**介绍**

　　在Java SE 1.5之前，没有泛型的情况的下，通过对类型Object的引用来实现参数的“任意化”，“任意化”带来的缺点是要做显式的[强制类型转换](http://baike.baidu.com/edit/id=1436058)，而这种转换是要求开发者对[实际参数](http://baike.baidu.com/edit/id=1436058)类型可以预知的情况下进行的。对于[强制类型转换](http://baike.baidu.com/edit/id=1436058)错误的情况，[编译器](http://baike.baidu.com/edit/id=1436058)可能不提示错误，在运行的时候才出现异常，这是一个安全隐患。

　　泛型的好处是在编译的时候检查[类型安全](http://baike.baidu.com/edit/id=1436058)，并且所有的[强制转换](http://baike.baidu.com/edit/id=1436058)都是自动和隐式的，提高代码的重用率。

**规则和限制**

　　1、泛型的类型参数只能是类类型（包括自定义类），不能是简单类型。

　　2、同一种泛型可以对应多个版本（因为参数类型是不确定的），不同版本的[泛型类](http://baike.baidu.com/edit/id=1436058)实例是不兼容的。

　　3、泛型的类型参数可以有多个。

　　4、泛型的参数类型可以使用extends语句，例如<T extends superclass>。习惯上称为“有界类型”。

　　5、泛型的参数类型还可以是[通配符](http://baike.baidu.com/edit/id=1436058)类型。例如Class<?> classType = Class.forName("java.lang.String");

　 　泛型还有接口、方法等等，内容很多，需要花费一番功夫才能理解掌握并熟练应用。在此给出我曾经了解泛型时候写出的两个例子（根据看的印象写的），实现同 样的功能，一个使用了泛型，一个没有使用，通过对比，可以很快学会泛型的应用，学会这个基本上学会了泛型70%的内容。

　　例子一：使用了泛型

　　class Gen<T> {

　　private T ob; //定义泛型[成员变量](http://baike.baidu.com/edit/id=1436058)

　　public Gen(T ob) {

　　this.ob = ob;

　　}

　　public T getOb() {

　　return ob;

　　}

　　public void setOb(T ob) {

　　this.ob = ob;

　　}

　　public void showType() {

　　System.out.println("T的实际类型是: " + ob.getClass().getName());

　　}

　　}

　　public class GenDemo {

　　public static void main(String[] args){

　　//定义[泛型类](http://baike.baidu.com/edit/id=1436058)Gen的一个Integer版本

　　Gen<Integer> intOb=new Gen<Integer>(88);

　　intOb.showType();

　　int i= intOb.getOb();

　　System.out.println("value= " + i);

　　System.out.println("----------------------------------");

　　//定义[泛型类](http://baike.baidu.com/edit/id=1436058)Gen的一个String版本

　　Gen<String> strOb=new Gen<String>("Hello Gen!");

　　strOb.showType();

　　String s=strOb.getOb();

　　System.out.println("value= " + s);

　　}

　　}

　　例子二：没有使用泛型

　　class Gen2 {

　　private Object ob; //定义一个通用类型成员

　　public Gen2(Object ob) {

　　this.ob = ob;

　　}

　　public Object getOb() {

　　return ob;

　　}

　　public void setOb(Object ob) {

　　this.ob = ob;

　　}

　　public void showTyep() {

　　System.out.println("T的实际类型是: " + ob.getClass().getName());

　　}

　　}

　　public class GenDemo2 {

　　public static void main(String[] args) {

　　//定义类Gen2的一个Integer版本

　　Gen2 intOb = new Gen2(new Integer(88));

　　intOb.showTyep();

　　int i = (Integer) intOb.getOb();

　　System.out.println("value= " + i);

　　System.out.println("---------------------------------");

　　//定义类Gen2的一个String版本

　　Gen2 strOb = new Gen2("Hello Gen!");

　　strOb.showTyep();

　　String s = (String) strOb.getOb();

　　System.out.println("value= " + s);

　　}

　　}

　　运行结果：

　　两个例子运行Demo结果是相同的,控制台输出结果如下：

　　T的实际类型是:

　　java.lang.Integer

　　value= 88

　　----------------------------------

　　T的实际类型是: java.lang.String

　　value= Hello Gen!

　　Process finished with exit code 0

　　看明白这个，以后基本的泛型应用和代码阅读就不成问题了。

**逐渐深入泛型**

**1、没有任何重构的原始代码**

　　有两个类如下，要构造两个类的对象，并打印出各自的成员x。

　　public class StringFoo {

　　private String x;

　　public StringFoo(String x) {

　　this.x = x;

　　}

　　public String getX() {

　　return x;

　　}

　　public void setX(String x) {

　　this.x = x;

　　}

　　}

　　public class DoubleFoo {

　　private Double x;

　　public DoubleFoo(Double x) {

　　this.x = x;

　　}

　　public Double getX() {

　　return x;

　　}

　　public void setX(Double x) {

　　this.x = x;

　　}

　　}

　　以上的代码实在无聊，就不写如何实现了。

**2、对上面的两个类进行重构，写成一个类**

　　因为上面的类中，成员和方法的逻辑都一样，就是类型不一样，因此考虑重构。Object是所有类的父类，因此可以考虑用Object做为成员类型，这样就可以实现通用了，实际上就是“Object泛型”，暂时这么称呼。

　　public class ObjectFoo {

　　private Object x;

　　public ObjectFoo(Object x) {

　　this.x = x;

　　}

　　public Object getX() {

　　return x;

　　}

　　public void setX(Object x) {

　　this.x = x;

　　}

　　}

　　写出Demo方法如下：

　　public class ObjectFooDemo {

　　public static void main(String args[]) {

　　ObjectFoo strFoo = new ObjectFoo(new StringFoo("Hello Generics!"));

　　ObjectFoo douFoo = new ObjectFoo(new DoubleFoo(Double("33")));

　　ObjectFoo objFoo = new ObjectFoo(new Object());

　　System.out.println("strFoo.getX="+(StringFoo)strFoo.getX());

　　System.out.println("douFoo.getX="+(DoubleFoo)douFoo.getX());

　　System.out.println("objFoo.getX="+objFoo.getX());

　　}

　　}

　　运行结果如下：

　　strFoo.getX=Hello Generics!

　　douFoo.getX=33.0

　　objFoo.getX=java.lang.Object@19821f

　　解说：在Java 5之前，为了让类有通用性，往往将参数类型、返回类型设置为Object类型，当获取这些返回类型来使用时候，必须将其“强制”转换为原有的类型或者接口，然后才可以调用对象上的方法。

**3、Java1.5泛型来实现**

[强制类型转换](http://baike.baidu.com/edit/id=1436058)很麻烦，我还要事先知道各个Object具体类型是什么，才能做出正确转换。否则，要是转换的类型不对，比如将“Hello Generics!”字符串[强制转换](http://baike.baidu.com/edit/id=1436058)为Double,那么编译的时候不会报错，可是运行的时候就挂了。那有没有不[强制转换](http://baike.baidu.com/edit/id=1436058)的办法----有，改用 Java5泛型来实现。

　　public class GenericsFoo<T> {

　　private T x;

　　public GenericsFoo(T x) {

　　this.x = x;

　　}

　　public T getX() {

　　return x;

　　}

　　public void setX(T x) {

　　this.x = x;

　　}

　　}

　　public class GenericsFooDemo {

　　public static void main(String args[]){

　　GenericsFoo<String> strFoo=new GenericsFoo<String>("Hello Generics!");

　　GenericsFoo<Double> douFoo=new GenericsFoo<Double>(new Double("33"));

　　GenericsFoo<Object> objFoo=new GenericsFoo<Object>(new Object());

　　System.out.println("strFoo.getX="+strFoo.getX());

　　System.out.println("douFoo.getX="+douFoo.getX());

　　System.out.println("objFoo.getX="+objFoo.getX());

　　}

　　}

　　运行结果：

　　strFoo.getX=Hello Generics!

　　douFoo.getX=33.0

　　objFoo.getX=java.lang.Object@19821f

　　和使用“Object泛型”方式实现结果的完全一样，但是这个Demo简单多了，里面没有[强制类型转换](http://baike.baidu.com/edit/id=1436058)信息。

　　下面解释一下上面[泛型类](http://baike.baidu.com/edit/id=1436058)的语法：

　　使用<T>来声明一个类型持有者名称，然后就可以把T当作一个类型代表来声明成员、参数和返回值类型。

　　当然T仅仅是个名字，这个名字可以自行定义。

　　class GenericsFoo<T> 声明了一个[泛型类](http://baike.baidu.com/edit/id=1436058)，这个T没有任何限制，实际上相当于Object类型，实际上相当于 class GenericsFoo<T extends Object>。

　　与Object[泛型类](http://baike.baidu.com/edit/id=1436058)相比，使用泛型所定义的类在声明和构造实例的时候，可以使用“<实际类型>”来一并指定泛型类型持有者的真实类型。类如

　　GenericsFoo<Double> douFoo=new GenericsFoo<Double>(new Double("33"));

　　当然，也可以在构造对象的时候不使用尖括号指定[泛型类](http://baike.baidu.com/edit/id=1436058)型的真实类型，但是你在使用该对象的时候，就需要[强制转换](http://baike.baidu.com/edit/id=1436058)了。比如：GenericsFoo douFoo=new GenericsFoo(new Double("33"));

　　实际上，当构造对象时不指定类型信息的时候，默认会使用Object类型，这也是要[强制转换](http://baike.baidu.com/edit/id=1436058)的原因。

**泛型的高级应用**

**1、限制泛型的可用类型**

　　在上面的例子中，由于没有限制class GenericsFoo<T>类型持有者T的范围，实际上这里的限定类型相当于Object，这和“Object泛型”实质是一样的。限制比如我们要限制T为集合接口类型。只需要这么做：

　　class GenericsFoo<T extends Collection>，这样类中的泛型T只能是[Collection接口](http://baike.baidu.com/edit/id=1436058)的实现类，传入非Collection接口编译会出错。

　　注意：<T extends Collection>这里的限定使用[关键字](http://baike.baidu.com/edit/id=1436058) extends，后面可以是类也可以是接口。但这里的extends已经不是继承的含义了，应该理解为T类型是实现Collection接口的类型，或者T是继承了XX类的类型。

　　下面继续对上面的例子改进，我只要实现了集合接口的类型：

　　public class CollectionGenFoo<T extends Collection> {

　　private T x;

　　public CollectionGenFoo(T x) {

　　this.x = x;

　　}

　　public T getX() {

　　return x;

　　}

　　public void setX(T x) {

　　this.x = x;

　　}

　　}

　　实例化的时候可以这么写：

　　public class CollectionGenFooDemo {

　　public static void main(String args[]) {

　　CollectionGenFoo<ArrayList> listFoo = null;

　　listFoo = new CollectionGenFoo<ArrayList>(new ArrayList());

　　//出错了,不让这么干。

　　// CollectionGenFoo<Collection> listFoo = null;

　　// listFoo=new CollectionGenFoo<ArrayList>(new ArrayList());

　　System.out.println("实例化成功!");

　　}

　　}

　 　当前看到的这个写法是可以编译通过，并运行成功。可是注释掉的两行加上就出错了，因为<T extends Collection>这么定义类型的时候，就限定了构造此类实例的时候T是确定的一个类型，这个类型实现了Collection接口，但是实现 Collection接口的类很多很多，如果针对每一种都要写出具体的子类类型，那也太麻烦了，我干脆还不如用Object通用一下。别急，泛型针对这种 情况还有更好的解决方案，那就是“[通配符](http://baike.baidu.com/edit/id=1436058)泛型”。

**2、通配符泛型**

　 　为了解决类型被限制死了不能动态根据实例来确定的缺点，引入了“通配符泛型”，针对上面的例子，使用通配泛型格式为<? extends Collection>，“？”代表未知类型，这个类型是实现Collection接口。那么上面实现的方式可以写为：

　　public class CollectionGenFooDemo {

　　public static void main(String args[]) {

　　CollectionGenFoo<ArrayList> listFoo = null;

　　listFoo = new CollectionGenFoo<ArrayList>(new ArrayList());

　　//现在不会出错了

　　CollectionGenFoo<? extends Collection> listFoo1 = null;

　　listFoo=new CollectionGenFoo<ArrayList>(new ArrayList());

　　System.out.println("实例化成功!");

　　}

　　}

　　注意：

　　1、如果只指定了<?>，而没有extends，则默认是允许Object及其下的任何Java类了。也就是任意类。

　　2、[通配符](http://baike.baidu.com/edit/id=1436058)泛型不单可以向下限制，如<? extends Collection>，还可以向上限制，如<? super Double>，表示类型只能接受Double及其上层父类类型，如Number、Object类型的实例。

　　3、[泛型类](http://baike.baidu.com/edit/id=1436058)定义可以有多个泛型参数，中间用逗号隔开，还可以定义泛型接口，泛型方法。这些都与[泛型类](http://baike.baidu.com/edit/id=1436058)中泛型的使用规则类似。

**泛型方法**

　　是否拥有泛型方法，与其所在的类是否泛型没有关系。要定义泛型方法，只需将泛型[参数列表](http://baike.baidu.com/edit/id=1436058)置于返回值前。如:

　　public class ExampleA {

　　public <T> void f(T x) {

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

　　}

　　public static void main(String[] args) {

　　ExampleA ea = new ExampleA();

　　ea.f(" ");

　　ea.f(10);

　　ea.f('a');

　　ea.f(ea);

　　}

　　}

　　输出结果：

　　java.lang.String

　　java.lang.Integer

　　java.lang.Character

　　ExampleA

　　使用泛型方法时，不必指明参数类型，[编译器](http://baike.baidu.com/edit/id=1436058)会自己找出具体的类型。泛型方法除了定义不同，调用就像普通方法一样。

　　需要注意，一个static方法，无法访问[泛型类](http://baike.baidu.com/edit/id=1436058)的类型参数，所以，若要static方法需要使用泛型能力，必须使其成为泛型方法。