NAME

Module::Build::Cookbook - Examples of Module::Build Usage

DESCRIPTION

`Module::Build` isn't conceptually very complicated, but examples are always helpful. The following recipes should help developers and/or installers put together the pieces from the other parts of the documentation.

BASIC RECIPES

Installing modules that use Module::Build

In most cases, you can just issue the following commands:

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

There's nothing complicated here - first you're running a script called *Build.PL*, then you're running a (newly-generated) script called *Build* and passing it various arguments.

The exact commands may vary a bit depending on how you invoke perl scripts on your system. For instance, if you have multiple versions of perl installed, you can install to one particular perl's library directories like so:

```
/usr/bin/perl5.8.1 Build.PL
./Build
./Build test
./Build install
```

If you're on Windows where the current directory is always searched first for scripts, you'll probably do something like this:

```
perl Build.PL
Build
Build test
Build install
```

On the old Mac OS (version 9 or lower) using MacPerl, you can double-click on the *Build.PL* script to create the *Build* script, then double-click on the *Build* script to run its `build`, `test`, and `install` actions.

The *Build* script knows what perl was used to run *Build.PL*, so you don't need to re-invoke the *Build* script with the complete perl path each time. If you invoke it with the *wrong* perl path, you'll get a warning or a fatal error.

Modifying Config.pm values

`Module::Build` relies heavily on various values from perl's `Config.pm` to do its work. For example, default installation paths are given by `installsitelib` and `installvendorman3dir` and friends, C linker & compiler settings are given by `ld`, `lddlflags`, `cc`, `ccflags`, and so on. *If you're pretty sure you know what you're doing*, you can tell `Module::Build` to pretend there are different values in *Config.pm* than what's really there, by passing arguments for the `--config` parameter on the command line:

```
perl Build.PL --config cc=gcc --config ld=gcc
```

Inside the `Build.PL` script the same thing can be accomplished by passing values for the `config`

parameter to `new()`:

```
my $build = Module::Build->new
  (
   ...
   config => { cc => 'gcc', ld => 'gcc' },
   ...
  );
```

In custom build code, the same thing can be accomplished by calling the *"config" in Module::Build* method:

```
$build->config( cc => 'gcc' );     # Set
$build->config( ld => 'gcc' );     # Set
...
my $linker = $build->config('ld'); # Get
```

Installing modules using the programmatic interface

If you need to build, test, and/or install modules from within some other perl code (as opposed to having the user type installation commands at the shell), you can use the programmatic interface. Create a Module::Build object (or an object of a custom Module::Build subclass) and then invoke its `dispatch()` method to run various actions.

```
my $build = Module::Build->new
  (
   module_name => 'Foo::Bar',
   license     => 'perl',
   requires    => { 'Some::Module'   => '1.23' },
  );
$build->dispatch('build');
$build->dispatch('test', verbose => 1);
$build->dispatch('install');
```

The first argument to `dispatch()` is the name of the action, and any following arguments are named parameters.

This is the interface we use to test Module::Build itself in the regression tests.

Installing to a temporary directory

To create packages for package managers like RedHat's `rpm` or Debian's `deb`, you may need to install to a temporary directory first and then create the package from that temporary installation. To do this, specify the `destdir` parameter to the `install` action:

```
./Build install --destdir /tmp/my-package-1.003
```

This essentially just prepends all the installation paths with the */tmp/my-package-1.003* directory.

Installing to a non-standard directory

To install to a non-standard directory (for example, if you don't have permission to install in the system-wide directories), you can use the `install_base` or `prefix` parameters:

```
./Build install --install_base /foo/bar
```

See *"INSTALL PATHS" in Module::Build* for a much more complete discussion of how installation paths are determined.

Installing in the same location as ExtUtils::MakeMaker

With the introduction of `--prefix` in Module::Build 0.28 and `INSTALL_BASE` in ExtUtils::MakeMaker 6.31 its easy to get them both to install to the same locations.

First, ensure you have at least version 0.28 of Module::Build installed and 6.31 of ExtUtils::MakeMaker. Prior versions have differing (and in some cases quite strange) installation behaviors.

The following installation flags are equivalent between ExtUtils::MakeMaker and Module::Build.

```
MakeMaker             Module::Build
PREFIX=...            --prefix ...
INSTALL_BASE=...      --install_base ...
DESTDIR=...           --destdir ...
LIB=...               --install_path lib=...
INSTALLDIRS=...       --installdirs ...
INSTALLDIRS=perl      --installdirs core
UNINST=...            --uninst ...
INC=...               --extra_compiler_flags ...
POLLUTE=1             --extra_compiler_flags -DPERL_POLLUTE
```

For example, if you are currently installing MakeMaker modules with this command:

```
perl Makefile.PL PREFIX=~
make test
make install UNINST=1
```

You can install into the same location with Module::Build using this:

```
perl Build.PL --prefix ~
./Build test
./Build install --uninst 1
```

prefix vs install_base

The behavior of `prefix` is complicated and depends on how your Perl is configured. The resulting installation locations will vary from machine to machine and even different installations of Perl on the same machine. Because of this, it's difficult to document where `prefix` will place your modules.

In contrast, `install_base` has predictable, easy to explain installation locations. Now that Module::Build and MakeMaker both have `install_base` there is little reason to use `prefix` other than to preserve your existing installation locations. If you are starting a fresh Perl installation we encourage you to use `install_base`. If you have an existing installation installed via `prefix`, consider moving it to an installation structure matching `install_base` and using that instead.

Running a single test file

`Module::Build` supports running a single test, which enables you to track down errors more quickly. Use the following format:

```
./Build test --test_files t/mytest.t
```

In addition, you may want to run the test in verbose mode to get more informative output:

```
./Build test --test_files t/mytest.t --verbose 1
```

I run this so frequently that I define the following shell alias:

```
alias t './Build test --verbose 1 --test_files'
```

So then I can just execute `t t/mytest.t` to run a single test.

ADVANCED RECIPES

Making a CPAN.pm-compatible distribution

New versions of CPAN.pm understand how to use a *Build.PL* script, but old versions don't. If authors want to help users who have old versions, some form of *Makefile.PL* should be supplied. The easiest way to accomplish this is to use the `create_makefile_pl` parameter to `Module::Build->new()` in the `Build.PL` script, which can create various flavors of *Makefile.PL* during the `dist` action.

As a best practice, we recommend using the "traditional" style of *Makefile.PL* unless your distribution has needs that can't be accomplished that way.

The `Module::Build::Compat` module, which is part of `Module::Build`'s distribution, is responsible for creating these *Makefile.PL*s. Please see *Module::Build::Compat* for the details.

Changing the order of the build process

The `build_elements` property specifies the steps `Module::Build` will take when building a distribution. To change the build order, change the order of the entries in that property:

```
# Process pod files first
my @e = @{$build->build_elements};
my $i = grep {$e[$_] eq 'pod'} 0..$#e;
unshift @e, splice @e, $i, 1;
```

Currently, `build_elements` has the following default value:

```
[qw( PL support pm xs pod script )]
```

Do take care when altering this property, since there may be non-obvious (and non-documented!) ordering dependencies in the `Module::Build` code.

Adding new file types to the build process

Sometimes you might have extra types of files that you want to install alongside the standard types like *.pm* and *.pod* files. For instance, you might have a *Bar.dat* file containing some data related to the `Foo::Bar` module and you'd like for it to end up as *Foo/Bar.dat* somewhere in perl's `@INC` path so `Foo::Bar` can access it easily at runtime. The following code from a sample `Build.PL` file demonstrates how to accomplish this:

```
use Module::Build;
my $build = Module::Build->new
  (
   module_name => 'Foo::Bar',
   ...other stuff here...
  );
$build->add_build_element('dat');
$build->create_build_script;
```

This will find all *.dat* files in the *lib/* directory, copy them to the *blib/lib/* directory during the `build` action, and install them during the `install` action.

If your extra files aren't located in the `lib/` directory in your distribution, you can explicitly say where they are, just as you'd do with *.pm* or *.pod* files:

```
use Module::Build;
my $build = new Module::Build
  (
   module_name => 'Foo::Bar',
```

```
    dat_files => {'some/dir/Bar.dat' => 'lib/Foo/Bar.dat'},
    ...other stuff here...
  );
$build->add_build_element('dat');
$build->create_build_script;
```

If your extra files actually need to be created on the user's machine, or if they need some other kind of special processing, you'll probably want to subclass `Module::Build` and create a special method to process them, named `process_${kind}_files()`:

```
use Module::Build;
my $class = Module::Build->subclass(code => <<'EOF');
  sub process_dat_files {
    my $self = shift;
    ... locate and process *.dat files,
    ... and create something in blib/lib/
  }
EOF
my $build = $class->new
  (
   module_name => 'Foo::Bar',
   ...other stuff here...
  );
$build->add_build_element('dat');
$build->create_build_script;
```

If your extra files don't go in *lib/* but in some other place, see *Adding new elements to the install process* for how to actually get them installed.

Please note that these examples use some capabilities of Module::Build that first appeared in version 0.26. Before that it could still be done, but the simple cases took a bit more work.

Adding new elements to the install process

By default, Module::Build creates seven subdirectories of the *blib* directory during the build process: *lib*, *arch*, *bin*, *script*, *bindoc*, *libdoc*, and *html* (some of these may be missing or empty if there's nothing to go in them). Anything copied to these directories during the build will eventually be installed during the `install` action (see *"INSTALL PATHS" in Module::Build*.

If you need to create a new custom type of installable element, e.g. `conf`, then you need to tell Module::Build where things in *blib/conf/* should be installed. To do this, use the `install_path` parameter to the `new()` method:

```
my $build = Module::Build->new
  (
   ...other stuff here...
   install_path => { conf => $installation_path }
  );
```

Or you can call the `install_path()` method later:

```
$build->install_path(conf => $installation_path);
```

The user may also specify the path on the command line:

```
perl Build.PL --install_path conf=/foo/path/etc
```

The important part, though, is that *somehow* the install path needs to be set, or else nothing in the *blib/conf/* directory will get installed, and a runtime error during the `install` action will result.

See also *Adding new file types to the build process* for how to create the stuff in *blib/conf/* in the first place.

EXAMPLES ON CPAN

Several distributions on CPAN are making good use of various features of Module::Build. They can serve as real-world examples for others.

SVN-Notify-Mirror

http://search.cpan.org/~jpeacock/SVN-Notify-Mirror/

John Peacock, author of the `SVN-Notify-Mirror` distribution, says:

1. Using `auto_features`, I check to see whether two optional modules are available - SVN::Notify::Config and Net::SSH;

2. If the S::N::Config module is loaded, I automatically generate testfiles for it during Build (using the `PL_files` property).

3. If the `ssh_feature` is available, I ask if the user wishes to perform the ssh tests (since it requires a little preliminary setup);

4. Only if the user has `ssh_feature` and answers yes to the testing, do I generate a test file.

    I'm sure I could not have handled this complexity with EU::MM, but it was very easy to do with M::B.

Modifying an action

Sometimes you might need an to have an action, say `./Build install`, do something unusual. For instance, you might need to change the ownership of a file or do something else peculiar to your application.

You can subclass `Module::Build` on the fly using the `subclass()` method and override the methods that perform the actions. You may need to read through `Module::Build::Authoring` and `Module::Build::API` to find the methods you want to override. All "action" methods are implemented by a method called "ACTION_" followed by the action's name, so here's an example of how it would work for the `install` action:

```
# Build.PL
use Module::Build;
my $class = Module::Build->subclass(
    class => "Module::Build::Custom",
    code => <<'SUBCLASS' );

sub ACTION_install {
    my $self = shift;
    # YOUR CODE HERE
    $self->SUPER::ACTION_install;
}
SUBCLASS

$class->new(
    module_name => 'Your::Module',
    # rest of the usual Module::Build parameters
)->create_build_script;
```

AUTHOR

Ken Williams <kwilliams@cpan.org>

COPYRIGHT

SEE ALSO

perl(1), *Module::Build*(3), *Module::Build::Authoring*(3), *Module::Build::API*(3)