COURS PRATIQUE PERL

Marc SCHAEFER

1er octobre 2005

Programme

- Motivations de ce cours pratique
- Premier projet pratique
- Notions de base
- Accès SGBD : DBI
- Echange de données XML
- Réutilisation de code
- Conclusion
- Références
- Questions

Motivations de ce cours pratique

Perl offre ...

- un excellent langage efficace dans plusieurs domaines
- un environnement bien connu, intégrant les concepts UNIX usuels
- une large bibliothèque logicielle (**modules**)
- un langage assez complexe et très libre, portable

Notre but : présenter la puissance de Perl dans des applications concrètes et variées, apprendre les rudiments, éviter les écueils classiques.

N'est pas notre but : décrire des fonctions peu utilisées ou obscures de Perl, les modes non 'programmation' de Perl (voir `man perlrun`).

Perl est un langage qui peut s'apprendre par couches. Commençons par la couche la plus stable :)

Perl : historique

- Practical Extraction and Report Language
- 1987, Larry Wall, version 1.000
- au début : fusion des fonctionnalités de `sed`, **C**, `awk` et du shell UNIX `sh` (Bourne shell).
- Perl 5 : 1994, réécriture.

`http://history.perl.org/PerlTimeline.html`

Perl : domaines d'application

- administration système, lorsque le shell n'est plus suffisant (performance, maintenance, réutilisabilité, sécurité, ...)
- transformation/traitement de données
- prototypage
- applications WWW
- applications générales

résoudre des **problèmes !**

Dans mon expérience personnelle, je tape des commandes shell. Lorsqu'elles deviennent trop complexe, il m'arrive d'utiliser Perl en mode petit programme (un peu comme awk). Et quand cela devient encore plus compliqué, je fais un script shell. Et si c'est vraiment quelque chose de complexe que je pourrais réutiliser un jour, j'en fais un script Perl.

Parfois il m'arrive de grouper des fonctions souvent utilisées dans un module Perl, voire même dans un package Debian local.

Et de temps en temps je fais de véritables développements en Perl, planifiés et structurés dès le début.

Premier projet pratique

On désire effectuer des statistiques de courrier électronique (serveur Postfix) : émetteur, destinataire, longueur, etc.

Le format source ressemble à :

```
Dec  7 06:50:15 shakotay postfix/qmgr[30003]: 0B4E93CC261:
  from=<pgsql-general-owner+M53780=postgresql@postgresql.org>,
  size=3879, nrcpt=1 (queue active)

Dec  7 06:50:23 shakotay postfix/pipe[19695]: 0B4E93CC261:
   to=<alphanet-postgresql-general@alphanet.ch>, relay=filter,
   delay=8, status=sent (shakotay.alphanet.ch)
```

But :

Il s'agit de réunir les lignes correspondant à un même `Message-ID`, ici `0B4E93CC261` (que l'on considère unique pour un fichier de log donné) et d'en extraire les informations utiles.

Analyse du problème

- il s'agit d'un format texte, relativement structuré, orienté ligne : les outils UNIX de base et Perl sont donc de bons candidats
- il s'agit de relier les lignes qui comportent le même `Message-ID`
- dans une première approche on peut négliger le fait qu'un message peut être rejeté ou adressé à plusieurs destinataires, ou encore être retransmis.

Comment feriez-vous ?

Proposez des idées, dans n'importe quel langage. Notamment insistez sur les points fondamentaux, les outils nécessaires, etc.

Une façon correcte de répondre à cette question pourrait être : je cherche un module Perl qui a déjà été écrit et qui implémente cette fonctionnalité (ou n'importe quoi en n'importe quel langage, en logiciel libre).

Mais bien sûr, notre but est de montrer les rudiments de la programmation Perl !

Outils nécessaires (1)

- le programme se bornera donc à classer des informations en utilisant le `Message-ID` comme clé
  - à l'esprit viennent immédiatement les outils de base de données (SGBD), les concepts d'indexes et de fichiers indexés (DBM) et de structures de données en arbres
  - ou plus simplement les tableaux associatifs (*hash tables*, tables de hâchage), offerts par Perl

Outils nécessaires (2)

- l'extraction des données peut se faire en décrivant chaque ligne sous forme d'une expression régulière
  - des outils comme `sed` ou `awk` permettent de sortir des données de formats connus assez facilement
  - Perl intègre et étend ces concepts d'une manière assez intuitive.

Pseudo-code

Notre programme est un *filtre* typique :

1. lire une ligne de l'entrée standard
2. déterminer si la ligne est une ligne d'expéditeur ou de destinataire (livraison), cela peut se faire en vérifiant laquelle des expressions régulières (**regexp**) s'applique
3. extraire soit les informations ('from', 'size', 'message-id') ou les informations ('to', 'message-id')
4. pairer ces informations par 'message-id'
5. afficher les données dans le format désiré sur la sortie standard

Il nous manque encore les notions de base de Perl pour implémenter cela, nous allons les apprendre au fur et à mesure.

Notions de base

- structure de base d'un programme
- appel d'un programme Perl
- définitions et utilisation de variables, types de base
- boucles et contrôle
- travail d'entrée/sortie (fichiers)
- documentation

`man perl`, `man perlintro`

Structure de base d'un programme

```
#! /usr/bin/perl -w

use strict;

print "Hello world!\n";

exit 0;
```

`man perlsyn`

Appel d'un programme Perl

- p.ex. `./nom_programme`, avec `#! /usr/bin/perl` et permission `x`
- `perl nom_programme`
- `perl -e 'texte; du; programme;'`
- parfois le suffixe `.pl` est utilisé

`man perlrun`

Fonctions

```
#! /usr/bin/perl -w

use strict;

sub calcul($$) {
   my ($a, $b) = @_;

   return $a + $b;
}

print calcul(42, 24), "\n";

exit 0;
```

Variables et types de base

– utilisation de variables :

déclaration `my $variable;`

affectation `$variable = 1;`

suppression `undef $variable;`

– trois types de base
  – scalaire (scalar) : `$variable`
  – tableau (array, list) : `@array`
  – tableau de hâchage (hash) : `%hash`
– type référence : scalaire !

`man perldata`, `man perlvar`, `man perlref`

La suppression rend le contenu de la variable non initialisé. L'opérateur `defined($variable)` permet de tester si une variable a été initialisé. On utilise souvent cette propriété de manière similaire à la valeur `NULL` des bases de données.

Hashes

– **déclaration** `my %hash;`

affectation `$hash{$key} = $value;`

affectation

```
my %hash = ($key => $value, $key2 => $value2, ......);
```

suppression `delete $hash{$key};`

test d'existence `exists($hash{$key})`

Références

– p.ex. : passage par référence d'une valeur à une fonction
– prise de référence sur une variable
– exemples
  – `my $hash_ref = \%hash;`
  – `my $var_ref = \$variable;`
  – `my $array_ref = \@array;`
– application : tableau de tableaux, hash de hashes, structures en arbre, etc.

Boucles et contrôle

- `while ($condition) { operation; }`
- `if ($condition) { op1; } elsif ($cond2) { op2; } else { op3; }`

`man perlrun`

Entrées/sorties

- noms de fichiers standard d'UNIX
  - STDIN
  - STDOUT
  - STDERR
- opérateur diamant : lecture d'une ligne
  - `<STDIN>`
  - <> : STDIN ou les fichiers spécifiés en ligne de commande, un après l'autre
- sortie avec `print`

`man perlopentut`, `man perlop`

Expressions régulières

- `^` début, `$` fin, `[]` ensemble, `.` un caractère quelconque, `.*` wildcard, etc, etc.
- `if ($var =~ /$regexp/) { ... }` en cas de groupage via `()`, variables `$1` et autres
- substitutions avec `$var =~ s/SRC/DEST/gi`
- développons une regexp passive pour notre cas
- groupage (vers une regexp active)

`man perlre`

Documentation

- `man perl` et pages man référencées
- `perldoc`, p.ex. `perldoc -f map`

Revenons à notre but

A l'aide du chablon (skeleton) dans `../exemples/mail-parser/skeleton`, de la documentation sur votre système (voir les transparents précédents) ainsi que des exemples dans `../exemples/input-output` et `../exemples/hashes` implémentez une version simplifiée de notre `parser`.

Je passerai pour résoudre vos problèmes ou répondre à vos questions.

Ensuite nous examinerons ensemble une solution possible.

Solution possible

```
#! /usr/bin/perl -w

use strict;

my $sender_regexp = '[^:]+: ([^:]+): from=<([^>]+)>, size=([0-9]+), ';
my $delivery_regexp = '[^:]+: ([^:]+): to=<([^>]+)>, ';

my $result = 0; # SUCCESS
my $error_reason = "unknown";

my %messages;

sub dump_messages($$);

# BUGS
#    - only the last error will be shown

while (my $line = <STDIN>) {
   if ($line =~ /$sender_regexp/) {
```



```
      my ($message_id, $from, $size) = ($1, $2, $3);

      if (exists($messages{$message_id})) {
         $result = 1; # FAILURE
         $error_reason = 'duplicate Message-ID or wrong sequencing';
      }
      else {
         $messages{$message_id} = { 'from' => $from,
                                    'size' => $size,
                                    'to' => []
                                  };
      }
   }
   elsif ($line =~ /$delivery_regexp/) {
      my ($message_id, $to) = ($1, $2);

      if (!exists($messages{$message_id})) {
         $result = 1; # FAILURE
         $error_reason = 'no matching Message-ID or wrong sequencing';
      }
```



```
      else {
         push(@{$messages{$message_id}->{'to'}},
                          $to);
      }
   }
   else {
      print STDERR "tossing: ", $line, "\n";
      $result = 1; # FAILURE
      $error_reason = 'tossed unparseable lines';
   }
}

if ($result == 0) {
   my $msg = dump_messages(\%messages, \$error_reason);
   if (defined($msg)) {
      print $msg, "\n";
   }
   else {
      $result = 1; # FAILURE
      $error_reason = 'dump_messages() failed: ' . $error_reason;
```

```
    }
}

if ($result != 0) {
   print STDERR $0, ": failed: ", $error_reason, "\n";
}

exit $result;

sub dump_entry($$$) {
   my ($entry_ref, $spaces, $error_reason_ref) = @_;

   my $str = '';

   foreach my $key (keys %{$entry_ref}) {
      $str .= $spaces . $key . ': ';

      if ($key eq 'to') {
         $str .= join(', ', @{$entry_ref->{$key}});
      }
```



```
      else {
         $str .= $entry_ref->{$key};
      }

      $str .= "\n";
   }

   return $str;
}

sub dump_messages($$) {
   my ($messages_hash_ref, $error_reason_ref) = @_;

   my $str = '';

   while (my ($key, $value) = each(%{$messages_hash_ref})) {
      $str .= 'message-id: ' . $key . "\n";

      my $tmp = dump_entry($value, '   ', $error_reason_ref);
      if (defined($tmp)) {
```



```
         $str .= $tmp;
      }
      else {
         $$error_reason_ref = 'failed with '
                            . $key
                            . ': '
                            . $$error_reason_ref;
         return undef;
      }

      $str .= "\n";
   }

   return $str;
}
```


Erreurs Perl typiques

- oubli de `-w` et `use strict;`
- oubli/mauvaise utilisation de `my $var;` (ou `local`)
- abus du `$_` (p.ex. comme variable de contrôle d'une boucle qui appelle une fonction ou dans une autre boucle)
- `$a eq $b` et `$a ne $b` sont les fonctions de comparaison de chaîne (== et != sont numériques !)
- nombre et type de paramètres à une fonction, pas de prototypes
- traitement de texte SQL plutôt que les opérateurs de binding, danger de cross-scripting, et SQL injection

- cross-scripting WWW si mauvaise utilisation de `HTML::Entities` et de `URI::Escape`
- version à 2 paramètres de `open`
- danger du contexte de liste (p.ex. `<STDIN>` vs `(<STDIN>)`

Il y a malheureusement quelques constructions dangereuses en Perl, ou quelques syntaxes ou sémantiques non triviales. En produisant du code clair et pas trop concis, on évite en général la plupart des écueils.

Amélioration de mail-parser : stockage

Le but est, plutôt que d'afficher les résultats à l'écran, de **stocker** les données de manière permanente.

Avez-vous des propositions ? Quelles méthodes connaissez-vous dans d'autres langages ?

Stockage avec Perl

- dans un premier temps : utilisez le module `Storable` (p.ex. `man Storable`) pour *sérialiser* le hash et le stocker sur le disque (voir `../exemples/stockage/demo-storable.pl`)
- dans un deuxième temps : liez le hash à un fichier indexé DBM (cf `perldoc -f tie` et `man DB_File` ; voir `../exemples/stockage/demo-tie.pl`)
- dans un troisième temps : consultez `/usr/share/doc/gppds/examples/` et effectuez la sauvegarde dans une base de données simple via le module `gppds`.

(je suis toujours à disposition)

Accès SGBD : DBI

- abstraction permettant l'accès à des bases de données orientées SQL
- exemples : `DBD::DBM`, `DBD::MySQL`, `DBD::Pg`
- à part les fonctionnalités avancées (transactions, etc) qui ne sont pas supportées par tous les pilotes (DBDs), simplifie la migration entre formats plats, systèmes de gestions de table et véritables SGBD
- relativement bas niveau : il existe de nombreuses bibliothèques d'abstraction de plus haut niveau (notamment notre `gppds`, spécifique à PostgreSQL)

Notre `gppds` est un module d'accès simplifié, il n'est pas forcément toujours recommandé. DBI est plus puissant et indépendant de la base de données. Il existe également de nombreux packages qui permettent des niveaux d'abstraction supérieurs.

Echange de données XML

- de plus en plus de programmes *échangent* (par opposition à stockent !) les données dans des formats XML plus ou moins bien spécifiés
- Perl propose de nombreux modules pour générer, ou parser du XML
- SAX, DOM et des méthodes non standards mais plus simples sont supportées.

Réutilisation de code

- modules Perl
- Perl orienté objet (OO)
- modules de CPAN (Comprehensive Perl Archive Network)
- packages Debian

Exercice : module

En partant de `../exemples/stockage/demo-storable.pl` ainsi que de `../exemples/modules` créez un module d'affichage (offrant les deux fonctions `affichage_*`), et utilisez-le.

Application : association de données

module Perl qui peut associer un fichier à un identificateur unique.

- idée : éviter le recours à des BLOBS de DB, ou carrément éviter une DB
- application : stockage de données de fax

Application : écriture de clients du module

But : écrire des outils utilisant notre module Perl

- logiciel de nettoyage : vérifie régulièrement la cohérence
- interfaçage à mgetty/réception de fax

Problème : client distant

Problème : comment implémenter un client distant à notre module ?

- une façon : par les Web services / SOAP Perl
- votre tâche : modifiez le client de réception de fax pour qu'il travaille à distance, et transformez notre module Perl en *Web Service*
- aidez-vous de `../exemples/soap`

Ecriture de GUI simples

But : créer une petite application d'affichage des données présentes

– on peut utiliser Perl : :Gtk ou Perl : :Tk (peu recommandé) pour implémenter facilement des GUI. Dans notre cas, pour gagner du temps nous allons utiliser un module Perl prévu pour.
– aidez-vous de `../exemples/gui`

De nouveau, notre façon de faire est simplifiée, Gtk est un environnement fort complexe et Perl peut accéder à l'entier de ce dernier.

Applications WWW

– 3 méthodes de conception (combinables)
  1. par template (p.ex. `HTML::Mason` ou `HTML::Template` (plus simple).
  2. par code dans HTML (méthode PHP). Est une extension brouillonne de la méthode des template. Le code n'est pas limité dans la portée de ses actions. Maintenance très difficile.
  3. par génération d'objets HTML (module CGI de Perl). S'apparente à de la programmation. S'adapte aux modifications du standard facilement. Difficile cependant de faire des présentations `jolies`

Applications WWW (2)

– 2 méthodes d'activation
  1. exécution du script Perl à chaque invocation (mode CGI). Avantage : suEXEC possible même en Apache 1.3 (sécurité). Inconvénient : performance.
  2. script Perl résident (mode `mod_perl`)

  (Apache 2 offre un intermédiaire entre ces deux méthodes, le concept de processus d'interprète Perl)

Applications WWW (3)

- 2 méthodes de gestion de données
  1. stupide : données passives, pas de contraintes d'intégrité, la concurrence et la cohérence sont assurées par le logiciel applicatif
  2. données structurées, actives, procédures stockées, views actives, transactions et cohérence gérée par la couche de données

Templates On crée l'interface WWW via des outils graphiques interactifs, puis on complète par du markup qui indique l'emplacement de variables, ou d'objets complexes, voire de constructions de générateurs (boucles, extraction de données similaire à XSLT).

Avantage : développement rapide, interface jolie.

Inconvénient : adaptation aux changements de standards totalement manuelle ou via l'outil de génération s'il peut reprendre le markup ajouté. Certains templates peuvent être générés par langage de programmation (p.ex. outil `wml`).

Méthode suggérée pour WWW

- petits scripts Perl WWW : utiliser exclusivement `CGI`
- application intégrée : bénéficier de `CGI::Application` (évite les `if` en cascade pour les *modes* de l'application)
- sessions et authentification : laisser Apache s'en charger
- performance : `mod_perl`
- beauté de l'interface : templates (laisser le design HTML aux professionnels)

Sessions Apache Voir par exemple
`http://www.perl.com/pub/a/2001/06/05/cgi.html`
et les modules `Apache::Auth:*`.

Ecriture d'une application WWW simple

But : créer une petite application de création de fax

- le module de base est CGI : il permet de créer facilement des objets HTML indépendamment de la version du standard, et de manière propre
- nous allons utiliser des modules simplifiés pour montrer ce qu'il est possible de faire.
- testez les exemples dans `../exemples/www`
- nos exemples comprennent un formulaire, un envoi de fichier, un peu de Javascript, illustrent la problématique du cross-scripting et certains utilisent le concept de sessions.

De nouveau, notre façon de faire est simplifiée.

Exemple d'application complexe : RT

présentation rapide ...

Conclusion

Perl est un langage général, qui a ses faiblesses, mais également ses forces dans beaucoup de domaines ! Attention aux trolls :)

Références

1-56592-149-6 Programming Perl. Larry WALL, Tom CHRISTIANSEN

1-56592-284-0 Learning Perl. Tom CHRISTIANSEN

`http://www.cpan.org` Comprehensive Perl Archive

`http://www.perl.com` Articles en ligne

Cours ESNIG Perl dans `others/`

Cours ESNIG SGBD dans `others/`

Questions

Merci de votre attention. Je suis à disposition pour vos questions.

`$Id: cours_perl.tex,v 1.5 2005/09/29 09:32:41 schaefer Exp $`