Русский (ru)

11
1032248286 Мохамед Муса Абдальрахман Закр

Архитектура компьютеров и операционные системы. Раздел "Операционные системы" (02.03.00, УГСН)
Участники
Отчет по задачам
Оценки
Общее
Методические заметки
Общая информация о курсе (02.03.01)
Общая информация о курсе (02.03.02)
Информационные материалы
Индивидуальный проект Персональный сайт научного работника
Лабораторная работа № 1
Лабораторная работа № 2
Лабораторная работа № 3
Лабораторная работа № 4
Лабораторная работа № 5
Лабораторная работа № 6
Лабораторная работа № 7
Лабораторная работа № 8
Лабораторная работа № 9
Лабораторная работа № 10
Лабораторная работа № 11
Лабораторная работа № 12
Лабораторная работа № 13
Лабораторная работа № 14
Внешние курсы
Лекция № 1
Лекция № 2
Лекция № 3
Лекция № 4
Лекция № 5
Лекция № 6
Лекция № 7
Лекция № 8
Итоговый контроль знаний

Домашняя страница
Личный кабинет
Календарь
Расписание занятий
Личные файлы
Помощь
Мои курсы
Архитектура компьютеров и операционные системы. Раздел "Архитектура

```
компьютеров" (02.03.00, УГСН)
Архитектура компьютеров и операционные системы. Раздел "Операционные
системы" (02.03.00, УГСН)
Второй иностранный язык (02.03.00, УГСН, НКАбд)
Дискретная математика и математическая логика. Раздел "Математическая
логика" (02.03.00)
Дифференциальные уравнения (02.03.00, УГСН, НКАбд)
История религий России (НКАбд)
Компьютерная алгебра (02.03.00)
Математический анализ (02.03.00, УГСН, НКА)
Основы машинного обучения и нейронные сети (02.03.00, УГСН)
Основы военной подготовки. Безопасность жизнедеятельности 24/25

Домашняя страница
Мои курсы
Факультет физико-математических и естественных наук
02.00.00 Компьютерные и информационные науки
УГСН 02.03.00 Компьютерные и информационные науки (бакалавриат, 1-2
курсы)
Архитектура компьютеров и операционные системы. Раздел "Операционные
системы" (02.03.00, УГСН)
Лабораторная работа № 4
Лабораторная работа № 4
```

Архитектура компьютеров и операционные системы. Раздел "Операционные системы" (02.03.00, УГСН)

Лабораторная работа № 4

```
Цель работы
Теоретические сведения
    Рабочий процесс Gitflow
        Общая информация
        Процесс работы с Gitflow
    Семантическое версионирование
        Краткое описание семантического версионирования
        Программное обеспечение
    Общепринятые коммиты
        Описание
Задание
Последовательность выполнения работы
    Установка программного обеспечения
        Установка git-flow
        Установка Node.js
        Настройка Node.js
        Общепринятые коммиты
    Практический сценарий использования git
        Создание репозитория git
        Работа с репозиторием git
```

Лабораторная работа Продвинутое использование git.

Цель работы

```
Получение навыков правильной работы с репозиториями git.
```

Теоретические сведения

Рабочий процесс Gitflow

```
Рабочий процесс Gitflow Workflow. Будем описывать его с использованием
пакета git-flow.
```

Общая информация

```
Gitflow Workflow опубликована и популяризована Винсентом Дриссеном.
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.
```

Процесс работы с Gitflow

```
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
```

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
```

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

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 обновлённым 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
```

Семантическое версионирование

Семантический подход в версионированию программного обеспечения.

Краткое описание семантического версионирования

```
Семантическое версионирование описывается в манифесте семантического
версионирования.

Кратко его можно описать следующим образом:
    Версия задаётся в виде кортежа МАЖОРНАЯ_ВЕРСИЯ.МИНОРНАЯ_ВЕРСИЯ.ПАТЧ.
    Номер версии следует увеличивать:
        МАЖОРНУЮ версию, когда сделаны обратно несовместимые изменения
API.
        МИНОРНУЮ версию, когда вы добавляете новую функциональность, не
нарушая обратной совместимости.
        ПАТЧ-версию, когда вы делаете обратно совместимые исправления.
    Дополнительные обозначения для предрелизных и билд-метаданных
возможны как дополнения к МАЖОРНАЯ.МИНОРНАЯ.ПАТЧ формату.
```

Программное обеспечение

```
Для реализации семантического версионирования создано несколько
программных продуктов.
При этом лучше всего использовать комплексные продукты, которые
используют информацию из коммитов системы версионирования.
Коммиты должны иметь стандартизованный вид.
```

```
В семантическое версионирование применяется вместе с общепринятыми
коммитами.

Пакет Conventional Changelog
    Пакет Conventional Changelog является комплексным решением по
управлению коммитами и генерации журнала изменений.
    Содержит набор утилит, которые можно использовать по-отдельности.
```

Общепринятые коммиты

Использование спецификации Conventional Commits.

Описание

Спецификация Conventional Commits:

```
Соглашение о том, как нужно писать сообщения commit'ов.
Совместимо с SemVer. Даже вернее сказать, сильно связано с семантическим
версионированием.
Регламентирует структуру и основные типы коммитов.

Структура коммита

<type>(<scope>): <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>

Или, по-русски:

<тип>(<область>): <описание изменения>
<пустая линия>
[необязательное тело]
<пустая линия>
[необязательный нижний колонтитул]

    Заголовок является обязательным.
    Любая строка сообщения о фиксации не может быть длиннее 100
символов.
    Тема (subject) содержит краткое описание изменения.
        Используйте повелительное наклонение в настоящем времени:
«изменить» ("change" not "changed" nor "changes").
        Не используйте заглавную первую букву.
        Не ставьте точку в конце.
    Тело (body) должно включать мотивацию к изменению и
противопоставлять это предыдущему поведению.
        Как и в теме, используйте повелительное наклонение в настоящем
времени.
    Нижний колонтитул (footer) должен содержать любую информацию о
критических изменениях.
```

Следует использовать для указания внешних ссылок, контекста коммита или другой мета информации.

Также содержит ссылку на issue (например, на github), который закрывает эта фиксация.

Критические изменения должны начинаться со слова BREAKING CHANGE: с пробела или двух символов новой строки. Затем для этого используется остальная часть сообщения фиксации.

Типы коммитов

Базовые типы коммитов

fix: — коммит типа fix исправляет ошибку (bug) в вашем коде (он соответствует PATCH в SemVer).

feat: — коммит типа feat добавляет новую функцию (feature) в ваш код (он соответствует MINOR в SemVer).

BREAKING CHANGE: — коммит, который содержит текст BREAKING CHANGE: в начале своего не обязательного тела сообщения (body) или в подвале (footer), добавляет изменения, нарушающие обратную совместимость вашего API (он соответствует MAJOR в SemVer). BREAKING CHANGE может быть частью коммита любого типа.

revert: — если фиксация отменяет предыдущую фиксацию. Начинается с revert:, за которым следует заголовок отменённой фиксации. В теле должно быть написано: Это отменяет фиксацию <hash> (это SHA-хэш отменяемой фиксации).

Другое: коммиты с типами, которые отличаются от fix: и feat:, также разрешены. Например, @commitlint/config-conventional (основанный на The Angular convention) рекомендует: chore:, docs:, style:, refactor:, perf:, test:, и другие.

Соглашения The Angular convention

Одно из популярных соглашений о поддержке исходных кодов — конвенция Angular (The Angular convention).

Типы коммитов The Angular convention

PROFI

Конвенция Angular (The Angular convention) требует следующие типы коммитов:

build: — изменения, влияющие на систему сборки или внешние зависимости (примеры областей (scope): gulp, broccoli, npm).

ci: — изменения в файлах конфигурации и скриптах CI (примеры областей: Travis, Circle, BrowserStack, SauceLabs).

docs: — изменения только в документации.

feat: — новая функция.

fix: — исправление ошибок.

perf: — изменение кода, улучшающее производительность.

refactor: — Изменение кода, которое не исправляет ошибку и не добавляет функции (рефакторинг кода).

style: — изменения, не влияющие на смысл кода (пробелы, форматирование, отсутствие точек с запятой и т. д.).

test: — добавление недостающих тестов или исправление существующих тестов.

Области действия (scope)

```
        Областью действия должно быть имя затронутого пакета npm (как
его воспринимает человек, читающий журнал изменений, созданный из
сообщений фиксации).

        Есть несколько исключений из правила «использовать имя пакета»:
            packaging — используется для изменений, которые изменяют
структуру пакета, например, изменения общедоступного пути.
            changelog — используется для обновления примечаний к выпуску
в CHANGELOG.md.
            отсутствует область действия — полезно для изменений стиля,
тестирования и рефакторинга, которые выполняются во всех пакетах
(например, style: добавить отсутствующие точки с запятой).

    Соглашения @commitlint/config-conventional

    Соглашение @commitlint/config-conventional входит в пакет
Conventional Changelog. В целом в этом соглашении придерживаются
соглашения Angular.
```

Задание

```
Выполнить работу для тестового репозитория.
Преобразовать рабочий репозиторий в репозиторий с git-flow и
conventional commits.
```

Последовательность выполнения работы

Установка программного обеспечения

Установка git-flow

PROFI

```
Linux

    Fedora

        Установка из коллекции репозиториев Copr
(https://copr.fedorainfracloud.org/coprs/elegos/gitflow/):

        # Enable the copr repository
        dnf copr enable elegos/gitflow
        # Install gitflow
        dnf install gitflow
```

Установка Node.js

На Node.js базируется программное обеспечение для семантического версионирования и общепринятых коммитов.

```
Fedora

dnf install nodejs
dnf install pnpm
```

Настройка Node.js

Для работы с Node.js добавим каталог с исполняемыми файлами, устанавливаемыми yarn, в переменную PATH.

```
Запустите:

pnpm setup

Перелогиньтесь, или выполните:

source ~/.bashrc
```

Общепринятые коммиты

```
commitizen

    Данная программа используется для помощи в форматировании коммитов.

    pnpm add -g commitizen

    При этом устанавливается скрипт git-cz, который мы и будем
использовать для коммитов.

standard-changelog

    Данная программа используется для помощи в создании логов.

    pnpm add -g standard-changelog
```


Практический сценарий использования git

Создание репозитория git

```
Подключение репозитория к github

    Создайте репозиторий на GitHub. Для примера назовём его git-
```

extended.

    Делаем первый коммит и выкладываем на github:

    git commit -m "first commit"
    git remote add origin git@github.com:<username>/git-extended.git
    git push -u origin master

Конфигурация общепринятых коммитов

    Конфигурация для пакетов Node.js

    pnpm init

    Необходимо заполнить несколько параметров пакета.
        Название пакета.
        Лицензия пакета. Список лицензий для npm:
https://spdx.org/licenses/. Предлагается выбирать лицензию CC-BY-4.0.

    Сконфигурим формат коммитов. Для этого добавим в файл package.json
команду для формирования коммитов:

    "config": {
            "commitizen": {
                "path": "cz-conventional-changelog"
            }
    }

    Таким образом, файл package.json приобретает вид:

    {
        "name": "git-extended",
        "version": "1.0.0",
        "description": "Git repo for educational purposes",
        "main": "index.js",
        "repository": "git@github.com:username/git-extended.git",
        "author": "Name Surname <username@gmail.com>",
        "license": "CC-BY-4.0",
        "config": {
            "commitizen": {
                "path": "cz-conventional-changelog"
            }
        }
    }

    Добавим новые файлы:

    git add .

    Выполним коммит:

    git cz

```
    Отправим на github:

    git push

Конфигурация git-flow

    Инициализируем git-flow

    git flow init

    Префикс для ярлыков установим в v.

    Проверьте, что Вы на ветке develop:

    git branch

    Загрузите весь репозиторий в хранилище:

    git push --all

    Установите внешнюю ветку как вышестоящую для этой ветки:

    git branch --set-upstream-to=origin/develop develop

    Создадим релиз с версией 1.0.0

    git flow release start 1.0.0

    Создадим журнал изменений

    standard-changelog --first-release

    Добавим журнал изменений в индекс

    git add CHANGELOG.md
    git commit -am 'chore(site): add changelog'

    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. Для этого будем использовать утилиты
работы с github:

    gh release create v1.0.0 -F CHANGELOG.md
```

```
Разработка новой функциональности

    Создадим ветку для новой функциональности:

    git flow feature start feature_branch

    Далее, продолжаем работу с git как обычно.

    По окончании разработки новой функциональности следующим шагом
следует объединить ветку feature_branch с develop:

    git flow feature finish feature_branch

Создание релиза git-flow

    Создадим релиз с версией 1.2.3:

    git flow release start 1.2.3

    Обновите номер версии в файле package.json. Установите её в 1.2.3.

    Создадим журнал изменений

    standard-changelog

    Добавим журнал изменений в индекс

    git add CHANGELOG.md
    git commit -am 'chore(site): update changelog'

    Зальём релизную ветку в основную ветку

    git flow release finish 1.2.3

    Отправим данные на github

    git push --all
    git push --tags

    Создадим релиз на github с комментарием из журнала изменений:

    gh release create v1.2.3 -F CHANGELOG.md
```

PROFI