

day15【多线程】

今日内容

- 多线程

教学目标

- ☐ 说出进程和线程的概念
- ☐ 能够理解并发与并行的区别
- ☐ 能够描述Java中多线程运行原理
- ☐ 能够使用继承类的方式创建多线程
- ☐ 能够使用实现接口的方式创建多线程
- ☐ 能够说出实现接口方式的好处
- ☐ 能够解释安全问题的出现的原因

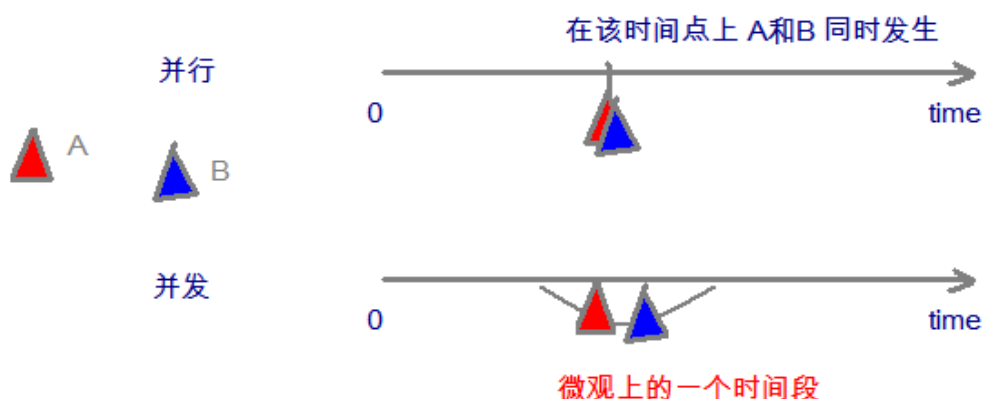
第一章 多线程

我们在之前，学习的程序在没有跳转语句的前提下，都是由上至下依次执行，那现在想要设计一个程序，边打游戏边听歌，怎么设计？

要解决上述问题,咱们得使用多进程或者多线程来解决。

1.1 并发与并行

- **并行**：指两个或多个事件在**同一时刻**发生（同时执行）。
- **并发**：指两个或多个事件在**同一个时间段内**发生(交替执行)。



在操作系统中，安装了多个程序，并发指的是在一段时间内宏观上有多个程序同时运行，这在单 CPU 系统中，每一时刻只能有一道程序执行，即微观上这些程序是分时的交替运行，只不过是给人的感觉是同时运行，那是因为分时交替运行的时间是非常短的。

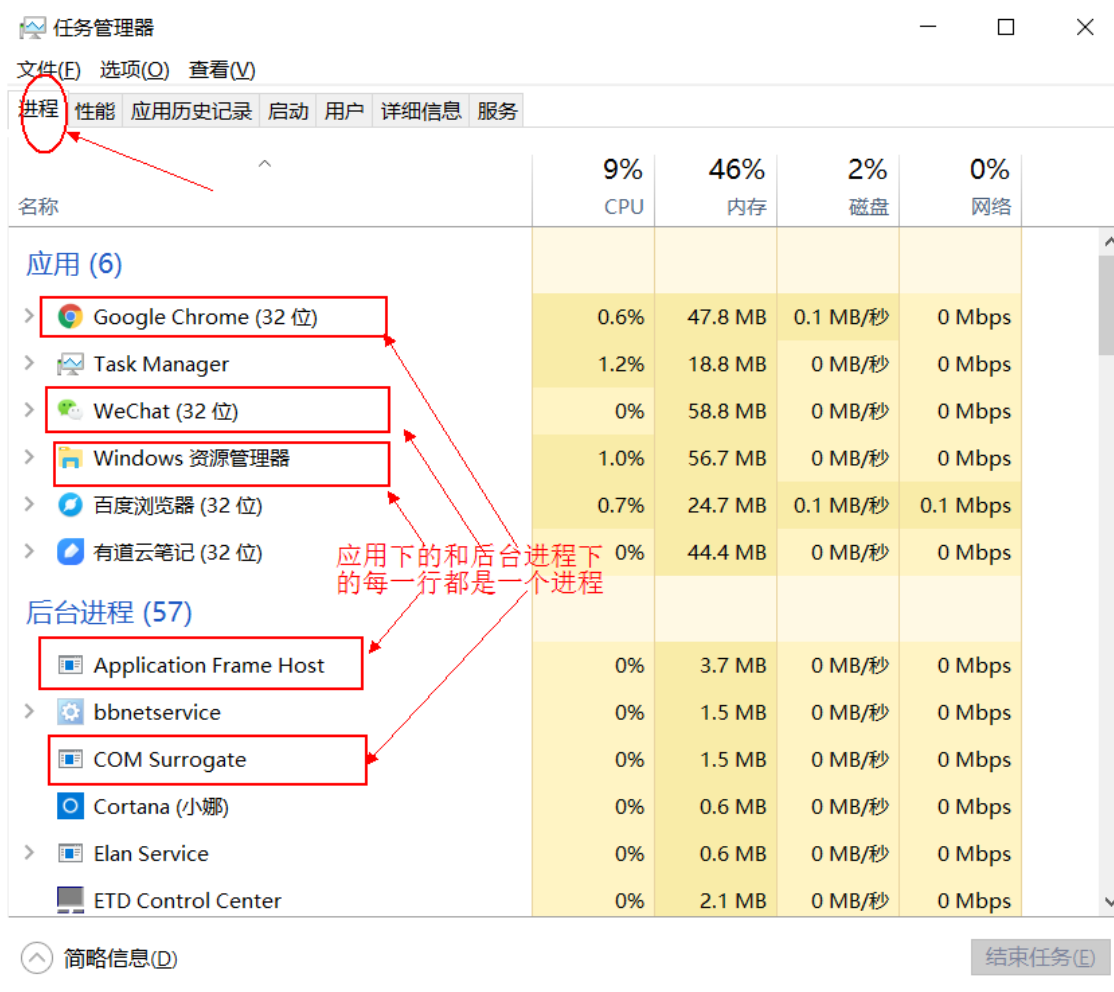
而在多个 CPU 系统中，则这些可以并发执行的程序便可以分配到多个处理器上（CPU），实现多任务并行执行，即利用每个处理器来处理一个可以并发执行的程序，这样多个程序便可以同时执行。目前电脑市场上说的多核 CPU，便是多核处理器，核越多，并行处理的程序越多，能大大的提高电脑运行的效率。

注意：单核处理器的计算机肯定是不能并行的处理多个任务的，只能是多个任务在单个CPU上并发运行。同理,线程也是一样的，从宏观角度上理解线程是并行运行的，但是从微观角度上分析却是串行运行的，即一个线程一个线程的去运行，当系统只有一个CPU时，线程会以某种顺序执行多个线程，我们把这种情况称之为线程调度。

1.2 线程与进程

- **进程**：是指一个内存中运行的应用程序，每个进程都有一个独立的内存空间，一个应用程序可以同时运行多个进程；进程也是程序的一次执行过程，是系统运行程序的基本单位；系统运行一个程序即是一个进程从创建、运行到消亡的过程。
- **线程**：是进程中的一个执行单元，负责当前进程中程序的执行，一个进程中至少有一个线程。一个进程中是可以有多个线程的，这个应用程序也可以称之为多线程程序。

进程



任务管理器

文件(F) 选项(O) 查看(V)

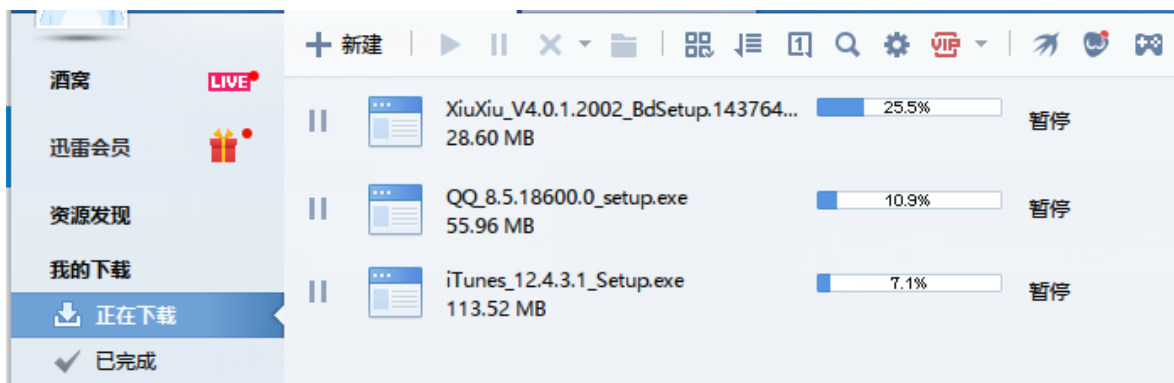
进程 性能 应用历史记录 启动 用户 详细信息 服务

名称	9% CPU	46% 内存	2% 磁盘	0% 网络
应用 (6)				
Google Chrome (32 位)	0.6%	47.8 MB	0.1 MB/秒	0 Mbps
Task Manager	1.2%	18.8 MB	0 MB/秒	0 Mbps
WeChat (32 位)	0%	58.8 MB	0 MB/秒	0 Mbps
Windows 资源管理器	1.0%	56.7 MB	0 MB/秒	0 Mbps
百度浏览器 (32 位)	0.7%	24.7 MB	0.1 MB/秒	0.1 Mbps
有道云笔记 (32 位)	0%	44.4 MB	0 MB/秒	0 Mbps
后台进程 (57)				
Application Frame Host	0%	3.7 MB	0 MB/秒	0 Mbps
bbnet service	0%	1.5 MB	0 MB/秒	0 Mbps
COM Surrogate	0%	1.5 MB	0 MB/秒	0 Mbps
Cortana (小娜)	0%	0.6 MB	0 MB/秒	0 Mbps
Elan Service	0%	0.6 MB	0 MB/秒	0 Mbps
ETD Control Center	0%	2.1 MB	0 MB/秒	0 Mbps

应用下的和后台进程下的每一行都是一个进程

简略信息(D) 结束任务(E)

线程



进程与线程的区别

- 进程：有独立的内存空间，进程中的数据存放空间（堆空间和栈空间）是独立的，至少有一个线程。
- 线程：堆空间是共享的，栈空间是独立的，线程消耗的资源比进程小的多。

注意：下面内容为了解知识点

1:因为一个进程中的多个线程是并发运行的，那么从微观角度看也是有先后顺序的，哪个线程执行完全取决于 CPU 的调度，程序员是干涉不了的。而这就造成的多线程的随机性。

2:Java 程序的进程里面至少包含两个线程，主进程也就是 main()方法线程，另外一个垃圾回收机制线程。每当使用 java 命令执行一个类时，实际上都会启动一个 JVM，每一个 JVM 实际上就是在操作系统中启动了一个线程，java 本身具备了垃圾的收集机制，所以在 Java 运行时至少会启动两个线程。

3:由于创建一个线程的开销比创建一个进程的开销小的多，那么我们在开发多任务运行的时候，通常考虑创建多线程，而不是创建多进程。

线程调度:

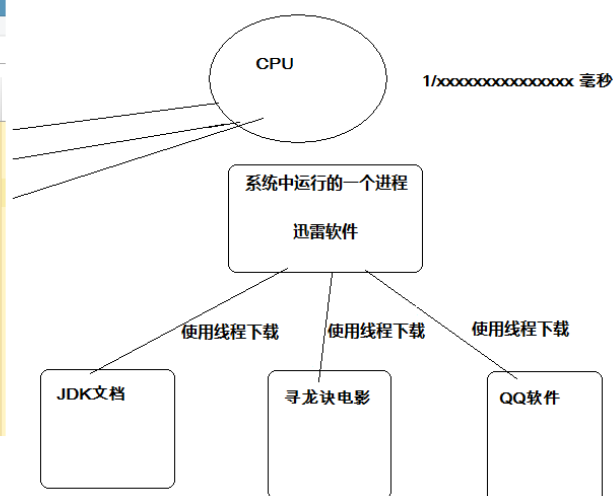
- 分时调度

所有线程轮流使用 CPU 的使用权，平均分配每个线程占用 CPU 的时间。

- 抢占式调度

优先让优先级高的线程使用 CPU，如果线程的优先级相同，那么会随机选择一个(线程随机性)，Java使用的为抢占式调度。

任务管理器				
进程				
名称	状态	CPU	内存	
腾讯QQ (32 位)		0%	109.7 MB	
Windows 资源管理器		0.4%	69.9 MB	
迅雷7 (32 位)		0.6%	44.2 MB	
Microsoft Word (32 位)		0%	36.3 MB	
ThunderPlatform应用程序 (32...		0.2%	35.2 MB	
Microsoft Windows Search ...		0%	26.7 MB	
mysqld.exe (32 位)		0%	25.2 MB	
XBrowser (32 位)		0.7%	22.5 MB	
画图 (32 位)		0%	18.7 MB	
Foxmail 7.0 (32 位)		0%	18.4 MB	
360安全卫士 安全防护中心模块...		0.2%	18.0 MB	



1.3 Thread类

线程开启我们需要用到了 `java.lang.Thread` 类，API中该类中定义了有关线程的一些方法，具体如下：

构造方法：

- `public Thread()` :分配一个新的线程对象。
- `public Thread(String name)` :分配一个指定名字的新的线程对象。
- `public Thread(Runnable target)` :分配一个带有指定目标新的线程对象。
- `public Thread(Runnable target, String name)` :分配一个带有指定目标新的线程对象并指定名字。

常用方法：

- `public String getName()` :获取当前线程名称。
- `public void start()` :导致此线程开始执行; Java虚拟机调用此线程的run方法。
- `public void run()` :此线程要执行的任务在此处定义代码。
- `public static void sleep(long millis)` :使当前正在执行的线程以指定的毫秒数暂停（暂时停止执行）。
- `public static Thread currentThread()` :返回对当前正在执行的线程对象的引用。

翻阅API后得知创建线程的方式总共有两种，一种是继承Thread类方式，一种是实现Runnable接口方式，方式一我们上一天已经完成，接下来讲解方式二实现的方式。

1.4 创建线程方式一_继承方式

Java使用 `java.lang.Thread` 类代表**线程**，所有的线程对象都必须是Thread类或其子类的实例。每个线程的作用是完成一定的任务，实际上就是执行一段程序流即一段顺序执行的代码。Java使用线程执行体来代表这段程序流。Java中通过继承Thread类来**创建并启动多线程**的步骤如下：

1. 定义Thread类的子类，并重写该类的run()方法，该run()方法的方法体就代表了线程需要完成的任务,因此把run()方法称为线程执行体。
2. 创建Thread子类的实例，即创建了线程对象
3. 调用线程对象的start()方法来启动该线程

代码如下：

测试类：

```
public class Demo01 {
    public static void main(String[] args) {
        //创建自定义线程对象
        MyThread mt = new MyThread("新的线程!");
        //开启新线程
        mt.start();
        //在主方法中执行for循环
        for (int i = 0; i < 200; i++) {
            System.out.println("main线程! "+i);
        }
    }
}
```

自定义线程类：

```
public class MyThread extends Thread {
    //定义指定线程名称的构造方法
    public MyThread(String name) {
        //调用父类的String参数的构造方法，指定线程的名称
    }
}
```

```

        super(name);
    }
    public MyThread() {
        //不指定线程的名字,线程有默认的名字Thread-0
    }
    /**
     * 重写run方法,完成该线程执行的逻辑
     */
    @Override
    public void run() {
        for (int i = 0; i < 200; i++) {
            System.out.println(getName()+"：正在执行！ "+i);
        }
    }
}

```

1.5 创建线程的方式二_实现方式

采用 `java.lang.Runnable` 也是非常常见的一种,我们只需要重写run方法即可。

步骤如下:

1. 定义Runnable接口的实现类,并重写该接口的run()方法,该run()方法的方法体同样是该线程的线程执行体。
2. 创建Runnable实现类的实例,并以此实例作为Thread的target来创建Thread对象,该Thread对象才是真正的线程对象。
3. 调用线程对象的start()方法来启动线程。

代码如下:

```

public class MyRunnable implements Runnable{
    @Override
    public void run() {
        for (int i = 0; i < 20; i++) {
            System.out.println(Thread.currentThread().getName()+" "+i);
        }
    }
}

```

```

public class Demo {
    public static void main(String[] args) {
        //创建自定义类对象 线程任务对象
        MyRunnable mr = new MyRunnable();
        //创建线程对象
        Thread t = new Thread(mr, "小强");
        t.start();
        for (int i = 0; i < 20; i++) {
            System.out.println("旺财 " + i);
        }
    }
}

```

通过实现Runnable接口,使得该类有了多线程类的特征。run()方法是多线程程序的一个执行目标。所有的多线程代码都在run方法里面。Thread类实际上也是实现了Runnable接口的类。

在启动的多线程的时候，需要先通过Thread类的构造方法Thread(Runnable target) 构造出对象，然后调用Thread对象的start()方法来运行多线程代码。

实际上所有的多线程代码都是通过运行Thread的start()方法来运行的。因此，不管是继承Thread类还是实现Runnable接口来实现多线程，最终还是通过Thread的对象的API来控制线程的，熟悉Thread类的API是进行多线程编程的基础。

tips:Runnable对象仅作为Thread对象的target，Runnable实现类里包含的run()方法仅作为线程执行体。而实际的线程对象依然是Thread实例，只是该Thread线程负责执行其target的run()方法。

Thread和Runnable的区别

如果一个类继承Thread，则不适合资源共享。但是如果实现了Runnable接口的话，则很容易的实现资源共享。

总结：

实现Runnable接口比继承Thread类所具有的优势：

1. 适合多个相同的程序代码的线程去共享同一个资源。
2. 可以避免java中的单继承的局限性。
3. 增加程序的健壮性，实现解耦操作，代码可以被多个线程共享，代码和线程独立。
4. 线程池只能放入实现Runnable或Callable类线程，不能直接放入继承Thread的类。

1.6 匿名内部类方式

使用线程的内匿名内部类方式，可以方便的实现每个线程执行不同的线程任务操作。

使用匿名内部类的方式实现Runnable接口，重新Runnable接口中的run方法：

```
public class NoNameInnerClassThread {
    public static void main(String[] args) {
        //      new Runnable(){
        //          public void run(){
        //              for (int i = 0; i < 20; i++) {
        //                  System.out.println("张宇:"+i);
        //              }
        //          }
        //      }; //---这个整体 相当于new MyRunnable()
        Runnable r = new Runnable(){
            public void run(){
                for (int i = 0; i < 20; i++) {
                    System.out.println("张宇:"+i);
                }
            }
        };
        new Thread(r).start();

        for (int i = 0; i < 20; i++) {
            System.out.println("费玉清:"+i);
        }
    }
}
```

第二章 高并发及线程安全

2.1 高并发及线程安全

- **高并发**：是指在某个时间点上，有大量的用户(线程)同时访问同一资源。例如：天猫的双11购物节、12306的在线购票在某个时间点上，都会面临大量用户同时抢购同一件商品/车票的情况。
- **线程安全**：在某个时间点上，当大量用户(线程)访问同一资源时，由于多线程运行机制的原因，可能会导致被访问的资源出现"数据污染"的问题。

2.2 多线程的运行机制

- 当一个线程启动后，JVM会为其分配一个独立的"线程栈区"，这个线程会在这个独立的栈区中运行。
- 看一下简单的线程的代码：

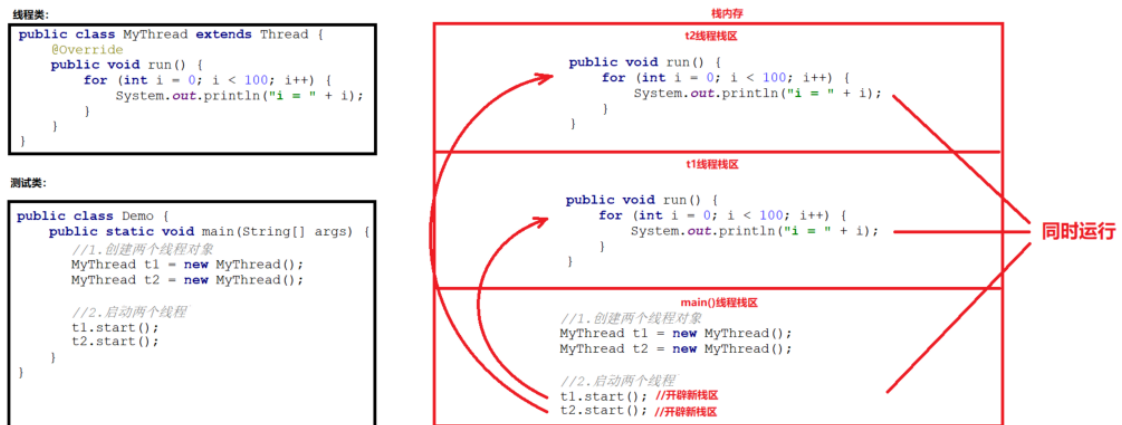
1. 一个线程类：

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

1. 测试类：

```
public class Demo {  
    public static void main(String[] args) {  
        //1.创建两个线程对象  
        MyThread t1 = new MyThread();  
        MyThread t2 = new MyThread();  
  
        //2.启动两个线程  
        t1.start();  
        t2.start();  
    }  
}
```

- 启动后，内存的运行机制：



- 多个线程在各自栈区中独立、无序的运行，当访问一些代码，或者同一个变量时，就可能会产生一些问题

2.3 多线程的安全性问题-可见性

- 例如下面的程序，先启动一个线程，在线程中将一个变量的值更改，而主线程却一直无法获得此变量的新值。

1. 线程类：

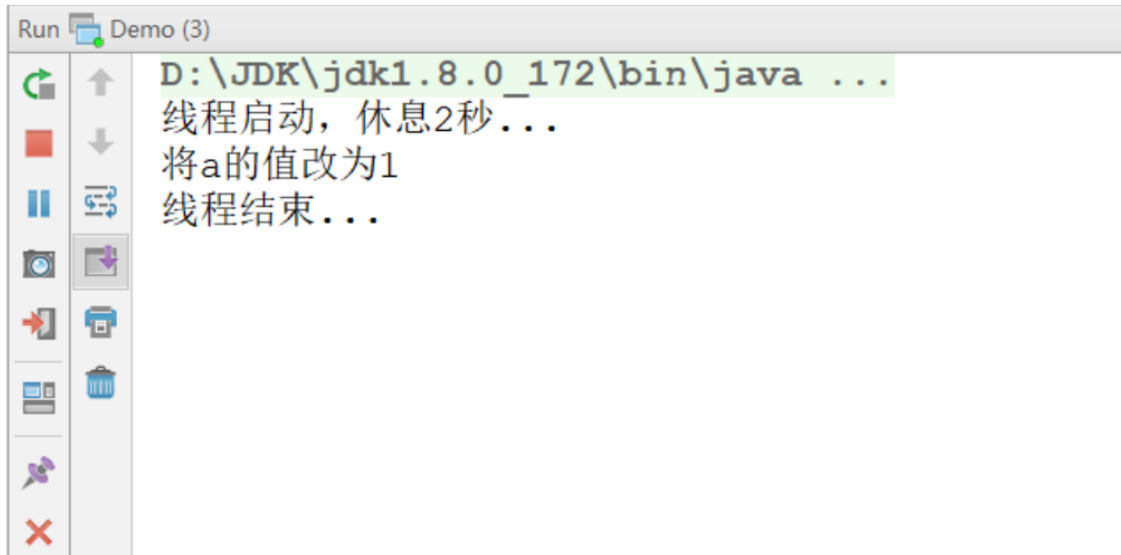
```
public class MyThread extends Thread {
    public static int a = 0;
    @Override
    public void run() {
        System.out.println("线程启动，休息2秒...");
        try {
            Thread.sleep(1000 * 2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("将a的值改为1");
        a = 1;
        System.out.println("线程结束...");
    }
}
```

1. 测试类：

```
public class Demo {
    public static void main(String[] args) {
        //1.启动线程
        MyThread t = new MyThread();
        t.start();

        //2.主线程继续
        while (true) {
            if (MyThread.a == 1) {
                System.out.println("主线程读到了a = 1");
            }
        }
    }
}
```


1. 启动后，控制台打印：



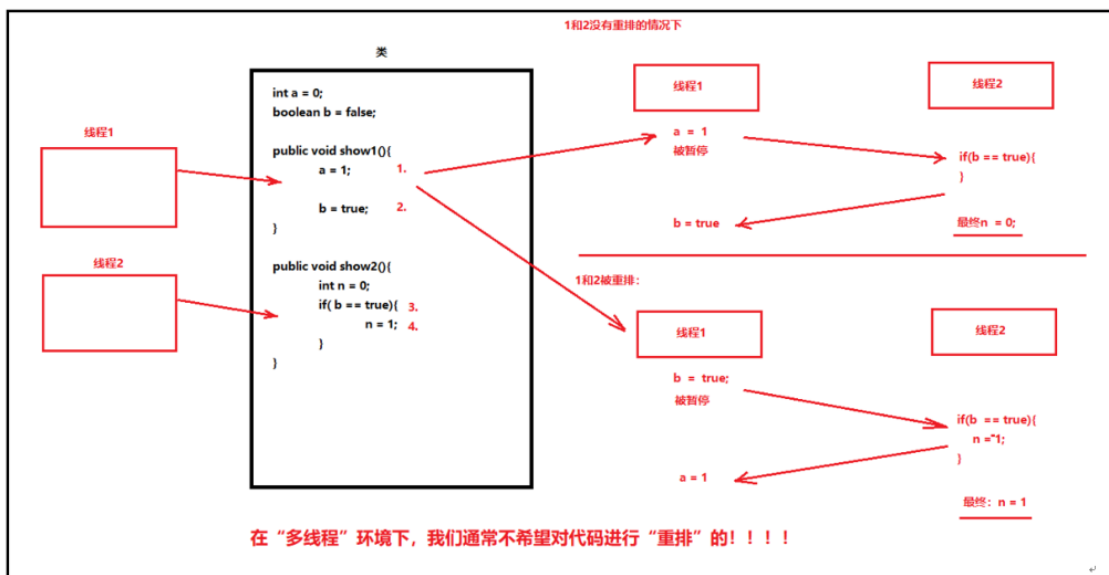
2.4 多线程的安全性问题-有序性

- 有些时候“编译器”在编译代码时，会对代码进行“重排”，例如：

```
int a = 10; //1  
int b = 20; //2  
int c = a + b; //3
```

第一行和第二行可能会被“重排”：可能先编译第二行，再编译第一行，总之在执行第三行之前，会将1,2编译完毕。1和2先编译谁，不影响第三行的结果。

- 但在“多线程”情况下，代码重排，可能会对另一个线程访问的结果产生影响：



多线程环境下，我们通常不希望对一些代码进行重排的！！

2.5 多线程的安全性问题-原子性

- 请看以下示例：

1.制作线程类

```
public class MyThread extends Thread {
    public static int a = 0;

    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            a++;
        }
        System.out.println("修改完毕! ");
    }
}
```

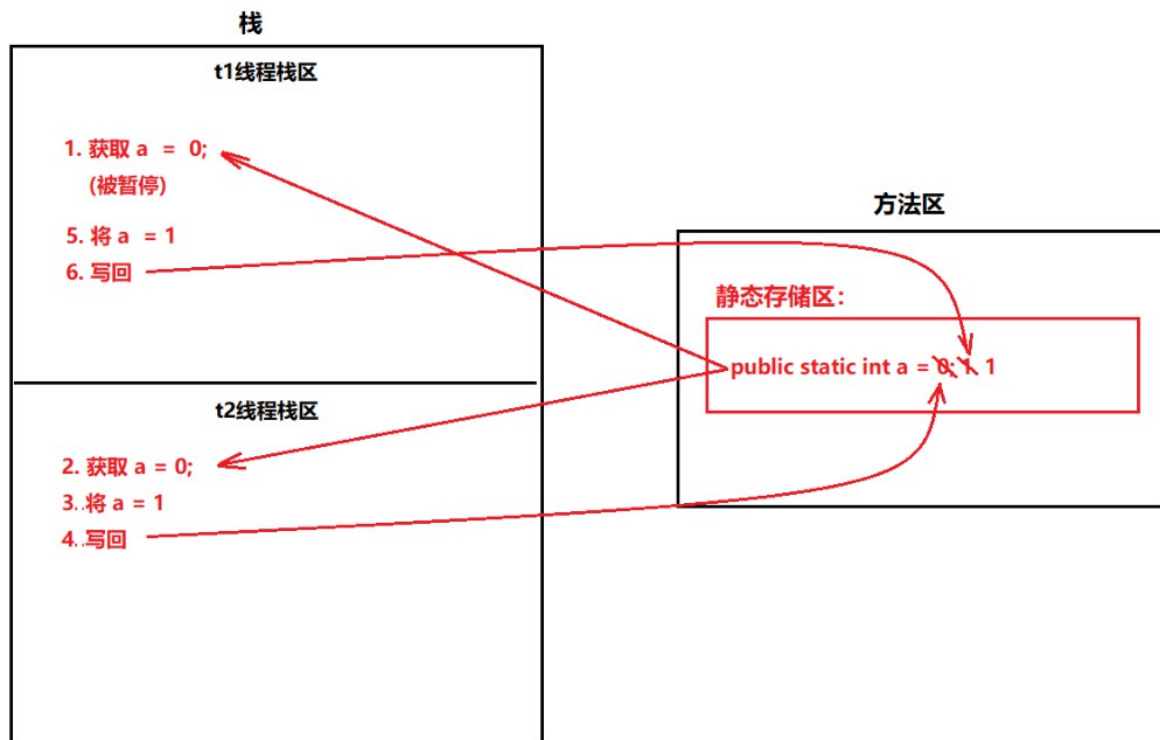
2.制作测试类

```
public class Demo {
    public static void main(String[] args) throws InterruptedException {
        //1.启动两个线程
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

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

        Thread.sleep(1000);
        System.out.println("获取a最终值: " + MyThread.a); //总是不准确的。原因：两个
        //线程访问a的步骤不具有：原子性
    }
}
```

内存工作原理图示：



线程t1先读取a 的值为：0

t1被暂停

线程t2读取a的值为：0

t2将a = 1

t2将a写回主内存

t1将a = 1

t1将a写回主内存(将t2更改的1，又更改为1)

所以两次加1，但结果仍为1，少加了一次。

原因：两个线程访问同一个变量a的代码不具有"原子性"