Zsh Plugin Standard

Sebastian Gniazdowski

Version 0.93, 07/20/2019

Table of Contents

What is a Zsh plugin?

Zsh plugins were first defined by Oh-My-Zsh. They provide for a way to package together files that extend or configure the shell's functionality in a particular way.

At a simple level, a plugin:

1. Has its directory added to `$fpath` (zsh doc).
2. Has its first `*.plugin.zsh` file sourced (or `*.zsh`, `init.zsh`, `*.sh`, these are non-standard).

The first point allows plugins to provide completions and functions that are loaded via Zsh's `autoload` mechanism (a single function per-file).

From a more broad perspective, a plugin consists of:

1. A directory containing various files (main script, autoload functions, completions, Makefiles, backend programs, documentation).
2. A sourcable script that obtains the path to its directory via `$0` (see the next section for a related enhancement proposal).
3. A Github (or other site) repository identified by two components **username**/**pluginname**.
4. A software package containing any type of command line artifacts – when used with advanced plugin managers that have hooks, can run Makefiles, add directories to `$PATH`.

Below follow proposed enhancements and codifications of the definition of a "Zsh plugin" and the actions of plugin managers – the proposed standardization.

---

1. Standardized $0 handling

To get the plugin's location, plugins should do:

```
0="${${ZERO:-${0:#$ZSH_ARGZERO}}:-${(%):-%N}}"
0="${${(M)0#/*}:-$PWD/$0}"

# Then ${0:h} to get plugin's directory
```

The one-line code above will:

1. Be backwards-compatible with normal `$0` setting and usage.
2. Use `ZERO` if it's not empty,
   - plugin manager will be easily able to alter effective `$0` before loading a plugin,
   - this allows for e.g. `eval "$(<plugin)"`, which can be faster than `source` (comparison, note that it's not for a compiled script).

3. Use `$0` if it doesn't contain the path to the Zsh binary,
    - plugin manager will still be able to set `$0`, although more difficultly (requires `unsetopt function_argzero` before sourcing plugin script, and `0=⋯` assignment),
    - `unsetopt function_argzero` will be detected (it causes `$0` not to contain plugin-script path, but path to Zsh binary, if not overwritten by `0=⋯` assignment),
    - `setopt posix_argzero` will be detected (as above).
4. Use `%N` prompt expansion flag, which always gives absolute path to script,
    - plugin manager cannot alter this (no advanced loading of plugin is possible), but simple plugin-file sourcing (without a plugin manager) will be saved from breaking caused by the mentioned `*_argzero` options, so this is a very good last-resort fallback.
5. Finally, in the second line, it will ensure that `$0` contains an absolute path by prepending it with `$PWD` if necessary.

The goal is flexibility, with essential motivation to support `eval "$(<plugin)"` and definitely solve `setopt no_function_argzero` and `setopt posix_argzero` cases.

A plugin manager will be even able to convert a plugin to a function (author implemented such proof of concept functionality, it's fully possible – also in an automatic fashion), but performance differences of this are yet unclear. It might however provide a use case.

The last, 5th point also allows to use the `$0` handling in scripts (i.e. runnables with the hashbang `#!⋯`) to get the directory in which the script file resides.

2. Unload function

If a plugin is named e.g. `kalc` (and is available via `an-user/kalc` plugin-ID), then it can provide a function, `kalc_unload_plugin`, that can be called by a plugin manager to undo the effects of loading that plugin.

A plugin manager can implement its own tracking of changes made by a plugin so this is in general optional. However, to properly unload e.g. a prompt, detailed tracking (easy to do by the plugin creator) can provide better, predictable results. Any special, uncommon effects of loading a plugin are possible to undo only by a dedicated function.

However, an interesting compromise approach is available – to withdraw only the special effects of loading a plugin via the dedicated, plugin-provided function and leave the rest to the plugin manager. The value of such approach is that maintaining of such function (if it is to withdraw **all** plugin side-effects) can be a daunting task requiring constant monitoring of it during the plugin develoment process.

3. Plugin manager activity indicator

Plugin managers should set the `$zsh_loaded_plugins` array to contain all previously loaded plugins and the plugin currently being loaded (as the last element). This will allow plugins to:

1. Check which plugins are already loaded.
2. Check if it is being loaded by a plugin manager (i.e. not just sourced).

The first item allows a plugin to e.g. issue a notice about missing dependencies. Instead of issuing a notice, it may be able to satisfy the dependencies from resources it provides. For example, `pure` prompt provides `zsh-async` dependency library, which is a separate project and can be loaded by the user directly. Consequently, the prompt can decide to source its private copy of `zsh-async`, having also reliable `$0` defined by previous section (note: `pure` doesn't normally do this).

The second item allows a plugin to e.g. set up `$fpath`, knowing that plugin manager will not handle this:

```
if [[ ( ${+zsh_loaded_plugins} = 0 || ${zsh_loaded_plugins[-1]} != */kalc ) && -z
"${fpath[(r)${0:h}]}" ]]
then
    fpath+=( "${0:h}" )
fi
```

This will allow user to reliably source the plugin without using a plugin manager.

4. Global parameter with PREFIX for make, configure, etc.

Plugin managers may export the parameter `$ZPFX` which should contain a path to a directory dedicated for user-land software, i.e. for directories `$ZPFX/bin`, `$ZPFX/lib`, `$ZPFX/share`, etc. Suggested name of the directory is `polaris`, Zplugin uses this name and places this directory at `~/.zplugin/polaris` by default.

User can then configure hooks (feature of e.g. zplug and Zplugin) to invoke e.g. `make PREFIX=$ZPFX install` to install software like e.g. tj/git-extras. This is a developing role of Zsh plugin managers as package managers, where `.zshrc` has a similar role to Chef or Puppet configuration and allows to **declare** system state, and have the same state on different accounts / machines.

No-narration facts-list related to `$ZPFX`:

1. `export ZPFX="$HOME/polaris"` (or e.g. `$HOME/.zplugin/polaris`)
2. `make PREFIX=$ZPFX install`
3. `./configure --prefix=$ZPFX`
4. `cmake -DCMAKE_INSTALL_PREFIX=$ZPFX .`
5. `zplugin ice make"PREFIX=$ZPFX install"`
6. `zplug ⋯ hook-build:"make PREFIX=$PFX install"`

Appendix A: Revision history (history of updates to the document)

v0.93, 07/20/2019: 1/ Add the second line to the `$0` handling. 2/ Reformat to 80 columns
v0.92, 07/14/2019: 1/ Rename LOADED_PLUGINS to zsh_loaded_plugins. 2/ Suggest that $ZPFX is optional.
v0.91, 06/02/2018: Fix the link to the PDF for Github.
v0.9, 12/12/2017: Remove ZERO references (bad design), add TOC.
Reminder: The date format that uses slashes is `MM/DD/YYYY`.