GIT

Всё необходимое в одном месте

Содержание презентации

Что такое Git [вдруг вы не знали]

Git — это распределённая система контроля версий (СКВ, VCS, Version Control Systems). Основная задача — хранить в течение времени состояния файлов (версии) и, в случае необходимости, давать возможность вернуться к любому из состояний.

Git появился в 2005 как основной инструмент контроля версий разработчиков ядра Linux (в их числе — Линус Торвальдс, создатель Linux). В новой СКВ старались учитывать чужие ошибки и сделать максимально эффективный инструмент для больших групп разработчиков (к разработке ядра причастно множество программистов со всего мира).

Системы контроля версий

Системы контроля версий (СКВ, VCS, Version Control Systems) позволяют разработчикам сохранять все изменения, внесённые в код.

В случае «аварии» вы сможете откатить код до рабочего состояния и не тратить время на отлов ошибок.

Также СКВ обеспечивает совместную работу над одним проектом для нескольких разработчиков. Внесённые изменения сохраняются независимо друг от друга. Каждый участник команды видит, над чем работают коллеги.

FTP vs GIT

Операция	FTP	GIT
Работа с удалённым сервером	да	да
Локальная работа (независимость от сервера)	требует дополнительных действий	да
Версионирование (сохранение истории изменений)	нет	да
Возврат к предыдущим состояниям	нет	да
Совместная работа	да/нет (могут возникать конфликты при работе с одними и теми же документами)	да
Логирование при совместной работе	нет	да
Бэкапы	нужны	не нужны

Преимущества Git

- **Бесплатный и open-source**
  Загрузка и редактирование кода — всё бесплатно.
- **Компактный и шустрый**
  Git очень быстрый из-за локального выполнения операций. Весь репозиторий хранится в небольшом файле, качество данных при этом не страдает.
- **Бэкапы**
  Ещё никто не жаловался на то, что потерял файлы во время работы с Git.
- **Простое ветвление**
  Git очень легко управляется с тысячами параллельных веток.
- **Децентрализованный**
  Клиенты скачивают репозиторий целиком. Если с сервером что-то случится, клиентский репозиторий может быть перенесён на другой сервер для продолжения работы.

Начинаем работать с GIT

Как и где использовать Git

Существует некоторое количество платных и бесплатных сервисов, которые позволяют использовать git для работы с ними. Вот некоторые из них:

- GitHub
- GitLab
- Bitbucket

Работая с этими сервисами вы получаете доступ к онлайн хранилищу с помощью интерфейса командной строки Git и можете использовать его для разработки вашего проекта, получая всю мощь децентрализованного удалённого репозитория с системой контроля версий.

Но если вам это не нужно - можно использовать Git локально и в одиночку.

Установка Git

Чтобы установить Git, скачайте Git для Windows, macOS или Linux и проверьте версию командой git --version.

Настройка конфигурационного файла

Первым делом — настройте имя пользователя и email для авторизации. Эти настройки хранятся в конфигурационном файле.
Вы можете напрямую отредактировать файл .gitconfig в текстовом редакторе или сделать это командой git config --global --edit.
Отдельные поля обрабатываются через git config --global <поле> <значение> — поля user.name и user.email.

Настройка конфигурационного файла

Также можно настроить текстовый редактор для написания сообщений коммитов, используя поле core.editor. А вот поле commit.template позволяет указать шаблон, который будет использоваться при каждом коммите.

Ещё одно полезное поле — alias, которое привязывает команду к псевдониму. Например, git config --global alias.st "status -s" позволяет использовать git st вместо git status –s

Команда git config --list выведет все поля и их значения из конфигурационного файла.

Создаём Git-репозиторий

Git хранит данные в виде набора снимков миниатюрной файловой системы. При каждом сохранении состояния проекта Git запоминает, как выглядит каждый файл в конкретный момент и сохраняет ссылку на этот снимок.

Для инициализации нового репозитория .git введите git init или, если хотите скопировать существующий, git clone <адрес репозитория>.

Локальные и удалённые репозитории

Просмотр изменений в репозитории

Команда git status отображает все файлы, которые были изменены.

У файлов есть 4 состояния:

1. Неотслеживаемый (untracked) — находится в рабочей директории, но отсутствуют версии в HEAD или в области подготовленных файлов (Git не знает о файле).
2. Изменён (modified) — в рабочей директории есть более новая версия, чем та, что хранится в HEAD или в области подготовленных файлов (изменения не находятся в следующем коммите).
3. Подготовлен (staged) — в рабочей директории и области подготовленных файлов находится более новая версия, чем та, что хранится в HEAD (готов к коммиту).
4. Без изменений — одна версия файла во всех разделах, то есть последний коммит содержит актуальную версию.

Просмотр изменений и сравнение

Чтобы получить более лаконичный вывод (по строке на файл), можете использовать опцию `-s` для команды `git status`.

Если файл не отслеживается, то выскочит `??`; если он был изменён, то имя загорится красным, а если подготовлен — зелёным.

Чтобы посмотреть сами изменения, а не изменённые файлы, можно использовать следующие команды:

- `git diff` — сравнение рабочей директории с областью подготовленных файлов;
- `git diff --staged` — сравнение области подготовленных файлов с HEAD.

Если использовать аргумент `<файл/папка>`, то `diff` покажет изменения только для указанных файлов/папок, например `git diff src/`.

Обновление файловых систем

Команда git add <файл/папка> обновляет область подготовленных файлов версиями файлов/папок из рабочей директории.

Команда git commit обновляет HEAD новым коммитом, который делает снимки файлов в области подготовленных файлов.

Игнорирование файлов

Зачастую нам не нужно, чтобы Git отслеживал все файлы в репозитории, потому что в их число могут входить:

- файлы с чувствительной информацией, например, пароли;
- большие бинарные файлы;
- файлы, специфичные для ОС/IDE, например, *.DS_Store* для macOS или *.iml* для IntelliJ IDEA — нам нужно, чтобы репозиторий как можно меньше зависел от системы.

Игнорирование файлов

Для игнорирования используется файл .gitignore. Чтобы отметить файлы, которые мы хотим игнорировать, можно использовать шаблоны поиска (считайте их упрощёнными регулярными выражениями):

- /___ — позволяет избежать рекурсивности — соответствует файлам только в текущей директории;
- __/ — соответствует всем файлам в указанной директории;
- ___ — соответствует всем файлам с указанным окончанием;
- ! — игнорирование файлов, попадающих под указанный шаблон;
- [__] — соответствует любому символу из указанных в квадратных скобках;
- ? — соответствует любому символу;
- /**/ — соответствует вложенным директориям, например, a/**/d соответствует a/d, a/b/d, a/b/c/d и т. д.

Удалённые репозитории

Итак, ранее мы говорили про Git только на локальной машине. В то же время, мы можем хранить историю коммитов удалённых репозиториев, которую можно отслеживать и обновлять.

`git remote -v` выводит список удалённых репозиториев, которые мы отслеживаем, и имена, которые мы им присвоили.

При использовании команды `git clone <url репозитория>` мы не только загружаем себе копию репозитория, но и неявно отслеживаем удалённый сервер, который находится по указанному адресу и которому присваивается имя origin.

Удалённые репозитории

Наиболее популярные команды:

- `git remote add <имя> <url>` — добавляет удалённый репозиторий с заданным именем;
- `git remote remove <имя>` — удаляет удалённый репозиторий с заданным именем;
- `git remote rename <старое имя> <новое имя>` — переименовывает удалённый репозиторий;
- `git remote set-url <имя> <url>` — присваивает репозиторию с именем новый адрес;
- `git remote show <имя>` — показывает информацию о репозитории.

Удалённые репозитории

Команды для работы с удалёнными ветками:

- git fetch <имя> <ветка> — получает данные из ветки заданного репозитория, но не сливает изменения;
- git pull <имя> <ветка> — сливает данные из ветки заданного репозитория;
- git push <имя> <ветка> — отправляет изменения в ветку заданного репозитория. Если локальная ветка уже отслеживает удалённую, то можно использовать просто git push или git pull.

Таким образом несколько людей могут запрашивать изменения с сервера, редактировать локальные копии и затем отправлять их на удалённый сервер, что позволяет взаимодействовать друг с другом в пределах одного репозитория.

Коммиты

Коммиты

Одна из ключевых вещей при работе с Git – это коммиты (commits).
Каждый коммит является точкой сохранения и применения изменений.
Команда `git commit` откроет текстовый редактор для ввода сообщения коммита.
В этой команде могу быть аргументы:

- `m` выводит сообщение вместе с командой, без участия редактора.
  Например, `git commit -m "Добавил комментариев"`;
- `a` переносит все отслеживаемые файлы в область подготовленных файлов и добавляет их в коммит (можете пропустить `git add` перед коммитом);
- `-amend` заменяет последний коммит новым изменённым коммитом – спасает, если вы неправильно набрали сообщение последнего коммита или забыли включить в него какие-то файлы.

История коммитов в Git

Коммиты хранят состояние файловой системы в определённый момент времени и отсылки на предыдущие коммиты.

Каждый коммит содержит уникальную контрольную сумму — идентификатор, который Git использует, чтобы ссылаться на коммит. Чтобы отслеживать историю, Git хранит указатель HEAD, который указывает на первый коммит (мы следуем по цепочке коммитов в обратном порядке, чтобы попасть к предыдущим коммитам).

История коммитов в Git

Мы можем ссылаться на коммит либо через его контрольную сумму, либо через его позицию относительно HEAD, например HEAD~4 ссылается на коммит, который находится 4 коммитами ранее HEAD.

Перенос отдельного коммита

Кроме слияния/перемещения всех коммитов в тематической ветке, вас может интересовать только определённый коммит.

Допустим, у вас есть локальная ветка drafts, где вы работаете над несколькими потенциальными статьями, но хотите опубликовать только одну из них.

Для этого можно использовать команду git cherry-pick. Чтобы получить определённые коммиты, из которых мы хотим выбирать, можно использовать git log <основная ветка>..<тематическая>.

Откат коммитов — revert и reset

Горячие споры возникают и тогда, когда вы хотите откатить коммит. Команда `git revert <коммит>` создаёт новый коммит, отменяющий изменения, но сохраняющий историю, в то время как `git reset <коммит>` перемещает указатель `HEAD`, предоставляя более чистую историю (будто этого коммита и не было вовсе).

Важно отметить, что это также означает, что вы больше не сможете вернуться обратно к этим изменениям, например, если вы всё-таки решите, что отмена коммита была лишней.

Чище — не значит лучше!

Работа с ветками

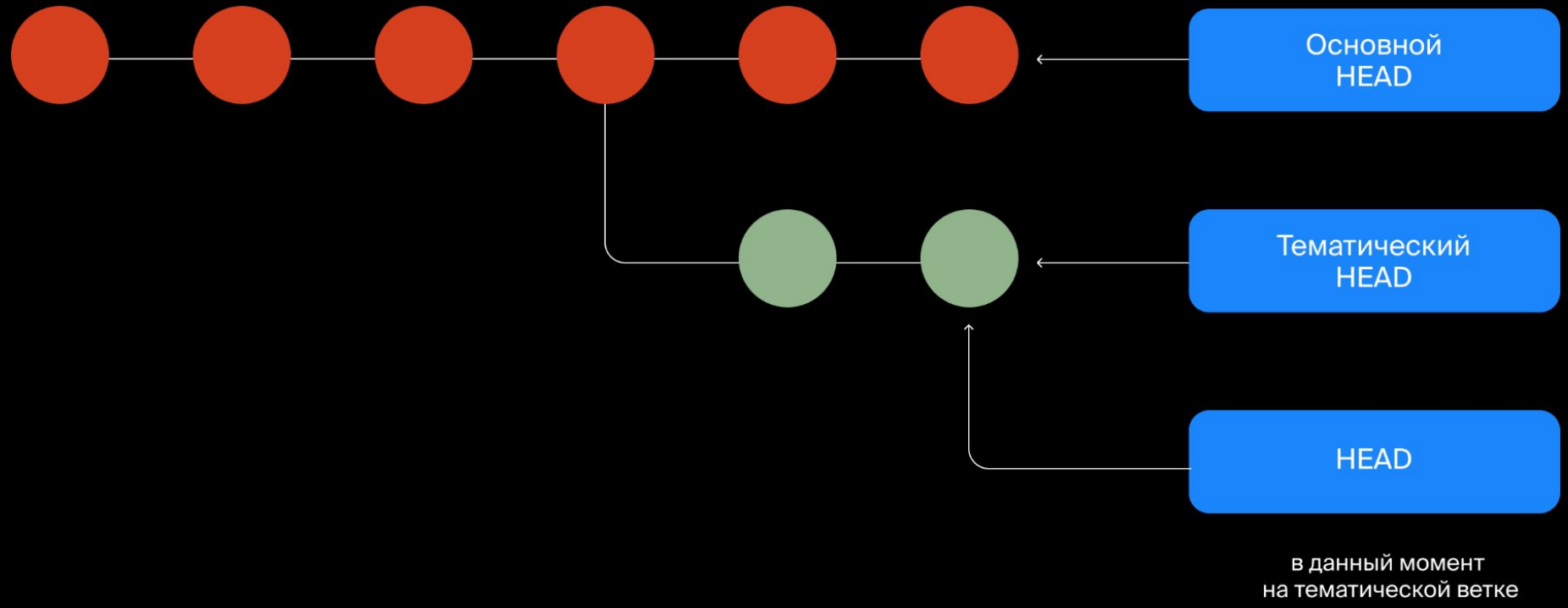
Работа с ветками

Ветвление — это возможность работать над разными версиями проекта: вместо одного списка с упорядоченными коммитами история будет расходиться в определённых точках.

Каждая ветвь содержит легковесный указатель HEAD на последний коммит, что позволяет без лишних затрат создать много веток.

Ветка по умолчанию называется master(иногда main), но лучше назвать её так, чтобы вы сразу понимали, для чего она создана(например, release). То же самое относится к веткам которые вы будете создавать сами.

Работа с ветками



У нас есть общий указатель **HEAD** и HEAD для каждой ветки. Переключение между ветками предполагает только перемещение **HEAD** в HEAD соответствующей ветки.

Работа с ветками

Команды:

- `git branch <имя ветки>` — создаёт новую ветку с HEAD, указывающим на **HEAD**. Если не передать аргумент `<имя ветки>`, то команда выведет список всех локальных веток;
- `git checkout <имя ветки>` — переключается на эту ветку. Можно передать опцию `b`, чтобы создать новую ветку перед переключением;
- `git branch -d <имя ветки>` — удаляет ветку.

Работа с ветками

Локальный и удалённый репозитории могут иметь достаточно ветвей, поэтому когда вы отслеживаете удалённый репозиторий — отслеживается удалённая ветка (git clone привязывает вашу ветку master к ветке origin/master удалённого репозитория).

Работа с ветками

Привязка к удалённой ветке:

- `git branch -u <имя удалённого репозитория>/<удалённая ветка>` — привязывает текущую ветку к указанной удалённой ветке;
- `git checkout --track <имя удалённого репозитория>/<удалённая ветка>` — аналог предыдущей команды;
- `git checkout -b <ветка> <имя удалённого репозитория>/<удалённая ветка>` — создаёт новую локальную ветку и начинает отслеживать удалённую;
- `git branch -vv` — показывает локальные и отслеживаемые удалённые ветки;
- `git checkout <удалённая ветка>` — создаёт локальную ветку с таким же именем, как у удалённой, и начинает её отслеживать.

В общем, `git checkout` связан с изменением места, на которое указывает HEAD ветки, что похоже на то, как `git reset` перемещает общий HEAD.

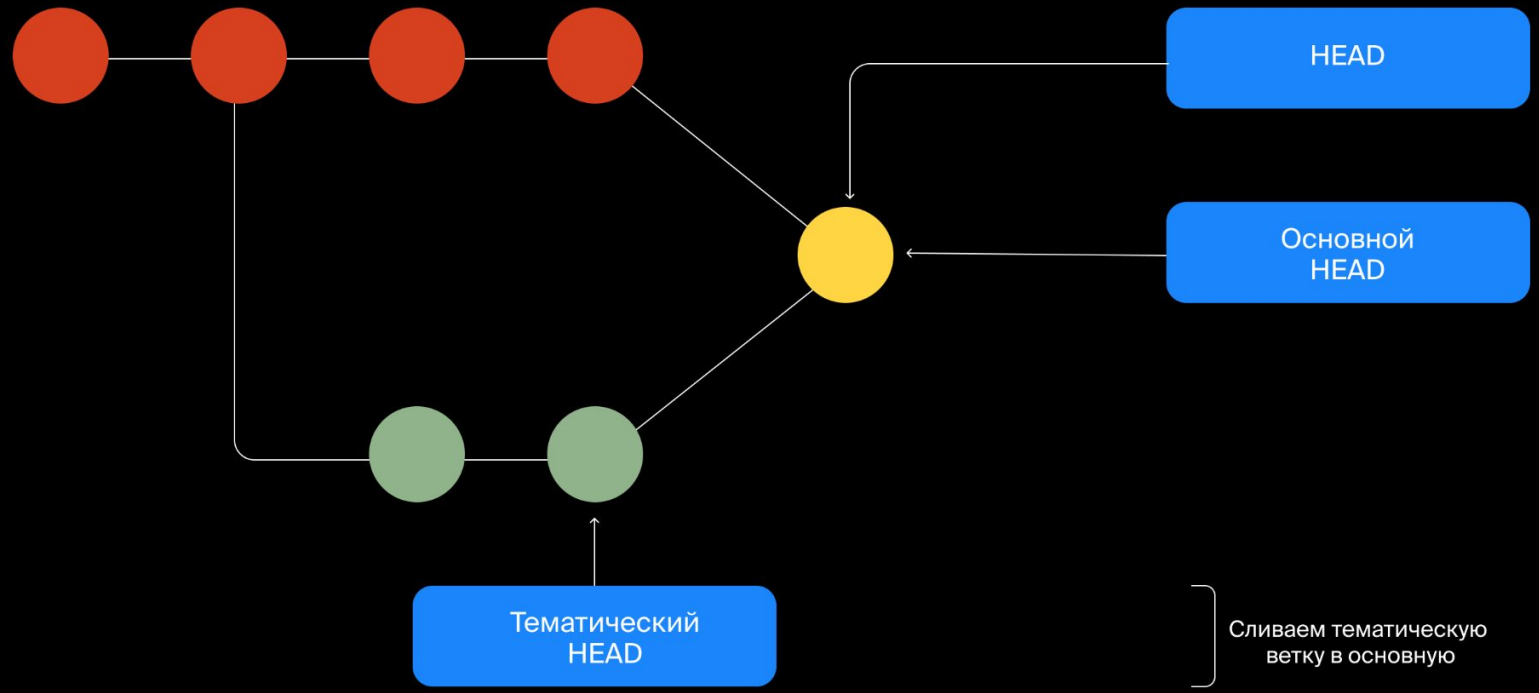
Merge

Ветку, в которую мы хотим добавить (слить) изменения, будем называть основной, а ветку, из которой мы будем их сливать, — тематической.

Слияние включает в себя создание нового коммита, который основан на общем коммите-предке двух ветвей и указывает на оба HEAD в качестве предыдущих коммитов.

Для слияния мы переходим на основную ветку и используем команду git merge <тематическая ветка>.

Merge



Если обе ветви меняют одну и ту же часть файла, то возникает конфликт слияния — ситуация, в которой Git не знает, какую версию файла сохранить, поэтому разрешать конфликт нужно вручную.

Чтобы увидеть конфликтующие файлы, вводите git status.

Merge

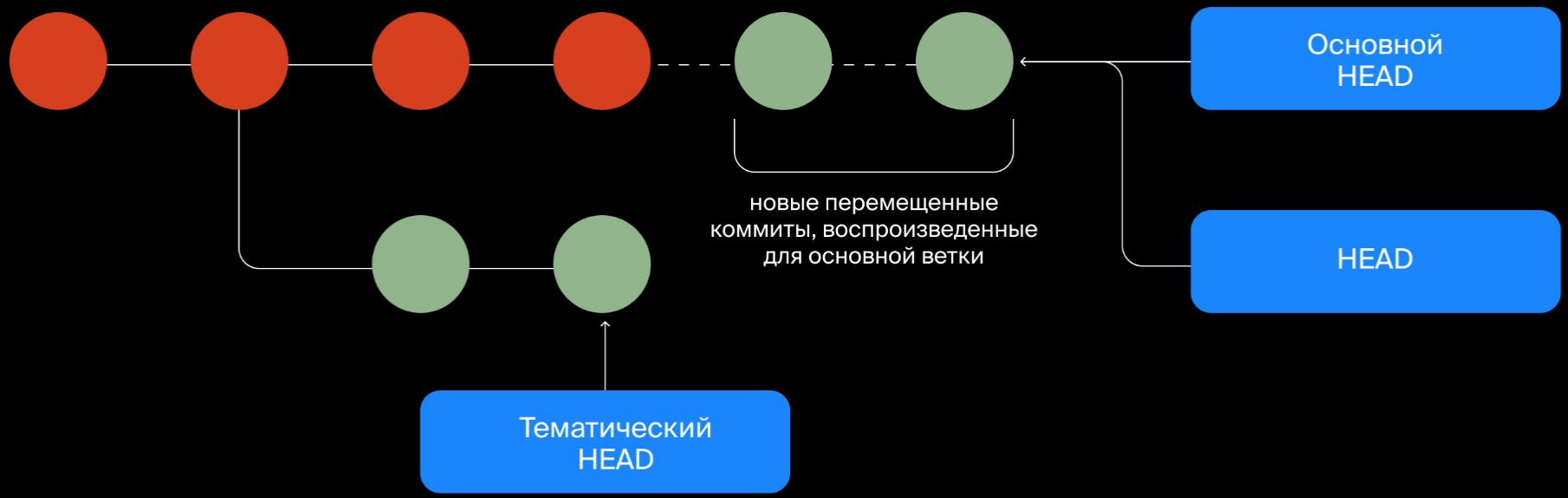
После открытия таких файлов вы увидите похожие маркеры разрешения конфликта:

```
<<<<<<< HEAD:index.html
Everything above the ==== is the version in master.
=======
Everything below the ==== is the version in the test branch.
>>>>>>> test:index.html
```

Замените в этом блоке всё на версию, которую вы хотите оставить, и подготовьте файл. После разрешения всех конфликтов можно использовать `git commit` для завершения слияния.

Rebase

Вместо совмещения двух ветвей коммитом слияния, перемещение заново воспроизводит коммиты тематической ветки в виде набора новых коммитов базовой ветки, что выливается в более чистую историю коммитов.



Для перемещения используется команда git rebase <основная ветка> <тематическая ветка>, которая воспроизводит изменения тематической ветки на основной; HEAD тематической ветки указывает на последний воспроизведённый коммит.

Rebase vs Merge

После слияния в логе с историей может возникнуть беспорядок.

С другой стороны, перемещение позволяет переписать историю в нормальной, последовательной форме.

Но перемещение не спасёт вас от запутанных логов: перемещённые коммиты отличаются от оригинальных, хотя и имеют одного и того же автора, сообщение и изменения.

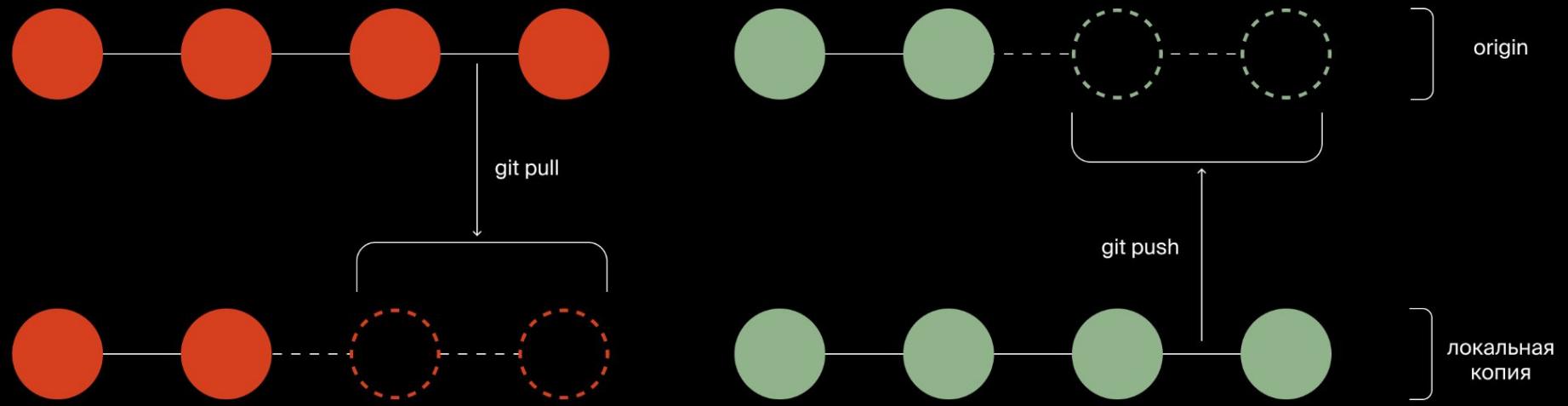
Rebase vs Merge

Как можно выкрутиться:

- Создаёте несколько коммитов в своей ветке и сливаете их в мастер-ветку.
- Кто-то ещё решает поработать на основе ваших коммитов.
- Вы решаете переместить ваши коммиты и отправить их на сервер.
- Когда кто-то попытается слить свою работу на основе ваших изначальных коммитов, в итоге мы получим две параллельные ветки с одним автором, сообщениями и изменениями, но разными коммитами.

Перемещайте изменения только на вашей приватной локальной ветке — не перемещайте коммиты, от которых зависит ещё кто-то.

Удалённые репозитории



Таким образом несколько людей могут запрашивать изменения с сервера, делать изменения в локальных копиях и затем отправлять их на удалённый сервер, что позволяет взаимодействовать друг с другом в пределах одного репозитория.

Объединение нескольких коммитов

Незначительные и небольшие коммиты могут засорить историю проекта. Особенно важно при слиянии веток. Поэтому несколько коммитов можно в один большой.

Объединение нескольких коммитов

Сжатие коммитов в интерактивном режиме:

- `git cherry -v master` — покажет, какие коммиты были сделаны в сравнение с веткой master.
- `git rebase -i HEAD~3:` ключ `-i` означает работу в интерактивном режиме, а `HEAD~3` — что объединяться будут последние 3 коммита.
- `pick` — это первый коммит.
- `squash` — коммиты, которые вы будете сливать с первым.

Сжатие коммитов при слиянии веток:

- `git merge --squash feature-branch` — коммиты из ветки feature-branch будут объединены и слиты с текущей веткой.

Отслеживание изменений в репозитории

Файловая система Git

Git отслеживает файлы в трёх основных разделах:

- рабочая директория (файловая система вашего компьютера);
- область подготовленных файлов (staging area, хранит содержание следующего коммита);
- HEAD (последний коммит в репозитории).

Просмотр изменений

Для просмотра истории предыдущих коммитов в обратном хронологическом порядке можно использовать команду `git log`.
Ей можно передать разные опции:

- `p` показывает изменения в каждом коммит
- `-stat` показывает сокращённую статистику для коммитов, например изменённые файлы и количество добавленных/удалённых строк в каждом их них;
- `n` показывает *n* последних коммитов;
- `-since=___` и `-until=___` позволяет отфильтровать коммиты по промежутку времени, например `-since="2019-01-01"` покажет коммиты с 1 января 2019 года;
- `-pretty` позволяет указать формат логов (например, `-pretty=oneline`), также можно использовать `-pretty=format` для большей кастомизации, например `-pretty=format:"%h %s"`;

Просмотр изменений

Для просмотра истории предыдущих коммитов в обратном хронологическом порядке можно использовать команду `git log`. Ей можно передать разные опции:

- `-grep` и `S` фильтруют коммиты с сообщениями/изменениями кода, которые содержат указанную строку, например, `git log -S имя_функции` позволяет посмотреть добавление/удаление функции;
- `-no-merges` пропускает коммиты со слиянием веток;
- `ветка1..ветка2` позволяет посмотреть, какие коммиты из ветки 2 не находятся в ветке 1 (полезно при слиянии веток). Например, `git log master..test` покажет, каких коммитов из ветки test нет в master (о ветках поговорим чуть позже);
- `-left-right ветка1...ветка2` показывает коммиты, которые есть либо в ветке 1, либо в ветке 2, но не в обеих; знак `<` обозначает коммиты из `ветка1`, а `>` — из `ветка2`;
- `L` принимает аргумент `начало,конец:файл` или `:функция:файл` и показывает историю изменений переданного набора строк или функции в файле: используются три точки, а не две.

Просмотр изменений

Другой полезной командой является git blame <файл>, которая для каждой строки файла показывает автора и контрольную сумму последнего коммита, который изменил эту строку. -L <начало>, <конец> позволяет ограничить эту команду заданными строками. Это можно использовать, например, для выяснения того, какой коммит привёл к определённому багу (чтобы можно было его откатить).

Наконец, есть команда git grep, которая ищет по всем файлам в истории коммитов (а не только в рабочей директории, как grep) по заданному регулярному выражению. Опция -n отображает соответствующий номер строки в файле для каждого совпадения, а --count показывает количество совпадений для каждого файла.

Примечание
Смотрите, не перепутайте git grep с git log --grep! Первый ищет по файлам среди коммитов, а последний ориентируется на сообщения логов.

Лайфхаки по работе с Git

- Чаще используйте коммиты.
- Одно изменение — один коммит: не помещайте все не связанные между собой изменения в один коммит, разделите их, чтобы проще откатиться.
- Формат сообщений: заголовок должен быть в повелительном наклонении, меньше 50 символов в длину и должен логически дополнять фразу this commit will ___(this commit will fix bugs — этот коммит исправит баги). Сообщение должно пояснять, почему был сделан коммит, а сам коммит показывает, что изменилось.
- Если у вас много незначительных изменений, хорошим тоном считается делать небольшие коммиты при разработке, а при добавлении в большой репозиторий объединять их в один коммит.

Занавес!

Тот, кто добрался до этого текста, благополучно вооружился знаниями по Git, либо предался ностальгии и заново прошёл избранные трюки и команды.

Вы всегда можете подсматривать в данную презентацию, пока движетесь по курсу DevOps. Пусть она будет полезной Git-шпаргалкой.