مقدمة في بيرل للمبتدئين

وثائق أعجوبة
http://docs.ojuba.org

رابط الوثيقة : http://www.ojuba.org/wiki/docs:perlintro
ساهم في تحرير هذه الوثيقة :
أحمد المحمودي - الصادق الشريدي - مؤيد السعدي - أشرف علي خلف - Eai Ahmed Mohamed - أسامة عقّاد

أول تحرير بواسطة أحمد المحمودي بتاريخ 2008/04/27 20:17
آخر تحرير بواسطة أشرف علي خلف بتاريخ 2008/11/21 01:59
تم تصدير هذه الوثيقة بتاريخ : 2010/01/02 09:01

تنويه : تمثل هذه الوثيقة تصديرا لنص على موقع أعجوبة، ولكن رغم ذلك
لا يتحمل الموقع أية مسئولية قانونية عن صحة أو خطا ما يرد فيها.

يسمح لك نسخ أو توزيع أو تعديل هذا المستند
وفق شروط الرخصة الحرة المحددة
حقوق النسخ محفوظة ©

جدول المحتويات

1. مقدمة في بيرل للمبتدئين.....4
1.1. الوصف.....4
1.2. ماهي بيرل؟.....4
1.3. تشغيل برامج بيرل.....4
1.4. عرض القواعد الأساسية.....4
1.5. أنواع المتغيرات (Variables) في بيرل.....4
1.5.1. Scalars.....4
1.5.2. المصفوفات.....4
1.5.3. Hashes.....4
1.6. Variable scoping.....4
1.7. Conditional and looping constructs.....4
1.8. Builtin operators and functions.....4
1.8.1. Arithmetic.....4
1.8.2. Numeric comparison.....4
1.8.3. String comparison.....4
1.8.4. المنطق البولي.....4
1.8.5. Miscellaneous.....4
1.9. Files and I/O.....4
1.10. Regular expressions.....4
1.10.1. Simple matching.....4
1.10.2. Simple substitution.....4
1.10.3. More complex regular expressions.....4
1.10.4. Parentheses for capturing.....4
1.10.5. Other regexp features.....4
1.11. Writing subroutines.....4
1.12. OO Perl.....4
1.13. Using Perl modules.....4
1.14. المؤلف.....4

~~ODT~~

1. مقدمة في بيرل للمبتدئين

1.1. الوصف

هذه الوثيقة تهدف إلى إعطائك لمحة سريعة عن لغة البرمجة بيرل، إلى جانب مؤشرات لمزيد من الوثائق.
وهي بمثابة نقطة انطلاق ودليل لأولئك المستخدمين الجدد للغة ، ويقدم معلومات كافية فقط لك لتكون قادرا على أن تقرأوتفهم- تقريبا- ما يقوم به المستخدمون الآخرون لبيرل، أو كتابة سكربتات بسيطة خاصة بك.
هذه الوثيقة التمهيدية لا تهدف إلى الكمال. ولا تهدف كذلك إلى أن تكون دقيقة بمعنى الكلمة. ففي بعض الحالات قد يتم التضحية بالكمال بغرض الحصول على فكرة عامة من خلال ذلك. وننصحك نصيحة خالصة بمتابعة هذه المقدمة مع مزيد من المعلومات من الدليل الكامل لبيرل ، وقائمة المحتويات التي يمكن العثور عليها في perltoc.

وطوال هذه الوثيقة سترى إشارات إلى أجزاء أخرى من وثائق بيرل. ويمكنك أن تقرأ تلك الوثائق باستخدام الأمر perldoc أو بأي طريقة تستخدمها لقراءة هذه الوثيقة.

1.2. ماهي بيرل؟

بيرل هي لغة برمجة عامة الأغراض و طورت أساسا للتحكم (التلاعب) manipulation في النصوص و الآن تستخدم لعديد من المهام من ضمنها إدارة الأنظمة و تطوير الويب و برمجة الشبكات و تطوير واجهة المستخدم الرسومية GUI و غير ذلك.

هذه اللغة يقصد بها أن تكون تطبيقية (سهلة الاستعمال, فعّالة, كاملة(اضافة للجمال (الصغر, الأناقة , الحد الأدنى). خاصتها الرئيسية أنها سهلة الاستعال, تدعم كلا من البرمجة بالاجراء والبرمجة المتعلقة بالكائنات, فيها دعم مبني بها لمعالجة النصوص, و لها احدى اكثر مجموعات الوحدات Modules روعةً في العالم ,و المقدمة من طرف ثالث third-party. The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). Its major features are that it's easy to use, supports both procedural and object-oriented (OO) programming, has powerful built-in support for text processing, and has one of the world's most impressive collections of third-party modules.

التعاريف المختلفة لبيرل موجود في perl و perlfaq1, وبلا شك في أماكن اخرى. من هذا يمكننا أن نقرر أن بيرل تختلف من شخص لآخر, لكن الكثير من الناس يظنون أنها على الأقل تستحق الكتابة عنها.

Different definitions of Perl are given in perl, perlfaq1 and no doubt other places. From this we can determine that Perl is different things to different people, but that lots of people think it's at least worth writing about.

1.3. تشغيل برامج بيرل

لتشغيل برامج بيرل من سطر أوامر يونكس:

```
perl progname.pl
```

أيضاً يمكن وضع الآتي في أول سطر من البرنامج:

```
#!/usr/bin/env perl
```

... و من ثم تشغيل البرنامج بـ path/to/script.pl. بالطبع يحتاج البرنامج قبل هذا أن يكون تنفيذيا (executable)، أي chmod 755 script.pl (في يونكس).

لمزيدٍ من المعلومات, و للإرشادات لمنصات أخرى مثل Windows و Mac OS، اقرأ دليل perlrun.

1.4. عرض القواعد الأساسية

برنامج أو سكريبت بيرل يتكون من جملة واحدة أو أكثر.

تتألف لغة البيرل او الأسكربت من واحد الي عدة بيانات. هذه البيانات ببساطة مكتوبة في شكل صريح. ليس هناك حاجة لوجود () او وظيفة او اي شيء من هذا القبيل.

جمل بيرل تنتهي بفاصلة منقوطة semi-colon.

```
print "Hello, world";
```

التعليقات تبدأ بعلامة مربع و تستمر إلى نهاية السطر

```
# This is a comment
```

المسافات البيضاء غير معتبرة:

```
print
    "Hello, world"
    ;
```

... إلا ما دخل بين حاصرتين:

```
# سيطبع هذا مع نزول في السطر عند المنتصف
print "Hello
world";
```

الحاصرة المفردة أو المزدوجة يمكن استخدامها حول النص الحرفي:

```
print "Hello, world";
print Hello, world;
```

على كل, فقط الحاصرات المزدوجة ستظهر المتغيرات و المحارف الخاصة مثل سطر-جديد (\ n):

```
print "Hello, $name\n";      # تعمل جيدا
print Hello, $name\n;      # تطبع $name\n حرفيا
```

الأرقام لا تحاط بعلامتي اقتباس:

```
print 42;
```

يمكنك استخدام الأقواس للوظائف' المعطيات أو أضفهم تبعا لمزاجك الخاص. يُحاج لها أحيانا لتوضيح مسائل الاسبقية.

```
print("Hello, world\n");
print "Hello, world\n";
```

يمكنك إيجاد معلومات أكثر تفصيلا عن قواعد الصياغة syntax في بيرل في perlsyn.

1.5. أنواع المتغيرات (Variables) في بيرل

لبيرل ثلاث أنواع رئيسية من المتغيرات: scalars, arrays, and hashes.

1.5.1. Scalars

scalar يمثل قيمة مفردة :

```
    my $animal = "camel";
```

```
my $answer = 42;
```

قيم Scalar قد تكون نصوصا, أعدادا صحيحة أو فاصلة-عائمة, و بيرل ستحول تلقائيا بينها عند الطلب. ليس هناك حاجة لإعلان-مسبق لأنواع متغيراتك. قيم Scalar يمكن استخدامها بطرق متعددة:

```
print $animal;
print "The animal is $animal\n";
print "The square of $answer is ", $answer * $answer, "\n";
```

هناك عدد من Scalar سحرية باسماء تبدو مثل علامات ترقيم او ضوضاء السطر. هذه المتغيرات تستخدم لكل الأغراض, و موثقة في perlvar. الوحيد الذي عليك ان تعرفه الآن هو $_ و هو المتغير الافتراضي. يستخدم كمعطى افتراضي لعدد من الوظائف في بيرل, و تضمّن مجموعته set ببنى حلقية محددة. Fix Me! it's set implicitly by certain looping constructs

```
print;          # prints contents of $_ by default
```

1.5.2. المصفوفات

المصفوفات تمثل قائمة من القيم:

```
my @animals = ("camel", "llama", "owl");
my @numbers = (23, 42, 69);
my @mixed   = ("camel", 42, 1.23);
```

المصفوفات مفهرسة بدءا من الصفر. وها هنا كيفية حصولك على العناصر في مصفوفة :

```
print $animals[0];              # prints "camel"
print $animals[1];              # prints "llama"
```

المتغير الخاص array#$ يخبرك بفهرس العنصر الأخير من المصفوفة:

```
print $mixed[$#mixed];          # last element, prints 1.23
```

ولعلك يتم إغراؤك لاستخدام array + 1#$ لتخبرك بعدد العناصر الموجودة في مصفوفة . لا تنزعج ؛حيث إن ذلك يحدث، استخدام@array حيث تتوقع بيرل إيجاد قيمة scalar (في السياق الخاص ب scalar)سوف تعطيك عدد العناصر في المصفوفة :

```
if (@animals < 5) { ... }
```

العناصر التي حصلنا عليها من المصفوفة تبدأ بـ $ لأننا حصلنا فقط على قيمة فردية ناتجة من المصفوفة. قد سألت عن scalar وها أنت قد حصلت على scalar. للحصول على قيم متعددة من مصفوفة :

```
@animals[0,1];                  # gives ("camel", "llama");
@animals[0..2];                 # gives ("camel", "llama", "owl");
@animals[1..$#animals];         # gives all except the first element
```

وهذا ما يدعى بـ "array slice". يمكنك عمل عدة أشياء مفيدة للقوائم:

```
my @sorted    = sort @animals;
my @backwards = reverse @numbers;
```

هناك أيضا حزمة خاصة من المصفوفات، مثل @ARGV (قيم سطر أوامر في السكربت الخاص بك)، و @_ (القيم الممررة إلى الروتين الفرعي subroutine). وقد تم توثيق ذلك في perlvar

1.5.3. Hashes

A hash represents a set of key/value pairs:

```
my %fruit_color = ("apple", "red", "banana", "yellow");
```

You can use whitespace and the ⇒ operator to lay them out more nicely:

```
my %fruit_color = (
    apple  => "red",
    banana => "yellow",
```

```
    );
```

To get at hash elements:

```
    $fruit_color{"apple"};          # gives "red"
```

You can get at lists of keys and values with keys() and values().

```
    my @fruits = keys %fruit_colors;
    my @colors = values %fruit_colors;
```

Hashes have no particular internal order, though you can sort the keys and loop through them. Just like special scalars and arrays, there are also special hashes. The most well known of these is %ENV which contains environment variables. Read all about it (and other special variables) in perlvar.

Scalars, arrays and hashes are documented more fully in perldata.

More complex data types can be constructed using references, which allow you to build lists and hashes within lists and hashes.

A reference is a scalar value and can refer to any other Perl data type. So by storing a reference as the value of an array or hash element, you can easily create lists and hashes within lists and hashes. The following example shows a 2 level hash of hash structure using anonymous hash references.

```
my $variables = {
    scalar  =>  {
                 description => "single item",
                 sigil  => $,
                },
    array   =>  {
                 description => "ordered list of items",
                 sigil  => @,
                },
    hash    =>  {
                 description => "key/value pairs",
                 sigil  => %,
                },
};
print "Scalars begin with a $variables->{scalar}->{sigil}\n";
```

Exhaustive information on the topic of references can be found in perlreftut, perllol, perlref and perldsc.

1.6. Variable scoping

في الأقسام السابقة، أستخدمت قواعد الصياغة syntax في كل الأمثلة.

```
my $var = "value";
```

The my is actually not required; you could just use:

```
$var = "value";
```

However, the above usage will create global variables throughout your program, which is bad programming practice. my creates lexically scoped variables instead. The variables are scoped to the block (i.e. a bunch of statements surrounded by curly-braces) in which they are defined.

```
my $a = "foo";
if ($some_condition) {
```

```
    my $b = "bar";
    print $a;           # prints "foo"
    print $b;           # prints "bar"
}
print $a;               # prints "foo"
print $b;               # prints nothing; $b has fallen out of scope
```

Using my in combination with a use strict; at the top of your Perl scripts means that the interpreter will pick up certain common programming errors. For instance, in the example above, the final print $b would cause a compile-time error and prevent you from running the program. Using strict is highly recommended.

1.7. Conditional and looping constructs

Perl has most of the usual conditional and looping constructs except for case/switch (but if you really want it, there is a Switch module in Perl 5.8 and newer, and on CPAN. See the section on modules, below, for more information about modules and CPAN).

The conditions can be any Perl expression. See the list of operators in the next section for information on comparison and boolean logic operators, which are commonly used in conditional statements.

if

```
    if ( condition ) {
        ...
    } elsif ( other condition ) {
        ...
    } else {
        ...
    }
There's also a negated version of it:
    unless ( condition ) {
        ...
    }
This is provided as a more readable version of if (!CIcondition).
Note that the braces are required in Perl, even if you've only got one
line in the block.  However, there is a clever way of making your one-line
conditional blocks more English like:
    # the traditional way
    if ($zippy) {
        print "Yow!";
    }
    # the Perlish post-condition way
    print "Yow!" if $zippy;
    print "We have no bananas" unless $bananas;
```

while

```
    while ( condition ) {
        ...
    }
There's also a negated version, for the same reason we have unless:
    until ( condition ) {
        ...
    }
```

```
You can also use while in a post-condition:
    print "LA LA LA\n" while 1;          # loops forever
```

for

```
Exactly like C:
    for ($i=0; $i <= $max; $i++) {
        ...
    }
The C style for loop is rarely needed in Perl since Perl provides
the more friendly list scanning foreach loop.
```

foreach

```
    foreach (@array) {
        print "This element is $_\n";
    }
    # you dont have to use the default $_ either...
    foreach my $key (keys %hash) {
        print "The value of $key is $hash{$key}\n";
    }
```

For more detail on looping constructs (and some that weren't mentioned in this overview) see perlsyn.

1.8. Builtin operators and functions

Perl comes with a wide selection of builtin functions. Some of the ones we've already seen include print, sort and reverse. A list of them is given at the start of perlfunc and you can easily read about any given function by using perldoc -f CIfunctionname.

Perl operators are documented in full in perlop, but here are a few of the most common ones:

1.8.1. Arithmetic

```
    +    جمع
    -    طرح
    *    ضرب
    /    قسمة
```

1.8.2. Numeric comparison

```
    ==   equality
    !=   inequality
    <    less than
    >    greater than
    <=   less than or equal
    >=   greater than or equal
```

1.8.3. String comparison

```
    eq   equality
    ne   inequality
    lt   less than
    gt   greater than
    le   less than or equal
    ge   greater than or equal
```

```
(Why do we have separate numeric and string comparisons? Because we don't
have special variable types, and Perl needs to know whether to sort
numerically (where 99 is less than 100) or alphabetically (where 100 comes
before 99).
```

1.8.4. المنطق البولي

```
    &&  and
    ||  or
    !   not
(and, or and not aren't just in the above table as descriptions
of the operators they're also supported as operators in their own
right. They're more readable than the C-style operators, but have
different precedence to && and friends. Check perlop for more
detail.)
```

1.8.5 .Miscellaneous

```
    =   assignment
    .   string concatenation
    x   string multiplication
    ..  range operator (creates a list of numbers)
```

:Many operators can be combined with a = as follows

```
$a += 1;        # same as $a = $a + 1
$a -= 1;        # same as $a = $a - 1
$a .= "\n";     # same as $a = $a . "\n";
```

1.9. Files and I/O

You can open a file for input or output using the open() function. It's :documented in extravagant detail in perlfunc and perlopentut, but in short

```
open(INFILE,  "input.txt")    or die "Cant open input.txt:  $!";
open(OUTFILE, ">output.txt")  or die "Cant open output.txt:  $!";
open(LOGFILE, ">>my.log")     or die "Cant open logfile:  $!";
```

You can read from an open filehandle using the <> operator. In scalar context it reads a single line from the filehandle, and in list context it reads the whole file :in, assigning each line to an element of the list

```
my $line  = <INFILE>;
my @lines = <INFILE>;
```

Reading in the whole file at one time is called slurping. It can be useful but it may be a memory hog. Most text file processing can be done a line at a time .with Perl's looping constructs

:The <> operator is most often seen in a while loop

```
while (<INFILE>) {      # assigns each line in turn to $_
    print "Just read in this line: $_";
}
```

We've already seen how to print to standard output using print(). However, print() can also take an optional first argument specifying which filehandle to :print to

```
print STDERR "This is your final warning.\n";
print OUTFILE $record;
```

```
print LOGFILE $logmessage;
```

When you're done with your filehandles, you should close() them (though to be honest, Perl will clean up after you if you forget):

```
close INFILE;
```

1.10. Regular expressions

Perl's regular expression support is both broad and deep, and is the subject of lengthy documentation in perlrequick, perlretut, and elsewhere. However, in short:

1.10.1. Simple matching

```
    if (/foo/)        { ... }  # true if $_ contains "foo"
    if ($a =~ /foo/)  { ... }  # true if $a contains "foo"
The // matching operator is documented in perlop.  It operates on
$_ by default, or can be bound to another variable using the =~
binding operator (also documented in perlop).
```

1.10.2. Simple substitution

```
    s/foo/bar/;                # replaces foo with bar in $_
    $a =~ s/foo/bar/;          # replaces foo with bar in $a
    $a =~ s/foo/bar/g;         # replaces ALL INSTANCES of foo with bar in $a
The s/// substitution operator is documented in perlop.
```

1.10.3. More complex regular expressions

```
You don't just have to match on fixed strings.  In fact, you can match
on just about anything you could dream of by using more complex regular
expressions.  These are documented at great length in perlre, but for
the meantime, here's a quick cheat sheet:
    .                   a single character
    \s                  a whitespace character (space, tab, newline)
    \S                  non-whitespace character
    \d                  a digit (0-9)
    \D                  a non-digit
    \w                  a word character (a-z, A-Z, 0-9, _)
    \W                  a non-word character
    [aeiou]             matches a single character in the given set
    [^aeiou]            matches a single character outside the given set
    (foo|bar|baz)       matches any of the alternatives specified
    ^                   start of string
    $                   end of string
Quantifiers can be used to specify how many of the previous thing you
want to match on, where thing means either a literal character, one
of the metacharacters listed above, or a group of characters or
metacharacters in parentheses.
```

- * zero or more of the previous thing
- + one or more of the previous thing

```
    ?                   zero or one of the previous thing
    {3}                 matches exactly 3 of the previous thing
    {3,6}               matches between 3 and 6 of the previous thing
    {3,}                matches 3 or more of the previous thing
```

```
Some brief examples:
    /^\d+/                  string starts with one or more digits
    /^$/                    nothing in the string (start and end are adjacent)
    /(\d\s){3}/             a three digits, each followed by a whitespace
                            character (eg "3 4 5 ")
    /(a.)+/                 matches a string in which every odd-numbered letter
                            is a (eg "abacadaf")
    # This loop reads from STDIN, and prints non-blank lines:
    while (<>) {
        next if /^$/;
        print;
    }
```

Parentheses for capturing .1.10.4

```
As well as grouping, parentheses serve a second purpose.  They can be
used to capture the results of parts of the regexp match for later use.
The results end up in $1, $2 and so on.
    # a cheap and nasty way to break an email address up into parts
    if ($email =~ /([^@]+)@(.+)/) {
        print "Username is $1\n";
        print "Hostname is $2\n";
    }
```

Other regexp features .1.10.5

```
Perl regexps also support backreferences, lookaheads, and all kinds of
other complex details.  Read all about them in perlrequick,
perlretut, and perlre.
```

Writing subroutines .1.11

:Writing subroutines is easy

```
sub log {
    my $logmessage = shift;
    print LOGFILE $logmessage;
}
```

What's that shift? Well, the arguments to a subroutine are available to us as a special array called @_ (see perlvar for more on that). The default argument to the shift function just happens to be @_. So my $logmessage = shift; shifts the .first item off the list of arguments and assigns it to $logmessage

:We can manipulate @_ in other ways too

```
my ($logmessage, $priority) = @_;        # common
my $logmessage = $_[0];                  # uncommon, and ugly
```

:Subroutines can also return values

```
sub square {
    my $num = shift;
    my $result = $num * $num;
    return $result;
}
```

.For more information on writing subroutines, see perlsub

1.12. OO Perl

OO Perl is relatively simple and is implemented using references which know what sort of object they are based on Perl's concept of packages. However, OO Perl is largely beyond the scope of this document. Read perlboot, perltoot, perltooc and perlobj.

As a beginning Perl programmer, your most common use of OO Perl will be in using third-party modules, which are documented below.

1.13. Using Perl modules

Perl modules provide a range of features to help you avoid reinventing the wheel, and can be downloaded from CPAN (http://www.cpan.org/). A number of popular modules are included with the Perl distribution itself.

Categories of modules range from text manipulation to network protocols to database integration to graphics. A categorized list of modules is also available from CPAN.

To learn how to install modules you download from CPAN, read perlmodinstall

To learn how to use a particular module, use perldoc CIModule::Name. Typically you will want to use CIModule::Name, which will then give you access to exported functions or an OO interface to the module.

perlfaq contains questions and answers related to many common tasks, and often provides suggestions for good CPAN modules to use.

perlmod describes Perl modules in general. perlmodlib lists the modules which came with your Perl installation.

If you feel the urge to write Perl modules, perlnewmod will give you good advice.

1.14. المؤلف

Kirrily Skud Robert skud@cpan.org