线程

2018年7月23日 23:16

如果程序只有一条执行路径，那么该程序为单线程程序
如果程序有多条执行路线，那么该程序为多线程程序

想要了解线程，要先了解进程，因为线程是依赖于进程而存在的

什么是进程?
进程：正在运行的程序。是系统进行资源分配和调用的独立单位，每一个进程都有它自己的内存空间和系统资源

多进程有什么意义?
单进程的计算机只能做一件事情
现代计算机都能同时做多件事情，支持多进程，就可以在一个时间段内执行多个任务
提高CPU的使用率

一边玩游戏，一边听音乐是同时进行的吗?
不是，因为CPU在某一个时间点上只能做一件事情
在玩游戏或听音乐时，CPU在程序间的**高效**切换让我们觉得是同时进行的

什么是线程?
在一个进程内又可以执行多个任务，每一个任务可以看成是一个线程
线程：是程序的执行单元，执行路径。是程序使用CPU的最基本单位

多线程有什么意义?
多线程存在，不是提高程序的执行率。其实是为了提高程序的**使用率**
程序的执行其实都是在抢CPU的资源，CPU的执行权
某一个进程的执行路径（线程）比较多，就会有更高的几率抢到CPU的执行权
不能保证某一个线程能够在某个时刻抢到，所以程序的执行有**随机性**

并行与并发
前者是逻辑上同时发生，指在某一个时间内同时运行多个程序
后者是物理上同时发生，指在某一个时间点同时运行多个程序 高并发

java程序的运行原理

2018年7月26日 14:47

由java命令启动JVM，JVM启动就相当于启动了一个进程
接着由该进程创建了一个主线程去调用main方法

jvm虚拟机的启动是多线程还是多线程?

多线程
原因是垃圾回收线程也要先启动，否则很容易出现内存溢出
最低启用两个线程：主线程+垃圾回收线程

Thread类

2018年7月26日 15:28

```
/*
 * 为什么重写run()方法
 * 不是线程中所有的代码都需要被线程执行的，为了区分哪些代码能够被线程执行，Thread中提供了run方法
 */
public class Mythread extends Thread {
	@Override
	public void run() {
		// 被线程执行的代码一般是比较耗时的
		for(int x =0;x<100;x++) {
			System.out.println(x);
		}
	}
}
```

多线程代码实现

2018年7月26日 15:28

```
/*
 * 进程与线程
 *
 * 举例：扫雷程序、迅雷下载
 * 需求：实现多线程的程序
 * 由于线程是由于进程而存在的，应该先创建一个进程
 * 进程是由系统创建的，java不能直接调用系统功能，java可以调用C/C++写好的程序实现
 * 提供一些类供使用，实现java的多线程
 *
 * Thread  有2种方式实现多线程
 * 1.继承Thread类，重写run()方法，创建对象，启动线程
 */
public class MyThreadDemo {
	public static void main(String[] args) {
		// Mythread my = new Mythread();
		// my.run();
		// my.run();
		/*
		 * 调用run()方法其实就相当于普通方法调用，其实是单线程的 面试题：run()和start()的区别
		 * run():仅仅是封装该线程执行的代码，直接调用是普通方法 start():首先启动了线程，然后再由jvm去调用run()方法
		 */

		// my.start();
		// my.start();
		// IllegalThreadStateException非法的线程状态异常，相当于同一线程被调用两次

		Mythread my1 = new Mythread();
		Mythread my2 = new Mythread();
		my1.start();
		my2.start();
	}
}
```

线程对象名称

2018年7月26日 15:28

```
public class MyThread extends Thread {
	public MyThread() {}
	public MyThread(String name) {
		super(name);
	}

	@Override
	public void run() {
		for (int x = 0; x < 100; x++) {
			System.out.println(getName()+":"+x);
		}
	}
}
```

```
/*
 * 如何获取线程对象的名称
 * public final String getName():获取线程的名称
 * public final void setName(String name)：设置线程的名称
 */
public class MyThreadDemo {
	public static void main(String[] args) {
		// 无参构造+setXxx
		MyThread my1 = new MyThread();
		MyThread my2 = new MyThread();

		my1.setName("王若潇");
		my2.setName("椰子皮");

		my1.start();
		my2.start();

		// 带参构造
		MyThread my3 = new MyThread("屠龙");
		my3.start();

		// 获取main方法所在线程对象的名称
		// 返回当前正在执行的线程对象
		System.out.println(Thread.currentThread().getName());// main
	}
}
```

线程优先级

2018年7月26日 15:54

```
/*
 * 线程优先级
 * 获取线程对象的优先级:public final int getPriority()
 * 线程默认优先级为5
 * 设置线程优先级：public final void setPriority(int newPriority)
 * 最高优先级为10，最低优先级为1
 * 线程优先级仅仅表示线程获取CPU时间片的几率，但是要在多次运行的时候才能看到比较好的效果
 */
public class ThreadPriorityDemo {
	public static void main(String[] args) {
		ThreadPriority tp1 = new ThreadPriority();
		ThreadPriority tp2 = new ThreadPriority();
		ThreadPriority tp3 = new ThreadPriority();

		tp1.setName("王若潇");
		tp2.setName("椰子皮");
		tp3.setName("屠龙");

		System.out.println(tp1.getPriority());// 5
		System.out.println(tp2.getPriority());// 5
		System.out.println(tp3.getPriority());// 5

		tp1.setPriority(10);
		tp2.setPriority(1);

		tp1.start();
		tp2.start();
		tp3.start();
	}
}
```

线程休眠

2018年7月26日 16:11

```
public class ThreadSleep extends Thread {
    @Override
    public void run() {
        for (int x = 0; x < 100; x++) {
            System.out.println(getName() + ":" + x + ",日期" + new Date());
            // 睡眠1秒种
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}
```

```
/*
 * 线程休眠：public static void sleep(long millis)
 */
public class ThreadSleepDemo {
    public static void main(String[] args) {
        ThreadSleep ts1 = new ThreadSleep();
        ThreadSleep ts2 = new ThreadSleep();
        ThreadSleep ts3 = new ThreadSleep();

        ts1.setName("王若潇");
        ts2.setName("椰子皮");
        ts3.setName("屠龙");

        ts1.start();
        ts2.start();
        ts3.start();
    }

}
```

加入线程

2018年7月26日 16:24

```
/*
 * public final void join():等待该线程终止
 */
public class ThreadJoinDemo {
	public static void main(String[] args) throws InterruptedException {
		ThreadJoin tj1 = new ThreadJoin();
		ThreadJoin tj2 = new ThreadJoin();
		ThreadJoin tj3 = new ThreadJoin();

		tj1.setName("a");
		tj2.setName("b");
		tj3.setName("c");

		tj1.start();
		tj1.join();// 该线程执行完毕再执行下面线程
		tj2.start();
		tj3.start();

	}
}
```

礼让线程

2018年7月26日 16:30

```
public class ThreadYield extends Thread {
	@Override
	public void run() {
		for (int x = 0; x < 100; x++) {
			System.out.println(getName() + ":" + x);
			Thread.yield();
		}
	}
}
```

```
/*
 * public static void yield():暂停当前的线程，并执行其它的线程
 * 让多个线程的执行更和谐，但不能保证一人一次
 */
public class ThreadYieldDemo {
	public static void main(String[] args) {
		ThreadYield ty1 = new ThreadYield();
		ThreadYield ty2 = new ThreadYield();

		ty1.setName("a");
		ty2.setName("b");

		ty1.start();
		ty2.start();
	}
}
```

守护线程

2018年7月26日 16:37

```
/*
 * public final void setDaemon(boolean on)将该线程标记为守护线程或用户线程
 * 当运行的线程都是守护线程时，java虚拟机退出
 * 该方法在启动前调用
 */
public class ThreadDaemonDemo {
	public static void main(String[] args) {
		ThreadDaemon td1 = new ThreadDaemon();
		ThreadDaemon td2 = new ThreadDaemon();

		td1.setName("关羽");
		td2.setName("张飞");

		td1.setDaemon(true);
		td2.setDaemon(true);

		td1.start();
		td2.start();

		Thread.currentThread().setName("刘备");
		for (int x = 0; x < 5; x++) {
			System.out.println(Thread.currentThread().getName() + ":" + x);
		}
		// 刘备死后关羽和张飞也活不了
	}
}
```

中断线程

2018年7月26日 16:49

```
public class ThreadStop extends Thread {
	@Override
	public void run() {
		System.out.println("开始执行"+new Date());

		try {
			Thread.sleep(10000);
		} catch (InterruptedException e) {
			System.out.println("线程中断");
		}

		System.out.println("结束执行"+new Date());
	}
}
```

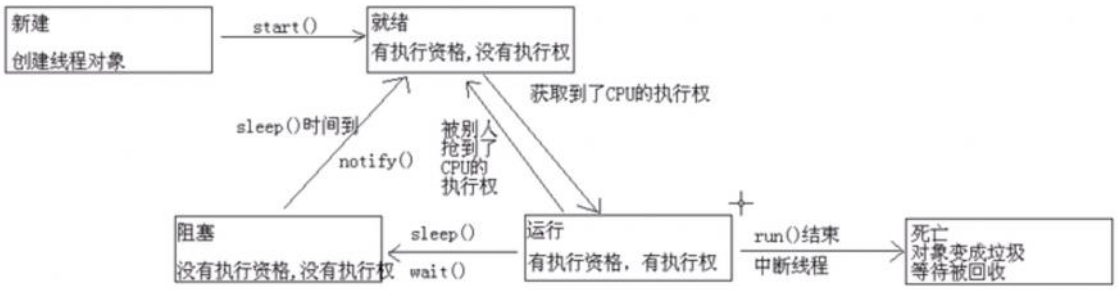
```
/*
 * public final void stop():方法已过时
 * public void interrupt():中断线程,把线程的状态终止，并抛出InterruptedException
 */
public class ThreadStopDemo {
	public static void main(String[] args){
		ThreadStop ts = new ThreadStop();
		ts.start();

		//超过3秒结束
		try {
			Thread.sleep(3000);
			//ts.stop();不建议使用
			ts.interrupt();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
}
```

图解

2018年7月26日 22:44

面试题：线程的生命周期？

新建：创建线程对象
就绪：有执行资格，没有执行权
运行：有执行资格，有执行权
　　阻塞：由于一些操作让线程处于了该状态。没有执行资格，没有执行权
　　而另一些操作却可以把它给激活，激活后处于就绪状态。
死亡：线程对象变成垃圾，等待被回收



实现多线程方式2

2018年7月26日 22:45

```
public class MyRunnable implements Runnable{
	@Override
	public void run() {
		for(int x=0;x<100;x++) {
			//由于实现了接口，不能使用getName()方法
			System.out.println(Thread.currentThread().getName()+":"+x);
		}
	}
}

/*
 * 实现多线程的方式2：实现Runnable接口
 * 步骤：
 *		1.自定义类MyRunnable实现Runnable接口
 *		2.重写run()方法
 *		3.创建MyRunnable类对象
 *		4.创建Thread类的对象，并把C步骤的对象作为构造参数传递
 */
public class MyRunnableDemo {
	public static void main(String[] args) {
		MyRunnable my = new MyRunnable();

//		Thread t1 = new Thread(my);
//		Thread t2 = new Thread(my);
//
//		t1.setName("王若潇");
//		t2.setName("屠龙");

		Thread t1 = new Thread(my,"王若潇");
		Thread t2 = new Thread(my,"屠龙");

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

有了方式1，问什么还有方式2?

1.解决了但继承的局限性
2.适合多个相同程序的代码去处理同一个资源的情况把线程同程序的代码，数据有效分离，较好的体现了面向对象的设计思想

售票案例

2018年7月26日 23:02

```
public class SellTicket implements Runnable {
	// 若用方式1：为了让多个线程共享这100张票，应该用静态修饰
	// private static int tickets = 100;
	// 采用方式2：不需要static
	private int tickets = 100;

	@Override
	public void run() {
		while (true) {
			if (tickets > 0) {
				// 考虑网络的延迟，稍作休息
				try {
					Thread.sleep(100);
				} catch (InterruptedException e) {
					// TODO Auto-generated catch block
					e.printStackTrace();
				}
				System.out.println(Thread.currentThread().getName() + "正在售
				第" + (tickets--) + "张票");
			}
		}
	}
}



/*
 * 模拟电影院售票，共100张票，共有3个售票窗口
 * 3个线程
 *
 * 模拟延迟100毫秒
 * 出现了两个问题：线程安全问题
 * 出现重复：CPU的一次行操作必须是原子性的 tickets--
 * 出现负数：随机性和延迟导致的
 *
 */
public class SellTicketDemo {
```

```
    public static void main(String[] args) {
        SellTicket st = new SellTicket();

        Thread t1 = new Thread(st, "窗口1");
        Thread t2 = new Thread(st, "窗口2");
        Thread t3 = new Thread(st, "窗口3");

        t1.start();
        t2.start();
        t3.start();       }
}
```

线程安全

2018年7月27日 0:30

如何解决线程安全问题?
1.是否是多线程环境
2.是否有共享数据
3.是否有多条语句对共享数据进行操作

```
public class SellTicket implements Runnable {
	private int tickets = 100;
	private Object obj = new Object();
	@Override
	public void run() {
		while (true) {
			synchronized (obj) {
				if (tickets > 0) {
					try {
						Thread.sleep(100);
					} catch (InterruptedException e) {
						e.printStackTrace();
					}
					System.out.println(Thread.currentThread().getName() + "正
					在售第" + (tickets--) + "张票");
				}
			}
		}
	}
}

/*
 * 出现了两个问题：线程安全问题
 * 出现重复：CPU的一次行操作必须是原子性的 tickets--
 * 出现负数：随机性和延迟导致的
 * java提供了同步代码块方法
 * synchronized(对象){需要同步的代码;}
 *
 * 同步可以解决安全问题的根本原因在于对象，该对象如同锁的功能
 * 多个线程必须是同一把锁
 */
```

```
public class SellTicketDemo {
	public static void main(String[] args) {
		SellTicket st = new SellTicket();

		Thread t1 = new Thread(st, "窗口1");
		Thread t2 = new Thread(st, "窗口2");
		Thread t3 = new Thread(st, "窗口3");

		t1.start();
		t2.start();
		t3.start();	}
}
```

同步代码块

2018年7月27日 10:02

同步代码块的对象可以是任何对象

同步方法的格式

把同步关键字加在方法上

同步方法的锁对象-->**this**

public synchronized void run(){}

静态方法及所对象

同步方法的锁对象-->**当前类的class文件（反射）**

线程安全的类

2018年7月27日 10:38

```
public class ThreadDemo {
	public static void main(String[] args) {
		// 线程安全的类（都加锁了）
		StringBuffer sb = new StringBuffer();
		Vector<String> v = new Vector<>();
		Hashtable<String, String> h = new Hashtable<>();

		// Vector是线程安全时才考虑使用的，但也不使用

		// 创建线程安全的List
		List<String> list = Collections.synchronizedList(new ArrayList<String>());

	}
}
```