


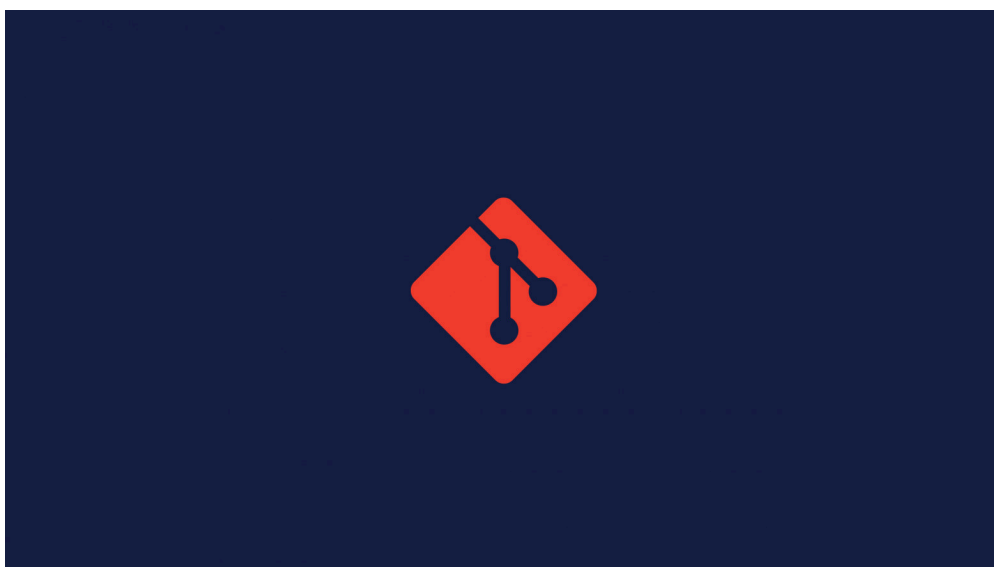


Git. Руководство по оформлению веток и КОММИТОВ

GitВведениеСтатей на тему что такое git и как им пользоваться на просторах интернета не мало. Я же хочу предложить вам несколько иной взгляд на привычные вещи, а...

 By Никита Алексов

 Jun 10, 2024 09:00 AM ·  7 мин. на чтение ·  [Посмотреть оригинал](#)



Введение

Статей на тему что такое git и как им пользоваться на просторах интернета не мало. Я же хочу предложить вам несколько иной взгляд на привычные вещи, а именно, на оформление веток и

коммитов, рассмотреть что такое WIP-коммиты, для чего они нужны и как с помощью них можно повысить свою продуктивность и поддерживать чистоту в истории вашего репозитория, в особенности, если вы работаете в команде. Поехали.

Оформление ветки

Создание ветки в git это одна из базовых операций, которую совершает разработчик для того, что бы отделить свой код от основной ветки проекта. Не смотря на то, что сама собой данная операция не представляет ничего сверхъестественного и сложность ее выполнения практически равна 0, проблемы начинаются на этапе выбора правильного наименования для будущей ветки.

Если вы любитель создавать названия по типу abcd, yourname, myawesomebranch, new-feature, test-1234 или, на худой конец, DEV-666, по номеру задачи в трекере, то спешу вас расстроить, все эти варианты далеки от идеала. По названию ветки должно быть понятно, что именно она делает, какая у нее основная функция. В этом нам поможет гибкая система типов и краткое именование поставленной задачи. Обо всем по порядку.

Формирование имени ветки

`<type>/<business-group>-<issue-number>-<short-description>`

В git можно создавать папки и подпапки. Достаточно добавить / в имени ветки и ваша ветка получит следующую структуру `folder/name`. Эта возможность помогает сортировать ветки в git. Правильность сортировки задает тип ветки, который ставится в начале ее названия:

- `build`: изменения, касающиеся процесса сборки(`npm`, `vite`);
- `chore`: изменения, не касающиеся кода напрямую, то что не увидит конечный пользователь(установка/удаление зависимостей, настройка проекта/инструментов);
- `ci`: изменения, касающиеся CI/CD;
- `docs`: изменения, касающиеся документации;
- `feat`: новая фича;
- `fix`: баг-фикс;
- `perf`: изменения, касающиеся улучшения производительности;
- `refactor`: изменения, не относящиеся ни к новой фиче, ни к фиксу бага;
- `revert`: отмена коммита;
- `style`: изменения, относящиеся к стилизации, форматированию;
- `test`: добавление недостающих тестов или корректирование уже существующих тестов.

Типы могут быть разными в зависимости от задач, которые вы или ваша команда разработки перед собою ставите. Они могут изменяться, дополняться, объединяться, но их присутствие в названии ветки является ее неотъемлемой частью.

После типа ветки идет /, затем через дефисы указывается либо бизнес-группа(DEV, SUP и т.д.) и номер вашей задачи из трекера + короткое описание задачи, либо просто задается короткое описание задачи. Само описание задачи должно составлять не более 5-6 слов и вносить ясность в то, что именно делает данная ветка("add custom input", "fix issue with user avatar dimensions", "upgrade vite version" и т.д.)

Примеры правильного именования веток:

feat/DEV-666-add-custom-input

fix/DEV-1125-issue-with-user-avatar-dimensions

chore/DEV-25-upgrade-vite-version

style/add-outline-to-primary-button

Оформление коммита

Ветка создана. Код написан. Что дальше? Создается коммит - снэпшот(от англ. snapshot - снимок) ваших наработок. Его сопровождает поясняющее сообщение, в котором описывается какая именно работа была проделана. Для того что бы быстро ориентироваться в общем потоке информации и сделать историю разработки как можно более прозрачной стоит уделить особое внимание составлению сообщения.

Формирование сообщения коммита

<type>: <short description>

Для начала посмотрите на типы, которые мы с вами ввели для именования веток и проведите анализ кода, который хотите закоммитить. Каждый коммит

должен содержать законченный, логически связанный код(исключением являются WIP-коммиты, об этом поговорим чуть позже). То есть если вы привыкли просто бездумно выполнять команду `git add .`, тем самым добавляя в коммит все что сделали в процессе работы, то сейчас самое время пересмотреть это действие.



Логические куски кода легко поддаются сортировке и выбор типа коммита становится тривиальной задачей. Вы добавили новую функциональность - это тип `feat`. Исправили баг - это тип `fix`. Провели рефакторинг кода без изменения основной логики работы - это тип `refactor`. И т.д.

После добавления типа коммита вы ставите `:`, пробел и пишете короткое обобщенное название того, что именно сделали. Само сообщения должно писаться в повелительном наклонении, начинаться, как правило, с глагола("добавить", "убрать", "исправить", "изменить" и т.д) и не содержать какие-либо знаки препинания в конце.

```
feat: add booking widget to product page
```

```
refactor: remove info button in tariff card
```

ci: update CI/CD for performance reason

Если вы хотите развернуто описать сделанную работу, то создайте большое сообщение для коммита с заголовком, телом сообщения и подвалом(опционально). В этом вам поможет команда `git commit`, которая открывает встроенный в оболочку командной строки редактор кода(vim, nano и т.д.). Вы также можете воспользоваться GUI-вариантом в вашей IDE.

HEADING

feat: add button for loading new posts

BODY

Add a button for loading new posts on blog page. It can be hidden if there're no more info and shown if it exists.

There is a small delay on button hiding(maybe it's a problem with rerendering), so it needs to be fixed on future refactoring.

FOOTER(optional)

Signed-of-by: John Doe

Issue: DEV-123

P.S. Если вы хотите чуть больше погрузиться в правильность написания сообщений для коммитов, то есть замечательное [соглашение](#), которое поддерживается силами сообщества. В нем вы сможете найти всю исчерпывающую информацию и дополнения к данному разделу.

P.P.S. Также хочу вам порекомендовать линтер [CommitLint](#) для проверки правильности оформления сообщений на основе заданных правил.

Git trailers

В подвале сообщения к коммиту может содержаться различная уточняющая информация именуемая трейлерами(от англ. trailer - прицеп), которая способна значительно упростить поиск конкретного коммита/ов.

Для облегчения процесса создания трейлеров вы можете настроить свои алиасы для ключей через конфиг git'a.

```
git config --global trailer.<keyAlias>.key '<full-key-name>'
```

Примеры добавления алиасов для ключей трейлеров:

```
git config --global trailer.issue.key 'Issue'
```

```
git config --global trailer.sign.key 'Signed-of-by'
```

```
git config --global trailer.milestone.key 'Milestone'
```

Для того что бы добавить трейлер для вашего коммита достаточно использовать опцию `--trailer 'keyAlias: <trailer-description>'` при создании сообщения.

```
git commit -m 'feat: add new counter' --trailer 'issue: DEV-111'
```

```
git commit --trailer 'sign: John Doe'
```

Для последующего поиска коммитов по трейлерам можно воспользоваться несколькими командами:

```
git log --grep="<full-key-name>: <your-search-info>"
```

Данная команда ищет все совпадения в логе git'a по полному имени ключа и описанию трейлера.

Примеры:

```
git log --grep="Issue: DEV-111"
```

```
git log --grep="Signed-of-by: John Doe"
```

Или

```
git log --format="%h %(trailers:key=<keyAlias>)" --grep="<your-search-info>"
```

Эта команда также ищет совпадения в логе git', но при этом использует алиас ключа трейлера.

Примеры:

```
git log --format="%h %(trailers:key=issue)" --grep="DEV-111"
```

```
git log --format="%h %(trailers:key=sign)" --grep="John Doe"
```

P.S. Для большего ознакомления с трейлерами и их возможностями есть [статья на git-scm](#). Также вы можете ознакомиться со статьей Брука Кульманна - [Git Trailers](#), где он подробно рассказывает про трейлеры с примерами ключей, которые он использует в своей работе.

WIP-коммиты

И вот мы плавно подошли к еще одной важной теме для познания дзен в процессе разработки - WIP-коммитам(от англ. Work In Progress - в процессе работы). Вы наверняка слышали это словосочетание и даже, возможно, пару раз создавали такие коммиты сами, но вряд ли до конца осознавали для чего они нужны и как ими правильно пользоваться.



Первое правило WIP-коммита: WIP-коммит не должен попадать в основную рабочую ветку.

Второе правило WIP-коммита: WIP-коммит НИКОГДА не должен попадать в основную рабочую ветку.

Третье правило WIP-коммита: WIP-коммит всегда создается и отправляется в вашу удаленную ветку в конце рабочего дня, если задача еще не завершена и нельзя оформить полноценный коммит.

WIP-коммит - это промежуточный коммит применяемый в процессе разработки. Его главная задача сохранить изменения до момента оформления полноценного коммита с конкретным типом. Данный коммит своего рода шаг в сторону безопасности на случай, если что-то случится с вашей рабочей машиной и локальный код потеряется.

Для создания такого коммита достаточно добавить вашему сообщению к коммиту тип WIP - и вуаля! -

ВАШ КОММИТ СТАЛ WIP'ОМ.

```
git commit -m 'WIP: <your-commit-message>'
```

Сложности начинаются на этапе когда вы накопили пачку таких WIP'ов и решили, что ваша задача теперь может считаться завершенной. Как сказано выше, WIP'ы не должны попасть в общую ветку. Одним из грамотных решений будет провести переоценку ваших коммитов через интерактивную перебазировку.

```
git rebase -i <branch-for-rebase>
```

Если вы работаете через командную строку, то далее вам откроется встроенный в нее редактор кода, где вас явно попросят указать, что именно делать с ранее созданными коммитами.

```
reword f7f3f6d WIP: working on new product card UI
```

```
fixup 310154e WIP: update card styles
```

```
fixup a5f4a0d WIP: update card skeleton
```

```
# Rebase 710f0f8..a5f4a0d onto 710f0f8
```

```
#
```

```
# Commands:
```

```
# p, pick <commit> = use commit
```

```
# r, reword <commit> = use commit, but edit the commit message
```

```
# e, edit <commit> = use commit, but stop for amending
```

```
# s, squash <commit> = use commit, but meld into previous commit
```

```
# f, fixup <commit> = like "squash", but discard this commit's  
log message
```

```
# x, exec <command> = run command (the rest of the line) using  
shell
```

```
# b, break = stop here (continue rebase later with 'git rebase --  
continue')
```

```
# d, drop <commit> = remove commit
```

```
# l, label <label> = label current HEAD with a name
```

```
# t, reset <label> = reset HEAD to a label
```

```
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
```

```
# .      create a merge commit using the original merge commit's
```

```
# .      message (or the oneline, if no original merge commit
```

```
was
# . specified). Use -c <commit> to reword the commit
message.
#
# These lines can be re-ordered; they are executed from top to
bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Для минимального использования вам достаточно
понять, что делают всего несколько команд:

- `pick`: взять данный коммит;
- `reword`: взять данный коммит, но отредактировать сообщение коммита;
- `squash`: взять данный коммит, но объединить с предыдущим коммитом;
- `fixup`: делает тоже самое, что и `squash`, но отменяет сообщение данного коммита;
- `drop`: удалить данный коммит.

Допустим вы хотите слить все три коммита в один. Первый коммит в списке("f7f3f6d") является самым ранним коммитом. Его можно взять за основу к слиянию. Рядом с ним вы прописываете команду `reword`, сообщение этого коммита будет позже изменено. Всем остальным коммитам вы прописываете команду `fixup`, они будут слиты в выбранный коммит("f7f3f6d").

После сохранения данных вы перейдете к этапу создания сообщения для вашего нового коммита. По

окончанию процесса вы получите один единственный коммит, в котором будут сохранены все ваши наработки.

P.S. Если в ходе разработки вы отправляли WIP-коммиты также в вашу удаленную ветку(настоятельно рекомендуется), а в локальной ветке у вас уже произошла перебазировка, то необходимо провести синхронизацию коммитов. Для этого можно воспользоваться командой `git push -force`. Эта команда заменит коммиты в вашей удаленной ветке на локальные.

P.P.S. Подробнее ознакомиться с интерактивной перебазировкой вы можете на соответствующих ресурсах([git-scm](#), [atlassian](#)).

Заключение

Умение правильно оформлять ветки и коммиты является важным навыком на пути к совершенствованию разработки. Стандартизируя данные процессы вы помогаете себе и другим членам команды быстрее и точнее определять информацию в общем потоке данных. Так система типов и короткое описание вносят уточнение в название ветки и заголовка сообщения к коммиту, `git trailers` расширяет информацию самого сообщения давая возможность указать номер задачи, проверяющего и т.д., а WIP-коммит с последующей перебазировкой помогает сохранить вашу промежуточную работу при этом не захламляя историю `git'a`.

Послесловие

Всем спасибо за прочтение данной статьи. Для сбора информации к ней я основывался на разных зарубежных источниках и своем опыте работы. Постарался собрать максимум данных в одном месте для того, чтобы получилось расширенное руководство, которого мне так не хватало все эти годы и, которое, уверен, поможет многим разработчикам, командам различной численности в независимости от стека технологий стандартизировать свои процессы по оформлению веток и коммитов в git.

Буду рад услышать ваш фидбэк в комментариях!