第3章 分布式对象系统设计与IDL定义

同一般的面向对象系统的开发一样，分布式对象系统的开发也需要首先从系统的设计开始。系统设计的结果将为系统的实现提供依据。同一般面向对象系统不同的是，分布式对象系统的设计还包括对IDL接口的设计。IDL接口的设计为分别独立地实现客户端程序和服务器程序打下基础。本章主要介绍分布式对象系统的设计过程以及IDL语言的基本语法构成和定义方法。

3.1 分布式对象系统的开发流程

利用CORBA规范来开发分布式对象系统的基本流程如图3.1所示。

由图3.1可知，分布式对象系统的开发流程是由系统设计、IDL接口定义和语言映射、客户端开发、服务器开发以及总体测试与运行等几部分组成的。

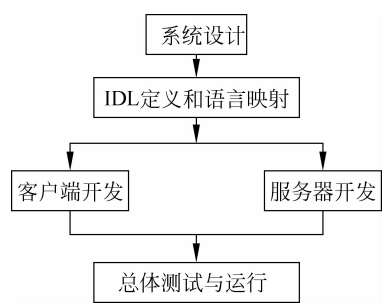

图3.1 分布式对象系统的开发流程

1. 系统设计

基于CORBA系统开发的第一步是系统设计。一般来讲，分布式对象系统的设计同本地对象系统的设计是一样的。

面向对象程序设计的结果(对象的表述及对象的联系等)，可以利用统一建模语言(Unified Modeling Language，UML)来描述。

2. IDL定义和语言映射

从这里开始是CORBA特有的部分。在上述系统设计的结果当中，作为分布式对象实现的对象接口，需要利用IDL来进行描述。这通常是利用手工操作来完成的，但是，如果系统设计的结果是利用UML来表述的话，也可以自动生成(即自动将UML表述转换成相应的IDL定义)。

不管怎样，当IDL接口描述结束之后，就需要利用ORB产品提供的IDL编译器，将IDL定义映射为相应的语言，也就是将IDL定义转换为Stub和Skeleton等对于系统开发所必需的Java语言等代码，这一生成过程被称为映射。在CORBA规范中，对每种程序设计语言都有相应的映射规定。这里需要注意的是，不是所有的程序设计语言都可以用来开

发基于 CORBA 的分布式对象系统的，换句话说，如果能够利用某种语言来开发 CORBA 应用系统的话，那一定是存在将 IDL 接口定义映射为那种语言的编译程序。

3. 客户端开发

从这一步开始就进入了实际的程序设计阶段了。一般来讲，客户端开发是在 IDL 接口映射之后进行的，这是由于在客户端开发过程中需要利用 IDL 映射后所生成的类，比如 Stub 类等。这里需要注意的是，客户端与服务器之间的区别只是相对的。一个程序对于某个分布式对象来讲可以是客户端，而对别的分布式对象来讲又可以是服务器。

客户端开发就是通过利用 IDL 接口中定义的方法，来实现相应的程序功能。

4. 服务器开发

服务器开发主要包括分布式对象实现的定义以及实现 CORBA 服务器进程的程序开发。同客户端开发相似，服务器开发一般也是在 IDL 接口映射之后进行的，这是由于在服务器开发过程中需要利用 IDL 映射后所生成的类，比如 Skeleton 类等。

在 IDL 接口定义的支持下，客户端开发和服务器开发可以同时进行，这是由于客户端所要利用的分布式方法以及服务器需要定义的分布式方法的原型定义在 IDL 接口定义中都确定下来了，这样，只要按照所给出的原型进行调用和定义即可。

5. 综合测试与运行

在此阶段，将分别开发的客户端程序和服务器程序进行综合测试，在此基础上进行实际运行。为了运行分布式对象系统，需要做许多系统设置的准备工作，而不是单纯的设计程序与运行。在实际运行时，还需要考虑系统维护等工作。

3.2 基于 CORBA 分布式对象系统设计

对分布式对象系统来讲，系统设计是系统开发过程中最重要的阶段。在进行系统设计时，除了需要给出具体的功能等设计之外，还需要考虑如何来表示设计结果的问题。

1. 系统设计的基本步骤

一般来讲，基于 CORBA 的分布式对象系统设计可以由如下几部分组成。

1）归纳系统的要求，即功能设计

功能设计，即归纳出系统的需求关系。需要根据系统的具体功能要求来设计。有时，在进行功能设计时，还需要给出具体的 QoS 要求。

2）对象的确定

为了满足上述功能设计的要求，需要确定提供相应服务的对象，这部分的重点并不在于对象的内部构造定义，而在于将哪些对象组合起来，以提供相应的服务。

3）对象的接口设计

所需要的对象确定了以后，就要设计相应对象的接口。在这一阶段，重要的是要明确需要什么样的接口，而不需要考虑接口是怎样实现的。

4）接口的构造设计

每个分布式对象的接口确定了以后，就需要对实现这一接口的构造进行设计。这一步工作的目的就是有利于用 IDL 进行接口定义。如果接口的构造设计结束了，则描述 IDL 定义的信息也就齐全了。

5）对象实现的设计

接口的构造设计结束以后，就需要对实现这一接口的对象实现(object implementation)进行设计，也就是对实现 IDL 接口定义功能的类进行设计。基于这一阶段的设计，CORBA 客户端和服务器的开发基本上就可以进行了。

6）运行环境设计

运行环境设计就是要进行服务器进程的配置等物理方面的设计。由于物理设计直接影响到系统的运行性能，因此，即使系统开发结束以后，也有可能进行改变。

2. 设计结果的表示

软件开发大体上是由三个阶段组成的：需求分析、设计和编码。这三个阶段在面向对象程序设计中分别被称为 OOA(Object Oriented Analysis)、OOD(Object Oriented Design)和 OOP(Object Oriented Programming)。

在上述面向对象程序设计的过程中，特别是在 OOA 和 OOD 的阶段已有很多方法可以利用，这些方法尽管各有差异，但都是在找出对象的静态构造、功能(作用)和动态关系之后，利用相应的描述方法进行描述。目前最为流行的方法是 UML。对于分布式对象的设计结果来讲，也可以利用 UML 来表示。在利用 UML 来形式化表示设计结果的情况下，就有可能自动地将设计结果转化为 IDL 接口定义。

下面以类的描述为例，来说明 UML 的基本用法。

1）类的 UML 表示

类的 UML 表示形式如图 3.2 所示，这里“名字区域”用于表示类的名字，“属性区域”用于表示类的数据成员，“操作区域”用于表示类的成员方法。图 3.3 是类的 UML 表示的具体例子。

图 3.2　类的 UML 表示

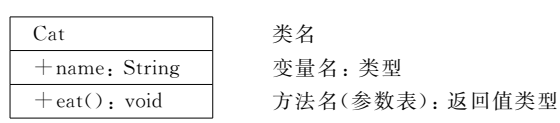

图 3.3　类的 UML 表示的例子

在图 3.3 中，Cat 是类名，name 是数据成员名，冒号后面的部分为该数据成员的类型名。eat 是成员方法名，冒号后面的部分为该函数的返回值类型。在数据成员和成员方法名前面的＋号，表示访问权限。可以设置的访问权限符号及其意义如下：

- ＋：public 权限；
- －：private 权限；
- ＃：protected 权限。

类中的抽象方法可用斜体来表示，而抽象类是在类名的下面用{ }括起来的类的属性来表示的，如图 3.4 所示。

2）类的实例的 UML 表示

类的实例(即对象)的 UML 表示如图 3.5 所示。

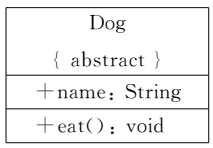

图 3.4 抽象类的表示

C: Cat	对象名：类名
name: "Nothing"	对象中的变量名：值

图 3.5 类的实例的 UML 表示例子

在图 3.5 中，C 是 Cat 类的对象，该对象中的 name 数据成员的值为字符串 Nothing。

3）继承的 UML 表示

继承是面向对象程序设计中的重要概念，在 UML 中也提供了有关继承的表示方法。图 3.6 是 UML 表示继承的一个例子。

图 3.6 的 UML 图说明了 Mammal 是基类，Dog 和 Cat 都是通过继承 Mammal 类而定义的派生类。

UML 的基本内容很多，感兴趣的读者可以查阅相关书籍。总之，分布式对象的设计结果是可以利用 UML 来进行描述的。

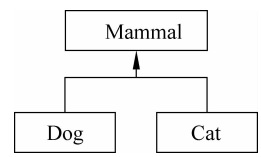

图 3.6 继承的 UML 表示

3.3 IDL 接口定义语言

IDL(Interface Definition Language，接口定义语言)是用于描述分布式对象接口的定义语言，利用 IDL 进行接口定义之后，就确定了客户端与服务器之间的接口，这样即使客户端和服务器独立进行开发，也能够正确地定义和调用所需要的分布式方法。

3.3.1 IDL 的作用

IDL 是 CORBA 的重要的组成部分，这是由于 CORBA 分布式对象能够提供怎样的接口，完全是由 IDL 来规定的。换句话说，CORBA 能够实现的分布式对象接口，是在 IDL 所能够定义的范围之内的。

IDL 定义与语言映射是基于 CORBA 分布式对象系统开发的第二步，这里，从系统设计结果之一的接口定义来生成 IDL 定义，同时，为了在程序设计中能够利用 IDL 定义，需要进行语言映射。在具体介绍 IDL 定义与语言映射之前，先看一看 IDL 的作用。

IDL 的作用就是使分布式对象的接口能够独立于程序设计语言来进行描述。在能够处理各种程序设计语言的系统环境当中，不依赖于程序设计语言而能够进行接口描述是重要的功能。

为了使 IDL 定义的接口能够被实际的程序设计语言所接受，就必须将其转换为相应的程序设计语言的代码，这一过程被称为语言映射，实现这一功能的工具被称为 IDL 编译器。

同一个 IDL 接口，经过不同的语言编译器编译之后，就能够被映射为不同语言所需要的类，这样，就可以采用不同的语言来分别实现服务器和客户端，也就是说客户端和服务器不一定需要采用同一种语言来实现。

我们知道，IDL 是用于定义 CORBA 分布式对象的接口的，IDL 接口的定义是在客户端

程序设计和服务器程序设计之前完成的。目前 IDL 定义还是通过手工编码来实现的，也许在不远的将来，能够从 UML 的设计数据自动生成 IDL 接口定义。但不管怎样，掌握 IDL 的语法构成都是必须的。

为了定义 CORBA 对象接口，IDL 提供了 7 种形式的类型定义，同时还提供了多种数据类型。由 IDL 提供的 7 种定义是：类型定义、常量定义、异常定义、属性定义、操作定义、接口定义和模块定义。

下面具体介绍 IDL 语法及其功能。

3.3.2　数据类型

CORBA 版本不同能够使用的 IDL 语言中的数据类型也不相同。在 CORBA 2.0 及其以后的版本中，能够使用的 IDL 中的数据类型包括对象引用类型、基本类型以及复合类型等。如表 3.1～表 3.3 所示。

1. 对象引用类型

IDL 对象引用类型如表 3.1 所示，其类型名为 Object。

表 3.1　对象引用类型

类型名	说　明
Object	指向 CORBA 对象的指针

2. 基本数据类型

IDL 提供了大量的基本数据类型，如表 3.2 所示。

表 3.2　基本数据类型

基本数据类型	说　明
short	16 位有符号整数
long	32 位有符号整数
long long	64 位有符号整数
unsigned short	16 位无符号整数
unsigned long	32 位无符号整数
unsigned long long	64 位无符号整数
float	32 位单精度浮点数
double	64 位双精度浮点数
long double	128 位双精度浮点数
fixed	32 位定点(小)数
char	1 字节字符
wchar	支持多字节代码集的字符数据类型
string	字符串
wstring	多字节字符串
boolean	布尔值：TRUE 或 FALSE
octet	在传递中不发生变换的 8 位数据(如传递二进制数)
any	可以存放任意类型的数据

3. 复合数据类型

复合数据类型如表 3.3 所示。

表 3.3 复合数据类型

类型名	说　明	类型名	说　明
enum	枚举	union	联合(共用体)
sequence	任意类型的一维数组	数组[ ]	任意类型的任意维数组
struct	结构(结构体)		

在 IDL 定义中,对变量名以及操作名(方法名)等标识符的命名规则如下：

(1) 标识符是由英文字母(a～z,A～Z)、数字(0～9)以及下划线(_)组成的,且标识符的第一个字符必须是英文字母。

由上述命名规则可知,下面的标识符名是合法的：

```
isEmpty,intfol,MAX_SIZE
```

而下面的标识符名是不合法的：

```
10Base,_Data,$STR
```

(2) IDL 定义中的大写字母和小写字母是有区别的,即作为不同的字母来看待。如标识符 find 和 Find 将分别表示两个不同的标识符。

(3) 关键字不能作为一般用户定义的标识符。

下面将具体介绍在 IDL 中可以利用的 7 种类型的定义格式及其作用。

3.3.3 类型定义

类型定义主要包括 typedef、struct、union、enum 和 sequence。

1. typedef 定义

typedef 的定义格式如下：

```
typedef 已有类型名  别名;
```

例如：

```
typedef  string  Name;              //可以利用 Name 来定义 string 类型的变量
typedef  long  Matrix[10][10];      //可以利用 Matrix 来定义 10×10 二维 long 类型数组
```

在程序中使用 typedef 的主要目的是为了提高程序的可读性。

2. struct 定义

struct 是用于定义结构类型的关键字,其基本用法如下：

```
struct 结构名
{
      类型名  变量名;
      类型名  变量名;
       ⋮
};
```

例如：

```
struct  sPoint
{
    long  x;
    long  y;
};
```

其中，sPoint 为结构类型名，该结构包含两个 long 类型的数据成员。

3. union 定义

union 是用于定义联合类型的关键字，其基本用法如下：

```
union  联合名  switch(区分符的类型名)
{
      case 区分符的值 1:类型名  变量名;
      case 区分符的值 2:类型名  变量名;
      ⋮
      default:类型名  变量名;
};
```

例如：

```
union uItem switch(short)
{
    case 1:long lval;
    case 2:double dval;
    default:string sval;
};
```

其中，uItem 是联合类型名，switch 后面括号中的 short 表示该联合类型是由短整型值区分的，如果该短整型的值为 1，则利用 lval 成员来存储 long 类型数据，如果该短整型的值为 2，则利用 dval 成员来存储 double 类型数据，如果该短整型的值既不是 1 也不是 2，则利用 sval 成员来存储 string 类型数据。

需要注意的是，switch 后面括号中的“区分符的类型名”只能是 integer、char、boolean 和 enum 等类型，同时，case 后面的值的类型必须与“区分符的类型名”相匹配。例如：

```
enum EnumType{a,b,c,d};
union UnionType switch(EnumType)
{
    case a:long field_a;
    case b:string field_b;
    default:boolean field_z;
};
```

4. enum 定义

enum 是用来定义枚举类型的关键字，其基本用法如下：

```
enum  枚举类型名 {name1,name2,…,nameN};
```

例如：

```
enum CalcMode{HEX,OCT,BIN,DEC};
```

在此定义的基础上，如果定义了 CalcMode 类型变量，则该变量可以取 HEX、OCT、BIN 和 DEC 中的任一个值。例如：

```
enum ABCDType{A,B,C,D};
union UType switch(ABCDType)
{
    case A:long AType;
    case B:string BType;
    case C:double CType
    default:boolean DType;
};
```

5. sequence 定义

sequence 用于定义任意类型的一维数组。其基本用法如下：

```
sequence<类型名>变量名;
```

例如：

```
sequence<short>sv;
```

它表示 sv 是一个无界的短整型一维数组(或称一维序列)。如果希望限定一维序列的上限，则可以采用 sequence<类型名,上限>的方式来定义。例如：

```
sequence<short,100>sx;
```

它表示 sx 是一个上限不超过 100 的短整型一维序列。

需要注意的是，类型定义中的很多定义与 C 语言中的相应定义是不同的，有的理解起来还有一定的难度，但这里给出的只是形式上的定义而已，这里的形式定义只能在 IDL 接口定义中直接使用，而真正在程序中需要使用的则是其映射后的结果。有关类型定义的映射请参考本章的后续内容。

3.3.4 常量定义

IDL 常量定义的格式如下：

```
const 类型名 常量名 = 常量表达式;
```

例如：

```
const long MAXSIZE = 100;
```

利用这种方式定义的常量，不但在接口定义中可以引用，而且在实现接口的程序中也可以引用。能够定义常量的数据类型有整型、浮点型、定点小数类型、字符类型、字符串类型和布尔类型等。

常量表达式不仅仅可以是一个常数，而且可以是包括运算符的常量表达式。在常量表

达式中可以包含如下一些运算符：

- 位运算符：|（按位或）、&（按位与）、^（按位异或）、～（按位取反）；
- 移位运算符：>>（右移）、<<（左移）；
- 加减乘除运算符：＋（加法）、－（减法）、*（乘法）、/（除法）、%（求余）；
- 单项运算符：＋（正）、－（负）。

3.3.5　异常定义

IDL 异常定义的格式如下：

```
exception  异常名
{
    类型名 1  变量名 1;
    类型名 2  变量名 2;
      ⋮
};
```

在实际应用过程中，一般在异常类中不定义变量或只定义一个变量。例如：

```
exception ServerException { };
exception NotFoundException { string module_name;};
```

在 ORB 中，当因某种原因调用函数（或称操作）失败时，为了识别发生了何种异常，需要将必要的信息存入特定的数据结构（异常中定义的数据结构）中，并将此数据结构传递给调用此函数的程序中。换句话说，在调用分布式方法失败时，需要将异常中定义的数据结构返回给客户端程序。

需要注意的是，在 CORBA 中不支持异常的继承操作。

CORBA 中规定的异常有两种：系统异常和用户异常。系统异常的入口由 ORB 全部规定好，当发生系统异常时，被传递的数据结构也都确定好了。用户异常是异常名和数据结构都需要由用户来定义的异常，其定义的形式如前所述。

如上面的例子那样，异常定义中可以为空。

在调用哪个操作时发生用户定义的异常，将在后面叙述。

3.3.6　属性定义

服务器向客户端公开的属性是利用属性定义来描述的。

属性定义的格式如下：

```
[readonly] attribute  类型名  属性名;
```

例如：

```
attribute long count;
```

如果我们把 IDL 定义中的操作看作为 Java 类中的公有方法的话，那么属性就可以被看作是公有的变量。

作为属性定义的数据可以被客户端所访问（包括读和写），但是，如果在定义属性的前面

加有 readonly 关键字，则只能对该属性进行读操作。属性定义只能在后面将要叙述的 interface 定义中使用。

在对属性进行访问时，也可能发生标准异常。

属性的基本概念可通过对下面两个等价的模块定义的比较来理解。

下面的模块定义是比较原始的，仅仅是为了说明问题而已，一般在实际运用中并不这样定义。

```
module common
{
    interface commonItem
    {
        string get_name();
        void set_name(in string name);
        long get_amount();
        void set_amount(in long amount);
        double get_price();
        void set_price(in double price);
    };
    ⋮
};
```

从上面的接口定义可知，非常烦琐，接口中的 6 个操作定义仅仅能够读/写对象的三个状态变量，而且 6 个定义可以分为 3 组，每组之间都非常相似。显然，这种定义方式非常烦琐。

考虑到读取、修改对象状态变量是一个相当普遍的操作，在 IDL 中引入了属性的概念，用 attribute 定义。利用 attribute 重新定义上述接口如下：

```
module common
{
    interface commonItem
    {
        attribute string name;
        attribute long amount;
        attribute double price;
    };
    ⋮
};
```

显然，后一种定义方式比前一种定义方式更简单明了。

3.3.7 操作定义

操作相当于 Java 类中的公有方法，其定义格式如下：

```
[oneway] 返回值类型  操作名(参数 1,参数 2,…)
                [ raises(异常名 1,异常名 2,…)]
                [context("字符串 1","字符串 2",…)];
```

例如：

```
string getDocument(in string url)
        raises(NotFound)context("HTML_VERSION");
```

操作的定义是同来自于客户端的调用请求相对应的。在操作的定义中包含操作名(函数名)、数据的传递列表(参数表)和返回值类型。

CORBA的操作有两种,它是由在操作定义的前面是否加有 oneway 来决定的。oneway是用于对操作的属性进行定义的。如果在操作定义的前面没有加上 oneway,则在调用操作时,操作执行的成功与否可以在调用方进行确认,即发生异常时失败,其余情况成功;如果在操作定义的前面加上 oneway,则只能保证调用请求信息的发送,而以后的执行情况就没有任何保证了。换句话说,对于不加 oneway 的操作来讲,直到返回执行结果都一直处于同步等待状态;而对于加有 oneway 的操作来讲,当分布式对象的请求发出以后,可以立即转入下一个处理,而不等待操作的执行。由此可知,oneway 操作不能有返回值,同时 oneway 操作中的参数属性只能是 in,也就是只能从客户端向服务器传递参数,而不能从服务器向客户端传递数据。

操作定义中的参数说明格式如下:

```
in | out | inout   类型名   参数名
```

例如:

```
in string url
```

这里 in、out 和 inout 是参数属性,它用于指定参数的传递方向。同时只能指定一个参数属性。

表 3.4 给出了 in、out 和 inout 参数属性的意义。

表 3.4　操作中参数属性的意义

in	从客户端向 CORBA 对象传递参数
	在调用操作之前,在客户端设置参数区域,在客户端设置参数值。在调用操作时,复制参数值,并将复制的参数值由客户端传递给 CORBA 对象。参数可以是常量,也可以是变量
out	从 CORBA 对象向客户端传递数据
	在操作的处理过程中,在 CORBA 对象中设置参数区域,在 CORBA 对象中设置参数值。操作处理结束以后,复制参数值,并将复制的参数值由 CORBA 对象传递给客户端。参数只能是变量
inout	在客户端和 CORBA 对象之间传递参数和数据
	在调用操作之前,在客户端设置参数区域,在客户端设置参数值。在调用操作时,参数的引用(指针)被从客户端传递给 CORBA 对象。在 CORBA 对象中,在操作的处理过程中,通过使用被传递过来的参数引用,来读取和更新客户端中的参数值。参数只能是变量

在定义操作时参数的属性必须要指定。

操作定义中的 raises,用于列出所有在调用操作过程中可能发生的异常。用 raises 能够列出的异常名只限于事先利用前述的异常定义进行描述的异常。

操作定义中的 context 不是作为参数使用的,它是从客户端向服务器传递数据的手段。

通过指定 context 后面的 context 名，就可以在发出调用请求的同时，来决定所要传递的内容(context)。一般来讲，可以利用 context 来传递 ORB 执行平台的环境变量等信息。

例如，可以在调用操作时利用 context 来指定所要传递的信息的种类(如，LANGUAGE，POSTSCRIPT_VERSION，JDK_VERSION)以及 CORBA 对象所连接的网络能够使用的资源(PRINTER，SCANNER)等。

采用 context 上下文对象来传递非参数信息时，该信息仅在一段时间内有效，而且客户端必须通过 ORB 提供的方法来处理上下文对象。服务器可以根据要求来读取上下文信息，以决定自己的行为方式。同时，上下文对象的取值必须为 string 类型。由 context 所指定的内容，相当于环境变量，它表示要在给定的方法中利用此环境变量的值，其真正的值要在客户端或服务器进行设置。

在客户端调用带有 context 的方法时，需要同时将所设定的 context 变量名及其值也传递给服务器，以便在服务器方进行使用。客户端在设置 context 变量及其值时需要利用系统定义的 context object 对象。

3.3.8 接口定义

IDL 接口定义的格式如下：

```
interface  接口名[:继承的接口名 1,接口名 2,…]
{
     (1) 类型定义
     (2) 常量定义
     (3) 异常定义
     (4) 属性定义
     (5) 操作定义
};
```

例如：

```
interface MatrixCalc
{
       const long SIZE_X = 100;
       const long SIZE_Y = 100;
       typedef long Matrix[SIZE_X][SIZE_Y];
       Matrix add(in Matrix m1,in Matrix m2);
       Matrix sub(in Matrix m1,in Matrix m2);
};
```

说明：

(1) 接口定义是利用前述的类型定义、常量定义、异常定义、属性定义和操作定义等来描述 CORBA 对象的接口。尽管在使用这些定义的顺序上没有什么要求，但对于后面使用的类型定义以及接口等必须在使用前进行定义。

(2) 一个接口定义与 CORBA 对象的一个接口相对应。

(3) 在接口定义中可以继承已有的接口。为了继承已有的接口，可直接在冒号的后面列出要继承的接口名。通过继承而生成的接口中将包含被继承的接口中的所有内容(类型、常量、异常、属性和操作等)。操作的说明不支持多态性(这里的多态性是指具有不同参数个

数和类型的同名操作可以定义多个)。

3.3.9　模块定义

IDL 模块定义的格式如下：

```
module 模块名
{
    (1) 类型说明
    (2) 常量说明
    (3) 异常说明
    (4) 接口说明
    (5) 模块说明
};
```

在 interface 定义中,如果定义的内容很多,则会使各定义之间的关系变得不明确,同时,也容易使各定义中使用的名字出现重复。

模块(module)定义将相关的多个定义放在一个模块中(module),这样,一个模块就可以形成一个有效范围。

如果利用 module 定义,就可以在下面两个模块中定义同名的类型 xxx 和同名的接口 yyy,也就是说不同模块中的类型定义和接口定义等可以重名。

```
module MA
{
  typedef long xxx;
  interface yyy {…};
};
module MB
{
  typedef sequence<long>xxx;
  interface yyy {…};
};
```

在使用模块定义时,本模块内的标识符可直接引用(这就是所谓的命名作用域问题),但其他模块中的标识符不能直接引用。为了引用其他模块中的标识符,就需要在标识符的前面加上“::”以及相应的模块名。例如：

```
module MA
{
   typedef long xxx;
   interface yyy {…};
};
module MC
{
   struct PPP
   {
       string name;
       MA::xxx val;
   };
   interface zzz:MA::yyy
```

```
    {
        ⋮
    };
};
```

在上述定义中，作为 PPP 结构成员的 val 是利用 MA 模块中定义的 xxx 类型来定义的，而接口 zzz 则是通过继承 MA 模块中的 yyy 接口生成的。

在模块定义的内部还可以包含模块定义，以形成模块的层次结构，这时，可通过“::”来进行引用。例如：

```
module MA1
{
  module MA2
  {
     module MA3
     {
        typedef string RRR;
     };
  };
};
module MD
{
   typedef MA1::MA2::MA3::RRR DDD;
};
```

这里，模块 MD 中的类型 DDD 是利用 MA1::MA2::MA3 模块中的 RRR 类型来定义的。

3.3.10 预处理器

IDL 的预处理器同 C++的预处理器具有等价的功能，其主要内容如表 3.5 所示。

表 3.5 IDL 预处理器

功　能	举　例
注释(/*…*/,//)	/* This is comment */ String name; //length<10
宏定义(#define,#undef)	#define MaxSIZE 100 #define SEQ(type)sequence<type>
条件编译(#if,#ifdef, #ifndef,#else,#endif)	#ifndef BASIC_H #define BASIC _H #endif
文件包括(#include)	#include“comment. idl”
行号控制(#line)	#line 100
依赖于实现的动作(#pragma)	#pragma iiop

由表 3.5 可知，IDL 预处理器与 C++语言中的预处理器在功能上基本是一样的，下面仅对#line 和#pragma 进行简单说明。

1. #line 行号

#line 预处理指令表示设定该行所在的下一行的行号。例如，假如在接口定义中有下面两条语句：

```
#line 100
const short a = 200;
```

它表示第二条语句所在行号为 100。

设置#line 预处理指令的主要目的是用于确定当发生错误时该错误所在的行号。

2. #pragma 操作

#pragma 指令用于在程序中向编译器发出指示，以便使编译器能够进行相应的处理。一般向编译器发出指令是在命令行上进行的，而#pragma 则可以在程序中向编译器发送指令。具体能够发送何种指令，完全是由编译器决定的。

3.4　从 IDL 到 Java 的映射

CORBA 对象可以利用各种语言来实现，说某种语言能够实现 CORBA 对象是指存在从 IDL 到那种语言的映射。例如，在 CORBA 2.2 中，就规定了从 IDL 到 C、C++和 Java 等语言的映射规则。只有将 IDL 定义的接口信息映射为相应的语言之后，才能在程序中利用这种语言来进行分布式对象系统的设计。本节主要介绍从 IDL 到 Java 语言的映射过程，由于即使映射为相同的语言，但如果 CORBA 系统不同其映射结果也可能是不一样的，因此，这里只是给出一种可能的映射结果。

3.4.1　接口定义的映射

在 IDL 到 Java 的映射过程中，一个 IDL 接口(interface)定义被映射为一个 Java 语言的接口以及 4 个主要的类。

下面以下述的 HelloServer 的 IDL 接口定义为例来说明 IDL 编译器的映射结果。

```
interface HelloServer
{
    string sayHello(in string name);
};
```

HelloServer 接口经 IDL 编译器映射为 Java 语言之后，所生成的 Java 语言程序主要是由下述五个部分组成，其中的 Java 语言接口以及各类的名字是以 IDL 接口的名字为基础来命名的。下面给出每一部分的映射结果并说明其具体作用。

1. Java 语言接口

IDL 接口定义中的 interface 首先被映射为 Java 语言中的 interface，其映射结果如下：

```
public interface HelloServer extends org.omg.CORBA.Object
```

```
{
    String sayHello(java.lang.String name);
}
```

其中，org.omg.CORBA.Object 定义了所有 CORBA 对象的通用方法，以后将具体介绍。

作为映射结果的 Java 语言接口，是需要在下面介绍的 stub 类和 skeleton 类中进行实现的接口。stub(代理对象)是通过该接口被客户端所调用的，skeleton 则是通过具有该接口的实现程序来调用每个操作的。该接口的定义必须要继承 org.omg.CORBA.Object 接口。在 org.omg.CORBA.Object 接口中，定义了 CORBA 对象应该具有的功能，其中包括对象引用的复制、比较以及用于生成 Request 类的实例等函数。org.omg.CORBA.Object 接口的实现是由各厂家提供的，用户不需要实现。

2. skeleton 类

skeleton 类的定义如下：

```
public abstract class  _HelloServerImplBase
                        extends org.omg.CORBA.DynamicImplementation
                        implements HelloServer
{
    ⋮
}
```

其中，_HelloServerImplBase 是 skeleton 类名，它是随着系统的不同而不同的。skeleton 类中的具体内容将在后续的章节中进行介绍。

skeleton 类的作用是作为分布式对象实现的基类。由上述定义可知，skeleton 类本身是由 IDL 编译器产生的抽象类。skeleton 类的父类是随厂家的不同而不同的，在这里其父类是 org.omg.CORBA.DynamicImplementation。

skeleton 类的父类主要完成将发送给 CORBA 对象的请求信息传递给 skeleton 类。

由上述 skeleton 类的定义不难看出，作为映射结果的 Java 语言的接口 HelloServer，在 skeleton 类中要进行实现。

3. stub 类

stub 类的定义如下：

```
public class  _HelloServerStub
              extends org.omg.CORBA.portable.ObjectImpl
              implements HelloServer
{
    ⋮
    String sayHello(java.lang.String name){…}
    ⋮
}
```

其中，_HelloServerStub 是 stub 类名，它是随着系统的不同而不同的。stub 类中的具体内容将在后续的章节中进行介绍。

stub 类是作为客户端程序的代理对象的类由 IDL 编译器生成的。stub 类必须要继承

org. omg. CORBA. portable. ObjectImpl 类。org. omg. CORBA. portable. ObjectImpl 类主要完成从 stub 类中取出请求信息并将其传递到 CORBA 对象中的任务。

由 stub 类的定义不难看出，作为映射结果的 Java 语言接口 HelloServer，要在 stub 类中给出其定义。由于 stub 是客户端的代理对象，因此，在客户端调用分布式对象时并不是直接调用服务器上的分布式对象，而是调用 stub 类中的同名方法，在该方法中来完成分布式方法的调用过程。这样，从程序设计的角度来看，对分布式方法的调用就如同调用本地方法一样的简单，许多复杂的调用细节都被 stub 类屏蔽掉了。

4. Holder 类

Holder 类是与 IDL 数据类型相对应的 Java 语言的类，即每个 Holder 类中都用于保存着一种类型的数据。每一个接口定义、每一个基本数据类型以及每一个用户自定义类型都有一个 Holder 类与其对应。

在利用 IDL 来定义接口信息时，可以指定操作的每个参数的传递方向（in、out 和 inout），这种处理方式与 Java 语言中的参数处理方式是不同的，下面对参数的处理方式进行简单的说明。

1）in 参数

如果参数的传递方向被指定为 in，则表示只是从客户端向服务器传递参数。在服务器中的参数修改不会被传递给客户端。当将 IDL 操作中的 in 参数映射为 Java 语言中的基本数据类型时，正好能够满足这一要求，这是因为以 Java 语言中的基本数据类型作为参数所采用的就是传值的方式，也就是只能从客户端向服务器传递参数。

2）out 参数

如果参数的传递方向被指定为 out，则只能从服务器向客户端传递参数值，客户端不能向服务器传递参数，服务器也不能取参数。参数的这种传递方式利用 Java 语言中的参数处理方式是无法实现的。为此，为了实现 out 参数传递，需要利用 Holder 类这样的间接方式来实现。

在所有的 Holder 类中，都定义了保存对应类型数据的公有的数据成员和两个构造函数，其中一个构造函数是不为数据成员设置任何初值的默认构造函数，另一个构造函数是具有参数的、用于为数据成员设置初值的。

对于 out 参数来讲，一般是利用默认构造函数来生成对应参数类型的 Holder 类的对象，并作为参数传递给服务器。这时，在客户端的 ORB 中不处理 out 参数中的值，在来自客户端的请求信息中不包含 out 参数（尽管不包含 out 参数，但相应的 Holder 类对象还必须要创建，这是由于在从服务器返回参数值时需要利用该 Holder 类）。

在服务器端的 ORB 中，利用默认构造函数来生成不保存数据的 Holader 类，并传递给分布式对象实现，在分布式对象实现中将处理结果保存到 Holder 对象中。在服务器端的分布式方法执行完之后，在服务器端的 ORB 中取出保存在 Holder 对象中的参数值并生成回答信息（reply message）后发送给客户端。在客户端的方法调用返回之后，在客户端程序中通过访问 Holder 类中的数据成员，就可以使用从服务器返回的参数数据。

3）inout 参数

如果参数的传递方向被指定为 inout，则既从客户端向服务器传递参数，同时在分布式

方法执行完了之后，也通过该参数向客户端传递数据。在 Java 映射中，这种方式的参数传递需要利用 Holder 类这样的间接方式来实现。

在 inout 参数处理中，为了从客户端向服务器传递数据，可以利用 Holder 类中能够指定初值的带参数的构造函数来生成 Holder 类的对象，并以该 Holder 对象为参数来调用分布式方法。在客户端的 ORB 中，通过利用 Holder 类中的数据来生成请求信息（request message）并将其发送给服务器。

服务器端的 ORB 从请求信息中取出数据，生成保存该数据的 Holder 对象并传递个分布式对象实现。在分布式对象实现中，从 Holder 对象中取出数据并利用之，同时将返回给客户端的参数值保存到 Holder 对象中。

在服务器端的分布式方法执行完之后，在服务器端的 ORB 中取出保存在 Holder 对象中的参数值并生成回答信息（reply message）后发送给客户端。在客户端的方法调用返回之后，在客户端程序中通过访问 Holder 类中的数据成员，就可以使用从服务器返回的参数数据了，这一过程与 out 参数处理完全一样。

上述 IDL 接口定义被映射为 Java 语言中的 Holder 类的形式如下：

```
public final class HelloServerHolder
{
    ⋮
}
```

在 IDL 接口定义中描述的接口定义、用户自定义类型以及 IDL 可用的基本类型等都有一一对应的 Holder 类定义。

基本类型的 Holder 类在 CORBA 中已定义好，需要时直接利用即可。用户定义类型的 Holder 类在 IDL 编译时产生。有关 Holder 类的具体内容将在后续章节中详细叙述。

5. Helper 类

IDL 接口定义被映射为 Java 语言中的 Helper 类的形式如下：

```
public final class HelloServerHelper
{
    ⋮
}
```

作为映射结果的 Helper 类用于提供各种实用功能。例如，对 any 类型数据的操作以及 narrow 方法等。在 Java 映射中，在 IDL 接口定义中描述的接口定义和用户定义类型都有一一对应的 Helper 类映射结果。有关 Helper 类的详细说明请见后续章节。

3.4.2 实现引用传递的 Holder 类

在利用 IDL 编译器来实现服务器和客户端程序时，开发者可以直接与 Holder 类打交道。因此，在本节再详细介绍一下 Holder 类。

如前所述，Holder 类是为了在调用接口中提供的方法时实现引用传递而定义的类。所谓引用传递，是指不是单纯的传递数据，而是将该数据的地址引用也传递过去。这种参数传递方式在 C/C++语言中可通过指针来实现。

需要进行引用传递的接口定义的例子如下：

```
interface HelloServerByRef
{
    void sayHello(in string name,out string reply);
};
```

在此定义接口中，sayHello 函数的返回值为 void，参数 name 的属性为 in，而参数 reply 的属性则为 out 类型。

out 类型的参数用于将数据从 CORBA 对象传递到客户端。在各种语言的映射中，可直接利用语言中“引用传递”的功能来实现 out 类型的处理。同时，inout 类型的参数也具有引用传递功能。

由于 Java 中的基本类型数据作参数时只能传值，因此在映射的时候，就不能直接利用 Java 中的基本类型数据来进行 out 和 inout 参数传递。

如前面所述，在 CORBA 中是利用 Holder 类来处理 out 和 inout 参数的，这本身也是利用 Java 语言中对象作为函数参数是采用引用传递的特点，当然，由于客户端和服务器不在一个进程中，采用 Holder 类来处理 out 和 inout 参数是属于一种间接的处理方式，与同一个进程中的引用传递是不同的。上述 HelloServerByRef 接口中的 sayHello 方法可映射为如下的 Java 方法：

```
void sayHello(java.lang.String name,CORBA.StringHolder reply)
{
    ⋮
}
```

从上述映射结果不难看出，IDL 语言中的 in 属性的 string 类型被映射为 Java 语言中的 java.lang.String 类型，而 out 属性的 string 类型则被映射为 CORBA.StringHolder 类。

在 IDL 到 Java 语言的映射处理中，如果在方法中指定了 out/inout 类型的参数，则此时不是将其直接映射为 Java 语言中的基本数据类型，而是将其映射为那种类型的 Holder 类。

由于在程序设计中可以直接使用 Holder 类，因此下面给出几个 Holder 类的例子。

1. long 类型的 Holder 类

由于 IDL 中的 long 类型与 Java 语言中的 int 类型等价（都是 32 位），因此属性为 out 和 inout 的 IDL 中的 long 类型参数将被映射为 Java 语言中的 IntHolder 类，该类的定义如下：

```
final public class IntHolder
{
    public int value;
    public IntHolder(){ }
    public IntHolder(int initial)
    {
        value = initial;
    }
```

```
}
```

由于所有成员都是 public 的(包括数据成员),因此,在任何函数中都可以利用“对象名.value”取出 value 的值。

2. string 类型的 Holder 类

属性为 out 和 inout 的 IDL 中的 string 类型参数将被映射为 Java 语言中的 StringHolder 类,该类的定义如下:

```
final public class StringHolder
{
      public java.lang.String value;
      public StringHolder(){ }
      public StringHolder(java.lang.String initial)
      {
            value = initial;
      }
}
```

3. 接口以及用户定义类型的 Holder 类的基本形式

下面的程序段给出了接口及用户定义类型的 Holder 类的基本形式。

```
final public class xxxxHolder
                  implements org.omg.CORBA.portable.Streamable
{
      public xxxx value;                                  //相应类型的变量
      public xxxxHolder(){ }
      public xxxxHolder(xxxx initial){ value = initial;}
      public void _read(org.omg.CORBA.portable.InputStream i)
      {…}                                                 //从“流”中读数据到 value 中
      public void _write(org.omg.CORBA.portable.OutputStream o)
      {…}                                                 //将 value 数据写入“流”中
      public org.omg.CORBA.TypeCode _type(){…}            //返回 xxxx 类型的 TypeCode
}
```

其中,xxxx 表示接口以及用户定义类型的标识符。

前已述及,在 IDL 到 Java 的映射过程中,不仅仅映射到 Java 语言的基本数据类型,对于在 IDL 中能够使用的所有数据类型(包括用户定义类型)来讲,被指定为 out/inout 属性的参数的传递处理都是通过使用 Holder 类来实现的。

3.4.3 提供各种实用功能的 Helper 类

Helper 类也是由 IDL 编译器产生的,程序设计者可以直接利用其功能。不但接口定义,像结构、联合这样的用户定义类型也被映射为相应的 Helper 类,其中提供了利用接口及用户定义类型的功能,但绝大部分方法都是由 stub 和 skeleton 类使用的,只有用于 Any 类型操作的方法以及 narrow 方法等,在程序设计时才能使用到。

Helper 类提供了使用接口及用户定义类型来编写程序时所需要的实用方法,其定义的

基本形式如下。

```
public class xxxxHelper
{
```

(1) 用于 Any 类型处理。

```
public static void insert(org.omg.CORBA.Any a,xxxx t)
{…}     //将 xxxx 类型数据插入到 Any 类型变量中
public static xxxx extract(Any a)
{…}     //从 Any 类型中抽取出 xxxx 类型数据
```

(2) 用于取出类型码(Typecode)。

```
public static org.omg.CORBA.TypeCode type()
{…}     //返回 xxxx 类型的 TypeCode 对象
```

(3) 用于取出 RepositoryID。

```
public static string id()
{…}
```

(4) 用于序列化(serialize)。

```
public static xxxx read(org.omg.CORBA.portable.InputStream istream)
{…}
public static void write(org.omg.CORBA.portable.OutputStream ostream,xxxx value)
{…}
```

(5) 下面的方法仅在有 interface 定义的时候存在。

```
    public static xxxx narrow(org.omg.CORBA.Object obj)
    {…}
}
```

其中,xxxx 表示接口以及用户定义类型的标识符。

下面根据程序中的序号对上述 Helper 类的定义进行简单的说明。

(1) 用于将 xxxx 类型的对象或者接口插入 Any 类型的对象中,或者从 Any 对象中取出 xxxx 类型的对象或接口。

(2) 由 IDL 定义(描述)的类型,都具有一个被称为类型码(TypeCode)的字符串类型名,此方法就是为了取出类型码的。

(3) 用于取出 xxxx 的类型定义或者接口定义在接口仓库中的 ID,即 Repository ID。

(4) 用于将 xxxx 类型的对象或接口变换成“流式”数据,即序列化处理。

(5) 用于生成代理对象,并将 CORBA 对象的对象引用与其代理对象的引用进行关联处理。也就是根据由服务器传递过来的对象引用来生成相应 stub 类的代理对象。

Helper 类中的 narrow 方法在程序设计过程中是经常被使用的方法,该方法的具体意义在前面的章节中已经进行了详细说明。

3.4.4　其他 IDL 定义的映射

1. 模块定义的映射

IDL 中的模块定义(module)是为了形成名字的作用域而使用的定义。在 IDL 到 Java

的映射中，模块定义被映射为 Java 语言中用于形成名字作用域的包(package)。表 3.6 给出了 IDL 模块定义的一种可能的映射结果。

表 3.6 IDL 模块定义的 Java 映射结果

IDL 定义	IDL 到 Java 映射结果

module test
{
interface InterTest
{…};
};
	package test;
	public interface InterTest
	extends org.omg.CORBA.Object
	{…}

其中，作为 module 名的 test 被映射为 Java 语言的 package 名。

2. 常量定义的映射

IDL 的常量定义有两种形式，一种是常量直接在模块中定义，另一种是常量在接口中定义。这两种定义的 Java 映射结果是有区别的。表 3.7 给出了 IDL 常量定义的一种可能的映射结果。

表 3.7 IDL 常量定义的 Java 映射结果

IDL 定义	IDL 到 Java 的映射结果

直接在模块中定义的常量
const 类型 标识符＝常量；
	public interface 标识符{
	public static final 映射后的类型
	value＝常量；
	}

在接口中定义的常量
public interface test {
const 类型 标识符＝常量；
};
	public interface test {
	public static final 映射后的类型
	标识符＝常量；
	}

由表 3.7 映射结果可知，不管是直接在模块中定义的常量还是在接口中定义的常量，都被映射为 Java 语言接口中的常量。例如：

```
module test
{
    const string do = "Nothing";
};
```

经 IDL 编译器映射之后，将生成如下 Java 语言代码：

```
package test;
public interface do
{
    public static final java.lang.String value = "Nothing";
}
```

例如：

```
module test
{
    interface contest
    {
        const string do = "Nothing";
    };
};
```

经 IDL 编译器映射之后，将生成如下 Java 语言代码：

```
package test;
public interface contest
{
    public static final java.lang.String do = "Nothing";
}
```

3. 属性定义的映射

IDL 中的属性定义有两种方式，一种是带有 readonly 关键字，另一种是不带该关键字。这两种定义方式的映射结果是不同的。表 3.8 给出了 IDL 属性定义的一种可能的映射结果。

表 3.8　IDL 属性定义的 Java 映射结果

IDL 定义	IDL 到 Java 的映射
attribute 类型 标识符；	映射后的类型　标识符()； void 标识符(映射后的类型 value)；
readonly attribute 类型　标识符；	映射后的类型　标识符()；

由表 3.8 可知，不带 readonly 关键字的属性定义，被映射为两个函数说明，一个具有 get 功能(取出属性值)，一个具有 set 功能(设置属性值)。而带有 readonly 关键字的属性定义，则被映射为一个函数说明，该函数只具有 get 功能(取出属性值)。例如：

假如有如下 IDL 属性定义：

```
attribute long count;
```

该属性定义的 Java 映射结果如下：

```
int count();
void count(int value);
```

又如，假如有如下 IDL 属性定义：

```
readonly attribute long count;
```

该属性定义的 Java 映射结果如下：

```
int count();
```

4. 异常定义的映射

表 3.9 给出了 IDL 异常定义及其可能的映射结果。

表 3.9 IDL 异常定义的 Java 映射结果

IDL 定义	IDL 到 Java 的映射

exception 标识符 {
　　类型 1　变量 1;
　　⋮
　　类型 n　变量 n;
};
	final public class　标识符
	　　extends org. omg. CORBA. UserException {
	　　public 映射后的类型 1　变量 1;
	　　⋮
	　　public 映射后的类型 n　变量 n;
	　　public 标识符()　{…}
	　　public 标识符(
	　　映射后的类型 1　变量 1,
	　　⋮
	　　映射后的类型 n　变量 n)
	　　{
	　　⋮
	　　}
	}
	final public class　标识符 Holder implements
	　　org. omg. CORBA. portable. Streamable
	{…}

由表 3.9 可知，IDL 中的异常定义被映射为 Java 语言中的两个类，一个是异常处理类，另一个则是与异常定义对应的 Holder 类。例如：

```
//假如有如下 IDL 接口定义
module Sample{
     exception UnknownException { };
};
//该 IDL 定义的 Java 映射结果如下
package Sample;
final public class UnknownException
                  extends org. omg. CORBA. UserException
{
     public UnknownException(){…}
}
final public class UnknownExceptionHolder
                  implements org. omg. CORBA. portable. Streamable
{…}
```

5. 操作定义的映射

IDL 中的操作定义，相当于 Java 语言中的方法定义。表 3.10 给出了 IDL 操作定义及

其可能的映射结果。

表 3.10　IDL 操作定义的 Java 映射结果

IDL 定义	IDL 到 Java 的映射
类型　操作名(typel 类型 1　变量名 1, …,typel 类型 n　变量名 n) raises(异常 1,…,异常 n)	映射后的类型　方法名(映射后的类型 1　变量 1, …,映射后的类型 n　变量 n) throws 映射后的异常 1,…, 映射后的异常 n;

由表 3.10 可知,IDL 中的操作定义,被映射为 Java 语言中的方法,如果有异常抛出的话,也需要进行相应的映射。例如:

假如有如下 IDL 操作定义:

```
long div(in long x,in long y)raises(ZeroException);
```

该 IDL 定义的 Java 映射结果如下:

```
int div(int x,int y)throws ZeroException;
```

又如,假如有如下 IDL 操作定义:

```
oneway void PrintHistory()context("Encoding","File");
```

该 IDL 定义的 Java 映射结果如下:

```
public void PrintHistory(org.omg.CORBA.Context _c);
```

由映射结果可知,如果在操作定义时包含 context 项,则在映射时,系统自动将 context 对象添加到参数列表中,也就是在原有参数的基础上再增加一个参数,以便传递 context 信息。

6. 用户定义类型的映射

在 IDL 接口定义语言中,用户定义类型包括 struct(结构)、union(联合)和 enum(枚举)等。为了叙述方便,先定义如下 IDL 接口:

```
module Test
{
    struct StrType
    {
        long field1;
        string field2;
    };
    union UnionType switch(short)
    {
        case 1:string a;
        case 2:long c;
    };
    enum Day{Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday};
```

```
    ⋮
};
```

1）结构定义的映射

上面 Test 模块中定义的结构 StrType 的 Java 映射结果如下。

```
final public class StrType
{
    //与结构成员对应的域的定义
    public int field1;
    public java.lang.String field2;
    //构造函数
    public StrType(){ }
    public StrType(int field1,java.lang.String field2)
    {
        this.field1 = field1;
        this.field2 = field2;
    }
    ⋮
}
```

从上面的映射结果不难看出，IDL 中的结构被映射为与其同名的 Java 语言的 final 类。此类是由与结构成员相对应的公共数据成员和构造函数构成的。由于每个数据成员的访问权限都是 public，因此，对结构成员的访问可以通过对类中对应的数据成员的访问来实现。

2）联合定义的映射

IDL 中的联合的映射处理是比较复杂的。上述 Test 模块中定义的联合 UnionType 被映射为如下形式的 Java 类。

```
public final class UnionType
{
    private short _discriminator;           //区分符
    private java.lang.Object _value;        //存放联合值的对象
    public UnionType()                      //构造函数
    {…}
    public short discriminator()throws org.omg.CORBA.BAD_OPERATION
    {  …
      return _discriminator;                //获取区分符
    }
    public String a()throws org.omg.CORBA.BAD_OPERATION
    {  …
      switch(_discriminator)
      {
          case 1:
              break
          default:
              throw new org.omg.CORBA.BAD_OPERATION();  //出错
      }
      return((org.omg.CORBA.StringHolder)_value).value;
    }
    public void a(String value)             //为 a 元素赋值
```

```
    {  …
       _discriminator = 1;                    //设置区分值为 1
       _value = new org.omg.CORBA.stringHolder(value);
    }
    ⋮
}
```

从上面的映射结果不难看出，IDL 中的联合被映射为与其同名的 Java 语言的 final 类。同时，在类中为每个数据成员都定义了 get 方法（比如，String a()）和 set 方法（比如，void a (String value)）。在 get 方法中先判断区分符，如果正确的话，返回对应的值，否则抛出异常信息；在 set 方法中，先设置区分符的值，然后再将相应的值保存到数据成员中。尽管在上述映射结果中只给出了 string 类型的数据成员 a 的映射结果，但从中可以看出联合的映射特点有两个：一是利用区分符来标识当前所保存的数据类型，这是由于在任一时刻联合变量中只能保存一种类型的数据；二是所有数据成员都被保存在 java.lang.Object 类型的成员变量中，这是由于在 Java 语言中，java.lang.Object 是所有类的基类的缘故。这里，org.omg.CORBA.BAD_OPERATION 是 CORBA 定义的系统异常，它表示请求了一个不正确的操作。

3）枚举定义的映射

上述 Test 模块中定义的枚举 Day 被映射为如下形式的 Java 类。

```
public final class Day
{
    //与枚举元素对应的各常量的定义
    public static final int _Sunday = 0,
                            _Monday = 1,
                            _Tuesday = 2,
                            ⋮
                            _Saturday = 6;
    //与枚举元素对应的各对象的定义
    public static final Day Sunday = new Day(_Sunday);
    public static final Day Monday = new Day(_Monday);
    ⋮
    public static final Day Saturday = new Day(_Saturday);
    public int value()
    {
        return _value;          //返回枚举元素值(整数)
    }
    //根据给定的整数来获取相应的对象：
    public static final Day from_int(int i)throws org.omg.CORBA.BAD_PARAM
    {
          switch(i)
          {
               case _Sunday:
                         return Sunday;
               case _Monday:
                         return Monday;
               ⋮
               case _Saturday:
                         return Saturday;
               default:
```

```
                throw new org.omg.CORBA.BAD_PARAM();
            }
        }
        private Day(int _value)     //构造函数
        {
            this._value = _value;
        }
        private int _value;         //存放枚举元素值的整型变量
    }
```

从上面的映射结果不难看出，IDL 中的枚举被映射为与其同名的 Java 语言的 final 类。这里，org.omg.CORBA.BAD_PARAM 是 CORBA 定义的系统异常，它表示传递了一个不正确的参数。

7. sequence 的映射

sequence 被映射为 Java 语言中的数组。

假如有下述 IDL 定义：

```
sequence<long,20>x;
```

该 IDL 定义的 Java 映射结果如下：

```
int x[20];
```

8. string 和 wstring 的映射

string 和 wstring 都被映射为 Java 语言中的 String 类。

9. 数组的映射

IDL 中的数组被映射为 Java 语言中的数组。

假如有下述 IDL 定义：

```
long LArr[2][3];
```

该 IDL 定义的 Java 映射结果如下：

```
int LArr[2][3];
```

10. 整型量的映射

IDL 中的整型量包括 short 和 long 等。表 3.11 给出了 IDL 整型量的定义及其可能的映射结果。

表 3.11 IDL 整型量的 Java 映射结果

IDL 数据类型	Java 数据类型	IDL 数据类型	Java 数据类型
short	short	unsigned long	int
unsigned short	short	long long	long
long	int	unsigned long long	long

由于 Java 语言中没有无符号整数，所以对无符号数的处理需要在程序中进行。

11. char 和 wchar 的映射

IDL 中的 char 和 wchar 类型都被映射为 Java 中的 char 类型。

12. boolean 类型的映射

IDL 中的 boolean 类型被映射为 Java 语言中的 boolean 类型。

13. octet 类型的映射

IDL 中的 octet 类型被映射为 Java 语言中的 byte 类型。
假如有下述 IDL 定义：

```
octet x;
```

该 IDL 定义的 Java 映射结果如下：

```
byte x;
```

14. any 类型的映射

IDL 中的 any 类型被映射为 org. omg. CORBA. Any 类型。

15. fixed 类型的映射

IDL 中的 fixed 类型被映射为 Java. lang. BigDecimal 类型(但有的系统不可用)。
假如有下述 IDL 定义：

```
typedef fixed<20,2>Dollar;  //整数 18 位,小数 2 位的固定小数类型
Dollar Deposit(in Dollar amount);
```

该 IDL 定义的 Java 映射结果如下：

```
//存款 amount = $250.00
BigDecimal amount = BigDecimal.valueOf(25000,2);
//调用 Deposit
BigDecimal newBalance = account.Deposit(amount) ;
```

16. 浮点类型的映射

IDL 中的浮点类型包括 float 和 double 等。表 3.12 给出了 IDL 浮点类型的定义及其可能的映射结果。

表 3.12　IDL 浮点类型的 Java 映射结果

IDL	Java
float	float
double	double
long double	没有对应类型

3.4.5 IDL 映射后的使用

前面介绍了从 IDL 定义到 Java 语言的映射过程。这里需要注意的是，语言映射的目的是为了在实现客户端或服务器程序时能够使用这些映射结果。下面简单介绍几个 Java 映射结果的使用过程。主要介绍用户定义类型的映射结果的使用，所使用的结构类型定义、联合类型定义和枚举类型定义及其 Java 语言映射等请参考“用户定义类型的映射”部分内容。

1. IDL 结构定义映射结果的使用

下面的程序段给出了利用 IDL 结构定义映射结果的过程。

```
try
{
    StrType c = new StrType(50,"ABC");
    ⋮
    System.out.println(c.field1);
    System.out.println(c.field2);
    ⋮
}
catch(…)
{
    ⋮
}
```

由上述使用过程可知，对 IDL 结构定义映射结果的使用，同使用 Java 语言中的一般的类是一样的，先利用带有参数的构造函数来定义对象，在需要时，可直接利用对象来访问其中的数据成员，这是由于每个数据成员的访问权限都是 public。

2. IDL 联合定义映射结果的使用

对 IDL 联合定义映射结果的使用是由两部分组成的：一是写操作，二是读操作。

```
//写操作
try
{
    UnionType u = new UnionType();
    String  s = "String data";
    u.a(s);                          //设置区分符并保存数据
    ⋮
}
catch(…)
{…}
//读操作
try
{
    switch(u.discriminator())    //判断其中的数据类型
    {
        case 1:
```

```
            String s = u.a();
            System.out.println(s);
            break;
        case 2:
            ⋮
    }
}
catch(…)
{…}
```

对联合变量的使用可分为两部分，一是读操作，另一个是写操作。在读操作之前先要根据区分符来判断联合变量中当前所保存的数据类型，根据该类型来读取对应的数据；在写操作之前先要生成对象，然后调用相应的 set 方法（比如，a(String)）向联合对象中保存数据，需要注意的是，在保存数据之前先要设置区分符。

3. IDL 枚举定义映射结果的使用

对 IDL 枚举定义映射结果的使用也是由两部分组成的：一是写操作，二是读操作。

```
//写操作
try
{
    Day day = Day.from_int(Day._Sunday);  //_Sunday 是静态变量
    ⋮
}
catch(…)
{…}

//读操作
try
{
    ⋮
    String dayString;
    switch(day.value())
{
    case Day._Sunday:
        dayString = "Sunday";
        break;
    case Day._Monday:
        dayString = "Monday";
        break;
    ⋮
    case Day._Saturday:
        dayString = "Saturday";
        break;
    default:
        dayString = "Invalid value";
    }
}
catch(…)
```

{…}

与联合变量的使用类似，对枚举变量的使用也是由两部分组成的：一是写操作，二是读操作。写操作主要就是根据枚举常量来创建枚举对象。读操作主要就是根据枚举对象中的枚举常量所对应的值来获取所对应的枚举常量的名称。

在学过本章的内容之后，就可以继续学习后续章节的内容了，其中包括 CORBA 客户端设计和服务器设计。在这些章节中将具体介绍 CORBA 应用系统的程序设计过程。