**泛型的问题引出**

泛型从JDK.1.5之后加入到java语言中，其主要目的就是为了解决ClassCastException 的问题，在进行对象的向下转型永远都可能存在有安全隐患，而Java希望通过泛型可以慢慢的解决此类问题。

1. **泛型的问题引出**
   1. 现在假设定义一个描述x与y 坐标的处理类，并且在这个类中允许开发者保存有三类数据

·整型数据: x = 10 、y = 20 ;

·浮点型数据: x = 10.1 、y = 20. 1；

·字符串型数据: x = 东经120度 y = 北纬30度 ；

于是在设计的时候就需要考虑到具体的 x, y的属性类型 String 、 int 还是double ?

在传统的开发中我们使用Object ，因为Object 可以接收所有的类，他们存在如下关系；

· 整型数据：基本数据类型 -> 包装成Integer 类对象 -> 自动向上转型为Object

· 浮点型数据： 基本数据类型 -> 包装成Double类对象 -> 自动向上转型为Object

·字符串数据：基本数据类型 -> 包装成String类对象 -> 自动向上转型为Object

|  |
| --- |
| class Point {  private Object x ;  private Object y ;  public void setX (Object x ){  this.x = x;  }  public void setY (Object y ){  this.y = y;  }  public Object getX(){  return this.x;  }  public Object getY(){  return this.y;  }  }  public class JavaDemo{  public static void main(String[] args) {  Point point = new Point();  //第一步根据需求设计内容  point.setX(10);  point.setY(20);  //第二步获取数据  int x = (int)point.getX();  int y = (int)point.getY();  System.out.println("x的坐标为:"+ x + ",y的坐标:"+ y );  }  } |

本程序可以解决当前问题，主要在于Object 可以接受所有数据类型，但是也正因为如此，也会出现严重的错误。

**范例：观察错误**

|  |
| --- |
| public class JavaDemo{  public static void main(String[] args) {  Point point = new Point();  //第一步根据需求设计内容  point.setX(10); // 自动装箱  **point.setY("北纬20度"); // 自动装箱**  //第二步获取数据  int x = (int)point.getX();  int y = (int)point.getY();  System.out.println("x的坐标为:"+ x + ",y的坐标:"+ y );  }  } |

注意: 这里将setY中的参数改变，在编译时不会报错，但是在执行期会报” ClassCastException” 异常类型，所以本程序的设计时存在安全隐患的！而这个安全隐患的根源就在于使用了Object，因为Object 的蕴含范围太广，而如果这样的错误可以直接在编译的过程中出现，那么就可以避免运行时错误。

1. 泛型的定义

如果想要避免项目之中出现 “ClassCastException” 异常，最好的做法就是可以直接回避掉对象的强制转换。所以JDK1.5之后出现了泛型技术，而泛型的本质在于，**类中的属性或方法类型可以在对象实例化时动态的赋予。**

那么此时就需要在类定义的时候明确定义占位符(泛型标记)；

|  |
| --- |
| class Point <T> {  private <T> x ;  private <T> y ;  public void setX (T x ){  this.x = x;  }  public void setY (T y ){  this.y = y;  }  public T getX(){  return this.x;  }  public T getY(){  return this.y;  }  }  public class JavaDemo2{  public static void main(String[] args) {  **Point point = new Point();**  //第一步根据需求设计内容  point.setX(10); // 自动装箱  point.setY("北纬20度"); // 自动装箱  //第二步获取数据  int x = (int)point.getX();  int y = (int)point.getY();  System.out.println("x的坐标为:"+ x + ",y的坐标:"+ y );  }  } |

此时Point 类中的属性和方法的类型，是由外部决定的，看到我们标红的实例化，此时我们没有赋予类型，所以默认情况下是Object ；

·由于泛型是JDK1.5之后的产物，但是之前我们已经有不少的内置程序类或者是接口广泛应用在项目开发中，于是为了保证这些类的追加了泛型之后，原程序依然可以使用，所以如果不设置泛型时，自动将Object 作为类型，以保证程序的正常执行，但是在编译的时候会出现警告信息；

泛型定义之后可以在实例化对象实例的时候给定类型；

|  |
| --- |
| **Point<Integer> point = new Point<Interger>();** |

这样Point 中所有的泛型占位符都被 Integer 给代替了（只局限于在本类之中），这样处理就会发现当设置的内容有误时能够在编译器就发现；同时也避免了对象向下转型的处理（避免了安全隐患）

**泛型的使用注意：**

**·**泛型之中只允许设置应用类型，如果现在要操作基本类型必须使用包装类（意识时不能使用<int > 而必须使用< Integer>）

·从jdk1.7之后 后面的泛型可以省略:

Point<Integer> point = new Point<>();

使用泛型可以解决大部分的类对象的强制转换，这样的设计才是一个合理的设计。

1. 泛型的通配符
   1. 通配符的引出

|  |
| --- |
| class Message <T> {  private T content;  public void setContent(T content){  this.content = content ;  }  public T getContent(){  return this.content;  }  }  public class JavaDemo3{  public static void main(String[] args) {  Message <String> msg = new Message<> ();  msg.setContent("www.baidu.com");  fun(msg);  }  public static void **fun (Message <String> temp){**  System.out.println(temp.getContent());  }  } |

那么此时问题就来了，既然我们使用了泛型，那么fun 方法就不能只接收一个String 类型 ，而是应该能接收任何Message类型。

那么如果我们不设置类型是否能够满足要求呢？

|  |
| --- |
| class Message <T> {  private T content;  public void setContent(T content){  this.content = content ;  }  public T getContent(){  return this.content;  }  }  public class JavaDemo3{  public static void main(String[] args) {  Message <String> msg = new Message<> ();  msg.setContent("www.baidu.com");  fun(msg);  }  public static void fun (Message temp){  **temp.setContent(1.1);**  System.out.println(temp.getContent());  }  } |

那么此时我们可以发现，在fun方法里面我们可以修改属性，也可以接收属性，这是一种非常**危险**的设计，因此我们需要设计一种：**可以接收所有泛型类型，并且不能够改变里面的数据（但是允许访问）。**

此时我们就需要用到泛型通配符了。<?>

|  |
| --- |
| class Message <T> {  private T content;  public void setContent(T content){  this.content = content ;  }  public T getContent(){  return this.content;  }  }  public class JavaDemo3{  public static void main(String[] args) {  Message <String> msg = new Message<> ();  msg.setContent("www.baidu.com");  fun(msg);  }  public static void fun (Message **<?>** temp){  temp.setContent(“new Content”);  System.out.println(temp.getContent());  }  } |

此时我们可以知道当我们在fun中增加时temp.setContent(“new Content”);会在编译期报”不兼容的类型: String无法转换为CAP#1”的异常，因为在fun 中我们使用的是 <?> 在使用temp.setContent()时我们无法知道通配符中的占位符是什么类型，因此会报错。

由此可知使用通配符可以使fun 方法更加安全。并达到了我们的需求；

在”？“ 这个通配符基础上实际上还提供了两类小的通配符：

·？extends 类:设置泛型上线

例如:定义:”?extends Number ”:表示该泛型类型只允许设置Number 或Number 的子类

·？Super 类：设置泛型下限

例如:定义:”?extends Number ”:表示该泛型类型只允许设置Number 或Number 的父类

1. 泛型接口

泛型除了可以在类上使用也可以在接口上使用；

|  |
| --- |
| interface IMessage<T> {  public String echo(T t) ;  } |

对于泛型接口的实现有两种方式；

实现方式一：在子类中继续设置泛型

|  |
| --- |
| interface **IMessage<T>** {  public String echo(T t) ;  }  class **MessageImpl<T> implements IMessage<T>**{  public String echo(T t){  return "echo:" + t ;  }  }  public class JavaDemo5 {  public static void main(String[] args) {  IMessage<String> mes = new MessageImpl<String>();  System.out.println(mes.echo("www.gsdx.com"));  }  } |

泛型接口实现的第二种方法：

在子类实现父类接口的时候直接给其定义泛型类型：

|  |
| --- |
| interface IMessage<T> {  public String echo(T t) ;  }  class MessageImpl implements IMessage<String>{  public String echo(String t){  return "echo:" + t ;  }  }  public class JavaDemo5 {  public static void main(String[] args) {  IMessage<String> mes = new MessageImpl();  System.out.println(mes.echo("www.gsdx.com"));  }  } |