Transforming a Perl array using map

map transform list array

Prev Next

The `map` function of Perl provides a simple way to transform a list of values to another list of values. Usually this is a one-to-one transformation but in general the resulting list can be also shorter or longer than the original list.

We saw that grep of Perl is a generalization of the UNIX grep command. It selects some or all of the elements from the original list, and returns them unchanged.

`map`, on the other hand, is used when you would like to make changes to the values of the original list.

The syntax is similar. You provide a block of code and a list of values: an array or some other expression that returns a list of values. For every value in the original list, the value is placed in `$_`, the default variable of Perl, and the block is executed. The resulting values are passed on, to the left-hand side of the assignment.

map for simple transformation

```
1. my @numbers = (1..5);
   print "@numbers\n";       # 1 2 3 4 5
   my @doubles = map {$_ * 2} @numbers;
   print "@doubles\n";       # 2 4 6 8 10
```

Building fast look-up table

Sometimes we have a list of values and during the execution of the code we need to check if a given value is within this list. We can usegrep every time, to check if the current value is in the list. We can even use the any function from List::MoreUtils, but it can be much more readable and faster if we used a hash for lookup.

We need to create a hash once, where the keys are the elements of the array, and the values of the hash are all 1s. Then, a simple hash lookup can replace the `grep`.

```
1. use Data::Dumper qw(Dumper);
```

```
my @names = qw(Foo Bar Baz);
my %is_invited = map {$_ => 1} @names;

my $visitor = <STDIN>;
chomp $visitor;

if ($is_invited{$visitor}) {
   print "The visitor $visitor was invited\n";
}

print Dumper \%is_invited;
```

This is the output of the `Dumper` call:

```
$VAR1 = {
            'Bar' => 1,
            'Baz' => 1,
            'Foo' => 1
          };
```

In this code we don't really care about the values of the hash elements, other than the fact that they should evaluate to true in Perl.

This solution is only interesting if you do repeated look-ups on a large set of values. (The exact meaning of "large" might depend on the system.) Otherwise `any` or even `grep` will do it.

As you can see, in this example, for every element in the original array, `map` returned two values. The original value and 1.

```
my @names = qw(Foo Bar Baz);
my @invited = map {$_ => 1} @names;
print "@invited\n"
```

will print:

```
Foo 1 Bar 1 Baz 1
```

Fat arrow

In case you are wondering `=>` is called the **fat arrow** or **fat comma**. It basically acts as a regular comma `,` with an exception that is not relevant in our case. (There is a description of it in the article about Perl hashes.)

Complex expressions in map

You can put more complex expressions with map:

```
1. my @names = qw(Foo Bar Baz);
   my @invited = map { $_ =~ /^F/ ? ($_ => 1) : () } @names;
   print "@invited\n"
```

Will print

```
Foo 1
```

Inside the block we have a ternary operator that returns either a pair like earlier or an empty list. Apparently we only want to let people in if their name starts with an F.

```
1. $_ =~ /^F/ ? ($_ => 1) : ()
```

Schwartzian transform

One of the many uses of `map` is in the Schwartzian transformimproving the speed of `sort` in certain cases.

perldoc

For further explanation with a couple of strange cases, check out perldoc -f map.