perldebguts

Table des matières

1 NAME/NOM **1**

2 DESCRIPTION **1**

3 Éléments Internes du Débogueur **2**

3.1 Écrire Votre Propre Débogueur . . . 2

4 Exemples de Listages des Frames **3**

5 Débogage des expressions rationnelles **6**

5.1 Trace lors de la compilation . . . 6

5.2 Types de noeuds . . . 7

5.3 Sortie lors de l'exécution . . . 9

6 Débogage de l'utilisation de la mémoire par Perl **10**

6.1 Utilisation de `$ENV{PERL_DEBUG_MSTATS}` . . . 10

6.2 Exemple d'utilisation de l'option **-DL** . . . 11

6.3 Détails sur **-DL** . . . 13

6.4 Limitations des statistiques **-DL** . . . 13

7 VOIR AUSSI **13**

8 TRADUCTION **13**

8.1 Version . . . 13

8.2 Traducteur . . . 13

8.3 Relecture . . . 13

9 À propos de ce document **14**

1 NAME/NOM

perldebguts - Les entrailles du débogage de Perl

2 DESCRIPTION

Ceci n'est pas la page de manuel perldebug(1), qui vous indique comment utiliser le débogueur. Cette page donne des détails de bas niveau dont la compréhension va de difficile à impossible pour quiconque n'étant pas incroyablement intime avec les entrailles de Perl. Caveat lector (Ben me v'là frais, NDT).

3 Éléments Internes du Débogueur

Perl a des hooks spéciaux de débogage à la compilation et à l'exécution qui sont utilisés pour créer des environnements de débogage. Ces hooks ne doivent pas être confondes avec la commande *perl -Dxxx* décrite dans *perlrun*, qui n'est utilisable que si l'on utilise une version spéciale de Perl compilée selon les instructions du fichier pod *INSTALL* dans l'arborescence source de Perl (la phrase de la VO est incomplète... NDT).

Par exemple, lorsque vous appelez la fonction intégrée `caller` de Perl depuis le paquetage DB, les arguments avec lesquels a été appelée la frame correspondante de la pile sont copiés dans le tableau @DB::args. Le mécanisme général est validé en appelant Perl avec l'option **-d**, et les caractéristiques supplémentaires suivantes sont disponibles (cf. $^P in *perlvar*) :

- Perl insère le contenu de `$ENV{PERL5DB}` (ou de `BEGIN {require 'perl5db.pl'}` en son absence) avant la premiè ligne de l'application.
- Le tableau `@{"_<$filename"}` est le contenu ligne à ligne de $filename pour tous les fichiers compilés. Même chose pour les chaînes `eval`uées contenant des sous-programmes, ou qui sont actuellement exécutées. Le `$filename` pour les chaînes `eval`uées ressemble à `(eval 34)`. Les assertions de code dans les expressions rationnelles ressemblent à `(re_eval 19)`.
- Le hachage `%{"_<$filename"}` contient les points d'arrêt et les actions (ses clés sont les numéros de lignes), et des entrées individuelles sont modifiables (par opposition au hachage tout entier). Seul true/false est important pour Perl, même si les valeurs utilisées par *perl5db.pl* ont la forme `"$break_condition\0$action"`. Les valeurs sont magiques dans un contexte numérique : ce sont des zéros si la ligne ne peut pas être le lieu d'un point d'arrêt.
  Idem pour les chaînes évaluées qui contiennent des sous-programmes, ou qui sont en cours d'exécution. Le $filename pour les chaînes `eval`uées ressemble à `(eval 34)` ou à `(re_eval 19)`.
- Le scalaire `${"_<$filename"}` contient `"_<$filename"`. Idem pour les chaînes évaluées qui contiennent des sous-programmes, ou qui sont en cours d'exécution. Le $filename pour les chaînes `eval`uées ressemble à `(eval 34)` ou à `(re_eval 19)`.
- Après la compilation de chaque fichier exigé par `require`, mais avant son exécution, `DB::postponed(*{"_<$filename"})` est appelé (si le sous-programme `DB::postponed` existe). Ici le $filename est le nom développé du fichier exigé, tel que trouvé dans les valeurs de %INC.
- Après la compilation de chaque sous-programme `subname`, l'existence de `$DB::postponed{subname}` est vérifiée. Si cette clé existe, `DB::postponed(subname)` est appelé si le sous-programme `DB::postponed` existe.
- Un hachage `%DB::sub` est maintenu, dont les clés sont les noms des sous-programmes et dont les valeurs ont la forme `filename:startline-endline`. `filename` a la forme `(eval 34)` pour les sous-programmes définis dans des `eval`s, ou `(re_eval 19)` pour ceux qui se trouvent dans des assertions de code d'expression rationnelle.
- Lorsque l'exécution de votre programme atteint un endroit pouvant avoir un point d'arrêt, le sous-programme `DB::DB()` est appelé si l'une des variables $DB::trace, $DB::single ou $DB::signal est vraie. Ces variables ne sont pas `local`isables. Cette caractéristique est invalidée lorsque le contrôle est à l'intérieur de `DB::DB()` ou de fonctions appelées à partir de lui (À moins que `$^D & (1<30)`) soit vrai.
- Lorsque l'exécution de l'application atteint un appel de sous-programme, un appel à `&DB::sub`(*args*) est réalisé à la place, avec `$DB::sub` contenant le nom du sous-programme appelé. Ceci ne se produit pas si le sous-programme ait été compilé dans le paquetage `DB`.

Notez que si `&DB::sub` a besoin de données externes pour son bon fonctionnement, aucun appel de sous-programme n'est possible tant que ce n'est pas fait. Pour le débogueur standard, la variable `$DB::deep` (profondeur des niveaux de récursion dans le débogueur que vous pouvez atteindre avant un arrêt obligatoire) donne un exemple de telle dépendance.

3.1 Écrire Votre Propre Débogueur

Le débogueur fonctionnel minimal consiste en une seule ligne

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

ce qui est bien pratique comme contenu de la variable d'environnement `PERL5DB` :

```
$ "PERL5DB=sub DB::DB {}" perl -d your-script
```

Un autre débogueur minimal, un petit peu plus utile, pourrait être créé, la ligne unique étant

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

Ce débogueur afficherait le nombre séquentiel d'instructions recontrées, et attendrait que vous appuyiez sur entrée pour continuer.

Le débogueur suivant est plutôt fonctionnel :

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

Il affiche le nombre séquentiel d'appels de sous-programmes et le nom des sous-programmes appelés. Notez que `&DB::sub` doit être compilé dans le paquetage `DB`.

Au démarrage, le débogueur lit votre fichier rc (*./.perldb* ou *~/.perldb* sous Unix), qui peut définir des options importantes. Ce fichier peut définir un sous-programme `&afterinit` devant être exécuté après l'initialisation du débogueur.

Après la lecture du fichier rc, le débogueur lit la variable d'environnement PERLDB_OPTS et l'analyse comme si c'était le reste de la ligne `O ...` dans le prompt du débogueur.

Il maintient aussi des varibles internes magiques, telles que `@DB::dbline` et `%DB::dbline` qui sont des alias de `@{"::_<fichier_courant"}` et `%{"::_<fichier_courant"}`. Ici, `fichier_courant` est le fichier actuellement sélectionné, soit choisi explicitement par la commande `f` du débogueur, ou implicitement par le flux de l'exécution).

Certaines fonctions sont fournies pour simplifier la personnalisation. Voir Options Configurables in *perldebug* pour une description des options analysées par `DB::parse_options(string)`. La fonction `DB::dump_trace(skip[, count])` saute le nombre spécifié de frames et retourne une liste contenant des informations sur les frames de l'appelant (la totalité d'entre elles si `count` est manquant). Chaque entrée est une référence à un hachage contenant le `contexte` des clés (soit `.`, `$` ou `@`), `sub` (nom du sous-programme, ou infos sur `eval`), `args` (`undef` ou une référence à un tableau), `fichier`, et `ligne`.

La fonction `DB::print_trace(FH, skip[, count[, short]])` affiche des infos formatées sur les frames de l'appelant. Les deux dernières focntions peuvent être pratiques comme arguments des commandes < et >.

Notez que toute variable et toute fonction qui n'est pas documentée ici (ou dans *perldebug*) est considérée comme réservée à un usage interne uniquement, et est en tant que telle sujette à changement sans préavis.

4 Exemples de Listages des Frames

L'option `frame` peut être utilisée pour contrôler la sortie des informations sur les frames. Par exemple, comparez cette trace d'une expression :

```
$ perl -de 42
Stack dump during die enabled outside of evals.

Loading DB routines from perl5db.pl patch level 0.94
Emacs support available.

Enter h or `h h' for help.

main::(-e:1):   0
  DB<1> sub foo { 14 }

  DB<2> sub bar { 3 }

  DB<3> t print foo() * bar()
main::((eval 172):3):   print foo() + bar();
main::foo((eval 168):2):
main::bar((eval 170):2):
42
```

avec celle-ci, une fois que l'Option `frame=2` a été validée :

```
  DB<4> O f=2
               frame = '2'
  DB<5> t print foo() * bar()
3:      foo() * bar()
entering main::foo
```

```
 2:     sub foo { 14 };
exited main::foo
entering main::bar
 2:     sub bar { 3 };
exited main::bar
42
```

Pour les besoins de la démonstration, nous présentons ci-dessous un listage laborieux obtenu en plaçant votre variable d'environnement `PERLDB_OPTS` à la valeur `f=n N`, et en exécutant *perl -d -V* sur la ligne de commande. Les exemples utilisent diverses valeurs de `n` pour vous montrer la différence entre ces réglages. Aussi longs qu'ils puissent paraître, ils ne sont pas des listages complets, mais seulement des extraits.

valeur 1

```
entering main::BEGIN
 entering Config::BEGIN
  Package lib/Exporter.pm.
  Package lib/Carp.pm.
 Package lib/Config.pm.
 entering Config::TIEHASH
 entering Exporter::import
  entering Exporter::export
entering Config::myconfig
 entering Config::FETCH
 entering Config::FETCH
 entering Config::FETCH
 entering Config::FETCH
```

valeur 2

```
entering main::BEGIN
 entering Config::BEGIN
  Package lib/Exporter.pm.
  Package lib/Carp.pm.
 exited Config::BEGIN
 Package lib/Config.pm.
 entering Config::TIEHASH
 exited Config::TIEHASH
 entering Exporter::import
  entering Exporter::export
  exited Exporter::export
 exited Exporter::import
exited main::BEGIN
entering Config::myconfig
 entering Config::FETCH
 exited Config::FETCH
 entering Config::FETCH
 exited Config::FETCH
 entering Config::FETCH
```

valeur 4

```
in  $=main::BEGIN() from /dev/null:0
 in  $=Config::BEGIN() from lib/Config.pm:2
  Package lib/Exporter.pm.
  Package lib/Carp.pm.
 Package lib/Config.pm.
 in  $=Config::TIEHASH('Config') from lib/Config.pm:644
 in  $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
  in  $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from li
in  @=Config::myconfig() from /dev/null:0
 in  $=Config::FETCH(ref(Config), 'package') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'baserev') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'PERL_VERSION') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'PERL_SUBVERSION') from lib/Config.pm:574
```

```
 in  $=Config::FETCH(ref(Config), 'osname') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'osvers') from lib/Config.pm:574
```

valeur 6

```
in  $=main::BEGIN() from /dev/null:0
 in  $=Config::BEGIN() from lib/Config.pm:2
  Package lib/Exporter.pm.
  Package lib/Carp.pm.
 out $=Config::BEGIN() from lib/Config.pm:0
 Package lib/Config.pm.
 in  $=Config::TIEHASH('Config') from lib/Config.pm:644
 out $=Config::TIEHASH('Config') from lib/Config.pm:644
 in  $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
  in  $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/
  out $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/
 out $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
out $=main::BEGIN() from /dev/null:0
in  @=Config::myconfig() from /dev/null:0
 in  $=Config::FETCH(ref(Config), 'package') from lib/Config.pm:574
 out $=Config::FETCH(ref(Config), 'package') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'baserev') from lib/Config.pm:574
 out $=Config::FETCH(ref(Config), 'baserev') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'PERL_VERSION') from lib/Config.pm:574
 out $=Config::FETCH(ref(Config), 'PERL_VERSION') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'PERL_SUBVERSION') from lib/Config.pm:574
```

valeur 14

```
in  $=main::BEGIN() from /dev/null:0
 in  $=Config::BEGIN() from lib/Config.pm:2
  Package lib/Exporter.pm.
  Package lib/Carp.pm.
 out $=Config::BEGIN() from lib/Config.pm:0
 Package lib/Config.pm.
 in  $=Config::TIEHASH('Config') from lib/Config.pm:644
 out $=Config::TIEHASH('Config') from lib/Config.pm:644
 in  $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
  in  $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/E
  out $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/E
 out $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
out $=main::BEGIN() from /dev/null:0
in  @=Config::myconfig() from /dev/null:0
 in  $=Config::FETCH('Config=HASH(0x1aa444)', 'package') from lib/Config.pm:574
 out $=Config::FETCH('Config=HASH(0x1aa444)', 'package') from lib/Config.pm:574
 in  $=Config::FETCH('Config=HASH(0x1aa444)', 'baserev') from lib/Config.pm:574
 out $=Config::FETCH('Config=HASH(0x1aa444)', 'baserev') from lib/Config.pm:574
```

valeur 30

```
in  $=CODE(0x15eca4)() from /dev/null:0
 in  $=CODE(0x182528)() from lib/Config.pm:2
  Package lib/Exporter.pm.
 out $=CODE(0x182528)() from lib/Config.pm:0
 scalar context return from CODE(0x182528): undef
 Package lib/Config.pm.
 in  $=Config::TIEHASH('Config') from lib/Config.pm:628
 out $=Config::TIEHASH('Config') from lib/Config.pm:628
 scalar context return from Config::TIEHASH:   empty hash
 in  $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
  in  $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/Exporter.pm:171
  out $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/Exporter.pm:171
  scalar context return from Exporter::export: ''
 out $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
 scalar context return from Exporter::import: ''
```

Dans tous les cas montrés ci-dessus, l'indentation des lignes montre l'arbre d'appels. Si le bit 2 de `frame` est mis, alors une ligne est affichée aussi à la sortie d'un sous-programme. Si le bit 4 est mis, alors les arguments sont aussi affichés ainsi que les infos sur l'appelant. Si le bit 8 est mis, les arguments sont affichés même s'ils sont liés ou sont des références. Si le bit 16 est mis, la valeur de retour est aussi affichée.

Lorsqu'un paquetage est compilé, une ligne comme celle-ci

```
    Package lib/Carp.pm.
```

est affichée avec l'indentation adéquate.

5 Débogage des expressions rationnelles

Il y a deux façons d'obtenir une trace de débogage pour les expressions rationnelles.

Si votre perl est compilé avec `-DDEBUGGING` activé, vous pouvez utiliser l'option de ligne de commande **-Dr**.

Sinon, vous pouvez indiquer `use re 'debug'`, qui est effectif à la fois à la compilation et lors de l'exécution. Il n'a *pas* de portée lexicale.

5.1 Trace lors de la compilation

La trace de débogage lors de la compilation a cette allure :

```
  compiling RE '[bc]d(ef*g)+h[ij]k$'
  size 43 first at 1
     1: ANYOF(11)
    11: EXACT <d>(13)
    13: CURLYX {1,32767}(27)
    15:   OPEN1(17)
    17:     EXACT <e>(19)
    19:     STAR(22)
    20:       EXACT <f>(0)
    22:     EXACT <g>(24)
    24:   CLOSE1(26)
    26:   WHILEM(0)
    27: NOTHING(28)
    28: EXACT <h>(30)
    30: ANYOF(40)
    40: EXACT <k>(42)
    42: EOL(43)
    43: END(0)
  anchored 'de' at 1 floating 'gh' at 3..2147483647 (checking floating)
                                    stclass 'ANYOF' minlen 7
```

La première ligne montre la forme de l'expression avant sa compilation. La seconde sa taille une fois compilée (avec une unité arbitraire, habituellement des mots de 4 octets) et l'*id* du label du premier noeud qui lui correspond.

La dernière ligne (coupée sur deux lignes ci-dessus) contient les infos de l'optimiseur. Dans l'exemple donné, l'optimiseur a trouvé que la correspondance devait contenir une sous-chaîne `de` à l'offset 1, et une sous-chaîne `gh` à un offset quelconque entre 3 et l'infini. Qui plus est, en vérifiant ces sous-chaînes (pour abandonner rapidement les correspondances impossibles) il recherchera la sous-chaîne `gh` avant la sous-chaîne `de`. L'optimiseur peut aussi utiliser le fait qu'il sait que la correspondance doit commencer (au `premier` *id*) par un caractère, et qu'elle ne doit pas faire moins de 7 caractères.

Les champs intéressants qui peuvent apparaître dans la dernière ligne sont

`anchored` ***STRING*** **`at`** ***POS***

`floating` ***STRING*** **`at`** ***POS1..POS2***

Voir ci-dessus.

`matching floating/anchored`

Quelle sous-chaîne rechercher en premier.

minlen
: La longueur minimale de la correspondance.

stclass *TYPE*
: Type du premier noeud correspondant.

noscan
: Ne pas rechercher les sous-chaînes trouvées.

isall
: Indique que les infos de l'optimiseur représentent en fait tout ce que contient l'expression rationnelle, on n'a donc pas du tout besoin d'entrer dans le moteur d'expressions rationnelles.

GPOS
: Mis si le motif contient `\G`.

plus
: Mis si le motif débute par un caractère répété (comme dans `x+y`).

implicit
: Mis si le motif commence par `.*`.

with eval
: Mis si le motif contient des groupes eval, tels que `(?{ code })` et `(??{ code })`.

anchored(TYPE)
: Mis si le motif ne peut correspondre qu'à quelques endroits (avec `TYPE` valant `BOL`, `MBOL` ou `GPOS`, voir la table ci-dessous).

Si une sous-chaîne est connue comme ne pouvant correspondre qu'à une fin de ligne, elle peut être suivie de `$`, comme dans `floating 'k'$`.

Les infos spécifiques de l'optimiseur sont utilisées pour éviter d'entrer dans un moteur d'expressions rationnelles (lent) pour des chaînes qui ne correspondront certainement pas. Si le drapeau `isall` est mis, un appel du moteur d'expressions rationnelles peut être évité même lorsque l'optimiseur a trouvé un endroit approprié pour la correspondance.

Le reste de la sortie contient la liste des *noeuds* de la forme compilée de l'expression. Chaque ligne a pour format

id: *TYPE OPTIONAL-INFO* (*next-id*)

5.2 Types de noeuds

Voici la liste des types possibles accompagnés de courtes descriptions :

```
# TYPE arg-description [num-args] [longjump-len] DESCRIPTION

# Points de sortie
END         no      Fin du programme.
SUCCEED     no      Retour d'un sous-programme, simplement.

# Ancres
BOL         no      Correspond à "" en début de ligne.
MBOL        no      Idem, sur plusieurs lignes.
SBOL        no      Idem, sur une seule ligne.
EOS         no      Correspond à "" en fin de chaîne.
EOL         no      Correspond à "" en fin de ligne.
MEOL        no      Idem, sur plusieurs lignes.
SEOL        no      Idem, sur une seule ligne.
BOUND       no      Correspond à "" à une frontière entre mots.
BOUNDL      no      Correspond à "" à une frontière entre mots.
NBOUND      no      Correspond à "" en-dehors d'une frontière.
NBOUNDL     no      Correspond à "" en-dehors d'une frontière.
GPOS        no      Correpondance là où le dernier m//g s'est arrêté.
```

```
# Alternatives [spéciales]
ANY         no      Correspond à n'importe quel caractère (sauf
                    nouvelle ligne)
SANY        no      Correspond à un caractère.
ANYOF       sv      Correspond à un caractère dans (ou hors de)
                    cette classe.
ALNUM       no      Correspond à un caractère alphanumérique.
ALNUML      no      Correspond à un caractère alphanumérique local.
NALNUM      no      Correspond à un caractère non alphanumérique
NALNUML     no      Correspond à un caractère non alphanumérique local.
SPACE       no      Correspond à un blanc.
SPACEL      no      Correspond à un blanc local.
NSPACE      no      Correspond à un caractère non blanc.
NSPACEL     no      Correspond à un caractère non blanc local.
DIGIT       no      Correspond à un caractère numérique.
NDIGIT      no      Correspond à un caractère non numérique.

# BRANCH    L'ensemble de branches consituant un simple choix,
#           accompagnées de leurs pointeurs "suivant", puisque la
#           précédence empêche quoi que ce soit d'être concaténé à
#           une branche particulière. Le pointeur "suivant" de la
#           dernière BRANCH dans un choix pointe vers ce qui suit
#           le choix complet. C'est aussi là que pointe le
#           pointeur "suivant" final de chaque branche ; chaque
#           branche débute par le noeud opérande d'un noeud BRANCH.
#
BRANCH      node    Correspond à cette alternative, ou la suivante...

# BACK      Les pointeurs "suivant" normaux pointent tous
#           implicitement vers l'avant ; BACK existe pour rendre
#           les structures de boucles possibles.
# non utilisé
BACK        no      Correspond à "", le pointeur "suivant" pointe
                    vers l'arrière.

# Littéraux
EXACT       sv      Correspond à cette chaîne (précédée de sa
                    longueur).
EXACTF      sv      Correspond à cette chaîne, repliée (? NDT) (avec sa
                    longueur).
EXACTFL     sv      Correspond à cette chaîne locale, repliée
                    (avec sa longueur).

# Ne fait rien
NOTHING     no      Correspond à la chaîne vide.
# Une variante du précédent, qui délimite un groupe, arrêtant
# ainsi les optimisations
TAIL        no      Correspond à la chaîne vide. On peut sauter
                    d'ici vers l'extérieur.

# STAR,PLUS '?', et les complexes '*' et '+', sont implémentés
#           sous la forme de structure BRANCH circulaires
#           utilisant BACK. Les cas simples (un caractère par
#           correspondance) sont implémentés avec STAR et PLUS
#           pour leur rapidité et pour minimiser les plongées
#           récursives.
#
STAR        node    Correspond à ce (simple) truc 0 ou plusieurs fois.
PLUS        node    Correspond à ce (simple) truc 1 ou plusieurs fois.
```

```
CURLY       sv 2    Correspond à ce simple truc {n,m} fois.
CURLYN      no 2    Match next-after-this simple thing
#                   {n,m} times, set parenths.
CURLYM      no 2    Correspond à ce truc de complexité moyenne {n,m} fois.
CURLYX      sv 2    Correspond à ce truc complexe {n,m} fois.

# Ce terminateur crée une structure de boucle pour CURLYX
WHILEM      no      Effectue un traitement des accolades et voit
                    si le reste correspond.

# OPEN,CLOSE,GROUPP ... sont dénombrés à la compilation.
OPEN        num 1   Marque ce point de l'entrée comme début #n.
CLOSE       num 1   Analogue à OPEN.

REF         num 1   Correspond à une chaîne déjà trouvée.
REFF        num 1   Correspond à une chaîne déjà trouvée, repliée.
REFFL       num 1   Correspond à une chaîne déjà trouvée, repliée,
                    locale.

# assertions de groupage
IFMATCH     off 1 2 Réussit si la suite correspond.
UNLESSM     off 1 2 Rate si la suite correspond.
SUSPEND     off 1 1 Sous expression rationnelle "indépendente".
IFTHEN      off 1 1 Switch, devrait être précédé par un switcher.
GROUPP      num 1   Si le groupe correspond.

# Support des expressions rationnelles longues
LONGJMP     off 1 1 Saute loin en avant.
BRANCHJ     off 1 1 BRANCH avec un offset long.

# Le travailleur de force
EVAL        evl 1   Exécute du code Perl.

# Modifieurs
MINMOD      no      L'opérateur suivant n'est pas avide.
LOGICAL     no      L'opcode suivant doit placer le drapeau
                    uniquement (? NDT).

# Ceci n'est pas encore utilisé
RENUM       off 1 1 Groupe ayant des parenthèses dénombrées
                    indépendamment.

# Ceci n'est pas vraiment un noeud, mais un morceau optimisé d'un
# noeud "long". Pour simplifie la sortie de débogage, nous
# l'indiquons comme si c'était un noeud
OPTIMIZED   off     Conteneur pour vidage.
```

5.3 Sortie lors de l'exécution

Tout d'abord, lors de la recherche d'une correspondance, on peut n'avoir aucune sortie même si le débogage est validé. Ceci signifie qu'on n'est jamais entré dans le moteur d'expressions rationnelles, et que tout le travail a été fait par l'optimiseur.

Si on est entré dans le moteur d'expressions rationnelles, la sortie peut avoir cette allure :

```
Matching '[bc]d(ef*g)+h[ij]k$' against 'abcdefg__gh__'
  Setting an EVAL scope, savestack=3
   2 <ab> <cdefg__gh_>    |  1: ANYOF
   3 <abc> <defg__gh_>    | 11: EXACT <d>
   4 <abcd> <efg__gh_>    | 13: CURLYX {1,32767}
   4 <abcd> <efg__gh_>    | 26:   WHILEM
```

```
                            0 out of 1..32767  cc=effff31c
 4 <abcd> <efg__gh_>     | 15:     OPEN1
 4 <abcd> <efg__gh_>     | 17:     EXACT <e>
 5 <abcde> <fg__gh_>     | 19:     STAR
                          EXACT <f> can match 1 times out of 32767...
Setting an EVAL scope, savestack=3
 6 <bcdef> <g__gh__>     | 22:       EXACT <g>
 7 <bcdefg> <__gh__>     | 24:       CLOSE1
 7 <bcdefg> <__gh__>     | 26:       WHILEM
                                  1 out of 1..32767  cc=effff31c
Setting an EVAL scope, savestack=12
 7 <bcdefg> <__gh__>     | 15:         OPEN1
 7 <bcdefg> <__gh__>     | 17:         EXACT <e>
   restoring \1 to 4(4)..7
                                  failed, try continuation...
 7 <bcdefg> <__gh__>     | 27:         NOTHING
 7 <bcdefg> <__gh__>     | 28:         EXACT <h>
                                  failed...
                               failed...
```

L'information la plus significative de la sortie est celle concernant le *noeud* particulier de l'expression rationnelle compilée qui est en cours de test vis-à-vis de la chaîne cible. Le format de ces lignes est le suivant

STRING-OFFSET <PRE-STRING> <POST-STRING> |ID: TYPE

Les infos de *TYPE* sont indentées en fonction du niveau de trace. D'autres informations incidentes apparaissent entremêlées au reste.

6 Débogage de l'utilisation de la mémoire par Perl

Perl est *très* frivole avec la mémoire. Il y a un dicton qui dit que pour estimer l'utilisation de la mémoire par Perl, il faut envisager un algorithme d'allocation raisonnable et multiplier votre estimation par 10, et même si vous êtes peut-être encore loin du compte, au moins vous ne serez pas trop surpris. Ce n'est pas absolument vrai, mais cela peut vous donner une bonne idée de ce qui se passe.

Disons qu'un entier ne peut pas occuper moins de 20 octets en mémoire, qu'un flottant ne peut pas prendre moins de 24 octets, qu'une chaîne ne peut pas prendre moins de 32 octets (tous ces exemples valant pour des architectures 32 bits, les résultats étant bien pires sur les architectures 64 bits). Si on accède à une variable de deux ou trois façons différentes (ce qui requiert un entier, un flottant ou une chaîne), l'empreinte en mémoire peut encore augmenter de 20 octets. Une implémentation peu soignée de malloc() augmentera encore plus ces nombres.

À l'opposé, une déclaration comme

```
sub foo;
```

peut prendre jusqu'à 500 octets de mémoire, selon la version de Perl que vous utilisez.

Des estimations à la louche et anecdotiques sur un code bouffi donnent un facteur d'accroissement d'environ 8. Cela signifie que la forme compilée d'un code raisonnable (commenté normalement, indenté proprement, etc.) prendra approximativement 8 fois plus de place que l'espace disque nécessaire au code.

Il existe deux façons spécifiques à Perl d'analyser l'usage de la mémoire : $ENV{PERL_DEBUG_MSTATS} et l'option de ligne de commande **-DL**. La première est disponible seulement si perl est compilé avec le malloc() de Perl, la seconde seulement si Perl a été compilé avec l'option `-DDEBUGGING`. Voir les instructions sur la façon dont on fait cela dans la page pod *INSTALL* à la racine de l'arborescence des sources de Perl.

6.1 Utilisation de `$ENV{PERL_DEBUG_MSTATS}`

Si votre perl utilise le malloc() de Perl, et s'il a été compilé avec les options correctes (c'est le cas par défaut), alors il affichera des statistiques sur l'usage de la mémoire après avoir compilé votre code lorsque `$ENV{PERL_DEBUG_MSTATS} > 1`, et avant la fin du programme lorsque `$ENV{PERL_DEBUG_MSTATS} >= 1`. Le format du rapport est similaire à celui de l'exemple suivant :

```
$ PERL_DEBUG_MSTATS=2 perl -e "require Carp"
Memory allocation statistics after compilation: (buckets 4(4)..8188(8192)
   14216 free:   130   117    28     7     9   0   2     2   1 0 0
               437    61    36     0     5
   60924 used:   125   137   161    55     7   8   6    16   2 0 1
                74   109   304    84    20
Total sbrk(): 77824/21:119. Odd ends: pad+heads+chain+tail: 0+636+0+2048.
Memory allocation statistics after execution:   (buckets 4(4)..8188(8192)
   30888 free:   245    78    85    13     6   2   1     3   2 0 1
               315   162    39    42    11
  175816 used:   265   176  1112   111    26  22  11    27   2 1 1
               196   178  1066   798    39
Total sbrk(): 215040/47:145. Odd ends: pad+heads+chain+tail: 0+2192+0+6144.
```

Il est possible de demander de telles statistiques à un moment arbitraire de votre exécution en utilisant la fonction mstats() du module standard Devel::Peek::mstats().

Voici l'explication des différentes parties du format :

buckets SMALLEST(APPROX)..GREATEST(APPROX)

Le malloc() de Perl utilise des allocations par buckets. Chaque requête est arrondie à la plus proche taille de bucket disponible, et un bucket de cette taille est pris dans le pool de buckets correspondant.

La ligne ci-dessus décrit les limites des buckets en cours d'utilisation. Chaque bucket a deux tailles : l'empreinte en mémoire, et la taille maximale des données utilisateur qui peuvent être placées dans ce bucket. Supposez dans l'exemple ci-dessus que la taille du bucket le plus petit est de 4. Le plus grand bucket aura une taille utilisable de 8188, et son empreinte en mémoire sera de 8192.

Free/Used

La rangée ou les deux rangées suivante(s) de nombres correspond(ent) au nombre de buckets de chaque taille entre SMALLEST et GREATEST. Dans la première rangée, les tailles (empreintes mémoire) des buckets sont des puissances de deux (ou peut-être d'une page plus grandes). Dans la seconde rangée (si elle est présente) les empreintes en mémoire des buckets sont entre les empreintes mémoire des deux buckets de la rangée au-dessus.

Par exemple, supposons dans l'exemple précédent que les empreintes mémoire soient de

```
free:    8     16    32    64    128  256 512 1024 2048 4096 8192
       4     12    24    48    80
```

Sans DÉBOGAGE de perl les buckets ayant une longueur supérieure à 128 ont un en-tête de 4 octets, un bucket de 8192 octets de long peut ainsi supporter des allocations de 8188 octets.

Total sbrk(): SBRKed/SBRKs:CONTINUOUS

Les deux premiers champs donnent la quantité totale de mémoire que perl a sbrk()é (ess-broken? :-), et le nombre de sbrk()s utilisés. Le troisième nombre est ce que perl pense de la continuité des morceaux retournés. Tant que ce nombre est positif, malloc() présumera qu'il est probable que sbrk() fournira une mémoire continue.

La mémoire allouée par les bibliothèques externes n'est pas comptée.

pad: 0

La quantité de mémoire sbrk()ée nécessaire pour garder les buckets alignés.

heads: 2192

Tandis que l'en-tête en mémoire des buckets les plus grands est gardée à l'intérieur du bucket, pour les buckets plus petits, il est stocké dans des zones séparées. Ce champ donne la taille totale de ces zones.

chain: 0

malloc() peut vouloir diviser un gros bucket en buckets plus petits. Si seulement une part du bucket décédé est laissée non subdivisée, le reste gardé comme élément d'une liste chaînée. Ce champ donne la taille totale de ces morceaux.

tail: 6144

Pour minimiser la quantité de sbrk()s malloc() demande plus de mémoire. Ce champ donne la taille de la partie non encore utilisée, qui est sbrk()ée, mais jamais touchée.

6.2 Exemple d'utilisation de l'option -DL

Ci-dessous nous montrons comment analyser l'usage de la mémoire par

```
do 'lib/auto/POSIX/autosplit.ix';
```

Le fichier en question contient un en-tête et 146 lignes similaires à

```
sub getcwd;
```

AVERTISSEMENT : la discussion ci-dessous suppose une architecture 32 bits. Dans les version de perl les plus récentes, l'usage de la mémoire des constructions discutées ici est nettement améliorée, mais ce qui suit est une histoire vraie. Cette histoire est impitoyablement laconique, et suppose un peu plus qu'une connaissance superficielle du fonctionnement interne de Perl. Appuyez sur espace pour continuer, 'q' pour quitter (en fait, vous voudrez juste passer à la section suivante).

Voici la liste détaillée des allocations réalisées par Perl pendant l'analyse de ce fichier :

```
!!! "after" at test.pl line 3.
   Id  subtot   4   8  12  16  20  24  28  32  36  40  48  56  64  72  80 80+
 0 02   13752   .   .   .   . 294   .   .   .   .   .   .   .   .   .   .   4
 0 54    5545   .   .   8 124  16   .   .   .   1   1   .   .   .   .   .   3
 5 05      32   .   .   .   .   .   .   .   1   .   .   .   .   .   .   .   .
 6 02    7152   .   .   .   .   .   .   .   .   .   . 149   .   .   .   .   .
 7 02    3600   .   .   .   .   . 150   .   .   .   .   .   .   .   .   .   .
 7 03      64   .  -1   .   1   .   .   2   .   .   .   .   .   .   .   .   .
 7 04    7056   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   7
 7 17   38404   .   .   .   .   .   .   .   1   .   . 442 149   .   . 147   .
 9 03    2078  17 249  32   .   .   .   .   2   .   .   .   .   .   .   .   .
```

Pour voir cette liste, insérez deux instructions `warn('!...')` autour de l'appel :

```
warn('!');
do 'lib/auto/POSIX/autosplit.ix';
warn('!!! "after"');
```

et exécutez-le avec l'option **-DL**. Le premier warn() affichera les infos sur l'allocation mémoire avant l'analyse du fichier, et mémorisera les statistiques à cet instant (nous ignorons ce qu'il affiche). Le second warn() affichera les variations par rapport à ces statistiques. Cela donne la sortie précédente.

Les différents *Id*entifiants sur la gauche correspondent aux différents sous-systèmes de l'interpréteur perl, ils sont juste les premiers arguments donnés à l'API d'allocation mémoire New() de perl. Pour déterminer ce que `9 03` signifie faites un `grep` dans le source de perl à la recherche de `903`. Vous verrez que c'est la fonction savepn() dans *util.c*. Cette fonction est utilisée pour stocker une copie d'un morceau existant de mémoire. En utilisant un débogueur C, on peut voir qu'elle est appelée soit directement depuis gv_init(), ou via sv_magic(), et gv_init() est appelée depuis gv_fetchpv() - qui est appelée depuis newSUB(). S'il-vous-plaît, veuillez faire une pause pour reprendre votre souffle maintenant.

NOTE : pour atteindre cet endroit dans le débogueur et sauter tous les appels à savepvn pendant la compilation du script principal, placez un point d'arrêt C dans Perl_warn(), `continuez` jusqu'à ce que ce point soit atteint, *puis* placez un point d'arrêt dans Perl_savepvn(). Notez que vous pouvez avoir besoin de sauter une poignée de Perl_savepvn() qui ne correspondent pas à une production de masse de CV (il y a plus d'allocations `903` que les 146 lignes identiques de *lib/auto/POSIX/autosplit.ix*). Notez aussi que les préfixes `Perl_` sont ajoutés par du code de macroisation dans les fichiers d'en-tête perl pour éviter des conflits avec les bibliothèques externes.

En tout cas, nous voyons que les ids `903` correspondent à la création de globs, deux fois par glob - pour le nom du glob, et pour la magie de transformation en chaîne du glob (Wouarf ! NDT).

Voici des explications pour les autres *Id*s ci-dessus :

`717`

Crée de plus grosses structures `XPV*`. Dans le cas ci-dessus, il crée 3 `AV` par sous-programme, un pour une liste de noms de variables lexicales, un pour un scratchpad (qui contient les variables lexicales et les `cibles`), et un pour le tableau des scratchpads nécessaire pour la récursivité.

Il crée aussi un `GV` et un `CV` par sous-programme, tous appelés depuis start_subparse().

`002`

Crée un tableau C correspondant à l'`AV` des bloc-notes (NDT ?), et le bloc-note lui-même. La première entrée fausse de ce bloc-note est créée même si le sous-programme lui-même n'est pas encore défini.

Il crée aussi des tableaux C pour conserver les données mises de côté (stash NDT ?) (c'est un HV (NDT ?), mais il grossit, il se produit donc 4 grosses allocations : les gros paquets ne sont pas libérés, et son conservés comme arènes (NDT ?) additionnelles pour les allocations de `SV`).

054

Crée un `HEK` pour le nom du glob (NDT ?) du sous-programme (ce nom est une clé dans un *stash*).
Les grosses allocations ayant cet *Id* correspondent à des allocations de nouvelles arènes pour stocker `HE`.

602

Crée un `GP` pour le glob du sous-programme.

702

Crée le `MAGIC` pour le glob du sous-programme.

704

Crée des *arènes* qui stockent les SV.

6.3 Détails sur -DL

Si Perl est exécuté avec l'option **-DL**, alors les warn()s qui débutent par '!' se comportent de façon particulière. Ils affichent une liste des *catégories* d'allocations de mémoire, et des statistiques sur leurs tailles.

Si la chaîne warn() commence par

! ! !

affiche uniquement les catégories ayant changé, affiche les variations en nombre d'allocations.

! !

affiche uniquement les catégories ayant grandi, leurs nombres en valeur absolue, et leurs totaux.

!

affiche les catégories non vides, leurs nombres en valeur absolue, et leurs totaux.

6.4 Limitations des statistiques -DL

Si une extension ou une bibliothèque externe n'utilise pas l'API Perl pour attribuer la mémoire, de telles allocations ne sont pas comptées.

7 VOIR AUSSI

perldebug, *perlguts*, *perlrun re*, et *Devel::Dprof*.

8 TRADUCTION

8.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.6.0. Pour en savoir plus concernant ces traductions, consultez `http://perl.enstimac.fr/`.

8.2 Traducteur

Roland Trique <*roland.trique@uhb.fr*>

8.3 Relecture

Gérard Delafond

9 À propos de ce document

Ce document est la traduction française du document orginal distribué avec perl. Vous pouvez retrouver l'ensemble de la documentation française Perl (éventuellement mise à jour) en consultant l'URL <http://perl.enstimac.fr/>.

Ce document PDF a été produit Paul Gaborit. Si vous utilisez la version PDF de cette documentation (ou une version papier issue de la version PDF) pour tout autre usage qu'un usage personnel, je vous serai reconnaissant de m'en informer par un petit message <mailto:Paul.Gaborit@enstimac.fr>.

Si vous avez des remarques concernant ce document, en premier lieu, contactez la traducteur (vous devriez trouver son adresse électronique dans la rubrique TRADUCTION) et expliquez-lui gentiment vos remarques ou critiques. Il devrait normalement vous répondre et prendre en compte votre avis. En l'absence de réponse, vous pouvez éventuellement me contacter.

Vous pouvez aussi participer à l'effort de traduction de la documentation Perl. Toutes les bonnes volontés sont les bienvenues. Vous devriez trouver tous les renseignements nécessaires en consultant l'URL ci-dessus.