NAME

Module::Runtime − runtime module handling

SYNOPSIS

```
use Module::Runtime qw(
        $module_name_rx is_module_name check_module_name
        module_notional_filename require_module
);

if($module_name =~ /\A$module_name_rx\z/o) { ...
if(is_module_name($module_name)) { ...
check_module_name($module_name);

$notional_filename = module_notional_filename($module_name);
require_module($module_name);

use Module::Runtime qw(use_module use_package_optimistically);

$bi = use_module("Math::BigInt", 1.31)->new("1_234");
$widget = use_package_optimistically("Local::Widget")->new;

use Module::Runtime qw(
        $top_module_spec_rx $sub_module_spec_rx
        is_module_spec check_module_spec
        compose_module_name
);

if($spec =~ /\A$top_module_spec_rx\z/o) { ...
if($spec =~ /\A$sub_module_spec_rx\z/o) { ...
if(is_module_spec("Standard::Prefix", $spec)) { ...
check_module_spec("Standard::Prefix", $spec);

$module_name =
        compose_module_name("Standard::Prefix", $spec);
```

DESCRIPTION

The functions exported by this module deal with runtime handling of Perl modules, which are normally handled at compile time. This module avoids using any other modules, so that it can be used in low-level infrastructure.

The parts of this module that work with module names apply the same syntax that is used for barewords in Perl source. In principle this syntax can vary between versions of Perl, and this module applies the syntax of the Perl on which it is running. In practice the usable syntax hasn't changed yet. There's some intent for Unicode module names to be supported in the future, but this hasn't yet amounted to any consistent facility.

The functions of this module whose purpose is to load modules include workarounds for three old Perl core bugs regarding `require`. These workarounds are applied on any Perl version where the bugs exist, except for a case where one of the bugs cannot be adequately worked around in pure Perl.

Module name syntax

The usable module name syntax has not changed from Perl 5.000 up to Perl 5.19.8. The syntax is composed entirely of ASCII characters. From Perl 5.6 onwards there has been some attempt to allow the use of non-ASCII Unicode characters in Perl source, but it was fundamentally broken (like the entirety of Perl 5.6's Unicode handling) and remained pretty much entirely unusable until it got some attention in the Perl 5.15 series. Although Unicode is now consistently accepted by the parser in some places, it remains broken for module names. Furthermore, there has not yet been any work on how to map Unicode module names into filenames, so in that respect also Unicode module names are unusable.

The module name syntax is, precisely: the string must consist of one or more segments separated by `::`; each segment must consist of one or more identifier characters (ASCII alphanumerics plus "_");

the first character of the string must not be a digit. Thus "`IO::File`", "`warnings`", and "`foo::123::x_0`" are all valid module names, whereas "`IO::`" and "`1foo::bar`" are not. `'` separators are not permitted by this module, though they remain usable in Perl source, being translated to `::` in the parser.

Core bugs worked around

The first bug worked around is core bug [perl #68590], which causes lexical state in one file to leak into another that is `require`/`use`d from it. This bug is present from Perl 5.6 up to Perl 5.10, and is fixed in Perl 5.11.0. From Perl 5.9.4 up to Perl 5.10.0 no satisfactory workaround is possible in pure Perl. The workaround means that modules loaded via this module don't suffer this pollution of their lexical state. Modules loaded in other ways, or via this module on the Perl versions where the pure Perl workaround is impossible, remain vulnerable. The module Lexical::SealRequireHints provides a complete workaround for this bug.

The second bug worked around causes some kinds of failure in module loading, principally compilation errors in the loaded module, to be recorded in `%INC` as if they were successful, so later attempts to load the same module immediately indicate success. This bug is present up to Perl 5.8.9, and is fixed in Perl 5.9.0. The workaround means that a compilation error in a module loaded via this module won't be cached as a success. Modules loaded in other ways remain liable to produce bogus `%INC` entries, and if a bogus entry exists then it will mislead this module if it is used to re-attempt loading.

The third bug worked around causes the wrong context to be seen at file scope of a loaded module, if `require` is invoked in a location that inherits context from a higher scope. This bug is present up to Perl 5.11.2, and is fixed in Perl 5.11.3. The workaround means that a module loaded via this module will always see the correct context. Modules loaded in other ways remain vulnerable.

REGULAR EXPRESSIONS

These regular expressions do not include any anchors, so to check whether an entire string matches a syntax item you must supply the anchors yourself.

`$module_name_rx`
: Matches a valid Perl module name in bareword syntax.

`$top_module_spec_rx`
: Matches a module specification for use with ''compose_module_name'', where no prefix is being used.

`$sub_module_spec_rx`
: Matches a module specification for use with ''compose_module_name'', where a prefix is being used.

FUNCTIONS

Basic module handling

is_module_name(ARG)
: Returns a truth value indicating whether *ARG* is a plain string satisfying Perl module name syntax as described for ''$module_name_rx''.

is_valid_module_name(ARG)
: Deprecated alias for ''is_module_name''.

check_module_name(ARG)
: Check whether *ARG* is a plain string satisfying Perl module name syntax as described for ''$module_name_rx''. Return normally if it is, or `die` if it is not.

module_notional_filename(NAME)
: Generates a notional relative filename for a module, which is used in some Perl core interfaces. The *NAME* is a string, which should be a valid module name (one or more `::`-separated segments). If it is not a valid name, the function `die`s.

  The notional filename for the named module is generated and returned. This filename is always in Unix style, with `/` directory separators and a `.pm` suffix. This kind of filename can be used as an argument to `require`, and is the key that appears in `%INC` to identify a module, regardless of actual local filename syntax.

require_module(NAME)

This is essentially the bareword form of `require`, in runtime form. The *NAME* is a string, which should be a valid module name (one or more `::`-separated segments). If it is not a valid name, the function `die`s.

The module specified by *NAME* is loaded, if it hasn't been already, in the manner of the bareword form of `require`. That means that a search through `@INC` is performed, and a byte-compiled form of the module will be used if available.

The return value is as for `require`. That is, it is the value returned by the module itself if the module is loaded anew, or `1` if the module was already loaded.

Structured module use

use_module(NAME[, VERSION])

This is essentially `use` in runtime form, but without the importing feature (which is fundamentally a compile-time thing). The *NAME* is handled just like in `require_module` above: it must be a module name, and the named module is loaded as if by the bareword form of `require`.

If a *VERSION* is specified, the `VERSION` method of the loaded module is called with the specified *VERSION* as an argument. This normally serves to ensure that the version loaded is at least the version required. This is the same functionality provided by the *VERSION* parameter of `use`.

On success, the name of the module is returned. This is unlike ''require_module'', and is done so that the entire call to ''use_module'' can be used as a class name to call a constructor, as in the example in the synopsis.

use_package_optimistically(NAME[, VERSION])

This is an analogue of ''use_module'' for the situation where there is uncertainty as to whether a package/class is defined in its own module or by some other means. It attempts to arrange for the named package to be available, either by loading a module or by doing nothing and hoping.

An attempt is made to load the named module (as if by the bareword form of `require`). If the module cannot be found then it is assumed that the package was actually already loaded by other means, and no error is signalled. That's the optimistic bit.

This is mostly the same operation that is performed by the base pragma to ensure that the specified base classes are available. The behaviour of base was simplified in version 2.18, and later improved in version 2.20, and on both occasions this function changed to match.

If a *VERSION* is specified, the `VERSION` method of the loaded package is called with the specified *VERSION* as an argument. This normally serves to ensure that the version loaded is at least the version required. On success, the name of the package is returned. These aspects of the function work just like ''use_module''.

Module name composition

is_module_spec(PREFIX, SPEC)

Returns a truth value indicating whether *SPEC* is valid input for ''compose_module_name''. See below for what that entails. Whether a *PREFIX* is supplied affects the validity of *SPEC*, but the exact value of the prefix is unimportant, so this function treats *PREFIX* as a truth value.

is_valid_module_spec(PREFIX, SPEC)

Deprecated alias for ''is_module_spec''.

check_module_spec(PREFIX, SPEC)

Check whether *SPEC* is valid input for ''compose_module_name''. Return normally if it is, or `die` if it is not.

compose_module_name(PREFIX, SPEC)

This function is intended to make it more convenient for a user to specify a Perl module name at runtime. Users have greater need for abbreviations and context-sensitivity than programmers, and Perl module names get a little unwieldy. *SPEC* is what the user specifies, and this function translates it into a module name in standard form, which it returns.

SPEC has syntax approximately that of a standard module name: it should consist of one or more name segments, each of which consists of one or more identifier characters. However, `/` is

permitted as a separator, in addition to the standard `::`. The two separators are entirely interchangeable.

Additionally, if *PREFIX* is not `undef` then it must be a module name in standard form, and it is prefixed to the user-specified name. The user can inhibit the prefix addition by starting *SPEC* with a separator (either `/` or `::`).

BUGS

On Perl versions 5.7.2 to 5.8.8, if `require` is overridden by the `CORE::GLOBAL` mechanism, it is likely to break the heuristics used by ''use_package_optimistically'', making it signal an error for a missing module rather than assume that it was already loaded. From Perl 5.8.9 onwards, and on 5.7.1 and earlier, this module can avoid being confused by such an override. On the affected versions, a `require` override might be installed by Lexical::SealRequireHints, if something requires its bugfix but for some reason its XS implementation isn't available.

SEE ALSO

Lexical::SealRequireHints, base, ''require'' in perlfunc, ''use'' in perlfunc

AUTHOR

Andrew Main (Zefram) <zefram AT fysh DOT org>

COPYRIGHT

LICENSE