15 线程 描述与创建方式

JAVAEE高级

一：程序 - 进程 - 线程 概念

1、程序

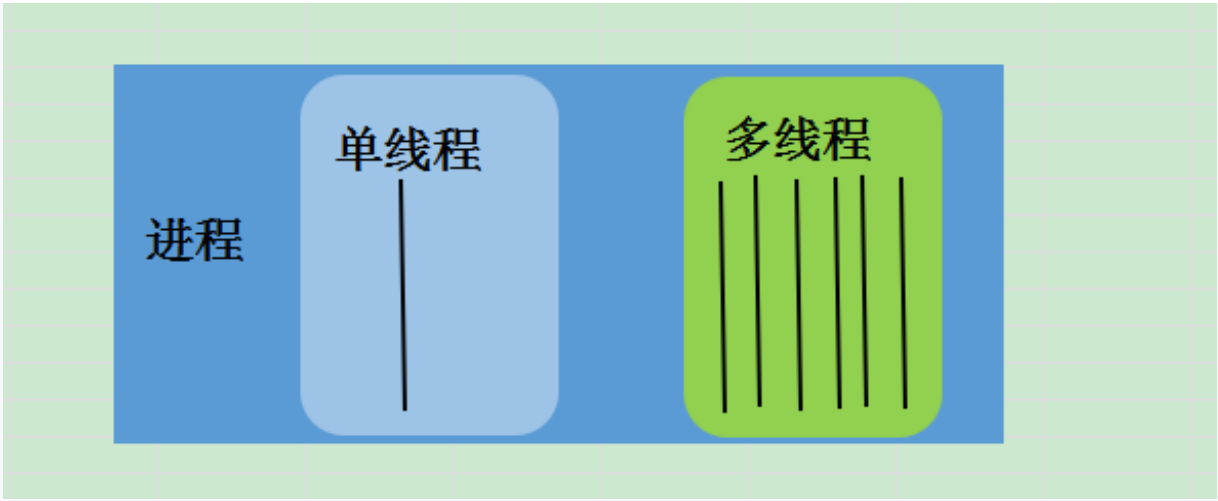
> 1、是为完成特定任务、用某种语言编写的一组指令的集合
>
> 2、即指一段静态的代码，静态对象

2、进程

> 1、是程序的一次执行过程，或是正在运行的一个程序
>
> 2、动态过程：有它自身的产生、存在和消亡的过程
>
> 3、运行中的Eclipse ，输入法等...、程序是静态的，进程是动态的

3、线程

> 1、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
>
> 2、若一个程序可同一时间执行多个线程，就是支持多线程的
>
> 3、每个Java程序都有一个隐含的主线程： main 方法
>
> 4、多线程并发执行可以提高程序的效率, 可以同时完成多项工作



1、多线程的应用场景

1、百度网盘开启多条线程一起下载

2、微信同时和多个人一起视频

3、服务器同时处理多个客户端请求

2、多线程并行和并发的区别

1、并行就是两个任务同时运行，就是甲任务进行的同时，乙任务也在进行。(需要多核CPU)

2、并发是指两个任务都请求运行，而处理器只能按受一个任务，就把这两个任务安排轮流进行，由于时间间隔较短，使人感觉两个任务都在运行。

3、JVM的启动线程

1、JVM启动至少启动了垃圾回收线程和主线程，所以是多线程的

```
public class TestJava {
    @Test
    public void getVoid() {

        for (int i = 0; i < 10; i++) {
            new Finalize();
        }
        for (int i = 0; i < 10; i++) {
```

```
            System.out.println("我是主线程的执行代码");
            System.gc();
        }
    }
}

class Finalize {
    @Override
    public void finalize() {
        System.out.println("垃圾被清扫了");
    }
}
```

二：多线程创建与启动

1、方式一 继承Thread

> 1、定义子类继承Thread类
>
> 2、子类中重写Thread类中的run方法
>
> 3、通过Thread类含参构造器创建线程对象
>
> 4、创建Thread子类对象，即创建了线程对象
>
> 5、调用线程对象start方法：启动线程，调用run方法

```
public class TestJava {
    @Test
    public void getVoid() {
        MyThread mt = new MyThread(); // 创建Thread类的子类对象
        mt.start(); // 开启线程
    }
}
class MyThread extends Thread { // 继承Thread
    public void run() { // 重写run方法
        for (int i = 0; i < 100; i++) { // 将要执行的代码写在run方法中
            System.out.println("asdf");
        }
    }
```

```
}
```

2、方式二 实现Runnable接口(推荐使用)

> 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类的构造方法中
>
> 5、调用Thread类的start方法：开启线程，调用Runnable子类接口的run方法
>
> 6、实现接口的好处
> 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。

```java
public class TestJava {
    @Test
    public void getVoid() {
        MyRunnable mr = new MyRunnable(); //创建Runnable的子类对象
        Thread t = new Thread(mr); //将其当作参数传递给Thread的构造函数
        t.start(); // 开启线程
    }
}

class MyRunnable implements Runnable { //定义一个类实现Runnable
    @Override
    public void run() { // 重写run方法
        for (int i = 0; i < 100; i++) { //将要执行的代码写在run方法中
            System.out.println("qqq");
        }
    }
}
```

①、实现Runnable的原理

> 1、查看源码

> 2、看Thread类的构造函数,传递了Runnable接口的引用

```
//源码展示
 public Thread(Runnable target) {
        init(null, target, "Thread-" + nextThreadNum(), 0);
    }
```

> 3、通过init()方法找到传递的target给成员变量的target赋值

```
//源码展示
    private void init(ThreadGroup g, Runnable target, String name,
                      long stackSize, AccessControlContext acc) {
                ......
                  this.target = target;
                ......
                        }
```

> 4、查看run方法,发现run方法中有判断,如果target不为null就会调用Runnable接口子类对象的run方法

```
//源码展示
    @Override
    public void run() {
        if (target != null) {
            target.run();
        }
    }
```

②、生命周期

> 开启->运行->就绪->休眠->死亡

三：Runnable与Thread类的区别

1、查看源码的区别

> 1、继承Thread :
> 由于子类重写了Thread类的run(),
> 当调用start()时, 直接找子类的run()方法(JVM帮我们完成)
>
> 2、实现Runnable :
> 构造函数中传入了Runnable的引用, 成员变量记住了它
> start()调用run()方法时内部判断成员变量Runnable的引用是否为空
> 不为空编译时看的是Runnable的run(),运行时执行的是子类的run()方法

2、继承Thread

> 1、好处是:
> 可以直接使用Thread类中的方法、代码简单
>
> 2、弊端是:
> 如果已经有了父类、就不能用这种方法因为Java只能单继承

3、实现Runnable接口

> 1、好处是:
> 即使自己定义的线程类有了父类也没关系、因为有了父类也可以实现接口、而且接口是可以多实现的
>
> 2、弊端是:
> 不能直接使用Thread中的方法需要先获取到线程对象后、才能得到Thread的方法、代码复杂

四：匿名内部类实现线程方式

1、继承Thread类

```
1.      @Test
2.      public void getVoid() {
```

```
3.
4.          new Thread() {
5.              public void run() { //重写run方法
6.                  for (int i = 0; i < 100; i++) { // 将要执行的代码写在run方法中
7.                      System.out.println("aaa");
8.                  }
9.              }
10.         }.start(); //开启线程
11.     }
```

2、继承Runnable类

```
1.  @Test
2.      public void getVoid01() {
3.
4.          new Thread(new Runnable() { //将Runnable的子类对象传递给Thread的构造方法
5.              public void run() { //重写run方法
6.                  for (int i = 0; i < 1000; i++) { //将要执行的代码写在run方法中
7.                      System.out.println("ssss");
8.                  }
9.              }
10.         }).start();
11.     }
```