Borg User Manual

for version 2.0.0 (v2.0.0-64-g738f749+1)

Jonas Bernoulli

Table of Contents

1 Introduction

The Borg assimilate Emacs packages as Git submodules.

Borg is a bare-bones package manager for Emacs packages. It provides only a few essential features and should be combined with other tools such as Magit, `epkg`, `use-package`, and `auto-compile`.

Borg assimilates packages into the `~/.emacs.d` repository as Git submodules. An assimilated package is called a drone and a borg-based `~/.emacs.d` repository is called a collective.

It is possible to clone a package repository without assimilating it. A cloned package is called a clone.

To learn more about this project, also read the blog post[1] in which it was announced.

[1] `https://emacsair.me/2016/05/17/assimilate-emacs-packages-as-git-submodules`.

2 Installation

To get started clone the repository of the `emacs.g` collective. Currently this is the only available collective and, for the time being this manual assume that you use that.

This collective already assimilated a few drones in addition to `borg` itself, namely `magit`, `epkg`, `use-package`, `auto-compile`, `git-modes`, `diff-hl`, and their dependencies. These drones are not required by `borg` but their use is highly recommended.

Instructions on how to bootstrap a configuration without basing it on the `emacs.g` collective will be added in the near future.

Clone the `emacs.g` repository to either `~/.emacs.d`, or for testing purposes to any other location. This repository contains a `Makefile` that imports `lib/borg/borg.mk` and defines an additional target whose purpose is to make that file and `lib/borg/borg.sh` available. Run `make bootstrap-borg` to clone the `borg` repository. That does not completely setup the `borg` repository but it makes the latest version of the mentioned files available. Now that these files are available you can run `make bootstrap` to get and configure all submodules (including the `borg` submodule) and to build all drones.

```
git clone git@github.com:emacscollective/emacs.g.git ~/.emacs.d
cd ~/.emacs.d
make bootstrap-borg
make bootstrap
```

If you cloned to somewhere else than `~/.emacs.d`, then you can use that configuration using `emacs -Q --load /path/to/emacs.g/init.elc`.

For drones whose upstreams are on Github or Gitlab the `emacs.g` collective uses the `ssh` protocol by default, which is a problem if you don't have accounts there and have not properly setup your keys. Luckily this can easily be fixed using the following right after cloning the super-repository.

```
git config --global url.https://github.com/.insteadOf git@github.com:
git config --global url.https://gitlab.com/.insteadOf git@gitlab.com:
```

During package compilation you may notice the submodules relating to those packages become dirty due to the compilation outputs not being ignored in those submodules. For this reason it is useful to ignore these outputs globally, for example in your `~/.config/git/ignore` file:

```
*.elc
*-autoloads.el
dir
```

You may discover more things that you'll want to ignore this way as you use `borg`.

3 Startup

The `user-init-file`, `~/.emacs.d/init.el`, has to contain a call to `borg-initialize`. It should also set `package-enable-at-startup` to `nil` unless you really want to use both `borg` and `package` at the same time.

`borg-initialize` [Function]

This function initializes assimilated drones using `borg-activate`.

To skip the activation of the drone named DRONE, temporarily disable it by setting the value of the Git variable `submodule.DRONE.disabled` to true in `~/.emacs.d/.gitmodules`.

`borg-activate` *clone* [Command]

This function activates the clone named CLONE by adding the appropriate directories to the `load-path` and to `Info-directory-list`, and by loading the autoloads file, if it exits.

Unlike `borg-initialize`, this function ignores the Git variable `submodule.DRONE.disabled` and can be used to activate clones that have not been assimilated.

4 Assimilation

A third-party package is assimilated by adding it as a submodule and, if necessary, by configuring it in `~/.emacs.d/init.el`. Built-in packages are assimilated merely by configuring them.

To begin the assimilation of a third-party package use the command `borg-assimilate`, which adds the package's repository as a submodule and attempts to build the drone.

A safer alternative is to first clone the package without assimilating it, using `borg-clone`. This gives you an opportunity to inspect the cloned package for broken or malicious code before it gets a chance to run arbitrary code. Later you can proceed with the assimilation using `borg-assimilate`, or remove the clone using `borg-remove`.

Building the drone can fail, for example due to missing dependencies. Failure to build a drone is not considered as a failure to assimilate. If a build fails, then a buffer containing information about the issue pops up. If the failure is due to unsatisfied dependencies, then assimilate those too, and then build any drone which previously couldn't be built by using the Emacs command `borg-build` or `make lib/DRONE`. Alternatively you can just rebuild everything using `make build`.

If you wish to avoid such complications, you should use the command `epkg-describe-package` before assimilating a package. Among other useful information, it also provides a dependency tree.

Once the packages have been added as submodules and the drones have been built, the assimilation is completed by creating an assimilation commit.

If you assimilate a single package, then it is recommended that you use a message similar to this:

```
Assimilate foo v1.0.0
```

Or if one or more dependencies had to be assimilated, something like:

```
Assimilate foo and dependencies

Assimilate foo v1.0.0
Assimilate bar v1.1.0
Assimilate baz v0.1.0
```

It's usually a good idea not to assimilate unrelated packages in the same commit, but something like this might make sense:

```
Assimilate ido and extensions

Assimilate flx               v0.6.1-3-gae0981b
Assimilate ido-at-point      v1.0.0
Assimilate ido-ubiquitious   v3.12-2-g7354d98
Assimilate ido-vertical-mode v0.1.6-33-gb42e422
Assimilate smex               3.0-13-g55aaebe
```

Version strings as those shown above can be obtained using `git describe --tags`, or by looking inside the "Modules" section of the Magit status buffer of the `~/.emacs.d` repository.

`borg-assimilate` *package url* **&optional** *partially* [Command]
This command assimilates the package named PACKAGE from URL.

If `epkg` is available, then only the name of the package is read in the minibuffer and the url stored in the Epkg database is used. If `epkg` is unavailable, the package is not in the database, or if a prefix argument is used, then the url too is read in the minibuffer.

If a negative prefix argument is used, then the submodule is added but the build and activation steps are skipped. This is useful when assimilating a package that require special build steps. After configuring the build steps use `borg-build` to complete the assimilation.

`borg-clone` *package url* [Command]
This command clones the package named PACKAGE from URL, without assimilating it. This is useful when you want to inspect the package before potentially executing malicious or broken code.

Interactively, when the `epkg` package is available, then the name is read in the minibuffer and the url stored in the Epkg database is used. If `epkg` is unavailable, the package is unknown, or when a prefix argument is used, then the url is also read in the minibuffer.

`borg-remove` *clone* [Command]
This command removes the cloned or assimilated package named CLONE, by removing the working tree from `borg-drone-directory`, regardless of whether that repository belongs to an assimilated package or a package that has only been cloned for review using `borg-clone`. The Git directory is not removed.

`borg-build` *clone* **&optional** *activate* [Command]
This command builds the clone named CLONE. Interactively, or when optional ACTIVATE is non-nil, then also activate the drone using `borg-activate`.

`borg-update-autoloads` *clone* **&optional** *path* [Function]
This function updates the autoload file for the libraries belonging to the clone named CLONE in the directories in PATH. PATH can be omitted or contain file-names that are relative to the top-level of CLONE's repository.

`borg-byte-compile` *clone* **&optional** *path* [Function]
This function compiles the libraries for the clone named CLONE in the directories in PATH. PATH can be omitted or contain file-names that are relative to the top-level of CLONE's repository.

`borg-makeinfo` *clone* [Function]
This function generates the Info manuals and the Info index for the clone named CLONE.

`borg-batch-rebuild` **&optional** *quick* [Function]
This function rebuilds all assimilated drones in alphabetic order, except for Org which is rebuilt first. It also rebuilds `init.el` and `USER-REAL-LOGIN-NAME.el`.

This function is not intended for interactive use, but used to implement the `make` targets described in the following section.

When optional QUICK is non-nil, then do not build drones for which `submodule.DRONE.build-step` is set, assuming that those are the drones that take longer to be built.

`borg-batch-rebuild-init` [Function]

This function rebuilds `init.el` and `USER-REAL-LOGIN-NAME.el`. It is not intended for interactive use.

5 Updating drones

Borg does not provide an update command. By not doing so, it empowers you to update to exactly the commit you wish to update to, instead of to "the" new version.

To determine the drones with you *might* want to update, visit the Magit status buffer of the `~/.emacs.d` repository and press `f m` to fetch inside all submodules. After you have done so, and provided there actually are any modules with new upstream commits, a section titled "Modules unpulled from @{upstream}" appears.

Each subsection of that section represents a submodule with new upstream commits. Expanding such a subsection lists the new upstream commits. These commits can be visited by pressing `RET`, and the status buffer of a submodule can be visited by pressing `RET` while point is inside the heading of the respective submodule section. To return to the status buffer of `~/.emacs.d` press `q`.

Inside the status buffer of a submodule, you can pull the upstream changes as usual, using `F u`. If you wish you can inspect the changes before doing so. And you can also choose to check out another commit instead of the upstream `HEAD`.

Once you have "updated" to a new commit, you should also rebuild the drone using the command `borg-build`. This may fail, e.g. due to new dependencies.

Once you have resolved all issues you should create an "update commit". You can either create one commit per updated drone or you can create a single commit for all updated drones, which ever you find more appropriate. However it is recommended that you use a message similar to:

```
Update foo to v1.1.0
```

Or for multiple packages:

```
Update 2 drones

Update foo to v1.1.0
Update bar to v1.2.1
```

To update the Epkg package database use the command `epkg-update`.

6 Patching drones

By using Borg you can not only make changes to assimilated packages, you can also keep track of those patches and share them with others.

If you created some commits in a drone repository and are the maintainer of the respective package, then you can just push your changes to the "origin" remote. You don't have to do this every time you created some commits, but at important checkpoints, such as after creating a release, you should record the changes in the `~/.emacs.d` repository. To do so proceed as described in Chapter 5 [Updating drones], page 7.

But for most packages you are not the maintainer and if you create commits for such drones, then you have to create a fork and push there instead. You should configure that remote as the push-remote using `git config remote.pushDefault FORK`, or pressing `b C M-p` in Magit. After you have done that you can continue to pull from the upstream using `P u` in Magit and you can also push to your fork using `P p`.

Of course you should also occasionally record the changes in the `~/.emacs.d` repository. Additionally, and ideally when you first fork a drone, you should also record information about your personal remote in the super-repository by setting `submodule.DRONE.remote` in `~/.emacs.d/.gitmodules`.

`submodule.DRONE.remote` "*NAME URL*" [Variable]

This variable specifies an additional remote named NAME that is fetched from URL. This variable can be specified multiple times. Note that "NAME URL" is a single value and that the two parts of that value are separated by a single space.

`make bootstrap` automatically adds all remotes that are specified like this to the DRONE repository by setting `remote.NAME.url` to URL and using the standard value for `remote.NAME.fetch`.

`borg.pushDefault` = *FORK* [Variable]

This variable specifies a name used for push-remotes. Because this variable can only have one value it is recommended that you use the same name, FORK, for your personal remote in all drone repositories in which you have created patches that haven't been merged into the upstream repository (yet). A good value may be your username.

For all DRONES for which one value of `submodule.DRONE.remote` specifies a remote whose NAME matches FORK, `make bootstrap` automatically configures FORK to be used as the push-remote by setting `remote.pushDefault` to FORK.

7 Make targets

The following `make` targets are available in `~/.emacs.d/Makefile`. To use them you have to be in `~/.emacs.d` in a shell.

`make` *help* [Command]
: This target prints information about the following targets.

`make` *build* [Command]
: This target builds all drones.

  It also builds `init.el` and `USER-REAL-LOGIN-NAME.el`, if that exists. Also see `make build-init` below.

`make` *quick* [Command]
: This target builds *most* drones. Excluded are all drones for which the Git variable `submodule.DRONE.build-step` is set, assuming that those are the drones that take longer to build.

  It also builds `init.el` and `USER-REAL-LOGIN-NAME.el`, if that exists. Also see `make build-init` below.

`make` *lib/DRONE* [Command]
: This target builds the drone named DRONE.

`make` *build-init* [Command]
: This target builds `init.el` and `USER-REAL-LOGIN-NAME.el`, if that exists.

  If you publish your `~/.emacs.d` repository but would like to keep some settings private, then you can do so by putting these in a file `~/.emacs.d/FILE-NAME.el`. If FILE-NAME matches the value of the variable `user-real-login-name`, then the `init.el` of the emacs.g collective automatically loads it. The downside of this approach is that you will have to somehow synchronize that file between your machines without checking it into Git.

`make` *bootstrap* [Command]
: This target attempts to bootstrap the drones. To do so it runs `git submodule init`, `borg.sh` (which see), and `make build`.

  If an error occurs during the `borg.sh` phase, then you can just run that command again to process the remaining drones. The drone that have already been bootstrapped or that have previously failed will be skipped. If a drone cannot be cloned from any of the known remotes, then you should temporarily remove it using `git submodule deinit lib/DRONE`. When done with `borg.sh` also manually run `make build` again.

8 Variables

The values of the following variables are set at startup and should not be changed by the user.

`borg-drone-directory` [Variable]
: The value of this constant is the directory beneath which drone submodules are placed. The value is set based on the location of the `borg` library and should not be changed.

`borg-user-emacs-directory` [Variable]
: The value of this constant is the directory beneath which additional per-user Emacs-specific files are placed. The value is set based on the location of the `borg` library and should not be changed. The value is usually the same as that of `user-emacs-directory`, except when Emacs is started with `emacs -q -l /path/to/init.el`.

`borg-gitmodules-file` [Variable]
: The value of this constant is the ".gitmodules" file of the super-repository.

The values of the borg-specific Git variables have to be set in the file `~/.emacs.d/.gitmodules`. The variables `borg.pushDefault` and `submodule.DRONE.remote` are described in Chapter 6 [Patching drones], page 8.

`borg.collective` = *REMOTE* [Variable]
: This variable specifies the name used for remotes that reference a repository that has been patched by the collective. If a NAME matches REMOTE, then it is configured as the upstream of the current branch of the respective DRONE.

  If the file ".hive-maint" exists, then this variable has the same effect as "borg.pushDefault". This special case is only useful for maintainers of the collective (but not for maintainers of individual drones).

Because most repositories used to maintain Emacs packages follow some common-sense conventions, Borg usually does not have to be told how to build a given drone. Building is done using `borg-build`, which in turn usually does its work using `borg-update-autoloads`, `borg-byte-compile`, and `borg-makeinfo`.

However some packages don't follow the conventions either because they are too complex to do so, or for the sake of doing it differently. But in either case resistance is futile; by using the following variables you can tell Borg how to build such packages.

`submodule.DRONE.build-step` *COMMAND* [Variable]
: By default drones are built using the lisp functions `borg-update-autoloads`, `borg-byte-compile`, and `borg-makeinfo`, but if this variable has one or more values, then DRONE is built using these COMMANDs **instead**.

  Each COMMAND can be one of the default steps, an S-expression, or a shell command. The COMMANDs are executed in the specified order.

  If a COMMAND matches one of default steps, then it is evaluated with the appropriate arguments. Otherwise if the COMMAND begins with a parenthesis, then it is evaluated as an Elisp expression. Otherwise it is assumed to be a shell command and executed with `shell-command`.

  ```
  [submodule "mu4e"]
  ```

```
        path = lib/mu4e
        url = git@github.com:djcb/mu.git
        build-step = test -e ./configure || autoreconf -i
        build-step = ./configure
        build-step = make -C mu4e > /dev/null
        build-step = borg-update-autoloads
        load-path = mu4e
```

To skip generating "autoloads" (e.g. using `use-package` to create "autoloads" on the fly), just provide the required build steps to build the package, omitting `borg-update-autoloads`. Borg silently ignores a missing "autoloads" file during initialization (`borg-initialize`).

```
[submodule "multiple-cursors"]
        path = lib/multiple-cursors
        url = git@github.com:magnars/multiple-cursors.el.git
        build-step = borg-byte-compile
```

Note that just because a package provides a `Makefile`, you do not necessarily have to use it.

Even if `make` generates the Info file, you might still have add `borg-makeinfo` as an additional build-step because the former might not generate a Info index file (named `dir`), which Borg relies on.

`borg-build-shell-command` [Variable]

This variable can be used to change how shell commands specified by `submodule.DRONE.build-step` are run. The default value is `nil`, meaning that each build step is run unchanged using `shell-command`.

If the value is a string, then that is combined with each build step in turn and the results are run using `shell-command`. This string must contain either %s, which is replaced with the unchanged build step, or %S, which is replaced with the result of quoting the build step using `shell-quote-argument`.

If the value is a function, then that is called once with the drone as argument and must return either a string or a function. If the returned value is a string, then that is used as described above.

If the value returned by the first function is another function, then this second function is called for each build step with the drone and the build step as arguments. It must return a string or `nil`. If the returned value is a string, then that is used as described above.

Finally the second function may execute the build step at its own discretion and return `nil` to indicate that it has done so.

Notice that if the value of this variable is a function, this function must a) be defined in a drone; and b) be registered as an autoload. This is because build happens in a separate Emacs process started with `-Q --batch`, which only receives the name of the function.

`submodule.DRONE.load-path` *PATH* [Variable]

This variable instructs `borg-activate` to add PATH to the `load-path` instead of the directory it would otherwise have added. Likewise it instructs `borg-byte-compile`

to compile the libraries in that directory. PATH has to be relative to the top-level of the repository of the drone named DRONE. This variable can be specified multiple times.

Normally Borg uses `lisp/` as the drone's `load-path`, if that exists, or else the top-level directory. If this variable is set, then it *overrides* the default location. Therefore, to *add* an additional directory, you also have to explicitly specify the default locaction.

```
[submodule "org"]
        path = lib/org
        url = git://orgmode.org/org-mode.git
        build-step = make
        load-path = lisp
        load-path = contrib/lisp
        info-path = doc
```

`submodule.DRONE.no-byte-compile` *PATH* [Variable]

This variable instructs `borg-byte-compile` to not compile the library at PATH. PATH has to be relative to the top-level of the repository of the drone named DRONE. This variable can be specified multiple times.

Sometimes a drone comes with an optional library which adds support for some other third-party package, which you don't want to use. For example `emacsql` comes with a PostgreSQL back-end, which is implemented in the library `emacsql-pg.el`, which requires the `pg` package. The standard Borg collective `emacs.g` assimilates `emacsql`, for the sake of the `epkg` drone, which only requires the SQLite back-end. To avoid an error about `pg` not being available, `emacs.g` instructs Borg to not compile `emacsql-pg.el`. (Of course if you want to use the PostgreSQL back-end and assimilate `pg`, then you should undo that.)

`submodule.DRONE.recursive-byte-compile` *BOOLEAN* [Variable]

Setting this variable to `true` instructs `borg-byte-compile` to compile DRONE's directories recursively. This isn't done by default because there are more repositories in which doing so would cause issues than there are repositories that would benefit from doing so.

Unfortunately many packages put problematic test files or (usually outdated) copies of third-party libraries into subdirectories. The latter is a highly questionable thing to do, but the former would be perfectly fine, if only the non-library elisp files did not provide a feature (which effectivly turns them into libraries) and/or if a file named `.nosearch` existed in the subdirectory. That file tells functions such as `normal-top-level-add-subdirs-to-load-path` and `borg-byte-compile` to ignore the containing directory.

`borg-byte-compile-recursive` [Variable]

Setting this variable to a non-nil value instructs `borg-byte-compile` to compile all drones recursively. Doing so is discouraged.

`submodule.DRONE.info-path` *PATH* [Variable]

This variable instructs `borg-initialize` to add PATH to `Info-directory-list`. PATH has to be relative to the top-level of the repository of the drone named DRONE.

`submodule.DRONE.no-makeinfo` *PATH* [Variable]

This variable instructs `borg-makeinfo` to not create an Info file for the Texinfo file at PATH. PATH has to be relative to the top-level of the repository of the drone named DRONE. This variable can be specified multiple times.

`submodule.DRONE.disabled` *true*|*false* [Variable]

If the value of this variable is `true`, then it is skipped by `borg-initialize`.

9 Low-level functions

You normally should not have to use the following low-level functions directly. That being said, you might want to do so anyway if you build your own tools on top of Borg.

`borg-worktree` *clone* [Function]
: This function returns the top-level of the working tree of the clone named CLONE.

`borg-gitdir` *clone* [Function]
: This function returns the Git directory of the clone named CLONE.

  It always returns `BORG-USER-EMACS-DIRECTORY/.git/modules/CLONE`, even when CLONE's Git directory is actually located inside the working tree.

`borg-get` *clone variable* **&optional** *all* [Function]
: This function returns the value of the Git variable `submodule.CLONE.VARIABLE` defined in `~/.emacs.d/.gitmodules`. If optional ALL is non-nil, then it returns all values as a list.

`borg-get-all` *clone variable* [Function]
: This function returns all values of the Git variable `submodule.CLONE.VARIABLE` defined in `~/.emacs.d/.gitmodules` as a list.

`borg-load-path` *clone* [Function]
: This function returns the `load-path` for the clone named CLONE.

`borg-info-path` *clone* **&optional** *setup* [Function]
: This function returns the `Info-directory-list` for the clone named CLONE.

  If optional SETUP is non-nil, then it returns a list of directories containing `texi` and/or `info` files. Otherwise it returns a list of directories containing a file named `dir`.

`borg-drones` **&optional** *include-variables* [Function]
: This function returns a list of all assimilated drones.

  The returned value is a list of the names of the assimilated drones, unless optional INCLUDE-VARIABLES is non-nil, in which case elements of the returned list have the form `(NAME . PLIST)`.

  PLIST is a list of paired elements. Property names are symbols and correspond to a VARIABLE defined in the Borg repository's `.gitmodules` file as `submodule.NAME.VARIABLE`.

  Each property value is either a string or a list of strings. If INCLUDE-VARIABLES is `raw` then all values are lists. Otherwise a property value is only a list if the corresponding property name is a member of `borg--multi-value-variables`. If a property name isn't a member of `borg--multi-value-variables` but it does have multiple values anyway, then it is undefined with value is included in the returned value.

`borg-clones` [Function]
: This function returns a list of all cloned packages.

  The returned value includes the names of all drones, as well as the names of all other repositories that are located directly inside `borg-drone-directory` but aren't tracked as submodules.

`borg-read-package` *prompt* **&optional** *edit-url* [Function]

This function reads a package name and the url of its upstream repository from the user, and returns them as a list.

When the `epkg` package is available, then the user is only prompted for the name of the package, and the upstream url is retrieved from the Epkg database. If the package isn't in the database then the url has to be provided by the user. If optional EDIT-URL is non-nil, then the url from the database, if any, is provided as initial input for the user to edit.

PROMPT is used when prompting for the package name.

`borg-read-clone` *prompt* [Function]

This function reads the name of a cloned package from the user.

There exist a few more functions, but those are considered to be internal and might therefore change in incompatible ways without that being noted in the changelog.

`borg--maybe-absorb-gitdir` *pkg* [Function]

`borg--maybe-reuse-gitdir` *pkg* [Function]

`borg--restore-worktree` *pkg* [Function]

`borg--call-git` *pkg* **&rest** *args* [Function]

`borg--expand-load-path` *drone path* [Function]

`borg--sort-submodule-sections` [Function]