浅析Hessian协议

> Hessian二进制的网络协议使不需要引入大型框架下就可以使用，并且不需要学习其它的入门的协议。因为它是二进制协议，它更擅长于发送二进制数据，而不需要引入其它附件去扩展它的协议。

Hessian支持很多种语言，例如Java，Flash/Flex,python,c++,.net/c#,D,Erlang,PHP,Ruby,Object C等

下面我们就一起阅读一下Hessian2.0的文档，文档链接

介绍

Hessian是一个动态类型,二进制序列化,也是网络协议为了对象的定向传输。

设计目标

Hessian是一个动态类型，简洁的，可以移植到各个语言

Hessian协议有以下的设计目标：

1. 它必须自我描述序列化的类型，即不需要外部架构和接口定义
2. 它必须是语言语言独立的，要支持包括脚本语言
3. 它必须是可读可写的在单一的途径
4. 它要尽可能的简洁
5. 它必须是简单的，它可以有效地测试和实施
6. 尽可能的快
7. 必须要支持Unicode编码
8. 它必须支持八位二进制文件，而不是逃避或者用附件
9. 它必须支持加密,压缩,签名,还有事务的上下文

Hessian语法

序列化语法：

```
            # starting production
top         ::= value

            # 8-bit binary data split into 64k chunks
binary      ::= x41 b1 b0 <binary-data> binary # non-final chunk
            ::= 'B' b1 b0 <binary-data>        # final chunk
            ::= [x20-x2f] <binary-data>        # binary data of
                                                  #  length 0-15
            ::= [x34-x37] <binary-data>        # binary data of
                                                  #  length 0-1023

            # boolean true/false
boolean     ::= 'T'
            ::= 'F'

            # definition for an object (compact map)
class-def   ::= 'C' string int string*

            # time in UTC encoded as 64-bit long milliseconds since
            #  epoch
date        ::= x4a b7 b6 b5 b4 b3 b2 b1 b0
            ::= x4b b3 b2 b1 b0       # minutes since epoch

            # 64-bit IEEE double
double      ::= 'D' b7 b6 b5 b4 b3 b2 b1 b0
```

```
            ::= x5b                     # 0.0
            ::= x5c                     # 1.0
            ::= x5d b0                  # byte cast to double
                                        #  (-128.0 to 127.0)
            ::= x5e b1 b0               # short cast to double
            ::= x5f b3 b2 b1 b0         # 32-bit float cast to double

            # 32-bit signed integer
int         ::= 'I' b3 b2 b1 b0
            ::= [x80-xbf]               # -x10 to x3f
            ::= [xc0-xcf] b0            # -x800 to x7ff
            ::= [xd0-xd7] b1 b0         # -x40000 to x3ffff

            # list/vector
list        ::= x55 type value* 'Z'   # variable-length list
       ::= 'V' type int value*    # fixed-length list
            ::= x57 value* 'Z'          # variable-length untyped list
            ::= x58 int value*          # fixed-length untyped list
       ::= [x70-77] type value*  # fixed-length typed list
       ::= [x78-7f] value*       # fixed-length untyped list

            # 64-bit signed long integer
long        ::= 'L' b7 b6 b5 b4 b3 b2 b1 b0
            ::= [xd8-xef]               # -x08 to x0f
            ::= [xf0-xff] b0            # -x800 to x7ff
            ::= [x38-x3f] b1 b0         # -x40000 to x3ffff
            ::= x59 b3 b2 b1 b0         # 32-bit integer cast to long

            # map/object
map         ::= 'M' type (value value)* 'Z'  # key, value map pairs
       ::= 'H' (value value)* 'Z'       # untyped key, value

            # null value
null        ::= 'N'

            # Object instance
object      ::= 'O' int value*
       ::= [x60-x6f] value*

            # value reference (e.g. circular trees and graphs)
ref         ::= x51 int            # reference to nth map/list/object

            # UTF-8 encoded character string split into 64k chunks
string      ::= x52 b1 b0 <utf8-data> string  # non-final chunk
            ::= 'S' b1 b0 <utf8-data>         # string of length
                                              #  0-65535
            ::= [x00-x1f] <utf8-data>         # string of length
                                              #  0-31
            ::= [x30-x34] <utf8-data>         # string of length
                                              #  0-1023

            # map/list types for OO languages
type        ::= string                        # type name
            ::= int                           # type reference

            # main production
value       ::= null
            ::= binary
            ::= boolean
            ::= class-def value
            ::= date
            ::= double
            ::= int
            ::= list
            ::= long
            ::= map
            ::= object
            ::= ref
            ::= string
```

````

## Serialization

Hessian的对象有八种原始类型：

1. 原生二进制数据
2. Boolean
3. 64位毫秒值的日期
4. 64位double
````

```
4. 64位double
5. 32位int
6. 64位long
7. null
8. utf-8的string

它有三种循环的类型：

1. list for lists and arrays
2. map for maps and dictionaries
3. object for objects

最后，他有一种特殊组成：

1. 共享和循环对象引用

Hessian 2.0 有三种内置的map：

1. 一个object/list参考的map
2. 一个类参考定义的map
3. 一个类参考的map

### 4.1 二进制数据
```

binary ::= b b1 b0 binary ::= B b1 b0 ::= [x20-x2f]

```
二进制数据被编码成二进制编码块，'B'代表最后一块，'b'代表任编码块非最后一块的部分，每一个编码块有16 bit的长度。

len = 256 * b1 + b0

### 4.1.1 可压缩：短的二进制
```

Binary data with length less than 15 may be encoded by a single octet length [x20-x2f].

len = code - 0x20

```
当二进制数据少于15的时候，可以用一个字节将他们编码。

### 4.1.2 Binary Examples
```

x20 # zero-length binary data

x23 x01 x02 x03 # 3 octet data

B x10 x00 .... # 4k final chunk of data

b x04 x00 .... # 1k non-final chunk of data

```
### 4.2 boolean
```

boolean ::= T ::= F

```
The octet 'F' represents false and the octet T represents true.

```
### 4.3 日期

时间编码：
```

date ::= x4a b7 b6 b5 b4 b3 b2 b1 b0 ::= x4b b4 b3 b2 b1 b0
```

```
Date represented by a 64-bit long of milliseconds since Jan 1 1970 00:00H, UTC.

### 4.3.1 时间用分钟表示

The second form contains a 32-bit int of minutes since Jan 1 1970 00:00H, UTC.

### 4.3.2 日期例子

x4a x00 x00 x00 xd0 x4b x92 x84 xb8   # 09:51:31 May 8, 1998 UTC

x4b x4b x92 x0b xa0                 # 09:51:00 May 8, 1998 UTC

### 4.4 double

double算法
```

double ::= D b7 b6 b5 b4 b3 b2 b1 b0 ::= x5b ::= x5c ::= x5d b0 ::= x5e b1 b0 ::= x5f b3 b2 b1 b0

```
### 4.4.1 紧凑的二进制的0

可以用x5b代替double的0.0

### 4.4.2 紧凑的二进制的1

可以用x5c代替double的1.0

### 4.4.3 double 的八位字节

double 介于-128到127之间的无符号的可以用两个byte value来替代。

value = (double)b0

### 4.4.4

double 介于-32768和32767之间无符号的double，可以用三个八位字节来替代。

value = (double) (256 * b1 + b0)

### 4.4.5

32位浮点数可以转换成4位8字节二进制数

### 4.4.6 double例子
```

x5b # 0.0 x5c # 1.0

x5d x00 # 0.0 x5d x80 # -128.0 x5d x7f # 127.0

x5e x00 x00 # 0.0 x5e x80 x00 # -32768.0 x5e x7f xff # 32767.0

D x40 x28 x80 x00 x00 x00 x00 x00 # 12.25

```
### 4.5 int
```

int ::= 'I' b3 b2 b1 b0 ::= [x80-xbf] ::= [xc0-xcf] b0 ::= [xd0-xd7] b1 b0

```
一个32bit有符号整数，一个整数以'I'开头，后面跟着4个八个字节.

value = (b3 << 24) + (b2 << 16) + (b1 << 8) + b0;

### 4.5.1 单一的八字节数字

数字介于-16至47之间的可以被编码成一个单独在x80到xbf之间。

value = code - 0x90

### 4.5.2 两个八字节的数字

数字介于 -2048至2047，可以被编码成，两个八字节字符，规则是：

value = ((code - 0xc8) << 8) + b0;

### 4.5.4 用三个八位字节可以表示

Integers between -262144 and 262143 can be encoded in three bytes with the leading byte in the range

value = ((code - 0xd4) << 16) + (b1 << 8) + b0;

### 4.5.4 Integer Examplses
```

x90 # 0 x80 # -16 xbf # 47

xc8 x00 # 0 xc0 x00 # -2048 xc7 x00 # -256 xcf xff # 2047

xd4 x00 x00 # 0 xd0 x00 x00 # -262144 xd7 xff xff # 262143

I x00 x00 x00 x00 # 0 I x00 x00 x01 x2c # 300

```
### 4.6 list

list 语法
```

list ::= x55 type value *'Z' # variable-length list ::= 'V' type int value* # fixed-length list ::= x57 value *'Z' # variable-length untyped list ::= x58 int value* # fixed-length untyped list ::= [x70-77] type value # *fixed-length typed list ::= [x78-7f] value* # fixed-length untyped list

```
一个整齐的list，例如array。这两个list提供一个固定长度和可编长度的list，两个list都有一个类型，这个类型的String必须是一个

每个列表项都添加到引用列表中，以处理共享和循环元素。参见REF元素。

任何希望列表的解析器也必须接受null或共享引用。

类型的有效值没有在本文档中指定，并可能取决于特定的应用程序。例如，在静态类型的语言中实现的一个服务器，它公开了一个Hessian接

### 4.6.1 确定长度的list

Hesssian 2.0  允许一个紧凑的形式列表的连续列表相同的类型，其中的长度是事先已知的。类型和长度由整数编码，其中类型是对较早指

### 4.6.2 List示例

序列化一个int类型的数组，int[] = {0, 1}
```

V # fixed length, typed list x04 [int # encoding of int[] type x92 # length = 2 x90 # integer 0 x91 # integer 1

```
没有类型长度可变的列表 list={0,1}
```

x57 # variable-length, untyped x90 # integer 0 x91 # integer 1 Z

```
定长类型
```

x72 # typed list length=2 x04 [int # type for int[] (save as type #0) x90 # integer 0 x91 # integer 1

x73 # typed list length = 3 x90 # type reference to int[] (integer #0) x92 # integer 2 x93 # integer 3 x94 # integer 4

```
### 4.7 long

long 语法
```

long ::= L b7 b6 b5 b4 b3 b2 b1 b0 ::= [xd8-xef] ::= [xf0-xff] b0 ::= [x38-x3f] b1 b0 ::= x4c b3 b2 b1 b0

```
64位有符号整数。一个长的字节x4c代表（L）随后在后面跟着八个比特。

### 4.7.1 一位八比特能表示的数

long在-8至15是被一个八位比特替代的，在范围xd8至xef。

value = (code - 0xe0)

### 4.7.2 两位八比特能表示的long

long在-2048值2047之间，使用两位byte保存，在范围xf0至xff
value = ((code - 0xf8) << 8 ) + b0

### 4.7.3

三位八比特能表示的数，范围在-262144至262143
value = ((code - 0x3c) << 16) + (b1 << 8) + b0

### 4.7.4

long能用32bite表示的数

value = (b3 << 24) + (b2 << 16) + (b1 << 8) + b0

### long例子
```

xe0 # 0 xd8 # -8 xef # 15

xf8 x00 # 0 xf0 x00 # -2048 xf7 x00 # -256 xff xff # 2047

x3c x00 x00 # 0 x38 x00 x00 # -262144 x3f xff xff # 262143

x4c x00 x00 x00 x00 # 0 x4c x00 x00 x01 x2c # 300

L x00 x00 x00 x00 x00 x00 x01 x2c # 300

```
### 4.8

Map语法

``map         ::= M type (value value)* Z ``

表示序列化的映射，并表示对象。类型元素描述映射的类型。

这个类型可能为空，一个长度为0，程序解释器会自动选择一个类型，如果一个值是没有类型的，对于对象，一个未确认的key将会被忽略。

每一个映射被添加到参考列表中，一些时间，语法解析程序期望一个映射，它必须支持空或者引用,这个类型被服务器选择。


### 4.8.1 Map的例子
```

map = new HashMap(); map.put(new Integer(1), "fee"); map.put(new Integer(16), "fie"); map.put(new Integer(256), "foe");

H # untyped map (HashMap for Java) x91 # 1 x03 fee # "fee"

xa0 # 16 x03 fie # "fie"

xc9 x00 # 256 x03 foe # "foe"

Z

```
集合表示一个java对象
```

public class Car implements Serializable { String color = "aquamarine"; String model = "Beetle"; int mileage = 65536; }

M x13 com.caucho.test.Car # type

x05 color # color field x0a aquamarine

x05 model # model field x06 Beetle

x07 mileage # mileage field I x00 x01 x00 x00 Z

```
### 4.9 null

null语法

null代表一个空指针
'N'代表空的值

### 4.10 object

object 语法
```

class-def ::= 'C' string int string*

object ::= 'O' int value *::= [x60-x6f] value*

```
### 4.10.1 类定义

Hessian2.0有一个紧凑的对象,字段只会序列化一次，以下对象只会序列化它们的值。

对象定义包括强制类型字符串、字段数和字段名。对象定义存储在对象定义映射中，并且将由具有整数引用的对象实例引用。


### 4.10.2 对象实例

Hessian2.0有一个紧凑的对象,字段只会序列化一次，以下对象只会序列化它们的值。

对象实例化根据前面的定义创建一个新对象。整数值是指对象定义。


### 4.10.3 对象示例

对象序列化
```

class Car { String color; String model; }

out.writeObject(new Car("red", "corvette")); out.writeObject(new Car("green", "civic"));

C # object definition (#0) x0b example.Car # type is example.Car x92 # two fields x05 color # color field name x05 model # model field name

O # object def (long form) x90 # object definition #0 x03 red # color field value x08 corvette # model field value

x60 # object def #0 (short form) x05 green # color field value x05 civic # model field value

```

```

enum Color { RED, GREEN, BLUE, }

out.writeObject(Color.RED); out.writeObject(Color.GREEN); out.writeObject(Color.BLUE); out.writeObject(Color.GREEN);

C # class definition #0 x0b example.Color # type is example.Color x91 # one field x04 name # enumeration field is "name"

x60 # object #0 (class def #0) x03 RED # RED value

x60 # object #1 (class def #0) x90 # object definition ref #0 x05 GREEN # GREEN value

x60 # object #2 (class def #0) x04 BLUE # BLUE value

x51 x91 # object ref #1, i.e. Color.GREEN

```
### 4.11 ref

Ref 语法

ref ::= x51 int

指上一个列表、映射或对象实例的整数。当每个列表、映射或对象从输入流中读取时，它被分配到流中的整数位置，即第一个列表或映射是“0

REF可以引用不完全读项。例如，在整个列表被读取之前，循环链表将引用第一个链接。

一个可能的实现是将每个映射、列表和对象添加到数组中，因为它是被读取的。REF将从数组返回相应的值。为了支持循环结构，在填充内容之

每个映射或列表在解析时存储在数组中。REF选择一个存储的对象。第一个对象编号为“0”。

# 4.11.1 Ref 例子
```

list = new LinkedList(); list.data = 1; list.tail = list;

C x0a LinkedList x92 x04 head x04 tail

o x90 # object stores ref #0 x91 # data = 1 x51 x90 # next field refers to itself, i.e. ref #0

```
ref仅涉及到list，map和object。

### 4.12 string

string 语法
```

string ::= x52 b1 b0 string ::= S b1 b0 ::= [x00-x1f] ::= [x30-x33] b0

```
一个16比特，利用utf-8的双字节编码，字符串会按块编码，非最后一块会用'R'来表示，最后一块会用'S'来表示，每一块都有一个16比特

16比特字符的长度，可能与字节的个数不相同。

字符串可能不能成对拆分。

### 4.12.1 短字符串

长度小于32的字符串可能使用单字节编码。

value = code;

### 4.12.2 String 例子
```

x00 # "", empty string x05 hello # "hello" x01 xc3 x83 # "\u00c3"

S x00 x05 hello # "hello" in long form

x52 x00 x07 hello, # "hello, world" split into two chunks x05 world

```
### 4.12 type

type 语法
```

type ::= string ::= int

```
一个map或者list包含的属性，这个属性的属性名将会被标示，用在面向对象的语言中。每一个类型都会被添加到type map中，给未来提供

### 4.14 类型参考

重复类型的String的key，将会用type map来查询先前用过的类型，在解析过程中，这个类型应该是不依赖于任何类型的。

## Bytecode map
```

x00 - x1f # utf-8 string length 0-32 x20 - x2f # binary data length 0-16 x30 - x33 # utf-8 string length 0-1023 x34 - x37 # binary data length 0-1023 x38 - x3f # three-octet compact long (-x40000 to x3ffff) x40 # reserved (expansion/escape) x41 # 8-bit binary data non-final chunk ('A') x42 # 8-bit binary data final chunk ('B') x43 # object type definition ('C') x44 # 64-bit IEEE encoded double

('D') x45 # reserved x46 # boolean false ('F') x47 # reserved x48 # untyped map ('H') x49 # 32-bit signed integer ('I') x4a # 64-bit UTC millisecond date x4b # 32-bit UTC minute date x4c # 64-bit signed long integer ('L') x4d # map with type ('M') x4e # null ('N') x4f # object instance ('O') x50 # reserved x51 # reference to map/list/object - integer ('Q') x52 # utf-8 string non-final chunk ('R') x53 # utf-8 string final chunk ('S') x54 # boolean true ('T') x55 # variable-length list/vector ('U') x56 # fixed-length list/vector ('V') x57 # variable-length untyped list/vector ('W') x58 # fixed-length untyped list/vector ('X') x59 # long encoded as 32-bit int ('Y') x5a # list/map terminator ('Z') x5b # double 0.0 x5c # double 1.0 x5d # double represented as byte (-128.0 to 127.0) x5e # double represented as short (-32768.0 to 327676.0) x5f # double represented as float x60 - x6f # object with direct type x70 - x77 # fixed list with direct length x78 - x7f # fixed untyped list with direct length x80 - xbf # one-octet compact int (-x10 to x3f, x90 is 0) xc0 - xcf # two-octet compact int (-x800 to x7ff) xd0 - xd7 # three-octet compact int (-x40000 to x3ffff) xd8 - xef # one-octet compact long (-x8 to xf, xe0 is 0) xf0 - xff # two-octet compact long (-x800 to x7ff, xf8 is 0) ```