?????????????? git

Теперь, когда вы обзавелись настроенным удалённым Git-репозиторием, т. е. местом, где разработчики могут обмениваться своим кодом, а также познакомились с основными командами Git для локальной работы, мы рассмотрим, как задействовать некоторые распределённые рабочие процессы, предлагаемые Git.
В этой главе мы рассмотрим работу с Git в распределённой среде как в роли рядового разработчика, так и в роли системного интегратора. То есть вы научитесь успешно вносить свой код в проект, делая это как можно более просто и для вас, и для владельца проекта, а также научитесь тому, как сопровождать проекты, в которых участвует множество разработчиков.

- Сопровождение проекта

?????????????? ???????

В дополнение к эффективному участию в проекте, было бы неплохо знать как его сопровождать. Сопровождение может включать в себя принятие и применение патчей, сгенерированных с помощью `format-patch` и отправленных вам по почте, или интеграцию изменений в ветках удалённых репозиториев. Независимо от того, поддерживаете ли вы канонический репозиторий или просто хотите помочь в проверке или применении патчей, вам необходимо знать каким образом следует принимать работу, чтобы это было наиболее понятно для других участников и было бы приемлемым для вас в долгосрочной перспективе.

?????? ? ????????????? ???????

Перед интеграцией новых изменений желательно проверить их в тематической ветке — временной ветке, специально созданной для проверки работоспособности новых изменений. Таким образом, можно применять патчи по одному и пропускать неработающие, пока не найдётся время к ним вернуться. Если вы создадите ветку с коротким и понятным названием, основанным на тематике изменений, например, `ruby_client` или что-то похожее, то без труда можно будет вернуться к ней, если пришлось на какое-то время отказаться от работы с ней. Сопровождающему Git проекта свойственно использовать пространство имен для веток, например, `sc/ruby_client`, где `sc` — это сокращение от имени того, кто проделал работу. Как известно, ветки можно создавать на основании базовой ветки, например:

```
$ git branch sc/ruby_client master
```

Если вы хотите сразу переключиться на создаваемую ветку, то используйте опцию `checkout -b`:

```
$ git checkout -b sc/ruby_client master
```

Теперь вы можете добавить новые изменения в созданную тематическую ветку и определить хотите ли слить эти изменения в ваши долгосрочные ветки.

?????????? ??????, ?????????? ?? ?????

Если вы получили патч по почте и его нужно интегрировать в проект, то следует проанализировать его, применив сначала в тематической ветке. Существует два варианта применения полученного по почте патча: `git apply` или `git am`.

?????????? ????? ???????? apply

Если полученный по почте патч был создан командой `git diff` или Unix командой `diff` (что не рекомендуется делать), то применить его можно командой `git apply`. Предположим, патч сохранен здесь `/tmp/patch-ruby-client.patch`, тогда применить его можно вот так:

```
$ git apply /tmp/patch-ruby-client.patch
```

Это действие модифицирует файлы в вашем рабочем каталоге. Выполнение команды практически эквивалентно выполнению команды `patch -p1`, однако, является более параноидальным и принимает меньше неточных совпадений, чем `patch`. При этом обрабатываются добавления, удаления и переименования файлов, указанные в формате `git diff`, тогда как `patch` этого не делает. Наконец, `git apply` использует модель «применить всё или отменить всё», где изменения либо применяются полностью, либо не применяются вообще, тогда как `patch` может частично применить патч файлы, приведя ваш рабочий каталог в непонятное состояние. В целом, `git apply` более консервативен, чем `patch`. После выполнения команды новый коммит не создаётся и его нужно делать вручную.

Командой `git apply` можно проверить корректность применения патча до его фактического применения, используя `git apply --check`:

```
$ git apply --check 0001-see-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

Если ошибок не выведено, то патч может быть применён без проблем. Так же, в случае ошибки эта команда возвращает отличное от 0 значение, что позволяет использовать её в скриптах.

?????????? ????? ???????? `am`

Если участник проекта пользователь Git и умеет пользоваться командой `format-patch` для генерации патчей, то вам будет легче, так как в патч включается информация об авторе и сообщение коммита. Если возможно, требуйте от ваших участников использовать команду `format-patch` вместо `diff` для генерации патчей. Вам останется использовать `git apply` только для устаревших патчей и подобного им.

Для применения патча, созданного с помощью `format-patch`, используйте `git am` (команда названа `am` потому что применяет «apply» набор патчей в формате «mailbox»). С технической точки зрения она просто читает mbox-файл, в котором в виде обычного текста хранится одно или несколько электронных писем. Этот файл имеет следующий вид:

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] Add limit to log function
```

```
Limit log functionality to the first 20
```

Это начало вывода команды `format-patch`, которая рассматривалась в предыдущем разделе; это так же представляет собой валидный формат mbox. Если кто-то отправил патч, корректно сформированный командой `git send-email`, и вы сохранили его в формате mbox, то можно указать передать этот файл в качестве аргумента команде `git am`, которая начнёт применять все найденные в файле патчи. Если вы используете почтовый клиент, который умеет сохранять несколько писем в формате mbox, то можно сохранить сразу серию патчей в один файл, а затем применить их за раз, используя `git am`.

Так или иначе, если кто-нибудь загрузит созданный с помощью `format-patch` патч файл в систему управления задачами, то вы сможете сохранить его себе и применить локально с помощью `git am`:

```
$ git am 0001-limit-log-function.patch
Applying: Add limit to log function
```

Как вы могли заметить, патч применился без конфликтов, а так же был создан новый коммит. Информация об авторе была извлечена из заголовков письма `From` и `Date`, а сообщение коммита — из заголовка `Subject` и тела письма (до патча). Например, для применённого патча из примера выше коммит будет выглядеть следующим образом:

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author:     Jessica Smith <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit:     Scott Chacon <schacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700

   Add limit to log function

   Limit log functionality to the first 20
```

`Commit` информация указывает на того, кто применил патч и когда это было сделано. `Author` информация указывает на того, кто изначально создал патч и когда это было сделано.

Однако, возможна ситуация, когда патч не может быть бесконфликтно применён. Возможно, ваша основная ветка сильно расходится с той веткой, на основании которой был создан патч, или он зависит от другого, ещё не применённого патча. В таком случае работа `git am` будет прервана, а так же выведена подсказка со списком возможных действий:

```
$ git am 0001-see-if-this-helps-the-gem.patch
Applying: See if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

Эта команда добавит соответствующие маркеры во все файлы где есть конфликты, аналогично конфликтам слияния или перебазирования. Для решения такой проблемы используется аналогичный подход — отредактируйте файлы исправив конфликты, добавьте их в индекс и выполните команду `git am --resolved` для перехода к следующему патчу:

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: See if this helps the gem
```

При желании, вы можете указать опцию `-3`, чтобы Git попробовал провести трёхстороннее слияние. Эта опция не включена по умолчанию, так как она не будет работать, если коммит, на который ссылается патч, отсутствует в вашем репозитории. Если у вас есть тот коммит, на который ссылается конфликтующий патч, то использование опции `-3` приводит к более умному применению конфликтующего патча:

```
$ git am -3 0001-see-if-this-helps-the-gem.patch
Applying: See if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

В данном случае, без использования опции `-3` патч будет расценён как конфликтующий. При использовании опции `-3` патч будет применён без конфликтов.

Если вы применяете несколько патчей из файла mbox, то можно запустить `git am` в интерактивном режиме, в котором перед обработкой каждого патча будет задаваться вопрос о дальнейших действиях:

```
$ git am -3 -i mbox
Commit Body is:
```

```
-------------------------
See if this helps the gem
-------------------------
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

Это отличная возможность посмотреть содержимое патча перед его применением или пропустить его, если он уже был применён.

Когда все патчи применены и созданы коммиты в текущей ветке, вы уже можете решить стоит ли и как интегрировать их в более долгоживущую ветку.

?????????? ????????? ?????

Если участник проекта создал свой Git репозиторий, отправил в него свои изменения, а затем прислал вам ссылку и название ветки, куда были отправлены изменения, то вы можете добавить этот репозиторий как удалённый и провести слияние локально.

К примеру, Джессика отправила вам письмо, в котором сказано, у неё есть новый функционал в ветке `ruby-client` её репозитория. Добавив удалённый репозиторий и получив изменения из этой ветки, вы можете протестировать изменения извлекая их локально:

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

Если она снова пришлёт вам письмо с указанием на новый функционал уже в другой ветке, то для его получения достаточно `fetch` и `checkout`, так как удалённый репозиторий уже подключён.

Это очень полезно, когда вы постоянно работаете с этим человеком. Однако, от тех, кто редко отправляет небольшие патчи, будет проще принимать их по почте, чем требовать от всех поддержания собственных серверов с репозиториями, постоянно добавлять их как удалённые, а затем удалять и всё это ради нескольких патчей. Так же вы вряд ли захотите иметь сотни удалённых репозиториев, каждый из которых нужен только для одного или нескольких патчей. К счастью, скрипты и различные сервисы облегчают задачу, но во многом зависят от того как работаете вы и участники вашего проекта.

Отличительным преимуществом данного подхода является получение истории коммитов. Хоть возникновение конфликтов слияния и закономерно, но вы знаете с какого момента это произошло; корректное трёхстороннее слияние более предпочтительно, чем указать опцию `-3` и надеяться, что патч основан на коммите, к которому у вас есть доступ.

Если вы с кем-то не работаете постоянно, но всё равно хотите использовать удалённый репозиторий, то можно указать ссылку на него в команде `git pull`. Это приведёт к

однократному выполнению, а ссылка на репозиторий сохранена не будет.

```
$ git pull https://github.com/onetimeguy/project
From https://github.com/onetimeguy/project
 * branch            HEAD       -> FETCH_HEAD
Merge made by the 'recursive' strategy.
```

??????????? ??????????? ?????????

На текущий момент у вас есть тематическая ветка, содержащая предоставленные изменения. Сейчас вы можете определиться что с ними делать. В этом разделе рассматривается набор команд, которые помогут вам увидеть что именно будет интегрировано, если вы решите слить изменения в основную ветку.

Обычно, полезно просмотреть все коммиты текущей ветки, которые ещё не включены в основную. Вы можете исключить коммиты, которые уже есть в вашей основной ветке добавив опцию `--not` перед её названием. Это аналогично указанию использовавшегося ранее формата `master..contrib`. Например, если участник проекта отправил вам два патча, а вы создали ветку с названием `contrib` и применили их, то можно выполнить следующую команду:

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700

    See if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700

    Update gemspec to hopefully work better
```

Для просмотра изменений, представленных в каждом коммите, можно использовать опцию `-p` команды `git log`, которая выведет разницу по каждому коммиту.

Для просмотра полной разницы того, что произойдёт если вы сольёте изменения в другую ветку, вам понадобится использовать возможно странный способ для получения корректных результатов:

```
$ git diff master
```

Эта команда может вводить в заблуждение, но точно покажет разницу. Если ваша `master` ветка продвинулась вперед с тех пор как вы создали тематическую ветку, то вы получите на первый взгляд странные результаты. Это происходит потому, что Git непосредственно сравнивает снимки последних коммитов текущей и `master` веток. Например, если вы добавили строку в файл в ветке `master`, то прямое сравнение снимков будет выглядеть как будто тематическая ветка собирается удалить эту строку.

Это не проблема, если ветка `master` является непосредственным родителем вашей тематической ветки, но если история обоих веток изменилась, то разница будет выглядеть как добавление всех изменений из тематической ветки и удаление всего нового из `master` ветки.

Что действительно нужно видеть, так это изменения тематической ветки, которые предстоит слить в `master` ветку. Это можно сделать, сказав Git сравнивать последний коммит тематической ветки с первым общим родителем для обоих веток.

Технически это делается за счёт явного указания общего коммита и применения разницы к нему:

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

или более кратко:

```
$ git diff $(git merge-base contrib master)
```

Однако это не удобно, поэтому Git предоставляет более короткий способ: синтаксис троеточия. При выполнении команды `diff`, следует поставить три точки после имени ветки для получения разницы между ней и текущей веткой, относительно общего родителя с другой веткой:

```
$ git diff master...contrib
```

Данная команда отобразит проделанную работу только из тематической ветки, относительно общего родителя с веткой `master`. Полезно запомнить указанный синтаксис.

?????????? ?????????? ??????

Когда все изменения в текущей тематической ветке готовы к интеграции с основной веткой, возникает вопрос как это сделать. Кроме этого, какой рабочий процесс вы хотите использовать при сопровождении вашего проекта? У вас несколько вариантов, давайте рассмотрим некоторые из них.

????? ???????

В простом рабочем процессе проделанная работа просто сливается в ветку `master`. При таком сценарии у вас есть ветка `master`, которая содержит стабильный код. Когда работа в тематической ветке завершена или вы проверили чью-то работу, вы сливаете её в ветку `master` и удаляете, затем процесс повторяется.

Если в репозитории присутствуют две ветки `ruby_client` и `php_client` с проделанной работой, как показано на рисунке История с несколькими тематическими ветками, и вы сначала сливаете ветку `ruby_client`, а затем `php_client`, то состояние вашего репозитория будет выглядеть как показано на рисунке Слияние тематической ветки.

История с несколькими тематическими ветками
Рисунок 72. История с несколькими тематическими ветками
Слияние тематической ветки
Рисунок 73. Слияние тематической ветки

Это, пожалуй, простейший рабочий процесс и его использование проблематично в больших или более стабильных проектах, где вы должны быть более осторожны с предоставленными изменениями.

Если у вас очень важный проект, то возможно вам стоит использовать двухступенчатый цикл слияния. При таком сценарии у вас имеются две долгоживущие ветки `master` и `develop`, где в `master` сливаются только очень стабильные изменения, а все новые доработки интегрируются в ветку `develop`. Обе ветки регулярно отправляются в публичный репозиторий. Каждый раз, когда новая тематическая ветка готова к слиянию (Перед слиянием тематической ветки), вы сливаете её в `develop` (После слияния тематической ветки); затем, когда вы выпускаете релиз, ветка `master` смещается на стабильное состояние ветки `develop` (После релиза проекта).

Перед слиянием тематической ветки
Рисунок 74. Перед слиянием тематической ветки
После слияния тематической ветки
Рисунок 75. После слияния тематической ветки
После релиза тематической ветки
Рисунок 76. После релиза проекта

Таким образом, люди могут клонировать репозиторий вашего проекта и использовать ветку `master` для сборки последнего стабильного состояния и получения актуальных изменений или использовать ветку `develop`, которая содержит самые последние изменения. Вы также можете продолжить эту концепцию, имея интеграционную ветку `integrate`, в которой объединяется вся работа. После того, как кодовая база указанной ветки стабильна и пройдены все тесты, она сливается в ветку `develop`, а после того, как стабильность слитых изменений доказана, вы перемещаете состояние ветки `master` на стабильное.

????? ? ??????? ??????????? ???????

В проекте Git присутствуют четыре долгоживущие ветки: `master`, `next`, `seen` (ранее `pu` — предложенные обновления) для новой работы и `maint` для поддержки обратной совместимости. Предложенные участниками проекта наработки накапливаются в тематических ветках основного репозитория по ранее описанному принципу (рис. Управление сложным набором параллельно разрабатываемых тематических веток). На этом этапе производится оценка содержимого тематических веток, чтобы определить, работают ли предложенные фрагменты так, как положено, или им требуется доработка. Если всё в порядке, тематические ветки сливаются в ветку `next`, которая отправляется на сервер, чтобы у каждого была возможность опробовать результат интеграции.

Управление сложным набором параллельно разрабатываемых тематических веток
Рисунок 77. Управление сложным набором параллельно разрабатываемых тематических веток

Если содержимое тематических веток требует доработки, оно сливается в ветку `seen`. Когда выясняется, что предложенный код полностью стабилен, он сливается в ветку `master`. Затем ветки `next` и `seen` перестраиваются на основании `master`. Это означает, что `master` практически всегда двигается только вперед, `next` время от времени перебазируется, а `seen` перебазируется ещё чаще:

Слияние тематических веток в долгоживущие ветки интеграции
Рисунок 78. Слияние тематических веток в долгоживущие ветки интеграции

После того, как тематическая ветка окончательно слита в `master`, она удаляется из репозитория. Репозиторий также содержит ветку `maint`, которая ответвляется от последнего релиза для предоставления патчей, если требуется поддержка обратной совместимости. Таким образом, после клонирования проекта у вас будет четыре ветки, дающие возможность перейти на разные стадии его разработки, в зависимости от того, на сколько передовым вы хотите быть или как вы собираетесь участвовать в проекте; вместе с этим, рабочий процесс структурирован, что помогает сопровождающему проекта проверять поступающий код. Рабочий процесс проекта Git специфицирован. Для полного понимания процесса обратитесь к Git Maintainer’s guide.

????? ? ???????????????? ? ???????

Некоторые сопровождающие предпочитают перебазировать или выборочно применять (cherry-pick) изменения относительно ветки `master` вместо слияния, что позволяет поддерживать историю проекта в линейном виде. Когда проделанная работа из тематической ветки готова к интеграции, вы переходите на эту ветку и перебазируете её относительно ветки `master` (или `develop` и т. д.). Если конфликты отсутствуют, то вы можете просто сдвинуть состояние ветки `master`, что обеспечивает линейность истории проекта.

Другим способом переместить предлагаемые изменений из одной ветки в другую является их отбор коммитов (cherry-pick). Отбор в Git похож на перебазирование для одного коммита. В таком случае формируется патч для выбранного коммита и применяется к текущей ветке. Это полезно, когда в тематической ветке присутствует несколько коммитов, а вы хотите взять только один из них, или в тематической ветке только один коммит и вы предпочитаете использовать отбор вместо перебазирования. Предположим, ваш проект выглядит так:

Пример истории, из которой нужно отобрать отдельные коммиты
Рисунок 79. Пример истории, из которой нужно отобрать отдельные коммиты

Для применения коммита `e43a6` к ветке `master` выполните команду:

```
$ git cherry-pick e43a6
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
 3 files changed, 17 insertions(+), 3 deletions(-)
```

Это действие применит изменения, содержащиеся в коммите `e43a6`, но будет сформирован новый коммит с другим значением SHA-1. После этого история будет выглядеть так:

История после отбора коммита из тематической ветки
Рисунок 80. История после отбора коммита из тематической ветки

Теперь тематическую ветку можно удалить, отбросив коммиты, которые вы не собираетесь включать в проект.

??????????? «Rerere»

Если вы часто производите перебазирование и слияние или поддерживаете долгоживущие тематические ветки, то в Git есть специальная возможность под названием «rerere», призванная вам помочь.

Rerere означает «reuse recorded resolution» (повторно использовать сохранённое решение) — это способ сокращения количества операций ручного разрешения конфликтов. Когда эта опция включена, Git будет сохранять набор образов до и после успешного слияния, а также разрешать конфликты самостоятельно, если аналогичные конфликты уже были разрешены ранее.

Эта возможность реализована как команда и как параметр конфигурации. Параметр конфигурации называется `rerere.enabled`, который можно включить глобально следующим образом:

```
$ git config --global rerere.enabled true
```

После этого любое разрешение конфликта слияния будет записано на случай повторного использования.

Если нужно, вы можете обращаться к кэшу «rerere» напрямую, используя команду `git rerere`. Когда команда вызвана без параметров, Git проверяет базу данных и пытается найти решение для разрешения текущего конфликта слияния (точно так же как и при установленной настройке `rerere.enabled` в значение `true`). Существует множество дополнительных команд для просмотра, что именно будет записано, удаления отдельных записей из кэша, а так же его полной очистки. Более детально «rerere» будет рассмотрено в разделе Rerere главы 7.

????????? ???? ??????

После выпуска релиза, возможно, вы захотите пометить текущее состояние так, чтобы можно было вернуться к нему в любой момент. Для этого можно добавить тег, как было описано в главе Основы Git. Кроме этого, вы можете добавить цифровую подпись для тега, выглядеть это будет вот так:

```
$ 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@gmail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

Если вы используете цифровую подпись при расстановке тегов, то возникает проблема распространения публичной части PGP ключа, использованного при создании подписи. Сопровождающий Git проекта может решить эту проблему добавив в репозиторий свой публичный ключ как бинарный объект и установив ссылающийся на него тег. Чтобы это сделать, выберите нужный ключ из списка доступных, который можно получить с помощью команды `gpg --list-keys`:

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
---------------------------------
pub   1024D/F721C45A 2009-02-09 [expires: 2010-02-09]
uid                  Scott Chacon <schacon@gmail.com>
sub   2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

Затем экспортируйте выбранный ключ и поместите его непосредственно в базу данных Git при помощи команды `git hash-object`, которая создаст новый объект с содержимым ключа и вернёт SHA-1 этого объекта:

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Теперь, когда ваш публичный ключ находится в репозитории, можно поставить указывающий на него тег, используя полученное ранее значение SHA-1:

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Выполнив команду `git push --tags`, `maintainer-pgp-pub` тег станет общедоступным. Теперь все, кто захочет проверить вашу подпись, могут импортировать ваш публичный ключ, предварительно получив его из репозитория:

```
$ git show maintainer-pgp-pub | gpg --import
```

После этого можно проверять цифровую подпись ваших тегов. Кроме этого, вы можете включить дополнительные инструкции по проверке вашей подписи в сообщение тега, которое будет отображаться каждый раз при вызове команды `git show <tag>`.

????????? ?????? ??????

Git не использует монотонно возрастающие идентификаторы для коммитов, поэтому если вы хотите получить читаемые имена коммитов, то воспользуйтесь командой `git describe` для нужного коммита. Git вернёт имя ближайшего тега, количество коммитов после него и частичное значение SHA-1 для указанного коммита (с префиксом в виде буквы «g» — означает Git):

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

Таким образом, вы можете сделать снимок или собрать сборку и дать ей понятное для человека название. К слову, если вы клонируете репозиторий Git и соберете его из исходного кода, то вывод команды `git --version` будет примерно таким же. Если попытаться получить имя коммита, которому назначен тег, то результатом будет название самого тега.

По умолчанию, команда `git describe` поддерживает только аннотированные теги (созданные с использованием опций `-a` или `-s`); если вы хотите использовать легковесные (не аннотированные) теги, то укажите команде параметр `--tags`. Также это название можно использовать при выполнении команд `git checkout` и `git show`, но в будущем они могут перестать работать из-за сокращённого значения SHA-1. К примеру, ядро Linux недавно перешло к использованию 10 символов в SHA-1 вместо 8 чтобы обеспечить уникальность каждого объекта, таким образом предыдущие результаты `git describe` стали недействительными.

?????????? ??????

Время делать релиз сборки. Возможно, вы захотите сделать архив последнего состояния вашего кода для тех, кто не использует Git. Для создания архива выполните команду `git archive`:

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

Открывший этот tarball-архив пользователь получит последнее состояние кода проекта в каталоге `project`. Точно таким же способом можно создать zip-архив, просто добавив опцию `--format=zip` для команды `git archive`:

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

В итоге получим tarball- и zip-архивы с релизом проекта, которые можно загрузить на сайт или отправить по почте.

??????? ??????? (Shortlog)

Сейчас самое время оповестить людей из списка рассылки, которые хотят знать что происходит с вашим проектом. С помощью команды `git shortlog` можно быстро получить список изменений, внесённых в проект с момента последнего релиза или предыдущей рассылки. Она собирает все коммиты в заданном интервале; например, следующая команда выведет список коммитов с момента последнего релиза с названием v1.0.1:

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (6):
      Add support for annotated tags to Grit::Tag
      Add packed-refs annotated tag support.
      Add Grit::Commit#to_patch
      Update version and History.txt
      Remove stray `puts`
      Make ls_tree ignore nils

Tom Preston-Werner (4):
      fix dates in history
      dynamic version method
      Version bump to 1.0.2
      Regenerated gemspec for version 1.0.2
```

И так, у вас есть готовая к отправке сводка коммитов начиная с версии v1.0.1, сгруппированных по авторам.