Система контроля версий Git

М. А. Ложников

25 февраля 2019 г.

Это предварительная невыверенная версия. Читайте на свой страх и риск.

Введение

Система контроля версий Git позволяет вести совместную удалённую разработку приложений (или иных текстовых проектов). Как правило для этого заводится git сервер, который хранит все данные. Один сервер может содержать несколько проектов, называемых репозиториями. Ключевое отличие репозитория состоит в том, что он хранит информацию обо всех изменениях когда-либо внесённых в код проекта с самого момента его основания. Таким образом, разработчикам доступна вся история развития проекта, разделённая на коммиты, где коммит это некоторый набор изменений, объединённых в группу, представленный в истории развития проекта неделимым куском. Работает это следующим образом:

1) сначала разработчик модифицирует файлы проекта;
2) затем, если он решил, что завершил некоторый логический кусок проекта (или просто оказался ленивым и ушёл после работы домой);
3) в этом случае он объединяет накопившиеся изменения в коммит;
4) при этом в истории проекта появляется запись о том, что таким-то таким-то тогда-то тогда-то были сделаны такие-то такие-то изменения;
5) после этого коммит можно залить на сервер для того, чтобы эти изменения могли выкачать другие разработчики.

Таким образом, коммит это некоторая контрольная точка в развитии проекта.

Стоит отметить, что Git позволяет оптимально хранить изменения только текстовых файлов, например, исходных кодов программ или текстов статей в формате LaTeX. Однако в случае бинарных файлов Git не может определить список их отличий, таким образом, при изменении бинарного файла Git будет хранить в репозитории старую и новую версии целиком. Репозиторий, в котором часто меняются бинарные файлы, будет быстро увеличиваться в размере. Следовательно, по возможности следует избегать хранения бинарных файлов в git репозиториях. Если этого избежать не получается, то следует удалять из репозитория прежнюю версию бинарного файла при добавлении новой версии. Отметим, что для этого придётся профильтровать все ранее сделанные коммиты и вычистить из них всю информацию об этом бинарном файле, в противном случае он удалится из текущей версии, но не удалится из истории изменений. Кроме того, если уж приходится добавлять бинарные файлы, лучше это делать в отдельных коммитах.

1 Основные команды Git

В этом разделе обсудим основные команды, необходимые для работы с Git. Все команды, приведённые ниже за исключением `git clone` должны запускаться из корневого каталога репозитория.

1.1 Клонирование репозитория

Доступ к репозиторию можно настроить по различным протоколам, например, по `https` или `ssh`. При доступе по `https` необходимо указывать логин и пароль. При доступе по `ssh` необходимо иметь приватный RSA ключ. Доступ на чтение по `https` бывает публичным.

1.1.1 Доступ по протоколу https

```
# Склонировать репозиторий (полностью выкачать репозиторий в текущую папку)
git clone <URL репозитория>

# Например, скачаем репозиторий ensmallen проекта mlpack с сервера github.com
# по протоколу https
git clone https://github.com/mlpack/ensmallen

# В текущем каталоге появится папка ensmallen, в которой будет содержаться
# полная выгрузка репозитория.
```

Если репозиторий приватный, то команда `git clone` спросит логин и пароль.

1.1.2 Доступ по протоколу ssh

Для генерации RSA ключа можно воспользоваться командой `ssh-keygen`, её можно запустить без аргументов. Команда спросит название файла, куда нужно сохранить ключ, пароль для разблокировки ключа (можно оставить пустой) и подтверждение пароля. Предположим, пользователь указал имя ключа `myrsakey`, тогда команда `ssh-keygen` создаст два файла: `myrsakey` — приватная часть ключа (должна быть известна только создавшему её разработчику) и `myrsakey.pub` — публичная часть ключа (должна быть прописана в настройках git сервера для того, чтобы предоставить доступ разработчику к репозиторию).

Для того, чтобы работать с репозиторием по протоколу `ssh` нужно добавить наш приватный RSA ключ в менеджер RSA ключей командой `ssh-add`. Делать это нужно каждый раз после перезагрузки компьютера.

```
ssh-add <Путь к приватному RSA ключу>

# Например, в Linux это выглядело бы так
ssh-add ~/.ssh/myrsakey
```

Команда `ssh-add` спросит пароль для разблокировки ключа, если вы его указали при создании этого ключа. Отметим, что в Linux требуется, чтобы все ключи лежали в файлах с правами доступа `600`.

```
# Создаём папку, в которую положим ключи (необязательно)
mkdir ~/.ssh                # Создаём папку
chmod 700 ~/.ssh            # Устанавливаем права доступа к папке

# Если папка ~/.ssh/ существует
cp myrsakey ~/.ssh/         # Копируем ключ
chmod 600 ~/.ssh/myrsakey   # Устанавливаем права доступа к ключу
```

Под Windows команда `ssh-add` может написать

```
Could not open a connection to your authentication agent.
```

В этом случае нужно запустить менеджер ключей

```
ssh-agent # Запускаем команду

# Она выведет на экран следующие строки (значения переменных будут другие)
SSH_AUTH_SOCK=/tmp/ssh-0sWrKeNNhYK6/agent.6780; export SSH_AUTH_SOCK;
SSH_AGENT_PID=6692; export SSH_AGENT_PID;
echo Agent pid 6692;

# Следует их скопировать и вызвать в терминале команды
export SSH_AUTH_SOCK=/tmp/ssh-0sWrKeNNhYK6/agent.6780
export SSH_AGENT_PID=6692
```

и повторить команду `ssh-add`.

После того, как ключ добавлен в менеджер ключей, можно склонировать репозиторий по протоколу `ssh`. Например,

```
git clone ssh://git@github.com:mlpack/ensmallen
```

Можно автоматизировать выставление переменных окружения следующим образом:

```
# Это действие нужно делать каждый раз после перезагрузки компьютера
ssh-agent > env.txt # Запускаем команду и сохраняем её вывод в файл env.txt

# Далее нужно обновить переменные окружения, записанные в файле env.txt
# Делать это нужно каждый раз при перезапуске терминала
. env.txt # Первый способ (написано точка пробел)

# Или чуть длиннее так:
source env.txt # Второй способ
```

Если файл `env.txt` находится в другой папке, то к нему нужно указывать полный или относительный путь.

Предложенный способ можно ещё улучшить, прописав запуск `ssh-agent` и выставление переменных окружения в файл `~/.bashrc` или `~/.profile` (подробности см. здесь `https://help.github.com/en/articles/working-with-ssh-key-passphrases`).

1.2 Ветви репозитория

В одном репозитории может быть несколько разных ветвей, в которых ведётся разработка. Каждая ветвь не зависит от другой, что позволяет вести параллельно разработку разных версий проекта. Ветви можно создавать из других ветвей, а также сливать изменения одной ветви в другую. Часто бывает полезным вести разработку в отдельной ветви и сливать изменения в основную ветвь как только эти изменения станут достаточно стабильными. По умолчанию основная ветвь проекта называется `master`. Для работы с ветвями полезно знать следующие команды:

```
# Посмотреть список всех ветвей проекта
git branch -a

# Создать ветвь с названием BRANCH_NAME из текущей ветви
git branch BRANCH_NAME

# Сменить текущую ветвь проекта на ветвь BRANCH_NAME. После вызова этой команды
# содержимое папки локального репозитория будет соответствовать ветви BRANCH_NAME
git checkout BRANCH_NAME

# Удалить ветвь BRANCH_NAME
git branch -D BRANCH_NAME
```

Отметим, что все эти команды работают только с локальным репозиторием. Они не заливают изменения на сервер.

1.3 Создание коммита

Для того, чтобы добавить изменения в список изменений, которые должны войти в коммит, используют команду `git add`. Команду следует вызывать в корневой папке репозитория. Для этого нужно передать в список аргументов этой команды пути к файлам, изменения в которых должны войти в коммит. Команду можно вызывать несколько раз, передавая в неё разные файлы, в таком случае, все эти файлы будут добавлены в список.

```
git add path/to/file1 path/to/file2 ... path/to/fileN

# Пусть в проекте есть файлы file1.txt и dir1/file2.txt
# то есть file2.txt лежит в папке dir1. Чтобы добавить эти файлы в формируемый
# коммит, нужно вызвать команду
git add file1.txt dir1/file2.txt

# Если нужно добавить все изменения в формирующийся коммит, то
# можно использовать команду
git add -A
```

Обратите внимание, что команда `git add -A` добавит все изменённые файлы в текущем каталоге и его подкаталогах. Среди этих файлов могут оказаться временные файлы, а также бинарные файлы, созданные при компиляции программы. Для того, чтобы команда `git add -A` игнорировала файлы, которые не нужно добавлять в коммит, необходимо создать в репозитории файл с названием `.gitignore` и перечислить в нём список всех файлов и каталогов, которые не нужно добавлять в репозиторий (по одному файлу в строке). Допускается использование символа `*` для обозначения произвольной (возможно, пустой) последовательности символов. Сам файл `.gitignore` нужно также “закоммитить”, то есть сделать его неотъемлемой частью проекта. Приведём в качестве примера возможное содержание файла `.gitignore`

```
data
build*
*.out
*~
```

Например, каталог `data` может использоваться для хранения временных файлов или результатов работы программы, в папке, начинающейся с `build` могут храниться исполняемые и объектные файлы, сгенерированные компилятором, файлы с символом `~` на конце могут создаваться текстовыми редакторами для восстановления текста после аварийного завершения работы редактора.

Для проекта в системе LaTeXможет оказаться полезным следующий шаблон файла `.gitignore`:

```
*.aux
*.log
*.nav
*.out
*.pdf
*.ps
*.dvi
*.snm
*.synctex.gz
*.toc
*.blg
*.bbl
*.bcf
*.xml
*-blx.bib
*~
```

После того, как вы добавили все изменения, нужно создать сам коммит. Делается это командой `git commit`. После того, как вы вызовете эту команду, откроется текстовый редактор, в котором нужно будет написать краткую информацию о коммите. После выхода из редактора с сохранением будет создан коммит. Выход без сохранения отменяет коммит. Отметим, что коммит будет создан в локальной копии репозитория. Чтобы его увидели другие разработчики, нужно залить изменения на сервер.

1.3.1 Работа с редактором vim

Поскольку редактором по-умолчанию сообщений в коммитах является `vim`, стоит сказать пару слов о том, как им пользоваться. Данный редактор работает в командной строке, у него нет графического интерфейса. В редакторе есть два режима работы: режим ввода текста и командный режим.

В режиме ввода текста можно только вводить текст, в командном режиме можно сохранять файл, выходить из редактора, копировать и вставлять текст, открывать новые вкладки, переключаться между ними и т. д. Для перехода в режим ввода нужно нажать клавишу `i` (причём в системе должна быть выбрана английская раскладка), для перехода в режим команд, нужно нажать клавишу `Esc`.

Для работы с `git commit` вам потребуется знать две команды редактора `vim`, доступные в командном режиме (должна быть активна английская раскладка):

1. `:x` —выход с сохранением;
2. `:q!` — выход без сохранения.

После ввода команды необходимо нажать клавишу `Enter`.

1.4 Работа с репозиторием на сервере

К локальному репозиторию можно добавить несколько удалённых (от слова *remote*) серверов (вернее удалённых репозиториев). Представьте, что вы создали свой проект, сделав *fork* другого проекта, то

есть просто скопировав его. Соответственно, есть сервер исходного проекта и сервер вашего проекта. Исходный проект периодически обновляется и вам требуется периодически синхронизировать кодовую базу вашего проекта с исходным. Для этих целей локальному репозиторию можно назначить несколько серверов. Работает это следующим образом: сначала выбирается ветвь с сервера исходного проекта, затем изменения из этой ветви вливаются в ваш проект, вы исправляете конфликты и другие ошибки, делаете коммит и заливаете изменения на свой сервер.

При клонировании проекта в настройки локального репозитория прописывается сервер по умолчанию, который называется `origin`. Его URL совпадает с тем, который вы ввели при клонировании проекта. Приведём список команд для работы с сервером

```
# Посмотреть список названий серверов (вернее удалённых репозиториев) и их URL
git remote -v

# Выкачать ветвь BRANCH_NAME из удалённого репозитория REMOTE_REPO_NAME
git pull REMOTE_REPO_NAME BRANCH_NAME

# Записать изменения локальной ветви BRANCH_NAME в удалённый репозиторий REMOTE_REPO_NAME
git push REMOTE_REPO_NAME BRANCH_NAME

# Удалить ветвь BRANCH_NAME из удалённого репозитория REMOTE_REPO_NAME
git push --delete REMOTE_REPO_NAME BRANCH_NAME
```

Таким образом, если вы работаете в ветви по-умолчанию на сервере по-умолчанию, то команды `pull` и `push` будут выглядеть следующим образом:

```
# Выкачать изменения с сервера
git pull origin master

# Записать изменения на сервер
git push origin master
```

Обратите внимание на то, что нельзя залить коммиты на сервер, не выкачав оттуда все коммиты. То есть для отправки данных на сервер локальная копия репозитория должна содержать все коммиты, присутствующие на сервере. Иными словами, сначала делается `pull` (если на сервере нет новых изменений, то не обязательно), а потом `push`. В противном случае `git` затрёт все коммиты на сервере, которых нет в локальной версии. Разумеется, он спросит подтверждение.

1.5 Разрешение конфликтов

Предположим, что вы решили слить две ветви вашего проекта, которые некоторое время развивались независимо, или, например, в то время, как вы модифицировали код, кто-либо другой поменял те же самые строчки, которые меняли вы и залил изменения на сервер. Вы сделали коммит, затем сделали `git pull`. Команда `git pull` увидела, что вы поменяли несколько одинаковых строк и выдала ошибку, поскольку она не знает, какой вариант из двух выбрать. Такая ситуация называется конфликтом.

Поясним на примере. Предположим, что Вася и Петя писали программу по определению среднего арифметического массива. Они написали файл `avg.cpp` и залили его на сервер.

```cpp
#include <vector>

double avg(const std::vector<double>& vec) {
  double result = 0;

  for (double item : vec)
    result += item;

  result /= vec.size();

  return result;
}
```

Затем они независимо друг от друга увидели, что забыли сделать проверку деления на нуль. Петя поменял строчку `result /= vec.size();` на следующую конструкцию

```cpp
if (vec.size() == 0)
  std::abort();
```

сделал коммит и залил его на сервер. В то же время, Вася написал другие изменения:

```cpp
if (vec.size() != 0)
  result /= vec.size();
```

Вася сделал коммит, сделал `git pull`. Напомним, что сделать `git push` без `git pull` Вася не мог, потому что на сервере уже лежал Петин коммит, которого не было в Васином локальном репозитории. Команда `git pull` сообщила Васе о конфликте:

```
Auto-merging avg.cpp
CONFLICT (content): Merge conflict in avg.cpp
Automatic merge failed; fix conflicts and then commit the result.
```

При этом в Васином локальном репозитории файл `avg.cpp` стал выглядеть так:

```cpp
#include <vector>

double avg(const std::vector<double>& vec) {
  double result = 0;

  for (double item : vec)
    result += item;

<<<<<<< HEAD
  if (vec.size() != 0)
    result /= vec.size();
=======
  if (vec.size() == 0)
```

```
    std::abort();

  result /= vec.size();
>>>>>>> 3db953b4863dd6beb375b0f0c949ba4564f32041

  return result;
}
```

То есть Git сделал в файле два возможных варианта: первый — вариант в Васином коммите и второй — вариант в Петином коммите. Для того, чтобы разрешить этот конфликт, Вася должен выбрать правильный вариант (может оказаться так, что нужно будет придумывать третий вариант, если код поменялся значительно), то есть исправить файл, сформировать коммит с необходимыми изменениями и залить его на сервер.

2 Дополнительные команды

2.1 Просмотр изменений

Посмотреть список всех коммитов можно с помощью команды `git log`. Эта команда выведет список коммитов в том числе уникальный идентификатор коммита и краткую информацию о нём.

Посмотреть текущие изменения в репозитории с момента последнего коммита можно с помощью команды `git diff`. Если передать в эту команду два идентификатора коммитов, полученные из команды `git log`, то она выведет список всех изменений от первого коммита до второго. Флаг `--color` сохраняет ASCII цвета при перенаправлении вывода в файл. Если добавить флаг `--color-words`, то команда выведет изменения не построчно, а с точностью до слова. Флаг `--name-only` позволяет вывести только имена изменённых файлов.

```
# Показать изменения с момента последнего коммита
git diff

# Показать изменения от коммита UUID1 до коммита UUID2
git diff UUID1 UUID2

# Показать изменения с точностью до слова
git diff --color-words

# Вывести в файл out.txt изменения вместе с цветами ASCII
git diff --color > out.txt

# Вывести только имена изменённых файлов
git diff --name-only
```

2.2 Возвращение репозитория к исходному состоянию

Чтобы вернуть все файлы, содержащиеся в репозитории к состоянию, актуальному на момент какого-либо коммита текущей ветви, используется команда `git reset`.

```
# Откатить все изменения к состоянию, актуальному на момент последнего коммита
git reset --hard HEAD
```

```
3
4 # Откатить все изменения к состоянию, актуальному на момент коммита UUID
5 git reset --hard UUID
6
7 # Откатить файл path/to/file к состоянию, актуальному на момент последнего коммита
8 git checkout path/to/file
```

2.3 Удаление ненужных файлов

Иногда из репозитория требуется полностью удалить файл вместе со всеми его изменениями. Сделать это обычным коммитом нельзя, поскольку останется версия этого файла в более ранних коммитах. Вместо этого можно использовать команду `git filter-branch`, которая вычистит упоминание об этом файле из каждого когда-либо созданного коммита. Полное удаление файла из репозитория может быть полезно, например, при удалении неактуальных картинок из статьи или при обновлении набора тестовых данных для разрабатываемого приложения, а также при изменении бинарных файлов. Синтаксис у этой команды следующий:

```
1 # Удалить файл или папку <file/dir>
2 git filter-branch --index-filter 'git rm -r --cached --ignore-unmatch <file/dir>' HEAD
```

Отметим, что данная команда изменит коммиты локального репозитория. Для того, чтобы записать эти изменения на сервер потребуется вызвать команду `git push` с флагом `--force`. Тогда данные отправятся на сервер. Отметим, что если на сервере были коммиты, которых не было в локальном репозитории, то они потеряются.

3 Прожиточный минимум

Приведём необходимый прожиточный минимум команд Git.

```
1 # Склонировать репозиторий, расположенный по ссылке URL
2 git clone URL
3
4 # Выкачать изменения с сервера
5 git pull origin master
6
7 # Добавить изменения файла в формируемый коммит
8 git add path/to/file
9
10 # Или если настроен файл .gitignore
11 git add -A
12
13 # Сделать коммит
14 git commit
15
16 # Отправить изменения на сервер
17 git push origin master
```

Список литературы

[1] Официальный сайт проекта Git `https://git-scm.com/`

[2] Хостинг для проектов `https://github.com/`