

Chp13 多线程

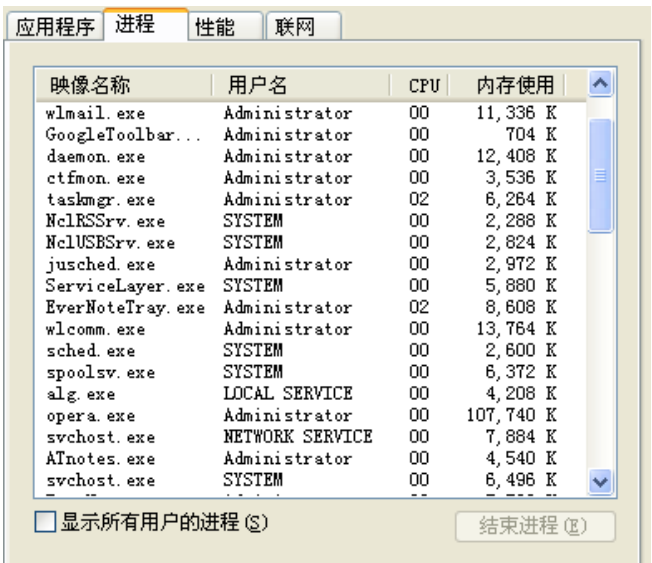
本章导读

并发编程是现代操作系统以及现代编程语言中很重要的一个组成部分。Java 语言中，对并发编程的支持很好，使用 Java 语言能够非常简单的写出多线程编程的代码。

1 多线程与并发的概念

1.1 并发与多进程

首先，我们来介绍一下并发的基本概念和原理。对于现代操作系统来说，大部分操作系统都是多任务操作系统。什么是“多任务”呢？指的是每个系统在同时能够运行多个进程。例如，在 windows 中，可能开着一个 Word 窗口写文档，开着一个 eclipse 写代码，开着一个 QQ 聊天，开着一个 IE 浏览网页，开着一个 Winamp 听歌……这就意味着，在一个操作系统中，同时有多个程序在内存中运行着。每一个运行着的程序，就是操作系统中运行着的一个任务，也就是我们所说的“进程”。例如，在 windows 中，可以通过“任务管理器”来查看系统中有多少进程正在运行。如下图：



The screenshot shows the Windows Task Manager window with the 'Processes' tab selected. It displays a list of running processes with columns for Name, Username, CPU usage, and Memory usage. The list includes various system and user processes like explorer.exe, chrome.exe, and system services.

映像名称	用户名	CPU	内存使用
explorer.exe	Administrator	00	11,336 K
chrome.exe	Administrator	00	704 K
daemon.exe	Administrator	00	12,408 K
ctfmon.exe	Administrator	00	3,536 K
taskmgr.exe	Administrator	02	6,264 K
NtLsmSrv.exe	SYSTEM	00	2,288 K
NtLsmSrv.exe	SYSTEM	00	2,824 K
jusched.exe	Administrator	00	2,972 K
ServiceLayer.exe	SYSTEM	00	5,880 K
EverNoteTray.exe	Administrator	02	8,608 K
wlcomm.exe	Administrator	00	13,764 K
sched.exe	SYSTEM	00	2,600 K
spoolsv.exe	SYSTEM	00	6,372 K
alg.exe	LOCAL SERVICE	00	4,208 K
opera.exe	Administrator	00	107,740 K
svchost.exe	NETWORK SERVICE	00	7,884 K
ATnotes.exe	Administrator	00	4,540 K
svchost.exe	SYSTEM	00	6,496 K

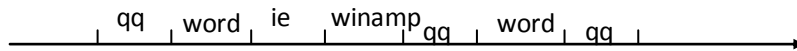
从上面的例子我们可以看出，在一个操作系统中可以同时运行多个进程，这就是进程的并发。

我们知道，一般来说，在个人电脑上，往往只有一个 CPU。同时，每个程序运行的时候，都需要在 CPU 上完成运算，也就是说，所有的程序运行时，都需要占用 CPU。那么，在系统中，是如何做到使用单 CPU 同时运行多个程序的呢？

接下来，我们来阐述一下单 CPU 执行多任务的原理。

假设我们同时开了多个程序：Word，IE，QQ，Winamp，对于操作系统来说，这意味着有四个进程要同时运行。为了解决这个问题，计算机规定了：让这四个进程轮流使用 CPU，每个进程用一小会儿。这个“一小会儿”，往往是若干个毫秒，这段时间被称为一个“CPU 时间片”。这样，每一秒钟可能有成百上千个 CPU 时间片，也就是说，在一秒钟之内，这四

个程序可能各自能够占用一小会儿 CPU，从而运行一下。从微观上来看，每一个特定的时刻，CPU 上只有一个程序在运行。示意图如下：



如上图所示，在某个特定的 CPU 时间片中，只运行一个程序。而操作系统控制 CPU，让多个程序不停的切换，从而保证多个程序能够轮流使用 CPU。

从本质上说，单 CPU 一次只能运行一个任务。但是，由于 CPU 时间片非常短，每次从一个程序切换到另一个程序的时候速度很快。在一秒钟内，可能有成百上千个 CPU 时间片，也就是说系统可能进行了成百上千次任务的切换。由于人的反应相对计算机来说，是比较慢的，因此对于人来说，感受就是在一秒钟内，这几个进程同时在运行。

这就是单 CPU 执行多任务的原理：利用很短的 CPU 时间片运行程序，在多个程序之间进行 CPU 时间片的进行快速切换。可以把这种运行方式总结成为一句话：宏观上并行，微观上串行。这是说，从宏观上来看，一个系统同时运行多个程序，这些程序是并行执行的；而从微观上说，每个时刻都只有一个程序在运行，这些程序是串行执行的。

1.2 线程的概念

上面所说的，是操作系统与多进程的概念。但是，由于 Java 代码是运行在 JVM 中的，对于某个操作系统来说，一个 JVM 就相当于一个进程。而 Java 代码不能够越过 JVM 直接与操作系统打交道，因此，Java 语言中没有多进程的概念。也就是说，我们无法通过 Java 代码写出一个多进程的程序来。

Java 中的并发，采用的是线程的概念。简单的来说，一个操作系统可以同时运行多个程序，也就是说，一个系统并发多个进程；而对于每个进程来说，可以同时运行多个线程，也就是：一个进程并发多个线程。

从上面的描述上我们可以看出，线程就是在一个进程中并发运行的一个程序流程。当若干的 CPU 时间片分配给 JVM 进程的时候，系统还可以把时间片进一步细分，分给 JVM 中的每一个线程来执行，从而达到“宏观上并行，微观上串行”的线程执行效果。

目前为止，我们见到的 Java 程序只有一个线程，也就是说，只有一个程序执行流程。这个流程从 main 方法的第一行开始执行，以 main 方法的退出作为结束。这个线程我们称之为“主线程”。

那么，一个线程运行，需要哪些条件呢？首先，必须要给线程赋予代码。通俗的说，就是必须要为线程写代码。这些代码是说明，启动线程之后，这个线程完成了什么功能，我们需要这个线程来干什么。

其次，为了能够运行，线程需要获得 CPU。只有获得了 CPU 之后，线程才能真正启动并且执行线程的代码。CPU 的调度工作是由操作系统来完成的。

第三，运行线程时，线程必须要获得数据。也就是说，在进行运算的时候，线程需要从内存空间中获取数据。关于线程的数据，有一个结论，叫做“堆空间共享，栈空间独立”。所谓的“堆空间”，保存的是我们利用 new 关键字创建出来的对象；而所谓的栈空间，保存的是程序运行时的局部变量。“堆空间共享，栈空间独立”的意思是：多线程之间共享同一个堆空间，而每个线程又拥有各自独立的栈空间。因此，运行程序时，多个不同线程能够访问相同的对象，但是多个不同线程彼此之间的局部变量是独立的，不可能出现多个线程访问同一个局部变量的情况。

CPU、代码、数据，是线程运行时所需要的三大条件。在这三大条件中，CPU 是操作系

统分配的，Java 程序员无法控制；数据这部分，需要把握住“堆空间共享，栈空间独立”的概念（关于这个概念，我们在后面还会继续提到）；而下面一个章节，将为大家介绍，如何为线程赋予代码。

2 Thread 类与 Runnable 接口

在 Java 中，要为线程赋予代码，有两种方式：一种是继承 Thread 类，一种是实现 Runnable 接口。下面我们分别来看一下如何来使用这两种方式为线程赋予代码。

2.1 Thread 类

第一种方式是创建一个类，让这个类继承 java.lang.Thread 类，然后覆盖 Thread 类中的 run() 方法，在 run() 中提供线程的代码。注意，Thread 类中的 run() 签名如下：

```
public void run()
```

这就意味着我们创建的 Thread 中的 run() 签名也应当写成这样。

例如，假设我们想要创建两个线程：一个线程输出 1000 遍“\$\$\$”；另一个线程输出 1000 遍“###”。在为线程赋予代码的时候，我们只需要创建两个新的类继承 Thread，并且覆盖 Thread 中的 run() 即可。示例代码如下：

```
class MyThread1 extends Thread{
    public void run(){
        for(int i = 1; i<=1000; i++){
            System.out.println(i + " $$$");
        }
    }
}
```

```
class MyThread2 extends Thread{
    public void run(){
        for(int i = 1; i<=1000; i++){
            System.out.println(i + " ###");
        }
    }
}
```

上面的代码中，我们定义了两个线程类，并为这两个线程类赋予了代码。

有了这两个线程类之后，我们就可以利用这两个类来创建和处理线程了。那应该如何利用自定义的线程类来创建新线程呢？

首先，应当利用自定义的线程类创建线程对象，之后，调用线程的 start()，就能够启动新线程。注意，在我们自定义的线程类（MyThread1 和 MyThread2）中，我们并没有自己添加 start() 方法，这个方法是从 Thread 类中继承来的。相关代码如下：

```
public class TestThread{
    public static void main(String args[]){
        Thread t1 = new MyThread1();
        Thread t2 = new MyThread2();

        t1.start();
    }
}
```

```
        t2.start();  
    }  
}
```

我们可以执行一下这个 `TestThread` 程序，在我当前的 Windows XP 机器上，执行中的部分结果如下：

```
1 $$$  
1 ###  
2 ###  
3 ###  
4 ###  
2 $$$  
5 ###  
3 $$$  
6 ###  
7 ###  
4 $$$  
8 ###  
9 ###  
10 ###  
11 ###  
12 ###  
13 ###  
14 ###  
15 ###  
16 ###  
17 ###  
18 ###  
19 ###  
20 ###  
21 ###  
5 $$$  
6 $$$  
7 $$$  
8 $$$  
9 $$$  
10 $$$  
11 $$$  
12 $$$  
13 $$$  
.....  
###和$$$一直交替输出，一直到最后：  
949 $$$  
.....  
973 $$$
```

```
988 ###
989 ###
.....
998 ###
999 ###
1000 ###
974 $$$
975 $$$
976 $$$
.....
997 $$$
998 $$$
999 $$$
1000 $$$
```

可以看出，两个线程交替打印出\$\$\$和###字符串，说明这两个线程交替的占用了 CPU。上面的代码中，有好几处需要注意的地方。首先，上面的程序中有这样两行代码：

```
Thread t1 = new MyThread1();
Thread t2 = new MyThread2();
```

这两行代码创建了 t1 和 t2 两个线程对象，但是，并没有在系统中创建新的线程。我们利用 new 关键字创建出来的，是 JVM 中的两个对象。这两个对象并不等于系统中的线程，但是通过操作这两个对象，能够来操作系统中的线程。

怎么来理解呢？就好比我们去银行办理业务，归根结底是去银行修改我们在银行的账户信息，比如存款就是增加账户余额，而取款就是减少账户余额，等等。但是我们去银行，是不能直接操作自己的账户的，而必须通过银行的职员来操作。换句话说，我们以银行职员为“代理”来操作自己的账户。

与上面的例子相似，我们创建的线程对象，就好像是系统中的线程的“代理”。也就是说，我们能够通过线程对象来对系统中的线程进行操作，可以请求系统创建线程、销毁线程等等。但是，线程对象本身不是线程：创建线程对象时，系统中没有创建新的线程；而销毁线程时，线程对象有可能还存在，要一直等到垃圾回收时，线程对象才会被销毁。

创建完线程对象之后，只有调用 start()，系统中才真正启动了一个新线程。就像我们之前所说的那样，start()是 Thread 类中的方法。特别要注意的是，当我们创建线程时，覆盖的是 run()，而不要去覆盖 start()！

调用完 start()之后，线程就开始真正启动了。可以猜测，在 start()内部，会通过 JVM 跟底层的操作系统进行交互，请求系统创建一个新线程。创建完新线程之后，这个线程执行的就是 run()中的代码。换句话说，我们可以当做在 start()内部，调用了我们覆盖以后的 run()。

另外，当我们对两个对象调用了 start 方法之后，在我们的 JVM 中总共有几个线程在运行呢？要注意，现在总共有三个线程在运行：

```
t1 线程：执行 MyThread1 中的 run 方法
t2 线程：执行 MyThread2 中的 run 方法
主线程：执行 main 方法。
```

注意，除了我们创建的 t1 和 t2 两个线程之外，还有一个主线程。主线程是所有 Java 程序运行时首先启动的第一个线程。而 t1 和 t2 这两个线程，都是在主方法内部调用 start 以后

才启动起来的，因此也就是说，这两个线程是从主线程中启动的。

那能不能直接调用 `run()` 方法呢？从语法上来说，完全可以对线程对象调用 `run()` 方法。但是，那样的话，并没有创建新线程。例如，如果在主方法中这样来调用：

```
t1.run();
t2.run();
```

这样就意味着调用了这两个对象的 `run` 方法，执行的时候，会先执行 `t1` 对象的 `run` 方法，当 `t1` 的 `run` 方法返回之后，再执行 `t2` 线程的 `run` 方法。也就是说，在主线程中运行了这两个 `run` 方法，程序自始至终只有一个线程在执行。

2.2 Runnable 接口

除了继承 `Thread` 类之外，我们还可以采用另外一种方式创建线程，就是实现 `Runnable` 接口。`Runnable` 接口在 `java.lang` 包中定义，在这个接口中，只包含一个方法：`run()` 方法。该方法签名如下：

```
void run()
```

由于这个方法定义在接口中，因此默认就是 `public` 的。在实现 `Runnable` 接口的时候，只需要实现这个 `run` 方法就可以了。

下面我们修改刚刚的 `MyThread2` 这个类，把这个类改名为 `MyRunnable2`，并且由继承 `Thread` 类改为实现 `Runnable` 接口。相应代码如下：

```
class MyRunnable2 implements Runnable{
    public void run(){
        for(int i = 1; i<=1000; i++){
            System.out.println(i + " ###");
        }
    }
}
```

在上面的代码中，对 `run()` 方法的实现没有任何修改，而只是改动了第一行：把继承 `Thread` 类改成了实现 `Runnable` 接口。

修改完了线程的实现之后，还必须要修改一下创建线程的代码。由于 `MyRunnable2` 类不是 `Thread` 类的子类，因此创建的 `MyRunnable2` 类型的对象不能直接赋值给 `Thread` 类型的引用。换句话说，我们利用 `MyRunnable2` 创建出来的不是一个线程对象，而是一个所谓的“目标”对象。代码如下：

```
Runnable target = new MyRunnable2();
```

创建了目标对象之后，可以利用目标对象再来创建线程对象，代码如下：

```
Thread t2 = new Thread(target);
```

通过上面的两行代码，我们就可以利用 `Runnable` 接口的实现类来创建线程对象。创建完线程对象之后，需要启动线程时，同样要调用线程对象的 `start()` 方法。完整的代码如下：

```
class MyThread1 extends Thread{
    public void run(){
        for(int i = 1; i<=1000; i++){
            System.out.println(i + " $$$");
        }
    }
}
```

```

}

class MyRunnable2 implements Runnable{
    public void run(){
        for(int i = 1; i<=1000; i++){
            System.out.println(i + " ###");
        }
    }
}

public class TestThread{
    public static void main(String args[]){
        Thread t1 = new MyThread1();
        Runnable target = new MyRunnable2();
        Thread t2 = new Thread(target);

        t1.start();
        t2.start();
    }
}

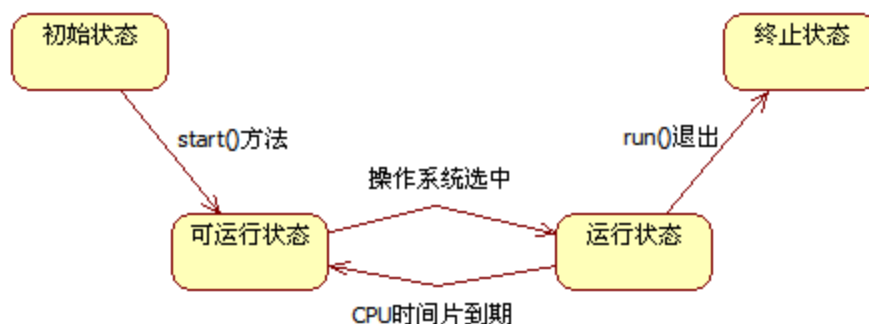
```

3 线程的状态

在上一节的内容中，我们为大家介绍了如何为线程写代码。在这一小节中，我们将更加详细的为大家介绍线程的相关知识，首先来介绍线程在整个程序运行中的状态转换。

3.1 线程状态图

在上一节的例子中，我们创建的程序总共有这样四个状态：初始状态、可运行状态、运行状态、终止状态。这四个状态的转换如下图：



下面对这四种状态进行详细的介绍。

初始状态：当我们创建了一个线程对象，而没有调用这个线程对象的 `start()` 方法时，此时线程处于初始状态。在上一小节中，我们介绍过，创建了一个线程对象，不等于在系统中创建了一个新线程。当我们创建一个线程对象时，此时线程处于初始状态。也就是说，线程

对象自身进行了一些初始化的操作，而并没有向操作系统申请创建系统中的线程。

另外，要注意的是，当一个线程调用 `start()` 方法之后，由初始状态进入了可运行状态。这个状态的转换过程是一个单向的过程，只能有初始状态转换为可运行状态，而不能由可运行状态转换为初始状态。也就是说，一旦调用完了 `start` 方法之后，线程就进入了可运行状态而不会回到初始状态，因此在线程对象的整个生命周期中，只能调用一次 `start()` 方法。如果对同一个线程对象多次调用 `start()` 方法，会产生一个 `IllegalStateException` 异常。

可运行状态：处于可运行状态下的线程，具有这样的特点：线程已经为运行做好了完全的准备，只等着获得 CPU 来运行。也就是说，线程是万事俱备，只欠 CPU。

运行状态：处于这种状态的线程获得了 CPU 时间片，正在执行代码。由于系统中只有一个 CPU，因此，同时只能有一个线程处于运行状态。

当 CPU 空闲的时候，操作系统会检查是否有可运行状态的线程。如果有一个或多个线程处于可运行状态，系统会根据一定的规则挑选一个线程，为这个线程分配一个 CPU 时间片，从而使得这个线程进入了运行状态。

当然，一个线程不能永远的占用 CPU，不可能永远的处于运行状态。系统为了实现多任务同时进行，会为线程分配一个 CPU 时间片。当 CPU 时间片到期而线程没有执行完毕时，操作系统会把处于运行状态的线程转换成可运行状态，然后再重新从可运行状态中选取一个线程，为其分配 CPU 时间片。这就是运行状态和可运行状态之间的转换。

终止状态：当一个线程执行完了 `run()` 方法中的代码，该线程就会进入终止状态。要注意的是，这个状态转换也是一个单向箭头。也就是说，一旦一个线程从运行状态进入了终止状态，那这个线程就进入了生命的尾声，我们无法通过任何手段重新启动这个线程。

特别要提醒的是，在上一小节中，除了我们创建的两个线程之外，系统中还存在第三个线程：主线程。主线程的启动没有必要经历初始状态，当我们启动 JVM 执行程序时，JVM 会执行某个类的主方法，此时主方法就在主线程中执行。此后，当主方法结束之后，主线程就进入了终止状态。需要注意的是，主线程进入终止状态，并不意味着整个程序就结束了。例如上一小节的例子：

```
public class TestThread{
    public static void main(String args[]){
        Thread t1 = new MyThread1();
        Runnable target = new MyRunnable2();
        Thread t2 = new Thread(target);

        t1.start();
        t2.start();
    }
}
```

主方法调用完了两个线程的 `start()` 方法之后，就进入了终止状态。但是由于现在的程序中，除了主线程之外，还有两个线程 `t1` 和 `t2`，这两个线程还在没有终止，因此程序不会终止执行。

我们之前曾经说过，程序启动之后执行主方法，主方法退出时整个程序退出。应当说，这样的说法是不准确的，必须等程序中所有线程都进入终止状态之后，整个程序才会终止。只不过之前我们写的所有程序都只有主线程一个线程而已，当这个线程进入终止状态之后，

由于没有其他的线程存在，程序就终止了。而现在我们既然学习了多线程，就应当更新我们的概念。请记住：主方法退出时，程序未必退出；只有整个程序中所有线程都进入了终止状态，整个程序才会退出。

此外，还有一个小问题：为什么对一个线程调用 `start()` 方法之后，这个线程不能马上进入运行状态，而一定要首先进入可运行状态呢？

要回答这个问题，请思考：对于上面的例子来说，当程序运行到调用 `t1.start()` 方法启动 `t1` 线程时，有没有线程处于运行状态？

事实上，由于 `t1.start()` 这行代码处于主方法中，因此当这行代码被执行时，意味着主线程正处于运行状态！换句话说，由于运行状态有一个线程正在运行，因此被启动的新线程只能先处于可运行状态。

启动线程时，必须有别的线程调用 `start` 方法（例如，主线程中调用 `t1.start()` 方法），而调用 `start()` 方法又意味着，运行状态有别的线程正在运行（例如，调用 `t1.start()` 方法意味着主线程正在运行），因此新启动的线程只能处于可运行状态。

3.2 `sleep()`与阻塞

除了上面所说的四种状态之外，还有一个很重要的状态：阻塞状态。

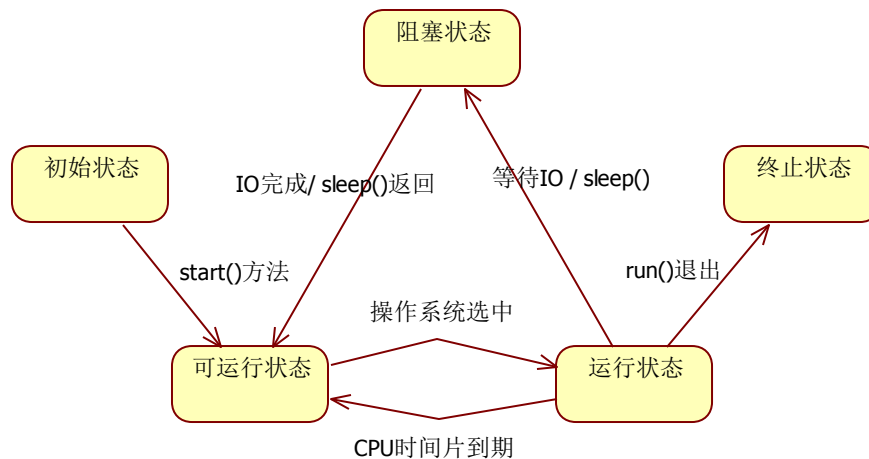
什么是阻塞状态呢？我们知道，如果一个线程要进入运行状态，则必须要获得 CPU 时间片。但是，在某些情况下，线程运行不仅需要 CPU 进行运算，还需要一些别的条件，例如等待用户输入、等待网络传输完成，等等。如果线程正在等待其他的条件完成，在这种情况下，即使线程获得了 CPU 时间片，也无法真正运行。因此，在这种情况下，为了能够不让这些线程白白占用 CPU 的时间，会让这些线程会进入阻塞状态。最典型的例子就是等待 I/O，也就是说，如果线程需要与 JVM 外部进行数据交互的话（例如等待用户输入、读写文件、网络传输等），这个时候，当数据传输没有完成时，线程即使获得 CPU 也无法运行，因此就会进入阻塞状态。

可以这么来理解：我们可以把 CPU 当做是银行的柜员，而去银行的准备办理业务的顾客就好比是线程。顾客希望能够让银行的柜员为自己办事，就好像线程希望获得 CPU 来运行线程代码一样。但有时会出现这种情况：比如有个顾客想去银行汇款，排队等到了银行的职员为自己服务，却事先没有填好汇款单。这个时候，这个银行职员即使想为你服务，也无法做到；这就相当于线程没有完成 I/O，此时线程即使获得 CPU 也无法运行。一般这个时候，银行的工作人员也不会傻等，而是会非常礼貌的给顾客一张单子，让他去旁边找个地方填写汇款单，而职员自己则开始接待下一位顾客；这也就相当于让一个线程进入阻塞状态。

当所等待的条件完成之后，处于阻塞状态的线程就会进入可运行状态，等待着操作系统为自己分配时间片。还是用我们银行职员的比喻：当顾客填写汇款单的时候，就好比是线程进入了阻塞状态；而当顾客把汇款单填完了之后，这就好比是线程的 I/O 完成，所等待的条件已经满足了。这样，这个顾客就做好了办理业务所需要的准备，就等着能够有一个银行职员来为自己服务了。

那为什么从阻塞状态出来之后，线程不能马上进入运行状态呢？因为当线程所等待的条件完成的时候，可能有一个线程正处于运行状态。由于处于运行状态的线程只能有一个，因此当线程从阻塞状态出来之后，只能在可运行状态中暂时等待。这就好比，当顾客填完汇款单之后，银行的职员可能正在为其他顾客服务。这个时候，你不能把其他用户赶走，强行让银行职员为你服务，而只能重新在后面排队，等着有银行职员为你服务。

加上阻塞状态之后，线程的转换图如下：



如上图所示，除了等待 I/O 会进入阻塞状态之外，还可以调用 `Thread` 类的 `sleep` 方法进入阻塞状态。顾名思义，`sleep` 方法就是让线程进入“睡眠”状态。你可以想象，如果一个顾客在银行办理业务的时候睡着了，那样的话，银行职员会让这个顾客在一旁先待着，等什么时候顾客睡醒了，再什么时候为这个顾客服务。

下面我们来看一下怎么调用 `sleep` 方法。首先，`sleep` 方法的方法签名为：

```
public static void sleep(long millis) throws InterruptedException
```

这个方法是一个 `static` 方法，也就意味着可以通过类名来直接调用这个方法。`sleep` 方法的参数是一个 `long` 类型，表示要让这个线程“睡”多少个毫秒（注意，1 秒=1000 毫秒）。另外，这个方法抛出一个 `InterruptedException` 异常，这个异常是一个已检查异常，根据异常处理的规则，这个异常必须要处理。

我们修改上一节的代码，让两个线程 `MyThread1` 和 `MyRunnable2` 每次输出之后都“睡”200 毫秒。首先修改 `MyThread1` 类：

```
class MyThread1 extends Thread{
    public void run(){
        for(int i = 1; i<=1000; i++){
            System.out.println(i + " $$$");
            try{
                Thread.sleep(200);
            }catch (InterruptedException e) {}
        }
    }
}
```

在 `for` 循环中增加了 `Thread.sleep` 方法。要注意的是，由于 `sleep` 方法抛出一个已检查异常，所以必须要处理。由于 `Thread` 类中的 `run` 方法没有抛出任何异常，根据方法覆盖的要求，`MyThread1` 类中的 `run` 方法也不能抛出任何异常。因此，对于 `sleep` 方法的抛出的异常，程序员只能用 `try-catch` 的方法处理。

下面是修改后的 `MyRunnable2` 的代码：

```
class MyRunnable2 implements Runnable{
    public void run(){
        for(int i = 1; i<=1000; i++){
```

```

        System.out.println(i + " ###");
        try{
            Thread.sleep(200);
        }catch (InterruptedException e) {}
    }
}

```

然后编译运行主程序，输出结果如下：

```

1 $$$
1 ###
2 $$$
2 ###
.....
998 $$$
998 ###
999 $$$
999 ###
1000 $$$
1000 ###

```

从输出结果可以看出，这两个线程基本上是交替进行输出。我们可以结合线程状态图来分析一下运行的流程。

首先，创建了 **t1** 对象和 **t2** 对象之后，这两个线程都进入了初始状态。之后，随着主方法中调用了 **t1.start()** 以及 **t2.start()**，这两个方法就进入了可运行状态。接着主线程进入终止状态，操作系统会从 **t1** 和 **t2** 中挑选一个线程进入运行状态。

假设首先选中的是 **t1** 线程，此时这个线程会输出“1 \$\$\$”，执行完这个输出语句之后，执行 **sleep** 语句，使得 **t1** 线程就进入了阻塞状态。此时，没有线程处于运行状态，于是操作系统会从可运行状态中挑选一个线程进入运行状态。由于可运行状态中只有 **t2** 一个线程，因此 **t2** 线程得到运行。

t2 线程运行时，会首先输出“2 ###”，输出之后，会执行 **sleep** 语句，使得 **t2** 线程也进入了阻塞状态。此时，**t1** 和 **t2** 线程都处于阻塞状态，而没有线程处于可运行状态以及运行状态，因此这时没有代码执行。

当 200 毫秒过去之后，**t1** 线程的 **sleep** 方法返回，此时 **t1** 线程由阻塞状态进入可运行状态。由于之前可运行状态和运行状态都没有线程，因此，这时操作系统会选中唯一一个处于可运行状态的 **t1** 线程进入运行状态。

再之后，**t2** 线程的 **sleep** 方法返回，**t2** 也进入了可运行状态。于是……
.....

进过多次状态的反复和转换，**t1** 和 **t2** 两个线程都输出完毕之后，程序中所有线程都进入了终止状态，整个程序结束。

当然，如果多次运行的话，可以发现，**t1** 和 **t2** 两个线程的交替输出不是绝对的，有可能出现下面的情况：

```

5 ###
5 $$$
6 $$$

```

```
6 ###
7 ###
7 $$$
8 $$$
8 ###
9 $$$
9 ###
```

也就是说，利用 `sleep` 方法对线程的控制是非常不精确的。试图用 `sleep` 方法严格的要求线程间进行交替输出，是非常的不可靠的。

3.3 join()

除了使用 `sleep()` 和等待 IO 之外，还有一个方法会导致线程阻塞，这就是线程的 `join()` 方法。我们首先来看下面一段代码：

```
class MyThread1 extends Thread{
    public void run(){
        for(int i = 0; i<100; i++){
            System.out.println(i + " $$$");
        }
    }
}

class MyThread2 extends Thread{
    public void run(){
        for(int i = 0; i<100; i++){
            System.out.println(i + " ###");
        }
    }
}

public class TestJoin{
    public static void main(String args[]){
        MyThread1 t1 = new MyThread1();
        MyThread2 t2 = new MyThread2();
        t1.start();
        t2.start();
    }
}
```

这段代码创建了两个线程对象，并且启动了两个新线程。这两个线程会交替输出\$\$\$和###。

接下来，我们修改一下 `MyThread2` 类，并修改主方法，修改后的代码如下：

```
class MyThread2 extends Thread{
    Thread t;
```

```

        public void run(){
            try{
                t.join();
            }catch(Exception e){}
            for(int i = 0; i<100; i++){
                System.out.println(i + " ###");
            }
        }
    }

    public class TestJoin{
        public static void main(String args[]){
            MyThread1 t1 = new MyThread1();
            MyThread2 t2 = new MyThread2();
            t2.t = t1;
            t1.start();
            t2.start();
        }
    }

```

上面这个程序中，**MyThread2** 对象增加了一个属性 **t**。在主方法中，把 **t1** 对象赋值给 **t**，也就是让 **t** 属性和 **t1** 引用指向同一个对象。

最重要的一点是，在 **MyThread2** 类的 **run()**方法中，调用了 **t** 属性的 **join()**方法。这个方法能够让 **MyThread2** 线程由运行状态进入阻塞状态，直到 **t** 线程结束。下面结合程序的状态转换图，来说明一下执行的过程。

首先 **main** 方法中，创建了两个线程对象，并且调用了 **t1** 和 **t2** 线程的 **start()**方法，这样，这两个线程就进入了可运行状态。假设操作系统首先挑选了 **t1** 线程进入了可运行状态，于是输出若干个“\$\$\$”。经过一段时间之后，由于 **CPU** 时间片到期，**t1** 线程进入了可运行状态。假设经过了一段时间之后，操作系统选择了 **t2** 线程进入了可运行状态。进入了 **t2** 线程的 **run()**方法之后，调用了 **t** 属性的 **join()**方法。由于 **t** 属性与 **t1** 指向同一个对象，因此这也就意味着在 **t2** 线程中，调用了 **t1** 线程的 **join()**方法。调用之后，**t2** 线程会进入阻塞状态。此时，运行状态没有线程在执行，因此系统会从可运行状态中选择一个线程执行。由于可运行状态此时只有一个 **t1** 线程，因此这个线程会一直占用 **CPU**，直到线程代码执行结束。当 **t1** 线程结束之后，**t2** 才会由阻塞状态进入可运行状态，此时才能够执行 **t2** 的代码。因此，从输出结果上来看，会先执行 **t1** 线程的所有代码，然后再执行 **t2** 线程的所有代码。部分输出结果如下：

```

0 $$$
1 $$$
2 $$$
.....
97 $$$
98 $$$
99 $$$
0 ###
1 ###

```

```
2 ###
3 ###
.....
97 ###
98 ###
99 ###
```

要注意的是，我们在 t2 线程中调用 t1 线程的 join()方法，结果是 t2 阻塞，直到 t1 线程结束。t2 线程是 join()方法的调用者，而 t1 线程是被调用者，在调用 join()方法的过程中，方法的调用者被阻塞，阻塞到被调用的线程结束。

我们可以用一个生活中的比喻来解释 join()方法。假设顾客到饭店里去吃饭，那么每一个顾客就可以认为是一个线程。顾客点菜，就可以当做是顾客线程要求启动一个厨师线程为自己做饭，在做饭过程中，顾客线程只能等待。因此，这也可以当做是顾客线程调用了厨师线程的 join()方法，等厨师做完饭了，顾客才能继续下一步：吃饭。在这个例子中，顾客线程就调用了厨师线程的 join()方法。

在调用 join 方法的过程中，要注意，不能让两个线程相互 join()。例如，如果一个顾客点菜，相当于调用了厨师的 join()方法，顾客打算等厨师做完饭以后，吃完饭再给钱；而厨师呢，在拿到顾客下的单之后，希望顾客先给钱，之后再开始做饭，于是厨师也调用了顾客的 join()方法。这样的结果就是两边互相等待，结果谁都无法继续下去。我们拿代码模拟一下这种情况：

```
class MyThread1 extends Thread{
    Thread t;
    public void run(){
        try{
            t.join();
        }catch(Exception e){}
        for(int i = 0; i<100; i++){
            System.out.println(i + " $$$");
        }
    }
}

class MyThread2 extends Thread{
    Thread t;
    public void run(){
        try{
            t.join();
        }catch(Exception e){}
        for(int i = 0; i<100; i++){
            System.out.println(i + " ###");
        }
    }
}
```

```

public class TestJoin{
    public static void main(String args[]){
        MyThread1 t1 = new MyThread1();
        MyThread2 t2 = new MyThread2();
        t2.t = t1;
        t1.t = t2;
        t1.start();
        t2.start();
    }
}

```

这段代码会迟迟无法运行下去，原因就在于，**t1** 线程等待 **t2** 线程结束，而 **t2** 线程等待 **t1** 线程结束，这样的结果就是两边的线程都处于阻塞状态而无法运行。

那怎么解决这个问题呢？首先，写程序的时候应当小心，尽量不应该出现这样的代码。因为这样的代码在编译和运行时都不会出现任何错误和异常。另一方面，Java 也为 `join()` 方法提供了一个替换的方案。

在 `Thread` 类中，除了有一个无参的 `join()` 方法之外，还有一个有参的 `join()` 方法，方法签名如下：

```
public final void join(long millis) throws InterruptedException
```

这个方法接受一个 `long` 类型作为参数，表示 `join()` 最多等待多少毫秒。也就是说，调用这个 `join()` 方法的时候，不会一直处于阻塞状态，而是有一个时间限制。就好像顾客等待厨师做饭时，不会无限制的等下去，如果菜一段时间内还不上，则顾客就会离开，而不会一直傻等下去。利用这个方法，修改上面的 `MyThread1` 类：

```

class MyThread1 extends Thread{
    Thread t;
    public void run(){
        try{
            t.join(1000);
        }catch (Exception e){}
        for(int i = 0; i<100; i++){
            System.out.println(i + " $$$");
        }
    }
}

```

这样，`MyThread1` 就不会无限制的等待下去，而是当等待 1000 毫秒之后，就会从阻塞状态转为可运行状态，从而执行下去。

4 线程同步

下面要介绍的是线程中非常重要的一部分：线程同步。这部分的课程并不是太好理解，希望各位读者在学习过程中，能够耐心和细致一些。

4.1 临界资源与数据不一致

我们首先来看一个例子。我们使用数组来实现一个“栈”的数据结构。

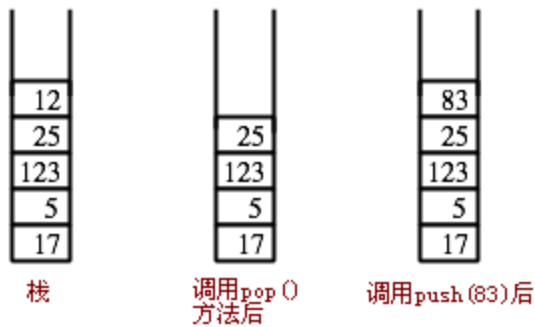
栈在表示上，就如同一个单向开口的盒子，每当有新数据进入时，都是进入栈顶。其基本操作为 `push` 和 `pop`。`push` 表示把一个元素加入栈顶，`pop` 表示把栈顶元素弹出。

示意图如下：

栈是一种数据结构

`pop()` 方法表示把栈顶端的元素返回，并删除该元素。如下图：调用 `pop()` 方法后

`push()` 方法表示把数据加入栈，加入时新数据放在栈顶。如下图：调用 `push(83)` 后



我们利用一个数组来实现栈操作，相关代码如下：

```
class MyStack{
    char[] data = {'A', 'B', ' '};
    int index = 2;
    public void push(char ch){
        data[index] = ch;
        index ++;
    }

    public void pop(){
        index --;
        data[index] = ' ';
    }

    public void print(){
        for(int i = 0; i<data.length; i++){
            System.out.print(data[i] + "\t");
        }
        System.out.println();
    }
}

public class TestMyStack{
    public static void main(String args[]){
        MyStack ms = new MyStack();
        ms.push('C');
        ms.print();
        ms.pop();
    }
}
```



```

        ms.print();
    }
}

```

上面的代码会输出：

```

A   B   C
A   B

```

符合我们对栈的认识。**MyStack** 类中的 **data** 属性表示用来储存栈的信息，而 **index** 这个变量保存的是栈中有效元素的个数。

然后，我们为 **push** 方法增加一个 **Thread.sleep()** 语句。修改后的 **push()** 方法如下：

```

public void push(char ch) {
    data[index] = ch;
    try{
        Thread.sleep(1000);
    }catch (Exception e) {}
    index ++;
}

```

在修改 **data** 值和维护 **index** 中间，增加了一个 **sleep()** 方法。增加了这个方法之后，运行结果没有区别。

接下来，我们使用多线程来访问 **MyStack** 类。创建两个线程类：**PopThread** 和 **PushThread**，这两个类用来访问 **MyStack** 对象。**PushThread** 线程负责向栈中添加一个字符，而 **PopThread** 线程负责从栈中取出一个字符。修改后完整的代码如下：

```

class MyStack{
    char[] data = {'A', 'B', ' '};
    int index = 2;
    public void push(char ch) {
        data[index] = ch;
        try{
            Thread.sleep(1000);
        }catch (Exception e) {}
        index ++;
    }

    public void pop(){
        index --;
        data[index] = ' ';
    }

    public void print(){
        for(int i = 0; i<data.length; i++){
            System.out.print(data[i] + "\t");
        }
        System.out.println();
    }
}

```

```

}

class PopThread extends Thread{
    MyStack ms;
    public PopThread(MyStack ms){
        this.ms = ms;
    }
    public void run(){
        ms.pop();
        ms.print();
    }
}

class PushThread extends Thread{
    MyStack ms;
    public PushThread(MyStack ms){
        this.ms = ms;
    }
    public void run(){
        ms.push('C');
        ms.print();
    }
}

public class TestMyStack{
    public static void main(String args[]){
        MyStack ms = new MyStack();
        Thread t1 = new PushThread(ms);
        Thread t2 = new PopThread(ms);

        t1.start();
        t2.start();
    }
}

```

上述代码的输出结果为：

```

A      C
A      C

```

注意，这就产生了问题：我们的栈要求后入先出，栈内原有两个元素 A、B，结果先启动了 Push 线程，后启动 Pop 线程之后，却把 B 元素 pop 出去，而把 C 元素保留了。更关键的是，C 元素和 A 元素中有一个空位，也就是说，栈内的元素变得不连续了！

很显然，这是一个程序运行中的错误。那为什么会造成这个错误呢？

我们结合线程状态的转换来看这个问题。在主方法中，t1 和 t2 线程都被调用了 start()

方法被启动。假设 t1 线程先进入运行状态，而 t2 线程保持在可运行状态。此外，要注意的是，t1 线程和 t2 线程中，都包含有一个 MyStack 类型的引用。在主方法中，我们只创建了一个 MyStack 类型的对象，并且通过构造方法，让 t1 和 t2 的 ms 属性都指向了同一个 MyStack 类型的对象。

在 t1 线程进入运行状态之后，它会调用 MyStack 类中的 push('C')方法。这个在执行这个方法过程中，会首先修改 data 数组的值，把下标为 2 的位置设为 C。之后，应当把 index 加 1，但是线程调用了 sleep()方法，使得 t1 线程进入了阻塞状态。此时，data 数组中有 3 个元素，但是 index 却是 2！这个时候，表示栈的两个数据 data 数组和 index，出现了信息不一致的情况。

当 t1 处于阻塞状态的时候，t2 线程处于可运行状态。于是，操作系统就会选中 t2 线程进行运行。此时，t2 的 run()方法中会调用 pop 方法，会首先把 index 减 1，使得 index 的值为 1，然后把 data 数组中下标为 1 的元素设为空。要注意的是，现在栈顶元素应当是 data 数组中下标为 2 的 C，而不是下标为 1 的元素 B！这样就造成了我们上面运行的结果，破坏了栈的数据结构。

下图说明了元素入栈和出栈的正确情况：

0	1	2	index=	运行状态线程	代码	备注
A	B		2			初始状态
A	B	C	2	t1	data[index]='C'	t1 线程执行了 C 字符入栈
A	B	C	3	t1	index++	t1 线程执行 index 加 1, 入栈过程完成
A	B	C	2	t2	index--	
A	B		2	t2	data[index]=''	t2 线程执行出栈过程完成 打印 'C' 出栈

而下图则揭示了出现错误的数组情况：

0	1	2	index=	运行状态线程	代码	备注
A	B		2			初始状态
A	B	C	2	t1	data[index]='C'	t1 线程执行了 C 字符入栈
A	B	C	1	t2	index--	t1 休眠 t2 获得 CPU 时间片
A		C	1	t2	data[index]=''	t2 执行出栈过程完成 打印 'B' 出栈
A		C	2	t1	index++	t1 恢复运行，执行 index++

造成这种错误的原因是什么呢？第一个原因：t1 线程中的两个步骤：1、修改 data 数组；2、index 变量加 1。这两个步骤当一起完成，才能组成一个完整的 push()操作；如果这两个步骤中只完成了一个，就会产生问题。具体来说，就会产生数据不一致的问题，对于 MyStack

来说，所谓的数据不一致，指的就是 `data` 数组中元素的个数和 `index` 所表示的元素的个数不一致。

第二个原因，则是因为有 `t1` 和 `t2` 两个线程，这两个线程并发访问同一个 `MyStack` 对象。由于当 `t1` 线程没有完成的操作的时候，`t2` 线程就开始对 `MyStack` 对象进行了访问，从而就会造成数据不一致。

总结一下上面的两个原因：多个线程并发访问同一个对象，如果破坏了不可分割的操作，则有可能产生数据不一致的情况。

这其中，有两个专有名词：被多个线程并发访问的对象，也被称之为“临界资源”；而不可分割的操作，也被称之为“原子操作”。产生的数据不一致的问题，也被称之为“同步”问题。要产生同步问题，多线程访问“临界资源”，破坏了“原子操作”，这两个条件缺一不可。

4.2 synchronized

4.2.1 锁的概念

分析了产生同步问题的原因之后，下面就应该设法解决同步问题。那同步问题应该怎么处理和解决呢？

首先，由于在 `push()` 方法中存在 `sleep()` 方法才造成了原子操作被破坏，那如果把 `sleep()` 方法去掉，是不是就能解决同步的问题了呢？

要注意的是，把 `sleep()` 方法去掉之后，这样的代码可能运行时一时不会出问题，但是并不代表这代码是没有安全问题的。例如，当 `push` 线程进行了 `push` 的第一步操作之后，CPU 时间片到期了，然后接下来换成 `pop` 线程运行，此时，上一节我们描述的同步问题，同样有可能发生。

虽然上面所说的情况可能发生的概率非常的低，但是却不能不说是代码中的一个隐患。并且，当我们的程序成为一个大型企业应用系统的一部分时，由于每天都有大量的客户访问我们的程序，也就有大量的线程在同时访问同一个对象，此时，无论发生同步问题的概率有多小，在大量访问和重复的过程中，发生问题几乎是必然的。而同步问题一旦发生，就有可能造成极为严重的损失。因此，我们在写代码，尤其是有可能被多线程访问的类和对象时，一定要慎重设计，把所有的隐患都杜绝。

那应该怎么杜绝这个隐患呢？我们首先从生活中的一个例子说起。

有两位电工，长期驻扎在一个小区作为物业，一个电工 A，上早班，上班时间是 6: 00 ~ 18: 00，另一位电工 B，上班时间为 18: 00 ~ 6: 00。这两位电工轮流在小区值班。

某一天，在下午 17: 55 的时候，电工 A 接到小区居民投诉：小区中的电压不稳，希望电工能够修复一下。虽然马上要到下班时间了，但是电工 A 还是决定去查看一下。为了爬上电线杆进行高空带电作业，于是电工 A 暂时把小区的电闸给关了，然后再到高空进行电压不稳的检查。

在 A 在工作过程中，不知不觉到了 18: 01，电工 B 上班了。这个时候，电工 B 接到小区居民投诉：小区停电了。电工 B 也决定去查看一下。当他查看到电闸时，一下就明白了：怪不得小区停电呢，谁把闸关了啊。于是，电工 B 就把电闸打开了。

于是，听到一声惨叫，A 从天上掉了下来……

半个月以后，等 A 把伤养得差不多了以后，变电所决定吸取教训，争取再也不要发生这样的问题。那这个问题是怎么发生的呢？

我们可以把 A 和 B 当做是两个线程，而电闸就当做是临界资源。因此，这样就形成了两个线程共同访问临界资源，由于 A 检修电路时，“关掉电闸、爬上电线杆检修、打开电闸恢复供电”是不可分割的原子操作，而一旦其中的某一步被打断，就有可能产生问题，这就是两个线程数据不一致的情况。

那怎么办呢？变电所决定，要解决这个问题，这就借助于一件东西：锁。在电闸上挂一个挂锁。平时没有问题时，锁是打开的；但是一旦有一个电工需要操作电闸的话，为了防止别人动电闸，他可以把电闸给锁上，并把唯一的钥匙随身携带。这样，当他进行原子操作时，由于临界资源被他锁上了，其他线程就访问不了这个临界资源，因此就能保证他的原子操作不被破坏。

4.2.2 synchronized 与同步代码块

Java 中采取了类似的机制，也采用锁来保护临界资源，防止数据不一致的情况产生。下面我们就来介绍 Java 中的同步机制以及 `synchronized` 关键字。

在 Java 中，每个对象都拥有一个“互斥锁标记”，这就好比是我们说的挂锁。这个锁标记，可以用来分给不同的线程。之所以说这个锁标记是“互斥的”，因为这个锁标记同时只能分配给一个线程。

光有锁标记还不行，还要利用 `synchronized` 关键字进行加锁的操作。`synchronized` 关键字有两种用法，我们首先介绍第一种：`synchronized + 代码块`。

这种用法的语法如下：

```
synchronized(obj){  
    代码块...  
}
```

`synchronized` 关键字后面跟一个圆括号，括号中的是某一个引用，这个引用应当指向某一个对象。后面紧跟一个代码块，这个代码块被称为“同步代码块”。

这种语法的含义是，如果某一个线程想要执行代码块中的代码，必须先获得 `obj` 所指向对象的互斥锁标记。也就是说，如果有一个线程 `t1` 要想进入同步代码块，必须先获得 `obj` 对象的锁标记；而如果 `t1` 线程正在同步代码块中运行，这意味着 `t1` 有着 `obj` 对象的互斥锁标记；而这个时候如果有一个 `t2` 线程想要访问同步代码块，会因为拿不到 `obj` 对象的锁标记而无法继续运行下去。

需要注意的是，`synchronized` 与同步代码块是与对象紧密结合在一起的，加锁是对对象加锁。例如下面的例子，假设有两个同步代码块：

```
synchronized(obj1){  
    代码块 1;  
}  
  
synchronized(obj1){  
    代码块 2;  
}  
  
synchronized(obj2){  
    代码块 3;
```

```
}
```

假设有一个线程 **t1** 正在代码块 1 中运行，那假设另有一个线程 **t2**，这个 **t2** 线程能否进入代码块 2 呢？能否进入代码块 3 呢？

由于 **t1** 正在代码块 1 中运行，这也就意味着 **obj1** 对象的锁标记被 **t1** 线程获得，而此时 **t2** 线程如果要进入代码块 2，也必须获得 **obj1** 对象的锁标记。但是由于这个标记正在 **t1** 手中，因此 **t2** 线程无法获得锁标记，因此 **t2** 线程无法进入代码块 2。

但是 **t2** 线程能够进入代码块 3，原因在于：如果要进入代码块 3 中，要获得的是 **obj2** 对象的锁标记，这个对象与 **obj1** 不是同一个对象，此时 **t2** 线程能够顺利的获得 **obj2** 对象的锁标记，因此能够成功的进入代码块 3。

从上面这个例子中，我们可以看出，在分析、编写同步代码块时，一定要搞清楚，同步代码块锁的是哪个对象。只有把这个问题搞清楚了之后，才能正确的分析多线程以及同步的相关问题。

我们下面利用同步代码块修改一下 **MyStack** 类，为这个类增加同步的机制。

首先，要为 **MyStack** 类增加一个属性 **lock**，这个属性用来表示我们所说的“锁”。利用这个对象的互斥锁标记，我们完成对 **MyStack** 的同步。

修改之后的 **MyStack** 代码如下：

```
class MyStack{
    char[] data = {'A', 'B', ' '};
    int index = 2;
    private Object lock = new Object();
    public void push(char ch){
        synchronized(lock){
            data[index] = ch;
            try{
                Thread.sleep(1000);
            }catch(Exception e){}
            index ++;
        }
    }

    public void pop(){
        synchronized(lock){
            index --;
            data[index] = ' ';
        }
    }

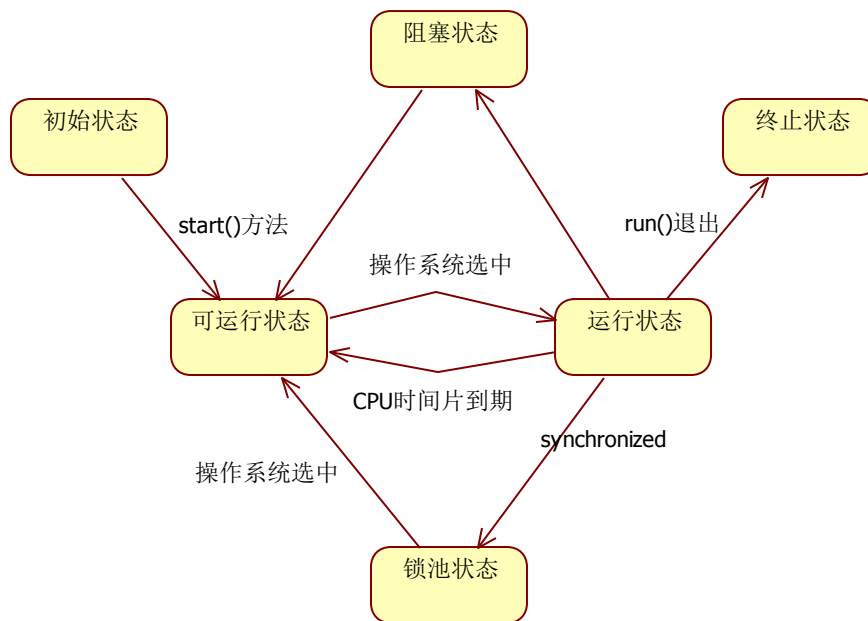
    public void print(){
        for(int i = 0; i<data.length; i++){
            System.out.print(data[i] + "\t");
        }
        System.out.println();
    }
}
```

}

在 `push` 方法和 `pop` 方法中，我们对 `lock` 属性进行加锁。也就是说，线程如果要执行 `pop` 或者 `push` 时，需要获得 `lock` 对象的互斥锁标记。

下面我们结合线程的状态转换，来考察一下 `synchronized` 关键字在程序运行中的作用。

首先，如果一个线程获得不了某个对象的互斥锁标记，这个线程就会进入一个状态：锁池状态。如下图：



当运行中的线程，运行到某个同步代码块，但是获得不了对象的锁标记时，会进入锁池状态。在锁池状态的线程，会一直等待某个对象的互斥锁标记。如果有多个线程都需要获得同一个对象的互斥锁标记，则可以有多个线程进入锁池，而某个线程获得锁标记，执行同步代码块中的代码。

当对象的锁标记被某一个线程释放之后，其他在锁池状态中的线程就可以获得这个对象的锁标记。假设有多个线程在锁池状态中，那么会由操作系统决定，把释放出来的锁标记分配给哪一个线程。当在锁池状态中的线程获得锁标记之后，就会进入可运行状态，等待获得 CPU 时间片，从而运行代码。

下面我们结合线程状态的转换，来分析一下修改之后的 `MyStack` 代码。

当 `t1` 线程启动的时候，它会首先执行 `push` 方法。要想执行 `push` 方法，必须要获得 `lock` 对象的互斥锁标记。由于此时 `lock` 对象的锁标记没有分配给其他线程，因此这个锁标记被分配给 `t1` 线程。当 `t1` 线程调用 `sleep()` 方法之后，`t1` 进入阻塞状态，于是 `t2` 线程开始运行。由于 `t2` 线程调用 `pop()` 方法，要调用这个方法，也必须获得 `lock` 对象的互斥锁标记。由于这个锁标记已经分配给了 `t1` 线程，因此 `t2` 线程只能进入 `lock` 对象的锁池，从而进入了锁池状态。

之后，当 `t1` 线程 `sleep()` 方法结束，`t1` 重新进入可运行状态→运行状态，执行完了整个 `push()` 方法，退出对 `lock` 加锁的同步代码块。此时，`t2` 线程就能获得 `lock` 对象的锁标记，于是 `t2` 线程就由锁池状态转为了可运行状态。

从上面的分析我们可以看出，虽然 `MyStack` 对象还是被两个线程 `t1`、`t2` 同时访问，但是由于对 `lock` 对象加锁的原因，当 `t1` 线程执行 `push` 方法执行到一半进入阻塞状态时，`t2` 线程同样无法操作 `MyStack` 对象。这样，就不会破坏原子操作 `push()`，从而保护了临界资源，解决了同步问题。

加入了锁机制之后的数组情况如下图：

0	1	2	index=	运行状态线程	代码	备注
A	B		2			初始状态
A	B	C	2	t1	<code>data[index]='C'</code>	t1 线程执行了 C 字符入栈
A	B	C	2	t2		t1 休眠，但 t2 无法获得锁对象的锁标记，因此 t2 无法执行代码
A	B	C	3	t1	<code>index++</code>	t1 线程执行 <code>index</code> 加 1，入栈过程完成
A	B	C	2	t2	<code>index--</code>	t2 获得锁标记，执行代码
A	B		2	t2	<code>data[index]=''</code>	t2 线程执行出栈过程完成 打印 'C' 出栈

4.2.3 同步方法

在上面的例子中，我们专门创建了一个 `Object` 类型的 `lock` 对象，用来在 `pop` 方法和 `push` 方法中加锁。然而，对于上面的程序来说，除了 `lock` 对象有锁标记之外，`MyStack` 对象本身，也具有互斥锁标记。对于 `pop` 方法和 `push` 方法来说，也能够对 `MyStack` 对象（也就是所谓的“当前对象”）加锁。例如：

```
class MyStack{
    char[] data = {'A', 'B', ' '};
    int index = 2;
    public void push(char ch){
        synchronized(this){
            data[index] = ch;
            try{
                Thread.sleep(1000);
            }catch(Exception e){}
            index ++;
        }
    }

    public void pop(){
        synchronized(this){
            index --;
            data[index] = ' ';
        }
    }
}
```



```

    }

    public void print(){
        for(int i = 0; i<data.length; i++){
            System.out.print(data[i] + "\t");
        }
        System.out.println();
    }

}

```

在上面的代码中，我们去掉了 `lock` 属性，并且对 “`this`” 加锁，也就是对当前对象加锁。上面的代码同样能够完成我们同步的要求。

对于这种，在整个方法内部对 “`this`” 加锁的情况，我们可以使用 `synchronized` 作为修饰符修饰方法，来表达同样的意思。

用 `synchronized` 关键字修饰的方法称之为同步方法，所谓的同步方法，指的是同步方法中整个方法的实现，需要对 “当前对象” 加锁。哪个线程能够拿到对象的锁标记，哪个线程才能调用对象的同步方法。

上面的 `MyStack` 代码可以等价的改为下面的情况：

```

class MyStack{
    char[] data = {'A', 'B', ' '};
    int index = 2;
    public synchronized void push(char ch){
        data[index] = ch;
        try{
            Thread.sleep(1000);
        }catch(Exception e){}
        index ++;
    }

    public synchronized void pop(){
        index --;
        data[index] = ' ';
    }

    public void print(){
        for(int i = 0; i<data.length; i++){
            System.out.print(data[i] + "\t");
        }
        System.out.println();
    }

}

```

当一个线程 `t1` 在访问某一个对象的同步方法时（例如 `pop`），另一个线程 `t2` 能否访问同

一个对象的其他同步方法（例如 `push`）？

我们举例来说：假设有一个 `MyStack` 对象 `ms`，一个线程 `t1` 正在访问 `pop` 方法，这意味着 `ms` 对象的互斥锁标记正在 `t1` 线程手中。而另一个线程 `t2`，如果想要方法 `push` 方法的话，必须要获得 `ms` 对象的互斥锁标记。由于这个锁标记正在 `t1` 线程手中，因此 `t2` 线程无法获得，从而 `t2` 线程就无法访问 `push` 方法。因此，这是一个结论：

当一个线程正在访问某个对象的同步方法时，其他线程不能访问同一个对象的任何同步方法。

5 wait 与 notify

在 `synchronized` 关键字的作用下，还有可能产生新的问题：死锁。

5.1 死锁

考虑下面的代码，假设 `a` 和 `b` 是两个不同的对象。

```
synchronized(a){
    ...//1
    synchronized(b){
    }
}
synchronized(b){
    ...//2
    synchronized(a){
    }
}
```

假设现在有两个线程，`t1` 线程运行到了 `//1` 的位置，而 `t2` 线程运行到了 `//2` 的位置，接下来会发生什么情况呢？

此时，`a` 对象的锁标记被 `t1` 线程获得，而 `b` 对象的锁标记被 `t2` 线程获得。对于 `t1` 线程而言，为了进入对 `b` 加锁的同步代码块，`t1` 线程必须获得 `b` 对象的锁标记。由于 `b` 对象的锁标记被 `t2` 线程获得，`t1` 线程无法获得这个对象的锁标记，因此它会进入 `b` 对象的锁池，等待 `b` 对象锁标记的释放。而对于 `t2` 线程而言，由于要进入对 `a` 加锁的同步代码块，由于 `a` 对象的锁标记在 `t1` 线程手中，因此 `t2` 线程会进入 `a` 对象的锁池。

此时，`t1` 线程在等待 `b` 对象锁标记的释放，而 `t2` 线程在等待 `a` 对象锁标记的释放。由于两边都无法获得所需的锁标记，因此两个线程都无法运行。这就是“死锁”问题。

5.2 wait 与 notify

在 `Java` 中，采用了 `wait` 和 `notify` 这两个方法，来解决死锁机制。

首先，在 `Java` 中，每一个对象都有两个方法：`wait` 和 `notify` 方法。这两个方法是定义在 `Object` 类中的方法。对某个对象调用 `wait()` 方法，表明让线程暂时释放该对象的锁标记。例如，上面的代码就可以改成：

```
synchronized(a){
    ...//1
    a.wait();
    synchronized(b){
```

```

    }
}
synchronized(b){
    ...//2
    synchronized(a){
        ...
        a.notify();
    }
}
}

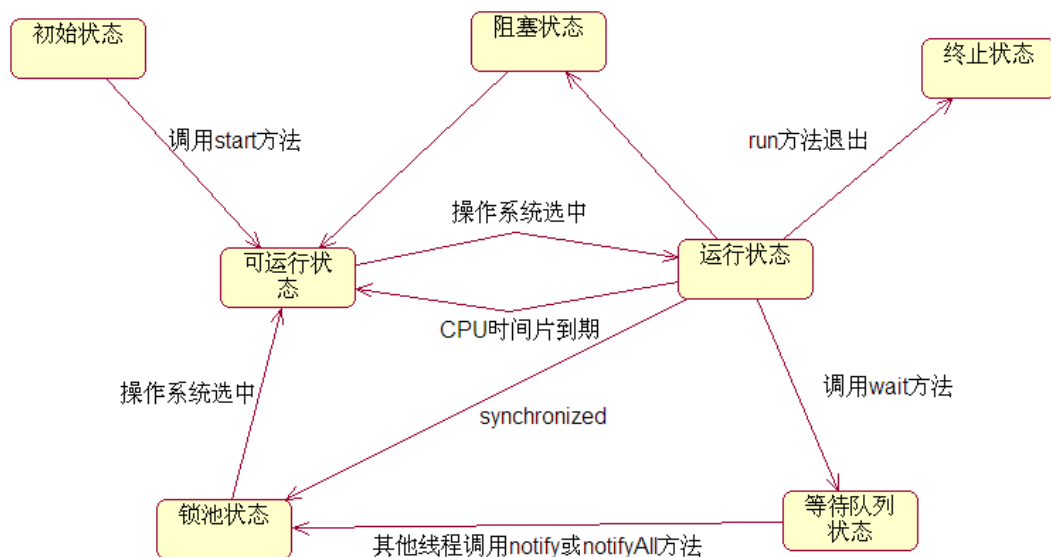
```

这样的代码改完之后，在//1 后面，t1 线程就会调用 a 对象的 wait 方法。此时，t1 线程会暂时释放自己拥有的 a 对象的锁标记，而进入另外一个状态：等待状态。

要注意的是，如果要调用一个对象的 wait 方法，前提是线程已经获得这个对象的锁标记。如果在没有获得对象锁标记的情况下调用 wait 方法，则会产生异常。

由于 a 对象的锁标记被释放，因此，t2 对象可以获得 a 对象的锁标记，从而进入对 a 加锁的同步代码块。在同步代码块的最后，调用 a.notify()方法。这个方法与 wait 方法相对应，是让一个线程从等待状态被唤醒。

那么 t2 线程唤醒 t1 线程之后，t1 线程处于什么状态呢？由于 t1 线程唤醒之后还要在对 a 加锁的同步代码块中运行，而 t2 线程调用了 notify()方法之后，并没有立刻退出对 a 加锁的同步代码块，因此此时 t1 线程并不能马上获得 a 对象的锁标记。因此，此时，t1 线程会在 a 对象的锁池中进行等待，以期待获得 a 对象的锁标记。也就是说，一个线程如果之前调用了 wait 方法，则必须要被另一个线程调用 notify()方法唤醒。唤醒之后，会进入锁池状态。线程状态转换图如下：



由于可能有多个线程先后调用 a 对象 wait 方法，因此在 a 对象等待状态中的线程可能有多。而调用 a.notify()方法，会从 a 对象等待状态中的多个线程里挑选一个线程进行唤醒。与之对应的，有一个 notifyAll()方法，调用 a.notifyAll() 会把 a 对象等待状态中的所有线程都唤醒。

5.3 wait 与 notify 应用：生产者/消费者问题

下面我们结合一个实际的例子，来为大家演示如何使用 `wait` 和 `notify/notifyAll` 方法。我们使用一个数组来模拟一个我们比较熟悉的数据结构：栈。代码如下所示：

```
class MyStack{
    private char[] data = new char[5];
    private int index = 0;

    public char pop(){
        index -- ;
        return data[index];
    }

    public void push(char ch){
        data[index] = ch;
        index++;
    }

    public void print(){
        for (int i=0; i<index; i++){
            System.out.print(data[i] + "\t");
        }
        System.out.println();
    }

    public boolean isEmpty(){
        return index == 0;
    }

    public boolean isFull(){
        return index == 5;
    }
}
```

注意，我们为 `MyStack` 增加了两个方法，一个用来判断栈是否为空，一个用来判断栈是否已经满了。

然后我们创建两个线程，一个线程每隔一段随机的时间，就会往栈中增加一个数据；另一个线程每隔一段随机的时间，就会从栈中取走一个数据。为了保证 `push` 和 `pop` 的完整性，在线程中应当对 `MyStack` 对象加锁。

但是我们发现，入栈线程和出栈线程并不是在任何时候都可以工作的。当数组满了的时候，入栈线程将不能工作；当数组空了的时候，出栈线程也不能工作。违反了上面的条件，我们将得到一个数组下标越界异常。

为此，我们可以用 `wait/notify` 机制。在入栈线程执行入栈操作时，如果发现数组已满，则会调用 `wait` 方法，去等待。同样，出栈线程在执行出栈操作时，如果发现数组已空，同样调用 `wait` 方法去等待。在入栈线程结束入栈工作之后，会调用 `notifyAll` 方法，释放那些正在等待的出栈线程（因为数组现在已经不是空的了，他们可以恢复工作了）。同样，当出栈线程结束出栈工作之后，也会调用 `notifyAll` 方法，释放正在等待的入栈线程。

相关代码如下：

```
class Consumer extends Thread{
    private MyStack ms;

    public Consumer(MyStack ms) {
        this.ms = ms;
    }

    public void run(){
        while(true){
            //为了保证 push 和 pop 操作的完整性
            //必须加 synchronized
            synchronized(ms) {
                //如果栈空间已满，则 wait() 释放 ms 的锁标记
                while(ms.isEmpty()){
                    try {
                        ms.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                char ch = ms.pop();
                System.out.println("Pop " + ch);
                ms.notifyAll();
            }
            //push 之后随机休眠一段时间
            try {
                sleep( (int)Math.abs(Math.random() * 100) );
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

//生产者
class Producer extends Thread{
    private MyStack ms;

    public Producer(MyStack ms) {
        this.ms = ms;
    }
}
```

```

public void run(){
    while(true){
        //为了保证 push 和 pop 操作的完整性
        //必须加 synchronized
        synchronized(ms) {
            //如果栈空间已满，则 wait() 释放 ms 的锁标记
            while(ms.isFull()){
                try {
                    ms.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            ms.push('A');
            System.out.println("push A");
            ms.notifyAll();
        }
        //push 之后随机休眠一段时间
        try {
            sleep( (int)Math.abs(Math.random() * 200) );
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class TestWaitNotify {
    public static void main(String[] args) {
        MyStack ms = new MyStack();
        Thread t1 = new Producer(ms);
        Thread t2 = new Consumer(ms);
        t1.start();
        t2.start();
    }
}

```