多线程

1.进程

> 进程：进行中的应用程序 属于CPU分配资源的最小单位

2. 线程

> 线程：包含在进程之内 属于CPU调度执行的最小单位

3. 进程和线程的关系

> 线程是包含在进程之内的，一个进程至少包含一个线程，否则将无法执行。

4. 线程是否为越多越好?

> 不是的，要结合实际的硬件环境而言。

5.关于多核心

> 多核心是将一个CPU分成多部分，同时协同分工合作，提高用户体验。

6. 关于线程的执行

> 在单核心CPU下，多个线程是轮流交替执行的，而非并行执行。
>
> 底层是以时间片来切换多个线程的，每个线程最多执行20ms，执行完以后切换下一个线程，如此随机轮流交替执行，所以我们看上去，就好像是同时执行的，其实是轮流交替执行的。

7. 并发和并行

> 并发：同时发生，轮流交替执行。
>
> 并行：真正意义上的同时执行。

8.主线程

> Thread 类 线程类
>
> currentThread() 获取当前正在执行的线程对象

> getName() 获取当前线程名称
>
> setName(String name) 设置当前线程名称

```
package com.atguigu.test1;

/**
 *  Thread 类 线程类
 *  currentThread() 获取当前正在执行的线程对象
 *  getName() 获取当前线程名称
 *  setName(String name) 设置当前线程名称
 */
public class TestThread {
    public static void main(String[] args) {
        Thread thread = Thread.currentThread();

        System.out.println(thread.getName());

        thread.setName("主线程");

        System.out.println(thread.getName());




    }
}
```

9.创建线程

> 创建线程方式1：继承Thread
>
> run() 方法中的内容为子线程最终要执行的内容
>
>
> 面试题：调用start方法调用run方法的区别?
>
> 调用start方法会开启新的线程
>
> 调用run方法 属于使用main线程调用run方法 不会开启新的线程

```
package com.atguigu.test1;

/**
 *  创建线程方式1：继承Thread
 *  run() 方法中的内容为子线程最终要执行的内容
 *
 *  面试题：调用start方法调用run方法的区别?
 *  调用start方法会开启新的线程
 *  调用run方法 属于使用main线程调用run方法 不会开启新的线程
 */
```

```
public class MyThread1 extends Thread{
    @Override
    public void run() {
        for(int i = 0;i < 10;i++){
            System.out.println(Thread.currentThread().getName() + "----" + i);
        }
    }

    public static void main(String[] args) {
        MyThread1 th1 = new MyThread1(); // 创建线程对象
        th1.setName("线程A");
        th1.start(); // 启动线程

//        th1.run();

        MyThread1 th2 = new MyThread1();
        th2.setName("=====线程B=====");
        th2.start();

//        th2.run();
    }




}
```

> 创建线程方式2： 实现Runnable接口 重写 run方法
>
> Thread(Runnable runnable) 传入一个Runnable实现类构造Thread对象
>
> Thread(Runnable runnable, String name) 传入一个Runnable实现类 和 线程名称 构造Thread对象

```
package com.atguigu.test1;

/**
 *  创建线程方式2 ： 实现Runnable接口 重写 run方法
 *  Thread(Runnable runnable) 传入一个Runnable实现类构造Thread对象
 *  Thread(Runnable runnable, String name) 传入一个Runnable实现类 和 线程名称  构造Thread对象
 */
public class RunnableImpl implements Runnable{
    @Override
    public void run() {
        for(int i = 1;i < 10;i++){
            System.out.println(Thread.currentThread().getName() + "---" +i);
        }
    }

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

```
        RunnableImpl runnable = new RunnableImpl();

        Thread th1 = new Thread(runnable);

        th1.setName("线程A");

        th1.start();

        Thread th2 = new Thread(runnable, "***线程B***");

        th2.start();

    }


}
```

10.线程的状态

> 线程的状态： 创建 ---》就绪 ----》运行 ----》阻塞 ----》 死亡

```
package com.atguigu.test2;

/**
 *  线程的状态 ： 创建 ---》就绪 ----》运行 ----》阻塞 ----》 死亡
 *
 */
public class TestThreadStatus extends Thread{
    @Override
    public void run() { // 运行
        System.out.println("run方法开始执行");
        try {
            Thread.sleep(3000); // 阻塞
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println(Thread.currentThread().getName());
    } // 死亡


    public static void main(String[] args) {
        TestThreadStatus th1 = new TestThreadStatus(); // 创建

        th1.start(); // 就绪
    }
}
```

11.线程的礼让

> 线程的礼让：暂停当前线程 ， 礼让其他线程先执行，但是不保证其他线程一定会先执行
>
> yield()
>
> 线程插队和礼让的区别?
>
> 插队表示将当前线程停止执行 一定会先执行其他的插队线程
>
> 而礼让表示当前线程愿意做出使用CPU的让步 优先执行其他线程 但是CPU可以忽略这个让步 所以 不一定其他线程
>
> 会要优先执行

```
package com.atguigu.test2;

/**
 *  线程的礼让：暂停当前线程 ， 礼让其他线程先执行，但是不保证其他线程一定会先执行
 *  yield()
 *
 *  线程插队和礼让的区别?
 *  插队表示将当前线程停止执行 一定会先执行其他的插队线程
 *  而礼让表示当前线程愿意做出使用CPU的让步  优先执行其他线程 但是CPU可以忽略这个让步 所以 不一定其他线程
 *  会要优先执行
 *
 *
 */
public class TestThreadYield extends Thread{
    @Override
    public void run() {
        for(int i = 1;i <= 20;i++){
            if(i == 5){
                System.out.println("线程礼让");
                Thread.yield();
            }
            System.out.println(Thread.currentThread().getName()  + "===" + i);
        }
    }

    public static void main(String[] args) {
        TestThreadYield th1 = new TestThreadYield();
        th1.setName("*****线程A*****");
        th1.start();


        TestThreadYield th2 = new TestThreadYield();
        th2.setName("线程B");
        th2.start();
    }


}
```

12.线程的插队

> 线程的插队：当前线程等待插队线程先执行完毕 或者 等待指定之间 然后当前线程再执行
>
> public final void join() 等待插队线程执行完毕
>
> public final void join(long mills) 等待插队线程执行指定时间
>
> public final void join(long mills,int nanos)等待插队线程执行指定时间

```
package com.atguigu.test2;

/**
 *  线程的插队：当前线程等待插队线程先执行完毕 或者 等待指定之间 然后当前线程再执行
 *  public final void join() 等待插队线程执行完毕
 *  public final void join(long mills) 等待插队线程执行指定时间
 *  public final void join(long mills,int nanos)等待插队线程执行指定时间
 */
public class TestThreadJoin extends Thread{

    @Override
    public void run() {
        for(int i = 1;i <= 10;i ++){
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            System.out.println(Thread.currentThread().getName() + "----" + i);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        TestThreadJoin th1 = new TestThreadJoin();

        th1.setName("*****线程A*****");

        th1.start();

        for(int i = 1;i < 20;i++){
            if(i == 10){
                System.out.println("子线程插队");
                th1.join(600);
            }
            System.out.println(Thread.currentThread().getName() + "----" + i);
        }



    }


}
```

13.线程的优先级

> 线程的优先级：线程优先级 从1~ 10 1最低 10最高 默认为5
>
> 优先级高的线程获取CPU资源的概率较大 并不一定能够保证绝对的优先执行

```
package com.atguigu.test2;

/**
 *  线程的优先级：线程优先级 从1~ 10  1最低 10最高 默认为5
 *  优先级高的线程获取CPU资源的概率较大 并不一定能够保证绝对的优先执行
 *
 */
public class TestThreadPriority extends Thread{
    @Override
    public void run() {
        for(int i = 1;i <= 10;i++){
            System.out.println(Thread.currentThread().getName() + "===" + i);
        }
    }

    public static void main(String[] args) {
        TestThreadPriority th1 = new TestThreadPriority();
        System.out.println(th1.getPriority());
        th1.setName("线程A");
        th1.setPriority(MIN_PRIORITY); // 1
        th1.start();


        TestThreadPriority th2 = new TestThreadPriority();
        System.out.println(th2.getPriority());
        th2.setName("******线程B******");
        th2.setPriority(Thread.MAX_PRIORITY); // 10
        th2.start();

    }


}
```

14.线程安全

> 我们可以通过Java中的synchronized关键字实现线程安全
>
> synchronized 单词释义： 同步

> synchronized 用于修饰方法 或者 代码块
>
> 分别表示在同一时间只能有一个线程 访问这个方法 或者代码块 其他的线程排队 等待前边的线程执行完毕 再继续执行
>
> 同步代码块中小括号内书写this 也可以书写其他的对象 如果需要多个线程同步
>
> 则必须保证锁定的是同一个对象才可以

```
package com.atguigu.test4;

/**
 *  使用多线程模拟抢票   三个人 抢十张票
 *  必须保证：不能超卖 不能重卖
 *
 *  目前所存在的线程安全问题原因：
 *  当一个条件成立的情况下 可能会进入到循环中多个线程
 *
 *  解决方案：我们应该保证同一时间只能有一个线程访问买票的逻辑代码    其他的线程排队等待
 *  等待前边的线程买票完毕 后续的线程再继续买票
 *
 *  具体方法：我们可以通过Java中的synchronized关键字实现线程安全
 *  synchronized 单词释义：  同步
 *  synchronized 用于修饰方法 或者 代码块
 *  分别表示在同一时间只能有一个线程 访问这个方法 或者代码块   其他的线程排队 等待前边的线程执行完毕 再继续执行
 *
 *
 *  手写同步关键字：synchronized
 *
 *  同步代码块中小括号内书写this 也可以书写其他的对象 如果需要多个线程同步
 *  则必须保证锁定的是同一个对象才可以
 */
public class BuyTicket2 implements Runnable{
    private int ticketCount = 10;

    private Object obj = new Object();

    @Override
    public void run() {
        while(true){
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            synchronized (this) {
                if (ticketCount == 0) {
                    break;
                }
                ticketCount--;
                System.out.println(Thread.currentThread().getName() + "抢到第"+ (10 - ticketCount)
+ "张票，还剩余" + ticketCount +"张票");
            }
        }
```

```
    }
    public static void main(String[] args) {
        BuyTicket2 runnable = new BuyTicket2();

        Thread th1 = new Thread(runnable,"赵四");
        Thread th2 = new Thread(runnable,"广坤");
        Thread th3 = new Thread(runnable,"大拿");

        th1.start();
        th2.start();
        th3.start();
    }


}
```

```
package com.atguigu.test4;

import java.util.Hashtable;
import java.util.Vector;

/**
 * 使用多线程模拟抢票  三个人 抢十张票
 * 必须保证：不能超卖 不能重卖
 * <p>
 * 目前所存在的线程安全问题原因：
 * 当一个条件成立的情况下 可能会进入到循环中多个线程
 * <p>
 * 解决方案：我们应该保证同一时间只能有一个线程访问买票的逻辑代码    其他的线程排队等待
 * 等待前边的线程买票完毕 后续的线程再继续买票
 * <p>
 * 具体方法：我们可以通过Java中的synchronized关键字实现线程安全
 * synchronized 单词释义：  同步
 * synchronized 用于修饰方法 或者 代码块
 * 分别表示在同一时间只能有一个线程 访问这个方法 或者代码块  其他的线程排队 等待前边的线程执行完毕 再继续执行
 *
 * 回顾我们之前所接触线程安全的类 都是使用synchronized关键字修饰方法实现线程安全的
 * StringBuffer Hashtable Vector
 *
 */
public class BuyTicket3 implements Runnable {

    private int ticketCount = 10;

    @Override
    public synchronized void run() {
        while(ticketCount > 0){
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
```

```
            ticketCount--;
            System.out.println(Thread.currentThread().getName() + "抢到第" + (10 - ticketCount) + "张
票，还剩余" + ticketCount + "张票");
        }
    }

    public static void main(String[] args) {
        BuyTicket3 runnable = new BuyTicket3();

        Thread th1 = new Thread(runnable, "赵四");
        Thread th2 = new Thread(runnable, "广坤");
        Thread th3 = new Thread(runnable, "大拿");

        th1.start();
        th2.start();
        th3.start();
    }


}
```

15.生产者消费者模式

> 线程之间的通信：多个线程之间进行数据的传递 接收
>
> 生产者消费者模式(不属于设计模式)
>
> 1.生产什么 消费什么
>
> 2.持续生产 持续消费
>
> 3.没有生产 不能消费
>
> 4.不能重复消费
>
> 5.保证产品的完整性

```
package com.atguigu.test5;

/**
 *  线程之间的通信：多个线程之间进行数据的传递  接收
 *  生产者消费者模式(不属于设计模式)
 *  1.生产什么  消费什么
 *  2.持续生产  持续消费
 *  3.没有生产  不能消费
 *  4.不能重复消费
 *  5.保证产品的完整性
 */
public class Computer {
    private String mainFrame; // 电脑主机
    private String screen; // 显示器
```

```
    private boolean flag ; // 默认为false表示可以生产 不能消费 为true 可以消费 不能生产

    public boolean isFlag() {
        return flag;
    }

    public void setFlag(boolean flag) {
        this.flag = flag;
    }

    @Override
    public String toString() {
        return "Computer{" +
                "mainFrame='" + mainFrame + '\'' +
                ", screen='" + screen + '\'' +
                '}';
    }

    public Computer(String mainFrame, String screen) {
        this.mainFrame = mainFrame;
        this.screen = screen;
    }

    public Computer() {
    }

    public String getMainFrame() {
        return mainFrame;
    }

    public void setMainFrame(String mainFrame) {
        this.mainFrame = mainFrame;
    }

    public String getScreen() {
        return screen;
    }

    public void setScreen(String screen) {
        this.screen = screen;
    }
}
```

```
package com.atguigu.test6;

/**
 *  wait() 导致当前线程等待，直到另一个线程调用该对象的 notify()方法或 notifyAll()方法
 *  notify()  唤醒正在等待的线程 如果有多个线程正在等待 随机唤醒其中某一个正在等待的线程
 *
 *  notify()  和 notifyAll() 的区别?
 *      notify() 唤醒一个正在等待的线程 如果有多个线程正在等待 随机唤醒其中某一个正在等待的线程
```

```
 *      notifyAll() 唤醒所有正在等待的线程
 *
 *
 */
public class Producer extends Thread{
    private Computer computer;

    public Producer(Computer computer) {
        this.computer = computer;
    }

    @Override
    public void run() {
        for(int i = 1;i <= 20;i++){
            synchronized (computer) {
                if(computer.isFlag() == true){
                    try {
                        computer.wait();
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
                if(i % 2 == 0){
                    computer.setMainFrame(i + "号联想主机");
                    computer.setScreen(i + "号联想显示器");

                    System.out.println("生产了"+ i +"号联想电脑");
                }else{
                    computer.setScreen(i + "号华为显示器");
                    computer.setMainFrame(i + "号华为主机");

                    System.out.println("生产了"+ i +"号华为电脑");
                }

                computer.setFlag(true);
                computer.notify();
            }
        }
    }
}
```

```
package com.atguigu.test6;

public class Consumer extends Thread{
    private Computer computer;

    public Consumer(Computer computer) {
        this.computer = computer;
    }
```

```
    @Override
    public void run() {


        for(int i = 1;i <= 20;i++){
            synchronized (computer) {
                if(!computer.isFlag()){
                    try {
                        computer.wait();
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
                System.out.println(computer.getMainFrame() + "====" + computer.getScreen());
                computer.setFlag(false);
                computer.notify();
            }
        }
    }
}
```

```
package com.atguigu.test5;

public class Test {
    public static void main(String[] args) {
        Computer computer = new Computer();

        Producer producer = new Producer(computer);
        Consumer consumer = new Consumer(computer);

        producer.start();
        consumer.start();

    }
}
```

16.锁

> 锁 Lock
>
> 面试题：什么是CAS
>
> CAS属于乐观锁思想 当前线程认为其他任何线程都没有操作改变当前线程正在操作的数据
>
> 每次对数据进行操作先将数据进行读取 再操作 如果数据没有改变 则直接进行操作
>
> 如果数据已经改变 则再次读取 再次操作 重复这个过程
>
> 乐观锁 CAS Compare And Swap 自旋锁(无锁)

> 悲观锁 synchronized
>
> 同步锁 死锁(逻辑错误) 可重入锁 可重入锁之读锁 可重入锁之写锁
>
> 什么是可重入锁?
>
> 同一个线程对象 重复的获得一个锁对象 不应该产生死锁
>
> synchronized就属于可重入锁
>
> 什么是死锁?
>
> 因为逻辑错误 导致的多个线程 竞争同一资源 互不相让 僵持不下的一种现象
>
> 公平锁和非公平锁?
>
> 公平锁是指多个线程 按照某种规则轮流访问资源 都可以访问到指定资源对象
>
> 非公平锁表示没有任何规则 哪个线程抢占到资源就可以执行 容易产生某个线程一直占用资源的情况

> 使用多线程模拟抢票 三个人 抢十张票
>
> 必须保证：不能超卖 不能重卖
>
> Lock接口实现类 ReentrantLock 可重入锁
>
> lock() 方法表示上锁
>
> unlock() 表示解锁 解锁的操作通常放在finally代码块中

```
package com.atguigu.test1;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 *  使用多线程模拟抢票  三个人 抢十张票
 *  必须保证：不能超卖 不能重卖
 *
 *  Lock接口实现类 ReentrantLock  可重入锁
 *  lock() 方法表示上锁
 *  unlock() 表示解锁 解锁的操作通常放在finally代码块中
 *
 */
public class BuyTicket2 implements Runnable{
    private int ticketCount = 10;

    private Lock lock = new ReentrantLock();

    @Override
    public void run() {
        while(true){
            try {
```

```
                Thread.sleep(500);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            try {
                lock.lock();
                if (ticketCount == 0) {
                    break;
                }
                ticketCount--;
                System.out.println(Thread.currentThread().getName() + "抢到第"+ (10 - ticketCount) +
"张票，还剩余" + ticketCount +"张票");
            } catch (Exception e) {
                throw new RuntimeException(e);
            } finally {
                lock.unlock();
            }



            }




    }
    public static void main(String[] args) {
        BuyTicket2 runnable = new BuyTicket2();

        Thread th1 = new Thread(runnable,"赵四");
        Thread th2 = new Thread(runnable,"广坤");
        Thread th3 = new Thread(runnable,"大拿");

        th1.start();
        th2.start();
        th3.start();
    }


}
```

17 .其他创建线程的方式

> 回顾我们目前创建线程的方式存在哪些问题?
>
> 目前我们是通过继承Thread类 以及 实现 Runnable 接口的方式实现创建线程
>
> 1.这两种方式都无法在线程执行完毕以后 返回数据
>
> 2.这两种方式都无法抛出、声明异常

> 我们还可以使用Callable接口实现创建线程
>
> FutureTask 属于 RunnableFuture的实现类
>
> RunnableFuture接口 继承 自 Runnable接口
>
> 所以 FutureTask也属于 Runnable的实现类
>
> FutureTask构造方法支持传入一个Callable接口实现类
>
> 最终 FutureTask依然作为参数传入到Thread构造方法中
>
> FutureTask的泛型 要和 Callable的泛型保持一致

```
package com.atguigu.test2;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;

/**
 *  回顾我们目前创建线程的方式存在哪些问题?
 *  目前我们是通过继承Thread类 以及 实现 Runnable 接口的方式实现创建线程
 *  1.这两种方式都无法在线程执行完毕以后 返回数据
 *  2.这两种方式都无法抛出、声明异常
 *
 *  我们还可以使用Callable接口实现创建线程
 *
 *  FutureTask 属于 RunnableFuture的实现类
 *  RunnableFuture接口 继承 自  Runnable接口
 *  所以 FutureTask也属于 Runnable的实现类
 *
 *  FutureTask构造方法支持传入一个Callable接口实现类
 *
 *  最终 FutureTask依然作为参数传入到Thread构造方法中
 *
 *  FutureTask的泛型 要和 Callable的泛型保持一致
 *
 */
public class TestCreateThread implements Callable<Integer> {

    @Override
    public Integer call() throws Exception {
        System.out.println("线程名称: " + Thread.currentThread().getName() );
        return 100;
    }

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        TestCreateThread callable = new TestCreateThread();

        FutureTask<Integer> futureTask = new FutureTask<>(callable);

        Thread thread = new Thread(futureTask, "线程A");

        thread.start();
```

```
        System.out.println(futureTask.get());

    }

}
```

> 回顾我们目前所使用三种创建线程的方式
>
> 1.继承Thread类
>
> 2.实现Runnable接口
>
> 3.实现Callable接口
>
> 以上三种方式创建线程存在如下问题:
>
> 1.创建的线程都属于独立的对象 不能统一管理
>
> 2.如果在多线程场景下 独立的线程对象将会频繁的创建 和 销毁 影响程序性能
>
> 3.以上创建线程的方式不能执行一些定时任务
>
> 这些问题我们可以通过线程池解决
>
> 线程池属于是一个用于管理多个线程对象的容器 可以通过Executors这个类的静态方法创建
>
> 线程池中的线程对象 使用完毕以后 并不会立即回收 而是存放在线程池中 以便于后续继续使用
>
> static newCachedThreadPool() 根据当前任务需要创建一个不定数量线程的线程池
>
> static newFixedThreadPool(int nThreads) 创建一个指定线程数量的线程池
>
> static newScheduledThreadPool(int corePoolSize) 创建一个指定数量 并且可以执行定时任务的线程池
>
> static newSingleThreadExecutor() 创建一个单个线程的线程池
>
> ThreadPoolExecutor()构造方法参数描述
>
> corePoolSize 核心线程数
>
> keepAliveTime 保持空闲时间 60L 表示线程对象的空闲时间为60秒 在60秒之后没有任务 将被回收
>
> TimeUnit 时间单位 SECONDS

```
package com.atguigu.test2;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
```

```
import java.util.concurrent.TimeUnit;

/**
 *  回顾我们目前所使用三种创建线程的方式
 *  1.继承Thread类
 *  2.实现Runnable接口
 *  3.实现Callable接口
 *
 *
 *  以上三种方式创建线程存在如下问题:
 *  1.创建的线程都属于独立的对象 不能统一管理
 *  2.如果在多线程场景下 独立的线程对象将会频繁的创建 和 销毁 影响程序性能
 *  3.以上创建线程的方式不能执行一些定时任务
 *
 *  这些问题我们可以通过线程池解决
 *  线程池属于是一个用于管理多个线程对象的容器 可以通过Executors这个类的静态方法创建
 *  线程池中的线程对象 使用完毕以后 并不会立即回收 而是存放在线程池中 以便于后续继续使用
 *
 *  static newCachedThreadPool()  根据当前任务需要创建一个不定数量线程的线程池
 *  static newFixedThreadPool(int nThreads)  创建一个指定线程数量的线程池
 *  static newScheduledThreadPool(int corePoolSize) 创建一个指定数量 并且可以执行定时任务的线程池
 *  static newSingleThreadExecutor() 创建一个单个线程的线程池
 *
 *
 *  ThreadPoolExecutor()构造方法参数描述
 *  corePoolSize 核心线程数
 *  keepAliveTime 保持空闲时间  60L  表示线程对象的空闲时间为60秒 在60秒之后没有任务 将被回收
 *  TimeUnit 时间单位  SECONDS
 *
 *
 */
public class TestThreadPool {
    public static void main(String[] args) {
        ExecutorService es1 = Executors.newCachedThreadPool();


        es1.submit(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName() + "---hello 线程池1");
            }
        });

        es1.execute(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName() + "***hello 线程池1");
            }
        });


        ExecutorService es2 = Executors.newFixedThreadPool(10);
```

```
        es2.submit(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName() + "***hello 线程池2");
            }
        });


        es2.execute(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName() + "***hello 线程池2");
            }
        });


        ScheduledExecutorService es3 = Executors.newScheduledThreadPool(10);

        es3.schedule(new Runnable() {
            @Override
            public void run() {
                System.out.println("延迟执行: " + Thread.currentThread().getName() + "***hello 线程池
3");
            }
        }, 5, TimeUnit.SECONDS);

        ExecutorService es4 = Executors.newSingleThreadExecutor();

        es4.submit(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName() + "***hello 线程池4");
            }
        });


        es4.execute(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName() + "***hello 线程池4");
            }
        });



    }
}
```

18.递归

> 递归：将一个大的问题 拆分为若干个小的问题 逐步解决 必须有正确的出口
>
> 举例： 赵四 广坤 小宝 大拿
>
> 递归可以解决哪些问题呢?
>
> 斐波那契数列
>
> 汉诺塔问题
>
> 等等
>
> 1.求1~5之间的和

```
package com.atguigu.test2;

/**
 *  递归：将一个大的问题  拆分为若干个小的问题  逐步解决    必须有正确的出口
 *
 *  举例：  赵四    广坤     小宝      大拿
 *
 *  递归可以解决哪些问题呢?
 *  斐波那契数列
 *  汉诺塔问题
 *  等等
 *
 *
 *  1.求1~5之间的和
 *
 *
 */
public class Test {
    public static int getNum5(int num){
        return num + getNum4(num -1);
    }
    private static int getNum4(int num) {
        return num + getNum3(num -1);
    }
    private static int getNum3(int num) {
        return num + getNum2(num -1);
    }
    private static int getNum2(int num) {
        return num + getNum1(num -1);
    }
    private static int getNum1(int num) {
        return 1;
    }


    public static int getNum(int num){
        if(num == 1){
            return 1;
        }
        return num + getNum(num -1);
```

```
    }


    public static int getFactorial(int num){
        if(num == 1){
            return 1;
        }
        return num * getFactorial(num -1);
    }


    public static void main(String[] args) {
        int num5 = getNum5(5);

        System.out.println("num5 = " + num5);

        System.out.println(getNum(7));

        System.out.println(getFactorial(5));
    }







    public static void m1(){
        m1();
    }



}
```