# Java泛型

# 摘要：

本章不仅会介绍如何通过泛型来实现编译时检查集合元素的类型，而且还会深入介绍Java泛型的详细用法，包括：

1. 定义泛型类

2. 泛型接口

3. 以及类型通配符

4. 泛型方法

# 一、泛型概述

## 1.1 Java集合设计

Java集合有个缺点：当我们把一个对象“丢进”集合里后，集合就会忘记这个对象的数据类型，当再次取出该对象时，该对象的编译类型就变成了Object类型（其运行时类型没变）。

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用它来保存什么类型的对象，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。但这样做带来如下两个问题：

1. 集合对元素类型没有任何限制，这样可能引发异常。

2. 取出集合元素后通常还需要进行强制类型转换。这种强制类型转换既增加了编程的复杂度，也可能引发ClassCastExccption异常。

## 1.2 泛型的引入

从Java 5以后，Java引入了参数化类型(parameterized type)的概念，允许我们在创建集合

时指定集合元素的类型，Java的参数化类型被称为泛型（Generic）。

在JDK 1.5增加泛型支持， 增加了泛型支持后的集合，完全可以记住集合中元素的类型，并可以在编译时检查集合中元素的类型，如果试图向集合中添加不满足类型要求的对象，编译器就会提示错误。增加泛型后的集合，可以让代码更加简洁，程序更加健壮（Java泛型可以保证如果程序在编译时没有发出警告，运行时就不会产生ClassCastException异常）。

## 1.3 Java7中的泛型

在Java 7以前，如果使用带泛型的接口、类定义变量，那么调用构造器创建对象时构造器的后面也必须带泛型，这显得有些多余了。例如如下两条语句：

List<String> strList = **new** ArrayList<**String**>();

Map<String,Integer> scores = **new** HashMap<**String,Integer>**();

上面两条语句中的粗体字代码部分完全是多余的，在Java 7以前这是必需的，不能省略。从Java 7开始，Java允许在构造器后不需要带完整的泛型信息，只要给出一对尖括号即可：

List<String> strList = **new** ArrayList< >();

Map<String,Integer> scores = **new** HashMap< >();

# 二、深入泛型

**所谓泛型**，就是允许在定义类、接口、方法时使用类型形参，这个类型形参将在声明变量、创建对象、调用方法时动态地指定（即传入实际的类型参数，也可称为类型实参）。Java 5改写了集合框架中的全部接口和类，为这些接口、类增加了泛型支持，从而可以在声明集合变量、创建集合对象时传入类型实参。

## 2.1 定义泛型接口、类

包含泛型声明的类型可以在定义变量、创建对象时传入一个类型实参，从而可以动态地生成无数多个逻辑上的子类，但这种子类在物理上并不存在。

我们可以为任何类、接口增加泛型声明，并不是只有集合类才可以使用泛型声明，虽然集合类是泛型的重要使用场所。下面自定义一个Apple类，这个Apple类就可以包含一个泛型声明

**public** **class** Apple <E>{

**private** E info;

**public** Apple(){}

**public** Apple(E info) {

**super**();

**this**.info = info;

}

**public** E getInfo() {

**return** info;

}

**public** **void** setInfo(E info) {

**this**.info = info;

}

@Override

**public** String toString() {

**return** "Apple [info=" + info + "]";

}

**public** **static** **void** main(String[] args) {

Apple<String> apple = **new** Apple<>("苹果");

System.*out*.println(apple);

Apple<Double> apple2 = **new** Apple<>(5.69);

System.*out*.println(apple2);

}

}

## 2.2 从泛型类派生子类

当创建了带泛型声明的接口、父类之后，可以为该接口创建实现类，或从该父类派生子类，但需要指出的是，当使用这些接口、父类时不能再包含类型形参。例如，下面代码就是错误的

**class** Apple **extends** Fruit<E>

如果想从Apple类派生一个于类，则可以改为如下代码：

**class** Apple<E,T> **extends** Fruit<E> / **class** Apple<E> **extends** Fruit<E>

**class** Apple **extends** Fruit<String>

**class** Apple **extends** Fruit

**（1）方式一**

当继承的父类包含类型形参时，子类必须也要包含泛型参数，如上当使用Fruit<E>作为父类时，子类必须也要加上泛型参数E，并且可以在此基础上在定义一种新的泛型参数T如下：

**public** **class** Pear<E,T> **extends** Fruit<E>{

**private** T version;

@Override

**public** E getInfo() {

**return** **super**.getInfo();

}

@Override

**public** **void** setInfo(E info) {

**super**.setInfo(info);

}

**public** T getVersion() {

**return** version;

}

**public** **void** setVersion(T version) {

**this**.version = version;

}

@Override

**public** String toString() {

**return** "Pear [version=" + version + "]";

}

}

**（2）方式二**

如果从Fruit <String>类派生子类，则在Apple类中所有使用E类型形参的地方都将被替换成String类型，即它的子娄将会继承到String getinfo()和void setinfo(String info)两个方法，如果子类需要重写父类的方法，就必须注意这一点。如下：

**public** **class** Banana **extends** Fruit<String>{

//正确重写了父类，返回值与父类Fruit<String>的返回值相同

**public** String getInfo(){

**return** **super**.getInfo();

}

//编译错误，重写方法时参数类型不一致

// public void setInfo(Object obj){

// }

}

**（3）方式三**

如果使用父类Fruit时没有传入实际的类型参数，Java编译器可能发出警告：使用了未经检查或不安全的操作——这就是泛型检查的警告。此时，系统会把Fruit<E>类里的E形参当成Object类型处理。如下程序所示:

**public** **class** Orange **extends** Fruit{

//super.getInfo()返回的是Object类型

**public** String getInfo(){

**return** (String) **super**.getInfo();

}

**public** **void** setInfo(String ele){

**super**.setInfo(ele);

}

}

## 2.3 不存在泛型类

不管为泛型的类型形参传入哪一种类型实参，对于Java来说，它们依然被当成同一个类处理，在内存中也只占用一块内存空间，因此在静态方法、静态初始化块或者静态变量的声明和初始化中不允许使用类型形参。下面程序演示了这种错误

**public** **class** R<T>{

**private** T info;

/\*\* 编译错误，不能在静态Fields中声明使用类型形参 \*\*/

// private static T tag;

**public** **void** foo(T msg){}

/\*\* 编译错误，不能在静态方法中使用类型形参 \*\*/

// public void static bar(T msg){}

}

由于系统中并不会真正生成泛型类，所以instanceof运算符后不能使用泛型类。如下：

Collection cs = new ArrayList<String>(); if(cs instanceof List<String>){…..}

在使用泛型类时，虽然传入了不同的泛型实参，但并没有真正意义上生成不同的类型，传入不同泛型实参的泛型类在内存上只有一个，即还是原来的最基本的类型，当然，在逻辑上我们可以理解成多个不同的泛型类型。

究其原因，在于Java中的泛型这一概念提出的目的，导致其只是作用于代码编译阶段，在编译过程中，对于正确检验泛型结果后，会将泛型的相关信息擦出。也就是说，成功编译过后的class文件中是不包含任何泛型信息的，泛型信息不会进入到运行时阶段。

**对此总结成一句话：泛型类型在逻辑上看以看成是多个不同的类型，实际上都是相同的基本类型。**

# 三、类型通配符

为了表示各种泛型List的父类，我们需要使用类型通配符，类型通配符是一个问号(?)，将一个问号作为类型实参传给List集合，写作：List<?>意思是未知类型元素的List。这个问号(?)被称为通配符，它的元素类型可以匹配任何类型。

（1）List list = new ArrayList<String>(); **√**

（2）List<Object> list = new ArrayList<String>(); **X**

（3）List<?> list = new ArrayList<String>(); **√**

由此可知，虽然String是Object的子类，但是List<String>并不能最为List<Object>的子类。而List<?>可以作为所有List泛型的父类。

但这种带通配符的List仅表示它是各种泛型List的父类，并不能把元素加入到其中。例如，如下：

List<?> list2 = **new** ArrayList<String>();

list2.add(**null**);

list2.add("JOney");//编译错误

上面的list2除了可以添加null元素外，其他元素均不能加入。另一方面，程序可以调用get()方法来返回List<?>集合指定索引处的元素，其返回值是一个未知类型，但可以肯定的是，它总是一个Object。

## 3.1 通配符上限

通配符上限，用来表示它是某一类泛型父类。如下：

**public abstract** class Shape {}

**public class** Circle **extends** Shape{}

**public** **class** Rectangle **extends** Shape{}

List<? **extends** Shape> shapes = List<Circle> √

List<? **extends** Shape> shapes = List<Rectangle> √

List<? **extends** Shape> shapes = List<Apple> X

由此可知，List<? **extends** Shape>代表的是所有shape泛型的父类，前面List<?>代表的是所有**List泛型的父类。**List<? **extends** Shape>是受限制通配符的例子，此处的问号（?）代表一个未知的类型，就像前面看到的通配符一样。但是此处的这个未知类型一定是Shape的子类型（也可以是Shape本身），因此我们把Shape称为这个通配符的上限（upper bound）。

因为我们不知道这个受限制的通配符的具体类型，所以不能把Shape对象或其子类的对象加入这个泛型集合中。例如，下面代码就是错误的

List<? **extends** Shape> shapes = **new** ArrayList<Circle>();

shapes.add(**new** Circle()); //编译错误，类型不匹配

## 3.2 类型参数上限

Java泛型不仅允许在使用通配符形参时设定上限，而且可以在定义类型形参时设定上限，用于表示传给该类型形参的实际类型要么是该上限类型，要么是该上限类型的子类。下面程序示范了这种用法：

**public** **class** Graph<T **extends** Shape> {}

Graph<Circle> gc = **new** Graph<>();

Graph<Rectangle> gr = **new** Graph<>();

Graph<String> gs = new Graph<>();//编译错误String不是Shape的子类

在一种更极端的情况下，程序需要为类型形参设定多个上限（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（是父类本身也行），并且实现多个上限接口。如下代码所示:

**public** **class** Graph<T **extends** Shape & Serializable> {}

表示类型参数T必须是Shape的子类同时实现Serializable接口

# 四、泛型方法

## 4.1 泛型方法定义

所谓泛型方法，就是在声明方法时定义一个或多个类型形参。泛型方法的用法格式如下：

修饰符 <T，S> 返回值类型 方法名(形参列表){

方法体

}

采用上述格式定义的泛型方法示例如下：

**static** <T> **void** FromArrayToCollecton(T[] arys,Collection<T> collects){

**for** (T t : arys) {

collects.add(t);

}

}

Object[] oa = **new** Object[100];

Collection<Object> co = **new** ArrayList<>();

String[] sa = **new** String[100];

Collection<String> cs = **new** ArrayList<>();

与类、接口中使用泛型参数不同的是，方法中的泛型参数无须显式传入实际类型参数，如上面程序所示，当程序调用fromArrayToCollection()方法时，无须在调用该方法前传入String、Object等类型，但系统依然可以知道类型形参的数据类型，因为编译器根据实参推断类型形参的值，它通常推断出最直接的类型参数。

上面代码中cs是一个Collection<String>类型，与方法定义时的fromArrayToCollection(T[] a，

Collection<T> c)进行比较，只比较泛型参数，不难发现该T类型形参代表的实际类型是String类型。

当方法中传入的两个参数表都是泛型参数时，编译器无法正确识别T所代表的实际类型，为了避免这种修改如下：

**static** <T> **void** CopyToCollection2(Collection<? **extends** T> from,Collection<T> to){

**for** (T ele : from) {

to.add(ele);

}

}

## 4.2 泛型方法和类型通配符

大多数时候都可以使用泛型方法来代替类型通配符。例如，对于Java的Collection接口中两个方法定义：

**public** **interface** Collection<E>{

**boolean** containsAll(Collection<?> c);

**boolean** addAll(Collection<? **extends** E> c);

}

泛型方法实现如下：

**public** **interface** Collection<E>{

<T> **boolean** containsAll(Collection<T> c);

<T **extends** E> **boolean** addAll(Collection<T **extends** E> c);

}

上面两个方法中类型形参T只使用了一次，类型形参T产生的唯一效果是可以在不同的调用点传入不同的实际类型。对于这种情况，应该使甩通配符：通配符就是被设计用来支持灵活的子类化的。

泛型方法，允许类型形参被用来表示方法的一个或多个参数之间的类型依赖关系。或者方法返回值与参数之间的类型依赖关系。如果没有这样的类型依赖关系，就不应该使用泛型方法。

如果有需要我们可以同时使用泛型方法和通配符，如Java的Collections.copy()方法：

public interface Collections{

//泛型方法+通配符

public static <T> void copy(List<T> dest,List<? extends T> src){}

}

上面copy方法中的dest和src存在明显的依赖关系，从源List中复制出来的元素，必须可以“丢进”目标List中。所以源List集合元素的类型，只能是目标集合元素的类型的子类型或者它本身。但JDK定义形参类型时使用的是类型通配符，而不是泛型方法。这是因为：该方法无须向src集合中添加强元素，也无须修改src集合里的元素，所以可以使用类型通配符，无须使用泛型方法。

当然，也可以将上面的方法签名改为使用泛型方法，不使用类型通配符如下：

public interface Collections{

//泛型方法

public static <T,S extends T> void copy(List<T> dest,List<S> src){}

}

这个方法签名，可以代替前面的方法签名。但注意上面的类型形参S，它仅仅使用了一次，没有其他参数的类型、方法返回值的类型依赖于它，那类型形参S就没有存在的必要，即可以用通配符来代替S。

使用通配符比使用泛型方法（在方法签名中显式声明类型形参）更加清晰和准确，因此Java设计该方法时采用了通配符，而不是泛型方法。

类型通配符与泛型方法（在方法签名中显式声明类型形参）、还有个显著的区别：类型通配符既可以在方法签名中定义形参的类型，也可以用于定义变量的类型；但泛型方法中的类型形参，必须在对应方法中显式声明。

## 4.3 Java菱形语法和泛型构造器

正如泛型方法允许在方法签名中声明类型形参一样，Java也允许在构造器签名中声明类型形参这样就产生了所谓的泛型构造器。一旦定义了泛型构造器，接下来在调用构造器时，不仅可以让：

1. Java根据数据参数的类型来推断类型形参的类型

2. 程序员也可以显式地为构造器中的类型形参指定实际的类型

如下程序所示:

**public** **class** DemoGenericConstructor {

**public** **static** **void** main(String[] args) {

Foo f1 = **new** Foo("String");

Foo f2 = **new** Foo(11);

Foo f3 = **new** <String> Foo("<String>");

// Foo f4 = new <String> Foo(3.14);//编译错误

}

}

**class** Foo{

**public** <T> Foo(T t){

System.*out*.println(t);

}

}

Java 7新增的“菱形”语法，它允许调用构造器时在构造器后使用一对尖括号来代表泛型信息。但如果程序**显式指定了**泛型构造器中声明的类型形参的实际类型，则不可以使用“菱形一语法。如下程序：

**public** **class** DemoGenericDiamond {

**public** **static** **void** main(String[] args) {

//MyClass类声明中的E为String类型

//泛型构造器声明的T为Integer

//使用了菱形语法<>表示泛型信息

MyClass<String> mc1 = **new** MyClass<>(5);

//显示指定泛型构造器中声明的T为Integer

MyClass<String> mc2 = **new** <Integer> MyClass<String>(5);

//MyClass类声明中的E为String类型

//泛型构造器声明的T为Integer

//此时不能再次使用菱形语法来代表泛型信息

// MyClass<String> mc3 = new <Integer> MyClass<>(5);

}

}

**class** MyClass<E> {

**public** <T> MyClass(T t){

System.*out*.println("t的参数值为："+t);

}

}

## 4.4 设定通配符下限

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。下面程序采用设定通配符下限的方式改写了前面的copy方法。

**public** **static** <T> T copy(Collection<? **super** T> des, Collection<T> src){

T last=**null**;

**for** (T ele : src) {

last=ele;

des.add(ele);

}

**return** last;

}

上面实现的copy方法，可以准确的知道最后一个元素是Integer类型与src集合的元素类型的相同。

在Java集合框架中的TreeSet<E>有个构造器也用到了这种设定通配符下限的语法，如下：

//下面的E是定义TreeSet的类型形参

TreeSet(Comparator<? **super** E> c)

TreeSet会对集合中的元素按自然顺序或定制顺序进行排序。如果需要TreeSet对集合中的所有元素进行定制排序，则要求TreeSet对象有—个与之关联的Comparator对象。上面构

造器中的参数c就是进行定制排序的Comparator对象。

Comparator接口也是个带泛型声明的接口：

**public** **interface** Comparator<T> {

**int** compare(T o1, T o2);

}

通过这种带下限的通配符的语法，可以在创建TreeSet对象时灵活地选择合适的Comparator。假定需要创建一个TreeSet<String>集合，井传入一个可以比较String大小的Comparator，这个Comparator既可以是Comparator<String>，也可以是Comparator<Object>只要尖括号里传人的类型是String的父类型或它本身即可。如下程序所示:

**public** **class** DemoTreeSet {

**public** **static** **void** main(String[] args) {

//Comparator的实际类型是TreeSet里实际类型的父类

TreeSet<String> ts1 = **new** TreeSet<String>(**new** Comparator<Object>(){

@Override

**public** **int** compare(Object o1, Object o2) {

**return** o1.hashCode()> o2.hashCode()?1

: o1.hashCode()<o2.hashCode()?-1:0;

}

});

ts1.add("qeq");

ts1.add("ad2");

ts1.add("ada1");

System.*out*.println(ts1);

TreeSet<String> ts2 = **new** TreeSet<String>(**new** Comparator<String>(){

@Override

**public** **int** compare(String o1, String o2) {

**return** o1.length()>o2.length()?1

:o1.length()<o2.length()?-1:0;

}

});

ts2.add("12313123");

ts2.add("12");

ts2.add("2345");

System.*out*.println(ts2);

}

}

通过使用这种通配符下限的方式来定义TreeSet构造器的参数，就可以将所有可用的Comparator作为参数传入，从而增加了程序的灵活性。

## 4.5 泛型方法与泛型方法重载

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，从而允许在一个类里包含如下两个方法定义

**class** MyUtils{

**public** **static** <T> **void** Copy(Collection<T> des,Collection<? **extends** T> src){} ①

**public** **static** <T> T Copy(Collection<? **super** T> des,Collection<T> src){} ②

}

上面的MyUtils类中包含两个copy方法，这两个方法的参数列表存在一定的区别，但这种区别不是很明确：这两个方法的两个参数都是Collection对象，前一个集合里的集合元素类型是后一个集合里集合元素类型的父类。如果这个类仅包含这两个方法不会有任何错误，但只要调用这个方法就会引起编译错误。例如，对于如下代码：

List<Number> ln = **new** ArrayList<Number>();

List<Interger> li = **new** ArrayList<Integer>();

**copy(ln,li)**

上面程序中粗体字部分调用copy方法，但这个copy方法既可以匹配①号copy方法，此时T类型参数的类型是Number；也可以匹配②号copy方法，此时T参数的类型是Integer。编译器无法确定这行代码想调用哪个copy方法，所以这行代码将引起编译错误。

# 五、擦除和转换

## 5.1 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实际的类型参数。如果没有为这个泛型类指定实际的类型参数，则该类型参数被称作raw type（原始类型），默认是声明该参数时指定的第一个上限类型。

当把一个具有泛型信息的对象赋给另一个没有泛型信息的变量时，所有在尖括号之间的类型信息都将被扔掉。比如一个List<String>类型被转换为List，则该List对集合元素的类型检查变成了类型变量的上限(即Object)。下面程序示范了这种擦除:

**class** Foo<T **extends** Number>{

T size;

}

Foo<Integer> f1 = **new** Foo<>(); //f1的getSize()方法返回Integer对象

Integer size1 = f1.getSize();//把f1赋给f2，丢失尖括号里的类型信息

Foo f2 = f1;//f2只知道size的类型是Number

Number size2 = f2.getSize();//下面的代码将会引起编译错误

// Integer size3 = f2.getSize();

## 5.2 泛型转换

从逻辑上来看List<String>是List的子类，如果直接把一个List对象赋给一个List<String>对象应该引起编译错误，但实际上不会。对泛型而言，可以直接把一个List对象赋给一个List<String>对象，编译器仅仅提示“未经检查的转换”，看下面程序

**public** **class** DemoErasure {

**public** **static** **void** main(String[] args) {

List<Integer> list = **new** ArrayList<>();

list.add(1);

list.add(3);

List li = list;

//下面代码有“未经检查的转换”警告，在编译、运行时候正常

List<String> ls =li;

//只要访问ls里的元素就回出现运行异常

System.*out*.println(ls.get(0));

}

}