第5章 泛　型

在程序设计过程中，常出现数据类型不一样、对数据处理的过程和逻辑都一样的情况，应该如何实现这种程序呢？如果为每种数据类型编写不同的程序，不仅工作量大，而且不满足代码重用的原则。这就需要泛型。从字面意思上看，泛型是编写的代码适用于广泛的类型。如何做到这点呢？这需要引入参数化类型的概念。所谓的参数化类型，是指编写代码时，它所适用的类型并不立即指明，而是使用参数符号来代替，具体的适用类型延迟到用户使用时才指定。由于参数被延迟指定，因此在使用参数类型的时候，可以根据需要指定它适用的类型。

5.1 概述

JDK5 的新特性之一就是支持泛型，因此用 Java 编写的代码具有更加广泛的表达能力。当然，Java 泛型的作用不止于此，它还保证了程序的类型安全，并消除了一些烦琐的类型转换。然而，Java 的泛型也存在一些限制，大部分是由类型擦除导致的。


扫一扫

5.1.1 使用继承实现代码重用

引入泛型之前，一般的程序都使用多态与继承来提高代码的灵活性和重用性。最常见的用继承实现泛型的就是 List 容器：由于 List 容器不像数组限制得那么严格，允许存放任何 Java 定义的类型，因此可以向容器中添加诸如牙刷、房子、课程这些不相关的对象；实现的方法很简单，List 存放的都是 Object 类型，由于 Java 中除了基本类型外的所有类都继承自 Object，因此可以添加任何类型到 List 中，代码如下。

```
01  package org.ddd.generic.example1;
02  public class GenericTest {
```

```
03      public static void main(String[] args) {
04          List listInteger = new ArrayList();
05          List listString = new ArrayList();
06          List listDate = new ArrayList();
07          listInteger.add(new Integer(1));
08          listString.add(new String("字符串"));
09          listDate.add(new Date());
10          listInteger.add(new String("字符串"));          //逻辑不正确,但语法正确,
11                                                          //这很容易引起错误
12          Integer i = (Integer)listInteger.get(0);
13          String str = (String)listString.get(0);
14          Date date = (Date)listDate.get(0);
15          System.out.println("第一个数组中存放的是数字:" + i);
16          System.out.println("第二个数组中存放的是字符串:" + str);
17          System.out.println("第三个数组中存放的是日期:" + date.toString());
18      }
19  }
```

运行结果如下。

```
01  第一个数组中存放的是数字:1
02  第二个数组中存放的是字符串:字符串
03  第三个数组中存放的是日期:Tue Aug 09 11:03:48 CST 2011
```

这个例子定义了3个List容器list1、list2、list3，并分别向其中存放Integer类型的数组、String类型的字符串以及Date类型的日期。接着获取这些容器中的元素，并将其打印在屏幕上。从这个例子可以看出，List容器的设计并没有针对某个具体的类，可以向其中存放任何继承自Object的对象。

继承可以实现代码的重用，但是，Java满足里氏代换原则（任何父类可以出现的地方，子类一定可以出现），因此父类出现的地方都可以用子类代替，并且没有办法限制具体使用哪个子类。在一些场景中容易产生错误，如例子中的第10行，很容易产生类型转换的错误。Java作为一种强类型的语言，对类型有严格的检查，但是这里失去了作用，这就需要使用泛型。

5.1.2 泛型代码

使用继承实现的List虽然可以适用于很多类型，但也存在一些问题，最显而易见的就是当从List中取出元素时，必须显式地将其转型成需要的类型。有时可以确定List中存放的类型，比如规定在List中只允许存放Integer类型时，明明知道取出的必然是Integer类型，但是每次获取元素时仍然需要显式转换，显然这些转型的代码很烦琐，而且没有必要。还存在其他问题，比如当试图在只允许存放Integer的List中添加字符串类型时，编译器并不会报错，这样错误就会在从List中取出该元素并将其转换成Integer时发生。显然，这种可能出现的情况是无法接受的。针对以上可能出现的情况，泛型机制很好地解决了这些问

题，代码如下。

```
01  package org.ddd.generic.example2;
02  public class GenericTest {
03      public static void main(String[] args) {
04          List<Integer> list1 = new ArrayList<Integer>();
05          List<String> list2 = new ArrayList<String>();
06          List<Date> list3 = new ArrayList<Date>();
07          list1.add(new Integer(1));
08  //          list1.add(new String("测试"));      //编译错误
09          list2.add(new String("字符串"));
10          list3.add(new Date());
11          Integer i = list1.get(0);
12          String str = list2.get(0);
13          Date date = list3.get(0);
14          System.out.println("第一个数组中存放的是数字:" + i);
15          System.out.println("第二个数组中存放的是字符串:" + str);
16          System.out.println("第三个数组中存放的是日期:" +
17  date.toString());
18      }
19  }
```

运行结果如下。

```
01  第一个数组中存放的是数字:1
02  第二个数组中存放的是字符串:字符串
03  第三个数组中存放的是日期:Tue Aug 09 11:03:48 CST 2011
```

以上例子定义了 Java 5 出现的类型参数化 List：List<Integer> list1、List<String> list2、List<Date> list3。使用参数化类型定义 List 后，显而易见的变化就是泛型机制限制了 List 容器中元素的类型，当试图在 list1 中添加 String 类型时，编译器会提示 The method add(Integer) in the type List<Integer> is not applicable for the arguments (String)的错误，提示在 List<Integer>容器中添加 String 类型的元素是非法的。

5.1.3 算法与数据类型解耦

Java 语言是一种强类型语言。强类型语言通常是指在编译或运行时，数据的类型有较为严格的区分，不同数据类型的对象在转型时需要严格的兼容性检查。例如：类型 Integer 和类型 String 是两种不同的类型，编译以下代码时，Java 编译器在进行兼容性检查时将提示错误。

```
01  Integer age;
02  String ageString = "23";
03  age = ageString;
```

强类型的语言对提高程序的健壮性和开发效率都有利，但强类型语言导致一个问题：数据类型与算法在编译时绑定，这意味着必须为不同的数据类型编写相同逻辑的代码，例如比较两个数，必须为不同的数据类型编写如下代码。

```
01  public Integer compare(Integer a1,Integer a2)
02  {
03      return a1-a2;
04  }
05  public Float compare(Float a1, Float a2)
06  {
07      return a1-a2;
08  }
09  public Double compare(Double a1, Double a2)
10  {
11      return a1-a2;
12  }
```

以上代码是丑陋的(任何重复代码，或者重复模式的代码都是丑陋的)。

在软件开发过程中，经常出现这种情况：同一个算法适合所有数据类型，或者几种数据类型，而不是某一种具体数据类型，即编写通用的代码。继承是解决这个问题的方法之一：为适用这一算法的多种数据类型，抽象一个基类(或者接口)，针对基类(或者接口)编写算法。但在实际程序设计过程中，专门为特定的算法修改数据类型的设计不是好的习惯。另外，如果使用已经设计好的数据类型，就没有办法解决问题了。例如上面减法的例子，不可能再为 Integer、Float、Double 添加基类或者接口。泛型是解决“数据类型与算法在编译时绑定”问题的有效方法之一。泛型的最大价值在于：在保证类型安全的前提下，把算法与数据类型解耦。

5.2 泛型类型

上一节介绍了泛型的概念及特点，本节将介绍如何定义泛型类型。

5.2.1 泛型类


扫一扫

泛型类是指该类使用的参数类型作用于整个类，即在类的内部任何地方(不包括静态代码区域)都可把参数类型当作一个真实类型来使用，比如用它作为返回值、定义变量等。泛型类的定义很简单，只需在定义类的时候在类名后加入<T>这样一句代码即可，其中 T 是一个参数，是可变的，代码如下。

```
01  package org.ddd.generic.example4;
02  public class Person<T> {
03      protected T t;
04      public Person(T t){
```

```
        this.t = t;
    }
    public String toString(){
        return "变量 t 的类型是:" + t.getClass().getCanonicalName();
    }
}
```

以上例子定义了泛型类 Person。定义泛型类与普通类的区别在于：在泛型类中使用的类型参数必须在类名后指明，指明的方式就是采用<T>这种方式。当然，也可以为一个类指明多个类型参数，代码如下。

```
package org.ddd.generic.example4;
public class Teacher<V,S> extends Person {
    protected V v;
    private S s;
    public Teacher(Object t) {
        super(t);
    }
    public void set(V v, S s){
        this.v = v;
        this.s = s;
    }
    public String toString(){
        return super.toString()+"\n"+
                "变量 v 的类型是:" + v.getClass().getCanonicalName()+"\n"+
                "变量 s 的类型是:" + s.getClass().getCanonicalName()+"\n";
    }
}
```

定义方式很简单，但定义的时候会出现一个小问题：定义 Teacher 类的时候，编译器给出一个错误提示：由于 Person 类没有定义无参构造函数，因此要求实现一个与父类参数相同的构造函数。这本无可厚非，但当尝试定义一个 T 类型的构造函数时，发现子类中已经无法使用类型参数 T 了，那怎么办呢？看看实现的代码，使用 Object 代替 T，这样做的原因将在讨论泛型擦除时说明。

子类可不可以使用父类的类型参数呢？可以，但需要进行一点修改，代码如下。

```
package org.ddd.generic.example5;
public class Teacher<V,S> extends Person<V> {
    protected V v;
    private S s;
    public Teacher(V t) {
        super(t);
    }
    public void set(V v, S s){
```

```
        this.v = v;
        this.s = s;
    }
    public String toString(){
        return super.toString()+"\n"+
                "变量 v 的类型是:" + v.getClass().getCanonicalName()+"\n"+
                "变量 s 的类型是:" + s.getClass().getCanonicalName()+"\n";
    }
}
```

只是将 extends 后的 Person 改为了 Person<V>,子类就可以与父类一起共享类型参数 V 了,Person 的类型参数 T 被指定为类型参数 V 的类型。这时已经不需要使用 Object 来定义参数了。

泛型类的使用方法也很简单,只需在构造的时候指明参数类型即可,代码如下。

```
package org.ddd.generic.example6;
public class GenericTest<T> {
    public static void  main(String[] args){
        Person<Integer> person = new Person<Integer>(5);
        System.out.println("person======\n"+person);
        //实际的类型也可以不指定,编译器能自动推断出实际的类型
        Person<String> person1 = new Person<>("字符串");
        System.out.println("person1======\n"+person1);
        Teacher<String, Date> teacher = new Teacher<>("字符串");
        teacher.set("xcy",new Date());
        System.out.println("teacher======\n"+teacher);
        //person = person1;                    //报类型不兼容错误
    }
}
```

输出结果如下。

```
person======
变量 t 的类型是:java.lang.Integer
person1======
变量 t 的类型是:java.lang.String
teacher======
变量 t 的类型是:java.lang.String
变量 v 的类型是:java.lang.String
变量 s 的类型是:java.util.Date
```

以上例子首先定义了一个 Person 类的对象,并指明其参数类型为 Integer,然后在 Person 的 toString 方法中输出了传入的参数类型。然后定义另外一个 Person 类的对象,并指明其参数类型为 String。这里使用了菱形语法,即在构建泛型类的对象时不指定具体的

类型参数，而是由编译器根据上下文进行推断，在这个例子中，编译器根据变量 person1 的类型 Person＜String＞可以推断出类型参数的类为 String。

例子的最后试图把类型为 Person＜String＞的对象赋值给类型为 Person＜Integer＞的变量，编译器会报类型不匹配的语法错误，说明 Person＜String＞、Person＜Integer＞虽然都是来自于同一个泛型类，但是是不兼容的类型。

5.2.2 泛型方法

泛型方法是在方法上声明类型参数，它只可作用于声明它的方法上。

泛型方法中类型参数的定义与泛型类相同，都是通过＜ ＞中加入某一参数，如＜T＞来定义，但放的位置有所不同，以下代码定义了一个泛型方法。

```
01  package org.ddd.generic.example7;
02  public class Factory {
03      public <T> T generator(Class<T> t) throws Exception{
04          return t.newInstance();
05      }
06  }
```

下面来分析这段代码：首先是 public 后的＜T＞，作用是为该方法声明一个类型参数 T，声明该类型参数后，方法中就可以使用 T 作为一种类型使用了；接着是＜T＞后的 T，作用是声明方法的返回值类型，即参数 T 型；该方法的参数为 Class＜T＞ t，即 T 的类型信息；最后返回的是通过反射方法 newInstance()创建的 T 类的一个实例。

这个例子充分说明了反射与泛型联合使用时的强大功能，可以使用该方法创建任何需要的对象，这一点也正是泛型优点的体现。

泛型方法的使用与普通方法一样，你甚至感觉不出使用的是功能强大的泛型。实例代码如下。

```
01  package org.ddd.generic.example3_8;
02  public class GenericTest<T> {
03      public static void  main(String[] args) throws Exception{
04          Factory factory = new Factory();
05          Date date = factory.generator(Date.class);
06          System.out.println(date.toString());
07          Button button = factory.generator(Button.class);
08          System.out.println(button.toString());
09      }
10  }
```

输出结果如下。

```
01  Tue Aug 09 17:18:55 CST 2011
02  java.awt.Button[button0,0,0,0x0,invalid,slabel=]
```

代码简洁而优美、灵活而强大。可以使用该方法生成任何继承自 Object 类的实例，当然前提是该类有无参的构造方法。使用该方法时，你无须为那些烦琐的转型代码而烦恼，泛型系统确保了类型的正确性。

5.2.3 泛型接口

泛型除了可以作用在类和方法上，还可以应用于接口上，其定义如下。

```
01 package org.ddd.generic.example9;
02 public interface Factory<T> {
03     public T create();
04 }
```

泛型接口的定义与泛型类的定义相似，那么为什么还需要泛型接口呢？拿工厂的例子来说，不同工厂的生产方式各不一样，装载的零件也不相同，因此实现的方式也会各不相同，所以需要在具体的工厂中实现它独有的生产方式，代码如下。

```
01 package org.ddd.generic.example10;
02 public class Car {
03 }
04 public class Computer {
05 }
06 public class CarFactory implements Factory<Car> {
07     @Override
08     public Car create() {
09         System.out.println("装载发动机!");
10         System.out.println("装载座椅!");
11         System.out.println("装载轮子!");
12         return new Car();
13     }
14 }
15 public class ComputerFactory implements Factory<Computer> {
16     @Override
17     public Computer create() {
18         System.out.println("装载主板!");
19         System.out.println("装载 CPU!");
20         System.out.println("装载内存");
21         return new Computer();
22     }
23 }
24 public class GenericTest {
25     public static void  main(String[] args) throws Exception{
26         Factory<Car> carFactory = new CarFactory();
27         Factory<Computer> computerFactory = new ComputerFactory();
28         System.out.println("======开始生产汽车!=======");
```

```
        carFactory.create();
        System.out.println("=====开始生产电脑!========");
        computerFactory.create();
    }
}
```

输出结果如下。

```
======开始生产汽车!=======
装载发动机!
装载座椅!
装载轮子!
=====开始生产电脑!========
装载主板!
装载 CPU!
装载内存
```

以上例子实现了两个具体的工厂 CarFactory 和 ComputerFactory，它们都实现了泛型接口 Factory<T>，并在实现的时候指明了工厂所生成的具体类型，指明的方式只需将原有的类型参数 T 替换为具体的类型。如对 CarFactory 来说，将原有的 Factory<T>替换成 Factory<Car>。这样当覆盖接口中的方法 create 时，生成的产品就是 Car 类型了。在测试类中分别声明了 CarFactory 和 ComputerFactory，并使用这两个工厂分别生产了一件产品。

5.2.4 泛型与继承

泛型的继承容易给人一个误区，考虑下面的代码合法吗？

```
public class GenericTest {
    public static void  main(String[] args) throws Exception{
        Zoo<Animal> zoo = new Zoo<Animal>(new Animal());
        Zoo<Bird> birdZoo = new Zoo<Bird>(new Bird());
        zoo = birdZoo;
    }
}
```

直觉可能告诉你合法，因为 Animal 是 Bird 的父类，那么 Zoo<Animal>就是 Zoo<Bird>的父类。但事实并非如此，当试图将 birdZoo 赋值给 zoo 时，编译器抛出一个错误：Type mismatch：cannot convert from Zoo<Bird> to Zoo<Animal>，提示这样赋值不合法。其实 Zoo<Animal>与 Zoo<Bird>什么关系也没有，为什么要这样设计呢？是为了确保泛型的类型安全。假如编译器允许这样赋值，由于 Fish 也是 Animal 的子类，因此在 zoo 中添加一个 Fish 完全合理，但是这对 birdZoo 就不可接受了，因为你通过 zoo 的引用向 Zoo<Bird>中添加了 Fish 的实例。这样当运行获取该实例时，把它当作 Bird 来使用，显然是不合理的。

5.3 通配符

使用泛型实例时，需要为泛型指定具体的类型参数。如当使用 List 实例时，需要指明 List 中需要存放的类型 List<Integer>，然而有时泛型实例的作用域可能更加广泛，无法指明具体的参数类型。那么 Java 泛型机制是如何解决的呢？Java 设计者很聪明地设计了一种类型：通配符类型。它表示任何类型，通配符类型的符号是"?"，因此通配符类型可应用于所有继承自 Object 的类上。

5.3.1 通配符的使用


扫一扫

当无法确定泛型类的具体参数类型时，一般使用通配符类型代替，以下代码演示了如何使用通配符类型。

```
01  package org.ddd.generic.example11;
02  public class GenericTest<T> {
03      public static void  main(String[] args) throws Exception{
04          Class<?> clazz = Integer.class;
05          System.out.println(clazz.getCanonicalName());
06          clazz = String.class;
07          System.out.println(clazz.getCanonicalName());
08          clazz = Date.class;
09          System.out.println(clazz.getCanonicalName());
10      }
11  }
```

输出结果如下。

```
01  java.lang.Integer
02  java.lang.String
03  java.util.Date
```

以上例子在创建 Class 对象 clazz 时使用了通配符类型"?"，随后分别对其赋予 Integer.class、String.class、Date.class，这 3 个类型分别为 Class<Integer>、Class<String>以及 Class<Date>。可以看出使用通配符类型“?”后，clazz 对象就可以表示各种不同类型。使用通配符类型 Class<? >与原生类型 Class 的区别是：前者表明是因为暂时无法确定参数类型而使用了通配符类型，表示适合任何类型，而后者则可能是由于程序员疏忽或其他原因而没有指明参数类型，因此 Java 编译器会提出警告。

5.3.2 通配符的捕获

考虑下面的问题：现有一个方法用于交换 List 中的两个元素，由于事先不知道 List 中存放的元素类型，所以将参数设置成了含有通配符的实例，如 List<?> list。由于并不知