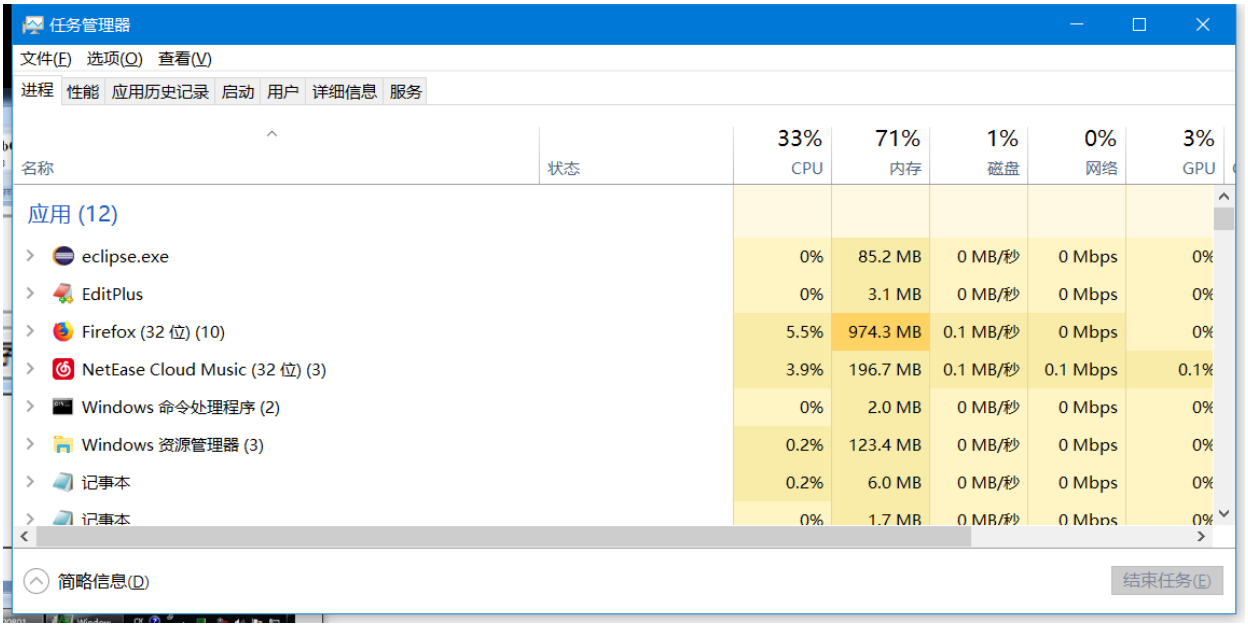
博客： https://www.cnblogs.com/HOsystem/p/14116443.html

2具体内容

在Java语言里面最大的特点是支持多线程的开发（也是为数不多支持多线程的编程语言），所以在整个的java技术的学习里面，如果不能够多多线程的概念有一个全面并且细致的了解，则在日后进行一些项目设计的过程之中尤其是并发访问设计的过程之中就会出现严重的技术缺陷。

如果要想了解线程，那么首先就需要了解一下进程的概念，在传统的DOS系统的时候，其本身有一个特征：如果电脑上出现了病毒，那么所有的程序将无法执行，而单线程处理最大的最大特点：在同一个时间段上只允许一个程序在执行。

那么后来到了Windows的时候就开启了多线程的设计，于是就表示在一个时间段上可以运行多个程序，并且这些程序将进行资源的轮流抢占，所以在同一个时间段上会有多个程序依次执行，但是在同一个时间点上只会有一个进程执行，而后来到了多核的CPU，由于可以处理的CPU多了，那么即便有再多的进程出现，也可以比单核CPU处理的速度有所提升。



线程是在进程基础之上划分的更小的程序单元，线程是在进程基础上创建并且使用的，所以线程依赖于进程的支持，但是线程的启动速度要比进程快许多，所以当使用多线程进行并发处理的时候，其执行的性能要高于进程。

Java是多线程的编程语言，所以Java在进行并发访问处理的时候可以得到更高的处理性能。

2、具体内容

如果要想在Java之中实现多线程的定义，那么就需要有一个专门的线程主体类进行线程的执行任务的定义，而这个主体类的定义是有要求的，必须实现特定的接口或者继承特定的父类才可以完成。

■继承Thread类实现多线程

Java里面提供有一个有java.lang.Thread的程序类，那么一个类只要继承了此类就表示这个类为线程的主体类，但是并不是说这个类就可以直接实现多线程处理了，因为还需要覆

写Thread类中提供的一个run()方法（public void run()），而这个方法就属于线程的主方法。

范例：多线程主体类

```
class MyThread extends Thread {  // 线程的主体类
	private String title ;
	public MyThread(String title) {
		this.title = title ;
	}
	@Override
	public void run() {   // 线程的主体方法
		for (int x = 0 ; x < 10 ; x ++) {
			System.out.println(this.title + "运行，x = " + x);
		}
	}
}
```

多线程要执行的功能都应该在run()方法中进行定义。需要说明的是：在正常情况下如果要想使用一个类中的方法，那么肯定要产生实例化对象，而后去调用类中提供的方法，但是run()方法是不能够被直接调用的，因为这里面牵扯到一个操作系统的资源调度问题，所以要想启动多线程必须使用start()方法完成（public void start()）。

范例：多线程启动

```
public class ThreadDemo {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
			new MyThread("线程A").start();
			new MyThread("线程A").start();
			new MyThread("线程A").start();
	}

}
```

通过此时的调用可以发现，虽然调用了是start()方法，但是最终执行的是run()方法，并且所有的线程对象都是交替执行的。

疑问？为什么多线程的启动不直接使用run()方法而必须使用Thread类中的start()方法呢？

如果要想清楚这个问题，最好的做法是查看一下start()方法的实现操作，可以直接通过源代码观察。

```
   public synchronized void start() {
      if (threadStatus != 0)      //判断线程的状态
         throw new IllegalThreadStateException();  //抛出一个异常
      group.add(this);
      boolean started = false;
      try {
         start0();       //在start()方法里面调用了start0()方法
         started = true;
      } finally {
         try {
```

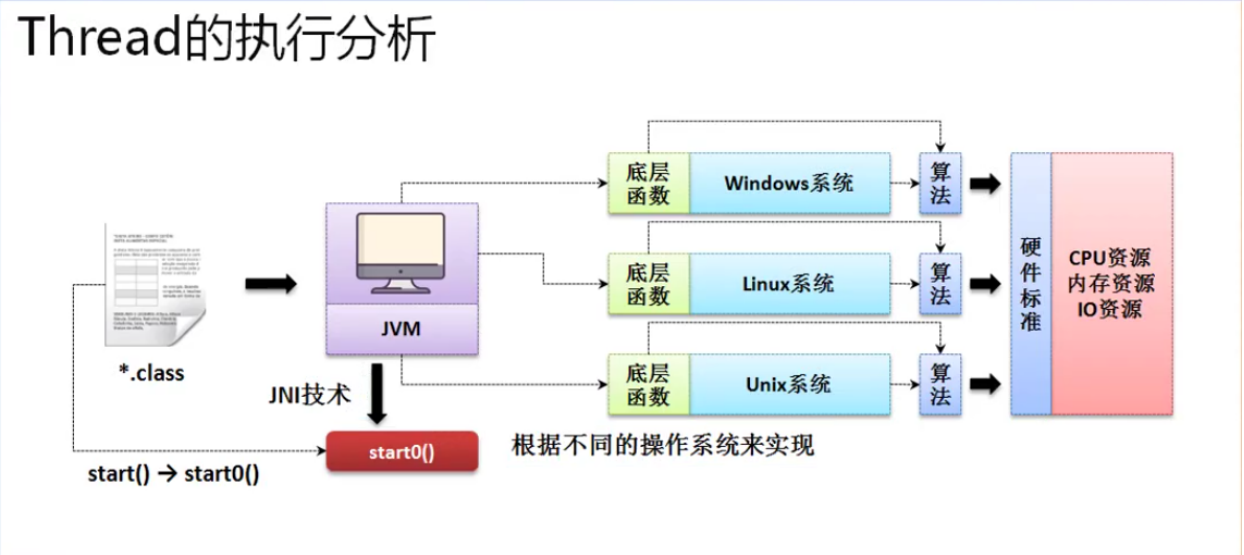
```
            if (!started) {
                group.threadStartFailed(this);
            }
        } catch (Throwable ignore) {
        }
    }
}

private native void start0();//只定义了方法名称，但是没有实现
```

发现在start()方法里面会抛出一个“IllegalThreadStateException”异常类对象，但是整个的程序并没有使用throws或者是明确的try……catch处理，因此该异常一定是RuntimeException的子类，每一个线程类的对象只允许启动一次，如果重复启动则就抛出此异常。例如：下面的代码就会抛出异常。

```
public class ThreadDemo {
    public static void main(String[] args) {
        MyThread mt = new MyThread("线程A") ;
        mt.start() ;
        mt.start() ; // 重复进行了线程的启动
    }
}
```

```
Exception in thread "main"  java.lang.IllegalThreadStateException
```

在Java程序执行的过程之中考虑到对于不同层次开发者的需要，所以其支持有本地的操作系统函数调用，而这项技术就被称为JNI（Java Native Inteface）技术，但是Java开发过程之中并不推荐这样使用，利用这项技术可以使用一些操作系统提供底层函数进行一些特殊的处理，而在Thread类里面提供的start0()就表示需要将此方法依赖于不同的操作系统实现。



任何情况下，只要定义了多线程，多线程的启动永远只有一种方案：Thread类中的start()方法。

■基于Runnable接口实现多线程

虽然可以通过thread类的继承来实现多线程的定义，但是在Java程序里面对于继承永远都是存在有单继承局限的，所以在Java里面又提供有第二种多线程的主体定义结构形式：实现java.lang.Runnable接口，此接口定义如下：

```
@FunctionalInterface
public interface Runnable{     //JDK1.8引入了Lambda表达式之后就变为了函数式接口
    public void run()
}
```

范例：通过Runnable实现多线程的主体类

```
class MyThread implements Runnable {// 线程的主体类
    private String title ;
    public MyThread(String title) {
        this.title = title ;
    }
    @Override
    public void run() {   // 线程的主体方法
        for (int x = 0 ; x < 10 ; x ++) {
            System.out.println(this.title + "运行，x = " + x);
        }
    }
}
```

但是此时由于不在继承Thread父类了，那么对于此时的MyThread类中也就不再支持有start()这个继承的方法，可是如果不使用Thread.start()方法是无法进行多线程启动的，那么这个时候就需要去观察一下Thread类提供的构造方法：

·构造方法：public Thread(Runnable target)；

范例：启动多线程

```
public class ThreadDemo {
    public static void main(String[] args) {
        Thread threadA = new Thread(new MyThread("线程对象A")) ;
        Thread threadB = new Thread(new MyThread("线程对象B")) ;
        Thread threadC = new Thread(new MyThread("线程对象C")) ;
        threadA.start(); // 启动多线程
        threadB.start(); // 启动多线程
        threadC.start(); // 启动多线程
    }
}
```

这个时候的多线程实现里面可以发现，由于只是实现了Runnable接口对象，所以此时线程主体类上就不再有单继承局限了，那么这样的设计才是一个标准型的设计。

可以发现从JDK18开始，Runnable接口使用了函数式接口定义，所以也可以会直接使用lambda表达式进行线程类实现。

范例：利用Lambda实现多线程定义

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

```
        for (int x = 0 ; x < 3 ; x ++) {
            String title = "线程对象-" + x ;
            Runnable run = ()->{
                for (int y = 0 ; y < 10 ; y ++) {
                    System.out.println(title + "运行，y = " + y);
                }
            } ;
            new Thread(run).start();
        }
    }
}
```

```
public class ThreadDemo {
    public static void main(String[] args) {
        for (int x = 0 ; x < 3 ; x ++) {
            String title = "线程对象-" + x ;
            new Thread(()-> {
                for (int y = 0 ; y < 10 ; y ++) {
                    System.out.println(title + "运行，y = " + y);
                }
            }).start();
        }
    }
}
```

在以后的开发之中对于多线程的实现，优先考虑的就是Runnable接口实现，并且永恒都是通过Thread类对象启动多线程。

■Thread与Runnable关系

经过一系列的分析之后可以发现，在多线程的实现过程之中已经有了两种做法：Thread类、Runnable接口，如果从代码的结构本身来讲肯定使用Runnable是最方便的，因为其可以避免单继承的局限，同时也可以更好的进行功能的扩充。

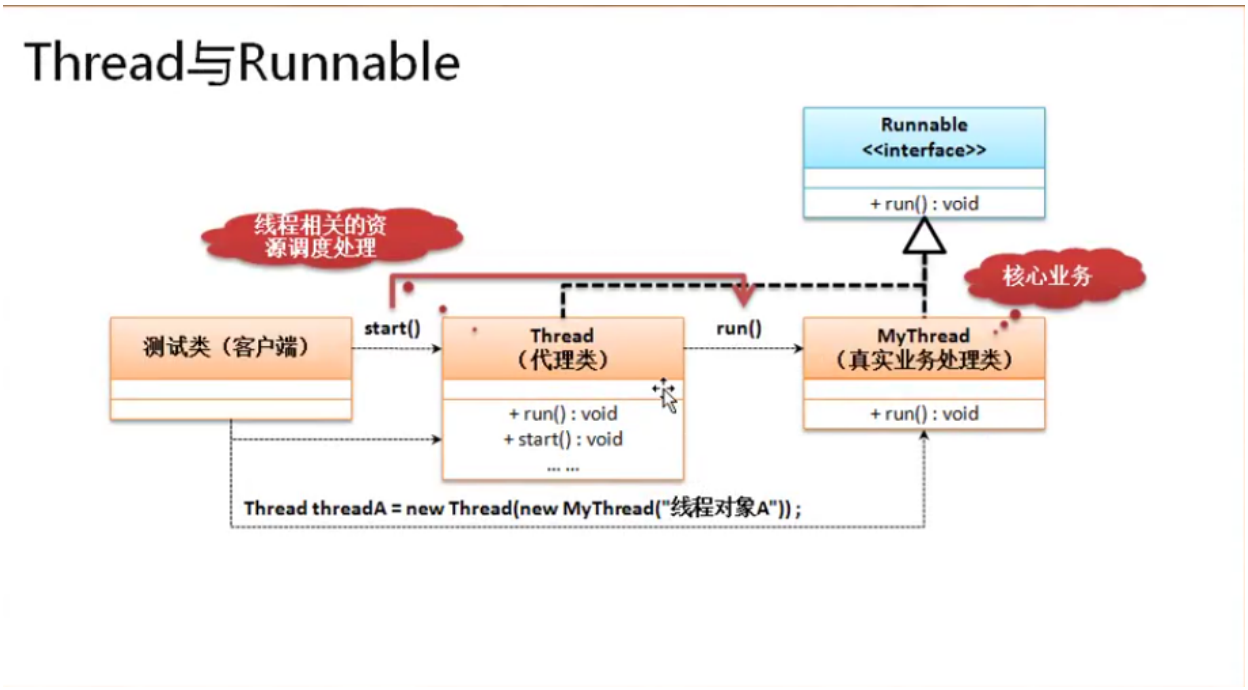
但是从结构上也需要来观察Thread与Runnable的联系，打开Thread类定义：

```
public class Thread extends Object implements Runnable{}
```

发现现在Thread类也是Runnable接口的子类，那么在之前继承Thread类的时候实际上覆写的还是Runnable接口的run()方法，也是此时来观察一下程序的类结构

```
class MyThread implements Runnable {// 线程的主体类
    private String title ;
    public MyThread(String title) {
        this.title = title ;
    }
    @Override
    public void run() {   // 线程的主体方法
        for (int x = 0 ; x < 10 ; x ++) {
            System.out.println(this.title + "运行，x = " + x);
        }
```

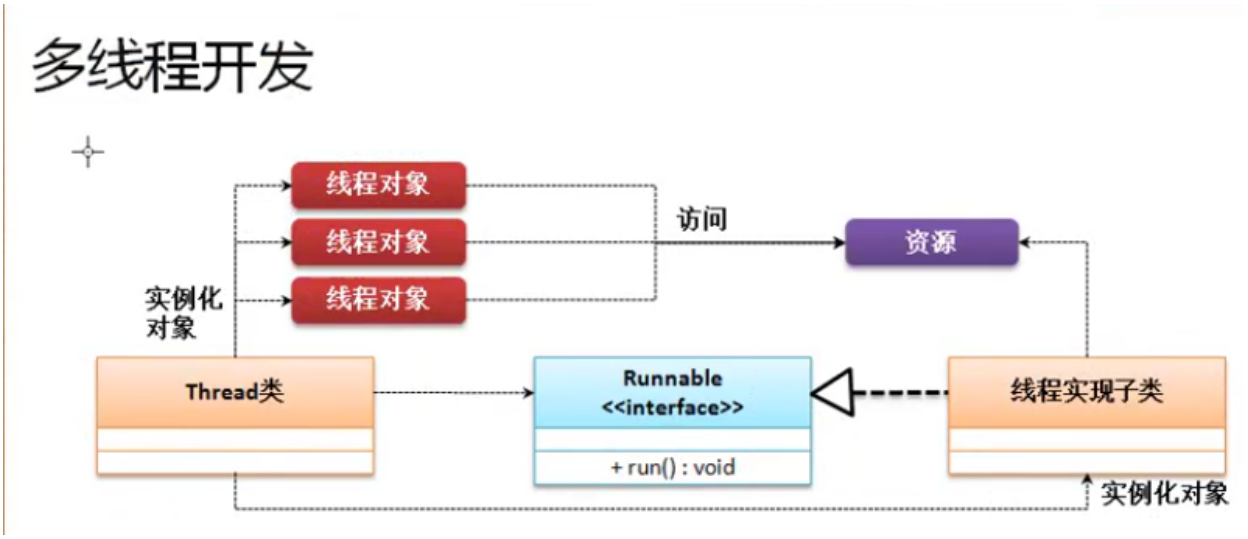
```
        }
}
public class ThreadDemo {
        public static void main(String[] args) {
                Thread threadA = new Thread(new MyThread("线程对象A")) ;
                Thread threadB = new Thread(new MyThread("线程对象B")) ;
                Thread threadC = new Thread(new MyThread("线程对象C")) ;
                threadA.start(); // 启动多线程
                threadB.start(); // 启动多线程
                threadC.start(); // 启动多线程
        }
}
```



多线程的设计之中，使用了代理设计模式的结构，用户自定义的线程主体只是负责项目核心功能的实现，而所有的辅助实现全部交由Thread类来处理。

在进行Thread启动多线程的时候调用的是start()方法，而后找到的是run()方法，当通过Thread类的构造方法传递了一个Runnable接口对象的时候，那么该接口对象将被Thread类中的target属性所保存，在start()方法执行的时候回调用Thread类中的run()方法，而这个run()方法去调用Runnable接口子类被覆写过的run()方法。

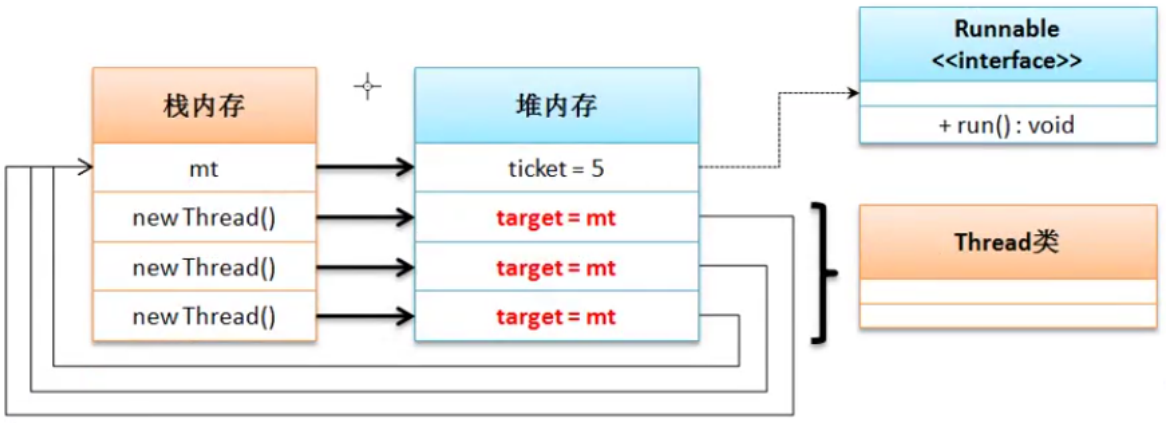
多线程开发的本质上是在于多个线程可以进行同一资源的抢占，那么Thread主要描述的是线程，而资源的描述是通过Runnable完成的。



范例：利用卖票程序来实现多个线程的资源并发访问

```
class MyThread implements Runnable {// 线程的主体类
	private int ticket = 5 ;
	@Override
	public void run() {   // 线程的主体方法
		for (int x = 0 ; x < 100 ; x ++) {
			if (this.ticket > 0) {
				System.out.println("卖票，ticket = " + this.ticket --);
			}
		}
	}
}
public class ThreadDemo {
	public static void main(String[] args) {
		MyThread mt = new MyThread() ;
		new Thread(mt).start();   // 第一个线程启动
		new Thread(mt).start();   // 第二个线程启动
		new Thread(mt).start();   // 第三个线程启动
	}
}
```

通过内存分析图来分析本程序的执行结构。

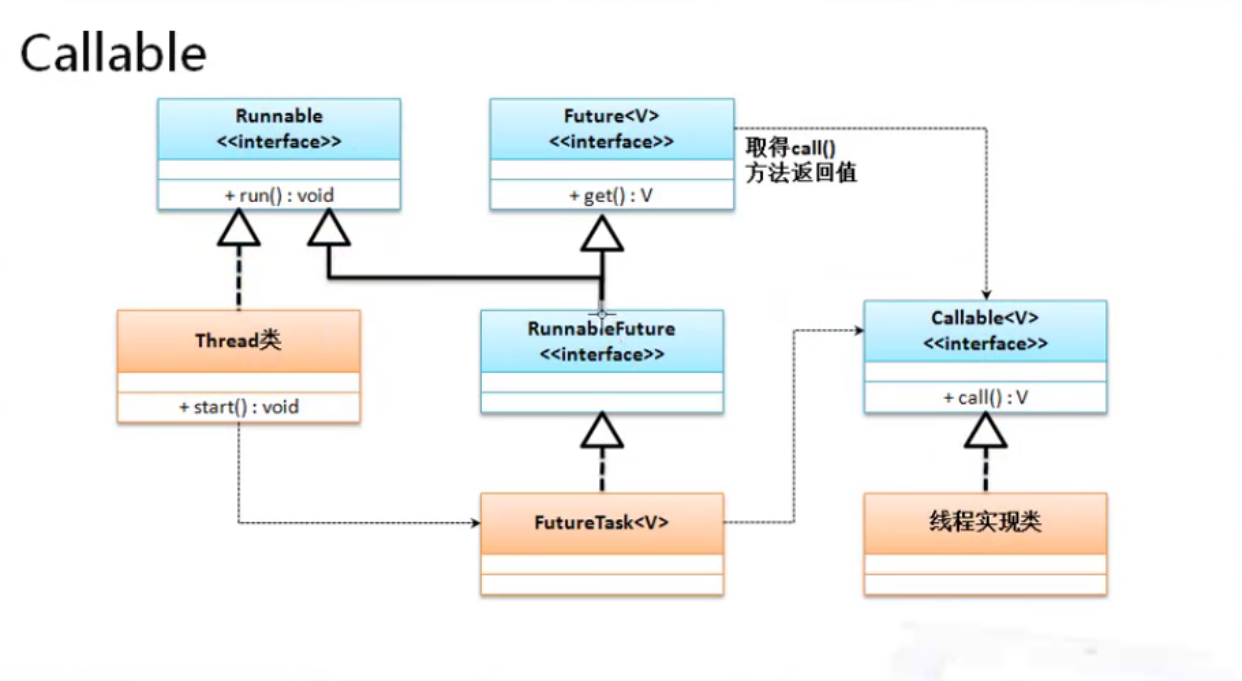


■Callable实现多线程

从最传统的开发来讲如果要进行多线程的实现肯定依靠的就是Runnable，但是Runnable接口有一个缺点：当线程执行完毕之后无法获取一个返回值，所以从JDK1.5之后就提出一个新的线程实现接口：java.util.concurrent.Callable接口，首先来观察这个接口的定义：

```
@FunctionalInterface
public interface Callable<V>{
    public V call() throws Exception;
}
```

可以发现Callable定义的时候可以设置一个泛型，此泛型的类型就是返回数据的类型，这样的好处是可以避免向下转型所带来的安全隐患。



范例：使用Callable实现多线程处理

```
import java.util.concurrent.Callable;
import java.util.concurrent.FutureTask;
class MyThread implements Callable<String> {
    @Override
    public String call() throws Exception {
        for (int x = 0 ; x < 10 ; x ++) {
            System.out.println("********* 线程执行、x = " + x);
        }
        return "线程执行完毕。";
    }
}
public class ThreadDemo {
    public static void main(String[] args) throws Exception {
        FutureTask<String> task = new FutureTask<>(new MyThread()) ;
        new Thread(task).start();
        System.out.println("【线程返回数据】" + task.get());
```

```
    }
}
```

面试题：请解释Runnable与Callable的区别?

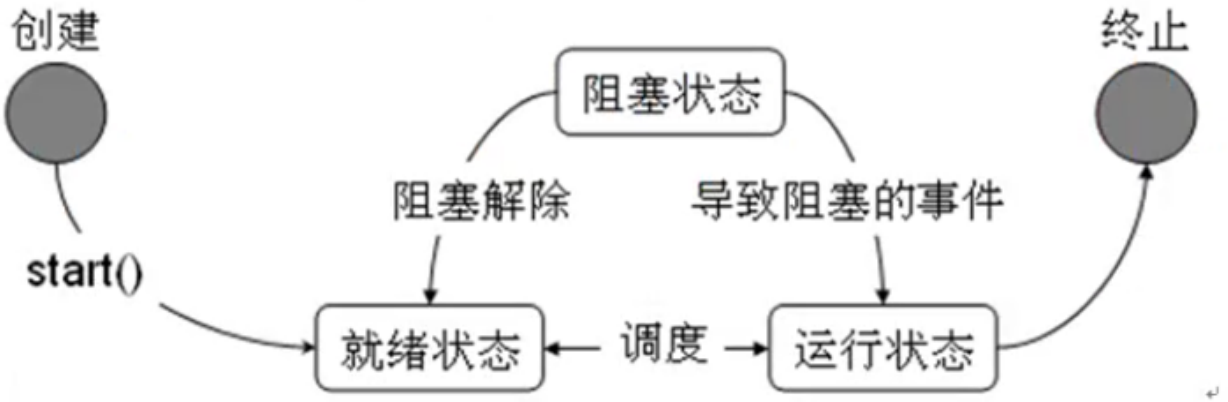
·Runnable是在JDK1.0的时候提出的多线程的实现接口，而Callable是在JDK1.5之后提出的；

·java.lang.Runnable接口之中只提供有一个run()方法，并且没有返回值；

·java.util.concurrent.Callable接口提供有call()方法，可以有返回值；

■线程运行状态

对于多线程的开发而言，编写程序的过程之中总是按照：定义线程主体类，而后通过Thread类进行线程的启动，但是并不意味着你调用了start()方法，线程就已经开始运行了，因为整体的线程处理有自己的一套运行的状态。

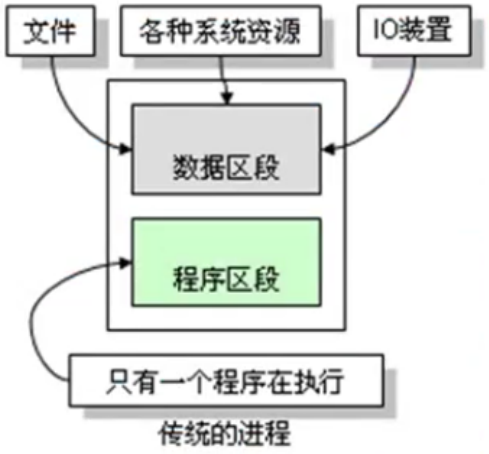


1、任何一个线程的徐爱那个都应该使用Thread类进行封装，所以线程的启动使用的是start()，但是启动的时候实际上若干个线程都将进入到一种就绪状态，现在并没有执行。

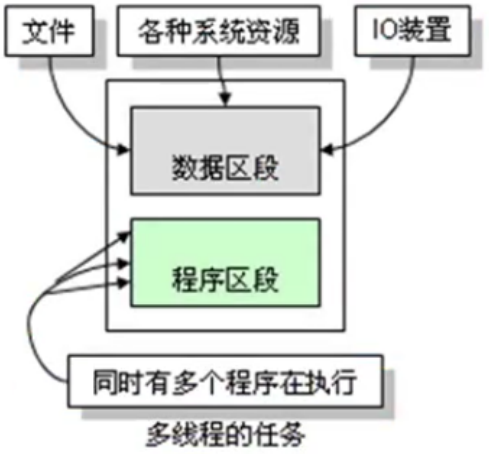
2、进入到就绪状态之后就需要等待进行资源调度，当某一个线程调度成功之后则进入到运行状态（run()方法）,但是所有的线程不可能一直持续执行下去，中间需要产生一些暂停的状态，例如：某个线程执行一段时间之后就需要让出资源，而后这个线程就将进入到阻塞状态，随后重新回归到就绪状态；

3、当run()方法执行完毕之后，实际上该线程的主要任务也就结束了，那么此时就可以直接进入到停止状态；

进程与线程的区别



传统的进程



多线程的任务