**关于java反射泛型的研究**

2017/3/12

裴卫

目录

[概述 3](#_Toc481658218)

[泛型 3](#_Toc481658219)

[1.Java的泛型是类型擦除的 3](#_Toc481658220)

[2.不能初始化泛型参数和数组 3](#_Toc481658221)

[3.强制声明泛型的实际类型 4](#_Toc481658222)

[4.注意泛型没有继承说 4](#_Toc481658223)

[5.不同场景使用不同的通配符 5](#_Toc481658224)

[6.建议采用顺序为List<T> List<?> List<Object> 5](#_Toc481658225)

[7.使用多重边界限定 5](#_Toc481658226)

[反射 6](#_Toc481658227)

[1.注意Class类的特殊性 6](#_Toc481658228)

[2.适时选择getDeclaredXXX和getXXX 6](#_Toc481658229)

[3.反射访问属性或方法时将Accessible设置为true 6](#_Toc481658230)

[4.使用forName动态加载类 7](#_Toc481658231)

[5.使用反射让模板方法更强大 7](#_Toc481658232)

[6.不要过分关注反射的效率 8](#_Toc481658233)

[泛型的使用 9](#_Toc481658234)

[1.自定义泛型类和泛型方法 9](#_Toc481658235)

[（1）定义带类型参数的类 9](#_Toc481658236)

[（2）定义带类型参数方法 9](#_Toc481658237)

[2.泛型类和泛型方法的使用 10](#_Toc481658238)

[（1）对带类型参数的类进行赋值 10](#_Toc481658239)

[（2）对带类型参数的方法进行赋值 11](#_Toc481658240)

[（3）对通配符类型参数进行赋值 12](#_Toc481658241)

[3.类型通配符上限与下限 12](#_Toc481658242)

[4.注意事项 13](#_Toc481658243)

[5.Class泛型的使用 13](#_Toc481658244)

[（1）定义接口： 14](#_Toc481658245)

[（2）定义查询方法： 14](#_Toc481658246)

[（3）调用（匿名类方案 ）： 14](#_Toc481658247)

# 概述

**在Java5后推出了泛型,使我们在编译期间操作集合或类时更加的安全,更方便代码的阅读,而让身为编译性语言的Java提供动态性的反射技术,更是在框架开发中大行其道,从而让Java活起来,下面看一下在使用泛型和反射需要注意和了解的事情**

# 泛型

## 1.Java的泛型是类型擦除的

**Java中的泛型是在编译期间有效的,在运行期间将会被删除,也就是所有泛型参数类型在编译后都会被清除掉.请看以下例子**

**Java代码**

**public static void test(List<Integer> testParameter) {**

**}**

**public static void test(List<String> testParameter) {**

**}**

**以上是一个最最简单的重载例子,在编译后泛型类型是会被擦除的,所以无论是List<Integer>或List<String>都会变成List<E>,所以参数相同,重载不成立,无法编译通过,而且我们可以尝试将不同泛型的类getClass()看看,他们的Class是一样的**

## 2.不能初始化泛型参数和数组

**请观察以下代码**

**Java代码**

**class Test<T> {**

**//编译不通过**

**private T t = new T();**

**//编译不通过**

**private T[] tArray = new T[5];**

**//编译通过**

**private List<T> list = new ArrayList<T>();**

**}**

**Java语言是一门强类型,编译型的安全语言,它需要确保运行期的稳定性和安全性,所以在编译时编译器需要严格的检查我们所声明的类型,在编译期间编译器需要获取T类型,但泛型在编译期间已经被擦除了,所以new T()和new T[5]都会出现编译错误,而为什么ArrayList却能成功初始化?这是因为ArrayList表面是泛型,但在编译期间已经由ArrayList内部转型成为了Object,有兴趣可以看一下ArrayList的源码,源码中容纳元素的是private transient Object[] elementData,是Object类型的数组**

## 3.强制声明泛型的实际类型

**在使用泛型时,在大多数情况下应该声明泛型的类型,例如该集合是存放User对象的,就应该使用List<User>来声明,这样可以尽量减少类型的转换,若只使用List而不声明泛型,则该集合等同于List<Object>**

## 4.注意泛型没有继承说

**请看以下例子**

**Java代码**

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

**// 编译不通过**

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

**// or**

**List<Object> list = new ArrayList<Double>();**

**}**

**上面代码的业务逻辑为有一组元素,但我不确定元素时整形还是浮点型,当我在确定后创建对应的泛型实现类,刚接触泛型的读者有可能会犯以上错误,理解为泛型之间是可以继承的,例如声明Object的泛型实际上创建Integer泛型,错误认为泛型之间也存在继承关系,但这是不正确的,泛型是帮助开发者编译期间类型检查安全.我们可以换种方式实现以上业务逻辑**

**Java代码**

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

**//Number 为Integer 和 Double 的公共父类**

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

**//使用通配符,代表实际类型是Number类型的子类**

**List<? extends Number> numberList2 = new ArrayList<Integer>();**

**//or**

**List<? extends Number> numberList3 = new ArrayList<Double>();**

**}**

## 5.不同场景使用不同的通配符

**泛型当中支持通配符,可以单独使用 '?' 来表示任意类型,也可以使用刚才上面例子中的extends关键字表示传入参数是某一个类或接口的子类,也可以使用super关键字表示接受某一个类型的父类型,extends和super的写法一样,extends是限定上界,super为限定下界**

**? extends T : T的子类  
? super T ： T的父类**

## 6.建议采用顺序为List<T> List<?> List<Object>

**以上三者都可以容纳所有的对象,但使用的顺序应该是首选List<T>,然后List<?>,最后才使用List<Object>,原因是List<T>是确定为某一类型的,安全的,也是Java所推荐的,而List<?>代表为任意类型,与List<T>类似,而List<Object>中全部为Object类型,这与不使用泛型无异,而List<T>是可读可写的,因为已经明确了T类型,而其他两者在读取时需要进行向下转型,造成了不必要的转型**

## 7.使用多重边界限定

**现在有一个业务需求,收钱时需要是我们本公司的员工(继承User),同时亦需要具备收银员的功能(实现Cashier),此时我们当然可以想到接受1个User然后判断User是否实现了Cashier接口,但在泛型的世界中,我们可以怎么做?请看以下例子**

**Java代码**

**public static <T extends User & Cashier> void CollectMoney(T t){**

**}**

**以上例子就表明,限定了上界,只要是User和Cashier的子类才可以作为参数传递到方法当中**

# 反射

## 1.注意Class类的特殊性

**Java语言是先把Java源文件编译成后缀为class的字节码文件,然后通过ClassLoader机制把类文件加载到内存当中,最后生成实例执行的,在描述一个类时,Java使用了一个元类来对类进行描述,这就是Class类,他是一个描述类的类,所以注定Class类是特殊的**

1. **Class类无构造函数,Class对象加载时由JVM通过调用类加载器的**

**defineClass方法来构造Class对象**

1. **Class类还可以描述基本数据类型,由于基本类型并不是Java中的对象,它们**

**一般存在于栈,但Class仍然可以对它们进行描述,例如使用int.class**

1. **其对象都是单例模式,一个Class的实现对象描述一个类,并且只描述一个类**

**所以只要是该被描述的类所有对象都只有一个Class实例**

1. **Class类是Java反射的入口**

## 2.适时选择getDeclaredXXX和getXXX

**Class类中提供了很多getDeclaredXXX和getXXX的方法,请看以下例子**

**Java代码**

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

**Class<User> cls = User.class;**

**//获取类方法**

**cls.getDeclaredMethods();**

**cls.getMethods();**

**//获取类构造函数**

**cls.getDeclaredConstructors();**

**cls.getConstructors();**

**//获取类属性**

**cls.getDeclaredFields();**

**cls.getFields();**

**}**

**getXXX的方式是获取所有公共的(public)级别的,包括从父类继承的方法,而getDeclaredXXX的方式是获取所有的,包括公共的(public),私有的(private),不受限与访问权限,**

## 3.反射访问属性或方法时将Accessible设置为true

**在调用构造函数或方法的invoke前检查accessible已经是公认的写法,例如以下代码**

**Java代码**

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

**Class<User> cls = User.class;**

**//创建对象**

**User user = cls.newInstance();**

**//获取test方法**

**Method method = cls.getDeclaredMethod("test");**

**//检查Accessible属性**

**if(!method.isAccessible()){**

**method.setAccessible(true);**

**}**

**method.invoke(user);**

**}**

**可以尝试获取Class的getMethod,也就是公开的方法,再输出isAccessible,可以看到输出的其实也是false,其实因为accessible属性的语义并不是我们理解的访问权限,而是指是否进行安全检查,而安全监察是非常消耗资源的,所以反射提供了Accessible可选项,让开发者逃避安全检查,有兴趣的读者可以查看AccessibleObject类观察其源码了解安全检查.**

## 4.使用forName动态加载类

**forName相信各位读者不会陌生,在使用JDBC时要动态加载数据库驱动就是使用forName的方式进行加载,同时亦可以从外部配置文件中读取类的全路径字符串进行加载,在使用forName时,被加载的类就会被加载到内存当中,只会加载类,并不会执行任何代码,而我们的数据库驱动就是利用static代码块来执行操作的,因为当类被加载到内存中时,会执行static代码块**

## 5.使用反射让模板方法更强大

**模板方法的定义是,定义一个操作的算法骨架,将某些步骤延迟到子类当中实现,而实现细节由子类决定,父类只决定骨架,以下是一个传统模板方法的事例**

**Java代码**

**public abstract class Test {**

**public final void doSomething() {**

**System.out.println("start...");**

**doInit();**

**System.out.println("end.....");**

**}**

**protected abstract void doInit();**

**}**

**此时子类只需要继承Test类实现doInit()方法即可嵌入到doSomething中,现在我们有一个需求,若我在doSomething中需要调用一系列的方法才能完成doSomething呢?而且我调用方法的数量并不确定,只需遵从某些规则则可将方法添加到doSomething方法当中.请看以下代码**

**public abstract class Test {**

**public final void doSomething() throws Exception {**

**Method[] methods = this.getClass().getDeclaredMethods();**

**System.out.println("start...");**

**for (Method method : methods) {**

**if (this.checkInitMethod(method)) {**

**method.invoke(this);**

**}**

**}**

**System.out.println("end.....");**

**}**

**private boolean checkInitMethod(Method method) {**

**// 方法名初始是否为init**

**return method.getName().startsWith("init")**

**// 是否为public修饰**

**&& Modifier.isPublic(method.getModifiers())**

**// 返回值是否为void**

**&& method.getReturnType().equals(Void.TYPE)**

**// 是否没有参数**

**&& !method.isVarArgs()**

**// 是否抽象类型**

**&& !Modifier.isAbstract(method.getModifiers());**

**}**

**}**

**看到上面的代码,读者是否有似曾相识的感觉?在使用Junit3时是不是只要遵守方法的签名约定,就能成为测试类?使用这种反射可以让模板方法更强大,下次需要使用多个方法在模板方法中时,不要创建多个抽象方法,尝试使用以上方式**

## 6.不要过分关注反射的效率

**反射的效率是一个老生常谈的问题,普通的调用方法,创建类,在反射的情况下需要调用诸多API才能实现,效率当然要比普通情况下低下,但在项目当中真正引起性能问题的地方,绝大数不会是由反射引起的,而反射带给我们的却是如此的美妙,当今的各系列开源框架,几乎都存在反射的身影,而且大量存在更比比皆是。**

# 泛型的使用

## 1.自定义泛型类和泛型方法

### （1）定义带类型参数的类

package com.soft;

public class MM<T, S extends T> {

public T say(){

T t=null;

return t;

}

}

**规则：**在定义带类型参数的类时，在紧跟类命之后的<>内，指定一个或多个类型参数的名字，同时也可以对类型参数的取值范围进行限定，多个类型参数之间用","进行分隔。

**说明：**定义完类型参数后，可以在类中定义位置之后的几乎任意地方使用类型参数（静态块，静态属性，静态方法除外），就像使用普通的类型一样。

**注意：**父类定义的类型参数不能被子类继承。

### （2）定义带类型参数方法

package com.soft;

public class WW {

public <T> void write(T t, T[] ta){

}

public <S> void read(S s, S[] sa){

}

public <T, S extends T> T test(T t, S s){

S ss=null;  
　　　　 return ss;

}

}

**规则：**在定义带类型参数的方法时，在紧跟可见范围修饰（如public）之后的<>内，指定一个或多个类型参数的名字，同时也可以对类型参数的取值范围进行限定，多个类型参数之间用","进行分隔。

**说明：** 定义完带类型参数的方法后，可以在方法中定义位置之后的任意地方使用类型参数，就像使用普通的类型一样。

**注意：**定义带类型参数的方法，其主要目的是为了表达多个参数以及返回值之间的关系。例如本例中T和S为继承关系，返回值的类型（T）和第一个类型参数（t）的类型（T）相同。

**补充：**

当我们需要一个在逻辑上可以用来表示同时是List<Integer>和List<Number>的类型时，类型通配符应运而生。类型通配符一般是使用"?"代替具体的类型实参。

如果仅仅是想实现**多态**，请优先使用**通配符**解决。

初始代码如下：

public <T> void test1(List<T> u){

...

}

public <S> void test2(List<S> u){

...

}

使用通配符的代码如下：

public void test(List<？> s){

...

}

## 2.泛型类和泛型方法的使用

### （1）对带类型参数的类进行赋值

 对带类型参数的类进行赋值有两种方式：

#### 声明类变量或者实例化时

如下：

List<String> list;

list = new ArrayList<String>();

如下赋值方式会报错：

List<Integer> a = new ArrayList<Integer>(712);

List<Number> b = a;

List<Number>不能视为List<Integer>的父类，请参照前面“序言->变化一”中的描述。

#### 继承类或者实现接口时

如下：

public class MyList<E> extends ArrayList<E> implements List<E> {...}

### （2）对带类型参数的方法进行赋值

当调用泛型方法时，编译器自动对类型参数进行赋值，当不能成功赋值时报编译错误，如下：

package com.soft;

import java.util.List;

public class WW {public <T, S extends T> T test(T t, S s){

return s;

}

public <T> T test1(T t, List<T> list){

return t;

}

public <T> T test2(List<T> list1, List<T> list2){

return null;

}

public void common(){

Object o = null;

Integer i = null;

Number n = null;

test(n, i);//此时T为Number, S为Integer

test(o, i);//T为Object, S为Integer

List<Integer> list1 = null;

List<Number> list2 = null;

test1(i, list1);//此时T为Integer，i为Integer类型，list1为List<Integer>类型

test1(i, list2);//此时T为Number，i为Integer类型，list2为List<Number>类型

test2(list1, list2);//编译报错

}

}

### （3）对通配符类型参数进行赋值

在上面两小节中，对是类型参数赋予具体的值，除此，还可以对类型参数赋予不确定值

假设有如下定义：

List<?> unknownList;

List<? extends Number> unknownNumberList;

List<? super Integer> unknownBaseLineIntgerList;

现对上述定义的变量赋值，如下：

List<String> listString = null;

unknownList = listString;

## 3.类型通配符上限与下限

有时候，我们会听到类型通配符上限和类型通配符下限这样的说法，其实在前面的例子中有下面这样一小段代码，在该代码中已经体现出一些上下限的思想

public <T, S extends T> T test(T t, S s){

S ss=null;

　　 return ss;

}

此处我们进一步说明

package com.soft;

public class GenericTest {

public static void main(String[] args) {

Student<String> name = new Student<String>("xiaoming");

Student<Integer> age = new Student<Integer>(24);

Student<Number> number = new Student<Number>(1001);

getData(name);

getData(age);

getData(number);

getUpperNumberData(name);//编译报错

getUpperNumberData(age);

getUpperNumberData(number);

}

public static void getData(Student<?> data) {

System.out.println("data :" + data.getData());

}

public static void getUpperNumberData(Student<? extends Number> data){  
//此处限定类型实参只能是Number类及其子类

System.out.println("data :" + data.getData());

}

}

class Student<T> {

private T data;

public Student() {

}

public Student(T data) {

this.data = data;

}

public T getData() {

return data;

}

}

我想大家通过上例肯定能体会到什么是类型通配符上限，类型通配符上限通过形如Student<? extends Number>形式定义，相对应的，类型通配符下限为Student<? super Number>形式，其含义与类型通配符上限正好相反，在此不作过多阐述了。

## 4.注意事项

**A、**当对类或方法的类型参数进行赋值时，要求对所有的类型参数进行赋值，否则，将得到一个编译错误。

**B、**在Java集合框架中，对于参数值是未知类型的容器类，只能读取其中的元素，不能向其中添加元素，因为其类型是未知，编译器无法识别添加元素的类型和容器的类型是否兼容，唯一的例外是NULL

**C、**可以使用带泛型参数的类声明数组，却不可以用于创建数组

List<Integer>[] iListArray;

iListArray=new ArrayList[10];

iListArray=new ArrayList<Integer>[10];//编译时错误

## 5.Class泛型的使用

　　Java5中的一个变化是类java.lang.Class是泛型化的（Class<T>），这是把泛型扩展到容器类之外的一个很有意思的例子。 你很可能会问，Class的类型参数T代表什么？它代表Class对象代表的类型，比如说，Class对象String.class的类型为Class<String>，Class对象Serializable.class的类型为Class<Serializable>，此处T代表什么请自行体会。

　　Java5为什么会有这样的变化，这可以被用来提高你反射代码的类型安全。特别的，因Class的newInstance()方法返回一个T，你可以在使用反射创建对象时得到更精确的类型。比如说，假定你要写一个工具方法来进行数据库查询，给定一个SQL语句，请返回一个数据库中符合查询条件的对象集合(collection)，你该怎么做？

解决方案：

### （1）定义接口：

interface Factory<T> {

public T[] make();

}

### （2）定义查询方法：

public <T> Collection<T> select(Factory<T> factory, String statement) {

Collection<T> result = new ArrayList<T>();

/\* run sql query using jdbc \*/

for ( int i=0; i<10; i++ ) { /\* iterate over jdbc results \*/

T item = factory.make();

/\* use reflection and set all of item’s fields from sql results \*/

result.add( item );

}

return result;

}

### （3）调用（匿名类方案 ）：

select(new Factory<EmpInfo>(){

public EmpInfo make() {

return new EmpInfo();

}

} , ”selection string”);

除了上述方式外，也可以声明一个类来支持Factory接口：

class EmpInfoFactory implements Factory<EmpInfo> { ...

public EmpInfo make() { return new EmpInfo();}

}

select(getMyEmpInfoFactory(), "selection string");//调用语句，其中getMyEmpInfoFactory()方法返回自定义的支持Factory接口的类

解说：上述两种解决方案虽然可行，却很不自然，其缺点是它们需要下面的二者之一：

　　 a.为每个要使用的类型定义冗长的匿名工厂类；

　　 b.为每个要使用的类型声明一个工厂类并传递其对象给调用处

此时可以选择使用class类型参数，它可以被反射使用，没有泛型的代码可能如下：

Collection emps = sqlUtility.select(EmpInfo.class, ”select \* from emps”);

public static Collection select(Class c, String sqlStatement) {

Collection result = new ArrayList();

/\* run sql query using jdbc \*/

for ( /\* iterate over jdbc results \*/ ) {

Object item = c.newInstance();

/\* use reflection and set all of item’s fields from sql results \*/

result.add(item);

}

return result;

}

上述方案虽然能达到目的，却不能给我们返回一个精确类型的集合，因Java5及以后版本中Class是泛型的，我们可以按下面这种方式写：

Collection<EmpInfo> emps=sqlUtility.select(EmpInfo.class, ”select \* from emps”);

public static <T> Collection<T> select(Class<T>c, String sqlStatement) {

Collection<T> result = new ArrayList<T>();

/\* run sql query using jdbc \*/

for ( /\* iterate over jdbc results \*/ ) {

T item = c.newInstance();

/\* use reflection and set all of item’s fields from sql results \*/

result.add(item);

}

return result;

}