<<深入理解 OpenFast 架构>>

莫大 著

目录

第1章. 引言

1.1 版本说明

本书中使用的 FAST 规范版本是 1.1，FIX 协议是 5.2，OpenFast 是 1.1.2。

1.2 章节编排

本书主要有四部分组成，（1）FIX、FAST 协议基本介绍；（2）FAST 协议规范分析；（3）OpenFast 设计与实现分析；（4）FAST 行情分析示例。

第一部分“FIX\FAST 协议基本介绍”

对于 FIX、FAST 协议的基本规范和框架进行介绍。

第二部分“FAST 协议规范分析”

对 FAST 协议中字段指令、数据类型、数据编码规范等进行分析。

第三部分“OpenFast 设计与实现分析”

结合 FAST 协议，对于 OpenFast 中相关协议内容、流程、方法的设计与实现进行分析。

第四部分“FAST 行情分析示例”

基于欧洲期货交易所的行情数据，直接手工对二进制 FAST 行情数据进行解析。

1.3 行文规定

代码规则

- 本书使用 Java 语言来展示代码：
- 在分析 FAST 协议时，笔者基于 OpenFast 中给出的测试用例做了一定地修改；
- 在描述 OpenFast 设计与实现时，通常会粘帖 Java 代码。

第一部分

FIX/FAST 协议基本介绍

第2章. FIX 协议介绍

FIX（The Financial Information eXchange Protocol）金融信息交换协议是一种标准，定义了传输协议以及消息格式【1】。通过采用 FIX 协议，减少了各机构之间传输接口的多样性和复杂性，目前已在证券、银行等金融企业中有着非常广泛的使用。

2.1 协议概述

作为一个证券金融行业主要的通信协议，FIX 与其他订单处理、交易处理系统进行了融合。主要分为两层：会话层和应用层。会话层主要是负责传输数据，而应用层则是处理相关的业务数据。与之对应的消息定义也分为两部分：管理消息和应用消息。管理消息主要包括：登录、退出、心跳、数据请求等消息。应用消息主要包括：交易消息如公告、订单、证券价格等消息。

FIX 协议消息主要有三部分组成：消息头部+消息体+消息尾部组成。其中各部分组成如下：

- 消息头部：消息类型、消息长度、接收方/发送方名称、消息编号等；
- 消息体：包括会话层、应用层的消息；
- 消息尾部：校验和或者可选的签名。

各消息之间通过分割符进行划分。分割符号定义为：SOH (Start of header – ASCII 0x01)。

2.2 FIX 消息定义语法

每个消息由多个字段组成，FIX 协议通过特定语法对字段名称和对应的字段值进行定义。

2.2.1 标签-值语法

字段标签-值定义语法为：<Tag>=<Value>。Tag 为字段名称，Value 为该字段对应的值，比如：

Price=100.5 定义了字段“Price”的值为 100.5。

每个消息通常由多个字段组成，每个字段之间通过分隔符（SOH）进行分割。消息示例如下：

```
8=FIX.5.0^9=251^35=D^49=AFUNDMGR^56=ABROKER^34=2^52=20030615-
01:14:49^11=12345^1=111111^63=0^64=20030621^21=3^110=1000^111=50000^55=IBM
^48=459200101^22=1^54=1^60=2003061501:14:49
38=5000^40=1^44=15.75^15=USD^59=0^10=127
```

其中，“^”表示了分隔符 SOH。

FIX 协议委员会根据不同的字段业务用途，对于 Tag 编号进行了预先分配，各交易所再根据自身业务的特点对自动进行扩充，定义其他的字段编号。

2.2.2 FIXML 语法

FIXML 语法通过 XML 定义和描述 FIX 消息，示例如下：

```
<FIXML v="4.4" r="20030618" s="20040109">
   <Order ClOrdID="123456" Side="2" TransactTm="2001-09-11T09:30:47-05:00"
OrdTyp="2" Px="93.25" Acct="26522154">
     <Instrmt Sym="IBM" ID="459200101" IDSrc="1"/>
     <OrdQty Qty="1000"/>
   </Order>
</FIXML>
```

但是通过 XML 语言进行定义，字段定义和标识所需要的编码长度较长，因此在实际中 FIXML 语法定义使用并不广泛。

2.2.3 重复组

FIX 协议允许一组字段在重复组（repeated group）内重复出现。示例如下：

```
384=2<SOH>372=6<SOH>385=R<SOH>372=7<SOH>385=R<SOH>
```

表示了包含了两个重复实体的重复组，tag 为 372 的字段为重复组中的第一个字段。

FIX 消息定义中重复组是非常重要的概念，比如：订单消息，一个 FIX 消息中可能需要传输多个订单消息，而订单字段的定义是相同的。

2.3 FIX 协议数据类型

FIX 协议根据传输字段的不同类型定义了对应的数据类型（Data Type），主要类型有：

数据类型	类型名称	字段应用	字段说明
int	整数类型	Length	长度字段
		NumInGroup	重复组个数
		SeqNum	消息序号
		TagNum	标签序号
float	浮点数类型	Price	价格
		Qty	交易量

		§ PriceOffset	价格差异
		Amt	交易金额
char	字符类型	Boolean	bool 值
string	字符串类型	Currency	币种
		Exchange	交易所
data	原始数据类型	无特定格式限制	其他数据

2.4 FIX 消息示例

FIX 协议根据不同的业务类型定义了对应的 FIX 消息：比如：市场数据请求消息（MARKET DATA REQUEST）、证券定义消息（SECURITY DEFINITION）等。而各交易所根据自身的业务特定，也可能扩充了其他字段和消息类型。比如：指数统计信息的消息类型【2】，消息定义为：

消息编号	消息名称	消息描述
279	MDUpdateAction	最新值
269	MDEntryType	指数值
83	RptSeq	序号
48	SecurityID	指数 ID
22	SecurityIDSource	指数数据源
207	SecurityExchange	指数所在交易所
272	MDEntryDate	日期
273	MDEntryTime	时间
270	MDEntryPx	当前值
274	TickDirection	指数变动方向

通常采用 SecurityID、SecurityIDSource、SecurityExchange 这三个组成了指数的标识符。

针对上述的 FIX 消息定义采用<Tag>=<Value>语法，消息示例如下：

```
279=0^269=3^83=1^48=NASDAQ^22=NYS^207=NYS^272=20140730^273=09:14:49^270=12
34.123^274=0
```

该消息中一共 10 个字段，通过<Tag>=<Value>方式设置了消息中各字段的值。我们可以通过网站【3】对该消息进行解析，得到结果如下：

标签编号	原始值	标签名称	解析值
279	0	MDUpdateAction	0 - New
269	3	MDEntryType	3 - IndexValue
83	1	RptSeq	1

48	NASDAQ	SecurityID	NASDAQ
22	NYS	SecurityIDSource	NYS
207	NYS	SecurityExchange	NYS
272	20140730	MDEntryDate	20140730
273	09:14:49	MDEntryTime	09:14:49
270	1234.123	MDEntryPx	1234.123
274	0	TickDirection	0 - PlusTick

需要说明的是：

（1）该网站的输入中字段之间的分隔符为“|”，输入时需要将之前 FIX 消息中的“^”进行替换；

（2）该网站仅作字段解析，并没有对于 FIX 消息格式规则和顺序进行检查，

通常在实际数据传输业务中，一个 FIX 消息中包含了多个重复组，假设市场数据 FIX 消息定义如下：

标签编号	标签名称	标签描述
279	MDUpdateAction	证券数据的更新方式
269	MDEntryType	证券数据类型
83	RptSeq	序号
273	MDEntryTime	时间
271	MDEntrySize	证券数量
270	MDEntryPx	当前价格
48	SecurityID	证券 ID
22	SecurityIDSource	证券数据源
1020	TradeVolume	成交量总和

消息示例如下：

```
35=x|268=2|279=0|269=2|270=9462.50|271=5|48=800123|22=8|1020=10|279=0|269=0|270=9462.00|271=175|1023=1|48=800123|22=8|1020=15
```

通过之前网站对于该消息进行解析，得到结果如下：

标签编号	原始值	标签名称	解析值
35	X	MsgType	x - SecurityListRequest
268	2	NoMDEntries	2
279	0	MDUpdateAction	0 - New
269	2	MDEntryType	2 - Trade
270	9462.50	MDEntryPx	9462.50
271	5	MDEntrySize	5
48	800123	SecurityID	800123

22	8	SecurityIDSource	8 - ExchangeSymbol
346	10	NumberOfOrders	10
279	0	MDUpdateAction	0 - New
269	0	MDEntryType	0 - Bid
270	9462.00	MDEntryPx	9462.00
271	175	MDEntrySize	175
1023	1	MDPriceLevel	1
48	800123	SecurityID	800123
22	8	SecurityIDSource	8 - ExchangeSymbol
346	15	NumberOfOrders	15

字段 NoMDEntries（tag=268）表示该 FIX 消息中的重复组，本消息中一共有 2 个重复组。

2.5 FIX 消息字段顺序规则

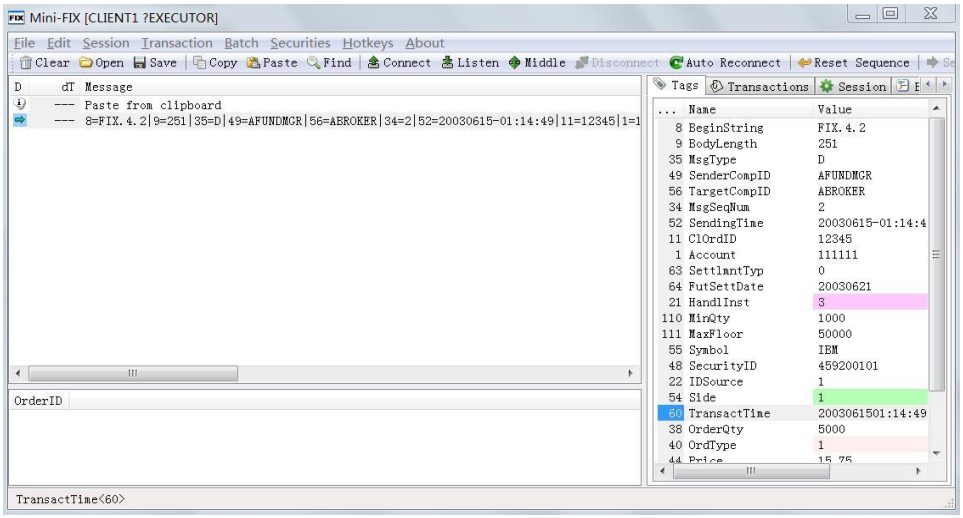
FIX 消息中的字段通过<tag>=<value>进行表示，而字段在消息中的位置可以是任意的，但是需要遵守以下几个规则：

- 消息由消息头部、消息体、消息尾部组成，这三者的顺序不能改变；
- 消息头部开始的三个字段为：BeginString（tag 为 8），BodyLength（tag 为 9），MsgType（tag 为 35）；
- 消息尾部的最后一个字段为 CheckSum（tag 为 10）；
- 一个字段只出现一次。

消息示例如下：

```
8=FIX.4.2^9=251^35=D^49=AFUNDMGR^56=ABROKER^34=2^52=20030615-
01:14:49^11=12345^1=111111^63=0^64=20030621^21=3^110=1000^111=50000^55=IBM
^48=459200101^22=1^54=1^60=2003061501:14:49^38=5000^40=1^44=15.75^15=USD^5
9=0^10=127
```

可以通过 minfix 工具【3】进行验证，minfix 工具显示结果如下：



该工具对于 FIX 消息中各字段按照“标签=值”的方式进行解析，得到结果如下：

标签编号	原始值	标签名称	解析值
8	FIX.4.2	BeginString	FIX.4.2
9	251	BodyLength	251
35	D	MsgType	D-NewOrderSingle
49	AFUNDMGR	SenderCompID	AFUNDMGR
56	ABROKER	TargetCompID	ABROKER
34	2	MsgSeqNum	2
52	20030615-01:14:49	SendingTime	20030615-01:14:49
11	12345	ClOrdID	12345
1	111111	Account	111111
63	0	SettlmntTyp	0-Regular
64	20030621	FutSettDate	20030621
21	3	HandlInst	3-ManualOrder
110	1000	MinQty	1000
111	50000	MaxFloor	50000
55	IBM	Symbol	IBM
48	459200101	SecurityID	459200101
22	1	IDSource	1-CUSIP
54	1	Side	1–Buy
60	2003061501:14:49	TransactTime	2003061501:14:49
38	5000	OrderQty	5000

40	1	OrdType	1-Market
44	15.75	Price	15.75
15	USD	Currency	USD
59	0	TimeInForce	0–Day
10	127	CheckSum	127

2.6 重点回顾

- FIX 协议是一种数据传输协议；
- 主要通过“标签=值”的方式定义；
- FIX 规定了消息字段顺序。

2.7 参考资料

【1】http://www.fixtradingcommunity.org/

【2】UMDF_MarketDataSpecification_v2.0.13.pdf

【3】http://fixdecoder.com/fix_decoder.html

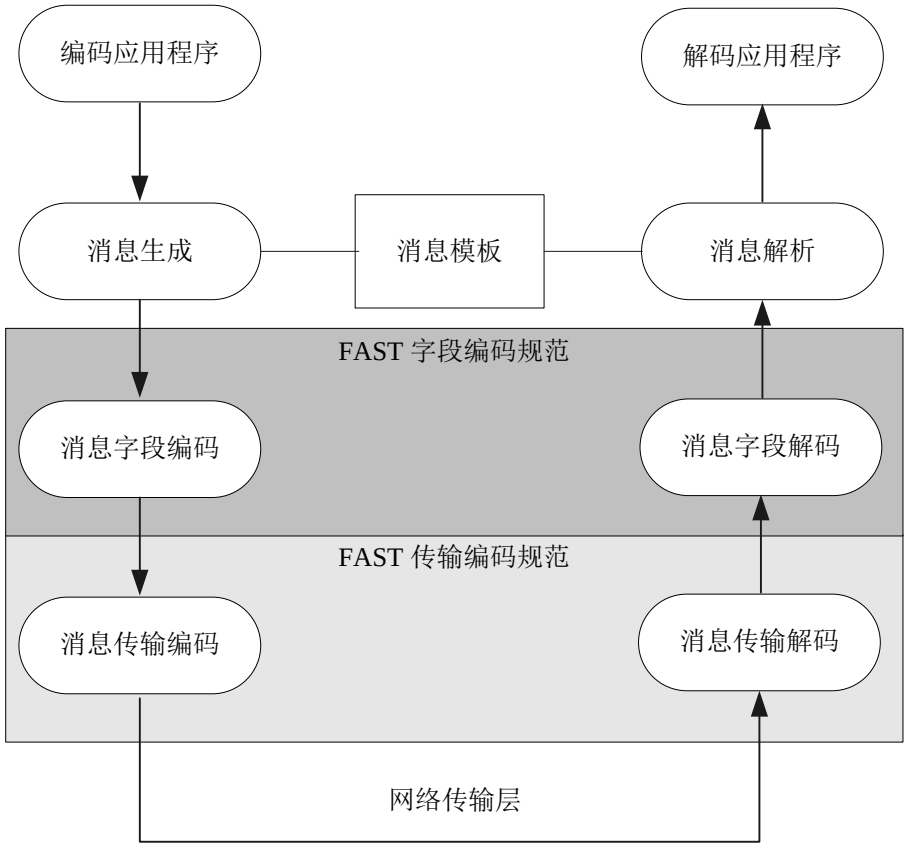
【4】http://elato.se/minifix/

第3章. FAST 协议介绍

FAST 是 FIX Adapted for Streaming 的缩写，是一种面向消息数据流的二进制编码方法。本章对于 FAST 协议基本内容进行介绍。

3.1 FAST 协议基本框架

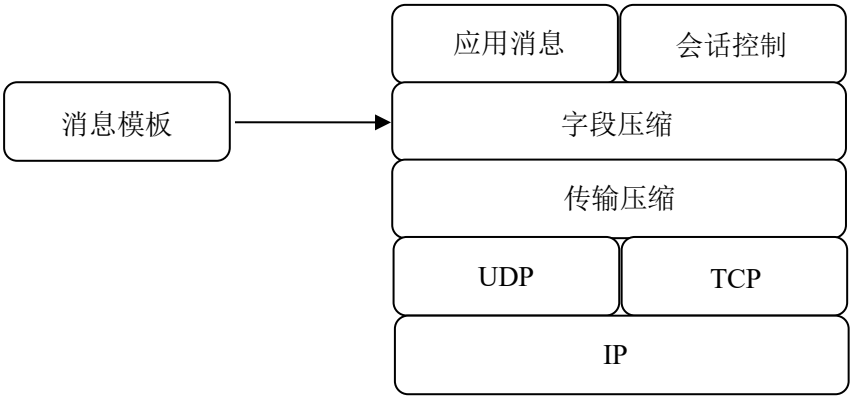
FAST 协议基本框架【1】如下：



主要有三方面组成：

- 编解码应用双方基于商定的消息模板生成需要传输的消息；
- 根据字段相邻的值，对字段进行编解码；
- 传输过程中对于字段值进行编解码。

协议应用框架如下：



FAST 协议通过消息模板对 FIX 协议中定义的消息类型进行描述，服务端通过该模板进行数据的压缩，然后将压缩之后的数据通过 UDP 或者 TCP 协议进行传输，客户端接收到数据之后基于模板对于压缩的数据进行解析。

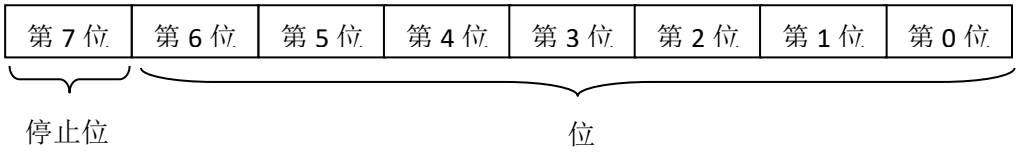
FAST 协议主要从两个方面对于数据进行压缩：

- 数据本身的压缩，尽可能减少数据的冗余；
- 数据传输过程中的压缩，减少传输过程中的数据冗余。

一个 FAST 消息基本结构如下：

3.2 停止位编码方式

FAST 协议为了减少字段值本身的冗余，采用了停止位编码方式（Stop Bit Encoding）。一个停止位编码对应一个字节序列，每个字节序列的最高有效位为停止位，当 Stop Bit 为 0 时，表示下一字节元组属于同一个字节序列；如果 Stop bit 位为 1，则标识该字节序列的结束。即：每个字节通过 7 个 bit 位表示需要传输的数据，字节的最高有效位标识了序列是否结束。



除了停止位之外，剩余的 7 个 bit 是有效数据位。实体值通过连接每一个字节的有效数据位进行表示。

对于如下序列：

0	0	0	0	0	1	1	1	**1**	1	1	0	1	0	0	0

该字节序列一共有两个字节组成，第一个字节 00000111 的最高有效为 0，表示下一个字节属于同一个序列，第二个字节为 11101000，最高有效位为 1，表示该字节为字节序列的最后一个字节。

3.3 整数变长压缩算法

FAST 协议中对整数采用了变长压缩方法，根据整数的大小确定其编码所需要的字节数目。算法基本步骤为：

- 根据整数大小，获得表示该整数所需要的字节个数；
- 将整数的二进制 bit 位，按照 7bit 进行划分；
- 设置每个字节的最高有效位；
- 得到表示整数的压缩字节。

比如 32 位整数在 Java int 类型中占用了 4 个字节，如果数值为 1，二进制编码为：

00000000 00000000 00000000 00000001

而通过停止位编码之后的数值为：10000001，减少了 3 个字节的长度。

3.4 字节位序列

所有的整数类型采用大端（big-endian）方式，即：网络字节序进行表示。对于 32 位整数由四个字节组成，最低字节为字节 1，最高字节为字节 4：

字节 4	字节 3	字节 2	字节 1

对于整数 1024 的二进制表示为：00000111 11011110

FAST 协议在编码时，处理的字节顺序为：00000111 11011110。

3.5 操作符类型

在数据传输层面 FAST 协议对传输前后的字段值进行比较，根据字段所使用的操作进行压缩。FAST 协议定义了以下几种操作符类型：

- 无操作符（None Operator）
  传输前后的消息没有关系，且消息中的数据均需要进行传输；
- 常量操作符（Constant Operator）
  消息字段值为常量，该值不进行传输；

- 默认操作符（default Operator）
  消息中的字段有默认值，只有输入的值与默认值不同才进行传输；
- 拷贝操作符（Copy Operator）
  传输前后消息如果一致，则不需要进行传输；
- 递增操作符（Increment Operator）
  如果传输的后值等于前值递增，则不需要进行传输；
- 差值操作符（Delta Operator）
  仅传输消息中前后值的差值。

3.6 重点回顾

- FAST 协议从两个层面进行数据的压缩；
- 停止位编码方式是压缩算法基础。

3.7 参考资料

【1】A Basic Guide to FAST v1.0.pdf。
【2】FAST Specification 1 x 1.pdf。

第4章. FAST 应用类型

应用类型（application type）在 FAST 协议中表示分组或者消息的类型。主要包含了基础类型（primitive type）、分组（group）、字段（field）、序列（sequence）四种类型。

本章对于 FAST 协议应用类型进行说明的同时，也通过 OpenFast 中对应的类、进行测试和验证。

4.1 类型说明

在 FAST 协议中包含了如下的基本类型（primitive）：整数、ASCII 字符串、UNICODE 字符串、字节向量、十进制数、日期。

每个基本类型涉及到类型的表示以及类型值的编解码，因此在 OpenFast 中主要通过两种类进行描述：

- 基本类型描述类；
- 基本类型值编解码的处理类。

在 OpenFast 中，基本类型通过 org.openfast.ScalarValue 类进行描述。主要有两个静态成员变量定义：

- undefined--值未定义
- NULL—空值，表示值不存在（缺失）

上述两个定义非常重要，之后的章节中会频繁提及。

整数类型、ASCII 字符串、字节向量等类型继承了 ScalarValue 类，根据自身的类型定义了不同的操作方法。

而 org.openfast.template.type.codec.TypeCodec 类描述了基本类型值的编解码处理过程。其中的静态成员变量包含了基本类型的编解码处理对象：

```
    public static final TypeCodec UINT = new UnsignedInteger();
    public static final TypeCodec INTEGER = new SignedInteger();
    public static final TypeCodec ASCII = new AsciiString();
    public static final TypeCodec UNICODE = new UnicodeString();
    public static final TypeCodec BIT_VECTOR = new BitVectorType();
    public static final TypeCodec BYTE_VECTOR = new ByteVectorType();
    public static final TypeCodec SF_SCALED_NUMBER = new
SingleFieldDecimal();
    public static final TypeCodec STRING_DELTA = new StringDelta();
    public static final TypeCodec NULLABLE_UNSIGNED_INTEGER = new
NullableUnsignedInteger();
    public static final TypeCodec NULLABLE_INTEGER = new
NullableSignedInteger();
    public static final TypeCodec NULLABLE_ASCII = new
```

```
NullableAsciiString();
    public static final TypeCodec NULLABLE_UNICODE = new
NullableUnicodeString();
    public static final TypeCodec NULLABLE_BYTE_VECTOR_TYPE = new
NullableByteVector();
    public static final TypeCodec NULLABLE_SF_SCALED_NUMBER = new
NullableSingleFieldDecimal();
    public static final TypeCodec NULLABLE_STRING_DELTA = new
NullableStringDelta();
```

4.2 整数类型

FAST协议中定义了整数类型（Integer Numbers），根据整数符号分为：无符号数、有符号数。整数类型的停止位编码过程为：

- 根据整数值的范围确定表示该整数需要的字节数目；
- 将整数的二进制 bit 位，按照 7bit 一组进行划分；
- 设置每个字节的最高有效位；
- 返回编码后的所有字节。

在 OpenFast 中，org.openfast.IntegerValue 表示整数类型，主要的方法有：

方法名称	功能
Decrement	将整数减一
Increment	将整数加一
Subtract	两个整数相减，返回差值
Add	两个整数相减，返回之和

org.openfast.UnsignedInteger、org.openfast.SignedInteger 类对应了整数的编解码处理过程，主要的方法有：

方法名称	功能
encodeValue	对于整数进行编码，返回字节数组
Decode	从数据流中根据停止位编码规则读取字节数组，返回整数值

为了便于处理，OpenFast 在 TypeCodec 类中定义了静态成员变量 INTEGER 和 UINT，对于整数类型的编解码进行处理。

```
    public static final TypeCodec UINT = new UnsignedInteger();
    public static final TypeCodec INTEGER = new SignedInteger();
```

假设整数类型为有符号数，值为 1000，其值编码过程如下：

```
    public void testUnsignedIntegerEncodeDecode() {
        ScalarValue value = new IntegerValue(1000);
        byte[] encoding = TypeCodec.UINT.encode(value);
        String res = ByteUtil.convertByteArrayToBitString(encoding);
```

```
        assertEquals("00000111 11101000", res);
    }
```

其中，ByteUtil.convertByteArrayToBitString【参见 TODO】静态方法将传入的字节数组转换成 8bit 为一组的 bit 流，并通过空格进行分割便于展示和分析。

过程分析如下：

停止位编码字节序列	**0**0000111	**1**1101000
去除停止位后的字节序列	0000111	1101000
整数值二进制序列表示	00001111101000	
整数数值	1000	

需要注意的是，整数编码可能会出现 7 位数值位均为 0 的情况，此时并不能将该字节丢弃。比如：对于整数 64，二进制编码为：01000000，根据停止位编码方式，应该拆分为两个字节，其最高位的“0”，也需要用一个字节进行表示。

编码过程如下：

```
    public void testUnsignedIntegerPrefixEncodeDecode() {
        ScalarValue value = new IntegerValue(64);
        byte[] encoding = TypeCodec.INTEGER.encode(value);
        String res = ByteUtil.convertByteArrayToBitString(encoding);
        assertEquals("00000000 11000000", res);
    }
```

过程分析如下：

停止位编码字节序列	**0**0000000	**1**1000000
去除停止位后的字节序列	0000000	1000000
整数值二进制序列表示	00000001000000	
整数数值	64	

如果将上述例子中整数 64 编码的最高 bit 位’0’直接丢弃，则会被认为是-64。验证分析如下：

```
    public void testMinus64Integer() {
        ScalarValue value = null;
        String interger = "11000000"; // -64的二进制编码
        InputStream in = ByteUtil.createByteStream(interger);
        value = TypeCodec.INTEGER.decode(in);
        assertEquals(-64, value.toInt());


    }
```

其中 ByteUtil.createByteStream 静态方法【参见 TODO】将字符串转换为字节数组。输入的 bit 编码值为"11000000"，解析后得到整数值为-64.

对于负整数，需要进行编码的二进制字节流为对应正整数的补码。假设整数类型为有符号数，值为-1024，编码过程如下：

```
public void testSignedInterEncodeDecode() {
    ScalarValue value = new IntegerValue(-1024);
    byte[] encoding = TypeCodec.INTEGER.encode(value);
    String res = ByteUtil.convertByteArrayToBitString(encoding);
    assertEquals("01111000 10000000", res);
}
```

过程分析如下：

停止位编码字节序列	**0**1111000	**1**0000000
去除停止位后的字节序列	1111000	0000000
整数值二进制表示	11110000000000 最高符号为 1，表示负数	
二进制补码表示	00010000000000=1024	
整数数值	-1024	

4.3 ASCII 字符串类型

FAST 协议中定义了 ASCII 字符串类，其停止位编码过程为：

- ASCII 字符串的实体值为 7bit 的 ASCII 字符序列，设置停止位后直接返回字节数组；
- 如果字符串为空或者值为”\0”等情况，FAST 协议定义了特殊的编码字节数组来表示。

在 OpenFast 中，org.openfast.StringValue 类表示 ASCII 字符串类型。主要的方法有：

方法名称	功能
toByte()	将字符串转换为字节序列
toShort()	将字符串转换为整数
toDouble()	将字符串转换为浮点数

org.openfast.AsciiString 类对应了 ASCII 字符串的编解码处理过程。主要的方法有：

方法名称	功能
encodeValue	对于字符串进行编码，返回字节数组
Decode	从数据流中根据停止位编码规则读取字节数组，返回 ScalarValue 对象
fromString	从字符串返回 StringValue 对象

假设字符串为“CME”，编码过程如下：

```
public void testAsciiStrEncodeDecode() {
    ScalarValue value = new StringValue("CME");
    byte[] encoding = TypeCodec.ASCII.encode(value);
    String res = ByteUtil.convertByteArrayToBitString(encoding);
    assertEquals("01000011 01001101 11000101", res);
}
```

过程分析如下：

停止位编码字节序列	01000011	01001101	11000101
去除停止位编码字节序列	1000011	1001101	1001101
字符串的值	127='C'	77=’M’	69=‘E’

对于空字符串””，以及字符串“\0”，FAST 协议中通过零前导（实体值的 7 个 bit 位均为 0）进行定义。编码过程如下：

```
public void testAsciiStrEncodeDecode() {
    ScalarValue value = new StringValue("");
    byte[] encoding = TypeCodec.ASCII.encode(value);
    String res = ByteUtil.convertByteArrayToBitString(encoding);
    assertEquals("10000000", res);

    value = new StringValue("\0");
    encoding = TypeCodec.ASCII.encode(value);
    res = ByteUtil.convertByteArrayToBitString(encoding);
    assertEquals("00000000 10000000", res);
}
```

如果移除零前导之后，如果剩余的字节序列前 7 个 bit 不全为 0，则为超长错误。相应字符串的编码定义可以参照参照 FAST 规范中的 10.6.3 节，如下所示：

实体值	是否可空	说明
0x00	否	空字符串
0x00 0x00	否	字符串为”\0”
0x00	是	NULL
0x00 0x00	是	空字符串
0x00 0x00 0x00	是	字符串为”\0”

需要注意的是，在上面的表格中 0x00 表示的是一个实体值，为一个字节中除去最高有效位之后的值。

4.4 字节向量类型

FAST 协议中定义了字节向量类型，其停止位编码过程为：

- 计算字节向量的长度；
- 将长度按照整数类型进行编码；
- 将各字节序列直接追加在长度编码之后，得到编码后的字节向量序列。

向量长度逻辑上为 32 位无符号整数。

在openfast中，org.openfast.ByteVectorValue类表示字节向量类型。主要的方法有：

方法名称	功能
Serialize()	将字节向量转换为字符串

org.openfast.ByteVectorType类对应了字节向量的编解码处理过程，主要的方法有：

方法名称	功能
Encode	对于字节向量进行编码，返回字节数组
Decode	从数据流中根据停止位编码规则读取字节向量，返回 ByteVectorValue 对象

假设字节向量为‘abc’，编码过程如下：

```
public void testVetorEncode() {
    ScalarValue value = new ByteVectorValue("abc".getBytes());
    byte[] encoding = TypeCodec.BYTE_VECTOR.encode(value);
    String res = ByteUtil.convertByteArrayToBitString(encoding);
    assertEquals("10000011 01100001 01100010 01100011", res);
}
```

过程分析如下：

停止位编码字节序列	10000011	01100001	01100010	01100011
去除停止位编码字节序列	0000011	01100001	01100010	01100011
字节向量长度编码值	0000011=3			
字节向量的值	01100001=’a’	01100010=’b’	01100011=’c’	

对于字节向量的编码过程，向量字节的值仍然通过一个字节表示，该处理与前述ASCII字符串类型有所不同。

4.5 Unicode 字符串

FAST协议中定义了unicode字符串类型，字符串采用utf-8 编码格式。其停止位编码方式与字节向量相同。

在OpenFast中，org.openfast.StringValue类表示字符串类型。主要的方法有：

方法名称	功能
toByte()	将字符串转换为字节序列
toShort()	将字符串转换为整数
toDouble()	将字符串转换为浮点数

org.openfast.UnicodeString类对应了unicode字符串的编解码处理过程，主要的方法有：

方法名称	功能
encodeValue	对于字符串进行编码，返回字节数组
Decode	从数据流中根据停止位编码规则读取字节数组，返回 StringValue 对象
fromString	从字符串返回 StringValue 对象

假设unicode字符串值为“交易所”，”交”采用utf8 编码格式为：E4BAA4，“易”采用utf8 编码为：E69893，“所”采用utf8 编码为：E68980，一个占用了 9 个字节。

编码过程如下：

```
    public void testUnicodeStrEncodeDecode() {
        ScalarValue value = new StringValue("交易所");
        byte[] encoding = TypeCodec.UNICODE.encode(value);
        String res = ByteUtil.convertByteArrayToBitString(encoding);
        assertEquals("10001001 11100100 10111010 10100100 11100110 10011000
"
                + "10010011 11100110 10001001 10000000", res);
    }
```

过程分析如下：

停止位编码字节序列	10001001	11100100	10111010	10100100	11100110
	10011000	10010011	11100110	10001001	10000000
去除停止位编码字节序列	0001001	11100100	10111010	10100100	11100110
	10011000	10010011	11100110	10001001	10000000
向量长度整数值	10001001=9				
字节向量 unicode 字符	11100100 10111010 10100100 = E4BAA4 （交）				

串的值	11100110 10011000 10010011 = E69893 （易）
	11100110 10001001 10000000 = E68980 （所）

4.5.1 十进制数类型

FAST协议中定义了DecimalValue十进制数类型。采用了指数和尾数的表示方式：number = mantissa * $10^{exponent}$。

其中mantissa为尾数，exponent为指数，底为 10，指数和尾数类型均为带符号的整数。

十进制数停止位编码过程为：

- 将十进制数通过公式进行表示，得到指数和尾数；
- 对指数进行编码；
- 对尾数进行编码；
- 返回指数+尾数的编码序列。

在OpenFast中，org.openfast.DecimalValue类表示十进制数类型。主要的方法有：

方法名称	功能
Subtract	十进制数相减
Add	十进制数相加

org.openfast.SingleFieldDecimal类对应了十进制数的编解码过程，主要的方法有：

方法名称	功能
encodeValue	判断指数是否过长，对于指数、尾数进行编码，返回字节数组
Decode	从数据流中根据停止位编码规则读取尾数和指数，返回 DecimalValue 对象

假设十进制数为 1204.01，编码过程如下：

```
public void testDecimalEncode() {
    ScalarValue value = new DecimalValue(1204.01);
    byte[] encoding = TypeCodec.SF_SCALED_NUMBER.encode(value);
    String res = ByteUtil.convertByteArrayToBitString(encoding);
    assertEquals("11111110 00000111 00101100 11010001", res);
}
```

过程分析如下：

停止位编码字节序列	11111110	00000111	00101100	11010001

去除停止位编码字节序列	1111110	0000111	0101100	1010001
指数二进制及其值	1111110=-2，符号位为 1			
尾数二进制及其值	00001110101100101000 1=102401			
十进制数值	$201401 \times 10^{-2} = 2014.01$			

4.6 字段（field）

字段由名称和类型组成，名称在分组中必须唯一。类型可以为基本类型、序列、分组中的任意一种。

在 OpenFast 中，org.openfast.Field 类描述了字段，其主要的方法有：

方法名称	功能
Encode	对于 group 类型进行编码
Decode	对于读入的 bit 流进行解码
getTemplate	获得 field 定义的模板
isPresenceMapBitSet	该字段对应的 PMAP 是否设置

4.7 分组（group）

分组由无序的字段集合组成。笔者认为对应了 FIX 协议中的重复组（repeating group）。

在 OpenFast 中，org.openfast.Group 类描述了分组，其主要的方法有：

方法名称	功能
encode	对于 group 类型进行编码
decode	对于读入的 bit 流进行解码
determinePresenceMapUsage	group 中的 field 是否使用存在图
getField	获得 group 中的 field 信息
hasField	group 中是否包含了指定的字段

4.8 序列（sequence）

序列由一个长度和有序的元素组成，每个元素为分组类型。

在 openfast 中，org.openfast.Sequence 类表示了序列，其主要的方法有：

方法名称	说明
getLength	获得序列中 group 分组的数目
Encode	对于 group 类型进行编码

Decode	对于读入的 bit 流进行解码
getFieldCount	获得 sequence 中的 Field 数目

之前两个字符串和字节序列转换的类描述。TODO

4.9 重点回顾

- 基本类型包括了整数、字符串、字节向量、十进制数等类型；
- 在 OpenFast 中分为类型和类型值的处理过程。

4.10 参考资料

【1】http://www.fixtradingcommunity.org/pg/discussions/topicpost/169550/1063-zeropreambles-for-ascii-string