Написание плагинов Vim

Опубликовано 6 сентября 2011 г.

Некоторое время назад я написал пост о переходе обратно на Vim . С тех пор я написал два плагина для Vim, один из которых был официально "выпущен".

Несколько человек спросили меня, не напишу ли я руководство по созданию плагинов Vim. Я не чувствую себя достаточно уверенным, чтобы написать официальное "руководство", но у меня есть несколько советов для авторов плагинов Vim, которые могут быть полезны.

Другие люди, которые знают больше, чем я

Написание двух приличных по размеру плагинов Vim дало мне некоторый опыт, но есть много людей, которые знают гораздо больше меня. В частности, на ум приходят два. Я бы хотел, чтобы они написали руководства (или даже книги) о современном написании скриптов Vim.

Тим Поуп

Первый — Тим Поуп . Он написал кучу плагинов для Vim, таких как Pathogen , Surround , Repeat , Speeddating и Fugitive . Каждый из них ясен, сфокусирован и отполирован.

Было бы здорово почитать его руководство по тонкостям написания скриптов в Vim.

Прокрутите вниз

Другой человек, который приходит на ум, — это Скрулоуз , автор NERDTree , NERDCommenter и Syntastic .

Его плагины большие и полнофункциональные, но работают невероятно хорошо, учитывая, насколько сложно и мучительно работать с Vimscript. Я бы с удовольствием прочитал его руководство по написанию масштабных плагинов Vim.

Быть совместимым с патогенами

На дворе 2011 год. При написании плагина, *пожалуйста* , сделайте его исходный код совместимым с Pathogen . Сделать это очень просто — просто настройте файлы вашего проекта следующим образом:

```
yourplugin/
    doc/
        yourplugin.txt
    plugin/
        yourplugin.vim
    ...
    README
    LICENSE
```

Это позволит пользователям использовать Pathogen (или Vundle) для установки и использования вашего плагина.

Дни "распаковки и перетаскивания файлов в нужные каталоги" и ужасы Vimballs прошли. Pathogen и Vundle — правильный способ управления плагинами, так что позвольте вашим пользователям использовать их.

Пожалуйста, ради Бога, используйте обычный режим!

Мой первый совет по написанию скриптов — это что-то простое, но важное. Если вы пишете плагин Vim и вам нужно выполнить некоторые действия, у вас может возникнуть соблазн использовать `normal`. Не надо. Вместо этого вам нужно использовать `normal!`.

`normal!` похоже на `normal`, но игнорирует сопоставления, установленные пользователем. Если вы используете plain old `normal dd`, а я переназначил, `dd` чтобы сделать что-то другое, вызов будет использовать мое сопоставление и, вероятно, не сделает то, что ожидает ваш плагин. Использование `normal!` гарантирует, что вызов сделает то, что вы ожидаете, независимо от того, что сопоставил пользователь.

Это единичный случай более общей темы. Vim очень настраиваемый, и пользователи будут делать много безумных вещей в своих `.vimrc` файлах. Если клавишу можно сопоставить или изменить настройку, вы *должны* предположить, что какой-то пользователь вашего плагина сопоставил или изменил это.

Правильное расположение клавиш

Большинство плагинов добавляют сопоставления клавиш, чтобы сделать их более удобными в использовании. К сожалению, это может быть сложно сделать правильно. Вы никогда не можете знать, какие клавиши ваши пользователи уже сопоставили сами, а копирование чьего-то любимого сопоставления клавиш сломает их мышечную память и будет раздражать их бесконечно.

Когда сопоставлять ключи

Первый вопрос, который следует задать, — нужно ли вообще вашему плагину сопоставлять ключи?

У моего плагина Gundo есть только одна функция, которую нужно сопоставить с клавишей, чтобы она стала полезной: действие «включить Gundo».

Gundo не отображает этот ключ сам по себе, потому что независимо от того, какое сопоставление "по умолчанию" я выберу, кто-то уже его сопоставил. Вместо этого я добавил раздел прямо в файл README, который показывает, как пользователь может сопоставить ключ самостоятельно:

```
nnoremap <F5> :GundoToggle<CR>
```

Заставляя пользователей добавлять эту строку самостоятельно, `.vimrc` мы показываем им, какая клавиша используется для переключения Gundo (что им в любом случае нужно знать), а также делаем очевидным, как изменить ее в соответствии со своими вкусами.

imap и nmap — чистое зло

Иногда заставить пользователя сопоставить свои собственные клавиши не сработает. Возможно, ваш плагин имеет много сопоставлений, которые пользователю было бы утомительно настраивать вручную (как мой плагин Threesome), или его сопоставления мнемонические и не будут иметь смысла, если сопоставить их с другими клавишами.

Я расскажу подробнее о том, как с этим справиться, чуть позже, но самое важное, что следует помнить при сопоставлении собственных клавиш, — это то, что вы всегда, *всегда* , ***всегда*** должны использовать `noremap` формы различных `map` команд.

Если вы сопоставляете ключ с `nmap` и пользователь переназначил ключ, который использует ваше сопоставление, ваш сопоставленный ключ почти наверняка не будет делать то, что вы хотите. Использование `nnoremap` будет игнорировать сопоставления пользователя и делать то, что вы ожидаете.

Это тот же принцип, что `normal` и `normal!`: *никогда не* доверяйте конфигурациям своих пользователей.

Позвольте мне настроить сопоставления

Если вы считаете, что ваш плагин должен сопоставлять некоторые клавиши, сделайте эти сопоставления настраиваемыми каким-либо образом.

Есть несколько способов сделать это. Самый простой способ — предоставить опцию конфигурации, которая отключает все сопоставления. Пользователь может переназначить клавиши по своему усмотрению. Например:

```
if !exists('g:yourplugin_map_keys')
    let g:yourplugin_map_keys = 1
endif

if g:yourplugin_map_keys
    nnoremap <leader>d :call <sid>YourPluginDelete()<CR>
endif
```

Обычные пользователи получат автоматическую настройку сопоставлений, а опытные пользователи могут переназначить клавиши по своему усмотрению, чтобы избежать теневого копирования своих собственных сопоставлений.

Если все сопоставления вашего плагина начинаются с общего префикса (например `<leader>`, или `<localleader>`), у вас есть другой вариант: разрешить пользователям настраивать этот префикс. Это подход, который я использовал в Threesome . Это работает так:

```
if !exists('g:yourplugin_map_prefix')
    let g:yourplugin_map_prefix = '<leader>'
endif

execute "nnoremap"  g:yourplugin_map_prefix."d"  ":call <sid>YourPluginDelete()<CR>"
```

Команда `execute` позволяет динамически создавать строку сопоставления, чтобы пользователи могли изменять префикс сопоставления.

Есть третий вариант решения этой проблемы: `hasmapto()` функция Vim. Некоторые плагины будут использовать это для сопоставления команды с клавишей , *если только* пользователь уже не сопоставил эту команду с чем-то другим. Мне лично не нравится этот вариант, потому что он кажется мне менее понятным, но я знаю, что другие люди думают иначе, поэтому я хотел упомянуть об этом.

Локализовать сопоставления и настройки

Следующий шаг в создании хорошего плагина Vim — попытаться минимизировать эффекты ваших сопоставлений клавиш и изменений настроек. Некоторым плагинам потребуются глобальные эффекты, а другим — нет.

Например: если вы пишете плагин для работы с файлами Python, он должен действовать только для буферов Python, а не для всех буферов.

Локализация сопоставлений

Связывание клавиш легко локализовать для отдельных буферов. Все команды `noremap` могут принимать дополнительный `<buffer>` аргумент, который локализует сопоставление для текущего буфера.

```
" Remaps <leader>z globally
nnoremap <leader>z :YourPluginFoo<cr>

" Remaps <leader>z only in the current buffer
nnoremap <buffer> <leader>z :YourPluginFoo<cr>
```

Однако проблема в том, что вам нужно запустить эту команду в каждом буфере, в котором вы хотите, чтобы отображение было активным. Для этого ваш плагин может использовать `autocommand`. Вот полный пример, использующий эту концепцию плюс ранее упомянутые параметры конфигурации:

```
if !exists('g:yourplugin_map_keys')
    let g:yourplugin_map_keys = 1
endif

if !exists('g:yourplugin_map_prefix')
    let g:yourplugin_map_prefix = '<leader>'
endif

if g:yourplugin_map_keys
    execute "autocommand FileType python" "nnoremap <buffer>" g:yourplugin_map_prefix."d"  ":call
<sid>YourPluginDelete()<CR>"
endif
```

Теперь ваш плагин будет определять сопоставление клавиш только для буферов Python, а ваши пользователи смогут отключать или настраивать это сопоставление по своему усмотрению.

Эта команда сопоставления довольно уродлива. К сожалению, такова цена использования Vimscript и попытки сделать плагин, который будет работать для многих пользователей. Позже я расскажу об одном возможном решении этой уродливости.

Настройки локализации

Так же, как вы должны делать сопоставления локальными для буферов, когда это уместно, вы должны делать то же самое с настройками, такими как `foldmethod`, `foldmarker` и `shiftwidth`. Не все настройки могут быть установлены локально в буфере. Вы можете прочитать, `:help <settingname>` чтобы узнать, возможно ли это.

Вы можете использовать `setlocal` вместо `set` для локализации настроек в отдельных буферах. Как и в случае с сопоставлениями, вам нужно будет использовать автокоманду для запуска `setlocal` команды каждый раз, когда пользователи открывают новый буфер.

Автозагрузка — ваш друг

Если ваш плагин будет использоваться пользователями постоянно, вы можете пропустить этот раздел.

Если вы пишете что-то, что будет использоваться только в определенных случаях, вы можете помочь своим пользователям, используя `autoload` функционал Vim, позволяющий отложить загрузку кода до тех пор, пока пользователь фактически не попытается его использовать.

Работает этот способ `autoload` довольно просто. Обычно вы привязываете клавишу к вызову одной из функций вашего плагина примерно так:

```
nnoremap <leader>z :call YourPluginFunction()<CR>
```

Вы можете использовать автозагрузку, добавив `yourplugin#` к имени функции:

```
nnoremap <leader>z :call yourplugin#YourPluginFunction()<CR>
```

При запуске этого сопоставления Vim выполнит следующие действия:

1. Проверьте, `YourPluginFunction` определено ли уже. Если да, вызовите его.
2. В противном случае найдите `~/.vim/autoload/` файл с именем `yourplugin.vim`.
3. Если он существует, проанализируйте и загрузите файл (который, предположительно, определяется `YourPluginFunction` где-то внутри).
4. Вызовите функцию.

Это означает, что вместо того, чтобы помещать весь код вашего плагина, `plugin/yourplugin.vim` вы можете поместить туда только код сопоставления клавиш, а остальное вынести в `autoload/yourplugin.vim`.

Если ваш плагин содержит приличный объем кода, это может значительно сократить время запуска Vim.

Ознакомьтесь с полной документацией `autoload` по запуску, `:help autoload` чтобы узнать больше.

Обратная совместимость — это большое дело

После того, как вы написали свой плагин Vim и выпустили его в свет, вам нужно его поддерживать. Пользователи найдут ошибки и попросят новые функции.

Частью ответственного разработчика любого рода, включая автора плагина Vim, является поддержание обратной совместимости, *особенно* для инструментов, которые пользователи будут использовать каждый день и которые запечатлеются в их мышечной памяти. Пользователи полагаются на инструменты для работы, а инструменты, которые нарушают обратную совместимость, быстро потеряют доверие пользователей.

Поддержание обратной совместимости приведет к тому, что код вашего плагина местами станет некачественным, но это цена сохранения удовлетворенности ваших пользователей.

Что имеет значение для совместимости Backards?

Для плагина Vim наиболее важной частью обеспечения обратной совместимости является обеспечение того, чтобы сопоставления клавиш, настроенные или нет, продолжали выполнять ожидания пользователей.

Если ваш плагин отображает клавишу `X` на do `Y`, то нажатие `X` должно *всегда* do `Y`, даже если вы измените способ `Y` вызова, переименовав `Y` его в `Z`. Это может означать изменение `Y` на функцию-обертку, которая просто вызывает `Z`.

Есть много других аспектов обратной совместимости, которые вам придется учитывать, в зависимости от цели вашего плагина. Правило, которому вы должны следовать, таково: если пользователь использует этот плагин ежедневно и его использование запечатлено в его мышечной памяти, обновление плагина не должно заставить его переучиваться чему-либо.

Используйте семантическое версионирование, чтобы оставаться в здравом уме

Быстрый, простой и легкий способ документировать состояние вашего плагина — использовать семантическое управление версиями .

Семантическое версионирование — это просто идея, что вместо выбора произвольных номеров версий для релизов вашего проекта вы используете номера версий, которые осмысленно описывают состояние обратной совместимости.

Вкратце, эти правила описывают, как следует выбирать номера версий для новых релизов:

- Номера версий состоят из трех компонентов: `major.minor.bugfix`. Например: `1.2.4` или `2.13.0`.
- Версии с основной версией 0 (например `0.2.3`,) не дают никаких гарантий относительно обратной совместимости. Вы вольны сломать все, что захотите. Только после релиза `1.0.0` вы начинаете давать обещания.
- Если в выпуске представлены обратно несовместимые изменения, увеличьте основной номер версии.
- Если версия обратно совместима, но добавляет *новые* функции, увеличьте номер младшей версии.
- Если релиз просто исправляет ошибки, рефакторит код или улучшает производительность, увеличьте номер версии исправления ошибок.

Эта простая схема позволяет пользователям легко определить (в широком смысле), что изменилось при обновлении вашего проекта.

Если изменился только номер исправления ошибки, они могут без опасений обновиться и продолжить работу, не беспокоясь об изменениях, если только им не любопытно.

Если номер младшей версии изменился, они могут просмотреть журнал изменений, чтобы узнать, какие новые функции им хотелось бы использовать, но если они заняты, они могут просто выполнить обновление и двигаться дальше.

Если основной номер версии изменился, это серьезный тревожный сигнал, и им следует внимательно прочитать журнал изменений, чтобы увидеть, что изменилось.

Некоторым не нравится семантическое версионирование по следующей причине:

> Если мне придется увеличивать основной номер версии каждый раз, когда я вношу обратно несовместимые изменения, я быстро доберусь до уродливых версий вроде 24.1.2!

На это я отвечаю: «Да, но если это происходит, то вы изначально что-то делаете неправильно».

Держите свой проект в состоянии "бета" (т. е. версии `0.*.*`) столько, сколько вам нужно для свободного экспериментирования. *Не торопитесь* и убедитесь, что вы сделали все (в основном) правильно. После релиза `1.0.0` пора начинать быть ответственным и заботиться об обратной совместимости.

Постоянная поломка функциональности вредит вашим пользователям, снижая их производительность и раздражая их. Да, это означает добавление некоторого хлама в ваш код с течением времени, но это цена того, чтобы не быть злым.

Документируйте все

Критическая часть выпуска плагина Vim в мир — написание документации для него. У Vim есть фантастическая документация, поэтому ваши плагины должны следовать его примеру и предоставлять исчерпывающую документацию.

Выберите несколько требований и придерживайтесь их

Самая важная часть вашей документации — это сообщение пользователям о том, что им нужно иметь для использования вашего плагина. Vim работает практически на любой мыслимой системе и может быть скомпилирован многими различными способами, поэтому конкретное описание требований вашего плагина избавит пользователей от множества проб и ошибок.

- Ваш плагин работает только с Vim версии XY или более поздней?
- Требуется ли поддержка Python/Ruby и т. д. в компиляции? Какая версия?
- Не работает на Windows?
- Полагается ли он на внешний инструмент?

Если ответ на любой из этих вопросов «да», вы *должны* указать это в документации.

Написать README

Первый шаг к документированию вашего плагина — написать файл README для репозитория. Вы также можете использовать текст этого файла в качестве описания, если вы загружаете свой плагин на сайт vim , или содержимое сайта вашего плагина, если вы его создаете.

Вот несколько примеров того, что следует включить в файл README:

- Обзор того, что делает плагин.
- Скриншоты, если возможно.
- Требования.
- Инструкция по установке.
- Распространенные параметры конфигурации, которые захотят знать многие пользователи.
- Ссылки на:
    - Канонический веб-адрес для поиска плагина.
    - Баг-трекер для плагина.
    - Исходный код или репозиторий плагина.

Создайте простой веб-сайт

Это не обязательно, но наличие простого веб-сайта для вашего плагина — это дополнительный штрих, который придаст ему более изысканный вид.

Он также предоставляет вам канонический URL-адрес, по которому люди могут получить самую свежую информацию о вашем плагине.

Я сделал простые сайты для обоих моих плагинов: Gundo и Threesome . Можете свободно использовать их в качестве примера или даже взять их код и использовать его для своих собственных сайтов плагинов, если хотите.

Напишите справочный документ Vim

Основная часть документации вашего плагина должна быть в форме справочного документа Vim. Пользователи привыкли использовать Vim `:help`, и они ожидают, что смогут использовать его для изучения вашего плагина.

Создать справочный документ так же просто, как создать `doc/yourplugin.txt` файл в вашем проекте. Он будет автоматически проиндексирован, `pathogen#helptags()` так что ваши пользователи будут иметь документы под рукой.

Два простых способа изучить синтаксис файлов справки — прочитать `:help help-writing` и использовать в качестве примера существующий файл справки плагина.

Не торопитесь и создайте красивый файл справки, которым вы сможете гордиться. Не бойтесь добавить немного индивидуальности в свои документы, чтобы разбить сухость. Синтастический файл справки — отличный пример (особенно `About` раздел).

Что следует включить в документацию:

- Краткий обзор плагина.
- Более подробное описание использования плагина.
- Каждое сопоставление клавиш, создаваемое плагином.
- Способы расширения плагина, если применимо.
- Все переменные конфигурации (включая их значения по умолчанию!).
- Журнал изменений плагина.
- Лицензия плагина.
- Ссылки на репозиторий плагина и систему отслеживания ошибок.

Вкратце: ваш файл справки должен содержать *всю* необходимую пользователю информацию о вашем плагине.

Ведите журнал изменений

Последняя часть документирования вашего проекта — ведение журнала изменений. Вы можете пропустить это, пока ваш проект все еще находится в стадии «бета» (т.е. ниже версии `1.0.0`), но как только вы официально выпустите настоящую версию, вам нужно будет информировать своих пользователей о том, что изменилось между выпусками.

Я предпочитаю включать этот журнал в файл README, на веб-сайт плагина и в документацию, чтобы пользователям было максимально просто увидеть, что изменилось.

Постарайтесь, чтобы язык журнала изменений был достаточно высоким, чтобы ваши пользователи могли его понять, не зная ничего о реализации вашего плагина. Такие вещи, как «добавлена функция X» и «исправлена ошибка Y» хороши, а такие вещи, как «рефакторинг внутренней работы функции утилиты Z» лучше оставить в сообщениях коммита.

Делаем Vimscript удобным

Худшая часть написания плагинов Vim — это, без сомнения, работа с Vimscript. Это эзотерический язык, который развивался органически на протяжении многих лет, по-видимому, без какого-либо четкого направления дизайна.

В Vim добавляются функции, затем в Vimscript добавляются функции для управления этими функциями, затем добавляются хакерские обходные пути для обеспечения гибкости.

Синтаксис краткий, уродливый и непоследовательный. `" foo` Комментарий? Иногда.

Большую часть времени, которое вы потратите на написание своего первого плагина, вы потратите на изучение того, как делать вещи в Vimscript. Справочная документация по всем его функциям является подробной, но может быть трудно найти то, что вы ищете, если вы не знаете точного названия. Просмотр других плагинов часто очень полезен, указывая вам на то, что вам нужно.

Есть несколько способов облегчить работу с Vimscript, и я кратко расскажу о двух из них.

Обернуть. Всё.

Первый совет, который я могу дать, таков: если вы хотите, чтобы ваши плагины были читабельными и удобными для поддержки, то вам нужно упаковать функциональность даже больше, чем в других языках.

Например, мой плагин Gundo имеет несколько служебных функций, которые выглядят следующим образом:

```
function! s:GundoGoToWindowForBufferName(name)"{{{
    if bufwinnr(bufnr(a:name)) != -1
        exe bufwinnr(bufnr(a:name)) . "wincmd w"
        return 1
    else
        return 0
    endif
endfunction"}}}
```

Эта функция перейдет к окну для указанного имени буфера и изящно обработает случай, когда буфер/окно не существует. Это многословно, но гораздо более читабельно, чем альтернатива использования этого `if` оператора в каждом месте, где мне нужно переключать окна.

По мере написания плагина вы будете «выращивать» ряд этих служебных функций. Каждый раз, когда вы дублируете код, вам следует подумать о его создании, но вам также следует делать это каждый раз, когда вы пишете особенно сложную строку Vimscript. Вытягивание сложных строк в именованные функции сэкономит вам много времени на просмотр и переосмысление в дальнейшем.

Написание скриптов Vim с использованием других языков

Another option for making Vimscript less painful is to simply not use it much at all. Vim includes support for creating plugins in a number of other languages like Python and Ruby. Many plugin authors choose to move nearly all of their code into another language, using a small Vimscript "wrapper" to expose it to the user.

I decided to try this approach with Threesome after seeing it used in the vim-orgmode plugin to great effect. Overall I consider it to be a good idea, with a few caveats.

First, using another language will requires your plugin's users to use a version of Vim compiled with support for that version. In this day and age it's usually not a problem, but if you want your plugin to run everywhere then it's not an option.

Using another language adds overhead. You need to not only learn Vimscript but also the interface between Vim and the language. For small plugins this can add more complexity to the project than it saves, but for larger plugins it can pay for itself. It's up to you to decide whether it's worth it.

Finally, using another language does not entirely insulate you from the eccentricities of Vimscript. You still need to learn how to do most things in Vimscript — using another language simply lets you wrap most of this up more neatly than you otherwise could.

Unit Testing Will Make You Drink

Unit testing (and other types of testing) is becoming more and more popular today. In particular the Python and Ruby communities seem to be getting more and more excited about it as time goes on.

Unfortunately, unit testing Vim plugins lies somewhere between "painful" and "garden-weasel`ing your face" on the difficulty scale.

I tried adding some unit tests to Gundo, but even after looking at a number of frameworks I was spending hours simply trying to get my tests to function.

I didn't even bother trying to add tests to Threesome because for every hour I would have spent fighting Vim to create tests I could have cleaned up the code and fixed bugs instead.

I'll gladly change my opinion on the subject if someone writes a unit testing framework for Vim that's as easy to use as Cram. In fact, I'll even buy the author a $100 bottle of scotch (or whatever they prefer).

Until that happens I personally don't think it's worth your time to unit test Vim plugins. Spend your extra hours reading documentation, testing things manually with a variety of settings, and thinking hard about your code instead.

TL;DR

Writing Vim plugins is tricky. Vimscript is a rabbit hole of sadness and despair, and trying to please all your users while maintaining backwards compatibility is a monumental task.

With that said, creating something that people use every day to help them make beautiful software projects is extremely rewarding. Even if your plugin doesn't get many users, being able to use a tool *you wrote* is very satisfying.

So if you've got an idea for a plugin that would make Vim better just sit down, learn about Vimscript, create it, and release it so we can all benefit.

If you have any questions or comments feel free to hit me up on Twitter. You might also enjoy following @dotvimrc where I try to tweet random, bite-sized lines you might like to put in your `.vimrc` file.