Лабораторная работа № 2. Управление версиями

2.1. Цель работы

Изучить идеологию и применение средств контроля версий.

2.2. Системы контроля версий. Общие понятия

Системы контроля версий (Version Control System, VCS) применяются при работе нескольких человек над одним проектом. Обычно основное дерево проекта хранится в локальном или удалённом репозитории, к которому настроен доступ для участников проекта. При внесении изменений в содержание проекта система контроля версий позволяет их фиксировать, совмещать изменения, произведённые разными участниками проекта, производить откат к любой более ранней версии проекта, если это требуется.

В классических системах контроля версий используется централизованная модель, предполагающая наличие единого репозитория для хранения файлов. Выполнение большинства функций по управлению версиями осуществляется специальным сервером. Участник проекта (пользователь) перед началом работы посредством определённых команд получает нужную ему версию файлов. После внесения изменений, пользователь размещает новую версию в хранилище. При этом предыдущие версии не удаляются из центрального хранилища и к ним можно вернуться в любой момент. Сервер может сохранять не полную версию изменённых файлов, а производить так называемую дельта-компрессию — сохранять только изменения между последовательными версиями, что позволяет уменьшить объём хранимых данных.

Системы контроля версий поддерживают возможность отслеживания и разрешения конфликтов, которые могут возникнуть при работе нескольких человек над одним файлом. Можно объединить (слить) изменения, сделанные разными участниками (автоматически или вручную), вручную выбрать нужную версию, отменить изменения вовсе или заблокировать файлы для изменения. В зависимости от настроек блокировка не позволяет другим пользователям получить рабочую копию или препятствует изменению рабочей копии файла средствами файловой системы ОС, обеспечивая таким образом, привилегированный доступ только одному пользователю, работающему с файлом.

Системы контроля версий также могут обеспечивать дополнительные, более гибкие функциональные возможности. Например, они могут поддерживать работу с несколькими версиями одного файла, сохраняя общую историю изменений до точки ветвления версий и собственные истории изменений каждой ветви. Кроме того, обычно доступна информация о том, кто из участников, когда и какие изменения вносил. Обычно такого рода информация хранится в журнале изменений, доступ к которому можно ограничить.

В отличие от классических, в распределённых системах контроля версий центральный репозиторий не является обязательным.

Среди классических VCS наиболее известны CVS, Subversion, а среди распределённых — Git, Bazaar, Mercurial. Принципы их работы схожи, отличаются они в основном синтаксисом используемых в работе команд.

2.3. Указания к лабораторной работе

Система контроля версий Git представляет собой набор программ командной строки. Доступ к ним можно получить из терминала посредством ввода команды `git` с различными опциями.

Благодаря тому, что Git является распределённой системой контроля версий, резервную копию локального хранилища можно сделать простым копированием или архивацией.

2.3.1. Основные команды git

Наиболее часто используемые команды `git`:

- создание основного дерева репозитория:
  ```
  git init
  ```
- получение обновлений (изменений) текущего дерева из центрального репозитория:
  ```
  git pull
  ```
- отправка всех произведённых изменений локального дерева в центральный репозиторий:
  ```
  git push
  ```
- просмотр списка изменённых файлов в текущей директории:
  ```
  git status
  ```
- просмотр текущих изменения:
  ```
  git diff
  ```
- сохранение текущих изменений:
  - добавить все изменённые и/или созданные файлы и/или каталоги:
    ```
    git add .
    ```
  - добавить конкретные изменённые и/или созданные файлы и/или каталоги:
    ```
    git add имена_файлов
    ```
  - удалить файл и/или каталог из индекса репозитория (при этом файл и/или каталог остаётся в локальной директории):
    ```
    git rm имена_файлов
    ```
- сохранение добавленных изменений:
  - сохранить все добавленные изменения и все изменённые файлы:
    ```
    git commit -am 'Описание коммита'
    ```
  - сохранить добавленные изменения с внесением комментария через встроенный редактор:
    ```
    git commit
    ```
- создание новой ветки, базирующейся на текущей:
  ```
  git checkout -b имя_ветки
  ```
- переключение на некоторую ветку:
  ```
  git checkout имя_ветки
  ```
  (при переключении на ветку, которой ещё нет в локальном репозитории, она будет создана и связана с удалённой)
- отправка изменений конкретной ветки в центральный репозиторий:
  ```
  git push origin имя_ветки
  ```
- слияние ветки с текущим деревом:
  ```
  git merge --no-ff имя_ветки
  ```
- удаление ветки:
  - удаление локальной уже слитой с основным деревом ветки:
    ```
    git branch -d имя_ветки
    ```
  - принудительное удаление локальной ветки:
    ```
    git branch -D имя_ветки
    ```
  - удаление ветки с центрального репозитория:
    ```
    git push origin :имя_ветки
    ```

2.3.2. Стандартные процедуры работы при наличии центрального репозитория

Работа пользователя со своей веткой начинается с проверки и получения изменений из центрального репозитория (при этом в локальное дерево до начала этой процедуры не должно было вноситься изменений):

```
git checkout master
git pull
git checkout -b имя_ветки
```

Затем можно вносить изменения в локальном дереве и/или ветке.

После завершения внесения какого-то изменения в файлы и/или каталоги проекта необходимо разместить их в центральном репозитории. Для этого необходимо проверить, какие файлы изменились к текущему моменту:

```
git status
```

и при необходимости удаляем лишние файлы, которые не хотим отправлять в центральный репозиторий.

Затем полезно просмотреть текст изменений на предмет соответствия правилам ведения чистых коммитов:

```
git diff
```

Если какие-либо файлы не должны попасть в коммит, то помечаем только те файлы, изменения которых нужно сохранить. Для этого используем команды добавления и/или удаления с нужными опциями:

```
git add ...
git rm ...
```

Если нужно сохранить все изменения в текущем каталоге, то используем:

```
git add .
```

Затем сохраняем изменения, поясняя, что было сделано:

```
git commit -am "Some commit message"
```

и отправляем в центральный репозиторий:

```
git push origin имя_ветки
```

или

```
git push
```

2.3.3. Работа с локальным репозиторием

Создадим локальный репозиторий.

Сначала сделаем предварительную конфигурацию, указав имя и email владельца репозитория:

```
git config --global user.name "Имя Фамилия"
git config --global  user.email "work@mail"
```

и настроив utf-8 в выводе сообщений `git`:

```
git config --global quotepath false
```

Для инициализации локального репозитория, расположенного, например, в каталоге `~/tutorial`, необходимо ввести в командной строке:

```
cd
mkdir tutorial
cd tutorial
git init
```

После это в каталоге `tutorial` появится каталог `.git`, в котором будет храниться история изменений.

Создадим тестовый текстовый файл `hello.txt` и добавим его в локальный репозиторий:

```
echo 'hello world' > hello.txt
git add hello.txt
git commit -am 'Новый файл'
```

Воспользуемся командой `status` для просмотра изменений в рабочем каталоге, сделанных с момента последней ревизии:

```
git status
```

Во время работы над проектом так или иначе могут создаваться файлы, которые не требуется добавлять в последствии в репозиторий. Например, временные файлы, создаваемые редакторами, или объектные файлы, создаваемые компиляторами. Можно прописать шаблоны игнорируемых при добавлении в репозиторий типов файлов в файл `.gitignore` с помощью сервисов. Для этого сначала нужно получить список имеющихся шаблонов:

```
curl -L -s https://www.gitignore.io/api/list
```

Затем скачать шаблон, например, для C и C++

```
curl -L -s https://www.gitignore.io/api/c >> .gitignore
curl -L -s https://www.gitignore.io/api/c++ >> .gitignore
```

2.3.4. Работа с сервером репозиториев

Для последующей идентификации пользователя на сервере репозиториев необходимо сгенерировать пару ключей (приватный и открытый):

```
ssh-keygen -C "Имя Фамилия <work@mail>"
```

Ключи сохраняться в каталоге `~/.ssh/`.

Существует несколько доступных серверов репозиториев с возможностью бесплатного размещения данных. Например, `https://github.com/`.

Для работы с ним необходимо сначала завести на сайте `https://github.com/` учётную запись. Затем необходимо загрузить сгенерённый нами ранее открытый ключ. Для этого зайти на сайт `https://github.com/` под своей учётной записью и перейти в меню GitHub setting. После этого выбрать в боковом меню GitHub setting » SSH-ключи и нажать кнопку Добавить ключ. Скопировав из локальной консоли ключ в буфер обмена

```
cat ~/.ssh/id_rsa.pub | xclip -sel clip
```

вставляем ключ в появившееся на сайте поле.

После этого можно создать на сайте репозиторий, выбрав в меню Репозитории » Создать репозиторий, дать ему название и сделать общедоступным (публичным).

Для загрузки репозитория из локального каталога на сервер выполняем следующие команды:

```
git remote add origin
  ssh://git@github.com/<username>/<reponame>.git
git push -u origin master
```

Далее на локальном компьютере можно выполнять стандартные процедуры для работы с `git` при наличии центрального репозитория.

2.4. Рабочий процесс Gitflow

Рабочий процесс *Gitflow Workflow*. Будем описывать его с использованием пакета `gitflow`.

2.4.1. Общая информация

- Gitflow Workflow опубликована и популяризована Винсентом Дриссеном из компании vie.
- Gitflow Workflow предполагает выстраивание строгой модели ветвления с учётом выпуска проекта.
- Данная модель отлично подходит для организации рабочего процесса на основе релизов.
- Работа по модели Gitflow включает создание отдельной ветки для исправлений ошибок в рабочей среде.
- Последовательность действий при работе по модели Gitflow:
  - Из ветки `master` создаётся ветка `develop`.
  - Из ветки `develop` создаётся ветка `release`.
  - Из ветки `develop` создаются ветки `feature`.
  - Когда работа над веткой `feature` завершена, она сливается с веткой `develop`.
  - Когда работа над веткой релиза `release` завершена, она сливается в ветки `develop` и `master`.
  - Если в `master` обнаружена проблема, из `master` создаётся ветка `hotfix`.
  - Когда работа над веткой исправления `hotfix` завершена, она сливается в ветки `develop` и `master`.

2.4.2. Установка программного обеспечения

- Для Windows используется пакетный менеджер Chocolatey. Git-flow входит в состав пакета git.
  ```
  choco install git
  ```
- Для MacOS используется пакетный менеджер Homebrew.
  ```
  brew install git-flow
  ```
- Linux
  - Gentoo
    ```
    emerge dev-vcs/git-flow
    ```
  - Ubuntu
    ```
    apt-get install git-flow
    ```

2.4.3. Процесс работы с Gitflow

2.4.3.1. Основные ветки (master) и ветки разработки (develop)

Для фиксации истории проекта в рамках этого процесса вместо одной ветки `master` используются две ветки. В ветке `master` хранится официальная история релиза, а ветка `develop` предназначена для объединения всех функций. Кроме того, для удобства рекомендуется присваивать всем коммитам в ветке `master` номер версии.

При использовании библиотеки расширений `git-flow` нужно инициализировать структуру в существующем репозитории:

```
git flow init
```

Для github параметр `Version tag prefix` следует установить в `v`.

После этого проверьте, на какой ветке Вы находитесь:

```
git branch
```

2.4.3.2. Функциональные ветки (feature)

Под каждую новую функцию должна быть отведена собственная ветка, которую можно отправлять в центральный репозиторий для создания резервной копии или совместной работы команды. Ветки `feature` создаются не на основе `master`, а на основе `develop`. Когда работа над функцией завершается, соответствующая ветка сливается обратно с веткой `develop`. Функции не следует отправлять напрямую в ветку `master`.

Как правило, ветки `feature` создаются на основе последней ветки `develop`.

Создание функциональной ветки Создадим новую функциональную ветку:

```
git flow feature start feature_branch
```

Далее работаем как обычно.

Окончание работы с функциональной веткой По завершении работы над функцией следует объединить ветку `feature_branch` с `develop`:

```
git flow feature finish feature_branch
```

2.4.3.3. Ветки выпуска (release)

Когда в ветке `develop` оказывается достаточно функций для выпуска, из ветки develop создаётся ветка `release`. Создание этой ветки запускает следующий цикл выпуска, и с этого момента новые функции добавить больше нельзя — допускается лишь отладка, создание документации и решение других задач. Когда подготовка релиза завершается, ветка `release` сливается с `master` и ей присваивается номер версии. После нужно выполнить слияние с веткой `develop`, в которой с момента создания ветки релиза могли возникнуть изменения.

Благодаря тому, что для подготовки выпусков используется специальная ветка, одна команда может дорабатывать текущий выпуск, в то время как другая команда продолжает работу над функциями для следующего.

Создать новую ветку release можно с помощью следующей команды:

```
git flow release start 1.0.0
```

Для завершения работы на ветке `release` используются следующие команды:

```
git flow release finish 1.0.0
```

2.4.3.4. Ветки исправления (hotfix)

Ветки поддержки или ветки `hotfix` используются для быстрого внесения исправлений в рабочие релизы. Они создаются от ветки `master`. Это единственная ветка, которая должна быть создана непосредственно от master. Как только исправление завершено, ветку следует объединить с `master` и `develop`. Ветка `master` должна быть помечена обновленным номером версии.

Наличие специальной ветки для исправления ошибок позволяет команде решать проблемы, не прерывая остальную часть рабочего процесса и не ожидая следующего цикла релиза.

Ветку `hotfix` можно создать с помощью следующих команд:

```
git flow hotfix start hotfix_branch
```

По завершении работы ветка `hotfix` объединяется с `master` и `develop`:

```
git flow hotfix finish hotfix_branch
```

2.5. Последовательность выполнения работы

2.5.1. Настройка git

1. Создайте учётную запись на `https://github.com`.
2. Настройте систему контроля версий `git`, как это описано выше с использованием сервера репозиториев `https://github.com/`.
3. Создайте структуру каталога лабораторных работ согласно пункту М.2.

2.5.2. Подключение репозитория к github

- Создайте репозиторий на GitHub. Для примера назовём его `os-intro`.
- Рабочий каталог будем обозначать как `laboratory`. Вначале нужно перейти в этот каталог:
  ```
  cd laboratory
  ```
- Инициализируем системы git:
  ```
  git init
  ```
- Создаём заготовку для файла `README.md`:
  ```
  echo "# Лабораторные работы" >> README.md
  git add README.md
  ```
- Делаем первый коммит и выкладываем на github:
  ```
  git commit -m "first commit"
  git remote add origin
   ↪  git@github.com:<username>/sciproc-intro.git
  git push -u origin master
  ```

2.5.3. Первичная конфигурация

- Добавим файл лицензии:
  ```
  wget https://creativecommons.org/licenses/by/4.0/legalcode.txt
   ↪  -O LICENSE
  ```
- Добавим шаблон игнорируемых файлов. Просмотрим список имеющихся шаблонов:
  ```
  curl -L -s https://www.gitignore.io/api/list
  ```
  Затем скачаем шаблон, например, для C:
  ```
  curl -L -s https://www.gitignore.io/api/c >> .gitignore
  ```
  Можно это же сделать через web-интерфейс на сайте `https://www.gitignore.io/`.
- Добавим новые файлы:
  ```
  git add .
  ```
- Выполним коммит:
  ```
  git commit -a
  ```
- Отправим на github:
  ```
  git push
  ```

2.5.4. Конфигурация git-flow

- Инициализируем git-flow
  ```
  git flow init
  ```
  Префикс для ярлыков установим в `v`.
- Проверьте, что Вы на ветке `develop`:
  ```
  git branch
  ```
- Создадим релиз с версией 1.0.0

```
  git flow release start 1.0.0
```
- Запишем версию:
```
  echo "1.0.0" >> VERSION
```
- Добавим в индекс:
```
  git add .
  git commit -am 'chore(main): add version'
```
- Зальём релизную ветку в основную ветку
```
  git flow release finish 1.0.0
```
- Отправим данные на github
```
  git push --all
  git push --tags
```
- Создадим релиз на github.

2.6. Содержание отчёта

1. Титульный лист с указанием номера лабораторной работы и ФИО студента.
2. Формулировка цели работы.
3. Описание результатов выполнения задания:
   - скриншоты (снимки экрана), фиксирующие выполнение лабораторной работы;
   - листинги (исходный код) программ (если они есть);
   - результаты выполнения программ (текст или снимок экрана в зависимости от задания).
4. Выводы, согласованные с целью работы.
5. Ответы на контрольные вопросы.

2.7. Контрольные вопросы

1. Что такое системы контроля версий (VCS) и для решения каких задач они предназначаются?
2. Объясните следующие понятия VCS и их отношения: хранилище, commit, история, рабочая копия.
3. Что представляют собой и чем отличаются централизованные и децентрализованные VCS? Приведите примеры VCS каждого вида.
4. Опишите действия с VCS при единоличной работе с хранилищем.
5. Опишите порядок работы с общим хранилищем VCS.
6. Каковы основные задачи, решаемые инструментальным средством `git`?
7. Назовите и дайте краткую характеристику командам `git`.
8. Приведите примеры использования при работе с локальным и удалённым репозиториями.
9. Что такое и зачем могут быть нужны ветви (branches)?
10. Как и зачем можно игнорировать некоторые файлы при commit?