多线程详解

线程简介

• 程序、进程、线程

在操作系统中运行的程序就是进程。

一个进程可以有多个线程，如视频中同时听声音，看图像，看弹幕，等等。

• 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。是系统资源分配的单位。
- 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觉。

• 一些核心概念

- 线程就是独立执行的路径
- 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，先后顺序是不能人为干预的
- 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

三种创建方式

- 继承Thread类（重点）
- 实现Runnable接口（重点）
- 实现Callable接口（了解）

Thread

- 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

```
public class StartThread extends Thread {
    // 线程入口点
    @Override
    public void run() {
        // 线程体
    }
}
```

```
public static void main(String[] args) {
    // 创建线程对象
    StartThread t = new StartThread();
    t.start();
}
```

> ⚠️ 注意：线程不一定立即执行，由CPU安排调度。

案例：图片下载

```
import org.apache.commons.io.FileUtils;

import java.io.File;
import java.io.IOException;
import java.net.URL;

public class TestThread extends Thread {

    private String url; // 网络图片地址
```

```java
    private String name; // 保存的文件名

    public static void main(String[] args) {
        TestThread t1 = new TestThread("https://blog.kuangstudy.con/usr/themes/handsome/usr/img/sj/2.jpg", "1.jpg");
        TestThread t2 = new TestThread("https://blog.kuangstudy.con/usr/themes/handsome/usr/img/sj/2.jpg", "2.jpg");
        TestThread t3 = new TestThread("https://blog.kuangstudy.con/usr/themes/handsome/usr/img/sj/2.jpg", "3.jpg");

        new Thread(t1).start();
        new Thread(t2).start();
        new Thread(t3).start();
    }

    public TestThread(String url, String name) {
        this.url = url;
        this.name = name;
    }

    // 下载图片线程的执行体
    @Override
    public void run() {
        WenDownloader wenDownloader = new WenDownloader();
        wenDownloader.downloader(url, name);
        System.out.println("下载了文件名为: " + name);
    }
}

// 下载器
class WenDownloader {
    // 下载方法
    public void downloader() {
        try {
            FileUtils.copyURLToFile(new URL(url), new File(name));
        } catch (IOException e) {
            e.printStackTrace();
            System.out.println("IO异常, downloader方法出现问题");
        }
    }
}
```

- **实现Runnable接口**

```
public class TestThread implements Runnable {

    // 票数
    private int ticketNums = 10;

    @Override
    public void run() {
        while (true) {
            if (ticketNums <= 0) {
                break;
            }
            // 模拟延迟
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            System.out.println(Thread.currentThread().getName() + "拿到了第" + ticketNums-- + "票");
        }
    }

    public static void main(String[] args) {
        TestThread ticket = new TestThread();

        new Thread(ticket, "小明").start();
        new Thread(ticket, "张三").start();
        new Thread(ticket, "李四").start();
    }
}
```

- **案例：龟兔赛跑**

```
// 模拟龟兔赛跑
public class Race implements Runnable {

    // 胜利者
    private static String winner;

    @Override
    public void run() {
        for (int i = 0; i <= 100; i++) {

            // 模拟兔子休息
            if (Thread.currentThread().getName().equals("兔子") && i % 10 == 0) {
                try {
```

```
                    Thread.sleep(200);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }

            // 判断比赛是否结束
            boolean flag = gameOver(i);

            if (flag) {
                // 如果比赛结束了，就停止程序
                break;
            }

            System.out.println(Thread.currentThread().getName() + "⟶跑了" + i + '步');
        }
    }

    // 判断是否完成比赛
    private boolean gameOver(int steps) {
        // 判断是否有胜利者
        if (winner != null) {
            // 已经存在胜利者
            return true;
        }
        {           if (steps >= 100) {
                winner = Thread.currentThread().getName();
                System.out.println("winner is " + winner);
                return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        Race race = new Race();

        new Thread(race, "兔子").start();
        new Thread(race, "乌龟").start();
    }
}
```

- **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<Boolean> result1 = ser.submit(t1);`
6. 获取结果：`boolean r1 = result1.get();`
7. 关闭服务：`ser.shutdownNow();`

演示：利用callable改造下载图片案例

```
import org.apache.commons.io.FileUtils;

import java.io.IOException;
import java.util.concurrent.*;

public class TestCallable implements Callable<Boolean> {

    private String url; // 网络图片地址
    private String name; // 保存的文件名

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        TestCallable t1 = new TestCallable("https://blog.kuangstudy.con/usr/themes/handsome/usr/img/sj/2.jpg", "1.jpg");
        TestCallable t2 = new TestCallable("https://blog.kuangstudy.con/usr/themes/handsome/usr/img/sj/2.jpg", "2.jpg");
        TestCallable t3 = new TestCallable("https://blog.kuangstudy.con/usr/themes/handsome/usr/img/sj/2.jpg", "3.jpg");

        // 创建执行服务
        ExecutorService ser = Executors.newFixedThreadPool(3);

        // 提交执行
        Future<Boolean> result1 = ser.submit(t1);
        Future<Boolean> result2 = ser.submit(t2);
        Future<Boolean> result3 = ser.submit(t3);

        // 获取结果
        boolean r1 = result1.get();
        boolean r2 = result2.get();
        boolean r3 = result3.get();

        System.out.println(r1);
        System.out.println(r2);
        System.out.println(r3);

        // 关闭服务
        ser.shutdownNow();
    }

    public TestCallable(String url, String name) {
```

```
        this.url = url;
        this.name = name;
    }

    // 下载图片线程的执行体
    @Override
    public Boolean call() throws Exception {
        WenDownloader wenDownloader = new WenDownloader();
        wenDownloader.downloader(url, name);
        System.out.println("下载了文件名为: " + name);
        return true;
    }
}

// 下载器
class WenDownloader {
    // 下载方法
    public void downloader() {
        try {
            FileUtils.copyURLToFile(new URL(url), new File(name));
        } catch (IOException e) {
            e.printStackTrace();
            System.out.println("IO异常, downloader方法出现问题");
        }
    }
}
```

> callable 的好处:
>
> 1. 可以定义返回值
> 2. 可以抛出异常

- **静态代理**

```
public class TestStaticProxy {
    public static void main(String[] args) {
        You you = new You();

        new Thread(() -> System.out.println("我爱你")).start();

        new WeddingCompany(you).HappyMarry();
    }
}

interface Marry{
    void HappyMarry();
}
```

```
14
15  // 真实角色
16  class You implements Marry{
17
18      @Override
19      public void HappyMarry() {
20          System.out.println("xxx要结婚了! ");
21      }
22  }
23
24  // 代理角色
25  class WeddingCompany implements Marry{
26
27      private Marry target;
28
29      public WeddingCompany(Marry target) {
30          this.target = target;
31      }
32
33      @Override
34      public void HappyMarry() {
35          before();
36          this.target.HappyMarry();
37          after();
38      }
39
40      private void after() {
41          System.out.println("结婚之后, 收尾款");
42      }
43
44      private void before() {
45          System.out.println("结婚之前, 布置现场");
46      }
47  }
```

> “
>
> 静态代理模式总结:
>
> 1. 真实对象和代理对象都要实现同一个接口
> 2. 代理对象要代理真实角色
> 3. 好处
>   1. 代理对象可以做很多真实对象做不了的事情
>   2. 真实对象专注做自己的事情

Lambda表达式

- 避免匿名内部类定义过多

- 其实质属于函数式编程的概念

```
(params) -> expression [表达式]
(params) -> statement [语句]
(params) -> {statements}
```

```
a -> System.out.println("i like lamda⟶" + a);
```

```
new Thread(() -> System.out.println("多线程学习 ... ")).start();
```

- 为什么要使用Lambda表达式
  - 避免匿名内部类定义过多
  - 可以让代码看起来更简洁
  - 去掉了一堆没有意义的代码，只留下核心的逻辑
- 理解Functional Interface（函数式接口）是学习Java8 Lambda 表达式的关键所在
- 函数式接口的定义：
  - 任何接口，如果只包含唯一一个抽象方法，那么它就是一个函数式接口

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

  - 对于函数式接口，我们可以通过Lambda表达式来创建该接口的对象
  - 演示：代码推导Lambda表达式

```
/**
 * 推到Lambda表达式
 */
public class TestLambda {

    // 3、静态内部类
    static class Like2 implements ILike{

        @Override
        public void lambda() {
            System.out.println("i like lambda2");
        }
    }

    public static void main(String[] args) {
        ILike like = new Like();
```

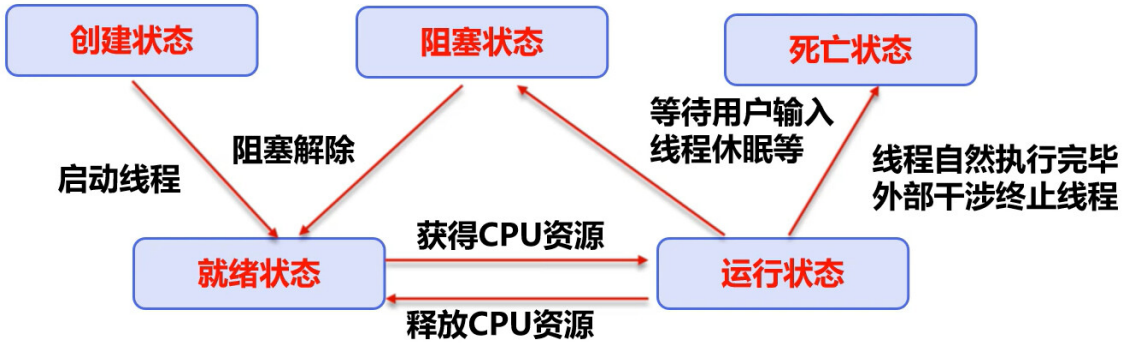
```
17          like.lambda();
18
19          like = new Like2();
20          like.lambda();
21
22          // 4、局部内部类
23          class Like3 implements ILike{
24
25              @Override
26              public void lambda() {
27                  System.out.println("i like lambda3");
28              }
29          }
30
31          like = new Like3();
32          like.lambda();
33
34          // 5、匿名内部类：没有类的名称，必须借助接口或者父类
35          like = new ILike() {
36              @Override
37              public void lambda() {
38                  System.out.println("i like lambda4");
39              }
40          };
41          like.lambda();
42
43          // 6、用Lambda简化
44          like = () -> System.out.println("i like lambda5");
45          like.lambda();
46      }
47  }
48
49  // 1、定义一个函数式接口
50  interface ILike{
51      void lambda();
52  }
53
54  // 2、实现类
55  class Like implements ILike{
56
57      @Override
58      public void lambda() {
59          System.out.println("i like lambda");
60      }
61  }
```

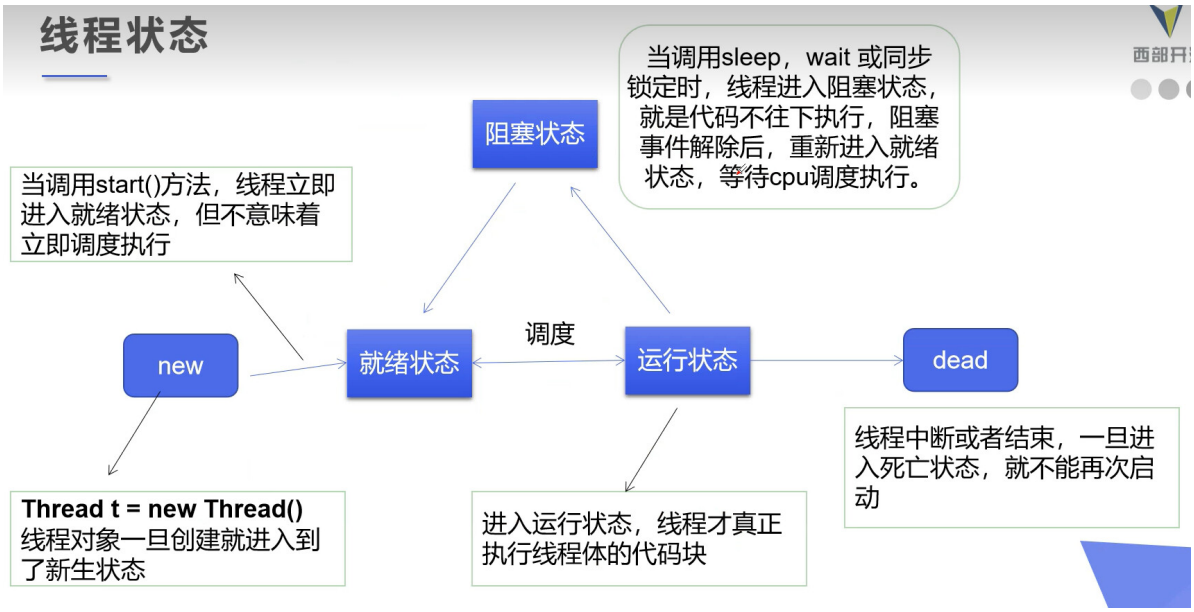
> 🔍总结：
>
> 1. Lambda表达式在只有一行代码的情况下才能简化成为一行，如果有多行，那么就用代码块包裹

2. 前提是接口时函数式接口
3. 多个参数也可以去掉参数类型，要去掉就都去掉，但必须加上括号

线程状态

• 五大状态





• 线程方法

方法	说明
setPriority(int newPriority)	更改线程优先级
static void sleep(long millis)	在指定的毫秒数内让当前正在执行的线程休眠
void join()	等待该线程终止
static void yield()	暂停当前正在执行的线程对象，并执行其他线程
void interrupt()	中断线程，**别用这个方式**
boolean isAlive()	测试线程是否处于活动状态

停止线程

- 不推荐使用JDK提供的 stop()、destory() 方法。【已废弃】
- 推荐线程自己停下来
- 建议使用一个标志位进行终止变量，当 flag=false，则线程终止运行

```java
public class TestStop implements Runnable {
    // 1. 线程中定义线程使用的标识
    private boolean flag = true;

    @Override
    public void run() {
        // 2. 线程体使用该标识
        while (flag) {
            System.out.println("run ... Thread");
        }
    }

    // 3. 对外提供方法改变标识
    public void stop() {
        this.flag = false;
    }

    public static void main(String[] args) {
        TestStop testStop = new TestStop();

        new Thread(testStop).start();

        for (int i = 0, i < 1000, i++) {
            System.out.println("main" + i);
            if (i == 900) {
                // 调用stop方法切换标志位，让线程停止
                testStop.stop();
                System.out.println("线程该停止了");
            }
        }
    }
}
```

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 时间到达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

- **演示：计时**

```java
import java.text.SimpleDateFormat;
import java.util.Date;

// 模拟倒计时
public class TestSleep {

    public static void main(String[] args) throws InterruptedException {
        tenDown();

        // 打印当前系统时间
        Date startTime = new Date(System.currentTimeMillis());

        while (true) {
            Thread.sleep(1000);
            System.out.println(new
SimpleDateFormat("HH:mm:ss").format(startTime));
            // 更新当前时间
            startTime = new Date(System.currentTimeMillis());
        }
    }

    public static void tenDown() throws InterruptedException {
        int num= 10;
        while (true ) {
            Thread.sleep(1000);
            System.out.println(num--);
            if (num⩽0) {
                break;
            }
        }
    }
}
```

• 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**

```java
// 测试礼让线程
// 礼让不一定成功，看cpu心情
public class TestYield {
    public static void main(String[] args) {
        MyYield myYield = new MyYield();

        new Thread(myYield, "a").start();
```

```
            new Thread(myYield, "b").start();
        }
    }

    class MyYield implements Runnable {

        @Override
        public void run() {
            System.out.println(Thread.currentThread().getName() + "线程开始执行");

            // 礼让
            Thread.yield();
            System.out.println(Thread.currentThread().getName() + "线程停止执行");

        }
    }
```

• 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

```
    public class TestJoin implements Runnable {
        @Override
        public void run() {
            for (int i = 0; i < 100; i++) {
                System.out.println("线程VIP来了" + i);
            }
        }

        public static void main(String[] args) throws InterruptedException {
            // 启动线程
            TestJoin testJoin = new TestJoin();
            Thread thread = new Thread(testJoin);
            thread.start();

            //主线程
            for (int i = 0; i < 1000; i++) {
                if (i == 200) {
                    // 插队
                    thread.join();
                }
                System.out.println("main" + i);
            }
        }
    }
```

• 线程状态观测

- Thread.State
  - **NEW**

    尚未启动的线程处于此状态
  - **RUNNABLE**

    在Java虚拟机中执行的线程处于此状态
  - **BLOCKED**

    被阻塞等待监视器锁定的线程处于此状态
  - **WAITING**

    正在等待另一个线程执行特定动作的线程处于此状态
  - **TIMED_WAITING**

    正在等待另一个线程执行动作达到指定等待时间的线程处于此状态
  - **TERMINATED**

    已退出的线程处于此状态
- 一个线程可以在给定时间点处于一个状态。这些状态是不反映任何操作系统线程状态的虚拟机状态。

```
// 观察测试线程的状态
public class TestState {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
            System.out.println(".....");
        });

        // 观察状态
        Thread.State state = thread.getState();
        System.out.println(state); // NEW

        // 观察启动后
        thread.start();
        state = thread.getState();
        System.out.println(state); // RUNNABLE

        // 只要线程不终止就一直输出状态
        while (state != Thread.State.TERMINATED) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
```

```
                throw new RuntimeException(e);
            }
            state = thread.getState();
            System.out.println(state);
        }
    }
}
```

• 线程优先级

- Java提供一个线程调度器来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;`
- 使用以下方法改变或获取优先级
  - `getPriority()`
  - `setPriority(int xxx)`

```
// 测试线程优先级
public class TestPriority {
    public static void main(String[] args) {
        // 主线程默认优先级
        System.out.println(Thread.currentThread().getName() + "-->" +
Thread.currentThread().getPriority());

        MyPriority myPriority = new MyPriority();

        Thread thread1 = new Thread(myPriority);
        Thread thread2 = new Thread(myPriority);
        Thread thread3 = new Thread(myPriority);
        Thread thread4 = new Thread(myPriority);
        Thread thread5 = new Thread(myPriority);
        Thread thread6 = new Thread(myPriority);

        // 先设置优先级再启动
        thread1.start();

        thread2.setPriority(1);
        thread2.start();

        thread3.setPriority(4);
        thread3.start();

        thread4.setPriority(Thread.MAX_PRIORITY); // 10
```

```
            thread4.start();

            thread5.setPriority(8);
            thread5.start();

            thread6.setPriority(7);
            thread6.start();
        }
    }

    class MyPriority implements Runnable {

        @Override
        public void run() {
            System.out.println(Thread.currentThread().getName() + "⟶" +
    Thread.currentThread().getPriority());
        }
    }
```

> “
>
> ⚠️优先级低只是意味着获得调度的概率低，并不是优先级低就不会被调用了，这都是看CPU的调度。

• 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、垃圾回收等等

```
// 测试守护线程
public class TestDaemon {
    public static void main(String[] args) {
        God god = new God();
        You you = new You();

        Thread thread = new Thread(god);
        thread.setDaemon(true); // 默认是false，表示是用户线程
        thread.start();

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

// 上帝
class God implements Runnable {
    @Override
```

```
        public void run() {
            while (true) {
                System.out.println("上帝包保佑者你");
            }
        }
    }

    // 你
    class You implements Runnable {
        @Override
        public void run() {
            for (int i = 0; i < 36500; i++) {
                System.out.println("你一生都开心的活着");
            }
            System.out.println("Goodbye World!");
        }
    }
```

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
- 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。

• 线程同步

- 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，也带来了访问冲突的问题，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他锁，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

```
    // 不安全地买票
    public class UnsafeBuyTicket {
        public static void main(String[] args) {
            BuyTicket station = new BuyTicket();
```

```
5
6           new Thread(station, "苦逼的我").start();
7           new Thread(station, "牛逼的你").start();
8           new Thread(station, "可恶的黄牛").start();
9       }
10  }
11
12  class BuyTicket implements Runnable {
13
14      // 票
15      private int ticketNums = 10;
16      boolean flag = true;
17
18      @Override
19      public void run() {
20          // 买票
21          while (flag) {
22              buy();
23          }
24      }
25
26      private void buy() {
27          // 判断是否有票
28          if (ticketNums <= 0) {
29              flag = false;
30              return;
31          }
32
33          // 模拟延时
34          try {
35              Thread.sleep(100);
36          } catch (InterruptedException e) {
37              throw new RuntimeException(e);
38          }
39
40          // 买票
41          System.out.println(Thread.currentThread().getName() + "拿到" +
    ticketNums--);
42      }
43  }
```

```
1   // 不安全的取钱
2   public class UnsafeBank {
3       public static void main(String[] args) {
4           Account account = new Account(100, "结婚基金");
5
6           Drawing you = new Drawing(account, 50, "你");
7           Drawing girlFriend = new Drawing(account, 100, "女朋友");
8
9           you.start();
```

```
10          girlFriend.start();
11      }
12  }
13
14  // 账户
15  class Account {
16      // 余额
17      int money;
18      // 卡名
19      String name;
20
21      public Account(int money, String name) {
22          this.money = money;
23          this.name = name;
24      }
25  }
26
27  // 银行：模拟取钱
28  class Drawing extends Thread {
29      // 账户
30      Account account;
31      // 取了多少钱
32      int drawingMoney;
33      // 现在手里有多少钱
34      int nowMoney;
35
36      public Drawing(Account account, int drawingMoney, String name) {
37          super(name);
38          this.account = account;
39          this.drawingMoney = drawingMoney;
40      }
41
42      // 取钱
43      @Override
44      public void run() {
45          // 判断银行有没有钱
46          if (account.money - drawingMoney < 0) {
47              System.out.println(Thread.currentThread().getName() + "钱不够，取不
    了");
48              return;
49          }
50
51          // 模拟延时
52          try {
53              Thread.sleep(1000);
54          } catch (InterruptedException e) {
55              throw new RuntimeException(e);
56          }
57
58          // 卡内余额 = 余额 - 你取的钱
59          account.money -= drawingMoney;
```

```
            // 你手里的钱
            nowMoney += drawingMoney;

            System.out.println(account.name + "账户余额为: " + account.money);
            // Thread.currentThread().getName() = this.getName()
            System.out.println(this.getName() + "手里的钱: " + nowMoney);
        }
    }
```

```
    import java.util.ArrayList;
    import java.util.List;

    // 线程不安全的集合
    public class UnsafeList {
        public static void main(String[] args) throws InterruptedException {
            List<String> list = new ArrayList<String>();
            for (int i = 0; i < 10000; i++) {
                new Thread(() -> {
                    list.add(Thread.currentThread().getName());
                }).start();
            }
            Thread.sleep(3000);
            System.out.println(list.size());
        }
    }
```

• 同步方法

- 由于我们可以通过 private 关键字来保证数据对象只能被方法访问，所以我们只需要针对方法提出一套机制，这套机制就是 synchronized 关键字，它包括两种用法：**synchronized 方法** 和 **synchronized 块**
- 同步方法：`public synchronized void method(int argd) {}`
- synchronized方法控制“对象”的访问，每个对象对应一把锁，每个synchronized方法都必须获得调用方法的对象的锁才能执行，否则线程就会阻塞，方法一旦执行，就独占该锁，直到该方法返回才释放锁，后面被阻塞的线程才能获得这个锁，继续执行。
- 缺陷：**若将一个大的方法声明为synchronized将会影响效率**

```
    // 不安全地买票
    public class UnsafeBuyTicket {
        public static void main(String[] args) {
            BuyTicket station = new BuyTicket();

            new Thread(station, "苦逼的我").start();
            new Thread(station, "牛逼的你").start();
            new Thread(station, "可恶的黄牛").start();
        }
```

```
}

class BuyTicket implements Runnable {

    // 票
    private int ticketNums = 10;
    boolean flag = true;

    @Override
    public void run() {
        // 买票
        while (flag) {
            buy();
        }
    }

    private synchronized void buy() {
        // 判断是否有票
        if (ticketNums <= 0) {
            flag = false;
            return;
        }

        // 模拟延时
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

        // 买票
        System.out.println(Thread.currentThread().getName() + "拿到" +
ticketNums--);
    }
}
```

同步方法弊端

- 方法里面需要修改的内容才需要锁，锁的太多，浪费资源

同步块

- 同步块：`synchronized (Obj) {}`
- Obj 称之为**同步监视器**
  - Obj 可以是任何对象，但是推荐使用共享资源作为同步监视器
  - 同步方法中无需指定同步监视器，因为同步方法的同步监视器就是this，就是这个对象本身，或是 class

- 同步监视器的执行过程

    1. 第一个线程访问，锁定同步监视器，执行其中代码
    2. 第二个线程访问，发现同步监视器被锁定，无法访问
    3. 第一个线程访问完毕，解锁同步监视器
    4. 第二个线程访问，发现同步监视器没有锁，然后锁定并访问

```java
// 不安全的取钱
public class UnsafeBank {
    public static void main(String[] args) {
        Account account = new Account(100, "结婚基金");

        Drawing you = new Drawing(account, 50, "你");
        Drawing girlFriend = new Drawing(account, 100, "女朋友");

        you.start();
        girlFriend.start();
    }
}

// 账户
class Account {
    // 余额
    int money;
    // 卡名
    String name;

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

// 银行：模拟取钱
class Drawing extends Thread {
    // 账户
    Account account;
    // 取了多少钱
    int drawingMoney;
    // 现在手里有多少钱
    int nowMoney;

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

    // 取钱
    @Override
```

```
    public void run() {
        synchronized (account) {
            // 判断银行有没有钱
            if (account.money - drawingMoney < 0) {
                System.out.println(Thread.currentThread().getName() + "钱不够，取不了");
                return;
            }

            // 模拟延时
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }

            // 卡内余额 = 余额 - 你取的钱
            account.money -= drawingMoney;
            // 你手里的钱
            nowMoney += drawingMoney;

            System.out.println(account.name + "账户余额为: " + account.money);

            // Thread.currentThread().getName() = this.getName()
            System.out.println(this.getName() + "手里的钱: " + nowMoney);
        }
    }
}
```

```
import java.util.ArrayList;
import java.util.List;

// 线程不安全的集合
public class UnsafeList {
    public static void main(String[] args) throws InterruptedException {
        List<String> list = new ArrayList<String>();
        for (int i = 0; i < 10000; i++) {
            new Thread(() -> {
                synchronized (list) {
                    list.add(Thread.currentThread().getName());
                }
            }).start();
        }
        Thread.sleep(3000);
        System.out.println(list.size());
    }
}
```

- **CopyOnWriteArrayList**

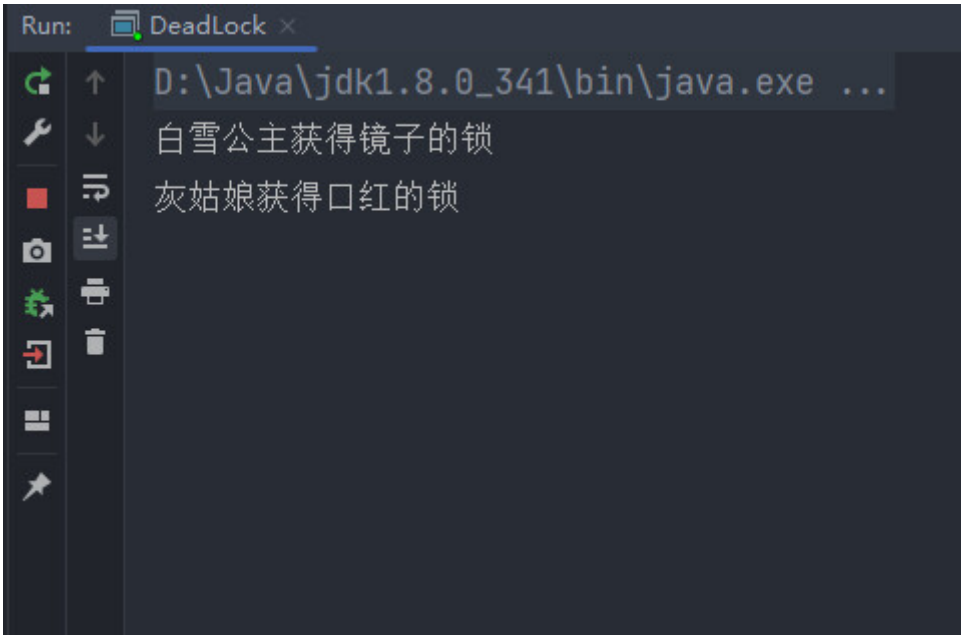
```
// 测试JUC安全类型的集合
public class TestJUC {
    public static void main(String[] args) {
        CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<String>
();
        for (int i = 0; i < 10000; i++) {
            new Thread(() -> {
                list.add(Thread.currentThread().getName());
            }).start();
        }
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println(list.size());
    }
}
```

- **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”的问题。

```
// 死锁：多个线程互相抱着对方需要的资源，然后形成僵持。
public class DeadLock {
    public static void main(String[] args) {
        Makeup g1 = new Makeup(0, "灰姑娘");
        Makeup g2 = new Makeup(1, "白雪公主");

        g1.start();
        g2.start();
    }
}

// 口红
class Lipstick {

}

// 镜子
class Mirror {

}

```

```java
class Makeup extends Thread {

    // 需要的资源只有一份，用static来保证只有一份
    static Lipstick lipstick = new Lipstick();
    static Mirror mirror = new Mirror();

    int choice;
    String girlName;

    Makeup(int choice, String girlName) {
        this.choice = choice;
        this.girlName = girlName;
    }

    @Override
    public void run() {
        // 化妆
        try {
            makeup();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    // 化妆，互相持有对方的锁，就是需要拿到对方的资源
    private void makeup() throws InterruptedException {
        if (choice == 0) {
            // 获得口红的锁
            synchronized (lipstick) {
                System.out.println(this.girlName + "获得口红的锁");
                Thread.sleep(1000);

                // 一秒钟后想获得镜子
                synchronized (mirror) {
                    System.out.println(this.girlName + "获得镜子的锁");
                }
            }
        } else {
            // 获得镜子的锁
            synchronized (mirror) {
                System.out.println(this.girlName + "获得镜子的锁");
                Thread.sleep(2000);

                // 一秒钟后想获得口红
                synchronized (lipstick) {
                    System.out.println(this.girlName + "获得口红的锁");
                }
            }
        }
    }
}
```

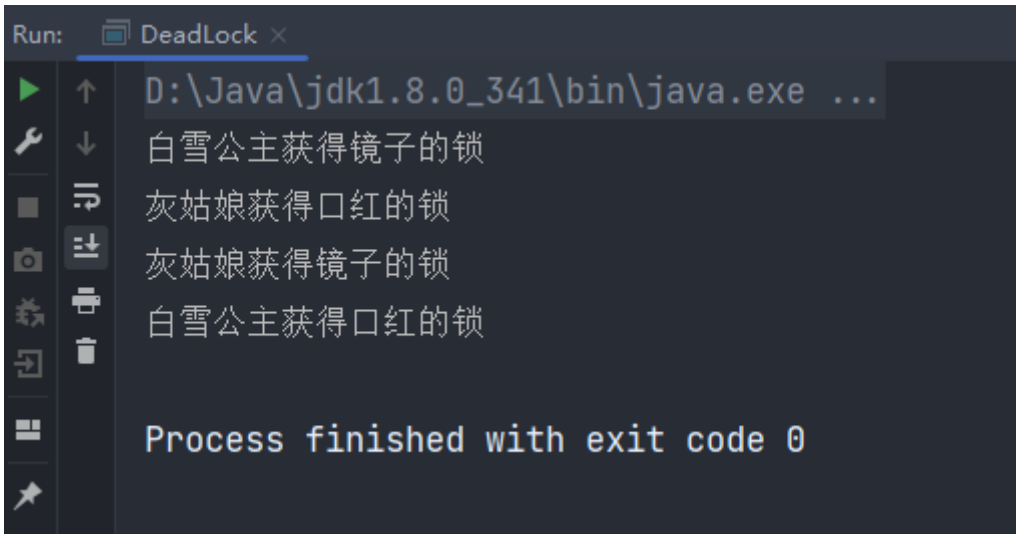


死锁避免方法

- 产生死锁的四个必要条件
  1. **互斥条件**：一个资源一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
- **只需想办法破坏上述四个条件中的任意一个或多个条件就可以避免死锁**

```java
// 化妆，互相持有对方的锁，就是需要拿到对方的资源
private void makeup() throws InterruptedException {
    if (choice == 0) {
        // 获得口红的锁
        synchronized (lipstick) {
            System.out.println(this.girlName + "获得口红的锁");
            Thread.sleep(1000);
        }
        // 一秒钟后想获得镜子
        synchronized (mirror) {
            System.out.println(this.girlName + "获得镜子的锁");
        }
    } else {
        // 获得镜子的锁
        synchronized (mirror) {
            System.out.println(this.girlName + "获得镜子的锁");
            Thread.sleep(2000);
        }
        // 一秒钟后想获得口红
        synchronized (lipstick) {
            System.out.println(this.girlName + "获得口红的锁");
        }
    }
```

```
24  }
```



- **Lock（锁）**

  - 从JDK5.0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 [1] 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。

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

public class TestLock {
    public static void main(String[] args) {
        TestLock2 testLock2 = new TestLock2();

        new Thread(testLock2).start();
        new Thread(testLock2).start();
        new Thread(testLock2).start();
    }
}

class TestLock2 implements Runnable {

    int ticketNums = 10;

    // 定义Lock锁
    private final ReentrantLock lock = new ReentrantLock();

    @Override
    public void run() {
```

```
22          while (true) {
23              try {
24                  // 加锁
25                  lock.lock();
26
27                  if (ticketNums > 0) {
28                      try {
29                          Thread.sleep(1000);
30                      } catch (InterruptedException e) {
31                          throw new RuntimeException(e);
32                      }
33                      System.out.println(ticketNums--);
34                  } else {
35                      break;
36                  }
37              } finally {
38                  // 解锁
39                  lock.unlock();
40              }
41          }
42      }
43  }
```

synchronized 与 Lock 对比

- Lock 时显式锁（手动开启和关闭锁）；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，分配了相应的资源）> 同步方法（在方法体之外）

线程协作

生产者消费者模式

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
- 分析：
    - 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，互为条件**
    - 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）
- Java提供了几个方法解决线程之间的通信问题

方法名	作用
wait()	表示线程一直等待，直到其他线程通知，与sleep不同，会释放锁
wait(long timeout)	指定等待的毫秒数
notify()	唤醒一个处于等待状态的线程
notifyAll()	唤醒同一个对象上所有调用wait()方法的线程，优先级别高的线程优先调度

> “
> ⚠️注意：
> 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

并发协作模型“生产者/消费者模式”---> 管程法

- 生产者：负责生产数据的模块（可能是方法、对象、线程、进程）
- 消费者：负责处理数据的模块（可能是方法、对象、线程、进程）
- 缓冲区：消费者不能直接使用生产者的数据，它们之间有个“缓冲区“
- **生产者将生产好的数据放入缓冲区，消费者从缓冲区拿数据**

```
public class TestPC {
    public static void main(String[] args) {
        SynContainer container = new SynContainer();

        new Producer(container).start();
        new Consumer(container).start();
```

```
        }
    }

    // 生产者
    class Producer extends Thread {
        SynContainer container;

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

        // 生产
        @Override
        public void run() {
            for (int i = 0; i < 100; i++) {
                container.push(new Chicken(i));
                System.out.println("生产了" + i + "只鸡");
            }
        }
    }

    // 消费者
    class Consumer extends Thread {
        SynContainer container;

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

        // 消费
        @Override
        public void run() {
            for (int i = 0; i < 100; i++) {
                System.out.println("消费了-->" + container.pop().id + "只鸡");
            }
        }
    }

    // 产品
    class Chicken {
        // 产品编号
        int id;

        public Chicken(int id) {
            this.id = id;
        }
    }

    // 缓冲区
    class SynContainer {
        // 容器大小
```

```java
    Chicken[] chickens = new Chicken[10];

    // 容器计数器
    int count = 0;

    // 生产者放入产品
    public synchronized void push(Chicken chicken) {
        // 如果容器满了，就需要等待消费者消费
        if (count == chickens.length) {
            // 通知消费者消费，生产者等待
            try {
                this.wait();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }

        // 如果没有满，就需要生产者放入产品
        chickens[count] = chicken;
        count++;

        // 可以通知消费者消费了
        this.notifyAll();
    }

    // 消费者消费产品
    public synchronized Chicken pop() {
        // 判断能否消费
        if (count == 0) {
            // 等待生产者生产，消费者等待
            try {
                this.wait();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }

        // 如果可以消费
        count--;
        Chicken chicken = chickens[count];

        // 吃完了，通知生产者生产
        this.notifyAll();

        return chicken;
    }
}
```

- **解决方式2**
  - 并发协作模型“生产者/消费者模式”---> 信号灯法

```java
public class TestPC2 {
    public static void main(String[] args) {
        TV tv = new TV();

        new Player(tv).start();
        new Watcher(tv).start();
    }
}

// 生产者 ⟶ 演员
class Player extends Thread {
    TV tv;

    public Player(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");
            } else {
                this.tv.play("抖音: 记录美好生活");
            }
        }
    }
}

// 消费者 ⟶ 观众
class Watcher extends Thread {
    TV tv;

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

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

// 产品 ⟶ 节目
```

```java
class TV {

    // 表演的节目
    String voice;

    boolean flag = true;

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

        if (!flag) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }

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

    // 观看
    public synchronized void watch() {
        if (flag) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
        System.out.println("观众观看了: " + voice);

        // 观看完毕通知演员表演
        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。
  - 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
  - `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

- JDK5.0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
  - `<T> Future<T> submit(Callable<T> task)`：执行任务，有返回值，一般用来执行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

```
import org.apache.commons.io.FileUtils;

import java.io.IOException;
import java.util.concurrent.*;

public class TestCallable implements Callable<Boolean> {

    private String url; // 网络图片地址
    private String name; // 保存的文件名

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        TestCallable t1 = new TestCallable("https://blog.kuangstudy.con/usr/themes/handsome/usr/img/sj/2.jpg", "1.jpg");
        TestCallable t2 = new TestCallable("https://blog.kuangstudy.con/usr/themes/handsome/usr/img/sj/2.jpg", "2.jpg");
        TestCallable t3 = new TestCallable("https://blog.kuangstudy.con/usr/themes/handsome/usr/img/sj/2.jpg", "3.jpg");

        // 创建执行服务
        ExecutorService ser = Executors.newFixedThreadPool(3);

        // 提交执行
        Future<Boolean> result1 = ser.submit(t1);
```

```
        Future<Boolean> result2 = ser.submit(t2);
        Future<Boolean> result3 = ser.submit(t3);

        // 获取结果
        boolean r1 = result1.get();
        boolean r2 = result2.get();
        boolean r3 = result3.get();

        System.out.println(r1);
        System.out.println(r2);
        System.out.println(r3);

        // 关闭服务
        ser.shutdownNow();
    }

    public TestCallable(String url, String name) {
        this.url = url;
        this.name = name;
    }

    // 下载图片线程的执行体
    @Override
    public Boolean call() throws Exception {
        WenDownloader wenDownloader = new WenDownloader();
        wenDownloader.downloader(url, name);
        System.out.println("下载了文件名为: " + name);
        return true;
    }
}

// 下载器
class WenDownloader {
    // 下载方法
    public void downloader() {
        try {
            FileUtils.copyURLToFile(new URL(url), new File(name));
        } catch (IOException e) {
            e.printStackTrace();
            System.out.println("IO异常, downloader方法出现问题");
        }
    }
}
```

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

public class TestPool {
    public static void main(String[] args) {
        // 创建线程池
```

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

        executorService.execute(new MyThread());
        executorService.execute(new MyThread());
        executorService.execute(new MyThread());
        executorService.execute(new MyThread());

        // 关闭连接
        executorService.shutdown();
    }
}

class MyThread implements Runnable {

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

---

1. 可重入锁 ↩