

多线程

本章内容



- 知识点1: 进程和线程基本概念
- 知识点2: 多线程的优缺点
- 知识点3: 主线程
- 知识点4: 创建线程的三种方式
- 知识点5: 线程的状态
- 知识点6: 线程的调度与控制
- 知识点7: 线程同步
- 知识点8: 线程死锁
- 知识点9: 线程通讯
- 知识点10: 线程的生命周期

本章内容



- 知识点11: 线程池调度器
- 知识点12: 信号量
- 知识点13: Lock对象
- 知识点14: ThreadLocal

知识点1-进程和线程基本概念

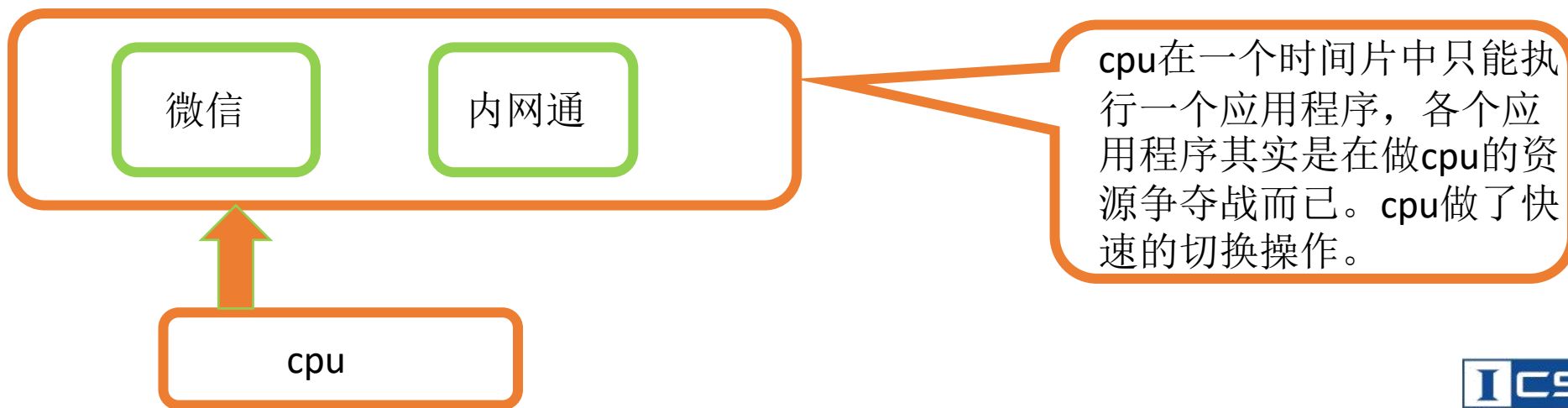
• 进程：

- 运行中的应用程序叫进程，每个进程运行时，进程负责了内存空间的划分。它是系统进行资源分配和调度的一个独立单位。
- 操作系统都是支持多进程的



Windows是多任务的操作系统，那么Windows是同时运行多个应用程序吗？

- 从宏观角度：Windows确实是在同时运行多个应用程序。
- 从微观角度：cpu是做了一个快速切换执行的动作，由于速度太快，所以我们感觉不到切换而已。



知识点1-进程和线程基本概念



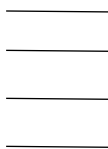
• 线程：

- 线程是轻重级的进程，是进程中一个负责程序执行的控制单元
- 线程是由进程创建的（寄生在进程中）
- 一个进程可以拥有多个线程
- 线程有几种状态（新建new，就绪Runnable，运行Running，阻塞Blocked，死亡Dead）
- 开启多个线程是为了同时运行多部分代码，每个线程都有自己的运行的内容，这个内容可以称线程要执行的任务。



打开360卫士，就是打开一个进程，一个进程里面有很多代码，这些代码就是谁来执行的呢？线程来执行这些代码。

360卫士

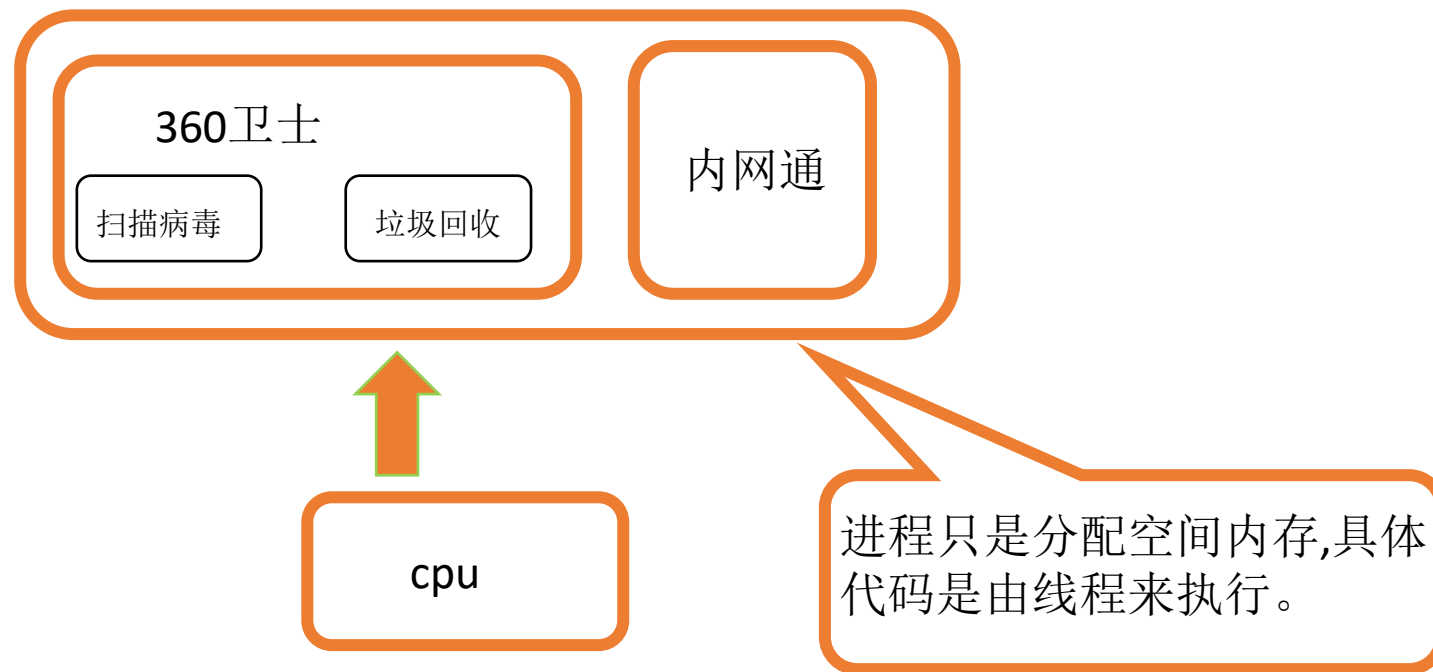
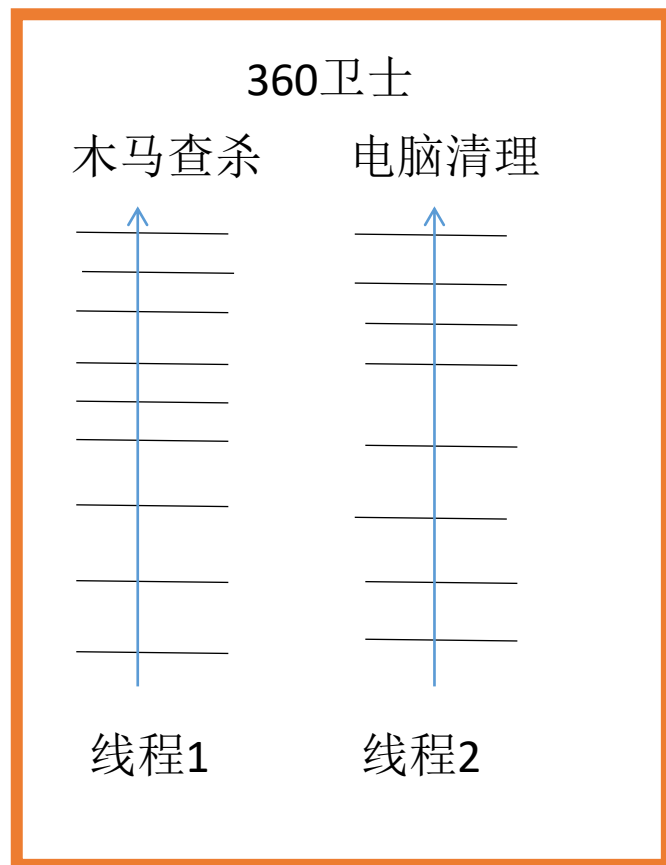


线程

知识点1-进程和线程基本概念

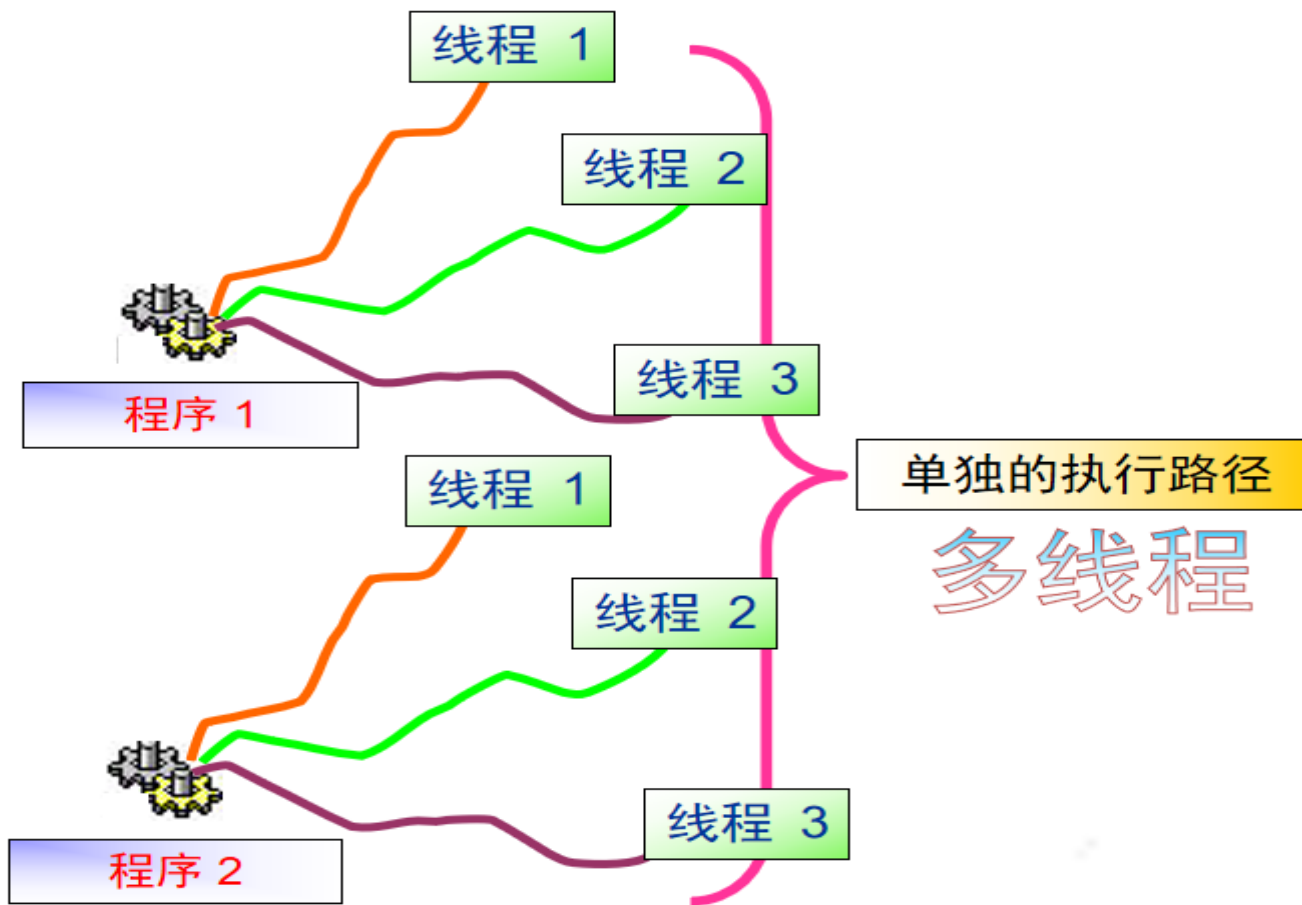


- 多线程：在一个进程中有多线程同时在执行不同的任务。



知识点1-进程和线程基本概念

- 多线程是指程序中包含多个执行流，即在一个程序中可以同时运行多个不同的线程来执行不同的任务，也就是说允许单个程序创建多个并行执行的线程来完成各自的任务
- 如：迅雷 QQ。



知识点1-进程和线程基本概念

- 同时执行多个任务的优势：妖怪，看我的三头六臂

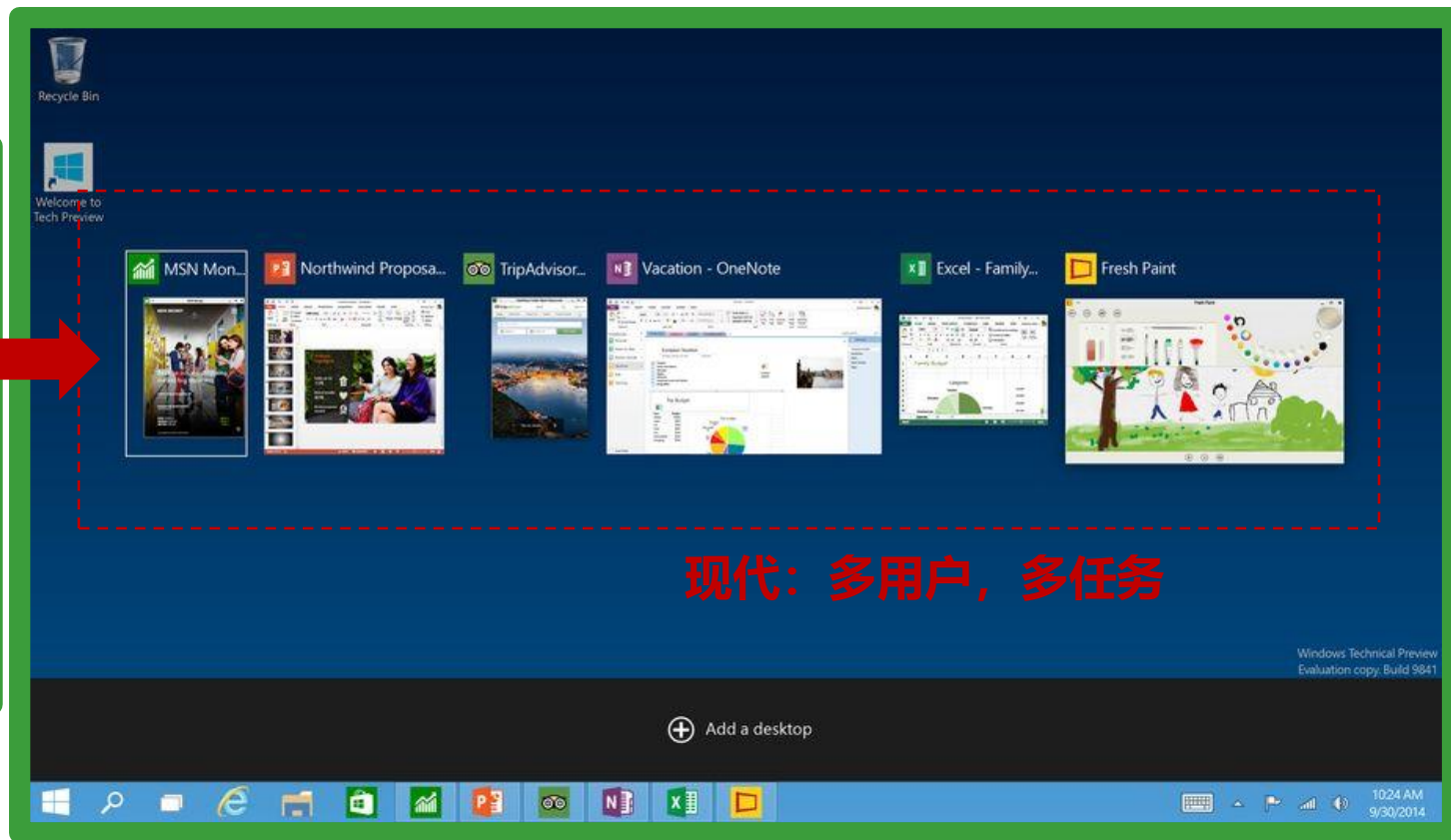


知识点1-进程和线程基本概念

- 计算机操作系统的进化史：
- 当前操作系统支持多线程

```
Starting MS-DOS 7.1...  
  
The following file is missing or corrupted: .\HIMEM.SYS  
  
Microsoft(R) MS-DOS 7.1  
(C)Copyright Microsoft Corp 1981-1999.  
C:\>_
```

早期：单用户，单任务



现代：多用户，多任务



知识点2-多线程的优缺点



- 多线程优点:

- Java支持编写多线程的程序;
- 多线程最大的好处在于可以同时并发执行多个任务;
- 多线程可以最大限度地减低CPU的闲置时间,从而提高CPU的利用率。

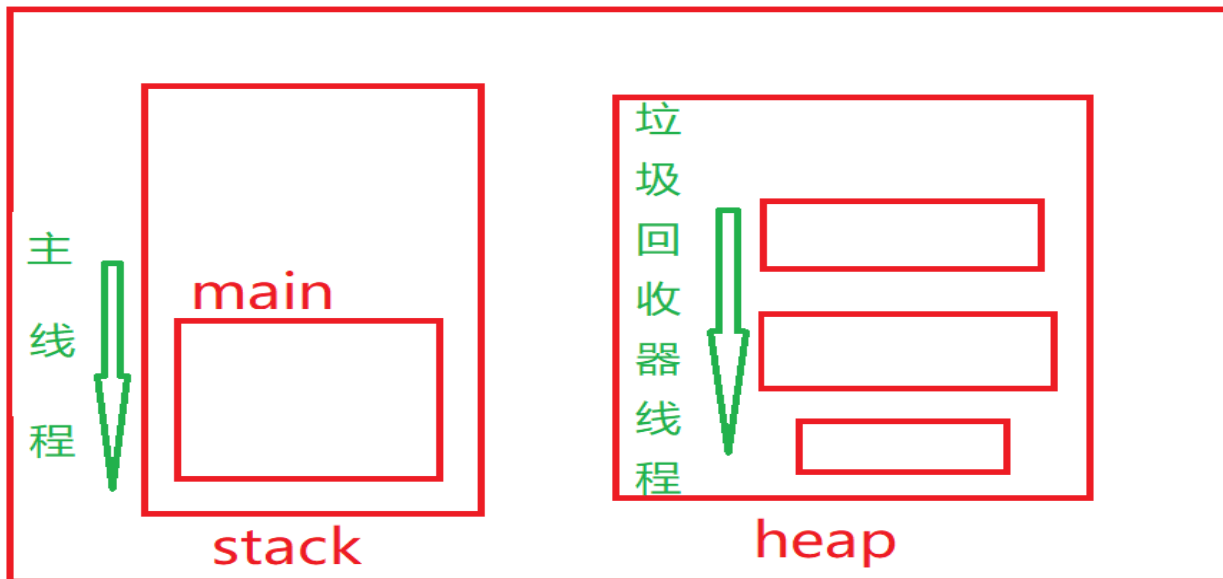
- 多线程缺点:

- 线程也是程序,所以线程需要占用内存,线程越多占用内存也越多;
- 多线程需要协调和管理,所以需要CPU时间跟踪线程;
- 线程之间对共享资源的访问会相互影响,必须解决竞用共享资源的问题;
- 线程太多会导致控制太复杂,最终可能造成很多Bug

知识点3-主线程



- Java支持多线程
- 主线程的特殊之处在于：
 - 任何一个Java程序启动时，一个线程立刻运行，它执行main方法，这个线程称为程序的主线程。
 - 一个Java应用程序至少有两个线程，一个是主线程负责main方法代码执行；一个是垃圾回收器线程，负责了回收垃圾。

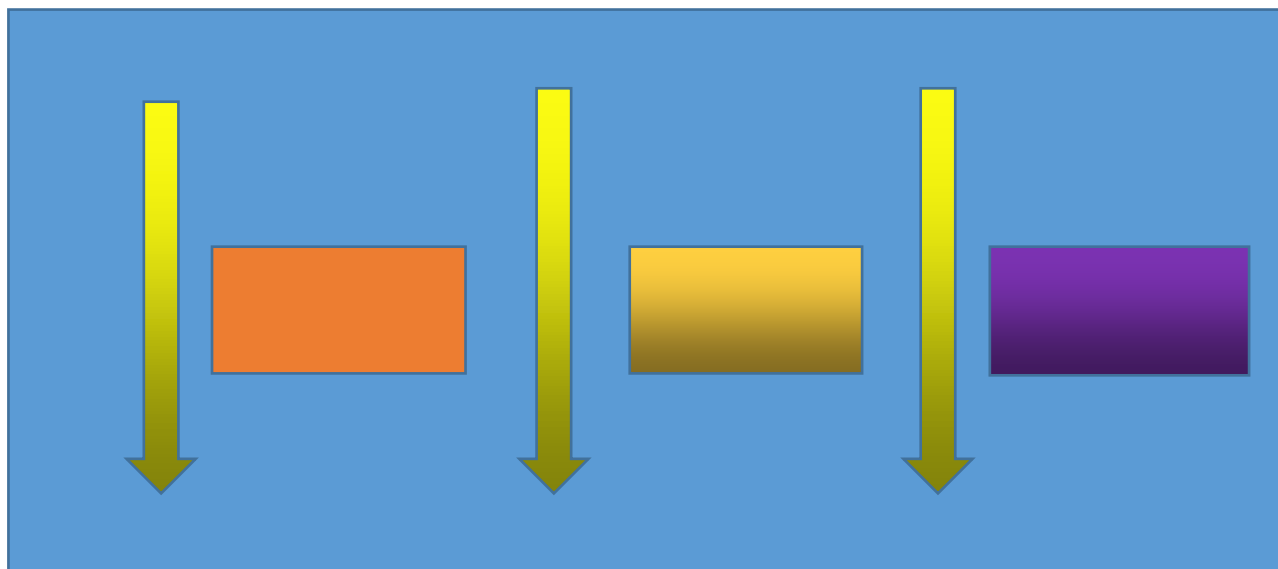


知识点4-创建线程的三种方式



- 方式一：继承Thread类
- 方式二：实现Runnable接口
- 方式三：实现Callable接口

并发（一同，一起）执行



知识点4-创建线程的三种方式-继承Thread类

- 继承Java.lang.Thread类，并重写run() 方法。

定义：

```
class MyThread extends Thread {  
    //把自定义线程的任务代码写在run方法中  
    public void run( ) {  
        /* 覆盖该方法*/  
    }  
}
```

创建多线程并启动线程：

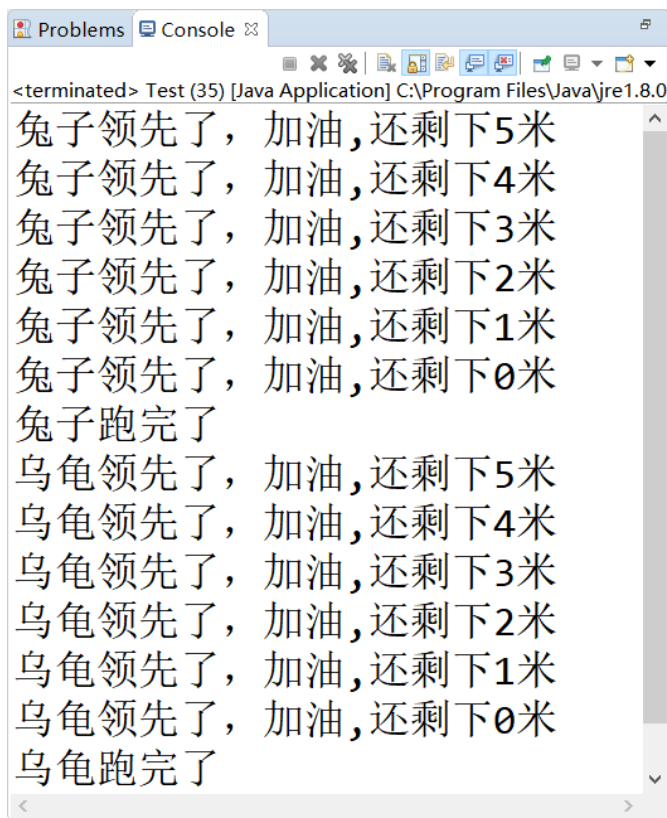
```
MyThread thread = new MyThread();  
thread.start();
```



- 1.start()方法的调用后并不是立即执行多线程代码，而是**使该线程变为就绪状态**，什么时候运行是由操作系统调度决定的。
- 2.调度后，**自动调用其run方法**，run方法是等着被自动调用的(**run方法千万不能直接调用**，如果直接调用就相当于调用了一个普通方法而已，并没有开启线程)。

知识点4-创建线程的三种方式-继承Thread类

- 案例1：打印输出0-100的数字，创建两个线程交替执行。
- 案例2：模拟龟兔赛跑



```
<terminated> Test (35) [Java Application] C:\Program Files\Java\jre1.8.0
兔子领先了, 加油, 还剩下5米
兔子领先了, 加油, 还剩下4米
兔子领先了, 加油, 还剩下3米
兔子领先了, 加油, 还剩下2米
兔子领先了, 加油, 还剩下1米
兔子领先了, 加油, 还剩下0米
兔子跑完了
乌龟领先了, 加油, 还剩下5米
乌龟领先了, 加油, 还剩下4米
乌龟领先了, 加油, 还剩下3米
乌龟领先了, 加油, 还剩下2米
乌龟领先了, 加油, 还剩下1米
乌龟领先了, 加油, 还剩下0米
乌龟跑完了
```



知识点4-创建线程的三种方式-实现Runnable接口



- 实现Java.lang.Runnable接口，并重写run() 方法;

定义多线程类:

```
class MyThread implements Runnable{  
    public void run( ) {  
        /* 实现该方法*/  
    }  
}
```

创建多线程对象并启动多线程:

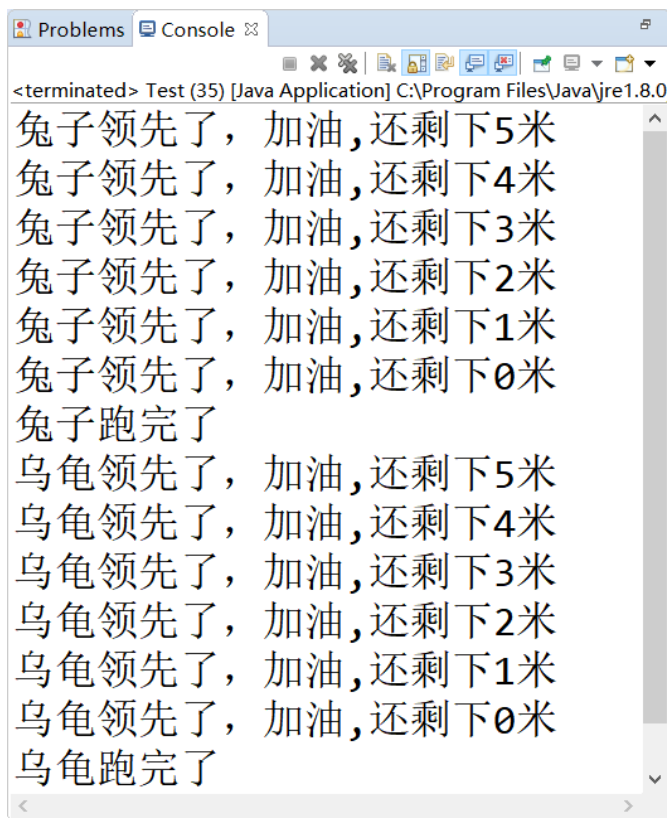
```
MyThread a= new MyThread ();  
//创建一个线程作为外壳，将Runnable实现类对象包起来  
Thread thread1 = new Thread(a); //创建线程  
Thread thread2 = new Thread(a); //创建线程  
thread1.start();//start开启线程，并自动调用run()方法执行  
thread2.start();//不可多次调用
```



注意：Runnable接口的存在主要是为了解决Java中不允许多继承的问题。

知识点4-创建线程的三种方式-实现Runnable接口

- 案例1：打印输出0-100的数字，创建两个线程交替执行。
- 案例2：模拟龟兔赛跑



```
<terminated> Test (35) [Java Application] C:\Program Files\Java\jre1.8.0
兔子领先了, 加油, 还剩下5米
兔子领先了, 加油, 还剩下4米
兔子领先了, 加油, 还剩下3米
兔子领先了, 加油, 还剩下2米
兔子领先了, 加油, 还剩下1米
兔子领先了, 加油, 还剩下0米
兔子跑完了
乌龟领先了, 加油, 还剩下5米
乌龟领先了, 加油, 还剩下4米
乌龟领先了, 加油, 还剩下3米
乌龟领先了, 加油, 还剩下2米
乌龟领先了, 加油, 还剩下1米
乌龟领先了, 加油, 还剩下0米
乌龟跑完了
```



知识点4-创建线程的三种方式-实现Callable接口



- 使用Callable和Future创建线程

使用Callable创建线程和Runnable接口方式创建线程比较相似，不同的是，Callable接口提供了一个call() 方法作为线程执行体，而Runnable接口提供的是run()方法，同时，call()方法可以有返回值，而且需要用FutureTask类来包装Callable对象。

- 步骤：

- 1、创建Callable接口的实现类，实现call() 方法
- 2、创建Callable实现类实例，通过FutureTask类来包装Callable对象，该对象封装了Callable对象的call()方法的返回值。
- 3、将创建的FutureTask对象作为target参数传入，创建Thread线程实例并启动新线程。
- 4、调用FutureTask对象的get方法获取返回值。

知识点4-创建线程的三种方式-实现Callable接口



• 示例

```
public class CallDemo {
    public static void main(String[] args) {
        // (1)创建Callable实现类的实例化对象
        CallableThreadDemo ctd = new CallableThreadDemo();
        // (2)创建FutureTask对象，并将Callable对象传入FutureTask的构造方法中
        // 注意：Callable需要依赖FutureTask，用于接收运算结果。
        // 一个产生结果，一个拿到结果。FutureTask是Future接口的实现类。
        FutureTask<Integer> result = new FutureTask<>(ctd);
        // (3)实例化Thread对象，并在构造方法中传入FutureTask对象
        Thread t = new Thread(result);
        // (4)启动线程
        t.start();
        try {
            Integer sum = result.get(); // 检索结果
            System.out.println(sum);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

```
class CallableThreadDemo implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        int sum = 0;
        for (int i = 0; i < 100; i++) {
            sum += i;
        }
        return sum;
    }
}
```

知识点4-创建线程的三种方式-比较



- 继承Thread类：

- 优势：Thread类已实现了Runnable接口，故使用更简单
- 劣势：无法继承其它父类

- 实现Runnable接口：

- 优势：可以继承其它类
- 劣势：编程方式稍微复杂，多写一行代码

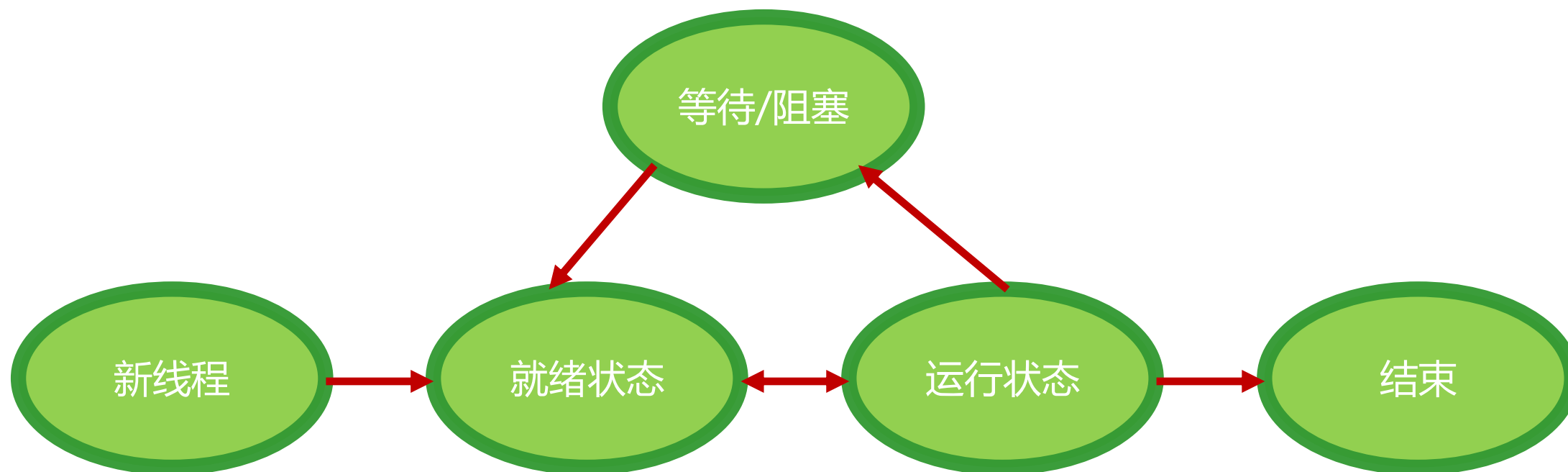
- 实现Callable接口：

- 类似于Runnable，两者都是为那些其实例可能被另一个线程执行的类设计的，方法可以有返回值，并且可以抛出异常。但是Runnable不行。

知识点5-线程的状态



- Java中线程状态转换:



知识点5-线程的状态



- Java中线程存在以下几种状态（后续将详细讲解对应代码）：
- **新线程：**
 - 新创建了一个线程对象，此时它仅仅作为一个对象实例存在，JVM没有为其分配CPU时间片和其他线程运行资源。
- **就绪状态：**
 - 在处于创建状态的线程中调用start方法将线程的状态转换为就绪状态。这时，线程已经得到除CPU时间之外的其它系统资源，只等JVM的线程调度器按照线程的优先级对该线程进行调度，从而使该线程拥有能够获得CPU时间片的机会
- **运行状态：**
 - 就绪态的线程获得cpu就进入运行态
- **等待/阻塞：**
 - 阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。
- **死亡状态：**
 - 线程执行完它的任务时，由JVM收回线程占用的资源。

知识点6-线程的调度与控制-Thread类



- 使用线程的步骤:



**按照语言要求编写
不同的线程中需要
执行的指令代码**

知识点6-线程的调度与控制-Thread类



- Thread类提供了较多的构造方法重载版本，以下列出了常用的几个版本：

构造方法	说明
Thread()	创建一个新的线程
Thread(String name)	创建一个指定名称的线程
Thread(Runnable target)	利用Runnable对象创建一个线程，启动时将执行该对象的run方法
Thread(Runnable target, String name)	利用Runnable对象创建一个线程，并指定该线程的名称

- Thread类用于创建和操作线程，其中包括几个很重要的静态方法，用于控制当前线程：

静态方法	说明
static Thread currentThread()	返回对当前正在执行的线程对象的引用
static void sleep(long millis) throws InterruptedException	使当前正在执行的线程休眠（暂停执行）
static void yield()	让出时间片
static boolean interrupted()	判断当前线程是否已经中断

知识点6-线程的调度与控制-Thread类



- Thread类中的常用方法:

常用方法	说明
void start()	启动线程
final void setName(String name)	设置线程的名称
final String getName()	返回线程的名称
final void setPriority(int newPriority)	设置线程的优先级
final int getPriority()	返回线程的优先级
final void join() throws InterruptedException	等待该线程终止。(强行介入)
final boolean isAlive()	判断线程是否处于活动状态
void interrupt()	中断线程

知识点6-线程的调度与控制-启动



- 线程启动

执行start()方法，但是不能多次执行

```
public class CreateThreadByExtends extends Thread {  
    public void run() {  
        System.out.println("FirstThread with extend Thread @ Chinasofti");  
    }  
  
    public static void main(String[] args) {  
        CreateThreadByExtends ctbe = new CreateThreadByExtends();  
        ctbe.start();  
    }  
}
```

知识点6-线程的调度与控制-停止



- 线程的停止远比线程的启动情况复杂

在Thread线程类中提供了stop()方法用于停止一个已经启动的线程，但是它已经被废弃，不建议使用，因为它本质上就是不安全的，它会解锁线程所有已经锁定的监视程序，在这个过程中会出现不确定性，导致这些监视程序监视的对象出现不一致的情况，或者引发死锁。

- 那么应该如何安全的停止一个正在运行的线程呢？

线程对象的run()方法所有代码执行完成后，线程会自然消亡，因此如果需要在运行过程提前停止线程，可以通过改变共享变量值的方法让run()方法执行结束。

知识点6-线程的调度与控制-停止

- 使用共享变量停止线程的示例：

```
public class ThreadStopExample extends Thread {
```

```
    private boolean flag = true;
```

```
    public void stopThread() {
```

```
        flag = false;
```

修改共享变量，促使run()方法中的循环结束，从而完成方法的调用，线程自然消亡

```
    }  
    public void run() {
```

```
        while (flag)
```

当变量为true时，线程会持续运行而不会退出

```
        ;
```

```
    }
```

```
    public static void main(String[] args) throws Exception {
```

```
        ThreadStopExample tse = new ThreadStopExample();
```

```
        tse.start();
```

```
        tse.stopThread();
```

停止线程

程序启动后迅速结束，说明线程已经成功结束

```
    }
```

Console

application] C:\Program Files (x86)\Java\jdk1.7.0_02\bin\javaw.exe (2017年3月14日 下午4:59:24)

```
tse.start();
```

```
//tse.stopThread();
```

不调用停止方法，线程将会持续执行

rogram Files (x86)\Java\jre1.8.0_121\bin\javaw.exe (2017年3月14日 下午5:06:19)

知识点6-线程的调度与控制-停止

- 如果一个线程由于等待某些事件的发生而被阻塞，又该怎样停止该线程呢？
使用Thread提供的**interrupt()方法**，因为该方法虽然不会中断一个正在运行的线程，但是它可以使一个被阻塞的线程抛出一个中断异常，从而使线程提前结束阻塞状态，退出堵塞代码。
- 中断阻塞线程的示例：

程序启动后迅速结束，
说明线程已经成功结束

```
public class StopSleepThread {  
    public static void main(String[] args) {  
        SleepThread st = new SleepThread();  
        System.out.println("线程启动");  
        st.start();  
        st.interrupt();  
    }  
}  
  
class SleepThread extends Thread {  
    boolean flag = true;  
    public void run() {  
        while (flag) {  
            try {  
                Thread.sleep(1000 * 60 * 60);  
            } catch (InterruptedException e) {  
                // TODO Auto-generated catch block  
                System.out.println("线程执行被中断!");  
                flag = false;  
            }  
        }  
    }  
}
```

中断线程，此时线程代码将抛出
InterruptedException异常

```
<terminated> StopSleepThread [Java Application] C:\Program Files (x86)\Java\jre1.8.0_121\bin\javaw.exe (2017年3月14日 下午7:49:  
线程启动  
线程执行被中断!
```

```
st.start();  
Thread.sleep(100);  
st.flag = false;  
// st.interrupt();
```

不中断线程，即便改变共享变量
的值也无效

```
StopSleepThread [Java Application] C:\Program Files (x86)\Java\jre1.8.0_121\bin\javaw.exe (2017年3月14日 下午7:53:39)  
线程启动
```

处理InterruptedException异常时改变共享变量的值

知识点6-线程的调度与控制-停止



- 线程停止总结
- Thread.stop()方法已经过时不再使用
- 定义循环结束标记

标记: `flag = false(true)`

- 使用interrupt（中断）方法:该方法是结束线程的就绪状态,使线程回到运行状态中来

中断线程: `public void interrupt()`

- 停止线程方法一般用: 标记+interrupt()联合使用

知识点6-线程的调度与控制-设置及获取线程的名称



- 设置及获取线程的名称

- 当开启多个线程时，如何区分当前正在运行的线程是哪个？

- 类实现Runnable时用的是Thread.currentThread().getName()

- 类继承Thread时用的是getName()

知识点6-线程的调度与控制-设置线程的优先级



• 设置线程的优先级

- 多个线程处于可运行状态时，将对cpu时间片进行竞争，优先级高的，获取的可能性则高；
- 优先级较高的线程有更多获得CPU的机会，反之亦然；
- 线程优先级用1 ~ 10 表示，10的优先级最高，默认值是5或继承其父线程优先级；
- 通过setPriority和getPriority方法来设置或返回优先级；

• Thread类有如下3个静态常量来表示优先级

- MAX_PRIORITY：取值为10，表示最高优先级。
- MIN_PRIORITY：取值为1，表示最底优先级。
- NORM_PRIORITY：取值为5，表示默认的优先级

```
Thread t1=new Thread(d);  
t2.setPriority(1);           //用数字设置优先级  
t1.setPriority(NORM_PRIORITY); //用静态常量设置优先级  
t2.getPriority();            //获得到优先级  
System.out.println(Thread.currentThread()); //Thread[one,1,main]
```

知识点6-线程的调度与控制-守护线程



- Java中将线程划分为了两类：
 - 用户线程 (User Thread)
 - 守护线程 (Daemon Thread)
- 所谓守护线程，是指在程序运行的时候在后台提供一种通用服务的线程，比如垃圾回收线程就是一个很称职的守护者，并且这种线程并不属于程序中不可或缺的部分。因此，当所有的非守护线程结束时，程序也就终止了，同时会杀死进程中的所有守护线程。反过来说，只要任何非守护线程还在运行，程序就不会终止

方 法	说 明
final void setDaemon(boolean on)	将该线程设置为守护线程
final boolean isDaemon()	判断该线程是否为守护线程

默认创建的线程对象为非守护的用户线程；
要将某个线程设置为守护线程的话，必须在它启动之前调用setDeamon方法；

知识点6-线程的调度与控制-sleep方法

- sleep用法

- 使线程停止运行一段时间，此时将处于阻塞状态。
- 阻塞的时间由指定的毫秒数决定
- Thread.sleep(long millis)和Thread.sleep(long millis, int nanos)静态方法强制当前正在执行的线程休眠（暂停执行），以“减慢线程”。当线程睡眠时，它入睡在某个地方，在苏醒之前不会返回到可运行状态。当睡眠时间到期，则返回到可运行状态，不是运行状态，因此sleep()方法不能保证该线程睡眠到期后就开始执行。

- 线程睡眠的原因：线程执行太快，或者需要强制进入下一轮

- 睡眠的实现：调用静态方法。

```
try {  
    Thread.sleep(1000);  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}
```



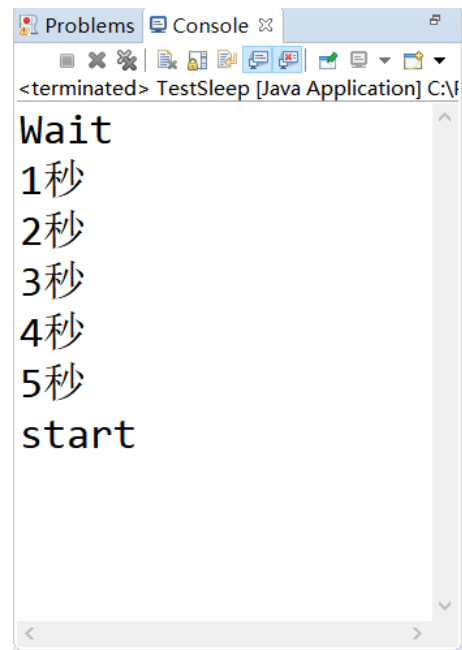
知识点6-线程的调度与控制-sleep方法

- sleep方法示例:

```
public class TestSleep {  
    public static void main(String[] args) {  
        System.out.println("Wait");  
        // 让主线程等待5秒再执行  
        Wait.bySec(5);  
        // 提示恢复执行  
        System.out.println("start");  
    }  
}  
  
class Wait {  
    public static void bySec(long s) {  
        // sleep s个1秒  
        for (int i = 0; i < s; i++) {  
            System.out.println(i + 1 + "秒");  
            try {  
                // sleep 1秒  
                Thread.sleep(1000);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

当前线程阻塞1秒钟，1秒钟后
由阻塞状态转变为可运行状态

如果调用了sleep方法之后，没有其他等待执行的
线程，这个时候当前线程不会马上恢复执行！



知识点6-线程的调度与控制-yield方法



- yield() :

- 出让时间片，但不会阻塞线程，转入可运行状态。

```
public class YeildTest {  
    public static void main(String[] args) {  
        TheThread mt = new TheThread();  
        MyNewThread mnt = new MyNewThread();  
        mt.start();  
        mnt.start();  
    }  
}  
class TheThread extends Thread {  
    public void run() {  
        for (int i = 0; i < 5; i++) {  
            System.out.println("第一个线程的第 " + (i + 1) + "次运行");  
            Thread.yield();  
        }  
    }  
}  
class MyNewThread extends Thread {  
    public void run() {  
        for (int i = 0; i < 5; i++) {  
            System.out.println("第二个线程的第 " + (i + 1) + "次运行");  
            Thread.yield();  
        }  
    }  
}
```

当前线程停止运行，但仍处于可运行状态。可以和其他的等待执行的线程竞争处理器资源

如果调用了yield方法之后，如果没有其他等待执行的线程，这个时候当前线程就会马上恢复执行！

<terminated> YeildTest [Java Application] C:\Program Files\
第一个线程的第 1次运行
第一个线程的第 2次运行
第一个线程的第 3次运行
第一个线程的第 4次运行
第一个线程的第 5次运行
第二个线程的第 1次运行
第二个线程的第 2次运行
第二个线程的第 3次运行
第二个线程的第 4次运行
第二个线程的第 5次运行

知识点6-线程的调度与控制-sleep和yield对比



- sleep和yield对比

	sleep	yield
暂停后的状态	进入被阻塞的状态，直到经过指定时间后，才进入可运行状态	直接将当前线程转入可运行状态
没有其他等待运行的线程	当前线程会等待指定的时间后执行	当前线程有可能会马上恢复执行
等待线程的优先级别	不考虑优先级	优先级相同或更高的线程运行

知识点6-线程的调度与控制-join方法



- join() :

- 阻塞当前线程，强行介入执行。

```
t1.start()
try {
    //强制其它线程等待t1结束后再执行。
    //t1.join();
    //强制其它线程等待5秒后再执行。
    t1.join(5000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
t2.start();
```

知识点7-线程同步-线程安全问题



- 需求：模拟三个窗口同时在售50张票

- 问题1：为什么50张票被卖出了150次？

出现原因：因为num是非静态的，非静态的成员变量数据是在每个对象中都会维护一份数据的，三个线程对象就会有三份。

解决方案：把num票数共享出来给三个对象使用。使用static修饰。

- 问题2：刚刚的问题是解决了，但是为什么会出现两个窗口卖同一张票呢？--出现了线程安全问题

解决方案：一个时间只能由一个线程操作内容---线程同步机制

窗口一售出了第50张票
窗口三售出了第50张票

```
public void run() {  
    while(true) {  
        if (num>0) {  
            System.out.println(this.getName()+"售出了第"+num+"张票");  
            num--;  
        }else {  
            System.out.println("卖完了");  
            break;  
        }  
    }  
}
```

知识点7-线程同步-线程安全问题



- 线程安全问题产生的原因
 - 多个线程操作共享数据
 - 共享数据的线程代码有多条
 - 当线程在执行操作共享数据的多条代码过程中，其它线程参与了运算，就会导致线程的安全问题的产生

知识点7-线程同步-概念

- 线程是一份独立运行的程序，有自己专用的运行栈。线程有可能和其他线程**共享**一些资源，比如，内存，文件，数据库等；
- 多个线程同时读写同一份共享资源时，可能会引起冲突。所以引入线程“**同步**”机制，即各线程间要有先来后到；
- 同步就是**排队**：几个线程之间**要排队**，一个个对共享资源进行操作，而不是同时进行操作；
- 确保一个时间点只有一个线程访问共享资源。可以给共享资源加一把锁，这把锁只有一把钥匙。哪个线程获取了这把钥匙，才有权利访问该共享资源。

知识点7-线程同步-同步代码块

- 线程同步机制方式一：同步代码块
- 语法：

```
synchronized (共享对象名)
{
    被同步的代码段
}
```

- 同步代码块要注意的事项：

- 1、任意一个对象都可以作为锁对象，它锁定的是该对象；
- 2、线程中实现同步块一般是在run方法中添加；
- 2、在同步代码块中调用sleep方法并不会释放锁对象的；
- 3、只有真正存在线程安全问题的时候才使用同步代码块，否则会降低效率；
- 4、多线程操作的锁对象必须是唯一共享的，否则无效；

```
@Override
public void run() {
    // 永真循环
    while (true) {
        synchronized (obj) { ← static Object obj = new Object();
            // 余票大于0的时候，卖票
            if (num > 0) {
                System.out.println(this.getName() + "卖了一张票，还剩" + (num - 1) + "张票");
                num--;
            } else { // 等于或者小于0的时候，显示卖完了
                System.out.println("票卖完了");
                break;
            }

            try {
                Thread.sleep(5000); // 休眠5秒钟，在同步代码块中调用sleep方法并不会释放锁对象的
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

知识点7-线程同步-同步代码块



- 线程同步案例：
- 思考：一个银行账户5000块钱，两夫妻一个拿着折，一个拿着卡，开始取钱比赛，每次只能取一千块钱，要求不准出现线程安全问题

工作三连

查bug



改bug



写bug



知识点7-线程同步-同步方法



- 同步方法
 - 防止多个线程同时访问这个类中的synchronized 方法。它可以对类的所有对象实例起作用。

访问修饰符 `synchronized` 数据返回类型 方法名()`{...}`

- 同步方法注意：
 - 如果是一个非静态的同步方法的锁，对象是this对象；如果是静态的同步方法的锁，对象是当前函数所属的类的字节码文件（Class对象）。
 - 同步方法的锁对象是固定的，程序员无法指定。

知识点7-线程同步



- 线程同步案例：
- 需求：作一个讨债软件：
 - 1、有一个人欠雪姐50万，一直不给。
 - 2、雪姐公司有50个小弟，今天早晨开会，给大家安排的任务就是讨债。
 - 3、要求50个小弟全体出动去要钱，要求单独行动。每次一个小弟只能讨10万
 - 4、收到的回来报告，有提成。



知识点7-线程同步-优缺点



- 同步的前提
 - 同步需要两个或者两个以上的线程。
 - 多个线程使用的是同一个锁未满足这两个条件，不能称其为同步。
- 同步的弊端
 - 性能下降
 - 会带来死锁
- 同步的优势
 - 解决了线程安全问题、对公共资源的使用有序化。
- 同步注意
 - 不要盲目对不需要同步的代码使用Synchronized,以避免影响性能和效率。

知识点8-线程死锁

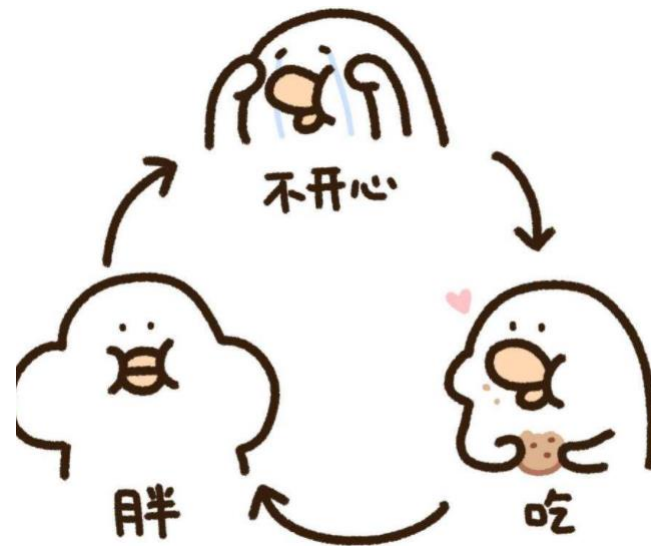


- 线程死锁

- 线程死锁指的两个线程互相持有对方依赖的共享资源，造成都无限阻塞。
- 导致死锁的根源在于不适当地运用“synchronized”关键词来管理线程对特定对象的访问。

- 解决死锁的方法

- 让线程持有独立的资源。
- 尽量不采用嵌套的synchronized语句。
- 死锁要通过通过优良的设计来尽量降低死锁的发生。



知识点8-线程死锁



- 死锁案例：电池和遥控器（张三拿电池，李四拿遥控器，谁也不肯放手）

```
class DeadLock extends Thread{
    public DeadLock(String name) {
        super(name);
    }
    @Override
    public void run() {
        if ("张三".equals(Thread.currentThread().getName())) {
            synchronized ("遥控器") {
                System.out.println("张三拿到了遥控器，准备拿电池");
                synchronized ("电池") {
                    System.out.println("张三拿到了遥控器也拿到了电池，吹着空调...");
                }
            }
        } else if ("李四".equals(Thread.currentThread().getName())) {
            synchronized ("电池") {
                System.out.println("李四拿到了遥控器，准备拿电池");
                synchronized ("遥控器") {
                    System.out.println("李四拿到了遥控器也拿到了电池，吹着空调...");
                }
            }
        }
    }
}
```

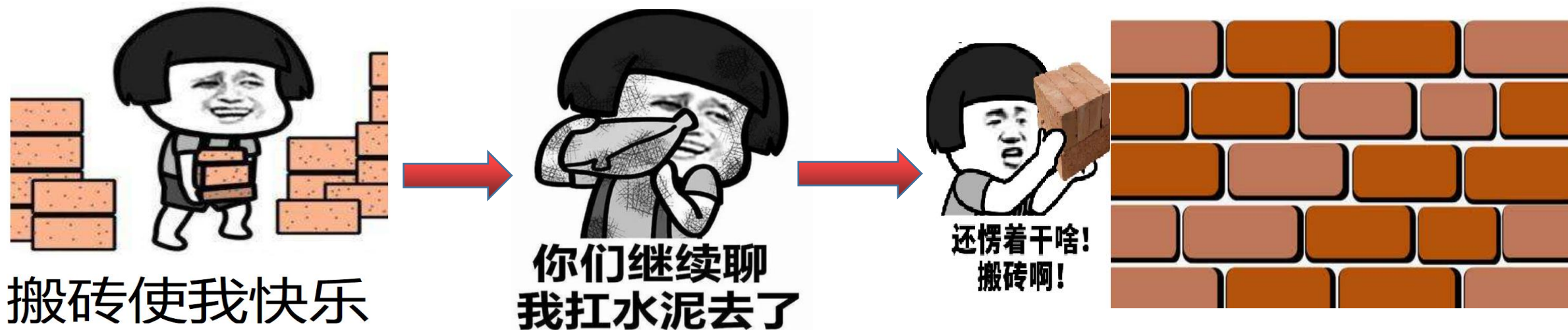
张三等着李四的电池，李四等着张三的遥控器，无限等待中

思考：这种情况一定会发生吗？不一定，当张三快速的执行完毕时

知识点9-线程通讯

- 线程通讯的概念

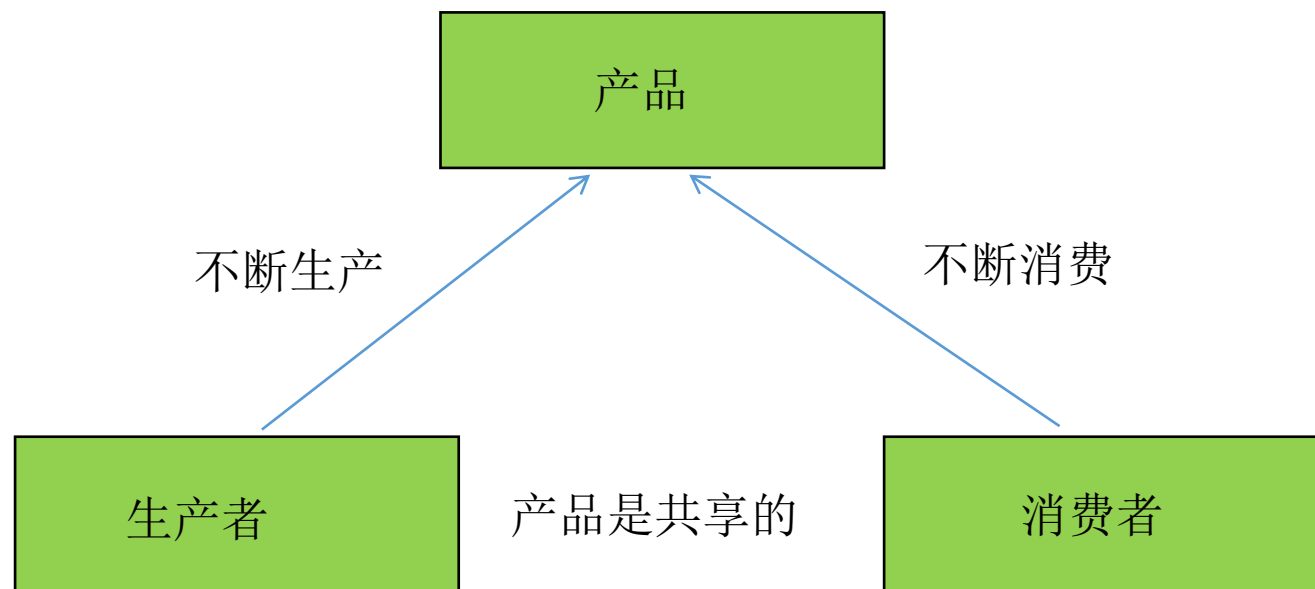
- 线程通讯指的是多个线程通过消息传递实现相互牵制，相互调度，即线程间的相互作用。
- 前面学的是多个线程执行同一动作（一个run方法）即同一个任务，现在线程还是多个，但运行的任务却是不同的，不过它们处理的资源是相同的。



- Java代码中基于对共享数据进行“wait()、notify()、notifyAll()”来实现多个线程的通讯。
- 经典例子：生产者和消费者的问题

知识点9-线程通讯-生产者与消费者问题

- 假设仓库中只能存放一件产品，生产者将生产出来的产品放入仓库，消费者将仓库中的产品取走消费。如果仓库中没有产品，则生产者可以将产品放入仓库，否则停止生产并等待，直到仓库中的产品被消费者取走为止。如果仓库中放有产品，则消费者可以将产品取走消费，否则停止消费并等待，直到仓库中再次放入产品为止。
- 对于生产者，在生产者没有生产之前，要通知消费者等待；在生产者生产之后，马上又通知消费者消费；对于消费者，在消费者消费之后，要通知生产者已经消费结束，需要继续生产新的产品以供消费。



知识点9-线程通讯

• wait() / notify() / notifyAll() 用法

wait()和notify()方法(包括上述的所有方法, 下同) 都是Object类的最终方法, 所以每个类默认都拥有该方法。

方 法 原 型	说 明
final void notify()	唤醒在此对象监视器上等待的单个线程。
final void notifyAll()	唤醒在此对象监视器上等待的所有线程。
final void wait() throws InterruptedException	导致当前的线程等待, 直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法。
final void wait(long timeout) throws InterruptedException	导致当前的线程等待, 直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法, 或者超过指定的时间量。
final void wait(long timeout, int nanos) throws InterruptedException	导致当前的线程等待, 直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法, 或者其他某个线程中断当前线程, 或者已超过某个实际时间量。

为确保wait()和notify()的功能, 即一定要和synchronized一起使用。

如synchronized的锁对象是obj的话, wait和notify正确的使用方法是obj.wait()和obj.notify()。

知识点9-线程通讯



- sleep()和wait()的区别



	sleep ()	wait ()
归属	Thread的静态方法 必须指定时间	Object的方法，可以指定时间 也可以不指定时间
作用	让本线程进入睡眠状态 到点就醒	让本线程进入“等待”状态 需要有人叫醒
是否释放同步锁	不会释放同步锁	会释放同步锁

知识点9-线程通讯-管道流通讯

- Java中的Pipe管道输入流与Pipe管道输出流实现了类似管道的功能，用于不同线程之间的相互通信：

```
class Sender extends Thread {  
    private Random rand = new Random();  
    private PipedWriter out = new PipedWriter();  
    public PipedWriter getPipedWriter() {  
        return out;  
    }  
    public void run() {  
        while (true) {  
            for (char c = 'A'; c <= 'z'; c++) {  
                try {  
                    out.write(c);  
                    sleep(rand.nextInt(500));  
                } catch (Exception e) {  
                    throw new RuntimeException(e);  
                }  
            }  
        }  
    }  
}
```

```
class Receiver extends Thread {  
    private PipedReader in;  
    public Receiver(Sender sender) throws IOException {  
        in = new PipedReader(sender.getPipedWriter());  
    }  
    public void run() {  
        try {  
            while (true) {  
                System.out.println("Read: " + (char) in.read());  
            }  
        } catch (IOException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

```
public class PipedIO {  
    public static void main(String[] args) throws Exception {  
        Sender sender = new Sender();  
        Receiver receiver = new Receiver(sender);  
        sender.start();  
        receiver.start();  
    }  
}
```

运行结果

Read: A
Read: B
Read: C
Read: D
Read: E

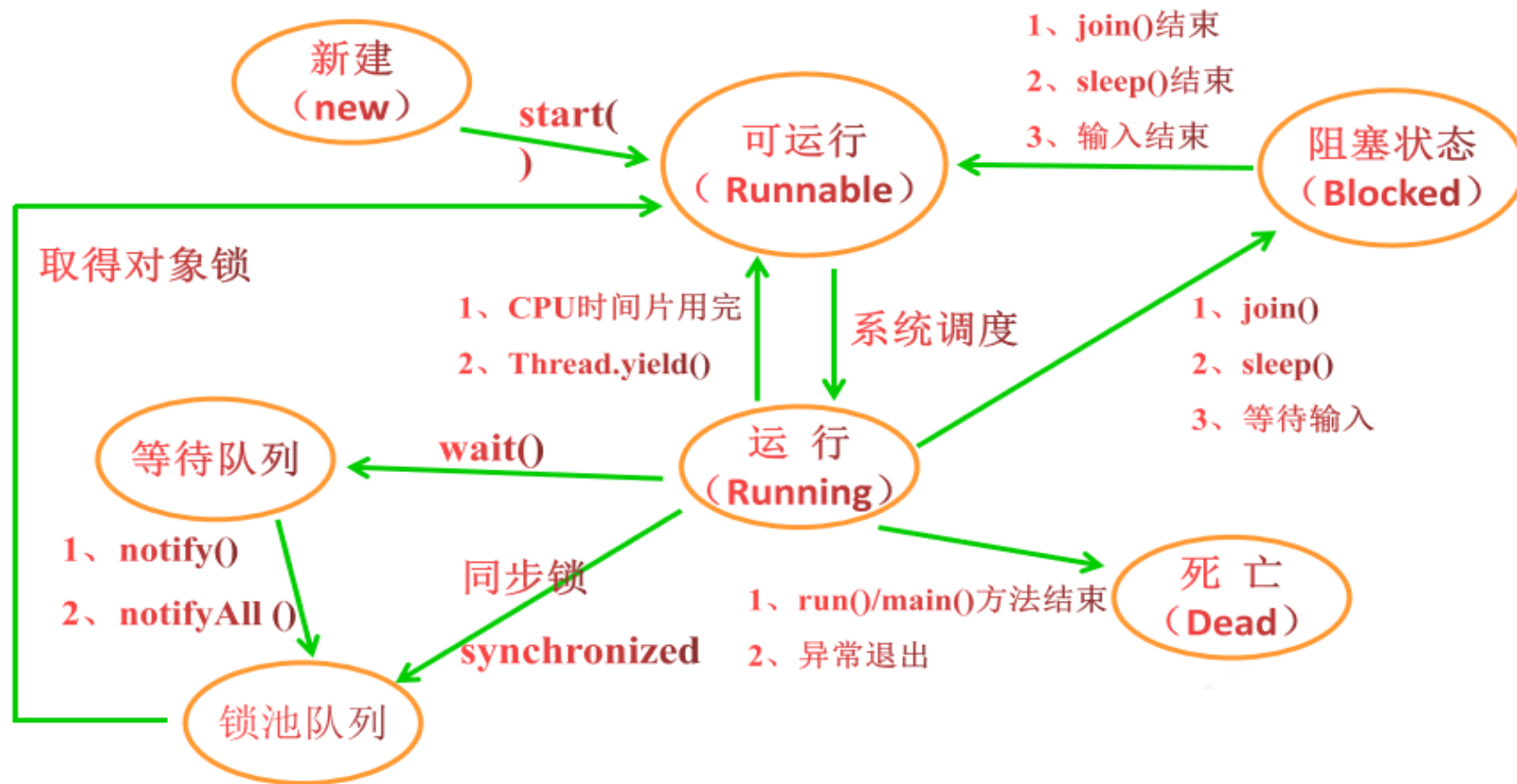
知识点9-线程通讯-管道流通讯

- Java在它的jdk文档中提到不要在一个线程中同时使用管道输入流和管道输出流，这可能会造成死锁

知识点10-线程的生命周期



- Java线程在它的生命周期中会处于不同的状态:



知识点10-线程的生命周期



- 新建状态 (New) : 使用new关键字创建线程对象, 仅仅被分配了内存
- 可运行状态 (Runnable) : 线程具备获得CPU时间片的能力。线程进入可运行状态的情况如下:
 - 线程start()方法被调用
 - 当前线程sleep()、其它线程join()结束、等待用户输入完毕;
 - 某个线程取得对象锁;
 - 当前线程时间片用完了, 调用当前线程的yield()方法

知识点10-线程的生命周期

- 运行状态 (Running) : 执行run方法, 此时线程获得CPU的时间片;
- 阻塞状态 (Blocked) : 线程由于某些事件放弃CPU使用权, 暂停运行。直到线程重新进入可运行状态, 才有机会转到运行状态。阻塞状态分为如下几种: :
 - 同步阻塞 – 线程获得同步锁, 若被别的线程占用, 线程放入锁池队列中
 - 等待阻塞 – 线程执行wait()方法, 线程放入等待队列中。某个线程取得对象锁;
 - 其它阻塞 – 线程执行sleep()或join()或发出I/O请求。
- 死亡状态 (Dead) : run、main() 方法执行结束, 或者因异常退出了run()方法, 线程进入死亡状态, 不可再次复生。

知识点11-线程池调度器-线程池介绍

- 线程池概念

我们可以把并发执行的任务传递给一个线程池，来替代为每个并发执行的任务都启动一个新的线程。只要池里有空闲的线程，任务就会分配给一个线程执行。在线程池的内部，任务被插入一个阻塞队列（**Blocking Queue**），线程池里的线程会去取这个队列里的任务。当一个新任务插入队列时，一个空闲线程就会成功的从队列中取出任务并且执行它。

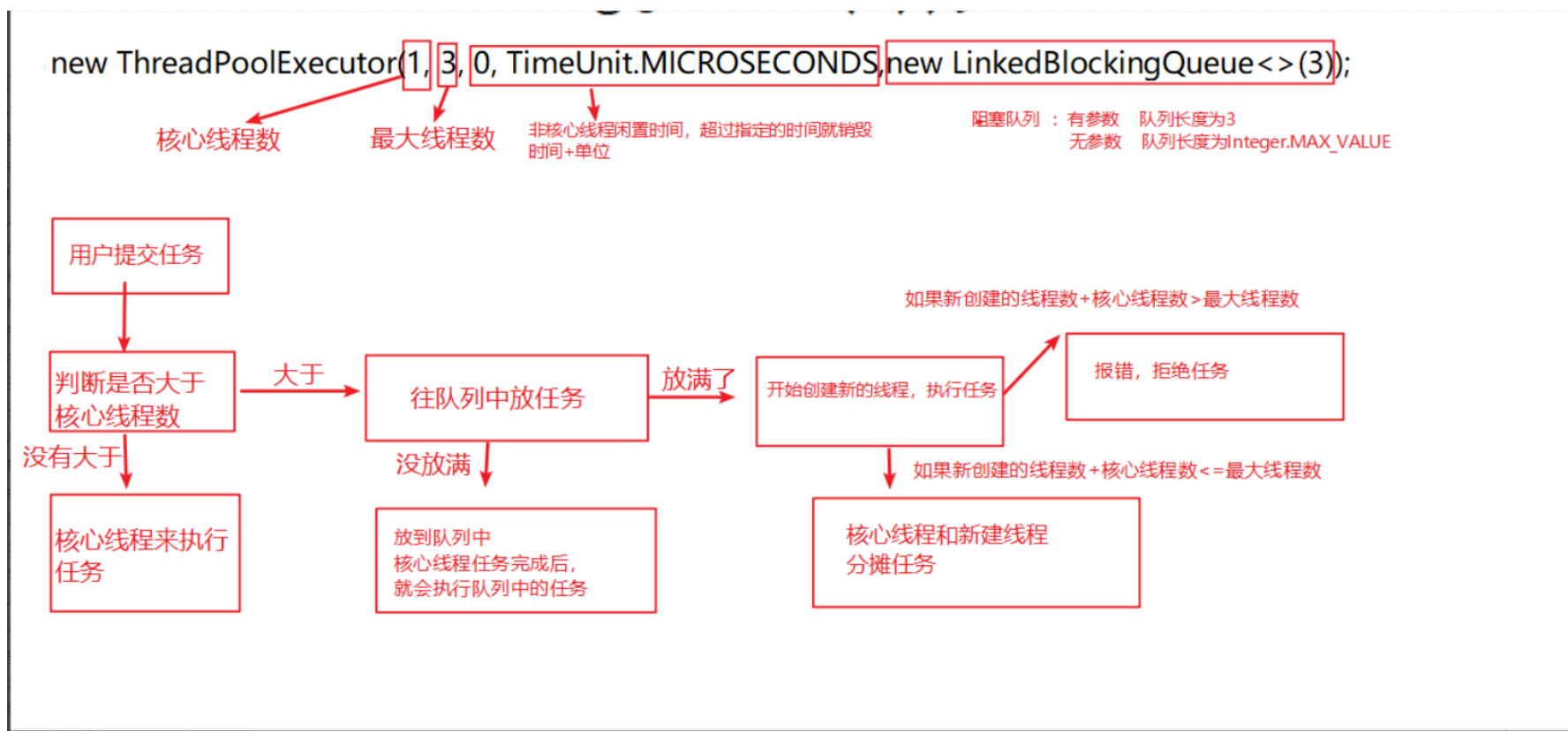
- 为什么要使用线程池？

当需要处理的任务较少时，我们可以自己创建线程去处理，但在高并发场景下，我们需要处理的任务数量很多，由于创建销毁线程开销很大，这样频繁创建线程就会大大降低系统的效率。此时，我们就可以使用线程池，线程池中的线程执行完一个任务后可以复用，并不被销毁。合理使用线程池有以下几点好处：

- 1、减少资源的开销。通过复用线程，降低创建销毁线程造成的消耗。
- 2、多个线程并发执行任务，提高系统的响应速度。
- 3、可以统一的分配，调优和监控线程，提高线程的可管理性。

知识点11-线程池调度器-线程池创建

- 通过ThreadPoolExecutor来创建一个线程池。
- java.util.concurrent.ThreadPoolExecutor类是线程池中最核心的一个类, 它有四个构造方法。



知识点11-线程池调度器-线程池创建



• 构造方法参数说明：

1. **corePoolSize**: 核心线程数，默认情况下核心线程会一直存活，即使处于闲置状态也不会受存**keepAliveTime**限制。除非将**allowCoreThreadTimeOut**设置为**true**。
2. **maximumPoolSize**: 线程池所能容纳的最大线程数。超过这个数的线程将被阻塞。当任务队列为没有设置大小的**LinkedBlockingDeque**时，这个值无效。
3. **keepAliveTime**: 非核心线程的闲置超时时间，超过这个时间就会被回收。
4. **unit**: 指定**keepAliveTime**的单位，如**TimeUnit.SECONDS**。当将**allowCoreThreadTimeOut**设置为**true**时对**corePoolSize**生效。
5. **workQueue**: 线程池中的任务队列(阻塞队列)。
常用的有三种队列，**SynchronousQueue**, **LinkedBlockingDeque**, **ArrayBlockingQueue**。

知识点11-线程池调度器-线程池创建



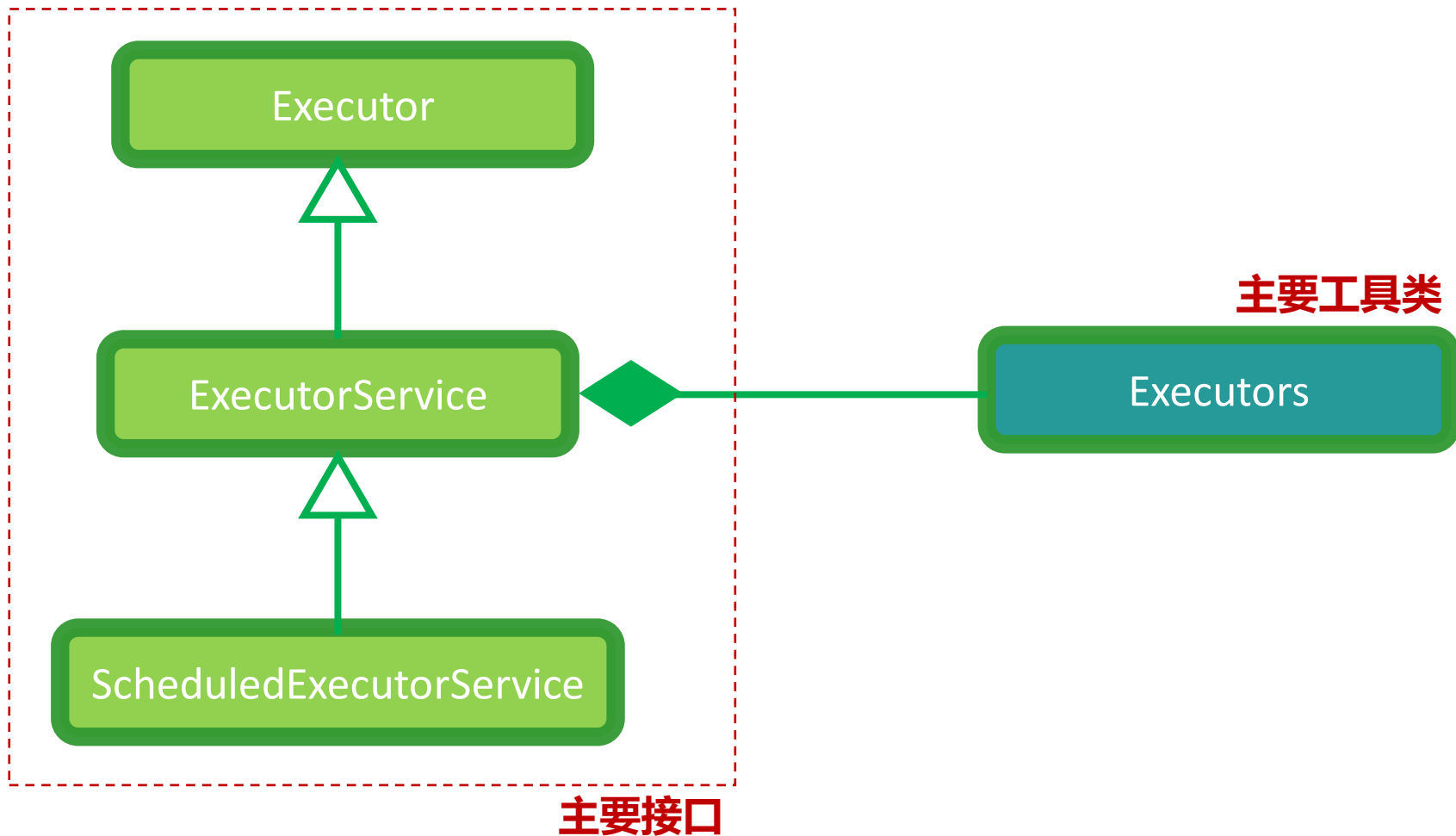
- 常用方法：

- 1.int getCorePoolSize():返回核心线程数。
2. int getPoolSize(): 返回池中当前的线程数。
- 3.BlockingQueue<Runnable> getQueue(): 返回此执行程序使用的任务队列。
- 4.void shutdown(): 关闭线程池
- 5.void execute(Runnable command): 执行任务

知识点11-线程池调度器



- Executor框架的重要核心接口和类：



知识点11-线程池调度器

- Executor是一个可以提交可执行任务的工具，这个接口解耦了任务提交和执行细节（线程使用、调度等），Executor主要用来替代显示的创建和运行线程
- ExecutorService提供了异步的管理一个或多个线程终止、执行过程(Future)的方法
- Executors类提供了一系列工厂方法用于创建任务执行器，返回的任务执行器都实现了ExecutorService接口（绝大部分执行器完成了池化操作）

知识点11-线程池调度器



- 内置的常见工厂方法及生成的任务调度器特征：

工厂方法	构造器说明
ExecutorService newFixedThreadPool (int nThreads)	创建固定数目线程的线程池
ExecutorService newCachedThreadPool ()	创建一个可缓存的线程池，调用execute 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程
ExecutorService newSingleThreadExecutor ()	创建一个单线程化的Executor，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行
ScheduledExecutorService newScheduledThreadPool (int corePoolSize)	创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代Timer类



- 有几种不同的方式将任务委托给一个 ExecutorService:
 - execute(Runnable)
 - submit(Runnable)
 - submit(Callable)
 - invokeAny(...)
 - invokeAll(...)

知识点11-线程池调度器



- `execute(Runnable)`

- 接收一个`java.lang.Runnable`对象作为参数，并且以异步的方式执行它，使用这种方式没有办法获取执行`Runnable`之后的结果，如果你希望获取运行之后的返回值，就必须使用 接收`Callable`参数的`execute()` 方法

- `submit(Runnable)`

- 同样接收一个`Runnable`的实现作为参数，但是会返回一个`Future` 对象。这个`Future`对象可以用于判断`Runnable`任务是否结束执行

- `submit(Callable)`

- 和方法`submit(Runnable)`比较类似，但是区别在于它们接收不同的参数类型。`Callable`的实例与`Runnable`的实例很类似，但是`Callable`的`call()`方法可以返回一个结果而方法`Runnable.run()`则不能返回结果
- `Callable`的返回值可以从方法`submit(Callable)`返回的`Future`对象中获取

知识点11-线程池调度器



- `inVokeAny()`

- 接收一个包含Callable对象的集合作为参数。调用该方法不会返回Future对象，而是返回集合中某个Callable对象的结果，而且无法保证调用之后返回的结果是集合中的哪个Callable结果，只知道它是这些Callable中的一个
- 如果一个任务运行完毕或者抛出异常，方法会取消其它的Callable的执行

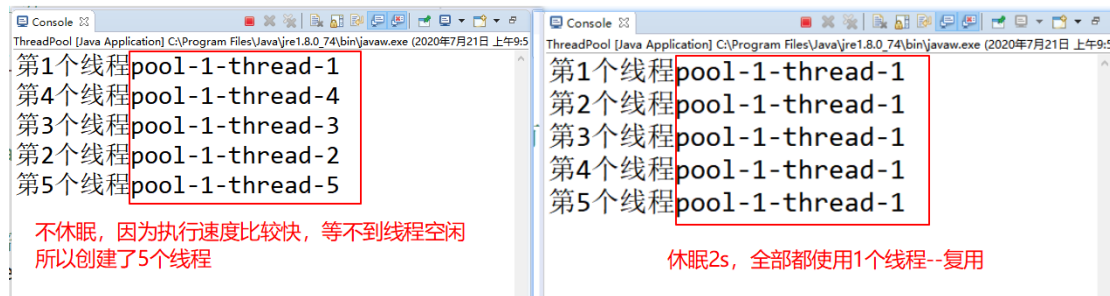
- `invokeAll()`

- 会调用存在于参数集合中的所有 Callable 对象，并且返回一个包含Future对象的集合，可以通过这个返回的集合来管理每个Callable的执行结果
- 需要注意的是，任务有可能因为异常而导致运行结束，所以它可能并不是真的成功运行了。但是我们没有办法通过 Future 对象来了解到这个差异

知识点11-线程池调度器-newCachedThreadPool

- newCachedThreadPool创建一个可缓存线程池，先查看池中有没有以前建立的线程，如果有，就直接使用。如果没有，就建一个新的线程加入池中，缓存型池子通常用于执行一些生存期很短的异步型任务。线程池为无限大，当执行当前任务时上一个任务已经完成，会复用执行上一个任务的线程，而不用每次新建线程。

```
13  /*
14  newCachedThreadPool创建一个可缓存线程池，先查看池中有没有以前建立的线程，
15  如果有，就直接使用。如果没有，就建一个新的线程加入池中
16  */
17  */
18  private static void test01() {
19      //创建一个可缓存的线程池
20      ExecutorService es1 = Executors.newCachedThreadPool();
21      //执行5次任务
22      for (int i = 1; i <= 5; i++) {
23          int index = i;
24          try {
25              Thread.sleep(2000);
26          } catch (InterruptedException e) {
27              e.printStackTrace();
28          }
29          //内部类---new Runnable(){run{}}相当于 自定义类实现Runnable接口重写run方法
30          es1.execute(new Runnable() {
31              @Override
32              public void run() {
33                  System.out.println("第"+index+"个线程"+Thread.currentThread().getName());
34              }
35          });
36      }
37  }
```



第1个线程 pool-1-thread-1
第4个线程 pool-1-thread-4
第3个线程 pool-1-thread-3
第2个线程 pool-1-thread-2
第5个线程 pool-1-thread-5

不休眠，因为执行速度比较快，等不到线程空闲
所以创建了5个线程

第1个线程 pool-1-thread-1
第2个线程 pool-1-thread-1
第3个线程 pool-1-thread-1
第4个线程 pool-1-thread-1
第5个线程 pool-1-thread-1

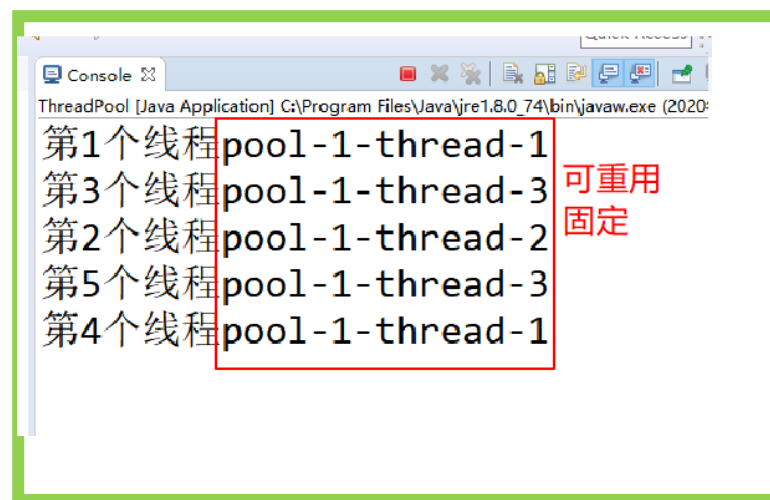
休眠2s，全部都使用1个线程--复用

知识点11-线程池调度器-newFixedThreadPool

- newFixedThreadPool创建一个可重用固定个数的线程池，以共享的无界队列方式来运行这些线程。因为线程池大小为3，每个任务输出打印结果后sleep 2秒，所以每两秒打印3个结果。定长线程池的大小最好根据系统资源进行设置。
- 创建方式： Executors.newFixedThreadPool(int nThreads); //nThreads为线程的数量

```
//newFixedThreadPool创建一个可重用固定个数的线程池
private static void test02() {
    //newFixedThreadPool创建一个可重用固定个数的线程池
    ExecutorService es = Executors.newFixedThreadPool(3);

    //结果解析：由于设置最大线程数为3，所以在输出三个数后等待1s才继续输出
    for (int i = 1; i <= 20; i++) {
        int index = i;
        es.execute(new Runnable() { //执行任务
            @Override
            public void run() {
                System.out.println("第"+index+"个线程"+Thread.currentThread().getName());
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
    }
}
```



ThreadPool [Java Application] C:\Program Files\Java\jre1.8.0_74\bin\javaw.exe (2020:

第1个线程	pool-1-thread-1
第3个线程	pool-1-thread-3
第2个线程	pool-1-thread-2
第5个线程	pool-1-thread-3
第4个线程	pool-1-thread-1

可重用
固定

知识点11-线程池调度器-newSingleThreadExecutor

- newSingleThreadExecutor创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务。
- 创建方式：Executors.newSingleThreadExecutor()；

```
//newSingleThreadExecutor创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务。  
private static void test03() {  
    //newSingleThreadExecutor创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务。  
    ExecutorService es = Executors.newSingleThreadExecutor();  
    for (int i = 1; i <= 5; i++) {  
        int index = i;  
        es.execute(new Runnable() { //执行任务  
            @Override  
            public void run() {  
                System.out.println("第"+index+"个线程"+Thread.currentThread().getName());  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        });  
    }  
}
```

知识点11-线程池调度器-newScheduledThreadPool

- newScheduledThreadPool创建一个定长线程池，支持定时及周期性任务执行

```
7 //newScheduledThreadPool创建一个定长线程池，支持定时及周期性任务执行
8 private static void test04() {
9     //创建一个定长线程池，支持定时及周期性任务执行
10    ScheduledExecutorService ses = Executors.newScheduledThreadPool(3);
11    //执行任务
12    ses.schedule(new Runnable() {
13        @Override
14        public void run() {
15            System.out.println("延迟3秒");
16        }
17    }, 3, TimeUnit.SECONDS);
18 }
```

延迟执行的时间+单位

执行的任务
延迟了你指定的时候后执行的操作

```
//周期性执行任务
private static void test05() {
    //创建一个定长线程池，支持定时及周期性任务执行
    ScheduledExecutorService ses = Executors.newScheduledThreadPool(3);
    ses.scheduleAtFixedRate(new Runnable() {
        @Override
        public void run() {
            System.out.println("延迟1秒后每3秒执行一次任务"+Thread.currentThread().getName());
        }
    }, 1, 3, TimeUnit.SECONDS);
}
```

要循环执行的任务

每隔多久执行一次任务---循环

延迟多久执行首次任务

知识点12-信号量

- 信号量为多线程协作提供了更为强大的控制方法。信号量是对锁的扩展。锁一次都只允许一个线程访问一个资源，而信号量却可以指定多个线程，同时访问某一个资源。
- eg:车位是共享资源，每辆车好比是一个线程，看门人是信号量。

知识点12-信号量-示例

```
public class SemapDemo implements Runnable {
    // 创建信号量对象，只能5个线程同时访问
    Semaphore semp = new Semaphore(5);

    @Override
    public void run() {
        try {
            semp.acquire(); // 获取许可
            Thread.sleep(5000);
            System.out.println(Thread.currentThread().getName() + ", 完成");
            // 访问完 释放 如果把下面语句注释掉，控制台只能打印5条数据，就一直阻塞
            semp.release();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        SemapDemo semapDemo = new SemapDemo();
        // 创建固定数目线程的线程池
        ExecutorService es = Executors.newFixedThreadPool(20);
        // 模拟20个客户端访问
        for (int i = 0; i < 20; i++) {
            es.execute(semapDemo); // 执行任务
        }
        // 退出线程池
        es.shutdown();
    }
}
```

pool-1-thread-2, 完成
pool-1-thread-6, 完成
pool-1-thread-4, 完成
pool-1-thread-5, 完成
pool-1-thread-1, 完成
pool-1-thread-3, 完成
pool-1-thread-13, 完成
pool-1-thread-9, 完成
pool-1-thread-14, 完成
pool-1-thread-7, 完成
pool-1-thread-8, 完成
pool-1-thread-11, 完成
pool-1-thread-10, 完成
pool-1-thread-18, 完成
pool-1-thread-17, 完成
pool-1-thread-12, 完成
pool-1-thread-15, 完成
pool-1-thread-16, 完成
pool-1-thread-19, 完成
pool-1-thread-20, 完成

休眠5s
信号量对象的参数是5
允许5个线程同时访问

5个5个一起执行

知识点13-Lock对象

- 虽然我们现在已经可以理解同步代码块和同步方法的锁对象问题，但是我们并不能直接看到在哪里加了锁，在哪里释放了锁，为了更清晰的表达如何加锁和释放锁，JDK5中提供了一个新的锁对象Lock（接口），提供了更为清晰的语义
- JDK5中有一个Lock的默认实现：ReentrantLock
 - 可重入的独占锁。该对象与synchronized关键字有着相同的表现和更清晰的语义，而且还具有一些扩展的功能。可重入锁被最近的一个成功lock的线程占有（unlock后释放）。该类有一个重要特性体现在构造器上，构造器接受一个可选参数，是否是公平锁，默认是非公平锁
 - 公平锁：
 - 先来一定先排队，一定先获取锁
 - 非公平锁：
 - 不保证上述条件。非公平锁的吞吐量更高

知识点13-Lock对象

- 利用Lock改造线程同步代码：

```
public class Resource3 {  
    private Lock lock = new ReentrantLock();  
    public void f() {  
        System.out.println(Thread.currentThread().getName()  
            + ":not synchronized in f()");  
        lock.lock();  
        try {  
            for (int i = 0; i < 5; i++) {  
                System.out.println(Thread.currentThread().getName()  
                    + ":synchronized in f()");  
                try {  
                    TimeUnit.SECONDS.sleep(3);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        } finally {  
            lock.unlock();  
        }  
    }  
    public void g() {  
        System.out.println(Thread.currentThread().getName()  
            + ":not synchronized in g()");  
        lock.lock();  
        try {
```

创建一个锁对象

锁定，进入同步块

释放锁，离开同步块

```
        for (int i = 0; i < 5; i++) {  
            System.out.println(Thread.currentThread().getName()  
                + ":synchronized in g()");  
            try {  
                TimeUnit.SECONDS.sleep(3);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    } finally {  
        lock.unlock();  
    }  
}  
  
public void h() {  
    System.out.println(Thread.currentThread().getName()  
        + ":not synchronized in h()");  
    lock.lock();  
    try {  
        for (int i = 0; i < 5; i++) {  
            System.out.println(Thread.currentThread().getName()  
                + ":synchronized in h()");  
            try {  
                TimeUnit.SECONDS.sleep(3);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    } finally {  
        lock.unlock();  
    }  
}  
  
public static void main(String[] args) {  
    final Resource3 rs = new Resource3();  
    new Thread(() -> rs.f()).start();  
    new Thread(() -> rs.g()).start();  
    rs.h();  
}
```



- ThreadLocal的作用：
 - ThreadLocal用来解决多线程程序的并发问题
 - ThreadLocal并不是一个Thread,而是Thread的局部变量,当使用ThreadLocal维护变量时,ThreadLocal为每个使用该变量的线程提供独立的变量副本,所以每个线程都可以独立地改变自己的副本,而不会影响其它线程所对应的副本
 - 从线程的角度看, 目标变量就象是线程的本地变量, 这也是类名中“Local”所要表达的意思

知识点14-ThreadLocal



- ThreadLocal类中的方法:
 - void set(T value)
 - 将此线程局部变量的当前线程副本中的值设置为指定值
 - void remove()
 - 移除此线程局部变量当前线程的值
 - protected T initialValue()
 - 返回此线程局部变量的当前线程的“初始值”
 - T get()
 - 返回此线程局部变量的当前线程副本中的值