

**【线程池、Lambda表达式】**

**内容**

等待与唤醒案例

线程池

Lambda表达式

**目标**

能够理解线程通信概念

能够理解等待唤醒机制

能够描述Java中线程池运行原理

能够理解函数式编程相对于面向对象的优点

能够掌握Lambda表达式的标准格式

能够使用Lambda标准格式使用Runnable 与Comparator接口

能够掌握Lambda表达式的省略格式与规则

能够使用Lambda省略格式使用Runnable 与Comparator接口

能够通过Lambda的标准格式使用自定义的接口（有且仅有一个抽象方法）

能够通过Lambda的省略格式使用自定义的接口（有且仅有一个抽象方法）

能够明确Lambda的两项使用前提

**第一章 等待唤醒机制**

**1.1 线程间通信**

**概念：**多个线程在处理同一个资源，但是处理的动作（线程的任务）却不相同。

比如：线程A用来生成包子的，线程B用来吃包子的，包子可以理解为同一资源，线程A与线程B处理的动作，一个 是生产，一个是消费，那么线程A与线程B之间就存在线程通信问题。

**为什么要处理线程间通信：**

多个线程并发执行时, 在默认情况下CPU是随机切换线程的，当我们需要多个线程来共同完成一件任务，并且我们 希望他们有规律的执行, 那么多线程之间需要一些协调通信，以此来帮我们达到多线程共同操作一份数据。

**如何保证线程间通信有效利用资源：**

多个线程在处理同一个资源，并且任务不同时，需要线程通信来帮助解决线程之间对同一个变量的使用或操作。 就 是多个线程在操作同一份数据时， 避免对同一共享变量的争夺。也就是我们需要通过一定的手段使各个线程能有效 的利用资源。而这种手段即—— **等待唤醒机制。**

**1.2 等待唤醒机制**

**什么是等待唤醒机制**

这是多个线程间的一种**协作**机制。谈到线程我们经常想到的是线程间的**竞争（race）**，比如去争夺锁，但这并不是 故事的全部，线程间也会有协作机制。就好比在公司里你和你的同事们，你们可能存在在晋升时的竞争，但更多时 候你们更多是一起合作以完成某些任务。

就是在一个线程进行了规定操作后，就进入等待状态（**wait()**）， 等待其他线程执行完他们的指定代码过后 再将 其唤醒（**notify()**）;在有多个线程进行等待时， 如果需要，可以使用 notifyAll()来唤醒所有的等待线程。

wait/notify 就是线程间的一种协作机制。

**等待唤醒中的方法**

等待唤醒机制就是用于解决线程间通信的问题的，使用到的3个方法的含义如下：

1. wait：线程不再活动，不再参与调度，进入 wait set 中，因此不会浪费 CPU 资源，也不会去竞争锁了，这时 的线程状态即是 WAITING。它还要等着别的线程执行一个**特别的动作**，也即是“**通知（notify）**”在这个对象 上等待的线程从wait set 中释放出来，重新进入到调度队列（ready queue）中
2. notify：则选取所通知对象的 wait set 中的一个线程释放；例如，餐馆有空位置后，等候就餐最久的顾客最先 入座。
3. notifyAll：则释放所通知对象的 wait set 上的全部线程。

注意：

哪怕只通知了一个等待的线程，被通知线程也不能立即恢复执行，因为它当初中断的地方是在同步块内，而 此刻它已经不持有锁，所以她需要再次尝试去获取锁（很可能面临其它线程的竞争），成功后才能在当初调 用 wait 方法之后的地方恢复执行。

总结如下：

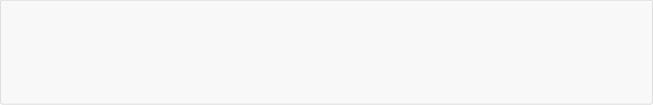
如果能获取锁，线程就从 WAITING 状态变成 RUNNABLE 状态；

否则，从 wait set 出来，又进入 entry set，线程就从 WAITING 状态又变成 BLOCKED 状态

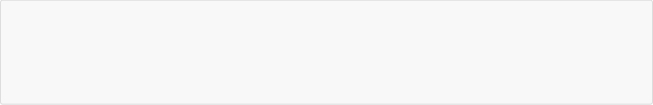
**调用wait和notify方法需要注意的细节**

1. wait方法与notify方法必须要由同一个锁对象调用。因为：对应的锁对象可以通过notify唤醒使用同一个锁对 象调用的wait方法后的线程。
2. wait方法与notify方法是属于Object 类的方法的。因为：锁对象可以是任意对象，而任意对象的所属类都是继 承了Object类的。
3. wait方法与notify方法必须要在同步代码块或者是同步函数中使用。因为：必须要通过锁对象调用这2个方 法。

**1.3 生产者与消费者问题**



 包子铺线程生产包子，吃货线程消费包子。当包子没有时（包子状态为false），吃货线程等待，包子铺线程生产包子 （即包子状态为true），并通知吃货线程（解除吃货的等待状态）,因为已经有包子了，那么包子铺线程进入等待状态。 接下来，吃货线程能否进一步执行则取决于锁的获取情况。如果吃货获取到锁，那么就执行吃包子动作，包子吃完（包 子状态为false），并通知包子铺线程（解除包子铺的等待状态）,吃货线程进入等待。包子铺线程能否进一步执行则取 决于锁的获取情况。

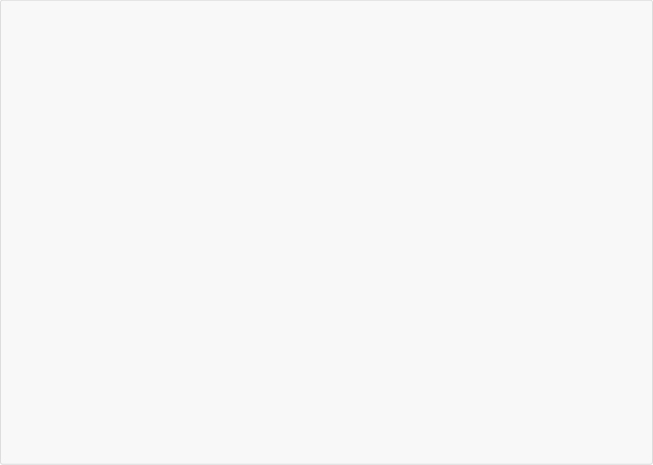


 public class BaoZi {

     String  pier ;

     String  xianer ;

     boolean  flag = false ;//包子资源 是否存在  包子资源状态 }



**1.3 生产者与消费者问题**

等待唤醒机制其实就是经典的“生产者与消费者”的问题。 就拿生产包子消费包子来说等待唤醒机制如何有效利用资源：

**代码演示：** 包子资源类：

吃货线程类：

 public class ChiHuo extends Thread{

    private BaoZi bz;

    public ChiHuo(String name,BaoZi bz){

        super(name);

        this.bz = bz;

    }

    @Override

    public void run() {

        while(true){

            synchronized (bz){

                if(bz.flag == false){//没包子

                    try {

                        bz.wait();

                    } catch (InterruptedException e) {                         e.printStackTrace();

                    }

                }

                System.out.println ("吃货正在吃"+bz.pier+bz.xianer+"包子");                 bz.flag = false;

                bz.notify();

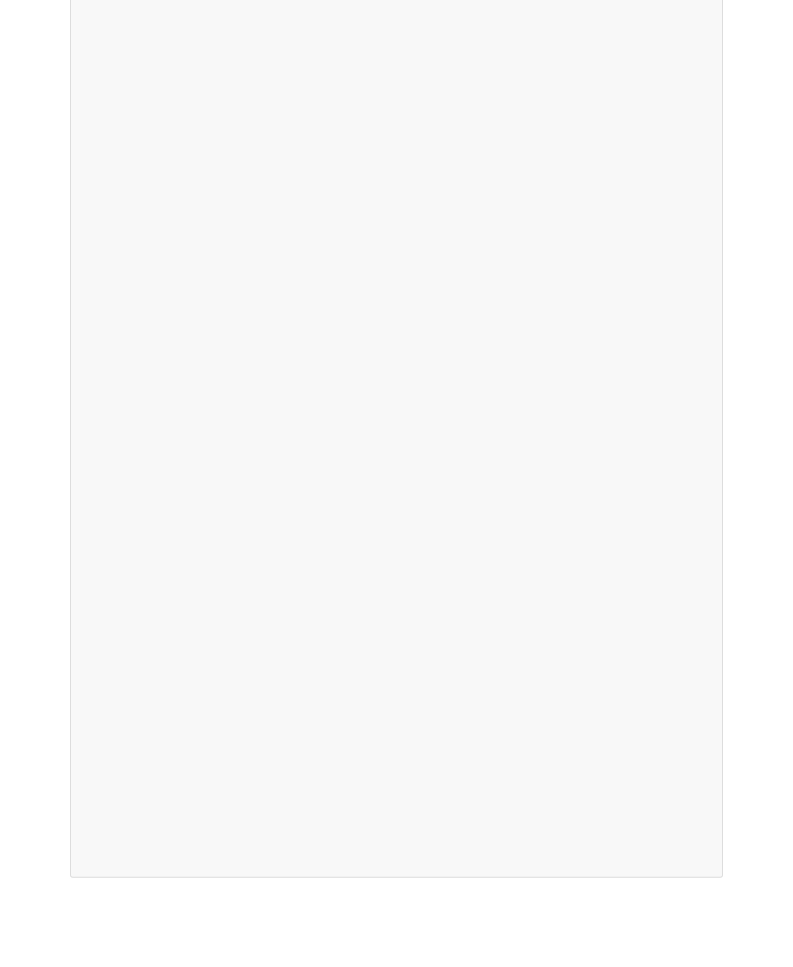
            }

        }

    }

}

包子铺线程类：



 public class BaoZiPu extends Thread {

    private BaoZi bz;

    public BaoZiPu (String  name,BaoZi bz){

        super(name);

        this.bz = bz;

    }

    @Override

    public void run() {

        int count = 0;

        //造包子

        while(true){

            //同步

            synchronized (bz){

                if(bz.flag == true){//包子资源  存在                     try {

                        bz.wait();

                    } catch (InterruptedException e) {                         e.printStackTrace();

                    }

                }

                // 没有包子  造包子

                System.out.println ("包子铺开始做包子");                 if(count%2 == 0){

                    // 冰皮  五仁

                    bz.pier = "冰皮";

                    bz.xianer = "五仁";

                }else{

                    // 薄皮  牛肉大葱

                    bz.pier = "薄皮";

                    bz.xianer = "牛肉大葱";

                }

                count++;

                bz.flag=true;

                System.out.println ("包子造好了："+bz.pier+bz.xianer);                 System.out.println ("吃货来吃吧");

                //唤醒等待线程 （吃货）

                bz.notify();

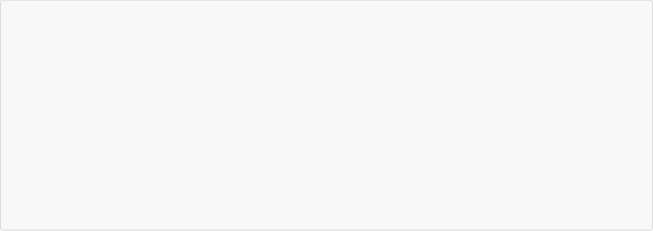
            }

        }

    }

}

测试类：



 public class Demo {

    public static void main(String[] args) {         //等待唤醒案例

        BaoZi bz = new BaoZi();

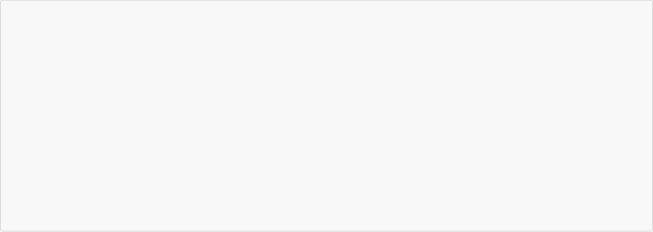
        ChiHuo ch = new ChiHuo("吃货",bz);         BaoZiPu bzp = new BaoZiPu("包子铺",bz);

        ch.start();

        bzp.start();

    }

}



 包子铺开始做包子

包子造好了：冰皮五仁 吃货来吃吧

吃货正在吃冰皮五仁包子 包子铺开始做包子

包子造好了：薄皮牛肉大葱 吃货来吃吧

吃货正在吃薄皮牛肉大葱包子 包子铺开始做包子

包子造好了：冰皮五仁 吃货来吃吧

吃货正在吃冰皮五仁包子

执行效果：

**第二章 线程池 2.1 线程池思想概述**



我们使用线程的时候就去创建一个线程，这样实现起来非常简便，但是就会有一个问题：

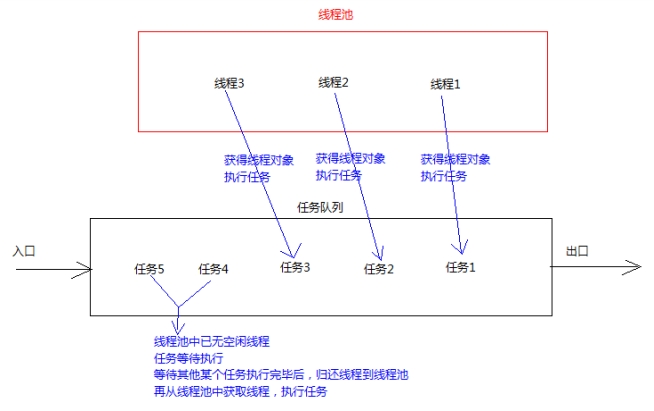
如果并发的线程数量很多，并且每个线程都是执行一个时间很短的任务就结束了，这样频繁创建线程就会大大降低 系统的效率，因为频繁创建线程和销毁线程需要时间。

那么有没有一种办法使得线程可以复用，就是执行完一个任务，并不被销毁，而是可以继续执行其他的任务？ 在Java中可以通过线程池来达到这样的效果。今天我们就来详细讲解一下Java的线程池。

**2.2 线程池概念**

**线程池：**其实就是一个容纳多个线程的容器，其中的线程可以反复使用，省去了频繁创建线程对象的操作， 无需反复创建线程而消耗过多资源。

由于线程池中有很多操作都是与优化资源相关的，我们在这里就不多赘述。我们通过一张图来了解线程池的工作原 理：



合理利用线程池能够带来三个好处：

1. 降低资源消耗。减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。
2. 提高响应速度。当任务到达时，任务可以不需要的等到线程创建就能立即执行。
3. 提高线程的可管理性。可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为消耗过多的内 存，而把服务器累趴下(每个线程需要大约1MB内存，线程开的越多，消耗的内存也就越大，最后死机)。

**2.3 线程池的使用**

Java里面线程池的顶级接口是 java.util.concurrent.Executor ，但是严格意义上讲 Executor 并不是一个线程 池，而只是一个执行线程的工具。真正的线程池接口是 java.util.concurrent.ExecutorService 。

要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，很有可能配置的线程池不是较优 的，因此在 java.util.concurrent.Executors 线程工厂类里面提供了一些静态工厂，生成一些常用的线程池。官 方建议使用Executors工程类来创建线程池对象。

Executors类中有个创建线程池的方法如下： public static ExecutorService newFixedThreadPool(int nThreads) ：返回线程池对象。(创建的是有界线

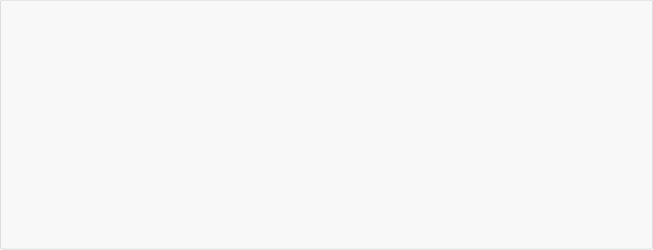
程池,也就是池中的线程个数可以指定最大数量)

获取到了一个线程池ExecutorService 对象，那么怎么使用呢，在这里定义了一个使用线程池对象的方法如下： public Future<?> submit(Runnable task) :获取线程池中的某一个线程对象，并执行

Future接口：用来记录线程任务执行完毕后产生的结果。线程池创建与使用。

使用线程池中线程对象的步骤：

1. 创建线程池对象。
2. 创建Runnable 接口子类对象。(task)



 public class MyRunnable implements Runnable {

    @Override

    public void run() {

        System.out.println ("我要一个教练");

        try {

            Thread.sleep(2000);

        } catch (InterruptedException  e) {

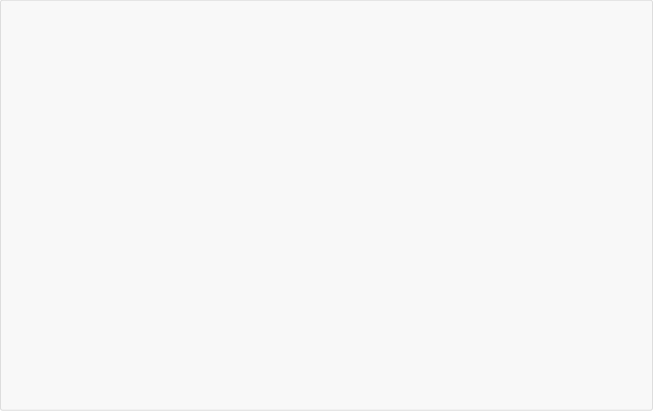
            e.printStackTrace();

        }

        System.out.println ("教练来了： " + Thread.currentThread ().getName());         System.out.println ("教我游泳,交完后，教练回到了游泳池");

    }

}



3. 提交Runnable 接口子类对象。(take task) 4. 关闭线程池(一般不做)。

Runnable 实现类代码：

线程池测试类：

 public class ThreadPoolDemo  {

    public static void main(String[] args) {

        // 创建线程池对象

        ExecutorService service = Executors.newFixedThreadPool(2);//包含2个线程对象         // 创建Runnable 实例对象

        MyRunnable r = new MyRunnable();

        //自己创建线程对象的方式

        // Thread t = new Thread(r);

        // t.start(); ‐‐‐> 调用MyRunnable中的run()

        // 从线程池中获取线程对象,然后调用MyRunnable中的run()

        service.submit(r);

        // 再获取个线程对象，调用MyRunnable中的run()

        service.submit(r);

        service.submit(r);

        // 注意：submit方法调用结束后，程序并不终止，是因为线程池控制了线程的关闭。         // 将使用完的线程又归还到了线程池中

        // 关闭线程池

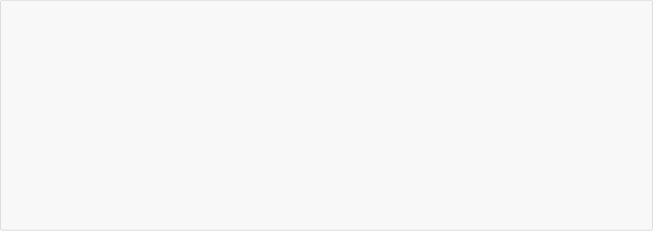
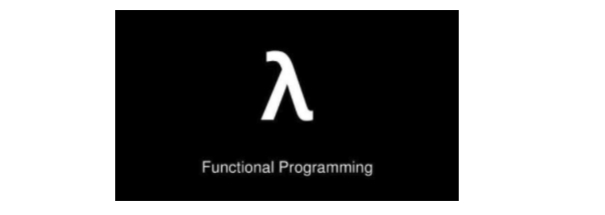
        //service.shutdown();

    }

}

**第三章 Lambda表达式**

**3.1 函数式编程思想概述**



 public class Demo01Runnable  {

    public static void main(String[] args) {

        // 匿名内部类

        Runnable  task = new Runnable() {             @Override

            public void run() { // 覆盖重写抽象方法                 System.out.println ("多线程任务执行！");             }

        };

        new Thread(task).start(); // 启动线程

    }

}



在数学中，**函数**就是有输入量、输出量的一套计算方案，也就是“拿什么东西做什么事情”。相对而言，面向对象过 分强调“必须通过对象的形式来做事情”，而函数式思想则尽量忽略面向对象的复杂语法——**强调做什么，而不是以 什么形式做**。

面向对象的思想:

做一件事情,找一个能解决这个事情的对象,调用对象的方法,完成事情.

函数式编程思想:

只要能获取到结果,谁去做的,怎么做的都不重要,重视的是结果,不重视过程

**3.2 冗余的Runnable代码**

**传统写法**

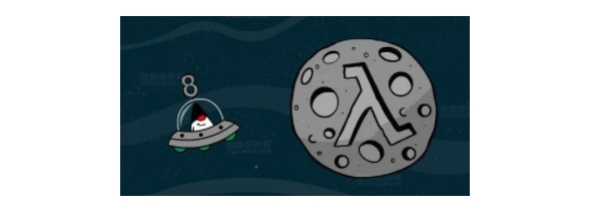
当需要启动一个线程去完成任务时，通常会通过 java.lang.Runnable 接口来定义任务内容，并使用 java.lang.Thread 类来启动该线程。代码如下：

本着“一切皆对象”的思想，这种做法是无可厚非的：首先创建一个 Runnable 接口的匿名内部类对象来指定任务内 容，再将其交给一个线程来启动。

**代码分析**

对于 Runnable 的匿名内部类用法，可以分析出几点内容：

Thread 类需要 Runnable 接口作为参数，其中的抽象 run 方法是用来指定线程任务内容的核心；



为了省去定义一个 实现类的麻烦，**不得不**使用匿名内部类；

必须覆盖重写抽象 以方法名称、方法参数、方法返回值**不得不**再写一遍，且不能写错； 而实际上，**似乎只有方法体才是关键所在**。

**3.3 编程思想转换**

**做什么，而不是怎么做**

我们真的希望创建一个匿名内部类对象吗？不。我们只是为了做这件事情而**不得不**创建一个对象。我们真正希望做 的事情是：将 run 方法体内的代码传递给 Thread 类知晓。

**传递一段代码**——这才是我们真正的目的。而创建对象只是受限于面向对象语法而不得不采取的一种手段方式。 那，有没有更加简单的办法？如果我们将关注点从“怎么做”回归到“做什么”的本质上，就会发现只要能够更好地达 到目的，过程与形式其实并不重要。

**生活举例**

为了指定 run 的方法体，**不得不**需要 Runnable 接口的实现类；

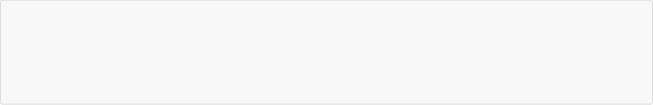
|  |  |
| --- | --- |
| RunnableImpl | |
| run | 方法，所 |

当我们需要从北京到上海时，可以选择高铁、汽车、骑行或是徒步。我们的真正目的是到达上海，而如何才能到达 上海的形式并不重要，所以我们一直在探索有没有比高铁更好的方式——搭乘飞机。

而现在这种飞机（甚至是飞船）已经诞生：2014年3月Oracle所发布的Java 8（JDK 1.8）中，加入了**Lambda表达 式**的重量级新特性，为我们打开了新世界的大门。

**3.4 体验Lambda的更优写法**

借助Java 8的全新语法，上述 Runnable 接口的匿名内部类写法可以通过更简单的Lambda表达式达到等效：



 public class Demo02LambdaRunnable  {

    public static void main(String[] args) {

        new Thread(() ‐> System.out.println("多线程任务执行！")).start(); // 启动线程     }

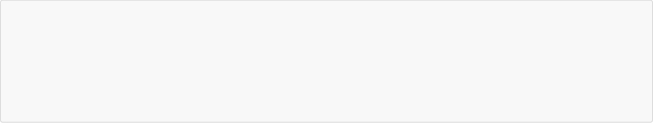
}



 () ‐> System.out.println("多线程任务执行！")



public Thread(Runnable target)

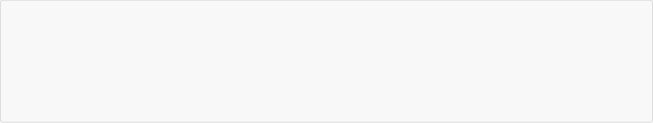


 public class RunnableImpl  implements Runnable {     @Override

    public void run() {

        System.out.println("多线程任务执行！");     }

}



 public class Demo03ThreadInitParam  {

    public static void main(String[] args) {         Runnable  task = new RunnableImpl();         new Thread(task).start();

    }

}



这段代码和刚才的执行效果是完全一样的，可以在1.8或更高的编译级别下通过。从代码的语义中可以看出：我们 启动了一个线程，而线程任务的内容以一种更加简洁的形式被指定。

不再有“不得不创建接口对象”的束缚，不再有“抽象方法覆盖重写”的负担，就是这么简单！

**3.5 回顾匿名内部类**

Lambda是怎样击败面向对象的？在上例中，核心代码其实只是如下所示的内容：

为了理解Lambda的语义，我们需要从传统的代码起步。

**使用实现类**

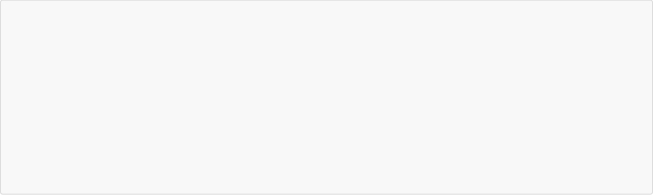
要启动一个线程，需要创建一个 Thread 类的对象并调用 start 方法。而为了指定线程执行的内容，需要调用 Thread 类的构造方法：

为了获取 Runnable 接口的实现对象，可以为该接口定义一个实现类 RunnableImpl ：

然后创建该实现类的对象作为 Thread 类的构造参数：

**使用匿名内部类**

这个 RunnableImpl 类只是为了实现 Runnable 接口而存在的，而且仅被使用了唯一一次，所以使用匿名内部类的 语法即可省去该类的单独定义，即匿名内部类：



 public class Demo04ThreadNameless  {

    public static void main(String[] args) {         new Thread(new Runnable() {

            @Override

            public void run() {

                System.out.println ("多线程任务执行！");             }

        }).start();

    }

}



public abstract void run();



 () ‐> System.out.println("多线程任务执行！")



 (参数类型 参数名称) ‐> { 代码语句 }



**匿名内部类的好处与弊端**

一方面，匿名内部类可以帮我们**省去实现类的定义**；另一方面，匿名内部类的语法——**确实太复杂了！ 语义分析**

仔细分析该代码中的语义， Runnable 接口只有一个 run 方法的定义：

即制定了一种做事情的方案（其实就是一个函数）：

**无参数**：不需要任何条件即可执行该方案。

**无返回值**：该方案不产生任何结果。

**代码块**（方法体）：该方案的具体执行步骤。

同样的语义体现在 Lambda 语法中，要更加简单：

前面的一对小括号即 run 方法的参数（无），代表不需要任何条件；

中间的一个箭头代表将前面的参数传递给后面的代码；

后面的输出语句即业务逻辑代码。

**3.6 Lambda标准格式**

Lambda省去面向对象的条条框框，格式由**3个部分**组成：

一些参数

一个箭头

一段代码

Lambda表达式的**标准格式**为：

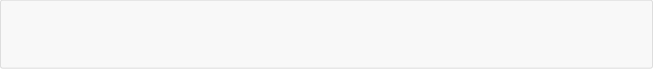
格式说明：

小括号内的语法与传统方法参数列表一致：无参数则留空；多个参数则用逗号分隔。

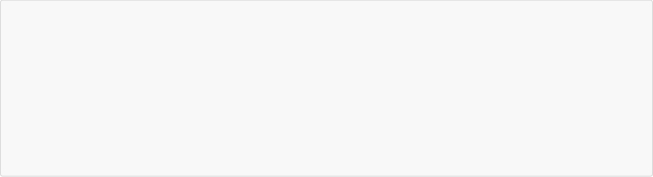
-> 是新引入的语法格式，代表指向动作。

大括号内的语法与传统方法体要求基本一致。

**3.7 练习：使用Lambda标准格式（无参无返回）**



 public interface Cook {     void makeFood(); }



 public class Demo05InvokeCook {

    public static void main(String[] args) {

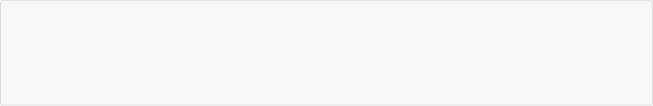
        // TODO 请在此使用Lambda【标准格式】调用invokeCook方法     }

    private static void invokeCook(Cook cook) {

        cook.makeFood();

    }

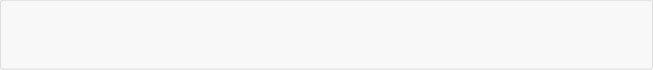
}



 public static void main(String[] args) {     invokeCook (() ‐> {

        System.out.println ("吃饭啦！");     });

}



 需求:

    使用数组存储多个Person对象

    对数组中的Person对象使用Arrays的sort方法通过年龄进行升序排序



public abstract int compare(T o1, T o2);



当需要对一个对象数组进行排序时， 方法需要一个 Comparator 接口实例来指定排序的规则。假设有 一个 Person 类，含有 String name 和 个成员变量：

**3.7 练习：使用Lambda标准格式（无参无返回） 题目**

给定一个厨子 Cook 接口，内含唯一的抽象方法 makeFood ，且无参数、无返回值。如下：

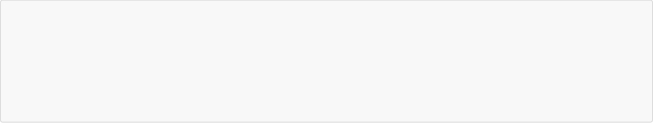
在下面的代码中，请使用Lambda的**标准格式**调用 invokeCook 方法，打印输出“吃饭啦！”字样：

**解答**

备注：小括号代表 Cook 接口 makeFood 抽象方法的参数为空，大括号代表 makeFood 的方法体。 **3.8 Lambda的参数和返回值**

下面举例演示 java.util.Comparator<T> 接口的使用场景代码，其中的抽象方法定义为：

|  |  |  |
| --- | --- | --- |
| Arrays.sort | | |
|  | int age | 两 |

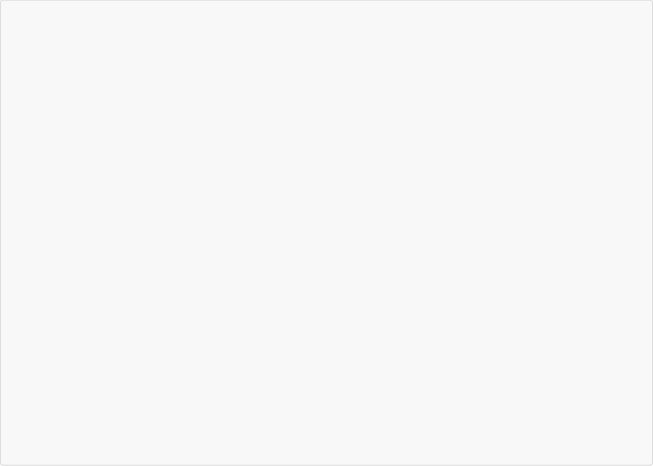


 public class Person {

    private String name;

    private int age;

    // 省略构造器、toString 方法与Getter Setter }



为了排序， Arrays.sort 方法需要排序规 接口的实例，抽象方法 compare 是关键； 为了指定 compare 的方法体，**不得不**需要 实现类；

**传统写法**

如果使用传统的代码对 Person[] 数组进行排序，写法如下：

 import java.util.Arrays;

import java.util.Comparator ;

public class Demo06Comparator {

    public static void main(String[] args) {

        // 本来年龄乱序的对象数组

        Person[] array = {

            new Person("古力娜扎", 19),

            new Person("迪丽热巴", 18),

            new Person("马尔扎哈", 20) };

        // 匿名内部类

        Comparator<Person> comp = new Comparator<Person>() {

            @Override

            public int compare(Person o1, Person o2) {

                return o1.getAge() ‐ o2.getAge();

            }

        };

        Arrays.sort(array, comp); // 第二个参数为排序规则，即Comparator 接口实例

        for (Person person : array) {

            System.out.println (person);

        }

    }

}

这种做法在面向对象的思想中，似乎也是“理所当然”的。其中 Comparator 接口的实例（使用了匿名内部类）代表 了“按照年龄从小到大”的排序规则。

**代码分析**

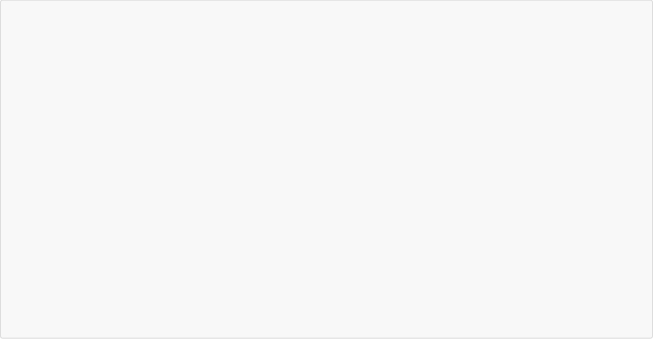
下面我们来搞清楚上述代码真正要做什么事情。

为了省去定义一个 ComparatorImpl 实现类的麻烦，**不得不**使用匿名内部类；

必须覆盖重写抽象 compare 方法，所以方法名称、方法参数、方法返回值**不得不**再写一遍，且不能写错； 实际上，**只有参数和方法体才是关键**。

**Lambda写法**

|  |  |  |
| --- | --- | --- |
| 则，即 | Comparator | |
| Comparator | | 接口的 |



 import java.util.Arrays;

public class Demo07ComparatorLambda {

    public static void main(String[] args) {         Person[] array = {

            new Person("古力娜扎", 19),

            new Person("迪丽热巴", 18),

            new Person("马尔扎哈", 20) };

        Arrays.sort(array, (Person a,Person b)‐> {             return a.getAge() ‐ b.getAge();         });

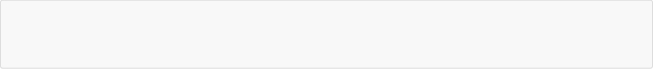
        for (Person person : array) {

            System.out.println (person);

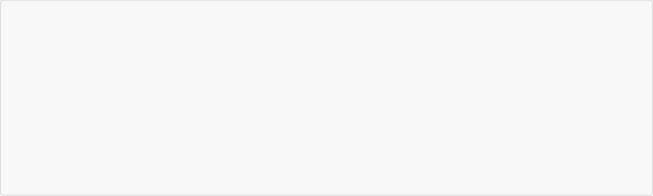
        }

    }

}



 public interface Calculator {     int calc(int a, int b); }



 public class Demo08InvokeCalc {

    public static void main(String[] args) {

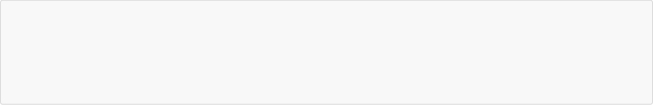
        // TODO 请在此使用Lambda【标准格式】调用invokeCalc 方法来计算120+130的结果ß     }

    private static void invokeCalc(int a,int b, Calculator  calculator) {         int result = calculator.calc(a, b);

        System.out.println ("结果是：" + result);

    }

}



 public static void main(String[] args) {     invokeCalc (120, 130, (int a,int b) ‐> {         return a + b;

    });

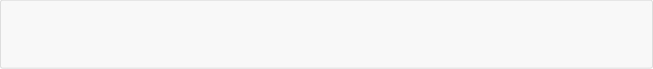
}

**3.9 练习：使用Lambda标准格式（有参有返回） 题目**

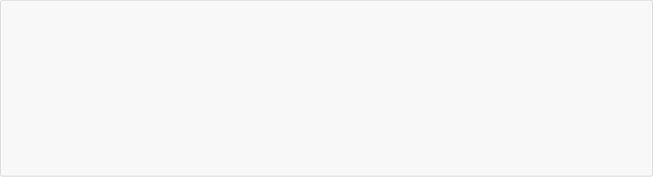
给定一个计算器 Calculator 接口，内含抽象方法 calc 可以将两个int数字相加得到和值：

在下面的代码中，请使用Lambda的**标准格式**调用 invokeCalc 方法，完成120和130的相加计算：

**解答**



 public static void main(String[] args) {     invokeCalc(120, 130, (a, b) ‐> a + b); }



 public class Demo09InvokeCook {

    public static void main(String[] args) {

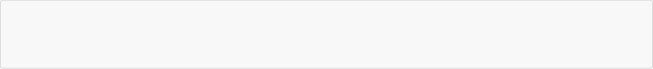
        // TODO 请在此使用Lambda【省略格式】调用invokeCook方法     }

    private static void invokeCook(Cook cook) {

        cook.makeFood();

    }

}



 public static void main(String[] args) {

    invokeCook(() ‐> System.out.println("吃饭啦！")); }

备注：小括号代表 Calculator 接口 calc 抽象方法的参数，大括号代表 calc 的方法体。

**3.10 Lambda省略格式**

**可推导即可省略**

Lambda强调的是“做什么”而不是“怎么做”，所以凡是可以根据上下文推导得知的信息，都可以省略。例如上例还可 以使用Lambda的省略写法：

**省略规则**

在Lambda标准格式的基础上，使用省略写法的规则为：

1. 小括号内参数的类型可以省略；
2. 如果小括号内**有且仅有一个参**，则小括号可以省略；
3. 如果大括号内**有且仅有一个语句**，则无论是否有返回值，都可以省略大括号、return关键字及语句分号。 备注：掌握这些省略规则后，请对应地回顾本章开头的多线程案例。

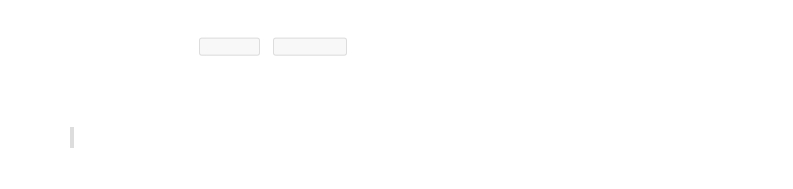
**3.11 练习：使用Lambda省略格式**

**题目**

仍然使用前文含有唯一 makeFood 抽象方法的厨子 Cook 接口，在下面的代码中，请使用Lambda的**省略格式**调用 invokeCook 方法，打印输出“吃饭啦！”字样：

**解答**

**3.12 Lambda的使用前提**



Lambda的语法非常简洁，完全没有面向对象复杂的束缚。但是使用时有几个问题需要特别注意：

1. 使用Lambda必须具有接口，且要求**接口中有且仅有一个抽象方法**。

无论是JDK内置的 Runnable 、 Comparator 接口还是自定义的接口，只有当接口中的抽象方法存在且唯一 时，才可以使用Lambda。

2. 使用Lambda必须具有**上下文推断**。

也就是方法的参数或局部变量类型必须为Lambda对应的接口类型，才能使用Lambda作为该接口的实例。 备注：有且仅有一个抽象方法的接口，称为“**函数式接口**”。