1、课程名称：多线程

2、知识点

2.1、上次课程的主要知识点

1、 包的定义及导入
2、 四种访问控制权限

2.2、本次预计讲解的知识点

1、 多线程的操作中，对于线程功能的开发并不要求，但是其基本概念及各个操作语句必须熟练
2、 掌握多线程的两种实现方式及区别
3、 了解多线程的主要操作方法
4、 掌握线程的同步与死锁的概念

3、具体内容

3.1、进程与线程（了解）

传统的 DOS 系统有一个很明显的特点，一旦程序中出现了病毒的话，则整个电脑将处于瘫痪的状态，这是因为传统的 DOS 操作系统采用的是单进程的处理方式，即：在同一个时间段上只能有一个进程在运行着。

到了 windows 时代可以发现电脑中即使有了病毒也照样可以使用。因为 windows 本身属于多进程的操作系统，可以在同一个时间段上运行多个程序，所有的程序都是并发运行的，但是在同一个时间点上只能有一个进程在执行。

线程实际上是在进程基础上的进一步划分，一个进程可以划分成多个线程。

就好比 word 一样的拼写检查一样，只有在 word 运行的时候（相当于一个进程）才可能存在拼写检查的操作（相当于一个线程）。

3.2、Java 的线程实现（理解）

在 Java 中如果要想进行多线程代码的实现有两种方式：

- 继承 Thread 类
- 实现 Runnable 接口

下面通过代码分别来验证以上的两种方式及区别。

3.2.1、继承 Thread 类

当一个类需要按照多线程的方式处理时，可以让这个类直接继承自 Thread 类即可，而且继承的时候要覆写好 Thread 类中提供的 run()方法：

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

范例：按照要求定义一个线程类

```
class MyThread extends Thread {         // 继承 Thread 类
    public void run(){                  // 做为线程的主体
        for(int x=0;x<5;x++){
            System.out.println("x = " + x--) ;
        }
    }
};
```

一个线程类已经定义完成，既然是按照类的形式进行操作的，则肯定需要通过对象进行具体功能的调用，但是如果要想启动一个线程并不是依靠 run()方法而是 start()方法。

```
class MyThread extends Thread {         // 继承 Thread 类
    private String name ;
    public MyThread(String name){
        this.name = name ;
    }
    public void run(){                  // 做为线程的主体
```

```
            for(int x=0;x<100;x++){
                System.out.println(this.name + "运行， x = " + x) ;
            }
        }
};
public class ThreadDemo01 {
        public static void main(String args[]){
            MyThread mt1 = new MyThread("线程 A") ;
            MyThread mt2 = new MyThread("线程 B") ;
            MyThread mt3 = new MyThread("线程 C") ;
            mt1.start() ;
            mt2.start() ;
            mt3.start() ;
        }
};
```

此时通过 start()方法执行线程的操作，操作中可以发现，每一个线程间都属于交替的运行状态，即：所有的线程都是交替运行的，且：那个线程抢到了 CPU 资源，那个线程就执行。

问题：为什么启动线程的时候必须是 start(0 方法，而不能是 run()方法呢？

如果要想解释这个问题，肯定需要观察 Thread 类的源代码。

```
    public synchronized void start() {
        if (threadStatus != 0)
            throw new IllegalThreadStateException();
            group.add(this);
            start0();
            if (stopBeforeStart) {
            stop0(throwableFromStop);
        }
    }
    private native void start0();
```

可以发现在 start()方法中会抛出一个 IllegalThreadStateException 的异常，当一个线程重复启动的时候会产生此异常。

但是在这个方法中最需要关注的是一个：stop0()这个方法。但是这个方法没有实现，而且使用了 native 关键字声明。

native 实际表示的是一个方法要调用本机操作系统的函数支持，可以想象一下，对于多线程由于要进行 CPU 的抢占，等待资源调度及分配，所以这种功能只能依靠不同的操作系统而有所不同的进行实现。

那么这种可以通过 Java 程序调用本机操作系统程序的功能也称为 JNI（Java Native Interface），但是这样一来也会有个问题，如果某一个程序与特定的操作系统绑定的话，那么可移植性将彻底的丧失，所以标准开发中是不推荐使用的。

虽然现在可以实现了多线程的操作，但是由于现在必须继承 Thread 类，根据 Java 的单继承局限，这种实现方式肯定是不好的，那么解决单继承靠的是接口。

3.2.2、实现 Runnable 接口

线程实现的第二种手段，实际上就是可以实现 Runnable 接口来实现线程的操作类，Runnable 接口定义如下：

```
public interface Runnable{
    public void run() ;
```

```
}
```

可以发现这个接口中也定义了一个 run()方法，下面观察线程的实现：

```
class MyThread implements Runnable {        // 实现 Runnable 接口
    private String name ;
    public MyThread(String name){
        this.name = name ;
    }
    public void run(){                      // 做为线程的主体
        for(int x=0;x<10;x++){
            System.out.println(this.name + "运行， x = " + x) ;
        }
    }
};
```

线程确实已经实现了，但是需要注意的是，如果要想启动一个线程肯定是 Thread 类中的 start()方法完成，观察 Thread 类中提供的构造方法：public Thread(Runnable target)

通过构造发现，Thread 类可以接收 Runnable 子类的对象，所以一切的线程都可以通过 Thread 类进行启动。

```
public class ThreadDemo02 {
    public static void main(String args[]){
        MyThread mt1 = new MyThread("线程 A") ;
        MyThread mt2 = new MyThread("线程 B") ;
        MyThread mt3 = new MyThread("线程 C") ;
        new Thread(mt1).start() ;
        new Thread(mt1).start() ;
        new Thread(mt2).start() ;
        new Thread(mt3).start() ;
    }
};
```

此时，通过 Thread 类进行了线程的启动。

3.2.3、两种实现方式的区别

对于 Thread 类和 Runnable 接口本身都是可以进行多线程的实现，那么两者到底该使用谁更好呢？

1、 继承局限：使用 Runnable 接口可以避免单继承的局限，而 Thread 类则有此局限；

2、 资源共享：使用 Runnable 接口实现多线程，可以实现资源（对象属性）的共享，而 Thread 类却无法实现。

|- 此点只是相对而言，因为两者的此种区别是有其应用范围的。

范例：观察资源共享

```
class MyThread extends Thread {
    private int count = 5 ;
    public void run(){
        for(int x=0;x<50;x++){
            if(this.count>0){
                System.out.println("count = " + this.count--) ;
            }
```

```
            }
        }
};
public class ThreadDemo03 {
        public static void main(String args[]){
                new MyThread().start() ;
                new MyThread().start() ;
                new MyThread().start() ;
        }
};
```

现在的程序中每一个线程都各自占有各自的 count 属性，所以并没有达到资源共享的目的，那么如果现在换成了 Runnable 呢？

```
class MyThread implements Runnable {
        private int count = 5 ;
        public void run(){
                for(int x=0;x<50;x++){
                        if(this.count>0){
                                System.out.println("count = " + this.count--) ;
                        }
                }
        }
};
public class ThreadDemo04 {
        public static void main(String args[]){
                MyThread mt = new MyThread() ;
                new Thread(mt).start() ;
                new Thread(mt).start() ;
                new Thread(mt).start() ;
        }
};
```

现在的代码中可以发现，count 属性已经被所有的线程对象所共同拥有了。

3.2.4、两种实现方式的联系

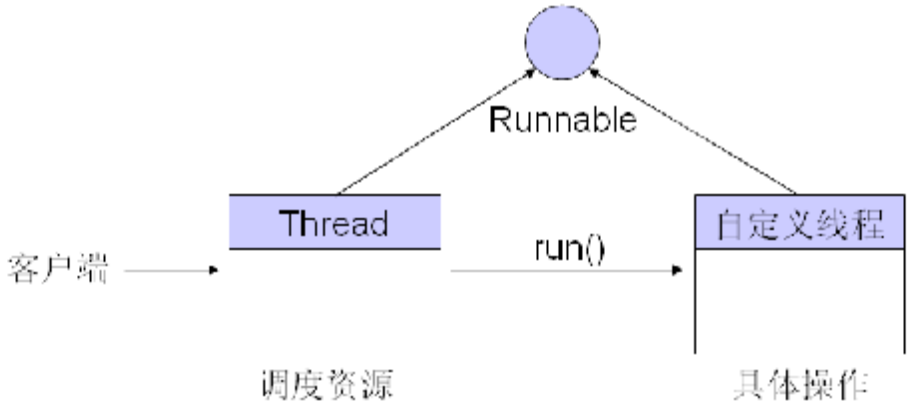
从 Thread 类和 Runnable 接口中都可以发现，都必须同时覆写 run()方法，那么两者的关系如何呢？观察 Thread 类的定义：

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

发现 Thread 类实际上是 Runnable 的子类。而且 Thread 类也要去接收 Runnable 其他子类的对象，而且所有的线程中，通过 Runnable 接口实现的线程类里面都是编写的具体功能，而并没有所谓的 CPU 调度，而真正意义上的 CPU 调度由操作系统完成（通过 Thread 类的 start()方法调用的）。

Thread 类要去协调操作系统，并且最终还要执行具体的线程主体的方法，而线程的主体呢，现在只专著于具体的功能实现，至于如何调度根本不管。

Thread 代理自定义的线程类的对象，如图所示：



从图的关系上可以清楚的发现，现在在线程中应用的设计思路就是代理设计模式。

3.2.5、线程的状态

每一个线程对象都要经历五个步骤：

1、 初始化：当创建了一个新的线程对象时

2、 等待：调用了 start()方法

3、 执行：调用 run()执行的操作的过程

4、 停止：因为所有的线程都需要进行 CPU 资源的抢占，那么当一个线程执行完部分代码要交出资源，留给其他线程继续执行。

5、 卸载：所有的线程的操作代码都执行完毕之后，就将线程对象卸载下来。

3.3、线程的操作方法（理解）

在 Java 中所有的线程的操作方法都是在 Thread 类中定义的，那么由于方法众多，所以，只介绍几个相对主要的方法。

3.3.1、命名和取得

每一个线程实际上都可以为其设置名字，而且也可以取得每一个线程的名字：

- 设置线程名称：public final void setName(String name)
- 取得线程名称：public final String getName()

但是有一点也非常的麻烦，由于线程的操作不属于固定的状态，所以对于取得线程名称的操作，应该是取得的当前正在运行的线程名称才合适，那么可以通过如下的方法取得一个当前正在运行的线程对象：

- 取得当前线程：public static Thread currentThread()

除了以上的设置名称的方法外，在 Thread 类中也提供了两个构造方法：

- public Thread(String name)
- public Thread(Runnable target,String name)

注意：一般都在线程启动前设置好名字，当然也可以为已经启动的线程修改名字或设置重名线程，不过这样不好。

```
class MyThread implements Runnable {
    public void run(){
        for(int x=0;x<5;x++){
```

```
                System.out.println(Thread.currentThread().getName() + "运行，x = " + x) ;
            }
        }
};
public class NameDemo {
    public static void main(String args[]){
        MyThread mt = new MyThread() ;
        new Thread(mt).start() ;
        new Thread(mt,"线程 A").start() ;
        new Thread(mt).start() ;
        new Thread(mt).start() ;
        new Thread(mt).start() ;
    }
};
```

本程序明白之后，下面再观察以下的程序输出：

```
class MyThread implements Runnable {
    public void run(){
        for(int x=0;x<5;x++){
            System.out.println(Thread.currentThread().getName() + "运行，x = " + x) ;
        }
    }
};
public class NameDemo {
    public static void main(String args[]){
        MyThread mt = new MyThread() ;
        new Thread(mt,"自定义线程").start() ;
        mt.run() ;  // main 线程
    }
};
```

在本程序中发现出现了一个 main 线程，那么这个线程肯定是主方法产生的，之前一直强调，java 本身是属于多线程的处理机制，所以每次 java 运行的时候，实际上都会启动一个 JVM 的进程。

那么既然是多线程的处理机制，实际上主方法是在一个 JVM 上产生的一个线程而已，那么一个 JVM 启动的时候至少启动几个线程呢？两个：main、GC。

3.3.2、线程的休眠（重点）

所谓的休眠就是指减缓程序的运行速度，如果要休眠使用如下的方法：

- 休眠：public static void sleep(long millis) throws InterruptedException，指定休眠时间

```
class MyThread implements Runnable {
    public void run(){
        for(int x=0;x<5;x++){
            System.out.println(Thread.currentThread().getName() + "运行，x = " + x) ;
            try{
```

```
                    Thread.sleep(300) ;
                } catch (Exception e){}
            }
        }
};
public class SleepDemo {
        public static void main(String args[]){
            MyThread mt = new MyThread() ;
            new Thread(mt,"自定义线程").start() ;
            mt.run() ;  // main 线程
        }
};
```

3.3.3、线程的优先级

实际上所有的线程启动之后并不是立刻运行的，都需要等待 CPU 进行调度，但是调度的时候本身也是存在“优先”级的，如果优先级高则有可能最先被执行。

如果要想设置优先级可以使用：public final void setPriority(**int newPriority**)

这个优先级需要接收一个整型的数字，这个数字只能设置三个内容：

- 最高优先级：public static final int MAX_PRIORITY
- 中等优先级：public static final int NORM_PRIORITY
- 最低优先级：public static final int MIN_PRIORITY

范例：观察优先级设置的影响

```
class MyThread implements Runnable {
        public void run(){
            for(int x=0;x<5;x++){
                System.out.println(Thread.currentThread().getName() + "运行，x = " + x) ;
            }
        }
};
public class ProDemo {
        public static void main(String args[]){
            MyThread mt = new MyThread() ;
            Thread t1 = new Thread(mt) ;
            Thread t2 = new Thread(mt) ;
            Thread t3 = new Thread(mt) ;
            t1.setPriority(Thread.MIN_PRIORITY) ;
            t1.start() ;
            t2.setPriority(Thread.MAX_PRIORITY) ;
            t2.start() ;
            t3.setPriority(Thread.NORM_PRIORITY) ;
            t3.start() ;
        }
```

```
};
```

问题：主方法的优先级是什么？

```
public class MainDemo {
    public static void main(String args[]){
        System.out.println(Thread.currentThread().getPriority()) ;
        System.out.println("MAX_PRIORITY " + Thread.MAX_PRIORITY) ;
        System.out.println("MIN_PRIORITY " + Thread.MIN_PRIORITY) ;
        System.out.println("NORM_PRIORITY " + Thread.NORM_PRIORITY) ;
    }
};
```

主方法属于中等优先级。

3.4、线程的同步与死锁（理解）

由于多个线程可以对同一个资源进行操作，那么就必须进行同步的处理，但是如果过多的引入了同步的处理，也可能造成死锁。

3.4.1、同步

当多个线程同时进行一种资源操作，为了保证操作的完整性，引入了同步处理。

```
class MyThread implements Runnable {
    private int ticket = 10 ;
    public void run(){
        for(int x=0;x<50;x++){
            if(this.ticket > 0){
                System.out.println(Thread.currentThread().getName()
                    + "卖票，剩余：" + this.ticket--) ;
            }
        }
    }
};
public class SynDemo01 {
    public static void main(String args[]){
        MyThread mt = new MyThread() ;
        new Thread(mt,"票贩子 A").start() ;
        new Thread(mt,"票贩子 B").start() ;
        new Thread(mt,"票贩子 C").start() ;
    }
};
```

但是，从实际的操作来看，都是使用网络卖票，既然是网络卖票的话，则有可能出现延迟。

```
class MyThread implements Runnable {
    private int ticket = 10 ;
    public void run(){
```

```
            for(int x=0;x<50;x++){
                if(this.ticket > 0){
                    try{
                        Thread.sleep(300) ;
                    } catch (Exception e){}
                    System.out.println(Thread.currentThread().getName()
                        + "卖票，剩余：" + this.ticket--) ;
                }
            }
        }
};
public class SynDemo01 {
        public static void main(String args[]){
            MyThread mt = new MyThread() ;
            new Thread(mt,"票贩子 A").start() ;
            new Thread(mt,"票贩子 B").start() ;
            new Thread(mt,"票贩子 C").start() ;
        }
};
```

本程序中可以发现一旦加入了延迟（不加延迟也可能有问题）之后发现有的人卖出的票数是负数。

由于现在是有多个操作，那么最好的解决方法是加入一个锁的标记，锁的标记中将判断和修改同时进行，要想完成这种锁的程序可以通过两种语句实现：同步代码块、同步方法。

范例：使用同步代码块完成

```
class MyThread implements Runnable {
        private int ticket = 10 ;
        public void run(){
            for(int x=0;x<50;x++){
                synchronized(this){     // 将当前对象锁定
                    if(this.ticket > 0){
                        try{
                            Thread.sleep(300) ;
                        } catch (Exception e){}
                        System.out.println(Thread.currentThread().getName()
                            + "卖票，剩余：" + this.ticket--) ;
                    }
                }
            }
        }
};
public class SynDemo02 {
        public static void main(String args[]){
            MyThread mt = new MyThread() ;
            new Thread(mt,"票贩子 A").start() ;
            new Thread(mt,"票贩子 B").start() ;
```

```
            new Thread(mt,"票贩子 C").start() ;
        }
};
```

程序中加入了同步操作之后可以发现执行的速度变慢了。

除了可以使用同步代码块之外还可以使用同步方法完成以上的操作。

```
class MyThread implements Runnable {
        private int ticket = 10 ;
        public void run(){
            for(int x=0;x<50;x++){
                this.sale() ;
            }
        }
        public synchronized void sale() {
            if(this.ticket > 0){
                try{
                    Thread.sleep(300) ;
                } catch (Exception e){}
                System.out.println(Thread.currentThread().getName()
                    + "卖票，剩余：" + this.ticket--) ;
            }
        }
};
public class SynDemo03 {
        public static void main(String args[]){
            MyThread mt = new MyThread() ;
            new Thread(mt,"票贩子 A").start() ;
            new Thread(mt,"票贩子 B").start() ;
            new Thread(mt,"票贩子 C").start() ;
        }
};
```

3.4.2、死锁

多个线程间进行互相等待的情况。

甲对乙说：你给我一根笔我给你书

乙对甲说：你给我说，我给你笔。

由于死锁是在程序运行中出现的一种状态，所以下面通过代码来模拟，但是本代码没有任何的意义。

```
class Jia {
        public synchronized void say(Yi yi){
            System.out.println("甲对乙说：你给我一根笔我给你书") ;
            yi.give() ;
        }
        public synchronized void give(){
```

```
            System.out.println("甲给乙书了。") ;
        }
};
class Yi {
        public synchronized void say(Jia jia){
            System.out.println("乙对甲说：你给我书，我给你笔。") ;
            jia.give() ;
        }
        public synchronized void give(){
            System.out.println("乙给甲笔了。") ;
        }
};
public class DeadLock implements Runnable {
        private Jia j = new Jia() ;
        private Yi y = new Yi() ;
        public DeadLock(){
            new Thread(this).start() ;
            j.say(y) ;
        }
        public void run(){
            y.say(j) ;
            try{
                Thread.sleep(1000) ;
            } catch (Exception e){}
        }
        public static void main(String args[]){
            new DeadLock() ;
        }
};
```

一般死锁都是在程序运行中出现的一种状态，以上只是模拟，代码没有任何的实际的作用。

多个线程访问同一个资源的时候需要进行同步，但是过多的同步会产生死锁。

现在就可以给出一个方法的完整定义格式

[public | protected | private] [static] [final] [synchronized]
返回值类型 方法名称(参数列表) [throws 异常 1,异常 2,…]{
    [return 返回值 ;]
}

3.5、线程操作的经典案例 —— 生产者和消费者

以上就是多线程的基本操作，但是在整个多线程存在一个经典的交互案例，生产者和消费者。先通过代码观察问题，现在假设说要生产的是一组信息，此组信息有两种选项：

- oracle à 数据库
- java à www.sun.com.cn

既然要生产信息，则肯定要建立一个信息的保存对象。

```
class Info {
	private String name = "oracle" ;
	private String desc = "数据库" ;
	public void setName(String name){
		this.name = name ;
	}
	public void setDesc(String desc){
		this.desc = desc ;
	}
	public String getName(){
		return this.name ;
	}
	public String getDesc(){
		return this.desc ;
	}
};
```

下面建立生产者和消费者，同时为了保证两者之间存在共同的生产线，所以两个类分别都占有同一个 Info 的引用。

```
class Pro implements Runnable {
	private Info info ;
	public Pro(Info info){
		this.info = info ;
	}
	public void run(){
		for(int x=0;x<50;x++){
			if(x%2==0){
				this.info.setName("Java") ;
				try{
					Thread.sleep(300) ;
				} catch (Exception e){}
				this.info.setDesc("www.sun.com") ;
			} else {
				this.info.setName("Oracle") ;
				try{
					Thread.sleep(300) ;
				} catch (Exception e){}
				this.info.setDesc("数据库") ;
			}
		}
	}
};
class Cus implements Runnable {
	private Info info ;
	public Cus(Info info){
```

```
            this.info = info ;
        }
        public void run(){
            for(int x=0;x<50;x++){
                try{
                    Thread.sleep(300) ;
                } catch (Exception e){}
                System.out.println(this.info.getName() + " --> " + this.info.getDesc()) ;
            }
        }
};
public class CommDemo {
        public static void main(String args[]){
            Info info = new Info() ;
            Pro p = new Pro(info) ;
            Cus c = new Cus(info) ;
            new Thread(p).start() ;
            new Thread(c).start() ;
        }
};
```

通过以上的操作代码可以发现，现在的程序有两点问题：

1、 数据的设置错误

2、 出现了重复取或重复设置的问题。

3.5.1、解决错误数据

需要使用同步的方法来解决第一个问题，可以通过以下的修改完成。

```
class Info {
        private String name = "oracle" ;
        private String desc = "数据库" ;
        public synchronized void set(String name,String desc){
            this.setName(name) ;
            try{
                Thread.sleep(300) ;
            } catch (Exception e){}
            this.setDesc(desc) ;
        }
        public synchronized void get(){
            try{
                Thread.sleep(300) ;
            } catch (Exception e){}
            System.out.println(this.name + " --> " + this.desc) ;
        }
```

```
	public void setName(String name){
		this.name = name ;
	}
	public void setDesc(String desc){
		this.desc = desc ;
	}
	public String getName(){
		return this.name ;
	}
	public String getDesc(){
		return this.desc ;
	}
};
class Pro implements Runnable {
	private Info info ;
	public Pro(Info info){
		this.info = info ;
	}
	public void run(){
		for(int x=0;x<50;x++){
			if(x%2==0){
				this.info.set("Java","www.sun.com") ;
			} else {
				this.info.set("Oracle","数据库") ;
			}
		}
	}
};
class Cus implements Runnable {
	private Info info ;
	public Cus(Info info){
		this.info = info ;
	}
	public void run(){
		for(int x=0;x<50;x++){
			this.info.get() ;
		}
	}
};
public class CommDemo {
	public static void main(String args[]){
		Info info = new Info() ;
		Pro p = new Pro(info) ;
		Cus c = new Cus(info) ;
```

```
            new Thread(p).start() ;
            new Thread(c).start() ;
        }
};
```

现在的程序代码之中，已经增加了同步操作。

现在虽然避免了错误数据的问题，但是重复取和重复设置的问题依然存在。

3.5.2、Object 类对线程的支持

在 Object 类中提供了以下的方法可以实现对线程的等待及唤醒的处理：

- 等待：public final void wait() throws InterruptedException
- 唤醒：public final void notify()，唤醒第一个等待的线程
- 唤醒：public final void notifyAll()，唤醒全部等待的线程

下面就可以利用这种机制观察对线程的处理操作，解决重复读取和设置的问题，修改 Info 类即可。

```
class Info {
    private String name = "oracle" ;
    private String desc = "数据库" ;
    private boolean flag = false ;// 定义一个标记
    /*
        flag = true，表示可以生产，但是不能取得
        flag = false，表示可以取得，但是不能生产
    */
    public synchronized void set(String name,String desc){
        if(!this.flag){
            try{
                super.wait() ;
            }catch(Exception e){}
        }
        this.setName(name) ;
        try{
            Thread.sleep(300) ;
        } catch (Exception e){}
        this.setDesc(desc) ;
        this.flag = false ;
        super.notify() ;   // 唤醒
    }
    public synchronized void get(){
        if(this.flag){
            try{
                super.wait() ;
            }catch(Exception e){}
        }
        try{
```

```
                Thread.sleep(300) ;
            } catch (Exception e){}
            System.out.println(this.name + " --> " + this.desc) ;
            this.flag = true ;
            super.notify() ;
        }
        public void setName(String name){
            this.name = name ;
        }
        public void setDesc(String desc){
            this.desc = desc ;
        }
        public String getName(){
            return this.name ;
        }
        public String getDesc(){
            return this.desc ;
        }
};
```

4、总结

1、 两种实现方式及区别

2、 理解同步与死锁的概念

5、预习任务

Java 的类库，StringBuffer、Runtime、Date、Calendar、SimpleDateFormat、Random、Math。