1 NAME

perlxstut – 写 XSUBs 指南

2 DESCRIPTION

这个指南让你明白创建一个 Perl 扩展的步骤。我假定你已经读过了 perlguts， perlapi 和 perlxs。

我将以一个非常简单的例子开始，然后加入新的例子和物性，使它逐渐变得复杂。 为了让你更容易开始，一些概念要到后面才完全向你解释。

这个指南是从 Unix 的角度写的。其它操作系统可能会不一样，我会尽我所知的 把它列出来。如果你发现我遗漏的地方，请告诉我。

3 SPECIAL NOTES

make 程序

我假定 Perl 配置的 make 程序称为 `make`。在以下的例子中，你需要把 “make”替换为你的 Perl 配置使用的 make 程序。运行 perl -V:make 将知 道你的 make 程序是什么。

版本说明

当写一个通常用途的 Perl 扩展，你应该要考虑到这个扩展可能被用于与你机器 上版本不同的 Perl。当你读这篇文章时，你的 Perl 的版本可能是 5.005 或 者更高。但是使用你的扩展的用户却很可能只有比这古老的版本。

要了解各种可能考虑的兼容性问题，或者应付你机器上的 Perl 版本比这篇文章更古 老的特殊情况，参考“消除这些例子中的问题”这一节。

如果你的扩展使用了一些旧版本的 Perl 不具有的特性，你的用户很希望你能够尽 早提醒他。你可以把这些信息写到 README 文件中。但是现在扩展的安装通 常都是由 CPAN.pm 或者其它工具自动完成了。

在基于 MakeMaker 的安装中，Makefile.PL 提供了第一个进行版本检查的机 会。你可以在 Makefile.PL 文件中像这样写：

```
eval { require 5.007 }
   or die <<EOD;
############
### This module uses frobnication framework which is not available before
### version 5.007 of Perl.  Upgrade your Perl before installing Kara::Mba.
############
EOD
```

动态导入与静态导入

通常会认为如果系统不能动态的导入库的话，就无法创建 XSUBs。其实这是错误。 你是可以创建的，但是你必须把 XSUBs 和 Perl 的其它部分链接起来，创建 一个新的可执行的程序。这与 Perl 4 的情形很类似。

这个指南仍然对这样的系统仍然能够适用。XSUB 的创建机制将检查系统类型， 如果可以动态导入，就创建一个动态导入的库，否则创建一个静态库和一个与这 个静态库链接的可执行程序。

在以下所有的例子中，如果你希望在可以动态导入库的系统上创建一个静态链接 的可执行程序，你可以用“make perl”命令而不是没有参数的“make”。

如果你创建一个这样的静态链接的可执行程序，那你应该用“make test_static”而不是“make test”。在不能创建动态导入库的系统上，就用 “make test”就够了。

4 TUTORIAL

好了，现在让我们开始吧!

例子 1

我们的第一个扩展将是非常的简单。当我们调用这个扩展的函数时，它将输出那 个有名的句子然后返回。

运行“h2xs -A -n Mytest”。这将创建一个名为 Mytest 的目录。如果当前目 录下有一个叫 ext/ 的目录，可能在这个目录下创建。在 Mytest 目录中将创建 几个文件，包括 MANIFEST， Makefile.PL， Mytest.pm， Mytest.xs， test.pl 和 Changes。

MANIFEST 文件内包含所有刚才在 Mytest 目录下创建的文件名。

Makefile.PL 文件应该是这样的：

```
use ExtUtils::MakeMaker;
# See lib/ExtUtils/MakeMaker.pm for details of how to influence
# the contents of the Makefile that is written.
WriteMakefile(
    NAME         => 'Mytest',
    VERSION_FROM => 'Mytest.pm', # finds $VERSION
    LIBS         => [''],   # e.g., '-lm'
    DEFINE       => '',     # e.g., '-DHAVE_SOMETHING'
    INC          => '',     # e.g., '-I/usr/include/other'
);
```

Mytest.pm 文件应该像这样：

```
package Mytest;

use strict;
use warnings;

require Exporter;
require DynaLoader;

our @ISA = qw(Exporter DynaLoader);
# Items to export into callers namespace by default. Note: do not export
# names by default without a very good reason. Use EXPORT_OK instead.
# Do not simply export all your public functions/methods/constants.
our @EXPORT = qw(

);
our $VERSION = '0.01';

bootstrap Mytest $VERSION;

# Preloaded methods go here.

# Autoload methods go after __END__, and are processed by the autosplit program.
```

```
1;
__END__
# Below is the stub of documentation for your module. You better edit it!
```

其它的 .pm 文件是扩展的文档代码样例。
最后，Mytest.xs 文件应该是这样的：

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

MODULE = Mytest        PACKAGE = Mytest
```

在这个文件的最后加上这几行：

```
void
hello()
   CODE:
      printf("Hello, world!\n");
```

以“CODE:”开头的那一行不缩进也没有关系。但是为了可读性，还是建议你缩进 这一行，接下来的几行应当再缩进一层。
现在运行“perl Makefile.PL”。这将创建一个 make 程序需要的真正的 Makefile。它的输出应该是这样的：

```
% perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Mytest
%
```

现在，运行 make 将产生这样的输出（为了清楚一些，过长的输出行被缩短了， 有几行被省略了）：

```
% make
umask 0 && cp Mytest.pm ./blib/Mytest.pm
perl xsubpp -typemap typemap Mytest.xs >Mytest.tc && mv Mytest.tc Mytest.c
Please specify prototyping behavior for Mytest.xs (see perlxs manual)
cc -c Mytest.c
Running Mkbootstrap for Mytest ()
chmod 644 Mytest.bs
LD_RUN_PATH="" ld -o ./blib/PA-RISC1.1/auto/Mytest/Mytest.sl -b Mytest.o
chmod 755 ./blib/PA-RISC1.1/auto/Mytest/Mytest.sl
cp Mytest.bs ./blib/PA-RISC1.1/auto/Mytest/Mytest.bs
chmod 644 ./blib/PA-RISC1.1/auto/Mytest/Mytest.bs
Manifying ./blib/man3/Mytest.3
%
```

你可以忽略了“prototyping behavior”，在perlxs的“The PROTOTYPES: Keyword”一节中将说明这一点。
如果你在 Win32 系统，由于与 C 库函数的 linker 错误导致创建过程中止，请 检查你的 Perl 是否配置为使用 PerlCRT（运行 perl -V:libc 将告诉你答 案）。如果 Perl 是配置为使用 PerlCRT，你看看 PerlCRT.lib 是否复制到 msvcrt.lib 相同的目录下了。必须在相同的位置，编译器才能找到它。 msvcrt.lib 通常在 Visual C 编译器的 lib 目录（例如 C:/DevStudio/VC/lib）。
Perl 有它自己独特而简单的方式来写测试脚本。但是只对于这个例子，我们自 己创建测试脚本。创建一个这样名为的 hello 的文件：

```
#! /opt/perl5/bin/perl

use ExtUtils::testlib;

use Mytest;

Mytest::hello();
```

现在让这个脚本可执行（chmod +x hello），运行脚本，就应该可以看到这 样的输出：

```
% ./hello
Hello, world!
%
```

例子 2

现在加入我们扩展——一个的函数，它接收一个整数为参数，如果是偶数就返回 0，否则返回1。

在 Mytest.xs 的末尾加入：

```
int
is_even(input)
        int     input
    CODE:
        RETVAL = (input % 2 == 0);
    OUTPUT:
        RETVAL
```

同样，以“int input”开头的行不需要缩进，但是缩进使它更好读。在这一行 末尾可以加上一个分号。在“int”和“input”之间可以有任意的空格。

现在重新运行 make 来创建新的库。

重复前面相同的步骤，创建一个 Makefile 然后运行 make。

为了测试我们的扩展，让我们来看看 test.pl 这个文件。这个文件照抄了 Perl 自己的测试结构。通过这个测试脚本，进行一系列的测试来确定扩展的功能：如 果测试是正确的输出“ok”，否则输出“not ok”。修改 BEGIN 块中的 print 语句 来输出“1..4”，然后在文件末尾加入这些代码：

```
print &Mytest::is_even(0) == 1 ? "ok 2" : "not ok 2", "\n";
print &Mytest::is_even(1) == 0 ? "ok 3" : "not ok 3", "\n";
print &Mytest::is_even(2) == 1 ? "ok 4" : "not ok 4", "\n";
```

通过命令“make test”来调用测试脚本。你将看到这样的输出：

```
% make test
PERL_DL_NONLAZY=1 /opt/perl5.004/bin/perl (lots of -I arguments) test.pl
1..4
ok 1
ok 2
ok 3
ok 4
%
```

我们做了些什么呢?

h2xs 是创建扩展的起点。在接下来的例子中我们将看到可以用 h2xs 来读取头 文件并产生连接到 C 函数的模板。

h2xs 在扩展所在的目录里创建一系列的文件。Makefile.PL 文件是一个将产生 真正 Makefile 的 perl 脚本。我们将更深入的了解它。

.pm 和 .xs 文件包含了扩展的主要内容。.xs 文件里是构成扩展的 C 函数。.pm 文件告诉 Perl 怎样导入你的扩展。

产生 Makefile 和运行 make 在当前目录下创建一个叫 blib 的目录(意思是 "build library")。这个目录将包含创建的共享库。当我们测试完毕后,它将安装 在最终的位置。

通过"make test"调用测试脚本的过程中发生一些很重要的事。它使用了所有 的 -I 选项来调用 perl 以保证 perl 能够找到属于扩展的文件。使用 "make test"对于测试你的扩展是很重要的。如果你试图自己运行测试脚本, 这将产生一个错误。另一个使用"make test"的原因是,如果你在测试一个已 经存在的版本升级的模块,这可以保证你测试的是你的新版本,而不是原来那个。

当 Perl 看到 use extension; 时,它将搜索一个具有 .pm 后缀、与 use 后 extension 名字相同的文件。如果找不到这样的文件,Perl 将产生 一个致命错误而终止。默认的搜索路径是在 @INC 数组中。

在我们这个例子中,Mytest.pm 告诉 perl 它将需要 Exporter 和 Dynamic Loader 扩展。它设置 @ISA 和 @EXPORT 数组以及 $VERSION 标量; 最终它告诉 perl 引导这个模块。perl 将调用它的动态导入函数(如果有的话)并 导入共享库。

@ISA 和 @EXPORT 这两个数组是非常重要的。@ISA 数组包含将要用来 搜索当前包中不存在的方法(或者函数)的其它包。这对于面向对象的扩展通常 是十分重要的(我们将在以后谈及这个问题),所以通常是不需要修改的。

@EXPORT 数组告诉 Perl 扩展中的哪些变量和函数将放到调用者的名字空间。 因为你不知道用户是否使用了你的变量名或者函数名,所以有选择的导出是至关 重要。没有好的理由不要用默认导出变量和方法。

作为一个普遍的规则,如果模块是面向对象的,就不要导出任何东西。如果只是 一个方法和变量的集合,你可以通过另一个称为 EXPORT_OK 的数组导出。这 个数组并不自动的把变量和函数名放入名字空间,除非用户有特别需要才这样做。

参考 perlmod 得到更多的内容。

$VERSION 变量保证 .pm 文件和共享库是互相同步的。当你改动 .pm 或者 .xs 文件时,这个变量的值也要相应的增加。

写好测试脚本

写一个好的测试脚本的重要性是不容置疑的。你可以用 Perl 本身使用的 "ok/not ok"风格,这样操作容易而且每个测试的输出非常明确。当你找到并修 正一个错误时,一定要为此增加一个测试。

通过运行"make test",你可以确信你的 test.pl 运行了,而且使用的是正确 的版本。如果你有很多的测试,你也许会模仿 Perl 的测试风格:在扩展所在的 目录下创建一个名叫"t"的目录,添加后缀".t"到你的测试文件。当你运行 "make test"时,所有这些文件都将执行。

例子 3

我们第三个例子将用一个参数作为输入,四舍五入,然后将参数设置为取整 后的值。

在 Mytest.xs 末尾加入:

```
void
```

```
round(arg)
      double  arg
   CODE:
      if (arg > 0.0) {
            arg = floor(arg + 0.5);
      } else if (arg < 0.0) {
            arg = ceil(arg - 0.5);
      } else {
            arg = 0.0;
      }
   OUTPUT:
      arg
```

编辑 Makefile.PL 的相应行，使它成为这样：

```
'LIBS'     => ['-lm'],   # e.g., '-lm'
```

产生 Makefile 并运行 make。修改 BEGIN 块来输出“1..9”，然后在 test.pl 中加入：

```
$i = -1.5; &Mytest::round($i); print $i == -2.0 ? "ok 5" : "not ok 5", "\n";
$i = -1.1; &Mytest::round($i); print $i == -1.0 ? "ok 6" : "not ok 6", "\n";
$i = 0.0; &Mytest::round($i); print $i == 0.0 ? "ok 7" : "not ok 7", "\n";
$i = 0.5; &Mytest::round($i); print $i == 1.0 ? "ok 8" : "not ok 8", "\n";
$i = 1.2; &Mytest::round($i); print $i == 1.0 ? "ok 9" : "not ok 9", "\n";
```

运行“`make test`”后应该输出这九个测试都通过。

注意到在新的测试中，传递给 round 的参数是一个标量。你可能想知道如果传递一个常数或者文字将发生什么。在 test.pl 中暂时加入这样一行：

```
&Mytest::round(3);
```

运行“`make test`”，应该看到 Perl 由于致命错误而退出。Perl 不会让你修改一个常数的值!

有什么新东西?

- 我们对 Makefile.PL 做了一些修改。在这个例子中，我们指定了一个额外的库—— math 库 libm 链接到扩展的共享库中。接下来，我们将介绍如果在 XSUBs 中调用库中的所有函数。
- 传递的值没有做为函数的返回值，而是直接修改这个值。当你看到 round 函数的返回值类型是“void”，你可能已经猜到了这一点。

输入和输出参数

在声明函数的返回值和函数名之后的一行你需要指定传递给 XSUB 的参数。每个声明传入参数的行都以可选的空白开始，以可选的分号结尾。

输出参数列在每个函数的结尾，在 OUTPUT: 指令之后。使用 RETVAL 则告诉 Perl 你希望这个值将作为这个 XSUB 函数的返回值。在例子 3 中，我们想要返回值取代原来传入的值，所以我们把它列在 OUTPUT: 部分，而不是 RETVAL。

XSUBPP 程序

xsubpp 程序将 .xs 文件中的 XS 代码翻译成 C 代码，存储在一个后缀为 .c 的文件中。创建的 C 代码将能在 Perl 中使用 C 的函数。

TYPEMAP 文件

xsubpp 程序使用一些规则将 Perl 的数据（标量，数组等等）转换成 C 的数据类型（int，char 等等）。这些规则存储在 typemap 文件中（$PERLLIB/ExtUtils/typemap）。这个文件可以分为三个部分。

第一部分将各种 C 的数据类型映射到一个对应于 Perl 数据类型的一个名字。第二部分是 xsubpp 用来处理输入参数的 C 代码。第三部分包含 xsubpp 处理输出参数的 C 代码。

让我们来看看为我们的扩展而创建的 .c 文件的一部分。这个文件名为 Mytest.c：

```
XS(XS_Mytest_round)
{
   dXSARGS;
   if (items != 1)
      croak("Usage: Mytest::round(arg)");
   {
      double  arg = (double)SvNV(ST(0));      /* XXXXX */
      if (arg > 0.0) {
            arg = floor(arg + 0.5);
      } else if (arg < 0.0) {
            arg = ceil(arg - 0.5);
      } else {
            arg = 0.0;
      }
      sv_setnv(ST(0), (double)arg);   /* XXXXX */
   }
   XSRETURN(1);
}
```

注意到两行注释了的“XXXXX”。如果你检查 typemap 文件的第一部分，你将看到 doubles 是类型 T_DOUBLE。在 INPUT 一段中，一个 T_DOUBLE 的参数通过调用 SvNV 函数赋值给 arg 变量。类似的，在 OUTPUT 一段中，arg 一旦有了最后的值，通过 sv_setnv 函数把它传递到调用的函数中。这两函数在 perlguts 中有详细解释。对于“ST[0]”的含义将在参数栈中说明。

对于输出参数的警告

一般而言，不要在扩展中修改输入的参数，像例子 3 那样。而应当以一个数组的形式返回多个值，让调用者来处理（之后的例子中，我们将这样做）。然而，为了能与调用已经存在的 C 函数（这些函数通常是修改输入参数的）兼容，这种做还是可以的。

例子 4

在这个例子中，我们要写一个与已经定义的 C 库交互。首先，我们创建一个自己的库，然后让 h2xs 写 .pm 和 .xs 文件。

在与 Mytest 同一层目录内创建一个名为 Mytest2 的新目录。在 Mytest2 目录中，创建另一个叫 mylib 的目录，cd 到这个目录。

在这里，我们要创建一些产生测试库的文件，包括一个 C 源文件和一个头文件，还要在这个目录中创建一个 Makefile.PL。然后使在 Mytest2 中运行 make 时自动执行 Makefile.PL 产生 Makefile。

在 mylib 目录中，创建一个像这样的 mylib.h 文件：

```
#define TESTVAL 4
```

```
extern double   foo(int, long, const char*);
```

同时创建一个 mylib.c 文件：

```
#include <stdlib.h>
#include "./mylib.h"

double
foo(int a, long b, const char *c)
{
        return (a + b + atof(c) + TESTVAL);
}
```

最后创建这样一个 Makefile.PL 文件：

```
use ExtUtils::MakeMaker;
$Verbose = 1;
WriteMakefile(
    NAME   => 'Mytest2::mylib',
    SKIP   => [qw(all static static_lib dynamic dynamic_lib)],
    clean  => {'FILES' => 'libmylib$(LIB_EXT)'},
);

sub MY::top_targets {
        '
all :: static

pure_all :: static

static ::       libmylib$(LIB_EXT)

libmylib$(LIB_EXT): $(O_FILES)
        $(AR) cr libmylib$(LIB_EXT) $(O_FILES)
        $(RANLIB) libmylib$(LIB_EXT)

';
}
```

在“$(AR)”和“$(RANLIB)”的行首，确信你使用的是一个制表符而不是一个空格。如果你用的是空格，make 将不能正常运行。在 Win32 系统中$(AR)的“cr”参数 不是必要的。

现在，我们要创建最高层的 Mytest2 文件。转到 Mytest2 的上层目录，运行下 面的命令：

```
% h2xs -O -n Mytest2 ./Mytest2/mylib/mylib.h
```

这将输出一个警告要覆盖 Mytest2，这没有关系。我们的文件是在 Mytest2/mylib，不会被覆盖的。

h2xs 产生的普通 Makefile.PL 是不会知道 mylib 目录的。我们需要告诉它这 有一个子目录，我们将在里面创建一个库。像这样添加一个 MYEXTLIB 参数到 WriteMakefile 调用中：

```
WriteMakefile(
    'NAME'      => 'Mytest2',
    'VERSION_FROM' => 'Mytest2.pm', # finds $VERSION
    'LIBS'      => [''],   # e.g., '-lm'
    'DEFINE'    => '',     # e.g., '-DHAVE_SOMETHING'
    'INC'       => '',     # e.g., '-I/usr/include/other'
    'MYEXTLIB' => 'mylib/libmylib$(LIB_EXT)',
);
```

然后在最后加入一个函数（这将覆盖已经存在的函数）。记得使用制表符缩进以“cd”开头的那一行！

```
sub MY::postamble {
'
$(MYEXTLIB): mylib/Makefile
	cd mylib && $(MAKE) $(PASSTHRU)
';
}
```

修改 MANIFEST 文件使它准确反映在我们扩展里的内容。“mylib”这一行要用这 三行替换了：

```
mylib/Makefile.PL
mylib/mylib.c
mylib/mylib.h
```

为了使我们的名字空间不被污染，编辑 .pm 文件并修改 @EXPORT 为 EXPORT_OK。最后，在 .xs 文件中，编辑 #include 行：

```
#include "mylib/mylib.h"
```

然后在 .xs 文件末尾加入下面的函数定义：

```
double
foo(a,b,c)
	int		a
	long		b
	const char *	c
   OUTPUT:
	RETVAL
```

现在，我们需要创建一个 typemap 文件，因为 Perl 现在还不支持 const char* 类型。 在 Mytest2 目录中，创建一个名为 typemap 的文件，并写入：

```
const char *	T_PV
```

好了，现在在最顶层运行 Makefile.PL吧。注意在 mylib 目录中也同时生成了 一个 Makefile 文件。运行 make 然后观察到它确实进入 mylib 目录中，在那 里也运行了 make。

编辑 test.pl 脚本，修改 BEGIN 块为输出“1..4”，然后在脚本的末尾加入下面 几行：

```
print &Mytest2::foo(1, 2, "Hello, world!") == 7 ? "ok 2\n" : "not ok 2\n";
print &Mytest2::foo(1, 2, "0.0") == 7 ? "ok 3\n" : "not ok 3\n";
print abs(&Mytest2::foo(0, 0, "-3.4") - 0.6) <= 0.01 ? "ok 4\n" : "not ok 4\n";
```

（当处理浮点数的比较，最好不要检查它们是否相等，而是考察期望值与真实值 的差是否比某个数（称为误差，epsilon）小。在这里，我们使用 0.01。）

运行“make test”，应该是正常的吧。

什么发生了?

不像前一个例子，我们现在是对一个真实的 include 文件使用 h2xs。这对 .pm 和 .xs 文件都带来好处。

- 在 .xs 文件中，现在有一个 #include 指令引入一个到 mylib.h 头文件的绝对路径。我们修改这个路径为相对路径，这样如果需要可以移动扩展的目录。
- 现在一些新的 C 代码加入到 .xs 文件中。constant 函数的功能是让头文件中使用 #define 的值能被 Perl 访问（通过 TESTVAL 或者 &Mytest2::TESTVAL）。一些 XS 代码也能够调用 constant 函数。
- .pm 文件最初将 TESTVAL 名字放在 @EXPORT 数组中。这可能会导致名字冲突。一个更好的规则是如果 #define 只是被 C 的函数使用，而不是用户，那么要从 @EXPORT 中移除。或者，如果你不介意使用变量的全称（“full qualified name”），你可以把 @EXPORT 中的大部分甚至全部放到 @EXPORT_OK 数组中。
- 如果我们的 include 文件也有 #include 指令，h2xs 不能处理。现在还没有好的解决办法。
- 我们还要告诉 Perl 在 mylib 子目录中我们建了一个库。这只要在 WriteMakefile 的调用中添加一个 MYEXTLIB 变量，替换 postamble 函数为 cd 到子目录并运行 make。库的 Makefile.PL 有点复杂，但是也不会太复杂。我们再次替换 postamble 函数，加入我们的代码。这个代码只是简单的指定这个库是一个静态存档库（与动态导入库对应），然后提供创建的命令。

解析 .xs 文件

在“例子 4”中的 .xs 文件包含了一些新的元素。要理解这些元素的含义，请注意这一行：

```
MODULE = Mytest2                PACKAGE = Mytest2
```

在此之前的代码是 plain C 代码，指出要包含的头文件和定义一些方便使用的函数。这一部分是不转换的。编译器会跳过内嵌的 POD 文档，在 C 文件中输出同样的代码。

在此之后的是 XSUB 函数的说明。这些说明由 xsubpp 翻译成 C 代码。翻译后的 C 函数是按 Perl 调用的规则，能被 Perl 的解释器所看的。

特别要注意 constant 函数。在产生的 .xs 文件中这个名字出现了两次：一次在第一部分，作为一个 static C 函数，另一次在第二部分，在定义一个使用这个 static C 函数的 XSUB 接口时。

这对 .xs 文件来说是非常典型的。通常 .xs 文件提供了对已经存在的 C 函数的接口。C 函数是在别的地方定义（在一个外部的库或者在 .xs 文件的第一部分），而这个函数的 Perl 接口（例如“Perl glue”）在 .xs 文件的第二部分。而在“例子 1”、“例子 2”和“例子 3”中，所有的事情都在“Perl glue”中完成，这是一个不符合这一规则的特例。

给 XSUB 减肥

在“例子 4” .xs 文件的第二部分包含了对 XSUB 的如下描述：

```
double
foo(a,b,c)
        int             a
        long            b
        const char *    c
    OUTPUT:
        RETVAL
```

注意到和“例子 1”、“例子 2” 和 “例子 3” 相比，这个描述不包含 Perl 函数 foo 调用使用的真正代码。为了明白这里究竟是怎么回事，我们 可以加入一个 CODE 部分到这个 XSUB：

```
double
foo(a,b,c)
        int           a
        long          b
        const char *    c
    CODE:
        RETVAL = foo(a,b,c);
    OUTPUT:
        RETVAL
```

这两个 XSUB 几乎产生相同的 C 代码：xsubpp 编译器很聪明，它能够从这个 XSUB 描述的头两行推测出 CODE: 部分。那 OUTPUT: 部分呢？这也一样。 OUTPUT: 部分完全可以去掉。当 *CODE:* 或者 *PPCODE:* 没有指定 时，xsubpp 知道它需要产生一个函数调用部分，同样会产生一个 OUTPUT 部 分。这样我们可以把这个 XSUB 缩减成：

```
double
foo(a,b,c)
        int           a
        long          b
        const char *    c
```

是不是也可以对“例子 2” 的 XSUB 进行这样的操作：

```
int
is_even(input)
        int     input
    CODE:
        RETVAL = (input % 2 == 0);
    OUTPUT:
        RETVAL
```

如果你要这样做的话，必须定义一个 C 函数 `int is_even(int input)`。在 解析 .xs 文件一节中，我们知道这个定义应当放在 .xs 文件的第一部分。 这样一个 C 函数就可以：

```
int
is_even(int arg)
{
        return (arg % 2 == 0);
}
```

一个简单的 `#define` 也可以：

```
#define is_even(arg)    ((arg) % 2 == 0)
```

在 .xs 文件的第一部分有了这些之后， “Perl glue”部分就可以缩减成这样

```
int
is_even(input)
        int     input
```

这个把粘合部分和工作部分分离是要付出代价的：如果你想改变 Perl 的接口， 你需要更改代码的两个地方。但是这避免了很多混乱，使工作部分的代码独立于 Perl 调用的规定（事实上，在描述上方的代码不需要特定为 Perl 而写。另一 个版本的 xsubpp 同样可以把它翻译成 TCL glue 或者 Python glue ）。

XSUB 的参数

在完成例子 4 后，我们有了一个简单的方法模拟一些接口不是很好的库。现在 我们要讨论一下传递给 xsubpp 编译器的参数。

当你指定 .xs 文件中函数的参数时，你实际上是对每个参数传递了三个信息。 一是参数的顺序，二是参数的类型，由参数的类型声明（例如，int，char* 等 等）组成。三是库函数调用的名字。

和 Perl 通过引用向函数传递参数不同，C 是按值传递参数的，想要修改“参数” 的数据，实际的参数应该是这个数据的指针。因此这两个 C 函数的声明可能有 完全不同的语义：

```
int string_length(char *s);
int upper_case_char(char *cp);
```

第一个可能是检查 s 指向的字符数组，而第二个可能解除 cp 的引用，只操 纵 *cp（返回值作为成功与否的指示）。在 Perl 里，你需要用不同的方式来 使用这些函数。

你需要通过用 & 替换 * 来告诉 xsubpp 这个信息。& 意指这个参 数需要把地址传递给库函数。上面两个函数如果转换成 XSUB 的话，应该是：

```
int
string_length(s)
        char *  s

int
upper_case_char(cp)
        char    &cp
```

例如，对于这个例子：

```
int
foo(a,b)
        char    &a
        char *  b
```

这个函数的第一个 Perl 参数将当作一个字符赋值给变量 a，它的地址也将传递 给函数 foo。第二个 Perl 参数将当作一个字符串指针赋值给变量 b。b 的值 将传递给函数 foo。xsubpp 产生的函数 foo 的调用是这样的：

```
foo(&a, b);
```

xsubpp 对这样的参数列表的解析是完全一样的：

```
char    &a
char&a
char    & a
```

但是为了更容易理解，最好把“&”放在靠近变量名，远离变量类型，把“*”放在靠 近变量类型，远离变量名。这样做是为了方便明白你将要传递什么到 C 函数中， 要传递的就是最后一栏的东西。

你要用很大努力才能做到传递给函数它想要的数据类型。但是这对将来会减少很多 麻烦。

参数栈

除了第一个例子，如果仔细看前几个例子产生的 C 代码，你会发现“ST(n)”出现了很多次，这里 n 通常是 0。“ST”事实上是一个指向参数栈第 n 个参数的宏。ST(0)就是栈的第一个参数，所以第一个参数传递给了 XSUB，ST(1) 是第二个参数，以此类推。

当你在 .xs 文件中列出 XSUB 的参数时，这就告诉 xsubpp 哪个参数对应于栈内的第几个参数（例如，列出的第一个参数就是第一个参数，等等）。如果你不是像函数中那样列出参数，这将导致很严重的后果。

参数栈内的真实值是传入值的地址。当一个参数列在 OUTPUT 中，对应的它将是在栈内的一个值（例如，当它是第一个参数则是 ST(0)）。你可以通过例子 3 产生的 C 代码验证这一点。round() XSUB 函数的代码中有这样几行：

```
double  arg = (double)SvNV(ST(0));
/* Round the contents of the variable arg */
sv_setnv(ST(0), (double)arg);
```

arg 变量由 ST(0) 中的值来初始化，然后在函数的末尾存储回 ST(0)。

XSUB 是允许返回列表的，不是仅仅标量。这需要以一种精细的方式对栈 ST(0)、ST(1) 等等的值进行操作。更详细的情况请参考 perlxs。

XSUB 也允许避免从 Perl 函数参数到 C 函数参数的自动转换。参考 perlxs。尽管可以自动转换，一些人宁愿通过观察 ST(i) 来自己转换。他们说这样会使 XSUB 的调用的逻辑性更强。（*这里可能不准确，因为我不理解*）就像“给 XSUB 减肥”中分离“Perl glue”和“苦力（workhorse）”的代价一样。

XSUBs are also allowed to avoid automatic conversion of Perl function arguments to C function arguments. See perlxs for details. Some people prefer manual conversion by inspecting ST(i) even in the cases when automatic conversion will do, arguing that this makes the logic of an XSUB call clearer. Compare with §?? for a similar tradeoff of a complete separation of "Perl glue" and "workhorse" parts of an XSUB.

While experts may argue about these idioms, a novice to Perl guts may prefer a way which is as little Perl-guts-specific as possible, meaning automatic conversion and automatic call generation, as in §??. This approach has the additional benefit of protecting the XSUB writer from future changes to the Perl API.

扩展你的扩展

某些时候，你可能想提供一些另外的方法或者函数来使 Perl 和你的扩展之间的接口和更简单或者容易明白。这些函数应该要放在 .pm 文件中。它们是在扩展本身被导入时自动导入还是只在调用时才被导入取决于函数的定义放在 .pm 文件的什么地方。你可以查看 AutoLoader，它给出另外一种保存和导入其它函数的办法。

为你的扩展提供文档

一定要为你的扩展提供文档。文档的内容放在 .pm 文件中。这个文件通常会提供给 pod2man，然后内嵌的文档将转换成 manpage 格式放到 blib 目录中。在安装扩展的时候，将复制到 Perl 的 manpage 目录中。

在 .pm 文件中，你也可以交替的写 Perl 代码和文档。事实上，如果你想使用 autoloading 方法，你必须这样做，就像在 .pm 文件中注释中说的那样。

关于 pod 的格式，请参考 perlpod。

安装你的扩展

一旦你的扩展完成并通过所有的测试之后，安装是非常简单的：你只要运行 “make install” 。你需要有对 Perl 安装目录的写权限或者让你的系统管理员来 为你运行。

还有一个办法，你可以指定释放扩展文件的目录，在 make install 之后加上 “PREFIX=/destination/directory” 就可以了（如果你有 brain-dead 版本的 make，你可以放在 make 和 install 中间）。这对创建一个最终对多个系统发行 的扩展是很有用的。你可以只把文件放到目标目录中，然后对你的目标系统发行 它们。

例子 5

在这个例子中，我们将对参数栈进行一些操作。前面的例子中全部只返回一个值。 现在，我们要创建一个返回一个数组的扩展。

这个扩展是非常 Unix 化（Unix-oriented）的（statfs 结构和 statfs 系统调 用等等）。如果你不是在 Unix 系统下运行，你可以用返回多个值的其它函数来 代替。或者干脆不完成这个例子。如果你修改 XSUB，请记得修改测试来符合你的 修改。

This extension is very Unix-oriented (struct statfs and the statfs system call). If you are not running on a Unix system, you can substitute for statfs any other function that returns multiple values, you can hard-code values to be returned to the caller (although this will be a bit harder to test the error case), or you can simply not do this example. If you change the XSUB, be sure to fix the test cases to match the changes.

回到 Mytest 目录，在 Mytest.xs 末尾加入这些代码：

```
void
statfs(path)
        char *  path
    INIT:
        int i;
        struct statfs buf;

    PPCODE:
        i = statfs(path, &buf);
        if (i == 0) {
                XPUSHs(sv_2mortal(newSVnv(buf.f_bavail)));
                XPUSHs(sv_2mortal(newSVnv(buf.f_bfree)));
                XPUSHs(sv_2mortal(newSVnv(buf.f_blocks)));
                XPUSHs(sv_2mortal(newSVnv(buf.f_bsize)));
                XPUSHs(sv_2mortal(newSVnv(buf.f_ffree)));
                XPUSHs(sv_2mortal(newSVnv(buf.f_files)));
                XPUSHs(sv_2mortal(newSVnv(buf.f_type)));
                XPUSHs(sv_2mortal(newSVnv(buf.f_fsid[0])));
                XPUSHs(sv_2mortal(newSVnv(buf.f_fsid[1])));
        } else {
                XPUSHs(sv_2mortal(newSVnv(errno)));
        }
```

你还要在 .xs 文件的头部，在 include “XSUB.h” 后一行加入下面的代码，

```
#include <sys/vfs.h>
```

然后在 test.pl 中修改 BEGIN 块为 “1..11” ，加入下面一段：

```
@a = &Mytest::statfs("/blech");
print ((scalar(@a) == 1 && $a[0] == 2) ? "ok 10\n" : "not ok 10\n");
@a = &Mytest::statfs("/");
print scalar(@a) == 9 ? "ok 11\n" : "not ok 11\n";
```

这个例子中的新东西

这个例子增加了很多新概念，下面一一进行解释。

- INIT: 指令包含了将在参数栈被转换之后立即加入的代码。C 不允许变量的声明放在函数的任意位置。所以这是通常是最好的办法来声明 XSUB 需要的局部变量（或者你可以把 PPCODE: 全放到一个尖括号里，然后把这个声明放到最顶部）。

- 这个函数还根据调用 statfs 的成功与否返回不同数量的参数。如果发生错误，错误号将做为只有一个元素的数组返回。如果调用成功，将返回一个 9 个元素的数组。由于只传递了一个参数给这个函数，我们对返回的栈进行扩充使之能够放入 9 个元素。

  我们通过 PPCODE: 指令而不是 CODE: 指令来达到这个目的。PPCODE: 告诉 xsubpp 我们将自己操作返回值，并把返回值放到参数栈中。

- 当想要把要返回给调用者的值放到栈中时，我们使用了一系列以“PUSH”开头的宏。它有五种不同的类型，分别针对放置整数、非负整数、双精度数、字符串和 Perl 标量。在这个例子中，我们放置了一个 Perl 标量到栈中（实际上，这是唯一能够用于返回多个值的宏）。

  XPUSH* 宏会自动扩充返回栈防止溢出。你可以按你想让调用的程序看到的顺序放到把它们栈中。

- 放入 XSUB 返回栈的值事实上是暂时的（mortal）SV。所以一旦值被调用程序复制，这些保存返回值的标量将会被释放。如果它们不是暂时的，则在 XSUB 函数返回之后它们还将继续存在，但是已经不能再访问了。这将导致内存泄漏。

- 如果我们更关心代码的可执行性（performance），而不是代码的简洁（compactness），在那条成功的分支我们可以不使用 XPUSHs 宏，而用 PUSHs 宏，这可以在放入返回值之前预先扩充栈：

  ```
  EXTEND(SP, 9);
  ```

  这样做的代价是你需要先计算返回值的数目（虽然过多扩充栈一般也不会有什么坏处，只是需要更多的内存）。

  同样，在那条失败的分支，我们可以不用扩充栈，而直接使用 PUSHs：Perl 函数的引用在 XSUB 的栈内，因此这个栈总是可以放入一个返回值的。

例子 6

在这个例子中，我们将接受一个数组的引用作为输入参数，然后返回一个散列数组的引用，用来演示在 XSUB 中如何操纵复杂的 Perl 数据类型。

这个扩展有点不太自然。它是基于前面例子的代码。它多次调用 statfs 函数，接受一个文件名数组的引用作为输入，然后返回一个包含每个文件数据的散列数组的引用。

回到 Mytest 目录，在 Mytest.xs 文件的末尾加入下面的代码：

```
SV *
multi_statfs(paths)
        SV * paths
    INIT:
        AV * results;
        I32 numpaths = 0;
        int i, n;
        struct statfs buf;
```

```
    if ((!SvROK(paths))
        || (SvTYPE(SvRV(paths)) != SVt_PVAV)
        || ((numpaths = av_len((AV *)SvRV(paths))) < 0))
    {
        XSRETURN_UNDEF;
    }
    results = (AV *)sv_2mortal((SV *)newAV());
CODE:
    for (n = 0; n <= numpaths; n++) {
        HV * rh;
        STRLEN l;
        char * fn = SvPV(*av_fetch((AV *)SvRV(paths), n, 0), l);

        i = statfs(fn, &buf);
        if (i != 0) {
            av_push(results, newSVnv(errno));
            continue;
        }

        rh = (HV *)sv_2mortal((SV *)newHV());

        hv_store(rh, "f_bavail", 8, newSVnv(buf.f_bavail), 0);
        hv_store(rh, "f_bfree",  7, newSVnv(buf.f_bfree),  0);
        hv_store(rh, "f_blocks", 8, newSVnv(buf.f_blocks), 0);
        hv_store(rh, "f_bsize",  7, newSVnv(buf.f_bsize),  0);
        hv_store(rh, "f_ffree",  7, newSVnv(buf.f_ffree),  0);
        hv_store(rh, "f_files",  7, newSVnv(buf.f_files),  0);
        hv_store(rh, "f_type",   6, newSVnv(buf.f_type),   0);

        av_push(results, newRV((SV *)rh));
    }
    RETVAL = newRV((SV *)results);
OUTPUT:
    RETVAL
```

然后在 test.pl 文件中修改 BEGIN 块中的“1..11”为“1..13”，同时加入以下的代码：

```
$results = Mytest::multi_statfs([ '/', '/blech' ]);
print ((ref $results->[0]) ? "ok 12\n" : "not ok 12\n");
print ((! ref $results->[1]) ? "ok 13\n" : "not ok 13\n");
```

这个例子中的新东西

这个例子中又引入了不少新的概念，以下是它们的描述：

- 这个函数不是使用 typemap，而是声明接受一个 SV*（标量）参数，然后返回一 个 SV* 值。在代码中，我们处理（populating）这些标量。由于我们只返回一 个值，所以不必用 PPCODE: 指令，而是使用 CODE: 和 OUTPUT: 指令。
- 当处理引用时，我们一定要小心谨慎。在 INIT: 块中，首先检查 SvROK 的返回值是否为真，这可以判断 paths 是否是合法的引用。然后验证 paths 引 用的对象确实是一个数组。使用 SvRV 来解除 paths 的引用，SvTYPE 来 得到它的类型。还有一个测试是使用 av_len（返回 −1 说明数组为空） 来检 测 paths 所引用的数组是否为空。在三个测试没有都满足的话，XSRETURN_UNDEF 宏用来退出 XSUB 并返回undefined 值。

- 在这个 XSUB 中，我们操纵了几个数组。数组在内部用 AV* 指针表示。操作数组 的函数和宏与 Perl 中了很类似：av_len 返回 AV* 中最大的索引，类似于 $#array；av_fetch 通过索引从数组中得到一个标量的值；av_push 将一 个标量放到数组的末尾，如果必要的话，自动扩充数组。

  我们每次从输入的数组中读入一个路径名，然后把结果按同样的顺序存储在输出 的数组（也就是 results）中。如果 statfs 失败了，放入返回数组中的值是失 败的错误号。如果 statfs 成功，放入数组中的是一个包含 statfs 结构信息的 散列的引用。

  和返回栈相同，因为我们知道有多少元素需要返回，在放入数组之前我们可以预 先扩充返回的数组（执行性能更好一些）：

  ```
  av_extend(results, numpaths);
  ```

- 我们在这个函数中只有一个散列操作，使用 hv_store 对一个标量按一个关 键值存储。散列是用 HV* 指针来表示。和数组一样，XSUB 中操作散列的函数是 和 Perl 中的是一一对应的。参考 perlguts 和 perlapi。

- 为了创建一个引用，我们使用了 newRV 函数。注意在这个例子（很多都一 样）中，可以把一个 AV* 或者 HV* 类型转换成 SV*。这可以使你可以对数组、 散列、标量用相同的函数操作。与此相对，SvRV 函数只返回一个 SV*。因此 可能对于不是标量的情况（用 SvTYPE 检查）需要转换成合适的类型。

- 在这个例子中，xsubpp 只做了一点点事情，Mytest.xs 和 Mytest.c 的差别很 小。

例子 7 (Coming Soon)

XPUSH args 和设置 RETVAL 和将返回值赋值给一个数组

例子 8 (Coming Soon)

设置 $!

例子 9 传递一个打开的文件给 XS

你可能认为给 XS 传递一个文件是很困难的，用到别名等等东西。其实并非如此。

假设为了某个奇怪的理由，我们需要包装标准 C 库函数 fputs()。这是所有 我们需要做的事情：

```
#define PERLIO_NOT_STDIO 0
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

#include <stdio.h>

int
fputs(s, stream)
      char *        s
      FILE *        stream
```

标准的 typemap 完成了真正的工作。

但是你没有看到 perlio 层做的完美工作。这只是可以调用标准输入输出函数 `fputs()`，而对此一无所知。

标准 typemap 提供了三个 PerlIO * 的变体：`InputStream` (T_IN)，`InOutStream` (T_INOUT) 和 `OutputStream` (T_OUT)。单独的 `PerlIO *` 被当作一个 T_INOUT。 The standard typemap offers three variants of PerlIO *: . A bare `PerlIO *` is considered a T_INOUT. If it matters in your code (see below for why it might) #define or typedef one of the specific names and use that as the argument or result type in your XS file.

The standard typemap does not contain PerlIO * before perl 5.7, but it has the three stream variants. Using a PerlIO * directly is not backwards compatible unless you provide your own typemap.

For streams coming from perl the main difference is that `OutputStream` will get the output PerlIO * – which may make a difference on a socket. Like in our example...

For streams being handed to perl a new file handle is created (i.e. a reference to a new glob) and associated with the PerlIO * provided. If the read/write state of the PerlIO * is not correct then you may get errors or warnings from when the file handle is used. So if you opened the PerlIO * as "w" it should really be an `OutputStream` if open as "r" it should be an `InputStream`.

Now, suppose you want to use perlio layers in your XS. We'll use the perlio `PerlIO_puts()` function as an example.

In the C part of the XS file (above the first MODULE line) you have

```
#define OutputStream    PerlIO *
```

or

```
typedef PerlIO *        OutputStream;
```

And this is the XS code:

```
int
perlioputs(s, stream)
        char *          s
        OutputStream    stream
CODE:
        RETVAL = PerlIO_puts(stream, s);
OUTPUT:
        RETVAL
```

We have to use a `CODE` section because `PerlIO_puts()` has the arguments reversed compared to `fputs()`, and we want to keep the arguments the same.

Wanting to explore this thoroughly, we want to use the stdio `fputs()` on a PerlIO *. This means we have to ask the perlio system for a stdio `FILE *`:

```
int
perliofputs(s, stream)
        char *          s
        OutputStream    stream
PREINIT:
        FILE *fp = PerlIO_findFILE(stream);
CODE:
        if (fp != (FILE*) 0) {
                RETVAL = fputs(s, fp);
        } else {
                RETVAL = -1;
        }
OUTPUT:
        RETVAL
```

Note: PerlIO_findFILE() will search the layers for a stdio layer. If it can't find one, it will call PerlIO_exportFILE() to generate a new stdio FILE. Please only call PerlIO_exportFILE() if you want a new FILE. It will generate one on each call and push a new stdio layer. So don't call it repeatedly on the same file. PerlIO()_findFILE will retrieve the stdio layer once it has been generated by PerlIO_exportFILE().

This applies to the perlio system only. For versions before 5.7, PerlIO_exportFILE() is equivalent to PerlIO_findFILE().

消除这些例子中的问题

在文档的开头就提到，如果你如果在这些例子中遇到一些问题的话，可以看看这 些是不是能帮上忙。

- 对于 5.002 prior 到 gamma 版本，例子 1 的测试脚本可能不能正常运行。你 要将“use lib”这一行改为：

  ```
  use lib './blib';
  ```

- 对于 5.002 prior 到 5.002b1h 版本，h2xs 没有自动生成 test.pl。这意味着 你不能用“make test”来运行测试脚本。你需要在“use extension”语句前加入：

  ```
  use lib './blib';
  ```

- 对于 5.000 到 5.001 版本，需要使用下一行而不是上面那一行：

  ```
  BEGIN { unshift(@INC, "./blib") }
  ```

- 这个文档假定你的名为“perl”可执行的程序是 Perl version 5。一些系统可能以 “perl5”为名安装 Perl version 5。

5 See also

更多信息，请查看 perlguts， perlapi， perlxs， perlmod 和 perlpod。

6 Author

Jeff Okamoto <okamoto@corp.hp.com>

Dean Roehrich， Ilya Zakharevich， Andreas Koenig 和 Tim Bunce 修订和协助完成。

PerlIO 的材料由 Lupe Christoph 提供，Nick Ing-Simmons 对此做了一些完善。

Last Changed

2002/05/08

7 TRANSLATORS

YE Wenbin