

Лекция №9

«Системы контроля версий»

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

(пример: выложили версию программы, начали работу над новыми «фичами» и вдруг обнаружили ошибки. Нужно не потеряв новых наработок вернуться к рабочей версии, исправить ошибки).

ЧТО ТАКОЕ КОНТРОЛЬ ВЕРСИЙ, И ЗАЧЕМ ОН НУЖЕН?

- избыточность (дублируется весь код, а не только изменения)
- нет механизмов для распределения работы между несколькими разработчиками
- нет данных о том что именно изменилось (обычно пишут history файл с общей информацией об изменениях)

ЧТО ТАКОЕ КОНТРОЛЬ ВЕРСИЙ, И ЗАЧЕМ ОН НУЖЕН?

Локальные системы контроля версий

Для решения части из этих проблем были разработаны **локальные СКВ**.

Одной из **первых** и наиболее популярных **СКВ** такого типа являлась **RCS (Revision Control System)**,

ЦЕНТРАЛИЗОВАННЫЕ СИСТЕМЫ КОНТРОЛЯ ВЕРСИЙ

Основной проблемой оказалась необходимость сотрудничать с разработчиками за **другими компьютерами**. Чтобы решить её, были созданы **централизованные системы контроля версий (ЦСКВ)**.

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.

ЦЕНТРАЛИЗОВАННЫЕ СИСТЕМЫ КОНТРОЛЯ ВЕРСИЙ

ТАКОЙ ПОДХОД ИМЕЕТ МНОЖЕСТВО ПРЕИМУЩЕСТВ

ЦЕНТРАЛИЗОВАННЫЕ СИСТЕМЫ КОНТРОЛЯ ВЕРСИЙ

**ОДНАКО ПРИ ТАКОМ ПОДХОДЕ ЕСТЬ И НЕСКОЛЬКО СЕРЬЁЗНЫХ
НЕДОСТАТКОВ.**

РАСПРЕДЕЛЁННЫЕ СИСТЕМЫ КОНТРОЛЯ ВЕРСИЙ

В таких системах как **Git**, **Mercurial**, **Bazaar** или **Darcs** клиенты не просто выгружают последние версии файлов, а полностью копируют весь репозиторий.

Кроме того, в большей части этих систем можно работать с несколькими удалёнными репозиториями.

Основы GIT.

Правило 1. Почти все операции — локальные

Для совершения **большинства операций в Git'e** необходимы только **локальные файлы** и ресурсы, т.е. обычно информация с других компьютеров в сети не нужна.

Основы GIT.

Правило 2. Git следит за целостностью данных

Перед сохранением любого файла **Git** вычисляет **контрольную сумму**, и она становится **индексом** этого **файла**. Поэтому невозможно изменить содержимое файла или каталога так, чтобы Git не узнал об этом.

Основы GIT.

Правило 3. Чаще всего данные в Git только добавляются

Практически все действия, которые вы совершаете в **Git'e**, **ТОЛЬКО** добавляют данные в базу. Очень сложно заставить систему удалить данные или сделать что-то неотменяемое.

Основы GIT.

Правило 3. Три состояния файлов

В Git'e файлы могут находиться в одном из трёх состояний: зафиксированном, изменённом и подготовленном.

Основы GIT.

Правило 3. Три состояния файлов

"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

Основы GIT.

Правило 3. Три состояния файлов

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

ОСНОВЫ GIT.

Установка GIT (Windows)

Установить Git в Windows очень просто. Просто скачайте дистрибутив и запустите его:

- [Git для Windows](#)
- [Git для Mac \(но на маках он вроде уже должен быть\)](#)

После установки у вас будет как консольная версия (включающая SSH-клиент, который пригодится позднее), так и стандартная графическая.

Основы GIT.

Выбор сервера для центрального репозитория

- собственный сервер в локальной сети, обычно используется в крупных компаниях или из-за режима секретности (например, gitlab)
- сервер в Сети, например github. На таких серверах обычно есть аккаунты для публичных репозиториев, и возможность создания приватных репозиториев

Основы GIT.

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

Основы GIT.

Создание Git-репозитория

Здесь я бы мог расписать про то как создавать, свои каталоги с помощью консольных команд и не только, но на данный момент я не знаю, пригодится ли вам это, потому как все современные IDE предлагают удобный веб-интерфейс для работы с тем же GITом, конечно для ОБЩИХ знаний это информация была бы не лишней, но обдумав, решил, что пока не надо заострять внимание на этом. А рассмотрим только «БАЗУ» и основные определения.

Основы GIT.

Запись изменений в репозиторий

Вам нужно делать **НЕКОТОРЫЕ ИЗМЕНЕНИЯ** и фиксировать “снимки” состояния (snapshots) этих изменений в вашем репозитории **КАЖДЫЙ РАЗ**, когда проект достигает состояния, которое вам хотелось бы сохранить (обычно рекомендуют фиксировать **КАЖДОЕ АТОМАРНОЕ ИЗМЕНЕНИЕ**, т.е. **ФУНКЦИЮ, КЛАСС ИЛИ ЗАКОНЧЕННЫЙ АЛГОРИМ**).

Основы GIT.

Запись изменений в репозиторий

Каждый файл в вашем рабочем каталоге может находиться в одном из **двух состояний**: под **версионным контролем** (отслеживаемые) и **нет** (неотслеживаемые).

Основы GIT.

Запись изменений в репозиторий

Каждый файл в вашем рабочем каталоге может находиться в одном из **двух состояний**: под **версионным контролем** (отслеживаемые) и **нет** (неотслеживаемые).

Основы GIT.

Запись изменений в репозиторий

Отслеживаемые файлы — это те файлы, которые были в последнем слепке(сборке) состояния проекта; они могут быть неизменёнными, изменёнными или подготовленными к коммиту.

Основы GIT.

Запись изменений в репозиторий

Неотслеживаемые файлы — это всё остальное, любые файлы в вашем рабочем каталоге, которые не входили в ваш последний слепок состояния и не подготовлены к коммиту.

Основы GIT.

Определение состояния файлов

Основной инструмент, используемый для определения,
какие файлы в каком состоянии находятся
— это команда **git status**

Основы GIT.

Игнорирование файлов

Зачастую, имеется группа файлов, которые вы не только не хотите автоматически добавлять в репозиторий.

Вы можете создать в корне проекта файл .gitignore с перечислением шаблонов соответствующих таким файлам

```
*.log  
*.~*
```

Основы GIT.

Игнорирование файлов

- Пустые строки, а также строки, начинающиеся с #, игнорируются.
- Можно использовать стандартные glob шаблоны.
- Можно заканчивать шаблон символом слэша (/) для указания каталога.
- Можно инвертировать шаблон, используя восклицательный знак (!) в качестве первого символа.

Основы GIT.

Игнорирование файлов

Glob-шаблоны представляют собой упрощённые регулярные выражения используемые командными интерпретаторами.

Символ * соответствует 0 или более символам

Основы GIT.

Игнорирование файлов

```
# комментарий — эта строка игнорируется
# не обрабатывать файлы, имя которых заканчивается на .a
*.a
# НО отслеживать файл lib.a, несмотря на то, что мы игнорируем все .a файлы с помощью
предыдущего правила
!lib.a
# игнорировать только файл TODO находящийся в корневом каталоге, не относится к файлам вида
subdir/TODO
/TODO
# игнорировать все файлы в каталоге build/
build/
# игнорировать doc/notes.txt, но не doc/server/arch.txt
doc/*.txt
# игнорировать все .txt файлы в каталоге doc/
doc/**/*.txt
```

Основы GIT.

Фиксация изменений

Запомните, что коммит сохраняет снимок состояния вашего индекса.

Каждый раз, когда вы делаете коммит, вы сохраняете снимок состояния вашего проекта, который позже вы можете восстановить или с которым можно сравнить текущее состояние.

Основы GIT.

Работа с удалёнными репозиториями

Совместная работа включает в себя управление удалёнными репозиториями и **помещение (push)** и **получение (pull)** данных в и из них тогда, когда нужно обмениваться результатами работы.

Основы GIT.

Работа с удалёнными репозиториями

Совместная работа включает в себя управление удалёнными репозиториями и **помещение (push)** и **получение (pull)** данных в и из них тогда, когда нужно обмениваться результатами работы.

Основы GIT.

Работа с удалёнными репозиториями

Если вы хотите слить новые данные с вашими, то вы можете использовать команду `git pull`.

Она автоматически извлекает и затем сливает данные из удалённой ветки в вашу текущую ветку.

Основы GIT.

Работа с удалёнными репозиториями

Когда вы хотите поделиться своими наработками, вам необходимо отправить (**push**) их в главный репозиторий.

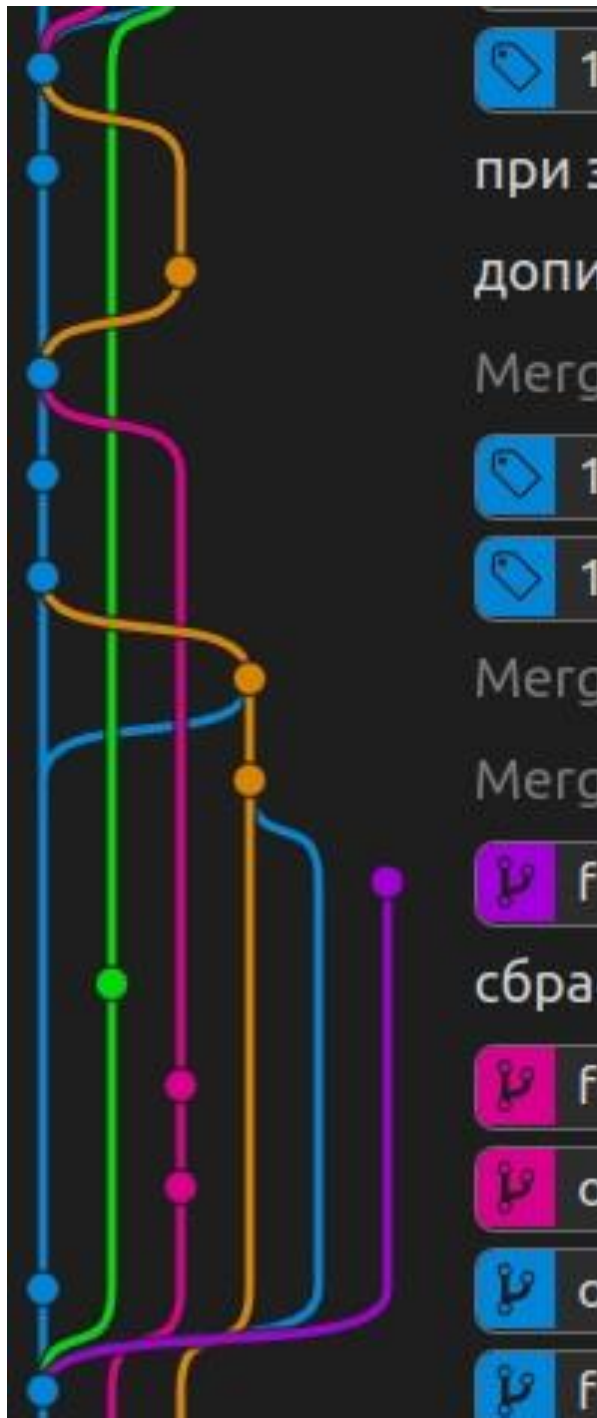
Основы GIT.

Ветвление в Git

Ветвление означает, что вы отклоняетесь от основной линии разработки и продолжаете работу, не вмешиваясь в основную линию.

Основы GIT.

Ветвление в Git



Основы GIT.

Завершение работы с веткой

Когда работа над новой фичей (в ветке) закончена, необходимо перенести изменения в основную ветку. Для этого используется команда **git merge**.

Вы сначала переключаетесь на ту ветку, в которую хотите слить изменения, а затем объединяете их

Основы GIT.

Формат Markdown

В корне репозитория принято размещать файл **readme.md**, в котором описывается что это за репозиторий, для чего он нужен, инструкции по установке, если это необходимо.

Основы GIT.

Формат Markdown

Markdown — облегчённый язык разметки, созданный с целью обозначения форматирования в простом тексте, с максимальным сохранением его читаемости человеком, и пригодный для машинного преобразования.

Основы GIT.

Формат Markdown

Текст с выделением

выделение (например, *курсив*)

****сильное выделение**** (например, **полужирное** начертание)

****комбинация**** курсива и полужирного начертания

| Причем без разницы в каком порядке использовать символы `*` и `_`

~~зачёркнутый~~ текст

Основы GIT.

Формат Markdown

Программный код

Выделяется знаком апострофа "`", может быть как `внутри строки`

```
```txt
```

```
так
```

```
и
```

```
отдельным
```

```
блоком
```



# Основы GIT.

## Формат Markdown

### Заголовки

Создание заголовков производится путём помещения знака решетки перед текстом заголовка. Количество знаков «#» соответствует уровню заголовка. Поддерживается 6 уровней заголовков.

## # Заголовок первого уровня

...

### ### Заголовок третьего уровня

...

##### Заголовок шестого уровня

# Основы GIT.

## Формат Markdown

### Цитаты (комментарии)

Любой элемент разметки (текст, список, картинка) могут быть помечены как цитата, если добавить в начале строки знак ">"

# Основы GIT.


## Формат Markdown

### Ссылки

- Ссылки на внешние ресурсы `[Текст ссылки](http://example.com/`  
`"Необязательный заголовок ссылки")`
- Ссылки внутри документа `[Текст ссылки](#якорь)` В качестве "якоря" может выступать заголовок или html-тег `<a>` (но это уже нетривиальное использование разметки)
- Ссылки внутри репозитория `[Текст ссылки](относительный путь на файл внутри репозитория)`

**ПОВТОРЕНИЕ**


# ОПЕРАТОРЫ ОТНОШЕНИЯ

Оператор	Значение
==	
!=	
>	
<	
>=	
<=	

# ОПЕРАТОРЫ ОТНОШЕНИЯ

Оператор	Значение
==	Равно
!=	Не равно
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно

# ЛОГИЧЕСКЕ ОПЕРАТОРЫ

Оператор	Значение
&	
^	
&&	
!	

# ЛОГИЧЕСКИЕ ОПЕРАТОРЫ

Оператор	Значение
&	И
	ИЛИ
^	Исключающее ИЛИ
&&	Укороченное И
	Укороченное ИЛИ
!	НЕ



## ОПЕРАТОР IF

Для организации условного ветвления язык C# унаследовал от C и C++ конструкцию if...else.

## ОПЕРАТОР SWITCH

Многонаправленное ветвление программы.

Следовательно, этот оператор позволяет сделать выбор среди нескольких альтернативных вариантов дальнейшего выполнения программы.

# ОПЕРАТОР SWITCH

```
switch(выражение) {
 case константа1:
 последовательность операторов
 break;
 case константа2:
 последовательность операторов
 break;
 case константа3:
 последовательность операторов
 break;

 ...

 default:
 последовательность операторов
 break;
}
```

# ЦИКЛЫ

## Циклы

В C# имеются четыре различных вида циклов (`for`, `while`, `do...while` и `foreach`), позволяющие выполнять блок кода повторно до тех пор, пока удовлетворяется определенное условие. В этой лекции мы познакомимся с циклами *for* и *while*.

# ЦИКЛЫ FOR

**Цикл `for`** в C# предоставляет механизм итерации, в котором определенное условие проверяется перед выполнением каждой итерации.

# ЦИКЛЫ while

Подобно **for**, **while** также является циклом с предварительной проверкой. Синтаксис его аналогичен, но циклы **while** включают только одно выражение