Java语言与系统设计

第9章 多线程

- 程序、进程和线程
- 多线程的创建
- 多线程的实现
- 多线程的同步
- 死锁
- 多线程的通信

1 程序、进程和线程

- 同学们也许对操作系统中的多任务已经很熟悉了，所谓的多任务，是指在同一时刻有多个程序在同时运行的能力。例如，可以在进行QQ聊天的同时，进行下载并播放音乐。
- 当然，除非计算机中有多个CPU，否则操作系统会将CPU的时间划分成小的片段，并将其分配给不同的程序，使人产生一种并行处理的错觉。这种资源分配方法之所以可行，是因为CPU的处理速度非常快，大部分时间是空闲的。

- 程序（program）：是计算机指令的集合，它以文件的形式存储在磁盘上，是应用程序执行的静态代码。
- 进程（process）：是一个程序在其自身的地址空间中的一次执行活动。进程是资源申请、调度和独立运行的单位，使用系统中的运行资源，具有生命周期。
- 程序不能申请系统资源，不能被系统调度，也不能作为独立运行的单位，它不占用系统的运行资源。程序可以被多次加载到系统的不同内存区域分别执行，形成不同的进程。允许计算机同时运行两个或更多的程序。

- 线程（thread）：是进程中的一个单一的顺序控制流程，一个进程在执行过程中，可以产生多个线程。每个线程也有自己产生、存在和消亡的过程。
- 线程又称为轻量级进程，它和进程一样拥有独立的执行控制，由操作系统负责调度，每个线程拥有独立的运行栈和程序计数器，线程切换的开销小。
- 一个进程中的多个线程共享相同的内存单元/内存地址空间，则它们从同一堆中分配对象，可以访问相同的变量和对象。这就使得线程间通信更简便、高效。但多个线程操作共享的系统资源可能就会带来安全的隐患。

- 线程(Thread)和进程(Process)的关系很紧密，进程和线程是两个不同的概念，进程的范围大于线程。
- 通俗地说，进程就是一个程序，线程是这个程序能够同时做的各件事情。比如，媒体播放机运行时就是一个进程，而媒体播放机同时做的下载文件和播放歌曲，就是两个线程。因此，可以说进程包含线程。
- 从另一个角度讲，每个进程都拥有一组完整的属于自己的变量，而线程则共享一个进程内的这些数据。

- 主线程：程序启动时，一个线程立刻运行，该线程通常称为程序的主线程。在Java中，main（）就是主线程。其他子线程都是由主线程产生的，主线程通常必须最后完成执行，因其需执行各种关闭动作。下面通过程序来演示主线程。

执行主线程示例

```
public class MainThreadDemo  {
   public static void main(String args[]) {
      Thread t = Thread.currentThread();
      System.out.println("当前线程名称是: " + t.getName());
      System.out.println("输出当前线程: " + t);
   }
}
```

代码分析:

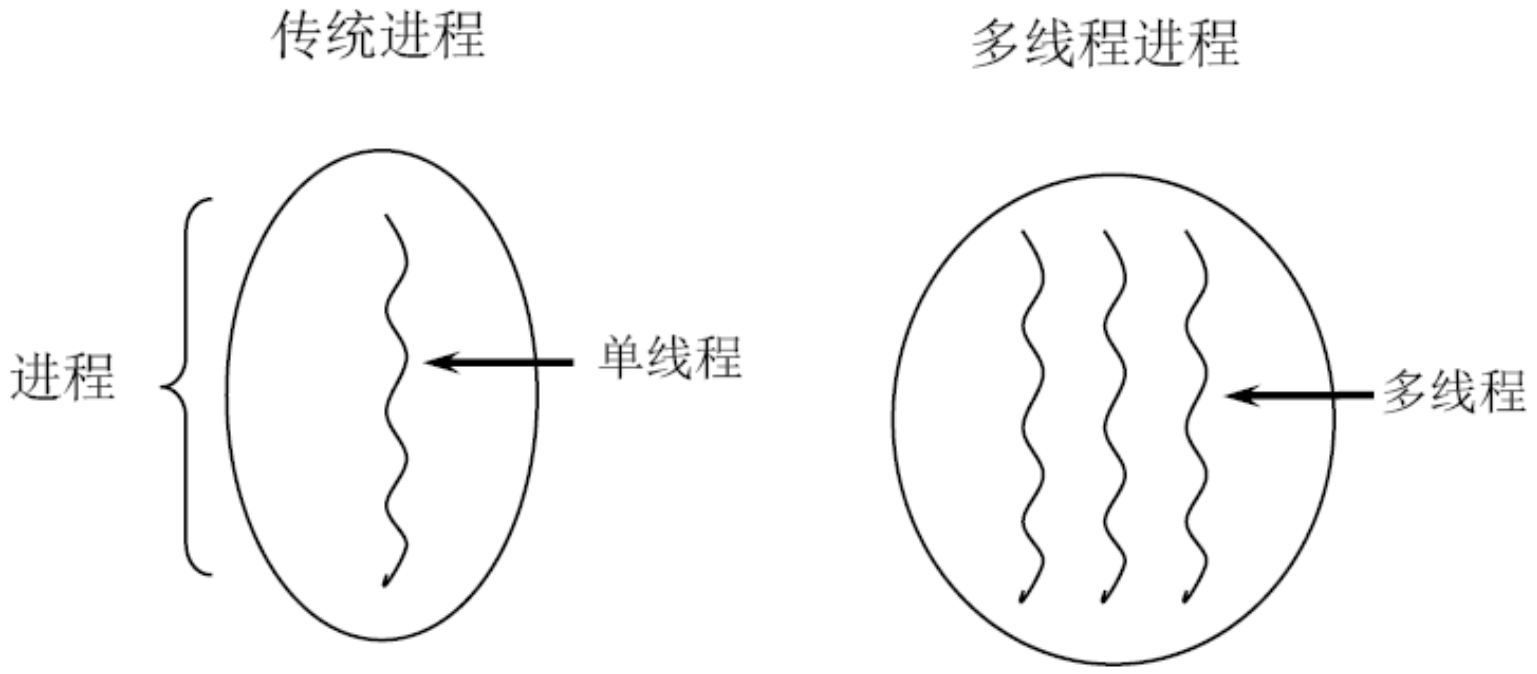
1）Thread.currentThread() 是一个静态方法，返回正在执行的线程对象的引用。我们知道main函数是程序的入口点，这里引用的线程就是主线程，通过getName( )方法得到当前引用线程的名称，从程序的结果我们可以看出，名称是main。

2）当将一个线程对象作为输出时，将产生一个数组输出，其格式为[线程名称，优先级别，线程组名]。这里的输出是[main,5,main]。

3）每个线程都属于一个线程组，如果没有制定，那么由JVM来设定。

- 一个Java应用程序java.exe，其实至少有三个线程：main()主线程，gc()垃圾回收线程，异常处理线程。当然如果发生异常，会影响主线程。

怎么同时运行多个线程呢?


传统进程
进程
单线程
多线程进程
多线程

- 实现多线程方法：并行和并发

（1）并发：一个CPU(采用时间片)同时执行多个任务。

（2）并行：多个CPU同时执行多个任务。比如：多个人同时做不同的事。

- 单核和多核CPU

（1）单核CPU，其实是一种假的多线程，因为在一个时间单元内，也只能执行一个线程的任务。

（2）如果是多核的话，才能更好的发挥多线程的效率。（现在的服务器、手机都是多核的）

- 多线程优点:

(1) 提高应用程序的响应。对图形化界面更有意义，可增强用户体验。

(2) 现在CPU的主频很高，提高计算机系统CPU的利用率。

(3) 编程时将既长又复杂的进程分为多个线程，改善程序结构，独立运行，利于理解和修改

- 何时需要多线程：：

（1）程序需要同时执行两个或多个任务。比如main线程和异常处理线程。

（2）程序需要实现一些需要等待的任务时，如网页浏览中采用多线程下载不同对象等。

（3）需要一些后台运行的程序时。

习题

- Java为什么要引入线程机制，线程、程序、进程之间的关系是怎样的。

答：线程可以彼此独立的执行，它是一种实现并发机制的有效手段，可以同时使用多个线程来完成不同的任务，并且一般用户在使用多线程时并不考虑底层处理的细节。

程序是一段静态的代码，是软件执行的蓝本。进程是程序的一次动态执行过程，即是处于运行过程中的程序。

线程是比进程更小的程序执行单位，一个进程可以启动多个线程同时运行，不同线程之间可以共享相同的内存区域和数据。多线程程序是一个进程之内的，有一个以上线程同时运行的情况的程序。

2 创建多线程

- 在计算机编程中有一个基本概念，就是在同一时刻处理多个任务的思想。
- 多线程的任务相比传统的进程而言（只有一个主线程），可以同时有多个地方执行代码。这种方式如何实现的呢?

顺序执行实例

```
package thread;
public class ThreadTest1 {
public static void main(String[] args) throws Exception {
System.out.println("传送文件1");
Thread.sleep(1000 * 10);
System.out.println("文件1传送完毕");
System.out.println("传送文件2");
Thread.sleep(1000 * 10);
System.out.println("文件2传送完毕");
System.out.println("传送文件3");
Thread.sleep(1000 * 10);
System.out.println("文件3传送完毕");
}
}
```

不是多线程

- 最初，程序员们用所掌握的有关机器底层的知识来编写中断服务程序，主进程的挂起是通过硬件中断来触发的。
- 尽管这么做可以解决问题，但是其难度太大，而且不能移植，所以使得将程序移植到新型号的机器上时，既费时又费力。

- 我们只是想把问题切分成多个可独立运行的部分（任务），从而提高程序的响应能力。在程序中，这些彼此独立运行的部分称之为线程。

2 多线程的创建

- Java提供了类 java.lang.Thread 来方便多线程编程，这个类提供了大量的方法方便控制线程.
- Thread 类最重要的方法是 run() ，它为 Thread 类的方法 start() 所调用。为了指定我们自己的代码，需要覆盖run()方法，来提供我们的线程所要执行的代码。

2.1 继承Thread类创建线程

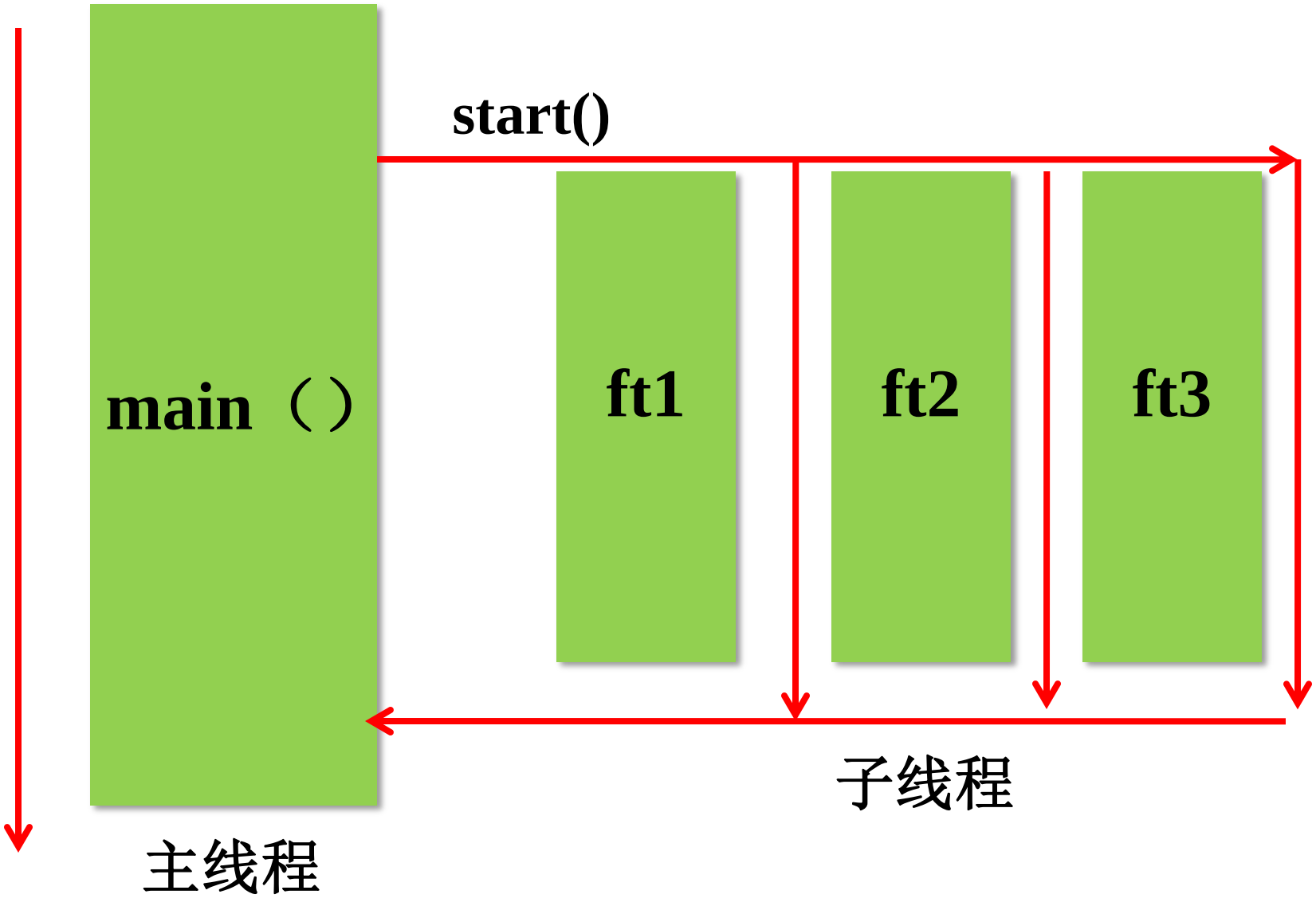
- 方法一：继承java.lang.Thread类，覆盖run() 方法。
- 在创建的 Thread 类的子类中重写 run() ,加入线程所要执行的代码即可。

```
class mythread extends Thread {
 public void run( ) {
    /* 覆盖该方法*/ }}
```

并行执行实例

```
package thread;
class FileTransThread extends Thread{
private String fileName;
public FileTransThread(String fileName){
this.fileName = fileName;
}
public void run(){
System.out.println("传送" + fileName);
try{
Thread.sleep(1000 * 10);
}catch(Exception ex){}
System.out.println(fileName + "传送完毕");
}
}
```

```
public class ThreadTest2 {
public static void main(String[] args) throws
      Exception {
FileTransThread ft1 = new
      FileTransThread("文件1");
FileTransThread ft2 = new
      FileTransThread("文件2");
FileTransThread ft3 = new
      FileTransThread("文件3");
ft1.start();
ft2.start();
ft3.start();
}
}
```


start()
main（）
ft1
ft2
ft3
子线程
主线程

- 线程为什么能够实现“程序看起来同时做好几件事情”的功能呢？这主要和操作系统的运行机制有关。
- 多线程的机制实际上相当于CPU交替分配给不同的代码段来运行：也就是说，某一个时间片，某线程运行，下一个时间片，另一个线程运行，各个线程都有抢占CPU的权利，至于决定哪个线程抢占，是操作系统需要考虑的事情。
- 由于时间片的轮转非常快，用户感觉不出各个线程抢占CPU的过程，看起来好像计算机在“同时”做好几件事情。

继承Thread类创建线程

1) 定义子类继承Thread类。
2) 子类中重写Thread类中的run方法。
3) 创建Thread子类对象，即创建了线程对象。
4) 调用线程对象start方法：启动线程，调用run方法。

注意：

1、如果自己手动调用run()方法，那么就只是普通方法，没有启动多线程模式。想要启动多线程，必须调用start方法。

2、一个线程对象只能调用一次start()方法启动，如果重复调用了，则将抛出以上的异常“IllegalThreadStateException”。

3、 run()方法由JVM调用，什么时候调用，执行的过程控制都有操作系统的CPU调度决定。

2.2 实现Runnable接口创建线程

- 继承java.lang.Thread类方法简单明了，符合大家的习惯，但是，它也有一个很大的缺点. 那就是如果我们的类已经从一个类继承则无法再继承 Thread 类，这时如果我们又不想建立一个新的类，该怎么办？这就是我们下面来讨论的方法二。

- 方法二：实现java.lang.Runnable接口，并实现run()方法。Runnable 接口只有一个方法 run()，我们声明自己的类实现 Runnable 接口并提供这一方法，将线程代码写入其中，就完成了这一部分的任务。

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

并行执行实例

```
package thread;
class FileTransRunnable implements
      Runnable
{ private String fileName;
public FileTransRunnable(String fileName)
{ this.fileName = fileName; }
public void run(){
System.out.println("传送" + fileName);
try{
Thread.sleep(1000 * 10);
}catch(Exception ex){}
System.out.println(fileName + "传送完毕");
}
}
```

```
public class ThreadTest3 {
public static void main(String[] args) {
Thread ft1 = new Thread(new
      FileTransRunnable("文件1"));
Thread ft2 = new Thread(new
      FileTransRunnable("文件2"));
Thread ft3 = new Thread(new
      FileTransRunnable("文件3"));
ft1.start();
ft2.start();
ft3.start();
}
}
```

- 实现Runnable接口创建多线程

  1) 定义类，实现Runnable接口。

  2) 类中重写Runnable接口中的run方法。

  3) 通过Thread类含参构造函数创建线程对象，将Runnable接口的子类对象作为实际参数传递给Thread类的构造函数中。

  4) 调用Thread类的start方法：开启线程，调用Runnable子类接口的run方法。

- 使用继承Thread的方式来创建线程，比较容易理解，但由于JAVA语言的单根继承特点，使这种方式在应用时受到一定的限制。
- 通常我们会采用使用runnable接口创建多线程，这种方式的优点如下：

  1）对象可以自由地继承自另一个类。

  2）多个线程可以共享同一个接口实现类的对象，更适合多个线程共享数据情况。
- 两种方法都要重写run（）方法

3 thread类常用方法

- void start(): 启动线程，并执行对象的run()方法
- run(): 线程在被调度时执行的操作
- static Thread currentThread(): 返回当前线程。在Thread子类中就是this，通常用于主线程和Runnable实现类
- String getName(): 返回线程的名称
- void setName(String name):设置该线程名称

- static void yield()：线程让步

（1）暂停当前正在执行的线程，把执行机会让给优先级相同或更高的线程

（2）若队列中没有同优先级的线程，忽略此方法

- join():当线程A中调用线程B的join() 方法时，线程A将被阻塞，直到线程B执行完为止，线程A才结束阻塞

线程方法实例

```
class HelloThread extends Thread {
    public void run(){
        try{   for(int i=0;i<100;i++){
                Thread.currentThread().setName("thread 1");
                //Thread.currentThread().setPriority(10);
                if (i%2==0)
                //sleep(1000);
System.out.println(Thread.currentThread().getName() + ":"+i);
                }
        }catch(Exception ex){}
    }
}
```

```
public class ThreadTest4{
    public static void main(String[] args){
    HelloThread th1 = new HelloThread();
    th1.start();
    //Thread.currentThread().setPriority(1);
    for(int i=0;i<100;i++){
                Thread.currentThread().setName("thread 0");
                if (i%2==0)
System.out.println(Thread.currentThread().getName() + ":"+i);
                //if(i==50)
                //try{th1.join();}
                //catch (Exception ex){}  } }}
```

join方法示例

```
public class ThreadTest5{
private int sum = 0;
class CalThread extends Thread{//负责计算的线程
public void run(){
for(int i=1;i<=100000;i++){
sum += i;
} } }
class SaveThread extends Thread{//负责保存的线程
public void run(){
System.out.println("写入数据库:" + sum);
}
}
```

在运行主线程时，命令等待线程ct运行完毕，才能抢占CPU进行运行线程st。 在Java语言中，只需要调用线程ct的join()方法，就能够让系统等其运行完毕才能运行接下来的代码

```
public void work() throws Exception{
CalThread ct = new CalThread();
SaveThread st = new SaveThread();
ct.start();
ct.join();
st.start();
}
public static void main(String[] args)
   throws Exception {
new ThreadCooperateTest2().work();
}
}
```

- static void sleep(long millis)：(指定时间:毫秒)

（1）令当前活动线程在指定时间段内放弃对CPU控制,使其他线程有机会被执行,时间到后重排队。

（2）抛出InterruptedException异常

- stop(): 强制线程生命期结束，已过时
- boolean isAlive()：返回boolean，判断线程是否还活着

- 线程的优先级等级

MIN _PRIORITY：1

NORM_PRIORITY：5

MAX_PRIORITY：10

- **getPriority()：返回线程优先值**
- **setPriority(int newPriority)：改变线程的优先级**

习题

- 比较创建线程有哪两种方式(jdk5.0之前)。

【区别】

继承Thread: 线程代码存放Thread子类run方法中。

实现Runnable：线程代码存在实现接口的类中的run方法。

【实现方法的好处】

1）避免了单继承的局限性

2）多个线程可以共享同一个接口子类的对象，非常适合多个相同线程来处理同一份资源。

习题

- JDK5.0 新增线程方式1：实现Callable接口
  - 相比run()方法，可以有返回值；方法可以抛出异常

- 新增线程方式2：使用线程池ThreadPoolExecutor
  - 经常创建和销毁、使用量特别大的资源，比如并发情况下的线程，对性能影响很大。
  - 提前创建好多个线程，放入线程池中，使用时直接获取，使用完放回池中。可以避免频繁创建销毁、实现重复利用。类似生活中的公共交通工具。

- Runnable接口包括哪些抽象方法？Thread类有哪些主要域和方法？

答：Runnable接口中仅有run()抽象方法。

Thread类主要域有：MAX_PRIORITY，MIN_PRIORITY，NORM_PRIORITY。

主要方法有start()，run()，sleep()，currentThread()，setPriority()，getPriority(),join()等。

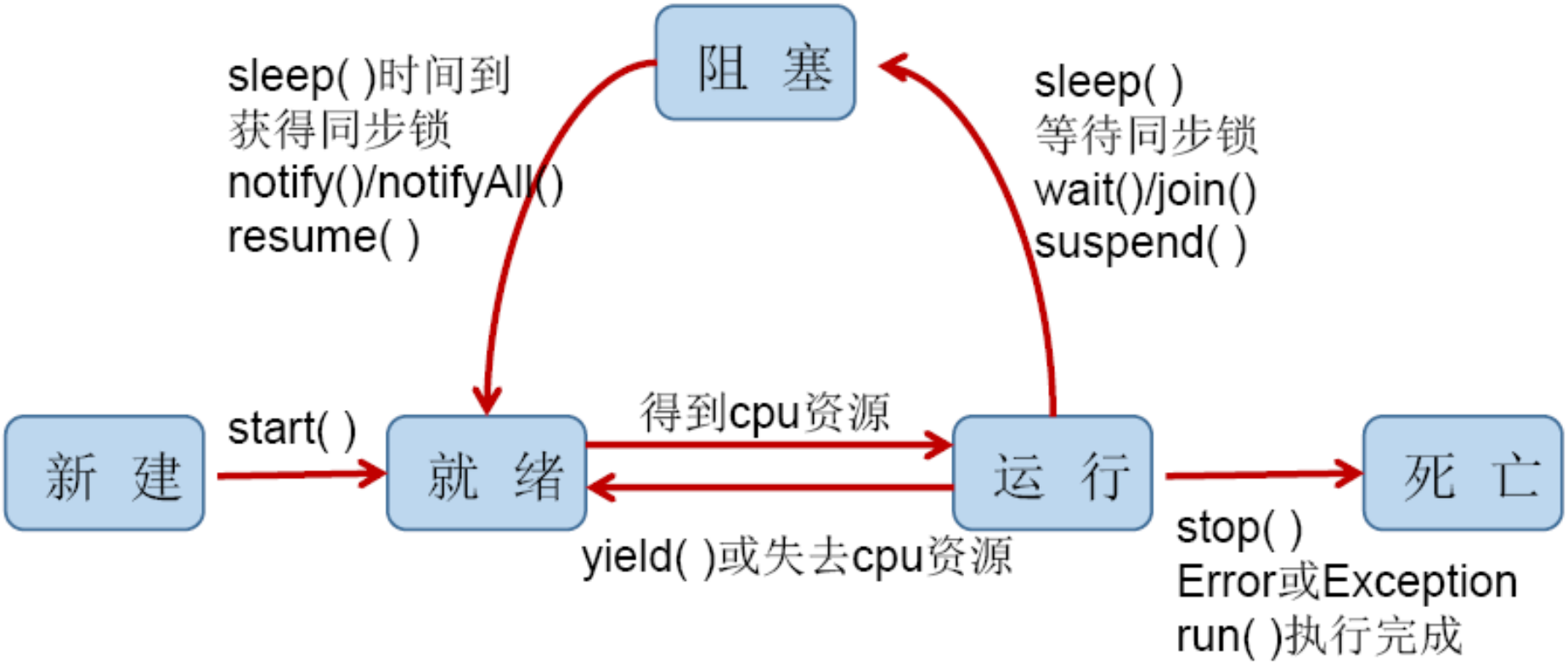
习题

- Thread.start ()与 Thread.run ()有什么区别?

  - Thread.start ()方法启动线程，使之进入就绪状态，这并不意味着线程就会立即运行。
  - 当 cpu 分配时间该线程时，由 JVM 调度执行 run ()方法。

4 线程生命周期

- 线程从创建、启动到终止的整个过程，叫一个生命周期。在其间的任何一个时刻，线程总是处于某个特定的状态。



- 1）新建状态（New）：即创建了一个线程对象后，还没有在其上调用start()方法。产生的新线程进入新建状态。
- 如：`MyThreadClass  myThread = new MyThreadClass();`

- 2）就绪状态（Runnable）：线程对象创建后，其他线程调用了该对象的start()方法。处于该状态的线程位于可运行线程池中，成为可运行，等待获取CPU的使用权。线程有资格运行，但调度程序还没有把它选定为运行线程时线程所处的状态。当start()方法调用时，线程首先进入可运行状态。在线程运行之后或者从阻塞、等待或睡眠状态回来后，也返回到可运行状态。

- 3）运行状态（Running）：线程调度程序从可运行池中选择一个线程作为当前线程时线程所处的状态，这也是线程进入运行状态的唯一一种方式。就绪状态的线程获取了CPU，执行程序代码。

- 4）阻塞状态（Blocked）：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。阻塞的情况分三种：

（1）等待阻塞：运行的线程执行wait()方法，该线程放入等待池中。

（2）同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，该线程放入锁池中。

（3）其他阻塞：运行的线程执行sleep()或join()方法，该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时时，线程重新转入就绪状态。

- 5）死亡状态（Dead）：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。这个线程对象也许还存在，但是，它已经不是一个单独执行的线程。线程一旦死亡，就不能复生。 如果在一个死去的线程上调用start()方法，会抛出java.lang.IllegalThreadStateException异常。

简单的说：

1. 创建状态：使用new运算符创建一个线程。
2. 可运行状态：使用start()方法启动一个线程后，系统分配了资源。
3. 运行中状态：执行线程的run()方法。
4. 阻塞状态：运行的线程因某种原因停止继续运行。
5. 死亡状态：线程结束。

习题

1.Java 语言中提供了一个___线程，自动回收动态分配的内存。

A 异步

B 消费者

C 守护

D 垃圾收集

D

2.当___方法终止时，能使线程进入死亡状态。

A run

B setPrority

C yield

D sleep

A

习题

3.用___方法可以改变线程的优先级。 B

A run

B setPrority

C yield

D sleep

4.线程通过___方法可以使具有相同优先级线程获得处理器。C

A run

B setPrority

C yield

D sleep

习题

5.线程通过___方法可以休眠一段时间，然后恢复运行。**D**

A run

B setPrority

C yield

D sleep

6. ___方法可以用来暂时停止当前线程的运行。 **BCD**

A stop( )

B sleep( )

C wait( )

D join()

- 线程的基本概念、线程的基本状态以及状态之间的关系

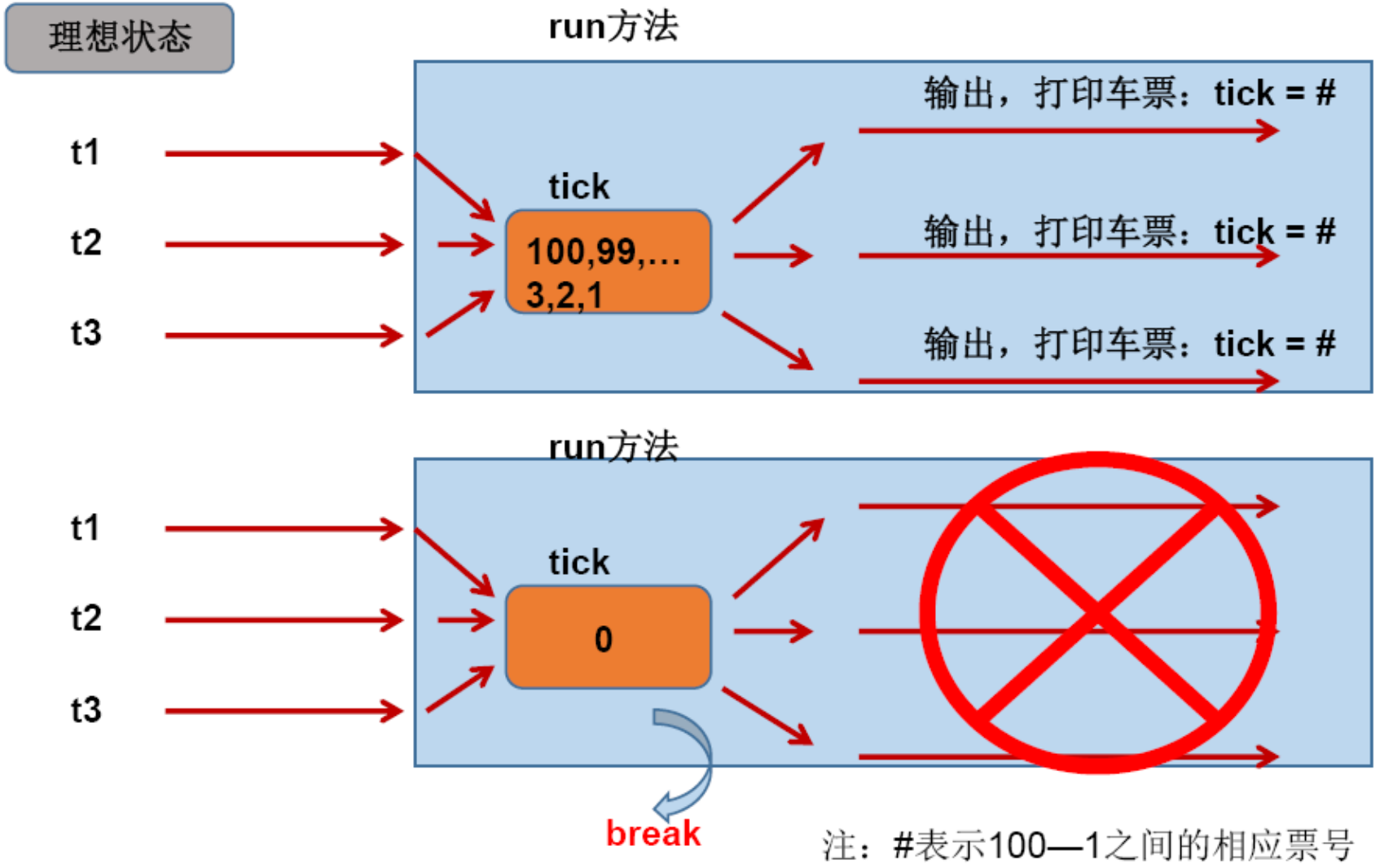
答：线程指在程序执行过程中，能够执行程序代码的一个执行单位，每个程序至少都有一个线程，也就是程序本身。

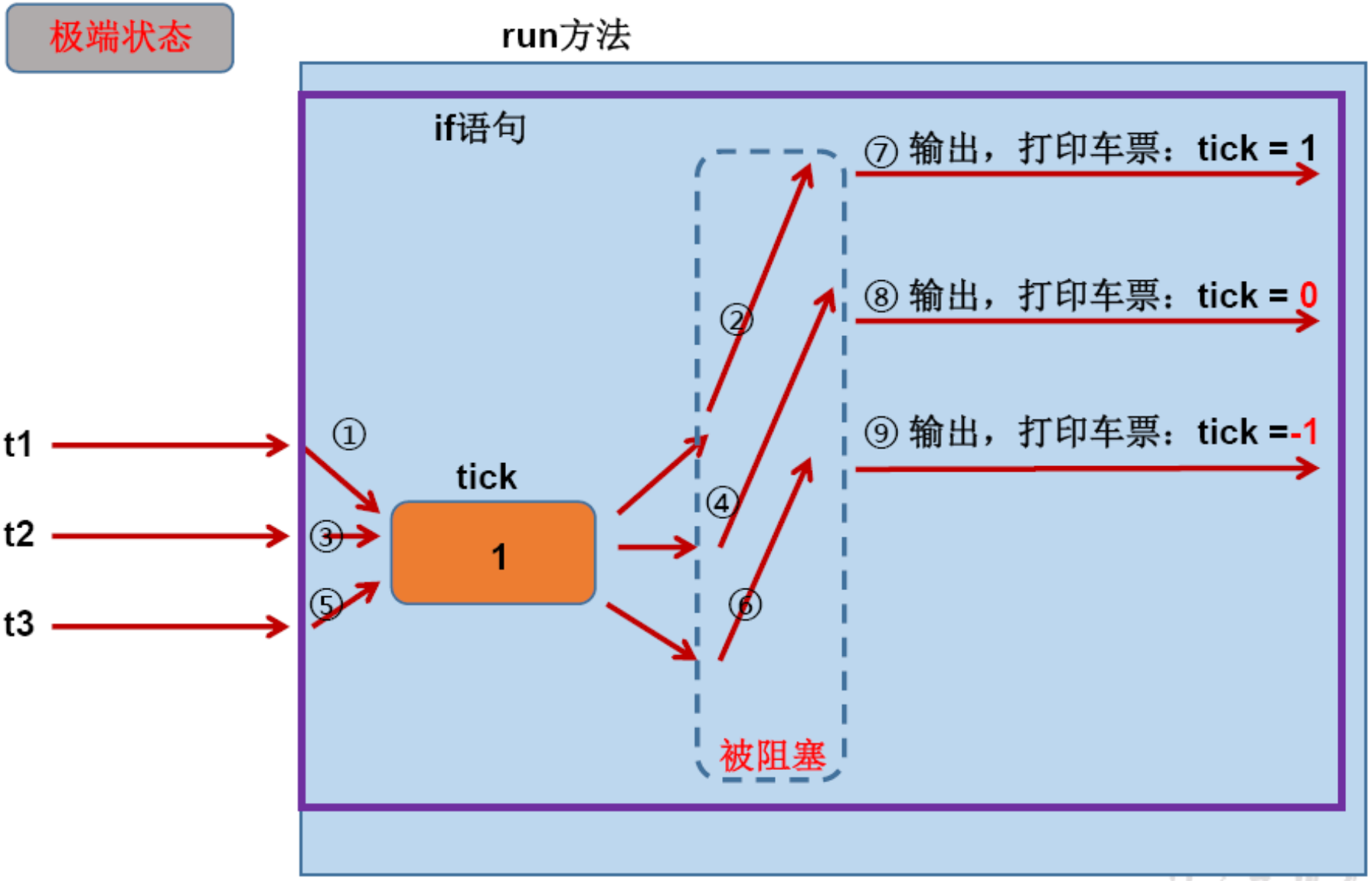
Java中的线程有5种状态分别是：创建、就绪、运行、阻塞、结束

5. 多线程的同步

- 在多线程的程序中，多个线程可能会对同一个资源并发访问。这种情况下，如果不对共享的资源进行保护，就可能产生问题。

经典的多用户买票问题


理想状态
run方法
t1
t2
t3
tick
100,99,…
3,2,1
输出，打印车票：tick = #
输出，打印车票：tick = #
输出，打印车票：tick = #
run方法
t1
t2
t3
tick
0
break
注：#表示100—1之间的相应票号


极端状态
run方法
if语句
⑦ 输出，打印车票：tick = 1
⑧ 输出，打印车票：tick = 0
⑨ 输出，打印车票：tick =-1
t1
t2
t3
①
②
③
④
⑤
⑥
tick
1
被阻塞

观察程序中的代码行1处的注释，当只剩下一张票时，线程1卖出了最后一张票，接着要运行 ticketNum--，但在 ticketNum--还没来得及运行的时候，线程2有可能抢占 CPU，来判断当前有无票可卖，此时，由于线程1还没有运行 ticketNum--，当然票数还是1，线程2判断还可以买票，这样，最后一张票卖出了两次。

```
class TicketRunnable implements Runnable{
  private int ticketNum = 3;  //以3 张票为例
  public void run(){
    while(true){
    String tName = Thread.currentThread().getName();
    if(ticketNum<=0){System.out.println(tName +  " no ticket");
    break; }
       else{
       try{Thread.sleep(100);
       ticketNum--; //代码行 1
       System.out.println(tName+" bought one ticket, left "+ticketNum
    +" ticket"); }
       catch (Exception ex){}
             }
    }
  }}
```

```
public class ThreadSynTest1 {
  public static void main(String[] args){
  TicketRunnable tr = new TicketRunnable();
  Thread th1 = new Thread(tr, "thread 1");
  Thread th2 = new Thread(tr, "thread 2");
  th1.start();
  th2.start();
 }
}
```

以上案例是多个线程消费有限资源的情况，该情况下还有很多其他案例，如：多个线程，向有限空间写数据时：线程1写完数据，空间满了，但没来得及告诉系统；此时另一个线程抢占CPU，也来写，不知道空间已满，造成溢出。

- 问题的原因：当多条语句在操作同一个线程共享数据时，一个线程对多条语句只执行了一部分，还没有执行完，另一个线程参与进来执行。导致共享数据的错误。
- 怎样解决这个问题？很简单，就是让一个线程卖票时其他线程不能抢占 CPU。
- 通俗地讲，可以给共享资源（在本例中为票）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。

- 解决思路：一个线程是否能够抢占CPU，必须考虑另一个线程中的某种条件，而不能随便让操作系统按照默认方式分配CPU，如果条件不具备，就应该等待另一个线程运行，直到条件具备。

- 对于多线程之间资源争用的安全问题提供了专业的解决方式：同步机制
- 所谓同步(synchronize)，就是在发出一个功能调用时，在没有得到结果之前，该调用就不返回，同时其它线程也不能调用这个方法。

- Java中每个对象都有一个内置锁，当程序运行到非静态的synchronized同步方法上时，自动获得与正在执行代码类的当前实例（this实例）有关的锁。获得一个对象的锁也称为获取锁、锁定对象、在对象上锁定或在对象上同步。
- 一个对象只有一个锁。所以，如果一个线程获得该锁，就没有其他线程可以获得锁，直到第一个线程释放（或返回）锁。这也意味着任何其他线程都不能进入该对象上的synchronized方法或代码块，直到该锁被释放。释放锁是指持锁线程退出了synchronized同步方法或代码块。

在Java中通过互斥锁标志Synchronized关键字的运用来实现同步。Java中同步有两种方法:

（1）同步方法

```
synchronized void method( ) {
//同步的方法}
```

a)实现方法：在要标志为同步的方法前加上synchronized关键字。如：public synchronized void call(String msg){ }

b)实现原理：当调用对象的同步方法时，线程取得对象锁（lock）或监视器；如果另一个线程试图执行任何同步方法时，他就会发现他被锁住了，进入挂起状态，直到对象监视器上的锁被释放时为止。当锁住方法的线程从方法中返回时，只有一个排队等候的线程可以访问对象。

c）锁的作用域：该方法被执行的整个时间。

（2）同步代码块

```
synchronized(object) {
 //要同步的语句
 }
```

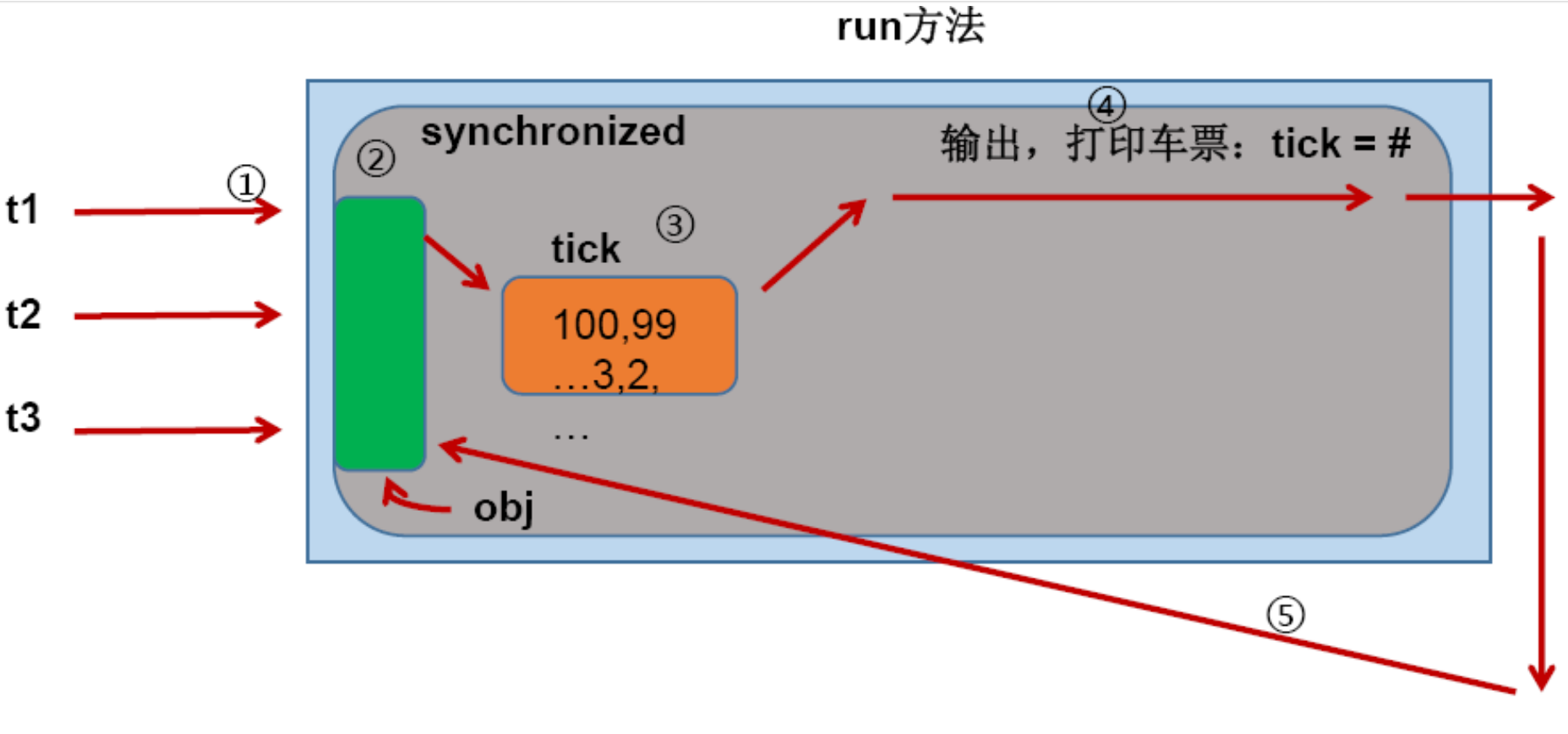
a)临界区：只希望防止多个线程同时访问方法内部的部分代码，而不是防止访问整个方法，通过这种方式分离出来的代码段被称为“临界区”，即需要进行互斥的代码段。

b)实现方法：用synchronized来指定某个对象，此对象的锁被用来对花括号内的代码进行同步控制。如：
synchronized(target){ target.call(msg); }

c)实现原理：在进入同步代码前，必须得到object对象的锁，如果其他线程已经得到这个锁，那么就得等到锁被释放后才能进入临界区。

d)锁的作用域：只在代码块运行的时间内。

```
class TicketRunnable implements Runnable {
  private int ticketNum = 3; //  以3 张票为例
  public void run() {
  while (true) {
   String tName = Thread.currentThread().getName();
   //  将需要独占 CPU的代码用 synchronized(this)包围起来
    synchronized (this) {
    if (ticketNum <= 0) { System.out.println(tName + “无票”);
     break;
    } else {
     try { Thread.sleep(100);//  程序休眠 100 毫秒
           ticketNum--; // 代码行1
System.out.println(tName + “卖出一张票,还剩”  +ticketNum+“张票”);
         } catch (Exception ex) { }
      }
    }
   }
  }
}
```


run方法
synchronized
②
①
t1
t2
t3
tick
③
100,99
…3,2,
…
obj
④
输出，打印车票：tick = #
⑤

- 从以上代码可以看出，该方法的本质是将需要独占 CPU 的代码用synchronized(this)包围起来。如前所述，一个线程进入这段代码之后，就在 this 上加了一个标记，直到该线程将这段代码运行完毕，才释放这个标记。如果其他线程想要抢占 CPU，先要检查 this 上是否有这个标记。若有，就必须等待。

synchronized的锁是什么？

- 任意对象都可以作为同步锁。所有对象都自动含有单一的锁（监视器）。
- 同步的锁：静态方法（类名.class）、非静态方法（this）

注意：

- 必须确保使用同一个资源的多个线程共用一把锁，这个非常重要，否则就无法保证共享资源的安全（上例中共享同一对象tr）
- 一个线程类中的所有静态方法共用同一把锁（类名.class），所有非静态方法共用同一把锁（this），同步代码块（指定需谨慎）

- 1、如何找问题，即代码是否存在线程安全?

（1）明确哪些代码是多线程运行的代码

（2）明确多个线程是否有共享数据

（3）明确多线程运行代码中是否有多条语句操作共享数据

- 如何解决呢?

对多条操作共享数据的语句，只能让一个线程都执行完，在执行过程中，其他线程不可以参与执行。所有操作共享数据的这些语句都要放在同步范围中

- 但是可以看出，该代码实际上运行较慢，因为一个线程的运行，必须等待另一个线程将同步代码段运行完毕。因此，从性能上讲，线程同步是非常耗费资源的一种操作。
- 我们要尽量控制线程同步的代码段范围，理论上说，同步的代码段范围越小，段数越少越好，因此在某些情况下，推荐将小的同步代码段合并为大的同步代码段。

Lock锁实现同步机制

- 从JDK 5.0开始，Java提供了更强大的线程同步机制——通过显式定义同步锁对象来实现同步。同步锁使用Lock对象充当。
- java.util.concurrent.locks.Lock接口是控制多个线程对共享资源进行访问的工具。锁提供了对共享资源的独占访问，每次只能有一个线程对Lock对象锁，线程开始访问共享资源之前应先获得Lock对象。

- Lock锁实现同步机制
  - ReentrantLock 类实现了Lock ，它拥有与 synchronized 相同的并发性和内存语义，在实现线程安全的控制中，比较常用的是ReentrantLock，可以显式加锁、释放锁。

- **synchronized 与Lock 的对比**
  - 1. Lock是显式锁（手动开启和关闭锁，别忘记关闭锁），synchronized是隐式锁，出了作用域自动释放
  - 2.Lock只有代码块锁，synchronized有代码块锁和方法锁
  - 3.使用Lock锁，JVM将花费较少的时间来调度线程，性能更好。并且具有更好的扩展性（提供更多的子类）
- **优先使用顺序：**
  - Lock>同步代码块（已经进入了方法体，分配了相应资源）>同步方法（在方法体之外）

6.死锁

如果不当地使用代码段的同步，会出现什么情况呢?

- 如果出现一种极端情况，一个线程等候另一个对象，而另一个对象又在等候下一个对象，以此类推。这个“等候链”如果进入封闭状态，也就是说，最后那个对象等候的是第一个对象。
- 此时，所有线程都会陷入无休止的相互等待状态，造成死锁。尽管这种情况并非经常出现，但一旦碰到，程序的调试将变得异常艰难。

```
public class DeadLockTest implements Runnable {
  static Object S1 = new Object(), S2 = new Object();
public void run() {
  if (Thread.currentThread().getName().equals("th1")) {
   synchronized (S1) {
    System.out.println("线程 1锁定 S1 "); // 代码段 1
    synchronized (S2) {
     System.out.println("线程 1锁定 S2 "); // 代码段 2
    }    }
  } else {
   synchronized (S2) {
    System.out.println("线程 2锁定 S2 "); // 代码段 3
    synchronized (S1) {
     System.out.println("线程 2锁定 S1 "); // 代码段 4
    }  }  } }
```

```
public static void main(String[] args) {
  Thread t1 = new Thread(new DeadLockTest(), "th1");
  Thread t2 = new Thread(new DeadLockTest(), "th2");
  t1.start();
  t2.start();
 }
}
```

两个线程陷入无休止的等待。观察 run()函数中的代码，当 th1运行后，进入代码段 1，锁定了 S1，如果此时 th2 运行，抢占 CPU，进入代码段 3，锁定 S2，那么 th1就无法运行代码段 2，但是又没有释放 S1，此时，th2 也就不能运行代码段 4。造成互相等待。

- 发生死锁必须同时满足的四个条件：

（1）互斥条件。线程中使用的资源中至少要有一个是不能共享的。

（2）至少有一个线程它必须持有一个资源且正在等待获取一个当前被别的线程持有的资源。

（3）资源不能被线程抢占。

（4）必须有循环等待，这时，一个线程等待其他线程所持有的资源，后者又在等待另一个线程所持有的资源。

- 如何避免死锁呢？就语言本身来说，尚未直接提供防止死锁的帮助措施，需要谨慎的设计来避免。一般情况下，我们主要是针对死锁产生的四个必要条件来进行破坏，用以避免和预防死锁。在系统设计、线程开发等方面，注意如何不让这四个必要条件成立，如何确定资源的合理分配算法，避免线程永久占据系统资源。
- 避免死锁具体方法：专门的算法、原则；尽量减少同步资源的定义；尽量避免嵌套同步

- 解决死锁没有简单的方法，这是因为线程产生死锁都各有各的原因，而且往往具有很高的负载。从技术上讲，可以用如下方法来进行死锁排除：

1. 可以撤消陷于死锁的全部线程。
2. 可以逐个撤消陷于死锁的进程，直到死锁不存在。
3. 从陷于死锁的线程中逐个强迫放弃所占用的资源，直至死锁消失。

- 解决死锁没有简单的方法，这是因为线程产生死锁都各有各的原因，而且往往具有很高的负载。从技术上讲，可以用如下方法来进行死锁排除：

1. 可以撤消陷于死锁的全部线程。
2. 可以逐个撤消陷于死锁的进程，直到死锁不存在。
3. 从陷于死锁的线程中逐个强迫放弃所占用的资源，直至死锁消失。

控制线程实例

```
public class ThreadControlTest1 implements Runnable{
private int percent = 0;
public void run(){
while(true){
System.out.println("传输进度:" + percent + "%");
try{
Thread.sleep(1000);
}catch(Exception ex){}
percent += 10;
if(percent==100){
System.out.println("传输完毕");
break;
}
}
}
```

```
public static void main(String[] args) throws Exception {
ThreadControlTest1 ft = new ThreadControlTest1();
Thread th = new Thread(ft);
th.start();
}
}
```

7. 线程的调度

- 两个线程交替打印的例子

```
class Number implements Runnable{
   private int number =1;//Object obj = new Object();
   public void run(){
      while(true){
        synchronized(this){//obj
            notify();
            if (number<100)
{System.out.println(Thread.currentThread().getName()+":"+number);
                  number++;
                  try{wait();}
                  catch(Exception ex){}
            }else{break;} }}}}
```

两个线程交替打印的例子

```
public class ComTest{
    public static void main(String[] args){
        Number number = new Number();
        Thread t1= new Thread(number);
        Thread t2= new Thread(number);
        t1.setName("Thread 1");
        t2.setName("Thread 2");
        t1.start();
        t2.start();
    }
}
```

- **wait()**：令当前线程挂起并放弃CPU、同步资源并等待，使别的线程可访问并修改共享资源，而当前线程排队等候其他线程调用**notify()**或**notifyAll()**方法唤醒，唤醒后等待重新获得对监视器的所有权后才能继续执行
- **notify()**：唤醒正在排队等待同步资源的线程中优先级最高者结束等待
- **notifyAll()**：唤醒正在排队等待资源的所有线程结束等待

- 这三个方法只有针对同一对象的synchronized方法或synchronized代码块中才能使用，否则会报java.lang.IllegalMonitorStateException异常。
- 因为这三个方法必须有锁对象调用，而任意对象都可以作为synchronized的同步锁，因此这三个方法只能在Object类中声明。

wait() 方法

- 在当前线程中调用方法：对象名.wait()
- 使当前线程进入等待（某对象）状态,直到另一线程对该对象发出notify (或notifyAll) 为止。
- 调用方法的必要条件：当前线程必须具有对该对象的监控权（加锁）
- 调用此方法后，当前线程将释放对象监控权，然后进入等待
- 在当前线程被notify后，要重新获得监控权，然后从断点处继续代码的执行。

- **notify()/notifyAll()**
  - 在当前线程中调用方法：对象名.notify(), 对象名.notifyAll()
  - 功能：唤醒等待该对象监控权的一个/所有线程。
  - 调用方法的必要条件：当前线程必须具有对该对象的监控权（加锁）

wait() 方法和sleep()方法的异同

- 相同：wait() 方法和sleep()方法都可以当前线程进入阻塞状态。
- 不同：

（1）两个方法声明位置不同，Thread类声明sleep()，Object类声明wait()；

（2）调用要求不同，sleep()可以在任何场景调用，wait()必须在同步代码块或方法中调用

（3）是否释放同步监视器，如果两个方法都在同步代码块或方法中使用，sleep()不会释放锁，而wait()会释放锁。

习题

1.C 和 Java 都是多线程语言。 错

2.如果线程死亡，它便不能运行。 对

3.在 Java 中，高优先级的可运行线程会抢占低优先级线程。 对

4.程序开发者必须创建一个线程去管理内存的分配。 错

5.一个线程在调用它的 start 方法之前，该线程将一直处于出生期。 对

6.当调用一个正在进行线程的 stop()方法时，该线程便会进入休眠状态。 错，sleep方法

7.如果线程的 run 方法执行结束或抛出一个不能捕获的例外，线程便进入等待状态。 错，dead状态

- 多线程有几种实现方法,都是什么?同步有几种实现方法,都是什么?

答：多线程有两种实现方法，分别是继承Thread类与实现Runnable接口

同步的实现方面有两种，分别是synchronized, wait与notify

```
public class Test {
    public static void main(String[] args) {
        MyThread m = new MyThread();
        Thread t = new Thread(m);
        t.start();
                {1}
        int j = m.i;
        System.out.println(j);
    }
}
class MyThread implements Runnable{
    int i;
    public void run(){
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        i=100;
    }
}
```

在{1}添加什么代码，可以保证如下代码输出100

t.join();