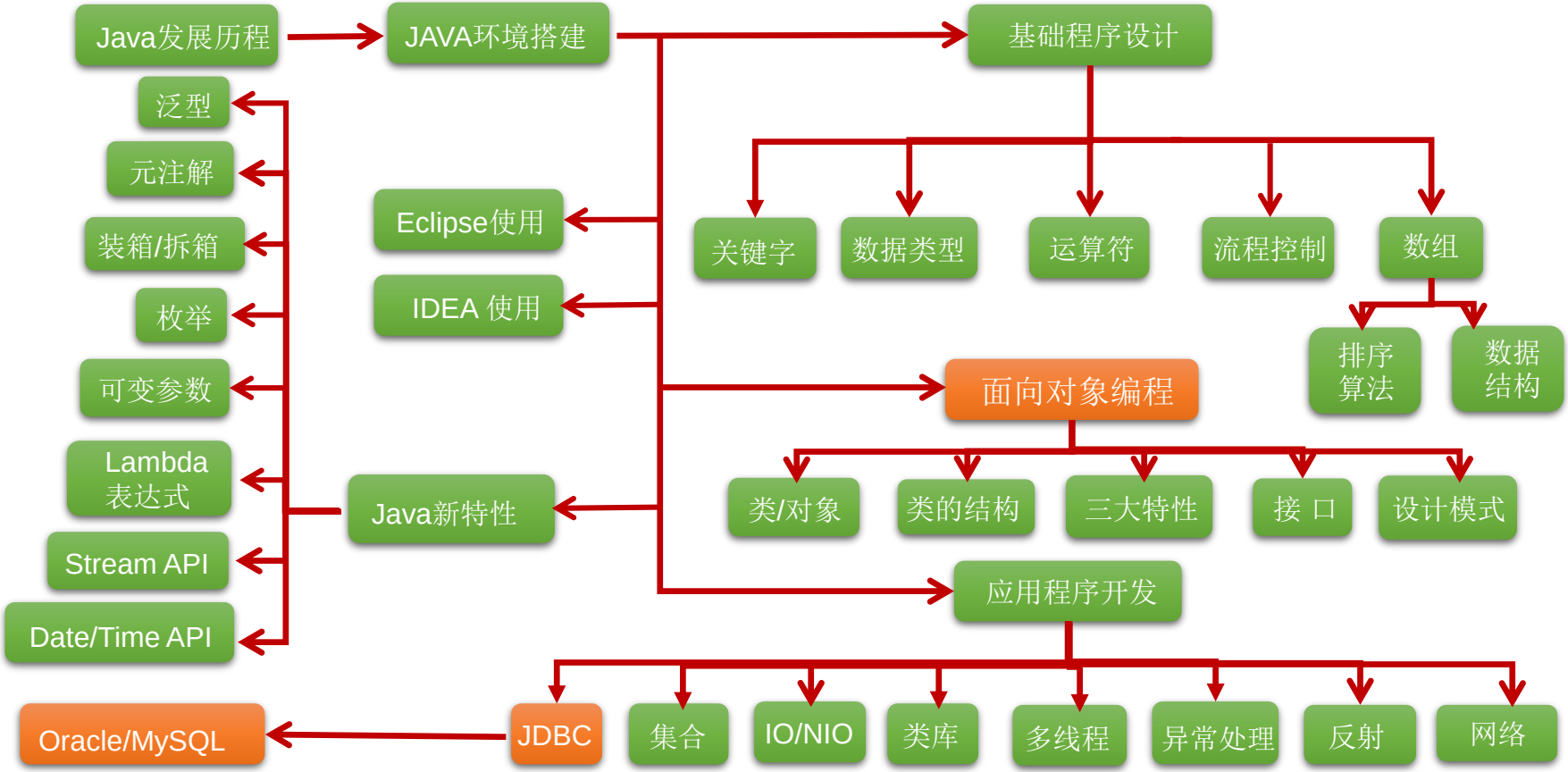
第8章
多线程

讲师：宋红康
新浪微博：尚硅谷-宋红康

Java基础知识图解



目录

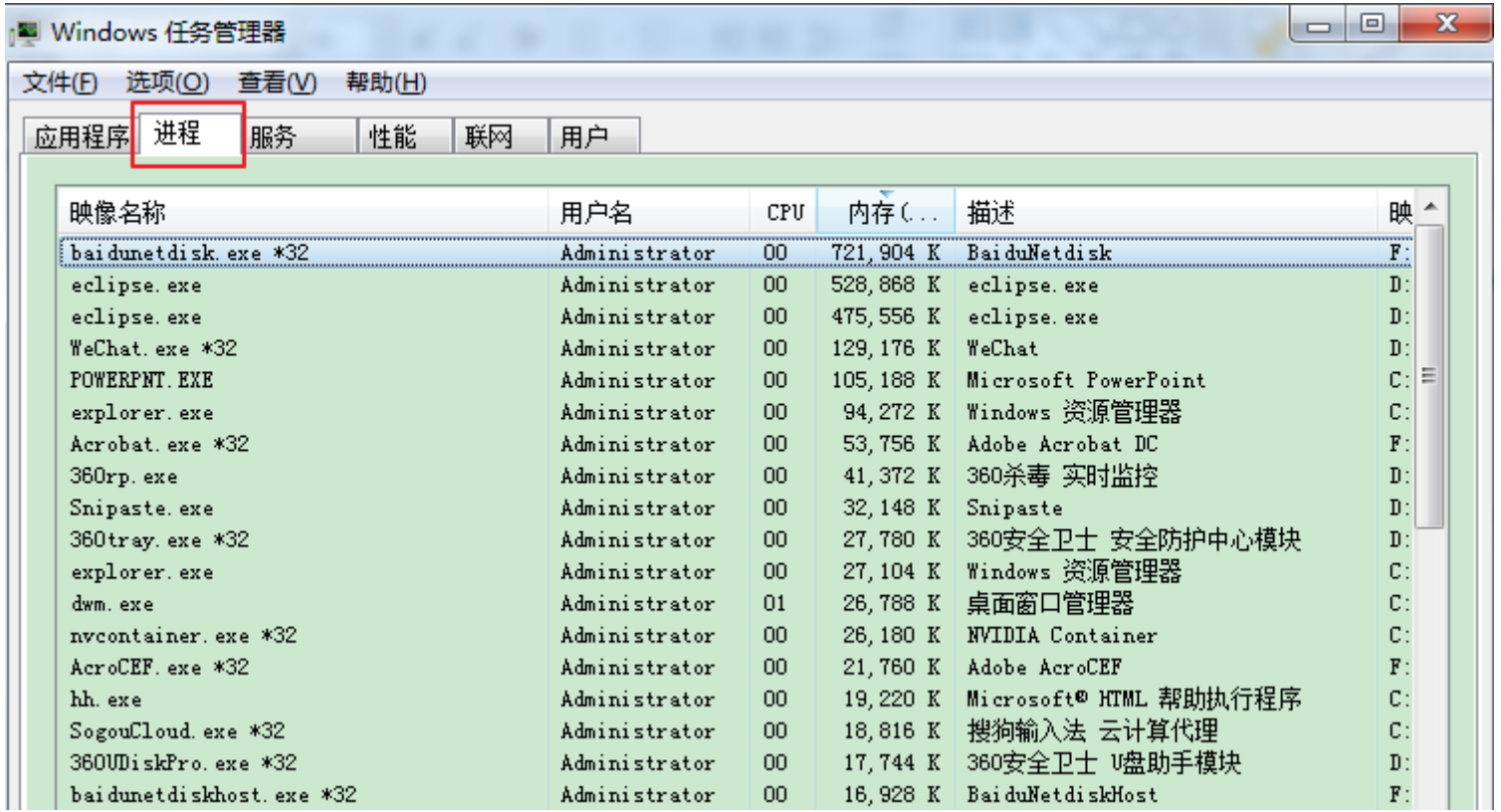
尚硅谷

8-1 基本概念：程序、进程、线程

8.1 基本概念：程序、进程、线程

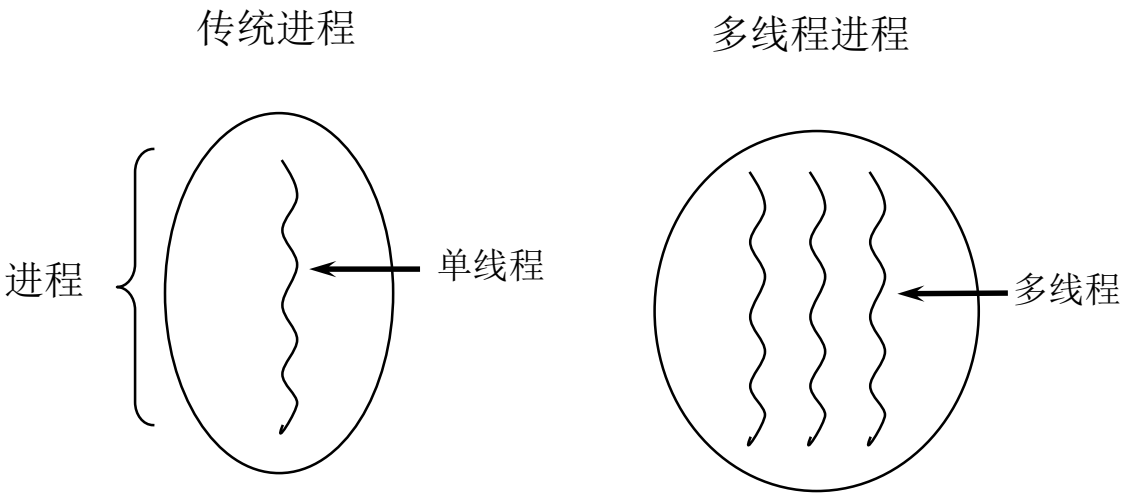
- **程序(program)**是为完成特定任务、用某种语言编写的一组指令的集合。即指一段静态的代码，静态对象。
- **进程(process)**是程序的一次执行过程，或是正在运行的一个程序。是一个动态的过程：有它自身的产生、存在和消亡的过程。——生命周期
  - 如：运行中的QQ，运行中的MP3播放器
  - 程序是静态的，进程是动态的
  - 进程作为资源分配的单位，系统在运行时会为每个进程分配不同的内存区域
- **线程(thread)**，进程可进一步细化为线程，是一个程序内部的一条执行路径。
  - 若一个进程同一时间并行执行多个线程，就是支持多线程的
  - 线程作为调度和执行的单位，每个线程拥有独立的运行栈和程序计数器(pc)，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→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。

8.1 基本概念：程序、进程、线程



映像名称	用户名	CPU	内存(...	描述	映
baidunetdisk.exe *32	Administrator	00	721,904 K	BaiduNetdisk	F:
eclipse.exe	Administrator	00	528,868 K	eclipse.exe	D:
eclipse.exe	Administrator	00	475,556 K	eclipse.exe	D:
WeChat.exe *32	Administrator	00	129,176 K	WeChat	D:
POWERPNT.EXE	Administrator	00	105,188 K	Microsoft PowerPoint	C:
explorer.exe	Administrator	00	94,272 K	Windows 资源管理器	C:
Acrobat.exe *32	Administrator	00	53,756 K	Adobe Acrobat DC	F:
360rp.exe	Administrator	00	41,372 K	360杀毒 实时监控	D:
Snipaste.exe	Administrator	00	32,148 K	Snipaste	D:
360tray.exe *32	Administrator	00	27,780 K	360安全卫士 安全防护中心模块	D:
explorer.exe	Administrator	00	27,104 K	Windows 资源管理器	C:
dwm.exe	Administrator	01	26,788 K	桌面窗口管理器	C:
nvcontainer.exe *32	Administrator	00	26,180 K	NVIDIA Container	C:
AcroCEF.exe *32	Administrator	00	21,760 K	Adobe AcroCEF	F:
hh.exe	Administrator	00	19,220 K	Microsoft® HTML 帮助执行程序	C:
SogouCloud.exe *32	Administrator	00	18,816 K	搜狗输入法 云计算代理	C:
360UDiskPro.exe *32	Administrator	00	17,744 K	360安全卫士 U盘助手模块	D:
baidunetdiskhost.exe *32	Administrator	00	16,928 K	BaiduNetdiskHost	F:

进程与线程



- 单核CPU和多核CPU的理解
  - 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。例如：虽然有多车道，但是收费站只有一个工作人员在收费，只有收了费才能通过，那么CPU就好比收费人员。如果有某个人不想交钱，那么收费人员可以把他“挂起”（晾着他，等他想通了，准备好了钱，再去收费）。但是因为CPU时间单元特别短，因此感觉不出来。
  - 如果是多核的话，才能更好的发挥多线程的效率。（现在的服务器都是多核的）
  - 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。
- 并行与并发
  - 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。
  - 并发：一个CPU(采用时间片)同时执行多个任务。比如：秒杀、多个人做同一件事。

使用多线程的优点

背景：以单核CPU为例，只使用单个线程先后完成多个任务（调用多个方法），肯定比用多个线程来完成用的时间更短，为何仍需多线程呢？

多线程程序的优点：

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. 提高计算机系统CPU的利用率
3. 改善程序结构。将既长又复杂的进程分为多个线程，独立运行，利于理解和修改

何时需要多线程

- 程序需要同时执行两个或多个任务。
- 程序需要实现一些需要等待的任务时，如用户输入、文件读写操作、网络操作、搜索等。
- 需要一些后台运行的程序时。

8-2 线程的创建和使用

```
public class Sample {
    public void method1(String str) {
        System.out.println(str);
    }

    public void method2(String str) {
        method1(str);
    }

    public static void main(String[] args) {
        Sample s = new Sample();
        s.method2("hello!");
    }
}
```

注意：

左边的程序不是多线程！

线程的创建和启动

- Java语言的JVM允许程序运行多个线程，它通过**java.lang.Thread**类来体现。

- Thread类的特性
  - 每个线程都是通过某个特定Thread对象的run()方法来完成操作的，经常把run()方法的主体称为**线程体**
  - 通过该Thread对象的start()方法来启动这个线程，而非直接调用run()

Thread类

- 构造器
  - **Thread()**：创建新的Thread对象
  - **Thread(String threadname)**：创建线程并指定线程实例名
  - **Thread(Runnable target)**：指定创建线程的目标对象，它实现了Runnable接口中的run方法
  - **Thread(Runnable target, String name)**：创建新的Thread对象

API中创建线程的两种方式

- JDK1.5之前创建新执行线程有两种方法：
  - 继承Thread类的方式
  - 实现Runnable接口的方式

- 方式一：**继承Thread类**

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。

定义

```
class MyThread02 implements Runnable {
   public void run() {
      System.out.println("-----MyThread02");
   }
}
```

实现的时候第二次实现不能在第一次上面启动，必须重新创建一次

```
MyThread t1 = new MyThread();
t1.start();
MyThread t2 = new MyThread();
t2.start();
```

mt子线程的创建和启动过程

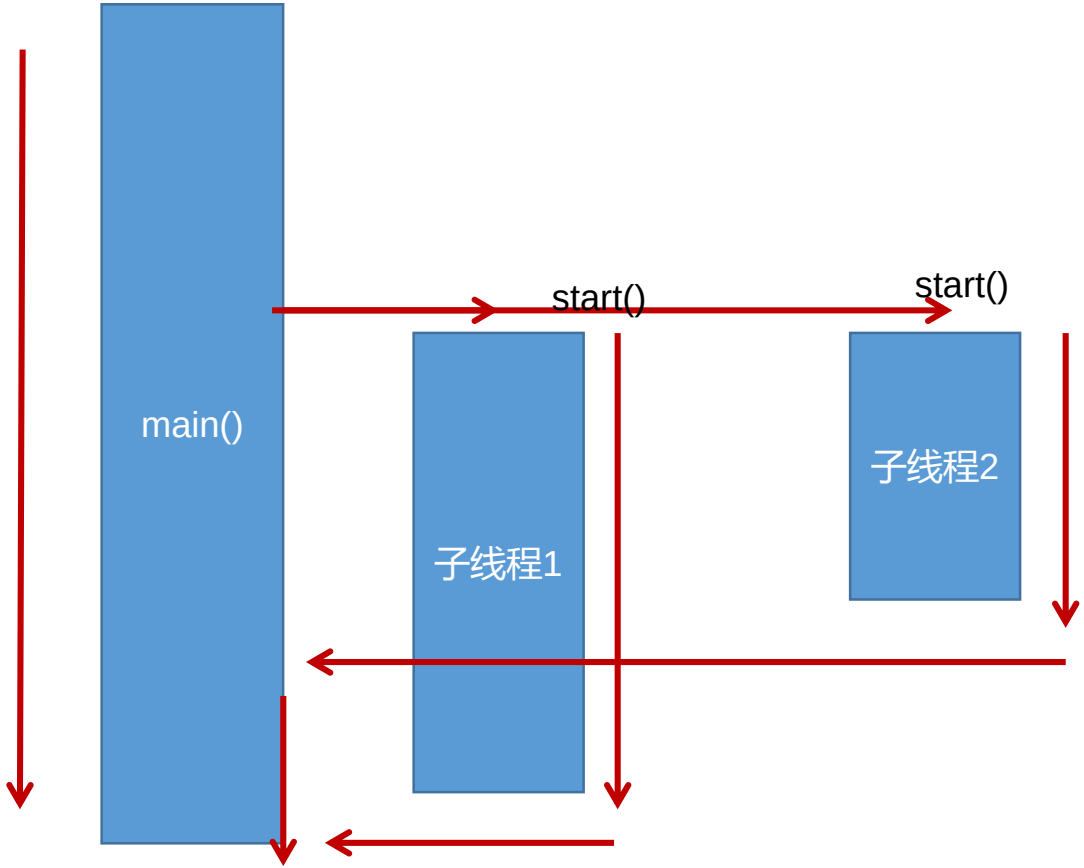
```
class MyThread extends Thread{
    public MyThread(){
        super();
    }
    public void run(){
        for(int i = 0;i<100;i++){
            System.out.println("子线程："+i);
        }
    }
}
public class TestThread {
    public static void main(String[] args) {
        //1.创建线程
        MyThread mt = new MyThread();
        //2.启动线程，并调用当前线程的run()方法。
        mt.start();
```

3

1

2

8.2 线程的创建和使用



● 注意点：

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。
2. 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。
3. 想要启动多线程，必须调用start方法。
4. 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”。

API中创建线程的两种方式

● 方式二：实现Runnable接口

1) 定义子类，实现Runnable接口。

2) 子类中重写Runnable接口中的run方法。

3) 通过Thread类含参构造器创建线程对象。

4) 将Runnable接口的子类对象作为实际参数传递给Thread类的构造器中。

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

具体创建代码

```
class MyThread01 extends Thread {
    @Override
    public void run() {
        System.out.println("-----MyThread01");
    }
}

class MyThread02 implements Runnable {
    public void run() {
        System.out.println("-----MyThread02");
    }
}
```

开发中优先选择实现Runnable接口的方式，因为

1. 实现的方式没类的单继承性的局限性
2. 实现的方式更适合来处理多个线程共享数据的情况

继承方式和实现方式的联系与区别

public class **Thread** extends Object implements Runnable

- 区别
  - 继承Thread：线程代码存放Thread子类run方法中。
  - 实现Runnable：线程代码存在接口的子类的run方法。

两种方法：继承Thread类，实现Runnable方法。

- 实现方式的好处
  - 避免了单继承的局限性
  - 多个线程可以共享同一个接口实现类的对象，非常适合多个相同线程来处理同一份资源。

练　习

创建两个分线程，让其中一个线程输出1-100之间的偶数，另一个线程输出1-100之间的奇数。

Thread类的有关方法(1)

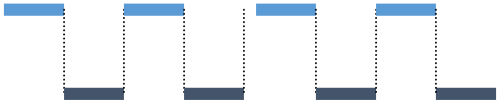
- **void start():** 启动线程，并执行对象的run()方法
- **run():** 线程在被调度时执行的操作
- **String getName():** 返回线程的名称
- **void setName(String name)**:设置该线程名称
- **static Thread currentThread():** 返回当前线程。在Thread子类中就是this，通常用于主线程和Runnable实现类

Thread类的有关方法(2)

- **static void yield()：** 线程让步
  - 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
  - 若队列中没有同优先级的线程，忽略此方法
- **join()：** 当某个程序执行流中调用其他线程的 join() 方法时，调用线程将被阻塞，直到 join() 方法加入的 join 线程执行完为止
  - 低优先级的线程也可以获得执行
- **static void sleep(long millis)：** (指定时间:毫秒)
  - 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。
  - 抛出InterruptedException异常
- **stop():** 强制线程生命期结束，不推荐使用
- **boolean isAlive()：** 返回boolean，判断线程是否还活着

线程的调度

- 调度策略
  - 时间片



  - 抢占式：高优先级的线程抢占CPU

- Java的调度方法
  - 同优先级线程组成先进先出队列（先到先服务），使用时间片策略
  - 对高优先级，使用优先调度的抢占式策略

线程的优先级

- 线程的优先级等级
  - MAX_PRIORITY：10
  - MIN _PRIORITY：1
  - NORM_PRIORITY：5
- 涉及的方法
  - getPriority()：返回线程优先值
  - setPriority(int newPriority)：改变线程的优先级
- 说明
  - 线程创建时继承父线程的优先级
  - 低优先级只是获得调度的概率低，并非一定是在高优先级线程之后才被调用

补充：线程的分类

Java中的线程分为两类：一种是**守护线程**，一种是**用户线程**。

- 它们在几乎每个方面都是相同的，唯一的区别是判断JVM何时离开。
- 守护线程是用来服务用户线程的，通过在start()方法前调用**thread.setDaemon(true)**可以把一个用户线程变成一个守护线程。
- Java垃圾回收就是一个典型的守护线程。
- 若JVM中都是守护线程，当前JVM将退出。
- 形象理解：兔死狗烹，鸟尽弓藏

8-3 线程的生命周期

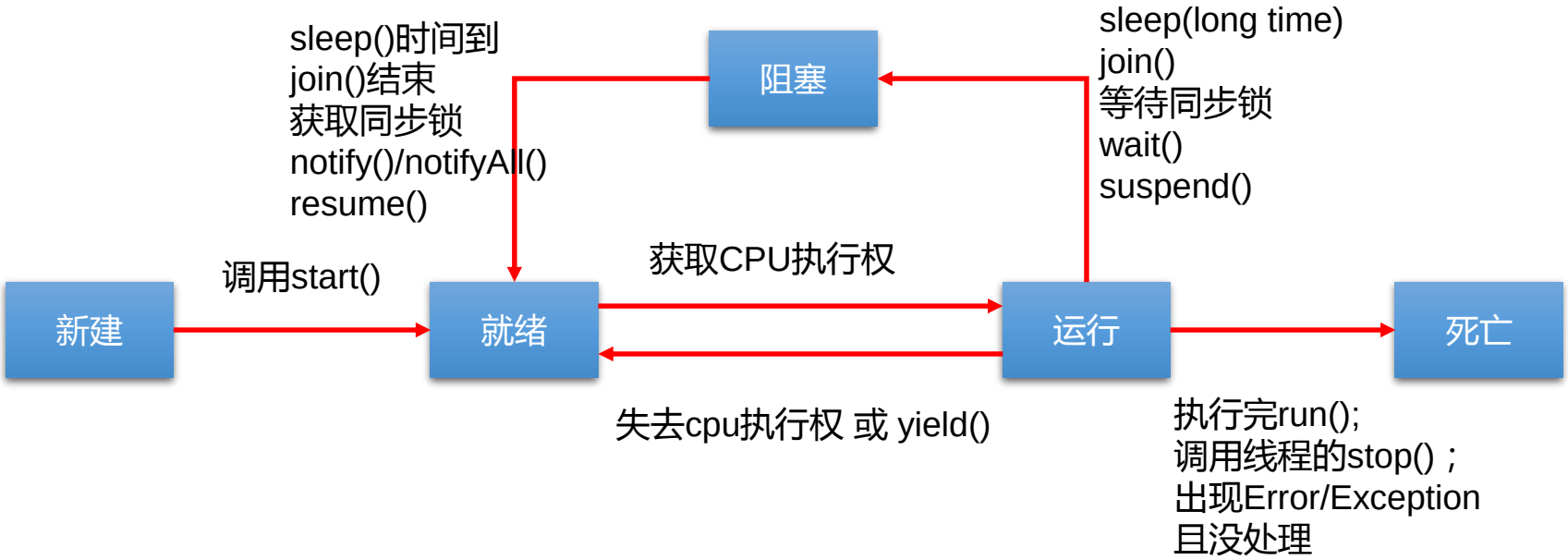
12.3 线程的生命周期

- **JDK中用Thread.State类定义了线程的几种状态**

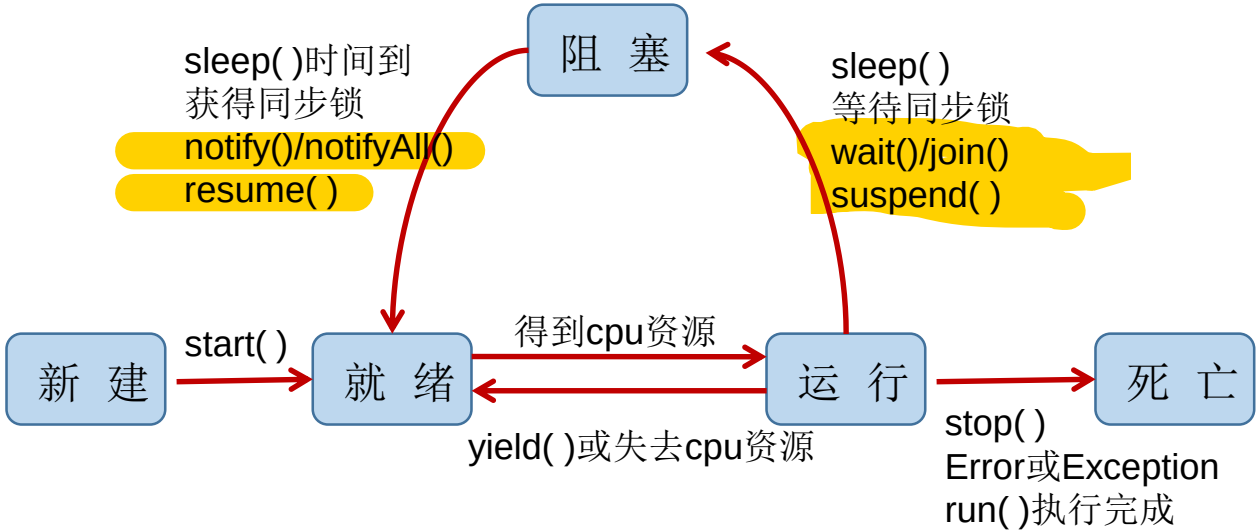
要想实现多线程，必须在主线程中创建新的线程对象。Java语言使用Thread类及其子类的对象来表示线程，在它的一个完整的生命周期中通常要经历如下的**五种状态**：

- **新建：** 当一个Thread类或其子类的对象被声明并创建时，新生的线程对象处于新建状态
- **就绪：** 处于新建状态的线程被start()后，将进入线程队列等待CPU时间片，此时它已具备了运行的条件，只是没分配到CPU资源
- **运行：** 当就绪的线程被调度并获得CPU资源时,便进入运行状态， run()方法定义了线程的操作和功能
- **阻塞：** 在某种特殊情况下，被人为挂起或执行输入输出操作时，让出 CPU 并临时中止自己的执行，进入阻塞状态
- **死亡：** 线程完成了它的全部工作或线程被提前强制性地中止或出现异常导致结束

线程的生命周期



12.3 线程的生命周期

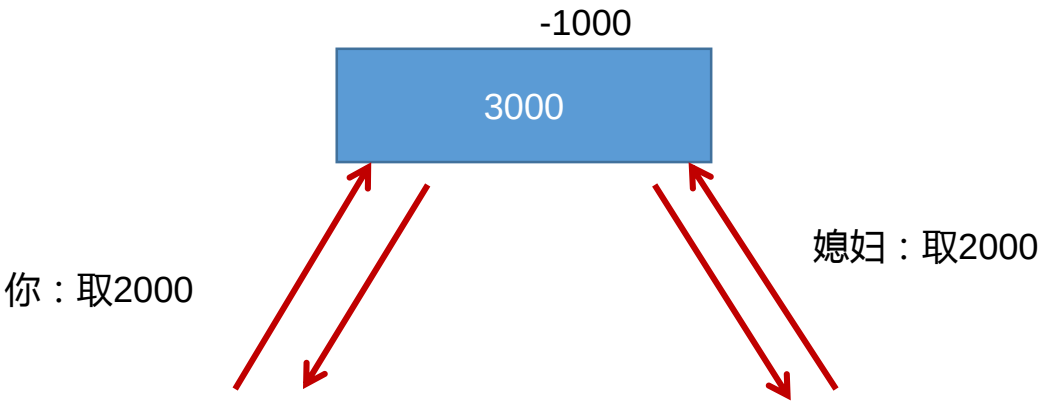


线程状态转换图

8-4 线程的同步

问题的提出

- 多个线程执行的不确定性引起执行结果的不稳定
- 多个线程对账本的共享，会造成操作的不完整性，会破坏数据。



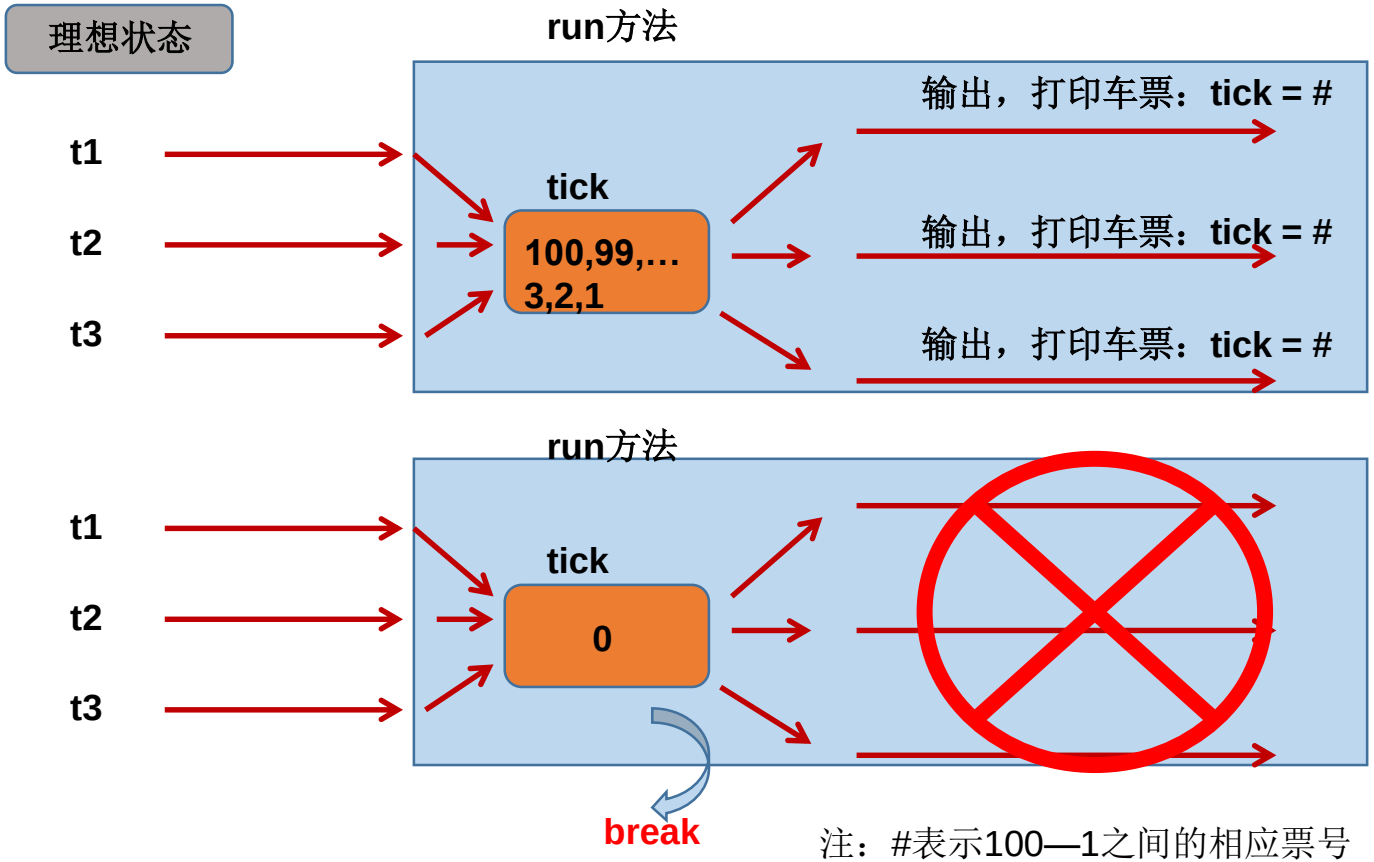
例 题

模拟火车站售票程序，开启三个窗口售票。

```
class Ticket implements Runnable {
    private int tick = 100;

    public void run() {
        while (true) {
            if (tick > 0) {
System.out.println(Thread.currentThread
().getName() + "售出车票，tick号为：" +
tick--);
            } else
                break;
        }
    }
}
```

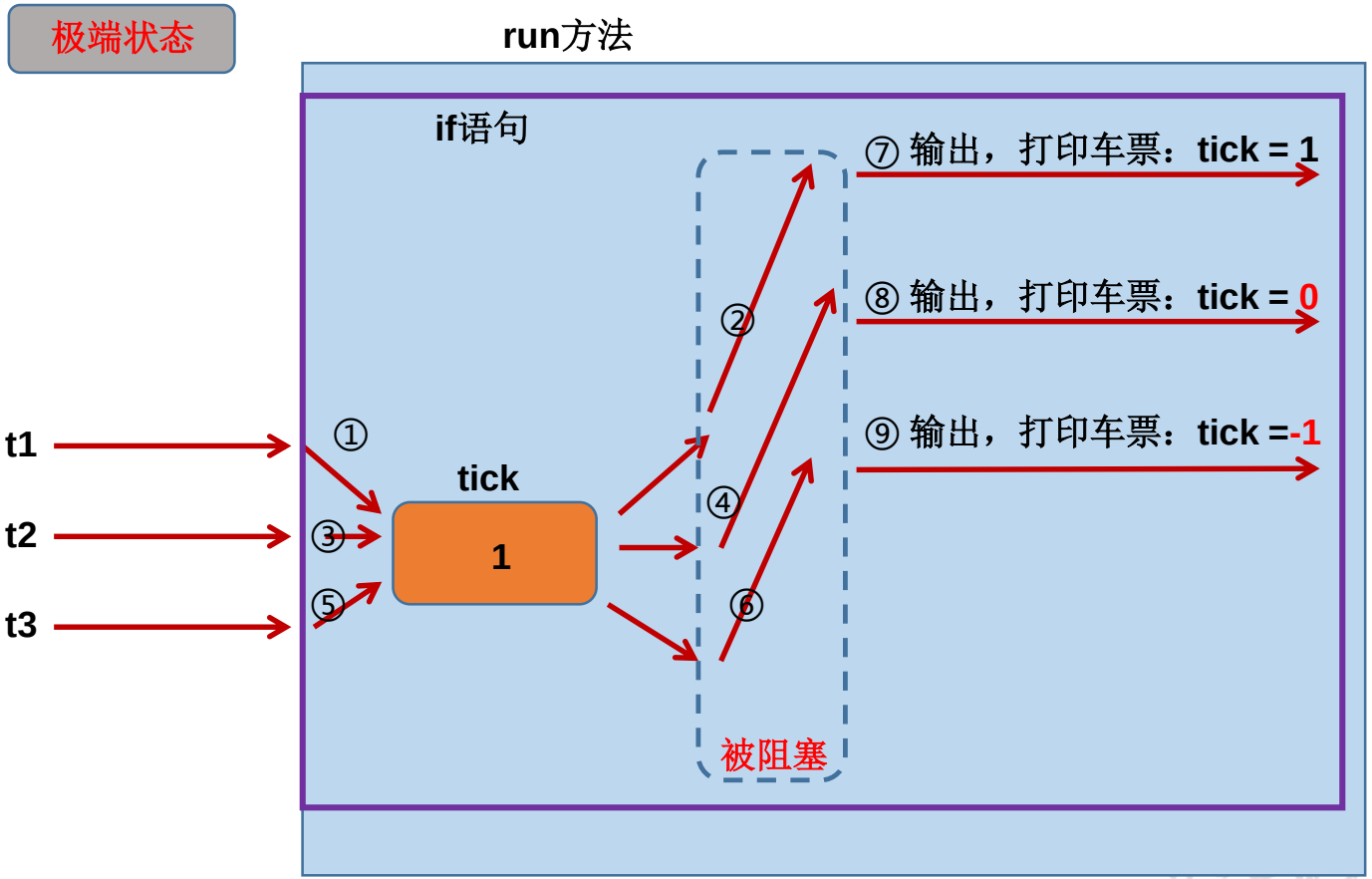
```
class TicketDemo {
public static void main(String[] args) {

    Ticket t = new Ticket();

    Thread t1 = new Thread(t);
    Thread t2 = new Thread(t);
    Thread t3 = new Thread(t);
    t1.setName("t1窗口");
    t2.setName("t2窗口");
    t3.setName("t3窗口");
    t1.start();
    t2.start();
    t3.start();
  }
}
```

8.4 线程的同步



注：#表示100—1之间的相应票号

8.4 线程的同步


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

8.4 线程的同步

```
private int tick = 100;
public void run(){
    while(true){
       if(tick>0){
          try{
             Thread.sleep(10);
          }catch(InterruptedException e){ e.printStackTrace();}
         System.out.println(Thread.currentThread().getName()+“售出车票，tick号为：       "+tick--);
} } }
```

1. 多线程出现了安全问题

2. 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。

3. 解决办法：

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。

Synchronized的使用方法

- Java对于多线程的安全问题提供了专业的解决方式：同步机制

1. 同步代码块：

```
synchronized (对象){
    // 需要被同步的代码;
}
```

需要被同步的代码：需要被同步的代码

共享数据：多个线程共同操作的变量

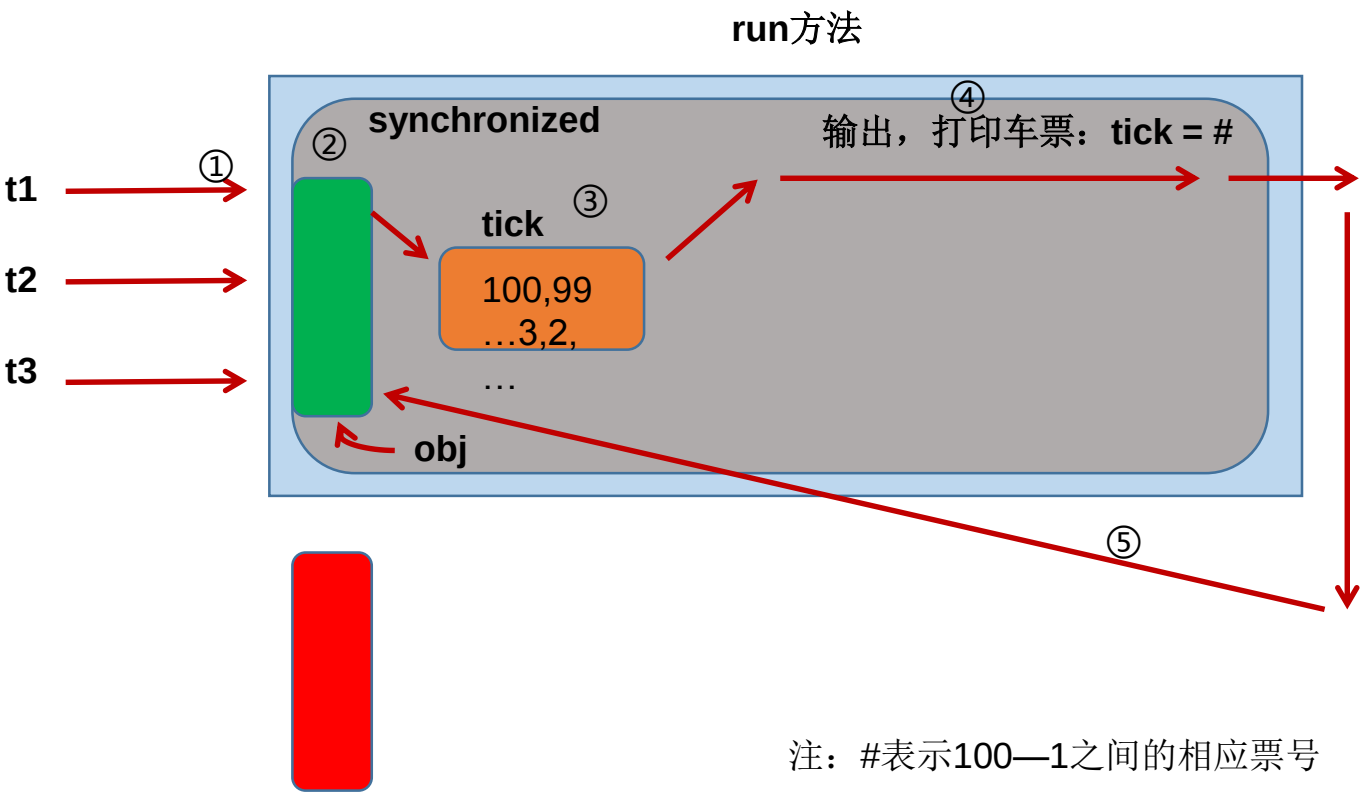
同步监视器(对象)即锁。任何一个类的对象，都可以充当锁。

2. synchronized还可以放在方法声明中，表示整个方法为同步方法。

例如：

```
public synchronized void show (String name){
    ....
}
```

分析同步原理



注：#表示100—1之间的相应票号

同步机制中的锁

- **同步锁机制：**

在《Thinking in Java》中，是这么说的：对于并发工作，你需要某种方式来防止两个任务访问相同的资源（其实就是共享资源竞争）。防止这种冲突的方法就是当资源被一个任务使用时，在其上加锁。第一个访问某项资源的任务必须锁定这项资源，使其他任务在其被解锁之前，就无法访问它了，而在其被解锁之时，另一个任务就可以锁定并使用它了。

- **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）。
  - 同步方法的锁：静态方法（类名.class）、非静态方法（this）
  - 同步代码块：自己指定，很多时候也是指定为this或类名.class
- **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
  - 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）

同步的范围

synchronized(this)
或者Object obj = new Object(),
synchronized(obj)

1、如何找问题，即代码是否存在线程安全？（非常重要）

（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

2、如何解决呢？（非常重要）

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。
即所有操作共享数据的这些语句都要放在同步范围中

3、切记：

- 范围太小：没锁住所有有安全问题的代码
- 范围太大：没发挥多线程的功能。

能够调用synchronized(Window2.class)运行的原因：
1.类也是对象
Class clazz = Window2.class
clazz为变量，Window2.class为变量，主要在后面反射的时候用到
2.类是唯一的，后面反射会讲到类只会被加载一次
在实现Runnable接口创建多线程的过程中，我们可以考虑使用this充当同步监视器

3.在继承Thread类创建多线程的方式中，慎用this充当同步监视器，考虑使用当前类充当同步监视器(总之一定保证唯一即可)

```
private synchronized void show()//同步监视器：this
//因为这样写同步监视器唯一
```

释放锁的操作

- 当前线程的同步方法、同步代码块执行结束。
- 当前线程在同步代码块、同步方法中遇到break、return终止了该代码块、该方法的继续执行。
- 当前线程在同步代码块、同步方法中出现了未处理的Error或Exception，导致异常结束。
- 当前线程在同步代码块、同步方法中执行了线程对象的**wait()**方法，当前线程暂停，并释放锁。

不会释放锁的操作

- 线程执行同步代码块或同步方法时，程序调用Thread.sleep()、Thread.yield()方法暂停当前线程的执行

- 线程执行同步代码块时，其他线程调用了该线程的suspend()方法将该线程挂起，该线程不会释放锁（同步监视器）。
  - 应尽量避免使用suspend()和resume()来控制线程

Thread.yield()作用：让步。它能让当前线程由运行状态进入到就绪状态，从而让其他具有相同优先级的等待线程获取执行权，但是并不能保证在当前线程掉哟个yield()之后，其它具有相同优先级的线程就一定能获得执行权，也有可能又进入到运行状态继续运行！

单例设计模式之懒汉式(线程安全)

```
class Singleton {
        private static Singleton instance = null;
        private Singleton(){}
        public static Singleton getInstance(){
                if(instance==null){
                        synchronized(Singleton.class){
                                if(instance == null){
                                        instance=new Singleton();
                                }       }       }
                return instance;
        }       }
public class SingletonTest{
        public static void main(String[] args){
                Singleton s1=Singleton.getInstance();
                Singleton s2=Singleton.getInstance();
                System.out.println(s1==s2);
        }       }
```

懒汉线程原理：
原始的懒汉线程定义如下：

```
class Bank{
   private static Bank instance = null;
   public static Bank getInstance()
   {
       if(instance == null)
       {
          instance = new Bank();
          return instance;
       }
   }
}
```

static变量在类的初始化的时候就会调用，只需要初始化一次即可，如果不加锁的情况下可能多次初始化，与实际情况不符，因此这里需要对相应内容进行加锁。

原始的懒汉式问题，定义如下：

```
class Bank{
    private static Bank instance = null;
    public static Bank getInstance()
    {
        if(instance == null)
        {
            instance = new Bank();
        }
        return instance;
    }
}
```

```
public static Bank getInstance(){
   //方法一：效率稍差
   synchronized(Bank.class)
   {
       if(instance == null)
       {
           instance = new Bank();
       }
       return instance;
   }
}
```

这里的instance可能被多次初始化，这显然不对。

```
public static Bank getInstance()
{
    if(instance == null)
    {
        synchronized(Bank.class)
        {
            if(instance == null)
            {
                instance = new Bank();
            }
        }
    }
}
```

线程的死锁问题

- 死锁
  - 不同的线程分别占用对方需要的同步资源不放弃，都在等待对方放弃自己需要的同步资源，就形成了线程的死锁
  - 出现死锁后，不会出现异常，不会出现提示，只是所有的线程都处于阻塞状态，无法继续
- 解决方法
  - 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

DeadLock.java

8.4 线程的同步

同步锁构成的死锁

```
public class DeadLockTest {
    public static void main(String[] args) {
        final StringBuffer s1 = new StringBuffer();
        final StringBuffer s2 = new StringBuffer();
        new Thread() {
            public void run() {
                synchronized (s1) {
                    s2.append("A");
                    synchronized (s2) {
                        s2.append("B");
                        System.out.print(s1);
                        System.out.print(s2);
                    }
                }
            }
        }.start();

        new Thread() {
            public void run() {
                synchronized (s2) {
                    s2.append("C");
                    synchronized (s1) {
                        s1.append("D");
                        System.out.print(s2);
                        System.out.print(s1);
                    }
                }
            }
        }.start();
    }
}
```

3. Lock(锁)

- 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对象加锁，线程开始访问共享资源之前应先获得Lock对象。

- 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。

3. Lock(锁)

```
class A{
        private final ReentrantLock lock = new ReenTrantLock();
        public void m(){
                lock.lock();
                try{
                        //保证线程安全的代码;
                }
                finally{
                        lock.unlock();
                }
        }
}
```

注意：如果同步代码有异常，要将unlock()写入finally语句块

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）

同步代码块：

```
synchronize{
}
```

同步方法：

```
private synchronized
show()
```

练 习1

银行有一个账户。

有两个储户分别向同一个账户存3000元，每次存1000，存3次。每次存完打印账户余额。

问题：该程序是否有安全问题，如果有，如何解决？

【提示】

1，明确哪些代码是多线程运行代码，须写入run()方法

2，明确什么是共享数据。

3，明确多线程运行代码中哪些语句是操作共享数据的。

拓展问题：可否实现两个储户交替存钱的操作

这里我专门查找了一下考研操作系统中的PV操作来理解同步和互斥的进程

1.使用信号量实现进程同步

```
P1(){
   x;
   V(S);
}
P2(){
   P(S);
   y;
}
```

2.使用信号量实现进程互斥

```
semaphore S=1;//初始化信号量(资源个数)
P1(){
   P(S);
   进程P1的临界值;
   V(S);
}
P2(){
   P(S);
   进程P2的临界值
   V(S);
}
```

P相当于notify()(唤醒其他线程),V相当于wait()(阻塞当前线程)，在P和V的外面还需要加入一个synchronized(this)锁

8-5 线程的通信

例 题

使用两个线程打印 1-100。线程1, 线程2 交替打印

使用synchronized没有办法达到交替的效果，线程1、线程2的出现是随机的。
要想交替运行，线程1运行之后必须阻塞，这里调用wait方法，而如果想要恢复到就绪状态，则需要调用notify()以及notifyAll()这两个方法

实际上就是将P操作换成notify(),V操作换成wait()即可，注意外面还要加上synchronized(this)锁，这样就由原先随机的PV操作但是不保证交替运行变换为现在的交替运行机制

```
class Communication implements Runnable {
    int i = 1;
    public void run() {
        while (true) {
            synchronized (this) {
                notify();
                if (i <= 100) {
                    System.out.println(Thread.currentThread().getName() +
":" + i++);
                } else
                    break;
                try {
                    wait();打印完成需要阻塞一下，使用wait()方法，方便其他进程运行
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

System.out.println
简写sout

alt+shift+z快捷键，能够将选中部分代码给包起来

如果只有wait()方法，线程1运行wait()阻塞，线程2运行wait()之后也阻塞，这样就没法继续打印了

题目之中只有两线程，线程1会把线程2唤醒，线程2会把线程1唤醒，因此这里直接使用notify()函数即可
线程2运行的时候notify()，线程1被唤醒，但是此时线程1进入不了，因为线程2握住这个锁(synchronized(this))，所以此时线程2会继续运行，直至wait()函数部分，这里wait()会将锁释放，并且阻塞当前的线程，

这也就是在运行的过程中为什么不能先wait()然后再notify()的原因，这样一直wait()一直阻塞，始终拿不到对应的锁的内容

注意这里测试代码运行的时候，需要两个线程共用一个类
只有这样才能够保证使用的是一个锁，否则会进入死锁状态
进入死锁状态的原因是synchronized(this)握的不是同一把锁

```
public static void main(String[] args)
{
    Communication c = new Communication();
    new Thread(c).start();
    new Thread(c).start();
}
```

- **wait() 与 notify() 和 notifyAll()**
  - **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.
- 这三个方法只有在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()
- 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）

面试题：sleep()和wait()的异同
1.相同点：一旦执行方法，都可以使得当前的线程进入阻塞状态
2.不同点：1）两个方法声明的位置不同，Thread类中声明sleep()，Object类中声明wait()
2)调用的要求不同：sleep()可以在任何需要的场景下调用，wait()必须使用在同步代码下
3)关于是否释放同步监视器：如果两个方法都使用在同步代码块或同步方法中，sleep不会释放锁，而wait会释放锁

经典例题：生产者/消费者问题

- 生产者(Productor)将产品交给店员(Clerk)，而消费者(Customer)从店员处取走产品，店员一次只能持有固定数量的产品(比如:20），如果生产者试图生产更多的产品，店员会叫生产者停一下，如果店中有空位放产品了再通知生产者继续生产；如果店中没有产品了，店员会告诉消费者等一下，如果店中有产品了再通知消费者来取走产品。
- 这里可能出现两个问题：
  - 生产者比消费者快时，消费者会漏掉一些数据没有取到。
  - 消费者比生产者快时，消费者会取相同的数据。

所以这里需要一个生产者一个消费者，一个生产者一个消费者的进行运转

这里如果使用一个生产者和一个消费者，代码如下：

```
public void addProduct(){
    while (true) {
        Thread.sleep((long) (Math.random()*100));
        synchronized (this)
        {
            notifyAll();
            this.product++;
            System.out.println("生成者生产了
第"+Integer.toString(this.product));
            wait();
        }
    }
```

```
public void getProduct()
    {
        while(true)
        {

            Thread.sleep((long) (Math.random()*100));
            synchronized (this)
            {
                notifyAll();
                if(this.product >= 0)
                {
                    System.out.println("消费者消费了第" +
Integer.toString(this.product));
                    this.product--;
                }
                wait();
            }
        }
    }
```

如果想要取或者拿有连续性，则在每次取之后调用，在if后面加上else，这样在拿或者取之后，并没有将当前线程阻塞，也就是说下一次还可以竞争锁的拿取，而不是直接阻塞不让拿锁了，对应代码如下：

```
public void addProduct(){
    while (true) {
        Thread.sleep((long) (Math.random()*100));
      synchronized (this)
      {
        notifyAll();
        if(this.product <= 20)
        {
          this.product++;
          System.out.println("生成者生产了
第"+Integer.toString(this.product));
        }
        else
        {
          wait();
        }
      }
    }
```

```
  public void getProduct()
  {
    while(true)
    {
      Thread.sleep((long) (Math.random()*100));
      synchronized (this)
      {
        notifyAll();
        if(this.product >= 0)
        {
          System.out.println("消费者消费了第" +
Integer.toString(this.product));
          this.product--;
        }
        else
        {
          wait();
        }
      }
    }
  }
```

```
class Clerk { // 售货员
    private int product = 0;

    public synchronized void addProduct() {
        if (product >= 20) {
            try {
                wait();
            } catch (InterruptedException e)
{
                e.printStackTrace();
            }
        } else {
            product++;
            System.out.println("生产者生产了
第" + product + "个产品");
            notifyAll();
        }
    }
```

负责结合生成者的消费者处理具体关系的

```
public synchronized void getProduct() {
    if (this.product <= 0) {
        try {
            wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    } else {
        System.out.println("消费者取走了第" +
product + "个产品");
        product--;
        notifyAll();
    }
   }
}
```

```
class Productor implements Runnable { // 生产者
    Clerk clerk;

    public Productor(Clerk clerk) {
        this.clerk = clerk;
    }

    public void run() {
        System.out.println("生产者开始生产产品");
        while (true) {
            try {
                Thread.sleep((int) Math.random() * 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            clerk.addProduct();
        }
    }
}
```

生产者、消费者都调用了clerk之中的函数，进而间接地调用了clerk中的锁。

```
class Consumer implements Runnable { // 消费者
    Clerk clerk;

    public Consumer(Clerk clerk) {
        this.clerk = clerk;
    }

    public void run() {
        System.out.println("消费者开始取走产品");
        while (true) {
            try {
                Thread.sleep((int) Math.random() * 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            clerk.getProduct();
        }
    }
}
```

```
public class ProductTest {
    public static void main(String[] args) {
        Clerk clerk = new Clerk();
        Thread productorThread = new Thread(new Productor(clerk));
        Thread consumerThread = new Thread(new Consumer(clerk));
        productorThread.start();
        consumerThread.start();
    }
}
```

练 习 2

模拟银行取钱的问题

1.定义一个Account类

1）该Account类封装了账户编号（String）和余额（double）两个属性
2）设置相应属性的getter和setter方法
3）提供无参和有两个参数的构造器
4）系统根据账号判断与用户是否匹配，需提供hashCode()和equals()方法的重写

2.提供两个取钱的线程类：小明、小明’s wife

1）提供了Account类的account属性和double类的取款额的属性
2）提供带线程名的构造器
3）run()方法中提供取钱的操作

3.在主类中创建线程进行测试。考虑线程安全问题。

8.5 线程的通信

```
class Account {
private String accountId;
private double balance;
public Account() {
}
public Account(String accountId, double balance)
{
this.accountId = accountId;
this.balance = balance;
}
public String getAccountId() {
return accountId;
}
public void setAccountId(String accountId) {
this.accountId = accountId;
}
public double getBalance() {
return balance;
}
public void setBalance(double balance) {
this.balance = balance;
}
public String toString() {
return "Account [accountId=" + accountId + ",
balance=" + balance + "]";
}
```

```
public int hashCode() {
final int prime = 31;
int result = 1;
result = prime * result + ((accountId == null) ? 0 :
accountId.hashCode());
long temp;
temp = Double.doubleToLongBits(balance);
result = prime * result + (int) (temp ^ (temp >>>
32));
return result;
}
public boolean equals(Object obj) {
if (this == obj)
return true;
if (obj == null)
return false;
if (getClass() != obj.getClass())
return false;
Account other = (Account) obj;
if (accountId == null) {
if (other.accountId != null)
return false;
} else if (!accountId.equals(other.accountId))
return false;
if (Double.doubleToLongBits(balance) !=
Double.doubleToLongBits(other.balance))
return false;
return true;
}
}
```

```
class WithDrawThread extends Thread {
Account account;
// 要取款的额度
double withDraw;
public WithDrawThread(String name, Account account, double amt) {
super(name);
this.account = account;
this.withDraw = amt;
}
public void run() {
synchronized (account) {
if (account.getBalance() > withDraw) {
System.out.println(Thread.currentThread().getName() + ":取款成功，取现的金额为：" + withDraw);
try {
Thread.sleep(50);
} catch (InterruptedException e) {
e.printStackTrace();
}
account.setBalance(account.getBalance() - withDraw);
} else {
System.out.println("取现额度超过账户余额，取款失败");
}
System.out.println("现在账户的余额为：" + account.getBalance());
}
}
}
```

```
public class WithDrawThreadTest {
    public static void main(String[] args) {
        Account account = new Account("1234567", 10000);
        Thread t1 = new WithDrawThread("小明", account, 8000);
        Thread t2 = new WithDrawThread("小明's wife", account, 2800);
        t1.start();
        t2.start();
    }
}
```

8-6 JDK5.0新增线程创建方式

开发中经常使用线程池

将此Callable接口实现类的对象作为传递到FutureTask构造器中，创建FutureTask的对象

```
NumThread numThread = new NumThread();
FutureTask futureTask = new FutureTask(numThread);
```

这其中的继承类调用

```
class MyThread03 implements Callable<Integer>{}
```

新增方式一：实现Callable接口

- 与使用Runnable相比， Callable功能更强大些
  - 相比run()方法，可以有返回值
  - 方法可以抛出异常
  - 支持泛型的返回值
  - 需要借助FutureTask类，比如获取返回结果

重写call方法

新增方式一：实现Callable接口

- Future接口
  - 可以对具体Runnable、Callable任务的执行结果进行取消、查询是否完成、获取结果等。
  - FutrueTask是Futrue接口的唯一的实现类
  - FutureTask 同时实现了Runnable, Future接口。它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值

新增方式二：使用线程池

- **背景：**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。
- **思路：**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。
- **好处：**
  - 提高响应速度（减少了创建新线程的时间）
  - 降低资源消耗（重复利用线程池中线程，不需要每次都创建）
  - 便于线程管理
    - ✓ corePoolSize：核心池的大小
    - ✓ maximumPoolSize：最大线程数
    - ✓ keepAliveTime：线程没有任务时最多保持多长时间后会终止
    - ✓ …

线程池相关API

- JDK 5.0起提供了线程池相关API：**ExecutorService 和 Executors**
- ExecutorService：真正的线程池接口。常见子类ThreadPoolExecutor
  - void execute(Runnable command) ：执行任务/命令，没有返回值，一般用来执行Runnable
  - <T> Future<T> submit(Callable<T> task)：执行任务，有返回值，一般又来执行Callable
  - void shutdown() ：关闭连接池
- Executors：工具类、线程池的工厂类，用于创建并返回不同类型的线程池
  - Executors.newCachedThreadPool()：创建一个可根据需要创建新线程的线程池
  - Executors.newFixedThreadPool(n); 创建一个可重用固定线程数的线程池
  - Executors.newSingleThreadExecutor() ：创建一个只有一个线程的线程池
  - Executors.newScheduledThreadPool(n)：创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

线程池的创建与调用

```
//创建并使用多线程的第四种方法：使用线程池
class MyThread implements Runnable {

	@Override
	public void run() {
		for (int i = 1; i <= 100; i++) {
			System.out.println(Thread.currentThread().getName() + ":" + i);
		}
	}

}

public class ThreadPool {
	public static void main(String[] args) {
		// 1.调用Executors的newFixedThreadPool(),返回指定线程数量的ExecutorService
		ExecutorService pool = Executors.newFixedThreadPool(10);
		// 2.将Runnable实现类的对象作为形参传递给ExecutorService的submit()方法中，
开启线程
		// 并执行相关的run()
		pool.execute(new MyThread());
		pool.execute(new MyThread());
		pool.execute(new MyThread());
		// 3.结束线程的使用
		pool.shutdown();

	}
}
```


尚硅谷
让天下没有难学的的技术