Java泛型指南

泛型的由来

简介

Java泛型(Generic)是J2 SE1.5中引入的一个新特性，其本质是参数化类型，也就是说所操作的数据类型被指定为一个参数（type parameter）这种参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口、泛型方法。

官方说明：

When you take an element out of a `Collection`, you must cast it to the type of element that is stored in the collection. Besides being inconvenient, this is unsafe. The compiler does not check that your cast is the same as the collection's type, so the cast can fail at run time.

Generics provides a way for you to communicate the type of a collection to the compiler, so that it can be checked. Once the compiler knows the element type of the collection, the compiler can check that you have used the collection consistently and can insert the correct casts on values being taken out of the collection.

参照：https://docs.oracle.com/javase/1.5.0/docs/guide/language/generics.html

官网查找路径：

https://www.oracle.com/cn/java/technologies/java-se-api-doc.html

jdk程序员指南-5.0英文版:https://docs.oracle.com/javase/1.5.0/docs/

New Features and Enhancements:https://docs.oracle.com/javase/1.5.0/docs/relnotes/features.html#generics

Generics:https://docs.oracle.com/javase/1.5.0/docs/guide/language/generics.html

为什么要使用泛型程序设计

泛型程序设计（Generic programming）意味着编写的代码可以被很多不同类型的对象所重用。

存在问题

- 编译期间没有代码检查
- 调用期间发现转换异常

泛型的演变过程

jdk5以前没有泛型的情况

```
package com.naixue.vip.p6.before5;

import java.io.File;

/**
 * @Description 模拟JDK5以前的没有泛型之前对象重用的问题
 * @Author 向寒 奈学教育
 * @Date 2020/7/6 3:04 PM
```

```
 **/
public class ArrayList {

    private int size = 0;

    private Object[] elements = {};

    public Object get(int i) {
        if (size > i) {
            return elements[i];
        }
        throw new IndexOutOfBoundsException();
    }

    public void add(Object o) {
        size++;
        Object[] array = new Object[size];
        for (int i = 0; i < elements.length; i++) {
            array[i]=elements[i];
        }
        array[size-1]=o;
        elements=array;
    }

    public static void main(String[] args) {
        ArrayList arrayList = new ArrayList();
        arrayList.add(1);
        arrayList.add("a");
        // 这里没有错误检查。可以向数组列表中添加任何类的对象
        arrayList.add(new File("/"));

        //对于这个调用，编译和运行都不会出错。然而在其他地方，如果将get的结果强制类型转换为String类型，就会产生一个错误
        String file=(String)arrayList.get(2);
    }
}
```

使用泛型后

```
package com.naixue.vip.p6.evolution;

/**
 * @Description 模拟JDK5以前的没有泛型之前对象重用的问题
 * @Author 向寒 奈学教育
 * @Date 2020/7/6 3:04 PM
 **/
public class ArrayListNew<E> {

    private int size = 0;

    private Object[] elements = {};

    public Object get(int i) {
        if (size > i) {
            return elements[i];
        }
        throw new IndexOutOfBoundsException();
```

```
    }

    public void add(E o) {
        size++;
        Object[] array = new Object[size];
        for (int i = 0; i < elements.length; i++) {
            array[i]=elements[i];
        }
        array[size-1]=o;
        elements=array;
    }

    public static void main(String[] args) {
        ArrayListNew<String> arrayList = new ArrayListNew<String>();
        arrayList.add("a");
        String s=(String)arrayList.get(0);
        //演示时放开注释
//        arrayList.add(1);
//        // 编译不通过，不会导致运行后才发生错误
//        arrayList.add(new File("/"));
//        String file=(String)arrayList.get(2);
    }
}
```

泛型的可重用性

```
package com.naixue.vip.p6.feature;

import org.junit.Test;

import static junit.framework.TestCase.assertTrue;

/**
 * @Description 体现泛型的代码可重用性
 * @Author 向寒 奈学教育
 * @Date 2020/7/6 3:04 PM
 **/
public class Reuse<T extends Comparable>
{
    public Integer compareTo(T t1, T t2) {
        return t1.compareTo(t2);
    }

    @Test
    public void testByGenericT()
    {
        Integer n1=1;
        Integer n2=2;
        Reuse<Integer> integerReuse =new Reuse<Integer>();
        assertTrue(integerReuse.compareTo(n1,n2) == -1);

        String s1="100";
        String s2="201";
        Reuse<String> integerReuse2 =new Reuse<String>();
        assertTrue(integerReuse2.compareTo(s1,s2) == -1);
    }
}
```

泛型的类型

泛型类

泛型类（generic class）就是具有一个或多个类型变量的类

```
public class Pair<T> {
    private T first;
    private T second;
    public Pair() { first = null; second = null;  }
    public Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }
    public T getFirst() { return first; }
    public T getSecond() { return second; }
    public void setFirst(T newValue) { first = newValue; }
    public void setSecond(T newValue) { second = newValue; }
}
```

泛型方法

```
package com.naixue.vip.p6.funciton;

/**
 * @Description 泛型方法的使用
 * @Author 向寒 奈学教育
 * @Date 2020/7/6 17:58
 **/
public class ArrayAlg {

    public static <T> T getMiddle(T... a) {
        return a[a.length / 2];
    }

    public static void main(String[] args) {
        String middle =ArrayAlg.<String>getMiddle("a","b","c");
        System.out.println(middle);
        //因为有两种方式处理。所有的都自动装箱，则得到 两个 Integer 和一个Double
        // 之后寻找公共超类 Number 和公共接口 Comparable 于是编译器不知道该如何处理了。
//        Integer n =ArrayAlg.<Integer>getMiddle(1,2,3.2);
//        System.out.println(n);
    }
}
```

子类泛型

class<T extends 父类>

```
package com.naixue.vip.p6.wildcard;
```

```
import com.naixue.vip.p6.classes.Pair;
import com.naixue.vip.p6.vo.Man;
import com.naixue.vip.p6.vo.Parson;
import com.naixue.vip.p6.vo.Woman;

/**
 * @Description 泛型类型限定只能使用子类
 * @Author 向寒 奈学教育
 * @Date 2020/7/6 19:56
 **/
public class Programmer<T extends Parson> {

    public <T extends Parson> Pair<T> youngAndOld(T[] a) {
        if(a == null || a.length == 0) {
            return null;
        }
        T min = a[0];
        T max = a[1];
        for(int i = 0; i < a.length; i++) {
            if(min.getAge().compareTo(a[i].getAge()) > 0) {
                min = a[i];
            }
            if(max.getAge().compareTo(a[i].getAge()) < 0) {
                max = a[i];
            }
        }
        return new Pair<>(min, max);
    }

    public static void main(String[] args) {
        Programmer<Parson> programmer = new Programmer<Parson>();
        Man man=new Man("陈先生",48,"劳力士","幻影");
        Woman woman=new Woman("刘女士",27,"LV","迪奥");
        Man man1=new Man("张先生",32,"浪琴","奥迪");
        Woman woman1=new Woman("吴女士",18,"LV","圣罗兰");
        Parson[] parsons=new Parson[4];
        parsons[0]=man;
        parsons[1]=woman;
        parsons[2]=man1;
        parsons[3]=woman1;

        Pair<Parson> parsonPair = programmer.youngAndOld(parsons);

        //子类型规则，即任何参数化的类型是原生态类型的一个子类型，
        //Programmer<T>是Programmer<Parson>类型的一个子类型，而不是
Programmer<Object>的子类型。
//        Programmer<String> programmer1 = new Programmer<String>();
//        Programmer<Object> programmer1 = new Programmer<Object>();
    }

}
```

泛型的规则

类型擦除

类擦除案例

jdk1.5之前代码

```
public class Node{
   private Object obj;

   public Object get(){
       return obj;
   }

   public void set(Object obj){
       this.obj=obj;
   }

   public static void main(String[] argv){

    Student stu=new Student();
    Node  node=new Node();
    node.set(stu);
    Student stu2=(Student)node.get();
   }
}
```

使用泛型的代码

```
public class Node<T>{

    private T obj;

    public T get(){

        return obj;
    }

    public void set(T obj){
        this.obj=obj;
    }

    public static void main(String[] argv){

    Student stu=new Student();
    Node<Student>  node=new Node<>();
    node.set(stu);
    Student stu2=node.get();
  }
}
```

两个版本生成的.class文件

```
  public Node();
    Code:
       0: aload_0
```

```
       1: invokespecial #1                  // Method java/lang/Object."<init>":
()V
       4: return
  public java.lang.Object get();
    Code:
       0: aload_0
       1: getfield      #2                  // Field obj:Ljava/lang/Object;
       4: areturn
  public void set(java.lang.Object);
    Code:
       0: aload_0
       1: aload_1
       2: putfield      #2                  // Field obj:Ljava/lang/Object;
       5: return
}
```

```
public class Node<T> {
  public Node();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":
()V
       4: return
  public T get();
    Code:
       0: aload_0
       1: getfield      #2                  // Field obj:Ljava/lang/Object;
       4: areturn

  public void set(T);
    Code:
       0: aload_0
       1: aload_1
       2: putfield      #2                  // Field obj:Ljava/lang/Object;
       5: return
}
```

可以看到泛型就是在使用泛型代码的时候，将**类型信息**传递给具体的泛型代码。而经过编译后，生成的`.class`文件和原始的代码一模一样，就好像传递过来的**类型信息**又被擦除了一样。

方法擦除案例

```
public class ArrayAlg {

    public static <T> T getMiddle(T... a) {
        return a[a.length / 2];
    }

    //编译后的实际代码，演示方法擦除
    public static Object getMiddle(Object ... a) {
        return a[a.length / 2];
    }
}
```

编译器桥接

```
package com.naixue.vip.p6.bridging;

/**
 * @Description
 * @Author 向寒 奈学教育
 * @Date 2020/7/7 11:05
 **/
public class Node<T> {
    public T data;

    public void setData(T data) {
        this.data = data;
    }

    public Node(T data) {
        System.out.println("Node.setData");
        this.data = data;
    }
    public static class MyNode extends Node<Integer>{

        public MyNode(Integer data) {
            super(data);
        }

        @Override
        public void setData(Integer data) {
            System.out.println("MyNode.setData");
            super.setData(data);
        }

        //模拟编译器产生的桥接方法
//        public void setData(Object data){
//            setData((Integer)data);
//        }
    }
    public static void main(String[] args) {
        MyNode mn=new MyNode(5);
        Node n=mn;
        //java.lang.ClassCastException: java.lang.String cannot be cast to
java.lang.Integer
        n.setData("Hello");
        Integer x=mn.data;
        System.out.println(x);
    }
}
```

堆污染

Heap pollution(堆污染), 指的是当把一个不带泛型的对象赋值给一个带泛型的变量时, 就有可能发生堆污染.

```
package com.naixue.vip.p6.heappollution;

import org.junit.Test;

import java.util.ArrayList;
import java.util.List;
```

```
/**
 * @Description 堆污染案例
 * Heap pollution(堆污染), 指的是当把一个不带泛型的对象赋值给一个带泛型的变量时, 就有可能发生堆污染.
 * @Author 向寒 奈学教育
 * @Date 2020/7/7 11:24
 **/
public class Pollution {

    public void funa() {
        List intList = new ArrayList<Integer>();
        intList.add(1);
        //堆污染：类型转换异常
//        List<String> strList = intlist;
    }

    @Test
    public void funb() {
        List<Integer> intList = new ArrayList<>();
        intList.add(1);

        List<String> strList = new ArrayList<>();
        strList.add("a");

        //通过泛型做到了转换，编译通过
        List list=intList;
        List<String> lst=list;

        //运行时会发生转换异常
        //java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
        System.out.println(lst.get(0));

    }
}
```

翻译泛型表达式

```
Pair<Parson> pair=new Pair<Parson>();
pair.getFirst();
```

擦除getFirst的返回类型后将返回Object类型。编译器自动插入Employee的强制类型转换。

编译器把这个方法调用翻译为两条虚拟机指令：

- 对原始方法Pair.getFirst的调用。
- 将返回的Object类型强制转换为Parson类型。

子类型规则

类型边界

泛型T在最终会擦除为Object类型，只能使用Object的方法

通配符

假设有一种场景，你不知道这个类型是啥，它可以是Object，也可以是其他类那咋办？

这种场景就需要用到通配符

```
<T extends Parson>
<T super Parson>
<?>
<? extends Parson>
<? super Parson>
```

无限制通配符

使用原生态类型是很危险的，但是如果不确定或不关心实际的类型参数。那么在Java 1.5之后Java有一种安全的替换方法，称之为无限制的通配符类型（unbounded wildcard type），可以用一个“?”代替，比如Set<?>表示某个类型的集合，可以持有任何集合。

那么无限制通配类型与原生态类型有啥区别呢？原生态类型是可以插入任何类型的元素，但是无限制通配类型的话，不能添加任何元素（null除外）。

```
Set<?> set=new HashSet<>();
set.add("abc");//编译错误
```

它的出现归根结底是为了防止破坏集合类型约束条件，并且可以根据需要使用泛型方法或者有限制的通配符类型（bound wildcard type）接口某些限制，提高安全性。

上界通配符

可以扩展为父类，来调用其方法。必须是子类或者本身。

父类Parson

```
package com.naixue.vip.p6.vo;

/**
 * @Description 人的class
 * @Author 向寒 奈学教育
 * @Date 2020/7/6 17:03
 **/
public class Parson {

    private String name;

    private Integer age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }
```

```
    public void setAge(Integer age) {
        this.age = age;
    }

    public Parson() {
    }

    public Parson(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return name+","+age;
    }
}
```

子类Man

```
package com.naixue.vip.p6.vo;

/**
 * @Description 男人的class
 * @Author 向寒 奈学教育
 * @Date 2020/7/6 17:03
 **/
public class Man extends Parson{

    private String watch;

    private String car;

    public Man(String name, Integer age, String watch, String car) {
        super(name, age);
        this.watch = watch;
        this.car = car;
    }

    @Override
    public String toString() {
        return super.getName() + "," + super.getAge()+","+watch+","+car;
    }
}
```

子类Woman

```
package com.naixue.vip.p6.vo;

/**
 * @Description 女人的class
 * @Author 向寒 奈学教育
 * @Date 2020/7/6 17:03
 **/
public class Woman extends Parson {

    private String bag;
```

```
    private String lipstick;

    public Woman() {

    }

    public Woman(String name, Integer age, String bag, String lipstick) {
        super(name, age);
        this.bag = bag;
        this.lipstick = lipstick;
    }

    @Override
    public String toString() {
        return super.getName() + "," + super.getAge()
               + "," + bag + "," + lipstick;
    }
}
```

上界通配符和类型限定

```
package com.naixue.vip.p6.wildcard;

import com.naixue.vip.p6.classes.Pair;
import com.naixue.vip.p6.vo.Man;
import com.naixue.vip.p6.vo.Parson;
import com.naixue.vip.p6.vo.Woman;
import org.junit.Test;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

/**
 * @Description 泛型类型限定，只能使用子类
 * @Author 向寒 奈学教育
 * @Date 2020/7/6 19:56
 **/
public class Programmer<T extends Parson> {

    public <T extends Parson> Pair<T> youngAndOld(T[] a) {
        if(a == null || a.length == 0) {
            return null;
        }
        T min = a[0];
        T max = a[1];
        for(int i = 0; i < a.length; i++) {
            if(min.getAge().compareTo(a[i].getAge()) > 0) {
                min = a[i];
            }
            if(max.getAge().compareTo(a[i].getAge()) < 0) {
                max = a[i];
            }
        }
        return new Pair<>(min, max);
    }
```

```
    public void count(Collection<Parson> persons) {
        System.out.println(persons.size());
    }

    @Test
    public void countTest() {
        List<Man> manList=new ArrayList<>();
        //违反子类型化原则，编译报错
//          new Programmer().count(manlist);
    }

    public static void main(String[] args) {
        Programmer<Parson> programmer = new Programmer<Parson>();
        Man man=new Man("陈先生",48,"劳力士","幻影");
        Woman woman=new Woman("刘女士",27,"LV","迪奥");
        Man man1=new Man("张先生",32,"浪琴","奥迪");
        Woman woman1=new Woman("吴女士",18,"LV","圣罗兰");
        Parson[] parsons=new Parson[4];
        parsons[0]=man;
        parsons[1]=woman;
        parsons[2]=man1;
        parsons[3]=woman1;

        //符合上界通配符规则
        Pair<Parson> parsonPair = programmer.youngAndOld(parsons);

        //子类型规则，即任何参数化的类型是原生态类型的一个子类型，
        //Programmer<T>是Programmer<Parson>类型的一个子类型，而不是
Programmer<Object>的子类型。
//          Programmer<String> programmer1 = new Programmer<String>();
//          Programmer<Object> programmer1 = new Programmer<Object>();
    }
}
```

多限定

```
public class Programmer<T extends Parson & Serializable> {

}
```

下界通配符

```
    public <T> void func(List<? super Man> src) {

    }

    @Test
    public void testCopy() {
        Guide programmer = new Guide();
        programmer.func(new ArrayList<Man>());
        programmer.func(new ArrayList<Parson>());
        //违反下界通配符原则,编译不通过
//          programmer.func(new ArrayList<WoMan>());
    }
```

泛型的作用域

```
package com.naixue.vip.p6.wildcard;

import com.naixue.vip.p6.vo.Man;
import com.naixue.vip.p6.vo.Parson;
import com.naixue.vip.p6.vo.Woman;

import java.util.ArrayList;
import java.util.List;

/**
 * @Description 泛型的作用域
 * @Author 向寒 奈学教育
 * @Date 2020/7/6 20:54
 **/
public class Guide<T> {
    /**
     * Hardworkinger的T的作用域是整个class,func的T的作用域就是本方法
     * 当上述两个类型参数冲突时，在方法中，方法的T会覆盖类的T，即和普通变量的作用域一样，内部覆盖外部，外部的同名变量是不可见的
     * @param t
     * @param <T>
     */
    public <T> void func(T t) {

    }

    /**
     * 可以定义不同类型泛型来区分作用域
     * @param s
     * @param <S>
     */
    public <S> void fund(S s) {

    }
}
```

类型上限

```
package com.naixue.vip.p6.wildcard;

import com.naixue.vip.p6.vo.Man;
import com.naixue.vip.p6.vo.Parson;
import com.naixue.vip.p6.vo.Woman;

import java.util.ArrayList;
import java.util.List;

/**
 * @Description 泛型的类型上限
 * @Author 向寒 奈学教育
 * @Date 2020/7/6 20:54
 **/
```

```
public class Guide<T> {
    /**
     * <T extends Parson>指定泛型方法的类型参数的上限
     * @param src
     * @param <T>
     * @return
     */
    public <T extends Parson> T funa(List<T> src) {
        return null;
    }

    /**
     * 不能在方法参数中定义上限
     * @param src
     * @param <T>
     */
//    public <T> T funb(List<T extends Parson> src) {
//        return null;
//    }

    public <T> void copy(List<T> dest,List<? extends T> src) {
        for (T t : src) {
            dest.add(t);
        }
    }

    /**
     * 使用?还可以强制避免你对src做不必要的修改，增加的安全性
     * @param src
     * @param <Parson>
     */
    public <Parson> void updateError(List<? extends Parson> src) {
        for (Parson parson : src) {
            //No candidates found for method call t.setAge(1).
//            parson.setage(1);
        }
    }
}
```

正确使用?的类型

?表示不可修改的类型

```
package com.naixue.vip.p6.wildcard;

import com.naixue.vip.p6.vo.Man;
import com.naixue.vip.p6.vo.Parson;
import com.naixue.vip.p6.vo.Woman;

import java.util.ArrayList;
import java.util.List;

/**
 * @Description 泛型的作用域 and 类型上限 and 使用?的不可修改类型
 * @Author 向寒 奈学教育
 * @Date 2020/7/6 20:54
 **/
```

```
public class Guide<T> {

    public <T> void copy(List<T> dest,List<? extends T> src) {
        for (T t : src) {
            dest.add(t);
        }
    }

    /**
     * 使用?还可以强制避免你对src做不必要的修改，增加的安全性
     * @param src
     * @param <Parson>
     */
    public <Parson> void updateError(List<? extends Parson> src) {
        for (Parson parson : src) {
            //No candidates found for method call t.setAge(1).
//            parson.setage(1);
        }
    }

    public static void main(String[] args) {
        Guide<Parson> programmer = new Guide<Parson>();
        Man man=new Man("陈先生",48,"劳力士","幻影");
        Woman woman=new Woman("刘女士",27,"LV","迪奥");
        Man man1=new Man("张先生",32,"浪琴","奥迪");
        Woman woman1=new Woman("吴女士",18,"LV","圣罗兰");
        List<Parson> parsons=new ArrayList<Parson>();
        parsons.add(man);
        parsons.add(woman);
        parsons.add(man1);
        parsons.add(woman1);

        List<Parson> dest=new ArrayList<Parson>();


        programmer.copy(dest,parsons);
        for (Parson parson : dest) {
            System.out.println(parson);
        }
    }
}
```

构造函数使用泛型

```
package com.naixue.vip.p6.kv;

/**
 * @Description 构造函数的通用方法使用
 * @Author 向寒 奈学教育
 * @Date 2020/7/6 16:51
 **/
public class GenericKV<K,V> {
    private K key;
    private V value;

    /**
     * 构造函数中使用泛型
```

```
     * @param key
     * @param value
     */
    public GenericKV(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public void setKey(K key) {
        this.key = key;
    }

    public V getValue() {
        return value;
    }

    public void setValue(V value) {
        this.value = value;
    }

    /**
     * 静态方法中使用泛型
     * @param p1
     * @param p2
     * @param <K>
     * @param <V>
     * @return
     */
    public static <K,V> boolean compare(GenericKV<K,V> p1,GenericKV<K,V> p2){
        return p1.getKey().equals(p2.getKey()) &&
p1.getValue().equals(p2.getValue());
    }

    public static void main(String[] args) {
        GenericKV<Integer,String> a=new GenericKV<Integer,String>(1, "a");
        GenericKV<Integer,String> b=new GenericKV<Integer,String>(2, "b");
        System.out.println(compare(a, b));
    }
}
```

使用规范

add时只能向下转型；向上转型要强转；
具有上界的通配符泛型只能get，不能add除null外的对象；
具有下界的通配符泛型可以add，但get获取对象为object类型；

```
package com.nx.qiuping.vip.generic.wildcard;

import com.nx.qiuping.vip.generic.vo.Jason;
import com.nx.qiuping.vip.generic.vo.Man;
import com.nx.qiuping.vip.generic.vo.Person;

import java.util.ArrayList;
```

```java
import java.util.List;

/**
 * @Description 泛型上限定通配符和下限定通配符案例
 * @Author 向寒 奈学教育
 * @Date 2020/7/11 18:38
 **/
public class Limited {
    public static  <T> void funa(List<? extends Man> src) {

    }

    public static  <T> void funb(List<? super Man> src) {

    }

    public static  <T> Integer func(List<? super Man> src) {
        return src.size();
    }

    public void test() {
        Limited.funa(new ArrayList<Man>());
        //上界通配符
        //限定参数只能是Man的子类和本身
//        Limited.funa(new ArrayList<Person>());
//        Limited.funa(new ArrayList<Woman>());

        //下界通配符
        //限定参数只能是Man的父类和本身
        Limited.funb(new ArrayList<Man>());
        Limited.funb(new ArrayList<Person>());
//        Limited.funb(new ArrayList<Woman>());

        /**
         * 上界的list只能get，不能add（确切地说不能add出除null之外的对象，包括Object）。
         * 下界的list只能add，不能get。
         */
        List<? extends Person> flistTop = new ArrayList<Person>();
        flistTop.add(null);
        //上界add 对象会报错
        //add无法确定add是哪个子类，所以不允许add
//        flistTop.add(new Man());
//        flistTop.add(new Woman());
//        flistTop.add(new Person());

        //子类直接可以赋值给父类，所以可以get
        Person fruit2 =new Man();
        Person fruit1 = flistTop.get(0);

        //下界
        List<? super Man> flistBottem = new ArrayList<Man>();
        flistBottem.add(new Man());
        flistBottem.add(new Jason());
        //因为父类不能直接赋值给子类所以不能add
//        Man man=new Person();
//        flistBottem.add(new Woman());

        //get的对象是? super Man 类型，模糊类型的所以不能直接复制给Man
```

```
//        Man man3=flistBottem.get(0);
        //强转就可以了
        Man man4=(Man) flistBottem.get(0);

    }
}
```

Java泛型转换的事实

- 虚拟机中没有泛型，只有普通的类和方法。
- 所有的类型参数都用它们的限定类型替换。
- 桥方法被合成来保持多态。
- 为保持类型安全性，必要时插入强制类型转换。

jdk定义了7种泛型的使用限制

- 不能用简单类型来实例化泛型实例
- 不能直接创建类型参数实例
- 不能申明静态属性为泛型的类型参数
- 不能对参数化类型使用cast或instanceof
- 不能创建数组泛型
- 不能create、catch、throw参数化类型对象
- 重载的方法里不能有两个相同的原始类型的方法