站在巨著上谈泛型

第一节：Java 泛型的起源

人总是懒惰的，总想着一劳永逸。程序员也不例外，还美其名曰：追求程序的通用性。这种追求是孜孜不倦的，前仆后继的。梳理一下人们所进行的努力吧：

第一阶段：在类和方法中，人们只能使用具体的类型：要么是基本类型，要么是自定义的类。这一阶段，程序的通用性很差。

第二阶段：在面向对象编程语言中，使用多态机制。例如，可以将方法的参数类型设为基类，那么该方法就可以接受从这个基类中导出的任何类作为参数。这样的方法更加通用一些，可应用的地方也多一些。但是，需要注意的是 final 类不能扩展。

第三阶段：有时候，拘泥于单继承体系，也会使程序受限太多。如果方法的参数是一个接口而不是一个类，这种限制就放松了很多。因为任何实现了该接口的类都满足该方法。这个阶段，使用了面向接口的编程思想，让程序的通用性更强大了。

第四阶段：有时候，即使是使用了接口，对程序的约束也还是太强了，因为一旦指明了接口，它就要求你的代码必须使用特定的接口。而我们希望达到的目的是编写更通用的代码，要是代码应用于“某种不具体的类型”，而不是一个具体的接口或者类。这种“不具体的类型”，属于参数化类型，也就是说：把类型当做参数。当定义的时候，我们把类型当做一个参数；在使用的时候，我们才将这个参数用具体的类型来代替。为了实现参数化类型的目标，从而出现了泛型的机制。

第二节：泛型的用武之处

人们为什么要发明泛型呢？最引人注目的原因，应该就是为了创造容器类。容器，就是存放要使用对象的地方。数组也是如此，只不过数组非常简单。而容器类更加灵活，具备更多不同的功能。可能这么说，大家不一定都理解了。我再细说一下吧：

（1）泛型 T 的作用就是：你提前定义一个类型，然后让编译器替你转型。让你转型的话，你可能随意转型，会发生错误。而编译器能避免这种错误的发生。而使用 Object 只能开发者自己转型，有发生错误的风险

（2）可以将容器分类。分为以读容器和写容器。

第三节：泛型的表示形式

3.1 <T>：定义类型 T

3.2 T：使用类型 T

```
<T> T print(String str)
```

这个写法是非常合理的，如果写成这样：`T print(String str)` 会造成：T 是什么，令人费解，突然使用 T 感觉很突兀。而加上<T>，定义了一个泛型 T。在 Java 中，任何事物都遵守一个基本原则：先定义，后使用。泛型也不例外。

同样道理，这种写法也不错：`t.<T> print(st);`

3.3 <?>：表示通配符

表示使用了泛型。如果不使用泛型，编译器会提示的。如下面所示：

```
public class Test
```

```
{
    public static void main(String[] args)
    {
        @SuppressWarnings(
        { "rawtypes", "unchecked" })
        List list1 = new ArrayList<String>();
        list1.add("");

        List<?> list2 = new ArrayList<String>();
        list2.add("");
    }
}
```

List list1 表示 Object 类型的 List，这个不是泛型，所以需要压制警告。
List<?> list2 表示是泛型，类型是某个类型，虽然不需要压制警告，但是不能往里面存放东西。

3.4 区别

看下面的六个类，红字标出：

```
class WeakReference<T> extends Reference<T>

class ThreadLocal<T>

class ThreadLocalMap{
    class Entry extends WeakReference<ThreadLocal<?>>
}
```

问题：为什么使用 ThreadLocal<?>，而不是使用 ThreadLocal<T>呢？

第四节：泛型的种类

4.1 泛型类

```
public class Test<T>//定义类型T
{
    T t;//使用类型T

    public void println(T t)
    {
        System.out.println(t);
    }
}
public class Test<T, U>
{
    T t;
    U u;
```

```
    public void println(T t, U u)
    {
        System.out.println(t);
        System.out.println(u);
    }
}
```

4.2 泛型接口

```
public interface Test<T>
{
    public void println(T t);
}

public class SubTest<T> implements Test<T>
{

    public void println(T t)
    {

    }
}
```

4.3 泛型方法

泛型可以用在类上，也可以用在方法上，方法所在的类可以是泛型类，也可以不是泛型类。

定义泛型方法，要将泛型参数列表置于返回值之前。如下所示：

```
public class Test
{
    public <T> void println(T t)
    {
        System.out.println(t);
    }
}
public class Test
{
    public <T, U> void println(T t, U u)
    {
        System.out.println(t);
        System.out.println(u);
    }
}
```

类型推断和显示类型声明

```
public class Test
{
    public <T> void println(T t)
    {
        System.out.println(t);
```

```
    }

    public static void main(String[] args)
    {
        Test test = new Test();
        String str = "hello world!";
        test.println(str);//类型推断
        test.<String> println(str);//显示类型声明
    }
}
```

当使用泛型类时候，必须在创建对象的时候指定类型参数的值，而使用泛型方法的时候，通常不必指明参数类型，因为编译器会为我们找到具体的类型，这称为类型参数推断。因为，我们可以像调用普通方法一样调用 println 方法。

注意：

类型推断只对赋值操作有效，其他时候并不起作用。如果你将一个泛型方法调用的结果作为参数，传递给另外一根方法，这个时候编译器并不会执行类型推断。在这种情况下，编译器认为：调用泛型方法后，返回值被赋予一个 object 类型的变量。

4.4 总结

第五节：<?>和<Object>的区别

第一：?只用在声明类型(声明类与接口属性或者函数参数)的时候，无法用在实例化对象的时候，而 Object 可以用在声明类型时候，也可以用在实例化对象时候。如下面代码所示：

```
public class Test
{
    public static void main(String[] args)
    {
        List<?> list1 = new ArrayList<?>();//编译错误
        List<Object> list2 = new ArrayList<Object>();
    }
}
```

第二：使用?声明的泛型可以接受任意类型的泛型（类似于数组的协变），编译的时候?类型会被擦除，相当于原生类型，最后都转为 Object 类型。如下代码所示：

```
public class Test
{
    public static void main(String[] args)
    {
        //?声明的泛型
        List<?> list1 = null;
        list1 = new ArrayList<String>();
        list1 = new ArrayList<Integer>();
        //原生类泛型
        List list2 = null;
        list2 = new ArrayList<String>();
```

```
        list2 = new ArrayList<Integer>();
        list2 = new ArrayList<Object>();
        //数组的协变
        Object[] array1 = null;
        array1 = new Object[10];
        array1 = new String[10];
    }
}
```

第三：Object 声明的泛型只能接受 Object 参数的泛型。泛型不支持协变。

```
public class Test
{
    public static void main(String[] args)
    {
        List<Object> list = null;
        list = new ArrayList<String>();//编译错误
        list = new ArrayList<Integer>();//编译错误
        list = new ArrayList<Object>();
    }
}
```

第六节：协变与逆变

协变和逆变指的是宽类型和窄类型在某种情况下的替换或交换的特性。简单的说，协变就是用一个窄类型替代宽类型，而逆变则用宽类型覆盖窄类型。

6.1 协变

在 Java 中协变的例子非常常见，例如，面向对象的多态，以及数组的协变特性，下面看一下协变的例子：

```
public class Test
{
    public static void main(String[] args)
    {
        Number num1 = new Integer(0);
        Number[] num2 = new Integer[10];
    }
}
```

而在泛型是不支持协变的，看下面的代码：

```
public class Test
{
    public static void main(String[] args)
    {
        List<Object> list = new ArrayList<String>();//编译错误
        list = new ArrayList<Object>();
    }
}
```

虽然泛型不支持协变的，但是可以通过通配符进行模拟：

```
public class Test
{
    public static void main(String[] args)
    {
        List<? extends Object> list = new ArrayList<String>();
    }
}
```

注意：? extends Object 的含义是：运行 Object 的子类，也包括 Object，作为泛型参数。

6.2 逆变

在 Java 中不允许将父类变量赋值给子类变量。泛型自然也不支持逆变。但是在泛型中可以通过通配符进行模拟，如下例子：

```
public class Test
{
    public static void main(String[] args)
    {
        List<? super Integer> list = new ArrayList<Number>();
    }
}
```

第七节：< ? extends E >和< ? super E >的用法

7.1 关于直觉的错误认识

先看下面的代码：

```
public class Test
{
    public static void main(String[] args)
    {
        Object obj = new String("10");
        @SuppressWarnings("rawtypes")
        List list = new ArrayList<>();
        List<Object> list2 = new ArrayList<String>();
    }
}
```

分析如下：

```
List<Object> list2 = new ArrayList<String>();
```

这句话会出现编译错误，错误提示为：

不兼容的类型：ArrayList<String>无法转换为 List<Object>。

虽然 String 类型的可以赋值给 Object 类型的变量，ArrayList<>类型的可以赋值给 List 类型的变量，这符合人类的直觉，人类在直觉中，上述代码存在着子类和父类的概念。但是 ArrayList<String>类型不能赋值给 List<Object>类型，这违反了人类的直觉。

假设 ArrayList<String>可以赋值给 List<Object>类型，那么 list2 里面的元素就是 Object 类型了，这样就可以被其他别有用心的人任意转换类型了，而这种转换往往是不成功的，所以为了避免不必要的错误发生，只能拒绝将 ArrayList<String>类型赋值给

List<Object>类型。我想着应该就是 ArrayList<String>无法转换为 List<Object>的原因 吧。

```
Object obj = new String("10");
```

虽然上面的代码能够赋值成功，但是本身存在着隐患，可以看下面的代码：

```
public class Test
{
    public static void main(String[] args)
    {
        Object obj = new String("10");
        Integer i = (Integer) obj;//转型失败，String cannot be cast to Integer
    }
}
```

上述代码存在着隐患。

7.2 容器的分类

从字面来看：<？extends E >和<？super E >，我们可以发现两个信息：

（1）子类和父类的信息。在我们的知识体系中，子类可以向上转型成父类，而父类不可以转换为子类。

（2）泛型的信息。在我们的知识体系中，泛型是使用在容器当中的，而在使用容器的过程中，基本操作就是存操作和取操作。

总之，<？extends E >和<？super E >这个概念提出的背景应该是：在使用容器的存取过程中，结合子类和父类相互转化的原则，把容器划分了类。

下面从容器归类的角度来看下面的代码。首先看的一类容器是<？extends E >类型的容器。这类容器存的都是 E 的子类。

```
public class Test
{
    public static void main(String[] args)
    {
        //容器list存放的都是Number的子类
        List<? extends Number> list = new ArrayList<Integer>();

        //存操作，因为无法确定到底存入Number的哪个子类，所以无法存入具体的数值，只能存入null
        list.add(null);

        //取操作，因为已经确定容器中存在的都是Number的子类，所有这些子类都可以转型成其父类Number，所以可以取出number类型数据
        Number number = list.get(0);

    }
}
```

接着看的一类容器是<？super E >类型的容器。这类容器存的都是 E 的父类。

```
public class Test
{
    public static void main(String[] args)
```

```
    {
        //容器list存放的都是Integer的父类
        List<? super Integer> list = new ArrayList<Number>();

        //存操作，因为可以确定存入的都是Integer的父类，所以可以存入Integer，然后向上转型其父类
        list.add(new Integer(10));

        //取操作，因为无法确定容器中存的是Integer的那个父类，所以只能取出Object类型的数据。
        Object obj = list.get(0);

    }
}
```

总之，容器分为两类：

（1）以存操作为主的容器，在存的过程中，把子类转换为父类，容器里面存的是父类，所以使用< ? super E >类型的容器。

（2）以取操作为主的容器，在取的过程中，把子类转换为父类，容器里面存的都是子类，所以使用< ? extends E >类型的容器。

第八节：泛型实例化

类型可以通过参数来实现，例如泛型中的 T t，但是我想生成 T 对象，那怎么实现呢？按照以往的经验，我们很容易想到这种方式：

```
public class Test<T>
{
    T t;

    public void create()
    {
        t = new T();//编译错误：Cannot instantiate the type T
    }

}
```

此时，由于 T 的具体类型我们无法获得，所以 new T()是无法通过编译的。换一种思路，我们不妨使用反射机制，通过 T 的 Class 对象的 newInstance()方法来获取它的实例，而 T 的 class 对象表示为：Class<T>，所以生成 T 对象的代码如下所示：

```
public class Test<T>
{
    T t;

    public void create(Class<T> clazz)
    {
        try
        {
            t = clazz.newInstance();
        }
        catch (InstantiationException e)
```

```
        {
            e.printStackTrace();
        }
        catch (IllegalAccessException e)
        {
            e.printStackTrace();
        }
    }
}
```