2. Основы работы с Git

Введение

Git (произн. «гит») — распределённая система управления версиями файлов. Проект был создан Линусом Торвальдсом для управления разработкой ядра Linux. На сегодняшний день поддерживается Джунио Хамано.

Система спроектирована как набор программ, специально разработанных с учётом их использования в скриптах. Это позволяет удобно создавать специализированные системы контроля версий на базе Git или пользовательские интерфейсы. Например, Cogito является именно таким примером фронтенда к репозиториям Git, а StGit использует Git для управления коллекцией патчей.

Git поддерживает быстрое разделение и слияние версий, включает инструменты для визуализации и навигации по нелинейной истории разработки. Как и Darcs, BitKeeper, Mercurial, SVK, Bazaar и Monotone, Git предоставляет каждому разработчику локальную копию всей истории разработки; изменения копируются из одного репозитория в другой.

Удалённый доступ к репозиториям Git обеспечивается git-daemon, gitosis, SSH- или HTTP-сервером. TCP-сервис git-daemon входит в дистрибутив Git и является наряду с SSH наиболее распространённым и надёжным методом доступа. Метод доступа по HTTP, несмотря на ряд ограничений, очень популярен в контролируемых сетях, потому что позволяет использовать существующие конфигурации сетевых фильтров.

Основы работы с удаленным репозиторием

git clone — создание копии (удаленного) репозитория

Для начала работы с центральным репозиторием, следует создать копию оригинального проекта со всей его историей локально.

Клонируем репозиторий, используя протокол http:

```
git clone http://user@somehost:port/~user/repository/project.git
```

Клонируем репозиторий с той же машины в директорию myrepo:

```
git clone /home/username/project myrepo
```

Клонируем репозиторий, используя безопасный протокол ssh:

```
git clone ssh://user@somehost:port/~user/repository
```

У git имеется и собственный протокол:

```
git clone git://user@somehost:port/~user/repository/project.git/
```

Импортируем svn репозиторий, используя протокол http:

```
git svn clone -s http://repo/location
```

-s – понимать стандартные папки SVN (trunk, branches, tags)

git fetch и git pull — забираем изменения из центрального репозитория

Для синхронизации текущей ветки с репозиторием используются команды git fetch и git pull.

git fetch — забрать изменения удаленной ветки из репозитория по умолчания, основной ветки; той, которая была использована при клонировании репозитория. Изменения обновят удаленную ветку (remote tracking branch), после чего надо будет провести слияние с локальной ветку командой git merge.

git fetch /home/username/project — забрать изменения из определенного репозитория.

Возможно также использовать синонимы для адресов, создаваемые командой git remote:

```
git remote add username-project /home/username/project
```

git fetch username-project — забрать изменения по адресу, определяемому синонимом.
Естественно, что после оценки изменений, например, командой git diff, надо создать коммит слияния с основной:

```
git merge username-project/master
```

Команда git pull сразу забирает изменения и проводит слияние с активной веткой.
Забрать из репозитория, для которого были созданы удаленные ветки по умолчанию:

```
git pull
```

Забрать изменения и метки из определенного репозитория:

```
git pull username-project --tags
```

Как правило, используется сразу команда git pull.

git push — вносим изменения в удаленный репозиторий

После проведения работы в экспериментальной ветке, слияния с основной, необходимо обновить удаленный репозиторий (удаленную ветку). Для этого используется команда git push.
Отправить свои изменения в удаленную ветку, созданную при клонировании по умолчанию:

```
git push
```

Отправить изменения из ветки master в ветку experimental удаленного репозитория:

```
git push ssh://yourserver.com/~you/proj.git master:experimental
```

В удаленном репозитории origin удалить ветку experimental:

```
git push origin :experimental
```

В удаленную ветку master репозитория origin (синоним репозитория по умолчанию) ветки локальной ветки master:

```
git push origin master:master
```

Отправить метки в удаленную ветку master репозитория origin:

```
git push origin master --tags
```

Изменить указатель для удаленной ветки master репозитория origin (master будет такой же как и develop)

```
git push origin origin/develop:master
```

Добавить ветку test в удаленный репозиторий origin, указывающую на коммит ветки develop:

```
git push origin origin/develop:refs/heads/test
```

Работа с локальным репозиторием
Базовые команды

git init — создание репозитория

Команда git init создает в директории пустой репозиторий в виде директории .git, где и будет в дальнейшем храниться вся информация об истории коммитов, тегах — о ходе разработки проекта:

```
mkdir project-dir

cd project-dir

git init
```

git add и git rm — индексация изменений

Следующее, что нужно знать — команда git add. Она позволяет внести в индекс — временное хранилище — изменения, которые затем войдут в коммит. Примеры использования:
индексация измененного файла, либо оповещение о создании нового:

```
git add EDITEDFILE
```

внести в индекс все изменения, включая новые файлы:

```
git add .
```

Из индекса и дерева проекта одновременно файл можно удалить командой git rm:
отдельные файлы:

```
git rm FILE1 FILE2
```

хороший пример удаления из документации к git, удаляются сразу все файлы txt из папки:

```
git rm Documentation/\*.txt
```

внести в индекс все удаленные файлы:

```
git rm -r --cached .
```

Сбросить весь индекс или удалить из него изменения определенного файла можно командой git reset:
сбросить весь индекс:

```
git reset
```

удалить из индекса конкретный файл:

```
git reset — EDITEDFILE
```

Команда git reset используется не только для сбрасывания индекса, поэтому дальше ей будет уделено гораздо больше внимания.

git status — состояние проекта, измененные и не добавленные файлы, индексированные файлы

Команду git status, пожалуй, можно считать самой часто используемой наряду с командами коммита и индексации. Она выводит информацию обо всех изменениях, внесенных в дерево директорий проекта по сравнению с последним коммитом рабочей ветки; отдельно выводятся внесенные в индекс и неиндексированные файлы. Использовать ее крайне просто:

```
git status
```

Кроме того, git status указывает на файлы с неразрешенными конфликтами слияния и файлы, игнорируемые git.

git commit — совершение коммита

Коммит — базовое понятие во всех системах контроля версий, поэтому совершаться он должен легко и по возможности быстро. В простейшем случае достаточно после индексации набрать:

```
git commit
```

Если индекс не пустой, то на его основе будет совершен коммит, после чего пользователя попросят прокомментировать вносимые изменения вызовом команды edit. Сохраняемся, и вуаля! Коммит готов.

Есть несколько ключей, упрощающих работу с git commit:

```
git commit -a
```

совершит коммит, автоматически индексируя изменения в файлах проекта. Новые файлы при этом индексироваться не будут! Удаление же файлов будет учтено.

```
git commit -m «commit comment»
```

комментируем коммит прямо из командной строки вместо текстового редактора.

```
git commit FILENAME
```

внесет в индекс и создаст коммит на основе изменений единственного файла.

git reset — возврат к определенному коммиту, откат изменений, «жесткий» или «мягкий»

Помимо работы с индексом (см. выше), git reset позволяет сбросить состояние проекта до какого-либо коммита в истории. В git данное действие может быть двух видов: «мягкого»(soft reset) и «жесткого» (hard reset).

«Мягкий» (с ключом --soft) резет оставит нетронутыми ваши индекс и все дерево файлов и директорий проекта, вернется к работе с указанным коммитом. Иными словами, если вы обнаруживаете ошибку в только что совершенном коммите или комментарии к нему, то легко можно исправить ситуацию:

1. git commit — некорректный коммит
2. git reset --soft HEAD^ — переходим к работе над уже совершенным коммитом, сохраняя все состояние проекта и проиндексированные файлы
3. edit WRONGFILE
4. edit ANOTHERWRONGFILE
5. git add .
6. git commit -c ORIG_HEAD — вернуться к последнему коммиту, будет предложено редактировать его сообщение. Если сообщение оставить прежним, то достаточно изменить регистр ключа -c:

```
git commit -C ORIG_HEAD
```

Обратите внимание на обозначение HEAD^, оно означает «обратиться к предку последнего коммита». Подробней описан синтаксис такой относительной адресации будет ниже, в разделе

«Хэши, тэги, относительная адресация». Соответственно, HEAD — ссылка на последний коммит. Ссылка ORIG_HEAD после «мягкого» резета указывает на оригинальный коммит.

Естественно, можно вернуться и на большую глубину коммитов,

«Жесткий» резет (ключ --hard) — команда, которую следует использовать с осторожностью. git reset --hard вернет дерево проекта и индекс в состояние, соответствующее указанному коммиту, удалив изменения последующих коммитов:

```
git add .

git commit -m «destined to death»

git reset --hard HEAD~1 — больше никто и никогда не увидит этот позорный коммит...

git reset --hard HEAD~3 — ...вернее, три последних коммита. Никто. Никогда!
```

Если команда достигнет точки ветвления, удаления коммита не произойдет.

Для команд слияния или выкачивания последних изменений с удаленного репозитория примеры резета будут приведены в соответствующих разделах.

git revert — отмена изменений, произведенных в прошлом отдельным коммитом

Возможна ситуация, в которой требуется отменить изменения, внесенные отдельным коммитом. git revert создает новый коммит, накладывающий обратные изменения.

Отменяем коммит, помеченный тегом:

```
git revert config-modify-tag
```

Отменяем коммит, используя его хэш:

```
git revert cgsjd2h
```

Для использования команды необходимо, чтобы состояние проекта не отличалось от состояния, зафиксированного последним коммитом.

git log — разнообразная информация о коммитах в целом

Иногда требуется получить информацию об истории коммитов; коммитах, изменивших отдельный файл; коммитах за определенный отрезок времени и так далее. Для этих целей используется команда git log.

Простейший пример использования, в котором приводится короткая справка по всем коммитам, коснувшимся активной в настоящий момент ветки (о ветках и ветвлении подробно узнать можно ниже, в разделе «Ветвления и слияния»):

```
git log
```

Получить подробную информацию о каждом в виде патчей по файлам из коммитов можно, добавив ключ -p (или -u):

```
git log -p
```

Статистика изменения файлов, вроде числа измененных файлов, внесенных в них строк, удаленных файлов вызывается ключом --stat:

```
git log --stat
```

За информацию по созданиям, переименованиям и правам доступа файлов отвечает ключ --summary:

```
git log --summary
```

Чтобы просмотреть историю отдельного файла, достаточно указать в виде параметра его имя (кстати, в моей старой версии git этот способ не срабатывает, обязательно добавлять " — " перед «README»):

```
git log README
```

или, если версия git не совсем свежая:

```
git log — README
```

Далее будет приводится только более современный вариант синтаксиса. Возможно указывать время, начиная в определенного момента («weeks», «days», «hours», «s» и так далее):

```
git log --since=«1 day 2 hours» README

git log --since=«2 hours» README
```

изменения, касающиеся отдельной папки:

```
git log --since=«2 hours» dir/
```

Можно отталкиваться от тегов.
Все коммиты, начиная с тега v1:

```
git log v1...
```

Все коммиты, включающие изменения файла README, начиная с тега v1:

```
git log v1... README
```

Все коммиты, включающие изменения файла README, начиная с тега v1 и заканчивая тегом v2:

```
git log v1..v2 README
```

Интересные возможности по формату вывода команды предоставляет ключ --pretty.
Вывести на каждый из коммитов по строчке, состоящей из хэша (здесь — уникального идентификатора каждого коммита, подробней — дальше):

```
git log --pretty=oneline
```

Лаконичная информация о коммитах, приводятся только автор и комментарий:

```
git log --pretty=short
```

Более полная информация о коммитах, с именем автора, комментарием, датой создания и внесения коммита:

```
git log --pretty=full/fuller
```

В принципе, формат вывода можно определить самостоятельно:

```
git log --pretty=format:'FORMAT'
```

Определение формата можно поискать в разделе по git log из *Git Community Book* или справке. Красивый ASCII-граф коммитов выводится с использованием ключа --graph.

git diff — отличия между деревьями проекта, коммитами и т.д.

Своего рода подмножеством команды git log можно считать команду git diff, определяющую изменения между объектами в проекте - деревьями (файлов и директорий).
Показать изменения, не внесенные в индекс:

```
git diff
```

Изменения, внесенные в индекс:

```
git diff --cached
```

Изменения в проекте по сравнению с последним коммитом:

```
git diff HEAD
```

Предпоследним коммитом:

```
git diff HEAD^
```

Можно сравнивать «головы» веток:

```
git diff master..experimental
```

или активную ветку с какой-либо:

```
git diff experimental
```

git show — показать изменения, внесенные отдельным коммитом

Посмотреть изменения, внесенные любым коммитом в истории, можно командой git show:

```
git show COMMIT_TAG
```

git blame и git annotate — команды, помогающие отслеживать изменения файлов

При работе в команде часто требуется выяснить, кто именно написал конкретный код. Удобно использовать команду git blame, выводящую построчную информацию о последнем коммите, коснувшемся строки, имя автора и хэш коммита:

```
git blame README
```

Можно указать и конкретные строки для отображения:

```
git blame -L 2,+3 README — выведет информацию по трем строкам, начиная со второй.
```

Аналогично работает команда git annotate, выводящая и строки, и информацию о коммитах, их коснувшихся:

```
git annotate README
```

git grep — поиск слов по проекту, состоянию проекта в прошлом

git grep, в целом, просто дублирует функционал знаменитой юниксовой команды. Однако он позволяет слова и их сочетания искать в прошлом проекта, что бывает очень полезно.

Поиск слова *tst* в проекте:

```
git grep tst
```

Подсчитать число упоминаний *tst* в проекте:

```
git grep -c tst
```

Поиск в старой версии проекта:

```
git grep tst v1
```

Команда позволяет использовать логическое И и ИЛИ.

Найти строки, где упоминаются и первое слово, и второе:

```
git grep -e 'first' --and -e 'another'
```

Найти строки, где встречается хотя бы одно из слов:

```
git grep --all-match -e 'first' -e 'second'
```

Ветвление

git branch — создание, перечисление и удаление веток

Работа с ветками — очень легкая процедура в git, все необходимые механизмы сконцентрированы в одной команде:

Просто перечислить существующие ветки, отметив активную:

```
git branch
```

Создать новую ветку new-branch:

```
git branch new-branch
```

Удалить ветку, если та была залита (merged) с разрешением возможных конфликтов в текущую:

```
git branch -d new-branch
```

Удалить ветку в любом случае:

```
git branch -D new-branch
```

Переименовать ветку:

```
git branch -m new-name-branch
```

Показать те ветки, среди предков которых есть определенный коммит:

```
git branch --contains v1.2
```

git checkout — переключение между ветками, извлечение файлов

Команда git checkout позволяет переключаться между последними коммитами (если упрощенно) веток:

```
checkout some-other-branch
```

Создаст ветку, в которую и произойдет переключение

```
checkout -b some-other-new-branch
```

Если в текущей ветке были какие-то изменения по сравнению с последним коммитом в ветке(HEAD), то команда откажется производить переключение, дабы не потерять произведенную работу. Проигнорировать этот факт позволяет ключ -f:

```
checkout -f some-other-branch
```

В случае, когда изменения надо все же сохранить, следует использовать ключ -m. Тогда команда перед переключением попробует залить изменения в текущую ветку и, после разрешения возможных конфликтов, переключиться в новую:

```
checkout -m some-other-branch
```

Вернуть файл (или просто вытащить из прошлого коммита) позволяет команда вида:
Вернуть *somefile* к состоянию последнего коммита:

```
git checkout somefile
```

Вернуть *somefile* к состоянию на два коммита назад по ветке:

```
git checkout HEAD~2 somefile
```

git merge — слияние веток (разрешение возможных конфликтов)

Слияние веток, в отличие от обычной практики централизованных систем, в git происходит практически каждый день. Естественно, что имеется удобный интерфейс к популярной операции.

Попробовать объединить текущую ветку и ветку new-feature:

```
git merge new-feature
```

В случае возникновения конфликтов коммита не происходит, а по проблемным файлам расставляются специальные метки а-ля svn; сами же файлы отмечаются в индексе как «не соединенные» (unmerged). До тех пор пока проблемы не будут решены, коммит совершить будет нельзя.

Например, конфликт возник в файле TROUBLE, что можно увидеть в git status.

Произошла неудачная попытка слияния:

```
git merge experiment
```

Смотрим на проблемные места:

```
git status
```

Разрешаем проблемы:

```
edit TROUBLE
```

Индексируем наши изменения, тем самым снимая метки:

```
git add .
```

Совершаем коммит слияния:

```
git commit
```

Вот и все, ничего сложного. Если в процессе разрешения вы передумали разрешать конфликт, достаточно набрать (это вернёт обе ветки в исходные состояния):

```
git reset --hard HEAD
```

Если же коммит слияния был совершен, используем команду:

```
git reset --hard ORIG_HEAD
```

git rebase — построение ровной линии коммитов

Предположим, разработчик завел дополнительную ветку для разработки отдельной возможности и совершил в ней несколько коммитов. Одновременно по какой-либо причине в основной ветке также были совершены коммиты: например, в нее были залиты изменения с удаленного сервера, либо сам разработчик совершал в ней коммиты.

В принципе, можно обойтись обычным git merge. Но тогда усложняется сама линия разработки, что бывает нежелательно в слишком больших проектах, где участвует множество разработчиков.

Предположим, имеется две ветки, master и топик, в каждой из которых было совершенно несколько коммитов начиная с момента ветвления. Командаgit rebase берет коммиты из ветки topic и накладывает их на последний коммит ветки master.

Вариант, в котором явно указывается, что и куда накладывается:

```
git-rebase master topic
```

на master накладывается активная в настоящий момент ветка:

```
git-rebase master
```

После использования команды история становится линейной. При возникновении конфликтов при поочередном накладывании коммитов работа команды будет останавливаться, а в проблемные местах файлов появятся соответствующие метки. После редактирования — разрешения конфликтов — файлы следует внести в индекс командой git add и продолжить наложение следующих коммитов командой git rebase --continue. Альтернативными выходами будут команды git rebase --skip (пропустить наложение коммита и перейти к следующему) или git rebase --abort (отмена работы команды и всех внесенных изменений).

С ключом -i (--interactive) команда будет работать в интерактивном режиме. Пользователю будет предоставлена возможность определить порядок внесения изменений, автоматически будет вызывать редактор для разрешения конфликтов и так далее.

git cherry-pick — применение к дереву проекта изменений, внесенных отдельным коммитом

Если ведется сложная история разработки, с несколькими длинными ветками разработками, может возникнуть необходимость в применении изменений, внесенных отдельным коммитом одной ветки, к дереву другой (активной в настоящий момент).

Изменения, внесенные указанным коммитом будут применены к дереву, автоматически проиндексированы и станут коммитом в активной ветке:

```
git cherry-pick BUG_FIX_TAG
```

Ключ -n показывает, что изменения надо просто применить к дереву проекта без индексации и создания коммита

```
git cherry-pick BUG_FIX_TAG -n
```

Прочие команды и необходимые возможности

Хэш — уникальная идентификация объектов

В git для идентификации любых объектов используется уникальный (то есть с огромной вероятностью уникальный) хэш из 40 символов, который определяется хэшируюшей функцией на основе содержимого объекта. Объекты — это все: коммиты, файлы, тэги, деревья. Поскольку хэш уникален для содержимого, например, файла, то и сравнивать такие файлы очень легко — достаточно просто сравнить две строки в сорок символов.

Больше всего нас интересует тот факт, что хэши идентифицируют коммиты. В этом смысле хэш — продвинутый аналог ревизий Subversion. Несколько примеров использования хэшей в качестве способа адресации:

найти разницу текущего состояния проекта и коммита за номером… сами видите, каким:

```
git diff f292ef5d2b2f6312bc45ae49c2dc14588eef8da2
```

То же самое, но оставляем только шесть первых символов. Git поймет, о каком коммите идет речь, если не существует другого коммита с таким началом хэша:

```
git diff f292ef5
```

Иногда хватает и четырех символов:

```
git diff f292
```

Читаем лог с коммита по коммит:

```
git log febc32...f292
```

Разумеется, человеку пользоваться хэшами не так удобно, как машине, именно поэтому были введены другие объекты — тэги.

git tag — тэги как способ пометить уникальный коммит

Тэг (tag) — это объект, связанный с коммитом; хранящий ссылку на сам коммит, имя автора, собственное имя и некоторый комментарий. Кроме того, разработчик может оставлять на таких тегах собственную цифровую подпись.

Кроме этого в git представленные так называемые «легковесные тэги» (*lightweight tags*), состоящие только из имени и ссылки на коммит. Такие тэги, как правило, используются для упрощения навигации по дереву истории; создать их очень легко.

Создать «легковесный» тэг, связанный с последним коммитом; если тэг уже есть, то еще один создан не будет:

```
git tag stable-1
```

Пометить определенный коммит:

```
git tag stable-2 f292ef5
```

Удалить тег:

```
git tag -d stable-2
```

Перечислить тэги:

```
git tag -l
```

Создать тэг для последнего коммита, заменить существующий, если таковой уже был:

```
git tag -f stable-1.1
```

После создания тэга его имя можно использовать вместо хэша в любых командах вроде git diff, git log и так далее:

```
git diff stable-1.1...stable-1
```

Обычные тэги имеет смысл использовать для приложения к коммиту какой-либо информации, вроде номера версии и комментария к нему. Иными словами, если в комментарии к коммиту пишешь «исправил такой-то баг», то в комментарии к тэгу по имени «v1.0» будет что-то вроде «стабильная версия, готовая к использованию».

Создать обычный тэг для последнего коммита; будет вызван текстовый редактор для составления комментария:

```
git tag -a stable
```

Создать обычный тэг, сразу указав в качестве аргумента комментарий:

```
git tag -a stable -m "production version"
```

Команды перечисления, удаления, перезаписи для обычных тэгов не отличаются от команд для «легковесных» тэгов.

Относительная адресация

Вместо ревизий и тэгов в качестве имени коммита можно опираться на еще один механизм — относительную адресацию. Например, можно обратиться прямо к предку последнего коммита ветки master:

```
git diff master^
```

Если после «птички» поставить цифру, то можно адресоваться по нескольким предкам коммитов слияния:
найти изменения по сравнению со вторым предком последнего коммита в master; HEAD здесь — указатель на последний коммит активной ветки:

```
git diff HEAD^2
```

Аналогично, тильдой можно просто указывать, насколько глубоко в историю ветки нужно погрузиться:
что привнес «дедушка» нынешнего коммита:

```
git diff master^^
```

То же самое:

```
git diff master~2
```

Обозначения можно объединять, чтобы добраться до нужного коммита:

```
git diff master~3^~2

git diff master~6
```

файл .gitignore — объясняем git, какие файлы следует игнорировать

Иногда по директориям проекта встречаются файлы, которые не хочется постоянно видеть в сводке git status. Например, вспомогательные файлы текстовых редакторов, временные файлы и прочий мусор.
Заставить git status игнорировать определенные файлы можно, создав в корне или глубже по дереву (если ограничения должны быть только в определенных директория) файл .gitignore. В этих файлах можно описывать шаблоны игнорируемых файлов определенного формата.
Пример содержимого такого файла:

```
#комментарий к файлу .gitignore

#игнорируем сам .gitignore

.gitignore

#все html-файлы...
```

```
*.html

#...кроме определенного

!special.html

#не нужны объектники и архивы

*.[ao]
```

Существуют и другие способы указания игнорируемых файлов, о которых можно узнать из справки git help gitignore.

Серверные команды репозитория

; git update-server-info : Команда создает вспомогательные файлы для dumb-сервера в $GIT_DIR/info и $GIT_OBJECT_DIRECTORY/info каталогах, чтобы помочь клиентам узнать, какие ссылки и пакеты есть на сервере.

; git count-objects : Проверка, сколько объектов будет потеряно и объём освобождаемого места при перепаковке репозитория.

; git gc : Переупаковка локального репозитория.

Рецепты

Создание пустого репозитория на сервере

```
repo="repo.git"

mkdir $repo

cd $repo

git init --bare

chown git. -R ./

cd ../
```

Импорт svn репозитория на Git-сервер

```
repo="repo.svn"

svnserver="http://svn.calculate.ru"

git svn clone -s $svnserver/$repo $repo

mv $repo/.git/refs/remotes/tags $repo/.git/refs/tags

rm -rf $repo/.git/refs/remotes

rm -rf $repo/.git/svn

mv $repo/.git $repo.git

rm -rf $repo

cd $repo.git

chown git. -R ./

cd ../
```

Ссылки

- правка коммитов http://www.calculate-linux.org/main/ru/recommit
- Перенос SVN-репозитория в git http://leonid.shevtsov.me/22-07-2009/perenos-svn-repozitariya-v-git/
- Учебник-введение в git http://freesource.info/wiki/RuslanHihin/GitTutorial1?v=3pr
- Руководство пользователя GIT http://freesource.info/wiki/RuslanHihin/GitUserManual?v=3fj
- 20 повседневных команд git http://freesource.info/wiki/RuslanHihin/20povsedevnyxkomandgit
- Внутренности git http://los-t.livejournal.com/tag/git+guts
- Введение в структуру хранилища git http://blog.tarantsov.com/2008/11/essential-git.html
- Практическое введение в git http://admdev.blogspot.com/2009/02/git.html
- Работа с git для начинающих http://blog.nsws.ru/rabota-s-git-dlya-nachinayushhix.html
- Переходим с SVN на Git http://git.or.cz/course/svn.html
- XX полезных советов для пользователей Git среднего уровня. Часть 1 http://habrahabr.ru/blogs/Git/75728/
- XX полезных советов для пользователей Git среднего уровня. Часть 2 http://habrahabr.ru/blogs/Git/76084/
- Внешние зависимости в гите: submodule или subtree? http://habrahabr.ru/blogs/Git/75964/
- Командная работа в Git http://habrahabr.ru/blogs/Git/75990/