多线程

什么是多线程

普通方法：

只有主程序一条执行路径

多线程：

多条执行路径，主线程和子线程并行交替执行

Process与Thread

说起进程，就不得不说下程序。

程序是指令和数据的有序集合，其本身没有任何运行的含义，是一个静态的概念。

而进程则是执行程序的一次执行过程，它是一个动态的概念。是系统资源分配的位

通常在一个进程中可以包含若干个线程，当然**一个进程中至少有一个线程**，不然没 有存在的意义。线程是CPU调度和执行的的单位。

注意：很多多线程是模拟出来的，真正的多线程是指有多个cpu，即多核，如服务器。

如果是模拟出来的多线程，即在一个cpu的情况下，在同一个时间点，cpu只能执行一个代码，因为切换的很快，所以就有同时执行的错局。

核心概念

- 线程就是独立的执行路径；
- 在程序运行时，即使没有自己创建线程，后台也会有多个线程，如主线程，gc线程；
- main() 称之为主线程，为系统的入口，用于执行整个程序；
- 在一个进程中，如果开辟了多个线程，线程的运行由调度器安排调度，调度器是与操作系统紧密相关的，先后顺序是不能认为的干预的。
- 对同一份资源操作时，会存在资源抢夺的问题，需要加入并发控制；
- 线程会带来额外的开销，如cpu调度时间，并发控制开销。
- 每个线程在自己的工作内存交互，内存控制不当会造成数据不一致

线程创建

Thread 继承

Runnable 接口

Callable 接口

继承Thread类

- 自定义线程类继承Thread类
- 重写run()方法，编写线程执行体
- 创建线程对象，调用start()方法启动线程

```
//创建自定义线程类继承thread
public class StartGames extends Thread{
```

```
    public void run(){
        for (int i = 0; i < 10; i++) {
            System.out.print("run--"+i+"\t");
        }

    }

}

//主方法中调用线程类并且使用start()方法启动线程
public static void main(String[] args) {

        StartGames games1 = new StartGames();
        StartGames games2 = new StartGames();
        StartGames games3 = new StartGames();

        games1.start();
        games2.start();
        games3.start();

    }
```

实现Runnable

- 定义MyRunnable类实现Runnable接口
- 实现run()方法，编写线程执行体
- 创建线程对象，调用start()方法启动线

```
//实现Runnable接口
public class StartGames implements Runnable{

    public void run(){
        for (int i = 0; i < 10; i++) {
            System.out.print("run--"+i+"\t");
        }

    }

}

public static void main(String[] args) {

        StartGames games = new StartGames();
        //创建线程对象，调用start()方法启动线程
        Thread thread = new Thread(games);

        thread.start();

    }
```

推荐使用Runnable对象，因为Java单继承的局限性

小结

继承Thread类

- 子类继承Thread类具备多线程能力
- 启动线程：子类对象. start()
- 不建议使用：避免OOP单继承局限性

实现Runnable接口

- 实现接口Runnable具有多线程能力
- 启动线程：传入目标对象+Thread对象.start()
- 推荐使用：避免单继承局限性，灵活方便，方便同一个对象被多个线程使用

实现Callable接口（了解即可）

1. 实现Callable接口，需要返回值类型
2. 重写call方法，需要抛出异常
3. 创建目标对象
4. 创建执行服务：ExecutorService ser = Executors.newFixedThreadPool(1);
5. 提交执行：Future result1 = ser.submit(t1);
6. 获取结果：boolean r1 = result1.get()
7. 关闭服务：ser.shutdownNow()

Lamda表达式

简化代码

前提必须为**函数式接口**（接口里只有一个方法）

lamda表达式只有一行代码的时候才能简化成一行

静态代理

- 你：真实角色
- 婚庆公司：代理你，帮你处理结婚的事
- 结婚：实现都实现结婚接口即可

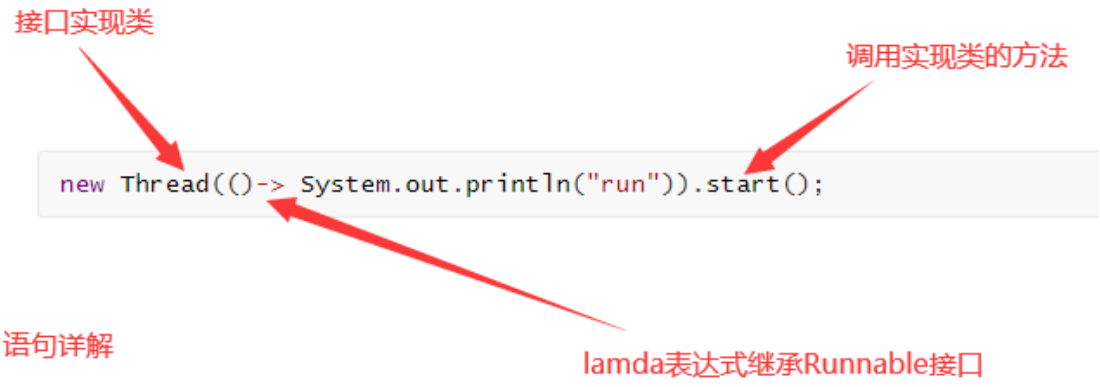
```
package com.lwq.snake;

public class Rabbit {
    public static void main(String[] args) {
        WeddingCompany company = new WeddingCompany(new you());
        company.Happy();
    }
}


interface Marry{
    void Happy();
}


//真实角色
class you implements Marry{

    @Override
    public void Happy() {
        System.out.println("真实角色真实结婚");
    }
```

```
}

//代理角色
class WeddingCompany implements Marry{

    private Marry target;

    public WeddingCompany(Marry target) {
        this.target = target;
    }

    @Override
    public void Happy() {
        before();
        System.out.println("代理角色帮助你结婚");
        after();
    }

    private void before() {
        System.out.println("结婚前");
    }

    private void after() {
        System.out.println("结婚后");
    }


}
```

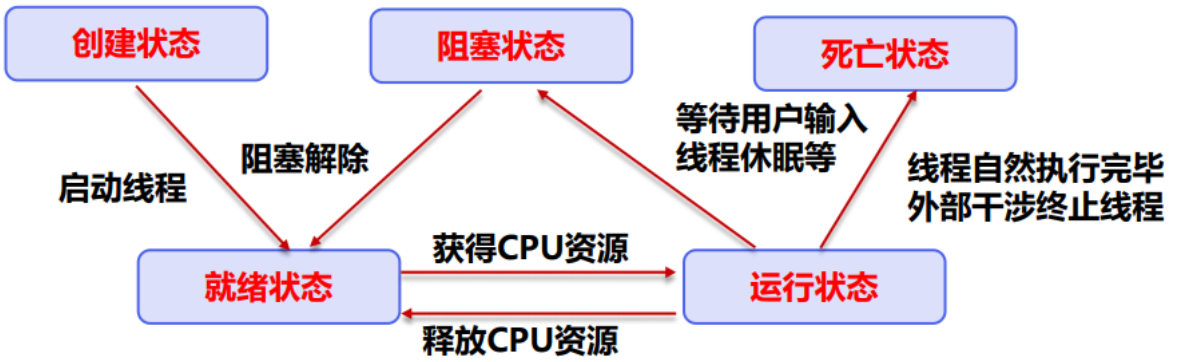
总结

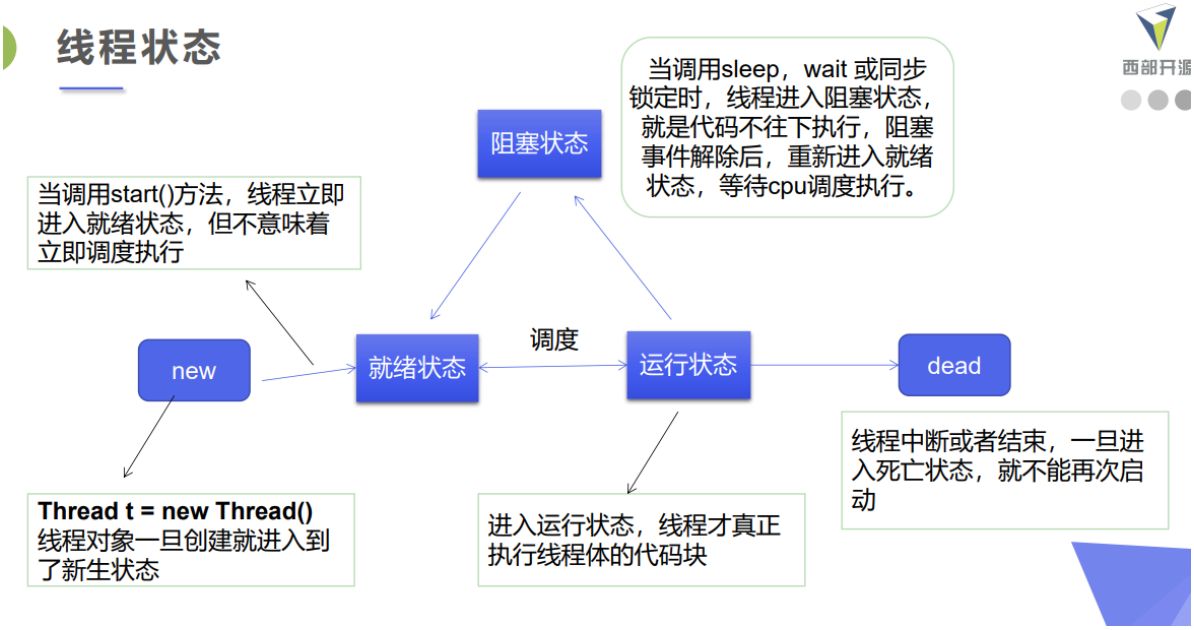
真实对象和代理对象都实现同一个接口，代理对象要代理真实角色，

```
new Thread(()-> System.out.println("run")).start();
```



线程状态





停止线程

不推荐使用JDK提供的 stop()、 destroy()方法。【已废弃】

推荐线程自己停止下来

建议使用一个标志位进行终止变量 当flag=false，则终止线程运行。

```
public class TestStop implements Runnable{
    private boolean flag=true;

    @Override
    public void run{
        while(flag){
            System.out,println("run...Thread");
        }
    }

    public void stop{
        this.flag=false;
    }
}
```

线程休眠

- sleep (时间) 指定当前线程阻塞的毫秒数；
- sleep存在异常InterruptedException；
- sleep时间达到后线程进入就绪状态；
- sleep可以模拟网络延时，倒计时等。
- 每一个对象都有一个锁，sleep不会释放锁；

线程礼让

礼让线程，让当前**正在执行**的线程**暂停**，但不阻塞

将线程从运行状态转为就绪状态

让cpu重新调度，礼让不一定成功！看CPU心情

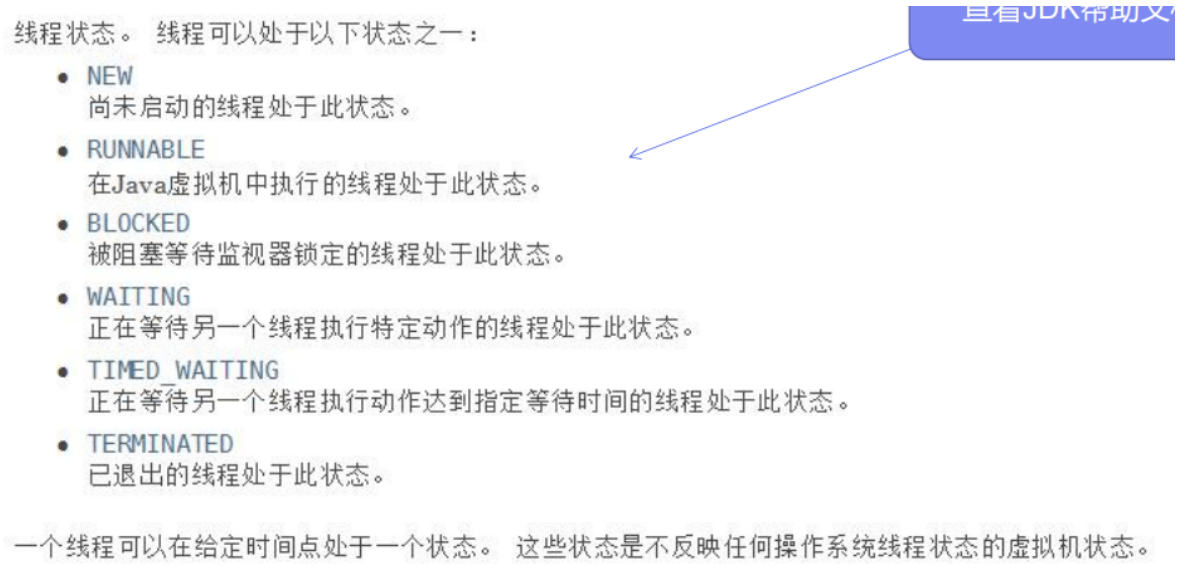
Join

Join合并线程，待此线程执行完成后，再执行其他线程，其他线程阻塞

可以想象成插队

线程状态观测

Thread.State



线程优先级

Java提供一个线程调度器来监控程序中启动后进入就绪状态的所有线程，线程调度 器按照优先级决定应该调度哪个线程来执行。

线程的优先级用数字表示，范围从1~10.

Thread.MIN_PRIORITY = 1;

Thread.MAX_PRIORITY = 10;

Thread.NORM_PRIORITY = 5;

使用以下方式改变或获取优先级

getPriority() . setPriority(int xxx)

守护(daemon)线程

线程分为**用户线程**和**守护线程**

虚拟机必须确保用户线程执行完毕

虚拟机不用等待守护线程执行完毕

如,后台记录操作日志,监控内存,垃圾回收等待.

```
package com.lwq.snake;

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

        God god = new God();

        You you = new You();

        Thread thread = new Thread(god);

        thread.setDaemon(true);

        thread.start();


        Thread thread1 = new Thread(you);

        thread1.start();

    }
}


class God implements Runnable{

    @Override
    public void run() {
        while (true){
            System.out.println("上帝保佑着你");
        }
    }
}


class You implements Runnable{

    @Override
    public void run() {
        for (int i = 0; i < 36500; i++) {
            System.out.println("开心的活着");
        }
        System.out.println("你没了");
    }
}
```

线程同步

多个线程操作同一个资源

并发

并发 : 同一个对象被**多个线程**同时操作

线程同步

现实生活中,我们会遇到 " 同一个资源 , 多个人都想使用 " 的问题 , 比如,食堂排队 打饭 , 每个人都想吃饭

处理多线程问题时 , 多个线程访问同一个对象 , 并且某些线程还想修改这个对象 . 这时候我们就需要线程同步 . 线程同步其实就是一种**等待**机制 , 多个需要同时访问 此对象的线程进入这个**对象的等待池形成队列**, 等待前面线程使用完毕 , 下一个线程再使用

队列 和 锁

由于同一进程的多个线程共享同一块存储空间 , 在带来方便的同时,也带来了访问冲突问题 , 为了保证数据在方法中被访问时的正确性 , 在访问时加入锁机制 synchronized , 当一个线程获得对象的排它锁 , 独占资源 , 其他线程必须等待 , 使用后释放锁即可.

存在以下问题 :

一个线程持有锁会导致其他所有需要此锁的线程挂起 ;

在多线程竞争下 , 加锁 , 释放锁会导致比较多的上下文切换和调度延时,引起性能问题 ;

如果一个优先级高的线程等待一个优先级低的线程释放锁会导致优先级倒置 , 引起性能问题

同步块 : synchronized (Obj) { }

```
package com.lwq.snake;

public class Rabbit{
    public static void main(String[] args) {
        BuyTicket buyTicket = new BuyTicket();

        new Thread(buyTicket,"我").start();
        new Thread(buyTicket,"你们").start();
        new Thread(buyTicket,"黄牛").start();
    }
}

class BuyTicket implements Runnable{

    private int ticketNum=10;
    boolean flag = true;

    @Override
    public void run() {
        while (flag) {
            try {
                buy();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
```

```
    }
//加上synchronized用来保证线程同步
    private synchronized void buy() throws InterruptedException {
        if (ticketNum<=0){
            flag=false;
            return;
        }
        Thread.sleep(200);
        System.out.println(Thread.currentThread().getName()+"买到第"+ticketNum--
+"张票");
    }
}
```

```
public class Competition {
    public static void main(String[] args) {
        Account account = new Account(5000, "结婚基金");
        Bank you = new Bank(account,50,"you");
        Bank gf = new Bank(account,5000,"gf");

        you.start();
        gf.start();
    }
}

class Account{

    int money;

    String name;

    public Account(int money, String name) {
        this.money = money;
        this.name = name;
    }
}

class Bank extends Thread{
    Account account;

    int  drawingMoney;

    int nowMoney;

    public Bank(Account account, int drawingMoney, String name){
        super(name);
        this.account=account;
        this.drawingMoney=drawingMoney;

    }

    @Override
    public void run() {
        //同步代码块
        synchronized (account) {
            if (account.money - drawingMoney < 0) {
                System.out.println("尊敬的" + Thread.currentThread().getName() +
", 您的余额不足");
```

```
                return;
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            account.money -= drawingMoney;
            nowMoney += drawingMoney;

            System.out.println("账户余额：" + account.money);
            System.out.println(this.getName() + "手里有" + nowMoney);
        }
    }
}
```

死锁

多个线程各自占有一些共享资源，并且互相等待其他线程占有的资源才能运行，而 导致两个或者多个线程都在等待对方释放资源，都停止执行的情形 . 某一个同步块 同时拥有 “ 两个以上对象的锁 ” 时，就可能会发生 “

死锁避免方法

产生死锁的四个必要条件：

1. 互斥条件：一个资源每次只能被一个进程使用。
2. 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
3. 不剥夺条件 : 进程已获得的资源，在未使用完之前，不能强行剥夺。
4. 循环等待条件 : 若干进程之间形成一种头尾相接的循环等待资源关系。

Lock(锁)

从JDK 5.0开始，Java提供了更强大的线程同步机制——通过显式定义同步锁对象来实现同步。同步锁使用Lock对象充当

java.util.concurrent.locks.Lock接口是控制多个线程对共享资源进行访问的工具。 锁提供了对共享资源的独占访问，每次只能有一个线程对Lock对象加锁，线程开始访问共享资源之前应先获得Lock对象

ReentrantLock 类实现了 Lock ， 它拥有与 synchronized 相同的并发性和内存语 义，在实现线程安全的控制中，比较常用的是ReentrantLock，可以显式加锁、释放锁

```
public class Rabbit{
    public static void main(String[] args) {
        BuyTicket buyTicket = new BuyTicket();

        new Thread(buyTicket).start();
        new Thread(buyTicket).start();
        new Thread(buyTicket).start();
    }
}

class BuyTicket implements Runnable{

    private int ticketNum=10;
```

```
    private final ReentrantLock Lock = new ReentrantLock();

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

            try {
                Lock.lock();
                if (ticketNum>0){
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println(ticketNum--);
                }else
                    break;
            }finally {
                Lock.unlock();
            }
        }
    }
}
```

synchronized 与 Lock 的对比

Lock是显式锁（手动开启和关闭锁，别忘记关闭锁）synchronized是隐式锁，出了 作用域自动释放

Lock只有代码块锁，synchronized有代码块锁和方法锁

使用Lock锁，JVM将花费较少的时间来调度线程，性能更好。并且具有更好的扩展 性（提供更多的子类）

优先使用顺序:

Lock > 同步代码块（已经进入了方法体，分配了相应资源）> 同步方法（在方 法体之外）

线程协作

线程通信

应用场景 : 生产者和消费者问题

- 假设仓库中只能存放一件产品 , 生产者将生产出来的产品放入仓库 , 消费者将仓库中产品取走消费 .
- 如果仓库中没有产品 , 则生产者将产品放入仓库 , 否则停止生产并等待 , 直到仓库中的产品被消费者取走为止 .
- 如果仓库中放有产品 , 则消费者可以将产品取走消费 , 否则停止消费并等待 , 直到仓库中再次放入产品为止 .

这是一个线程同步问题 , 生产者和消费者共享同一个资源 , 并且生产者和消费者之 间相互依赖

对于生产者 , 没有生产产品之前 , 要通知消费者等待 . 而生产了产品之后 , 又需要马上通知消费者消费

对于消费者 , 在消费之后 , 要通知生产者已经结束消费 , 需要生产新的产品 以供消费.

在生产者消费者问题中 , 仅有synchronized是不够的

synchronized 可阻止并发更新同一个共享资源 , 实现了同步

synchronized 不能用来实现不同线程之间的消息传递 (通信)

wait(),wait(long timeout),notify(),notifyAll()

均是Object类的方法，都只能在同步方法或者同步代码块中 使用,否则会抛出异常 IllegalMonitorStateException

解决方式1

管程法

```
//管程法
public class Rabbit{
    public static void main(String[] args) {
        SynContainer synContainer = new SynContainer();

        new Producer(synContainer).start();
        new Customer(synContainer).start();

    }
}
//生产者生产产品
class Producer extends Thread{
    SynContainer container;

    public Producer(SynContainer container){
        this.container=container;
    }

    @Override
    public void run() {
        for (int i = 1; i <= 100; i++) {
            System.out.println("生产了第"+i+"只鸡");
            container.push(new Chicken(i));
        }

    }
}
//消费者消费产品
class Customer extends Thread{
    SynContainer container;

    public Customer(SynContainer container){
        this.container=container;
    }

    @Override
    public void run() {
        for (int i = 1; i <= 100; i++) {
            System.out.println("消费了第"+container.pop().id+"只鸡");
        }
    }
}
//产品
class Chicken{
    int id;

    public Chicken(int id) {
```

```
        this.id = id;
    }
}
//缓冲区
class SynContainer{
    //需要一个容器大小
    Chicken[] chickens=new Chicken[10];

    //计算容器的数量
    int count=0;

    //放入产品
    public synchronized void push(Chicken chicken){
        //如果容器满了就等待消费者
        if (count== chickens.length){
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        //如果没有满我们就要丢入产品
        chickens[count]=chicken;
        count++;
        //可以消费了
        this.notifyAll();
    }
    //消费产品
    public synchronized Chicken pop(){
        //判断能否消费
        if (count==0){
            //等待生产者生产，消费者消费
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        //如果可以消费
        count--;
        Chicken chicken=chickens[count];

        this.notifyAll();
        return chicken;
    }
}
```

信号灯法

```
public class Rabbit{
    public static void main(String[] args) {
        TV tv = new TV();

        new actor(tv).start();
        new visitor(tv).start();
    }
}
```

```java
//生产--演员
class actor extends Thread{
    TV tv;

    public actor(TV tv) {
        this.tv = tv;
    }

    @Override
    public void run() {
        for (int i = 0; i < 20; i++) {
            if (i%2==0){
                this.tv.play("快乐大本营播放中");
            }else
                this.tv.play("看广告");
        }
    }
}

//消费--观众
class visitor extends Thread{
    TV tv;

    public visitor(TV tv) {
        this.tv = tv;
    }
    @Override
    public void run() {
        for (int i = 0; i < 20; i++) {
            tv.watch();
        }
    }
}

//产品--节目
class TV{
    //演员表演，观众等待
    //观众观看，演员等待
    String voice;//表演饿节目
    boolean flag = true;

    //表演
    public synchronized void play(String voice){
        System.out.println("演员表演了"+voice);

        //通知观众观看
        this.notifyAll();//通知唤醒
        this.voice=voice;
        this.flag = !this.flag;
    }

    //观看
    public synchronized void watch(){
        if (flag){
            try {
                this.wait();
            } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }
        }
        System.out.println("观看了"+voice);
        //通知演员表演
        this.notifyAll();
        this.flag=!this.flag;
    }
}
```

线程池

背景：经常创建和销毁、使用量特别大的资源，比如并发情况下的线程，对性能影 响很大。

思路：提前创建好多个线程，放入线程池中，使用时直接获取，使用完放回池中。 可以避免频繁创建销毁、实现重复利用。类似生活中的公共交通工具。

好处：

- 提高响应速度（减少了创建新线程的时间）
- 降低资源消耗（重复利用线程池中线程，不需要每次都创建）
- 便于线程管理(....) u corePoolSize：核心池的大小

maximumPoolSize：最大线程数

keepAliveTime：线程没有任务时最多保持多长时间后会终止

使用线程池

JDK 5.0起提供了线程池相关API：ExecutorService 和 Executors

ExecutorService：真正的线程池接口。常见子类ThreadPoolExecutor

- void execute(Runnable command)：执行任务/命令，没有返回值，一般用来执 行Runnable
- Future submit(Callable task)：执行任务，有返回值，一般又来执行 Callable
- void shutdown()：关闭连接池

Executors：工具类、线程池的工厂类，用于创建并返回不同类型的线程池

```
public class Competition {
    public static void main(String[] args) {
        //创建服务，创建线程池
        //newFixedThreadPool 参数为池子的大小
        ExecutorService service = Executors.newFixedThreadPool(10);
        service.execute(new MyThread());
        service.execute(new MyThread());
        service.execute(new MyThread());
        service.execute(new MyThread());

        service.shutdown();
    }
}

class MyThread implements Runnable{
    @Override
    public void run() {
            System.out.println(Thread.currentThread().getName());
    }
}
```