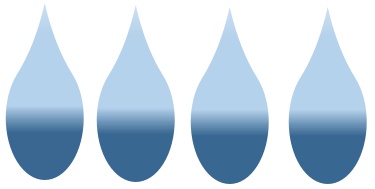
# 第12章 泛型

## 12.1 泛型的概念

### 12.1.1 没有泛型的麻烦

举例：下面有四个瓶子，准备用来装“醋”、“酱油”、“黄酒”、“花椒油”，如果瓶子没有标签的话，那么当这些调料一旦装进去之后，我们就容易忘记哪个瓶子里装了什么，每次用的时候，需要“闻”或“尝”等确认一下才能用，否则就容易加错调料。



生产瓶子的厂家不知道我们将来用这个瓶子装什么，因此我们为了后面的“方便”、“安全”，可以在装完调料之后贴“标签”。

同理，在前一章的时候，我们使用集合时，一旦把一个对象“丢进”Java集合中，集合就会忘记对象的类型，把所有的对象当成Object类型处理。当程序从集合中取出对象后，就需要进行强制类型转换，这种强制类型转换不仅代码臃肿，而且容易引起ClassCastException异常。

代码示例：

|  |
| --- |
| **package** com.atguigu.part01;  **import** java.util.ArrayList;  **import** java.util.List;  **public** **class** TestListErr {  **public** **static** **void** main(String[] args) {  //strList集合，本来只想装字符串对象  List strList = **new** ArrayList();  strList.add("佟刚");  strList.add("宋红康");  strList.add("尚硅谷");  //“不小心”把Integer对象装进去了  strList.add(666);    **for** (**int** i=0; i<strList.size(); i++) {  //因为strList中是按照Object处理的，所以必须强制类型转换  //最后一个元素将出现ClassCastException  String str = (String) strList.get(i);  System.***out***.println( str + "的长度：" + str.length());  }  }  } |

Java集合之所以被设计成这样，是因为设计集合的程序员不会知道我们需要用它来装什么类型的对象，所以他们把集合设计成能保存任何类型的对象，只要求具有很好的通用性。但这样做也带来了两个问题：

集合对元素类型没有任何限制，这样可能引发一些问题：例如上面的示例中，只想存储字符串对象，却不小心把Integer对象轻易的放进去，因为编译期间没有类型检查。

由于把对象“丢进”集合时，在编译期间，集合就忘记了对象的实际类型，集合只知道它盛装的是Object，因此取出集合元素后，该对象的编译时类型就变成了Object类型（其实际的运行时类型没变），如果要使用还需要强制类型转换。这种强制类型转换既会增加编程的复杂度，也可能引发ClassCastException。

### 12.1.2 泛型的引入

如何解决呢？生活的智慧往往可以启迪我们，我们可以给集合“贴标签”。

那么如何实现“贴标签”呢？

回忆一下：我们要实现求两个整数的最大值，如何设计方法的？

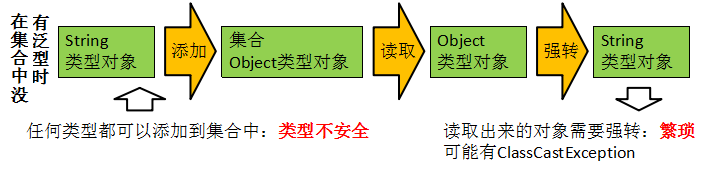
|  |
| --- |
| **public** **static** **int** max(**int** a, **int** b) {  **return** a > b ? a : b;  } |
| **public** **static** **void** main(String[] args) {  int max = *max*(3,6);  System.***out***.println(max);  } |

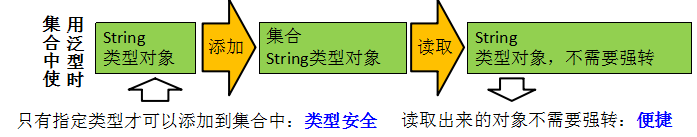
这里我们设计了两个形参a,b，因为我们无法确定“两个整数”的值，我们就用形参a,b来代替未知的整数的值来完成该方法的实现。当调用方法时，再确定a,b的实际值，我们称为实参，例如3,6就是给a,b赋值的实参。

同样的JDK1.5改写了集合框架中的全部接口和类，为这些接口、类增加了**“类型形参”，**这个类型形参将在声明变量、创建对象时确定，即传入实际的类型，我么称为**“类型实参”**。我们把这个“参数化的类型”称为泛型（Generic）。

代码示例：

|  |
| --- |
| **public** **class** ArrayList<E> //省略.....{  **public** **boolean** add(E e) {  //.....省略  }  **public** E get(**int** index) {  //....省略  }  } |
| **package** com.atguigu.part01;  **import** java.util.ArrayList;  **public** **class** TestList {  **public** **static** **void** main(String[] args) {  // strList集合，本来只想装字符串对象  ArrayList<String> strList = **new** ArrayList<String>();  strList.add("佟刚");  strList.add("宋红康");  strList.add("尚硅谷");  // 编译器将阻止我们把Integer对象装进去  strList.add(666);  **for** (**int** i = 0; i < strList.size(); i++) {  // 因为strList中都是String，所以不需要强制类型转换  String str = strList.get(i);  System.***out***.println(str + "的长度：" + str.length());  }  }  } |





那么，ArrayList<E>的<E>就是“类型形参”，ArrayList<String>的<String>就是“类型实参”，String类型就是用来确定E的类型用的。

## 12.2 定义泛型类、接口

提示：为了区别，我们可以将**int** max(**int** a, **int** b)中a,b称为**数据形参**，将 int max = *max*(3,6);中3,6称为**数据实参**。

**泛型实参必须是引用数据类型，不能是基本数据类型。**

下面是JDK1.5改写后ArrayList接口、Iterator接口、Map的代码片段：

|  |
| --- |
| **public** **class** ArrayList<E> //省略.....{  **public** **boolean** add(E e) {  //.....省略  }  **public** E get(**int** index) {  //....省略  }  **public** Iterator<E> iterator(){  //....省略  }  } |
| **public** **class** ArrayList<E> {  **boolean** hasNext();  E next();  } |
| **public** **interface** Map<K,V> {  V get(Object key);  V put(K key, V value);  Set<K> keySet();  } |

从上面的代码中，我们可以看出可以在定义接口、类时指定类型形参，例如：E，K，V。当我们使用这些集合时，就可以为E，K，V指定具体的类型实参。

代码示例：

|  |
| --- |
| **package** com.atguigu.part01;  **import** java.util.HashMap;  **import** java.util.Iterator;  **import** java.util.Set;  **public** **class** TestHashMap {  **public** **static** **void** main(String[] args) {  HashMap<Integer,String> map = **new** HashMap<Integer,String>();  map.put(1, "尚硅谷");  map.put(2, "atguigu");  Set<Integer> keySet = map.keySet();  Iterator<Integer> iter = keySet.iterator();  **while**(iter.hasNext()){  Integer key = iter.next();  System.***out***.println(key + "->" + map.get(key));  }  }  } |

### 12.2.1 定义泛型类、接口

* 在定义类和接口的时候，都可以指定泛型，并不是只有集合类才可以使用泛型声明。
* 见名知意：泛型形参的命名一般使用单个的大写字母，如果有多个类型形参，那么使用逗号分隔，例如：Map<K,V>。

常见字母：

* T：Type
* K，V：Key，Value
* E：Element

|  |
| --- |
| **package** com.atguigu.part02;  **public** **class** TestStudent {  **public** **static** **void** main(String[] args) {  Student<Integer> s1 = **new** Student<Integer>("张三",95);  Student<String> s2 = **new** Student<String>("张三","优秀");  Student<Double> s3 = **new** Student<Double>("张三",80.5);  }  }  **class** Student<T>{  **private** String name;  **private** T score;    **public** Student() {  **super**();  }  **public** Student(String name, T score) {  **super**();  **this**.name = name;  **this**.score = score;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** T getScore() {  **return** score;  }  **public** **void** setScore(T score) {  **this**.score = score;  }  @Override  **public** String toString() {  **return** "姓名：" + name + ", 成绩：" + score;  }  } |

**结论：**

**泛型类、接口的泛型可以当做正常类型用在类内部的任何地方：**

属性类型、方法的形参类型、方法返回值类型等。

不能用于静态变量、静态方法、静态代码块中。

因为静态成员的初始化是随着类的初始化而初始化的，此时泛型实参无法指定，那么泛型形参的类型就不确定。所以请不要在静态成员上使用类或接口上的泛型形参。

提示：当创建带泛型声明的类时，为该类定义构造器时，构造器名还是原来的类名，不需要增加泛型声明。例如：Student<T>类定义的构造器依然是Student()，而不是Student<T>，但调用构造器时缺可以使用Student<T>的形式，而且应该为T类型形参传入实际的类型实参。

### 12.2.2 指定泛型实参

#### 1、泛型实参的要求

* **首先泛型实参必须是引用数据类型，不能是基本数据类型。**

|  |
| --- |
| //如下指定为int是错误的  ArrayList<**int**> list = **new** ArrayList<**int**>();    //必须是引用数据类型，因此可以指定为包装类型  ArrayList<Integer> list = **new** ArrayList<Integer>(); |

#### 2、什么时候指定泛型实参？

**（1）在用泛型类、接口声明变量时**

|  |
| --- |
| **class** EmployeeManager{  **private** ArrayList<Employee> list;  } |
| **public** **static** **void** test(ArrayList<String> list){  //....省略代码  } |

**（2）在继承泛型类或实现泛型接口时，如果子类不延续使用该泛型，那么必须明确指定实际类型。此时子类不再是泛型类了。**

|  |
| --- |
| //继承泛型类  //此处ArrayList<String>就不能写成ArrayList<E>  //因为MyArrayList不再是泛型类了，因此E必须给出具体的类型  **class** MyArrayList **extends** ArrayList<String>{    }  //实现泛型接口  **class** Employee **implements** Comparable<Employee>{  @Override  **public** **int** compareTo(Employee o) {  **return** 0;  }  } |

**（3）在创建泛型类对象时**

|  |
| --- |
| ArrayList<String> list = **new** ArrayList<String>(); |

JDK1.7之后支持如下简化写法：

|  |
| --- |
| ArrayList<String> list = **new** ArrayList<>(); |

### 12.2.3 延续使用父类、父接口的泛型形参

提示，如果继承泛型类、实现泛型接口时，想要继续保留父类、父接口的泛型，必须在父类、父接口和子类、子接口中都要保留泛型。

代码示例：

|  |
| --- |
| //SubArrayList<E>和ArrayList<E>保持一样的泛型字母即可  **class** SubArrayList<E> **extends** ArrayList<E>{  //在子类中泛型形参还是E  } |
| //SubArrayList<T>和ArrayList<T>保持一样的泛型字母即可  **class** SubArrayList<T> **extends** ArrayList<T>{  //在子类中泛型形参是T，父类中使用泛型形参的位置也自动修改为T  } |

集合中大量是这种情况：

|  |
| --- |
| **public** **interface** Iterable<T> {  //....省略了代码  } |
| **public** **interface** Collection<E> **extends** Iterable<E> {  //....省略了代码  } |
| **public** **interface** List<E> **extends** Collection<E> {  //....省略了代码  } |
| **public** **class** ArrayList<E> **extends** AbstractList<E>  **implements** List<E>, RandomAccess, Cloneable, java.io.Serializable{  //...省略了代码  } |

### 12.2.4 泛型形参的上限

案例需求：定义学生类，其中的学生成绩可以是如下数字类型：

* Integer
* Float
* Double
* .....

|  |
| --- |
| **class** Student<T **extends** Number> {  **private** String name;  **private** T score;  **public** Student() {  **super**();  }  **public** Student(String name, T score) {  **super**();  **this**.name = name;  **this**.score = score;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** T getScore() {  **return** score;  }  **public** **void** setScore(T score) {  **this**.score = score;  }  @Override  **public** String toString() {  **return** "姓名：" + name + ", 成绩：" + score;  }  } |

* **如果泛型形参没有设定上限，那么泛型实参可以是任意引用数据类型。如果泛型形参设定了上限（例如：T extends 父类上限），那么只能指定为该父类本身或其各子类类型。**

代码示例：

|  |
| --- |
| **public** **class** TestStudentUpperBound {  **public** **static** **void** main(String[] args) {  Student<Integer> s1 = **new** Student<Integer>("张三", 95);  Student<Double> s3 = **new** Student<Double>("张三", 80.5);  //以下代码编译报错，因为String不是Number的子类  Student<String> s2 = **new** Student<String>("张三", "优秀");  }  } |

* **在一种更极端的情况下，程序需要为形参设定多个上限（至多有一个父类上限，可以有多个接口上限）表明该类型形参必须是其父类的子类（包括是父类本身也行），并且实现多个上限接口。**

代码示例：

|  |
| --- |
| **class** Student<T **extends** Number & java.io.Serializable> {  //...省略其他代码  } |

* **为类型形参指定多个上限时，所有的接口上限必须位于类上限之后。**
* **如果没有指定泛型上限，就按照Object作为上限**
* **泛型擦除：在创建泛型类的实例对象的时候，没有指定泛型实参**
  + **按照上限类型进行处理**
  + **多个上限，按照上限类进行处理**

## 12.3 定义泛型方法

* **泛型方法的使用场景：**

前面介绍了在定义类、接口时可以使用类型形参，在该类的方法和属性定义、接口的方法定义中，这些类型形参可被当成普通类型来用。

在另外一些情况下：

1. **如果我们定义类、接口时没有使用类型形参，但是某个方法定义时，想要自己定义类型形参；**
2. **另外我们之前说类和接口上的类型形参是不能用于静态方法中，那么当某个静态方法想要定义类型形参。**

在JDK1.5之后，还提供了泛型方法的支持。

### 12.3.1 声明泛型方法

例如：假设需要实现这样的一个方法，该方法负责将一个数组的所有元素添加到一个Collection集合中，考虑采用如下代码来实现该方法：

|  |
| --- |
| **public** **static** **void** fromArrayToCollection(Object[] a,Collection<Object> c){  **for** (Object object : a) {  c.add(object);  }  } |

* **注意：Collection<Object>不是Collection<String>类的父类，形参c只支持Collection<Object>类型的实参。**

|  |
| --- |
| String[] array = {"hello","world","java"};  Collection<String> coll = **new** ArrayList<String>();  //编译报错，因为Collection<Object>形参不接收Collection<String>实参，因为它俩不是父子类关系  *fromArrayToCollection*(array,coll); |

为了解决这个问题，可以使用泛型方法：

|  |
| --- |
| **public** **static** <T> **void** fromArrayToCollection(T[] a,Collection<T> c){  **for** (T object : a) {  c.add(object);  }  } |
| **public** **static** **void** main(String[] args) {  String[] array = {"hello","world","java"};  Collection<String> coll = **new** ArrayList<String>();  *fromArrayToCollection*(array,coll);  } |

泛型方法的定义：

* 泛型方法的方法签名比普通方法的方法签名多了类型形参声明，类型形参声明以尖括号括起来。
* 泛型形参列表中有多个，使用逗号分隔，和定义泛型类、接口时一样。
* 而且<泛型形参列表>必须在修饰符和返回值类型之间。

|  |
| --- |
| [访问限定符符] [修饰符列表]  **<泛型形参列表>**  返回类型 方法名([形参列表]) [抛出的异常列表]{  //方法体...  } |

泛型方法使用说明：

* 泛型方法声明中定义的泛型形参只能在当前方法中使用，和其他方法无关。
* 方法声明中定义的泛型形参无需显式传入实际类型参数，编译器可以根据实参类型直接推断形参的实际类型。

### 12.3.2 设定泛型形参的上限

例如：我们有几个图形类，代码示例：

|  |
| --- |
| **package** com.atguigu.part02**;**  **public abstract** **class** Graphic{  **public** **abstract** **double** getArea();  } |
| **package** com.atguigu.part02**;**  **public class** Circle **extends** Graphic{  **private** **double** radius;  **public** Circle(**double** radius) {  **super**();  **this**.radius = radius;  }  @Override  **public** **double** getArea() {  **return** Math.***PI*** \* radius \* radius;  }  } |
| **package** com.atguigu.part02**;**  **public class** Rectangle **extends** Graphic{  **private** **double** length;  **private** **double** width;  **public** Rectangle(**double** length, **double** width) {  **super**();  **this**.length = length;  **this**.width = width;  }  @Override  **public** **double** getArea() {  **return** length \* width;  }  } |

接下来我们想定义一个方法，可以求多个图形的面积。

代码示例：

|  |
| --- |
| **package** com.atguigu.part02;  **import** java.util.ArrayList;  **import** java.util.List;  **public** **class** TestGraphic {  **public** **static** <T **extends** Graphic> **void** printArea(List<T> graphics){  **for** (T t : graphics) {  System.***out***.println(t.getArea());  }  }  **public** **static** **void** main(String[] args) {  ArrayList<Circle> cList = **new** ArrayList<Circle>();  cList.add(**new** Circle(1.2));  cList.add(**new** Circle(2.3));  *printArea*(cList);    ArrayList<Rectangle> rList = **new** ArrayList<Rectangle>();  rList.add(**new** Rectangle(1,2));  rList.add(**new** Rectangle(2,3));  *printArea*(rList);  }  } |

提示：其实没有设定泛型形参上限的，可以看成它的上限默认是Object。

## 12.4 类型通配符

当我们声明一个方法时，某个形参的类型是一个泛型类或泛型接口类型，但是在声明方法时，又不确定该泛型实际类型，我们可以考虑使用类型通配符。

### 12.4.1 类型通配符

**以遍历集合来说明类型通配符的作用：**

* **1.集合作为参数，不指定集合的泛型:**

|  |
| --- |
| **public** **static** **void** test(List c){  **for** (**int** i = 0; i < c.size(); i++) {  System.***out***.println(c.get(i));  }  } |

代码缺点：程序没有问题，但是使用List接口时没有传入实际类型参数，这将引起泛型警告。如何消除这个警告呢？

* **方式一：普通方法，泛型参数**

|  |
| --- |
| **public** **static** **void** test1(List<Object> c){  **for** (**int** i = 0; i < c.size(); i++) {  System.***out***.println(c.get(i));  }  } |

但是这样的形参太局限了，只能传入List<Object>类型的实参。

* **方式二：泛型方法，泛型参数**

|  |
| --- |
| **public** **static** <T> **void** test2(List<T> c){  **for** (**int** i = 0; i < c.size(); i++) {  System.***out***.println(c.get(i));  }  } |

* **方式三：普通方法，类型通配符参数**

|  |
| --- |
| **public** **static** **void** test3(List<?> c){  **for** (**int** i = 0; i < c.size(); i++) {  System.***out***.println(c.get(i));  }  } |

* **方式二和方式三的区别：**

test3方法带通配符的List仅表示它可以接受指定了任意泛型实参的List，并不能把元素加入其中，例如如下代码将会引起编译错误**：**

|  |
| --- |
| **public** **static** **void** test(List<?> c, String str){  c.add(str);  } |

因为我们不知道上面程序中c集合里元素的类型，所以不能向其中添加对象，除了null对象，因为它是所有引用数据类型的实例。

test2方法带泛型的List，表示该集合的元素类型是T，因此允许T系列的对象加入其中，例如如下代码是可行的：

|  |
| --- |
| **public** **static** <T> **void** test(List<T> c, T t){  c.add(t);  } |

结论：当不涉及添加元素到带泛型的集合中，那么普通泛型和泛型通配符都可以，如果涉及到添加元素到带泛型的集合中，使用类型通配符<?>的不支持。

### 12.4.2 类型通配符的上限

当直接使用List<?>这种形式时，即表明这个List集合接收泛型实参指定为任意类型的List。但有时候我们不想这样，只希望接收某些类型的List。

例如：一个图形的抽象父类Graphic，两个子类Circle和Rectangle，代码见$12.3.2示例。接下来我们想定义一个方法，可以打印不同图形的面积。

代码示例：

|  |
| --- |
| **public** **static** **void** printArea(List<Graphic> graphics){  **for** (Graphic g : graphics) {  System.***out***.println(g.getArea());  }  } |

但是，List<Graphic>的形参只能接收List<Graphic>的实参，如果想要接收List<Circle>，List<Rectangle>的集合，可以使用List<?>。

代码示例：

|  |
| --- |
| **public** **static** **void** printArea(List<?> graphics){  **for** (Object obj : graphics) {  Graphic g = (Graphic) obj;  System.***out***.println(g.getArea());  }  } |

但是这样有两个问题，一个是List<?>可以接收任意类型，不仅仅图形，第二个是需要强制类型转换。为了解决这个问题，Java允许设定通配符的上限：<? extends Type>，这个通配符表示它必须是Type本身，或是Type的子类。

代码示例：

|  |
| --- |
| **public** **static** **void** printArea(List<? **extends** Graphic> graphics){  **for** (Graphic g : graphics) {  System.***out***.println(g.getArea());  }  } |

与前面的完全相同，因为我们不知道这个受限制的通配符的具体类型，所以我们不能把Graphic对象或其子类对象加入这个泛型集合中。

|  |
| --- |
| **public** **static** **void** printArea(List<? **extends** Graphic> graphics){  graphics.add(**new** Circle());//编译错误，因为不知道？的具体类型，也可能是Rectangle  } |

如果要需要将Graphic对象或其子类对象加入这个泛型集合，那么就只能用泛型方法了，示例代码请看$12.3.2。

### 12.4.3 设定通配符的下限

假设自己实现一个工具方法：实现将src集合里元素复制到dest集合中的功能，因为dest集合需要接受src的所有元素，所以dest集合元素的类型应该是src集合元素的父类。为了表示两个参数之间的类型依赖，考虑同时使用通配符、泛型形参类实现该方法。

代码示例：

|  |
| --- |
| **public** **static** <T> **void** copy(Collection<T> dest,Collection<? **extends** T> src){  **for** (T t : src) {  dest.add(t);  }  } |

上面的方法实现了前面的功能。现在假设该方法需要一个返回值，返回最后一个被复制的元素，可以把上面方法修改如下：

|  |
| --- |
| **public** **static** <T> T copy(Collection<T> dest,Collection<? **extends** T> src){  T last = **null**;  **for** (T t : src) {  dest.add(t);  last = t;  }  **return** last;  } |

表面上看这个实现没有问题，实际上有一个问题，当遍历src元素的类型是不确定（但可以肯定是T的子类），程序只能用T来笼统的表示，所以返回值类型是T。

|  |
| --- |
| **public** **static** **void** main(String[] args) {  ArrayList<String> src = **new** ArrayList<String>();  src.add("尚硅谷");  src.add("atguigu");    ArrayList<Object> dest = **new** ArrayList<Object>();    Object last = *copy*(dest, src);  } |

发现返回的T是Object，也就是说，程序在复制集合元素的过程中，丢失了src集合元素的类型String。

对于上面的copy方法，可以这样理解两个集合参数之间的依赖关系：不管src集合元素的类型是什么，只要dest集合元素类型与src的元素类相同或是它的父类即可。为了表示这种约束关系，Java允许设定通配符的下限：<? super Type>，这个通配符表示它必须是Type本身或是Type的父类。

代码示例：

|  |
| --- |
| **public** **static** <T> T copy(Collection<? **super** T> dest,Collection<T> src){  T last = **null**;  **for** (T t : src) {  dest.add(t);  last = t;  }  **return** last;  } |
| **public** **static** **void** main(String[] args) {  ArrayList<String> src = **new** ArrayList<String>();  src.add("尚硅谷");  src.add("atguigu");    ArrayList<Object> dest = **new** ArrayList<Object>();    String last = *copy*(dest, src);  } |

这样一看，如果泛型方法中定义了泛型作为返回值类型，在实际调用此方法的时候，返回值类型就是按照该泛型所属的上限的类型作为返回值类型。

注意**：只有类型通配符才可以设定下限，泛型形参是不能设定下限的。**

public class test2 {  
 public static void main(String[] args) {  
 }  
 public static void test(List<? super Farther> c, Son str){  
 //因为c里面装的必是Farther及其子类，所以只能添加Farther及Farther子类  
 c.add(str);  
 System.out.println(c.get(0));  
 }  
  
 public static void test3(List<? extends Farther> c, Son str){  
 //c里装的是Farther及其子类，不明确到底是什么类型，因此添加子类元素会报错  
 //c.add(str); 报错  
 System.out.println(c.get(0));  
 }  
  
 public static void test2(List<? super Farther> c){  
 //因为c里装的是Farther及其父类，因此对于获取来说不知道能获取到什么类型的元素  
 // ye y = c.get(0); 编译报错  
 }  
 public static void test4(List<? extends Farther> c){  
 //c装的只能是Farther及其子类，因此接收好接收。  
 ye y = c.get(0);  
 }  
  
}

### 12.4.4 泛型方法与方法重载

因为泛型既允许设定通配符的上限，也允许设定通配符的下限，如果在一个类里包含这样两个方法定义，会报错：

|  |
| --- |
| **public** **static** <T> T copy(Collection<? **super** T> dest,Collection<T> src){  //....省略代码  }    **public** **static** <T> T copy(Collection<T> dest,Collection<? **extends** T> src){  //....省略代码  } |

## 12.5 泛型擦除

在严格的泛型代码里，带泛型声明的类**总应该**带着类型参数。

但为了与老的Java代码保持一致，也允许在使用带泛型声明的类时不指定类型参数。如果没有为这个泛型类指定类型参数，则该类型参数被称作一个原始类型（raw type），默认是该类型形参的第一个上限（如果没有上限，默认就是Object）。

* 也就是在使用泛型类的时候，声明中不指定泛型实参，此实例变量会当做原始类型；

代码示例如下：

|  |
| --- |
| **public** **class** TestErase {  **public** **static** **void** main(String[] args) {  //ArrayList<String>被转换为了ArrayList  ArrayList list = **new** ArrayList<String>();  list.add("尚硅谷");  list.add("atguigu");  //泛型被擦除，按照默认上限Object处理  Object object = list.get(1);    Student s = **new** Student("张三",12);    //泛型被擦除，按照第一个上限Number处理  Number score = s.getScore();  }  }  **class** Student<T **extends** Number & java.io.Serializable>{  **private** String name;  **private** T score;  **public** Student(String name, T score) {  **super**();  **this**.name = name;  **this**.score = score;  }  **public** T getScore() {  **return** score;  }  } |

## 12.6 并不存在泛型类的Class对象

### 12.6.1 Type<String>与Type<Integer>的区别?

我们通过泛型使得一个类型的功能增强了，好像扩展出好多子类一样。例如：ArrayList<E>，通过指定泛型，可以延伸出ArrayList<String>只能处理字符串类型的集合，ArrayList<Employee>只能处理Employee员工类型的集合。但是，实际上，**系统并没有为ArrayList<String>等生成新的class文件，而且也不会把ArrayList<String>当成新的类型处理。**

看下面的代码的打印结果是什么？

|  |
| --- |
| ArrayList<Integer> list1 = **new** ArrayList<Integer>();  ArrayList<String> list2 = **new** ArrayList<String>();  System.***out***.println(list1.getClass());  System.***out***.println(list2.getClass());  System.***out***.println(list1.getClass() == list2.getClass()); //true |

* **因此可以看出ArrayList<String>不是ArrayList<Object>的子类；他们在编译时类型有区别，但是运行时类型都是ArrayList。**

因此不允许如下赋值操作。

|  |
| --- |
| ArrayList<Object> list = **new** ArrayList<String>(); //错误的 |

* **这点和数组不同，因为数组是要生成新的Class对象的，String[]仍然是Object[]的子类**因此允许如下赋值操作。

|  |
| --- |
| Object[] arr = **new** String[5]; |

但是数组这么处理也是有风险的，如下操作编译正确，但运行时会报ArrayStoreException，所以请谨慎这样操作。

|  |
| --- |
| Object[] arr = **new** String[5];  arr[0] = 12;//因为12不是字符串对象 |

### 12.6.2 instanceof后面不支持泛型类

由于系统中并不会真正生成泛型类，所以如下操作也都是不允许的：

|  |
| --- |
| ArrayList<String> list = **new** ArrayList<String>();  **//编译错误，instanceof后面不能使用泛型类**  **if**(list **instanceof** ArrayList<String>){    } |

### 12.6.3 泛型与数组

* **数组的元素类型不能包含类型变量或类型形参，除非是无上限的类型通配符。**

|  |
| --- |
| **public** **static** **void** main(String[] args) {  List<?>[] arr = **new** List<?>[2];  arr[0] = Arrays.*asList*("hello","java");  arr[1] = Arrays.*asList*(1,2,3);  **for** (List<?> list : arr) {  **for** (Object object : list) {  System.***out***.println(object);  }  }  } |

只能声明这样的数组，但是不能创建这样的数组对象，以下代码编译错误：

|  |
| --- |
| ArrayList<String>[] array = **new** ArrayList<String>[5]; //编译错误 |

以下代码也是错误的：

|  |
| --- |
| **class** MyClass<T>{  **private** T[] arr;  **public** MyClass(**int** length) {  **super**();  **this**.arr = **new** T[length];  }  } |

### 12.6.4 泛型与异常

不能在try-catch中使用类型变量。

try{

}catch(T t){}

}

也不能在throws 后面使用类型变量。

## 12.7 泛型嵌套：由外到内分析

案例：已知有省份Provice类型，有属性省份编号id和名称name，有城市City类型，有属性城市编号id和名称name,所属省份编号pid。如果要存储如下信息到一个Map中，该如何指定泛型呢？其中key为省份对象，value为该省份对应的所有城市对象。

分析：key的类型为Provice，而value要保存多个城市对象，因此是一个List或Set，其泛型实参为City类型，例如：TreeMap<Province,TreeSet<City>>。

|  |  |
| --- | --- |
| 1:北京市  1:北京市  2:海南省  1:海口市  2:三亚市 | 3:浙江省  1:绍兴市  2:温州市  3:湖州市  4:嘉兴市  5:台州市  6:金华市  7:舟山市  8:衢州市  9:丽水市 |

完整代码示例：

|  |
| --- |
| **package** com.atguigu.part05;  **public** **class** Province **implements** Comparable<Province>{  **private** **int** id;  **private** String name;  **public** Province(**int** id, String name) {  **super**();  **this**.id = id;  **this**.name = name;  }  **public** Province() {  **super**();  }  **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  @Override  **public** String toString() {  **return** id + ":" + name;  }  @Override  **public** **int** compareTo(Province o) {  **return** **this**.id - o.id;  }  } |
| **package** com.atguigu.part05;  **public** **class** City **implements** Comparable<City>{  **private** **int** id;  **private** String name;  **private** **int** pid;  **public** City(**int** id, String name, **int** pid) {  **super**();  **this**.id = id;  **this**.name = name;  **this**.pid = pid;  }  **public** City() {  **super**();  }  **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **int** getPid() {  **return** pid;  }  **public** **void** setPid(**int** pid) {  **this**.pid = pid;  }  @Override  **public** String toString() {  **return** id + ":" + name;  }  @Override  **public** **int** compareTo(City o) {  **return** **this**.id - o.id;  }  } |
| **package** com.atguigu.part05;  **import** java.util.Map.Entry;  **import** java.util.Set;  **import** java.util.TreeMap;  **import** java.util.TreeSet;  **public** **class** AreaManager {  **public** **static** **void** main(String[] args) {  //key为Province，value是一个TreeSet<City>  TreeMap<Province,TreeSet<City>> map = **new** TreeMap<Province,TreeSet<City>>();    TreeSet<City> bj = **new** TreeSet<City>();  bj.add(**new** City(1,"北京市",1));  map.put(**new** Province(1,"北京市"), bj);    TreeSet<City> hn = **new** TreeSet<City>();  hn.add(**new** City(1,"海口市",2));  hn.add(**new** City(2,"三亚市",2));  map.put(**new** Province(2,"海南省"), hn);    TreeSet<City> zj = **new** TreeSet<City>();  zj.add(**new** City(1,"绍兴市",3));  zj.add(**new** City(2,"温州市",3));  zj.add(**new** City(3,"湖州市",3));  zj.add(**new** City(4,"嘉兴市",3));  zj.add(**new** City(5,"台州市",3));  zj.add(**new** City(6,"金华市",3));  zj.add(**new** City(7,"舟山市",3));  zj.add(**new** City(8,"衢州市",3));  zj.add(**new** City(9,"丽水市",3));  map.put(**new** Province(3,"浙江省"), zj);  //Map中实际存储的是一个个的Entry对象，所有的Entry就组成了一个Set集合  //而Entry类型的key是Province，value是TreeSet<City>  Set<Entry<Province, TreeSet<City>>> entrySet = map.entrySet();  **for** (Entry<Province, TreeSet<City>> entry : entrySet) {  Province key = entry.getKey();  System.***out***.println(key);  TreeSet<City> value = entry.getValue();  **for** (City city : value) {  System.***out***.println("\t" + city);  }  }  }  } |

## 12.8 Collections工具类

参考操作数组的工具类：Arrays。

Collections 是一个操作 Set、List 和 Map 等集合的工具类。Collections 中提供了一系列静态的方法对集合元素进行排序、查询和修改等操作，还提供了对集合对象设置不可变、对集合对象实现同步控制等方法：

* public static <T> boolean addAll(Collection<? super T> c,T... elements)将所有指定元素添加到指定 collection 中。
* public static <T> int binarySearch(List<? extends Comparable<? super T>> list,T key)在List集合中查找某个元素的下标，但是List的元素必须是T或T的子类对象，而且必须是可比较大小的，即支持自然排序的。而且集合也事先必须是有序的，否则结果不确定。
* public static <T> int binarySearch(List<? extends T> list,T key,Comparator<? super T> c)在List集合中查找某个元素的下标，但是List的元素必须是T或T的子类对象，而且集合也事先必须是按照c比较器规则进行排序过的，否则结果不确定。
* public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)在coll集合中找出最大的元素，集合中的对象必须是T或T的子类对象，而且支持自然排序
* public static <T> T max(Collection<? extends T> coll,Comparator<? super T> comp)在coll集合中找出最大的元素，集合中的对象必须是T或T的子类对象，按照比较器comp找出最大者
* public static void reverse(List<?> list)反转指定列表List中元素的顺序。
* public static void shuffle(List<?> list) List 集合元素进行随机排序，类似洗牌
* public static <T extends Comparable<? super T>> void sort(List<T> list)根据元素的自然顺序对指定 List 集合元素按升序排序
* public static <T> void sort(List<T> list,Comparator<? super T> c)根据指定的 Comparator 产生的顺序对 List 集合元素进行排序
* public static void swap(List<?> list,int i,int j)将指定 list 集合中的 i 处元素和 j 处元素进行交换
* public static int frequency(Collection<?> c,Object o)返回指定集合中指定元素的出现次数
* public static <T> void copy(List<? super T> dest,List<? extends T> src)将src中的内容复制到dest中
* public static <T> boolean replaceAll(List<T> list，T oldVal，T newVal)：使用新值替换 List 对象的所有旧值
* Collections 类中提供了多个 synchronizedXxx() 方法，该方法可使将指定集合包装成线程同步的集合，从而可以解决多线程并发访问集合时的线程安全问题
* Collections类中提供了多个unmodifiableXxx()方法，该方法返回指定 Xxx的不可修改的视图。
* 说明：public sComparable<? super T>> T max(Collection<? extends T> coll)在coll集合中找说明：public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)方法的泛型形参为什么要设定上限为Object & Comparable，这是为了当泛型擦除时，按照Object处理，而不是Comparable处理，这样就可以和JDK1.5之前没有泛型的API保持一致。

## 12.9 本章小结

JDK1.5的泛型有一个很重要的设计原则：如果一段代码在编译时系统没有产生：“[unchecked]未经检查的转换”警告，则程序在运行时不会引发“ClassCastException”异常。

本章主要介绍了JDK1.5提供的泛型支持，泛型给程序带来了简洁性和健壮性。本章详细讲解了如何定义泛型接口、泛型类、方法方法，包括如何声明泛型形参，如何指定泛型实参，还有如何设定泛型形参的上下，此外还介绍了类型通配符的用法，包括设定类型通配符的上下、下限等，以及泛型方法与类型通配符之间的区别。相信学完这章，对于集合的使用会更顺手，也会对Java的类型有更进一步的认识。恭喜你，你的技能值又提高了800。