Java进阶1 第7天

**【学习目标】理解、了解、应用、记忆**

通过今天的学习，参训学员能够：（解释的时候说出二级目标的掌握程度）

1. **【应用】面向对象练习**
2. 【应用】能够独立分析类与类,类与接口的继承或实现关系
3. 【应用】能够根据案例定义出抽象类和接口,并测试
4. **【应用】API练习**
5. 【应用】能够独立写出删除一个字符串中指定的子串案例
6. **【应用】集合练习**
7. 【应用】 能够独立完成统计每个班级的平均分和总分的案例
8. **【应用】IO递归练习**
9. 【应用】 能够独立写出文件夹递归拷贝案例
10. **【应用】多线程概述**
11. 【理解】能够理解线程与进程的概念
12. 【理解】能够理解多线程提高运行效率原因
13. **【应用】多线程实现**
14. 【理解】能够阐述Java是否支持多线程以及如何在多个线程上执行
15. 【应用】能够独立写出线程的两种创建方式
16. 【理解】能够理解用匿名内部类创建线程的代码
17. 【应用】能够独立写出多线程卖票的案例
18. **【应用】线程池与线程生命周期**
19. 【理解】 能够阐述线程四个基本状态
20. 【应用】 能够独立完成利用Runnable接口向线程池提交任务的代码
21. 【应用】 能够独立完成利用Callable接口向线程池提交任务的代码

# 面向对象练习

## 需求:

描述笔记本类，主要实现笔记本使用USB设备功能，最终达到要求的打印结果。

笔记本类:

开机功能

关机功能

定义使用USB设备的功能，要求：既能使用鼠标也能使用键盘，使用 USB功能内部调用开启和关闭功能

USB接口:

开启功能

工作方法

关闭功能

硬件设备类:

工作方法

电脑屏幕类:

工作方法为显示画面

鼠标类:

工作方法为点击

属于硬件设备，但要符合USB接口

键盘类:

工作方法为按键

属于硬件设备，但要符合USB接口

测试类:

创建电脑对象，依次调用开机方法，使用USB设备，关机方法

打印效果如下:

开机

显示屏显示画面

连接鼠标的USB

鼠标点击

断开鼠标的USB

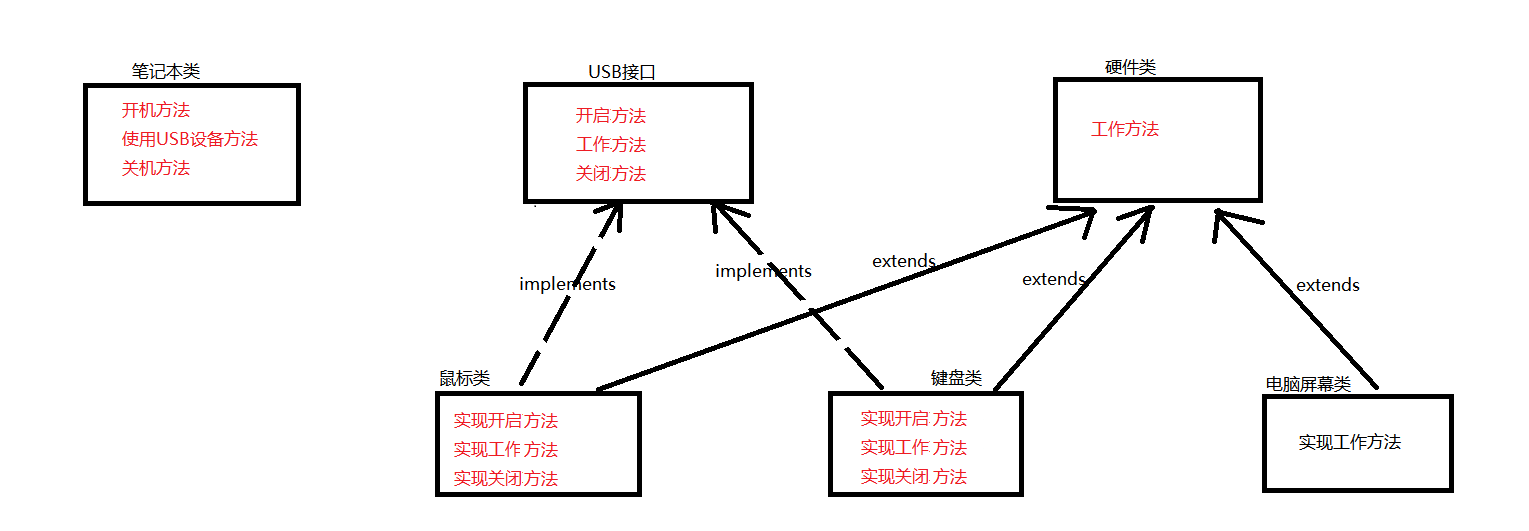
连接键盘的USB

键盘按键

断开键盘的USB

关机

## 需求分析:



## 需求实现

### 案例代码一:

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: USB  \* **@Description**: USB接口  \* **@date** 2017年11月24日 上午9:51:43  \* Company www.igeekhome.com  \*  \* USB接口中方法:  \* 开启功能参数:无 返回值:无  \* 工作方法参数:无 返回值:无  \* 关闭功能参数:无 返回值:无  \*/  **public** **interface** USB {    /\*\*  \* **@Title**: open  \* **@Description**: 开启功能  \*/  **public** **abstract** **void** open();    /\*\*  \* **@Title**: work  \* **@Description**: 工作功能  \*/  **public** **abstract** **void** work();    /\*\*  \* **@Title**: close  \* **@Description**: 关闭功能  \*/  **public** **abstract** **void** close();  } |

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: Hardware  \* **@Description**: 硬件设备类  \* **@date** 2017年11月24日 上午9:53:40  \* Company www.igeekhome.com  \*  \* 硬件设备类方法:  \* 工作方法参数:无  \* 返回值:无  \* 方法内容:由于硬件设备可以设计为抽象类,所以工作方法设计为抽象方法,无方法体  \*/  **public** **abstract** **class** Hardware {  /\*\*  \* **@Title**: work  \* **@Description**: 工作方法  \*/  **public** **abstract** **void** work();  } |

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: Mouse  \* **@Description**: 鼠标类  \* **@date** 2017年11月24日 上午9:57:23  \* Company www.igeekhome.com  \*  \* 鼠标是硬件类的子类,是USB接口的实现类  \*  \* 重写方法:  \* 开启功能参数:无 返回值:无 方法内容:连接鼠标  \* 工作方法参数:无 返回值:无 方法内容:打印鼠标点击  \* 关闭功能参数:无 返回值:无 方法内容:断开鼠标  \*/  **public** **class** Mouse **extends** Hardware **implements** USB {  /\*\*  \* **@Title**: open  \* **@Description**: 开启功能  \* **@see** com.igeek\_01.USB#open()  \*/  @Override  **public** **void** open() {  System.***out***.println("连接鼠标");  }  /\*\*  \* **@Title**: close  \* **@Description**: 关闭功能  \* **@see** com.igeek\_01.USB#close()  \*/  @Override  **public** **void** close() {  System.***out***.println("断开鼠标");  }  /\*\*  \* **@Title**: work  \* **@Description**: 工作方法  \* **@see** com.igeek\_01.Hardware#work()  \*/  @Override  **public** **void** work() {  System.***out***.println("鼠标点击");  }  } |

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: Keyboard  \* **@Description**: 键盘类  \* **@date** 2017年11月24日 上午10:00:30  \* Company www.igeekhome.com  \*  \* 键盘是硬件类的子类,是USB接口的实现类  \*  \* 重写方法:  \* 开启功能参数:无 返回值:无 方法内容:连接键盘  \* 工作方法参数:无 返回值:无 方法内容:打印键盘按键  \* 关闭功能参数:无 返回值:无 方法内容:断开键盘  \*/  **public** **class** Keyboard **extends** Hardware **implements** USB {  /\*\*  \* **@Title**: open  \* **@Description**: 开启功能  \* **@see** com.igeek\_01.USB#open()  \*/  @Override  **public** **void** open() {  System.***out***.println("连接键盘");  }  /\*\*  \* **@Title**: close  \* **@Description**: 关闭功能  \* **@see** com.igeek\_01.USB#close()  \*/  @Override  **public** **void** close() {  System.***out***.println("断开键盘");  }  /\*\*  \* **@Title**: work  \* **@Description**: 工作方法  \* **@see** com.igeek\_01.Hardware#work()  \*/  @Override  **public** **void** work() {  System.***out***.println("键盘按键");  }  } |

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: Screen  \* **@Description**: 电脑屏幕子类  \* **@date** 2017年11月24日 上午10:04:42  \* Company www.igeekhome.com  \*  \* 电脑屏幕是硬件类的子类  \*  \* 重写方法:  \* 工作方法参数:无 返回值:无 方法内容:打印显示屏显示画面  \*/  **public** **class** Screen **extends** Hardware {  /\*\*  \* **@Title**: work  \* **@Description**: 工作方法  \* **@see** com.igeek\_01.Hardware#work()  \*/  @Override  **public** **void** work() {  System.***out***.println("显示屏显示画面 ");  }  } |

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: Computer  \* **@Description**: 笔记本类  \* **@date** 2017年11月24日 上午10:07:19  \* Company www.igeekhome.com  \*  \* 笔记本类中方法:  \* 开机功能参数:无 返回值:无 功能内容:打印开机,创建电脑屏幕类对象,调用显示画面方法  \* 关机功能参数:无 返回值:无 功能内容:打印关机  \* 使用USB设备的功能  \* 参数:USB接口参数(形式参数)  \* 返回值:无  \* 功能内容:调用USB设备的三个功能:开启功能,工作方法,关闭功能  \*/  **public** **class** Computer {    /\*\*  \* **@Title**: begin  \* **@Description**: 开机方法  \*/  **public** **void** begin() {  System.***out***.println("开机");  //创建电脑屏幕类对象  Screen sc = **new** Screen();  //调用显示画面方法  sc.work();  }    /\*\*  \* **@Title**: over  \* **@Description**: 关机方法  \*/  **public** **void** over() {  System.***out***.println("关机");  }    /\*\*  \* **@Title**: useUSB  \* **@Description**: 使用USB功能  \* **@param** usb  \*/  **public** **void** useUSB(USB usb) {  usb.open();  usb.work();  usb.close();  }  } |

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: ComputerTest  \* **@Description**: 面向对象练习题,计算机综合测试类  \* **@date** 2017年11月24日 上午10:14:38  \* Company www.igeekhome.com  \*  面向对象题目:    分析以下需求，并用代码实现:  描述笔记本类，主要实现笔记本使用USB设备功能，最终达到要求的打印结果。  笔记本类:  开机功能  关机功能  定义使用USB设备的功能，要求：既能使用鼠标也能使用键盘，使用USB功能内部调用开启和关闭功能    USB接口:  开启功能  工作方法  关闭功能    硬件设备类:  工作方法    电脑屏幕类:  工作方法为显示画面    鼠标类:  工作方法为点击  属于硬件设备，但要符合USB接口    键盘类:  工作方法为按键  属于硬件设备，但要符合USB接口    测试类:  创建电脑对象，依次调用开机方法，使用USB设备，关机方法    打印效果如下:  开机  显示屏显示画面  连接鼠标的USB  鼠标点击  断开鼠标的USB  连接键盘的USB  键盘按键  断开键盘的USB  关机  ========================================  \* 分析过程一:类间关系  \*  \* 笔记本类使用其他的类  \* USB接口  \* 鼠标  \* 键盘  \* 硬件设备类  \* 电脑屏幕类  \* 鼠标  \* 键盘  \* 测试类创建笔记本类对象调用方法  \*  \* 分析过程二:类中功能(方法)  \* 笔记本类中方法:  \* 开机功能 参数:无 返回值:无 功能内容:打印开机,创建电脑屏幕类对象,调用显示画面方法  \* 关机功能 参数:无 返回值:无 功能内容:打印关机  \* 使用USB设备的功能  \* 参数:USB接口参数(形式参数)  \* 返回值:无  \* 功能内容:调用USB设备的三个功能:开启功能,工作方法,关闭功能  \*  \* USB接口中方法:  \* 开启功能 参数:无 返回值:无  \* 工作方法 参数:无 返回值:无  \* 关闭功能 参数:无 返回值:无  \*  \* 硬件设备类方法:  \* 工作方法 参数:无 返回值:无 方法内容:由于硬件设备可以设计为抽象类,所以工作方法设计为抽象方法,无方法体  \*  \* 电脑屏幕类:  \* 工作方法 参数:无 返回值:无 方法内容:打印显示屏显示画面  \*  \* 鼠标类:  \* 开启功能 参数:无 返回值:无 方法内容:连接鼠标  \* 工作方法 参数:无 返回值:无 方法内容:打印鼠标点击  \* 关闭功能 参数:无 返回值:无 方法内容:断开鼠标  \*  \* 键盘类:  \* 开启功能 参数:无 返回值:无 方法内容:连接键盘  \* 工作方法 参数:无 返回值:无 方法内容:打印键盘按键  \* 关闭功能 参数:无 返回值:无 方法内容:断开键盘  \*  \* 分析过程三:完成测试  \* 测试类创建笔记本对象,调用笔记本对象的方法:  \* 开机方法  \* 使用USB设备方法:  \* 创建鼠标对象  \* 调用使用USB设备方法  \* 使用USB设备方法:  \* 创建键盘对象  \* 调用使用USB设备方法  \* 关机方法  \*/  **public** **class** ComputerTest {  **public** **static** **void** main(String[] args) {  //测试类创建笔记本对象  Computer computer = **new** Computer();  //开机方法  computer.begin();    //创建鼠标对象,调用使用USB设备方法  Mouse mouse = **new** Mouse();  computer.useUSB(mouse);    //创建键盘对象,调用使用USB设备方法  Keyboard keyboard = **new** Keyboard();  computer.useUSB(keyboard);    //关机方法  computer.over();  }  } |

# API练习

## 需求

键盘录入一个字符串，要求删除该字符串中的所有java字符串（最终的字符串中不能包含java），要求打印删除后的结果以及删除了几个java字符串

比如键盘录入："java woaijava,i like jajavava,i enjoy java"

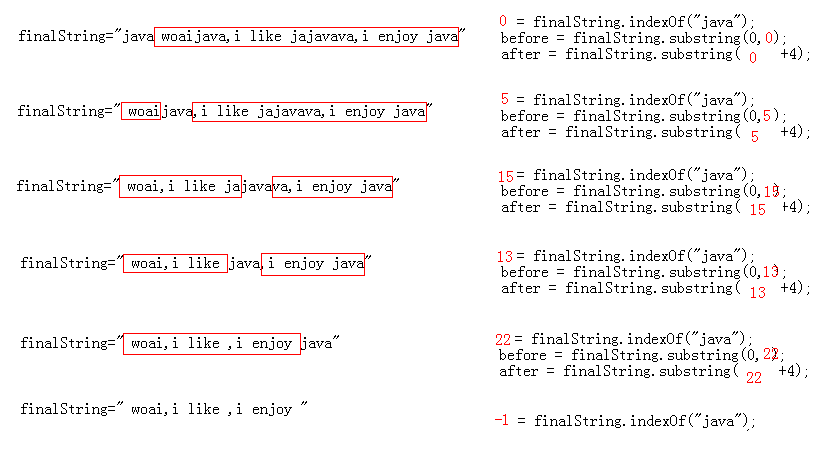
程序输出结果：

原字符串："java woaijava,i like jajavava,i enjoy java"中

总共包含：5个java删除java后的字符串为：" woai,i like ,i enjoy "

## 需求分析

截取过程：



## 需求实现

### 案例代码二:

|  |
| --- |
| **package** com.igeek\_01;  **import** java.util.Scanner;  /\*\*  \* **@ClassName**: StringTest  \* **@Description**: API练习题，字符串操作  \* **@date** 2018年2月2日 下午1:52:18  \* Company www.igeekhome.com  \*  \* 需求：  \* 键盘录入一个字符串，要求删除该字符串中的所有java字符串（最终的字符串中不能包含java），要求打印删除后的结果以及删除了几个java字符串  \* 比如键盘录入："java woaijava,i like jajavava,i enjoy java"  \* 程序输出结果：  \* 原字符串："java woaijava,i like jajavava,i enjoy java"中  \* 总共包含：5个java，删除java后的字符串为：" woai,i like ,i enjoy "  \*  \*====================================================================  \* 分析：  \* 1、使用键盘录入Scanner接收一个初识被判断的字符串，用变量initString接收  \* 2、定义变量，用于记录每次变化的那个字符串，即最终的字符串finalString  \* 3、定义变量，记录java的个数  \* 4、  \* 定义变量，记录java是否存在于字符串中，用索引表示(indexOf方法)  \* while循环删除：  \* 条件：  \* 为只要字符串中没有java字符串即不再循环  \* 如果索引不为-1，就说明要删除字符串中的java；  \* 如果索引为-1，就说明已经不再包含java了，循环终止；  \*  \* 循环删除过程：  \* 由于进入了循环，所以判断字符串中有一个java，个数计数器+1  \* 将java前的字符串截取记录(substring方法(0,索引i),从0到i)  \* 将java后的字符串截取记录(substring方法(索引i+4),所i+4到最后)  \* 将java前后的字符串再拼接成新的字符串  \* 再判断新字符串中是否存在java，使用indexOf方法重新获取索引  \*  \* 5、打印结果：  \* 打印原字符串  \* 打印java个数  \* 打印不包含java的新字符串  \*/  **public** **class** StringTest {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  //1、使用键盘录入Scanner接收一个初识被判断的字符串，用变量initString接收  Scanner scanner = **new** Scanner(System.***in***);  //初始化数据  String initString = scanner.nextLine();    //2、定义变量，用于记录每次变化的那个字符串，即最终的字符串finalString  String finalString = initString;    //3、定义变量，记录java的个数  **int** javaNumber=0;    //4、循环过程  //定义变量，记录java是否存在于字符串中，用索引表示(indexOf方法)  **int** index = finalString.indexOf("java");    //条件：  //为只要字符串中没有java字符串即不再循环  //如果索引不为-1，就说明要删除字符串中的java；  //如果索引为-1，就说明已经不再包含java了，循环终止；  **while**(index!=-1){  System.***out***.println(index);    //加入语句，让循环每秒执行一次  Thread.*sleep*(1000);    //反复删除  //由于进入了循环，所以判断字符串中有一个java，个数计数器+1  javaNumber++;    //将java前的字符串截取记录(substring方法(0,索引i),从0到i)  String before = finalString.substring(0, index);    //将java后的字符串截取记录(substring方法(索引i+4),所i+4到最后)  String after = finalString.substring(index+4);    //将java前后的字符串再拼接成新的字符串  finalString = before+after;    //再判断新字符串中是否存在java，使用indexOf方法重新获取索引  index = finalString.indexOf("java");    System.***out***.println(finalString);  }    //5、打印结果  //格式  //原字符串："java woaijava,i like jajavava,i enjoy java"中  //总共包含：5个java，删除java后的字符串为：" woai,i like ,i enjoy "  System.***out***.println("============================================");    System.***out***.println("原字符串："+initString+"中");  System.***out***.println("总共包含："+javaNumber+"个java，删除java后的字符串为："+finalString);  }  } |

# 集合练习

## 需求:

分析以下需求，并用代码实现：

(1)定义一个学生类Student，属性：姓名(String name)、班级班号(String class\_number)、分数(double score)

(2)初始化数据将若干Student对象存入List集合

(3)以班级为单位,使用Map存储所有该班学生

(4)统计每个班级的总分和平均分

可选解法提示:

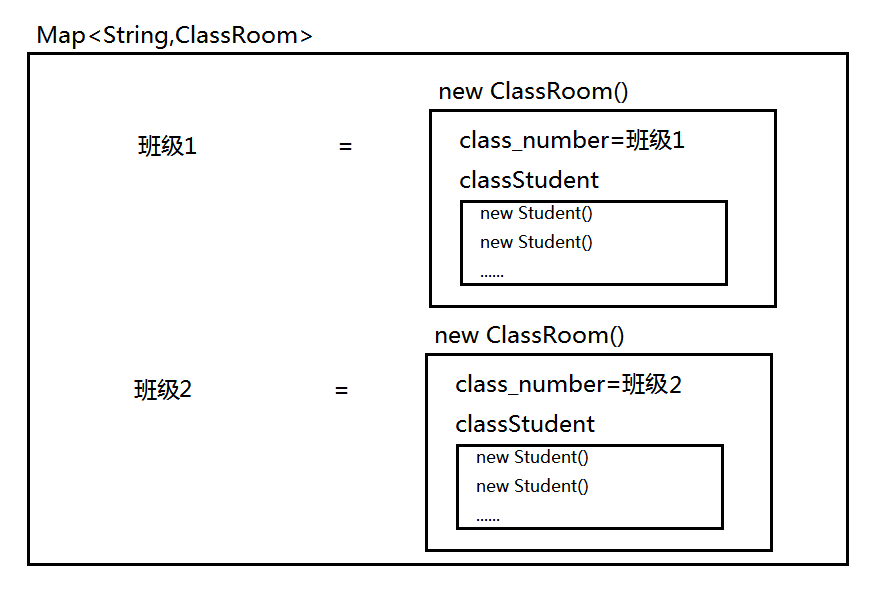
a.采用面向对象的思想(组合)

b.不推荐使用Map<String,List<Student>>

c.推荐使用Map<String，ClassRoom>

## 需求分析:

保存的结构图:



## 需求实现:

### 案例代码三:

|  |
| --- |
| **package** com.igeek\_01;  **import** java.util.ArrayList;  **import** java.util.Collections;  **import** java.util.HashMap;  **import** java.util.Set;  /\*\*  \* **@ClassName**: ListTest  \* **@Description**: 集合练习题  \* **@date** 2018年2月2日 下午2:53:15  \* Company www.igeekhome.com  \*  \* 分析以下需求，并用代码实现：  \* (1)定义一个学生类Student，属性：姓名(String name)、班级班号(String class\_number)、分数(double score)  \* (2)初始化数据将若干Student对象存入List集合  \* (3)以班级为单位,使用Map存储所有该班学生  \* map.put("班级1",班级1的学生);  \* map.put("班级2",班级2的学生);  \* ......  \* (4)统计每个班级的总分和平均分  \*  \* 可选解法提示:  \* a.采用面向对象的思想(组合)  \* b.不推荐使用Map<String,List<Student>>  \* c.推荐使用Map<String，ClassRoom>  \*  \*=========================================================  \* 分析：  \* 1、定义Student类  \* 2、创建多个Student对象，放到集合中，数据一共有3个班级，班级1、班级2、班级3，作为初始数据  \* 3、定义班级类：  \* 班号  \* 该班所有的学生  \*  \* 将该班级所有学生以Map<String,ClassRoom>的方式，存储到Map集合中  \* 定义变量，记录Map集合，用于存储所有班级(这里简化需求，只有三个班级，班级1、班级2、班级3)  \* 定义三个ArrayList，记录三个班级对象的所有学生信息  \* 迭代所有学生初始化数据集合，依次获取到每一个同学  \* 判断每个学生是班级1、班级2还是班级3，将该学生放到对应的ArrayList集合中  \* 创建三个ClassRoom对象  \* 再将三个班级加入到对应的Map集合中  \* 4、  \* 遍历Map集合，分别拿到每个班级的班级对象  \* 获取班级对象中所有学生的列表  \* 定义变量，记录总分  \* 定义变量，记录平均分  \* 循环遍历列表，累加总分  \* 计算平均分  \* 打印数据  \*/  **public** **class** ListTest {  **public** **static** **void** main(String[] args) {  //初始化数据  Student s = **new** Student("Jack", "班级1", 89);  Student s2 = **new** Student("Jack", "班级2", 29);  Student s3 = **new** Student("Jack", "班级1", 39);  Student s4 = **new** Student("Jack", "班级3", 49);  Student s5 = **new** Student("Jack", "班级2", 59);  Student s6 = **new** Student("Jack", "班级1", 50);  Student s7 = **new** Student("Jack", "班级1", 88);    //将若干Student对象存入List集合  ArrayList<Student> allStudents = **new** ArrayList<Student>();  Collections.*addAll*(allStudents, s,s2,s3,s4,s5,s6,s7);    //将该班级所有学生以Map<String,ClassRoom>的方式，存储到Map集合中  HashMap<String, ClassRoom> allClassMap = **new** HashMap<String, ClassRoom>();    //定义三个ArrayList，记录三个班级对象的所有学生信息  ArrayList<Student> class1Students = **new** ArrayList<Student>();  ArrayList<Student> class2Students = **new** ArrayList<Student>();  ArrayList<Student> class3Students = **new** ArrayList<Student>();    //迭代所有学生初始化数据集合，依次获取到每一个同学  **for** (Student thisStudent : allStudents) {  //判断每个学生是班级1、班级2还是班级3，将该学生放到对应的ArrayList集合中  **if**("班级1".equals(thisStudent.getClassNumber())){  class1Students.add(thisStudent);  }**else** **if**("班级2".equals(thisStudent.getClassNumber())){  class2Students.add(thisStudent);  }**else** **if**("班级3".equals(thisStudent.getClassNumber())){  class3Students.add(thisStudent);  }  }    //创建三个ClassRoom对象  ClassRoom class1 = **new** ClassRoom("班级1", class1Students);  ClassRoom class2 = **new** ClassRoom("班级2", class2Students);  ClassRoom class3 = **new** ClassRoom("班级3", class3Students);    //再将三个班级加入到对应的Map集合中  allClassMap.put("班级1", class1);  allClassMap.put("班级2", class2);  allClassMap.put("班级3", class3);      //遍历Map集合，分别拿到每个班级的班级对象  Set<String> classNumbers = allClassMap.keySet();    //迭代班级班号集合  **for** (String classNumber : classNumbers) {  //根据班级班号获取班级对象  ClassRoom classRoom = allClassMap.get(classNumber);  //调用方法，求总分与平均分  *getClassInfo*(classRoom);  }  }    /\*\*  \* **@Title**: getClassInfo  \* **@Description**: 接收一个班级，打印出该班级的总分与平均分  \* **@param** classRoom  \*/  **public** **static** **void** getClassInfo(ClassRoom classRoom){  //获取班级对象中所有学生的列表  ArrayList<Student> classStudent = classRoom.getStudentsList();  //定义变量，记录总分  **double** sum=0;  //定义变量，记录平均分  **double** avg=0;  //循环遍历列表，累加总分  **for** (Student student : classStudent) {  sum+=student.getScore();  }  //计算平均分  avg=sum/classStudent.size();  //打印数据  System.***out***.println(classRoom.getClassNumber()+"的总分为："+sum+",平均分为："+avg);  }  } |

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: Student  \* **@Description**: 学生类  \* **@date** 2018年2月2日 下午3:33:23  \* Company www.igeekhome.com  \*  \* 定义一个学生类Student，属性：姓名(String name)、班级班号(String class\_number)、分数(double score)  \*/  **public** **class** Student {  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;  /\*\*  \* **@Fields** classNumber : 班级班号  \*/  **private** String classNumber;  /\*\*  \* **@Fields** score : 分数  \*/  **private** **double** score;  /\*\*  \* **@Title**: Student  \*/  **public** Student() {  **super**();  }  /\*\*  \* **@Title**: Student  \* **@param** name  \* **@param** classNumber  \* **@param** score  \*/  **public** Student(String name, String classNumber, **double** score) {  **super**();  **this**.name = name;  **this**.classNumber = classNumber;  **this**.score = score;  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  /\*\*  \* **@return** the classNumber  \*/  **public** String getClassNumber() {  **return** classNumber;  }  /\*\*  \* **@param** classNumber the classNumber to set  \*/  **public** **void** setClassNumber(String classNumber) {  **this**.classNumber = classNumber;  }  /\*\*  \* **@return** the score  \*/  **public** **double** getScore() {  **return** score;  }  /\*\*  \* **@param** score the score to set  \*/  **public** **void** setScore(**double** score) {  **this**.score = score;  }  } |

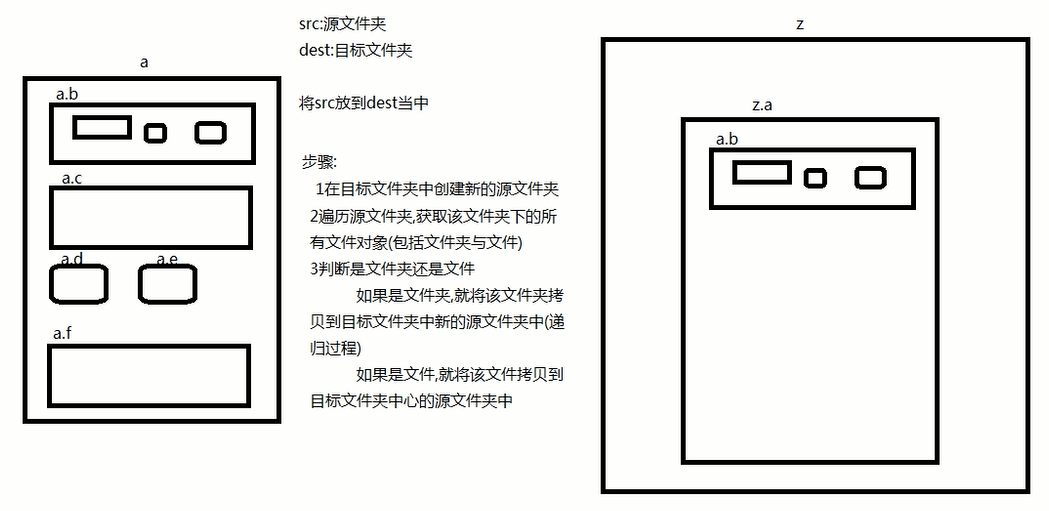
|  |
| --- |
| **package** com.igeek\_01;  **import** java.util.ArrayList;  /\*\*  \* **@ClassName**: ClassRoom  \* **@Description**: 班级类  \* **@date** 2018年2月2日 下午3:39:15  \* Company www.igeekhome.com  \*  \* 班级类：班级班号、学生列表  \*/  **public** **class** ClassRoom {  /\*\*  \* **@Fields** classNumber : 班级班号  \*/  **private** String classNumber;  /\*\*  \* **@Fields** students : 学生列表  \*/  **private** ArrayList<Student> studentsList;  /\*\*  \* **@Title**: ClassRoom  \*/  **public** ClassRoom() {  **super**();  }  /\*\*  \* **@Title**: ClassRoom  \* **@param** classNumber  \* **@param** studentsList  \*/  **public** ClassRoom(String classNumber, ArrayList<Student> studentsList) {  **super**();  **this**.classNumber = classNumber;  **this**.studentsList = studentsList;  }  /\*\*  \* **@return** the classNumber  \*/  **public** String getClassNumber() {  **return** classNumber;  }  /\*\*  \* **@param** classNumber the classNumber to set  \*/  **public** **void** setClassNumber(String classNumber) {  **this**.classNumber = classNumber;  }  /\*\*  \* **@return** the studentsList  \*/  **public** ArrayList<Student> getStudentsList() {  **return** studentsList;  }  /\*\*  \* **@param** studentsList the studentsList to set  \*/  **public** **void** setStudentsList(ArrayList<Student> studentsList) {  **this**.studentsList = studentsList;  }    } |

# IO递归练习

## 需求:

将一个文件夹中的内容(包含子文件夹中的所有内容)复制到指定目的地

## 需求分析:



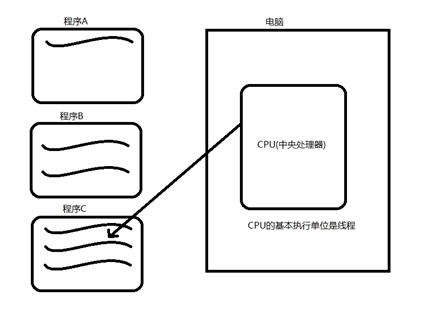
## 需求实现:

### 案例代码四:

|  |
| --- |
| **package** com.igeek\_01;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  /\*\*  \* **@ClassName**: IOTest  \* **@Description**: IO练习题  \* **@date** 2018年2月2日 下午4:09:51  \* Company www.igeekhome.com  \*  \* 需求:  \* 完成多级文件夹复制功能(不允许使用CommonsIO)  \*  \* 分析见图:多级文件夹复制.png  \*/  **public** **class** IOTest {  **public** **static** **void** main(String[] args) {    //给出要复制的源文件夹与目标文件夹  File src = **new** File("a");  File dest = **new** File("z");    *copyDir2Dir*(src, dest);  }  /\*\*  \* **@Title**: copyDir2Dir  \* **@Description**: 将源文件夹src复制到目标文件夹dest  \* **@param** src  \* **@param** dest  \*/  **public** **static** **void** copyDir2Dir(File src,File dest) { |

# 多线程概述

## 线程与进程概念



A：进程

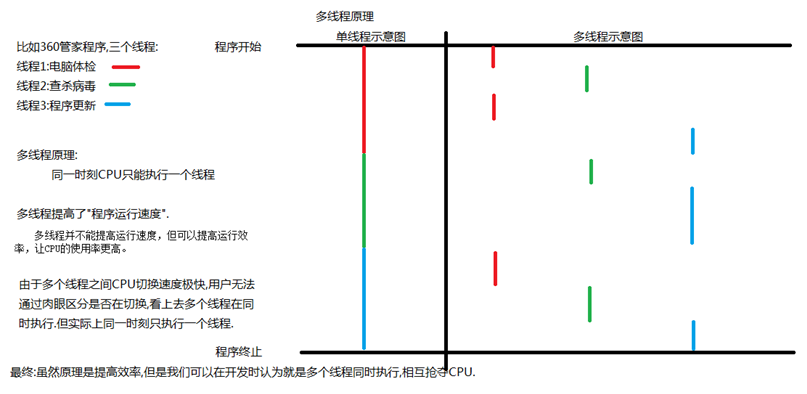
正在运行的程序。确切的来说，当一个程序进入内存运行，即是一个或多个进程在运行，具有一定独立功能。

B：线程

线程是进程中的一个执行单元，负责当前进程中程序的执行，一个进程中至少有一个线程。一个进程中是可以有多个线程的，这个应用程序也可以称之为多线程程序。

简而言之：一个程序运行后至少有一个进程，一个进程中可以包含多个线程。

## 多线程提高运行效率



多线程并不能提高运行速度，但可以提高运行效率，让CPU的使用率更高。

并且，如果多线程有安全问题的处理时，运算速度反而更低。

# 多线程实现

## Java中的多线程

A：java支持多线程。

B：当java程序执行main方法的时候，就是在执行一个名字叫做main的线程。

可以在main方法执行时，开启多个线程A、B、C。

多个线程main，A，B，C同时执行，相互抢夺CPU。

C：Thread类是java.lang包下的一个常用类，每一个Thread类的对象，就代表一个处于某种状态的线程。

#### 案例代码五:

|  |
| --- |
| **package** com.igeek\_01;  **import** java.util.Date;  /\*\*  \* **@ClassName**: ThreadDemo  \* **@Description**: Java中的多线程  \* **@date** 2018年2月4日 上午10:38:55  \* Company www.igeekhome.com  \*  \* A：java支持多线程。  \*  \* B：当java程序执行main方法的时候，就是在执行一个名字叫做main的线程。  \* 可以在main方法执行时，开启多个线程A、B、C。  \* 多个线程main，A，B，C同时执行，相互抢夺CPU。  \*  \* C：Thread类是java.lang包下的一个常用类，每一个Thread类的对象，就代表一个处于某种状态的线程。  \*/  **public** **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  Date today = **new** Date();  //......  }  } |

## 创建线程

该如何创建线程呢？通过API中搜索，查到Thread类。通过阅读Thread类中的描述。Thread是程序中的执行线程。Java 虚拟机允许应用程序并发地运行多个执行线程。

我们需要用到Thread类的一些方法：

public final String getName() 获取线程名称

public final void setName(String name) 指定线程名称

public static Thread currentThread() 获取当前线程对象

### 继承Thread类创建线程

将类声明为Thread的子类。该子类重写Thread类的run方法。创建对象，开启线程。

开启线程的步骤：

1、指定线程执行目标：定义一个Thread类的子类，重写run方法，将相关逻辑实现

public void run() 线程要执行的业务逻辑方法，相当于该线程的"main方法"

2、创建自定义的线程子类对象

3、开启线程动作

public void start() 使该线程开始执行

#### 案例代码六:

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: ThreadDemo  \* **@Description**: Java中的多线程  \* **@date** 2018年2月4日 上午10:42:57  \* Company www.igeekhome.com  \*  \* A：java支持多线程。  \*  \* B：当java程序执行main方法的时候，就是在执行一个名字叫做main的线程。  \* 可以在main方法执行时，开启多个线程A、B、C。  \* 多个线程main，A，B，C同时执行，相互抢夺CPU。  \*  \* C：Thread类是java.lang包下的一个常用类，每一个Thread类的对象，就代表一个处于某种状态的线程。  \*  \* 第一种方式创建线程：  \* 开启线程的步骤：  \* 1、指定线程执行目标：定义一个Thread类的子类，重写run方法，将相关逻辑实现  \* public void run() 线程要执行的业务逻辑方法，相当于该线程的"main方法"  \* 2、创建自定义的线程子类对象  \* 3、开启线程动作  \* public void start() 使该线程开始执行  \*  \* Thread类的一些方法：  \* public final String getName() 获取线程名称  \* public final void setName(String name) 指定线程名称  \* public static Thread currentThread() 获取当前线程对象  \*/  **public** **class** ThreadDemo {  **public** **static** **void** main(String[] args) {    //开启线程  //创建线程对象  MyThread thread = **new** MyThread();  MyThread thread2 = **new** MyThread();    //开启线程动作  thread.start();  thread2.start();    //返回main线程  Thread mainThread = Thread.*currentThread*();    **for**(**int** i=0;i<50;i++){  //System.out.println("main:"+i);  System.***out***.println(mainThread.getName()+":"+i);  }    }  } |

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: MyThread  \* **@Description**: 自定义线程类  \* **@date** 2018年2月4日 上午11:34:45  \* Company www.igeekhome.com  \*  \* 定义了一个Thread类的子类线程  \*/  **public** **class** MyThread **extends** Thread{  /\*\*  \* **@Title**: run  \* **@Description**: 重写run方法，将相关逻辑实现  \* **@see** java.lang.Thread#run()  \*/  @Override  **public** **void** run() {  **for**(**int** i=0;i<50;i++){  //System.out.println("MyThread:"+i);  System.***out***.println(**this**.getName()+":"+i);  }  }    } |

### 实现Runnable接口创建线程

创建线程的另一种方法是声明实现 Runnable 接口的类，该类然后实现 run 方法。然后创建Runnable的实现类对象，传入到某个线程的构造方法中，开启线程。

为何要实现Runnable接口，Runnable是啥玩意呢？继续API搜索。

查看Runnable接口说明文档：Runnable接口用来指定每个线程要执行的任务。包含了一个 run 的无参数抽象方法，需要由接口实现类重写该方法。

开启线程的步骤：

1、指定线程执行目标：定义Runnable线程执行目标实现类，重写run方法，指定目标逻辑

2、通过指定线程执行目标的构造方法创建线程对象

public Thread(Runnable target)

a)创建线程执行目标对象

b)通过线程执行目标创建线程对象

3、开启线程动作 (start开启线程)

#### 案例代码七:

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: ThreadDemo2  \* **@Description**: Java中的多线程  \* **@date** 2018年2月4日 上午10:42:57  \* Company www.igeekhome.com  \*  \* A：java支持多线程。  \*  \* B：当java程序执行main方法的时候，就是在执行一个名字叫做main的线程。  \* 可以在main方法执行时，开启多个线程A、B、C。  \* 多个线程main，A，B，C同时执行，相互抢夺CPU。  \*  \* C：Thread类是java.lang包下的一个常用类，每一个Thread类的对象，就代表一个处于某种状态的线程。  \*  \* 第一种方式创建线程：  \* 开启线程的步骤：  \* 1、指定线程执行目标：定义一个Thread类的子类，重写run方法，将相关逻辑实现  \* public void run() 线程要执行的业务逻辑方法，相当于该线程的"main方法"  \* 2、创建自定义的线程子类对象  \* 3、开启线程动作  \* public void start() 使该线程开始执行  \*  \* Thread类的一些方法：  \* public final String getName() 获取线程名称  \* public final void setName(String name) 指定线程名称  \* public static Thread currentThread() 获取当前线程对象  \*  \* 第二种方式创建线程：  \* 开启线程的步骤：  \* 1、指定线程执行目标：定义Runnable线程执行目标实现类，重写run方法，指定目标逻辑  \* 2、通过指定线程执行目标的构造方法创建线程对象  \* public Thread(Runnable target)  \* a)创建线程执行目标对象  \* b)通过线程执行目标创建线程对象  \* 3、开启线程动作 (start开启线程)  \*/  **public** **class** ThreadDemo2 {  **public** **static** **void** main(String[] args) {    //开启线程  //创建线程执行目标  MyRunnable mr = **new** MyRunnable();  //通过指定线程执行目标的构造方法创建线程对象  Thread thread = **new** Thread(mr);  thread.setName("Jack");  Thread thread2 = **new** Thread(mr);  thread2.setName("Rose");    //开启线程动作  thread.start();  thread2.start();    //返回main线程  Thread mainThread = Thread.*currentThread*();    **for**(**int** i=0;i<50;i++){  //System.out.println("main:"+i);  System.***out***.println(mainThread.getName()+":"+i);  }    }  } |

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: MyRunnable  \* **@Description**: Runnable的实现类  \* **@date** 2018年2月4日 下午12:36:55  \* Company www.igeekhome.com  \*  \* 定义了线程执行目标实现类  \*/  **public** **class** MyRunnable **implements** Runnable{  /\*\*  \* **@Title**: run  \* **@Description**: 指定线程执行目标逻辑  \* **@see** java.lang.Runnable#run()  \*/  @Override  **public** **void** run() {    //返回当前线程  Thread thisThread = Thread.*currentThread*();    **for**(**int** i=0;i<50;i++){  //System.out.println("线程名称:"+i);  System.***out***.println(thisThread.getName()+":"+i);  }  }  } |

### 两种实现方式对比

继承Thread类，线程对象和线程任务耦合在一起。一旦创建Thread类的子类对象，既是线程对象，有又有线程任务。

实现Runnable接口，将线程任务单独分离出来封装成对象，类型就是Runnable接口类型。

Runnable接口对线程对象和线程任务进行解耦。

A：第二种方式实现Runnable接口避免了单继承的局限性，所以较为常用。

B：实现Runnable接口的方式，更加的符合面向对象，线程分为两部分，一部分线程对象，一部分线程任务。

### 利用匿名内部类创建线程

匿名内部类格式：

new 父类/父接口(){

//重写父类或父接口的方法

};

//不但定义了该父类/父接口的子类而且创建该子类的对象

#### 案例代码八:

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: AnonymousThreadDemo  \* **@Description**: 利用匿名内部类创建线程  \* **@date** 2018年2月4日 下午1:13:08  \* Company www.igeekhome.com  \*  \* 使用匿名内部类开启线程  \*  \*/  **public** **class** AnonymousThreadDemo {  **public** **static** **void** main(String[] args) {  //方式一：  //使用匿名内部类创建线程的子类对象  Thread thread = **new** Thread(){  /\*\*  \* **@Title**: run  \* **@Description**: **TODO**(这里用一句话描述这个方法的作用)  \* **@see** java.lang.Thread#run()  \*/  @Override  **public** **void** run() {  System.***out***.println("我的线程执行了");  }    };  //开启线程  thread.start();    //使用匿名内部类创建线程的子类匿名对象  **new** Thread(){  /\*\*  \* **@Title**: run  \* **@Description**: **TODO**(这里用一句话描述这个方法的作用)  \* **@see** java.lang.Thread#run()  \*/  @Override  **public** **void** run() {  System.***out***.println("我的线程执行了2");  }    }.start();    //方式二：  //使用匿名内部类的方式，创建线程执行目标类的对象  Runnable runnable = **new** Runnable() {    @Override  **public** **void** run() {  System.***out***.println("我的线程执行目标，执行了");  }  };    //通过目标创建线程对象  Thread thread2 = **new** Thread(runnable);  //开启线程  thread2.start();    //使用匿名内部类的方式，创建线程执行目标类的匿名对象  //通过目标创建线程对象  Thread thread3 = **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  System.***out***.println("我的线程执行目标，执行了2");    }  });  //开启线程  thread3.start();    //使用匿名内部类的方式，创建线程执行目标类的匿名对象  //通过目标创建线程对象  //开启线程  **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  System.***out***.println("我的线程执行目标，执行了3");  }  }).start();  }  } |

**package** com.igeek01.thread;

/\*

\* 使用匿名内部类开启线程

\*/

**publicclass** Demo02anonymous\_thread {

**publicstaticvoid** main(String[] args) {

//方式一:

//使用匿名内部类创建线程的子类对象

Thread thread = **new**Thread() {

@Override

**publicvoid** run() {

System.***out***.println("我的线程执行了");

}

};

thread.start();

//使用匿名内部类创建线程的子类匿名对象

**new** Thread() {

@Override

**publicvoid** run() {

System.***out***.println("我的线程执行了2");

}

}.start();

//方式二:

//使用匿名内部类的方式,创建线程执行目标类对象

//创建线程执行目标类对象

Runnable runnable = **new**Runnable() {

@Override

**publicvoid**run() {

System.***out***.println("我的线程执行目标,执行了");

}

};

//通过目标创建线程对象

Thread thread2 = **new**Thread(runnable);

//开启线程

thread2.start();

//使用匿名内部类的方式,创建线程执行目标类匿名对象

//创建线程执行目标类对象

//通过目标创建线程对象

Thread thread3 = **new**Thread(**new** Runnable() {

@Override

**publicvoid** run() {

System.***out***.println("我的线程执行目标,执行了2");

}

});

//开启线程

thread3.start();

//使用匿名内部类的方式,创建线程执行目标类匿名对象,并且创建的是线程的匿名对象

//创建线程执行目标类对象

//通过目标创建线程对象

**new** Thread(**new** Runnable() {

@Override

**publicvoid** run() {

System.***out***.println("我的线程执行目标,执行了3");

}

}).start();

}

}

## 多线程卖票案例

火车站有多个售票窗口，这些售票窗口一块卖100张票，即100张票被多个窗口共享。

用线程模拟售票窗口；用输出语句模拟卖出的票。

### 案例代码九:

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: TicketDemo  \* **@Description**: 多线程模拟火车站卖票  \* **@date** 2018年2月4日 下午1:59:57  \* Company www.igeekhome.com  \*  \* 使用第二种创建并开启线程的方式，方便数据共享  \*  \* 1、定义卖票的线程执行目标  \* 在成员变量位置将票定义为数字100，卖一张票，该数字减1，一直到小于0为止。  \* 重写run方法，完成卖票逻辑  \* while(true){  \* if(有票){  \* //卖票  \* 打印，xxx线程正在卖第几张票  \* 每卖完一张票，要将票数-1  \* }else{  \* //没有票，跳出循环，结束程序  \* }  \* }  \* 2、创建卖票的线程执行目标对象  \* 3、使用该卖票线程执行目标对象创建多个线程  \* 4、开启多个线程  \*/  **public** **class** TicketDemo {  **public** **static** **void** main(String[] args) {  //创建卖票的线程执行目标对象  Ticket ticket = **new** Ticket();    //使用该卖票线程执行目标对象创建多个线程  Thread thread = **new** Thread(ticket,"Jack");  Thread thread2 = **new** Thread(ticket,"Rose");  Thread thread3 = **new** Thread(ticket,"Trump");    //开启多个线程  thread.start();  thread2.start();  thread3.start();  }  } |

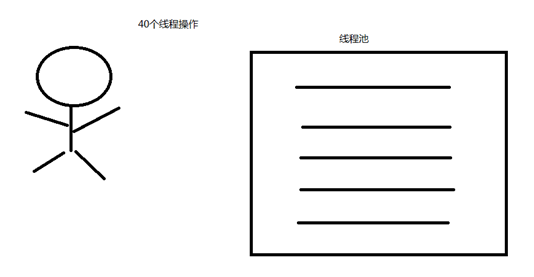
|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Ticket  \* **@Description**: 卖票类  \* **@date** 2018年2月4日 下午2:14:50  \* Company www.igeekhome.com  \*  \* 定义卖票的线程执行目标类  \*/  **public** **class** Ticket **implements** Runnable{  //在成员变量位置将票定义为数字100  /\*\*  \* **@Fields** number :票数  \*/  **private** **int** number = 100;    /\*\*  \* **@Title**: run  \* **@Description**: 完成卖票的线程逻辑  \* **@see** java.lang.Runnable#run()  \*/  @Override  **public** **void** run() {  //车站不停地在卖票  **while**(**true**){  //有票就买票  **if**(number>0){  System.***out***.println(Thread.*currentThread*().getName()+"正在销售第"+(number--)+"张票");  }**else**{  //没有票，就跳出循环，不再卖票  **break**;  }  }  }  } |

# 线程池与线程生命周期

## 线程生命周期



## 线程池



A：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。

B：我们详细的解释一下为什么要使用线程池？

在java中，如果每个请求到达就创建一个新线程，开销是相当大的。在实际使用中，创建和销毁线程花费的时间和消耗的系统资源都相当大，甚至可能要比在处理实际的用户请求的时间和资源要多的多。除了创建和销毁线程的开销之外，活动的线程也需要消耗系统资源。如果在一个jvm里创建太多的线程，可能会使系统由于过度消耗内存或“切换过度”而导致系统资源不足。为了防止资源不足，需要采取一些办法来限制任何给定时刻处理的请求数目，尽可能减少创建和销毁线程的次数，特别是一些资源耗费比较大的线程的创建和销毁，尽量利用已有对象来进行服务。

线程池主要用来解决线程生命周期开销问题和资源不足问题。通过对多个任务重复使用线程，线程创建的开销就被分摊到了多个任务上了，而且由于在请求到达时线程已经存在，所以消除了线程创建所带来的延迟。这样，就可以立即为请求服务，使用应用程序响应更快。另外，通过适当的调整线程中的线程数目可以防止出现资源不足的情况。

### 利用Runnable接口向线程池提交任务

通常，线程池都是通过线程池工厂创建，再调用线程池中的方法获取线程，再通过线程去执行任务方法。

A：Executors：线程池创建工厂类

public static ExecutorService newFixedThreadPool(int nThreads)： 返回线程池对象

B：ExecutorService：线程池类

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。

C：使用线程池中线程对象的步骤（线程池创建与使用）：

创建线程池对象

创建Runnable接口实现类对象

提交Runnable接口实现类对象

关闭线程池

#### 案例代码十:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;  /\*\*  \* **@ClassName**: ThreadPoolDemo  \* **@Description**: 线程池的使用  \* **@date** 2018年2月4日 下午3:48:52  \* Company www.igeekhome.com  \*  \* 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。  \*  \* Executors：线程池创建工厂类  \* 返回线程池方法：  \* public static ExecutorService newFixedThreadPool(int nThreads)：  \* 返回固定线程个数的线程池  \*  \* ExecutorService：线程池类  \* 线程池赋值线程的生命周期，我们只需要向线程提交执行目标，线程池会自动分配线程，执行对应的操作  \*  \* 1、定义线程执行目标  \* 这里使用卖票的线程执行目标  \* 2、向线程池提交线程  \* Future<?> submit(Runnable task):  \* 接收一个Runnable，执行该线程执行目标  \*  \*/  **public** **class** ThreadPoolDemo {  **public** **static** **void** main(String[] args) {  //返回一个线程池  ExecutorService threadPool = Executors.*newFixedThreadPool*(3);    //创建线程执行目标  Ticket ticket = **new** Ticket();    //向线程池提交任务  threadPool.submit(ticket);  threadPool.submit(ticket);  threadPool.submit(ticket);  threadPool.submit(ticket);  threadPool.submit(ticket);    //可以在适当的时候，关闭线程池  //threadPool.shutdown();  }  } |

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: Ticket  \* **@Description**: 卖票类  \* **@date** 2018年2月4日 下午2:14:50  \* Company www.igeekhome.com  \*  \* 定义卖票的线程执行目标类  \*/  **public** **class** Ticket **implements** Runnable{  //在成员变量位置将票定义为数字100  /\*\*  \* **@Fields** number :票数  \*/  **private** **int** number = 100;    /\*\*  \* **@Title**: run  \* **@Description**: 完成卖票的线程逻辑  \* **@see** java.lang.Runnable#run()  \*/  @Override  **public** **void** run() {  //车站不停地在卖票  **while**(**true**){  //有票就买票  **if**(number>0){  System.***out***.println(Thread.*currentThread*().getName()+"正在销售第"+(number--)+"张票");  }**else**{  //没有票，就跳出循环，不再卖票  **break**;  }  }  }  } |

**package** com.igeek03.threadpool;

/\*

\* 定义卖票的线程执行目标类

\*/

**publicclass** Ticket **implements** Runnable {

//在成员位置将票定义为数字100

**int**number = 100;

/\*

\* 完成卖票的线程逻辑

\*/

@Override

**publicvoid** run() {

//车站不停地在卖票

**while**(**true**) {

//有票,就卖票

**if**(number>0) {

System.***out***.println(Thread.*currentThread*().getName()+"正在销售第"+ number-- +"张票");

}**else** {

//没有票,就跳出循环,不再卖票

**break**;

}

}

}

}

**package** com.igeek03.threadpool;

**import**java.util.concurrent.ExecutionException;

**import**java.util.concurrent.ExecutorService;

**import**java.util.concurrent.Executors;

**import**java.util.concurrent.Future;

/\*

\* 将线程放置到同一个线程池中，其中的线程可以反复使用，无需反复创建线程而消耗过多资源。

\* Executors：线程池创建工厂类

\* 返回线程池方法

\* public static ExecutorServicenewXXXThreadPool(int n)

\* public static ExecutorServicenewFixedThreadPool(intnThreads) 返回固定线程个数的线程池

\*

\* ExecutorService:线程池类

\*

\* 线程池负责线程的生命周期,我们只需要向线程提交线程执行目标,线程池会自动分配线程,执行对应的操作

\* 1,定义线程执行目标

\* 这里使用卖票的线程执行目标

\* 2,向线程池提交线程

\* a)

\* Future<?>submit(Runnable task) 接受一个runnable,执行该线程执行目标

\*/

**publicclass** Demo01ThreadPool {

**publicstaticvoid** main(String[] args) **throws**InterruptedException, ExecutionException {

//返回一个线程池

ExecutorServicethreadPool = Executors.*newFixedThreadPool*(2);

//创建线程执行目标

Ticket task = **new**Ticket();

//向线程池提交任务

threadPool.submit(task);

threadPool.submit(task);

}

### 利用Callable接口向线程池提交任务

A：Callable接口：

与Runnable接口功能相似，用来指定线程的任务。其中的call()方法，用来返回线程任务执行完毕后的结果，call方法可抛出异常。

B：ExecutorService：线程池类

<T> Future<T> **submit**(Callable<T> task)：

获取线程池中的某一个线程对象，并执行线程中的call()方法

Future接口：

用来记录线程任务执行完毕后产生的结果。

C：使用线程池中线程对象的步骤（线程池创建与使用）：

创建线程池对象

创建Callable接口实现类对象

提交Callable接口实现类对象

关闭线程池

#### 案例代码十一:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.util.concurrent.ExecutionException;  **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;  **import** java.util.concurrent.Future;  /\*\*  \* **@ClassName**: ThreadPoolDemo2  \* **@Description**: 线程池的使用  \* **@date** 2018年2月4日 下午3:48:52  \* Company www.igeekhome.com  \*  \* 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。  \*  \* Executors：线程池创建工厂类  \* 返回线程池方法：  \* public static ExecutorService newFixedThreadPool(int nThreads)：  \* 返回固定线程个数的线程池  \*  \* ExecutorService：线程池类  \* 线程池赋值线程的生命周期，我们只需要向线程提交执行目标，线程池会自动分配线程，执行对应的操作  \*  \* 1、定义线程执行目标  \* 这里使用卖票的线程执行目标  \* 2、向线程池提交线程  \* a、Future<?> submit(Runnable task):  \* 接收一个Runnable，执行该线程执行目标  \* b、<T> Future<T> submit(Callable<T> task)：  \* 接收一个Callable，执行该线程执行目标  \* Callable<T>:相当于有返回值的Runnable  \* V call() throws Exception  \* Future：任务的结果  \* V get() throws InterruptedException, ExecutionException  \* 返回call方法的结果  \*  \*/  **public** **class** ThreadPoolDemo2 {  **public** **static** **void** main(String[] args) **throws** InterruptedException, ExecutionException {  //返回一个线程池  ExecutorService threadPool = Executors.*newFixedThreadPool*(3);    //创建线程执行目标  Ticket ticket = **new** Ticket();    //向线程池提交任务  threadPool.submit(ticket);  threadPool.submit(ticket);  threadPool.submit(ticket);  threadPool.submit(ticket);  threadPool.submit(ticket);    //对应后边Callable的返回值Future，没有返回值的run方法，返回值是null，没有意义，不会这样使用  Future<?> ticketFuture = threadPool.submit(ticket);  System.***out***.println(ticketFuture.get());    //创建带返回值的线程执行目标  MyCallable callable = **new** MyCallable();  //向线程池提交任务，并返回线程执行目标的结果  Future<String> future = threadPool.submit(callable);  //从执行结果中返回call的具体返回值  String result = future.get();  System.***out***.println(result);    //可以在适当的时候，关闭线程池  //threadPool.shutdown();  }  } |

|  |
| --- |
| **package** com.igeek\_02;  **import** java.util.concurrent.Callable;  /\*\*  \* **@ClassName**: MyCallable  \* **@Description**: 我的Callable实现类  \* **@date** 2018年2月4日 下午4:11:39  \* Company www.igeekhome.com  \*  \* 定义线程执行目标  \*/  **public** **class** MyCallable **implements** Callable<String>{  /\*\*  \* **@Title**: call  \* **@Description**: 定义线程执行目标逻辑  \* **@return**  \* **@throws** Exception  \* **@see** java.util.concurrent.Callable#call()  \*/  @Override  **public** String call() **throws** Exception {  **return** "我是线程任务的返回值结果";  }  } |

重点和总结

1、面向对象的回顾

2、常用API的回顾

3、集合工具类的回顾

4、IO操作及递归的回顾

5、多线程的基本概念

6、多线程的实现方式

7、线程的生命周期

8、线程池