PowerBuilder 中调用 DLL 时参数对应类型分析

PowerBuilder 中可以使用外部 DLL 来扩展程序功能。但在实际使用中，许多人并不了解如何 做好类型对应声明。类型声明错误，甚至调用错误，会导致隐藏 bug，往往在多次调用后系统会崩溃而不自知。本文就 DLL 声明参数做一些分析，希望对一些使用者有一些指导作用。

注：以下所有内容均基于 win32 位系统。

一、从内存和 C/C++指针说起

1、内存和越界

我们知道，对于计算机而言，不管是代码，还是数据，一切都必须在内存里。

我们可以把内存理解为连续的一个大的空间，这个大空间的基础单位是字节。例如 1G 内存就是 1 * 1024 * 1024 * 1024 个字节，对于其中指定位置的内存，用一个数字标记，这个数字我们称之为指针。我们需要一块空闲内存时，向操作系统申请，操作系统会分配一块内存，并且返回这块内存的起始地址，这是一个数字，也就是指针。通常，我们不用关心这个指针的具体值 ，只管使用这个值 ，并且一定要知道分配出来的这块内存的大小，这个必须要清楚。内存用完后，要释放，还给操作系统。如果不释放，会内存耗尽，程序出错退出。

而对于 PowerBuilder 这样的高级语言来说，当然不需要我们自己去分配和释放内存，由语言本身自动完成。

理论知识总是枯燥无味的，下面具体举例来说明一下。

```
void *ptr = 0; //初始时，ptr 这个指针为 0
ptr = malloc(10);   //分配了 10 个字节的内存了空间，首地址返回给 ptr，这时候 ptr
```

是一个数字，指向一块内存

memcpy(ptr,"01234567890123456789012345678 9",10); // 虽然给的"01234567890123456789012345678 9"超过 10 个字节，但长度参数为 10，所以，实际只会复制 10 个字节到 ptr 指向的内存，一切正常。

memcpy(ptr,"01234567890123456789012345678 9",**20**); //如果长度改为 20，而分配的内存只有 10 个字节，此时还是一样成功，看不出来问题，但多出来的 10 个字节，也许复制到其他不可知的指针空间去了，而那个指针空间里面，可能是数据，可能是代码。如果正好是代码，那代码执行到这这部分时，必然会出错，程序会崩溃。通常我们称之为指针**越界**了。这也是指针让人胆战心惊的原因之一。

越界是 PowerBuilder 程序员常犯的错误。也许你认为：我没有分配内存啊，怎么越界了呢？那我举个例子说明：

假如有个医保接口函数声明如下：

Function long ybcall(string InJson,ref string OutJson) library "yb.dll"

调用时：

String ls_in_json,ls_out_json

Ls_in_json = json.tostring()

Ybcall(ls_in_json,ls_out_json)

这段代码有问题吗？当然有问题，程序运行也许一两次正常，但最终必然会崩溃。因为没有为 ls_out_json 分配内存，DLL 内部是会有 memcpy 内存复制动作的，把结果复制到没有分配空间位置，而这个位置默认为 0，0 指向的是程序初始代码那些位置，必然会破坏系统内容，程序崩溃。

那我们把程序改成这样：

String ls_in_json,ls_out_json

Ls_in_json = json.tostring()

ls_out_json = space(1000)

Ybcall(ls_in_json,ls_out_json)

这时候有没有问题呢？答案是不知道。这里虽然给了 1000 个空格的空间，但返回的值长度是不是大于 1000 呢？只有写 DLL 的人知道。如果长度小于 1000 个字符，一切正常，如果长度大于 1000 个字符，那么后果就是未知，也许运行几次没问题，但随时会崩溃。

如何回避这个问题呢？

作为合格的 C/C++程序员，在写 DLL 时，必然要解决这个问题。

方法一：文档约定必须要多少字符的空间。

例如如果约定长度不得小于 800000，那么

ls_out_json = space(800000)

Ybcall(ls_in_json,ls_out_json)

这样就是安全调用。

方法二：ybcall 实现判断。

这个函数应该重新设计成这样：

Function long ybcall(string InJson,ref string OutJson，long OutJsonLen) library "yb.dll"

Ybcall 这个函数 C++内部实现应该是这样：

Int len = strlen(szJson);

If(OutJsonLen < len)

return len;

memcpy(OutJson,szJson,len);

return len;

正确调用方法是：

Long ll_len

ll_len = Ybcall(ls_in_json,ls_out_json,0)

ls_out_json = space(ll_len)

ll_len = Ybcall(ls_in_json,ls_out_json,ll_len)

这样就可以确保不会内存越界。

2、指针与内存块

上面我们说了，指针指向内存中某个已分配的地址，它是一个字节的位置。实际使用中，内存使用是以**内存块**的方式出现的，**内存块**的大小是 1-n 个字节。指针总是指向内存块开头的那个字节的位置。至于这个内存块有多大，具体要看申请多少。而申请内存，有些是是明确申请大小，有些是隐含申请大小。

我们知道，PowerBuilder 是 32 位应用程序。所以基本类型都是 32 位，一个字节是 8 位，32 位就是 4 个字节。也就是说，**基本类型都是 32 位，是 4 个字节**。接下来，我们看看内存块是如何“玩魔术”的。

void *p = malloc(16); //申请 16 个字节

void 是没有类型的，所以大小未知。那么我们可以给它一个类型。

char *p = malloc(16); //申请 16 个字节

p															

此时 p 指向这 16 个字节的第一个字节位置。

char *p1 = p + 1;

此时 p1 指向 p 的下一个字节指针。是这样的：

p	p1														

我们改一下类型

```
int *p = (int *)malloc(16); //分配了 16 个字节
int *p1 = p + 1;
```

此时 p1 在哪呢？也许你理解的内存是这样的：

p	p1														

但不是，实际上是这样的：

p				p1											

为什么是这样呢？因为 int 是 32 位类型，也就是 4 个字节。这个由编译器直接管理了指向位置。

因为是分配了 16 个字节，16/4=4，所以，p 指针的内存可以存 4 个 int 类型数据。

```
p[0] = 1;
p[1] = 1;
p[2] = 1;
p[3] = 1;
```

这样就是存放 4 个字节，如果 p[4] = 1，那就不行了，越界了，程序就有可能崩溃。

对于其他数据类型，long 是 4 字节,short 是 2 字节，int64 是 8 字节，double 是 8 字节,float 是 4 字节。

例如：

```
Int64 *p = (int64 *)malloc(16);
Int64 *p1 = p + 1;
```

p								p1							

对于 int64 是这样的，因为明确指定了使用 64 位，是 32 位的双倍长，所以是 8 个字节。在 PowerBuilder 里，int64 就是 longlong 类型。

如果是一个结构，那这个结构就是它所有成员字节数的总和（这里忽略内存对齐的概念）。例如：

Struct strTest

{

Int a;

Long b;

}

这个结构的大小就是 4+4=8

3、一个特殊的类型：字符串

通常我们说“字符串”，也就是由一串字符组成的一个内存块。在 PowerBuilder 里就是 string 类型，在 c/c++里就是 char *或 wchar_t*，即字符指针指向的一块内存块。

大多数同学都接触过 C 语言，知道 char，对 wchar_t 就比较陌生。那么，这个字符类型，它特殊在哪里呢？答案就是：字符集编码。

在 windows 操作系统里(linux 系统里与 windows 系统不完全一样) ,有 2 种字符类型：多字符 char 和宽字符 wchar_t，即 ansi 和 unicode 字符集编码。它们的区别是：char 是一个字节，wchar_t 是 2 个字节，所以 wchart_t 又可以用 short 表示。

例如“中国”这两个汉字，ansi 编码它是 char 类型的 4 个字节 4 个字符：0xd6 0xd0 0xb9 0xfa ；unicode 编码是 wchar_t 类型的 4 个字节 2 个字符 0x4e2d 0x56fd。取字符串长度，ansy 编码的结果是 4，unicode 编码的结果是 2。

字符串是连续的字符内存空间，以 0 为结尾。Ansi 是 0x00 占一个字节，unicode 是 0x0000 占两个字节。

char 作为字符串类型，同时它是一个字节，所以它也可以代表连续的字节内存空间。往往伴随着还有个字节数表示空间大小。

PowerBuilder 有众多版本，可以分为 2 个种字符集编码,**9.0 及以下版本是 ansi 字符集编码，内部使用的是 char；10.0 及以上版本是 unicode，内部使用的是 wchar_t 类型。**

常用的还有 utf8 编码，内部实际是以 char 串，即字节串来表示。Windows 系统不能直接显示 utf8 编码，必须进行转换后才可以显示。因此，utf8 在 windows 系统里，只是传输时使用，显示时通常会被认为是“乱码”。

二、PowerBuilder 里关于 DLL 参数的使用

1、PowerBuilder 里可以使用什么样的 DLL？

DLL 是 windows 操作系统的动态库。然而，windows 操作系统的动态库有许多类型，里面的函数也有众多函数接口调用约定。使用 COM、OLE、__cdecl、__stdcall 等。COM、OLE 是作为对象使用，OLEObject http ;http.ConnectToNewObject("Msxml2.XMLHTTP ")，然后可以呼叫 http 对象的成员函数。这类调用不在此处讨论。此处只讨论导出函数的 DLL。

作为导出函数的 DLL，有通常有__cdecl、__stdcall 两种调用约定。__cdecl 由 C/C++内部使用，不同库之间可以直接使用。**PowerBuilder 只能使用__stdcall 调用约定导出的函数。**

切记：PowerBuilder 只能使用__stdcall 调用约定导出的函数。

如果不是__stdcall 调用约定的 DLL，PowerBuilder 是无法调用的，会报错，然后崩溃。

2、常见 C/C++类型与 PowerBuilder 类型对应关系

C/C++类型	PowerBuilder 类型
short	int
unsigned short WORD	uint
BOOL bool	Long
int	long
unsigned int UINT	ulong
Long	Long
unsigned long DWORD	Ulong
HANDLE	Ulong
COLORREF	Ulong
Short*	Ref int
BOOL* bool*	不能想当然地使用 ref boolean，PB 里的 boolean 是 PB 的 int 类型，2 字节；C/C++ 里的 BOOL 是定义为 int，在 PB 里对应的是 long，是 4 字节。
unsigned short* WORD*	Ref uint
int*	Ref long
unsigned int* UINT*	Ref ulong
Long*	Ref Long
unsigned long* DWORD*	Ref Ulong
COLORREF*	Ref Ulong

以上是 win32 平台上的固定对应关系。对于字符类型来说，就不能这样简单对应了。

char	范围在-127-127，它是传值 ，不是传地址，不涉及分配内存大小问题，所以 PB 里可以使用 int long,10.0 以上版本可以使用 byte 类型。
BYTE unsigned char	范围是 0-255，它是传值 ，不是传地址，不涉及分配内存大小问题，所以 PB 里可以使用 int long,10.0 以上版本可以使用 byte 类型。

char * unsigned char* BYTE*	字符串的含义非常复杂，**切记：不能单纯理解为 string**。这需要具体分析数据表示的内容。如果仅仅仅表示为类似于姓名之类的字符串内容，可以声明为 string；如果表示一段 内存区域，比如读取身份证的照片，某参数接收照片数据，那绝不能声明为 string 类型，而应该声明为 blob 类型。 PB9.0 及以下版本中，声明为 string 或 blob 即可。 PB10.0 及以上版本中，可以声明为 blob 类型，作为纯字符串内容，可以再进行转换 string(blbData,EncodingAnsi!)转为默认的 string 类型。或者声明为 string 类型，然后声明函数里要加上 alias for “xxxx;ansi”字样，这种方法强烈不建议。应该使用相应的 unicode 版本函数。 如果是输出类型函数，需要添加 ref 关键字。
WCHAR* Wchar_t*	PB9.0 及以下版本中，声明为 string 是不行的，必须声明为 blob。在后再使用 WINAPI 函数 WideCharToMultiByte 转换为 string 类型。 PB10.0 及以上版本中，直接声明为 string 类型即可。 如果是输出类型函数，需要添加 ref 关键字。

3、特别说明

1。、对于 ansi 和 unicode 字符集编码，即 PB9.0 及以下版本和 PB10.0 及以上版本，在处理字符串时，应该使用相应版本的函数。例如：

PB9.0 及以下：

Int GetWindowText(ulong hWnd,ref string lpString,long nMaxCount) library “user32.dll” alias for “GetWindowTextA”

PB10.0 及以上:

Int GetWindowText(ulong hWnd,ref string lpString,long nMaxCount) library “user32.dll” alias for “GetWindowTextW”

请注意这两个声明，实际是 2 个函数，分别是 GetWindowTextA 和 GetWindowTextW，注意尾部的“A”和“W”，“A”表示这是一个 ansi 编码函数，“W”表示这是一个 unicode 编码函数。操作系统已经提供了。

然而 PB 比较傻，将 PB9 程序升级到高版本时，它不会将 GetWindowTextA 改为 GetWindowTextW，而是改为 GetWindowTextA;ansi，注意它添加了一个";ansi “，表示按 ansi 编码标准来调用这个函数。这种只能算是将就使用，许多时候是有问题的。例如：

如果一个窗口的标题是“中国 “，

PB9 里的 GetWindowTextA 返回的是 4，PB10 以上 GetWindowTextW 返回的是 2；GetWindowTextA;ansi 同样还是返回 4。PB10 以上循环按位置取时，按照长度 4 来循环，显然是错误，因为它实际是 2 个字符。

因此，**正确的方法是：PB9 及以下和 PB10 及以上版本里，应尽量使用各自不同编码方式的字符函数。**

2、万能的类型 blob，神一般的存在

我们知道，所有内容都在内存里面，而内存是以字节为单位，内存块有相应大小。Blob 类型的定义就是

```
struct blob{
int len;
char *data
};
```

从定义里可知，blob 就是指定了大小的一内存块。这个内存块里，什么都可以放得下。可以说，上面类型对照表里，PB 相应的类型，可以全部改为 blob。是不是神一般的存在？

我们对字符串做个测试：

PB9 里面“中国“可以是 4 个字节：0xd6 0xd0 0xb9 0xfa，对应的 10 进制数是 214 208 185 250。

blob {5}data //5 字节，因为 4 字节是内容，最后一个字节是空，即字符串结尾

char c1,c2,c3,c4

c1= char(214) ;c2=char(208);c3=char(185);c4=char(250)

blobedit(data,1,c1)

blobedit(data,2,c2)

blobedit(data,3,c3)

blobedit(data,4,c4)

messagebox("",string(data))

PB115 里面“中国“是 2 个字符 0x4e2d 0x56fd，也是 4 个字节，对应的 10 进制数是 78 45 86 253

blob {6}data//6 字节,每字符占 2 字节，因为 4 字节是内容，最后 2 个字节是空，即字符串结尾

byte c1,c2,c3,c4

c1= 214 ;c2=208;c3=185;c4=250

blobedit(data,1,c1)

blobedit(data,2,c2)

blobedit(data,3,c3)

blobedit(data,4,c4)

messagebox("",string(data,encodingansi!))

blob 可以有更多的用法，例如接收一个结构时，可以直接声明为一个大内存块 blob{1024} data，这样去接收这块内容，只要分配的内存大于所需要的内存，调用就是安全的。

与 blob 可以一起使用的还有一个神一般的 WINAPI 函数 RtlMoveMemory。

RtlMoveMemory 可以有各种声明方式，完成各种基于内存的复制操作。

例如可以这样声明：

SUBROUTINE CopyLongToBlob(ref blob dst,ref long n, long size) LIBRARY "kernel32" ALIAS FOR "RtlMoveMemory"
SUBROUTINE CopyBlobToLong(ref long dst,ref blob n, long size) LIBRARY "kernel32" ALIAS FOR "RtlMoveMemory"

然后这样测试：

```
blob {5}data//long 是 4 个字节，内存多于 4 个字节就是安全的

long n1,n2
n1 = 1234

CopyLongToBlob(data,n1,4)//从指向 n1 内存的位置复制 4 个字节到 data 内存

CopyBlobToLong(n2,data,4)//从 data 内存复制 4 个字节到指向 n2 内存的位置

MessageBox("",n2)
```

结果相当于： n2 = n1，实现内存复制

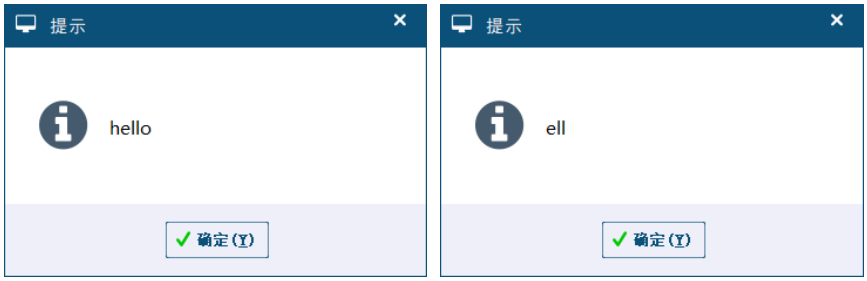
上面的例子是从 blob 或者其他变量 ref 指向的地址开始复制，那么，如何从这个地址指向的后面位置开始复制呢？PB 里没有直接获取地址的函数。pbidea 里将添加一个 AddressOf 函数来获取变量地址。

subroutine CopyString(ref string dst,ulong fromAddress,long size) library "kernel32.dll" alias for "RtlMoveMemory"

PB9.0 里面：

```
string ls_text,ls_copy
ls_text = "hello"
ulong ll_address
uo_utils u
```

```
u = create uo_utils
ll_address = u.AddressOf(ls_text)
if ll_address > 0 then
	messagebox("",string(ll_address,"address"))
	ls_copy = space(2)
	CopyString(ls_copy,ll_address+1,3)
	MessageBox("",ls_copy)
end if
destroy u
```



PB 的 string 函数，可以直接将一个数字地址复制到字符变量里面。

CopyString(ls_copy,ll_address+1,3) 即从第 2 位取 3 个字节。

最后，留一个题目，如果在 PB10 以上版本里面，用 CopyString 从第 2 位取 3 个字符出来，应该怎么写呢？

写在最后：

PowerBuilder 调用 DLL，一定要严格类型转换，必要时必须预留足够内存空间，否则 ，程序很容易崩溃。许多人写的程序，运行一段时间会出现这样那样的意外崩溃，总以为什么冲突，其实不然，主要还是要去查这些 DLL 调用。

作为 DLL，pbidea 比较另类，使用的是 PowerBuilder system library 方式用 C++写成。该方式基于 PowerBuilder SDK，可以直接在 DLL 内部调用 PB 的内存管理方式进行内存分配，因此不需要调用时做内存分配，大大降低了因程序员出现内存分配错误而造成的系统崩溃概率。

大自在

2021/11/24