МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

ЛЕКЦИОННЫЕ МАТЕРИАЛЫ

Технология разработки программных приложений

(наименование дисциплины (модуля) в соответствии с учебным планом)

Уровень **бакалавриат**
(бакалавриат, магистратура, специалитет)

Форма обучения **очная**
(очная, очно-заочная, заочная)

Направление(-я) подготовки **09.03.03 «Прикладная информатика»**
(код(-ы) и наименование(-я))

Институт **информационных технологий (ИТ)**
(полное и краткое наименование)

Кафедра **Математического обеспечения и стандартизации информационных технологий (МОСИТ)**
(полное и краткое наименование кафедры, реализующей дисциплину(модуль))

Лектор **преподаватель, Миронов Антон Николаевич**
(сокращенно – ученая степень, ученое звание; полностью – ФИО)

Используются в данной редакции с учебного года 2022/23
(учебный год цифрами)

Проверено и согласовано «___»_____2023 г. _____ Зуев А.С.
(подпись директора Института/Филиала с расшифровкой)

Москва 2023 г.

1. ЗАДАЧИ СИСТЕМ КОНТРОЛЯ И УПРАВЛЕНИЯМИ ВЕРСИЯМИ

1.1. Задачи системы контроля и управлениями версиями

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. В частности, системы управления версиями применяются в САПР, обычно в составе систем управления данными об изделии (PDM). Управление версиями используется в инструментах конфигурационного управления (Software Configuration Management Tools).

Общие сведения

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.

Задачами системы контроля версий являются следующие:

- фиксация изменений для отслеживаемых, добавляемых или удаляемых файлов исходного кода в рабочем дереве проекта;
- хранение истории зафиксированных изменений исходного кода;

- сравнение и поиск ранее зафиксированных изменений исходного кода;
- объединение наработок и экспериментальных изменений, ведущихся различными разработчиками над одним исходным кодом.

Цепочка зафиксированных изменений исходного кода называется историей программного проекта. История программного проекта хранится в репозитории программного проекта.

1.2. Локальные системы контроля версий

Один из примеров локальной СУВ предельно прост: многие предпочитают контролировать версии, просто копируя файлы в другой каталог (как правило, добавляя текущую дату к названию каталога). Такой подход очень распространён, потому что прост, но он и чаще даёт сбои. Очень легко забыть, что ты не в том каталоге, и случайно изменить не тот файл, либо скопировать файлы не туда, куда хотел, и затереть нужные файлы. Чтобы решить эту проблему, программисты уже давно разработали локальные СКВ с простой базой данных, в которой хранятся все изменения нужных файлов

Одной из наиболее популярных СКВ такого типа является RCS (Revision Control System, Система контроля ревизий), которая до сих пор устанавливается на многие компьютеры. Даже в современной операционной системе Mac OS X утилита rcs устанавливается вместе с Developer Tools. RCS была разработана в начале 1980-х годов Вальтером Тичи (Walter F. Tichy). Система позволяет хранить версии только одного файла, таким образом управлять несколькими файлами приходится вручную. Для каждого файла находящегося под контролем системы информация о версиях хранится в специальном файле с именем оригинального файла к которому в конце добавлены символы ',v'. Например для файла file.txt версии будут храниться в файле file.txt,v. Эта утилита основана на работе с наборами патчей между парами версий (патч — файл, описывающий различие между файлами). Это позволяет пересоздать любой файл на любой момент времени, последовательно накладывая патчи. Для хранения версий система использует утилиту diff. Хотя RCS соответствует минимальным требованиям к системе контроля версий она имеет следующие основные недостатки, которые также послужили стимулом для создания следующей рассматриваемой системы:

- Работа только с одним файлом, каждый файл должен контролироваться отдельно;
- Неудобный механизм одновременной работы нескольких пользователей с системой, хранилище просто блокируется пока заблокировавший его пользователь не разблокирует его;
- От бекапов вас никто не освобождает, вы рискуете потерять всё.

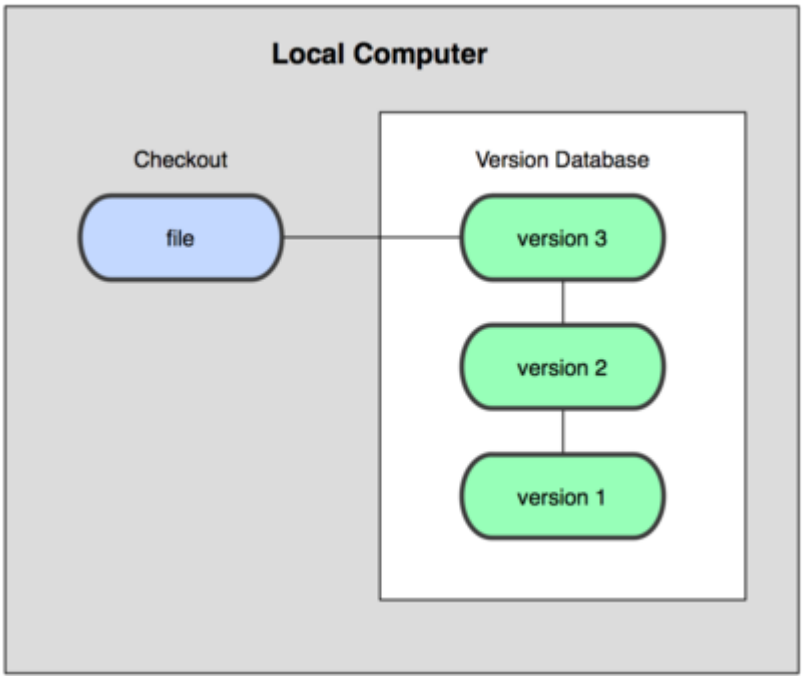

Рисунок 1.1. Схема локальной СКВ.

1.3. Централизованные системы контроля версий

Следующей основной проблемой оказалась необходимость сотрудничать с разработчиками за другими компьютерами. Чтобы решить её, были созданы централизованные системы контроля версий (ЦСКВ). В таких системах, например, CVS, Subversion и Perforce, есть центральный сервер, на котором хранятся все файлы под версионным контролем, и ряд клиентов, которые получают копии файлов из него. Много лет это было стандартом для систем контроля версий.

Такой подход имеет множество преимуществ, особенно над локальными СКВ. К примеру, все знают, кто и чем занимается в проекте. У администраторов есть чёткий контроль над тем, кто и что может делать, и, конечно, администрировать ЦСКВ намного легче, чем локальные базы на каждом клиенте. Однако при таком подходе есть и несколько серьёзных недостатков. Наиболее очевидный — централизованный сервер является уязвимым местом всей системы. Если сервер выключается на час, то в течение часа разработчики не могут взаимодействовать, и никто не может сохранить новой версии своей работы. Если же повреждается диск с центральной базой данных и нет резервной копии, вы теряете абсолютно всё — всю историю проекта, разве что за исключением нескольких рабочих версий, сохранившихся на рабочих машинах пользователей.

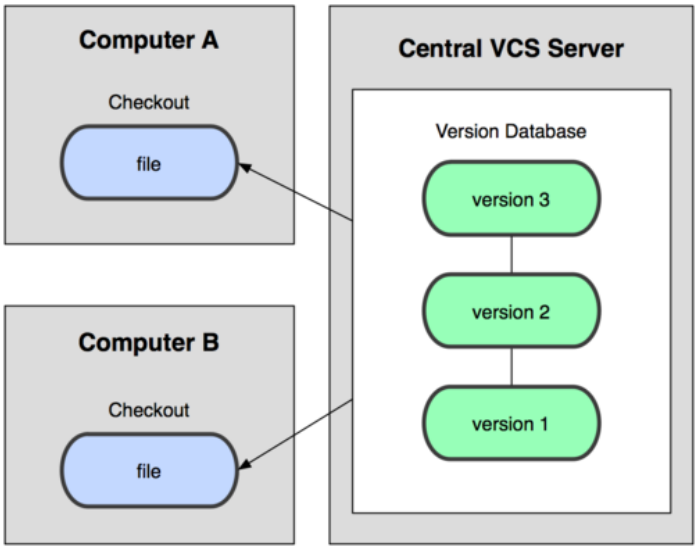

Рисунок 1.2. Схема централизованного контроля версий.

1.4. Система совместных версий CVS

CVS (Concurrent Versions System, Система совместных версий) пока остается самой широко используемой системой, но быстро теряет свою популярность из-за недостатков, которые будут рассмотрены ниже. Дик Грун (Dick Grune) разработал CVS в середине 1980-х. Для хранения индивидуальных файлов CVS (также как и RCS) использует файлы в RCS формате, но позволяет управлять группами файлов расположенных в директориях. Также CVS использует клиент-сервер архитектуру, в которой вся информация о версиях хранится на сервере. Использование клиент-сервер архитектуры позволяет использовать CVS даже географически распределенным командами пользователей где каждый пользователь имеет свой рабочий директорий с копией проекта. Как следует из названия пользователи могут использовать систему совместно.

Возможные конфликты при изменении одного и того же файла разрешаются тем, что система позволяет вносить изменения только в самую последнюю версию файла. Таким образом всегда рекомендуется перед заливкой своих изменений обновлять свою рабочую копию файлов на случай возможных конфликтующих изменений. При обновлении система вносит изменения в рабочую копию автоматически и только в случае конфликтующих изменений в одном из мест файла требуется ручное исправление места конфликта.

CVS также позволяет вести несколько линий разработки проекта с помощью ветвей (branches) разработки. Таким образом, как уже упоминалось выше, можно исправлять ошибки в первой версии проекта и параллельно разрабатывать новую функциональность.

CVS использовалась большим количеством проектов, но конечно не была лишена недостатков, которые позднее привели к появлению следующей рассматриваемой системы. Рассмотрим основные недостатки:

- Так как версии хранятся в файлах RCS нет возможности сохранять версии директорий. Стандартный способ обойти это препятствие - это сохранить какой-либо файл (например, README.txt) в директории;

- Перемещение, или переименование файлов не подвержено контролю версий. Стандартный способ сделать это: сначала скопировать файл, удалить старый с помощью команды cvs remove и затем добавить с его новым именем с помощью команды cvs add;

Система управления параллельными версиями (Concurrent Versions System) – логическое развитие системы управления пересмотрами версий (RCS), использующая ее стандарты и алгоритмы по управлению версиями, но значительно более функциональная, и позволяющая работать не только с отдельными файлами, но и с целыми проектами.

CVS основана на технологии клиент-сервер, взаимодействующих по сети. Клиент и сервер также могут располагаться на одной машине, если над проектом работает только один человек, или требуется вести локальный контроль версий.

Работа CVS организована следующим образом. Последняя версия и все сделанные изменения хранятся в репозитории сервера. Клиенты, подключаясь к серверу, проверяют отличия локальной версии от последней версии, сохраненной в репозитории, и, если есть отличия, загружают их в свой локальный проект. При необходимости решают конфликты и вносят требуемые изменения в разрабатываемый продукт. После этого все изменения загружаются в репозиторий сервера. CVS, при необходимости, позволяет откатываться на нужную версию разрабатываемого проекта и вести управление несколькими проектами одновременно.

Приведем основные достоинства и недостатки системы управления параллельными версиями.

Достоинства:

- Несколько клиентов могут одновременно работать над одним и тем же проектом.
- Позволяет управлять не одним файлом, а целыми проектами.
- Обладает огромным количеством удобных графических интерфейсов, способных удовлетворить практически любой, даже самый требовательный вкус.
- Широко распространена и поставляется по умолчанию с большинством операционных систем Linux.
- При загрузке тестовых файлов из репозитория передаются только изменения, а не весь файл целиком.

Недостатки:

- При перемещении, переименовании файла или директории теряются все, привязанные к этому файлу или директории, изменения.
- Сложности при ведении нескольких параллельных веток одного и того же проекта.
- Ограниченная поддержка шрифтов.

- Для каждого изменения бинарного файла сохраняется вся версия файла, а не только внесенное изменение.
- С клиента на сервер измененный файл всегда передается полностью.
- Ресурсоемкие операции, так как требуют частого обращения к репозиторию, и сохраняемые копии имеют некоторую избыточность.

Несмотря на то, что CVS устарела и обладает серьезными недостатками, она все еще является одной из самых популярных систем контроля версий и отлично подходит для управления небольшими проектами, не требующих создания нескольких параллельных версий, которые надо периодически объединять. CVS можно порекомендовать, как промежуточный шаг в освоении работы систем контроля версий, ведущий к более мощным и современным видам таких программ.

1.5. Subversion (SVN)

Subversion – эта централизованная система управления версиями, созданная в 2000 году и основанная на технологии клиент-сервер. Она обладает всеми достоинствами CVS и решает основные ее проблемы (переименование и перемещение файлов и каталогов, работа с двоичными файлами и т.д.). Часто ее называют по имени клиентской части – SVN. SVN также, как и CVS использует клиент-сервер архитектуру.

Принцип работы с Subversion очень походит на работу с CVS. Клиенты копируют изменения из репозитория и объединяют их с локальным проектом пользователя. Если возникают конфликты локальных изменений и изменений, сохраненных в репозитории, то такие ситуации разрешаются вручную. Затем в локальный проект вносятся изменения, и полученный результат сохраняется в репозитории.

При работе с файлами, не позволяющими объединять изменения, может использоваться следующий принцип:

1. Файл скачивается из репозитория и блокируется (запрещается его скачивание из репозитория).
2. Вносятся необходимые изменения.
3. Загружается файл в репозиторий и снимается блокировка (разрешается его скачивание из репозитория другим клиентам).

Во многом, из-за простаты и схожести в управлении с CVS, но в основном, из-за своей широкой функциональности, Subversion с успехом конкурирует с CVS и даже успешно ее вытесняет.

Однако, и у Subversion есть недостатки. Давайте рассмотрим ее слабые и сильные стороны для сравнения с другими системами управления версиями.

Достоинства:

- Система команд, схожая с CVS.
- Поддерживается большинство возможностей CVS.
- Разнообразные графические интерфейсы и удобная работа из консоли.

- Отслеживается история изменения файлов и каталогов даже после их переименования и перемещения.
- Высокая эффективность работы, как с текстовыми, так и с бинарными файлами.
- Встроенная поддержка во многие интегрированные средства разработки, такие как KDevelop, Zend Studio и многие другие.
- Возможность создания зеркальных копий репозитория.
- Два типа репозитория – база данных или набор обычных файлов.
- Возможность доступа к репозиторию через Apache с использованием протокола WebDAV.
- Наличие удобного механизма создания меток и ветвей проектов.
- Можно с каждым файлом и директорией связать определенный набор свойств, облегчающий взаимодействие с системой контроля версии.
- Широкое распространение позволяет быстро решить большинство возникающих проблем, обратившись к данным, накопленным Интернет-сообществом.

Недостатки:

- Полная копия репозитория хранится на локальном компьютере в скрытых файлах, что требует достаточно большого объема памяти.
- Существуют проблемы с переименованием файлов, если переименованный локально файл одним клиентом был в это же время изменен другим клиентом и загружен в репозиторий.
- Слабо поддерживаются операции слияния веток проекта.
- Сложности с полным удалением информации о файлах, попавших в репозиторий, так как в нем всегда остается информация о предыдущих изменениях файла, и непредусмотрено никаких штатных средств для полного удаления данных о файле из репозитория.

Subversion – современная система контроля версий, обладающая широким набором инструментов, позволяющих удовлетворить любые нужды для управления версиями проекта с помощью централизованной системы контроля. В Интернете множество ресурсов посвящено особенностям Subversion, что позволяет быстро и качественно решать все возникающие в ходе работы проблемы.

Простота установки, подготовки к работе и широкие возможности позволяют ставить subversion на одну из лидирующих позиций в конкурентной гонке систем контроля версий.

1.6. Распределённые системы контроля версий

И в этой ситуации в игру вступают распределённые системы контроля версий (РСКВ). В таких системах как Git, Mercurial, Bazaar или Darcs клиенты не просто выгружают последние версии файлов, а полностью копируют весь репозиторий. Поэтому в случае, когда "умирает" сервер, через который шла работа, любой клиентский репозиторий может быть скопирован обратно на сервер, чтобы восстановить базу данных. Каждый раз, когда клиент забирает свежую версию файлов, он создаёт себе полную копию всех данных.

Кроме того, в большей части этих систем можно работать с несколькими удалёнными репозиториями, таким образом, можно одновременно работать по-разному с разными группами людей в рамках одного проекта. Так, в одном проекте можно одновременно вести несколько типов рабочих процессов, что невозможно в централизованных системах.

Назначение распределенные систем

Как следует из названия, одна из основных идей распределенных систем — это отсутствие четко выделенного центрального хранилища версий - репозитория. В случае распределенных систем набор версий может быть полностью, или частично распределен между различными хранилищами, в том числе и удаленными. Такая модель отлично вписывается в работу распределенных команд, например, распределенной по всему миру команды разработчиков, работающих над одним проектом с открытым исходным кодом. Разработчик такой команды может скачать себе всю информацию по версиям и после этого работать только на локальной машине. Как только будет достигнут результат одного из этапов работы, изменения могут быть залиты в один из центральных репозиториев или, опубликованы для просмотра на сайте разработчика, или в почтовой рассылке. Другие участники проекта, в свою очередь, смогут обновить свою копию хранилища версий новыми изменениями, или попробовать опубликованные изменения на отдельной, тестовой ветке разработки. К сожалению, без хорошей организации проекта отсутствие одного центрального хранилища может быть минусом распределенных систем. Если в случае централизованных систем всегда есть один общий репозиторий откуда можно получить последнюю версию проекта, то в случае распределенных систем нужно организационно решить какая из веток проекта будет основной. Почему распределенная система контроля версий может быть интересна кому-то, кто уже использует централизованную систему - такую как Subversion? Любая работа подразумевает принятие решений, и в большинстве случаев необходимо пробовать различные варианты: при работе с системами контроля версий для рассмотрения различных вариантов и работы над большими изменениями служат ветки разработки. И хотя это достаточно естественная концепция, пользоваться ей в Subversion достаточно непросто. Тем более, всё усложняется в случае множественных последовательных объединений с одной ветки на другую — в этом случае нужно безошибочно указывать начальные и конечные версии каждого изменения, чтобы избежать конфликтов и ошибок. Для распределенных систем контроля версий ветки разработки являются одной из основополагающих концепций — в большинстве случаев каждая копия хранилища версий является веткой разработки. Таким образом, механизм объединения изменений с одной ветки на другую в случае

распределенных систем является одним из основных, что позволяет пользователям прикладывать меньше усилий при пользовании системой.

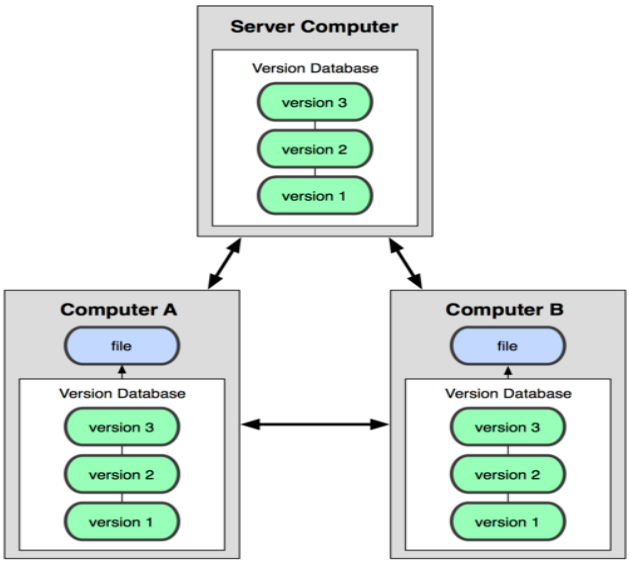

Рисунок 1.3. Схема распределённой системы контроля версий.

Краткое описание популярных распределенных СУВ

- Git - распределенная система контроля версий, разработанная Линусом Торвальдсом. Изначально Git предназначалась для использования в процессе разработки ядра Linux, но позже стала использоваться и во многих других проектах — таких, как, например, X.org и Ruby on Rails, Drupal. На данный момент Git является самой быстрой распределенной системой, использующей самое компактное хранилище ревизий. Но в тоже время для пользователей, переходящих, например, с Subversion интерфейс Git может показаться сложным;
- Mercurial - распределенная система, написанная на языке Python с несколькими расширениями на C. Из использующих Mercurial проектов можно назвать, такие, как, Mozilla и MoinMoin.
- Bazaar - система разработка которой поддерживается компанией Canonical — известной своими дистрибутивом Ubuntu и сайтом httpss://launchpad.net/. Система в основном написана на языке Python и используется такими проектами, как, например, MySQL.
- Codeville - написанная на Python распределенная система использующая инновационный алгоритм объединения изменений (merge). Система используется, например, при разработке оригинального клиента BitTorrent.
- Darcs - распределенная система контроля версий, написанная на Haskell используемая, например, проектом Buildbot.
- Monotone - система, написанная на C++ и использующая SQLite как хранилище ревизий.

1.7. Система управления версиями Git

С февраля 2002 года для разработки ядра Linux'а большинством программистов стала использоваться система контроля версий BitKeeper. Довольно долгое время с ней не возникало проблем, но в 2005 году Лари МакВоем (разработчик BitKeeper'а) отозвал бесплатную версию программы.

Разрабатывать проект масштаба Linux без мощной и надежной системы контроля версий – невозможно. Одним из кандидатов и наиболее подходящим проектом оказалась система контроля версий Monotine, но Торвальдса Линуса не устроила ее скорость работы. Так как особенности организации Monatone не позволяли значительно увеличить скорость обработки данных, то 3 апреля 2005 года Линус приступил к разработке собственной системы контроля версий – Git.

Практически одновременно с Линусом (на три дня позже), к разработке новой системы контроля версий приступил и Мэтт Макал. Свой проект Мэтт назвал Mercurial, но об этом позже, а сейчас вернемся к распределенной системе контроля версий Git.

Git – это гибкая, распределенная (без единого сервера) система контроля версий, дающая массу возможностей не только разработчикам программных продуктов, но и писателям для изменения, дополнения и отслеживания изменения «рукописей» и сюжетных линий, и учителям для корректировки и развития курса лекций, и администраторам для ведения документации, и для многих других направлений, требующих управления историей изменений.

У каждого разработчика, использующего Git, есть свой локальный репозиторий, позволяющий локально управлять версиями. Затем, сохраненными в локальный репозиторий данными, можно обмениваться с другими пользователями.

Часто при работе с Git создают центральный репозиторий, с которым остальные разработчики синхронизируются. Пример организации системы с центральным репозиторием – это проект разработки ядра Linux'а.

В этом случае все участники проекта ведут свои локальны разработки и беспрепятственно скачивают обновления из центрального репозитория. Когда необходимые работы отдельными участниками проекта выполнены и отлажены, они, после удостоверения владельцем центрального репозитория в корректности и актуальности проделанной работы, загружают свои изменения в центральный репозиторий.

Наличие локальных репозиторием также значительно повышает надежность хранения данных, так как, если один из репозиториев выйдет из строя, данные могут быть легко восстановлены из других репозиториев.

Работа над версиями проекта в Git может вестись в нескольких ветках, которые затем могут с легкостью полностью или частично объединяться, уничтожаться, откатываться и разрастаться во все новые и новые ветки проекта.

Можно долго обсуждать возможности Git'а, но для краткости и более простого восприятия приведем основные достоинства и недостатки этой системы управления версиями

Достоинства:

- Надежная система сравнения ревизий и проверки корректности данных, основанные на алгоритме хеширования SHA1 (Secure Hash Algorithm 1).
- Гибкая система ветвления проектов и слияния веток между собой.
- Наличие локального репозитория, содержащего полную информацию обо всех изменениях, позволяет вести полноценный локальный контроль версий и заливать в главный репозиторий только полностью прошедшие проверку изменения.
- Высокая производительность и скорость работы.
- Удобный и интуитивно понятный набор команд.
- Множество графических оболочек, позволяющих быстро и качественно вести работы с Git'ом.
- Возможность делать контрольные точки, в которых данные сохраняются без дельта компрессии, а полностью. Это позволяет уменьшить скорость восстановления данных, так как за основу берется ближайшая контрольная точка, и восстановление идет от нее. Если бы контрольные точки отсутствовали, то восстановление больших проектов могло бы занимать часы.
- Широкая распространенность, легкая доступность и качественная документация.
- Гибкость системы позволяет удобно ее настраивать и даже создавать специализированные контроля системы или пользовательские интерфейсы на базе git.
- Универсальный сетевой доступ с использованием протоколов http, ftp, rsync, ssh и др.

Недостатки:

- Возможные (но чрезвычайно низкие) совпадения хеш - кода отличных по содержанию ревизий.
- Не отслеживается изменение отдельных файлов, а только всего проекта целиком, что может быть неудобно при работе с большими проектами, содержащими множество несвязных файлов.
- При начальном (первом) создании репозитория и синхронизации его с другими разработчиками, потребуется достаточно длительное время для скачивания данных, особенно, если проект большой, так как требуется скопировать на локальный компьютер весь репозиторий.

Git – гибкая, удобная и мощная система контроля версий, способная удовлетворить абсолютное большинство пользователей. Существующие недостатки постепенно удаляются и не приносят серьезных проблем пользователям. Если вы ведете большой проект, территориально удаленный, и тем более, если часто приходится разрабатывать

программное обеспечение, не имея доступа к другим разработчикам (например, вы не хотите терять время при перелете из страны в страну или во время поездки на работу), можно делать любые изменения и сохранять их в локальном репозитории, откатываться, переключаться между ветками и т.д.). Git – один из лидеров систем контроля версий.

1.8. Система управления версиями Mercurial.

Распределенная система контроля версий Mercurial разрабатывалась Мэттом Макалом параллельно с системой контроля версий Git, созданной Торвальдсом Линусом.

Первоначально, она была создана для эффективного управления большими проектами под Linux'ом, а поэтому была ориентирована на быструю и надежную работу с большими репозиториями. На данный момент mercurial адаптирован для работы под Windows, Mac OS X и большинство Unix систем.

Большая часть системы контроля версий написана на языке Python, и только отдельные участки программы, требующие наибольшего быстродействия, написаны на языке Си.

Идентификация ревизий происходит на основе алгоритма хеширования SHA1 (Secure Hash Algorithm 1), однако, также предусмотрена возможность присвоения ревизиям индивидуальных номеров.

Так же, как и в git'е, поддерживается возможность создания веток проекта с последующим их слиянием.

Для взаимодействия между клиентами используются протоколы HTTP, HTTPS или SSH.

Набор команд - простой и интуитивно понятный, во многом схожий с командами subversion. Так же имеется ряд графических оболочек и доступ к репозиторию через веб-интерфейс. Немаловажным является и наличие утилит, позволяющих импортировать репозитории многих других систем контроля версий.

Достоинства:

- Быстрая обработка данных.
- Кросплатформенная поддержка.
- Возможность работы с несколькими ветками проекта.
- Простота в обращение.

Возможность конвертирования репозиториев других систем поддержки версий, таких как CVS, Subversion, Git, Darcs, GNU Arch, Bazaar и др.

Недостатки:

- Возможные (но чрезвычайно низкие) совпадения хеш - кода отличных по содержанию ревизий.
- Ориентирован на работу в консоли.

Простой и отточенный интерфейс, и набор команд, возможность импортировать репозитории с других систем контроля версий, - сделают переход на Mercurial и обучение основным особенностями безболезненным и быстрым. Вряд ли это займет больше нескольких дней.

Надежность и скорость работы позволяют использовать его для контроля версий огромных проектов. Все это делает mercurial достойным конкурентом git'a.

2. СИСТЕМА КОНТРОЛЯ ВЕРСИЙ GIT

2.1. Базовые понятия

Репозиторий (repository, repo) — место, где СКВ хранит свои метаданные и базу данных объектов проекта

Локальный репозиторий — репозиторий, расположенный на локальном компьютере разработчика в каталоге. Именно в нём происходит разработка и фиксация изменений, которые отправляются на удалённый репозиторий.

Удалённый репозиторий — репозиторий, находящийся на удалённом сервере. Это общий репозиторий, в который приходят все изменения и из которого забираются все обновления.

Ветка (Branch) — это параллельная версия репозитория. Она включена в этот репозиторий, но не влияет на главную версию, тем самым позволяя свободно работать в параллельной. Когда вы внесли нужные изменения, то вы можете объединить их с главной версией.

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. Приведены английские термины, в литературе на русском языке используется тот или иной перевод или транслитерация.

amend

Внести изменения, не создавая новой версии — обычно когда разработчик ошибочно зафиксировал (commit) версию, но не залил (push) её на сервер.

blame

Понять, кто внёс изменение.

branch

Ветвь — 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.

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.

check-in, commit, submit

Создание новой версии, фиксация изменений. В некоторых СУВ (Subversion) — новая версия автоматически переносится в хранилище документов.

check-out, clone

Извлечение документа из хранилища и создание рабочей копии.

conflict

Конфликт — 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).

graft, backport, cherry-picking, transplant

Использовать встроенный в СУВ 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.

head, trunk

Основная версия — самая свежая версия для ветви/ствола, находящаяся в хранилище. Сколько ветвей, столько основных версий.

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.

pull, update

Получить новые версии из хранилища. В некоторых СУВ (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.

push

Залить новые версии в хранилище. Многие распределённые СУВ (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.

rebase

Использовать встроенный в СУВ алгоритм слияния, чтобы перенести точку ветвления (версию, от которой начинается ветвь) на более позднюю версию ствола. Чаще всего применяется в таком сценарии: Борис внёс изменения и обнаруживает, что не может залить (push) их, поскольку Анна ранее изменила совершенно другое место кода. Можно

просто слить их (merge). Но дерево будет линейным и более читабельным, если отказаться от своей редакции, но внести те же изменения в редакцию Анны — это и есть rebase. Если Анна и Борис работают над одним и тем же местом кода, мешая друг другу и разрешая конфликты вручную, rebase не рекомендуется.

repository, depot

Хранилище документов — место, где система управления версиями хранит все документы вместе с историей их изменения и другой служебной информацией.

revision

Версия документа. Системы управления версиями различают версии по номерам, которые назначаются автоматически.

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.

strip

Удалить целую ветвь из хранилища.

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.

trunk, mainline, master

Ствол — 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.

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.

working copy

Рабочая (локальная) копия документов.

2.2. Особенности Git

Слепки вместо патчей

Главное отличие Git'a от любых других СКВ (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.

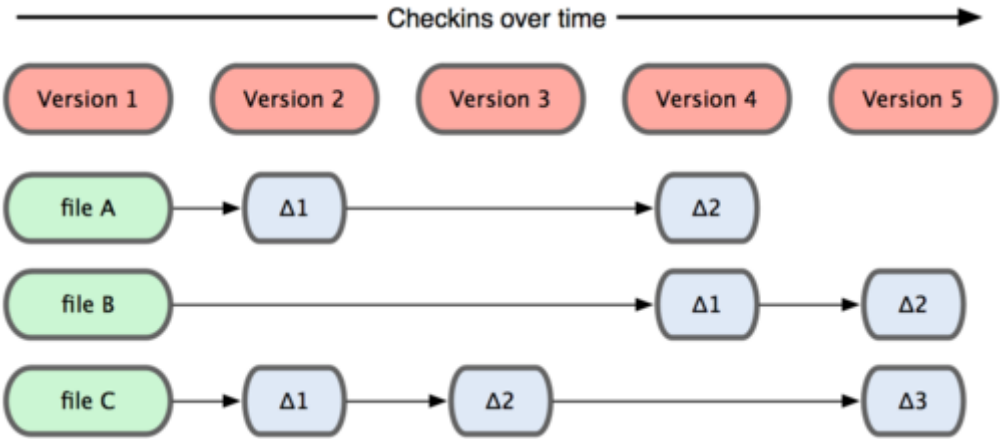

Рисунок 2.1. Другие системы хранят данные как изменения к базовой версии для каждого файла.

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.

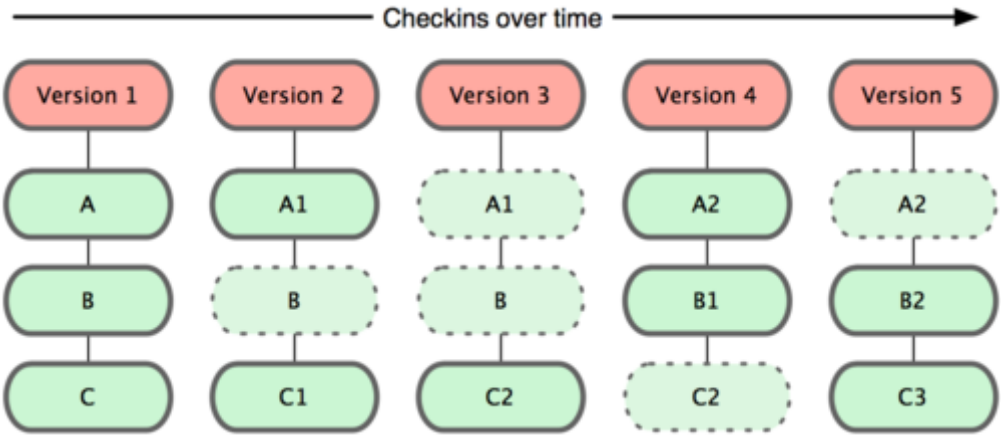

Рисунок 2.2. Git хранит данные как слепки состояний проекта во времени.

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 СКВ. Позже, коснувшись работы с ветвями в Git'е, мы узнаем, какие преимущества даёт такое понимание данных.

Почти все операции — локальные

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 СКВ-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 не возможно 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.

Чаще всего данные в Git только добавляются

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

Поэтому пользоваться Git'ом — удовольствие, потому что можно экспериментировать, не боясь что-то серьёзно поломать. Подробнее о том, как Git хранит свои данные и как восстановить то, что кажется уже потерянным, мы рассмотрим позже в курсе лекций.

Три состояния

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 отличаются по содержанию, но ещё не были зафиксированы.
- Подготовленные файлы — это изменённые файлы, отмеченные для включения в следующий коммит

В проектах, использующих Git, есть три части:

1. каталог Git’a (Git directory),
2. рабочий каталог (working directory),
3. область подготовленных файлов (staging area).

Рисунок 2.3. Рабочий каталог, область подготовленных файлов, каталог Git’a.

Каталог Git’a — это место, где Git хранит метаданные и базу данных объектов вашего проекта. Это наиболее важная часть Git’a, 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’a и помещаются на диск для того, чтобы вы их просматривали и редактировали.

Область подготовленных файлов — это обычный файл, обычно хранящийся в каталоге Git’a, 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).

2.3. Стандартный рабочий процесс с использованием Git

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. Поподробнее об этих трёх состояниях и как можно либо воспользоваться этим, либо пропустить стадию подготовки мы рассмотрим на следующей лекции.

2.4. Создание Git-репозитория

Для создания Git-репозитория существуют два основных подхода. Первый подход — импорт в Git уже существующего проекта или каталога. Второй — клонирование уже существующего репозитория с сервера.

Создание репозитория в существующем каталоге

Если вы собираетесь начать использовать Git для существующего проекта, то вам необходимо перейти в проектный каталог и в командной строке ввести

```
$ git init
```

Эта команда создаёт в текущем каталоге новый подкаталог с именем .git содержащий все необходимые файлы репозитория — основу Git-репозитория. На этом этапе ваш проект ещё не находится под версионным контролем. (Более подробно рассмотрим файлы содержащихся в только что созданном вами каталоге `.git` позже, в рамках следующих лекций)

Если вы хотите добавить под версионный контроль существующие файлы (в отличие от пустого каталога), вам стоит проиндексировать эти файлы и осуществить первую фиксацию изменений. Осуществить это вы можете с помощью нескольких команд `git add` указывающих индексируемые файлы, а затем `commit`:

```
$ git add *.c
$ git add README
$ git commit -m 'initial project version'
```

Мы разберём, что делают эти команды чуть позже. На данном этапе, у вас есть Git-репозиторий с добавленными файлами и начальным коммитом.

Клонирование существующего репозитория

Если вы желаете получить копию существующего репозитория Git, например, проекта, в котором вы хотите поучаствовать, то вам нужна команда `git clone`. Если вы знакомы с другими системами контроля версий, такими как Subversion, то заметите, что

команда называется `clone`, а не `checkout`. Это важное отличие — Git получает копию практически всех данных, что есть на сервере. Каждая версия каждого файла из истории проекта забирается (pulled) с сервера, когда вы выполняете `git clone`. Фактически, если серверный диск выйдет из строя, вы можете использовать любой из клонов на любом из клиентов, для того чтобы вернуть сервер в то состояние, в котором он находился в момент клонирования (вы можете потерять часть серверных перехватчиков (server-side hooks) и т.п., но все данные, помещённые под версионный контроль, будут сохранены, подробнее см. в главе 4).

Клонирование репозитория осуществляется командой `git clone [url]`. Например, если вы хотите клонировать библиотеку Ruby Git, известную как Grit, вы можете сделать это следующим образом:

```
$ git clone git://github.com/libgit2/rugged.git
```

Эта команда создаёт каталог с именем `rugged`, инициализирует в нём каталог `.git`, скачивает все данные для этого репозитория и создаёт (checks out) рабочую копию последней версии. Если вы зайдёте в новый каталог `rugged`, вы увидите в нём проектные файлы, пригодные для работы и использования. Если вы хотите клонировать репозиторий в каталог, отличный от `rugged`, можно это указать в следующем параметре командной строки:

```
$ git clone git://github.com/libgit2/rugged.git myrugget
```

Эта команда делает всё то же самое, что и предыдущая, только результирующий каталог будет назван myrugget.

В Git’е реализовано несколько транспортных протоколов, которые вы можете использовать. В предыдущем примере использовался протокол `git://`, вы также можете встретить `http(s)://` или `user@server:/path.git`, использующий протокол передачи SSH. В рамках следующих лекций мы познакомимся со всеми доступными вариантами конфигурации сервера для обеспечения доступа к вашему Git-репозиторию, а также рассмотрим их достоинства и недостатки.

2.5. Запись изменений в репозиторий

Итак, у вас имеется настоящий Git-репозиторий и рабочая копия файлов для некоторого проекта. Вам нужно делать некоторые изменения и фиксировать “снимки” состояния (snapshots) этих изменений в вашем репозитории каждый раз, когда проект достигает состояния, которое вам хотелось бы сохранить.

Запомните, 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 неизменёнными, изменёнными или подготовленными к коммиту (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 неизменёнными, 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 изменённые, т.к. вы изменили их с момента последнего коммита. Вы индексируете (stage) эти изменения и затем фиксируете все индексированные изменения, а затем цикл повторяется. Этот жизненный цикл изображён на рисунке.

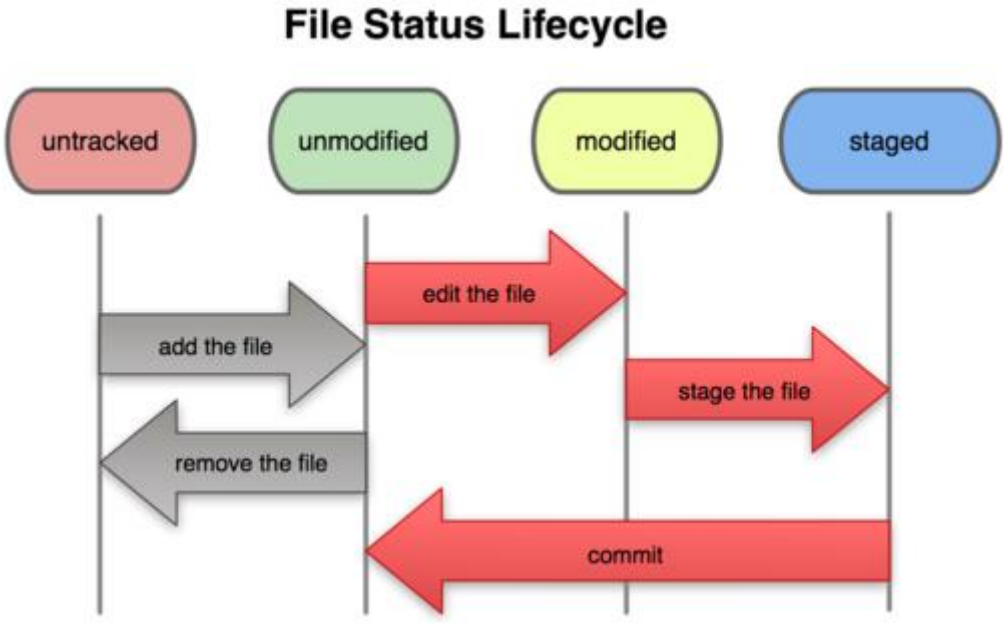

Рисунок 2.4. Жизненный цикл состояний файлов.

Определение состояния файлов

Основной инструмент, используемый для определения, какие файлы в каком состоянии находятся — это команда `git status`. Если вы выполните эту команду сразу после клонирования, вы увидите что-то вроде этого:

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Это означает, что у вас чистый рабочий каталог, другими словами — в нём нет отслеживаемых изменённых файлов. Git также не обнаружил неотслеживаемых файлов, в противном случае они бы были перечислены здесь. И наконец, команда сообщает вам на какой ветке (branch) вы сейчас находитесь. Пока что это всегда ветка `master` — это ветка по умолчанию. Позже подробно поработаем с ветками и ссылками.

Предположим, вы добавили в свой проект новый файл, простой файл README. Если этого файла раньше не было, и вы выполните `git status`, вы увидите свой неотслеживаемый файл вот так:

```
$ touch README
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       README
nothing added to commit but untracked files present (use "git add" to track)
```

Понять, что новый файл README неотслеживаемый можно по тому, что он находится в секции “Untracked files” в выводе команды `status`. Статус “неотслеживаемый

файл”, по сути, означает, что Git видит файл, отсутствующий в предыдущем снимке состояния (коммите); Git не станет добавлять его в ваши коммиты, пока вы его явно об этом не попросите. Это предохранит вас от случайного добавления в репозиторий сгенерированных бинарных файлов или каких-либо других, которые вы и не думали добавлять. Мы хотели добавить README, так давайте сделаем это.

Отслеживание новых файлов

Для того чтобы начать отслеживать (добавить под версионный контроль) новый файл, используется команда `git add`. Чтобы начать отслеживание файла README, вы можете выполнить следующее:

```
$ git add README
```

Если вы снова выполните команду `status`, то увидите, что файл README теперь отслеживаемый и индексированный:

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

Вы можете видеть, что файл проиндексирован по тому, что он находится в секции “Changes to be committed”. Если вы выполните коммит в этот момент, то версия файла, существовавшая на момент выполнения вами команды `git add`, будет добавлена в историю снимков состояния. Как вы помните, когда вы ранее выполнили `git init`, вы затем выполнили `git add (файлы)` — это было сделано для того, чтобы добавить файлы в вашем каталоге под версионный контроль. Команда `git add` принимает параметром путь к файлу или каталогу, если это каталог, команда рекурсивно добавляет (индексирует) все файлы в данном каталоге.

Индексация изменённых файлов

Давайте модифицируем файл, уже находящийся под версионным контролем. Если вы измените отслеживаемый файл benchmarks.rb и после этого снова выполните команду `status`, то результат будет примерно следующим:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   benchmarks.rb
#
```

Файл benchmarks.rb находится в секции “Changes not staged for commit” — это означает, что отслеживаемый файл был изменён в рабочем каталоге, но пока не проиндексирован. Чтобы проиндексировать его, необходимо выполнить команду `git add` (это многофункциональная команда, она используется для добавления под версионный контроль новых файлов, для индексации изменений, а также для других целей, например

для указания файлов с исправленным конфликтом слияния). Выполним `git add`, чтобы проиндексировать benchmarks.rb, а затем снова выполним `git status`:

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#       modified:   benchmarks.rb
#
```

Теперь оба файла проиндексированы и войдут в следующий коммит. В этот момент вы, предположим, вспомнили одно небольшое изменение, которое вы хотите сделать в benchmarks.rb до фиксации. Вы открываете файл, вносите и сохраняете необходимые изменения и вроде бы готовы к коммиту. Но давайте-ка ещё раз выполним `git status`:

```
$ vim benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#       modified:   benchmarks.rb
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   benchmarks.rb
#
```

Теперь benchmarks.rb отображается как проиндексированный и непроиндексированный одновременно. Как такое возможно? Такая ситуация наглядно демонстрирует, что Git индексирует файл в точности в том состоянии, в котором он находился, когда вы выполнили команду `git add`. Если вы выполните коммит сейчас, то файл benchmarks.rb попадёт в коммит в том состоянии, в котором он находился, когда вы последний раз выполняли команду `git add`, а не в том, в котором он находится в вашем рабочем каталоге в момент выполнения `git commit`. Если вы изменили файл после выполнения `git add`, вам придётся снова выполнить `git add`, чтобы проиндексировать последнюю версию файла:

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#       modified:   benchmarks.rb
#
```

Игнорирование файлов

Зачастую, у вас имеется группа файлов, которые вы не только не хотите автоматически добавлять в репозиторий, но и видеть в списках неотслеживаемых. К таким файлам обычно относятся автоматически генерируемые файлы (различные логи, результаты сборки программ и т.п.). В таком случае, вы можете создать файл .gitignore с перечислением шаблонов соответствующих таким файлам. Вот пример файла .gitignore:

```
$ cat .gitignore
*.[oa]
*~
```

Первая строка предписывает Git'у игнорировать любые файлы заканчивающиеся на .o или .a — объектные и архивные файлы, которые могут появиться во время сборки кода. Вторая строка предписывает игнорировать все файлы заканчивающиеся на тильду (~), которая используется во многих текстовых редакторах, например Emacs, для обозначения временных файлов. Вы можете также включить каталоги log, tmp или pid; автоматически создаваемую документацию; и т.д. и т.п. Хорошая практика заключается в настройке файла .gitignore до того, как начать серьёзно работать, это защитит вас от случайного добавления в репозиторий файлов, которых вы там видеть не хотите.

К шаблонам в файле .gitignore применяются следующие правила:

- Пустые строки, а также строки, начинающиеся с #, игнорируются.
- Можно использовать стандартные glob шаблоны.
- Можно заканчивать шаблон символом слэша (`/`) для указания каталога.
- Можно инвертировать шаблон, использовав восклицательный знак (`!`) в качестве первого символа.

Glob-шаблоны представляют собой упрощённые регулярные выражения, используемые командными интерпретаторами. Символ `*` соответствует 0 или более символам; последовательность `[abc]` — любому символу из указанных в скобках (в данном примере a, b или c); знак вопроса (`?`) соответствует одному символу; `[0-9]` соответствует любому символу из интервала (в данном случае от 0 до 9).

Вот ещё один пример файла .gitignore:

```
# комментарий — эта строка игнорируется
# не обрабатывать файлы, имя которых заканчивается на .a
*.a
# НО отслеживать файл lib.a, несмотря на то, что мы игнорируем все .a файлы с
помощью предыдущего правила
!lib.a
# игнорировать только файл TODO находящийся в корневом каталоге, не относится
к файлам вида subdir/TODO
/TODO
# игнорировать все файлы в каталоге build/
build/
# игнорировать doc/notes.txt, но не doc/server/arch.txt
doc/*.txt
# игнорировать все .txt файлы в каталоге doc/
doc/**/*.txt
```

Шаблон `**/` доступен в Git, начиная с версии 1.8.2.

Просмотр индексированных и неиндексированных изменений

Если результат работы команды `git status` недостаточно информативен для вас — вам хочется знать, что конкретно поменялось, а не только какие файлы были изменены — вы можете использовать команду `git diff`. Позже мы рассмотрим команду `git diff` подробнее; вы, скорее всего, будете использовать эту команду для получения ответов на два вопроса: что вы изменили, но ещё не проиндексировали, и что вы проиндексировали и собираетесь фиксировать. Если `git status` отвечает на эти вопросы слишком обобщённо,

то `git diff` показывает вам непосредственно добавленные и удалённые строки — собственно заплатку (patch).

Допустим, вы снова изменили и проиндексировали файл README, а затем изменили файл benchmarks.rb без индексирования. Если вы выполните команду `status`, вы опять увидите что-то вроде:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   benchmarks.rb
#
```

Чтобы увидеть, что же вы изменили, но пока не проиндексировали, наберите `git diff` без аргументов:

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
           @commit.parents[0].parents[0].parents[0]
         end

+        run_code(x, 'commits 1') do
+          git.commits.size
+        end
+
         run_code(x, 'commits 2') do
           log = git.commits('master', 15)
           log.size
```

Эта команда сравнивает содержимое вашего рабочего каталога с содержимым индекса. Результат показывает ещё не проиндексированные изменения.

Если вы хотите посмотреть, что вы проиндексировали и что войдёт в следующий коммит, вы можете выполнить `git diff --cached`. (В Git’е версии 1.6.1 и выше, вы также можете использовать `git diff --staged`, которая легче запоминается.) Эта команда сравнивает ваши индексированные изменения с последним коммитом:

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+grit
+ by Tom Preston-Werner, Chris Wanstrath
+ http://github.com/mojombo/grit
+
+Grit is a Ruby library for extracting information from a Git repository
```

Важно отметить, что `git diff` сама по себе не показывает все изменения, сделанные с последнего коммита — только те, что ещё не проиндексированы. Такое

поведение может сбивать с толку, так как если вы проиндексируете все свои изменения, то `git diff` ничего не вернёт.

Другой пример: вы проиндексировали файл benchmarks.rb и затем изменили его, вы можете использовать `git diff` для просмотра как индексированных изменений в этом файле, так и тех, что пока не проиндексированы:

```
$ git add benchmarks.rb
$ echo '# test line' >> benchmarks.rb
$ git status
# On branch master
#
# Changes to be committed:
#
#       modified:   benchmarks.rb
#
# Changes not staged for commit:
#
#       modified:   benchmarks.rb
#
```

Теперь вы можете используя `git diff` посмотреть непроиндексированные изменения

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index e445e28..86b2f7c 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -127,3 +127,4 @@ end
 main()

 ##pp Grit::GitRuby.cache_client.stats
+# test line
```

а также уже проиндексированные, используя `git diff --cached`:

```
$ git diff --cached
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..e445e28 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
           @commit.parents[0].parents[0].parents[0]
         end

+        run_code(x, 'commits 1') do
+          git.commits.size
+        end
+
         run_code(x, 'commits 2') do
           log = git.commits('master', 15)
           log.size
```

Фиксация изменений

Теперь, когда ваш индекс настроен так, как вам и хотелось, вы можете зафиксировать свои изменения. Запомните, всё, что до сих пор не проиндексировано — любые файлы, созданные или изменённые вами, и для которых вы не выполнили `git add` после момента редактирования — не войдут в этот коммит. Они останутся изменёнными файлами на вашем диске. В нашем случае, когда вы в последний раз выполняли `git`

`status`, вы видели что всё проиндексировано, и вот, вы готовы к коммиту. Простейший способ зафиксировать изменения — это набрать `git commit`:

```
$ git commit
```

Эта команда откроет выбранный вами текстовый редактор. (Редактор устанавливается системной переменной `$EDITOR` — обычно это vim или emacs, хотя вы можете установить ваш любимый с помощью команды `git config --global core.editor`, как было показано в предыдущей лекции).

В редакторе будет отображён следующий текст (это пример окна Vim’a):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#       modified:   benchmarks.rb
~
~
~
".git/COMMIT_EDITMSG" 10L, 283C
```

Вы можете видеть, что комментарий по умолчанию для коммита содержит закомментированный результат работы (“выхлоп”) команды `git status` и ещё одну пустую строку сверху. Вы можете удалить эти комментарии и набрать своё сообщение или же оставить их для напоминания о том, что вы фиксируете. (Для ещё более подробного напоминания, что же именно вы поменяли, можете передать аргумент `-v` в команду `git commit`. Это приведёт к тому, что в комментарий будет также помещена дельта/diff изменений, таким образом вы сможете точно увидеть всё, что сделано.) Когда вы выходите из редактора, Git создаёт для вас коммит с этим сообщением (удаляя комментарии и вывод diff’a).

Есть и другой способ — вы можете набрать свой комментарий к коммиту в командной строке вместе с командой `commit`, указав его после параметра -m, как в следующем примере:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master]: created 463dc4f: "Fix benchmarks for speed"
 2 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 README
```

Итак, вы создали свой первый коммит! Вы можете видеть, что коммит вывел вам немного информации о себе: на какую ветку вы выполнили коммит (master), какая контрольная сумма SHA-1 у этого коммита (`463dc4f`), сколько файлов было изменено, а также статистику по добавленным/удалённым строкам в этом коммите.

Запомните, что коммит сохраняет снимок состояния вашего индекса. Всё, что вы не проиндексировали, так и торчит в рабочем каталоге как изменённое; вы можете сделать ещё один коммит, чтобы добавить эти изменения в репозиторий. Каждый раз, когда вы делаете коммит, вы сохраняете снимок состояния вашего проекта, который позже вы можете восстановить или с которым можно сравнить текущее состояние.

Игнорирование индексации

Несмотря на то, что индекс может быть удивительно полезным для создания коммитов именно такими, как вам и хотелось, он временами несколько сложнее, чем вам нужно в процессе работы. Если у вас есть желание пропустить этап индексирования, Git предоставляет простой способ. Добавление параметра `-a` в команду `git commit` заставляет Git автоматически индексировать каждый уже отслеживаемый на момент коммита файл, позволяя вам обойтись без `git add`:

```
$ git status
# On branch master
#
# Changes not staged for commit:
#
#       modified:   benchmarks.rb
#
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 files changed, 5 insertions(+), 0 deletions(-)
```

Обратите внимание на то, что в данном случае перед коммитом вам не нужно выполнять `git add` для файла benchmarks.rb.

Удаление файлов

Для того чтобы удалить файл из Git'а, 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".

Если вы просто удалите файл из своего рабочего каталога, он будет показан в секции "Changes not staged for commit" ("Изменённые но не обновлённые" — читай не проиндексированные) вывода команды `git status`:

```
$ rm grit.gemspec
$ git status
# On branch master
#
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#
#       deleted:    grit.gemspec
#
```

Затем, если вы выполните команду `git rm`, удаление файла попадёт в индекс:

```
$ git rm grit.gemspec
rm 'grit.gemspec'
$ git status
# On branch master
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    grit.gemspec
#
```

После следующего коммита файл исчезнет и больше не будет отслеживаться. Если вы изменили файл и уже проиндексировали его, вы должны использовать принудительное удаление с помощью параметра `-f`. Это сделано для повышения безопасности, чтобы

предотвратить ошибочное удаление данных, которые ещё не были записаны в снимок состояния и которые нельзя восстановить из Git'а.

Другая полезная штука, которую вы можете захотеть сделать — это удалить файл из индекса, оставив его при этом в рабочем каталоге. Другими словами, вы можете захотеть оставить файл на винчестере, и убрать его из-под бдительного ока Git'а. Это особенно полезно, если вы забыли добавить что-то в файл `.gitignore` и по ошибке проиндексировали, например, большой файл с логами, или кучу промежуточных файлов компиляции. Чтобы сделать это, используйте опцию `--cached`:

```
$ git rm --cached readme.txt
```

В команду `git rm` можно передавать файлы, каталоги или glob-шаблоны. Это означает, что вы можете вытворять что-то вроде:

```
$ git rm log/\*.log
```

Обратите внимание на обратный слэш (`\`) перед `*`. Он необходим из-за того, что Git использует свой собственный обработчик имён файлов вдобавок к обработчику вашего командного интерпретатора. Эта команда удаляет все файлы, которые имеют расширение `.log` в каталоге `log/`. Или же вы можете сделать вот так:

```
$ git rm \*~
```

Эта команда удаляет все файлы, чьи имена заканчиваются на `~`.

Перемещение файлов

В отличие от многих других систем версионного контроля, Git не отслеживает перемещение файлов явно. Когда вы переименовываете файл в Git'е, в нём не сохраняется никаких метаданных, говорящих о том, что файл был переименован. Однако, Git довольно умён в плане обнаружения перемещений постфактум — мы рассмотрим обнаружение перемещения файлов чуть позже.

Таким образом, наличие в Git'е команды `mv` выглядит несколько странным. Если вам хочется переименовать файл в Git'е, вы можете сделать что-то вроде:

```
$ git mv file_from file_to
```

и это отлично сработает. На самом деле, если вы выполните что-то вроде этого и посмотрите на статус, вы увидите, что Git считает, что произошло переименование файла:

```
$ git mv README.txt README
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    README.txt -> README
#
```

Однако, это эквивалентно выполнению следующих команд:

```
$ mv README.txt README
$ git rm README.txt
$ git add README
```

Git неявно определяет, что произошло переименование, поэтому неважно, переименуете вы файл так или используя команду `mv`. Единственное отличие состоит лишь в том, что `mv` — это одна команда вместо трёх — это функция для удобства. Важнее

другое — вы можете использовать любой удобный способ, чтобы переименовать файл, и затем воспользоваться add/rm перед коммитом.

2.6. Просмотр истории коммитов

После того как вы создадите несколько коммитов, или же вы склонируете репозиторий с уже существующей историей коммитов, вы, вероятно, захотите оглянуться назад и узнать, что же происходило с этим репозиторием. Наиболее простой и в то же время мощный инструмент для этого — команда `git log`.

Данные примеры используют очень простой проект, названный simplegit, который я часто использую для демонстраций. Чтобы получить этот проект, выполните:

В результате выполнения `git log` в данном проекте, вы должны получить что-то вроде этого:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

По умолчанию, без аргументов, `git log` выводит список коммитов созданных в данном репозитории в обратном хронологическом порядке. То есть самые последние коммиты показываются первыми. Как вы можете видеть, эта команда отображает каждый коммит вместе с его контрольной суммой SHA-1, именем и электронной почтой автора, датой создания и комментарием.

Существует превеликое множество параметров команды `git log` и их комбинаций, для того чтобы показать вам именно то, что вы ищете. Здесь мы покажем вам несколько наиболее часто применяемых.

Один из наиболее полезных параметров — это `-p`, который показывает дельту (разницу/diff), привнесенную каждым коммитом. Вы также можете использовать `-2`, что ограничит вывод до 2-х последних записей.

Этот параметр показывает ту же самую информацию плюс внесённые изменения, отображаемые непосредственно после каждого коммита. Это очень удобно для инспекций кода или для того, чтобы быстро посмотреть, что происходило в результате последовательности коммитов, добавленных коллегой.

В некоторых ситуациях гораздо удобней просматривать внесённые изменения на уровне слов, а не на уровне строк. Чтобы получить дельту по словам вместо обычной

дельты по строкам, нужно дописать после команды `git log -p` опцию `--word-diff`. Дельты на уровне слов практически бесполезны при работе над программным кодом, но они буду очень кстати при работе над длинным текстом, таким как книга или диссертация. Рассмотрим пример:

```
$ git log -U1 --word-diff
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -7,3 +7,3 @@ spec = Gem::Specification.new do |s|
    s.name      =   "simplegit"
    s.version   =   [-"0.1.0"-]{+"0.1.1"+}
    s.author    =   "Scott Chacon"
```

Как видите, в этом выводе нет ни добавленных ни удалённых строк, как для обычного diff'a. Вместо этого изменения показаны внутри строки. Добавленное слово заключено в `{+ +}`, а удалённое в `[- -]`. Также может быть полезно сократить обычные три строки контекста в выводе команды `diff` до одной строки, так как контекстом в данном случае являются слова, а не строки. Сделать это можно с помощью опции `-U1` как было показано в примере выше.

С командой `git log` вы также можете использовать группы суммирующих параметров. Например, если вы хотите получить некоторую краткую статистику по каждому коммиту, вы можете использовать параметр `--stat`:

Как видно из лога, параметр `--stat` выводит под каждым коммитом список изменённых файлов, количество изменённых файлов, а также количество добавленных и удалённых строк в этих файлах. Он также выводит сводную информацию в конце. Другой действительно полезный параметр — это `--pretty`. Он позволяет изменить формат вывода лога. Для вас доступны несколько предустановленных вариантов. Параметр `oneline` выводит каждый коммит в одну строку, что удобно если вы просматриваете большое количество коммитов. В дополнение к этому, параметры `short`, `full`, и `fuller`, практически не меняя формат вывода, позволяют выводить меньше или больше деталей соответственно:

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test code
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

Наиболее интересный параметр — это `format`, который позволяет вам полностью создать собственный формат вывода лога. Это особенно полезно, когда вы создаёте отчёты для автоматического разбора (парсинга) — поскольку вы явно задаёте формат и уверены в том, что он не будет изменяться при обновлениях Git'a:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 11 months ago : changed the version number
085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code
a11bef0 - Scott Chacon, 11 months ago : first commit
```

Следующий содержит список наиболее полезных параметров формата.

```
Параметр        Описание выводимых данных
%H              Хеш коммита
%h              Сокращённый хеш коммита
%T              Хеш дерева
%t              Сокращённый хеш дерева
%P              Хеши родительских коммитов
%p              Сокращённые хеши родительских коммитов
%an             Имя автора
%ae             Электронная почта автора
%ad             Дата автора (формат соответствует параметру `--date=`)
%ar             Дата автора, относительная (пр. "2 мес. назад")
%cn             Имя коммитера
%ce             Электронная почта коммитера
%cd             Дата коммитера
%cr             Дата коммитера, относительная
%s           Комментарий
```

Вас может заинтересовать, в чём же разница между *автором* и *коммитером*. Автор — это человек, изначально сделавший работу, тогда как коммитер — это человек, который последним применил эту работу. Так что если вы послали патч (заплатку) в проект и один из основных разработчиков применил этот патч, вы оба не будете забыты — вы как автор, а разработчик как коммитер.

Параметры `oneline` и `format` также полезны с другим параметром команды `log` — `--graph`. Этот параметр добавляет миленький ASCII-граф, показывающий историю ветвлений и слияний.

Ограничение вывода команды log

Кроме опций для форматирования вывода, `git log` имеет ряд полезных ограничительных параметров, то есть параметров, которые дают возможность отобразить часть коммитов. Вы уже видели один из таких параметров — параметр `-2`, который отображает только два последних коммита. На самом деле, вы можете задать `-<n>`, где `n` это количество отображаемых коммитов. На практике вам вряд ли придётся часто этим пользоваться потому, что по умолчанию Git через канал (pipe) отправляет весь вывод на pager, так что вы всегда будете видеть только одну страницу.

А вот параметры, ограничивающие по времени, такие как `--since` и `--until`, весьма полезны. Например, следующая команда выдаёт список коммитов, сделанных за последние две недели:

```
$ git log --since=2.weeks
```

Такая команда может работать с множеством форматов — вы можете указать точную дату (“2008-01-15”) или относительную дату, такую как “2 years 1 day 3 minutes ago”.

Вы также можете отфильтровать список коммитов по какому-либо критерию поиска. Опция `--author` позволяет фильтровать по автору, опция `--grep` позволяет искать по ключевым словам в сообщении. (Заметим, что, если вы укажете и опцию `author`, и опцию `grep`, то будут найдены все коммиты, которые удовлетворяют первому ИЛИ второму критерию. Чтобы найти коммиты, которые удовлетворяют первому И второму критерию, следует добавить опцию `--all-match`.)

Последняя действительно полезная опция-фильтр для `git log` — это путь. Указав имя каталога или файла, вы ограничите вывод log теми коммитами, которые вносят

изменения в указанные файлы. Эта опция всегда указывается последней и обычно предваряется двумя минусами (--), чтобы отделить пути от остальных опций.

В спсике для справки приведён список часто употребляемых опций.

```
Опция               Описание
-(n)                Показать последние n коммитов
--since, --after        Ограничить коммиты теми, которые сделаны после
указанной даты.
--until, --before       Ограничить коммиты теми, которые сделаны до указанной
даты.
--author                Показать только те коммиты, автор которых
соответствует указанной строке.
--committer             Показать только те коммиты, коммитер которых
соответствует указанной строке.
```

Использование графического интерфейса для визуализации истории

Если у вас есть желание использовать какой-нибудь графический инструмент для визуализации истории коммитов, можно попробовать распространяемую вместе с Git'ом программу gitk, написанную на Tcl/Tk. В сущности gitk — это наглядный вариант `git log`, к тому же он принимает почти те же фильтрующие опции, что и `git log`. Если наберёте в командной строке gitk, находясь в проекте, то увидите что-то наподобие.

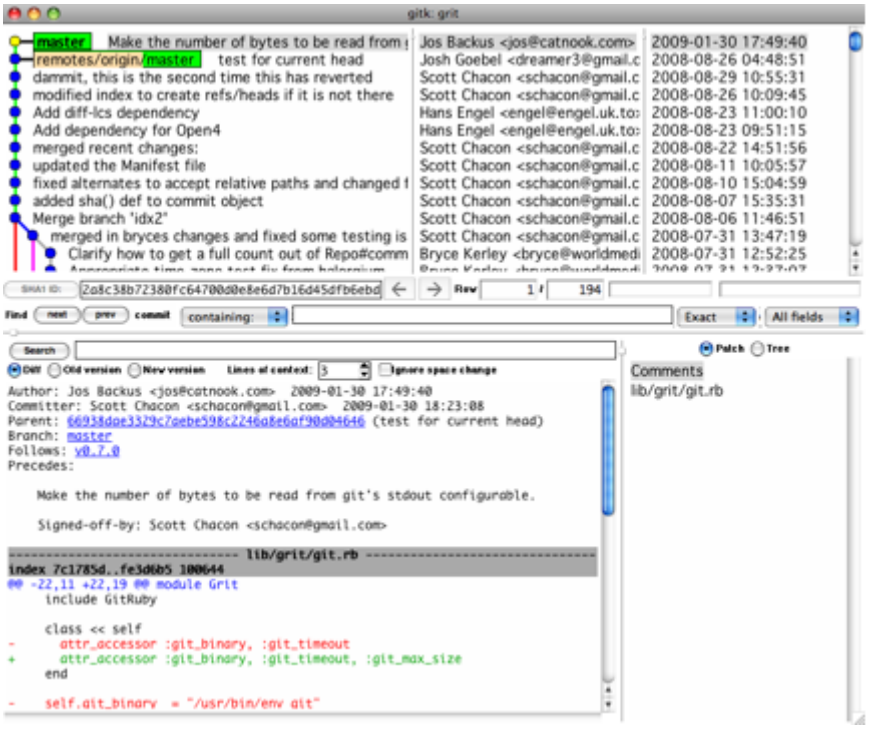

Рисунок 2.5. Визуализация истории с помощью gitk.

В верхней части окна располагается история коммитов вместе с подробным графом наследников. Просмотрщик дельт в нижней половине окна отображает изменения, сделанные выбранным коммитом. Указать коммит можно с помощью щелчка мышью.

2.7. Отмена изменений

На любой стадии может возникнуть необходимость что-либо отменить. Здесь мы рассмотрим несколько основных инструментов для отмены произведённых изменений. Будьте осторожны, ибо не всегда можно отменить сами отмены. Это одно из немногих мест в Git'е, где вы можете потерять свою работу если сделаете что-то неправильно.

Изменение последнего коммита

Одна из типичных отмен происходит тогда, когда вы делаете коммит слишком рано, забыв добавить какие-то файлы, или напутали с комментарием к коммиту. Если вам хотелось бы сделать этот коммит ещё раз, вы можете выполнить 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 комментарием к последнему коммиту. Вы можете отредактировать это сообщение так же, как обычно, и оно перепишет предыдущее.

Для примера, если после совершения коммита вы осознали, что забыли проиндексировать изменения в файле, которые хотели добавить в этот коммит, вы можете сделать что-то подобное:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

Все три команды вместе дают один коммит — второй коммит заменяет результат первого.

Отмена индексации файла

В следующих двух разделах мы продемонстрируем, как переделать изменения в индексе и в рабочем каталоге. Приятно то, что команда, используемая для определения состояния этих двух вещей, дополнительно напоминает о том, как отменить изменения в них. Приведём пример. Допустим, вы внесли изменения в два файла и хотите записать их как два отдельных коммита, но случайно набрали `git add .` и проиндексировали оба файла. Как теперь отменить индексацию одного из двух файлов? Команда `git status` напомнит вам об этом:

```
$ git add .
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#       modified:   benchmarks.rb
#
```

Сразу после надписи “Changes to be committed”, написано использовать `git reset HEAD <файл>...` для исключения из индекса. Так что давайте последуем совету и отменим индексацию файла benchmarks.rb:

```
$ git reset HEAD benchmarks.rb
benchmarks.rb: locally modified
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   benchmarks.rb
#
```

Эта команда немного странновата, но она работает. Файл benchmarks.rb изменён, но снова не в индексе.

Отмена изменений файла

Что, если вы поняли, что не хотите оставлять изменения, внесённые в файл benchmarks.rb? Как быстро отменить изменения, вернуть то состояние, в котором он находился во время последнего коммита (или первоначального клонирования, или какого-то другого действия, после которого файл попал в рабочий каталог)? К счастью, `git status` говорит, как добиться и этого. В выводе для последнего примера, неиндексированная область выглядит следующим образом:

```
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   benchmarks.rb
#
```

Здесь довольно ясно сказано, как отменить сделанные изменения (по крайней мере новые версии Git’a, начиная с 1.6.1, делают это; если у вас версия старее, мы настоятельно рекомендуем обновиться, чтобы получать такие подсказки и сделать свою работу удобней). Давайте сделаем то, что написано:

```
$ git checkout -- benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#
```

Как вы видите, изменения были отменены. Вы должны понимать, что это опасная команда: все сделанные вами изменения в этом файле пропали — вы просто скопировали поверх него другой файл. Никогда не используйте эту команду, если вы не полностью уверены, что этот файл вам не нужен. Если вам нужно просто сделать, чтобы он не мешался, мы рассмотрим прятание (stash) и ветвление; эти способы обычно более предпочтительны.

Помните, что всё, что является частью коммита в Git'е, почти всегда может быть восстановлено. Даже коммиты, которые находятся на ветках, которые были удалены, и коммиты переписанные с помощью `--amend` могут быть восстановлены. Несмотря на это, всё, что никогда не попадало в коммит, вы скорее всего уже не увидите снова.

2.8. Работа с удалёнными репозиториями

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 (push) и получение (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 отслеживаемые (tracked) или нет и прочее. Данный раздел охватывает все перечисленные навыки по управлению удалёнными репозиториями.

Отображение удалённых репозиториев

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 склонировали:

```
$ git clone git://github.com/schacon/ticgit.git
Initialized empty Git repository in /private/tmp/ticgit/.git/
remote: Counting objects: 595, done.
remote: Compressing objects: 100% (269/269), done.
remote: Total 595 (delta 255), reused 589 (delta 253)
Receiving objects: 100% (595/595), 73.31 KiB | 1 KiB/s, done.
Resolving deltas: 100% (255/255), done.
$ cd ticgit
$ git remote
origin
```

Чтобы посмотреть, какому URL соответствует сокращённое имя в Git, можно указать команде опцию `-v`:

```
$ git remote -v
origin  git://github.com/schacon/ticgit.git (fetch)
origin  git://github.com/schacon/ticgit.git (push)
```

Если у вас больше одного удалённого репозитория, команда покажет их все. Например, мой репозиторий Grit выглядит следующим образом.

```
$ cd grit
$ git remote -v
bakkdoor  git://github.com/bakkdoor/grit.git
cho45     git://github.com/cho45/grit.git
defunkt   git://github.com/defunkt/grit.git
koke      git://github.com/koke/grit.git
origin    git@github.com:mojombo/grit.git
```

Это означает, что мы легко можем получить изменения от любого из этих пользователей. Но, заметьте, что origin — это единственный удалённый сервер прописанный как SSH-ссылка, поэтому он единственный, в который я могу помещать свои изменения (мы рассмотрим подробнее этот момент в рамках следуюзих лекций).

Добавление удалённых репозиториев

В предыдущих разделах мы упомянули и немного продемонстрировали добавление удалённых репозиториев, сейчас мы рассмотрим это более детально. Чтобы добавить новый удалённый Git-репозиторий под именем-сокращением, к которому будет проще обращаться, выполните `git remote add [сокращение] [url]`:

```
$ git remote
origin
$ git remote add pb git://github.com/paulboone/ticgit.git
$ git remote -v
origin  git://github.com/schacon/ticgit.git
pb      git://github.com/paulboone/ticgit.git
```

Теперь вы можете использовать в командной строке имя pb вместо полного URL. Например, если вы хотите извлечь (fetch) всю информацию, которая есть в репозитории Павла, но нет в вашем, вы можете выполнить `git fetch pb`:

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

Ветка `master` Павла теперь доступна локально как `pb/master`. Вы можете слить (merge) её в одну из своих веток или перейти на эту ветку, если хотите её проверить.

Fetch и Pull

Как вы только что узнали, для получения данных из удалённых проектов, следует выполнить:

```
$ git fetch [имя удал. сервера]
```

Данная команда связывается с указанным удалённым проектом 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. Теперь эти ветки в любой момент могут быть просмотрены или слиты.

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 (для дополнительной информации смотри следующий раздел и главу 3), то вы можете использовать команду `git pull`. Она автоматически 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.

Push

Когда вы хотите поделиться своими наработками, вам необходимо отправить (push) их в главный репозиторий. Команда для этого действия простая: `git push [удал. сервер] [ветка]`. Чтобы отправить вашу ветку master на сервер `origin` (повторимся, что клонирование, как правило, настраивает оба этих имени автоматически), вы можете выполнить следующую команду для отправки наработок 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 вытянуть (pull) их изменения и объединить с вашими. Только после этого вам будет позволено выполнить push.

Инспекция удалённого репозитория

Если хотите получить побольше информации об одном из удалённых репозиториев, вы можете использовать команду `git remote show [удал. сервер]`. Если вы выполните эту команду с некоторым именем, например, `origin`, вы получите что-то подобное:

```
$ git remote show origin
* remote origin
  URL: git://github.com/schacon/ticgit.git
  Remote branch merged with 'git pull' while on branch master
    master
  Tracked remote branches
    master
    ticgit
```

Она выдаёт URL удалённого репозитория, а также информацию об отслеживаемых ветках. Эта команда любезно сообщает вам, что если вы, находясь на ветке master, выполните `git pull`, ветка master с удалённого сервера будет автоматически влита в вашу сразу после получения всех необходимых данных. Она также выдаёт список всех полученных ею ссылок.

Это был пример для простой ситуации, и наверняка вы встретились с чем-то подобным. Однако, если вы используете Git более интенсивно, вы можете увидеть гораздо большее количество информации от `git remote show`:

```
$ git remote show origin
* remote origin
  URL: git@github.com:defunkt/github.git
  Remote branch merged with 'git pull' while on branch issues
    issues
```

```
  Remote branch merged with 'git pull' while on branch master
    master
  New remote branches (next fetch will store in remotes/origin)
    caching
  Stale tracking branches (use 'git remote prune')
    libwalker
    walker2
  Tracked remote branches
    acl
    apiv2
    dashboard2
    issues
    master
    postgres
  Local branch pushed with 'git push'
    master:master
```

Данная команда показывает какая именно локальная ветка будет отправлена на удалённый сервер по умолчанию при выполнении `git push`. Она также показывает, каких веток с удалённого сервера у вас ещё нет, какие ветки всё ещё есть у вас, но уже удалены на сервере. И для нескольких веток показано, какие удалённые ветки будут в них влиты при выполнении `git pull`.

Удаление и переименование удалённых репозиториев

Для переименования ссылок в новых версиях Git'а можно выполнить `git remote rename`, это изменит сокращённое имя, используемое для удалённого репозитория. Например, если вы хотите переименовать `pb` в `paul`, вы можете сделать это следующим образом:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

Стоит упомянуть, что это также меняет для вас имена удалённых веток. То, к чему вы обращались как `pb/master`, стало `paul/master`.

Если по какой-то причине вы хотите удалить ссылку (вы сменили сервер или больше не используете определённое зеркало, или, возможно, контрибьютор перестал быть активным), вы можете использовать `git remote rm`:

```
$ git remote rm paul
$ git remote
Origin
```

2.9. Работа с метками

Как и большинство СКВ, Git имеет возможность помечать (tag) определённые моменты в истории как важные. Как правило, этот функционал используется для отметки моментов выпуска версий (v1.0, и т.п.). В этом разделе вы узнаете, как посмотреть имеющиеся метки (tag), как создать новые. А также вы узнаете, что из себя представляют разные типы меток.

Просмотр меток

Просмотр имеющихся меток (tag) в Git’е делается просто. Достаточно набрать `git tag`:

```
$ git tag
v0.1
v1.3
```

Данная команда перечисляет метки в алфавитном порядке; порядок их появления не имеет значения.

Для меток вы также можете осуществлять поиск по шаблону. Например, репозиторий Git’а содержит более 240 меток. Если вас интересует просмотр только выпусков 1.4.2, вы можете выполнить следующее:

```
$ git tag -l 'v1.4.2.*'
v1.4.2.1
v1.4.2.2
v1.4.2.3
v1.4.2.4
```

Создание меток

Git использует два основных типа меток: легковесные и аннотированные. Легковесная метка — это что-то весьма похожее на ветку, которая не меняется — это просто указатель на определённый коммит. А вот аннотированные метки хранятся в базе данных Git’а как полноценные объекты. Они имеют контрольную сумму, содержат имя поставившего метку, e-mail и дату, имеют комментарий и могут быть подписаны и проверены с помощью GNU Privacy Guard (GPG). Обычно рекомендуется создавать аннотированные метки, чтобы иметь всю перечисленную информацию; но если вы хотите сделать временную метку или по какой-то причине не хотите сохранять остальную информацию, то для этого годятся и легковесные метки.

Аннотированные метки

Создание аннотированной метки в Git’е выполняется легко. Самый простой способ это указать `-a` при выполнении команды `tag`:

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

Опция `-m` задаёт меточное сообщение, которое будет храниться вместе с меткой. Если не указать сообщение для аннотированной метки, Git запустит редактор, чтоб вы смогли его ввести.

Вы можете посмотреть данные метки вместе с коммитом, который был помечен, с помощью команды `git show`:

```
$ git show v1.4
tag v1.4
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 14:45:11 2009 -0800

my version 1.4
```

```
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

Она показывает информацию о выставившем метку, дату отметки коммита и аннотирующее сообщение перед информацией о коммите.

Подписанные метки

Вы также можете подписывать свои метки с помощью GPG, конечно, если у вас есть ключ. Всё что нужно сделать, это использовать `-s` вместо `-a`:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gee-mail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

Если вы выполните `git show` на этой метке, то увидите прикреплённую к ней GPG-подпись:

```
$ git show v1.5
tag v1.5
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:22:20 2009 -0800

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.8 (Darwin)

iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN
Ki0An2JeAVUCAiJ7Ox6ZEtK+NvZAj82/
=WryJ
-----END PGP SIGNATURE-----
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

Чуть позже вы узнаете, как верифицировать метки с подписью.

Легковесные метки

Легковесная метка — это ещё один способ отметки коммитов. В сущности, это контрольная сумма коммита, сохранённая в файл — больше никакой информации не хранится. Для создания легковесной метки не передавайте опций `-a`, `-s` и `-m`:

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

На этот раз при выполнении `git show` на этой метке вы не увидите дополнительной информации. Команда просто покажет помеченный коммит:

```
$ git show v1.4-lw
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
```

```
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

Верификация меток

Для верификации подписанной метки, используйте `git tag -v [имя метки]`. Эта команда использует GPG для верификации подписи. Вам нужен открытый ключ автора подписи, чтобы команда работала правильно:

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
gpg:                 aka "[jpeg image of size 1513]"
Primary key fingerprint: 3565 2A26 2040 E066 C9A7  4A7D C0C6 D9A4 F311 9B9A
```

Если у вас нет открытого ключа автора подписи, вы вместо этого получите что-то подобное:

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

Выставление меток позже

Также возможно помечать уже пройденные коммиты. Предположим, что история коммитов выглядит следующим образом:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Теперь предположим, что вы забыли отметить версию проекта v1.2, которая была там, где находится коммит “updated rakefile”. Вы можете добавить метку и позже. Для отметки коммита укажите его контрольную сумму (или её часть) в конце команды:

```
$ git tag -a v1.2 -m 'version 1.2' 9fceb02
```

Можете проверить, что коммит теперь отмечен:

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
```

```
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    updated rakefile
...
```

Обмен метками

По умолчанию, команда `git push` не отправляет метки на удалённые серверы. Необходимо явно отправить (push) метки на общий сервер после того, как вы их создали. Это делается так же, как и выкладывание в совместное пользование удалённых веток — нужно выполнить `git push origin [имя метки]`.

```
$ git push origin v1.5
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [new tag]         v1.5 -> v1.5
```

Если у вас есть много меток, которые хотелось бы отправить все за один раз, можно использовать опцию `--tags` для команды `git push`. В таком случае все ваши метки отправятся на удалённый сервер (если только их уже там нет).

Теперь, если кто-то склонирует (clone) или выполнит `git pull` из вашего репозитория, то он получит вдобавок к остальному и ваши метки.

3. ВЕТВЛЕНИЕ И РАЗРЕШЕНИЕ КОНФЛИКТОВ В GIT

3.1. Ветвление в Git

Почти каждая СКВ имеет в какой-то форме поддержку ветвления. Ветвление означает, что вы отклоняетесь от основной линии разработки и продолжаете работу, не вмешиваясь в основную линию. Во многих СКВ это в некотором роде дорогостоящий процесс, зачастую требующий от вас создания новой копии каталога с исходным кодом, что может занять продолжительное время для больших проектов.

Некоторые говорят, что модель ветвления Git’а это его “killer feature“ и она безусловно выделяет Git в СКВ-сообществе. Что же в ней такого особенного? Способ ветвления в Git’е чрезвычайно легковесен, что делает операции ветвления практически мгновенными и переключение туда-сюда между ветками обычно так же быстрым. В отличие от многих других СКВ, Git поощряет процесс работы, при котором ветвление и слияние осуществляется часто, даже по несколько раз в день. Понимание и владение этой функциональностью даёт вам уникальный мощный инструмент и может буквально изменить то, как вы ведёте разработку.

Что такое ветка?

Чтобы на самом деле разобраться в том, как Git работает с ветками, мы должны сделать шаг назад и рассмотреть, как Git хранит свои данные. Как вы, наверное, помните из первой лекции, Git хранит данные не как последовательность изменений или дельт, а как последовательность снимков состояния (snapshot).

Когда вы создаёте коммит в Git'е, Git записывает в базу объект-коммит, который содержит указатель на снимок состояния, записанный ранее в индекс, метаданные автора и комментария и ноль и более указателей на коммиты, являющиеся прямыми предками этого коммита: ноль предков для первого коммита, один — для обычного коммита и несколько — для коммита, полученного в результате слияния двух или более веток.

Для наглядности давайте предположим, что у вас есть каталог, содержащий три файла, и вы хотите добавить их все в индекс и сделать коммит. При добавлении файлов в индекс для каждого из них вычислится контрольная сумма (SHA-1 хеш, о котором мы упоминали в главе 1), затем эти версии файлов будут сохранены в Git-репозиторий (Git обращается к ним как к двоичным данным), а их контрольные суммы добавятся в индекс:

```
$ git add README test.rb LICENSE
$ git commit -m 'initial commit of my project'
```

Когда вы создаёте коммит, выполняя `git commit`, Git вычисляет контрольную сумму каждого подкаталога (в нашем случае только корневого каталога) и сохраняет эти объекты-деревья в Git-репозиторий. Затем Git создаёт объект для коммита, в котором есть метаданные и указатель на объект-дерево для корня проекта. Таким образом, Git сможет воссоздать текущее состояние, когда будет нужно.

Ваш Git-репозиторий теперь содержит пять объектов:

- по одному блобу для содержимого каждого из трёх файлов,
- одно дерево, в котором перечислено содержимое каталога и определено соответствие имён файлов и блобов,
- один коммит с указателем на тот самый объект-дерево для корня и со всеми метаданными коммита.

Схематично данные в этом Git-репозитории выглядят так, как показано на рисунке.

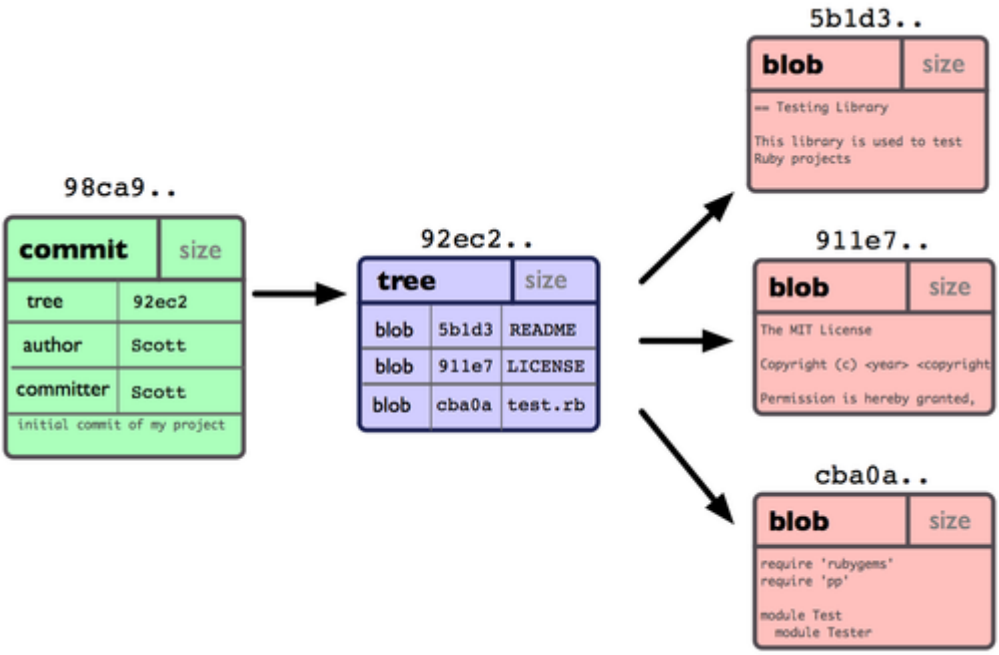

Рисунок 3.1. Данные репозитория с единственным коммитом.

Если вы сделаете некоторые изменения и создадите новый коммит, то следующий коммит сохранит указатель на коммит, который шёл непосредственно перед ним. После следующих двух коммитов история может выглядеть, как на рисунке.

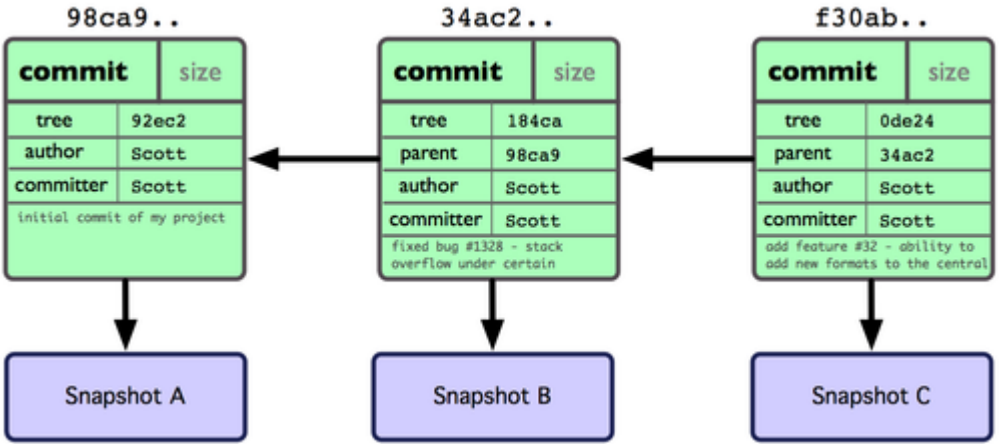

Рисунок 3.2. Данные объектов Git'а для нескольких коммитов.

Ветка в Git'е — это просто легковесный подвижный указатель на один из этих 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.

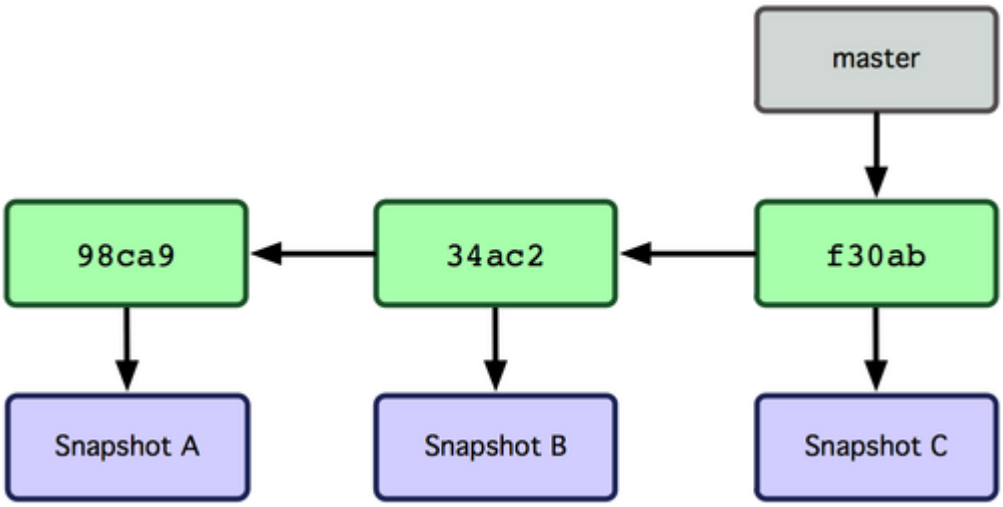

Рисунок 3.3. Ветка указывает на историю коммитов.

Что произойдёт, если вы создадите новую ветку? Итак, этим вы создадите новый указатель, который можно будет перемещать. Скажем, создадим новую ветку под названием `testing`. Это делается командой `git branch`:

```
$ git branch testing
```

Эта команда создаст новый указатель на тот самый коммит, на котором вы сейчас находитесь.

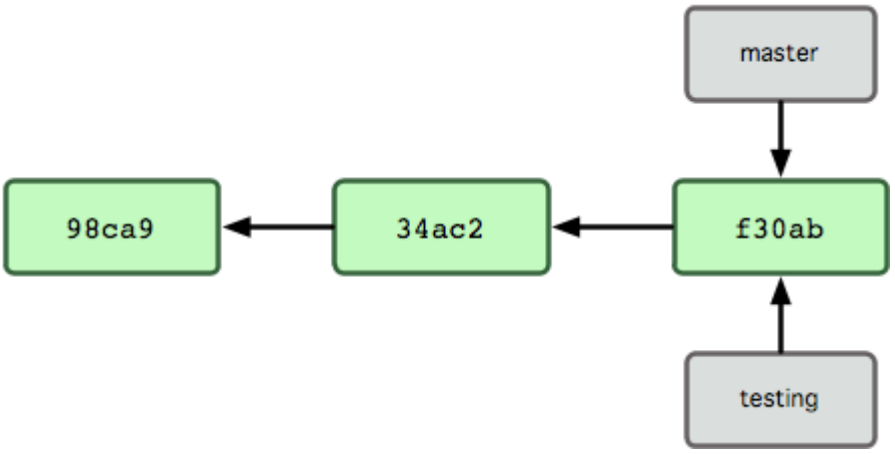

Рисунок 3.4. Несколько веток, указывающих на историю коммитов.

Откуда Git узнает, на какой ветке вы находитесь в данный момент? Он хранит специальный указатель, который называется HEAD (верхушка). Учтите, что это сильно отличается от концепции HEAD в других СКВ, таких как Subversion или CVS, к которым вы, возможно, привыкли. В Git'е это указатель на локальную ветку, на которой вы находитесь. В данный момент вы всё ещё на ветке `master`. Команда `git branch` только создала новую ветку, она не переключила вас на неё.

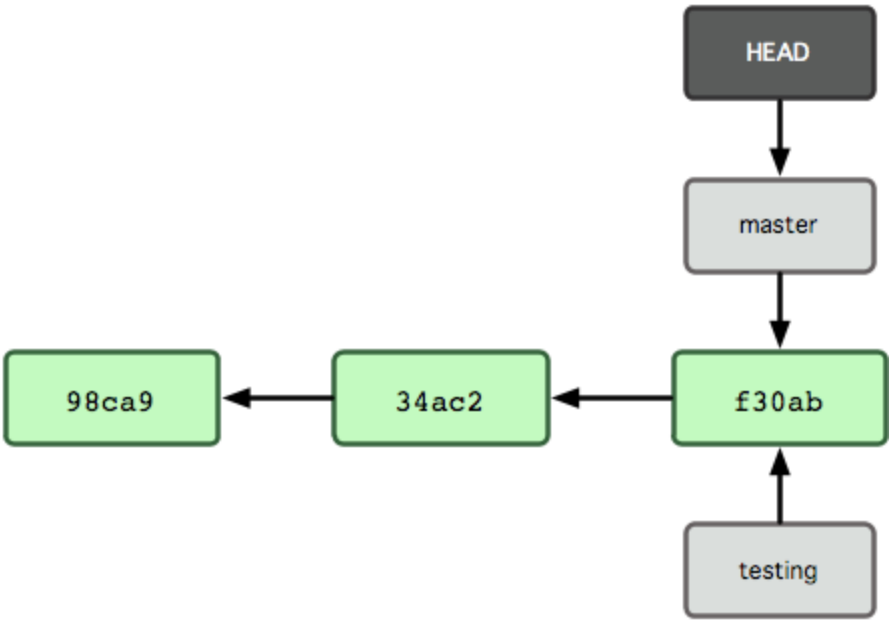

Рисунок 3.5. Файл HEAD указывает на текущую ветку.

Чтобы перейти на существующую ветку, вам надо выполнить команду `git checkout`. Давайте перейдём на новую ветку `testing`:

```
$ git checkout testing
```

Это действие передвинет HEAD так, чтобы тот указывал на ветку `testing`.

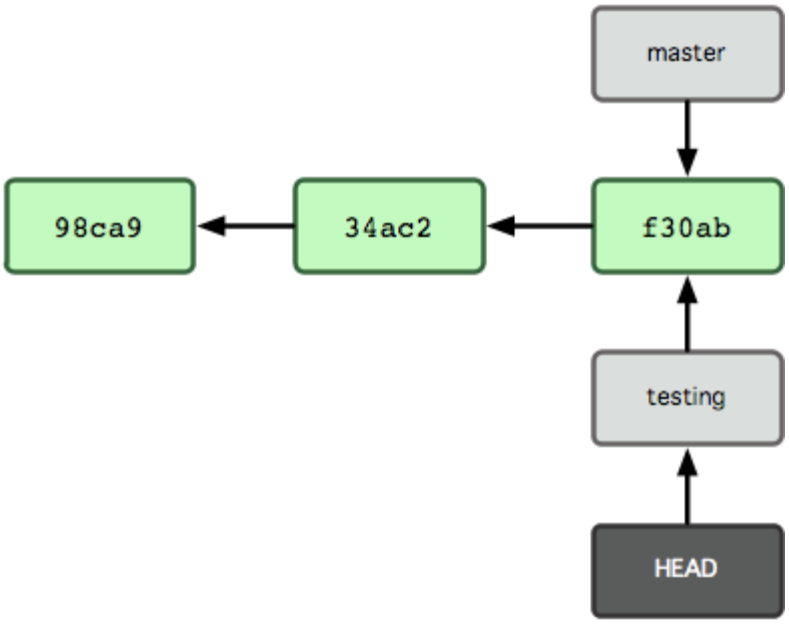

Рисунок 3.6. HEAD указывает на другую ветку после переключения веток.

В чём же важность этого? Давайте сделаем ещё один коммит:

```
$ vim test.rb
$ git commit -a -m 'made a change'
```

На рисунке показан результат.

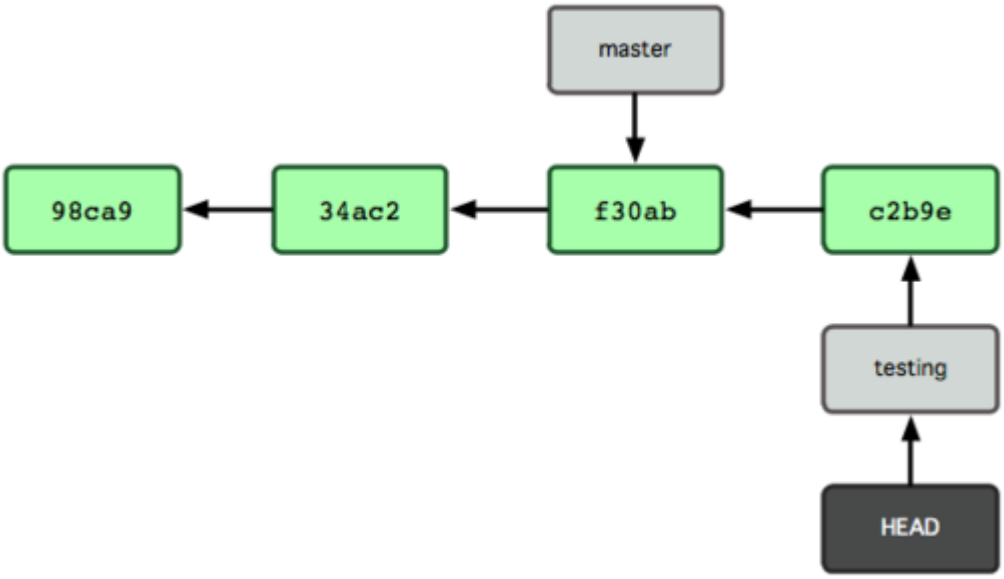

Рисунок 3.7. Ветка, на которую указывает HEAD, движется вперёд с каждым коммитом.

Это интересно, потому что теперь ваша ветка `testing` передвинулась вперёд, но ветка `master` всё ещё указывает на коммит, на котором вы были, когда выполняли `git checkout`, чтобы переключить ветки. Давайте перейдём обратно на ветку `master`:

```
$ git checkout master
```

На рисунке можно увидеть результат.

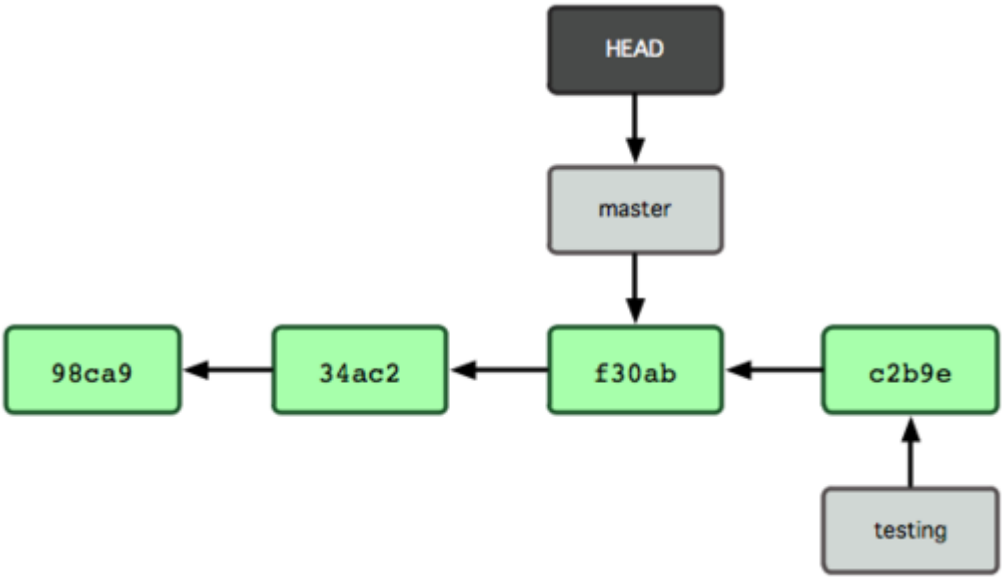

Рисунок 3.8. HEAD перемещается на другую ветку при checkout'е.

Эта команда выполнила два действия. Она передвинула указатель HEAD назад на ветку `master` и вернула файлы в вашем рабочем каталоге назад, в соответствие со снимком состояния, на который указывает `master`. Это также означает, что изменения, которые вы делаете, начиная с этого момента, будут ответвляться от старой версии проекта. Это, по сути, откатывает изменения, которые вы временно делали на ветке `testing`, так что дальше вы можете двигаться в другом направлении.

Давайте снова внесём немного изменений и сделаем коммит:

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

Теперь история вашего проекта разветвилась. Вы создали новую ветку, перешли на неё, поработали на ней немного, переключились обратно на основную ветку и выполнили другую работу. Оба эти изменения изолированы в отдельных ветках: вы можете переключаться туда и обратно между ветками и слить их, когда будете готовы. И всё это было сделано простыми командами `branch` и `checkout`.

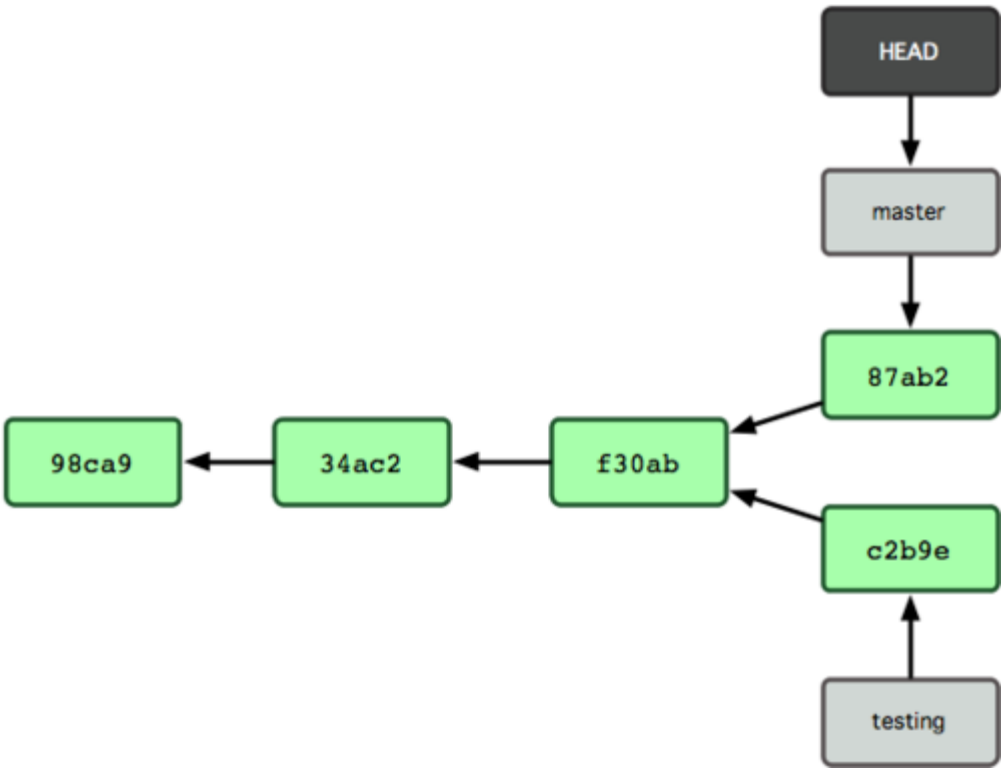

Рисунок 3.9. История с разошедшимися ветками.

Из-за того, что ветка в Git’е на самом деле является простым файлом, который содержит 40 символов контрольной суммы SHA-1 коммита, на который он указывает, создание и удаление веток практически беззатратно. Создание новой ветки настолько же быстрое и простое, как запись 41 байта в файл (40 символов + символ новой строки).

Это разительно отличается от того, как в большинстве СКВ делается ветвление. Там это приводит к копированию всех файлов проекта в другой каталог. Это может занять несколько секунд или даже минут, в зависимости от размера проекта, тогда как в Git’е это всегда происходит моментально. Также благодаря тому, что мы запоминаем предков для каждого коммита, поиск нужной базовой версии для слияния уже автоматически выполнен за нас, и в общем случае слияние делается легко. Эти особенности помогают поощрять разработчиков к частому созданию и использованию веток.

Давайте поймём, почему и вам стоит так делать.

3.2. Основы ветвления и слияния

Давайте рассмотрим ветвление и слияние на простом примере с таким процессом работы, который вы могли бы использовать в настоящей разработке. Мы выполним следующие шаги:

1. Поработаем над веб-сайтом.
2. Создадим ветку для работы над новой задачей.
3. Выполним некоторую работу на этой ветке.

На этом этапе вам поступит звонок о том, что сейчас критична другая проблема, и её надо срочно решить. Мы сделаем следующее:

1. Вернёмся на ветку для версии в производстве.
2. Создадим ветку для исправления ошибки.
3. После тестирования ветки с исправлением сольём её обратно и отправим в продакшн.
4. Вернёмся к своей исходной задаче и продолжим работать над ней.

Основы ветвления

Для начала представим, что вы работаете над своим проектом и уже имеете пару коммитов.

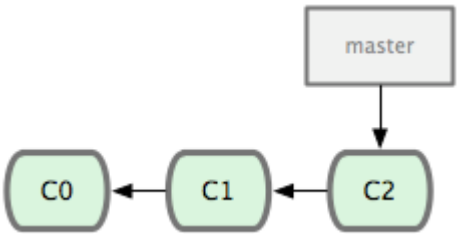

Рисунок 3.10. Короткая и простая история коммитов.

Вы решили, что вы будете работать над проблемой №53 из системы отслеживания ошибок, используемой вашей компанией. Разумеется, Git не привязан к какой-то определенной системе отслеживания ошибок. Так как проблема №53 является обособленной задачей, над которой вы собираетесь работать, мы создадим новую ветку и будем работать на ней. Чтобы создать ветку и сразу же перейти на неё, вы можете выполнить команду `git checkout` с ключом `-b`:

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

Это сокращение для:

```
$ git branch iss53
$ git checkout iss53
```

Результат.

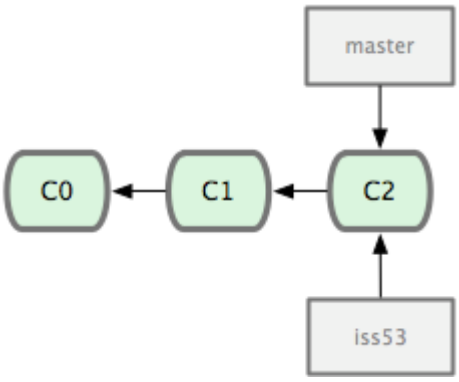

Рисунок 3.11. Создание новой ветки / указателя.

Во время работы над своим веб-сайтом вы делаете несколько коммитов. Эти действия сдвигают ветку `iss53` вперёд потому, что вы на неё перешли (то есть ваш HEAD указывает на неё:

```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```

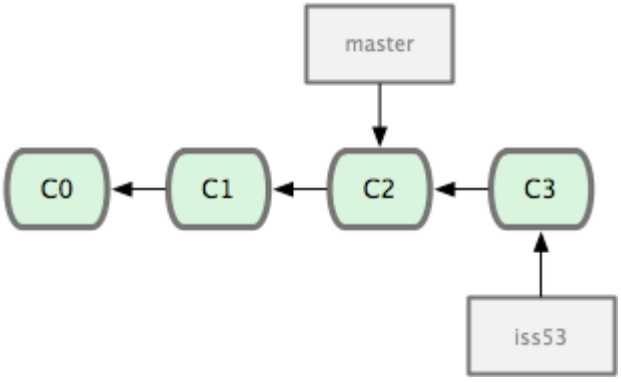

Рисунок 3.12. Ветка iss53 передвинулась вперёд во время работы.

Теперь вы получаете звонок о том, что есть проблема с веб-сайтом, которую необходимо немедленно устранить. С Git'ом вам нет нужды делать исправления для неё поверх тех изменений, которые вы уже сделали в `iss53`, и нет необходимости прикладывать много усилий для отмены этих изменений перед тем, как вы сможете начать работать над решением срочной проблемы. Всё, что вам нужно сделать, это перейти на ветку `master`.

Однако, прежде чем сделать это, учтите, что если в вашем рабочем каталоге или индексе имеются незафиксированные изменения, которые конфликтуют с веткой, на которую вы переходите, Git не позволит переключить ветки. Лучше всего при переключении веток иметь чистое рабочее состояние. Существует несколько способов добиться этого (а именно, прятанье (stash) работы и правка (amend) коммита), которые мы рассмотрим позже. А на данный момент представим, что все изменения были добавлены в коммит, и теперь вы можете переключиться обратно на ветку `master`:

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

Теперь рабочий каталог проекта находится точно в таком же состоянии, что и в момент начала работы над проблемой №53, так что вы можете сконцентрироваться на

исправлении срочной проблемы. Очень важно запомнить: Git возвращает ваш рабочий каталог к снимку состояния того коммита, на который указывает ветка, на которую вы переходите. Он добавляет, удаляет и изменяет файлы автоматически, чтобы гарантировать, что состояние вашей рабочей копии идентично последнему коммиту на ветке.

Итак, вам надо срочно исправить ошибку. Давайте создадим для этого ветку, на которой вы будете работать:

```
$ git checkout -b hotfix
Switched to a new branch "hotfix"
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix]: created 3a0874c: "fixed the broken email address"
 1 files changed, 0 insertions(+), 1 deletions(-)
```

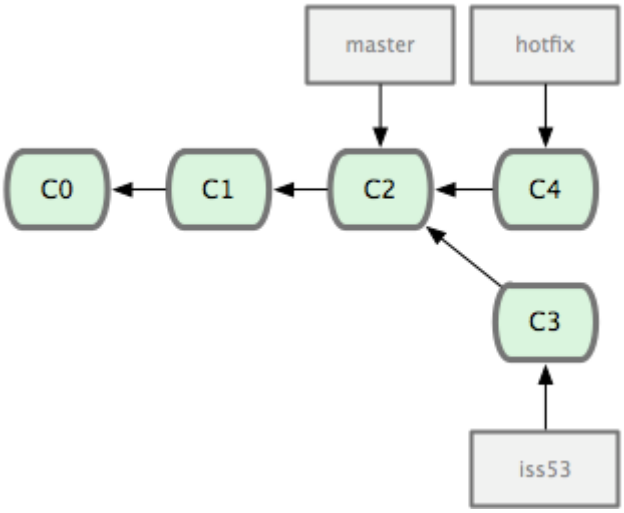

Рисунок 3.13. Ветка для решения срочной проблемы базируется на ветке master.

Вы можете запустить тесты, убедиться, что решение работает, и слить (merge) изменения назад в ветку `master`, чтобы включить их в продукт. Это делается с помощью команды `git merge`:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast forward
 README |    1 -
 1 files changed, 0 insertions(+), 1 deletions(-)
```

Наверное, вы заметили фразу “Fast forward” в этом слиянии. Так как ветка, которую мы слили, указывала на коммит, являющийся прямым родителем коммита, на котором мы сейчас находимся, Git просто сдвинул её указатель вперёд. Иными словами, когда вы пытаетесь слить один коммит с другим таким, которого можно достигнуть, проследовав по истории первого коммита, Git поступает проще, перемещая указатель вперёд, так как нет расходящихся изменений, которые нужно было бы сливать воедино. Это называется “перемотка” (fast forward).

Ваши изменения теперь в снимке состояния коммита, на который указывает ветка `master`, и вы можете включить изменения в продукт.

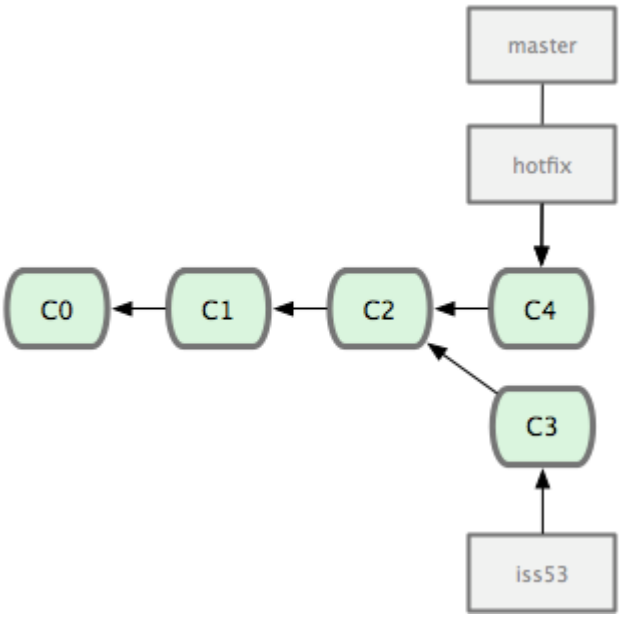

Рисунок 3.14. После слияния ветка master указывает туда же, куда и ветка hotfix.

После того как очень важная проблема решена, вы готовы вернуться обратно к тому, над чем вы работали перед тем, как вас прервали. Однако, сначала удалите ветку `hotfix`, так как она больше не нужна — ветка `master` уже указывает на то же место. Вы можете удалить ветку с помощью опции `-d` к `git branch`:

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

Теперь вы можете вернуться обратно к рабочей ветке для проблемы №53 и продолжить работать над ней:

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53]: created ad82d7a: "finished the new footer [issue 53]"
 1 files changed, 1 insertions(+), 0 deletions(-)
```

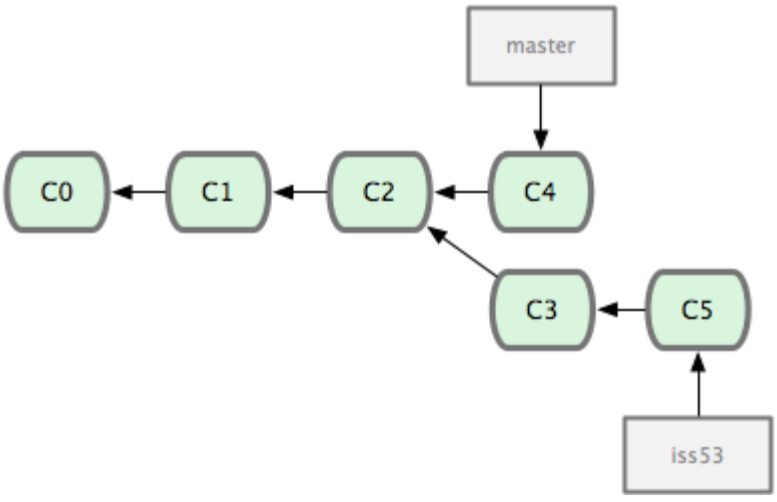

Рисунок 3.15. Ветка iss53 может двигаться вперёд независимо.

Стоит напомнить, что работа, сделанная на ветке `hotfix`, не включена в файлы на ветке `iss53`. Если вам это необходимо, вы можете слить ветку `master` в ветку `iss53`

посредством команды `git merge master`. Или же вы можете подождать с интеграцией изменений до тех пор, пока не решите включить изменения на `iss53` в продуктовую ветку `master`.

Основы слияния

Допустим, вы разобрались с проблемой №53 и готовы объединить эту ветку и свой `master`. Чтобы сделать это, мы сольём ветку `iss53` в ветку `master` точно так же, как мы делали это ранее с веткой `hotfix`. Всё, что вам нужно сделать, — перейти на ту ветку, в которую вы хотите слить свои изменения, и выполнить команду `git merge`:

```
$ git checkout master
$ git merge iss53
Merge made by recursive.
 README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Это слияние немного отличается от слияния, сделанного ранее для ветки `hotfix`. В данном случае история разработки разделилась в некоторой точке. Так как коммит на той ветке, на которой вы находитесь, не является прямым предком для ветки, которую вы сливаете, Git'у придётся проделать кое-какую работу. В этом случае Git делает простое трёхходовое слияние, используя при этом те два снимка состояния репозитория, на которые указывают вершины веток, и общий для этих двух веток снимок-прародитель. На рисунке выделены три снимка состояния, которые Git будет использовать для слияния в данном случае.

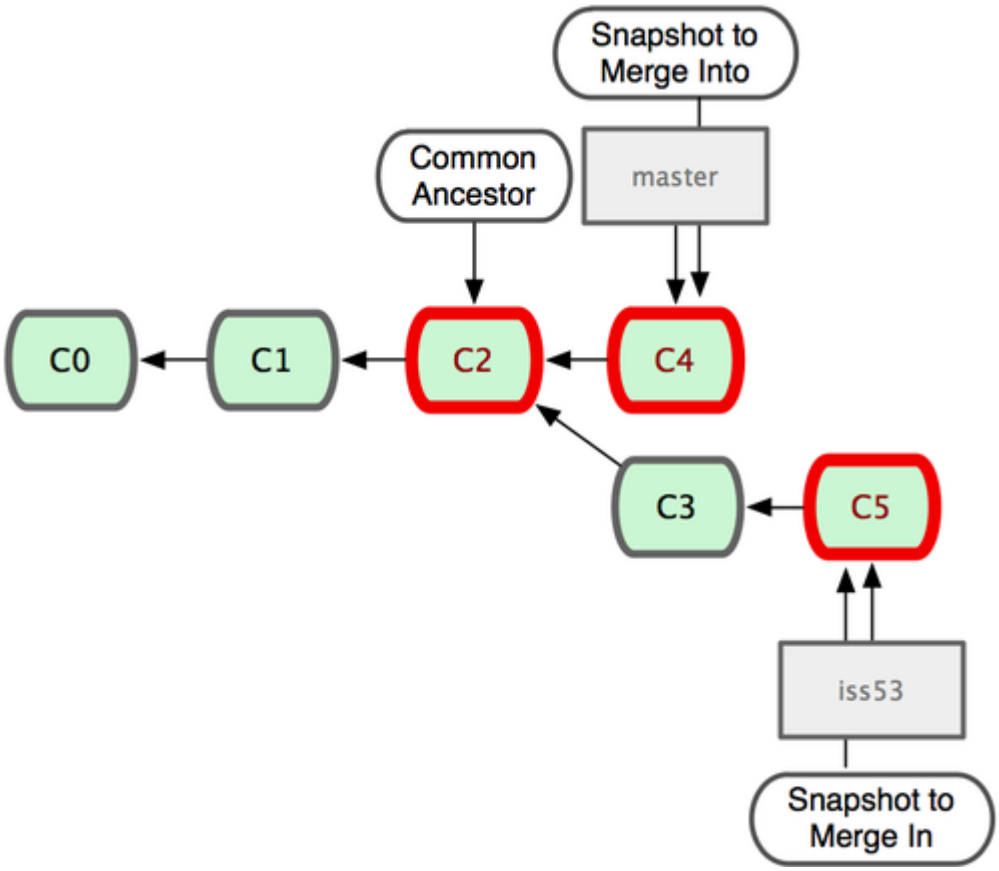

Рисунок 3.16. Git автоматически определяет наилучшего общего предка для слияния веток.

Вместо того чтобы просто передвинуть указатель ветки вперёд, Git создаёт новый снимок состояния, который является результатом трёхходового слияния, и автоматически создаёт новый коммит, который указывает на этот новый снимок состояния. Такой коммит называют коммит-слияние, так как он является особенным из-за того, что имеет больше одного предка.

Стоит отметить, что Git сам определяет наилучшего общего предка для слияния веток; в CVS или Subversion (версии ранее 1.5) этого не происходит. Разработчик должен сам указать основу для слияния. Это делает слияние в Git'е гораздо более простым занятием, чем в других системах.

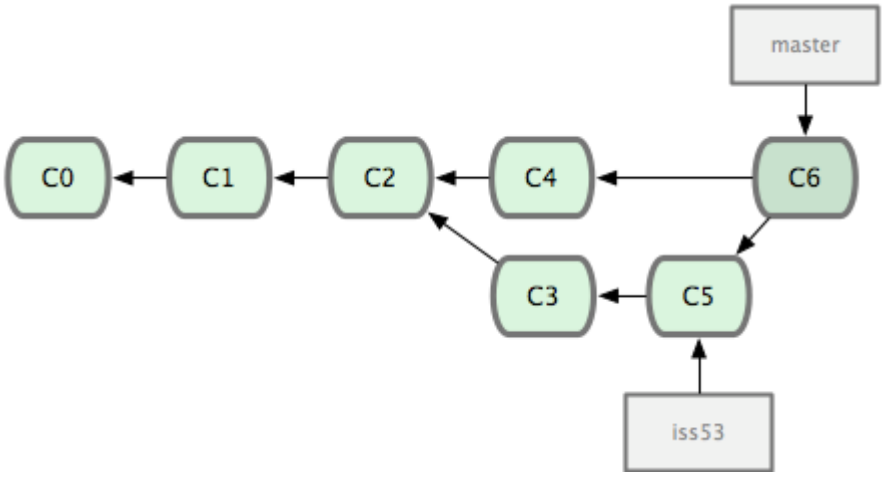

Рисунок 3.17. Git автоматически создаёт новый коммит, содержащий результаты слияния.

Теперь, когда вы осуществили слияние ваших наработок, ветка `iss53` вам больше не нужна. Можете удалить её и затем вручную закрыть карточку (ticket) в своей системе:

```
$ git branch -d iss53
```

3.3. Основы конфликтов при слиянии

Иногда процесс слияния не идёт гладко. Если вы изменили одну и ту же часть файла по-разному в двух ветках, которые собираетесь слить, Git не сможет сделать это чисто. Если ваше решение проблемы №53 изменяет ту же часть файла, что и `hotfix`, вы получите конфликт слияния, и выглядеть он будет примерно так:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git не создал новый коммит для слияния. Он приостановил этот процесс до тех пор, пока вы не разрешите конфликт. Если вы хотите посмотреть, какие файлы не прошли слияние (на любом этапе после возникновения конфликта), выполните команду `git status`:

```
[master*]$ git status
index.html: needs merge
# On branch master
# Changes not staged for commit:
```

```
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       unmerged:   index.html
#
```

Всё, что имеет отношение к конфликту слияния и что не было разрешено, отмечено как unmerged. Git добавляет стандартные маркеры к файлам, которые имеют конфликт, так что вы можете открыть их вручную и разрешить эти конфликты. Ваш файл содержит секцию, которая выглядит примерно так:

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

В верхней части блока (всё что выше `=======`) это версия из HEAD (вашей ветки master, так как именно на неё вы перешли перед выполнением команды merge), всё, что находится в нижней части — версия в `iss53`. Чтобы разрешить конфликт, вы должны либо выбрать одну из этих частей, либо как-то объединить содержимое по своему усмотрению. Например, вы можете разрешить этот конфликт заменой всего блока, показанного выше, следующим блоком:

```
<div id="footer">
  please contact us at email.support@github.com
</div>
```

Это решение содержит понемногу из каждой части, и я полностью удалил строки `<<<<<<<`, `=======` и `>>>>>>>`. После того как вы разобрались с каждой из таких секций в каждом из конфликтных файлов, выполните git add для каждого конфликтного файла. Индексирование будет означать для Git'а, что все конфликты в файле теперь разрешены. Если вы хотите использовать графические инструменты для разрешения конфликтов, можете выполнить команду git mergetool, которая запустит соответствующий графический инструмент и покажет конфликтные ситуации:

```
$ git mergetool
merge tool candidates: kdiff3 tkdiff xxdiff meld gvimdiff opendiff emerge
vimdiff
Merging the files: index.html

Normal merge conflict for 'index.html':
  {local}: modified
  {remote}: modified
Hit return to start merge resolution tool (opendiff):
```

Если вы хотите использовать другой инструмент для слияния, нежели выбираемый по умолчанию (Git выбрал `opendiff` для меня, так как я выполнил команду на Mac'е). Вы можете увидеть все поддерживаемые инструменты, указанные выше после "merge tool candidates". Укажите название предпочтительного для вас инструмента. В рамках следующих лекций мы обсудим, как изменить это значение по умолчанию для вашего окружения.

После того как вы выйдете из инструмента для выполнения слияния, Git спросит вас, было ли оно успешным. Если вы отвечаете, что да — файл индексируется (добавляется в область для коммита), чтобы дать вам понять, что конфликт разрешён.

Можете выполнить `git status` ещё раз, чтобы убедиться, что все конфликты были разрешены:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   index.html
#
```

Если вы довольны тем, что получили, и удостоверились, что всё, имевшее конфликты, было проиндексировано, можете выполнить `git commit` для завершения слияния. По умолчанию сообщение коммита будет выглядеть примерно так:

```
Merge branch 'iss53'

Conflicts:
  index.html
#
# It looks like you may be committing a MERGE.
# If this is not correct, please remove the file
# .git/MERGE_HEAD
# and try again.
#
```

Вы можете дополнить это сообщение информацией о том, как вы разрешили конфликт, если считаете, что это может быть полезно для других в будущем. Например, можете указать почему вы сделали то, что сделали, если это не очевидно, конечно.

3.4. Управление ветками

Теперь, когда вы уже попробовали создавать, объединять и удалять ветки, пора познакомиться с некоторыми инструментами для управления ветками, которые вам пригодятся, когда вы начнёте использовать ветки постоянно.

Команда `git branch` делает несколько больше, чем просто создаёт и удаляет ветки. Если вы выполните её без аргументов, то получите простой список имеющихся у вас веток:

```
$ git branch
  iss53
* master
  testing
```

Обратите внимание на символ `*`, стоящий перед веткой `master`: он указывает на ветку, на которой вы находитесь в настоящий момент. Это означает, что если вы сейчас выполните коммит, ветка `master` переместится вперёд в соответствии с вашими последними изменениями. Чтобы посмотреть последний коммит на каждой из веток, выполните команду `git branch -v`:

```
$ git branch -v
  iss53   93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

Ещё одна полезная возможность для выяснения состояния веток состоит в том, чтобы оставить в этом списке только те ветки, которые вы слили (или не слили) в ветку, на которой сейчас находитесь. Для этих целей в Git’е есть опции `--merged` и `--no-merged`.

Чтобы посмотреть те ветки, которые вы уже слили с текущей, можете выполнить команду `git branch --merged`:

```
$ git branch --merged
  iss53
* master
```

Из-за того что мы ранее слили `iss53`, мы видим её в этом списке. Те ветки из этого списка, перед которыми нет символа `*`, можно смело удалять командой `git branch -d`; вы уже включили наработки из этих веток в другую ветку, так что вы ничего не потеряете.

Чтобы увидеть все ветки, содержащие наработки, которые вы пока ещё не слили в текущую ветку, выполните команду `git branch --no-merged`:

```
$ git branch --no-merged
  testing
```

Вы увидите оставшуюся ветку. Так как она содержит ещё не слитые наработки, попытка удалить её командой `git branch -d` не увенчается успехом:

```
$ git branch -d testing
error: The branch 'testing' is not an ancestor of your current HEAD.
If you are sure you want to delete it, run 'git branch -D testing'.
```

Если вы действительно хотите удалить ветку и потерять наработки, вы можете сделать это при помощи опции `-D`, как указано в подсказке.

3.5. Приёмы работы с ветками

Теперь, когда вы познакомились с основами ветвления и слияния, что вам делать с ветками дальше? В этом разделе мы рассмотрим некоторые стандартные приёмы работы, которые становятся возможными благодаря лёгкости осуществления ветвления. И вы сможете выбрать, включить ли вам какие-то из них в свой цикл разработки.

Долгоживущие ветки

Так как Git использует простое трёхходовое слияние, периодически сливать одну ветку с другой на протяжении большого промежутка времени достаточно просто. Это значит, вы можете иметь несколько веток, которые всегда открыты и которые вы используете для разных стадий вашего цикла разработки; вы можете регулярно сливать их одну в другую.

Многие разработчики Git’а придерживаются такого подхода, при котором ветка `master` содержит исключительно стабильный код — единственный выпускаемый код. Для разработки и тестирования используется параллельная ветка, называемая `develop` или `next`, она может не быть стабильной постоянно, но в стабильные моменты её можно слить в `master`. Эта ветка используется для объединения завершённых задач из тематических веток (временных веток наподобие `iss53`), чтобы удостовериться, что эти изменения проходят все тесты и не вызывают ошибок.

В действительности же, мы говорим об указателях, передвигающихся вверх по линии коммитов, которые вы делаете. Стабильные ветки далеко внизу линии вашей истории коммитов, наиболее свежие ветки находятся ближе к верхушке этой линии.

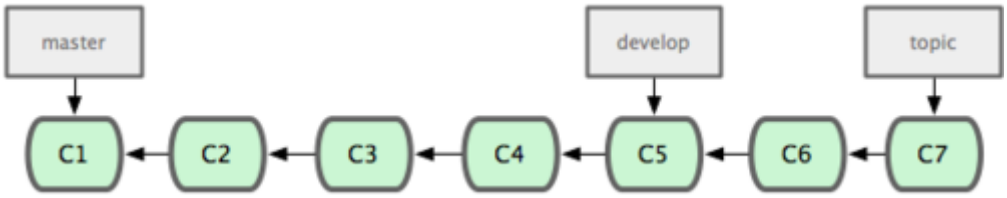

Рисунок 3.18. Более стабильные ветки, как правило, находятся дальше в истории коммитов.

В общем, об этом проще думать как о силосных башнях, где набор коммитов переходит в более стабильную башню только тогда, когда он полностью протестирован.

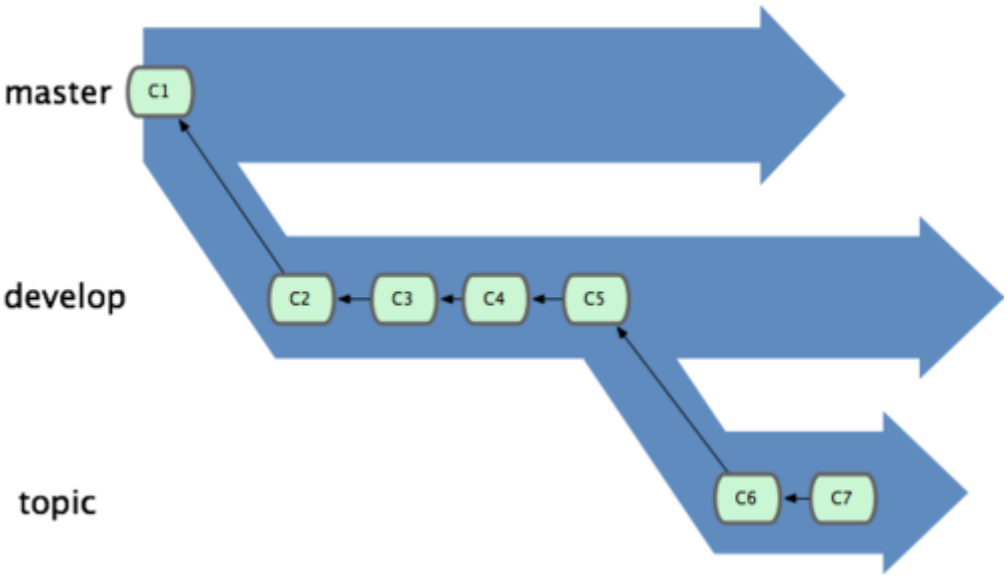

Рисунок 3.19. Может быть полезным думать о ветках как о силосных башнях.

Вы можете применять эту идею для нескольких разных уровней стабильности. Некоторые большие проекты также имеют ветку `proposed` или `pu` (proposed updates — предлагаемые изменения), которые включают в себя ветки, не готовые для перехода в ветку `next` или `master`. Идея такова, что ваши ветки находятся на разных уровнях стабильности; когда они достигают более высокого уровня стабильности, они сливаются с веткой, стоящей на более высоком уровне. Опять-таки, иметь долгоживущие ветки не обязательно, но зачастую это полезно, особенно когда вы имеете дело с очень большими и сложными проектами.

Тематические ветки

Тематические ветки, однако, полезны в проектах любого размера. Тематическая ветка — недолговечная ветка, которую вы создаёте и используете для работы над некоторой отдельной функциональностью или для вспомогательной работы. Это то, чего вы, вероятно, никогда не делали с системами контроля версий раньше, так как создание и слияние веток обычно слишком затратно. Но в Git’е принято создавать ветки, работать над ними, сливать и удалять их по несколько раз в день.

Мы видели подобное в последнем разделе, где вы создавали ветки `iss53` и `hotfix`. Вы сделали всего несколько коммитов на этих ветках и удалили их сразу же после слияния с основной веткой. Такая техника позволяет быстро и полноценно переключать контекст. Ибо когда все изменения разбиты по веткам и определённым темам, намного

проще понять, что было сделано, во время проверки и просмотра кода. Вы можете сохранить там изменения на несколько минут, дней или месяцев, а затем, когда они готовы, слить их в основную ветку, независимо от порядка, в котором их создавали или работали над ними.

Рассмотрим пример, когда при выполнении некоторой работы в ветке `master`, делается новая ветка для решения некой проблемы (`iss91`), выполняется немного работы на ней, от неё ответвляется ещё одна ветка для другого пути решения той же задачи (`iss91v2`), потом осуществляется переход назад на основную ветку (`master`), и некоторое время работа ведётся на ней, затем делается ответвление от неё для выполнения чего-то, в чём вы не уверены, что это хорошая идея (ветка `dumbidea`). Ваша история коммитов будет выглядеть примерно так как на рисунке.

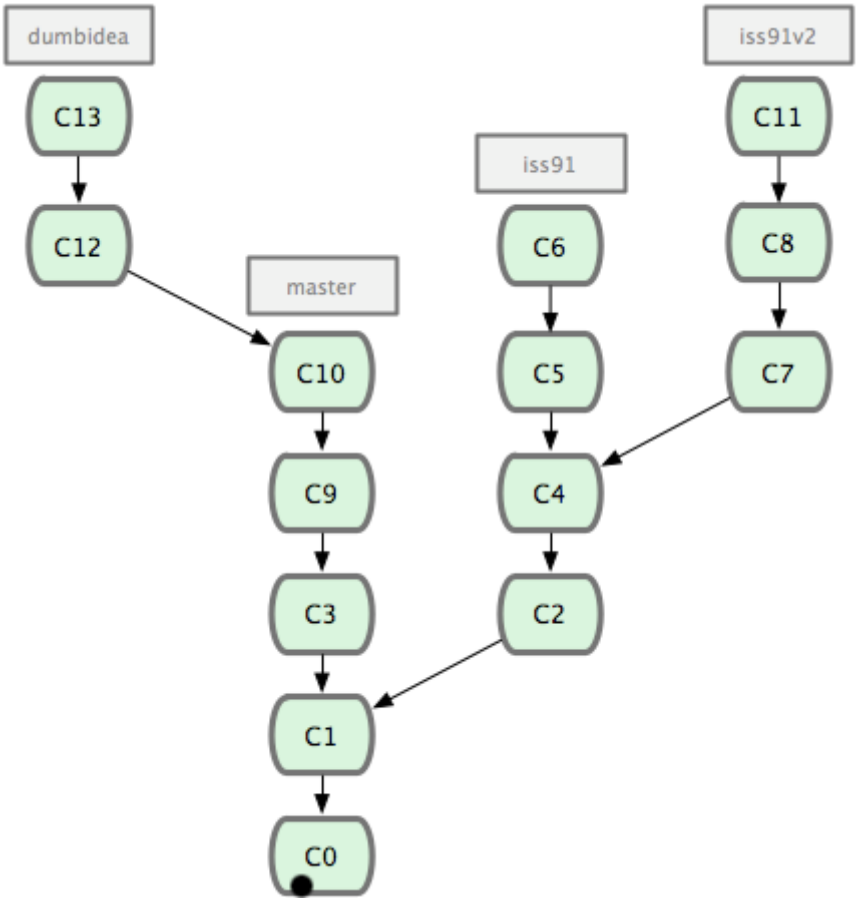

Рисунок 3.20. История коммитов с несколькими тематическими ветками.

Теперь представим, вы решили, что вам больше нравится второе решение для вашей задачи (`iss91v2`); и вы показываете ветку `dumbidea` вашим коллегам и оказывается, что она просто гениальна. Так что вы можете выбросить оригинальную ветку `iss91` (теряя при этом коммиты C5 и C6) и слить две другие. Тогда ваша история будет выглядеть как на рисунке.

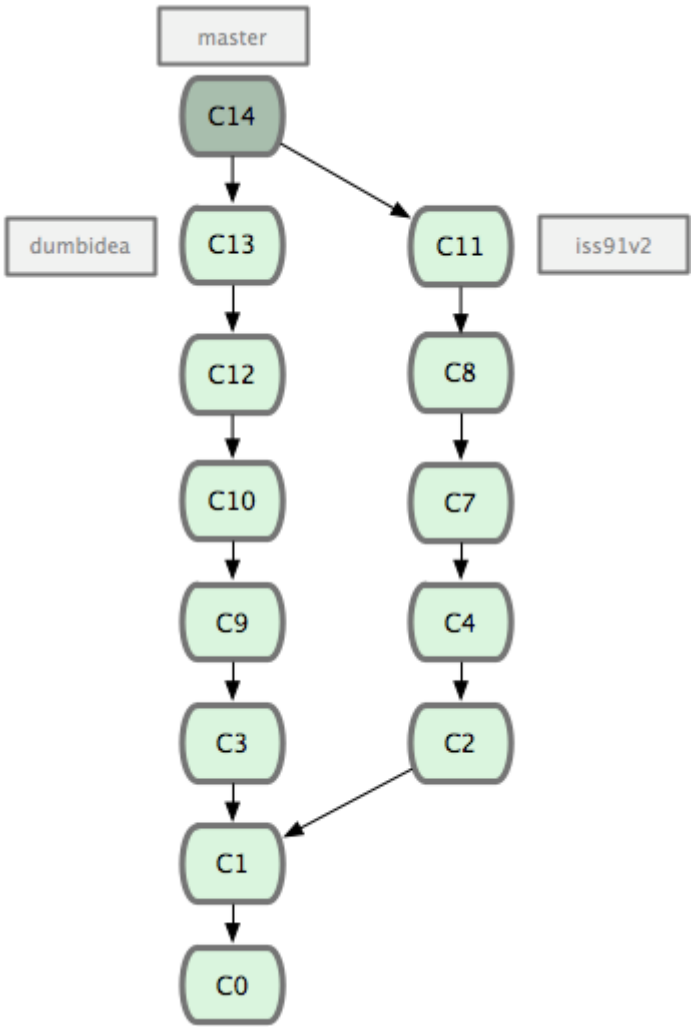

Рисунок 3.21. Ваша история после слияния dumbidea и iss91v2.

Важно запомнить, что когда вы выполняете все эти действия, ветки являются полностью локальными. Когда вы выполняете ветвление и слияние, всё происходит только в вашем репозитории — связь с сервером не осуществляется.

3.6. Удалённые ветки

Удалённые ветки — это ссылки на состояние веток в ваших удалённых репозиториях. Это локальные ветки, которые нельзя перемещать; они двигаются автоматически всякий раз, когда вы осуществляете связь по сети. Удалённые ветки действуют как закладки для напоминания о том, где ветки в удалённых репозиториях находились во время последнего подключения к ним.

Они выглядят как `(имя удал. репоз.)/(ветка)`. Например, если вы хотите посмотреть, как выглядела ветка `master` на сервере `origin` во время последнего соединения с ним, проверьте ветку `origin/master`. Если вы с партнёром работали над одной проблемой, и он выложил ветку `iss53`, у вас может быть своя локальная ветка `iss53`; но та ветка на сервере будет указывать на коммит в `origin/iss53`.

Всё это, возможно, сбивает с толку, поэтому давайте рассмотрим пример. Скажем, у вас в сети есть свой Git-сервер на `git.ourcompany.com`. Если вы с него что-то

склонируете (clone), Git автоматически назовёт его `origin`, заберёт оттуда все данные, создаст указатель на то, на что там указывает ветка `master`, и назовёт его локально `origin/master` (но вы не можете его двигать). Git также сделает вам вашу собственную ветку `master`, которая будет начинаться там же, где и ветка `master` в origin, так что вам будет с чем работать.

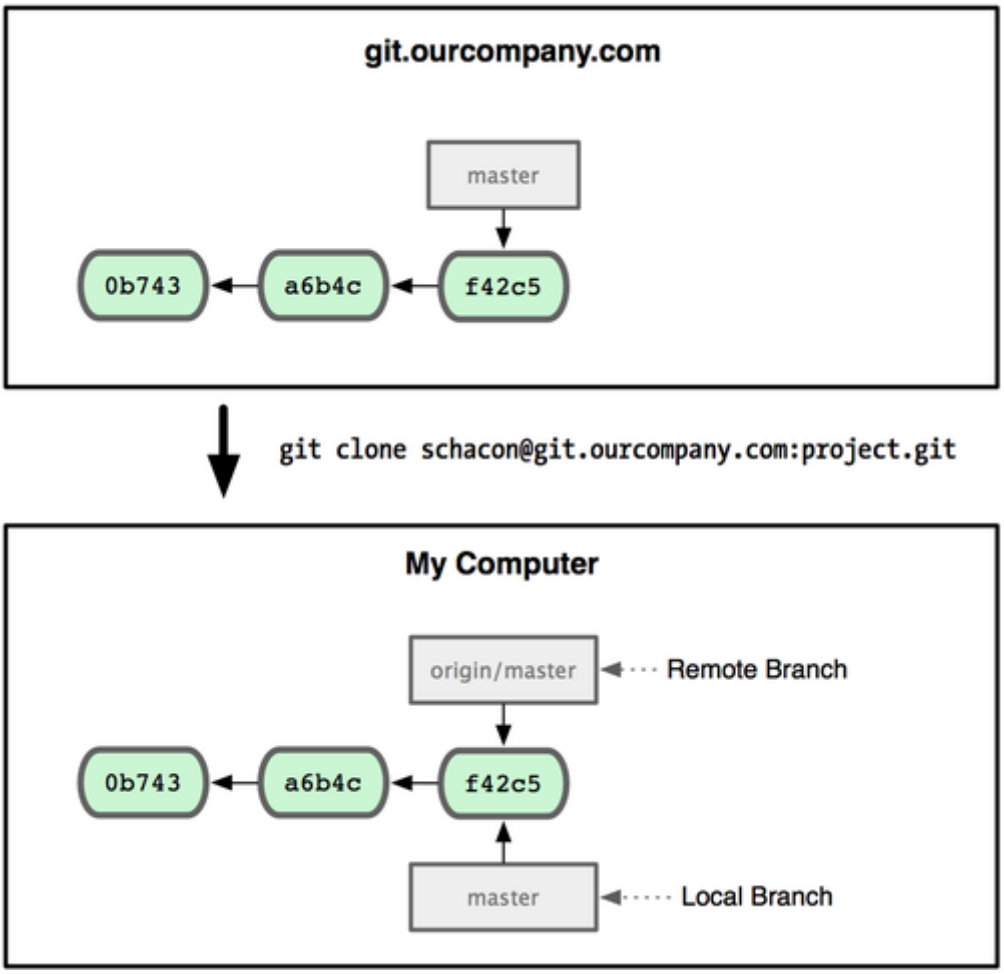

Рисунок 3.22. Клонирование Git-проекта даёт вам собственную ветку master и origin/master, указывающий на ветку master в origin.

Если вы сделаете что-то в своей локальной ветке `master`, а тем временем кто-то ещё отправит (push) изменения на `git.ourcompany.com` и обновит там ветку `master`, то ваши истории продолжатся по-разному. Ещё, до тех пор, пока вы не свяжетесь с сервером origin, ваш указатель `origin/master` не будет сдвигаться.

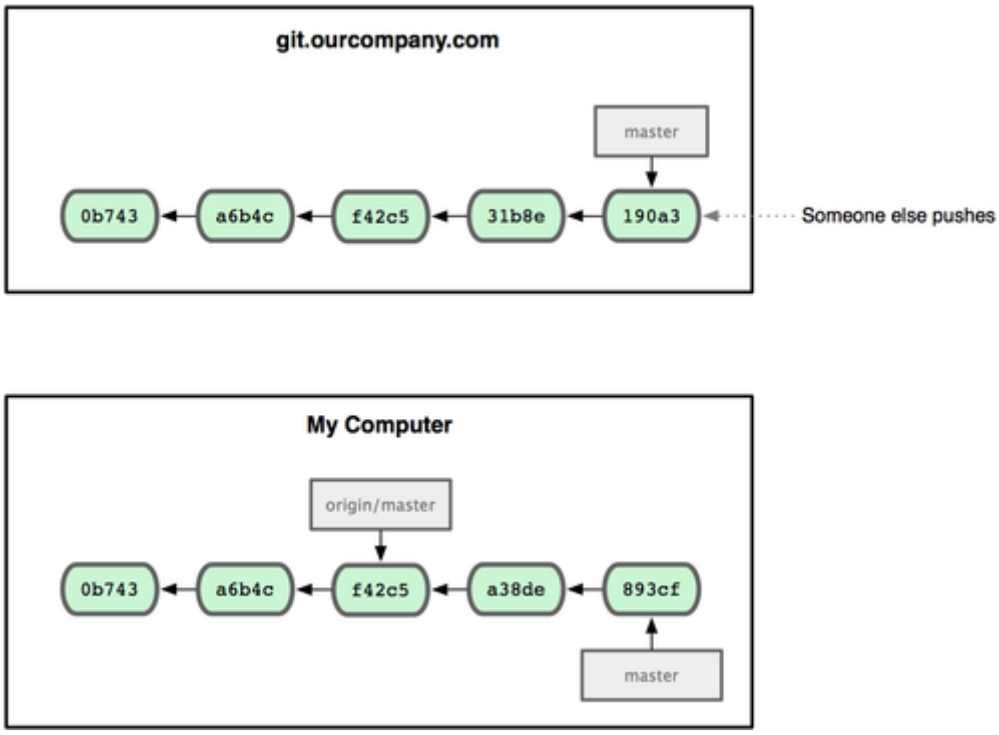

Рисунок 3.23. При выполнении локальной работы и отправке кем-то изменений на удалённый сервер каждая история продолжается по-разному.

Для синхронизации вашей работы выполняется команда `git fetch origin`. Эта команда ищет, какому серверу соответствует origin (в нашем случае это `git.ourcompany.com`); извлекает оттуда все данные, которых у вас ещё нет, и обновляет ваше локальное хранилище данных; сдвигает указатель `origin/master` на новую позицию.

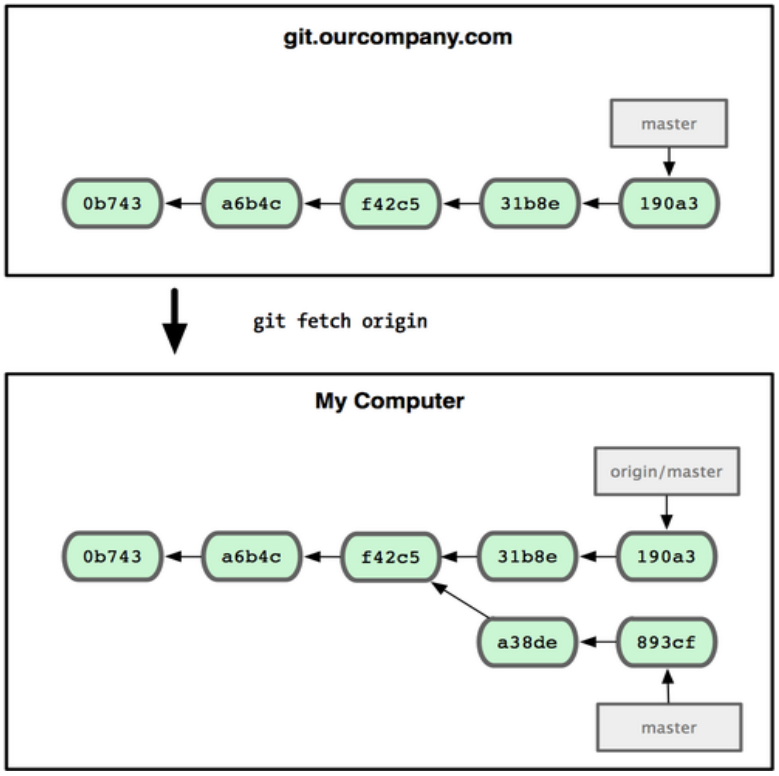

Рисунок 3.24. Команда git fetch обновляет ваши удалённые ссылки.

Чтобы продемонстрировать то, как будут выглядеть удалённые ветки в ситуации с несколькими удалёнными серверами, предположим, что у вас есть ещё один внутренний Git-сервер, который используется для разработки только одной из ваших команд разработчиков. Этот сервер находится на `git.team1.ourcompany.com`. Вы можете добавить его в качестве новой удалённой ссылки на проект, над которым вы сейчас работаете с помощью команды `git remote add` так же, как было описано ранее. Дайте этому удалённому серверу имя `teamone`, которое будет сокращением для полного URL.

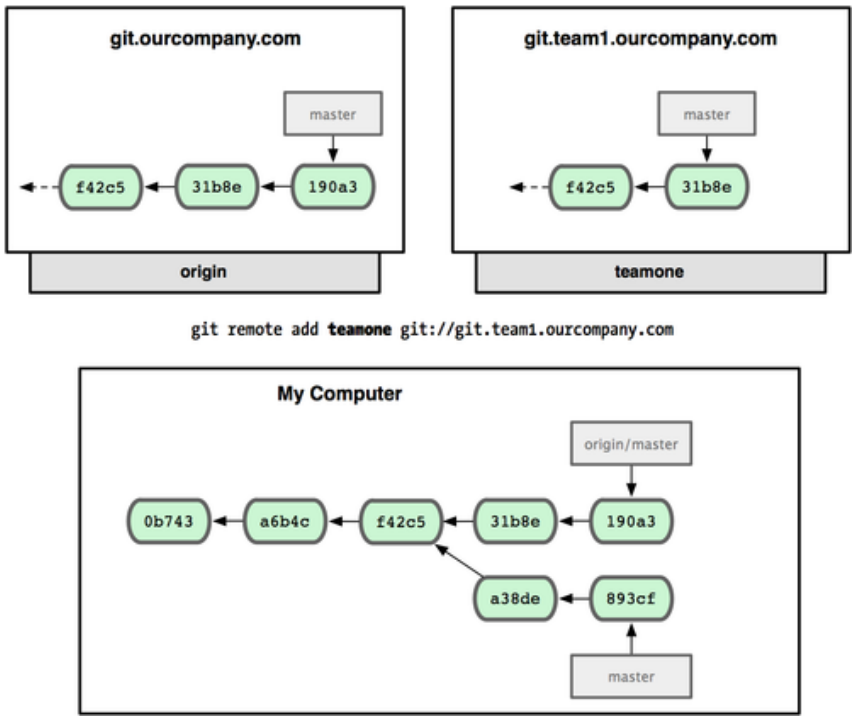

Рисунок 3.25. Добавление дополнительного удалённого сервера.

Теперь можете выполнить `git fetch teamone`, чтобы извлечь всё, что есть на сервере и нет у вас. Так как в данный момент на этом сервере есть только часть данных, которые есть на сервере `origin`, Git не получает никаких данных, но выставляет удалённую ветку с именем `teamone/master`, которая указывает на тот же коммит, что и ветка `master` на сервере `teamone`.

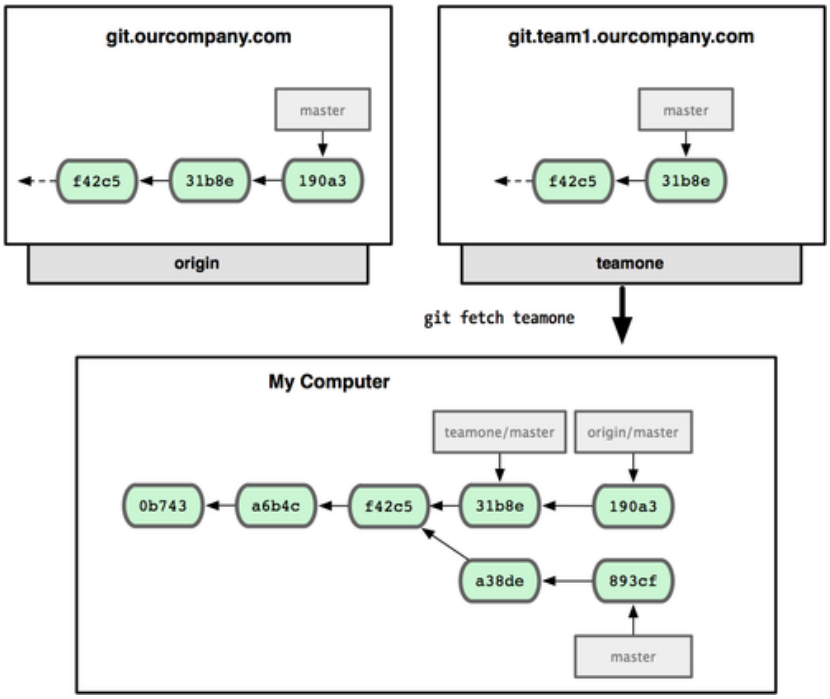

Рисунок 3.26. У вас появилась локальная ссылка на ветку master на teamone-е.

Отправка изменений

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. Таким образом, вы можете использовать свои личные ветки для работы, которую вы не хотите показывать, и отправлять только те тематические ветки, над которыми вы хотите работать с кем-то совместно.

Если у вас есть ветка `serverfix`, над которой вы хотите работать с кем-то ещё, вы можете отправить её точно так же, как вы отправляли вашу первую ветку. Выполните `git push (удал. сервер) (ветка)`:

```
$ git push origin serverfix
Counting objects: 20, done.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 1.74 KiB, done.
Total 15 (delta 5), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new branch]      serverfix -> serverfix
```

Это в некотором роде сокращение. Git автоматически разворачивает имя ветки `serverfix` до `refs/heads/serverfix:refs/heads/serverfix`, что означает “возьми мою локальную ветку serverfix и обнови из неё удалённую ветку serverfix”. Мы подробно обсудим часть с `refs/heads/` в рамках следующих лекций, но обычно её можно опустить. Вы также можете выполнить `git push origin serverfix:serverfix` — произойдёт то же самое — здесь говорится “возьми мой serverfix и сделай его удалённым serverfix”. Можно использовать этот формат для отправки локальной ветки в удалённую ветку с другим именем. Если вы не хотите, чтобы ветка называлась `serverfix` на удалённом сервере, то вместо предыдущей команды выполните `git push origin`

`serverfix:awesomebranch`. Так ваша локальная ветка `serverfix` отправится в ветку `awesomebranch` удалённого проекта.

В следующий раз, когда один из ваших соавторов будет получать обновления с сервера, он получит ссылку на то, на что указывает `serverfix` на сервере, как удалённую ветку `origin/serverfix`:

```
$ git fetch origin
remote: Counting objects: 20, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 15 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
From git@github.com:schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

Важно отметить, что когда при получении данных у вас появляются новые удалённые ветки, вы не получаете автоматически для них локальных редактируемых копий. Другими словами, в нашем случае вы не получите новую ветку `serverfix` — только указатель `origin/serverfix`, который вы не можете менять.

Чтобы слить эти наработки в свою текущую рабочую ветку, выполните `git merge origin/serverfix`. Если вам нужна своя собственная ветка `serverfix`, над которой вы сможете работать, то вы можете создать её на основе удалённой ветки:

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

Это даст вам локальную ветку, на которой можно работать. Она будет начинаться там, где и `origin/serverfix`.

Отслеживание веток

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. Аналогично выполнение `git pull` на одной из таких веток сначала получает все удалённые ссылки, а затем автоматически делает слияние с соответствующей удалённой веткой.

При клонировании репозитория, как правило, автоматически создаётся ветка `master`, которая отслеживает `origin/master`, поэтому `git push` и `git pull` работают для этой ветки “из коробки” и не требуют дополнительных аргументов. Однако, вы можете настроить отслеживание и других веток удалённого репозитория. Простой пример, как это сделать, вы увидели только что — `git checkout -b [ветка] [удал. сервер]/[ветка]`. Если вы используете Git версии 1.6.2 или более позднюю, можете также воспользоваться сокращением `--track`:

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

Чтобы настроить локальную ветку с именем, отличным от имени удалённой ветки, вы можете легко использовать первую версию с другим именем локальной ветки:

```
$ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch refs/remotes/origin/serverfix.
```

```
Switched to a new branch "sf"
```

Теперь ваша локальная ветка sf будет автоматически отправлять (push) и получать (pull) изменения из origin/serverfix.

Удаление веток на удалённом сервере

Скажем, вы и ваши соавторы закончили с нововведением и слили его в ветку `master` на удалённом сервере (или в какую-то другую ветку, где хранится стабильный код). Вы можете удалить ветку на удалённом сервере, используя несколько бестолковый синтаксис `git push [удал. сервер] :[ветка]`. Чтобы удалить ветку `serverfix` на сервере, выполните следующее:

```
$ git push origin :serverfix
To git@github.com:schacon/simplegit.git
 - [deleted]         serverfix
```

Хлоп. Нет больше ветки на вашем сервере. Можно запомнить эту команду вернувшись к синтаксису `git push [удал. сервер] [лок. ветка]:[удал. ветка]`, который мы рассматривали немного раньше. Опуская часть `[лок. ветка]`, вы, по сути, говорите “возьми ничто в моём репозитории и сделай так, чтобы в `[удал. ветка]` было то же самое”.

3.7. Перемещение

В Git’е есть два способа включить изменения из одной ветки в другую: `merge` (слияние) и `rebase` (перемещение). В этом разделе вы узнаете, что такое перемещение, как его осуществлять, почему это удивительный инструмент и в каких случаях вам не следует его использовать.

Основы перемещения

Если мы вернёмся назад к одному из ранних примеров из раздела про слияние, увидим, что мы разделили свою работу на два направления и сделали коммиты на двух разных ветках.

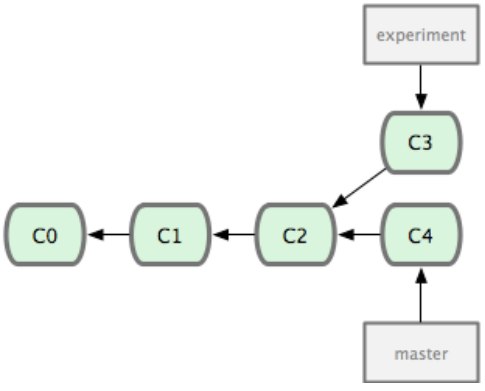

Рисунок 3.27. Впервые разделенная история коммитов.

Наиболее простое решение для объединения веток, как мы уже выяснили, команда `merge`. Эта команда выполняет трёхходовое слияние между двумя последними снимками состояний из веток (C3 и C4) и последним общим предком этих двух веток (C2), создавая новый снимок состояния (и коммит), как показано на рисунке 3-28.

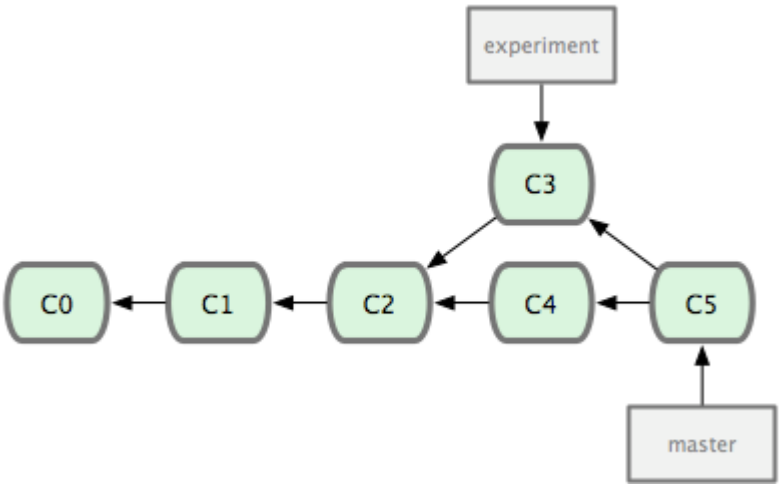

Рисунок 3.28. Слияние ветки для объединения разделившейся истории разработки.

Однако, есть и другой путь: вы можете взять изменения, представленные в C3, и применить их поверх C4. В Git'е это называется *перемещение* (rebasing). При помощи команды `rebase` вы можете взять все изменения, которые попали в коммиты на одной из веток, и повторить их на другой.

Для этого примера надо выполнить следующее:

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

Перемещение работает следующим образом: находится общий предок для двух веток (на которой вы находитесь сейчас и на которую вы выполняете перемещение); для каждого из коммитов в текущей ветке берётся его дельта и сохраняется во временный файл; текущая ветка устанавливается на тот же коммит, что и ветка, на которую выполняется перемещение; и, наконец, одно за другим применяются все изменения. Рисунок иллюстрирует этот процесс.

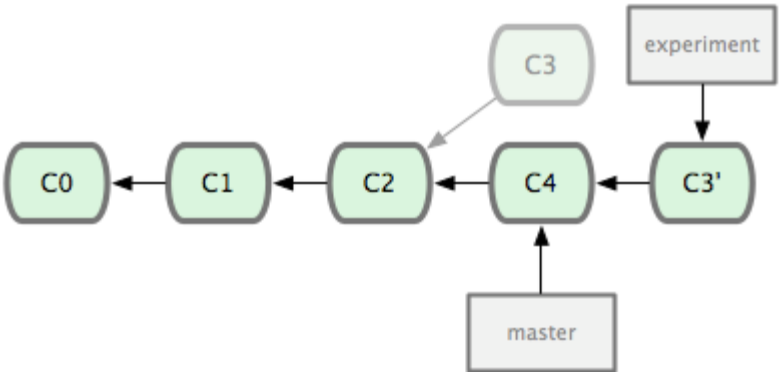

Рисунок 3.29. Перемещение изменений, сделанных в C3, на C4.

На этом этапе можно переключиться на ветку `master` и выполнить слияние-перемотку (fast-forward merge).

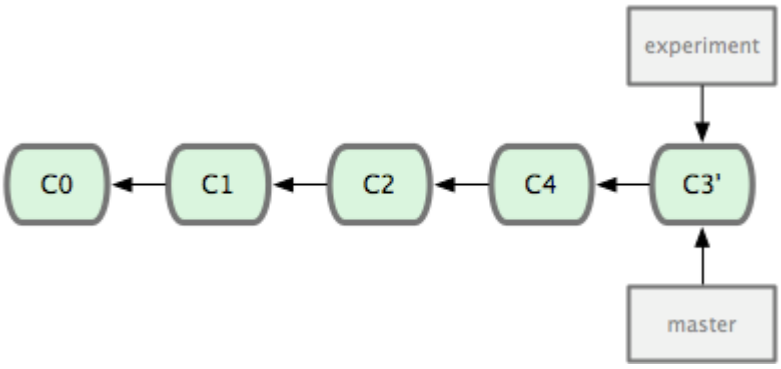

Рисунок 3.30. Перемотка ветки master.

Теперь снимок состояния, на который указывает C3', точно такой же, как тот, на который указывал C5 в примере со слиянием. Нет никакой разницы в конечном результате объединения, но перемещение выполняется для того, чтобы история была более аккуратной. Если вы посмотрите лог для перемещённой ветки, то увидите, что он выглядит как линейная история работы: выходит, что вся работа выполнялась последовательно, когда в действительности она выполнялась параллельно.

Часто вы будете делать это, чтобы удостовериться, что ваши коммиты правильно применяются для удалённых веток — возможно для проекта, владельцем которого вы не являетесь, но в который вы хотите внести свой вклад. В этом случае вы будете выполнять работу в какой-нибудь ветке, а затем, когда будете готовы внести свои изменения в основной проект, выполните перемещение вашей работы на `origin/master`. Таким образом, владельцу проекта не придётся делать никаких действий по объединению — просто перемотка (fast-forward) или чистое применение патчей.

Заметьте, что снимок состояния, на который указывает последний коммит, который у вас получился, является ли этот коммит последним перемещённым коммитом (для случая выполнения перемещения) или итоговым коммитом слияния (для случая выполнения слияния), есть один и тот же снимок — разной будет только история. Перемещение применяет изменения из одной линии разработки в другую в том порядке, в котором они были представлены, тогда как слияние объединяет вместе конечные точки двух веток.

Более интересные перемещения

Можно также сделать так, чтобы при перемещении воспроизведение коммитов начиналось не от той ветки, на которую делается перемещение. Возьмём, например, историю разработки: Вы создали тематическую ветку (`server`), чтобы добавить в проект некоторый функционал для серверной части, и сделали коммит. Затем вы выполнили ответвление, чтобы сделать изменения для клиентской части, и несколько раз выполнили коммиты. Наконец, вы вернулись на ветку `server` и сделали ещё несколько коммитов.

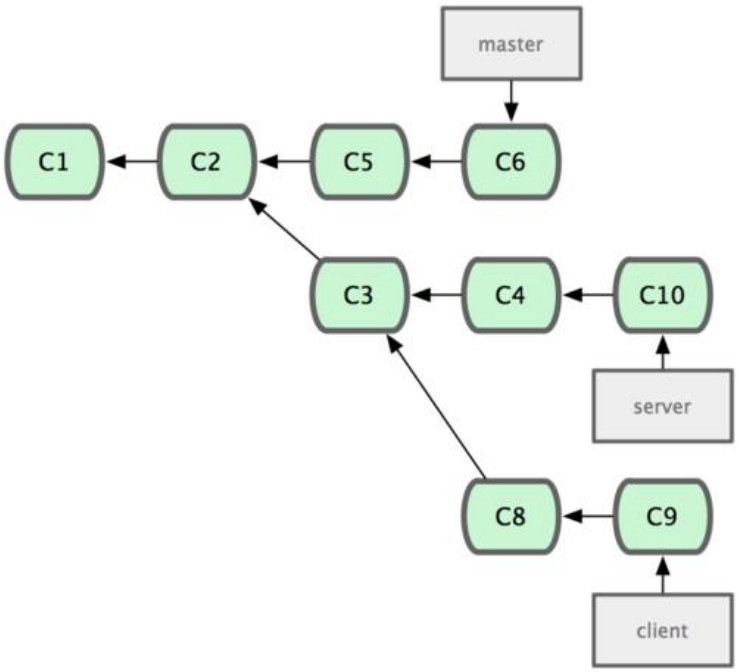

Рисунок 3.31. История разработки с тематической веткой, ответвлённой от другой тематической ветки.

Предположим, вы решили, что хотите внести свои изменения для клиентской части в основную линию разработки для релиза, но при этом хотите оставить в стороне изменения для серверной части, пока они не будут полностью протестированы. Вы можете взять изменения из ветки `client`, которых нет в `server` (C8 и C9), и применить их на ветке `master` при помощи опции `--onto` команды `git rebase`:

```
$ git rebase --onto master server client
```

По сути, это указание “переключиться на ветку `client`, взять изменения от общего предка веток `client` и `server` и повторить их на `master`”. Это немного сложно; но результат, показанный на рисунке 3-32, довольно классный.

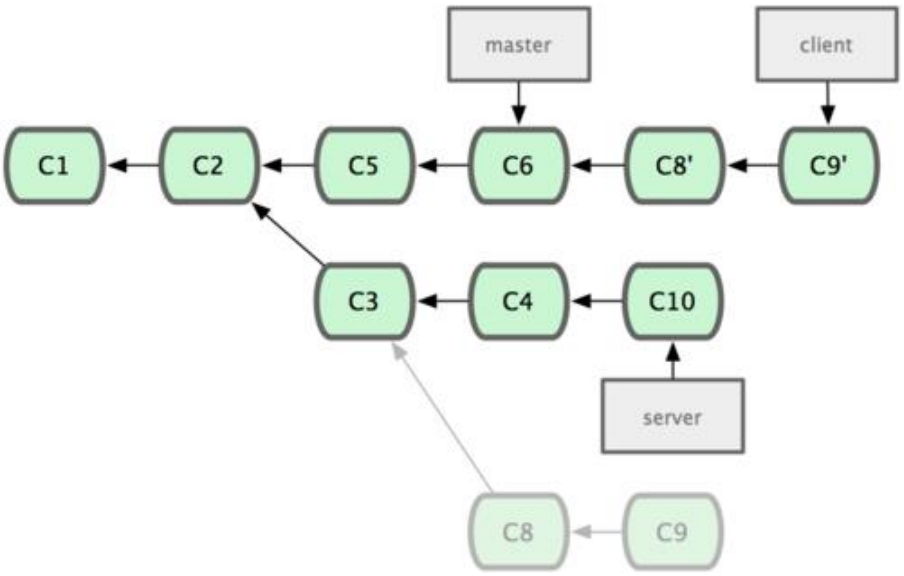

Рисунок 3.32. Перемещение тематической ветки, ответвлённой от другой тематической ветки.

Теперь вы можете выполнить перемотку (fast-forward) для ветки `master`:

```
$ git checkout master
$ git merge client
```

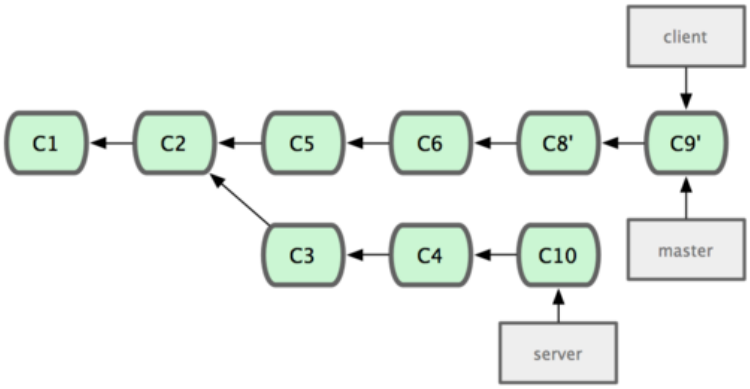

Рисунок 3.33. Перемотка ветки master для добавления изменений из ветки client.

Представим, что вы решили включить работу и из ветки `server` тоже. Вы можете выполнить перемещение ветки `server` на ветку `master` без предварительного переключения на эту ветку при помощи команды `git rebase [осн. ветка] [тем. ветка]` — которая устанавливает тематическую ветку (в данном случае `server`) как текущую и применяет её изменения на основной ветке (`master`):

```
$ git rebase master server
```

Эта команда применит изменения из вашей работы над веткой `server` на вершину ветки `master`, как показано на рисунке.

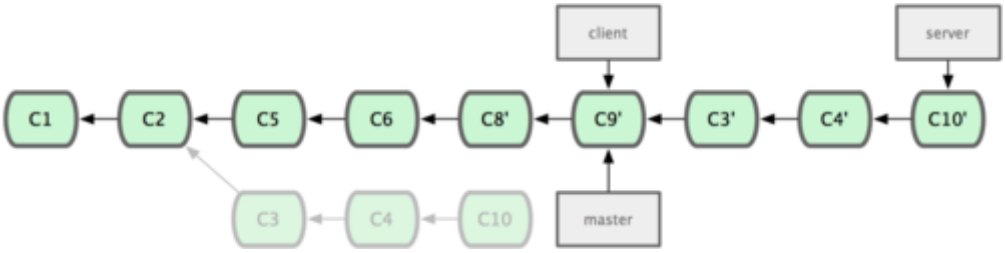

Рисунок 3.34. Перемещение ветки server на вершину ветки master.

Затем вы можете выполнить перемотку основной ветки (`master`):

```
$ git checkout master
$ git merge server
```

Вы можете удалить ветки `client` и `server`, так как вся работа из них включена в основную линию разработки и они вам больше не нужны. При этом полная история вашего рабочего процесса выглядит как на рисунке:

```
$ git branch -d client
$ git branch -d server
```

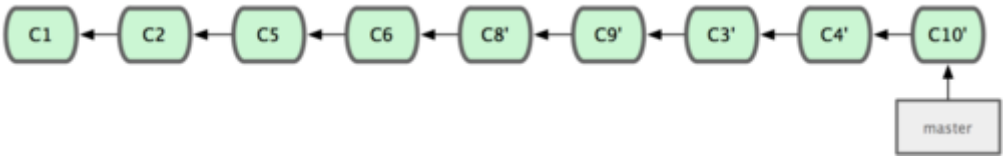

Рисунок 3.35. Финальная история коммитов.

Возможные риски перемещения

Всё бы хорошо, но кое-что омрачает всю прелесть использования перемещения. Это выражается одной строчкой:

Не перемещайте коммиты, которые вы уже отправили в публичный репозиторий.

Если вы будете следовать этому указанию, всё будет хорошо. Если нет — люди возненавидят вас, вас будут презирать ваши друзья и семья.

Когда вы что-то перемещаете, вы отменяете существующие коммиты и создаёте новые, которые похожи на старые, но являются другими. Если вы выкладываете (push) свои коммиты куда-нибудь, и другие забирают (pull) их себе и в дальнейшем основывают на них свою работу, а затем вы переделываете эти коммиты командой `git rebase` и выкладываете их снова, ваши коллеги будут вынуждены заново выполнять слияние для своих наработок. В итоге вы получите путаницу, когда в очередной раз попытаетесь включить их работу в свою.

Давайте рассмотрим пример того, как перемещение публично доступных наработок может вызвать проблемы. Представьте себе, что вы склонировали себе репозиторий с центрального сервера и поработали в нём. И ваша история коммитов выглядит как на рисунке.

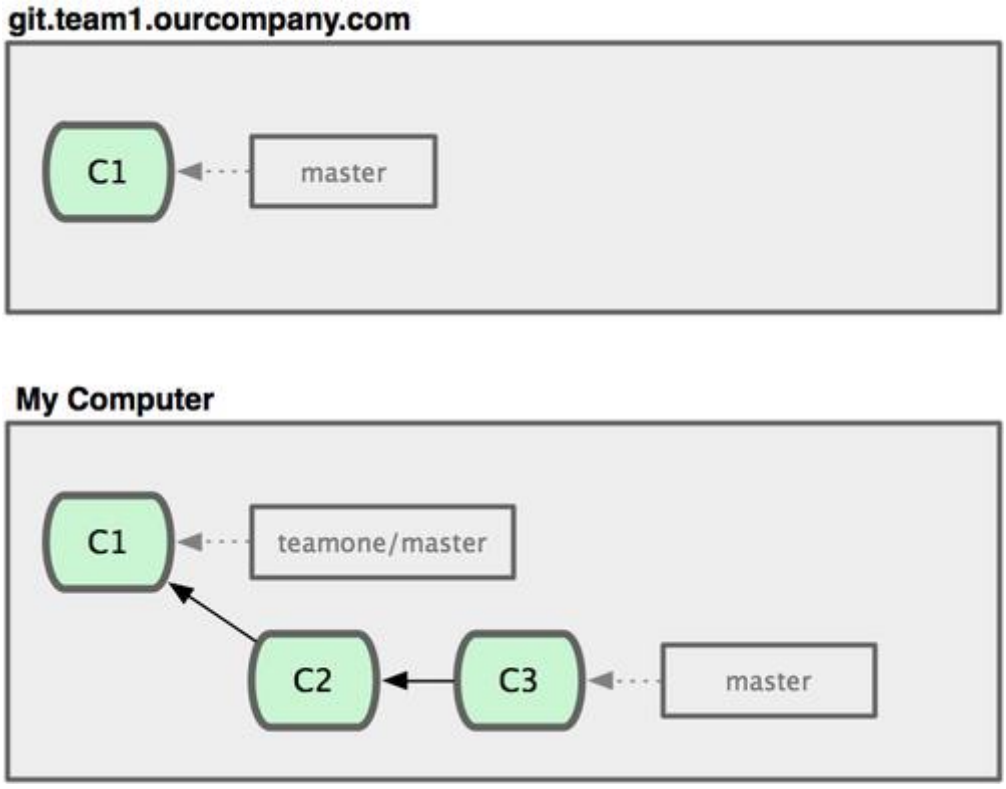

Рисунок 3.36. Клонирование репозитория и выполнение в нём какой-то работы.

Теперь кто-то ещё выполняет работу, причём работа включает в себя и слияние, и отправляет свои изменения на центральный сервер. Вы извлекаете их и сливаете новую удалённую ветку со своей работой. Тогда ваша история выглядит как на рисунке.

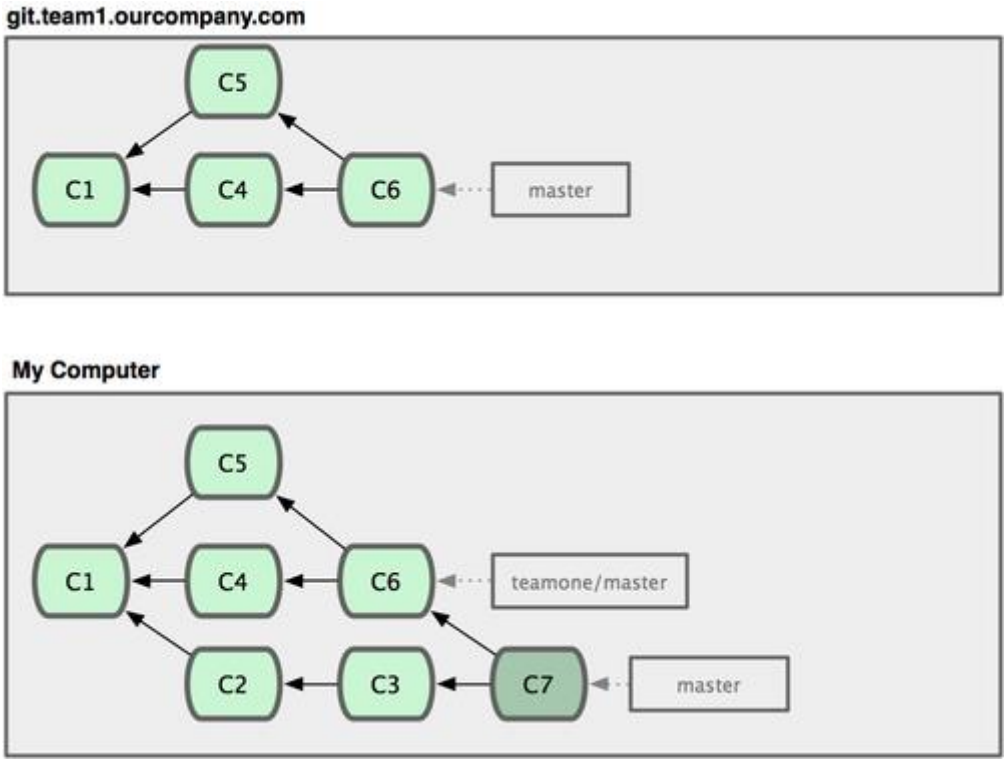

Рисунок 3.37. Извлечение коммитов и слияние их со своей работой.

Далее, человек, выложивший коммит, содержащий слияние, решает вернуться и вместо слияния (merge) переместить (rebase) свою работу; он выполняет `git push --force`, чтобы переписать историю на сервере. Затем вы извлекаете изменения с этого сервера, включая и новые коммиты.

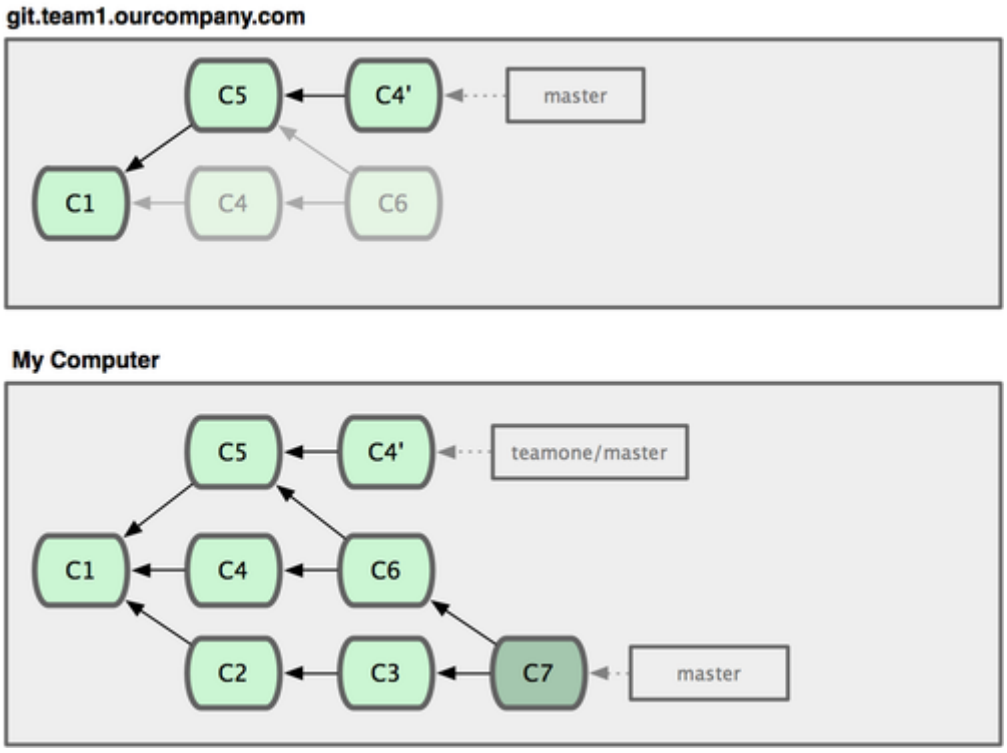

Рисунок 3.38. Кто-то выложил перемещённые коммиты, отменяя коммиты, на которых вы основывали свою работу.

На этом этапе вы вынуждены объединить эту работу со своей снова, даже если вы уже сделали это ранее. Перемещение изменяет у этих коммитов SHA-1 хеши, так что для Git'а они выглядят как новые коммиты, тогда как на самом деле вы уже располагаете наработками из C4 в своей истории.

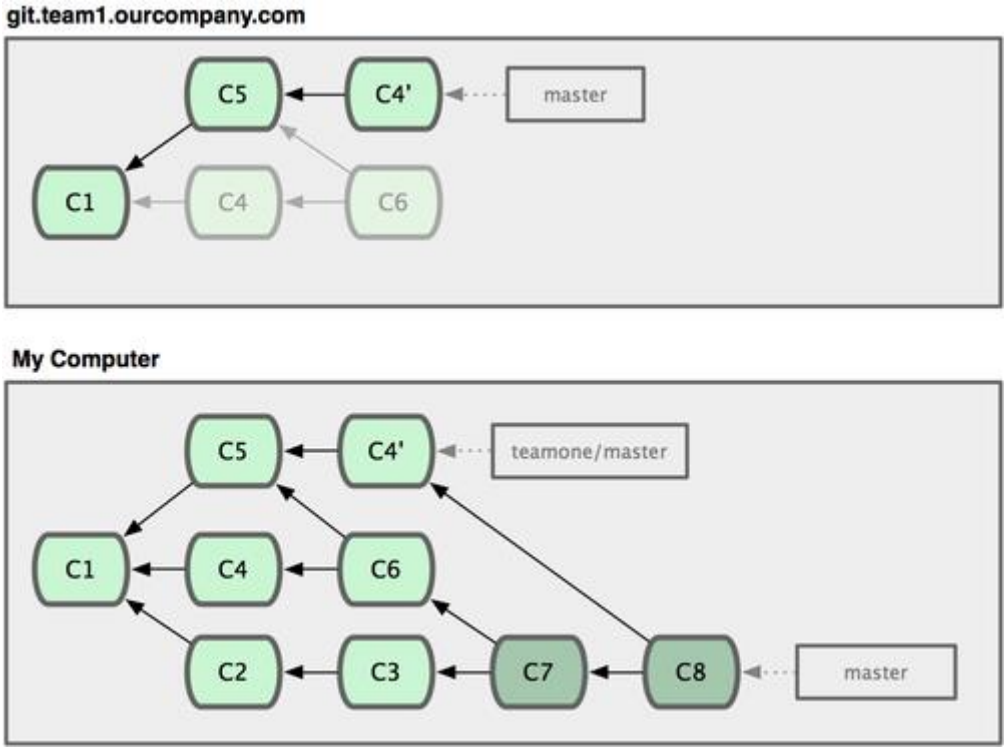

Рисунок 3.39. Вы снова выполняете слияние для той же самой работы в новый коммит слияния.

Вы вынуждены объединить эту работу со своей на каком-либо этапе, чтобы иметь возможность продолжать работать с другими разработчиками в будущем. После того, как вы сделаете это, ваша история коммитов будет содержать оба коммита — C4 и C4', которые имеют разные SHA-1 хеши, но представляют собой одинаковые изменения и имеют одинаковые сообщения. Если вы выполните команду `git log`, когда ваша история выглядит таким образом, вы увидите два коммита, которые имеют одинакового автора и одни и те же сообщения. Это сбивает с толку. Более того, если вы отправите такую историю обратно на сервер, вы добавите все эти перемещенные коммиты в репозиторий центрального сервера, что может ещё больше запутать людей.

Если вы рассматриваете перемещение как возможность наведения порядка и работы с коммитами до того, как выложили их, и если вы перемещаете только коммиты, которые никогда не находились в публичном доступе — всё нормально. Если вы перемещаете коммиты, которые уже были представлены для общего доступа, и люди, возможно, основывали свою работу на этих коммитах, тогда вы можете получить наказание за разные неприятные проблемы.