1) Отличия гит и свн. Централизованный, распределенный.

Коммит это снепшот а не patch как в SVN

Принцип организации веток.

В нашей ситуации распределенность не так важна. Все все равно работают через центральный репозиторий.

Плюшка. Локальная копия репозитория всегда есть. Нет зависимости от связи с сервером. История доступна всегда и работает быстро (гораздо быстрее чем в svn)

Многие, кто прежде работал с SVN переходя на Git боятся работать с ветками. Это как я считаю основная проблема. Git предоставляет удобный механизм работы с ветками и пренебрегать ими в git значит потерять

Для начала рассмотрим как устроен коммит в гит.

Коммит является центральной сущностью git, и для его полного понимания нужно знать о внутреннем устройстве репозитория.

Коммит по сути является снапшотом («снимком состояния») файлов. Когда вы коммитите изменения командой git commit, система создаёт новый снапшот. Это не набор изменений, а именно полный снапшот со всеми файлами репозитория. Git хорошо сжимает коммиты, так что итоговый размер получается небольшим.

Коммит состоит из следующих компонентов:

* набор файлов — это просто все файлы из снапшота в виде блобов (blob), в каждом блобе содержимое одного файла, блоб идентифицируется его SHA1-хешем, это только содержимое, без пути к файлу и без имени файла;
* дерево файлов — это древовидная структура, описывающая положение файлов в каталоге файловой системы, каждый «файл» — это его имя с путём, а также SHA1-сумма/идентификатор соответствующего блоба;
* метаданные — это набор параметров коммита, например, сообщение, автор, дата и так далее;
* идентификатор родительского коммита (или несколько родительских коммитов. Исключение первый коммит без родителя).

От всего этого набора данных считается SHA1-сумма, которая становится идентификатором коммита.

Все коммиты образуют ориентированный граф, направление идёт от «дочернего» к «родительскому» коммиту. Другими словами, каждый коммит знает своего родителя, но родители не знают своих детей.

Эту информацию можно получить зная sha1 коммита. Но придется воспользоваться консолью.

<СЛАЙД>

таким образом зная sha1 можно получить всю информацию о коммите и восстановить всю историю ДО него.

## А где каталоги?

git не хранит информацию о каталогах, он хранит только информацию о путях к файлам. Поэтому вы не сможете добавить пустой каталог в проект, во внутренней базе git просто нет объекта для этого. Если вам всё же он нужен, можете его создать, положить внутрь какой-нибудь пустой файл и закоммитить этот файл.

## Автор коммита и автор изменений

У объекта-коммита есть автор (author) и коммитер (committer). Можно интерпретировать это так, что автор — это кто написал изначальный код, а коммитер — это тот, кто его закоммитил.

## Теги, референсы и ветки

Коммиты-снапшоты в git образуют ориентированный граф, эти коммиты никогда не меняются и мы можем любой из них пометить тегом (tag). Тег — это уникальная метка-строка, указывающая на конкретный коммит, по сути он является удобным коротким именем для длинного идентификатора коммита. Обычно тегами помечаются важные коммиты, например, коммит для конкретной версии продукта.

Вы можете использовать тег везде, где требуется идентификатор коммита. Например, вы можете переключить рабочую копию на коммит конкретной версии командой типа git checkout v1.1.1.

Описанные выше теги — это внешние сущности для внутренней базы объектов, они никак не фигурируют в зависимостях между коммитами и при желании тег можно удалить и «повесить» на другой коммит.

Тег является ссылкой (reference, ref). Референс всегда «указывает» на какой-нибудь конкретный идентификатор коммита.

Ещё одним типом референса является ветка (branch). Ветки в git вызывают серьёзные затруднения у новичков. В subversion коммиты представляют собой наборы патчей, а ветка (branch) в терминологии SVN — это всего лишь отдельный каталог в общем дереве файлов. Ветка в git — это совершенно иная сущность, не имеющая ничего общего с ветками в subversion.

По сути ветка в git является динамическим тегом, меткой, которая указывает на конкретный коммит, но значение этой метки при каждом новом коммите автоматически меняется на новое. Как и тег, ветка является внешней сущностью для базы объектов и может указывать на любой коммит и никак не зависит от предыдущих коммитов в ветке.

В нашем образцовом репозитории веткой по умолчанию является v4-dev, её полное имя — refs/heads/v4-dev (да, здесь очередной пример неконсистентности внутренней структуры git, категория для веток в иерархии компонентов полного имени референса называется не branches, а heads). Посмотрим на последний коммит в ветке v4-dev:

## **Специальный референс: HEAD**

HEAD — это специальный референс, содержащий идентификатор коммита, на который в данный момент переключена рабочая копия. При каждом переключении его значение меняется, также он обновляется при коммите.

Фраза detached HEAD означает, что в рабочей копии репозитория находится сейчас отдельный коммит, а не ветка. Если вы сделаете какие-либо изменения и закоммитите их, то получите «потерянный» коммит, не привязанный ни к какой ветке, вы можете легко его потерять, у вас нет простого способа на него снова переключиться и так далее.

Референс HEAD удобно использовать везде, где нужно ссылаться на текущий актуальный коммит в рабочей копии.

## **Ветка master**

Ветка с названием master — это исторически сложившаяся традиция, master можно спокойно переименовать или даже удалить. Эта ветка никак особо не обрабатывается git и совершенно равноправна по сравнению с другими.

Обычно master используется как главная ветка проекта.

## Удаление ветки не удаляет коммиты

Ветка по сути является внешней динамически изменяемой меткой, которая перемещается по графу репозитория при каждом новом коммите. Таким образом, удаление ветки всего лишь удаляет эту метку, не трогая сами коммиты.

Если вы случайно удалили ветку, то ваши коммиты до какого-то времени останутся в локальном репозитории, однако со временем сборщик мусора их удалит, если на них не будет ссылок со стороны тегов или других веток.

Кроме того, ветки — очень «лёгкие», это всего лишь ссылка на коммит и практически не занимает места.

Взаимодействие с внешним репозиторием.

После клонирования удаленного репозитория создается его полная копия со всеми коммитами и ветками. Но удаленный репозиторий постоянно обновляется другими разработчиками и нужно переодически забирать себе изменения.

Операция fetch — не деструктивная, её можно безопасно запускать практически в любое время, она не вносит никаких изменений в локальную рабочую копию. Локальные ветки никак не будут затронуты этой оперрацией. Референсы указывающие на удаленные ветки будут обновлены (передвинуты на их актуальный коммит). Если серверная ветка была удалена, это не приверет к удалению ее локальной копии.

Еще раз. Это абсолютно безопасная операция. В отличии от pull.

Pull это комбинация 2-х операций, fetch+merge.

Вторая операция может привести к конфликтам, в этом случае пользователь должен будет сам исправить конфликт.

**Важно:** pull лучше делать на чистых исходниках, если захотите откатить результат merge то все что было не закомичено - потерячеете. А лучше вообще pull не делать.

Делайте отдельно fetch и merge или rebase в зависимости от ситуации

Ветки. Локальные и удалённые

HEAD and origin/HEAD

DETACHED HEAD — не коммитьте в этой ситуации

Rebase vs Merge.

Вообще коммада Rebase дает широкие возможности изменений истории.

Например, поменять сообщение коммита, поменять коммиты местами, Объединить несколько коммитов в один.

TortoiseGit из всего перечисленного позволяет только Объединить несколько коммитов в один. Но консоль дает куда большие возможности.

Есть еще один способ изменения истории, самого последнего коммита. Опция ammed позволяет докомитить изменения в предыдущий коммит. Это бывает полезно если после коммита выяснилось что закомичены ненужные изменения или наоборот какой то файл был пропущен, кроме того можно изменить сообщение.