

第十三章， 重载

↓ 第十三章， 重载

- ↓ 13.1 overload 用法
- ↓ 13.3 可重载操作符
- ↓ 13.4 拷贝构造器 (=)
- ↓ 13.5 当缺失重载句柄的时候 (nomethod 和 fallback)
- ↓ 13.6 重载常量
- ↓ 13.7 公共重载函数
- ↓ 13.8 继承和重载
- ↓ 13.9 运行时重载
- ↓ 13.10 重载诊断

对象非常酷，但有时候它有点太酷了。有时候你会希望它表现得少象一点对象而更象普通的数据类型一点。但是实现这个却有问题：对象是用引用代表的引用，而引用除了当引用以外没什么别的用途。你不能在引用上做加法，也不能打印它们，甚至也不能给它们使用许多 Perl 的内建操作符。你能做的唯一一件事就是对它们解引用。因此你会发现自己在写许多明确的方法调用，象：

```
print $object->as_string;
$new_object = $subject->add($object);
```

象这样的明确的解引用通常都是好事；你决不能把你的引用和指示物混淆，除非你想混淆它们。下面可能就是你想混淆的情况之一。如果你把你的类设计成使用重载，你可以装做看不到引用而只是说：

```
print $object;
$new_object = $subject + $object;
```

当你重载某个 Perl 的内建操作符的时候，你实际上定义了把它应用于某特定类型的对象时的特性。有很多Perl的模块利用了重载，比如 `Math::BigInt`，它让你可以创建 `Math::BigInt` 对象，这些对象的性质和普通整数一样，但是没有尺寸限制。你可以用 `+` 把它们相加，用 `/` 把它们相除，用 `<=>` 比较它们，以及用 `print` 打印它们。

请注意重载和自动装载（`autoload`）是不一样的，自动装载是根据需要装载一个缺失的函数或方法。重载和覆盖（`overriding`）也是不一样的，覆盖是一个函数或方法覆盖了另外一个。重载什么东西也不隐藏；它给一个操作添加了新含义，否则在区区引用上进行该操作就是无聊的举动。

13.1 overload 用法

`use overload` 用法实现操作符重载。你给它提供一个操作符和对应的性质的键字/数值列表：

```
package MyClass;

use overload      '+' => \&myadd,          # 代码引用
'<' => "less_then";                        # 命名方法
'abs' => sub { return @_ },                # 匿名子过程
```

现在，如果你想相加两个 `MyClass2` 类，则调用 `myadd` 子过程来计算结果。

当你试图用 `<` 操作符比较两个 `MyClass2` 对象，Perl 注意到该性质被声明为一个字串，然后就把该字串当作一个方法名字而不仅仅是一个子过程名字。在上面的例子里，`less_then` 方法可以由 `MyClass2` 包本身提供或者从 `MyClass2` 的基类中继承过来，但是 `myadd` 子过程必须由当前包提供。匿名子过程 `abs` 自己提供得甚至更直接。不过这些过程你提供的，我们叫它们句柄（`handler`）。

对于单目操作符（那些只有一个操作数的东西，比如 **abs**），当该操作符应用于该类的一个对象的时候则调用为该类声明的句柄。

对于双目操作符而言，比如 **+** 或 **<**，当第一个操作数是该类的一个对象或当第二个操作数是该类的对象而且第一个操作数没有重载性质的时候，则调用该句柄。因此你可以用下面两种句法：

```
$object + 6
```

或：

```
6 + $object
```

而不用担心操作数的顺序。（在第二个例子里，在传递给句柄的时候操作数会对换）。如果我们的表达式是：

```
$animal + $vegetable
```

并且 **\$animal** 和 **\$vegetable** 是不同的类的对象，两个都使用了重载技术，那么 **\$animal** 的重载性质将被触发。（我们希望该动物喜欢蔬菜。）

在 **Perl** 里只有一个三目操作符，**?:**，而且幸运的是你不能重载它。

*重载句柄

在操作一个重载了的操作符的时候，其对应的句柄是带着三个参数调用的。前两个参数是两个操作数。如果该操作符只使用一个操作数，第二个参数是 **undef**。

第三个参数标明前两个参数是否交换。即使是在普通算术的规则里，有些操作也不怎么在乎它们的参数的顺序，比如加法和乘法；不过其他的東西，比如减法和除法则关心。（注：（注：当然，我们并不要求你重载的对象遵循普通算术，不过最好不要让人们吃惊。很奇怪的是，许多语言错误地用字符串连接功能重载了 **+**，它是不能交换的，而且只是暧昧的相加。要找一个不同的解决方法，请参阅 **Perl**。）看看下面两个的区别：

```
$object - 6
```

和：

1. - \$object

如果给一个句柄的头两个参数已经被交换过了，第三个参数将是真。否则，第三个参数是假，这种情况下还有一个更好的区别：如果该句柄被另外一个参与赋值的句柄触发（就象在 **+=** 里用 **+** 表示如何相加），那么第三个参数就不仅仅是假，而是 **undef**。这个区别可以应用一些优化。

举个例子，这里是一个类，它让你操作一个有范围限制的數字。它重载了 **+** 和 **-**，这样对象相加或相减的范围局限在 **0** 和 **255** 之间：

```
package ClipByte;

use overload '+' => \&clip_add,
              '-' => \&clip_sub;

sub new {
    my $class = shift;
    my $value = shift;
    return bless \$value => $class;
```

```

}

sub clip_add {
    my ($x, $y) = @_;
    my ($value) = ref($x) ? $$x : $x;
    $value += ref($y) ? $$y : $y;
    $value = 255 if $value > 255;
    $value = 0 if $value < 0;
    return bless \$value => ref($x);
}

sub clip_sub {
    my ($x, $y, $swap) = @_;
    my ($value) = (ref $x) ? $$x : $x;
    $value      -= (ref $y) ? $$y : $y;
    if ($swap) { $value = -$value }
    $value = 255 if $value > 255;
    $value = 0 if $value < 0;
    return bless \$value => ref($x);
}

package main;

$byte1 = ClipByte->new(200);
$byte2 = ClipByte->new(100);

$byte3 = $byte1 + $byte2;      # 255
$byte4 = $byte1 - $byte2;      # 100
$byte5 = 150 - $byte2;         # 50

```

你可以注意到这里的每一个函数实际上都是一个构造器，所以每一个都使用 **bless** 把它的新的对象赐福回给当前类--不管是什么；我们假设我们的类可以被继承。我们还假设如果 **\$y** 是一个引用，它是指向一个我们自己类型的对象的引用。除了测试 **ref(\$y)** 以外，如果我们想更彻底一些（也慢一些）我们也可以调用 **\$y->isa("ClipByte")**。

13.3 可重载操作符

你只能重载一部分操作符，它们在表 13-1 列出。当你用 **use overload** 时，操作符也在 **%overload::ops** 散列列出供你使用，不过其内容和这里的有一点区别。

表13-1。重载操作符

范畴	操作符
转换	"" 0+ bool
算术	+ - * / % ** x . neg
逻辑	!
位操作	& ~ ^ ! << >>
赋值	+= -= *= /= %= **= x= .= <=> >=> ++ --
比较	= < <> >= = <=> lt le gt ge eq ne cmp
数学	atan2 cos sin exp abs log sqrt
文本	<> </td>
解引用	\${} @{} %{} &{} *{}</td>

```

伪  ||nomethod fallback =>

```

请注意 **neg**, **bool**, **nomethod**, 和 **fallback** 实际上不是 Perl 的操作符。五种解引用, `"",` 和 `0+` 可能看起来也不象操作符。不过, 它们都是你给 **use overload** 提供的参数列表的有效键字。这不是什么问题。我们会告诉你一个小秘密: 说 **overload** 用法重载了操作符是一个小花招。它重载了下层的操作符, 不管是通过他们的“正式”操作符明确调用的, 还是通过一些相关的操作符隐含调用的。(我们提到的伪操作符只能隐含地调用。)换句话说, 重载不是在语句级发生的, 而是在语义级。原因是我们不是为了好看而是为了正确。请随意进行概括。

请注意 `=` 并不象你预料的那样重载 Perl 的赋值操作符。那样做是错的, 稍后详细介绍这个。

我们将从转换操作符开始讨论, 但并不是因为它们最显眼(它们可不抢眼), 而是因为它们是最有用的。许多类除了重载用 `" "` 键字(没错, 就是一行上的两个双引号。)声明的字串化以外不会重载任何东西。

转换操作符: `"", 0+, bool` 这三个键字让你给 Perl 提供分别自动转换成字串, 数字和布尔值的性质。

当一个非字串变量当作字串使用的时候, 我们就说是发生了字串化。当你通过打印, 替换, 连接, 或者是把它用做一个散列键字等方法把一个变量转换成字串时就发生 这个动作。字串化也是当你试图 **print** 一个对象时看到象 **SCALAR (0xba5fe0)** 这样的东西的原因。

我们说当一个非数字值在任意数字环境下转换成一个数字时发生的事情叫数字化, 这些数字环境可以是任意数学表达式, 数组下标, 或者是 `...` 范围操作符的操作数。

最后, 尽管咱们这里没有谁急于把它称做布尔化, 你还是可以通过创建一个 **bool** 句柄来定义一个对象在布尔环境里应该如何解释(比如 **if**, **unless**, **while**, **for**, **and**, **or**, **&&**, **||**, **?:**, 或者 **grep** 表达式的语句块)。

如果你已经有了它们中的任意一个, 你就可以自动生成这三个转换操作符的任何一个(我们稍后解释自动生成)。你的句柄可以返回你喜欢的任何值。请注意如果 触发转换的操作也被重载, 则该重载将在后面立即发生。

这里是一个 `" "` 的例子, 它在字串化时调用一个对象的 **as_string** 句柄。别忘记 引起引号:

```

package Person;

use overload q("") => \&as_string;

sub new { my $class = shift; return bless {@_} => $class; }

sub as_string {

my $self = shift; my ($key, $value, $result); while (( $key, $value) = each %$self)
{ $result .= "$key => $value\n"; } return $result; }

$obj = Person->new(height => 72, weight => 165, eyes => "vrown"); print $obj;

```

这里会打印下面的内容(一散列顺序), 而不是什么 **Person=HASH(0xba1350)**之类 的东西:

```
weight => 165 ...
```

(我们真诚的希望此人不是用公斤和厘米做单位的。)

算术操作符: `+`, `-`, `*`, `/`, `%`, `**`, `x`, `..`, **neg**

除了`neg` 以外这些应该都很熟悉，`neg` 是一个用于单目负号（`-123`里的 `-`）的特殊 重载键字。`neg` 和 `-` 键字之间的区别允许你给单目负号和双目负号（更常见的 叫法是减号）声明不同的性质。

如果你重载了 `-` 而没有重载 `neg`，然后试图使用一个单目负号，Perl 会为你模拟 一个 `neg` 句柄。这就是所谓的自动生成，就是说某些操作符可以合理地从其他 操作符中归纳出来（以“该重载操作符将和该普通操作符有一样的关系”为假设）。因为单目符号可以表示成双目符号的一个函数（也就是，`-123` 等于 `0-123`），在 `-` 够用的时候，Perl 并不强制你重载 `neg`。（当然，如果你已经独裁地定义了 双目负号用来把第二个参数和第一个参数分开，那么单目操作符会是一个很好的 抛出被零除例外的好方法。）

由 `.` 操作符进行的连接可以通过字串化句柄（见上面的 `""`）自动生成。

逻辑操作符：！

如果没有声明用于 `!` 的句柄，那么它可以用 `bool`，`""`，或者 `0+` 句柄自动生成。如果你重载了 `!` 操作符，那么当表现你的请求的时候，同样还会触发 `not` 操作符。（还记得我们的小秘密吗？）你可能觉得奇怪：其他逻辑操作符哪里去了？但是大多数逻辑操作符不能重载，因为它们是短路的。他们实际上是控制流 操作符，这样它们就可以推迟对它们的一些参数的计算。那也是 `?:` 操作符不能重载的原因。

位运算操作符：`&`，`|`，`~`，`^`，`<<`，`>>` `~` 操作符是一个单目操作符；所有其他的都是双目。下面是我们如何重载 `>>`，使之能做类似 `chop` 的工作：

```
package ShiftString2;

use overload '>>' => \&right_shift, '""' => sub { ${ $_[0] } };

sub new { my $class = shift; my $value = shift; return bless \$value => $class; }

sub right_shift { my ($x, $y) = @_ ; my $value = $$x; substr($value, -$y) = ""; return
bless \$value => ref($x); }

$camel = ShiftString2->new("Camel"); $ram = $camel >>2; print $ram; # Cam
```

赋值操作符：`+=`，`-=`，`*=`，`/=`，`%=`，`**=`，`x=`，`.=`，`<<=`，`>>=`，`++`，`--` 这些赋值操作符可以改变它们的参数的值或者就那样把它的参数放着。结果是只有 新值和旧值不同时才把结果赋值给左手边的操作数。这样就允许同一个句柄同时 用于重载 `+=` 和 `+`。尽管这么做是可以的，不过我们并不建议你这么做，因为根据 我们稍后将在“当缺失重载句柄的时候（`nomethod` 和`fallback`）”里描述的 语义，Perl 将为 `+` 激活该句柄，同时假设 `+=` 并没有直接重载。

连接（`.=`）可以用字串化后面跟着普通的字串连接自动生成。`++` 和 `--` 操作符 可以从 `+` 和 `-`（或者 `+=` 和 `-=`）自动生成。

实现 `++` 和 `--` 的句柄可能会变更（更改）他们的参数。如果你希望自增也能对 字母有效，你可以用类似下面的句柄实现：

```
package MagicDec;

use overload
q(-- ) => \&decrement,
q('') => sub { ${ $_[0] } };

sub new {
    my $class = shift;
    my $value = shift;
```

```

        bless \$value => \$class;
    }

sub decrement {
    my @string = reverse split (//, ${ $_[0] } );
    my $i;
    for ( $i = 0; $i < @string; $i++ ) {
        last unless $string[$i] =~ /a/i;
        $string[$i] = chr( ord($string[$i]) + 25 );
    }
    $string[$i] = chr( ord($string[$i]) - 1 );
    my $result = join(' ', reverse @string);
    $_[0] = bless \$result => ref($_[0]);
}

package main;

for $normal (qw/perl NZ pa/) {
    $magic = MagicDec->new($normal);
    $magic --;
    print "$normal goes to $magic\n";
}

```

打印出:

```

perl goes to perk
NZ goes to NY
Pa goes to Oz

```

很准确地对 Perl 神奇的字串自增操作符做了反向工程。

++\$a 操作符可以用 **\$a += 1**或 **\$a = \$a + 1** 自动生成, 而 **\$a--** 使用 **\$a -= 1** 或 **\$a = \$a - 1**。不过, 这样并不会触发真正的 **++** 操作符会触发的拷贝性质。参阅本章稍后的“拷贝构造器”。

比较操作符: **==**, **<**, **<=**, **>**, **>=**, **!=**, **<=>**, **lt**, **le**, **gt**, **ge**, **eq**, **ne**, **cmp** 如果重载了 **<=>**, 那么它可以用于自动生成 **<**, **<=**, **>**, **>=**, **=** 和 的性质。类似地, 如果重载了**cmp**, 那么它可以用于自动生成 **lt**, **le**, **gt**, **ge**, **eq** 和 **ne** 的性质。

请注意重载 **cmp** 不会让你的排序变成你想象的那么简单, 因为将要比较的是字串 化的版本的对象, 而不是对象本身。如果你的目的是对象本身那么你还会希望重载 **""**。

数学函数: **atan2**, **cos**, **sin**, **exp**, **abs**, **log**, **sqrt** 如果没有 **abs**, 那它可以从 **<** 或 **<=>** 与单目负号或者减号组合中自动生成。

重载可以用于为单目负号或者为 **abs** 函数自动生成缺失的句柄, 而它们本身也 可以独立的被重载。(没错, 我们知道 **abs** 看起来象个函数, 而单目负号象个 操作符, 但是从 Perl 的角度而言它们的差别没这么大。)

反复(迭代)操作符: **<>** 使用 **readline** (在它从一个文件句柄读入数据的时候, 象 **while ()** 里的) 或者 **glob** (当它用于文件聚团的时候, 比如在 **@files = <*. *>** 里)。

```

package LuckyDraw;

use overload
'<>' => sub {
    my $self = shift;

```

```

        return splice @$self, rand @$self, 1;
    };

    sub new {
        my $class = shift;
        return bless [ @_ ] => $class;
    }

    package main;

    $lotto = new LuckyDraw 1 .. 51;

    for (qw(1st 2nd 3rd 4th 5th 6th)) {
        $lucky_number = <$lotto>;
        print "The $_ lucky number is: $lucky_number.\n";
    }

    $lucky_number = <$lotto>;
    print "\nAnd the bonus number is: $lucky_number.\n";

```

在加州, 这些打印出:

```

The 1st lucky number is: 18
The 2nd lucky number is: 11
The 3rd lucky number is: 40

```

```

The 4th lucky number is: 7
The 5th lucky number is: 51
The 6th lucky number is: 33

```

```

And the bonus number is: 5

```

解引用操作符: `${}`, `@{}`, `%{}`, `&{}`, `*{}` 对标量, 数组, 散列, 子过程, 和团的解引用可以通过重载这五个符号来截获。

Perl 的 **overload** 的在线文档演示了你如何才能使用这些操作符来仿真你自己的 伪散列。下面是一个更简单的例子, 它实现了一个有匿名数组的对象, 但是使用 散列引用。别试图把他当作真的散列用; 你不能从该对象删除键字/数值对。如果 你想合并数组和散列符号, 使用一个真的伪散列。

```

package PsychoHash;

use overload '%{}' => \&as_hash;

sub as_hash {
    my ($x) = shift;
    return { @$x };
}

sub new {
    my $class = shift;
    return bless [ @_ ] => $class;
}

$scritter = new PsychoHash( height => 72, weight => 365, type => "camel" );

print $scritter->{weight};      # 打印 365

```


又见第十四章，捆绑变量，那里有一种机制你可以用于重新定义在散列，数组，和标量上的操作。

当重载一个操作符的时候，千万不要试图创建一个带有指向自己的引用的操作符。比如，

```
use overload '+' => sub { bless [ $_[0], $_[1] ] };
```

这么干是自找苦吃，因为如果你说 `$animal += $vegetable`，结果将令 `$animal` 是一个引用一个赐福了的数组引用，而该数组引用的第一个元素是 `$animal`。这是循环引用，意味着即使你删除了 `$animal`，它的内存仍然不会释放，直到你的进程（或者解释器）终止。参阅第八章，引用，里面的“垃圾收集，循环引用，和弱引用”。

13.4 拷贝构造器 (=)

尽管 `=` 看起来像一个普通的操作符，它做为一个重载的键字有点特殊的含义。它并不重载 Perl 的赋值操作符。它不能那样重载，因为赋值操作符必须保留起来用于赋引用的值，否则所有的东西就都完了。

`=` 句柄在下面的情况下使用：一个修改器（比如 `++`，`--`，或者任何赋值操作符）施用于一个引用，而该引用与其他引用共享其对象。`=` 句柄令你截获这些修改器并且让你自己拷贝该对象，这样拷贝本身也经过修改了。否则，你会把原来的对象给改了（*）。

```
$copy = $original;      # 只拷贝引用
++$copy;                 # 修改下边的共享对象
```

现在，从这开始。假设 `$original` 是一个对象的引用。要让 `++$copy` 只修改 `$copy` 而不是 `$original`，先复制一份 `$copy` 的拷贝，然后把 `$copy` 赋给一个指向这个新对象的引用。直到 `++$copy` 执行之后才执行这个动作，所以在自增之前 `$copy` 和 `$original` 是一致的——但是自增后就不一样了。换句话说，是 `++` 识别出拷贝的需要，并且调用你的拷贝构造器。

只有象 `++` 或 `+=`，或者 `nomethod` 这样的修改器才能知晓是否需要拷贝，我们稍后描述 `nomethod`。如果此操作是通过 `+` 自动生成的，象：

```
$copy = $original;
$copy = $copy + 1;
```

这样，那么不会发生拷贝，因为 `+` 不知道它正被当作修改器使用。

如果在某些修改器的运行中需要拷贝构造器，但是没有给 `=` 声明句柄，那么只要该对象是一个纯标量而不是什么更神奇的东西，就可以自动生成 `=` 的句柄。

比如，下面的实际代码序列：

```
$copy = $original;
...
++$copy;
```

可能最终会变成象下面这样的东西：

```
$copy = $original;
...
$copy = $copy->clone(undef, "");
$copy->incr(undef, "");
```

这里假设 `$original` 指向一个重载的对象，`++` 是用 `\&incr` 重载的，而 `=` 是用 `\&clone` 重载的。

类似的行为也会在 `$copy = $original++` 里触发，它解释成 `$copy = $original; ++$original`。

13.5 当缺失重载句柄的时候（**nomethod** 和 **fallback**）

如果你在一个对象上使用一个未重载的操作符，Perl 首先用我们早先描述过的规则，尝试从其他重载的操作符里自动生成一个行为。如果这样失败了，那么 Perl 找一个重载 **nomethod** 得到的行为，如果可行，则用之。这个句柄之于操作符就好象 **AUTOLOAD** 子过程之于子过程：它是你走投无路的时候干的事情。

如果用了 **nomethod**，那么 **nomethod** 键字应该后面跟着一个引用，该引用指向一个接受四个参数的句柄。前面三个参数和任何其他句柄里的没有区别；第四个是一个字串，对应缺失句柄的那个操作符。它起的作用和 **AUTOLOAD** 子过程里的 **\$AUTOLOAD** 变量一样。

如果 Perl 不得不找一个 **nomethod** 句柄，但是却找不到，则抛出一个例外。

如果你想避免发生自动生成，或者你想要一个失效的自动生成，这样你就可以得到一个完全没有重载的结果，那么你可以定义特殊的 **fallback** 重载键字。它有三个有用的状态：

- **undef** :
如果没有设置 **fallback**，或者你把它明确地设为 **undef**，那么不会影响重载事件序列：先找一个句柄，然后尝试自动生成，最后调用 **nomethod**。如果都失败则抛出例外。
- **false**:
如果把 **fallback** 设置为一个预定义的，但是为假的值（比如 **0**），那么永远不会进行自动生成。如果存在 **nomethod** 句柄，那么 Perl 将调用之，否则则抛出例外。
- **true**:
和 **undef** 有几乎一样的性质，但是如果不能通过自动生成合成一个合适的句柄，那么也不会抛出例外。相反，Perl 回复到该操作符没有重载的性质，就好像根本没有在那个类里面使用 **use overload** 一样。

13.6 重载常量

你可以用 **overload::constant** 改变 Perl 解释常量的方法，这个用法放在一个包的 **import** 方法里很有用。（如果这么做了，你应该在该包的 **unimport** 方法里正确地调用 **overload::remove_constant**，这样当你需要的时候，包可以自动清理自身。）

overload::constant 和 **overload::remove_constant** 都需要一个键字/数值对的列表。这些键字应该是 **integer**, **float**, **binary**, **q** 和 **qr** 中的任何东西，而且每个数值都应该是一个子过程的名字，一个匿名子过程，或者一个将操作这些常量的代码引用。

```
sub import { overload::constant ( integer => \&integer_handler,  
    float => \&float_handler,  
    binary => \&base_handler,  
    q => \&string_handler,  
    qr => \&regex_handler) }
```

当 Perl 记号查找器碰到一个常量数字的时候，就会调用你提供的任何用于 **integer** 和 **float** 的句柄。这个功能是和 **use constant** 用法独立的；象下面这样的简单语句：

```
$year = cube(12) + 1;    # 整数  
$pi = 3.14159265358979; # 浮点
```

会触发你要求的任何句柄。

binary 键字令你截获二进制，八进制和十六进制常量。**q** 处理单引号字串（包括用 **q** 引入的字串）和 **qq-** 里面的子字串，还有 **qx** 引起的字串和“此处”文档。最后，**qr** 处理在正则表达式里的常量片段，就好象在第五章，模式匹配，结尾处描述的那样。

Perl 会给句柄传递三个参数。第一个参数是原来的常量，形式是它原来提供给 Perl 的形式。第二个参数是 Perl 实际上如何解释该常量；比如，`123_456` 将显示为 `123456`。

第三个参数只为那些由 `q` 和 `qr` 处理字串的句柄定义，而且是 `qq`，`q`，`s`，或者 `tr` 之一，具体是哪个取决于字串的使用方式。`qq` 意味着该字串是从一个代换过的环境来的，比如双引号，反斜杠，`m//` 匹配或者一个 `s///` 模式的替换。`q` 意味着该字串来自一个未代换的 `s` 意味着此常量是一个在 `s///` 替换中来的替换字串，而 `tr` 意味着它是一个 `tr///` 或者 `y///` 表达式的元件。

该句柄应该返回一个标量，该标量将用于该常量所处的位置。通常，该标量会是一个重载了的对象的引用，不过没什么东西可以让你做些更卑鄙的事情：

```
package DigitDoubler;          # 一个将放在 DigitDoubler.pm 里的模块
use overload;

sub import { overload::constant ( integer => \&handler,
                                   float  => \&handler ) }

sub handler {
    my ($orig, $intergp, $context) = @_;
    return $interp * 2;          # 所有常量翻番
}

1;
```

请注意该 `handler`（句柄）由两个键字共享，在这个例子里运行得不错。现在当你说：

```
use DigitDoubler;

$strouble = 123;          # trouble 现在是 246
$jeopardy = 3.21;        # jeopardy现在是 6.42
```

你实际上把所有东西都改变了。

如果你截获字串常量，我们建议你同时提供一个连接操作符（`."`），因为所有代换过的表达式，象 `"ab$cd!!"`，仅仅只是一个更长的 `'ab'.$cd.'!!'` 的缩写。类似的，负数被类似的，负数被认为是正数常量的负值，因此，如果你截获整数或者浮点数，你应该给 `neg` 提供一个句柄。（我们不需要更早做这些，因为我们是返回真实数字，而不是重载的对象引用。）

请注意 `overload::constant` 并不传播进 `eval` 的运行时编译，这一点可以说是臭虫也可以说是特性——看你怎么看了。

13.7 公共重载函数

对 Perl 版本 5.6 而言，`use overload` 用法为公共使用提供了下面的函数：

1. `overload::StrVal(OBJ)`:
这个函数返回缺少字串化重载（`""`）时 `OBJ` 本应该有的字串值。
2. `overload::Overloaded(OBJ)`:
如果 `OBJ` 经受了任何操作符重载，此函数返回真，否则返回假。
3. `overload::Method(OBJ, OPERATOR)`:
这个函数返回 `OPERATOR` 操作 `OBJ` 的时候重载 `OPERATOR` 的代码，或者 在不存在重载时返回 `undef`。

13.8 继承和重载

继承和重载以两种方式相互交互。第一种发生在当一个句柄命名为一个字串而不是以一个代码引用或者匿名子过程的方式提供的时候。如果命名为字串，那么该句柄被解释成为一种方法，并且因此可以从父类中继承。

第二种重载和继承的交互是任何从一个重载的类中衍生出来的类本身也经受该重载。换句话说，重载本身是继承的。一个类里面的句柄集是该类的所有父类的句柄的联合（递归地）。如果在几个不同的祖先里都有某个句柄，该句柄的使用是由方法继承的通用规则控制的。比如，如果类 **Alpha** 按照类 **Beta**，类 **Gamma** 的顺序继承，并且类 **Beta** 用 `\&Beta::plus_sub` 重载 `+`，而 **Gamma** 用字串 `"plus_meth"` 重载 `+`，那么如果你对一个 **Alpha** 对象用 `+` 的时候，将调用 `Beta::plus`。

因为键字 `fallback` 的值不是一个句柄，它的继承不是由上面给出的规则控制的。在当前的实现里，使用来自第一个重载祖先的 `fallback` 值，不过这么做是无意的，并且我们可能会在不加说明的情况下改变（当然，是不加太多说明）。

13.9 运行时重载

因为 `use` 语句是在编译时重载的，在运行时改变重载的唯一方法是：

```
eval "use overload '+' =>\&my_add";
```

你还可以说：

```
eval "no overload '+', '--', '<='";
```

当然，在运行时使用这些构造器是很有问题的。

13.10 重载诊断

如果你的 **Perl** 是带着 `-DDEBUGGING` 编译的，你就可以用 `-Do` 开关或者等价物在运行时查看重载的诊断信息。你还可以用 **Perl** 内置的调试器的 `m` 命令推导是重载了哪个操作。

如果你现在觉得“重载”了，下一章可能会把这些东西给你约束回来。

Revision: r1.3 - 31 Aug 2005 - 15:28 - [TingYu](#)

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

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](#)