SDDS

快速构建二进制通讯解析程序的利器

© ByteFerry <byteferry@qq.com>

什么是SDDS？

- 数据结构
  - SDDS是stream data dynamic structure的缩写，即流数据动态结构。
- DSL
  - SDDS是一种DSL (domain-specific language)。 SDDS使用了程序员可读且程序可理解的方式用来描述二进制流数据的结构。
- 解析引擎
  - 基于SDDS规范实现的二进制流数据解析引擎，可以帮助你快速完成通讯程序中数据收发的解析的开发。

为什么要用SDDS

- 简单易用
  - SDDS使用JSON格式定义通讯协议的schema。几乎是零学习成本。
- 快速高效
  - 使用了SDDS，则不再需要编写通讯协议的编码与解码程序，只需要用JSON来定义通讯协议的schema即可。
  - 本来需要一两个月，现在只要一两周
- 可扩展性
  - 使用了SDDS，通讯协议有所变化时，绝大多数情部分下只要修改通讯协议的schema即可。
- 开源免费
  - SDDS是开源的项目，使用MIT授权，你可以自由使用它。

SDDS引擎的优势——重造轮子的理由

- 内核加扩展模式
  - 内核支持绝大多数常用数据类型。
  - 可通过扩展增加所需要特殊类型。
- 单一函数接口
  - 解码只要调用decode方法，传入byte_buffer,返回key-value结构的列表
  - 编码只要调用encode方法，传入key-value结构的列表，返回byte_buffer。
- 不需要编译
  - 不同于其它的二进制读写DSL，SDDS不需要编译生成对应的类。需要做的只是，编写Schema，同时给SDDS引擎添加有限的几个扩展函数，即可运行。

简单的示例

```
"body_length"：{
   “id"："body_length",
   “name"："body_length",
   "position"：4,
   "type"："int32",
   "unsigned"：true
}
```

- 这一示例是要求SDDS引擎从数据流第4字节读出一个无符号的32位整数

SDDS支持的数据类型

- 整数类型：

名称	字节数	位数	别名
int8	1	8	char, byte
int16	2	16	short, word
int24	3	24	
int32	4	32	int, dword
int64	8	64	long

  - 符号：
    - 整数类型同时支持有符号与无符号类型，
  - 字节序
    - 同时也支持不同的字节序（大端(big-endian)、小端(little-endian)）
  - 别名
    - 所有的别名均可以用在SDDS Schema中， SDDS 引擎可以读出正确的类型。

SDDS支持的数据类型

- 浮点数类型：

名称	字节数	位数	别名
float32	4	8	single
float64	8	16	double

  - 别名
    - 所有的别名均可以用在SDDS Schema中， SDDS 引擎可以读出正确的类型。
  - 64位支持
    - Float64，int64需要系统运行环境支持

SDDS支持的数据类型

- 其它基本类型：

名称	说明
bytes	读取、写入指定的字节数
bit	读取、写入一个二进制位(bit)
bits	读取、写入指定的二进制位(bit)数
string	读取、写入指定字节数的字符串
bcd8421	读取、写入基于8421算法的BCD码
hex	读取、写入16进制字符串

  - 操作方式
    - sdds引擎不仅要支持读(read)，同时也要支持写(write),插入(insert),替换(replace）。

SDDS快速入门向导

- 基本说明：
  - 本向导以一个设备监控向导为例，带您创建一套SDDS的schema。
- 需求假定：
  - 我们需要监控每一个终端控制箱中的温度，湿度，PH值，转速，并可以分别让终端控制箱启动与停止。

第1步：规划消息ID

- 消息ID
  - 消息来源
    - 我们规定：消息ID中第一个字节，表明消息来源
    - 0x10表示服务器， 0x20表示终端
  - 消息类型
    - 0x01、控制命令，启动，停止
    - 0x02、即时状态查询，
    - 0x03、参数设置
    - 0x04、历史数据
  - 请求与回复
    - 0x01 请求
    - 0x02 回复
  - 完整的消息ID
    - 消息来源+消息类型+请求与回复,共3个字节。

第2步，复制SDDS Schema 总体结构

```
{
  "meta"：{
    "comment":"注释，程序注释",
    "version":"版本",
    },
  "options"：{
    "endianness":2,
    "min_length":15,
    "debug":true,
    "top_node":"message"
    },
  "sdds"：{
    "decode":{
       "message"{
       }
     },
    "encode":{
       "message"{
       }
     }
   }
}
```

- 在这个范本中，meta的注释与版本，不用填了，因为是演示。同时，程序也不会处理它。
- 程序运行的选项Options中， endianness字节顺序也用2（大端模式），具体消息的最少长度，放到后面再修改。
- min_length属性是用来改善性能的。即预分配多大的内存，减少动态内存分配次数。

第3步，编辑消息节点

```
"message":{
        "name": "message",
        "position":0,
        "byte_fields": {
                "stx":{},
                "head":{},
                "body":{},
                "tail":{}
                "etx":{},
        }
}
```

- 因为，SDDS引擎是从主节点内遍历读或写所有子节点来完成一个消息的读写，所以，message中我们用byte_fields分支列出了消息的第一层结构
- 即标识位，消息头，消息体，消息尾（一般是校验码），标识位。

第4步，编辑标识位节点

```
"stx":{
   "comment": "数据包标识位"
   "type":"int8",
   “unsigned”:true,
   "length":1,
   "position":0,
   "value":125
}
“etx":{
   "comment": "数据包标识位"
   "type":"int8",
   “unsigned”:true,
   "length":1,
   "position":0,
   "value":125
}
```

- 此节点用到6个属性。
- comment：写注解是一个好习惯！
- type：数据类型
- unsigned：是否是无符号的。
- length：数据长度
- position：数据位置
- value:数据值

- 其中，标识位，value肯下是固定值，所以，如果是编码（encode）中的节点，可以直接写上。解码就不用了。
- 我们可以顺便把结束标识位也编写好，因为二者是一样的。

第5步：编写head节点

```
"head":{
   "id":"message_header",
   "type":"message_header",
   "position":3,
}
```

- 因为消息头本身也是有结构的。但不能在message节点中展开。所以，我们在数据类型中使用了message_header这个类型。接下来，我们在与message同一层次增加一个节点，名为：message_header。
- 本节点中，使用了id属性，这一属于可以帮助我们使用节点的path找到此节点。

第6步：编写message_header节点

```
"message_header":{
   "name":"header",
   "id":"header",
   "length":11,
   "position":3,
   "byte_fields":{
      "message_id":{},
      "flow_id":{},
      "body_property":{},
      "time":{}
   }
}
```

- 这里，我们消息头中定义了4个字段，其中， time类型是bcd8421, message_id是int24, flow_id是int16,
- 这三个字段，均可以按stx,etx的格式展开。
- 但是， body_property，为了节省空间，我们使用位存储。由于字段内不允许再嵌套，所以，body_property需要增加一个自定义类型的节点。

第7步，完成body_property_detail节点

```
"body_property_detail":{
   "id":"body_property_detail",
   "type":"bytes",
   "length":2,
   "position":3,
   "bit_fields":{
      "body_length":{},
      “message_type":{},
      " crypt ":{},
   }
}
```

- 我们在前面的字段中定义了：

```
"body_property"：{
   "id"："body_property",
   "type"："body_property_detail"
}
```

所以，我们要完成：body_property_detail

而此节点中则包括bit_fields分支，即位存储的分支。所以，此节点中type一定是bytes，且length必不可少。

第8步 完成body_length节点

```
"body_length":{
   "id":"body_length",
   "type":"int",
   "length":10,
   "position":0,
   "selector": "id"
}
```

- body_length是一个位操作节点，所以，我们要指定位操作的类型：bit、 bits或int。这里肯定是int，所以，还要指定length.
- 如果是对要发消息进行编码，我们是完成后才能写入此节点，所以，要给节点配一个选择器，好让程序后续可以找到它。
- 这里，我们用的选择器是id，所以，id属性同时出要指定。

第9步，完成body节点(1)

- 由于body节点是message中具体的数据部分，所以，会因不同的消息分类而对应不同的数据。
- 所以，第一步，我们将body部分作为自定义类型，名为message_body,指向到Schema中的message_body。
- 其次，因为不同的消息分类而对应不同的数据，我们要解决的两个问题：
  - message_body的长度，我们在body_length节点有定义，这里可以读到它。
  - 其次是不同的消息类型，对应不同的message_body的格式，我们要获得message_type。前面我们在body_property中定义了message_type，我们可以读到，如果未定义，我们则需要通过自定义函数在message_id中算出message_type。

第9步，完成body节点(2)

```
"mssage_body": {
  "id": "mssage_body",
  "one_of": {
     "key": {
        "comment": "",
        "name": "message_key",
        "value": "#message_type",
        "format": "0x%04X",
        "id": "message_key"
     },
     "list": {
        "0x200201":{},
        "0x200402":{}
  }
}
```

- 根据前面分析，我们message_body节点应当是这样的：
- 我们使用one_of分支，让sdds引擎自动选择message_body所对应的具体的节点。
- 此分支中有两大分支，一是key，指定选择所用的关键字。value中通过选择器指定数值的来源。通过format指定key的格式。
- list中是备选列表，每一项就是前面key值获取数值后格式化的结果。
- list中如果是自定义类型，同样也要指向到新的节点。而不是在本地展开。
- one_of list中的属性名不可以用数字，所以，如果是数字，则要用16进制字串，所以，format必不可少

参数传递：选择器(1)

- SDDS中有4种选择器：
  - "#"：id选择器
  - "@"：路径(path)选择器
  - "$"：索引(index)选择器
  - "*"：函数选择器
- 以上选择器一般是写在value属性中，比如：
  - "value": "#message_type"
- 则是表示数值是读取id为message_type的value

参数传递：ID选择器

- "#" id选择器
  - 指定读取对应id的值。
  - 要使用id选择器，首先要在被操作节点中增加选择器属性： "selector":"id"
  - 其次，在要操作的节点中，通过“value”:“#body_length”, 指定读取哪个id节点的值。这里读取的是body_length，这也就是说指定“selector”: “id”的节点中，肯定有属性：
  - "id":"body_length"。
  - 注意，id在同一SDDS文件中应当唯一。如果不唯一，则应当使用其它的选择器。

参数传递： 路径(path)选择器

- "@": path选择器
  - 路径(path)是用“.”分隔的id字符串。比如：
  - "value":"&body.message_body.body_length",
  - 路径是先向上搜索，搜索到相同节点后，再向下搜索。所以，路径只要指定到分叉节点即可，不一定要指定到根节点。
  - 路径(path)选择器中的path不宜太长。如果过长，则要考虑更换为id选择器。
  - 如果不知path，则可以通过“trace”:“path”来调试查看对应节点的path
  - 因为repeat节点的分支id会重复，所以，不能用path选择器。

参数传递：索引(index)选择器

- "$" 索引(index)选择器
- 索引选择器，专门针对repeat(重复多项的)的分支节点。所以，通过将其下标(索引)作为搜索参数，以方便找到此节点。
- 要使用索引(index)选择器，首先要在希望被操作的节点中增加selector属性。指定：,"selector":"index"
- 其次，在要进行操作的节点中，通过"value":"$item_length"这样的方式去读取被操作节点的值。这里，读取的是item_length节点。

参数传递："*" 函数选择器

- "*" 函数选择器
  - 调用在扩展（继承类）中实现的自定义函数。
  - 任一属性值中，如果第一个字符是以上的选择器字符，SDDS将会调用选择器程序读取数据。
  - 所以，如果开头是以上选择器字符，则要使用转义。比如："\#","\$"。一旦使用转义，则需要在扩展程序中做相关处理。 SDDS不会自动清除转义。
  - 函数选择器，仅用来选择自定义函数。所以，如果使用它，你必须在扩展类中增加自己的自定义函数。

第10步 完成历史数据节点

```
" 0x200402 ": {
   "id": "history",
   "name": "history",
   "position": 0,
   "length": "#body_length",
   "repeat": true,
   "count": null,
   "until": "",
   "type": "history_data"
}
```

- 终端给服务器的历史数据消息，根据前面规划，是0x200402
- 我们定主义为前面的one_of的list 中的子节点。
- 此节点定义为重复节点，即，相同的数据项有多项。
- 重复节点的定义，首先是："repeat": true,
- 其次，定义重复的条件，length，一般在解码时，肯定有，比如通过特定的length节点，或直接用body_length。在写入时，肯定了解，要写多少个，所以，可以指定count，如果二者都没有，复杂条件可以在until中指定自定义函数的名称。在扩展中实现自定义函数。
- 最后，则是要重复的节点，通过type中给定自定义类型的名称，指定对应的节点。

后续工作

- 完成所有节点
  - 到目前为止，我们已经学习了SDDS中所有节点的类型。我们可以完成其它的所有节点。在此基础上完成的schema，理论上SDDS引擎是可以正确解析了。但是，我们仍需要测试。
- 测试Schema
  - 我们可以通过SDDS引擎提供的测试工具进行测试，测试时，我们将options中的debug置为true，并且，在任意有问题的节点中将debug置为true，我们即可以获得相关输出信息。从而纠错。
  - 另外，也可在任意节点将trace值设为要跟踪的属性，我们即可以获得相关输出信息。从而纠错。
- 编写验包、拆包程序
  - 此外， SDDS引擎并不处理验包，拆包，所以，我们要自己编写相关的验包，拆包程序。

更多特性

- 目前，SDDS节点属性共有26个。前面我们涉及到了16个属性，以下是未涉及到的10个属性：
- after_change
  - 定义读写操作后调用的自定义函数，多个函数名时，要用逗号分隔
- before_change
  - 定义读写操作前调用的自定义函数，多个函数名时，要用逗号分隔
- comment
  - 节点或schema的文字注解。写注解是一种好习惯。
- default
  - 定义要写入的默认值，如果你不希望程序传值。程序无值时，会读取此值并写入。
- format
  - 用来定义获取数值后的数据格式化处理。格式化是key节点必不可少的，但是，对于其它节点，也可以使用。

更多特性

- formula
  - 用来定义解码读取后或编码写入前所用的公式。SDDS中提供了公式计算引擎。可以将你的公式算出结果，公式中，要代入的读出的value的变量名永远是"A"
- ignore_errors
  - 定义此节点是否忽略错误。如果遇到错误，又不想改动程序，如果忽略错误可以，则可以在当前这个节点"ignore_errors":true
- name
  - 定义节点的变量名。只有有name的才会被写入到读出的结果中。因此，假如你不要某个节点的值，或许此节点无值，你可以去掉name属性。
- position
  - 定义数值读写时在Stream中的Offset. 在insert,replace操作中，position是必不可少的。其它情况， position也可以帮助我在编写schema时验证其正确性。
- required
  - 指明要写入的值是否为必须。正常为false. 由此，如果未给定值，程序会写入NULL。

7种语义结构

- SDDS仅通过7种语义结构完成了流数据的动态结构解析，这7种语义分别是：
  1. 操作一个是否有符号的一定长度的基本类型
  2. 使用一个自定义类型，指向到自定义类型节点
  3. 操作一个无符号的一定长度的字节数用于位解析
  4. 操作一个一定长度的位转换为整数
  5. 定义一个节点，由多个字节型的节点组成
  6. 按（长度，数量，函数）中的一个条件重复一个自定义类型
  7. 根据指定的关键字key在list中选择一个节点。

常见问题

- 1、SDDS引擎是否可以做数据包有效性验证？
  - SDDS引擎不做数据包的有效性验证。这一部分必须在SDDS引擎之外完成。这样可以适合不同的通讯协议。但若需要SDDS引擎在验证前读出相关数据结构，则可以在SDDS引擎中定义validate节点进行读取。
- 2、 SDDS引擎未读出数据，或未写入数据。但无任何错误或异常。
  - 这是因为，缺少name属性。添加name属性，即可解决。
- 3、如何实现repeat多个one_of节点？
  - 只要schema中，定义父节点是repeat，子节点使用one_of即可。

常见问题

- 4、attribute和property区别是什么？
  - attribute和property都可称为属性。但是，attribute是所有可以在schema中定义的字段。而property则是SDDS引擎中使用的属性。我们一样要了解property，因为程序需要对它进行操作。 SDDS约定：所有property均以"_"开头。
- 5、 SDDS中没有if条件，如何处理？
  - SDDS中不需要if条件，这样的需求可以通过one_of节点结构来实现。
- 6、选择器是不是只能搜索本节点前的已加载的节点？未加载的如何处理？
  - 是的，选择器只能搜索已存在的节点。要处理未加载的节点，不要在前面的节点中处理后面的节点，而是在后面的节点中读写前面的节点。

扩展应用

- SDDS不仅可以用于通讯数据流的编码与解码，同样可以用于：
- 二进制文件的读写，包括：
  - 图像文件
  - 音视频文件
  - 压缩包文件
  - PDF文档

  - 更多应用，需要你的参与！

相关资源

- SDDS规范
  - https://github.com/byteferry/sdds/blob/master/SDDS%E8%A7%84%E8%8C%831.0%20Beta.md
- SDDS For PHP
  - https://github.com/byteferry/sdds_php
- SDDS Cheat Sheet
  - https://github.com/byteferry/sdds/blob/master/SDDS%20Cheat%20Sheet.pdf
- SDDS客服支持QQ群：
  - 769010107

请给打一个星

- github Star的作用：
  - 可以在Your Stars中方便找到这个项目，下次使用时不再需要搜索。
  - 可以告诉别人，这是一个好项目，让更多的人使用它。

- github fork的作用：
  - 如果你有兴趣进一步改进这个项目，你也可以fork这个项目。
  - 一旦你fork了这个项目，在你的名下，也有了这个项目，同时也会显示fork的来源。
  - 并且，你也就可以共同参与这个项目了。

Thank you!

<byteferry@qq.com>