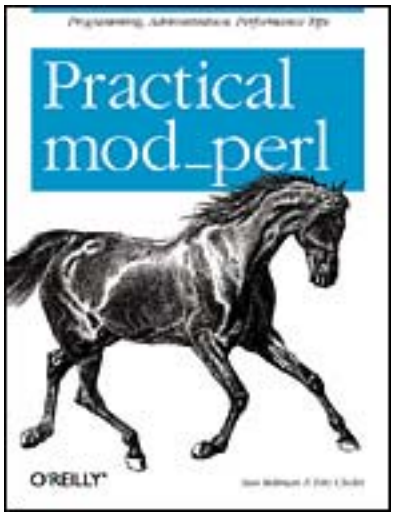
mod_perl 编程指南



Written by Stas Bekman, Eric Cholet
Translated by Jeff Peng

译者序

mod_perl 是个 Apache 模块，它巧妙的将 Perl 程序语言封装在 Apache web 服务器内。在 mod_perl 下，CGI 脚本比平常运行快 50 倍。另外，可将数据库与 web 服务器集成在一起，用 Perl 编写 Apache 模块，在 Apache 的配置文件里插入 Perl 代码，甚至以 server-side include 方式使用 Perl。在 mod_perl 下，Apache 不仅是一个 web 服务器，而变成了一个功能完善的程序平台。

原书名为"Practical Mod_perl"，由 mod_perl 专家 Stas Bekman 等写成，是本经典之作。本人在业余时间翻译了原书的第 6 章，即 mod_perl 代码指南部分。其他的 mod_perl 安装，配置，性能，数据库等内容，请读者自行阅读原书。

当前国内mod_perl主题相关的书很少，若遇到这方面的问题，可求助于mod_perl的用户邮件列表，有很多人愿意解答你的问题。如果是基本的perl问题，可在perl beginners邮件列表上询问。这两个都是高质量的邮件列表。前者的加入方法是发送一封空信到modperl-subscribe@perl.apache.org，后者的加入方法是发送一封空信到beginners-subscribe@perl.org。我本人有时在上面问问题，也有时回答别人的问题。

Jeff Peng
pangj@earthlink.net
2006 年 2 月

目　录

1 前言

1.1 看文档

在开始 mod_perl 编程之前，要求有一定的 perl 编程经验。至少要学会看文档，perldoc 是个很有用的工具，不多述。

1.2 打开 strict 标记

写任何 perl 程序时，请在脚本开头处加上：

```
use strict;
```

当然，也可在部分代码块里关闭这个标记的部分功能，例如：

```
use strict;
{
    no strict 'refs';
    my $var_ref = 'foo';
    $$var_ref = 1;
}
```

1.3 激活 warnings

在脚本开头处加上：

```
use warnings;
```

这对程序 debug 绝对有好处。

2 揭开 Apache::Registry 的神秘面纱

让我们以一个简单的脚本开始。如下简单的 cgi 脚本初始变量$counter 为 0，并递增的将其值 print 到浏览器：

```
#!/usr/bin/perl -w
use strict;

print "Content-type: text/plain\n\n";

my $counter = 0;
```

```
for (1..5) {
    increment_counter( );
}

sub increment_counter {
    $counter++;
    print "Counter is equal to $counter !\n";
}
```

当我们请求/perl/counter.pl（运行在 mod_perl 下），我们期望看到如下结果：

```
Counter is equal to 1 !
Counter is equal to 2 !
Counter is equal to 3 !
Counter is equal to 4 !
Counter is equal to 5 !
```

当第一次执行脚本时，会看到上述结果。现在，我们 reload 它几次。在 reload 后，计数器不会初始化为 0 了。当我们继续 reload 时，会发现这个计数器保持一直增长，但是没有规律，数字也许是随机的，例如：

```
Counter is equal to 6 !
Counter is equal to 7 !
Counter is equal to 8 !
Counter is equal to 9 !
Counter is equal to 10 !
```

两个异常之处：

1）计数器值超过 5；
2）在 reload 时，计数器增长没有规律。

出现这个现象的理由是，尽管$counter 在每次请求时都会增长，但它不会初始化为 0 了，尽管我们设置了这行：

```
my $counter = 0;
```

为什么这点在 mod_perl 下不能工作？

2.1 第一个秘密：为什么脚本运行的计数器超过 5？

假如我们看看 error_log 文件（没有激活 warnings），会看到类似如下的信息：

```
Variable "$counter" will not stay shared
```

at /home/httpd/perl/counter.pl line 13.

若脚本包含的命名嵌套函数（相对于匿名函数）使用了在该函数作用域之外定义的词法变量（my 声明的变量），就会产生如上警告。

你在上述脚本里看到了命名嵌套函数了吗？没有。但这并非 bug，因为在 mod_perl 下，perl 解析器会从不同的角度来看待问题。最容易找到问题的方法是在调试器上运行 code。

因为我们对运行在 web server 上的 perl 代码进行 debug，所以普通的调试器没什么作用，因为调试器必须在 web server 内部调用。幸运的是，可用 Doug MacEachern 的 Apache::DB 模块来进行 debug。Apache::DB 允许交互式的进行 debug，然而这里并不需要。

为了激活调试器，照如下方式修改 httpd.conf:

```
PerlSetEnv PERLDB_OPTS "NonStop=1 LineInfo=/tmp/db.out AutoTrace=1 frame=2"
PerlModule Apache::DB
<Location /perl>
    PerlFixupHandler Apache::DB
    SetHandler perl-script
    PerlHandler Apache::Registry
    Options ExecCGI
    PerlSendHeader On
</Location>
```

我们使用 PERLDB_OPTS 环境变量增加了一个调试器配置，它与在命令行调用调试器效果一样。我们也装载和激活了 Apache::DB，作为一个 PerlFixupHandler。

另外，还在<Perl>节装载了 Carp 模块，这点也可以在 startup.pl 文件里做：

```
<Perl>
    use Carp;
</Perl>
```

在做了上述改变后，我们重启 web 服务器，并且发布请求到/perl/counter.pl。跟以前一样，表面上没什么变化，输出也一样。然而，实际上在后台发生了 2 件事：

1）创建了/tmp/db.out 文件，里面记录了 code 的完整 trace。

2）既然我们 load 了 Carp 模块，error_log 文件现在包含了被实际执行的真正代码。它扩充了前面我们见到的错误日志的信息。

如下是实际执行的 code:

```
package Apache::ROOT::perl::counter_2epl;
```

```
use Apache qw(exit);
sub handler {
    BEGIN {
        $^W = 1;
    };
    $^W = 1;

    use strict;

    print "Content-type: text/plain\n\n";

    my $counter = 0;

    for (1..5) {
        increment_counter( );
    }

    sub increment_counter {
        $counter++;
        print "Counter is equal to $counter !\n";
    }
}
```

当然，code 在 error_log 里出现时可不会自动缩进，我们缩进它只是为了可清楚的看出，这些 code 实际被包裹在 handler()子函数里。

通过观察这个 code，我们知道每个 Apache::Registry 脚本被 cache 在一个 package 里，包的名字由 Apache::Root:: 的前缀和脚本的 URI（/perl/counter.pl）组成，将所有"/"替换成了"::"，将"."替换成了"_2e"。这就是 mod_perl 从用户请求里知道该获取哪个脚本的原因--每个脚本被转换成 package，它的名字唯一，并且只包含单一的子函数 handler()，子函数里的代码才是用户真正写的 code。

本质上，发生异常的原因是因为 increment_counter()这个子函数，它使用了在该函数作用域之外定义的词法变量，这样这个子函数就变成了一个闭包。闭包正常来说并不触发警告，但在该例子里，这个子函数是个嵌套子函数。这意味着子函数 handler()第一次被调用时，2 个子函数都使用了同一变量。然而从那一刻起，increment_counter()将保持它自己的$counter 的私有拷贝（所以$counter 不是共享的）并且递增它自己的拷贝。因为这点，$counter 的值保持一直增长，并且永不会初始化为 0 了。

假如我们在脚本里打开 diagnostics，它会把简要的警告信息转换为详细的报告，这样就可以在警告文本里见到嵌套子函数的参考信息。通过观察执行的 code，很清楚的知道 increment_counter()是个命名嵌套子函数，它定义在 handler()子函数里。

任何在脚本主体里定义的子函数，在 Apache::Registry 下执行时会变成嵌套子函数。假

如 code 位于库文件或 module 里，它被当前脚本 require 或 use 调用时，不会发生这个问题。

例如，假如我们将 code 从脚本移到 run()子函数里，将子函数放置在 mylib.pl 文件里，将这个文件存放在脚本自身的目录，然后 require()它，这时没有任何问题。如下 2 个示例展示了这个用法。

Example 6-1. mylib.pl

```
my $counter;
sub run {
    $counter = 0;
    for (1..5) {
        increment_counter( );
    }
}
sub increment_counter {
    $counter++;
    print "Counter is equal to $counter !\n";
}
1;
```

Example 6-2. counter.pl

```
use strict;
require "./mylib.pl";
print "Content-type: text/plain\n\n";
run( );
```

这种解决方案是最容易和最快速的解决嵌套子函数问题的方法。你要做的就是将 code 移到一个独立的文件，并封装在子函数里，以便你以后调用它。这样就可将导致问题的词法变量放在子函数之外。

通常来讲，最好将所有 code 放在外部库文件里（除非脚本非常短），在主脚本里只需少数几行 code 即可。通常主脚本简单的调用库文件里的主函数，一般命名为 init()或 run()。这样，你就不必担心命名嵌套子函数的影响。

然而你会意识到，这种快速解决方法会不同程度的有问题。假如你有许多脚本，你可能会将多个脚本的 code 写进相似的文件名，例如 mylib.pl。这样，更清晰的解决方案是花多点时间来定义 package 名称，如下 2 例所示：

Example 6-3. Book/Counter.pm

```
package Book::Counter;
```

```
my $counter = 0;

sub run {
        $counter = 0;
        for (1..5) {
                increment_counter( );
        }
}

sub increment_counter {
        $counter++;
        print "Counter is equal to $counter !<BR>\n";
}

1;
_   _END_   _
```

Example 6-4. counter-clean.pl

```
use strict;
use Book::Counter;

print "Content-type: text/plain\n\n";
Book::Counter::run( );
```

唯一的不同就是 package 的声明。只要包名唯一，你就不必担心会与运行在本服务器上的其他脚本冲突。

另一个解决该问题的方法是将词法变量改变为全局变量。有 2 种方式的全局变量：

1）使用 vars 参数。在 use strict 'vars'情况下，全局变量能使用 use vars 来声明。例如，如下 code:

```
use strict;
use vars qw($counter $result);
# later in the code
$counter = 0;
$result   = 1;
```

类似于没有 use strict 的下述情况：

```
$counter = 0;
```

```
$result    = 1;
```

显然，前者要清晰的多，它允许你先声明全局变量再使用它们，这样就避免了不小心拼写错误的变量被作为未声明的全局变量来对待。

使用 use vars 的唯一弊端是，每个这样声明的全局变量会比不声明但完全有效的全局变量浪费更多内存，我们会在随后见到。

2）使用完全包限定的变量。不同于$counter，我们应该使用$Foo::counter，这样将全局变量$counter 放置在 package Foo 里。注意我们不知道 Apache::Registry 会指派哪个包名给脚本，因为它依赖于该脚本被调用的位置。记住全局变量在使用前，必须被初始化。

Perl 5.6.x 引进了另一种方法，叫做 our()声明。our()能被用在不同的作用域内，类似于 my()，但它创建了全局变量。

最后，如果我们总是将变量作为参数传递给函数，就可完全避免上述问题。

Example 6-5. counter2.pl

```
#!/usr/bin/perl -w
use strict;

print "Content-type: text/plain\n\n";

my $counter = 0;

for (1..5) {
    $counter = increment_counter($counter);
}

sub increment_counter {
    my $counter = shift;

    $counter++;
    print "Counter is equal to $counter !\n";

    return $counter;
}
```

在本实例里，没有变量共享问题。该方法的弊端是增加了子函数传递和返回变量的性能开销。但另一方面，它确保你的 code 做正确的事，不依赖于该子函数是否封装在其他块里，而这正是 Apache::Registry 家族所做的事。

当 Stas（原文作者）刚开始使用 mod_perl 时，并不熟悉这种嵌套子函数的问题，他要

写一个复杂的注册程序，运行在 mod_perl 下。这里展示了他的脚本的有趣的部分：

```
use CGI;
$q = CGI->new;
my $name = $q->param('name');
print_response( );

sub print_response {
    print "Content-type: text/plain\n\n";
    print "Thank you, $name!";
```

Stas 和他的老板在开发服务器上检查了程序，它运行良好。于是他们决定将它放到产品环境。一切看起来正常，但老板决定多检查程序几遍。这样就发生了令人惊异的事，在反复提交几次后，老板见到了"Thank you,Stas!"，而不是"Thank you, The Boss!"。

在研究该问题后，他们知道问题根源在于嵌套子函数。为什么在开发环境下没有注意到这个问题呢？我们会简要解释它。

对第一个神秘问题的结论是，记得在开发服务器上打开脚本的 warnings 模式，并且观察 error_log 日志里的警告信息。

2.2 第二个秘密：在 reload 时无规律的增长

让我们返回到最开始的示例，继续解释见到的第二个神秘。为什么在 reload 时，会见到无规律的增长结果呢？

每次父进程接受到客户端请求时，它将请求递交给子进程。每个子进程运行该脚本的自己的一份拷贝。这意味着每个子进程有它自己的$counter 的拷贝，它将会独立增长。所以不仅是每个$counter 的值在每次调用时独立增长，也因为不同的子进程在不同的时间处理请求，这样页面上看到的增长就毫无规律。例如，假如有 10 个 httpd 子进程，头 10 个 reload 也许是正常的（假如每个请求被分配到不同的子进程）。但一旦 reload 再次调用这些子进程，就会发生奇怪的现象。

而且，请求的出现是随机的，子进程也不总是处理同一个请求。在任何给定的时刻，子进程之一可能运行同一脚本许多次，而其他子进程可能从来没运行过这个脚本。

Stas 和他的老板没有发现前述用户注册系统的问题，因为在开发环境上 error_log 里充满着多个子进程的各种警告信息。

为了直观的重现这个问题，你必须以单进程方式运行 web 服务器。可以用-X 选项来运行 web 服务器：

```
panic% httpd -X
```

既然没有其他子进程在运行，就可以在第 2 次 reload 时重现问题。

激活 warnings 模式并且观察 error_log 日志，会帮助你检测到大部分可能的错误。某些警告会变成错误，我们已见到。应该检查每个 warning 并消除它，让它们不再出现在 error_log 里。假如 error_log 文件在每次脚本调用时会产生数百行的信息，就会难以观察和锁定真正的问题，并且若在产品环境下，如果你的站点很流行，这个文件会很快增长直到超出磁盘空间。

（译者注：实际上闭包所引起的问题不是三言两语能说清楚的，请看看其他关于闭包即 closure 的文档。若仍有不明，可来信咨询本人。）

3 命名空间

假如服务由单个脚本组成，可能没有命名空间问题。但 web 服务通常包含许多脚本和句柄。在下述章节里，我们将研究可能的命名空间问题，及其解决方案。首先请理解 perl 的两个特殊变量，@INC 和%INC。

3.1 @INC 数组

Perl 的@INC 数组类似于 shell 程序的 PATH 环境变量。就如同 PATH 包含了用于搜索可执行程序的目录列表一样，@INC 包含了可以装载 perl 模块和库文件的目录。

在使用 use(),require(),或 do()来装载文件或模块时，perl 从@INC 里得到一个目录列表，并且在这个目录列表里搜索需要装载的文件。假如你想要 load 的文件没有在这个目录列表里，就得告诉 perl 到哪里去找这个文件。你可以提供一个相对于@INC 里某个目录的路径，或提供文件的绝对路径。

3.2 %INC 哈希

%INC hash 用于缓存已被 use(),require()或 do()装载和编译过的文件和模块名。每次一个文件或模块被成功装载后，一个新的 key-value 对就被增加到%INC。key 就是文件或模块的名字。假如文件或模块在@INC 目录（除了"."）里可找到，文件名就包含了完整路径。每个 perl 解析器，和 mod_perl 下的每个进程，有它自己的%INC hash，用于存储已编译过的模块信息。

在试图用 use()或 require()装载文件或模块前，perl 检查是否它已经位于%INC hash 里。假如已存在，load 和编译过程就不会发生。否则，文件被载入内存，并试图去编译它。注意 do()无条件的装载文件或模块，它不检查%INC hash。在下面的例子里会看到实际中这点如何工作。

首先，让我们检查系统中@INC 的内容：

```
panic% perl -le 'print join "\n", @INC'
/usr/lib/perl5/5.6.1/i386-linux
/usr/lib/perl5/5.6.1
/usr/lib/perl5/site_perl/5.6.1/i386-linux
/usr/lib/perl5/site_perl/5.6.1
/usr/lib/perl5/site_perl
.
```

注意，当前目录"."是列表里的最后一个目录。

现在载入模块 strict.pm 并观察%INC 的内容：

```
panic% perl -le 'use strict; print map {"$_ => $INC{$_}"} keys %INC'
strict.pm => /usr/lib/perl5/5.6.1/strict.pm
```

既然 strict.pm 存在于 /usr/lib/perl5/5.6.1/目录里，并且 /usr/lib/perl5/5.6.1/是@INC 的一部分，%INC 就包含了 strict.pm 的完整路径作为 hash 的 value。

我们创建一个最简单的模块/tmp/test.pm:

```
1;
```

这样做绝对没问题，它在装载时返回一个真值，这足够告诉 perl 它已被正确装载。我们以不同的方式装载它：

```
panic% cd /tmp
panic% perl -e 'use test; \
        print map { "$_ => $INC{$_}\n" } keys %INC'
test.pm => test.pm
```

既然文件位于当前目录，相对路径被用做 hash 的 value。
现在再在@INC 里增加目录/tmp:

```
panic% cd /tmp
panic% perl -e 'BEGIN { push @INC, "/tmp" } use test; \
        print map { "$_ => $INC{$_}\n" } keys %INC'
test.pm => test.pm
```

这里仍会得到相对路径，这是因为模块首先会在当前目录(".")里发现，而/tmp 目录在@INC 列表里的位置在"."之后。假如我们在不同的目录执行相同的代码，"."目录就不会匹配到：

```
panic% cd /
panic% perl -e 'BEGIN { push @INC, "/tmp" } use test; \
        print map { "$_ => $INC{$_}\n" } keys %INC'
```

test.pm => /tmp/test.pm

这样就得到了完整路径。也可以用 unshift()来预增加路径，以便它在"."之前就被匹配到。如下也能得到完整路径：

```
panic% cd /tmp
panic% perl -e 'BEGIN { unshift @INC, "/tmp" } use test; \
        print map { "$_ => $INC{$_}\n" } keys %INC'
test.pm => /tmp/test.pm
```

代码：

```
BEGIN { unshift @INC, "/tmp" }
```

能被替换成更典雅的方式：

```
use lib "/tmp";
```

两者差不多，但后者更值得推荐。

上述这些修改@INC 的方法也有一定弊端：在文件系统里移动脚本，可能要求修改路径。

3.3 模块和库的名字冲突

3.3.1 第一个冲突情况

在同一个 server 上使用相同名称的 2 个模块是不可能的。仅仅第一个被 use()或 require()发现的模块会被载入和编译。所有随后的对相同名称的模块的请求将被跳过，因为 Perl 发现已在%INC hash 里存在这个模块的入口。

在如下例子里，2 个独立的项目位于不同的目录，projectA 和 projectB，它们运行在同一 server 上。2 个项目都使用了一个名为 MyConfig.pm 的模块，但是每个项目的 MyConfig.pm 模块里的 code 是完全不同的。如下显示了项目在文件系统中的位置（都位于/home/httpd/perl 目录下）：

```
projectA/MyConfig.pm
projectA/run.pl
projectB/MyConfig.pm
projectB/run.pl
```

请看如下的示例代码：

Example 6-6. projectA/run.pl

```
use lib qw(.);
```

```
use MyConfig;
print "Content-type: text/plain\n\n";
print "Inside project: ", project_name( );
```

Example 6-7. projectA/MyConfig.pm

```
sub project_name { return 'A'; }
1;
```

Example 6-8. projectB/run.pl

```
use lib qw(.);
use MyConfig;
print "Content-type: text/plain\n\n";
print "Inside project: ", project_name( );
```

Example 6-9. projectB/MyConfig.pm

```
sub project_name { return 'B'; }
1;
```

2 个项目都包含了一个脚本，名为 run.pl，它载入模块 MyConfig.pm 并且打印一个标明身份的消息，这依赖于 MyConfig.pm 模块里的 project_name() 函数。当发布请求到 /perl/projectA/run.pl 时，期望输出：

Inside project: A

类似的，/perl/projectB/run.pl 期望响应：

Inside project: B

当在单进程服务器模式下测试时，仅仅第一个运行的脚本会载入 MyConfig.pm 模块，尽管 2 个 run.pl 脚本都调用了 use MyConfig。当第二个脚本运行时，perl 会跳过 use MyConfig;陈述，因为 MyConfig.pm 已经位于%INC 里了。perl 会在 error_log 里报告这个问题：

```
Undefined subroutine
&Apache::ROOT::perl::projectB::run_2epl::project_name called at
/home/httpd/perl/projectB/run.pl line 4.
```

这是因为模块没有申明成包名，所以 project_name()子函数被插入了 projectA/run.pl 的名字空间里，Apache::ROOT::perl::projectA::run_2epl。projectB 没有载入这个模块，所以它根本就未获取到这个子函数。

注意假如 use 了库文件而不是模块（例如 config.pl 而不是 MyConfig.pm），结果会是一样的。

对库文件和模块都一样，某个文件被载入时，它的名字会被插入%INC。

3.3.2 第二个冲突情况

考虑如下情况：

project/MyConfig.pm
project/runA.pl
project/runB.pl

这个项目它有 2 个脚本，runA.pl 和 runB.pl，两者都试图载入相同的模块，MyConfig.pm，见如下示例：

Example 6-10. project/MyConfig.pm

```
sub project_name { return 'Super Project'; }
1;
```

Example 6-11. project/runA.pl

```
use lib qw(.);
use MyConfig;
print "Content-type: text/plain\n\n";
print "Script A\n";
print "Inside project: ", project_name( );
```

Example 6-12. project/runB.pl

```
use lib qw(.);
use MyConfig;
print "Content-type: text/plain\n\n";
print "Script B\n";
print "Inside project: ", project_name( );
```

这种情况会遭遇跟前面一样的问题：仅仅第一个脚本会正确运行，第二个会失败。问题发生的原因，是因为没有包申明。

随后会讨论解决这些问题的方法。

3.3.3 快速但低效的解决方案

如下解决方案可作为临时使用的方法。通过修改%INC 或者用 do()来代替 use()和 require()，可以强迫载入模块。

假如在调用 require()或 use()前，删除%INC 里的模块入口，那这个模块就会每次都被载入和重新编译。如下所示：

Example 6-13. project/runA.pl

```
BEGIN {
    delete $INC{"MyConfig.pm"};
}
use lib qw(.);
use MyConfig;
print "Content-type: text/plain\n\n";
print "Script A\n";
print "Inside project: ", project_name( );
```

对 runB.pl 做同样的修改。

另一个方法是通过 do()来强迫 reload 模块，如下所示：

Example 6-14. project/runA.pl 用 do 而不是 use 来强迫载入模块

```
use lib qw(.);
do "MyConfig.pm";
print "Content-type: text/plain\n\n";
print "Script B\n";
print "Inside project: ", project_name( );
```

对 runB.pl 做同样的修改。

如果你需要从被载入模块里 import 进某些东西，那就显式的调用 import()方法。例如，假如有这样的情况：

```
use MyConfig qw(foo bar);
```

那么 code 可这样写：

```
do "MyConfig.pm";
MyConfig->import(qw(foo bar));
```

两种解决方案都很低效，因为模块在每次请求里都要被重新载入，降低了响应速度。因此，上述方法只应在快速解决问题的情况下使用，真正有效的解决方法请见如下所述。

3.3.4 第一种解决方案

上节提的第一个冲突情况，可以通过将模块放在子目录结构，以便它们有不同的路径前缀来解决。目录情况看起来如下：

projectA/ProjectA/MyConfig.pm
projectA/run.pl
projectB/ProjectB/MyConfig.pm
projectB/run.pl

run.pl 也做相应的修改：

use ProjectA::MyConfig;
以及：
use ProjectB::MyConfig;

然而，假如以后我们想增加新的脚本到上述项目中，会又面临上节的第二种冲突情况。所以本方法只是半个解决方案。

3.3.5 第二种解决方案

另一种方法是在脚本里使用全路径，以便作为%INC hash 里的 key。

require "/home/httpd/perl/project/MyConfig.pm";

用这种方法，我们把 2 种冲突情况都解决了，但失去了某些便利性。每次项目在文件系统中移动，就必须调整路径。这点使得多开发者环境的版本控制无法进行，如果每个开发者都想把 code 放到不同的绝对路径的话。

3.3.6 第三种解决方案

该解决方案在模块里用到了包名声明。例如：

package ProjectA::Config;

类似的，对 ProjectB，包名就是 ProjectB::Config。

每个包名相对于同一 httpd 服务器的其他包名，应该是唯一的。%INC 就可以使用唯一包名作为 key，而不是用模块的文件名。至少应该使用 2 部分的包名作为你的私有模块名（例如 MyProject::Carp 而不是 Carp），因为后者容易与已存在的标准包名冲突。即使同一个名字的包现在不会存在于产品发布中，但以后别人也许会选择你已使用的名字，从而造成冲突。

包声明的意义是什么呢？在模块里没有包声明，使用 use()和 require()就非常便利，因为所有的来自被载入包的变量和子函数，会驻留在脚本自身的同一包里。它们可被方便的使用，就如同它们定义在脚本自身的同一范围内一样。该方法的弊端就是模块里的变量可能与主脚本里的变量冲突，这样会造成难以发现的 bug。

在模块里声明了包，事情变得复杂一点。假定包名是 PackageA，PackageA::project_name()这样的语法用于调用子函数 project_name()。如果未声明包，就可以简单的直接调用 project_name()。类似的，全局变量$foo 现在必须以$PackageA::foo 方式来使用，而不是$foo。在 PackageA 里的词法变量（my 声明的）在该包之外不可访问。

如果把全局变量和子函数 import 进当前脚本的名字空间，那就可以不带包限定词来使用它们。例如：

use MyPackage qw(:mysubs sub_b $var1 :myvars);

模块可以 export 出任何全局符号，但通常仅仅子函数和全局变量可被 export 出。注意该方法的弊端是浪费更多内存。请见 perldoc Exporter 关于 export 其他变量和符号的更多信息。

我们重写代码，以真正清晰的方式来解决前面的第二个冲突。如下显示文件的位置情况，它们相对于/home/httpd/perl 目录：

project/MyProject/Config.pm
project/runA.pl
project/runB.pl

如下是代码示例：

Example 6-15. project/MyProject/Config.pm

```
package MyProject::Config
sub project_name { return 'Super Project'; }
1;
```

Example 6-16. project/runB.pl

```
use lib qw(.);
use MyProject::Config;
print "Content-type: text/plain\n\n";
print "Script B\n";
print "Inside project: ", MyProject::Config::project_name( );
```

Example 6-17. project/runA.pl

```
use lib qw(.);
```

```
use MyProject::Config;
print "Content-type: text/plain\n\n";
print "Script A\n";
print "Inside project: ", MyProject::Config::project_name( );
```

你可见到，我们创建了 MyProject/Config.pm 文件，并且在顶部增加了包声明：

```
package MyProject::Config
```

现在 2 个脚本都载入这个模块，并且访问模块的子函数 project_name()，必须用完全包限定的名字，MyProject::Config::project_name( )。

请顺便阅读 perlmodlib 和 perlmod 的 manpage。

从上述讨论中，也清楚的知道不能在同一 Apache 服务器上运行开发环境和产品环境。必须对每种环境运行独立的服务器。假如你想在同一服务器上运行不止一个开发环境，可使用 Apache::PerlVINC。

4 mod_perl 环境下的 perl 规范

在下述节里，我们讨论 mod_perl 下的 perl 行为规范。

4.1 exit()

perl 的 exit()函数不能用于 mod_perl 代码。调用它会导致 mod_perl 进程退出，这违背了使用 mod_perl 的初衷。Apache::exit()函数可用作代替。从 perl 版本 5.6.0 开始，mod_perl 使用 CORE::GLOBAL::这个新的魔法包覆盖了 exit()。

关于 CORE::Package

CORE::是个特殊的包，它提供对 perl 内置函数的访问。可以用这个包来覆盖某些内建函数。例如，假如你想覆盖 exit()内建函数，那可以这样写：

```
use subs qw(exit);
exit( ) if $DEBUG;
sub exit { warn "exit( ) was called"; }
```

现在当你调用 exit()时，程序不会退出，代替的，它打印一条警告消息"exit( ) was called"。如果你想用原始的内建函数，就需这样写：

```
# the 'real' exit
CORE::exit( );
```

Apache::Registry 和 Apache::PerlRun 默认用 Apache::exit()覆盖了 exit()。这样，运行在这些模块下的脚本不必刻意修改来使用 Apache::exit()。

假如运行在 mod_perl 下的脚本调用了 CORE::exit()，子进程会退出，但当前请求不会被 log 下来。更重要的是，不会执行正确的退出。例如，假如有某些数据库句柄，它们会保持打开，导致内存浪费甚至数据库连接泄露。

假如需要 kill 子进程，该用 Apache::exit(Apache::Constants::DONE)代替。这会导致服务文雅的退出，完成 log 功能和协议请求。

假如在请求完成后，需要干净的 kill 子进程，请用$r->child_terminate 方法。该方法可在 code 里的任何地方调用，而不仅是末尾。该方法设置 MaxRequestsPerChild 配置指令的值为 1，并且清除 keepalive 标记。在请求处理完后，当前连接会中断，因为 keepalive 标记设置为 false，并且父进程会告诉子进程干净的退出，因为 MaxRequestsPerChild 值小于或等于已服务请求的数量。

在 Apache::Registry 脚本里可以这样写：

```
Apache->request->child_terminate;
```

或在 httpd.conf 里：

```
PerlFixupHandler "sub { shift->child_terminate }"
```

仅在你希望每次调用 register 句柄后子进程都终止时，才会用到第 2 种配置。这可能不是你想要的。

也可以使用 post-processing 句柄来引发子进程终止。如果你想在进程退出前，执行自己的清除代码，那可以这样写：

```
my $r = shift;
$r->post_connection(\&exit_child);

sub exit_child {
    # some logic here if needed
    $r->child_terminate;
}
```

上述 code 是 Apache::SizeLimit 模块所用的，它终止那些 size 增长得大于预设值的子进程。

4.2 die()

die()通常用于发生错误情况下，退出程序执行流。例如，通常打开文件会这样写：

open FILE, "foo" or die "Cannot open 'foo' for reading: $!";

假如文件不能打开，脚本会 die()；程序执行流会退出，打印 die 的理由，perl 解析器也会终止。

在编写良好的 perl 脚本里至少有一个 die()陈述。

运行在 mod_cgi 下的 CGI 脚本在完成时会退出，perl 解析器也会退出。这样，不必在意解析器退出是因为脚本自然完成，还是因为 die()而引起的。

然而在 mod_perl 下，我们不想子进程退出。这样，mod_perl 在背后谨慎的处理了这点，die()调用不会导致子进程退出。当调用 die()时，mod_perl 记录错误消息，并且调用 Apache::exit()而不是 CORE::die()。这样，脚本终止了，但是进程没有退出。当然，这里讨论的 die()并没有被 code 封装在某个 exception 句柄里（例如 eval{}块），或者是$SIG{__DIE__} 信号句柄，它允许你覆盖 die()行为。

4.3 全局变量的持续存在行为

在 mod_perl 下子进程在处理完单个请求后不会退出。这样，全局变量跨越请求而持续存在于同一个进程里。这意味着你必须很谨慎的处理那些在请求开始处，没有被初始化的全局变量的值。例如：

```
# the very beginning of the script
use strict;
use vars qw($counter);
$counter++;
```

perl 初始化未定义的$counter 值为 0，++这个自增操作符，会将这个值设为 1。然而，当同一代码在相同的进程里执行第二次时，$counter 不再是未定义的了。代替的，它会储藏脚本前次执行完后所获得的值。这样，干净的写法应该是：

```
use strict;
use vars qw($counter);
$counter = 0;
$counter++;
```

实际中，应该避免使用全局变量，除非没得选择。全局变量导致的大部分问题在于它们的值跨越函数，并且难于跟踪哪个函数在哪里修改了全局变量。通过 local()将这些变量限制在一个区域内，可以解决问题。但假如你已在这样做，那么用词法变量更好，因为它的范围有清楚的定义，而不像 local()的变量，它仍可在 code 里的任何地方可见和修改。参考 perlsub 获取更多细节。现在示例可以这样写：

```
use strict;
```

```
my $counter = 0;
$counter++;
```

请注意，既申明又初始化变量是好的编程实践，因为这样做会清楚的传达你的意图给代码维护者。

对 perl 的特殊变量尤其要谨慎，它们不能是词法范围的。对特殊变量，必须使用 local()。例如，假如你想一次读取整个文件，就需要 undef()输入记录分隔符。下列 code 读取整个文件到一个标量：

```
open IN, $file or die $!;
$/ = undef;
$content = <IN>; # slurp the whole file in
close IN;
```

因为全局的修改了 perl 的特殊变量$/，它会影响其他运行在同一个进程里的 code。假如 code 里的某处需要逐行读取文件内容（它依赖于$/的默认值），这个 code 就不会正确工作。将对这个特殊变量的修改行为限制在某个范围内，可解决这样的潜在问题：

```
{
  local $/; # $/ is undef now
  $content = <IN>; # slurp the whole file in
}
```

注意这个 local()的作用域被封装在块里。当程序控制走出这个块时，以前的$/的值会被自动还原。

4.4 STDIN,STDOUT 和 STDERR 流

在 mod_perl 下，STDIN 和 STDOUT 被绑定到请求进来的 socket 上。假如你使用某个第三方的模块，它打印一些输出到 STDOUT，但你想避免这点，就必须临时性的重定向 STDOUT 到/dev/null。然后，如果仍想发送响应到客户端，就必须恢复 STDOUT 到原始的句柄。下述 code 给出一个示例：

```
{
    my $nullfh = Apache::gensym( );
    open $nullfh, '>/dev/null' or die "Can't open /dev/null: $!";
    local *STDOUT = $nullfh;
    call_something_thats_way_too_verbose( );
    close $nullfh;
}
```

这个 code 定义了一个块，在块里 STDOUT 的输出被临时重定向到/dev/null。当控制走出这个块时，STDOUT 还原到以前的值。

STDERR 被绑定到 ErrorLog 指令定义的文件。若激活了 native syslog 支持，则 STDERR 输出会重定向到/dev/null。

4.5 重定向 STDOUT 到一个标量

有时候你会遇到某个"黑匣"函数，它打印输出到默认的文件句柄（通常是 STDOUT），然而你宁愿它将输出放到一个标量里。这点在 mod_perl 下可能尤其需要，因为 STDOUT 被绑定到了 Apache 的请求目标。在此形势下，IO::String 包特别有用。你可以通过对 IO::String 目标执行一个简单的 select()操作，重新绑定 STDOUT（或任何其他文件句柄）到一个串。在 code 尾部再次对原始文件句柄调用 select()，就可改变 STDOUT 到它默认的输出流：

```
my $str;
my $str_fh = IO::String->new($str);

my $old_fh = select($str_fh);
black_box_print( );
select($old_fh) if defined $old_fh;
```

在该示例里，创建了新的 IO::String 目标。然后选择该目录，并调用 black_box_print()函数，这时它的输出会放到目标串里。最后，通过重新 select()旧的句柄，恢复了原始文件句柄。$str 变量包含了所有 black_box_print()函数的输出。

4.6 print()

在 mod_perl 下，CORE::print()（使用 STDOUT 作为文件句柄参数或者根本没有参数）将其输出重定向到 Apache::print()，因为 STDOUT 文件句柄被绑定到了 Apache。也就是说，如下 2 行功能相等：

```
print "Hello";
$r->print("Hello");
```

假如$r->connection->aborted 返回真，Apache::print()会立即返回，不打印任何东西。若客户端中断了连接（例如按下浏览器的"停止"按钮），这点就会发生。

Apache::print()还有另一个优化：任何传递到该函数的标量引用，会被自动解引用。这避免了在传递大字串给函数时，无用的 copy 开销。例如，下述 code 会打印$long_string 的实际值：

```
my $long_string = "A" x 10000000;
$r->print(\$long_string);
```

若要打印引用自身，请使用双引用：

$r->print(\\$long_string);

当 Apache::print()见到传进来的值是引用，它仅解引用它一次，并打印真正的引用值：

SCALAR(0x8576e0c)

4.7 格式化输出

用 perl 的 tie()函数把文件句柄链接到变量，访问这个句柄的接口当前并不完整。format()和 write()函数不可用。假如你配置 perl 激活 sfio，那 write()和 format()会工作完好。

代替 format()，可以用 printf()。例如，如下格式是相等的：

```
format      printf
---------------
##.##       %2.2f
####.##     %4.2f
```

打印固定长度的字串，使用 printf()格式%n.ms，这里 n 是分配给字串的域宽，m 是从字串里取的最大数量的字符。例如：

```
printf "[%5.3s][%10.10s][%30.30s]\n",
        12345, "John Doe", "1234 Abbey Road"
```

打印：

```
[   123][    John Doe][                    1234 Abbey Road]
```

注意第一个串在 output 里分配了 5 个字符的域宽，但仅仅用了 3 个，因为 m=5 和 n=3(%5.3s)。假若想确保文本总是完整的打印而没有截断，n 应总是大于或等于 m。

在%后加一个"-"可让文本左对齐，例如：

```
printf "[%-5.5s][%-10.10s][%-30.30s]\n",
        123, "John Doe", "1234 Abbey Road"
```

打印：

```
[123  ][John Doe   ][1234 Abbey Road                ]
```

也可使用+号来强制右对其，例如：

```
printf "[%+5s][%+10s][%+30s]\n",
```

123, "John Doe", "1234 Abbey Road"

打印：

[123][John Doe][1234 Abbey Road]

另一种对 format()和 printf()的代替方法是使用 CPAN 的 Text::Reform 模块。

在上述示例里我们将数字 123 作为字符串打印（%s 格式），但数字能以数字格式打印，请见 perldoc -f sprintf 的完整细节。

4.8 系统调用的输出

system()，exec()，和 open(PIPE,"|program")调用的输出不会送到浏览器，除非 perl 配置成激活 sfio。为了知道你的 Perl 版本是否激活了 sfio，请观察 perl -V 的输出，并留意 useperlio 和 d_sfio 字串。

可以这样打印系统调用的输出：

print `command here`;

但该方法效率很低，因为它 fork 了一个新的进程。

4.9 BEGIN 块

perl 在编译阶段尽可能快的执行 BEGIN 块。这点在 mod_perl 下同样如此。然而，既然 mod_perl 正常只编译脚本和模块一次，不管是在父进程还是每个子进程，BEGIN 块只运行一次。perlmod 的 manpage 解释，一旦 BEGIN 块运行完，它立刻变成 undef。在 mod_perl 环境下，这意味着 BEGIN 块不会在响应客户请求期间被运行，除非请求正好导致了代码编译。然而，有些情况下 BEGIN 块会在每个请求进来时重新运行。

模块和文件里的 BEGIN 块，它们通过 require()或 use()引入时会如下执行：

1）只运行一次，如果被父进程引入；
2）每个子进程运行一次，如果没有被父进程引入；
3）每个子进程另外运行一次，如果该模块被 Apache::StatINC 从磁盘 reload；
4）父进程在每次 restart 时另外运行一次，如果 PerlFreshRestart 设为 On；
5）每个请求里，若带有 BEGIN 的模块从%INC 里删除了，这样模块就必须重编译。同样的情况是 do()调用，它强制 reload 模块。

在 Apache::Registry 脚本里的 BEGIN 块如下执行：

1）只运行一次，如果被父进程通过 Apache::RegistryLoader 引入；

2）每个子进程运行一次，如果没有被父进程引入；
3）每个子进程另外运行一次，若这个文件在磁盘上改动了；
4）父进程在每次 restart 时另外运行一次，假如它被父进程通过 Apache::RegistryLoader 引入，并且 PerlFreshRestart 是 On。

注意第二种情况仅适用于脚本自身，第一种情况是对脚本使用的模块的。

4.10 END 块

perlmod 的 manpage 解释，END 子函数在 perl 解析器退出时执行。在 mod_perl 环境下，perl 解析器仅当子进程退出时才退出。通常一个子进程在退出前会服务许多请求，所以 END 块不能使用，也不要期望它会在每个请求处理的末尾做一些事。

假如有此必要，需要在请求处理完后运行某些代码，可使用$r->register_cleanup()函数。该函数接受一个指向某个函数的引用为参数，后者在 PerlCleanupHandler 阶段被调用，这个行为就类似于正常的 perl 环境下的 END 块。例如：

```
$r->register_cleanup(sub { warn "$$ does cleanup\n" });
```

或：

```
sub cleanup { warn "$$ does cleanup\n" };
$r->register_cleanup(\&cleanup);
```

会在每个请求的末尾运行 registered 代码，类似于 mod_cgi 下的 END 块。

现在你已知道，Apache::Registry 处理事情不同。它在每个请求末尾编译 Apache::Registry 脚本的过程中，确实执行所有遇到的 END 块，类似于 mod_cgi 做的。这包括任何由 use()引入的 package 里定义的 END 块。

假如在父进程关闭或重启的过程中，想运行某些 code 仅一次，可在 startup.pl 里使用 register_cleanup()：

```
warn "parent pid is $$\n";
Apache->server->register_cleanup(
    sub { warn "server cleanup in $$\n" });
```

在服务停止或重启时，若希望执行一些服务级的清理，那这点就有用。

5 CHECK 和 INIT 块

CHECK 和 INIT 块在编译完成后，程序开始执行前运行。CHECK 可以意味着"检查点"，"

复查"，或者甚至是"终止"。INIT 代表"初始化"。它们的区别很微妙：CHECK 块在编译完成时执行，而 INIT 块在运行时间开始时执行（这样，-c 命令行标记只运行到 CHECK 块，而不到 INIT 块）。

perl 仅在 perl_parse()阶段调用这些块，这样 mod_perl 只在启动时调用它们一次。因此，CHECK 和 INIT 块不能工作在 mod_perl 下，同理如下也不能工作：

```
panic% perl -e 'eval qq(CHECK { print "ok\n" })'
panic% perl -e 'eval qq(INIT  { print "ok\n" })'
```

5.1 $^T 和 time()

在 mod_perl 下，进程在处理完某个请求后不会退出。这样$^T 在服务启动时被初始化，并保留其值一直到进程终止。即使你不直接使用这个变量，那也该知道 perl 内部会引用$^T 的值。

例如，perl 在-M,-C 或-A 文件测试操作时，使用$^T。结果是，在子进程启动后创建的文件会显示为负年龄。-M 返回脚本文件相对于$^T 变量值的年龄。

假如想要-M 报告文件相对于当前请求的年龄，请复位$^T，跟其他 perl 脚本里的做法一样。增加下列行在脚本的开始处：

```
local $^T = time;
```

也可以这样：

```
local $^T = $r->request_time;
```

第 2 种技术性能更好，因为它跳过了 time()系统调用，并且使用了请求开始时的时间戳（通过$r->request_time 方法获取）。

假如这个修正要用于大量处理句柄上，那么可指定一个 fixup 处理句柄，它在 fixup 阶段被执行：

```
sub Apache::PerlBaseTime::handler {
    $^T = shift->request_time;
    return Apache::Constants::DECLINED;
}
```

然后增加下列行到 httpd.conf：

```
PerlFixupHandler Apache::PerlBaseTime
```

现在不必修改脚本内容了。

5.2 命令行开关

当某个 perl 脚本从命令行运行时，shell 通过脚本第一行的#!/bin/perl 指令调用 perl 解析器。对运行在 mod_cgi 下的脚本，你可能会使用 perlrun 里描述的 perl 开关，例如-w,-T,或-d。在 Apache::Registry 家族，所有开关除了-w 都被忽略（使用-T 会触发警告）。支持-w，是为了保持对 mod_cgi 的向后兼容性。

大部分命令行开关有特殊的 perl 变量对应，它们可在代码里 set 和 unset。参考 perlvar 的更多细节。

mod_perl 提供它自己的对应于-w 和-T 的配置指令，随后讨论。

最后，假如仍需要设置其他的 perl 启动标识，例如-d 和-D，可使用 PERL5OPT 环境变量。在这个变量里的开关在每个 perl 命令行里生效。参考 perlrun 的 manpage，仅仅-[DIMUdmw]开关可用。

5.2.1 Warnings

有 3 种方法激活 warnings：

1．对所有进程的全局设置：

在 httpd.conf 里，设置：
PerlWarn On

然后可调整代码，通过在脚本里设置$^W 变量来关闭或打开 warnings。

2．仅在当前脚本里设置：

包含下列行：
#!/usr/bin/perl -w

会在当前脚本范围内打开 warnings。如前所述，也可通过在脚本里设置$^W 变量，来关闭或打开 warnings。

3．仅在块里设置：

如下 code 仅在当前块范围内打开 warnings：

```
{
    local $^W = 1;
    # some code
}
```

```
# $^W assumes its previous value here
```

这样关闭它：

```
{
    local $^W = 0;
    # some code
}
# $^W assumes its previous value here
```

假如$^W 没有正确的 local 化，该 code 会影响当前请求，以及由该子进程处理的所有随后请求。这样：

```
$^W = 0;
```

会关闭 warnings，而不管其他。

假如想在整个文件范围内打开 warnings，跟前面提的一样，你可以增加这行：

```
local $^W = 1;
```

在文件的开头处。既然文件也是个有效块，文件范围类似于块的封闭范围（{}），所以在文件开头处的 local $^W 会对整个文件有效。

然而，打开 warnings 模式本质上用于开发服务器，在产品服务器上，不应该全局的打开它。打开 warnings 导致不可忽略的性能开销。假如每个请求产生一条警告，并且你的服务器每天处理数百万请求，error_log 文件会吃光磁盘空间，系统会变得不正常。

Perl 5.6.x 引进了 warnings 参数，它允许对 wanrings 的更灵活控制。该参数允许你激活或禁止以组为单位的 warnings。例如，仅仅激活语法警告，可以这样写：

```
use warnings 'syntax';
```

在以后的代码里，假如想禁止语法警告，并且激活信号相关的警告，就可这样写：

```
no    warnings 'syntax';
use warnings 'signal';
```

通常你会这样写：

```
use warnings;
```

这等价于：

use warnings 'all';

假如想让 code 真正干净，将所有警告作为错误考虑，perl 可帮你做到。在下列 code 里，在词法范围内的任何警告会触发致命错误：

use warnings FATAL => 'all';

当然，可调整 warnings group，仅让特定 group 的警告变成致命错误。例如，仅让闭包问题致命，可以这样写：

use warnings FATAL => 'closure';

使用 warnings 参数，也能在本地禁止 warnings：

```
{
  no warnings;
  # some code that would normally emit warnings
}
```

在该方法里，你可避免某些不想要的警告。

关于 warnings 参数的更多信息，请参考 perllexwarn 的 manpage。

5.2.2 Taint 模式

perl 的-T 开关激活了 taint 模式。在 taint 模式下，perl 对传递给程序的数据进行检查。例如，taint 模式能阻止程序在未经过严格污染检查的情况下，传递外部数据给系统调用，这就避免了很多潜在的安全漏洞。若没有强迫脚本运行在 taint 模式下，那意味着会留一些可利用的漏洞给恶意用户。

既然-T 开关不能从 perl 脚本里打开（这是因为当 perl 已在运行时，再将所有外部数据标记为污染已太迟），mod_perl 提供了 PerlTaintCheck 指令来全局的打开 taint 检查。使用如下指令：
PerlTaintCheck On
可放在 httpd.conf 里的任何地方。

关于 taint 检查和如何去除数据的污染属性的更多信息，请参考 perlsec 的 manpage。

5.3 编译正则表达式

若使用的正则表达式包含内插变量，并且确认变量在程序执行期间不会改变，那么标准的加速技术是对正则表达式加上/o 参数。这样在脚本的整个存活期间只编译正则表达式一次，而不是每次执行 regex 时都要编译。如下示例：

```
my $pattern = '^\d+$'; # likely to be input from an HTML form field
foreach (@list) {
    print if /$pattern/o;
}
```

这在遍历列表进行循环，或进行 grep()或 map()操作时，效率会更高。

mod_perl 脚本和处理句柄的存活期长，这样，在每次调用时，变量可能会改变。在这种情况下，上述的记忆方式会导致问题。被 mod_perl 子进程处理的第一个请求，会编译正则表达式并执行正确的搜索。然而，所有随后来的请求，若在相同的进程里运行同一份 code，就会使用已编译的旧的表达式，而不是用户提供的新的。这样代码就会失效。

假设你在运行搜索引擎服务，某人键入一个搜索关键字并且获得了正确的结果。然后另一个人输入了不同的关键字，若他的搜索请求被前述同一个子进程处理，他的搜索结果就会和前面那人的一样。

解决这问题有 2 个方案。

第一个方案是使用 eval q//结构，强迫代码每次运行时都重新编译。重要的是 eval 块必须包括整个循环处理，而不仅是匹配表达式自身。

原始的代码片断可重写如下：

```
my $pattern = '^\d+$';
eval q{
    foreach (@list) {
        print if /$pattern/o;
    }
}
```

假如我们这样写：

```
foreach (@list) {
    eval q{ print if /$pattern/o; };
}
```

正则表达式将对列表里的每个元素重新编译，而不是对整个列表只编译一次，这样/o 参数就毫无用处。

然而，在 eval 内若使用来自不信任源的数据，就必须谨慎，它们可能包含危及系统的 perl 代码，所以请确保先进行安全检查。

假如只有一个正则表达式操作符（例如 m//或 s//），就可依赖空模式的特性，空模式重用最

后一个见到的模式。这就是第二个解决方案，不必使用 eval。

上述代码片断变成：

```
my $pattern = '^\d+$';
"0" =~ /$pattern/; # dummy match that must not fail!
foreach (@list) {
    print if //;
}
```

唯一警告是这个虚拟匹配必须成功，否则模式不会被缓存住，并且//会匹配任何东西。假如匹配内容是变化的，为了保证匹配成功，可有 2 个选择。

假如能保证模式变量不包含元字符（such as *, +, ^, $, \d, etc），就可用模式自身的虚拟匹配：

```
$pattern =~ /\Q$pattern\E/; # guaranteed if no metacharacters present
```

\Q 参数保证任何特殊的正则表达式字符被跳过。

假如存在模式变量包含元字符的可能性，就应该匹配模式自身或匹配“不被搜索”的\377 字符，如下：

```
"\377" =~ /$pattern|^\377$/; # guaranteed if metacharacters present
```

5.3.1 重复匹配模式

另一个技术也可用，依赖于所用正则表达式的复杂性。在重复匹配一组模式里的某一项时，通常编译过的正则表达式更有效。

为了易于理解，我们使用来自 Jeffrey Friedl 的 Mastering Regular Expressions 一书里的稍做修改的子函数：

```
sub build_match_many_function {
    my @list = @_;
    my $expr = join '||',
        map { "\$_[0] =~ m/\$list[$_]/o" } (0..$#list);
    my $matchsub = eval "sub { $expr }";
    die "Failed in building regex @list: $@" if $@;
    return $matchsub;
}
```

该子函数接受模式列表作为参数，建立一个正则表达式，这个表达式用||操作符连起来，一旦模式列表里的某项匹配成功就终止匹配。这个模式匹配链放入一个串里（$expr），然后将这个串放在匿名子函数里，并调用 eval 在运行时编译这个匿名子函数。假如 eval 失败，编

译过程会 die 掉；否则，对子函数的引用会返回给调用者。

如下显示它的用法：

```
my @agents = qw(Mozilla Lynx MSIE AmigaVoyager lwp libwww);
my $known_agent_sub = build_match_many_function(@agents);

while (<ACCESS_LOG>) {
    my $agent = get_agent_field($_);
    warn "Unknown Agent: $agent\n"
        unless $known_agent_sub->($agent);
}
```

上述 code 打开 access_log 文件到 ACCESS_LOG 句柄，从 log 文件里的每行抽取到 agent 域，并将这个 agent 与已知 agents 列表进行匹配。每次匹配失败，会将未知的 anget 名字作为警告打印出来。

另一个方法是使用 qr//操作符，它用于编译正则表达式。前述示例可被重写成：

```
my @agents = qw(Mozilla Lynx MSIE AmigaVoyager lwp libwww);
my @compiled_re = map qr/$_/, @agents;

while (<ACCESS_LOG>) {
    my $agent = get_agent_field($_);
    my $ok = 0;
    for my $re (@compiled_re) {
        $ok = 1, last if /$re/;
    }
    warn "Unknown Agent: $agent\n"
        unless $ok;
}
```

在本 code 里，在使用正则表达式模式之前编译它们一次，类似于前面示例的 build_match_many_function()函数，但现在我们节省了子函数调用的开销。简单的压力测试表明，这个示例比前面一个快 2.5 倍。

6 Apache::Registry 规范

下面列出的代码仅运行在 Apache::Registry 及类似的处理器环境，例如 Apache::PerlRun。

6.1 __END__和__DATA__标记

Apache::Registry 脚本不能包含__END__或__DATA__标记，因为 Apache::Registry 将原始脚本代码封装在一个名为 handler()的子函数里，后者被真正调用。考虑下述脚本，位于/perl/test.pl：

```
print "Content-type: text/plain\n\n";
print "Hi";
```

该脚本在 Apache::Registry 下执行时，它被封装在 handler()子函数里，如下：

```
package Apache::ROOT::perl::test_2epl;
use Apache qw(exit);
sub handler {
    print "Content-type: text/plain\n\n";
    print "Hi";
}
```

如果碰巧将__END__标记放在代码里，象这样：

```
print "Content-type: text/plain\n\n";
print "Hi";
_ _END_ _
Some text that wouldn't be normally executed
```

它就会变成：

```
package Apache::ROOT::perl::test_2epl;
use Apache qw(exit);
sub handler {
    print "Content-type: text/plain\n\n";
    print "Hi";
    _ _END_ _
    Some text that wouldn't be normally executed
}
```

当发布请求到/perl/test.pl 时，会发生如下错误：

```
Missing right bracket at .... line 4, at end of line
```

perl 将__END__后的任何代码 cut 掉，所以，子函数 handler()的右封闭花括号就丢掉了。__DATA__标记同样如此。

6.2 符号链接

Apache::Registry 将脚本缓存在包里，包名由脚本的访问 url 组成。假如同一个脚本可通过不通的 url 访问，那很可能你使用了符号链接或别名。同一个脚本存放在内存里不止一次，纯属浪费。

例如，假如已有一个脚本为/home/httpd/perl/news/news.pl，可对它创建符号链接：

panic% ln -s /home/httpd/perl/news/news.pl /home/httpd/perl/news.pl

现在脚本可通过 2 个 url 来访问：/news/news.pl 和/news.pl。如果 2 个 url 都发布了，用户可通过 2 者来访问同一脚本，就会有问题。

现在以单进程方式启动 web 服务器，发布请求到 2 个 url：

http://localhost/perl/news/news.pl
http://localhost/perl/news.pl

为了呈现问题，请使用 Apache::Status 模块，它可显示所有编译过的 Apache::Registry 脚本（用它们各自的包）。假如使用默认的配置指令，就可访问这个 url：

http://localhost/perl-status?rgysubs

或访问它的主菜单：

http://localhost/perl-status

并点击"Compiled Registry Scripts"按钮。

假如通过 2 个 url 访问了脚本，可见到如图所示的输出。

Figure: Compiled Registry Scripts output

可运行一个链接检查程序，通过跟踪链接，来递归遍历 web 服务的所有页面目录，然后使用 Apache::Status 来发现符号链接的复本（不用重启 web 服务）。为了弄明白需要找什么，请先找到所有的符号链接。例如，在本示例里，下述命令显示我们仅有一个符号链接：

```
panic% find /home/httpd/perl -type l
/home/httpd/perl/news.pl
```

现在我们能从"Compiled Registry Scripts"的输出里查找该符号链接。

注意假如在多进程服务的模式下执行该测试，某些子进程可能只显示一个访问 url 或根本没显示，因为它们不一定会服务那个带符号链接的脚本请求。

6.3 返回 Code

Apache::Registry 正常情况下假设返回 code 是 OK(200)，并将它发送给你。假如需要返回不同的 code，可使用$r->status()。例如，为了返回 code 404(Not Found)，可使用如下代码：

```
use Apache::Constants qw(NOT_FOUND);
$r->status(NOT_FOUND);
```

假如使用了该方法，就没必要调用$r->send_http_header()（假如设置了 PerlSendHeader Off）。

7 从 mod_cgi 脚本转换到 Apache 处理器

假如你不必保持对 mod_cgi 的向后兼容性，就可转换 mod_cgi 脚本使用 mod_perl 规范的 API。这样可让你从 mod_cgi 没有的功能中受益，并在实现同样功能的前提下，给你更好的性能。我们已经见到，Apache::Registry 把脚本转换到 Apache 处理器是多么简单。大多数情况下，这种转换很直观。

让我们看一个转换示例。下述脚本是一个 mod_cgi 兼容的脚本，运行在 Apache::Registry 下，先转换它到 Perl 内容处理器而不使用任何 mod_perl 规范的模块，然后再转换它使用 Apache::Request 和 Apache::Cookie 模块，它们仅在 mod_perl 环境下可用。

7.1 从 mod_cgi 兼容的脚本开始

如下展示原始的脚本代码：

Example 6-18. cookie_script.pl

```
use strict;
```

```
use CGI;
use CGI::Cookie;
use vars qw($q $switch $status $sessionID);

init( );
print_header( );
print_status( );

sub init {
    $q = new CGI;
    $switch = $q->param("switch") ? 1 : 0;
    my %cookies = CGI::Cookie->fetch;
    $sessionID = exists $cookies{'sessionID'}
        ? $cookies{'sessionID'}->value
        : '';

    # 0 = not running, 1 = running
    $status = $sessionID ? 1 : 0;
    # switch status if asked to
    $status = !$status if $switch;

    if ($status) {
        # preserve sessionID if it exists or create a new one
        $sessionID ||= generate_sessionID( ) if $status;
    } else {
        # delete the sessionID
        $sessionID = '';
    }
}

sub print_header {
    my $c = CGI::Cookie->new(
        -name    => 'sessionID',
        -value   => $sessionID,
        -expires => '+1h'
    );

    print $q->header(
        -type    => 'text/html',
        -cookie => $c
    );
}

# print the current Session status and a form to toggle the status
```

```
sub print_status {

    print qq{<html><head><title>Cookie</title></head><body>};

    print "<B>Status:</B> ",
        $status
          ? "Session is running with ID: $sessionID"
          : "No session is running";


    # change status form
    my $button_label = $status ? "Stop" : "Start";
    print qq{<hr>
        <form>
          <input type=submit name=switch value=" $button_label ">
        </form>
              };

    print qq{</body></html>};

}

# A dummy ID generator
# Replace with a real session ID generator
########################
sub generate_sessionID {
    return scalar localtime;
}
```

该代码相当简单。当按下 Start 按钮时，它产生一个 session；当按下 Stop 按钮时，它删除这个 session。该 session 使用 cookie 来获取和存储。

上述 code 拆分成 3 个子函数。init()初始化全局变量，并解析进来的数据。print_header()打印 HTTP 头部，包括 cookie 头部。最后，print_status()产生输出。随后，我们会看到这种逻辑分离的代码，容易转换成 perl 内容处理器。

我们使用了一些全局变量，因为不想把它们在函数之间传来传去。在大的项目里，应该非常严谨的使用全局变量。在任何情形下，init()子函数确保所有的变量被重新初始化。

这里使用了一个非常简单的 generate_sessionID()函数，它返回当前时间串作为 session ID。在正式应用的情况下，你该替换这个函数，以确保它能产生唯一的不可预知的 session ID。

7.2 转换到 Perl 内容处理器

现在转换这个脚本到内容处理器。该任务包括 2 部分：首先配置 Apache 以 Perl 处理器方式运行新代码，然后修改代码自身。

首先增加下列片断到 httpd.conf：

```
PerlModule Book::Cookie
<Location /test/cookie>
    SetHandler perl-script
    PerlHandler Book::Cookie
</Location>
```

然后重启服务器。

当某个请求（它的 URI 以/test/cookie 开始）进来时，Apache 执行 Book::Cookie::handler()子函数（随后会见到）作为内容处理器。我们用 PerlModule 指令，确保在服务启动时预装载 Book::Cookie 模块。

再修改脚本自身。我们 copy 它的内容到 Cookie.pm，并将这个 pm 文件放到@INC 中的目录下。在本示例里，使用/home/httpd/perl，这个目录已增加到@INC。因为要以 Book::Cookie 的名字调用这个包，那就将 Cookie.pm 放到 /home/httpd/perl/Book/目录中。

更改后的代码如下。因为没有修改子函数的代码，所以不在这里展示它们，这样可更清楚的见到差别。

Example 6-19. Book/Cookie.pm

```
package Book::Cookie;
use Apache::Constants qw(:common);

use strict;
use CGI;
use CGI::Cookie;
use vars qw($q $switch $status $sessionID);

sub handler {
    my $r = shift;

    init( );
    print_header( );
    print_status( );

    return OK;
}
```

```
# all subroutines unchanged

1;
```

在 code 的开头处增加了 2 行：

```
package Book::Cookie;
use Apache::Constants qw(:common);
```

第一行声明包名，第二行 import 进 mod_perl 通用的常量，用来返回状态码。在本示例里，在从 handler()子函数返回时，仅使用了 OK 常量。

下述 code 未改变：

```
use strict;
use CGI;
use CGI::Cookie;
use vars qw($q $switch $status $sessionID);
```

围绕着子函数调用，我们增加了一些新的 code：

```
sub handler {
    my $r = shift;

    init( );
    print_header( );
    print_status( );

    return OK;
}
```

每个内容处理器（以及任何其他的处理器）应该以名为 handler()的子函数开始。当请求的 URI 以/test/cookie 开始时，这个子函数被调用。也可选用不同的子函数名--例如，execute()--但这时你必须明确的在配置指令里指定函数名，如下：

```
PerlModule Book::Cookie
<Location /test/cookie>
    SetHandler perl-script
    PerlHandler Book::Cookie::execute
</Location>
```

这里使用默认名字，handler()。

handler()子函数类似任何其他子函数，但通常它有如下结构：

```
sub handler {
    my $r = shift;

    # the code

    # status (OK, DECLINED or else)
    return OK;
}
```

首先，通过从@_里 shift 的方式，获取一个对请求目标的引用，并将它分配给$r 变量。我们随后需要它。

其次，编写代码处理请求。

第三，返回执行状态。有许多可能的状态，最常用的是 OK 和 DECLINED。OK 告诉服务器，处理器已完成了请求。DECLINED 意味着相反，在这种情形下其他处理器会处理这个请求。Apache::Constants 导出这些通用状态码的常量。

在本示例里，我们要做的事是将这 3 个子函数调用：

```
init( );
print_header( );
print_status( );
```

封装在 handler()结构内：

```
sub handler {
    my $r = shift;

    return OK;
}
```

最后，必须在模块的尾部增加'1;'，这点跟其他 perl 模块的做法一样，确保在装载 Book::Cookie 时不至于失败。

总的来说，我们拿来原始的脚本代码，并增加了如下七行：

```
package Book::Cookie;
use Apache::Constants qw(:common);

sub handler {
    my $r = shift;
```

```
    return OK;
}
1;
```

现在就拥有了一个完整的 Perl 内容处理器。

7.3 转换到使用 mod_perl API 和 mod_perl 专有模块

现在已有了一个完整的 PerlHandler，我们转换它使用 mod_perl API 和 mod_perl 专有模块。首先，这给予我们更好的性能，API 内部是用 C 执行的。其次，这样做释放了 mod_perl API 驱动的 Apache 的全部力量，而在 mod_cgi 兼容的模块下，这些力量只部分可用。

我们准备替换 CGI.pm 和 CGI::Cookie 成 mod_perl 专有的替代者：Apache::Request 和 Apache::Cookie。这 2 个模块用 C 写成，提供 XS 接口给 Perl，所以使用这些模块的 code 运行更快。

Apache::Request 有个类似于 CGI 的 API，Apache::Cookie 也有个类似于 CGI::Cookie 的 API。这就让转换更直观。本质上，我们仅替换：

```
use CGI;
$q = new CGI;
```

成：

```
use Apache::Request ( );
$q = Apache::Request->new($r);
```

以及替换：

```
use CGI::Cookie ( );
my $cookie = CGI::Cookie->new(...)
```

成：

```
use Apache::Cookie ( );
my $cookie = Apache::Cookie->new($r, ...);
```

如下是 Book::Cookie2 的新代码。

Example 6-20. Book/Cookie2.pm

```
package Book::Cookie2;
use Apache::Constants qw(:common);

use strict;
use Apache::Request ( );
use Apache::Cookie ( );
```

```
use vars qw($r $q $switch $status $sessionID);

sub handler {
    $r = shift;

    init( );
    print_header( );
    print_status( );

    return OK;
}

sub init {

    $q = Apache::Request->new($r);
    $switch = $q->param("switch") ? 1 : 0;

    my %cookies = Apache::Cookie->fetch;
    $sessionID = exists $cookies{'sessionID'}
        ? $cookies{'sessionID'}->value : '';

    # 0 = not running, 1 = running
    $status = $sessionID ? 1 : 0;
    # switch status if asked to
    $status = !$status if $switch;

    if ($status) {
        # preserve sessionID if it exists or create a new one
        $sessionID ||= generate_sessionID( ) if $status;
    } else {
        # delete the sessionID
        $sessionID = '';
    }
}

sub print_header {
    my $c = Apache::Cookie->new(
          $r,
          -name       => 'sessionID',
          -value      => $sessionID,
          -expires => '+1h');

    # Add a Set-Cookie header to the outgoing headers table
    $c->bake;
```

```
    $r->send_http_header('text/html');
}

# print the current Session status and a form to toggle the status
sub print_status {

    print qq{<html><head><title>Cookie</title></head><body>};

    print "<B>Status:</B> ",
        $status
            ? "Session is running with ID: $sessionID"
            : "No session is running";

    # change status form
    my $button_label = $status ? "Stop" : "Start";
    print qq{<hr>
        <form>
          <input type=submit name=switch value=" $button_label ">
        </form>
            };

    print qq{</body></html>};

}

# replace with a real session ID generator
sub generate_sessionID {
    return scalar localtime;
}

1;
```

其他唯一改变的地方是 print_header()函数。以前是将 cookie 代码传递给 CGI 的 header()函数，由它返回一个相应的 HTTP 头部，象这样：

```
print $q->header(
    -type    => 'text/html',
    -cookie => $c);
```

现在分 2 步完成它。首先，下列行增加一个 Set-Cookie 头部到外出的 HTTP 头部里：

```
$c->bake;
```

然后该行设置 Content-Type 头部为 text/html，并送出整个 HTTP 头部：

```
$r->send_http_header('text/html');
```

代码的其他部分未改变。

要做的最后事情是增加下述片断到 httpd.conf:

```
PerlModule Book::Cookie2
<Location /test/cookie2>
    SetHandler perl-script
    PerlHandler Book::Cookie2
</Location>
```

现在请求上述 code 的 URI 将以/test/cookie2 开头。这里将代码存储在/home/httpd/perl/Book/Cookie2.pm 文件里，既然声明了该包为 Book::Cookie2。

你已看到，将编写良好的 CGI 代码转换到 mod_perl 处理器代码的过程很直观。利用 mod_perl 的专有功能和模块也通常很简单。在转换脚本时，要做的事很少。

注意为了让示例前后一致，我们没有改变原始包的风格。但在做真正的代码转换时，请考虑如下事项：使用词法变量代替全局变量，尽可能的使用 mod_perl API 函数，等等。

8 load 和 reload 模块

在开发和产品环境中，都经常需要 reload 模块。mod_perl 尽量避免无必要的模块 reload，但有时候（特别是在开发阶段），我们在修改模块后想要 reload 它们。下面讨论模块的 load 和 reload 问题。

8.1 mod_perl 下的@INC 数组

在 mod_perl 下，@INC 仅在服务启动时可修改它。在每个请求后，mod_perl 复位@INC 的值到它的初始值。

假如 mod_perl 遇到下述陈述：

```
use lib qw(foo/bar);
```

它仅在代码的解析和编译阶段修改@INC。在那之后，@INC 被复位到原始值。这样，永久的改变@INC 的唯一办法是在服务启动时修改它。

在服务启动时改变@INC 有 2 个办法：

1）在配置文件里，写入：
PerlSetEnv PERL5LIB /home/httpd/perl

或者：
PerlSetEnv PERL5LIB /home/httpd/perl:/home/httpd/mymodules

2）在 startup.pl 文件里：
use lib qw(/home/httpd/perl /home/httpd/mymodules);
1;

当然，startup 文件要从 httpd.conf 装载：
PerlRequire /path/to/startup.pl

为了确保正确的设置了@INC，请在服务器里配置 perl-status（在原书的 21 章里有讲到）。点主菜单里的"Loaded Modules"，在产生的页面底部，会显示@INC 的内容：

@INC =
/home/httpd/mymodules
/home/httpd/perl
/usr/lib/perl5/5.6.1/i386-linux
/usr/lib/perl5/5.6.1
/usr/lib/perl5/site_perl/5.6.1/i386-linux
/usr/lib/perl5/site_perl/5.6.1
/usr/lib/perl5/site_perl
.
/home/httpd/httpd_perl/
/home/httpd/httpd_perl/lib/perl

在本处设置中，有 2 个定制的目录放到这个数组里的开头位置。剩余的目录是 perl 发布版的标准目录，以及 $ServerRoot 和$ServerRoot/lib/perl 这 2 个 mod_perl 的系统目录（mod_perl 自动增加）。

8.2 reload 模块和文件（require 进来的）

当运行在 mod_cgi 下时，你可改变代码，并从浏览器重新运行脚本来观察效果。既然脚本不 cache 在内存里，服务器对每个请求都启动一个新的 perl 解析器，它装载和重编译脚本。这样任何改变都立杆见影。

在 mod_perl 下，形势完全不同，因为 mod_perl 的根本目的就是取得最大性能。默认情况下，服务器不会花时间去检查任何包含的库或模块是否已改变。它假设它们不会改变，从而节省了对脚本里引入的模块或库的源文件进行 stat()调用的时间。

假如脚本运行在 Apache::Registry 下，唯一执行的检查是看主脚本是否改变。假如脚本没有

use()或 require()任何其他的 perl 模块或包，就没什么可担心的。然而，假如你在开发的脚本包含了其他模块，你 use()或 require()的文件就不会被检查是否已修改，这样你需要针对这个问题做一些事。

有一些技术，可让装配了 mod_perl 的服务器察觉到库或模块的改变。在下面的节里将讨论它们。

8.2.1 重启服务

最简单的方法是，在每次改变了代码后，重启服务。然而在重启了 50 次服务后，你会觉得厌倦，并去找其他方法。

8.2.2 使用 Apache::StatINC

当 perl 使用 require()引入文件时，它在全局变量%INC 里存储这个文件名作为哈希 key，哈希 value 是文件的完整路径。Apache::StatINC 遍历%INC，并立刻 reload 已在磁盘上更新的任何文件。

为了激活这个模块，在 httpd.conf 里增加 2 行：

```
PerlModule Apache::StatINC
PerlInitHandler Apache::StatINC
```

为了保证它工作，在开发系统上打开 debug 模式，增加 PerlSetVar StatINCDebug On 到配置文件即可。配置看起来如下：

```
PerlModule Apache::StatINC
PerlInitHandler Apache::StatINC
<Location /perl>
    SetHandler perl-script
    PerlHandler Apache::Registry
    Options ExecCGI
    PerlSendHeader On
    PerlSetVar StatINCDebug On
</Location>
```

请记住仅仅位于@INC 里的模块会在改变后被 reload，并且只能在服务启动时改变@INC（在 startup 文件里）。

另外要注意的是："."代表当前目录，它位于@INC 里，当文件的路径名是当前脚本的相对目录时，perl 知道如何去 require()它。然而，在 code 被解析后，服务器不会记住这个路径。所以假如代码装载的模块 MyModule 位于脚本的目录，并且该目录不在@INC 里，就可在%INC 里见到下列条目：

'MyModule.pm' => 'MyModule.pm'

当 Apache::StatINC 试图去检查是否文件已修改时，它找不到这个文件，既然 MyModule.pm 不位于@INC 的任何路径中。为了纠正这个问题，请在@INC 里增加该模块的路径。

8.2.3 使用 Apache::Reload

Apache::Reload 是一个更新的模块，它有代替 Apache::StatINC 的趋势。它提供功能多一些，并且灵活性更好。

为了让 Apache::Reload 在每个请求里检查所有载入的模块，仅需增加下列行到 httpd.conf：

PerlInitHandler Apache::Reload

若只 reload 指定的模块（在它们改变时），可有 3 个选择：隐式登记模块，显式登记模块，创建虚拟文件动态的 reload 模块。

隐式登记模块时，先关闭 ReloadALL 变量，它默认打开。

PerlInitHandler Apache::Reload
PerlSetVar ReloadAll Off

增加下列行到想要 reload 的模块：

use Apache::Reload;

另外，显式登记模块的方法是，在 httpd.conf 里：

PerlInitHandler Apache::Reload
PerlSetVar ReloadModules "Book::Foo Book::Bar Foo::Bar::Test"

注意它们以空格分开，且模块列表必须被双引起来，否则 Apache 会试图去解析参数列表自身。

也可用通配符*来登记模块组：

PerlSetVar ReloadModules "Foo::* Bar::*"

在上述示例里，所有以 Foo::和 Bar::开头的模块会被登记。这样可让某一项目里的所有模块都被登记上。

第 3 种方式是用 touch 命令创建一个虚拟文件，并触发 reload 的执行：

PerlSetVar ReloadTouchFile /tmp/reload_modules

当想要 reload 时，简单的在命令行敲入：

panic% touch /tmp/reload_modules

假如设置了这个，并且没有 touch 文件，则 reload 不会发生，而不管模块是怎样被登记的。

在产品服务器环境里，这个功能非常便利。但对比完全的 restart，预装载模块内存共享的优势就失去了，因为每个子进程会得到它自己的 reload 模块的内存拷贝。

注意若 Apache::Reload 要 reload 的单个模块，它包含了多个使用 pseudo-hashes 的包时，会有问题。解决方法是：不要使用 pseudo-hashes。pseudo-hashes 会在新版本的 perl 里移除掉。

跟 Apache::StatInc 类似，假如你装载的模块的目录没有位于@INC 里，Apache::Reload 不会找到这些文件。这是因为@INC 被 reset 到其初始值，尽管它在脚本编译阶段临时改变过。解决方法是在服务启动时扩展@INC，让它包含所有要装载的模块目录。

8.3 使用动态配置文件

有时候想要应用程序去监视它自己的配置文件，并在配置文件改变时 reload 它，但并不想重启服务让配置生效。解决方法就是使用动态配置文件。

如果想给管理员提供一个配置工具，允许在线修改应用，动态配置文件就特别有用。这种方法消除了提供 shell 访问服务器的必要。另外，管理程序也可验证提交上来的修改。

也可使用 Apache::Reload，仍会有少量的 stat()调用的负载。但这样做要求配置者有权限修改 httpd.conf 来配置 Apache::Reload。下面描述的方法无此必要。

8.3.1 编写配置文件

下面描述不同的配置文件编写方法，包括其长处和弱处。

假如配置文件包含的变量不多，那怎样写这个文件都无所谓。然而在实际中，配置文件经常随着项目开发而增长，特别是对那些产生 HTML 文件的项目尤其如此，因为站点 SP 总是声称他们的配置多么容易，例如页眉页脚，模板，颜色等。

CGI 程序员惯用的方法是在一个独立的文件里定义所有配置变量。例如：

```
$cgi_dir  = '/home/httpd/perl';
$cgi_url  = '/perl';
$docs_dir = '/home/httpd/docs';
$docs_url = '/';
```

```
$img_dir    = '/home/httpd/docs/images';
$img_url    = '/images';
# ... many more config params here ...
$color_hint     = '#777777';
$color_warn     = '#990066';
$color_normal = '#000000';
```

use strict;参数要求所有变量预先声明。当在 mod_perl 下使用这些变量时，必须以 use vars 来声明它们。脚本开始如下：

```
use strict;
use vars qw($cgi_dir $cgi_url $docs_dir $docs_url
            # ... many more config params here ....
            $color_hint    $color_warn $color_normal
           );
```

维护如此一个脚本是个噩梦，特别是假如并非所有属性都可硬编码--我们必须能增加或移除变量名。出于 clean code 的目的，我们以 use strict 开头来编写配置文件；所以必须以 use vars 来列出变量--一个要维护的变量列表。这样，随着编写的脚本越来越多，可能会导致配置文件之间的名字冲突。

解决方法是使用 Perl 的包定义，分配一个唯一包名给每个配置文件。例如，可以声明下列包名：

```
package Book::Config0;
```

现在每个配置文件被限制在它自己的命名空间里。脚本如何使用这些变量呢？我们不能再简单的 require()文件和使用变量，因为它们现在已属于不同的包。代替的，必须修改所有的脚本，使用完全包名限定的配置变量（例如，使用$Book::Config0::cgi_url 而不是$cgi_url）。

你会发现敲入完全包名限定的变量很乏味，而且升级其他的程序也麻烦。如果这样，就可在脚本里 import 进需要的变量，并使用它们。首先，配置包必须 export 出这些变量。见如下示例：

Example 6-21. Book/Config0.pm

```
package Book::Config0;
use strict;

BEGIN {
  use Exporter ( );

  @Book::HTML::ISA          = qw(Exporter);
  @Book::HTML::EXPORT       = qw( );
```

```
   @Book::HTML::EXPORT_OK = qw($cgi_dir $cgi_url $docs_dir $docs_url
                               # ... many more config params here ....
                               $color_hint $color_warn $color_normal);
}

use vars qw($cgi_dir $cgi_url $docs_dir $docs_url
            # ... many more config params here ....
            $color_hint    $color_warn $color_normal
           );

$cgi_dir    = '/home/httpd/perl';
$cgi_url    = '/perl';
$docs_dir = '/home/httpd/docs';
$docs_url = '/';
$img_dir    = '/home/httpd/docs/images';
$img_url    = '/images';
# ... many more config params here ...
$color_hint      = "#777777';
$color_warn      = "#990066';
$color_normal = "#000000';
```

使用该包的脚本会这样开头：

```
use strict;
use Book::Config0 qw($cgi_dir $cgi_url $docs_dir $docs_url
                     # ... many more config params here ....
                     $color_hint    $color_warn $color_normal
                   );
use vars          qw($cgi_dir $cgi_url $docs_dir $docs_url
                     # ... many more config params here ....
                     $color_hint    $color_warn $color_normal
                   );
```

然而，假如我们改变了配置变量的命名，就必须升级至少三个变量列表。这里虽只有一个脚本用到了这个配置文件，但在真正的产品环境里会有许多不同的脚本。

这里也有个性能弊端：export 变量增加了内存开销，在 mod_perl 环境里，这个开销会因为进程的数量而成倍扩大。

有一些技术可消除这些问题。首先，变量可被分组成命名组，叫做 tags。tags 随后用于 import() 或 use()调用的参数。你也许熟悉这个语法：

```
use CGI qw(:standard :html);
```

使用 Exporter 的 export_ok_tags()，可相当容易的做到这点。例如：

```
BEGIN {
   use Exporter ( );
   use vars qw( @ISA @EXPORT @EXPORT_OK %EXPORT_TAGS );
   @ISA                = qw(Exporter);
   @EXPORT             = ( );
   @EXPORT_OK          = ( );

   %EXPORT_TAGS = (
         vars => [qw($firstname $surname)],
         subs => [qw(reread_conf untaint_path)],
   );
   Exporter::export_ok_tags('vars');
   Exporter::export_ok_tags('subs');
}
```

在使用这个配置的脚本里，如下写：

```
use Book::Config0 qw(:subs :vars);
```

子函数正如变量一般被 export 出来，因为符号才是实际被 export 出的东西。注意我们没有使用 export_tags()，因为它自动 export 出用户没有请求的变量（这被认为是不好的编程风格）。假如某个模块使用 export_tags()自动 export 出变量，就可在脚本里用如下语法避免无必要的 import：

```
use Book::Config0 ( );
```

也可将组 tags 嵌入其他的命名组。例如，来自 CGI.pm 的:all tag 是包含所有其他组的组 tag。它的实现稍微麻烦点，但可节省你查看 CGI.pm 代码寻求某个特定解决方法的时间。

然而，随着变量数量的增长，配置文件会变得很笨拙。请考虑将所有的变量放在一个单一的 hash 结构里，hash 里包含了对其他标量，匿名数组，和 hash 的引用。见如下示例：

Example 6-22. Book/Config1.pm

```
package Book::Config1;
use strict;

BEGIN {
   use Exporter ( );

   @Book::Config1::ISA         = qw(Exporter);
   @Book::Config1::EXPORT      = qw( );
```

```
   @Book::Config1::EXPORT_OK = qw(%c);
}

use vars qw(%c);

%c = (
    dir => {
          cgi    => '/home/httpd/perl',
          docs => '/home/httpd/docs',
          img    => '/home/httpd/docs/images',
         },
    url => {
          cgi    => '/perl',
          docs => '/',
          img    => '/images',
         },
    color => {
             hint      => '#777777',
             warn      => '#990066',
             normal => '#000000',
            },
  );
```

良好的 perl 风格，建议在每个列表的末尾放置一个逗号，以便于增加新的条目。

脚本现在看起来如下：

```
use strict;
use Book::Config1 qw(%c);
use vars                  qw(%c);
print "Content-type: text/plain\n\n";
print "My url docs root: $c{url}{docs}\n";
```

现在没有混乱了。只需关注一个变量。

该方法的一个小弊端是 hash 变量的'自动唤醒'功能。例如，假如我们错误的写成$c{url}{doc}，perl 会默默的创建这个 hash 元素，值是 undef。当使用 use strict;时，perl 会告诉我们关于对简单标量的这种错拼情况，但这种检查不对 hash 元素执行。这就将责任放到我们自己身上，必须额外小心。

使用 hash 方法的益处很明显。现在我们可完全消除 Exporter 的调用，从配置文件里删除所有的 export 代码。见如下示例：

Example 6-23. Book/Config2.pm

```
package Book::Config2;
use strict;
use vars qw(%c);

%c = (
    dir => {
          cgi    => '/home/httpd/perl',
          docs => '/home/httpd/docs',
          img    => '/home/httpd/docs/images',
         },
    url => {
          cgi    => '/perl',
          docs => '/',
          img    => '/images',
         },
    color => {
              hint      => '#777777',
              warn      => '#990066',
              normal => '#000000',
             },
  );
```

脚本要修改成使用完全包名限定的变量：

```
use strict;
use Book::Config2 ( );
print "Content-type: text/plain\n\n";
print "My url docs root: $Book::Config2::c{url}{docs}\n";
```

为了避免敲键的麻烦，我们使用 perl 的魔术变量来给配置变量做一个别名：

```
use strict;
use Book::Config2 ( );
use vars qw(%c);
*c = \%Book::Config2::c;
print "Content-type: text/plain\n\n";
print "My url docs root: $c{url}{docs}\n";
```

将*c 这个 glob 别名给 hash 变量的引用。从现在起，对所有实际目的，%Book::Config2::c 和 %c 都指向同一个 hash。

最后一个问题：重复的冗余被引进了配置变量。考虑：

```
$cgi_dir    = '/home/httpd/perl';
$docs_dir = '/home/httpd/docs';
$img_dir    = '/home/httpd/docs/images';
```

明显的，基本路径/home/httpd 应该赋值给独立的变量，这样假如应用移动到文件系统中的其他位置，仅需改变该变量即可。

```
$base         = '/home/httpd';
$cgi_dir    = "$base/perl";
$docs_dir = "$base/docs";
$img_dir    = "$docs_dir/images";
```

这点不能在 hash 里实现，因为不能在 hash 定义完成前，引用它的值。也就是说，这样不能运行：

```
%c = (
    base => '/home/httpd',
    dir => {
          cgi    => "$c{base}/perl",
          docs => "$c{base}/docs",
          img    => "$c{base}{docs}/images",
         },
  );
```

但是我们可增加用 my()声明的词法变量。下列代码是正确的：

```
my $base = '/home/httpd';
%c = (
    dir => {
          cgi    => "$base/perl",
          docs => "$base/docs",
          img    => "$base/docs/images",
         },
  );
```

现在，我们已知道了如何编写易于维护的配置文件，以及如何避免在脚本的命名空间里 import 进变量，从而节省内存。再来看看关于配置文件的 reload。

8.3.2 reload 配置文件

首先看一个简单的示例，在这里我们列出一个简单的配置文件。假设某个脚本告诉你，谁是当前 perl 发布版本的补丁作者。

```
use CGI ( );
```

```
use strict;

my $firstname = "Jarkko";
my $surname = "Hietaniemi";
my $q = CGI->new;

print $q->header(-type=>'text/html');
print $q->p("$firstname $surname holds the patch pumpkin" .
                    "for this Perl release.");
```

该脚本非常简单：它初始化 CGI 目标，打印相应的 HTTP 头部，并告之于众，谁是当前的补丁作者。补丁作者的名字是硬编码的。

每次补丁作者改变时，我们不想修改这个脚本，所以将$firstname 和$surname 变量放进一个配置文件里：

```
$firstname = "Jarkko";
$surname = "Hietaniemi";
1;
```

注意在上述文件里没有包声明，所以 code 会在调用者的包空间或 main::包里被计算。这意味着这 2 个变量$firstname 和$surname 会覆盖（或初始化）调用者包的命名空间里的同名变量。这仅对全局变量有效--不能用这种技术来更新词法变量（my 申明的）。

现在假设已启动服务，并且任何事工作完美。一会儿后，我们决定修改配置文件。如何不重启而让正在运行的服务知道配置文件已更改呢？记住，在产品环境下，重启服务是相当昂贵的。最简单的解决方法之一是在脚本开始做实事之前，调用 stat()来获取文件的修改时间。假如观察到文件更新过，就能强迫重新配置位于该文件里的变量。我们把 reload 配置文件的函数叫做 reread_conf()，并把配置文件的相对路径作为唯一参数传给它。

Apache::Registry 在它开始执行脚本前，会调用 chdir()去到脚本的目录。所以，如果 CGI 脚本在 Apache::Registry 处理器下调用，就可将配置文件放在脚本的同一目录下。另外，可将文件放在脚本目录的子目录下，并使用相对路径来访问。然而，你必须确保文件可被找到。注意 do()在@INC 目录里查找库文件。

```
use vars qw(%MODIFIED);
sub reread_conf {
    my $file = shift;
    return unless defined $file;
    return unless -e $file and -r _;
    my $mod = -M _;
    unless (exists $MODIFIED{$file} and $MODIFIED{$file} =    = $mod) {
        unless (my $result = do $file) {
            warn "couldn't parse $file: $@" if $@;
```

```
            warn "couldn't read $file: $!" unless defined $result;
            warn "couldn't run $file"      unless          $result;
        }
        $MODIFIED{$file} = $mod; # Update the MODIFICATION times
    }
}
```

注意在检查文件的修改时间戳时，用= =匹配符，因为我们要知道的是，是否该文件已改变过。

当 require()，use()，和 do()操作成功的返回时，被操作文件会插入%INC。hash 元素的 key 是文件名，value 是文件的路径。当 perl 在 code 里见到 require()或 use()时，它首先测试%INC 以验证该文件是否已存在，然后才决定 load 与否。假如测试返回真，就节省了重读和重编译 code 的性能开销。然而，do()调用会强行 load 或 reload 文件，而不管是否先前已 load 过。

我们使用 do()而不是 require()来 reload 文件里的 code，是因为 do()无条件的 reload 文件。假如 do()不能读取文件，它返回 undef 并设置$!报告错误。假如 do()能读取文件，但不能编译它，它返回 undef 并将错误消息放在$@里。假如文件成功编译，do()返回最后计算的表达式的值。

假如某人不正确的修改了配置文件，它就会不能执行。因为我们不希望使用该文件的整个服务都被轻易破坏，所以要捕获 do()调用的可能失败情况，并通过 reset 修改时间的方式来忽略改变。假如 do()装载文件失败，最好发封邮件给系统管理员。

然而，既然 do()象 require()一样更新%INC，假如你使用了 Apache::StatINC，它会试图在 reread_conf()调用之前先 reload 这个文件。假如该文件没有编译，请求会失败。Apache::StatINC 不应该用在产品环境，因为它要 stat()所有在%INC 里列举的文件，从而减慢了响应速度。所以上述问题也不用担心。

注意，我们假设该函数的整个目的是在配置文件改变时 reload 它。它是自我保护的，假如出现异常，就不去修改服务配置而简单的返回。动态配置文件不应该用于在它第一次调用时，做初始化变量的事。若要那样做，必须替换每个 return()和 warn()事件为 die()。

我们使用上述方法，对某个大配置文件，让它在服务启动时装载；对另一个包含少许变量的小配置文件，让它由手工更新或通过 web 接口来更新。这些变量都在主配置文件里初始化。假如管理员在手工更新动态配置文件时，打破了配置文件的语法，它不会影响主配置文件，也不会妨碍程序的正常执行。在下面的节里，我们可看到一个简单的 web 接口，它允许我们修改配置文件，而不用冒打破它的风险。

下面展示一个示例脚本，用到了前面的 reread_conf()函数。

Example 6-24. reread_conf.pl

```
use vars qw(%MODIFIED $firstname $surname);
```

```
use CGI ( );
use strict;

my $q = CGI->new;
print $q->header(-type => 'text/plain');
my $config_file = "./config.pl";
reread_conf($config_file);
print $q->p("$firstname $surname holds the patch pumpkin" .
                    "for this Perl release.");

sub reread_conf {
      my $file = shift;
      return unless defined $file;
      return unless -e $file and -r _;
      my $mod = -M _;
      unless ($MODIFIED{$file} and $MODIFIED{$file} == $mod) {
            unless (my $result = do $file) {
                  warn "couldn't parse $file: $@" if $@;
                  warn "couldn't read $file: $!"    unless defined $result;
                  warn "couldn't run $file"             unless $result;
            }
            $MODIFIED{$file} = $mod; # Update the MODIFICATION time
      }
}
```

假如你在修改$^T 变量，就应该用(stat $file)[9]代替-M $file。这是因为-M 返回相对于 perl 解析器启动时的修改时间，存放在$^T 里。在某些脚本里，会有必要 reset $^T 到脚本调用的时间，local $^T = time( )即可。这样，-M 和-X 文件状态测试就相对于脚本调用时间而执行，而不是进程启动时间。

假如配置文件更灵活—例如，假如它声明了包并 export 出变量--上述 code 仍运行良好。变量不必再次 import()进来：当 do()重编译脚本时，原来 import 进的变量会被更新成 reload 的代码里的值。

8.3.3 动态更新配置文件

下述 CGI 脚本允许系统管理员通过 web 接口动态更新配置文件。该脚本结合前面的代码，展示了一个动态重配置而不用重启服务的系统。它可在任何有浏览器的机器上执行。

假设有下述配置文件：

Example 6-25. Book/MainConfig.pm

```
package Book::MainConfig;
```

```
use strict;
use vars qw(%c);

%c = (
        name         => "Larry Wall",
        release    => "5.000",
        comments => "Adding more ways to do the same thing :)",

        other       => "More config values",

        colors      => { foreground => "black",
                         background => "white",
                       },

        machines => [qw( primary secondary tertiary )],

);
```

我们想让变量 name,release 和 comments 动态配置。需要一个 web 接口的 input 表单来修改这些变量。也需要更新配置文件，并告知当前运行的所有进程已发生的变化。

执行的主要步骤如下：

1）创建一个表单，预设变量的当前值；
2）让管理员修改变量，并提交改变；
3）验证提交的信息（例如数字域的数字应在指定范围内等）；
4）更新配置文件；
5）在当前进程的内存里更新修改的值；
6）在表单上显示修改后的值。

看起来唯一执行困难之处是配置文件的更新。假如更新文件失败，整个服务会不能运行。假如文件非常大，并包含注释和复杂的数据结构，解析这个文件会很困难。

让我们来简化任务。假如只需更新少数变量，为什么不能创建一个小的配置文件，仅包含那些变量呢？可以通过 web 接口修改它，假如有东西需要改变，就简单的覆盖它，这样在更新之前不必解析这个文件。假如主配置文件改变了，不必担心，因为根本无须依赖它。

动态更新的变量会在主配置文件和动态配置文件里重复存在。这样作简化了维护任务。当新的 release 版本上线后，动态配置文件不会存在--它仅在第一次更新时才会创建。如前面见到的，在主代码里的唯一改变是增加一小段 code，假如动态文件存在和改变了时，就装载这个文件。

这段附加的 code，必须在主配置文件已装载后执行。这样，更新的变量会覆盖主文件里的

默认值。见如下示例。

Example 6-26. manage_conf.pl

```
# remember to run this code in taint mode
use strict;
use vars qw($q %c $dynamic_config_file %vars_to_change %validation_rules);

use CGI ( );

use lib qw(.);
use Book::MainConfig ( );
*c = \%Book::MainConfig::c;

$dynamic_config_file = "./config.pl";

# load the dynamic configuration file if it exists, and override the
# default values from the main configuration file
do $dynamic_config_file if -e $dynamic_config_file and -r _;

# fields that can be changed and their captions
%vars_to_change =
  (
   'name'         => "Patch Pumpkin's Name",
   'release'    => "Current Perl Release",
   'comments' => "Release Comments",
  );

# each field has an associated regular expression
# used to validate the field's content when the
# form is submitted
%validation_rules =
  (
   'name'         => sub { $_[0] =~ /^[\w\s\.]+$/;      },
   'release'    => sub { $_[0] =~ /^\d+\.[\d_]+$/; },
   'comments' => sub { 1;                                        },
  );

# create the CGI object, and print the HTTP and HTML headers
$q = CGI->new;
print $q->header(-type=>'text/html'),
        $q->start_html( );

# We always rewrite the dynamic config file, so we want all the
```

```
# variables to be passed, but to save time we will only check
# those variables that were changed.    The rest will be retrieved from
# the 'prev_*' values.
my %updates = ( );
foreach (keys %vars_to_change) {
        # copy var so we can modify it
        my $new_val = $q->param($_) || '';

        # strip a possible ^M char (Win32)
        $new_val =~ s/\cM//g;

        # push to hash if it was changed
        $updates{$_} = $new_val
            if defined $q->param("prev_" . $_)
                and $new_val ne $q->param("prev_" . $_);
}

# Note that we cannot trust the previous values of the variables
# since they were presented to the user as hidden form variables,
# and the user could have mangled them. We don't care: this can't do
# any damage, as we verify each variable by rules that we define.

# Process if there is something to process. Will not be called if
# it's invoked the first time to display the form or when the form
# was submitted but the values weren't modified (we'll know by
# comparing with the previous values of the variables, which are
# the hidden fields in the form).

process_changed_config(%updates) if %updates;

show_modification_form( );

# update the config file, but first validate that the values are
# acceptable
sub process_changed_config {
        my %updates = @_;

        # we will list here all variables that don't validate
        my %malformed = ( );

        print $q->b("Trying to validate these values<br>");
        foreach (keys %updates) {
            print "<dt><b>$_</b> => <pre>$updates{$_}</pre>";
```

```
        # now we have to handle each var to be changed very carefully,
        # since this file goes immediately into production!
        $malformed{$_} = delete $updates{$_}
                unless $validation_rules{$_}->($updates{$_});

}

if (%malformed) {
        print $q->hr,
                $q->p($q->b(qq{Warning! These variables were changed
                                        to invalid values. The original
                                        values will be kept.})
                        ),
                join ",<br>",
                        map { $q->b($vars_to_change{$_}) . " : $malformed{$_}\n"
                            } keys %malformed;
}

# Now complete the vars that weren't changed from the
# $q->param('prev_var') values
map { $updates{$_} = $q->param('prev_' . $_)
            unless exists $updates{$_} } keys %vars_to_change;

# Now we have all the data that should be written into the dynamic
# config file

# escape single quotes "'" while creating a file
my $content = join "\n",
        map { $updates{$_} =~ s/(['\\])/\\$1/g;
                '$c{' . $_ . "} = '" . $updates{$_} . "';\n"
            } keys %updates;

# add '1;' to make require( ) happy
$content .= "\n1;";

# keep the dummy result in $res so it won't complain
eval {my $res = $content};
if ($@) {
        print qq{Warning! Something went wrong with config file
                        generation!<p> The error was :</p> <br><pre>$@</pre>};
        return;
}

print $q->hr;
```

```
    # overwrite the dynamic config file
    my $fh = Apache::gensym( );
    open $fh, ">$dynamic_config_file.bak"
        or die "Can't open $dynamic_config_file.bak for writing: $!";
    flock $fh, 2; # exclusive lock
    seek $fh, 0, 0; # rewind to the start
    truncate $fh, 0; # the file might shrink!
    print $fh $content;
    close $fh;

    # OK, now we make a real file
    rename "$dynamic_config_file.bak", $dynamic_config_file
        or die "Failed to rename: $!";

    # rerun it to update variables in the current process! Note that
    # it won't update the variables in other processes. Special
    # code that watches the timestamps on the config file will do this
    # work for each process. Since the next invocation will update the
    # configuration anyway, why do we need to load it here? The reason
    # is simple: we are going to fill the form's input fields with
    # the updated data.
    do $dynamic_config_file;

}

sub show_modification_form {

    print $q->center($q->h3("Update Form"));

    print $q->hr,
        $q->p(qq{This form allows you to dynamically update the current
            configuration. You don't need to restart the server in
            order for changes to take an effect}
              );

    # set the previous settings in the form's hidden fields, so we
    # know whether we have to do some changes or not
    $q->param("prev_$_", $c{$_}) for keys %vars_to_change;

    # rows for the table, go into the form
    my @configs = ( );

    # prepare text field entries
```

```
    push @configs,
        map {
          $q->td( $q->b("$vars_to_change{$_}:") ),
          $q->td(
           $q->textfield(
                    -name           => $_,
                    -default      => $c{$_},
                    -override    => 1,
                    -size          => 20,
                    -maxlength => 50,
                   )
          ),
        } qw(name release);

    # prepare multiline textarea entries
    push @configs,
        map {
          $q->td( $q->b("$vars_to_change{$_}:") ),
          $q->td(
           $q->textarea(
                    -name         => $_,
                    -default    => $c{$_},
                    -override => 1,
                    -rows         => 10,
                    -columns    => 50,
                    -wrap         => "HARD",
                    )
          ),
        } qw(comments);

    print $q->startform(POST => $q->url), "\n",
          $q->center(
               $q->table(map {$q->Tr($_), "\n",} @configs),
               $q->submit(", 'Update!'), "\n",
          ),
          map ({$q->hidden("prev_" . $_, $q->param("prev_".$_)) . "\n" }
                keys %vars_to_change), # hidden previous values
          $q->br, "\n",
          $q->endform, "\n",
          $q->hr, "\n",
          $q->end_html;

}
```

例如，在 2002 年 7 月 19 日，perl 5.8.0 发布了。那天 Jarkko Hietaniemi 惊叫：

The pumpking is dead! Long live the pumpking!

Hugo van der Sanden 是新的 perl 5.10 的补丁作者。这样，我们运行 manage_conf.pl 并更新数据。一旦更新，脚本用如下内容覆盖了以前的 config.pl 文件：

```
$c{release}    =    '5.10';

$c{name}    =    'Hugo van der Sanden';

$c{comments}    =    'Perl rules the world!';

1;
```

如果不想自己编码，就可用 CPAN 的 CGI::QuickForm 模块来减少工作量。见如下示例。

Example 6-27. manage_conf.pl

```
use strict;
use CGI qw( :standard :html3 ) ;
use CGI::QuickForm;
use lib qw(.);
use Book::MainConfig ( );
*c = \%Book::MainConfig::c;

my $TITLE = 'Update Configuration';
show_form(
      -HEADER => header . start_html( $TITLE ) . h3( $TITLE ),
      -ACCEPT => \&on_valid_form,
      -FIELDS => [
            {
                  -LABEL          => "Patch Pumpkin's Name",
                  -VALIDATE      => sub { $_[0] =~ /^[\w\s\.]+$/;      },
                  -default        => $c{name},
            },
            {
                  -LABEL          => "Current Perl Release",
                  -VALIDATE      => sub { $_[0] =~ /^\d+\.[\d_]+$/; },
                  -default        => $c{release},
            },
            {
                  -LABEL          => "Release Comments",
                  -default        => $c{comments},
```

```
            },
            ],
    );

sub on_valid_form {
    # save the form's values
}
```

show_form( )创建和显示一个带提交按钮的表单。当用户提交后，值被检查。假如所有的域都有效，则调用 on_valid_form( )函数；否则，表单会高亮呈现错误。

9 处理用户按下"Stop"按钮的情况

当用户按下 Stop 或 Reload 按钮时，当前 socket 连接会打断。如果 apache 能立即检测到这种事件，则再好不过。不幸的是，除非有读写 socket 的动作，否则没办法去检查连接是否有效。

注意，假如连向后台 mod_perl 服务的请求，是来自前端的 mod_proxy，则没有可行技术能检测连接有效性。这是因为当用户中断连接时，mod_proxy 不会断开到后台服务的连接。

假如读取请求数据的动作已完成，并且 mod_perl 不往客户端写任何东西了，则这种断开的连接难以察觉。当试图往客户端写哪怕一个字符时，断开的连接会引起注意，并发送 SIGPIPE 信号到后台进程。这时程序就可执行它自己的清理代码。

在 apache 1.3.6 之前，SIGPIPE 由 apache 处理。当前，apache 不处理 SIGPIPE，改由 mod_perl 处理。

在 mod_perl 下，$r->print（或者就是 print()）在成功时返回真值，失败时返回假值。后者常在连接中断时发生。

假如想用旧的 SIGPIPE 行为（apache 1.3.6 之前），请增加下列配置指令：

```
PerlFixupHandler Apache::SIG
```

当用上 apache 的 SIGPIPE 处理器时，perl 驻留在 eval()上下文中间，当随后的请求被该子进程处理时，会导致离奇的错误。在用上 Apache::SIG 时，它安装一个不同的 SIGPIPE 处理器，在处理新请求前，该处理器会根据上下文情况判断 perl 是否处于正常状态，从而阻止前述错误的发生。通常情况下，不必使用 Apache::SIG。

假如使用了 Apache::SIG，并想在 access_log 里体现因为 SIGPIPE 而被取消的请求情况，则需在 httpd.conf 里定义一个 logFormat，例如：

```
PerlFixupHandler Apache::SIG
```

LogFormat "%h %l %u %t \"%r\" %s %b %{SIGPIPE}e"

假如服务器注意到请求被 SIGPIPE 取消，日志行以 1 结尾。否则，该行以'-'结尾。例如：

```
127.0.0.1 - - [09/Jan/2001:10:27:15 +0100]
"GET /perl/stopping_detector.pl HTTP/1.0" 200 16 1
127.0.0.1 - - [09/Jan/2001:10:28:18 +0100]
"GET /perl/test.pl HTTP/1.0"                    200 10 -
```

9.1 检测中断的连接

现在利用已有的知识，来跟踪代码的执行，并观察所有发生的事件。如下 Apache::Registry 脚本有意的在运行时挂起服务进程，如下：

Example 6-28. stopping_detector.pl

```
my $r = shift;
$r->send_http_header('text/plain');

print "PID = $$\n";
$r->rflush;

while (1) {
    sleep 1;
}
```

该脚本通过 shift()方式从@_参数列表（由 handler()子函数传进来的，它由 Apache::Registry 在线创建）获取一个请求对象$r。然后该脚本发送一个 Content-Type 的头部告诉客户端，准备发送一个明文响应。

下一步，脚本打印出处理请求的进程 ID，这点必须知道以用于后台调试。然后刷新 apache 的 STDOUT 缓存。假如不刷新缓存，就不会看到打印的信息（因为输出信息短于 print()函数的缓冲区 size，并且脚本有意挂起，所以缓冲区不会自动刷新）。

然后进入一个无限循环，除了 sleep()外不做任何事，这用于模拟不产生任何输出的代码运行情况。例如，可能在进行海量数据计算，数据库查询，或搜索外星生命。

在后台运行 strace -p PID，PID 是浏览器上显示的进程 ID，我们可看到每秒都有下列输出：

```
rt_sigprocmask(SIG_BLOCK, [CHLD], [    ], 8)    = 0
rt_sigaction(SIGCHLD, NULL, {SIG_DFL}, 8) = 0
rt_sigprocmask(SIG_SETMASK, [    ], NULL, 8)    = 0
nanosleep({1, 0}, {1, 0})                       = 0
time([978969822])                                 = 978969822
```

```
time([978969822])                       = 978969822
```

另外，可以单进程方式运行服务。在单进程下，不必打印进程 ID，因为 PID 就是正在运行的 mod_perl 进程的 ID。当进程在后台启动时，shell 程序通常会打印进程的 PID，如下所示：

```
panic% httpd -X &
 [1] 20107
```

再次运行 strace:

```
panic% strace -p 20107
rt_sigprocmask(SIG_BLOCK, [CHLD], [    ], 8)    = 0
rt_sigaction(SIGCHLD, NULL, {SIG_DFL}, 8) = 0
rt_sigprocmask(SIG_SETMASK, [    ], NULL, 8)    = 0
nanosleep({1, 0}, {1, 0})                       = 0
time([978969822])                           = 978969822
time([978969822])                           = 978969822
```

可见到和前面一样的输出。

让 strace 继续运行并按下 Stop 按钮。事情改变了吗？NO。同样的系统调用每秒还在进行，这意味着 apache 没有检测到打断的连接。

现在我们写一个\0（空字节）到客户端，试图尽可能快的检测到中断的连接。既然它是空字节，就不会见到输出。这样，修改循环代码如下：

```
while (1) {
    $r->print("\0");
    last if $r->connection->aborted;
    sleep 1;
}
```

增加一个 print()用以打印空字节，然后通过$r->connection->aborted 方法 ，检查连接是否已断开。假如连接断开了，我们中断循环。

现在运行这个脚本，并如前面一样运行 strace，但见到它仍不能工作--当按下 Stop 按钮时，脚本没有停止。

问题在于没有刷新缓存。空字符不会打印出来，直到缓存满了自动刷新后。因为我们想实时的写往连接管道，这样增加一个$r->rflush()调用。修改后的代码如下：

Example 6-29. stopping_detector2.pl

```
my $r = shift;
```

```
$r->send_http_header('text/plain');

print "PID = $$\n";
$r->rflush;

while (1) {
      $r->print("\0");
      $r->rflush;
      last if $r->connection->aborted;
      sleep 1;
}
```

在运行 strace 并按下 Stop 按钮后，可见到如下输出：

```
rt_sigprocmask(SIG_BLOCK, [CHLD], [    ], 8)    = 0
rt_sigaction(SIGCHLD, NULL, {SIG_DFL}, 8) = 0
rt_sigprocmask(SIG_SETMASK, [    ], NULL, 8)    = 0
nanosleep({1, 0}, {1, 0})                       = 0
time([978970895])                                 = 978970895
alarm(300)                                         = 0
alarm(0)                                           = 300
write(3, "\0", 1)                                  = -1 EPIPE (Broken pipe)
--- SIGPIPE (Broken pipe) ---
chdir("/usr/src/httpd_perl")                    = 0
select(4, [3], NULL, NULL, {0, 0})          = 1 (in [3], left {0, 0})
time(NULL)                                         = 978970895
write(17, "127.0.0.1 - - [08/Jan/2001:19:21"..., 92) = 92
gettimeofday({978970895, 554755}, NULL) = 0
times({tms_utime=46, tms_stime=5, tms_cutime=0,
   tms_cstime=0}) = 8425400
close(3)                                           = 0
rt_sigaction(SIGUSR1, {0x8099524, [    ], SA_INTERRUPT|0x4000000},
   {SIG_IGN}, 8) = 0alarm(0)                                          = 0
rt_sigprocmask(SIG_BLOCK, NULL, [    ], 8)    = 0
rt_sigaction(SIGALRM, {0x8098168, [    ], SA_RESTART|0x4000000},
   {0x8098168, [    ], SA_INTERRUPT|0x4000000}, 8) = 0
fcntl(18, F_SETLKW, {type=F_WRLCK, whence=SEEK_SET,
   start=0, len=0}) = 0
```

apache 检测到了中断的管道，可从如下片断体现出来：

```
write(3, "\0", 1)                                  = -1 EPIPE (Broken pipe)
--- SIGPIPE (Broken pipe) ---
```

然后它停止脚本，并执行所有的清理动作，例如写访问日志：

write(17, "127.0.0.1 - - [08/Jan/2001:19:21"..., 92] = 92

这里 17 是打开的 access_log 文件的文件描述符。

9.2 清理代码的重要性

清理代码用于处理中断脚本中的临界情况。例如，锁定资源发生了什么情况？它们会被释放吗？假如不释放锁，使用同一锁机制的脚本可能会永远挂起，等待这些资源被释放。

假如文件打开后永不关闭，会发生什么呢？在某些情况下，这会导致文件描述符用尽。在长期运行中，这种描述符耗费会导致系统不可用：当所有文件描述符都用完时，系统不能打开新文件。

首先，简要回顾下在 mod_cgi 下，会发生什么问题和怎样解决。在 mod_cgi 下，仅在不使用 flock()，而使用外部锁文件时，才会导致资源锁问题。假如运行在 mod_cgi 下的脚本在 lock 和 unlock 代码之间崩溃了，并且你没有编写清理代码来删除旧的僵死的锁，就会陷入巨大的麻烦。

解决方法是将清理代码放在 END 块里：

```
END {
    # code that ensures that locks are removed
}
```

当脚本中断时，perl 解析器在关闭时会运行 END 块。

假如使用了 flock()，事情非常简单，因为所有打开的文件会随着脚本退出而关闭。当文件关闭时，锁也删除了--所有锁定的资源被释放。有些系统不支持 flock()，在这些系统中，可使用 perl 的模拟函数。

在 mod_perl 下，当使用全局变量作为文件句柄时，事情变得更复杂。因为进程在处理完某个请求后不会退出，文件不会关闭，除非显示的调用 close()或用 open()重新打开它们，open()会先关闭文件。让我们看看会遇到什么问题，以及可能的解决方案。

9.2.1 临界代码

首先，讨论下临界代码情况，以某个资源锁机制开始。正确的锁技术的步骤如下：

1）锁定资源
   <临界代码开始>

2）对资源做某些操作
    <临界代码结束>

3）释放资源锁

假如锁是专用的，仅仅一个进程在任意给定时刻能处理这个资源，这意味着所有其他进程必须等待。在 lock 和 unlock 函数之间的代码不能打断，这样它就变成了一个服务瓶颈。这就是为什么这段 code 被称为临界的原因。它的执行时间应该尽可能的短。

即使使用共享锁机制，它允许许多进程同时访问资源，也要尽可能的保持临界代码短小，有可能某个进程需要对这个资源的专用锁。

如下示例使用了共享锁，但临界代码设计很糟糕。

Example 6-30. critical_section_sh.pl

```
use Fcntl qw(:flock);
use Symbol;

my $fh = gensym;
open $fh, "/tmp/foo" or die $!;

# start critical section
flock $fh, LOCK_SH;    # shared lock, appropriate for reading
seek $fh, 0, 0;
my @lines = <$fh>;
for (@lines) {
      print if /foo/;
}
close $fh; # close unlocks the file
# end critical section
```

该代码打开文件用于读，锁住资源并将指针指向文件开头，从文件里读取所有行，并打印包含'foo'的行。

gensym()函数由 Symbol 模块 import 进来，它创建一个匿名 glob 数据结构，并返回对其的引用。这样的 glob 引用能被用于文件或目录句柄。这样，它允许使用词法变量作为文件句柄。

Fcntl 使用:flock 的组标签，导入文件锁符号到脚本的名字空间，例如 LOCK_SH,LOCK_EX，以及其他。参考 Fcntl 的 manpage 关于这些符号的更多信息。

假如文件很大，会花很长时间去搜索和打印这些行。在这个时间内，文件保持打开并被共享锁锁住。在其他进程可读取这个文件的同时，某个进程可能想要修改这个文件（这需要一个专用锁），这样该进程就会阻塞直到临界代码执行完。

可以按如下方式优化临界代码。一旦文件已打开，其实就已拥有所需要的一切信息。为了让示例更简单，我们这里仅打印匹配的行。在实际中，这段代码可能要长的多。

在循环执行时，不必让文件保持打开，因为不会在循环内部访问文件。在开始循环之前关闭文件，这样就允许其他进程在必要时获得该文件的专用锁，而不是被无端阻塞。

如下示例是前述示例的改进版本，在临界代码里我们仅读取文件内容，在后面再处理它，这样就不会导致可能的瓶颈。

Example 6-31. critical_section_sh2.pl

```
use Fcntl qw(:flock);
use Symbol;

my $fh = gensym;
open $fh, "/tmp/foo" or die $!;

# start critical section
flock $fh, LOCK_SH;
seek $fh, 0, 0;
my @lines = <$fh>;
close $fh; # close unlocks the file
# end critical section

for (@lines) {
    print if /foo/;
}
```

如下示例简单的使用了专用锁。脚本读取整个文件，并重新写回它，在文件开头增加了一些新文本行。

Example 6-32. critical_section_ex.pl

```
use Fcntl qw(:flock);
use Symbol;

my $fh = gensym;
open $fh, "+>>/tmp/foo" or die $!;

# start critical section
flock $fh, LOCK_EX;
seek $fh, 0, 0;
my @add_lines =
```

```
  (
    qq{Complete documentation for Perl, including FAQ lists,\n},
    qq{should be found on this system using 'man perl' or\n},
    qq{'perldoc perl'. If you have access to the Internet, point\n},
    qq{your browser at http://www.perl.com/, the Perl Home Page.\n},
  );

my @lines = (@add_lines, <$fh>);
seek $fh, 0, 0;
truncate $fh, 0;
print $fh @lines;
close $fh; # close unlocks the file
# end critical section
```

因为想要读取文件并修改它，然后再写回它，在这过程不允许其他进程改变这个文件，这样就需要一个专用锁来锁定它。不能想当然的先打开这个文件读取内容，再重新打开它用于写，因为在这 2 个事件之间，其他进程可能已改变了文件。

下一步，准备要增加到文件开头处的文本行，并将它们和文件的内容合并到@lines 数组。现在有了要写回文件的完整内容，所以我们 seek()到文件的开始处，并 truncate()它到 0 大小。若文件内容可能变短，则 truncate 是必要的。在本处示例里，文件总是增加的，所以这里实际没必要去 truncate 它。然而，总是使用 truncate()是良好的编程习惯，因为你永远不会知道代码将来会面临什么改变，并且 truncate()不会明显影响性能。

最后，我们将数据写回文件并关闭它，这样也释放了资源锁。

我们在尽可能清楚的地方创建需要写入文件开头位置的文本行。这种做法让代码意图更明确，但让临界代码变得更长。在这种情况下，你应该想办法让临界代码尽可能短。改进后的代码版本如下，在这里临界代码更短。

Example 6-33. critical_section_ex2.pl

```
use Fcntl qw(:flock);
use Symbol;

my @lines =
  (
    qq{Complete documentation for Perl, including FAQ lists,\n},
    qq{should be found on this system using 'man perl' or\n},
    qq{'perldoc perl'. If you have access to the Internet, point\n},
    qq{your browser at http://www.perl.com/, the Perl Home Page.\n},
  );

my $fh = gensym;
```

```
open $fh, "+>>/tmp/foo" or die $!;

# start critical section
flock $fh, LOCK_EX;
seek $fh, 0, 0;
push @lines, <$fh>;

seek $fh, 0, 0;
truncate $fh, 0;
print $fh @lines;
close $fh; # close unlocks the file
# end critical section
```

这里有 2 个重要的不同。首先，我们在文件锁定之前准备好文本行。其次，与创建一个新数组，并从一个数组拷贝内容到另一个数组不同，我们直接追加文件内容到@lines 数组。

9.2.2 安全资源锁和清理代码

现在回到本节的主要内容，安全资源锁。假如你没有养成关闭所有的打开文件的习惯，就会面临许多问题（除非使用 Apache::PerlRun 处理器，它自己做清理）。打开文件如果不关闭，可以导致文件描述符用尽。因为文件描述符的数量是有限的，这样就可能用尽它们导致服务不正常。在高负载的服务器上这点尤其容易发生。

可以使用系统工具来观察打开和锁住的文件，跟进程自己打开和锁住文件一样。在 FreeBSD 上，使用 fstat 工具。在其他的许多 Unix 上，使用 lsof。对支持/proc 文件系统的 OS，可在 /proc/PID/fd 目录下见到打开的文件描述符，这里 PID 是实际的进程 ID。

然而，文件描述符用尽，相对于程序退出但文件没被解锁的情况来说，还不算什么。任何其他的进程对同一文件（或资源）请求锁时，会一直等待它被解锁。如果服务器不重启，那么锁不会解开，所有请求这一资源锁的进程就会挂起。

如下示例显示这个可怕的错误：

```
use Fcntl qw(:flock);
open IN, "+>>filename" or die "$!";
flock IN, LOCK_EX;
# do something
# quit without closing and unlocking the file
```

该代码安全吗？NO。忘记了关闭文件。所以增加 close()调用如下：

Example 6-35. flock2.pl

```
use Fcntl qw(:flock);
```

```
open IN, "+>>filename" or die "$!";
flock IN, LOCK_EX;
# do something
close IN;
```

现在代码安全吗？不幸的是，它仍然不。假如用户在临界代码阶段中断请求（例如，按下浏览器的 Stop 或 Reload 按钮），脚本会在有机会 close()文件之前而退出，这跟忘记关闭文件的情况一样。

事实上，假如同一进程再次运行同一代码，open()调用会 close()该文件先，这样会释放资源锁。这是因为 IN 是全局变量。但是很可能创建资源锁的进程在一段时间内不会响应同一请求，因为它可能忙于处理其他请求。在这段时间内，文件会对其他进程锁住，导致它们挂起。所以依赖同一进程去重新打开文件是坏主意。

问题仅在你使用全局变量作为文件描述符时才会发生。如下示例有相同的问题。

Example 6-36. flock3.pl

```
use Fcntl qw(:flock);
use Symbol ( );
use vars qw($fh);
$fh = Symbol::gensym( );
open $fh, "+>>filename" or die "$!";
flock $fh, LOCK_EX;
# do something
close $fh;
```

$fh 仍然是个全局变量，这样使用它的 code 会面临前面同样的问题。

最简单的解决方案是总使用词法变量（由 my 创建）。词法变量总不会超出词法范围而存在（假设它没有用于闭包，在本文开头处有介绍），无论在 close()调用前脚本退出或者是你简单的忘记了 close()文件。这样，假如文件锁住了，code 走出词法范围后，文件会关闭和解锁。如下是好的代码版本：

Example 6-37. flock4.pl

```
use Fcntl qw(:flock);
use Symbol ( );
my $fh = Symbol::gensym( );
open $fh, "+>>filename" or die "$!";
flock $fh, LOCK_EX;
# do something
close $fh;
```

假如你使用该方法，请不要推断从此不必关闭文件，认为它们会自动关闭。不关闭文件是坏的编程风格，应该避免。

注意 Perl 5.6 提供了一个 Symbol.pm，它类似于内建的函数库，所以可直接这样写：

```
open my $fh, ">/tmp/foo" or die $!;
```

$fh 会自动成为有效的文件句柄。不必再使用 Symbol::gensym 和 Apache::gensym，假如没有向后兼容性需求的话。

也可使用 IO::*模块，例如 IO::File 或 IO::Dir。它们比 Symbol 模块大得多（事实是，这些模块自身也使用 Symbol 模块），假如你需要它们提供的附加功能的话，才有必要去使用。如下是用法示例：

```
use IO::File;
use IO::Dir;
my $fh = IO::File->new(">filename");
my $dh = IO::Dir->new("dirname");
```

另外，也有轻量级的 FileHandle 和 DirHandle 模块。

假如仍必须使用全局文件句柄，也有一些方法用于在脚本不正常中断时做清理工作。

假如运行在 Apache::Registry 下，END 块可执行清理工作。跟 mod_cgi 或普通的 Perl 脚本一样的使用 END 块。简单的增加 cleanup 代码到这个块，就可以安全了。

例如，假如在用 DBM 文件工作，刷新 DBM 缓存是很重要的，使用 sync()方法：

```
END {
    # make sure that the DB is flushed
    $dbh->sync( );
}
```

在 mod_perl 下，上述 code 仅在 Apache::Registry 和 Apache::PerlRun 下能工作。否则，END 块会延缓到进程终止才执行。假如你编写了一个 mod_perl API 的处理器，请使用 register_cleanup()方法代替。它接受一个到子函数的引用作为参数。可以重写 DBM 同步的 code 如下：

```
$r->register_cleanup(sub { $dbh->sync( ) });
```

这点也能在 Apache::Registry 下工作。

更好的方式是去检查是否客户端连接已中断。否则，总是执行清理代码可能不是你想要的（对正常的脚本完成，就不必去执行它）。执行这个检查的方法如下：

```
$r->register_cleanup(
   # make sure that the DB is flushed
   sub {
         $dbh->sync( ) if Apache->request->connection->aborted( );
   }
);
```

或者，假如在 END 块里，使用：

```
END {
      # make sure that the DB is flushed
      $dbh->sync( ) if Apache->request->connection->aborted( );
}
```

注意假如使用了 register_cleanup()，它会在脚本的开头处调用，或者随着你使用的变量可用而调用。假如在脚本尾部使用它，并且脚本碰巧在执行这个代码前中断了，就不会执行清理动作。

例如，CGI.pm 在它的 new()方法里注册了一个 cleanup 子函数：

```
sub new {
   # code snipped
   if ($MOD_PERL) {
         Apache->request->register_cleanup(\&CGI::_reset_globals);
         undef $NPH;
   }
   # more code snipped
}
```

为 mod_perl API 处理器注册 cleanup 代码的另一个方法，是在配置文件里使用 PerlCleanupHandler：

```
<Location /foo>
      SetHandler perl-script
      PerlHandler             Apache::MyModule
      PerlCleanupHandler Apache::MyModule::cleanup( )
      Options ExecCGI
</Location>
```

这里 Apache::MyModule::cleanup 执行清理。

10　处理服务超时及使用$SIG{ALRM}

类似于用户按下 Stop 按钮来终止脚本的情况，浏览器自身可能在一段时间内（通常是几分钟）没有输出后，自行中断脚本。

有时候脚本执行很长时间的操作，可能久于客户端的超时时间。

这在执行海量数据库查询时可能发生。另一个例子是脚本与外部应用程序进行交互，然而外部程序的响应时间不能得到保证。考虑一个脚本，它从另一个站点抓取页面，处理完后返回给用户。显然，没什么能保证页面可被快速抓取到。

在该情形下，使用$SIG{ALRM}来阻止超时：

```
my $timeout = 10; # seconds
eval {
   local $SIG{ALRM} =
         sub { die "Sorry, timed out. Please try again\n" };
   alarm $timeout;
   # some operation that might take a long time to complete
   alarm 0;
};
die $@ if $@;
```

在本 code 里，将可能长时间运行的操作放在 eval 块里。首先我们初始化一个本地化的 ALRM 信号处理器，它位于特殊的%SIG 哈希里。假如触发了该处理器，它会调用 die()并且中断 eval 块。然后可以根据 eval 的返回情况来做相应的处理，这里我们选择终止脚本执行。大多数情况下，你可能需要向用户报告，操作已超时。

实际的操作放在 2 个 alarm()调用之间。第一个 alarm 开始计时器，第二个取消它。这里计时器运行 10 秒。假如在这 10 秒里，第二个 alarm 没有发生，则产生 SIGALRM 信号，存储在$SIG{ALRM}里的处理器被调用。在本示例里，这样会异常的退出 eval 块。

假如在 2 个 alarm 之间的操作在 10 秒内完成，alarm 时钟会终止，eval 块成功返回，不会触发 ALRM 处理器。

注意在给定时间内，仅仅一个计时器可用。alarm()的返回值是剩余时间的数量。所以实际上可利用这点来粗略的估计执行时间。

通常容易混淆 alarm()和 sleep()调用。sleep()由系统内在的执行。而 alarm()会中断先前的 alarm()设置，因为每个新的 alarm()调用会取消前面的 alarm。

最后，实际上这里的时间方案并不精确，超时周期会在正负 1 秒之间有差异。超时值可能在 9-11 秒之间变化。要得到精确度小于 1 秒的计时器，可使用 Perl 的 4 参数版本的 select()，

保留前面 3 个参数为空。也有其他技术存在，但在这里用不上，这里我们用 alarm()来执行超时。

11 产生正确的 HTTP 头部

HTTP 头部至少包含 2 个域：HTTP 响应码和 MIME 类型（Content-Type）：

```
HTTP/1.0 200 OK
Content-Type: text/plain
```

在头部后增加一个新行，就可开始输出内容。更复杂的头部包括 data 时间戳和服务器类型。例如：

```
HTTP/1.0 200 OK
Date: Tue, 10 Apr 2001 03:01:36 GMT
Server: Apache/1.3.19 (Unix) mod_perl/1.25
Content-Type: text/plain
```

为了通知客户端服务器配置成 KeepAlive Off，客户端必须知道在内容发送完后，连接会关闭：

```
Connection: close
```

也有其他的头部，例如 cache 控制头部和其他符合 HTTP 协议规范的头部。可以用一个 print() 陈述来编码 HTTP 头部：

```
print qq{HTTP/1.1 200 OK
   Date: Tue, 10 Apr 2001 03:01:36 GMT
   Server: Apache/1.3.19 (Unix) mod_perl/1.25
   Connection: close
   Content-Type: text/plain

   };
```

或用"here"风格的 print():

```
print <<'EOT';
   HTTP/1.1 200 OK
   Date: Tue, 10 Apr 2001 03:01:36 GMT
   Server: Apache/1.3.19 (Unix) mod_perl/1.25
   Connection: close
   Content-type: text/plain
```

EOT

别忘了在 HTTP 头部之后包含 2 个新行。在 Apache::Util::ht_time()的协助下，可产生用于 Date:域的时间戳。

假如想发送非默认的头部，使用 header_out()方法。例如：

```
$r->header_out("X-Server" => "Apache Next Generation 10.0");
$r->header_out("Date" => "Tue, 10 Apr 2001 03:01:36 GMT");
```

当头部设置完成后，send_http_header()方法会刷新头部，增加一个新行用以指明内容的开始处。

```
$r->send_http_header;
```

某些头部有特殊别名。例如：

```
$r->content_type('text/plain');
```

等同于：

```
$r->header_out("Content-Type" => "text/plain");
```

无论何时，若有具体的方法可用，则应使用那些方法，而不是直接设置头部。

典型的处理器看起来如下：

```
use Apache::Constants qw(OK);
$r->content_type('text/plain');
$r->send_http_header;
return OK if $r->header_only;
```

为了兼容 HTTP 协议，假如客户端发布 HTTP HEAD 请求，而不是通常的 GET，我们就只发送 HTTP 头部。当 Apache 接受到 HEAD 请求后，header_only()返回真。这样，在本示例里，在发送完头部后，处理器立刻返回。

某些情形下，若 Apache 基于请求能识别出正确的 MIME 类型，则可跳过显式的 content-type 设置。例如，假如请求 HTML 文件，默认的 text/html 会用于响应的 content 类型。Apache 在 mime.types 文件里查找 MIME 类型。可以重设默认的 content 类型。

在 Apache::Registry 和类似的处理器下，形势有所不同。考虑一个基本的 CGI 脚本：

```
print "Content-type: text/plain\n\n";
print "Hello world";
```

默认的，这样不会工作，因为它看起来象正常的文本，没有发送 HTTP 头部。可以改变这个形势，增加：

PerlSendHeader On

到配置文件的 Apache::Registry <Location>节。
现在响应行和普通头部会以 mod_cgi 同样的方式发送出去。跟 mod_cgi 一样，即使设置了 PerlSendHeader On，脚本仍须发送 MIME 类型，以及一个终止的双新行：

print "Content-type: text/html\n\n";

PerlSendHeader On 指令告诉 mod_perl 截取任何看起来象头部行的输出（例如 Content-Type: text/plain）并且自动转换它到正确格式的 HTTP 头部，非常类似于运行在 mod_cgi 下的 CGI 脚本。这点允许你保持 CGI 脚本无须修改。

可使用$ENV{PERL_SEND_HEADER}来发现 PerlSendHeader 是打开还是关闭的。

```
if ($ENV{PERL_SEND_HEADER}) {
    print "Content-type: text/html\n\n";
}
else {
    my $r = Apache->request;
    $r->content_type('text/html');
    $r->send_http_header;
}
```

注意总可以使用上述示例的'else'部分的 code，无论 PerlSendHeader 是打开还是关闭的。

假如你使用 CGI.pm 的 header()函数来产生 HTTP 头部，则不必激活该指令，因为 CGI.pm 会检测 mod_perl 并自动调用 send_http_header()。

天下没有免费的午餐--激活这个指令获取了 mod_cgi 的行为，然而增加了少许解析发送文本的负载。注意 mod_perl 假设一个头部不会跨越多个 print()陈述。

Apache::print()函数必须收集脚本输出的头部，并 pass 给$r->send_http_header。这点发生在 src/modules/perl/Apache.xs (print( ))和 Apache/Apache.pm(send_cgi_header())。如果你使用多个 print()来产生一个 Set-Cookie 如下：

```
print "Content-type: text/plain\n";
print "Set-Cookie: iscookietext\; ";
print "expires=Wednesday, 09-Nov-1999 00:00:00 GMT\; ";
print "path=\/\; ";
print "domain=\.mmyserver.com\; ";
```

```
print "\n\n";
print "Hello";
```

产生的 Set-Cookie 头部被拆分成多个 print()，这样就丢失了。上述示例不会工作。试试这个：

```
my $cookie = "Set-Cookie: iscookietext\; ";
$cookie .= "expires=Wednesday, 09-Nov-1999 00:00:00 GMT\; ";
$cookie .= "path=\/\; ";
$cookie .= "domain=\.mmyserver.com\; ";
print "Content-type: text/plain\n",
print "$cookie\n\n";
print "Hello";
```

使用特殊目的的 cookie 产生模块（例如 Apache::Cookie or CGI::Cookie）是更清晰的解决方案。

有时候调用脚本时，会看到一个"Content-Type: text/html"显示在页面顶部，并且经常 HTML 内容不会正确的呈现在浏览器上。如同所见一样，这通常是因为 code 发送了 2 次头部。

假如有一个复杂的应用，头部可能依赖于逻辑代码，而从许多不同的地方发送。这就需要编写一个子函数，它用于发送头部并跟踪是否头部已被发送过。可使用一个全局变量来标明是否发送过头部，如下所示：

Example 6-38. send_header.pl

```
use strict;
use vars qw($header_printed);
$header_printed = 0;

print_header("text/plain");
print "It worked!\n";
print_header("text/plain");

sub print_header {
    return if $header_printed;

    my $type = shift || "text/html";
    $header_printed = 1;
    my $r = Apache->request;
    $r->content_type($type);
    $r->send_http_header;
}
1;
```

$header_printed 是个布尔变量，指明是否头部已被发送。在每次调用 code 之初，它初始化为 false(0)。注意在同一请求里第二次调用 print_header()时，该函数立即返回，因为 $header_printed 在第一次调用后，已变为 true。

可继续改进该子函数去处理其他的头部，例如 cookie。

12 方法处理器：两个内容处理器示例

假设需要执行一个处理器，它允许浏览文档根目录及下级目录的文件。可浏览目录（以便你能遍历目录树），但文件不可阅读（可以见到可用文件，但不可双击打开）。

这里就需要编写一个简单的文件浏览器。我们了解客户的行为，所以假设客户不久又会需求与其相似的定制模块。为了避免以后的重复劳动，我们决定编写一个基本类，它的方法可随时重写。这个基本类叫做 Apache::BrowseSee。

先申明包名并使用 strict 参数：

```
package Apache::BrowseSee;
use strict;
```

然后导入通用常量（例如 OK,NOT_FOUND 等），装载 File::Spec::Functions 和 File::Basename 模块，并导进一些路径处理函数：

```
use Apache::Constants qw(:common);
use File::Spec::Functions qw(catdir canonpath curdir updir);
use File::Basename 'dirname';
```

现在编写函数，以简单的构造器开始：

```
sub new { bless {    }, shift;}
```

实际的入口点即处理器，申明原型为($$)：

```
sub handler ($$) {
    my($self, $r) = @_;
    $self = $self->new unless ref $self;
    $self->{r} = $r;
```

然后找到请求记录的 path_info 参数：

```
$self->{dir} = $r->path_info || '/';
```

例如，假如请求是/browse/foo/bar，/browse 是处理器的位置，这时 path_info 元素就是/foo/bar。

假如没有指定路径，就使用默认的'/'。

然后重设 dirs 和 files 的接口：

```
$self->{dirs}  = {  };
$self->{files} = {  };
```

这点是必要的，因为有可能$self 目标在处理器之外创建（例如，在 startup 文件里），并且在多个请求之间存在。

现在试图获取目录的内容：

```
eval { $self->fetch( ) };
return NOT_FOUND if $@;
```

假如 fetch()方法 die 掉，错误消息包含在$@内，我们返回 NOT_FOUND。你可以选择不同的处理方法，返回一个错误消息解释发生了什么。也有可能在返回之前需要记录日志：

```
warn($@), return NOT_FOUND if $@;
```

正常的这点不会发生，除非用户搅乱参数（请小心，用户确实会这么做）。

当 fetch()函数成功执行后，剩下的事就是使用 head()方法发送 HTTP 头部，呈现响应，使用 tail()函数发送 HTML 尾部，并最终返回 OK 常量告诉服务器请求已处理完毕。

```
    $self->head;
    $self->render;
    $self->tail;

    return OK;
}
```

响应由 3 个函数产生。head()方法非常简单--它发送 HTTP 头部 text/html，并使用当前目录名作为 title 打印一个 HTML 导言：

```
sub head {
    my $self = shift;
    $self->{r}->send_http_header("text/html");
    print "<html><head><title>Dir: $self->{dir}</title><head><body>";
}
```

tail()方法完成 HTML 文档：

```
sub tail {
```

```
    my $self = shift;
    print "</body></html>";
}
```

fetch()方法读取存储在目标的 dir 属性（相对于文档根目录）里的目录内容，并将内容排序成 2 组，即目录和文件：

```
sub fetch {
    my $self = shift;
    my $doc_root = Apache->document_root;
    my $base_dir = canonpath( catdir($doc_root, $self->{dir}));

    my $base_entry = $self->{dir} eq '/' ? '' : $self->{dir};
    my $dh = Apache::gensym( );
    opendir $dh, $base_dir or die "Cannot open $base_dir: $!";
    for (readdir $dh) {
        next if $_ eq curdir( );    # usually '.'

        my $full_dir = catdir $base_dir, $_;
        my $entry = "$base_entry/$_";
        if (-d $full_dir) {
            if ($_ eq updir( )) { # '..'
                $entry = dirname $self->{dir};
                next if catdir($base_dir, $entry) eq $doc_root;
            }
            $self->{dirs}{$_} = $entry;
        }
        else {
            $self->{files}{$_} = $entry;
        }
    }
    closedir $dh;
}
```

通过使用 canonpath()，我们确保没人搅乱 path_info 元素。"/."是 unix 上的，这里采取相应的动作处理其他操作系统的情况。在开发应用程序时，使用 File::Spec 和其他的跨平台函数是很重要的。

在遍历目录接口时，我们使用 curdir()函数（从 File::Spec::Functions 导入，它等同于 unix 的"."）跳过当前目录，并通过匹配 updir()函数（等同于 unix 上的".."）处理父目录接口。dirname()函数返回父目录，然后我们检查该目录是否与文档根目录不同。如果相同，则跳过这个接口。

注意因为使用 path_info 参数来传递相对于文档根目录的目录名，我们就依赖于 Apache 去处理用户试图在 URL 里搅乱目录的情况（例如加入".."试图获取不被允许的上级目录的文件）。

最后，看看 render()方法：

```
sub render {
    my $self = shift;
    print "<p>Current Directory: <i>$self->{dir}</i><br>";

    my $location = $self->{r}->location;
    print qq{<a href="$location$self->{dirs}{$_}">$_</a><br>}
        for sort keys %{ $self->{dirs} || {   } };
    print qq{$_<br>}
        for sort keys %{ $self->{files} || {   } };
}
```

render()方法实际上获取 fetch()方法准备的文件和目录，并显示给用户。首先显示当前目录的名字，然后是目录和文件。因为模块允许浏览目录，就给它们加上超链接。文件不被链接，因为位于"可见但不可摸"模式。

最后，增加一个 1;以确保该模块可被成功装载。__END__标记允许我们在程序后面放置不同的注释和 POD 文档，Perl 不会抱怨。

```
1;
_ _END_ _
```

如下显示整个包内容：

Example 6-39. Apache/BrowseSee.pm

```
package Apache::BrowseSee;
use strict;

use Apache::Constants qw(:common);
use File::Spec::Functions qw(catdir canonpath curdir updir);
use File::Basename 'dirname';

sub new { bless {}, shift;}

sub handler ($$) {
    my($self, $r) = @_;
    $self = $self->new unless ref $self;

    $self->{r}        = $r;
    $self->{dir}      = $r->path_info || '/';
    $self->{dirs}     = {};
```

```
    $self->{files} = {};

    eval { $self->fetch( ) };
    return NOT_FOUND if $@;

    $self->head;
    $self->render;
    $self->tail;

    return OK;
}

sub head {
    my $self = shift;
    $self->{r}->send_http_header("text/html");
    print "<html><head><title>Dir: $self->{dir}</title><head><body>";
}

sub tail {
    my $self = shift;
    print "</body></html>";
}

sub fetch {
    my $self = shift;
    my $doc_root = Apache->document_root;
    my $base_dir = canonpath( catdir($doc_root, $self->{dir}));

    my $base_entry = $self->{dir} eq '/' ? '' : $self->{dir};
    my $dh = Apache::gensym( );
    opendir $dh, $base_dir or die "Cannot open $base_dir: $!";
    for (readdir $dh) {
        next if $_ eq curdir( );

        my $full_dir = catdir $base_dir, $_;
        my $entry = "$base_entry/$_";
        if (-d $full_dir) {
            if ($_ eq updir( )) {
                $entry = dirname $self->{dir};
                next if catdir($base_dir, $entry) eq $doc_root;
            }
            $self->{dirs}{$_} = $entry;
        }
        else {
```

```
                $self->{files}{$_} = $entry;
            }
        }
        closedir $dh;
}

sub render {
        my $self = shift;
        print "Current Directory: <i>$self->{dir}</i><br>";

        my $location = $self->{r}->location;
        print qq{<a href="$location$self->{dirs}{$_}">$_</a><br>}
            for sort keys %{ $self->{dirs} || {} };
        print qq{$_<br>}
            for sort keys %{ $self->{files} || {} };
}

1;
_ _END_ _
```

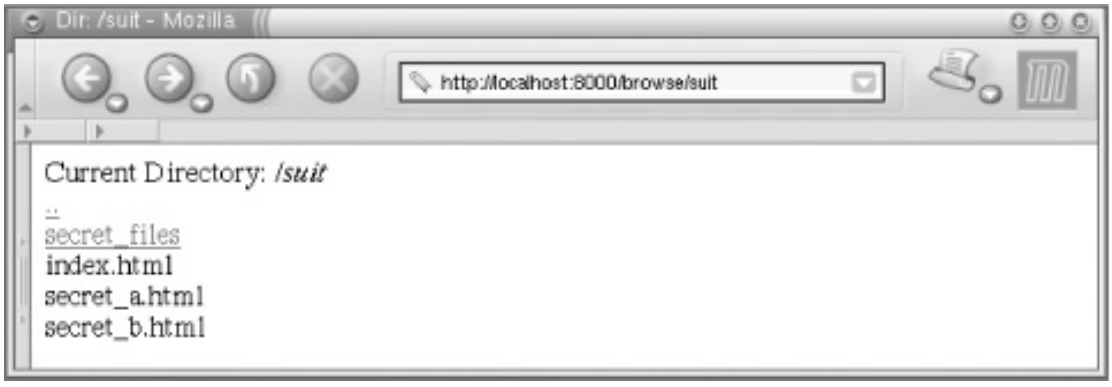
该模块存放为 Apache/BrowseSee.pm，并放在@INC 里的目录。例如，假如/home/httpd/perl 位于@INC 里，就可将它存为/home/httpd/perl/Apache/BrowseSee.pm。

为了配置模块，需要增加下列片断到 httpd.conf:

```
PerlModule Apache::BrowseSee
<Location /browse>
        SetHandler perl-script
        PerlHandler Apache::BrowseSee->handler
</Location>
```

用户访问服务器的/browse 时，可以浏览到文档根目录和下级目录的内容，但不能浏览文件的内容，如下图所示：

Figure 6-2. The files can be browsed but not viewed



现在假设该模块运行了一段时间，客户又跑回来告诉我们，他需要一个类似的应用，但文件可被浏览。这是因为后来他想仅允许授权用户阅读文件，但所有人都可看到目录内容。

我们知道这点迟早会来，还记得开头说过的吧？既然懒得完全重写代码，就需要做最少量的工作，但仍让客户满意。这次打算执行 Apache::BrowseRead 模块：

```
package Apache::BrowseRead;
use strict;
use base qw(Apache::BrowseSee);
```

将新模块存放为 Apache/BrowseRead.pm，申明新包，并使用 base 参数告诉 perl，该包继承于 Apache::BrowseSee。最后一行大概等同于：

```
BEGIN {
    require Apache::BrowseSee;
    @Apache::BrowseRead::ISA = qw(Apache::BrowseSee);
}
```

既然这个类除了呈现文件内容不同外，与 Apache::BrowseSee 做的工作差不多，所以我们要做的就是重写 render()方法：

```
sub render {
    my $self = shift;
    print "<p>Current Directory: <i>$self->{dir}</i><br>";

    my $location = $self->{r}->location;
    print qq{<a href="$location$self->{dirs}{$_}">$_</a><br>}
        for sort keys %{ $self->{dirs} || {   } };
    print   qq{<a href="$self->{files}{$_}">$_</a><br>}
        for sort keys %{ $self->{files} || {   } };
}
```

如见到的一样，这里唯一的不同是超链接了真正的文件。

然后用 1;和_ _END_ _来完成这个包：

```
1;
_ _END_ _
```

如下显示整个包：

Example 6-40. Apache/BrowseRead.pm

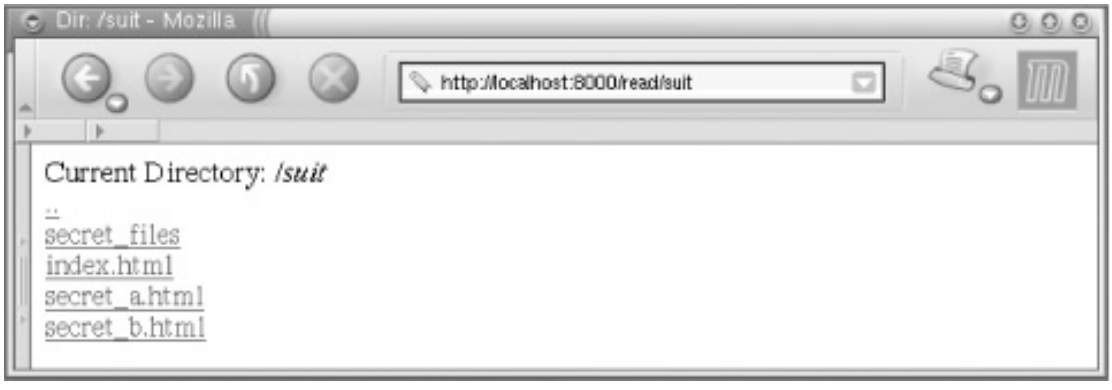
```
package Apache::BrowseRead;
use strict;
use base qw(Apache::BrowseSee);

sub render {
    my $self = shift;
    print "<p>Current Directory: <i>$self->{dir}</i><br>";

    my $location = $self->{r}->location;
    print qq{<a href="$location$self->{dirs}{$_}">$_</a><br>}
        for sort keys %{ $self->{dirs} || {} };
    print   qq{<a href="$self->{files}{$_}">$_</a><br>}
        for sort keys %{ $self->{files} || {} };
}
1;
_ _END_ _
```

最后，在 httpd.conf 里增加一个新的配置节：

```
PerlModule Apache::BrowseRead
<Location /read>
    SetHandler perl-script
    PerlHandler Apache::BrowseRead->handler
</Location>
```

现在，当通过/read 访问文件时，可浏览到文件的内容（见下图所示）。一旦我们增加一些授权/验证方式，就拥有了一个新的服务，任何人能浏览，但仅仅授权用户可阅读。

Figure 6-3. The files can be browsed and read



你可能想知道，为什么要编写一个特殊模块去做 Apache 自身已能做的事。首先，这是一个使用方法处理器的示例，所以我们尽力保持它简单但仍然展示一些真正代码。第二，该示例可被扩展--例如，它能处理虚拟文件，这些文件不存在于文件系统里，但会在线产生或从数据库里获取。最后，它可被轻易修改去做任何 Apache 不能实现，但你（或者你的客户）想

做的事。

13 参考

- "Just the FAQs: Coping with Scoping," an article by Mark-Jason Dominus about how Perl handles variables and namespaces, and the difference between `use vars( )` and `my( )`: http://www.plover.com/~mjd/perl/FAQs/Namespaces.html.
- It's important to know how to perform exception handling in Perl code. Exception handling is a general Perl technique; it's not mod_perl-specific. Further information is available in the documentation for the following modules:
  - `Error.pm`, by Graham Barr.
  - `Exception::Class` and `Devel::StackTrace`, by Dave Rolsky.
  - `Try.pm`, by Tony Olekshy, available at http://www.avrasoft.com/perl6/try6-ref5.txt.
  - There is also a great deal of information concerning error handling in the mod_perl online documentation (e.g., http://perl.apache.org/docs/general/perl_reference/perl_reference.html).
- Perl Module Mechanics: http://world.std.com/~swmcd/steven/perl/module_mechanics.html.

  This page describes the mechanics of creating, compiling, releasing, and maintaining Perl modules, which any mod_perl developer planning on sharing code with others will find useful.