

## 第十一章 模块

- ↓ 第十一章 模块
  - ↓ 11.1 使用模块
  - ↓ 11.2 创建模块
    - ↓ 11.2.1 模块私有和输出器
      - ↓ 11.2.1.1 不用 `Exporter` 的输入方法进行输出
      - ↓ 11.2.1.2 版本检查
      - ↓ 11.2.1.3 管理未知符号
      - ↓ 11.2.1.4 标签绑定工具函数
  - ↓ 11.3 覆盖内建的函数

模块是 Perl 里重复使用的基本单元。在它的外皮下面，它只不过是定义在一个同名文件（以 `.pm` 结尾）里面的包。本章里，我们将探究如何使用别人的模块以及创建你自己的模块。

Perl 是和一大堆模块捆绑在一起安装的，你可以在你用的 Perl 版本的 `lib` 目录里找到它们。那里面的许多模块将在第三十二章，标准模块，和第三十一章，用法模块里描述。所有标准模块都还有大量的在线文档，很可能比这本书更新。如果你的 `man` 命令里没有更丰富的东西，那么请试着使用 `perldoc` 命令。

综合 Perl 库网络（CPAN）是包含全世界的 Perl 社区所贡献的 Perl 模块的仓库，我们将在第二十二章，CPAN 里介绍它。同样也请参阅 <http://www.cpan.org>。

### 11.1 使用模块

模块有两种风格：传统的和面向对象的。传统模块为调用者的输入和使用定义了子过程和变量。面向对象的模块的运转类似类声明并且是通过方法调用来访问的，在第十二章，对象，里描述。有些模块有上面两种类型的东西。

Perl 模块通常用下面的语句包含入你的程序：

```
use MODULE LIST;
```

或者只是：

```
use MODULE;
```

`MODULE` 必须是一个命名模块的包和文件的标识符。（这里描述的语法只是建议性的；`use` 语句的详细描述在第二十九章，函数，里。）

`use` 语句在编译的时候对 `MODULE` 进行一次预装载，然后把你需要的符号输入进来，这样剩下的编译过程就可以使用这些符号了。如果你没提供你想要的符号的 `LIST`（列表），那么就使用在模块的内部 `@EXPORT` 数组里命名的符号——假设你在用 `Exporter` 模块，有关 `Exporter` 的内容在本章稍后的“模块私有和输出器”里介绍。（如果你没有提供 `LIST`，那么所有你的符号都必须在模块的 `@EXPORT` 或者 `@EXPORT_OK` 数组里提及，否则否则就会发生一个错误。）

因为模块使用 `Exporter` 把符号输入到当前包里，所以你可以不加包限制词地使用来自该模块的符号：

```
use Fred;          # 如果Fred.pm 有 @EXPORT = qw(flintstone)
flintstone();      # ...这里调用 Fred::flintstone()。
```

所有 Perl 的模块文件都有 `.pm` 的扩展名。`use` 和 `require` 都做这种假定（和引起），因此你不用说 `"MODULE.pm"`。只使用描述符可以帮助我们吧新模块和老版本的 Perl 中使用用的 `.pl` 和 `.ph`

库区别开。它还把 **MODULE** 当作一个正式模块名，这样可以在某些有歧义的场所帮助分析器。在模块名字中的任何双冒号都被解释成你的系统的目录分隔符，因此如果你的模块的名字是

**Red::Blue::Green**，Perl 就会把它看作 **Red/Blue/Green.pm**。

Perl 将在 **@INC** 数组里面列出的每一个目录里面查找模块。因为 **use** 在编译的时候装载模块，所以任何对 **@INC** 的修改都需要在编译时发生。你可以使用第三十一章里描述的 **lib** 用法或者一个

**BEGIN** 块来实现这个目的。一旦包含了一个模块，那么就会向 **%INC** 哈希表里增加一个键字/数值对。这里的键字将是模块的文件名（在我们的例子中是 **Red/Blue/Green.pm**）而数值将是全路径名。如果是在一个 **windows** 系统上合理安装的模块，这个路径可能是

**C:/perl/site/lib/Red/Blue/Green.pm**。

除非模块起用法的作用，否则它们的名字应该首字母大写。用法是有效的编译器指示器（给编译器的提示），因此我们把小写的用法名字留给将来使用。

当你 **use** 一个模块的时候，在模块里的所有代码都得到执行，就好象 **require** 里的通常情况一样。如果你不在乎模块是在编译的时候还是在运行的时候引入的，你可以只说：

```
require MODULE;
```

不过，通常我们更愿意用 **use** 而不是 **require**，因为它在编译的时候就查找模块，因此你可以更早知道有没有问题。

下面的两个语句做几乎完全一样的事情：

```
require MODULE;
require "MODULE.pm";
```

不过，它们在两个方面不太一样。在第一个语句里，**require** 把模块名字里的任何双冒号转换成你的系统的目录分隔符，就象 **use** 那样。第二种情况不做转换，强制你在文本上声明你的模块的路径名，这样移植性比较差。另外一个区别是第一个 **require** 告诉编译器说，带有关于 **"MODULE"** 的间接对象符号的表达式（比如 **\$ob = purge MODULE**）都是模块调用，而不是函数调用。（如果你在自己的模块里有冲突的 **purge** 定义，那么这里就有区别了。）

因为 **use** 声明和相关的 **no** 声明都隐含有一个 **BEGIN** 块，编译器就会一看到这个声明就装载这个模块（并且运行里面的任何可执行初始化代码），然后才编译剩下的文件。这就是用法如何改变编译器的性质的方法，以及为什么模块可以声明一些子过程，这些子过程可以作为列表操作符用于剩下的编译过程。如果你用 **require** 代替 **use**，这些事情就不会发生。使用 **require** 的唯一原因就是你有两个模块，这两个模块都需要来自对方的函数。（我们不知道这是不是个好理由。）

Perl 模块总是装载一个 **.pm** 文件，但是这个文件随后可以装载相关的文件，比如动态链接的 **C** 或 **C++** 库或者自动装载的 Perl 子过程定义。如果是这样，那么附加的东西对模块用户而言是完全透明的。装载（或者安排自动）任何附加的函数或功能的责任在 **.pm** 文件。正巧是 **POSIX** 模块动态装载和自动装载两种方法都要用，不过用户可以只说：

```
use POSIX;
```

就可以获取所有输出了的函数和变量。

## 11.2 创建模块

---

我们前面说过，一个模块可以有两个方法把它的接口提供给你的程序使用：把符号输出或者允许方法调用。我们在这里先给你演示一个第一种风格的例子；第二种风格用于面向对象的模块，我们将在下一章里描述。（面向对象的模块应该不输出任何东西，因为方法最重要的概念就是 Perl 以该对象的类型为基础自动帮你找到方法自身。）

构造一个叫 **Bestiary** 的模块，创建一个看着象下面这样的叫 **Bestiary.pm** 的文件：

```
package    Bestiary;
require    Exporter;

our @ISA   =qw(Exporter);
our @EXPORT =qw(camel);      # 缺省输出的符号
our @EXPORT_OK =qw($weight); # 按要求输出的符号
our $VERSION = 1.00;         # 版本号

### 在这里包含你的变量和函数

sub camel { print "One-hump dromedary" }

$weight = 1024;

1;
```

一个程序现在可以说 `use Bestiary` 就能访问 `camel` 函数（但是不能访问 `$weight` 变量），或者 `use Bestiary qw(camel, $weight)` 可以访问函数和变量。

你还可以创建动态装载 C 写的代码的模块。参阅第二十一章，内部和外部，获取细节。

### 11.2.1 模块私有和输出器

Perl 不会自动在它的模块的私有/公有边界上进行检查——和 C++，JAVA 和 Ada 这样的语言不同，Perl 不会被强加的私有性质搞糊涂。Perl 希望你呆在它的起居室外面是因为你没有收到邀请，而不是因为你拿着一把手枪。

Perl 模块和其用户之间有一种约定，有一部分是常见规则而另外一部分是单写的。常见规则部分约定是说禁止一个模块修改任何没有允许它修改的名字空间。为模块单写的约定（也就是文档）可以有其他约束。不过，如果你读过约定以后，我们就假设你知道自己说 `use ReadlineTheWorld2` 的时候就是在重定义世界，并且你愿意面对其后果。重定义世界的最常用的方法是使用 **Exporter** 模块。我们稍后在本章中就能看到，你甚至可以用这个模块重定义内建的东西。

当你 `use` 一个模块，通常是这个模块提供了你的程序可以使用的几个函数或者变量，或者更准确地说，为你的程序的当前包提供了函数和变量。这种从模块输出符号（并且把它们输入到你的程序里）的动作有时候被称为污染你的名字空间。大多数模块使用 **Exporter** 来做这些事情；这就是为什么在接近顶端的地方说这些东西：

```
require Exporter;
our @ISA = ("Exporter");
```

这两行令该模块从 **Exporter** 类中继承下来。我们在下一章讲继承，但在这里你要知道的所有东西就是我们的 **Bestiary** 模块现在可以用类似下面的行把符号输出到其他包里：

```
our @EXPORT =qw($camel %wolf ram);      # 缺省输出
our @EXPORT =qw(leopard @llama $emu);   # 请求时输出
our %EXPORT_TAGS = (
  camelids => [qw($camel @llama)],
  critters => [qw(ram $camel %wolf)],
);
```

从输出模块的角度出发，`@EXPORT` 数组包含缺省时要输出的变量和函数的名字：当你的程序说 `use Bestary` 的时候得到的东西。在 `@EXPORT_OK` 里的变量和函数只有当程序在 `use` 语句里面特别

要求它们的时候才输出。最后， `%EXPORT_TAGS` 里的键字/数值对允许程序包含那些在 `@EXPORT` 和 `@EXPORT_OK` 里面列出的特定的符号组。

从输入包的角度出发，`use` 语句声明了一列可以输入的符号，一组在 `%EXPORT_TAGS` 里面的名字，一个符号的模式或者什么也没有，这时在 `@EXPORT` 里的符号将从模块里输入到你的程序里。

你可以包含任意的这些语句，从 **Bestiary** 模块里输入符号：

```
use Bestiary;           # 输入@EXPORT符号
use Bestiary();         # 什么也不输入
use Bestiary qw(ram @llama); # 输入ram函数 和@llama数组
use Bestiary qw(:camelids); # 输入$camel和@llama
use Bestiary qw(:DEFAULT); # 输入@EXPORT符号
use Bestiary qw(/am/);   # 输入$camle, @llama, 和 ram
use Bestiary qw(/^\$/); # 输入所有标量
use Bestiary wq(:critters !ram); # 输入critters但是把ram排除
use Bestiary wq(:critters !:camelids);
# 输入 critters, 但是不包括camelids
```

把一个符号排除在输出列表之外（或者用感叹号明确地从输入列表里删除）并不会让使用模块的程序无法访问它。该程序总是可以通过带完整修饰词的包名来访问模块的包的内容，比如 `%Bestiary::gecko`。（因为词法变量不属于包，所以私有属性仍然可实现：参阅下一章的“私有方法”。）

你可以说 `BEGIN { $Exporter::Verbose=1 }`，这样就可以看到声明是如何处理的，以及实际上有什么东西输入到你的包里。

**Exporter** 本身是一个 Perl 模块，如果你觉得奇怪，你可以看看类型团巧妙地使用它把符号从一个包输出的另一个包，在 **Export** 模块里，起关键作用的函数叫 `import`，它做一些必要的别名工作，把一个包里的符号体现在另外一个包里。实际上，一个 `use Bestiary LIST` 语句和下面的语句完全一样：

```
BEGIN {
    require Bestiary;
    import Bestiary LIST;
}
```

这意味着你的模块并不一定要使用 **Exporter**。当你使用一个模块的时候，它可以做任何它喜欢干的事情，因为 `use` 只是为那个模块调用普通的 `import` 方法，而你可以把这个方法定义为处理任何你想干的事情。

### 11.2.1.1 不用 **Exporter** 的输入方法进行输出

**Export** 定义一个叫 `export_to_level` 的方法，用于你（因为某些原因）不能直接调用 **Exporter** 的 `import` 方法的情况下。`export_to_level` 方法用下面的方法调用：

```
MODULE->export_to_level($where_to_export, @what_to_export);
```

这里的 `$where_to_export` 是一个整数，标识调用模块的堆栈把你的符号输出了多远，而 `@what_to_export` 是一个数组，里面列出要输出的所有符号（通常是 `@_`）。

比如，假设我们的 **Bestiary** 有一个自己的 `import` 函数：

```
package Bestiary; @ISA = qw(Exporter); @EXPORT_OK = qw($zoo);

sub import { $Bestiary::zoo = "menagerie"; }
```

这个 `import` 函数的存在抑制了对 `Exporter` 的 `import` 函数的继承。如果你希望 `Bestiary` 的 `import` 函数在设置了 `$Bestiary::zoo` 之后的性质和 `Exporter` 的 `import` 函数一样，那么你应该象下面那样定义它：

```
sub import { $Bestiary::zoo = "menagerie"; Bestiary->export_to_level(1,@_); }
```

这样就把符号从当前包输出到“上面”一层的包里。也就是说，输出到使用 `Bestiary` 的程序或者模块里。

### 11.2.1.2 版本检查

如果你的模块定义了一个 `$VERSION` 变量，使用你的模块的程序可以识别该模块足够新。比如：

```
use Bestiary 3.14;          # Bestiary 必须是版本3.14或者更新
use Bestiary v1.0.4;       # Bestiary 必须是版本1.0.4或者更新
```

这些东西都转换成对 `Bestiary->require_version` 的调用，然后你的模块就继承了它们。

### 11.2.1.3 管理未知符号

有时候，你可能希望避免某些符号的输出。通常这样的情况出现在你的模块里有一些函数或者约束对某些系统而言没有什么用的时候。你可以通过把它们放在 `@EXPORT_FAIL` 数组里面避免把这些符号输出。

如果一个程序想输入这些符号中的任何一个，`Exporter` 在生成一个错误之前给模块一个处理这种情况的机会。它通过带着一个失败符号列表调用 `export_fail` 的方法来实现这个目的，你可以这样定义 `export_fail`（假设你的模块使用 `Carp` 模块）：

```
sub export_fail {
    my $class = shift;
    carp "Sorry, these symbols are unavailable: @_";
    return @_;
}
```

`Exporter` 提供缺省的 `export_fail` 方法，它只是简单地不加改变地返回该列表并且令 `use` 失败，同时给每个符号产生一个例外。如果 `export_fail` 返回一个空列表，那么就不会记录任何错误并且输出所有请求的符号。

### 11.2.1.4 标签绑定工具函数

因为在 `%EXPORT_TAGS` 里列出的符号必须同时在 `@EXPORT` 或者 `@EXPORT_OK` 里面出现，所以 `Exporter` 提供了两个函数让你可以增加这些标签或者符号：

```
%EXPORTER_TAGS = (foo => [qw(aa bb cc)], bar => [qw(aa cc dd)]);

Exporter::export_tags('foo');          # 把aa, bb和cc加到@EXPORT
Exporter::export_ok_tags('bar');       # 把aa, cc和dd加到@EXPORT_OK
```

声明非标签名字是错误的。

## 11.3 覆盖内建的函数

许多内建的函数都可以覆盖，尽管（就象在你的墙里面打洞一样）你应该只是偶然才做这些事情并且只有必要时才这么做。通常，那些试图在一个非 `Unix` 系统上仿真一些 `Unix` 系统的功能的包要这种用法。（不要把覆盖和重载两个概念混淆了，重载给内建的操作符增加了面向对象的含义，但并不覆盖什么东西。参阅第十三章里的重载模块的讨论，重载，获取更多信息。）



我们可以通过从一个模块里输入名字来实现重载——预定义的不够好。更准确地说，触发覆盖的是对一个指向类型团的代码引用的赋值动作，就象 `*open = \&myopen` 里一样。另外，赋值必须出现在其他的包里；这样就不大可能通过故意的类型团别名导致偶然的覆盖。不过，如果你真的是希望做你自己的覆盖，那也别失望，因为 `subs` 用法令你通过输入语法预定义子过程，这样，这些名字就覆盖了内建的名字：

```
use subs qw(chdir chroot chmod chown);
chdir $somewhere;
sub chdir {...}
```

通常，模块不应该把 `open` 或 `chdir` 这样的内建的名字放在缺省的 `@EXPORT` 列表里输出，因为这些名字可能会不知不觉地跑到别人的名字空间里，并且在人们不知情的情况下把语意改变了。如果模块包含的是在 `@EXPORT_OK` 列表里的名字，那么输入者就需要明确地请求那些要覆盖的内建的名字，这样才能保证每个人都是可信的。

内建的函数的最早的版本总是可以通过伪包 `CORE` 来访问。因此，`CORE::chdir` 将总是最初编译进 Perl 里的版本，即使 `chdir` 关键字已经被覆盖了。

不过，覆盖内建函数的机制总是被有意地限制在那些要求这样输入的包中。不过有一个更有覆盖性的机制可以让你在任何地方覆盖一个内建的函数，而不用考虑名字空间的限制。这是通过在 `CORE::GLOBAL` 伪包里定义该函数来实现的。下面是是用一个可以理解正则表达式的东西替换 `glob` 操作符的例子。（请注意，这个例子没有实现干净地覆盖 Perl 的内建 `glob` 的所有东西，`glob` 在不同的标量或者列表环境里的行为是不一致的。实际上，许多 Perl 内建都有这种环境敏感的行为，而一个写得好的覆盖应该充分支持这些行为。有关全功能的 `glob` 覆盖的例子，你可以学习和 Perl 绑定在一起的 `File::Glob` 模块。）总之，下面的是一个不全面的例子：

```
*CORE::GLOBAL::glob = sub {
    my $pat = shift;
    my @got;
    local *D;
    if (opendir D, '.') {
        $got = grep /$pat/, readdir D;
        closedir D;
    }
    return @got;
}

package Whatever;

print <^[a-z]+\..pm$>;      # 显示当前目录里的所有用法
```

通过全局覆盖 `glob`，这样的抢占强制在任何名字空间里使用一个新的（并且是破坏性的）`glob` 操作符，而不需要拥有该名字空间模块的认可和协助。自然，这么做必须非常小心——如果必须这么做的话。但很可能不是必须的。

我们对覆盖的态度是：变得比较重要很好，但是更重要的是变得更好。

---

Revision: r1.2 - 27 Aug 2005 - 13:22 - [TingYu](#)

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

---

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](#)