

Система контроля версий Git — advanced

Методичка к уроку 1

Работа с удалёнными репозиториями





Введение	2
Обзор систем GitHub и GitLab	3
Подключение к удалённому репозиторию	9
Синхронизация изменений в ветка	14
Настройки связей с удалёнными репозиториями	17
Разрешение типичных конфликтов	20
Домашнее задание	6
Что можно почитать еще?	6
Используемая литература	6

Введение

Вы уже знакомы с системой контроля версий Git, и в предыдущем курсе, состоявшем из трёх лекций, попробовали этот инструмент на практике, познакомились с основными командами, которые есть в Git, научились работать с ветками, добавлять их в свой репозиторий, переключаться между ними и удалять, сливать их и разрешать конфликты слияния, а также просматривать историю изменений в ветках репозитория.

Кроме того, вы познакомились с системой GitHub, научились клонировать из него репозитории, вносить в них изменения и отправлять их обратно, а также создавать pull request'ы — по сути, запросы на вливание проделанной вами работы в основную ветку репозитория.

В этом курсе вас ждёт углублённое знакомство с системой контроля версий Git. Курс также состоит из трёх лекций, после каждой из которых вы пройдёте семинарское занятие и закрепите полученные в лекции знания на практике. В первой сегодняшней лекции мы углубимся в работу с удалёнными репозиториями.

Во второй лекции мы поговорим о работе с изменениями: о том, какие бывают изменения в Git, каковы их жизненный цикл, как их просматривать на разных стадиях этого жизненного цикла, как отменять сохранённые и



несохранённые изменения, как откладывать изменения и даже как отменять слияние веток.

В третьей лекции мы рассмотрим две наиболее популярные в настоящее время модели работы с ветками, посмотрим, как выглядит процесс работы с Git в команде целиком, а также познакомимся с некоторыми наиболее важными общепринятыми правилами работы с Git. Кроме того, мы с вами сравним работу с Git в командной строке и в разных средах разработки.

Темы, которые вас ждут в этой лекции:

- Обзор систем GitHub и GitLab, сравнение их функционала и особенностей
- Подключение к удалённому репозиторию: изучим несколько способов
- Синхронизация изменений между локальными и удалёнными ветками
- Настройки связей с удалёнными репозиториями
- Разрешение типичных конфликтов при работе с удалёнными репозиториями

Обзор систем GitHub и GitLab

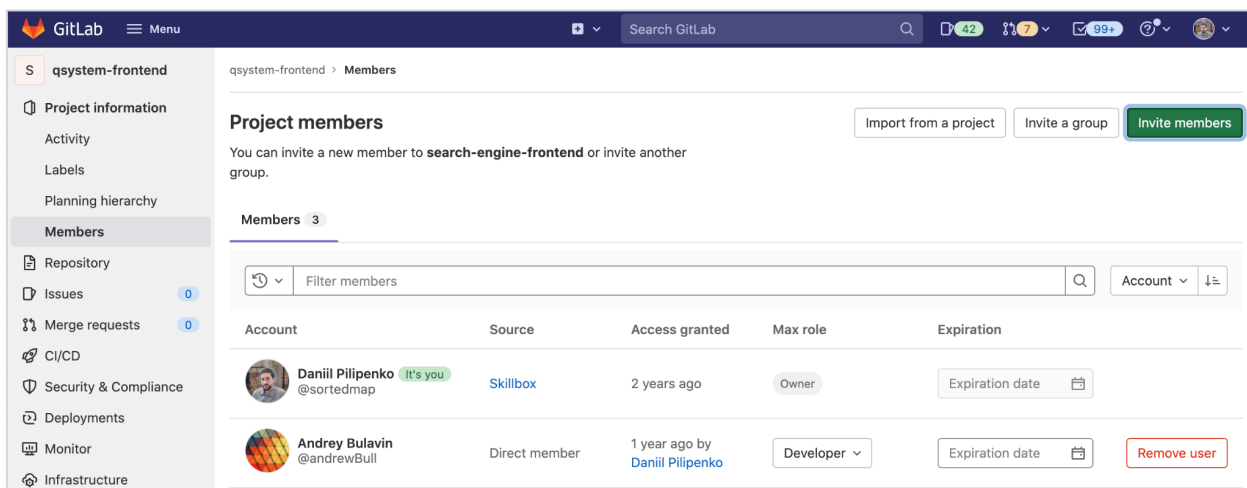
Системы управления Git-репозиториями GitHub и GitLab — не единственные в своём роде, но самые популярные, поэтому знакомиться мы будем именно с ними. С системой GitHub вы уже поверхностно знакомы, поэтому основная задача этого блока — сравнить эти системы и показать вам их главные отличия. Кроме того, все скриншоты будут показаны на примере системы GitLab. Начнём мы с того, для чего эти системы нужны.

Самое главное назначение таких систем — это **хранение удалённых репозиторияв**. Системы GitHub и GitLab как раз предоставляют возможность такого хранения.

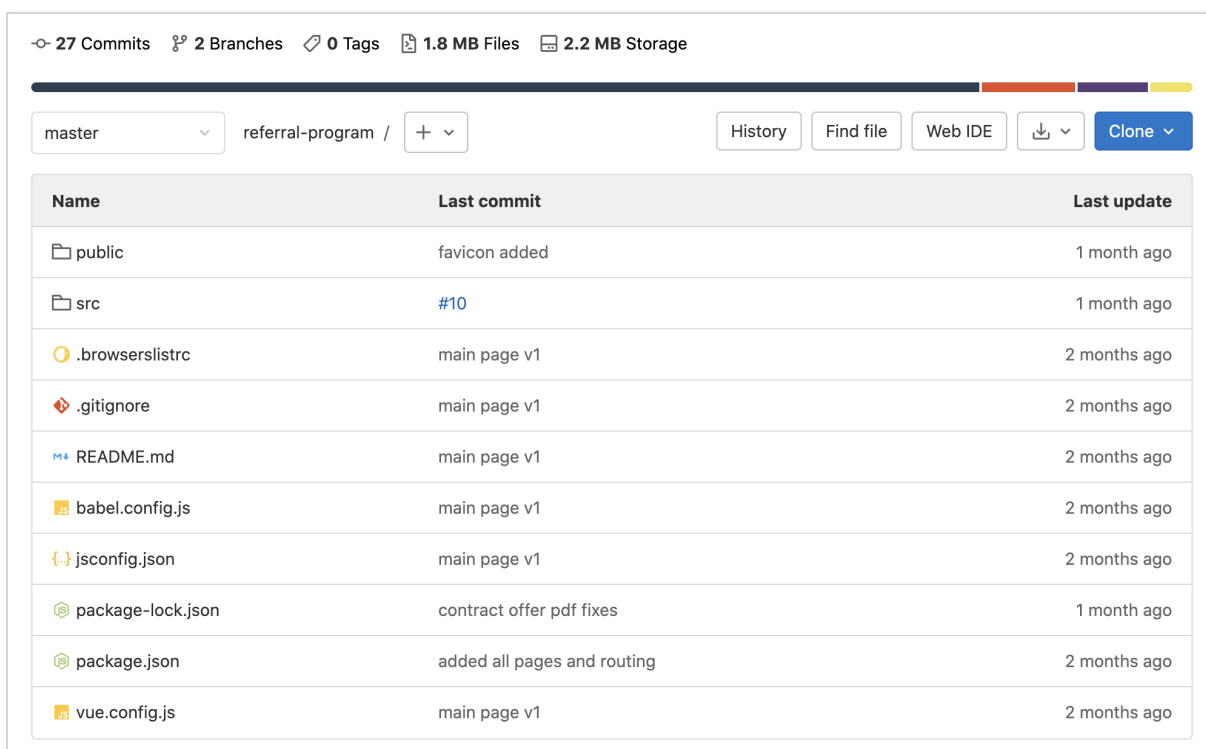
Второе, чем удобны такие системы — это возможностью **расширенного управления доступами к репозиторию**. Конечно, репозиторий можно просто положить на сервер и к нему подключаться безо всяких дополнительных систем, но вот сделать так, чтобы доступ к нему имели только определённые люди, будет уже сложнее, и системы GitHub и GitLab успешно решают эту задачу.



Вы можете приглашать к работе с репозиторием отдельных пользователей с разными правами — например, правами только на чтение или на внесение изменений, или даже на управления доступами, как у вас, а также просматривать список существующих пользователей, удалять их из него, менять их права или ограничивать доступ по времени. В GitLab интерфейс управления доступами к репозиторию выглядит следующим образом:



Ещё одно важное преимущество таких систем, как GitHub и GitLab — это возможность **наглядного представления репозитория** и всех изменений в них. В частности, наглядного представления файлов и папок репозитория:





Причём в GitHub и GitLab есть возможность просмотра не только последней версии репозитория, но и любой версии на момент любого коммита. Также есть возможность просматривать всю историю коммитов, просматривать изменения между двумя коммитами и сравнивать как структуру файлов и папок, так и отдельные файлы построчно в удобных интерфейсах:

The screenshot shows the GitHub Compare interface. At the top, it says 'Compare dev and latest version'. Below this is a search bar 'Search files (⌘P)'. On the left, a file tree shows the directory structure: 'src/main/resources' > 'static/assets' > 'css'. The file 'style.css' is selected, showing a diff of +32 lines and -14 lines. The right pane shows the diff for 'style.css', with line numbers 714 to 728. The diff shows changes to the CSS file, including the addition of a new selector '.form_label-5' and modifications to '.form_label-2'.

В таких системах также обычно встроены **таск-трекеры** — системы управления задачами. Вот фрагмент перечня задач из системы GitLab:

The screenshot shows the GitLab Issues page. At the top, there are tabs for 'Open' (37), 'Closed' (250), and 'All' (287). Below this is a search bar 'Search or filter results...' and a 'Recent searches' dropdown. The main content area shows a list of issues. Each issue has a title, a number, a creation date, and a creator. The issues are:

- Ведомость начислений** (#285) - created 6 days ago by Anton Semenov - updated 4 days ago
- Поле Категория расходов при создании расхода** (#288) - created 6 days ago by Anton Semenov - updated 4 days ago
- Админские права пользователей аккаунта, Доработка настроек** (#281) - created 1 week ago by Anton Semenov - updated 5 days ago
- Доработки 03.08.2022** (#284) - created 1 week ago by Anton Semenov - updated 5 days ago (marked as **СРОЧНО**)
- Выплаты** (#287) - created 6 days ago by Anton Semenov - updated 5 days ago



Встроенные task-трекеры позволяют ставить задачи конкретным пользователям, писать по ним комментарии, переписываться с этими пользователями по этим задачам, обсуждать их, прикреплять к ним файлы и, что очень важно и удобно, — автоматически связывать конкретные изменения в коде с этими задачами. Вот здесь видно, что коммиты автоматически отображаются в перечне комментариев по задаче:

Open Backend. Не выводить на фронт ошибки в явном виде

Alexey Petrov @alexey.fortis mentioned in commit [221ad77c](#) 1 month ago

Alexey Petrov @alexey.fortis mentioned in commit [3387f185](#) 1 month ago

Alexey Petrov @alexey.fortis mentioned in commit [7bdfca86](#) 1 month ago

Alexey Petrov @alexey.fortis mentioned in commit [8a401e26](#) 1 month ago

Alexey Petrov @alexey.fortis mentioned in commit [99320d6b](#) 1 month ago

Alexey Petrov @alexey.fortis mentioned in commit [ae1a5c58](#) 1 month ago

Daniil Pilipenko @sortedmap assigned to [@alexey.petrov](#) and unassigned [@sortedmap](#) 1 month ago

Daniil Pilipenko @sortedmap · 1 month ago

Author Owner 😊 💬 ✎ ⋮

Правильно ли я понимаю, что все ошибки обязательно логируются?

Может, будем выводить какой-то код ошибки, по которому её в логе потом найти было можно? И попросим юзеров сообщать нам код ошибки?

И ещё одна удобная особенность таких систем — **возможность комментировать** и даже **обсуждать** как отдельные строки кода в отдельных файлах, так и любые другие изменения:

445 442 @@ -445,8 +442,7 @@ function canSaveAsNew() {
446 443 btn_new_order.classList.remove('btn_green');
447 444 }
447 444 } else {

Daniil Pilipenko @sortedmap · just now

Owner ✅ 😊 💬 ✎ ⋮

Слишком большая вложенность кода

Reply...

Resolve thread

Start a new discussion...

448 449 - canSave()
449 445 + canSave();
450 446 }
451 447 } else {
452 448 if (btn_new_order) {

6



На изображении выше виден комментарий к отдельной строке кода, на который разработчик может ответить — либо возразить, либо учесть его в своей работе и внести дополнительные изменения в код, либо вообще его отклонить.

Таким образом, можем выделить следующие преимущества систем управления Git-репозиториями:

- Хранение удалённых репозиториях
- Расширенное управление доступами
- Наглядное представление репозиториях
- Управление задачами
- Связь изменений с задачами
- Обсуждение изменений в коде

Конечно, это не все преимущества таких систем. В них есть ещё много всего другого. Если вы уже начали пользоваться этими системами, то наверняка уже никогда не перестанете этого делать из-за удобства :)

Давайте теперь посмотрим на ключевые отличия GitHub и GitLab, чтобы вы могли легко определить, какую из них следует использовать для того или иного проекта:

Отличие	GitHub	GitLab
Возможность размещения на своём сервере	Нет	Да
Бесплатные возможности для командной работы и DevOps	Нет	Да
Возможность доработок	Нет	Да

Первое их отличие — это возможность размещать на своём сервере. GitHub размещать на своём сервере нельзя, а GitLab — можно. Для многих проектов это критичное отличие, поскольку очень часто программный код проекта является коммерческой тайной.

Здесь есть плюсы и минусы. Например, GitHub очень удобно использовать для так называемых open source проектов — проектов с открытым исходным



кодом. Чтобы любой желающий мог этот код посмотреть, использовать или внести в него свои улучшения.

Также GitHub отлично подходит для размещения своих личных небольших проектов, которые вы хотите демонстрировать другим людям, в том числе, потенциальным работодателям.

С другой стороны, для размещения кодов проектов, которые принадлежат компании, использовать GitHub не рекомендуется. Во-первых, это может нарушать политики безопасности, которые в компании приняты. А во-вторых, внешний ресурс, даже если это GitHub, может перестать работать в самый неподходящий момент, как из-за собственных проблем, так и из-за проблем с Интернетом.

Это, конечно, редкая история, но существуют компании, где час или день простоя сотрудников или работы каких-то проектов может стоить миллионы рублей или даже долларов, и в таких компаниях зависимость от всех внешних обстоятельств необходимо снижать до минимума.

В то же время, если вам нужно быстро начать пользоваться системой управления репозиториями, и нет ни времени, ни ресурсов на разворачивание GitLab на своём сервере, вы можете воспользоваться GitHub.

Второе важное отличие этих систем связано с первым. GitHub предлагает всякие полезные опции, доступные для коллективной, командной разработки и корпоративных проектов, в основном платно, в то время как в GitLab они доступны в бесплатной версии.






GitLab в бесплатной версии больше подходит для организации процессов командной работы, подключения различных DevOps-инструментов, всякой автоматизации и разнообразных тонких настроек. В GitHub функциональность, связанная с DevOps, более слабая и реализуется, в основном, при помощи сторонних инструментов.

Обе эти системы достаточно мощные по функционалу, и у обеих из них есть как бесплатные, так и платные тарифы. Они часто меняются, поэтому мы их с вами рассматривать детально не будем. Но вы всегда можете сами перейти на их официальные сайты и сравнить.



Третье отличие также связано с первым и состоит в том, что исходный код GitLab доступен и открыт, и GitLab можно дорабатывать под нужды компании и проекта, а у GitHub таких возможностей нет, поскольку он размещается только на серверах самого GitHub-а.

Кроме того, между этими системами есть большое количество других точечных отличий, поскольку системы сами по себе достаточно разные. Но эти отличия — ключевые.

GitHub	GitLab
 Open-source-проекты	 Коммерческие проекты
 Публичный доступ	 Ограниченный доступ
 На серверах GitHub	 На собственном сервере
 Нет автоматизации DevOps-процессов	 Автоматизация DevOps-процессов

Таким образом, если вы хотите размещать коды open-source-проектов, собираетесь демонстрировать коды своих проектов другим людям, например, будущим работодателям, или же не имеете возможности развернуть систему на своём сервере, лучше всего вам подойдёт GitHub.

Если же вы собираетесь работать над проектом в команде, над проектом, доступ к которому должен быть ограничен и предоставлен узкому кругу лиц и хотите с помощью этой системы автоматизировать DevOps-процессы, вам следует использовать GitLab.

Подключение к удалённому репозиторию

Теперь давайте рассмотрим несколько способов подключения к удалённым репозиториям в зависимости от того, пустой ли удалённый ли репозиторий или нет, и создан ли у вас локальный репозиторий. Здесь возможно три варианта:



Локальный репозиторий	Удалённый репозиторий
Нет	Есть
Нет	Нет
Есть	Нет

Первый вариант подключения к удалённому репозиторию применяется в ситуации, когда у вас ещё нет локального, а удалённый есть и он не пустой, в нём уже лежат какие-то файлы. Это происходит всегда, когда вы начинаете работу над уже существующим проектом, и с этим способом вы уже знакомы — это команда `git clone`. Вот пример выполнения такой команды:

```
git clone git@github.com:sortedmap/git-advanced.git
```

Выполнение этой команды приведёт к тому, что в папке, в которой вы её выполняете, появится папка “git-advanced”, содержащая локальную копию данного репозитория. Напомним, что эту копию ещё называют рабочей копией.

Если вы будете вносить в такую рабочую копию какие-то изменения, вы сможем сразу их отправлять на удалённый сервер в исходный репозиторий, поскольку они уже автоматически связаны.

Давайте теперь рассмотрим другой вариант, в котором вы начинаете работать над новым проектом, и у вас ещё нет ни локального, ни удалённого репозитория. В этом случае вам нужно сначала создать удалённый репозиторий.

Вы можете сразу добавить в него какие-либо файлы и просто клонировать его к себе на компьютер, как было показано выше, а можете оставить его пустым. Давайте рассмотрим оба этих варианта.


Добавлять файлы в новый репозиторий вы можете разными способами. Для этого в системах GitHub и GitLab предусмотрено множество удобных инструментов. Во-первых, при создании репозитория вы можете в разделе “Initialize this repository with” выбрать какой-нибудь файл, который будет сразу добавлен в репозиторий, который вы создаёте:



Create a new repository


A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).


Owner * **Repository name ***

 sortedmap / new-repo ✓

Great repository names are short and memorable. Need inspiration? How about [bug-free-chainsaw?](#)

Description (optional)

☒  **Public**
Anyone on the internet can see this repository. You choose who can commit.

☐  **Private**
You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.

☐ **Add a README file**
This is where you can write a long description for your project. [Learn more](#).

Add .gitignore
Choose which files not to track from a list of templates. [Learn more](#).


.gitignore template: None ▾

Choose a license
A license tells others what they can and can't do with your code. [Learn more](#).

License: None ▾

Второй вариант — создать новый файл или загрузить существующий в уже созданный репозиторий:

Quick setup — if you've done this kind of thing before

 Set up in Desktop or **HTTPS** **SSH** `git@github.com:sortedmap/new-repo.git`

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

Таким образом вы сделаете так, что удалённый репозиторий будет уже непустой, и, как уже было сказано выше, настроить с ним связь можно будет просто



выполнив команду “git clone”. Но можно сделать иначе. В GitHub и GgitLab обычно даже описан этот вариант:

...or create a new repository on the command line

```
echo "# new-repo" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M master
git remote add origin git@github.com:sortedmap/new-repo.git
git push -u origin master
```

Здесь рекомендуется создать у себя на компьютере в некой папке файл README.md с текстом, инициализировать в этой папке репозиторий командой “git init”, добавить этот первый файл в коммит и закоммитить его, поменять главную ветку репозитория на “master”, а затем командой установить в созданном вами репозитории связь с удалённым репозиторием командой:

```
git remote add origin git@github.com:sortedmap/new-repo.git
```

И следующей командой отправить изменения, сделанные в локальном репозитории, в удалённый командой:

```
git push -u origin master
```

Это был вариант, при котором у вас изначально нет ни удалённого, ни локального репозитория.

Повторим ещё раз: в этом случае вы создаёте сначала удалённый, а дальше либо добавляете в него новый файл и клонируете на свой компьютер, либо следуете инструкции: создаёте локальный репозиторий, подключаете его к удалённому и отправляете все изменения из локального в удалённый. Оба этих способа абсолютно равноценны и приводят к одному и тому же результату.



Следующий вариант: у вас уже есть локальный репозиторий с проектом, а удалённого репозитория ещё нет. В этом случае первым действием вам нужно создать пустой удалённый репозиторий, не добавляя в него никакие файлы.

Имя репозитория лучше задать такое, чтобы оно совпадало с именем проекта. Это необязательно, но это правило хорошего тона. Если имя проекта содержит заглавные буквы, то в имени репозитория пишем строчные, а слова разделяем дефисами. Например, если проект у вас называется SpecialApp, то репозиторий лучше назвать special-app.

В GitHub и GitLab после создания пустого репозитория есть вторая инструкция, как раз на случай, если у вас локально уже есть репозиторий. Она выглядит следующим образом:

...or push an existing repository from the command line

```
git remote add origin git@github.com:sortedmap/new-repo.git
git branch -M master
git push -u origin master
```

С этими командами вы уже знакомы: сначала вы связываете локальный репозиторий с удалённым, затем меняете главную ветку на “master” и отправляете данные из локального репозитория в удалённый.

Давайте теперь повторим команды, которые мы с вами изучили в рамках этой темы:

- `git init` — команда инициализации репозитория в текущей папке. Фактически, при её выполнении в текущей папке создаётся папка “.git” и в ней сохраняется информация о текущем состоянии всех файлов и в будущем обо всех изменениях;
- `git clone path` — команда, которая клонирует удалённый репозиторий *path* на ваш компьютер, создавая так называемую локальную копию, которая уже сразу автоматически связана с удалённым репозиториум, из которого она была получена;



- `git remote add origin path` — команда, которая устанавливает в настройках локального репозитория путь к удалённому репозиторию *path*. Давайте разберём эту команду по словам:
 - `git` — это, собственно, программа “git”;
 - `remote` — это указание этой программе на то, что сейчас мы будем выполнять какую-то операцию с настройками удалённых репозиториях, таких операций и настроек может быть несколько;
 - `add` — означает, что мы добавляем новую запись в эти настройки;
 - `origin` — это название удалённого репозитория по умолчанию, переводится с английского как “источник” или “начало”, ведь фактически удалённый репозиторий при работе в команде является основным — источником кода проекта и всех новых изменений, попадающих в него от разных разработчиков;
 - `path` — путь к репозиторию, который мы добавляем в настройки локального репозитория в качестве этого источника — репозитория с именем “origin”.
- `git push -u origin master` — команда, которая отправляет изменения из локального репозитория на сервер. В ней написан параметр “-u”, который говорит о том, что необходимо установить связь текущей ветки в локальном репозитории с соответствующей веткой в удалённом таким образом, чтобы вы могли дальше выполнять команду “git push” и некоторые другие команды без дополнительных параметров.

Синхронизация изменений в ветках

Теперь давайте поговорим о синхронизации изменений в разных ветках между удалённым и локальным репозиториями. Начнём мы с изучения команды:

```
git remote show origin
```

Эта команда позволяет просмотреть, какие локальные ветки связаны с ветками удалённого репозитория и каков их статус. Эта команда выводит информацию в следующем формате:



```
1 * remote origin
2  Fetch URL: git@github.com:sortedmap/git-advanced.git
3  Push  URL: git@github.com:sortedmap/git-advanced.git
4  HEAD branch: master
5  Remote branches:
6    1-some-issue tracked
7    master      tracked
8  Local branches configured for 'git pull':
9    1-some-issue merges with remote 1-some-issue
10   master      merges with remote master
11  Local refs configured for 'git push':
12    1-some-issue pushes to 1-some-issue (fast-forwardable)
13    master      pushes to master      (up to date)
```

В начале этого вывода написано “* remote origin”, что указывает на то, что эта информация относится к состоянию локального репозитория относительно удалённого репозитория origin. Далее указаны пути к этому репозиторию для операций fetch и push и то, что основная ветка — master.

Затем идёт блок информации о состоянии веток. Вначале перечислены удалённые ветки и информация о том, что они отслеживаются локальными ветками (в примере выше — пометка “tracked”).

После этого указан перечень веток и их конфигурация относительно удалённых веток для команды git pull. В примере выше, в частности, написано, что локальные ветки будут сливаться с удалёнными ветками при выполнении команды git pull.

И в конце блока идёт конфигурация локальных веток для команды git push. Здесь отображается информация о том, в какие удалённые ветки будет отправляться информация из локальных и каков статус локальных веток. Статус “up to date” говорит о том, что локальная и удалённая ветки синхронизированы. Статус “fast-forwardable” сообщает нам о том, что в локальной ветке произошли изменения и мы их можем отправить в удалённую ветку.

Здесь также может быть статус “local out of date”, который говорит о том, что локальная ветка отстаёт от удалённой: в удалённом репозитории в этой ветке произошли какие-то изменения, которых ещё нет локально.



Вы можете скачать все изменения из ветки удалённого репозитория в локальный, но без обновления самой ветки обновлять не будем. Это делается при помощи команды:

```
git fetch
```

После её выполнения вы можете посмотреть, какие изменения были внесены в удалённую ветку, для этого можно воспользоваться командой `git log`, в параметрах которой указав имена удалённой и локальной веток:

```
git log origin/1-issue ^1-issue
```

Такой вариант команды выведет те коммиты, которые отличают удалённую ветку от локальной. Содержимое каждого такого коммита вы можете просмотреть командой:

```
git diff hash
```

В данном случае, `hash` — это хэш коммита. Если вы захотите влить удалённую ветку в свою локальную, вы можете выполнить уже известную вам команду:

```
git pull
```

Эту команду также можно выполнять для отдельных веток, указав имя удалённого репозитория и название нужной ветки:

```
git pull origin 1-issue
```

Таким образом, команда `git pull` затягивает изменения из удалённого репозитория и сливает их с изменениями в локальном репозитории, в то время как команда `git fetch` их просто затягивает, но слияния не производит.



Настройки связей с удалёнными репозиториями

Ранее мы показывали, как подключаться к удалённому репозиторию. Это можно делать командой `git clone`, а можно при помощи команды:

```
git remote add origin path
```

Давайте вспомним, из чего состоит эта команда:

- `git` — это, собственно, программа “git”;
- `remote` — это указание этой программе на то, что сейчас мы будем работать с настройками удалённых репозитория;
- `add` — означает, что мы добавляем новый удалённый репозиторий;
- `origin` — это название удалённого репозитория, который мы добавляем;
- `path` — путь к репозиторию, который мы добавляем в настройки локального репозитория в качестве репозитория с именем “origin”.

С помощью команды `git remote` мы также просматривали информацию об этом удалённом репозитории, добавив после неё команду `show` и имя `origin`:

```
git remote show origin
```

Теперь давайте изучим другие команды, с помощью которых можно управлять связями с удалёнными репозиториями. Первое, что хотелось бы показать, это команду, с помощью которой вы можете вывести информацию о путях к вашим удалённым репозиториям:

```
git remote -v
```

Вывод этой команды обычно выглядит следующим образом:

```
1 origin  git@github.com:sortedmap/git-advanced.git (fetch)
2 origin  git@github.com:sortedmap/git-advanced.git (push)
```



При необходимости вы можете заменить репозиторий `origin`. Например, поменять путь к этому репозиторию. Для этого сначала нужно удалить текущий репозиторий:

```
git remote remove origin
```

И затем снова добавить новый удалённый репозиторий:

```
git remote add origin path
```

Вы также можете поменять путь к этому репозиторию, если захотите:

```
git remote set-url origin new-path
```

Кроме того, вы можете поменять пути для каждой из операций — `fetch` и `push` — отдельно. Например, для операции `push` команда изменения пути будет такой:

```
git remote set-url --push origin new-path
```

Система контроля версий Git также позволяет добавить к своему локальному репозиторию несколько удалённых. Это может быть необходимо в ряде случаев, например, если вы работаете с репозиторием своей команды и с репозиторием некоего условного поставщика программного обеспечения. Давайте посмотрим, как это может работать. Вначале добавим к нашему репозиторию ещё один удалённый:

```
git remote add vendor vendor-repo-path
```

После такого действия вы сможете отправлять изменения из своего локального репозитория в какой-то один удалённый репозиторий или сразу в оба. Для отправки изменений в каждый репозиторий отдельно вам нужно просто выполнить последовательно две команды:

```
git push -u origin master  
git push -u vendor master
```

При этом затянуть изменения из всех репозиториях в свой локальный можно одной командой. Но это можно сделать только без слияния, потому что



возможны конфликты не только кода из удалённого репозитория и кодом из локального, но и кодов из двух удалённых репозиториях между собой:

```
git fetch --all
```

После такого действия вы сможете просматривать затянутые изменения и сравнивать удалённые ветки с ветками своего локального репозитория:

```
git log origin/master ^master  
git log vendor/master ^master
```

При необходимости вы можете также вливать нужные вам изменения командой `git merge`, например:

```
git merge origin/master
```

или:

```
git merge vendor/master
```

А теперь расскажем вам о способе, который позволяет сделать так, чтобы однократный “`git push`” отправлял изменения сразу в несколько репозиториях.

Для этого нужно придумать название некоего виртуального репозитория, который будет в себе содержать пути ко всем удалённым. Пусть он называется “all”. Давайте добавим его тоже с таким же путём, как путь к нашему первому репозиторию:

```
git remote add all path-1
```

Дальше вы можете добавить к этому репозиторию пути:

```
git remote set-url --add --push all path-1  
git remote set-url --add --push all path-2
```

И теперь вы можете отправлять наши изменения сразу во все репозитории, используя имя `all`:

```
git push all master
```

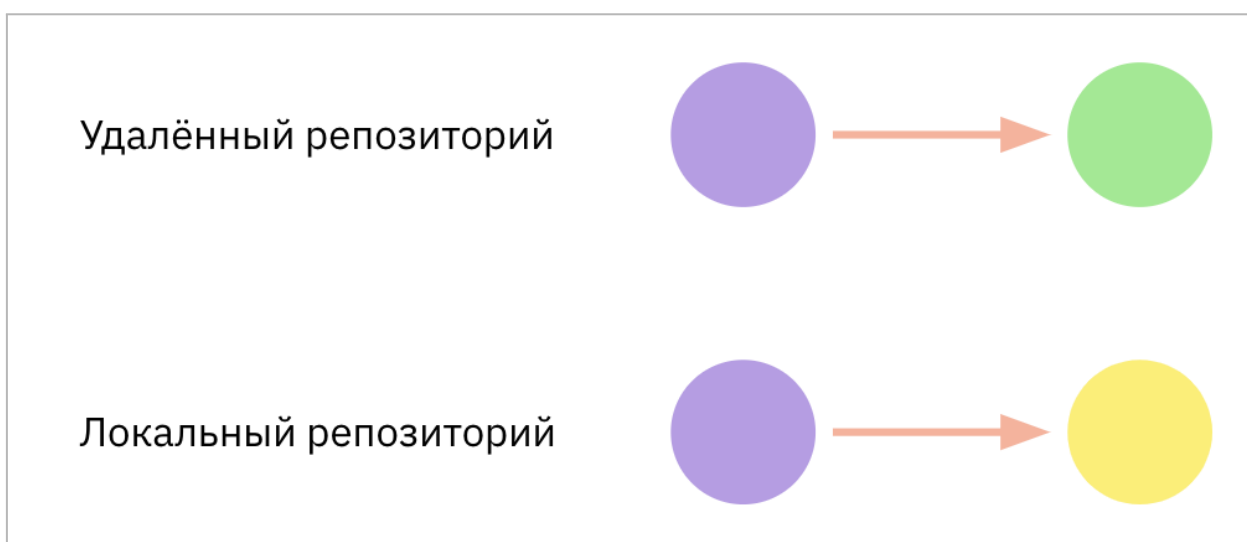


Вот так можно работать с несколькими репозиториями. Давайте теперь повторим команды, которые мы с вами использовали в работе.

Разрешение типичных конфликтов

В этом блоке мы поговорим о разрешении типичных конфликтов, возникающих при синхронизации локального репозитория с удалённым.

Напомним, что такое конфликт. Это ситуация, при которой на сервере в удалённом репозитории и у вас в локальном репозитории созданы разные коммиты:



Такие конфликты бывают двух видов: простые, которые могут быть разрешены автоматически, самим Git-ом, и сложные, которые можно разрешить только вручную.

Чтобы создать простой конфликт, вам нужно внести изменения в один файл в локальной ветке и в другой файл в удалённой. В этом случае команда “git status” покажет вам, что ваш локальный репозиторий на один коммит впереди последней версии удалённого репозитория, которую вы к себе загружали:



```
1 On branch master
2 Your branch is up to date with 'origin/master'.
3
4 nothing to commit, working tree clean
```

И, казалось бы, надо просто выполнить “git push”, чтобы отправить этот коммит в удалённый репозиторий и чтобы оба репозитория синхронизировались, но после выполнения этой команды вы видите, что ничего не получилось и появилось “страшное” сообщение об ошибке:

```
To github.com:sortedmap/git-advanced.git
! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'git@github.com:sortedmap/git-advanced.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

На самом деле, в этом сообщении нет ничего страшного. В нём лишь написано, что удалённый репозиторий содержит изменения, которых у вас ещё нет локально. И также в этом сообщении Git вам предлагает выполнить вначале команду “git pull”, чтобы загрузить эти изменения:

```
git pull
```

После выполнения этой команды может появиться текстовый редактор, в котором будет предложено ввести сообщение коммита. Поскольку у нас был конфликт версий, то есть две конфликтующие версии кода, при затягивании изменений Git попытается их слить с вашей рабочей копией.

Мы говорили с вами, что поменяем разные файлы, поэтому Git сможет слить изменения легко: изменения одного файла будут взяты из локального репозитория, изменения второго — из удалённого. На самом деле, в момент Git их уже слил, и результат слияния у нас лежит в текущей рабочей копии, и он не закоммичен. И чтобы его сразу закоммитить, Git предложит вам отредактировать сообщение коммита по умолчанию.



Редактор, в котором, как правило, предлагается это сделать, называется vim. Вы можете изменить в нём сообщение коммита, затем нажать “Esc”, чтобы выйти из режима редактирования, и далее выйти из самого редактора и сохранить сообщение коммита командой:

```
:wq
```

Эта команда сохраняет изменённое сообщение коммита. Вы можете также и просто выйти из этого редактора, набрав команду “:q” без буквы “w”, и тогда сообщение коммита будет установлено по умолчанию.

После этого слияние файлов удалённого репозитория с файлами локального завершится, и у вас появится ещё один коммит, но он появится только локально. Чтобы другие разработчики его увидели и увидели ваши изменения в коде, которые вы делали до этого, вам нужно отправить его в удалённый репозиторий:

```
git push
```

Теперь давайте рассмотрим сложный конфликт, который Git не сможет разрешить автоматически. Создать его можно, внося разные изменения в одну и ту же строку одного и того же файла. В этом случае вам также следует сначала выполнить команду:

```
git pull
```

Появится сообщение об ошибке: “Automatic merge failed; fix conflicts and then commit the result”. Git, опять же, сольёт файл из удалённого репозитория с файлом, который был в вашем локальном репозитории, но пометит в этом файле обе версии кода, которые он не смог слить, и не сделал коммит.

Локальная версия кода будет размещена между строками “<<<<<< HEAD” и “=====”, а удалённая версия — между строками “=====” и “>>>>>> hash”, где hash — это хэш коммита из удалённого репозитория. Вот пример того, как это может выглядеть в файле:



```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Система управления проектами</title>
5     <script src="script.js"></script>
6   </head>
7   <body>
8     <h1>Система управления проектами</h1>
9     <p>Текст описания системы, подробный и детальный, текст
10    <textarea id="text"></textarea><br>
11    <button class="button">Рассчитать длину текста</button>
12    <<<<<<< HEAD
13    <button class="button">Посчитать длину и высоту</button>
14    =====
15    <button class="button">Посчитать длину и ширину</button>
16    >>>>>>> eb685334d70dca138e7bc13a1e6e2cd271520dc5
17    <hr>
18  </body>
19 </html>
```

Такой конфликт слияния вам нужно разрешить вручную: отредактировать этот файл и выбрать ту часть, которая будет более правильной, или объединить их как-то по-своему, после чего сохранить и закоммитить сделанные изменения.

Вообще алгоритм решения любых проблем с Git очень прост:

1. Первым шагом вам нужно ознакомиться с содержанием сообщения о возникшей ошибке и с рекомендациями, которые даёт сам Git.
2. На втором шаге вам нужно осознать, что, собственно, случилось, вникнуть и действительно понять проблему.
3. После этого либо решить проблему самостоятельно, либо последовать рекомендациям самого Git-а.

Важно, что решать проблему имеет смысл только тогда, когда вы её поняли. Крайне не рекомендуется сразу при возникновении сообщения об ошибке гуглить решения и выполнять их в командной строке. Это часто приводит ко всё большему непониманию происходящего и ухудшению ситуации, из



которой выбраться будет сложнее. Поэтому вначале вам следует прочитать, что пишет Git, попытаться понять, что действительно случилось — что с чем конфликтует и почему, а потом уже решать проблему.

Что можно почитать еще?

1. [Синхронизация в Git](#)
2. [Разрешение конфликтов в Git](#)
3. [Официальная документация по git remote](#)