抽象形式语法(ASN.1 ITUT-T X.680)的对象封装及协议数据单元的 PER (Packed Encoding Rules ITU-T X.691)编码实现*

余爱清　余胜生　周敬利
华中理工大学计算机系国家外存储系统专业实验室　(武汉　430074)
e-mail:yuaq@wtwh.com.cn

摘　要　抽象形式语法(ASN.1)标注的网络协议数据单元可读性好,但是用传统的过程调用的方法去实现这些协议单元的编解码工作将是一件非常复杂而繁重的工作,文章使用面向对象的思想对抽象形式语法的基本数据类型进行封装,在对抽象形式语法标注的协议数据单元不作改动或稍作改动的情况下,直接利用原有的协议数据单元定义就完成这些协议数据单元的编解码工作,从而达到简化的目的。该文将重点叙述 ITU-T 的抽象形式语法的对象封装,并以 ITU-T 的 X.691 PER 编码作为例子来叙述抽象形式语法对象的编码方法是如何简化抽象形式语法所表示的协议数据单元的程序实现的。

关键词　抽象形式语法　对象封装　ASN.1　X.680　PER　X.691

Object-oriented Implementation for ASN.1 of ITU-T X.680 in Packed Encoding Rules of ITU-T X.691

Yu Aiqing　Yu Shengsheng　Zhou Jingli
(Huazhong University of Science and Technology,Wuhan 430074)

Abstract: the paper shows how object-oriented implementation of built-in types of ASN.1 of ITU-T X.680 can simplify the encoding and decoding procedures of networking protocols annotated with ASN.1.

keywords: object oriented implementation, ASN.1, ITU-T X.680, ITU-T X.691

1　基本原理和简单数据类型的封装

抽象形式语法(ASN.1)是 ITU-T 推荐的一个网络协议描述语言标准,它主要用来描述定义网络通讯中的信息形式。该标准[1]:定议了许多简单基本数据类型;提供了用基本数据类型构造复合数据类型的方法。抽象形式语法并不局限于用来定义应用协议,还可用来对抽象信息进行描述定义。

一般来说,描述定义一个应用协议是一件很复杂的工作,然而将已定义好的一个应用协议代码化也不是一件容易的事。在文中,将抽象形式语法的数据类型用对象类封装后,可以简化抽象形式语法表示的应用协议的编码与解码过程,并保持抽象形式语法的简洁性与可读性。

无论是抽象形式语法的简单基本数据类型,还是其复合数据类型都将被封装为对象类,并且编码规则用对象方法来封装,其实现原理如下:

```
struct CType : CBasic
{ //下列方法对 ITU-T X690 BER 编码规则进行封装
   void BER_Encode(CBitStreamBuffer*);
   void BER_Decode(CBitStreamBuffer*);
 //下列方法对 ITU-T X691 PER 对齐方式编码规则进行封装
   void PERA_Encode(CBitStreamBuffer*);
   void PERA_Decode(CBitStreamBuffer*);
 //下列方法对 ITU-T X691 PER 非对齐方式编码规则进行封装
   void PERUA_Encode(CBitStreamBuffer*);
   void PERUA_Decode(CBitStreamBuffer*);
   CXX value; //表示抽象形式语法数据类型的实际值
};
```

CType——表示抽象形式语法数据类型值 CXX value 被封装后的对象类。

CBitStreamBuffer——比特流缓冲区,供对象进行编码和解码用

CBasic——一个非常简单的基类,只有唯一的一个属性成员"bool present",该值用来表示一个具有 DEFAULT 或 OPTIONAL 属性的数据成员是否有值

根据 X.680 的定义,抽象形式语法的数据类型可分为两大类[2]:

简单基本数据类型:

BitStringType, BooleanType, CharacterStringType, EmbeddedPDVType, ExternalType, InstanceOfType, IntegerType, NullType ObjectClassFieldType, ObjectIdentifierType, OctetStringType, RealType, TaggedType

复合数据类型:

*该文章得到军口项目"K 口 Modem 的可视电话的开发与研究"支持。

作者简介:余爱清,博士生,从事过如下项目的开发研究:国家 863 武汉邮电科学研究院 CIMS 工程的开发研究;国际合作项目可视电话的开发;军口项目"K 口 Modem 的可视电话的开发";主持多媒体交互软件电子白板的开发。余胜生,周敬利:教授,博导,主要从事磁盘伺服刻写和网络多媒体的存贮与传输的研究。

ChoiceType, EnumeratedType, SequenceType, SequenceOfType, SetType, SetOfType

除了以上数据类型外，还有一个非常重要的数据类型——限制性数据类型[3]，它是基本数据类型附加上一些限制条件后形成的一种数据类型。有如下一些限制条件：值域限制条件，大小限制条件，OPTIONAL，以及数据类型结构是否具有可扩展性。实际上，在协议数据单元定义中，绝大多数的数据类型都表现为限制性数据类型。在某些编码规则中，比如 X.691 中，这些限制性条件是编码规则的极其重要的参数。在应用协议代码化过程中，这些限制条件最好能在数据类型实例化时确定下来。一般的想法是，对象类的构造函数是传递这些限制条件参数的最佳选择。然而，事情并不如此简单。因为在新数据类型的构造过程中，数据成员变量不能用其对象类的构造函数直接进行初始化，如下所示：

```
struct CNewDataRecord
{CSomeDatatype a (3,5);//错误，用构造函数直接初始化变量 a;
 ...
};
```

为了使协议数据单元的定义与其代码化的形式一致，此处使用了"模板类(template class)"的概念，如下例所示：

```
template<int parm1,float param2 >    struct CNewRecord
struct CType1{                       { CType1<2,4.0> a; //在
                                              C++ 中正确
      ......
};                                   };
```

因此，利用模板类可以解决抽象形式语法对象类初始化过程中限制性条件参数的传递。在 C++中模板有如下的语法表示形式：

template < [typelist] [,[arglist]] > declaration[4]

上述语法形式描述了模板类或模板函数的定义。template 关键词的参数列表是一系列抽象对象类名，或普通函数参数的列表。这些模板参数对模板定义体内的所有方法均是可见的。declaration 是普通函数名或普通对象类名的声明。在模板类的实例化(Instance)过程中，除了模板参数必须出现在"< >"中外，其它形式与普通类的实例化完全相同。下面的例子是抽象形式语法中整型数据类型的模板对象类的完整定义(任何其它简单基本数据类型都可类似定义)：

```
template<CTAGS Tag,int TagNo,int lowerBound=MIN,int upper-
        Bound=MAX,bool ExtMarker = false>
struct TCINTEGER : public CBasicType
{ //下列方法对 ITU-T X691 PER 对齐方式编码规则进行封装
    void PERA_Encode(CBitStreamBuffer*); void PERA_Decode
(CBitStreamBuffer*);
    long value;
};
template <long lowerBound=MIN,long upperBound=MAX,bool
        ExtMarker=false>
struct CINTEGER:public TCINTEGER<UNIVERSAL,2,lower-
                Bound,upperBound,ExtMarker>{ };
```

如上例所示，抽象形式语法对象类的命名有如下规则(这种规则不是本质必须的，只是为了增强程序代码的可读性而遵守的一个不成文规定)：在抽象形式语法所表示的数据类型名前加一大写字母 ' C '，表示这是对象类数据类型；或在抽象形式语法所表示的数据类型名前加一大写字母 ' T '，表示这是具有 Tag 属性的对象类数据类型。CTAGS 是抽象形式语法系统所定义的一组枚举值，在 C++中其实现形式如下：

```
enum CTAGS{NULL_TAG=-1,UNIVERSAL=0X00,
           APPLICATION=0X01,
           CONTEXT_SPECIFIC=0X10,PRIVATE=0X11 };
```

参数 TagNo 是抽象形式语法系统针对其自身所定义的不同基本数据类型所给的一系列内定值，而对具有 Tag 属性的数据类型，该值可由应用协议设计者确定。参数 lowerBound，upperBound 该整型数据类型定义域的上下限值。lowerBound=MIN，upperBound=MAX 表示整型数据类型定义域上下无界。ExtMarker=false 表示整型数据类型的取值可以超出定义域所规定的范围。使用上例所用的类似模板参数，支持抽象形式语法的所有编码规则均可用模板对象类的方法函数进行形式统一的封装。

除了简单基本数据对象类型的编解码方法函数的实现必须严格按照相应的协议规则执行外，复合数据对象类型的编解码方法函数的实现只是一个非常简单的机械重复性工作。只要保证基本数据对象类型的编解码方法函数是正确的，复合数据对象类型的编解码方法函数的出错的机率就不大，即使出了错，也容易发现和纠正。为了表达的简洁性，文章只讨论 ITU-T X691 PER 对齐方式编码规则，但其构思与解决问题的模式适合于任何其它的编解码规则。

2 复合数据类型的对象封装

虽然简单数据类型的编解码方法函数的封装复杂，但其对象封装却较简单。与此相反，复合数据对象的编解码规则的方法函数封装简单，而其对象封装却相对较复杂。下面主要讨论抽象形式语法的复合数据类型封装。

2.1 枚举数据类型的对象封装

首先用宏定义语句将 ENUMERATED 定义为 C++语言的的枚举类关键词 enum，并用 ENUMERATED 来定义一个用户自定义枚举类型。第二步，定义一个模板类 TCENUMERATED 对枚举类型的编解码规则进行封装：

```
#define ENUMERATED enum
template<CTAGS Tag,int TagNo,class Type,Type startIndex,Type
        lastIndex,bool ExtMarker=false> struct TCENUMER-
        ATED:public CBasicType
{ int PERA_Encode(CBitStreamBuffer*);
  int PERA_Decode(CBitStreamBuffer*);
 Type value;
};
template<class Type,Type startIndex,Type lastIndex,bool
        ExtMarker=false>
struct CENUMERATED:public TCENUMERATED < UNIVER-
                SAL,10,Type,startIndex,lastIndex,ExtMarker >{};
```

下面的例子是一个 ENUMERATED 标注的对象类实现：

```
ENUMERATED Sex{ male=1,female,unknow };  //抽象形式语法
                                          标注
typedef CENUMERATED < Sex,male,unknow > CSex; //建立在
                上述枚举定义上的抽象形式语法标注的实现
```

通过该例可以看出，经过类模板封装后，抽象形式语法的标注形式与其代码实现非常相似，这既保持了代码的可读性，又增强了代码的可重用性，使枚举类型不必再写各自的编解码

函数。

SequenceOfType 和 SetOfType 的对象封装与 TCENUMERATED 对象类的定义非常相似,为了节省篇幅,在此不予讨论。

2.2 复合数据类型——结构化数据类型与集合数据类型的对象封装

结构化数据类型(SequenceType)与集合数据类型(SetType)是抽象形式语法构造新数据类型的主要方法,结构化数据类型与集合数据类型的编解码规则完全相同,因此在此只讨论与结构化数据类型有关的问题。结构化数据类型在编解码过程中有一个头部信息,它记录着该结构是否具有可扩展性[5],以及该结构中哪些成员元素有 optional 或 default 属性。因此对象类 TCSEQUENCE 以及它的一些方法函数被用来封装这些编解码规则。

```
Template < CTAGS Tag,int TagNo,bool ExtMarker=false >
struct TCSEQUENCE : public CBasicType
{//ellipse stands for the present of elements with attribute of
    optional or default in bool type
void PER_ENCInit(CBitStreamBuffer* pBitStreamBuffer,int numO-
                    fOptions=0,...);
//ellipse stands for the present of elements with attribute of op-
  tional or default in bool* type
void PER_DECInit(CBitStreamBuffer* pBitStreamBuffer,int numO-
                    fOptions=0,...);
//the following methods are used when extension additions are
  present in the encoding rules
//ellipse stands for the present of extension additional elements
  in bool type
void PER_ENCExtAddInit(CBitStreamBuffer* pBitStreamBuffer,int
                          numOfExtAddition=0,...);
long* PER_DECExtAddInit(CBitStreamBuffer* pBitStreamBuffer);
//SequenceType with extension marker must call the following
  method in its decoding method
//even though it does not have extension additions within.
void PER_DECUnKnowExtAdd(CBitStreamBuffer* pBitStream-
                          Buffer,long*pNumberOfAdd=0);
bool extAddtionPresent; //indicating whether values of extension
                          additions are actually
                        //present when encoding
};
```

所有的结构化数据对象类都必须从 TCSEQUENCE 继承。为了使结构化数据对象类的定义与抽象形式语法中结构化数据类型的定义相似,有如下的宏定义:

```
#define SEQUENCE(ClassName) struct ClassName:public TCSE-
                          QUENCE<UNIVERSAL,16,false>
#define SEQUENCEEx(ClassName)struct ClassName:public TCSE-
                          QUENCE<UNIVERSAL,16,true>
#define TSEQUENCE(Tag,TagNo,ClassName) \
    struct ClassName:public TCSEQUENCE<Tag,TagNo,false>
#define TSEQUENCEEx(Tag,TagNo,ClassName) \
    struct ClassName:public TCSEQUENCE<Tag,TagNo,true>
```

XXXEx 表示结构化数据类型的结构具有可扩展性。每一个与结构化数据类型相对应的结构化数据对象都有自己的编解码方法函数。下面是一个结构化数据对象的例子:

```
SEQUENCEEx (CChildInformation)
{ void PERA_Encode(CBitStreamBuffer* pBitStreamBuffer)
void PERA_Decode(CBitStreamBuffer* pBitStreamBuffer);
CName name;
CDate dateOfBirth; //OPTIONAL
CEmployNumber number;
//...
CENUMERATED < CSex,male,unknow > sex;
};
```

从该结构中可以看出,除增加了两个编解码函数外,该代码与抽象形式语法标注形式基本相似。

结构化数据对象编码方法函数 PERA_Encode(CBitStreamBuffer*)的实现规则:

(1)先调用结构化数据对象的基类方法 PER_ENCInit(…),

(2)接着按结构化数据对象的成员变量在结构中出现的顺序调用成员变量的编码方法函数。如果某成员变量在该结构化数据对象的定义中有属性 default 或 optional,则其按如下形式调用:

```
if (component.present) component.PERA_Encode(pBitStreamBuffer);
```

(3)如果该结构化数据对象中不存在扩展成员变量(additional component),则编码过程完成,

(4)否则,将要附加如下一些方法函数的调用

首先调用 PER_ENCExtAddInit(...);

接着按扩展成员变量在该结构化数据对象中出现的顺序,用如下的形式调用其编码方法函数:

```
if (additionComponent. present)  additionComponent. PERA_Encode
(pBitStreamBuffer);
```

下面这个例子是结构化数据对象 CChildInformation 的编码方法函数的实现:

```
void CChildInformation ::PERA_Encode (CBitStreamBuffer* pBit-
                                        StreamBuffer)
 { PER_ENCInit( pBitStreamBuffer,1,dateOfBirth.present );
   name. PERA_Encode(pBitStreamBuffer);
   if(dateOfBirth. present) dateOfBirth.PERA_Encode(pBitStream-
                                                    Buffer);
   number. PERA_Encode(pBitStreamBuffer);
   //following procedure process extension components
  PER_ENCExtAddInit(pBitStreamBuffer,1,sex.present);
  if (sex. present)PERA_EnOpenType( pBitStreamBuffer,&sex );
  }
```

结构化数据对象解码方法函数 PERA_ Decode (CBitStreamBuffer*)的实现与编码方法函数的实现非常相似,为了节省篇幅,在此不予讨论。

2.3 复合数据类型——选择数据类型的对象封装

选择数据类型是抽象形式语法用来定义协义数据单元最有力的工具,它与结构化数据类型的对象封装基本类似,首先定义一个基类来封装选择数据类型的编解码头部信息:

```
template<CTAGS Tag,int TagNo,ULONG lastRootIndex,bool
          ExtMarker=false>
struct CCHOICE : public CBasicType
{template<class Type>void PER_ENCInit(CBitStreamBuffer* pBit-
                                        StreamBuffer,Type* pValue);
   void PER_DECInit(CBitStreamBuffer* pBitStreamBuffer);
unsigned int itemSelected; //for index of chosen alternatives
 };
```

所有的选择数据对象类都必须从 CCHOICE 继承。为了使选择数据对象类的定义与抽象形式语法中选择数据类型的定义相似,有如下的宏定义:

```
#define CHOICE(ClassName,lastRootIndex)                    \
                struct ClassName:public CCHOICE<UNIVERSAL,
                                    -1,lastRootIndex,false >{
#define CHOICEEx(ClassName,lastRootIndex)                  \
                struct ClassName:public CCHOICE<UNIVERSAL,
                                    -1,lastRootIndex,true >{
#define CHOICE_BEGIN                                       union{
#define CHOICE_END                                         };};
#define CHOICE_END_WITH_INSTANCE(instance)      };}instance;
```

XXXEx 表示选择数据类型的结构具有可扩展性。每一个与选择数据类型相对应的选择数据对象都有自己的编解码方法函数。下面是一个选择数据对象的例子:

```
CHOICE( CTest,lastRootIndex)
        void PERA_Decode(CBitStreamBuffer* pBitStreamBuffer)
        void PERA_Encode(CBitStreamBuffer* pBitStreamBuffer)
CHOICE_BEGIN
        CVisibleString<128> no_0_Component;
        CVisibleString<128> no_1_Component;
      ...
        CLastComponent lastRootComponent;
        //....
        CVisibleString<128> firstAdditionComponent;
CHOICE_END
```

从该结构中可以看出,除增加了两个编解码函数外,该代码与抽象形式语法标注形式基本相似。选择数据对象的编码方法函数的实现非常简单,任何选择数据对象的编码方法函数的实现与下例相同:

```
void CTest :: PERA_Encode(CBitStreamBuffer* pBitStreamBuffer)
{ switch(itemSelected)
{case 0:PER_ENCInit(pBitStreamBuffer,&no_0_Component ); break;
case 1:PER_ENCInit(pBitStreamBuffer,&no_1_Component ); break;
                                                    //so on
    case lastRootIndex : PER_ENCInit(pBitStreamBuffer,&last-
                         RootComponent); break;    //....
    case lastRootIndex + 1 : PER_ENCInit(pBitStreamBuffer,&
                         firstAdditionComponent); break;
  }
}
```

选择数据对象的解码方法函数的实现也非常简单,任何选择数据对象的解码方法函数的实现与下例相同:

```
void CTest :: PERA_Decode(CBitStreamBuffer* pBitStreamBuffer)
{ PER_DECInit(pBitStreamBuffer);
  switch(itemSelected)
{case 0:no_0_Component.PERA_Decode (pBitStreamBuffer); break;
 case 1:no_1_Component.PERA_Decode(pBitStreamBuffer ); break;
          //so on
 case lastRootIndex:lastRootComponent.PERA_Decode(pBitStream
                    Buffer); break;          //....
 case lastRootIndex+1: PERA_DeOpenType(pBitStreamBuffer,&
                    firstAdditionComponent); break;
   default: //for extended additional unknown component
    PERA_DeOpenType( pBitStreamBuffer,(CBOOLEAN*)NULL );
  }
}
```

函数: template<class T> void PERA_EnOpenType(CBitStreamBuffer * pBitStreamBuffer,T* pvalue)使用 ITU-T X.691 的 Opentype 编码规则对抽象对象 T* pvalue 进行编码。

3 总结

复合数据类型是抽象形式语法定义新数据类型的工具,从第二部分的讨论可以看出,如果采用模板类的思想对抽象形式语法的复合数据类型进行对象类封装,那么,这些对象类的编解码方法函数的实现不仅规则而且简单,并且保持了原标抽象形式语法标注的简洁可读性。所有这些有效地简化了网络协议的编解实现,并且提高了协议代码化过程中的正确性。

(收稿日期:1999 年 9 月)

参考文献

1.1.Scope (ASN.1) X.680 Telecommunication Standardization Sector of ITU

2.14.Definition of types and Values (ASN.1) X.680 Telecommunication Standardization Sector of ITU

3.45 Subtype elements (ASN.1) X.680 Telecommunication Standardization Sector of ITU

4.help for template keyword in Microsoft VC 4.2

5.18.Encoding the sequence type (PER) X.691 Telecommunication Standardization Sector of ITU

(上接 51 页)

动规则则为一由多个自变量因子决定的函数。

此方法的一个优点是初始化表面的基本形状在迭代中保持,这样最终表面的拓扑结构由初始表面预先决定,初始表面类型的确定则需要用户大量的背景知识。

(收稿日期:1999 年 9 月)

参考文献

1.J V Miller,D E Breen,W E Lorensen.Geometrically Deformed Models:A method for extracting closed geometric models from volume data.Computer Graphics,1991

2.Hultquist J.Constructing Stream Surface in Steady 3D Vector Fields.Visualization'92,1992

3.石教英,蔡文立.科学计算可视化算法与系统.科学出版社,1996