ГЛАВА 1

Знакомство с Git

Говоря простыми словами, Git — это система отслеживания содержимого, в ней используются те же принципы, что и в большинстве систем контроля версий. Тем не менее уникальной Git делает ее распределенная природа. Благодаря этому система быстрая и масштабируемая, имеет множество наборов команд, дающих доступ к высокоуровневым и низкоуровневым операциям, а также оптимизирована для выполнения локальных операций.

В этой главе вы узнаете фундаментальные принципы Git, ее характеристики и базовые команды. Вы также получите общее руководство по созданию репозитория и внесению в него изменений.

Настоятельно рекомендуем ознакомиться с приведенными здесь ключевыми принципами. Эти темы — строительные блоки Git, которые помогут понять общие и продвинутые техники управления репозиторием в рамках вашей ежедневной работы. Эти базовые принципы также ускорят ваше обучение, когда мы будем разбирать внутреннее устройство Git в части II «Основы Git», части III «Базовые навыки» и части IV «Продвинутые навыки».

Компоненты Git

Прежде чем углубиться в мир команд `git`, давайте поговорим о компонентах, составляющих экосистему Git. На рис. 1.1 показана взаимосвязь этих компонентов.

Инструменты графического интерфейса (GUI) Git выступают в качестве фронтенда для командной строки этой системы. Некоторые из них имеют расширения, которые интегрируются с популярными платформами хостинга Git. Клиентские инструменты Git в основном работают с локальной копией вашего репозитория.

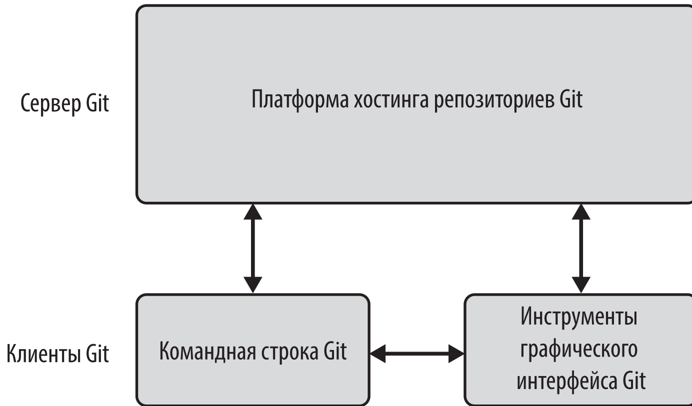


Рис. 1.1. Обзор компонентов Git

При работе с Git типичная конфигурация включает сервер и клиентов Git. Вы можете исключить сервер, но это усложнит поддержку и управление репозиториями при обмене вносимыми в новые ревизии изменениями в совместных проектах и затруднит согласованность (об этом поговорим в главе 11). Сервер и клиенты Git работают так:

Сервер Git

Сервер Git упрощает совместную работу, поскольку обеспечивает доступность центрального и достоверного источника истины для репозиториев, в которых вы будете работать. На этом сервере также хранятся ваши удаленные репозитории Git. Как это принято, в репозитории хранятся наиболее актуальные и стабильные исходники ваших проектов. Вы можете установить и настроить собственный сервер Git либо исключить эту дополнительную нагрузку и разместить свои репозитории в надежной сторонней системе — GitHub, GitLab или BitBucket.

Клиенты Git

Клиенты Git взаимодействуют с вашими локальными репозиториями, а вы можете взаимодействовать с клиентами через командную строку Git или инструменты GUI. Установив и настроив клиент Git, вы сможете получать доступ к удаленным репозиториям, работать над локальной копией репозитория, а также передавать изменения обратно на сервер Git. Если вы только знакомитесь с Git, то советую начать с командной строки, изучить наиболее распространенный набор команд `git`, необходимый для выполнения повседневных операций, а затем уже переходить к предпочтительному инструменту GUI.

Причина такого подхода в том, что инструменты GUI могут использовать собственные термины или абстракции, которые отличаются от стандартных команд Git . Например, инструмент с опцией `sync`, которая маскирует внутреннее связывание в цепочку двух или более команд `git` для получения нужного результата. Если по какой-то причине вы введете подкоманду `sync` в командную строку, то можете получить такой непонятный вывод:

```
$ git sync

git: 'sync' is not a git command. See 'git --help'.

The most similar command is
       svn
```

`git sync` не является допустимой подкомандой `git`. Чтобы обеспечить синхронизацию локальной копии репозитория с изменениями из удаленного репозитория Git, нужно выполнить комбинацию следующих команд: `git fetch`, `git merge`, `git pull` или `git push`.

Вам доступно огромное число инструментов GUI. Некоторые из них достаточно сложны и могут расширяться через подключаемую модель, позволяющую использовать функциональность сторонних сервисов хостинга Git. Несмотря на все удобство освоения Git через GUI, мы будем применять инструмент командной строки, чтобы заложить прочную основу знаний и дать вам бо́льшую гибкость при работе с этой системой.

Характеристики Git

Теперь, когда у нас есть общее представление о компонентах Git, перейдем к знакомству с характеристиками системы. Их понимание позволит вам перестроить ментальную модель с централизованного управления версиями на распределенное. Я называю это «философией Git».

Git сохраняет изменения ревизий в виде снимков

Первый принцип: Git сохраняет множество ревизий файла, над которым вы работаете. В отличие от других систем контроля версий, Git не отслеживает изменение ревизий в виде серии модификаций, которые обычно называют *дельтами*. Git делает моментальные снимки (снапшоты) внесенных в репозиторий изменений в конкретные моменты времени. В терминологии Git это называется *фиксацией* или *коммитом*. Это можно сравнить с фиксацией момента во времени при фотосъемке.

Git расширена под локальную разработку

В Git вы работаете с копией репозитория на своей локальной машине для разработки. Эта копия называется *локальным репозиторием* или клоном удаленного

репозитория на сервере Git. Ваш локальный репозиторий будет иметь ресурсы и снимки изменений ревизии, внесенные в эти ресурсы, все в одном месте. В Git эти коллекции связанных снимков называются *историей коммитов репозитория* или просто *историей репозитория*. Это позволяет вам работать в автономной среде, поскольку Git не нуждается в постоянном подключении к серверу для контроля версий ваших изменений. В результате вы получаете возможность работать над крупными сложными проектами распределенными командами без снижения эффективности и качества операций контроля версий.

Git действует в соответствии с инструкциями

Команды `git` — явные. Git ожидает, пока вы предоставите инструкции о том, что и когда нужно сделать. Например, Git не будет автоматически синхронизировать изменения из локального репозитория с удаленным репозиторием, равно как и сохранять снимок ревизии в историю локального репозитория. Каждое действие ожидает от вас явной команды или инструкции, которая сообщит системе, что от нее требуется, включая создание новых коммитов, исправление существующих, передачу изменений с локального репозитория на удаленный и даже извлечение последних изменений с удаленного репозитория. Вам нужно проявлять свои намерения в действиях. Сюда же относится указание Git, какие файлы вы хотите отслеживать, поскольку система не будет автоматически добавлять для контроля версий новые файлы.

Git предназначена для поддержки нелинейной разработки

С Git вы можете создавать различные реализации функций и экспериментировать с ними, искать оптимальные решения для вашего проекта, отклоняться в сторону от линии основной, стабильной кодовой базы и работать параллельно с ней. Эта методология называется *ветвлением* (branching). Это очень распространенная практика, которая обеспечивает целостность основной линии разработки и исключает любые случайные изменения, которые могли бы ее нарушить.

В Git принцип ветвления считается легковесным и незатратным, поскольку ветка — это всего лишь указатель на последний коммит в серии связанных коммитов. Git отслеживает серию коммитов в каждой создаваемой вами ветке. Вы можете переключаться между ветками локально. После этого система сохраняет состояние проекта в последний момент, в который был создан снимок указанной ветки. Когда вы решите внести изменения из какой-либо ветки в основную линию разработки, Git совместит эту серию коммитов, применив методики, о которых пойдет речь в главе 6.

Поскольку Git предлагает множество новшеств, нужно помнить, что принципы и практики других систем контроля версий могут работать здесь иначе или вообще не подходить.

Командная строка Git

Интерфейс командной строки Git очень прост и дает пользователю полный контроль. В связи с этим есть много способов реализовать одно и то же действие. Сосредоточьтесь на тех командах, которые важны в вашей повседневной работе, тем самым вы сможете упростить их изучение.

Для начала выполните **git version** или **git --version**, чтобы определить, установлена ли у вас Git. Вывод должен быть примерно таким:

```
$ git --version
git version 2.37.0
```

Если в вашей системе нет Git, загляните в приложение Б, где описано, как установить ее на вашу ОС, и уже потом переходите к следующему разделу.

Перед установкой выполните **git** без аргументов. На это Git выдаст список доступных опций и наиболее актуальных подкоманд:

```
$ git

usage: git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           [--super-prefix=<path>] [--config-env=<name>=<envvar>]
           <command> [<args>]
```

Вот список распространенных команд Git для различных ситуаций:

```
Запуск рабочей области (также см.: git help tutorial)
   clone     Копирует репозиторий в новый каталог
   init      Создает пустой репозиторий Git либо повторно инициализирует текущий

Работа с текущими изменениями (также см.: git help everyday)
   add       Добавляет в индекс содержимое файла
   mv        Перемещает или переименовывает файл, каталог или символическую
             ссылку
   restore   Восстанавливает файлы рабочего дерева
   rm        Удаляет файлы из рабочего дерева и индекса

Анализ истории и состояния (также см.: git help revisions)
   bisect    С помощью двоичного поиска ищет коммит, который внес ошибку
   diff      Показывает изменения между коммитами, коммитом и рабочим деревом
             и т. д.
   grep      Выводит строки, соответствующие паттерну
   log       Показывает логи коммитов
   show      Показывает различные типы объектов
   status    Показывает статус рабочего дерева

Расширение, разметка и корректировка истории
   branch    Вывод списка, создание или удаление ветвей
   commit    Регистрирует изменения в репозитории
```

```
   merge     Объединяет две или более истории разработки
   rebase    Повторно применяет коммиты поверх другого базового коммита
   reset     Сбрасывает текущую HEAD до указанного состояния
   switch    Переключает ветви
   tag       Создание, вывод списка, удаление или проверка объекта тега,
             подписанного с помощью GPG

Совместная работа (также см.: git help workflows)
   fetch    Скачивает объекты и ссылки из другого репозитория
   pull     Получение из/интеграция с другим репозиторием или локальной веткой
   push     Обновляет удаленные ссылки вместе со связанными объектами

'git help -a' и 'git help -g' выводят доступные подкоманды и некоторые
руководства по принципам.  Для ознакомления
с конкретной подкомандой или принципом введите 'git help <command>'
или 'git help <concept>'
Для общего обзора системы выполните 'git help git'.
```

Для вывода всего списка подкоманд `git` введите **`git help --all`**.

Как видно из подсказки, к `git` применимо не так много опций. Большинство из них, представленные как `[ARGS]`, применимы к конкретным подкомандам.

Например, опция `--version` влияет на команду `git` и выводит номер версии:

```
$ git --version
git version 2.37.0
```

В противоположность этому `--amend` является примером опции, специфичной для подкоманды `git.commit`:

```
$ git commit --amend
```

Некоторые вызовы требуют использования обеих опций (здесь дополнительные пробелы в командной строке просто служат для визуального отделения подкоманды от базовой команды и необязательны):

```
$ git --git-dir=project.git     repack -d
```

При этом можно получить документацию для каждой подкоманды `git`, используя `git help subcommand`, `git --help subcommand`, `git subcommand --help` или `man git-` *`subcommand`*.

Полная документация по Git доступна онлайн (https://oreil.ly/7njp8).

Исторически Git представляла собой набор из множества простых, отдельных и самостоятельных команд, разработанных в соответствии с философией Unix, согласно которой нужно создавать небольшие, функционально совместимые инструменты. Каждая команда имела название с дефисом, например `git-commit` и `git-log`. Современные версии Git больше не поддерживают такие виды команд и используют единый исполняемый элемент `git` с подкомандой.

Команды `git` понимают как «короткие», так и «длинные» опции. Например, для команды `git commit` следующие примеры будут равнозначны:

```
$ git commit -m "Fix a typo."
$ git commit --message="Fix a typo."
```

В короткой форме, `-m`, используется один дефис, в то время как в длинной, `--message`, их два. (Это соответствует стилю длинных опций GNU.) Некоторые опции могут быть доступны только в одной из форм.

Вы можете создать сводку по коммиту и подробное сообщение для этой сводки, используя опцию `-m` дважды:

```
$ git commit -m "Summary" -m "Detail of Summary"
```

Наконец, вы можете отделять опции от списка аргументов с помощью *двойного тире*. Например, используйте двойное тире для отделения управляющей части командной строки от списка операндов, таких как имена файлов:

```
$ git diff -w main origin -- tools/Makefile
```

Вам может потребоваться использовать двойное тире для отделения и явного обозначения имен файлов, чтобы не спутать их с остальной частью команды. К примеру, если у вас будут и файл, и тег под именем `main.c`, то потребуется выполнять операции по-отдельности:

```
# Переключиться на тег "main.c"
$ git checkout main.c

# Переключиться на файл "main.c"
$ git checkout -- main.c
```

Использование Git: быстрый старт

Чтобы посмотреть Git в действии, создайте новый репозиторий, внесите в него содержимое и отследите несколько ревизий. Создать репозиторий можно двумя путями: либо с нуля, заполнив его содержимым, либо работать с существующим, *клонировав* его с удаленного сервера Git.

Подготовка к работе с Git

Независимо от того, создаете вы новый репозиторий или же работаете с существующим, есть базовые конфигурации, которые нужно настроить после установки Git на локальную машину. Подобным образом вы настроили бы новую фотокамеру: сначала установили корректную дату и язык и только потом начали фотографировать.

Как минимум Git потребует от вас ввести имя и адрес электронной почты, прежде чем создать первый коммит в репозитории. Предоставленный вами идентификатор отображается в качестве автора коммита вместе с другими метаданными снимка. Вы можете сохранить свой идентификатор в конфигурационном файле с помощью команды `git config`:

```
$ git config user.name "Jon Loeliger"
$ git config user.email "jdl@example.com"
```

Если вы решите не вносить данные в файл конфигурации, то придется указывать их при выполнении каждой подкоманды `git commit`, дополняя ее аргументом `--author` в конце:

```
$ git commit -m "log message" --author="Jon Loeliger <jdl@example.com>"
```

Имейте в виду, что это трудоемкий вариант, который быстро вам надоест.

Вы также можете указать свой идентификатор, передав имя и e-mail в переменных `GIT_AUTHOR_NAME` и `GIT_AUTHOR_EMAIL` соответственно. Если эти переменные установлены, то они переопределяют все настройки конфигурации. Для спецификаций, устанавливаемых в командной строке, Git будет переопределять значения, передаваемые в файле конфигурации и переменной среды.

Работа с локальным репозиторием

Теперь, когда вы настроили идентификатор, можно начать работу с репозиторием. Первым делом создайте новый репозиторий на локальной машине. Мы начнем с простого и будем постепенно продвигаться к изучению методик работы с общим репозиторием на сервере Git.

Создание начального репозитория

Смоделируем типичную ситуацию, создав репозиторий для личного сайта. Предположим, что вы начинаете с нуля и собираетесь добавить содержимое для вашего проекта в локальный каталог ~/my_website, который поместите в репозиторий Git.

Введите следующие команды для создания этого каталога и поместите в файл `index.html` какое-нибудь содержимое:

```
$ mkdir ~/my_website
$ cd ~/my_website
$ echo 'My awesome website!' > index.html
```

Чтобы преобразовать ~/my_website в репозиторий Git, выполните `git init`. Здесь мы сопровождаем эту команду опцией `-b` и предустановленным именем ветки `main`:

```
$ git init -b main

Инициализирован пустой репозиторий Git в ../my_website/.git/
```

Если вы хотите сначала инициализировать пустой репозиторий, а затем добавить в него файлы, то можете сделать это с помощью следующих команд:

```
$ git init -b main ~/my_website

Инициализирован пустой репозиторий Git в ../my_website/.git/

$ cd ~/my_website
$ echo 'My awesome website!' > index.html
```

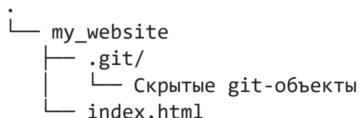
Вы можете инициализировать как абсолютно пустой каталог, так и существующий каталог с файлами. В любом случае процесс преобразования каталога в репозиторий Git будет одинаковым.

Команда `git init` создает скрытый каталог .git на корневом уровне проекта. Вся информация о ревизии вместе с метаданными и расширениями Git хранится на верхнем уровне, в скрытом каталоге .git.

Git воспринимает ~/my_website как *рабочий каталог*. В нем содержится текущая версия файлов для вашего сайта. Когда вы вносите изменения в текущие файлы или добавляете в проект новые, Git регистрирует эти изменения в скрытом каталоге .git.

Чтобы проиллюстрировать принцип инициализации нового репозитория Git, мы будем ссылаться на два виртуальных каталога, которые назовем Local History и Index. Каталоги Local History и Index будем разбирать в главах 4 и 5 соответственно.

Изложенное выше показано на рис. 1.2.



Пунктирные линии вокруг Local History и Index представляют скрытые каталоги в директории .git.

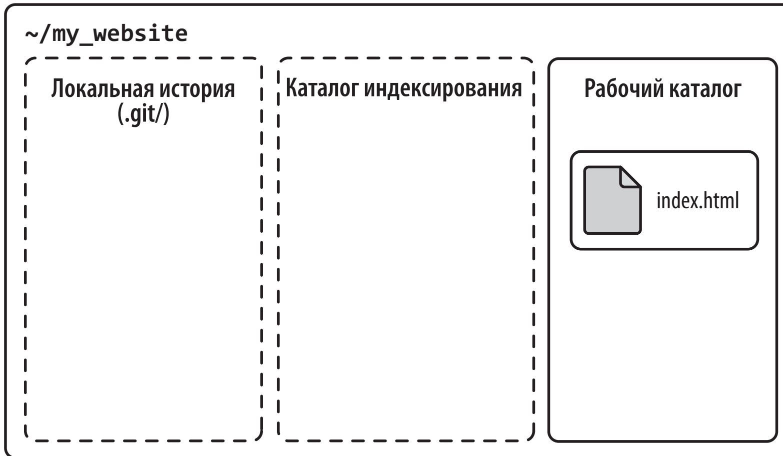


Рис. 1.2. Начальный репозиторий

Добавление файла в репозиторий

К этому моменту вы лишь *создали* новый репозиторий. Иными словами, этот репозиторий пуст. И хотя в каталоге ~/my_website есть файл index.html, для Git это *рабочий каталог*, представляющий собой своего рода блокнот или каталог, где вы часто изменяете свои файлы.

Когда вы закончили с изменением файлов и хотите внести эти изменения в репозиторий, сделайте это явно с помощью команды `git add` *`file`*:

```
$ git add index.html
```

Хотя вы и можете позволить Git добавить все файлы в каталоге и его подкаталогах, используя команду `git add .`, это проиндексирует все содержимое. Советуем избирательно подходить к тому, что вы хотите проиндексировать: главным образом для того, чтобы исключить передачу в коммитах чувствительной информации или нежелательных файлов. Для этого вы можете использовать файл `.gitignore`, который будет описан в главе 5.

Аргумент `.` (точка) — это сокращение для текущего каталога.

С помощью команды `git add` вы даете Git понять, что хотите включить в репозиторий в качестве ревизии последнюю итерацию изменений файла index.html. До сих пор Git просто размещал файл, что было промежуточным шагом перед созданием снимка с помощью коммита.

Git разделяет этапы `add` и `commit`, чтобы избежать нестабильности и в то же время обеспечить гибкость и детальность в фиксировании изменений. Представьте, насколько раздражающим, запутанным и долгим было бы обновление репозитория каждый раз, когда вы добавляете, удаляете или изменяете файл. Вместо этого

несколько предварительных и связанных с этим процессом шагов, например добавление, можно *сгруппировать*, сохранив репозиторий в стабильном и согласованном состоянии. Этот метод также позволит создать описание причин изменений в коде. В главе 4 мы разберем этот принцип подробно.

Старайтесь группировать пакеты логических изменений, прежде чем делать коммит. Это называется *атомарным* коммитом, он поможет в ситуациях, требующих выполнения продвинутых операций, о которых речь пойдет в следующих главах.

Запуск команды `git status` показывает промежуточный статус файла index.html:

```
$ git status

On branch main

No commits yet

Changes to be committed:

(use "git rm --cached <file>..." to unstage)
        new file: index.html
```

Эта команда сообщает, что во время следующего коммита в репозиторий будет добавлен новый файл index.html.

После индексирования этого файла следующим шагом будет его отправка в репозиторий. После коммита этот файл станет частью *истории репозитория*. При каждом выполнении коммита Git записывает вместе с ним другие элементы данных, в частности сообщение коммита и автора вносимого изменения.

При выполнении команды `git commit` разработчику, вносящему коммит, нужно предоставить ясное и информативное сообщение в журнале коммитов, в котором указано, что именно было изменено или добавлено. Это очень помогает, когда вам нужно перебрать историю репозитория для прослеживания конкретного изменения или быстро определить изменения коммита, не углубляясь в их детали. Эту тему мы более подробно разберем в главах 4 и 8.

Зафиксируем проиндексированный файл index.html для вашего сайта:

```
$ git commit -m "Initial contents of my_website"

[main (root-commit) c149e12] Initial contents of my_website
 1 file changed, 1 insertion(+)
 create mode 100644 index.html
```

Подробности об авторе, сделавшем коммит, извлекаются из конфигурации Git, которую мы настроили ранее.

В этом примере мы передали аргумент `-m`, чтобы можно было предоставить сообщение журнала напрямую в командной строке. Если вы предпочитаете

предоставлять подробное сообщение журнала через интерактивный сеанс редактора, то это тоже вполне допустимо. Для этого настройте Git для запуска редактора во время `git commit` (опустите аргумент `-m`); если он еще не настроен, то можете установить переменную среды `$GIT_EDITOR` так:

```
# В bash или zsh
$ export GIT_EDITOR=vim

# В tcsh
$ setenv GIT_EDITOR emacs
```

Git будет ориентироваться на базовый текстовый редактор, установленный в переменных среды оболочки `VISUAL` и `EDITOR`. Если ни один из них не установлен, Git вернется к редактору `vi`.

После коммита файла index.html в репозиторий выполните `git status` для получения обновления по текущему состоянию этого репозитория. В нашем примере выполнение `git status` должно показывать, что для коммита нет ожидающих изменений:

```
$ git status

On branch main
nothing to commit, working tree clean
```

Git также сообщает, что ваш *рабочий каталог* пуст, то есть не содержит новых или измененных файлов, отличающихся от находящихся в репозитории.

На рис. 1.3 изображены все описанные выше шаги.

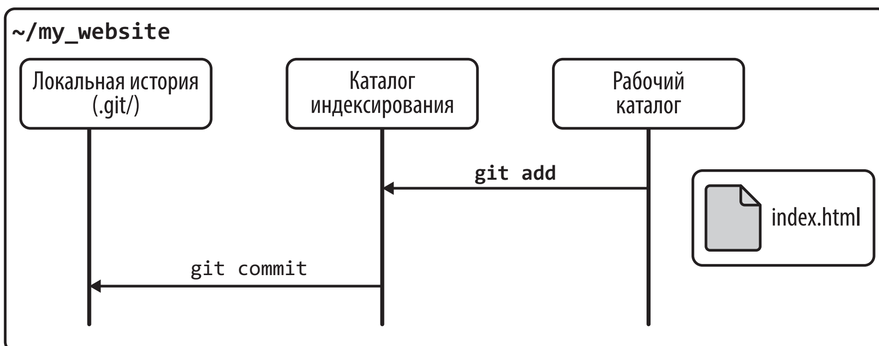


Рис. 1.3. Индексирование и добавление файла в репозиторий

Разницу между `git add` и `git commit` можно сравнить с тем, как вы организуете группу школьников в нужном порядке для создания идеальной общей фотографии: `git add` выполняет организацию, а `git commit` делает снимок.