perlmod

Table des matières

1 NAME/NOM

perlmod - Modules Perl (paquetages et tables de symboles)

2 DESCRIPTION

2.1 Paquetages

Perl fournit un mécanisme de namespace alternatif pour éviter aux paquetages de s'écraser mutuellement les variables. En fait, il n'y a rien qui ressemble aux variables globales en perl (bien que quelques identificateurs appartiennent par défaut au paquetage principal plutôt qu'au paquetage courant). L'instruction package déclare l'unité de compilation comme étant le namespace utilisé. La visibilité d'une déclaration de paquetage est de la déclaration jusqu'à la fin du bloc, `eval`, `sub`, ou la fin du fichier (c'est la même visibilité que les opérateurs my() et local()). Tous les identificateurs dynamiques suivants seront dans le même namespace. L'instruction package affecte uniquement les variables dynamiques – ainsi que celles sur lesquelles vous avez utilisé local() – mais *pas* sur les variables lexicales créées a l'aide de my(). Typiquement, cela serais la premiers déclaration dans un fichier a être inclue par un `require` ou un `use`. Vous pouvez inclure un paquetage dans plusieurs endroits; cela n'a quasiment aucune influence sur la table de symbole utilisée par le compilateur pour le reste du bloc. Vous pouvez utiliser les variables et les fichiers d'autres paquetages en préfixant l'identificateur avec le nom du paquetage et d'un double ":" : `$Package::Variable`. Si le nom de paquetage est nul, le `main` est utilisé. Donc, `$::sail` est équivalent à `$main::sail`.

L'ancien délimiteur de paquetage étais une apostrophe, mais un double ":" est maintenant utilisé, parce que c'est plus facilement compréhensible par les humains, et parce que c'est plus pratique pour les macros d'**emacs**. Cela fait aussi croire aux programmeurs C++ qu'ils comprennent ce qui se passe – en opposition a l'apostrophe qui faisait penser aux programmateurs Ada qu'ils comprenaient ce qui se passait. Comme l'ancienne méthode est toujours supportée pour préserver la compatibilité ascendante, si vous essayez une chaîne comme `"This is $owner's house"`, vous allez en fait accéder à `$owner::s`; c'est a dire, la variable $s du paquetage `owner`, ce qui n'est probablement pas ce que vous vouliez. Utilisez des accolades pour supprimer l'ambiguïté, comme ça : `"This is ${owner}'s house"`.

Les paquetages peuvent être imbriqués dans d'autres paquetages : `$EXTERNE::INTERNE::variable`. D'ailleurs, ceci n'implique rien dans l'ordre de recherche des noms. Tous les symboles sont soit locaux dans le paquetage courant, soit doivent avoir leur nom complet. Par exemple, dans le paquetage `EXTERNE`, `$INTERNET::var` ne se réfère pas à `$EXTERNE::INTERNE::var`. Il croira que le paquetage `INTERNE` est un paquetage totalement séparé.

Seuls les identificateurs commençant par une lettre (ou un underscore) sont stockés dans la table de symboles des paquetages. Tous les autres symboles sont gardés dans le paquetage `main`, ceci inclue les variables de ponctuations telles $_. De plus, les identificateurs STDIN, STDOUT, STDERR, ARGV, ARGVOUT, ENV, INC, et SIG sont stockés dans le paquetage `main` lorsque ils ne sont pas redéfinis, même si ils sont utilisés dans un autre but que celui pour lequel ils ont été créés. Notez aussi que si vous avez un paquetage appelé `m`, `s`, ou `y`, alors, vous ne pourrez pas utiliser la forme qualifiée d'un identificateur, car il sera interprété comme un patron de recherche, substitution, ou remplacement.

(Les variables qui commencent par un underscore étaient à l'origine dans le paquetage `main`, mais nous avons décidés qu'il serais plus utile aux programmeurs de paquetages de préfixer leurs variables locales et noms de méthodes avec un underscore. $_ est bien sur toujours global.)

Les chaînes qui sont utilisés avec eval() sont compilées dans le paquetage ou l'eval() à été compilé. (Les assignements à `$SIG{}`, d'un autre coté, supposent que le signal spécifié est dans le paquetage `main`. Mais vous pouvez dire au signal d'être dans le paquetage.) Par exemple, examinez *perldb.pl* dans la librairie Perl. Il passe dans le paquetage `DB` pour éviter que le débogueur n'interfère avec les variables du script que vous essayez de déboguer. De temps en temps, il revient au paquetage `main` pour évaluer différentes expressions dans le contexte du paquetage `main` (ou de la ou vous veniez). Referez vous a *perldebug*.

Le symbole spécial `__PACKAGE__` contient le paquetage courant, mais ne peut pas être (facilement) utilisé pour construire des variables.

Référez vous à *perlsub* pour de plus amples informations sur my() et local(), et *perlref* pour les détails.

2.2 Tables de symboles

Les tables de symboles pour un paquetage sont stockées dans un hash du même nom avec deux ":" a la fin. La table de symbole de main est donc `%main::`, ou `%::` pour raccourcir. De même, la table de symbole d'un paquetage imbriqué est nommée `%EXTERNE::INTERNE::`.

La valeur de chaque entrée du hash est ce à quoi vous vous référez quand vous utilisez la notation `*name`. En fait, les deux instructions suivantes ont le même effet, bien que la première soit plus efficace car il y a une vérification lors de la compilation :

```
local *main::truc    = *main::machin;
local $main::{truc}  = $main::{machin};
```

Vous pouvez les utilisez pour imprimer toutes les variables d'un paquetage, par exemple, la librairie standard *dumpvar.pl* et le module CPAN Devel::Symdump l'utilisent.

L'assignement a un typeglob crée juste un alias, par exemple :

```
*dick = *richard;
```

font que les variables, sous fonctions, formats, et noms de fichiers et de répertoires accessibles via l'identificateur `richard` aussi accessible via l'identificateur `dick`. Si vous voulez juste faire un alias d'une variable, ou d'une sous fonction, vous devrez assigner une référence a la place :

```
*dick = \$richard;
```

Ce qui fait de $richard et $dick la même chose, mais laisse les tableaux @richard et @dick différents. Pas mal hein ?

Ce mécanisme peut être utilisé pour passer et retourner des références vers ou depuis une sous fonction si vous ne voulez pas copier l'ensemble. Cela fonctionne uniquement avec les variables dynamiques, pas les lexicales.

```
%un_hash = ();                      # ne peut pas être my()
*un_hash = fn( \%un_autre_hash );
sub fn {
    local *hash_pareil = shift;
    # maintenant, utilisez %hashsym normalement,
    # et vous changerez aussi le %un_autre_hash
    my %nhash = (); # Faites ce que vous voulez
    return \%nhash;
}
```

Au retours, la référence écrasera le hash dans la table de symboles spécifié par le typeglob *un_hash. Ceci est une manière rapide de jouer avec les références quand vous ne voulez pas avoir a déréférencer des variables explicitement.

Une autre utilisation des tables de symboles est d'avoir des variables "constantes".

```
*PI = \3.14159265358979;
```

Maintenant, vous ne pouvez plus modifier $PI, ce qui est une bonne chose après tout. Ceci n'est pas la même chose qu'une sous fonction constante, qui est sujette a des optimisations lors de la compilation. Ce n'est pas la même chose. Un sous fonction constante est une qui ne prends pas d'arguments, et retourne une expression constante. Référez vous à *perlsub* pour plus de détails. Le pragma `use constant` est un truc pratique pour ce genre de choses.

Vous pouvez dire `*foo{PACKAGE}` et `*foo{NAME}` pour trouver de quels noms et paquetages le symbole *foo provient. Ceci peut être utile dans une fonction qui reçoit des typeglob comme arguments :

```
sub identify_typeglob {
    my $glob = shift;
    print 'Vous m\'avez donné ', *{$glob}{PACKAGE}, '::', *{$glob}{NAME}, "\n";
}
identify_typeglob *foo;
identify_typeglob *bar::baz;
```

Ceci imprimera

```
Vous m'avez donné main::foo
Vous m'avez donné bar::baz
```

La notation *foo{THING} peut aussi être utilisé pour obtenir une référence a des éléments de *foo. Référez vous à *perlref*.

2.3 Constructeurs et Destructeurs de paquetage

Il y a deux définitions de fonctions spéciales qui servent de constructeur et de destructeur de paquetage. Elles sont `BEGIN` et `END`. Le `sub` est optionnel pour ces deux routines.

Une fonction `BEGIN` est exécutée des que possible, c'est a dire, le moment ou le paquetage est complètement défini, avant que le reste du fichier soit parsé. Vous pouvez avoir plusieurs blocs `BEGIN` dans un fichier – ils seront exécutés dans l'ordre d'apparition. Parce que un bloc `BEGIN` s'exécute immédiatement, il peut définir des sous fonctions ainsi que pas mal de choses depuis d'autres fichiers pour les rendre visibles depuis le reste du fichier. Dès qu'un `BEGIN` a été exécuté, toutes les ressources qu'il utilisait sont détruites et sont rendues a Perl. Cela signifie que vous ne pouvez pas appeler explicitement un `BEGIN`.

Une fonction `END` est exécutée aussi tard que possible, c'est a dire, quand l'interpréteur se termine, même si sa sortie est due a un appel à die(). (Mais pas si il se relance dans un autre via `exec`, ou est terminé par un signal – vous aurez a gérer ça vous même (si c'est possible).) Vous pouvez avoir plein de blocs `END` dans un fichier – ils seront exécutés dans l'ordre inverse de leur définition, c'est à dire le dernier d'abord (last in, first out (LIFO)).

Au sein d'une fonction `END`, `$?` contient la valeur que le script va passer a `exit()`. vous pouvez modifier `$?` pour changer la valeur de sortie du script. Attention a ne pas changer `$?` par erreur (en lançant quelque chose via `system`).

Notez que lorsque vous utilisez **-n** et **-p** avec Perl, `BEGIN` et `END` marchent exactement de la même façon qu'avec **awk**, sous forme dégénérée. De la façon dont sont réalisés (et sujet a changer, vu que cela ne pourrais être pire), les blocs `BEGIN` *et* `END` sont exécutés lorsque vous utilisez **-c** qui ne fait que tester la syntaxe, bien que votre code principal ne soit pas exécuté.

2.4 Classes Perl

Il n'y a pas de syntaxe de classe spéciale en Perl, mais un paquetage peut fonctionner comme une classe si il fournis des fonctions agissant comme des méthodes. Un tel paquetage peut dériver quelques unes de ses méthodes d'une autre classe (paquetage) en incluant le nom de l'autre paquetage dans son tableau global @ISA (qui doit être global, pas lexical).

Pour plus de détails, référez vous à *perltoot* et *perlobj*.

2.5 Modules Perl

Un module est juste un paquetage qui est défini dans un fichier de même nom, et qui est destiné à être réutilisé. Il peut arriver a cette effet en fournissant un mécanisme qui exportera certains de ses symboles dans la table de symboles du paquetage qui l'utilise. Ou bien, il peut fonctionner comme une classe et rendre possible l'accès a ses variables via des appels de fonctions, sans qu'il soit nécessaire d'exporter un seul symbole. Il peut bien sur faire un peu des deux.

Par exemple, pour commencer un module normal appelé Some::Module, Créez un fichier appelé Some/Module.pm et commencez avec ce patron :

```
package Some::Module;  # suppose Some/Module.pm

use strict;

BEGIN {
    use Exporter   ();
    use vars       qw($VERSION @ISA @EXPORT @EXPORT_OK %EXPORT_TAGS);

    # On défini une version pour les vérifications
    $VERSION     = 1.00;
    # Si vous utilisez RCS/CVS, ceci serais préférable
    # le tout sur une seule ligne, pour MakeMaker
    $VERSION = do { my @r = (q$Revisio: XXX $ =~ /\d+/g); sprintf "%d."."%02d" x $#r, @r };

    @ISA         = qw(Exporter);
    @EXPORT      = qw(&func1 &func2 &func4);
    %EXPORT_TAGS = ( );     # ex. : TAG => [ qw!name1 name2! ],

    # vos variables globales a être exporter vont ici,
    # ainsi que vos fonctions, si nécessaire
    @EXPORT_OK   = qw($Var1 %Hashit &func3);
}
use vars      @EXPORT_OK;

# Les globales non exportées iront là
use vars      qw(@more $stuff);

# Initialisation de globales, en premier, celles qui seront exportées
$Var1   = '';
%Hashit = ();

# Ensuite, les autres (qui seront accessible via $Some::Module::stuff)
$stuff  = '';
@more   = ();

# Toutes les lexicales doivent être crées avant
# les fonctions qui les utilisent.

# les lexicales privées vont là
my $priv_var    = '';
my %secret_hash = ();

# Voici pour finir une fonction interne a ce fichier,
# Appelée par &$priv_func;  elle ne peut être prototypée.
my $priv_func = sub {
    # des trucs ici.
};

# faites toutes vos fonctions, exporté ou non;
# n'oubliez pas de mettre quelque chose entre les {}
sub func1      {}    # pas de prototype
sub func2()    {}    # proto void
sub func3($$)  {}    # proto avec 2 scalaires
```

```
# celle là n'est pas exportée, mais peut être appelée !
sub func4(\%)  {}    # proto'd avec 1 hash par référence

END { }       # on met tout pour faire le ménage ici (destructeurs globaux)
```

Enfin, continuez en déclarant et en utilisant vos variables dans des fonctions sans autres qualifications. Référez vous a *Exporter* et *perlmodlib* pour plus de détails sur les mécanismes et les règles de style a adopter lors de la création de modules.

Les modules Perl sont inclus dans vos programmes en disant

```
use Module;
```

ou

```
use Module LIST;
```

Ce qui revient exactement a dire

```
BEGIN { require Module; import Module; }
```

ou

```
BEGIN { require Module; import Module LIST; }
```

Et plus spécifiquement

```
use Module ();
```

est équivalent a dire

```
BEGIN { require Module; }
```

Tous les modules perl ont l'extension *.pm*. `use` le suppose pour que vous n'ayez pas a taper "*Module.pm*" entre des guillemets. Ceci permet aussi de faire la différence entre les nouveaux modules des vieux fichiers *.pl* et *.ph*. Les noms de modules commencent par une majuscule, a moins qu'ils fonctionnent comme pragmas, les "Pragmas" sont en effet des directives du compilateur, et sont parfois appelés "modules pragmatiques" (ou même "pragmata" si vous êtes puristes).

Les deux déclarations :

```
require UnModule;
require "UnModule.pm";
```

différent en deux points. Dans le premier cas, les doubles deux points dans le nom du module, comme dans `Un::Module`, sont transformés en séparateur système, généralement "/". Le second ne le fait pas, ce devra être fait manuellement. La deuxième différence est que l'apparition du premier `require` indique au compilateur que les utilisations de la notation objet indirecte impliquant "UnModule", comme dans `$ob = purge UnModule`, sont des appels de méthodes et non des appels de fonctions. (Oui, ceci peut vraiment faire une différence).

Parce que l'instruction `use` implique un bloc `BEGIN`, l'importation des sémantiques intervient au moment ou le `use` est compilé. C'est de cette façon qu'il lui est possible de fonctionner comme pragma, et aussi la manière dont laquelle les modules sont capables déclarer des fonctions qui seront visibles comme des opérateurs de liste pour le reste du fichier courant. Ceci ne sera pas vrai si vous utilisez `require` a la place de `use`. Avec `require`, vous allez au devant de ce problème :

```
require Cwd;                # rends Cwd:: accessible
$here = Cwd::getcwd();

use Cwd;                    # importe les noms depuis Cwd::
$here = getcwd();

require Cwd;                # rends Cwd:: accessible
$here = getcwd();           # oups ! y'a pas de  main::getcwd()
```

En général, `use Module ()` est recommandé a la place de `require Module`, car cela détermine si le module est la au moment de la compilation, pas en plein milieu de l'exécution de votre programme. Comme exception, je verrais bien, le cas ou deux modules essayent de se `use` l'un l'autre, et que chacun appelle une fonction de l'autre module. Dans ce cas, il est facile d'utiliser `require` a la place.

Les paquetages Perl peuvent être inclus dans d'autre paquetages, on peut donc avoir des noms de paquetages contenant `::`. mais si nous utilisons le nom du paquetage directement comme nom de fichier, cela donnera des nom peu manipulables, voir impossibles sur certains systèmes. Par conséquent, si le nom d'un module est, disons, `Texte::Couleur`, alors, la définition se trouvera dans le fichier *Texte/Couleur.pm*.

Les modules Perl ont toujours un fichier *.pm*, mais ils peuvent aussi être des exécutables dynamiquement liés, ou des fonctions chargées automatiquement associées au module. Si tel est le cas, ce sera totalement transparent pour l'utilisateur du module. c'est le fichier *.pm* qui doit se charger de charger ce dont il a besoin. Le module POSIX est en fait, dynamique et autochargé, mais l'utilisateur a juste a dire `use POSIX` pour l'avoir.

Pour plus d'informations sur l'écriture d'extensions, referez vous a *perlxstut* et *perlguts*.

3 VOIR AUSSI

perlmodlib pour les questions générales sur comment faire des modules et des classes Perl, ainsi que la description de la librairie standard et du CPAN, *Exporter* pour savoir comment marche le mécanisme d'import/export de Perl, *perltoot* pour des explications en profondeurs sur comment créer des classes, *perlobj* pour un document de référence sur les objets, et *perlsub* pour une explication sur les fonction et la portée de celle ci.

4 TRADUCTION

4.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.005_02. Pour en savoir plus concernant ces traductions, consultez http://perl.enstimac.fr/.

4.2 Traducteur

Mathieu Arnold <arn_mat@club-internet.fr>

4.3 Relecture

Simon Washbrook <swashbro@tlse.marben.fr>

5 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.