

第二十章 Perl 调试器

- ↓ 第二十章 Perl 调试器
 - ↓ 20.1 使用调试器
 - ↓ 20.2 调试器命令
 - ↓ 20.2.1 部件和运行
 - ↓ 20.2.2 断点
 - ↓ 20.2.3 跟踪
 - ↓ 20.2.4 显示
 - ↓ 20.2.5 定位代码
 - ↓ 20.2.6 动作和命令执行
 - ↓ 20.2.7 杂项命令
 - ↓ 20.3 客户化调试器
 - ↓ 20.3.1 调试器的编辑器支持
 - ↓ 20.3.2 用 `init` 文件进行客户化
 - ↓ 20.3.3 调试器选项
 - ↓ 20.4 不被注意的执行
 - ↓ 调试器支持
 - ↓ 20.5.1 书写你自己的调试器
 - ↓ 20.6 Perl 调节器

首先，你试着用过 `use warnings` 用法？

如果你运行 `Perl` 的时候使用了 `-d` 开关，那么你的程序将在 `Perl` 的调试器里运行。它运行得象一个交互式的 `Perl` 环境，给你提示输入调试器命令，这样你就可以检查你的源码，设置断点，输出你的函数调用堆栈，修改变量值等等。任何调试器不能识别的命令都会当作正在被调试的包里面的 `Perl` 代码直接执行（用 `eval`）。（调试器使用 `DB` 包存储自己的状态信息，以避免破坏你的程序的状态。）这个功能非常好用，以至于有时候人们经常使用调试器来交互地测试 `Perl` 的构造。这时候，它不在乎你让 `Perl` 调试什么程序，所以我们经常选择一个没有什么意义的：

```
%perl -de 42
```

在 `Perl` 里，调试器不是一个与被调试的程序完全分离的程序，通常在一些典型的编程环境中都是这样的。与之不同的是，`-d` 标志告诉编译器向它准备交给解释器的分析树里插入源码信息。这就意味着在编译器工作之前，你的程序必须首先正确编译，如果成功，解释器预先装载一个包含调试器本身的特殊 `Perl` 库文件。

```
%perl -d /path/to/program
```

该程序会在第一个运行时可执行语句前面停止（参阅有关编译时语句的“使用调试器”一节）然后询问你输入调试器命令。当调试器停止并且显示你的一行代码的时候，它显示的是准备执行的代码，而不是刚才执行过的。

当调试器看到一行，他首先检查有没有断点，打印该行（调试器应该处于跟踪模式），执行任何动作（用 `a` 命令创建，稍后在“调试器命令”里介绍），最后如果有断点或者调试器处于单步模式则提示用户。如果没有断点，它象平时一样计算该行然后继续下一行。

20.1 使用调试器

调试器的提示符是象下面这样的东西：

```
DB<8>
```

或者是：

```
DB<<17>>
```

这里的数字显示你执行了多少命令。一个类似 **cs** 的历史机制可以让你通过数字访问前面的命令。比如 **17** 将重复命令数为 **17** 的命令。尖括号的数目表示调试器的深度。比如，如果你已经在断点了，然后又打印一个函数调用的结果，而且那个函数里面也有一个断点，那么你就会看到多于一对尖括号。

如果你想输入多行命令，比如带几个语句的子过程定义什么的，你可以用反斜杠逃逸那些通常会终止调试器的新行。下面是一个例子：

```
DB<1> for (1..3) {          cont:  print "ok\n";          cont: }
ok
ok
ok
```

假设你想在你的一个小程序上运行调试器（假设就叫 **camel_flea**），然后一碰到一个叫 **infested** 的函数就停下。下面就是方法：

```
% perl -d camel_flea
```

```
Loading DB routines from perl5db.pl version 1.07
Editor support available.
```

```
Enter h or `h h' for help, or `man perldebug' for more help.
```

```
main::(camel_flea:2):      pests('bactrian', 4);
DB<1>
```

调试器在第一个运行时可执行语句（请参考后文看看编译时语句的问题）之前停止你的程序运行，然后让你输入一条命令。同时，调试器停下来显示一行代码，它显示的是将要执行的一行，而不是刚刚执行完的那行。显示出的这行程序可能和你的源文件里的看上去不完全一样，尤其是你用预处理器对它进行过处理之后。

现在，你想让你的程序一到达 **infested** 函数就停止下来，所以你可以在那里设立一个断点，象这样：

```
DB<1>b infested
DB<2>c
```

现在调试器继续运行直到它到达该函数，在该点，调试器说：

```
DB<1> b infested
DB<2> c
```

要开一个“窗口”看看断点周围的源程序，你可以使用 **w** 命令：

```
DB<2> w
5      }
6
7      sub infested {
8==>b      my $bugs = int rand(3);
9:         our $Master;
10:        contaminate($Master);
11:        warn "needs wash"
12:         if $Master && $Master->isa("Human");
13
```

```
14:         print "got $bugs\n";
```

```
DB<2>
```

当你看到 `==>` 标记的时候，你的当前行是第八行，并且在那里的 **b** 告诉你在那个地方有一个断点。如果你已经设置了一个动作，那么那里应该也有一个 **a**。行号里带冒号的行是可以中断的；其他的则不行。

想看看是谁调用的谁，你可以用 **T** 命令看看堆栈的回溯跟踪：

```
DB<2> T
```

```
$ = main::infested called from file `Ambulation.pm' line 4
@ = Ambulation::legs(1, 2, 3, 4) called from file `camel_flea' line 5
. = main::pests('bactrian', 4) called from file `camel_flea' line 2
---
```

开头的字符（**\$**，**@** 或者 **.**）告诉你该函数分别是在标量环境，列表环境，还是空环境里调用的。这里有三行是因为当你运行堆栈回溯的时候你已经在三层函数深度了。下面是每一行的含义：

- 第一行告诉你你运行堆栈跟踪的时候正在函数 `main::infested` 里。它告诉你该函数是从文件 `Ambulation.pm` 的第四行里的一个标量环境里调用的。它还显示了调用它时没有任何参数，意思是说它是当作 `&infested` 而不是象平常那样当作 `infested()` 调用的。
- 第二行显示函数 `Ambulation::legs` 是在文件 `camel_flea` 的第五行，列表环境里调用的，带有那四个参数。
- 第三行显示 `main::pests` 是从文件 `camel_flea` 里的第二行的一个空环境里调用的。

如果你有编译阶段的执行语句，比如在 **BEGIN** 和 **CHECK** 块里的代码或者 **use** 语句，它们通常不能被调试器停止下来，不过 **require** 和 **INIT** 块可以，因为它们发生在向运行阶段转换的过程中（参阅第十八章，编译）。编译阶段的语句可以通过在 `PERLDB_OPTS` 里设置 [AutoTrace²](#) 选项来跟踪。

你可以在你的 Perl 程序本身内部对 Perl 调试器施加一些影响。比如，你可以这么做：当某个程序在调试器里运行时，在某个子过程里设置一个自动的断点。不过，在你的程序里，你可以用下面的语句把控制交还给调试器，如果调试器没有运行，它是无害的：

```
$DB::single = 1;
```

如果你把 `$DB::single` 设置为 **2**，则等效于 **n** 命令，而值为 **1** 模拟 **s** 命令。`$DB::trace` 变量应该设置为 **1** 以模拟 **t** 命令。

另外一个调试模块的方法是在装载的时候设置断点：

```
DB<7>b load c:/perl/lib/Carp.pm
Will stop on load of `c:/perl/lib/Carp.pm'.
```

然后用 **R** 命令重新启动调试器。想获得更好的控制，你可以用 **b compile subname** 命令，这样在某个子过程完成编译以后马上停止。

20.2 调试器命令

当你向调试器里键入命令的时候，你不需要用分号来结束他们。用反斜杠来接续一行（但只是在调试器里）。

因为调试器使用 **eval** 执行命令，一旦命令返回，**my**，**our**，和 **local** 设置都会消失。如果某条调试

器命令和你自己的程序里的某个函数一样，那你只需要在那个函数调用前面加上任何让它看上去不象调试器命令的东西就行了，比如一个前导的 `;` 或者一个 `+`。

如果一个调试器内建命令滚过了你的屏幕，那么你只需要在该命令的前面放一个前导的管道符号就可以了，这样它就会运行你的分页器：

```
DB<1> |h
```

调试器有很多命令，我们把它们分成（有些武断）步进和运行，断点，跟踪，显示，代码定位，自动命令执行，当然还有杂项几类。

可能最重要的命令是 `h`，它提供帮助。如果你在调试器提示符上键入 `h h`，你能得到一个精简的帮助列表，刚好一页大小，是我们特地设计成那样的。如果你键入命令 `h COMMAND`，你得到的是那条调试器命令的帮助。

20.2.1 部件和运行

调试器一行一行地运行你的程序。下面的命令让你控制什么时候跳过，什么时候停止。

- `s`
 - `s EXPR`

`s` 调试器命令单步运行程序。也就是说，调试器将执行你的下一行程序，直到到达另外一个语句，必要时步进到子过程里。如果要执行的下一行涉及函数调用，那么调试器将在那个函数的第一行停止。如果使用的 `EXPR` 里包含一个函数调用，那么它也会单步运行。

- `n`
 - `n EXPR`

`n` 命令执行子过程调用，而不用单步进入它们，直到抵达同层（或者更高层）的下一条语句的开头时停止。如果使用的 `EXPR` 里包含函数调用，那么会执行那些函数然后在每个语句前面停下来。

- 回车

如果你只是在调试器提示符上敲回车，那么重复执行前面一条 `n` 或者 `s` 命令。

- `.(点)`
- `.(点)` 命令返回指向刚刚执行完的行的调试器内部指针，并且打印该行。
- `r` 这条命令延续到当前正在执行的子过程返回。如果设置了 [PrintRet²](#) 选项，它显示返回值，这是缺省。

20.2.2. 断点

- `b`
- `b LINE`
- `b CONDITION`
- `b LINE CONDITION`
- `b SUBNAME`
- `b SUBNAME CONDITION`
- `b postpone SUBNAME`
- `b postpone SUBNAME CONDITION`
- `b compile SUBNAME`
- `b load FILENAME`

调试器命令 **b** 在 **LINE** 之前设置一个断点，告诉调试器在该点停止程序，这样你就可以检查一番。如果省略了 **LINE**，则在将要执行的行上设置断点。如果声明了 **CONDITION**，那么每次到达该语句的时候都会计算它：只有 **CONDITION** 为真的时候才会触发一个断点。你只能在一个开始一个执行语句的行上设置断点。请注意这里的条件不用 **if**：

```
b 237 $x > 30
b 237 ++$count237 < 11
b 33 /pattern/i
```

- **b SUBNAME**
 - 形式在该名字的子过程的第一行前面设置一个（可能是有条件的）断点。**SUBNAME** 可以是一个包含一个代码引用的变量；如果是这样，则不支持 **CONDITION**。

有好几种在还没有编译的代码上设置断点的方法。**b postpone** 形式在编译完 **SUBNAME** 以后，在它的第一行设置一个（可能是有条件的）断点。

- **b compile**
 - 形式在编译完 **SUBNAME** 后要执行的第一条语句上设置一个断点。请注意和 **postpone** 形式不同的是，这个语句是在有问题的子过程的外边的，因为该子过程还没有被调用，只是编译了。
- **b load** 形式在该文件的第一个执行行设置一个断点。**FILENAME** 应该象那些在 **%INC** 的值里面的文件一样是全路径名。
- **d**
 - **d LINE** 这条命令删除在 **LINE** 行的断点；如果省略 **LINE**，它删除将要执行的行的断点。
- **D** 这条命令删除所有断点。
- **L** 这条命令列出所有断点和动作。
- **c**
 - **c LINE** 这条命令继续执行，可以在声明的 **LINE** 处插入一个一次性的断点。

20.2.3 跟踪

- **T** 这条命令输出一个堆栈回溯。
- **t**
 - **t EXPR** 这条命令切换跟踪模式，调试器在运行你的程序的时候打印出每一行。参阅本章稍后讨论的 [AutoTrace²](#) 选项。如果给出了一个 **EXPR**，调试器将跟踪它的执行。又见稍后的“没人注意的执行”。
- **W**
 - **W EXPR** 这条命令加一个 **EXPR** 作为一个全局的观察表达式。（观察表达式是一个在其值改变的时候会产生断点的表达式。）如果没有给出 **EXPR**，则删除所有观察表达式。

20.2.4 显示

- Perl 的调试器有好几条命令能在你的程序在一个断点停下来时候让你检查数据结构。
- **p**
 - **p EXPR** 这条命令和当前包里的 **print DB::OUT EXPR** 一样。要注意的是，因为它只是一个 Perl 自己的 **print** 函数，所以不显示嵌套的数据结构和对象——那些场合用 **x** 命令。**DB::OUT** 句柄向你的终端打印（或者可能是一个编译器窗口），而不管标准输出重定向到哪里。

- **x**
 - **x EXPR x** 命令在列表环境里计算它的表达式并显示输出得比较漂亮的结果。也就是说，递归地打印出嵌套的数据结构并且把那些不可见字符用合适的编码输出。
- **V**
 - **V PKG**
 - **V PKG VARS** 这条命令以漂亮的格式输出所声明的 **PKG**（缺省是 **main** 包）里面的所有（或者是你声明的一部分，**VARS** 里面的）变量。散列显示它们的键字和数值，控制字符明了地显示，嵌套数据结构用清晰的风格打印出来，等等。这么做类似于在每个可用的变量上调用 **x** 命令，不同之处是 **x** 还可以显示词法变量。还有，你在这里键入标识的时候不用键入象 **\$** 或者 **@** 这样的类型标识符，象这样：

```
V Pet::Camel SPOT FIDO
```

在变量名字 **VARS** 的位置，你可以用 **~PATTERN** 或者 **PATTERN** 打印一个现有的变量，该变量名与声明的模式匹配或者不匹配。

- **X**
 - **X VARS** 这条命令和 **V CURRENTPACKAGE** 相同，而 **CURRENTPACKAGE** 是当前行编译进入的包。
- **H**
 - **H -NUMBER** 这条命令显示最后的 **NUMBER** 条命令。只有长于一个字符的命令才在历史里存储。（否则，它们中的大多数会是 **s** 或者 **n**。）如果省略 **NUMBER**，列出所有命令。

20.2.5 定位代码

在调试器里，你可以用下面的命令抽取和显示你的部分程序。

- **l**
- **l LINE**
- **l SUBNAME**
- **l MIN+INCR**
- **l MIN-MAX**
- **l** 命令列出你的程序的下面几行，或者是你给出的 **LINE** 行，或者 **SUBNAME** 子过程的头几行或者代码引用。
- **l MIN+INCR** 形式列出从 **MIN** 开始的 **INCR+1** 行。**l MIN-MAX** 形式列出从 **MIN** 到 **MAX** 的行。
- **-** 这条命令列出你的程序的前面几行。
- **w**
 - **w LINE** 列出一个在你给出的源代码行 **LINE** 周围的程序的一个窗口（几行程序），如果没有给出 **LINE** 则是当前行。
- **f FILENAME** 这条命令让你观察不同的程序或者 **eval** 语句。如果 **FILENAME** 不是象 **%INC** 里的值那样的路径全名，那调试器就把它理解为一个正则表达式来查找你指的文件。
- **/PATTERN/** 这条命令向前查找 **PATTERN**，最后的 **/** 是可选的。整个 **PATTERN** 也是可选的，如果省略，则重复前一次搜索。
- **?PATTERN?** 这条命令向后查找 **PATTERN**，最后的 **?** 是可选的，如果省略 **PATTERN** 则重复前面的查找。
- **s**
 - **s PATTERN**

- **S PATTERN S** 命令列出那些匹配（或者带 **!** 时是不匹配）**PATTERN** 的子过程名。如果没有提供 **PATTERN**，则列出所有子过程名。

20.2.6 动作和命令执行

在调试器里，你可以声明在特定时间里要做的动作。你也可以运行内部程序。

- **a**
- **a COMMAND**
- **a LINE**
- **a LINE COMMAND**

这条命令设置一个在执行第 **LINE** 行程序之前的动作，如果省略了 **LINE**，则是当前行。比如，下面这条命令每当到达第 **53** 行的时候就打印出 **\$foo**：

```
a 53 print "DB FOUND $foo\n"
```

如果没有声明 **COMMAND**，则在指定的 **LINE** 处的动作被删除。如果既没有 **LINE**，也没有 **COMMAND**，则在当前行处的动作被删除。

- **A** 调试器命令 **A** 删除所有动作。
- **<**
- **< ?**
- **< EXPR**
- **<< EXPR**
- **< EXPR** 形式声明在每次调试器提示符出现之前的要计算的一个 Perl 表达式。你可以用 **<< EXPR** 形式增加另外一个表达式，用 **< ?** 列出所有表达式，用一个 **<** 把删除所有东西。
- **>**
- **> ?**
- **> EXPR**
- **>> EXPR**
 - **>** 命令的行为类似它们的兄弟 **<**，只不过是在调试器提示符之后而不是之前执行。
- **{**
- **{ ?**
- **{ COMMAND**
- **{{ COMMAND** 调试器命令 **{** 的行为类似 **<**，不过它是声明一个要在调试器提示符之前执行的调试器命令，而不是一个 Perl 表达式。如果看上去是你不小心输入了一个程序块，那么会发出一个警告。如果你真的想输入一个程序块，用 **#{ .. }** 或者 **do { ... }** 形式写。
- **!**
- **! NUMBER**
- **! -NUMBER**
- **! PATTERN** 一个单一的 **!** 重复前面一条命令。**NUMBER** 声明执行历史里面的哪一条命令；比如，**! 3** 执行键入调试器的第三条命令。如果在 **NUMBER** 前面有一个负号，那么命令是反向计算的：**! -3** 执行倒数第三条命令。如果给出的是 **PATTERN**（没有反斜杠）而不是 **NUMBER**，那么执行以 **PATTERN** 开头的最后的一条命令。又见调试器选项 **recallCommand**。
- **!! CM**
 - 这条调试器命令在一个子过程里运行外部命令 **CMD**，它从 **DB::IN** 读入，然后写出到 **DB::OUT**。又见调试器选项 **shellBang**。这条命令使用在 **\$ENV{SHELL}** 里的 **shell**，这个 **shell** 有时候会干涉状态，信号，和核心倾倒的正确解释。如果你想要一个该命令的一致性的退出值，那么把 **\$ENV{SHELL}** 设置为 **/bin/sh**。
- **|**
- **| DBCMD**

- `|| PERLCMD`
- `| DBCMD` 命令运行调试器命令 `DBCMD`，把 `DB::OUT` 定向到 `$ENV{PAGER}`。它常用于那些产生长输出的命令，比如：
- `DB<1> |V main`
 - 请注意这条命令是用于调试器命令的，而不是那些你在 `shell` 里键入的命令。如果你想把外部命令 `who` 的输出定向到你的分页器，你可以用下面这样的命令：

```
DB<1> !!who | more
```

- `|| PERLCMD` 命令类似 `| DBCMD`，但是 `DB::OUT` 也暂时 `select` 中了，所以任何没有带文件句柄调用的 `print`，`printf`，或者 `write` 也都会被送到管道。比如，如果你有一个用 `print` 生成大量输出的函数，你可以用下面这条命令代替上面那条把输出分页显示：

```
DB<1> sub saywho { print "Users: ", `who` } DB<2> || saywho()
```

20.2.7 杂项命令

- `q` 和 `^D`
 - 这些命令退出调试器。这是我们建议的退出的方式，尽管有时候键入 `exit` 两次也可以。如果你想走出程序以后仍然留在调试器里，那么把 `inhibit_exit` 选项设置为 `0`。如果你想走过全局删除，你可能还要设置 `$DB::finished` 为 `0`。
- `R` 通过 `exec` 一个新的会话过程重起调试器。调试器会试图在两次会话间保留你的历史，不过有时候可能会丢失内部设置和命令行选项。目前，下面的设置是保留的：历史，断点，动作，调试器选项和 Perl 命令行选项 `-w`，`-I` 和 `-e`。
- `=`
- `= ALIAS`
- `= ALIAS VALUE` 如果没有给出 `VALUE`，那么这条命令打印当前的 `ALIAS` 的值。如果有 `VALUE`，那么它用 `ALIAS` 的别名定义一条新的调试器命令。如果 `ALIAS` 和 `VALUE` 都忽略了，列出当前所有别名。比如：
- `= quit q`
 - 所有 `ALIAS` 应该都是简单的标识，并且也应该转换成简单的标识符。你可以通过向 `%DB::aliases` 目录增加自己的记录来实现更复杂的别名。参阅本章稍后的“客户化调试器”。
- `man`
 - `man MANPAGE` 这条命令在给出的分页器上调用你的系统的缺省文档阅读器，如果 `MANPAGE` 省略，则是阅读器本身。如果该阅读器是 `man`，那么用当前的 `%Config` 信息调用它。必要时，调试器会自动为你加上“perl”前缀；这样就让你可以在调试器里键入 `man debug` 和 `man op`。

在那些通常没有 `man` 工具的系统上，调试器调用 `perldoc`；如果你想修改这个性质，把 `$DB::doccmd` 设置为你喜欢的阅读器。这些设置可以放在一个 `rc` 文件里或者用直接赋值的方法来实现。

- `O`
- `O OPTION ...`
- `O OPTION? ...`
- `O OPTION=VALUE...`
- `O` 命令让你操作调试器选项，这些选项在本章稍后的“调试器”选项中列出。
 - `O OPTION` 形式把每个列出的调试器选项设置为 `1`。如果在 `OPTION` 后面跟着一个问号，那么显示它的当前值。
- `O OPTION=VALUE` 形式设置选项的值；如果 `VALUE` 里面有一个空白，那么你应该用引号把数值包围起来。比如，你可以设置
- `O pager="less -MQeicsNfr"` 来利用带那些指定的标志的 `less`。你可以用单引号或者双引

号，不过不管你用哪个，你都应该逃逸选项值里面的同样的引号。你还要逃逸那些在引号前面，但并不是用于逃逸引号的反斜杠。换句话说，不管用的是什么引号，只要遵循单引号规则就可以了。调试器的响应是显示你刚设置的值，它自己的输出总是使用单引号：

```
DB<1> O OPTION='this isn\'t bad'
        OPTION = 'this isn\'t bad'

DB<2> O OPTION="She said, \" Isn't it?\""
        OPTION = 'She said, "Isn\'t it?"'
```

由于历史原因，`=VALUE` 是可选的，但是只是对那些可以为 `1` 的才缺省为

1. `--`也就是说，那些布尔选项。最好给用 `=` 给选项赋一个特定的值 `VALUE`。 `OPTION` 可以缩写，不过除非你想在内部保密，否则不应该这么干。你可以一次设置好几个选项，参阅“调试器选项”获取选项列表。

20.3 客户化调试器

调试器包含很多配置挂钩，你可能永远不用自己修改这些东西。你可以在调试器里面用 `O` 命令修改调试器的行为，也可以在命令行上通过 `PERLDB_OPTS` 环境变量来做，或者是用存放在 `rc` 文件里的预设命令来实现。

20.3.1 调试器的编辑器支持

调试器的命令行历史机制不象许多 `shell` 那样提供命令行编辑功能：你不能用 `^p` 检索以前的行，或者用 `^a` 移动到行的开头，尽管你可以用类似与 `shell` 用户的叹号来执行前面的行。不过，如果你安装了 CPAN 里的 `Term::ReadKey` 和 `Term::ReadLine` 模块，你就拥有了类似 GNU `readline` (3) 提供的所有功能。

如果你在你的系统上安装了 `emacs`，它可以与 Perl 调试器交互地工作，这样就给你提供一个类似它给 C 调试器提供的那样的集成开发环境。Perl 带着一个启动文件，可以令 `emacs` 能够理解 Perl 的语法，表现得象一个面向语法的编辑器。请查看一下 Perl 源程序目录的 `emacs` 目录。使用 `vi` 的用户也可以看看 `vim`（和 `gvim`，用得最多的版本），获取 Perl 关键字的彩色显示。

还有一个我们做的（Tom）类似的设置文件，可以用于任何厂商提供的 `vi` 和 `X11` 窗口系统。它运行起来很象 `emacs` 提供的多窗口支持，它是调试器驱动编译器。不过，在我们写这些的时候，它在 Perl 发布的最终位置还不明确。不过我们认为你应该知道这是可能的。

20.3.2 用 `init` 文件进行客户化

你可以通过设置 `.perldb` 或者 `perldb.ini` 文件（用哪个取决于你的操作系统）来做一些客户化工作。该文件里包含初始化代码。这个初始化文件包含的是 Perl 代码，不是调试器命令，并且是在查看 `PERLDB_OPTS` 环境变量之前处理。比如，你可以用下面的方法在 `%DB::alias` 散列里增加记录来制作别名：

```
$alias{len} = 's/^len(.*)/p length($1)/';
$alias{stop} = 's/^stop (at|in) /b/';
$alias{ps} = 's/^ps\b/p scalar /';
$alias{quit} = 's/^quit(\s*)/exit/';
$alias{help} = 's/^help\s*$/|h/';
```

你可以在你的初始化文件里使用函数调用调试器内部的 `API` 修改那些选项：

```
parse_options("NonStop=1 LineInfo=db.out AutoTrace=1 frame=2");
```

如果你的初始化文件定义了子过程 **afterinit**，则该函数是在调试器初始化结束以后调用的。**init** 文件可以位于当前目录或者家目录。因为这个文件包含任何 **Perl** 命令，出于安全原因，它必须为超级用户或者当前用户所有，并且只有它的所有者有写权限。

如果你想修改调试器，把 **Perl** 库目录的 **perl5db.pl** 拷贝成另外一个名字然后把它修改为你的所需。然后你就需要把你的 **PERL5DB** 环境变量设置为下面这样的东西：

```
BEGIN {require "myperl5db.pl" }
```

最后，你可以用 **PERL5DB** 直接设置内部变量或者调用内部调试器函数来客户化调试器。不过，要小心的是，任何没有在这里或联机的 **perldebug**，**perldebuguts**，或者 **DB** 手册页里提到的变量和函数都被认为是只用于内部目的并且容易在不知觉的情况下被修改。

20.3.3 调试器选项

调试器有无数的选项，你可以用 **O** 命令设置它们，可以通过交互地调用 **O** 命令或者从环境变量设置或者用一个 **init** 文件设置。

```
recallCommand, ShellBang
```

用语回忆一条命令或者派生一个 **shell** 的字符。缺省时，两个都设置为 **!**。

- **pager**
 - 用于输出的分页管道命令（那些以 **|** 字符开头的命令）。缺省时，使用 **\$ENV {PAGER}**。因为调试器使用你的当前终端性质做宽字符和下划线处理，如果你选择的分页器不能透明地传递逃逸序列，那么一些调试器命令的输出在送给分页器以后将无法阅读。
- **tkRunnking** 在显示提示符的时候运行在 **Tk** 模块下（有 [ReadLine²](#)）。
- **signalLevel**, **warnLevel**, **dieLevel** 设置冗长级别。缺省时，调试器不处理你的例外和警告，因为处理它们可能破坏正确运行的程序。

要关闭这个缺省的安全模式，把这些值设置得比 **0** 大。在级别 **1**，你得到 包括任何类型的警告（通常很讨厌）或例外（通常很有用）的回溯。糟糕的是，调试器无法区分致命错误和非致命错误。如果 **dieLevel** 为 **1**，那么 调试器还会跟踪你的非致命例外，并且如果它们来自 **eval** 的字串或者来自 你准备装载的模块里面的任何 **eval**，那么它们会被调试器随意修改。如果 **dieLevel** 为 **2**，则调试器不关心它们来自何方：它篡夺你的例外句柄然后 打印一个回溯，然后用它自己的方法修改所有例外。这个特性可能对某些跟踪 用途有用，不过可能会把任何认真对待例外处理的程序搞得混乱不堪。

如果调试器捕获到任何 **INT**，**BUS**，或者 **SEGV** 信号，那么它会试图打印一条 消息。如果你正在一个慢速的系统调用里（比如 **wait** 或者 **accept**，或者从 你的键盘或者套接字里的 **read**），而且没有设置你自己的 **\$SIG{INT}** 句柄， 那么你就无法 **Control-C** 退回到调试器里，因为调试器自己的 **\$SIG{INT}** 句柄并不知道它要抛出一个例外以 **longjmp(3)** 跳出慢速的系统调用。

- [AutoTrace²](#) 设置跟踪模式（类似 **t** 命令，但是可以放到 **PERLDB_OPTS**里）。
- [LineInfo²](#) 给出打印行数信息的文件或者管道。如果它是一个管道（比如，**|Visual_perl_db**），则使用一条短消息。这是用于与从编辑器或者可视化调试器交互的机制，比如特殊的 **vi** 或者 **emacs** 挂钩，或者 **ddd** 图形调试器。
- **inhibit_exit** 如果为 **0**，允许走出程序的结束。
- [PrintRet²](#) 在设置了 **r** 命令以后（缺省），打印返回值。
- **ornaments** 影响该命令行的屏幕显示效果（参阅 **Term::ReadLine** 的联机文档）。目前没

有办法关闭装饰处理，这样的话一些经过处理的输出可能对于某些显示器或者某些分页器来说是非法的。我们认为这是一个臭虫。

- **frame** 影响进入或者离开子过程时打印的消息。如果 **frame & 2** 为假，那么只有进入子过程时才打印消息。（如果和其他消息混合在一起，离开子过程时的打印可能是有用的。）

如果 **frame & 4**，打印函数的参数，以及环境和调用者信息。如果 **frame & 8**，那么打印参数里也会有重载的 **stringify** 和 **tie** 过了的 **FETCH**。如果 **frame & 16**，还打印子过程的返回值。

参数列表的截断长度由下一个选项控制。

- **maxTraceLen** 如果设置了 **frame** 选项的第四位，这个选项就是截断参数列表的长度。

下面的选项影响 **V**，**X** 和 **x** 命令的使用：

- **arrayDepth**, **hashDepth** 只打印带头 **n** 个元素。如果省略了 **n**，打印所有元素。
- **compactDump**, **veryCompact** 修改数组和散列的输出风格。如果打开 **compactDump**，短的数组可能可以用一行打印出来。
- **globPrint** 打印类型团的内容。
- [DumpDBFiles²](#) 显示持有调试过的文件的数组。
- [DumpPackages²](#) 显示包的符号表。
- [DumpReused²](#) 显示“reused”地址的内容。
- **quote**, [HighBit²](#), **undefPrint** 修改字符串显示的风格。**quote** 的缺省值是 **auto**；你可以通过把它设置为 **"** 或 **'** 分别打开双引号或者单引号风格的格式。缺省时，设置了高位的字符是逐字打印的。
- [UsageOnly²](#) 打开这个选项以后，调试器就不再显示一个包的变量的内容了，而是显示基于在包的变量中找到的字符串的总尺寸输出的一个每个包的基本内存使用信息。因为这里用的是包的符号表，所以词法范围的变量被忽略。

20.4 不被注意的执行

在启动的过程中，选项是从 **\$ENV{PERLDB_OPTS}** 里初始化的。你可以在那里放上初始化选项 **TTY**，**noTTY**，**ReadLine**，和 [NonStop²](#)。

如果你的初始化文件包含：

```
parse_options( "NonStop=1 LineInfo=tperl.out AutoTrace");
```

那么你的程序就会在没有人的干涉的情况下运行，把跟踪信息放到 **db.out** 文件里。（如果你中断了调试过程，如果你还想看见什么东西的话，那你最好把 [LineInfo²](#) 重新设置为 **/dev/tty**。）

下面的选项只能在启动的时候声明。如果你要在初始化文件里设置它们，调用 **parse_options ("OPT=VAL")**。

- **TTY**
 - 用于调试 I/O 的终端。
- **noTTY**
 - 如果设置了，调试器进入 [NonStop²](#) 模式并且不会连接到一个终端。如果调试器被中断（或者控制通过明确地 **\$DB::signal** 或者 **\$DB::single** 设置回到了调试器），那么调试器与启动时声明的 **TTY** 选项连接，或者连接到一个你在运行时用

Term::Rendezvous 模块选取的终端。

这个模块应该实现一个叫 `new` 的方法，它返回带两个方法的对象：`IN` 和 `OUT`。这些方法应该为调试器返回文件句柄分别用于输入和输出。`new` 方法 在启动的时候应该检查包含 `$ENV{PERLDB_NOTTY}` 的参数，或者是 `"/tmp/perldebugtty$$"`。调试器并没有检查这个文件的所有权和是否放开了 写权限，所以理论上是有安全风险的。

- [ReadLine²](#) 如果为假，在调试器里的 [ReadLine²](#) 支持将被关闭，这样才便于调试器那些本身带有 [ReadLine²](#) 模块的程序。
- [NonStop²](#) 如果设置了这个选项，调试器进入非交互模式，直到中断，或者是你的程序设置了 `$DB::signal` 或 `$DB::single`。

有时候选项可以用第一个字母的缩写唯一地标识，不过我们建议你使用全称，主要是为易读性和将来的兼容性考虑。

下面是一个利用 `PERLDB_OPTS` 环境变量自动设置选项的例子。（注：我们这里用的是 `sh` 语法来设置环境变量。使用其他 `shell` 的读者应该做相应的调整。）这个例子以非交互方式运行你的程序，为每个子过程的入口和执行的每一行打印一条信息。调试器的跟踪输出放到文件 `tperl.out`。这样可以让你的程序仍然使用正常的标准输入和输出，而不用担心在中间混杂有跟踪信息。

```
$ PERLDB_OPTS="NonStop frame=1 AutoTrace2 LineInfo2=tperl.out" perl -d myprog
```

如果你中断了该程序，你需要循序重置为 `O LineInfo2=/dev/tty` 或者是你的平台相关的设备。否则，你将看不到调试器的提示符。

调试器支持

Perl 为在编译器或者运行时创建象标准调试器这样的调试环境提供了特殊的调试挂钩(hook)。不过不要把这些挂钩和 `perl -D` 选项混淆，`-D` 选项只有在你的 Perl 是带 `-DDEBUGGING` 支持编译的时候才可以用。

比如，当你从 `DB` 包里调用 Perl 内建的 `caller` 函数的时候，调用时对应的堆栈帧被拷贝到 `@DB::args` 数组。当你带着 `-d` 开关调用 Perl 时，同时还打开了下面这几项附加的特性：

- Perl 在你的程序的第一行之前插入 `$ENV{PERL5DB}`（如果没有这个变量，则是 `BEGIN {require 'perl5db.pl'}`）的内容。
- 数组 `@{"_<$filename"}` 里为Perl 编译的所有文件保存有 `$filename` 里所有的行。对那些包含子过程的 `eval` 的字串或者当前正在执行的子过程也做这个处理。`eval` 过的字串看起来象 `(eval 34)`。在正则表达式里的代码断言看起来象 `(re_eval 19)`。
- 散列 `%{"_<$filename"}` 包含用行号做键字的断点和动作。你可以设置个别记录，也可以设置整个散列。在这里，Perl 只关心布尔真值，尽管 `perl5db.pl` 使用的值形如 `"$break_condition\0$action"`。在这个散列里的值在数字环境里有特殊含义：如果该行是不可中断的，那么它们是零。`
` 计算包含子过程的字串或者当前正在执行的子过程的时候也会存储这些数据。`eval` 出来的字串的 `$filename` 看起来象 `(eval 34)` 或者 `(re_eval 19)`。
- 标量 `${"_<$filename"}` 包含 `"_<$filename"`。计算包含子过程的字串或者当前正在执行的子过程的时候也是如此。`eval` 出来字串的 `$filename` 看起来象 `(eval 34)` 或者 `(re_eval 19)`。
- 在每个 `require` 的文件都编译之后，但在它们执行之前，如果存在子过程 `DB::postponed`，则执行 `DB::postponed(*{"_<$filename"})`。`
` 这里，`$filename` 是 `require` 的文件的扩展过后的名字，象那些在 `%INC` 里的一样。

- 在每个子过程 `subname` 编译完成之后，检查 `$DB::postponed{subname}` 是否存在。如果该键字存在，如果还存在 `DB::postponed` 子过程则调用 `DB::postponed(subname)`。
- 调试器维护一个 `%DB::sub` 散列，其键字是子过程名而其值的形式是 `filename:startline-endline`。`filename` 对于那些定义在 `eval` 里面的子过程，形如 `(eval 34)`，或者对那些在正则表达式代码断言里的形如： `(re_eval 19)`。
- 当你的程序的执行到达一个可能有断点的位置时，如果变量 `$DB::trace`，`$DB::single`，或者 `$DB::signal` 中的任何一个为真，则调用 `DB::DB()` 子过程。这些变量是不能 `local` 化的。如果在 `DB::DB()` 内部执行，这个特性被关闭，包括那些在它内部调用的函数，除非 `$^D & (1<<30)` 保存真值。
- 如果程序的执行到达一个子过程调用，则用一个 `&DB::sub(args)` 的调用取代，这里 `$DB::sub` 保存被调用子过程的名字。如果该子过程是在 `DB` 包里编译的，那么不会发生这个动作。

请注意，如果 `&DB::sub` 需要需要外部数据来允许，那么在它完成之前，不可能有其他子过程调用。就标准调试器而言，`$DB::deep` 变量（你在强制的中断之前最多可以递归调用调试器的深度）给出了这么一个依赖性的例子。

20.5.1 书写你自己的调试器

最小的可以用的调试器包含一行：

```
sub DB::DB {}
```

这行代码因为没有做任何事情，因此可以很容易地通过 `PERL5DB` 环境变量定义：

```
% PERL5DB="sub DB::DB {}" perl -d your-program
```

另外一个微型调试器，稍微有点用，可以象下面这样创建：

```
sub DB::DB {print ++$i; scalar }
```

这个小调试器将在每个它碰到的语句前面打印一个序列数，然后在继续之前等待你输入一个回车。

下面的调试器，尽管看起来很小，但是相当有用：

```
{
    package DB;
    sub DB {}
    sub sub {print ++$i, " $sub\n"; &$sub}
}
```

它打印子过程调用的序号和子过程名字。请注意 `&DB::sub` 必须从 `DB` 包中编译，就象我们这里做的那样。

如果你以现有的调试器做你的新调试器，那么有一些挂钩可以帮助你客户化它。在启动的时候，调试器从当前目录或者你的家目录读取你的 `init` 文件。读完文件之后，调试器读取 `PERLDB_OPTS` 环境变量然后把它们当作一个 `O ...` 行的剩余部分——就象你可能在调试器提示符键入的那样。

调试器还维护内部特殊变量，比如 `@DB::dbline`，`%DB::dbline`，它们是 `@{":::<current_file"} %{":::<current_file"}` 的别名。这里 `current_file` 是当前选择的文件，可能是用调试器的 `f` 命令明确选择的或者是执行流隐含地选择的。

有些函数也可以帮助你客户化。`DB::parse_options(String)` 把一行象 `O` 选项那样分析。

`DB::dump_trace(SKIP[, COUNT])` 忽略里面声明的数目的（堆栈）帧并且返回包含有关调用帧的

信息列表。（如果没有 `COUNT`，就是所有帧）。每条记录都是一个指向散列的引用，这个散列有键字“`context`”（可以是 `.`, `$`, 或者 `@`），“`sub`”（子过程名或者有关 `eval` 的信息），“`args`”（`undef` 或者一个指向一个数组的引用），“`file`”，和“`line`”。`DB::print_TRACE(FH, SKIP[, COUNT[, short]])` 向你提供的文件句柄打印有关调用帧的格式化信息。最后两个函数可以方便地作为调试器的 `<` 和 `<<` 命令的参数。

你用不着学习所有的这些东西——我们中的大多数都用不着。实际上，如果我们需要调试一个程序，我们通常只是在程序里的一些位置插入几条 `print` 语句，然后再运行程序就可以了。

更好的情况下，我们甚至可以先打开警告。通常这样就足以定位问题的所在，这样就可以省下许多工作。就算这样也无法找出错误，我们还可以告诉你除了 `-d` 开关以外，还有一个很不错的调试器可以为你缝补几乎所有漏洞——除了不能替你找错误以外。

不过如果你想记住有关客户化调试器的某件事情的话，那么这件事情可能是：不要把臭虫局限于那些让 `Perl` 不正常的东西。如果你的程序让你觉得难受，那么这也是一只臭虫。早些时候，我们给你显示了几个非常简单的客户化调试器。在下一节里，我们将向你演示另外一种客户化调试器的例子，一个可能会（或者也可能不会）帮助你调试那些被认为是“这东西有完没完？”的臭虫。

20.6 Perl 调节器

你想让你的程序运行得快些吗？你当然想。不过，你首先得停下来问问自己，“我真的需要花时间让这东西跑的快些吗？”休闲性的优化是比较有趣的工作，（注：`Nathan Torkington` 说的，这节是他写的。）不过通常你有使用你的时间的更好的地方。有时候你只需要预先计划好然后当你去喝茶的时候再运行程序。（或者把运行程序当作去喝茶的借口。）但是如果你的程序绝对需要运行得更快，那你就开始调节它了。调节器可以告诉你你的程序哪一部分花了最多的时间运行，这样你就不用花太多时间在优化一个对总运行时间作用很小的子过程上。

`Perl` 带着一个调节器，`Devel::DProf` 模块。你可以键入下面的命令调节在 `mycode.pl` 里的 `Perl` 程序：

```
perl -d:DProf mycode.pl
```

虽然我们管它叫调节器——它的作用就是调节器，但 `DProf` 采用的机制和我们在本章早些时候讨论的内容完全一样。`DProf` 只是一个记录 `Perl` 进入和离开每个子过程的时间的调试器。

当你调节的脚本结束的时候，`DProf` 会向一个叫 `tmon.out` 的文件倾倒计时信息。和 `Perl` 一起发布的 `dprofpp` 程序知道如何分析 `tmon.out` 以及产生输出。你还可以把带 `-p` 开关的 `dprofpp` 用做整个处理过程的前端（见稍后的描述）。

给出下面程序：

```
outer();

sub outer {
    for (my $i=0; $i < 100; $i++) { inner () }
}

sub inner {
    my $total = 0;
    for (my $i=0; $i < 1000; $i++) { $total += $i }
}

inner();
```


dprofpp 的输出是:

```
Total Elapsed Time = 0.537654 Seconds
  User+System Time = 0.317552 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 85.0   0.270   0.269    101   0.0027 0.0027  main::inner
 2.83   0.009   0.279      1    0.0094 0.2788  main::outer
```

请注意这里的百分数加起来不等于 **100**。实际上，在这个例子里，它们离 **100** 差得很远，这样就给你一个暗示说你应该运行这段程序时间长一点。一个通用的规则就是，你能收集到的调节数据越多，你的得到的采样统计就越好。如果我们把外层循环增加到运行 **1000** 次而不是 **100** 次，我们就能得到更准确的结果：

```
Total Elapsed Time = 2.875946 Seconds
  User+System Time = 2.855946 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 99.3   2.838   2.834   1001   0.0028 0.0028  main::inner
 0.14   0.004   2.828      1    0.0040 2.8280  main::outer
```

第一行告诉你程序从开始到结束运行了多长时间。第二行显示了两个不同数据的总和：花在执行你的代码（“用户”）的时间和花在你的代码做的系统调用（“系统”）里的时间。（在这里我们必须容忍一点点精度上的错误——计算机的时钟几乎肯定不是以百万分之一秒记的。运气好的话可能是以百分之一秒记。）

你可以用 **dprofpp** 的命令行选项修改“用户+系统”时间。**-r** 显示总时间，**-s** 只显示系统时间，而**-u** 只显示用户时间。

剩下的报告就分解成每个子过程运行的时间。“**Exclusive Time**”行表示当子过程 **outer** 调用子过程 **inner** 的时候，花在 **inner** 上的时间没有算做 **outer** 花的时间。要改变这个模式，让 **inner** 花的时间算在 **outer** 花的时间之内，给 **dropp** 一个 **-I** 选项。

对每个子过程，都报告如下内容：**%Time**，在这个子过程调用花的时间的百分比；**ExclSec**²，在这个子过程上花的时间（以秒计，不包括它里面调用子过程花的时间）；**CumulS**²，在这个子过程上花的时间（以秒计，包括它里面调用子过程花的时间）；**#Calls**，调用该子过程的次数；**sec/call**，平均每次调用该子过程花费的时间（以秒计，不包括它里面调用子过程花的时间）；**Csec/c**，平均每次调用该子过程花费的时间（以秒计，包括它里面调用子过程花的时间）；

当然，最有用的数据是 **%Time**，它告诉你你的时间花在什么地方了。在我们的例子里，**inner** 子过程花的时间最多，所以我们可以尝试优化该子过程，或者找一个能够减少调用它的算法。:-)

dprofpp 的选项给你提供了访问其他信息的机会，或者修改时间计算的方法。你还可以让 **dprofpp** 先替你运行脚本，这样你就用不着老是记着用 **-d:DProf** 开关：

- **-p SCRIPT**
告诉 **dprofpp** 它应该调节给出的 **SCRIPT**，然后解释它的调节数据。又见 **-Q**。
- **-Q**
和 **-p** 一起使用，告诉 **dprofpp** 在调节完脚本后退出，而不用解释调节数据。
- **-a**
把输出按照子过程名字的字母顺序排序，而不是按照时间百分比降序排列。
- **-R**

独立地计算在同一个包里定义的匿名子过程。缺省的做法是把所有匿名子过程当作一个对待，命名为 `main::__ANON__`。

- **-I**
显示包含子过程开销的所有子过程时间。
- **-l**
按照调用子过程的次数排序。这样可以帮助标识可以内联的子过程。
- **-O COUNT**
只显示前 `COUNT` 个子过程，缺省是 `15`。
- **-q**
不显示字段头。
- **-T**
在标准输出上显示子过程调用树。不显示子过程统计。
- **-t**
在标准输出上显示子过程调用树。不显示子过程统计。在同一个调用级别调用了多次的函数只显示一次，和一个重复计数。
- **-S**
按照你的子过程相互调用的结构生成输出：

```
main::inner x 1          0.008s
main::outer x 1          0.467s = (0.000 + 0.468)s
main::inner x 100 0.468s
```

按照下面的文字理解上面的输出：你的程序的顶级调用了一次 `inner`，然后它运行了 `0.008` 秒，然后顶级调用了一次 `outer`，而它运行了 `0.467`（`0`秒 `outer` 本身，`0.468`秒`outer`里的子过程调用）秒，包括调用 `inner` `100` 次（花了 `0.468`秒）。明白了？

在同一级的分支里（比如，`inner` 调用了一次并且 `outer` 调用了一次）是按照包含的时间排序的。

- **-U**
不排序。显示裸调节文件里的顺序。
- **-V**
按照花在每次子过程调用的平均时间排序。这样有助于标识通过内联子过程体的手工优化。
- **-g subroutine**
忽略除了 `subroutine` 以及从它里面调用的所有子过程。

其他选项在 `dprofpp(1)`，它的标准手册页里描述。

`DProf` 并不是你的调节器的唯一选择。`CPAN` 还有 `Devel::SmallProf`，它可以汇报你的程序里每一行花费的时间。它可以帮你找出某些有着奇怪开销的 `Perl` 构造。大多数内建函数都相当高效，但是我们很容易不小心写了一个开销随着输入成指数增长的正则表达式。参阅第二十四章，常用练习，里的“效率”一节获取其他提示。

现在我们可以去喝杯茶了。你得看下一章了。

Revision: r1.5 - 09 Jan 2006 - 07:52 - [TingYu](#)

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

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