Введение в системы контроля версий

Часто разработчики трудятся в команде над одним проектом, а значит, сразу несколько человек могут изменять один файл одновременно. Чтобы избежать путаницы, в таких случаях используют систему контроля версий, которая позволяет хранить историю изменений проекта и при необходимости помогает вернуться к предыдущей версии.

Версионирование

Чтобы лучше понять проблему версионирования, рассмотрим пример дизайнера, который закончил работать над проектом и отправил финальную версию заказчику. У дизайнера есть папка, в которой хранится финальная версия проекта:

```
source/
barbershop_index_final.psd
```

Всё хорошо, дизайнер закончил работу, но заказчик прислал в ответ правки. Чтобы была возможность вернуться к старой версии проекта, дизайнер создал новый файл barbershop_index_final_2.psd, внёс изменения и отправил заказчику:

```
source/
barbershop_index_final.psd
barbershop_index_final_2.psd
```

Этим всё не ограничилось, в итоге структура проекта разрослась и стала выглядеть так:

```
source/
barbershop_index_final.psd
barbershop_index_final_2.psd
…
barbershop_index_final_19.psd
…
barbershop_index_latest_final.psd
barbershop_index_latest_final_Final.psd
```

Вероятно, многие уже сталкивались с подобным, например, при написании курсовых работ во время учёбы. В профессиональной разработке создавать новые файлы для версионирования — плохая практика. Обычно у разработчиков в папке проекта хранится множество файлов. Также над одним проектом может работать несколько человек. Если каждый разработчик для версионирования будет создавать новый файл, немного изменяя название предыдущей версии, то в скором времени в проекте начнётся хаос и никто не будет понимать, какие файлы нужно открывать.

Типы систем контроля версий

Теперь вы знаете, что такое система контроля версий. Однако они тоже бывают разными. Существует три типа СКВ: локальная, централизованная и распределённая.

Существует три типа СКВ: локальная, централизованная и распределенная.

Локальные системы контроля версий (ЛСКВ)

Принцип работы локальной системы контроля версий
В качестве метода контроля версий можно копировать файлы в отдельную директорию. Изменения сохраняются в виде наборов патчей, где каждый патч датируется и получает отметку времени. Таким образом, если код перестаёт работать, наборы патчей можно совместить, чтобы получить исходное состояние файла. Такой подход всё ещё распространён среди разработчиков.

Централизованные системы контроля версий (ЦСКВ)

Принцип работы централизованной системы контроля версий

ЦСКВ были созданы для решения проблемы взаимодействия с другими разработчиками. Такие системы имеют единственный сервер, содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из этого централизованного хранилища и там же их сохраняют. Тем не менее, такой подход имеет существенный недостаток — выход сервера из строя обернётся потерей всех данных. Кроме того, в таких системах может быть затруднена одновременная работа нескольких разработчиков над одним файлом.

Распределённые системы контроля версий (РСКВ)

Принцип работы распределённой системы контроля версий

Недостаток ЦСКВ был исправлен в РСКВ, клиенты которых не просто скачивают снимок всех файлов (состояние файлов на определённый момент времени), а полностью копируют репозиторий. Это значит, что у каждого клиента есть копия всего исходного кода и внесённых изменений. В этом случае, если один из серверов выйдет из строя, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Ещё одним преимуществом РСКВ является то, что они могут одновременно взаимодействовать с несколькими удалёнными репозиториями. Благодаря этому разработчики могут параллельно работать над несколькими проектами. Именно поэтому Git сейчас так популярен.

Основные понятия

Список терминов, которые будут вам полезны.

Репозиторий

Проект, в котором была инициализирована система Git, называется репозиторием. При инициализации в проект добавляется скрытая папка .git. Репозиторий хранит все рабочие файлы и историю их изменений.

Рабочая область и хранилище

```
barbershop/
    | - .git
    |  | - bea0f8e
    |  | - d516600
    |  |   Хранилище
    |
    | - css
    | - index.html
    |   Рабочая область
```

Корневая папка проекта — это рабочая область. В ней находятся все файлы и папки, необходимые для его работы.

Хранилище — это содержимое скрытой папки .git. В этой папке хранятся все версии рабочей области и служебная информация. Этим версиям система автоматически даёт название, состоящее из букв и цифр. В примере выше — это bea0f8e и d516600. Не стоит проводить манипуляции с папкой .git вручную. Вся работа с системой производится командами через специальные приложения или консоль.

Коммит

Точно так же, как и в игре, в системе контроля версий Git можно сохранить текущее состояние проекта. Для этого есть специальная команда — commit. Она делает так, что новая версия проекта сохраняется и добавляется в хранилище. В файле с сохранением отображаются: все изменения, которые происходили в рабочей области, автор изменений и краткий комментарий, описывающий суть изменений. Каждый коммит хранит полное состояние рабочей области, её папок и файлов проекта.

В итоге проект работает так:

1. Репозиторий хранит все версии проекта. В случае передачи этого проекта другому человеку, он увидит всё, что с ним происходило до этого.
2. Ничего не теряется и не удаляется бесследно. При удалении файла в новой версии добавляется запись о том, что файл был удалён.
3. Всегда можно вернуться к любой из версий проекта, загрузив её из хранилища в рабочую область.

Система контроля версий Git

Git — это распределённая и децентрализованная система управления версиями файлов. Децентрализованная система означает, что у каждого разработчика есть личный репозиторий проекта с полным набором всех версий. А все необходимые для работы файлы находятся на компьютере. При этом постоянное подключение к сети не требуется, поэтому система работает быстро. При командной разработке нужна синхронизация репозиториев, так как проект — один и его состояние должно быть у всех одинаковым.

Подход Git к хранению данных похож на набор снимков миниатюрной файловой системы. Каждый

раз, когда вы сохраняете состояние своего проекта в Git, система запоминает, как выглядит каждый файл в этот момент, и сохраняет ссылку на этот снимок.

Преимущества Git:

- **Бесплатный и open-source**. Можно бесплатно скачать и вносить любые изменения в исходный код;
- **Небольшой и быстрый**. Выполняет все операции локально, что увеличивает его скорость. Кроме того, Git локально сохраняет весь репозиторий в небольшой файл без потери качества данных;
- **Резервное копирование**. Git эффективен в хранении бэкапов, поэтому известно мало случаев, когда кто-то терял данные при использовании Git;
- **Простое ветвление**. В других системах контроля версий создание веток— утомительная и трудоёмкая задача, так как весь код копируется в новую ветку. В Git управление ветками реализовано гораздо проще и эффективнее.

Введение в Git

1. Настройка git

Прежде чем начинать работу с git необходимо его настроить под себя!

1.1 Конфигурационные файлы

- /etc/gitconfig — Общие настройки для всех пользователей и репозиториев
- ~/.gitconfig или ~/.config/git/config — Настройки конкретного пользователя
- .git/config — Настройки для конкретного репозитория

Есть специальная команда

```
git config [<опции>]
```

которая позволит вам изменить стандартное поведение git, если это необходимо, но вы можете редактировать конфигурационные файлы в ручную (я считаю так быстрее).В зависимости какой параметр вы передадите команде git config (--system, --global, --local), настройки будут записываются в один из этих файлов. Каждый из этих “уровней” (системный, глобальный, локальный) переопределяет значения предыдущего уровня!Что бы посмотреть в каком файле, какие настройки установлены используйте git config --list --show-origin.

Игнорирование файлов В git вы сами решаете какие файлы и в какой коммит попадут, но возможно вы бы хотели, что бы определённые файлы никогда не попали в индекс и в коммит, да и вообще не отображались в списке не отлеживаемых. Для этого вы можете создать специальный файл (.gitignore) в вашем репозитории и записать туда шаблон игнорируемых файлов. Если вы не хотите создавать такой файл в каждом репозитории вы можете определить его глобально с помощью core.excludesfile (см. Полезные настройки). Вы также можете скачать готовый .gitignore file для языка программирования на котором вы работаете. Для настройки .gitignore используйте регулярные выражения bash.

1.2 Настройки по умолчанию

Есть куча настроек git'a как для сервера так и для клиента, здесь будут рассмотрены только основные настройки клиента.Используйте

```
git config name value
```

где name это название параметра, а value его значение, для того что бы задать настройки. Пример:

```
git config --global core.editor nano
```

установит редактор по умолчанию nano.Вы можете посмотреть значение существующего параметра с помощью git config --get [name] где name это параметр, значение которого вы хотите получить.**Полезные настройки:**

- user.name — Имя, которое будет использоваться при создании коммита
- user.email — Email, который будет использоваться при создании коммита
- core.excludesfile — Файл, шаблон которого будет использоваться для игнорирования определённых файлов глобально
- core.editor — Редактор по умолчанию
- commit.template — Файл, содержимое которого будет использоваться для сообщения коммита по умолчанию (См. Работа с коммитами).
- help.autocorrect — При установке значения 1, git будет выполнять неправильно написанные команды.
- credential.helper [mode] — Устанавливает режим хранения учётных данных. [cache] — учётные данные сохраняются на определённый период, пароли не сохраняются (--timeout [seconds] количество секунд после которого данные удаляются, по умолчанию 15 мин). [store] — учётные данные сохраняются на неограниченное время в открытом виде (--file [file] указывает путь для хранения данных, по умолчанию ~/.git-credentials).

1.3 Псевдонимы (aliases)

Если вы не хотите печатать каждую команду для Git целиком, вы легко можете настроить псевдонимы. Для создания псевдонима используйте:

```
git config alias.SHORT_NAME COMMAND
```

где SHORT_NAME это имя для сокращения, а COMMAND команда(ы) которую нужно сократить. Пример:

```
git config --global alias.last 'log -1 HEAD'
```

после выполнения этой команды вы можете просматривать информацию о последнем коммите на текущей ветке выполнив git last.Я советую вам использовать следующие сокращения (вы также можете определить любые свои):

- st = status
- ch = checkout
- br = branch
- mg = merge
- cm = commit
- reb = rebase
- lg = «git log --pretty=format:'%h — %ar: %s'»

Для просмотра настроек конфигурации используйте: git config --list

Для просмотра настроек конфигурации используйте: git config --list.

2. Основы git

> Здесь перечислены только обязательные и полезные (на мой взгляд) параметры, ибо перечисление всех неуместно. Для этого используйте git command -help или --help, где command — название команды справку о который вы хотите получить.

2.1 Создание репозитория

- git init [<опции>] — Создаёт git репозитории и директорию .git в текущей директории (или в директории указанной после --separate-git-dir <каталог-git>, в этом случае директория .git будет находится в другом месте);
- git clone [<опции>] [--] <репозиторий> [<каталог>] [-o, --origin <имя>] [-b, --branch <ветка>] [--single-branch] [--no-tags] [--separate-git-dir <каталог-git>] [-c, --config <ключ=значение>] — Клонирует репозитории с названием origin (или с тем которое вы укажите -o <имя>), находясь на той ветке, на которую указывает HEAD (или на той которую вы укажите -b <ветка>). Также вы можете клонировать только необходимую ветку HEAD (или ту которую укажите в -b <ветка>) указав --single-branch. По умолчанию клонируются все метки, но указав --no-tags вы можете не клонировать их. После выполнения команды создаётся директория .git в текущей директории (или в директории указанной после --separate-git-dir <каталог-git>, в этом случае директория .git будет находится в другом месте);

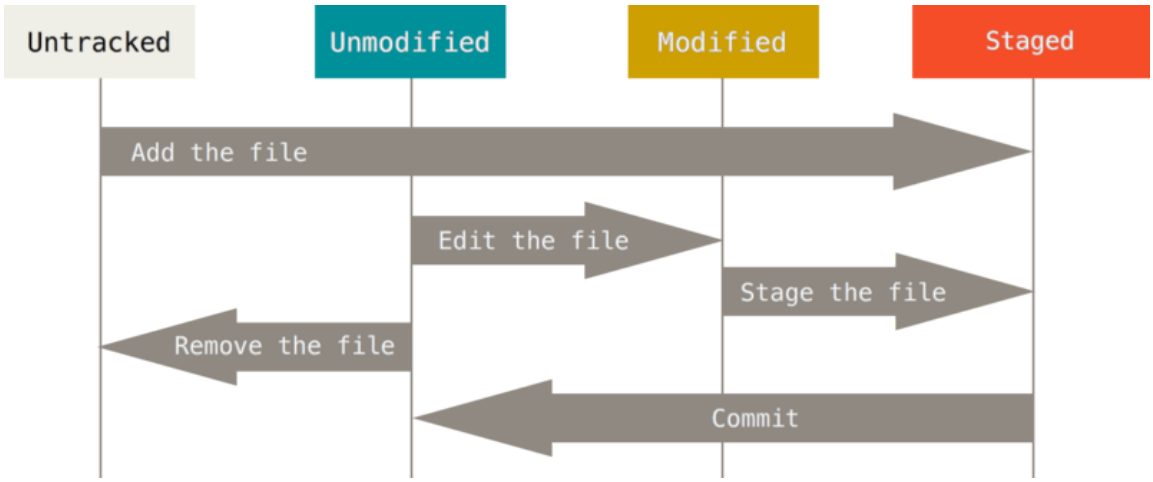
2.2 Состояние файлов

Для просмотра состояния файлов в вашем репозитории используйте:

```
git status [<опции>]
```

Эта команда может показать вам: на какой ветке вы сейчас находитесь и состояние всех файлов. Обязательных опций нет, из полезных можно выделить разве что -s которая покажет краткое представление о состояний файлов.

Жизненный цикл файлов



Как видно на картинке файлы могут быть не отслеживаемые (Untracked) и отслеживаемые. Отслеживаемые файлы могут находится в 3 состояниях: Не изменено (Unmodified), изменено (Modified), подготовленное (Staged). Если вы добавляете (с помощью git add) «Не отслеживаемый» файл, то он переходит в состояние «Подготовлено». Если вы изменяете файл в состояния «Не

изменено», то он переходит в состояние «Изменено». Если вы сохраняете изменённый файл (то есть находящийся в состоянии «Изменено») он переходит в состояние «Подготовлено». Если вы делаете коммит файла (то есть находящийся в состоянии «Подготовлено») он переходит в состояние «Не изменено». Если версии файла в HEAD и рабочей директории отличаются, то файл будет находится в состояний «Изменено», иначе (если версия в HEAD и в рабочем каталоге одинакова") файл будет находится в состояний «Не изменено». Если версия файла в HEAD отличается от рабочего каталога, но не отличается от версии в индексе, то файл будет в состоянии «Подготовлено».Этот цикл можно представить следующим образом: Unmodified -> Modified -> Staged -> Unmodified To есть вы изменяете файл сохраняете его в индексе и делаете коммит и потом все сначала.

2.3 Работа с индексом

Надеюсь вы поняли, как выглядит жизненный цикл git репозитория. Теперь разберём как вы можете управлять индексом и файлами в вашем git репозитории.Индекс — промежуточное место между вашим прошлым коммитом и следующим. Вы можете добавлять или удалять файлы из индекса. Когда вы делаете коммит в него попадают данные из индекса, а не из рабочей области.Что бы просмотреть индекс, используйте git status. Что бы добавить файлы в индекс используйте

```
git add [<опции>]
```

Полезные параметры команды git add:

- -f, --force — добавить также игнорируемые файлы
- -u, --update — обновить отслеживаемые файлы

Что бы удалить файлы из индекса вы можете использовать 2 команды git reset и git restore. git-restore — восстановит файлы рабочего дерева. git-reset — сбрасывает текущий HEAD до указанного состояния. По сути вы можете добиться одного и того же с помощью обеих команд.Что бы удалить из индекса некоторые файлы используйте:

```
git restore --staged <file>
```

таким образом вы восстановите ваш индекс (или точнее удалите конкретные файлы из индекса), будто бы git add после последнего коммита не выполнялся для них. С помощью этой команды вы можете восстановить и рабочую директорию, что бы она выглядела так, будто бы после коммита не выполнялось никаких изменений. Вот только эта команда имеет немного странное поведение — если вы добавили в индекс новую версию вашего файла вы не можете изменить вашу рабочую директорию, пока индекс отличается от HEAD. Поэтому вам сначала нужно восстановить ваш индекс и только потом рабочую директорию. К сожалению сделать это одной командой не возможно так как при передаче обеих аргументов (git restore -SW) не происходит ничего. И точно также при передаче -W тоже ничего не произойдет если файл в индексе и HEAD разный. Наверное, это сделали для защиты что бы вы случайно не изменили вашу рабочую директорию. Но в таком случае почему аргумент -W передаётся по умолчанию? В общем мне не понятно зачем было так сделано и для чего вообще была добавлена эта команда. По мне так reset справляется с этой задачей намного лучше, да и еще и имеет более богатый функционал так как может перемещать индекс и рабочую директорию не только на последний коммит но и на любой другой.Но собственно разработчики рекомендуют для сброса индекса использовать именно git restore -S . Вместо git reset HEAD .С помощью git status вы можете посмотреть какие файлы изменились но если вы также хотите узнать что именно изменилось в файлах то воспользуйтесь командой:

```
git diff [<options>]
```

таким образом выполнив команду без аргументов вы можете сравнить ваш индекс с рабочей директорией. Если вы уже добавил в индекс файлы, то используйте git diff --cached что бы посмотреть различия между последним коммитом (или тем который вы укажите) и рабочей директории. Вы также можете посмотреть различия между двумя коммитами или ветками передав их как аргумент. Пример: git diff 00656c 3d5119 покажет различия между коммитом 00656c и 3d5119.

2.4 Работа с коммитами

Теперь, когда ваш индекс находится в нужном состояний, пора сделать коммит ваших изменений. Запомните, что все файлы для которых вы не выполнили git add после момента редактирования — не войдут в этот коммит. На деле файлы в нём будут, но только их старая версия (если таковая имеется).Для того что бы сделать коммит ваших изменений используйте:

```
git commit [<опции>]
```

Полезные опции команды git commit:

- -F, --file [file] — Записать сообщение коммита из указанного файла
- --author [author] — Подменить автора коммита
- --date [date] — Подменить дату коммита
- -m, --mesage [message] — Сообщение коммита
- -a, --all — Закоммитеть все изменения в файлах
- -i, --include [files...] — Добавить в индекс указанные файлы для следующего коммита
- -o, --only [files...] — Закоммитеть только указанные файлы
- --amend — Перезаписать предыдущий коммит

Вы можете определить сообщение для коммита по умолчанию с помощью commit.template. Эта директива в конфигурационном файле отвечает за файл содержимое которого будет использоваться для коммита по умолчанию. Пример: git config --global commit.template ~/.gitmessage.txt.Вы также можете изменить, удалить, объединить любой коммит. Как вы уже могли заметить вы можете быстро перезаписать последний коммит с помощью git commit --amend. Для изменения коммитом в вашей истории используйте

```
git rebase -i <commit>
```

где commit это верхний коммит в вашей цепочке с которого вы бы хотели что либо изменить.После выполнения git rebase -i в интерактивном меню выберите что вы хотите сделать.

- pick <коммит> = использовать коммит
- reword <коммит> = использовать коммит, но изменить сообщение коммита
- edit <коммит> = использовать коммит, но остановиться для исправления
- squash <коммит> = использовать коммит, но объединить с предыдущим коммитом
- fixup <коммит> = как «squash», но пропустить сообщение коммита
- exec <команда> = выполнить команду (остаток строки) с помощью командной оболочки
- break = остановиться здесь (продолжить с помощью «git rebase --continue»)
- drop <коммит> = удалить коммит
- label <метка> = дать имя текущему HEAD
- reset <метка> = сбросить HEAD к указанной метке

Для изменения сообщения определённого коммита. Необходимо изменить pick на edit над коммитом который вы хотите изменить. Пример: вы хотите изменить сообщение коммита 750f5ae.pick 2748cb4 first commit edit 750f5ae second commit pick 716eb99 third commit После

сохранения скрипта вы вернётесь в командную строку и git скажет что необходимо делать дальше:Остановлено на 750f5ae … second commit You can amend the commit now, withgit commit --amend Once you are satisfied with your changes, rungit rebase --continueКак указанно выше необходимо выполнить git commit --amend для того что бы изменить сообщение коммита. После чего выполнить git rebase --continue. Если вы выбрали несколько коммитов для изменения названия то данные операций необходимо будет проделать над каждым коммитом.**Для удаления коммита** Необходимо удалить строку с коммитом. Пример: вы хотите удалить коммит 750f5ae Нужно изменить скрипт с такого: pick 2748cb4 third commit pick 750f5ae second commit pick 716eb99 first commit на такой: pick 2748cb4 first commit pick 716eb99 third commit**Для объединения коммитов** Необходимо изменить pick на squash над коммитами которые вы хотите объединить. Пример: вы хотите объединить коммиты 750f5ae и 716eb99. Необходимо изменить скрипт с такого: pick 2748cb4 third commit pick 750f5ae second commit pick 716eb99 first commit На такой pick 2748cb4 third commit squash 750f5ae second commit squash 716eb99 first commit3аметьте что в интерактивном скрипте коммиты изображены в обратном порядке нежели в git log. С помощью squash вы объедините коммит 750f5ae с 716eb99, а 750f5ae с 2748cb4. В итоге получая один коммит содержащий изменения всех трёх.

2.5 Просмотр истории

С помощью команды

```
git log [<опции>] [<диапазон-редакций>]
```

вы можете просматривать историю коммитов вашего репозитория. Есть также куча параметров для сортировки и поиска определённого коммита.Полезные параметры команды git log:

- -p — Показывает разницу для каждого коммита.
- --stat — Показывает статистику измененных файлов для каждого коммита.
- --graph — Отображает ASCII граф с ветвлениями и историей слияний.

Тагже можно отсортировать коммиты по времени, количеству и тд.

- -(n) Показывает только последние n коммитов.
- --since, --after — Показывает коммиты, сделанные после указанной даты.
- --until, --before — Показывает коммиты, сделанные до указанной даты.
- --author — Показывает только те коммиты, в которых запись author совпадает с указанной строкой.
- --committer — Показывает только те коммиты, в которых запись committer совпадает с указанной строкой.
- --grep — Показывает только коммиты, сообщение которых содержит указанную строку.
- -S — Показывает только коммиты, в которых изменение в коде повлекло за собой добавление или удаление указанной строки.

Вот несколько примеров: git log --since=3.weeks — Покажет коммиты за последние 2 недели git log --since=«2019-01-14» — Покажет коммиты сделанные 2019-01-14 git log --since=«2 years 1 day ago» — Покажет коммиты сделанные 2 года и один день назад.Также вы можете настроить свои формат вывода коммитов с помощью

```
git log --format:["format"]
```

Варианты форматирования для git log --format.

- %H — Хеш коммита
- %h — Сокращенный хеш коммита

- %T — Хеш дерева
- %t — Сокращенный хеш дерева
- %P — Хеш родителей
- %p — Сокращенный хеш родителей
- %an — Имя автора — %ae — Электронная почта автора
- %ad — Дата автора (формат даты можно задать опцией --date=option)
- %ar — Относительная дата автора
- %cn — Имя коммитера
- %ce — Электронная почта коммитера
- %cd — Дата коммитера
- %cr — Относительная дата коммитера
- %s — Содержание

Пример:

```
git log --pretty=format:"%h - %ar : %s"
```

покажет список коммитов состоящий из хэша времени и сообщения коммита.

2.6 Работа с удалённым репозиторием

Так как git это распределённая СКВ вы можете работать не только с локальными но и с внешними репозиториеми.Удалённые репозитории представляют собой версии вашего проекта, сохранённые на внешнем сервере.Для работы с внешними репозиториями используйте:

```
git remote [<options>]
```

Если вы с клонировали репозитории через http URL то у вас уже имеется ссылка на внешний. В другом случае вы можете добавить её с помощью

```
git remote add [<options>] <name> <adres>
```

Вы можете тут же извлечь внешние ветки с помощью -f, --fetch (вы получите имена и состояние веток внешнего репозитория). Вы можете настроить репозитории только на отправку или получение данных с помощью --mirror[=(push|fetch)]. Для получения меток укажите --tags. Для просмотра подключённых внешних репозиториев используйте git remote без аргументов или git remote -v для просмотра адресов на отправку и получение данных от репозитория.Для отслеживания веток используйте git branch -u <rep/br> где rep это название репозитория, br название внешней ветки, а branch название локальной ветки. Либо git branch --set-upstream local_br origin/br для того что бы указать какая именно локальная ветка будет отслеживать внешнюю ветку.Когда ваша ветка отслеживает внешнюю вы можете узнать какая ветка (локальная или внешняя) отстаёт или опережает и на сколько коммитов. К примеру если после коммита вы не выполняли git push то ваша ветка будет опережать внешнюю на 1 коммит. Вы можете узнать об этом выполнив git branch -vv, но прежде выполните git fetch [remote-name] (--all для получения обновления со всех репозиториев) что бы получить актуальные данные из внешнего репозитория. Для отмены отслеживания ветки используйте git branch --unset-upstream [<local_branch>].Для загрузки данных с внешнего репозитория используйте git pull [rep] [branch]. Если ваши ветки отслеживают внешние, то можете не указывать их при выполнение git pull. По умолчанию вы получите данные со всех отслеживаемых веток.Для загрузки веток на новую ветку используйте git checkout -b <new_branch_name> <rep/branch>. Для отправки данных на сервер используйте

```
git push [<rep>] [
]
```

где rep это название внешнего репозитория, a br локальная ветка которую вы хотите отправить. Также вы можете использовать такую запись git push origin master:dev. Таким образом вы выгрузите вашу локальную ветку master на origin (но там она будет называется dev). Вы не сможете отправить данные во внешний репозитории если у вас нет на это прав. Также вы не сможете отправить данные на внешнюю ветку если она опережает вашу (в общем то отправить вы можете используя -f, --forse в этом случае вы перепишите историю на внешнем репозитории). Вы можете не указывать название ветки если ваша ветка отслеживает внешнюю.Для удаления внешних веток используйте

```
git push origin --delete branch_name
```

Для получения подробной информации о внешнем репозитории (адреса для отправки и получения, на что указывает HEAD, внешние ветки, локальные ветки настроенные для git pull и локальные ссылки настроенные для git push)

```
git remote show <remote_name>
```

Для переименования названия внешнего репозитория используйте

```
git remote rename <last_name> <new_name>
```

Для удаления ссылки на внешний репозитории используйте

```
git remote rm <name>
```

3. Ветвление в git

Ветвление это мощные инструмент и одна из главных фич git'a поскольку позволяет вам быстро создавать и переключатся между различным ветками вашего репозитория. Главная концепция ветвления состоит в том что вы можете откланяться от основной линии разработки и продолжать работу независимо от нее, не вмешиваясь в основную линию. Ветка всегда указывает на последний коммит в ней, а HEAD указывает на текущую ветку (см. Указатели в git).

3.1 Базовые операций

Для создания ветки используйте

```
git branch <branch_name> [<start_commit>]
```

Здесь branch_name это название для новой ветки, a start_commit это коммит на который будет указывать ветка (то есть последний коммит в ней). По умолчанию ветка будет находится на последнем коммите родительской ветки. Опции git branch:

- -r | -a [--merged | --no-merged] — Список отслеживаемых внешних веток -r. Список и отслеживаемых и локальных веток -a. Список слитых веток --merged. Список не слитых веток --no-merged.
- -l, -f <имя-ветки> [<точка-начала>] — Список имён веток -l. Принудительное создание, перемещение или удаление ветки -f. Создание новой ветки <имя ветки>.
- -r (-d | -D) — Выполнить действие на отслеживаемой внешней ветке -r. Удалить слитую ветку -d. Принудительное удаление (даже не слитой ветки) -D.
- -m | -M [<Старая ветка>] <Новая ветка> — Переместить/переименовать ветки и ее журнал ссылок (-m). Переместить/переименовать ветку, даже если целевое имя уже существует -M.
- (-c | -C) [<

- (-c | -C) [<старая-ветка>] <новая-ветка> — Скопировать ветку и ее журнал ссылок -c. Скопировать ветку, даже если целевое имя уже существует -C.
- -v, -vv — Список веток с последним коммитом на ветке -v. Список и состояние отслеживаемых веток с последним коммитом на них.

Больше информации смотрите в git branch -h | --help.Для переключения на ветку используйте git checkout . Также вы можете создать ветку выполнив git checkout -b <ветка>.

3.2 Слияние веток

Для слияния 2 веток git репозитория используйте git merge .Полезные параметры для git merge:

- --squash — Создать один коммит вместо выполнения слияния. Если у вас есть конфликт на ветках, то после его устранения у вас на ветке прибавится 2 коммита (коммит с сливаемой ветки + коммит слияния), но указав этот аргумент у вас прибавится только один коммит (коммит слияния).
- --ff-only — Не выполнять слияние если имеется конфликт. Пусть кто ни будь другой разрешает конфликты :D
- -X [strategy] — Использовать выбранную стратегию слияния.
- --abort — Отменить выполнение слияния.

Процесс слияния. Если вы не выполняли на родительской ветке новых коммитов то слияние сводится к быстрой перемотке «fast-forward», будто бы вы не создавали новую ветку, а все изменения происходили прям тут (на родительской ветке).Если вы выполняли коммиты на обеих ветках, но при этом не создали конфликт, то слияния пройдёт в «recursive strategy», то есть вам просто нужно будет создать коммит слияния что бы применить изменения (используйте опцию --squash что бы не создавать лишний коммит). Если вы выполняли коммиты на обоих ветках, которые внесли разные изменения в одну и ту же часть одного и того же файла, то вам придётся устранить конфликт и зафиксировать слияние коммитом.При разрешении конфликта вам необходимо выбрать какую часть изменений из двух веток вы хотите оставить. При открытии конфликтующего файла, в нём будет содержатся следующее: <<<<<<< HEAD Тут будет версия изменения последнего коммита текущей ветки ====== Тут будет версия изменений последнего коммита сливаемой ветки >>>>>>> Тут название ветки с которой сливаемРазрешив конфликт вы должны завершить слияния выполнив коммит.Во время конфликта вы можете посмотреть какие различия в каких файлах имеются. git diff --ours — Разница до слияния и после git diff --theirs — Разница сливаемой ветки до слияния и после git diff --base — Разница с обеими ветками до слияния и послеЕсли вы не хотите разрешать слияние то используйте различные стратегии слияния, выбрав либо «нашу» версию (то есть ту которая находится на текущей ветке) либо выбрать «их» версию находящуюся на сливаемой ветке при этом не исправляя конфликт. Выполните git merge --Xours или git merge --Xtheirs соответственно.

3.3 Rerere

Rerere — «reuse recorded resolution” — “повторное использование сохраненных разрешений конфликтов». Механизм rerere способен запомнить каким образом вы разрешали некую часть конфликта в прошлом и провести автоматическое исправление конфликта при возникновении его в следующий раз.Что бы включить rerere выполните

```
git config --global rerere.enabled true
```

Таrже вы можите включить rerere создав каталог .git/rr-cache в нужном репозитории.Используйте git

rerere status для того что бы посмотреть для каких файлов rerere сохранил снимки состояния до начала слияния.Используйте git rerere diff для просмотра текущего состояния конфликта.Если во время слияния написано: Resolved 'nameFile' using previous resolution. Значит rerere уже устранил конфликт используя кэш.Для отмены автоматического устранения конфликта используйте git checkout --conflict=merge таким образом вы отмените авто устранение конфликта и вернёте файл(ы) в состояние конфликта для ручного устранения.

4. Указатели в git

в git есть такие указатели как HEAD branch. По сути всё очень просто HEAD указывает на текущую ветку, а ветка указывает на последний коммит в ней. Но для понимания лучше представлять что HEAD указывает на последний коммит.

4.1 Перемещение указателей

В книге Pro git приводится очень хороший пример того как вы можете управлять вашим репозиторием поэтому я тоже буду придерживается его. Представите что Git управляет содержимым трех различных деревьев. Здесь под “деревом” понимается “набор файлов”. В своих обычных операциях Git управляет тремя деревьями:

- HEAD — Снимок последнего коммита, родитель следующего
- Индекс — Снимок следующего намеченного коммита
- Рабочий Каталог — Песочница

Собственно git предоставляет инструменты для манипулировании всеми тремя деревьями. Далее будет рассмотрена команда git reset, позволяющая работать с тремя деревьями вашего репозитория.Используя различные опций этой команды вы можете:

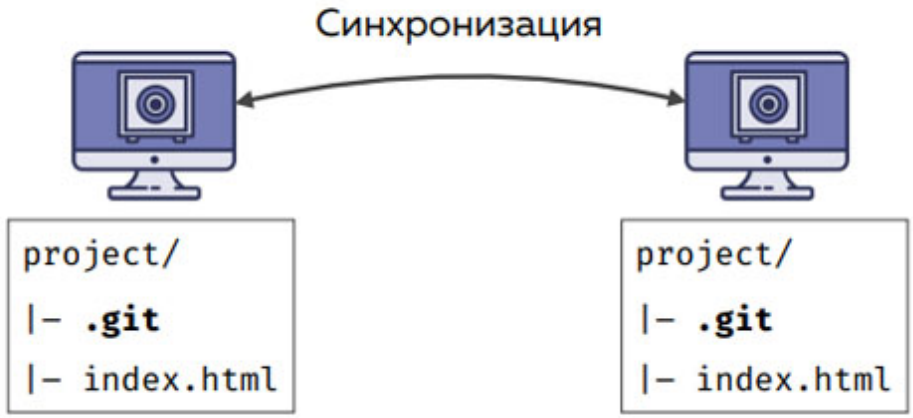
- --soft — Сбросить только HEAD
- --mixed — Сбросить HEAD и индекс
- --hard — Сбросить HEAD, индекс и рабочий каталог

Под сбросить понимается переместить на указанный коммит. По умолчанию выполняется --mixed.Примеру 1. Вы сделали 3 лишних коммита каждый из которых приносит маленькие изменения и вы хотите сделать из них один, таким образом вы можете с помощью git reset --soft переместить указатель HEAD при этом оставив индекс и рабочий каталог нетронутым и сделать коммит. В итоге в вашей истории будет выглядеть так, что все изменения произошли в одном коммите.Пример 2. Вы добавили в индекс лишние файлы и хотите их от туда убрать. Для этого вы можете использовать git reset HEAD <files...>. Или вы хотите что бы в коммите файлы выглядели как пару коммитов назад. Как я уже говорил ранее вы можете сбросить индекс на любой коммит в отличий от git restore который сбрасывает только до последнего коммита. Только с опцией mixed вы можете применить действие к указанному файлу!Пример 3. Вы начали работать над новой фичей на вашем проекте, но вдруг работодатель говорит что она более не нужна и вы в порыве злости выполняете git reset --hard возвращая ваш индекс, файлы и HEAD к тому моменту когда вы ещё не начали работать над фичей. А на следующей день вам говорят, что фичу всё таки стоит запилить. Но что же делать? Как же переместится вперёд ведь вы откатили все 3 дерева и теперь в истории с помощью git log их не найти. А выход есть — это журнал ссылок git reflog. С помощью этой команды вы можете посмотреть куда указывал HEAD и переместится не только вниз по истории коммитов но и вверх. Этот журнал является локальным для каждого пользователя.В общем думаю вы сможете придумать намного больше примеров чем я. В заключение скажу, что с помощью git reset можно творить магию

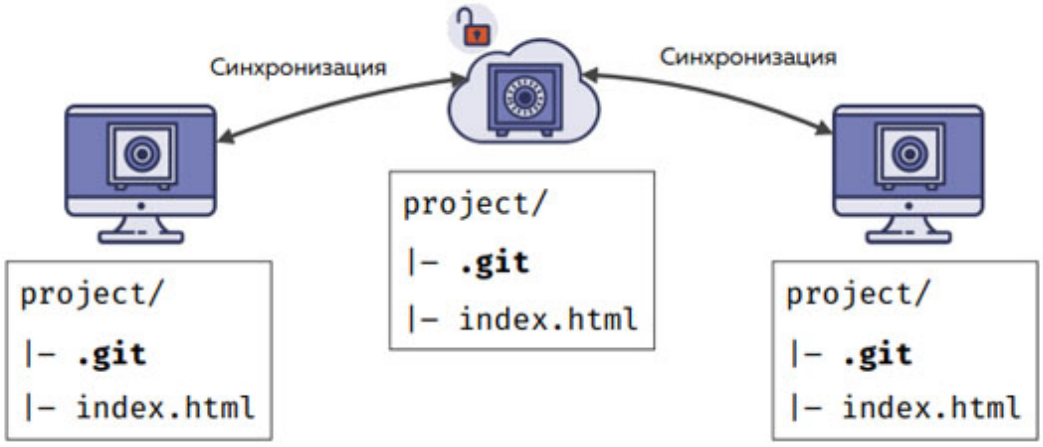
больше примеров? Чем я. В заключение скажу, что с помощью git reset можно творить магию…

5. Работа в команде

Как синхронизовать данные репозиториев между разработчиками? Изначально Git репозитории сами могут синхронизироваться от пользователя к пользователю. Дополнительные программы для этого не нужны. Есть специальные команды в консоли, позволяющие передавать данные из одного репозитория в другой.



Репозитории можно синхронизировать между пользователями.
Этот способ сложный и редко используется. Чаще всего разработчики синхронизируют локальный репозиторий с удалённым. Удалённый репозиторий — это тот же репозиторий, только его данные находятся в облаке.



Синхронизация через удалённый репозиторий.

Этапы синхронизации

Как сделать так, чтобы разработчик смог передать актуальную версию проекта коллеге?

Для взаимодействия с системой Git в консоль вводятся специальные команды. Не пугайтесь, работу с консолью можно будет заменить на работу с одной из программ, о которых расскажем ниже

с консолью можно будет заменить на работу с одной из программ, о которых расскажем ниже.
Но чтобы лучше понимать суть, придётся запомнить несколько команд. Все они начинаются с ключевого слова git. Для синхронизации есть две основных команды: pull (англ. «тянуть») и push (англ. «толкать»).

Pull

Если работа над проектом ведётся в команде, то перед тем как начать писать код, нужно получить последнюю версию проекта. Для этого нужно выполнить команду pull. Так мы забираем все изменения, которые были совершены со времени последней синхронизации с удалённым репозиторием. Теперь они у нас в репозитории на локальном компьютере.

Push

Чтобы отправить коллегам последнюю версию проекта выполняем команду push. Если в удалённом репозитории с момента последней синхронизации не было никаких изменений, то все сохранённые изменения успешно загрузятся в облако, и коллеги получат последнюю версию проекта, выполнив команду pull. Если же были изменения, то Git попросит вас перед отправкой подтянуть последние версии, сделав pull.

Синхронизация (push и pull) между локальными и удалённым репозиториями.

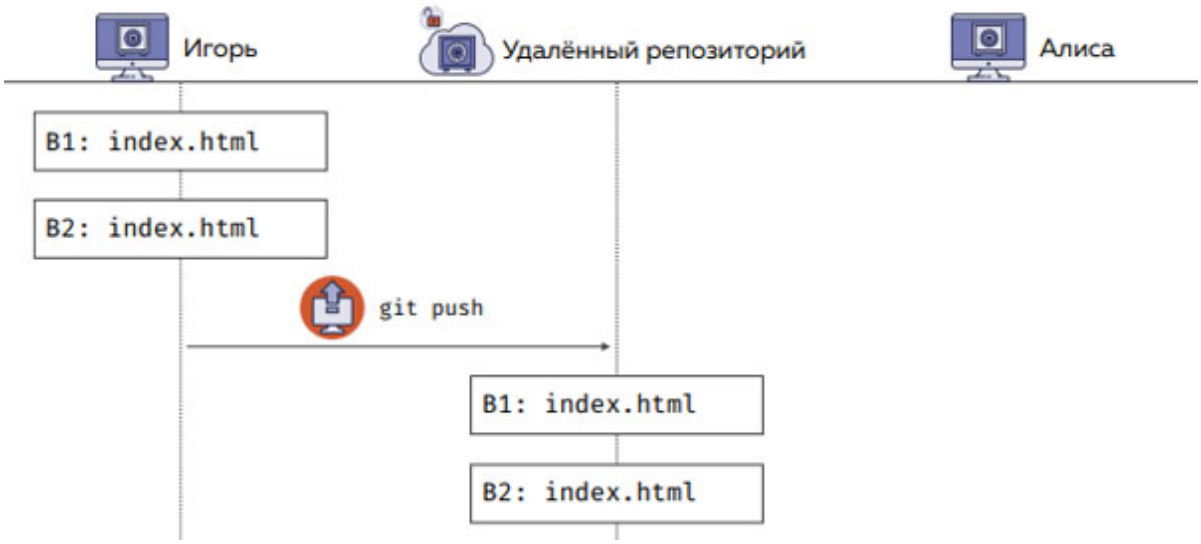
Типовой рабочий процесс с использованием Git

Разберём типовой процесс разработки сайта в команде. Представим, что Игорь и Алиса — разработчики на одном проекте. Игорь начал верстать проект и сделал первые коммиты, в которых зафиксировал изменения в файле index.html. Для схематичности названия коммитов будут простые: B1 и B2.

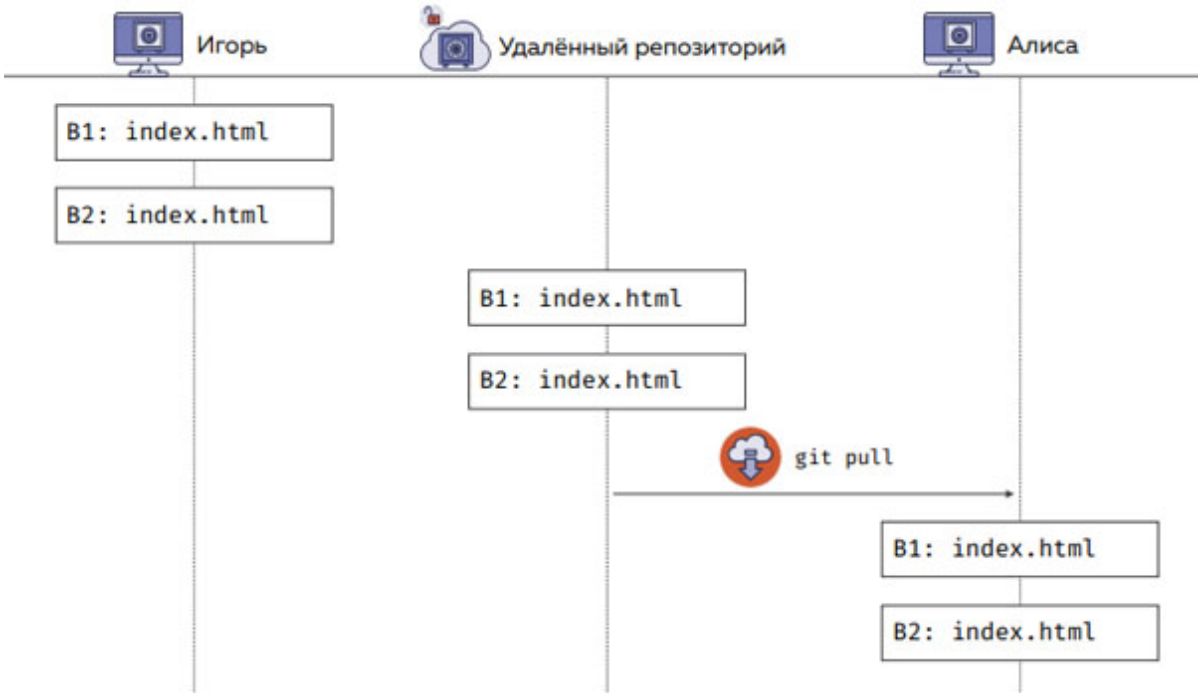
Коммиты B1 и B2.

После того как Игорь сделал два коммита, он захотел отправить свои изменения в удалённый репозиторий. Чтобы их передать, Игорь выполнил команду git push. После чего в облаке появилось две версии проекта. То есть Игорь отправил не только финальную версию проекта, но и все

сохранённые изменения.



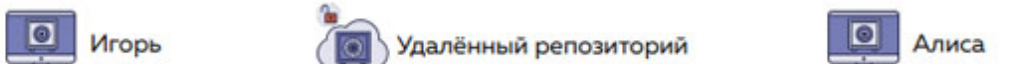
Игорь запушил свои коммиты.
После пуша данные синхронизировались с удалённым репозиторием. Но как Алисе теперь получить изменения? Для этого она выполняет команду git pull и получает все изменения из облака к себе на компьютер. Таким образом, состояние проекта у Игоря и Алисы синхронизировались, и они могут дальше продолжить работать над ним.

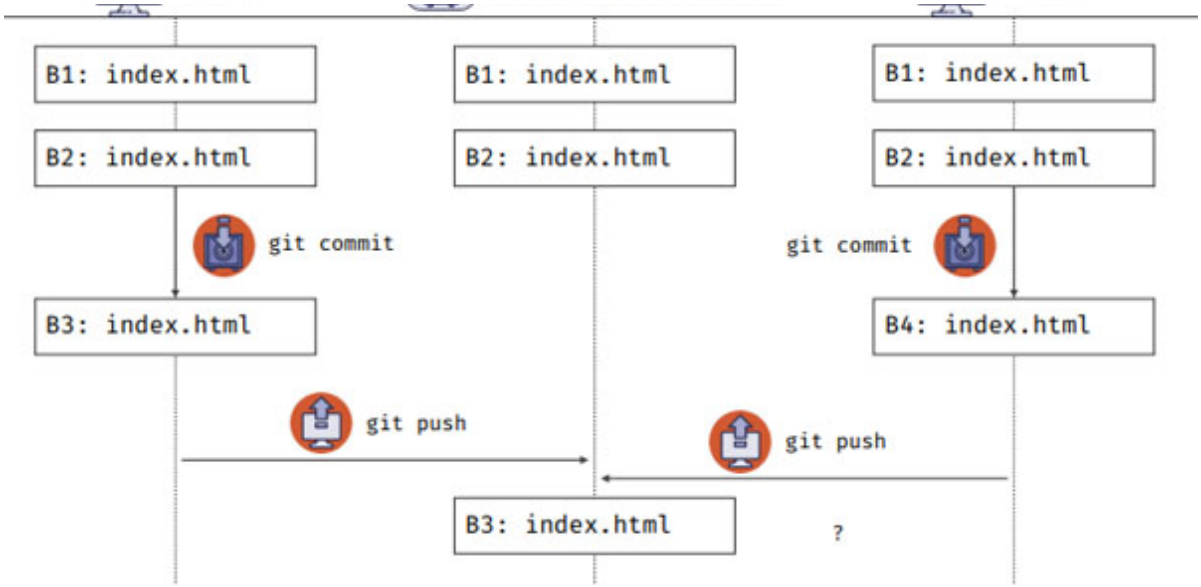


Данные у обоих разработчиков синхронизировались.

Параллельные изменения

Что произойдёт, если разработчики изменят одинаковый файл и сделают push? Предположим, что Игорь и Алиса изменили файл index.html, сделали коммит с изменениями и запушили его. Игорь оказался быстрее Алисы и сделал push первым.





Два пуша в одно время?
В этом случае Git сообщит Алисе, что нельзя пушить свои изменения, потому что она не делала pull. Дело в том, что после того как Игорь синхронизировался с удалённым репозиторием, версия проекта Алисы стала отличаться от той, что находится на удалённом репозитории, и Git это видит. Система сообщает, что перед тем, как выполнить команду push, нужно выполнить pull, чтобы забрать изменения. Алиса делает pull и ей вновь приходит уведомление от Git. В этот раз он сообщает Алисе о том, что произошёл конфликт.

Конфликт

Дело в том, что Игорь и Алиса изменили одинаковый файл и теперь Алисе предстоит решить конфликт.

Конфликт
Существуют два вида конфликтов:

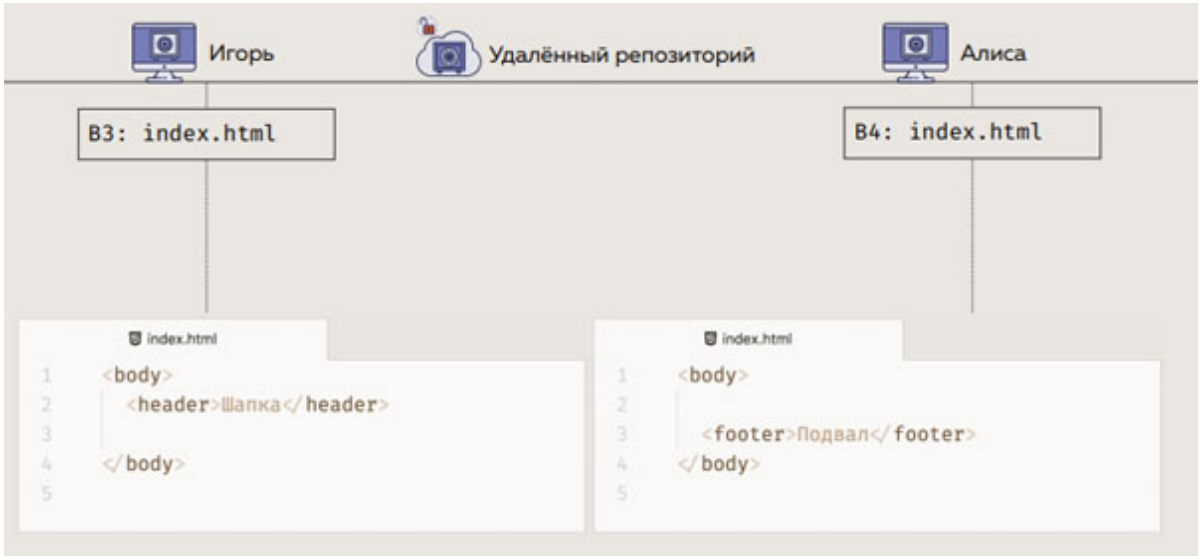
1. Автоматически разрешаемый конфликт.
2. Конфликт, который нужно разрешить вручную.

Ниже рассмотрим оба варианта.

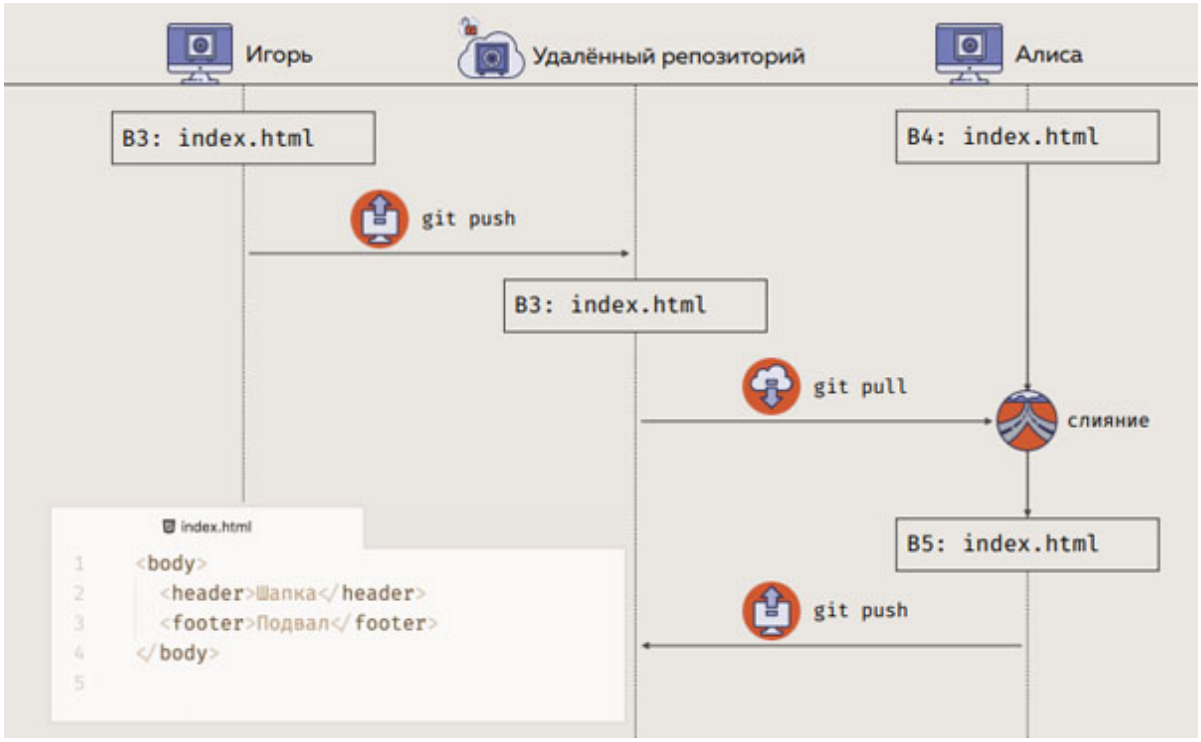
Слияние

Слияние

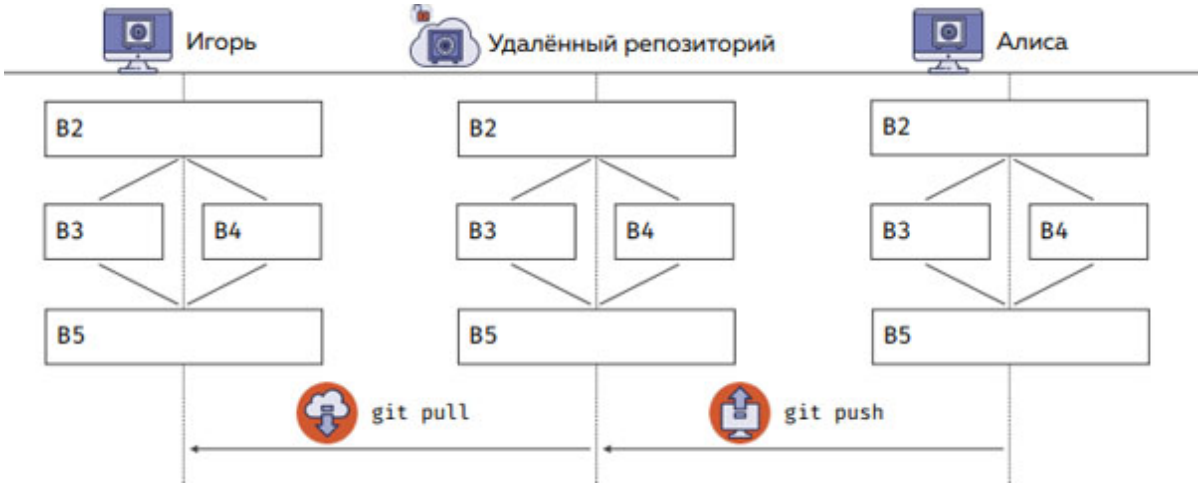
Допустим, что на третьей строке Игорь добавил в проект шапку, а на четвёртой Алиса добавила футер.



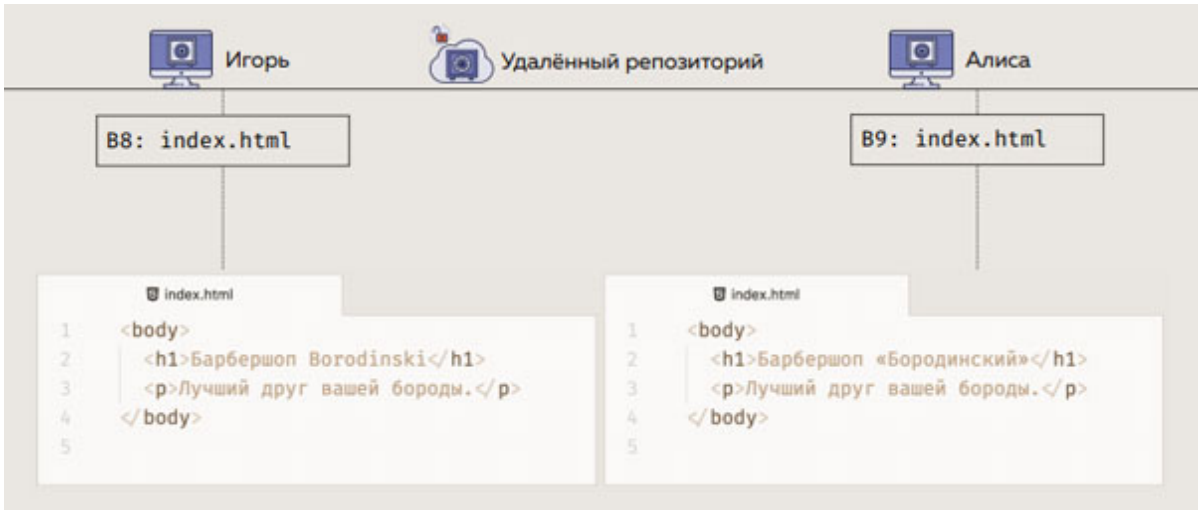
Игорь сделал шапку и отправил коммит, а Алиса добавила подвал.
Git видит, что произведённые изменения не затрагивают друг друга. Он сам объединит две версии проектов в одну, совершив слияние. После этого Алиса спокойно синхронизируется с удалённым репозиторием, отправив новую версию проекта.



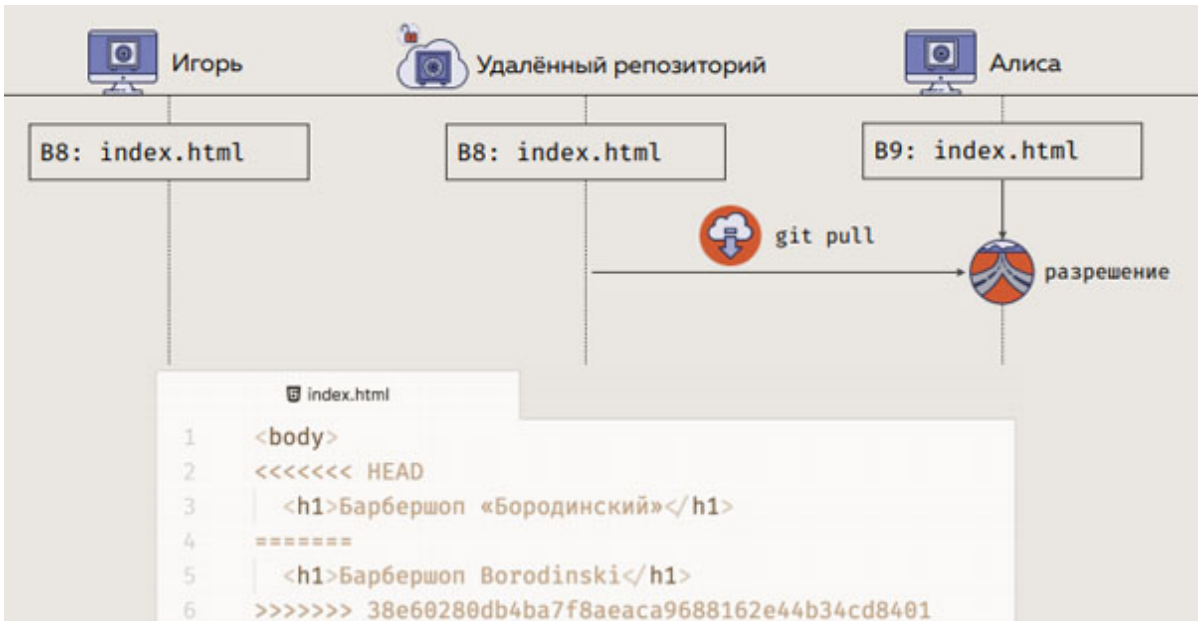
Изменения в проекте не пересекались и Git выполняет слияние сам.
Во время слияния Git не знает, в каком порядке расположить коммит B3 Игоря и коммит B4 Алисы, из-за которых случился конфликт. Поэтому Git разрешает существовать нескольким версиям проекта одновременно. Как раз для этого и нужен следующий коммит B5, в котором происходит слияние предыдущих параллельных версий. После того как Алиса запушит изменения, она отправляет все версии проектов на удалённый репозиторий. В следующий раз, когда Игорь сделает pull, он получит полную историю со слиянием конфликта.



Слияние.
Допустим, что Игорь и Алиса продолжили работать над проектом, но в этот раз изменили одинаковую строку в файле index.html. Вновь Игорь оказался быстрее и первым синхронизировал свои изменения с удалённым репозиторием. Алиса сделала pull и получила сообщение о конфликте.



Алиса и Игорь изменили один и тот же блок.
В таком случае Git не знает чья версия проекта правильная и поступает очень просто. Он изменяет файл index.html, добавляя в него изменения и Игоря и Алисы. После этого предупреждает Алису о конфликте и просит выбрать правильный вариант.



```
7     <p>Лучший друг вашей бороды.</p>
8   </body>
9
```