Fine Packet Builder

快速入门

Version 2.0 beta2

2009 年 8 月 29 日

变更履历

版数	项目	章-节-项	变更内容	日期
2.0	1	ALL	简体中文版首版	2009/8/2
2.0	1	1	Beta2 版发布，增加了 HTTP 协议的例子。	2009/8/17
	2	3		
2.0	1	3,4,5,6	新增内容	2009/8/27
	2	1,7	第 1 章说明部分有修改，原第 3 章改为第 7 章。	
2.0	1	2	新增“例子程序的执行”。	2009/8/29

目录

1. 概要

Fine Packet Builder（简称FPB）是开源免费数据包生成与协议分析工具，同时还提供了著名的PCAP[1]数据包截取软件包的Java封装。FPB使用独创的协议语法表示语言（Protocol Syntax Notation）来定义协议的格式与编码和解码规则，可以在基本不编程（或很少编程，如checksum计算）的情况下定义自己的协议。协议语法表示语言是建立在TTCN-3[2]语言的基础上的，它继承了TTCN-3 语言的协议定义相关部分的语法，并且根据协议分析功能的需要进行了扩展。另外通过FPB的通用Java API可以对面向协议栈的数据包进行创建、编辑、编码、解码、完整性验证和链路层发送和接收等操作。

由于 FPB 具有，用协议语法表示语言描述的协议语法直观易懂；Java API 的功能强大且简洁易用；内置协议众多且使用实例丰富；等优点，所以可以将其广泛地应用于流量生成、协议分析、协议模拟、协议学习和协议测试等方面。

- 已经实现（v2.0.beta2 版）的功能列表：

① 通用 CD（编码与解码器）框架。使用协议语法表示语言来描述，并且具有各种自动化功能，其特点如下：

A) 同时支持面向数据包（如 ICMP）和面向流（如 HTTP）两种类型的协议，并且可以使用统一的描述规则。

B) 提供编码规则描述属性集，使得编码和解码器的绝大多数功能可以仅通过书写描述属性就能够实现。

C) 字段的自动计算功能，通过描述属性定义的长度字段和对齐字段可以自动计算自己应有的值。

D) 数据包内容打印功能，可以以树形结构的文本形式打印数据包的内容和各个字段值的描述。

② 数据包内容完整性的验证。使用协议语法表示语言的类型限制条件规则（与 TTCN-3 语言的规则相同）来描述数据的完整性规则，可以通过 Java API 根据该完整性规则来自动验证数据包内数据的完整性。

③ PCAP 软件包的封装。用 Java 封装了 PCAP 软件包，支持所有 PCAP 原有的功能。为方便用户的使用，对 Pcap 原有基于函数的 API 进行了整理和归纳，提供了简洁明了的基于 Java 接口的 API。

④ 已提供的协议 CD 实现：

Ethernet, ARP, ICMPv4, IPv4, ICMPv6, IPv6, UDP, TCP, HTTP

⑤ 提供了多个应用例子，包括用 ARP 协议来取得 MAC 地址，用 ICMP 协议来实现 Ping 和 Traceroute 应用，以及 TCP 的连接开始和终了示例等。

- 正在开发的功能列表：

① 符合 TTCN-3 语言规范的模版功能。

② 协议 CD 实现：SIP, PPPoE, LT2Pv2, L2TPv3 等。

---

[1] Pcap是开源的数据包截取软件包，它在Windows上的实现是WinPcap。具体内容可以参考 www.winpcap.org 和 www.tcpdump.org。

[2] TTCN-3 是ETSI制定的通用协议测试语言标准，具体内容可以参考http://www.ttcn-3.org/ 和 http://www.ttcntest.com/。

本工具希望提供一个灵活、使用简便而且功能强大的 CD 框架，从而可以快速开发各种类型的 CD，并且用它实现尽可能多协议的 CD 供大家使用，也欢迎大家用它实现自己的协议并共享给别人使用。

以下是相关网站的链接：

FPB开源项目网站	http://code.google.com/p/fpb/
WinPcap官方网站	www.winpcap.org
Pcap官方网站	www.tcpdump.org
TTCN-3 官方网站	http://www.ttcn-3.org/
TTCN-3 中国社区网站	http://www.ttcntest.com/

2. 安装

Fine Packet Builder 的安装文件分为执行库和源程序两种形式，它们都是 ZIP 压缩文件形式，解压缩后就可以直接使用了。执行库文件里包括 FPB 执行所需要的环境，比如 Java 库文件等，文件名称里包括了版本信息和发布的时间，比如 fpb_2.0.beta1.v200907311552.zip。源程序文件里是 FPB 库本身的开发和调试所需的源程序和 Eclipse 工程目录。

(1) 执行库安装文件的安装和目录结构

执行库安装文件解压缩后就可以直接使用，其文件目录如下：

```
+ fpb_2.0.beta1                安装根目录
  |- doc              JavaDoc 形式的 Java API 说明
  |- lib  执行所需的第三方库文件
    |- antlr-runtime-3.1.3.jar
    |- …
  |-protocol          协议定义模块
    |- ArpProtocol.module
    |- …
  |- sample           FPB 使用例子的 Java 源程序。
  |- finepbuilder_2.0.beta1.v200907311514.jar      FPB 的 Java 类库文件
  |- fpblib.dll                                    FPB 的本地动态库文件
  |- fpbprotocols_2.0.beta1.v200907311528.jar      protocol 目录内协议定义模块所使用的功能扩充类库。
```

(2) 源程序安装文件的安装和目录结构

解压缩源程序安装文件后将内部的 Eclipse 工程文件导入到 Eclipse 开发环境后就可以使用了，文件名则类似于 fpb_2.0.beta1_src.v200907311559.zip。另外由于 com.fineqt.fpb.lib 项目里使用了目录链接，所以需要在在 Eclipse 的 Preferences->General->Workspace->Linked Resource 下加入如下变量

FPB_LIB_MODEL_PROJECT：D:\work\fpbdistworkspace\com.fineqt.fpb.lib.model

解压缩后的目录结构如下：

```
+
|- com.fineqt.fpb.lib          主工程目录
|- com.fineqt.fpb.lib.model            模型工程目录
|- com.fineqt.fpb.protocol 协议模块扩展功能工程目录
```

(3) FPB 执行环境的安装和设置

因为本工具是以 Java 语言来开发的，所以需要事先安装 Java Runtime。链路层通信和数据包截取文件相关联的功能使用了 WinPcap 库，所以 WinPcap 的安装也是需要的。下面是安装方法的说明。

① Java Runtime 的安装

JRE(Java Runtime Enviroment)(1.6 以上)的安装文件和安装说明可从以下链接下载。

http://java.sun.com/javase/ja/6/download.html

② WinPcap 的安装

WinPcap（3.1 以上）的下载链接和安装说明在以下的场所。

http://www.winpcap.org/install/default.htm

③ FPB 运行库的安装

FPB 运行库文件可从如下场所下载，下载后直接解压缩即可。

http://code.google.com/p/fpb/downloads/list

④ 环境参数的设置

在执行时 FPB 需要知道安装目录的位置从而载入本地动态库(.dll 文件)和默认协议定义模块文件，所以如果没有在程序中强制指定的话，就要设置 FPB_HOME 环境变量来告知 FPB 运行库。Windows 下的执行命令如下：

set FPB_HOME=D:\tools\fpb_2.0.beta1

(4) 例子程序的执行

在运行库安装目录下 sample 目录里有多个 FPB API 和协议模块的使用例源程序，清单如下：

+ sample\com\fineqt\fpb\protocol\sample
|- ArpApp.java 通过 Arp 协议来取得设置为特定 IP 地址的机器的 MAC(物理)地址。
|- PingApp.java 使用 ICMPv4 协议的 Echo Request 和 Echo Reply 报文来实现 Ping 应用。
|- TracerouteApp.java 使用 ICMPv4 协议的 Echo Request 和 Echo Reply 报文和 IPv4 协议的 TTL 字段来实现 Traceroute 应用。
|- TcpConnectionSample.java 模拟 TCP 协议连接的三向握手开始和终了。
|- SampleMain.java 用于执行各个例子程序的主程序类。
|- HttpGetSample.java 使用 HTTP 协议的 GET 命令来取得网站的资源。

执行例子程序前先要按照“2(3)”的说明准备好运行环境，接着通过Eclipse[3]等Java开发工具建立Java项目，然后在项目中引入安装目录下”lib”目录下的所有第三方库和finepbuilder_*.jar和fpbprotocols_*.jar两个主库文件后就可以执行例子程序的主程序com.fineqt.fpb.protocol.sample. SampleMain类了。另外由于例子中使用了本地IP地址和物理地址（MAC地址）等参数，在执行前需要修改一下SampleMain.java文件中的相关设置，SampleMain.java文件的说明具体可以参考“4.2(2)(a)”，各个例子的说明可以具体参考“7”。

如果希望不通过Eclipse等开发工具就直接执行例子程序则可以使用安装目录下的“runsample.xml” ANT[4] 批处理文件，该文件首先将所有的例子程序用javac编译后，然后直接用java命令来执行com.fineqt.fpb.protocol.sample. SampleMain类。Windows下的执行命令如下，并且执行时当前目录必须是FPB的安装目录。

ant -v -f runsample.xml

另外，由于使用了 javac 命令，所以 JAVA_HOME 环境变量需要指向 JDK（不是 JRE）的安装目录，并且 ant 命令也需要在 OS 的检索路径（PATH 环境变量）中可见。还有 SampleMain.java 使用的是 UTF8 编码，编辑时需要使用支持 UTF8 的编辑器（比如 Eclipse）。

[3] Eclipse是最为流行的用于开发Java的IDE，可见http://www.eclipse.org/。
[4] ANT是Java环境下最为流行的编译和批处理工具，可见http://ant.apache.org/。

3. 系统构成

FPB主要由三部分构成，分别为协议模块模型类库、运行库和内置协议定义。协议模块模型类库是为了描述协议语法表示语言（Protocol Syntax Notation,简称PSN）的结构而用EMF[5]定义的模型类库，定义协议的模块文件的内存模型就使用该模型类库来构建。运行库是FPB的主体，它包括文件编译器、通用CD架构和Pcap通信等功能模块。内置协议定义由根据PSN所描述的协议模块文件和协议编码扩展类库两部分构成。

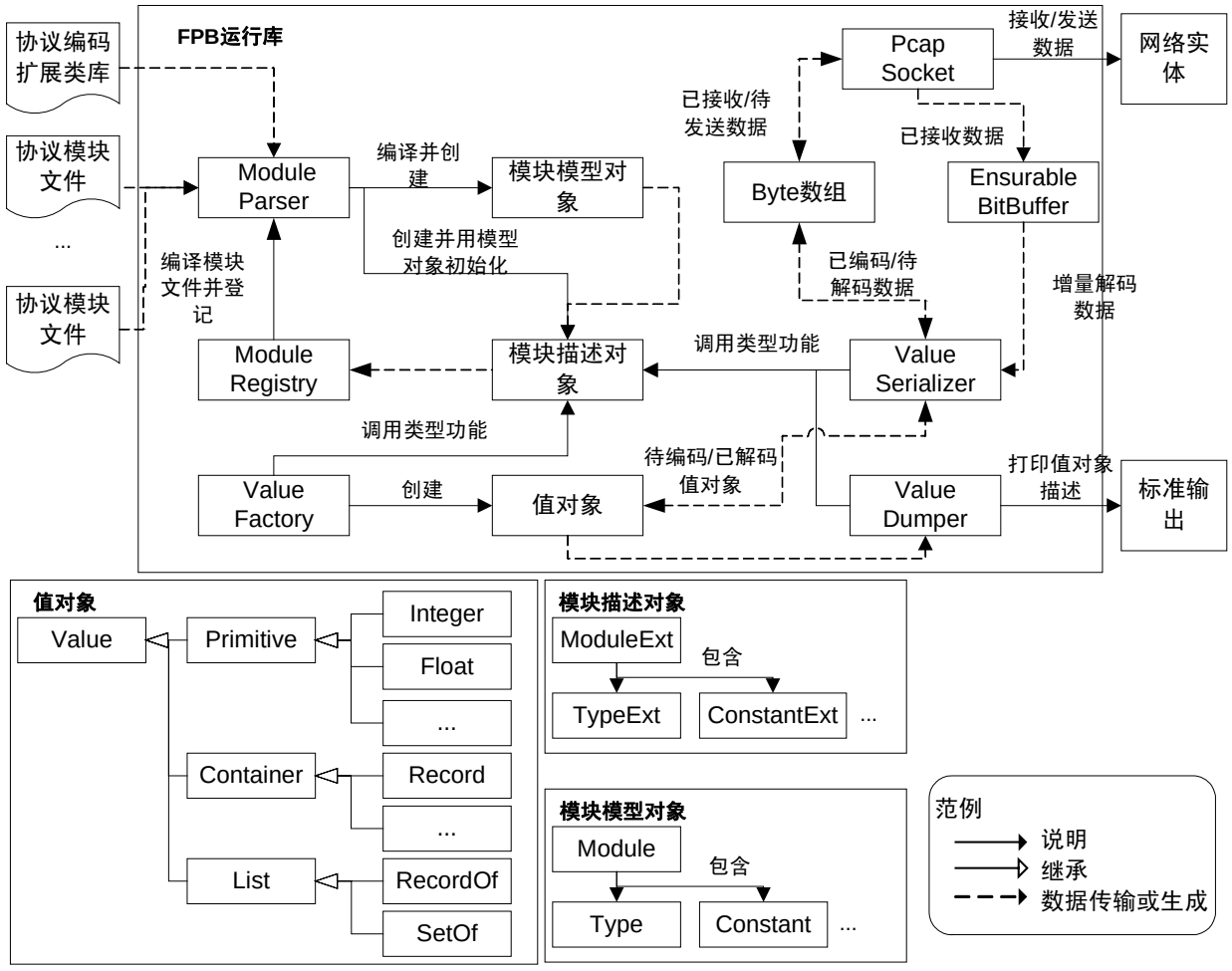


图 3-1 FPB 系统构成图

图 3-1 描述了FPB的系统构成概况。系统的三大构成部分中，模块模型由“模块模型对象”来表示，运行库由“FPB运行库”来表示，内置协议定义由“模块文件”和“协议编码扩展类库”来表示。图中“模块描述对象”内的元素与“模块模型对象”内的元素一一对应，模型对象只具有模块内元素的数据模型，而编码和解码等具体功能则包含在描述对象内，所以解码、编码、数据完整性验证以及值对象的创建和文本描述的取得等功能最终都要通过“模块描述对象”来完成。图中“值对象”与模块文件中所能描述的值类型的种类相匹配，用来存储协议中字段的具体值。

FPB中的协议定义以协议模块为单位来管理，在系统启动时并不将所有的协议模块载入，只是将所有搜索路径上的合法（.module为扩展名）协议模块文件以Proxy形式在Module Registry中登记一下，在实际

使用时才将完整内容载入内存，这就是所谓的“延迟载入技术”。比如在创建协议模块文件中所描述类型的值对象时，首先通过Module Registry来取得Value Factory，Value Factory与协议模块文件是一一对应的，当该协议模块还没有载入时Module Registry则会调用Module Parser(通过ANTLR[6]编译类库)来编译外部的协议模块文件，然后从得到的ModuleExt(模块描述对象)哪里取得Value Factory；接着用Value Factory来创建值对象。另外，编码和解码操作通过Value Serializer来完成，值对象的文本描述打印通过Value Dumper来完成，链路层的通信则通过Pcap Socket来完成。还需特别指出，针对HTTP协议等基于流协议的解码FPB提供了增量解码的功能，也就是将非连续的数据存入Ensurable Bit Buffer中，Value Serializer则通过该Buffer对象来取得数据并用阻塞方式完成面向流的解码。

[5] EMF是Eclipse Model Framework的缩写，用来定义通用软件模型的架构。
[6] ANTLR是Another Tool for Language Recognition的缩写，是开发编译器的辅助工具。

4. 一个完整的协议定义与 API 使用例

本章以 Ethernet 协议为例说明协议模块文件的结构和书写规则，用 ARP 协议为例说明 FPB 的 Java API 的使用方法，力求通过本章使读者对 FPB 的使用方法有一个整体的了解。

4.1. Ethernet 协议的定义

所谓协议就是控制网络中不同实体（比如计算机和路由器）间数据通信的规则。协议定义要传送什么，怎样进行通信，以及何时进行通信。协议格式就是协议互相传送数据的结构或格式，而数据包或帧等（后统称为数据包）是通信中传送数据的单位，通常由一个或多个协议所构成。要用协议来进行通信的话，那就离不开根据各协议格式来生成数据包、解析数据包了。FPB 使用独有的协议语法表示语言（PSN）来描述协议的格式以及数据包的编码、解码、数据完整性验证和描述文本表示等规则。下面以 Ethernet（以太网）协议为例进行详细的说明。

首先我们看一下 Ethernet 协议的格式。

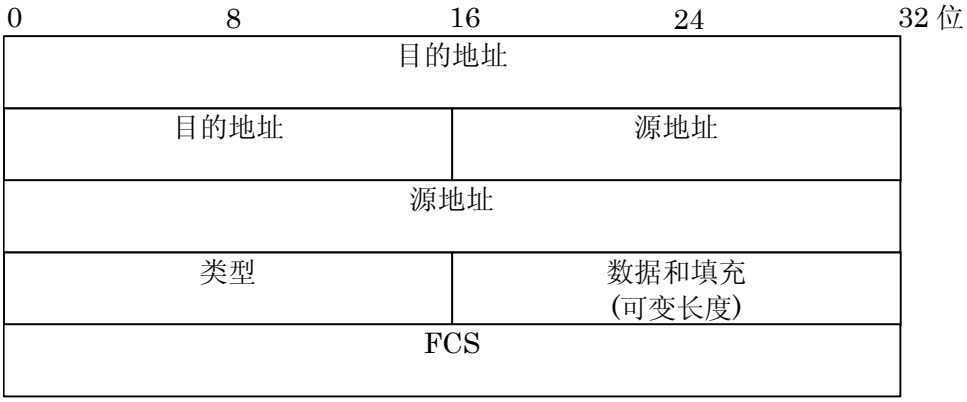


図 4- 1 Ethernet 协议格式

上图各字段的内容：

- **目的地址** 6 字节字段。下一节点的物理地址（MAC 地址）。
- **源地址** 6 字节字段。前一节点的物理地址（MAC 地址）。
- **类型** 2 字节字段。上层协议的类型。比如 IP 是 0x0800，ARP 是 0x0806。
- **数据** 数据部分。最小长度 46 字节，最大长度 1500 字节。
- **FCS** 4 字节字段。差错检测信息，使用的是 CRC-32 算法。

接下来看一下 Ethernet 协议的协议模块文件。

```
module EtherProtocol {                                    ①
    import from BasicTypeAndValues {                      ②
        type Oct6, UInt16, Oct4, MacAddress
    }
    import from Ipv4Protocol {
        type Ipv4Protocol
    }
    import from ArpProtocol {
        type ArpProtocol
    }
    import from GlobalEnumSets {
        enumset EtherTypes
```

```
}
import from Ipv6Protocol {
        type Ipv6Protocol
}

type record EtherProtocol {
        MacAddress dstAddress,
        MacAddress srcAddress,
        UInt16 etherType,
        EtherPayload payload,
        Oct4 fcs optional
} with {
        variant Protocol(true); PushEmptyField([DECODE], "etherType")
        variant(etherType) SetField([DECODE])
        enumref(etherType) EtherTypes
}

type union EtherPayload {
        Ipv4Protocol ipv4,
        ArpProtocol arp,
        Ipv6Protocol ipv6,
        octetstring data
} with {
        variant CaseRef("-etherType", #UInt16); CaseDefault(data)
        variant(ipv4) CaseCond(UInt16 (0x0800))
        variant(arp) CaseCond(UInt16 (0x0806))
        variant(ipv6) CaseCond(UInt16 (0x86DD))
}

} with {
        encode "FPB"
        variant ByteOrder(BIG_ENDIAN)
}
```

④ ③ ⑤ ⑥ ⑦ ⑧

表 4.1-1 描述 Ethernet 协议的协议模块文件

① 协议模块的定义。用 module 语句来定义模块名称为 EtherProtocol 的模块，协议模块里可以定义类型、模板、常量、模块参数和枚举标签集合等元素。模块是顶级元素，一个文件内只能有一个模块，其他元素都需要在模块内部定义。

② 引入别的模块的元素。用 import 语句引入了 BasicTypeAndValues 协议模块里所定义的 Oct6、UInt16、Oct4、MacAddress 四种值类型。为了使用别的模块里定义的值类型等元素，必须事先用 import 语句引入它们。可以使用原模块内的名称来引用引入类型等引入元素，当引入元素之间或与本模块元素之间的名称发生冲突时，可以使用被引入模块名加上“.”的前缀来区分它们。

③ 记录（record）类型定义。用 type 语句来定义名称为 EtherProtocol 的记录类型。记录类型是容器类型的一种，它可以包含复数个字段，各个字段在协议格式中是顺序排列的关系。

④ 字段定义。定义类型为 MacAddress，名称为 dstAddress 的字段。该字段的附加属性可以通过尾部的 with 语句来定义。

⑤ EtherProtocol 类型的描述属性的定义。使用 with 语句来定义类型和字段的描述属性。描述属性是对于类型或字段的扩展描述，在 FPB 中可以有 encode(编码)、variant(编码变量)、enum(枚举标签集合)、enumref(枚举标签集合引用)和 runtime（运行时设置）五种类型。“variant”等关键字之

后没有括号表示该描述属性针对整个类型，有括号则针对括号内指定的字段。“variant Protocol(true); PushEmptyField([DECODE], "etherType")”语句表示 EtherProtocol 类型是协议的根类型，并且在解码（decode）时在堆栈中设置一个名为“etherType”的堆栈字段。“variant(etherType) SetField([DECODE])”语句表示在解码“etherType”字段后，将该字段的值设置到同名的堆栈字段中，即前面 PushEmptyField 设置的堆栈字段。“enumref(etherType) EtherTypes”语句表示引用名为“EtherTypes”的枚举标签集合来解释“etherType”字段的文本描述，该文本描述将用于数据包的文本内容打印(通过 Value Dumper)。

⑥ 联合(union)类型的定义。用 type 语句来定义名称为 EtherPayload 的联合类型。联合类型与记录类型一样都是容器类型的一种，它可以定义被包含的字段，不过与记录类型不同的是联合类型同时只能有一个字段有效。

⑦ EtherPayload 类型的描述属性的定义。“variant CaseRef("-etherType", #UInt16); CaseDefault(data)”语句声明在解码时，EtherPayload 类型将引用“etherType”堆栈字段的值来判断那个子字段有效，当没有找到有效的子字段时将以“data”字段为默认有效字段。“variant(ipv4) CaseCond(UInt16 (0x0800))”语句声明在解码时，如果 CaseRef 指定的引用字段（”etherType”堆栈字段）值是 UInt16 类型的 0x0800，则“ipv4”字段为有效子字段。

⑧ EtherProtocol模块级别描述属性的定义。“encode "FPB""语句表示该模块内的默认编码类型为“FPB”。“ variant ByteOrder(BIG_ENDIAN)”语句表示该模块内部类型的默认字节顺序[7]将是BIG_ENDIAN类型。

上述协议模块文件内的的每个字段是和协议格式相对应的，下面给出它的对应关系。

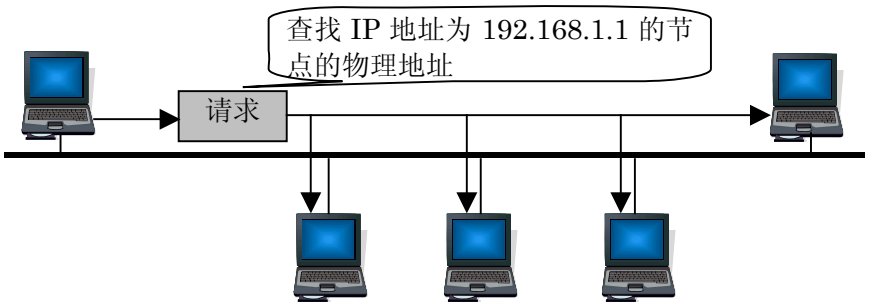
类型	类型字段	字段类型	对应协议格式字段
EtherProtocol	dstAddress	MacAddress	目的地址，MAC 地址。
	srcAddress	MacAddress	源地址，MAC 地址。
	etherType	UInt6	类型，无符号 16 位整数。
	payload	EtherPayload	数据
	fcs	Oct4	FCS
EtherPayload	ipv4	Ipv4Protocol	上位协议是 Ipv4 时有效
	arp	ArpProtocol	上位协议是 Arp 时有效
	ipv6	Ipv6Protocol	上位协议是 Ipv6 时有效
	data	octetstring	其他情况下有效

表 4.1-2 协议模块内字段和协议格式字段的对应关系

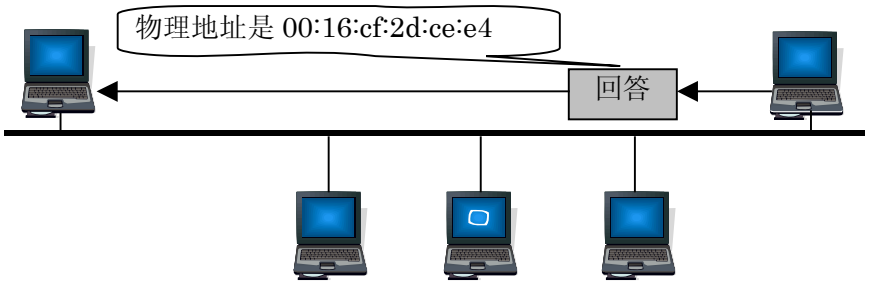
[7] 字节顺序表示编码和解码时按什么顺序来解释多字节的整数或者浮点数。比如说BIG_ENDIAN情况下对于四个字节的无符号整数 0x04030201，将按照从高到低的字节顺序来解释，即 0x04 -> 0x03 -> 0x02 -> 0x01。如果是LITTLE_ENDIAN,前述情况将按 0x01 -> 0x02 -> 0x03 -> 0x4 顺序来解释。

4.2. ARP 协议模拟

以太网（局域网）内两台计算机或设备互相通信的时候，需要知道对方的MAC地址（物理地址）才能将数据通过以太网协议传送给对方。但往往通信的时候只知道对方的IP地址（逻辑地址），那我们是怎样和对方打交道的呢？这就是ARP的功劳了，ARP的工作就是通过IP地址来查询MAC地址。那么看一下ARP是怎么工作的（図 4- 2 ）。



A)ARP 请求用广播发送



B)ARP 回答用单播发送

図 4- 2 ARP 的工作原理

ARP 发送广播请求数据包，向同一网上所有设备询问特定 IP 地址相对应的 MAC 地址。其它设备接到 ARP 请求数据包后，先看一下请求的 IP 地址是不是自己，不是的话就丢弃它，是的话就将自己的 MAC 地址填入 ARP 回答数据包，并将其用单播数据包发回请求方。这样，请求方就知道了通信对方的 MAC 地址了。

(1) 协议格式

那么，ARP数据包里都放些什么信息呢？那就来看一下ARP协议的协议格式（図 4- 3 ）。

ARP 数据包具有以下的一些字段：

- **硬件类型** 这是 16 位字段，用来定义运行 ARP 的网络的类型。每一个局域网基于其类型被指派给一个整数。例如，以太网是类型 1。ARP 可用在任何网络上。
- **协议类型** 这是 16 位字段，用来定义协议的类型。例如，对于IPv4 协议，该字段的值是 0x0800[8]。ARP可用于任何高层协议。
- **硬件长度** 这是 8 位字段，用来定义以字节为单位的物理地址长度。例如，对以太网是 6。
- **协议长度** 这是 8 位字段，用来定义以字节为单位的逻辑地址长度。例如，IPv4 是 4。
- **操作** 这是 16 位字段，用来定义数据包的类型。已定义了两种类型：ARP 请求（1），ARP 回答（2）。
- **发送端硬件地址** 这是可变长度字段。用来填写发送端物理地址。长度由“硬件长度”字段决定。

- **发送端协议地址** 这是可变长度字段。用来填写发送端逻辑地址。长度由“协议长度”字段决定。
- **目标硬件地址** 这是可变长度字段。用来填写目标物理地址。长度由“硬件长度”字段决定。
- **目标协议地址** 这是可变长度字段。用来填写目标端逻辑地址。长度由“协议长度”字段决定。

図 4- 3 ARP 协议的协议格式

ARP 协议的协议模块文件是“ArpProtocol.module”，该文件在 FPB 的安装目录里可以找到。下面给出协议模块文件内类型和字段的说明，以及与协议格式原来字段的对应关系。

类型	类型字段	字段类型	对应协议格式字段
ArpProtocol	hwType	UInt16	硬件类型。
	protocolType	UInt16	协议类型，0x0800 为 IPv4，0x86DD 为 IPv6
	hlen	UInt8	硬件长度。
	plen	UInt8	协议长度。
	operation	UInt16	操作。
	senderHwAddress	octetstring	发送端硬件地址
	senderProtocolAddress	octetstring	发送端协议地址
	targetHwAddress	octetstring	目标硬件地址
	targetProtocolAddress	octetstring	目标协议地址

(2) 协议客户端模拟实现

接着来说明如何用 Java API 来实现 ARP 客户端的模拟。也就是，向客户端所在的局域网所有的设备询问特定 IP 地址对应的 MAC 地址。具体来说，先将请求的 IP 地址填入 ARP 请求数据包并将其用广播形式发送，然后接收以单播形式发回来的 ARP 回答数据包并从中取出对方的 MAC 地址，如果等待一定时间回答数据包仍然没有到达的话就认为超时。

在 FPB 安装目录下 sample 目录里有多个 FPB API 和协议模块的使用例源程序，其中 ARP 客户端模拟的主体程序是 com.fineqt.fpb.protocol.sample.ArpAp 类，ArpApp 类的使用则在相同包的 SampleMain 类里。SampleMain 类内包含有主函数 mian()，所以实验时只要直接执行该类就可以了。

(a) SampleMain 类及 ArpApp 类使用例的源程序及说明

[8] 0x作为前缀的数字是以 16 进制来表示。

```
1   public class SampleMain {
2           //本地的物理地址
3           private static final String LOCAL_MAC = "00:16:D4:17:25:C4";
4           //本地的 IP 地址
5           private static final String LOCAL_IP = "192.168.11.5";
6           //FPB 的安装根目录
7           private static final String FPB_HOME = "D:/work/ttcnworkspace/com.fineqt.fpb.lib";
8           //FPB 协议定义文件所在的目录，通常是 FPB 的安装根目录下的 protocol 目录。
9           private static final String FPB_PROTOCOL_FOLDER =
10                  "D:/work/ttcnworkspace/com.fineqt.fpb.protocol/protocol";
11          //网络接口数组
12          private static NetworkInterface[] ifs;
13          //默认使用网络接口序号
14          final static int INTERFACE_INDEX = 1;
15
16          public static void main(String[] args) throws Exception {
17                  //FPB 系统初始化
18                  IFpbSystem.INSTANCE.init(FPB_HOME, FPB_PROTOCOL_FOLDER);
19                  SampleMain main = new SampleMain();
20                  //Arp
21                  main.testArp();
22    . . .
23          }
24          public void testArp() throws Exception {
25                  String dstIp = "192.168.11.1";
26                  //生成 Pcap 套接字来实现链路层通信
27                  IPcapSocket socket = IPcapSocketFactory.INSTANCE.createPcapSocket();
28                  socket.setDevice(getInterface().name);
29                  //生成 Arp 对象
30                  ArpApp arp = new ArpApp(socket, LOCAL_MAC, LOCAL_IP);
31                  //设置是否打印数据通信内容
32                  arp.setDumpPacket(true);
33                  //取得目标设备的 MAC 地址
34                  String dstMac = arp.askMac(dstIp);
35                  //打印
36                  System.out.println("Mac for " + dstIp + " is " + dstMac + ".");
37          }
38
39  . . .
40
41          public static NetworkInterface getInterface() throws SocketException {
42                  if (ifs == null) {
43                          ifs = IPcapSocketFactory.INSTANCE.getDeviceList();
44                  }
45                  return ifs[INTERFACE_INDEX];
46          }
47
48  }
```

表 4.2-1 SampleMain 类

3 行　　本地物理(MAC)地址的定义，Windows 下可以用“ipconfig /all”命令取得。

5 行　本地 IP(v4)地址的定义，Windows 下可以用“ipconfig /all”命令取得。

6-10 行　定义 FPB 的安装目录及协议模块文件所在目录。安装目录是 FPB 安装文件解压缩后所在的目录，而协议模块文件目录则是安装目录下的“protocol”目录。

11-14 行　定义本机所有的网络接口描述对象数组，并指定使用哪个接口来进行通信。由于需要使用链路层通信，所以通常指定物理上存在的网卡作为通信接口。可以将“ifs”数组内的所有接口都打印出来，然后与“ipconfig /all”命令中列出的设备比对一下再决定使用哪个接口。

16-21 行　主函数。

24-37 行　ArpApp 类执行函数。

25 行　目标设备的 IP 地址，请求的就是 IP 地址与它相同的节点的 MAC 地址。

27 行　创建 Pcap 套接字。

28 行　设置通信所使用的网络设备（网卡）名称。

30 行　创建 ArpApp 对象。LOCAL_MAC 和 LOCAL_IP 是常量，定义本地 MAC 和 IP 地址。

32 行　设置是否打印数据通信内容

34 行　执行 Arp 请求通信，取得目标设备的 MAC 地址。

36 行　打印执行结果。

41-46 行　取得本机的所有网络接口对象数组，并选择指定接口返回。

(b) SampleMain 类内 ArpApp 类使用例的执行结果例

```
Mac for 192.168.11.1 is 00:16:01:15:A4:88
```

(c) Arp 协议通信报文的打印例

- Arp Request 报文

```
0:42   EtherProtocol.EtherProtocol
0:6      dstAddress              FF:FF:FF:FF:FF:FF
6:6      srcAddress              00:16:D4:17:25:C4
12:2     etherType               2054 ARP
14:28    payload
14:28      arp                   ArpProtocol.ArpProtocol
14:28 ArpProtocol.ArpProtocol
14:2     hwType                   6 IEEE 802 Networks
16:2     protocolType            2048 IPv4
18:1     hlen                     6 MAC
19:1     plen                     4 IPv4
20:2     operation                1 Arp Request
22:6     senderHwAddress           '0016D41725C4'O
28:4     senderProtocolAddress 'C0A80B05'O
32:6     targetHwAddress           '000000000000'O
38:4     targetProtocolAddress 'C0A80B01'O
42:0     fcs                     omit
```

- Arp Response 报文

```
0:46   EtherProtocol.EtherProtocol
0:6      dstAddress              00:16:D4:17:25:C4
6:6      srcAddress              00:16:01:15:A4:88
12:2     etherType               2054 ARP
```

```
14:28    payload
14:28      arp                      ArpProtocol.ArpProtocol
14:28 ArpProtocol.ArpProtocol
14:2     hwType                     1 Ethernet - 10Mb
16:2     protocolType               2048 IPv4
18:1     hlen                       6 MAC
19:1     plen                       4 IPv4
20:2     operation                  2 Arp Response
22:6     senderHwAddress            '00160115A488'O
28:4     senderProtocolAddress      'C0A80B01'O
32:6     targetHwAddress            '0016D41725C4'O
38:4     targetProtocolAddress      'C0A80B05'O
42:4     fcs                        '00000000'O
```

(d) ArpApp 类的源程序及说明

```
。。。
public class ArpApp {
    private static IModule module = IModuleRegistry.INSTANCE
            .resolveModule("EtherProtocol");
    private static IFactory factory = module.getFactory();
    private static IValueSerializer ser = IValueSerializer.INSTANCE;
    private String srcMac;
    private String srcIp;
    private IPcapSocket socket;
    private int timeout = ICommonSocket.DEFAULT_TIMEOUT;
    private boolean dumpPacket;

    /**
     * 构造函数
     * @param socket Pcap 套接字。
     * @param srcMac 源设备的 MAC 地址。
     * @param srcIp 源设备的 IP 地址。
     */
    public ArpApp(IPcapSocket socket, String srcMac, String srcIp) {
        this.socket = socket;
        this.srcMac = srcMac;
        this.srcIp = srcIp;
    }

    /**
     * 生成 ARP 请求报文。协议栈为 Ehternet/Arp.
     * @param dstIp
     * @return
     */
    private IRecordSetValue createArpRequest(String dstIp) throws MetaException {
        // Ethernet
        IRecordSetValue etherPrtl = (IRecordSetValue) factory
                .createValue("EtherProtocol");
        etherPrtl.getField("dstAddress").setTextAs("ff:ff:ff:ff:ff:ff",
                IValueTextStyle.MAC_ADDRESS);
        etherPrtl.getField("srcAddress").setTextAs(srcMac, IValueTextStyle.MAC_ADDRESS);
        etherPrtl.getField("etherType").setText("0x0806");
```

```
38          // Arp
39          IRecordSetValue arpPrtl = (IRecordSetValue) etherPrtl.findField("payload/arp",
40                  true);
41          arpPrtl.getField("operation").setText("1");
42          arpPrtl.getField("senderHwAddress")
43                  .setTextAs(srcMac, IValueTextStyle.MAC_ADDRESS);
44          arpPrtl.getField("senderProtocolAddress").setTextAs(srcIp,
45                  IValueTextStyle.IPV4_ADDRESS);
46          arpPrtl.getField("targetHwAddress").setTextAs("00:00:00:00:00:00",
47                  IValueTextStyle.MAC_ADDRESS);
48          arpPrtl.getField("targetProtocolAddress").setTextAs(dstIp,
49                  IValueTextStyle.IPV4_ADDRESS);
50          return etherPrtl;
51      }
52
53      /**
54       * 根据目标设备的 IP 地址来取得目标设备的 MAC 地址。
55       * @param dstIp 目标设备的 IP 地址。
56       * @return 目标设备的 MAC 地址文本描述。
57       * @throws Exception
58       */
59      public String askMac(String dstIp) throws Exception {
60          byte[] resData = new byte[1514];
61          socket.setTimeout(timeout);
62          // 打开套接字
63          socket.open();
64          try {
65              // 设置 Pcap 套接字的过滤条件。（过滤条件的书写规则可以参考 WinPcap 的用户手册）
66              socket.setFilter("ether dst " + srcMac + " and arp", true);
67              // 生成 ARP 请求报文
68              IRecordSetValue arpRequest = createArpRequest(dstIp);
69              // 对报文数据包内的可计算字段进行自动计算（这里是 Arp 协议的 Hlen 和 Plen 字段）
70              ser.calculate(arpRequest);
71              if (dumpPacket) {
72                  IValueDumper.INSTANCE.dump(arpRequest);
73              }
74              // 编码
75              byte data[] = ser.encode(arpRequest, false);
76              // 发送编码数据
77              socket.write(data);
78              // 接收应答数据
79              int len = socket.read(resData);
80              // 解码（底层协议为 Ethernet 所以使用 EtherProtocol 作为推测类型）
81              IRecordSetValue arpResponse = (IRecordSetValue) ProtocolUtils.decode(resData,
82                      0, len, module.getType("EtherProtocol"));
83              // 打印解码后的 ARP 应答报文对象
84              if (dumpPacket) {
85                  IValueDumper.INSTANCE.dump(arpResponse);
86              }
87              // 从 EthernetProtocol 的 Payload 字段取得 ARP 协议字段
88              IRecordSetValue arpPrtl = (IRecordSetValue) arpResponse
89                      .findField("payload/arp");
90              // 取得 ARP 协议的 SenderHwAddress 从而得到目标设备的 MAC 地址
91              return arpPrtl.getField("senderHwAddress").getTextAs(
```

```
92                          IValueTextStyle.MAC_ADDRESS);
93              } finally {
94                  // 关闭套接字
95                  socket.close();
96              }
97          }
98
99      。。。
100
101     }
```

表 4.2-2 ArpApp 类

3-4 行 用模块登记器（IModuleRegistry）取得以太网协议（EtherProtocol）。

5 行 取得值工厂对象。

6 行 取得值对象串行化器对象。

7 行 发送端 MAC 地址属性。

8 行 发送端 IP 地址属性。

9 行 Pcap 套接字对象，用于链路层通信。

10 行 超时时间。

11 行 是否打印通信数据包。

19-23 行 构造函数。

30-51 行 创建 Arp 请求报文值对象函数的定义。

32-32 行 用值工厂对象生成类型为 EtherProtocol 的值对象。

34-35 行 对于 EtherProtocol 的 dstAddress 字段，以文本方式设置 MAC 地址。值对象的“getField”方法用来取得容器值内的字段，“setTextAs”方法则已指定的文本样式设置字段值，这里的文本样式是 MAC_ADDRESS 即 MAC 地址。

37 行 用 setText 方法以文本形式设置字段的值。

39 行 用值对象的findField方法按照指定路径取得对应字段。参数“payload/arp”表示路径为EtherProtocol类型下的payload字段下的arp字段。参数“true”表示指定路径上的字段为空或者omit[9]时用默认值填充。

59-97 行 执行 Arp 请求函数的定义。

60 行 定义用于存放通信数据的字节数组。

61 行 设置 Pcap 套接字的超时时间。

63 行 打开套接字。

66 行 设置Pcap套接字的过滤条件。因为FPB底层使用WinPcap[10] 来通信，所以过滤条件需按照WinPcap的规则来书写。ether dst XXX是指定目标MAC地址为XXX。arp是只收ARP协议的数据包，也就是指定Ethernet协议的“类型”字段值为 0x0806。

---

9 字段可以是可选（optional）的，对于可选的字段使用omit值来表示该字段不存在。

10 WinPcap是Pcap库的Windows上的实现。而过滤条件表达式的书写规则可参照WinPcap用户指南的“过滤串表达式的语法”章节。网上可以找到中文的翻译。

68 行 生成 ARP 请求报文的值对象。

70 行 对报文数据包内的可计算字段进行自动计算（这里是 Arp 协议的 Hlen 和 Plen 字段）。

71-73 行 打印值对象的文本表示。

75 行 对值对象进行编码，返回编码后的字节数组。“false”参数表示不再进行自动计算，因为在 70 行已经做过了。

77 行 使用 Pcap 套接字发送编码数据。

79 行 使用 Pcap 套接字接收应答数据，接收的数据被写入“resData”参数，函数返回已接收数据的长度（字节单位）。

81-82 行 对接收数据进行解码。由于底层协议为 Ethernet 所以使用 EtherProtocol 作为推测类型。

84-86 行 打印解码后的 ARP 应答报文的值对象。

88-89 行 从 EthernetProtocol 的 payload 字段取得 ARP 协议字段。

91-92 行 取得 ARP 协议的 SenderHwAddress 从而得到目标设备的 MAC 地址。这里用 getTextAs 方法来按照指定的文本样式取得字段的值，该文本样式为 MAC 地址。

95 行 关闭套接字。

5. 协议语法表示语言

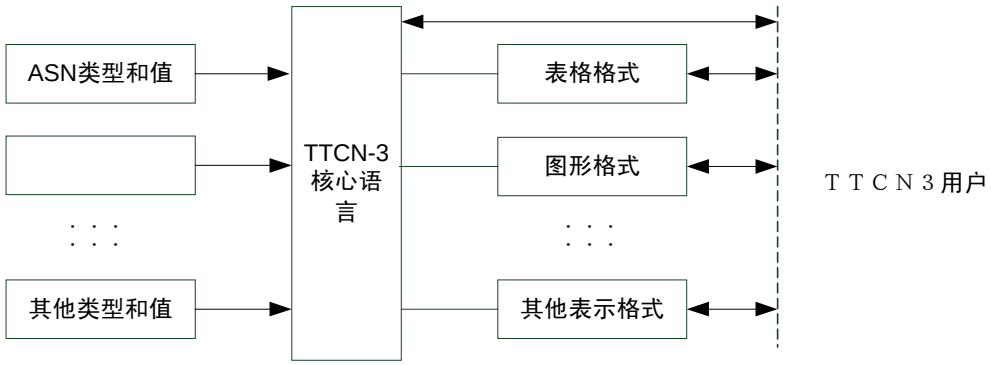
FPB使用独创的协议语法表示语言（Protocol Syntax Notation，简称PSN）来定义协议的格式与编码和解码规则，用户可以在基本不编程（或很少编程，如checksum计算）的情况下定义自己的协议。协议语法表示语言建立在TTCN-3[11]语言的基础上的，它继承了TTCN-3 语言的协议定义相关部分的语法，并且根据协议分析功能的需要进行了扩展。本章将在简要介绍一下TTCN-3 语言之后，对PSN的语法规则进行详细说明，以便读者在阅读本章之后能够读懂定义协议格式和协议编码解码规则的协议模块文件，然后在此基础上就可以运用FPB的Java API（见 6 用户API）来进行数据包的创建和协议的分析工作了。另外，所有FPB内置的协议模块文件可以在安装目录下的“protocol”目录里找到。

5.1. TTCN-3 语言简介

TTCN-3 语言是ETSI[12]制定的一个标准，它是一种灵活和强有力的语言，主要用于描述在多种通信端口上的各种响应系统测试。它应用的典型领域是协议测试（包括移动和互联网协议）、服务测试、基于平台的CORBA测试、API测试等等。TTCN-3 并不仅限于一致性测试，它可以用于许多其他种类的测试，如互操作性测试、健壮性测试、回归测试、系统和集成测试。

从语法的角度看，TTCN-3 与在 ISO/IEC 9646-3 中定义的该语言的早期版本有很大区别。然而，它保留了大量 TTCN 的经证实的基本功能性，并在某些方面做了改进。TTCN-3 包括以下重要特性：

- 描述动态并发测试配置的能力；
- 基于过程的操作和基于消息的通信；
- 描述编码信息和其他属性（包括用户扩展性）的能力；
- 描述数据和带有强有力的匹配机制的属性模板的能力；
- 类型和值的参数化；
- 赋值和测试判定的处理；
- 测试套参数化和测试例选择机制；
- TTCN-3 使用和 ASN.1 的结合（以及与其他语言结合使用的潜力，如与 SDL 的结合）；
- 良好定义的语法，格式的互换以及静态语义；
- 不同的表示格式（如：表格和图形表示格式）；
- 精确的执行算法（操作语义）。



[11] TTCN-3 的全称是Testing and Test Control Notation version 3，它是ETSI制定的通用协议测试语言标准，具体内容可以参考http://www.ttcn-3.org/ 和http://www.ttcntest.com/。

[12] ETSI是European Telecommunications Sdandards Institute（欧洲电信标准协会）的缩写。

図 5-1 TTCN-3 核心语言和各种表示格式的用户视图

TTCN-3 有多种表示方式来描述测试用例，FPB中所指的TTCN-3 语言是指它的核心语言，図 5-1 是它的各种表示格式。FPB只实现了TTCN-3 语言中的与协议定义有关的部分，表 5.1-1是FPB的协议语法表示语言与TTCN-3 核心语言在语法和功能上的比对表。

语言元素	相关联的关键字	TTCN-3 支持	PSN 支持
模块定义	module	○	○
其他模块的定义引入	import	○	○
组定义	group	○	
数据类型定义	type	○	○
通信端口定义	port	○	
测试成分定义	component	○	
特征定义	signature	○	
外部函数/常量定义	external	○	
常量定义	const	○	○
数据/特征模板定义	template	○	○
函数定义	function	○	
可选步定义	altstep	○	
测试例定义	testcase	○	
变量声明	var	○	
定时器声明	timer	○	
模块参数	modulepar	○	○
枚举标签集合	enumset		○

表 5.1-1 TTCN-3 与 PSN 的语言元素比对表

5.2. PSN 的语言元素概要

PSN在FPB中主要用于定义协议的语法，包括协议的格式、编码解码规则和字段文本描述规则。PSN是以模块(模块文件)为单位来管理定义内容的，一个文件内有且只能有一个模块，模块文件名称必须与模块名一致。“4.1Ethernet协议的定义”中例举了一个完整的协议模块文件，虽然该模块文件的内容比较简单，但基本表现了模块文件的基本结构和主要元素。

如前所述，一个模块文件内可以定义一个模块，模块内又可以包含其他模块元素的引入（import）、类型（type）、模板（template）、常量（const）、模块参数（modulepar）、枚举标签集合（enumset）这六种模块元素。为了定义编码和解码等额外的语法规则，还需要使用描述属性（with）来对各个元素进行扩展。描述属性共有编码(encode)、编码变量（variant）、枚举标签集合（enum）、枚举标签集合引用（enumref）和运行时设置（runtime）这五种类型。 接下来对各种元素和描述属性的语法和用法进行分别说明。

5.3. 模块

模块（又称协议模块）是PSN管理定义的逻辑单位，同时模块与模块文件（又称协议模块文件）是一一对应的，所以它也是管理定义的物理单位。“4.1Ethernet协议的定义”给出了一个完整的模块定义例子，如其所述，模块的结构具体如図 5-2 所示。

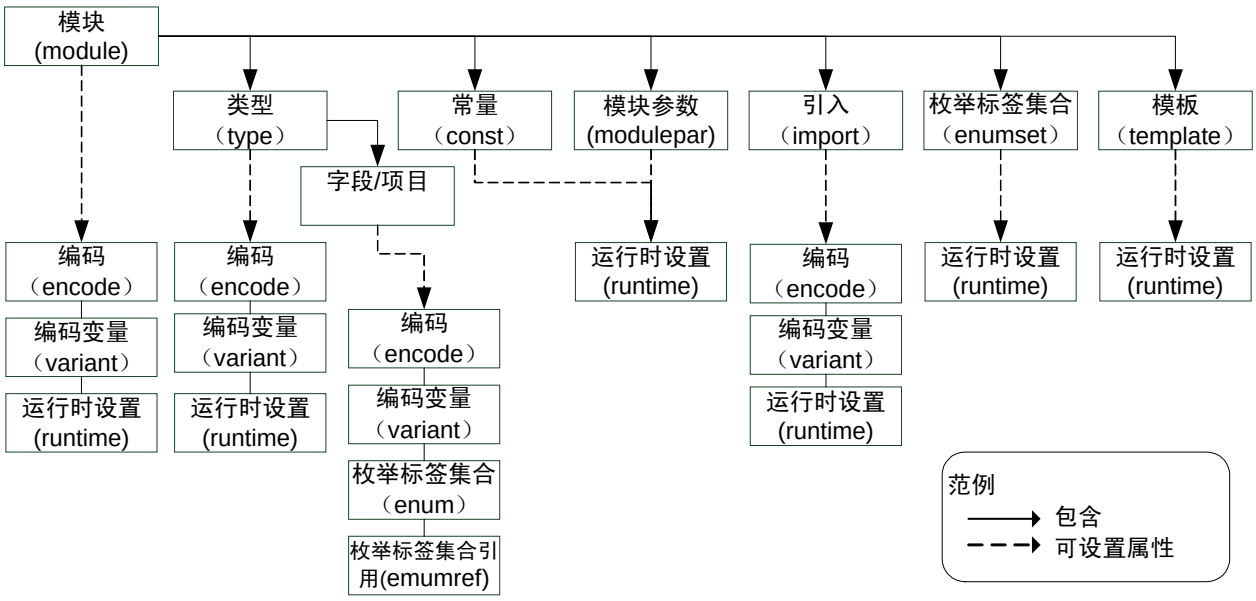


図 5- 2 模块构成图

図 5- 2 说明了模块内可以包含的模块元素类型，以及各个元素所能设置的描述属性的种类。其中“字段/项目”是指record、set、union这些容器类型内说包含的字段元素和record of、set of这些列表类型内所包含的项目元素。字段和项目元素也有自己的类型，不过它们可以设置描述属性来覆盖和扩展原类型的属性设置。

5.4. 注释

(1) 概要

PSN 中有多行和单行两种注释，语法基本与 Java 的注释一样。

(2) 语法结构

下例是多行注释。

例 1：

```
/* 这是一个注释块
占两行 */
注释块不应嵌套。.
/* 这不是 /* 一个合法的 */ 注释 */
```

下例是单行注释。

例 2：

```
// 这是一个行注释
// 占两行
```

注释行可以接在 PSN 语句的后面，但是不能嵌套在一个语句中。

例 3：

```
// 下面是不合法的
```

```
const // This is MyConst integer MyConst := 1;
// 下面是合法的
const integer MyConst := 1; // This is MyConst
```

5.5. 类型和值

5.5.1. 概要

PSN 支持许多预定义的基本类型。这些基本类型包括一般程序语言所有的基本类型，如整形（integer）、布尔类型（boolean）和串类型，也包括一些 PSN 特殊的类型，如对象标识类型（objid）和判定类型(verdicttype）。可以通过这些基本类型来构造结构类型，如记录类型（record）、集合类型（set）、联合类型（union）和枚举类型（enumerated）。

PSN的类型汇总见表 5.5-1。

类型分类		关键字	子类型
简单基本类型		integer	range, list
		float	range, list
		boolean	list
		objid	list
		verdicttype	list
基本串类型		bitstring	list, length
		hexstring	list, length
		octetstring	list, length
		charstring	list, length
		universal charstring	list, length
结构类型	容器类型	record	list
		union	list
		set	list
	列表类型	record of	list, length
		set of	list, length
	枚举类型	enumerated	list

表 5.5-1PSN 类型一览表

5.5.2. 基本类型和值

(1) 简单基本类型和值

PSN 支持下列基本类型：

(a) integer：整型，其值为所有的正、负整数和零。

整型值应该用一个或多个数字表示，且除 0 值外其第一位不应该是 0，而 0 应该由一位数字表示（即单个 0）。

(b) float：浮点类型，描述浮点数的一个类型。

浮点数表示作： <尾数> × <基数>$^{<指数>}$

其中，<尾数>是一个正或负整数， <基数>是一个正整数（多数情况为 2、10 或 16），<指数>为一个正或负整数。

浮点数的表示限定为以值 10 为基数，浮点值可以使用下列任一方式表示：

① 在一个数字序列中使用小数点的正常表示，如 1.23（表示 123*10^{-2}），2.783（即 2783 × 10^{-3}），或-123.456789（表示-123456789 × 10^{-6}）。

② 使用E来分开两个数字来表示，前一个数字描述尾数，第二个数字描述指数，例如 12.3E4（表示 12.3×10^4），-12.3E-4（表示-12.3×10^{-4}）。

(c) boolean：布尔类型，该类型有两个不同值。

布尔类型的值应使用 true 和 false 来表示。

(d) objid：对象标识类型，其值为与 ITU-T Recommendation X.660 [16]的章节 6.2 一致的所有对象标识符的集合。

FPB2.0 尚未支持。

(e) verdicttype：判定类型，该类型有五个不同的值。

Verdicttype 类型的值应该使用 pass、fail、inconc、none 和 error 表示。

(2) 基本串类型和值

PSN支持下列基本串类型[13]：

(a) bitstring：比特串类型，其值为不同的 0 位、1 位或多位的 0、1 序列。

bitstring 类型值应该用任意数目的比特数 0、1 来表示（可能为 0 位），以字符“'”开始，后接字符“'B”。

例：'01101'B

(b) hexstring：十六进制串类型，其值为 0 位、1 位或多位十六进制数的有序序列，每个十六进制数与一个有序的四比特序列相符。

Hexstring 类型值应该用任意数目的十六进制数字(0 1 2 3 4 5 6 7 8 9 A B C D E F)来表示（可能为 0 位），以字符“'”开始，后接字符“'H”；使用十六进制表示法每个十六进制数字用于表示半个八位组的值。

例：'AB01D'H

(c) octetstring：八位组串类型，0 个或正偶数个十六进制数字的有序序列，每对数字与一个有序的八比特序列相应。

octetstring 类型值应该用任意但必须是偶数数目的十六进制数字（0 1 2 3 4 5 6 7 8 9 A B C D E F）来表示（可能为 0 位），以字符“'”开始，后接字符“'O”；使用十六进制表示法每个十六进制数字用于表示半个八位组的值。

例：'FF96'O

[13] PSN中通用术语“ 串” 或“ 串类型” 指的是比特串（bitstring ） 、十六进制串（ hexstring ） 、八位组串（octetstring ） 、字符串（charstring ） 和通用字符串（universal charstring）。

(d) charstring：字符串类型，其值为 0 个、1 个或多个与 ISO/IEC 646 [5]的章节 8.2 中描述的国际参考版本（International Reference Version，IRV）相符的 ISO/IEC 646 [5]版本的字符（characters of the version ofISO/IEC 646 [5]）。

Charstring 类型值应该由来自相关字符集合的任意数目个字符表示，并使用双引号（"）把它括起来。在串中需要包含字符双引号（"）的情况下，在同一行中使用一对双引号来表示该双引号字符，且其间没有空格字符。

例："""abcd"""表示文字串"abcd"。

(e) universal charstring：通用字符串类型，其值为 0 个、1 个或多个与 ISO/IEC 10646 [6]相符的字符。

FPB2.0 尚未支持。

5.5.3. 基本类型的子类型

(1) 概要

用户定义类型用关键字type表示，可以根据表 5.5-1使用用户定义类型在简单基本类型和简单串类型上创建子类型（如列表（lists）、范围（ranges）和长度（length）限制）。

(2) 值列表

PSN允许对表 5.5-1中给出的任一类型的不同值列表的描述。该列表的值应该是源类型的值，且应该是该源类型定义的值集合的真子集，被这个列表定义的子类型限定了该子类型的允许值为列表中的那些值。

例：

```
type bitstring MyListOfBitStrings ('01'B, '10'B, '11'B);
type float pi (3.1415926);
```

(3) 值域

(a) 概要

PSN 允许对类型 integer 和 float 类型（或这些类型的派生类型）值域的描述。这个值域定义的子类型限定了该子类型的值可为值域中的值，且包括该值域的上下界。

例：

```
type integer MyIntegerRange (0 .. 255);
type float piRange (3.14 .. 3142E-3);
```

(b) 无限值域

为了描述一个无限的整型或浮点型的值域，可以使用关键字 infinity 来代替一个值来表示没有上下边界。上边界应该大于等于下边界。

例：

```
type integer MyIntegerRange (-infinity .. -1); // 所有负整数
```

(c) 列表和值域的混合

对于类型 integer 和 float（或这些类型的派生类型）的值来说，也可以混合使用列表和值域。

例：

type integer MyIntegerRange (1, 2, 3, 10 .. 20, 99, 100);

(4) 串长度限定

PSN 允许在串类型上对长度限定进行描述。根据使用长度边界的串类型的不同，该长度边界具有不同的含义。在所有的情况中，这些边界都应该为非负整型值（或是派生的整型值）。关键字 infinity 用于上界的时候表示长度没有上限。上限应该大于等于下限。

例：

```
type bitstring MyByte length(8); // 固定长度值 8
type bitstring MyByte length(8 .. 8); //固定长度值 8
type bitstring MyNibbleToByte length(4 .. 8); // 最小长度为 4，最大长度为 8
```

表 5.5-2描述了了不同串类型的长度单位。

类型	长度单位
bitstring	比特
hexstring	十六进制数字，四位
octetstring	八位组
charstring	字符

表 5.5-2长度描述字段中使用的长度单位

5.5.4. 结构化的类型和值

(1) 概要

关键字 type 也用于描述结构化的类型，如记录类型（record）、记录列表类型（record of）、集合类型（set）、集合列表类型（set of）、枚举类型（enumerated）和联合类型（union）。

可以使用一个明确的赋值表示或一个简写的值列表给出这些类型的值。

例 1：

```
const MyRecordType MyRecordValue:= // 赋值表示
{
        field1 := '11001'B,
        field2 := true,
        field3 := "A string"
}
// 或者
const MyRecordType MyRecordValue:= {'11001'B, true, "A string"} // 值列表表示
```

当使用赋值表示方法描述部分值的时候（即仅设置一个结构变量字段子集的值），只有被赋值的字段才必须被描述。在结构中使用值列表表示时，应该使用一个值来描述所有的字段，用符号“-”或关键字 omit 表示不使用的字段。

例 2：

```
const MyRecordType MyVariable:= // 赋值
{
        field1 := '11001'B,
        field3 := "A string"
}
// 或
```

```
const MyRecordType MyVariable:= {'11001'B, -, "A string"} // 值列表表示
```

在同一（紧接着的）上下文中，不允许混合使用这两种值表示方法。

例 3：

```
// 这是不允许的
const MyRecordType MyRecordValue:= {MyIntegerValue, field2 := true, "A string"}
```

无论是在赋值表示方法还是值列表表示方法中，都应该对可选字段使用明确的值omit来表示该字段不存在。~~省略一个字段会引起相关字段值变成未定义字段，而不管该字段以前具有什么样的值。~~对非可选字段不应该使用关键字omit。

(2) 记录类型和值

(a) 概要

PSN 支持有序的结构化类型，即记录类型（record）。一个 record 类型元素可以是基本类型或用户定义数据类型(如其它记录或集合类型)的任一种，一个 record 值应该与该 record 字段的类型兼容。对于 record，其元素标识符是该 record 的本地标识符，且在该 record 中是唯一的（但不必全局唯一）。record 类型的常量应该既不直接也不间接包含模块参数作为字段值。

```
type record MyRecordType
{
        integer field1,
        MyOtherRecordType field2 optional,
        charstring field3
}
type record MyOtherRecordType
{
        bitstring field1,
        boolean field2
}
```

可以定义记录没有字段（即作为一个空记录）。

例 1：

```
type record MyEmptyRecord {}
```

把一个记录值赋值给一个单个的元素。

例 2：

```
const integer MyIntegerValue:= 1;
const MyOtherRecordType MyOtherRecordValue:=
{
        field1 := '11001'B,
        field2 := true
}
const MyRecordType MyRecordValue:=
{
        field1 := MyIntegerValue,
        field2 := MyOtherRecordValue,
        field3 := "A string"
```

```
}
```

或使用一个值列表。

例 3：

```
const MyRecordType MyRecordValue := {MyIntegerValue, {'11001'B, true}, "A string"};
```

应该使用省略符号省略可选字段。

例 4：

```
const MyRecordType MyRecordValue := {MyIntegerValue, omit , "A string"};
// 注意这与下面的写法不同
// const MyRecordType MyRecordValue:= {MyIntegerValue, -, "A string"}
// 后面的写法意味着 field2 的值不变（为 null）
```

(b) 引用一个 record 类型的字段

应使用点号来对 record 类型进行引用：类型或值标识符.元素标识符，类型或值标识符用来解析一个结构化类型或变量的名字，元素标识符用来解析结构化类型中一个字段的名字。

例：

```
MyVar1 := MyRecord1.myElement1;
// 如果一个 record 类型嵌套在另外一个类型中，那么对它的应用可以看起来象下面的格式
MyVar2 := MyRecord1.myElement1.myElement2;
```

(c) record 类型的可选元素

应使用关键字 optional 来描述一个 record 类型的可选元素。

例：

```
type record MyMessageType
{
        FieldType1 field1,
        FieldType2 field2 optional,
        :
        FieldTypeN fieldN
}
```

(3) 集合（Set）类型和值

(a) 概要

PSN 支持无序的结构化类型，即集合类型（set）。set 类型和值与 record 类型很相似，只是 set 类型字段的顺序是没有意义的。

例：

```
type set MySetType
{
        integer field1,
        charstring field2
}
```

Set 类型字段标识符对于 set 类型来说是本地的，且在该 set 类型中应该唯一（但是不必是全局唯一

的）。set 类型值不应使用值列表表示方法。

(b) 对集合类型字段的引用

应使用点号对set类型元素进行引用（见5.5.4(2)(b)）。

例：

```
MyVar3 := MySet1.myElement1;
// 如果一个 set 类型嵌套在另一个类型中，那么该引用可以看起来象下面的格式
MyVar4 := MyRecord1.myElement1.myElement2;
// 注意，带有被引用标识符 myElement2 的 set 类型嵌套在一个 record 类型中。
```

(c) 集合中的可选元素

应使用关键字 optional 来描述 set 类型中的可选元素。

(4) 记录和集合列表类型

PSN 支持对所有元素为同一类型的 record 或 set 类型(记录列表或集合列表类型)，并使用关键字 of 来表示。这些记录和集合没有元素标识符，可以认为它们分别与有序和无序数组相似。

使用关键字 length 来限定 record of 和 set of 类型的长度。

例 1：

```
type record length(10) of integer MyRecordOfType; // 是一个记录，正好有 10 个整数
type record length(0 .. 10) of integer MyRecordOfType; // 是一个记录，最多有 10 个整数
type record length(10 .. infinity) of integer MyRecordOfType; // 最少 10 个整数的记录类型
type set of boolean MySetOfType; // 布尔值的一个无限集合
// 一个记录列表类型，最多有 10 个串，每个串正好 12 个字符
type record length(0 .. 10) of charstring StringArray length(12);
```

record of 和 set of 类型的值表示应该是一个值列表表示法或是一个对各元素进行索引的表示方法。

当使用值列表表示法时，列表中的第一个值被赋值给第一个元素，第二个值被赋值给第二个元素，依此类推。可以使用“-”占位符号来给例表项目赋空值。

索引值表示法既可以用在赋值符号的左边也可以用在赋值符号的右边。第一个元素的索引值应为 0，且该索引值不应超出子类型设定的长度限制。

例 2：

```
type record of integer MyRecordOf;
//列表表示法来赋值
const MyRecordOf MyRecordVar := { 0, 1, 2, 3 };
//将列表中的第 0 个项目赋值给常量，intItemValue1 将被赋值为 0
const integer intItemValue1 := MyRecordVar[0];
//索引表示法来赋值，将导致 MyRecordVar2 的项目值为{0, null, null, null, 4}。
const MyRecordOf MyRecordVar2 := {
  [0] := 0,
  [4] := 4
}

//给列表中项目赋空值，将导致 MyRecordVar3 的项目值为{0, 1, null, -2, omit}。
const MyRecordOf MyRecordVar3 := { 0, 1, -, 2, omit };
```

嵌套的 record of 和 set of 类型会导致一个类似多维数组的一个数据结构。

例 3：

```
//相当于定义一维数组
type record of integer MyBasicRecordOfType;
//相当于定义两维数组
type record of MyBasicRecordOfType MyRecordOfType;
//索引表示法来赋值，将导致 myRecordOfArray 的值为{null, {null, null, 2}, {null, 1}}。
const MyRecordOfType myRecordOfArray := {
    [1] := { [2] := 2 },
    [2] := { [1] := 1}
}

//取得 myRecordOfArray 索引为 1 的 MyBasicRecordOfType 类型项目的索引为 2 的 integer 项目
// intItemValue2 将被赋值为 2
const integer intItemValue2 := myRecordOfArray[1][2];
```

(5) 枚举类和值

PSN 支持枚举类型（enumerated）。枚举类型用于类型的可能值是固定个数的枚举元素的情况下，每个枚举元素应该有一个标识符。对枚举类型的操作应该仅使用这些标识符。枚举元素标识符应该对该枚举类型来说是唯一的（但不必是全局唯一的），因而也就仅在给定类型中的上下文中可见。

例 1：

```
type enumerated MyFirstEnumType {
        Monday, Tuesday, Wednesday, Thursday, Friday
};
type record MyNewRecordType {
        MyFirstEnumType firstField,
        integer secondField
};
// 通过 MyNewRecordType 的 firstField 元素隐式地引用 MyFirstEnumType
const MyNewRecordType newRecordValue := { Monday, 0 }
```

每个枚举元素可以选择性地在其枚举元素名字后面的括号中定义一个分配给它的整型值，在一个 enumerated 类型内部，每个元素分配的整型数应该是不同的。对于没有分配整型值的枚举元素，系统按文本顺序陆续关联一个整型数，从左边、以 0 为开始，步长为 1，同时跳过任一手工分配值占用的数字。

(6) 联合类型

(a) 概要

PSN 支持联合类型（union），联合类型是字段的集合，这些字段每个都由一个标识符来标识。 联合类型与记录和集合类型不同的是，它同时只能有一个元素（字段）有效，通常用来描述实际类型是有限个可选类型之一的情况。

例：

```
type union MyUnionType
{
        integer number,
        charstring string
};
```

```
const MyUnionType numberVar := { number := 10 };
const MyUnionType strVar := { string := "abc" };
```

用于设置值的值列表表示法不应该用于联合类型的值。

(b) 对联合类型字段的引用

应使用点号来引用联合类型（见5.5.4(2)(b)）。

例：

```
const MyVar5 := MyUnion1.myChoice1;
// 如果一个 union 类型嵌套在另外一个类型中，那么对它的引用可以看起来象如下格式
const MyVar6 := MyRecord1.myElement1.myChoice2;
// 注意，被引用的带有标识符为 myChoice2 的字段的 union 类型嵌套在一个 record 类型中。
```

(c) 可选性和联合类型

联合类型不允许使用可选字段，这就意味着关键字 optional 不会与联合类型一起被使用。

5.5.5. 类型兼容性

通常，在赋值、实例化和比较时，PSN 要求值的类型兼容。

在 PSN 中判断两个类型是否兼容的条件是类型的根类型是否相同，相同则认为是兼容，否则为不兼容。所谓根类型就是实体类型（非子类型）或子类型的派生链中最顶端的父类型（也应为实体类型）。

例：

```
//integer 的子类型，4bit 长度的限制条件
type integer UINT4 (0 .. 15);
// integer 的子类型，8bit 长度的限制条件
type integer UINT8 (0 .. 255);

const UINT8 uint8Value := 100;
//虽然 UINT8 和 UINT4 的限制条件不匹配，但是由于它们的根类型都是 integer，所以它们是兼容的
//但是通过 uint8Value 赋值给 unint4Value 的值不符合限制条件，所以在 Java API 中对 uint4Value
//进行值的完整性验证会失败
const UINT4 uint4Value := uint8Value;

//uint8Value 的根类型不是 float，所以与 float 类型不兼容，以下语句在编译时会出错
const float floatValue := uint8Value;
```

5.6. 引入

(1) 概要

PSN使用引入（import）语句来引入别的模块内的模块元素。除了引入模块元素（被引入的其它模块的元素）之外，模块内部定义的实体元素[14]都可以被别的模块引用。并且模块之间可以互相（双向）引用，即A引入B的同时B也可以引入A，但是模块间的双向引用仅限于类型，其它模块元素的引用将限于单向。

[14] 对于类型来说包括实体类型和非引入的子类型。

(2) 语法结构

下例是引入另一模块的内所有实体元素。

例 1

```
//从 module2 引入所有实体元素
import from module2 all;
```

下例是引入另一模块的内指定类型的多个元素。

例 2

```
//从 module2 引入类型 UINT4、StringToken 和 EnumType1；常量 cValue1 和 cValue2。
//枚举标签集合(enumset)、模块参数（modulepar）、模板（template）也可以使用
//同样方法
import from module2 {
        type UINT4, StringToken, EnumType1;
        const cValue1, cValue2;
};
```

下例是引入另一模块内指定类型的所有元素。

例 3

```
//从 module2 中引入所有枚举标签集合(enumset)、常量（const）、模块参数（modulepar）。
//类型(type)、模板（template）也可以使用同样的方法
import from module2 {
      enumset all;
      const all;
      modulepar all;
}
```

(3) 限制条件

被引入的模块元素名可以在引入模块中直接使用，但是当引入元素之间或与本模块的其它实体元素之间有名称冲突的话，可以使用“被引入模块名.被引入元素名”格式的全称来引用。

5.7. 常量

(1) 概要

常量用来定义静态的常数，在模块初始化时被初始化，初始化之后就不能改变了。

(2) 语法结构

下例是定义一个子类型的常量。

例：

```
type integer UINT4 (0 .. 15);
const UINT4 intVar1 := 1;
```

(3) 限制条件

常量初始化语句中只能引用别的常量，而不能够引用模块参数。

5.8. 模块参数

(1) 概要

模块参数用来定义动态的参数，在模块初始化时可以被赋初值也可以初值为空。模块参数可以设置为线程独立的，当设置为线程独立时不同的线程会保存各自独立的模块参数的状态。

(2) 语法结构

下例是单类型的定义。

例 1：

```
//定义有初始值和无初始值的模块参数，intMpar1 的初始值为 1，intMpar2 没有初始值。
modulepar UINT4 intMpar1 := 1, intMpar2;
```

下例是多类型的定义。

例 2：

```
//同时定义 StringToken、FloatType1 和 UINT4 三种类型的模块参数。
modulepar {
    StringToken strMpar1 := "abc", strMpar2;
    FloatType1 fMpar1 := 100.0;
    UINT4 intMpar4 := intVar1; //intVar1 是常量
}
```

下例是线程独立的模块参数。

例 3：

```
thread modulepar UINT4 intMpar3 := 3;
```

(3) 限制条件

模块参数的初始化语句中只能引用常量，而不能够引用别的模块参数。

5.9. 枚举标签集合

(1) 概要

枚举标签通常用在打印字段值的文本描述时。枚举标签集合由复数个枚举标签组成，每个枚举标签则由选择条件和标签文本两部分组成。在生成字段值的文本描述时，顺序根据枚举标签的选择条件来判断标签是否与字段值匹配，第一个匹配枚举标签的标签文本将作为字段值的文本描述。

(2) 语法结构

下例是由三个枚举标签构成的集合，其选择条件的基类型是 integer。

例：

```
enumset integer Protocols {
    (0)     "IPv6 Hop-by-Hop Options",
    (1)     "ICMP: Internet Control Message Protocol",
    (2)     "IGMP: Internet Group Management Protocol"
}
```

5.10. 模板

FPB2.0 尚未实现

5.11. 描述属性

5.11.1. 概要

(1) 概要

描述属性用来扩展 PSN 语言元素的特性，比如可以用来描述类型的编码、解码、值的文本描述和运行时设置等。所有描述属性都在 with 语句中书写，并且根据描述内容的特点不同类型的描述属性有不同的书写格式。

(2) 语法结构

下例是定义用于 import 语句的描述属性。

例 1：

```
import from langtest1 { type UINT4, StringToken, EnumType1 } with {
    //指定 UINT4 类型的编码变量属性
    variant(UINT4) Length(4); Signed(false); ByteOrder(BIG_ENDIAN);
    //指定 UINT4 类型的运行时设置属性
    runtime(UINT4) TypeJavaClass="com.fineqt.fpb.lib.type.impl.PImportTypeExtImpl"
}
```

下例是定义 module 语句的描述属性。

例 2：

```
module Ipv4Protocol {
…
} with {
    encode "FPB"
    variant ByteOrder(BIG_ENDIAN)
}
```

下例是定义类型和字段的描述属性。

例 3：

```
type record EtherProtocol {
    MacAddress dstAddress,
    MacAddress srcAddress,
    UInt16 etherType,
    EtherPayload payload,
    Oct4 fcs optional
} with {
    //对于整个 EtherProtocol 类型的编码变量属性
    variant Protocol(true); PushEmptyField([DECODE], "etherType")
    //对于 etherType 字段的编码变量属性
    variant(etherType) SetField([DECODE])
    //对于 etherType 字段的枚举标签集合引用属性
    enumref(etherType) EtherTypes
}
```

下例是定义列表项目的描述属性。

例 4：

```
type record length(0 .. 100) of integer UINT16List (0 .. 65535) with {
    //列表项目的编码变量属性
    variant(item) Length(16); Signed(false); ByteOrder(NONE);
    //列表类型 UINT16List 整体的运行时设置属性
    runtime ElementID="125" TypeJavaClass="fpbtest.module.builtintest.impl.Uint16ListType"
}
```

5.11.2. 编码和编码变量描述属性

(1) 编码描述属性

编码描述属性用来定义特定类型或字段和项目在进行编码、解码和打印内容操作时适用哪种编码。编码与编码变量属性是配套的，也就是不同的编码有与之配套的不同编码属性。FPB2.0 中还只有" FPB"编码类型一种。

例：

```
encode "FPB"
```

编码属性是可继承的，当特定元素的编码没有指定时它将按照以下顺序继承上层元素的编码属性。

对于字段或列表项目：字段/项目自己->所在的类型

对于类型：类型自己->所在模块->被派生类型

(2) 编码变量描述属性

编码变量属性与编码属性相匹配，用来表示具体的属性内容。编码变量属性内容可以通过多个子特性来描述，下例是典型的书写格式。

例：

```
type record Ipv4Protocol {
    Ipv4Header header,
    Ipv4Payload payload
} with {
    //针对整个 Ipv4Protocol 的编码变量属性，由 Protocol、SelectRefRegion、SelectCond、LengthRef
    // LengthMultiplier 和 PushEmptyField 这 6 个特性来构成，每个特性都由参数表来描述其内容，
    //大部分特性只有一个参数表，而像 PushEmptyField 则允许有多个参数表
    variant Protocol(true); SelectRefRegion(0, 4); SelectCond(integer (4));
            LengthRef("+header/totalLength"); LengthMultiplier(8);
            PushEmptyField([DECODE], "protocol") ([ENCODE], "Ipv4Protocol.header")
    //针对 header 字段的编码变量属性
    variant(header) SetField([ENCODE], "Ipv4Protocol.header")
}
```

(3) FPB 编码的编码变量

(a) Protocol

Protocol 特性用来指定特定容器类型为协议的根类型。

例：Protocol(true);

(b) PushEmptyField、SetField

PushEmptyField 特性用来在对特定的字段或项目进行编码、解码等操作前，在系统的字段堆栈中设置一个空的堆栈字段，在该字段或项目的操作结束后该堆栈字段将会自动弹出堆栈。SetField 特性则将所在字段的值设置入指定的堆栈字段中，以供别的字段所引用。

例 1：

```
//在解码时（DECODE）设置堆栈字段"protocol"，在编码时设置堆栈字段"Ipv4Protocol.header"
PushEmptyField([DECODE], "protocol") ([ENCODE], "Ipv4Protocol.header")
```

例 2：

```
//在 header 字段的编码操作开始前将其值时设置入"Ipv4Protocol.header"堆栈字段中。
variant(header) SetField([ENCODE], "Ipv4Protocol.header")
```

(c) DefaultPresent

用于指定可选字段在父容器类型的值对象初始生成时是否被以默认值生成。默认操作是 DefaultPresent 为 false 时的动作。

例：DefaultPresent(true)

(d) DefaultValue

用来设置所在字段在初始生成时的值。

例：DefaultValue((" "));

(e) ByteOrder

指定特定 integer 或 float 类型在编码或解码时使用何种编码。默认操作是 ByteOrder 为 none 时的动作。ByteOrder 特性是动态可传递的，也就是在进行编码或解码操作时，如果自己的 ByteOrder 特性没有指定，将使用最近有效的上层类型或字段的 ByteOrder 特性。

例：ByteOrder(BIG_ENDIAN)

(f) Signed

指定特定 integer 或 float 类型在编码或解码时是否带符号。默认操作是 Signed 是 false 时的操作。

例：Signed(true)

(g) FlagType

FlagType 为非 none 是，在解码或编码的时候，将把 Record 类型中连续的设置了相同 FlagType 特性的字段连接在一起以一个整体进行操作，并且各个字段是按造 Big Endian 编码的顺序来排列的。默认操作是 FlagType 是 none 时的动作。

例：FlagType(OCTET2)

(h) OverrideWhole

因为在字段中可以同时设置该字段所具有的类型的特性，默认情况下子段中设置的类型特性将会覆盖该类型原有的相同特性值，而原类性中已有而字段中没有设置的特性将会被保留，即默认操作是

OverrideWhole 是 false 时的动作。为了使字段中的类型特性设置完全覆盖原类型中的特性就必须将 OverrideWhole 特性设置为 true。

例：OverrideWhole(true)

(i) Length、DefaultLength

Length 特性用于 integer 和 float 类型的解码和编码时确定值对象的长度。在生成 string 类类型的初始值对象时根据 DefaultLength 特性来确定它们的长度。

例 1：Length(8)

例 2：DefaultLength(4)

(j) LengthMultiplier、LengthIncrement、LengthRef、RefreshLengthRef

字段值的长度可以通过引用别的字段（所谓引用字段，见5.11.3）值来计算得出。所谓的引用字段由 LengthRef特性来确定，在解码时字段长度的计算公式是：

长度（比特） = 引用字段的值 * LengthMultiplier + LengthIncrement

公式 1 根据引用字段计算字段长度

在自动计算时，如果 RefreshLengthRef 为 true，则会根据如下公式对被引用字段的值进行更新：

引用字段的值 =(字段长度（比特）- LengthIncrement) / LengthMultiplier

公式 2 根据字段长度更新引用字段值

例：LengthRef("+header/totalLength"); LengthMultiplier(8); LengthIncrement(8);
RefreshLengthRef(true)

(k) OnelineMode

在基于文本的协议中，OnelineMode 特性用于指定解码时根据哪种换行符来确定行长度，然后用行长度来作为字段值的长度。

例：OneLineMode(CRLF);

(l) RestOfData

RestOfData 用于在解码时指定根据祖先字段的剩余长度来计算自己的长度，可以和 LengthIncrement 特性搭配使用。计算公式如下：

长度（比特） = 祖先子段的剩余长度 + LengthIncrement

例：RestOfData(true); LengthIncrement(-16);

(m) RepeatRef、RefreshRepeatRef

对列表（record of、set of）进行解码和自动计算时可以通过 RepeatRef 特性来引用别的字段，从而进行解码时的列表项目数的预测和自动计算时的引用字段的更新。解码时的计算公式如下：

列表项目数 = 引用字段的值

公式 3 根据引用字段计算列表项目数

自动计算时的计算公式如下：

引用字段的值 = 列表项目数

公式 4 根据列表项目数更新引用字段值

例：RepeatRef("^lines1Count/token1"); RefreshRepeatRef(true);

(n) SelectRefRegion、SelectRef、SelectCond

在解码时对于可选字段、Union 的子字段或 Set 的子字段都要判断特定字段是否有效或被选中，这时可以使用 SelectRefRegion、SelectRef 和 SelectCond 的特性组合来描述选择规则。先用 SelectCond 特性来记述选择判断条件，然后可以用 SelectRefRegion 特性确定的固定数据区域或 SelectRef 确定的引用字段来取得判断数据，再根据 SelectCond 的判断条件来判断是否选中。

例 1：SelectRef("-hasBody", #boolean); SelectCond(boolean (true));

例 2：SelectRefRegion(0, 4); SelectCond(integer (4));

(o) CaseRefRegion、CaseRef、CaseCond、CaseDefault

对于 Union 类型字段，在解码中判断选择哪个分支字段时除了可以使用各个分支字段的 SelectCond 特性外还可以使用效率更高的 CaseRefRegion、CaseRef、CaseCond 和 CaseDefault 特性组合。CaseRefRegion 和 CaeRef 和 CaseCond 的用法和功能基本上和前述的 SelectRef 等相同，用来描述各个分支字段的判断条件，而 CaseDefault 则用来指定当所有的 CaseCond 条件都判断失败时选择的默认分支字段。对于 Set 字段来说也可以使用这种方法来确定当前解码的是那个子字段。

例 1：

```
variant CaseRefRegion(0, 8); CaseDefault(last)
variant(security) CaseCond(integer (130))
```

例 2：

```
variant CaseRef("-protocol", #integer); CaseDefault(data)
variant(icmpv4) CaseCond(integer (1))
```

(p) PadType、PadValue

计算机内部计算的数据单位通常是固定的长度比如 16 比特、32 比特或 64 比特，出于性能上的考虑网络中进行数据传输时往往需要将不满该长度的数据进行对齐操作，也就是补充字节或比特至单位长度。这种字节对齐可以通过 PadType 和 PadValue 特性组合来进行，PadType 指定对齐的长度，PadValue 则指定填充数据的内容。对齐操作在自动计算时表现为自动填充对齐字段的不足长度，在解码时表现为自动计算对齐字段的长度。

例：PadType(OCTET2); PadValue(('10'B))

5.11.3. 字段的引用

在协议的解码中通常会遇到根据别的字段的值来确定如何对当前字段进行解码的情况，比如根据别的字段的值来计算当前字段的长度、判断当前字段的类型或判断某一字段的有无等，那么在这个计算或判断中被使用的其他字段就称为引用字段。引用字段一般在 LengthRef、RepeatRef、SelectRef 和 CaseRef 中被描述，它的参数描述规则如下：

“(” FieldRefPath [“,” FieldType] “)”

FieldRefPath ::= “”” [-+$^@] PPath “””

PPath ::= PPathFragment (“/” PPathFragment)*

PPathFragment ::= FieldName | “[” ListIndex “]” | FieldName “[” ListIndex “]”

FieldName ::= (“a”..”z”|”A”..”Z”) (“a”..”z”|”A”..”Z”|”_”|”0”..”9”)*

ListIndex ::= NUMBER

FieldType ::= TypeRefArgument

FieldRefPath 参数表示整个字段引用的描述规则，FieldType 参数则表示引用字段的类型名称。可以从“-+$^@”中任意选一个符号来表示使用何种引用方法。PPath 描述字段的引用路径，由于字段是可以被包含且有层次关系的，所以使用 PPathFragment 来描述各个不同层次的引用规则。FieldName 表示容器字段（record、set 和 union）的子字段名，ListIndex 表示列表（record of、set of）的成员索引。FieldName 和 ListIndex 可以单独存在也可以混合使用，混合使用时 ListIndex 用来表示通过 FieldName 来指出的列表字段的成员索引。

字段的引用方法则分为堆栈字段引用、上下文属性引用、同级字段引用、下级字段引用和解码结果对象属性引用这五中，具体说明如下表：

类型	符号	说明	例子
堆栈字段引用	-	引用通过 PushEmptyField 特性被压入字段堆栈的字段，并且引用的是同名字段中在堆栈顶层的字段。 必须用 FieldType 参数来指定字段的类型。	CaseRef("-protocol", #integer)
上下文属性引用	$	引用程序内部放入解码、编码和全局上下文内的属性。属性的设置通过下列 Java 接口的 setAttribute 方法来进行。 com.fineqt.fpb.lib.meta.context. DecodeContext com.fineqt.fpb.lib.meta.context. EncodeContext com.fineqt.fpb.lib.system. FpbSystem 必须用 FieldType 参数来指定属性的类型。	
同级字段引用	^	引用与目标字段（使用引用字段的字段）在同一个容器下同级的字段。 由于引用字段的类型在初始化时已知，所以可以省略 FieldType 参数。	LengthRef("^hlen")
下级字段引用	+	引用目标字段（使用引用字段的字段）的子字段。而且该种引用字段只能在 Record 类型目标字段的解码时被使用？。 由于引用字段的类型在初始化时已知，所以可以省略 FieldType 参数。	LengthRef("+header/totalLength")
解码结果对象属性引用	@	引用解码结果对象内的任意 Java 属性。引用属性需符合 Java Bean 的属性定义规范，即为了引用属性 int abc;结果类内必须定义 int getAbc();方法。解码结果对象可以参考如下 Java 接口。 com.fineqt.fpb.lib.type. DecodeResult	

5.11.4. 枚举标签集合和枚举标签集合引用描述属性

枚举标签集合和枚举标签集合引用都是在打印字段值内容文本时，用来指定如何生成字段值的文本描述的。

下例对于字段 protocol 适用枚举标签集合 Protocols 来表示文本内容。

例 1：enumref(protocol) Protocols

下例对于字段 security 定义独立的枚举标签集合。

例 2：

```
enum(security) {
    (0x0000)    "Unclassified",
    (0xF135)    "Confidential"
}
```

5.11.5. 运行时设置描述属性

运行时设置属性用于设置用户自己的 Java 扩展类等运行时信息来扩展 FPB 内置的默认功能。

例：

```
//用"com.fineqt.fpb.protocol.Ipv4TypeEVExtFactory"类来代替系统默认的类型编码变量类工厂
//的类（TypeEvFactoryJavaClass）。
runtime TypeEvFactoryJavaClass="com.fineqt.fpb.protocol.Ipv4TypeEVExtFactory"
```

6. 用户 API

6.1. 概要

用户通过FPB提供的Java API来进行数据包创建、编辑、编码、解码、文本描述打印以及链路层通信操作。所有的用户API都以Java接口形式提供，并且分类为以“com.fineqt.fpb.lib.api.”为前缀的“module”、“value”、“util”、“util.buffer”、“comm”和“comm.pcap”六个包。用户API的完整使用例可以参照“4.2(2)协议客户端模拟实现”，详细使用方法可以参照安装目录下“doc”目录中的Java Doc文档，接下来对各个包的内容作简要的介绍。

6.2. com.fineqt.fpb.lib.api.module 包

module 包主要用来管理和取得协议模块文件的内容。下面是类图

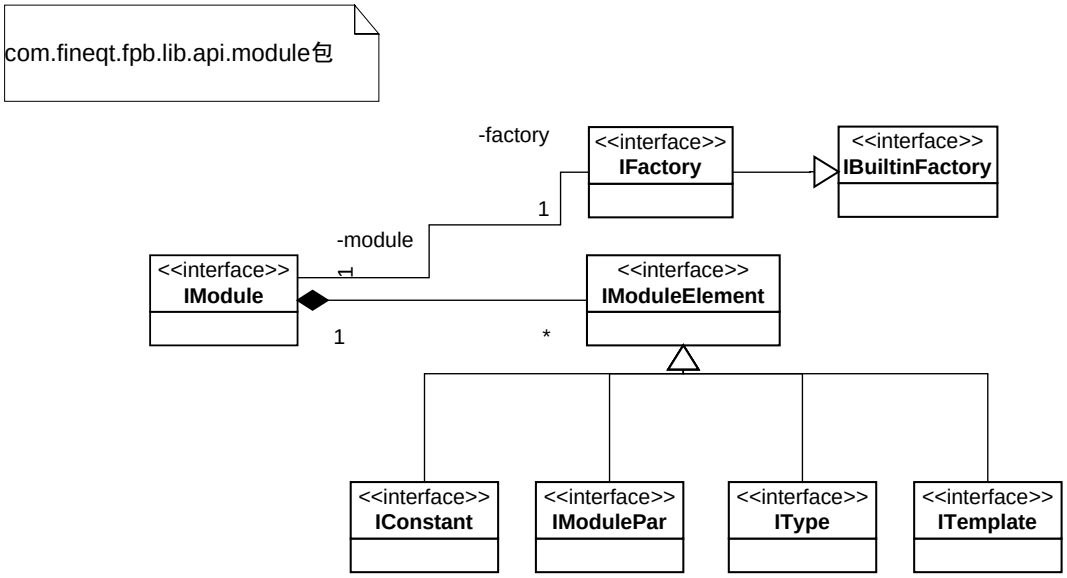


図 6-1 com.fineqt.fpb.lib.api.module 包类图

下面是各个接口的简要说明。

① IModule接口

与PSN中的模块相对应，用来存取协议模块及所包含的类型等模块元素信息。IModule接口可以通过util（见6.4）包中的IModuleRegistry接口来取得。

② IFactory 接口

与 IModule 接口一一对应，用来创建对应模块中类型的值对象。

③ IBuiltinFactory

具有创建“builtin”模块内类型的值对象功能。“builtin”模块包含所有内置基本类型，包括 integer、float、charstring 等。

④ IModuleElement

模块元素的基接口。

⑤ IConstant

与 PSN 中的常量(const)相对应。

⑥ IModulePar

与 PSN 中的模块参数(modulepar)相对应。

⑦ IType

与 PSN 中的类型(type)相对应。

⑧ ITemplate

与 PSN 中的模板(template)相对应。

6.3. com.fineqt.fpb.lib.api.value 包

value 包用来管理不同类型的值接口。下图是它的类图。

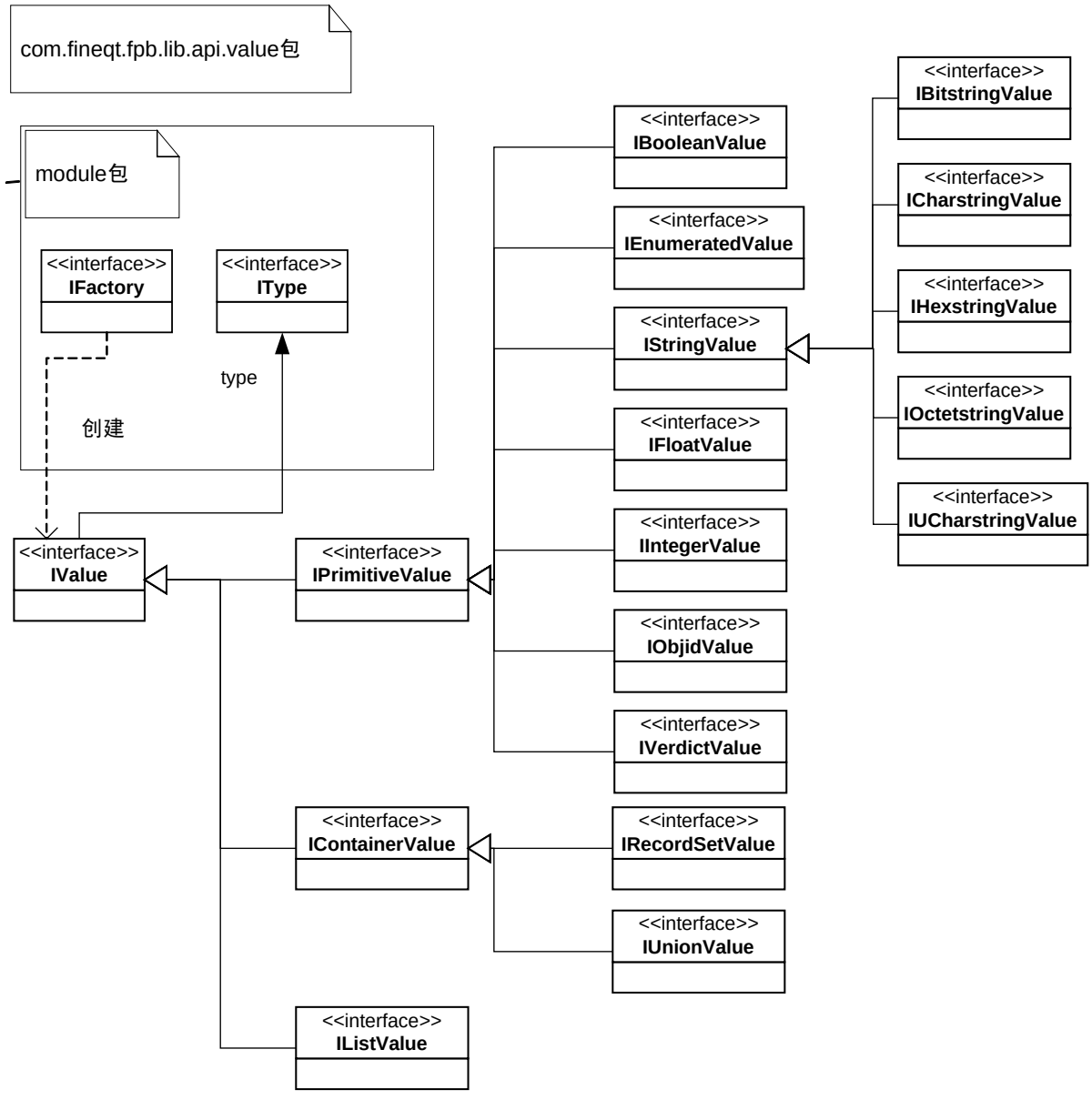


図 0- 1 com.fineqt.fpb.lib.api.value 包类图

上图中的各个接口基本上与 PSN 中的各种类型相对应，其中较特别的是 IUCharstringValue 与

universal charstring 相对应，IRecordSetValue 与 record 和 set 相对应，IListValue 与 record of 和 set of 相对应。各种值对象(IValue)可以通过 IFactory 接口来创建，值对象的类型信息可以通过 IType 属性来取得。

6.4. com.fineqt.fpb.lib.api.util 包

util 包包含了常用的几个工具接口。下面是它的类图。

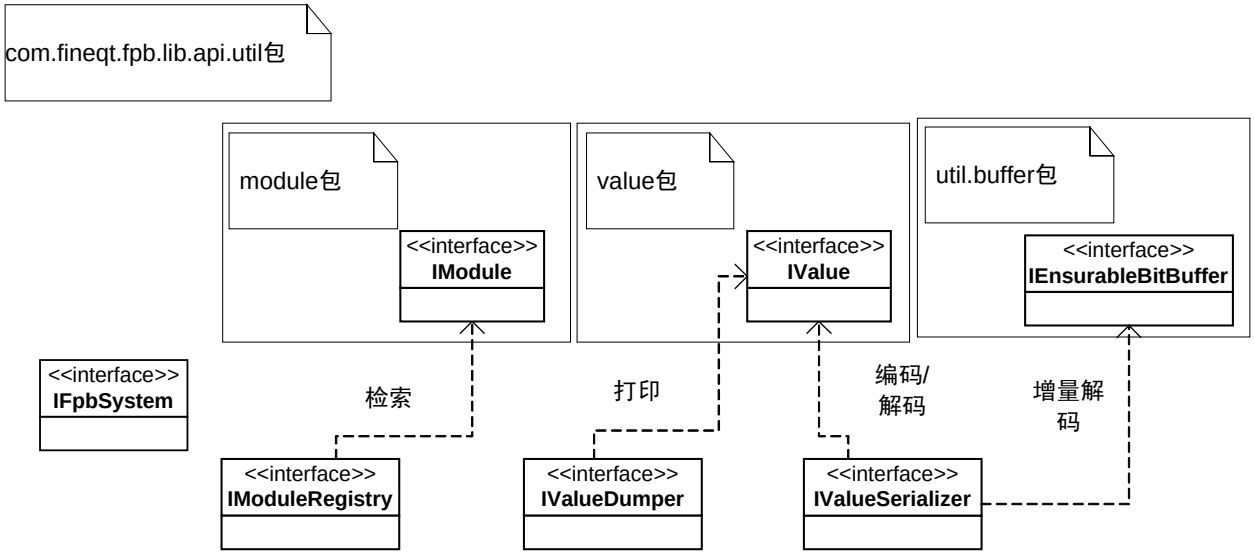


图 0- 2 com.fineqt.fpb.lib.api.util 包类图

下面是各个接口的简要说明：

① IFpbSystem

用于初始化 FPB 系统，取得或设置全局环境属性。

② IModuleRegistry

用于检索和管理协议模块。

③ IValueDumper

用于打印值对象内容的文本描述。

④ IValueSerializer

用于值对象的编码和解码操作。

6.5. com.fineqt.fpb.lib.api.util.buffer 包

util.buffer包内包含数据缓存操作相关的对象。图 0- 3 是它的类图。

下面是各个接口的简要说明：

① IBitBuffer

比特缓存的基类。比特缓存具有存取比特单位数据的功能，并且缓存长度可以设置为固定也可以设置为可变。

② IReadableBitBuffer

具有读功能的 BitBuffer。

③ IWritableBitBuffer

具有写功能的 BitBuffer。

④ IReadWritableBitBufer

同时具有读写功能的 BitBuffer。

⑤ IEnsuableBitBuffer

具有判断特定条件的数据是否已经写入缓存的功能，通常用在值对象的增量解码中。

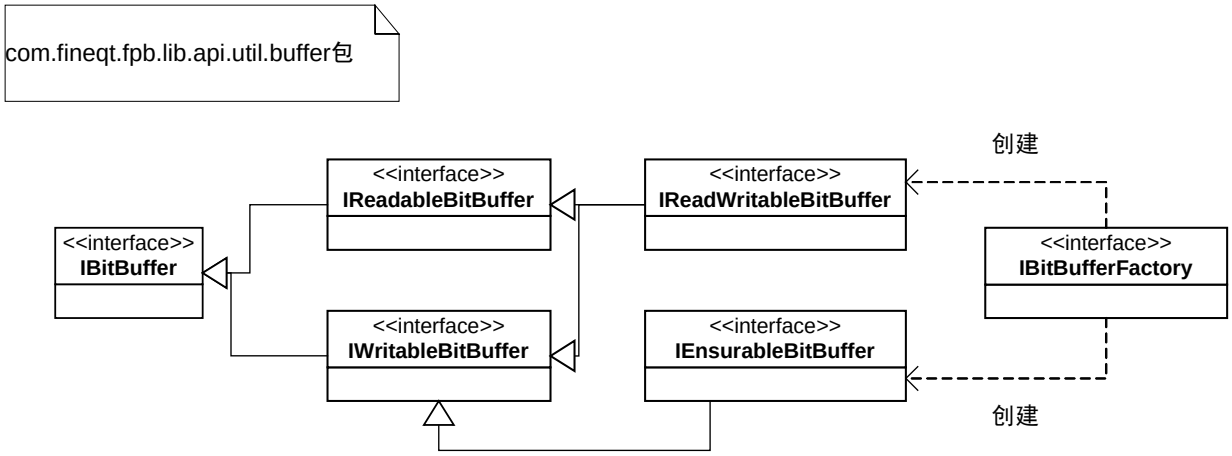


図 0- 3 com.fineqt.fpb.lib.api.util.buffer 包类图

6.6. com.fineqt.fpb.lib.api.comm.pcap 包

comm.pcap 包提供了使用 pcap 软件包进行链路层通信的功能。下面是它的类图。

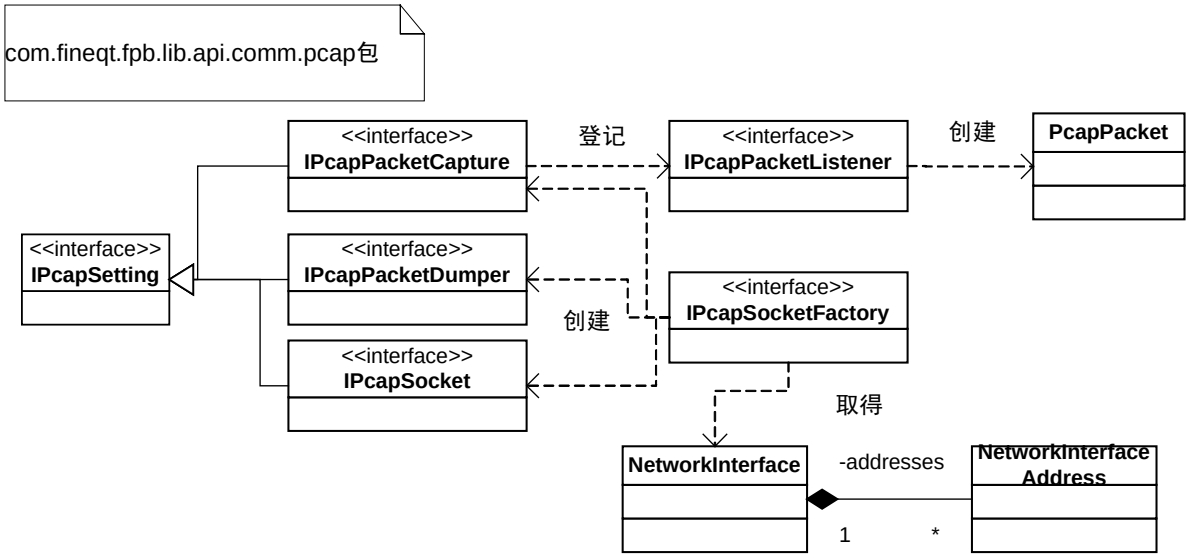


図 0- 4 com.fineqt.fpb.lib.api.comm.pcap 包类图

下面是各个接口的简要说明：

① IPcapSetting

Pcap 对象设置操作接口，封装了 Pcap 软件包的设置功能。

② IPcapPacketCapture

封装了 PCAP 的数据包截取功能。

③ IPcapPacketDumper

用来创建和存取 Pcap 格式的数据包截取文件。

④ IPcapSocket

Pcap 套接字接口。使用 Pcap 库来进行链路层的数据通信。

⑤ IPcapPacketListener

数据包监听器。可以登记到数据包截取器（PacketCapture）中，用来监听截取到的数据包。

⑥ IPcapSocketFactory

用来创建 IPcapPacketCapture、IPcapPacketDumper、IPcapSocket 及取得网络接口信息。

⑦ NetworkInterface

网络接口类。

⑧ NetworkInterfaceAddress

网络接口上的网络地址。

⑨ PcapPacket

数据包对象，用来存放通过 Pcap 库取得的数据包。

7. API 和协议定义模块使用例

在运行库安装目录下sample目录里有多个FPB API和协议模块的使用例源程序，例子程序的清单和执行方法可以参照“2(4)例子程序的执行”。

7.1. ARP 协议使用例

ARP协议使用例在“4.2ARP协议模拟”中已经详细说明过，这里不再复述了。

7.2. Ping 应用

使用 ICMPv4 协议的 Echo Request 和 Echo Reply 报文来实现 Ping 应用。用已定义的 EtherProtocol 类型和 Ipv4Protocol 和 Icmpv4Protocol 类型来实现 PingApp 类，然后可以用该 PingApp 类来执行 Ping 功能。

- PingApp 类的使用方法例：

```
String dstIp = "192.168.11.1";
//生成 Pcap 套接字来实现链路层通信
IPcapSocket socket = IPcapSocketFactory.INSTANCE.createPcapSocket();
socket.setDevice(getInterface().name);
//生成 Arp 对象
ArpApp arp = new ArpApp(socket, LOCAL_MAC, LOCAL_IP);
//设置是否打印数据通信内容
arp.setDumpPacket(true);
//取得目标设备的 MAC 地址
String dstMac = arp.askMac(dstIp);
//生成 Ping 对象
PingApp ping = new PingApp(socket, LOCAL_MAC, dstMac, LOCAL_IP);
//设置是否打印数据通信内容
ping.setDumpPacket(true);
//执行 Ping(循环三次)
ping.ping(3, dstIp);
```

- 上述代码的执行结果例:

```
Reply from 192.168.11.1 bytes=32 time<=0.031(s) TTL=64
Reply from 192.168.11.1 bytes=32 time<=0.0(s) TTL=64
Reply from 192.168.11.1 bytes=32 time<=0.0(s) TTL=64
```

- Echo Request 报文的打印例：

```
0:74   EtherProtocol.EtherProtocol
0:6      dstAddress              00:16:01:15:A4:88
6:6      srcAddress              00:16:D4:17:25:C4
12:2     etherType               2048 IPv4
14:60    payload
14:60      ipv4                  Ipv4Protocol.Ipv4Protocol
14:60 Ipv4Protocol.Ipv4Protocol
14:20    header
14:1       0000----              4 version 0xF0
14:1       ----0000              5 headerLength 0x0F
15:1       typeOfService         '00000000'B
16:2       totalLength           60
18:2       identification        1
20:1       0---------------      0 reserved 0x8000
```

```
20:1        -0----------     false doNotFrag 0x4000
20:1        --0---------     false moreFrags 0x2000
20:2        ---0000000000000     0 fragmentOffset 0x1FFF
22:1        timeToLive            255
23:1        protocol              1 ICMP: Internet Control Message Protocol
24:2        hcs                   '0000'O
26:4        sourceAddress         192.168.11.5
30:4        destinationAddress 192.168.11.1
34:0        options               omit
34:40     payload
34:40       icmpv4                Icmpv4Protocol.Icmpv4Protocol
34:40 Icmpv4Protocol.Icmpv4Protocol
34:40     echoRequest
34:1        icmpType              8
35:1        icmpCode              0
36:2        checksum              '0000'O
38:2        identifier            17076
40:2        sequenceNumber        0
42:32       data                  {}
          0000 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f   ................
          0010 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f   ................
74:0      fcs                   omit
```

- Echo Reply 报文的打印例：

```
0:74    EtherProtocol.EtherProtocol
0:6       dstAddress            00:16:D4:17:25:C4
6:6       srcAddress            00:16:01:15:A4:88
12:2      etherType             2048 IPv4
14:60     payload
14:60       ipv4                Ipv4Protocol.Ipv4Protocol
14:60 Ipv4Protocol.Ipv4Protocol
14:20     header
14:1        0000----            4 version 0xF0
14:1        ----0000            5 headerLength 0x0F
15:1        typeOfService       '00000000'B
16:2        totalLength         60
18:2        identification      51406
20:1        0----------     0 reserved 0x8000
20:1        -0----------     false doNotFrag 0x4000
20:1        --0---------     false moreFrags 0x2000
20:2        ---0000000000000     0 fragmentOffset 0x1FFF
22:1        timeToLive            64
23:1        protocol              1 ICMP: Internet Control Message Protocol
24:2        hcs                   '1A9C'O
26:4        sourceAddress         192.168.11.1
30:4        destinationAddress 192.168.11.5
34:0        options               omit
34:40     payload
34:40       icmpv4                Icmpv4Protocol.Icmpv4Protocol
34:40 Icmpv4Protocol.Icmpv4Protocol
34:40     echoReply
34:1        icmpType              0
35:1        icmpCode              0
36:2        checksum              'CC4A'O
```

```
38:2        identifier              17076
40:2        sequenceNumber          0
42:32       data                    {}
        0000 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f   ................
        0010 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f   ................
74:0    fcs                         omit
```

- PingApp 类的源代码

```
/**
 * 使用 ICMPv4 协议实现的 Ping 应用。 使用 ICMPv4 协议的 Echo Request 和 Echo Reply 报文来实现 Ping 应用。
 */
public class PingApp {
    protected static IModule module = IModuleRegistry.INSTANCE
            .resolveModule("EtherProtocol");
    protected static IFactory factory = module.getFactory();
    protected static IValueSerializer ser = IValueSerializer.INSTANCE;

    private static int ICMP_REQUEST_DATA_BYTES = 32;
    protected String srcMac;
    protected String dstMac;
    protected String srcIp;
    protected IPcapSocket socket;
    protected int timeout = ICommonSocket.DEFAULT_TIMEOUT;
    protected boolean dumpPacket;

    /**
     * 构造函数
     *
     * @param socket PCAP 套接字
     * @param srcMac 源设备的 MAC 地址
     * @param dstMac 目标设备的 MAC 地址
     * @param srcIp 源设备的 IP 地址
     */
    public PingApp(IPcapSocket socket, String srcMac, String dstMac, String srcIp) {
        this.socket = socket;
        this.srcMac = srcMac;
        this.dstMac = dstMac;
        this.srcIp = srcIp;
    }

    /**
     * 生成 ICMPv4 的 Echo Request 报文数据包。协议栈为 Ethernet/IPv4/ICMPv4。
     *
     * @param dstIp 目标设备的 IP 地址。
     * @param icmpId Echo Request 中的标识符
     * @param icmpSeq Echo Request 中的序号
     * @return
     */
    protected IRecordSetValue createPingRequest(String dstIp, int icmpId, int icmpSeq)
            throws MetaException {
        IRecordSetValue etherPrtl = (IRecordSetValue) factory
                .createValue("EtherProtocol");
```

```
        etherPrtl.getField("dstAddress").setTextAs(dstMac, IValueTextStyle.MAC_ADDRESS);
        etherPrtl.getField("srcAddress").setTextAs(srcMac, IValueTextStyle.MAC_ADDRESS);
        etherPrtl.getField("etherType").setText("0x0800");

        IRecordSetValue ipv4Prtl = (IRecordSetValue) etherPrtl.findField("payload/ipv4",
                true);
        IRecordSetValue header = (IRecordSetValue) ipv4Prtl.getField("header");
        header.getField("identification").setText("1");
        header.getField("timeToLive").setText("255");
        header.getField("protocol").setText("1");
        header.getField("sourceAddress").setTextAs(srcIp, IValueTextStyle.IPV4_ADDRESS);
        header.getField("destinationAddress").setTextAs(dstIp,
                IValueTextStyle.IPV4_ADDRESS);

        IRecordSetValue echoRequest = (IRecordSetValue) ipv4Prtl.findField(
                "payload/icmpv4/echoRequest", true);
        ((IIntegerValue) echoRequest.getField("identifier")).setInteger(icmpId);
        ((IIntegerValue) echoRequest.getField("sequenceNumber")).setInteger(icmpSeq);

        byte[] data = new byte[ICMP_REQUEST_DATA_BYTES];
        for (int i = 0; i < data.length; i++) {
            data[i] = (byte) i;
        }
        ((IOctetstringValue) echoRequest.getField("data", true)).setValue(data);
        return etherPrtl;
    }

    /**
     * 根据指示的目标设备 IP 地址通过 ICMPv4 协议来检验源设备到目标设备的网路是否连通。检验
结果会被打印在 标准输出中。
     *
     * @param cnt 检验次数，即 Echo 请求的发送次数。
     * @param dstIp 目标设备的 IP 地址
     * @throws Exception
     */
    public void ping(int cnt, String dstIp) throws Exception {
        byte[] resData = new byte[1514];
        socket.setTimeout(timeout);
        // 打开套接字
        socket.open();
        try {
            // 设置 Pcap 套接字的过滤条件。（过滤条件的书写规则可以参考 WinPcap 的用户手册）
            socket.setFilter("ip dst " + srcIp + " and icmp", true);
            // 生成随机的 ICMP Echo 请求的标识符
            int icmpId = (int) (Math.random() * 0xFFFF);
            // 进行多次 Ping 检验
            for (int i = 0; i < cnt; i++) {
                // ICMP Echo 请求的序号
                int seqNo = i;
                // 生成 ICMP Echo 请求报文
                IRecordSetValue pingRequest = createPingRequest(dstIp, icmpId, seqNo);
                // 对报文数据包内的可计算字段进行自动计算（这里是 IP 协议的 HeaderLength 和
TotalLength 字段）
                ser.calculate(pingRequest);
```

```
// 打印报文内容
if (dumpPacket) {
    IValueDumper.INSTANCE.dump(pingRequest);
}
// 编码
byte data[] = ser.encode(pingRequest, false);
// 发送编码数据
socket.write(data);
// 起始时间
long startTime = System.currentTimeMillis();
// 已花费时间（计算超时）
long totalTime = 0;
// 循环接收数据包直到超时
while (totalTime < timeout) {
    // 接收数据
    int len;
    try {
        len = socket.read(resData);
    } catch (CommTimeoutException e) {
        totalTime = timeout;
        break;
    }
    // 解码（底层协议为 Ethernet 所以使用 EtherProtocol 作为推测类型）
    IRecordSetValue pingResponse = (IRecordSetValue) ProtocolUtils
            .decode(resData, 0, len, module.getType("EtherProtocol"));
    // 打印报文
    if (dumpPacket) {
        IValueDumper.INSTANCE.dump(pingResponse);
    }
    // 取得花费时间
    totalTime = System.currentTimeMillis() - startTime;
    // 从 Icmpv4Protocol 中取出 EchoReply 字段
    IRecordSetValue echoReply = (IRecordSetValue) pingResponse
            .findField("payload/ipv4/payload/icmpv4/echoReply");
    if (echoReply == null) {
        continue;
    }
    // 判断 EchoReply 的标识符和序号是否与 EchoRequest 的一致
    if (((IIntegerValue) echoReply.getField("identifier")).getInteger() != icmpId
            || ((IIntegerValue) echoReply.getField("sequenceNumber"))
                    .getInteger() != seqNo) {
        continue;
    }
    // 取得目标设备的 IP 地址
    IRecordSetValue ipv4Header = (IRecordSetValue) pingResponse
            .findField("payload/ipv4/header");
    String tempIp = ipv4Header.getField("sourceAddress").getTextAs(
            IValueTextStyle.IPV4_ADDRESS);
    // 取得 EchoReply 中的 Data 字段的长度（字节数）
    int dataLen = ((IOctetstringValue) echoReply.getField("data"))
            .getLength();
    // 取得 IPv4 的 TTL 字段值
    int ttl = ((IIntegerValue) ipv4Header.getField("timeToLive"))
```

```
                        .getInteger();
                // 打印检验结果
                String msg = "Reply from " + tempIp + " bytes=" + dataLen + " time<="
                        + totalTime / 1000.0 + "(s)" + " TTL=" + ttl;
                System.out.println(msg);
                break;
            }
            // 判断是否超时
            if (totalTime >= timeout) {
                System.out.println("timeout");
            }
        }
    } finally {
        // 关闭套接字
        socket.close();
    }
}

…

}
```

7.3. Traceroute 应用

使用 ICMPv4 协议的 Echo Request 和 Echo Reply 报文和 IPv4 协议的 TTL 字段来实现 Traceroute 应用。用了已定义的 EtherProtocol 类型和 Ipv4Protocol 和 Icmpv4Protocol 类型来实现了 TracerouteApp 类，然后可以用该 TracerouteApp 类来执行 Traceroute 功能。

- TracerouteApp 类的使用方法例：

```
//网关 IP 地址
String gatewayIp = "192.168.11.1";
//目标 IP 地址
String dstIp = "64.233.189.147";
//生成 Pcap 套接字来实现链路层通信
IPcapSocket socket = IPcapSocketFactory.INSTANCE.createPcapSocket();
socket.setDevice(getInterface().name);
//生成 Arp 对象
ArpApp arp = new ArpApp(socket, LOCAL_MAC, LOCAL_IP);
//设置是否打印数据通信内容
arp.setDumpPacket(true);
//取得目标设备的 MAC 地址
String dstMac = arp.askMac(gatewayIp);
//生成 TracerouteApp 对象
TracerouteApp trace = new TracerouteApp(socket, LOCAL_MAC, dstMac, LOCAL_IP);
trace.setDumpPacket(true);
//执行
trace.trace(dstIp);
```

- 上述代码的执行结果例:

```
(1) from 192.168.11.1, 16(ms),   ttl=64.
(2) from 218.11.160.248, 78(ms),   ttl=126.
(3) from 222.72.254.129, 47(ms),   ttl=253.
```

```
(4) from 124.74.210.165, 46(ms),  ttl=252.
(5) from 61.152.86.42, 32(ms),  ttl=251.
(6) from 202.97.35.118, 47(ms),  ttl=250.
(7) from 202.97.34.50, 15(ms),  ttl=249.
(8) from 202.97.33.5, 62(ms),  ttl=248.
(9) from 202.97.61.38, 78(ms),  ttl=247.
(10) from 202.97.62.214, 63(ms),  ttl=245.
(11) from 209.85.241.56, 78(ms),  ttl=244.
(12) from 66.249.94.6, 78(ms),  ttl=244.
(13) from 64.233.189.147, 110(ms),  ttl=243.
Traceroute completed. 64.233.189.147 reached.
```

- Echo Request 报文的打印例：

除了 IPv4 的 TTL 字段外，与 Ping 应用例基本相同。

- Echo Reply 报文的打印例：

除了 IPv4 的 TTL 字段外，与 Ping 应用例基本相同。

- TracerouteApp 类的源代码：

```
/**
 * 使用 ICMPv4 协议实现的 Traceroute 应用。 使用 ICMPv4 协议的 Echo Request 和 Echo
 * Reply 报文和 IPv4 的 TTL 字段来实现 Traceroute 应用。
 */
public class TracerouteApp extends PingApp {
    public final static int DEFAULT_MAX_HOPS = 40;
    private int maxHops = DEFAULT_MAX_HOPS;

    public TracerouteApp(IPcapSocket socket, String srcMac, String dstMac,
            String srcIp) {
        super(socket, srcMac, dstMac, srcIp);
    }

    public void trace(String dstIp) throws Exception {
        byte[] resData = new byte[1514];
        socket.setTimeout(timeout);
        // 打开套接字
        socket.open();
        try {
            // 设置 Pcap 套接字的过滤条件。（过滤条件的书写规则可以参考 WinPcap 的用户手册）
            socket.setFilter("ip dst " + srcIp + " and icmp", true);
            // 生成随机的 ICMP Echo 请求的标识符
            int icmpId = (int) (Math.random() * 0xFFFF);
            boolean reachedDst = false;
            // 持续尝试直到到达了最大的 HOP 数
            for (int i = 1; i <= maxHops; i++) {
                // sequence NO. of ICMP
                int seqNo = i;
                // field of TTL in IP, increase 1 each time
                int ttl = i;
                // 生成 ICMP Echo 请求报文
                IRecordSetValue pingRequest = createPingRequest(dstIp, icmpId,
```

```
                seqNo);
        pingRequest.findField("payload/ipv4/header/timeToLive")
                .setText(Integer.toString(ttl));
        // 对报文数据包内的可计算字段进行自动计算（这里是 IP 协议的 HeaderLength 和
TotalLength 字段）
        ser.calculate(pingRequest);
        // 打印报文内容
        if (dumpPacket) {
            IValueDumper.INSTANCE.dump(pingRequest);
        }
        // 编码
        byte data[] = ser.encode(pingRequest, false);
        // 发送编码数据
        socket.write(data);
        // 起始时间
        long startTime = System.currentTimeMillis();
        // 已花费时间（计算超时）
        long totalTime = 0;
        // 循环接收数据包直到超时
        while (totalTime < timeout) {
            // 接收数据
            int len;
            try {
                len = socket.read(resData);
            } catch (CommTimeoutException e) {
                totalTime = timeout;
                break;
            }
            // 解码（底层协议为 Ethernet 所以使用 EtherProtocol 作为推测类型）
            IRecordSetValue pingResponse = (IRecordSetValue) ProtocolUtils
                    .decode(resData, 0, len, module
                            .getType("EtherProtocol"));
            // 打印报文
            if (dumpPacket) {
                IValueDumper.INSTANCE.dump(pingResponse);
            }
            // 取得花费时间
            totalTime = System.currentTimeMillis() - startTime;
            // 从 Icmpv4Protocol 中取出 EchoReply 字段
            IRecordSetValue echoReply = (IRecordSetValue) pingResponse
                    .findField("payload/ipv4/payload/icmpv4/echoReply");
            if (echoReply != null) {
                // 判断 EchoReply 的标识符和序号是否与 EchoRequest 的一致
                if (((IIntegerValue) echoReply.getField("identifier"))
                        .getInteger() != icmpId
                        || ((IIntegerValue) echoReply
                                .getField("sequenceNumber"))
                                .getInteger() != seqNo) {
                    continue;
                }
                reachedDst = true;
            }
            if (echoReply != null
                    || pingResponse
```

```
                                        .findField("payload/ipv4/payload/icmpv4/timeExceeded")  !=
null) {
                        // 取得目标设备的 IP 地址
                        IRecordSetValue ipv4Header = (IRecordSetValue) pingResponse
                                .findField("payload/ipv4/header");
                        String tempIp = ipv4Header.getField("sourceAddress")
                                .getTextAs(IValueTextStyle.IPV4_ADDRESS);
                        String newTtl = ipv4Header.getField("timeToLive")
                                .getText();
                        // 打印检验结果
                        String msg = "(" + i + ") from " + tempIp + ", "
                                + totalTime + "(ms), " + " ttl=" + newTtl + ".";
                        System.out.println(msg);
                        break;
                    }
                }
                // 判断是否超时
                if (totalTime >= timeout) {
                    System.out.println("timeout");
                }
                // 判断是否到达了目标点
                if (!reachedDst) {
                    continue;
                } else {
                    break;
                }
            }
            // 打印统计信息
            String msg = "Traceroute completed. ";
            if (reachedDst) {
                msg += dstIp + " reached.";
            } else {
                msg += dstIp + " didn't reach.";
            }
            System.out.println(msg);

        } finally {
            // 关闭套接字
            socket.close();
        }

    }

...

}
```

7.4. TCP 协议连接的模拟

模拟 TCP 协议连接的三向握手开始和终了。使用用 EtherProtocol、IPv4Protocol 和 TcpProtocol 类型来实现 TcpConnectionSample 类，然后执行该类来模拟 TCP 的连接。

- TcpConnectionSample 类的使用方法例：

```
//网关 IP 地址
```

```
String gatewayIp = "192.168.11.1";
//目标 IP 地址
String dstIp = "64.233.189.147";
//生成 Pcap 套接字来实现链路层通信
IPcapSocket socket = IPcapSocketFactory.INSTANCE.createPcapSocket();
socket.setDevice(getInterface().name);
//生成 Arp 对象
ArpApp arp = new ArpApp(socket, LOCAL_MAC, LOCAL_IP);
//设置是否打印数据通信内容
arp.setDumpPacket(true);
//取得目标设备的 MAC 地址
String dstMac = arp.askMac(gatewayIp);
//生成 Connection 对象
TcpConnectionSample conn = new TcpConnectionSample(socket, LOCAL_MAC, dstMac, LOCAL_IP);
//设置是否打印数据通信内容
conn.setDumpPacket(true);
if (!conn.openConnection(dstIp, 12923, 80)) {
        assert false;
}
conn.closeConnection();
```

- TCP 报文的打印例：

```
0:58   EtherProtocol.EtherProtocol
0:6      dstAddress              00:16:01:15:A4:88
6:6      srcAddress              00:16:D4:17:25:C4
12:2     etherType               2048 IPv4
14:44    payload
14:44      ipv4                  Ipv4Protocol.Ipv4Protocol
14:44 Ipv4Protocol.Ipv4Protocol
14:20    header
14:1       0000----              4 version 0xF0
14:1       ----0000              5 headerLength 0x0F
15:1       typeOfService         '00000000'B
16:2       totalLength           44
18:2       identification        1
20:1       0---------------      0 reserved 0x8000
20:1       -0--------------      false doNotFrag 0x4000
20:1       --0-------------      false moreFrags 0x2000
20:2       ---0000000000000      0 fragmentOffset 0x1FFF
22:1       timeToLive            255
23:1       protocol              6 TCP: Transmission Control Protocol
24:2       hcs                   '0000'O
26:4       sourceAddress         192.168.11.5
30:4       destinationAddress 64.233.189.147
34:0       options               omit
34:24    payload
34:24      tcp                   TcpProtocol.TcpProtocol
34:24 TcpProtocol.TcpProtocol
34:24    header
34:2       sourcePort            12923
36:2       destinationPort       80 WWW-HTTP
38:4       sequenceNumber        25372
```

```
42:4        ackNumber              0
46:1        0000------------     6 dataOffset 0xF000
46:2        ----000000------     0 reserved 0x0FC0
47:1        ----------0-----     false urgent 0x0020
47:1        -----------0----     false ack 0x0010
47:1        ------------0---     false push 0x0008
47:1        -------------0--     false reset 0x0004
47:1        --------------0-     true syn 0x0002
47:1        ---------------0     false fin 0x0001
48:2        windowSize             0
50:2        checksum               '0000'O
52:2        urgentPointer          0
54:4        options
54:4          [0]
54:4            maxSs
54:1              optionCode     2 Maximum Segment Lifetime
55:1              optionLength 4
56:2              value          1460
58:0        padding                ''O
58:0      payload                  omit
58:0      fcs                      omit
```

- TcpConnectionSample 类的源代码：

```
/**
 * 模拟 TCP 协议连接的三向握手开始和终了。
 *
 * @author JiangMin
 *
 */
public class TcpConnectionSample {
    protected static IModule module = IModuleRegistry.INSTANCE
            .resolveModule("EtherProtocol");
    protected static IFactory factory = module.getFactory();
    protected static IValueSerializer ser = IValueSerializer.INSTANCE;

    protected String srcMac;
    protected String dstMac;
    protected String srcIp;
    protected IPcapSocket socket;
    protected int timeout = ICommonSocket.DEFAULT_TIMEOUT;
    protected boolean dumpPacket;
    protected String dstIp;
    protected int srcPort;
    protected int dstPort;

    private long lastRemoteSeqNo;
    private long currentLocalSeqNo;

    /**
     * 生成用于发送的 TCP 数据包。协议栈为 Ethernet/IPv4/TCP。
     * @return TcpProtocol 类型的 Record 对象
     */
    protected IRecordSetValue createTcpPacket() throws MetaException {
        // Ethernet
```

```
        IRecordSetValue etherPrtl = (IRecordSetValue) factory
                .createValue("EtherProtocol");
        etherPrtl.getField("dstAddress").setTextAs(dstMac, IValueTextStyle.MAC_ADDRESS);
        etherPrtl.getField("srcAddress").setTextAs(srcMac, IValueTextStyle.MAC_ADDRESS);
        etherPrtl.getField("etherType").setText("0x0800");
        // Ipv4
        IRecordSetValue ipv4Prtl = (IRecordSetValue) etherPrtl.findField("payload/ipv4",
                true);
        IRecordSetValue header = (IRecordSetValue) ipv4Prtl.getField("header");
        header.getField("identification").setText("1");
        header.getField("timeToLive").setText("255");
        header.getField("protocol").setText("6");
        header.getField("sourceAddress").setTextAs(srcIp, IValueTextStyle.IPV4_ADDRESS);
        header.getField("destinationAddress").setTextAs(dstIp,
                IValueTextStyle.IPV4_ADDRESS);
        // Tcp
        IRecordSetValue tcpPrtl = (IRecordSetValue) ipv4Prtl.findField("payload/tcp",
                true);
        ((IIntegerValue) tcpPrtl.findField("header/sourcePort")).setInteger(srcPort);
        ((IIntegerValue) tcpPrtl.findField("header/destinationPort")).setInteger(dstPort);
        return etherPrtl;
    }

    /**
     * 构造函数
     * @param socket PCAP 套接字
     * @param srcMac 源设备的 MAC 地址
     * @param dstMac 目标设备的 MAC 地址
     * @param srcIp 源设备的 IP 地址
     */
    public TcpConnectionSample(IPcapSocket socket, String srcMac, String dstMac,
            String srcIp) {
        this.socket = socket;
        this.srcMac = srcMac;
        this.dstMac = dstMac;
        this.srcIp = srcIp;
    }

    /**
     * 使用三向握手的连接建立,非三向的握手将返回失败。
     * @param dstIp
     * @param localPort
     * @param remotePort
     * @return
     * @throws Exception
     */
    public boolean openConnection(String dstIp, int localPort, int remotePort)
            throws Exception {
        srcPort = localPort;
        dstPort = remotePort;
        this.dstIp = dstIp;
        socket.setTimeout(timeout);
        // 打开套接字
        socket.open();
```

```
        // 设置 Pcap 套接字的过滤条件。（过滤条件的书写规则可以参考 WinPcap 的用户手册）
        socket.setFilter("ip dst " + srcIp + " and ip src " + dstIp
                + " and tcp dst port " + srcPort + " and tcp src port " + dstPort, true);
        // 生成随机的 TCP 序号
        currentLocalSeqNo = (int) (Math.random() * 0xFFFF);

        // send syn
        IRecordSetValue etherPrtl = createTcpPacket();
        IRecordSetValue tcpPrtl = (IRecordSetValue) etherPrtl
                .findField("payload/ipv4/payload/tcp");
        ((IIntegerValue) tcpPrtl.findField("header/sequenceNumber"))
                .setLong(currentLocalSeqNo);
        ((IBooleanValue) tcpPrtl.findField("header/syn")).setBoolean(true);
        // Maximum Segment Size Option
        tcpPrtl.findField("header/options[0]/maxSs/value", true).setText("1460");
        encodeAndWrite(etherPrtl);

        // receive syn + ack
        etherPrtl = readAndDecode();
        tcpPrtl = (IRecordSetValue) etherPrtl.findField("payload/ipv4/payload/tcp");
        boolean reset = ((IBooleanValue) tcpPrtl.findField("header/reset")).getBoolean();
        lastRemoteSeqNo = ((IIntegerValue) tcpPrtl.findField("header/sequenceNumber"))
                .getLong();
        boolean syn = ((IBooleanValue) tcpPrtl.findField("header/syn")).getBoolean();
        if (reset || !syn) {
            return false;
        }

        // send ack
        etherPrtl = createTcpPacket();
        tcpPrtl = (IRecordSetValue) etherPrtl.findField("payload/ipv4/payload/tcp");
        ((IIntegerValue) tcpPrtl.findField("header/sequenceNumber"))
                .setLong(currentLocalSeqNo);
        ((IIntegerValue) tcpPrtl.findField("header/ackNumber"))
                .setLong(lastRemoteSeqNo + 1);
        ((IBooleanValue) tcpPrtl.findField("header/ack")).setBoolean(true);
        ((IIntegerValue) tcpPrtl.findField("header/windowSize")).setInteger(65535);
        encodeAndWrite(etherPrtl);

        return true;
    }

    private IRecordSetValue readAndDecode() throws Exception {
        IRecordSetValue etherPrtl;
        byte[] resData = new byte[1514];
        int len = socket.read(resData);
        // 解码（底层协议为 Ethernet 所以使用 EtherProtocol 作为推测类型）
        etherPrtl = (IRecordSetValue) ProtocolUtils.decode(resData, 0, len, module
                .getType("EtherProtocol"));
        // 打印报文内容
        if (dumpPacket) {
            IValueDumper.INSTANCE.dump(etherPrtl);
        }
        return etherPrtl;
```

```
    }

    private void encodeAndWrite(IRecordSetValue etherPrtl) throws EncodeException,
            IOException {
        // 对报文数据包内的可计算字段进行自动计算(这里是 IP 协议的 HeaderLength 和 TotalLength
和
        // TCP 协议的 dataOffset 等字段)
        ser.calculate(etherPrtl);
        // 打印报文内容
        if (dumpPacket) {
            IValueDumper.INSTANCE.dump(etherPrtl);
        }
        // 编码
        byte data[] = ser.encode(etherPrtl, false);
        // 发送编码数据
        socket.write(data);
    }

    /**
     * 使用三向握手的连接终止, 如果是半关闭将直接终止。
     * @return
     * @throws Exception
     */
    public void closeConnection() throws Exception {
        // send RST
        IRecordSetValue etherPrtl = createTcpPacket();
        IRecordSetValue tcpPrtl = (IRecordSetValue) etherPrtl
                .findField("payload/ipv4/payload/tcp");
        ((IIntegerValue) tcpPrtl.findField("header/sequenceNumber"))
                .setLong(++currentLocalSeqNo);
        ((IIntegerValue) tcpPrtl.findField("header/ackNumber"))
                .setLong(lastRemoteSeqNo + 1);
        ((IBooleanValue) tcpPrtl.findField("header/fin")).setBoolean(true);
        encodeAndWrite(etherPrtl);

        // receive RST+ACK
        etherPrtl = readAndDecode();
        tcpPrtl = (IRecordSetValue) etherPrtl.findField("payload/ipv4/payload/tcp");
        boolean fin = ((IBooleanValue) tcpPrtl.findField("header/fin")).getBoolean();
        lastRemoteSeqNo = ((IIntegerValue) tcpPrtl.findField("header/sequenceNumber"))
                .getLong();

        if (fin) {
            // send ACK
            etherPrtl = createTcpPacket();
            tcpPrtl = (IRecordSetValue) etherPrtl.findField("payload/ipv4/payload/tcp");
            ((IIntegerValue) tcpPrtl.findField("header/sequenceNumber"))
                    .setLong(currentLocalSeqNo);
            ((IIntegerValue) tcpPrtl.findField("header/ackNumber"))
                    .setLong(lastRemoteSeqNo + 1);
            ((IBooleanValue) tcpPrtl.findField("header/ack")).setBoolean(true);
            encodeAndWrite(etherPrtl);
        }
        // 关闭套接字
```

```
        socket.close();
    }

...

}
```

7.5. 模拟 HTTP 的 GET 命令

使用 HTTP 协议的 GET 命令来取得网站的资源.通过发送 HTTP Reqeust 报文，然后接收 HTTP Response 报文，从而获得所要得资源。通过执行 HttpGetSample 类的 doGetAction 来执行例子。

- HttpGetSample 类的使用方法

```
//目标主机地址
String remoteHost = "www.google.com";
//目标端口
int remotePort = 80;
HttpGetSample sample = new HttpGetSample();
sample.setDumpPacket(true);
//执行 HTTP GET 动作（取得 URI 为/的资源，超时时间为 1 分钟）
IValue response = sample.doGetAction(remoteHost, remotePort, "/", 60000);
//取得应答报文中的数据
assert "200".equals(response.findField("response/responseLine/status").getText());
IValue body = response.findField("response/message/httpBody");
//设置是否打印数据通信内容
IValueDumper.INSTANCE.dump(body);
byte[] bodyData = IValueSerializer.INSTANCE.encode(body, false);
System.out.println(new String(bodyData));
```

- 上述代码的执行结果例

上述代码执行后 HTTP 应答报文内的数据部分将会以文本形式被打印在标准输出中，详细内容这里省略。

- HTTP Request 报文打印例

```
0:247   HttpProtocol.HttpProtocol
0:247     request
0:16        requestLine
0:3           method      "GET"
3:1           space1      " "
4:1           uri         "/"
5:1           space2      " "
6:8           version
6:5             prefix    "HTTP/"
11:3            number    "1.1"
14:2          crlf        '0D0A'O
16:231      message
16:229        headers
16:13           [0]
16:6              name    "Accept"
22:1              colon ":"
23:1              space " "
24:3              value "*/*"
27:2              crlf    '0D0A'O
29:24           [1]
```

```
29:15                name    "Accept-Language"
44:1                 colon ":"
45:1                 space " "
46:5                 value "en_US"
51:2                 crlf    '0D0A'O
53:26              [2]
53:15                name    "Accept-Encoding"
68:1                 colon ":"
69:1                 space " "
70:7                 value "deflate"
77:2                 crlf    '0D0A'O
79:120             [3]
79:10                name    "User-Agent"
89:1                 colon ":"
90:1                 space " "
91:106               value {}
         "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1
         ; SV1; GTB6; .NET CLR 2.0.50727; .NET CLR 1.1.4322
         ; CIBA)"
197:2                crlf    '0D0A'O
199:22             [4]
199:4                name    "Host"
203:1                colon ":"
204:1                space " "
205:14               value "www.google.com"
219:2                crlf    '0D0A'O
221:24             [5]
221:10               name    "Connection"
231:1                colon ":"
232:1                space " "
233:10               value "Keep-Alive"
243:2                crlf    '0D0A'O
245:2          emptyLine '0D0A'O
247:0          httpBody    omit
247:0          trailers    omit
```

- HttpGetSample 类源代码

```
/**
 * 使用 HTTP 协议的 GET 命令来取得网站的资源.通过发送 HTTP Reqeust 报文，然后接收 HTTP Response 报文，
 * 从而获得所要得资源。
 *
 * @author JiangMin
 *
 */
public class HttpGetSample {
    private static String USER_AGENT =
    "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; GTB6; .NET CLR 2.0.50727; .NET CLR 1.1.4322; CIBA)";
    private static IValueSerializer ser = IValueSerializer.INSTANCE;
    private static IModule module = IModuleRegistry.INSTANCE.resolveModule("HttpProtocol");
    private static IFactory factory = module.getFactory();
    private static IType httpPrtlType =    module.getType("HttpProtocol");
```

```
    private boolean dumpPacket;

    /**
     * 执行 HTTP1.1 的 GET 动作，返回 HTTP 应答报文。
     * @param remoteHost 目标主机名或 IP 地址
     * @param remotePort 目标主机的端口号
     * @param httpUri 请求资源的 URI
     * @param timeout 读取数据的超时时间，毫秒单位。
     * @return 解码后的应答报文对象。
     */
    public IValue doGetAction(String remoteHost, int remotePort, String httpUri, int timeout)
    throws MetaException, IOException, ExecutionException, InterruptedException {
        //生成 Get Request 报文
        IValue prtl = factory.createValue("HttpProtocol");
        IValue request = prtl.findField("request", true);
        //uri
        request.findField("requestLine/method").setText("GET");
        request.findField("requestLine/uri").setText(httpUri);
        IValue message = request.findField("message");
        //headers
        setHeader(message, 0, "Accept", "*/*");
        setHeader(message, 1, "Accept-Language", "en_US");
        setHeader(message, 2, "Accept-Encoding", "deflate");
        setHeader(message, 3, "User-Agent", USER_AGENT);
        setHeader(message, 4, "Host", remoteHost);
        setHeader(message, 5, "Connection", "Keep-Alive");

        //编码
        byte[] data = ser.encode(prtl, false);
        if (dumpPacket) {
            IValueDumper.INSTANCE.dump(prtl);
        }

        Socket socket = new Socket();
        socket.connect(new InetSocketAddress(remoteHost, remotePort));
//          socket.setSoTimeout(1000);
        OutputStream outStm = socket.getOutputStream();
        final InputStream inStm = socket.getInputStream();
        final IEnsurableBitBuffer buffer =
IBitBufferFactory.INSTANCE.createEnsurableBuffer();
        buffer.setEnsureTimeout(timeout);
        //发送 Request
        outStm.write(data);
        //接收数据
        ExecutorService service = Executors.newSingleThreadExecutor();
        Future<Integer> recvJob = service.submit(new Callable<Integer>() {

            @Override
            public Integer call() throws Exception {
                byte[] recvData = new byte[1024];
                try {
                    int len = inStm.read(recvData);
                    while(len >= 0) {
//                      System.out.println("read:"+len);
```

```
                        //写数据
                        buffer.putByte(recvData, 0, len);
                        //读数据
                        len = inStm.read(recvData);
                    }
                } catch (IOException e) {
                    //忽略 Socket 关闭情况
                    if (!"socket closed".equals(e.getMessage())) {
                        throw e;
                    }

                }
                //EOF
                buffer.setEof();
                return 1;
            }
        });

        //增量解码
        IValue response = ProtocolUtils.incrementDecode(buffer, httpPrtlType);
        if (dumpPacket) {
            IValueDumper.INSTANCE.dump(response);
        }
        //关闭 Socket
        socket.close();
        //等待接收任务结束
        recvJob.get();
        //停止线程池
        service.shutdown();
        //返回应答报文
        return response;
    }

    private void setHeader(IValue message, int index, String name, String value) throws
MetaException {
        IContainerValue header = (IContainerValue)message.findField(
                "headers["+index+"]", true);
        header.getField("name").setText(name);
        header.getField("value").setText(value);
    }

    public boolean isDumpPacket() {
        return dumpPacket;
    }

    public void setDumpPacket(boolean dumpPacket) {
        this.dumpPacket = dumpPacket;
    }
}
```