第 12 章　泛型程序设计

- ▲ 为什么要使用泛型程序设计
- ▲ 定义简单泛型类
- ▲ 泛型方法
- ▲ 类型变量的限定
- ▲ 泛型代码和虚拟机
- ▲ 约束与局限性
- ▲ 泛型类型的继承规则
- ▲ 通配符类型
- ▲ 反射和泛型

从 Java 程序设计语言 1.0 版发布以来，变化最大的部分就是泛型。致使 Java SE 5.0 中增加泛型机制的主要原因是为了满足在 1999 年制定的最早的 Java 规范需求之一（JSR 14）。专家组花费了 5 年左右的时间用来定义规范和测试实现。

泛型正是我们需要的程序设计手段。使用泛型机制编写的程序代码要比那些杂乱地使用 Object 变量，然后再进行强制类型转换的代码具有更好的安全性和可读性。泛型对于集合类尤其有用，例如，ArrayList 就是一个无处不在的集合类。

至少在表面上看来，泛型很像 C++ 中的模板。与 Java 一样，在 C++ 中，模板也是最先被添加到语言中支持强类型集合的。但是，多年之后人们发现模板还有其他的用武之地。学习完本章的内容可以发现 Java 中的泛型在程序中也是如此。

12.1　为什么要使用泛型程序设计

泛型程序设计（Generic programming）意味着编写的代码可以被很多不同类型的对象所重用。例如，我们并不希望为聚集 String 和 File 对象分别设计不同的类。实际上，也不需要这样做，因为一个 ArrayList 类可以聚集任何类型的对象。这是一个泛型程序设计的实例。

在 Java 中增加范型类之前，泛型程序设计是用继承实现的。ArrayList 类只维护一个 Object 引用的数组：

```
public class ArrayList // before generic classes
{
   private Object[] elementData;
   . . .
   public Object get(int i) { . . . }
   public void add(Object o) { . . . }
}
```

这样的实现有两个问题。当获取一个值时必须进行强制类型转换。

```
ArrayList files = new ArrayList();
. . .
String filename = (String) files.get(0);
```

51cto.com 拼吾爱 学习资源分享下载

此外，这里没有错误检查。可以向数组列表中添加任何类的对象。

```
files.add(new File(". . ."));
```

对于这个调用，编译和运行都不会出错。然而在其他地方，如果将 get 的结果强制类型转换为 String 类型，就会产生一个错误。

泛型提供了一个更好的解决方案：类型参数（type parameters）。ArrayList 类有一个类型参数用来指示元素的类型：

```
ArrayList<String> files = new ArrayList<String>();
```

这使得代码具有更好的可读性。人们一看就知道这个数组列表中包含的是 String 对象。

注释：前面已经提到，在 Java SE 7 及以后的版本中，构造函数中可以省略泛型类型：

```
ArrayList<String> files = new ArrayList<>();
```

省略的类型可以从变量的类型推断得出。

编译器也可以很好地利用这个信息。当调用 get 的时候，不需要进行强制类型转换，编译器就知道返回值类型为 String，而不是 Object：

```
String filename = files.get(0);
```

编译器还知道 ArrayList<String> 中 add 方法有一个类型为 String 的参数。这将比使用 Object 类型的参数安全一些。现在，编译器可以进行检查，避免插入错误类型的对象。例如：

```
files.add(new File(". . ."); // can only add String objects to an ArrayList<String>
```

是无法通过编译的。出现编译错误比类在运行时出现类的强制类型转换异常要好得多。类型参数的魅力在于：使得程序具有更好的可读性和安全性。

谁想成为泛型程序员？

使用像 ArrayList 的泛型类很容易。大多数 Java 程序员都使用 ArrayList<String> 这样的类型，就好像它们已经构建在语言之中，像 String[] 数组一样。（当然，数组列表比数组要好一些，因为它可以自动扩展。）

但是，实现一个泛型类并没有那么容易。对于类型参数，使用这段代码的程序员可能想要内置（plug in）所有的类。他们希望在没有过多的限制以及混乱的错误消息的状态下，做所有的事情。因此，一个泛型程序员的任务就是预测出所用类的未来可能有的所有用途。

这一任务难到什么程度呢？下面是标准类库的设计者们肯定产生争议的一个典型问题。ArrayList 类有一个方法 addAll 用来添加另一个集合的全部元素。程序员可能想要将 ArrayList<Manager> 中的所有元素添加到 ArrayList<Employee> 中去。然而，反过来就不行了。如果只能允许前一个调用，而不能允许后一个调用呢？ Java 语言的设计者发明了一个具有独创性的新概念，通配符类型（wildcard type），它解决了这个问题。通配符类型非常抽象，然而，它们能让库的构建者编写出尽可能灵活的方法。

泛型程序设计划分为 3 个能力级别。基本级别是，仅仅使用泛型类——典型的是像 ArrayList 这样的集合——不必考虑它们的工作方式与原因。大多数应用程序员将会停留在这

一级别上，直到出现了什么问题。当把不同的泛型类混合在一起时，或是在与对类型参数一无所知的遗留的代码进行衔接时，可能会看到含混不清的错误消息。如果这样的话，就需要学习 Java 泛型来系统地解决这些问题，而不要胡乱地猜测。当然，最终可能想要实现自己的泛型类与泛型方法。

应用程序员很可能不喜欢编写太多的泛型代码。JDK 开发人员已经做出了很大的努力，为所有的集合类提供了类型参数。凭经验来说，那些原本涉及许多来自通用类型（如 Object 或 Comparable 接口）的强制类型转换的代码一定会因使用类型参数而受益。

本章介绍实现自己的泛型代码需要了解的各种知识。希望大多数读者可以利用这些知识解决一些疑难问题，并满足对于参数化集合类的内部工作方式的好奇心。

12.2 定义简单泛型类

一个泛型类（generic class）就是具有一个或多个类型变量的类。本章使用一个简单的 Pair 类作为例子。对于这个类来说，我们只关注泛型，而不会为数据存储的细节烦恼。下面是 Pair 类的代码：

```
public class Pair<T>
{
   private T first;
   private T second;

   public Pair() { first = null; second = null; }
   public Pair(T first, T second) { this.first = first; this.second = second; }

   public T getFirst() { return first; }
   public T getSecond() { return second; }

   public void setFirst(T newValue) { first = newValue; }
   public void setSecond(T newValue) { second = newValue; }
}
```

Pair 类引入了一个类型变量 T，用尖括号（< >）括起来，并放在类名的后面。泛型类可以有多个类型变量。例如，可以定义 Pair 类，其中第一个域和第二个域使用不同的类型：

```
public class Pair<T, U> { . . . }
```

类定义中的类型变量指定方法的返回类型以及域和局部变量的类型。例如，

```
private T first; // uses the type variable
```

注释： 类型变量使用大写形式，且比较短，这是很常见的。在 Java 库中，使用变量 E 表示集合的元素类型，K 和 V 分别表示表的关键字与值的类型。T（需要时还可以用临近的字母 U 和 S）表示“任意类型”。

用具体的类型替换类型变量就可以实例化泛型类型，例如：

```
Pair<String>
```

pin51.com 拼吾爱 最新最全资源分享下载

可以将结果想象成带有构造器的普通类：

```
Pair<String>()
Pair<String>(String, String)
```

和方法：

```
String getFirst()
String getSecond()
void setFirst(String)
void setSecond(String)
```

换句话说，泛型类可看作普通类的工厂。

程序清单 12-1 中的程序使用了 Pair 类。静态的 minmax 方法遍历了数组并同时计算出最小值和最大值。它用一个 Pair 对象返回了两个结果。回想一下 compareTo 方法比较两个字符串，如果字符串相同则返回 0；如果按照字典顺序，第一个字符串比第二个字符串靠前，就返回负值，否则，返回正值。

C++ 注释： 从表面上看，Java 的泛型类类似于 C++ 的模板类。唯一明显的不同是 Java 没有专用的 template 关键字。但是，在本章中读者将会看到，这两种机制有着本质的区别。

程序清单 12-1 pair1/PairTestl.java

```
package pair1;

/**
 * @version 1.01 2012-01-26
 * @author Cay Horstmann
 */
public class PairTest1
{
   public static void main(String[] args)
   {
      String[] words = { "Mary", "had", "a", "little", "lamb" };
      Pair<String> mm = ArrayAlg.minmax(words);
      System.out.println("min = " + mm.getFirst());
      System.out.println("max = " + mm.getSecond());
   }
}

class ArrayAlg
{
   /**
    * Gets the minimum and maximum of an array of strings.
    * @param a an array of strings
    * @return a pair with the min and max value, or null if a is null or empty
    */
   public static Pair<String> minmax(String[] a)
   {
      if (a == null || a.length == 0) return null;
      String min = a[0];
      String max = a[0];
      for (int i = 1; i < a.length; i++)
```

```
31          {
32             if (min.compareTo(a[i]) > 0) min = a[i];
33             if (max.compareTo(a[i]) < 0) max = a[i];
34          }
35          return new Pair<>(min, max);
36       }
37    }
```

12.3 泛型方法

前面已经介绍了如何定义一个泛型类。实际上，还可以定义一个带有类型参数的简单方法。

```
class ArrayAlg
{
   public static <T> T getMiddle(T... a)
   {
      return a[a.length / 2];
   }
}
```

这个方法是在普通类中定义的，而不是在泛型类中定义的。然而，这是一个泛型方法，可以从尖括号和类型变量看出这一点。注意，类型变量放在修饰符（这里是 public static）的后面，返回类型的前面。

泛型方法可以定义在普通类中，也可以定义在泛型类中。

当调用一个泛型方法时，在方法名前的尖括号中放入具体的类型：

```
String middle = ArrayAlg.<String>getMiddle("John", "Q.", "Public");
```

在这种情况（实际也是大多数情况）下，方法调用中可以省略 <String> 类型参数。编译器有足够的信息能够推断出所调用的方法。它用 names 的类型（即 String[ ]）与泛型类型 T[ ] 进行匹配并推断出 T 一定是 String。也就是说，可以调用

```
String middle = ArrayAlg.getMiddle("John", "Q.", "Public");
```

几乎在大多数情况下，对于泛型方法的类型引用没有问题。偶尔，编译器也会提示错误，此时需要解译错误报告。看一看下面这个示例：

```
double middle = ArrayAlg.getMiddle(3.14, 1729, 0);
```

错误消息会以晦涩的方式指出（不同的编译器给出的错误消息可能有所不同）：解释这句代码有两种方法，而且这两种方法都是合法的。简单地说，编译器将会自动打包参数为 1 个 Double 和 2 个 Integer 对象，而后寻找这些类的共同超类型。事实上，找到 2 个这样的超类型：Number 和 Comparable 接口，其本身也是一个泛型类型。在这种情况下，可以采取的补救措施是将所有的参数写为 double 值。

提示： 如果想知道编译器对一个泛型方法调用最终推断出哪种类型，Peter von der Ah é 推荐了这样一个窍门：有目的地引入一个错误，并研究所产生的错误消息。例如，看一

下调用 ArrayAlg.getMiddle("Hello",0,null)。将结果赋给 JButton，这不可能正确。将会得到一个错误报告：

```
found:
java.lang.Object&java.io.Serializable&java.lang.Comparable<? extends
java.lang.Object&java.io.Serializable&java.lang.Comparable<?>>
```

大致的意思是：可以将结果赋给 Object、Serialiable 或 Comparable。

C++ 注释：在 C++ 中将类型参数放在方法名后面，有可能会导致语法分析的歧义。例如，g(f<a, b>(c)) 可以理解为"用 f<a, b>(c) 的结果调用 g"，或者理解为"用两个布尔值 f<a 和 b>(c) 调用 g"。

12.4 类型变量的限定

有时，类或方法需要对类型变量加以约束。下面是一个典型的例子。我们要计算数组中的最小元素：

```
class ArrayAlg
{
   public static <T> T min(T[] a) // almost correct
   {
      if (a == null || a.length == 0) return null;
      T smallest = a[0];
      for (int i = 1; i < a.length; i++)
         if (smallest.compareTo(a[i]) > 0) smallest = a[i];
      return smallest;
   }
}
```

但是，这里有一个问题。请看一下 min 方法的代码内部。变量 smallest 类型为 T，这意味着它可以是任何一个类的对象。怎么才能确信 T 所属的类有 compareTo 方法呢？

解决这个问题的方案是将 T 限制为实现了 Comparable 接口（只含一个方法 compareTo 的标准接口）的类。可以通过对类型变量 T 设置限定（bound）实现这一点：

```
public static <T extends Comparable> T min(T[] a) . . .
```

实际上 Comparable 接口本身就是一个泛型类型。目前，我们忽略其复杂性以及编译器产生的警告。第 12.8 节讨论了如何在 Comparable 接口中适当地使用类型参数。

现在，泛型的 min 方法只能被实现了 Comparable 接口的类（如 String、Date 等）的数组调用。由于 Rectangle 类没有实现 Comparable 接口，所以调用 min 将会产生一个编译错误。

C++ 注释：在 C++ 中不能对模板参数的类型加以限制。如果程序员用一个不适当的类型实例化一个模板，将会在模板代码中报告一个（通常是含糊不清的）错误消息。

读者或许会感到奇怪——在此为什么使用关键字 extends 而不是 implements？毕竟，Comparable 是一个接口。下面的符号

```
<T extends BoundingType>
```

表示 T 应该是绑定类型的子类型（subtype）。T 和绑定类型可以是类，也可以是接口。选择关键字 extends 的原因是更接近子类的概念，并且 Java 的设计者也不打算在语言中再添加一个新的关键字（如 sub）。

一个类型变量或通配符可以有多个限定，例如：

```
T extends Comparable & Serializable
```

限定类型用“&”分隔，而逗号用来分隔类型变量。

在 Java 的继承中，可以根据需要拥有多个接口超类型，但限定中至多有一个类。如果用一个类作为限定，它必须是限定列表中的第一个。

在程序清单 12-2 的程序中，重新编写了一个泛型方法 minmax。这个方法计算泛型数组的最大值和最小值，并返回 Pair<T>。

程序清单 12-2 pair2/PairTest2.java

```
1  package pair2;
2
3  import java.util.*;
4
5  /**
6   * @version 1.01 2012-01-26
7   * @author Cay Horstmann
8   */
9  public class PairTest2
10 {
11    public static void main(String[] args)
12    {
13       GregorianCalendar[] birthdays =
14          {
15             new GregorianCalendar(1906, Calendar.DECEMBER, 9), // G. Hopper
16             new GregorianCalendar(1815, Calendar.DECEMBER, 10), // A. Lovelace
17             new GregorianCalendar(1903, Calendar.DECEMBER, 3), // J. von Neumann
18             new GregorianCalendar(1910, Calendar.JUNE, 22), // K. Zuse
19          };
20       Pair<GregorianCalendar> mm = ArrayAlg.minmax(birthdays);
21       System.out.println("min = " + mm.getFirst().getTime());
22       System.out.println("max = " + mm.getSecond().getTime());
23    }
24 }
25
26 class ArrayAlg
27 {
28    /**
29       Gets the minimum and maximum of an array of objects of type T.
30       @param a an array of objects of type T
31       @return a pair with the min and max value, or null if a is
32       null or empty
33    */
34    public static <T extends Comparable> Pair<T> minmax(T[] a)
35    {
```

```
36        if (a == null || a.length == 0) return null;
37        T min = a[0];
38        T max = a[0];
39        for (int i = 1; i < a.length; i++)
40        {
41           if (min.compareTo(a[i]) > 0) min = a[i];
42           if (max.compareTo(a[i]) < 0) max = a[i];
43        }
44        return new Pair<>(min, max);
45     }
46  }
```

12.5 泛型代码和虚拟机

虚拟机没有泛型类型对象——所有对象都属于普通类。在泛型实现的早期版本中，甚至能够将使用泛型的程序编译为在 1.0 虚拟机上运行的类文件！这个向后兼容性在 Java 泛型开发的后期被放弃了。

无论何时定义一个泛型类型，都自动提供了一个相应的*原始类型*（raw type）。原始类型的名字就是删去类型参数后的泛型类型名。擦除（erased）类型变量，并替换为限定类型（无限定的变量用 Object）。

例如，Pair<T> 的原始类型如下所示：

```
public class Pair
{
   private Object first;
   private Object second;

   public Pair(Object first, Object second)
   {
      this.first = first;
      this.second = second;
   }

   public Object getFirst() { return first; }
   public Object getSecond() { return second; }

   public void setFirst(Object newValue) { first = newValue; }
   public void setSecond(Object newValue) { second = newValue; }
}
```

因为 T 是一个无限定的变量，所以直接用 Object 替换。

结果是一个普通的类，就好像泛型引入 Java 语言之前已经实现的那样。

在程序中可以包含不同类型的 Pair，例如，Pair<String> 或 Pair<GregorianCalendar>。而擦除类型后就变成原始的 Pair 类型了。

C++ 注释：就这点而言，Java 泛型与 C++ 模板有很大的区别。C++ 中每个模板的实例化产生不同的类型，这一现象称为“模板代码膨胀”。Java 不存在这个问题的困扰。

原始类型用第一个限定的类型变量来替换，如果没有给定限定就用 Object 替换。例如，类 Pair<T> 中的类型变量没有显式的限定，因此，原始类型用 Object 替换 T。假定声明了一个不同的类型。

```
public class Interval<T extends Comparable & Serializable> implements Serializable
{
   private T lower;
   private T upper;
   . . .
   public Interval(T first, T second)
   {
      if (first.compareTo(second) <= 0) { lower = first; upper = second; }
      else { lower = second; upper = first; }
   }
}
```

原始类型 Interval 如下所示：

```
public class Interval implements Serializable
{
   private Comparable lower;
   private Comparable upper;
   . . .
   public Interval(Comparable first, Comparable second) { . . . }
}
```

注释：读者可能想要知道切换限定：class Interval<Serializable & Comparable> 会发生什么。如果这样做，原始类型用 Serializable 替换 T，而编译器在必要时要向 Comparable 插入强制类型转换。为了提高效率，应该将标签（tagging）接口（即没有方法的接口）放在边界列表的末尾。

12.5.1 翻译泛型表达式

当程序调用泛型方法时，如果擦除返回类型，编译器插入强制类型转换。例如，下面这个语句序列

```
Pair<Employee> buddies = . . .;
Employee buddy = buddies.getFirst();
```

擦除 getFirst 的返回类型后将返回 Object 类型。编译器自动插入 Employee 的强制类型转换。也就是说，编译器把这个方法调用翻译为两条虚拟机指令：

- 对原始方法 Pair.getFirst 的调用。
- 将返回的 Object 类型强制转换为 Employee 类型。

当存取一个泛型域时也要插入强制类型转换。假设 Pair 类的 first 域和 second 域都是公有的（也许这不是一种好的编程风格，但在 Java 中是合法的）。表达式：

```
Employee buddy = buddies.first;
```

也会在结果字节码中插入强制类型转换。

12.5.2 翻译泛型方法

类型擦除也会出现在泛型方法中。程序员通常认为下述的泛型方法

```
public static <T extends Comparable> T min(T[] a)
```

是一个完整的方法族，而擦除类型之后，只剩下一个方法：

```
public static Comparable min(Comparable[] a)
```

注意，类型参数 T 已经被擦除了，只留下了限定类型 Comparable。

方法的擦除带来了两个复杂问题。看一看下面这个示例：

```
class DateInterval extends Pair<Date>
{
   public void setSecond(Date second)
   {
      if (second.compareTo(getFirst()) >= 0)
         super.setSecond(second);
   }
   . . .
}
```

一个日期区间是一对 Date 对象，并且需要覆盖这个方法来确保第二个值永远不小于第一个值。这个类擦除后变成

```
class DateInterval extends Pair // after erasure
{
   public void setSecond(Date second) { . . . }
   . . .
}
```

令人感到奇怪的是，存在另一个从 Pair 继承的 setSecond 方法，即

```
public void setSecond(Object second)
```

这显然是一个不同的方法，因为它有一个不同类型的参数——Object，而不是 Date。然而，不应该不一样。考虑下面的语句序列：

```
DateInterval interval = new DateInterval(. . .);
Pair<Date> pair = interval; // OK--assignment to superclass
pair.setSecond(aDate);
```

这里，希望对 setSecond 的调用具有多态性，并调用最合适的那个方法。由于 pair 引用 DateInterval 对象，所以应该调用 DateInterval.setSecond。问题在于类型擦除与多态发生了冲突。要解决这个问题，就需要编译器在 DateInterval 类中生成一个桥方法（bridge method）：

```
public void setSecond(Object second) { setSecond((Date) second); }
```

要想了解它的工作过程，请仔细地跟踪下列语句的执行：

```
pair.setSecond(aDate)
```

变量 pair 已经声明为类型 Pair<Date>，并且这个类型只有一个简单的方法叫 setSecond，即 setSecond(Object)。虚拟机用 pair 引用的对象调用这个方法。这个对象是 DateInterval 类型的，因而将会调用 DateInterval.setSecond(Object) 方法。这个方法是合成的桥方法。它调用

DateInterval.setSecond(Date)，这正是我们所期望的操作效果。

桥方法可能会变得十分奇怪。假设 DateInterval 方法也覆盖了 getSecond 方法：

```
class DateInterval extends Pair<Date>
{
   public Date getSecond() { return (Date) super.getSecond().clone(); }
   . . .
}
```

在擦除的类型中，有两个 getSecond 方法：

```
Date getSecond() // defined in DateInterval
Object getSecond() // overrides the method defined in Pair to call the first method
```

不能这样编写 Java 代码（在这里，具有相同参数类型的两个方法是不合法的）。它们都没有参数。但是，在虚拟机中，用参数类型和返回类型确定一个方法。因此，编译器可能产生两个仅返回类型不同的方法字节码，虚拟机能够正确地处理这一情况。

注释： 桥方法不仅用于泛型类型。第 5 章已经讲过，在一个方法覆盖另一个方法时可以指定一个更严格的返回类型。例如：

```
public class Employee implements Cloneable
{
   public Employee clone() throws CloneNotSupportedException { ... }
}
```

Object.clone 和 Employee.clone 方法被说成具有协变的返回类型（covariant return types）。

实际上，Employee 类有两个克隆方法：

```
Employee clone() // defined above
Object clone() // synthesized bridge method, overrides Object.clone
```

合成的桥方法调用了新定义的方法。

总之，需要记住有关 Java 泛型转换的事实：

- 虚拟机中没有泛型，只有普通的类和方法。
- 所有的类型参数都用它们的限定类型替换。
- 桥方法被合成来保持多态。
- 为保持类型安全性，必要时插入强制类型转换。

12.5.3 调用遗留代码

设计 Java 泛型类型时，主要目标是允许泛型代码和遗留代码之间能够互操作。

下面看一个具体的示例。要想设置一个 JSlider 标签，可以使用方法：

```
void setLabelTable(Dictionary table)
```

在这里，Dictionary 是一个原始类型，因为实现 JSlider 类时 Java 中还不存在泛型。不过，填充字典时，要使用泛型类型。

```
Dictionary<Integer, Component> labelTable = new Hashtable<>();
```

```
labelTable.put(0, new JLabel(new ImageIcon("nine.gif")));
labelTable.put(20, new JLabel(new ImageIcon("ten.gif")));
. . .
```

将 Dictionary<Integer, Component> 对象传递给 setLabelTable 时，编译器会发出一个警告。

```
slider.setLabelTable(labelTable); // WARNING
```

毕竟，编译器无法确定 setLabelTable 可能会对 Dictionary 对象做什么操作。这个方法可能会用字符串替换所有的关键字。这就打破了关键字类型为整型（Integer）的承诺，未来的操作有可能会产生强制类型转换的异常。

这个警告对操作不会产生什么影响，最多考虑一下 JSlider 有可能用 Dictionary 对象做什么就可以了。在这里十分清楚，JSlider 只阅读这个信息，因此可以忽略这个警告。

现在，看一个相反的情形，由一个遗留的类得到一个原始类型的对象。可以将它赋给一个参数化的类型变量，当然，这样做会看到一个警告。例如：

```
Dictionary<Integer, Components> labelTable = slider.getLabelTable(); // WARNING
```

这就行了。再看一看警告，确保标签表已经包含了 Integer 和 Component 对象。当然，从来也不会有绝对的承诺。恶意的编码者可能会在滑块中设置不同的 Dictionary。然而，这种情况并不会比有泛型之前的情况更糟糕。最差的情况就是程序抛出一个异常。

在查看了警告之后，可以利用注释（annotation）使之消失。注释必须放在生成这个警告的代码所在的方法之前，如下：

```
@SuppressWarnings("unchecked")
Dictionary<Integer, Components> labelTable = slider.getLabelTable(); // No warning
```

或者，可以标注整个方法，如下所示：

```
@SuppressWarnings("unchecked")
public void configureSlider() { . . . }
```

这个标注会关闭对方法中所有代码的检查。

12.6 约束与局限性

在下面几节中，将阐述使用 Java 泛型时需要考虑的一些限制。大多数限制都是由类型擦除引起的。

12.6.1 不能用基本类型实例化类型参数

不能用类型参数代替基本类型。因此，没有 Pair<double>，只有 Pair<Double>。当然，其原因是类型擦除。擦除之后，Pair 类含有 Object 类型的域，而 Object 不能存储 double 值。

这的确令人烦恼。但是，这样做与 Java 语言中基本类型的独立状态相一致。这并不是一个致命的缺陷——只有 8 种基本类型，当包装器类型（wrapper type）不能接受替换时，可以使用独立的类和方法处理它们。

pdfbook.com 拼吾爱 专业提供资源分享下载

12.6.2 运行时类型查询只适用于原始类型

虚拟机中的对象总有一个特定的非泛型类型。因此，所有的类型查询只产生原始类型。例如：

```
if (a instanceof Pair<String>) // ERROR
```

实际上仅仅测试 a 是否是任意类型的一个 Pair。下面的测试同样如此：

```
if (a instanceof Pair<T>) // ERROR
```

或强制类型转换：

```
Pair<String> p = (Pair<String>) a; // WARNING--can only test that a is a Pair
```

要记住这一风险，无论何时使用 instanceof 或涉及泛型类型的强制类型转换表达式都会看到一个编译器警告。

同样的道理，getClass 方法总是返回原始类型。例如：

```
Pair<String> stringPair = . . .;
Pair<Employee> employeePair = . . .;
if (stringPair.getClass() == employeePair.getClass()) // they are equal
```

其比较的结果是 true，这是因为两次调用 getClass 都将返回 Pair.class。

12.6.3 不能创建参数化类型的数组

不能实例化参数化类型的数组，例如：

```
Pair<String>[] table = new Pair<String>[10]; // ERROR
```

这有什么问题呢？擦除之后，table 的类型是 Pair[]。可以把它转换为 Object[]：

```
Object[] objarray = table;
```

数组会记住它的元素类型，如果试图存储其他类型的元素，就会抛出一个 ArrayStoreException 异常：

```
objarray[0] = "Hello"; // ERROR--component type is Pair
```

不过对于泛型类型，擦除会使这种机制无效。以下赋值：

```
objarray[0] = new Pair<Employee>();
```

能够通过数组存储检查，不过仍会导致一个类型错误。出于这个原因，不允许创建参数化类型的数组。

需要说明的是，只是不允许创建这些数组，而声明类型为 Pair<String>[] 的变量仍是合法的。不过不能用 new Pair<String>[10] 初始化这个变量。

注释： 可以声明通配类型的数组，然后进行类型转换：

```
Pair<String>[] table = (Pair<String>[]) new Pair<?>[10];
```

结果将是不安全的。如果在 table[0] 中存储一个 Pair<Employee>，然后对 table[0].getFirst() 调用一个 String 方法，会得到一个 ClassCastException 异常。

拼音爱 资源分享下载

提示： 如果需要收集参数化类型对象，只有一种安全而有效的方法：使用 ArrayList:ArrayList<Pair<String>>。

12.6.4 Varargs 警告

上一节中已经了解到，Java 不支持泛型类型的数组。这一节中我们再来讨论一个相关的问题：向参数个数可变的方法传递一个泛型类型的实例。

考虑下面这个简单的方法，它的参数个数是可变的：

```
public static <T> void addAll(Collection<T> coll, T... ts)
{
   for (t : ts) coll.add(t);
}
```

应该记得，实际上参数 ts 是一个数组，包含提供的所有实参。

现在考虑以下调用：

```
Collection<Pair<String>> table = ...;
Pair<String> pair1 = ...;
Pair<String> pair2 = ...;
addAll(table, pair1, pair2);
```

为了调用这个方法，Java 虚拟机必须建立一个 Pair<String> 数组，这就违反了前面的规则。不过，对于这种情况，规则有所放松，你只会得到一个警告，而不是错误。

可以采用两种方法来抑制这个警告。一种方法是为包含 addAll 调用的方法增加标注 @SuppressWarnings("unchecked")。或者在 Java SE 7 中，还可以用 @SafeVarargs 直接标注 addAll 方法：

```
@SafeVarargs
public static <T> void addAll(Collection<T> coll, T... ts)
```

现在就可以提供泛型类型来调用这个方法了。对于只需要读取参数数组元素的所有方法，都可以使用这个标注，这仅限于最常见的用例。

注释： 可以使用 @SafeVarargs 标注来消除创建泛型数组的有关限制，方法如下：

```
@SafeVarargs static <E> E[] array(E... array) { return array; }
```

现在可以调用：

```
Pair<String>[] table = array(pair1, pair2);
```

这看起来很方便，不过隐藏着危险。以下代码：

```
Object[] objarray = table;
objarray[0] = new Pair<Employe>();
```

能顺利运行而不会出现 ArrayStoreException 异常（因为数组存储只会检查擦除的类型），但在处理 table[0] 时你会在别处得到一个异常。

拼吾爱 最新最全资源分享下载

12.6.5 不能实例化类型变量

不能使用像 new T(...)，new T[...] 或 T.class 这样的表达式中的类型变量。例如，下面的 Pair<T> 构造器就是非法的：

```
public Pair() { first = new T(); second = new T(); } // ERROR
```

类型擦除将 T 改变成 Object，而且，本意肯定不希望调用 new Object( )。但是，可以通过反射调用 Class.newInstance 方法来构造泛型对象。

遗憾的是，细节有点复杂。不能调用：

```
first = T.class.newInstance(); // ERROR
```

表达式 T.class 是不合法的。必须像下面这样设计 API 以便可以支配 Class 对象：

```
public static <T> Pair<T> makePair(Class<T> cl)
{
   try { return new Pair<>(cl.newInstance(), cl.newInstance()) }
   catch (Exception ex) { return null; }
}
```

这个方法可以按照下列方式调用：

```
Pair<String> p = Pair.makePair(String.class);
```

注意，Class 类本身是泛型。例如，String.class 是一个 Class<String> 的实例（事实上，它是唯一的实例）。因此，makePair 方法能够推断出 pair 的类型。

不能构造一个泛型数组：

```
public static <T extends Comparable> T[] minmax(T[] a) { T[] mm = new T[2]; . . . } // ERROR
```

类型擦除会让这个方法永远构造 Object[2] 数组。

如果数组仅仅作为一个类的私有实例域，就可以将这个数组声明为 Object[]，并且在获取元素时进行类型转换。例如，ArrayList 类可以这样实现：

```
public class ArrayList<E>
{
   private Object[] elements;
   . . .
   @SuppressWarnings("unchecked") public E get(int n) { return (E) elements[n]; }
   public void set(int n, E e) { elements[n] = e; } // no cast needed
}
```

实际的实现没有这么清晰：

```
public class ArrayList<E>
{
   private E[] elements;
   . . .
   public ArrayList() { elements = (E[]) new Object[10]; }
}
```

这里，强制类型转换 E[ ] 是一个假象，而类型擦除使其无法察觉。

由于 minmax 方法返回 T[ ] 数组，使得这一技术无法施展，如果掩盖这个类型会有运行时错误结果。假设实现代码：

51.com 拼吾爱 最新 精品 资源 下载 站

```
public static <T extends Comparable> T[] minmax(T... a)
{
   Object[] mm = new Object[2];
   . . .
   return (T[]) mm; // compiles with warning
}
```

调用

```
String[] ss = minmax("Tom", "Dick", "Harry");
```

编译时不会有任何警告。当 Object[] 引用赋给 String[] 变量时，将会发生 ClassCastException 异常。

在这种情况下，可以利用反射，调用 Array.newInstance:

```
public static <T extends Comparable> T[] minmax(T... a)
{
   T[] mm = (T[]) Array.newInstance(a.getClass().getComponentType(), 2);
   . . .
}
```

ArrayList 类的 toArray 方法就没有这么幸运。它需要生成一个 T[] 数组，但没有成分类型。因此，有下面两种不同的形式：

```
Object[] toArray()
T[] toArray(T[] result)
```

第二个方法接收一个数组参数。如果数组足够大，就使用这个数组。否则，用 result 的成分类型构造一个足够大的新数组。

12.6.6 泛型类的静态上下文中类型变量无效

不能在静态域或方法中引用类型变量。例如，下列高招将无法施展：

```
public class Singleton<T>
{
   private static T singleInstance; // ERROR

   public static T getSingleInstance() // ERROR
   {
      if (singleInstance == null) construct new instance of T
      return singleInstance;
   }
}
```

如果这个程序能够运行，就可以声明一个 Singleton<Random> 共享随机数生成器，声明一个 Singleton<JFileChooser> 共享文件选择器对话框。但是，这个程序无法工作。类型擦除之后，只剩下 Singleton 类，它只包含一个 singleInstance 域。因此，禁止使用带有类型变量的静态域和方法。

win5i.com 拼吾爱 最新最全源分享下载站

12.6.7 不能抛出或捕获泛型类的实例

既不能抛出也不能捕获泛型类对象。实际上，甚至泛型类扩展 Throwable 都是不合法的。例如，以下定义就不能正常编译：

```
public class Problem<T> extends Exception { /* . . . */ } // ERROR--can't extend Throwable
```

catch 子句中不能使用类型变量。例如，以下方法将不能编译：

```
public static <T extends Throwable> void doWork(Class<T> t)
{
   try
   {
      do work
   }
   catch (T e) // ERROR--can't catch type variable
   {
      Logger.global.info(...)
   }
}
```

不过，在异常规范中使用类型变量是允许的。以下方法是合法的：

```
public static <T extends Throwable> void doWork(T t) throws T // OK
{
   try
   {
      do work
   }
   catch (Throwable realCause)
   {
      t.initCause(realCause);
      throw t;
   }
}
```

可以消除对已检查异常的检查

Java 异常处理的一个基本原则是，必须为所有已检查异常提供一个处理器。不过可以利用泛型消除这个限制。关键在于以下方法：

```
@SuppressWarnings("unchecked")
public static <T extends Throwable> void throwAs(Throwable e) throws T
{
   throw (T) e;
}
```

假设这个方法包含在类 Block 中，如果调用

```
Block.<RuntimeException>throwAs(t);
```

编译器就会认为 t 是一个未检查的异常。以下代码会把所有异常都转换为编译器所认为的未检查的异常：

```
try
{
   do work
```

pin51.com 拼吾爱 最新编程资源

```
}
catch (Throwable t)
{
   Block.<RuntimeException>throwAs(t);
}
```

下面把这个代码包装在一个抽象类中。用户可以覆盖 body 方法来提供一个具体的动作。调用 toThread 时，会得到 Thread 类的一个对象，它的 run 方法不会介意已检查的异常。

```
public abstract class Block
{
   public abstract void body() throws Exception;

   public Thread toThread()
   {
      return new Thread()
         {
            public void run()
            {
               try
               {
                  body();
               }
               catch (Throwable t)
               {
                  Block.<RuntimeException>throwAs(t);
               }
            }
         };
   }

   @SuppressWarnings("unchecked")
   public static <T extends Throwable> void throwAs(Throwable e) throws T
   {
        throw (T) e;
   }
}
```

例如，以下程序运行了一个线程，它会抛出一个已检查的异常。

```
public class Test
{
   public static void main(String[] args)
   {
      new Block()
         {
            public void body() throws Exception
            {
               Scanner in = new Scanner(new File("ququx"));
               while (in.hasNext())
                  System.out.println(in.next());
            }
         }
      .toThread().start();
   }
}
```

pin5i.com 拼吾爱 最新最全下载站

运行这个程序时，会得到一个栈轨迹，其中包含一个 FileNotFoundException（当然，假设你没有提供一个名为 ququx 的文件）。

这有什么意义呢？正常情况下，你必须捕获线程 run 方法中的所有已检查的异常，把它们“包装”到未检查的异常中，因为 run 方法声明为不抛出任何已检查异常。

不过在这里并没有做这种“包装”。我们只是抛出异常，并“哄骗”编译器，让它认为这不是一个已检查异常。

通过使用泛型类、擦除和 @SuppressWarnings 标注，就能消除 Java 类型系统的部分基本限制。

12.6.8 注意擦除后的冲突

当泛型类型被擦除时，无法创建引发冲突的条件。下面是一个示例。假定像下面这样将 equals 方法添加到 Pair 类中：

```
public class Pair<T>
{
   public boolean equals(T value) { return first.equals(value) && second.equals(value); }
   . . .
}
```

考虑一个 Pair<String>。从概念上讲，它有两个 equals 方法：

```
boolean equals(String) // defined in Pair<T>
boolean equals(Object) // inherited from Object
```

但是，直觉把我们引入歧途。方法擦除

```
boolean equals(T)
```

就是

```
boolean equals(Object)
```

与 Object.equals 方法发生冲突。

当然，补救的办法是重新命名引发错误的方法。

泛型规范说明还提到另外一个原则：“要想支持擦除的转换，就需要强行限制一个类或类型变量不能同时成为两个接口类型的子类，而这两个接口是同一接口的不同参数化。”例如，下述代码是非法的：

```
class Calendar implements Comparable<Calendar> { . . . }
class GregorianCalendar extends Calendar implements Comparable<GregorianCalendar>
   { . . . } // ERROR
```

GregorianCalendar 会实现 Comparable<Calendar> 和 Comparable<GregorianCalendar >，这是同一接口的不同参数化。

这一限制与类型擦除的关系并不十分明确。毕竟，下列非泛型版本是合法的。

```
class Calendar implements Comparable { . . . }
class GregorianCalendar extends Calendar implements Comparable { . . . }
```

其原因非常微妙，有可能与合成的桥方法产生冲突。实现了 Comparable<X> 的类可以获得一个桥方法：

51cto.com 技术成就梦想 版权所有

```
public int compareTo(Object other) { return compareTo((X) other); }
```

对于不同类型的 X 不能有两个这样的方法。

12.7 泛型类型的继承规则

在使用泛型类时，需要了解一些有关继承和子类型的准则。下面先从许多程序员感觉不太直观的情况开始。考虑一个类和一个子类，如 Employee 和 Manager。Pair<Manager> 是 Pair<Employee> 的一个子类吗？答案是“不是”，或许人们会感到奇怪。例如，下面的代码将不能编译成功：

```
Manager[] topHonchos = . . .;
Pair<Employee> result = ArrayAlg.minmax(topHonchos); // ERROR
```

minmax 方法返回 Pair<Manager>，而不是 Pair<Employee>，并且这样的赋值是不合法的。无论 S 与 T 有什么联系（如图 12-1 所示），通常，Pair<S> 与 Pair<T> 没有什么联系。

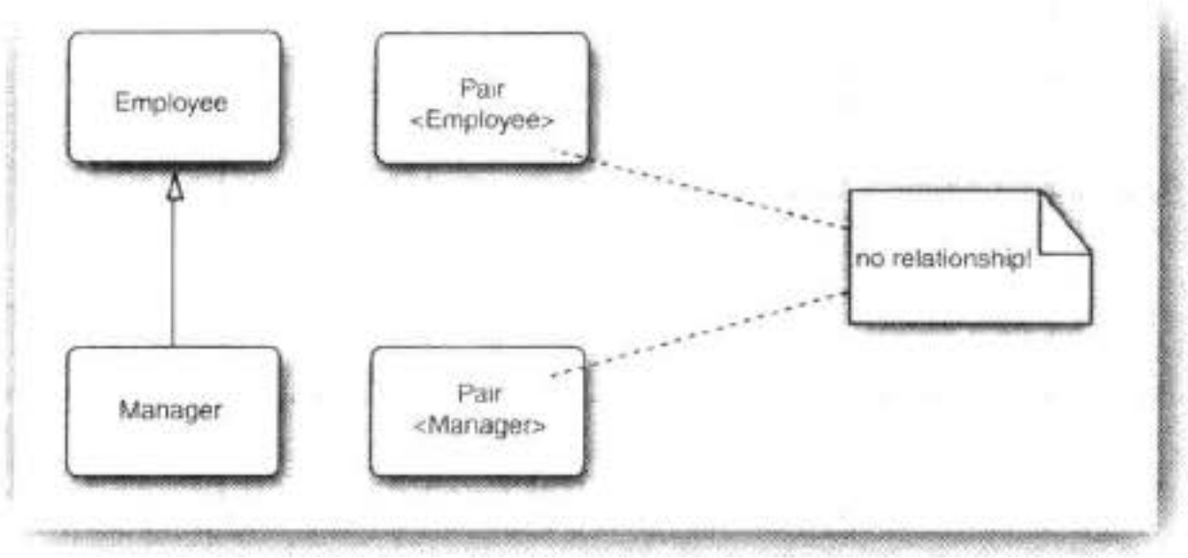

图 12-1 pair 类之间没有继承关系

这一限制看起来过于严格，但对于类型安全非常必要。假设允许将 Pair<Manager> 转换为 Pair<Employee>。考虑下面代码：

```
Pair<Manager> managerBuddies = new Pair<>(ceo, cfo);
Pair<Employee> employeeBuddies = managerBuddies; // illegal, but suppose it wasn't
employeeBuddies.setFirst(lowlyEmployee);
```

显然，最后一句是合法的。但是 employeeBuddies 和 managerBuddies 引用了同样的对象。现在将 CFO 和一个普通员工组成一对，这对于 Pair<Manager> 来说应该是不可能的。

注释： 必须注意泛型与 Java 数组之间的重要区别。可以将一个 Manager[] 数组赋给一个类型为 Employee[] 的变量：

```
Manager[] managerBuddies = { ceo, cfo };
Employee[] employeeBuddies = managerBuddies; // OK
```

然而，数组带有特别的保护。如果试图将一个低级别的雇员存储到 employeeBuddies[0]，虚拟机将会抛出 ArrayStoreException 异常。

永远可以将参数化类型转换为一个原始类型。例如，Pair<Employee> 是原始类型 Pair 的一个子类型。在与遗留代码衔接时，这个转换非常必要。

转换成原始类型之后会产生类型错误吗？很遗憾，会！看一看下面这个示例：

```
Pair<Manager> managerBuddies = new Pair<>(ceo, cfo);
Pair rawBuddies = managerBuddies; // OK
rawBuddies.setFirst(new File(". . ."); // only a compile-time warning
```

听起来有点吓人。但是，请记住现在的状况不会再比旧版 Java 的情况糟糕。虚拟机的安全性还没有到生死攸关的程度。当使用 getFirst 获得外来对象并赋给 Manager 变量时，与通常一样，会抛出 ClassCastException 异常。这里失去的只是泛型程序设计提供的附加安全性。

最后，泛型类可以扩展或实现其他的泛型类。就这一点而言，与普通的类没有什么区别。例如，ArrayList<T> 类实现 List<T> 接口。这意味着，一个 ArrayList<Manager> 可以被转换为一个 List<Manager>。但是，如前面所见，一个 ArrayList<Manager> 不是一个 ArrayList <Employee> 或 List<Employee>。图 12-2 展示了它们之间的联系。

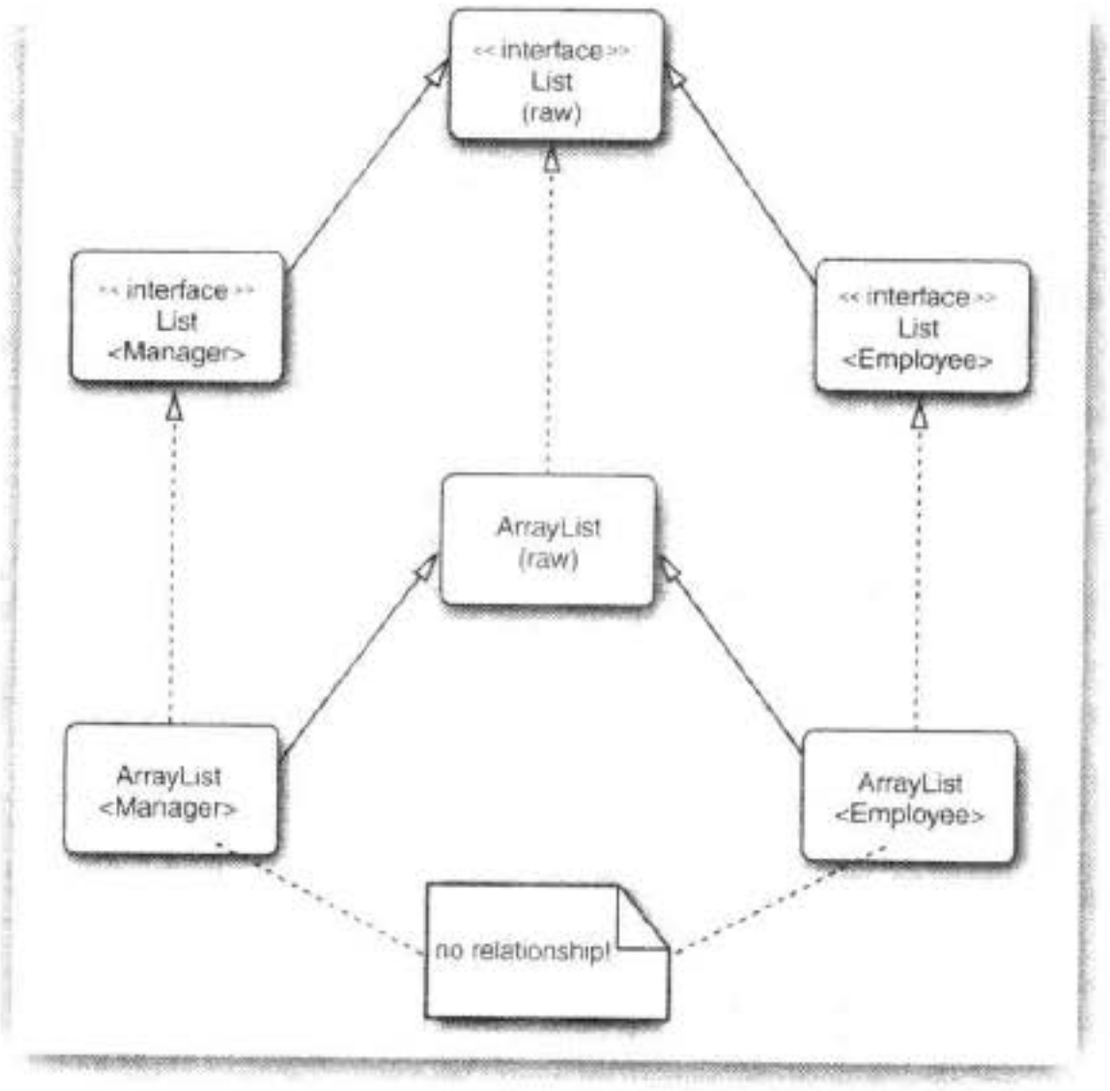

图 12-2 泛型列表类型中子类型间的联系

12.8 通配符类型

固定的泛型类型系统使用起来并没有那么令人愉快，类型系统的研究人员知道这一点已经有一段时间了。Java 的设计者发明了一种巧妙的（仍然是安全的）“解决方案”：通配符类型。例如，通配符类型

```
Pair<? extends Employee>
```

表示任何泛型 Pair 类型，它的类型参数是 Employee 的子类，如 Pair<Manager>，但不是 Pair<String>。

假设要编写一个打印雇员对的方法，像这样：

```
public static void printBuddies(Pair<Employee> p)
{
   Employee first = p.getFirst();
   Employee second = p.getSecond();
   System.out.println(first.getName() + " and " + second.getName() + " are buddies.");
}
```

正如前面讲到的，不能将 Pair<Manager> 传递给这个方法，这一点很受限制。解决的方法很简单：使用通配符类型：

```
public static void printBuddies(Pair<? extends Employee> p)
```

类型 Pair<Manager> 是 Pair<? extends Employee> 的子类型（如图 12-3 所示）。

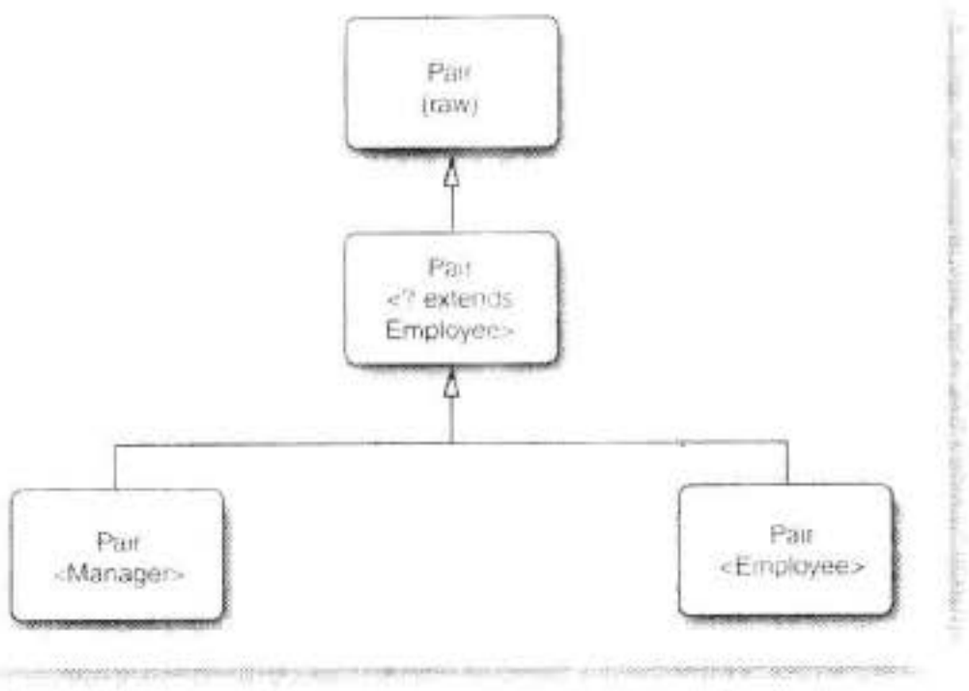

图 12-3 使用通配符的子类型的联系

使用通配符会通过 Pair<? extends Employee> 的引用破坏 Pair<Manager> 吗？

```
Pair<Manager> managerBuddies = new Pair<>(ceo, cfo);
Pair<? extends Employee> wildcardBuddies = managerBuddies; // OK
wildcardBuddies.setFirst(lowlyEmployee); // compile-time error
```

这可能不会引起破坏。对 setFirst 的调用有一个类型错误。要了解其中的缘由，请仔细看一看类型 Pair<? extends Employee>。其方法似乎是这样的：

```
? extends Employee getFirst()
void setFirst(? extends Employee)
```

这样将不可能调用 setFirst 方法。编译器只知道需要某个 Employee 的子类型，但不知道具体是什么类型。它拒绝传递任何特定的类型。毕竟？不能用来匹配。

使用 getFirst 就不存在这个问题：将 getFirst 的返回值赋给一个 Employee 的引用完全合法。

这就是引入有限定的通配符的关键之处。现在已经有办法区分安全的访问器方法和不安

全的更改器方法了。

12.8.1 通配符的超类型限定

通配符限定与类型变量限定十分类似，但是，还有一个附加的能力，即可以指定一个超类型限定（supertype bound），如下所示：

```
? super Manager
```

这个通配符限制为 Manager 的所有超类型。（已有的 super 关键字十分准确地描述了这种联系，这一点十分令人感到欣慰。）

为什么要这样做呢？带有超类型限定的通配符的行为与 12.8 节介绍的相反。可以为方法提供参数，但不能使用返回值。例如，Pair<? super Manager> 有方法

```
void setFirst(? super Manager)
? super Manager getFirst()
```

编译器不知道 setFirst 方法的确切类型，但是可以用任意 Manager 对象（或子类型，例如，Executive）调用它，而不能用 Employee 对象调用。然而，如果调用 getFirst，返回的对象类型就不会得到保证。只能把它赋给一个 Object。

下面是一个典型的示例。有一个经理的数组，并且想把奖金最高和最低的经理放在一个 Pair 对象中。Pair 的类型是什么？在这里，Pair<Employee> 是合理的，Pair<Object> 也是合理的（如图 12-4 所示）。下面的方法将可以接受任何适当的 Pair：

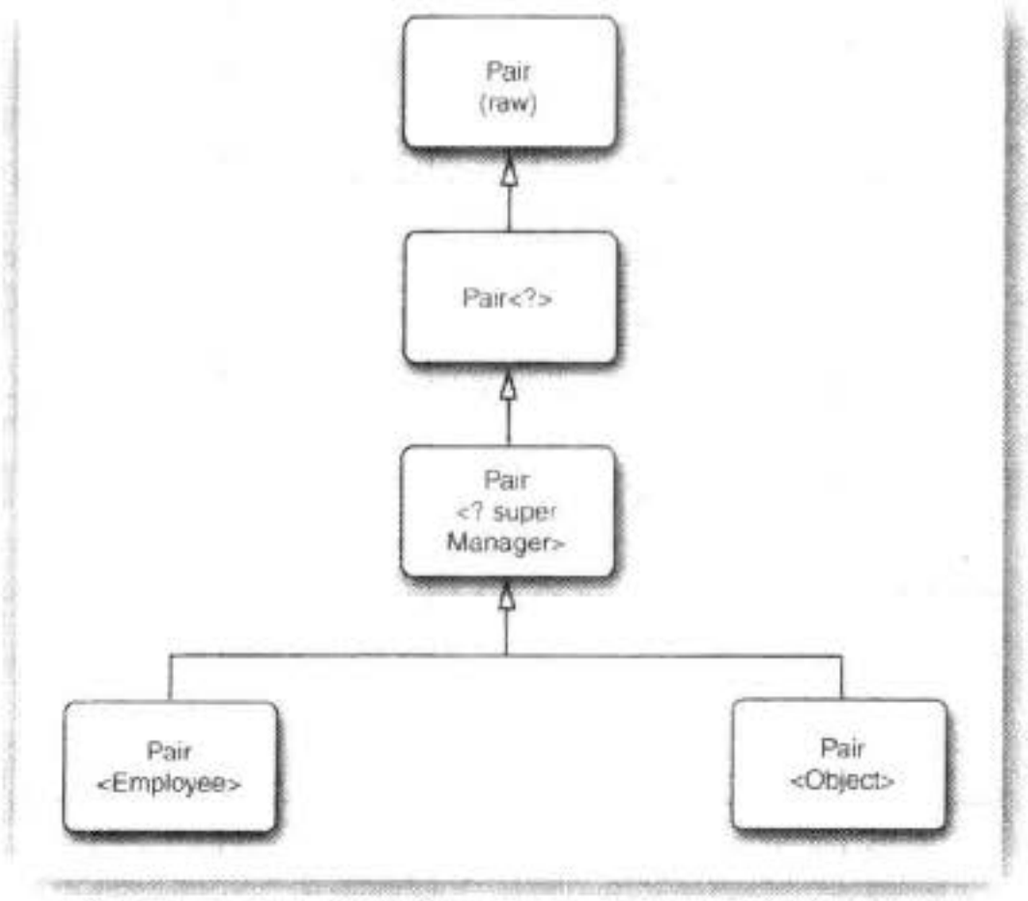

图 12-4 带有超类型边界的通配符

```
public static void minmaxBonus(Manager[] a, Pair<? super Manager> result)
{
   if (a == null || a.length == 0) return;
   Manager min = a[0];
   Manager max = a[0];
```

```
        for (int i = 1; i < a.length; i++)
        {
           if (min.getBonus() > a[i].getBonus()) min = a[i];
           if (max.getBonus() < a[i].getBonus()) max = a[i];
        }
        result.setFirst(min);
        result.setSecond(max);
     }
```

直观地讲，带有超类型限定的通配符可以向泛型对象写入，带有子类型限定的通配符可以从泛型对象读取。

下面是超类型限定的另一种应用。Comparable 接口本身就是一个泛型类型。声明如下：

```
public interface Comparable<T>
{
   public int compareTo(T other);
}
```

在此，类型变量指示了 other 参数的类型。例如，String 类实现 Comparable <String>，它的 compareTo 方法被声明为

```
public int compareTo(String other)
```

很好，显式的参数有一个正确的类型。在 Java SE 5.0 之前，other 是一个 Object，并且这个方法的实现需要强制类型转换。

由于 Comparable 是一个泛型类型，也许可以把 ArrayAlg 类的 min 方法做得更好一些？可以这样声明：

```
public static <T extends Comparable<T>> T min(T[] a)
```

看起来，这样写比只使用 T extents Comparable 更彻底，并且对许多类来讲，工作得更好。例如，如果计算一个 String 数组的最小值，T 就是 String 类型的，而 String 是 Comparable<String> 的子类型。但是，当处理一个 GregorianCalendar 对象的数组时，就会出现问题。Gregorian Calendar 是 Calendar 的子类，并且 Calendar 实现了 Comparable<Calendar>。因此 Gregorian Calendar 实现的是 Comparable<Calendar>，而不是 Comparable<GregorianCalendar>。

在这种情况下，超类型可以用来进行救助：

```
public static <T extends Comparable<? super T>> T min(T[] a) . . .
```

现在 compareTo 方法写成

```
int compareTo(? super T)
```

有可能被声明为使用类型 T 的对象，也有可能使用 T 的超类型（如当 T 是 Gregorian Calendar）。无论如何，传递一个 T 类型的对象给 compareTo 方法都是安全的。

对于初学者来说，<T extends Comparable<? super T>> 这样的声明看起来有点吓人。很遗憾，因为这一声明的意图在于帮助应用程序员排除调用参数上的不必要的限制。对泛型没有兴趣的应用程序员很可能很快就学会掩盖这些声明，想当然地认为库程序员做的都是正确的。如果是一名库程序员，一定要习惯于通配符，否则，就会受到用户的责备，还要在代码中随意地添加强制类型转换直至代码可以编译。

12.8.2 无限定通配符

还可以使用无限定的通配符，例如，Pair<?>。初看起来，这好像与原始的 Pair 类型一样。实际上，有很大的不同。类型 Pair<?> 有方法如下所示：

```
? getFirst()
void setFirst(?)
```

getFirst 的返回值只能赋给一个 Object。setFirst 方法不能被调用，甚至不能用 Object 调用。Pair<?> 和 Pair 本质的不同在于：可以用任意 Object 对象调用原始的 Pair 类的 setObject 方法。

注释： 可以调用 setFirst(null)。

为什么要使用这样脆弱的类型？它对于许多简单的操作非常有用。例如，下面这个方法将用来测试一个 pair 是否包含一个 null 引用，它不需要实际的类型。

```
public static boolean hasNulls(Pair<?> p)
{
   return p.getFirst() == null || p.getSecond() == null;
}
```

通过将 hasNulls 转换成泛型方法，可以避免使用通配符类型：

```
public static <T> boolean hasNulls(Pair<T> p)
```

但是，带有通配符的版本可读性更强。

12.8.3 通配符捕获

编写一个交换一个 pair 元素的方法：

```
public static void swap(Pair<?> p)
```

通配符不是类型变量，因此，不能在编写代码中使用“？”作为一种类型。也就是说，下述代码是非法的：

```
? t = p.getFirst(); // ERROR
p.setFirst(p.getSecond());
p.setSecond(t);
```

这是一个问题，因为在交换的时候必须临时保存第一个元素。幸运的是，这个问题有一个有趣的解决方案。我们可以写一个辅助方法 swapHelper，如下所示：

```
public static <T> void swapHelper(Pair<T> p)
{
   T t = p.getFirst();
   p.setFirst(p.getSecond());
   p.setSecond(t);
}
```

注意，swapHelper 是一个泛型方法，而 swap 不是，它具有固定的 Pair<?> 类型的参数。

现在可以由 swap 调用 swapHelper：

www.minxue51.com 精品最新书籍

```
public static void swap(Pair<?> p) { swapHelper(p); }
```

在这种情况下，swapHelper 方法的参数 T 捕获通配符。它不知道是哪种类型的通配符，但是，这是一个明确的类型，并且 <T>swapHelper 的定义只有在 T 指出类型时才有明确的含义。

当然，在这种情况下，并不是一定要使用通配符。我们已经直接实现了没有通配符的泛型方法 <T> void swap(Pair<T> p)。然而，下面看一个通配符类型出现在计算中间的示例：

```
public static void maxminBonus(Manager[] a, Pair<? super Manager> result)
{
   minmaxBonus(a, result);
   PairAlg.swap(result); // OK--swapHelper captures wildcard type
}
```

在这里，通配符捕获机制是不可避免的。

通配符捕获只有在有许多限制的情况下才是合法的。编译器必须能够确信通配符表达的是单个、确定的类型。例如，ArrayList<Pair<T>> 中的 T 永远不能捕获 ArrayList<Pair<?>> 中的通配符。数组列表可以保存两个 Pair<?>，分别针对 ? 的不同类型。

程序清单 12-3 中的测试程序将前几节讨论的各种方法综合在一起，读者从中可以看到它们彼此之间的关联。

程序清单 12-3 pair3/PairTest3.java

```
1  package pair3;
2
3  /**
4   * @version 1.01 2012-01-26
5   * @author Cay Horstmann
6   */
7  public class PairTest3
8  {
9     public static void main(String[] args)
10    {
11       Manager ceo = new Manager("Gus Greedy", 800000, 2003, 12, 15);
12       Manager cfo = new Manager("Sid Sneaky", 600000, 2003, 12, 15);
13       Pair<Manager> buddies = new Pair<>(ceo, cfo);
14       printBuddies(buddies);
15
16       ceo.setBonus(1000000);
17       cfo.setBonus(500000);
18       Manager[] managers = { ceo, cfo };
19
20       Pair<Employee> result = new Pair<>();
21       minmaxBonus(managers, result);
22       System.out.println("first: " + result.getFirst().getName()
23          + ", second: " + result.getSecond().getName());
24       maxminBonus(managers, result);
25       System.out.println("first: " + result.getFirst().getName()
26          + ", second: " + result.getSecond().getName());
27    }
28
29    public static void printBuddies(Pair<? extends Employee> p)
```

```
   {
      Employee first = p.getFirst();
      Employee second = p.getSecond();
      System.out.println(first.getName() + " and " + second.getName() + " are buddies.");
   }

   public static void minmaxBonus(Manager[] a, Pair<? super Manager> result)
   {
      if (a == null || a.length == 0) return;
      Manager min = a[0];
      Manager max = a[0];
      for (int i = 1; i < a.length; i++)
      {
         if (min.getBonus() > a[i].getBonus()) min = a[i];
         if (max.getBonus() < a[i].getBonus()) max = a[i];
      }
      result.setFirst(min);
      result.setSecond(max);
   }
   public static void maxminBonus(Manager[] a, Pair<? super Manager> result)
   {
      minmaxBonus(a, result);
      PairAlg.swapHelper(result); // OK--swapHelper captures wildcard type
   }
}

class PairAlg
{
   public static boolean hasNulls(Pair<?> p)
   {
      return p.getFirst() == null || p.getSecond() == null;
   }

   public static void swap(Pair<?> p) { swapHelper(p); }

   public static <T> void swapHelper(Pair<T> p)
   {
      T t = p.getFirst();
      p.setFirst(p.getSecond());
      p.setSecond(t);
   }
}
```

12.9 反射和泛型

现在，Class 类是泛型的。例如，String.class 实际上是一个 Class<String> 类的对象（事实上，是唯一的对象）。

类型参数十分有用，这是因为它允许 Class<T> 方法的返回类型更加具有针对性。下面 Class<T> 中的方法就使用了类型参数：

```
T newInstance()
T cast(Object obj)
T[] getEnumConstants()
Class<? super T> getSuperclass()
Constructor<T> getConstructor(Class... parameterTypes)
Constructor<T> getDeclaredConstructor(Class... parameterTypes)
```

newInstance 方法返回一个实例，这个实例所属的类由默认的构造器获得。它的返回类型目前被声明为 T，其类型与 Class<T> 描述的类相同，这样就免除了类型转换。

如果给定的类型确实是 T 的一个子类型，cast 方法就会返回一个现在声明为类型 T 的对象，否则，抛出一个 BadCastException 异常。

如果这个类不是 enum 类或类型 T 的枚举值的数组，getEnumConstants 方法将返回 null。

最后，getConstructor 与 getdeclaredConstructor 方法返回一个 Constructor<T> 对象。Constructor 类也已经变成泛型，以便 newInstance 方法有一个正确的返回类型。

API java.lang.Class<T> 1.0

- `T newInstance() 5.0`

  返回默认构造器构造的一个新实例。
- `T cast(Object obj) 5.0`

  如果 obj 为 null 或有可能转换成类型 T，则返回 obj；否则抛出 BadCastException 异常。
- `T[ ] getEnumConstants( ) 5.0`

  如果 T 是枚举类型，则返回所有值组成的数组，否则返回 null。
- `Class<? super T> getSuperclass( ) 5.0`

  返回这个类的超类。如果 T 不是一个类或 Object 类，则返回 null。
- `Constructor<T> getConstructor(Class. . . parameterTypes) 5.0`
- `Constructor<T> getDeclaredConstructor(Class. . . parameterTypes) 5.0`

  获得公有的构造器，或带有给定参数类型的构造器。

API java.lang.reflect.Constructor<T> 1.1

- `T newInstance(Object... parameters) 5.0`

  返回用指定参数构造的新实例。

12.9.1 使用 Class<T> 参数进行类型匹配

有时，匹配泛型方法中的 Class<T> 参数的类型变量很有实用价值。下面是一个具有一定权威的示例：

```
public static <T> Pair<T> makePair(Class<T> c) throws InstantiationException,
      IllegalAccessException
{
   return new Pair<>(c.newInstance(), c.newInstance());
}
```

pin51.com 拼吾爱 最新最全资源下载站

如果调用

```
makePair(Employee.class)
```

Employee.class 是类型 Class<Employee> 的一个对象。makePair 方法的类型参数 T 同 Employee 匹配，并且编译器可以推断出这个方法将返回一个 Pair<Employee>。

12.9.2 虚拟机中的泛型类型信息

Java 泛型的卓越特性之一是在虚拟机中泛型类型的擦除。令人感到奇怪的是，擦除的类仍然保留一些泛型祖先的微弱记忆。例如，原始的 Pair 类知道源于泛型类 Pair<T>，即使一个 Pair 类型的对象无法区分是由 Pair<String> 构造的还是由 Pair<Employee> 构造的。

类似地，看一下方法

```
public static Comparable min(Comparable[] a)
```

这是一个泛型方法的擦除

```
public static <T extends Comparable<? super T>> T min(T[] a)
```

可以使用反射 API 来确定：

- 这个泛型方法有一个叫做 T 的类型参数。
- 这个类型参数有一个子类型限定，其自身又是一个泛型类型。
- 这个限定类型有一个通配符参数。
- 这个通配符参数有一个超类型限定。
- 这个泛型方法有一个泛型数组参数。

换句话说，需要重新构造实现者声明的泛型类以及方法中的所有内容。但是，不会知道对于特定的对象或方法调用，如何解释类型参数。

注释： 包含在类文件中，让泛型反射可用的类型信息与旧的虚拟机不兼容。

为了表达泛型类型声明，Java SE 5.0 在 java.lang.reflect 包中提供了一个新的接口 Type。这个接口包含下列子类型：

- Class 类，描述具体类型。
- TypeVariable 接口，描述类型变量（如 T extends Comparable<? super T>）。
- WildcardType 接口，描述通配符（如 ? super T）。
- ParameterizedType 接口，描述泛型类或接口类型（如 Comparable<? super T>）。
- GenericArrayType 接口，描述泛型数组（如 T[ ]）。

图 12-5 给出了继承层次。注意，最后 4 个子类型是接口，虚拟机将实例化实现这些接口的适当的类。

程序清单 12-4 中使用泛型反射 API 打印出给定类的有关内容。如果用 Pair 类运行，将会得到下列报告：

```
class Pair<T> extends java.lang.Object
public T getFirst()
public T getSecond()
```

www.51.com 拼吾爱 最新最全资源分享下载

```
public void setFirst(T)
public void setSecond(T)
```

如果使用 PairTest2 目录下的 ArrayAlg 运行，将会得到下列报告：

```
public static <T extends java.lang.Comparable> Pair<T> minmax(T[])
```

本节末尾的 API 注释描述了示例程序中使用的这些方法。

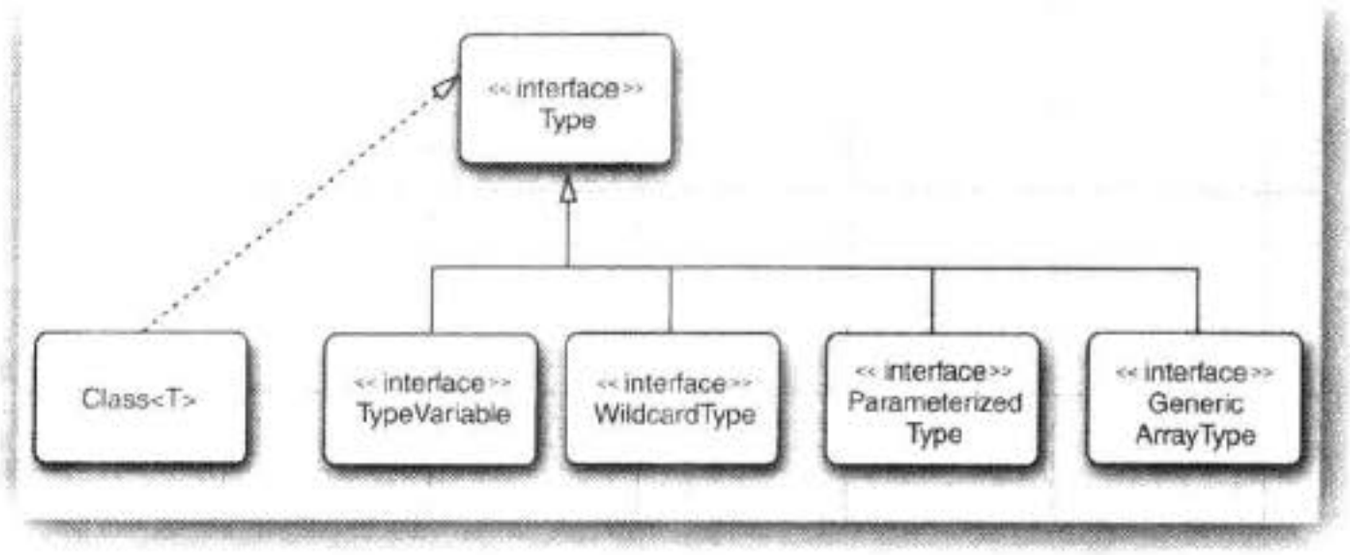

图 12-5 Type 类和它的后代

程序清单 12-4 genericReflection/GenericReflectionTest.java

```
1  package genericReflection;
2
3  import java.lang.reflect.*;
4  import java.util.*;
5  /**
6   * @version 1.10 2007-05-15
7   * @author Cay Horstmann
8   */
9  public class GenericReflectionTest
10 {
11    public static void main(String[] args)
12    {
13       // read class name from command line args or user input
14       String name;
15       if (args.length > 0) name = args[0];
16       else
17       {
18          Scanner in = new Scanner(System.in);
19          System.out.println("Enter class name (e.g. java.util.Collections): ");
20          name = in.next();
21       }
22
23       try
24       {
25          // print generic info for class and public methods
26          Class<?> cl = Class.forName(name);
27          printClass(cl);
28          for (Method m : cl.getDeclaredMethods())
29             printMethod(m);
30       }
```

```java
      catch (ClassNotFoundException e)
      {
         e.printStackTrace();
      }
   }

   public static void printClass(Class<?> cl)
   {
      System.out.print(cl);
      printTypes(cl.getTypeParameters(), "<", ", ", ">", true);
      Type sc = cl.getGenericSuperclass();
      if (sc != null)
      {
         System.out.print(" extends ");
         printType(sc, false);
      }
      printTypes(cl.getGenericInterfaces(), " implements ", ", ", "", false);
      System.out.println();
   }
   public static void printMethod(Method m)
   {
      String name = m.getName();
      System.out.print(Modifier.toString(m.getModifiers()));
      System.out.print(" ");
      printTypes(m.getTypeParameters(), "<", ", ", "> ", true);

      printType(m.getGenericReturnType(), false);
      System.out.print(" ");
      System.out.print(name);
      System.out.print("(");
      printTypes(m.getGenericParameterTypes(), "", ", ", "", false);
      System.out.println(")");
   }

   public static void printTypes(Type[] types, String pre, String sep, String suf,
         boolean isDefinition)
   {
      if (pre.equals(" extends ") && Arrays.equals(types, new Type[] { Object.class })) return;
      if (types.length > 0) System.out.print(pre);
      for (int i = 0; i < types.length; i++)
      {
         if (i > 0) System.out.print(sep);
         printType(types[i], isDefinition);
      }
      if (types.length > 0) System.out.print(suf);
   }

   public static void printType(Type type, boolean isDefinition)
   {
      if (type instanceof Class)
      {
         Class<?> t = (Class<?>) type;
         System.out.print(t.getName());
      }
```

```
      else if (type instanceof TypeVariable)
      {
         TypeVariable<?> t = (TypeVariable<?>) type;
         System.out.print(t.getName());
         if (isDefinition)
            printTypes(t.getBounds(), " extends ", " & ", "", false);
      }
      else if (type instanceof WildcardType)
      {
         WildcardType t = (WildcardType) type;
         System.out.print("?");
         printTypes(t.getUpperBounds(), " extends ", " & ", "", false);
         printTypes(t.getLowerBounds(), " super ", " & ", "", false);
      }
      else if (type instanceof ParameterizedType)
      {
         ParameterizedType t = (ParameterizedType) type;
         Type owner = t.getOwnerType();
         if (owner != null)
         {
            printType(owner, false);
            System.out.print(".");
         }
         printType(t.getRawType(), false);
         printTypes(t.getActualTypeArguments(), "<", ", ", ">", false);
      }
      else if (type instanceof GenericArrayType)
      {
         GenericArrayType t = (GenericArrayType) type;
         System.out.print("");
         printType(t.getGenericComponentType(), isDefinition);
         System.out.print("[]");
      }
   }
}
```

API java.lang.Class<T> 1.0

- `TypeVariable[ ] getTypeParameters( ) 5.0`

  如果这个类型被声明为泛型类型，则获得泛型类型变量，否则获得一个长度为 0 的数组。

- `Type getGenericSuperclass( ) 5.0`

  获得被声明为这一类型的超类的泛型类型；如果这个类型是 Object 或不是一个类类型（class type），则返回 null。

- `Type[ ] getGenericInterfaces( ) 5.0`

  获得被声明为这个类型的接口的泛型类型（以声明的次序），否则，如果这个类型没有实现接口，返回长度为 0 的数组。

API java.lang.reflect.Method 1.1

- `TypeVariable[ ] getTypeParameters( ) 5.0`

如果这个方法被声明为泛型方法，则获得泛型类型变量，否则返回长度为 0 的数组。

- `Type getGenericReturnType( ) 5.0`

  获得这个方法被声明的泛型返回类型。

- `Type[ ] getGenericParameterTypes( ) 5.0`

  获得这个方法被声明的泛型参数类型。如果这个方法没有参数，返回长度为 0 的数组。

API Java.lang.reflect.TypeVariable 5.0

- `String getName( )`

  获得类型变量的名字。

- `Type[ ] getBounds( )`

  获得类型变量的子类限定，否则，如果该变量无限定，则返回长度为 0 的数组。

API Java.lang.reflect.WildcardType 5.0

- `Type[ ] getUpperBounds( )`

  获得这个类型变量的子类（extends）限定，否则，如果没有子类限定，则返回长度为 0 的数组。

- `Type[ ] getLowerBounds( )`

  获得这个类型变量的超类（super）限定，否则，如果没有超类限定，则返回长度为 0 的数组。

API Java.lang.reflect.ParameterizedType 5.0

- `Type getRawType( )`

  获得这个参数化类型的原始类型。

- `Type[ ] getActualTypeArguments( )`

  获得这个参数化类型声明时所使用的类型参数。

- `Type getOwnerType( )`

  如果是内部类型，则返回其外部类型，如果是一个顶级类型，则返回 null。

API Java.lang.reflect.GenericArrayType 5.0

- `Type getGenericComponentType( )`

  获得声明该数组类型的泛型组合类型。

现在已经学习了如何使用泛型类以及在必要时如何自定义泛型类和泛型方法。同样重要的是，学习了如何解译在 API 文档和错误消息中遇到的泛型类型声明。要想了解有关 Java 泛型更加详尽的信息，可以到 http://angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html 上求助。那里有一个很好的常见问题解答列表（也有一些不太常见的）。

在下一章中，将学习 Java 集合框架如何使用泛型。

拼吾爱 最新网站 下载站 分享 book.51.com