Планирование

А также использование Git

Фаза планирования

- Процесс проектирования (design - что создавать?)
- Процесс планирования (planning - как создавать?)
- Разработка календарного графика (scheduling - когда создавать?)

Иерархическая структура работ

Иерархическая структура работ (Work Breakdown Structure - WBS) – это структуризация работ проекта, отражающая его основные результаты и определяющая его рамки. Работа, не описанная в WBS, находится вне границ проекта. Для лидеров групп и ролевого кластера “Управление программой” WBS – это инструмент управления проектом, облегчающий создание планов и календарных графиков.

WBS создают все ролевые кластеры

Процесс создания WBS

При определении фронта необходимых работ и его разбиении на меньшие части, члены ролевых кластеров совместно анализируют спецификации составляющих решения. Этот процесс называется декомпозицией работы (work breakdown / work decomposition).

Один из результатов процесса управления рисками MSF – появление дополнительных задач, являющихся реакцией на имеющиеся риски. Эта работа может быть включена в WBS, оценена, спланирована и внесена в календарный график точно так же, как и другие задачи.

Рекомендации по декомпозиции

• Затраты на каждую задачу должны быть реалистично оцениваемы.
• Оценка времени исполнения каждой задачи не должна быть менее одного или более 40 дней (для IT-проектов).
• Каждая задача должна иметь однозначное описание как её самой, так и ожидаемого результата.
• Задачи выделены правильно, если их выполнение может производиться без существенных пауз.

Рекомендации по декомпозиции

• За исключением двух верхних уровней, задачи должны формулироваться в повелительном наклонении (например, “Спроектировать схему базы данных” вместо “Схема базы данных”).
• В WBS должно быть от трех до пяти уровней определения задач.
• По ходу работы над проектом WBS последовательно дорабатывается, но обычно ее формирование производится на фазе планирования

Рекомендации по декомпозиции

• Ответственность за каждую задачу должна быть поручена одному работнику.
• Каждая задача может предполагать дальнейшее разбиение на элементарные подзадачи.
• Деятельность, сопряженная с большими рисками, должна детализироваться больше, чем деятельность, сопряженная с меньшими рисками

Оценка снизу вверх

предварительные оценки длительности задач следует получать от тех, кто будет затем выполнять оцениваемую работу. Оценка снизу вверх (bottom-up estimating) – это процесс выработки и интеграции оценок многими членами команды. Такой подход обладает следующими преимуществами:

- Большая точность.
- Ответственность.
- Уполномоченость (empowerment) проектной группы.

Календарный план

- Упорядочивайте задачи
- Задавайте временные рамки
- Учитывайте риски
- Создавайте временные буферы

Буферное время

- Не добавляйте буферы в качестве резерва времени для запланированных задач.
- Буферное время должно выделяться как будто бы под дополнительно существующую задачу. Временные буфера всегда должны дополнять критический путь проекта (project’s critical path).
- Использование буферного времени по ходу проекта должно подвергаться жесткому контролю.
- Если потребовалось расширить функциональность решения или уменьшилось количество доступных ресурсов, не компенсируйте это использованием буферного времени.
- Если буферное время исчерпано, поставьте в известность всю проектную группу о том, что любой сбой или задержка будут ударом по работе над проектом и создадут опасность выхода за временные рамки.

Git

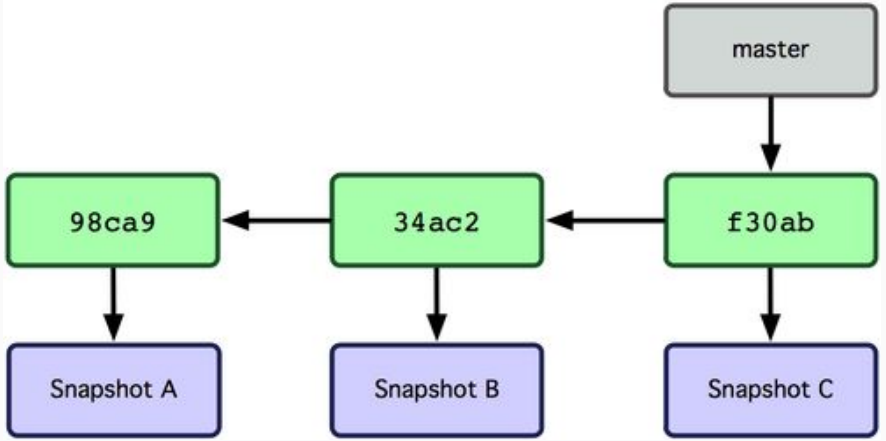
Git — распределённая VCS. Это значит, что мы работаем не с одним репозиторием на сервере, а каждый имеет у себя локальную копию репозитория. Соответственно, такие операции, как checkout и commit производятся с локальным репозиторием. Друг с другом же (или с тем, что на сервере) репозитарии синхронизируются специально предназначенными командами pull (fetch) и push.

Зачем нужны ветки(brunch)

Ветка позволяет сохранить текущее состояние кода, и экспериментировать. Например, вы пишите новый модуль. Логично делать это в отдельной ветке. Звонит начальство и говорит, что в проекте баг и срочно нужно пофиксить, а у вас модуль не дописан. Как же заливать нерабочие файлы? Просто переключитесь на рабочую ветку без модуля, пофиксите баг и заливайте файлы на сервер. А когда «опасность» миновала — продолжите работу над модулем. И это один из многих примеров пользы веток.

Ветки в git

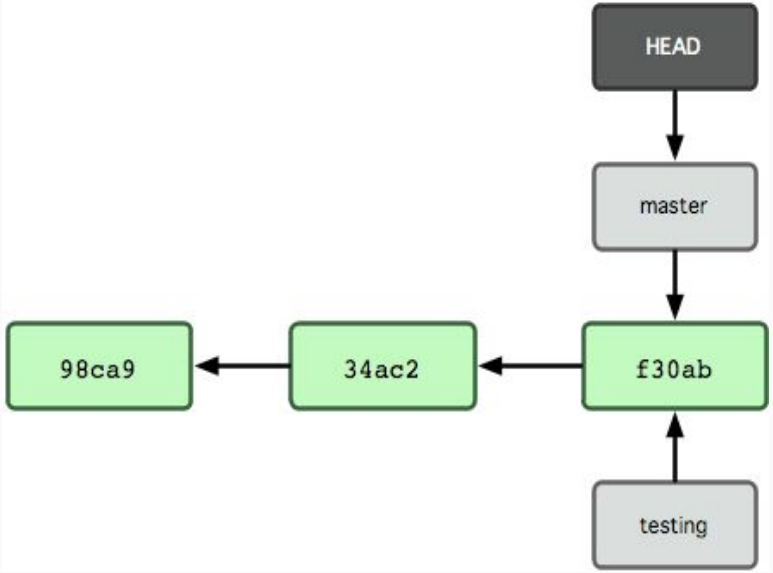
Ветка в Git'е — это просто легковесный подвижный указатель на один из коммитов. Ветка по умолчанию в Git'е называется master. Когда вы создаёте коммиты на начальном этапе, вам дана ветка master, указывающая на последний сделанный коммит. При каждом новом коммите она сдвигается вперёд автоматически.



Создание новой ветки

Что произойдёт, если вы создадите новую ветку? Итак, этим вы создадите новый указатель, который можно будет перемещать.

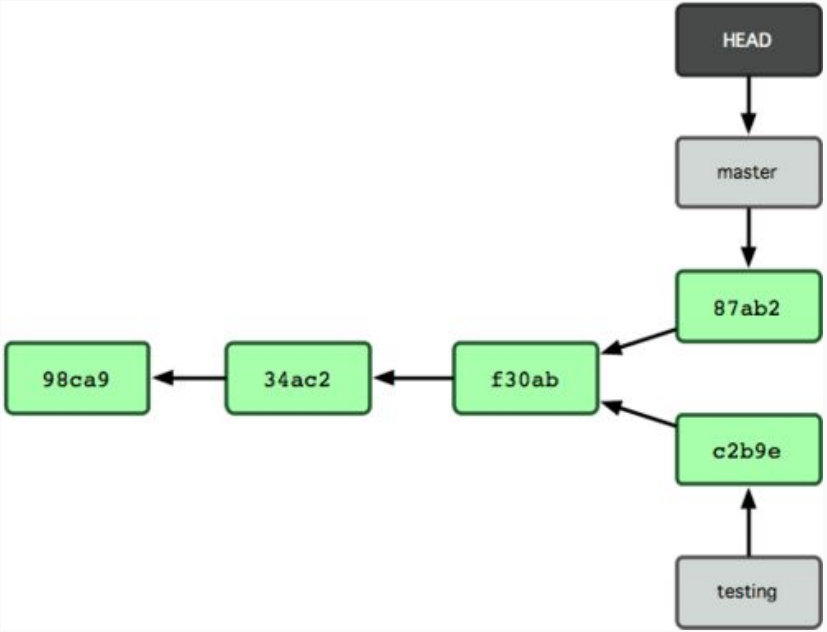
Откуда Git узнает, на какой ветке вы находитесь в данный момент? Он хранит специальный указатель, который называется HEAD. При создании ветки указатель head не смещается.



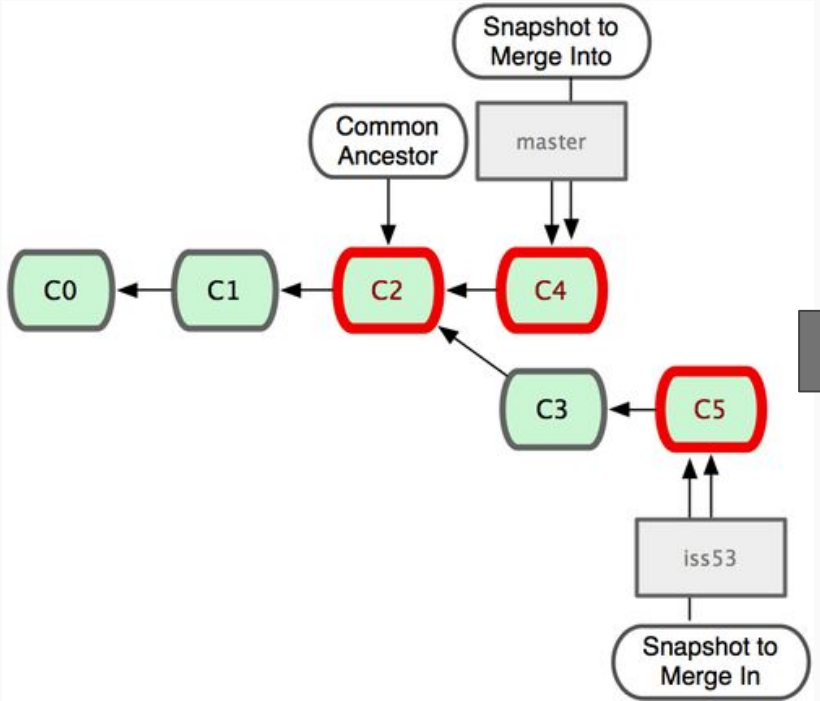
Ветвление веток

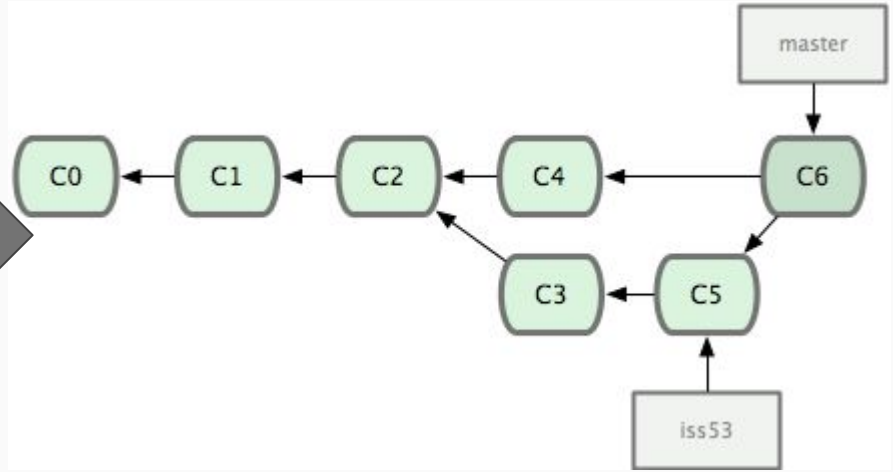
Если теперь изменить текущую ветку, сделать в ней commit, а потом вернуться в master и сделать commit там, получим вот такую картину.

Оба эти изменения изолированы в отдельных ветках: вы можете переключаться туда и обратно между ветками и слить их, когда будете готовы.

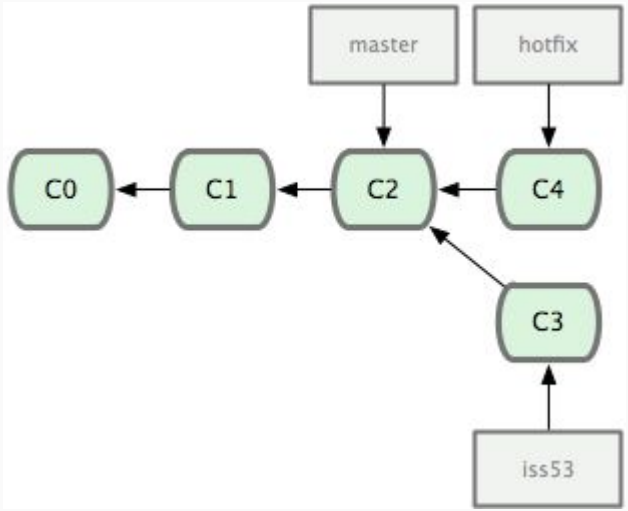


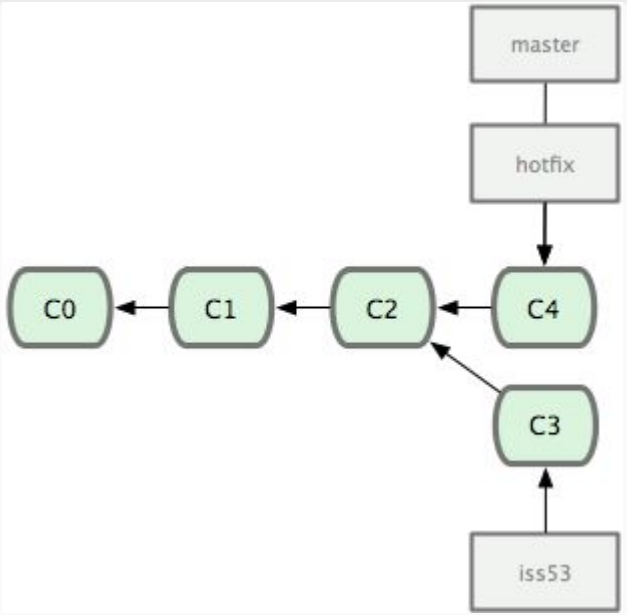
Слияние веток(трехходовое)





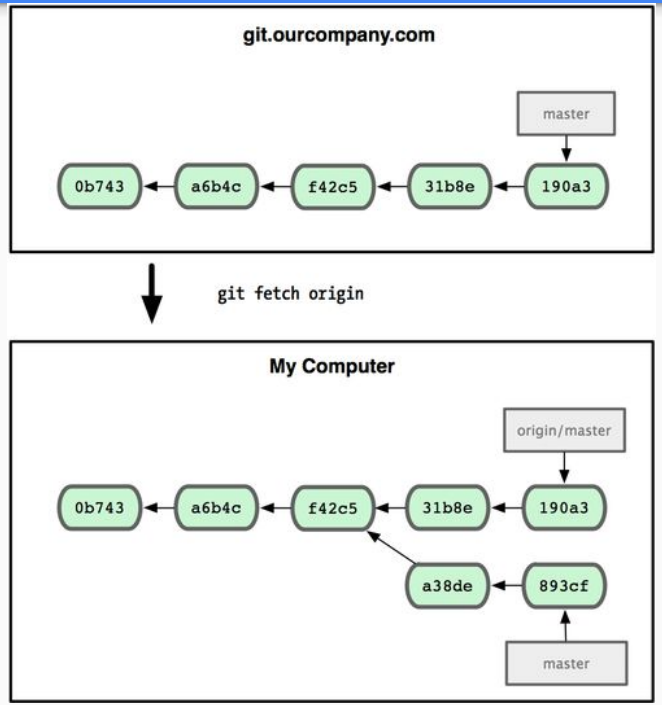
Слияние веток(fast forward)





Работа с удаленными(remote) ветками

При выполнении локальной работы и отправке кем-то изменений на удалённый сервер каждая история продолжается по-разному. Ее необходимо синхронизировать отдельной командой(Git fetch). Так можно забрать изменения выполненные другими членами команды.



Работа с удаленными ветками

Когда вы хотите поделиться веткой с окружающими, вам необходимо отправить (push) её на удалённый сервер, на котором у вас есть права на запись. Ваши локальные ветки автоматически не синхронизируются с удалёнными серверами — вам нужно явно отправить те ветки, которыми вы хотите поделиться.

Получение локальной ветки с помощью git checkout из удалённой ветки автоматически создаёт то, что называется отслеживаемой веткой. Отслеживаемые ветки — это локальные ветки, которые напрямую связаны с удалённой веткой. Если, находясь на отслеживаемой ветке, вы наберёте git push, Git уже будет знать, на какой сервер и в какую ветку отправлять изменения.

Полезные принципы работы с ветками

master — это та ветка, которая всегда, в любой (!) момент должна быть готова к поставке заказчику. Поэтому мы никогда не делаем новые фичи и багфиксы сразу в master, используем для этого ветки.Очень важный принцип: одна фича — одна ветка. Один багфикс (если предполагается длиннее двух коммитов) — одна ветка. Один эксперимент — одна ветка. Одна фича внутри эксперимента — ветка от ветки.
Всегда пишем вразумительные комментарии к коммитам, плюс каждый коммит должен быть связан с задачей(дефектом). После того, как фича (багфикс) написаны, оттестированы и готовы к продакшну, мержим ветку в master.

Стандартные операции в git

Начало работы — клонирование репозитория. Предполагается, что у вас уже есть и настроен gitosis, на котором лежит проект.Этот шаг делается один раз.

```
git clone
gitosis@git.yourserver.com:
yourproject.git
```

Результатом будет папка yourproject с проектом у вас на жёстком диске. Команда clone делает следующие вещи: клонирует удалённый репозитарий в новую папку (yourproject в данном случае), создаёт в локальном репозитарии remote-tracking ветки для всех веток удалённого репозитария, создаёт локальную копию активной в данный момент удалённой ветки и делает из неё checkout.

Стандартные операции в git

git branch — смотрим, какие ветки у нас есть в данный момент в репозитории.

Сразу после клонирования у вас будет видна только одна, активная в данный момент в удалённом репозитории, ветка (в нашем случае это по умолчанию master, т.к. удалённый репозиторий находится на сервере и в нём ветки никто не переключает).

Если в репозитории есть другие ветки, их можно увидеть, добавив ключ -a.

Звездочкой * отмечена выбранная ветка

```
$ git branch -a
* master
origin/HEAD
origin/master
origin/feature
```

Стандартные операции в git

Допустим, мы хотим реализовать задачу feature. Создаём локально новую ветку с помощью команды branch. Командой checkout можно переключаться между ветками:

```
$ git branch feature
$ git checkout feature
Switched to branch "feature"
```

Либо

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

Стандартные операции в Git

Если мы хотим разрабатывать новый функционал совместно, нам нужно опубликовать нашу ветку на сервере, чтобы другие могли с ней работать.

Но при таком способе публикации, не устанавливается связь между локальной версией ветки и опубликованной. Т.е. если кто-то закоммитит изменения в эту удаленную ветку и вы сделаете git pull, то будет ошибка

```
$ git push origin feature:refs/heads/feature
Counting objects: 4, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 273 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To gitosis@git.yourserver.com:yourproject.git
* [new branch] feature -> feature
```

Стандартные операции в git

Как присоединиться к работе над веткой. Предполагается, что вы уже склонировали себе репозиторий. Главное здесь — правильно подключить удалённую ветку. Сделать это можно с помощью ключа --track команды git checkout. Команда создаёт локальную ветку feature и подключает её к удалённой ветке origin/feature, после чего переключается в эту ветку.

```
$ git checkout --track -b feature
origin/feature
Branch feature set up to track remote
branch refs/remotes/origin/feature.
Switched to a new branch "feature"
```

Стандартные операции в git

Как переключиться в другую ветку, когда в текущей есть изменения и коммитить их рано.

```
$ git status
# On branch feature
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# new file: somefile
#
$ git stash
Saved working directory and index state "WIP
on feature: b0a86e0... blabla"
HEAD is now at b0a86e0 blabla
(To restore them type "git stash apply")
$ git status
# On branch feature
nothing to commit (working directory clean)
```

По возвращении в эту ветку сделать

```
$ git stash apply
# On branch feature
# Changes to be committed:
# (use "git reset HEAD
<file>..." to unstage)
#
# new file: somefile
#
```

Стандартные операции git

merge и rebase. Использовать какую-либо из этих команд вам понадобится в 2-х случаях:
Вы хотите подлить свежие изменения из master к себе в ветку;
Вы хотите слить свою ветку в master.

Общее правило такое: если мы работаем с веткой самостоятельно и не планируем публиковать её на сервере — то выгоднее использовать rebase. Если же мы публикуем ветку командой push, то использовать rebase НЕЛЬЗЯ, иначе мы автоматически инвалидируем работу коллег.

```
$ git checkout master
Switched to branch "master"
$ git rebase feature
First, rewinding head to
replay your work on top of
it...
HEAD is now at 9bfac0a
feature1
Applying file2
```

```
$ git checkout master
Switched to branch "master"
$ git merge feature
Merge made by recursive.
feature1 | 1 +
```

Стандартные операции в git

После того, как мы закончили работать с веткой и слили изменения в master (или в другую ветку), можно удалить ветку.

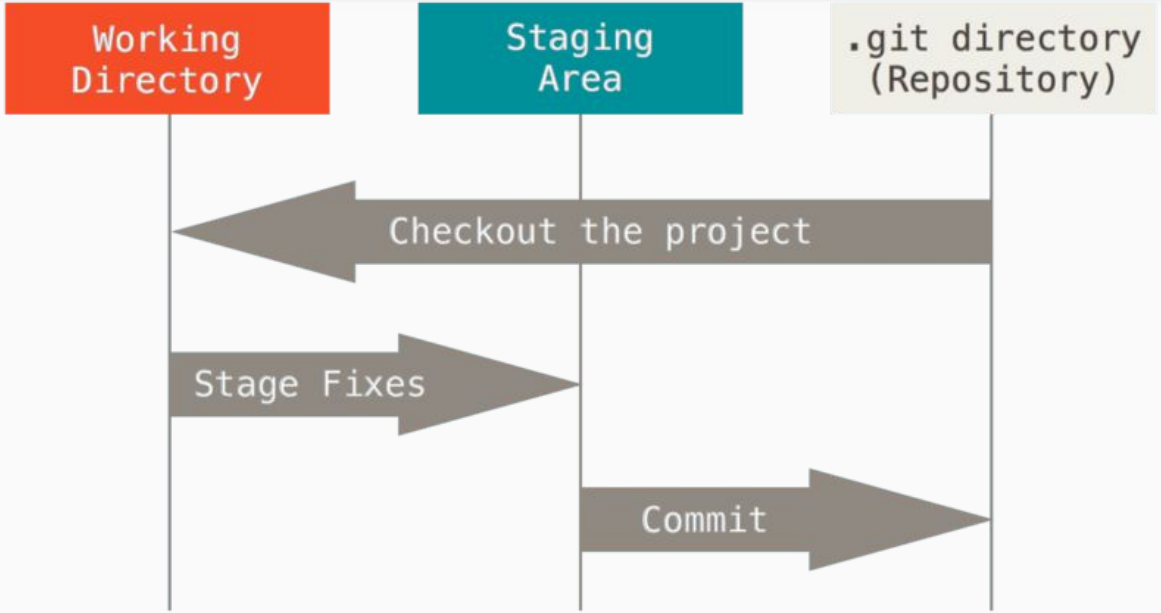
```
$ git branch -d feature
Deleted branch feature.
```

Для remote ветки:

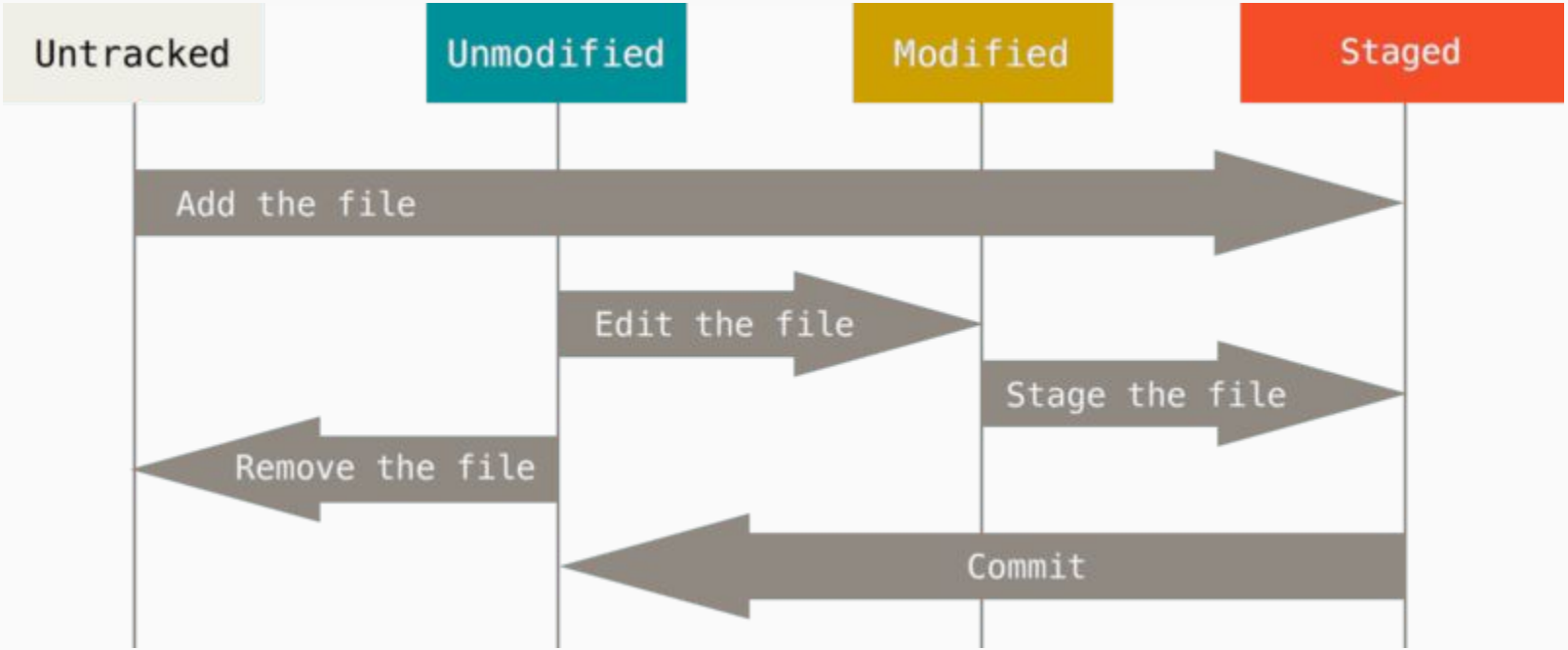
```
$ git push origin :feature
- [deleted] feature
```

Три состояния git

Три состояния файлов: зафиксированное, изменённое и подготовленное.
Зафиксированный(committed) значит, что файл уже сохранён в вашей локальной базе.
Измененный(modified) относятся файлы, которые поменялись, но ещё не были зафиксированы.
Подготовленные(staged)— это изменённые файлы, отмеченные для включения в следующий коммит.



Жизненный цикл файлов в git



Стандартные операции git

В git есть такое понятие как индекс(stage area). Например, команда commit добавляет в репозиторий только те файлы, которые есть в данный момент в индексе. Поэтому, во время работы не забываем добавлять (git add) или удалять (git rm) файлы в/из индекса репозиторий . Обратите внимание, что, если вы изменили файл, его заново нужно добавить в индекс командой git add.

```
$ git add file1 file2
$ git commit -m "adding file1 & file2"
Created initial commit 8985f44:
adding file1 & file2
2 files changed, 2 insertions(+), 0
deletions(-)
create mode 100644 file1
create mode 100644 file2
```

Стандартные операции git

Команда git add . добавляет все untracked файлы в индекс (рекурсивно), а ключ -a у команды commit позволяет автоматически добавить все модифицированные (но не новые!) файлы.

Текущее состояние индекса можно посмотреть командой git status:

```
$ git status
# On branch feature
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# new file: file1
# new file: file2
#
```

Полезные ссылки

Книга Pro Git https://git-scm.com/book/ru/v1

Git клиент для windows TortoiseGit https://tortoisegit.org

Другие популярные графические git клиенты тут:https://git-scm.com/downloads/guis