进程和线程

进程： 相当于电脑中可以运行的应用软件

线程： 相当于应用软件中可以同时运行的功能

多线程

> 作用：让程序同时做多件事情，提高效率

CPU可以在多个线程之间进行切换运行，将空闲时间进行利用

应用场景：

1. 拷贝、迁移大文件
2. 加载大量资源
3. .....

并发和并行

并发：在同一时刻，有多个指令在单个CPU上交替执行，简单来说就是一个cpu一会执行线程a，一会执行线程b，间隔时间非常短，看起来就像在同时执行线程a和线程b

并行：在同一时刻，有多个执行在指令在多个cpu上同时执行，简单来说就是有两个cpu，一个执行线程a，一个执行线程b

关于CPU参数 有 x核y线程

意思就是如果你电脑上同时运行的线程数小于等于y，那么此时cpu就是在并行工作

但是如果线程数大于y，那么就是cpu在并发工作，同时也是在并行工作

多线程的实现方式

继承Thread类

> 首先编写一个类继承Thread类

```
public class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(getName() + "hello");
        }
    }
}
```

> 接下来在主函数中创建这个对象然后执行start方法即可开启线程

```
public class demo01 {
    public static void main(String[] args) {
        MyThread myThread1  = new MyThread();
        MyThread myThread2 = new MyThread();

        myThread1.setName("01");
        myThread2.setName("02");

        myThread1.start();
        myThread2.start();
    }
}
```

实现Runnable接口

> 首先编写一个类实现Runnable接口

```
public class MyRunnable implements Runnable{

    public void run() {
        // 在Runnable接口中是无法使用getName方法获取线程名，所以这里选择获取当前线程然后获取线程名
        Thread thread = Thread.currentThread();
        for (int i = 0; i < 100; i++) {
            System.out.println(thread.getName() + "hello");
        }
```

> 接下来在主函数中创建Thread对象然后将自定义的Runnable接口传入即可

```
public class demo02 {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable());
        Thread t2 = new Thread(new MyRunnable());
        t1.setName("t1");
        t2.setName("t2");

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

利用Callable和Future接口方式实现

> 第三种方式实现起来较为复杂，但是特点是相比于前两种方式只是单纯的执行线程中的内容，第三种方式可以获取到线程的返回值，相比于前两种更加的灵活

> 首先是新建一个类实现Callable接口

```
public class MyCallable implements Callable<Integer> {
    public Integer call() throws Exception {
        int sum = 0;
        for (int i = 0; i <= 100; i++) {
            sum += i;
        }
        return sum;
    }
}
```

> 这里的泛型决定了返回什么值
>
> 接下来编写主函数代码

```
public class demo03 {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        /**
         * 特点：可以获取到多线程运行的结果
         */

        MyCallable mc = new MyCallable();
        // 创建FutureTask对象用来接收线程的返回值
        FutureTask<Integer> futureTask = new FutureTask<Integer>(mc);
        //将FutureTask传给Thread去运行
        Thread t1 = new Thread(futureTask);
        t1.start();
        //待线程运行完毕后即可从futureTask中获取到线程的返回值
        Integer resutlt = futureTask.get();

        System.out.println(resutlt);
    }
}
```

常见的成员方法

方法名称	说明
String getName()	返回此线程的名称
void setName(String name)	设置线程的名字（构造方法也可以设置名字）
static Thread currentThread()	获取当前线程的对象
static void sleep(long time)	让线程休眠指定的时间，单位为毫秒
setPriority(int newPriority)	设置线程的优先级
final int getPriority()	获取线程的优先级
final void setDaemon(boolean on)	设置为守护线程
public static void yield()	出让线程/礼让线程
public static void join()	插入线程/插队线程

一些细节：

- 当JVM虚拟机启动的时候，会自动的启动多条线程，其中一条线程名字就叫做main，用来执行main方法中的所有代码
- 关于sleep(x)方法，哪条线程执行到此方法，就会在此停留x毫秒

线程的优先级

- 抢占式调度：多个线程抢夺cpu的资源，**随机性**
- 非抢占式调度：多个线程排队使用cpu的资源

线程优先级越高，抢到cpu资源的概率也就越大

```
public class demo01 {
    public static void main(String[] args) {
        MyRunnable mr = new MyRunnable();

        Thread t1 = new Thread(mr, "t1");
        Thread t2 = new Thread(mr, "t2");

        //获取默认优先级
        System.out.println(t1.getPriority());
        System.out.println(t2.getPriority());

        System.out.println(Thread.currentThread().getPriority());

        //优先级高不代表t2线程就一定先执行完毕，只是说占有cpu的概率高
        t1.setPriority(2);
        t2.setPriority(10);

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

守护线程

> 当设置线程a为守护线程，线程b为非守护线程的时候
>
> 线程b执行完毕后，无论线程a有没有执行完毕，都会陆续结束

```
public class demo02 {
    public static void main(String[] args) {
        MyThread1 myThread1 = new MyThread1();
        MyThread2 myThread2 = new MyThread2();

        myThread2.setDaemon(true);

        myThread1.setName("非守护线程");
        myThread2.setName("守护线程");


        myThread1.start();
        myThread2.start();
    }
}
```

> 应用案例：
>
> 简单的聊天系统分为两个线程，一个线程是主要聊天功能，另一个线程用来发送文件或者接收文件
>
> 当聊天窗口关闭的时候，收发文件也没有存在的必要了，所以一般会设置收发文件线程为守护线程

礼让线程

> 作用：让出当前cpu的资源，然后再次与其它线程进行竞争cpu资源
>
> 有一定的平衡各个线程占用资源的问题，但是也不一定会平衡，因为抢占cpu是一个概率的事情

```
public class MyThread1 extends Thread{
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(getName() + "hello");
            //让出当前cpu的执行权
            Thread.yield();
        }
    }
}

public class demo03 {
    public static void main(String[] args) {
        MyThread1 mt1 = new MyThread1();
        MyThread1 mt2 = new MyThread1();
```

```
        mt1.setName("t1");
        mt2.setName("t2");

        mt1.start();
        mt2.start();
    }
}
```

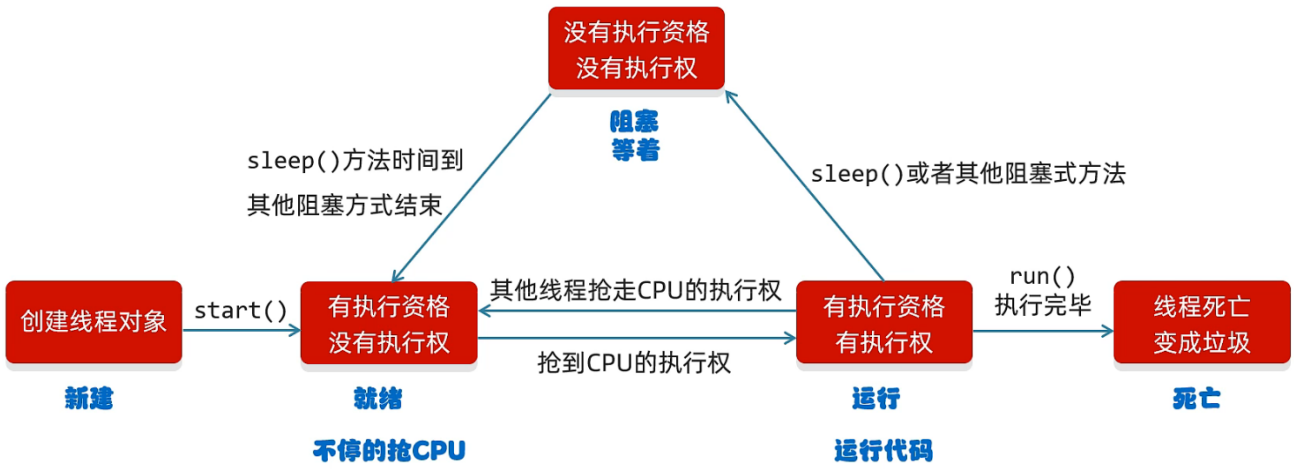
插入线程

> 作用：阻塞线程，等待插入线程执行完毕后再去执行下面的代码

```
public class demo04 {
    public static void main(String[] args) throws InterruptedException {
        MyThread2 mt2 = new MyThread2();

        mt2.start();
        //等mt2线程执行完毕后，再往下执行
        mt2.join();

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

线程的生命周期

线程的生命周期



线程的安全问题

例子：

一共100张票，开放三个通道同时售票，编写Java程序模拟实现

```
public class MyThread extends Thread {
    //使用static关键字，使得该类的所有对象共享一份ticket数据
    private static int ticket = 0;

    @Override
    public void run() {
        while (ticket < 100){
            ticket ++;
            System.out.println(getName() + "抢到了第" + ticket + "张票");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class demo01 {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();

        t1.setName("通道1");
        t2.setName("通道2");
        t3.setName("通道3");

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

> 上述代码会出现问题
>
> 因为三个线程同时进行，那么不可避免地就是三个线程都在操作ticket，有可能在线程1做自增还没运行到输出代码，线程2就已经完成了自增，这个时候线程1去执行输出代码，岂不是就是自增了两次
>
> 这里就很像MySQL中事务并发所带来的不可重复读问题
>
> 解决方案也就是给代码块上锁，在同一时刻只有一个线程能执行

```
public class MyThread extends Thread {
    //使用static关键字，使得该类的所有对象共享一份ticket数据
    private static int ticket = 0;
```

```
    //保证锁对象是唯一的
    static Object obj = new Object();

    @Override
    public void run() {
        while (ticket < 100){
            //同步代码块
            synchronized (obj) {
                ticket++;
                if(ticket > 100){
                    return;
                }
                System.out.println(getName() + "抢到了第" + ticket + "张票");
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

- synchronized不能写在循环外面，因为此代码块内部的代码会让一个线程执行完毕
- 关于线程锁，如果不唯一，那么就是一个线程对应一把锁，线程a上了自己的锁但是并不影响线程b，因为线程b的锁并没有上，所以两个线程还是同时执行同一段代码

> 关于同步方法，如果synchronized作用域一个方法，那么锁对象不能随意指定，应为当前类的字节码对象

```
public class MyThread implements Runnable {
    //使用static关键字，使得该类的所有对象共享一份ticket数据
    private static int ticket = 0;

    //保证锁对象是唯一的
    static Object obj = new Object();

    public void run() {
        while (true){
            if(method()) break;
        }
    }

    private synchronized boolean method() {
        ticket++;
        if(ticket > 100){
            return true;
        }
        System.out.println(Thread.currentThread().getName() + "抢到了第" + ticket
+ "张票");
        try {
```

```
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return false;
    }
}
```

> 这里使用的是实现Runnable接口方式，如果使用继承Thread方式，那么记得方法上需要加上static关键词声明成静态方法，否则三个对象就是三把锁

```
public class MyThread extends Thread {
    //使用static关键字，使得该类的所有对象共享一份ticket数据
    private static int ticket = 0;

    //保证锁对象是唯一的
    static Object obj = new Object();

    @Override
    public void run() {
        while (true){
            if(method()) break;
        }
    }

    private static synchronized boolean method() {
        ticket++;
        if(ticket > 100){
            return true;
        }
        System.out.println(Thread.currentThread().getName() + "抢到了第" + ticket
+ "张票");
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return false;
    }
}
```

关于StringBuilder和StringBuffer

> 通过翻阅源码可以发现
>
> StringBuilder是线程不安全的
>
> StringBuffer是线程安全的，因为其所有方法上都上了synchronized关键字，上了锁

使用场景：

当涉及到多个线程操作同一个字符串的时候，使用StringBuffer，如果只有一个main线程，则使用StringBuilder

Lock锁

> 相比于synchronized， lock可以控制从什么地方开始进入锁，并且从哪释放锁，更加的灵活
>
> Lock是一个接口，一般使用ReentrantLock这个实现类，但是需要注意的是如果使用继承Thread类方法来编写多线程，需要给锁加上static关键字

```
static ReentrantLock lock = new ReentrantLock();

    @Override
    public void run() {
        while (true){
            //上锁
            lock.lock();
            try {
                ticket++;
                if(ticket > 100){
                    break;
                }
                System.out.println(Thread.currentThread().getName() + "抢到了第" +
ticket + "张票");
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            } catch (Exception e) {
                e.printStackTrace();
            } finally {    //finally代码块中的代码是一定会执行一遍的，所以在此释放
                //释放锁
                lock.unlock();
            }
        }
    }
```

死锁

> 死锁不是一个技术，是一个错误，学习死锁是为了避免这个错误

```java
public class MyThread extends Thread{

    static Object objA = new Object();
    static Object objB = new Object();

    @Override
    public void run() {
        //1.循环
        while(true){
            if("线程A".equals(getName())){
                synchronized (objA){
                    System.out.println("线程A拿到了A锁，准备拿B锁");
                    synchronized (objB){
                        System.out.println("线程A拿到了B锁，顺利执行完一轮");
                    }
                }
            }else if("线程B".equals(getName())){
                if("线程B".equals(getName())){
                    synchronized (objB){
                        System.out.println("线程B拿到了B锁，准备拿A锁");
                        synchronized (objA){
                            System.out.println("线程B拿到了A锁，顺利执行完一轮");
                        }
                    }
                }
            }
        }
    }
}
```

```java

public class ThreadDemo {
    public static void main(String[] args) {
        /*
            需求：
                死锁
        */


        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        t1.setName("线程A");
        t2.setName("线程B");

        t1.start();
        t2.start();

    }
}
```

生产者和消费者（等待唤醒机制）

生产者和消费者（等待唤醒机制）

1. 判断桌子上是否有食物
2. 如果没有就等待
3. 如果有就开吃
4. 吃完之后，唤醒厨师继续做

1. 判断桌子上是否有食物
2. 有：等待
3. 没有：制作食物
4. 把食物放在桌子上
5. 叫醒等待的消费者开吃

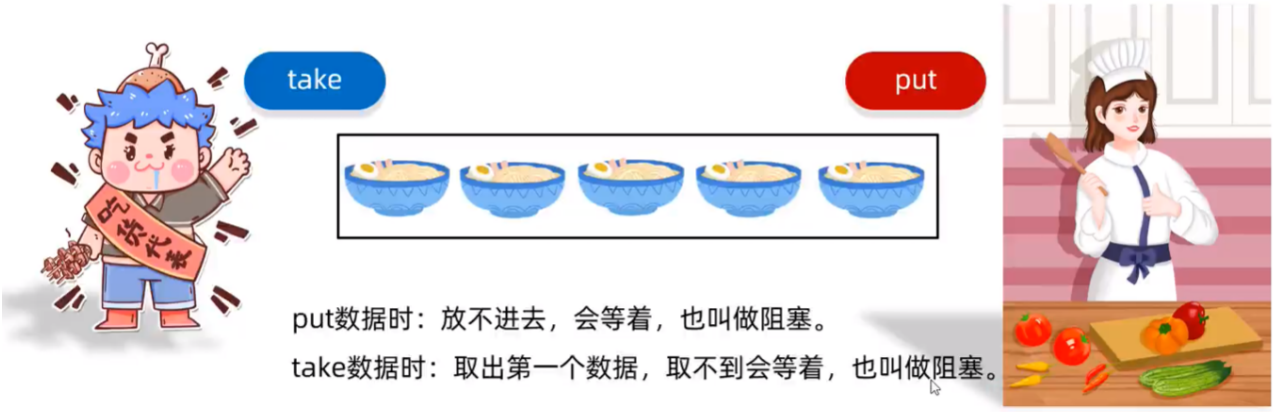
方法名称	作用
void wait()	当前线程等待，直到被其它线程唤醒
void notify()	随机唤醒单个线程
void notifyAll()	唤醒所有线程

```
/**
```

```
 * 控制生产者和消费者的执行
 */
public class Desk {
    /**
     * 是否有食物 0 - 无 1 - 有
     */
    public static int isHasFood = 0;

    /**
     * 总数
     */
    public static int count = 10;

    /**
     * 锁对象
     */
    public static final Object lock = new Object();
}


/**
 * 消费者--顾客
 */
public class Customer extends Thread{

    @Override
    public void run() {
        //1. 循环
        while (true){
            //2. 同步代码块
            synchronized (Desk.lock){
                //3. 根据需求将同步代码块改成同步方法或者使用Lock锁
                //4. 判断共享数据状态 （执行核心逻辑）
                if(Desk.count <= 0){
                    break;
                }
                if(Desk.isHasFood == 0){
                    try {
                        Desk.lock.wait(); //将锁和线程进行绑定
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                Desk.count--;
                System.out.println("Customer正在消费，还剩" + Desk.count + "可以消
费");
                Desk.isHasFood = 0;
                Desk.lock.notifyAll();  //唤醒这把锁绑定的所有线程
            }
        }
    }
}

/**
```

```
 * 生产者--厨师
 */
public class Cooker extends Thread{
    @Override
    public void run() {
        //1. 循环
        while (true){
            //2. 同步代码块
            synchronized (Desk.lock){
                //3. 根据需求将同步代码块改成同步方法或者使用Lock锁
                //4. 判断共享数据状态 （执行核心逻辑）
                if(Desk.count <= 0){
                    break;
                }
                if(Desk.isHasFood == 1){
                    try {
                        Desk.lock.wait(); //将锁和线程进行绑定
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.println("Cooker正在生产");
                Desk.isHasFood = 1;
                Desk.lock.notifyAll();  //唤醒这把锁绑定的所有线程
            }
        }
    }
}

public class ThreadDemo {
    public static void main(String[] args) {
        Cooker cooker = new Cooker();
        Customer customer = new Customer();

        cooker.setName("cooker");
        customer.setName("customer");


        cooker.start();
        customer.start();
    }
}
```

阻塞队列

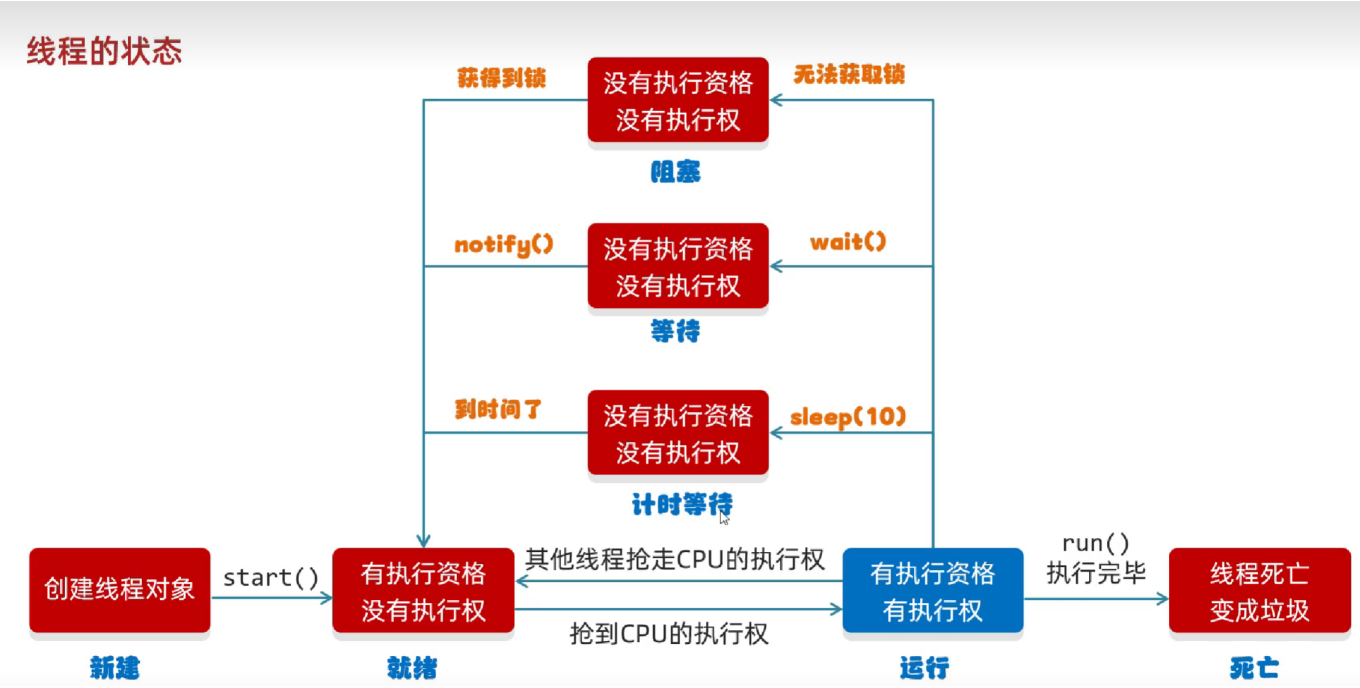



```
import java.util.concurrent.ArrayBlockingQueue;

public class Cooker extends Thread{
    ArrayBlockingQueue<String> queue;

    int count = 10;

    public Cooker(ArrayBlockingQueue<String> queue){
        this.queue = queue;
    }

    @Override
    public void run() {
        while (count-- > 0){
            try {
                queue.put("面条");
                System.out.println("生产者生产了一碗面条");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }
}

import java.util.concurrent.ArrayBlockingQueue;

public class Customer extends Thread{

    ArrayBlockingQueue<String> queue;

    int count = 10;

    public Customer(ArrayBlockingQueue<String> queue){
        this.queue = queue;
    }

    @Override
```

```
    public void run() {
        while (count-- > 0){
            try {
                String food = queue.take();
                System.out.println("消费者消费了一碗" + food);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }
}

public class Demo {
    public static void main(String[] args) {
        ArrayBlockingQueue<String> queue = new ArrayBlockingQueue<String>(1);

        Customer customer = new Customer(queue);
        Cooker cooker = new Cooker(queue);

        customer.start();
        cooker.start();
    }
}
```

> 这次的代码运行会出现连续打印相同内容的情况，出现原因是将输出代码写在了锁外面，两个线程都能执行

线程的状态



> 值得一提的是运行状态并没有被记录在Java虚拟机中，严格意义上来说线程的状态并不包含运行

> 因为线程的运行不归Java虚拟机去管理，而是由操作系统去管理

线程栈

> 每一个线程都有自己的栈，如果不加上static关键字，那么两个线程之间的数据是不共享的

线程池

> 作用：避免资源浪费，一个线程可以被多次使用

```
public class MyRunnable implements Runnable{
    public void run() {
        System.out.println(Thread.currentThread().getName() + "----");
    }
}

public class demo {
    public static void main(String[] args) throws InterruptedException {
        ExecutorService service = Executors.newCachedThreadPool();

        service.submit(new MyRunnable());
        Thread.sleep(100);
        service.submit(new MyRunnable());
        Thread.sleep(100);
        service.submit(new MyRunnable());

        service.shutdown();
    }
}
```

> 此段代码是创建一个无限制的线程池，说是无限制，其实也是有限制，最大数为int的最大值
>
> 这里在每个submit后休眠0.1s是为了保证上一个线程已经运行结束
>
> 所以这里打印出来的线程名都会是同一个，这就代表这个线程在运行结束后不会销毁而是会被线程池接管

自定义线程池

> 根据需求的不同，普通的线程池一般不能满足使用，这个时候需要使用自定义线程池
>
> 自定义线程池一般分为三块，核心线程、临时线程、队伍长度
>
> 核心线程就是主要进行工作的线程
>
> 临时线程是在核心线程满载，队伍长度也排满之后再去工作的线程
>
> 队伍长度就是等候线程的数量

> 假设现在核心线程为3，临时线程为3，队伍长度也为3
>
> 那么最多就是能容纳九个任务
>
> 需要注意的是，从第四个任务开始后面三个应进队列等待，到第七个的时候触发临时线程
>
> 但是如果出现第十个任务，那么就会触发任务拒绝策略

```
import com.object.ThreadPool.MyRunnable;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class demo {
    public static void main(String[] args) {
        ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(
                3,  //核心线程数量
                6, //最大线程数量
                60, //空闲线程的最大存活时间，就是在队列中最多等待多少时间
                TimeUnit.SECONDS, //设置时间单位
                new ArrayBlockingQueue<Runnable>(3), //设置阻塞队列
                Executors.defaultThreadFactory(), //设置线程工厂
                new ThreadPoolExecutor.AbortPolicy() //设置任务拒绝策略
        );

        poolExecutor.submit(new MyRunnable());
        poolExecutor.submit(new MyRunnable());
        poolExecutor.submit(new MyRunnable());

        poolExecutor.shutdown();
    }
}
```