1 NAME

perlembed - 在 C 程序中嵌入 perl

2 DESCRIPTION

导言

你是想要:

在 Perl 中使用 C?

> 阅读 perlxstut、perlxs、h2xs、perlguts 和 perlapi。

在 Perl 中使用 Unix 程序?

> 阅读反引用符（back-quote）和 L <perlfunc> 中的 `system` 以及 `exec`。

在 Perl 中使用 Perl?

> 阅读 do in perlfunc、eval in perlfunc、require in perlfunc 以及 use in perlfunc。

在 C 中使用 C?

> 重新考虑一下你的设计。

在 C 中使用 Perl?

> 请继续……

路标

- 编译你的 C 程序
- 在你的 C 程序中加入一个 Perl 解释器
- 在 C 程序中调用一个 Perl 函数
- 在 C 程序中对一个 Perl 语句求值
- 在 C 程序中进行 Perl 模式匹配和替换
- 在 C 程序中修改 Perl 参数栈
- 保持一个持久的解释器
- 保持多个解释器实例
- 在 C 程序中使用 Perl 模块，模块本身使用 C 库
- 在 Win32 下内嵌 Perl

编译你的 C 程序

你不是唯一一个在编译本文档的例子时遇到困难的。一个重要规则是：用编译你的 Perl 相同规则来编译程序（对不起，对你大声喊了）。

每个使用 Perl 的 C 程序都必须链接到 perl 库。perl 库是什么？Perl 本身是用 C 来写的，perl library 是一系列编译过的 C 程序，这些将用于创建你的可执行 perl 程序（/usr/bin/perl 或者等价的东西）。（推论：除非 Perl 是在你的机器上编译的，或者合适安装的，否则你将不能在 C 程序中使用 Perl——这也是为什么你不应该从另一台机器中复制 Perl 的可执行程序而不复制 lib 目录。）

当你在 C 中使用 Perl 时，你的 C 程序将（通常是这样）分配、运行然后释放一个 PerlInterpreter 对象，这个对象是在 perl 库中定义的。

如果你的 Perl 足够新，包含了本文档（版本 5.002 或者更新的），那么 perl 库（还有必须的 EXTERN.h 和 perl.h）将在看上去像这样的目录中：

```
/usr/local/lib/perl5/your_architecture_here/CORE
```

或者可能就是

```
/usr/local/lib/perl5/CORE
```

或者可能像这样

```
/usr/opt/perl5/CORE
```

执行这样的语句可以找到 CORE：

```
perl -MConfig -e 'print $Config{archlib}'
```

这是在我的 Linux 机器上编译下一节中例子 Adding a Perl interpreter to your C program 的方法：

```
% gcc -O2 -Dbool=char -DHAS_BOOL -I/usr/local/include
-I/usr/local/lib/perl5/i586-linux/5.003/CORE
-L/usr/local/lib/perl5/i586-linux/5.003/CORE
-o interp interp.c -lperl -lm
```

（就这一行。）在我的 DEC Alpha 使用旧的 5.003_05，这个“咒语”有一点不同：

```
% cc -O2 -Olimit 2900 -DSTANDARD_C -I/usr/local/include
-I/usr/local/lib/perl5/alpha-dec_osf/5.00305/CORE
-L/usr/local/lib/perl5/alpha-dec_osf/5.00305/CORE -L/usr/local/lib
-D__LANGUAGE_C__ -D_NO_PROTO -o interp interp.c -lperl -lm
```

怎样知道应该加上什么呢？假定你的 Perl 中在 5.001 之后，执行 `perl -V` 命令，特别要注意“cc”和“ccflags”信息。

你必须选择合适的编译器（cc、gcc 等等）。在你的机器上：`perl -MConfig -e 'print $Config{cc}'` 将告诉你要使用什么。

你还要为你的机器选择合适的库目录（/usr/local/lib/...）。如果你的编译器抱怨某个函数没有定义，或者它找不到 -lperl，这时你需要更改在 -L 之后的路径。如果它抱怨找不到 EXTERN.h 和 perl.h，你需要更改在 -I 之后的路径。

你可能还要加上一些额外的库。加什么呢？可能是用下面语句输出的那些：

```
perl -MConfig -e 'print $Config{libs}'
```

如果你的 perl 库配置是适当的，已经安装了 ExtUtils::Embed 模块，它会 为你决定所有的这些信息：

```
% cc -o interp interp.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

如果 ExtUtils::Embed 模块不是你的 Perl 发行版的一部分，你可以从 http://www.perl.com/perl/CPAN/modules/by-module/ExtUtils/ 获得。（如果本文档是来自你的 Perl 发行版，那你用的是 5.004 或者更好， 你就已经有这个模块了。）

CPAN 上 ExtUtils::Embed 套装也包含本文档例子的所有源代码，测试，额 外的例子以及其它可能有用的信息。

在 C 程序中加入 Perl 解释器

在某种意义上说，perl（这里指 C 程序）是一个内嵌 Perl（这里指语言）的一 个很好的例子。所以我将用包含在发行版源文件中的 miniperlmain.c 来演 示。这是一个拙劣的、不可移植的 miniperlmain.c 版本，但是包含了内嵌 的本质：

```
#include <EXTERN.h>               /* from the Perl distribution     */
#include <perl.h>                 /* from the Perl distribution     */

static PerlInterpreter *my_perl;  /***    The Perl interpreter    ***/

int main(int argc, char **argv, char **env)
{
    PERL_SYS_INIT3(&argc,&argv,&env);
    my_perl = perl_alloc();
    perl_construct(my_perl);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
    perl_parse(my_perl, NULL, argc, argv, (char **)NULL);
    perl_run(my_perl);
    perl_destruct(my_perl);
    perl_free(my_perl);
    PERL_SYS_TERM();
}
```

注意，我们没有用到 env 指针。通常只是作为 perl_parse 的最后一个 参数提供给它。这里 env 用 NULL 代替了，表示使用当前的环境。 PERL_SYS_INIT3() 和 PERL_SYS_TERM() 宏为 Perl 解释器的运行提供了必要 的、系统特定的 C 运行环境。由于 PERL_SYS_INIT3() 可能修改 env，所 有最好提供 perl_parse() 一个 env 参数。

现在编译成可执行程序（我称之为 interp.c）：

```
% cc -o interp interp.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

在成功编译后，你就可以用 interp 就像 perl 本身一样：

```
% interp
print "Pretty Good Perl \n";
print "10890 - 9801 is ", 10890 - 9801;
<CTRL-D>
Pretty Good Perl
10890 - 9801 is 1089
```

或者

```
% interp -e 'printf("%x", 3735928559)'
deadbeef
```

可以在你的 C 程序中读入和执行 Perl 语句，只需要在调用 perl_run 前放 置文件名在 argv[1] 中。

在 C 程序中调用 Perl 函数

要调用单个 Perl 函数，你可以使用任何一个在 perlcall 中介绍的 call_* 函数。 在这个例子中，我们使用 all_argv。

下面显示一个我称为 showtime.c 的程序：

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

int main(int argc, char **argv, char **env)
{
    char *args[] = { NULL };
    PERL_SYS_INIT3(&argc,&argv,&env);
    my_perl = perl_alloc();
    perl_construct(my_perl);

    perl_parse(my_perl, NULL, argc, argv, NULL);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;

    /*** skipping perl_run() ***/

    call_argv("showtime", G_DISCARD | G_NOARGS, args);

    perl_destruct(my_perl);
    perl_free(my_perl);
    PERL_SYS_TERM();
}
```

这里 showtime 是一个没有参数的 Perl 函数（就是 G_NOARGS），而且 忽略一返回值（就是 G_DISCARD）。在 perlcall 中有讨论这些以及其它 标签。

我在一个称为 showtime.pl 文件中定义这个 showtime 函数：

```
print "I shan't be printed.";

sub showtime {
    print time;
}
```

很简单。现在编译并运行：

```
% cc -o showtime showtime.c `perl -MExtUtils::Embed -e ccopts -e ldopts`

% showtime showtime.pl
818284590
```

产生从 1970 年 1 月 1 日（Unix 纪元的开始）到现在的秒数，这是我写这句 话的时间。

在这个特殊例子中，我们不必调用 perl_run，因为我们设置了 PL_exit_flag PERL_EXIT_DESTRUCT_END，这将在 perl_destruct 中执行 END 块。

如果你想要传递参数给 Perl 函数，你可以在以 NULL 结尾的 args 列表 中加入字符串传递给 call_argv。对于其它数据类型，或者要检查返回值类 型，你需要操作 Perl 参数栈。在 Fiddling with the Perl stack from your C program 中演示了这个过程。

在 C 程序中对 Perl 语句求值

Perl 提供两个 API 函数来对一小段 Perl 代码进行求值。这就是 eval_sv in perlapi 和 eval_pv in perlapi。

在 C 程序中只有这两个函数，你可以执行一段 Perl 代码。你的代码可以任意长，可以包含多个语句，你可以用 use in perlfunc、require in perlfunc、和 do in perlfunc 来引入一个 Perl 文件。

eval_pv 可以对单个的 Perl 字符串求值，然后可以提取出变量转换为 C 类型。下面这个程序 string.c 执行三个 Perl 字符串，第一个提取出一个 int 变量，第二个提取 `float` 变量，第三个提取 `char *` 变量。

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

main (int argc, char **argv, char **env)
{
    STRLEN n_a;
    char *embedding[] = { "", "-e", "0" };

    PERL_SYS_INIT3(&argc,&argv,&env);
    my_perl = perl_alloc();
    perl_construct( my_perl );

    perl_parse(my_perl, NULL, 3, embedding, NULL);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
    perl_run(my_perl);

    /** Treat $a as an integer **/
    eval_pv("$a = 3; $a **= 2", TRUE);
    printf("a = %d\n", SvIV(get_sv("a", FALSE)));

    /** Treat $a as a float **/
    eval_pv("$a = 3.14; $a **= 2", TRUE);
    printf("a = %f\n", SvNV(get_sv("a", FALSE)));

    /** Treat $a as a string **/
    eval_pv("$a = 'rekcaH lreP rehtonA tsuJ'; $a = reverse($a);", TRUE);
    printf("a = %s\n", SvPV(get_sv("a", FALSE), n_a));

    perl_destruct(my_perl);
    perl_free(my_perl);
    PERL_SYS_TERM();
}
```

所有在名字中含有 sv 的奇怪函数都是为了协助将 Perl 标量转换为 C 类型。这在 perlguts 和 perlapi 中有描述。

如果你编译并运行 string.c，你可以用 SvIV() 创建一个 `int`，SvNV() 创建一个 `float`，SvPV() 创建一个字符串，这样可以看到结果。

```
a = 9
a = 9.859600
a = Just Another Perl Hacker
```

在上面的例子中，我们创建了一个全局变量来临时保存求值后计算的结果。也可以，并在大多数情况下最好用 eval_pv() 的返回值。例如：

```
...
STRLEN n_a;
SV *val = eval_pv("reverse 'rekcaH lreP rehtonA tsuJ'", TRUE);
printf("%s\n", SvPV(val,n_a));
...
```

这样不用创建一个全局变量，可以避免污染名字空间，也同样使代码简化。

在 C 程序中进行 Perl 模式匹配和替换

eval_sv() 函数可以对 Perl 代码字符串求值，所以我们可以定义一些函数专门进行匹配和替换：match()，substitute() 和 matches()。

```
I32 match(SV *string, char *pattern);
```

假定有一个字符串和一个模式（例如 `m/clasp/` 或者 `/\b\w*\b/`，在你的 C 程序中可能是这样的 `"/\\b\\w*\\b/"`）。如果字符串匹配一个模式则返回 1，否则返回 0。

```
int substitute(SV **string, char *pattern);
```

假定有一个指向 SV 的指针和 =~ 操作符（例如 `s/bob/robert/g` 或者 `tr[A-Z][a-z]`），substitute() 根据这个操作符修改 SV，返回替换操作的次数。

```
int matches(SV *string, char *pattern, AV **matches);
```

假定有一个 SV，一个模式和一个指向一个空 AV 的指针，match() 在一个列表上下文中对 `$string =~ $pattern` 求值，在 matches 中填充数组，返回匹配的数目。

这是一个使用了三个函数的样例，match.c（过长的行折叠了）：

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

/** my_eval_sv(code, error_check)
** kinda like eval_sv(),
** but we pop the return value off the stack
**/
SV* my_eval_sv(SV *sv, I32 croak_on_error)
{
  dSP;
  SV* retval;
  STRLEN n_a;

  PUSHMARK(SP);
  eval_sv(sv, G_SCALAR);

  SPAGAIN;
  retval = POPs;
  PUTBACK;

  if (croak_on_error && SvTRUE(ERRSV))
    croak(SvPVx(ERRSV, n_a));
```

```
    return retval;
}

/** match(string, pattern)
**
** Used for matches in a scalar context.
**
** Returns 1 if the match was successful; 0 otherwise.
**/

I32 match(SV *string, char *pattern)
{
    SV *command = NEWSV(1099, 0), *retval;
    STRLEN n_a;

    sv_setpvf(command, "my $string = '%s'; $string =~ %s",
            SvPV(string,n_a), pattern);

    retval = my_eval_sv(command, TRUE);
    SvREFCNT_dec(command);

    return SvIV(retval);
}

/** substitute(string, pattern)
**
** Used for =~ operations that modify their left-hand side (s/// and tr///)
**
** Returns the number of successful matches, and
** modifies the input string if there were any.
**/

I32 substitute(SV **string, char *pattern)
{
    SV *command = NEWSV(1099, 0), *retval;
    STRLEN n_a;

    sv_setpvf(command, "$string = '%s'; ($string =~ %s)",
            SvPV(*string,n_a), pattern);

    retval = my_eval_sv(command, TRUE);
    SvREFCNT_dec(command);

    *string = get_sv("string", FALSE);
    return SvIV(retval);
}

/** matches(string, pattern, matches)
**
** Used for matches in a list context.
**
** Returns the number of matches,
** and fills in **matches with the matching substrings
**/
```

```
I32 matches(SV *string, char *pattern, AV **match_list)
{
   SV *command = NEWSV(1099, 0);
   I32 num_matches;
   STRLEN n_a;

   sv_setpvf(command, "my $string = '%s'; @array = ($string =~ %s)",
         SvPV(string,n_a), pattern);

   my_eval_sv(command, TRUE);
   SvREFCNT_dec(command);

   *match_list = get_av("array", FALSE);
   num_matches = av_len(*match_list) + 1; /** assume $[ is 0 **/

   return num_matches;
}

main (int argc, char **argv, char **env)
{
   char *embedding[] = { "", "-e", "0" };
   AV *match_list;
   I32 num_matches, i;
   SV *text;
   STRLEN n_a;

   PERL_SYS_INIT3(&argc,&argv,&env);
   my_perl = perl_alloc();
   perl_construct(my_perl);
   perl_parse(my_perl, NULL, 3, embedding, NULL);
   PL_exit_flags |= PERL_EXIT_DESTRUCT_END;

   text = NEWSV(1099,0);
   sv_setpv(text, "When he is at a convenience store and the "
     "bill comes to some amount like 76 cents, Maynard is "
     "aware that there is something he *should* do, something "
     "that will enable him to get back a quarter, but he has "
     "no idea *what*.  He fumbles through his red squeezey "
     "changepurse and gives the boy three extra pennies with "
     "his dollar, hoping that he might luck into the correct "
     "amount.  The boy gives him back two of his own pennies "
     "and then the big shiny quarter that is his prize. "
     "-RICHH");

   if (match(text, "m/quarter/")) /** Does text contain 'quarter'? **/
     printf("match: Text contains the word 'quarter'.\n\n");
   else
     printf("match: Text doesn't contain the word 'quarter'.\n\n");

   if (match(text, "m/eighth/")) /** Does text contain 'eighth'? **/
     printf("match: Text contains the word 'eighth'.\n\n");
   else
     printf("match: Text doesn't contain the word 'eighth'.\n\n");

   /** Match all occurrences of /wi../ **/
   num_matches = matches(text, "m/(wi..)/g", &match_list);
   printf("matches: m/(wi..)/g found %d matches...\n", num_matches);
```

```
    for (i = 0; i < num_matches; i++)
      printf("match: %s\n", SvPV(*av_fetch(match_list, i, FALSE),n_a));
    printf("\n");

    /** Remove all vowels from text **/
    num_matches = substitute(&text, "s/[aeiou]//gi");
    if (num_matches) {
      printf("substitute: s/[aeiou]//gi...%d substitutions made.\n",
            num_matches);
      printf("Now text is: %s\n\n", SvPV(text,n_a));
    }

    /** Attempt a substitution **/
    if (!substitute(&text, "s/Perl/C/")) {
      printf("substitute: s/Perl/C...No substitution made.\n\n");
    }

    SvREFCNT_dec(text);
    PL_perl_destruct_level = 1;
    perl_destruct(my_perl);
    perl_free(my_perl);
    PERL_SYS_TERM();
}
```

它产生这样的输出（过长的行再次折叠了）：

```
match: Text contains the word 'quarter'.

match: Text doesn't contain the word 'eighth'.

matches: m/(wi..)/g found 2 matches...
match: will
match: with

substitute: s/[aeiou]//gi...139 substitutions made.
Now text is: Whn h s t  cnvnnc str nd th bll cms t sm mnt lk 76 cnts,
Mynrd s wr tht thr s smthng h *shld* d, smthng tht wll nbl hm t gt bck
qrtr, bt h hs n d *wht*.  H fmbls thrgh hs rd sqzy chngprs nd gvs th by
thr xtr pnns wth hs dllr, hpng tht h mght lck nt th crrct mnt.  Th by gvs
hm bck tw f hs wn pnns nd thn th bg shny qrtr tht s hs prz. -RCHH

substitute: s/Perl/C...No substitution made.
```

在 C 程序中填充 Perl 参数栈

大多数计算机教科书对于栈的解释都是重复关于放置咖啡盘的比喻（most computer science textbooks mumble something about spring-loaded columns of cafeteria plates）：最后你放到栈中的东西就是你第一个取出的。这是我们的要做的：C 程序放置一些参数到“Perl 栈”中，当魔术发生时闭上它的眼睛，然后从栈上取出结果——Perl 函数的返回值（That'll do for our purposes: your C program will push some arguments onto "the Perl stack", shut its eyes while some magic happens, and then pop the results--the return value of your Perl subroutine--off the stack.）

首先，你要知道怎样在 C 类型和 Perl 类型之间转换，使用 newSViv()、sv_setnv、newAV() 以及其它它们的朋友。它们在 perlguts 和 perlapi 中有说明。

然后你要知道如何操纵 Perl 参数栈。在 perlcall 中有说明。

一旦你明白这些，在 C 中嵌入 Perl 是很简单的。

因为 C 没有内建的函数进行整数的指数运算，让我们用 Perl 的 ** 运算符实 现它（这比它听上去没用得多，因为 Perl 用 C pow() 函数实现 **）。首 先在 power.pl 中创建一个简短的指数函数：

```
sub expo {
    my ($a, $b) = @_;
    return $a ** $b;
}
```

现在我创建一个 C 程序 power.c，通过 PerlPower()（包含所有必须的 perlguts）将两个参数放到expo() 并取出返回值。深吸一口气：

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

static void
PerlPower(int a, int b)
{
  dSP;                            /* initialize stack pointer      */
  ENTER;                          /* everything created after here */
  SAVETMPS;                       /* ...is a temporary variable.   */
  PUSHMARK(SP);                   /* remember the stack pointer    */
  XPUSHs(sv_2mortal(newSViv(a))); /* push the base onto the stack  */
  XPUSHs(sv_2mortal(newSViv(b))); /* push the exponent onto stack  */
  PUTBACK;                        /* make local stack pointer global */
  call_pv("expo", G_SCALAR);      /* call the function             */
  SPAGAIN;                        /* refresh stack pointer         */
                                  /* pop the return value from stack */
  printf ("%d to the %dth power is %d.\n", a, b, POPi);
  PUTBACK;
  FREETMPS;                       /* free that return value        */
  LEAVE;                       /* ...and the XPUSHed "mortal" args.*/
}

int main (int argc, char **argv, char **env)
{
  char *my_argv[] = { "", "power.pl" };

  PERL_SYS_INIT3(&argc,&argv,&env);
  my_perl = perl_alloc();
  perl_construct( my_perl );

  perl_parse(my_perl, NULL, 2, my_argv, (char **)NULL);
  PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
  perl_run(my_perl);

  PerlPower(3, 4);                    /*** Compute 3 ** 4 ***/

  perl_destruct(my_perl);
  perl_free(my_perl);
  PERL_SYS_TERM();
}
```

编译并运行：

```
% cc -o power power.c `perl -MExtUtils::Embed -e ccopts -e ldopts`

% power
3 to the 4th power is 81.
```

保持一个持久的解释器

当开发一个交互而且（或者）可能是持久运行的应用程序，不要多次分配构建新 的解释器，保持一个持久的解释器是一个好主意。最主要的原因是速度：因为 Perl 只要导入到内存中一次。

尽管这样，当使用一个持久的解释器时要特别小心名字空间和变量作用域。在前 面的例子中，我们在默认的包 main 中使用全局变量。我们很清楚地知道代 码是怎样运行的，并且假定我们能够避免变量冲突和符号表的增长。

假定你的应用程序是一个服务器，它偶尔运行一些文件中的 Perl 代码。你的服 务器是不知道要运行什么代码的。这很危险。

如果文件用 perl_parse() 引入的，编译成一个新创建的解释器，然后接着 用 perl_destruct() 作一次清理，这样就可以屏蔽了大多数的名字空间的问 题。

一个避免名字空间冲突的方法是将文件名转换成一个唯一的包名，然后用 eval in perlfunc 将这段代码编译到这个包中。在下面的例子中，每个文件只 编译一次。或者这个应用程序在一个文件中的符号表不再需要时可能会清除这个 符号表。使用 call_argv in perlapi，我们调用在 persistent.pl 文件中 的 Embed::Persistent::eval_file，传递一个文件名以及一个清除或者缓冲 的标签作为参数。

注意到对于每个使用的文件，这个进程都要不断增长。另外，可能有 AUTOLOAD 函数或者其它条件导致 Perl 符号表的增长。你可能想加入一些逻 辑判断来跟踪进程的大小，或者在一定次数的请求之后重新启动一次，这样来保证内 存的消耗是保证最小的。你可能还会在可能的时候用 my in perlfunc 限定变量的范围。

```
package Embed::Persistent;
#persistent.pl

use strict;
our %Cache;
use Symbol qw(delete_package);

sub valid_package_name {
    my($string) = @_;
    $string =~ s/([^A-Za-z0-9\/])/sprintf("_%2x",unpack("C",$1))/eg;
    # second pass only for words starting with a digit
    $string =~ s|/(\d)|sprintf("/_%2x",unpack("C",$1))|eg;

    # Dress it up as a real package name
    $string =~ s|/|::|g;
    return "Embed" . $string;
}

sub eval_file {
    my($filename, $delete) = @_;
    my $package = valid_package_name($filename);
    my $mtime = -M $filename;
    if(defined $Cache{$package}{mtime}
       &&
       $Cache{$package}{mtime} <= $mtime)
    {
       # we have compiled this subroutine already,
       # it has not been updated on disk, nothing left to do
       print STDERR "already compiled $package->handler\n";
    }
    else {
       local *FH;
```

```
        open FH, $filename or die "open '$filename' $!";
        local($/) = undef;
        my $sub = <FH>;
        close FH;

        #wrap the code into a subroutine inside our unique package
        my $eval = qq{package $package; sub handler { $sub; }};
        {
            # hide our variables within this block
            my($filename,$mtime,$package,$sub);
            eval $eval;
        }
        die $@ if $@;

        #cache it unless we're cleaning out each time
        $Cache{$package}{mtime} = $mtime unless $delete;
    }

    eval {$package->handler;};
    die $@ if $@;

    delete_package($package) if $delete;

    #take a look if you want
    #print Devel::Symdump->rnew($package)->as_string, $/;
}

1;

__END__

/* persistent.c */
#include <EXTERN.h>
#include <perl.h>

/* 1 = clean out filename's symbol table after each request, 0 = don't */
#ifndef DO_CLEAN
#define DO_CLEAN 0
#endif

#define BUFFER_SIZE 1024

static PerlInterpreter *my_perl = NULL;

int
main(int argc, char **argv, char **env)
{
    char *embedding[] = { "", "persistent.pl" };
    char *args[] = { "", DO_CLEAN, NULL };
    char filename[BUFFER_SIZE];
    int exitstatus = 0;
    STRLEN n_a;

    PERL_SYS_INIT3(&argc,&argv,&env);
    if((my_perl = perl_alloc()) == NULL) {
        fprintf(stderr, "no memory!");
        exit(1);
    }
    perl_construct(my_perl);
```

```
    exitstatus = perl_parse(my_perl, NULL, 2, embedding, NULL);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
    if(!exitstatus) {
      exitstatus = perl_run(my_perl);

      while(printf("Enter file name: ") &&
            fgets(filename, BUFFER_SIZE, stdin)) {

        filename[strlen(filename)-1] = '\0'; /* strip \n */
        /* call the subroutine, passing it the filename as an argument */
        args[0] = filename;
        call_argv("Embed::Persistent::eval_file",
                  G_DISCARD | G_EVAL, args);

        /* check $@ */
        if(SvTRUE(ERRSV))
           fprintf(stderr, "eval error: %s\n", SvPV(ERRSV,n_a));
      }
    }

    PL_perl_destruct_level = 0;
    perl_destruct(my_perl);
    perl_free(my_perl);
    PERL_SYS_TERM();
    exit(exitstatus);
}
```

Now compile:

```
% cc -o persistent persistent.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

Here's an example script file:

```
#test.pl
my $string = "hello";
foo($string);

sub foo {
   print "foo says: @_\n";
}
```

Now run:

```
% persistent
Enter file name: test.pl
foo says: hello
Enter file name: test.pl
already compiled Embed::test_2epl->handler
foo says: hello
Enter file name: ^C
```

执行 END 块

传统的 END 块在 perl_run 的结束时执行了。对于不调用 perl_run 的应用程序这会有一些问题。从 perl 5.7.2 开始，你可以指定 `PL_exit_flags |= PERL_EXIT_DESTRUCT_END` 来获得新特性。这也可以在 perl_parse 失败之后调用 END 块，`perl_destruct` 将返回退出值。

保持多个解释器的实例

一些罕见的应用程序在一次对话中需要创建多个解释器。可能要偶然释放解释器 对应的资源。

这个程序要确保要在下一个解释器就做这些事。默认情况下，当 perl 不用任何 选项构建时，全局变量 PL_perl_destruct_level 设置为 0。因为在一个程 序生存期中只创建一个解释器是不需要进行额外的清理。

将 PL_perl_destruct_level 设置为 1 可以使所有的都清除了：

```
while(1) {
    ...
    /* reset global variables here with PL_perl_destruct_level = 1 */
    PL_perl_destruct_level = 1;
    perl_construct(my_perl);
    ...
    /* clean and reset _everything_ during perl_destruct */
    PL_perl_destruct_level = 1;
    perl_destruct(my_perl);
    perl_free(my_perl);
    ...
    /* let's go do it again! */
}
```

当 perl_destruct() 调用时，这个解释器的语法解析树和符号表就被清除， 全局变量也被重新设置。因为 perl_construct 会将 PL_perl_destruct_level 重新设置为 0，所以要再一次设置 PL_perl_destruct_level。

现在假定我们同时有多个解释器运行。这是可以做到的，但是只有在你创建 perl 时使用配置选项 -Dusemultiplicity 或者 -Dusethreads -Duseithreads。缺省情况下，打开这些配置选项中的一个就把这个 per-interpreter 全局变量 PL_perl_destruct_level 设置为 1。所以清理 是自动的，并且解释器变量变正确的初始化。即使你不用同时运行多个解释器，而是要像前面的例子那样顺序运行，但还是建议你用 -Dusemultiplicity 选项来编译 perl。否则一些解释器的变量在连续运行过程中不会正确的初始化，你 的运行程序可能会崩溃。

如果你打算在不同线程中并发运行多个解释器时，使用 -Dusethreads -Duseithreads 而不是-Dusemultiplicity 可能更合适。因为这可以对解释 器支持链接到系统的线程库。

让我们来试一下：

```
#include <EXTERN.h> #include <perl.h>

/* we're going to embed two interpreters */
/* we're going to embed two interpreters */

#define SAY_HELLO "-e", "print qq(Hi, I'm $^X\n)"

int main(int argc, char **argv, char **env)
{
    PerlInterpreter *one_perl, *two_perl;
    char *one_args[] = { "one_perl", SAY_HELLO };
    char *two_args[] = { "two_perl", SAY_HELLO };

    PERL_SYS_INIT3(&argc,&argv,&env);
    one_perl = perl_alloc();
    two_perl = perl_alloc();
```

```
    PERL_SET_CONTEXT(one_perl);
    perl_construct(one_perl);
    PERL_SET_CONTEXT(two_perl);
    perl_construct(two_perl);

    PERL_SET_CONTEXT(one_perl);
    perl_parse(one_perl, NULL, 3, one_args, (char **)NULL);
    PERL_SET_CONTEXT(two_perl);
    perl_parse(two_perl, NULL, 3, two_args, (char **)NULL);

    PERL_SET_CONTEXT(one_perl);
    perl_run(one_perl);
    PERL_SET_CONTEXT(two_perl);
    perl_run(two_perl);

    PERL_SET_CONTEXT(one_perl);
    perl_destruct(one_perl);
    PERL_SET_CONTEXT(two_perl);
    perl_destruct(two_perl);

    PERL_SET_CONTEXT(one_perl);
    perl_free(one_perl);
    PERL_SET_CONTEXT(two_perl);
    perl_free(two_perl);
    PERL_SYS_TERM();
}
```

注意 PERL_SET_CONTEXT() 的调用。这对于全局状态的初始化中必须的（ These are necessary to initialize the global state that tracks which interpreter is the "current" one on the particular process or thread that may be running it. ）如果你有多个解释器并且同时对这些解释器交叉调用 perl API，就应该总是使用它。

当 interp 在一个不是创建它的线程（使用 perl_alloc() 或者更深奥 的 perl_clone() ）使用时，也应该调用 PERL_SET_CONTEXT(interp)。

像通常那样编译：

```
% cc -o multiplicity multiplicity.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

赶快运行吧：

```
% multiplicity
Hi, I'm one_perl
Hi, I'm two_perl
```

在你的 C 程序中使用 Perl 模块，这些模块本身也使用 C 库

如果你在使用上面的例子中试图嵌入一个脚本，这个脚本调用一个使用 C 或者 C++ 库的 Perl 模块（例如 Socket），可能会发生：

```
Can't load module Socket, dynamic loading not available in this perl.
  (You may need to build a new perl executable which either supports
  dynamic loading or has the Socket module statically linked into it.)
```

出什么错了？

你的解释器不知道怎样与这些扩展交流。一个小小的粘合代码将会起到作用。直 到现在你还是用 NULL 作为第二个参数调用 perl_parse()。

```
perl_parse(my_perl, NULL, argc, my_argv, NULL);
```

这是使用粘合代码的地方，它在 Perl 和链接的 C/C++ 函数创建起始的连接。让我们看看在 perlmain.c 中的一段看看 Perl 是怎样做的：

```
static void xs_init (pTHX);

EXTERN_C void boot_DynaLoader (pTHX_ CV* cv);
EXTERN_C void boot_Socket (pTHX_ CV* cv);

EXTERN_C void
xs_init(pTHX)
{
      char *file = __FILE__;
      /* DynaLoader is a special case */
      newXS("DynaLoader::boot_DynaLoader", boot_DynaLoader, file);
      newXS("Socket::bootstrap", boot_Socket, file);
}
```

对于每个要链接到你的 Perl 可执行程序的扩展（由你电脑的初始化配置决定或者当加入一个新的扩展），创建一个 Perl 函数整合扩展中的函数。通常这个函数叫 Module::boostrap()，当你使用 use Module 就调用了这个函数。In turn, this hooks into an XSUB, boot_Module, which creates a Perl counterpart for each of the extension's XSUBs. Don't worry about this part; leave that to the xsubpp and extension authors. If your extension is dynamically loaded, DynaLoader creates Module::bootstrap() for you on the fly. In fact, if you have a working DynaLoader then there is rarely any need to link in any other extensions statically.

一旦你有这段代码，把它加到 perl_parse() 的第二个参数中：

```
perl_parse(my_perl, xs_init, argc, my_argv, NULL);
```

然后编译：

```
% cc -o interp interp.c `perl -MExtUtils::Embed -e ccopts -e ldopts`

% interp
  use Socket;
  use SomeDynamicallyLoadedModule;

  print "Now I can use extensions!\n"'
```

ExtUtils::Embed 也能自动写 xs_init 粘合代码：

```
% perl -MExtUtils::Embed -e xsinit -- -o perlxsi.c
% cc -c perlxsi.c `perl -MExtUtils::Embed -e ccopts`
% cc -c interp.c  `perl -MExtUtils::Embed -e ccopts`
% cc -o interp perlxsi.o interp.o `perl -MExtUtils::Embed -e ldopts`
```

详细内容参考 perlxs、perlguts 和 perlapi。

3 在 Win32 嵌入 Perl

一般，这里显示的所有代码在 Windows 下不用任何修改就能工作。

尽管这样，这里有一些命令行例子的警告。对于初学者，在 Win32 本身的命令 行中是不能使用反引号的。在 CPAN 的 ExtUtils::Embed 中有一个称为 genmake 脚本。这可以从单个的 C 源文件中创建一个简单的 makefile。可 以这样使用：

```
C:\ExtUtils-Embed\eg> perl genmake interp.c
C:\ExtUtils-Embed\eg> nmake
C:\ExtUtils-Embed\eg> interp -e "print qq{I'm embedded in Win32!\n}"
```

你可能想在 Microsoft Developer Studio 中使用更稳健的环境（ You may wish to use a more robust environment such as the Microsoft Developer Studio. ）。在这种情况下中，用这个来产生 perlxsi.c：

```
perl -MExtUtils::Embed -e xsinit
```

创建一个新的工程，然后 Insert -> Files 到工程中：perlxsi.c，perl.lib， 和你自己的源文件，例如 interp.c。一般你可以在 C:\perl\lib\CORE 中找 到 perl.lib。如果没有的话，你可以用 `perl -V:archlib` 中找到 CORE 目录。studio 还要知道在哪里找到 Perl 的 include 文件。这个路径可以通过 Tools -> Options -> Directories 菜单来加入。最后，选择 Build -> Build interp.exe，这样就好了。

4 隐藏 Perl_

在编译标签中加入 -DPERL_NO_SHORT_NAMES，你就可以隐藏 Perl 公共接口的简短 形式。这意味着你不能这样写：

```
warn("%d bottles of beer on the wall", bottlecount);
```

你必须写明确完全的形式：

```
Perl_warn(aTHX_ "%d bottles of beer on the wall", bottlecount);
```

（参考 Background and PERL_IMPLICIT_CONTEXT for the explanation of the `aTHX_`. in perlguts）隐藏简短的形式对于避免和其它软件包的冲突（C 预处理 或者其它）。（Perl 用简短名字定义了 2400 API，所以很有可能发生冲突。）

5 MORAL

有时可以在 C 中写出 运行更快的代码（write faster code），但是你总是可以在 Perl 中更快地写出代码（write code faster）。因为你可以相互使用对方，只 要你需要可以结合起来。

6 AUTHOR

Jon Orwant <orwant@media.mit.edu> and Doug MacEachern <dougm@covalent.net>, with small contributions from Tim Bunce, Tom Christiansen, Guy Decoux, Hallvard Furuseth, Dov Grobgeld, and Ilya Zakharevich.

Doug MacEachern has an article on embedding in Volume 1, Issue 4 of The Perl Journal (http://www.tpj.com/). Doug is also the developer of the most widely-used Perl embedding: the mod_perl

system (perl.apache.org), which embeds Perl in the Apache web server. Oracle, Binary Evolution, ActiveState, and Ben Sugars's nsapi_perl have used this model for Oracle, Netscape and Internet Information Server Perl plugins.

July 22, 1998

7 COPYRIGHT

8 TRANSLATORS

YE Wenbin