PB173 Perl

08 Menné priestory

Roman Lacko xlacko1@fi.muni.cz

2023-04-03

Obsah

Menné priestory

Menné priestory

Každý globálny symbol v Perli žije v nejakom **mennom priestore** (*namespace*).
Perl tieto priestory volá ‹package›.

Východzí menný priestor sa volá ‹main›. Okrem neho sme už videli ‹CORE› a ‹builtin›.

```
sub f(@n);

f(1, 2);                # All are the same.
main::f(1, 2);
::f(1, 2);
```

Terminológia

(plne) kvalifikovaný symbol	Meno symbolu s jeho menným priestorom.
nekvalifikovaný symbol	Meno symbolu bez menného priestoru.

Menné priestory

Symbol môžeme vytvoriť v mennom priestore priamo:

```
sub Portal::Spheres::space() {
    say "I'm in SPAAAAAAACE!";
}

space();                    # Compile error, searches for ‹main::space()›
Portal::Spheres::space();   # OK

$Portal::WEIGHTED_COMPANION_CUBES++;
```

Menné priestory môžeme do seba vnorovať.

Menné priestory: ‹package›

```
package NAMESPACE;
package NAMESPACE BLOCK;
```

Zmení aktuálny menný priestor na ‹NAMESPACE›.
Nekvalifikované symboly sa budú hľadať v danom mennom priestore.

Ak sa pridá aj ‹BLOCK›, potom zmení menný priestor len pre daný blok.

Menný priestor môžeme opustiť a vstúpiť doňho neskôr alebo v inom súbore.

Menné priestory: ‹package›

Kľúčové slovo ‹package› menný priestor vždy *prepína*.
Menný priestor nededí prefix od predchádzajúceho.

```
package Multiverse;              # Switch to ‹Multiverse›
package Froopyland;              # Switch to ‹Froopyland›
sub portal();                    # ‹Froopyland::portal()›
```

Ak chceme vyrobiť symbol v ‹Multiverse::Froopyland›,
musíme použiť vždy celé meno:

```
package Multiverse::Froopyland;
sub portal();                    # ‹Multiverse::Froopyland::portal()›
```

Menné priestory: ‹package›

‹package NAMESPACE BLOCK› prepne menný priestor len pre daný blok.

Ani tu sa nededí prefix menného priestoru.

```
package Multiverse {
    sub portal();               # ‹Multiverse::portal()›

    package Froopyland {
        sub portal();           # ‹Froopyland::portal()›
    }

    package Multiverse::Froopyland {
        sub portal();           # ‹Multiverse::Froopyland::portal()›
    }
}
```

Menné priestory

Len ‹package› dokáže prepnúť menný priestor.

```
package Arrakis;

sub Caladan::governor();
sub Caladan::call() {       # ‹Caladan::call()›
    governor();             # WRONG, ‹Arrakis::governor()›

    Arrakis::governor();    # OK, but meh
    package Arrakis;        # OK, but even mehher
    governor();
}
```

Používajte len na prekrývanie funkcií z iného menného priestoru.

Deklarátory premenných

Deklarátory premenných

Perl s ‹use strict› pragmou nedovolí použiť premennú
bez deklarácie *alebo* plne kvalifikovaného mena.

```
$x = 10;                # Global symbol "$x" requires explicit package name…
my $x = 10;             # OK, local variable declaration
```

Plne kvalifikované premenné sú viazané na menný priestor.

```
sub increase() {
    $C::x++;            # Package variable
}

increase for 0 .. 10;
say $C::x;              # Still works
```

Deklarátory premenných

Premenné deklarujeme kľúčovými slovami:

```
my $a = 1;              # Lexical variable for the given scope.
our $c = 2;             # Lexical alias to a package variable.

state $b = 10;          # Lexical variable, initialized only once.
```

Pre úplnosť existuje ešte `local`

```
local $d;               # Dynamically scoped variable.
```

Deklarátory premenných: ‹my›

```
my VARIABLE [= VALUE];
my (LIST) [= LIST];
```

Premenná ‹VARIABLE› alebo zoznam premenných ‹LIST› existujú len v danom bloku alebo súbore.

Deklarátory premenných: ‹my›

‹package NAME;› neruší platnosť lexikálnych premenných

```
my $target = "Earth";

package Outer {
    say $target;            # Earth
    my $target= "Omicron Persei 8";

    package Inner;
    say $target;            # Omicron Persei 8
}

say $target;                # Earth
```

Deklarátory premenných: ‹our›

```
our VARIABLE [= VALUE]
our (LIST) [= LIST];
```

Vytvorí **lexikálny** alias na premennú v aktuálnom mennom priestore.

```
package Winterfell;
$Winterfell::ZOMBIES = 9143658;

sub report() {
    say $ZOMBIES;       # Global symbol "$ZOMBIES" requires …

    our $ZOMBIES;       # ‹$ZOMBIES› → ‹$Winterfell::ZOMBIES›
    say $ZOMBIES;       # 9143658
}
```

Deklarátory premenných: ‹our›

‹our› je lexikálny alias, rovnako ako ‹my›!

```
package Smurfs;
our $VILLAIN = "Gargamel";  # ‹$VILLAIN› → ‹$Smurfs::VILLAIN›

package Dune;
say $VILLAIN;               # Gargamel
```

Riešením je

- Nepoužívať ‹our› a ‹my› v globálnom rozsahu
- Oddeľovať menné priestory do modulov (neskôr)

Deklarátory premenných: ‹state›

```
state VARIABLE [= VALUE];
state (LIST) [= LIST];
```

Ako ‹my›, ale premennú inicializuje len raz.

Ekvivalent ‹static› premennej v C a C++.

```
sub counter() {
    state $value = 0;
    return $value++;
}

say counter() for 0 .. 3;      # 0 1 2 3
```

Deklarátory premenných: ‹local›

```
local EXPR
```

Prekryje hodnotu premennej alebo výrazu do konca bloku,
efekt sa prejaví aj vo volaných funkciách.

Prekryť môžeme

- globálne premenné (*package variables*).
- časti (normálnych aj asociatívnych) polí, symbolické referencie.

Nemôžeme prekryť lexikálne premenné (‹my›, ‹state›).
Prečo?

Deklarátory premenných: ‹local›

```
$::USER = "client";

sub identify() {
    say $::USER;
}

sub sudo() {
    local $::USER = "root";
    identify();
}

identify();     # client
sudo();         # root
identify();     # client
```

Deklarátory premenných: ‹local›

```
my $AUTH = { login => 'client' };

sub identify() {
    say $AUTH->{login};
}

sub sudo() {
    local $AUTH->{login} = 'root';
    identify();
}

identify();     # client
sudo();         # root
identify();     # client
```

Moduly

Moduly: Režimy interpretu

Od spustenia skriptu až do jeho konca prechádza interpret rôznymi fázami.
V týchto fázach je možné registrovať vlastné funkcie pomocov špeciálnych blokov.

`BEGIN` Fáza kompilácie skriptu. Bloky `BEGIN` bežia čo najskôr.

`END` Tesne pred koncom skriptu, vrátane `die`.

`UNITCHECK`, `CHECK`, `INIT`
Stavy pri prepínaní kompilačnej a behovej fázy skriptu.

Moduly: ‹BEGIN›

Pre každý režim môžeme definovať blok, ktorý sa v tej fázi spustí.
Týchto blokov môže byť viac, preto ich píšeme bez ‹sub›.

```
BEGIN { ... }
```

- Spustí sa ihneď, ako parser prečíta koncovú ‹}›.
  - V tomto stave ešte Perl nepozná zvyšok kódu.
- Užitočné na import modulov a zapínanie pragiem.

Moduly: ‹END›

```
END { ... }
```

- Spúšťa sa tesne pred ukončením interpretu, vrátane ‹die()›.
  - Ale nie pri ‹exec()›!
- Interpret bloky spúšťa zo zásobníka (LIFO).
- Premenná ‹$?› drží hodnotu, ktorú Perl vráti systému, môže sa v tomto bloku zmeniť.

Moduly

Perl moduly sú obyčajné Perl skripty, typicky s príponou `.pm`.

Modul typicky obsahuje menný priestor odvodený od mena súboru, napr. `File/Find.pm` → `File::Find`.

Hello.pm

```
use v5.36;

package Hello;

sub greet() {
    say "Hello!";
}

1;
```

Moduly: ‹do›

```
do BLOCK
do FILENAME
```

Vykoná ‹BLOCK› alebo obsah súboru ‹FILENAME›.

V princípe podobné ‹eval qx"cat FILENAME"›, ale

- Kód v ‹eval› vidí lokálne premenné; ‹do› ich do ‹FILENAME› nepropaguje.
- Pri chybe ‹do› zobrazí chybu v kontexte ‹FILENAME›.

Moduly: ‹require›

```
require FILENAME
require MODULE
```

Chytrejšie ‹do FILENAME›, ktoré je vhodné na sprístupnenie modulu.

V prípade, že je zadaný názov modulu ako *bareword*, potom nahradí ‹::› za ‹/› a pridá príponu ‹.pm›:

```
require Some::Module;
require "Some/Module.pm";       # Same thing.
```

Moduly môžu mať inicializačný kód, ktorý musí indikovať úspech.
V prípade, že modul pri inicializácii nevráti pravdivú hodnotu, ‹require› vyhodí výnimku.

Na rozdiel od ‹do FILENAME› sa súbor importuje vždy len raz.

Moduly: ‹require›

Určenie cesty k modulu:

‹@INC›	zoznam ciest, kde sa hľadajú moduly
‹%INC›	mapovanie parametrov ‹require› na absolútne cesty k modulom

Premennú ‹@INC› je možné meniť v ‹BEGIN› bloku pred ‹require›,
prepínačom ‹-I PATH› pre interpret, alebo pohodlne pragmou ‹lib›:

```
use FindBin '$RealBin';
use lib $RealBin;
# Now we can require local modules
```

‹require› sa dá použiť na viac vecí, viď ‹perldoc -f require›.

Moduly: ‹use›

```
use MODULE
use MODULE LIST
```

Ako ‹require›, ale zároveň zavolá metódu ‹import()› na module,
ktorá tak môže vykonať nejakú inicializáciu. Zhruba ekvivalentné bloku:

```
BEGIN {
    require MODULE;
    MODULE->import(LIST);
}
```

‹no MODULE› funguje podobne, volá však metódu ‹unimport()›.

Moduly: ‹caller›

```
caller
caller N
```

Vráti informácie o kontexte funkcie, tj. o volajúcom kóde.
S argumentom vráti viac informácií o ‹N›-tom predkovi na zásobníku.

```
my ($package,   $filename, $line) = caller;

my ($package,   $filename, $line, $subroutine, $hasargs,
    $wantarray, $evaltext, ...) = caller 1;
```

Ak ‹caller› zavoláte v mennom priestore ‹DB›,
vyplní ‹@DB::args› parametrami funkcie.

Moduly: Metóda ‹import›

```
sub import($package, @args);
```

Metóda modulu, ktorú zavolá ‹use› s argumentami. Nemusí v module existovať.

Jej úlohou je obvykle exportovať požadované symboly do menného priestoru volajúceho kódu.

OOP bude podrobnejšie vysvetlené na budúcom seminári.

```
sub import($package, $symbol) {
    # Export our symbol into caller's namespace.
    my $our_symbol = join '::', $package, $symbol;
    my $their_symbol = join '::', (caller)[0], $symbol;
    *$their_symbol = *$our_symbol;
}
```

Moduly: Metóda ‹unimport›

```
sub unimport($package, @args);
```

Metóda, ktorú zavolá ‹no MODULE LIST›.

Obvykle sa používa na implementáciu vlastných pragiem.
Táto konštrukcia pragmu vypne.

```
{
    use Module 'foo';           # Module->import('foo');
    {
        no Module 'bar';        # Module->unimport('bar');
    }
}
```

Moduly: ‹Exporter›

Pohodlný spôsob exportu symbolov bez nutnosti implementácie ‹import()›.

```
package Hello;

our parent 'Exporter';
our @EXPORT = qw(hello);
our @EXPORT_OK = qw(greet);

sub hello() { ... }
sub greet() { ... }

1;
```

Špeciality

Deklarátory pre funkcie

Definícia funkcie pomocou ‹sub› vytvára globálny symbol:

```
package Navy;
sub ship(@people);
sub Navy::ship(@people);     # Same
```

Môžeme použiť ‹my›, ‹our› a ‹state›:

```
my sub ship(@people);
our sub ship(@people);
state sub ship(@people);
```

Deklarátory pre funkcie

```
package Navy;
our sub ship(@people);

package Army;
ship();                 # Calls ‹Navy::ship()›
```

Funkcia ‹AUTOLOAD›

Ak v mennom priestore, v ktorom existuje funkcia ‹AUTOLOAD› zavoláme neznámu funkciu, namiesto výnimky sa zavolá ‹AUTOLOAD›.

Kvalifikované meno funkcie sa uloží v globálnej premennej ‹AUTOLOAD› pre daný menný priestor.

```
sub Foo::AUTOLOAD(@args) {
    say "Called $Foo::AUTOLOAD(@args)";
}

Foo::test(1);           # Called Foo::test(1)
```

Typické použitia:

- Generovanie rozhraní z deklaratívneho popisu (napr. *XMLRPC API*).
- Testovanie (*mock interface*).