1 ModPerl::MethodLookup -- Map mod_perl 2.0 modules, objects and methods

1 ModPerl::MethodLookup -- Map mod_perl 2.0 modules, objects and methods

1.1 Synopsis

```
use ModPerl::MethodLookup;

# return all module names containing XS method 'print'
my($hint, @modules) =
    ModPerl::MethodLookup::lookup_method('print');

# return only module names containing method 'print' which
# expects the first argument to be of type 'Apache::Filter'
# (here $filter is an Apache::Filter object)
my($hint, @modules) =
    ModPerl::MethodLookup::lookup_method('print', $filter);
# or
my($hint, @modules) =
    ModPerl::MethodLookup::lookup_method('print', 'Apache::Filter');

# what XS methods defined by module 'Apache::Filter'
my($hint, @methods) =
    ModPerl::MethodLookup::lookup_module('Apache::Filter');

# what XS methods can be invoked on the object $r (or a ref)
my($hint, @methods) =
    ModPerl::MethodLookup::lookup_object($r);
# or
my($hint, @methods) =
    ModPerl::MethodLookup::lookup_object('Apache::RequestRec');

# preload all mp2 modules in startup.pl
ModPerl::MethodLookup::preload_all_modules();

# command line shortcuts
% perl -MApache2 -MModPerl::MethodLookup -e print_module \
  Apache::RequestRec Apache::Filter
% perl -MApache2 -MModPerl::MethodLookup -e print_object Apache
% perl -MApache2 -MModPerl::MethodLookup -e print_method \
  get_server_built request
% perl -MApache2 -MModPerl::MethodLookup -e print_method read
% perl -MApache2 -MModPerl::MethodLookup -e print_method read APR::Bucket
```

1.2 Description

mod_perl 2.0 provides many methods, which reside in various modules. One has to load each of the modules before using the desired methods. `ModPerl::MethodLookup` provides the Perl API for finding module names which contain methods in question and other helper functions, like figuring out what methods defined by some module, or what methods can be called on a given object.

1.3 API

1.3.1 lookup_method()

```
my($hint, @modules) =
    ModPerl::MethodLookup::lookup_method($method_name);
```

The `lookup_method()` function accepts the method name as the first argument.

The first returned value is a string returning a human readable lookup result. Normally suggesting which modules should be loaded, ready for copy-n-paste or explaining the failure if the lookup didn't succeed.

The second returned value is an array of modules which have matched the query, i.e. the names of the modules which contain the requested method.

```
my($hint, @modules) =
    ModPerl::MethodLookup::lookup_method($method_name, $object);
```

or:

```
my($hint, @modules) =
    ModPerl::MethodLookup::lookup_method($method_name, ref($object));
```

The `lookup_method()` function accepts a second optional argument, which is a blessed object or the class it's blessed into. If there is more than one matches this extra information is used to return only modules of those methods which operate on the objects of the same kind. This usage is useful when the `AUTOLOAD` is used to find yet-unloaded modules which include called methods.

Examples:

Return all module names containing XS method *print*:

```
my($hint, @modules) =
    ModPerl::MethodLookup::lookup_method('print');
```

Return only module names containing method *print* which expects the first argument to be of type `Apache::Filter`:

```
my $filter = bless {}, 'Apache::Filter';
my($hint, @modules) =
    ModPerl::MethodLookup::lookup_method('print', $filter);
```

or:

```
my($hint, @modules) =
    ModPerl::MethodLookup::lookup_method('print', 'Apache::Filter');
```

1.3.2 lookup_module()

```
my($hint, @methods) =
    ModPerl::MethodLookup::lookup_module($module_name);
```

The `lookup_module()` function accepts the module name as an argument.

The first returned value is a string returning a human readable lookup result. Normally suggesting, which methods the given module implements, or explaining the failure if the lookup didn't succeed.

The second returned value is an array of methods which have matched the query, i.e. the names of the methods defined in the requested module.

Example:

What XS methods defined by module `Apache::Filter`:

```
my($hint, @methods) =
    ModPerl::MethodLookup::lookup_module('Apache::Filter');
```

1.3.3 lookup_object()

```
my($hint, @methods) =
    ModPerl::MethodLookup::lookup_object($object);
```

or:

```
my($hint, @methods) =
    ModPerl::MethodLookup::lookup_object(
        $the_class_object_is_blessed_into);
```

The `lookup_object()` function accepts the object or the class name the object is blessed into as an argument.

The first returned value is a string returning a human readable lookup result. Normally suggesting, which methods the given object can invoke (including module names that need to be loaded to use those methods), or explaining the failure if the lookup didn't succeed.

The second returned value is an array of methods which have matched the query, i.e. the names of the methods that can be invoked on the given object (or its class name).

META: As of this writing this method may miss some of the functions/methods that can be invoked on the given object. Currently we can't programmatically deduct the objects they are invoked on, because these methods are written in pure XS and manipulate the arguments stack themselves. Currently these are mainly XS functions, not methods, which of course aren't invoked on objects. There are also logging function wrappers (`Apache::Log`).

Examples:

What XS methods can be invoked on the object `$r`:

```
my($hint, @methods) =
    ModPerl::MethodLookup::lookup_object($r);
```

or `$r`'s class -- `Apache::RequestRec`:

```
my($hint, @methods) =
    ModPerl::MethodLookup::lookup_object('Apache::RequestRec');
```

1.3.4 print_method()

`print_method()` is a convenience wrapper for `lookup_method()`, mainly designed to be used from the command line. For example to print all the modules which define method *read* execute:

```
% perl -MApache2 -MModPerl::MethodLookup -e print_method read
```

Since this will return more than one module, we can narrow the query to only those methods which expect the first argument to be blessed into class `APR::Bucket`:

```
% perl -MApache2 -MModPerl::MethodLookup -e print_method read APR::Bucket
```

You can pass more than one method and it'll perform a lookup on each of the methods. For example to lookup methods `get_server_built` and `request` you can do:

```
% perl -MApache2 -MModPerl::MethodLookup -e print_method \
  get_server_built request
```

The function `print_method()` is exported by default.

1.3.5 print_module()

`print_module()` is a convenience wrapper for `lookup_module()`, mainly designed to be used from the command line. For example to print all the methods defined in the module `Apache::RequestRec`, followed by methods defined in the module `Apache::Filter` you can run:

```
% perl -MApache2 -MModPerl::MethodLookup -e print_module \
  Apache::RequestRec Apache::Filter
```

The function `print_module()` is exported by default.

1.3.6 print_object()

`print_object()` is a convenience wrapper for `lookup_object()`, mainly designed to be used from the command line. For example to print all the methods that can be invoked on object blessed into a class `Apache::RequestRec` run:

```
% perl -MApache2 -MModPerl::MethodLookup -e print_object \
  Apache::RequestRec
```

Similar to `print_object()`, more than one class can be passed to this function.

The function `print_object()` is exported by default.

1.3.7 `preload_all_modules()`

The function `preload_all_modules()` preloads all mod_perl 2.0 modules, which implement their API in XS. This is similar to the mod_perl 1.0 behavior which has most of its methods loaded at the startup.

CPAN modules developers should make sure their distribution loads each of the used mod_perl 2.0 modules explicitly, and not use this function, as it takes the fine control away from the users. One should avoid doing this the production server (unless all modules are used indeed) in order to save memory.

1.4 Applications

1.4.1 `AUTOLOAD`

When Perl fails to locate a method it checks whether the package the object belongs to has an `AUTOLOAD` function defined and if so, calls it with the same arguments as the missing method while setting a global variable `$AUTOLOAD` (in that package) to the name of the originally called method. We can use this facility to lookup the modules to be loaded when such a failure occurs. Though since we have many packages to take care of we will use a special `UNIVERSAL::AUTOLOAD` function which Perl calls if can't find the `AUTOLOAD` function in the given package.

In that function you can query `ModPerl::MethodLookup`, require() the module that includes the called method and call that method again using the goto() trick:

```
use ModPerl::MethodLookup;
sub UNIVERSAL::AUTOLOAD {
    my($hint, @modules) =
        ModPerl::MethodLookup::lookup_method($UNIVERSAL::AUTOLOAD, @_);
    if (@modules) {
        eval "require $_" for @modules;
        goto &$UNIVERSAL::AUTOLOAD;
    }
    else {
        die $hint;
    }
}
```

However we don't endorse this approach. It's a better approach to always abort the execution which printing the `$hint`and use fix the code to load the missing module. Moreover installing `UNIVERSAL::AUTOLOAD` may cause a lot of problems, since once it's installed Perl will call it every time some method is missing (e.g. undefined `DESTROY` methods). The following approach seems to somewhat work for me. It installs `UNIVERSAL::AUTOLOAD` only when the the child process starts.

```
httpd.conf:
-----------
PerlChildInitHandler ModPerl::MethodLookupAuto

startup.pl:
-----------
{
  package ModPerl::MethodLookupAuto;
  use ModPerl::MethodLookup;

  use Carp;
  sub handler {

      # exclude DESTROY resolving
      my $skip = '^(?!DESTROY$';
      *UNIVERSAL::AUTOLOAD = sub {
          my $method = $AUTOLOAD;
          return if $method =~ /DESTROY/;
          my ($hint, @modules) =
              ModPerl::MethodLookup::lookup_method($method, @_);
          $hint ||= "Can't find method $AUTOLOAD";
          croak $hint;
      };
      return 0;
  }
}
```

This example doesn't load the modules for you. It'll print to STDERR what module should be loaded, when a method from the not-yet-loaded module is called.

A similar technique is used by `Apache::porting`.

1.4.2 Command Line Lookups

When a method is used and mod_perl has reported a failure to find it, it's often useful to use the command line query to figure out which module needs to be loaded. For example if when executing:

```
$r->construct_url();
```

mod_perl complains:

```
Can't locate object method "construct_url" via package
"Apache::RequestRec" at ...
```

you can ask `ModPerl::MethodLookup` for help:

```
% perl -MApache2 -MModPerl::MethodLookup -e print_method construct_url
To use method 'construct_url' add:
        use Apache::URI ();
```

and after copy-n-pasting the use statement in our code, the problem goes away.

One can create a handy alias for this technique. For example, C-style shell users can do:

```
% alias lookup "perl -MApache2 -MModPerl::MethodLookup -e print_method"
```

For Bash-style shell users:

```
% alias lookup="perl -MApache2 -MModPerl::MethodLookup -e print_method"
```

Now the lookup is even easier:

```
% lookup construct_url
to use method 'construct_url' add:
        use Apache::URI;
```

Similar aliases can be provided for `print_object()` and `print_module()`.

1.5 Todo

These methods aren't yet picked by this module (the extract from the map file):

```
modperl_filter_attributes     | MODIFY_CODE_ATTRIBUTES
modperl_spawn_proc_prog       | spawn_proc_prog
apr_sockaddr_ip_get           | sockaddr
apr_sockaddr_port_get         | sockaddr
apr_ipsubnet_create           | new
```

Please report if you find any other missing methods. But remember that as of this moment the module reports only XS function. In the future we may add support for pure perl functions/methods as well.

1.6 See Also

- the mod_perl 1.0 backward compatibility document
- porting Perl modules
- porting XS modules
- `Apache::porting`

1.7 Author

Stas Bekman

Table of Contents: