Система управления версиями

Общие сведения

Система управления версиями (от англ. `Version Control System`, `VCS` или `Revision Control System`) — программное обеспечение для облегчения работы с изменяющейся информацией. 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.

Такие системы наиболее широко используются при разработке программного обеспечения для хранения исходных кодов разрабатываемой программы. Однако они могут с успехом применяться и в других областях, в которых ведётся работа с большим количеством непрерывно изменяющихся электронных документов

Ситуация, в которой электронный документ за время своего существования претерпевает ряд изменений, достаточно типична. При этом часто бывает важно иметь не только последнюю версию, но и несколько предыдущих. В простейшем случае можно просто хранить несколько вариантов документа, нумеруя их соответствующим образом. Такой способ неэффективен (приходится хранить несколько практически идентичных копий), требует повышенного внимания и дисциплины и часто ведёт к ошибкам, поэтому были разработаны средства для автоматизации этой работы.

Традиционные системы управления версиями используют централизованную модель, когда имеется единое хранилище документов, управляемое специальным сервером, который и выполняет бóльшую часть функций по управлению версиями. Пользователь, работающий с документами, должен сначала получить нужную ему версию документа из хранилища; обычно создаётся локальная копия документа, так называемая “рабочая копия”. Может быть получена последняя версия или любая из предыдущих, которая может быть выбрана по номеру версии или дате создания, иногда и по другим признакам. После того, как в документ внесены нужные изменения, новая версия помещается в хранилище. В отличие от простого сохранения файла, предыдущая версия не стирается, а тоже остаётся в хранилище и может быть оттуда получена в любое время. Сервер может использовать т. н. дельта-компрессию - такой способ хранения документов, при котором сохраняются только изменения между последовательными версиями, что позволяет уменьшить объём хранимых данных. Поскольку обычно наиболее востребованной является последняя версия файла, система может при сохранении новой версии сохранять её целиком, заменяя в хранилище последнюю ранее сохранённую версию на разницу между этой и последней версией. Некоторые системы (например, `ClearCase`) поддерживают сохранение версий обоих видов: большинство версий сохраняется в виде дельт, но периодически (по специальной команде администратора) выполняется сохранение версий всех файлов в полном виде; такой подход обеспечивает максимально полное восстановление истории в случае повреждения репозитория.

Иногда создание новой версии выполняется незаметно для пользователя (прозрачно), либо прикладной программой, имеющей встроенную поддержку такой функции, либо за счёт использования специальной файловой системы. В этом случае пользователь просто работает с файлом, как обычно, и при сохранении файла автоматически создаётся новая версия.

Часто бывает, что над одним проектом одновременно работают несколько человек. Если два человека изменяют один и тот же файл, то один из них может случайно отменить изменения, сделанные другим. Системы управления версиями отслеживают такие конфликты и предлагают средства их решения. Большинство систем может автоматически объединить (слить) изменения, сделанные разными разработчиками. Однако такое автоматическое объединение изменений, обычно, возможно только для текстовых файлов и при условии, что изменялись разные (непересекающиеся) части этого файла. Такое ограничение связано с тем, что большинство систем управления версиями ориентированы на поддержку процесса разработки программного обеспечения, а исходные коды программ хранятся в текстовых файлах. Если автоматическое объединение выполнить не удалось, система может предложить решить проблему вручную.

Часто выполнить слияние невозможно ни в автоматическом, ни в ручном режиме, например, если формат файла неизвестен или слишком сложен. Некоторые системы управления версиями дают возможность заблокировать файл в хранилище. Блокировка не

позволяет другим пользователям получить рабочую копию или препятствует изменению рабочей копии файла (например, средствами файловой системы) и обеспечивает, таким образом, исключительный доступ только тому пользователю, который работает с документом.

Многие системы управления версиями предоставляют ряд других возможностей:

Позволяют создавать разные варианты одного документа, т. н. ветки, с общей историей изменений до точки ветвления и с разными - после неё.
Дают возможность узнать, кто и когда добавил или изменил конкретный набор строк в файле.
Ведут журнал изменений, в который пользователи могут записывать пояснения о том, что и почему они изменили в данной версии.
Контролируют права доступа пользователей, разрешая или запрещая чтение или изменение данных, в зависимости от того, кто запрашивает это действие.

История развития `Git`

О данной `VCS` будет идти речь далее, поэтому здесь приведена только историческая справка.

Разработка ядра `Linux` велась на проприетарной системе `BitKeeper`, которую автор, - Ларри Маквой, сам разработчик `Linux`, - предоставил

проекту по бесплатной лицензии. Разработчики, высококлассные программисты, написали несколько утилит, и для одной Эндрю Триджелл произвел реверс-инжиниринг формата передачи данных `BitKeeper`. В ответ Маквой обвинил разработчиков в нарушении соглашения и отозвал лицензию, и Торвальдс взялся за новую систему: ни одна из открытых систем не позволяла тысячам программистов кооперировать свои усилия (тот же конфликт привёл к написанию `Mercurial`). Идеология была проста: взять подход `CVS` и перевернуть с ног на голову, и заодно добавить надёжности.

Начальная разработка велась меньше, чем неделю: 3 апреля 2005 года разработка началась, и уже 7 апреля код `Git` управлялся неготовой системой. 16 июня `Linux` был переведён на `Git`, а 25 июля Торвальдс отказался от обязанностей ведущего разработчика.

Торвальдс так саркастически отозвался о выбранном им названии `git` (что на английском сленге означает “мерзавец”): “Я эгоистичный ублюдок, и поэтому называю все свои проекты в честь себя. Сначала `Linux`, теперь `git`.”

Более детальную информацию о `VCS`, версионность, принцип паботы - можно почитать с татье `WiKi` Git

Терминология

Общепринятой терминологии не существует, в разных системах могут использоваться различные названия для одних и тех же действий. Ниже приводятся некоторые из наиболее часто используемых

вариантов.

1. `amend` - внести изменения, не создавая новой версии - обычно когда разработчик ошибочно зафиксировал (`commit`) версию, но не залил (`push`) её на сервер.
2. `blame` - понять, кто внёс изменение.
3. `branch` - направление разработки, независимое от других. Ветвь представляет собой копию части (как правило, одного каталога) хранилища, в которую можно вносить свои изменения, не влияющие на другие ветви. Документы в разных ветвях имеют одинаковую историю до точки ветвления и разные - после неё.
4. `changeset, changelist, activity` - набор изменений. Представляет собой поименованный набор правок, сделанных в локальной копии для какой-то общей цели. В системах, поддерживающих наборы правок, разработчик может объединять локальные правки в группы и выполнять фиксацию логически связанных изменений одной командой, указывая требуемый набор правок в качестве параметра. При этом прочие правки останутся незафиксированными. Типичный пример: ведётся работа над добавлением новой функциональности, а в этот момент обнаруживается критическая ошибка, которую необходимо немедленно исправить. Разработчик создаёт набор изменений для уже сделанной работы и новый - для исправлений. По завершении исправления ошибки отдаётся команда фиксации только второго набора правок.
5. `check-in, commit, submit` - создание новой версии, фиксация

изменений. В некоторых `VCS` (`Subversion`) - новая версия автоматически переносится в хранилище документов.

6. `check-out, clone` - извлечение документа из хранилища и создание рабочей копии.
7. `conflict` - ситуация, когда несколько пользователей сделали изменения одного и того же участка документа. Конфликт обнаруживается, когда один пользователь зафиксировал свои изменения, а второй пытается зафиксировать и система сама не может корректно слить конфликтующие изменения. Поскольку программа может быть недостаточно разумна для того, чтобы определить, какое изменение является “корректным”, второму пользователю нужно самому разрешить конфликт (`resolve`).
8. `graft, backport, cherry-picking, transplant` - использовать встроенный в `VCS` алгоритм слияния, чтобы перенести отдельные изменения в другую ветвь, не сливая их. Например, исправили ошибку в экспериментальной ветви — вносим эти же изменения в стабильный ствол.
9. `head` - основная версия - самая свежая версия для ветви/ствола, находящаяся в хранилище. Сколько ветвей, столько основных версий.
10. `merge, integration` - слияние - объединение независимых изменений в единую версию документа. Осуществляется, когда два человека изменили один и тот же файл или при переносе изменений из одной ветки в другую.
11. `pull, update` - получить новые версии из хранилища. В некоторых `VCS` (`Subversion`) - происходит и `pull`, и `switch`, то есть загружаются изменения, а потом рабочая копия доводится до последнего состояния. Будьте внимательны,

понятие update двусмысленно и в `Subversion` и `Mercurial` значит разное.

12. `push` - залить новые версии в хранилище. Многие распределённые `VCS` (`Git`, `Mercurial`) предполагают, что commit надо давать каждый раз, когда программист выполнил какую-то законченную функцию. А залить - когда есть интернет и другие хотят ваши изменения. `Commit` обычно не требует ввода имени и пароля, а `push` - требует.
13. `rebase` - перенос точки ветвления (версии, от которой начинается ветвь) на более позднюю версию основной ветви. Например, после выпуска версии `1.0` проекта в стволе продолжается доработка (исправление ошибок, доработка имеющейся функциональности), одновременно начинается работа над новой функциональностью в новой ветви. Через какое-то время в основной ветви происходит выпуск версии `1.1` (с исправлениями); теперь желательно, чтобы ветвь разработки новой функциональности включала изменения, произошедшие в стволе. Вообще, это можно сделать базовыми средствами, с помощью слияния (`merge`), выделив набор изменений между версиями `1.0` и `1.1` и слив его в ветвь. Но при наличии в системе поддержки перебазирования ветви эта операция делается проще, одной командой: по команде rebase (с параметрами: ветвью и новой базовой версией) система самостоятельно определяет нужные наборы изменений и производит их слияние, после чего для ветви базовой версией становится версия `1.1`; при последующем слиянии ветви со стволом система не рассматривает повторно изменения, внесённые между версиями `1.0` и `1.1`, так как ветвь

логически считается выделенной после версии `1.1`.

14. `repository, depot` - хранилище - место, где система управления версиями хранит все документы вместе с историей их изменения и другой служебной информацией.
15. `revision` - версия документа. Системы управления версиями различают версии по номерам, которые назначаются автоматически.
16. `shelving` - откладывание изменений. Предоставляемая некоторыми системами возможность создать набор изменений (`changeset`) и сохранить его на сервере без фиксации (`commit'a`). Отложенный набор изменений доступен на чтение другим участникам проекта, но до специальной команды не входит в основную ветвь. Поддержка откладывания изменений даёт возможность пользователям сохранять незавершённые работы на сервере, не создавая для этого отдельных ветвей.
17. `strip` - удалить целую ветвь из хранилища.
18. `tag, label` - метка, которую можно присвоить определённой версии документа. Метка представляет собой символическое имя для группы документов, причём метка описывает не только набор имён файлов, но и версию каждого файла. Версии включённых в метку документов могут принадлежать разным моментам времени.
19. `trunk, mainline, master` - основная ветвь разработки проекта. Политика работы со стволом может отличаться от проекта к проекту, но в целом она такова: большинство изменений вносится в ствол; если требуется серьёзное изменение, способное привести к нестабильности, создаётся ветвь, которая сливается со стволом, когда нововведение будет

в достаточной мере испытано; перед выпуском очередной версии создаётся ветвь для последующего выпуска, в которую вносятся только исправления.

20. `update, sync, switch` - синхронизация рабочей копии до некоторого заданного состояния хранилища. Чаще всего это действие означает обновление рабочей копии до самого свежего состояния хранилища. Однако при необходимости можно синхронизировать рабочую копию и к более старому состоянию, чем текущее.
21. `working copy` - рабочая (локальная) копия документов.

Git

`Git` - распределённая система управления версиями. Проект был создан Линусом Торвальдсом для управления разработкой ядра `Linux`, первая версия выпущена 7 апреля 2005 года. На сегодняшний день его поддерживает Джунио Хамано.

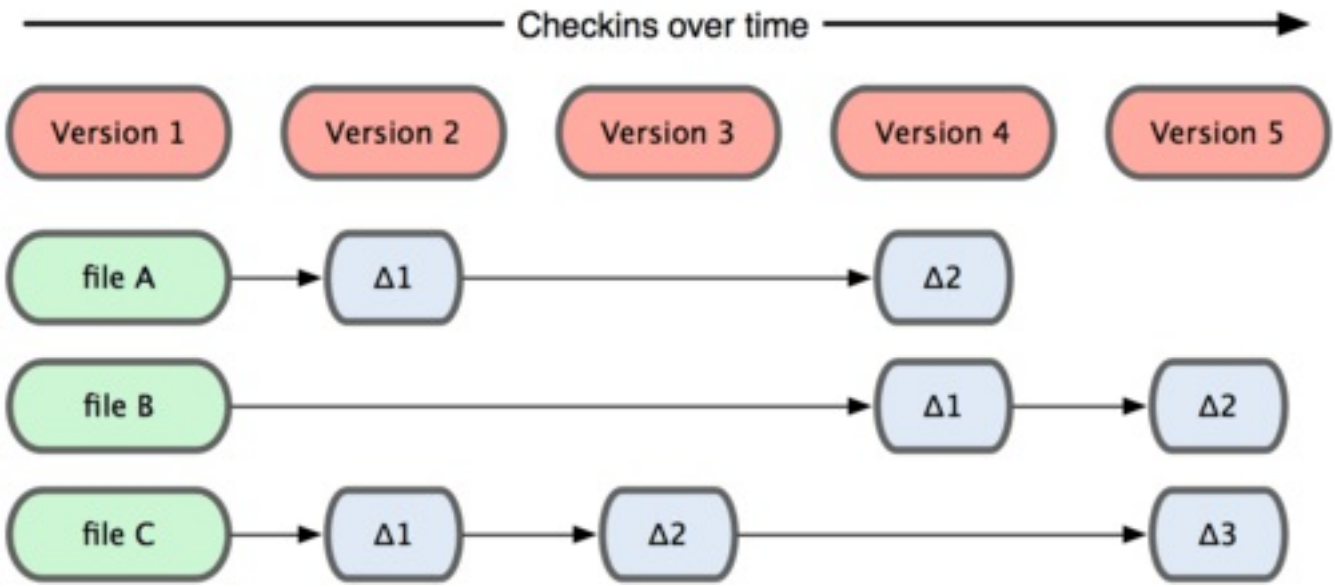
Далее, материал взять с официального ресурса `Git` - `Git-Scm`

Общие сведения

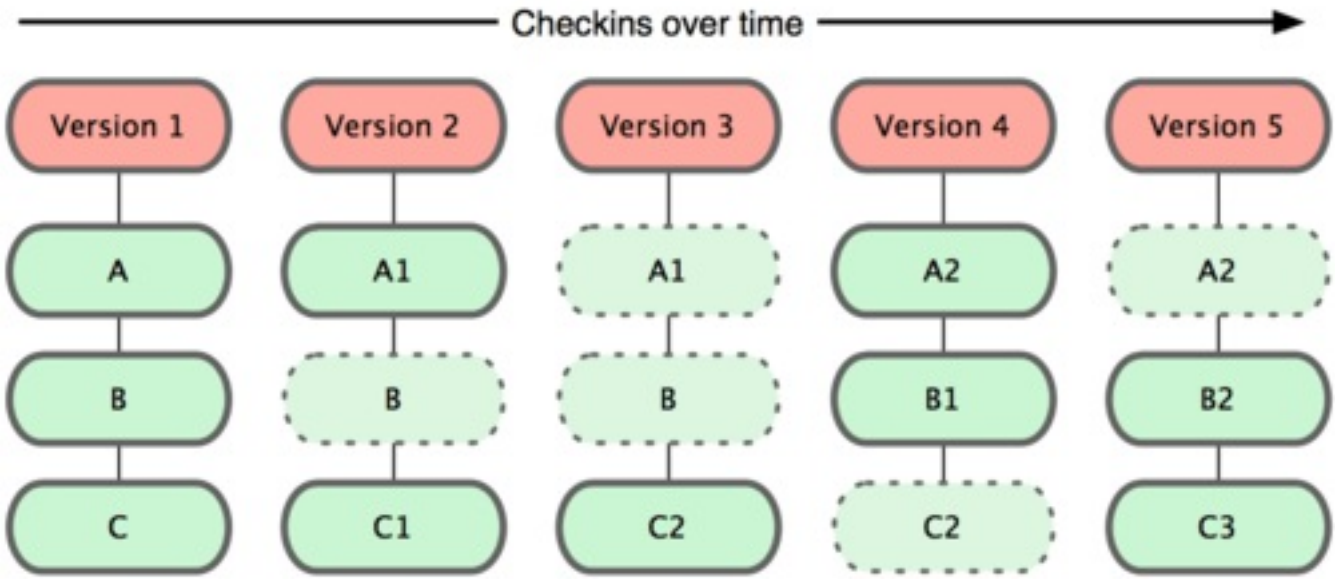
Слепки вместо патчей

Главное отличие `Git`'а от любых других `VCS` (например, `Subversion` и ей подобных) - это то, как `Git` смотрит на свои данные. В принципе, большинство других систем хранит информацию как список изменений (патчей) для файлов. Эти системы (`CVS`, `Subversion`, `Perforce`, `Bazaar` и другие) относятся к хранимым данным как к набору файлов

и изменений, сделанных для каждого из этих файлов во времени, как показано на рисунке



`Git` не хранит свои данные в таком виде. Вместо этого `Git` считает хранимые данные набором слепков небольшой файловой системы. Каждый раз, когда вы фиксируете текущую версию проекта, `Git`, по сути, сохраняет слепок того, как выглядят все файлы проекта на текущий момент. Ради эффективности, если файл не менялся, `Git` не сохраняет файл снова, а делает ссылку на ранее сохранённый файл. То, как `Git` подходит к хранению данных, похоже на рисунок ниже



Это важное отличие `Git`'а от практически всех других систем контроля версий. Из-за него `Git` вынужден пересмотреть практически все аспекты контроля версий, которые другие системы переняли от своих предшественниц. `Git` больше похож на небольшую файловую систему с невероятно мощными инструментами, работающими поверх неё, чем на просто `VCS`. Далее, мы узнаем, какие преимущества даёт такое понимание данных.

Почти все операции локальны

Для совершения большинства операций в `Git`'е необходимы только локальные файлы и ресурсы, т.е. обычно информация с других компьютеров в сети не нужна. Если вы пользовались централизованными системами, где практически на каждую операцию накладывается сетевая задержка, вы, возможно, подумаете, что боги наделили `Git` неземной силой. Поскольку вся история проекта хранится локально у вас на диске, большинство операций кажутся практически мгновенными.

К примеру, чтобы показать историю проекта, `Git`'у не нужно скачивать её с сервера, он просто читает её прямо из вашего локального репозитория. Поэтому историю вы увидите практически мгновенно. Если вам нужно просмотреть изменения между текущей версией файла и версией, сделанной месяц назад, `Git` может взять файл месячной давности и вычислить разницу на месте, вместо того чтобы запрашивать разницу у `VCS`-сервера или качать с него старую версию файла и делать локальное сравнение.

Кроме того, работа локально означает, что мало чего нельзя сделать без доступа к Сети или `VPN`. Если вы в самолёте или в поезде и хотите немного поработать, можно спокойно делать коммиты, а затем отправить их, как только станет доступна сеть. Если вы пришли домой, а `VPN`-клиент не работает, всё равно можно продолжать работать. Во многих других системах это невозможно или же крайне неудобно. Например, используя Perforce, вы мало что можете сделать без соединения с сервером. Работая с `Subversion` и `CVS`, вы можете редактировать файлы, но сохранить изменения в вашу базу данных нельзя (потому что она отключена от репозитория). Вроде ничего серьёзного, но потом вы удивитесь, насколько это меняет дело.

`Git` следит за целостностью данных

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`'а и является важной составляющей его философии. Если информация

потеряется при передаче или повредится на диске, `Git` всегда это выявит.

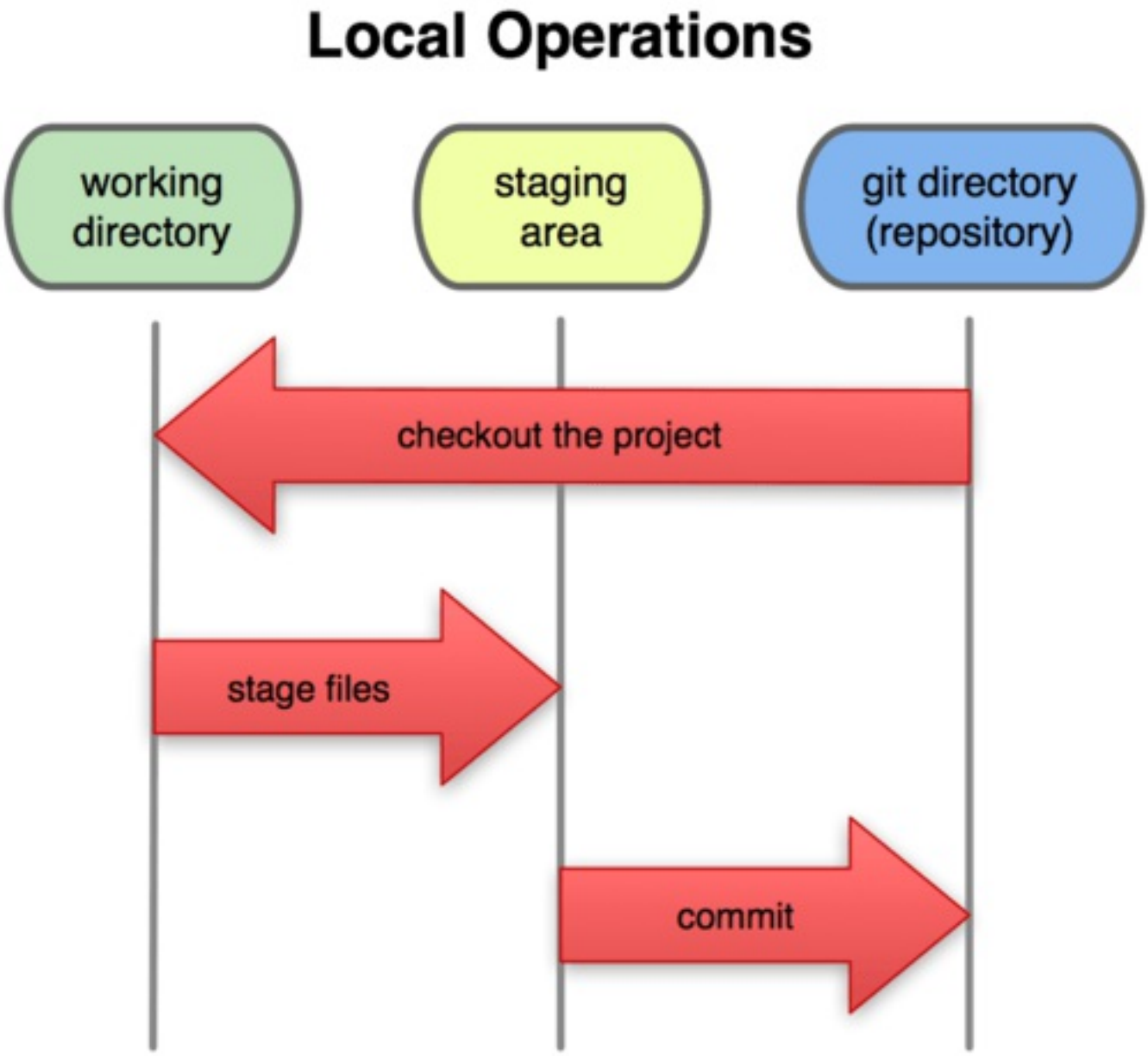
Механизм, используемый `Git`'ом для вычисления контрольных сумм, называется `SHA-1` хешем. Это строка из 40 шестнадцатеричных символов (`0-9` и `a-f`), вычисляемая в `Git`'е на основе содержимого файла или структуры каталога. `SHA-1` хеш выглядит примерно так: `24b9da6552252987aa493b52f8696cd6d3b00373`.

Работая с `Git`'ом, вы будете встречать эти хеши повсюду, поскольку он их очень широко использует. Фактически, в своей базе данных `Git` сохраняет всё не по именам файлов, а по хешам их содержимого.

Три состояния

Самое важное, что нужно помнить про `Git`, если вы хотите, чтобы дальше изучение шло гладко. В `Git`'е файлы могут находиться в одном из трёх состояний: зафиксированном, изменённом и подготовленном. “Зафиксированный” значит, что файл уже сохранён в вашей локальной базе. К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы. Подготовленные файлы - это изменённые файлы, отмеченные для включения в следующий коммит.

Таким образом, в проектах, использующих `Git`, есть три части: каталог `Git`'а (`Git directory`), рабочий каталог (`working directory`) и область подготовленных файлов (`staging area`).



Каталог `Git`'а - это место, где `Git` хранит метаданные и базу данных объектов вашего проекта. Это наиболее важная часть `Git`'а, и именно она копируется, когда вы клонируете репозиторий с другого компьютера.

Рабочий каталог - это извлечённая из базы копия определённой версии проекта. Эти файлы достаются из сжатой базы данных в каталоге `Git`'а и помещаются на диск для того, чтобы вы их просматривали и редактировали.

Область подготовленных файлов - это обычный файл, обычно хранящийся в каталоге `Git`'а, который содержит информацию о том, что должно войти в следующий коммит. Иногда его называют индексом (`index`), но в последнее время становится стандартом называть его областью подготовленных файлов (`staging area`).

Стандартный рабочий процесс с использованием `Git`'а выглядит примерно так:

1. Вы вносите изменения в файлы в своём рабочем каталоге.
2. Подготавливаете файлы, добавляя их слепки в область подготовленных файлов.
3. Делаете коммит, который берёт подготовленные файлы из индекса и помещает их в каталог `Git`'а на постоянное хранение.

Если рабочая версия файла совпадает с версией в каталоге `Git`'а, файл считается зафиксированным. Если файл изменён, но добавлен в область подготовленных данных, он подготовлен. Если же файл изменился после выгрузки из БД, но не был подготовлен, то он считается изменённым.

Команды. Общая работа.

Конфигурирование

Установка имени пользователя, под которым будут сохраняться изменения

```
git config --global user.name "[name]"
```

Установка мыла

```
git config --global user.email "[email address]"
```

Включение подсветки терминала при работе с `Git`

```
git config --global color.ui auto
```

Создание репозитория

Создание нового репозитория и назначение ему имени `project-name`

```
git init [project-name]
```

Создание репозитория в текущей директории

```
git init .
```

Клонирование удаленного репозитория, по `url`. При этом будет создана папка с именем репозитория в которую будет скопировано содержимое.

```
git clone [url]
```

Изменения

Просмотр текущих изменений в репозитории

```
git status
```

Просмотр что было изменено в файлах репозитория не попавших в индекс

```
git diff
```

Добавление файла в индекс

```
git add [file]
```

Просмотр изменений, которые попали в индекс

```
git diff --staged
```

Удаление файла из индекса

```
git reset [file]
```

Запись слепка с сообщением (коммит)

```
git commit -m "[descriptive message]"
```

`Branch`

Вывод локальных веток репозитория. При добавлении аргумента `-r`, будет выведены все удаленные ветки.

```
git branch
```

Создание новой ветки с именем `branch-name`

```
git branch [branch-name]
```

Переключение на ветку `branch-name`. Если добавить аргумент `-b` будет создана ветка `branch-name` и сделан переход на нее

```
git checkout [branch-name]
```

Слияние изменений текузей ветки и веткой `branch`

```
git merge [branch]
```

Удаление ветки `branch-name`

```
git branch -d [branch-name]
```

Работа с файлами

Удаление файла из индекса и рабочей директории

```
git rm [file]
```

Удаление файла из репозитория, но файл остается в директории

```
git rm --cached [file]
```

Смена имени файла

```
git mv [file-original] [file-renamed]
```

История

Просмотр всей истории изменений

```
git log
```

Простотр истории изменения конкретного файла

```
git log --follow [file]
```

Сравнение изменений между ветками `first-branch`, `second-branch`

```
git diff [first-branch]...[second-branch]
```

Просмотр информации о коммите

```
git show [commit]
```

Работа с фрагметами

Помещение изменений во временное хранилище

```
git stash
```

Возврат изменений из временного хранилища

```
git stash pop
```

Просмотр изменений во временном хранилище

```
git stash list
```

Удаление изменений из временного хранилища

```
git stash drop
```

Синхронизация изменений

Получение изменений из закладки `bookmark` (по умолчанию `origin`)

```
git fetch [bookmark]
```

Слияние изменений текузей ветки с веткой из закладки `bookmark`

ветки `branch`

```
git merge [bookmark]/[branch]
```

Выгрузка изменений

```
git push [alias] [branch]
```

Получение и применений изменений

```
git pull
```

Полезные ссылки

1. `Git CheatSheet`
2. `Git CheatSheet`