



主讲教师 张 智  
计算机学院软件工程系  
课程群: 421694618

## 9 Java多线程

### 9.1 线程概念

### 9.2 创建线程

### 9.3 线程状态

### 9.4 线程同步

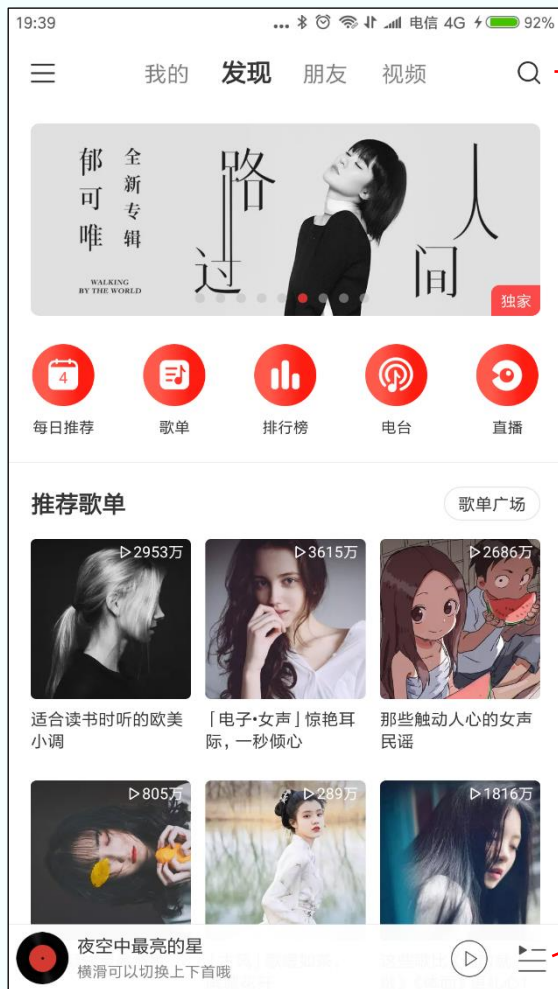
### 9.5 线程通信

## 9.1 线程概念

- **进程**：一个正在运行的程序
- 一个进程内部可以同时执行多个子任务，子任务就称为：**线程**
- 一个进程可以包含一到多个线程（至少一个）
- 线程是操作系统任务调度的最小单位

## 正在运行的网易云音乐：进程

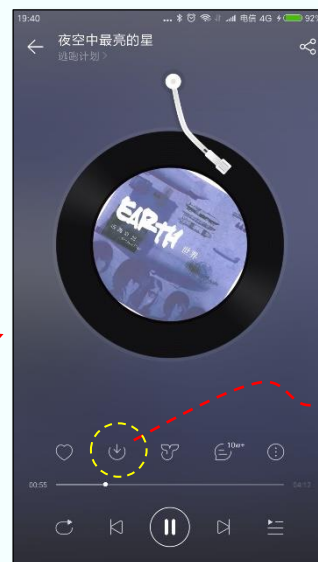
主线程



在线搜索线程

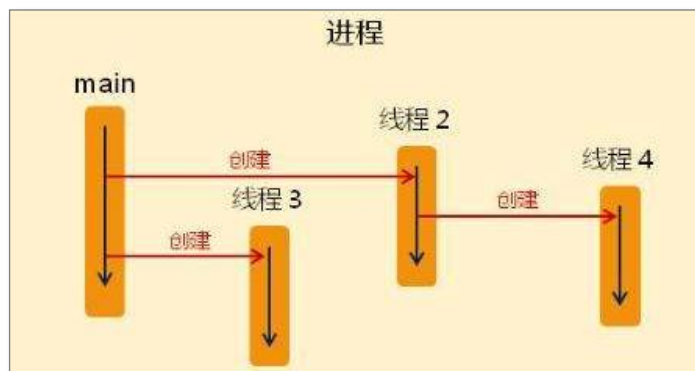


音乐播放线程



音乐下载线程

## Java线程概念

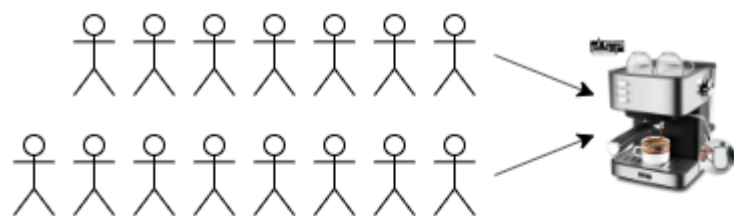


- Java程序入口就是由JVM启动的main主线程
- main线程又可以启动其他线程
- 当所有线程都结束时，JVM退出，进程结束

## 多线程目的

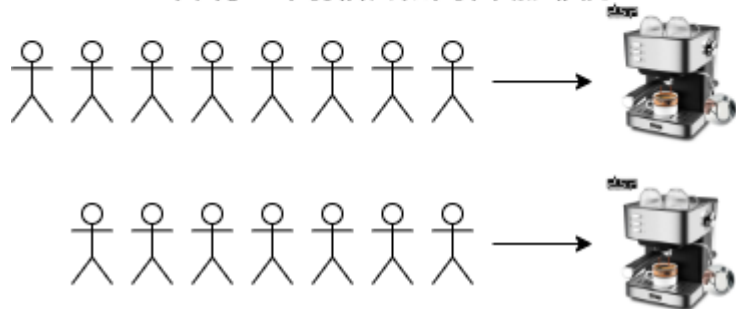
- **多线程目的**：实现多个线程**并发**执行，提高系统利用效率

并发 = 两排队和一台咖啡机



- 并发是一种程序结构技术，有多个控制线程，**是两个或多个事件在同一时间间隔发生**
- 从概念上看，线程同时执行，实际上是多个线程之间不断切换以达到多线程执行任务的目的，当然也可以在多个物理处理器上执行

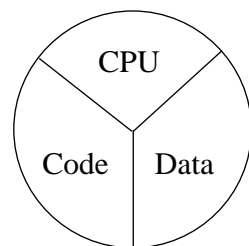
并行 = 两排队和两台咖啡机



- 并行是一种使用多种运算硬件(如多核CPU)，以便更快的执行计算，**是两个或多个事件在同一时间发生**
- 目的是将计算机不同的任务委派给同时执行的不同处理器来处理，以便尽早获得结果

## 线程主要组成

- CPU资源
- CPU执行的代码
- 代码操作的数据



A thread or  
execution context

【[返回](#)】

## 9.2 创建线程

### ■ 创建线程两种常用方法：

- 方法1：继承Thread类
- 方法2：实现Runnable接口（推荐）

【[返回](#)】



## 方法1：继承Thread类

```
class MyThread extends Thread {  
    @Override  
    public void run() {  
        // 执行子线程操作  
    }  
}
```

启动线程：

MyThread t = new MyThread(); //创建线程

t.setName("线程名"); //设置线程名称，非必须

**t.start();** //启动线程,会自动调用run()



匿名链式写法：new MyThread().start();

注意：

- 用start()方法才是启动子线程 (它将隐含调用run()方法)
- 直接调用run()方法不属于子线程启动概念

## 通过继承Thread类创建线程步骤

1. 首先定义一个类去继承Thread父类，然后重写父类中的run()方法，在run()方法中加入具体的任务处理代码
2. 直接新建一个该类对象，也可以利用多态性，变量声明为父类Thread的类型
3. 对象调用start方法，启动线程，隐含调用的是run()方法

## 多线程示例 1

```
class MyThread extends Thread {  
    int count = 0;  
    @Override  
    public void run() {  
        System.out.println("线程名称: " + Thread.currentThread().getName() );  
        while( count < 10 ) {  
            System.out.println("count=" + count);  
            count++;  
            Thread.sleep(500); // 暂停500毫秒(阻塞式)  
        }  
    }  
}
```

**Thread.currentThread():** 获得当前线程 (这里可使用this代替)

获得当前线程名(默认名称如: Thread-0)

需加 try/catch, 此处略

```
public class Test {  
    public static void main(String[] args) {  
        MyThread t = new MyThread();  
        t.start();  
    }  
}
```

运行结果

```
线程名称: Thread-0  
count=0  
count=1  
count=2  
count=3  
count=4  
count=5  
count=6  
count=7  
count=8  
count=9
```

[【返回】](#)

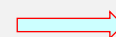
## 方法2：实现Runnable接口

```
class MyThread implements Runnable {  
    @Override  
    public void run() {  
        // 执行子线程操作  
    }  
}
```

启动线程：

```
MyThread myThread = new MyThread();
```

```
Thread t = new Thread( myThread ); //创建线程，Runnable接口实例作为参数  
t.start();
```



匿名链式写法：new Thread( new MyThread() ).start();



设置线程名称：Thread thread = new Thread( myThread, "线程名" );

## 通过实现Runnable接口创建线程步骤

1. 定义一个类实现Runnable接口，然后重写接口中的run()方法，在run()方法中加入具体的任务处理代码
2. 创建一个Runnable接口实现类的对象
3. 创建一个Thread类的对象，需要封装该Runnable接口实现类对象
4. Thread对象调用start()方法，启动线程，隐含调用的是run()方法

```
class MyThread implements Runnable {  
    int total = 10;  
    @Override  
    public void run() {  
        while ( total > 0 ) {  
            System.out.println( Thread.currentThread().getName() + ": 售出了第" + total + "票");  
            Thread.sleep(500); // 需加 try/catch, 此处略  
            total--;  
        }  
    }  
}
```

注：此处不能使用this代替Thread.currentThread()



```
public class Test {  
    public static void main(String[] args) {  
        MyThread myThread = new MyThread();  
        Thread thread = new Thread( myThread, "窗口1" );  
        thread.start();  
    }  
}
```

自定义线程名

#### 运行结果

窗口1: 售出了第10票  
窗口1: 售出了第9票  
窗口1: 售出了第8票  
窗口1: 售出了第7票  
窗口1: 售出了第6票  
窗口1: 售出了第5票  
窗口1: 售出了第4票  
窗口1: 售出了第3票  
窗口1: 售出了第2票  
窗口1: 售出了第1票

进一步链式简写：**Runnable线程匿名类实现** (对象也匿名了)

推荐使用



```
new Thread( new Runnable() {  
    @Override  
    public void run() {  
        //执行耗时操作等  
    }  
}).start();
```

## 两种方法比较

### ■ 继承Thread类方式：

- 优点：编写简单，可直接使用this获得当前线程(无需Thread.currentThread()方法)
- 缺点：因为线程类已经继承了Thread类，所以不能再继承其他类

### ■ 实现Runnable接口方式：

- 优点：多个线程可共享同一个目标对象(9.4节)，适合多个线程来处理同一资源；还可以继承其他类；还可以使用匿名类实现简化代码结构
- 缺点：编程稍微复杂，如果需要访问当前线程，必须使用Thread.currentThread()方法



## 重要说明：

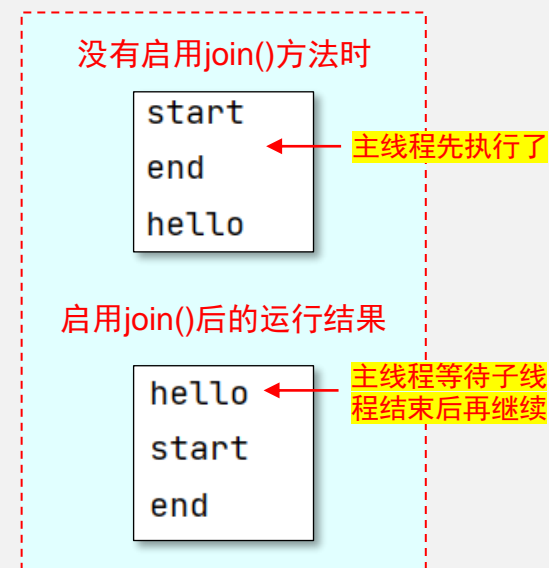
- 当多个线程同时运行时，线程的调度由操作系统决定，程序本身无法决定
- 调用子线程`join()`方法：会让"主线程"等待"子线程"结束之后才能继续运行

不仅仅指main主线程，  
应该理解为调用者线程

理解：比如在线程B中调用了线程A的`join()`方法，则直到线程A执行完毕后，才会继续执行线程B

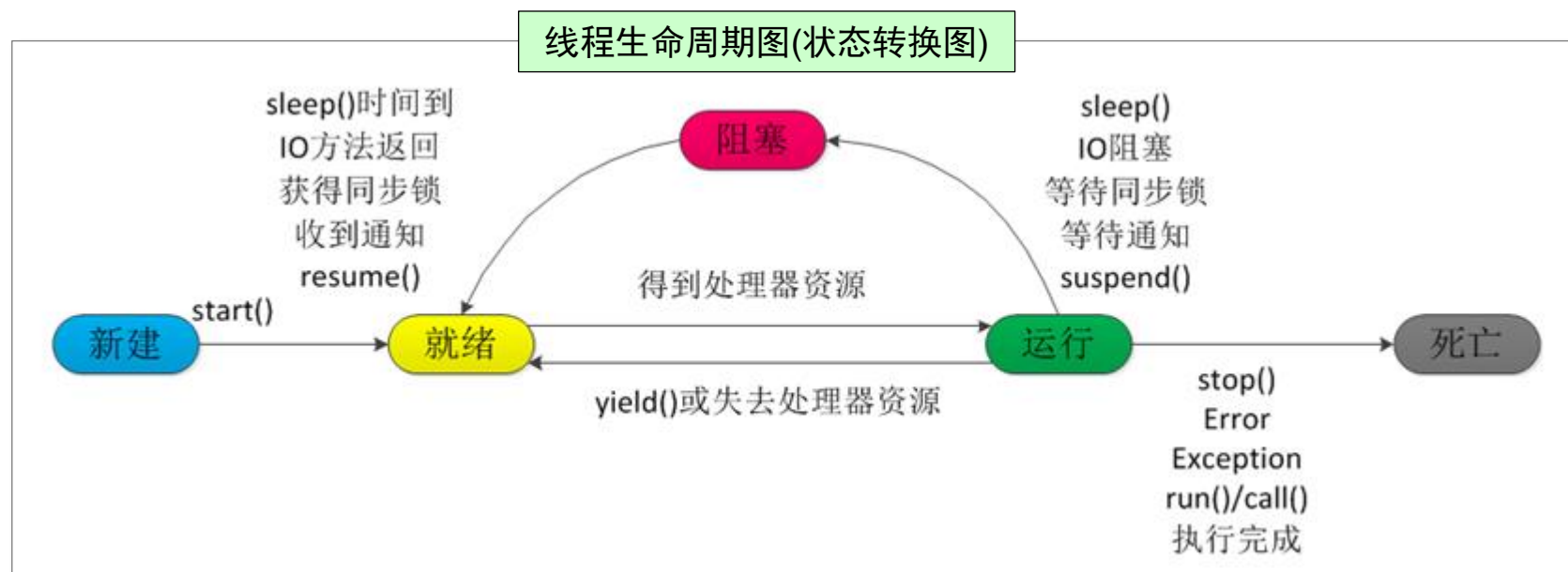
## join()方法示例

```
public class Test {  
    public static void main(String[] args) throws InterruptedException {  
        Thread thread = new Thread(new Runnable() {  
            @Override  
            public void run() {  
                System.out.println("hello");  
            }  
        });  
        thread.start();    //启动子线程  
        // thread.join();  //调用子线程的join()方法: 主线程将等待子线程结束后再继续 (抛出异常)  
        System.out.println("start");  
        System.out.println("end");  
    }  
}
```



【返回】

## 9.3 线程状态



- 线程的生命周期分为创建，就绪，运行，阻塞和死亡五个状态
- 当start()时候就是创建一个线程，当线程所有资源都准备就绪，就差cpu执行的时候那么就进入就绪队列等待
- 当获得cpu时间片的时候就处于运行状态，倘若线程时间片用完而任务没有完成的话就会再次进入就绪队列排队等待，线程任务完成之后就会被杀掉
- 在运行过程中被挂起就会进入阻塞队列，阻塞队列中的线程被唤醒后就会进入就绪队列继续等待cpu时间片

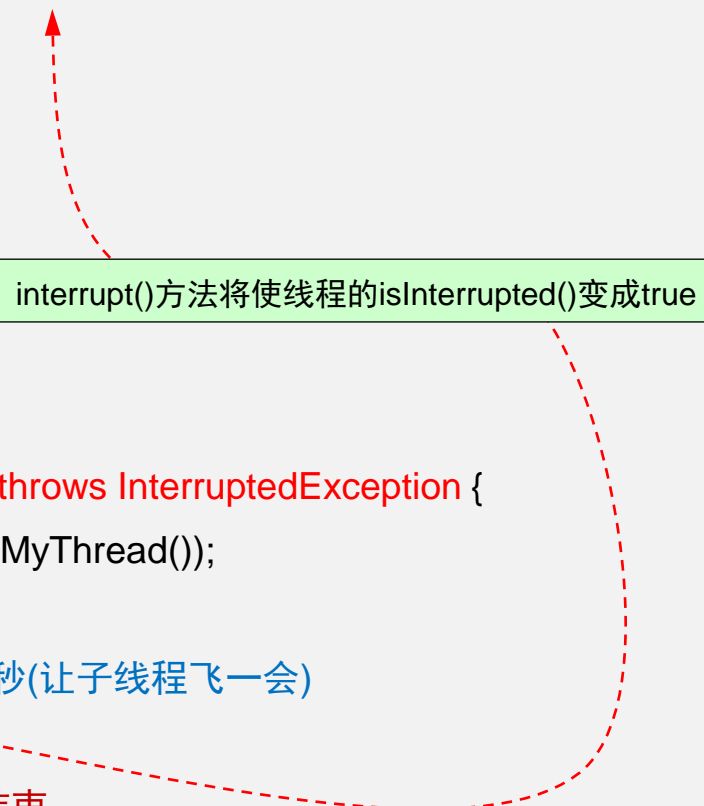
## 线程终止：

- **线程正常终止：** run()方法执行完或有return语句返回
- **线程意外终止：** run()方法因为异常导致线程终止
- **线程强制终止：** 线程调用stop()方法（强烈不推荐）
- **中断线程：** 其他线程给该线程发一个信号，该线程收到信号后，调用interrupt()方法结束执行run()方法，使得自身线程能立刻结束运行

## 中断线程示例

```
class MyThread implements Runnable {  
    @Override  
    public void run() {  
        int n = 0;  
        while ( ! Thread.currentThread().isInterrupted() ) {  
            n++;  
            System.out.println(n);  
        }  
    }  
}  
  
public class Test {  
    public static void main(String[] args) throws InterruptedException {  
        Thread thread = new Thread(new MyThread());  
        thread.start();        // 启动线程  
        Thread.sleep(10);      // 暂停10毫秒(让子线程飞一会)  
        thread.interrupt();     // 中断线程  
        thread.join();          // 等待线程结束  
        System.out.println("end");  
    }  
}
```

interrupt()方法将使线程的isInterrupted()变成true



## 关于守护线程（Daemon Thread）

- 如果某个线程一直不结束，会导致JVM进程无法结束。
- 将线程设置为守护线程：调用`setDaemon(true)`方法。
- 这样JVM进程退出时，不必关心守护线程是否已结束。

```
class MyThread implements Runnable {  
    @Override  
    public void run() {  
        int n = 0;  
        while ( true ) {  
            n++;  
            System.out.println(n);  
        }  
    }  
}  
  
public class Test {  
    public static void main(String[] args) throws InterruptedException {  
        Thread thread = new Thread(new MyThread());  
        thread.setDaemon(true);    //设置为守护线程, JVM就能正常结束  
        thread.start();  
        System.out.println("end");  
    }  
}
```

如没设置为守护线程, 则死循环会导致JVM进程一直不结束

[【返回】](#)

## 9.4 线程同步

- 问题产生：当多个线程同时操作一个**共享的资源变量**时，可能将会导致数据不一致，因此需要一种机制，**确保每个线程看到一致的数据**
- **线程同步**：当有一个线程在对内存进行操作时，其他线程都不可以对这个内存进行操作，直到该线程完成操作
- **线程同步目的**：就是避免线程“同步”执行



## 线程同步问题示例

```
public class Station implements Runnable {
    int ticket = 10;    //线程内部共享的资源(加static也不行)
    public void run() {
        while (true) {
            if (ticket > 0) {
                System.out.println(Thread.currentThread().getName()
                    + "卖出了第" + ticket + "张票");
                ticket--;
            } else if (ticket == 0) {
                System.out.println("票卖完了");
                break;
            }
            try {
                Thread.sleep(1000);    // 休息1秒，用于模拟数据库存储
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

```
class Test1 {
    public static void main(String[] args)
        throws InterruptedException {
        Station station = new Station();
        Thread t1 = new Thread(station, "窗口1");
        Thread t2 = new Thread(station, "窗口2");
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("end");
    }
}
```

线程共享  
内部资源

一个运行结果

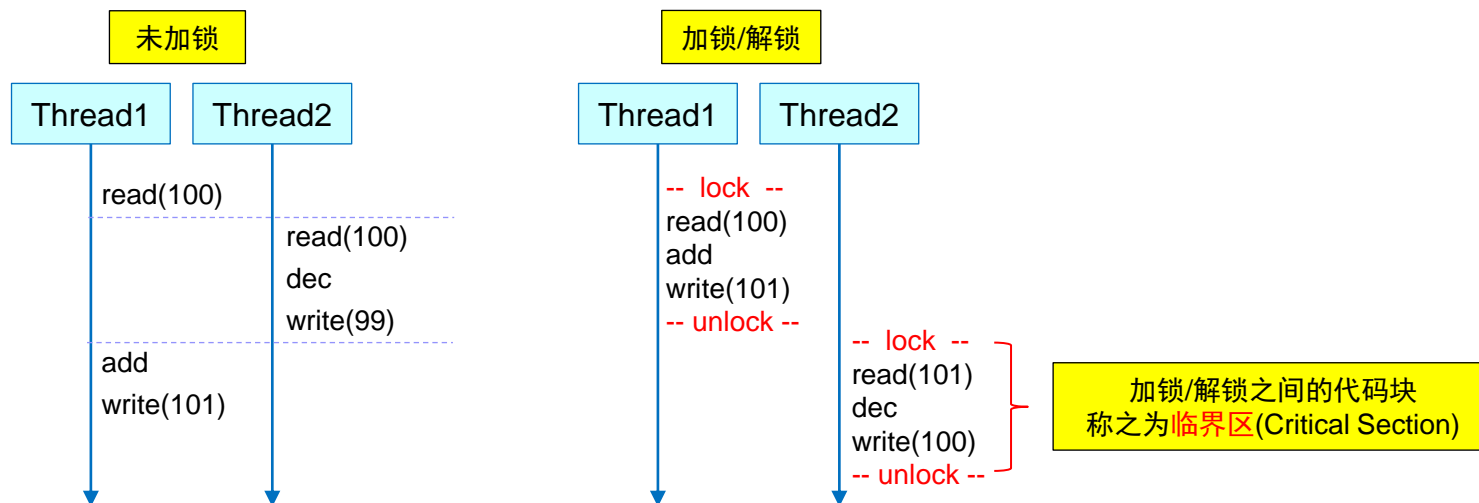
窗口2卖出了第10张票  
窗口1卖出了第10张票  
窗口1卖出了第8张票  
窗口2卖出了第8张票  
窗口1卖出了第6张票  
窗口2卖出了第5张票  
窗口1卖出了第4张票  
窗口2卖出了第3张票  
窗口1卖出了第2张票  
窗口2卖出了第1张票  
票卖完了  
票卖完了  
end

产生问题：数据不一致

## 同步基本策略

注：同步是一种高开销的操作，因此应该尽量减少同步的内容

- 在多线程模型下，要保证逻辑正确，对共享变量进行读写时，必须保证一组指令以原子方式执行：**即某一个线程执行时，其他线程必须等待（加锁和解锁）**



## synchronized关键字：实现加锁

- **synchronized代码块**：表明该代码块只能有一个线程能执行，其他线程必须等待它执行完以后才能执行
- **synchronized方法**：非静态方法和静态方法2种情况。

synchronized本质是一种**独占锁**，即某一时刻仅能有一个线程进入临界区，其他线程必须等待，处于阻塞状态(block)

## (1) synchronized代码块

### ■ 三种用法形式:

- 用法1: `Object lock = new Object();` //先定义锁对象

`synchronized(lock) { ... }` ← 同步代码块

- 用法2: `synchronized(this) { ... }`

- 用法3: `synchronized(类名.class) { ... }`

类名.class: 用于获取Class对象(涉及反射知识)

```
示例: Class claz = Student.class;  
System.out.println("claz的类名(包含包名): " + claz.getName() ); //edu.wust.examples.Student
```

# synchronized代码块示例

解决前例同步问题

```
public class Station implements Runnable {
```

```
    Object lock = new Object();
```

创建对象锁

```
    int ticket = 10; //线程内部共享的资源
```

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

```
        while (true) {
```

加锁

```
            synchronized (lock) { //给资源操作上锁
```

```
                if (ticket > 0) {
```

```
                    System.out.println(Thread.currentThread().getName() + "卖出了第" + ticket + "张票");
```

```
                    ticket--;
```

```
                } else if (ticket == 0) {
```

```
                    System.out.println("票卖完了");    break;
```

```
                }
```

```
            } //释放锁（不要把Thread.sleep锁了）
```

```
            Thread.sleep(1000); // try/catch略
```

```
        }
```

```
    }
```

```
}
```

- 或用this对象锁: synchronized (this) {...}
- 或用类锁: synchronized (Station.class) {...}

测试代码同前

数据一致

## 一个运行结果

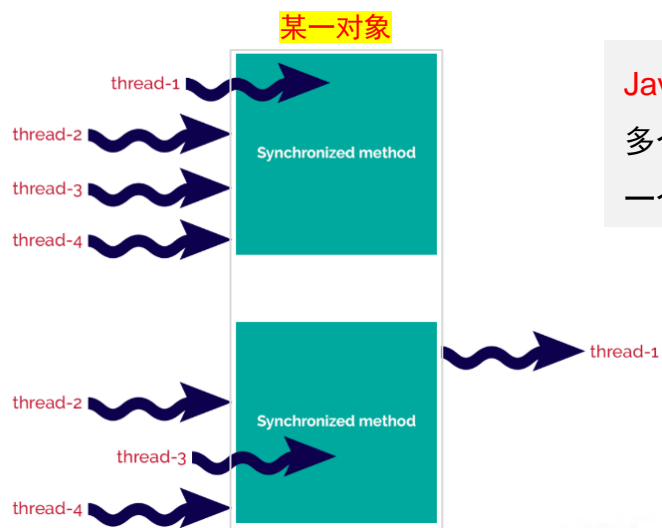
窗口1卖出了第10张票  
窗口2卖出了第9张票  
窗口2卖出了第8张票  
窗口1卖出了第7张票  
窗口1卖出了第6张票  
窗口2卖出了第5张票  
窗口1卖出了第4张票  
窗口2卖出了第3张票  
窗口1卖出了第2张票  
窗口2卖出了第1张票  
票卖完了  
票卖完了  
end

## (2) synchronized方法：实例方法

### ■ synchronized实例方法(非静态方法)：实质是对调用该方法的对象加锁(“对象锁”)

- synchronized实例方法锁住的是当前对象实例(同一个对象只有一把锁)，因此当一个线程正在访问一个对象的synchronized实例方法时，那么其他线程不能访问该对象的其他synchronized方法。

### ■ 如何理解：



Java中每个对象都有一个锁，并且是唯一的。假设分配的一个对象空间，里面有多方法，相当于空间里面有多小房间，如果我们把所有的小房间都加锁(同步方法)，因为这个对象只有一把钥匙，因此同一时间只能有一个人(线程)打开一个小房间，然后用完了还回去，再由JVM去分配下一个获得钥匙的人(线程)。

## synchronized实例方法

- **情况1：同一个对象在两个线程中分别访问该对象的两个同步方法**
  - **结果：会产生互斥。**
  - **解释：**因为锁针对的是对象，当对象调用一个synchronized方法时，其他同步方法需要等待其执行结束并释放锁后才能执行。
- **情况2：不同对象在两个线程中调用同一个同步方法**
  - **结果：不会产生互斥。**
  - **解释：**因为是两个对象，锁针对的是对象，并不是方法，所以可以并发执行，不会互斥。

# synchronized实例方法示例:

```
public class SynchronizedTest {
    public synchronized void method1() { ← 同步实例方法
        System.out.println("Method 1 start");
        try {
            System.out.println("Method 1 execute");
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Method 1 end");
    }

    public synchronized void method2() { ← 同步实例方法
        System.out.println("Method 2 start");
        try {
            System.out.println("Method 2 execute");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Method 2 end");
    }
}
```

```
class Test2 {
    public static void main(String[] args)
        throws InterruptedException {
        // 同一对象实例
        SynchronizedTest test1 = new SynchronizedTest();

        new Thread(new Runnable() {
            @Override
            public void run() {
                test1.method1();
            }
        }).start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                test1.method2();
            }
        }).start();
    }
}
```

启动两个子线程，分别执行  
同一对象的不同同步实例方法

结果：会产生互斥  
即：method2等待method1执行完之后再执行

运行结果

Method 1 start	method1
Method 1 execute	
Method 1 end	
Method 2 start	method2
Method 2 execute	
Method 2 end	

换成test2.method1();会如何

情况2测试：如果定义两个不同对象调用同一个同步方法 (不会互斥)



## (2) synchronized方法：静态方法

- synchronized静态方法：实质是对该类所有对象加锁(“类锁”)
  - synchronized静态方法锁住的是这个类的所有对象(类锁)。
  - synchronized静态方法的锁是类的Class对象(类名.class)，每个类只有一个Class对象，所以每个类只有一个类锁，因此只要是静态方法上的锁，它至始至终只有1个。
  - 静态方法加锁，能和所有其他加锁的静态方法进行互斥。

## synchronized静态方法

- **情况1：用类直接在两个线程中调用两个不同的静态同步方法（类名.静态方法）**
  - **结果：会产生互斥。**
  - **解释：类锁只有一个，相当于N个房间(同步方法)，一把锁，因此房间之间一定是互斥的。**
- **情况2：不同对象在两个线程中调用相同或不同的静态同步方法**
  - **结果：会产生互斥。**
  - **解释：虽然是多个对象，但类锁只有一个。**
- **情况3：同一对象在两个线程中分别调用一个静态同步方法和一个非静态同步方法（注意）**
  - **结果：不会产生互斥。**
  - **解释：虽是同一对象调用，但是两个方法的锁类型不同(一个类锁，一个对象锁)。因此会并发执行，不会互斥。**

# synchronized静态方法示例

```
public class SynchronizedTest {
    public synchronized static void method1() {
        System.out.println("Method 1 start");
        try {
            System.out.println("Method 1 execute");
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Method 1 end");
    }

    public synchronized static void method2() {
        System.out.println("Method 2 start");
        try {
            System.out.println("Method 2 execute");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Method 2 end");
    }
}
```

同步静态方法

同步静态方法

```
class Test2 {
    public static void main(String[] args)
        throws InterruptedException {

        new Thread(new Runnable() {
            @Override
            public void run() {
                SynchronizedTest.method1();
            }
        }).start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                SynchronizedTest.method2();
            }
        }).start();
    }
}
```

启动两个子线程  
分别执行不同的  
同步静态方法

注：method2等待method1执行完之后再执行

运行结果

Method 1 start	method1
Method 1 execute	
Method 1 end	
Method 2 start	method2
Method 2 execute	
Method 2 end	

情况2测试：如果定义两个不同对象，结果会如何？（仍然会互斥）

情况3测试：同一对象调用静态和非静态同步方法，结果会如何？（不会互斥）

## 线程同步示例

- 共享的静态资源：

```
public class Resouce {  
    public static int count = 0;  
}
```

- 创建2个线程来访问 Resouce.count

## 没有同步情况:

```
public class Resouce {    //共享的资源
    public static int count = 0;
}
```

```
class AddThread implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            Resouce.count++;
        }
    }
}
```

```
class DecThread implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            Resouce.count--;
        }
    }
}
```

```
class Test3 {
    public static void main(String[] args)
        throws InterruptedException {
        Thread t1 = new Thread(new AddThread());
        Thread t2 = new Thread(new DecThread());
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println( Resouce.count );
        System.out.println("end");
    }
}
```

让线程运行完毕

表面上看：一个加10000次，一个减10000次，最后结果应该是0  
结果产生问题：每次运行，结果实际上都是不一样的

一个运行的结果

```
-3279
end
```

```
class Resource { //共享的资源
    public static int count = 0;
    public static final Object lock = new Object(); ← 创建静态对象锁
}

class AddThread implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) { ← 锁不用加载循环上
            synchronized (Resource.lock) {
                Resouce.count++;
            } //释放锁
        }
    }
}

class DecThread implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            synchronized (Resource.lock) {
                Resouce.count--;
            } //释放锁
        }
    }
}
```

加锁（必须是同一个锁）

思考一下：  
synchronized (this) 是否可行 -- NO (锁是不同的)  
synchronized (Resource.class) 是否可行 -- OK

测试代码同前

运行结果

0  
end

方式2：使用同步静态方法解决

```
class Resource { //共享的资源
    public static int count = 0;
    public synchronized static void add() { count++; }
    public synchronized static void dec() { count--; }
}

class AddThread implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            Resource.add();
        }
    }
}

class DecThread implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            Resource.dec();
        }
    }
}
```

同步静态方法

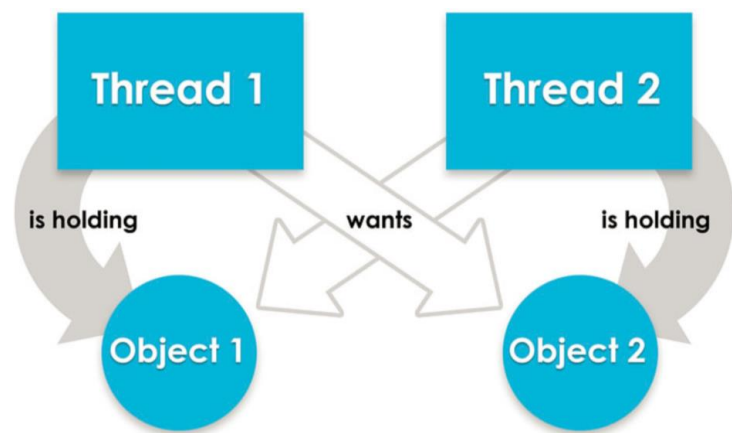
测试代码同前

运行结果

0  
end

## 关于死锁

- **死锁**：是指两个或两个以上的线程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。



线程在获得了一个锁L1，并且没有释放锁L1的情况下，又去申请获得锁L2(锁L1想要包含锁L2)，这个是产生死锁的最根本原因



## 死锁示例

```
public void method1( ) {  
    synchronized(lockA) {    // 获得lockA的锁  
        ...  
        synchronized(lockB) { // 获得lockB的锁  
            ...  
        } // 释放lockB的锁  
    }    // 释放lockA的锁  
}  
  
public void method2( ) {  
    synchronized(lockB) {    // 获得lockB的锁  
        ...  
        synchronized(lockA) { // 获得lockA的锁  
            ...  
        } // 释放lockA的锁  
    }    // 释放lockB的锁  
}
```

## 死锁产生的四个必要条件：

- ① **互斥使用**，即当资源被一个线程使用(占有)时，别的线程不能使用
- ② **不可抢占**，资源请求者不能强制从资源占有者手中夺取资源，资源只能由资源占有者主动释放。
- ③ **请求和保持**，即当资源请求者在请求其他的资源的同时保持对原有资源的占有。
- ④ **循环等待**，即存在一个等待队列：P1占有P2的资源，P2占有P3的资源，P3占有P1的资源，这样就形成了一个等待环路。

当上述四个条件**都成立**的时候，便形成死锁。在死锁情况下如果打破上述任何一个条件，便可让死锁消失

[【返回】](#)

## 9.5 线程通信

- synchronized解决了多线程竞争问题，但没解决多线程协调问题 → 线程之间的通信与协作
- 例如：
  - 浏览器显示图片线程displayThread和下载图片线程downloadThread：如果图片还没有下载完，displayThread可以暂停，当downloadThread完成了任务后，再通知displayThread继续执行
- 线程通信基本机制：
  - 等待/通知机制（wait/notify）
  - 当条件不满足时，线程进入等待状态；当条件满足时，线程被唤醒，继续执行任务

释放已持有的this锁

## 示例：模拟一个任务队列

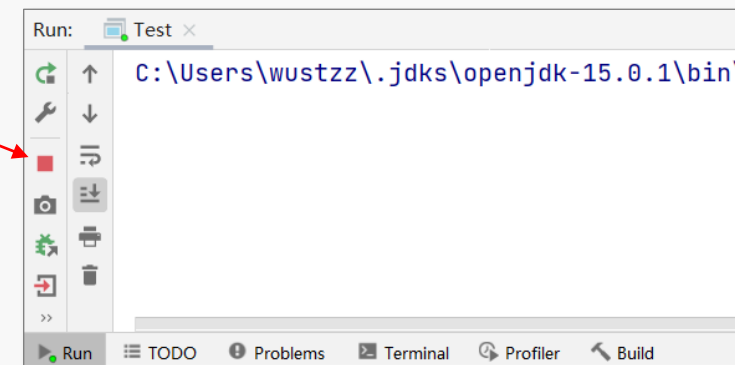
```
public class TaskQueue {  
    Queue<String> queue = new LinkedList<>();  
  
    public synchronized void addTask(String s) { // 往队列里面添加任务  
        queue.offer(s);  
    }  
  
    public synchronized String getTask() { // 从队列里取出任务  
        while ( queue.isEmpty() ) { // 循环等待，直到其他线程往队列中放入任务  
        }  
        return queue.poll();  
    }  
}
```

产生问题：实际上while()循环永远不会退出。因为线程在执行while()循环时，已经在getTask()入口(同步实例方法)获取了this对象锁(指当前某个TaskQueue对象锁)，其他线程根本无法调用这个对象的addTask()

```
class Test4 {  
    public static void main(String[] args){  
        TaskQueue tq = new TaskQueue();  
  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                System.out.println("获得任务: " + tq.getTask() );  
            }  
        }).start();  
  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                tq.addTask("任务1");  
            }  
        }).start();  
    }  
}
```

运行结果是JVM无法结束

需要强制结束



## 等待/通知机制的三个函数

### ■ wait()、notify()、notifyAll()

- 任何对象都有这三个方法（Object继承过来的）

表明已获得的锁

- 这三个方法只能出现在synchronized作用的范围内

- wait()方法：使线程进入等待(阻塞)状态，并释放已持有的this锁

- notify()方法：唤醒一个阻塞队列中的线程进入就绪队列

具体哪个依赖操作系统，有一定的随机性或按优先级最高

- notifyAll()方法：将唤醒所有当前正在this锁等待的线程

已唤醒的线程还需要重新获得锁后才能继续执行

## 前例改进：添加等待/通知机制

```
class TaskQueue {  
    Queue<String> queue = new LinkedList<>();  
  
    public synchronized void addTask(String s) { // 往队列里面添加任务  
        queue.add(s);  
        this.notifyAll(); // 或 notifyAll();  
    }  
  
    public synchronized String getTask() throws InterruptedException { // 从队列里取出任务  
        while ( queue.isEmpty() ) { // 循环等待，直到其他线程往队列中放入任务  
            this.wait(); // 或 wait();  
        }  
        return queue.remove();  
    }  
}
```

当条件满足时，唤醒所有等待的线程

条件不满足时，线程进入等待状态，并释放this锁

```
public class Test {  
    public static void main(String[] args){  
        TaskQueue tq = new TaskQueue();  
  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                try {  
                    System.out.println("获得任务: " + tq.getTask() );  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        }).start();  
  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                tq.addTask("任务1");  
            }  
        }).start();  
    }  
}
```

需要添加try/catch



运行结果





## 示例：生产者-消费者模型（多个情况）

```
public class Test {  
    public static void main(String[] args) {  
        Resource resouce = new Resource();    //共享的资源  
  
        Thread p1 = new Thread(new Producer(resouce), "生产者1");    // 生产者线程  
        Thread p2 = new Thread(new Producer(resouce), "生产者2");    // 生产者线程  
  
        Thread c1 = new Thread(new Consumer(resouce), "消费者1");    // 消费者线程  
        Thread c2 = new Thread(new Consumer(resouce), "消费者2");    // 消费者线程  
        Thread c3 = new Thread(new Consumer(resouce), "消费者3");    // 消费者线程  
  
        p1.start();  
        p2.start();  
  
        c1.start();  
        c2.start();  
        c3.start();  
    }  
}
```

```
class Resource {  
    private final int MAX_SIZE = 10;    // 仓库容量  
    private Queue<Object> list = new ArrayDeque<Object>();    // 仓库存储队列  
  
    public synchronized void produce() throws InterruptedException {  
        while ( list.size() + 1 > MAX_SIZE ) {    // 队列满了  
            System.out.println(Thread.currentThread().getName()+":仓库已满");  
            wait();  
        }  
        list.add( new Object() );    // 生产一个，入队  
        System.out.println(Thread.currentThread().getName() + ":生产一个产品，现库存" + list.size());  
        notifyAll();  
    }  
  
    public synchronized void consume() throws InterruptedException {  
        while ( list.size() == 0 ) {    // 队列空  
            System.out.println(Thread.currentThread().getName()+":仓库为空");  
            wait();  
        }  
        list.remove();    // 消费一个，出队  
        System.out.println(Thread.currentThread().getName()+ ":消费一个产品，现库存" + list.size());  
        notifyAll();  
    }  
}
```

```
class Producer implements Runnable {  
    Resource resource;    //需要定义一个资源成员，并通过构造函数初始化  
  
    public Producer(Resource resource) {  
        this.resource = resource;  
    }  
  
    @Override  
    public void run() {  
        while ( true ) {  
            try {  
                resource.produce();  
                Thread.currentThread().sleep(100);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

```
class Consumer implements Runnable {  
    Resource resource;    //需要定义一个资源成员，并通过构造函数初始化  
  
    public Consumer(Resource resource) {  
        this.resource = resource;  
    }  
  
    @Override  
    public void run() {  
        while ( true ) {  
            try {  
                resource.consume();  
                Thread.currentThread().sleep(300);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

## 运行结果(部分)

```
生产者1:生产一个产品，现库存1  
消费者3:消费一个产品，现库存0  
生产者2:生产一个产品，现库存1  
消费者1:消费一个产品，现库存0  
消费者2:仓库为空  
生产者2:生产一个产品，现库存1  
消费者2:消费一个产品，现库存0  
生产者1:生产一个产品，现库存1  
生产者2:生产一个产品，现库存2  
生产者1:生产一个产品，现库存3  
消费者3:消费一个产品，现库存2  
消费者1:消费一个产品，现库存1  
生产者2:生产一个产品，现库存2  
生产者1:生产一个产品，现库存3
```

【完】