Краткое введение в Git

Колян Ш.

Версия 1.3.4

12 августа 2015 г.

Содержание

1 Начало работы

1.1 Регистрация в Gitorious

1.1.1 Создание нового пользователя

Для регистрации нужно перейти по ссылке https://git.insysltd.ru/users/new. Будет предложено ввести логин, Email и пароль. Эти данные требуются исключительно для доступа к репозиториям и проектам Gitorious посредством Web-интерфейса. Вместо этих данных возможно использование OpenID (http://ru.wikipedia.org/wiki/OpenID).

1.1.2 Подтверждение регистрации

Для подтверждения вновь созданной учётной записи необходимо обратиться к любому из администраторов ресурса. Данная необходимость обусловлена защитой от несанкционированных регистраций со стороны калифорнийских школьников.

1.1.3 Загрузка OpenSSH-ключа

Данный ключ требуется для защищённой передачи данных между Git-клиентом и Git-сервером.
Сгенерировать пару ключей (приватный/общедоступный) можно посредством команды

```
ssh-keygen -t rsa
```

из пакета OpenSSH http://www.openssh.org/.

Установочный файл для Windows может быть найден по ссылке http://sourceforge.net/projects/sshwindows/files/OpenSSH%20for%20Windows%20-%20Release/. Полученный приватный ключ id_rsa следует хранить в безопасном от чужих глаз месте (в случае получения доступа к нему 3-их лиц следует сгенерировать новый, а старый публичный ключ (.pub) - удалить с Git-сервера).

Загрузка публичного ключа id_rsa.pub выполняется в Web-интерфейсе профиля пользователя Gitorious ▷ Dashboard ▷ Manage SSH keys ▷ Add SSH Key.

1.2 Создание нового репозитория

Если репозиторий добавляется в уже существующий проект, то на вкладке проекта в Web-интерфейсе Gitorious достаточно нажать кнопку Add Repository, ввести имя и описание.

Если требуется создать новый проект, то в Web-интерфейсе Gitorious нужно выбрать Gitorious ▷ Projects ▷ Create a new project, ввести обязательные поля Title(имя проекта), Slug(идентификатор, генерируется автоматически), Description(описание). При необходимости могут быть заданы Labels(метки для быстрого поиска), Owner(владелец проекта), License(лицензия), Home URL(домашняя страница проекта), Mailinglist URL(список рассылки), Bugtracker URL(баг-трекер). После нажатия кнопки Create project будет создан новый проект с указанными параметрами и предложено создать в нём первый репозиторий.

В Git имеется встроенная справка по всем командам, достаточно набрать

```
git help command
```

Для большинства опций имеются их сокращённые аналоги, например,

```
git rebase -i -p # эквивалентно git rebase --interactive --preserve-merges
```

и значения по умолчанию:

```
git log --decorate # эквивалентно git log --decorate=short
```

1.2.1 Первый коммит

Хорошей практикой считается при создании репозитория включение первым делом в него файла README с пусть даже минимальным описанием того, для чего он создан, какую цель преследует.

```
echo ‘‘This is my project for tests.’’ > README
git add README
git commit -m ‘‘Initial commit.’’
```

- Здесь мы создали файл с именем README;
- Добавили этот файл в индекс Git для включения его в ближайший коммит;
- Сделали коммит “Initial commit.”.

Индексом в Git называется промежуточное хранилище изменений, попадающих в ближайший коммит. Таким образом, Git даёт возможность избирательно включать те или иные изменения вплоть до отдельных частей файла. Пример:

```
git add -p git-tutorial.lyx
diff --git a/git-tutorial.lyx b/git-tutorial.lyx
index 8c6fa34..3aaafde 100644
--- a/git-tutorial.lyx
+++ b/git-tutorial.lyx
@@ -365,7 +365,10 @@ Initial commit.
\end_layout    \begin_layout Standard -В качестве
+Индексом в Git называется промежуточное хранилище изменений, попадающих
+ в ближайший коммит.
+ Таким образом, Git даёт возможность избирательно включать те или иные изменения
+ вплоть до отдельных частей файла.
\end_layout
\begin_layout Subsubsection
Stage this hunk [y,n,q,a,d,/,e,?]? y
```

Параметр -m у команды commit указывает на то, что следующим аргументом команды следует описание коммита, что удобно, когда нужно коммит сделать быстро. Без указания этого параметра откроется окно редактора, куда требуется внести описание изменений.

Также, у команды commit помимо параметра -m можно указывать параметр -a, чтобы в коммит добавились все изменения, в том числе и не включенные в индекс. Таким образом, последовательность команд

```
vim main.c # editing staged C-file...
vim main.h # editing staged H-file...
git commit -am ‘‘Both source and header files are changed.’’
```

эквивалентна

```
vim main.c # editing staged C-file...
vim main.h # editing staged H-file...
git add main.c main.h
git commit -m ‘‘Both source and header files are changed.’’
```

1.2.2 Статус текущих “незакоммиченных” изменений

Текущее состояние рабочей директории можно посмотреть командой status:

```
$ git status    # вывести текущие изменения
# On branch #238feature
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   git-tutorial.lyx
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       #git-tutorial.lyx#
```

```
#       git-tutorial.lyx~
#       git-tutorial.pdf
no changes added to commit (use "git add" and/or "git commit -a")
$ git status -s # использовать сокращённую запись
 M git-tutorial.lyx
?? #git-tutorial.lyx#
?? git-tutorial.lyx~
?? git-tutorial.pdf
```

Сами изменения можно посмотреть командой diff:

```
git diff          # вывести изменения рабочей директории
git diff --staged # вывести изменения, добавленные в индекс
```

1.3 Синхронизация с Git-сервером

1.3.1 Добавление удалённого репозитория

Для синхронизации с Git-сервером необходимо его добавить в список remote:

```
git remote add origin git@git.insysltd.ru:test_project/test_repo.git
```

origin является удалённым репозиторием по умолчанию при выполнении команд синхронизации и может быть опущен.

Нормальной практикой является добавление нескольких удалённых веток, например:

```
git remote add origin git@git.insysltd.ru:test_project/test_repo.git
git remote add github git@github.com:test_repo.git
git remote add usb /media/usb_stick/projects/test_repo.git
```

1.3.2 Принятие изменений, сделанных другими пользователями

В отличие от других систем управления исходным кодом Git работает со слияниями гибче. Поддерживаются так называемые fast-forward слияния (рис. 1.1).

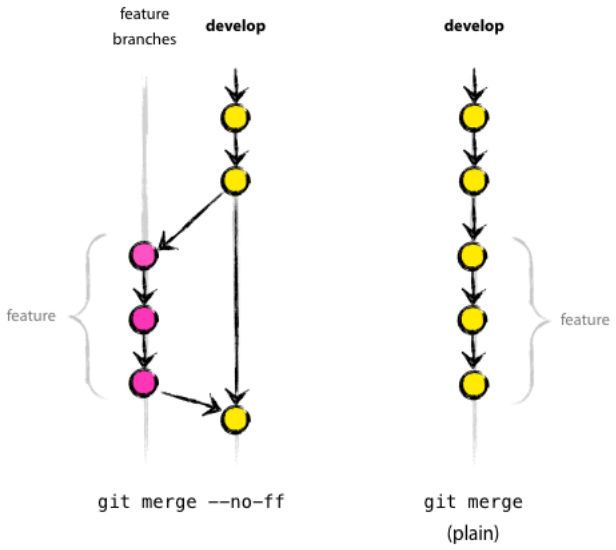

Рис. 1.1: fast-forward слияния

При выполнении команды git pull скачиваются удалённые изменения (git fetch) и выполняется слияние (git merge). При указании опции –rebase коммиты из удалённого репозитория займут своё место перед локальными коммитами - теми, которые ещё не были отправлены на Git-сервер.

```
git pull --rebase
```

Это даёт возможность пользователям работать, не мешая друг-другу, выполняя при этом коммиты тогда, когда хочется пользователю, а не тогда, когда доступен главный репозиторий.

Иногда более целесообразно указывать, из какой удалённо ветки требуется произвести слияние коммитов:

```
git pull --rebase origin develop
```

1.3.3 Слияние веток

Как правило, слияние веток необходимо по завершении той или иной фичи (от англ. feature) или исправления бага (от англ. bug). Приведём возможный сценарий для реализации некоторой возможности featureA.

```
git checkout -b featureA develop # ответвление featureA от develop
... edit ... commit ...
... edit ... commit ...
... edit ... commit ...
git reset --soft develop # вместо develop может быть хеш коммита
git commit -m ‘‘Feature A implemented.’’ # получение из нескольких коммитов одного
git checkout develop # возврат в ветку разработки develop
git merge --no-ff featureA # слияние без --no-ff, если предпочитаете линейную историю
git branch -d featureA # удаление ‘‘слитой’’ ветки featureA
```

При работе с системами управления проектами (ChilliProject, Redmine, Trac) по завершении работы с ветвью имеет смысл добавлять в сообщение коммита отметку о решении/закрытии бага/фичи. Система управления проектами автоматически пометит тикет как решённый и свяжет его с указанным коммитом, так что любой может определить какие изменения были внесены для решения конкретной задачи. Пример:

```
git commit -m ‘‘Feature A implemented. Closes #238.’’
git commit -m ‘‘Bug B fixed. Fixes #239.’’
```

Также возможно слияние изменений без создания нового коммита (следует отметить, что хеш последнего коммита всё-равно изменится, а с ним изменится и история) посредством команды:

```
git merge --squash
```

Иногда в списке удалённых веток (git branch -r) остаются несуществующие ссылки в результате того, что кто-то удалил эту ветвь или несколько веток ссылок указывали на одну удалённую ветвь. Обновить список можно командой fetch:

```
git fetch
git fetch github
```

или вручную удалить невалидную ссылку:

```
git branch -d -r origin/invalid_branch
git branch -d -r github/invalid_branch2
```

1.3.4 Отправка изменений на сервер

Для отправки локальных изменений из текущей ветки в репозиторий по умолчанию (origin) необходимо выполнить команду

```
git push
```

Также возможны более сложные операции, к примеру:

```
git push github # отправит изменения всех веток, которые есть в локальном
                # и удалённом (в данном случае GitHub) репозиториях
git push origin featureA # отправит в origin изменения в ветке featureA
git push origin localA:featureA # отправит в origin изменения локальной
                                # ветки localA в удалённую featureA
git push origin master develop featureB # отправит 3 указанные ветки
git push origin :featureC # удалит в удалённом репозитории ветку featureC
git push origin :v1.0.3 # удалит в origin тег v1.0.3
```

1.3.5 Права доступа в Gitorious

В Web-интерфейсе Gitorious на вкладке управления проектом есть кнопка Manage Access ▷ Make private, позволяющая сделать репозиторий "закрытым от посторонних глаз" и разрешить выборочный доступ на чтение для отдельных лиц или групп. В этом случае все репозитории, входящие в данный проект также становятся недоступными всем лицам, кроме одобренных одним из владельцев проекта/репозитория.

Для репозитория, как и для проекта, также возможна установка разрешений. Для этого есть кнопка Manage Read Access ▷ Make private.

Для возможности внесения изменений в репозиторий несколькими разработчиками требуется добавить этих разработчиков/группы в список разрешения Manage collaborators ▷ Add collaborators и установить для добавленных пользователей/групп их разрешения: Review, Commit, Administer.

1.4 Работа с чужим репозиторием, pull-requests

Политика распределённого хранилища подразумевает возможность независимой работы с клонами без вмешательства в репозитории других пользователей.

Как правило, у отдельно взятого репозитория имеется один или несколько коммитеров, которым разрешён полный доступ и на ком лежит ответственность за поддержание данного репозитория в рабочем состоянии. Все остальные, кто имеет доступ на чтение данного репозитория реализуют фичи (feature) в своих клонах. По завершении работы над фичей отправляется pull-request разработчику основного репозитория, который осуществляет слияние, когда у него есть на то свободное время. Таким образом разработчики делают каждый свою работу, не мешая друг другу.

Приведём пример такой работы.

1.4.1 Получение информации о репозитории

Общая информация о репозитории может быть получена уже из Web-интерфейса Gitorious, включая ветки, список коммитов, граф коммитов, pull-реквесты и т.п.

1.4.2 Авторство и время редактирования строк

Следующий пример покажет авторство, ревизии последних правок и время для нескольких строк в файле исходного кода main.c:

```
git blame -L 40,60 main.c   # вывести информацию о строках с 40 по 60 включительно
git blame -L 40,+21 main.c  # вывести информацию о 21-ой строке, начиная с 40-ой
```

Если требуется отфильтровать слишком старую историю, можно это сделать следующим образом:

```
git blame v2.6.18.. -- main.c        # игнорировать информацию, старше версии 2.6.18
git blame --since=3.weeks -- main.c  # игнорировать информацию, старше 3-х недель
```

Вот пример, как можно посчитать общий вклад разработчиков в конкретный файл проекта:

```
git blame --line-porcelain mainform.cpp | sed -n 's/^author //p' | sort | uniq -c | sort -rn
   1526 Kolan Sh
    691 egor_i@egor-837.insysltd.ru
    167 egor_i@EGOR-837
```

1.4.3 Клонирование

Более подробная информация, включая поиск по коммитам возможна после клонирования репозитория. Если Вы не хотите отвлекать от работы владельца репозитория, но при этом хотите коммитить, будет удобно сделать удалённый клон в Gitorious (кнопка Clone repository у любого проекта в Web-интерфейсе). Будет предложено ввести имя клона, которое будет уже заполнено по умолчанию (в начало), так что его можно не менять.

Далее выполняется стандартная процедура клонирования удалённого клона на локальный диск:

```
git clone git@git.insysltd.ru:~user1/test_project/user1-test_repo.git
```

Если репозиторий содержит субрепозитории (submodules) необходимо их инициализировать и синхронизировать:

```
git submodule update --init --recursive
```

1.4.4 pull-request - запрос на слияние

По завершении работы (реализации фичи или устранения бага) нужно залить в свой удалённый клон, при необходимости, обновив его (git pull –rebase) из исходного репозитория.

Далее в Web-интерфейсе клона нажать кнопку Request merge, добавить описание pull-запроса, указать исходный репозиторий и ветки. Ваш запрос на слияние появится в списке входящих сообщений пользователя, владеющего оригиналом репозитория.

1.4.5 format-patch - создание патча

Когда лень делать удалённый клон в Gitorious с pull-request-ом и есть уверенность, что Ваш патч примут сразу, а не спустя длительное время, когда в основном дереве накопится изменений, конфликтующих с патчем, можно использовать быстрое создание патча:

```
git checkout -b fix_the_bug master # создаём ответвление от master
... edit ...
git commit -am ‘‘The bug fixed.’’
git format-patch master # создать патч, включающий отличия текущей ревизии от ветки master
```

Далее полученный патч может быть отправлен по email или прямо в окно Jabber-клиента. Можно создать патч и отправить его по email одной командой:

```
git format-patch master --stdout | mail -s ’Please apply this patch, Leo’ leo@matrix.org
```

Применить полученный патч можно командой apply:

```
git apply --check the_bug_fixed.patch # тест на конфликты
git apply --stat # применение патча с включением в историю текущей ветви
                 # и выводом статистики изменений
```

1.5 Анализ истории, сравнение, поиск и фильтрация

1.5.1 Список ветвей

Выводится командой branch

```
git branch    # список локальных веток
git branch -r # список удалённых веток
git branch -a # список всех веток, включая удалённые
```

За более полным списком опций следует обратиться к справке по команде branch (git help branch).

1.5.2 Лог истории

Наболее полную информацию об истории изменений можно получить командой

```
git log --graph --decorate --stat
```

- Git выведет графическое изображение дерева истории, включая слияния, подписи и статистику изменений по каждому коммиту, где:

- --graph – выводить графическое дерево;
- --decorate – выводить у коммитов имена ссылок (ref names), например master, origin/develop;
- --stat – печатать статистику по изменениям в каждом файле.

Если нужно вывести лог на определённом временном промежутке, через две точки вводится начало и конец:

```
git log master..develop
git log f8a32c..3ab98c
```

Можно исключать из выводимого списка коммиты, входящие в какую-либо ветвь, например следующая команда выведет список коммитов из develop, не вошедшие в master:

```
git log develop ^master
```

Более полный список опций описан в руководстве (git help log).

1.5.3 Сравнение ревизий

Сравнение производится командой git diff. Примеры:

```
git diff          # показать текущие изменения в рабочей директории
git diff --staged # показать текущие изменения в рабочей директории, добавленные в индекс
git diff master..develop main.c # показать разницу main.c между двумя ветвями
git diff 581a32..8f292a --stat  # показать статистику различий между ревизиями
```

1.5.4 Поиск и фильтрация

Для поиска используется всё та же команда log, но с параметрами:

```
git log --author=''Vasya Pupkin''      # поиск по автору
git log --author="\(Adam\)\|\(Jon\)" # поиск по одному из авторов
git log -Ssome_string --source --all # поиск по строке some_string с выводом ссылок на
                                     # удалённые репозитории, из которых достижима ревизия
git log path/to/file                 # коммиты, затрагивающие данный файл
git log --grep=feature               # коммиты с сообщениями о фичах
```

2 Удобство в работе

2.1 Индекс, откат, временное хранилище

В Git имеются команды, делающие работу более удобной. Так, в частности, можно спрятать все незакомиченные изменения рабочей директории и достать их в любой момент обратно:

```
git stash                  # спрятать изменения
git stash pop              # применить их к рабочей директории и удалить их из stash-списка
git stash --keep-index     # спрятать изменения, не добавленные в индекс
git stash drop             # ''выбросить'' спрятанные изменения
git stash save --keep-index # спрятать изменения, не добавленные в индекс
git stash apply            # применить спрятанные изменения, не удаляя их из stash-списка
```

Можно получить любую версию файла или директории посредством команды checkout:

```
git checkout README         # получить README-файл, отменив изменения после последнего коммита
git checkout main.c 3f89a7f # получить main.c из ревизии 3f89a7f
git checkout .              # получить из головы (HEAD) содержимое текущей директории
```

Можно отменить изменения, попавшие в индекс для следующего коммита:

```
git reset         # очистить индекс
git reset main.h  # убрать из индекса main.h
```

Вся история команд работы с Git хранится в локальном списке reflog:

```
$ git reflog
39b103b HEAD@{2}: commit: Adding feature #238...
8958552 HEAD@{3}: reset: moving to develop
7e3dca9 HEAD@{4}: commit: akt_half_tv3-117vm-00.lyx added.
8958552 HEAD@{5}: reset: moving to develop
```

так, что имеется возможность локального восстановления в истории случайно удалённого коммита. Можно, к примеру, откатить голову (HEAD) ветки develop к моменту, когда был добавлен lyx-файл:

```
git checkout develop
git reset --hard 7e3dca9
git push -f origin develop
```

или просто получить в рабочей директории прежнее состояние, не изменяя истории:

```
git checkout 7e3dca9
```

2.2 Теги и версии

Как правило, теги создаются для определённых версий проекта, но могут и не следовать этому правилу.

В случае создания новой версии проекта, хорошим стилем является обновление записи о версии внутри проекта, например version.h с последующим коммитом.

Приведём типичный пример:

```
git checkout -b release-1.0.0 develop
vim version.h # update version number
git commit -am ‘‘Bumped version number to 1.0.0’’
git checkout master
git merge --no-ff release-1.0.0
git tag v1.0.0
git checkout develop
git merge --no-ff release-1.0.0
git branch -d release-1.0.0
git push origin master develop --tags
```

2.3 Редактирование истории

2.3.1 Редактирование недавних коммитов

Изменение сообщения последнего коммита выполняется командой

```
git commit --amend
```

- откроется текстовый редактор, в котором следует отредактировать сообщение последнего коммита.

Для отмены последних нескольких коммитов в текущей ветви требуется выполнить команду

```
git reset --hard HEAD~3 # отменит последние 3 коммита
git push -f             # и удалит их с центрального репозитория
```

2.3.2 Редактирование коммита в глубинах истории

Для редактирования коммита, совершённого на более ранних этапах следует прибегнуть к команде rebase:

```
git rebase bbc643cd^ --interactive --preserve-merges # редактировать коммит bbc643cd
        # --preserve-merges - пытаться сохранять точки слияния во время перестройки истории
vim main.c # редактирование main.c
vim main.h # редактирование main.h
git add main.c main.h # добавление main.c and main.h в индекс
git commit --amend # обновление коммита bbc643cd (теперь у него новый хеш)
git rebase --continue # обновить все последующие коммиты в данной ветке
git push -f # отправить изменённую ветвь на Git-сервер
```

Также, в некоторых случаях бывает необходимо отредактировать самый первый коммит (root-commit):

```
git rebase -i -p --root              # редактировать самый первый коммит
mkdir tmp && mv * tmp                # разобьём первый коммит на 2
touch README && git add README       # и добавим README-файл
git commit -am ‘‘Initial commit.’’
mv tmp/* . && rmdir tmp && git add . # добавим то, что было прежде 1-ым коммитом
git commit -m ‘‘2nd commit.’’         # во второй коммит.
```

Объединение нескольких коммитов в один можно выполнить ещё проще:

```
git rebase -i -p HEAD~5
# редактирование: заменить ’pick’ на ’squash’ для коммитов,
#                 которые хотим объединить, сохранить.
```

Разбить произвольный коммит на два последовательных коммита можно следующими командами:

```
git rebase -i <commit-to-split>^ branch_name
git rebase HEAD^
git add -p                    # add changes for the 1th commit
git commit -m ‘‘1st commit’’  # make 1st commit
git commit -am ‘‘2nd commit’’ # make 2nd commit
git rebase --continue         # complete rebase operation
```

2.3.3 Выделение поддиректории в отдельный репозиторий

Информация взята со stackoverflow.com.
Предположим, имеется репозиторий со структурой директорий

```
XYZ/

    .git/
    XY1/
    ABC/
    XY2/
```

и требуется его разбить на 2 репозитория, чтобы истории их не пересекались:

```
XYZ/

    .git/
    XY1/
    XY2/

ABC/
```

```
.git/
ABC/
```

Для этого воспользуемся командой filter-branch, изменяющей историю.

```
git clone XYZ ABC # клонируем XYZ в ABC
for branch in develop master; do git branch -t $branch origin/$branch; done # интересующие
                                                                             # нас ветви
git remote rm origin              # удаляем ссылку на родительский репозиторий,
                                  # чтобы обезопасить себя от изменений в нём
git filter-branch --index-filter "git rm -r -f --cached --ignore-unmatch XY1 XY2" \
     --prune-empty --tag-name-filter cat -- --all # удалить из истории дир-рии XY1 и XY2,
                                                  # оставив ABC
```

Аналогично создаётся репозиторий для XYZ

```
git clone XYZ XYZ-new && cd XYZ-new
for branch in develop master; do git branch -t $branch origin/$branch; done
git remote rm origin
git filter-branch --index-filter "git rm -r -f --cached --ignore-unmatch ABC" \
     --prune-empty --tag-name-filter cat -- --all
```

По опциям команды filter-branch смотрите документацию (git help filter-branch).

2.3.4 Удаление коммита из истории

Удаление последнего созданного коммита осуществляется одной командой и всегда завершается успехом.

```
git reset --hard HEAD~1 # удалить последний коммит
git reset --soft HEAD~1 # удалить последний коммит, сохранив изменения в ‘‘незакомиченными’’
```

Удаление коммита из глубин истории не всегда может быть выполнено успешно, может потребоваться разрешение конфликтов.

```
git rebase -i --preserve-merges HEAD~10 # редактировать последние 10 коммитов, сохраняя слияния
# В редакторе отметить удаляемый коммит для редактирования ‘‘edit’’
git reset --hard HEAD~1 # удалить требуемый коммит
git rebase --continue # продолжить перепостроение истории, начиная с удалённого коммита
```

2.3.5 Удаление тега в репозитории на сервере

Следующая команда удалит тег v0.1.2 на сервере:

```
git push origin :refs/tags/v0.1.2
```

Следует не забыть также удалить тег локально:

```
git tag -d v0.1.2
```

Иначе - при следующей отправки изменений он также попадёт на сервер.

2.3.6 Обновление указателя головы ветки на сервере без локальных изменений

Следующая команда обновит указатель головы на сервере:

```
git update-ref refs/heads/master 573f69d
```

Теперь голова ветки “master” в репозитории на сервере будет указывать на коммит с хешем 573f69d. Это бывает полезно, когда хочется поработать над частью локальных коммитов, отправляя невсе их сразу на сервер, а частями.

2.3.7 Синхронизация веток

По аналогии с командой git pull –rebase для принятия удалённых изменений с перемещением локальных коммитов на вершину истории синхронизация веток может осуществляться и локально.

```
# Синхронизировать текущую ветку с веткой branch1, переместив коммиты,
# отсутствующие в branch1 на вершину истории
git rebase branch1

# Синхронизировать текущую ветку с удалённой, переместив коммиты,
# отсутствующие в origin/master на вершину истории
git fetch origin master
git rebase origin/master
```

2.4 Пример использования gitflow

```
# Создание веток master/developer/release/hotfix
$ git flow init

# Начинаем работать над функционалом feature1 (ответвление от develop)
$ git flow feature start feature1
# делаем изменения
$ git add ...изменения...
$ git commit -m "изменения для feature1"

# Эта команда сделает слияние feature1 с develop и удалит ветку
$ git flow feature finish feature1

# Давайте начнём работу над релизом
$ git flow release start release1
# делаем изменения
$ git add ...изменения...
$ git commit -m "release1"

# Эта команда сделает слияние release1 с master
$ git flow release finish release1
```

2.5 Двоичный поиск ошибок

Выполняется командой bisect, пример:

```
git bisect start
git bisect bad         # текущая ревизия плохая
git bisect good v1.0.3 # в версии 1.0.3 проблема не наблюдалась
...
git bisect bad         # пометить текущую ревизию как плохую
git bisect good        # пометить текущую ревизию как хорошую
git bisect skip        # пропустить текущую ревизию
...
git bisect reset       # закончить двоичный поиск
```

Как только bisect нашёл источник ошибки - ревизию, в которой она была внесена, для нахождения ошибки остаётся проанализировать изменения в одном текущем коммите.

Дальнейшие действия зависят от вкусов/стиля разработки:

- ошибка может быть исправлена прямо в текущей ревизии и “смержена” в нужные ветви;
- может быть сделано ответвление от ветки develop или исправлено прямо в ней;

- может быть сделан хотфикс (hotfix) ответвлением от master с последующим выпуском новой версии и слиянием исправлений в master и develop.

Случаются ситуации, когда в дереве истории Git нужно найти коммит, где ошибка была исправлена, например, чтобы сообщить мэнтайнеру проекта для отметки в баг-трекере или бэкпортирования (cherry-pick) в другие долгие "longtime" ветки проекта. Для этого используются все те же самые команды, только вместо git bisect bad нужно вводить git bisect good и наоборот, так как вместо "плохого" ("bad") коммита с ошибкой мы ищем "хороший" ("good") коммит с нужным исправлением.

2.6 cherry-pick - обмен коммитами между ветками

Команда cherry-pick позволяет скопировать определённые коммиты из одной ветки в другую как если бы они были созданы изначально в ней. Например:

```
git cherry-pick master # скопирует последний коммит из master в текущую ветвь
git cherry-pick ..master          # применить все коммиты из master, отсутствующие в текущей ветке
git cherry-pick master~4 master~2 # скопировать 2 коммита из master
                                  # (3-ий и 4-ый, начиная с головы)
git cherry-pick -n master~1 next  # применить предпоследний из master и последний из next коммиты
                                  # к рабочей директории и индексу, не создавая новой ревизии
git cherry-pick --ff ..next       # если история линейная и HEAD является предшественником master,
                                  # обновить рабочую директорию и переустановить HEAD на последний
                                  # коммит master, в противном случае - скопировать все коммиты
                                  # из master
 git rev-list --reverse master -- README | git cherry-pick -n --stdin # вывести хеши ревизий
         # в master, затрагивающие файл README и применить их одним коммитом к текущей ветви
```

2.7 Субрепозитории

2.7.1 Субмодули

Или так называемые Git submodules. Применяются для включения одних проектов в другие или для создания суперпроектов. При этом обновления в подпроектах не затрагивают основной проект до тех пор, пока владелец проекта не захочет это сделать явно, убедившись, что эти изменения оставляют проект в рабочем состоянии.

Создать Git submodule очень просто, например добавим в некоторый проект my-project библиотеку my-lib:

```
cd my-project
git submodule add git@git.github.com:mynickname/my-lib.git
git commit -m ‘‘Added my-lib submodule.’’
git push
```

Команда

```
git submodule update --init --recursive
```

рекурсивно обновляет всем субмодули, которые в свою очередь могут содержать другие субмодули из origin remote. Следует отметить, что после выполнения данной команды сурепозитории могут оказаться в "detached state", то есть не привязанными к какой-либо ветке.

Если требуется внести изменения в субрепозиторий, то это делается как и с обычным проектом (edit, commit, push). Важно отметить, что изменения в субрепозитории не влияют на главный репозиторий до тех пор, пока в нём это явно не будет указано:

```
git add my-lib
git commit -m ‘‘my-lib submodule updated.’’
```

2.7.2 Поддеревья

Или так называемые Git subtrees.
Вначале нужно добавить репозиторий, который будет использоваться как subtree:

```
git remote add mylib_remote git@git.insysltd.ru:insys/mylib.git
git fetch mylib_remote
git checkout -b mylib_branch mylib_remote/master
git checkout master
```

Допустим, мы хотим поместить проект mylib в подкаталог с тем же именем:

```
git read-tree --prefix=mylib/ -u mylib_branch
```

В отличие от субмодулей данные поддерева хранятся физически в репозитории. Субмодули же, по своей сути, лишь ссылаются на данные в другом репозитории.
Обновление поддерева происходит довольно легко:

```
git checkout mylib_branch # переключиться на ветку поддерева
git pull                  # принять удалённые изменения
git checkout master       # переключиться на версию основного проекта
git merge --squash -s subtree --no-commit mylib_branch # смержить в mylib/, не создавая коммита
git commit   # зафиксировать изменения
```

Узнать о наличии разницы между подкаталогом mylib/ и кодом в mylib_branch можно при помощи “git diff-tree”:

```
git diff-tree -p mylib_branch        # сравнить с локальной веткой
git diff-tree -p mylib_remote/master # сравнить с удалённой веткой
```

3 Администрирование

3.1 Свободное место на диске

Для удаления временных файлов и сжатия истории можно использовать следующие команды:

```
git clean -fd # удалить неотслеживаемые файлы в репозитории
git prune # удалить все недостижимые объекты/коммиты из базы данных
git prune-packed # удалить также уже упакованные недостижимые объекты
git gc --aggressive --prune=all # удалить все бесполезные объекты и оптимизировать локальный репозиторий
```

3.2 shallow-репозиторий

Это такой репозиторий, который хранит лишь часть истории, что позволяет сэкономить место на диске и входящий трафик при его создании, но запрещает клонирование и другие операции с ним.
Для создания shallow-репозитория можно выполнить клонирование обычного с опцией –depth, например:

```
git clone --depth=1 git://github.com/gentoo-mirror/gentoo.git /var/portage/portage
```

Для создания shallow-репозитория “на месте” (“inplace”) без клонирования и передачи данных по сети подходит следующий вариант (взято со stackoverflow):

```
git show-ref -s HEAD > .git/shallow
git reflog expire --expire=0
git prune
git prune-packed
```

Последняя последовательность команд совместно с git gc –aggressive –prune=all может быть использована для периодической чистки shallow-репозитория от старой истории.

3.3 Создание архива

Иногда для передачи снимка исходного кода третьим лицам требуется создать архив, не включая Git-специфичных данных и временных файлов, созданных в процессе сборки проекта. Для этой цели существует команда “git archive”.

```
git archive -o myproject-1.0.0.zip v1.0.0 > # создать Zip-архив 1-ой стабильной версии проекта
git archive --format=tgz --prefix=myproject-master/ master > myproject-master.tgz # архивировать
                                                                                  # ветку master
git archive -o myproject-doc.zip HEAD:Documentation/ # создать архив с документацией
```

3.4 Проверка целостности репозитория

Git являет по своей сути файловую систему, расположенную внутри другой файловой системы (на диске). Иногда в работе компьютера случаются сбои и файловая система компьютера может быть повреждена. Также существует ненулевая вероятность повреждения системы объектов репозитория в результате типовой работы (напр., использование нестабильной версии Git), хотя она крайне мала.

Для проверки целостности объектов бд и истории репозитория имеется команда “git fsck”. Примеры использования:

```
git fsck # выполнить проверку
```

Иногда случается, что объект добавлен в индекс “git add”, а впоследствии удалён по ошибке, например командой “git reset –hard”. В Git предусмотрена возможность восстановления данных и на этот случай, пример:

```
git add main.c        # добавили изменения в индекс
git reset --hard      # случайно откатились к вершине истории
git fsck --lost-found # с большой долей вероятности изменённый main.c находится
                      # в .git/lost-found/other, только в место имени хеш ревизии
```

Бывает также, что пользователь отключил “reflog” в настройках Git по ошибке или по неопределённым личным обстоятельствам. В таких случаях может буквально спасти опция “–dangling” у “git fsck”:

```
$ git fsck --dangling
Checking object directories: 100% (256/256), done.
Checking objects: 100% (84/84), done.
dangling blob cd0120b458d5ab07efed5bb690b0eec8c1801b55
```

В конце выхлопа команды указаны хеши некогда существующих объектов, которые никогда не использовались (не добавлялись в репозиторий). Для восстановления требуемых объектов используется та же команда Git “checkout”.

4 Удачная модель ветвления

Материал взят с http://habrahabr.ru/post/106912/

В этой статье я представляю модель разработки, которую использую для всех моих проектов (как рабочих, так и частных) уже в течение года, и которая показала себя с хорошей стороны. Я давно собирался написать о ней, но до сих пор не находил свободного времени. Не буду рассказывать обо всех деталях проекта, коснусь лишь стратегии ветвления и управления релизами.

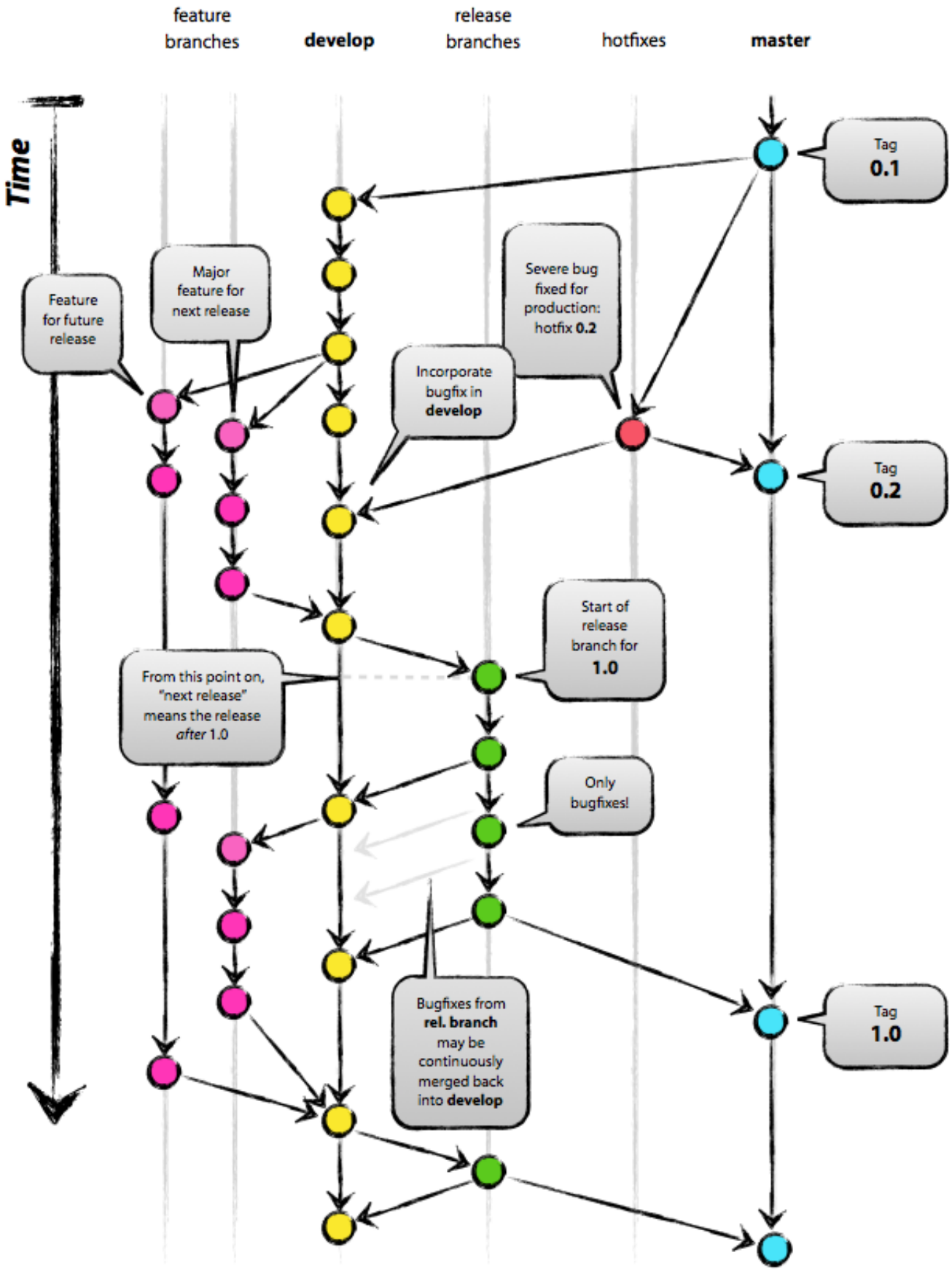

Рис. 4.1: Удачная модель ветвления

В качестве инструмента управления версиями всего исходного кода она использует Git.

4.1 Почему Git?

За полноценным обсуждением всех достоинств и недостатков Git в сравнении с централизованными системами контроля версий обращайтесь к всемирной сети. Там Вы найдёте достаточное количество споров на эту тему. Лично же я, как разработчик, на данный момент предпочитаю Git всем остальным инструментам. Git реально смог изменить отношение разработчиков к процессам слияния и ветвления. В классическом мире CVS/Subversion, из которого я пришёл, ветвление и слияние обычно считаются опасными («опасайтесь конфликтов слияния, они больно кусаются!»), и потому проводятся как можно реже.

Но с Git эти действия становятся исключительно простыми и дешёвыми, и потому на деле они становятся центральными элементами обычного *ежедневного* рабочего процесса. Просто сравните: в книгах по CVS/Subversion ветвление и слияние обычно рассматриваются в последних главах (для продвинутых пользователей), в то время как в любой книге про Git они бывают упомянуты уже к третьей главе (основы).

Благодаря своей простоте и предсказуемости, ветвление и слияние больше не являются действиями, которых стоит опасаться. Теперь инструменты управления версиями способны помочь в ветвлении и слиянии больше, чем какие-либо другие.

Но хватит говорить об инструментах, давайте перейдём к модели разработки. Модель, которую я хочу представить, — это, по сути, просто набор процедур, которые исполняет каждый член команды, чтобы все вместе могли достичь высокой управляемости процесса разработки.

4.2 Децентрализованный, но централизованный

Предлагаемая модель ветвления опирается на конфигурацию проекта, содержащую один центральный «истинный» репозиторий. Замечу, что этот репозиторий только *считается* центральным (так как Git является DVCS, у него нет такой вещи, как главный репозиторий, на техническом уровне). Мы будем называть этот репозиторий термином origin, т.к. это имя и так знакомо всем пользователям Git.

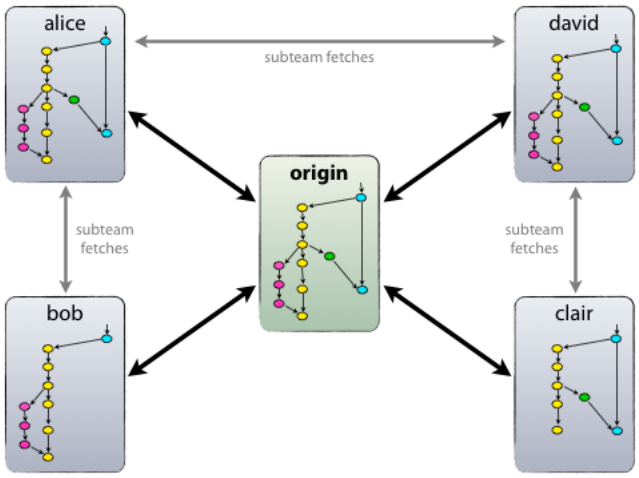

Рис. 4.2: Децентрализованный, но централизованный

Каждый разработчик забирает и публикует изменения (pull & push) в origin. Но, помимо централизованных отношений push-pull, каждый разработчик также может забирать изменения от остальных коллег внутри своей микрокоманды. Например, этот способ может быть удобен в ситуации, когда двое или более разработчиков работают вместе над большой новой фичей, но не могут издать незавершённую работу в origin раньше времени. На картинке выше изображены подгруппы Алисы и Боба, Алисы и Дэвида, Клэр и Дэвида.

Технически это реализуется несложно: Алиса создаёт удалённую ветку Git под названием bob, которая указывает на репозиторий Боба, а Боб делает то же самое с её репозиторием.

4.3 Главные ветви

Ядро модели разработки не отличается от большинства существующих моделей. Центральный репозиторий содержит две главные ветки, существующие всё время.

- master
- develop

Ветвь master создаётся при инициализации репозитория, что должно быть знакомо каждому пользователю Git. Параллельно ей также мы создаём ветку для разработки под названием develop.

Мы считаем ветку origin/master главной. То есть, исходный код в ней должен находиться в состоянии *production-ready* в любой произвольный момент времени.

Ветвь origin/develop мы считаем главной ветвью для разработки. Хранящийся в ней код в любой момент времени должен содержать самые последние изданные изменения, необходимые для следующего релиза. Эту ветку также можно назвать «интеграционной». Она служит источником для сборки автоматических ночных билдов.

Когда исходный код в ветви разработки (develop) достигает стабильного состояния и готов к релизу, все изменения должны быть определённым способом влиты в главную ветвь (master) и помечены тегом с номером релиза. Ниже мы рассмотрим этот процесс в деталях.

Следовательно, каждый раз, когда изменения вливаются в главную ветвь (master), мы *по определению* получаем новый релиз. Мы стараемся относиться к этому правилу очень строго, так что, в принципе, мы могли бы использовать хуки Git, чтобы автоматически собирать наши продукты и выкладывать их на рабочие сервера при каждом коммите в главную ветвь (master).

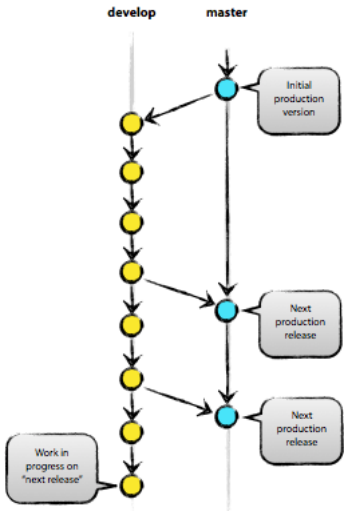

Рис. 4.3: Главные ветви

4.4 Вспомогательные ветви

Помимо главных ветвей master и develop, наша модель разработки содержит некоторое количество типов вспомогательных ветвей, которые используются для распараллеливания разработки между членами команды, для упрощения внедрения нового функционала (features), для подготовки релизов и для быстрого исправления проблем в производственной версии приложения. В отличие от главный ветвей, эти ветви всегда имеют ограниченный срок жизни. Каждая из них в конечном итоге рано или поздно удаляется.

Мы используем следующие типы ветвей:

- Ветви функциональностей (Feature branches)
- Ветви релизов (Release branches)
- Ветви исправлений (Hotfix branches)

У каждого типа ветвей есть своё специфическое назначение и строгий набор правил, от каких ветвей они могут порождаться, и в какие должны вливаться. Сейчас мы рассмотрим их по очереди.

Конечно же, с технической точки зрения, у этих ветвей нет ничего «специфического». Разбиение ветвей на категории существует только с точки зрения того, как они используются. А во всём остальном это старые добрые ветви Git.

4.4.1 Ветви функциональностей (feature branches)

Могут порождаться от: develop

Должны вливаться в: develop

Соглашение о наименовании: всё, за исключением master, develop, release-* или hotfix-*

Ветви функциональностей (feature branches), также называемые иногда тематическими ветвями (topic branches), используются для разработки новых функций, которые должны появиться в текущем или будущем релизах. При начале работы над функциональностью (фичей) может быть ещё неизвестно, в какой именно релиз она будет добавлена. Смысл существования ветви функциональности (feature branch) состоит в том, что она живёт так долго, сколько продолжается разработка данной функциональности (фичи). Когда работа в ветви завершена, последняя вливается обратно в главную ветвь разработки (что означает, что функциональность будет добавлена в грядущий релиз) или же удаляется (в случае неудачного эксперимента).

Ветви функциональностей (feature branches) обычно существуют в репозиториях разработчиков, но не в главном репозитории (origin).

4.4.1.1 Создание ветви функциональности (feature branch)

При начале работы над новой функциональностью делается ответвление от ветви разработки (develop).

```
$ git checkout -b myfeature develop
Switched to a new branch "myfeature"
```

4.4.1.2 Добавление завершённой функциональности в develop

Завершённая функциональность (фича) вливается обратно в ветвь разработки (develop) и попадает в следующий релиз.

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff myfeature
Updating ea1b82a..05e9557
(Отчёт об изменениях)
$ git branch -d myfeature
Deleted branch myfeature (was 05e9557).
$ git push origin develop
```

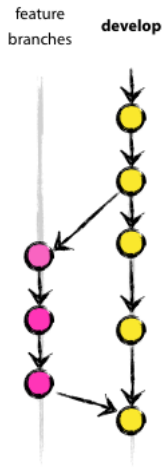

Рис. 4.4: Ветви функциональностей (feature branches)

Флаг –no-ff вынуждает Git всегда создавать новый объект коммита при слиянии, даже если слияние может быть осуществлено алгоритмом fast-forward. Это позволяет не терять информацию о том, что ветка существовала, и группирует вместе все внесённые изменения. Сравните:

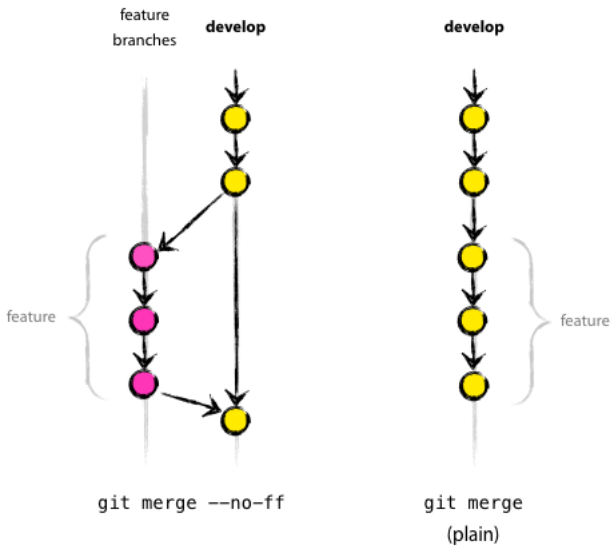

Рис. 4.5: no-fast-forward VS fast-forward

Во втором случае невозможно увидеть в истории изменений, какие именно объекты коммитов совместно образуют функциональность, — для этого придётся вручную читать все сообщения в коммитах. Отменить функциональность целиком (т.е., группу коммитов) в таком случае невозможно без головной боли, а с флагом –no-ff это делается элементарно.

Конечно, такой подход создаёт некоторое дополнительное количество (пустых) объектов коммитов, но получаемая выгода более чем оправдывает подобную цену.

К сожалению, я ещё не нашёл, как можно настроить Git так, чтобы –no-ff было поведением по-умолчанию при слияниях. Но этот способ должен быть реализован.

4.4.2 Ветви релизов (release branches)

Могут порождаться от: develop
Должны вливаться в: develop и master

Соглашение о наименовании: release-*

Ветви релизов (release branches) используются для подготовки к выпуску новых версий продукта. Они позволяют расставить финальные точки над i перед выпуском новой версии. Кроме того, в них можно добавлять минорные исправления, а также подготавливать метаданные для очередного релиза (номер версии, дата сборки и т.д.). Когда вся эта работа выносится в ветвь релизов, главная ветвь разработки (develop) очищается для добавления последующих фич (которые войдут в следующий большой релиз).

Новую ветку релиза (release branch) надо порождать в тот момент, когда состояние ветви разработки полностью или почти полностью соответствует требованиям, соответствующим новому релизу. По крайней мере, вся необходимая функциональность, предназначенная к этому релизу, уже влита в ветвь разработки (develop). Функциональность, предназначенная к следующим релизам, может быть и не влита. Даже лучше, если ветки для этих функциональностей подождут, пока текущая ветвь релиза не отпочкуется от ветви разработки (develop).

Очередной релиз получает свой номер версии только в тот момент, когда для него создаётся новая ветвь, но ни в коем случае не раньше. Вплоть до этого момента ветвь разработки содержит изменения для «нового релиза», но пока ветка релиза не отделилась, точно неизвестно, будет ли этот релиз иметь версию 0.3, или 1.0, или какую-то другую. Решение принимается при создании новой ветви релиза и зависит от принятых на проекте правил нумерации версий проекта.

4.4.2.1 Создание ветви релиза (release branch) Ветвь релиза создаётся из ветви разработки (develop). Пускай, например, текущий изданный релиз имеет версию 1.1.5, а на подходе новый большой релиз, полный изменений. Ветвь разработки (develop) готова к «следующему релизу», и мы решаем, что этот релиз будет иметь версию 1.2 (а не 1.1.6 или 2.0). В таком случае мы создаём новую ветвь и даём ей имя, соответствующее новой версии проекта:

```
$ git checkout -b release-1.2 develop
Switched to a new branch "release-1.2"
$ ./bump-version.sh 1.2
Files modified successfully, version bumped to 1.2.
$ git commit -a -m "Bumped version number to 1.2"
[release-1.2 74d9424] Bumped version number to 1.2
1 files changed, 1 insertions(+), 1 deletions(-)
```

Мы создали новую ветку, переключились в неё, а затем выставили номер версии (bump version number). В нашем примере bump-version.sh — это вымышленный скрипт, который изменяет некоторые файлы в рабочей копии, записывая в них новую версию. (Разумеется, эти изменения можно внести и вручную; я просто обращаю Ваше внимание на то, что *некоторые* файлы изменяются.) Затем мы делаем коммит с указанием новой версии проекта.

Эта новая ветвь может существовать ещё некоторое время, до тех пор, пока новый релиз окончательно не будет готов к выпуску. В течение этого времени к этой ветви (а не к develop) могут быть добавлены исправления найденных багов. Но добавление крупных новых изменений в эту ветвь строго запрещено. Они всегда должны вливаться в ветвь разработки (develop) и ждать следующего большого релиза.

4.4.2.2 Закрытие ветви релиза Когда мы решаем, что ветвь релиза (release branch) окончательно готова для выпуска, нужно проделать несколько действий. В первую очередь ветвь релиза вливается в главную ветвь (напоминаю, каждый коммит в master — это *по определению* новый релиз). Далее, этот коммит в master должен быть помечен тегом, чтобы в дальнейшем можно было легко обратиться к любой существовавшей версии продукта. И наконец, изменения, сделанные в ветви релиза (release branch), должны быть добавлены обратно в разработку (ветвь develop), чтобы будущие релизы также содержали внесённые исправления багов.

Первые два шага в Git:

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff release-1.2
Merge made by recursive.
(Отчёт об изменениях)
$ git tag -a 1.2
```

Теперь релиз издан и помечен тегом.

Замечание: при желании, Вы также можете использовать флаги -s или -u <ключ>, чтобы криптографически подписать тег.

Чтобы сохранить изменения и в последующих релизах, мы должны влить эти изменения обратно в разработку. Делаем это так:

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff release-1.2
Merge made by recursive.
(Отчёт об изменениях)
```

Этот шаг, в принципе, может привести к конфликту слияния (нередко бывает, что к причиной конфликта является изменение номера версии проекта). Если это произошло, исправьте их и издайте коммит.

Теперь мы окончательно разделались с веткой релиза. Можно её удалять, потому что она нам больше не понадобится:

```
$ git branch -d release-1.2
Deleted branch release-1.2 (was ff452fe).
```

4.4.3 Ветви исправлений (hotfix branches)

Могут порождаться от: master

Должны вливаться в: develop и master

Соглашение о наименовании: hotfix-*

Ветви для исправлений (hotfix branches) весьма похожи на ветви релизов (release branches), так как они тоже используются для подготовки новых выпусков продукта, разве лишь незапланированных. Они порождаются необходимостью немедленно исправить нежелательное поведение производственной версии продукта. Когда в производственной версии находится баг, требующий немедленного исправления, из соответствующего данной версии тега главной ветви (master) порождается новая ветвь для работы над исправлением.

Смысл её существования состоит в том, что работа команды над ветвью разработки (develop) может спокойно продолжаться, в то время как кто-то один готовит быстрое исправление производственной версии.

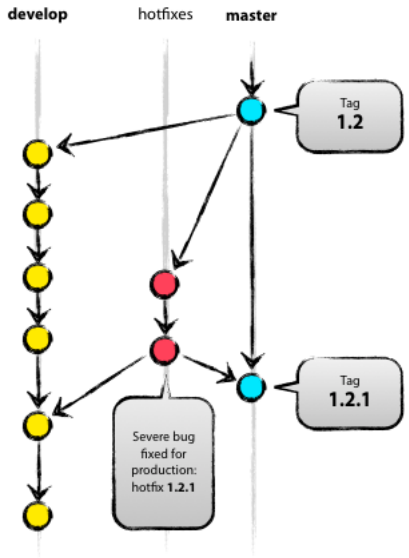

Рис. 4.6: Ветви исправлений (hotfix branches)

4.4.3.1 Создание ветви исправлений (hotfix branch)

Ветви исправлений (hotfix branches) создаются из главной (master) ветви. Пускай, например, текущий производственный релиз имеет версию 1.2, и в нём (внезапно!) обнаруживается серьёзный баг. А изменения в ветви разработки (develop) ещё недостаточно стабильны, чтобы их издавать в новый релиз. Но мы можем создать новую ветвь исправлений и начать работать над решением проблемы:

```
$ git checkout -b hotfix-1.2.1 master
Switched to a new branch "hotfix-1.2.1"
$ ./bump-version.sh 1.2.1
Files modified successfully, version bumped to 1.2.1.
$ git commit -a -m "Bumped version number to 1.2.1"
[hotfix-1.2.1 41e61bb] Bumped version number to 1.2.1
1 files changed, 1 insertions(+), 1 deletions(-)
```

Не забывайте обновлять номер версии после создания ветви!

Теперь можно исправлять баг, а изменения издавать хоть одним коммитом, хоть несколькими.

```
$ git commit -m "Fixed severe production problem"
[hotfix-1.2.1 abbe5d6] Fixed severe production problem
5 files changed, 32 insertions(+), 17 deletions(-)
```

4.4.3.2 Закрытие ветви исправлений Когда баг исправлен, изменения надо влить обратно в главную ветвь (master), а также в ветвь разработки (develop), чтобы гарантировать, что это исправление окажется и в следующем релизе. Это очень похоже на то, как закрывается ветвь релиза (release branch).

Прежде всего надо обновить главную ветвь (master) и пометить новую версию тегом.

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Отчёт об изменениях)
$ git tag -a 1.2.1
```

Замечание: при желании, Вы также можете использовать флаги -s или -u <ключ>, чтобы криптографически подписать тэг.

Следующим шагом переносим исправление в ветвь разработки (develop).

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Отчёт об изменениях)
```

У этого правила есть одно исключение: **если в данный момент существует ветвь релиза (release branch), то ветвь исправления (hotfix branch) должна вливаться в неё, а не в ветвь разработки (develop)**. В этом случае исправления войдут в ветвь разработки вместе со всей ветвью релиза, когда та будет закрыта. (Хотя, если работа в develop требует немедленного исправления бага и не может ждать, пока будет завершено издание текущего релиза, Вы всё же можете влить исправления (bugfix) в ветвь разработки (develop), и это будет вполне безопасно).

И наконец, удаляем временную ветвь:

```
$ git branch -d hotfix-1.2.1
Deleted branch hotfix-1.2.1 (was abbe5d6).
```

4.5 Заключение

Хотя в этой модели ветвления совершенно нет ничего принципиально нового, «большая картинка», с которой начинается эта статья, зарекомендовала себя в наших проектах с самой лучшей стороны. Она формирует элегантную мысленную модель, которую легко полностью охватить одним взглядом, и которая позволяет сформировать у команды совместное понимание процессов ветвления и слияния, действующих на проекте.

Высококачественная PDF-версия этой картинки свободна для скачивания здесь. Распечатайте её и повесьте у себя на стену, чтобы к ней можно было обратиться в любой момент.

Прим. переводчика: *статья не новая, ссылка на оригинал уже появлялась на хабре. Этот перевод — для тех, кому английский ещё даётся не так легко (а также для моих коллег, среди которых я занимаюсь пропагандой, хехе). Для автоматизации описанных в статье процедур автор создал проект gitflow, который можно найти на github.*

5 Рекомендуемая литература

1. Документация на официальном сайте Git (англ.) http://git-scm.com/documentation

A Пример Git-конфигурации ~/.gitconfig

Приведём пример типичного файла конфигурации. Все опции могут быть вписаны вручную, но могут быть заданы через команду git config, например:

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
```

Опция –global указывает, что настройки должны быть применены глобально, а не только к текущему репозиторию. Пример файла конфигурации, расположенного в домашней директории пользователя ~/.gitconfig:

```
[user]
    email = vasya.pupkin@gmail.com
    name = Vasya P
[color]
    ui = auto
    branch = auto
    diff = auto
    status = auto
[color "branch"]
    current = yellow reverse
    local = yellow
    remote = green
[color "diff"]
    meta = yellow bold
    frag = magenta bold
    old = red bold
    new = green bold
[color "status"]
    added = yellow
    changed = green
    untracked = cyan
[alias]
    unstage = reset HEAD --
    st = status
    s = status -uno
    ci = commit
    di = diff -b
    co = checkout
    up = checkout
    update = checkout
    l = log
    hgrevert = checkout
    strip = reset --hard
    branches = branch -a
    pull = fetch
[instaweb]
    local = true
    httpd = lighttpd -f
    port = 4321
    browser = firefox
```

```
[merge]
    tool = vimdiff
[core]
    quotepath = false
[push]
    default = matching
```