

第十章 包

↓ 第十章 包

↓ 10.1 符号表

↓ 10.2 自动装载

在本章里，我们开始有好玩的东西了，因为我们要开始讲有关软件设计的东西。如果我们要聊一些好的软件设计，那么我们就必须先侃侃懒惰，急躁，和傲慢，这几样好的软件设计需要的基本要素。

我们经常落到使用拷贝和粘贴（**ICP-I Copy & Paste**）的陷阱里，而如果一个循环或者一个子过程就足够了，（注：这是伪懒惰的一种形式）那么这时候我们实际上应该定义一个更高层次的抽象。但是，有些家伙却走向另外一个极端，定义了一层又一层的高层抽象，而这个时候他们应该用拷贝和粘贴。（注：这是伪傲慢的一种形式。）不过，通常来讲，我们大多数人都应该考虑使用更多的抽象。

落在中间的是那些对抽象深度有平衡观念的人，不过他们马上就开始写它们自己的抽象层，而这个时候它们应该重用现有的代码。（注：你也许已经猜到了——这是为急躁。不过，如果你准备推倒重来，那么你至少应该发明一种更好的东西。）

如果你准备做任何这样的事情，那么你都应该坐下来想想，怎样做才能从长远来看对你和你的邻居最有帮助。如果你准备把你的创造力引擎作用到一小块代码里，那么为什么不把这个你还要居住的这个世界变得更美好一些呢？（即使你的目的只是为了程序的成功，那你就确信你的程序能够符合社会生态学的要求。）

朝着生态编程的第一步是：不要在公园里乱丢垃圾（译注：否则砸到小朋友...或者花花草草...：）。当你写一段代码的时候，考虑一下给这些代码自己的名字空间，这样你的变量和函数就不会把别人的变量和函数搞砸了，反之亦然。名字空间有点象你的家，你的家里想怎么乱都行，只要你保持你的外部界面对其他公民来说是适度文明的就可以了。在 **Perl** 里，一个名字空间叫一个包。包提供了基本的制作块，在它上面构造更高级的概念，比如模块和类等。

和“家”的说法相似，“包”的说法也有一些模糊。包独立于文件。你可以在一个文件里有许多包，或者是一个包跨越多个文件，就好像你的家可以是在一座大楼里面的小小的顶楼（如果你是一个穷困潦倒的艺术家），或者你的家也可以由好多建筑构成（比如你的名字叫伊丽莎白女王）。但家的常见大小就是一座建筑，而包通常也是一个文件大，**Perl** 给那些想把一个包放到一个文件里的人们提供了一些特殊的帮助，条件只是你愿意给文件和包相同的名字并且使用一个 **.pm** 的扩展名，**pm** 是“**perl module**”的缩写。模块（**module**）是 **Perl** 里重复使用的最基本的模块。实际上，你使用模块的方法是 **use** 命令，它是一个编译器指示命令，可以控制从一个模块里输入子过程和变量。到目前为止你看到的每一个 **use** 的例子都是模块复用的例子。

如果其他人认为你的模块有用，那么你应该把它们放到 **CPAN**。**Perl** 的繁荣是和程序员愿意和整个社区分享他们劳动的果实分不开的。自然，**CPAN** 也是你可以找到那些其他人已经非常仔细地上载上去给别人用的模块的地方。参阅第二十二章，**CPAN**，以及 www.cpan.org 获取详细信息。

过去 25 年左右的时间里，设计计算机语言的趋势是强调某种偏执。你必须编制每一个模块，就好像它是一个围城的阶段一样。显然有些封建领地式的文化可以使用这样的方法，但并不是所有文化都喜欢这样。比如，在 **Perl** 文化里，人们让你离它们的房子远一点是因为他们没有邀请你，而不是因为窗户上有窗栅。（注：不过，如果需要，**Perl** 提供了一些窗栅。参阅第二十三章，安全，里的“处理不安全数据”。）

这本书不是讲面向对象的方法论的，并且我们在这里也不想把你推到面向对象的狂热中去，就算你想进去我们的态度也这样。关于这方面的东西已经有大量书籍了。**Perl** 对面向对象设计的原则和 **Perl** 对其他东西的原则是一样的：在面向对象的设计方法有意义的地方就用它，而在没有意义的地方就绕

开它。你的选择。

在 OO 的说法中，每个对象都属于一个叫做类的组。在 Perl 里，类和包以及模块之间的关系是如此地密切，以至于许多新手经常认为它们是可以互换的。典型的类是用一个定义了与该类同名的包名字的模块实现的。我们将在随后的几章里解释这些东西。

当你 **use** 一个模块的时候，你是从软件复用中直接受益。如果你用了类，那么如果一个类通过继承使用了另外一个类，那么你是间接地从软件复用中受益。而且用了类，你就获得了更多的一些东西：一个通往另外一个名字空间的干净的接口。在类里面，所有东西都是间接地访问的，把这个类和外部的世界隔离开。

就象我们在第八章，引用，里提到的一样，在 Perl 里的面向对象的编程是通过引用来实现的，这些引用的引用物知道它们属于哪些类。实际上，如果你知道引用，那么你就知道几乎所有有关对象的困难。剩下的就是“放在你的手指下面”，就象画家会说的那样。当然，你需要做一些练习。

你的基本练习之一就学习如何保护不同的代码片段，避免被其他人的变量不小心篡改。每段代码都属于一个特定的包，这个包决定它里面有哪些变量和代码可以使用。当 Perl 碰到一段代码的时候，这段代码就被编译成我们叫做当前包的东西。最初的当前包叫做“**main**”，不过你可以用 **package** 声明在任何时候把当前的包切换成另外一个。当前包决定使用哪个符号表查找你的变量，子过程，I/O 句柄和格式等。

任何没有和 **my** 关联在一起的变量声明都是和一个包相关联的——甚至是一些看起来无所不在的变量，比如 **\$_** 和 **%SIG**。实际上，在 Perl 里实际上没有全局变量这样的东西。（特殊的标识符，比如 **_** 和 **SIG**，只是看上去象全局变量，因为它们缺省时属于 **main** 包，而不是当前包。）

package 声明的范围从声明本身开始直到闭合范围的结束（块，文件，或者 **eval**——以先到为准）或者直到其他同层次的 **package** 声明，它会取代前面的那个。（这是个常见的实践。）

所有随后的标识符（包括那些用 **our** 声明的，但是不包括那些用 **my** 或者那些用其他包名字修饰的变量。）都将放到属于当前包的符号表中。（用 **my** 声明的变量独立于包；它们总是属于包围它们的闭合范围，而且也只属于这个范围，不管有什么包声明。）

通常，一个 **package** 声明如果是一个文件的第一个语句的话就意味着它将被 **require** 或者 **use** 包含。但这只是习惯，你可以在任何可以放一条语句的地方放一个 **package** 声明。你甚至可以把它放在一个块的结尾，这个时候它将没有任何作用。你可以在多于一个的地方切换到一个包里面；包声明只是为该块剩余的部分选择将要使用的符号表。（这也是一个包实现跨越多个文件的方法。）

你可以引用其他包里的标识符（注：我们说的标识符的意思是用做符号表键字的东西，可以用来访问标量变量，数组变量，子过程，文件或者目录句柄，以及格式等。从语法上来说，标签（**Label**）也是标识符，但是它们不会放到特定的符号表里；相反，它们直接附着在你的程序里的语句上面。标签不能用包名字修饰。），方法是用包名字和双冒号做前缀（“修饰”）：**\$Package::Variable**。如果包名字是空，那么就假设为 **main** 包。也就是说，**\$::sail** 等于 **\$main::sail**。（注：为了把另外一点容易混淆的概念理清楚，在变量名 **\$main::sail** 里，我们对 **main** 和 **sail** 使用术语“标识符”，但不把 **main::sail** 称做标识符。我们叫它一个变量名。因为标识符不能包含冒号。）

老的包分隔符还是一个单引号，因此在老的 Perl 程序里你会看到象 **\$main'sail** 和 **\$somepack'horse** 这样的变量。不过，双冒号是现在的优选的分隔符，部分原因是因为它更具有可读性，另一部分原因是它更容易被 **emacs** 的宏读取。而且这样表示也令 C++ 程序员觉得明白自己在做什么——相比之下，用单引号的时候就能让 **Ada** 的程序员知道自己在做什么。因为出于向下兼容的考虑，Perl 仍然支持老风格的语法，所以如果你试图使用象 **"This is \$owner's house"** 这样的字符串，那么你实际上就是在访问 **\$owner::s**；也就是说，在包 **owner** 里的 **\$s** 变量，这可能并不是你想要的。你可以用花括弧来消除歧义，就象 **"This is \${owner}'s house"**。

双冒号可以用于把包名字里的标识符链接起来：**\$Red::Blue::Var**。这就意味着 **\$var** 属于

`Red::Blue` 包。`Red::Blue` 包和任何可能存在的 `Red` 或者 `Blue` 包都没有关系。也就是说，在 `Red::Blue` 和 `Red` 或者 `Blue` 之间的关系可能对那些书写或使用这个程序的人有意义，但是它对于 `Perl` 来说没有任何意义。（当然，在当前的实现里，符号表 `Red::Blue` 碰巧存储在 `Red` 符号表里。但是 `Perl` 语言对此没有做任何直接的利用。）

由于这个原因，每个 `package` 声明都必须声明完整的包名字。任何包名字都没有做任何隐含的“前缀”的假设，甚至（看起来象）在一些其他包声明的范围里声明的那样也如此。

只有标识符（以字母或者一个下划线开头的名字）才存储在包的符号表里。所有其他符号都保存在 `main` 包里，包括所有非字母变量，比如 `$!`，`$?`，和 `$_`。另外，在没有加以修饰的时候，标识符 `STDIN`，`STDOUT`，`STDERR`，`ARGV`，`ARGVOUT`，`ENV`，`INC`，和 `SIG` 都强制在包 `main` 里，即使你是用做其他目的，而不是用做它们的内建功能也如此。不要把你的包命名为 `m`，`s`，`tr`，`q`，`qq`，`qr`，`qw`，或者 `qx`，除非你想自找一大堆麻烦。比如，你不能拿修饰过的标识符形式做文件句柄，因为它将被解释成一个模式匹配，一个替换，或者一个转换。

很久以前，用下划线开头的变量被强制到 `main` 包里，但是我们发现让包作者使用前导的下划线作为半私有的标识符标记更有用，这样它们就可以表示为只被该包内部使用。（真正私有的变量可以声明为文件范围的词汇，但是只有在包和模块之间有一对一的关系的时候，这样的做法才比较有效，虽然这样的一对一比较普遍，但并不是必须的。）

`%SIG` 散列（用于捕获信号；参阅第十六章，进程间通讯）也是特殊的。如果你把一个信号句柄定义为字串，那么 `Perl` 就假设它引用一个 `main` 包里的子过程，除非明确地使用了其他包名字。如果你想声明一个特定的包，那么你要使用一个信号句柄的全称，或者完全避免字串的使用：方法是改为赋予一个类型团或者函数引用：

```
$SIG{QUIT} = "Pkg::quit_catcher";    # 句柄全称
$SIG{QUIT} = "quit_catcher";        # 隐含的"main::quit_catcher"
$SIG{QUIT} = *quit_catcher;          # 强制为当前包的子过程
$SIG{QUIT} = \&quit_catcher;         # 强制为当前包的子过程
$SIG{QUIT} = sub { print "Caught SIGQUIT\n" }; # 匿名子过程
```

“当前包”的概念既是编译时的概念也是运行时的概念。大多数变量名查找发生在编译时，但是运行时查找发生在符号引用解引用的时候，以及 `eval` 分析新的代码的时候。实际上，在你 `eval` 一个字串的时候，`Perl` 知道该 `eval` 是在哪个包里调用的并且在计算该字串的时候把那个包名字传播到 `eval` 里面。（当然，你总是可以在 `eval` 里面切换到另外一个包，因为一个 `eval` 字串是当作一个块对待的，就象一个用 `do`，`require`，或者 `use` 装载的块一样。）

另外，如果一个 `eval` 想找出它在哪个包里，那么特殊的符号 **`PACKAGE`** 包含当前包名字。因为你可以把它当作一个字串看待，所以你可以把它用做一个符号引用来访问一个包变量。但如果你在这么做，那么你很有机会把该变量用 `our` 声明，作为一个词法变量来访问。

10.1 符号表

一个包的内容总体在一起称做符号表。符号表都存储在一个散列里，这个散列的名字和该包的名字相同，但是后面附加了两个冒号。因此 `main` 符号表的名字是 `%main::`。因为 `main` 碰巧也是缺省的包，`Perl` 把 `%::` 当作 `%main::` 的缩写。

类似，`Red::Blue` 包的符号表名字是 `%Red::Blue::`。同时 `main` 符号表还包含所有其他顶层的符号表，包括它本身。因此 `%Red::Blue::` 同时也是 `%main::Red::Blue::`。

当我们说到一个符号表“包含”其他的符号表的时候，我们的意思是它包含一个指向其他符号表的引用。因为 `main` 是顶层包，它包含一个指向自己的引用，结果是 `%main::` 和 `%main::main::`，和 `%main::main::main::`，等等是一样的，直到无穷。如果你写的代码包括遍历所有的符号表，

那么一定要注意检查这个特殊的情况。

在符号表的散列里，每一对键字/数值对都把一个变量名字和它的数值匹配起来。键字是符号标识符，而数值则是对应的类型团。因此如果你使用 ***NAME** 表示法，那么你实际上只在访问散列里的一个数值，该数值保存当前包的符号表。实际上，下面的东西有（几乎）一样的效果：

```
*sym = *main::variable;
*sym = $main::{"variable"};
```

第一种形式更高效是因为 **main** 符号表是在编译时被访问的。而且它还会在该名字的类型团不存在的时候创建一个新的，但是第二种则不会。

因为包是散列，因此你可以找出该包的键字然后获取所有包中的变量。因此该散列的数值都是类型团，你可以用好几种方法解引用。比如：

```
foreach $symname (sort keys %main::) {
    local *sym = $main::{$symname};
    print "\$$symname is defined\n" if defined $sym;
    print "@$symname is nonnull\n" if @sym;
    print "%$symname is nonnull\n" if %sym;
}
```

因为所有包都可以（直接或间接地）通过 **main** 包访问，因此你可以在你的程序里写出访问每一个包变量的 **Perl** 代码。当你用 **v** 命令要求 **Perl** 调试器倾倒所有你的变量的时候，它干的事情就是这个。请注意，如果你做这些事情，那么你将看不到用 **my** 声明的变量，因为它们都是独立于包的，不过你看得用 **our** 声明的变量。参阅第二十章，**Perl** 调试器。

早些时候我们说过除了在 **main** 里，其他的包里只能存储标识符。我们在这里撒了个小慌：你可以在一个符号表散列里使用任何你需要的字串作为键字——只不过如果你企图直接使用一个非标识符的时候它就不是有效的 **Perl**：

```
$!@#$$      = 0;          # 错，语法错
#{'!@#$$'}  = 1;          # 正确，用的是未修饰的

${'main::!@#$$'} = 2;      # 可以在字串里修饰。
print ${ $main::{'!@#$$'}} # 正确，打印 2
```

给一个匿名类型团赋值执行一个别名操作；也就是，

```
*dick = *richard;
```

导致所有可以通过标识符 **richard** 访问的变量，子过程，格式，文件和目录句柄也可以通过符号 **dick** 访问。如果你只需要给一个特定的变量或者子过程取别名，那么使用一个引用：

```
*dick = \$richard;
```

这样就令 **\$richard** 和 **\$dick** 成为同样的变量，但是 **@richard** 和 **@dick** 则剩下来是独立的数组。很高明，是吗？

这也是 **Exporter** 在从一个包向另外一个包输入符号的时候采用的方法。比如：

```
*SomePack::dick = \&OtherPack::richard;
```

从包 **OtherPack**² 输入 **&richard** 函数到 **SomePack**²，让它可以当作 **&dick** 函数用。

（**Exporter** 模块在下一章描述。）如果你用一个 **local** 放在赋值前面，那么，该别名将只持续到当前动态范围结束。

这种机制可以用于从一个子过程中检索一个引用，令该引用可以用做一个合适的数据类型：

```
*units = populate();      # 把 \%newhash 赋予类型团
print $units{kg};         # 打印 70；而不用解引用！

sub populate {
    my %newhash = (km => 10, kg => 70);
    return \%newhash;
}
```

类似，您还可以把一个引用传递到一个引用传递到一个子过程里并且不加解引用地使用它：

```
%units = (miles => 6, stones => 11);
fillerup( \%units );      # 传递进一个引用
print $units{quarts};     # 打印 4

sub fillerup {
    local *hashsym = shift; # 把 \%units 赋予该类型团
    $hashsym{quarts} = 4;   # 影响 \%units；不需要解引用！
}
```

上面这些都是廉价传递引用的巧妙方法，用在你不想明确地对它们进行解引用的时候。请注意上面两种技巧都是只能对包变量管用；如果我们用 **my** 声明了 **%units** 那么它们不能运行。

另外一个符号表的用法是制作“常量”标量：

```
*PI = \3.14159265358979;
```

现在你不能更改 **\$PI**，不管怎么说这样做可能是好事情。它和常量子过程不一样，常量子过程在编译时优化。常量子过程是一个原型定义为不接受参数并且返回一个常量表达式的子过程；参阅第六章，子过程，的“内联常量函数”获取细节。**use constant** 用法（参阅第三十一章，用法模块）是一个方便的缩写：

```
use constant PI => 3.14159;
```

在这个钩子下面，它使用 ***PI** 的子过程插槽，而不是前面用的标量插槽。它等效于更紧凑（不过易读性稍微差些）：

```
*PI = sub () {3.14159};
```

不过，这是一个很值得知道的俗语——把一个 **sub {}** 赋予一个类型团是在运行时给匿名子过程赋予一个名字的方法。

把一个类型团引用赋予另外一个类型团（***sym = *oldvar**）和赋予整个类型团是一样的。并且如果你把类型团设置为一个简单的字串，那么你就获得了该字串命名的整个类型团，因为 **Perl** 在当前符号表中寻找该字串。下面的东西互相都是一样的，不过头两个在编译时计算符号表记录，而后面两个是在运行时：

```
*sym = *oldvar;
*sym = \%oldvar;      # 自动解引用
*sym = *{"oldvar"};   # 明确的符号表查找
*sym = "oldvar";      # 隐含地符号表查找
```

当你执行任意下列的赋值的时候，你实际上只是替换了类型团里的一个引用：

```
*sym = \$frodo;
*sym = \@sam;
```



```
*sym = \%merry;
*sym = \&pippin;
```

如果你从另外一个角度来考虑问题，类型团本身可以看作一种散列，它里面有不同类型的变量记录。在这种情况下，键字是固定的，因为一个类型团正好可以包含一个标量，一个散列，等等。但是你可以取出独立的引用，象这样：

```
*pkg::sym{SCALAR}      # 和 \%pkg::sym 一样
*pkg::sym{ARRAY}       # 和 \@pkg::sym 一样
*pkg::sym{HASH}        # 和 \%pkg::sym 一样
*pkg::sym{CODE}        # 和 \&pkg::sym 一样
*pkg::sym{GLOB}        # 和 \*pkg::sym 一样
*pkg::sym{IO}          # 内部的文件/目录句柄，没有直接的等价物
*pkg::sym{NAME}        # “sym”（不是引用）
*pkg::sym{PACKAGE}     # “pkg”（不是引用）
```

你可以通过说 `*foo{PACKAGE}` 和 `*foo{NAME}` 找出 `*foo` 符号表记录来自哪个名字和包。这个功能对那些传递类型团做参数的子过程里很有用：

```
sub identify_typeglob {
    my $glob = shift;
    print 'You gave me ', *{$glob}{PACKAGE}, '::~', *{$glob}{NAME}, "\n";
}

identify_typeglob(*foo);
identify_typeglob(*bar::glarch);
```

它打印出：

```
You gave me main::foo
You gave me bar::glarch
```

`*foo{THING}` 表示法可以用于获取指向 `*foo` 的独立元素的引用。参阅第八章的“符号表引用”一节获取细节。

这种语法主要用于获取内部文件句柄或者目录句柄引用，因为其他内部引用已经都可以用其他方法访问。（老的 `*foo{FILEHANDLE}` 形式仍然受支持，表示 `*foo{IO}`，但是不要让这个名字把你涮了，它可不能把文件句柄和目录句柄区别开。）但是我们认为应该概括它，因为它看起来相当漂亮。当然，你可能根本不用记住这些东西，除非你想再写一个 Perl 调试器。

10.2 自动装载

通常，你不能调用一个没有定义的子过程。不过，如果在未定义的子过程的包（如果是在对象方法的情况下，在任何该对象的基类的包里）里有一个子过程叫做 **AUTOLOAD**，那么就会调用 **AUTOLOAD** 子过程，同时还传递给它原本传递给最初子过程的同样参数。你可以定义 **AUTOLOAD** 子过程返回象普通子过程那样的数值，或者你可以让它定义还不存在的子过程然后再调用它，就好象该子过程一直存在一样。

最初的子过程的全称名会神奇地出现在包全局变量 `$AUTOLOAD` 里，该包和 **AUTOLOAD** 所在的包是同一个包。下面是一个简单的例子，它会礼貌地警告你关于未定义的子过程调用，而不是退出：

```
sub AUTOLOAD {
    our $AUTOLOAD;
    warn "Attempt to call $AUTOLOAD failed.\n";
}
```

```
blarg(10);          # 我们的 $AUTOLOAD 将会设置为 main::blarg
print "Still alive!\n";
```

或者你可以代表该未定义的子过程返回一个数值：

```
sub AUTOLOAD {
    our $AUTOLOAD;
    return "I see $AUTOLOAD(@_) \n":
}

print blarg(20);      # 打印: I see main::blarg(20)
```

你的 **AUTOLOAD** 子过程可以用 **eval** 或者 **require** 为该未定义的子过程装载一个定义，或者是用我们前面讨论过的类型团赋值的技巧，然后使用特殊形式的 **goto** 执行该子过程，这种 **goto** 可以不留痕迹地抹去 **AUTOLOAD** 过程的堆栈帧。下面我们通过给类型团赋予一个闭合来定义该子过程：

```
sub AUTOLOAD {
    my $name = our $AUTOLOAD;
    *$AUTOLOAD = sub { print "I see $name(@_) \n"};
    goto &$AUTOLOAD;      # 重起这个新过程。
}

blarg(30);             # 打印: I see main::blarg(30)
blarg(40);             # 打印: I see main::blarg(40)
blarg(50);             # 打印: I see main::blarg(50)
```

标准的 [AutoSplit²](#) 模块被模块作者用于把他们的模块分裂成独立的文件（用以 **.al** 结尾的文件），每个保存一个过程。该文件放在你的系统的 Perl 库的 **auto/** 目录里，在那之后该文件可以根据标准的 [AutoLoader²](#) 模块的需要自动装载。

一种类似的方法被 [SelfLoader²](#) 模块使用，只不过它从该文件自己的 **DATA** 区自动装载函数，从某种角度来看，它的效率要差一些，但是从其他角度来看，它的效率又比较高。用 [AutoLoader²](#) 和 [SelfLoader²](#) 自动装载 Perl 函数是对通过 [DynaLoader²](#) 动态装载编译好的 **C** 函数的模拟，只不过自动装载是以函数调用的粒度进行实现的，而动态装载是以整个模块的粒度进行装载的，并且通常会一次链接进入若干个 **C** 或 **C++** 函数。（请注意许多 Perl 程序员不用 [AutoSplit²](#)，[AutoLoader](#)，[SelfLoader](#)，或者 [DynaLoader²](#) 模块过得也很好。你只需要知道它们的存在，以防哪天你不用它还真解决不了问题。）

我们可以在把 **AUTOLOAD** 过程当作其他接口的封装器中获取许多乐趣。比如，让我们假设任何没有定义的函数应该就是哪它的参数调用 **system**。你要做的就是：

```
sub AUTOLOAD {
    my $program = our $AUTOLOAD;
    $program =~ s/.*::://;      # 截去包名字
    system($program, @_);
}
```

（恭喜，你刚刚实现了和 Perl 一起发布的 **Shell** 模块的一种冗余的形式。）你可以象下面这样调用你的自动装载器（在 **Unix** 里）：

```
date();
who('am', 'i');
is('-l');
echo("Abadugabuadaredd...");
```

实际上，如果你预先按照这种方法声明你想要调用的函数，那么你就可以认为它们是内建的函数并且

在调用的时候忽略圆括弧:

```
sub date (;$$);          # 允许零到两个参数。
sub who (;$$$$);         # 允许零到四个参数
sub ls;                  # 允许任意数量的参数
sub echo ($@);           # 允许至少一个参数

date;
who "am", "i";
ls "-l";
echo "That's all, folks!";
```

Revision: r1.2 - 27 Aug 2005 - 10:00 - [TingYu](#)

[Perl](#) > [PerlProgramming3](#) > Packages

版权 © 1999-2006 归这里所有作者. [PostgreSQL](#) 的中文文档版权归何伟平所有.
向为这里贡献想法,文章的人致敬 [PostgreSQL](#) 中文网
[反馈意见](#)