一种组装式 ASTERIX 数据通用解码算法

龙滨，李建宏

民航重庆空管分局，重庆 400000

摘　　要： ASTERIX 是 Euro Control 组织开发的一种数据传输格式，在民航空管、军事、航海等领域有广泛应用。目前 ASTERIX 数据的解码有多种算法，本文提出一种独特的组装式 ASTERIX 通用解码算法，通过类似模块化组装的方式，可实现所有 CAT 类型 ASTERIX 数据的解码，并以 C++ 语言为例对该算法进行了阐述。该算法不同于其他算法的特点是，实现了对 ASTERIX 最底层数据的解码和访问，无需用户二次解码。

关 键 词： ASTERIX；通用；解码；组装式；模块化；插槽；插件

A general Decoding Algorithm for Assembled ASTERIX Data

Long Bin, Li Jianhong

CAAC Chongqing Air Traffic Control Branch, Chongqing 400000

Abstract: ASTERIX is a data transmission format developed by Euro Control organization, which is widely used in the fields of civil aviation air traffic control, military, navigation and so on. At present, there are many algorithms for decoding ASTERIX data. This paper proposes a unique assembly ASTERIX general decoding algorithm, which can realize the decoding of all CAT type ASTERIX data through modular assembly, and takes C + + language as an example. This algorithm is different from other algorithms in that it can decode and access the lowest level data of ASTERIX without the need of user secondary decoding.

Keywords: ASTERIX; general; decoding; assembly; modular; slot; plug-in

ASTERIX（通用结构化欧控监视信息交换）是 Euro Control 组织开发的一种数据传输方法，广泛用于雷达等监视数据的传输。在民航空管中，广泛用于一 / 二次雷达、场面监视雷达、ADS-B、多点定位、综合航迹等监视信息的输出。

本人查阅了大量 ASTERIX 各类文档，研究出一种 ASTERIX 数据解码算法。运用这种方法，可以实现对 ASTERIX 所有 CAT 类型数据的通用解码。该方法相对于传统算法有以下几个优点：

1. 组装方式简单。基于组装方法，实现无限层级组装，容易理解和实现。

2. 前后向兼容。算法充分满足 ASTERIX 规范要求规定的对 CAT 类型的前后向兼容。

3. 深度解码。可以实现对 ASTERIX 定义的最底层数据，无需用户二次解码。

4. 对字段和元素高低位进行翻转。ASTERIX 数据在传输过程和处理过程中存在字节高低位问题，该算法在处理过程中对字段高低位进行了合理化反转，方便用户进行后续处理。

5. 该算法保留多层级的 FSPEC 元素，方便用户判断该字段是否存在。

6. 低代码引用。以 C++ 为例，最终用户只需要三行代码，即可完成注入数据的解码。

一、ASTERIX 基本数据结构分析

在欧控规范监视数据交换第一章中，规定了一个 ASTERIX 数据由 CAT、LEN 和数据记录组成。数据记录又由 FSPEC 和各数据字段组成。FSPEC 表示以位为单位标识每一个字段是否存在。通过查阅大量 ASTERIX 文档，笔者发现，在实际的 CAT 类型中，由于欧控官方在各文档的字段类型方面不够严谨，比如未对 1+ 和 1+1+ 类型的区别进行说明、未对 a+ 类型类型进行分类等，笔者实际参考多个 CAT 文档总结出来的数据字段类型，不同于官方 [1] 定义，分类如下：

1. 固定长度类型。比如：1个字节、3个字节、8个字节等。

2. 可变长度类型，可变长度类型又分为4类：

1）FSPEC 类型，定义。

2）a+ 类型，a+ 类型又分为两种：

a. 扩展 a+ 类型：如 CAT048类型中的 Target Report Descriptor。

b. 重复 a+ 类型：如 CAT048类型中的 Warning/Error Conditions 是1+ 类型、CAT062中的 Composed Track Number

作者：龙滨（1976.03-），男，汉族，重庆市巫溪县人，大学本科（研究生学位），高级工程师，研究方向：空中交通管理通信、导航、监视。

用到了3+ 类型。

3）a+b*n 类型：如1+8n、1+2n 等。其中：

a 通常为1，表示1个字节，这个字节放置重复次数 n，目前 ASTERIX 只有 a=1 个字节的情况，不排除还存在 a>1（也就是后续数据总长度大于256字节）的情况。

b*n 表示以 b 个字节为重复，重复次数为 n。

因此，这个字段的总长度为 a+b*n。

4）保留扩展字段类型：

保留扩展字段的第一个字节表示长度，表示后续（不包含第一个字节）数据的字节总长度。后续的数据又是数据记录（DATA RECORD）类型，以上述3种类型组成。

根据 ASTERIX 数据格式结构，要进行解码，首先要通过 FSPEC 进行分析哪些数据项是存在的，对存在的数据项才处理。其次，数据长度是不确定的，要根据注入的数据进行分析后，才能确定数据长度。解决上述问题后，才能进行具体数据项的解码。

二、ASTERIX 数据结构的封装原理

为实现 ASTERIX 数据的数据封装解码，笔者提出一种插槽式的数据装配方法，该方法的核心思想是根据对于每一种 CAT 类型，将整体数据看做一个机框，每个机框中包含多个插槽，每个插槽中又可以包含不同类型的子插槽，子插槽中又可以包含子插槽和插件，插件不是容器，是最底层的模块。这样，我们就可以实现对整个数据的封装。

为方便数据处理，我们将数据类型定义为5种类型：

1. Cat 类型，固定一个字节，插件。

2. Len 类型，固定长度，2个字节，插件。

3. Fspec 类型，该类型的定义不同于欧控文档定义。Fspec 是一个基本的插槽。该类型包含数据的定义规范插槽和各子字段插槽。所有的数据项（Data item）均挂载在 FSPEC 类型上。

Fspec 分为两种类型，一是可变长度 Fspec，另一个是固定长度 Fspec。通常的 Fspec 都是可变长度的，根据每个字节的最后一位 Fx 确定是否扩展。但对于某些扩展保留字段来说，如 cat048扩展保留字段[3] 的 Fspec，文档定义是固定长度的，其最后一位不是 Fx。

4. Basic Item 类型，即基本数据项，我们可以认为是插件，是插槽中的最小组件。

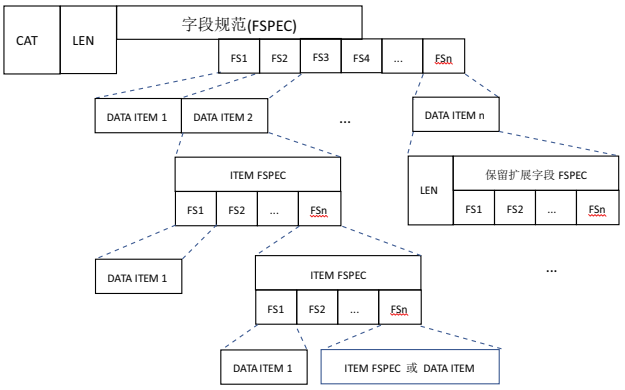
该类型包括三种基本数据项：固定长度类型、变长 a+ 类型、变长 a+bn 类型。

5. Link 类型

LINK 类型也是一种插槽类型，顾名思义为链接。链接不直接存储数据，而是一个容器。该类型主要用途是处理各种定义不规则的情况。相比于 Fspec 类型，可以更灵活地进行数据装配。以扩展保留字段为例，其通常包含1个字节的 len 字段、一个 Fspec，由于这个 len 字段长度不同于上面的 Len 类型，因此不能使用 FSPEC 类型。这时我们可以 Link 类型作为容器，将 len 类型作为插件，将扩展保留字段的 FSPEC 类型作为子插槽。当然该类型还可以用到其他定义不规范的情况。

另外还需要解决一个问题，我们需要知道数据项是否存在。笔者将数据分为3类：存在、不存在和未知。我们知道 ASTERIX 类型中有三个数据类型是必然存在的：Cat 类型、Len 类型和最顶层 Fspec 类型。其他的数据项在数据包处理前，是否存在是未知的，需要收到具体的数据后再确定。这三种类型涉及到算法的不同，如果数据不确定是否存在，需要从数据流的 FSPEC 的数值进行判断。

有了上述概念，我们可以考虑如下的数据结构组装方式。基本数据结构为树形结构，以下是一个示例：



> 图1 ASTERIX 组装式结构示意图

上图中，我们把 CAT、LEN 和 DATA ITEM 作为插件，将 FSPEC 作为插槽，把整个结构作为容器。需要注意的是扩展保留字段，由于其开头是一个 LEN 类型，不同于 FSPEC 类型，我们采用非常灵活的 Link 类型，以包容这种复杂类型，并扩展子插槽：保留扩展字段的 LEN 及其 FSPEC。

以 C++ 为例，我们可以设计一个函数 initDataItemList()，进行 CAT 类型装配[1]：

```
vector<DataItem> initDataItemList() {}
{
vector<DataItem> packageDataItemList;
vector<DataItem> fspecDataItemList;
setCatToDataItemList(&packageDataItemList, “PACKAGE”, “Category”, 48, EXISTED_TRUE); setLenToDataItemList(&packageDataItemList, “PACKAGE”, “Length”, 2, EXISTED_TRUE);
setBasicDataToDataItemList(&fspecDataItemList, “FSPEC”, “Data Source Identifier”, IS_DATA_ITEM_FIX_LENGTH,2, 2, false, 0, EXISTED_UNKNOWN);
...
setVariableLengthFspecToDataItemList(&packageDataItemList, “PACKAGE”, “FSPEC”, IS_VARIABLE_LENGTH_FSPEC, EXISTED_TRUE, fspecDataItemList);
...
}
```

值得一提的是，欧控文档中对每个数据项定义了 FRN 和 data item 名称（如：1048/010[2]）。因为其只定义了最外层 FSPEC 插槽中的 FRN 和 data item，对于1+1+ 类型中的底层基本数据项，

则未进行定义名称。因此，为保证唯一性，笔者采用对父节点和子节点命名的方法，比如："Category" 表示 cat 类型，”FSPEC”表示顶层的 Fspec，“Data Source Identifier”表示数据源标识符项，“Flight Plan Related Data FSPEC”表示 Fspec 下层的 FSPEC。这样，我们就可以通过当前节点的父节点和子节点名称（该 CAT 类型中，要求所有插槽和插件名称不相同），就能确定到当前唯一节点，以取代了 Asterix 文档中的 data item 名称的方式。

三、数据项的基本结构

数据结构封装完毕后，整个数据都是空的，需要注入相关的 CAT 数据才能获取到数据项。

为保存每个数据项，我们定义了一个 DATA ITEM 容器。该类型容器结构和数据结构的复合体，既可以定义插槽，也可以定义插件。所有的插槽、子插槽和插件均用该容器进行定义。其结构如下：

表1 ASTERIX 基本数据项结构

DATA ITEM 数据项	含义及作用
数据列表	容器，用于存储该数据项的字节数据
数据类型	包含5种基本类型：1.Cat 类型，2. 长度类型 Len，3.FSPEC 类型 4. 基本数据项类型，5.LINK 类型
项目名称	本节点数据项名称
项目父名称	父节点数据项名称
总长度	本数据项的总长度
数据项存在状态	包括三种状态：存在、不存在、未知
CAT 数据	数据类型为 CAT 类型时，存储 CAT 类型
LEN 数据	数据类型为 LEN 类型时，存储该数据包或容器总字节长度
数据长度类型	包含三种基本类型：固定长度、a+ 类型、a+b*n 类型（ASTERIX 文档中只存在1+b*n 类型，此处用 a 代替1是考虑未来可能的不规则写法）
数据头部长度	对于固定长度，等于固定长度 对于 a+ 类型，等于 a 对于 a+b*n 类型，等于 a
是否长度扩展	对于固定长度类型，等于 false 对于 a+ 类型，等于 true 对于 a+b*n 类型，等于 true
乘数	对于 a+b*n 类型，等于 b
重复次数	对于 a+b*n 类型，等于重复次数 n（该值从具体数据获取）
Fspec 类型	1. 普通 FSPEC 2. LINK 类型，用于保留扩展 FSPEC，保留扩展的 fspec 的第一个数据为 len 长度 =1，同普通的 fspec 不同
子节点列表	容器，用于存储下一层的所有子节点

上述的数据成员中的“子节点列表”，也就是插槽，是用于存储下一层的所有节点。我们只需要将下一层的每个节点插入到这个列表，即可实现整个结构数据的封装，在 c++ 中可以通过 vector<DataItem> 来实现这个树形结构的封装。

从上面的数据项定义可见，最上层是数据包 package，其中子节点列表中封装了三个数据：CAT、LEN 和 FSPEC。在 FSPEC 中，我们又可以将下一层的数据插入到这个列表中。通过这种方式，我们可以实现任何复杂的数据类型的封装。

四、对 CAT 数据基本格式进行结构定义

为实现访问到 ASTERIX 的最底层数据，也就是元素，我们需要按照 ASTERIX 的具体文档进行对该 CAT 类数据进行定义，以 C++ 为例，我们可以在 CAT048 头文件中定义一个 CAT048 类型 [2]：

```
class Cat048Data {
Public:
Struct Fspec {
BYTE fx1:1;
BYTE radarPlotCharacteristics:1;
...
BYTE dataSourceIdentifier:1;
BYTE fx2:1;
...
BYTE calculatedAcceleration:1;
...
} fspec;
struct DataSourceIdentifier
{
BYTE SIC;
BYTE SAC;
} dataSourceIdentifier;
...
struct ModeSMbData
{
BYTE REP; // 重复次数
ModeSMbDataItems *modeSMbDataItems; // 用指针可存储多个项
} modeSMbData;
}
```

这样，对于我们的一个对象实例，如 cat048Instance，在数据解码后，则可以直接通过如 cat048instance.dataSourceIdentifier.SIC 方式直接对最底层元素进行访问。对于重复性的如 a+bn 的结构，我们可以通过 cat048Instance.modeSMbData.ModeSMbDataItems[i].MB 来访问某个重复数据的元素。

这样就实现了对最底层数据的访问。对于其他编程语言，可以采用相应的类代替上述的 struct。

五、数据的解码

我们需要实现将实际数据进行注入和解码。我们可以对第二

节封装的数据包的树形结构进行遍历处理，在遍历的过程中，根据数据结构的定义，将 cat 实例的各数据进行填充。

我们在 asterix.h 文件中定义了 DataItem 结构（见第三节），还需要定义一个 Asterix 类，用于对数据结构进行初始化、具体数据项处理和数据解码 [3]。

```
class Asterix
{
public:
Cat048Data cat048data;
...
virtual vector<DataItem> initDataItemList() ;
virtual void processDataItemList(vector<DataItem>::iterator iter);
void dataParser(vector<DataItem> *packageDataItemList, BYTE *datas);
...
};
```

这里设计了虚函数 initDataItemList()，用于初始化某 CAT 类型的数据结构封装（见第三节），并生成一个 packageDataItemList 树形结构。

这里的 dataParser()，通过封装的 packageDataItemList 的树形数据结构定义，将实际数据进行注入，通过对该结构进行遍历，解码出各个数据，并将各数据填入到 cat 类对象的实例的数据列表中。由于 ASTERIX 的最底层元素的解码是没有一个统一规律，因此，对最底层数据的解码要逐项进行进行，因此我们设计了 processDataItemList() 虚函数，用于处理各具体的数据项，需要在继承该 Asterix 类的具体 CAT 类中实现。

上述的 initDataItemList() 和 processDataItemList() 由于必须结合具体 CAT 结构定义和数据具体来实现，因此设置为虚函数，是必须在 CAT 类中予以实现。

根据第二节 ASTERIX 数据的数据结构组装，可以看到，整个数据必须安装 CAT 文档定义的顺序进行组装。因此，在 dataParser() 函数中，我们分两个阶段对数据进行处理，第一步根据 packageDataItemList 中定义的数据结构对树形节点进行遍历，根据 FSPEC 定义的数据否存在，和数据类型和数据长度，确定该各数据项的具体内容，将具体的字节数据置入数据列表。由于组装过程是安装定义顺序的，因此遍历过程也是顺序执行的。初次遍历完成后，所有的数据都存入了 packageDataItemList 中。此时进行再次遍历，将 packageDataItemList 中各数据项的值写入到 cat 类型实例中。最终对解码出来数据的访问是通过 cat 类型实例来获取的。

```
void Asterix::dataParser(vector<DataItem> *packageDataItemList, BYTE datas[])
{
int position = 0;
initTraversal(packageDataItemList, datas, &position);
processTraversal(packageDataItemList);
}
```

在 initTraversal() 中，我们注入的数据，并对树形结构进行遍历，进行首次解析处理。

```
void Asterix::initTraversal(vector<DataItem> *dataItemList, BYTE *datas, int *position)
{
for (vector<DataItem>::iterator iter = dataItemList->begin(); iter != dataItemList->end(); ++iter) {
if (iter->dataType == IS_CAT && iter->existed == EXISTED_TRUE) { //category 类型
...
} else if (iter->dataType == IS_LEN && iter->existed == EXISTED_TRUE) { //len 类型
...
} else if(iter->dataType == IS_FSPEC  && iter->existed == EXISTED_TRUE) {
...
if (iter->fspecType == IS_VARIABLE_LENGTH_FSPEC) { // 是普通的变长 fspec
...
initTraversal(&iter->subdataItemList, datas, position); // 继续遍历
} else if (iter->fspecType == IS_FIXED_LENGTH_FSPEC) { // 保留扩展字段 fspec
...
initTraversal(&iter->subdataItemList, datas, position); // 继续遍历
}
}else if(iter->dataType == IS_BASIC_DATA && iter->existed == EXISTED_TRUE) {
if (iter->dataLengthType == IS_DATA_ITEM_FIX_LENGTH) { // 固定长度数据
...
} else if (iter->dataLengthType == IS_DATA_ITEM_A_PLUS) { // 变长，a+ 的情况
...
} else if (iter->dataLengthType == IS_DATA_ITEM_A_PLUS_B_N) {//a+bn
...
}
} else if (iter->dataType == IS_LINK && iter->existed == EXISTED_TRUE) { // 链接，不做处理，继续遍历
...
initTraversal(&iter->subdataItemList, datas, position); // 继续遍历
}
}
```

```
}
```

主要思路是每个层级对每个数据项进行遍历，如果数据项类型是 FSPEC 或 LINK，则继续往下遍历，否则就终止遍历，回到上层继续处理。通过这种遍历，我们可以根据当前数据的 position，确定当前数据的相关数值，并写入到当前数据项 iter 中。

以上将注入的数据全部都放到了 packageDataItemList，最后一步就是要再次进行遍历，packageDataItemList 中的数据转到 CAT 实例中。此处需要在 catxxx.cpp 中实现 processDataItemList() 函数。

以 cat048为例，进行如下处理：

```
void Cat048::processDataItemList(vector<DataItem>::iterator iter)
{
// 顶层的 CAT 类型和 LEN 在初始化时已处理，只需从 fspec 开始处理
if (iter->fatherName == “PACKAGE” && iter->itemName == “FSPEC”) {
setFspec(iter);
}
if (iter->fatherName == “FSPEC” && iter->itemName == “Data Source Identifier”)
{
setDataSourceIdentifier(iter);
}
...
}
```

上述处理过程中，我们可看到，我们处理一个某一个指定的数据包，是通过父节点名称和自身名称来判断的。

以 setDataSourceIdentifier(iter) 为例，以下简单介绍处理过程（处理方式可自行设计）：

```
void Cat048::setDataSourceIdentifier(vector<DataItem>::iterator iter)
{
BYTE* arr = new BYTE[iter->totalLength];
copy(iter->dataByteList.begin(), iter->dataByteList.end(), arr);
memcpy(destination, arr, iter->totalLength);
delete []arr; // 释放 arr 内存
}
```

通过上述内存拷贝方式，将数据传入给了 cat048Data 实例，后续就可以通过 cat048Data.dataSourceIdentifier.SIC 进行数据访问。

六、极简的数据解码过程

最终，用户只需要以下三行代码，就可以实现对某 CAT 类型的 ASTERIX 数据包进行解码：

```
Cat048 asterixCat048;
vector<DataItem> package = asterixCat048.initDataItemList();
asterixCat048.dataParser(&package, datas);
```

使用时，只需要 asterixCat048.cat048Data.dataSourceIdentifier.SIC 即可获取解码后的原始数值。当然，解码后，如果用户需要将经纬度等原始数据进行一些格式转换，可以自行进行处理。

七、总结

该解码算法是一种通用 ASTERIX 解码方法，不针对特定 CAT 类型。笔者通过上述算法，完成了 cat048 的 C++ 解码样例，并同组织完成对最新版本 cat010 和复杂的 cat062 进行了解码。通过编写 asterix 类，让 Cat 类继承 Asterix 类，并在 Cat 类的方法中，实现该类型的数据封装和具体数据项处理。极大降低了解码的难度。同时，对于用户来说只需要3行代码即可完成数据的深度解码。由于采用了父节点和子节点名称的命名方式确定唯一节点，完全做到了前向后兼容。

采用该方法，将极大地简化整个解码过程，并方便用户引用。

参考文献

[1] EUROCONTROL Specification for Surveillance Data Exchange – Part 1 – All Purpose Structured EUROCONTROL Surveillance Information Exchange (V3.1).

[2] EUROCONTROL Specification for Surveillance Data Exchange – ASTERIX Part 4 – Category 048: Monoradar Target Reports(v1.32).

[3] EUROCONTROL Specification for Surveillance Data Exchange – ASTERIX Part 4 Appendix A Category 048: Monoradar Target Reports – REF(V1.12).