Дорожная карта

0. Приступаем
1. Введение в Git
2. Начало работы с Git
3. Просмотр истории
4. Ветвление
5. Слияние
6. Отмена изменений
7. Рабочий процесс

Дорожная карта (продолжение)

8. Работа в команде
9. Метки
10. Последние штрихи
11. Завершаем 👈

Последние штрихи

Подмодули

- Механизм, который позволяет использовать один проект в другом в виде подкаталога, но работать с ними по отдельности.
- Обычный пример - это разработка библиотеки (первый репозиторий) и использование её в приложении (второй репозиторий).
- Другой пример - это синхронизация нескольких проектов ("umbrella")

Добавление подмодуля

- Для добавления нового подмодуля используйте команду `git submodule add` с URL проекта, который вы хотите начать отслеживать.

```
$ git submodule add https://github.com/chaconinc/DbConnector
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

Добавление подмодуля

- Результат выполнения `git status`:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   .gitmodules
    new file:   DbConnector
```

Файл `.gitmodules`

Файл `.gitmodules` - это конфигурационный файл, в котором хранится соответствие между URL проекта и локальной поддиректорией, в которую вы его выкачали:

```
$ cat .gitmodules
[submodule "DbConnector"]
    path = DbConnector
    url = https://github.com/chaconinc/DbConnector
```

- Если у вас несколько подмодулей, то и в этом файле у вас будет несколько записей.
- Важно заметить, что этот файл добавлен под управление Git. Благодаря этому другие люди, которые клонируют ваш проект, узнают откуда взять подмодули проекта.

Как Git Видит подмодуль

```
$ git diff --cached DbConnector
diff --git a/DbConnector b/DbConnector
new file mode 160000
index 0000000..c3f01dc
--- /dev/null
+++ b/DbConnector
@@ -0,0 +1 @@
+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc
```

- Git распознаёт директорию как подмодуль и не отслеживает ее содержимое, когда вы не находитесь в этой директории. Вместо этого, Git видит ее как некоторый отдельный коммит из этого репозитория.
- Директория имеет особые права доступа 160000, которые означают, что вы сохраняете коммит как элемент каталога, а не как поддиректорию или файл.

Клонирование проекта с подмодулями

- Когда вы клонируете такой проект, по умолчанию вы получите директории, содержащие подмодули, но ни одного файла в них не будет.
- Вы должны выполнить две команды:
  - `git submodule init` – для инициализации локального конфигурационного файла, и
  - `git submodule update` – для извлечения всех данных этого проекта и переключения на соответствующий коммит, указанный в вашем основном проекте.

Работа над проектом с подмодулями

- Получение изменений из вышестоящего репозитория
- Работа с подмодулем
- Объединение изменений подмодуля

Получение изменений из вышестоящего репозитория

Простейший вариант использования подмодулей в проекте состоит в том, что вы просто получаете сам подпроект и хотите периодически получать обновления, но в своей копии проекта ничего не изменяете.

1. Если вы хотите проверить наличие изменений в подмодуле, вы можете перейти в его директорию, выполнить `git fetch` и затем `git merge` для обновления локальной версии из вышестоящего репозитория.
2. Если вы не хотите вручную извлекать и сливать изменения в поддиректорию, то для вас существует более простой способ сделать тоже самое. Если вы выполните `git submodule update --remote`, то Git сам перейдет в ваши подмодули, заберет изменения и обновит их для вас.

```
$ git submodule update --remote
$ git submodule update --remote DbConnector
```

Работа с подмодулем

Рассмотрим пример, в котором мы одновременно с изменениями в основном проекте внесем изменения в подмодуль, зафиксировав и опубликовав все эти изменения в одно и тоже время.

- `git submodule update` оставлял подрепозиторий в состоянии “detached HEAD”

Для упрощения работы с подмодулями вам необходимо сделать две вещи:

1. перейти в каждый подмодуль и переключиться на ветку, в которой будете в дальнейшем работать
2. необходимо сообщить Git, что ему делать если вы внесли изменения, а затем командой git submodule update --remote получаете новые изменения из репозитория:

- вы можете слить их в вашу локальную версию
- или попробовать перебазировать ваши локальные наработки поверх новых изменений.

Пример

Первым делом, давайте перейдем в директорию нашего подмодуля и переключимся на нужную ветку.

```
$ git checkout stable
```

При обновлении:

- если ничего не указывать, то подрепозиторий перейдёт в "detached HEAD":

```
$ git submodule update --remote
Submodule path 'DbConnector': checked out '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

- если нужно слить изменения:

```
$ git submodule update --remote --merge
```

- если нужно перебазировать изменения:

```
$ git submodule update --remote --rebase
```

Публикация изменений в подмодуле

- Сделаны изменения, но не опубликованы.
- Нельзя отправлять изменения в суперпроекте без предварительного обновления подмодулей на сервере.
- Команда `git push` имеет вспомогательный параметр `--recurse-submodules=(check|on-demand)`.

`git push --recurse-submodules=check`

Использование значения `check` приведет к тому, что push просто завершится неудачей, если какой-то из зафиксированных подмодулей не был отправлен на сервер:

```
$ git push --recurse-submodules=check
The following submodule paths contain changes that can
not be found on any remote:
  DbConnector

Please try

    git push --recurse-submodules=on-demand

or cd to the path and use

    git push

to push them to a remote.
```

`git push --recurse-submodules=on-demand`

Другой вариант – это использовать значение `on-demand`, которое попытается сделать это все за вас:

```
$ git push --recurse-submodules=on-demand
Pushing submodule 'DbConnector'
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 3), reused 0 (delta 0)
To https://github.com/chaconinc/DbConnector
   c75e92a..82d2ad3  stable -> stable
Counting objects: 2, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 266 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)
To https://github.com/chaconinc/MainProject
   3d6d338..9a377d1  master -> master
```

Полезные советы для работы с подмодулями

- Команда `foreach`, которая позволяет выполнить произвольную команду в каждом подмодуле:

```
$ git submodule foreach 'git checkout -b featureA'
```

- Использовать псевдонимы:

```
$ git config alias.sdiff '!'"git diff && git submodule foreach 'git diff'"
$ git config alias.spush 'push --recurse-submodules=on-demand'
$ git config alias.supdate 'submodule update --remote --merge'
```

Перехватчики Git

- в Git есть возможность запускать собственные сценарии в те моменты, когда происходят некоторые важные действия.
- Существуют две группы подобных перехватчиков (hook): на стороне клиента и на стороне сервера.
- Перехватчики на стороне клиента предназначены для клиентских операций, таких как создание коммита и слияние.
- Перехватчики на стороне сервера нужны для серверных операций, таких как приём отправленных коммитов.
- Перехватчики могут быть использованы для выполнения самых различных задач.
- Их часто используют, чтобы обеспечить соблюдение определённых стандартов, хотя важно отметить, что данные сценарии не передаются при клонировании.

Установка перехватчика

- Все перехватчики хранятся в подкаталоге hooks в .git.
- По умолчанию Git заполняет этот каталог кучей примеров сценариев
- Чтобы активировать сценарий-перехватчик, положите файл в подкаталог hooks в Git-каталоге, дайте ему правильное имя и права на исполнение.
- Чтобы активировать примеры перехватчиков, их следует переименовать, удалив их расширение .sample.

Перехватчики на стороне клиента

Перехватчики для работы с коммитами:

- `pre-commit`
- `prepare-commit-msg`
- `commit-msg`
- `post-commit`

Эти сценарии призваны помочь разработчикам, и это обязанность разработчиков установить и сопровождать их, хотя разработчики и имеют возможность в любой момент подменить их или модифицировать.

Перехватчик `pre-commit`

- запускается перед тем, как вы наберёте сообщение коммита.
- используют, например,
  - что вы запустили тесты
  - проверить стиль кодирования (запускать lint или что-нибудь аналогичное)
  - проверить наличие пробельных символов в конце строк (перехватчик по умолчанию занимается именно этим)
  - или проверить наличие необходимой документации для новых методов.
- Завершение перехватчика с ненулевым кодом прерывает создание коммита
  - пропустить перехватчик можно с помощью `git commit --no-verify`.

Перехватчик `prepare-commit-msg`

- запускается до появления редактора с сообщением коммита, но после создания сообщения по умолчанию.
- Он позволяет отредактировать сообщение по умолчанию перед тем, как автор коммита его увидит.
- У этого перехватчика есть несколько опций:
  - путь к файлу, в котором сейчас хранится сообщение коммита
  - тип коммита
  - SHA-1 коммита (если в коммит вносится правка с помощью git commit --amend).

Как правило, данный перехватчик не представляет пользы для обычных коммитов; он скорее хорош для коммитов с автогенерируемыми сообщениями, такими как шаблонные сообщения коммитов, коммиты-слияния, уплотнённые коммиты (squashed commits) и коммиты с исправлениями (amended commits). Данный перехватчик можно использовать в связке с шаблоном для коммита, чтобы

Перехватчик `commit-msg`

- принимает один параметр - путь к временному файлу, содержащему текущее сообщение коммита.
- Когда сценарий завершается с ненулевым кодом, Git прерывает процесс создания коммита. Так что можно использовать его для проверки состояния проекта или сообщений коммита (соответствует требуемому шаблону) перед тем, как его одобрить.

Перехватчик `post-commit`

- После того, как весь процесс создания коммита завершён, запускается перехватчик post-commit.
- Он не принимает никаких параметров, но вы с лёгкостью можете получить последний коммит, выполнив `git log -1 HEAD`.
- Как правило, этот сценарий используется для уведомлений или чего-то в этом роде.

Перехватчики для работы с e-mail

Перехватчики для работы с коммитами:

- `applypatch-msg`
- Используются для рабочих процессов, основанных на электронной почте.
- Они вызываются командой `git am`.

Перехватчик `applypatch-msg`

- Первый запускаемый перехватчик.
- Принимает один аргумент - имя временного файла, содержащего предлагаемое сообщение коммита.
- Git прерывает наложение патча, если сценарий завершается с ненулевым кодом.
- Как правило, проверяет, что сообщение коммита правильно отформатировано или, чтобы нормализовать сообщение, отредактировав его на месте из сценария.

Перехватчик `pre-applypatch`

- Второй запускаемый перехватчик.
- У него нет аргументов, и он запускается после того, как патч наложен, поэтому его можно использовать для проверки снимка состояния перед созданием коммита.
- Можно запустить тесты или как-то ещё проверить рабочее дерево с помощью этого сценария. Если чего-то не хватает, или тесты не пройдены, выход с ненулевым кодом так же завершает сценарий `git am` без применения патча.

Перехватчик `post-applypatch`

- Третий (последний) запускаемый перехватчик.
- Его можно использовать для уведомления группы или автора патча о том, что вы его применили.
- Этим сценарием процесс наложения патча остановить уже нельзя.

Другие клиентские перехватчики

- `pre-rebase`
- `post-checkout`
- `post-merge`

Перехватчик `pre-rebase`

- Запускается перед перемещением чего-либо, и может остановить процесс перемещения, если завершится с ненулевым кодом.
- Этот перехватчик можно использовать, чтобы запретить перемещение любых уже отправленных коммитов.
- Пример перехватчика `pre-rebase`, устанавливаемый Git, это и делает, хотя он предполагает, что ветка, в которой вы публикуете свои изменения, называется `next`. Вам, скорее всего, нужно будет заменить это имя на имя своей публичной стабильной ветки.

Перехватчик `post-checkout`

- Запускается после успешного выполнения команды git checkout
- Его можно использовать для того, чтобы правильно настроить рабочий каталог для своей проектной среды. Например:
  - перемещение в каталог больших бинарных файлов, которые вам не хочется включать под версионный контроль
  - автоматическое генерирование документации

Перехватчик `post-merge`

- Запускается после успешного выполнения команды merge.
- Его можно использовать, например
  - для восстановления в рабочем дереве данных, которые Git не может отследить, таких как информация о правах
  - проверить наличие внешних по отношению к контролируемым Git файлов, которые вам нужно скопировать в каталог при изменениях рабочего дерева.

Заключение