NAME

CPAN - query, download and build perl modules from CPAN sites

SYNOPSIS

Interactive mode:

```
perl -MCPAN -e shell;
```

Batch mode:

```
use CPAN;

autobundle, clean, install, make, recompile, test
```

STATUS

This module will eventually be replaced by CPANPLUS. CPANPLUS is kind of a modern rewrite from ground up with greater extensibility and more features but no full compatibility. If you're new to CPAN.pm, you probably should investigate if CPANPLUS is the better choice for you. If you're already used to CPAN.pm you're welcome to continue using it, if you accept that its development is mostly (though not completely) stalled.

DESCRIPTION

The CPAN module is designed to automate the make and install of perl modules and extensions. It includes some primitive searching capabilities and knows how to use Net::FTP or LWP (or lynx or an external ftp client) to fetch the raw data from the net.

Modules are fetched from one or more of the mirrored CPAN (Comprehensive Perl Archive Network) sites and unpacked in a dedicated directory.

The CPAN module also supports the concept of named and versioned *bundles* of modules. Bundles simplify the handling of sets of related modules. See Bundles below.

The package contains a session manager and a cache manager. There is no status retained between sessions. The session manager keeps track of what has been fetched, built and installed in the current session. The cache manager keeps track of the disk space occupied by the make processes and deletes excess space according to a simple FIFO mechanism.

For extended searching capabilities there's a plugin for CPAN available, *`CPAN::WAIT`*. `CPAN::WAIT` is a full-text search engine that indexes all documents available in CPAN authors directories. If `CPAN::WAIT` is installed on your system, the interactive shell of CPAN.pm will enable the `wq`, `wr`, `wd`, `wl`, and `wh` commands which send queries to the WAIT server that has been configured for your installation.

All other methods provided are accessible in a programmer style and in an interactive shell style.

Interactive Mode

The interactive mode is entered by running

```
perl -MCPAN -e shell
```

which puts you into a readline interface. You will have the most fun if you install Term::ReadKey and Term::ReadLine to enjoy both history and command completion.

Once you are on the command line, type 'h' and the rest should be self-explanatory.

The function call `shell` takes two optional arguments, one is the prompt, the second is the default initial command line (the latter only works if a real ReadLine interface module is installed).

The most common uses of the interactive modes are

Searching for authors, bundles, distribution files and modules

There are corresponding one-letter commands `a`, `b`, `d`, and `m` for each of the four categories and another, `i` for any of the mentioned four. Each of the four entities is implemented as a class with slightly differing methods for displaying an object.

Arguments you pass to these commands are either strings exactly matching the identification string of an object or regular expressions that are then matched case-insensitively against various attributes of the objects. The parser recognizes a regular expression only if you enclose it between two slashes.

The principle is that the number of found objects influences how an item is displayed. If the search finds one item, the result is displayed with the rather verbose method `as_string`, but if we find more than one, we display each object with the terse method <as_glimpse>.

make, test, install, clean modules or distributions

These commands take any number of arguments and investigate what is necessary to perform the action. If the argument is a distribution file name (recognized by embedded slashes), it is processed. If it is a module, CPAN determines the distribution file in which this module is included and processes that, following any dependencies named in the module's Makefile.PL (this behavior is controlled by *prerequisites_policy*.)

Any `make` or `test` are run unconditionally. An

```
install <distribution_file>
```

also is run unconditionally. But for

```
install <module>
```

CPAN checks if an install is actually needed for it and prints *module up to date* in the case that the distribution file containing the module doesn't need to be updated.

CPAN also keeps track of what it has done within the current session and doesn't try to build a package a second time regardless if it succeeded or not. The `force` command takes as a first argument the method to invoke (currently: `make`, `test`, or `install`) and executes the command from scratch.

Example:

```
cpan> install OpenGL
OpenGL is up to date.
cpan> force install OpenGL
Running make
OpenGL-0.4/
OpenGL-0.4/COPYRIGHT
[...]
```

A `clean` command results in a

```
make clean
```

being executed within the distribution file's working directory.

get, readme, look module or distribution

`get` downloads a distribution file without further action. `readme` displays the README file of the associated distribution. `Look` gets and untars (if not yet done) the distribution file, changes to the appropriate directory and opens a subshell process in that directory.

ls author

`ls` lists all distribution files in and below an author's CPAN directory. Only those files that contain

modules are listed and if there is more than one for any given module, only the most recent one is listed.

Signals

CPAN.pm installs signal handlers for SIGINT and SIGTERM. While you are in the cpan-shell it is intended that you can press ^C anytime and return to the cpan-shell prompt. A SIGTERM will cause the cpan-shell to clean up and leave the shell loop. You can emulate the effect of a SIGTERM by sending two consecutive SIGINTs, which usually means by pressing ^C twice.

CPAN.pm ignores a SIGPIPE. If the user sets inactivity_timeout, a SIGALRM is used during the run of the `perl Makefile.PL` subprocess.

CPAN::Shell

The commands that are available in the shell interface are methods in the package CPAN::Shell. If you enter the shell command, all your input is split by the Text::ParseWords::shellwords() routine which acts like most shells do. The first word is being interpreted as the method to be called and the rest of the words are treated as arguments to this method. Continuation lines are supported if a line ends with a literal backslash.

autobundle

`autobundle` writes a bundle file into the `$CPAN::Config->{cpan_home}/Bundle` directory. The file contains a list of all modules that are both available from CPAN and currently installed within @INC. The name of the bundle file is based on the current date and a counter.

recompile

recompile() is a very special command in that it takes no argument and runs the make/test/install cycle with brute force over all installed dynamically loadable extensions (aka XS modules) with 'force' in effect. The primary purpose of this command is to finish a network installation. Imagine, you have a common source tree for two different architectures. You decide to do a completely independent fresh installation. You start on one architecture with the help of a Bundle file produced earlier. CPAN installs the whole Bundle for you, but when you try to repeat the job on the second architecture, CPAN responds with a `"Foo up to date"` message for all modules. So you invoke CPAN's recompile on the second architecture and you're done.

Another popular use for `recompile` is to act as a rescue in case your perl breaks binary compatibility. If one of the modules that CPAN uses is in turn depending on binary compatibility (so you cannot run CPAN commands), then you should try the CPAN::Nox module for recovery.

The four CPAN::* Classes: Author, Bundle, Module, Distribution

Although it may be considered internal, the class hierarchy does matter for both users and programmer. CPAN.pm deals with above mentioned four classes, and all those classes share a set of methods. A classical single polymorphism is in effect. A metaclass object registers all objects of all kinds and indexes them with a string. The strings referencing objects have a separated namespace (well, not completely separated):

```
        Namespace                         Class

  words containing a "/" (slash)      Distribution
   words starting with Bundle::           Bundle
         everything else            Module or Author
```

Modules know their associated Distribution objects. They always refer to the most recent official release. Developers may mark their releases as unstable development versions (by inserting an underbar into the module version number which will also be reflected in the distribution name when you run 'make dist'), so the really hottest and newest distribution is not always the default. If a module Foo circulates on CPAN in both version 1.23 and 1.23_90, CPAN.pm offers a convenient way to install version 1.23 by saying

```
install Foo
```

This would install the complete distribution file (say BAR/Foo-1.23.tar.gz) with all accompanying material. But if you would like to install version 1.23_90, you need to know where the distribution file resides on CPAN relative to the authors/id/ directory. If the author is BAR, this might be BAR/Foo-1.23_90.tar.gz; so you would have to say

```
install BAR/Foo-1.23_90.tar.gz
```

The first example will be driven by an object of the class CPAN::Module, the second by an object of class CPAN::Distribution.

Programmer's interface

If you do not enter the shell, the available shell commands are both available as methods (`CPAN::Shell->install(...)`) and as functions in the calling package (`install(...)`).

There's currently only one class that has a stable interface - CPAN::Shell. All commands that are available in the CPAN shell are methods of the class CPAN::Shell. Each of the commands that produce listings of modules (`r`, `autobundle`, u) also return a list of the IDs of all modules within the list.

expand($type,@things)

The IDs of all objects available within a program are strings that can be expanded to the corresponding real objects with the `CPAN::Shell->expand("Module",@things)` method. Expand returns a list of CPAN::Module objects according to the `@things` arguments given. In scalar context it only returns the first element of the list.

expandany(@things)

Like expand, but returns objects of the appropriate type, i.e. CPAN::Bundle objects for bundles, CPAN::Module objects for modules and CPAN::Distribution objects fro distributions.

Programming Examples

This enables the programmer to do operations that combine functionalities that are available in the shell.

```
    # install everything that is outdated on my disk:
    perl -MCPAN -e 'CPAN::Shell->install(CPAN::Shell->r)'

    # install my favorite programs if necessary:
    for $mod (qw(Net::FTP Digest::MD5 Data::Dumper)){
        my $obj = CPAN::Shell->expand('Module',$mod);
        $obj->install;
    }

    # list all modules on my disk that have no VERSION number
    for $mod (CPAN::Shell->expand("Module","/./")){
next unless $mod->inst_file;
        # MakeMaker convention for undefined $VERSION:
next unless $mod->inst_version eq "undef";
print "No VERSION in ", $mod->id, "\n";
    }

    # find out which distribution on CPAN contains a module:
    print CPAN::Shell->expand("Module","Apache::Constants")->cpan_file
```

Or if you want to write a cronjob to watch The CPAN, you could list all modules that need updating. First a quick and dirty way:

```
perl -e 'use CPAN; CPAN::Shell->r;'
```

If you don't want to get any output in the case that all modules are up to date, you can parse the output of above command for the regular expression //modules are up to date// and decide to mail the output only if it doesn't match. Ick?

If you prefer to do it more in a programmer style in one single process, maybe something like this suits you better:

```
  # list all modules on my disk that have newer versions on CPAN
  for $mod (CPAN::Shell->expand("Module","/./")){
    next unless $mod->inst_file;
    next if $mod->uptodate;
    printf "Module %s is installed as %s, could be updated to %s from
CPAN\n",
        $mod->id, $mod->inst_version, $mod->cpan_version;
  }
```

If that gives you too much output every day, you maybe only want to watch for three modules. You can write

```
  for $mod (CPAN::Shell->expand("Module","/Apache|LWP|CGI/")){
```

as the first line instead. Or you can combine some of the above tricks:

```
  # watch only for a new mod_perl module
  $mod = CPAN::Shell->expand("Module","mod_perl");
  exit if $mod->uptodate;
  # new mod_perl arrived, let me know all update recommendations
  CPAN::Shell->r;
```

Methods in the other Classes

The programming interface for the classes CPAN::Module, CPAN::Distribution, CPAN::Bundle, and CPAN::Author is still considered beta and partially even alpha. In the following paragraphs only those methods are documented that have proven useful over a longer time and thus are unlikely to change.

CPAN::Author::as_glimpse()

Returns a one-line description of the author

CPAN::Author::as_string()

Returns a multi-line description of the author

CPAN::Author::email()

Returns the author's email address

CPAN::Author::fullname()

Returns the author's name

CPAN::Author::name()

An alias for fullname

CPAN::Bundle::as_glimpse()

Returns a one-line description of the bundle

CPAN::Bundle::as_string()

Returns a multi-line description of the bundle

CPAN::Bundle::clean()

Recursively runs the `clean` method on all items contained in the bundle.

CPAN::Bundle::contains()

Returns a list of objects' IDs contained in a bundle. The associated objects may be bundles, modules or distributions.

CPAN::Bundle::force($method,@args)

Forces CPAN to perform a task that normally would have failed. Force takes as arguments a method name to be called and any number of additional arguments that should be passed to the called method. The internals of the object get the needed changes so that CPAN.pm does not refuse to take the action. The `force` is passed recursively to all contained objects.

CPAN::Bundle::get()

Recursively runs the `get` method on all items contained in the bundle

CPAN::Bundle::inst_file()

Returns the highest installed version of the bundle in either @INC or `$CPAN::Config->{cpan_home}`. Note that this is different from CPAN::Module::inst_file.

CPAN::Bundle::inst_version()

Like CPAN::Bundle::inst_file, but returns the $VERSION

CPAN::Bundle::uptodate()

Returns 1 if the bundle itself and all its members are uptodate.

CPAN::Bundle::install()

Recursively runs the `install` method on all items contained in the bundle

CPAN::Bundle::make()

Recursively runs the `make` method on all items contained in the bundle

CPAN::Bundle::readme()

Recursively runs the `readme` method on all items contained in the bundle

CPAN::Bundle::test()

Recursively runs the `test` method on all items contained in the bundle

CPAN::Distribution::as_glimpse()

Returns a one-line description of the distribution

CPAN::Distribution::as_string()

Returns a multi-line description of the distribution

CPAN::Distribution::clean()

Changes to the directory where the distribution has been unpacked and runs `make clean` there.

CPAN::Distribution::containsmods()

Returns a list of IDs of modules contained in a distribution file. Only works for distributions listed in the 02packages.details.txt.gz file. This typically means that only the most recent version of a distribution is covered.

CPAN::Distribution::cvs_import()

Changes to the directory where the distribution has been unpacked and runs something like

```
cvs -d $cvs_root import -m $cvs_log $cvs_dir $userid v$version
```

there.

CPAN::Distribution::dir()

Returns the directory into which this distribution has been unpacked.

CPAN::Distribution::force($method,@args)

Forces CPAN to perform a task that normally would have failed. Force takes as arguments a method name to be called and any number of additional arguments that should be passed to the called method. The internals of the object get the needed changes so that CPAN.pm does not refuse to take the action.

CPAN::Distribution::get()

Downloads the distribution from CPAN and unpacks it. Does nothing if the distribution has already been downloaded and unpacked within the current session.

CPAN::Distribution::install()

Changes to the directory where the distribution has been unpacked and runs the external command `make install` there. If `make` has not yet been run, it will be run first. A `make test` will be issued in any case and if this fails, the install will be canceled. The cancellation can be avoided by letting `force` run the `install` for you.

CPAN::Distribution::isa_perl()

Returns 1 if this distribution file seems to be a perl distribution. Normally this is derived from the file name only, but the index from CPAN can contain a hint to achieve a return value of true for other filenames too.

CPAN::Distribution::look()

Changes to the directory where the distribution has been unpacked and opens a subshell there. Exiting the subshell returns.

CPAN::Distribution::make()

First runs the `get` method to make sure the distribution is downloaded and unpacked. Changes to the directory where the distribution has been unpacked and runs the external commands `perl Makefile.PL` and `make` there.

CPAN::Distribution::prereq_pm()

Returns the hash reference that has been announced by a distribution as the PREREQ_PM hash in the Makefile.PL. Note: works only after an attempt has been made to `make` the distribution. Returns undef otherwise.

CPAN::Distribution::readme()

Downloads the README file associated with a distribution and runs it through the pager specified in `$CPAN::Config-`{pager}>.

CPAN::Distribution::test()

Changes to the directory where the distribution has been unpacked and runs `make test` there.

CPAN::Distribution::uptodate()

Returns 1 if all the modules contained in the distribution are uptodate. Relies on containsmods.

CPAN::Index::force_reload()

Forces a reload of all indices.

CPAN::Index::reload()

Reloads all indices if they have been read more than `$CPAN::Config-{index_expire}>` days.

CPAN::InfoObj::dump()

CPAN::Author, CPAN::Bundle, CPAN::Module, and CPAN::Distribution inherit this method. It prints the data structure associated with an object. Useful for debugging. Note: the data structure is considered internal and thus subject to change without notice.

CPAN::Module::as_glimpse()

Returns a one-line description of the module

CPAN::Module::as_string()

Returns a multi-line description of the module

CPAN::Module::clean()

Runs a clean on the distribution associated with this module.

CPAN::Module::cpan_file()

Returns the filename on CPAN that is associated with the module.

CPAN::Module::cpan_version()

Returns the latest version of this module available on CPAN.

CPAN::Module::cvs_import()

Runs a cvs_import on the distribution associated with this module.

CPAN::Module::description()

Returns a 44 character description of this module. Only available for modules listed in The Module List (CPAN/modules/00modlist.long.html or 00modlist.long.txt.gz)

CPAN::Module::force($method,@args)

Forces CPAN to perform a task that normally would have failed. Force takes as arguments a method name to be called and any number of additional arguments that should be passed to the called method. The internals of the object get the needed changes so that CPAN.pm does not refuse to take the action.

CPAN::Module::get()

Runs a get on the distribution associated with this module.

CPAN::Module::inst_file()

Returns the filename of the module found in @INC. The first file found is reported just like perl itself stops searching @INC when it finds a module.

CPAN::Module::inst_version()

Returns the version number of the module in readable format.

CPAN::Module::install()

Runs an `install` on the distribution associated with this module.

CPAN::Module::look()

Changes to the directory where the distribution associated with this module has been unpacked and opens a subshell there. Exiting the subshell returns.

CPAN::Module::make()

Runs a `make` on the distribution associated with this module.

CPAN::Module::manpage_headline()

If module is installed, peeks into the module's manpage, reads the headline and returns it. Moreover, if the module has been downloaded within this session, does the equivalent on the downloaded module even if it is not installed.

CPAN::Module::readme()

Runs a `readme` on the distribution associated with this module.

CPAN::Module::test()

Runs a `test` on the distribution associated with this module.

CPAN::Module::uptodate()

Returns 1 if the module is installed and up-to-date.

CPAN::Module::userid()

Returns the author's ID of the module.

Cache Manager

Currently the cache manager only keeps track of the build directory ($CPAN::Config->{build_dir}). It is a simple FIFO mechanism that deletes complete directories below `build_dir` as soon as the size of all directories there gets bigger than $CPAN::Config->{build_cache} (in MB). The contents of this cache may be used for later re-installations that you intend to do manually, but will never be trusted by CPAN itself. This is due to the fact that the user might use these directories for building modules on different architectures.

There is another directory ($CPAN::Config->{keep_source_where}) where the original distribution files are kept. This directory is not covered by the cache manager and must be controlled by the user. If you choose to have the same directory as build_dir and as keep_source_where directory, then your sources will be deleted with the same fifo mechanism.

Bundles

A bundle is just a perl module in the namespace Bundle:: that does not define any functions or methods. It usually only contains documentation.

It starts like a perl module with a package declaration and a $VERSION variable. After that the pod section looks like any other pod with the only difference being that *one special pod section* exists starting with (verbatim):

```
=head1 CONTENTS
```

In this pod section each line obeys the format

```
Module_Name [Version_String] [- optional text]
```

The only required part is the first field, the name of a module (e.g. Foo::Bar, ie. *not* the name of the distribution file). The rest of the line is optional. The comment part is delimited by a dash just as in the man page header.

The distribution of a bundle should follow the same convention as other distributions.

Bundles are treated specially in the CPAN package. If you say 'install Bundle::Tkkit' (assuming such a bundle exists), CPAN will install all the modules in the CONTENTS section of the pod. You can install your own Bundles locally by placing a conformant Bundle file somewhere into your @INC path. The autobundle() command which is available in the shell interface does that for you by including all currently installed modules in a snapshot bundle file.

Prerequisites

If you have a local mirror of CPAN and can access all files with "file:" URLs, then you only need a perl better than perl5.003 to run this module. Otherwise Net::FTP is strongly recommended. LWP may be required for non-UNIX systems or if your nearest CPAN site is associated with a URL that is not `ftp:` .

If you have neither Net::FTP nor LWP, there is a fallback mechanism implemented for an external ftp command or for an external lynx command.

Finding packages and VERSION

This module presumes that all packages on CPAN

- declare their $VERSION variable in an easy to parse manner. This prerequisite can hardly be relaxed because it consumes far too much memory to load all packages into the running program just to determine the $VERSION variable. Currently all programs that are dealing with version use something like this

```
perl -MExtUtils::MakeMaker -le \
    'print MM->parse_version(shift)' filename
```

  If you are author of a package and wonder if your $VERSION can be parsed, please try the above method.

- come as compressed or gzipped tarfiles or as zip files and contain a Makefile.PL (well, we try to handle a bit more, but without much enthusiasm).

Debugging

The debugging of this module is a bit complex, because we have interferences of the software producing the indices on CPAN, of the mirroring process on CPAN, of packaging, of configuration, of synchronicity, and of bugs within CPAN.pm.

For code debugging in interactive mode you can try "o debug" which will list options for debugging the various parts of the code. You should know that "o debug" has built-in completion support.

For data debugging there is the `dump` command which takes the same arguments as make/test/install and outputs the object's Data::Dumper dump.

Floppy, Zip, Offline Mode

CPAN.pm works nicely without network too. If you maintain machines that are not networked at all, you should consider working with file: URLs. Of course, you have to collect your modules somewhere first. So you might use CPAN.pm to put together all you need on a networked machine. Then copy the $CPAN::Config->{keep_source_where} (but not $CPAN::Config->{build_dir}) directory on a floppy. This floppy is kind of a personal CPAN. CPAN.pm on the non-networked machines works nicely with this floppy. See also below the paragraph about CD-ROM support.

CONFIGURATION

When the CPAN module is used for the first time, a configuration dialog tries to determine a couple of site specific options. The result of the dialog is stored in a hash reference `$CPAN::Config` in a file CPAN/Config.pm.

The default values defined in the CPAN/Config.pm file can be overridden in a user specific file: CPAN/MyConfig.pm. Such a file is best placed in $HOME/.cpan/CPAN/MyConfig.pm, because $HOME/.cpan is added to the search path of the CPAN module before the use() or require() statements.

The configuration dialog can be started any time later again by issueing the command `o conf init` in the CPAN shell.

Currently the following keys in the hash reference $CPAN::Config are defined:

```
build_cache         size of cache for directories to build modules
build_dir           locally accessible directory to build modules
index_expire        after this many days refetch index files
cache_metadata      use serializer to cache metadata
cpan_home           local directory reserved for this package
dontload_hash       anonymous hash: modules in the keys will not be
                    loaded by the CPAN::has_inst() routine
gzip       location of external program gzip
histfile            file to maintain history between sessions
histsize            maximum number of lines to keep in histfile
inactivity_timeout breaks interactive Makefile.PLs after this
                    many seconds inactivity. Set to 0 to never break.
inhibit_startup_message
                    if true, does not print the startup message
keep_source_where   directory in which to keep the source (if we do)
make                location of external make program
make_arg      arguments that should always be passed to 'make'
make_install_arg    same as make_arg for 'make install'
makepl_arg      arguments passed to 'perl Makefile.PL'
pager               location of external program more (or any pager)
prerequisites_policy
                    what to do if you are missing module prerequisites
                    ('follow' automatically, 'ask' me, or 'ignore')
proxy_user          username for accessing an authenticating proxy
proxy_pass          password for accessing an authenticating proxy
scan_cache      controls scanning of cache ('atstart' or 'never')
tar                 location of external program tar
term_is_latin       if true internal UTF-8 is translated to ISO-8859-1
                    (and nonsense for characters outside latin range)
unzip               location of external program unzip
urllist      arrayref to nearby CPAN sites (or equivalent locations)
wait_list           arrayref to a wait server to try (See CPAN::WAIT)
ftp_proxy,      }   the three usual variables for configuring
  http_proxy,   }   proxy requests. Both as CPAN::Config variables
  no_proxy      }   and as environment variables configurable.
```

You can set and query each of these options interactively in the cpan shell with the command set defined within the `o conf` command:

`o conf <scalar option>`

prints the current value of the *scalar option*

`o conf <scalar option> <value>`

Sets the value of the *scalar option* to *value*

`o conf <list option>`

prints the current value of the *list option* in MakeMaker's neatvalue format.

`o conf <list option> [shift|pop]`

shifts or pops the array in the *list option* variable

`o conf <list option> [unshift|push|splice] <list>`

works like the corresponding perl commands.

Note on urllist parameter's format

urllist parameters are URLs according to RFC 1738. We do a little guessing if your URL is not compliant, but if you have problems with file URLs, please try the correct format. Either:

```
file://localhost/whatever/ftp/pub/CPAN/
```

or

```
file:///home/ftp/pub/CPAN/
```

urllist parameter has CD-ROM support

The `urllist` parameter of the configuration table contains a list of URLs that are to be used for downloading. If the list contains any `file` URLs, CPAN always tries to get files from there first. This feature is disabled for index files. So the recommendation for the owner of a CD-ROM with CPAN contents is: include your local, possibly outdated CD-ROM as a `file` URL at the end of urllist, e.g.

```
  o conf urllist push file://localhost/CDROM/CPAN
```

CPAN.pm will then fetch the index files from one of the CPAN sites that come at the beginning of urllist. It will later check for each module if there is a local copy of the most recent version.

Another peculiarity of urllist is that the site that we could successfully fetch the last file from automatically gets a preference token and is tried as the first site for the next request. So if you add a new site at runtime it may happen that the previously preferred site will be tried another time. This means that if you want to disallow a site for the next transfer, it must be explicitly removed from urllist.

SECURITY

There's no strong security layer in CPAN.pm. CPAN.pm helps you to install foreign, unmasked, unsigned code on your machine. We compare to a checksum that comes from the net just as the distribution file itself. If somebody has managed to tamper with the distribution file, they may have as well tampered with the CHECKSUMS file. Future development will go towards strong authentication.

EXPORT

Most functions in package CPAN are exported per default. The reason for this is that the primary use is intended for the cpan shell or for one-liners.

POPULATE AN INSTALLATION WITH LOTS OF MODULES

Populating a freshly installed perl with my favorite modules is pretty easy if you maintain a private bundle definition file. To get a useful blueprint of a bundle definition file, the command autobundle can be used on the CPAN shell command line. This command writes a bundle definition file for all modules that are installed for the currently running perl interpreter. It's recommended to run this command only once and from then on maintain the file manually under a private name, say Bundle/my_bundle.pm. With a clever bundle file you can then simply say

```
cpan> install Bundle::my_bundle
```

then answer a few questions and then go out for a coffee.

Maintaining a bundle definition file means keeping track of two things: dependencies and interactivity. CPAN.pm sometimes fails on calculating dependencies because not all modules define all MakeMaker attributes correctly, so a bundle definition file should specify prerequisites as early as possible. On the other hand, it's a bit annoying that many distributions need some interactive configuring. So what I try to accomplish in my private bundle file is to have the packages that need to be configured early in the file and the gentle ones later, so I can go out after a few minutes and leave CPAN.pm untended.

WORKING WITH CPAN.pm BEHIND FIREWALLS

Thanks to Graham Barr for contributing the following paragraphs about the interaction between perl, and various firewall configurations. For further informations on firewalls, it is recommended to consult the documentation that comes with the ncftp program. If you are unable to go through the firewall with a simple Perl setup, it is very likely that you can configure ncftp so that it works for your firewall.

Three basic types of firewalls

Firewalls can be categorized into three basic types.

http firewall

This is where the firewall machine runs a web server and to access the outside world you must do it via the web server. If you set environment variables like http_proxy or ftp_proxy to a values beginning with http:// or in your web browser you have to set proxy information then you know you are running an http firewall.

To access servers outside these types of firewalls with perl (even for ftp) you will need to use LWP.

ftp firewall

This where the firewall machine runs an ftp server. This kind of firewall will only let you access ftp servers outside the firewall. This is usually done by connecting to the firewall with ftp, then entering a username like "user@outside.host.com"

To access servers outside these type of firewalls with perl you will need to use Net::FTP.

One way visibility

I say one way visibility as these firewalls try to make themselves look invisible to the users inside the firewall. An FTP data connection is normally created by sending the remote server your IP address and then listening for the connection. But the remote server will not be able to connect to you because of the firewall. So for these types of firewall FTP connections need to be done in a passive mode.

There are two that I can think off.

SOCKS

If you are using a SOCKS firewall you will need to compile perl and link it with the SOCKS library, this is what is normally called a 'socksified' perl. With this executable you will be able to connect to servers outside the firewall as if it is not there.

IP Masquerade

This is the firewall implemented in the Linux kernel, it allows you to hide a complete network behind one IP address. With this firewall no special compiling is needed as you can access hosts directly.

For accessing ftp servers behind such firewalls you may need to set the environment variable FTP_PASSIVE to a true value, e.g.

```
env FTP_PASSIVE=1 perl -MCPAN -eshell
```

or

```
perl -MCPAN -e '$ENV{FTP_PASSIVE} = 1; shell'
```

Configuring lynx or ncftp for going through a firewall

If you can go through your firewall with e.g. lynx, presumably with a command such as

```
/usr/local/bin/lynx -pscott:tiger
```

then you would configure CPAN.pm with the command

```
o conf lynx "/usr/local/bin/lynx -pscott:tiger"
```

That's all. Similarly for ncftp or ftp, you would configure something like

```
o conf ncftp "/usr/bin/ncftp -f /home/scott/ncftplogin.cfg"
```

Your mileage may vary...

FAQ

1)

I installed a new version of module X but CPAN keeps saying, I have the old version installed

Most probably you **do** have the old version installed. This can happen if a module installs itself into a different directory in the @INC path than it was previously installed. This is not really a CPAN.pm problem, you would have the same problem when installing the module manually. The easiest way to prevent this behaviour is to add the argument `UNINST=1` to the `make install` call, and that is why many people add this argument permanently by configuring

```
o conf make_install_arg UNINST=1
```

2)

So why is UNINST=1 not the default?

Because there are people who have their precise expectations about who may install where in the @INC path and who uses which @INC array. In fine tuned environments `UNINST=1` can cause damage.

3)

I want to clean up my mess, and install a new perl along with all modules I have. How do I go about it?

Run the autobundle command for your old perl and optionally rename the resulting bundle file (e.g. Bundle/mybundle.pm), install the new perl with the Configure option prefix, e.g.

```
./Configure -Dprefix=/usr/local/perl-5.6.78.9
```

Install the bundle file you produced in the first step with something like

```
cpan> install Bundle::mybundle
```

and you're done.

4)

When I install bundles or multiple modules with one command there is too much output to keep track of.

You may want to configure something like

```
o conf make_arg "| tee -ai /root/.cpan/logs/make.out"
o conf make_install_arg "| tee -ai
/root/.cpan/logs/make_install.out"
```

so that STDOUT is captured in a file for later inspection.

5)

I am not root, how can I install a module in a personal directory?

You will most probably like something like this:

```
o conf makepl_arg "LIB=~/myperl/lib \
                   INSTALLMAN1DIR=~/myperl/man/man1 \
```

```
                                INSTALLMAN3DIR=~/myperl/man/man3"
  install Sybase::Sybperl
```

You can make this setting permanent like all `o conf` settings with `o conf commit`.

You will have to add ~/myperl/man to the MANPATH environment variable and also tell your perl programs to look into ~/myperl/lib, e.g. by including

```
  use lib "$ENV{HOME}/myperl/lib";
```

or setting the PERL5LIB environment variable.

Another thing you should bear in mind is that the UNINST parameter should never be set if you are not root.

6)

How to get a package, unwrap it, and make a change before building it?

```
  look Sybase::Sybperl
```

7)

I installed a Bundle and had a couple of fails. When I retried, everything resolved nicely. Can this be fixed to work on first try?

The reason for this is that CPAN does not know the dependencies of all modules when it starts out. To decide about the additional items to install, it just uses data found in the generated Makefile. An undetected missing piece breaks the process. But it may well be that your Bundle installs some prerequisite later than some depending item and thus your second try is able to resolve everything. Please note, CPAN.pm does not know the dependency tree in advance and cannot sort the queue of things to install in a topologically correct order. It resolves perfectly well IFF all modules declare the prerequisites correctly with the PREREQ_PM attribute to MakeMaker. For bundles which fail and you need to install often, it is recommended sort the Bundle definition file manually. It is planned to improve the metadata situation for dependencies on CPAN in general, but this will still take some time.

8)

In our intranet we have many modules for internal use. How can I integrate these modules with CPAN.pm but without uploading the modules to CPAN?

Have a look at the CPAN::Site module.

9)

When I run CPAN's shell, I get error msg about line 1 to 4, setting meta input/output via the /etc/inputrc file.

Some versions of readline are picky about capitalization in the /etc/inputrc file and specifically RedHat 6.2 comes with a /etc/inputrc that contains the word `on` in lowercase. Change the occurrences of `on` to `On` and the bug should disappear.

10)

Some authors have strange characters in their names.

Internally CPAN.pm uses the UTF-8 charset. If your terminal is expecting ISO-8859-1 charset, a converter can be activated by setting term_is_latin to a true value in your config file. One way of doing so would be

```
    cpan> ! $CPAN::Config->{term_is_latin}=1
```

Extended support for converters will be made available as soon as perl becomes stable with regard to charset issues.

BUGS

We should give coverage for **all** of the CPAN and not just the PAUSE part, right? In this discussion CPAN and PAUSE have become equal -- but they are not. PAUSE is authors/, modules/ and scripts/. CPAN is PAUSE plus the clpa/, doc/, misc/, ports/, and src/.

Future development should be directed towards a better integration of the other parts.

If a Makefile.PL requires special customization of libraries, prompts the user for special input, etc. then you may find CPAN is not able to build the distribution. In that case, you should attempt the traditional method of building a Perl module package from a shell.

AUTHOR

Andreas Koenig <andreas.koenig@anima.de>

TRANSLATIONS

Kawai,Takanori provides a Japanese translation of this manpage at http://member.nifty.ne.jp/hippo2000/perltips/CPAN.htm

SEE ALSO

perl(1), CPAN::Nox(3)