初识线程

- 什么是线程

线程(thread)是一个程序内部的一条执行路径。是CPU调度和分派的基本单位
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
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。

s

多线程的创建

方式一：继承Thread类

- 概念

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

- 方式一创建多线程步骤

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）

```
public class FirstThreadDemo {
    public static void main(String[] args) {
        final Thread myThread = new MyThread();
        /*
         * 调用start()方法，是会把当前对象注册到队列中，成为一条线程(会使用底层真正的线程处理代码逻辑)
         *  group.add(this);
         *
         * 如果是使用run()方法，就是平常的通过main线程跑对象的方法(单线程)
         * */
        myThread.start();

        /*
         *main()方法需要执行的功能代码，需要放到，开启子线程之后，
         * 否则会先跑完main()的功能，再开一条新的线程跑其功能；相当于单线程
         * */
        for (int i = 0; i < 10; i++) {
            System.out.println("主线程跑~~~" + i);
        }
    }
}
```

```
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("子线程跑~~~" + i);
        }
    }
}
```

```
主线程跑~~~0
主线程跑~~~1
主线程跑~~~2
主线程跑~~~3
主线程跑~~~4
主线程跑~~~5
子线程跑~~~0
主线程跑~~~6
子线程跑~~~1
主线程跑~~~7
子线程跑~~~2
主线程跑~~~8
子线程跑~~~3
子线程跑~~~4
子线程跑~~~5
子线程跑~~~6
子线程跑~~~7
子线程跑~~~8
子线程跑~~~9
主线程跑~~~9
```

- 优缺点

优点：编码简单
缺点：线程类已经继承Thread，无法继承其他类，不利于扩展

- 问题

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会怎样?
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(不能看到多条线程竞争cpu，并发执行的现象)

方式二：实现Runnable接口

- 实现步骤

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
   可以使用无参构造创建线程，线程名称默认
   Thread类的入参可以是Runnable但不能是Callable，所以要将Callable包装成，Runnable类型(使用FutureTask类，它实现了Runnable接口)

Thread的构造器

构造器	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

---

public Thread() 线程类还有一个无参构造 线程名称默认
Runnable是一个任务类，是由线程调的它

- Runnable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。（无返回值）

- 实现代码

```
public class SecondThreadDemo {

    public static void main(String[] args) {
        final MyRunnable runnable = new MyRunnable();
        final Thread t = new Thread(runnable);//线程类的一个以任务实例为入参的构造器
        t.start();//在main线程存在的情况下，又开了一条子线程；此时两条线程竞争cpu

        for (int i = 0; i < 10; i++) {
            System.out.println("main线程在跑--" + i);
        }
    }
}

//接口与接口多继承，小心规范冲突；类与类单继承，多实现，小心规范冲突
//规范冲突:名称，但不构成重构
class MyRunnable implements Runnable {//只是实现了任务类接口，还可以继承其他类和实现其他接口；只要不规范冲突就行

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("子线程跑--" + i);
        }
    }
}
```

```
main线程在跑--0
子线程跑--0
子线程跑--1
子线程跑--2
子线程跑--3
子线程跑--4
子线程跑--5
main线程在跑--1
子线程跑--6
main线程在跑--2
子线程跑--7
main线程在跑--3
子线程跑--8
main线程在跑--4
子线程跑--9
main线程在跑--5
main线程在跑--6
main线程在跑--7
main线程在跑--8
main线程在跑--9
```

- **实现方案二：实现Runnable接口(匿名内部类形式)**

实现步骤：
可以创建Runnable的匿名内部类对象。
交给Thread处理。
调用线程对象的start()启动线程。

- 实现代码

```
public class SecondThreadDemo2 {
    public static void main(String[] args) {
        //Lambda表达式简化匿名内部类
        new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                System.out.println("子线程2#" + i);
            }
        }).start();

        //匿名内部类的方式创建匿名内部类对象，类型是Runnable的子类型
        new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 10; i++) {
                    System.out.println("子线程一@" + i);
                }
            }
        }).start();

        for (int i = 0; i < 10; i++) {
            System.out.println("main线程跑" + i);
        }
    }
}
```

```
子线程2#0
子线程2#1
子线程2#2
子线程2#3
子线程2#4
子线程2#5
子线程2#6
子线程2#7
子线程2#8
子线程2#9
main线程跑0
main线程跑1
main线程跑2
main线程跑3
main线程跑4
main线程跑5
main线程跑6
main线程跑7
子线程一@0
子线程一@1
子线程一@2
子线程一@3
子线程一@4
子线程一@5
子线程一@6
子线程一@7
子线程一@8
子线程一@9
main线程跑8
main线程跑9
```

方式三：JDK 5.0新增：实现Callable接口

- 实现步骤

1. 得到任务对象
   定义类实现Callable接口，重写call方法，封装要做的事情。
   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

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。

- 代码实现

```
public class ThirdTreadDemo {

    public static void main(String[] args) {

        /*
         * 初始化线程一，并运行
         * */
        final MyCallable myCallable = new MyCallable(12);
//        new Thread(myCallable);//报错，线程类没有入参为Callable类型的构造器

        /*
        * public class FutureTask<V> implements RunnableFuture<V>
           public interface RunnableFuture<V> extends Runnable 未来任务类是Runnable的实现类
          *  public FutureTask(Callable<V> callable) 未来任务类有以Callable为入参的构造器
          * 这个未来任务类是Runnable接口的实现类，所以线程类可以跑这个任务类（前提是把Callable实现类交给未来任务类）
          * 可以通过调用未来任务类的get方法获得线程返回值（他会等待线程执行完毕才会返回值）
         * */
        final FutureTask<String> futureTask = new FutureTask<>(myCallable);
        final Thread t1 = new Thread(futureTask);
        t1.start();//开启线程一

        /*
         * 初始化线程二并运行
         * */
```

```
        final MyCallable myCallable1 = new MyCallable(10);
        final FutureTask<String> futureTask1 = new FutureTask<>(myCallable1);
        new Thread(futureTask1).start();


        try {
            final String s = futureTask.get();//获取线程一结果，如果线程一没跑完，这里会等待，同步阻塞
            final String s1 = futureTask1.get();//获取线程二结果，如果线程二没跑完，这里会等待，同步阻塞
            System.out.println("线程1的执行结果为：" + s);
            System.out.println("线程2的执行结果为：" + s1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

```
class MyCallable implements Callable<String> {//这里申明的类型 作为call方法返回值类型
    private int number;

    public MyCallable() {

    }

    public MyCallable(int number) {
        this.number = number;
    }

    int sum = 0;

    @Override
    public String call() throws Exception {
        for (int i = 1; i <= number; i++) {
            sum += i;
        }
        return "结果为:" + sum;
    }
}
```

```
线程1的执行结果为：结果为:78
线程2的执行结果为：结果为:55
```

- 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：编码稍微复杂一点

- 三种创建线程方式对比

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

Thread类的常用方法

- 当有很多线程在执行的时候，我们怎么去区分这些线程

此时需要使用Thread的常用方法：getName()、setName()、
currentThread():这个是获取当前线程的静态方法

Thread获取和设置线程名称

方法名称	说明
String getName()	获取当前线程的名称，默认线程名称是Thread-索引
void setName(String name)	将此线程的名称更改为指定的名称，通过构造器也可以设置线程名称

Thread类获得当前线程的对象

方法名称	说明
public static Thread currentThread():	返回对当前正在执行的线程对象的引用

此方法是Thread类的静态方法，可以直接使用Thread类调用。
这个方法是当哪个线程执行调用，就会得到哪个线程对象

Thread的构造器

方法名称	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

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

        new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                System.out.println(Thread.currentThread().getName() + "正在跑~" + i);//获取当前现成的名字
            }
        }, "Lambda的Runnable任务类").start();//通过构造器给当前线程设置名字

        new MyThread("继承Thread的线程").start();//通过构造器给当前线程设置名字 super(name);

        Thread.currentThread().setName("主线程（main）");//获取当前线程，并给他设置名字
        for (int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread().getName() + "正在跑" + i);
        }
    }


class MyThread extends Thread {
    public MyThread() {
    }

    public MyThread(String name) {
        super(name);//拿到名字直接传给父类构造器为线程取名
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(this.getName() + "子线程跑~~~" + i);
        }
    }
}
```

```
Lambda的Runnable任务类正在跑~0
Lambda的Runnable任务类正在跑~1
Lambda的Runnable任务类正在跑~2
Lambda的Runnable任务类正在跑~3
Lambda的Runnable任务类正在跑~4
Lambda的Runnable任务类正在跑~5
Lambda的Runnable任务类正在跑~6
Lambda的Runnable任务类正在跑~7
Lambda的Runnable任务类正在跑~8
Lambda的Runnable任务类正在跑~9
主线程（main）正在跑0
主线程（main）正在跑1
主线程（main）正在跑2
主线程（main）正在跑3
主线程（main）正在跑4
主线程（main）正在跑5
主线程（main）正在跑6
继承Thread的线程子线程跑~~~0
继承Thread的线程子线程跑~~~1
继承Thread的线程子线程跑~~~2
继承Thread的线程子线程跑~~~3
继承Thread的线程子线程跑~~~4
继承Thread的线程子线程跑~~~5
继承Thread的线程子线程跑~~~6
继承Thread的线程子线程跑~~~7
继承Thread的线程子线程跑~~~8
继承Thread的线程子线程跑~~~9
主线程（main）正在跑7
主线程（main）正在跑8
主线程（main）正在跑9
```

Thread类的线程休眠方法

方法名称	说明
public static void sleep(long time)	让当前线程休眠指定的时间后再继续执行，单位为毫秒。

```
    public static void main(String[] args) throws Exception {
        for (int i = 0; i < 8; i++) {
            System.out.println("主线程正在跑" + i);
            if (i == 4) Thread.sleep(3000);
        }
    }
```

```
主线程正在跑0
主线程正在跑1
主线程正在跑2
主线程正在跑3
主线程正在跑4(等了三秒后输出下面的)
主线程正在跑5
主线程正在跑6
主线程正在跑7
```

- Thread常用方法、构造器

方法名称	说明
String getName()	获取当前线程的名称，默认线程名称是Thread-索引
void setName(String name)	设置线程名称
public static Thread currentThread():	返回对当前正在执行的线程对象的引用
public static void sleep(long time)	让线程休眠指定的时间，单位为毫秒。
public void run()	线程任务方法
public void start()	线程启动方法

构造器	说明
public Thread(String name)	可以为当前线程指定名称
public Thread(Runnable target)	把Runnable对象交给线程对象
public Thread(Runnable target , String name)	把Runnable对象交给线程对象，并指定线程名称

线程安全

- 概念

多个线程同时操作同一个共享资源的时候可能会出现业务安全问题，称为线程安全问题

- 出现的原因

多个线程同时访问同一个共享资源且存在修改该资源

- 线程不安全案例

```java
public class SaveMethod {
    public static void main(String[] args) {
        final Card icbc = new Card("ICBC", 10000);
        new DrawThread(icbc, "母").start();
        new DrawThread(icbc, "公").start();

    }
}


class DrawThread extends Thread {
    private Card card;

    public DrawThread() {

    }

    public DrawThread(Card card, String name) {
        super(name);
        this.card = card;
    }

    @Override
    public void run() {
        card.drawMoney(10000);
    }
}
```

```java
class Card {
    private String cardId;
    private Integer money;

    public String getCardId() {
        return cardId;
    }

    public void setCardId(String cardId) {
        this.cardId = cardId;
    }

    public Integer getMoney() {
        return money;
    }

    public void setMoney(Integer money) {
```

```
        this.money = money;
    }

    public Card() {
    }

    public Card(String cardId, Integer money) {
        this.cardId = cardId;
        this.money = money;
    }

    public void drawMoney(int money) {
        final String name = Thread.currentThread().getName();
        System.out.println(name + "来了");
        if (this.money >= money) {
            System.out.println(name + "取出了" + money + "钱");//此语句有点耗时，当有个线程跑到这里，大概率要让出
cpu，然后另一个线程也会跑到这；此时就有可能出事
            this.money -= money;
            System.out.println(name + "还剩" + this.money + "钱");
        } else {
            System.out.println("没钱");
        }
    }
}
```

```
母来了
公来了
母取出了10000钱
公取出了10000钱
母还剩0钱
公还剩-10000钱
```

线程同步

同步思想概述

- 如何才能保证线程安全

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

- 线程同步的核心思想

加锁，把共享资源进行上锁，每次只能一个线程进入访问完毕以后解锁，然后其他线程才能进来

方式一：同步代码块

- 作用

把出现线程安全问题的核心代码给上锁

- 原理

每次只能一个线程进入，执行完毕后自动解锁，其他线程才可以进来执行

- 样式

```
synchronized(同步锁对象) {
//操作共享资源的代码(核心代码)
}
```

- **锁对象要求**

对于当前同时执行的线程来说是同一个对象即可

- 加锁后

```
package thread_demo;


public class SaveMethod {
    public static void main(String[] args) {
        final Card icbc = new Card("ICBC", 10000);
        new DrawThread(icbc, "母").start();
        new DrawThread(icbc, "公").start();

    }
}
```

```
class DrawThread extends Thread {
    private Card card;

    public DrawThread() {

    }

    public DrawThread(Card card, String name) {
        super(name);
        this.card = card;
    }

    @Override
    public void run() {
        card.drawMoney(10000);
    }
}
```

```

class Card {
    private String cardId;
    private Integer money;
```

```
    public String getCardId() {
        return cardId;
    }

    public void setCardId(String cardId) {
        this.cardId = cardId;
    }

    public Integer getMoney() {
        return money;
    }

    public void setMoney(Integer money) {
        this.money = money;
    }

    public Card() {
    }

    public Card(String cardId, Integer money) {
        this.cardId = cardId;
        this.money = money;
    }

    public void drawMoney(int money) {
        final String name = Thread.currentThread().getName();
        System.out.println(name + "来了");
        /*
         * 人都是不患寡而患不均，当锁对象对每个人相同时，每个人都遵守规则:别人在用时，其他人不进去
         * 但这是不现实的，这种范围不会这么大，只能是一部分人，所以当锁对象对于某一部分人相同时，这部分人遵守，其他人不守
         * */
        synchronized (this) {
            if (this.money >= money) {
                System.out.println(name + "取出了" + money + "钱");
                this.money -= money;
                System.out.println(name + "还剩" + this.money + "钱");
            } else {
                System.out.println("没钱");
            }
        }
    }
}
```

```
母来了
公来了
母取出了10000钱
母还剩0钱
没钱
```

- **锁对象用任意唯一的对象好不好**

不好，会影响其他无关线程的执行；用唯一的对象，所有线程进入该代码块时，都要等待；但人家访问的是自己的资源；关你屁事；

- **锁对象的规范要求**

规范上：建议使用共享资源作为锁对象。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；因为类成员是被该类所有实例共享的

- 同步代码块是如何实现线程安全的

对出现问题的核心代码使用synchronized进行加锁
每次只能一个线程占锁进入访问

方式二：同步方法

- 作用

把出现线程安全问题的核心方法给上锁
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

- 格式

```
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
}
```

- 代码实现

```
public class SaveMethod {
    public static void main(String[] args) {
        final Card icbc = new Card("ICBC", 10000);
        new DrawThread(icbc, "母").start();
        new DrawThread(icbc, "公").start();
    }
}
```

```
class DrawThread extends Thread {
    private Card card;

    public DrawThread() {

    }

    public DrawThread(Card card, String name) {
        super(name);
        this.card = card;
    }

    @Override
    public void run() {
        card.drawMoney(10000);
```

```
    }
}
```

```
class Card {
    private String cardId;
    private Integer money;

    public String getCardId() {
        return cardId;
    }

    public void setCardId(String cardId) {
        this.cardId = cardId;
    }

    public Integer getMoney() {
        return money;
    }

    public void setMoney(Integer money) {
        this.money = money;
    }

    public Card() {
    }

    public Card(String cardId, Integer money) {
        this.cardId = cardId;
        this.money = money;
    }

    public synchronized void drawMoney(int money) {
        final String name = Thread.currentThread().getName();
        System.out.println(name + "来了");
        /*
         * 人都是不患寡而患不均，当锁对象对每个人相同时，每个人都遵守规则:别人在用时，其他人不进去
         * 但这是不现实的，这种范围不会这么大，只能是一部分人，所以当锁对象对于某一部分人相同时，这部分人遵守，其他人不守
         * */
        if (this.money >= money) {
            System.out.println(name + "取出了" + money + "钱");//此语句有点耗时，当有个线程跑到这里，大概率要让出
cpu，然后另一个线程也会跑到这；此时就有可能出事
            this.money -= money;
            System.out.println(name + "还剩" + this.money + "钱");
        } else {
            System.out.println("没钱");
        }

    }
}
```

```
母来了
母取出了10000钱
母还剩0钱
公来了
没钱
```

- 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。但是代码要【高度面向对象】！
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。

- 同步方法与同步代码块对比

同步代码块锁的范围更小，同步方法锁的范围更大

- 同步方法是如何保证线程安全的

对出现问题的核心方法使用synchronized修饰
每次只能一个线程占锁进入访问

- 同步方法的同步锁对象的原理

对于实例方法默认使用this作为锁对象。
对于静态方法默认使用类名.class对象作为锁对象

- java中对象锁与类锁的区别

```
/*
 *
 对象锁和【类锁】 全局锁的关系？
        对象锁是用于对象实例方法，或者一个对象实例上的    this
        类锁是用于类的静态方法或者一个类的class对象上的。 Ag.class

        我们知道，类的对象实例可以有很多个，但是每个类只有一个class对象，
        所以不同对象实例的对象锁是互不干扰的，但是每个类只有一个类锁。
 */
 public class SynchrDemo {
    public static void main(String[] args) {

        Thread1 thread1 = new Thread1();
        Thread1 thread2 = new Thread1();

        thread1.start();
        thread2.start();
    }
}
```

```
class Ag{
    public void show(){
        // this 是当前对象的实例，由于新建两个对象，不是同一对象。所以这里是锁不住的。    代码快的方式，比修饰在方法上更细化控制。
        synchronized (this) {
            for (int i = 0; i < 4; i++) {
                System.out.println(Thread.currentThread().getName() + " i=" + i);
            }
        }
    }

}
```

```
class Thread1 extends Thread{
    @Override
    public void run() {
        //这里是新建对象   主方法中new了两个thread，就是new了两个Ag对象
        Ag ag = new Ag();
        ag.show();

    }
}
```

- 小结

一 Java中的关键字 synchronized 是啥?
synchronized是Java提供的一个并发控制的关键字

用法：同步方法 和 同步代码块。
可以修饰方法 也可以 修饰代码块

作用： 被synchronized修饰的代码块及方法，在同一时间，只能被单个线程访问。【保证线程安全】

1 修饰方法和代码块有什么不同?
二者的结果是一样的
修饰方法时，作用域是整个方法，控制的范围大

synchronized 代码块 可控制具体的作用域，更精准控制提高效率。
减少阻塞带来的时间问题。

2 同步锁的给谁用的?
同步锁基于对象，只要锁的来源一致，即可达到同步的作用。
所以，但对象不一样，则不能达到同步效果

3 synchronized修饰方法，代码块，锁未释放，此时，其他线程调用同一对象的其他被synchronized修饰的方法，代码块，会如何?
当线程 A 调用某对象的synchronized 方法 或者 synchronized 代码块时，若同步锁未释放，
其他线程调用同一对象的其他 synchronized 方法 或者 synchronized 代码块时将被阻塞，直至线程 A 释放该对象的同步锁。（注意：重点是其他）

4 调用非synchronized方法 ，代码快呢?
当线程 A 调用某对象的synchronized 方法 或者 synchronized 代码块时，无论同步锁是否释放，
其他线程调用同一对象的其他 非 synchronized 方法 或者 非 synchronized 代码块时可立即调用

5 全局锁如何实现?
全局锁： 锁住整个 Class，而非某个对象或实例。1-4都是实例锁
实现： 静态 synchronized 方法

static 声明的方法为全局方法，与对象实例化无关，所以 static synchronized 方法为全局同步方法，与对象实例化无关。

6 synchronized 具体 Class 的代码块?
synchronized (Ag.class) 获得的同步锁是全局的，
static synchronized 获得的同步锁也是全局的，同一个锁，所以达到同步效果。

7 对象锁和【类锁】 全局锁的关系?
对象锁是用于对象实例方法，或者一个对象实例上的 this
类锁是用于类的静态方法或者一个类的class对象上的。 Ag.class

我们知道，类的对象实例可以有很多个，但是每个类只有一个class对象，

所以不同对象实例的对象锁是互不干扰的，但是每个类只有一个类锁。

方式三：Lock锁

- 概述

为了更清晰的表达如何加锁和释放锁，JDK5以后提供了一个新的锁对象Lock，更加灵活、方便。
Lock的使用比synchronized方法和语句更灵活
Lock是接口不能直接实例化，这里采用它的实现类ReentrantLock来构建Lock锁对象。

- 实现类的无参构造

方法名称	说明
public ReentrantLock()	获得Lock锁的实现类对象

Lock的API

方法名称	说明
void lock()	获得锁
void unlock()	释放锁

```java
public class SaveMethod {
    public static void main(String[] args) {
        final Card icbc = new Card("ICBC", 10000);
        new DrawThread(icbc, "母").start();
        new DrawThread(icbc, "公").start();
    }
}
```

```java
class DrawThread extends Thread {
    private Card card;

    public DrawThread() {

    }

    public DrawThread(Card card, String name) {
        super(name);
        this.card = card;
    }

    @Override
    public void run() {
        card.drawMoney(10000);
    }
}
```

```java
class Card {
    private String cardId;
    private Integer money;

    /*
     * Lock是锁接口，ReentrantLock是他的实现类
     * 建议锁的实例与共享资源同时实例
     * 这样这个资源自身就带着一把锁，这把锁仅对这个对象有用
     *使用final是为了防止有人想要修改锁，使它不能锁东西
     * */
    private final Lock lock = new ReentrantLock();

    public String getCardId() {
        return cardId;
    }

    public void setCardId(String cardId) {
        this.cardId = cardId;
    }

    public Integer getMoney() {
        return money;
    }

    public void setMoney(Integer money) {
        this.money = money;
    }

    public Card() {
    }

    public Card(String cardId, Integer money) {
        this.cardId = cardId;
        this.money = money;
    }

    public void drawMoney(int money) {
        final String name = Thread.currentThread().getName();
        System.out.println(name + "来了");
        lock.lock();
        try {
            if (this.money >= money) {
                System.out.println(name + "取出了" + money + "钱");//此语句有点耗时，当有个线程跑到这里，大概率要让出cpu，然后另一个线程也会跑到这；此时就有可能出事
                this.money -= money;
                System.out.println(name + "还剩" + this.money + "钱");
            } else {
                System.out.println("没钱");
            }
        } finally {
            lock.unlock();
        }

    }
}
```

```
母来了
公来了
母取出了10000钱
母还剩0钱
没钱
```

线程通信

- **什么是线程通信（实现线程间同步）**

所谓线程通信就是线程间相互发送数据，线程间共享一个资源即可实现线程通信

- **线程通信常见形式**

通过共享一个数据的方式实现。
根据共享数据的情况决定自己该怎么做，以及通知其他线程怎么做
synchronized wait、notify、notifAll
lock await async

- **线程通信实际应用场景**

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产生的数据。
要求：生产者线程生产完数据后唤醒消费者，然后等待自己，消费者消费完该数据后唤醒生产者，然后等待自己

- 响铃接电话模型

```
public class Phone {
    private boolean flags = false;//假为响铃，真为打电话

    public static void main(String[] args) {
        new Phone().run();
    }

    public void run() {
        new Thread(() -> {
            while (true) {
                try {
                    //这里的this是指调用该方法者，反正不是Phone；又由于这里是匿名内部类，可以访问外部类:外部类名.this拿到外部类对象
                    synchronized (Phone.this) {
                        if (flags) {
                            //进不来
                        } else {
                            System.out.println("钉钉d=====(￣▽￣*)b");
                            flags = true;
                            Phone.this.notify();//先唤醒别人，再睡；不然都睡了【只有锁对象知到该唤醒谁】
                            Phone.this.wait();//【由锁对象来是自己等待】等待会释放锁
                        }
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();

        new Thread(() -> {
            while (true) {
                try {
                    synchronized (Phone.this) {//使用手机实例作为锁对象，因为该生产者消费者模型是针对一部手机的
                        if (flags) {
                            System.out.println("通话中~~~");
                            Thread.sleep(2000);//睡眠不会释放锁
                            flags = false;
                            Phone.this.notify();
                            Phone.this.wait();
                        } else {
                            Phone.this.notify();
                            Phone.this.wait();
                        }
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }
}
```

```
钉钉d=====(￣▽￣*)b
通话中~~~
钉钉d=====(￣▽￣*)b
通话中~~~
钉钉d=====(￣▽￣*)b
通话中~~~
```

- 线程通信的前提

线程通信通常是在多个线程操作同一个共享资源的时候需要进行通信，且要保证线程安全

- **线程通信的三个常见方法**

Object类的等待和唤醒方法：

方法名称	说明
void wait()	让当前线程等待并释放所占锁，直到另一个线程调用notify()方法或 notifyAll()方法
void notify()	唤醒正在等待的单个线程
void notifyAll()	唤醒正在等待的所有线程

- 注意

上述方法应该使用当前同步锁对象进行调用

线程池

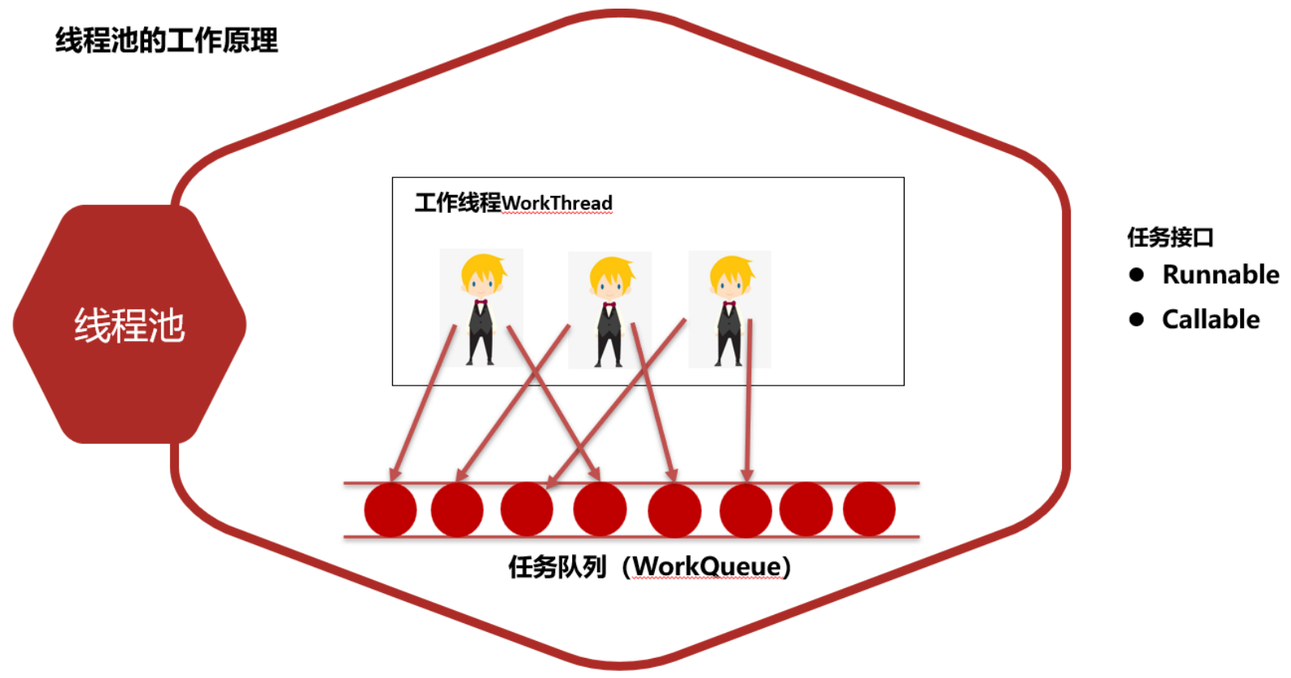
线程池概述

- **什么是线程池**

线程池就是一个可以复用线程的技术

- **不使用线程池的问题**

如果用户每发起一个请求，后台就创建一个新线程来处理，下次新任务来了又要创建新线程，而创建新线程的开销是很大的，这样会严重影响系统的性能



线程池实现的API、参数说明

- **谁代表线程池**

JDK 5.0起提供了代表线程池的接口：ExecutorService

- **如何得到线程池对象**

方式一：使用ExecutorService的实现类ThreadPoolExecutor自创建一个线程池对象(常用+常问)
方式二：使用Executors（线程池的工具类）调用方法返回不同特点的线程池对象

- **ThreadPoolExecutor构造器的参数说明**

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

```
参数一：指定线程池的线程数量（核心线程）：  corePoolSize  也称为工作线程，是在线程池中长久存在的线程                不能小于0
参数二：指定线程池可支持的最大线程数：  maximumPoolSize 最大线程池数=工作线程数+临时线程数                        最大数量 >=
核心线程数量
参数三：指定临时线程的最大存活时间：  keepAliveTime     这里指的是临时线程不进行工作后最大的存活时间              不能小于0
参数四：指定存活时间的单位(秒、分、时、天)：  unit                                                          时间单位
参数五：指定任务队列：  workQueue                                                                          不能为null
参数六：指定用哪个线程工厂创建线程：  threadFactory                                                         不能为null
参数七：指定线程忙，任务满的时候，新任务来了怎么办：  handler                                               不能为null
```

- **临时线程什么时候创建**

新任务提交时发现核心线程都在忙，任务队列也满了，并且还可以创建临时线程，此时才会创建临时线程

- **什么时候会开始拒绝任务**

核心线程和临时线程都在忙，任务队列也满了，新的任务过来的时候才会开始任务拒绝

线程池处理Runnable任务

- **ThreadPoolExecutor**创建线程池对象示例

```
ExecutorService pools = new ThreadPoolExecutor(3, 5
                        , 8 , TimeUnit.SECONDS, new ArrayBlockingQueue<>(6),
                        Executors.defaultThreadFactory() , new ThreadPoolExecutor.AbortPolicy());
```

ExecutorService的常用方法

方法名称	说明
void execute(Runnable command)	执行任务/命令，没有返回值，一般用来执行 Runnable 任务
Future<T> submit(Callable<T> task)	执行任务，返回未来任务对象获取线程结果，一般拿来执行 Callable 任务
void shutdown()	等任务执行完毕后关闭线程池
List<Runnable> shutdownNow()	立刻关闭，停止正在执行的任务，并返回队列中未执行的任务

新任务拒绝策略

策略	详解
ThreadPoolExecutor.AbortPolicy	丢弃任务并抛出RejectedExecutionException异常。**是默认的策略**
ThreadPoolExecutor.DiscardPolicy：	丢弃任务，但是不抛出异常 这是不推荐的做法
ThreadPoolExecutor.DiscardOldestPolicy	抛弃队列中等待最久的任务 然后把当前任务加入队列中
ThreadPoolExecutor.CallerRunsPolicy	由主线程负责调用任务的run()方法从而绕过线程池直接执行

- 线程池如何处理Runnable任务

```
使用ExecutorService的方法 ： void  execute(Runnable target)
executorService.executor(target);
```

- 线程池处理Runnable任务

```
public class ThreadPoolDemo1 {

    public static void main(String[] args) {


        /*
       public ThreadPoolExecutor(int corePoolSize,
                                 int maximumPoolSize,
                                 long keepAliveTime,
                                 TimeUnit unit,
                                 BlockingQueue<Runnable> workQueue,
                                 ThreadFactory threadFactory,
                                 RejectedExecutionHandler handler)
        * */
        ExecutorService executorService = new ThreadPoolExecutor(3, 5, 4
                , TimeUnit.SECONDS, new ArrayBlockingQueue<>(5), Executors.defaultThreadFactory()
                , new ThreadPoolExecutor.AbortPolicy());

        final MyRunnable target = new MyRunnable();

        executorService.execute(target);//1
        executorService.execute(target);//2
        executorService.execute(target);//3 三个核心线程在忙
        executorService.execute(target);//4
        executorService.execute(target);//5
        executorService.execute(target);//6
        executorService.execute(target);//7
        executorService.execute(target);//8 任务队列可以存放五个任务 【此时·还不会创建临时线程】
        executorService.execute(target);//9 此时核心线程在忙，任务队列已满，未达到最大线程数量，再来一个线程就开启临时线
程

        //10 上面在满的情况下加入了一个，然后又处理了一个；相当于还是处于满的状态；此时再来一个线程，判断核心线程在忙？，任务队
列已满？，未达到最大线程数量？创建一个新的临时线程
        executorService.execute(target);//10

        //11 java.util.concurrent.RejectedExecutionException
        executorService.execute(target);//11 此时 所有线程在忙，任务队列已满，再来一个任务，就会采用线程池配置的策略（报
错！！）



        /*
         * 这行代码无法执行，上面抛异常了；但jvm没有挂，说明线程池有捕获异常的机制；
         * try catch 中try语句块出错后的代码无法执行，
         * */
        executorService.shutdownNow();//即时任务没有完成，也要立即关闭

        executorService.shutdown();//等待所有任务执行完毕后，关闭
    }


}
```

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        try {
            final String name = Thread.currentThread().getName();
            for (int i = 0; i < 4; i++) {
                System.out.println(name + "输出了---->" + i);
            }
            System.out.println(name + "在忙~~~");
            Thread.sleep(2000000);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

线程池处理Callable任务

ExecutorService的常用方法

方法名称	说明
void execute(Runnable command)	执行任务/命令，没有返回值，一般用来执行 Runnable 任务
Future<T> submit(Callable<T> task)	执行Callable任务，返回未来任务对象获取线程结果
void shutdown()	等任务执行完毕后关闭线程池
List<Runnable> shutdownNow()	立刻关闭，停止正在执行的任务，并返回队列中未执行的任务

- 线程池如何处理Callable任务

使用ExecutorService的方法：
Future<T> submit(Callable<T> command)
并得到任务执行完后返回的结果。

```
Future f = pool.submit(mycallable);
f.get();
```

Executors工具类实现线程池

- **Executors**得到线程池对象的常用方法

ExecutorService的常用方法

方法名称	说明
void execute(Runnable command)	执行任务/命令，没有返回值，一般用来执行 Runnable 任务
Future<T> submit(Callable<T> task)	执行Callable任务，返回未来任务对象获取线程结果
void shutdown()	等任务执行完毕后关闭线程池
List<Runnable> shutdownNow()	立刻关闭，停止正在执行的任务，并返回队列中未执行的任务

- 注意

Executors的底层其实也是基于线程池的实现类ThreadPoolExecutor创建线程池对象的

- **Executors**使用可能存在的陷阱

方法名称	存在问题
public static ExecutorService newFixedThreadPool(int nThreads)	允许请求的任务队列长度是Integer.MAX_VALUE，可能出现OOM错误（java.lang.OutOfMemoryError）
public static ExecutorService newSingleThreadExecutor()	
public static ExecutorService newCachedThreadPool()	创建的线程数量最大上限是Integer.MAX_VALUE， 线程数可能会随着任务1:1增长，也可能出现OOM错误 （java.lang.OutOfMemoryError）
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)	

阿里巴巴 Java 开发手册

4．【强制】线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

说明：Executors 返回的线程池对象的弊端如下：

1）FixedThreadPool 和 SingleThreadPool：

允许的请求队列长度为 Integer.MAX_VALUE，可能会堆积大量的请求，从而导致 OOM。

2）CachedThreadPool 和 ScheduledThreadPool：

允许的创建线程数量为 Integer.MAX_VALUE，可能会创建大量的线程，从而导致 OOM。

- Executors是否适合做大型互联网场景的线程池方案

不合适。
建议使用ThreadPoolExecutor来指定线程池参数，这样可以明确线程池的运行规则，规避资源耗尽的风险

定时器

- 概念

定时器是一种控制任务延时调用，或者周期调用的技术

- **定时器的实现方式**

方式一：Timer
方式二： ScheduledExecutorService

Timer定时器

构造器	说明
public Timer()	创建Timer定时器对象

方法	说明
public void schedule(TimerTask task, long delay, long period)	开启一个定时器，按照计划处理TimerTask任务

- Timer实现定时器

```
public class TimerDemo1 {
    public static void main(String[] args) {
        final Timer timer = new Timer();//一个timer线程

        timer.schedule(new TimerTask() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName() + "AAAAAA" + new Date());
                try {
                    Thread.sleep(5000);//当在一个任务中执行的时间过长，就会影响另一个任务；
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, 3000, 2000);


        timer.schedule(new TimerTask() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName() + "BBBBBB" + new Date());
            }
        }, 3000, 2000);

        //当有一个任务出现异常，该定时器实例就会被干掉；影响其他线程
        timer.schedule(new TimerTask() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName() + "CCCC" + new Date());
                int c = 9 / 0;
            }
        }, 3000, 2000);
    }
}
```

- **Timer**定时器的特点和存在的问题

1、Timer是单线程，处理多个任务按照顺序执行，存在延时与设置定时器的时间有出入。
2、可能因为其中的某个任务的异常使Timer线程死掉，从而影响后续任务执行

ScheduledExecutorService定时器

- 概述

ScheduledExecutorService是 jdk1.5中引入了并发包，目的是为了弥补Timer的缺陷, ScheduledExecutorService内部为线程池。

Executors的方法	说明
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)	得到线程池对象

ScheduledExecutorService的方法	说明
public ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)	周期调度方法

用Executors创建线程池容易出问题的呀，这里的 newScheduledThredPool(i)
可以创建几乎无限个线程，且任务队列的大小也是无限；会出现oom异常

- **ScheduledExecutorService**的优点

基于线程池，某个任务的执行情况不会影响其他定时任务的执行

- **ScheduledExecutorService**定时器

```
import java.util.TimerTask;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;


public class ScheduleExecutorServiceDemo {

    public static void main(String[] args) {
        final ScheduledExecutorService schedulePool = Executors.newScheduledThreadPool(2);
        schedulePool.scheduleWithFixedDelay(() -> {
            System.out.println(Thread.currentThread().getName() + "AAA " + new Date());
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, 0, 2, TimeUnit.SECONDS);
```

```
        schedulePool.scheduleWithFixedDelay(() -> {
            System.out.println(Thread.currentThread().getName() + "BBB " + new Date());
        }, 0, 2, TimeUnit.SECONDS);

        schedulePool.scheduleWithFixedDelay(() -> {
            System.out.println(Thread.currentThread().getName() + "ccc " + new Date());
            System.out.println(9 / 0);//出现异常，线程池内部会处理掉，不会影响其他的线程
        }, 0, 2, TimeUnit.SECONDS);
    }
}
```

```
pool-1-thread-2BBB Tue Mar 15 19:25:19 CST 2022
pool-1-thread-1AAA Tue Mar 15 19:25:19 CST 2022    19
pool-1-thread-2ccc Tue Mar 15 19:25:19 CST 2022 出现异常，线程池内部会处理掉，不会影响其他的线程，该线程挂了
pool-1-thread-2BBB Tue Mar 15 19:25:21 CST 2022
pool-1-thread-2BBB Tue Mar 15 19:25:23 CST 2022
pool-1-thread-2BBB Tue Mar 15 19:25:25 CST 2022
pool-1-thread-1AAA Tue Mar 15 19:25:26 CST 2022    26 19+5=24（该任务执行时间）等待两秒开始继续执行该任务
pool-1-thread-2BBB Tue Mar 15 19:25:27 CST 2022
pool-1-thread-2BBB Tue Mar 15 19:25:29 CST 2022
```

并发、并行

- 概念

正在运行的程序（软件）就是一个独立的进程， 线程是属于进程的，多个线程其实是并发与并行同时进行的；
进程：运行起来的程序；进程是资源调度的基本单位，线程是cpu调度的基本单位；

- 并发理解

CPU同时处理线程的数量有限。
CPU会轮询为系统的每个线程服务，由于CPU切换的速度很快，给我们的感觉这些线程在同时执行，这就是并发；
并发是同一时刻，cpu只为一条线程服务；
并行是同一时刻，cpu能为多条线程服务

线程状态

- 概念

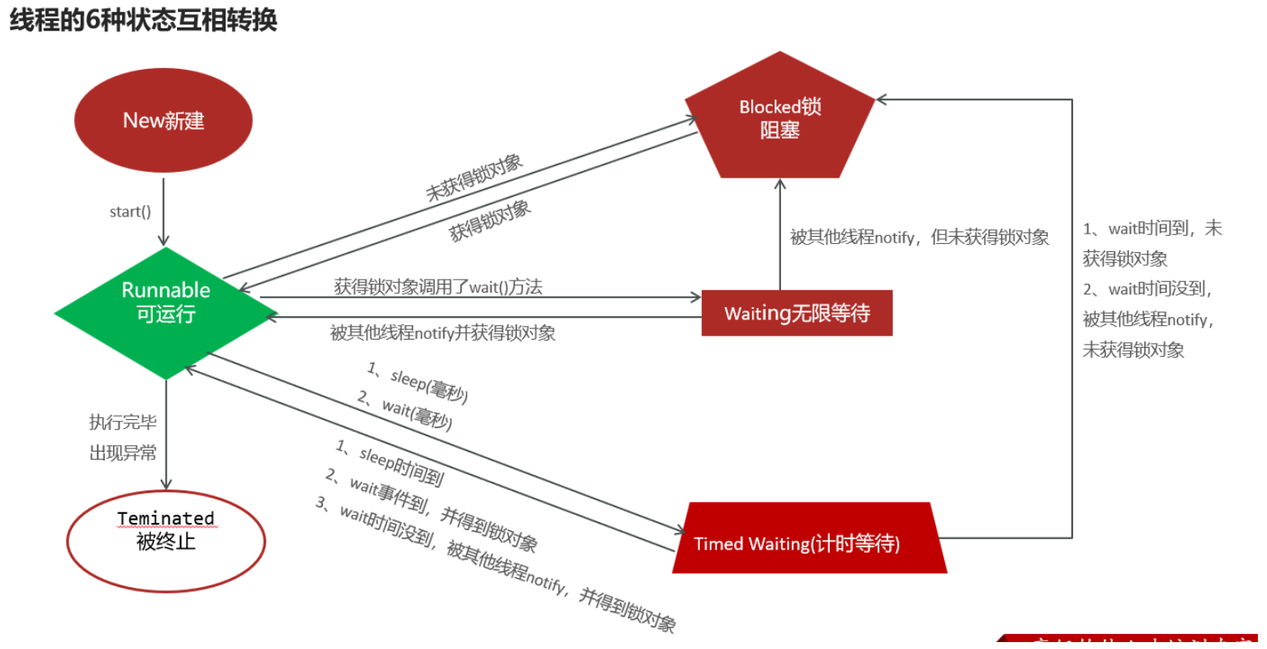
线程的状态：也就是线程从生到死的过程，以及中间经历的各种状态及状态转换。
理解线程的状态有利于提升并发编程的理解能力。
Java总共定义了6种状态
6种状态都定义在Thread类的内部枚举类中

- 枚举

```
public class Thread{
     ...
        public enum State {
        NEW,
        RUNNABLE,
        BLOCKED,
        WAITING,
        TIMED_WAITING,
        TERMINATED;
    }
    ...
}
```

- 线程的六种状态

线程的6种状态互相转换



1.NEW(新建)　　线程刚被创建，只有java的对象特征没有线程特征

2.Runnable(可运行)　线程已经调用了start()等待CPU调度

3.Blocked(锁阻塞)　线程在执行的时候未竞争到锁对象，则该线程进入Blocked状态

4.Waiting(无限等待)　该线程被锁对象调用了wait()方法；当一个线程进入Waiting状态，只有当另一个线程调用notify或者notifyAll方法才能够唤醒

5.Timed Waiting(计时等待) 同waiting状态，有几个方法有超时参数，调用他们将进入Timed Waiting状态。
　　带有超时参数的常用方法有Thread.sleep(毫秒值) 、Object.wait(毫秒值)
　　sleep(毫秒值)不会释放锁，wait(毫秒值)会释放锁；
　　wait(毫秒值)不会进入无限等待，等指定毫秒值的时间后就会自己唤醒自己，然后看有没有锁，没有进入阻塞状态;被人唤醒的话可以提前醒来;

6.Teminated(被终止) 因为run方法正常退出而死亡，或者因为没有捕获的异常终止了run方法而死亡