EECS 481 - Homework 6b

Project Report - neovim

Yilin Yang — yiliny@umich.edu

Selected Project and Context

neovim (https://neovim.io) is an open-source text editor forked from the well-known editor vim. Its creators were principally motivated by a desire to open the editor's codebase to wider community contribution, in addition to extending vim's features, providing better support for plugin development, and generally refactoring vim's codebase. Though I have not found concrete user statistics, neovim is likely used by the same groups with which vim is popular, i.e. Unix developers, sysadmins, computer science students, and so on.

The past reticence of the vim community to merge large feature patches from neovim's maintainers was a key impetus for the neovim project's inception. neovim prides itself on having a more "newbie friendly" development community than vim, and does see more community engagement. As of April 8th, 2019, neovim had seen 88 pull requests over the previous month of which 71 were merged into master; over that same period, vim saw six pull requests on GitHub, none of which were merged.

Project Governance

neovim's development is centered on its GitHub repository, with additional discussion occurring on the neovim gitter messaging app. To contribute, developers follow the standard GitHub workflow of forking the main repository, creating a feature branch, implementing their changes, opening a pull request, and responding to whatever points are raised.

neovim has a few "full-time" maintainers who handle most development tasks with incidental contributions from other community members, many of whom are prominent plugin developers. The project also accepts donations through BountySource, which (when frequent enough) are used to hire a developer to work on neovim full time: some of neovim's key features, like its built-in terminal emulator, were developed in this fashion.

neovim pull requests have a three stage lifecycle: [WIP] ("Work In Progress"), [RFC] ("Request For Comment"), and [RDY] ("ReaDY to Merge"). The square-bracketed tags are prepended to the title of the pull request to indicate its progress. A [WIP] pull request simply announces to the community that a task has been "taken," a [RFC] is a request for code review by a project maintainer, and [RDY] indicates that the pull request has passed review and is ready to be merged into the master branch. Discussion may still take place on a [WIP] pull request to decide the best approach to implementation, even though the pull request isn't strictly ready for code review.

To be considered for acceptance, submitted pull requests must include tests that cover the changed portions of the code base, in addition to passing a fairly comprehensive CI test suite. The CI tests typically take more than 90 minutes to run to completion, and include concurrent builds using Travis CI, AppVeyor, QuickBuild, and CodeCov.io. The submitted code is built using multiple different compilers (gcc, clang-4.0, "clang Xcode," MSVC), linted, and run through its test suite using an LLVM address sanitizer. The `CONTRIBUTING.md` claims that submitted PRs must pass *all* CI builds to be merged, but this doesn't seem to be strictly true; the CI builds sometimes fail nondeterministically, producing "false positive" build failures, but so long as any reported errors seem like false positives, the PR may still be merged.

Task Description

https://github.com/neovim/neovim/issues/9136

This task would involved adding, testing, and documenting an `nvim_set_keymap` function to neovim's remote plugins API; once implemented, users and plugin developers would gain the ability to set custom keymappings through remote procedure calls written in languages other than vimscript.

I started by writing a stub function (that would just `assert(false)` and crash the program), and invoked it from a test case to verify that I had a workable starting point. From there, I wrote a crude "draft" implementation that accepted arguments (mapmode, options, the mapping's "left-hand side," the mapping's "right-hand side"), concatenated them into an allocated C-string, and passed them into the "legacy" vim function for creating a keymapping, writing tests as I went. This implementation essentially consisted of declaring a vimscript "declare mapping" command and passing it to an "eval statement." In response to suggestions from project maintainers, I refined the function signature and added stricter input validation.

Having received comments from a maintainer during a recent code review, I plan to make further changes, most of which involve refactoring the legacy functions that I invoke rather than treating them as pure black boxes.

Submitted Artifacts

Pull Request

https://github.com/neovim/neovim/pull/9924

This pull request adds approximately 600 lines to the neovim codebase. About half of these are tests, a quarter are my implementation of `nvim_set_keymap`, and the remaining quarter is a helper function named `strip_whitespace`.

Actual code changes can be found here: https://github.com/neovim/neovim/pull/9924/files

QA Strategy

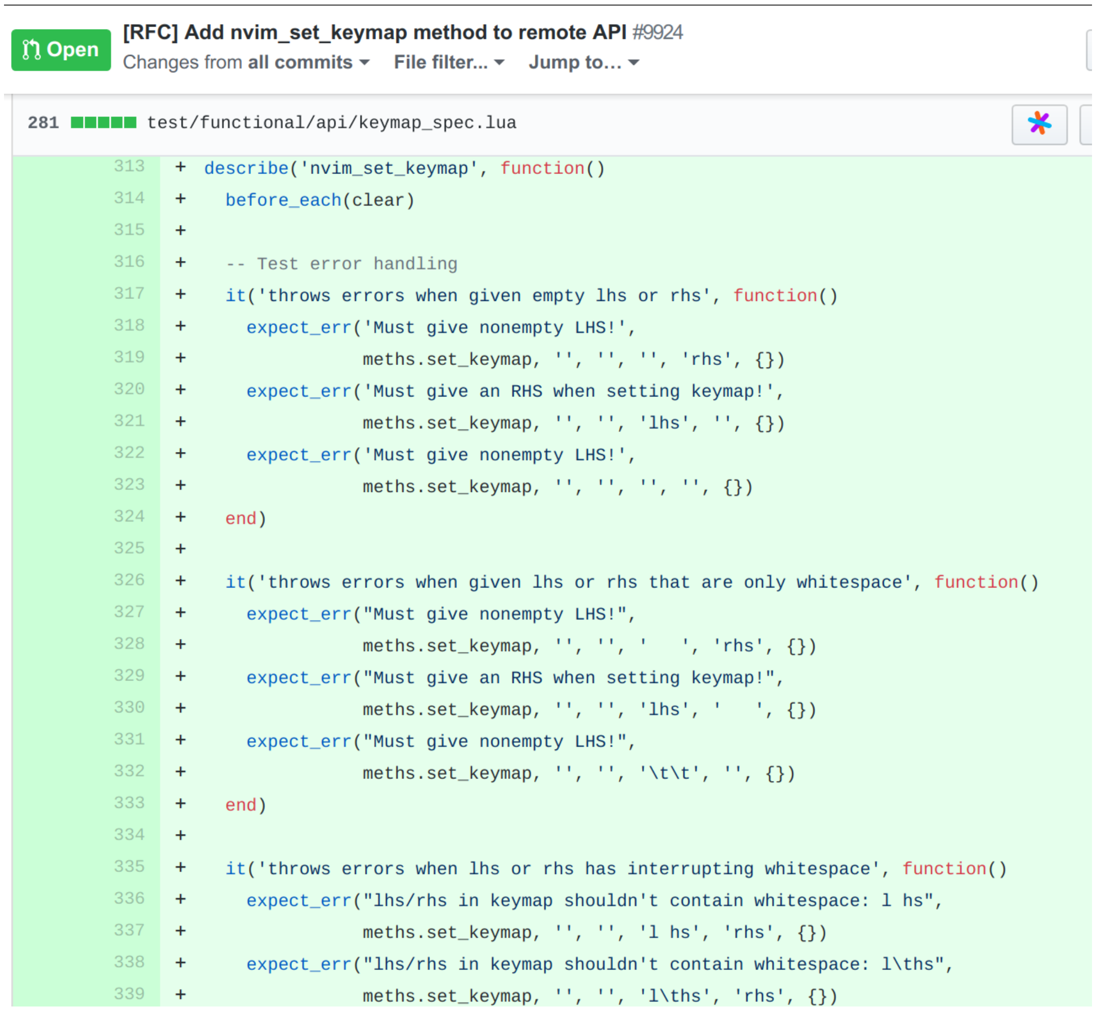
I followed a loosely test- and behavior-driven development process. I would write new code, write tests for that code, fix issues exposed by test failures, and then repeat that process.

I did this for two reasons: first, this is already my preferred process; and second, this is explicitly required by neovim's maintainers. neovim's codebase is large and unfamiliar to me, so running tests (both newly written and already existing tests) was the only way to gain confidence that my implementation worked as expected. They also gave an additional "entrance point" into my changes: though I was not able to directly attach to the running tests in GDB, I was able to copy-paste failing test inputs into the command line of a neovim instance and step through GDB from there. CodeCov.io reports that 92.9% of the lines touched by my changes were covered by the tests that I had written, greater than the 83.92% mandated by the CI build.

Aside from direct testing, I sought feedback from project maintainers regarding my work plan. neovim has an "api-contract" which, among other things, absolutely guarantees that remote plugin API call signatures will never change once they've been included in a release build. For this reason, it's particularly vital that the interface of `nvim_set_keymap` be robust to future change. neovim's maintainers have considerable experience with writing other components of the remote plugin API, and the suggestions that they made were authoritative and extremely useful.

QA Evidence

The test suite that I wrote can be found here:
https://github.com/neovim/neovim/pull/9924/files#diff-cfa8cca5afdd6030fe328d65e3eec567



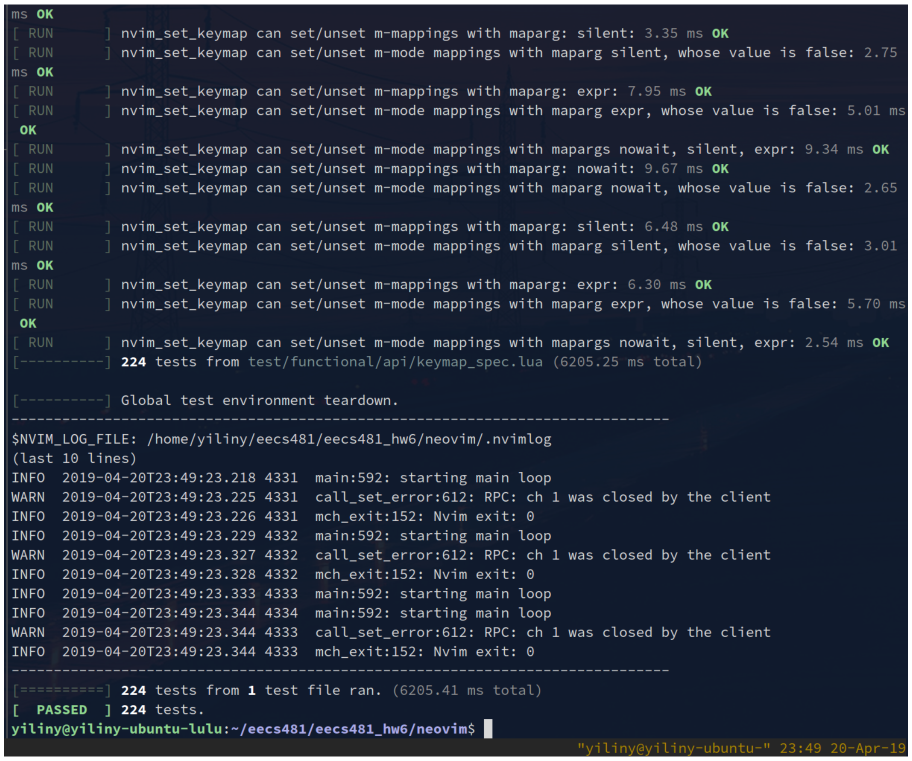
[RFC] Add nvim_set_keymap method to remote API #9924

281 test/functional/api/keymap_spec.lua

```
+ describe('nvim_set_keymap', function()
+   before_each(clear)
+
+   -- Test error handling
+   it('throws errors when given empty lhs or rhs', function()
+     expect_err('Must give nonempty LHS!',
+                meths.set_keymap, '', '', '', 'rhs', {})
+     expect_err('Must give an RHS when setting keymap!',
+                meths.set_keymap, '', '', 'lhs', '', {})
+     expect_err('Must give nonempty LHS!',
+                meths.set_keymap, '', '', '', '', {})
+   end)
+
+   it('throws errors when given lhs or rhs that are only whitespace', function()
+     expect_err("Must give nonempty LHS!",
+                meths.set_keymap, '', '', '   ', 'rhs', {})
+     expect_err("Must give an RHS when setting keymap!",
+                meths.set_keymap, '', '', 'lhs', '   ', {})
+     expect_err("Must give nonempty LHS!",
+                meths.set_keymap, '', '', '\t\t', '', {})
+   end)
+
+   it('throws errors when lhs or rhs has interrupting whitespace', function()
+     expect_err("lhs/rhs in keymap shouldn't contain whitespace: l hs",
+                meths.set_keymap, '', '', 'l hs', 'rhs', {})
+     expect_err("lhs/rhs in keymap shouldn't contain whitespace: l\ths",
+                meths.set_keymap, '', '', 'l\ths', 'rhs', {})
```

The tests I wrote for `nvim_set_keymap`.

The most recent Travis CI build (as of the time of writing), which runs the tests shown above, can be found here:
https://travis-ci.org/neovim/neovim/builds/522620100



```
ms OK
[ RUN      ] nvim_set_keymap can set/unset m-mappings with maparg: silent: 3.35 ms OK
[ RUN      ] nvim_set_keymap can set/unset m-mode mappings with maparg silent, whose value is false: 2.75
ms OK
[ RUN      ] nvim_set_keymap can set/unset m-mappings with maparg: expr: 7.95 ms OK
[ RUN      ] nvim_set_keymap can set/unset m-mode mappings with maparg expr, whose value is false: 5.01 ms
 OK
[ RUN      ] nvim_set_keymap can set/unset m-mode mappings with mapargs nowait, silent, expr: 9.34 ms OK
[ RUN      ] nvim_set_keymap can set/unset m-mappings with maparg: nowait: 9.67 ms OK
[ RUN      ] nvim_set_keymap can set/unset m-mode mappings with maparg nowait, whose value is false: 2.65
ms OK
[ RUN      ] nvim_set_keymap can set/unset m-mappings with maparg: silent: 6.48 ms OK
[ RUN      ] nvim_set_keymap can set/unset m-mode mappings with maparg silent, whose value is false: 3.01
ms OK
[ RUN      ] nvim_set_keymap can set/unset m-mappings with maparg: expr: 6.30 ms OK
[ RUN      ] nvim_set_keymap can set/unset m-mode mappings with maparg expr, whose value is false: 5.70 ms
 OK
[ RUN      ] nvim_set_keymap can set/unset m-mode mappings with mapargs nowait, silent, expr: 2.54 ms OK
[----------] 224 tests from test/functional/api/keymap_spec.lua (6205.25 ms total)

[----------] Global test environment teardown.
-----------------------------------------------------------------------------
$NVIM_LOG_FILE: /home/yiliny/eecs481/eecs481_hw6/neovim/.nvimlog
(last 10 lines)
INFO  2019-04-20T23:49:23.218 4331  main:592: starting main loop
WARN  2019-04-20T23:49:23.225 4331  call_set_error:612: RPC: ch 1 was closed by the client
INFO  2019-04-20T23:49:23.226 4331  mch_exit:152: Nvim exit: 0
INFO  2019-04-20T23:49:23.229 4332  main:592: starting main loop
WARN  2019-04-20T23:49:23.327 4332  call_set_error:612: RPC: ch 1 was closed by the client
INFO  2019-04-20T23:49:23.328 4332  mch_exit:152: Nvim exit: 0
INFO  2019-04-20T23:49:23.333 4333  main:592: starting main loop
INFO  2019-04-20T23:49:23.344 4334  main:592: starting main loop
WARN  2019-04-20T23:49:23.344 4333  call_set_error:612: RPC: ch 1 was closed by the client
INFO  2019-04-20T23:49:23.344 4333  mch_exit:152: Nvim exit: 0
-----------------------------------------------------------------------------
[==========] 224 tests from 1 test file ran. (6205.41 ms total)
[  PASSED  ] 224 tests.
yiliny@yiliny-ubuntu-lulu:~/eecs481/eecs481_hw6/neovim$ 
                                                      "yiliny@yiliny-ubuntu-" 23:49 20-Apr-19
```

A sampling of output from my tests for `nvim_set_keymap`. (The total number of tests reported also includes tests for `nvim_get_keymap`, which I did not write.)

My discussions with the neovim maintainers and their code review comments can be found at the pull request thread here:
https://github.com/neovim/neovim/pull/9924

Deviations from Initial Plan

The outcome of this project deviated from expectations in two major ways:

1. My pull request's life cycle was much longer than expected, and,
2. I was totally unable to work on my second proposed task (indent highlight groups/text objects) for lack of time.

I anticipated the latter item, to a degree; from the start, it was meant to act as a "fallback" option in case I completed `nvim_set_keymap` significantly faster than expected. `nvim_set_keymap` was nontrivial, and was – by itself – large enough to dominate the entirety of my work time.

The former came as a genuine surprise; I had hoped to submit my pull request and have it merged by the time I'd started writing this report. That was overly optimistic, in hindsight: it typically takes a few days for a pull request to be formally reviewed by a maintainer (as their attention is divided between multiple issues, pull requests, and their own personal lives), and when my pull request was reviewed, the maintainers noted substantial issues with my code's design. I intend to continue working, but it may take twice or even three times as long to complete my pull request than I'd initially expected. This speaks to some degree of overconfidence on my part, while also demonstrating that – as is common wisdom in industry – one's first implementation isn't necessarily the best implementation, and software projects in general can and will overrun their schedules at inopportune times.

Experiences and Recommendations

The following are several key lessons that this experience either reinforced or taught to me.

Documentation is vital to maintainable software

Between one third and half of the time I spent working was devoted to reading through existing code to understand its functionality and structure. I had chosen to contribute to neovim, in part, because of the quality of its developer documentation, and this proved to be a wise decision. **The depth and extent of neovim's documentation made it much easier to construct a work plan, and cut down the "warm-up time" necessary for me to develop a workflow.**

neovim's "own" functions are well documented with inline code comments and doxygen headers. The larger structure of its codebase is usefully summarized on its GitHub wiki. Even if a developer's text editor does not support a VSCode-esque "go-to definition" functionality for looking up function or variable declarations, that developer may search for that declaration on the neovim SourceGraph page, which performs automatic **static analysis** on neovim's code to better rank its results. For instance, if the user searches for "IS_WHITESPACE", SourceGraph will rank results that include a programming symbol (macro declaration, function, variable, etc.) named "IS_WHITESPACE" higher than results that include it incidentally, e.g. files that only include "IS_WHITESPACE" in a code comment.

Productive contributors must learn new tools quickly

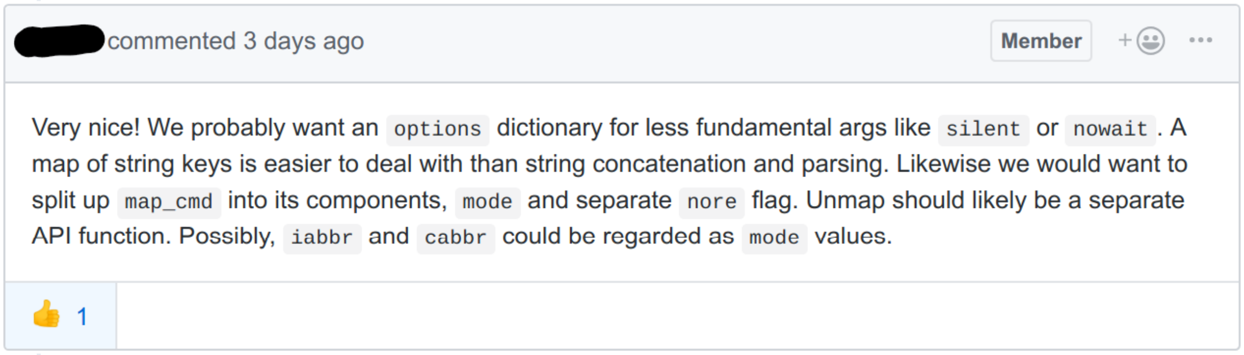
neovim has a massive infrastructure devoted to improving developer productivity, most of which is built atop existing libraries. neovim's build system relies on CMake – a popular, cross-platform build system that essentially automates Makefile generation – and `ninja`, an open-source replacement for `make` that can compile codebases orders of magnitude more quickly than the same. Its tests are written using `busted`, a testing library comparable to Mocha.js, but which is written in Lua rather than JavaScript.

All of these these tools solve important problems. The use of a monolithic Makefile, while ideal for course projects, becomes impractical – or at least suboptimal – when working with a codebase of neovim's size; CMake hides the boilerplate of code compilation behind a layer of abstraction. `ninja` massively accelerates the common compile-test-debug workflow, though it requires the installation and use of a new and unfamiliar software tool. `busted` tests allow developers to write **white box** and **black box tests** in the Lua scripting language, rather than C; this makes testing (writing helper functions, etc.) much easier – especially the boilerplate of initializing and communicating with neovim instances between tests – but adds a layer of complexity, as tests must cross a language barrier.

Learning to work with these tools is difficult, but refusing to learn or use those tools due to unfamiliarity would, in the long run, be even harder. For instance, the productivity lost to babysitting a custom Makefile, multiplied by dozens or hundreds of developers, is much greater than the “up-front” cost of teaching each developer to use CMake. A productive developer must be comfortable with learning new tools as the task requires.

Productive contributors communicate well and frequently, OR, Properly executed code review is an extremely valuable resource

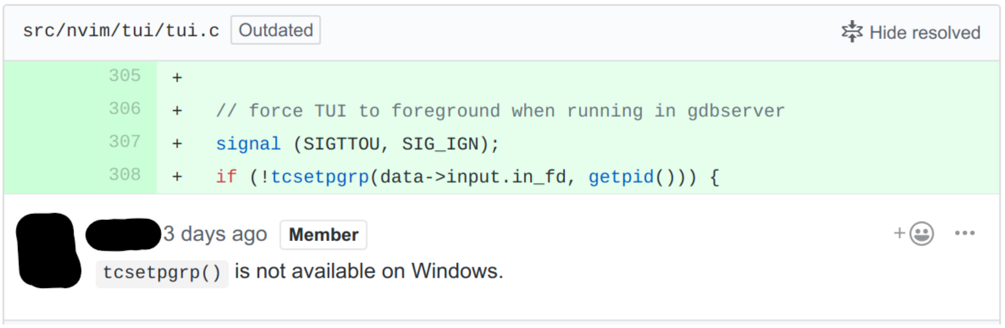
I’m happy to report that my interactions with neovim’s developer community has been almost universally positive. The maintainers that commented on my pull request were positive, encouraging, and respectful, on top of offering feedback that helped me gain insight into neovim’s code structure.


[redacted] commented 3 days ago Member

Very nice! We probably want an `options` dictionary for less fundamental args like `silent` or `nowait`. A map of string keys is easier to deal with than string concatenation and parsing. Likewise we would want to split up `map_cmd` into its components, `mode` and separate `nore` flag. Unmap should likely be a separate API function. Possibly, `iabbr` and `cabbr` could be regarded as `mode` values.

👍 1

*The most valuable feedback the maintainers offered pertained to `nvim_set_keymap`’s function signature. These are important, “high-level” design decisions; **eliciting these requirements** early proved wise, since a radical change to the function signature later on might have demanded a radical restructuring of the code that I’d written.*

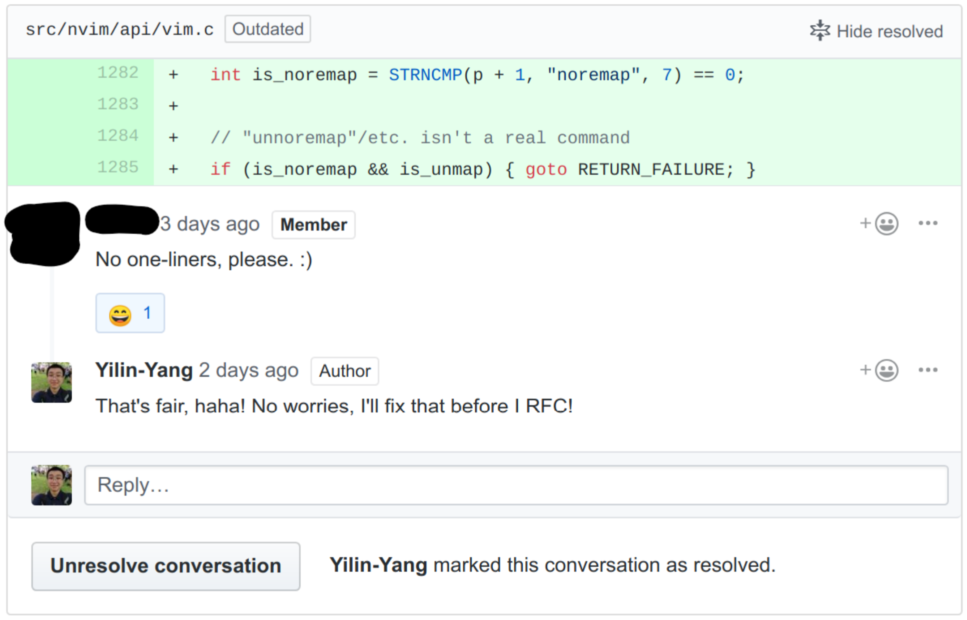

src/nvim/tui/tui.c Outdated Hide resolved

```
305 +
306 +   // force TUI to foreground when running in gdbserver
307 +   signal (SIGTTOU, SIG_IGN);
308 +   if (!tcsetpgrp(data->input.in_fd, getpid())) {
```

[redacted] [redacted] 3 days ago Member

`tcsetpgrp()` is not available on Windows.

This feedback was less helpful; the highlighted changes were suggested as “convenience code” for debugging by the neovim contributor’s wiki, and were not intended as part of the actual pull request. At the time, the pull request was still in [WIP].


src/nvim/api/vim.c Outdated Hide resolved

```
+   int is_noremap = STRNCMP(p + 1, "noremap", 7) == 0;
+
+   // "unnoremap"/etc. isn't a real command
+   if (is_noremap && is_unmap) { goto RETURN_FAILURE; }
```

[redacted] 3 days ago Member

No one-liners, please. :)

😄 1

Yilin-Yang 2 days ago Author

That's fair, haha! No worries, I'll fix that before I RFC!

Yilin-Yang marked this conversation as resolved.

neovim has a style guide to which its contributors stringently adhere. While some developers might use code review as an opportunity to "put down" their colleagues by nitpicking their style, this comment's feedback distinguished itself by being legitimate ("one-liners" are *prohibited by the style guide), non-threatening, and friendly in tone.*

Legacy code poses significant challenges

While neovim's "own" functions tend to be both elegant and well-commented, it is still built atop the codebase of "old" vim, which (itself) was built atop the older codebase of classic vi. Much of this legacy code (some might argue) does not adhere to established good practices and coding style: variables aren't descriptively named, function preconditions are undocumented, important code is heavily duplicated, function bodies are several hundred lines long, and the editor's state is largely maintained using global variables. Worse still, these functions are typically so tightly coupled to a global editor state that they are difficult, if not impossible to test; consequently, many of them are not directly covered by neovim's test suites.

I avoided modifying these functions for fear of breaking functionality in a manner that wouldn't be exposed by tests, even when doing so would have produced a "more elegant" solution. This experience clearly demonstrated the importance of **designing for maintainability**: much code, once written, will "live forever," and anticipating future developer needs, within reason, is key to the construction of long-lived software projects.

EECS *exams* may be more instructive than EECS projects, OR, Many impossible problems are tractable if you stop to seriously think about them

The single piece of legacy code with which I've spent the most time is the `int do_map()` function, which reads a command from the user and:

- Displays all user-set mappings and abbreviations, or,
- Displays all user-set mappings and abbreviations that match a particular “pattern,” or,
- Deletes a user-set mapping or abbreviation, or,
- Modifies a user-set mapping or abbreviation, or,
- Declares a new user-set mapping or abbreviation.

It does this with multiple consecutive for-loops that iterate over the global hashtables in which existing mappings and abbreviations are stored.

The sheer size of this function convinced me, at first sight, that modifying this function would be impossible, that I should treat it as a complete black box, and that I should work *around* it and its idiosyncrasies as much as possible. While I produced a workable solution using this approach, neovim’s maintainers pointed out, during code review, that an optimal implementation of `nvim_set_keymap` would necessarily refactor `int do_map()` rather than “acquiescing” to its faults, to the future benefit of plugin developers (who use the remote plugin API), but also to later neovim contributors who might want to call those refactored functions in the future.

Having decided to follow their advice, I literally printed the body of `do_map()` on paper and went over it in pen, as if I were taking an ENGR 101 or EECS 280 exam that asked me to determine the behavior of a code sample through **bottom-up comprehension.** While still difficult, this was easier than I expected (to their credit, vi/vim’s developers still did a good job of **writing “why” documentation** for this function, despite its lack of “prettiness”), and it was also very illuminating.

Understanding `do_map`’s implementation requires no more knowledge than that taught by EECS 280/281 and EECS 370. The function stores mappings and abbreviations in a (hardcoded) *256-bucket hash table* that performs collision resolution using *separately chained linked lists.* (It hashes the “mapmode” and first character of each mapping using bitwise operations.) Though this revelation may seem obvious, recognizing this reaffirmed that **our EECS courses do actually teach us material that is useful in the real world,** and that **in some respects, the skills we learn from taking EECS *exams* may be more representative of industry practice than our EECS projects.**

In summary,

If I were to start a similar open source project, being informed by my experiences contributing to neovim, I would:

- Strongly emphasize the value of openness and professionalism, especially when dealing with new contributors,
    - Opening my pull request was intimidating, but the friendliness of the maintainers who responded allayed most of my fears.
- Encourage communication between contributors and experienced “senior” developers, especially with respect to code review or requirements elicitation,

    - Learning that “you did a marvelous job doing the wrong thing” would be as demoralizing as it would be devastating, due to the amount of code that would need to be scrapped. Identifying problems early reduces the expense of changing course.
- Heavily document the “why” and “how” of any code that I write, as well as the initial “set-up process” necessary to build the project and run its tests.
    - Knowing “why” certain code is present helps future developers understand its purpose, making it easier for them to use it, and helping them to feel safe from “gotchas” if they were ever to refactor it.
- Reduce code coupling as much as possible, to maximize the degree to which code can be tested in isolation.
    - A comprehensive test suite is arguably the single most valuable quality assurance tool that a software project can have.

Advice for Future Students

You don’t understand something until you can explain it simply in your own words; study until you reach that point.

(I’m willing to let future students read this report.)