Managing Dotiles: A Hard Problem

tropf

ABSTRACT

notes on thoughts on dotfile management

1. Intro

1.1. The Problem

I use many programs on many machines, which I customized to a certain extent. These customizations are mostly the same everywhere, but not always. And I want to synchronize those configurations.

There are many "solutions" to this synchronization, though not a single one really feels like it covers all my usecases.

1.2. There are two types of...

By origin (automated vs manual) and scope (global vs local). Automated refers to "by a program", while manual means "by hand"; obviously no configuration change is purely automated. Unfourtunately, all of the combinations exist and occur in my setups:

	local	global
automated	cache paths	emacs packages
manual	key file paths	keybinds

And as all these cases exist, there is no simple way to classify new changes.

For Example, most programs allow for a very simple way to separate global and local changes: Just use two files, e.g. `.vimrc.local` and `.vimrc.global`. But now we are missing the `.vimrc` file itself, and therefore automated changes can't be handled.

However these classification gets implemented, for some cases the specific implementation will have severe drawbacks (sync everything: too many conflicts; flag changes to sync by hand: too cumbersome...) ultimately making the answers to these theoretical questions less important.

1.3. Heterogeneous Systems

I use a vastly different systems regularly, each with their own peculiarities (no root access, different software versions, available performance, no GUI available etc.) which I want to address. For a while I had multiple sets of dotfiles, but I've chosen the easy way out and found solutions for each of my problems:

- Light- vs. Heavyweight editor: vim for the former, emacs for the latter
- Installation of companion programs: just add some notes in the config, and let future me figure it out for the specific system
- If feasable: programmatic detection, e.g. "if this is established via SSH start a tmux session"

As a result I now only have one set of dotfiles which makes things much more simple.

But this environment actually places a much more strict restriction on my dotfile distribution: It must be portable (and preferably lightweight).

2. Approaches

2.1. Use Git!

Do I even need to say this?

2.2. Symlinking

To allow easy global propagation of local (and automated) changes, just symlink the synchronized versions of the files into your home directory.

This makes keeping local changes more difficult, as the now global dotfile has to check (1) are there local settings and (2) include those local settings.

› The other way around (symlink real dotfile into git) does not work: git will only sync the link, but not the content.

2.3. (Read-Only-) Copy

If you don't need automated changes, use read-only copies of your dotfiles. Which really screws with your ability to do anything automated.

2.4. Injecting Includes

For a long time my preferred way to use dotfiles: Inject a link to your global version into the current dotfile, e.g. `source ~/a/b/c/bashrc.global`.

This is special for each config format and therefore requires some binary during the installation.

2.5. *Just* use git

```
cd ~
echo '*' > .gitignore
git init
```

It works, but it has no additional features whatsoever.

3. My Requirements

After thinking about possible solutions to my theoretical problems, I came down to these requirements: My dotfile management should...

- Allow for local AND global changes.
- Be available on all platforms.
- Be self-contained (no dependencies unless **widely** available).
- Synchronize via git.
- Be able to undo all its changes.

Most dotfile tools have to be installed to apply their changes, which rules them out for my setups. But not using these tools means not using any of their useful features.

Ultimately I (and thereby you) have to pick my poison. No matter what, you will have to compromise on some factor.

4. My Approach

I thought long and hard, and because I hate dependencies and myself, I now just use a single git repo as described here ⟨`https://www.atlassian.com/git/tutorials/dotfiles`⟩.

The basic idea is to initialize `$HOME` as a git repo. This introduces challenges of its on, but those can be dealt with.

4.1. Clever Details

One of the main problems I was concerned about when initializing my home as a git repo was that I could accidentally mean to add something to a repo, and maybe even sensitive files. However, this can't happen: Instead of using `.git` for files provided by git I use `.dots` – and therefore normal git invocations don't recognize the home directory as a git repo. If I need to interact with this git repo I use `git --git-dir=$HOME/.dots/ --work-tree=$HOME`which I created an alias for .

Also instead of a `.gitignore` I use the file `~/.dots/info/exclude`. This is not under version control itself, but removes the necessity for a `.gitignore` file in my home.

By Setting `status.showUntrackedFiles` to `no` I prevent git annoying me that not everything inside my home is under version control.

4.2. Advantages

This is as portable as it gets. If git works, this approach works.

Also it is dead simple to understand: Dotfiles are copied as is. No funky magic.

Reverting everything is fairly simple as it just requires removing all existing dotfiles. Also git allows for more complex reset procedures. However, removing the managed dotfiles means removing your dotfiles alltogether – w/o additional work also local changes are not respected anymore.

4.3. Drawbacks

You can't mix global, version-controlled and local sections in the same file. This sometimes becomes a problem when tools automatically try to change your dotfiles. I mitigate this by using pointing to local versions of my dotfiles: E.g. my `~/.bashrc` contains this snippet:

```
if test -r ~/.bashrc.local; then
    source ~/.bashrc.local
fi
```

I move all local changes into my `~/.bashrc.local` and keep my global dotfiles clean. Similar inclusion mechanisms exist for at least ssh, tmux and git config files. However, this requires a manual intervention for each inclusion in every dotfile.

For documentation purposes my dotfiles repo contains a readme file. This means that my home directory has a single file named `README.md`, with all other contents being directories. I consider that not ideal, as this readme file is located in my home but has no connection to it – though that is a drawback I'm willing to accept.

4.4. Summary

Follow the guide linked above. Create an alias to`git --git-dir=$HOME/.dots/ --work-tree=$HOME`. On a new machine use:

```
git clone --bare <git-repo-url> $HOME/.dots
alias dotscfg='git --git-dir=$HOME/.dots/ --work-tree=$HOME'
dotscfg config --local status.showUntrackedFiles no
echo '*' >> ~/.dots/info/exclude
dotscfg checkout
```

Use this alias instead of the`git` command when interacting with the repo. Use the flag `-f` (force) to add files.

5. See also

- The best way to store your dotfiles ⟨`https://www.atlassian.com/git/tutorials/dotfiles`⟩: approach I use

- Managing my dotfiles as a git repository ⟨`https://drewdevault.com/2019/12/30/dotfiles.html`⟩: Blog Post by Drew DeVault on using plain git to sync dotfiles
- nix home manager ⟨`https://github.com/nix-community/home-manager`⟩: preferred way to manage dotfiles by the nix community
- dotfiles.github.io ⟨`https://dotfiles.github.io/`⟩: list of dotfile managers
- how to remove sensitive data from git ⟨`https://help.github.com/articles/remove-sensitive-data/`⟩: required after screwups