白菜全语言数据类型
封包与解包标准
V1.0

目录

1.0 支持的基本数据类型

- char（1字节有符号整型）
- short（2字节有符号整型）
- int（4字节有符号整型，默认整数型）
- int64（8字节有符号整型）
- uchar（1字节无符号整型）
- ushort（2字节无符号整型）
- uint（4字节无符号整型）
- bool（1字节逻辑型）
- float（4字节单浮点型）
- double（8字节双浮点型，默认浮点型）
- String（字符串类型）
- Bytes（字节集类型）

由给定原始语言（C/C++语言在struct声明中与上边列举类型必须一一对应，如没有的类型需要typedef给出、另外String和Bytes类型需自行实现或需使用白菜字节集）出发进行解析，翻译为目标编程语言的声明、封包和解封包代码。

1.1 关于不支持无符号64位整型的说明

在支持的列表中并无uint64，绝大部分(脚本)编程语言都会具备在有符号基础上一个最大支持的整数界限（即8字节的64位整数）由此来兼容在该位数容器下的任何一种整数类型，但唯独并不支持完整的无符号64位整型，且在实践过程中也不会出现达到这个数的需求（否则都另起定制的大数结构去存储了），关于PHP的最大64位整数不支持无符号的验证可详情：<3.0#int64not>

1.2 关于使用默认的整数型和浮点型

在没有明确给出任何整数型的编程语言 且其内部仅可识别是一个整数型的则默认按4字节有符号整型int去存档；
浮点型则由double作为默认类型存档，如果编程语言本身可以识别该整型容器所使用的确切类型则按确切类型进行封包（提取中用4字节、8字节可直接区分是float还是double）。
——更多存储和解析细节可参考<3.3>、<3.4>

2.0 公开的通用接口

2.1 接口概况（由PHP语言提供）

```
function FB(...$arr)

function deFB(&$dat, &...$arr)

function gFB(&$dat, $i)

function gFBi(&$dat, $i, $signed = true)

function gFBl(&$dat, $i)

function gFBf(&$dat, $i)

function gFBb(&$dat, $i)

function gFB_n(&$dat)

function gFB_p(&$dat, $i, &$size)
```

2.2 详细接口说明

子程序名	返回值类型	公开	备 注		
FB	字节集		将投入的各个数据参数进行封包		
参数名	类 型	参考	可空	数组	备 注
...arr	通用型 | 通用型数组			√	这是一个可变参数，可投入章节<1.0>中介绍的所有基本数据类型进行封包 这里没有'参考(引用)'标记主要是它可以允许投入字面量

子程序名	返回值类型	公开	备 注		
deFB	void		解包由FB封包给出的各个参数 （如果语言不支持传入引用，可使用返回"元组"的形式，如JS、Python）		
参数名	类 型	参考	可空	数组	备 注
dat	字节集	√			由FB封包得到的数据
...arr	通用型 | 通用型数组	√		√	这是一个引用传参的变量组，不一定全部投入原封包中的所有参数， 但一定要按原先参数的顺序投入方可进行引用输出

子程序名	返回值类型	公开	备 注		
gFB	字节集		g表示get的意思，所以意为获取封包的指定参数		
参数名	类 型	参考	可空	数组	备 注
dat	字节集	√			封包数据
i	整数型				第几个参数索引

子程序名	返回值类型	公开	备 注
gFBi	整数型		解析封包数据指定参数为本语言可识别的整数型（且根据封包中给出大小自动适配整数到32位整型结果） 注：如果强类型语言不想多出一步解析size，还可单独扩展gFBi8、gFBi16、gFBi32等（拿到地址后取该类型值）

参数名	类 型	参考	可空	数组	备 注
dat	字节集	√			
i	整数型				
signed=true	逻辑型		√		是否解析为有符号，在脚本语言中因统一使用64位整数，故原始值的符号性质需要明确指示 而强类型语言因有自动类型转换机制该参数可不必设计（直接memcpy该size即可）

子程序名	返回值类型	公开	备 注		
gFBl	长整数型		解析封包数据指定参数为64位整型（如果语言允许，否则一般仅给gFBi也可代表64位）		
参数名	类 型	参考	可空	数组	备 注
dat	字节集	√			
i	整数型				

子程序名	返回值类型	公开	备 注		
gFBf	双精度小数型		解析封包数据指定参数为最大精度的浮点型（根据提取到的大小4字节float转到double，8字节直接提取double） 若对浮点不想产生转换，欲严格匹配还可进一步区分gFBf、gFBd）		
参数名	类 型	参考	可空	数组	备 注
dat	字节集	√			
i	整数型				

子程序名	返回值类型	公开	备 注		
gFBb	逻辑型		解析封包数据指定参数为逻辑型（原始封包数据为1字节） 注：易语言中的逻辑型是4字节，解析结果以本语言为主		
参数名	类 型	参考	可空	数组	备 注
dat	字节集	√			
i	整数型				

子程序名	返回值类型	公开	备 注		
gFB_n	整数型		取值得该封包中的参数个数		
参数名	类 型	参考	可空	数组	备 注
dat	字节集	√			

子程序名	返回值类型	公开	备 注			
gFB_p	整数型 | 指针型		根据各自语言能够支持指针的则返回其指定参数所在的内存首地址， 不支持指针的则返回表示相对起始位置的以字节为单位的整数			
参数名	类 型	参考	可空	数组	备 注	
dat	字节集	√				
i	整数型					
size	整数型	√	√		该参数可选式设计，根据偏好返回值本身也可直接设计为引用视图	

3.0 封包数据类型的存储结构

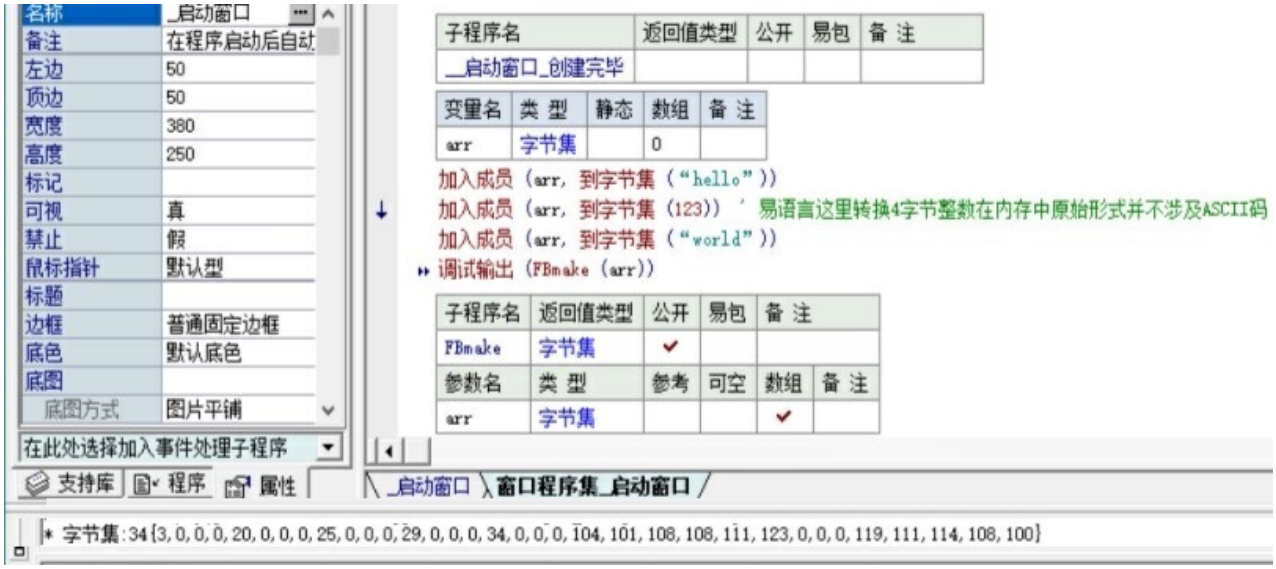
3.1 基本内存模型

```
/*假定封包中有两个参数(int, short)*/
FB:{
    header:{
        0: 参数个数(4字节)
        4: 第1个参数(所在起始封包)相对位置(4字节)
        8: 第2个参数相对位置(4字节)
        12: 表示结束尾的相对位置(4字节)
    }
    16: 第1个参数值(这里int为4字节)
    20: 第2个参数值(这里short为2字节)
}
```

该sizeof(FB)=22字节

这里先给个建议(尽管不是必须的)：

编程语言的开发者在实现完整版**FB**之前，可以先实现基于'字节集数组'作为参数的**FBmake**版本函数，该函数仅仅只是一个最简化封包实现的过渡，来测试各自字节数据的参数在封包后的完整字节集存储分布情况



字节集:34

{3,0,0,0,20,0,0,0,25,0,0,0,29,0,0,0,34,0,0,0,104,101,108,108,111,123,0,0,0,119,111,11

3.2 答疑其中设计的用意

1. Q：如何进行解析参数的提取？
   A：先提取首4个字节得知所有参数个数，从4开始分别偏移地去读4个字节得知各个参数所在的相对位置，用后一个参数的位置减去本次遍历中参数的位置即可得知本次参数的长度、然后就可以从本参数的偏移位置读那么多个字节长度用赋值或内存复制方式输出给deFB中引用传递的变量

2. Q：设计为相对位置好还是给出size好？

A：如果是一个封包在使用时都一定进行全部解析那么size的优势确实更好（因为可以少一次做减法才得知长度的步骤）；
但这样一来想访问任何一个参数位置的话都必须计算累加在它之前的所有参数的大小才能去知道，而gFB函数可以支持进行随机索引的快速访问

3. Q：为什么会多出"结束尾的相对位置"这个4字节占用
A：原则上若知道FB的整体大小则该值确实可以不需要，但为了实现gFB的获取连贯性，它总是会提取当前指定索引参数的下一个位置去作减法，所以从性能实行的角度来看并不需要任何多余判断，这是一种牺牲空间去换时间的做法（当然仅仅只是增加了4字节认为是可以接受的）
此外，在支持访问内存地址的语言中还可以允许在只给出一个封包数据首指针的情况下通过先提前count外加它就能够确定整个封包的完整大小了！

3.3 有符号/无符号整型存储和解析的差别

在计算机中无论是有符号还是无符号整型，其内存的存储结构都是原则上等价一致的，
比如在C语言中将-1分别存储为char和uchar它在内存底层都是8位容器的全1

```
#include "stdafx.h"
#include <字节集.hpp>

void main()
{
	char a = -1;
	uchar b = -1;
	printf(jzjj(字节集(&a,1))); //{255}
	printf(jzjj(字节集(&b,1))); //{255}
}
```

此外对char容器存入130这种超过了自身容器半数的有符号数与uchar的内存模型也是一致的

```
void main()
{
	char a = 130;
	uchar b = 130;
	printf("%d", a); //-126
	printf("%d", b); //130

	printf(jzjj(字节集(&a,1))); //{130}
	printf(jzjj(字节集(&b,1))); //{130}
}
```

也就是说重点不在于字面量所给的数它会对应于有符号或无符号的同等类型下的影响（因为字面量给的值通常认为不会超过所给容器最大无符号上限的值，否则会溢出则无意义，这也是较多现代脚本语言选用8字节作为基本整型存储的依据——当然也有可能使用变长形式存储，最大不会超过8字节），而在于整型值的解析上！

3.4 举例PHP中数值类型的封包实现

原先<1.2>我们说过对于PHP这种无法用精确类型表示整型的脚本语言，我们的方案默认是将其存储为有符号的4字节整数（尽管PHP内部是把它存储为8字节整数了），如下例子展示：

```
<?php
include 'FB.php';

$a = 123;
$fb = FB($a);
echo jzjj($fb);
```

```
Bytes:16{1,0,0,0,12,0,0,0,16,0,0,0,123,0,0,0}
```

那如果我就是要把整型的存储换为其他字节该怎么办？
我们直接构造该字节集就可以了，在PHP中使用pack()函数可以进行构造——基于某一参数为原型构造相关类型的二进制数据（注：PHP中String与Bytes的实现是等价的）；

下面摘抄自FB.php文件中的实现：

```
// region 对PHP不支持的数值类型提供临时封包方案
function i8($a)
{
  return pack('c', $a);
}
function i16($a)
{
  return pack('s', $a);
}
function i32($a)
{
  return pack('l', $a);
}
function i64($a)
{
  return pack('q', $a);
}
function f32($a)
{
  return pack('f', $a);
}
```

应用例子：

```
<?php
include 'FB.php';

$a = 123;
$fb = FB(i16($a));
echo jzjj($fb);
```

```
Bytes:14{1,0,0,0,12,0,0,0,14,0,0,0,123,0}
```

PS：之前我们讨论分析过有符号跟无符号整型的存储从底层二进制来看都是一样的，所以尽管PHP

中有大写版的"C、S、L"可以作为无符号类型，但它是适用于解析端的，跟这里搞存储上其实没有区别。

然后说说整型的解析方面，
首先由于PHP底层用于8字节存储，故所有比它位数之下的类型都能完美表示；
在封包的存储过程中由于没有定制任何关于表示确切类型的存储（仅有大小），则读取后的解析是无法判别该数究竟表示为有符号还是无符号，而如果编程语言本身支持确切类型容器了则按照这个类型去解析即可，那对于PHP该怎么搞个方案来实现呢？

先看看例子：

```php
<?php
include 'FB.php';

$a = 255;
$fb = FB(i8($a));

$a_ = I_sign;
deFB($fb, $a_);
echo $a_;      //-1
```

在FB.php中我的实现是：

```php
// region 解包整数类型的声明常量
const I_sign=-1;
const I_usign=1;
// endregion
```

只需提供两个值的标签、预先赋值给要输出的引用变量可确定用户究竟想解析为有符号还是无符号，而容器大小本身由封包数据得知，

```php
function deFB(&$dat, &...$arr)
{
    $i = 0;
    foreach ($arr as &$v){
        if (is_array($v)){
            _arr_get(gFB($dat, $i), $v);
        } elseif (is_string($v)){
            $v = gFB($dat, $i);
        } elseif (is_float($v)){
            $v = gFBf($dat, $i);
        } elseif(is_bool($v)){
            $v = gFBi($dat, $i)!=0;
        }else{
            $v = gFBi($dat, $i, $v<0);
        }
        $i++;
    }
}
```

```php
function gFBi(&$dat, $i, $signed = true)
{
    $p = unpack('l2', $dat, 4*($i+1));
    $size = $p[2]-$p[1];
    $is = $size == 4? 'l':($size == 1? 'c':
        ($size == 2? 's':'q'));
    if (!$signed) $is = strtoupper($is);
    return unpack($is, $dat, $p[1])[1];
}
```

PS：如果是有符号默认按l、c、s、q来解析，如果是无符号则把解析类型改为大写也就是L、C、S、Q，
另外无符号的64位(Q)我也测了，PHP内部确实是不支持的——只不过人家手册上故意写只为了对称示意

下面几个例子可以充分证明不支持无符号64位：

【例子1】—这里边换为deFB也是可以只不过换成PHP更加原生的实现好充分证明这点

```php
include 'FB.php';

$a = 18446744073709551615;
$fb = FB(pack('Q',$a));
echo jzjj($fb); //Bytes:20{1,0,0,0,12,0,0,0,20,0,0,0,0,0,0,0,0,0,0,0}
print_r(unpack('Q', gFB($fb, 0))); //0
```

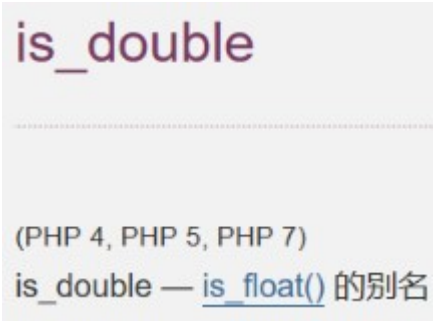
【例子2】

```php
include 'FB.php';

$a = 9223372036854775807;
$fb = FB(pack('Q',$a));
echo jzjj($fb); //Bytes:20{1,0,0,0,12,0,0,0,20,0,0,0,255,255,255,255,255,255,255,127}

print_r(unpack('Q', gFB($fb, 0))); //9223372036854775807
```

【例子3】

```php
include 'FB.php';

$a = 9223372036854775807+1;
$fb = FB(pack('Q',$a));
echo jzjj($fb); //Bytes:20{1,0,0,0,12,0,0,0,20,0,0,0,0,0,0,0,0,0,0,128}

print_r(unpack('Q', gFB($fb, 0))); //-9223372036854775808
```

最后顺带说说PHP中的浮点数：
虽然手册中有提到is_float、is_double，但官方说它们是同义词，



也就是PHP内部默认是以double作为对小数类型存储的，跟我们封包存储整型同理，我这里单独给出了f32函数来进行转换让它内部直接按该字节集进行存储即可。

而解析方面，由于浮点型要么是float要么是double，故只需给出其大小就能够完全区分这两了（大小本身在封包中就已知），且在PHP中也有单独pack类型'f'和'd'进行解析。

> 对于其他可知其确切类型的语言，实际上并不那么麻烦的处理，直接投入FB、deFB一次性就能存储跟解析到位了！

3.5 关于文本型与字节集的封包处理

对绝大部分语言而言文本型和字节集基本都是分家的，比如C语言习惯性用\0作为字符串的终止符（只需提供该内存首地址即可知道字符串的长度），若要表示字节集类型则需要缓冲区中提供长度这么一个字段（一般是首地址的前4个字节），而PHP中String和Bytes的实现则是完全一致的（它均用字节集的实现方案且它本身也支持按二进制的原始内容进行存储和处理），

但其他语言比如python、js这种，文本型和字节集是严格分家的，我们这里不过多讨论它们内部的实现，但凡是数据最终一定能够导出为字节集，**关键是我们要约定文本型这种要统一好编码**（这样在双方解码为自身字符串类型的时候才能有一个依据）；

在FB存储方案中，通通把这两都视为字节集按照<3.1 基本内存模型>进行参数封包，在文本型中建议应当本身统一的编码是UTF-8，然后导出字节集（结尾不强制需或无需空终止）进行存档；
而在FB解析时，只需根据用户投入的变量类型判别本次解包的参数应当是本语言的文本型还是字节集了！

比如下面易语言的封包和解封包实现：

子程序名	返回值类型	公开	备 注
__启动窗口_创建完毕			

变量名	类 型	静态	数组	备 注
fb	字节集			
a	文本型			
b	字节集			

```
fb ＝ FB (“hello”)
调试输出 (fb)

deFB (fb, a)
deFB (fb, b)
调试输出 (a)
调试输出 (b)
```

```
* 字节集:17{1,0,0,0,12,0,0,0,17,0,0,0,104,101,108,108,111}
* “hello”
* 字节集:5{104,101,108,108,111}
```

而PHP中由于String和Bytes不分家，则只需给出声明是字符串即可

```php
<?php
include 'FB.php';

$fb = FB('hello');
echo jzjj($fb);

$a='';
deFB($fb, $a);
echo $a,"\n";
echo jzjj($a);
```

```
Bytes:17{1,0,0,0,12,0,0,0,17,0,0,0,104,101,108,108,111}
hello
Bytes:5{104,101,108,108,111}
```

而在Python中有b""的则表示是bytes类型，没b的则是普通字符串类型(该类型不能直接处理二进制)

```python
fb = FB('hello');
print(jzjj(fb));
a='';
deFB(fb, a);
print(a);

b=b''
deFB(fb, b);
print(jzjj(b));
```

3.6 关于数组的封包方案

对于可知常量大小的基本数据类型的数组而言，实际上无需按照FB原本的头部是偏移位置方案去搞存储（否则太浪费空间了），我们只需用1字节去约定好当前封包的数组是何种类型的就可以了，下面是封包例子

```php
<?php
include 'FB.php';

$arr = [123, 321, 666];
$fb = FB('h', $arr);
echo jzjj($fb);
```

```
Bytes:34{2,0,0,0,16,0,0,0,17,0,0,0,34,0,0,0,104,3,0,0,0,123,0,0,0,65,1,0,0,154,2,0,0,4}
```

数组这一总体在封包层面还是按照<3.1 基本内存模型>去进行的没有问题，而arr数组单独作为一个序列化后的整体作为一个字节集形式存入FB的一个参数中（其数据类型由数组的第一个成员来确定），而其单个数组内部的序列化存储格式也很简单：

```
array:{0~3:数组成员数，4~开始都是该数组的成员按照线性排列的内容，尾部1字节表示该数组的类型}
```

若对于不可知常量大小的类型（如String、Bytes）则需按照FB标准封装格式给出每一个数组成员它的偏移量和尾偏移这部分头部（会多占据(成员数+1)*4的额外空间）

下面是FB规范了尾部1字节的类型的常量值

类型	值
char	-1 (0xFF)
short	-2 (0xFE)
int	-4 (0xFC)
int64	-8 (0xF8)
uchar	1
ushort	2
uint	4
bool	1
float	'f' (0x66)
double	'd' (0x64)
String	's' (0x73)
Bytes	'b' (0x62)

注：String和Bytes更多只是个示意实践中直接根据语言中的声明类型直接判断而无需理会，或者动态语言随意声明一个空数组[]，由封包中的最后一个字节类型得知解析并加入正确类型值到数组中！

4.0 结构体对象的封包方案

4.1 正确选择在语言中表示结构体的类型

在完全编译型语言中用struct/class关键字所表示的结构体确实是最简化的方案（其成员采用内存地址给出，访问则是T(0)开销），但在脚本语言中(带运行时解析特性的)却不是这样，它们的类实例化的对象是采用hash表结构来进行存储和访问，并且实例化一个对象并不仅仅只是创建了类中成员的那些数据类型（还包括一些隐藏的反射信息，继承信息等），所以脚本语言的运行时对象开销并不是T0级别；

故我这里想说的是在你确定好那门编程语言的结构体表示时，除了语言所表示的类跟对象外还可以考虑含键值对的关联数组，下面是一个性能测试来对比在PHP中用class创建结构体和array创建结构体

```php
<?php

class A
{
  function __construct()
  {
    $this->a = 0;
    $this->b = '';
  }
}

class B
{
  function __construct()
  {
    $this->aa = new A();
    $this->b = array_fill(0, 1, '');
  }
}

class C
{
  function __construct()
  {
    $this->aaa = array_fill(0, 2, new A());
    $this->bbb = array_fill(0, 1, new B());
  }
}

function new_A()
{
  return [
    'a' => 0,
    'b' => ''
  ];
}

function new_B()
{
  return [
    'aa' => new_A(),
    'bb' => array_fill(0, 1, '')
  ];
```

```php
}

function new_C()
{
  return [
    'aaa' => array_fill(0, 2, new_A()),
    'bbb' => array_fill(0, 1, new_B())
  ];
}

// --- 1. 创建多个对象的性能测试 ---

// 测试 new C() 创建多个对象
$start = microtime(true);
for ($i = 0; $i<1000000; $i++){
  $obj = new C();
}
$end = microtime(true);
$timeCreateObject = $end-$start;

// 测试 new_C() 创建多个数组对象
$start = microtime(true);
for ($i = 0; $i<1000000; $i++){
  $obj = new_C();
}
$end = microtime(true);
$timeCreateArray = $end-$start;

// --- 2. 访问成员的性能测试 ---

// 访问 C 类对象成员
$cObj = new C();
$start = microtime(true);
for ($i = 0; $i<1000000; $i++){
  $aaa = $cObj->aaa;
}
$end = microtime(true);
$timeAccessObject = $end-$start;

// 访问 new_C() 返回数组成员
$cArray = new_C();
$start = microtime(true);
for ($i = 0; $i<1000000; $i++){
  $aaa = $cArray['aaa'];
}
$end = microtime(true);
$timeAccessArray = $end-$start;

// --- 输出结果 ---

echo "Time to create objects using new C(): ".$timeCreateObject." seconds\n";
echo "Time to create objects using new_C(): ".$timeCreateArray." seconds\n";

echo "Time to access object member (new C()): ".$timeAccessObject." seconds\n";
echo "Time to access array member (new_C()): ".$timeAccessArray." seconds\n";
```

```
Time to create objects using new C(): 0.50881600379944 seconds
Time to create objects using new_C(): 0.2998321056366 seconds
```

```
Time to access object member (new C()): 0.020484924316406 seconds
Time to access array member (new_C()): 0.019924879074097 seconds
```

虽然1000000次的性能差距不大，但也很容易看出确实是array形式更胜一筹，我一开始也想说要不就选数组版吧，

不过考虑到IDE识别成员的易用性，最终成品还是选择了对象版

PS：array_fill()该函数用更短的Arr()封装。

```php
function Arr($v, $count)
{
   return array_fill(0, $count, $v);
}
```

4.2 对其成员展开式生成专用FB方法

先来个简单的结构体看看【PHP、易语言】的情况

【PHP】

```php
<?php
include 'FB.php';

class A
{
  function __construct()
  {
    $this->a = T_short;
    $this->b = T_String;
  }
}

$o = new A;
$o->a = 123;
$o->b = 'hello';
echo $o->a, " | ", $o->b; //123 | hello
```

【易语言】

数据类型名	公开	备 注		
A				
成员名	类 型	传址	数组	备 注
a	短整数型			
b	文本型			

对于编译型语言而言，其在编译后结构体的数据类型等信息是完全丢失的了（易语言仅支持在支持库层面对基本数据类型在运行时进行识别，而结构体对于支持库确实是无能为力），但PHP这种动态语言它依旧可以采用多层遍历等方式去动态提取一个关联数组、类成员的相关key值并在运行时判断它们的类型信息（即便可以要完整写一个通用对象的解析本身的难度和开销就不好说了~）

那我们不可能都保证所有语言都能按照PHP那样去搞解析吧，那如果在不存档一个类的元信息（也就是需要对结构体打表）我们还能否正确从封包字节集的序列化中还原一个结构体？（并且可确保所有语言都通用且依然保持最佳性能开销！）

这里我给该方案取了个名字“对其成员展开式生成专用FB模板方法”，正如此描述，我对每一个类专门生成一个特定的FB封包代码不就可以了吗，在代码中我是可以知道那些成员的类型的以及包括如何处理，比如对A有如下代码FB_A和deFB_A的函数生成

【易语言】

子程序名	返回值类型	公开	备 注		
FB_A	字节集				
参数名	类 型	参考	可空	数组	备 注
o	A	√			

```
返回 (FB (o.a, o.b))
```

子程序名	返回值类型	公开	备 注		
deFB_A					
参数名	类 型	参考	可空	数组	备 注
p	字节集	√			
o	A	√			

```
deFB (p, o.a, o.b)
```

子程序名	返回值类型	公开	备 注
__启动窗口_创建完毕			

变量名	类 型	静态	数组	备 注
o	A			
o_	A			
fb	字节集			

```
o.a = 123
o.b = “hello”
fb = FB_A (o)
调试输出 (fb) //字节集:23{2,0,0,0,16,0,0,0,18,0,0,0,23,0,0,0,123,0,104,101,108,108,111}

deFB_A (fb, o_)
调试输出 (o_.a, o_.b) //123 | “hello”
```

【PHP】
```php
<?php
include 'FB.php';

class A
{
  function __construct()
  {
    $this->a = T_short;
    $this->b = T_String;
  }
}
function FB_A(&$o)
{
  return FB(i16($o->a), $o->b);
}
function deFB_A(&$p, &$o)
{
  deFB($p, $o->a, $o->b);
}

$o = new A;
$o->a = 123;
$o->b = 'hello';

$fb_a = FB_A($o);
echo jzjj($fb_a);

$o_ = new A;
deFB_A($fb_a, $o_);
var_dump($o_);
```

```
Bytes:23{2,0,0,0,16,0,0,0,18,0,0,0,23,0,0,0,123,0,104,101,108,108,111}
array(2) {
  ["a"]=>
  int(123)
  ["b"]=>
  string(5) "hello"
}
```

PHP这里显示int对吗？其实是对的，之前我们一直提到过在PHP内部表示的整型都是统一采用8字节的，关键看对封包的存储（采用i16生成16位字节集）和 解析（直接根据结构体成员赋予的初值给的是有符号和无符号）上

```php
function FB_A(&$o)
{
  return FB(i16($o['a']), $o['b']);
}
```

```php
class A
{
    function __construct()
    {
        $this->a = T_short;
        $this->b = T_String;
    }
}
```

这里我不得不提一下，"T_"给对象专用的类型表示常量（在FB.php中记载）

```php
const T_char = -1;
const T_short = -2;
const T_int = -4;
const T_int64 = -8;
const T_uchar = 1;
const T_ushort = 2;
const T_uint = 4;
const T_bool = false;
const T_float = 4.0;
const T_double = 8.0;
const T_String = '';
const T_Bytes = '';
```

这些类型初值对于整型而言其正负数有表示无符号和有符号的意义，实际上在解包上它与常规数据类型中的负一和正一的处理是一致的

```php
// region 解包整数类型的声明常量
const I_sign=-1;
const I_usign=1;
// endregion
```

核心还是deFB中的处理

```php
function deFB(&$dat, &...$arr)
{
    $i = 0;
    foreach ($arr as &$v){
        if (is_array($v)){
            _arr_get(gFB($dat, $i), $v);
        } elseif (is_string($v)){
            $v = gFB($dat, $i);
        } elseif (is_float($v)){
            $v = gFBf($dat, $i);
        } else{
            $v = gFBi($dat, $i, $v<0);
        }
        $i++;
    }
}
```

从一开始我就已经设想到在结构体类中的成员也会这样去使用！—不由得夸一下自己的天才设计，哈哈！

4.3 含嵌套结构体的表示方案

我先上例子（这里B::bb采用了重定义数组个数的函数Arr，这里表示是含初始1个字节集成员的字节集数组）

【PHP】
```php
<?php
include 'FB.php';

class A
{
  function __construct()
  {
    $this->a = T_short;
    $this->b = T_String;
  }
}
class B
{
  function __construct()
  {
    $this->aa = new A;
    $this->bb = Arr(T_Bytes, 1);
  }
}

function FB_A(&$o)
{
```

```
  return FB(i16($o->a), $o->b);
}

function FB_B(&$o)
{
  return FB(FB_A($o->aa), $o->bb);
}

function deFB_A(&$p, &$o)
{
  deFB($p, $o->a, $o->b);
}

function deFB_B(&$p, &$o)
{
  $p_aa = ''; //这里并非代表字符串，而是提供拼接的缓冲区（如有支持指针或引用的语言应该为其引用该长度的视图）
  deFB($p, $p_aa, $o->bb);
  deFB_A($p_aa, $o->aa);
}

$o = new B;
$o->aa->a = 123;
$o->aa->b = 'hello';
$o->bb = ['world','_hello'];
$fb_b = FB_B($o);

$o_ = new B;
deFB_B($fb_b, $o_);
var_dump($o_);
```

```
class B#3 (2) {
  public $aa =>
  class A#4 (2) {
    public $a => int(123)
    public $b => string(5) "hello"
  }
  public $bb =>
  array(2) {
    [0] => string(5) "world"
    [1] => string(6) "_hello"
  }
}
```

是的，如果B嵌套了结构体那么只好借助原先FB_A进行封包好然后作为B的成员再进行单独的一部分参数封包，对于解包也是同理；

但解包这里其实还有优化的地方（只是对于PHP没有指针则无能为力），那就是deFB_B的第一层解包得到的可以不必是字节集类型的p_aa，因为一旦给出p_aa是字节集那么就不可避免的会产生从投入封包数据中的内存复制，**实际上这种内存复制是完全不需要的**（我之前有说过FB的设计是可以直接投入首指针就能够确定地遍历完整的子封包数据了）

在编译型语言中还可以有如下优化（**deFB的底层实现需要同时判别和支持指针型和字节集buf的投入，由于易语言本身没有指针类型，但有子程序指针故可拿来充当指针型——注意不能搞成整数型，因为没法区分要导出的是存储中的整数值还是把仅把所在地址保存给到上级解包变**

量）

数据类型名	公开	备 注		
A				
成员名	类 型	传址	数组	备 注
a	短整数型			
b	文本型			

数据类型名	公开	备 注		
B				
成员名	类 型	传址	数组	备 注
aa	A			
bb	字节集		1	

窗口程序集名	保 留	保 留	备 注
程序集1			

子程序名	返回值类型	公开	备 注		
FB_A	字节集				
参数名	类 型	参考	可空	数组	备 注
o	A				

```
返回 (FB (o.a, o.b))
```

子程序名	返回值类型	公开	备 注		
deFB_A					
参数名	类 型	参考	可空	数组	备 注
p	子程序指针				
o	A				

```
deFB (p, o.a, o.b)
```

子程序名	返回值类型	公开	备 注		
FB_B	字节集				
参数名	类 型	参考	可空	数组	备 注
o	B				

```
返回 (FB (FB_A (o.aa), o.bb))
```

子程序名	返回值类型	公开	备 注		
deFB_B					
参数名	类 型	参考	可空	数组	备 注
p	子程序指针				
o	B				

变量名	类 型	静态	数组	备 注
p_aa	子程序指针			

```
deFB (p, p_aa, o.bb)
deFB_A (p_aa, o.aa)
```

子程序名	返回值类型	公开	备 注

```
__启动窗口_创建完毕
```

变量名	类 型	静态	数组	备 注
o	B			
o_	B			
zz	字节集			
fb	字节集			

```
o.aa.a ＝ 123
o.aa.b ＝ “hello”
加入成员 (zz, 到字节集 (“world”))
加入成员 (zz, 到字节集 (“_hello”))
o.bb ＝ zz

fb ＝ FB_B (o)
deFB_B (pFB (fb), o_)
调试输出 (o_.aa.a, o_.aa.b)
调试输出 (o_.bb)
```

```
* 123 | “hello”
* 数组:2{字节集:5{119,111,114,108,100},字节集:6{95,104,101,108,108,111}}
```

PS：易语言这里我是先用pFB取得fb这个字节集的指针，之后deFB_B中第一层deFB取得的是o.aa的在封包数据中的内存指针，最后再投入deFB_A来解包得出o.aa

4.4 复合结构体中含嵌套结构体数组的成员

先来看给出的例子结构体

数据类型名	公开	备 注		
A				
成员名	类 型	传址	数组	备 注
a	短整数型			
b	文本型			

数据类型名	公开	备 注		
B				
成员名	类 型	传址	数组	备 注
aa	A			
bb	字节集		1	

数据类型名	公开	备 注		
C				
成员名	类 型	传址	数组	备 注
aaa	A		2	可随时动态调整对象数组长度
bbb	B		1	

其生成的封包方案需要在生成函数中实现将结构体的数组转换为一个个序列化过后的字节集数组，然后最后再进行对字节集数组的封包（我们的FB()函数本身就已经支持对字节集数组的封包）

```
【PHP】

function FB_C(&$o)
{
```

```
  $n = count($o['aaa']);
  $o_aaa = Arr('', $n);
  for ($i = 0; $i<$n; $i++){
    $o_aaa[$i] = FB_A($o['aaa'][$i]);
  }
  $n = count($o['bbb']);
  $o_bbb = Arr('', $n);
  for ($i = 0; $i<$n; $i++){
    $o_bbb[$i] = FB_B($o['bbb'][$i]);
  }
  return FB($o_aaa, $o_bbb);
}
```

【易语言】

子程序名	返回值类型	公开	备 注		
FB_C	字节集				
参数名	类 型	参考	可空	数组	备 注
o	C				

变量名	类 型	静态	数组	备 注
o_aaa	字节集			
i	整数型			
n	整数型			
o_bbb	字节集			

```
n ＝ 取数组成员数 (o.aaa)
重定义数组 (o_aaa, 假, n)
计次循环首 (n, i)
    o_aaa [i] ＝ FB_A (o.aaa [i])
计次循环尾 ()
n ＝ 取数组成员数 (o.bbb)
重定义数组 (o_bbb, 假, n)
计次循环首 (n, i)
    o_bbb [i] ＝ FB_B (o.bbb [i])
计次循环尾 ()
返回 (FB (o_aaa, o_bbb))
```

解包的话需要先使用FB_n提取该对象所在封包字节集的个数（那如何得知一个对象是数组呢？我说过我们在生成代码的预处理阶段其实我们就是神，我们能够得知它的类+型去知道这是一个数组然后才生成的代码），得到个数后就可以进行这种结构体的类型的数组重定义了，然后一个个投入在遍历中用原先就已经写好的相关生成函数进行解析

【PHP】

```
function deFB_C(&$p, &$o)
{
  $p_aaa = $p_bbb = '';
  deFB($p, $p_aaa, $p_bbb);
  $n = gFB_n($p_aaa);
  $o->aaa = Arr(new A, $n);
  for ($i = 0; $i<$n; $i++){
    deFB_A(gFB($p_aaa, $i), $o->aaa[$i]);
  }
```

```
  $n = gFB_n($p_bbb);
  $o->bbb = Arr(new B, $n);
  for ($i = 0; $i<$n; $i++){
    deFB_B(gFB($p_bbb, $i), $o->bbb[$i]);
  }
}
```

【易语言】

子程序名	返回值类型	公开	备 注		
deFB_C					
参数名	类 型	参考	可空	数组	备 注
p	子程序指针				
o	C				

变量名	类 型	静态	数组	备 注
n	整数型			
i	整数型			
p_aaa	子程序指针			
p_bbb	子程序指针			

```
deFB (p, p_aaa, p_bbb)
n = pFB_n (p_aaa)
重定义数组 (o.aaa, 假, n)
计次循环首 (n, i)
    deFB_A (pFB_p (p_aaa, i), o.aaa [i])
计次循环尾 ()
n = pFB_n (p_bbb)
重定义数组 (o.bbb, 假, n)
计次循环首 (n, i)
    deFB_B (pFB_p (p_bbb, i), o.bbb [i])
计次循环尾 ()
```

使用例子

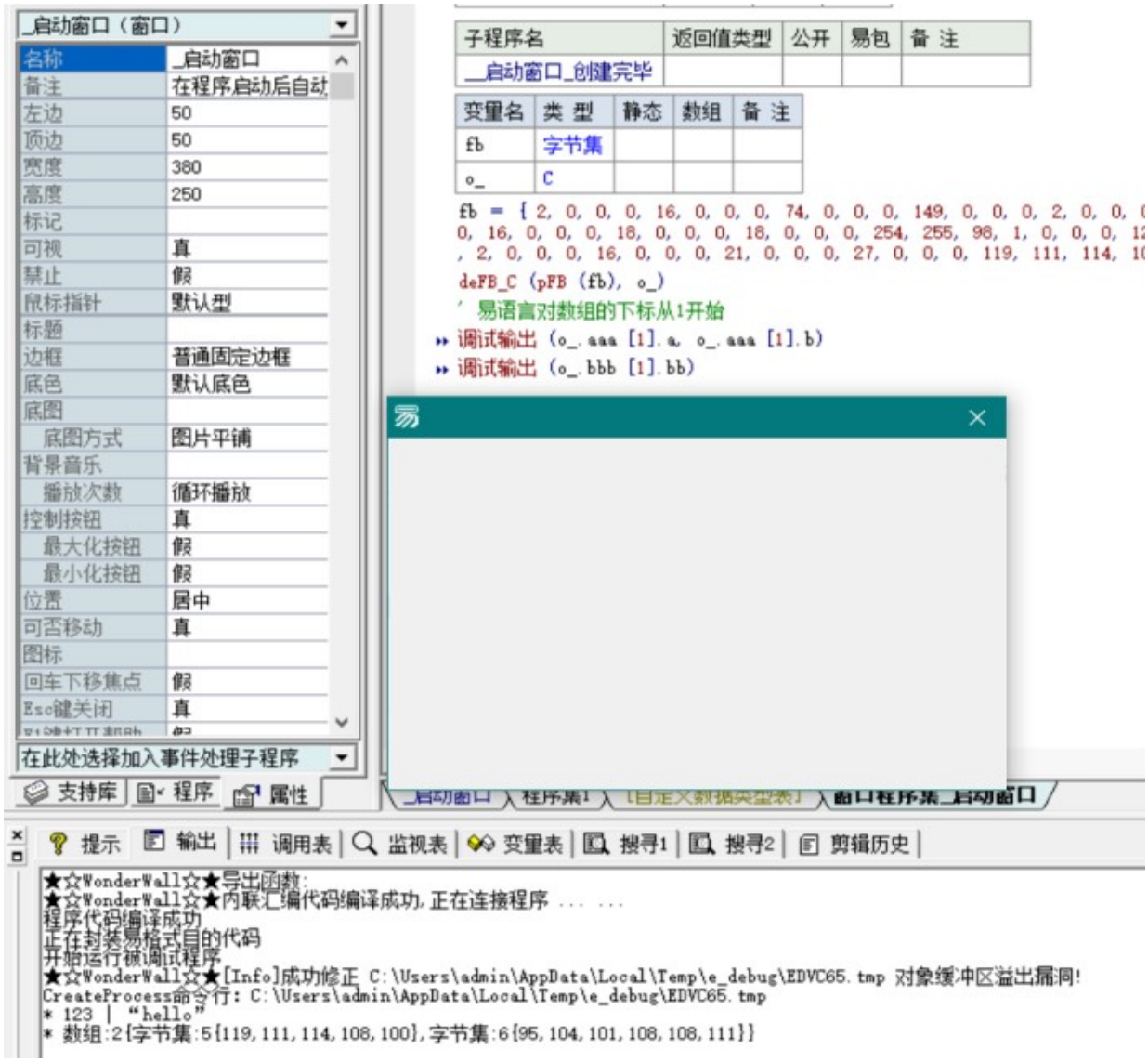
```
<?php
include 'FB_C.php';

TEST_C();

function TEST_C()
{
  $o = new C;
  $o->aaa[0]->a = 123;
  $o->aaa[0]->b = 'hello';
  $o->bbb[0]->bb = ['world','_hello'];
  $fb_c = FB_C($o);

  echo jzjj($fb_c);

  $o_ = new C;
  deFB_C($fb_c, $o_);
  var_dump($o_);
}
```

```
Bytes:154{2,0,0,0,16,0,0,0,79,0,0,0,154,0,0,0,2,0,0,0,16,0,0,0,39,0,0,0,62,0,0,0,2,0,0,
```

```
0,16,0,0,0,18,0,0,0,23,0,0,0,123,0,104,101,108,108,111,2,0,0,0,16,0,0,0,18,0,0,0,23,0,0,
0,123,0,104,101,108,108,111,98,1,0,0,0,12,0,0,0,74,0,0,0,2,0,0,0,16,0,0,0,34,0,0,0,62,0,
0,0,2,0,0,0,16,0,0,0,18,0,0,0,18,0,0,0,254,255,2,0,0,0,16,0,0,0,21,0,0,0,27,0,0,0,119,11
114,108,100,95,104,101,108,108,111,98,98}
```

```
class C#5 (2) {
  public $aaa =>
  array(2) {
    [0] =>
    class A#9 (2) {
      public $a =>
      int(123)
      public $b =>
      string(5) "hello"
    }
    [1] =>
    class A#9 (2) {
      public $a =>
      int(123)
      public $b =>
      string(5) "hello"
    }
  }
  public $bbb =>
  array(1) {
    [0] =>
    class B#6 (2) {
      public $aa =>
      class A#10 (2) {
        ...
      }
      public $bb =>
      array(2) {
        ...
      }
    }
  }
}
```

我们可以把这个字节集给易语言那边进行反序列化（这里pFB去取得字节集的指针，那是因为我们约定这些解包都统一投入"子程序指针"这个类型了，当然你怕麻烦也可以独自改为传入字节集，不过deFB_B这些由于是中间类型要传入字节集那么必然会有内部取子字节集时的二次拷贝）：



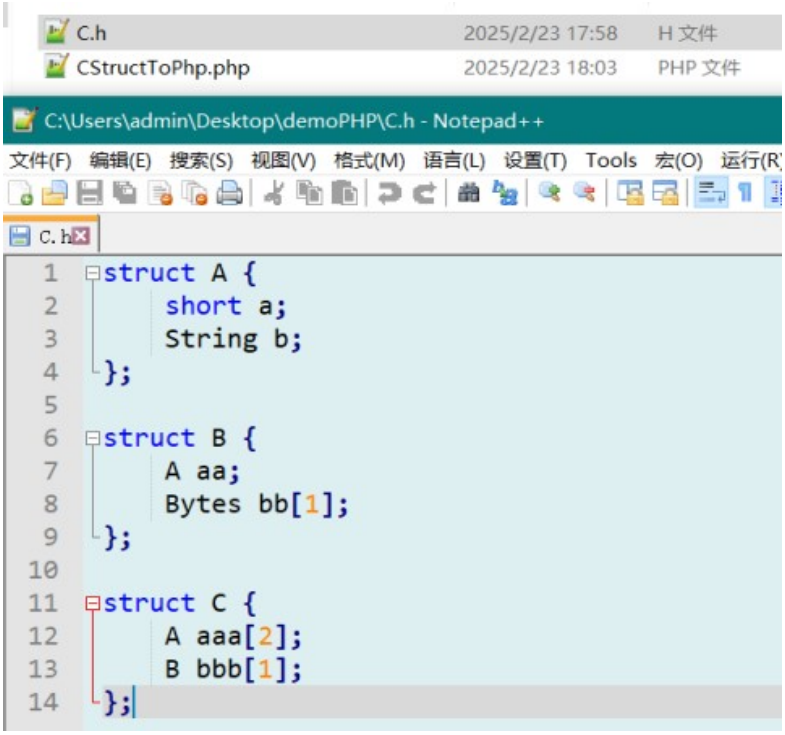
非常完美！

4.5 使用AI辅助生成目标语言的结构体声明文件

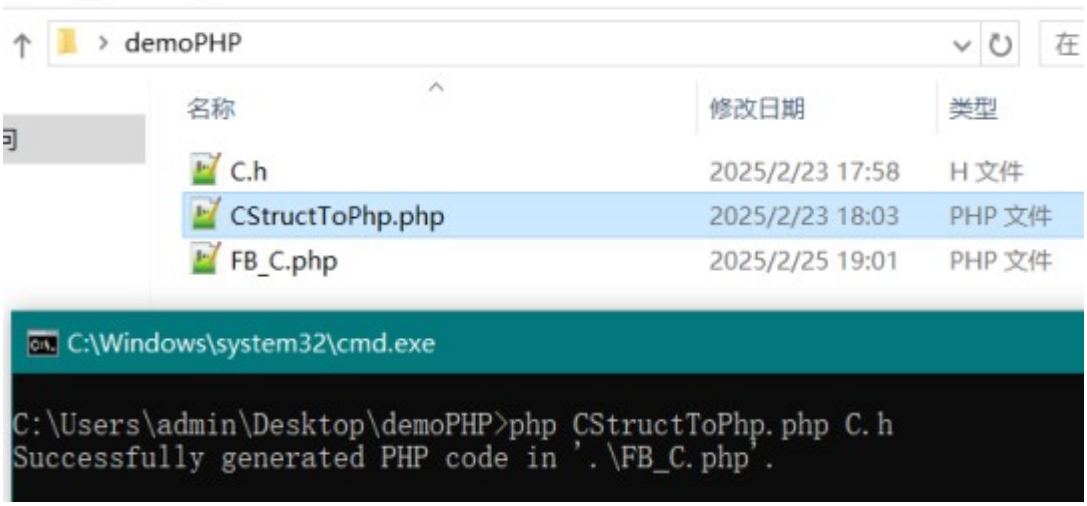
你应该也注意到了，手写 FB_结构/deFB_结构 等其实十分繁琐，一旦结构体嵌套的层次一多手写变得不那么现实，所以我们需要从一个已知的声明源文件（这里我以C语言头文件去写结构体声明作为基底）让AI去翻译给出生成目标语言结构体文件的代码的编写生成代码（我这句话意思是写生成器代码的工具脚本这个活让AI去做）

目前我已经独自完成了易语言和PHP的生成器工具，我下面以C语言转目标PHP为例演示一下用法：（需配置PHP运行时环境php.exe，解压代码仓库中的[php-core.zip]将其路径写入PATH环境变量即可使用，当然你也可以用其他语言实现该类似的工具）

1. CStructToPhp.php脚本是由AI生成的



2. 使用命令行脚本生成后得到FB_C.php

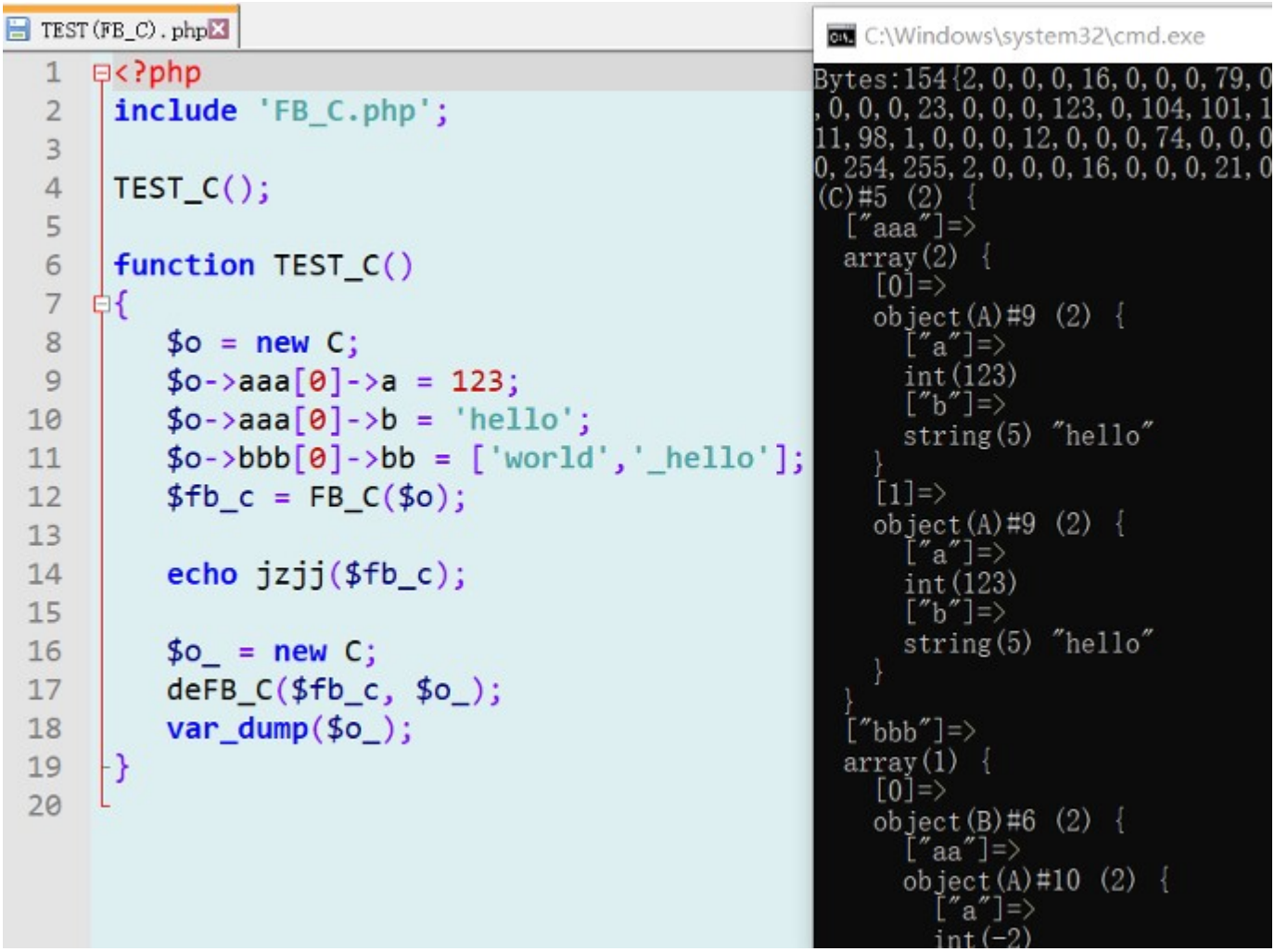


3. 生成的代码内容

FB_C.php

```php
<?php
include_once 'FB.php';

class A
{
    public $a;
    public $b;
    function __construct() {
        $this->a = T_short;
        $this->b = T_String;
    }
}

class B
{
    public $aa;
    public $bb;
    function __construct() {
        $this->aa = new A();
        $this->bb = Arr(T_Bytes, 1);
    }
}

class C
```

4. 在你的项目中使用该C类（依赖的引用只需FB.php、FB_C.php）


TEST(FB_C).php
1 <?php
2 include 'FB_C.php';
3
4 TEST_C();
5
6 function TEST_C()
7 {
8 $o = new C;
9 $o->aaa[0]->a = 123;
10 $o->aaa[0]->b = 'hello';
11 $o->bbb[0]->bb = ['world','_hello'];
12 $fb_c = FB_C($o);
13
14 echo jzjj($fb_c);
15
16 $o_ = new C;
17 deFB_C($fb_c, $o_);
18 var_dump($o_);
19 }
20
C:\Windows\system32\cmd.exe
Bytes:154{2,0,0,0,16,0,0,0,79,0
,0,0,0,23,0,0,0,123,0,104,101,1
11,98,1,0,0,0,12,0,0,0,74,0,0,0
0,254,255,2,0,0,0,16,0,0,0,21,0
(C)#5 (2) {
["aaa"]=>
array(2) {
[0]=>
object(A)#9 (2) {
["a"]=>
int(123)
["b"]=>
string(5) "hello"
}
[1]=>
object(A)#9 (2) {
["a"]=>
int(123)
["b"]=>
string(5) "hello"
}
}
["bbb"]=>
array(1) {
[0]=>
object(B)#6 (2) {
["aa"]=>
object(A)#10 (2) {
["a"]=>
int(-2)