[**ASN.1编码规则详解**](http://blog.csdn.net/HansenKevin/archive/2006/07/30/1001383.aspx)

**1 简介**

**🕮 注释：**

本章的内容主要翻译自《ASN.1 Communication between Heterogeneous Systems》和《ASN.1 Complete》。

**1.1 ASN.1简介**

ASN.1 (Abstract Syntax Notation dot one)，抽象记法1。数字1被ISO加在ASN的后边，是为了保持ASN的开放性，可以让以后功能更加强大的ASN被命名为ASN.2等，但至今也没有出现。

ASN.1是定义抽象数据类型规格形式的标准。是用于描述数据的表示、编码、传输、解码的灵活的记法。它提供了一套正式、无歧义和精确的规则，以描述独立于特定计算机硬件的对象结构。

ASN.1是通信协议中描述数据传输的正式标记（notation），它与语言实现和物理表示无关，与应用的复杂度无关。ASN.1特别适合表示现代通信应用中那些复杂的、变化的及可扩展的数据结构。

ASN.1发送任何形式（音频、视频、数据等等）的信息都必须用数字传送。ASN.1只能包含信息的结构方面（没有已经定义的或考虑到的处理数据值的操作）。它不是一个编程语言。

ASN.1 本身只定义了表示信息的抽象句法，但是没有限定其编码的方法。各种ASN.1 编码规则提供了由ASN.1描述其抽象句法的数据的值的传送语法（具体表达）。标准的ASN.1 编码规则有基本编码规则（BER，Basic Encoding Rules）、规范编码规则（CER，Canonical Encoding Rules）、唯一编码规则（DER，Distinguished Encoding Rules）、压缩编码规则（PER，Packed Encoding Rules）和XML编码规则（XER，XML Encoding Rules）。

ASN.1成功的一个主要理由是它采用了一些如BER(Basic Encoding Rules)或新制定的PER(Packed Encoding Rules)这样的标准化的编码规则，它对受带宽限制的应用程序很有用。这些编码规则描述了如何将定义在ASN.1中的值译成适合传输的电码（例如:他们可以被转化为可传输的字节，或反向转化），其与机器、编程语言或者在应用程序中的表示无关。ASN.1的编码方式比很多竞争者更先进，能够快速并可靠的传输可扩展信息－－这是无线带宽的一个优势。由于从1982年以后ASN.1已经成为一个国际标准，所以它的编码规则是成熟的并且它有长期的可靠性和互用性的跟踪记录。一个的ASN.1源文件可以非常容易地（由预处理器）映射为C或C++或Java数据结构，可用于通过应用程序代码，和支持的运行时库提供的编码和解码的申述或者一个XML或限值的格式，或非常紧凑的压缩编码格式。

几乎所有操作系统上的工具都支持ASN.1，ASN.1支持如java，c和c++这样流行的编程语言，和包括COBOL这样的较老的编程语言。ASN.1的广泛性的一个例子是，有很多工具已经支持超过150种不同的计算机平台。有大量经过检验的ASN.1工具已经被长期使用。

ASN.1是一种 ISO/ITU-T 标准，描述了一种对数据进行表示、编码、传输和解码的数据格式。它提供了一整套正规的格式用于描述对象的结构，而不管语言上如何执行及这些数据的具体指代，也不用去管到底是什么样的应用程序 — 不论是非常复杂的，还是非常简单的。

在任何需要以数字方式发送信息的地方，ASN.1 都可以发送各种形式的信息（声频、视频、数据等等）。ASN.1 和特定的 ASN.1 编码规则推进了结构化数据的传输，尤其是网络中应用程序之间的结构化数据传输，它以一种独立于计算机架构和语言的方式来描述数据结构。

ISO 协议套中的应用层协议使用了 ASN.1 来描述它们所传输的 PDU，这些协议包括：用于传输电子邮件的 X.400、用于目录服务的 X.500、用于 VoIP 的 H.323 和 SNMP。它的应用还可以扩展到通用移动通信系统（UMTS）中的接入和非接入层。

简洁的二进制编码规则（BER、CER、DER、PER，但不包括 XER）可当作更现代 XML 的替代。然而，ASN.1 支持对数据的语义进行描述，所以它是比 XML 更为高级的语言。

ASN.1 的描述可以容易地映被射成 C 或 C++ 或 Java 的数据结构，并可以被应用程序代码使用，并得到运行时程序库的支持，进而能够对编码和解码 XML 或 TLV 格式的，或一种非常紧凑的压缩编码格式的描述。

同时，ASN.1是一种用于描述结构化客体的结构和内容的语言。

抽象语法定义：

ASN.1是描述在网络上传输信息格式的标准方法。它有两部分：一部分描述信息内数据，数据类型及序列格式；另一部分描述如何将各部分组成消息。它原来是作为X.409的一部分而开发的，后来才自己独立成为一个标准。ASN.1在OSI的ISO 8824/ITU X.208（说明语法）和ISO 8825/ITU X.209（说明基本编码规则）规范。

例如：

Report ::= SEQUENCE {

author OCTET STRING,

title OCTET STRING,

body OCTET STRING,

biblio Bibliography

}

在这个例子中，"Report"是由名字类型的信息组成的，而SEQUENCE表示消息是许多数据单元构成的，前三个数据单元的类型是OCTET STRING，而最后一个数据类型则下面的ASN.1语法表示它的意义：

Bibliography ::= SEQUENCE {

author OCTET STRING

title OCTET STRING

publisher OCTET STRING

year OCTET STRING

}

ASN.1 提供了一些基本的预定义数据类型：

UNIVERSAL 0 保留给编码规则使用

UNIVERSAL 1 布尔类型

UNIVERSAL 2 整型

UNIVERSAL 3 二进制字符串类型

UNIVERSAL 4 八进制字符串类型

UNIVERSAL 5 空类型

UNIVERSAL 6 对象标识符类型

UNIVERSAL 7 对象描述符类型

UNIVERSAL 8 外部类型和类型实例

UNIVERSAL 9 实数类型

UNIVERSAL 10 枚举类型

UNIVERSAL 11 嵌入的 pdv 类型

UNIVERSAL 12 UTF8 字符串类型

UNIVERSAL 13 相关对象标识符 类型

UNIVERSAL 14-15 保留给本建议的以后版本和国际标准使用

UNIVERSAL 16 序列和类型序列

UNIVERSAL 17 集合和类型的集合

UNIVERSAL 18-22, 25-30 字符串 类型

UNIVERSAL 23-24 时间 类型

UNIVERSAL 31-... 保留给本建议以外的类型和国际标准使用

ASN.1 还能够定义如下的数据结构类型：

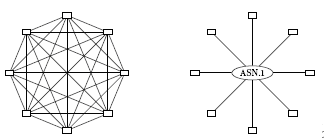
结构 ( SEQUENCE ),

列表 ( SEQUENCE OF ),

类型选择 ( CHOICE ),

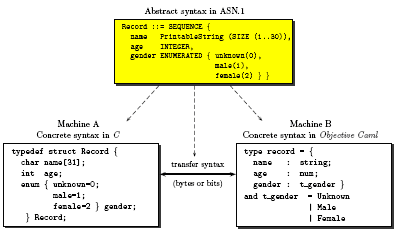
等等

不同体系的系统，不同编程语言之间，都存在通讯障碍。如Figure 1-1 a）所示极端情况下，为了保证网络中n个异体的节点能互通，我们需要为每个节点编写(n-1)个编解码程序，即需要总数为n\*(n-1)个编解码程序。在b）所示极端情况下，为了保证互通，需要为每个节点编写1个编码和1个解码程序，即总数为2n个编解码程序。



**Figure 1-1** 两种类型的通讯方式

对于给定的记法描述，ASN.1编译器能执行并产生如Figure 1-2图中虚线部分，这样我们以有限的代价就能完成相当多数量系统之间的互连。



**Figure 1-2** 语法三元组：实际语法、抽象语法和传输语法

(1)实际语法（Concrete Syntax）

指诸如C、Objective Caml等这样实际编程语言；

(2)抽象语法（Abstract Syntax）

指ASN.1，是协议采用ASN.1规范描述的描述文本。描绘了与任何表示数据的编码技术无关的通用数据结构。抽象语法使得人们能够定义数据类型，并指明这些类型的值。抽象语法只描述数据的结构形式，与具体的编码格式无关，同时也不涉及这些数据结构在计算机内如何存放。

(3)传输语法（Transfer Syntax）

指表示层交换数据的表示方法，是实际通讯系统间的码流。当数据在两个表示层实体之间传输时，这些数据的实际比特模式表示方法就是传送语法。

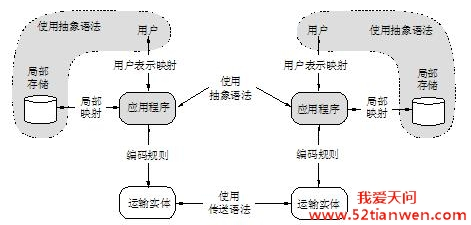
(4)编码

指用来表示数据值的完整的八位组序列。

(5)编码规则

从一个语法到另一个语法的映射规约。具体地说，编码规则从算法上定义了任何一组由抽象语法定义的数据值在传送语法中的表示。

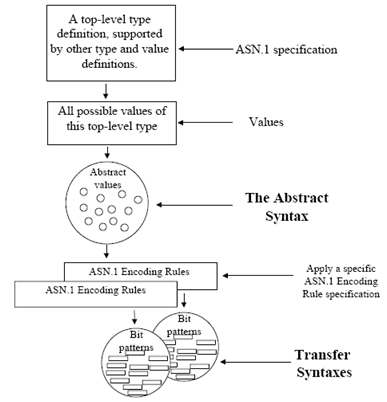
下图以两个端系统通过网络交换数据为例来说明上述的一些概念。



运输实体所看到的数据是应用实体交下来的、根据一定的编码规则进行编码的二进制代码。应用实体看到的则是一个用户观点的数据，通常是结构化的信息，如文本文档或可显示的图

象信息。用户主要关心的是数据的语义。因此应用实体必须提供数据的表示方法，使得这些数据能够转换为二进制值。也就是说，应用实体必须考虑到数据的语法。

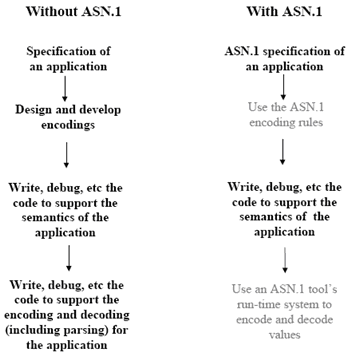
从抽象语法到传输语法，由ASN.1编译器按照编解码规则实现。



**Figure 1-3** 抽象语法与传输语法

其实早期的一些标准，如ASCII，它们既定义了抽象语法（比如字母A），又定义了传输语法（0x41）。ASN.1分离了这两种概念，以便可以选择一种适合要求的编解码方法。系统可以选择编码方法以使信息传送时效率很高，或者具有很高的可靠性等等。另一方面，定义好的编码规则也会很大地节约应用协议开发人员的时间，特别是当牵涉到的数据结构很复杂的时候。当使用ASN.1的项目较多时，这种节约更加明显，因为编解码程序可以只开发一次但很多应用程序都可以用它。

我们可以看出，能将通信编解码设计与开发工作转嫁给ASN.1编译器完成。从而不必手工编写编解码器。一方面大量减少了缺陷引入，另一方面更是大大加快了系统开发速度（不用编写编解码器，也不用详细调试，维护代价也很小）。



**Figure 1-4** 转嫁编解码器工作给ASN.1编译器

讨论ASN.1就不得不先提到标准组织ISO和ITU。

**1.2 ISO简介**

国际标准组织（ISO – International Organization for Standardization）于1946年在美国成立，其负责制定众多领域的国际标准；但除电气、电子和电子工艺领域外，这些主要是IEC（International Electrotechnical Commission）负责。

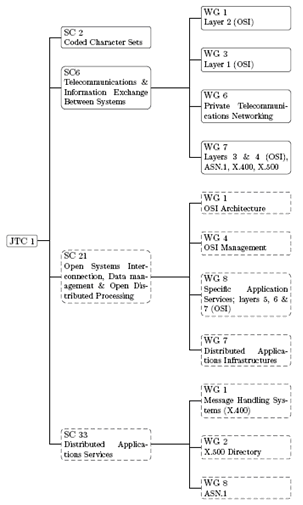
ISO汇集了一百多个代表其自己国家的标准委员会（称之为NB – National Body），ANSI （American National Standard Institute）代表美国，AFNOR （Association Francaise de NORmalisation）代表法国，BSI（British Standard Institute）代表英联邦，等等。其它一些组织可以参加讨论，提出议案，但不能参加投票。



**Figure 1-5** ISO组织结构

如Figure 1-5所示，ISO共分为172个技术委员会Technical Committee （TC）负责相应标准化领域。 所有议题都在子委员会SubCommittees中共享，Subcommittee又分为工作组Working Groups（WG）。

到1987年，著名的OSI标准就是TC97的成果，称之为“Telecommunications and Information Exchange Between Systems”。在1987年，ISO和IEC两个标准化组织一致认为都应当关注信息技术Information Technology，因此成立了一个联合技术委员会称为JTC1。JTC1的秘书处由ANSI负责。



**Figure 1-6** JTC1组织结构示意图

在ASN.1标准刚刚受到关注时，基于历史原因，ASN.1主要来自于CCITT的成果，有很多提案来自CCITT。在80年代，ISO在1990年左右成为主角前，两个委员会的确有过合作。

**1.3 ITU简介**

国际电信联盟ITU（International Telecommunication Union）总部位于日内瓦，于1947年成为联合国的专设机构，其前身为成立于1865年的国际电报联盟（International Telegraph Union）。ITU汇集了188个代表其国家的公共电信运营商和大约450个来自私有部门称之为RPOA（Registered Private Operating Authorities）的成员（如AT&T， Bell电话等）。来自其它组织的运营商也被接收，但不具有投票权利。ITU有5个常设组织，其中一个为CCITT（Consultative Committee on International Telephony and Telegraphy）负责电信网络，如有线传输语音，数据和电视。在1992年ITU重组后，CCITT成为了ITU-T（ITU-Telecommunication Standardization Sector）。

ITU-T发布的建议都是以一个字母开头（每个领域一个字母编号）和一个数字作为参考号。我们通常关注的有：

           F: Non-Telephone Telecommunications Services;

           H: Audiovisual and Multimedia Systems;

           Q: Switching and Signalling;

           T: Terminals for Telematic Services;

           V: Data Communications Over the Telephone Network;

           X: Data Networks and Open System Communications;

           Z: Programming Languages.

到1992年，所有CCITT已经通过其委员会多次论证后的建议每四年发布一次（即使某些部分还未完成），其封面用不同的颜色（即黄皮书、红皮书、兰皮书和白皮书）：黄色代表1976-1980，红色代表1980-1984，蓝色代表1984-1988，白色代表1988-1992。

ITU-T现在是当某个标准稳定后单独发布。此外，自1994年开始，为了防止出现不一致，ISO/IEC和ITU-T都是联合发布标准文本（各用不同的参考号）。 不一致的情况在ASN.1中出现过。

ITU-T分为研究组SG（Study Group），其中：

           SG VII，“Data Networks and Open System Communications”，课题广泛，包括互连（比如ASN.1），X.25网络标准，X.400 Email姓名地址录标准，网络安全和网络管理等。

           SG VIII，“Terminals for Telematic Services”负责Teletex，Videotex和字符集等。

每个SG的工作又被划分为**问题**Question：其中一些相当有名，比如涉及OSI模型的Q24/7；另外一些则是对以发布标准的更为精确的更新。每个问题最终结束于一个新标准或者一个现有标准的补充。

**1.4 ASN.1历史**

开始于1982年夏天，许多开发应用层标准的人意识到他们面临同一个问题：数据结构变得越来越复杂，难以用预定的规程来编解码bit或者Byte（在1978年，OSI中术语Open是至关重要的）。如同编译器取代了汇编器，通常认为编码器应当从描述Specification自动产生，这样使得描述也等效为一种计算机程序。

**1.4.1 诞生**

James White和Douglas Steedman分别提出一个基本的记法和算法，这套记法能定义Email消息控制系统协议MHS（Message Handling System）的编码比特格式。这种记法和编码规则是与机器无关的，能表述复杂的数据结构。

James White当时是CCITT X.400 （MHS）建议系列的起草人，并且为施乐公司（Xerox Corporation）设计了Courier记法，这种记法能翻译XNS（Xerox Network Services）的RPC（Remote Procedure Call）传送的数据。Courier作为第一种外部的数据记法而闻名。同时它对后来的XDR（来自Sun Microsystems Inc.）和NDR（来自Apollo Computer Inc.）都产生了积极的影响。

1984年，CCITT将这种记法标准化，参考号为X.409（红皮书）。X.409建议成为：Message Handing Systems: Presentation Transfer Syntax And Notation。它的目标和范围为：“the presentation transfer syntax used by application layer protocols in message handling systems and by the document interchange protocol for the telematic services. In the architecture of open systems interconnection (OSI), a presentation transfer syntax is used to represent information exchanged between application entities”

X.409记法和MHS系统是完全独立的，部分原因是Email协议处理的对象是十分复杂的，没有大小限制，多种字符串类型，链接的结构，为数众多的选项等。结果，许多OSI应用标准化工作组发现这种记法对他们也有用。

**1.4.2 1989和1990版本**

如前所述，1987年ISO和IEC成立了JTC 1。

1989年CCITT发布了两个文档X.208（ASN.1）和 X.209（BER）来替代X.409建议。其中很多新特性是由JTC 1引入的：subtypes, floats (REAL type), pointers (ANY DEFINED BY type) and the default tagging modes (IMPLICIT TAGS and EXPLICIT TAGS)。他们这套X.200系列称为“General OSI Infrastructure”，表示ASN.1成为应用层一种独立的描述语言。

1990年ISO发布了新的ISO 8824和ISO8825标准，和X.208相比，有三处地方相悖。具体问题请参考1.6.1 CCITT 1989版本和ISO 1990版本的冲突。

自1998年开始，ISO和ITU-T联合发布他们的标准文本，避免再次出现标准间的冲突。

**1.4.3 1994版本**

新的版本（实际是1994年完成草案） ASN.1 :1994于1995年被ISO通过，成为正式标准。为了阐述数目众多的条件和介绍新概念，该标准分为四个部分：

1)         ITU-T Rec. X.680 (1994) | ISO/IEC 8824-1:199518: Specification of Basic Notation,

2)         ITU-T Rec. X.681 (1994) | ISO/IEC 8824-2:1995: Information Object Specification,

3)         ITU-T Rec. X.682 (1994) | ISO/IEC 8824-3:1995: Constraint Specification,

4)         ITU-T Rec. X.683 (1994) | ISO/IEC 8824-4:1995: Parameterization of ASN.1 Specifications,

几个月之后，发布了两个补充和一个技术勘误表：

           ITU-T Rec. X.680/Amd.1 (1995) | ISO/IEC 8824-1:1995/Amd.1: Rules of Extensibility,

           ITU-T Rec. X.680/Corr.1 (1995) | ISO/IEC 8824-1:1995/Corr.1: Technical Corrigendum 1,

           ITU-T Rec. X.681/Amd.1 (1995) | ISO/IEC 8824-2:1995/Amd.1: Rules of Extensibility.

编码规则部分被分成了两个部分（和一个技术勘误表）：

           ITU-T Rec. X.690 (1994) | ISO/IEC 8825-1:1995: ASN.1 Encoding Rules: Specification of Basic Encoding Rules, Canonical Encoding Rules, and Distinguished Encoding Rules,

           ITU-T Rec. X.690/Corr.1 (1995) | ISO/IEC 8825-1:1995: Technical Corrigendum 1,

           ITU-T Rec. X.691 (1995) | ISO/IEC 8825-2:1995: ASN.1 Encoding Rules: Specification of Packed Encoding Rules

这些标准给出了官方正式编码规则，这对诸如传输加密数据情况来说是必需的。

1994版本和以前版本相比的差异和新特性，请参考1.6.2 **错误!未找到引用源。**。

**1.4.4 1997版本**

语义模型分为三个补充文件：

           ITU-T Rec. X.680 (1997)/Amd.2 (1999) | ISO/IEC 8824-1:1998/Amd.2

           ITU-T Rec. X.681 (1997)/Amd.1 (1999) | ISO/IEC 8824-2:1998/Amd.1

           ITU-T Rec. X.683 (1997)/Amd.1 (1999) | ISO/IEC 8824-4:1998/Amd.1

相关对象标识符新的RELATIVE-OID类型说明在三个补充文件：

           ITU-T Rec. X.680 (1997)/Amd.1 (1999) | ISO/IEC 8824-1:1998/Amd.1

           ITU-T Rec. X.690 (1997)/Amd.1 (1999) | ISO/IEC 8825-1:1998/Amd.1

           ITU-T Rec. X.691 (1997)/Amd.1 (1999) | ISO/IEC 8825-2:1998/Amd.1

新增四个技术勘误表：

           ITU-T Rec. X.680 (1997)/Corr.1 (1999) | ISO/IEC 8824-1:1998/Corr.1

           ITU-T Rec. X.681 (1997)/Corr.1 (1999) | ISO/IEC 8824-2:1998/Corr.1

           { ITU-T Rec. X.690 (1997)/Corr.1 (1999) | ISO/IEC 8825-1:1998/Corr.1

           { ITU-T Rec. X.691 (1997)/Corr.1 (1999) | ISO/IEC 8825-2:1998/Corr.1

1994版本和以前版本相比的差异和新特性，请参考1.6.3 1997版本新特性和1.6.4 **错误!未找到引用源。**。

**1.4.5 最新版本**

目前最新版本的协议为：

           **ITU-T X.680** (ISO/IEC 8824-1) - Abstract Syntax Notation One (ASN.1): Specification of Basic Notation

           **ITU-T X.681** (ISO/IEC 8824-2) - Abstract Syntax Notation One (ASN.1): Information Object Specification

           **ITU-T X.682** (ISO/IEC 8824-3) - Abstract Syntax Notation One (ASN.1): Constraint Specification

           **ITU-T X.683** (ISO/IEC 8824-4) - Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 Specifications

           **ITU-T X.690** (ISO/IEC 8825-1) - ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)

           **ITU-T X.691** (ISO/IEC 8825-2) - ASN.1 Encoding Rules: Specification of Packed Encoding Rules (PER)

           **ITU-T X.692** (ISO/IEC 8825-3) - ASN.1 Encoding Rules: Specification of Encoding Control Notation (ECN)

           **ITU-T X.693** (ISO/IEC 8825-4) - ASN.1 Encoding Rules: Specification of XML Encoding Rules (XER).

           **ITU-T X.694** (ISO/IEC 8825-5) - ASN.1 Encoding Rules: Mapping W3C XML schema definitions into ASN.1.

**1.5 ASN.1应用情况**

           Accredited Standards Committee X9 - Financial Services

           Manufacturing Message Specification (MMS)

           Molecular Biology Standards

           Request For Comments (RFC)

           RSA Public-Key Cryptography Standards

           Standard Security Label for Information Transfer

           Secure Electronic Transaction (SET) ASN.1

           T.120 Series Standards

           Telecommunications Management Network (TMN)

           Unicode Worldwide Character Standard

           X.400 Standards - Message Handling Systems

           X.500 Standards - The Directory

           H.323

           H.248

           3G on UMTS

等等。

现在一些协议，标准组织在给出协议文本时往往也同时给出了ASN.1的描述。

**2 基础知识**

**🕮 注释：**

本章的内容主要翻译自《ASN.1 Communication between Heterogeneous Systems》。

**2.1 相关背景知识**

**2.1.1 OSI参考模型**

**Figure 2-1** OSI 七层参考模型

虽然ASN.1和OSI密切相关，但实际上它也应用在很多非OSI模型的情况下。

**1)表示层 Presentation Layer**

两个系统在传输数据前需要协商共用的编码方式。**表示层**负责在两个应用系统间进行编码方式协商和实际信息的编码。

为了达到这个目的，表示层需使用以下概念：

(1)抽象语法：定义了数据的常用结构（包括不同的数据类型），并且建立了和应用层对话所依赖的框架。

(2)实际语法：本地的，并且定义本地系统的数据表示方法。

(3)传输语法：定义两个系统间的表示层间交换数据的表示方法。

(4)编码规则：提供从本地实际语法到传输语法和其相反操作的方法。(第1章中说过：从抽象语法到传输语法，由ASN.1编译器按照编解码规则实现。)

应用这些记法，表示层能够提供应用层这些服务：

(1)传输语法的协商（在会话开始前挑选一种传输语法的方法，改变传输语法的方法）；

(2)确定传输语法的集合（抽象语法的多种表示方式）；

(3)用实际语法（内部表示方式）的编码/解码规则来翻译为传输语法（外部表示方式）或者做相反操作；

(4)把一个协商后的传输语法和应用内部采用的抽象语法进行关联；

(5)访问会话层服务。

 **Figure 2-2** 表示层上下文协商

在 Figure **2-2** 表示层上下文协商表示层上下文协商中，详细描述了这样一个过程：

(1)应用A发送一个P-CONNECT.request原语给自己的表示层，并且声明操作这个传输的相关抽象语法名字为(AS1，AS2)。每个抽象语法的名字实际是一串成为对象标识符（Object Identifier，在ASN.1中是OBJECT IDENTIFIER类型的值）的数字，它能唯一的标识抽象语法。的确，我们应当注意到这是一个开放式的架构，能适应各种机器，各种抽象语法，各种传输语法。

(2)表示层为每个抽象语法关联传输语法，并且为会话层编码“表示(层)协议数据值”PPDV（Presentation Protocol Data Value），这些数据会被送往对方系统的表示层。PPDV中包含着可用的抽象语法。

(3)表示层B收到这个PPDV数据后，向自己的应用层回送P-CONNECT.indication原语，指示应用A的抽象语法可用。

(4)应用B以P-CONNECT.response原语应答，指示在这个传输中可用的抽象语法的名字（这里只指示AS2）。

(5)会话层B接收到原语后，发送PPDV指明传输语法，这个语法应当是已经协商中的一个（如上图中是T2、T3中的T2）。

(6)最后，表示层A收到PPDV后，检查对方推荐的传输语法，如果接受它则发送P-CONNECT.confirm原语给应用A。

应当注意，表示层没有参与到应用间可用抽象语法的确定过程中。通常，会用多个抽象语法/传输语法的组合关系。一个抽象语法可以用多个传输语法来表示；一个传输语法也可以用来表示多个抽象语法。抽象语法/传输语法组合的协商结果被称为“表示上下文”（Presentation Context）。上下文用整数来标识，为了防止重新分配时出现覆盖，一般让一个实体采用偶数而另一个实体采用奇数。其它上下文可以在通讯过程中动态协商。

在初始协商的最后，系统在它的配置中有了一套“表示上下文”，在任何时候，系统都能从中选出合适的上下文进行交换操作。

来自应用层的数据会被根据相关的“表示上下文”进行编码。当只有一个上下文被指定时，他们将被直接编码（简单编码 Simple Encoding）；否则应用数据包括嵌入数据（embedded data）都要在前面加上正确的上下文标识符（完整编码 Complete Encoding）。

**2)应用层 Application Layer**

采用OSI模型时，我们称一个应用的一个通讯方面为一个应用实体。通讯实体用通讯协议和表示业务来共享信息。

在ASN.1中，各个应用的数据结构作为“应用(层)协议数据值”APDV（Application Protocol Data Value）发送。每当要传输数据时，应用实体都会在给出APDV的同时，告知表示层自己的ASN.1名字。通过参考ASN.1的定义，表示层可以得知数据单元的类型和长度，以及传输时应当采用的编码方法。在连接的另一端，表示层分析收到数据结构中的ASN.1标识符，就能得知第一个数据单元有多少比特，第二个有多少，等等。有了这些信息，表示层按照接收方内部格式对数据做必要的转换。

自从ISO要求所有的应用层和表示层数据交换都要用ASN.1抽象语法描述后，ASN.1在OSI中只作为表示方法用。随着OSI模型的广泛应用，ASN.1主要用在高层中（部分因为在ASN.1出现前，许多低层已经存在了），但这不是一种限制。如果ASN.1能广泛应用在低层中，就能充分影响编码过程，阻止数据在不同层上反复出现。

**2.1.2 边界对齐**

对于同样一条消息，在计算机内存中是以Byte为单位存储的，在链路上则是以bit为单位传送的。对于其中每个信元IE，当从bit流映射到Byte流时，就涉及到边界对齐的问题：

如果一个信元的第一个bit也恰好是Byte流中某Byte的开始bit时，我们称之为开始于边界对齐的；

如果信元的最后一个bit也恰好是Byte流中某Byte的最后一个bit时，则可以称之为结束于边界对齐的。

对于不是结束于边界对齐的情况，一般要进行补位。有两种方式：

(1)对每个信元的结束立即进行补位，保证下一个信元是开始于边界对齐的；

(2)从信元结束的位置开始新的信元，到消息结束时再进行一次补位操作。(用于无线空口中)

通常第一种方式用在对消息大小要求不苛刻的编码过程中和通常的解码过程中；第二种方式用在对消息大小要求苛刻的编码过程中（如无线通讯系统的空中接口消息编码中）。

当把边界对齐问题和大小端问题放在一起时，常常会造成理解上的困难。某些协议中为了明确表示位域跨字节情况下的排列，用MSB、LSB标识最高、最低bit的位置所在。

**2.1.3 大小端(Big Endian vs Little Endian)**

    <1>大端方式(MSDN中说的网络序)：Motorola的PPC系列、IP协议中；(该方式和书写方式一致)

    <2>小端方式(MSDN中说的主机序)：VAX计算机、Intel的x86系列；

字节内部的bit高低次序相同(左高右低)，而字节之间的高低次序相反。

在32位系统中我们分以下这些情况来一一说明：

1)Byte类型(8bits)

在只有一个Byte的情况下，大端方式和小端方式没有分别。如：0x34

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **bit7** | **bit6** | **bit5** | **bit4** | **bit3** | **bit2** | **bit1** | **bit0** |  |
| **Big Endian** | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0x34 |
| **Little Endian** | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0x34 |

2)Short类型(16bits)

大小端方式之间有差别。如：0x1234

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **bit7** | **bit6** | **bit5** | **bit4** | **bit3** | **bit2** | **bit1** | **bit0** |  |
| **Big Endian (0)** | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0x12 |
| **Big Endian (1)** | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0x34 |
| **Little Endian (0)** | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0x34 |
| **Little Endian (1)** | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0x12 |

3)Long类型(32bits)

大小端方式之间有差别。如：0x12345678

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **bit7** | **bit6** | **bit5** | **bit4** | **bit3** | **bit2** | **bit1** | **bit0** |  |
| **Big Endian (0)** | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0x12 |
| **Big Endian (1)** | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0x34 |
| **Big Endian (2)** | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0x56 |
| **Big Endian (3)** | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0x78 |
| **Little Endian (0)** | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0x78 |
| **Little Endian (1)** | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0x56 |
| **Little Endian (2)** | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0x34 |
| **Little Endian (3)** | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0x12 |

4)位域的情况(边界对齐问题+大小端问题)：

大小端方式之间有明显的差别。

在一个字节内，如3-4-1结构下的{4, 15, 0}

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **bit7** | **bit6** | **bit5** | **bit4** | **bit3** | **bit2** | **bit1** | **bit0** |  |
| **Big Endian** | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0x9E |
| **Little Endian** | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0x7C |

跨字节，边界不对齐，如5-4-7结构下的{2, 15, 0}

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **bit7** | **bit6** | **bit5** | **bit4** | **bit3** | **bit2** | **bit1** | **bit0** |  |
| **Big Endian (0)** | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0x17 |
| **Big Endian (1)** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x80 |
| **Little Endian (0)** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0x01 |
| **Little Endian (1)** | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0xE2 |

**2.2 基本语法规则**

到目前为止，ASN.1记法仍然主要是BNF（Backus-Naur Form）形式的。

  1)在ASN.1中，符号的定义没有先后次序：只要能够找到该符号的定义即可，而不必关心在使用它之前是否被定义过。如：

employeeNumber EmployeeNumber ::= 12345

EmployeeNumber ::= [APPLICATION 2] INTEGER

2)所有的标识符、参考、关键字都要以一个字母开头，后接字母(大、小写都可以)、数字或者连字符“-”(但不能以连字符“-”结尾，也不能连续出现两个连字符)，不能出现下划线“\_”。

合法的：

INTEGER

v1515

No-final-dash

MY-CLASS

不合法的：

Final-dashdouble--

dash

under score

1515

3M

3)关键字一般都是全部大写的，除了一些字符串类型，如PrintableString，UTF8String，等，因为这些都是由原类型OCTET STRING衍生出来的。

4)在标识符中，只有类型和模块名字是以大写字母开头的，其它标识符都是以小写字母开头的。

5)字符串有三种形式：

           <1>用引号引用的字符串： “This is a string”

           <2>单引号引用的二进制串后加大写字母B： ‘01101’B

           <3>单引号引用的十六进制串后加大写字母H： ‘0123456789ABCDEF’H

6)带小数点的小数形式不能在ASN.1中直接使用，在ASN.1中实数实际定义为三个整数：尾数、基数和指数。

7)注释以两个连字符“--”开始，结束于行的结尾或者该行中另一个双连字符。

8)如同大多数计算机语言，ASN.1不对空格、制表符、换行符和注释做翻译。但是在定义符号（或者分配符号Assignment）“::=”中不能有分隔符，否则不能正确处理。

**2.3 类型与类型定义**

**2.3.1 ASN.1中的类型**

ASN.1中的一个重要概念就是类型。类型是一个非空的值的集合，可以被编码后传输。相比与高级语言中复杂的数据结构，ASN.1中的类型主要是为了数据的传输(比如为SEQUENCE和SET定义了OPTIONAL条目)。而BIT STRING和EMBEDDED PDV则是为了通讯专门设计的。

ASN.1中的基本类型如Table 2-1中所示，更为复杂的类型可以由基本类型通过如Table 2-2中的组合类型构造得到。

**Table 2-1** 内建数据类型

|  |  |
| --- | --- |
| **类型** | **含义** |
| NULL | 只包含一个值NULL，用于传送一个报告或者作为CHOICE类型中某些值 |
| INTEGER | 全部整数（包括正数和负数） |
| REAL | 实数，表示浮点数 |
| ENUMERATED | 标识符的枚举（实例状态机的状态） |
| BIT STRING | 比特串 |
| OCTET STRING | 字节串 |
| OBJECT IDENTIFIER,  RELATIVE-OID | 一个实体的标识符，它在一个全世界范围树状结构中注册 |
| EXTERNAL, EMBEDDED PDV | 表示层上下文交换类型 |
| …String(除了BIT STRING、OCTET STRING外) | 各种字符串，有NumericString、PrintableString、VisibleStirng、ISO64String、IA5String、TeletexStirng、T61String、VideotexString、GraphicString、GeneralString、UniversalString、BMPString和UTF8String |
| CHARACTER STRING | 允许为字符串协商一个明确的字符表 |
| UTCTime, GeneralizedTime | 日期 |

**Table 2-2** 组合类型

|  |  |
| --- | --- |
| **类型** | **含义** |
| CHOICE | 在类型中选择(相当于C中的联合) |
| SEQUENCE | 由不同类型的值组成一个有序的结构(相当于C中的结构体) |
| SET | 由不同类型的值组成一个无序的结构 |
| SEQUENCE OF | 由相同类型的值组成一个有序的结构(相当于C中的数组) |
| SET OF | 由相同类型的值组成一个无序的结构 |

**2.3.2 类型定义**

<新类型的名字> ::= <类型描述>

其中：

<新类型的名字>是一个以大写字母开头的标识符；

<类型描述>是基于内建类型或在其它地方定义的类型。

如：

Married ::= BOOLEAN

Age ::= INTEGER

Picture ::= BIT STRING

Form ::= SEQUENCE

{

name PrintableString,

age Age,

married Married,

marriage-certificate Picture OPTIONAL

}

Payment-method ::= CHOICE

{

check Check-number,

credit-card SEQUENCE

{

number Card-number,

expiry-date Date

}

}

**注意**：在SEQUENCE和SET等(好像应该是所有组合类型的)定义中，最后一个成员结尾没有逗号“,”。

为了接收方能正确解码，发送方为每个值的类型附加一个数，称为tag，在描述中以“[]”标识。缺省情况下，编码器会使用universal的tag。很多时候，缺省情况下不能消除所有的模糊性，有必要明确指出各成员的tag。

如：

Coordinates ::= SET

{

x [1] INTEGER, //这证明好像也可以用类来直接声明变量

y [2] INTEGER,

z [3] INTEGER OPTIONAL

}

Afters ::= CHOICE

{

cheese [0] PrintableString,

dessert [1] PrintableString

}

**注意**：ASN.1允许递归式的类型分配(Assignment)，但我们应当保证其中包含至少一个非递归的值，因为编码规则无法处理无限的值。当然，绝大多数结构类型的成员都终结于简单类型。

为了准确描述一个类型，我们需要对值的集合进行一定的限制。这用到子类型约束，在类型之后用圆括号进行标识。

如：

Lottery-number ::= INTERGER(1..49)

Lottery-draw ::= SEQUENCE SIZE(6) OF Lottery-number

Upper-case-words ::= IA5String (FROM(“A”..”Z”))

Phone-number ::= NumericString (FROM(“0”..”9”))(SIZE(10))

Coordinates-in-plan ::= Coordinates (WITH COMPONENTS {…, z ABSENT})

有约束的类型当然还是类型，可以用在任何一个可以使用类型的地方。

最后，因为版本升级，在新的描述中，出现新的成员被加入到SEQUENCE、SET或者CHOICE或者在上述类型基础上添加约束而衍生的子类型时，两个连接的机器(特别是在开放网络中)不一定使用的是相同版本描述而生成的编解码器。为了防止一方因收到过多或者过少数据而出现错误，ASN.1中用符号“…”来标记可能以后是其它类型的地方。这样即使是旧的编解码器也不会因为描述扩充而导致编解码错误。

如：

Type ::= SEQUENCE

{

component1 INTERGER,

component2 BOOLEAN, -- version 1

…

}

以后新的版本中，描述可能为：

Type ::= SEQUENCE

{

component1 INTERGER,

component2 BOOLEAN,

**…**,

[[component3 REAL]], -- version 2

**…**

}

**注意**：新加入的类型成员要嵌套在“[[]]”中。

**2.4 值定义**

<新的值的名字> <该值的类型> ::= <值描述>

其中：

<新的值的名字>是以小写字母开头的标识符；

<该值的类型>可以是一个类型的名字，也可以是类型描述；

<值描述>是基于整数、字符串、标识符的组合。

如：

counter Lottery-number ::= 45

sextuple Lottery-draw ::= { 7, 12, 23, 31, 33, 41 }

pair Coordinates ::= { x 5, y -3 }

son-choice Afters ::= dessert:"profiterolles"

date Date ::= "20000824"

在ASN.1的应用中，绝大多数的值都是动态获得的，但是指定某些值，特别是在约束型子类型中，提高描述的可读性。

如：

upper-bound INTEGER ::= 12

Interval ::= INTEGER (0..upper-bound)

default-value Interval ::= 0

Pair ::= SEQUENCE

{

first [0] Interval DEFAULT default-value,

second [1] Interval DEFAULT default-value

}

**2.5 信息对象类和信息对象**

<信息对象类> ::= CLASS <类描述>

WITH SYNTAX <信息描述>

有时我们需要表达比注释更为正式的一些信息，例如我们要说明这样一个事实：一个结构类型的成员依赖于和这个结构类型相关的值。

如：

OPERATION ::= CLASS

{

&number INTEGER UNIQUE,

&Argument-type,

&Return-result-type

}

WITH SYNTAX

{

OPERATION NUMBER &number

TAKES AN ARGUMENT OF TYPE &Argument-type

AND RETURNS A VALUE OF TYPE &Return-result-type

}

在ASN.1中我们没有指出操作的具体内容，但是我们给出了一个机器要求另一个机器执行这个操作所需要交换的数据结构。

域&number以小写字母开头，后面跟了类型INTEGER。这个整数值用作信息的标识符，因为后面加了UNIQUE关键字。

域&Argument-type和&Return-result-type以大写字母开头，后面什么都没有跟。他们是一些ASN.1类型的引用（这可能会因为类型没有定义而不能编码）。

WITH SYNTAX块定义了一个用户友好的语法，说明了这个类的成员。

上述信息对象类的一个对象定义如下：

plan-projection OPERATION ::= //为什么是小子字母开头

{

OPERATION NUMBER 12

TAKES AN ARGUMENT OF TYPE Coordinates

AND RETURNS A VALUE OF TYPE Coordinates-in-plan

}

因为两个通讯中的机器使用相同的描述，因此相同的信息对象就能在这两机器间共享。

一个给定类的所有对象，可以在描述中收集到一个对象集合中，如下：

SupportedOperations OPERATION ::= //为什么大写开头还要特别注明呢？一直不都是以大写开头的吗？

{

plan-projection | translation | symmetry, **...**

}

**注意**：和对象不同，对象集合以大写字母开头。字符“**…**”表明新的对象可以由通讯中的机器动态加入。

**信息对象永远不会被编码**，传递他们各项信息的唯一方法是在一个类型中引用他们。我们用上面这个信息对象集合来约束一个结构类型：

Execute-operation ::= SEQUENCE

{

code OPERATION.&number({SupportedOperations}),

argument OPERATION.&Argument-type

({SupportedOperations}{@code})

}

Receive-result ::= SEQUENCE

{

code OPERATION.&number({SupportedOperations}),

result OPERATION.&Return-result-type

({SupportedOperations}{@code})

}

这段晦涩的描述，声明了这样一种情况：

Execute-operation操作它的编号在code成员中，当其argument发送给对方时，一定要满足域&Argument-type的类型。链接（或者表约束）“@code”说明这个参数是依赖于操作的。

对方返回的Receive-result也有类似约束。

**2.6 模块定义**

<模块名字> DEFINITIONS <缺省Tag> ::=

BEGIN

EXPORTS <导出描述>

IMPORTS <导入描述>

<模块体描述>

END

一般协议由一个或者多个模块组成，模块用来收集数据结构定义。

模块名字必须以大写字母开头。模块能以一种“全局指针”(Universal Pointer)的方式来引用，称为对象标识符（Object Identifier），用花括号标识在名字之后。

如：

Module2 { iso member-body(2) f(250) type-org(1) ft(16)

asn1-book(9) chapter5(0) module2(1) }

DEFINITIONS AUTOMATIC TAGS ::=

BEGIN

EXPORTS Type2;

IMPORTS Type1, value FROM Module1 {iso member-body(2)

f(250) type-org(1) ft(16) asn1-book(9)

chapter5(0) module1(0)};

Type2 ::= SEQUENCE OF Choice

Choice ::= CHOICE

{

a INTEGER (0..value),

b Type1

}

END

AUTOMATIC TAGS或者缺省，则说明不关注模块的tag交给编译器自己自动处理；如果需指明具体的tag值，则需要用“[]”来标识。此外tag的模式还可以是EXPLICIT TAGS和IMPLICIT TAGS。还可以使用EXTENSIBILITY IMPLIED来使模块中所有SEQUENCE、SET、CHOICE和ENUMERATED类型可扩展。

条目IMPORTS和EXPORTS（不是必须的）定义模块的接口。其中IMPORTS声明在其它模块定义但在本模块会用到的类型或者值；EXPORT声明在本模块之外可以访问的类型或者值。

**注意**IMPORTS的语法为：

IMPORTS <名字>, value FROM < 其它模块的Object Identifier >**;**

EXPORTS的语法为：

EXPORTS <名字>**;**

使用IMPORTS时，没有类似IMPORTS ALL的语句能将一个模块中所有定义都导入，因此需要协议描述者逐条写出每个需要导入的符号。对于参数化的分配，需要使用花括号，如：

ModuleName DEFINITIONS ::=

BEGIN

IMPORTS T{} FROM Module1;

U ::= T{INTEGER}

END

在BEGIN之后，IMPORTS之前的语句，则不使用导入的符号。

**2.7 模块与分配(Assignment)**

**2.7.2 本地和外部引用**

除本地应用外，ASN.1还支持外部应用，但不推荐使用。如：

MyType ::= SET OF OtherModule.Type

最好的方式依然是使用IMPORTS。

**2.7.1 分配Assignment**

有6种不同形式的数据与分配相关：类型、值、值的集合、信息对象类、信息对象和信息对象集合。

**I. 定义一个新类型**

如前所述，定义新类型，如：

TypeReference ::= CHOICE

{

integer INTEGER,

boolean BOOLEAN

}

在协议最高层定义的类型中，不被其它类型引用的那种类型称为PDU（Protocol Data Unit），PDU是应用件交换的数据。

**II. 定义一个值（绝对值）**

如前所述，这种值不被编码，通常是用作DEFAULT，上下界或者信息对象中的。如：

value-reference TypeReference ::= integer:12

如果两个类型在语法上是完全一样的，则这两种类型的值可以相互赋值。如：

Pair ::= SEQUENCE

{

x INTEGER,

y INTEGER

}

Couple ::= SEQUENCE

{

x INTEGER,

y INTEGER

}

pair Pair ::= {x 5, y 13}

couple Couple ::= pair

-------------------------------------------------------------------------------------------------------

Lighter-state ::= ENUMERATED

{

on(0),

off(1),

out-of-order(2)

}

Kettle-state ::= ENUMERATED

{

on(0),

off(1),

out-of-order(2)

}

lighter Lighter-state ::= on

kettle Kettle-state ::= lighter

在最后一个例子中，这种类型和值之间的对应，是由称为“ASN.1语义模型”()来负责的。

**III. 定义值集合**

在语义上，一个值集合相当于一个添加约束后的类型。 如：

PrimeNumbers INTEGER ::= {2 | 3 | 5 | 7 | 11 | 13}

 总结类型定义、值定义和值集合定义，我们可以得出这样的结论：

**Table 2-3** 分配语义定义

|  |  |  |  |
| --- | --- | --- | --- |
| **第一个词首字母** | **第二个词首字母** | **::=** | **分配内容** |
| 大写 | / | ::= | 类型或者信息对象类 |
| 小写 | 大写 | ::= | 值或者信息对象 |
| 大写 | 大写 | ::= | 值集合或者信息对象集合 |

**2.8 基本类型**

**Table 2-4** 基本类型汇总表

| **类型** | **UNIVERSAL Tag** | **取值** |
| --- | --- | --- |
| BOOLEAN | 1 | TRUE，FALSE |
| NULL | 5 | NULL |
| INTEGER | 2 | 整数 |
| ENUMERATED | 10 | 类型定义中列出的成员 |
| REAL | 9 | 实数 |
| BIT STRING | 3 | 比特串 |
| OCTET STRING | 4 | 八位组串，字节流 |
| OBJECT IDENTIFIER | 6 |  |
| RELATIVE-OID | 13 |  |

**2.8.1 BOOLEAN类型**

是两个值的集合：TRUE，FALSE。如下：

RoundResult ::= BOOLEAN

ok RoundResult ::= TRUE

ko RoundResult ::= FALSE

该类型的UNIVERSAL Tag值为1。

**2.8.2 NULL类型**

空类型，只有一个值NULL。作为一种结果，空类型是传输报告和响应的典型情况。

Ack ::= NULL

它经常在时间信息中使用，表示传送时间没有赋值，如：

Clock ::= CHOICE

{

time UTCTime,

out-of-order NULL

}

当时钟电池没有电时，值为：

battery-down Clock ::= out-of-order:NULL

NULL也可以用在表示链表结束的空接点，如：

LinkedList ::= SEQUENCE

{

data Data,

next CHOICE

{

linked-list LinkedList,

end NULL

}

}

当然，可以使用SEQUENCE OF来达到类似目的：

LinkedList ::= SEQUENCE OF Data

  {

**…**

}

该类型的UNIVERSAL Tag值为5。

**2.8.3 INTEGER类型**

整数可以是正整数也可以是负整数，取值范围在协议中没有界定。所以必须由编解码器提前确定好整数的范围，分配必要的内存空间来处理。可以通过在描述中增加一个值声明来判断：

Interval ::= INTEGER(123456789..123456790)

在使用PER编解码规则时，会按照本地整数范围进行截断；相比之下BER编码规则中因为有明确的长度，所以不会出现截断的情况。

虽然在数学可以有“-0”，但在ASN.1描述中不能这样描述。

一些情况下，比如为了定义错误码，需要给一些数值特定的名字，可以使协议更易于理解，也改善了应用层和编解码器之间的接口。这些信息当然可以用注释给出，但是ASN.1编译器不会使用这些信息，在词法分析开始，这些就都被忽略了。ASN.1为INTEGER类型提供了一种特殊语法来解决这个问题。以软盘驱动器的错误码为例：

ErrorCode ::= INTEGER

{

disk-full(1),

no-disk(-1),

disk-not-formatted(2)

}

stupid-error ErrorCode ::= disk-full

这些名字只能被用于定义ErrorCode类型的值，不能用于其它类型值的定义。

命名的整数不需要排序。对于没有命名的整数，还是可以使用的；已经命名的整数，也可以直接使用数字：

ok ErrorCode ::= 0

stupid-error ErrorCode ::= 1

注意：这种命名使用的使花括号，不是约束（使用圆括号）。

这种方式和ENUMERATED类型相比，区别在于：

<1>必须指定数值，不能像ENUMERATED中那样缺省；

<2>不能使用扩展标记“…”，当然这实际上不是个问题，和ENUMERATED不同，没有命名的整数还可以使用。

此外，这种方式如果使用不当，会造成混乱，要特别小心。如：

alpha INTEGER ::= 1

Type1 ::= INTEGER { alpha(2) }

Type2 ::= INTEGER { alpha(3), beta(alpha) }

gamma Type2 ::= beta

delta Type2 ::= alpha

实际上gamma的值为1，delta的值为3。

**注意**：在ASN.1中，INTEGER和REAL类型之间没有兼容性，就是说不能将INTEGER的值赋给一个REAL类型的值。同样的，INTEGER和ENUMERATED类型间也不存在兼容性。

UNIVERSAL的Tag值为2。

**2.8.4 ENUMERATED类型**

ENUMERATED类型与INTEGER类型相比， 区别在于：

<1>取值是受限制的，没有列出的数值不能使用；

<2>数值是不能操作的，只能用于编码；

<3>不用明确指定数值，ASN.1可自动计算相应的数值（从0开始）；

<4>可以使用扩展，保证和新版本的编码兼容；

<5>是数值关联(显式或者隐式)到标识符，而INTEGER中则是标识符显式关联到数值。

ABRT-diagnostic ::= ENUMERATED

{

no-reason-given(1),

protocol-error(2),

authentication-mechanism-name-not-recognized(3),

authentication-mechanism-name-required(4),

authentication-failure(5),

authentication-required(6),

**...**

}

不能直接使用数字，如：

abert-diagnostic ABERT-diagnostic ::= 4 -- 错误的！

在使用扩展标记“**...**”时，需要注意的是，在扩展标记后的成员，其数值必须是升序排列的。

A ::= ENUMERATED {a, b, ..., c(0)}

-- 错误： a 和 c 都是0

B ::= ENUMERATED {a, b, ..., c, d(2)}

-- 错误： c 和 d 都是2

C ::= ENUMERATED {a, b(3), ..., c(1)}

-- 正确： c = 1

和CHOICE、SEQUENCE和SET不同，在使用扩展标记“...”后，增加新成员时，ENUMERATED类型中不需要使用版本标识“[[ ]]”，并且只能有一个扩展标记。

注意事项：

<1>按照ASN.1的语义模型，任意两个ENUMERATED类型都是不兼容的。

<2>当一个ENUMERATED类型被IMPORTS到另一个模块中，则其列表中的标识符也被导入，但只能用于该类型值的赋值。

<3>基本列表(在扩展标识符之前)中的值不需要连续，也不需要排序。

<4>扩展列表(在扩展标识符之后)中的值不需要连续，但需要升序排列。

<5>扩展列表中的值不能和基本列表中的值重复，不论该值是明确指定的，还是缺省计算的。

扩展列表中成员的值必须比该列表中前一个成员的值大。(也就是所谓的升序吧)

表示成员值的必须是数字，不能是标识符。

该类型的UNIVERSAL Tag值为10。

**2.8.5 REAL类型**

从语义上等效于：

SEQUENCE

{

mantissa INTEGER (ALL EXCEPT 0),

base INTEGER (2|10),

exponent INTEGER

}

但是REAL类型有自己特定的UNIVERSAL Tag值。

 pi REAL ::= {314159, 10, -5}

e REAL ::= {271828128459045235360287, 10, -23}

zero REAL ::= 0

**注意**：实数0不能用三元组的方式给出。

我们用**PLUS-INFINITY**和**MINUS-INFINITY**分别表示“正无穷大”和“负无穷大”，这两个也不能用三元组表示。

此外，我们也要注意到ASN.1中的实数只能以2或者10为底（base的约束）。

UNIVERSAL的Tag值为9。

**2.8.6 BIT STRING类型**

按照ASN.1的定义，BIT STRING长度可以为0。它可以是原始的bit流形式，也可以是布尔向量模型。

**1)比特流形式：**

pi-decimals BIT STRING ::=

'00100100001111110110101010001000100001

01101000110000100011010011000100110001

100110001010001011100000001101110000'B

pi-decimals BIT STRING ::= '243F6A8885A308D313198A2E0370'H

**注意**：这种形式下，要考虑大小端的影响。

**2)布尔向量模型：**

如Figure 2-4所示的权限示意图，当使用不同的比特表示权限时，在ASN.1中可以这样描述：

Rights ::= BIT STRING

{

user-read(0),

user-write(1),

group-read(2),

group-write(3),

other-read(4),

other-write(5)

}

group1 Rights ::= { group-read, group-write }

当然，也可以之间用码字来赋值：

group2 Rights ::= '0011'B

group2 Rights ::= '3'H

group3 Rights ::= '001100'B

weird-rights Rights ::= '0000001'B

**注意**：

命名的比特列表不影响值的声明；

命名的比特列表没有限制，位置也不必连续，也不对类型的长度产生影响。如weird-rights。

以码字赋值时，起始的比特是必须的（最左面的比特），用来定位。后续的0比特无意义。group2和group3 与group1的结果是一样的。

需要对BIT STRING长度进行约束时，可以使用SIZE，如：

RIGHTS ::= BIT STRING (SIZE(6))

 和命名INTEGER类型相似，在命名BIT STRING中也不能使用扩展标记“**…**”。

 和OCTET STRING类型一样，可以使用CONTAINING 和ENCODED BY来约束。

 UNIVERSAL的Tag值为3。

**2.8.7 OCTET STRING类型**

与BIT STRING类型类似，但是单位不是比特而是8个比特的八位组。

 UNIVERSAL的Tag值为4。

**2.8.8 OBJECT IDENTIFIER类型**

OBJECT IDENTIFIER类型是对象的唯一标识的集合，这里对象可以是（但不限于）：

<1>一种抽象语法，一定是在表示层无歧义的注册过（ISO8822）；

<2>一种传输语法，一定是注册过，在表示层可以被抽象语法使用（ISO8823-1）；

<3>一种应用实体（ISO7498-3）；

<4>一个ASN.1模块；

<5>一种ROSE操作（ISO9072-2）；

等等。

我们用注册树来表征所有注册的标识：

**Figure 2-5** 注册树

OBJECT IDENTIFIER类型就是该注册树所有叶子的路径集合。一个OBJECT IDENTIFIER类型的值可以是：

internet-id OBJECT IDENTIFIER ::=

{ iso(1) identified-organization(3) dod(6) internet(1) }

francetelecom-id OBJECT IDENTIFIER ::=

{ iso member-body f(250) type-org(1) ft(16) }

ber-id OBJECT IDENTIFIER ::= { 2 1 1 }

即可以用各叶子的名字，可以用编号，也可以混合使用。

此外，对于多个有相同根名字的值，可以如下定义：

ID ::= OBJECT IDENTIFIER

id-edims ID ::= { joint-iso-itu-t mhs-motif(6) edims(7) }

id-bp ID ::= { id-edims 11 }

id-bp-edifact-ISO646 ID ::= { id-bp 1 }

id-bp-edifact-T61 ID ::= { id-bp 2 }

id-bp-edifact-octet ID ::= { id-bp 3 }

id-bp-ansiX12-ISO646 ID ::= { id-bp 4 }

id-bp-ansiX12-T61 ID ::= { id-bp 5 }

id-bp-ansiX12-ebcdic ID ::= { id-bp 6 }

每当一个对象注册到注册树中时，ISO标准ISO9834-1都会对它增加一个相应的文本描述，称之为ObjectDescriptor。如对上面ber-id的ObjectDescriptor为：

ber-descriptor ObjectDescriptor ::= "Basic Encoding of a single ASN.1 type"

**2.8.9 RELATIVE-OID类型**

前面OBJECT IDENTIFIER类型可以看作是对象在注册树中的绝对路径，而RELATIVE-OID类型则是相对路径。其参考基准节点可以是一个OBJECT IDENTIFIER类型的值（动态的），或者是收发双方默认的（静态的）。

RELATIVE-OID可以使用CONSTRAINED BY进行约束。

**2.9 字符串类型**

Known-multiple Character String Type指字符串中每个字符编码所占字节数都一样。

**Table 2-5** 字符串类型一览表

| **类型名字** | **Tag** | **字符表** | **ESC** | **Multi** |
| --- | --- | --- | --- | --- |
| NumericString | 18 | 字符“0”到“9”，空格 |  | 是 |
| PrintableString | 19 | 字符“A” 到 “Z”，“a”到“z”，“0”到“9”，空格，单引号（’），圆括号（( ，)），加号（+），逗号（,），减号（-），点（.），斜杠（/），冒号（:），等号（=），问号（?） |  | 是 |
| VisibleString  ISO646String | 26 | [ISOReg] entry no. 6; space |  | 是 |
| IA5String | 22 | [ISOReg] entry no. 1 & 6;  space, delete |  | 是 |
| TeletexString  T61String | 20 | [ISOReg] entry no. 6, 87, 102,  103, 106, 107, 126, 144, 150,  153, 156, 164, 165, 168; space, delete | 是 |  |
| VideotexString | 21 | [ISOReg] entry no. 1, 13, 72,  73, 87, 89, 102, 108, 126, 128,  129, 144, 150, 153, 164, 165,  168; space, delete | 是 |  |
| GraphicString | 25 | all the graphical sets (called  `G') of [ISOReg]; space | 是 |  |
| GeneralString | 27 | all the graphical sets (called  `G') and all the control characters  (called `C') of [ISOReg];  space, delete | 是 |  |
| UniversalString | 28 | [ISO10646-1] |  | 是 |
| BMPString | 30 | the basic multilingual plane  [ISO10646-1] (65,536 cells) |  | 是 |
| UTF8String | 12 | [ISO10646-1] |  |  |

**2.10 组合类型与扩展原则**

**2.10.1 Tag**

传输数据时，一定要避免接收方收到数据解码时出现不清楚的地方。具体说，接收方要明确知道每个收到数据的类型，我们就要对各类型进行系统性的编号。在ASN.1的第一种编解码规则BER（Basic Encode Rule）中，对数据编码使用的是三元组规则：<type, length, value>或者称为TLV，这里type或者称为tag，它是类型的标识符，在ASN.1中每种类型的tag是唯一的。length是编码数据value的字节数。value是实际数据的编码。

**I. Tags与Tagging Classes**

分配给每个类型的tag实际上是一个值对：<tagging class, number>。有四种tagging class：UNIVERSAL，context-specific，APPLICATION和PRIVATE。这种方式实际提供了不同的Tag空间。在一个上下文中，如果Tagging Class不同，则相同number的两个Tag也是不同的。

UNIVERSAL的Tag是ASN.1标准定义的，在描述中不能修改。

**Table 2-6** ASN.1中定义的UNIVERSAL类Tag

| **Tag** | **类型** |
| --- | --- |
| 0 | BER保留 |
| 1 | BOOLEAN |
| 2 | INTEGER |
| 3 | BIT STRING |
| 4 | OCTET STRING |
| 5 | NULL |
| 6 | OBJECT IDENTIFIER |
| 7 | ObjectDescripion |
| 8 | EXTERNAL, INSTANCE OF |
| 9 | REAL |
| 10 | ENUMERATED |
| 11 | EMBEDDED PDV |
| 12 | UFT8String |
| 13 | RELATIVE-OID |
| 14 | 保留 |
| 15 | 保留 |
| 16 | SEQUENCE, SEQUENCE OF |
| 17 | SET, SET OF |
| 18 | NumericString |
| 19 | PrintableString |
| 20 | TeletexString, T61String |
| 21 | VideotexString |
| 22 | IA5String |
| 23 | UTCTime |
| 24 | GeneralizedTime |
| 25 | GraphicString |
| 26 | VisibleString, ISO646String |
| 27 | GeneralString |
| 28 | UniversalString |
| 29 | CHARACTER STRING |
| 30 | BMPString |
| 31 | 保留 |

context-specific Tag是对SEQUENCE、SET和CHOICE及其成员使用的，可以在描述中自己定义，只要不产生歧义，相同数值可以在不同结构中反复使用。如：

A-possible-type ::= SET

{

integer [0] CHOICE

{

a [0] INTEGER,

b [1] INTEGER

},

boolean [1] CHOICE

{

a [0] BOOLEAN,

b [1] BOOLEAN

}

}

APPLICATION的Tag是为了“to define a data type that finds wide, scattered use within a particular application and that must be distinguishable (by means of its [ abstract syntax]) from all other data types used in the application”。在APPLICATION作用域内，其Tag也是唯一的。如：

Order-number ::= [APPLICATION 0] NumericString (SIZE (12))

但是因为使用IMPORTS等方式下，很难保证唯一性，所以这种Tag类(APPLICATION的Tag)已经不推荐使用了。

PRIVATE的Tag是为了“use a private-use Tagged [ type] to define a data type that finds use within a particular organization or country and that must be distinguishable (by means of its [ abstract syntax]) from all other data types used by that organization or country”。如一个公司可能这样扩展传输层PDU：

RejectTPDU ::= SET

{

destRef [0] Reference,

yr-tu-nr [1] TPDUnumber,

credit [2] Credit,

extended [PRIVATE 0] BOOLEAN DEFAULT FALSE

}

PRIVATE类的Tag现在也不推荐使用了。

**II. Tagging Mode**

使用context-specific类Tag，如下：

Afters ::= CHOICE

{

cheese [0]IA5String,

dessert [1]IA5String

}

当没有其它更多描述信息，并且使用BER编码时，会将UNIVERSAL类和context-specific类的Tag同时**编码**出来。这种方式称为EXPLICIT模式。

在下面这种情况：

T1 ::= [0] INTEGER

变更为：

T2 ::= [0] CHOICE

{

integer INTEGER,

real REAL

}

当CHOICE取值为integer时不需要变更。

与之相对应的是IMPLICIT模式，它将**只编码**context-specific类的Tag，直到另一个UNIVERSAL类Tag出现。如：

T ::= [1] IMPLICIT T1

T1 ::= [5] IMPLICIT T2

T2 ::= [APPLICATION 0] IMPLICIT INTEGER

对于类型T，只有Tag值 1被编码。

这种模式需要接收方能保证清楚知道抽象语法，能正确解码。

**III. Global Tagging Mode**

可以在模块定义时，声明模块全局Tag模式。可以是EXPLICIT TAGS、IMPLICIT TAGS和AUTOMATIC TAGS。

**IMPLICIT TAGS：**模块内所有SEQUENCE、SET和CHOICE的成员都是IMPLICIT模式（除非它是CHOICE类型、开放类型或者一个参数类型）。但不影响IMPORTS的内容。

**AUTOMATIC TAGS：**模块内所有SEQUENCE、SET和CHOICE类型ASN.1编译器会自动从0开始，步长为1进行自动编码。而其中的成员则用IMPLICIT模式，除非它是CHOICE类型、开放类型或者一个参数类型。

下面两个定义是等效的：

M DEFINITIONS AUTOMATIC TAGS ::=

BEGIN

T ::= SEQUENCE

{

a INTEGER,

b CHOICE

{

i INTEGER,

n NULL

},

c REAL

}

END

M DEFINITIONS ::=

BEGIN

T ::= SEQUENCE

{

a [0] IMPLICIT INTEGER,

b [1] EXPLICIT CHOICE

{

i [0] IMPLICIT INTEGER,

n [1] IMPLICIT NULL

},

c [2] IMPLICIT REAL

}

END

如果使用全局AUTOMATIC TAGS模式时，在描述中给一个类型指定了一个Tag，那么对这个类似的AUTOMATIC TAGS就关闭。如果一个SEQUENCE或者SET类型包含COMPONENTS OF条目，而且这个条目在自动编号前展开；但自动标号又要求在展开前进行，ASN.1编译器要自己检查这种冲突。

对新的协议现在都推荐使用AUTOMATIC TAGS。

**2.10.2 SEQUENCE**

SEQUENCE是最常用的ASN.1类型之一，它可以嵌套使用，还可以使用OPTIONAL、DEFAULT和COMPONENTS OF等对成员进行修饰。

一个常见的SEQUENCE声明如：

Description ::= SEQUENCE

{

surname IA5String,

first-name IA5String,

age INTEGER

}

该类型的一个值为：

johnny Description ::=

{

surname "Smith",

first-name "John",

age 40

}

使用OPTIONAL和DEFAULT时，SEQUENCE类型应该能被明确解码，不存在模棱两可的地方。如下这个类型就有问题：

T ::= SEQUENCE

{

x INTEGER,

y INTEGER OPTIONAL,

z INTEGER DEFAULT 0,

t INTEGER

}

在使用BER编码规则时，对于值t1 T::={x 1, y 2, t 3}而言，接收方收到的码流如下：



**Figure 2-6** OPTIONAL引起的歧义

那么在解码过程中，V为2的值无法确定是成员y还是成员z。这种情况下，该类型的定义应该在Tag上做一些补充：

T ::= SEQUENCE

{

x INTEGER, -- [UNIVERSAL 2]

y [**0**] INTEGER OPTIONAL,

z [**1**] INTEGER DEFAULT 0,

t INTEGER -- [UNIVERSAL 2]

}

**注意**：对于接收方，没有收到声明DEFAULT的成员和收到DEFAULT的值是等价的。

在类型定义中，可以使用COMPONENTS OF将另一个类型的成员插入到当前类型中：

Registration ::= SEQUENCE

{

COMPONENTS OF Description,

marital-status ENUMERATED

{single, married, divorced,widowed}

}

COMPONENTS的效果使得上面的定义等效于：

Registration ::= SEQUENCE

{

surname IA5String,

first-name IA5String,

age INTEGER,

marital-status ENUMERATED

{single, married, divorced, widowed}

}

而**不是**：

Registration ::= SEQUENCE

{

description Description,

marital-status ENUMERATED

{single, married, divorced,widowed}

}

使用COMPONENTS OF的好处在于，如上例中，Description更新升级时，Registration的内容不需要手工修改。

**2.10.3 SET**

和SEQUENCE相比，SET的成员是没有顺序的。同样可以使用OPTIONAL、DEFAULT和COMPONENTS OF。

**2.10.4 SEQUENCE OF**

SEQUENCE OF相当于某些语言中的动态数组或者列表：所有成员都是一个类型，数目不定。如：

PariTierce DEFINITIONS ::=

BEGIN

SteepleChase ::= SEQUENCE OF INTEGER

END

相应的典型值为：

winningCombination SteepleChase ::= {5, 2, 12}

no-one-arrived SteepleChase ::= { }

ASN.1提供一种元标记来编写和描述项目的注释、文档，以“@”开头，后接模块名称、类型名字成员序号或者“\*”（表示全部），如：

-- @PariTierce.SteepleChase.\* must be distinct

-- @PariTierce.SteepleChase.0 = 5 (horse no. 5 always wins!)

如果是在模块内，则“@”和模块名都可以省略，如：

-- SteepleChase.\* must be distinct

-- SteepleChase.0 = 5 (horse no. 5 always wins!)

**2.10.5 SET OF**

和SEQUENCE OF相比，SET OF的成员是顺序的，其它类似。

**2.10.6 CHOICE**

CHOICE类型表示一种选择，类似UNION。该提供两个信息：被选择的选项，相应的值。也就是说对于类型：

Afters ::= CHOICE

{

cheese [0] IA5String,

dessert [1] IA5String

}

它的一个值而言：

mine Afters ::= dessert:"profiteroles"

（注意：表示方法先是选择的选项dessert，而后是“:”，最后是值 ）

没有必要先定义一个枚举类型来表征具体的选项。

和SEQUENCE、SET类型不同，CHOICE没有缺省的UNIVERSAL Tag值，因为它是一些类型的集合，其中被选择项目的Tag会被作为CHOICE相关Tag。在和SEQUENCE、SET一起使用时，要注意使用AUTOMATIC TAGS或者context-specific Tags。如下面，类型U的定义有问题（Tag 0 和 1都重复了两次）：

T ::= CHOICE { a [0] INTEGER,b [1] NULL }

U ::= SET

{

x [0] REAL,

y T,

z CHOICE

{

c [1] BIT STRING,

d [2] OCTET STRING

}

}

U的定义应该修改为类似下面：

U ::= SET

{

x [0] REAL,

y [1] T,

z [2] CHOICE

{

c [1] BIT STRING,

d [2] OCTET STRING

}

}

在协议中，CHOICE类型经常出现。应用层间交换的PDU，通常都是CHOICE结构的，如ROSE（Remote Operation Service Element －ISO13712-1）的简化：

ROS ::= CHOICE

{

invoke [1] Invoke,

returnResult [2] ReturnResult,

returnError [3] ReturnError,

reject [4] Reject

}

某些时候，我们想重用CHOICE类型中的某些选项的类型，此时需要使用在ASN.1中被称为选择符（Selection）的 符号“<”。如：

Element ::= CHOICE

{

atomic-no INTEGER (1..103),

symbol PrintableString

}

MendeleievTable ::= SEQUENCE SIZE (103) OF symbol < Element

einsteinium symbol < Element ::= "Es"

**2.10.7 类型扩展**

可以在定义中插入扩展标记“…”来定义一个类型是可扩展的。在ASN.1中可扩展的类型有ENUMERATED、SEQUENCE、SET和CHOICE。INTEGER和BIT STRING类型，基于前面的介绍，是隐含可扩展类型（对于类似ENUMERATED方式，未命名的数字和比特位置都可以使用）。如：

State ::= ENUMERATED {on, off, out-of-order, ...}

Description ::= SEQUENCE

{

surname IA5String,

first-name IA5String,

age INTEGER,

...

}

Dimensions ::= SET

{

x INTEGER,

y INTEGER,

...

}

Afters ::= CHOICE

{

cheese IA5String,

dessert IA5String,

...

}

按照标准，扩展标记可以有两个，他们将类型定义分割为两个扩展部分：

T ::= SEQUENCE

{

a A,

b B,

...,

...,

c C

}

在扩展时，新增的项目可能有多个，并且其中可能还带有OPTIONAL和DEFAULT。这种情况下，解码器在解码中就会遇到困难。为了区分不同版本，在扩展部分，可以使用版本符号“[[ ]]”，如：

Afters ::= CHOICE

{

cheese IA5String,

dessert IA5String,

...!ExtensionPb:greedy,

[[coffee NULL ]], -- version 2

[[cognac IA5String]] -- version 3

}

**Figure 2-7** 相同协议不同扩展之间接力示例

如果扩展部分只有一个成员，则版本符号可以省略。如：

State ::= ENUMERATED {on, off, out-of-order, ..., stand-by} -- version 2

Dimensions ::= SET

{

x INTEGER,

y INTEGER,

...,

z INTEGER -- version 2

}

Afters ::= CHOICE

{

cheese IA5String,

dessert IA5String,

...,

coffee NULL -- version 2

}

而类型T的新定义中：

T ::= SEQUENCE

{

a A,

b B,

...,

[[d D,

e E]],

...,

c C

}

d、e放在一个版本符号中，说明在T类型的一个值中，一个出现则另一个必须也出现。

当解码扩展类型值时，我们会发现：

           没有预期的扩展或者

           出现不期望的扩展

此时，解码器会产生一个协议错误，并且按照应用设计方指定的具体动作来操作：通常是忽略不期望扩展或者为预期的扩展赋缺省值；同时解码器可能还会以事先约定的信号通知应用层。这个约定的信号就是异常Exception，在抽象语法中以“!”开头，后接一个值（可以是数、字符串等），放在扩展标记之后。至于应当如何应对异常，就不是ASN.1关心的内容了。如：

Description ::= SEQUENCE

{

surname IA5String,

first-name IA5String,

age INTEGER,

...!extended-description

}

extended-description INTEGER ::= 1

Dimensions ::= SET

{

x INTEGER,

y INTEGER,

... !IA5String:"dimension error"

}

Afters ::= CHOICE

{

cheese IA5String,

dessert IA5String,

...!ExtensionPb:greedy,

coffee NULL,

cognac IA5String

}

ExtensionPb::= ENUMERATED {greedy, ...}

应用扩展时需要注意的规则：

           如果模块在头声明中使用了AUTOMATIC TAGS，而且组合类型的根（即类型入口）没有指明Tag，则自动编码过程中，先对根进行自动Tag编码，再对扩展部分进行Tag编码。

           如果根不是手工Tag编码，则扩展成员也不能手工编码。

           SET或者CHOICE类型中，扩展部分，要求是按照一定Tag的顺序列出的。Tagging Class的顺序是UNIVERSAL、APPLICATION、PRIVATE和context-specific，每个类中Tag都是升序排列的。

           当模块定义中使用了EXTENSIBILITY IMPLIED，或者在类型定义结尾只有一个扩展标记，或者在第二个扩展标记前，可以在SEQUENCE、SET和CHOICE类型的扩展插入点，加入一个虚拟成员或者可选项，这样BER、CER或者DER解码器必要时可以通过跟踪具体的异常来确认第一个版本的抽象语法是否能接收未知的扩展成员。这个虚拟成员的Tag和其它ASN.1的Tag不同，为[IMAGINARY 0]。如下：

Person ::= SET

{

surname [0] IA5String,

first-name [1] IA5String,

contact CHOICE

{

phone-number [2] NumericString,

e-mail-address [3] NumericString,

...,

imaginary [IMAGINARY 0] T

},

info CHOICE

{

age [4] INTEGER,

...,

imaginary [IMAGINARY 0] T

}

}

为了避免将来互通问题，对于正在发展的协议，建议在模块定义部分使用EXTENSIBILITY IMPLIED，如：

ModuleName DEFINITIONS AUTOMATIC TAGS

EXTENSIBILITY IMPLIED ::=

BEGIN

-- ...

END

**2.11 子类型约束**

**2.11.1 单值约束（Single Value Constraint）**

最简单的的子类型约束就是单值约束，即将一个类型限制为一个值，以圆括号将该值列在类型声明之后即可，如：

Two ::= INTEGER (2)

Day ::= ENUMERATED

{

monday(0), tuesday(1), wednesday(2), thursday(3), friday(4), saturday(5), sunday(6)

}

Wednesday ::= Day (wednesday)

FourZ ::= IA5String ("ZZZZ")

Afters ::= CHOICE

{

cheese IA5String,

dessert ENUMERATED

{

profiterolles(1), sabayon(2),fraisier(3)

}

}

CompulsoryAfters ::= Afters (dessert:sabayon)

如果该值的选择可以是多个中的一个，则可以用“|”将被选值列出，如：

WeekEnd ::= Day (saturday|sunday)

PushButtonDial ::= IA5String ("0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"|"\*"|"#")

**2.11.2 类型包含约束（Type Inclusion Constraint）**

当声明一个类型和另一个类型拥有同样的约束时，只需要将被参考类型列出，如：

FrenchWeekEnd ::= Day (WeekEnd)

还可以这样使用：

LongWeekEnd ::= Day (WeekEnd|monday)

注意：在1994以前的标准中必须使用INCLUDES关键字，以后的标准不再是必须的。

**2.11.3 值域约束（Value Range Constraint）**

对于数学中的整数和实数，可以用区间来约束。在ASN.1中使用“..”来表示区间，“<”用来限定边界，如：

Number ::= INTEGER

From3to15 ::= Number (3..15)

From3excludedTo15excluded ::= Number (3<..<15)

关键字MIN和MAX用来表征基本类型的最小、最大值。

对于实数类型，下面两个定义：

T ::= REAL (0..<fmantissa 5,base 10,exponent 0g)

U ::= T (fmantissa 2,base 10,exponent 0g..MAX)

的结果使得类型U等价于：

U ::= REAL ({mantissa 2,base 10,exponent 0}..<{mantissa 5,base 10,exponent 0})

**2.11.4 大小约束（Size Constraint）**

主要对类型内存空间大小进行约束，对基本类型的约束，如：

Exactly31BitsString ::= BIT STRING (SIZE (31))

StringOf31BitsAtTheMost ::= BIT STRING (SIZE (0..31))

EvenNumber ::= INTEGER (2|4|6|8|10)

EvenLengthString ::=IA5String (SIZE (EvenNumber))

NonEmptyString ::= OCTET STRING (SIZE (1..MAX))

前两个类型的区别在于，第二个的长度是一个区间，从0到31；而第一个的长度固定是31。

对SEQUENCE OF和SET OF的约束，如：

ListOfStringsOf5Characters ::= SEQUENCE OF PrintableString (SIZE (5))

ListOf5StringsOf5Characters ::=

SEQUENCE (SIZE (5)) OF PrintableString (SIZE (5))

**2.11.5 字符表约束（Alphabet Constraint）**

主要是对字符串类型的值进行约束，如：

Morse ::= PrintableString (FROM ("."|"-"|" "))

IDCardNumber ::=NumericString (FROM ("0".."9"))

PushButtonDialSequence ::=IA5String (FROM ("0".."9"|"\*"|"#"))

下面这个类型定义，虽然没有错，但却没有意义：

WrongType ::= IA5String (FROM ("Albert".."Zoe"))

因为字符串类型定义需要声明字符而不是字符串。

下面两个定义是等价的：

Dna ::= PrintableString (FROM ("TAGC"))

Dna::= PrintableString (FROM ("T"|"A"|"G"|"C"))

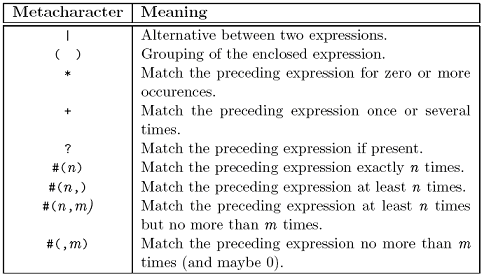
**2.11.6 正则表达式约束（Regular Expression Contraint）**

也是用于对字符串值的约束，通过关键字PATTERN来声明具体的正则表达式。如：

DateAndTime ::= VisibleString(PATTERN "\d#2/\d#2/\d#4-\d#2:\d#2")

-- DD/MM/YYYY-HH:MM

正则表达式的用法就不详细介绍，下面给出正则表达式的含义：



**Figure 2-8** 正则表达式含义

**2.11.7 对SEQUENCE OF或者SET OF成员的约束**

如前所述，我们可以对SEQUENCE OF或者SET OF进行约束。但是，当其成员是从其它模块IMPORTS而来时，我们不能添加约束来重定义。因此我们通过WITH COMPONENT来进入到被组合类型的定义，再添加约束。如：

TextBlock ::= SEQUENCE OF VisibleString

添加约束：

AddressBlock ::= TextBlock (WITH COMPONENT (SIZE (1..32)))

DigitBlock ::= TextBlock (WITH COMPONENT (NumericString))

这两个定义相当于：

AddressBlock ::= SEQUENCE OF VisibleString (SIZE (1..32))

DigitBlock ::= SEQUENCE OF VisibleString (NumericString)

当然，关键字WITH COMPONENT可以和其它约束一起使用，如：

IntegerMatrix ::= SEQUENCE SIZE (6) OF SEQUENCE SIZE (6) OF INTEGER

CoordinateMatrix ::=

IntegerMatrix (WITH COMPONENT (WITH COMPONENT (-100..100)))

相当于：

CoordinateMatrix ::=

SEQUENCE SIZE (6) OF SEQUENCE SIZE (6) OF INTEGER (-100..100)

**2.11.8 对SEQUENCE、SET或者CHOICE成员的约束**

对于SEQUENCE和SET多个成员进行约束，我们使用WITH COMPONENTS来添加约束，注意关键字结尾有S（因为SEQUENCE和SET成员是不同类型的）。

对类型：

Quadruple ::= SEQUENCE

{

alpha ENUMERATED {state1, state2, state3},

beta IA5String OPTIONAL,

gamma SEQUENCE OF INTEGER,

delta BOOLEAN DEFAULT TRUE

}

添加约束后为：

Quadruple1 ::= Quadruple (WITH COMPONENTS

{

...,

alpha (state1),

gamma (SIZE (5))

})

符号“…”指示只对显式声明的成员进行约束，其它成员保留。注意对SEQUENCE类型，声明约束时仍然要保持成员的顺序。

对于声明为OPTIONAL的成员（声明为DEFAULT的也类似），还可以使用关键字PRESENT和ABSENT来表示在约束中该成员出现还是不出现。如：

Quadruple2 ::= Quadruple (WITH COMPONENTS

{

alpha (state1),

beta (SIZE (5|12)) **PRESENT**,

gamma (SIZE (5)),

delta **OPTIONAL**

})

如果约束列出了成员列表，则PRESETN和ABSENT可以省略，如：

Quadruple3 ::=

Quadruple (WITH COMPONENTS falpha, beta, gammag)

WITH COMPONENTS还可以应用于REAL、EMBEDDED PDV、EXTERNAL、CHARACTER STRING和INSTANCE OF。

应用于CHOICE类型时，WITH COMPONENTS用来指示哪些选项被禁止或者哪些选项可供选择。

对类型：

Choice ::= CHOICE { a A, b B, c C, d D }

添加约束后为：

ChoiceCD ::= Choice (WITH COMPONENTS {..., a ABSENT, b ABSENT})

ChoiceA1 ::= Choice (WITH COMPONENTS {..., a PRESENT})

ChoiceA2 ::= Choice (WITH COMPONENTS {a PRESENT})

ChoiceBCD ::= Choice (WITH COMPONENTS {a ABSENT, b, c})

**2.11.9 对OCTET STRING内容的约束**

使用CONTAINING和ENCODED BY来对OCTET STRING类型的内容进行约束，以达到更为灵活的传输语法。

如：

MoreCompact ::= OCTET STRING (CONTAINING MyType ENCODED BY {joint-iso-itu-t asn1 packed-encoding(3) basic(0) unaligned(1)})

MyType ::= SEQUENCE { -- ....... -- }

这里明确指定了OCTET STRING的内容是MyType并且使用PER进行编码。则ASN.1编译器在这部分将调用PER编解码器。

ENCODE BY可以指明具体的编码规则，可以是PER、CER等，也可以是用ECN编码控制标记描述的自定义编码规则。如果没有ENCODED BY，则该部分使用和当前模块相同的编码方式。

对没有CONTAINING的部分，如：

Document ::= OCTET STRING (ENCODED BY pdf)

pdf OBJECT IDENTIFIER ::= { -- OID pour le codage PDF -- }

ASN.1编译器将调用对应的解码器如Acrobat Reader浏览器。

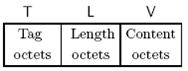
**2.11.10 约束的组合**

可以将前面所述的各种约束适当组合，得到更为准确的描述，如：

PhoneNumber ::= NumericString (FROM ("0".."9"))(SIZE (10))

描述的是电话号码。

除此之外，常见的组合类型有：



**Figure 2-9** 四种常见的组合操作

           C1 UNION C2或者C1|C2方式，前面已经介绍过。如：

PushButtonDialSequence ::= IA5String (FROM ("0".."9"|"\*"|"#"))

           C1 INTERSECTION C2或者C1^C2方式，如：

SaudiName ::= BasicArabic (SIZE (1..100) ^ Level2)

           C1 EXCEPT C2方式，如：

Lipogramme ::= IA5String (FROM (ALL EXCEPT ("e"|"E")))

           ALL EXCEPT C1方式，如：

InvokeId ::= CHOICE { present INTEGER, absent NULL }

DAP-InvokeIdSet ::= InvokeId (ALL EXCEPT absent:NULL)

**2.11.11 约束的扩展**

在约束中也可以添加扩展标记，如：

A ::= INTEGER (0..10, ...)

在新版本中，可以这样扩展：

A ::= INTEGER (0..10, ..., 12)

类型A的值可以为0到10和12。

和SEQUENCE、SET和CHOICE类型不同，在约束中不能第二个扩展标记，也没有版本描述符“[[ ]]”。此外，在模块声明中，EXTENSIBILITY IMPLIED语句不会对约束产生影响。

可以使用异常描述符来明确定义可以通知给接收方应用层不符合根约束时的异常，如：

E ::= INTEGER (1..10, ...!Exception:too-large-integer)

Exception ::= ENUMERATED {too-large-integer, ...}

特别在SIZE约束中使用扩展情况下，添加异常声明就很有用。

在后面讨论参数化时，当一个子类型约束包含一个抽象语法的参数，那么这个约束就是隐含扩展的，因为每个参数的实例都会修改约束。如：

ImplementedUnivStr{UniversalString:Level} ::=

UniversalString (FROM ((Level UNION BasicLatin))

!characterSet-problem)

characterSet-problem INTEGER ::= 4

通常也建议在自定义约束中添加异常描述。

**2.11.12 自定义约束**

除了ASN.1中前面介绍的约束，还可以通过关键字CONSTRAINED BY来引入自定义约束，它通常都会在编解码过程中引入特定的处理。例如，在ASN.1/C++接口（TeleManagement Forum, X/Open. { ASN.1/C++. Application Programming Interface. { Report 1.0 (draft 10a), 1996. ftp://ftp.tmforum.org/nmfsets/component/cs322/）中，在有用户自定义约束类型的一个值编码前或者解码后，一个程序员指定的函数就会被调用。这个函数可以检查信息约束已被遵守，并且数据中的操作已被执行。

自定义约束还能用来指示SEQUENCE或者SET间成员的复杂约束。

通常在CONSTRAINED BY之后是一些注释，这些在扫描阶段会被忽略。但自定义约束要求编译器在生成文件中编解码部分插入一个对用户自定义函数的调用。

示例：

Reject ::= SEQUENCE {

invokeId InvokeId,

problem CHOICE

{

general [0] GeneralProblem,

invoke [1] InvokeProblem,

returnResult [2] ReturnResultProblem,

returnError [3] ReturnErrorProblem

}

} (CONSTRAINED BY {-- must conform to the above --

-- definition --} ! RejectProblem:general-mistypedPDU)

用户自定义约束经常会用到参数，参数被放在花括号中。如：

Encrypted{TypeToBeEnciphered} ::= BIT STRING (CONSTRAINED

BY {-- must be the result of the encipherment --

-- of some BER-encoded value of --

TypeToBeEnciphered} !Error:securityViolation)

Error ::= ENUMERATED {securityViolation}

**2.12 表示上下文交换类型**

**2.12.1 EXTERNAL类型**

不推荐使用。可以用EMBEDDED PDV类型替代。

它的等效SEQUENCE结构为：

EXTERNAL ::= [UNIVERSAL 8] IMPLICIT SEQUENCE

{

direct-reference OBJECT IDENTIFIER OPTIONAL,

indirect-reference INTEGER OPTIONAL,

data-value-descriptor ObjectDescriptor OPTIONAL,

encoding CHOICE {

single-ASN1-type [0] ANY,

octet-aligned [1] IMPLICIT OCTET STRING,

arbitrary [2] IMPLICIT BIT STRING

}

}

**2.12.2 EMBEDDED PDV类型**

EMBEDDED PDV类型等效于：

EMBEDDED PDV ::= [UNIVERSAL 11] IMPLICIT SEQUENCE

{

identification CHOICE

{

syntaxes SEQUENCE

{

abstract OBJECT IDENTIFIER,

transfer OBJECT IDENTIFIER

},

syntax OBJECT IDENTIFIER,

presentation-context-id INTEGER,

context-negotiation SEQUENCE

{

presentation-context-id INTEGER,

transfer-syntax OBJECT IDENTIFIER

},

transfer-syntax OBJECT IDENTIFIER,

fixed NULL

},

data-value OCTET STRING

}

其中identification的选项：

           syntaxes（注意有s）表征编码规则中抽象语法和传输语法的对象标识；

           syntax表征抽象语法和编码规则；

           presentation-context-id是协商后的表示上下文（只用于OSI环境中，即一对抽象语法和传输语法）；

           context-negotiation用于表示上下文协商，可能在连接建议开始或者连接过程中上下文修改；

           transfer-syntax。此时，抽象语法应该是应用设计者固定采用的方式，并且发送、接收双方都已明确；

           fixed表明发送、接收双方都已知道抽象语法和传输语法。

**2.12.3 CHARACTER STRING类型**

CHARACTER STRING是EMBEDDED PDV的一个具体应用，在一些特殊场合下的字符串类型，可能不想如前述标准类型那样采用标准的编码方式，而是希望其字符抽象语法和字符传输语法能在通讯系统标识层间协商或者在环境合适时直接使用字符串。该类型等效于：

CHARACTER STRING ::= [UNIVERSAL 29] SEQUENCE

{

identification CHOICE

{

syntaxes SEQUENCE

{

abstract OBJECT IDENTIFIER,

transfer OBJECT IDENTIFIER

},

syntax OBJECT IDENTIFIER,

presentation-context-id INTEGER,

context-negotiation SEQUENCE

{

presentation-context-id INTEGER,

transfer-syntax OBJECT IDENTIFIER

},

transfer-syntax OBJECT IDENTIFIER,

fixed NULL

},

string-value OCTET STRING

}

与EMBEDDED PDV相比，蓝色部分不相同。

## 2.13 信息对象类、信息对象和信息对象集合

在ASN.1中，信息对象类用于正式描述那些类型和值单独不能表达的属性，这些属性通常被翻译作为类型和值之间的语法链接。在协议描述中，可以先不考虑这些限制性属性，以后再通过信息对象类添加。虽然信息对象不会被编码，但是ASN.1编译器在生成编解码器时却是要用到它的。

### 2.13.1 基本语法Default Syntax

信息对象类用关键字Class，名字以大写字母开头。其成员域以“&”开头，名字是小写字母或者大写字母开头。符号“&”使得信息对象类的域和SEQUENCE和SET的成员有明显的区别。因此，如果一个名字是以“&”开头，则一定是一个信息对象类的域。域名字的后面可以是一个类型或者其它类或者本类中其它域，而且可以用OPTIONAL、DEFAULT或者UNIQUE标识。和SEQUENCE、SET不同的是，类中域不仅可以是值，也可以是一个类型，值集合，信息对象或者信息对象集合，因此OPTIONAL和DEFAULT的含义更为广泛。如：

FUNCTION ::= CLASS

{

&ArgumentType ,

&ResultType DEFAULT NULL,

&Errors ERROR OPTIONAL,

&code INTEGER UNIQUE

}

对应的一个信息对象（表示两个数相加的操作）为：

addition-of-2-integers FUNCTION ::=

{

&ArgumentType SEQUENCE { a INTEGER, b INTEGER },

&ResultType INTEGER,

-- empty error list by default

&code 1

}

关于域的名字是大写还是小写字母，请看下表：

**Table 2-7** 域名字的大小写含义



下面这个示例使用到了上面所有7中情况：

OTHER-FUNCTION ::= CLASS

{

&code INTEGER (0..MAX) UNIQUE,

&Alphabet BMPString

DEFAULT {Latin1 INTERSECTION Level1},

&ArgumentType ,

&SupportedArguments &ArgumentType OPTIONAL,

&ResultType DEFAULT NULL,

&result-if-error &ResultType DEFAULT NULL,

&associated-function OTHER-FUNCTION OPTIONAL,

&Errors ERROR DEFAULT

{rejected-argument | memory-fault}

}

rejected-argument ERROR ::= {-- object definition --}

memory-fault ERROR ::= {-- object definition --}

相应的，代表两个数相加操作的信息对象为：

other-addition-of-2-integers OTHER-FUNCTION ::=

{

&ArgumentType Pair,

&SupportedArguments {PosPair | NegPair},

&ResultType INTEGER,

&result-if-error 0,

&code 1

}

Pair ::= SEQUENCE {a INTEGER, b INTEGER}

PosPair ::= Pair (WITH COMPONENTS {a(0..MAX), b(0..MAX)})

NegPair ::= Pair (WITH COMPONENTS {a(MIN..0), b(MIN..0)})

### 2.13.2 用户友好语法 User-Friendly Syntax

用户友好语法是由关键字WITH SYNTAX在信息对象类定义中引入的，放在定义之后。其中，对有OPTIONAL、DEFAULT标记的域，相应部分要用“[ ]”来吧标记。如：

OTHER-FUNCTION ::= CLASS

{

&code INTEGER (0..MAX) UNIQUE,

&Alphabet BMPString

DEFAULT {Latin1 INTERSECTION Level1},

&ArgumentType ,

&SupportedArguments &ArgumentType OPTIONAL,

&ResultType DEFAULT NULL,

&result-if-error &ResultType DEFAULT NULL,

&associated-function OTHER-FUNCTION OPTIONAL,

&Errors ERROR DEFAULT

{rejected-argument | memory-fault}

}

WITH SYNTAX

{

ARGUMENT TYPE &ArgumentType,

[SUPPORTED ARGUMENTS &SupportedArguments,]

[RESULT TYPE &ResultType,

[RETURNS &result-if-error IN CASE OF ERROR,]]

[ERRORS &Errors,]

[MESSAGE ALPHABET &Alphabet,]

[ASSOCIATED FUNCTION &associated-function,]

CODE &code

}

memory-fault ERROR ::= {-- object definition --}

注意：

           全部大写的词不能是类型或者值。

           如果出现连续的可选部分，他们的名字不能相同。

           可选部分RETURNS是在RESULT之中的，这样就不能不管&ResultType域而给&result-if-error分配值。

           为每个域之前添加一个大写的名字不是必须的，但是建议这样做。

当一个信息对象类有用户友好语法是，相应信息对象的定义时就要使用。如：

addition-of-2-integers OTHER-FUNCTION ::=

{

ARGUMENT TYPE Pair,

SUPPORTED ARGUMENTS {PosPair | NegPair},

RESULT TYPE INTEGER,

RETURNS 0 IN CASE OF ERROR,

CODE 1

}

这是，名字，逗号等必须和WITH SYNTAX中的顺序一致。

ASN.1的语法模型也会使用WITH SYNTAX，因此应当避免该部分过于详细、冗长。

### 2.13.3 示例：X.500协议中的ATTRIBUTE和MATCHING-RULE

信息对象类定义：

ATTRIBUTE ::= CLASS

{

&derivation ATTRIBUTE OPTIONAL,

&Type OPTIONAL,

&equality-match MATCHING-RULE OPTIONAL,

&ordering-match MATCHING-RULE OPTIONAL,

&substrings-match MATCHING-RULE OPTIONAL,

&single-valued BOOLEAN DEFAULT FALSE,

&collective BOOLEAN DEFAULT FALSE,

&no-user-modification BOOLEAN DEFAULT FALSE,

&usage Attribute-Usage

DEFAULT userApplications,

&id OBJECT IDENTIFIER UNIQUE

}

WITH SYNTAX

{

[SUBTYPE OF &derivation]

[WITH SYNTAX &Type]

[EQUALITY MATCHING RULE &equality-match]

[ORDERING MATCHING RULE &ordering-match]

[SUBSTRINGS MATCHING RULE &substrings-match]

[SINGLE VALUE &single-valued]

[COLLECTIVE &collective]

[NO USER MODIFICATION &no-user-modification]

[USAGE &usage]

ID &id

}

AttributeUsage ::= ENUMERATED

{userApplications(0), directoryOperation(1), distributedOperation(2), dSAOperation(3) }

MATCHING-RULE ::= CLASS

{

&AssertionType OPTIONAL,

&id OBJECT IDENTIFIER UNIQUE

}

WITH SYNTAX

{

[SYNTAX &AssertionType]

ID &id

}

信息对象定义：

name ATTRIBUTE ::=

{

WITH SYNTAX DirectoryString

EQUALITY MATCHING RULE caseIgnoreMatch

ID {joint-iso-itu-t ds(5) attributeType(4) 2} }

DirectoryString ::= CHOICE

{

teletexString TeletexString (SIZE (1..maxSize)),

printableString PrintableString (SIZE (1..maxSize)),

universalString UniversalString (SIZE (1..maxSize)),

bmpString BMPString (SIZE (1..maxSize)),

utf8String UTF8String (SIZE (1..maxSize))

}

maxSize INTEGER ::= 25

caseIgnoreMatch MATCHING-RULE ::=

{

SYNTAX DirectoryString

ID {id-mr 2}

}

id-mr OBJECT IDENTIFIER ::= { joint-iso-itu-t ds(5) matchingRule(13)

}

### 2.13.4 值集合与信息对象集合

前面已经多次介绍过值集合以及四种关系。这里要说明的是，当在信息对象集合中使用扩展标记时，这说明该集合是动态扩展的，通讯中的应用层在运行中可以增加或者删除该集合中的值。需要注意的是，类型、子类型、值集合使用的扩展标记是静态扩展的，只能在新版本的描述中增加或者删除，不能动态修改。如：

ExtensibleMatchingRules MATCHING-RULE ::=

{caseIgnoreMatch | booleanMatch | integerMatch, ... }

### 2.13.5 访问对象、对象集合中的信息

从对象中抽取信息，我们在对象参考后使用“.”。如：

caseIgnoreMatchValue caseIgnoreMatch.&AssertionType ::=

printableString:"Escher"

具体的约定如下表：

**Table 2-8** 对象信息抽取约定



### 2.13.6 示例

surname ATTRIBUTE ::=

{ -- family name

SUBTYPE OF name

WITH SYNTAX DirectoryString

ID id-at-surname

}

givenName ATTRIBUTE ::=

{ -- first name

SUBTYPE OF name

WITH SYNTAX DirectoryString

ID id-at-givenName

}

countryName ATTRIBUTE ::=

{ -- country

SUBTYPE OF name

WITH SYNTAX PrintableString (SIZE (2)) -- [ISO3166] codes

SINGLE VALUE TRUE

ID id-at-countryName

}

SupportedAttributes ATTRIBUTE ::={surname | givenName | countryName}

一个简单的应用信息对象的例子：

AttributeIdAndValue3 ::= SEQUENCE

{

ident ATTRIBUTE.&id({SupportedAttributes}),

value ATTRIBUTE.&Type({SupportedAttributes}{@ident})

}

注意使用了符号“@”，表示value与indent之间的关系。此外还使用对象集合SupportedAttributes来进行限定。该类型的一个值为：

val AttributeIdAndValue3 ::=

{

ident id-at-countryName ,

value PrintableString:"F"

}

当多个SEQUENCE、SET、CHOICE、SEQUENCE OF和SET OF嵌套在一起时，使用符号“@”指定组合类型的成员过程中，需要和“.”结合在一起使用，才能保证含义明确。如果形式为“@indent1.indent2.indent3”，那么indent1应当是从该组合类型的最高层开始的；如果形式为“@.indent1”，则indent应当是从该组合类型的最低层开始的。如前面AttributeIdAndValue3还可以定义为：

AttributeIdAndValue3 ::= SEQUENCE

{

ident ATTRIBUTE.&id({SupportedAttributes}),

value ATTRIBUTE.&Type({SupportedAttributes}{@.ident})

}

一个更为复杂的例子为：

FilterItem ::= CHOICE

{

equality [0] AttributeValueAssertion,

substrings [1] SEQUENCE {

type Attribute.&id({SupportedAttributes}),

strings SEQUENCE OF CHOICE

{

initial [0] ATTRIBUTE.&Type

({SupportedAttributes}{@substrings.type}),

any [1] ATTRIBUTE.&Type

({SupportedAttributes}{@substrings.type}),

final [2] ATTRIBUTE.&Type

({SupportedAttributes}{@substrings.type}) }

},

greaterOrEqual [2] AttributeValueAssertion,

lessOrEqual [3] AttributeValueAssertion,

present [4] AttributeType,

approximateMatch [5] AttributeValueAssertion,

extensibleMatch [6] MatchingRuleAssertion

}

又如：

Attribute-desc ::= SEQUENCE

{

usage ATTRIBUTE.&usage({SupportedAttributes}),

list SEQUENCE OF SEQUENCE

{

ident ATTRIBUTE.&id({SupportedAttributes}{@usage}),

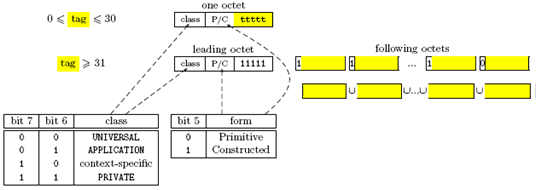
value ATTRIBUTE.&Type

({SupportedAttributes}{@usage,@.ident})

}

}

注意这里value部分使用了[{@usage,@.ident](mailto:%7b@usage,@.ident)}，既和usage相关，又和最低层的ident相关。用图表示该结构为：



**Figure 2-10** 双成员相关约束示例

该一个典型值为：

att-desc Attribute-desc ::=

{

usage userApplications,

list

{

{ ident id-at-objectClass, value oid },

{ ident id-at-aliasedEntryName, value distinguishedName }

}

}

### 2.13.7 预定义TYPE-IDENTIFIER信息对象类和INSTANCE OF类型

预定义信息对象类TYPE-IDENTIFIER的定义如下：

TYPE-IDENTIFIER ::= CLASS

{

&id OBJECT IDENTIFIER UNIQUE,

&Type

}

WITH SYNTAX

{

&Type IDENTIFIED BY &id

}

一个应用示例为：

Authentication-value ::= CHOICE

{

charstring [0] IMPLICIT GraphicString,

bitstring [1] BIT STRING,

external [2] EXTERNAL,

other [3] IMPLICIT SEQUENCE

{

other-mechanism-name MECHANISM-NAME.&id({ObjectSet}),

other-mechanism-value MECHANISM-NAME.&Type

({ObjectSet}{@.other-mechanism-name})

}

}

在ASN.1中经常出现从TYPE-IDENTIFIER中抽取信息的情况，为此提供预定义类型INSTANCE OF，其典型形式为：

SEQUENCE

{

type-id DefinedObjectClass.&id ({ObjectSet }),

value [0] DefinedObjectClass.&Type

({ObjectSet gf@.type-id})

}

INSTANCE OF只用来抽取TYPE-IDENTIFIER的信息，而且通常建议将对象集合列出。INSTANCE OF的UNIVERSAL Tag和EXTERNAL一样都是8。

一个应用为：

ExtendedBodyPart ::= SEQUENCE

{

parameters [0] INSTANCE OF TYPE-IDENTIFIER OPTIONAL,

data INSTANCE OF TYPE-IDENTIFIER

}

(CONSTRAINED BY {-- must correspond to the &parameters --

-- and &data fields of a member of -- IPMBodyPartTable})

### 2.13.8 预定义ABSTRACT-SYNTAX信息对象类

预定义信息对象类ABSTRACT-SYNTAX的定义如下：

ABSTRACT-SYNTAX ::= CLASS

{

&id OBJECT IDENTIFIER,

&Type ,

&property BIT STRING {handles-invalid-encodings(0)}

DEFAULT {}

}

WITH SYNTAX

{

&Type IDENTIFIED BY &id

[HAS PROPERTY &property]

}

该信息对象类应用比较少，主要是在PDU部分有应用。如：

ProtocolName-Abstract-Syntax-Module {iso member-body(2)

f(250) type-org(1) ft(16) asn1-book(9) chapter15(3) protocol-name(0)}

DEFINITIONS ::= BEGIN

IMPORTS ProtocolName-PDU FROM ProtocolName-Module {iso

member-body(2) f(250) type-org(1) ft(16) asn1-book(9) chapter15(3) protocol-name(0)module1(2)};

protocolName-Abstract-Syntax ABSTRACT-SYNTAX ::=

{ProtocolName-PDU IDENTIFIED BY protocolName-Abstract-Syntax-id}

protocolName-Abstract-Syntax-id OBJECT IDENTIFIER ::=

{iso member-body(2) f(250) type-org(1) ft(16) asn1-book(9) chapter15(3) protocol-name(0) abstract-syntax(0)}

protocolName-Abstract-Syntax-descriptor ObjectDescriptor

::= "Abstract syntax of ProtocolName"

protocolName-Transfer-Syntax-id OBJECT IDENTIFIER ::=

{iso member-body(2) f(250) type-org(1) ft(16) asn1-book(9) chapter15(3) protocol-name(0) transfer-syntax(1)}

protocolName-Transfer-Syntax-descriptor ObjectDescriptor

::= "Transfer syntax of ProtocolName"

END

在表示层数据定义中：

PDV-list ::= SEQUENCE

{

transfer-syntax-name Transfer-syntax-name OPTIONAL,

presentation-context-identifier Presentation-context-identifier,

presentation-data-values CHOICE

{

single-ASN1-type [0] ABSTRACT-SYNTAX.&Type

(CONSTRAINED BY {-- Type which corresponds to --

-- the presentation context identifier --}),

octet-aligned [1] IMPLICIT OCTET STRING,

arbitrary [2] IMPLICIT BIT STRING

}

}

## 2.14 宏Macro

自1994版本标准后，宏Macro已经被信息对象类和信息对象取代了。但在一些比较早的协议中（如X.400与X.500协议），还使用了Macro。

宏定义需要用到关键词：MACRO、BEGIN、END、TYPE NOTATION和VALUE NOTATION。

基本结构为：

MACRO-NAME MACRO ::=

BEGIN

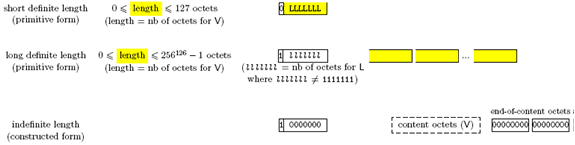
TYPE NOTATION ::= -- type syntax --

VALUE NOTATION ::= -- value syntax --

-- grammatical productions used for defining

-- the type syntax and the value syntax

END



### 2.14.1 例1-COMPLEX NUMBER

COMPLEX MACRO ::=

BEGIN

TYPE NOTATION ::= "Re" "=" type(ReType) ","

"Im" "=" type(ImType)

VALUE NOTATION ::= value(ReValue ReType) "+"

value(ImValue ImType) "i"

<VALUE SEQUENCE { real-p ReType,

imaginary-p ImType} ::=

{ real-p ReValue,

imaginary-p ImValue }>

END

相应的，取值可以为：

c1 COMPLEX

Re = INTEGER,

Im = INTEGER ::=

5 + 3 i

由“<”“>”部分定义的是宏等效的“纯”ASN.1定义，如c1等效定义为：

c1 SEQUENCE

{

real-p INTEGER,

imaginary-p INTEGER

} ::= {

real-p {5},

imaginary-p {3}

}

### 2.14.2 例2-OPERATION

OPERATION MACRO ::=

BEGIN

TYPE NOTATION ::= Argument Result Errors LinkedOperations

VALUE NOTATION ::= value (VALUE

CHOICE { localValue INTEGER,

globalValue OBJECT IDENTIFIER })

Argument ::= "ARGUMENT" NamedType | empty

Result ::= "RESULT" ResultType | empty

Errors ::= "ERRORS" "{" ErrorNames "}" | empty

LinkedOperations ::=

"LINKED" "{" LinkedOperationNames "}"

| empty

ResultType ::= NamedType | empty

NamedType ::= identifier type | type

ErrorNames ::= ErrorList | empty

ErrorList ::= Error | ErrorList "," Error

Error ::= value(ERROR) | type

LinkedOperationNames ::= OperationList | empty

OperationList ::= Operation | OperationList "," Operation

Operation ::= value(OPERATION) | type

END

相应一个值为：

get-subscriber-name OPERATION

ARGUMENT NumericString (SIZE (10))

RESULT IA5String

ERRORS {unknown, db-unavailable}

::= localValue:1

## 2.15 参数化Parameterization

为了描述更为通用的数据结构，在ASN.1中，可以对SEQUENCE、SET和CHOICE进行参数化，参数以花括号表示，放在名字之后。如：

DirectoryString{INTEGER:maxSize} ::= CHOICE

{

teletexString TeletexString (SIZE (1..maxSize)),

printableString PrintableString (SIZE (1..maxSize)),

universalString UniversalString (SIZE (1..maxSize)),

bmpString BMPString (SIZE (1..maxSize)),

utf8String UTF8String (SIZE (1..maxSize))

}

其中参数maxSize是以小写字母开头的，而且有INTEGER类型限定（governed）。

下表详细描述了各种参数化的情况：

**Table 2-9** 参数分类



使用上述参数化类型：

SubstringAssertion ::= SEQUENCE OF CHOICE

{

initial [0] DirectoryString{ub-match},

any [1] DirectoryString{ub-match},

final [2] DirectoryString{ub-match}

}

ub-match INTEGER ::= 128

当然SubstringAssertion类型更为通用的形式是（这种情况成为参数的传播）：

SubstringAssertion{INTEGER:maxSize} ::= SEQUENCE OF CHOICE

{

initial [0] DirectoryString{ub-match},

any [1] DirectoryString{ub-match},

final [2] DirectoryString{ub-match}

}

参数的作用域范围和它右面定义范围相同。

当参数是一个类型时，要注意其tag应当是明确的。如果模块不是AUTOMATIC TAGS模式，则需要明确指定。

此外，当参数是一个类型时，也可以用它作为另一个值或者值集合参数的限定，如：

GeneralForm{T, T:val} ::= SEQUENCE

{

info T DEFAULT val,

comments IA5String

}

该类型的一个特化是：

Form ::= GeneralForm{BOOLEAN, TRUE}

在信息对象中使用参数化：

MESSAGE-PARAMETERS ::= CLASS

{

&max-priority-level INTEGER,

&max-message-buffer-size INTEGER,

&max-reference-buffer-size INTEGER

}

WITH SYNTAX

{

MAXIMUM PRIORITY &max-priority-level

MAXIMUM MESSAGE BUFFER &max-message-buffer-size

MAXIMUM REFERENCE BUFFER &max-reference-buffer-size

}

Message-PDU{MESSAGE-PARAMETERS:param} ::= SEQUENCE

{

priority INTEGER (0..param.&max-priority-level !Exception:priority),

message UTF8String (SIZE(0..param.&max-message-buffer-size)

!Exception:message),

comments UTF8String (SIZE(0..param.&max-reference-buffer-size)

!Exception:comments)

}

Exception ::= ENUMERATED {priority(0), message(1), comments(2), ...}

当在模块中导入、导出参数化类型时，建议在其后用花括号给出参数化引用，如：

Forward{OPERATION:OperationSet} OPERATION ::=

{

OperationSet | OperationSet.&Linked.&Linked |

OperationSet.&Linked.&Linked.&Linked.&Linked

}

Reverse{OPERATION:OperationSet} OPERATION ::=

{

Forward{{OperationSet.&Linked}}

}

在模块导入、导出时：

ModuleName DEFINITIONS ::=

BEGIN

EXPORTS Forward{}, Reverse{}, ForwardAndReverse;

IMPORTS

OPERATION FROM Remote-Operations-Information-Objects

{joint-iso-itu-t remote-operations(4)

informationObjects(5) version1(0)}

Forward{}, Reverse{} FROM Remote-Operations-Useful-Definitions

{joint-iso-itu-t remote-operations(4)

useful-definitions(7) version1(0)};

-- dynamically extensible object set:

MyOperationSet OPERATION ::= {...}

-- non-parameterized definition:

ForwardAndReverse OPERATION ::=

{Forward{{MyOperationSet}} UNION Reverse{{MyOperationSet}}}

END

前面讨论过参数传播的情况，但是一般传播迟早都会结束于最高层的类型：PDU。它构成了通讯系统间交换消息的抽象语法（Abstract Syntax）。如果最终的PDU中因具体实现而还留有参数，这种称为抽象语法参数（Parameter of Abstract Syntax），它只能出现在子类型约束中。如：

CharacterString{INTEGER:max-length} ::= CHOICE

{

teletexString TeletexString (SIZE (1..max-length)

!exceeds-max-length),

printableString PrintableString (SIZE (1..max-length)

!exceeds-max-length)

}

exceeds-max-length INTEGER ::= 999

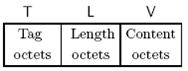
# 3 编码规则和传输语法

本章主要介绍BER和PER两种编码规则及其衍生规则。

## 3.1 BER

### 3.1.1 基本规则

BER（Basic Encoding Rules）是ASN.1中最早定义的编码规则，在讨论详细编码规则时，我们是基于正确的抽象描述上。BER传输语法的格式一直是TLV三元组<Type, Length, Value>也可以认为是<Tag, Length, Value>，见Figure 3-1。TLV每个域都是一系列八位组，对于组合结构，其中V还可以是TLV三元组，见Figure 3-2。BER传输语法是基于八位组（为了避免不同系统上的混淆，没有采用Byte为单位）的，自定界的编码，因为其中L明确界定了八位组的长度。BER是大端编码的，其八位组的高位比特在左手边，见Figure 3-3。



**Figure 3-1** TLV三元组

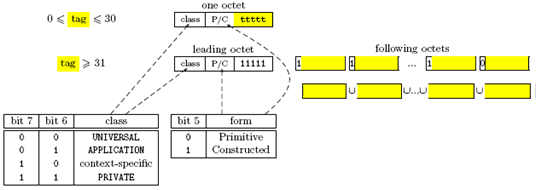


**Figure 3-2** 传输语法示例



**Figure 3-3** 大端编码

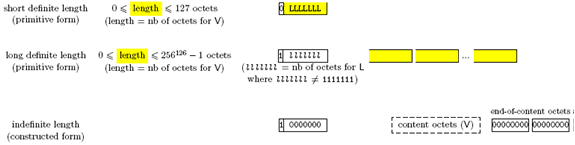
BER编码中的Tag（通常是一个八位组），指明了值的类型，其中一个比特表征是基本类型还是组合类型。Tag有如下两种形式：



**Figure 3-4** Tag的两种形式

当Tag不大于30时，Tag只在一个八位组中编码；当Tag大于30时，则Tag在多个八位组中编码。在多个八位组中编码时，第一个八位组后五位全部为1，其余的八位组最高位为1表示后续还有，为0表示Tag结束。Tag的值需要将上图中黄色部分拼接后才能得到。

BER编码中Length表示Value部分所占八位组的个数，有两大类：定长方式（Definite Form）和不定长方式（Indefinite Form）；在确定方式中，按照Length所占的八位组个数又分为短、长两种形式。具体如下：



**Figure 3-5** Length的三种形式

采用定长方式，当长度不大于127个八位组时，Length只在一个八位组中编码；当长度大于127时，在多个八位组中编码，此时第一个八位组低七位表示的是**Length**所占的长度，后续八位组表示**Value**的长度。

采用不定长方式时，Length所在八位组固定编码为0x80，但在Value编码结束后以两个0x00结尾。这种方式使得可以在编码没有完全结束的情况下，可以先发送部分消息给对方。

BER编码规则的Object Identifier注册为{joint-iso-itu-t(2) asn1(1) base-encoding(1)}，其Object Description为“"Basic Encoding of a single ASN.1 type”。

### 3.1.2 各类型的编码

本小节中以UNIVERSAL Tag和短型Value为例，讨论各种类型的BER编码，重点关注Value部分。在举例中，n10表示数字n是十进制数。

I. BOOLEAN

只能以primitive方式编码。

FALSE的编码为：



**Figure 3-6**  BOOLEAN: FALSE的编码

TRUE的编码（任何不是全0都可以）为：



或者：



**Figure 3-7** BOOLEAN: TRUE的编码

II. NULL

只能以primitive方式编码，且只有一个值：



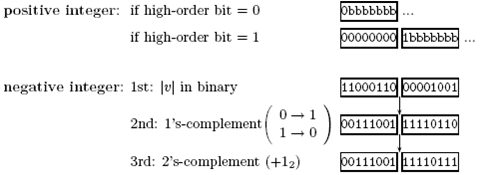
**Figure 3-8** NULL的编码

III. INTEGER

只能以primitive方式编码。

我们分编、解码两个过程，正数、负数两种情况来讨论。

1)         编码过程：

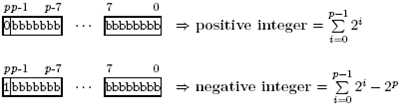


**Figure 3-9** INTEGER编码过程

           对于正数，如果最高比特位为0则直接编码；如果为1，则在最高比特位之前增加一个全0的八位组。

           对于负数，先取绝对值，再取反，最后加1。

2)         解码过程：



**Figure 3-10** INTEGER解码过程

基于前面所述的规则，整数-27,066的编码为：

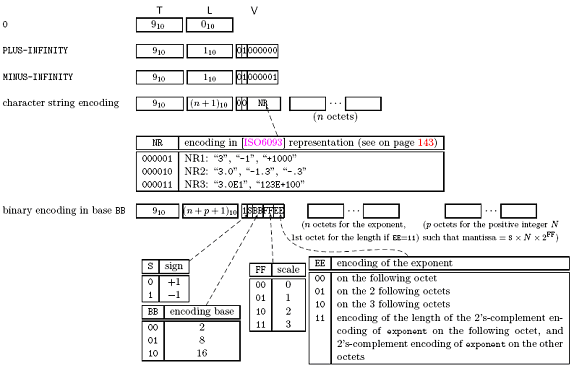


**Figure 3-11** 整数-27,066的编码

IV. ENUMERATED

ENUMERATED的值按照前面整数值的规则编码。

V. REAL



**Figure 3-12** REAL类型编码

VI. BIT STRING

可以primitive方式编码或者constructed方式编码。

采用primitive方式，对'1011011101011'B的编码规则如下：



**Figure 3-13** BIT STRING: '1011011101011'B的primitive form编码

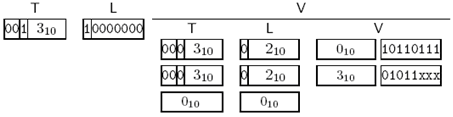
注意在'1011011101011'B前增加了一个八位组，取值为0到7，表征这个值最后补位的个数。由发送方决定补位采用0还是1。

如果BIT STRING的值为空，则编码时，长度为1，补充的八位组为全0。



**Figure 3-14** BIT STRING值为空的编码

constructed方式是在发送时，有部分编码还不能确定时采用的，前一个值的编码如下：



**Figure 3-15** BIT STRING: '1011011101011'B的constructed form编码

注意Length部分采用的是不定长编码。

VII. OCTET STRING

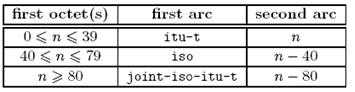
与BIT STRING类似，但是不需要增加表征补充位个数的八位组。

VIII. OBJECT IDENTIFIER

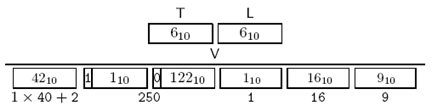
只能以primitive方式编码。

编码时，第一个八位组采用公式：first\_arc \* 40+second\_arc。

**Table 3-1** OBJECT IDENGTIFIER第一个八位组解码



对{iso member-body f(250) type-org(1) ft(16) asn1-book(9)}的编码为：

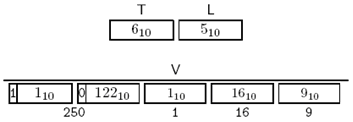


**Figure 3-16** OBJECT IDENTIFIER编码示例

注意对250的编码方式。

IX. RELATIVE-OID

与OBJECT IDENTIFIER类似，但是不需要对注册树前两段进行特殊处理。对{f(250) type-org(1) ft(16) asn1-book(9)}的编码为：



**Figure 3-17** RELATIVE-OID编码示例

X. 字符串和日期

和OCTET STRING编码类似，只是Tag不同。

XI. SEQUENCE

肯定是constructed形式的。

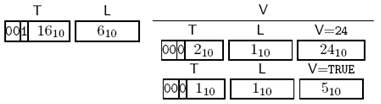
在编码SEQUENCE时，其每个成员都要以TLV三元组方式编码，而且顺序要与SEQUENCE定义的一致。对于标记为DEFAULT的成员，即使发送方应用层给出了值，也是由发送者决定是否对该成员进行编码。

对如下定义：

v SEQUENCE { age INTEGER, single BOOLEAN } ::=

{ age 24, single TRUE }

的编码为：



**Figure 3-18** SEQUENCE编码示例

如果SEQUENCE定义中包括扩展符，则省略；如果有扩展的成员则编码。

XII. SET

与SEQUENCE类似，但是成员顺序有发送者决定。

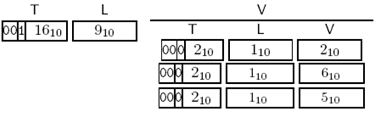
XIII. SEQUENCE OF

SEQUENCE OF的Tag与SEQUENCE相同，编码规则也相同。

对定义为：

triplet SEQUENCE OF INTEGER ::= {2, 6, 5}

的编码为：



**Figure 3-19** SEQUENCE OF编码示例

XIV. SET OF

与SEQUENCE OF类似。

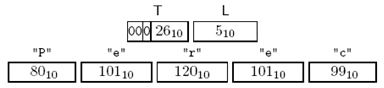
XV. CHOICE

严格说CHOICE类型在编码中并不存在，只是在描述中体现一种关系。编码时，是按照具体被选择的成员编码规则编码的。

对定义为：

famous CHOICE { name VisibleString, nobody NULL } ::= name:"Perec"

的编码为：



**Figure 3-20** CHOICE编码示例

如果CHOICE类型显式（EXPLICIT）指定了Tag，那么该Tag应当以constructed方式编码，具体在XVI. 中讨论。

如果CHOICE类型包含有扩展符则忽略；如果有扩展成员则编码。

XVI. Tagged Value

如果一个类型的Tag是隐式（IMPLICIT）的（或者在模块定义中声明了IMPLICIT TAGS或者AUTOMATIC TAGS），则只有出现在关键字IMPLICIT左侧的Tag才会被编码。

如定义：

v [1] IMPLICIT INTEGER ::= -38

的编码为：



**Figure 3-21** IMPLICIT TAG编码示例

如果一个类型的Tag是显式（EXPLICIT）的（或者在模块定义中声明了EXPLICIT TAGS），则要以constructed方式编码三元组系列。

如定义：

v [APPLICATION 0] EXPLICIT INTEGER ::= 38

的编码为：



**Figure 3-22** EXPLICIT TAG编码示例

注意第一个Tag是APPLICTION、constructed方式，Length是后续八位组的长度。

XVII. 子类型约束

因为子类型约束是在BER编码规则之后被引入ASN.1的，所以在编码规则中不能体现约束。

XVIII. EXTERNAL

该类型不推荐使用，略。

XIX. INSTANCE OF

定义为：

v INSTANCE OF TYPE-IDENTIFIER ::=

{

type-id {iso member-body f(250) type-org(1) ft(16) asn1-book(9)

chapter18(5) integer-type(0)},

value INTEGER:5

}

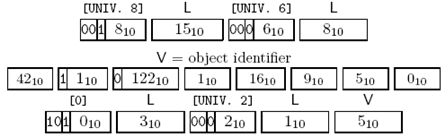
的编码应当和如下一个SEQUENCE类型的值相同：

{ direct-reference {iso member-body f(250) type-org(1) ft(16) asn1-book(9)

chapter18(5) integer-type(0)},

encoding single-ASN1-type:INTEGER:5 }

编码为：



**Figure 3-23** INSTANCE OF的编码示例

XX. EMBEDDED PDV

EMBEDDED PDV的编码和其等效的SEQUENCE结构编码类似。其嵌入部分的编码，应该是遵循identification成员指定的规则。

XXI. CHARACTER STRING

CHARACTER STRING的编码和其等效的SEQUENCE结构编码类似。其嵌入字符串部分的编码，应该是遵循identification成员指定的规则。

XXII. Information Objects and Object Sets

信息对象和对象集合永远都不编码。如前所述，传递他们所包含信息的途径是在值定义或者类型定义中引用他们。这样的结果，不是在编码中出现了这些信息，而是ASN.1编译器在生成编解码器时，会按照这些信息生成相应的约束表。

XXIII. Value Set

值集合的编码按照对应类型的编码规则进行。

### 3.1.3 BER编码规则的属性

           BER编码规则是机器无关的：通讯应用可以很容易支持大端和这种编码格式；而且能支持各种不同的整数长度。

           BER的传输语法是十分冗长的：T和L很多情况下都是可以省略的，但是这种冗余信息一些情况下也有显著的优势，它能很好的保藏抽象语法结构，BER传输语法能容易升级而且向上兼容。如每个类型都可以用CHOICE来代替：

T ::= IA5String -- old version

可以更新为：

T ::= CHOICE {

iA5String IA5String,

universalString UniversalString } -- new version

当旧的解码器收到universalString时，它可以明确知道收到的不是iA5String，而且知道具体长度，可以很容易忽略这个信元。

在此基础上，可以看出，BER规则中SEQUENCE、SET都是缺省可扩展的。另外，对ENUMERATED，BER也没有对边界进行限定，因此也是可扩展的。

当整个抽象语法都是显式（EXPLICIT）Tag时，解码器能在不了解具体抽象语法的情况下解码，能以更为用户友好的方式展示结果。如对BOOLEAN类型显示为“TRUE”或者“FALSE”而不是码字。

置于长度，如果系统性的使用长度，可以根据使用情况更为有效传递，而不用严格传输诸如SIZE（200）。

当然，BER的优势在某些方面也成了不足之处，这也导致了其它编码规则的产生。

### 3.1.4 一个完整的例子

这里，以一个HTTP的片段作为示例：

MyHTTP DEFINITIONS AUTOMATIC TAGS ::=

BEGIN

GetRequest ::= SEQUENCE

{

header-only BOOLEAN,

lock BOOLEAN,

accept-types AcceptTypes,

url Url,

...

}

AcceptTypes ::= SET

{

standards BIT STRING {html(0), plain-text(1), gif(2),

jpeg(3)} (SIZE (4)) OPTIONAL,

others SEQUENCE OF VisibleString (SIZE (4))

OPTIONAL

}

Url ::= VisibleString (FROM ("a".."z"|"A".."Z"|"0".."9"|"./-\_~%#"))

v GetRequest ::=

{

header-only TRUE,

lock FALSE,

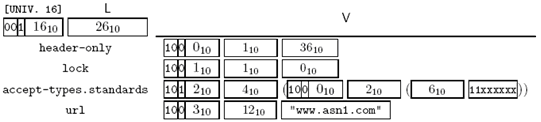
accept-types { standards {html,plain-text} },

url "www.asn1.com"

}

END

值v对应的BER编码结果为：



**Figure 3-24** BER编码示例

## 3.2 CER与DER

CER与DER（Canonical and Distinguished Encoding Rules）

### 3.2.1 更多限制规则的需求

在X.400 和X.500中应用接力传递消息，会使用到X.509数字签名（Digital Signature），其过程如下图所示：

**Figure 3-25** 接力传递消息

发送方发送v的编码结果c1(v)，同时发送其数字签名σ(c1(v))；中继收到消息后，解码得到v，保留签名，发送时重新编码发送结果为c2(v)；接收方收到结果后，解码得到v，同时对c2(v)计算数字签名σ(c2(v))，通过比较σ(c1(v))与σ(c2(v))是否相同来确定消息内容是否被修改。

在上面的过程中，如果直接使用BER编码规则，比如存在BOOLEAN类型的值“TRUE”，则因为两次编码过程c1和c2可能对TRUE的编码结果不同，从而导致数字签名比较不通过。

通常，我们称不给因具体实现和编码过程动态因素留出自由发挥余地的编码规则为规范编码规则（Canonical）。从BER中派生出两种规范编码规则，分别为CER（Canonical Encoding Rules）和DER（Distinguished Encoding Rules）。这两种规则是BER规范的特例，因此BER解码器能解码这两种规则的传输语法，反之则不然。

这两套编码规则为抽象值和他们的编码提供了一一对应的关系。应用这个属性，接收应用层可以不用了解具体结构和类型，就能比较接收的八位组流。

这两套规则的主要区别在于：CER针对不定长格式，而DER针对定长格式。因此CER常应用在需要传输大量数据的应用的。

**Table 3-1** CER、DER对BER的限制

| **CER** | **DER** |
| --- | --- |
| **长度域Length L** | |
|         primitive类型采用定长形式（有长、短两种情况）；          constructed类型采用不定长形式。 | 不管是primitive还是constructed类型，L都是定长形式的。 |
| **BOOLEAN** | |
|         值“TRUE”的编码为：11111111。 | |
| **REAL** | |
|         if the base component equals 10, a real value is encoded as a character string of NR3 format without spaces [ISO6093]; it has no “+" sign if the number is positive; the dot is the decimal separator; the mantissa must not begin nor end with zeros; it is followed by a dot and the E character; the exponent must not use the “+" sign nor begin with a 0 but for the null exponent denoted “+0" | |
|  |         [X.509] forbids base-10-real values in the Directory applications |
|         if base = 2, a real value is binary-encoded with mantissa M and exponent E so that the mantissa equals 0 or is an odd number; the mantissa and the exponent are encoded on the minimum number of octets; the scale factor F equals 0 | |
| **BIT STRING** | |
|         最后一个八位组中未使用的比特为0；          如果包含一个命名列表，最后的0比特不编码；如果该类型有SIZE约束，则按照需要补充并且编码最后的0比特；如果取值为空，则长度为1，值为全0的八位组。 | |
| **BIT STRING或者OCTET STRING或者CHARACTER STRING** | |
|         如果小于1000个八位组，则采用primitive形式；          否则采用constructed形式；除最后一个外，其余子部分，每个为1000个八位组。 |         一律采用primitive形式。 |
| **GeneralString** | |
| escape sequences must only be used when the requested character set differs from the usual C0, C1 and G sets (see also the defect report http://www.furniss.co.uk/maint/asn/dr8825 1 005.htm) | |
| **GeneralizedTime或者UTCTime** | |
|         秒不是必须的；秒片段中0有意义；如果没有秒片段，就没有小数点；          必须同步为Universal Time（即以z开头）；          GeneralizedTime类型，小数点必须使用。 | |
| **SEQUENCE或者SET** | |
|         各个成员等于其未编码的值。 | |
| **SET** | |
|         成员按照其Tag升序编码；如果模块声明包含AUTOMATIC TAGS，则按照描述的顺序编码； | |
|         当成员是一个无Tag的CHOICE，它要用其选项中最小Tag来参加排序（静态顺序） |         当成员是一个无Tag的CHOICE，它要用被选择项的Tag来参加培训（动态顺序） |
|         为了避免这种排序步骤，推荐用SEQUENCE替代SET。 | |
| **SET OF** | |
|         按照编码的升序排列：这些编码作为OCTET STRING进行比较，比较需要的话，后面补充空八位组；          为了避免这种动态的排序，推荐用SEQUENCE OF替代SET OF。 | |
| **EXTERNAL、EMBEDDED PDV或者CHARACTER STRING** | |
|         只有当抽象语法包含子类型约束避免传递表示上、下文标识的情况下，才能接力传递，如：  EMBEDDED PDV (WITH COMPONENTS {...,  identification (WITH COMPONENTS { ...,  presentation-context-id ABSENT,  context-negotiation ABSENT })})          和字符抽象语法相关联的字符传输语法注册在{iso standard 10646 level-1(1)}都是规范的；此外，符合注册为level-2(2)或者level-3(3)抽象语法的值也能接力传递。 | |

### 3.2.2 CER

CER比DER出现晚，它的Object Identifier为：{joint-iso-itu-t asn(1) ber-derived(2) canonical-encoding(0)}，Object Description为：”Canonical encoding of a single ASN.1 type”。

CER十分适合使用潜在重要编码的应用中，如ODA（Office Document Architecture）。对编码器，如果编码数据比可用内存空间大，就有必要使用不定长方式，在完成全部编码前开始发送。因为很少有编译器能产生这种形式的编码，所以CER在实际应用中比较少，这也是为什么DER作为缺省规范规则的原因。

### 3.2.3 DER

DER的Object Identifier为：{joint-iso-itu-t asn(1) ber-derived(2) distinguished-encoding(1)}，Object Description为：”Distinguished encoding of a single ASN.1 type”。

DER适合安全数据传输，特别是数字签名方面。在电子商务方面有独特的优势。适合传输平均大小的数据。因为采用定长编码，所以要求编码器有足够的内存空间。

## 3.3 PER

BER编码因其在大小上的开销过大而受人诟病，和真实编码数据相比，平均需要增加50%的额外数据。正式这个原因推动了PER（Packed Encoding Rules）的诞生。相同协议，PER编码与BER相比在大小上至少有40%到60%的改进。因而在VoIP、视频电话、多媒体以及3G等需要高速数据传输的领域有广泛应用。

### 3.3.1 基本规则

PER编码规则的黄金定律为：“obtain the most compact encoding using encoding rules as simple as possible”。

与BER中递归使用三元组TLV<Tag, Length, Value>不同，PER的格式为：’[P][L][V]’ <optional Preamble, optional Length, optional Value>，这里PLV中每个域都不再是八位组串而是比特串。

因为Length可以省略（甚至Value也可以省略），那么就不能从编码中得知边界，所以解码器必须知道抽象描述才能正确解码。PER编码中没有Tag域，因此PER不再缺省支持扩展，必须明确在描述中添加扩展符。

只有当长度没有被SIZE固定或者数据长度很重要的情况下，才对Length进行编码；对SEQUENCE或者SET类型的值编码时，汇总前面增加个bitmap来标识可选成员是否出现；同样，在编码CHOICE的被选择成员前，会增加一个序号指示其位置。

和BER相比，PER使得编解码器处理时间相对要少（但达不到两倍的处理速度），传输速度更快。

### 3.3.2 四种变形

PER编码规则可以分为基本的（Basic）和规范的（Canonical）两类，每一类又可以分为对齐（Aligned）和不对齐（Unaligned）两种。规范形式的优势在CER和DER中已经讨论过了，主要用在中继接力系统和安全系统等使用数字签名的场合。在基本形式中，一个抽象值可能有多个PER编码。但是，在有限测试后，我们知道基本形式编码器要比规范形式编码器速度更快。

对齐方式下，为了保持八位组对齐，可以增加值为0的比特。不对齐方式则编码更为紧凑，但是在编解码时需要花费更多的处理时间。不对齐方式下，不会检查八位组的对齐情况，只在整个数据编码结束后，才进行补位。

注意对齐和不对齐两种方式不能互通，即只能用同种的解码器解码同种编码器编码后的码流。在四种变形中，基本不对齐方式编码是最紧凑的。按照紧凑性降低的顺序，后续依次为：规范不对齐方式，基本对齐方式和规范对齐方式。

在表示层传输上下文协商中，我们需要用Object Identifier指明具体采用的是哪种变形。具体值，请参考第二章中Object Identifier注册树。

### 3.3.3 PER可见子类型约束

为了最大限度的压缩编码，PER需要依赖ASN.1描述中的子类型约束。约束增加的越具体，PER越能得到更优化的编码。而且PER会使用到的约束都是经常使用的约束，这样也使得PER编译器容易实现。此外，这些约束都是编译器在编译过程中“静态”使用的，不会增加实际编解码过程处理时间。

我们称这类约束为PER可见约束（PER-Visible Constraints），它们只包含下表所列：

**Table 3-3** PER可见约束

| **类型** | **PER可见约束** |
| --- | --- |
| BOOLEAN | 无 |
| NULL | 无 |
| INTEGER | 单值约束、值域约束、类型包含约束、约束组合、约束扩展 |
| ENUMERATED | 无 |
| REAL | 无 |
| BIT STRING、OCTET STRING | SIZE约束、约束组合、约束扩展 |
| OBJECT IDENTIFIER | 无 |
| NumericString、PrintableString、VisibleString、ISO646String、  IA5String、UniversalString、  BMPString | FROM约束、SIZE约束、类型包含约束、约束组合、SIZE约束中的扩展 |
| not-known-multiplier character  string types | 无 |
| GeneralizedTime、UTCTime、ObjectDescriptor | 无 |
| open types | 无 |
| SEQUENCE、SET | 无 |
| SEQUENCE OF、SET OF | SIZE、约束组合、约束扩展 |
| CHOICE | 无 |
| EXTERNAL | 无 |
| EMBEDDED PDV、CHARACTER STRING | WITH COMPONENTS |

open types指引用到一个类型域、一个可变类型的值域、或者一个可变类型值集合域。即引用到信息对象类中的类型。

不在上表中的，都不是PER可见约束， PER编译器也就不做相应优化。

### 3.3.4 数的编码

我们考察一个非负数的四种形式的编码，因为自然数常出现在长度域L、bitmap的大小、CHOICE中的序号以及INTEGER类型边界中。

对于INTEGER类型，PER可见约束之一是值域约束。对于有值域约束(bmin..bmax)的值n，如果下边界bmin足够大，PER编码n- bmin的代价就更小。如下：

**Figure 3-26** 有约束自然数的基本编码规则

#### I. 有约束数编码

有约束指值域的上、下边界都有限。如果d=1，即只有一个值，则收、发双方都知道，那么就没有编码的必要。

在对齐方式下：∞

           当2≤d≤255，n- bmin的编码占用log2d个比特。这些比特添加在待发送比特域之后，不进行八位组对齐，不编码L；

           当d=256，n- bmin的编码占用一个八位组，不编码L；

           当257≤d≤65,536，n- bmin的编码占用两个八位组，不编码L；

           当65,537≤d，n- bmin的编码占用log256d个八位组，并且在前面增加L的编码,该L是八位组的个数。

在不对齐方式下：

n- bmin的编码占用log2d个比特，不编码L。

#### II. 半约束数编码

半约束指值域没有上边界（上边界为+∞）。

n- bmin的编码占用log256d个八位组，并且在前面增加L的编码。

#### III. 无约束数编码

无约束指值域没有下边界（即使存在上边界）。

按照BER中整数的编码方式编码，并且在前面增加L的编码。

#### IV. 常见小自然数编码

这种情况经常出现在对表征SEQUENCE、SET类型可选成员的Bitmap长度进行编码时；或者CHOICE类型序号编码时。这种长度相当小，但是却没有一种限定。

           当0≤n≤63时，n以6个比特编码，并且在前面增加一个0比特（八位组不对齐）：

           当64≤n时，n以半约束数方式编码，下边界为0，并且在前面增加一个1比特：

### 3.3.5 长度域编码

与BER中长度域表征编码的八位组数不同，在PER的长度域出现下，如果编码为比特串则表征比特位数；如果编码为八位组串（OCTET STRING和open类型）则表征八位组个数；如果编码为known-multiplier character string则表征字符数；如果是SEQUENCE OF或者SET OF则表征成员个数。

每当ASN.1描述中 对类型通过(SIZE(lmin..lmax))做大小限定时（lmax可以是+∞），长度l的值要按照3.3.4 中的规则编码。作为特例，当lmin =lmax≤65,535时，长度不需要发送，因为解码器知道该长度。

对齐方式下：

           当l是一个bitmap的长度，l-1作为常见小自然数编码；

           当lmax≤65,535，l作为有约束的数编码（约束为(lmin..lmax)）；

           当65,535≤lmax，或者lmax是无穷大：

－ 当l≤127，l以一个八位组编码（八位组对齐），最高比特位为0；

－当128≤l≤16,383，l以两个八位组编码（八位组对齐），最高两个比特位为10；

－当16,384≤l，整个编码以f\*16K为单位分割（f取值为1，2，3或者4）。除最后的片段外，其余每段，长度都以一个八位组编码，最高两个比特位为11。如果编码恰好时16K的整倍数，则在最后补充一个全空的八位组；否则最后一个片段按照前两条进行编码。例如占147,457个单元的编码可以为：

不对齐方式下：

           当l是一个bitmap的长度，l-1作为常见小自然数编码；

           当lmax≤65,535，l-lmin以占用log2(lmin-lmax+1)个比特编码；

           当65,534≤lmax-lmin，或者上边界为无穷大：

－当l≤127，l以八比特编码，最高比特位为0；

－当128≤l≤16,383，l以十六比特编码，最高两个比特位为10；

－当16,384≤l，编码方式域对齐类似，但是不是八位组对齐的。

当类型有可扩展的SIZE约束，并且待发送值不在该约束扩展的根部分，则长度l作为半约束数编码（即lmin=0， lmax =+∞）。

### 3.3.6 各类型编码

I. BOOLEAN

以一个比特编码，1为TRUE，0为FLASE。

直接编码，不编码长度，也不做八位组对齐。

II. NULL

不编码。

如果NULL是作为CHOICE的一个选项或者SEQUENCE、SET可选成员值出现，则已经有bitmap能表征。

III. INTEGER

假设INTEGER类型拥有有效的值域范围(bmin..bmax)，当描述中INTEGER类型有至少一个可扩展的PER可见约束（PER-Visible Constraint），在编码时会在前面增加一个前导（Preamble）比特（不考虑八位组是否对齐）。当INTEGER类型的值在扩展的根部分，该比特为0；否则该比特为1。

在编码n- bmin过程中，如果需要编码L（前面讨论过），则增加长度域L的编码。长度L的约束为(1..lmax)，其中lmax =log2bmax。

如果待发送值n属于约束的扩展部分，则长度按照无约束自然数方式编码，如值：

v INTEGER (3..6, ..., 8..10) ::= 8

的对齐方式编码为：

IV. ENUMERATED

如果该ENUMERATED类型不是可扩展的，则先按照数值大小做升序排列，然后以0为起点，步长为1给每个成员编上序号。对该类型的值编码时，只将序号以值域约束(0..Indexmax)编码。如：

v ENUMERATED {orange(56), green(-2), red(2476)} ::= orange

的PER编码为“01”（因为此时为Indexmax 2，需要两个比特）。

如果该ENUMERATED类型是可扩展的，那么要在编码前增加一个前导（Preamble）比特。当值在扩展的根部分时，该比特为0；否则该比特为1。同时对扩展部分的成员重新进行编号，起点仍为0，步长为1。编码时对值在根部分的情况，就和该类型是不可扩展时一样；值在扩展部分的情况，对序号按照自然数方式编码。如：

v1 ENUMERATED {orange(56), green(-2), red(2476), …, yellow}

::= orange

的PER编码仍然为“01”；而

v2 ENUMERATED {orange(56), green(-2), red(2476), …, yellow, purple}

::= yellow

的PER编码则为“**10**000000”。

V. REAL

对值的编码规则和CER或者DER相同（而且是八位组对齐的），并且在前面增加长度域L的编码。这里L表征的是八位组数。

VI. BIT STRING

如果该BIT STRING类型有可扩展的PER可见大小约束（PER-Visible Size Constraint），那么要在编码前增加一个前导（Preamble）比特。当值在扩展的根部分时，该比特为0；否则该比特为1。

对于有大小约束(SIZE(lmin..lmax))的BIT STRING类型：

           当lmin＝lmax≤16比特，不发送长度，直接编码（不是八位组对齐的）；

           当17≤lmin＝lmax≤65,536比特，不发送长度，直接编码（在八位组对齐方式下是八位组对齐的）；

           当65,537≤lmin＝lmax，长度按照3.3.5 中约定编码；而值按照需要进行分段；

           当lmin≠lmax（或者没有有效的大小约束），长度按照3.3.5 中约定编码；而值按照需要进行分段。

如果BIT STRING类型包含一个命名位置列表，所有结尾的0比特都被去掉；此外，为了满足大小约束，可以增加或者删除结尾0比特来达到最小的长度。

VII. OCTET STRING

与BIT STRING规则相同，不过长度域L表征的是八位组的个数而不是比特数。

VIII. OBJECT IDENTIFIER

值的编码规则与BER相同，前面需要增加长度域L的编码。

IX. RELATIVE-OID

值的编码规则与BER相同，前面需要增加长度域L的编码。

X. 字符串与日期

首先描述known-multiplier字符串类型，如果这种类型是受约束的，带了一个形如（SIZE(lmin , lmax)）的有效的size约束和一个形如（FROM("c1"|"c2"|...|"cn")）的有效的字符约束。如果FROM约束是可扩展的，则有效的字符约束就包括了父类型包含的所有的字符。

如果SIZE约束是可扩展的，则在bit-field上加一个比特。如果串长属于扩展根的范围内，则该比特等于0，否则该比特等于1。记住，如果FROM约束中包含了可扩展的符号，则该约束对于PER不可见，即不会改变扩展比特的取值。

PER压缩known-multiplier字符串类是基于有效的字符约束的。假设n是有效字符约束中字符的个数，b ＝ [log2 n], c = [log2 b],则每个字符在aligned变体中被编码成B = 2 c 个比特（大于b的最小的2的次方），而在unaligned变体中被编码成B = b 个比特。设vmin 和vmax 分别是("c1"|"c2"|...|"cn")对应的最小和最大的字符值。

如果vmax≤ 2B -1, 即所有的系列里的("c1"|"c2"|...|"cn")都可以用B个比特来编码，串中的每个字符都可以作为在区间(vmin .. vmax)之间的一个整数来编码。这就避免了给字符重新排序赋值。否则，字符序列("c1"|"c2"|...|"cn")按照标准[ISO646] 或 [ISO10646-1]中定义的顺序加索引，索引从0开始，每次加1。这个新的索引号以B个比特来编码和发送。

如果字符串的长度是固定的(lmin = lmax)并且小于64K，则长度字段L就不需要了，在aligned变体情况下，只有当 B ×lmax ≥ 17 时才需要octet - aligned。.

如果字符串的长度不固定或者长度虽然固定但大于64K，则长度字段按照章节3.2 编码，然后附着到码流中（在在aligned变体情况下，只有当 B ×lmax ≥ 17 时才需octet - aligned）。

例如，字符串

v IA5String (FROM ("ACGT")^SIZE(3)) ::= "TAG"

被编码成 11 00 10 。而字符串

v IA5String ::= "TAG"

在unaligned变体情况下被编成（注：IA5String的vmin 和vmax分别为0和127）：

在aligned变体情况下被编成（下面的方框表示从整数个字节开始）

日期类型 GerneralizedTime和UTCTime 的编码方式在basic PER情况下与BER相同，在canonical PER情况下与DER相同。

对于不属于known-multiplier的字符串类型，所有的约束对PER都是不可见的。它们的编码方式在basic PER情况下与BER相同，在canonical PER情况下与DER相同。长度字段L作为一个无约束整数加在前面。

对于known-multiplier字符串类型在定义中最小和最大的字符值请参考下表：

**Table 3-4** known-multiplier字符串的最小最大字符值

|  |  |  |
| --- | --- | --- |
| **类型** | **vmin** | **vmax** |
| NumericString | 32 | 57 |
| PrintableString | 32 | 122 |
| VisibleString | 32 | 126 |
| IA5String | 0 | 127 |
| BMPString | 0 | 216 -1 |
| UniversalString | 0 | 232 -1 |

XI. Open类型

一个open类型的值实际上是一个任意类型的值，这个类型对于发送方和接收方而言都是已知的。因此这个值就被编成它的有效类型而没有类型的指示。通常情况下，这个值的类型已经在前面发送过了，解码器会存储类型到关联表中，而后在收到open类型能够的值后根据关联表就可以查询到对应的类型进行解码。

相应的值编码后，不是整数个字节的情况补充值为'0'的比特位以便使该码流为整数个字节（假设为n）。并在头部加上长度字段L表示字节数n，L作为无约束整数进行编码。

XII. SEQUENCE

首先，如果有COMPONENTS OF语句，则由相应的类型进行替换。

如果SEQUENCE类型是可扩展的，则在编码的头部加上一个比特的bit-field，如果SEQUENCE的取值中有属于扩展附加部分的成员，则该比特等于1，否则等于0。

如果SEQUENCE的定义中在扩展根部（extension root）有"n"个成员被置为OPTIONAL或DEFAULT，则在编码头部再添加"n"个比特的bit-field，该bit-field从第一个bit开始，依次指示被标记为OPTIONAL或DEFAULT的成员是否出现。如果为1，则该成员出现，否则没有出现。如果“n”小于64K，则这个 bit-field 应该直接添到码流中。如果"n"大于等于64K， 按照前面提到的处理方法把“n”个bit的bit-field分段并添加到域序列中，前面的长度字段L就作为一个有约束的整数编码，而约束的上限和下限都等于n。

以上为导言（preamble）部分，导言之后是各成员类型的编码，依次出现。

在canonical PER中，对于标记有DEFAULT的成员，如果要编码的值就是缺省值，那么这个成员的编码省略；在basic PER中，如果序列中标记为DEFAULT的成员是一个简单类型，当它的值为缺省值时，它的编码应省略。而对于结构化的成员(SEQUENCE, SET, SEQUENCE OF, SET OF, CHOICE) ，即使它的取值是缺省值时，是否被编码是由发送方来决定的。

如果类型是可扩展的但取值中没有出现扩展附加部分，则该SEQUENCE的值的编码已经结束。

如果类型是可扩展的并且取值中有p个扩展附加部分（双重方括号内的所有成员只当作一个附加扩展部分），则添加一个p个比特的bit-map，对应的比特取值为1则表示该扩展附加部分出现，否则就没有出现。为长度"p"的bit-map增加一个长度指示，编码方式为一个通常的小的非负整数的编码。

以后，依次出现的是各扩展附加部分的编码值。

扩展附加部分的编码方式如下：

           如果该扩展附加部分是单个的成员，则作为open类型来编码，即包括长度字段和整数个字节的值字段。

           如果该扩展附加部分是由双重方括号内的所有成员组成的，则该扩展附加部分的值作为open类型来编码。与上面不同的是：双重方括号内的所有成员的值被当成一个SEQUENCE类型的取值来编码，编码方式参见本节上段。

注意在SEQUENCE类型的值没有包括长度字段，因为根据扩展根部的可选成员对应的bit-map和扩展附加部分对应的bit-map使解码器推断出成员是否存在。

XIII. SET

SET类型的扩展根部（extension root）的成员是按照canonical order来排序的，具体法则如下：

1)         首先按标签类型排序，按照UNIVERSAL，APPLICATION，context-specific，PRIVATE四种类型依次排序，UNIVERSAL标签类型的元素在最前，PRIVATE标签类型的元素在最后；

2)         在标签类型内部，按照标签值的大小排序，小的在前，大的在后。

在排序过程中如果发现某个成员是没有标签的CHOICE类型，则该CHOICE类型的标签值就会等于其所有选项中标签的最小值。在CHOICE类型嵌套的情况下亦是如此。

如果SET类型是可扩展的且SET类型的值中有扩展附加部分，则扩展附加部分的排序按照其协议定义时的顺序，不做改动（因为在新增扩展附加部分时对应的标签必须符合canonical order）。

经过上述的排序后，SET类型的编码就按照SEQUENCE类型的编码。

XIV. SEQUENCE OF

显式PER限制仅仅对SEQUENCE OF类型的成员个数有效。

假设成员数目的上限为"ub"，下限为"lb"。如果没有上限或者上限值ub大于等于64K，就认为ub不确定；如果没有下限，"lb"置为0。假设实际的SEQUENCE OF的成员数量为“n”，编码按如下规则进行：

1)         如果有size约束且该约束可扩展，则在码流上添加一个1比特的bit-field（不需要octet-aligned）。 如果元素的个数属于约束的根部则这个比特等于0，反之该比特等于1。在前一种情况下，调用节3.2为生成的编码增加一个半约束数的长度指示，长度值等于n。

2)         如果成员值的数目固定且小于64K，则没有长度指示，SEQUENCE OF的各成员的值的编码依次填充到生成的编码中去。

3)         其它情况，增加长度指示成员值的数目。如果"ub"存在，则长度指示作为约束数编码，否则作为半约束数类型的整数编码。

需要注意的是在给每个成员编码时，在aligned变体情况下需要octet-aligned。

XV. SET OF

在basic variant情况下，把每个SET类型的元素按顺序编码即可，而不需要调换顺序。而在canonical variant情况下需要先把各SET中的元素按canonical order进行排序再编码，并在必要的时候需要做一些调整，如加上一些'0' bit以形成整数个字节，或加上值为0的字节以使短部分和长的部分长度一致。

XVI. CHOICE

先给CHOICE的各个选项标上索引（index），过程如下：

1)         对于CHOICE类型的根部（root）的选项，先按照canonical order对各选项进行排序，然后给排序后的选项标上索引，第一个选择项索引为0，第二个为1，直至根部的最后一个。 如果嵌套了没有标签的CHOICE类型，则该类型的标签是其择项中的最小的那个标签。

2)         如果CHOICE类型是可扩展的，并且有扩展附加选项，则给扩展附加选项也分别标上索引，第一个扩展附加选项的索引为0，第二个为1，直至扩展部分的最后一个。（标索引之前不需要对这些选项进行排序，是因为在新增扩展附加选项时对应的标签必须符合canonical order）。



在索引标完后，就可以对CHOICE类型的值进行编码，过程如下：

1)         如果仅有一个选择项，如果选择了该选择项的话，不要对索引进行编码。

2)         如果CHOICE类型是可扩展的，则在码流上加一个比特的bit-field（不需要octet-aligned）。如果该值属于扩展附加选项，则 该比特等于1，否则等于0。

3)         如果没有扩展标记，或者有扩展标记但选项在CHOICE类型的根部，则选项的索引值就作为一个INTEGER来编码（约束为0..n，假设n为根部的最大索引值），然后是选项的值的编码。

4)         如果有扩展标记且选项是CHOICE类型的扩展附加选项，则该索引被当作一个通常的小的非负整数进行编码，其"lb"置为0。把选项的值当作open类型进行编码。

XVII. Tagged Type

因为PER不是隐式支持Tag的，所以一个有Tag类型的值按照该类型的编码规则进行编码。在ASN.1模块内，特定Tag上的限制要遵守；建议在模块头的声明中增加AUTOMATIC TAGS，这不会影响已有的限制。

XVIII. EXTERNAL

略。

XIX. INSTANCE OF

一个INSTANCE OF类型的值按照其等效SEQUENCE类型值进行编码。

XX. EMBEDDED PDV或者CHARACTER STRING

如果EMBEDDED PDV或者CHARACTER STRING类型有WITH COMPONENTS约束，该约束限制可选项syntaxes是两个预定义object identifier的序列（即对收、发双方抽象语法、传输语法都明确）；或者限制成员identification选项为fixed，则PER编码中嵌入数据作为OCTET STRING类型编码。

如果成员identification没有按照前面两种方式之一约束，则这两种类型的值需要按照等效的SEQUENCE类型值进行编码。

XXI. Value Set

编码一个有值域Value Set约束类型的值时，值域被当作有该值域约束的类型看待。如值域：

Set1 INTEGER(1..20) ::= {1 | 5 | 7}

被等同作：

Set1 ::= INTEGER(1..20)(1|5|7)

XXII. Information Objects 与 Information Object Sets

信息对象和信息对象集合不编码。传输其中信息的方式，是在类型或者值中引用信息对象或者信息对象集合。编码时，按照所在值的编码规则进行编码。

### 3.3.7 一个完整例子

还是以BER中的例子，来看其对应的PER编码。

ASN.1描述：

MyHTTP DEFINITIONS AUTOMATIC TAGS ::=

BEGIN

GetRequest ::= SEQUENCE

{

header-only BOOLEAN,

lock BOOLEAN,

accept-types AcceptTypes,

url Url,

...

}

AcceptTypes ::= SET

{

standards BIT STRING {html(0), plain-text(1), gif(2),

jpeg(3)} (SIZE (4)) OPTIONAL,

others SEQUENCE OF VisibleString (SIZE (4))

OPTIONAL

}

Url ::= VisibleString (FROM ("a".."z"|"A".."Z"|"0".."9"|"./-\_~%#"))

v GetRequest ::=

{

header-only TRUE,

lock FALSE,

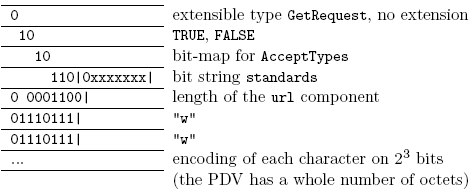
accept-types { standards {html,plain-text} },

url "www.asn1.com"

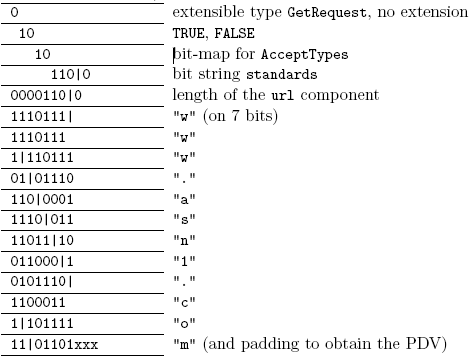
}

END

相应对齐方式的PER编码结果为：



不对齐方式的PER编码结果为：



## 3.4 其它编码规则

### 3.4.1 LWER

Light Weight Encoding Rules (LWER)，1985年有德国发起，1988年由法国继续推进。其目的是为了在BER基础上构造处更为轻量级编码规则，随着PER的出现和成熟，于1997年放弃。

### 3.4.2 BACnet

BACnet （Building Automation and Control Network）是由美国加热、冷冻、空调工程师协会（ASHRAE－American Society of Heating, Refrigerating and Air-conditioning Engineers）设计。用于在Internet上传输从分布在各地，监控集中加热、通风、空调或者烟雾报警器的控制设备来的数据。

现在BACnet编码规则，结合ECN（X.692, Encoding Control Rule），使得可以使用一个通用的工具来生成编解码规程。

BACnet编解码规则具体细节，可以访问网站：

http://www.bacnet.org

或者：

ftp://ftp.bacnet.org/Encoding.doc

### 3.4.3 OER

应用很少，略。

具体信息可以访问：http://www.viggen.com/ntcip/documents/oer.rtf

### 3.4.4 SER

SER（Signaling specific Encoding Rules）是由法电研发部（France Telecom R&D）和诺基亚（Nokia）联合开发的。其主要目的是想通过合适的编译器，为原本不是用ASN.1描述的协议，通过重新描述能自动生成编解码器。这类协议主要是在ASN.1出现之前就有的，如七号信令的协议，GSM接入协议等。

据悉，国内电信设备商H，基于同样目的也设计了类似编码规则。但和上述的工作原理有差异。

因专利保护关系，这些规则的详细内容不得而知。

此外，自1999年后，ASN.1中引入了XML。相应的有了XER（XML Encoding Rules）。