Системы контроля версий

Начнём с того, зачем системы контроля версий (version control systems, VCS) нужны. Допустим, вы пишете программу, она работает, потом вы вносите в неё какие-то изменения — она перестаёт работать, вы не помните, что поменяли и не понимаете, почему программа перестала работать. Или вы пишете решение задачи, отправляете её мне, я говорю, что что-то надо поправить, вы правите, отсылаете, я говорю, что надо поправить что-то ещё, вы с другого компьютера качаете старую программу с почты, что-то правите, отправляете, оказывается, что вы скачали и правили не ту программу, вам приходится всё переделать. Или вы приходите на занятие, и понимаете, что все ваши задачи остались на домашнем компьютере и вы не можете ничего сдать. Или, скажем, делали вы кусорт со встроенной сортировкой пузырьком, через некоторое время поняли, что самим вам это не сделать и привлекли к работе, товарища. Товарищ что-то написал, вы что-то написали — как вам синхронизировать свой код? Надо таскать друг другу исходники на флешке или посылать куски кода в аську/джаббер/скайп, вручную вставлять изменения, следить, чтобы ничего не затёрлось, а ещё остаётся та же проблема — всё ВНЕЗАПНО может перестать работать. А теперь представьте, что вы не решаете домашнюю задачку размером в 70 строк и суммарной трудоёмкостью 4 человекочаса, а работаете над проектом в группе из, скажем, 10 человек сроком порядка года и объёмом исходников порядка 70000 строк.

Как такие проблемы можно решать и как необразованные, но мудрые люди их обычно решают — они получают какую-либо работающую версию программы, копируют её в отдельную папочку, пишут дальше, получают ещё что-то работающее — копируют в другую папочку и т.д. пока весь винчестер не окажется забит ненужными исходниками или они не запутаются в изобилии папочек или не замучаются искать нужную, если что-то пошло не так, и т.д. Образованные люди пользуются системами контроля версий.

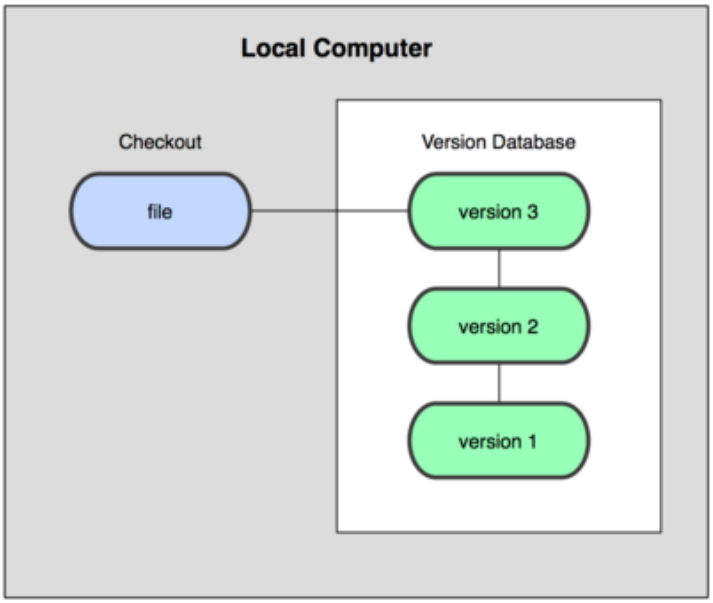


Схема локальной VSC

Система контроля версий — программа, специально предназначенная для работы с изменяющимися документами. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости, возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое. Как обычно устроены системы контроля версий: есть некое хранилище, где-нибудь в доступном всем участникам проекта месте, типа интернета или локальной сети, чтобы версионируемые документы можно было получить с любого компа и с любого компа отправить. В этом хранилище хранятся все документы, причём для каждого документа хранится каждая его версия. То есть, если кто-то поменял что-то в документе, создаётся его новая версия, но старая тоже сохраняется, и её при желании можно получить. Естественно, хранятся не версии документов целиком, а только изменения по отношению к предыдущей версии (это называется дельта-компрессией, а сами изменения — дельтой). Имея исходный документ и набор дельт для всех изменений можно построить любую версию документа.

Для того, чтобы внести изменения в версионируемый документ, его надо сначала получить из хранилища, скопировав куда-то к себе. Такая копия документа называется рабочей копией. Пользователь вносит в документ какие-то изменения и помещает документ обратно в хранилище, где создаётся новая версия для этого документа. Операции, связанные с версионированием, у пользователя обычно делаются какими-то клиентскими программами. В принципе, версионированием может заниматься сама программа, с которой работает пользователь, без явных операций получения/отправки копий документа, например Google Docs создаёт новую версию всякий раз, когда сохраняет изменения в документе, ещё некоторые версии борландовских сред разработки хранили где-то 20 последних версий редактируемых файлов во временных файлах.

Систем контроля версий довольно много, интересующиеся пусть посмотрят страничку в Википедии. Они делятся на два больших класса — централизованные и распределённые. Централизованные системы имеют единый сервер, который хранит версионируемые документы, и с ним синхронизируются все пользователи. Самая распространенная централизованная система на данный момент — это subversion.

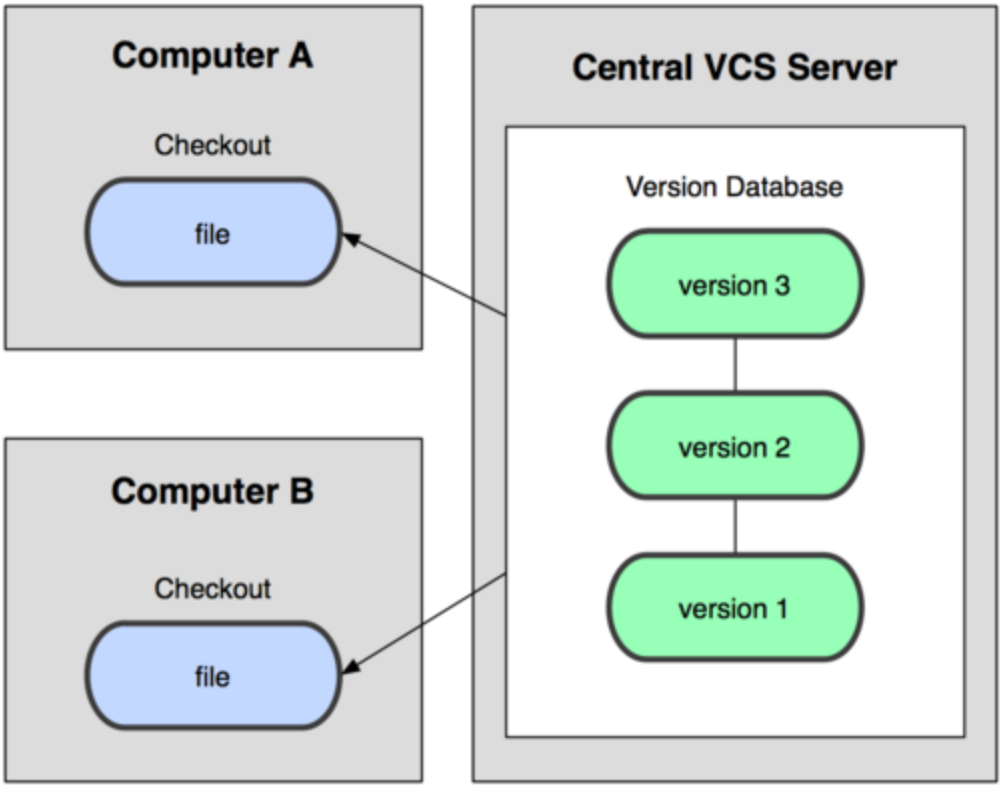


Схема централизованного управления версиями

Распределённые системы центрального хранилища не имеют, у каждого пользователя помимо рабочей копии существует свой репозиторий, и пользователи синхронизируются непосредственно друг с другом. Естественно, она запоминает локальные изменения и может грамотно применить исправления из другого хранилища. Из распределенных сейчас популярны git и Mercurial. Вот гитом-то мы и будем пользоваться.

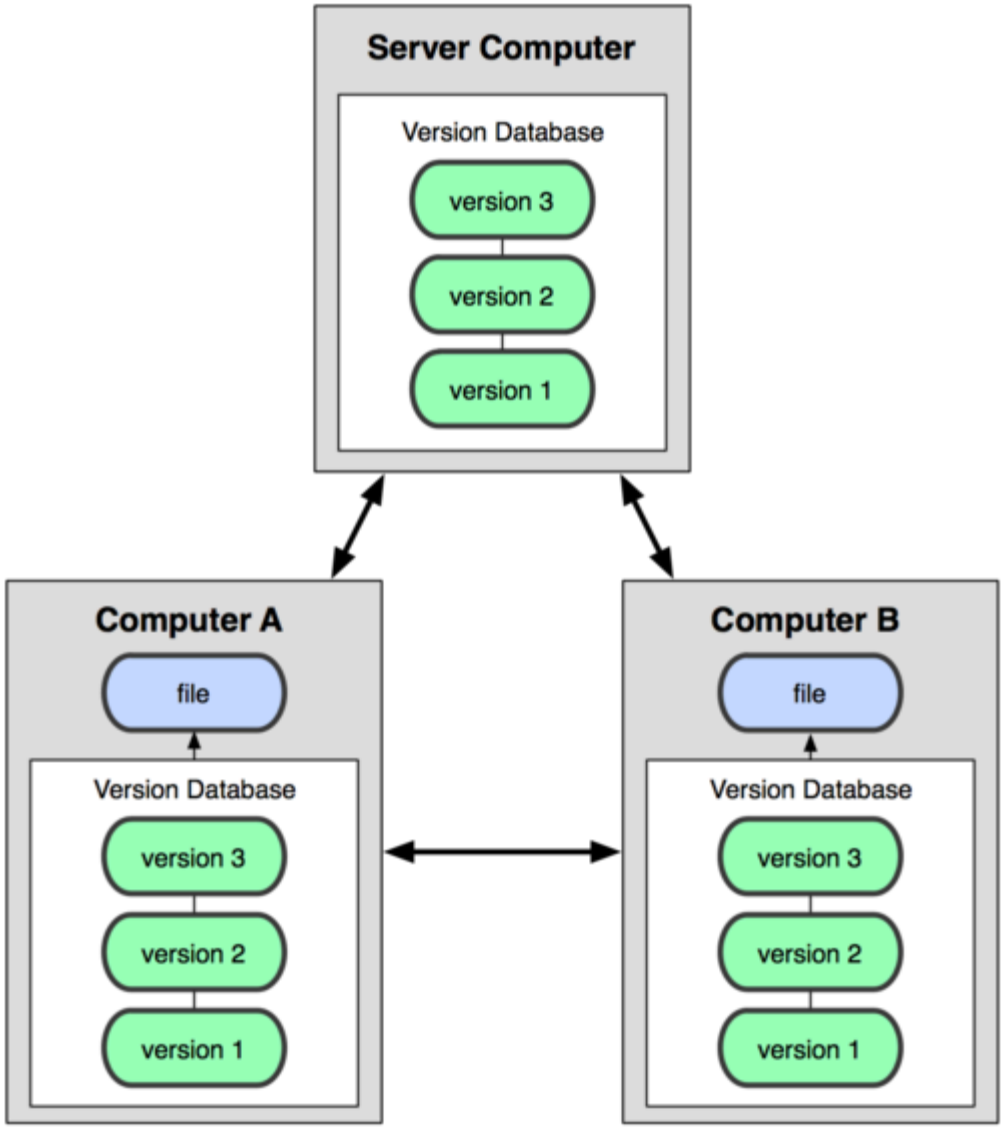


Схема распределённого управления версиями

Для совершения большинства операций в git необходимы только локальные файлы и ресурсы, т.е. обычно информация с других компьютеров в сети не нужна. В централизованных системах практически на каждую операцию накладывается сетевая задержка, а в git вся история проекта хранится локально у вас на диске, и большинство операций выглядят практически мгновенными.

Кроме того, работа локально означает, что мало чего нельзя сделать без доступа к сети. Например, если вы сидите на паре, или у вас просто отвалился wi-fi, вы все равно сможете спокойно делать коммиты и отправить их в интернет, как только станет доступна сеть. Во многих других системах это невозможно, или же крайне неудобно. Например, работая с Subversion и CVS, вы можете редактировать файлы, но сохранить изменения в вашу базу данных нельзя (потому что она отключена от репозитория).

Перед сохранением любого файла git вычисляет контрольную сумму, и она становится индексом этого файла. Поэтому невозможно изменить содержимое файла

или каталога так, чтобы git не узнал об этом. Эта функциональность встроена в сам фундамент git и является важной составляющей его философии. Если информация потеряется при передаче или повредится на диске, git всегда это выявит. Работая с git, вы будете постоянно встречать эти хеши, поскольку они широко используются. Фактически, в своей базе данных git сохраняет всё не по именам файлов, а по хешам их содержимого.

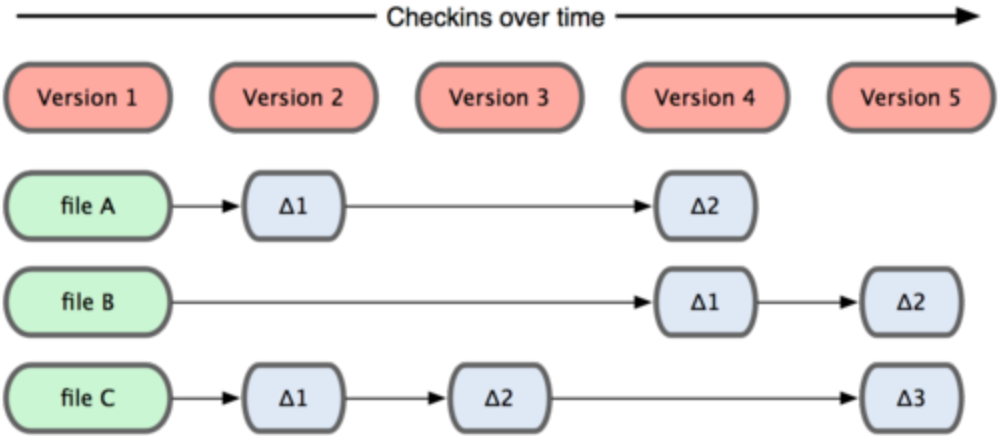
Практически все действия, которые вы совершаете в Git, только добавляют данные в базу. Довольно сложно заставить систему удалить данные или сделать что-то неотменяемое. Можно, как и в любой другой CVS, потерять данные, которые вы ещё не сохранили, но как только они зафиксированы, их сложно потерять, особенно если вы регулярно отправляете изменения в другой репозиторий.

Основные понятия VCS

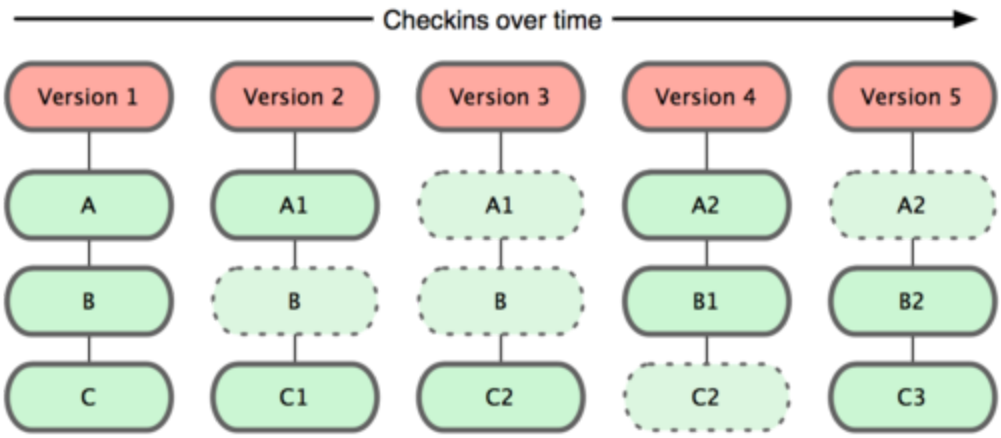
Терминология у всех систем разная, причём иногда в одной системе может использоваться разная терминология для одного и того же действия, впрочем, запутаться всё равно невозможно. Итак, основные термины:

Репозиторий — хранилище версионируемых документов. Команда **git clone** позволяет полностью скопировать весь репозиторий (например, с удаленного сервера к вам на компьютер).

Revision/version/changeset — версия документа или всего репозитория. SVN, например, версионирует весь репозиторий, то есть если вы меняете один или несколько файлов и отправляете изменения на сервер, номер текущей ревизии продвигается для всех документов в репозитории, даже не изменившихся. Собственно, поэтому и changeset — набор изменений.



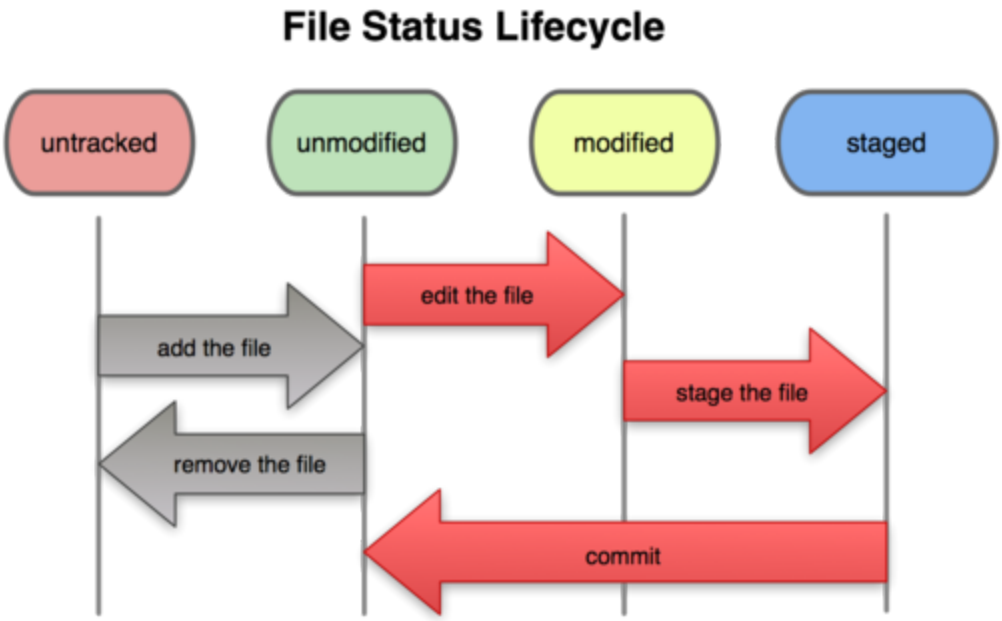
А вот в гите “единицей обработки” является набор изменений, или патч, а не файлы или каталоги. гит считает хранимые данные набором слепков небольшой файловой системы. Каждый раз, когда вы фиксируете текущую версию проекта, гит, по сути, сохраняет слепок того, как выглядят все файлы проекта на текущий момент. Ради эффективности, если файл не менялся, гит не сохраняет файл снова, а делает ссылку на ранее сохранённый файл. То, как гит подходит к хранению данных, похоже на следующий рисунок:



Это важное отличие гита от практически всех других систем управления версиями. Git больше похож на небольшую файловую систему с мощными инструментами, работающими поверх неё, чем на просто систему контроля версий. Из-за него гит вынужден пересмотреть практически все аспекты управления версиями, которые другие системы взяли от своих предшественниц. Вам то, понятное дело, всё равно, но вот те, кто много лет просидел под CVS, SVN и тому подобными VCS, довольно сложно морально понять и принять гит.

Каждый файл в вашем рабочем каталоге может находиться в одном из двух состояний: под версионным контролем (отслеживаемые) и нет (неотслеживаемые). **Отслеживаемые файлы** — это те файлы, которые были в последнем слепке состояния проекта (snapshot); они могут быть **неизмененными**, **измененными** или **подготовленными к коммиту/проиндексированными** (staged). **Неотслеживаемые файлы** — это всё остальное, любые файлы в вашем рабочем каталоге, которые не входили в ваш последний слепок состояния и не подготовлены к коммиту. Когда вы впервые клонируете репозиторий, все файлы будут отслеживаемыми и неизмененными, потому что вы только взяли их из хранилища (checked them out) и ничего пока не редактировали.

Как только вы отредактируете файлы, git будет рассматривать их как измененные, т.к. вы изменили их с момента последнего коммита. Вы индексируете (stage) эти изменения и затем фиксируете все индексированные изменения, а затем цикл повторяется.



Для того чтобы начать отслеживать (добавить под версионный контроль) новый файл, используется команда **git add**. Данная команда находит и подготавливает к фиксации (**commit'y**) сделанные в проекте или его части изменения (то есть её название "добавить" относится не к файлам, а к сделанным изменениям, которые добавляются в индекс для последующей фиксации/коммите). Это важная особенность гита, потому что svn, например, оперирует файлами целиком. В гите же возможно часть изменений внутри одного файла закоммитить, а часть оставить на будущее.

Команда **git status** позволяет узнать, какие файлы в каком состоянии находятся. Для того, чтобы узнать, что конкретно поменялось, а не только какие файлы были изменены — вы можете использовать команду **git diff**. Вы, скорее всего, будете использовать эти команды для получения ответов на два вопроса: что вы изменили, но еще не проиндексировали, и что вы проиндексировали и собираетесь коммитить. Если git status отвечает на эти вопросы слишком обобщенно, то git diff показывает вам непосредственно добавленные и удаленные строки — собственно патч (patch). Важно отметить, что git diff сама по себе не показывает все изменения сделанные с последнего коммита — только те, что еще не проиндексированы. Такое поведение может сбивать с толку, так как если вы проиндексируете все свои изменения, то git diff ничего не вернет.

Чтобы сделать коммит (фиксацию изменений), нужно выполнить команду **git commit**. Эта команда откроет выбранный вами текстовый редактор, в котором нужно будет написать комментарий к данному коммиту. При использовании графических виндовых утилит типа TortoiseGit у вас откроется специальное окошко для ввода комментария. Комментарии к коммитам — это очень хороший способ быстро понять, что было сделано в том или ином коммите. Т.е. если вы разрабатываете большую систему и хотите найти какой-то давний коммит, то это будет гораздо проще сделать, если у него будет осмысленный комментарий. Таким образом, у нас коммиты с пустым или неадекватным текстом в комменте будут страшно караться. В комменте не

нужно писать, какие файлы поменялись (это и так видно по диффу), пишите, **почему** они поменялись — что вы такого сделали, как изменилась логика, что добавилось/удалилось и т.п.

Очень типична (особенно поначалу) ситуация, когда хочется поправить последний коммит (например, забыли добавить какие-то файлы или написали плохой комментарий). Это можно сделать, выполнив commit с опцией --amend:

```
$ git commit --amend
```

Эта команда берёт индекс и использует его для коммита. Если после последнего коммита не было никаких изменений (например, вы запустили приведённую команду сразу после предыдущего коммита), то состояние проекта будет абсолютно таким же и всё, что вы измените это комментарий к коммиту. Появится всё тот же редактор для комментариев к коммитам, но уже с введённым комментарем к последнему коммиту. Вы можете отредактировать это сообщение так же как обычно, и оно перепишет предыдущее.

Для того чтобы удалить файл из git, вам необходимо удалить его из отслеживаемых файлов (точнее, удалить его из вашего индекса) а затем выполнить коммит. Это позволяет сделать команда **git rm**, которая также удаляет файл из вашего рабочего каталога, так что вы в следующий раз не увидите его как неотслеживаемый.

Историю коммитов позволяет увидеть команда **git log**. У нее есть много всяких параметров, которые позволяют красиво форматировать вывод истории. У меня, например, есть алиас на команду git log --pretty=format:\"%h %ad | %s%d [%an]\" --graph --date=short, который удобным образом показывает хэш и дерево коммитов, дату, комментарий и автора коммита. Всякие TortoiseGit позволяют рисовать красивые картинки. На github, которым мы будем пользоваться, тоже есть графопостроитель для красивого отображения дерева коммитов.

Чтобы откатить локальные непроиндексированные изменения (поправили файл и поняли, что лучше бы этого не делать и вернуть как было), нужно выполнить команду **git checkout [имяфайла]**.

Работа с удалёнными репозиторями

Чтобы иметь возможность совместной работы над каким-либо git-проектом, необходимо знать как управлять удалёнными репозиториями. Удалённые репозитории — это модификации проекта, которые хранятся в интернете или ещё где-то в сети. Их может быть несколько, каждый из которых как правило доступен для вас либо только на чтение, либо на чтение и запись. Совместная работа включает в себя управление удалёнными репозиториями и **помещение/пуш** (push) и **получение/пулл** (pull) данных в и из них тогда, когда нужно обменяться результатами работы. Управление удалёнными репозиториями включает умение добавлять удалённые репозитории, удалять те из них, которые больше не действуют, умение управлять различными удалёнными ветками и определять их как ослеживаемые (tracked) или нет и прочее. В нашем случае у каждого из вас будет репозиторий на вашем домашнем компьютере, еще один на компьютере в университете (если захотите тут покодить), и еще один на серверах гитхаба. И вот друг для друга все они будут удаленными, т.е. будет возможен обмен информацией между ними (ну чисто теоретически, поскольку на компьютер в универе нельзя будет добраться из дома).

Чтобы просмотреть какие удалённые серверы у вас уже настроены, следует выполнить команду **git remote**. Она перечисляет список имён-сокращений для всех уже указанных удалённых дескрипторов. Если вы склонировали ваш репозиторий, у вас должен отобразиться по крайней мере origin — это имя по умолчанию, которое git присваивает серверу, с которого вы этот репозиторий склонировали.

Когда ваш проект достигает момента, когда вы хотите поделиться своими наработками, вам необходимо **отправить** (push) их в главный репозиторий. Команда для этого действия простая: git push [удаленный_сервер] [ветка]. Чтобы отправить вашу ветку master на сервер origin (клонирование как правило настраивает оба этих имени автоматически), вы можете выполнить следующую команду для отправки наработок обратно на сервер:

$ git push origin master

Эта команда срабатывает только в случае, если вы клонировали с сервера, на котором у вас есть права на запись, и если никто другой с тех пор не выполнял команду push. Если вы и кто-то ещё одновременно клонируете, затем он выполняет команду push, а затем команду push выполняете вы, то ваш push точно будет отклонён. Вам придётся сначала запуллить (pull) их изменения и объединить с вашими. Только после этого вам будет позволено выполнить push.

Для получения данных из удалённых репозиториев, следует выполнить **git fetch [remote-name]**. Данная команда связывается с указанным удалённым репозиторием и забирает все те данные проекта, которых у вас ещё нет. После того как вы выполнили команду, у вас должны появиться ссылки на все ветки из этого удалённого проекта. В частности, теперь эти ветки в любой момент могут быть просмотрены или слиты (про это я вам как-нибудь потом расскажу, пока можно просто всегда делать pull).

Когда вы клонируете репозиторий, команда clone автоматически добавляет этот удалённый репозиторий под именем origin. Таким образом git fetch origin извлекает все наработки, отправленные (push) на этот сервер после того, как вы склонировали его (или получили изменения с помощью fetch). Важно отметить, что команда fetch забирает данные в ваш локальный репозиторий, но не сливает их с какими-либо вашими наработками, и не модифицирует то, над чем вы работаете в данный момент. Вам необходимо вручную слить эти данные с вашими, когда вы будете готовы.

Если у вас есть ветка, настроенная на отслеживание удалённой ветки, то вы можете использовать команду **git pull**. Она автоматичеси извлекает и затем сливает данные из удалённой ветки в вашу текущую ветку. Этот способ может для вас оказаться более простым или более удобным. К тому же по умолчанию команда git clone автоматически настраивает вашу локальную ветку master на отслеживание удалённой ветки master на сервере, с которого вы клонировали (подразумевается, что на удалённом сервере есть ветка master). Выполнение git pull как правило извлекает (fetch) данные с сервера, с которого вы изначально склонировали, и автоматически пытается **слить** (merge) их с кодом, над которым вы в данный момент работаете (про **git branch** и **git merge** сами почитайте или подробно я вам потом как-нибудь расскажу, пока не пригодится все равно).

Тут и начинается самое интересное — представьте, что вы что-то поменяли у себя в своей рабочей копии, а в это время кто-то ещё поменял что-то на сервере. Если изменения были в разных файлах, то система контроля версий просто скопирует новые файлы поверх старых неизменившихся, а изменённые вами файлы не тронет.

Если изменения были в одном файле, но в разных строчках, система сможет "смерджить" файл — поменять только те строчки, что изменились на сервере, сохранив ваши изменения. Если изменения были в одной и той же строчке, система не сможет понять, что надо сделать — то ли изменения на сервере свежее/правильнее локальных, то ли наоборот, то ли надо оставить оба изменения, но в любом случае сказать что-то пользователю надо, потому как высока вероятность, что изменилось что-то важное, над чем пользователь работал. Такая ситуация называется конфликтом. Конфликты разрешаются ручками.

Хорошие практики использования VCS

- выкладывать туда только исходный код и ресурсы типа картинок, конфигов, может быть тестовые файлы. Никогда не выкладывать файлы, которые могут быть сгенерены из тех, что есть в репозитории, в т.ч. .exe, .obj, .dll и т.п.
- см. также п. 2 литературы

Литература:

1. Википедия — общие сведения о VCS
2. Работа с git — мануал про то, как работать с гитом. В картинках.
3. Git magic — книжка про гит
4. Pro git — русский перевод известной книги. Просто и понятно обо всем, много букв.