Работа с ветками

Базовые команды

Основные темы

- Совместная работа над исходниками
- Создание веток
- Стратегии работы с ветками, workflow
- Слияние веток
- Merge-requests
- Защищенные ветки
- Разрешение конфликтов при слиянии
- Настройка прав репозитория

Совместная работа над исходниками

- Пока мы рассматривали работу над исходниками в одиночку
  - Некоторые его так и используют
- Системы контроля версий нужны в первую очередь для поддержки совместной работы
- В чем проблема:
  - Одновременно над проектом работает N человек
  - Они могут изменять одни и те же файлы
  - При этом неизбежно будут возникать конфликты

Совместная работа над исходниками

- Задачи, возникающие при работе над проектом
  - Отладка/тестирование изменений (конкретных пользователей и системы в целом)
  - Необходимость отслеживать изменения, внесенные разными участниками
  - Необходимость апрувить изменения junior сотрудников, проводить code review
  - Вносить срочные изменения (hotfix) при обнаружении критических ошибок
  - Ускорить процесс доставки разработанных модулей по пользователя, пройдя все стадии тестирования и проверки на безопасность
  - Сохранить понимание, кто, что и зачем исправил
  - И т.п.
- Процессом надо как-то управлять

Использование веток

- Использование веток для создания нового функционала
  - Feature driven development
- Паттерны работы с ветками
  - Политики создания веток (Gitflow, Github flow, Gitlab flow)
- Работа с ветками в git (создание, слияние, уничтожение)
- Слияние перемоткой

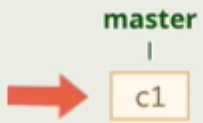
Использование веток

- В реальном проекте, не получится работать без выстраивания gitflow

Использование веток

- Ветка – изолированный поток разработки, в который можно делать коммиты так, что их не будет видно из других веток.
  - Как только мы делаем первый коммит, гит создает первую ветвь разработки, которая называется master (теперь main).
- Можно делать коммиты в одну ветку
  - Как мы делали до сих пор
- Но при совместной работе нескольких человек, неизбежно возникнут проблемы и неудобство
- Удачное решение для этого – разработка разных свойств и функционала в разных ветках
  - Тогда части кода будут «изолированы» друг от друга в процессе разработки

Как только мы делаем первый коммит, гит создает первую ветвь разработки, которая называется master.



Допустим, мы хотим добавить какую то функциональность в проект, и это требует времени.

Пока функциональность не завершена – мы не хотим ее незавершенный код смешивать с основным кодом проекта (т.е. мы не хотим держать Work In Progress в основном репозитории).

Мы можем в рамках того же проекта, в рамках того же репозитория создать новую ветвь разработки, и назвать её, например, feature.

Мы делаем ветку feature нашей **текущей** веткой, и коммитим уже в ней.

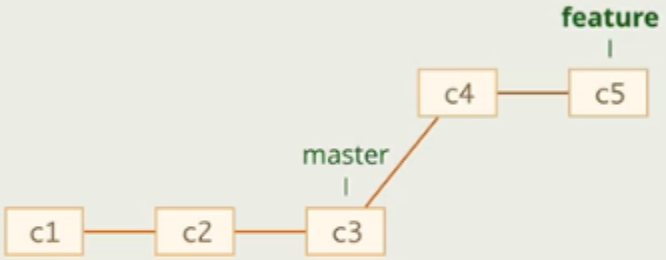
Использование веток

- Ветки позволяют
  - Разделить работы над разными feature
  - Разделить текущую работу разных людей
  - Держать незавершенную работу отдельно от основного кода, чтобы не сломать систему

Использование веток

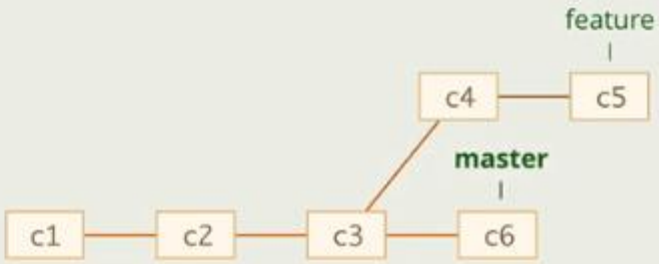
- Когда стоит делать ветку?
- Есть разные политики, но наиболее удачные сходятся на том, что для разработки новой отдельной feature стоит создать новую ветку
  - Это хорошо сочетается с методологией работы feature driven development
- Решили создать новую ветку для разрабатываемой feature, делаем коммиты в нее

Мы делаем ветку feature нашей **текущей** веткой, и коммитим уже в ней.

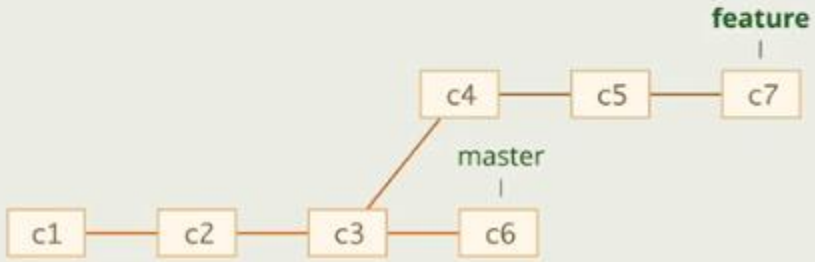


Мы делаем текущие коммиты в ветку feature (там может быть в том числе незавершенный или н до конца оттестированный код).

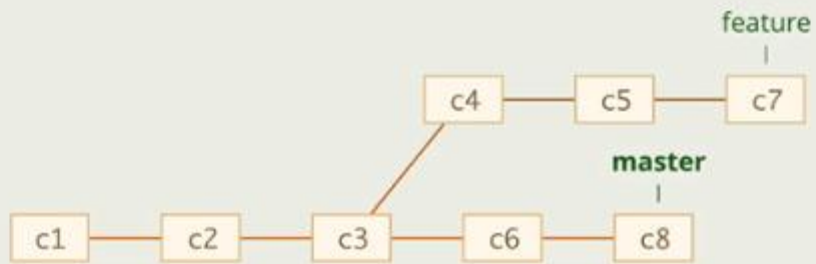
Git позволяет легко переключаться между ветками в любое время



Можно вернуться на ветку master, и поделать что-то в ней.



Потом можно вернуться в ветку feature, и опять поработать над новой идеей.

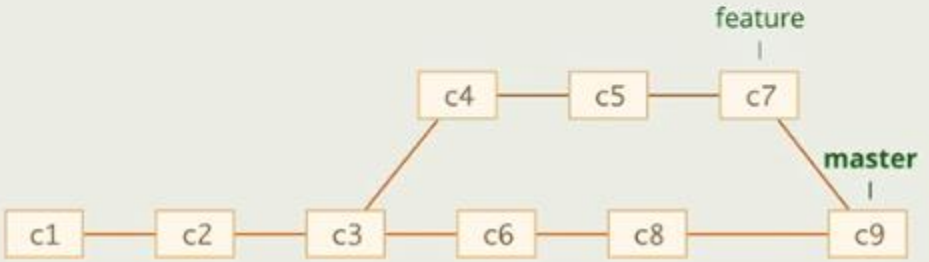

feature
c4
c5
c7
master
c1
c2
c3
c6
c8

Использование веток

- Можно переключатся с ветки на ветку в любой момент и работать над разными feature в разных ветках
- Git позволяет делать это с легкостью, и при переключении, вы видите ту версию файлов, которая относится к текущей активной ветке
- Можно создать ветку локально, поработать с ней, а потом удалить
- Можно создать ветку локально, и при push автоматически создать ее origin на сервере
- Можно сделать merge из ветки обратно в базовую ветку

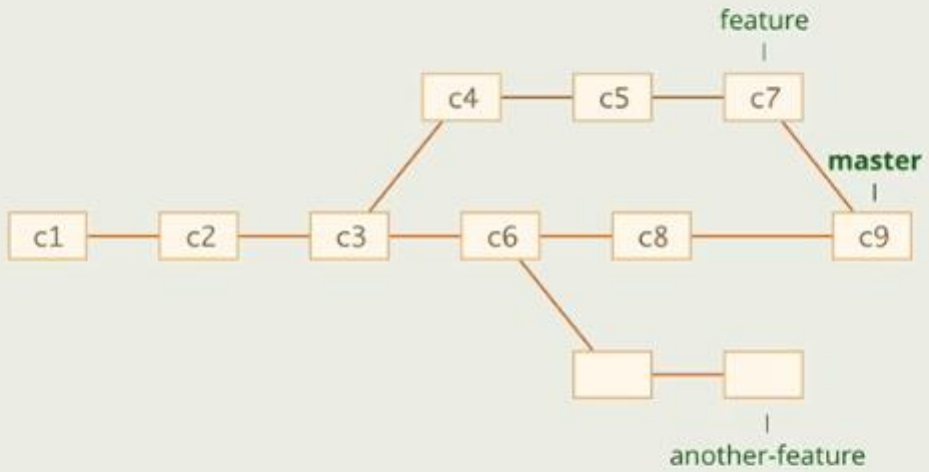
Если функционал из feature оказался не нужен, можно просто удалить ветку feature без вреда для основной ветки разработки.

Когда новый функционал готов, его можно объединить с основной веткой master.
Все изменения из feature ветки «вливаются» в основную ветку



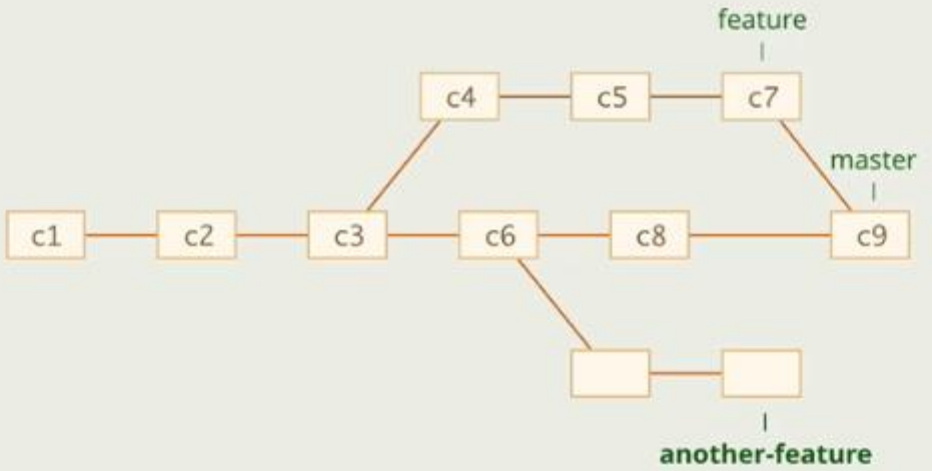
Подход называется – тематические ветки, новую функциональность мы реализуем в отдельных ветках, и интегрируем в master когда она готова

Тематических веток разработки может быть много, и каждая ветка будет касаться какой-то отдельной задачи



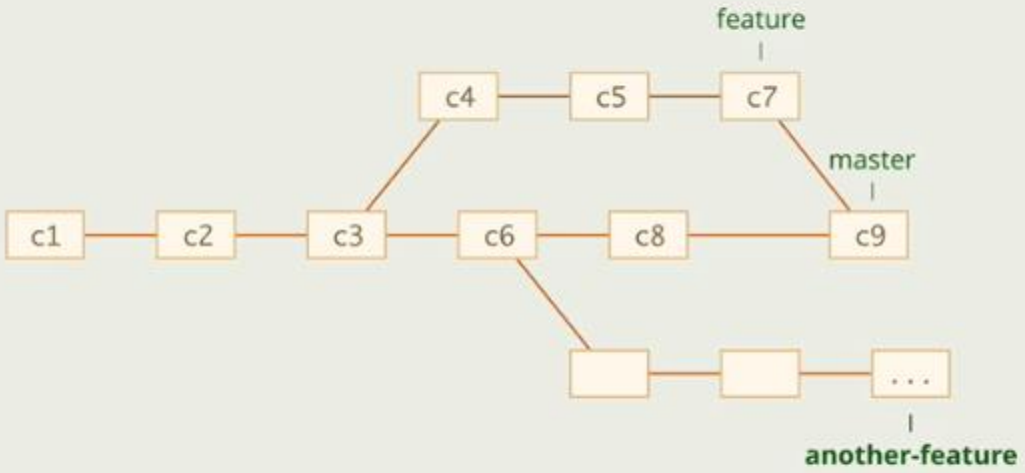
Каждый разработчик вносит изменения в свою ветку, не мешая остальным.

При этом мы всегда можем переключиться на ветку коллеги, посмотреть, что он делает.

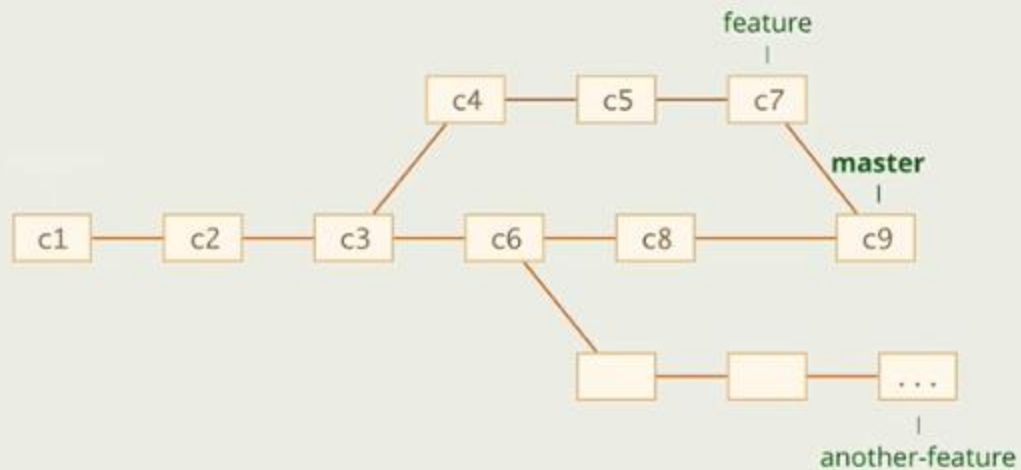


Например, senior разработчик может контролировать ход работ младших коллег

Можно даже внести какие-то правки

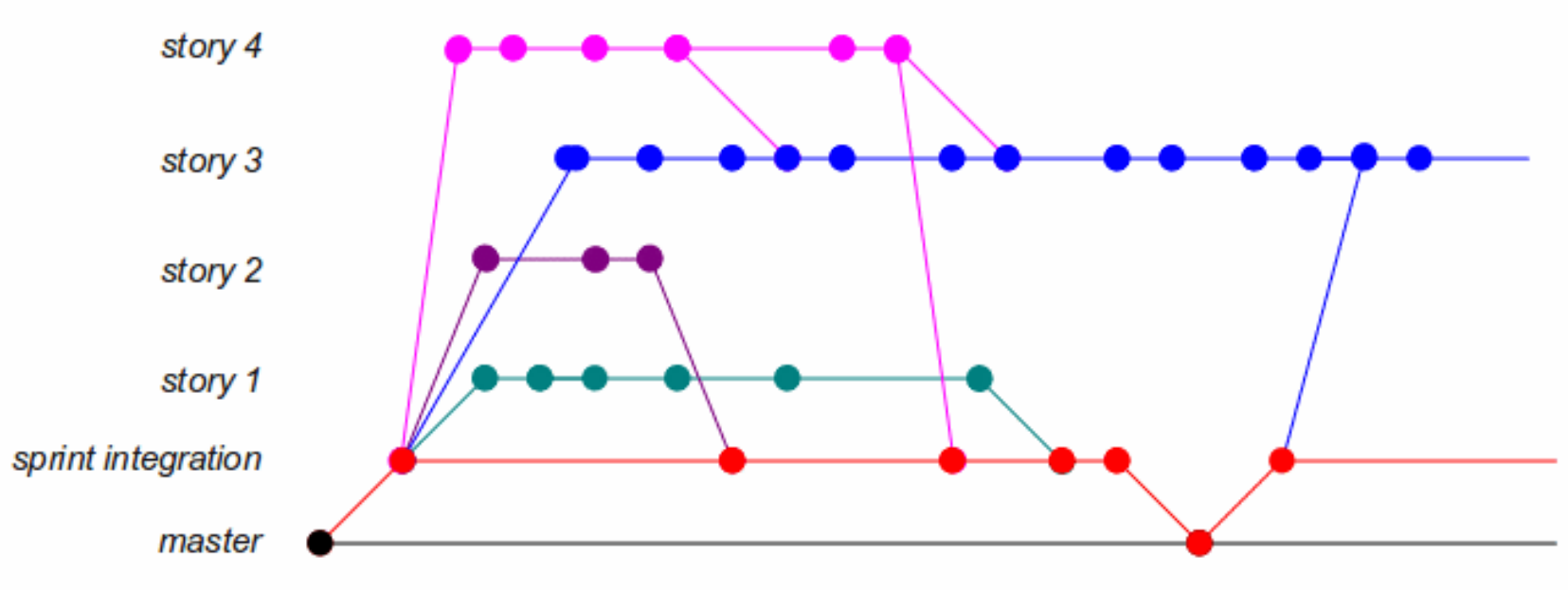


Например, senior разработчик может контролировать ход работ младших коллег


feature
c4
c5
c7
master
c1
c2
c3
c6
c8
c9
...
another-feature

Использование веток

- Дерево веток в итоге может быть очень сложным
  - Ясно, что должна быть какая-то общепринятая политика работы с ветками
  - Посмотрим еще несколько примеров

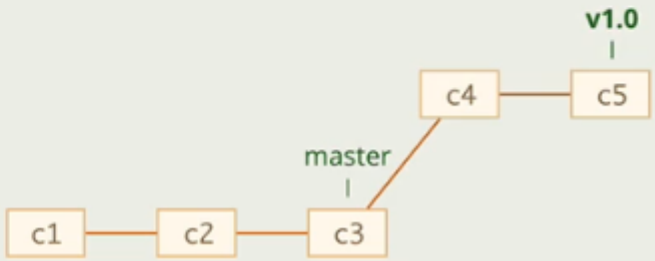



master
c1
c2
c3

Для нового релиза создадим ветку 1.0.

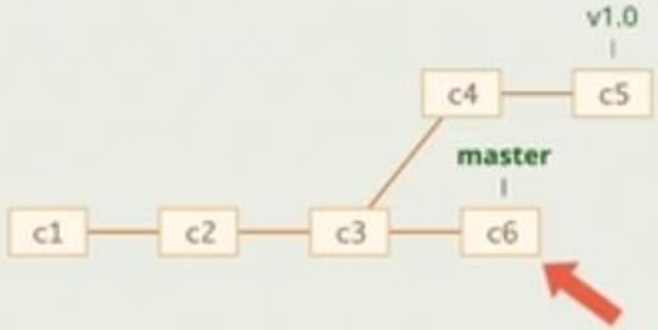
В ветке мастер мы продолжим делать новые фичи
В ветке 1.0 (релизной) мы будем шлифовать те возможности, которые были включены в релиз, но ничего нового мы писать не будем.

Переключаемся на ветку 1.0, готовим её к релизу.



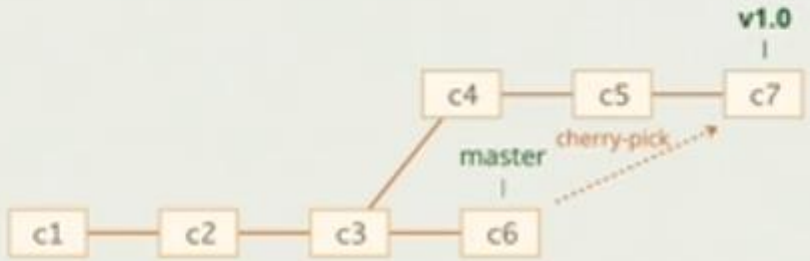
Параллельно может идти разработка в ветке master

Предположим, потребовалось что-то поправить в обеих ветках (хотфикс).

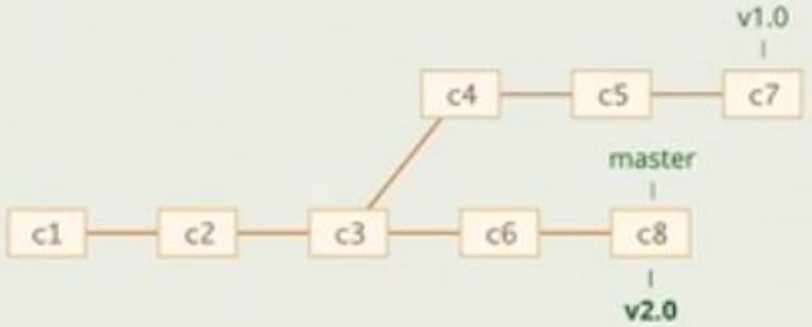


Исправляется ошибка в одной из веток, например, в ветке master (c6 – исправляющий коммит).

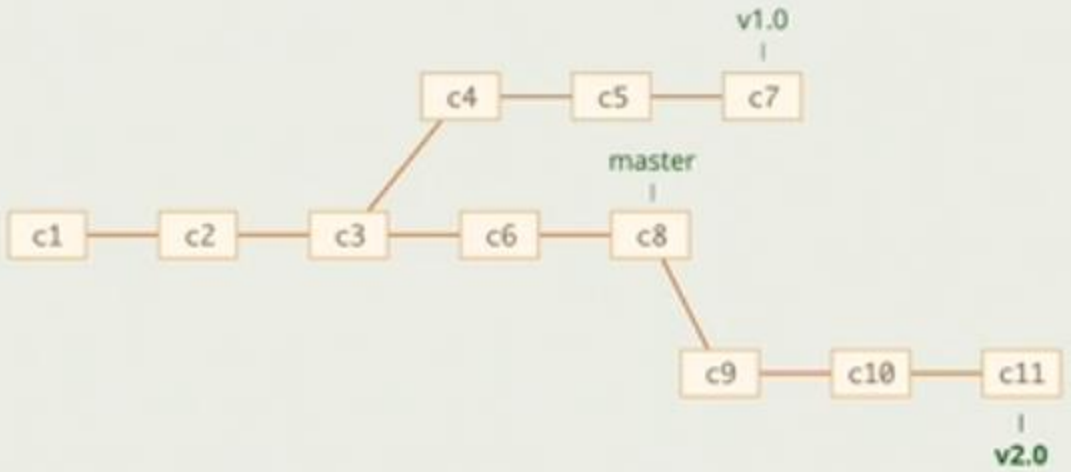
git позволяет взять любой коммит, и применить его к другой ветке (сделать так называемый cherry-picking).



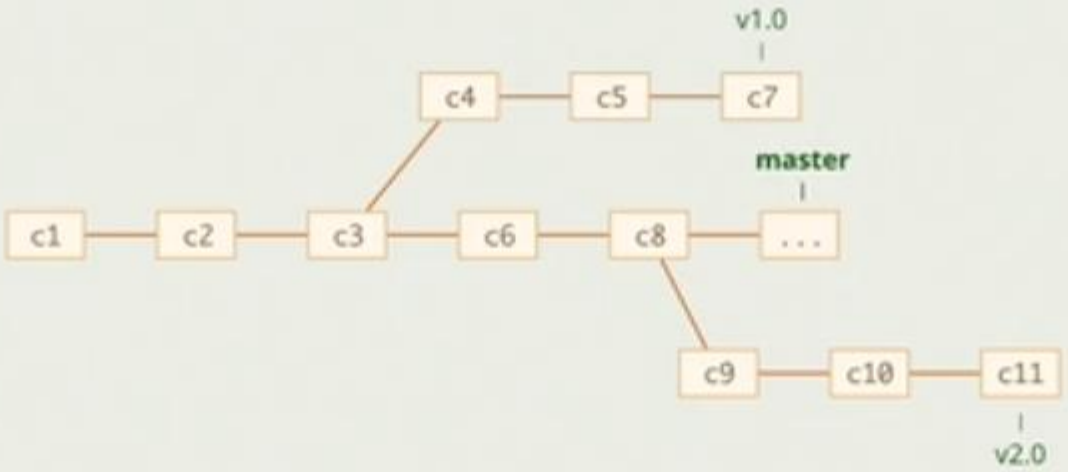
Через какое-то время нужно делать новый релиз.
Создаем ветку v2.0, переключаемся на неё.



Там идет шлифовка тех возможностей, которые было решено включить в релиз 2.0.

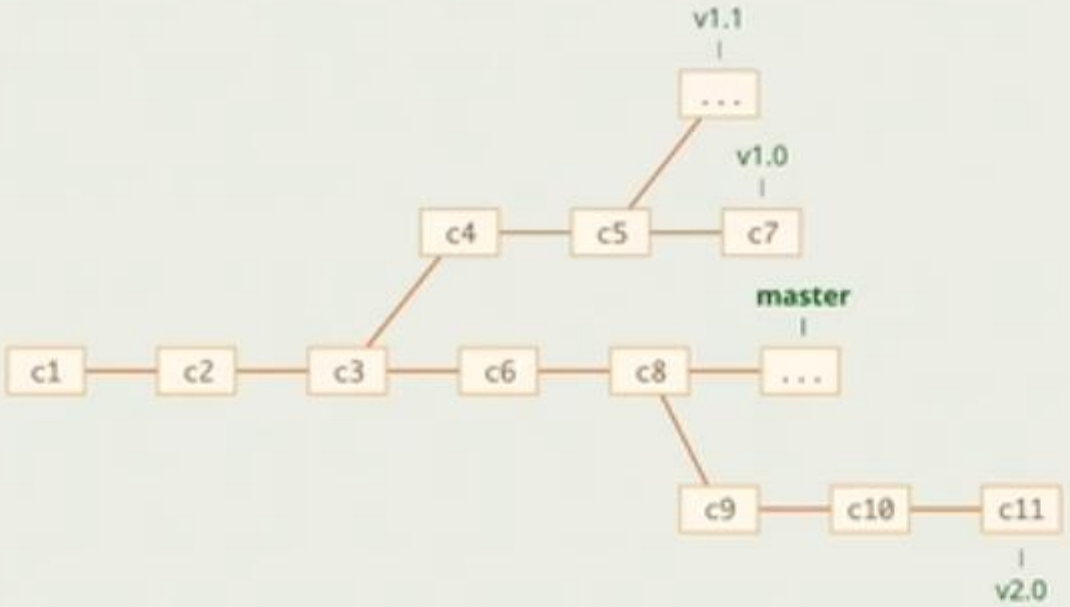


Там идет шлифовка тех возможностей, которые было решено включить в релиз 2.0.



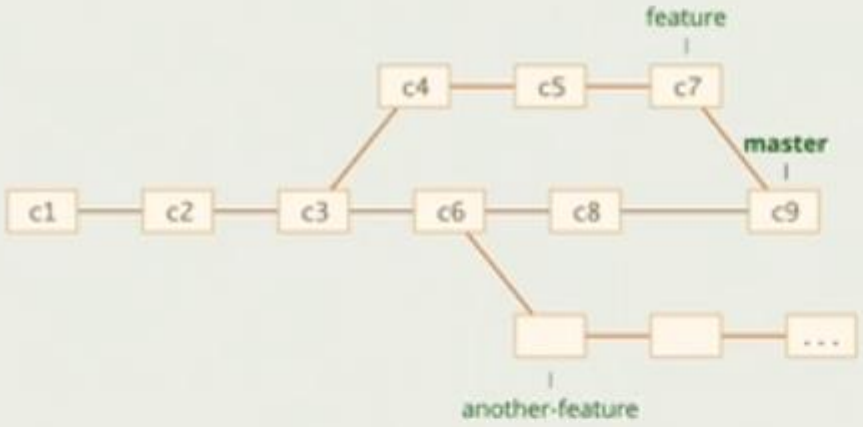
В ветке master делается то, что предназначено для следующих релизов, и.т.д.

git позволяет создавать любое количество веток от любого коммита

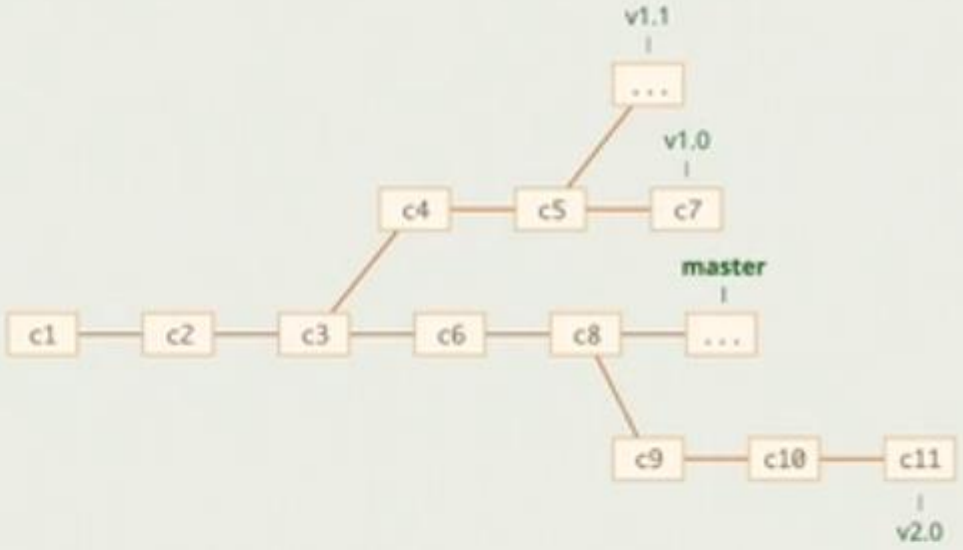


Например, от ветки 1.0 можно создать ветку 1.1

Тематические



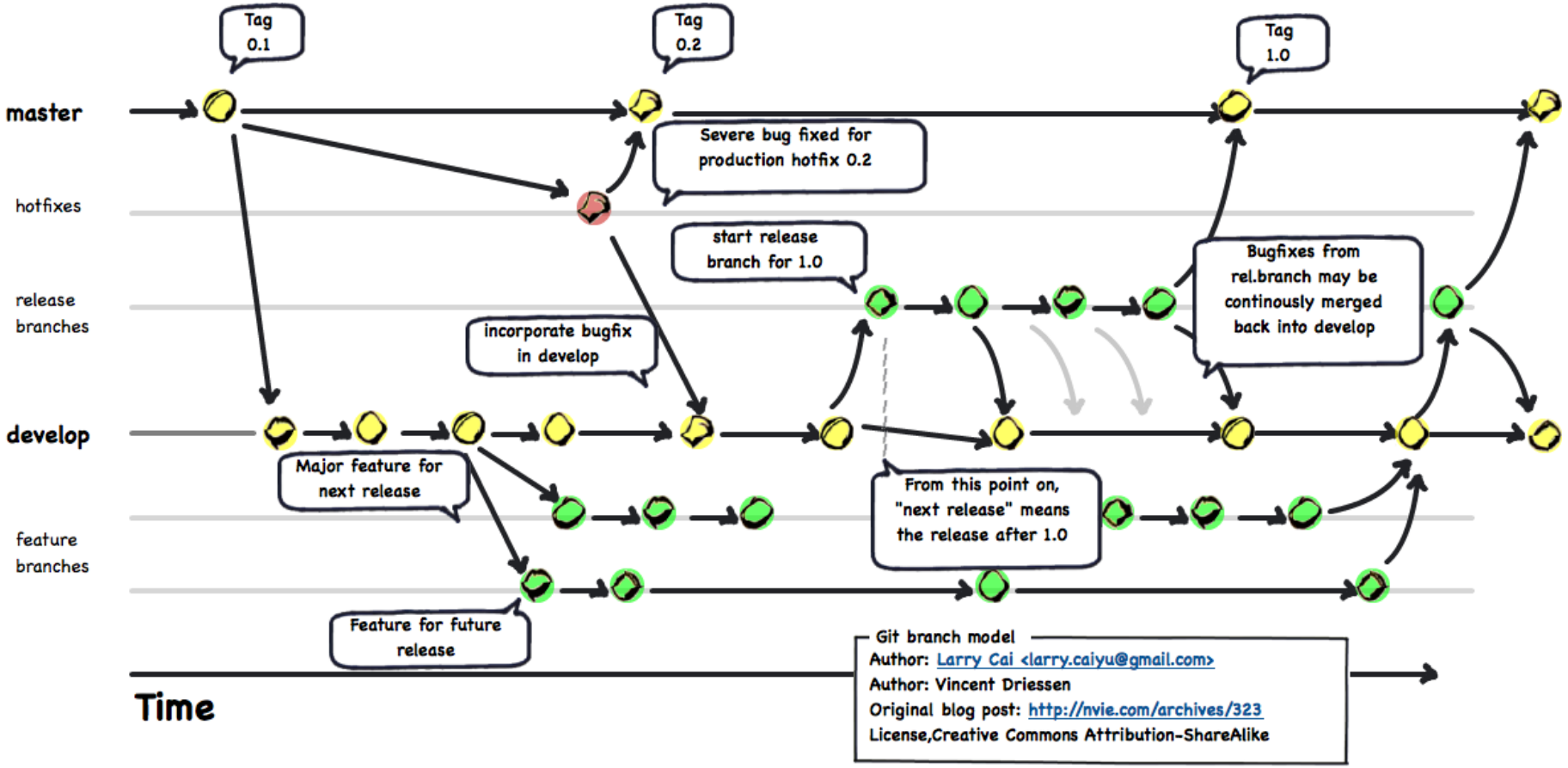
Релизные



Некоторые проекты используют тематические ветки, некоторые релизные, некоторые их сочетания

Использование веток

- В классическом gitflow, изобретенном 2010, все было еще запутаннее



Github flow

- Это лишь одна из моделей, которая использует одновременно тематические ветки, релизные ветки, ветки master/development, ветки для исправления багов
  - A successful git branching model
  - https://nvie.com/posts/a-successful-git-branching-model/
  - Кстати, для автоматизации этих процессов есть утилиты такие как gitflow-awh

Github flow

- Это лишь одна из моделей, которая использует одновременно тематические ветки, релизные ветки, ветки master/development, ветки для исправления багов
  - Основные ветки – master и develop
  - В master лежит текущая рабочая готовая к деплою версия
  - В develop – текущая рабочая версия с последними изменениями, принятыми и готовыми к внедрению

Github flow

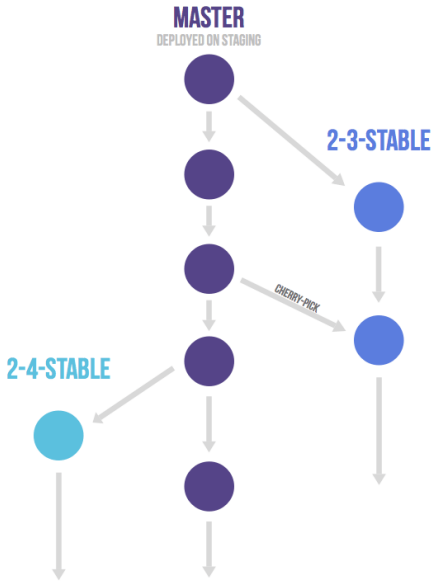
- Основные правила
  - Anything in the master branch is deployable
  - To work on something new, create a descriptively named branch off of master (ie: new-oauth2-scopes)
  - Commit to that branch locally and regularly push your work to the same named branch on the server
  - When you need feedback or help, or you think the branch is ready for merging, open a pull request (merge request)
  - After someone else has reviewed and signed off on the feature, you can merge it into master
  - Once it is merged and pushed to ‘master’, you can and should deploy immediately

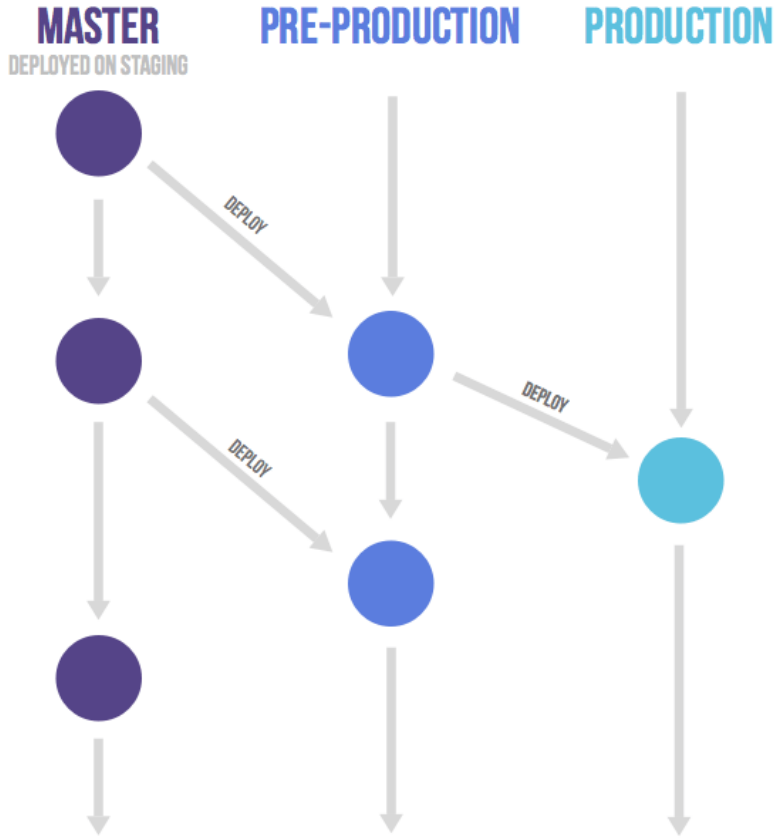
Github flow - критика

- Изобретена до активного внедрения continuous integration
- Многим кажется запутанной. Почему?
- Основные проблемы
  - Hotfix и release branches – не всегда они нужны, но могут запутать процесс
  - Внедрение CI/CD во многом убирает необходимость создавать эти ветки

Gitlab flow

- Основные правила
  - All features and fixes first go to master
  - Allows for production or stable branches
  - Bug fixes/hot fix patches are cherry-picked from master





Какой gitflow использовать

- Какую стратегию работы с ветками принимать – зависит от проекта
- Хорошая практика разделять develop/master ветки, но можно без этого обойтись
- Feature ветки не оспариваются
- Делать ли ветки для релизов и hotfix, решать в зависимости от требований проекта

Какой gitflow использовать

- Есть ветки для dev / stage
  - Из них выкладывается deploy на staging сервер для ручного тестирования
- Есть ветка для prod (это main или специальная ветка release)
- Изменения передаются из одних веток в другие по определенным правилам – когда создавать новую ветку, когда вливать обновления в свою ветку и т.п.
- Для релиза можно использовать как ветки, так и теги
- Для hotfix можно не создавать отдельную ветку (при корректной настройке CI/CD)
  - Изменения hotfix коммитов добавляются в dev используя cherry-pick

Зачем вообще столько веток?

- Контроль от попадания непроверенных изменений в продуктовые и рабочие ветки
- Ясное понимание того, зачем нужна каждая конкретная ветка
- Защита от устаревания и расхождения веток разных участников
  - Для этой цели служат периодические merge в свою ветку из рабочей
- Упрощение процедуры проверки внесения нового кода в рабочие ветки
- Регламентация процедур тестирования новых feature
  - При попадании в соответствующую ветку
- Кроме того, на такую схему отлично ложится CI/CD

Практика – работа с ветками

- Создание веток
  - git branch feature
- Переключение на ветку
  - git checkout feature
- Создание и переключение на ветку
  - git checkout –b feature
- Просмотр указателя
  - cat .git/refs/heads/feature

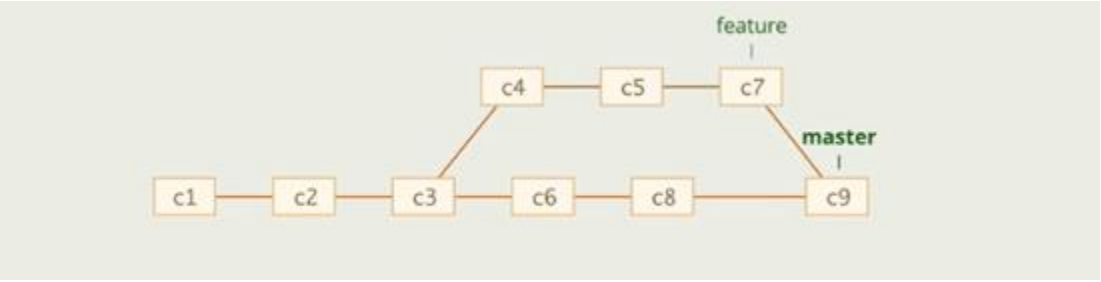
Практика – работа с ветками

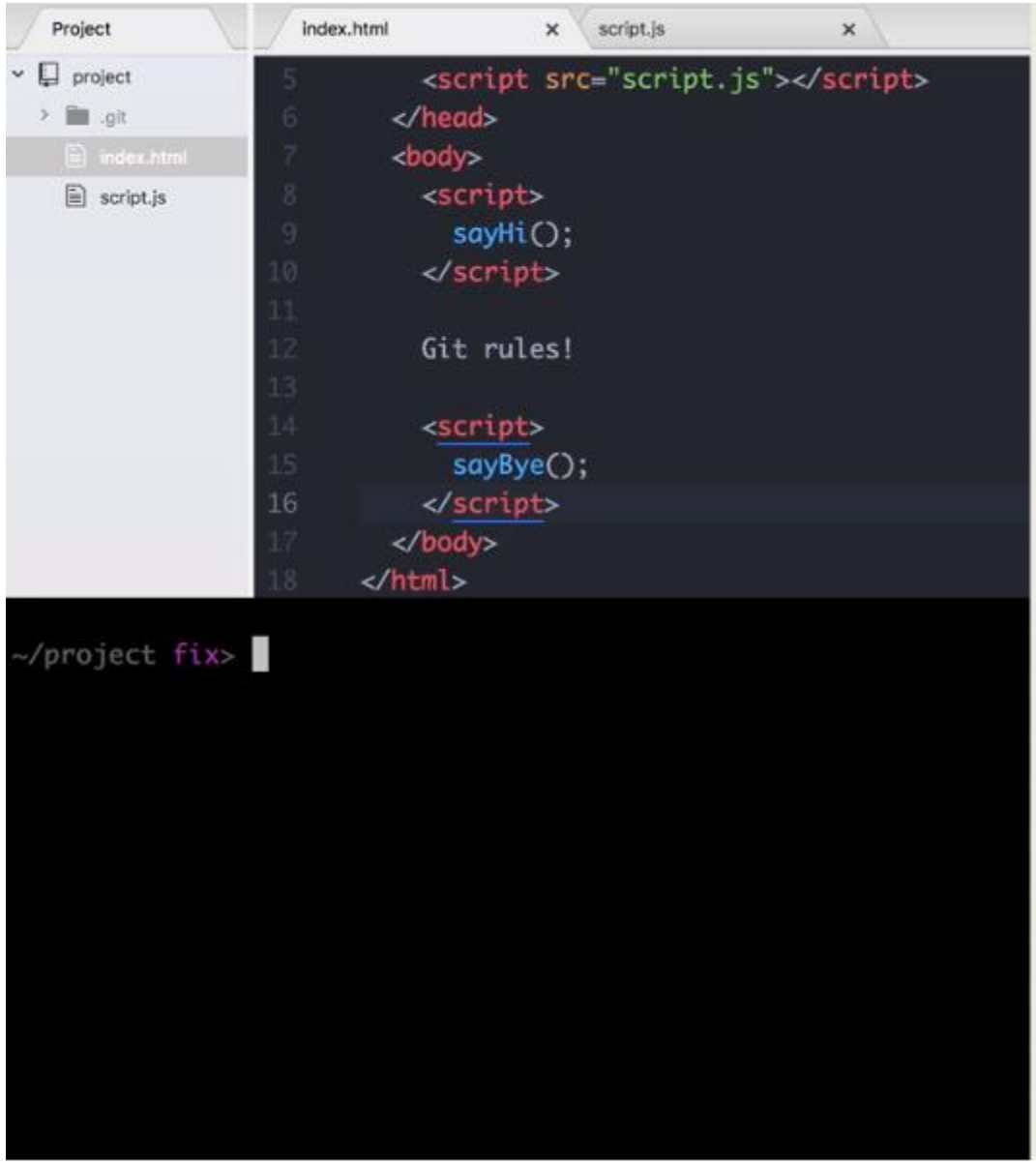
- Создание локальной ветки (branch)
- Переключение на локальную ветку (checkout)
- Переключение на удаленную (серверную) ветку и создание локальной
- Создание ветки под feature

Практика - ветки

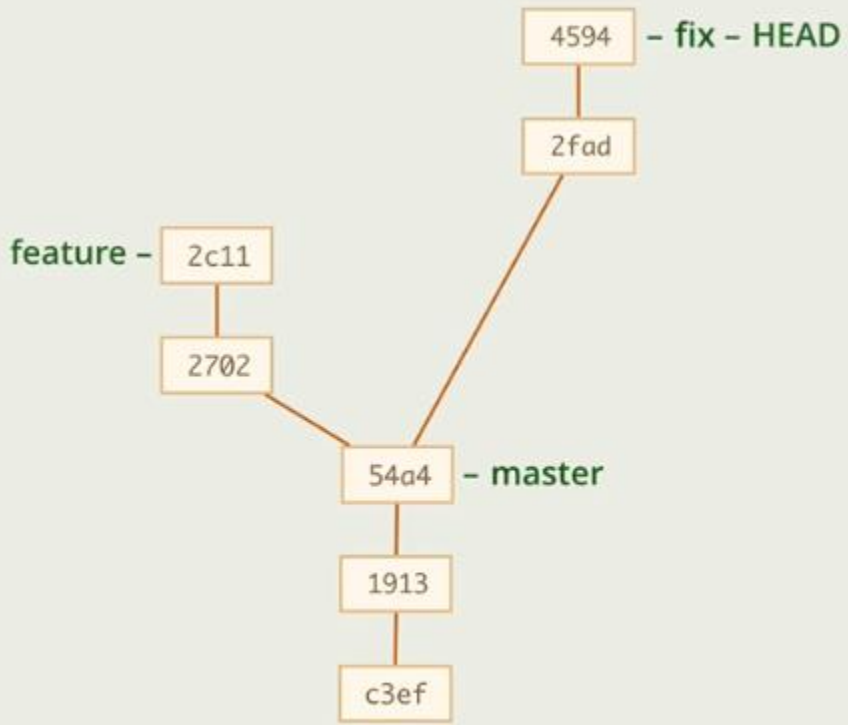
Слияние веток

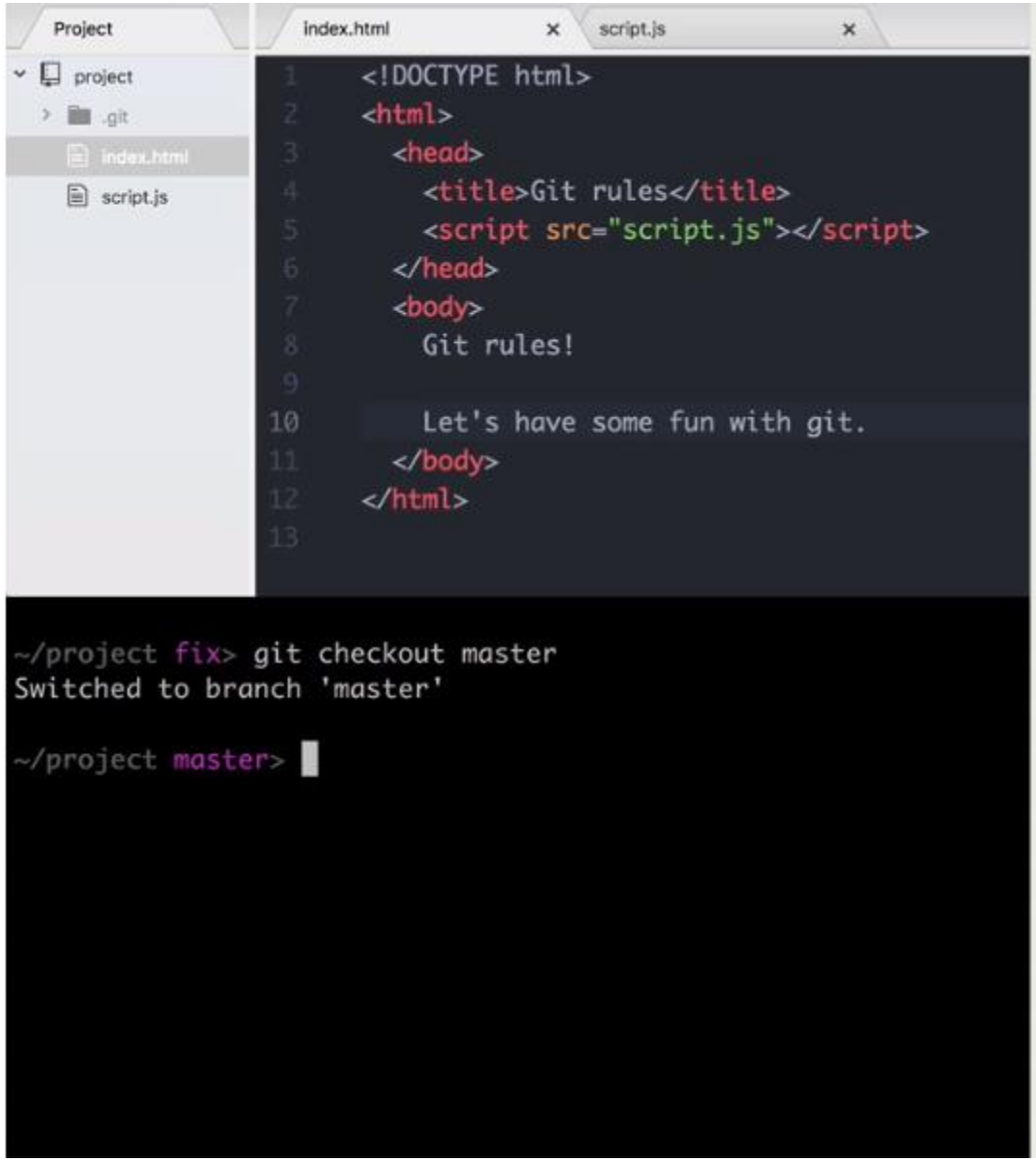
- В результате все равно необходимо сливать feature-ветку в основную ветку
  - Когда работа над feature завершена и все оттестировано
  - Например, вливаем изменения обратно в dev ветку
- Как правило, слияние происходит с той же веткой, от которой отпочковались
- Самый простой метод слияния – fast-forward





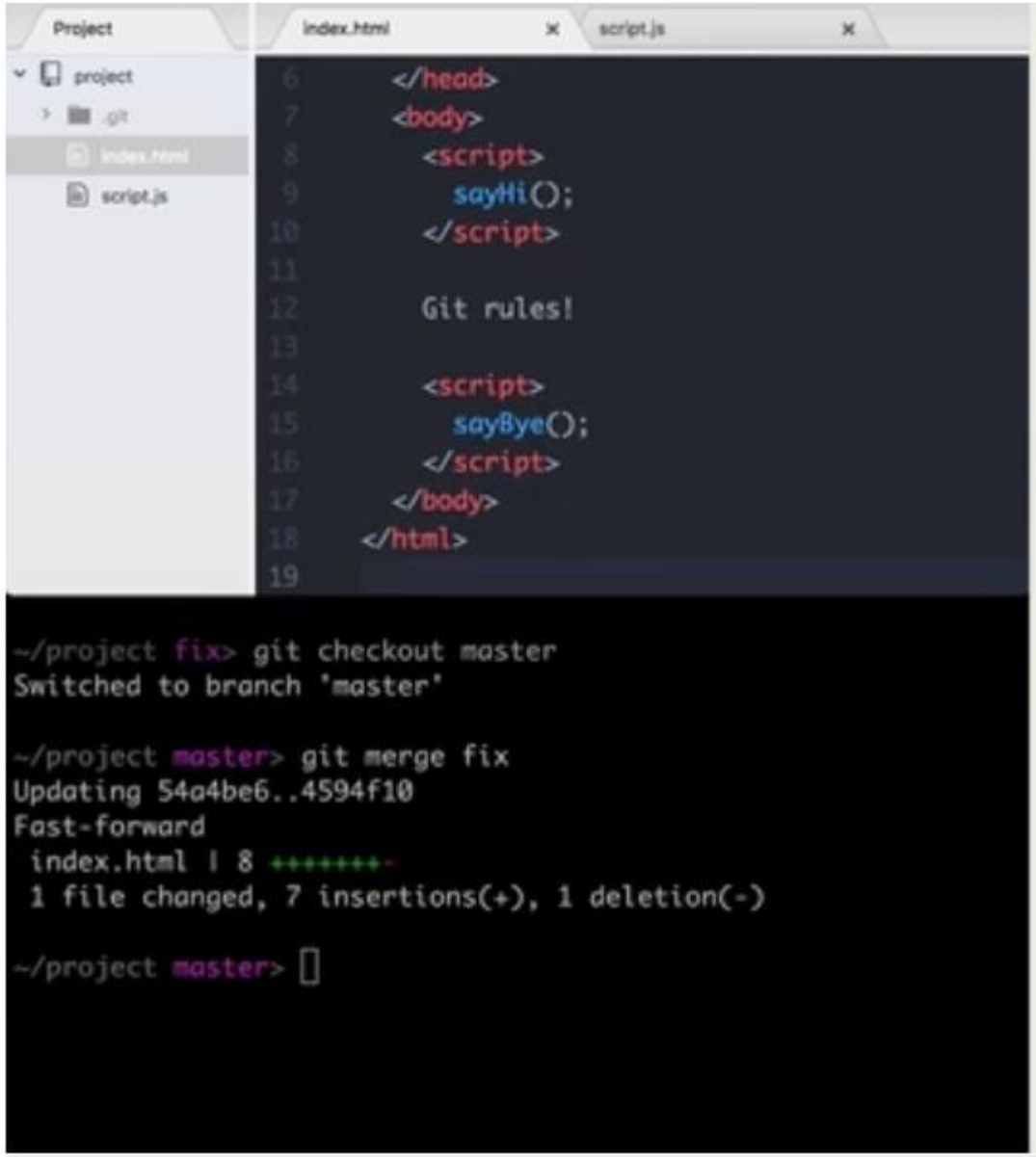
Мы собираемся добавить изменение, которое сделано в ветке fix– в основную ветку master.





Сначала переключимся на ветку master

в git есть команда git merge для слияния веток



Параметром команды merge - указываем, из какой ветки мы хотим добавить изменения

Project

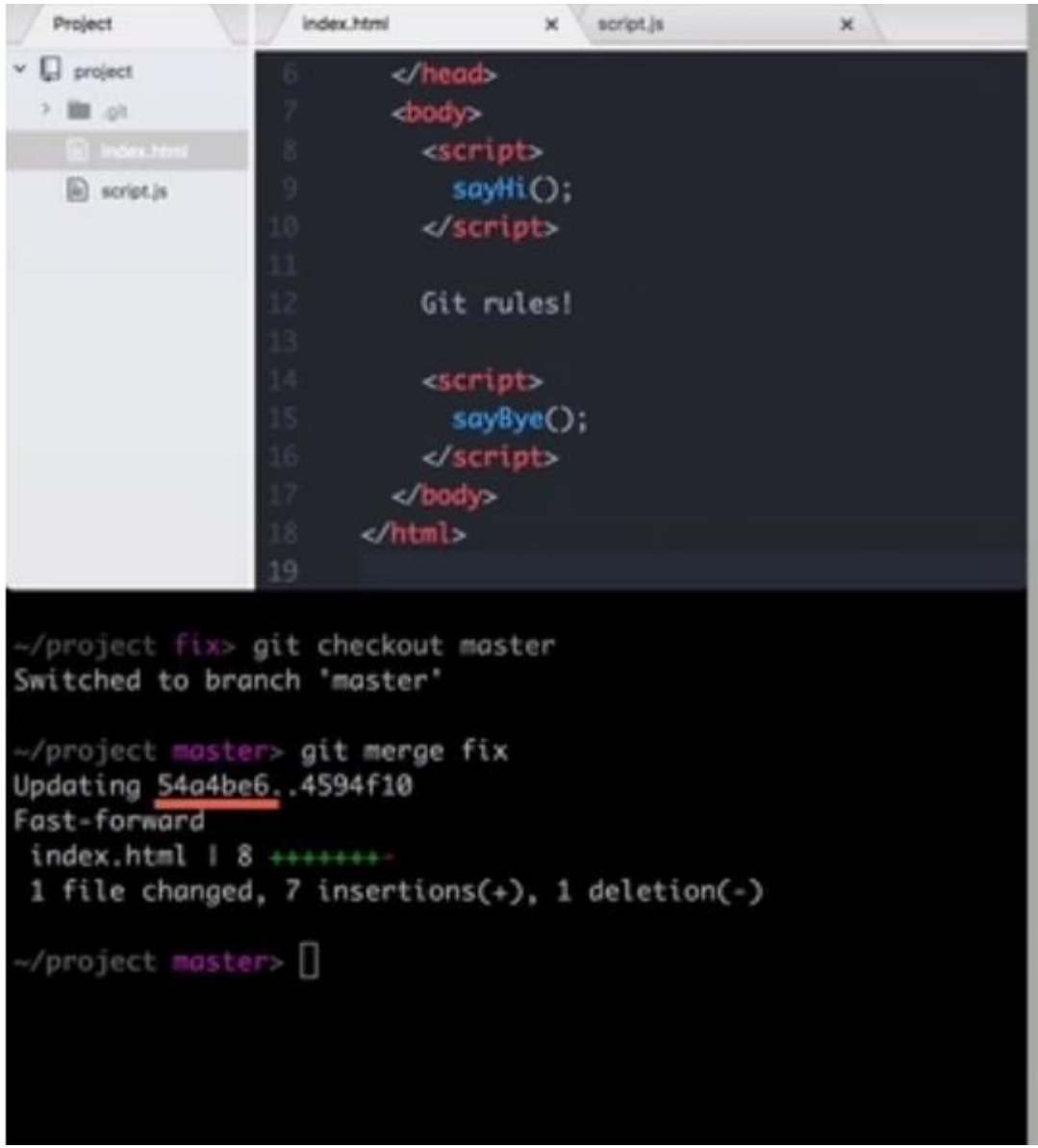
index.html × script.js ×

```
project
  .git
  index.html
  script.js
```

```
6       </head>
7       <body>
8         <script>
9           sayHi();
10        </script>
11
12        Git rules!
13
14        <script>
15          sayBye();
16        </script>
17      </body>
18    </html>
19
```

```
~/project fix> git checkout master
Switched to branch 'master'

~/project master> git merge fix
Updating 54a4be6..4594f10
Fast-forward
 index.html | 8 +++++++-
 1 file changed, 7 insertions(+), 1 deletion(-)

~/project master>
```

Посмотрим, что отображается в консоли при merge



Первый коммит - вершина ветки master

Project

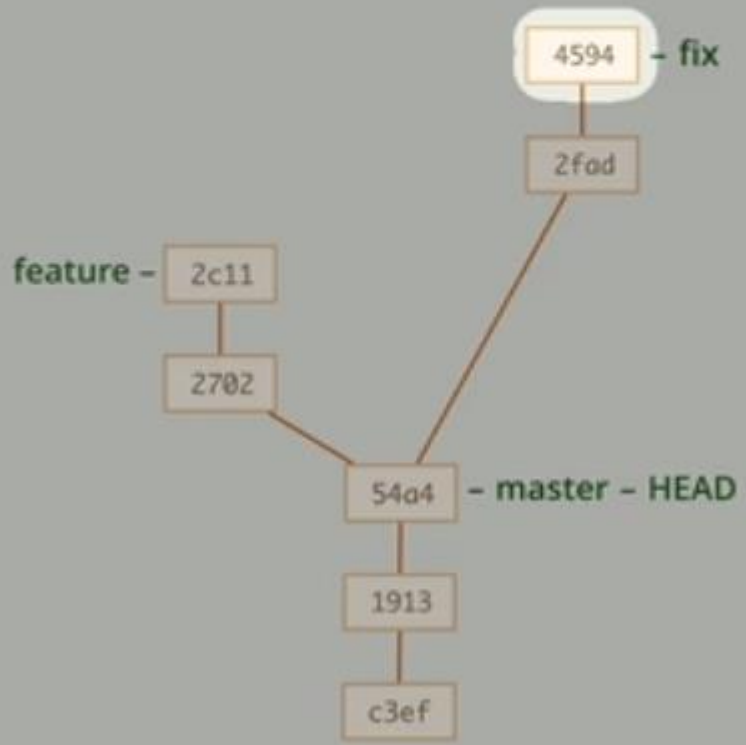
index.html × script.js ×

```
project
  .git
  index.html
  script.js
```

```
6       </head>
7       <body>
8         <script>
9           sayHi();
10        </script>
11
12        Git rules!
13
14        <script>
15          sayBye();
16        </script>
17      </body>
18    </html>
19
```

```
~/project fix> git checkout master
Switched to branch 'master'

~/project master> git merge fix
Updating 54a4be6..4594f10
Fast-forward
 index.html | 8 +++++++-
 1 file changed, 7 insertions(+), 1 deletion(-)

~/project master>
```

Второй коммит – последний коммит в ветке fix



Project

index.html × script.js ×

```
project
  .git
  index.html
  script.js
```

```
6       </head>
7       <body>
8         <script>
9           sayHi();
10        </script>
11
12        Git rules!
13
14        <script>
15          sayBye();
16        </script>
17      </body>
18    </html>
19
```

```
~/project fix> git checkout master
Switched to branch 'master'

~/project master> git merge fix
Updating 54a4be6..4594f10
Fast-forward
 index.html | 8 ++++++++
 1 file changed, 7 insertions(+), 1 deletion(-)

~/project master>
```

Fast forward – один из самых простых алгоритмов слияния.

Project

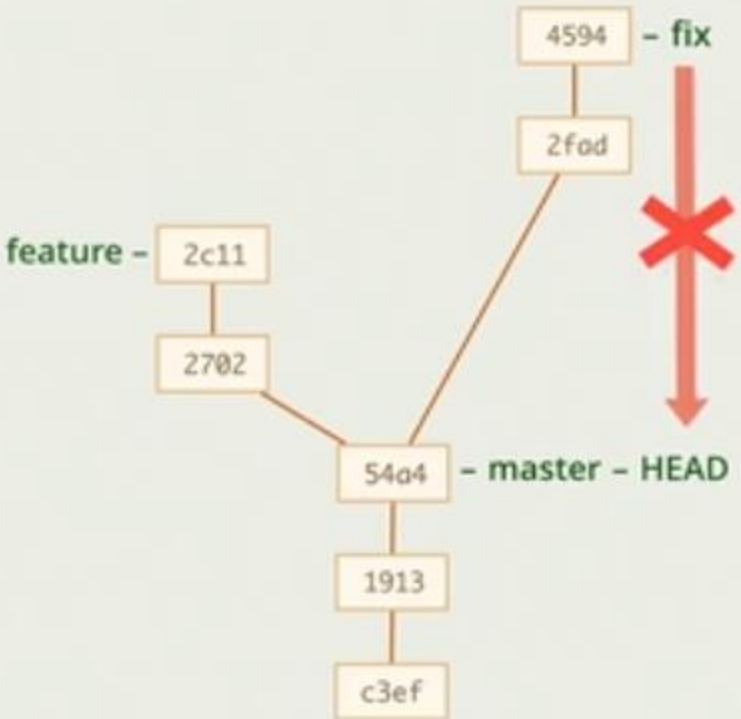
index.html × script.js ×

```
project
  .git
  index.html
  script.js
```

```
6       </head>
7       <body>
8         <script>
9           sayHi();
10        </script>
11
12        Git rules!
13
14        <script>
15          sayBye();
16        </script>
17      </body>
18    </html>
19
```

```
~/project fix> git checkout master
Switched to branch 'master'

~/project master> git merge fix
Updating 54a4be6..4594f10
Fast-forward
 index.html | 8 ++++++++-
 1 file changed, 7 insertions(+), 1 deletion(-)

~/project master>
```

изменения из ветки fix не добавляются в master



Project

index.html × script.js ×

```
project
  .git
  index.html
  script.js
```

```
6       </head>
7       <body>
8         <script>
9           sayHi();
10        </script>
11
12        Git rules!
13
14        <script>
15          sayBye();
16        </script>
17      </body>
18    </html>
19
```

```
~/project fix> git checkout master
Switched to branch 'master'

~/project master> git merge fix
Updating 54a4be6..4594f10
Fast-forward
 index.html | 8 ++++++++-
 1 file changed, 7 insertions(+), 1 deletion(-)

~/project master>
```

указатель master передвигается так, чтобы указывать на тот же коммит, что и ветка fix.

Project

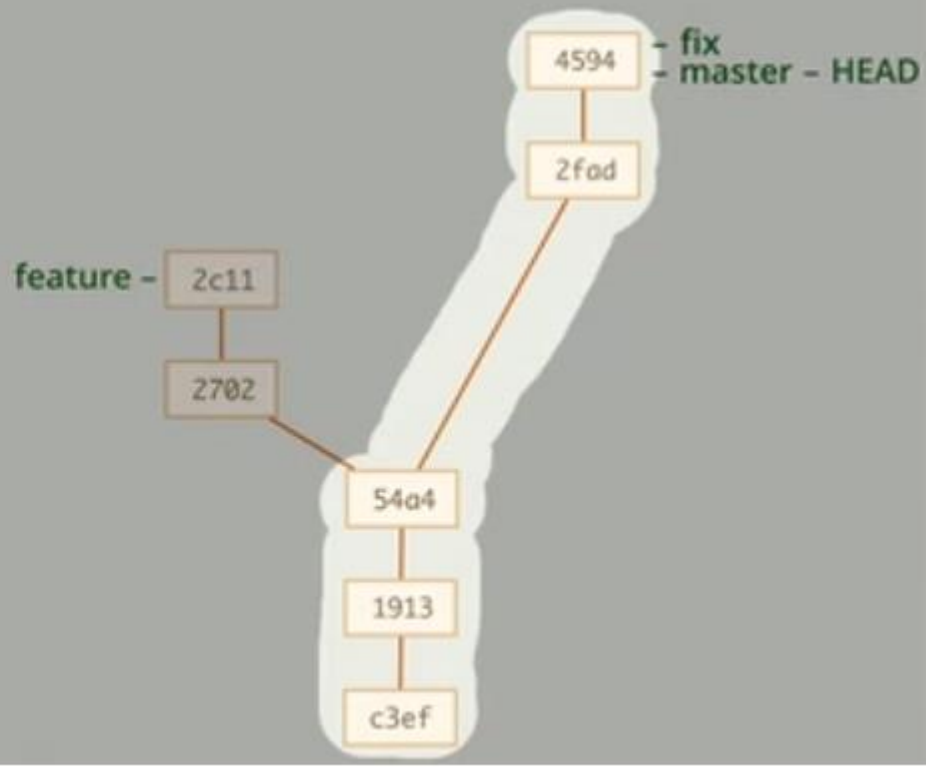
index.html × script.js ×

```
project
  .git
  index.html
  script.js
```

```
</head>
<body>
  <script>
    sayHi();
  </script>

  Git rules!

  <script>
    sayBye();
  </script>
</body>
</html>
```

```
~/project fix> git checkout master
Switched to branch 'master'

~/project master> git merge fix
Updating 54a4be6..4594f10
Fast-forward
 index.html | 8 +++++++-
 1 file changed, 7 insertions(+), 1 deletion(-)

~/project master>
```

Результат после переноса ветки master



Project

index.html

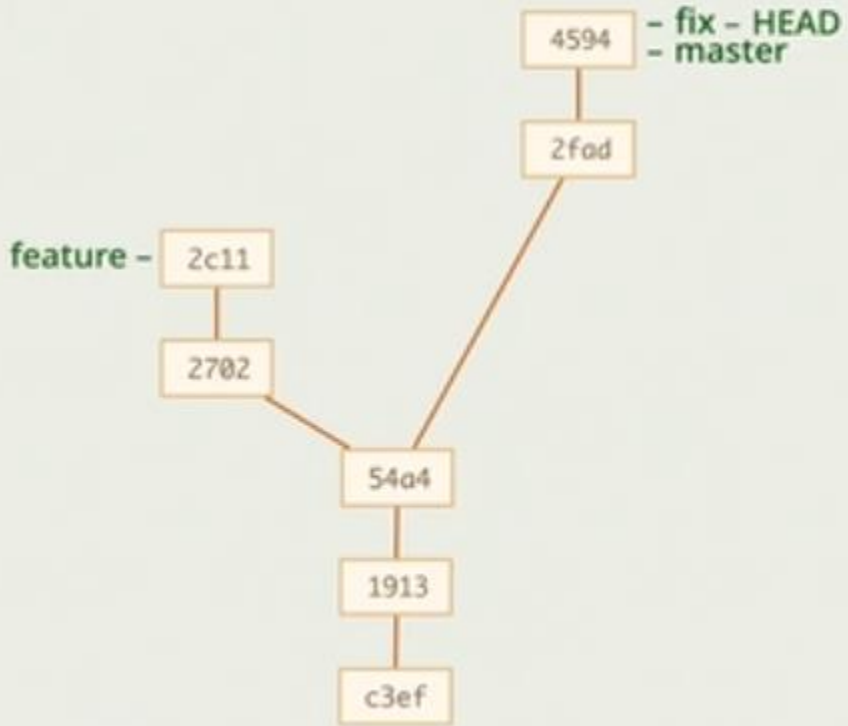
script.js

```
project
  .git
  index.html
  script.js
```

```
</head>
<body>
  <script>
    sayHi();
  </script>

  Git rules!

  <script>
    sayBye();
  </script>
</body>
</html>
```

```
~/project fix> git checkout master
Switched to branch 'master'

~/project master> git merge fix
Updating 54a4be6..4594f10
Fast-forward
 index.html | 8 +++++++-
 1 file changed, 7 insertions(+), 1 deletion(-)

~/project master> git checkout fix
Switched to branch 'fix'

~/project fix>
```

Теперь master и fix указывают на один и тот же коммит



Практика – работа с ветками

- Слияние веток – merge
  - Слияние с перемоткой
- Сливаем в ту же ветку, от которой отпочковались
- Как правильно работать, чтобы ветка не устаревала
  - Посмотрим на примере – слияние в свою ветку прежде чем выливать из нее изменения в исходную

Практика – работа с ветками

- Создаем development ветку
- Создаем feature1 ветку
- Вносим изменения в feature ветку
- Merge feature в development (локально)
- Merge development в master (локально)

Защищенные ветки

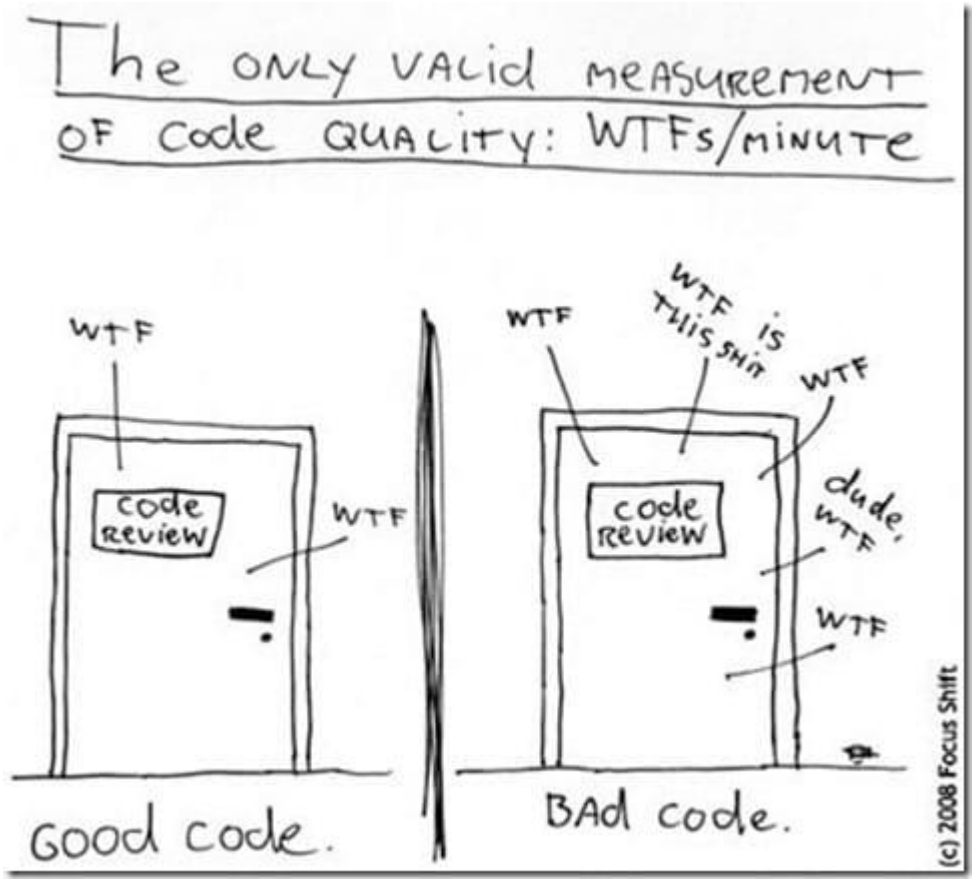
- В них нельзя напрямую push изменения (картинка)
- Только опытные разработчики могут делать merge в такие ветки
- Обычно это следующие ветки
  - Master – в ней может хранится продуктовая версия
  - Development – в ней хранится текущая рабочая версия

Защищенные ветки

- Как же попадают в проект изменения остальных членов команды?
- В gitlab используются следующие механизмы
  - Merge request
  - Code review & approve
- Т.е. слияние происходит на сервере

Code review

- Создается ветка для разработки feature
- В нее добавляются коммиты по мере разработки
- В конце разработки она заливается на сервер
- Делается merge request в develop с назначением старшему разработчику
- Опционально, запрашивается approve от других пользователей
- Дальше происходит обсуждение и принятие/непринятие кода



Практика – merge request

- Создаем feature2 ветку
- Вносим изменения в feature ветку
- Пушим ее на сервер
- Делаем merge-request в development ветку – можем назначить reviewer
- От лица reviewer смотрим запрос на слияние
  - Можем отправить на доработку, если нужно
  - Можем сделать approve
- В итоге, делаем merge
- Изменения попали в рабочую ветку

Практика – автоматически разрешимый конфликт

- Делаем ветку feature3.1
  - Вносим изменения в блок 1, commit
  - Делаем push ветки на сервер
- Делаем ветку feature3.2
  - Вносим изменения в блок 2, commit
  - Делаем push ветки на сервер
- У нас в двух ветках одновременно копятся изменения

Практика – автоматически разрешимый конфликт

- Делаем merge-request ветки feature3.1 в develop, принимаем merge
- Делаем merge-request ветки feature3.2 в develop, видим сообщение о проблеме
  - Проблема в том, что ветка develop «ушла вперед» с того момента, как была отпочкована feature3.2
  - Однако реально конфликта кода нет
- Обновимся из рабочей ветки
  - Нужно делать это время от времени, или чаще вливать свои коммиты
  - Переключимся на базовую ветку, обновимся
  - Вольем изменения из нее в свою ветку

Практика – автоматически разрешимый конфликт

- Делаем merge из ветки develop в feature3.2
  - Конфликтов нет – делаем push
- Смотрим «висящий» merge request – он обновился, в него добавился коммит «слияния» веток development и feature3.2
- Поскольку feature3.2 теперь обновился до актуальной версии development, нет больше препятствий обратному вливанию ветки 3.2 в develop