第3章 chapter 3

面向对象(上)

本章重点

- 面向对象的概念
- 类与对象
- 构造方法
- this 和 static 关键字
- 内部类

Java 是一种面向对象的程序设计语言,了解面向对象的编程思想对于学习 Java 开发相当重要。在接下来的两个章节中,将为大家详细讲解如何使用面向对象的思想开发 Java 应用。

3.1 面向对象的概念

面向对象是一种符合人类思维习惯的编程思想。现实生活中存在各种形态不同的事物,这些事物之间存在着各种各样的联系。在程序中使用对象来映射现实中的事物,使用对象的关系来描述事物之间的联系,这种思想就是面向对象。

提到面向对象,自然会想到面向过程,面向过程就是分析解决问题所需要的步骤,然后用函数把这些步骤一一实现,使用的时候一个一个依次调用就可以了。面向对象则是把解决的问题按照一定规则划分为多个独立的对象,然后通过调用对象的方法来解决问题。当然,一个应用程序会包含多个对象,通过多个对象的相互配合来实现应用程序的功能,这样当应用程序功能发生变动时,只需要修改个别的对象就可以了,从而使代码更容易得到维护。面向对象的特点主要可以概括为封装性、继承性和多态性,接下来针对这三种特性进行简单介绍。

1. 封装性

封装是面向对象的核心思想,将对象的属性和行为封装起来,不需要让外界知道具体实现细节,这就是封装思想。例如,用户使用电脑,只需要使用手指敲键盘就可以了,无须知道电脑内部是如何工作的,即使用户可能碰巧知道电脑的工作原理,但在使用时,

并不完全依赖电脑工作原理这些细节。

2. 继承性

继承性主要描述的是类与类之间的关系,通过继承,可以在无须重新编写原有类的情况下,对原有类的功能进行扩展。例如,有一个汽车的类,该类中描述了汽车的普通特性和功能,而轿车的类中不仅应该包含汽车的特性和功能,还应该增加轿车特有的功能,这时,可以让轿车类继承汽车类,在轿车类中单独添加轿车特性的方法就可以了。继承不仅增强了代码复用性,提高了开发效率,而且为程序的修改补充提供了便利。

3. 多态性

多态性指的是在程序中允许出现重名现象,它指在一个类中定义的属性和方法被其他类继承后,它们可以具有不同的数据类型或表现出不同的行为,这使得同一个属性和方法在不同的类中具有不同的语义。例如,当听到“Cut”这个单词时,理发师的行为是剪发,演员的行为是停止表演,不同的对象,所表现的行为是不一样的。

面向对象的思想光靠上面的介绍是无法真正理解的,只有通过大量的实践去学习和理解,才能将面向对象真正领悟。接下来的第 3 章、第 4 章将围绕着面向对象的三个特征(封装、继承、多态)来讲解 Java 这门编程语言。

3.2 类与对象

面向对象的编程思想力图在程序中对事物的描述与该事物在现实中的形态保持一致。为了做到这一点,面向对象的思想中提出两个概念,即类和对象。其中,类是对某一类事物的抽象描述,而对象用于表示现实中该类事物的个体。接下来通过一个图例来描述类与对象的关系,如图 3-1 所示。

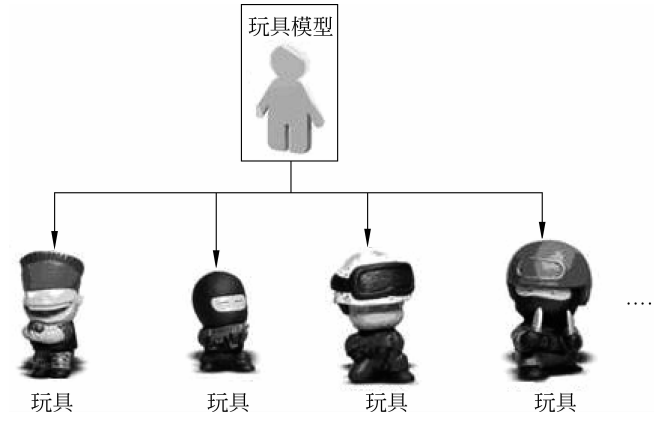

图 3-1 类与对象

在图 3-1 中,可以将玩具模型看作一个类,将一个个玩具看作对象,从玩具模型和玩

具之间的关系便可以看出类与对象之间的关系。类用于描述多个对象的共同特征，它是对象的模板。对象用于描述现实中的个体，它是类的实例。从图 3-1 可以明显看出对象是根据类创建的，并且通过一个类可以创建多个对象。

3.2.1 类的定义

在面向对象的思想中最核心的就是对象，为了在程序中创建对象，首先需要定义一个类。类是对象的抽象，它用于描述一组对象的共同特征和行为。类中可以定义成员变量和成员方法，其中成员变量用于描述对象的特征，也被称作属性，成员方法用于描述对象的行为，可简称为方法。接下来通过一个案例来学习如何定义一个类，如例 3-1 所示。

例 3-1 Person.java

```
1 class Person {
2     int age;            //定义 int 类型的变量 age
3     //定义 speak() 方法
4     void speak() {
5         System.out.println("大家好,我今年"+age+"岁!");
6     }
7 }
```

例 3-1 中定义了一个类。其中，Person 是类名，age 是成员变量，speak()是成员方法。在成员方法 speak()中可以直接访问成员变量 age。

脚下留心

在 Java 中，定义在类中的变量被称为成员变量，定义在方法中的变量被称为局部变量。如果在某一个方法中定义的局部变量与成员变量同名，这种情况是允许的，此时方法中通过变量名访问到的是局部变量，而并非成员变量，请阅读下面的示例代码：

```
class Person {
    int age=10;             //类中定义的变量被称作成员变量
    void speak() {
        int age=60;         //方法内部定义的变量被称作局部变量
        System.out.println("大家好,我今年"+age+"岁!");
    }
}
```

上面的代码中，在 Person 类的 speak()方法中有一条打印语句，访问了变量 age，此时访问的是局部变量 age，也就是说当有另外一个程序来调用 speak()方法时，输出的值为 60，而不是 10。

3.2.2 对象的创建与使用

应用程序想要完成具体的功能，仅有类是远远不够的，还需要根据类创建实例对象。

在 Java 程序中可以使用 new 关键字来创建对象，具体格式如下：

```
类名 对象名称=new 类名();
```

例如，创建 Person 类的实例对象代码如下：

```
Person p=new Person();
```

上面的代码中，“new Person()”用于创建 Person 类的一个实例对象，“Person p”则是声明了一个 Person 类型的变量 p。中间的等号用于将 Person 对象在内存中的地址赋值给变量 p，这样变量 p 便持有了对象的引用。本书接下来的章节为了便于描述，通常会将变量 p 引用的对象简称为 p 对象。在内存中变量 p 和对象之间的引用关系如图 3-2 所示。

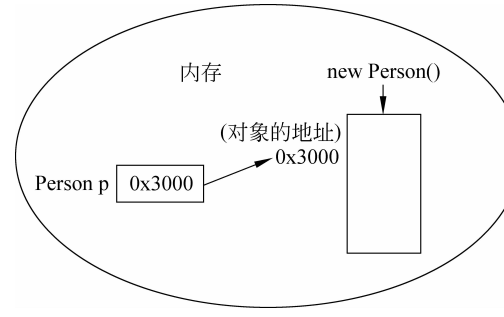

图 3-2 内存分析

在创建 Person 对象后，可以通过对象的引用来访问对象所有的成员，具体格式如下：

```
对象引用.对象成员
```

接下来通过一个案例来学习如何访问对象的成员，如例 3-2 所示。

例 3-2 Example01.java

```
1 class Example01 {
2     public static void main(String[] args) {
3         Person p1=new Person();          //创建第一个 Person 对象
4         Person p2=new Person();          //创建第二个 Person 对象
5         p1.age=18;                       //为 age 属性赋值
6         p1.speak();                      //调用对象的方法
7         p2.speak();
8     }
9 }
```

运行结果如图 3-3 所示。

图 3-3 例 3-2 运行结果

例 3-2 中，p1、p2 分别引用了 Person 类的两个实例对象。从图 3-3 所示的运行结果可以看出，p1 和 p2 对象在调用 speak()方法时，打印的 age 值不相同。这是因为 p1 对象和 p2 对象是两个完全独立的个体，它们分别拥有各自的 age 属性，对 p1 对象的 age 属性进行赋值并不会影响到 p2 对象 age 属性的值。程序运行期间 p1、p2 引用的对象在内存中的状态如图 3-4 所示。

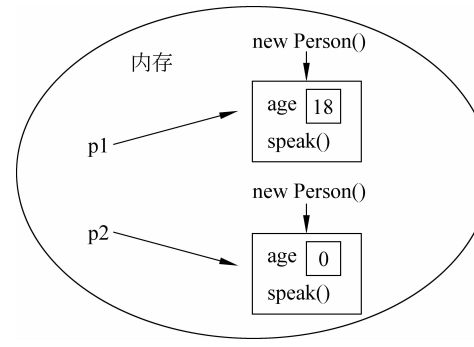

图 3-4　p1、p2 对象在内存中的状态

在例 3-2 中，通过“p1. age＝18”将 p1 对象的 age 属性赋值为 18，但并没有对 p2 对象的 age 属性进行赋值，按理说 p2 对象的 age 属性应该是没有值的。但从图 3-3 所显示的运行结果可以看出 p2 对象的 age 属性也是有值的，其值为 0。这是因为在实例化对象时 ，Java 虚拟机会自动为成员变量进行初始化，针对不同类型的成员变量，Java 虚拟机会赋予不同的初始值，如表 3-1 所示。

表 3-1　成员变量的初始化值

成员变量类型	初始值	成员变量类型	初始值
byte	0	double	0.0D
short	0	char	空字符，'\u0000'
int	0	boolean	false
long	0L	引用数据类型	null
float	0.0F		

当对象被实例化后，在程序中可以通过对象的引用变量来访问该对象的成员。需要注意的是，当没有任何变量引用这个对象时，它将成为垃圾对象，不能再被使用。接下来通过两段程序代码来分析对象是如何成为垃圾的。

第一段程序代码：

```
{
    Person p1=new Person();
    ……
}
```

上面的代码中使用变量 p1 引用了一个 Person 类型的对象，当这段代码运行完毕时，变量 p1 就会超出其作用域而被销毁，这时 Person 类型的对象就没有被任何变量引用，变成垃圾。

第二段程序代码，如例 3-3 所示。

例 3-3 Example02.java

```
1  class Person {
2      void say() {                                  //创建 say()方法,输出一句话
3          System.out.println("我是一个人!");
4      }
5  }
6  class Example02 {
7      public static void main(String[] args) {
8          Person p2=new Person();                   //创建 Person 对象
9          p2.say();                                 //调用 say()方法
10         p2=null;                                  //将 Person 对象置为 null
11         p2.say();
12     }
13 }
```

运行结果如图 3-5 所示。

图 3-5 例 3-3 运行结果

在例 3-3 中,创建了一个 Person 类的实例对象,并两次调用了该对象的 say()方法。第一次调用 say()方法时可以正常打印,但在第 10 行代码中将变量 p2 的值置为 null,当再次调用 say()方法时抛出了空指针异常。在 Java 中,null 是一种特殊的常量,当一个变量的值为 null 时,则表示该变量不指向任何一个对象。当把变量 p2 置为 null 时,被 p2 所引用的 Person 对象就会失去引用,成为垃圾对象,其过程如图 3-6 所示。

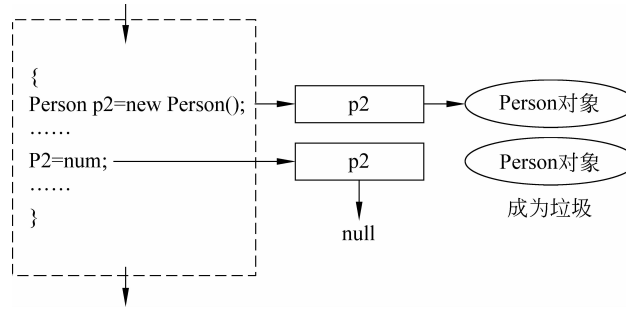

图 3-6 垃圾对象

3.2.3 类的设计

在 Java 中,对象是通过类创建出来的。因此,在程序设计时,最重要的就是类的设计。接下来通过一个具体的案例来学习如何设计一个类。

假设要在程序中描述一个学校所有学生的信息，可以先设计一个学生类(Student)，在这个类中定义两个属性 name、age 分别表示学生的姓名和年龄，定义一个方法 introduce()表示学生做自我介绍。根据上面的描述设计出来的 Student 类如例 3-4 所示。

例 3-4 Student.java

```
1 public class Student {
2     String name;
3     int age;
4     public void introduce() {
5         //方法中打印属性 name 和 age 的值
6         System.out.println("大家好,我叫"+name+",我今年"+age+"岁!");
7     }
8 }
```

在例 3-4 的 Student 类中，定义了两个属性 name 和 age。其中的 name 属性为 String 类型，在 Java 中使用 String 类的实例对象表示一个字符串，例如：

```
String name="李芳";
```

关于字符串的相关知识在本书的第 6 章将会进行详细地讲解，在此处可简单地将字符串理解为一连串的字符。

3.2.4 类的封装

接下来针对例 3-4 中设计的 Student 类创建对象，并访问该对象的成员，如例 3-5 所示。

例 3-5 Example03.java

```
1 public class Example03 {
2     public static void main(String[] args) {
3         Student stu=new Student();          //创建学生对象
4         stu.name="李芳";                    //为对象的 name 属性赋值
5         stu.age=-30;                        //为对象的 age 属性赋值
6         stu.introduce();                    //调用对象的方法
7     }
8 }
```

运行结果如图 3-7 所示。

图 3-7 例 3-5 运行结果

在例 3-5 的第 5 行代码中,将年龄赋值为一个负数－30,这在程序中不会有任何问题,但在现实生活中明显是不合理的。为了解决年龄不能为负数的问题,在设计一个类时,应该对成员变量的访问做出一些限定,不允许外界随意访问。这就需要实现类的封装。

所谓类的封装是指在定义一个类时,将类中的属性私有化,即使用 private 关键字来修饰,私有属性只能在它所在类中被访问。为了能让外界访问私有属性,需要提供一些使用 public 修饰的公有方法,其中包括用于获取属性值的 getXxx()方法和设置属性值的 setXxx()方法。接下来通过一个案例来实现类的封装,如例 3-6 所示。

例 3-6 Example04 .java

```
1  class Student {
2      private String name;                    //将 name 属性私有化
3      private int age;                        //将 age 属性私有化
4      //下面是公有的 getXxx()和 setXxx()方法
5      public String getName() {
6          return name;
7      }
8      public void setName(String stuName) {
9          name=stuName;
10     }
11     public int getAge() {
12         return age;
13     }
14     public void setAge(int stuAge) {
15         //下面是对传入的参数进行检查
16         if (stuAge<=0) {
17             System.out.println("年龄不合法……");
18         } else {
19             age=stuAge;                     //对属性赋值
20         }
21     }
22     public void introduce() {
23         System.out.println("大家好,我叫"+name+",我今年"+age+"岁!");
24     }
25 }
26 public class Example04 {
27     public static void main(String[] args) {
28         Student stu=new Student();
29         stu.setAge(-30);
30         stu.setName("李芳");
31         stu.introduce();
32     }
33 }
```

运行结果如图 3-8 所示。

图 3-8　例 3-6 运行结果

在例 3-6 的 Student 类中，使用 private 关键字将属性 name 和 age 声明为私有，对外界提供了几个公有的方法，其中 getName()方法用于获取 name 属性的值，setName()方法用于设置 name 属性的值，同理，getAge()和 setAge()方法用于获取和设置 age 属性的值。在 main()方法中创建 Student 对象，并调用 setAge()方法传入一个负数－30，在 setAge()方法中对参数 stuAge 的值进行检查，由于当前传入的值小于 0，因此会打印“年龄不合法”的信息，age 属性没有被赋值，仍为默认初始值 0。

3.3　构造方法

从前面所学到的知识可以发现，实例化一个类的对象后，如果要为这个对象中的属性赋值，则必须要通过直接访问对象的属性或调用 setXxx()方法的方式才可以。如果需要在实例化对象的同时就为这个对象的属性进行赋值，可以通过构造方法来实现。构造方法是类的一个特殊成员，它会在类实例化对象时被自动调用。接下来学习构造方法的具体用法。

3.3.1　构造方法的定义

在一个类中定义的方法如果同时满足以下三个条件，该方法称为构造方法，具体如下：

① 方法名与类名相同。

② 在方法名的前面没有返回值类型的声明。

③ 在方法中不能使用 return 语句返回一个值。

接下来通过一个案例来演示如何在类中定义构造方法，如例 3-7 所示。

例 3-7　Example05.java

```
1  class Person {
2      //下面是类的构造方法
3      public Person() {
4          System.out.println("无参的构造方法被调用了……");
5      }
6  }
7  public class Example05 {
```

```
8      public static void main(String[] args) {
9          Person p=new Person();          //实例化 Person 对象
10     }
11 }
```

运行结果如图 3-9 所示。

图 3-9 例 3-7 运行结果

在例 3-7 的 Person 类中定义了一个无参的构造方法 Person()。从运行结果可以看出,Person 类中无参的构造方法被调用了。这是因为第 9 行代码在实例化 Person 对象时会自动调用类的构造方法,“new Person()”语句的作用除了会实例化 Person 对象,还会调用构造方法 Person()。

在一个类中除了定义无参的构造方法,还可以定义有参的构造方法,通过有参的构造方法就可以实现对属性的赋值。接下来对例 3-7 进行改写,改写后的代码如例 3-8 所示。

例 3-8 Example06.java

```
1  class Person {
2      int age;
3      //定义有参的构造方法
4      public Person(int a) {
5          age=a;                          //为 age 属性赋值
6      }
7      public void speak() {
8          System.out.println("I am "+age+" years old.!");
9      }
10 }
11 public class Example06 {
12     public static void main(String[] args) {
13         Person p=new Person(20);        //实例化 Person 对象
14         p.speak();
15     }
16 }
```

运行结果如图 3-10 所示。

例 3-8 的 Person 类中定义了有参的构造方法 Person(int a)。第 13 行代码中的“new Person(20)”会在实例化对象的同时调用有参的构造方法,并传入了参数 20。在构造方法 Person(int a)中将 20 赋值给对象的 age 属性。通过运行结果可以看出,Person 对象

图 3-10　例 3-8 运行结果

在调用 speak()方法时，其 age 属性已经被赋值为 20。

3.3.2　构造方法的重载

与普通方法一样，构造方法也可以重载，在一个类中可以定义多个构造方法，只要每个构造方法的参数类型或参数个数不同即可。在创建对象时，可以通过调用不同的构造方法为不同的属性赋值。接下来通过一个案例来学习构造方法的重载，如例 3-9 所示。

例 3-9　Example07.java

```
1  class Person {
2      String name;
3      int age;
4      //定义两个参数的构造方法
5      public Person(String con_name,int con_age) {
6          name=con_name;              //为 name 属性赋值
7          age=con_age;                //为 age 属性赋值
8      }
9      //定义一个参数的构造方法
10     public Person(String con_name) {
11         name=con_name;              //为 name 属性赋值
12     }
13     public void speak() {
14         //打印 name 和 age 的值
15         System.out.println("大家好,我叫"+name+",我今年"+age+"岁!");
16     }
17 }
18 public class Example07 {
19     public static void main(String[] args) {
20         //分别创建两个对象 p1 和 p2
21         Person p1=new Person("陈杰");
22         Person p2=new Person("李芳",18);
23         //通过对象 p1 和 p2 调用 speak()方法
24         p1.speak();
25         p2.speak();
26     }
27 }
```

运行结果如图 3-11 所示。

图 3-11 例 3-9 运行结果

例 3-9 的 Person 类中定义了两个构造方法，它们构成了重载。在创建 p1 对象和 p2 对象时，根据传入参数的不同，分别调用不同的构造方法。从程序的运行结果可以看出，两个构造方法对属性赋值的情况是不一样的，其中一个参数的构造方法只针对 name 属性进行赋值，这时 age 属性的值为默认值 0。

脚下留心

① 在 Java 中的每个类都至少有一个构造方法，如果在一个类中没有定义构造方法，系统会自动为这个类创建一个默认的构造方法，这个默认的构造方法没有参数，在其方法体中没有任何代码，即什么也不做。

下面程序中 Person 类的两种写法效果是完全一样的。

第一种写法：

```
class Person
{
}
```

第二种写法：

```
class Person {
    public Person() {
    }
}
```

对于第一种写法，类中虽然没有声明构造方法，但仍然可以用 new Person()语句来创建 Person 类的实例对象。由于系统提供的构造方法往往不能满足需求，因此，我们可以自己在类中定义构造方法，一旦为该类定义了构造方法，系统就不再提供默认的构造方法了，具体代码如下所示。

```
class Person {
    int age;
    public Person(int x) {
        age=x;
    }
}
```

上面的 Person 类中定义了一个对成员变量赋初值的构造方法，该构造方法有一个参

数，这时系统就不再提供默认的构造方法，接下来再编写一个测试程序调用上面的 Person 类，如例 3-10 所示。

例 3-10 Example08.java

```
1 public class Example08 {
2     public static void main(String[] args) {
3         Person p=new Person();      //实例化 Person 对象
4     }
5 }
```

编译程序报错，结果如图 3-12 所示。

图 3-12 例 3-10 运行结果

从图 3-12 可以看出程序在编译时报错，其原因是调用 new Person()创建 Person 类的实例对象时，需要调用无参的构造方法，而我们并没有定义无参的构造方法，只是定义了一个有参的构造方法，系统将不再自动生成无参的构造方法。为了避免出现上面的错误，在一个类中如果定义了有参的构造方法，最好再定义一个无参的构造方法。

② 思考一下，声明构造方法时，可以使用 private 访问修饰符吗？下面就来运行一下例 3-11，看看会出现什么结果。

例 3-11 Example09.java

```
1  class Person {
2      //定义构造方法
3      private Person() {
4          System.out.println("调用无参的构造方法");
5      }
6  }
7  public class Example09 {
8      public static void main(String[] args) {
9          Person p=new Person();
10     }
11 }
```

编译程序报错，结果如图 3-13 所示。

从图 3-13 中可以看出，程序在编译时出现了错误，错误提示为 private 关键字修饰的

图 3-13 例 3-11 运行结果

构造方法 Person()只能在 Person 类中被访问。也就是说 Person()构造方法是私有的，不可以被外界调用，也就无法在类的外部创建该类的实例对象。因此，为了方便实例化对象，构造方法通常会使用 public 来修饰。

3.4 this 关键字

在例 3-9 中使用变量表示年龄时，构造方法中使用的是 con_age，成员变量使用的是 age，这样的程序可读性很差。这时需要将一个类中表示年龄的变量进行统一的命名，例如都声明为 age。但是这样做又会导致成员变量和局部变量的名称冲突，在方法中将无法访问成员变量 age。为了解决这个问题，Java 中提供了一个关键字 this，用于在方法中访问对象的其他成员。接下来将为大家详细地讲解 this 关键字在程序中的三种常见用法，具体如下：

① 通过 this 关键字可以明确地去访问一个类的成员变量，解决与局部变量名称冲突问题。具体示例代码如下：

```
class Person {
    int age;
    public Person(int age) {
        this.age=age;
    }
    public int getAge() {
        return this.age;
    }
}
```

在上面的代码中，构造方法的参数被定义为 age，它是一个局部变量，在类中还定义了一个成员变量，名称也是 age。在构造方法中如果使用“age”，则是访问局部变量，但如果使用“this.age”则是访问成员变量。

② 通过 this 关键字调用成员方法，具体示例代码如下：

```
class Person {
    public void openMouth() {
        ⋮
    }
```

```
    public void speak() {
        this.openMouth();
    }
}
```

在上面的 speak()方法中,使用 this 关键字调用 openMouth()方法。注意,此处的 this 关键字可以省略不写,也就是说上面的第 6 行代码写成“this. openMouth()”和“openMouth()”,效果是完全一样的。

③ 构造方法是在实例化对象时被 Java 虚拟机自动调用的,在程序中不能像调用其他方法一样去调用构造方法,但可以在一个构造方法中使用“this([参数 1,参数 2…])”的形式来调用其他的构造方法。接下来通过一个案例来演示,如例 3-12 所示。

例 3-12 Example10. java

```
1  class Person {
2      public Person() {
3          System.out.println("无参的构造方法被调用了……");
4      }
5      public Person(String name) {
6          this();                                    //调用无参的构造方法
7          System.out.println("有参的构造方法被调用了……");
8      }
9  }
10 public class Example10 {
11     public static void main(String[] args) {
12         Person p=new Person("itcast");      //实例化 Person 对象
13     }
14 }
```

运行结果如图 3-14 所示。

图 3-14 例 3-12 运行结果

例 3-12 中第 12 行代码在实例化 Person 对象时,调用了有参的构造方法,在该方法中通过 this()调用了无参的构造方法,因此运行结果中显示两个构造方法都被调用了。

在使用 this 调用类的构造方法时,应注意以下几点。

① 只能在构造方法中使用 this 调用其他的构造方法,不能在成员方法中使用。

② 在构造方法中,使用 this 调用构造方法的语句必须位于第一行,且只能出现一次。下面的写法是非法的。

```
public Person() {
    String name="小芳";
    this(name);          //调用有参的构造方法。由于不在第一行,编译错误!
}
```

③ 不能在一个类的两个构造方法中使用 this 互相调用,下面的写法编译会报错。

```
class Person {
    public Person() {
        this("小芳");                          //调用有参的构造方法
        System.out.println("无参的构造方法被调用了……");
    }
    public Person(String name) {
        this();                                //调用无参的构造方法
        System.out.println("有参的构造方法被调用了……");
    }
}
```

3.5 垃圾回收

在 Java 中,当一个对象成为垃圾后仍会占用内存空间,时间一长,就会导致内存空间的不足。针对这种情况,Java 中引入了垃圾回收机制。程序员不需要过多关心垃圾对象回收的问题,Java 虚拟机会自动回收垃圾对象所占用的内存空间。

一个对象在成为垃圾后会暂时地保留在内存中,当这样的垃圾堆积到一定程度时,Java 虚拟机就会启动垃圾回收器将这些垃圾对象从内存中释放,从而使程序获得更多可用的内存空间。除了等待 Java 虚拟机进行自动垃圾回收,也可以通过调用 System.gc()方法来通知 Java 虚拟机立即进行垃圾回收。当一个对象在内存中被释放时,它的 finalize()方法会被自动调用,因此可以在类中通过定义 finalize()方法来观察对象何时被释放。接下来通过一个案例来演示 Java 虚拟机进行垃圾回收的过程,如例 3-13 所示。

例 3-13 Example11.java

```
1  class Person {
2      //下面定义的 finalize 方法会在垃圾回收前被调用
3      public void finalize() {
4          System.out.println("对象将被作为垃圾回收……");
5      }
6  }
7  public class Example11{
8      public static void main(String[] args) {
9          //下面是创建了两个 Person 对象
```

```
10          Person p1=new Person ();
11          Person p2=new Person ();
12          //下面将变量置为 null,让对象成为垃圾
13          p1=null;
14          p2=null;
15          //调用方法进行垃圾回收
16          System.gc();
17          for (int i=0; i<1000000; i++) {
18              //为了延长程序运行的时间
19          }
20      }
21  }
```

运行结果如图 3-15 所示。

图 3-15　例 3-13 运行结果

在例 3-13 的 Person 类中定义了一个 finalize()方法，该方法的返回值必须为 void，并且要使用 public 来修饰。在 main()方法中创建了两个对象 p1 和 p2，然后将两个变量置为 null，这意味着新创建的两个对象成为垃圾了，紧接着通过“System. gc()”语句通知虚拟机进行垃圾回收。从运行结果可以看出，虚拟机针对两个垃圾对象进行了回收，并在回收之前分别调用两个对象的 finalize()方法。

需要注意的是，Java 虚拟机的垃圾回收操作是在后台完成的，程序结束后，垃圾回收的操作也将终止。因此，在程序的最后使用了一个 for 循环，延长程序运行的时间，从而能够更好地看到垃圾对象被回收的过程。

3.6　static 关键字

在 Java 中，定义了一个 static 关键字，它用于修饰类的成员，如成员变量、成员方法以及代码块等，被 static 修改的成员具备一些特殊性，接下来将对这些特殊性进行逐一地讲解。

3.6.1　静态变量

在定义一个类时，只是在描述某类事物的特征和行为，并没有产生具体的数据。只有通过 new 关键字创建该类的实例对象后，系统才会为每个对象分配空间，存储各自的数据。有时候，我们希望某些特定的数据在内存中只有一份，而且能够被一个类的所有

实例对象所共享。例如某个学校所有学生共享同一个学校名称，此时完全不必在每个学生对象所占用的内存空间中都定义一个变量来表示学校名称，而可以在对象以外的空间定义一个表示学校名称的变量让所有对象来共享。具体内存中的分配情况如图 3-16 所示。

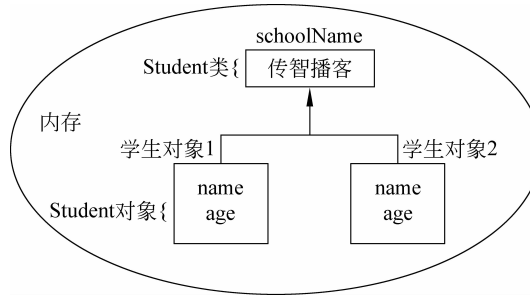

图 3-16 内存分配图

在一个 Java 类中，可以使用 static 关键字来修饰成员变量，该变量被称作静态变量。静态变量被所有实例共享，可以使用“类名.变量名”的形式来访问。接下来通过一个案例来实现图 3-16 描述的情况，如例 3-14 所示。

例 3-14 Example12.java

```
1  class Student {
2      static String schoolName;                    //定义静态变量 schoolName
3  }
4  public class Example12 {
5      public static void main(String[] args) {
6          Student stu1=new Student();              //创建学生对象
7          Student stu2=new Student();
8          Student.schoolName="传智播客";            //为静态变量赋值
9          System.out.println("我的学校是"+stu1.schoolName);   //打印第一个学生对象的学校
10         System.out.println("我的学校是"+stu2.schoolName);   //打印第二个学生对象的学校
11     }
12 }
```

运行结果如图 3-17 所示。

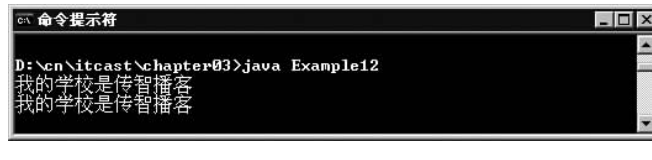

图 3-17 例 3-14 运行结果

例 3-14 的 Student 类中定义了一个静态变量 schoolName，用于表示学生所在的学校，它被所有的实例所共享。由于 schoolName 是静态变量，因此可以直接使用

Student.schoolName 的方式进行调用，也可以通过 Student 的实例对象进行调用，如 stu2.schoolName。第 8 行代码将变量 schoolName 赋值为“传智播客”，通过运行结果可以看出学生对象 stu1 和 stu2 的 schoolName 属性均为“传智播客”。

注意：static 关键字只能用于修饰成员变量，不能用于修饰局部变量，否则编译会报错，下面的代码是非法的。

```
public class Student {
    public void study() {
        static int num=10;     //这行代码是非法的，编译会报错
    }
}
```

3.6.2 静态方法

有时我们希望在不创建对象的情况下就可以调用某个方法，换句话说也就是使该方法不必和对象绑在一起。要实现这样的效果，只需要在类中定义的方法前加上 static 关键字即可，我们称这种方法为静态方法。同静态变量一样，静态方法可以使用“类名.方法名”的方式来访问，也可以通过类的实例对象来访问。接下来通过一个案例来学习静态方法的使用，如例 3-15 所示。

例 3-15 Example13.java

```
1  class Person {
2      public static void sayHello() {        //定义静态方法
3          System.out.println("hello");
4      }
5  }
6  class Example13 {
7      public static void main(String[] args) {
8          Person.sayHello();                 //调用静态方法
9      }
10 }
```

运行结果如图 3-18 所示。

图 3-18 例 3-15 运行结果

例 3-15 的 Person 类中定义了静态方法 sayHello()，在第 8 行代码处通过“Person.sayHello()”的形式调用了静态方法，由此可见静态方法不需要创建对象就可以调用。

注意：在一个静态方法中只能访问用 static 修饰的成员，原因在于没有被 static 修饰的成员需要先创建对象才能访问，而静态方法在被调用时可以不创建任何对象。

3.6.3 静态代码块

在 Java 类中，使用一对大括号包围起来的若干行代码被称为一个代码块，用 static 关键字修饰的代码块称为静态代码块。当类被加载时，静态代码块会执行，由于类只加载一次，因此静态代码块只执行一次。在程序中，通常会使用静态代码块来对类的成员变量进行初始化。接下来通过一个案例来学习静态代码块的使用，如例 3-16 所示。

例 3-16 Example14.java

```
1  class Example14 {
2      //静态代码块
3      static {
4          System.out.println("测试类的静态代码块执行了");
5      }
6      public static void main(String[] args) {
7          //下面的代码创建了两个 Person 对象
8          Person p1=new Person();
9          Person p2=new Person();
10     }
11 }
12 class Person {
13     static String country;
14     //下面是一个静态代码块
15     static {
16         country="china";
17         System.out.println("Person 类中的静态代码块执行了");
18     }
19 }
```

运行结果如图 3-19 所示。

图 3-19 例 3-16 运行结果

从图 3-19 所示的运行结果可以看出，程序中的两段静态代码块都执行了。在命令行窗口输入“java Example14”后，虚拟机首先会加载类 Example14，在加载类的同时就会执行该类的静态代码块，紧接着会调用 main()方法。在该方法中创建了两个 Person 对象，但在两次实例化对象的过程中，静态代码块只执行一次，这就说明类在第一次使用时才