NAME

bundle−exec − Execute a command in the context of the bundle

SYNOPSIS

bundle exec [−−keep−file−descriptors] *command*

DESCRIPTION

This command executes the command, making all gems specified in the [**Gemfile(5)**][Gemfile(5)] available to **require** in Ruby programs.

Essentially, if you would normally have run something like **rspec spec/my_spec.rb**, and you want to use the gems specified in the [**Gemfile(5)**][Gemfile(5)] and installed via bundle install(1) *bundle−install.1.html*, you should run **bundle exec rspec spec/my_spec.rb**.

Note that **bundle exec** does not require that an executable is available on your shell´s **$PATH**.

OPTIONS

−−keep−file−descriptors

Exec in Ruby 2.0 began discarding non−standard file descriptors. When this flag is passed, exec will revert to the 1.9 behaviour of passing all file descriptors to the new process.

BUNDLE INSTALL −−BINSTUBS

If you use the **−−binstubs** flag in bundle install(1) *bundle−install.1.html*, Bundler will automatically create a directory (which defaults to **app_root/bin**) containing all of the executables available from gems in the bundle.

After using **−−binstubs**, **bin/rspec spec/my_spec.rb** is identical to **bundle exec rspec spec/my_spec.rb**.

ENVIRONMENT MODIFICATIONS

bundle exec makes a number of changes to the shell environment, then executes the command you specify in full.

- make sure that it´s still possible to shell out to **bundle** from inside a command invoked by **bundle exec** (using **$BUNDLE_BIN_PATH**)
- put the directory containing executables (like **rails**, **rspec**, **rackup**) for your bundle on **$PATH**
- make sure that if bundler is invoked in the subshell, it uses the same **Gemfile** (by setting **BUNDLE_GEMFILE**)
- add **−rbundler/setup** to **$RUBYOPT**, which makes sure that Ruby programs invoked in the subshell can see the gems in the bundle

It also modifies Rubygems:

- disallow loading additional gems not in the bundle
- modify the **gem** method to be a no−op if a gem matching the requirements is in the bundle, and to raise a **Gem::LoadError** if it´s not
- Define **Gem.refresh** to be a no−op, since the source index is always frozen when using bundler, and to prevent gems from the system leaking into the environment
- Override **Gem.bin_path** to use the gems in the bundle, making system executables work
- Add all gems in the bundle into Gem.loaded_specs

Finally, **bundle exec** also implicitly modifies **Gemfile.lock** if the lockfile and the Gemfile do not match. Bundler needs the Gemfile to determine things such as a gem´s groups, **autorequire**, and platforms, etc., and that information isn´t stored in the lockfile. The Gemfile and lockfile must be synced in order to **bundle exec** successfully, so **bundle exec** updates the lockfile beforehand.

Loading

By default, when attempting to **bundle exec** to a file with a ruby shebang, Bundler will **Kernel.load** that file instead of using **Kernel.exec**. For the vast majority of cases, this is a performance improvement. In a rare few cases, this could cause some subtle side−effects (such as dependence on the exact contents of **$0** or **__FILE__**) and the optimization can be disabled by enabling the **disable_exec_load** setting.

Shelling out

Any Ruby code that opens a subshell (like **system**, backticks, or **%x{}**) will automatically use the current Bundler environment. If you need to shell out to a Ruby command that is not part of your current bundle, use the **with_clean_env** method with a block. Any subshells created inside the block will be given the environment present before Bundler was activated. For example, Homebrew commands run Ruby, but don´t work inside a bundle:

```
Bundler.with_clean_env do
  ‘brew install wget‘
end
```

Using **with_clean_env** is also necessary if you are shelling out to a different bundle. Any Bundler commands run in a subshell will inherit the current Gemfile, so commands that need to run in the context of a different bundle also need to use **with_clean_env**.

```
Bundler.with_clean_env do
  Dir.chdir "/other/bundler/project" do
    ‘bundle exec ./script‘
  end
end
```

Bundler provides convenience helpers that wrap **system** and **exec**, and they can be used like this:

```
Bundler.clean_system(´brew install wget´)
Bundler.clean_exec(´brew install wget´)
```

RUBYGEMS PLUGINS

At present, the Rubygems plugin system requires all files named **rubygems_plugin.rb** on the load path of *any* installed gem when any Ruby code requires **rubygems.rb**. This includes executables installed into the system, like **rails**, **rackup**, and **rspec**.

Since Rubygems plugins can contain arbitrary Ruby code, they commonly end up activating themselves or their dependencies.

For instance, the **gemcutter 0.5** gem depended on **json_pure**. If you had that version of gemcutter installed (even if you *also* had a newer version without this problem), Rubygems would activate **gemcutter 0.5** and **json_pure <latest>**.

If your Gemfile(5) also contained **json_pure** (or a gem with a dependency on **json_pure**), the latest version on your system might conflict with the version in your Gemfile(5), or the snapshot version in your **Gemfile.lock**.

If this happens, bundler will say:

You have already activated json_pure 1.4.6 but your Gemfile
requires json_pure 1.4.3. Consider using bundle exec.

In this situation, you almost certainly want to remove the underlying gem with the problematic gem plugin. In general, the authors of these plugins (in this case, the **gemcutter** gem) have released newer versions that are more careful in their plugins.

You can find a list of all the gems containing gem plugins by running

ruby −rrubygems −e "puts Gem.find_files(´rubygems_plugin.rb´)"

At the very least, you should remove all but the newest version of each gem plugin, and also remove all gem plugins that you aren´t using (**gem uninstall gem_name**).