Debugger

Einführung in den Perl Debugger

Real Programmers don‘t need debuggers,
they can read core dumps.

Larry Wall

Viele Perl-Programmierer mögen und nutzen den Perl-Debugger aus unterschiedlichsten Gründen nicht. Auch ich habe mich jahrelang gesträubt, den Perl-Debugger zu benutzen. Wenn man jedoch einige Male mit dem Debugger gearbeitet hat, wird man ihn gerne in seinen Werkzeugkasten aufnehmen.

Der nachfolgende Artikel soll einen einfachen Einstieg in den Perl-Debugger ermöglichen, indem die wichtigsten grundlegenden Kommandos und Tricks Schritt für Schritt erläutert werden.

Features oder Was kann der Debugger?

Der Perl-Debugger beherrscht die typischen Debuggerfeatures wie Haltepunkte (Breakpoints), Überprüfung von Variablen, Methoden, Code und Klassenhierachien, Verfolgung (Trace) der Code-Ausführung und Subroutinen-Argumente.

Zusätzlich erlaubt der Debugger die Ausführung eigener Befehle (Perl-Code) vor und nach jeder Programmcode-Zeile und die Änderung von Variablen und Code während der Ausführung.

Die Verfolgung geforkter Programme, Threads und remote debugging wird ebenfalls unterstützt.

Der Perl-Debugger kann in den Apache Web Server (mod_perl) integriert werden und es gibt natürlich auch graphische Benutzeroberflächen.

Bevor man den Debugger anwirft

If debugging is the process of removing bugs,
then programming must be the process of putting them in.

Edsger W. Dijkstra

Viele Bugs lassen sich durch einen defensiven Programmierstil vermeiden. Ein gutes Perl-Programm verwendet daher `strict` und `warnings`.

Debugger Starten

Der Perl-Debugger wird stets mit dem Kommandozeilenschalter `-d` aufgerufen.

Parameter, die an das zu debuggende Programm übergeben werden, können auf der Kommandozeile angegeben werden.

```
perl -d programm [arg0 arg1 ...]
```

Dies gilt auch für CGI-Programme, hier entsprechen die Parameter den Formularfeldern.

```
perl -d programm.cgi [param1= param2=]
```

Der Debugger kann auch als Perl-Shell gestartet werden.

```
perl -d -e 0
```

Erste Schritte - Der Debugger als Perl-Shell

Für die ersten Schritte mit dem Perl-Debugger bietet sich die Nutzung als Perl-Shell an.

```
perl -d -e 0
... Ausgabe unterdrückt
main::(-e:1):    0
  DB<1>
```

Der Debugger meldet sich mit einem eigenem Prompt (`DB<n>`) und wartet auf die Eingabe eines Kommandos. Zum Beenden des Debuggers dient das Kommando `q`, Hilfe erhält man über das Kommando `h`.

Code eingeben

Nun kann beliebiger Perl-Code eingeben werden.

```
perl -d -e 0

  DB<1> print "Hallo Foo-Magazin"
Hallo Foo-Magazin

  DB<2> print 6 * 7; print "\n"; print 6 x 7
42
6666666
```

Das Zeilenende wird als Anweisungsende aufgefasst, das übliche Semikolon am Zeilenende kann daher entfallen. Mehrere Anweisungen in einer Zeile müssen allerdings durch ein Semikolon getrennt werden.

Continuation Lines (Fortführungszeilen)

Falls mehr als eine Zeile zur Eingabe des Codes benötigt wird, muss das Zeilenende mit einem \ maskiert werden.

```
  DB<1> foreach my $number ( 1 .. 2 ) { \
  cont:    print "$number\n" \
  cont: }
1
2
```

Variableninhalte ausgeben mit `p`

Das Kommando `p` erlaubt die einfache Ausgabe der Werte von Variablen.

```
  DB<1> $scalar = "Text"
  DB<2> @array = qw(eins zwei)
  DB<3> %hash = ( 'a' => '1', 'b' => '2')
  DB<4> p $scalar
Text
  DB<5> p @array
einszwei
  DB<6> p %hash
a1b2
```

Datenstrukturen und Variableninhalte mit `x` anzeigen

Die Ausgabe von `p` ist für skalare Variablen durchaus brauchbar. Eine übersichtlichere Darstellung lässt sich mit dem Kommando `x` erzeugen.

Einfache Variablen und Datenstrukturen

Betrachten wir zunächst einmal die grundlegenden einfachen Perl-Datenstrukturen.

Das Kommando `x` evaluiert seine Parameter im Listenkontext.

```
  DB<7> x $scalar
0  'Text'
```

Bei skalaren Variablen wird eine Liste mit einem Element zurückgegeben.

```
  DB<8> x @array
0  'eins'
1  'zwei'
```

Bei Array-Variablen wird eine Liste der Elemente mit dem jeweiligen Index zurückgegeben.

```
  DB<9> x %hash
0  'a'
1  1
2  'b'
3  2
```

Auch bei Hashes wird eine Liste mit den entsprechenden Indizes ausgegeben.

Wenn man jedoch dem Kommando `x` eine Referenz auf eine Variable übergibt, wird die Ausgabe wesentlich informativer.

```
  DB<10> x \$scalar
0  SCALAR(0x8427f00)
   -> 'Text'

  DB<11> x \@array
0  ARRAY(0x8427e1c)
   0  'eins'
   1  'zwei'

  DB<12> x \%hash
0  HASH(0x8427d38)
   'a' => 1
   'b' => 2
```

Komplexe Datenstrukturen betrachten

Das Kommando `x` eignet sich weiterhin sehr gut zum Betrachten komplexer Datenstrukturen.

Beispiel: Hash of Arrays (HoA)

```
  DB<1>  %HoA = ( \
  cont: flintstones => [ "fred", "barney" ],\
  cont: jetsons => [ "george", "jane", \
                       "elroy" ],)

  DB<2> x \%HoA
0  HASH(0x8427de0)
   'flintstones' => ARRAY(0x81547bc)
   0 'fred'
      1  'barney'
   'jetsons' => ARRAY(0x8427d50)
      0  'george'
      1  'jane'
      2  'elroy'
```

Ich benutze `x` sehr gerne beim Einlesen von Daten aus Dateien, indem ich durch das Programm steppe (s.u.) und mir nach jedem eingelesenem Datensatz anzeigen lasse, ob die Daten auch korrekt und vollständig in die Datenstruktur übernommen werden.

Module, Packages und Klassen inspizieren

Geladene Module M

Das Kommando `M` zeigt alle geladenen Module, auch die Pakete, die vom Debugger oder anderen geladenen Modulen geladen werden, an.

```
 DB<1> M
'Carp.pm' => '1.04
     from /usr/share/perl/5.8/Carp.pm'
... Ausgabe gekürzt
```

Angezeigt werden neben dem geladenen Modul die Versionsnummer und auch der Ort von dem das Paket geladen wurde. Das erleichtert die Suche nach Fehlern, die auf unterschiedlichen Modulversionen auf dem Entwicklungs- und Produktionsserver beruhen erheblich.

Inheritance i

Das Kommando `i` zeigt den Vererbungsbaum (@ISA) für geladene Pakete an.

```
  DB<1> use FileHandle

  DB<2> i FileHandle
 FileHandle 2.01, IO::File 1.13, IO::Handle
1.25, IO::Seekable 1.1, Exporter 5.58
```

Dabei geht der Debugger rekursiv durch die Pakete, die vom untersuchten Paket geladen werden.

Ein Perl-Programm im Debugger

Beispielprogramm

Für die nachfolgenden Übungen wird ein kleines Beispielprogramm (`example.pl`) verwendet.

```
1 #!/usr/bin/perl
2 use warnings;
3 use strict;
4 foreach my $number (1..2) {
5  &display($number);
6 }
7 sub display {
8  my $number = shift;
9  $number = sprintf("*** %02d ***\n",
                     $number);
10  print $number;
11 }
```

Dieses simple Programm erzeugt folgende Ausgabe:

```
perl example.pl
*** 01 ***
*** 02 ***
```

Beispielprogramm im Debugger aufrufen

```
perl -d example.pl
...
main::(example.pl:4): \
            foreach my $number (1..2) {
```

Der Debugger stoppt das Programm in der ersten ausführbaren Zeile (hier 4).

Blättern im Code

Mit dem Kommando `l` kann man durch den Code vorwärts blättern, mit `-` rückwärts. Die Angabe von Zeilennummern bzw. Zeilenbereichen, Subroutinen oder Variablen ist ebenfalls möglich.

Beispiel Zeile 1 bis 15 anzeigen

```
 DB<1> l 1-15
1     #!/usr/bin/perl
2:    use warnings;
3:    use strict;
4==>    foreach my $number (1..2) {
5:     &display($number);
6     }
7     sub display {
8:     my $number = shift;
9:     $number = sprintf(„*** %02d ***\n", \
                         $number);
10:    print $number;
11    }
```

Doppelpunkte (:) nach der Zeilennummer zeigen an, wo Breakpoints oder Aktionen (s.u.) gesetzt werden können. Der nach rechts zeigende Pfeil (==>) gibt die aktuelle Position im Programm an.

Subroutinen auflisten S

Das Kommando `S [[!] ~pattern]` listet alle Unterprogramme auf, die dem regulären Ausdruck `pattern` (nicht !) entsprechen.

Ohne Muster werden alle Subroutinen, auch die Unterprogramme, die von Paketen oder dem Debugger selbst geladen werden, angezeigt. Diese Liste kann sehr, sehr lang werden.

Daher empfiehlt sich die Angabe eines Pakets

```
 DB<1> S main
main::BEGIN
main::display
```

oder die Verwendung eines Patterns.

```
 DB<2> S display
 Term::ReadLine::Gnu::XS::display_readline_
version
 Term::ReadLine::Gnu::XS::shadow_redisplay
 main::display
```

Hier sieht man sehr schön, das wirklich alle geladenen Subroutinen durchsucht werden.

Variablen anzeigen

Das Kommando `V` zeigt alle Variablen im aktuellen Paket an. Wie bei den Subroutinen werden alle geladenen Variablen angezeigt. Diese Liste kann ebenfalls sehr, sehr lang sein.

Daher empfiehlt sich auch hier die Verwendung des Paketnamens in Verbindung mit einem Suchbegriff (`eq`)

```
 DB<3> V Carp Verbose
$Verbose = 0
```

oder die Verwendung des Paketnamens in Verbindung mit einem regulärem Ausdruck

```
 DB<4> V Carp ~V
$Verbose = 0
$VERSION = 1.04
```

Mit `my` deklarierte Variablen werden jedoch mit `V` nicht angezeigt. Hierzu kann das Kommando `y` verwendet werden, welches das CPAN-Modul PadWalker benötigt.

Das Programm ausführen (continue)

Das Kommando `c [line|sub]` (continue) führt den Code bis zur angegebenen Zeile oder bis zum angegebenen Unterprogramm aus.

Ohne Parameter wird der Code von der aktuellen Zeile bis zum nächsten Breakpoint oder Watch (s.u.) oder bis zum Ende des Programms ausgeführt.

```
main::(example.pl:4): \
      foreach my $number (1..2) {
  DB<1> c
*** 01 ***
*** 02 ***
```

Da hier keine Breaks oder Watches (s.u.) gesetzt wurden, läuft das Programm einfach durch.

Restart R

Sobald das Programm einmal vollständig durchgelaufen ist, ist ein Neustart des Programms im Debugger notwendig. Intern verwendet der Perl Debugger (perl5db.pl) dazu die Funktion `exec()`.

```
 Debugged program terminated.  Use q to quit
or R to restart,
   use o inhibit_exit to avoid stopping after
program termination,
   h q, h R or h o to get additional info.
   DB<1>R
 Warning: some settings and command-line op-
tions may be lost!
 ...
 main::(example.pl:4):    foreach my $number
(1..2) {
```

Mit ActiveState-Perl 5.8.x unter Windows funktioniert das leider nicht. Hier ist ein Neustart des Debuggers auf der Kommandozeile zwingend erforderlich. Dank der History-Funktion von `command.com` ist dies kein wirkliches Hindernis für den Einsatz des Debuggers.

Step Into

`s` führt die nächste Programmzeile aus und springt dabei in die Unterprogramme (Step Into)

Beispiel Step Into

```
     DB<1> s
   main::(example.pl:5):
 &display($number);
     DB<1> s
   main::display(example.pl:8): \
      my $number = shift;
     DB<1> s
```

```
  main::display(example.pl:9): \
$number = sprintf("*** %02d ***\n",$number);
      DB<1> s
    main::display(example.pl:10):     print
$number;
      DB<1> s
    *** 01 ***
    main::(example.pl:5):
&display($number);
```

Step Over

`n` führt die nächste Programmzeile aus und springt dabei über die Unterprogramme (Step Over)

Beispiel Step Over

```
  DB<1> n
main::(example.pl:5):     &display($number);
  DB<1> n
*** 01 ***
main::(example.pl:5):     &display($number);
  DB<1> n
*** 02 ***
```

Aus Subroutinen aussteigen mit return

Das Kommando `r` (return) kehrt aus Unterprogrammen zurück und zeigt den Rückgabewert an. Das ist beispielsweise in Verbindung mit `s` (Step Into) recht nützlich.

```
  DB<1> s
main::(example.pl:5):     &display($number);
  DB<1> s
main::display(example.pl:8): \
      my $number = shift;
```

Der Debugger steht jetzt in der Subroutine display().

```
  DB<1> r
*** 01 ***
void context return from main::display
main::(example.pl:5):     &display($number);
```

Als Bonus wird der Rückgabewert der Subroutine, hier void, angezeigt.

Actions, Breakpoints und Watches

Actions, Breakpoints und Watches auflisten

Das Kommando `L [a|b|w]` listet die gesetzten Actions (a), Breakpoints (b) und Watches (w). Ohne Parameter werden alle Aktionen, Breakpoints und Watches angezeigt.

Breakpoints

Das Kommando `b [line|sub [condition]]` setzt einen Haltepunkt in der angegebenen Zeile bzw. vor Ausführung des Unterprogramms. Zusätzlich kann eine Bedingung (beliebiger Perl-Code) festgelegt werden.

`B (line|*)` löscht den Haltepunkt in Zeile line oder alle Haltepunkte.

Breakpoint beim Einstieg in die Subroutine display() setzen

```
  DB<1> b display
```

und prüfen, ob und wo er gesetzt wurde

```
  DB<2> L
example.pl:
 8:     my $number = shift;
   break if (1)
```

Der Haltepunkt befindet sich in Zeile 8. `c` führt das Programm bis zum nächsten Breakpoint aus.

```
  DB<2> c
main::display(example.pl:8): my $number =
shift;
```

Das Programm bis zum nächsten Haltepunkt weiter ausführen: Hier wird das Unterprogramm einmal durchlaufen.

```
  DB<2> c
*** 01 ***
main::display(example.pl:8): \
      my $number = shift;
```

Das Programm bis zum nächsten Haltepunkt weiter ausführen: Hier wird das Unterprogramm noch einmal durchlaufen.

```
  DB<2> c
*** 02 ***
```

Sobald ein Haltepunkt erreicht wird, stoppt der Debugger das Programm. Jetzt können Variablen mit den bereits vorgestellten Methoden inspiziert und geändert werden.

Beispiel:

Breakpoint in Zeile 5 setzen

```
  DB<1> b 5
```

und das Programm bis zum Haltepunkt ausführen.

```
  DB<2> c
main::(example.pl:5):     &display($number);
```

Die skalare Variable $number inspizieren,

```
    DB<2> x \$number
  0  SCALAR(0x825a11c)
     -> 1
```

ändern

```
  DB<3> $number = 42;
```

und das Programm bis zum nächsten Breakpoint laufen lassen.

```
  DB<4> c
 *** 42 ***
main::(example.pl:5):     &display($number);
```

Die Änderung der Variable erfolgt nur im Debugger, der Originalcode muss nicht (!) geändert werden.

Actions

Das Kommando `a [line] command [condition]` setzt in Zeile Nummer line eine Aktion (beliebiger Perl-Code). Zusätzlich kann eine Bedingung (beliebiger Perl-Code) festgelegt werden. Eine Aktion wird vor der Ausführung der Zeile ausgeführt. `A [line|*]` löscht die Action in Zeile line bzw. alle (*).

Aktion in Zeile 5 setzen.

```
  DB<1> a 5 print "A5a $number "; $number++;
print "A5b $number\n"
```

Und das Programm einmal durchlaufen lassen.

```
  DB<2> c
A5a 1 A5b 2
*** 02 ***
A5a 2 A5b 3
*** 03 ***
```

Übergabeparameter an Subroutinen lassen sich mit Actions aus `@_` ermitteln.

```
  DB<1> a 8 print "Parameter $_[0]\n"

   DB<2> c
Parameter 1
*** 01 ***
Parameter 2
*** 02 ***
```

List Code revisited

Sobald Haltepunkte oder Actions gesetzt sind, werden diese im Programmlisting `l` durch ein b bzw. a nach der Zeilennummer gekennzeichnet.

```
  DB<1> b display
  DB<2> a 5 print "A5 $number";
  DB<3> a 8 print "A8 $number";
  DB<4> l 4-8
4==>    foreach my $number (1..2) {
5:a     &display($number);
6     }
7     sub display {
8:ba    my $number = shift;
```

Ein Haltepunkt hat Vorrang vor einer Aktion.

Variablen beobachten - Watches

Das Kommando `w [expr]` setzt einen Beobachter für expr (beliebiger Perl-Code), während `W (expr|*a)` einen oder alle Watches löscht.

```
  DB<1> w $number
  DB<2> c
Watchpoint 0:    $number changed:
    old value:    ' '
    new value:    '1'
main::(example.pl:5):     &display($number);
```

Graphische Benutzeroberflächen

Als Alternative zur Kommandozeile gibt es auch einige graphische Benutzeroberflächen. Nachfolgend eine kleine Auswahl.

ptkdb - Der Klassiker

http://ptkdb.sourceforge.net/

ddd (DataDisplayDebugger)

http://www.gnu.org/software/ddd/

Eclipse Perl Integration

http://e-p-i-c.sourceforge.net/

Komodo IDE

Kommerzielles Produkt von Active-State
http://activestate.com/products/komodo_ide/

Affrus - für Macs
Kommerzielles Produkt von Late Night Software Ltd
http://www.latenightsw.com/affrus/index.html

Referenzkarten zum Ausdrucken

Andrew Ford: Perl Debugger Reference Card
http://refcards.com/refcard/perl-debugger-forda

The Perl Debugger Quick Reference Card, a PDF courtesy of O'Reilly
http://www.rfi.net/debugger-slides/reference-card.pdf

Perl Debugger Quick Reference
http://www.perl.com/2004/11/24/debugger_ref.pdf

Einführung in Perl Debugger
http://www.ims.uni-stuttgart.de/lehre/teaching/2006-WS/Perl/debugger4up.pdf

Weiterführende Literatur

In diesem Artikel kann ich nicht auf alle Features des Perl-Debuggers eingehen. Wer mehr wissen möchte, sollte sich die nachfolgenden Bücher einmal genauer ansehen.

Perl Debugger Pocket Reference.
Foley, Richard:
Perl Debugger Pocket Reference.
(O'Reilly) ISBN: 0596005032
Klein, fein, handlich, gut.

Pro Perl Debugging: From Professional to Expert
Foley, Richard, Lester, Andy:
Pro Perl Debugging: From Professional to Expert
(Apress) ISBN: 1590594541
FMTEYEWTK: Ein wirklich ausführliches und gutes Buch mit zahlreichen kommentierten Beispielen.

Perl Debugged
Peter Scott, Ed Wright:
Perl Debugged
ISBN: 02011700549
(Out of print, aber noch im Handel erhältlich)
Ein sehr gutes Buch über Perl-Programmierung und Debugging. Meines Erachtens eines der besten Perl-Bücher überhaupt.

Thomas Fahle