Объектно-ориентированное программирование

Лекция 1. Система контроля версий Git

Что такое контроль версий и зачем он нужен?

Система контроля версий (СКВ) — это система, регистрирующая изменения в одном или нескольких файлах с тем, чтобы в дальнейшем была возможность вернуться к определённым старым версиям этих файлов.

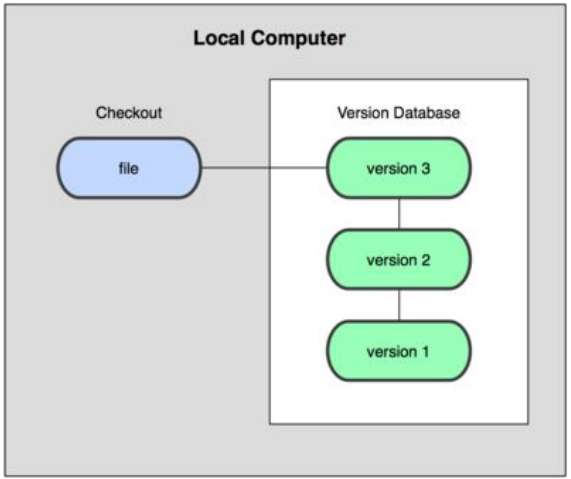
Виды:

- локальные
- централизованные
- распределенные

Локальные системы контроля версий

Одной из наиболее популярных СКВ такого типа является RCS (Revision Control System, Система контроля ревизий).
Для каждого файла находящегося под контролем системы информация о версиях хранится в специальном файле с именем оригинального файла к которому в конце добавлены символы `',v'`.
Эта утилита основана на работе с наборами патчей между парами версий (патч — файл, описывающий различие между файлами). Это позволяет пересоздать любой файл на любой момент времени, последовательно накладывая патчи. Для хранения версий система использует утилиту `diff`.
Хотя RCS соответствует минимальным требованиям к системе контроля версий она имеет следующие основные недостатки, которые также послужили стимулом для создания следующей рассматриваемой системы:

- Работа только с одним файлом, каждый файл должен контролироваться отдельно;
- Неудобный механизм одновременной работы нескольких пользователей с системой, хранилище просто блокируется пока заблокировавший его пользователь не разблокирует его;
- От бекапов вас никто не освобождает, вы рискуете потерять всё.

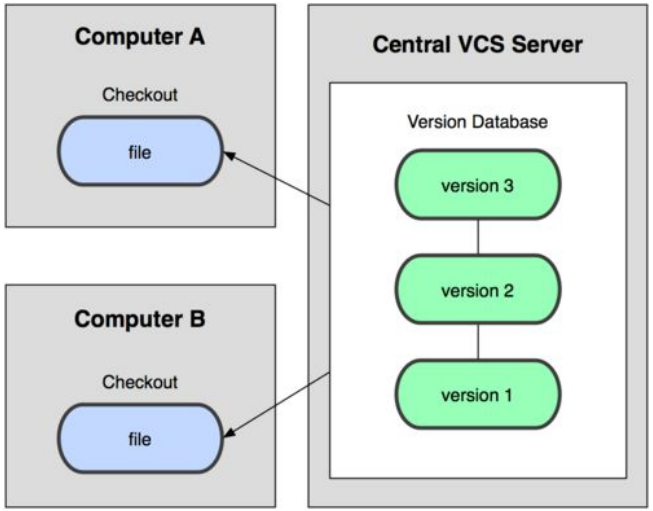


Централизованные системы контроля версий

В таких системах, например CVS, Subversion и Perforce, есть центральный сервер, на котором хранятся все файлы под версионным контролем, и ряд клиентов, которые получают копии файлов из него. Много лет это было стандартом для систем контроля версий.

Такой подход имеет множество преимуществ, особенно над локальными СКВ. К примеру, все знают, кто и чем занимается в проекте. У администраторов есть чёткий контроль над тем, кто и что может делать, и, конечно, администрировать ЦСКВ намного легче, чем локальные базы на каждом клиенте.

Однако при таком подходе есть и несколько серьёзных недостатков. Наиболее очевидный — централизованный сервер является уязвимым местом всей системы. Если сервер выключается на час, то в течение часа разработчики не могут взаимодействовать, и никто не может сохранить новой версии своей работы. Если же повреждается диск с центральной базой данных и нет резервной копии, вы теряете абсолютно всё — всю историю проекта, разве что за исключением нескольких рабочих версий, сохранившихся на рабочих машинах пользователей.

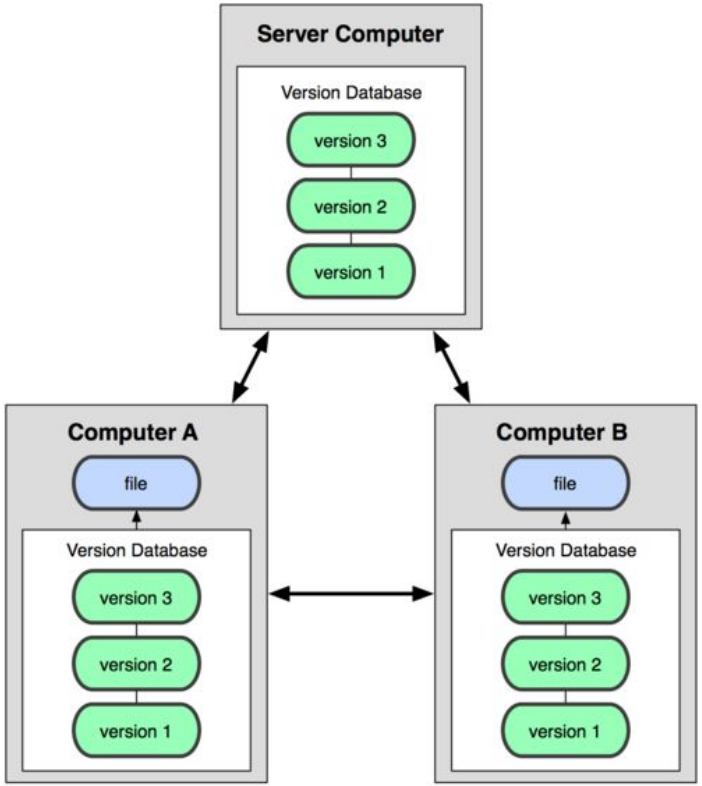


Распределенные системы контроля версий

В таких системах как Git, Mercurial, Bazaar или Darcs клиенты не просто выгружают последние версии файлов, а полностью копируют весь репозиторий. Поэтому в случае, когда “умирает” сервер, через который шла работа, любой клиентский репозиторий может быть скопирован обратно на сервер, чтобы восстановить базу данных. Каждый раз, когда клиент забирает свежую версию файлов, он создает себе полную копию всех данных.

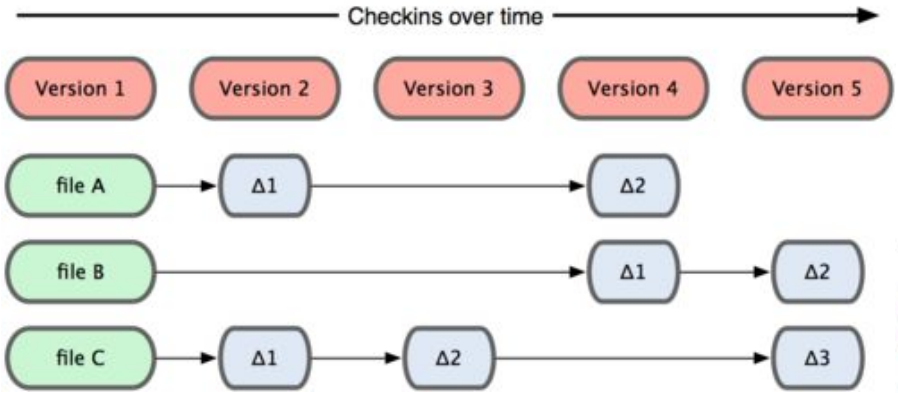
Основные идеи:

- набор версий может быть полностью, или частично распределен между различными хранилищами, в том числе и удаленными
- Если в случае централизованных систем всегда есть один общий репозиторий откуда можно получить последнюю версию проекта, то в случае распределенных систем нужно организационно решить какая из веток проекта будет основной.
- механизм объединения изменений с одной ветки на другую в случае распределенных систем является одним из основных, что позволяет пользователям прикладывать меньше усилий при пользовании системой.



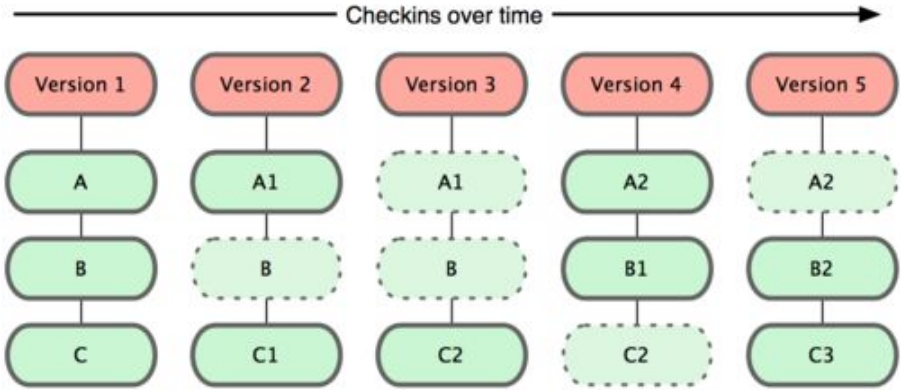
Основы Git

1) слепки вместо патчей



Другие системы хранят данные как изменения к базовой версии для каждого файла.

Git хранит данные как слепки состояний проекта во времени.



Основы Git

---

2) почти все операции - локальные

3) Git следит за целостностью данных

> Механизм, используемый Git’ом для вычисления контрольных сумм, называется SHA-1 хешем. Это строка из 40 шестнадцатеричных символов (0-9 и a-f), вычисляемая в Git’е на основе содержимого файла или структуры каталога.

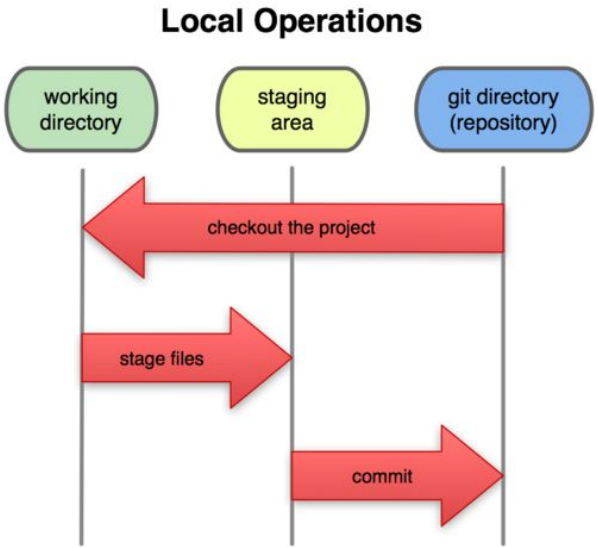
4) чаще всего данные в Git только добавляются

Основы Git. Состояния файлов

В Git’е файлы могут находиться в одном из трёх состояний: зафиксированном, изменённом и подготовленном. “Зафиксированный” значит, что файл уже сохранён в вашей локальной базе. К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы. Подготовленные файлы – это изменённые файлы, отмеченные для включения в следующий коммит.

Таким образом, в проектах, использующих Git, есть три части:

- каталог Git’а (Git directory),
- рабочий каталог (working directory),
- область подготовленных файлов (staging area).



Основы Git. Состояния файлов

Каталог Git’а — это место, где Git хранит метаданные и базу данных объектов вашего проекта. Это наиболее важная часть Git’а, и именно она копируется, когда вы клонируете репозиторий с другого компьютера.

Рабочий каталог — это извлечённая из базы копия определённой версии проекта. Эти файлы достаются из сжатой базы данных в каталоге Git’а и помещаются на диск для того, чтобы вы их просматривали и редактировали.

Область подготовленных файлов — это обычный файл, обычно хранящийся в каталоге Git’а, который содержит информацию о том, что должно войти в следующий коммит. Иногда его называют индексом (index), но в последнее время становится стандартом называть его областью подготовленных файлов (staging area).

Основы Git. Состояния файлов

Стандартный рабочий процесс с использованием Git’а выглядит примерно так:

- Вы вносите изменения в файлы в своём рабочем каталоге.
- Подготавливаете файлы, добавляя их слепки в область подготовленных файлов.
- Делаете коммит, который берёт подготовленные файлы из индекса и помещает их в каталог Git’а на постоянное хранение.

Если рабочая версия файла совпадает с версией в каталоге Git’а, файл считается зафиксированным. Если файл изменён, но добавлен в область подготовленных данных, он подготовлен. Если же файл изменился после выгрузки из БД, но не был подготовлен, то он считается изменённым.

Установка и первоначальная настройка Git

Linux

```
$ yum install git-core
```

```
$ apt-get install git
```

Windows

```
http://msysgit.github.com/
```

Указание имени и адреса электронной почты

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Помощь

```
$ git help <команда>
$ git <команда> --help
$ man git-<команда>
```

Проверка настроек

```
$ git config --list
```

Создание Git-репозитория

Для создания Git-репозитория существуют два основных подхода. Первый подход — импорт в Git уже существующего проекта или каталога. Второй — клонирование уже существующего репозитория с сервера.

1) Создание репозитория в существующем каталоге

```
$ git init
```

Если вы хотите добавить под версионный контроль существующие файлы (в отличие от пустого каталога), вам стоит проиндексировать эти файлы и осуществить первую фиксацию изменений. Осуществить это вы можете с помощью нескольких команд `git add` указывающих индексируемые файлы, а затем `commit`:

```
$ git add *.c
$ git add README
$ git commit -m 'initial project version'
```

Создание Git-репозитория

2) клонирование существующего репозитория

```
$ git clone git://github.com/libgit2/rugged.git
```

В Git’е реализовано несколько транспортных протоколов, которые вы можете использовать. В предыдущем примере использовался протокол `git://`, вы также можете встретить `http(s)://` или `user@server:/path.git`, использующий протокол передачи SSH.

Запись изменений в репозиторий

Определение состояния файлов

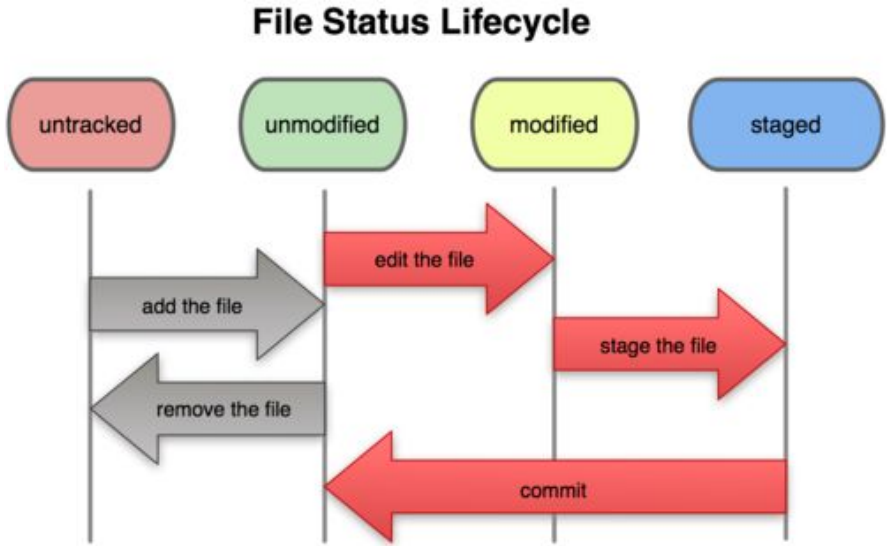
```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Отслеживание новых файлов

```
$ git add README
```

Игнорирование файлов

```
$ cat .gitignore
*.[oa]
*~
```



Фиксация изменений

```
$ git commit
```

```
$ git commit -m "Story 182: Fix benchmarks for speed"
```

Добавление удаленных репозиториев

`git remote add [сокращение] [url]`:

```
$ git remote
origin
$ git remote add pb git://github.com/paulboone/ticgit.git
$ git remote -v
origin  git://github.com/schacon/ticgit.gi
pb      git://github.com/paulboone/ticgit.g
```

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

Добавление удаленных репозиториев

Fetch - данная команда связывается с указанным удалённым проектом и забирает все те данные проекта, которых у вас ещё нет. После того как вы выполнили команду, у вас должны появиться ссылки на все ветки из этого удалённого проекта. Теперь эти ветки в любой момент могут быть просмотрены или слиты.

Pull - выполнение `git pull`, как правило, извлекает (fetch) данные с сервера, с которого вы изначально склонировали, и автоматически пытается слить (merge) их с кодом, над которым вы в данный момент работаете.

Push используется когда вы хотите поделиться своими наработками, вам необходимо отправить (push) их в главный репозиторий. Команда для этого действия простая:

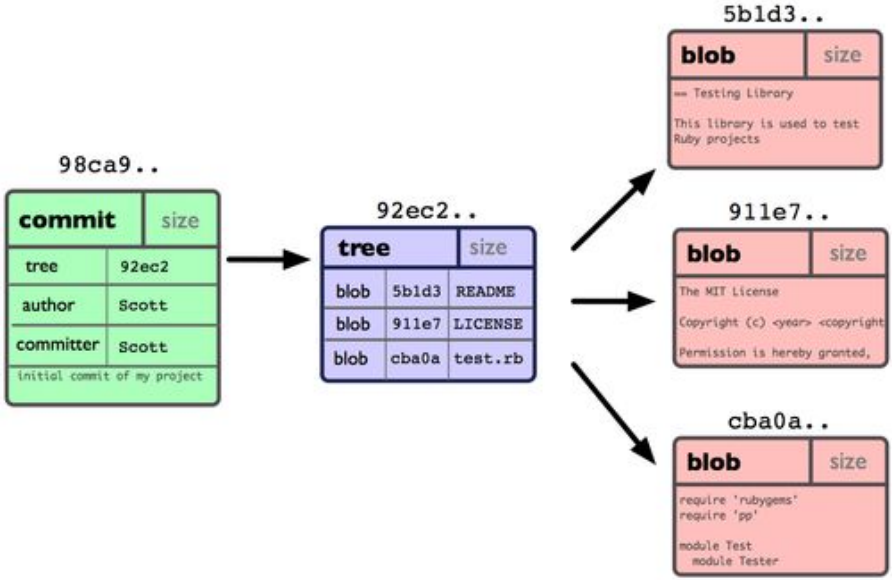
```
git push [удал. сервер] [ветка]
```

Ветвление в Git

```
$ git add README test.rb LICENSE
$ git commit -m 'initial commit of my project'
```

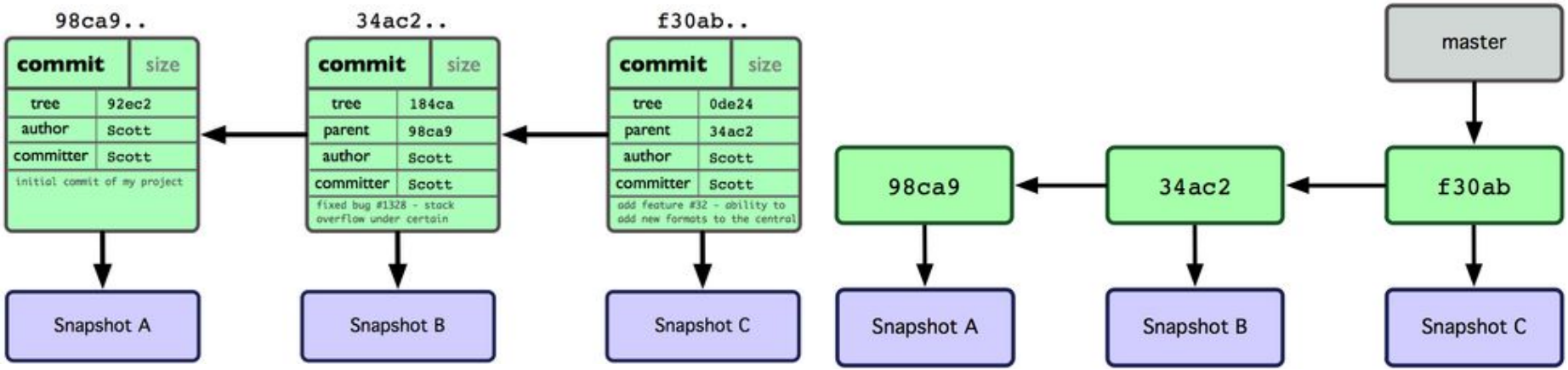
Git-репозиторий теперь содержит пять объектов:

- по одному блобу для содержимого каждого из трёх файлов,
- одно дерево, в котором перечислено содержимое каталога и определено соответствие имён файлов и блобов,
- один коммит с указателем на тот самый объект-дерево для корня и со всеми метаданными коммита.

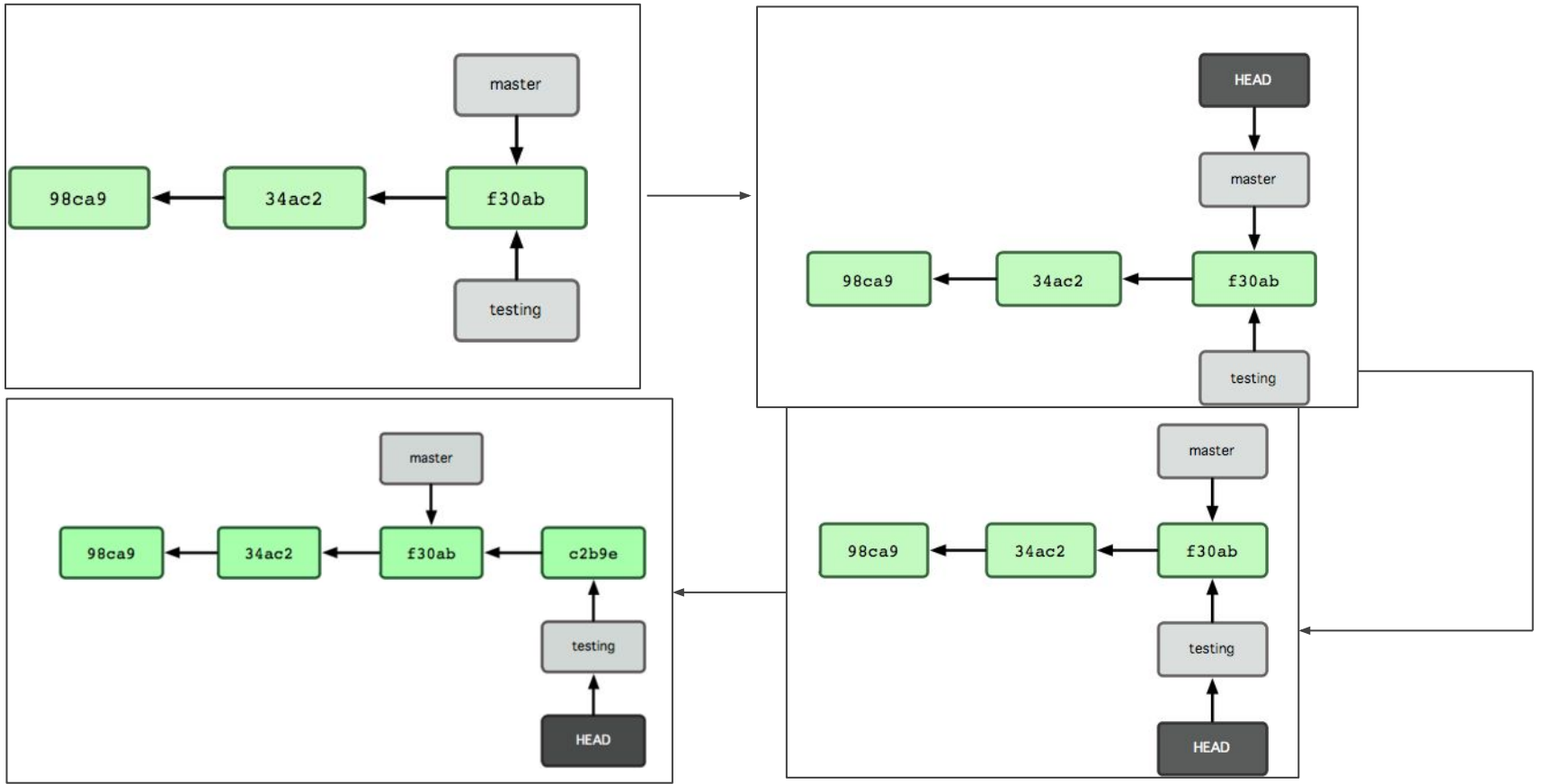


Ветвление в Git

Ветка в Git’е – это просто легковесный подвижный указатель на один из этих коммитов. Ветка по умолчанию в Git’е называется master. Когда вы создаёте коммиты на начальном этапе, вам дана ветка master, указывающая на последний сделанный коммит. При каждом новом коммите она сдвигается вперёд автоматически.



Ветвление в Git



Ветвление в Git