PB173 Perl

09 Menné priestory

Roman Lacko <xlacko1@fi.muni.cz>

Obsah

Menné priestory

Každý 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`, v ktorom žijú zabudované Perl funkcie.

```
sub f(@n);              # Declaration.

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

Menné priestory

Symbol môžeme vytvoriť v mennom priestore priamo:

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

space();                        # Compile error!
Portal::Spheres::space();       # OK
```

Menné priestory môžeme do seba vnorovať.

Menné priestory

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

Zmení aktuálny menný priestor na `NAMESPACE`.

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

Kľúčové slovo package

S názvrom menného priestoru bez bloku prepne menný priestor pre všetky symboly *bez* plného mena.

```
sub magic();            # ‹main::magic()›.

package Hogwarts;
sub magic();            # ‹Hogwarts::magic()›
magic();                # Calls ‹Hogwarts::magic()›
main::magic();          # Calls ‹main::magic()›

package main;           # This is the default package.
magic();                # Calls ‹main::magic()›
Hogwarts::magic();      # Calls ‹Hogwarts::magic()›
```

Do menného priestoru môžeme vstúpiť aj opakovane.

Kľúčové slovo `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()›
```

Kľúčové slovo `package`

Ak za `package NAMESPACE` uvedieme blok, potom sa zmena uplatní len v danom bloku.

Ani tu sa však 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 main;

sub P::foo();
sub P::bar() {          # This is ‹P::bar()›
    foo();              # but this would still call ‹main::foo()›!

    package P;          # OK, but meh
    foo();
}
```

Syntax vyššie používajte len na prekrývanie funkcií z iného menného priestoru. Ak to vôbec potrebujete.

Deklarátory premenných

Perl s `use strict` pragmou nedovolí použiť premennú bez deklarácie *alebo* plného mena.

```
$x = 10;                # Wrong, ‹$x› is undeclared.
my $x = 10;             # OK, local variable declaration.
```

Premenné vytvorené v mennom priestore (plným menom alebo `our`) sú viazané na daný menný priestor, nie na syntaktický blok.

```
sub increase() {
    $C::x++;            # OK, package variable, no ‹my› needed.
}

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

Deklarátory premenných

Premenné deklarujeme kľúčovými slovami:

```
my $a = 1;              # Lexical variable for the given scope.
state $b = 10;          # Lexical variable, initialized only once.
our $c = 2;             # Alias for a package variable.
```

Pre úplnosť existuje ešte `local`

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

Deklarátor `my`

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

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

`package NAME` rozsah týchto premenných neobmedzuje.

```
{
    package Foo;
    my $x = 42;             # This is «not» ‹$Foo::x›, just ‹$x›.
    package Bar;
    say $x;                 # Works, because ‹$x› is still in scope.
# Here ‹$x› goes out of scope.
}
```

Deklarátor `our`

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

Vytvorí lexikálny alias na *package variable* v aktuálnom mennom prostredí.

```
package P;
$P::var = 1;

sub foo() {
    say $var;           # Wrong, undeclared lexical variable.

    our $var;           # Make ‹$var› a lexical alias for ‹$P::var›.
    say $var;           # OK
}
```

Deklarátor `state`

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

Ako `my`, ale premennú inicializuje len raz.

Ako lokálna `static` premenná v C a C++.

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

say counter() for 0 .. 3;      # 1 2 3 4
```

Deklarátor `local`

```
local EXPR
```

Dočasne 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 lokalizovať lexikálne premenné (`my`, `state`).
Prečo?

Deklarátor `local`

```
$::x = 1;

sub foo() {
    say $::x;
}

sub bar() {
    local $x = 2;
    foo();
}

foo();                  # 1
bar();                  # 2
foo();                  # 1 again
```

Moduly

Režimy interpretu

Od spustenia skriptu až do jeho konca prechádza interpret rôznymi fázami. V týchto fázach je možné spustiť nejaký kód.

`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.

Režimy interpretu

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 ukončí `}`.
  - Teda ešte predtým, ako sa číta zvyšok kódu.
  - Logicky bežia v poradí, v akom ich interpret nájde (FIFO).
- Užitočné na import modulov a zapínanie pragiem.

Režimy interpretu

```
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.

Modulový systém

Perl moduly sú obyčajné Perl skripty, typicky s príponou `.pm`.
Modul obvykle definuje nejaký menný priestor, aby nedochádzalo ku konfilktu.

Modul vkladá kľúčové slovo `use`.

Hello.pm

```
use v5.32;

package Hello;

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

if (!caller) {
    greet();
}

1;
```

Kľúčové slovo `do`

```
do BLOCK
do FILENAME
```

Vykoná `BLOCK` alebo obsah súboru `FILENAME`.

V princípe podobné ako

```
eval qx"cat FILENAME"
```

s týmito rozdielmi:

- kód v `eval` vidí lokálne premenné, `do` ich do `FILENAME` nepropaguje,
- pri chybe `do` zobrazí chybu v kontexte `FILENAME`

Kľúčové slovo `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.

Kľúčové slovo `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 `use`,
prepínačom `-I PATH` pre interpret, alebo pohodlne pragmou `lib`:

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

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

Kľúčové slovo `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()`.

Odbočka: `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;
```

`$wantarray` je možné zistiť aj operátorom `wantarray`.

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;
}
```

Metóda `unimport`

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

Metóda modulu, 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');
    }
}
```

Modul `Exporter`

Pohodlný spôsob exportu symbolov bez nutnosti implementácie `import()`.

```
package Hello;

# ‹Exporter::import()› gets inherited.
our parent 'Exporter';

# Symbols that will be exported by default.
our @EXPORT = qw(hello);

# Symbols that can be exported if asked by ‹use Hello SYMBOLS...›.
our @EXPORT_OK = qw(greet);

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

1;
```

Funkcia AUTOLOAD

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

Plné meno originálnej funkcie sa uloží v globálnej premennej (*package variable*) $‹PACKAGE›::AUTOLOAD.

```
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*).