2、具体内容

泛型从JDK1.5之后追加到Java语言里面的，其主要目的是为了解决ClassCastException的问题，在进行对象的向下转型永远都可能存在安全隐患，而Java希望通过泛型可以慢慢解决掉此类问题。

■泛型的问题引出

现在假设说定义一个描述x与y坐标的处理类，而且在这个类之中允许开发者保存有三类数数据。

- 整型数据：x=10、y=20；
- 浮点型数据：x=10.1、y=20.9；
- 字符串型数据：x=东经120度、北纬30度

于是在设计Point类的时候就需要去考虑具体的x和y属性的类型，这个类型要求可以保存以上三种数据，很明显，最为原始的做法就是利用Object类来进行定义，因为存在有如下的转型关系：

- 整型数据：基本数据类型 -> 包装为Integer类对象 ->自动向上转型为Object；
- 浮点型数据：基本数据类型 -> 包装为Double类对象 ->自动向上转型为Object；
- 字符串型数据：String类对象->自动向上转型为Object；

范例：定义Point类如下

```
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 ;
    }
}
```

而后下面进行内容的设置。

范例：进行正确的内容操作

```
public class JavaDemo {
    public static void main(String args[]) {
        Point point = new Point() ;
        // 第一步：根据需求进行内容的设置
        point.setX(10) ;       // 自动装箱
        point.setY(20) ;       // 自动装箱
        // 第二步：从里面获取数据
        int x = (Integer) point.getX() ;
        int y = (Integer) point.getY() ;
        System.out.println("x坐标：" + x + "、y坐标：" + y) ;
    }
}
```

本程序之所以可以解决当前的设计问题，主要的原因在于，Object可以接收所有的数据类型，但是正因为如此，所以本代码也会出现有严重的错误。

范例：观察错误

```
public class JavaDemo {
    public static void main(String args[]) {
        Point point = new Point() ;
        // 第一步：根据需求进行内容的设置
        point.setX(10) ;       // 自动装箱
        point.setY("北纬20度") ;// 自动装箱
        // 第二步：从里面获取数据
        int x = (Integer) point.getX() ;
        int y = (Integer) point.getY() ;
        System.out.println("x坐标：" + x + "、y坐标：" + y) ;
    }
}
```

此时的程序明显出现了问题，如果在程序编译的时候实际上是不会有任何的错误产生的，而程序的执行的时候就会出现“ClassCastException”异常类型，所以本程序的设计是存在有安全隐患的。而这个安全隐患存在的依据在于使用了Object类型，因为Object可以涵盖的范围太广了，而对于这样的错误如果可以直接出现在编译的过程之中，那么就可以避免运行时的错误。

■泛型定义

如果想要避免项目 出现“ClassCastException”最好的做法是可以直接回避掉对象的强制转换，所以在JDK1.5以后提供有泛型技术，而泛型的本质在于，**类中的属性或方法的参数与返回值的类型可以由对象实例化的时候动态决定。**

那么此时就需要在类定义的时候明确的定义占位符（泛型标记）。

```
class Point <T> {    // T是Type的简写，可以定义多个泛型
    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 ;
    }
}
```

此时Point类中的x与y属性的数据类型并不确定，而是由外部来决定。

提示：关于默认的泛型类型

• 由于泛型是属于JDK1.5之后的产物，但是在这之前已经有不少内置的程序类或者是接口广泛的应用项目开发之中，于是为了保证这些类或接口追加了泛型之后，原始的程序类依然可以使用，所以如果不设置泛型类型时，自动将使用Object作为类型以保证程序的正常执行，但是在编译的过程之中会出现警告信息。

泛型定义完成后可以在实例化对象的时候进行泛型类型的设置，一旦设置之后，里面的x与y的属性类型就与当前对象直接绑定了。

```
public class JavaDemo {
    public static void main(String args[]) {
        Point<Integer> point = new Point<Integer>() ;
        // 第一步：根据需求进行内容的设置
        point.setX(10) ;       // 自动装箱
        point.setY(20) ;       // 自动装箱
        // 第二步：从里面获取数据
        int x = point.getX() ;
        int y = point.getY() ;
        System.out.println("x坐标：" + x + "、y坐标：" + y) ;
    }
}
```

现在的程序代码之中，由于Point类里面设置的泛型类型为Integer，这样所有的对应此番性的属性、变量、方法返回值就将全部替换为Integer（只局限于此对象之中），这样在进行处理的时候如果发现设置有内容错误，则会在程序编译的时候自动进行错误的提示，同时也避免了对象的向下转型处理（可以避免安全隐患）。

泛型的使用注意点：
·泛型之中只允许设置引用类型，如果现在要操作基本数据类型必须使用包装类；
·从JDK.17开始，泛型对象实例化可以简化为“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处理，这样的程序才是一个合理的设计。

■泛型通配符

虽然泛型帮助开发者解决了一系列的对象的强制转换所带来的安全隐患，但是从另一个角度来讲，泛型也带来了一些新的问题：引用传递处理

范例：观察问题的产生

```
class Message <T> {
	private T content ;
	public void setContent(T content) {
		this.content = content ;
	}
	public T getContent() {
		return this.content ;
	}
}
public class JavaDemo {
	public static void main(String args[]) {
		Message<String> msg = new Message<String>() ;
		msg.setContent("www.mldn.cn") ;
		fun(msg) ;	// 引用传递
	}
	public static void fun(Message<String> temp){
		System.out.println(temp.getContent()) ;
	}
}
```

但是这个时候问题就出现了，而问题的关键在于fun()方法上，如果真的去使用泛型不可能只是一种类型，也就是说fun()方法应该可以接收任意种泛型类型的Message对象。但是这个时候它只能够接收“Message<>”类型，这种情况下有就有同学想出了，老师不设置泛型了。

范例：不设置泛型

```
public class JavaDemo {
	public static void main(String args[]) {
		Message<Integer> msgA = new Message<Integer>() ;
		Message<String> msgB = new Message<String>() ;
		msgA.setContent(110) ;
		fun(msgA) ;	// 引用传递
		msgB.setContent("www.mldn.cn") ;
		fun(msgB) ;
	}
	public static void fun(Message temp){
		System.out.println(temp.getContent()) ;
	}
```

```
}
    public static void fun(Message temp){
        temp.setContent(1.1) ;
        System.out.println(temp.getContent()) ;
    }
```

这个时候发现如果不设置泛型，那么在方法之中就有可能对你的数据进行修改，所以此时需要找一种方案：可以接收所有的泛型类型，并且不能够修改里面的数据（允许获取），那么就需要通过通配符”?”来解决。

范例：使用通配符

```
class Message <T> {
    private T content ;
    public void setContent(T content) {
        this.content = content ;
    }
    public T getContent() {
        return this.content ;
    }
}
public class JavaDemo {
    public static void main(String args[]) {
        Message<Integer> msgA = new Message<Integer>() ;
        Message<String> msgB = new Message<String>() ;
        msgA.setContent(110) ;
        fun(msgA) ;    // 引用传递
        msgB.setContent("www.mldn.cn") ;
        fun(msgB) ;
    }
    public static void fun(Message<?> temp){
        System.out.println(temp.getContent()) ;
    }
}
```

此时在fun()方法里面由于采用了Message结合通配符的处理所以可以接收所有的类型，并且不允许修改只允许获取数据。

在“?”这个通配符的基础之上实际上还提供两类小的通配符：

- ? textends 类：设置泛型的上限：
  - 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：设置泛型的下限：
  - 例如：定义“? super String”：只能够使用String或其父类；

范例：观察泛型的上限配置

```
class Message <T extends Number> {
    private T content ;
    public void setContent(T content) {
        this.content = content ;
    }
```

```
	public T getContent() {
		return this.content ;
	}
}
public class JavaDemo {
	public static void main(String args[]) {
		Message<Integer> msgA = new Message<Integer>() ;
		msgA.setContent(110) ;
		fun(msgA) ;	// 引用传递
	}
	public static void fun(Message<? extends Number> temp){
		System.out.println(temp.getContent()) ;
	}
}
```

范例：设置泛型下限

```
class Message <T> {
	private T content ;
	public void setContent(T content) {
		this.content = content ;
	}
	public T getContent() {
		return this.content ;
	}
}
public class JavaDemo {
	public static void main(String args[]) {
		Message<String> msgA = new Message<String>() ;
		msgA.setContent("www.mldn.cn") ;
		fun(msgA) ;	// 引用传递
	}
	public static void fun(Message<? super String> temp){
		System.out.println(temp.getContent()) ;
	}
}
```

对于通配符而言是一个重要的概念，并且要求一定要理解此概念的定义，在日后学习Java一些系统类库的时候会见到大量的通配符使用。

■泛型接口

泛型除了可以在类上定义之外也可以直接在接口之中进行使用，例如：下面定义一个泛型接口；

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

对于泛型接口的子类而言现在就有两种实现方式。

实现方式一：在子类之中继续设置泛型定义

```
```

```
interface IMessage<T> {
	public String echo(T t) ;
}
class MessageImpl<S> implements IMessage<S> {
	public String echo(S t) {
		return "【ECHO】" + t ;
	}
}
public class JavaDemo {
	public static void main(String args[]) {
		IMessage<String> msg = new MessageImpl<String>() ;
		System.out.println(msg.echo("www.mldn.cn")) ;
	}
}
```

实现方式二：在子类实现父接口的时候直接定义出具体泛型类型

```
interface IMessage<T> {
	public String echo(T t) ;
}
class MessageImpl implements IMessage<String> {
	public String echo(String t) {
		return "【ECHO】" + t ;
	}
}
public class JavaDemo {
	public static void main(String args[]) {
		IMessage<String> msg = new MessageImpl() ;
		System.out.println(msg.echo("www.mldn.cn")) ;
	}
}
```

如果从概念和是线上来讲并不复杂，但是在日后会遇见大量出现有泛型的接口，这个时候一定要清楚两种实现原则；

■泛型方法

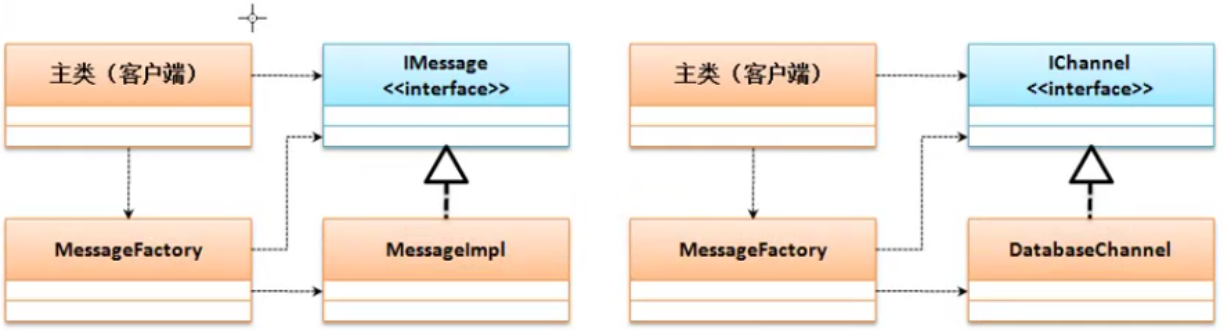
在之前的程序类里面实际上已经可以发现泛型类之中如果将泛型标记写在了方法上，那么这样的方法就被称为泛型方法，但是需要注意的是，泛型方法不一定非要出现在泛型类之中，即：如果一个类上没有定义泛型，那么也可以使用泛型方法。

```
public class JavaDemo {
	public static void main(String args[]) {
		Integer num [] = fun(1,2,3) ;  // 传入了整数，泛型类型就是Integer
		for (int temp : num) {
			System.out.print(temp + "、") ;
		}
	}
	public static <T> T[] fun(T ... args) {
		return args ;
	}
```

```
}
```

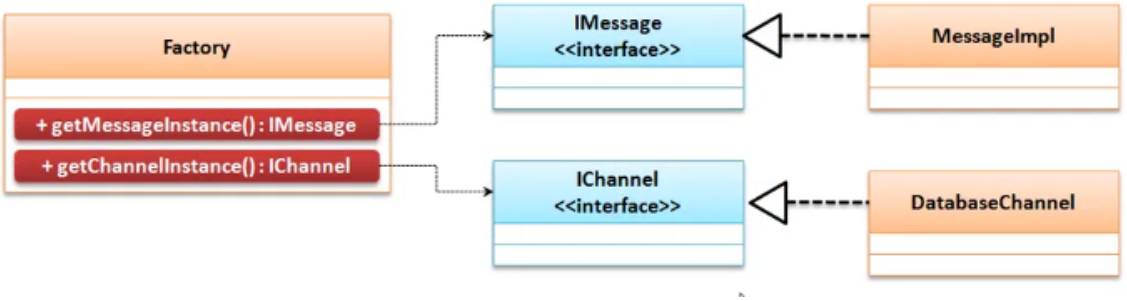
在后期进行项目开发的时候，这种泛型方法很常见，以之前的工厂设计为例。

传统工厂设计



如果此时一个项目有上千个接口，到时候都是绝望的身影。

传统工厂



范例：利用泛型改进工厂