# 泛型

## 1,基本属性

### 16.06\_集合框架(泛型概述和基本使用)generic(广泛之意)

**A:泛型概述**

**B:泛型好处**

**提高安全性(将运行期的错误转换到编译期)**

**省去强转的麻烦(若是没有加泛型,添加的东西都将提升为object类型),泛型擦除就是为了老版本的java着想**

**扩展性更强,符合编程的抽象开发的软件编程宗旨**

**C:泛型基本使用**

**<>中放的必须是引用数据类型 基本类型应该封装起来,int就是Integer.集合创建的时候和获取迭代器的时候,类上,方法上.**

**D:泛型使用注意事项**

**前后的泛型必须一致,或者后面的泛型可以省略不写(1.7的新特性菱形泛型),工作的时候必须得前后都得写上!工作中稳定用的版本是比较低的1.5或者是1.6.泛型最好不要定义成Object,没有意义 .有一边的为object也是不可以的!泛型也是有局限性的可以通过反射得到或者注入不合乎的类型数据.**

**但泛型还有一种较为准确的说法就是为了参数化类型，或者说可以将类型当作参数传递给一个类或者是方法。**

---------------------------------------------------------------------------------------------------------------------------

public class Demo1\_Generic {

public static void main(String[] args) {

//demo1();

//int[] arr = new byte[5]; //数组要保证前后的数据类型一致

//ArrayList<Object> list = new ArrayList<Person>(); //集合的泛型要保证前后的数据类型一致

//ArrayList<Object> list = new ArrayList<>(); //1.7版本的新特性,菱形泛型

ArrayList<Object> list = new ArrayList<>();//泛型最好不要定义成Object,没有意义

list.add("aaa");

list.add(true);

}

public static void demo1() {

ArrayList<Person> list = new ArrayList<Person>(); 集合加上泛型

// list.add(110);

// list.add(true);

list.add(new Person("张三", 23));

list.add(new Person("李四", 24));

Iterator<Person> it = list.iterator(); 迭代器加上泛型，注意和上边保持一致。

while(it.hasNext()) {

//System.out.println(it.next());

***//System.out.println(it.next().getName() + "..." + it.next().getAge());//next方法只能调用一次,如果调用多次会将指针向后移动多次，最好就是将it.next()定义为一个变量。***

***Person p = it.next();***

System.out.println(p.getName() + "..." + p.getAge());

}

}

}

---------------------------------------------------------------------------------------------------------------------------

### 16.07\_集合框架(ArrayList存储字符串和自定义对象并遍历泛型版)(掌握)

重点必须掌握.

A:案例演示

ArrayList存储字符串并遍历泛型版

---------------------------------------------------------------------------------------------------------------------------

package com.heima.generic;

import java.util.ArrayList;

import java.util.Iterator;

import com.heima.bean.Person;

public class Demo2\_Generic {

/\*\*

A:案例演示

ArrayList存储字符串并遍历泛型版

\*/

public static void main(String[] args) {

//demo1();

ArrayList<Person> list = new ArrayList<>();

list.add(new Person("张三", 23));

list.add(new Person("李四", 24));

list.add(new Person("王五", 25));

list.add(new Person("赵六", 26));

Iterator<Person> it = list.iterator();

while(it.hasNext()) {

Person p = it.next(); **//将集合中的每一个元素用Person记录**

System.out.println(p.getName() + "..." + p.getAge());

}

}

------------------------------------------------------------------------------------------------

public static void demo1() {

ArrayList<String> list = new ArrayList<>(); //创建集合对象

list.add("a");

list.add("b");

list.add("c");

list.add("d");

Iterator<String> it = list.iterator();

while(it.hasNext()) {

System.out.println(it.next());

}

}

---------------------------------------------------------------------------------------------------------------------------

### 16.08\_集合框架(泛型的由来),将编译时期的问题提到写代码和编译的时候.

A:案例演示

泛型的由来:通过Object转型问题引入

早期的Object类型可以接收任意的对象类型，但是在实际的使用中，会有类型转换的问题。也就存在这隐患，所以Java提供了泛型来解决这个安全问题。出现ClassCaseException问题.

就是说强转的时候书写是没有问题的,但是在运行的时候就有问题了.类型不对啊

--------------------------------------------------------------------

**public** **static** **void** demo1() {

Tool<Student> t = **new** Tool<>(); //创建工具类对象

t.setObj(**new** Student("张三",23));

Worker w = (Worker) t.getObj(); //向下转型

System.out.println(w); //$$$运行出错的!

}

------------------------------------------------------------

## 2,泛型的三种使用

### 16.09\_集合框架(泛型类的概述及使用)

重点:掌握使用的方法.

A:泛型类概述<T>

把泛型定义在类上,T就type类型的意思,E就是元素的意思,一般都是大写的,可随意.

泛型就和变量的定义是一样的,只是一个变量而已.

B:定义格式

public class 类名<泛型类型1,…> 注意可以有很多的类型

C:注意事项

**泛型类型必须是引用类型,泛型是不能创建对象的.泛型也不能对数组进行声明.**

**泛型就是将参数的声明提升到类上边,下边的方法和变量便于使用,方法和变量也是可以不同于类上边的泛型的,但是不推荐.**

D:案例演示

泛型类的使用

----------------------------------------------------------------------------------------------------

public class MultiType <E,T>{

E value1;

T value2;

public E getValue1(){

return value1;

}

public T getValue2(){

return value2;

}

}

----------------------------------------------------------------------------------------------------

### 16.10\_集合框架(泛型方法的概述和使用)

A:泛型方法概述

把泛型定义在方法上

B:定义格式

public <泛型类型> 返回类型 方法名(泛型类型 变量名)

C:案例演示

泛型方法的使用

---------------------------------------------------------------------------------------------------------------------------

**public** **class** Tool<Q> {

**private** Q q;

**public** Q getObj() {

**return** q;

}

**public** **void** setObj(Q q) {

**this**.q = q;

}

**public** **void** show(Q q) {

System.***out***.println(q); **//推荐**

}

**public**<T> **void** show(T t) { **//方法泛型最好与类的泛型一致**

System.***out***.println(t); **//如果不一致,需要在方法上声明该泛型,返回类型也可以是泛型T**

}

**public** **static**<W> **void** print(W w) { **//静态方法必须声明自己的泛型,因为他是静态的不能使用人家非静态.当可以相同为T,static<T>**

System.***out***.println(w);

}

}

**注意:所有的符号只是一个表示,当然是可以运用相同的符号表示的.**

---------------------------------------------------------------------------------------------------------------------------

### 16.11\_集合框架(泛型接口的概述和使用)

A:泛型接口概述

把泛型定义在接口上

B:定义格式

public interface 接口名<泛型类型>

C:案例演示

泛型接口的使用

---------------------------------------------------------------------------------------------------------------------------

**public** **class** Demo4\_Generic {

/\*\*

\* \* A:泛型接口概述

\* 把泛型定义在接口上

\* B:定义格式

\* public interface 接口名<泛型类型>

\* C:案例演示

\* 泛型接口的使用

\*/

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

}

}

**interface** Inter<T> {

**public** **void** show(T t);

}

/\*class Demo implements Inter<String> { //推荐用这种

@Override

public void show(String t) {

System.out.println(t);

}

}\*/

**class** Demo<T> **implements** Inter<T> { //没有必要在实现接口的时候给自己类加泛型

@Override

**public** **void** show(T t) {

System.***out***.println(t);

}

}

### 泛型类与泛型方法的共存现象

public class Test1<T>{

public void testMethod(T t){

System.out.println(t.getClass().getName());

}

public <T> T testMethod1(T t){

return t;

}

}

上面代码中，Test1<T>是泛型类，testMethod 是泛型类中的普通方法，而 testMethod1 是一个泛型方法。而泛型类中的类型参数与泛型方法中的类型参数是没有相应的联系的，**泛型方法始终以自己定义的类型参数为准**。

所以，针对上面的代码，我们可以这样编写测试代码。

Test1<String> t = new Test1();

t.testMethod("generic");

Integer i = t.testMethod1(new Integer(1));

泛型类的实际类型参数是 String，而传递给泛型方法的类型参数是 Integer，两者不想干。

但是，为了避免混淆，如果在一个泛型类中存在泛型方法，那么两者的类型参数最好不要同名。比如，Test1<T>代码可以更改为这样

public class Test1<T>{

public void testMethod(T t){

System.out.println(t.getClass().getName());

}

public <E> E testMethod1(E e){

return e;

}

}

---------------------------------------------------------------------------------------------------------------------------

## 3,通配符

### 16.12\_集合框架(泛型高级之通配符)

A:泛型通配符<?>

任意类型，如果没有明确，那么就是Object以及任意的Java类了,但是?并不是Object

B:? extends E

向下限定，E及其子类

C:? super E

向上限定，E及其父类

---------------------------------------------------------------------------------------------------------------------------

List<?> list = new ArrayList<Integer>(); //当右边的泛型是不确定时,左边可以指定为?,右边可以是String,等等,左边虽是?,但不能是object

ArrayList<Person> list1 = **new** ArrayList<>();

list1.add(**new** Person("张三", 23));

list1.add(**new** Person("李四", 24));

list1.add(**new** Person("王五", 25));

ArrayList<Student> list2 = **new** ArrayList<>();

list2.add(**new** Student("赵六", 26));

list2.add(**new** Student("周七", 27));

list1.addAll(list2);

System.***out***.println(list1); //Student extents Person,一般是规定用

---------------------------------------------------------------------------------------------------------------------------

Java 建议我们用单个大写字母来代表类型参数。常见的如：

T 代表一般的任何类。

E 代表 Element 的意思，或者 Exception 异常的意思。

K 代表 Key 的意思。

V 代表 Value 的意思，通常与 K 一起配合使用。

S 代表 Subtype 的意思，文章后面部分会讲解示意。

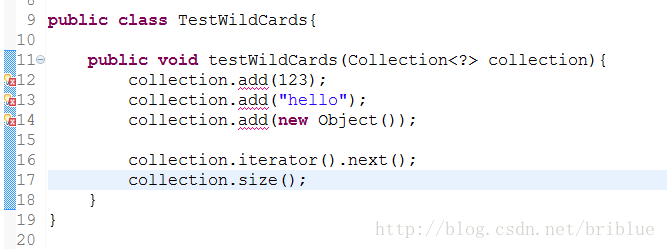
### 无限定通配符 <?>

无限定通配符经常与容器类配合使用，它其中的 ? 其实代表的是未知类型，所以涉及到 ? 时的操作，一定与具体类型无关。

public void testWildCards(Collection<?> collection){

}

上面的代码中，方法内的参数是被无限定通配符修饰的 Collection 对象，它隐略地表达了一个意图或者可以说是限定，**那就是 testWidlCards() 这个方法内部无需关注 Collection 中的真实类型，因为它是未知的。所以，你只能调用 Collection 中与类型无关的方法**。



我们可以看到，当 <?>存在时，Collection 对象丧失了 add() 方法的功能，编译器不通过。

我们再看代码。

List<?> wildlist = new ArrayList<String>();

wildlist.add(123);// 编译不通过

**有人说，<?>提供了只读的功能，也就是它删减了增加具体类型元素的能力，只保留与具体类型无关的功能。它不管装载在这个容器内的元素是什么类型，它只关心元素的数量、容器是否为空,***我想这种需求还是很常见的吧*。

有同学可能会想，<?>既然作用这么渺小，那么为什么还要引用它呢？ 

个人认为，提高了代码的可读性，程序员看到这段代码时，就能够迅速对此建立极简洁的印象，能够快速推断源码作者的意图。

### <? extends T>

<?>代表着类型未知，但是我们的确需要对于类型的描述再精确一点，我们希望在一个范围内确定类别，比如类型 A 及 类型 A 的子类都可以。

<? extends T> 代表类型 T 及 T 的子类。

```java public void testSub(Collection<? extends Base> para){ } ```

上面代码中，para 这个 Collection 接受 Base 及 Base 的子类的类型。 但是，它仍然丧失了写操作的能力。也就是说 ```java para.add(new Sub()); para.add(new Base()); ``` 仍然编译不通过。 没有关系，我们不知道具体类型，但是我们至少清楚了类型的范围。

```<? super T> ``` 这个和 ```<? extends T>```相对应，代表 T 及 T 的超类。

```java public void testSuper(Collection<? super Sub> para){ } ```

```<? super T>```神奇的地方在于，它拥有一定程度的写操作的能力。

```java public void testSuper(Collection<? super Sub> para){ para.add(new Sub());//编译通过 para.add(new Base());//编译不通过 } ```

通配符与类型参数的区别 一般而言，通配符能干的事情都可以用类型参数替换。 比如

```java public void testWildCards(Collection<?> collection){}

可以被

```java

public <T> void test(Collection<T> collection){}

取代。

值得注意的是，如果用泛型方法来取代通配符，那么上面代码中 collection 是能够进行写操作的。只不过要进行强制转换。

public <T> void test(Collection<T> collection){

collection.add((T)new Integer(12));

collection.add((T)"123");

}

需要特别注意的是，类型参数适用于参数之间的类别依赖关系，举例说明。

public class Test2 <T,E extends T>{

T value1;

E value2;

}

public <D,S extends D> void test(D d,S s){

}

E 类型是 T 类型的子类，显然这种情况类型参数更适合。

有一种情况是，通配符和类型参数一起使用。

public <T> void test(T t,Collection<? extends T> collection){

}

如果一个方法的返回类型依赖于参数的类型，那么通配符也无能为力。

public T test1(T t){

return value1;

}

## 4,泛型擦除

### 1,概述

泛型是 Java 1.5 版本才引进的概念，在这之前是没有泛型的概念的，但显然，泛型代码能够很好地和之前版本的代码很好地兼容。

这是因为，**泛型信息只存在于代码编译阶段，在进入 JVM 之后，与泛型相关的信息会被擦除掉，专业术语叫做类型擦除**。

通俗地讲，**泛型类和普通类在 java 虚拟机内是没有什么特别的地方**。回顾文章开始时的那段代码

List<String> l1 = new ArrayList<String>();

List<Integer> l2 = new ArrayList<Integer>();

System.out.println(l1.getClass() == l2.getClass());

打印的结果为 true 是因为 List<String>和 List<Integer>在 jvm 中的 Class 都是 List.class。

泛型信息被擦除了。

可能同学会问，那么类型 String 和 Integer 怎么办？

答案是泛型转译。

public class Erasure <T>{

T object;

public Erasure(T object) {

this.object = object;

}

}

Erasure 是一个泛型类，我们查看它在运行时的状态信息可以通过反射。

Erasure<String> erasure = new Erasure<String>("hello");

Class eclz = erasure.getClass();

System.out.println("erasure class is:"+eclz.getName());

打印的结果是

erasure class is:com.frank.test.Erasure

Class 的类型仍然是 Erasure 并不是 Erasure<T>这种形式，那我们再看看泛型类中 T 的类型在 jvm 中是什么具体类型。

Field[] fs = eclz.getDeclaredFields();

for ( Field f:fs) {

System.out.println("Field name "+f.getName()+" type:"+f.getType().getName());

}

打印结果是

Field name object type:java.lang.Object

**那我们可不可以说，泛型类被类型擦除后，相应的类型就被替换成 Object 类型呢？**

**这种说法，不完全正确**。

我们更改一下代码。

public class Erasure <T extends String>{

// public class Erasure <T>{

T object;

public Erasure(T object) {

this.object = object;

}

}

现在再看测试结果：

Field name object type:java.lang.String

**我们现在可以下结论了，在泛型类被类型擦除的时候，之前泛型类中的类型参数部分如果没有指定上限，如 <T>则会被转译成普通的 Object 类型，如果指定了上限如 <T extends String>则类型参数就被替换成类型上限**。

所以，在反射中。

public class Erasure <T>{

T object;

public Erasure(T object) {

this.object = object;

}

public void add(T object){

}

}

add() 这个方法对应的 Method 的签名应该是 Object.class。

Erasure<String> erasure = new Erasure<String>("hello");

Class eclz = erasure.getClass();

System.out.println("erasure class is:"+eclz.getName());

Method[] methods = eclz.getDeclaredMethods();

for ( Method m:methods ){

System.out.println(" method:"+m.toString());

}

打印结果是

method:public void com.frank.test.Erasure.add(java.lang.Object)

也就是说，如果你要在反射中找到 add 对应的 Method，你应该调用 getDeclaredMethod("add",Object.class)否则程序会报错，提示没有这么一个方法，原因就是类型擦除的时候，T 被替换成 Object 类型了。

### 2,类型擦除带来的局限性

**类型擦除，是泛型能够与之前的 java 版本代码兼容共存的原因。但也因为类型擦除，它会抹掉很多继承相关的特性，这是它带来的局限性**。

理解类型擦除有利于我们绕过开发当中可能遇到的雷区，同样理解类型擦除也能让我们绕过泛型本身的一些限制。比如

正常情况下，因为泛型的限制，编译器不让最后一行代码编译通过，因为类似不匹配，但是，基于对类型擦除的了解，利用反射，我们可以绕过这个限制。

public interface List<E> extends Collection<E>{

boolean add(E e);

}

上面是 List 和其中的 add() 方法的源码定义。

因为 E 代表任意的类型，所以类型擦除时，add 方法其实等同于

boolean add(Object obj);

那么，利用反射，我们绕过编译器去调用 add 方法。

public class ToolTest {

public static void main(String[] args) {

List<Integer> ls = new ArrayList<>();

ls.add(23);

// ls.add("text");

try {

Method method = ls.getClass().getDeclaredMethod("add",Object.class);

method.invoke(ls,"test");

method.invoke(ls,42.9f);

} catch (NoSuchMethodException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (SecurityException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IllegalAccessException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IllegalArgumentException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (InvocationTargetException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

for ( Object o: ls){

System.out.println(o);

}

}

}

打印结果是：

23

test

42.9

可以看到，利用类型擦除的原理，用反射的手段就绕过了正常开发中编译器不允许的操作限制。

泛型中值得注意的地方

泛型类或者泛型方法中，不接受 8 种基本数据类型。

所以，你没有办法进行这样的编码。

List<int> li = new ArrayList<>();

List<boolean> li = new ArrayList<>();

需要使用它们对应的包装类。

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

List<Boolean> li1 = new ArrayList<>();

对泛型方法的困惑

public <T> T test(T t){

return null;

}

**有的同学可能对于连续的两个 T 感到困惑，其实 <T>是为了说明类型参数，是声明,而后面的不带尖括号的 T 是方法的返回值类型。**

你可以相像一下，如果 test() 这样被调用

test("123");

那么实际上相当于

public String test(String t);

**Java 不能创建具体类型的泛型数组**

这句话可能难以理解，代码说明。

List<Integer>[] li2 = new ArrayList<Integer>[];

List<Boolean> li3 = new ArrayList<Boolean>[];

这两行代码是无法在编译器中编译通过的。原因还是类型擦除带来的影响。

List<Integer>和 List<Boolean>在 jvm 中等同于List<Object>，所有的类型信息都被擦除，程序也无法分辨一个数组中的元素类型具体是 List<Integer>类型还是 List<Boolean>类型。

但是，

List<?>[] li3 = new ArrayList<?>[10];

li3[1] = new ArrayList<String>();

List<?> v = li3[1];

借助于无限定通配符却可以，前面讲过 ？代表未知类型，所以它涉及的操作都基本上与类型无关，因此 jvm 不需要针对它对类型作判断，因此它能编译通过，但是，只提供了数组中的元素因为通配符原因，它只能读，不能写。比如，上面的 v 这个局部变量，它只能进行 get() 操作，不能进行 add() 操作，这个在前面通配符的内容小节中已经讲过。

### 3,泛型，并不神奇

**我们可以看到，泛型其实并没有什么神奇的地方，泛型代码能做的非泛型代码也能做。**

**而类型擦除，是泛型能够与之前的 java 版本代码兼容共存的原因。**

**可量也正因为类型擦除导致了一些隐患与局限。**

**但，我还是要建议大家使用泛型，如官方文档所说的，如果可以使用泛型的地方，尽量使用泛型。**

**毕竟它抽离了数据类型与代码逻辑，本意是提高程序代码的简洁性和可读性，并提供可能的编译时类型转换安全检测功能。**

类型擦除不是泛型的全部，但是它却能很好地检测我们对于泛型这个概念的理解程度。

我在文章开头将泛型比作是一个守门人，原因就是他本意是好的，守护我们的代码安全，然后在门牌上写着出入的各项规定，及“xxx 禁止出入”的提醒。但是同我们日常所遇到的那些门卫一般，他们古怪偏执，死板守旧，我们可以利用反射基于类型擦除的认识，来绕过泛型中某些限制，现实生活中，也总会有调皮捣蛋者能够基于对门卫们生活作息的规律，选择性地绕开他们的监视，另辟蹊径溜进或者溜出大门，然后扬长而去，剩下守卫者一个孤独的身影。

所以，我说泛型，并不神秘，也不神奇。