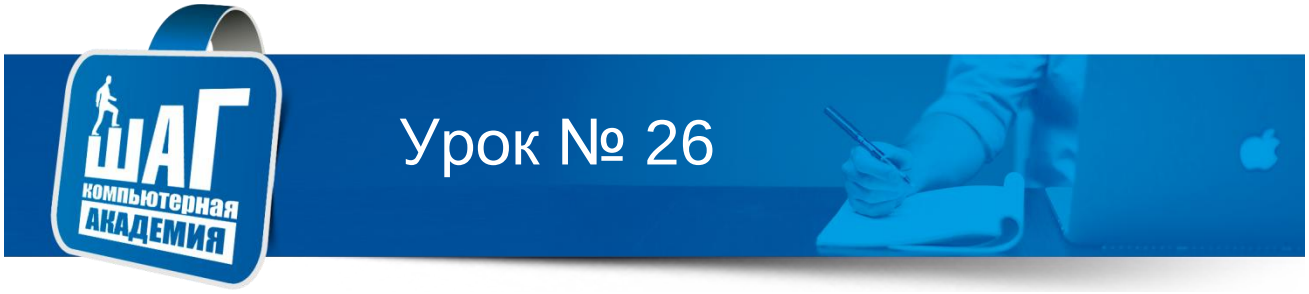

Курс: «Мануальное тестирование ПО»

Тема: Базовые принципы работы с GIT.

План

1. Краткая история Git.
2. Основы Git.
3. Три состояния.
4. Установка Git.
5. GitHub.
6. Первоначальная настройка Git.
7. Основы ветвления и слияния.

1. Краткая история Git

С 2002 года для разработки ядра Linux большинством программистов стала использоваться система контроля версий BitKeeper. Довольно долгое время с ней не возникало проблем, но в 2005 году отношения между сообществом разработчиков ядра Linux и компанией, разрабатывавшей BitKeeper, испортились, и право бесплатного пользования продуктом было отменено.

Это подтолкнуло разработчиков Linux (и в частности Линуса Торвальдса, создателя Linux) разработать собственную систему, основываясь на опыте, полученном за время использования BitKeeper. Основные требования к новой системе были следующими:

- Скорость
- Простота дизайна
- Поддержка нелинейной разработки (тысячи параллельных веток)
- Полная распределённость
- Возможность эффективной работы с такими большими проектами, как ядро Linux (как по скорости, так и по размеру данных)

С момента рождения в 2005 году Git развивался и эволюционировал, становясь проще и удобнее в использовании, сохраняя при этом свои первоначальные

качества. Он быстр, очень эффективен для больших проектов, а также обладает превосходной системой ветвления для нелинейной разработки.

2. Основы Git.

Так что же такое Git в двух словах? Эту часть важно усвоить, поскольку если Вы поймёте, что такое Git, и каковы принципы его работы, вам будет гораздо проще пользоваться им эффективно. Изучая Git, постарайтесь освободиться от всего, что вы знали о других СКВ, таких как Subversion или Perforce. В Git совсем не такие понятия об информации и работе с ней как в других системах, хотя пользовательский интерфейс очень похож. Знание этих различий защитит вас от путаницы при использовании Git.

Слепки вместо патчей

Главное отличие Git от любых других систем контроля версий (например, Subversion и ей подобных) — это то, как Git смотрит на свои данные. 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 1).

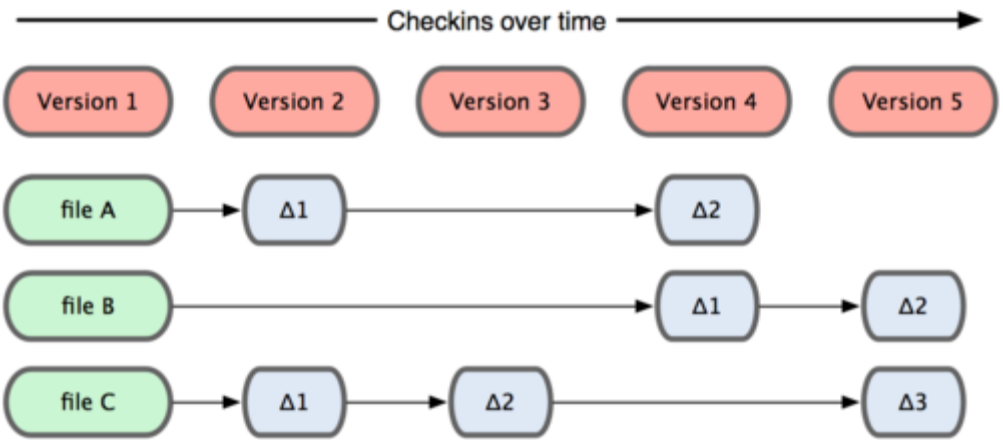

Рис. 1 Стандартный подход хранения изменений.

Другие системы хранят данные как изменения к базовой версии для каждого файла, но Git не хранит свои данные в таком виде. 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(Рисунок 2).

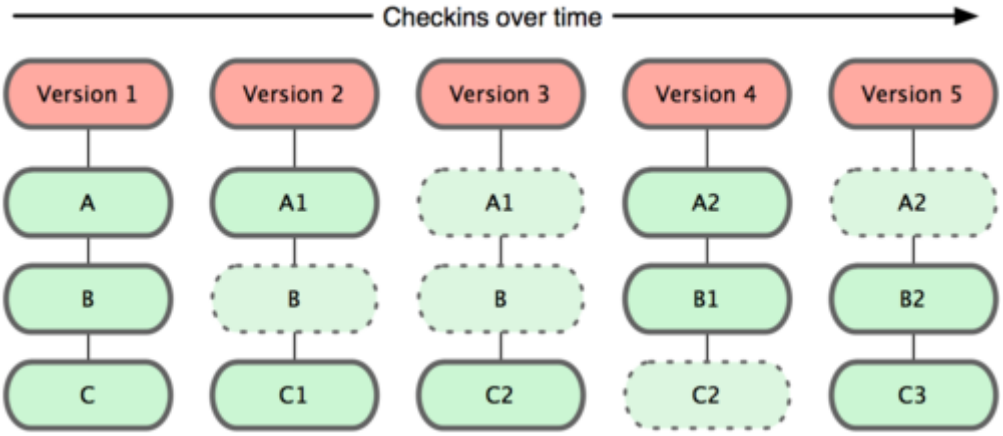

Рис. 2 Подход Git к хранению данных.

То, как Git подходит к хранению данных - это важное отличие Git от 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 очень мощными инструментами, работающими поверх неё, чем на просто систему контроля версий. Далее мы рассмотрим, какие преимущества даёт такое понимание данных.

Почти все операции — локальные

Для совершения большинства операций в Git необходимы только локальные файлы и ресурсы, т.е. обычно информация с других компьютеров в сети не нужна. На прошлом занятии Вы пользовались централизованной системой, где практически на каждую операцию накладывается сетевая задержка, и теперь Вы сполна можете оценить эту особенность Git. Поскольку вся история проекта хранится локально у вас на диске, большинство операций кажутся практически мгновенными.

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

Кроме того, работа локально означает, что мало чего нельзя сделать без доступа к Сети. 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 увидите, насколько это удобно.

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 и является важной составляющей его философии. Если информация потеряется при передаче или повредится на диске, Git всегда это выявит.

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

24b9da6552252987aa493b52f8696cd6d3b00373
Работая с Git, 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, только добавляют данные в базу. Очень сложно заставить систему удалить данные или сделать что-то неотменяемое. Можно, как и в любой другой VCS, 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 - это удобно потому, что можно экспериментировать, не боясь что-то серьёзно поломать.

3.Три состояния

Теперь внимание. Это самое важное, что нужно помнить про Git, если Вы хотите, чтобы дальше изучение шло гладко. В Git файлы могут находиться в одном из трёх состояний: зафиксированном, изменённом и подготовленном. "Зафиксированный" значит, что файл уже сохранён в вашей локальной базе. К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы. Подготовленные файлы — это изменённые файлы, отмеченные для включения в следующий коммит.

Таким образом, в проектах, использующих Git, есть три части(Рисунок 3):

- каталог Git (Git directory),
- рабочий каталог (working directory),
- область подготовленных файлов (staging area).

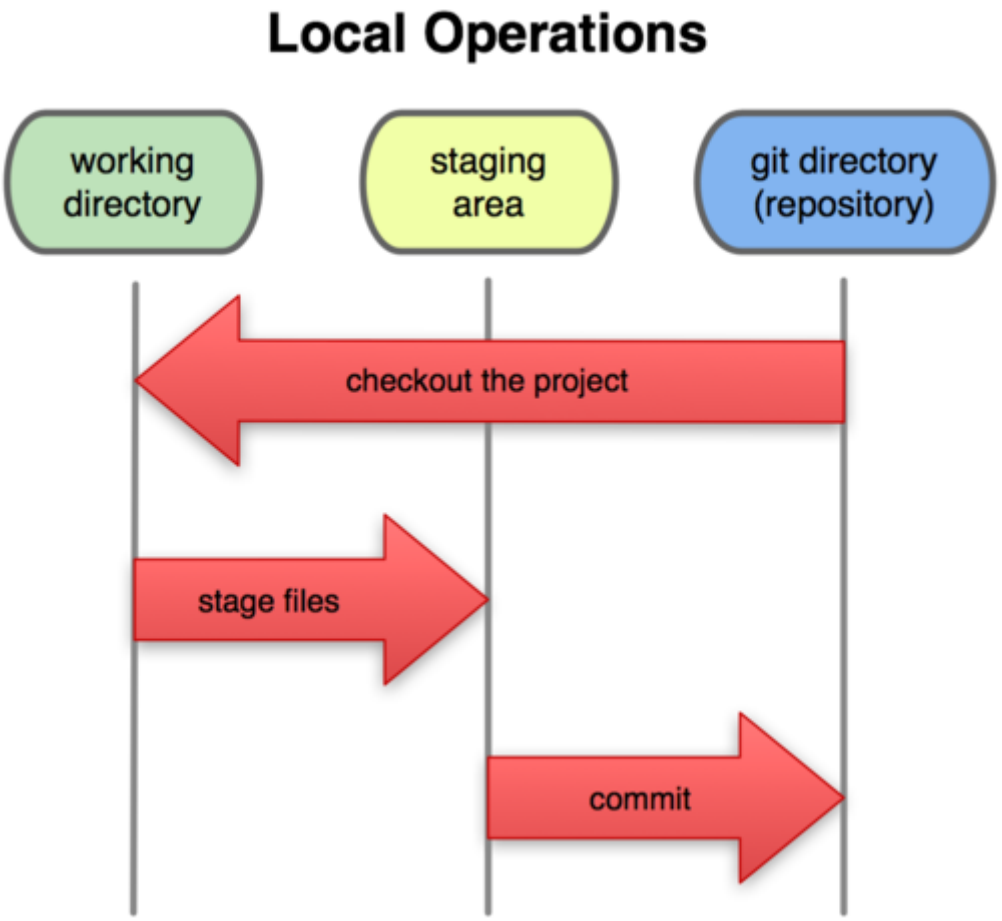

Рис. 3 Рабочий каталог, область подготовленных файлов, каталог Git.

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

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

Стандартный рабочий процесс с использованием Git выглядит примерно так:

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

Если рабочая версия файла совпадает с версией в каталоге Git, 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 Вы можете прочесть в литературе, приложенной к уроку.

4.Установка Git.

Первое, что Вам необходимо сделать, — установить Git. Есть несколько способов сделать это; два основных — установка из исходников и установка собранного пакета для вашей платформы.

Установка из исходников

Если есть возможность, то, как правило, лучше установить Git из исходных кодов, поскольку так вы получите самую свежую версию. Каждая новая версия Git обычно включает полезные улучшения пользовательского интерфейса, поэтому получение последней версии — часто лучший путь, если, конечно, вас не затрудняет установка программ из исходников. К тому же, многие дистрибутивы Linux содержат очень старые пакеты. Поэтому, если только Вы не на очень свежем дистрибутиве или используете пакеты из экспериментальной ветки, установка из исходников может быть самым лучшим решением.

Для установки Git Вам понадобятся библиотеки, от которых он зависит:

- curl,
- zlib,
- openssl,
- expat,
- libiconv.

Например, если в вашей системе менеджер пакетов — yum (Fedora), или apt-get (Debian, Ubuntu), можно воспользоваться следующими командами, чтобы разрешить все зависимости:

```
$ yum install curl-devel expat-devel gettext-devel openssl-devel zlib-devel
$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext libz-dev libssl-dev
```

Установив все необходимые библиотеки, можно идти дальше и скачать последнюю версию с сайта Git:

http://git-scm.com/download

Теперь скомпилируйте и установите:

```
$ tar -zxf git-1.7.2.2.tar.gz
$ cd git-1.7.2.2
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

После этого вы можете скачать Git с помощью самого Git, чтобы получить обновления:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

Как установить Git в Linux и на Mac Вы можете просмотреть в статьях и летературе приложенных к уроку, а мы рассмотрим установку Git в Windows.

Установить Git в Windows очень просто. У проекта msysGit процедура установки - одна из самых простых. Просто скачайте exe-файл инсталлятора со страницы проекта на GitHub и запустите его:

http://msysgit.github.com/

После установки у вас будет как консольная версия (включающая SSH-клиент, который пригодится позднее), так и стандартная графическая.

5. GitHub.

Сделаем небольшое отступление о том, что такое GitHub.

GitHub - самый крупный веб-сервис для хостинга IT-проектов и их совместной разработки. Основан на системе контроля версий Git и разработан на Ruby on Rails[5] и Erlang компанией GitHub, Inc.

Сервис абсолютно бесплатен для проектов с открытым исходным кодом и предоставляет им все возможности, а для частных проектов предлагаются различные платные тарифные планы.

Создатели сайта называют GitHub «социальной сетью для разработчиков». Кроме размещения кода, участники могут общаться, комментировать правки друг друга, а также следить за новостями знакомых. С помощью широких возможностей Git программисты могут объединять свои репозитории - GitHub предлагает удобный интерфейс для этого и может отображать вклад каждого участника в виде дерева.

6. Первоначальная настройка Git

Теперь, когда Git установлен в вашей системе, Вы наверняка захотите настроить среду для работы с Git под себя. Это нужно сделать только один раз — при обновлении версии Git настройки сохранятся. Но вы можете поменять их в любой момент, выполнив те же команды снова.

В состав Git входит утилита git config, которая позволяет просматривать и устанавливать параметры, контролирующие все аспекты работы Git и его внешний вид. Эти параметры могут быть сохранены в трёх местах:

- Файл /etc/gitconfig содержит значения, общие для всех пользователей системы и для всех их репозиториев. Если при запуске git config указать параметр --system, то параметры будут читаться и сохраняться именно в этот файл.
- Файл ~/.gitconfig хранит настройки конкретного пользователя. Этот файл используется при указании параметра --global.
- Конфигурационный файл в каталоге Git'a (.git/config) в том репозитории, где вы находитесь в данный момент. Эти параметры действуют только для данного конкретного репозитория. Настройки на каждом следующем уровне подменяют настройки из предыдущих уровней, то есть значения в .git/config перекрывают соответствующие значения в /etc/gitconfig.

В системах семейства Windows Git ищет файл .gitconfig в каталоге $HOME (C:\Documents and Settings\$USER или C:\Users\$USER для большинства пользователей). Кроме того Git ищет файл /etc/gitconfig, но уже относительно корневого каталога MSys, который находится там, куда вы решили установить Git, когда запускали инсталлятор.

Имя пользователя.

Первое, что Вам следует сделать после установки Git- это указать ваше имя и адрес электронной почты. Это важно, потому что каждый коммит в Git содержит эту информацию, и она включена в коммиты, передаваемые вами, и не может быть далее изменена:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Если указана опция --global, то эти настройки достаточно сделать только один раз, поскольку в этом случае Git будет использовать эти данные для всего, что вы делаете в этой системе. Если для каких-то отдельных проектов вы хотите указать другое имя или электронную почту, можно выполнить эту же команду без параметра --global в каталоге с нужным проектом.

Выбор редактора.

Вы указали своё имя, и теперь можно выбрать текстовый редактор, который будет использоваться, если будет нужно набрать сообщение в Git. По умолчанию Git использует стандартный редактор вашей системы, обычно это Vi или Vim. Если вы хотите использовать другой текстовый редактор, например, Emacs, можно сделать следующее:

$ git config --global core.editor emacs

Утилита сравнения.

Другая полезная настройка, которая может понадобиться — встроенная diff-утилита, которая будет использоваться для разрешения конфликтов слияния. Например, если вы хотите использовать vimdiff:

$ git config --global merge.tool vimdiff
Git умеет делать слияния при помощи kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge и opendiff, но вы можете настроить и другую утилиту.

Проверка настроек.

Если вы хотите проверить используемые настройки, можете использовать команду git config --list, чтобы показать все, которые Git найдёт:
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
Некоторые ключи (названия) настроек могут появиться несколько раз, потому что Git читает один и тот же ключ из разных файлов (например
из /etc/gitconfig и ~/.gitconfig). В этом случае Git использует последнее значение для каждого ключа.

Как получить помощь?

Если вам нужна помощь при использовании Git, есть три способа открыть страницу руководства по любой команде Git:

$ git help <команда>
$ git <команда> --help
$ man git-<команда>
Например, так можно открыть руководство по команде config:

$ git help config

Эти команды хороши тем, что ими можно пользоваться всегда, даже без подключения к сети. Если руководства недостаточно и Вам нужна персональная помощь, вы можете поискать её на каналах #git и #github IRC-сервера Freenode (irc.freenode.net). Обычно там сотни людей, отлично знающих Git, которые могут помочь.

7.Основы ветвления и слияния.

Основы ветвления

Представим, что вы работаете над своим проектом и уже имеете пару коммитов(Рисунок 4).

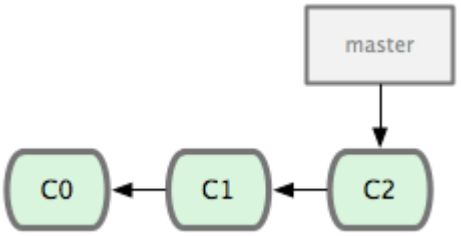

Рис.4 Короткая и простая история коммитов.

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 5).

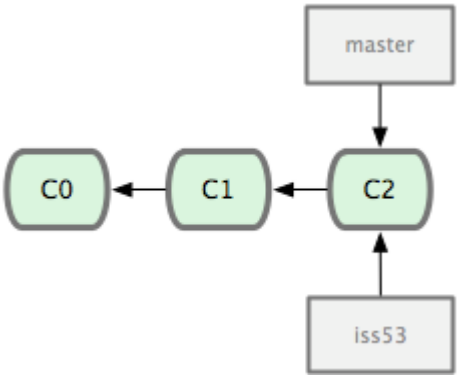

Рис. 5. Создание новой ветки / указателя.

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(Рисунок 6):
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'

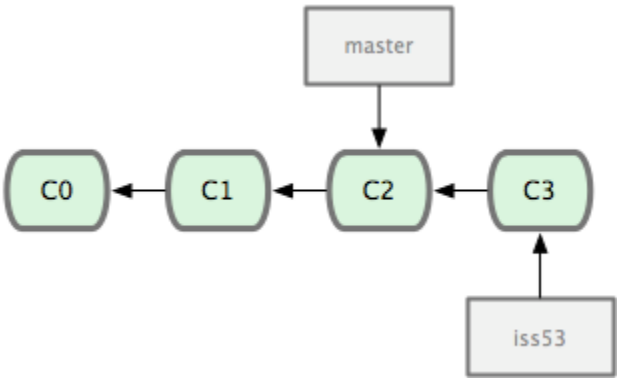

Рис. 6 Ветка iss53 передвинулась вперёд во время работы.

Теперь вы узнаёте о том, что есть проблема с веб-сайтом, которую необходимо немедленно устранить. С Git 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.
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).
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 которуюВы 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 (Рисунок 7):

$ 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(-)

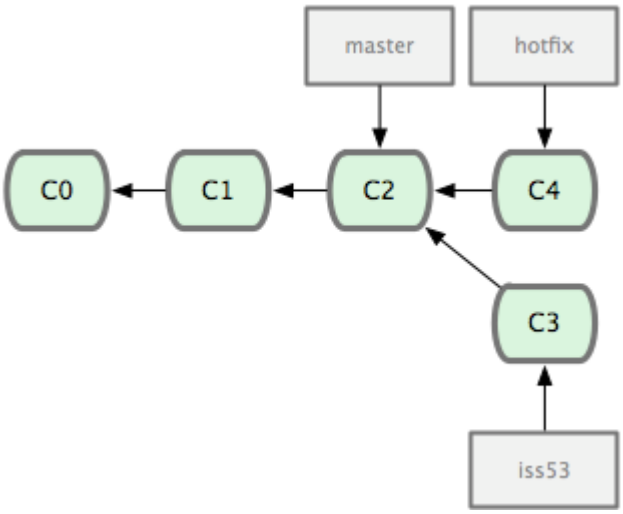

Рис. 7 Ветка для решения срочной проблемы базируется на ветке master.

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 8).

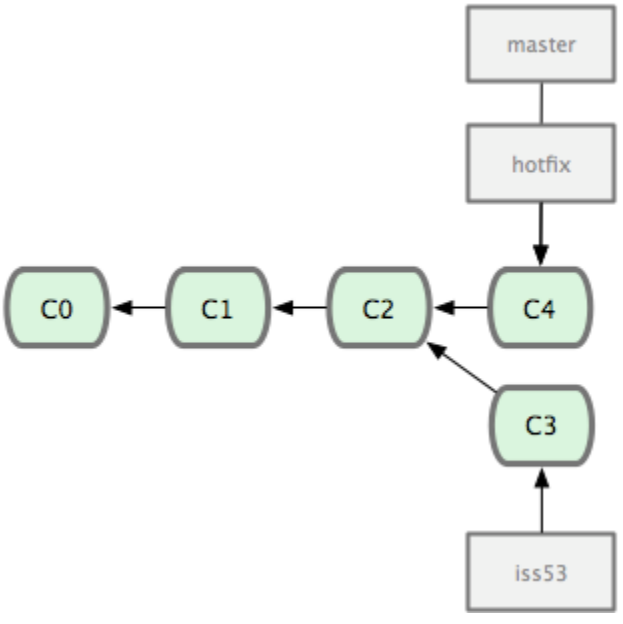

Рис. 8 После слияния ветка master указывает туда же, куда и ветка hotfix.

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(Рисунок 9):

```
$ 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(-)
```

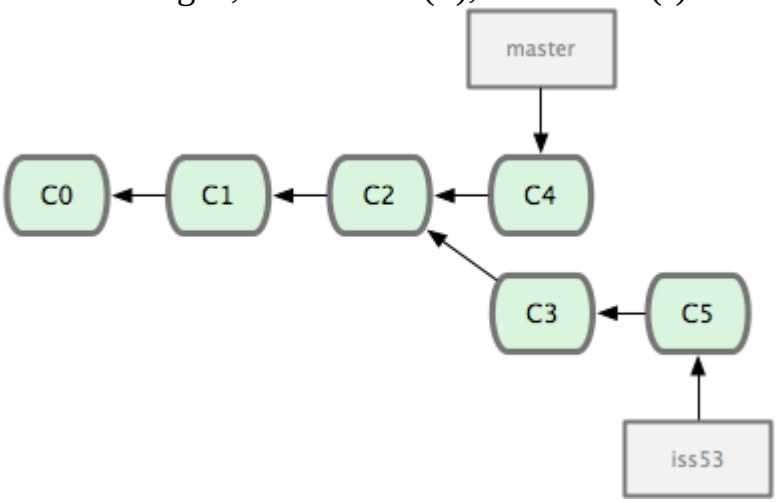

Рис. 9 Ветка iss53 может двигаться вперёд независимо.

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 10 выделены три снимка состояния, которые Git будет использовать для слияния в данном случае.

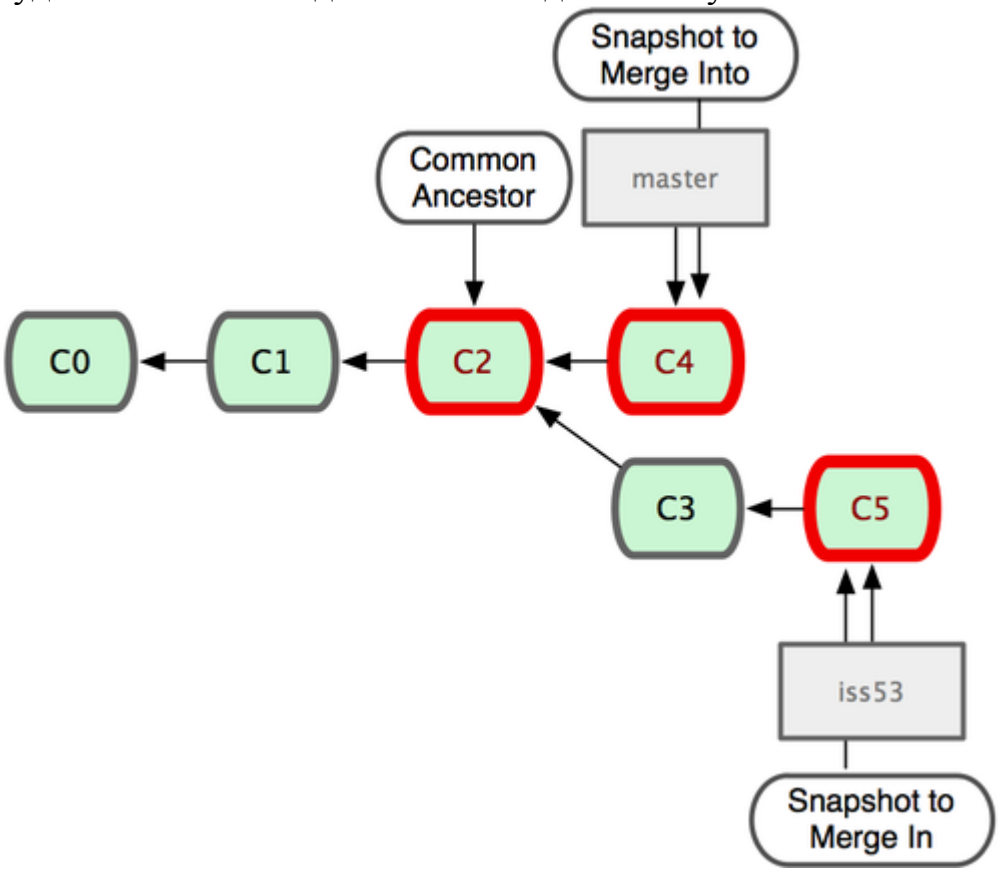

Рис. 10 Git автоматически определяет наилучшего общего предка для слияния веток.

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 гораздо более простым занятием, чем в других системах.

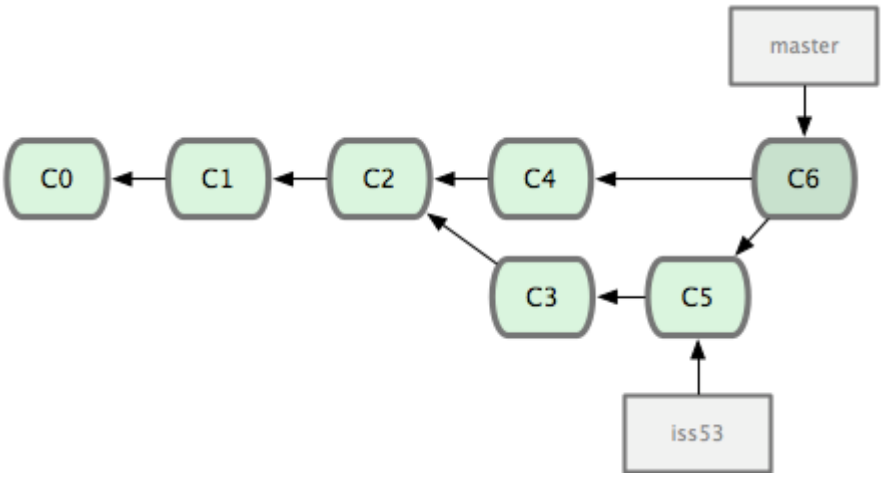

Рис. 11 Git автоматически создаёт новый коммит, содержащий результаты слияния.

Теперь, когда вы осуществили слияние ваших наработок(Рисунок 11),
ветка iss53 вам больше не нужна. Можете удалить её и затем вручную закрыть карточку (ticket) в своей системе:
$ git branch -d iss53

О Git можно говорить ещё долго. Мы с Вами рассмотрели общее понятие о том, что такое Git, и в чём его отличие от централизованных систем контроля версий, а все остальные подробности Вы можете прочесть в книгах и руководстве пользователя Git.