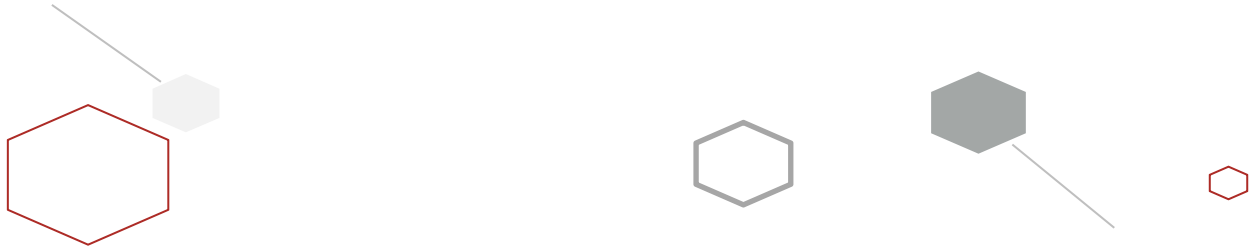


多线程

什么是线程？

- 线程(thread)是一个程序内部的一条执行路径。
- 我们之前启动程序执行后，**main方法的执行**其实就是**一条单独的执行路径**。

```
public static void main(String[] args) {
    // 代码…
    for (int i = 0; i < 10; i++) {
        System.out.println(i);
    }
    // 代码...
}
```

- 程序中如果只有一条执行路径，那么这个程序就是**单线程**的程序。

目录

Contents

多线程的创建

- 方式一：继承Thread类
- 方式二：实现Runnable接口
- 方式三：JDK 5.0新增：实现Callable接口

Thread类

- Java是通过java.lang.Thread 类来代表线程的。
- 按照面向对象的思想，Thread类应该提供了实现多线程的方式。

多线程的实现方案一：继承Thread类

① 定义一个**子类**MyThread**继承线程类**java.lang.Thread，**重写run()方法**

② 创建MyThread类的**对象**

③ 调用线程对象的**start()方法**启动线程（启动后还是执行run方法的）

```
目标：多线程的创建方式一：继承Thread类实现。
public class ThreadDemo1 {
    public static void main(String[] args) {
        // 3、new一个新线程对象
        Thread t = new MyThread();
        // 4、调用start方法启动线程（执行的还是run方法）
        t.start();

        for (int i = 0; i < 5; i++) {
            System.out.println("主线程执行输出：" + i);
        }

    }
}
```

```
子线程执行输出：2
子线程执行输出：3
主线程执行输出：0
子线程执行输出：4
主线程执行输出：1
```

1. 继承Thread类
2. 重写run方法
3. 创建线程对象
4. 调用start()方法启动。

```
1、定义一个线程类继承Thread类
class MyThread extends Thread{
    2、重写run方法，里面是定义线程以后要干啥
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("子线程执行输出：" + i);
        }
    }
}
```

1、为什么不直接调用了run方法，而是调用start启动线程。

- **直接调用run方法会当成普通方法执行，此时相当于还是单线程执行。**
- **只有调用start方法才是启动一个新的线程执行。**

2、把主线程任务放在子线程之前了。

- **这样主线程一直是先跑完的，相当于是一个单线程的效果了。**

1. 方式一是如何实现多线程的？

- 继承Thread类
- 重写run方法
- 创建线程对象
- 调用start()方法启动。

2. 优缺点是什么？

- 优点：编码简单
- 缺点：存在单继承的局限性，线程类继承Thread后，不能继承其他类，不便于扩展。

目录

Contents

多线程的创建

- 方式一：继承Thread类
- 方式二：实现Runnable接口
- 方式三：JDK 5.0新增：实现Callable接口

多线程的实现方案二：实现Runnable接口

① 定义一个线程任务类MyRunnable实现Runnable接口，重写run()方法

② 创建MyRunnable任务对象

③ 把MyRunnable任务对象交给Thread处理。

④ 调用线程对象的start()方法启动线程

Thread的构造方法

构造方法	说明
public Thread(String name)	可以为当前线程指定名称
public Thread(Runnable target)	封装Runnable对象成为线程对象
public Thread(Runnable target ， String name)	封装Runnable对象成为线程对象，并指定线程名称

方式二优缺点：

- 优点：线程任务类只是实现接口，可以继续继承类和实现接口，扩展性强。
- 缺点：编程多一层对象包装，如果线程有执行结果是不可以直接返回的。

```java
public class ThreadDemo2 {
    public static void main(String[] args) {
        // 3、创建一个任务对象
        Runnable target = new MyRunnable();
        // 4、把任务对象交给Thread处理
        Thread t = new Thread(target);
        // 5、启动线程
        t.start();

        for (int i = 0; i < 10; i++) {
            System.out.println("主线程执行输出：" + i);
        }
    }
}
```

1. 定义一个线程任务类MyRunnable实现Runnable接口，重写run()方法
2. 创建MyRunnable对象
3. 把MyRunnable任务对象交给Thread线程对象处理。
4. 调用线程对象的start()方法启动线程

```
"C:\Program Files\J
主线程执行输出：0
子线程执行输出：0
主线程执行输出：1
子线程执行输出：1
```

```java
// 1、定义一个线程任务类 实现Runnable接口
class MyRunnable  implements Runnable {
    // 2、重写run方法，定义线程的执行任务的
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("子线程执行输出：" + i);
        }
    }
}
```

1. 第二种方式是如何创建线程的？

- 定义一个线程任务类MyRunnable实现Runnable接口，重写run()方法
- 创建MyRunnable对象
- 把MyRunnable任务对象交给Thread线程对象处理。
- 调用线程对象的start()方法启动线程

2. 第二种方式的优点

- 优点：线程任务类只是实现了Runnale接口，可以继续继承和实现。
- 缺点：如果线程有执行结果是不能直接返回的。

目录

Contents

多线程的创建

- 方式一：继承Thread类
- 方式二：实现Runnable接口
- 方式三：JDK 5.0新增：实现Callable接口

1、前2种线程创建方式都存在一个问题：

- 他们重写的run方法均不能直接返回结果。
- 不适合需要返回线程执行结果的场景。

2、怎么解决这个问题呢？

- JDK 5.0提供了Callable和FutureTask来实现。
- 这种方式的优点是：可以得到线程执行的结果。

多线程的实现方案三：利用Callable、FutureTask接口实现。

① 、得到任务对象

1. 定义类实现Callable接口，重写call方法，封装要做的事情。

2. 用FutureTask把Callable对象封装成线程任务对象。

② 、把线程任务对象交给Thread处理。

③ 、调用Thread的start方法启动线程，执行任务

④ 、线程执行完毕后、通过FutureTask的get方法去获取任务执行的结果。

目标：学会线程的创建方式三：实现Callable接口，结合FutureTask完成。

```
public class ThreadDemo3 {
    public static void main(String[] args) throws Exec
        // 3、创建Callable对象（表示多线程要执行的任务）
        MyCallable mc = new MyCallable();
        // 4、创建FutureTask 对象（管理多线程运行的结果）
        FutureTask<Integer> f1 = new FutureTask<>(mc);
        // 5、创建线程的对象，交给线程处理
        Thread t1 = new Thread(f1);
        // 6、启动线程
        t1.start();
        //获取多线程运行的结果
        System.out.println(f1.get());
    }
}
```

```
"C:\Progra
5050
```

```
    //1、创建一个类实现Callable接口  应该申明线程任务执行
class MyCallable implements Callable<Integer>{
    //2、重写call方法（有返回值，表示多线程运行的结果）
    @Override
    public Integer call() throws Exception {
        int sum = 0;
        for (int i = 1; i <= 100 ; i++) {
            sum += i;
        }
        return sum;
    }
}
```

FutureTask的API

方法名称	说明
public FutureTask<>(Callable call)	把Callable对象封装成FutureTask对象。
public V get() throws Exception	获取线程执行call方法返回的结果。

方式三优缺点：

- 优点：线程任务类只是实现接口，可以继续继承类和实现接口，扩展性强。
- **可以在线程执行完毕后去获取线程执行的结果。**
- 缺点：编码复杂一点。

3种方式对比

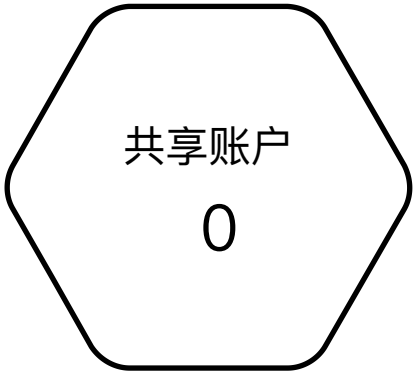
方式	优点	缺点
继承Thread类	编程比较简单，可以直接使用Thread类中的方法	扩展性较差，不能再继承其他的类，不能返回线程执行的结果
实现Runnable接口	扩展性强，实现该接口的同时还可以继承其他的类。	编程相对复杂，不能返回线程执行的结果
实现Callable接口	扩展性强，实现该接口的同时还可以继承其他的类。可以得到线程执行的结果	编程相对复杂

线程同步

- 解决线程安全问题的方案。

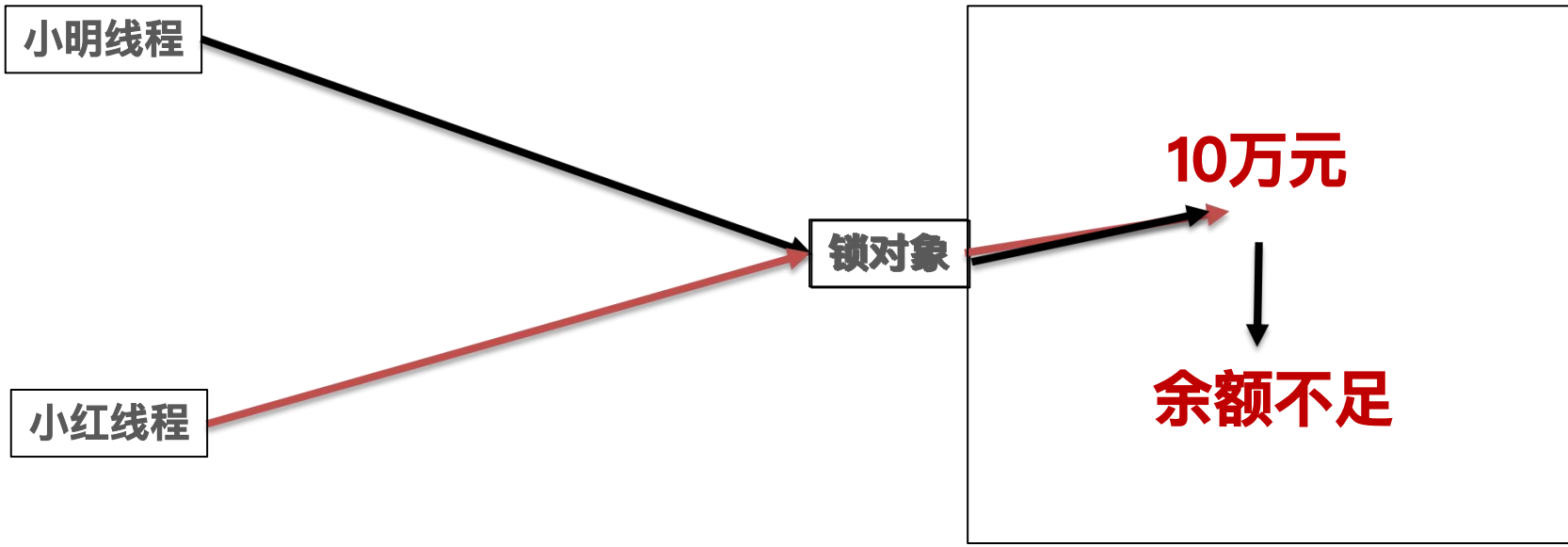
线程同步的思想

- 让多个线程实现先后依次访问共享资源，这样就解决了安全问题。



线程同步的常见方案

- **加锁：**每次只允许一个线程加锁，加锁后才能进入访问，访问完毕后自动解锁，然后其他线程才能再加锁进来。



同步代码块

- 作用：把访问共享资源的核心代码给上锁，以此保证线程安全。

```
synchronized(同步锁) {
        访问共享资源的核心代码
}
```

- 原理：每次只允许一个线程加锁后进入，执行完毕后自动解锁，其他线程才可以进来执行。

同步锁的注意事项

- 对于当前同时执行的线程来说，同步锁必须是同一把（同一个对象），否则会出bug。
  - **对于实例方法建议使用this作为锁对象。**
  - **对于静态方法建议使用字节码（类名.class）对象作为锁对象。**

```
class Ticket implements Runnable {
    private int ticket = 10;
     public void run() {
        for (int i = 0; i < 5; i++) {
            synchronized (this) {  // 同步代码块
                if (ticket > 0) {
                    System.out.println(Thread.currentThread().getName()
                        + " 卖出第 " + ticket + " 张票");
                    ticket--;
                }}}}
}

public class Demo {
    public static void main(String[] args) {
        Ticket t = new Ticket();
        Thread t1 = new Thread(t, "窗口A");
        Thread t2 = new Thread(t, "窗口B");

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

同步方法

- 作用：把访问共享资源的核心方法给上锁，以此保证线程安全。

```
修饰符 synchronized 返回值类型 方法名称(形参列表) {
        操作共享资源的代码
}
```

- 原理：每次只能一个线程进入，执行完毕以后自动解锁，其他线程才可以进来执行

同步方法底层原理

- 同步方法其实底层也是有隐式锁对象的，只是锁的范围是整个方法代码。
- 如果方法是实例方法：同步方法默认用this作为的锁对象。
- 如果方法是静态方法：同步方法默认用类名.class作为的锁对象。

```
class Ticket implements Runnable {
    private int ticket = 10;
 // 使用 synchronized 修饰方法，锁对象是 this（即当前 Ticket 实例）
    public synchronized void sale() {
        if (ticket > 0) {
            System.out.println(Thread.currentThread().getName()
                + " 卖出第 " + ticket + " 张票");
            ticket--;
        }}
    public void run() {
        for (int i = 0; i < 5; i++) {
            sale();
        }}}
public class Demo {
    public static void main(String[] args) {
        Ticket t = new Ticket();
        Thread t1 = new Thread(t, "窗口A");
        Thread t2 = new Thread(t, "窗口B");
        t1.start();
        t2.start();
    }}
```

什么是线程通信?

- 当多个线程共同操作共享的资源时，线程间通过某种方式互相告知自己的状态，以相互协调，并避免无效的资源争夺。

线程通信的常见模型（生产者与消费者模型）

- 生产者线程负责生产数据
- 消费者线程负责消费生产者生产的数据。
- 注意：生产者生产完数据应该等待自己，通知消费者消费；消费者消费完数据也应该等待自己，再通知生产者生产！

使用wait()、notify()和notifyall()协调同步线程

wait()、notify()和notifyAll()都是Object类中的final方法，被所有的类继承，且不允许重写。

wait()方法可以中断线程的执行，使本线程等待，暂时让出CPU的使用权，并允许其他线程使用这个同步方法。其他线程如果在使用这个同步方法时不需要等待，那么它使用完这个同步方法的同时，应当用**notifyAll()**方法通知所有的由于使用这个同步方法而处于等待的线程结束等待。曾中断的线程就会重新排队等待CPU资源，以便从刚才的中断处继续执行这个同步方法

注意：如果使用notify()，那么只是通知处于等待中的线程的某一个结束等待。