NAME

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

SYNOPSIS

Interactive mode:

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

--or--

```
cpan
```

Basic commands:

```
# Modules:

cpan> install Acme::Meta                        # in the shell

CPAN::Shell->install("Acme::Meta");             # in perl

# Distributions:

cpan> install NWCLARK/Acme-Meta-0.02.tar.gz     # in the shell

CPAN::Shell->
  install("NWCLARK/Acme-Meta-0.02.tar.gz");     # in perl

# module objects:

$mo = CPAN::Shell->expandany($mod);
$mo = CPAN::Shell->expand("Module",$mod);       # same thing

# distribution objects:

$do = CPAN::Shell->expand("Module",$mod)->distribution;
$do = CPAN::Shell->expandany($distro);          # same thing
$do = CPAN::Shell->expand("Distribution",
                          $distro);             # same thing
```

DESCRIPTION

The CPAN module automates or at least simplifies 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 some external download clients to fetch the distributions from the net.

These 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. 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.

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

CPAN::shell([$prompt, $command]) Starting Interactive Mode

The interactive mode is entered by running

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

or

```
cpan
```

which puts you into a readline interface. If `Term::ReadKey` and either `Term::ReadLine::Perl` or `Term::ReadLine::Gnu` are installed it supports both history and command completion.

Once you are on the command line, type h to get a one page help screen 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`.

`get`, `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 META.yml or Makefile.PL (this behavior is controlled by the configuration parameter `prerequisites_policy`.)

`get` downloads a distribution file and untars or unzips it, `make` builds it, `test` runs the test suite, and `install` installs it.

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. It does not repeat a test run if the test has been run successfully before. Same for install runs.

The `force` pragma may precede another command (currently: `get`, `make`, `test`, or `install`) and executes the command from scratch and tries to continue in case of some errors. See the section below on the `force` and the `fforce` pragma.

The `notest` pragma may be used to skip the test part in the build process.

Example:

```
cpan> notest install Tk
```

A `clean` command results in a

```
make clean
```

being executed within the distribution file's working directory.

`readme`, `perldoc`, `look` module or distribution

`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. `perldoc` displays the pod documentation of the module in html or plain text format.

`ls` author

`ls` globbing_expression

The first form lists all distribution files in and below an author's CPAN directory as they are stored in the CHECKUMS files distributed on CPAN. The listing goes recursive into all subdirectories.

The second form allows to limit or expand the output with shell globbing as in the following examples:

```
ls JV/make*
ls GSAR/*make*
ls */*make*
```

The last example is very slow and outputs extra progress indicators that break the alignment of the result.

Note that globbing only lists directories explicitly asked for, for example FOO/* will not list FOO/bar/Acme-Sthg-n.nn.tar.gz. This may be regarded as a bug and may be changed in future versions.

`failed`

The `failed` command reports all distributions that failed on one of `make`, `test` or `install` for some reason in the currently running shell session.

Persistence between sessions

If the `YAML` or the `YAML::Syck` module is installed a record of the internal state of all modules is written to disk after each step. The files contain a signature of the currently running perl version for later perusal.

If the configurations variable `build_dir_reuse` is set to a true value, then CPAN.pm reads the collected YAML files. If the stored signature matches the currently running perl the stored state is loaded into memory such that effectively persistence between sessions is established.

The `force` and the `fforce` pragma

To speed things up in complex installation scenarios, CPAN.pm keeps track of what it has already done and refuses to do some things a second time. A `get`, a `make`, and an `install` are not repeated. A `test` is only repeated if the previous test was unsuccessful. The diagnostic message when CPAN.pm refuses to do something a second time is one of *Has already been* `unwrapped|made|tested successfully` or something similar. Another situation where CPAN refuses to act is an `install` if the according `test` was not successful.

In all these cases, the user can override the goatish behaviour by prepending the command with

the word force, for example:

```
cpan> force get Foo
cpan> force make AUTHOR/Bar-3.14.tar.gz
cpan> force test Baz
cpan> force install Acme::Meta
```

Each *forced* command is executed with the according part of its memory erased.

The `fforce` pragma is a variant that emulates a `force get` which erases the entire memory followed by the action specified, effectively restarting the whole get/make/test/install procedure from scratch.

Lockfile

Interactive sessions maintain a lockfile, per default `~/.cpan/.lock`. Batch jobs can run without a lockfile and do not disturb each other.

The shell offers to run in *degraded mode* when another process is holding the lockfile. This is an experimental feature that is not yet tested very well. This second shell then does not write the history file, does not use the metadata file and has a different prompt.

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` or `perl Build.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.

hosts

Note: this feature is still in alpha state and may change in future versions of CPAN.pm

This commands provides a statistical overview over recent download activities. The data for this is collected in the YAML file `FTPstats.yml` in your `cpan_home` directory. If no YAML module is configured or YAML not installed, then no stats are provided.

mkmyconfig

mkmyconfig() writes your own CPAN::MyConfig file into your ~/.cpan/ directory so that you can save your own preferences instead of the system wide ones.

recent ***EXPERIMENTAL COMMAND***

The `recent` command downloads a list of recent uploads to CPAN and displays them *slowly*. While the command is running $SIG{INT} is defined to mean that the loop shall be left after having displayed the current item.

Note: This command requires XML::LibXML installed.

Note: This whole command currently is a bit klunky and will probably change in future versions of CPAN.pm but the general approach will likely stay.

Note: See also *smoke*

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.

report Bundle|Distribution|Module

The `report` command temporarily turns on the `test_report` config variable, then runs the `force test` command with the given arguments. The `force` pragma is used to re-run the tests and repeat every step that might have failed before.

smoke ***EXPERIMENTAL COMMAND***

***** WARNING: this command downloads and executes software from CPAN to your computer of completely unknown status. You should never do this with your normal account and better have a dedicated well separated and secured machine to do this. *****

The `smoke` command takes the list of recent uploads to CPAN as provided by the `recent` command and tests them all. While the command is running $SIG{INT} is defined to mean that the current item shall be skipped.

Note: This whole command currently is a bit klunky and will probably change in future versions of CPAN.pm but the general approach will likely stay.

Note: See also *recent*

upgrade [Module|/Regex/]...

The `upgrade` command first runs an `r` command with the given arguments and then installs the newest versions of all modules that were listed by that.

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.

Integrating local directories

Note: this feature is still in alpha state and may change in future versions of CPAN.pm

Distribution objects are normally distributions from the CPAN, but there is a slightly degenerate case for Distribution objects, too, of projects held on the local disk. These distribution objects have the same name as the local directory and end with a dot. A dot by itself is also allowed for the current directory at the time CPAN.pm was used. All actions such as `make`, `test`, and `install` are applied directly to that directory. This gives the command `cpan .` an interesting touch: while the normal mantra of installing a CPAN module without CPAN.pm is one of

```
perl Makefile.PL                  perl Build.PL
       ( go and get prerequisites )
make                              ./Build
make test                         ./Build test
make install                      ./Build install
```

the command `cpan .` does all of this at once. It figures out which of the two mantras is appropriate, fetches and installs all prerequisites, cares for them recursively and finally finishes the installation of the module in the current directory, be it a CPAN module or not.

The typical usage case is for private modules or working copies of projects from remote repositories on the local disk.

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 mkmyconfig command writes this file for you.

The `o conf` command has various bells and whistles:

completion support

> If you have a ReadLine module installed, you can hit TAB at any point of the commandline and `o conf` will offer you completion for the built-in subcommands and/or config variable names.

displaying some help: o conf help

Displays a short help

displaying current values: o conf [KEY]

Displays the current value(s) for this config variable. Without KEY displays all subcommands and config variables.

Example:

```
o conf shell
```

If KEY starts and ends with a slash the string in between is interpreted as a regular expression and only keys matching this regex are displayed

Example:

```
o conf /color/
```

changing of scalar values: o conf KEY VALUE

Sets the config variable KEY to VALUE. The empty string can be specified as usual in shells, with `''` or `""`

Example:

```
o conf wget /usr/bin/wget
```

changing of list values: o conf KEY SHIFT|UNSHIFT|PUSH|POP|SPLICE|LIST

If a config variable name ends with `list`, it is a list. `o conf KEY shift` removes the first element of the list, `o conf KEY pop` removes the last element of the list. `o conf KEYS unshift LIST` prepends a list of values to the list, `o conf KEYS push LIST` appends a list of valued to the list.

Likewise, `o conf KEY splice LIST` passes the LIST to the according splice command.

Finally, any other list of arguments is taken as a new list value for the KEY variable discarding the previous value.

Examples:

```
o conf urllist unshift http://cpan.dev.local/CPAN
o conf urllist splice 3 1
o conf urllist http://cpan1.local http://cpan2.local
ftp://ftp.perl.org
```

reverting to saved: o conf defaults

Reverts all config variables to the state in the saved config file.

saving the config: o conf commit

Saves all config variables to the current config file (CPAN/Config.pm or CPAN/MyConfig.pm that was loaded at start).

The configuration dialog can be started any time later again by issuing the command `o conf init` in the CPAN shell. A subset of the configuration dialog can be run by issuing `o conf init WORD` where WORD is any valid config variable or a regular expression.

Config Variables

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

```
applypatch         path to external prg
auto_commit        commit all changes to config variables to disk
build_cache        size of cache for directories to build modules
build_dir          locally accessible directory to build modules
```

```
    build_dir_reuse    boolean if distros in build_dir are persistent
    build_requires_install_policy
                       to install or not to install when a module is
                       only needed for building. yes|no|ask/yes|ask/no
    bzip2              path to external prg
    cache_metadata     use serializer to cache metadata
    commands_quote     prefered character to use for quoting external
                       commands when running them. Defaults to double
                       quote on Windows, single tick everywhere else;
                       can be set to space to disable quoting
    check_sigs         if signatures should be verified
    colorize_debug     Term::ANSIColor attributes for debugging output
    colorize_output    boolean if Term::ANSIColor should colorize output
    colorize_print     Term::ANSIColor attributes for normal output
    colorize_warn      Term::ANSIColor attributes for warnings
    commandnumber_in_prompt
                       boolean if you want to see current command number
    cpan_home          local directory reserved for this package
    curl               path to external prg
    dontload_hash      DEPRECATED
    dontload_list      arrayref: modules in the list will not be
                       loaded by the CPAN::has_inst() routine
    ftp                path to external prg
    ftp_passive        if set, the envariable FTP_PASSIVE is set for
  downloads
    ftp_proxy          proxy host for ftp requests
    getcwd             see below
    gpg                path to external prg
    gzip               location of external program gzip
    histfile           file to maintain history between sessions
    histsize           maximum number of lines to keep in histfile
    http_proxy         proxy host for http requests
    inactivity_timeout breaks interactive Makefile.PLs or Build.PLs
                       after this many seconds inactivity. Set to 0 to
                       never break.
    index_expire       after this many days refetch index files
    inhibit_startup_message
                       if true, does not print the startup message
    keep_source_where  directory in which to keep the source (if we do)
    load_module_verbosity
                       report loading of optional modules used by CPAN.pm
    lynx               path to external prg
    make               location of external make program
    make_arg           arguments that should always be passed to 'make'
    make_install_make_command
                       the make command for running 'make install', for
                       example 'sudo make'
    make_install_arg   same as make_arg for 'make install'
    makepl_arg         arguments passed to 'perl Makefile.PL'
    mbuild_arg         arguments passed to './Build'
    mbuild_install_arg arguments passed to './Build install'
    mbuild_install_build_command
                       command to use instead of './Build' when we are
                       in the install stage, for example 'sudo ./Build'
    mbuildpl_arg       arguments passed to 'perl Build.PL'
    ncftp              path to external prg
```

```
  ncftpget           path to external prg
  no_proxy           don't proxy to these hosts/domains (comma separated
list)
  pager              location of external program more (or any pager)
  password           your password if you CPAN server wants one
  patch              path to external prg
  prefer_installer   legal values are MB and EUMM: if a module comes
                     with both a Makefile.PL and a Build.PL, use the
                     former (EUMM) or the latter (MB); if the module
                     comes with only one of the two, that one will be
                     used in any case
  prerequisites_policy
                     what to do if you are missing module prerequisites
                     ('follow' automatically, 'ask' me, or 'ignore')
  prefs_dir          local directory to store per-distro build options
  proxy_user         username for accessing an authenticating proxy
  proxy_pass         password for accessing an authenticating proxy
  randomize_urllist  add some randomness to the sequence of the urllist
  scan_cache         controls scanning of cache ('atstart' or 'never')
  shell              your favorite shell
  show_unparsable_versions
                     boolean if r command tells which modules are
versionless
  show_upload_date   boolean if commands should try to determine upload
date
  show_zero_versions boolean if r command tells for which modules
$version==0
  tar                location of external program tar
  tar_verbosity      verbosity level for the tar command
  term_is_latin      deprecated: if true Unicode is translated to
ISO-8859-1
                     (and nonsense for characters outside latin range)
  term_ornaments     boolean to turn ReadLine ornamenting on/off
  test_report        email test reports (if CPAN::Reporter is installed)
  unzip              location of external program unzip
  urllist            arrayref to nearby CPAN sites (or equivalent
locations)
  use_sqlite         use CPAN::SQLite for metadata storage (fast and lean)
  username           your username if you CPAN server wants one
  wait_list          arrayref to a wait server to try (See CPAN::WAIT)
  wget               path to external prg
  yaml_load_code     enable YAML code deserialisation
  yaml_module        which module to use to read/write YAML files
```

You can set and query each of these options interactively in the cpan shell with the `o conf` or the `o conf init` command as specified below.

`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.

interactive editing: o conf init [MATCH|LIST]

Runs an interactive configuration dialog for matching variables. Without argument runs the dialog over all supported config variables. To specify a MATCH the argument must be enclosed by slashes.

Examples:

```
o conf init ftp_passive ftp_proxy
o conf init /color/
```

Note: this method of setting config variables often provides more explanation about the functioning of a variable than the manpage.

CPAN::anycwd($path): Note on config variable getcwd

CPAN.pm changes the current working directory often and needs to determine its own current working directory. Per default it uses Cwd::cwd but if this doesn't work on your system for some reason, alternatives can be configured according to the following table:

cwd

Calls Cwd::cwd

getcwd

Calls Cwd::getcwd

fastcwd

Calls Cwd::fastcwd

backtickcwd

Calls the external command cwd.

Note on the format of the urllist parameter

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/
```

The 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.

Maintaining the urllist parameter

If you have YAML.pm (or some other YAML module configured in `yaml_module`) installed, CPAN.pm collects a few statistical data about recent downloads. You can view the statistics with the `hosts` command or inspect them directly by looking into the `FTPstats.yml` file in your `cpan_home` directory.

To get some interesting statistics it is recommended to set the `randomize_urllist` parameter that introduces some amount of randomness into the URL selection.

The requires and build_requires dependency declarations

Since CPAN.pm version 1.88_51 modules declared as `build_requires` by a distribution are treated differently depending on the config variable `build_requires_install_policy`. By setting `build_requires_install_policy` to no such a module is not being installed. It is only built and tested and then kept in the list of tested but uninstalled modules. As such it is available during the build of the dependent module by integrating the path to the `blib/arch` and `blib/lib` directories in the environment variable PERL5LIB. If `build_requires_install_policy` is set ti `yes`, then both modules declared as `requires` and those declared as `build_requires` are treated alike. By setting to `ask/yes` or `ask/no`, CPAN.pm asks the user and sets the default accordingly.

Configuration for individual distributions (Distroprefs)

(**Note:** This feature has been introduced in CPAN.pm 1.8854 and is still considered beta quality)

Distributions on the CPAN usually behave according to what we call the CPAN mantra. Or since the event of Module::Build we should talk about two mantras:

```
perl Makefile.PL     perl Build.PL
make                 ./Build
make test            ./Build test
make install         ./Build install
```

But some modules cannot be built with this mantra. They try to get some extra data from the user via the environment, extra arguments or interactively thus disturbing the installation of large bundles like Phalanx100 or modules with many dependencies like Plagger.

The distroprefs system of `CPAN.pm` addresses this problem by allowing the user to specify extra informations and recipes in YAML files to either

- pass additional arguments to one of the four commands,
- set environment variables
- instantiate an Expect object that reads from the console, waits for some regular expressions and enters some answers
- temporarily override assorted `CPAN.pm` configuration variables
- specify dependencies that the original maintainer forgot to specify
- disable the installation of an object altogether

See the YAML and Data::Dumper files that come with the `CPAN.pm` distribution in the `distroprefs/` directory for examples.

Filenames

The YAML files themselves must have the `.yml` extension, all other files are ignored (for two exceptions see *Fallback Data::Dumper and Storable* below). The containing directory can be specified in `CPAN.pm` in the `prefs_dir` config variable. Try `o conf init prefs_dir` in the CPAN shell to set and activate the distroprefs system.

Every YAML file may contain arbitrary documents according to the YAML specification and every single document is treated as an entity that can specify the treatment of a single distribution.

The names of the files can be picked freely, `CPAN.pm` always reads all files (in alphabetical order) and takes the key `match` (see below in *Language Specs*) as a hashref containing match criteria that determine if the current distribution matches the YAML document or not.

Fallback Data::Dumper and Storable

If neither your configured `yaml_module` nor YAML.pm is installed CPAN.pm falls back to using Data::Dumper and Storable and looks for files with the extensions `.dd` or `.st` in the `prefs_dir` directory. These files are expected to contain one or more hashrefs. For Data::Dumper generated files, this is expected to be done with by defining `$VAR1`, `$VAR2`, etc. The YAML shell would produce these with the command

```
ysh < somefile.yml > somefile.dd
```

For Storable files the rule is that they must be constructed such that `Storable::retrieve(file)` returns an array reference and the array elements represent one distropref object each. The conversion from YAML would look like so:

```
perl -MYAML=LoadFile -MStorable=nstore -e '
    @y=LoadFile(shift);
    nstore(\@y, shift)' somefile.yml somefile.st
```

In bootstrapping situations it is usually sufficient to translate only a few YAML files to Data::Dumper for the crucial modules like `YAML::Syck`, `YAML.pm` and `Expect.pm`. If you prefer Storable over Data::Dumper, remember to pull out a Storable version that writes an older format than all the other Storable versions that will need to read them.

Blueprint

The following example contains all supported keywords and structures with the exception of `eexpect` which can be used instead of `expect`.

```
---
comment: "Demo"
match:
  module: "Dancing::Queen"
  distribution: "^CHACHACHA/Dancing-"
  perl: "/usr/local/cariba-perl/bin/perl"
  perlconfig:
    archname: "freebsd"
disabled: 1
cpanconfig:
  make: gmake
pl:
  args:
    - "--somearg=specialcase"

  env: {}
```

```
    expect:
      - "Which is your favorite fruit"
      - "apple\n"

  make:
    args:
      - all
      - extra-all

    env: {}

    expect: []

    commendline: "echo SKIPPING make"

  test:
    args: []

    env: {}

    expect: []

  install:
    args: []

    env:
      WANT_TO_INSTALL: YES

    expect:
      - "Do you really want to install"
      - "y\n"

  patches:
    - "ABCDE/Fedcba-3.14-ABCDE-01.patch"

  depends:
    configure_requires:
      LWP: 5.8
    build_requires:
      Test::Exception: 0.25
    requires:
      Spiffy: 0.30
```

Language Specs

Every YAML document represents a single hash reference. The valid keys in this hash are as follows:

comment [scalar]

A comment

cpanconfig [hash]

Temporarily override assorted `CPAN.pm` configuration variables.

Supported are: `build_requires_install_policy`, `check_sigs`, `make`, `make_install_make_command`, `prefer_installer`, `test_report`. Please report as a bug when you need another one supported.

depends [hash] *** EXPERIMENTAL FEATURE ***

All three types, namely `configure_requires`, `build_requires`, and `requires` are supported in the way specified in the META.yml specification. The current implementation *merges* the specified dependencies with those declared by the package maintainer. In a future implementation this may be changed to override the original declaration.

disabled [boolean]

Specifies that this distribution shall not be processed at all.

goto [string]

The canonical name of a delegate distribution that shall be installed instead. Useful when a new version, although it tests OK itself, breaks something else or a developer release or a fork is already uploaded that is better than the last released version.

install [hash]

Processing instructions for the `make install` or `./Build install` phase of the CPAN mantra. See below under *Processiong Instructions*.

make [hash]

Processing instructions for the `make` or `./Build` phase of the CPAN mantra. See below under *Processiong Instructions*.

match [hash]

A hashref with one or more of the keys `distribution`, `modules`, `perl`, and `perlconfig` that specify if a document is targeted at a specific CPAN distribution or installation.

The corresponding values are interpreted as regular expressions. The `distribution` related one will be matched against the canonical distribution name, e.g. "AUTHOR/Foo-Bar-3.14.tar.gz".

The `module` related one will be matched against *all* modules contained in the distribution until one module matches.

The `perl` related one will be matched against `$^X` (but with the absolute path).

The value associated with `perlconfig` is itself a hashref that is matched against corresponding values in the `%Config::Config` hash living in the `Config.pm` module.

If more than one restriction of `module`, `distribution`, and `perl` is specified, the results of the separately computed match values must all match. If this is the case then the hashref represented by the YAML document is returned as the preference structure for the current distribution.

patches [array]

An array of patches on CPAN or on the local disk to be applied in order via the external patch program. If the value for the `-p` parameter is `0` or `1` is determined by reading the patch beforehand.

Note: if the `applypatch` program is installed and `CPAN::Config` knows about it **and** a patch is written by the `makepatch` program, then `CPAN.pm` lets `applypatch` apply the patch. Both `makepatch` and `applypatch` are available from CPAN in the `JV/makepatch-*` distribution.

pl [hash]

Processing instructions for the `perl Makefile.PL` or `perl Build.PL` phase of the CPAN mantra. See below under *Processiong Instructions*.

test [hash]

Processing instructions for the `make test` or `./Build test` phase of the CPAN mantra. See below under *Processiong Instructions*.

Processing Instructions

args [array]

Arguments to be added to the command line

commandline

A full commandline that will be executed as it stands by a system call. During the execution the environment variable PERL will is set to $^X (but with an absolute path). If `commandline` is specified, the content of `args` is not used.

eexpect [hash]

Extended `expect`. This is a hash reference with four allowed keys, `mode`, `timeout`, `reuse`, and `talk`.

`mode` may have the values `deterministic` for the case where all questions come in the order written down and `anyorder` for the case where the questions may come in any order. The default mode is `deterministic`.

`timeout` denotes a timeout in seconds. Floating point timeouts are OK. In the case of a `mode=deterministic` the timeout denotes the timeout per question, in the case of `mode=anyorder` it denotes the timeout per byte received from the stream or questions.

`talk` is a reference to an array that contains alternating questions and answers. Questions are regular expressions and answers are literal strings. The Expect module will then watch the stream coming from the execution of the external program (`perl Makefile.PL`, `perl Build.PL`, `make`, etc.).

In the case of `mode=deterministic` the CPAN.pm will inject the according answer as soon as the stream matches the regular expression.

In the case of `mode=anyorder` CPAN.pm will answer a question as soon as the timeout is reached for the next byte in the input stream. In this mode you can use the `reuse` parameter to decide what shall happen with a question-answer pair after it has been used. In the default case (reuse=0) it is removed from the array, so it cannot be used again accidentally. In this case, if you want to answer the question `Do you really want to do that` several times, then it must be included in the array at least as often as you want this answer to be given. Setting the parameter `reuse` to 1 makes this repetition unnecessary.

env [hash]

Environment variables to be set during the command

expect [array]

`expect: <array>` is a short notation for

```
eexpect:
  mode: deterministic
  timeout: 15
  talk: <array>
```

Schema verification with Kwalify

If you have the `Kwalify` module installed (which is part of the Bundle::CPANxxl), then all your distroprefs files are checked for syntactical correctness.

Example Distroprefs Files

`CPAN.pm` comes with a collection of example YAML files. Note that these are really just examples and should not be used without care because they cannot fit everybody's purpose. After all the authors of

the packages that ask questions had a need to ask, so you should watch their questions and adjust the examples to your environment and your needs. You have beend warned:-)

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(...)`). Before calling low-level commands it makes sense to initialize components of CPAN you need, e.g.:

```
CPAN::HandleConfig->load;
CPAN::Shell::setup_output;
CPAN::Index->reload;
```

High-level commands do such initializations automatically.

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 for distributions. Note: it does not expand to CPAN::Author objects.

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 $mod (qw(Net::FTP Digest::SHA Data::Dumper)) {
    CPAN::Shell->install($mod);
}

# 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

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 it normally would have refused to do. 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. See also the section above on the `force` and the `fforce` pragma.

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::author

Returns the CPAN::Author object of the maintainer who uploaded this distribution

CPAN::Distribution::pretty_id()

Returns a string of the form "AUTHORID/TARBALL", where AUTHORID is the author's PAUSE ID and TARBALL is the distribution filename.

CPAN::Distribution::base_id()

Returns the distribution filename without any archive suffix. E.g "Foo-Bar-0.01"

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 it normally would have refused to do. 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. See also the section above on the `force` and the `fforce` pragma.

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.

This install method has only the power to install the distribution if there are no dependencies in the way. To install an object and all of its dependencies, use CPAN::Shell->install.

Note that install() gives no meaningful return value. See uptodate().

CPAN::Distribution::install_tested()

Install all the distributions that have been tested sucessfully but not yet installed. See also `is_tested`.

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::is_tested()

List all the distributions that have been tested sucessfully but not yet installed. See also `install_tested`.

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` or `perl Build.PL` and `make` there.

CPAN::Distribution::perldoc()

Downloads the pod documentation of the file associated with a distribution (in html format) and runs it through the external command lynx specified in `$CPAN::Config-{lynx}>`. If lynx isn't available, it converts it to plain text with external command html2text and runs it through the

pager specified in $CPAN::Config->{pager}>

CPAN::Distribution::prefs()

Returns the hash reference from the first matching YAML file that the user has deposited in the prefs_dir/ directory. The first succeeding match wins. The files in the prefs_dir/ are processed alphabetically and the canonical distroname (e.g. AUTHOR/Foo-Bar-3.14.tar.gz) is matched against the regular expressions stored in the $root->{match}{distribution} attribute value. Additionally all module names contained in a distribution are matched agains the regular expressions in the $root->{match}{module} attribute value. The two match values are ANDed together. Each of the two attributes are optional.

CPAN::Distribution::prereq_pm()

Returns the hash reference that has been announced by a distribution as the the requires and build_requires elements. These can be declared either by the META.yml (if authoritative) or can be deposited after the run of Build.PL in the file ./_build/prereqs or after the run of Makfile.PL written as the PREREQ_PM hash in a comment in the produced Makefile. *Note*: this method only works 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::reports()

Downloads report data for this distribution from cpantesters.perl.org and displays a subset of them.

CPAN::Distribution::read_yaml()

Returns the content of the META.yml of this distro as a hashref. Note: works only after an attempt has been made to make the distribution. Returns undef otherwise. Also returns undef if the content of META.yml is not authoritative. (The rules about what exactly makes the content authoritative are still in flux.)

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 not been read for 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 in four columns: The first column contains the word Module, the second column consists of one character: an equals sign if this module is

already installed and uptodate, a less-than sign if this module is installed but can be upgraded, and a space if the module is not installed. The third column is the name of the module and the fourth column gives maintainer or distribution information.

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::distribution()

Returns the CPAN::Distribution object that contains the current version of this module.

CPAN::Module::dslip_status()

Returns a hash reference. The keys of the hash are the letters D, S, L, I, and <P>, for development status, support level, language, interface and public licence respectively. The data for the DSLIP status are collected by pause.perl.org when authors register their namespaces. The values of the 5 hash elements are one-character words whose meaning is described in the table below. There are also 5 hash elements DV, SV, LV, IV, and <PV> that carry a more verbose value of the 5 status variables.

Where the 'DSLIP' characters have the following meanings:

```
  D - Development Stage  (Note: *NO IMPLIED TIMESCALES*):
    i   - Idea, listed to gain consensus or as a placeholder
    c   - under construction but pre-alpha (not yet released)
    a/b - Alpha/Beta testing
    R   - Released
    M   - Mature (no rigorous definition)
    S   - Standard, supplied with Perl 5

  S - Support Level:
    m   - Mailing-list
    d   - Developer
    u   - Usenet newsgroup comp.lang.perl.modules
    n   - None known, try comp.lang.perl.modules
    a   - abandoned; volunteers welcome to take over maintainance

  L - Language Used:
    p   - Perl-only, no compiler needed, should be platform
  independent
    c   - C and perl, a C compiler will be needed
    h   - Hybrid, written in perl with optional C code, no compiler
  needed
```

```
    +   - C++ and perl, a C++ compiler will be needed
    o   - perl and another language other than C or C++

  I - Interface Style
    f   - plain Functions, no references used
    h   - hybrid, object and function interfaces available
    n   - no interface at all (huh?)
    r   - some use of unblessed References or ties
    O   - Object oriented using blessed references and/or inheritance

  P - Public License
    p   - Standard-Perl: user may choose between GPL and Artistic
    g   - GPL: GNU General Public License
    l   - LGPL: "GNU Lesser General Public License" (previously known
as
          "GNU Library General Public License")
    b   - BSD: The BSD License
    a   - Artistic license alone
    2   - Artistic license 2.0 or later
    o   - open source: appoved by www.opensource.org
    d   - allows distribution without restrictions
    r   - restricted distribtion
    n   - no license at all
```

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

Forces CPAN to perform a task that it normally would have refused to do. 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. See also the section above on the `force` and the `fforce` pragma.

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::available_file()

Returns the filename of the module found in PERL5LIB or @INC. The first file found is reported. The advantage of this method over `inst_file` is that modules that have been tested but not yet installed are included because PERL5LIB keeps track of tested modules.

CPAN::Module::inst_version()

Returns the version number of the installed module in readable format.

CPAN::Module::available_version()

Returns the version number of the available 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::perldoc()

Runs a `perldoc` on this module.

CPAN::Module::readme()

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

CPAN::Module::reports()

Calls the reports() method on the associated distribution object.

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.

UTILITIES

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` or `Build.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 debugging the code of CPAN.pm itself in interactive mode some more or less useful debugging aid can be turned on for most packages within CPAN.pm with one of

o debug package...

  sets debug mode for packages.

o debug -package...

  unsets debug mode for packages.

o debug all

  turns debugging on for all packages.

o debug number

which sets the debugging packages directly. Note that `o debug 0` turns debugging off.

What seems quite a successful strategy is the combination of `reload cpan` and the debugging switches. Add a new debug statement while running in the shell and then issue a `reload cpan` and see the new debugging messages immediately without losing the current context.

o `debug` without an argument lists the valid package names and the current set of packages in debugging mode. o `debug` has built-in completion support.

For debugging of CPAN data there is the `dump` command which takes the same arguments as make/test/install and outputs each object's Data::Dumper dump. If an argument looks like a perl variable and contains one of $, @ or %, it is eval()ed and fed to Data::Dumper directly.

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.

Basic Utilities for Programmers

has_inst($module)

Returns true if the module is installed. Used to load all modules into the running CPAN.pm which are considered optional. The config variable `dontload_list` can be used to intercept the `has_inst()` call such that an optional module is not loaded despite being available. For example the following command will prevent that `YAML.pm` is being loaded:

```
cpan> o conf dontload_list push YAML
```

See the source for details.

has_usable($module)

Returns true if the module is installed and is in a usable state. Only useful for a handful of modules that are used internally. See the source for details.

instance($module)

The constructor for all the singletons used to represent modules, distributions, authors and bundles. If the object already exists, this method returns the object, otherwise it calls the constructor.

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. But we try to make it easy to add security on demand:

Cryptographically signed modules

Since release 1.77 CPAN.pm has been able to verify cryptographically signed module distributions using Module::Signature. The CPAN modules can be signed by their authors, thus giving more security. The simple unsigned MD5 checksums that were used before by CPAN protect mainly against accidental file corruption.

You will need to have Module::Signature installed, which in turn requires that you have at least one of Crypt::OpenPGP module or the command-line *gpg* tool installed.

You will also need to be able to connect over the Internet to the public keyservers, like pgp.mit.edu, and their port 11731 (the HKP protocol).

The configuration parameter check_sigs is there to turn signature checking on or off.

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.

ENVIRONMENT

When the CPAN shell enters a subshell via the look command, it sets the environment CPAN_SHELL_LEVEL to 1 or increments it if it is already set.

When CPAN runs, it sets the environment variable PERL5_CPAN_IS_RUNNING to the ID of the running process. It also sets PERL5_CPANPLUS_IS_RUNNING to prevent runaway processes which could happen with older versions of Module::Install.

When running `perl Makefile.PL`, the environment variable `PERL5_CPAN_IS_EXECUTING` is set to the full path of the `Makefile.PL` that is being executed. This prevents runaway processes with newer versions of Module::Install.

When the config variable ftp_passive is set, all downloads will be run with the environment variable FTP_PASSIVE set to this value. This is in general a good idea as it influences both Net::FTP and LWP based connections. The same effect can be achieved by starting the cpan shell with this environment variable set. For Net::FTP alone, one can also always set passive mode by running libnetcfg.

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 information 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 usually need to set the environment variable `FTP_PASSIVE` or the config variable ftp_passive to a true value.

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?

First of all, you will want to use your own configuration, not the one that your root user installed. If you do not have permission to write in the cpan directory that root has configured, you will be asked if you want to create your own config. Answering "yes" will bring you into CPAN's configuration stage, using the system config for all defaults except things that have to do with CPAN's work directory, saving your choices to your MyConfig.pm file.

You can also manually initiate this process with the following command:

```
% perl -MCPAN -e 'mkmyconfig'
```

or by running

```
mkmyconfig
```

from the CPAN shell.

You will most probably also want to configure something like this:

```
  o conf makepl_arg "LIB=~/myperl/lib \
                     INSTALLMAN1DIR=~/myperl/man/man1 \
                     INSTALLMAN3DIR=~/myperl/man/man3 \
                     INSTALLSCRIPT=~/myperl/bin \
                     INSTALLBIN=~/myperl/bin"
```

and then (oh joy) the equivalent command for Module::Build. That would be

```
  o conf mbuildpl_arg "--lib=~/myperl/lib \
                     --installman1dir=~/myperl/man/man1 \
                     --installman3dir=~/myperl/man/man3 \
                     --installscript=~/myperl/bin \
                     --installbin=~/myperl/bin"
```

You can make this setting permanent like all `o conf` settings with `o conf commit` or by setting `auto_commit` beforehand.

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.

While we're speaking about $ENV{HOME}, it might be worth mentioning, that for Windows we use the File::HomeDir module that provides an equivalent to the concept of the home directory on Unix.

Another thing you should bear in mind is that the UNINST parameter can be dangerous when you are installing into a private area because you might accidentally remove modules that other people depend on that are not using the private area.

6)

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

Have a look at the `look` (!) command.

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 META.yml file or 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 IF all modules declare the prerequisites correctly with the PREREQ_PM attribute to MakeMaker or the `requires` stanza of Module::Build. For bundles which fail and you need to install often, it is recommended to sort the Bundle definition file manually.

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 an error message about things in my /etc/inputrc (or ~/.inputrc) file.

These are readline issues and can only be fixed by studying readline configuration on your architecture and adjusting the referenced file accordingly. Please make a backup of the /etc/inputrc or ~/.inputrc and edit them. Quite often harmless changes like uppercasing or lowercasing some arguments solves the problem.

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> o conf term_is_latin 1
```

If other charset support is needed, please file a bugreport against CPAN.pm at rt.cpan.org and

describe your needs. Maybe we can extend the support or maybe UTF-8 terminals become widely available.

Note: this config variable is deprecated and will be removed in a future version of CPAN.pm. It will be replaced with the conventions around the family of $LANG and $LC_* environment variables.

11)

When an install fails for some reason and then I correct the error condition and retry, CPAN.pm refuses to install the module, saying `Already tried without success`.

Use the force pragma like so

```
force install Foo::Bar
```

Or you can use

```
look Foo::Bar
```

and then 'make install' directly in the subshell.

12)

How do I install a "DEVELOPER RELEASE" of a module?

By default, CPAN will install the latest non-developer release of a module. If you want to install a dev release, you have to specify the partial path starting with the author id to the tarball you wish to install, like so:

```
cpan> install KWILLIAMS/Module-Build-0.27_07.tar.gz
```

Note that you can use the `ls` command to get this path listed.

13)

How do I install a module and all its dependencies from the commandline, without being prompted for anything, despite my CPAN configuration (or lack thereof)?

CPAN uses ExtUtils::MakeMaker's prompt() function to ask its questions, so if you set the PERL_MM_USE_DEFAULT environment variable, you shouldn't be asked any questions at all (assuming the modules you are installing are nice about obeying that variable as well):

```
% PERL_MM_USE_DEFAULT=1 perl -MCPAN -e 'install My::Module'
```

14)

How do I create a Module::Build based Build.PL derived from an ExtUtils::MakeMaker focused Makefile.PL?

http://search.cpan.org/search?query=Module::Build::Convert

http://www.refcnt.org/papers/module-build-convert

15)

What's the best CPAN site for me?

The urllist config parameter is yours. You can add and remove sites at will. You should find out which sites have the best uptodateness, bandwidth, reliability, etc. and are topologically close to you. Some people prefer fast downloads, others uptodateness, others reliability. You decide which to try in which order.

Henk P. Penning maintains a site that collects data about CPAN sites:

```
http://www.cs.uu.nl/people/henkp/mirmon/cpan.html
```

16)

Why do I get asked the same questions every time I start the shell?

You can make your configuration changes permanent by calling the command `o conf commit`. Alternatively set the `auto_commit` variable to true by running `o conf init auto_commit` and answering the following question with yes.

COMPATIBILITY

OLD PERL VERSIONS

CPAN.pm is regularly tested to run under 5.004, 5.005, and assorted newer versions. It is getting more and more difficult to get the minimal prerequisites working on older perls. It is close to impossible to get the whole Bundle::CPAN working there. If you're in the position to have only these old versions, be advised that CPAN is designed to work fine without the Bundle::CPAN installed.

To get things going, note that GBARR/Scalar-List-Utils-1.18.tar.gz is compatible with ancient perls and that File::Temp is listed as a prerequisite but CPAN has reasonable workarounds if it is missing.

CPANPLUS

This module and its competitor, the CPANPLUS module, are both much cooler than the other. CPAN.pm is older. CPANPLUS was designed to be more modular but it was never tried to make it compatible with CPAN.pm.

SECURITY ADVICE

This software enables you to upgrade software on your computer and so is inherently dangerous because the newly installed software may contain bugs and may alter the way your computer works or even make it unusable. Please consider backing up your data before every upgrade.

BUGS

Please report bugs via *http://rt.cpan.org/*

Before submitting a bug, please make sure that the traditional method of building a Perl module package from a shell by following the installation instructions of that package still works in your environment.

AUTHOR

Andreas Koenig <andk@cpan.org>

LICENSE

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See *http://www.perl.com/perl/misc/Artistic.html*

TRANSLATIONS

Kawai,Takanori provides a Japanese translation of this manpage at *http://homepage3.nifty.com/hippo2000/perltips/CPAN.htm*

SEE ALSO

cpan, *CPAN::Nox*, *CPAN::Version*