




# Система контроля версий




# Дорожная Карта:

- Основы Git
  - Состояния файлов
  - Первоначальная настройка Git
  - Основной синтаксис для работы
- 



# 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. Поскольку вся история проекта хранится локально у вас на диске, большинство операций кажутся практически мгновенными.
- 

# 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. Эта функциональность встроена в сам фундамент Git'a и является важной составляющей его философии. Если информация потеряется при передаче или повредится на диске, Git всегда это выявит.
- 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'e на основе содержимого файла или структуры каталога. SHA-1 хеш выглядит примерно так:  
24b9da6552252987aa493b52f8696cd6d3b00373

# Состояния

## Зафиксированное состояние файла

- Значит, что файл уже сохранён в вашей локальной базе.

## Измененное состояние файла

- К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы.

## Подготовленное состояние файла

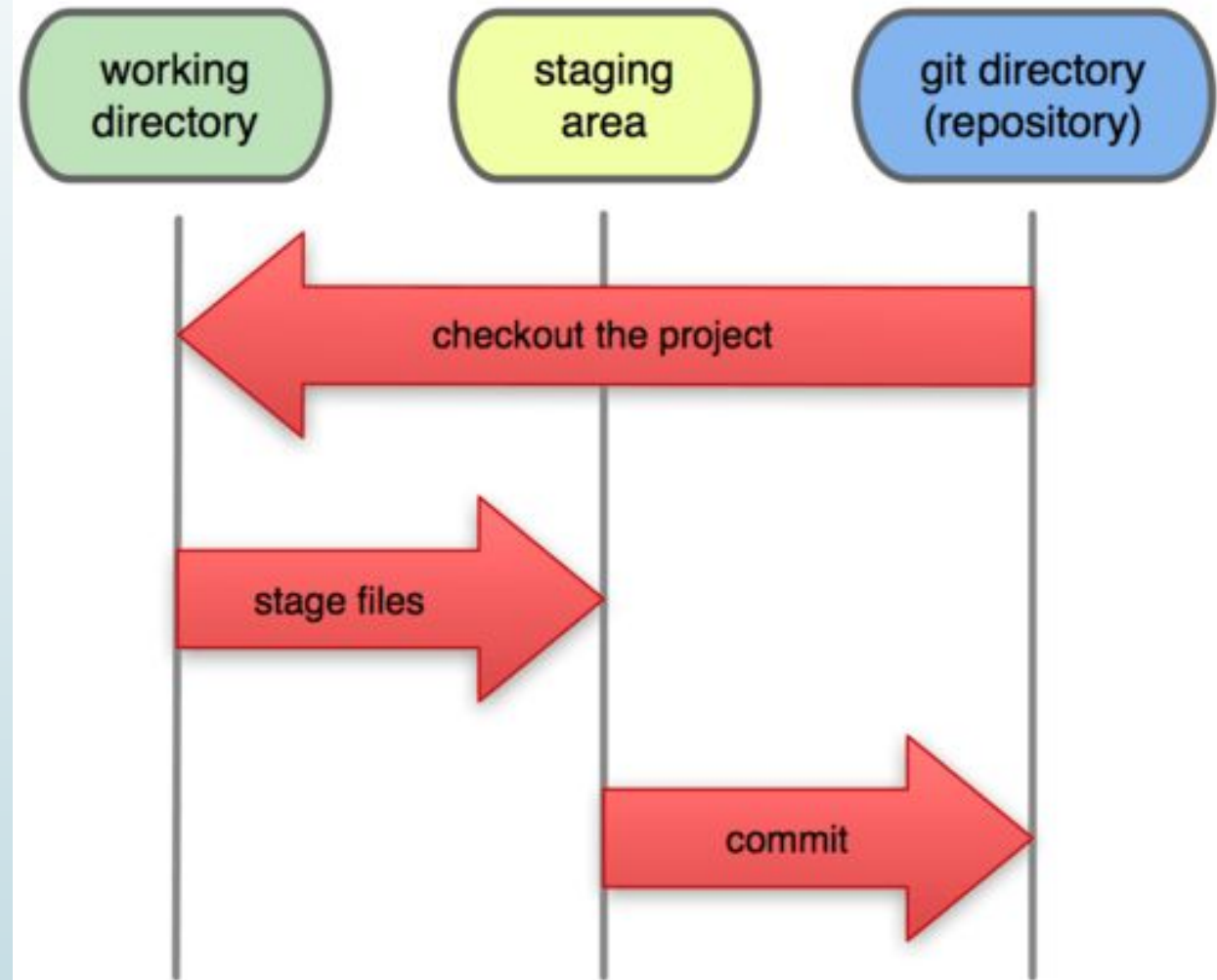
- это изменённые файлы, отмеченные для включения в следующий КОММИТ.

# Рабочий каталог, область подготовленных файлов, каталог Git'a

Области:

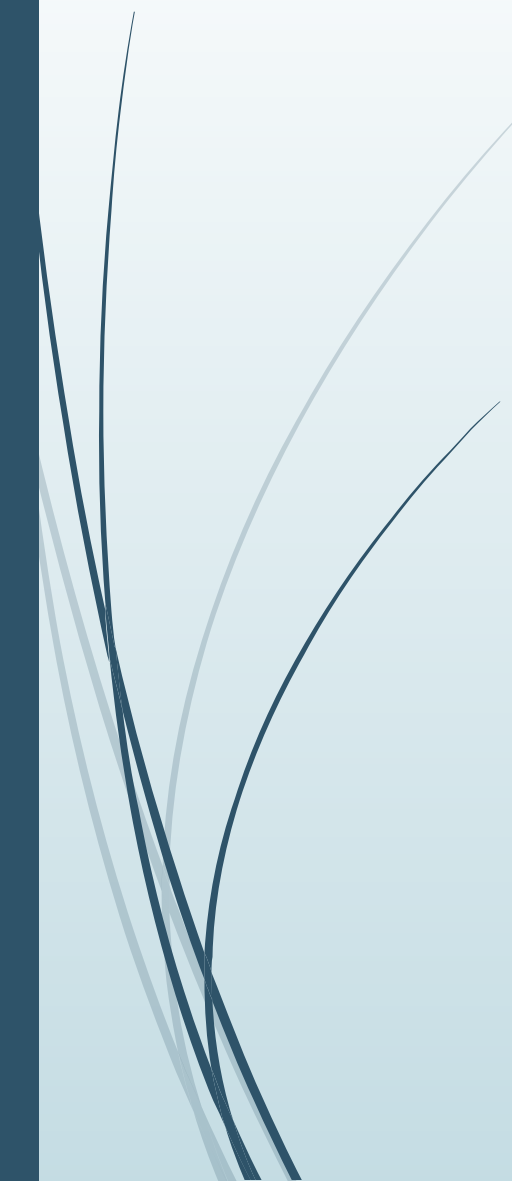
- Каталог Git'a (Git directory)
- Рабочий каталог (working directory)
- Область подготовленных файлов (staging area).

## Local Operations



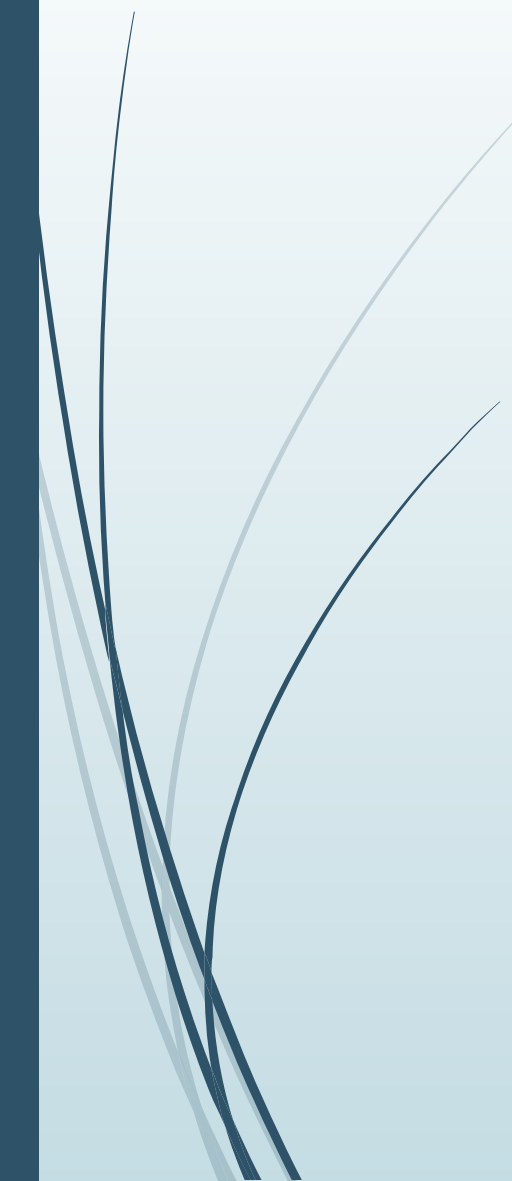


# Каталог Git'a

- Это место, где Git хранит метаданные и базу данных объектов вашего проекта. Это наиболее важная часть Git'a, и именно она копируется, когда вы клонируете репозиторий с другого компьютера.
- 



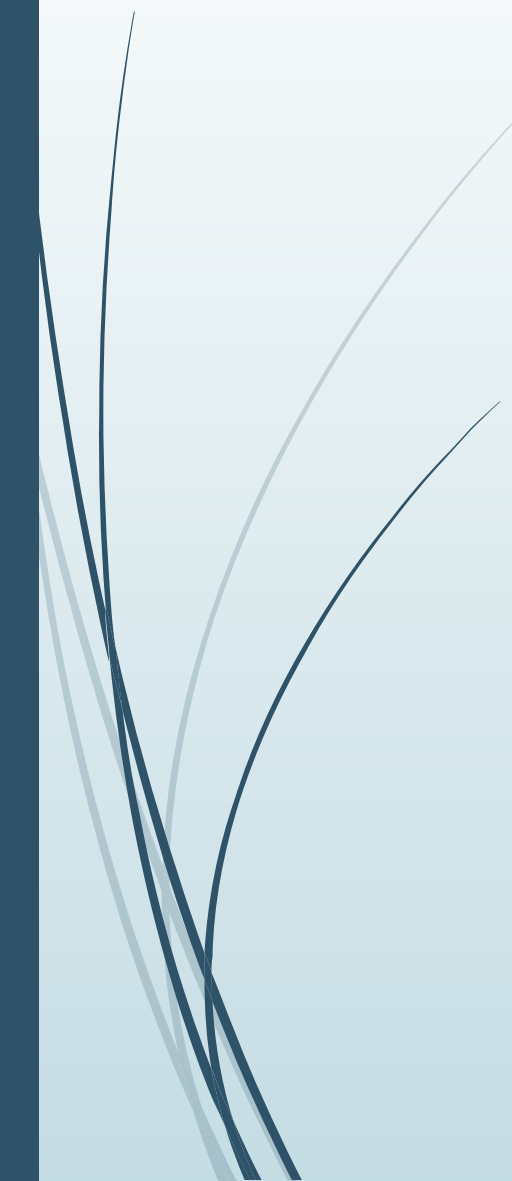
# Рабочий каталог

- Это извлечённая из базы копия определённой версии проекта. Эти файлы достаются из сжатой базы данных в каталоге Git'a и помещаются на диск для того, чтобы вы их просматривали и редактировали.
- 

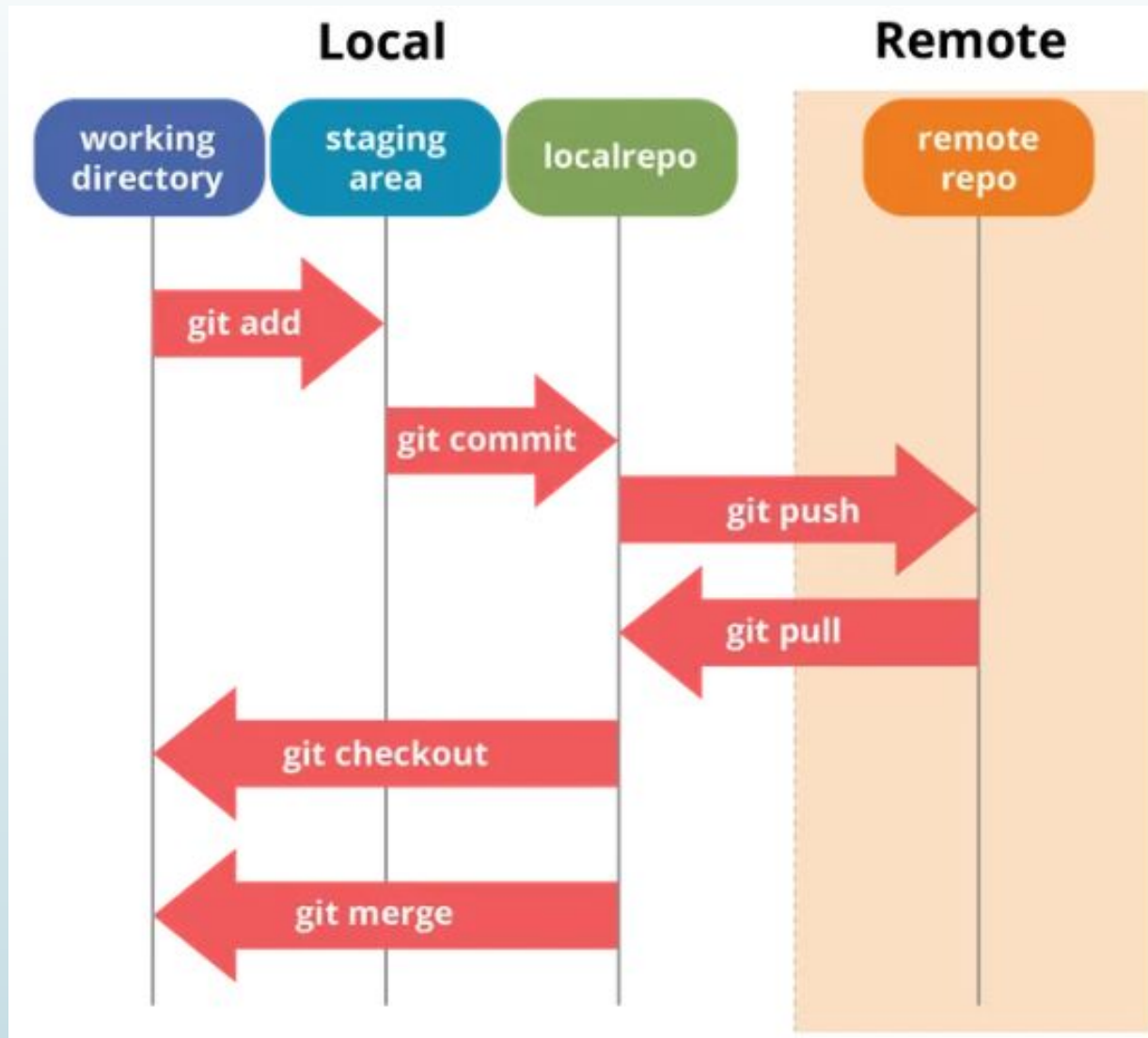


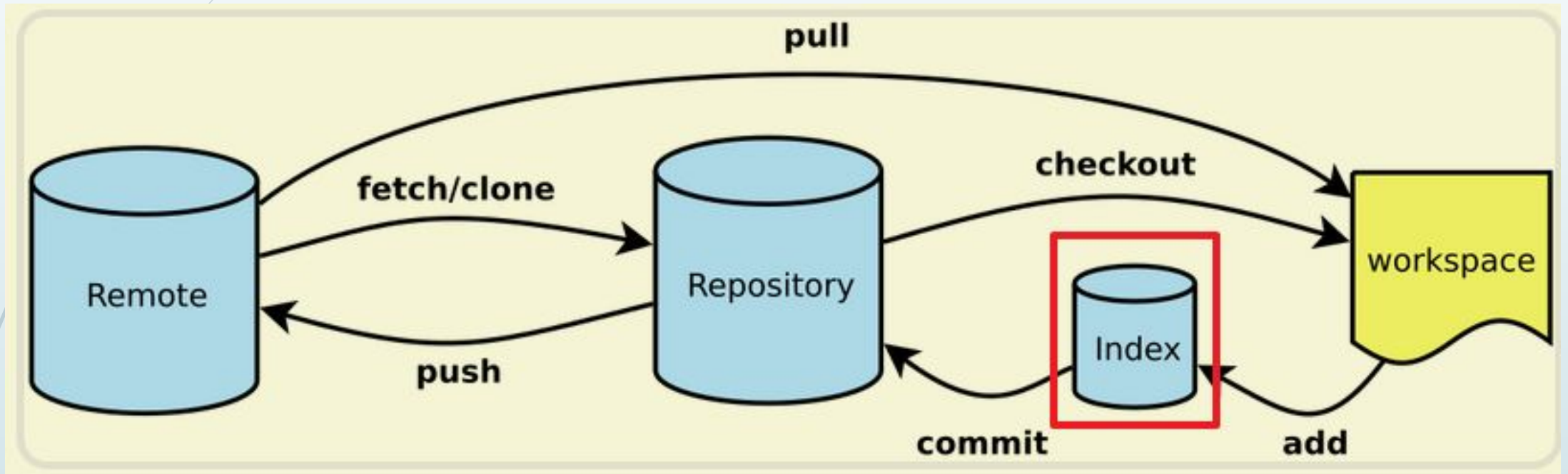


# Область подготовленных файлов

- Это обычный файл, обычно хранящийся в каталоге Git'a, 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).
- 

# Локальный и удаленный репозитории





# Стандартный рабочий процесс с использованием Git'a

- 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'a на постоянное хранение.

□ Комментарий к процессу:

Если рабочая версия файла совпадает с версией в каталоге Git'a, 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.



## Файл ~/.gitconfig

- Хранит настройки конкретного пользователя. Этот файл используется при указании параметра `--global`.
- В системах семейства Windows Git ищет файл `.gitconfig` в каталоге `$HOME` (`C:\$USER` или `C:\Users\$USER`).

# Имя пользователя

- Первое, что вам следует сделать после установки Git'a, — указать ваше имя и адрес электронной почты. Это важно, потому что каждый коммит в Git'e содержит эту информацию, и она включена в коммиты, передаваемые вами, и не может быть далее изменена:

```
$ git config --global user.name "John Doe"
```

```
$ git config --global user.email johndoe@example.com
```

Повторюсь, что, если указана опция `--global`, то эти настройки достаточно сделать только один раз, поскольку в этом случае Git будет использовать эти данные для всего, что вы делаете в этой системе. Если для каких-то отдельных проектов вы хотите указать другое имя или электронную почту, можно выполнить эту же команду без параметра `--global` в каталоге с нужным проектом

# Выбор редактора

- По умолчанию Git использует стандартный редактор вашей системы.. Если вы хотите использовать другой текстовый редактор, например, Notepad++, можно сделать следующее:

```
git config --global core.editor "E:\Games\Notepad++\notepad++.exe"  
-multinst -notabbar -nosession -noPlugin"
```

Открывать файл с кодом можно в любом удобном вам редакторе.

# Проверка настроек

- Если вы хотите проверить используемые настройки, можете использовать команду `git config --list`

Пример:

```
$ git config --list  
user.name=Scott Chacon  
user.email=scon@gmail.com  
color.status=auto  
color.branch=auto  
color.interactive=auto  
color.diff=auto  
...
```




# Как получить помощь?

- Например, так можно открыть руководство по команде config:

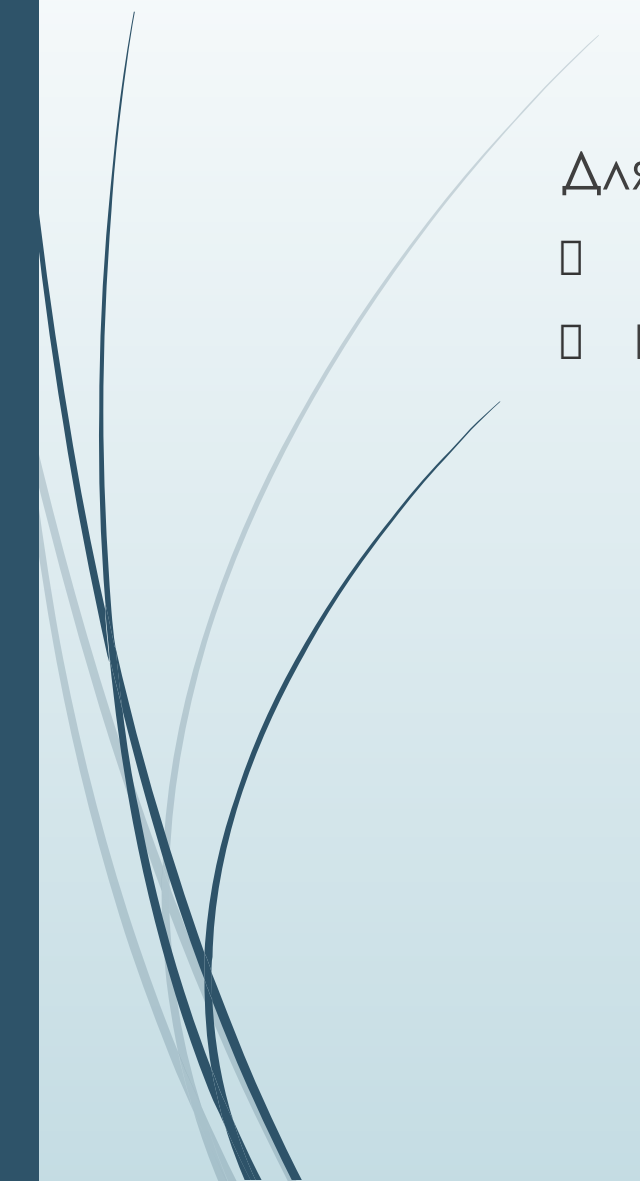
`git help config`

Эти команды хороши тем, что ими можно пользоваться всегда, даже без подключения к сети



# Создание Git – репозитория

Для создания Git-репозитория существуют два основных подхода.

- Импорт в Git уже существующего проекта или каталога.
  - Клонирование уже существующего репозитория с сервера.
- 

# Создание репозитория в существующем каталоге

- Если вы собираетесь начать использовать Git для существующего проекта, то вам необходимо перейти в проектный каталог и в командной строке ввести `git init`.
- Эта команда создаёт в текущем каталоге новый подкаталог с именем `.git` содержащий все необходимые файлы репозитория — основу Git-репозитория. На этом этапе ваш проект ещё не находится под версионным контролем.

# Файл .gitignore

- создаем txt файл = имя .gitignore
- Команды игнора для файлов папок
- # <- комментарий
- logs/(Имя папки)
- # txt files
- docs/\*.txt// Все файлы .txt в этой папке, будут проигнорированы
- показать не отслеживаемые файлы
- `$ git status --untracked-files=all`

# Добавления всех файлов проекта в Git

- Используем команду “`git add .`” и добавляем все файлы под версионный контроль
- Варианты команды:
- `git *.расширение` – все файлы одного расширения
- `Git Имя_файла. Расширение`
- Удаление файла из Git
- `$ git rm --cached ИмяФайла.разширение`
- Добавление всех файлов в коммит
- `$ git commit -a -m"init"(-a = all, -m = комментарий)`

# Клонирование существующего репозитория

- Если вы желаете получить копию существующего репозитория Git, например, проекта, в котором вы хотите поучаствовать, то вам нужна команда: **git clone**.
- Git получает копию практически всех данных, что есть на сервере. Каждая версия каждого файла из истории проекта забирается (pulled) с сервера, когда вы выполняете **git clone**.

Пример: **\$ git clone git://github.com/schacon/grid.git**

Эта команда создаёт каталог с именем **grid**, вы увидите в нём проектные файлы, пригодные для работы и использования. Если вы хотите клонировать репозиторий в каталог, отличный от **grid** то:

Пример: **\$ git clone git://github.com/schacon/grid.git mygrit**

Эта команда делает всё то же самое, что и предыдущая, только результирующий каталог будет назван **mygrit**.

# Запись изменений в репозиторий

Запомните, каждый файл в вашем рабочем каталоге может находиться в одном из двух состояний:

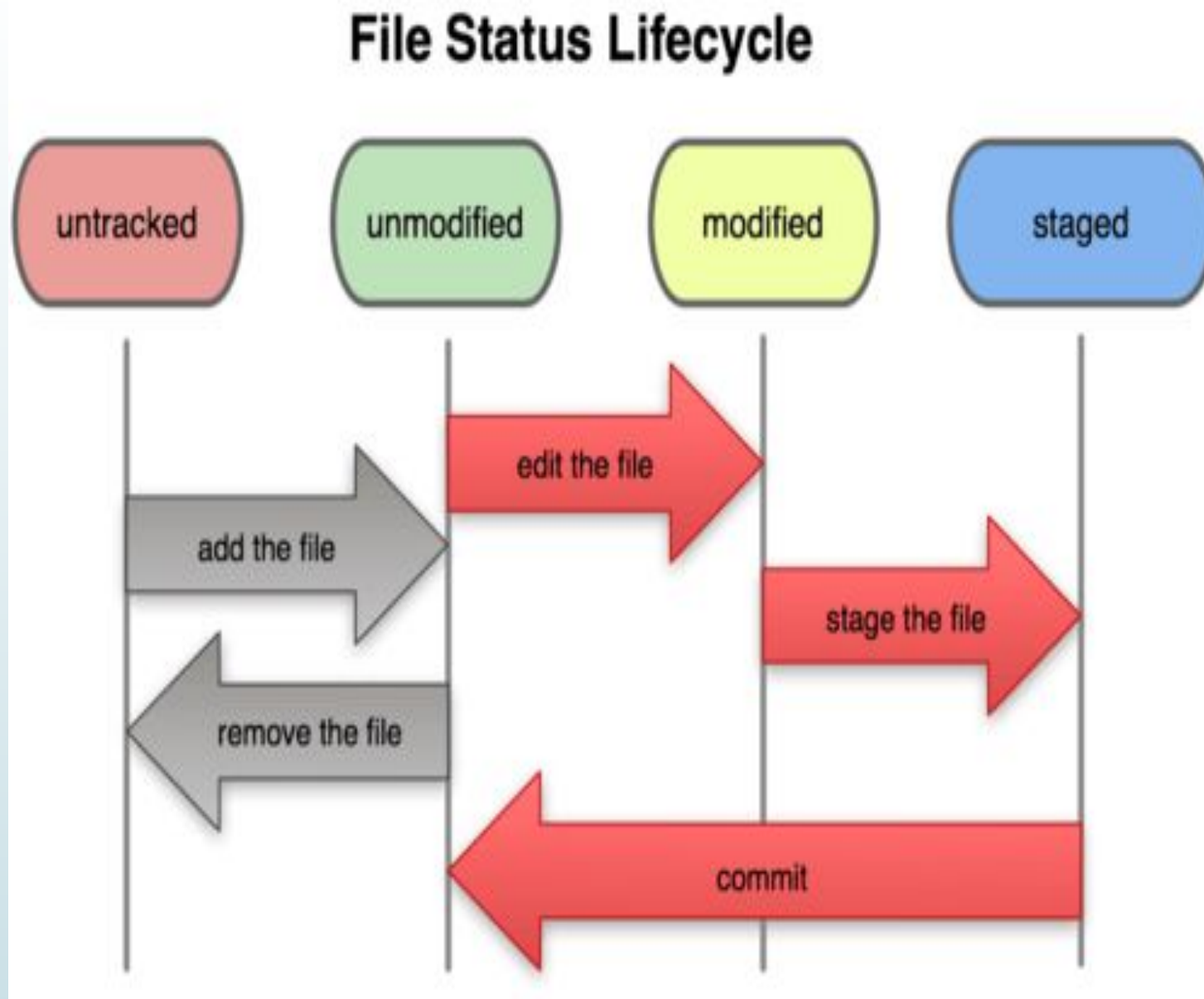
## 1. Под версионным контролем (отслеживаемые)

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

## 2. Нет (неотслеживаемые)

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

# Жизненный цикл состояний файлов





# Определение состояния файлов

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

```
$ git status
```

```
# On branch master
```

```
nothing to commit, working directory clean
```

Это означает, что у вас чистый рабочий каталог, другими словами — в нём нет отслеживаемых изменённых файлов. И наконец, команда сообщает вам на какой ветке (branch) вы сейчас находитесь. Пока что это всегда ветка `master` — это ветка по умолчанию.

# Добавление нового файла в проект

- Предположим, вы добавили в свой проект новый файл, простой файл README. Если этого файла раньше не было, и вы выполните `git status`, вы увидите свой не отслеживаемый файл вот так:
  - `$ git status`
  - `# On branch master`
  - `# Untracked files:`
  - `# (use "git add <file>..." to include in what will be committed)`
  - `#`
  - `# README`
  - `#nothing added to commit but untracked files present (use "git add" to track)`

# Отслеживания новых файлов

- Для того чтобы начать отслеживать (добавить под версионный контроль) новый файл, используется команда `git add`. Чтобы начать отслеживание файла README, вы можете выполнить следующее:

```
$ git add README
```

Если вы снова выполните команду `git status`, то увидите, что файл README теперь отслеживаемый и индексированный:

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
# (use "git reset HEAD <file>..." to unstage)
```

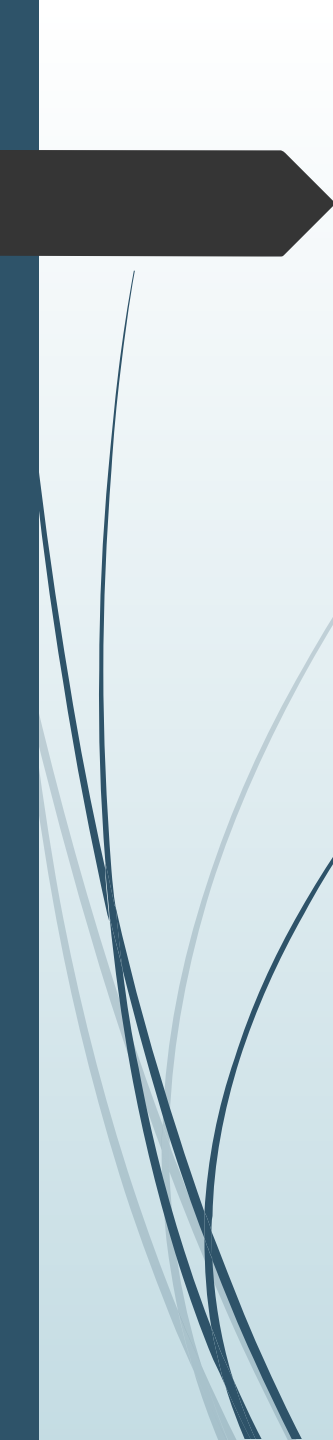
```
#
```

```
# new file: README
```

```
#
```


# Индексация измененных файлов

- Давайте модифицируем файл, уже находящийся под версионным контролем. Если вы измените отслеживаемый файл `benchmarks.rb` и после этого снова выполните команду `git status`, то результат будет примерно следующим:
- `$ git status` ( команда )
- `# On branch master` ( мы в ветке master )
- `# Changes to be committed:`
- `# (use "git reset HEAD <file>..." to unstage)`
- `# new file: README` ( ГОТОВ К КОММИТУ )
- `# Changes not staged for commit:`
- `# (use "git add <file>..." to update what will be committed)`
- `# modified: benchmarks.rb` ( модифицирован )

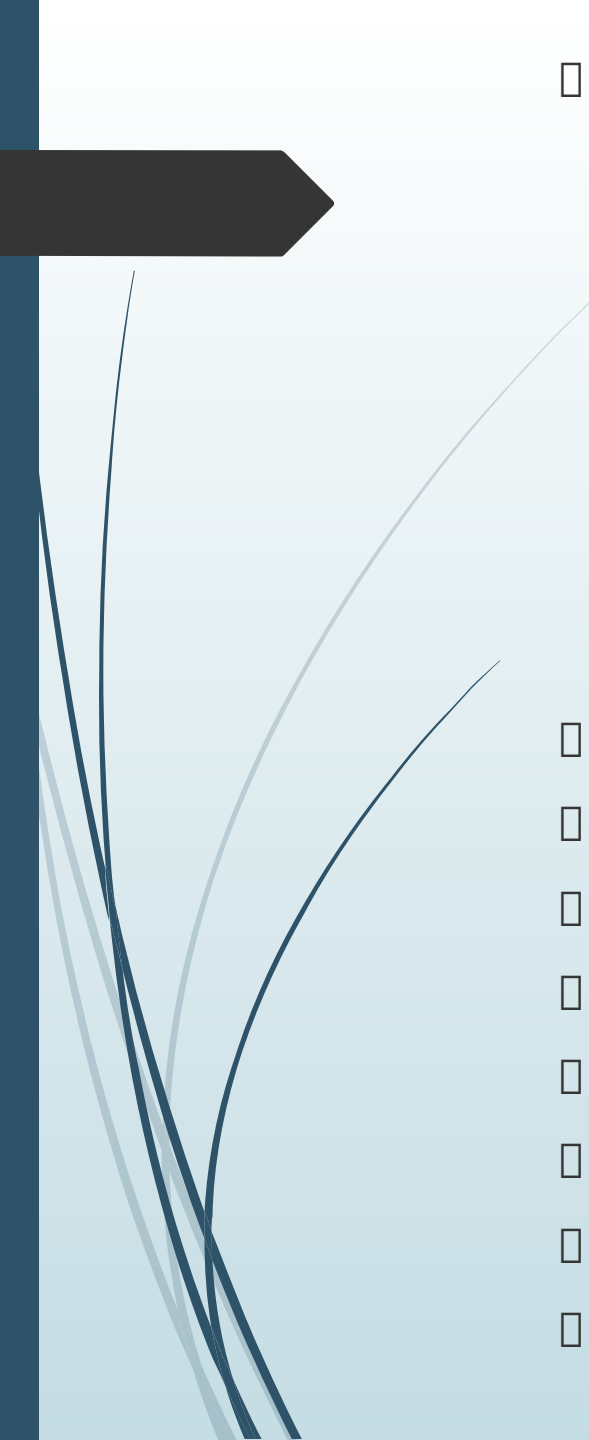


□ Файл `benchmarks.rb` находится в секции “**Changes not staged for commit**” — это означает, что отслеживаемый файл был изменён в рабочем каталоге, но пока не проиндексирован. Чтобы проиндексировать его, необходимо выполнить команду **git add** (это многофункциональная команда, она используется для добавления под версионный контроль новых файлов, для индексации изменений, а также для других целей, например для указания файлов с исправленным конфликтом слияния). Выполним **git add**, чтобы проиндексировать `benchmarks.rb`, а затем снова выполним **git status**:

- **\$ git add benchmarks.rb**
- **\$ git status**
- **# On branch master**
- **# **Changes to be committed:****
- **# (use "git reset HEAD <file>..." to unstage)**
- **#**
- **# new file: README**
- **# modified: benchmarks.rb**
- **#**
- Теперь оба файла проиндексированы и войдут в следующий коммит.

- 
- В этот момент вы, предположим, вспомнили одно небольшое изменение, которое вы хотите сделать в benchmarks.rb до фиксации. Вы открываете файл, вносите и сохраняете необходимые изменения и вроде бы готовы к коммиту. Но давайте-ка ещё раз выполним **git status**:

```
$ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# new file:   README
# modified:   benchmarks.rb
#
# Changes not staged for commit:
# (use "git add <file>..." to update what will be committed)
#
# modified:   benchmarks.rb
```



□ Что за чёрт? Теперь `benchmarks.rb` отображается как проиндексированный и неиндексированный одновременно. Как такое возможно? Такая ситуация наглядно демонстрирует, что Git индексирует файл в точности в том состоянии, в котором он находился, когда вы выполнили команду `git add`. Если вы выполните коммит сейчас, то файл `benchmarks.rb` попадёт в коммит в том состоянии, в котором он находился, когда вы последний раз выполняли команду `git add`, а не в том, в котором он находится в вашем рабочем каталоге в момент выполнения `git commit`. Если вы изменили файл после выполнения `git add`, вам придётся снова выполнить `git add`, чтобы проиндексировать последнюю версию файла:

- `$ git add benchmarks.rb`
- `$ git status`
- `# On branch master`
- `# Changes to be committed:`
- `# (use "git reset HEAD <file>..." to unstage)`
- `#`
- `# new file: README`
- `# modified: benchmarks.rb`

# Откат файла до того состояния которое находится в репозитории

- `git checkout -- LICENSE.php`
- `git revert <hash commit>`



# Просмотр индексированных и не индексированных изменений

- Если результат работы команды `git status` недостаточно информативен для вас — вам хочется знать, что конкретно поменялось, а не только какие файлы были изменены — вы можете использовать команду `git diff`

```
$ git diff
```

```
diff --git a/benchmarks.rb b/benchmarks.rb
```

```
index 3cb747f..da65585 100644
```

```
--- a/benchmarks.rb
```

```
+++ b/benchmarks.rb
```

```
@@ -36,6 +36,10 @@ def main
```

```
    @commit.parents[0].parents[0].parents[0]
```

```
    end
```

```
+ run_code(x, 'commits 1') do
```

```
+   git.commits.size
```

```
+ end + run_code(x, 'commits 2') do log = git.commits('master', 15) log.size
```

# Фиксация изменений

- Теперь, когда ваш индекс настроен так, как вам и хотелось, вы можете зафиксировать свои изменения. Помните, всё, что до сих пор не проиндексировано — любые файлы, созданные или изменённые вами, и для которых вы не выполнили `git add` после момента редактирования — не войдут в этот коммит. Простейший способ зафиксировать изменения — это набрать `git commit`: Эта команда откроет выбранный вами текстовый редактор.

Есть и другой способ — вы можете набрать свой комментарий к коммиту в командной строке вместе с командой `commit`, указав его после параметра `-m`, как в следующем примере:

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

```
[master]: created 463dc4f: "Fix benchmarks for speed"
```

```
2 files changed, 3 insertions(+), 0 deletions(-)
```

```
create mode 100644 README
```

Добавление всех файлов в коммит + индексация

```
$ git commit -a -m "init" (-a = all, -m = комментарий)
```

# Удаление файлов

- Другая полезная штука, которую вы можете захотеть сделать — это удалить файл из индекса, оставив его при этом в рабочем каталоге. Другими словами, вы можете захотеть оставить файл на винчестере, и убрать его из-под бдительного ока Git'a.

- `$ git rm --cached ИмяФайла.разширение`

Это особенно полезно, если вы забыли добавить что-то в файл `.gitignore`

# Перемещение файлов

- Таким образом, наличие в Git'e команды `mv` выглядит несколько странным. Если вам хочется переименовать файл в Git'e, вы можете сделать что-то вроде: `$ git mv file_from file_to`

# Просмотр истории комитов

- Наиболее простой и в то же время мощный инструмент для этого — команда `git log`.
- Один из наиболее полезных параметров — это `-p`, который показывает дельту (разницу/diff), внесенную каждым коммитом. Вы также можете использовать `-2` что ограничит вывод до 2-х последних записей.
- Наиболее интересный параметр — это `format`
- `$ git log --pretty=format:"%h - %an, %ar : %s"`
- `ca82a6d - Scott Chacon, 11 months ago : changed the version number`  
`085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code`  
`a11bef0 - Scott Chacon, 11 months ago : first commit`



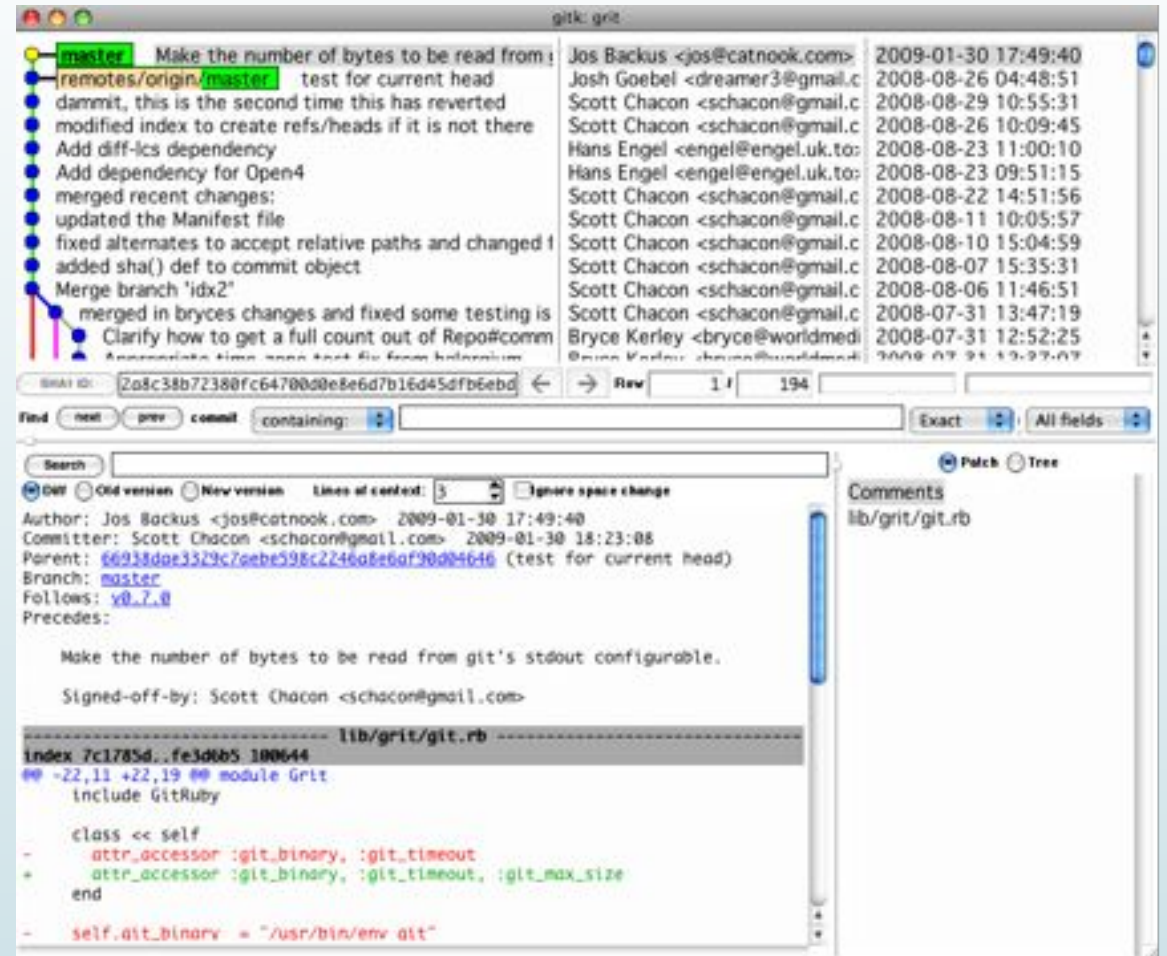
# Параметры формата

Параметр	Описание выводимых данных
%H	Хеш коммита
%h	Сокращённый хеш коммита
%T	Хеш дерева
%t	Сокращённый хеш дерева
%P	Хеши родительских коммитов
%p	Сокращённые хеши родительских коммитов
%an	Имя автора
%ae	Электронная почта автора
%ad	Дата автора (формат соответствует параметру --date=)
%ar	Дата автора, относительная (пр. "2 мес. назад")
%cn	Имя коммитера
%ce	Электронная почта коммитера
%cd	Дата коммитера
%cr	Дата коммитера, относительная
%s	Комментарий

# Ограничение вывода команды log

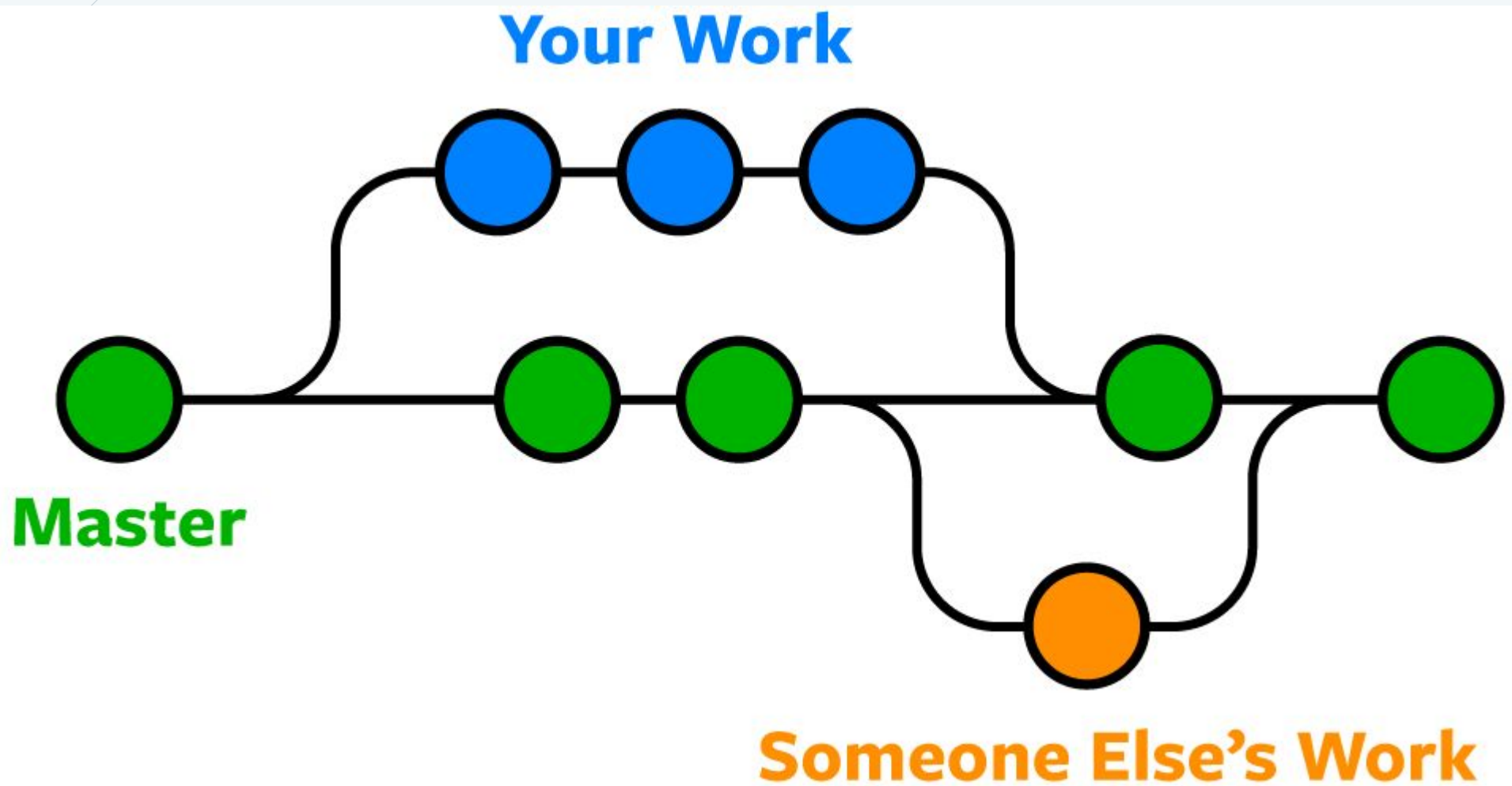
- вот параметры, ограничивающие по времени, такие как `--since` и `--until`, весьма полезны. Например, следующая команда выдаёт список коммитов, сделанных за последние две недели:
- `$ git log --since=2.weeks`

- Если наберёте в командной строке `gitk`, находясь в проекте, то увидите что-то наподобие



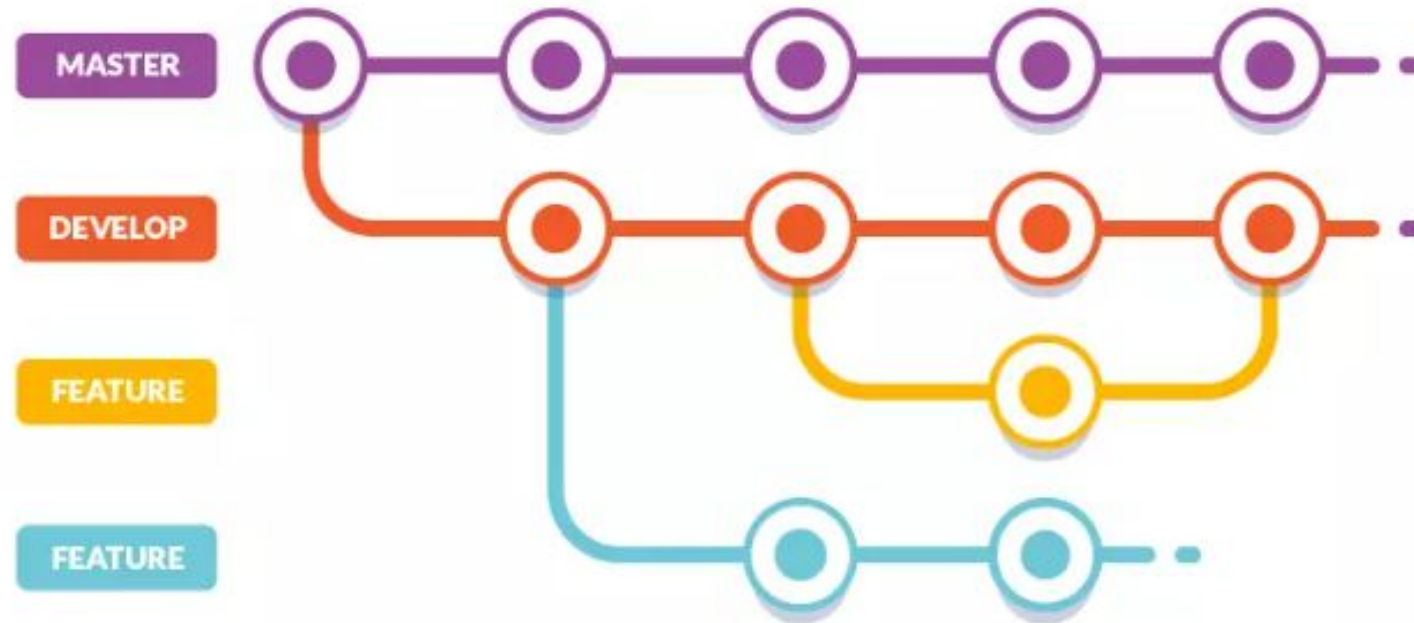


# Создание ветвлений в Git



# Создание ветвлений в Git

## Gitflow Branch Strategy





□ Команды:

□ `$ git checkout -b "name"//` Имя новой ветки, создать и сразу переключится

□ Посмотреть ветки

`$ git branch` (-v доп. инфо и последние комиты)

- Просто создание ветки без переключения

`$ git branch "namebranch"`

- Переключение на другую ветку

`$ git checkout "namebranch"`

# Слияние веток

## Разрешение конфликтов при слиянии

- Указание утилиты для слияния: `$ git config --global merge.tool kdiff3`
- Команда на слияние (заливаем в свою ветку другую ветку) `$ git merge master`
- Запускаем утилиту для мержа(которую мы прописали в конфиг) `$ git mergetool`
- Надо скачать `kdiff3` <https://sourceforge.net/projects/kdiff3/files/>
- И добавляем адресс в Git
- `$ git config --global mergetool.kdiff3.cmd "'F:\\\\KDiff3\\\\kdiff3' $BASE $LOCAL $REMOTE -o $MERGED'`
- теперь можно мержить
- `$ git mergetool`

# Регистрация на GitHub и делаем push(заталкиваем проект в веб)

- ❑ `git remote add origin https://github.com/....git`
- ❑ `git push -u origin master` (-u нужен только раз потом просто `push`)
- ❑ заталкать в веб `$ git push`
- ❑ посмотреть количество репозиториев `git remote` (-v путь к данному репозиторию)
- ❑ Команда для проталкивания в веб всех локальных веток:
- ❑ `$ git config --global push.default matching`

# Клонирование проекта себе в папку

- Клонировать весь проект `git clone https://github.com/....git` (ссылка взята с GitHub)
- Забрать файл из веб репозитория себе в локальную базу .git: `$ git fetch`
- для того что б файл появился в рабочей папке `$ git pull`