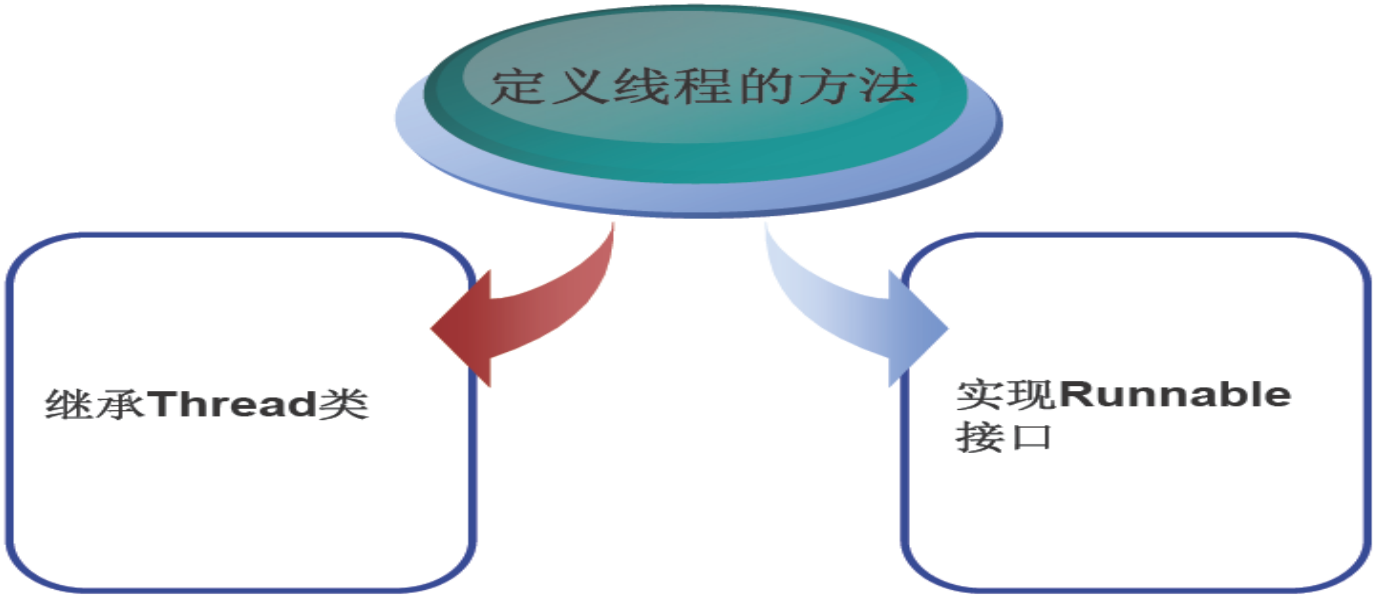
线程的作用

- 一个进程的多个子线程可以并发运行
- 多线程可以使程序反应更快、交互性更强、执行效率更高。
- 特别是Server端的程序，都是需要启动多个线程来处理大量来自客户端的请求
- 一个典型的GUI程序分为
  - GUI线程：处理UI消息循环，如鼠标消息、键盘消息
  - Worker线程：后台的数据处理工作，比如打印文件，大数据量的运算

30.2 Runnable接口和线程类Thread

创建线程方法：线程的执行逻辑（后面叫线程任务）必须实现java.lang.Runnable接口的唯一run方法。此外，由于Thread实现了Runnable接口，也可以通过Thread派生线程类。

因此有两种方法可以实现同一个或多个线程的运行：(1) 定义Thread类的子类并覆盖run方法；(2) 实现接口Runnable的run方法。



30.2 Runuable接口和线程类Thread

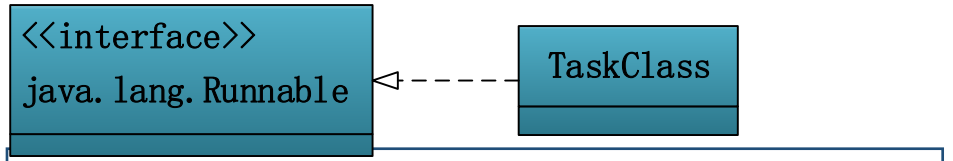
1: 通过实现Runnable接口创建线程

- 实现Runnable接口，需要实现唯一的接口方法run
  - void run( )
  - 该方法定义了线程执行的功能
- 创建实现Runnable接口的类的对象
- 利用Thread类的构造函数创建线程对象
  - public Thread (Runnable target)
- 通过线程对象的start()方法启动线程

new Thread对象时需要传入Runnable接口实例

30.2 Runnable接口和线程类Thread

1: 通过实现Runnable接口创建线程




```
//Custom task class
class TaskClass implements Runnable {
  ...//可以有自己的数据成员
  public TaskClass(...) {
    ...
  }

  //Implement the run method in Runnable
  public void run() {
    //Tell system how to run custom thread
    ...
  }
}
```

```
//Client Class
public class Client {
  ...
  public void someMethod(...) {
  ...
  // Create an instance of TaskClass
  Runnable task = new TaskClass(...);

  // Create a thread
  Thread thread = new Thread(task);

  // Start a thread
  thread.start(); // 启动后自动执行task.run
  }
}
```

1. 通过线程任务类（TaskClass）创建任务对象（task）
2. 以任务对象task为参数new Thread对象。Thread对象代表一个线程，线程的执行内容由任务对象task定义。
3. 通过线程对象thread启动线程thread.start()，任何线程只能启动一次，多次调用产生IllegalThreadStateException异常。

30.2 Runuable接口和线程类Thread

1: 通过实现Runnable接口创建线程的例子

- 程序创建并运行二个线程：第一个线程打印100次字母a；第二个线程打印100次字母b

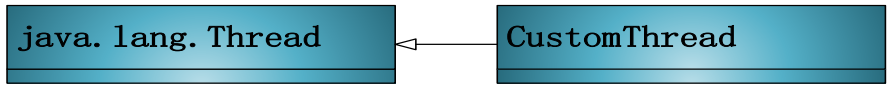
```
class PrintChar implements Runnable //实现Runnable接口
{
    private char charToPrint;  // The character to print
    private int times;  // The times to repeat
    public PrintChar(char c, int t){  charToPrint = c;  times = t; }
    public void run(){ //实现Runnable中声明的run方法
       for (int i=1; i < times; i++) System.out.print(charToPrint);
    }
}
```

new Thread对象时需要传入Runnable接口实例

```
public class RunnableDemo
{
   public static void main(String[] args){
      //以PrintChar对象实例为参数构造Thread对象
      Thread printA = new Thread(new PrintChar('a',100));
      Thread printB = new Thread(new PrintChar('b',100));
      printA.start();
      printB.start();
   }
}
```

30.2 Runuable接口和线程类Thread

2: 通过继承Thread类创建线程




```
// Custom thread class
class CustomThread extends Thread {
 //数据成员
 public CustomThread(...) {
   ...
 }

 public void run() {
  // Tell system how to perform this task
   ...
 }
 ...
}
```

```
//Client class
public class Client {
 ...
 public void someMethod() {
  ...
  // Create a thread
  Thread thread1 = new CustomThread();
  // Start thread
  thread1.start( ); //激活thread1对象的run

  // Create a thread
  Thread thread2 = new CustomThread( );
  // Start thread
  thread2.start(); //激活thread2对象的run
 }
 ...
}
```

1. 定义Thread类的扩展类（CustomThread）
2. 通过扩展类（CustomThread）创建线程对象（thread）
3. 通过线程对象thread启动线程thread.start()

线程和线程任务混在一起，不建议使用

Java不支持多继承，CustomThread继承了Thread类不能再继承其他类

30.2 Runuable接口和线程类Thread

2: 通过继承Thread类创建线程

```
class PrintChar extends Thread //继承Thread类
{
        private char charToPrint; //要打印的字符
        private int times; //打印的次数
        public PrintChar(char c, int t){ charToPrint = c; times = t; }
        public void run() {//覆盖run方法，定义线程要完成的功能
                for (int i=1; i < times; i++)
                        System.out.print(charToPrint);
        }
}
public class ThreadDemo {
        public static void main(String[] args) {
          Thread printA = new PrintChar('a',100); //创建二个线程对象
          Thread printB = new PrintChar('b',100);
          printA.start(); //启动线程
          printB.start();       //启动另外一个线程
        }
}
```

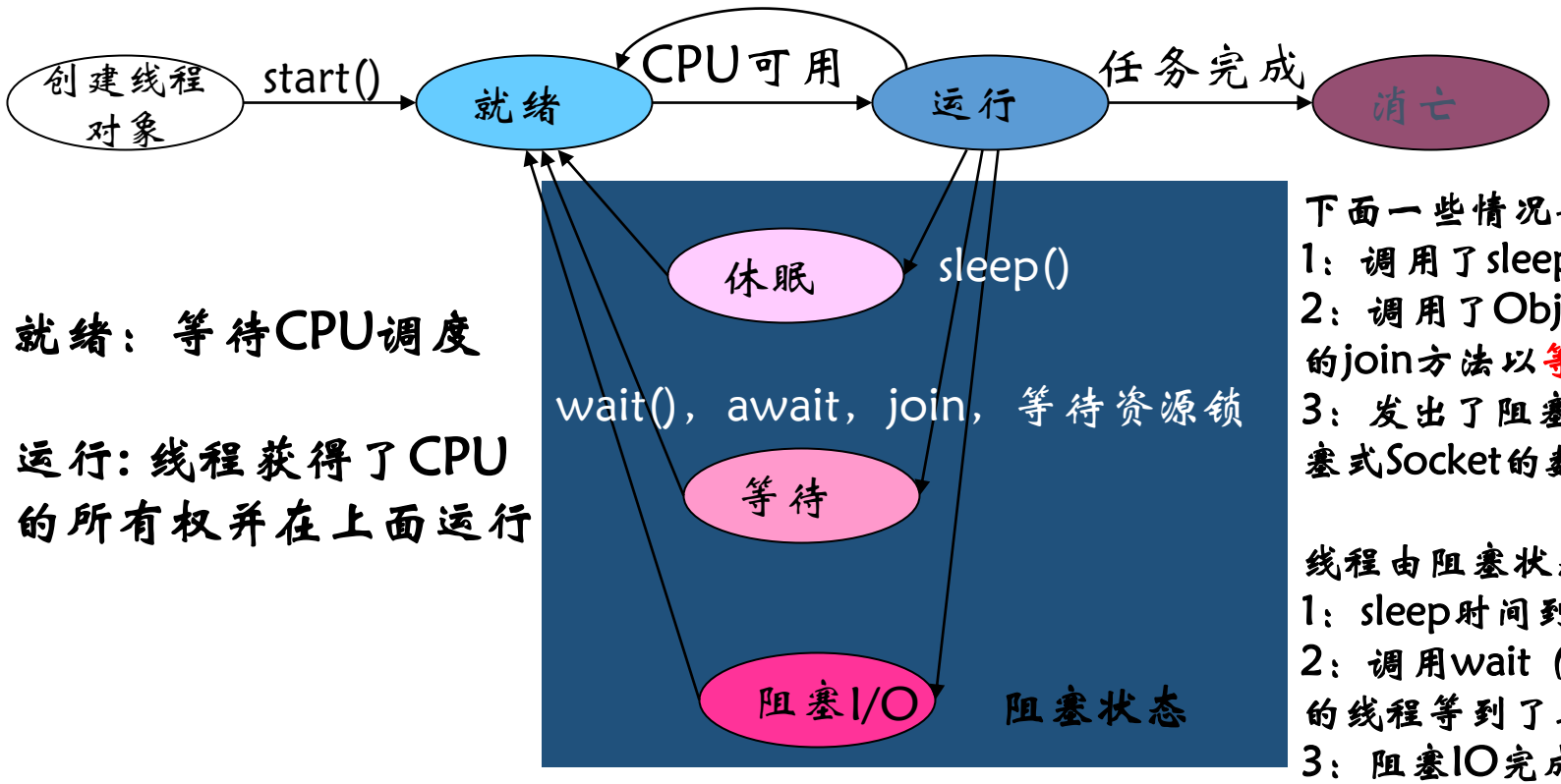
30.2 Runuable接口和线程类Thread

线程的状态切换

调用yield()方法主动放弃CPU的所有权，转到就绪态

yield()

消亡：当run()执行完毕后，线程就消亡。



就绪：等待CPU调度

运行：线程获得了CPU的所有权并在上面运行

下面一些情况导致线程从运行状态转到阻塞状态：

1：调用了sleep

2：调用了Object wait()方法、条件对象的await方法，Thread的join方法以等待其他线程，或者等待资源锁

3：发出了阻塞式IO操作请求，并等待IO操作结果（如等待阻塞式Socket的数据到来）

线程由阻塞状态被唤醒后，回到就绪态。唤醒的原因

1：sleep时间到

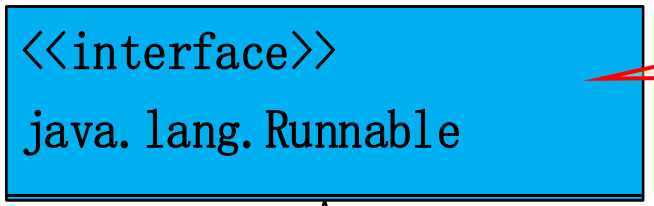
2：调用wait（await）的线程被其他线程notify，调用join方法的线程等到了其他线程的完成，线程拿到了资源锁

3：阻塞IO完成

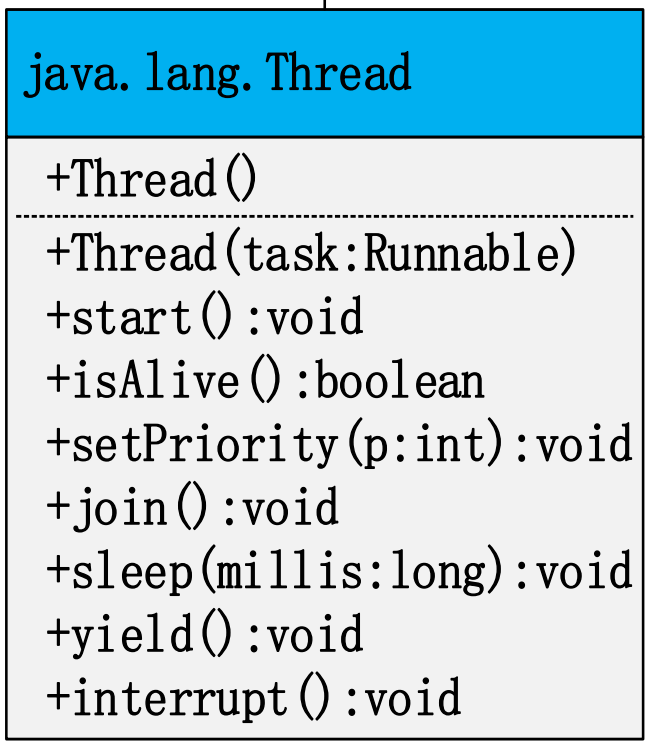
Object类定义了wait/notify/notifyAll方法

Thread类定义了sleep/yield/join方法

30.2 Runuable接口和线程类Thread



Runnable接口实例定义了线程任务，即线程要执行的逻辑。一个线程任务必须通过Thread线程对象来执行



Thread是对线程进行管理

创建一个空的线程

为指定的任务创建一个线程

开始一个线程，进入Ready状态，如无其它线程等待，可立即Run进入running状态

测试线程当前是否在运行

为该线程指定优先值 p(1～10)

等待线程结束

让当前线程休眠若干ms，监视器自动恢复其运行

将线程从running变为ready，允许其他线程执行（自己也可能立即执行）

中断该线程

30.2 Runuable接口和线程类Thread

线程的优先级

- 线程优先级范围从1－10，数字越高越能被优先执行。但优先级高并不代表能独自占用执行时间片，可能是优先级高得到越多的执行时间片，反之，优先级低的分到的执行时间少但不会分配不到执行时间
- 每个线程创建时赋予默认的优先级Thread.NORM_PRIORITY.
- 通过setPriority(int priority)为线程指定优先级.
- 用getPriority()方法获取线程的优先级.
- JAVA定义的优先级：1～10
- Thread类有int 类型的常量：
  - ◆Thread.MIN_PRIORITY（1）
  - ◆Thread.MAX_PRIORITY（10）
  - ◆Thread.NORM_PRIORITY（5）

30.2 Runuable接口和线程类Thread

线程的优先级

- 多个线程只能是“宏观上并行，微观上串行”
- 在有限个CPU的系统中确定多个线程的执行顺序称为线程的调度
- 自私的线程

```
run() {
        while (true) {
        }
}
```

应适当地在run()里sleep或yield一下，让其他线程有更多机会被运行。

不要编写依赖于线程优先级的程序

30.2 线程类Thread的yield，sleep方法

- 使用 yield() 方法为其他线程让出CPU时间：

```
public void run() {
    for (int i = 1; i < times; i++) {
        System.out.print(charToPrint);
        Thread.yield(); //挂起进入ready，给其它进程调度机会
    }
}
```

- sleep(long mills)方法将线程设置为休眠状态，确保其他线程执行：

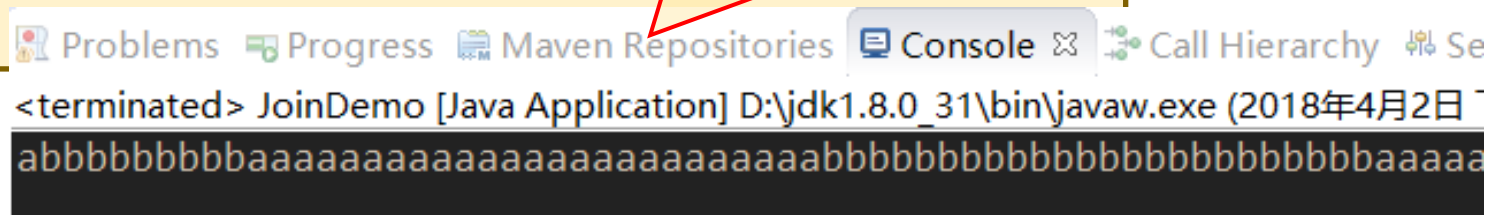
```
public void run() {
    try {//循环中使用sleep方法，循环放在try-catch块中
        for (int i = 1; i < times; i++) {
            System.out.print(charToPrint);
            if (i >= 50) Thread.sleep(1);
        }
    }
    // 必检异常：其它线程调当前线程（正在休眠）interupt方法会抛出该异常
    catch (InterruptedException ex) { }
}
```

处于阻塞状态（如在睡眠，在wait，在执行阻塞式IO）的线程，如果被其他线程打断（即处于阻塞的线程的interupt方法被其它线程调用），会抛出*InterruptedException*

30.2 线程类Thread的-join方法

```
public class JoinDemo {
        public static void main(String[] args) throws InterruptedException{
                Thread printA = new Thread(new PrintChar('a',100));
                Thread printB = new Thread(new PrintChar('b',100));
                printA.start();
                printB.start();
        }
}
class PrintChar implements Runnable //实现Runnable接口
{
        private char charToPrint;  // The character to print
        private int times;  // The times to repeat
        public PrintChar(char c, int t){  charToPrint = c;  times = t; }
        public void run(){ //实现Runnable中声明的run方法
                for (int i=1; i < times; i++)
                        System.out.print(charToPrint);
        }
}
```

可以看到屏幕上无规律的交替输出ab。这是多线程程序的特点，每次运行输出结果可能是不一样的。如果希望把所有a先打印完再打印b，怎么做？



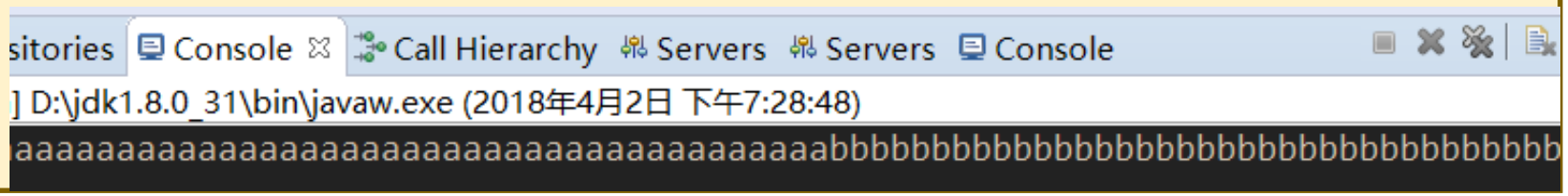
30.2 线程类Thread的-join方法

```
public class JoinDemo {
    public static void main(String[] args) throws InterruptedException{
        Thread printA = new Thread(new PrintChar('a',100));
        Thread printB = new Thread(new PrintChar('b',100));
        printA.start(); //在主线程里首先启动printA线程
        printA.join(); //主线程被阻塞，等待printA执行完
        printB.start(); //主线程被唤醒，启动printB线程
    }
}
class PrintChar implements Runnable
{
    private char charToPrint;  // The character to print
    private int times;  // The times to repeat
    public PrintChar(char c, int t){  charToPrint = c;  times = t; }
    public void run(){ //实现Runnable中声明的run方法
        for (int i=1; i < times; i++)
            System.out.print(charToPrint);
    }
}
```

join方法的作用：在A线程中调用了B线程（对象）的join()方法时，表示A线程放弃控制权（被阻塞了），只有当B线程执行完毕时，A线程才被唤醒继续执行。

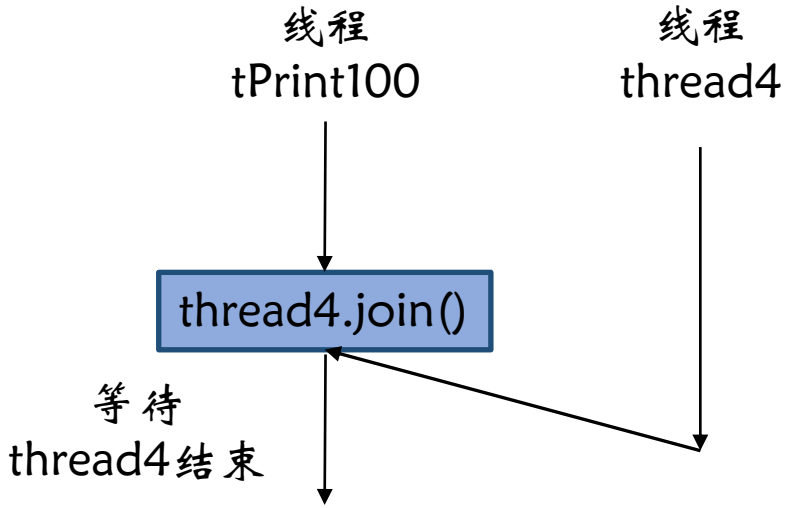
程序在main线程中调用printA线程（对象）的join方法时，main线程放弃cpu控制权（被阻塞），直到线程printA执行完毕，main线程被唤醒执行printB.start();

运行结果是全部a打印完才开始打印b



30.2 线程类Thread的-join方法

在线程任务对象print100的run中启动新线程Thread4，并调用Thread4 的join( ) 方法，，等待Thread4结束:



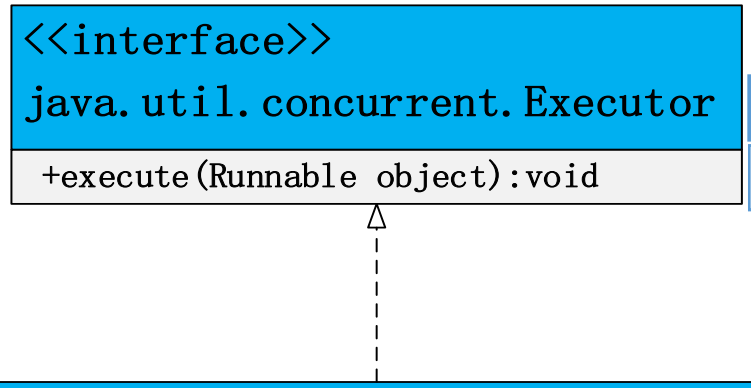

```
class PrintNum implements Runnable{ //实现新的线程任务类，打印数字
   private int lastNum;
   public PrintNum(int n){ lastNum = n; }
   @Override
   public void run() {
     Thread thread4=new Thread(new PrintChar('c',40));
      thread4.start();
      try{
         for(int i=1;i<lastNum;i++){
            System.out.print(" " + i);
            if(i == 50) thread4.join(); //join方法可以给参数指定至多等若干毫秒
         }
      }
      catch(InterruptedException e){ } //join方法可能会抛出这个异常
      }
}
```

启动tPrint100线程：

```
Runnable print100 = new PrintNum(100); //线程任务对象
Thread tPrint100 = new Thread(print100); //线程对象
tPrint100.start();
```

30.3 线程池

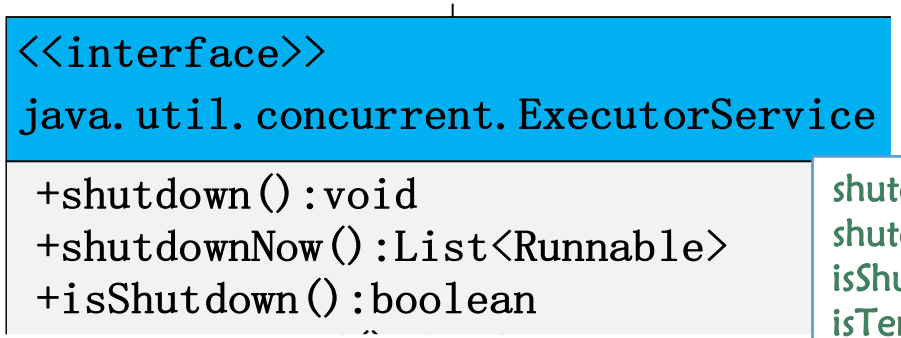
- 由于要为每一个线程任务创建一个线程（Thread对象），对于有大量线程任务的场景就不够高效（当线程任务执行完毕，即run方法结束后，Thread对象就消亡，然后又为新的线程任务去new新的线程对象…，当有大量的线程任务时，就不断的new Thread对象，Thread对象消亡，再new Thread对象…）
- 线程池适合大量线程任务的并发执行。线程池通过有效管理线程、“复用” 线程对象来提高性能.
- 从JDK 1.5 开始使用Executor接口（执行器）来执行线程池中的任务，Executor的子接口ExecutorService管理和控制任务



线程执行器接口，负责执行线程任务

执行任务，参数是线程任务对象

通过API接口可以看到，我们再不用去管Thread的创建，我们只需要把实例化好的线程任务对象（Runnable接口实例）交给执行器Executor就可以了。Thread由线程池内部来创建和维护。

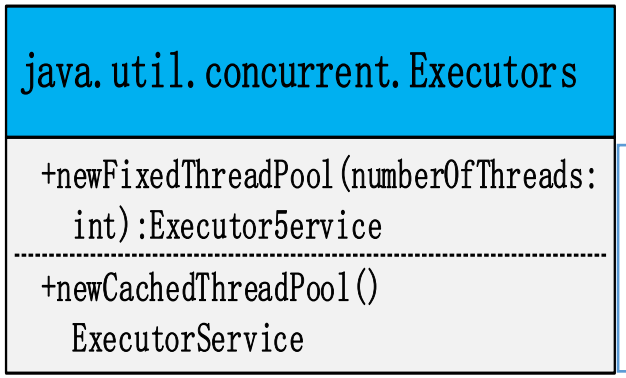


管理线程

shutdown：关闭执行器（允许执行器中的任务执行完）
shutdownNow:立刻关闭执行器，返回一个未完成行务的列表
isShutdown:如果执行器已经关闭.则返回true
isTerminated:如果池中所有任务终止.则返回true

30.3 线程池- 创建线程池

使用Executors的类方法创建一个线程池，Executors由Object派生



这里的任务就是指线程任务，是实现了Runnable接口的实例

创建一个可以运行指定数目线程的线程池，一个线程在当前任务已经完成的情况下可以重用，来执行另外一个任务

创建一个线程池，它会在必要的时候创建新的线程，但是如果之前已经创建好的线程可用，则先重用之前创建好的的线程（尽量复用，不够再创建新线程）

Executors还支持其它类型的线程池的创建方法如：newScheduledThreadPool、newSingleThreadPool

请自己查阅API

30.3 线程池-程序清单30-3

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

public class ExecutorDemo {
  public static void main(String[] args) {
    // Create a fixed thread pool with maximum three threads
     ExecutorService es= Executors.newFixedThreadPool(3);

    // Submit runnable tasks to the executor
            es.execute(new PrintChar('a', 100));
            es.execute(new PrintChar('b', 100));
            es.execute(new PrintNum(100));

    // Shut down
    es.shutdown();
  }
}
```

关闭，不接受新的线程任务，现有的任务将继续执行直到完成

30.3 线程池

进一步讨论：区分任务和线程

- 任务是实现了Runnable接口的类的实例，
- 这个任务的逻辑由run方法实现

```
class Task implements Runnable {
  //数据成员和其它方法...

  //Implement the run method in Runnable
  public void run() {
    //task logic...
  }
}
```

- 线程是Thread类的实例，是任务的运行载体
- 任务必须通过线程来运行

```
public class Client {
  public static void main(String... args) {
    Runnable task = new Task(...); // Create an instance of Task

    Thread thread = new Thread(task); // Create a thread
    thread.start(); // Start a thread
  }
}
```

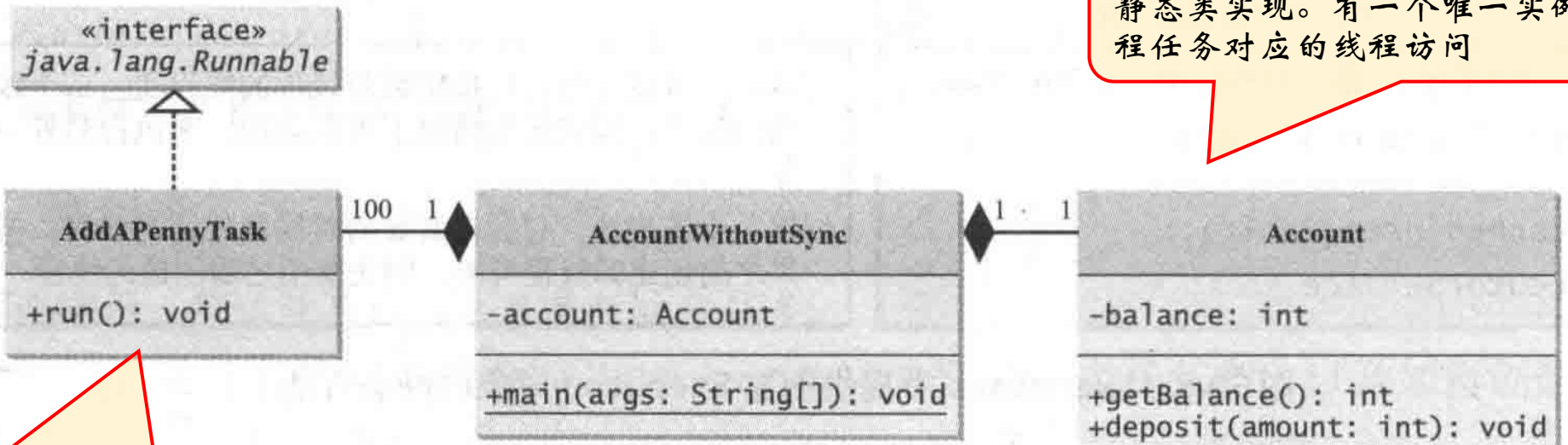
线程　　任务

```
Thread thread = new Thread(task);
```

如果这样直接new一个线程，线程启动后，执行任务的run方法，当任务的run方法执行完毕，线程对象使命就结束，被JVM回收。如果有大量的任务要运行，会导致频繁创建新线程、销毁线程。线程池维护多个创建好的线程，同时可以让多个任务“复用”线程，避免了线程的重复创建和销毁。但是一个线程任务被线程执行完后，线程就自动消亡。那么如何复用一个线程（即让线程去执行新的任务）？这个问题大家下去思考。

30.4 线程同步-程序清单30-4

- 如果一个共享资源（比如一个对象）被多个线程同时访问，如果不对访问的顺序进行控制，会造成不可预期的结果。这是需要对线程实施同步控制。
- Account是账户类，其数据成员balance为当前余额，deposit方法往账户存钱，getBalance方法读取账户余额
- AddPennyTask为任务，每次往账户里存一个便士
- 启动100个线程同时执行AddPennyTask任务



30.4 线程同步-程序清单30-4

```
public class AccountWithoutSync {
        private static class Account{
                private int balance = 0;
                public int getBalance() {
                        return balance;
                }
                public void deposit(int amount){
                        int newBalance = balance + amount;  //读取balance
                        try{ Thread.sleep(5); }
                        catch(InterruptedException e){      }
                        balance = newBalance;               //写balance
                }
        }
        private static class AddPennyTask implements Runnable{
                public void run() {  account.deposit(1); }

        }

         private static Account account = new Account();
}
```

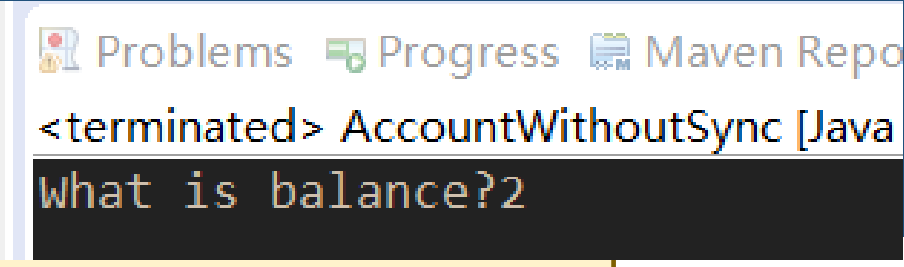
内部静态类Account

休眠是为了放大数据不一致的可能性

内部静态类AddPennyTask是线程任务类，实现Runnable接口
调用account对象的deposit方法

account对象被100个线程访问：每个线程的run方法都调用accoun.deposit方法

30.4 线程同步-程序清单30-4



原因：多个线程访问一个对象，没有同步

```
package ch30;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class AccountWithoutSync {


  public static void main(String[] args){
    ExecutorService executor = Executors.newCachedThreadPool();
    for(int i = 0; i < 100; i++){
      executor.execute(new AddPennyTask());
    }
    executor.shutdown();
    while(!executor.isTerminated()){ }
    System.out.println("What is balance?" + account.getBalance());

  }
}
```

通过线程池启动100个线程，执行AddPennyTask类型的任务

关闭线程池

等待线程池里线程全部结束

30.4 线程同步-程序清单30-4

原因：多个线程访问一个对象，没有同步

多个线程同时访问公共资源，会导致竞争状态（同时去修改公共资源）。为了避免竞争状态，应该防止多个线程同时进入程序的某一特定部分，这样的部分叫临界区。Account类的deposit方法就是临界区。可用synchronized关键字来同步，保证一次只有一个线程可以访问这个方法。当一个方法被synchronized修饰，这个方法就是原子的（一个线程开始执行这个方法，就不可中断）

task1和task2同时进入account对象的deposit方法

注意deposit里面每条语句都不是原子的（对应着一堆汇编指令），因此每条语句执行时，是随时可能被中断，切换到另外一个线程程的

Step	Balance	Task 1	Task 2
1	0	newBalance = balance + 1;	
2	0		newBalance = balance + 1;
3	1	balance = newBalance;	
4	1		balance = newBalance;

task1读取balance，值为0，因此newBalance=1

task2读取balance，值为0，因此newBalance=1

task1保存newBalance到balance，balance=1

task2保存newBalance到balance，balance=1

图 30-11 任务 1 和任务 2 都向同一余额里加 1

30.4 线程同步

- 线程同步用于协调多个线程访问公共资源

  公共资源被多个线程同时访问，可能会遭到破坏

  （程序清单30-4：AccountWithoutSync.java）

- 临界区(critical region)：可能被多个线程同时进入的程序的一部分区域
  - 所以需要对临界区同步，保证任何时候只能有1个线程进入临界区
- 可以用synchronized关键字来同步临界区
  - 临界区可以是方法，包括静态方法和实例方法，那么被synchronized关键字修饰的方法叫同步方法
  - 临界区也可以是语句块，也可以用synchronized关键字来同步语句块： 如synchronized(this){…}
- 除了用synchronized关键字，还可利用加锁同步临界区

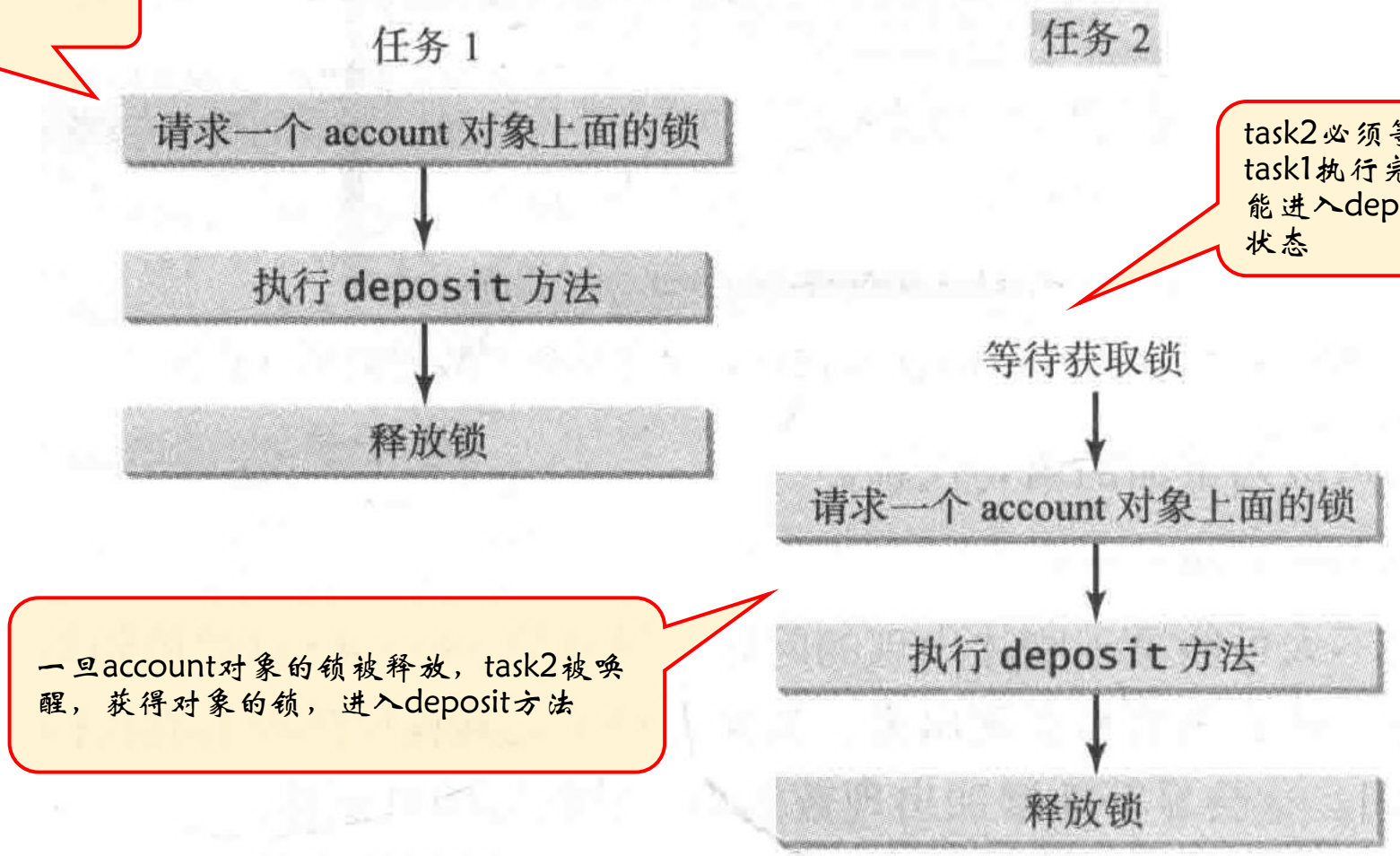
30.4 线程同步-synchronized

- synchronized可用于同步方法
- 使用关键字synchronized 来修饰方法：
  public synchronized void deposit(double amount)
- 一次只有一个线程可以进入这个同步方法
- synchronized关键字是如何做到方法同步的？通过加锁：一个线程要进入同步方法，首先拿到锁，进入方法后立刻上锁，导致其他要进入这个方法的线程被阻塞（等待锁）
  - 锁是一种实现资源排他使用的机制
  - 对于synchronized实例方法，是对调用该方法的对象（this对象）加锁
  - 对于synchronized静态方法，是对拥有这个静态方法的类加锁
- 当进入方法的线程执行完方法后，锁被释放，会唤醒等待这把锁的其他线程

30.4 线程同步-synchronized

■ public synchronized void deposit(double amount)

task1首先获得account对象的锁，进入deposit方法



task2必须等到account对象的锁（等到task1执行完deposit方法，锁被释放）才能进入deposit，因此这时task2进入阻塞状态

一旦account对象的锁被释放，task2被唤醒，获得对象的锁，进入deposit方法

图 30-12　任务 1 和任务 2 被同步

30.4 线程同步-synchronized

- synchronized也可以同步语句块
- 被synchronized关键字同步的语句块称为同步块(synchronized Block)

  synchronized (expr) { statements; } ，

- 表达式expr求值结果必须是一个对象的引用，因此可以通过对任何对象加锁来同步语句块
  - 如果expr指向的对象没有被加锁，则第一个执行到同步块的线程对该对象加锁，线程执行该语句块，然后解锁；
  - 如果expr指向的对象已经加了锁，则执行到同步块的其它线程将被阻塞
  - expr指向的对象解锁后，所有等待该对象锁的线程都被唤醒（唤醒后进入就绪态）
- 同步语句块允许同步方法中的部分代码，而不必是整个方法，增强了程序的并发能力
- 任何同步的实例方法都可以转换为同步语句块

```
public synchronized void xMethod() {
  // method body
}
```

```
public void xMethod() {
  synchronized (this) {
    // method body
  }
}
```

30.4 线程同步-synchronized

■因此前面的例子有如下几种修改方案

```
private static class Account{
        private int balance = 0;
        public int getBalance() {
                return balance;
        }

        public synchronized void deposit(int amount){
                int newBalance = balance + amount;
                try{ Thread.sleep(5); }
                catch(InterruptedException e){        }
                balance = newBalance;
        }
}
```

deposit方法改为同步的

sleep方法调用之后，并没有释放锁。使得线程仍然可以同步控制。sleep不会让出系统资源；

凡是在上了锁的临界区里，sleep方法不会释放锁

30.4 线程同步-synchronized

■ 因此前面的例子有如下几种修改方案

```
private static class Account{
        private int balance = 0;
        public int getBalance() {
                return balance;
        }
        public  void deposit(int amount){
                synchronized(this){
                        int newBalance = balance + amount;

                        try{ Thread.sleep(5); }
                        catch(InterruptedException e){        }
                        balance = newBalance;
                }
        }
}
```

修改共享资源Account类，
在deposit方法内部加同步块

30.4 线程同步-synchronized

■ 因此前面的例子有如下几种修改方案

```
private static class AddPennyTask implements Runnable{
        public void run() {
                synchronized(account){
                        account.deposit(1);
                }

        }

}
```

修改线程任务类AddPennyTask，在run方法里加同步块

注意：是对account对象加锁。不能对this对象加锁，即

synchronized(this)是错误的

因为this是AddPennyTask对象，100个线程任务对象各不相同，因此synchronized(this)是对100个线程任务对象分别加锁，根本没起到同步的作用。

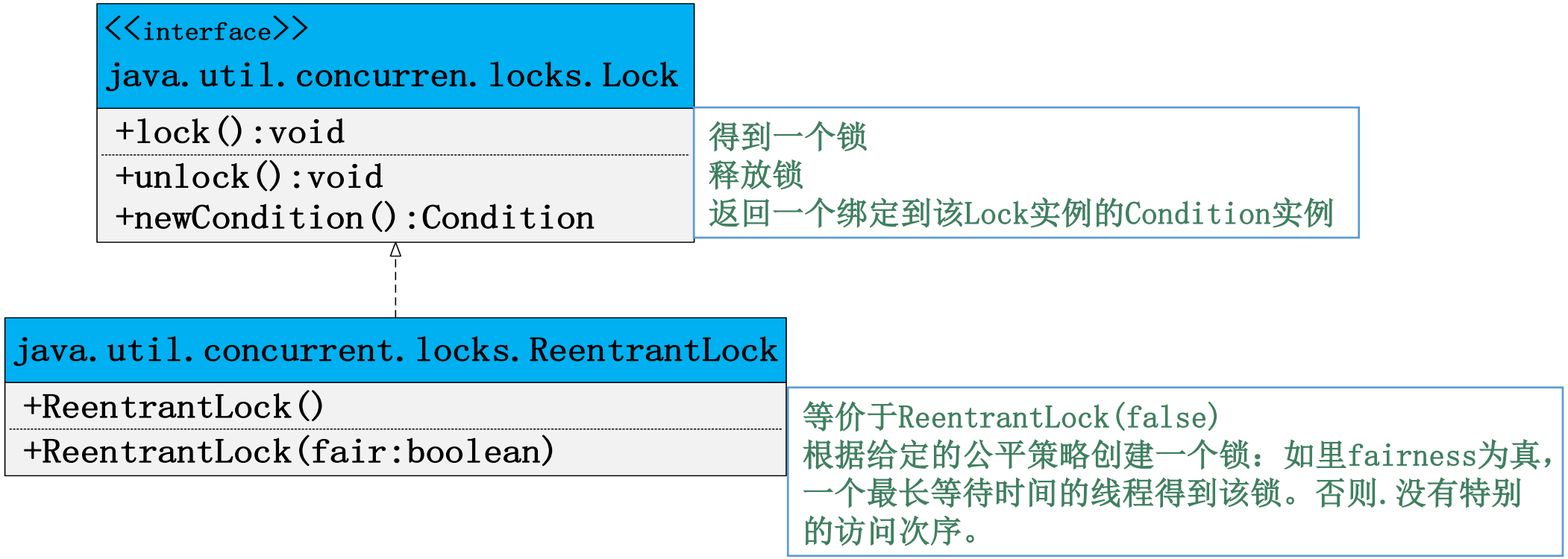
100个线程任务对象同时访问的是共享资源account对象，需要加锁同步的是account对象。

问题：public synchronized void run() { account.deposit(1); }

这种方案是否可以？

30.4 线程同步-加锁同步

- 采用synchronized关键字的同步要隐式地在对象实例或类上加锁，粒度较大影响性能
- JDK 1.5 可以显式地加锁，能够在更小的粒度上进行线程同步（后面会展开详细讨论）
- 一个锁是一个Lock接口的实例
- 类ReentrantLock是Lock的一个具体实现：可重入的锁



30.4 线程同步-加锁同步

- 可重入性锁描述这样的一个问题：一个线程在持有一个锁的时候，它能否再次（多次）申请该锁。如果一个线程已经获得了锁，它还可以再次获取该锁而不会死锁，那么我们就称该锁为可重入锁。通过以下伪代码说明：

```
void methodA(){
        lock.lock(); // 获取锁
        methodB();
        lock.unlock() // 释放锁
}

void methodB(){
        lock.lock(); // 再次获取该锁
         // 其他业务
        lock.unlock();// 释放锁
}
```

- Java关键字synchronized隐式支持重入性

30.4 线程同步-加锁同步-程序清单30-5

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
public class AccountWithSyncUsingLock {
  private static Account account = new Account();
  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();
    // Create and launch 100 threads
    for (int i = 0; i < 100; i++) {executor.execute(new AddAPennyTask());}
    executor.shutdown();
    // Wait until all tasks are finished
    while (!executor.isTerminated()) { }
    System.out.println("What is balance ? " + account.getBalance());
  }
  // A thread for adding a penny to the account
  public static class AddAPennyTask implements Runnable {
    public void run() {account.deposit(1); }
  }
```

30.4 线程同步-加锁同步

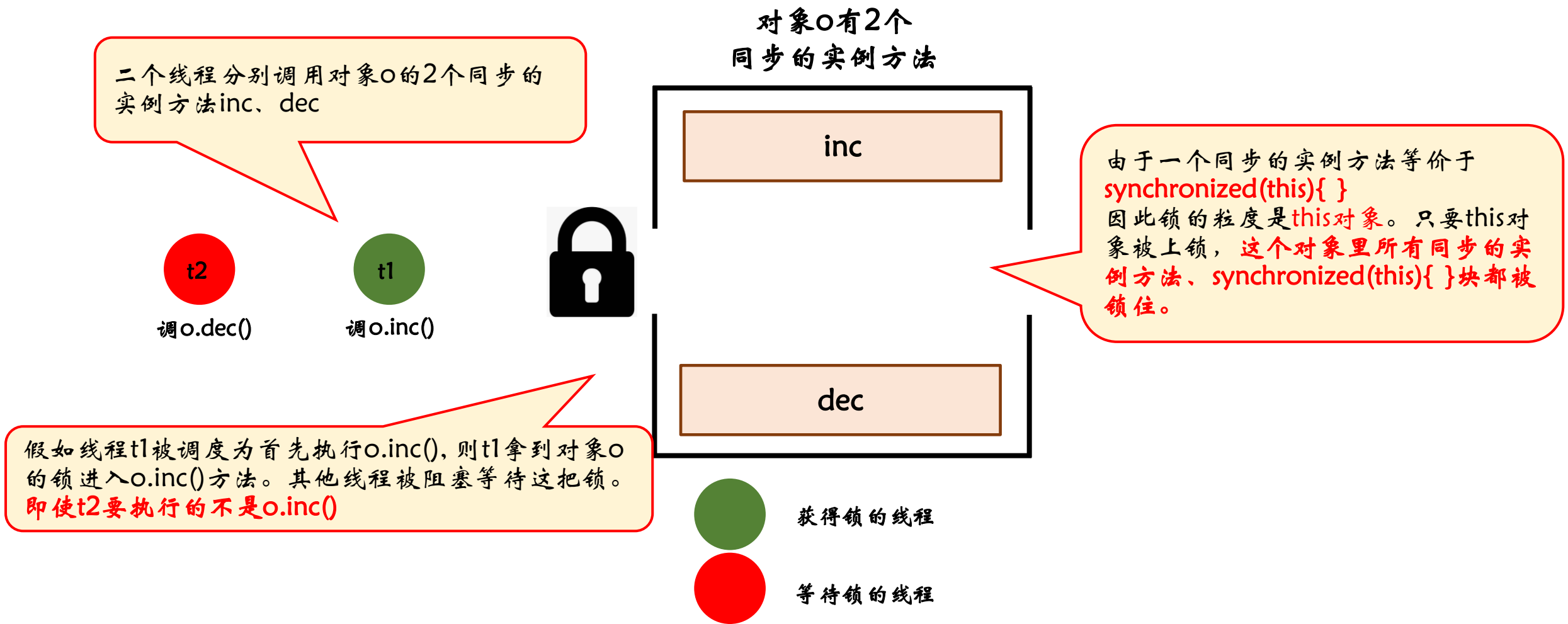
```
public static class Account {// An inner class for account， 主要变化在账户类
   private static Lock lock = new ReentrantLock(); // 注意这里是静态的， 被所有Account实例共享
   private int balance = 0;
   public int getBalance() {return balance;}
   public void deposit(int amount) {
      lock.lock( ); // Acquire the lock
      try {
         int newBalance = balance + amount;
         Thread.sleep(5);
         balance = newBalance;
      }
      catch (InterruptedException ex) {  }
      finally { lock.unlock(); // Release the lock，在finally中进行锁的释放。}
   }
}
}
```

在这里加锁（临界区开始），第一个进入这个方法的线程获得锁，把deposite方法锁住。其他进入方法的线程必须等待这把锁，因为进入阻塞状态

在finally块里释放锁，其它等待这把锁的线程被唤醒，第一个获得锁的线程可以进入该方法了，进去后又对deposite上锁…

30.4 线程同步-场景1

■ 假设一个类有多个用synchronized修饰的同步实例方法，如果多个线程访问这个类的同一个对象，当一个线程获得了该对象锁进入到其中一个同步方法时，这把锁会锁住这个对象所有的同步实例方法



30.4 线程同步-场景1

```
class Resource{//共享资源类，这个类的实例被多个线程访问
   private int value = 0;  //多个线程会同时对这个数据成员读写
   public int getValue(){return  value;}
   public synchronized void inc(int amount)  {
      System.out.print("\nThread " + Thread.currentThread().getId() + " 进入inc: ");
      int newValue = value + amount;
      try{ Thread.sleep(5); }  catch(InterruptedException e){      }
      value = newValue;
      System.out.print("-->Thread " + Thread.currentThread().getId() + " 离开inc.");
   }
   public synchronized void dec(int amount)  {
      System.out.print("\nThread " + Thread.currentThread().getId() + " 进入dec: ");
      int newValue = value - amount;
      try{ Thread.sleep(2); } catch(InterruptedException e){      }
      value = newValue;
      System.out.print("-->Thread " + Thread.currentThread().getId() + " 离开dec.");
   }
}
```

多个线程会对value读写，因此需要同步控制。value就是竞争性资源。Resource对象也因此成为竞争性资源

inc方法把value值增加，增加量为amount

inc、dec二个方法都是synchronized的，二个同步的实例方法

dec方法把value减少，减少量为amount

30.4 线程同步-场景1

```
class IncTask implements Runnable{
   private Resource r = null; //要访问的对象
   private int amount = 0; //每次增加量
   private int loops = 0;  //循环次数
   public IncTask(Resource r,int amount,int loops){
      this.r = r; this.amount = amount; this.loops = loops;
   }
   public void run() {
      for(int i = 0; i < loops; i++) { r.inc(amount); }
   }
}
```

IncTask是线程任务，会循环调用资源对象r.inc方法，loops是循环次数。

IncTask的构造函数会传入要访问的对象

```
class DecTask implements Runnable{
   private Resource r = null; //要访问的对象
   private int amount = 0; //每次减少量
   private int loops = 0;  //循环次数
   public DecTask(Resource r, int amount, int loops){
      this.r = r;this.amount = amount;this.loops = loops;
   }
   public void run() {
      for(int i = 0; i < loops; i++) {  r.dec(amount); }
   }
}
```

DecTask是线程任务，会循环调用资源对象r.dec方法，loops是循环次数。

DecTask的构造函数会传入要访问的对象

30.4 线程同步-场景1

```
public static void test1(){
    int incAmount = 10;
    int decAmount = 5;
    int loops = 100;

    Resource r = new Resource();
    Runnable incTask = new IncTask( r, incAmount, loops );
    Runnable decTask = new DecTask( r, decAmount, loops );

    ExecutorService es = Executors.newCachedThreadPool();
    es.execute(incTask); es.execute(decTask);
    es.shutdown();
    while(!es.isTerminated()){ }

    int correctValue = (incAmount - decAmount)  * loops;
    System.out.println("\nThe value: " +  r.getValue() + ", correct value: " + correctValue);
}
```

每次增加量为10，每次减少量5，循环次数100

创建资源对象r

构造线程任务incTask，注意传入的第一个参数为对象r，
构造线程任务decTask，注意传入的第一个参数也为对象r，
意味着这二个线程任务一旦执行，对应的二个线程调用的是同一个对象的方法，一个线程(t1)调用r.inc，另外一个线程(t2)调用r.dec

计算出正确的value值
显示实际的计算值和正确的值，验证同步是否正确

30.4 线程同步-场景1

这个例子说明，二个线程访问同一个对象时， 只要一个线程拿到对象锁，这个对象的所有同步实例方法都被锁
即incTask对应线程进入inc时，decTask的线程不能进入dec

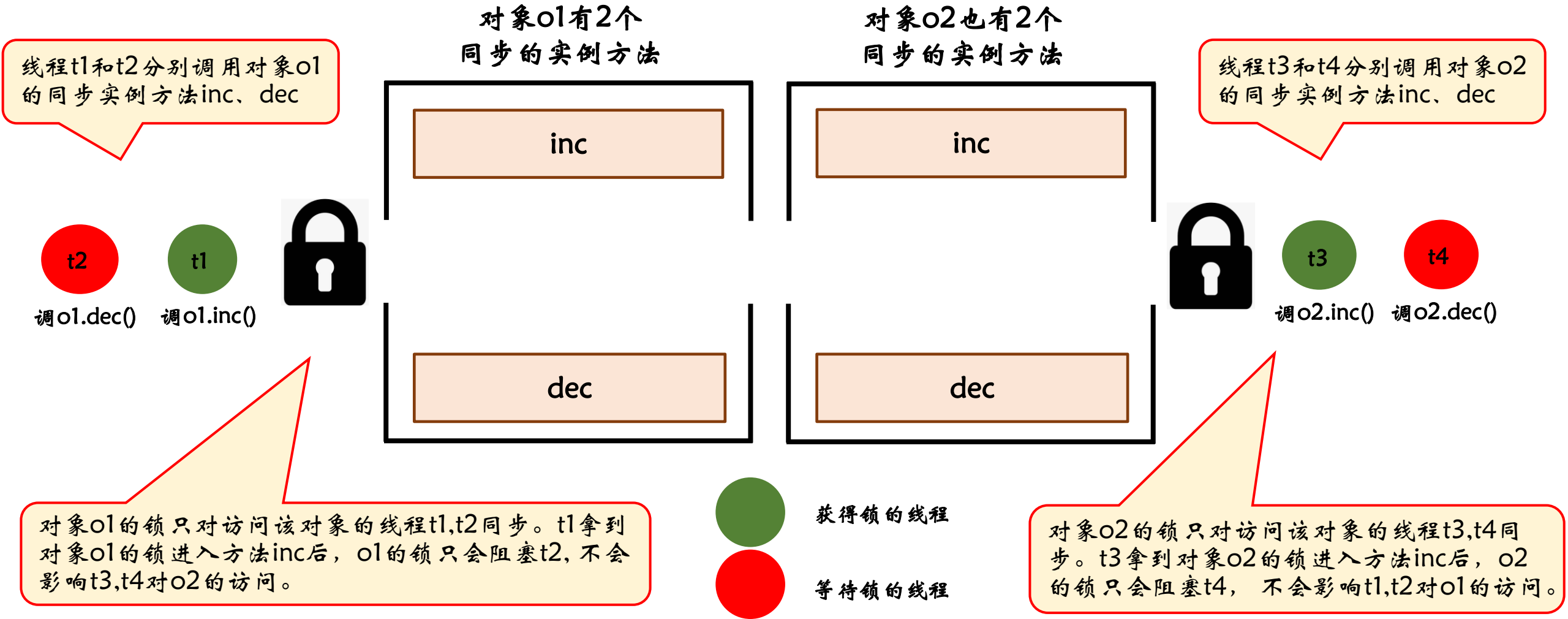
```
Thread 13 进入inc: -->Thread 13 离开inc.
…
Thread 13 进入inc: -->Thread 13 离开inc.
Thread 14 进入dec: -->Thread 14 离开dec.
Thread 14 进入dec: -->Thread 14 离开dec.
Thread 14 进入dec: -->Thread 14 离开dec.
Thread 14 进入dec: -->Thread 14 离开dec.
Thread 14 进入dec: -->Thread 14 离开dec.
Thread 13 进入inc: -->Thread 13 离开inc.
…
The value: 500, correct value: 500
```

可以看到线程13进入inc，对象被加锁，线程14必须等待线程13退出inc后，才能进入dec。所以输出结果里线程13和14的控制台输出没有乱。

最下面的输出的value值也是正确的

30.4 线程同步-场景2

■ 假设一个类有多个用synchronized修饰的同步实例方法，如果多个线程访问这个类的不同对象，那么不同对象的synchronized锁不一样，每个对象的锁只能对访问该对象的线程同步



30.4 线程同步-场景2

4个线程访问二个不同对象，不同对象锁不一样

incTask1进入对象r1的inc，和incTask2是否可以进入对象r2的inc无关

```
public static void test2(){
    int incAmount = 10,decAmount = 5, loops = 3;

    Resource r1 = new Resource();
    Resource r2 = new Resource();
    Runnable incTask1 = new IncTask(r1,incAmount,loops);    Runnable decTask1 = new DecTask(r1,decAmount,loops);
    Runnable incTask2 = new IncTask(r2,incAmount,loops);    Runnable decTask2 = new DecTask(r2,decAmount,loops);

    ExecutorService es = Executors.newCachedThreadPool();
    es.execute(incTask1);es.execute(decTask1);
    es.execute(incTask2);es.execute(decTask2);
    es.shutdown();
    while(!es.isTerminated()){ }
    int r1CorrectValue = (incAmount - decAmount)  * loops ; int r2CorrectValue = (incAmount - decAmount)  * loops;
    System.out.println("\nThe value of r1: " + r1.getValue() + ", correct value: " + r1CorrectValue);
    System.out.println("\nThe value of r2: " + r2.getValue() + ", correct value: " + r2CorrectValue);
}
```

每次增加量为10，每次减少量5，循环次数3

注意，和场景1不同，现在创建二个资源对象r1和r2

incTask1和decTask1访问对象r1；它们的运行线程被r1的对象锁同步；
incTask2和decTask2访问对象r2；它们的运行线程被r2的对象锁同步；
但是这二对线程之间没有同步约束，例如incTask1和incTask2的运行线程不会被同步

30.4 线程同步-场景2

```
Thread 13 进入inc:
Thread 15 进入inc: -->Thread 13 离开inc.
Thread 13 进入inc: -->Thread 15 离开inc.
Thread 15 进入inc: -->Thread 13 离开inc.
Thread 13 进入inc: -->Thread 15 离开inc.
Thread 15 进入inc: -->Thread 15 离开inc.-->Thread 13 离开inc.
Thread 14 进入dec:
Thread 16 进入dec: -->Thread 14 离开dec.
Thread 14 进入dec: -->Thread 16 离开dec.
Thread 16 进入dec: -->Thread 14 离开dec.
Thread 14 进入dec: -->Thread 16 离开dec.
Thread 16 进入dec: -->Thread 16 离开dec.-->Thread 14 离开dec.
The value of r1: 15, correct value: 15

The value of r2: 15, correct value: 15
```

4个线程访问二个不同对象，不同对象锁不一样

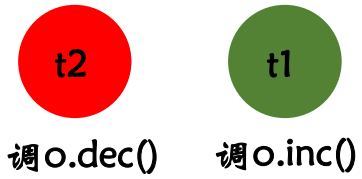
incTask1进入对象r1的inc，和incTask2是否可以进入对象r2的inc无关

计算结果正确。
13和15分别访问二个对象的inc，没有同步约束，因此它们的System.out.println输出是乱的；14和16也是如此。但是13和14是同步的，15和16也是同步的。如果只看13和14的输出，就没有乱。

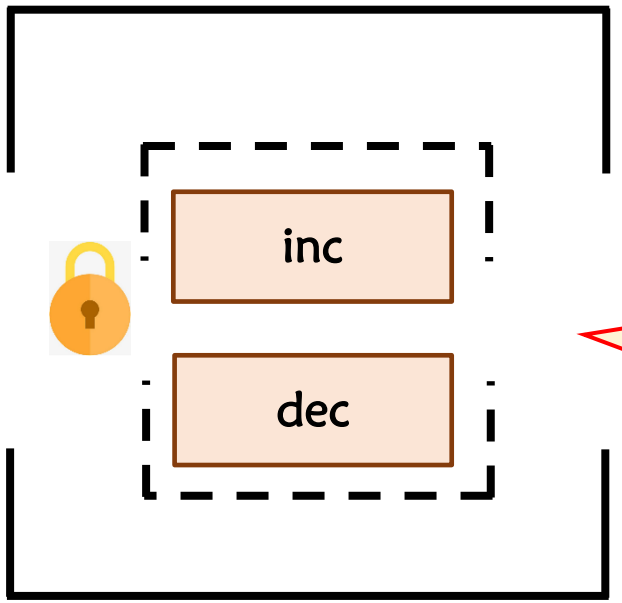
30.4 线程同步-场景3

■ 如果采用Lock锁进行同步，一旦Lock锁被一个线程获得，那么被这把锁控制的所有临界区都被上锁，这时所有其他访问这些临界区的线程都被阻塞。

2个线程分别调用对象o的2个实例方法inc、dec

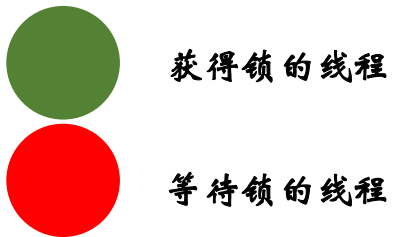


对象o有2个实例方法inc，dec都用同一个Lock对象上锁，这时inc和dec就是这个Lock锁对象控制的临界区



注意Lock锁和synchronized锁的粒度不一样。Lock锁不是锁整个对象，它只锁住使用该锁的临界区，如图中的虚线框里，inc，dec2个实例方法都是被同一个Lock锁控制的临界区。

如调度到线程t1执行o.inc()，则t1拿到Lock锁进入o.inc()方法。这时inc和dec方法都被锁住，t2线程被阻塞，即使t2要执行的不是o.inc()



30.4 线程同步-场景3

```
class ResourceWithLock {//重新定义资源类，采用Lock锁
   private  Lock lock = new ReentrantLock(); //创建Lock锁对象，注意是实例变量
   private int value = 0;  //多个线程会同时对这个数据成员读写
   public int getValue(){ return  value; }
   public void inc(int amount)  {
      lock.lock( ); // Acquire the lock
      try{ System.out.print("\nThread " + Thread.currentThread().getId() + " 进入inc: ");
         int newValue = value + amount;  try{ Thread.sleep(5); } catch(InterruptedException e){ }
         value = newValue; System.out.print("-->Thread " + Thread.currentThread().getId() + " 离开inc.");
      } finally{ lock.unlock(); }
   }
   public void dec(int amount) {
      lock.lock( ); // Acquire the lock
      try{ System.out.print("\nThread " + Thread.currentThread().getId() + " 进入dec: ");
         int newValue = value - amount; try{ Thread.sleep(2); } catch(InterruptedException e){ }
         value = newValue; System.out.print("-->Thread " + Thread.currentThread().getId() + " 离开dec.");
      } finally { lock.unlock(); }
   }
}
```

ResourceWithLock采用Lock对临界区上锁，而不是用synchronized关键字

inc方法在这加Lock锁（临界区开始）

inc退出方法前解Lock锁，退出临界区

dec方法用同一把Lock锁。

Inc，dec方法用同一把Lock锁。因此如果调用lock.lock(); 加锁，这个锁同时锁住inc和dec方法。

30.4 线程同步-场景3

```
class IncTaskWithLock implements Runnable{
   private ResourceWithLock r = null;
   private int amount = 0;
   private int loops = 0; //循环次数
   public IncTaskWithLock(ResourceWithLock r,int amount,int loops){
      this.r = r; this.amount = amount; this.loops = loops;
   }
   public void run() {
      for(int i = 0; i < loops; i++) { r.inc(amount); }
   }
}
```

IncTask是线程任务，会循环调用资源对象r.inc方法，loops是循环次数。

线程任务现在要访问的资源类型是ResourceWithLock

```
class DecTaskWithLock implements  Runnable{
   private ResourceWithLock r = null;
   private int amount = 0;
   private int loops = 0; //循环次数
   public DecTaskWithLock(ResourceWithLock r, int amount, int loops){
      this.r = r; this.amount = amount; this.loops = loops;
   }
   public void run() {
      for(int i = 0; i < loops; i++) { r.dec(amount); }
   }
}
```

DecTask是线程任务，会循环调用资源对象r.dec方法，loops是循环次数。

线程任务现在要访问的资源类型是ResourceWithLock

30.4 线程同步-场景3

```
public static void test1(){
    int incAmount = 10; int decAmount = 5; int loops = 50;

    ResourceWithLock r = new ResourceWithLock();
    Runnable incTask = new IncTaskWithLock( r ,incAmount,loops);
    Runnable decTask = new DecTaskWithLock( r ,decAmount,loops);

    ExecutorService es = Executors.newCachedThreadPool();
    es.execute(incTask); es.execute(decTask);
    es.shutdown(); while(!es.isTerminated()){ }

    int correctValue = (incAmount - decAmount)  * loops;
    System.out.println("\nThe value: " + r.getValue() + ", correct value: " + correctValue);
}
```

创建资源对象r，注意资源类型是ResourceWithLock

构造线程任务incTask，decTask，**注意:二个任务的线程访问的是同一个ResourceWithLock对象，用的是同一把Lock锁**。这时只要一个线程拿到锁，所有被这个锁控制的临界区都被锁住。即incTask的线程进入inc时，decTask的线程不能进入dec

30.4 线程同步-场景3

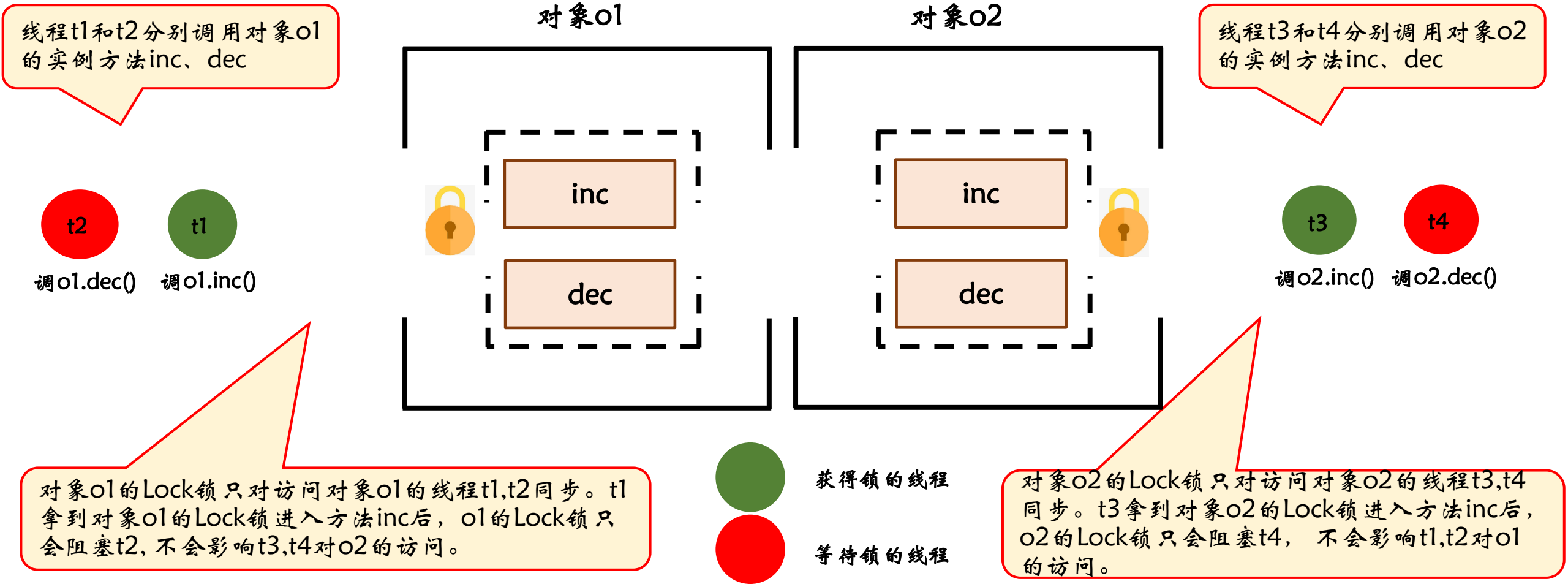
```
Thread 13 进入inc: -->Thread 13 离开inc.
Thread 14 进入dec: -->Thread 14 离开dec.
Thread 14 进入dec: -->Thread 14 离开dec.
Thread 14 进入dec: -->Thread 14 离开dec.
…
Thread 14 进入dec: -->Thread 14 离开dec.
Thread 13 进入inc: -->Thread 13 离开inc.
Thread 13 进入inc: -->Thread 13 离开inc.
…
The value: 250, correct value: 250
```

可以看到线程13进入inc，临界区inc和dec都被加锁，线程14必须等待线程13退出inc后，才能进入dec。所以输出结果里线程13和14的控制台输出没有乱。

最下面的输出的value值也是正确的

30.4 线程同步-场景4

■ 如果一个类采用Lock锁对临界区上锁，而且这个Lock锁也是该类的实例成员（见ResourceWithLock的里的lock对象定义），那么这个类的二个实例的Lock锁就是不同的锁，下面的动画演示了这种场景：对象o1的Lock锁和对象o2的Lock锁是不同的锁对象。



30.4 线程同步-场景4

```
public static void test2(){
    int incAmount = 10, decAmount = 5,loops = 20;
    ResourceWithLock r1 = new ResourceWithLock(); ResourceWithLock r2 = new ResourceWithLock();

    //incTask1，decTask1访问同一个对象r1，它们之间同步
    Runnable incTask1 = new IncTaskWithLock(r1,incAmount,loops);
    Runnable decTask1 = new DecTaskWithLock(r1,decAmount,loops);
    //incTask2，decTask2访问同一个对象r2，它们之间同步
    Runnable incTask2 = new IncTaskWithLock(r2,incAmount,loops);
    Runnable decTask2 = new DecTaskWithLock(r2,decAmount,loops);

    ExecutorService es = Executors.newCachedThreadPool();
    es.execute(incTask1); es.execute(decTask1); es.execute(incTask2); es.execute(decTask2);
    es.shutdown();
    while(!es.isTerminated()){ }
    int r1CorrectValue = (incAmount - decAmount) * loops ; int r2CorrectValue = (incAmount - decAmount) * loops;
    System.out.println("\nThe value of r1: " + r1.getValue() + ", correct value: " + r1CorrectValue);
    System.out.println("\nThe value of r2: " + r2.getValue() + ", correct value: " + r2CorrectValue);
}
```

注意，和场景3不同，现在创建二个资源对象r1和r2.注意资源类型是ResourceWithLock

四个线程访问二个不同对象
*incTask1，decTask1的执行线程访问同一个对象r1，它们之间同步
* incTask2，decTask2的执行线程访问同一个对象r2，它们之间同步
* incTask1执行线程进入对象r1的inc，和incTask2执行线程是否可以进入对象r2的inc无关，因为锁不一样

30.4 线程同步-场景4

Thread 13 进入inc:
Thread 15 进入inc: -->Thread 15 离开inc.
Thread 16 进入dec: -->Thread 13 离开inc.
Thread 14 进入dec: -->Thread 16 离开dec.
Thread 16 进入dec: -->Thread 14 离开dec.
Thread 14 进入dec: -->Thread 16 离开dec.
Thread 16 进入dec: -->Thread 14 离开dec.
Thread 14 进入dec: -->Thread 16 离开dec.
Thread 15 进入inc: -->Thread 14 离开dec.
Thread 13 进入inc: -->Thread 15 离开inc.
Thread 15 进入inc: -->Thread 13 离开inc.
Thread 13 进入inc: -->Thread 15 离开inc.-->Thread 13 离开inc.
The value of r1: 15, correct value: 15

The value of r2: 15, correct value: 15

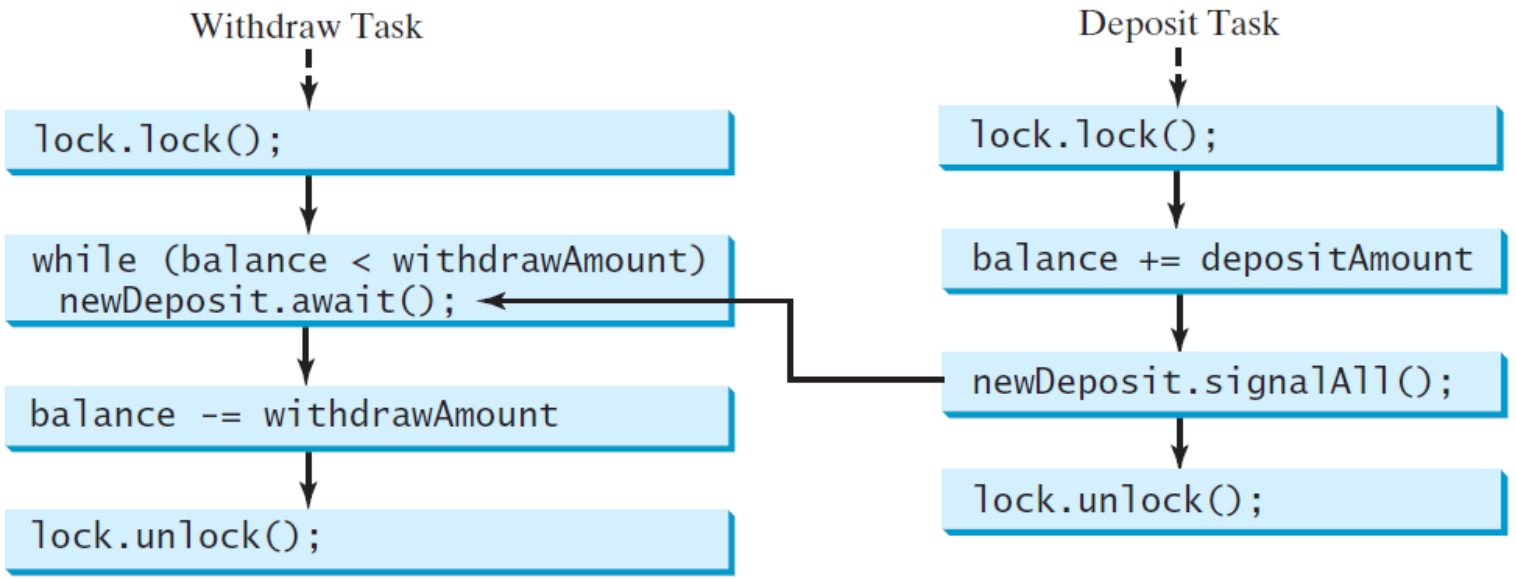
计算结果正确。
13和15分别访问二个对象的inc，没有同步约束，因此它们的输出是乱的；14和16也是如此。但是13和14是同步的，15和16也是同步的。如果只看13和14的输出，就没有乱。

30.4 线程同步-总结和思考

- 如果采用synchronized关键字对类A的实例方法进行同步控制，这时等价于synchronized(this){}
  - 一旦一个线程进入类A的对象o的synchronized实例方法，对象o被加锁，对象o所有的synchronized实例方法都被锁住，从而阻塞了要访问对象o的synchronized实例方法的线程，但是与访问A类其它对象的线程无关
- 如果采用synchronized关键字对类A的静态方法进行同步控制，这时等价于synchronized(A.class){}。一旦一个线程进入A的一个静态同步方法，A所有的静态同步方法都被锁（这个锁是类级别的锁），这个锁对所有访问该类静态同步方法的线程有效，不管这些线程是通过类名访问静态同步方法还是通过不同的对象访问静态同步方法。
- 如果通过Lock对象进行同步，首先看Lock对象对哪些临界区上锁，一旦Lock锁被一个线程获得，那么被这把锁控制的所有临界区都被上锁（如场景3）；另外要区分Lock对象本身是否是不同的：不同的Lock对象能阻塞的线程是不一样的（如场景4）。
  - 对于场景4，请思考，如果把ResourceWithLock里的实例成员lock改成静态成员，结果有什么不一样？

30.4 线程同步-线程协作

- 线程之间有资源竞争，synchronized和Lock锁这些同步机制解决的是资源竞争问题
- 线程之间还有相互协作的问题
- 假设创建并启动两个任务线程：
  - 存款线程用来向账户中存款
  - 提款线程从同一账户中提款
  - 当提款的数额大于账户的当前余额时，提款线程必须等待存款线程往账户里存钱
  - 如果存款线程存入一笔资金，必须通知提款线程重新尝试提款，如果余额仍未达到提款的数额，提款线程必须继续等待新的存款



例：30-6

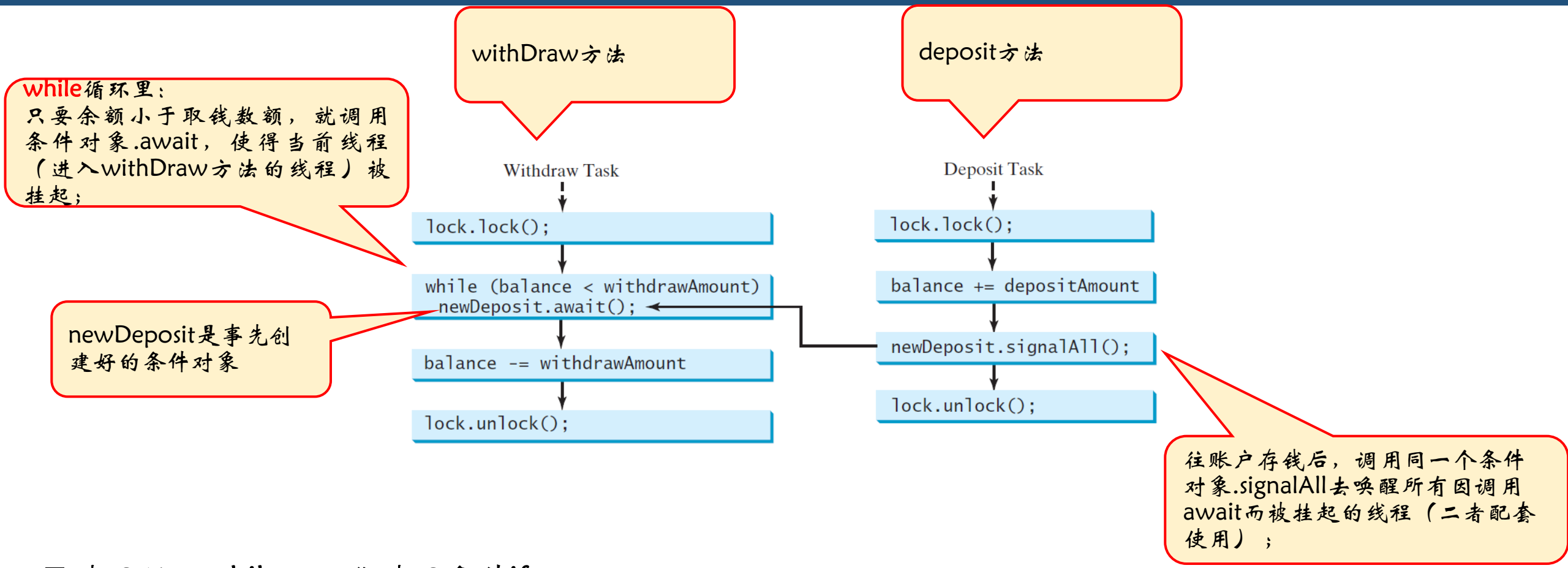
30.4 线程同步-线程协作

- 线程之间的相互协作：可通过Condition对象的await/signal/signalAll来完成
  - Condition (条件)对象是通过调用Lock实例的newCondition()方法而创建的对象
  - Condition对象可以用于协调线程之间的交互（使用条件实现线程间通信）
  - 一旦创建了条件对象condition，就可以通过调用condition.await()使当前线程进入等待状态，
  - 其它线程通过同一个条件对象调用signal和signalAll()方法来唤醒等待的线程，从而实现线程之间的相互协作
- 锁和条件是Java 5中的新内容，在Java 5之前，线程通信是使用对象的内置监视器（Object类的wait/signal/signalAll）编程实现
- 锁和条件比内置监视器更加强大且灵活，因此无须使用内置监视器，但要注意遗留代码中的内置监视器

<<interface>> java.util.concurren.Condition
+await():void
+signal():void +signalAll():Condition

引起当前线程（调用await的线程）等待，直到收到条件信号signal/signalAll
唤醒一个等待线程
唤醒所有等待线程

30.4 线程同步-线程协作



- 使用循环while而不能使用条件if
- 有await，就要有 signal()或者signalAll()，要不然一直等
- 条件对象由Lock对象创建，通过条件对象调用它的方法await/signal/signalAll()，为调用这些方法，必须首先拥有锁（即先调用lock方法）

30.4 线程同步-线程协作

```
public class ThreadCooperation {
    private static class Account {// An inner class for account
        private static Lock lock = new ReentrantLock();
        private static Condition newDeposit = lock.newCondition();
        private int balance = 0;
        public int getBalance() {return balance;}
        public void withdraw(int amount) {
            lock.lock(); // Acquire the lock
            try {
                while (balance < amount) {
                    System.out.println("\t\t\t\tWait for a deposit");
                    newDeposit.await();
                }
                balance -= amount;
                System.out.println("\t\t\t\tWithdraw " + amount +"\t\t\t\t" + getBalance());
            }
            catch (InterruptedException ex) {ex.printStackTrace();}
            finally {lock.unlock(); }
        }
        public void deposit(int amount){
            lock.lock();
            try{
                balance+=amount;
                System.out.println("deposit " + amount + "\t\t\t\t\t\t\t\t" + getBalance( ));
                newDeposit.signalAll( );
            }
            finally{ lock.unlock( ); }
        }
    }
}
```

创建lock对象
创建lock的Condition对象newDeposit

while循环{//必须是while，不能用if
只要余额小于取钱数额，就调用newDeposit.wait，使得当前线程（进入withDraw方法的线程）被挂起；
如果当前线程被唤醒，如果余额还小于取钱数额，继续等待
}
当执行到while循环的下一条语句，余额一定>=取钱数额

newDeposit.wait会导致当前线程被挂起**同时锁被释放（和sleep不一样）**，否则存钱线程永远没机会进入deposit方法

进入deposit的是另外一个线程，往账户存钱后，调用newDeposit.signalAll去唤醒所有因调用newDeposit.wait而被挂起的线程（二者配套使用）

30.4 线程同步-线程协作

```
public class ThreadCooperation {

    public static class DepositTask implements Runnable {
        public void run() {
            try { // Purposely delay it to let the withdraw method proceed
                while (true) {
                    account.deposit((int)(Math.random() * 10) + 1);
                    Thread.sleep(1000);
                }
            }
            catch (InterruptedException ex) { ex.printStackTrace();}
        }
    }

    public static class WithdrawTask implements Runnable {
        public void run() {
            while (true) {
                account.withdraw((int)(Math.random() * 10) + 1);
            }
        }
    }


}
```

存钱线程任务，死循环，每隔1秒存一次钱，每次存钱数量随机
调用account.deposit

取钱线程任务，死循环，每次取钱数量随机
调用account.withDraw

30.4 线程同步-线程协作

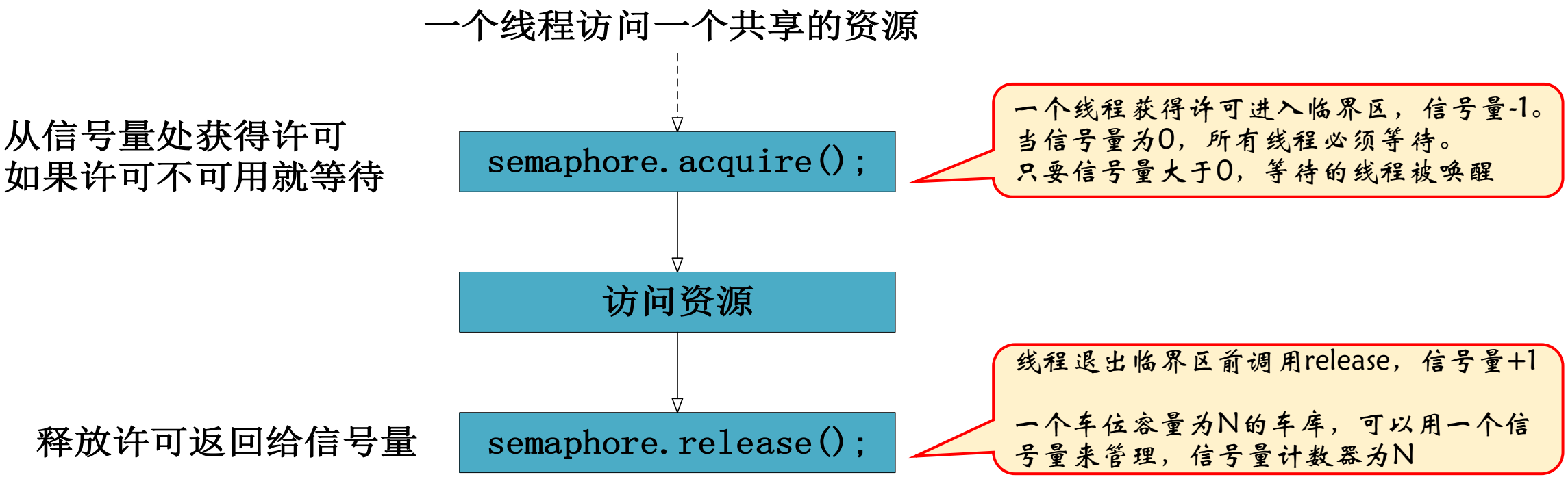
```
public class ThreadCooperation {

    private static Account account = new Account();

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);
        executor.execute(new DepositTask());
        executor.execute(new WithdrawTask());
        executor.shutdown();
        System.out.println("Thread 1\t\tThread 2\t\t\tBalance");
    }



}
```

二个线程都访问account对象，account对象就是竞争资源

创建取钱线程和存钱线程并启动

30.5 信号量

- 信号量用来限制访问一个共享资源的线程数，是一个有计数器的锁
- 访问资源之前，线程必须从信号量获取许可
- 访问完资源之后，该线程必须将许可返回给信号量



30.5 信号量

- 为了创建信号量，必须确定许可的数量（计数器最大值），同时可选用公平策略
- 任务通过调用信号量的acquire()方法来获得许可，信号量中可用许可的总数减1
- 任务通过调用信号量的release()方法来释放许可，信号量中可用许可的总数加1

java.util.coocurrent.Semaphore
+Semaphore(numberOfPermits:int)
+Semaphore(numberOfPermits:int, fair:boolean)
+acquire():void
+release():void

创建一个具有指定数量许可的信号量，公平性策略参数为假
创建一个具有指定数量许可，以及公平性策略的信号量

从该信号量获取一个许可，如果许可不可用，线程将被阻塞，直到一个许可可用
释放一个许可返回给信号量

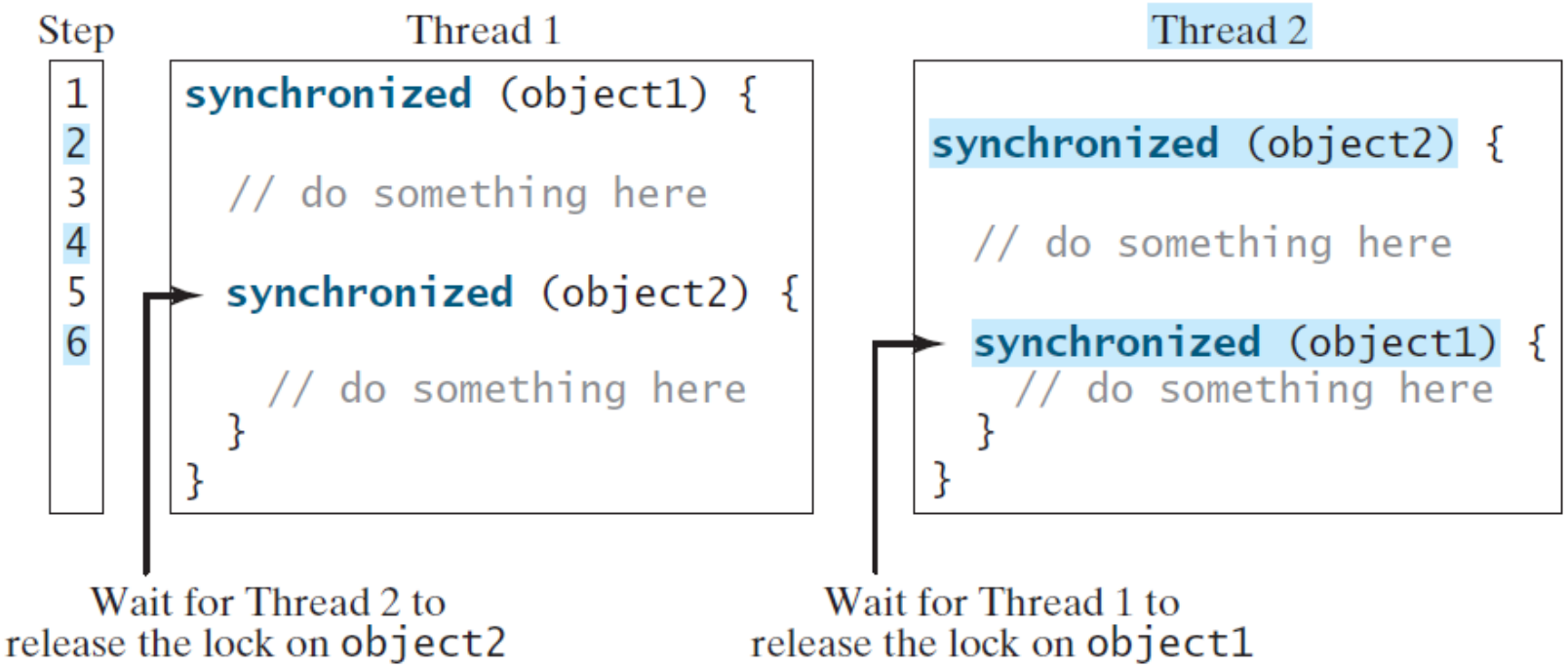
30.5 信号量

```
import java.util.concurrent.Semaphore;
  // An inner class for account
  private static class Account {
    // Create a semaphore
    private static Semaphore semaphore = new Semaphore(1);
    private int balance = 0;
    public int getBalance() {return balance;}

    public void deposit(int amount) {
      try {
        semaphore.acquire();
        int newBaance=balance+amount
          Thread.sleep(5);
        balance=new Balance;
      }
      finally {
        semaphore.release();
      }
    }
  }
}
```

一个许可为1的信号量就相当于互斥锁，因此可以用信号量来修改30-5中的Account类

30.5 避免死锁

- 死锁：如图所示，两个线程形成死锁



- 避免死锁：可以采用正确的资源排序来避免死锁
  - 给每一个需要上锁的对象指定一个顺序
  - 确保每个线程都按这个顺序来获取锁

线程2必须先获取object1上的锁，然后才能获取Object2上的锁

30.6 同步集合（Synchronized Collection）

- Java集合框架 包括：List 、Set 、Map接口及其具体子类，都不是线程安全的。
- 集合框架中的类不是线程安全的， 可通过为访问集合的代码临界区加锁或者同步等方式来保护集合中的数据
- Collections类提供6个静态方法来将集合转成同步版本（即线程安全的版本）
- 这些同步版本的类都是线程安全的，但是迭代器不是，因此使用迭代器时必须同步：

  synchronized(要迭代的集合对象){// 迭代}

java.util.Collections	
+synchronizedCollection(c:Collection):Collection	从一个给定的合集返回一个同步集合
+synchronizedList(list:List):List	从一个给定的线性表返回一个同步线性表
+synchronizedMap(m:Map):Map	从一个给定的映射表返回一个同步映射表
+synchronizedSet(s:Set):Set	从一个给定的集合返回一个同步集合
+synchronizedSortedMap(s:SortedMap):SortedMap	从一个给定的排序映射表返回一个同步排序映射表
+synchronizedSortedSet(s:SortedSet):SortedSet	从一个给定的排序集合返回一个同步排序集合

30.7 内部类

- 内部类也称为嵌套类，是在一个类的内部定义的类。通常一个内部类仅被其外部类使用时,同时也不想暴露出去，才定义为内部类。JDK16以前，内部类不能定义在方法中。但是JDK16以后方法里也可以类 (称为方法内部类，本课程不做介绍，不作要求掌握)
  - 内部类分为实例内部类和静态内部类
- 实例内部类内部不允许定义静态成员（JDK16以前）。从JDK16开始，实例内部类可以定义静态成员了。创建实例内部类的对象时需要使用 外部类的实例变量.new 实例内部类类名( )。(即只有当有了外部类的实例，才能实例化 实例内部类的对象)
- 静态内部类用static定义，其内部允许定义实例成员和静态成员。
- 静态内部类的方法不能访问外部类的实例成员变量。
- 创建静态内部类的对象时需要使用new 外部类.静态内部类( )

30.7 内部类

```
class Wrapper{
   private int x=0;
   private static int z = 0;
   //内部静态类
   static class A{
      int y=0;
      static int q=0; //可以定义静态成员，
      //不能访问外部类的实例成员x，可访问外部类静态成员z
      int g() { return ++q + ++y + ++z; }
   }
   //内部实例类,也定义静态成员（JDK16以后）
   //内部实例类可访问外部类的静态成员如z，实例成员如x
   class B{
      int y=0;
      public int g( ) {
         x++; y++;z++;
         return x+y;
      }
      public int getX(){return x;}
      //从JDK16开始，内部实例类可以定义静态成员
      static void f(){}
   }
}
```

```
public static void main(String[] args){//和class Wrapper同一个JAVA文件，即同一个包
      Wrapper w = new Wrapper(); //w.x = 0;
      //创建内部静态类实例
      Wrapper.A a = new Wrapper.A();          //a.y=0, a.q=0;
      Wrapper.A b = new Wrapper.A();          //b.y=0, b.q=0;
      a.g();
      //a,b的实例成员彼此无关，因此执行完a.g()后，a.y = 1, b.y = 0;
      //a,b共享静态成员q，所以a.q=b.q = 1;

      //创建内部实例类实例
      //不能用new Wrapper.B();必须通过外部类对象去实例化内部类对象
      Wrapper.B  c = w.new B(); //类型声明还是外部类.内部类
      c.y=0;
      c.g(); //c.y = 1 ,c.gextX() = 1

      //在外部类体外面，不能通过内部类对象访问外部类成员，只能在内部类里面访问，
      //编译器在这里只能看到内部类成员
      //   System.out.println(a.z); //错误
      //   System.out.println(c.x); //错误
      //不能通过c直接访问外部类的x，可通过c.gextX()
      System.out.println(c.getX());
}
```

内部类可以被成员访问控制符修饰（私有、缺省、保护、公有的），访问控制规则和类成员访问控制一样

一个内部类被编译成名为OuterClassName$InnerClassName的类

30.7 内部类

```
//定义Message接口
public interface MessageHandler {
    void handle(String message);
}
```

- 内部类作用：如果一个类A仅仅被某一个类B使用，且A无需暴露出去，可以把A作为B的内部类实现，内部类也可以避免名字冲突：因为外部类多了一层名字空间的限定。例如类Wrapper1、Wrapper2可以定义同名的内部类MessageHandlerImpl而不会导致冲突

```
public class Wrapper1 {
  //定义内部类MessageHandlerImpl实现Message接口
  class MessageHandlerImpl implements MessageHandler {
    @Override
    public void handle(String message) { System.out.println(message);}
  }

  //Wrapper1的实例变量
  private MessageHandler handler = new MessageHandlerImpl();
  public void sendMessage(String message){
    handler.handle(message);
  }
  public static void main(String[] args){
    new Wrapper1().sendMessage("Message from wrapper1");
  }
}
```

实例化内部类MessageHandlerImpl实例等价于this.new MessageHandlerImpl();

```
public class Wrapper2 {
  //定义内部类MessageHandlerImpl实现Message接口
  class MessageHandlerImpl implements MessageHandler {
    @Override
    public void handle(String message) { System.out.println(message);}
  }

  //Wrapper2的实例变量
  private MessageHandler handler = new MessageHandlerImpl();
  public void sendMessage(String message){
    handler.handle(message);
  }
  public static void main(String[] args){
    new Wrapper2().sendMessage("Message from wrapper2");
  }
}
```

30.7 匿名内部类：没有名字的内部类

- 匿名内部类可以简化编程。简化时使用匿名内部类的父类或者接口代替匿名内部类。

```
public class Wrapper3 {
  //Wrapper1的实例变量
   private MessageHandler handler = null;
   public void setHandler(MessageHandler handler){ this.handler = handler;}
   //定义内部类MessageHandlerImpl实现Message接口
   class MessageHandlerImpl implements MessageHandler {
      @Override
      public void handle(String message){ System.out.println(message);}
   }
   public void init(){
      setHandler(new MessageHandlerImpl());
   }
}
```

把MessageHandlerImpl的完整声明写出，左边代码相当于

```
public class Wrapper3 {
   //其它代码省略
   public void init(){
      setHandler(new class MessageHandlerImpl
                  implements MessageHandler(){
         @Override
         public void handle(String message) { System.out.println(message);}
      });
   }
}
```

```
public class Wrapper3 {
   //其它代码省略
   public void init(){
      setHandler(new MessageHandler(){
         @Override
         public void handle(String message) { System.out.println(message);}
      });
   }
}
```

MessageHandlerImpl这个类名其实不重要，重要的是需要实现MessageHandler接口，因此想去掉类名，这就是匿名内部类。但new后面必须有一个类型名，就用这个类所实现的接口名作为匿名内部类的类名。

30.7 匿名内部类：没有名字的内部类

- 匿名内部类可以简化编程。简化时使用匿名内部类的父类或者接口代替匿名内部类。

```
public class Wrapper3 {
  //Wrapper1的实例变量
   private MessageHandler handler = null;
   public void setHandler(MessageHandler handler){ this.handler = handler;}
   //定义内部类MessageHandlerImpl实现Message接口
   class MessageHandlerImpl implements MessageHandler {
      @Override
      public void handle(String message){ System.out.println(message);}
   }
   public void init(){
      setHandler(new MessageHandlerImpl());
   }
}
```

```
public class Wrapper3 {
   //其它代码省略
   public void init(){
      setHandler(new MessageHandler(){
         @Override
         public void handle(String message) { System.out.println(message);}
      });
   }
}
```

new一个内部匿名类对象时，new 后面直接用这个匿名内部类的父类或者所实现接口作为类型

```
new MessageHandler(){
        @Override
        public void handle(String message) { System.out.println(message);}
    });
```

实例化一个实现了MessageHandler接口的匿名内部类对象，作为setHandler方法的实参

30.7 匿名内部类

- 匿名内部类可以简化编程。简化时使用匿名内部类的父类或者所实现接口代替匿名内部类名字，作为new后面的类型。

匿名内部类的语法如下所示：

```
new SuperClassName/InterfaceName() {
  // Implement or override methods in superclass or interface

  // Other methods if necessary
}
```

- 匿名内部类总是使用父类的无参构造方法产生实例，对于接口使用Object（）。
- 匿名内部类必须实现父类或者接口的所有抽象方法。事件处理接口通常只有1个方法。
- 一个匿名内部类被编译成OuterClassName$n.class,如Test$1.class, Test$2.class

30.8 Lambda表达式

- Lambda表达式可以进一步简化事件处理的程序编写
- 编译器会将lamda表达式看待为匿名内部类对象，将这个对象理解为实现了MessageHandler接口的实例。下面例子中因为MessageHandler接口定义了参数为String类型的方法handle，因此编译器可以推断参数message的类型为String，并且message->{ }中右边的{ }就是handle方法的方法体。
- MessageHandler接口只有一个方法，只有一个方法的接口称为功能接口（函数式接口），每个 Lambda 表达式都能隐式地赋值给函数式接口，lamda表达式中的{ }就是函数式接口中接口方法的方法体。

```
setHandler(new MessageHandler(){
        @Override
        public void handle(String message){
            System.out.println(message);
        }
});
```

匿名内部类实例作为实参

```
setHandler(message ->{
            System.out.println(message);
});
```

Lambda表达式作为实参

30.8 Lambda表达式

- Lambda表达式本质上更像匿名函数。
- Java里规定Lambda表达式只能赋值给函数式接口。
- Lambda表达式的语法为：

  (type1 para1, ..., typen paran)->expression 或者

  (type1 para1, ..., typen paran)->{ 一条或多条语句}

```
(int a, int b) -> { return a + b; }
() -> System.out.println("Hello World")
(String s) -> { System.out.println(s); }
() -> 42
() -> { return 3.1415 ;}
```

- 当把Lambda表达式赋值给函数式接口时， Lambda表达式的参数的类型是可以推断的；如果只有一个参数，则可以省略圆括弧。从而使Lambda表达式简化为：

  e->处理e的expression 或者

  e->{ 处理e的statements; }

30.8 Lambda表达式

- Lambda 表达式的结构
  - 一个 Lambda 表达式可以有零个或多个参数
  - 参数的类型既可以明确声明，也可以根据上下文来推断。例如：(int a)与(a)效果相同（当可以推断类型时）
  - 所有参数需包含在圆括号内，参数之间用逗号相隔。例如：(a, b) 或 (int a, int b) 或 (String a, int b, float c)
  - 空圆括号代表参数集为空。例如：() -> 42
  - 当只有一个参数，且其类型可推导时，圆括号（）可省略。例如：a -> {return a*a；}
  - Lambda 表达式的主体可以是表达式或者是block，如果是表达式，不能有{}；如果是block，则必须加 { }

30.8 Lambda表达式

- 每个 Lambda 表达式都能隐式地赋值给函数式接口
  - Runnable接口就是函数式接口，里面定义接口方法void run( )，我们可以通过 Lambda 表达式创建一个接口实例 。`Runnable r = ( ) -> System.out.println("hello world");`
    - 上面语句的含义是：将一个实现了Runnable接口的类的实例赋值给Runnable接口引用r， Lambda 表达式的主体就是接口方法void run( )的具体实现
  - 当不是显式赋值给函数式接口时，编译器会自动解释这种转化：

```
new Thread(
        () -> System.out.println("hello world")
).start();
```

  - 在上面的代码中，编译器会自动推断：根据线程类的构造函数签名 public Thread(Runnable r) { }，将该 Lambda 表达式赋给 Runnable 接口。

30.8 Lambda表达式

- 函数式接口定义好后，我们可以在 API 中使用它，同时利用 Lambda 表达式。

```
//定义一个函数式接口
public interface WorkerInterface {
        public void doSomeWork();
}

public class WorkerInterfaceTest {
        public static void exec(WorkerInterface worker) {
                worker.doSomeWork();
        }

        public static void main(String [] args) {
                //invoke doSomeWork using Annonymous class
                exec( new WorkerInterface() {
                        @Override public void doSomeWork() {
                        System.out.println("Worker invoked using Anonymous class"); }
                } );
                //invoke doSomeWork using Lambda expression
                exec( () -> System.out.println("Worker invoked using Lambda expression") );
        }
}
```

new一个实现了WorkerInterface接口的匿名类对象传入exec方法

将一个Lambda表达式传入exec方法

30.8 Lambda表达式

```
public class LambdaDemo {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);
        //传入匿名内部类，Runnable接口实例
        executor.execute(new Runnable() {
            @Override
            public void run() { System.out.println("Runnable 1"); }
        });
        //传入Lambda表达式，Runnable接口实例，右边是Statements，必须放在{}
        executor.execute(()->{System.out.println("Runnable 2");});
        //传入Lambda表达式，Runnable接口实例，右边是expression，不能放在{}里，不带;
        executor.execute(()->System.out.println("Runnable 3"));

        executor.shutdown();
    }
}
```

30.8 Lambda表达式

- Lambda 神奇功能
  - 计算给定数组中每个元素平方后的总和。请注意，Lambda 表达式只用一条语句就能达到此功能，这也是 MapReduce 的一个初级例子。我们使用 map() 给每个元素求平方，再使用 reduce() 将所有元素计入一个数值：
  - java.util.stream.Stream 接口包含许多有用的方法，能结合 Lambda 表达式产生神奇的效果。

```
//Old way:
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7);
int sum = 0;
for(Integer n : list) {
            int x = n * n;
            sum = sum + x;
 }
System.out.println(sum);

//New way:
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7);
 int sum = list.stream().map(x -> x*x).reduce((x,y) -> x + y).get();
System.out.println(sum);
```

30.8 Lambda表达式

- Java Lambda 表达式的由来
  - Java 中的一切都是对象（除了基本数据类型），即使数组也是一种对象，每个类创建的实例也是对象。在 Java 中定义的函数或方法不可能完全独立，也不能将方法作为参数或返回一个方法给实例。为此，Java 8 增加了一个语言级的新特性，名为 Lambda 表达式。
  - 在函数式编程语言中，函数是一等公民，它们可以独立存在，你可以将其赋值给一个变量，或将他们当做参数传给其他函数。JavaScript 是最典型的函数式编程语言（当然也是面向对象的）。函数式语言提供了一种强大的功能——闭包。当一种编程语言支持函数返回类型为函数时，这种语言天然就支持闭包。
  - Java虽然不支持函数返回类型为函数，但可以用匿名内部类实现闭包，但这种闭包多了一个限制：要求捕获的的自由变量必须是final的。用Lambda表达式同样如此： Lambda表达式捕获的自由变量必须是final的
  - 为什么Java里的闭包多了这个限制：在Java的经典著作《Effective Java》、《Java Concurrency in Practice》大神们这么解释：如果Java闭包捕获的自由变量是非final的，会导致线程安全问题。 Python和Javascript则不用考虑这样的问题，所以它的闭包捕获的自由变量是可以任意修改的。

30.8 Lambda表达式

- 闭包(Closure)
  - 闭包(Closure)并不是一个新鲜的概念，很多函数式语言中都使用了闭包。例如在JavaScript中，当你在内嵌函数中使用外部函数作用域内的变量时，就是使用了闭包。用一个常用的类比来解释闭包和类（Class）的关系：类是带函数的数据，闭包是带数据的函数。

  - 闭包的本质：代码块+上下文

30.8 Lambda表达式

- 闭包(Closure)
  - 闭包的本质：代码块+上下文

```
function generator() {
        var i = 0; //被闭包捕获的自由变量
        return function() {//返回一个嵌套匿名函数，使用了外部函数作用域里变量i
                return i++;
        };
}
var gen1 = generator();     //得到一个自然数生成器，        gen1是一个闭包
var gen2 = generator();     //得到另一个自然数生成器，      gen2是一个闭包
var r1 = gen1();            //r1 = 0
var r2 = gen1();            //r2 = 1
var r3 = gen2();            //r3 = 0
var r4 = gen2();            //r4 = 1
console.log(r4);
```

30.8 Lambda表达式

- 闭包(Closure)
  - 闭包中捕获的自由变量有一个特性，就是它们不在父函数返回时释放，而是随着闭包生命周期的结束而结束。gen1和gen2分别使用了相互独立的变量i（在gen1的i自增1的时候，gen2的i并不受影响，反之亦然），只要gen1或gen2这两个变量没有被JavaScript引擎垃圾回收，他们各自的变量i就不会被释放。

```
function generator() {
        var i = 0; //被闭包捕获的自由变量
        return function() {//返回一个嵌套匿名函数，使用了外部函数作用域里变量i
                return i++;
        };
}
var gen1 = generator();     //得到一个自然数生成器，        gen1是一个闭包
var gen2 = generator();     //得到另一个自然数生成器，      gen2是一个闭包
var r1 = gen1();            //r1 = 0
var r2 = gen1();            //r2 = 1
var r3 = gen2();            //r3 = 0
var r4 = gen2();            //r4 = 1
console.log(r4);
```

闭包被创造出来显然是因为有场景需要的。一个最为普遍和典型的使用场合是：延迟执行。我们可以把一段代码封装到闭包里，你可以等到“时机”成熟时去执行它。

30.8 Lambda表达式

```
interface Closure<T>{
   T get();
}

class Dog{

}
```

```
Closure<Dog> c1 = testClosure1();  //返回闭包1
Closure<Dog> c2 = testClosure1();  //返回闭包2

//二个闭包里面还有dog，而且是不同的dog，这个时候testClosure1方法已经结束了
//二个闭包都捕获了局部变量dog，延长了dog的生命周期
System.out.println(c1.get() == c2.get()); //false
```

```
public static Closure<Dog> testClosure1(){
    //匿名内部类需要访问匿名内部类所在方法中的局部变量的时候，
    //必须给局部变量加final进行修饰
     final Dog  dog = new Dog();
     //Java里，匿名内部类方法要捕获的外部闭包环境的自由变量必须是final的
     return new Closure<Dog>() {
        @Override
        public Dog get() {
           //匿名对象的get方法捕获了外面的自由变量dog，
           //使得testClosure1中的局部变量dog生命周期延长
           return dog;
        }
     };
 }
```

30.8 Lambda表达式

```
interface Closure<T>{
   T get();
}

class Dog{

}
```

```
Closure<Integer> c3 = testClosure2();  //返回闭包1
Closure<Integer> c4 = testClosure2();  //返回闭包2

//二个闭包里面还有Integer，而且是不同的Integer
 System.out.println(c3.get() == c4.get());
```

```
public static Closure<Integer> testClosure2(){
    //从JDK1.8开始8加了一个语法糖：在lambda表达式以及匿名类内部，如果捕获某局部变量，则直接将其视为final。
    int i = 0; //不用final修饰，但是一旦在lambda表达式里修改i，立刻编译报错，换句话说，捕获的自由变量还是不可改
    return () -> { return i;}; //注意装箱操作，返回的是Integer对象。
}
```