Михаил Иванов @ivanych
Пользователь

21 ноября 2009 в 23:33

Командная работа в Git

Git*

Во всем множестве статей по git'у, которые я смог найти в сети, не хватает одного существенного момента — описания командной работы. То, что обычно описывают как командную работу, на самом деле является просто работой с удаленным репозиторием.

Ниже я хочу описать свой опыт командной работы над проектом с использованием git'а.

1. Общий принцип

Рабочий процесс у меня организован следующим образом.

Я ведущий разработчик тире руководитель отдела тире проджект менеджер. У меня в команде есть несколько разработчиков.

Моя работа заключается в следующем:

1) поговорить с заказчиком
2) превратить смутное и путанное пожелание заказчика в четко сформулированную задачу
3) поставить эту задачу разработчику
4) проверить результат выполнения задачи разработчиком
5) неудовлетворительный результат вернуть разработчику на доработку
6) удовлетворительный результат представить заказчику на утверждение или сразу отправить в продакшн
7) утвержденный заказчиком результат отправить в продакшн
8) неутвержденный заказчиком результат вернуть разработчику на доработку

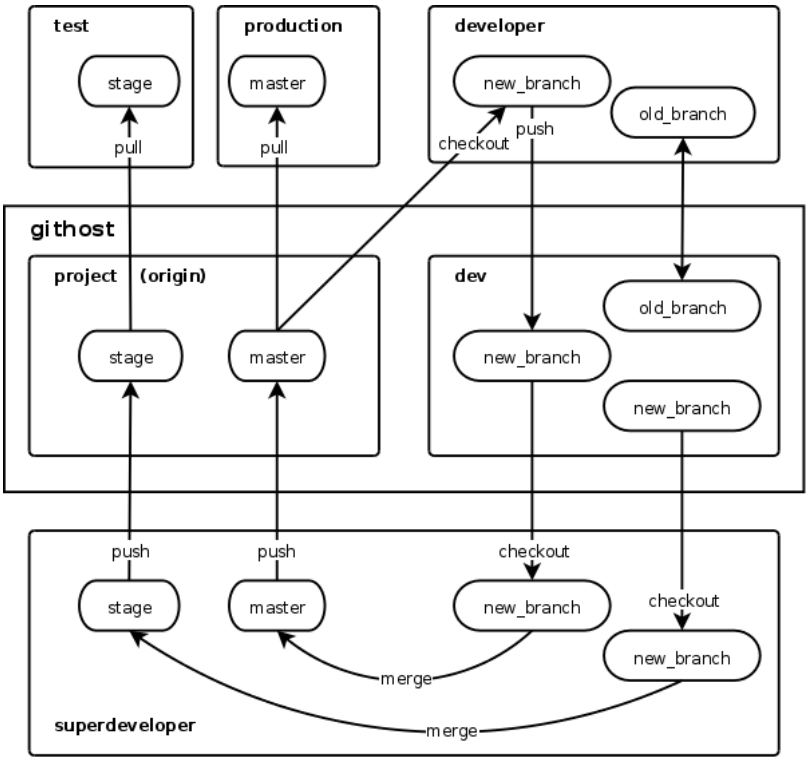
Работа разработчика, соответственно, заключается в следующем:

1) получить от меня задачу
2) выполнить ее
3) отправить мне результат
4) если задача вернулась на доработку — доработать и снова отправить мне

Для управления задачами я использую Trac. Так же в Trac'е ведется документация по проекту, как программерская, так и пользовательская.

Сформулировав задачу, я записываю ее в Trac и назначаю какому-то разработчику. Разработчик, выполнив задачу, переназначает ее мне. Я проверяю рез
и, либо снова переназначаю ее разработчику, либо отмечаю как выполненную.

Алгоритм работы с git'ом и общая картина происходящего схематически представлены на картинке:



Теперь от теоретической части перейдем к практической и разберемся, что все это значит и как работает.

2. Git

Установите Git. Прочитайте какой-нибудь мануал. Разберитесь с локальным репозиторием.

Обратите внимание — эта статья не про команды git'а, а про то, как увязать эти команды в нужную последовательность. Поэтому далее я буду приводить к git'а без подробных объяснений, предполагая, что назначение отдельных команд вам известно.

3. Доступ к репозиторию

При командной работе потребуется обеспечить доступ к репозиторию нескольким пользователям. Для удобного разруливания прав доступа к репозиторию использую gitosis.

Gitosis — это набор скриптов, реализующих удобное управление git-репозиториями и доступом к ним. Работает это так:

— на сервере заводится пользователь, которому будут принадлежать все репозитории
— все обращения к репозиториям происходят по SSH, под именем этого пользователя, авторизация пользователей производится по ключам
— при входе по SSH автоматически запускаются скрипты gitosis, которые, в зависимости от настройки, разрешают или запрещают дальнейшие действия с репозиториями

Скрипты и конфиги gitosis сами хранятся в репозитории и настраиваются путем отправки коммитов в этот репозиторий. Звучит безумно, но на деле ничего хитрого, вполне просто и удобно.

4. Установка gitosis

Для установки gitosis'а нужно (сюрприз!) вытянуть установочный репозиторий gitosis'а с сервера разработчика gitosis'а:

$ git clone git://eagain.net/gitosis.git

Установить gitosis:

$ cd gitosis
$ su
 # python setup.py install

Установочный репозиторий больше не понадобится, его можно удалить.

Теперь нужно создать в системе пользователя, которому будут принадлежать все репозитории:

$ su
 # adduser gituser

А затем инициализировать gitosis в домашнем каталоге созданного пользователя, с открытым ключом того, кто будет администратором gitosis'а. Открытый

понятно, нужно положить куда-то, откуда его сможет прочитать пользователь gituser:

 # su gituser
$ cd ~
$ gitosis-init < id_rsa.pub

Обратите внимание — пользователь gituser и администратор gitosis'a — это не одно лицо. Пользователь gituser является просто «хранителем» репозиториев никаких действий вообще никогда не выполняет.

Я в качестве администратора gitosis'a использую другого зарегистрированного в системе пользователя. Но, вообще говоря, администратору вовсе не обязат быть зарегистрированным в системе пользователем. Главное, указать при инициализации gitosis'a открытый ключ администратора.

После инициализации gitosis'a в домашнем каталоге пользователя gituser появится каталог repositories, в котором и будут храниться все репозитории. Снача будет только репозиторий gitosis-admin.git, в котором хранятся настройки самого gitosis'a.

Обратите внимание — по некоторым причинам, связанным с особенностями разных версий Питона, может понадобиться прописать права на исполнение ск post-update, находящемуся в репозитории gitosis'a:

$ chmod 755 ~/repositories/gitosis-admin.git/hooks/post-update

На этом установка gitosis'a закончена и начинается настройка.

5. Настройка gitosis'a и создание репозиториев

Настройка gitosis'a заключается в изменении администратором содержимого репозитория gitosis'a.

Становимся администратором gitosis'a (если администратор — зарегистрированный в системе пользователь) или вообще выходим из системы и логинимся администратор зарегистрирован (например, на своем ноутбуке).

Теперь вытягиваем настроечный репозиторий gitosis'a:

$ git clone gituser@githost:gitosis-admin.git

Где githost — это имя сервера, на котором мы установили gitosis (и где будем хранить репозитории).

Обратите внимание — какое бы имя не имел администратор, обращение к серверу всегда выполняется под именем пользователя gituser.

После этого в домашнем каталоге администратора появится каталог gitosis-admin. В этом каталоге нас интересует файл gitosis.conf, именно в нем производ настройка всех репозиториев.

По умолчанию в нем будет нечто такое:

```
[group gitosis-admin]
writable = gitosis-admin
members = admin@host
```

Что означает «пользователю admin разрешена запись в репозиторий gitosis-admin».

Создание нового репозитория заключается в добавлении в конфиг новой группы. Название группы может быть любым, оно просто для понятности. Нам для работы потребуются два репозитория:

```
[group project-write]
writable = project
members = superdeveloper

[group project-read]
readonly = project
members = developer1 developer2 developer3 user1 user2

[group dev]
writable = dev
members = superdeveloper developer1 developer2 developer3
```

где superdeveloper — ведущий разработчик, developer* — разработчики, user* — прочие интересующиеся

Этот конфиг означает следующее:

1) ведущему разработчику позволено писать в главный репозиторий
2) всем позволено читать из главного репозитория
3) ведущему разработчику и всем разработчикам позволено писать в рабочий репозиторий

Все эти пользователи, как и администратор gitosis'а, вовсе не обязательно должны быть зарегистрированными в системе пользователями. Главное, чтобы у были открытые ключи.

Открытые ключи указанных в конфиге пользователей нужно скопировать в каталог gitosis-admin/keydir. Файлы ключей должны иметь имена вида имя_пользователя.pub. В данном примере это будут имена superdeveloper.pub, developer1.pub, user1.pub и.т.д.

После правки конфига и копирования ключей нужно закоммитить изменения в локальный репозиторий:

$ git add.
$ git commit -am 'Add project Project'

И отправить коммит в центральный репозиторий, где сделанные настройки подхватит gitosis:

$ git push

Все, теперь наш сервер репозиториев настроен и нужные репозитории созданы (вру, репозитории еще не созданы, но будут созданы автоматически при пер коммите).

6. Назначение репозиториев

Если в предыдущем разделе у вас возник вопрос — почему для одного проекта нужно два репозитория — то вот ответ.

Первый репозиторий, он же главный — это продакшн. На коде из этого репозитория будет работать боевая копия проекта. Второй репозиторий, он же рабо это разработка. В этом репозитории ведется, понятно, разработка проекта.

Я пробовал разные схемы командной работы и наиболее удобной на данный момент мне представляется «двухрепозиторная» схема. Работа в ней ведется примерно следующим образом:

1) я ставлю задачу разработчику
2) разработчик берет актуальную ветку проекта из главного репозитория и делает с нее локальный бранч
3) в этом бранче разработчик решает поставленную задачу
4) бранч с выполненной задачей разработчик отправляет в рабочий репозиторий
5) я беру из рабочего репозитория этот бранч и проверяю его
6) если задача выполнена правильно, я сливаю этот бранч с актуальной веткой проекта в главном репозитории

Эта схема имеет два ключевых отличия от широко описанной схемы, когда вся разработка идет в ветке master одного репозитория.

Во-первых, в моей схеме разработчики не могут вносить несанкционированные изменения (как случайные, так и намеренные) в актуальную ветку, т.е. в про

Во-вторых, актуальная ветка у меня никогда не бывает в нерабочем состоянии.

Вторая причина, на самом деле, является более важной. Несанкционированные изменения в любой момент можно откатить, на то и система контроля верс вот разрулить нерабочее состояние актуальной ветки может быть весьма затруднительно.

Поясню на примере.

Допустим, разработчик Р1 сделал правку П1 и отправил ее в ветку master. Я проверил эту правку и нашел ее плохой. Плохую правку нужно переделывать.

Пока я проверял, разработчик Р2 сделал правку П2 и тоже отправил ее в master. Эта правка оказалась хорошей. Хорошую правку нужно отправлять в прод

Но теперь в ветке master находятся две правки, хорошая и плохая. Хорошую правку нужно бы отправить в продакшн, но присутствие плохой правки не позв это сделать. Придется ждать, пока плохая правка будет исправлена.

Или другой пример — все правки хорошие, но некоторые не утверждены заказчиком. Неутвержденные правки не дают попасть в продакшн утвержденным, утверждение может происходить довольно долго.

В общем, нужно сделать так, чтобы правками можно было рулить по отдельности друг от друга, не сваливая их все кучей в ветку master.

Для этого разработчики отправляют в репозиторий все свои бранчи по отдельности, не сливая их с master'ом. Сливанием занимаюсь я. Соответственно, в актуальную ветку попадают только проверенные мной правки. Плохие правки отправляются на доработку, а правки, ждущие утверждения заказчика, отправляются в тестовую ветку и просто… ждут. И никому не мешают.

Таким образом, актуальная ветка всегда находится в рабочем состоянии.

В принципе, отправку бранчей по отдельности можно производить и в одном репозитории. Но в одном репозитории нельзя разделить доступ к отдельным б т.е. нельзя разрешить писать в репозиторий новые бранчи и при этом запретить изменять актуальную ветку. Поэтому нужен второй репозиторий — в одном находится актуальная ветка, в другом — все новые рабочие ветки, появляющиеся по мере решения отдельных задач.

На самом деле, в больших проектах у каждого разработчика вообще должен быть свой отдельный репозиторий. Но мне пока вполне удобно живется с двум репозиториями — один мой (продакшн), другой — общий для всех разработчиков (разработка).

7. Первичная загрузка проекта в репозиторий

Обратите внимание — первичную загрузку делает ведущий разработчик, так как только у него есть право писать в главный репозиторий.

Создаем локальный репозиторий в каталоге проекта и загоняем в него файлы проекта:

```
$ cd /старый/каталог/проекта
$ git init
$ git add.
$ git commit -am 'poehali!'
```

Сообщаем локальному репозиторию о том, где находится главный репозиторий:

```
$ git remote add origin gituser@githost:project.git
```

Теперь оправляем коммит в главный репозиторий:

```
$ git push origin master
```

Все, проект отправлен в главный репозиторий. Теперь старый каталог проекта можно смело грохнуть и начать работу уже в новом каталоге (или вообще на компе), под управлением git'a.

8. Репозиторий ведущего разработчика

Переходим в новый каталог (или вообще на другой комп) и вытягиваем главный репозиторий:

```
$ cd /новый/каталог/проекта
$ git clone gituser@githost:project.git
```

Это мы получили копию главного репозитория. В ней находится актуальная ветка. Теперь нужно создать тестовую ветку.

Просто скопируем актуальную ветку в главный репозиторий под именем тестовой:

```
$ git push origin master:stage
```

А затем вытянем тестовую ветку из главного репозитория уже под собственным именем:

```
$ git checkout -b stage origin/stage
```

Теперь локальный репозиторий содержит две ветки, master и stage. Обе ветки связаны с одноименными ветками в главном репозитории.

В этом локальном репозитории ведущий разработчик будет проверять бранчи, присланные другими разработчиками, и сливать проверенны бранчи с веткам главного репозитория.

Кроме того, нужно указать местонахождение рабочего репозитория dev:

```
$ git remote add dev gituser@githost:dev.git
```

9. Инфраструктура проекта

Проект у нас включает не только репозитории, но и «исполняющие» узлы — продакшн и отладку.

Продакшн и отладка — суть локальные репозитории, обновляющиеся с главного репозитория. Разница между продакшном и отладкой заключается в том, продакшн обновляется с актуальной ветки, а отладка с тестовой.

Запуск продакшна, не вдаваясь в детали, был таким:

1) зарегистрировать в системе нового пользователя, из-под которого будет работать продакшн
2) добавить этого пользователя в gitosis (см. раздел 5, в моем примере это user*)
3) клонировать главный репозиторий project в каталог этого пользователя
4) Настроить виртуальный хост Apache (или что там вам больше нравится) на каталог проекта

И теперь для обновления продакшна достаточно просто зайти в этот каталог и выполнить одну-единственную команду:

```
$ git pull
```

Все обновления, находящиеся в актуальной ветке, во мгновение ока окажутся в продакшне.

Отладочная копия проекта устроена идентично, за исключением того, что после клонирования главного репозитория было сделано переключение с актуал ветки master на тестовую ветку stage:

$ git checkout -b stage origin/stage

Теперь pull в отладке будет вытягивать обновления из тестовой ветки.

10. Репозиторий разработчика

Приступая к работе над проектом, разработчик должен предварительно настроить свой локальный репозиторий. Разработчик будет иметь дело с двумя удаленными репозиториями — главным и рабочим.

Главный репозиторий нужно склонировать:

$ git clone gituser@githost:project.git

А рабочий репозиторий просто добавить в конфиг:

$ git remote add dev gituser@githost:dev.git

Эта настройка выполняется только один раз. Вся дальнейшая работа выполняется по стандартной схеме (распечатать памятку и прилепить на монитор).

11. Памятка разработчика

Проверить почту. Если пришло уведомление из Trac'а о новой задаче — начать работать:

$ quake exit

Создать новый бранч на основе актуальной ветки из главного репозитория (мы у себя договорились называть бранчи номерами тикетов из Trac'а):

$ git checkout -b new_branch origin/master

Или вернуться к работе над старым бранчем из рабочего репозитория

$ git checkout -b old_branch dev/old_branch

Вытянуть последние изменения:

$ git pull

Выполнить задачу в новом бранче. Тут происходит работа с кодом.

Если в процессе работы были созданы новые файлы — добавить их в бранч:

$ git add .

Сохранить изменения в локальном репозитории:

$ git commit -am 'komment'

Отправить новый бранч в рабочий репозиторий:

$ git push dev new_branch

Или повторно отправить туда же старый бранч:

$ git push

Выполненную задачу переназначить в Trac'е ведущему разработчику. Обратно разработчику она вернется либо с указанием на то, что нужно переделать, л
статусом closed.

10 goto проверить почту;

Тут надо сказать еще о паре тонкостей. Состояние локального репозитория каждого конкретного разработчика меня, в общем, не беспокоит, но все-таки, ч
люди не путались лишний раз, я рекомендую им делать две вещи.

Во-первых, бранчи, отправленные в рабочий репозиторий, более не нужны локально и могут быть смело удалены:

$ git branch -D new_branch

Во-вторых, список бранчей в удаленных репозиториях, выдаваемый командой

$ git branch -r

со временем устаревает, поскольку я удаляю оттуда проверенные и слитые с актуальной веткой бранчи. Чтобы обновить сведения об удаленных репозитор нужно выполнить команду

$ git remote prune dev

которая удалит из локального кеша отсутствующие в удаленном репозитории бранчи. Обратите внимание — обновлять сведения нужно только о рабочем репозитории. В главном репозитории никаких изменений никогда не происходит, там всегда находятся одни и те же бранчи — master и stage.

12. Действия ведущего разработчика

Получив уведомление из Trac'а о переназначенной мне задаче, я открываю эту задачу и смотрю, про что там вообще было, чтобы знать, что проверять.

Затем я обновляю данные об удаленных репозиториях:

$ git remote update

Теперь мне будут видны новые бранчи в рабочем репозитории, в том числе бранч, соответствующий проверяемой задаче:

$ git branch -r

Затем я вытягиваю проверяемый бранч из рабочего репозитория к себе (поскольку название бранча соответствует номеру тикета, то я всегда точно знаю, бранч вытягивать):

$ git checkout -b branch dev/branch

Если это не новый бранч, а исправление старого, то нужно еще вытянуть обновления:

$ git pull

Теперь я могу проверить работоспособность и правильность сделанной разработчиком правки.

Допустим, правка прошла проверку. Дальше действия зависят от того, является ли эта правка простой, такой, которую можно отправить в продакшн без согласования с заказчиком, или же это большая правка, которую надо согласовать.

Если правка простая, то я просто сливаю проверенный бранч с актуальной веткой и отправляю обновленную актуальную ветку в главный репозиторий:

$ git checkout master
$ git merge branch
$ git push

После этого я удаляю бранч у себя локально и из рабочего репозитория, больше этот бранч никому и никогда не понадобится:

$ git branch -D branch
$ git push dev :branch

Тут такой момент — сначала я хотел поручить удаление бранчей из рабочего репозитория разработчикам, чтобы каждый из них удалял свои бранчи. Но по решил не усложнять людям жизнь. Т.е. подход такой — если разработчик прислал бранч, который меня устроил, то больше этот бранч не должен разработ беспокоить.

Если же правка требует согласования, то проверяемый бранч сливается не с актуальной веткой, а с тестовой:

$ git checkout stage
$ git merge branch
$ git push

После отправки обновленной тестовой ветки в главный репозиторий я удаляю проверяемый бранч только у себя локально:

$ git branch -D branch

В рабочем же репозитории этот бранч остается. Вполне возможно, что заказчик захочет что-то переделать и тогда этот бранч понадобится разработчику д исправления.

В тестовой ветке проверяемый бранч лежит и есть не просит. Когда заказчик доберется до него и утвердит, тогда я снова вытяну этот бранч из рабочего репозитория и солью с актуальной веткой.

После того, как все бранчи слиты с нужными ветками и все это дело отправлено в главный репозиторий, я иду в продакшн и отладку и обновляю их.

(оригинал статьи)

git, gitosis, разработка, командная работа

+73 70,4k 512

Михаил Иванов @ivanych карма 8,0 рейтинг 0,0
Пользователь

ПОХОЖИЕ ПУБЛИКАЦИИ

19 декабря 2014 в 11:20
Лучший мессенджер для командной работы: Сравниваем HipChat, Slack и Kato
+4 86,1k 147 99

25 января 2013 в 14:43
Scrum — реальный опыт работы по методологии
+19 109k 253 49

11 июля 2012 в 10:51
Немного о командной работе
+3 3,9k 79 20

САМОЕ ЧИТАЕМОЕ Разраб

Сутки | Неделя | Месяц

А был ли взлом «Госуслуг»? Гипотеза Яндекса
+48 22,2k 28 55

Реверс-инжиниринг одной строчки JavaScript
+116 21,3k 133 17

Security Week 28: a Petya сложно открывался, в Android закрыли баг чипсета Broadcomm, Copycat заразил 14 млн девайсов
+14 11,3k 19 6

Дорога к C++20
+27 6,3k 22 13

IBM и ВВС США разрабатывают нейроморфный суперкомпьютер нового поколения
+14 8,3k 24 16

Комментарии (69)

HeadWithoutBrains 21 ноября 2009 в 23:47 #

Подскажите, как я понял, у вас для каждой правки отдельная ветка в репозитории? Или для например определенного задания?

ivanych 21 ноября 2009 в 23:58 # ↑

Одно задание — одна ветка. Я не различаю задания и правки. Может быть, переформулируете вопрос, я не совсем Вас понял.

remal 22 ноября 2009 в 00:40 #

Очень хорошая инструкция, только не написали где разработчикам хранить свои ключи.

ivanych 22 ноября 2009 в 00:57

Применяется стандартный способ авторизации SSH по ключам. Ключи, хранятся в домашних каталогах разработчиков. Я просто не стал вдаваться в де
чтобы совсем от темы не отходить.

Kolger 22 ноября 2009 в 02:22

А если задание совсем очень простое — как то поправить пару строчек — тоже делаете под это дело отдельные бранчи?
Я выполняю похожую на вашу роль руководителя, различие только в одном — репозиторий у нас один и работа с ветками устроена немного по другому:
master, как и у вас — стабилен. Туда сливаются только проверенные изменение, которых ждут на production'e. Если что то поменялось в мастере — значит
production'e пора пойти сделать pull.
Разработка же ведется в ветках с названиями dev<номер версии>, например, dev0.8, dev0.11, версии в моем случае — набор задач, который объединяется
версию. Когда цель достигнута — версия тестируется, сливается с мастером и выкладывается на продашкн.
В ситуации, если требуется какая то срочная правка мастера (багфикс) также создается временный бранч, который потом сливается с мастером.
Всякие экспериментальный фичи, которым не нашлось места в версии живет в именнованных бранчах.

Кстати, недавно отошел от трака в пользу redmine. Если вы ведете много проектов плюс очень большой — из коробки поддерживается много проектов, пра
настраиваются очень гибко. И видно все из одного места, не надо ползать по куче траков.

ivanych 22 ноября 2009 в 03:14

Тут существенное значение имеет то, на каком этапе разработки находится проект. Если говорить о самых первых этапах, то действительно, бывает сло
четко выделить конкретную задачу. Вроде надо сделать одну простую штуку, а начинаешь делать, оказывается, что сначала надо бы сделать другую шт
все это связано с третьей штукой и.т.д. В такой ситуации действительно имеет смысл вести разработку в «крупных» ветках, в которых отдельные правк
выделяются в отдельные бранчи.

Собственно, один из моих разработчиков сейчас так и работает. У него есть отдельная большая ветка, и я не проверяю у него каждый конкретный бранч
Просто время от времени мы совещаемся, смотрим, куда продвинулись, и сливаем его бранч с основной веткой.

Но большинство моих разработчиков работают над уже развитой частью проекта. Там любую задачу я могу четко формализовать. К тому же, любая пра
должна быть под чутким контролем. Поэтому основная часть работы ведется именно так, как описано в статье — отдельный бранч на каждую, пусть да
мелкую, задачу.

syndicut 22 ноября 2009 в 11:21

Да, Redmine очень удобен. Я даже не работая в команде использую его как хранилище данных о проекте и сам себе задачи назначаю =). Плюс очень удо
вещь там — учет времени на задачу.
Раньше тоже использовал Trac.

lasc 22 ноября 2009 в 03:08

а вебморды для gitosis не знаете? Чтобы пользователей и тд, добавлять через браузер

ivanych 22 ноября 2009 в 03:17

Нет, не возникало такой задачи. Но там же очень простой конфиг, так что я не заморачиваюсь, руками правлю.

akzhan 27 января 2013 в 20:33

Ставьте Gitolite и Gitlab.

НЛО прилетело и опубликовало эту надпись здесь

ivanych 22 ноября 2009 в 03:25

Эээ… Вы, простите, ерунду сказали. Файлы, которые должны быть изменены для работы на конкретной машине, вообще не должны быть под контролем

Скажем, логов в репозитории вообще не должно быть, а на конкретных машинах они должны быть проигнорированы путем создания файлов .gitignore. А
конфиги должны быть внесены в репозиторий в виде файлов .orig и на конкретных машинах должны копироваться в новые файлы, опять же, игнорируем
через .gitignore.

НЛО прилетело и опубликовало эту надпись здесь

ivanych 22 ноября 2009 в 04:04

Вот именно про файлы типа конфигов Вы и говорили. А я сказал, как нужно делать, чтобы эти файлы и в репозитории трекались, и локально прав
и куда не надо не коммитились.

Настройте все правильно и юзайте -am, вместо ручного перечисления полусотни измененных файлов.

smind 22 ноября 2009 в 13:47

откуда полсотни то возьмется? непонятно зачем писать так код, чтобы в коммит попадало так много изменений за раз, на мой взгляд это плоха
практика.

ivanych 22 ноября 2009 в 15:51

Ну как откуда? Это если задача «переименовать кнопку Сохранить в кнопку Записать», тогда конечно, в коммите будет одно изменение. А е
задача «исправить функцию расчета Итого в финансовых отчетах», то изменений будет ого-го.

Kolger 22 ноября 2009 в 04:25

Я обычно перед тем, как делать коммит делаю git status и вижу, что пойдет в коммит. Поэтому -am можно небояться.
А все конфиги должны быть в .gitignore. А их default версии в файликах .default.

smind 22 ноября 2009 в 13:45

От чего-ж ерунду, а мой взгляд -a совсем не лучший вариант, я тоже всегда перечисляю весь список файлов для коммита, на мой взгляд так гораздо не запихать ерунды в коммит.

ivanych 22 ноября 2009 в 15:54

Т.е. Вы предлагаете иметь ерунду в файлах, и постоянно волноваться, чтобы эту ерунду не отправить в репозиторий? Нет уж, спасибо, я лучше вос буду иметь ерунды.

ghisguth 22 ноября 2009 в 17:48

Ерундой может оказаться случайно измененный файл (ненужный пробел, перевод строки, ...), вывод отладочной информации (к примеру излиш подробный логгинг) и тому подобные изменения.
Лучше всё-таки просматривать код до коммита (git add -p, git gui). В крайнем случае — git status + git diff + git commit -a.

CWN 22 ноября 2009 в 09:26

чтобы в git commit не перечислять файлы, можно и нужно пользовать команду git add <маска_поиска_файлов>

После того как пометите все файлы которые идут в коммит (проверяете с помощью git status) использовать git commit -m «какой то коммит»

agladysh 22 ноября 2009 в 14:13

Я вот вообще hunk-ами стараюсь коммититься.

$ git add -p

рулит :-)

Коммитов больше, зато и контроля больше. Чётко видно, что, где и почему поменялось.

coodix 22 ноября 2009 в 10:34

Во-первых, спасибо за статью!

Еще было бы интересно узнать, используется ли в Вашей разработке какой-то Ваш движок, CMS, или фреймфорк и как происходит его параллельная разра вместе с работой над конкретным проектом (внешним).

Т.е. как правильно организовать схему репозиториев, чтобы во время разработки проекта для клиентов, можно было разрабатывать к примеру cms, при то изменения попадали в главную ветку cms, и сохранялись на пбудущее.

Такое полезно в небольших компаниях или даже фриланс-группах, где над внешними проектами и над внутренними (cms) работают одни и те же разработч

Очень много искал на эту тему но пока ничего дельного не придумалось.

ivanych 22 ноября 2009 в 15:58

У меня нет отдельно CMS и отдельно проекта/проектов на ней. Разрабатывается проект, и все функции, которые проекту требуются, сразу в него и встраиваются. Проект узкоспецифический, коробочная версия не предполагается.

allter 22 ноября 2009 в 22:24

Я предпочитаю разделять такие вещи по разным проектам и работа с ними независимо. Управлять слиянием конфигурации и движка надо извне (напри системой управления зависимостями платформы).

А так для меня есть 2 подхода для решения описанной вами задачи средствами только git: git submodule и git subtree. Первый плох тем, что недостаточно объединяет подмодуль и его контейнер: имея репозиторий с контейнером можно не иметь понятия о том, где брать подмодуль и наоборот. Второй плох т недостаточно устоялся и тем, что для выгрузки изменений из контейнера в репозиторий модуля надо делать промежуточное действие: выгрузку измене подкаталога внутри репозитория контейнера.

Sveolon 22 ноября 2009 в 10:43

Небольшую путаницу вносит то, что вы употребляете слова «ветка» и «бренч» в одном предложении в разных значениях. Мозг автоматически вопринимает слова как синонимы :)

А по сути — спасибо большое. Действительно, подобных статей я не встречал, разобрано именно то, что вегда упускается.

ivanych 22 ноября 2009 в 16:00

Ммм… Я на самом деле использую эти слова как синонимы. Если где-то получилось предложение, в котором эти слова означают разное — скажите, я постараюсь переформулировать.

Sveolon 22 ноября 2009 в 16:12

В тестовой ветке проверяемый бранч лежит и есть не просит. Когда заказчик доберется до него и утвердит, тогда я снова вы этот бранч из рабочего репозитория и солью с актуальной веткой."

ivanych 22 ноября 2009 в 16:32

Мда. Тут вместо слова «бранч» следовало бы употребить слово «коммит». Но тогда во всех остальных местах получается запутка… Я постараюсь переформулировать, спасибо за толковое замечание.

ghisguth 22 ноября 2009 в 13:03

Но в одном репозитории нельзя разделить доступ к отдельным бранчам, т.е. нельзя разрешить писать в репозиторий новые бранчи и при этом запретить из актуальную ветку.

можно. используя **pre-receive** хук:

```
#!/bin/sh
while read a; do
  branch=`echo $a | cut -d" " -f3`
  if [ $branch == "refs/heads/master" -o $branch == "refs/heads/stage" ];  then
    username=`whoami`
    if [ $username != 'superuser1' -a $username != 'superuser2' ];  then
      echo "* You not authorized to commit to $branch branch!!!        *" 1>&2
      exit 1
    fi
  fi
done
```

Также можно отослать себе email — с нотификацией, что кто-то хотел залиться в мастер.

GiV 22 ноября 2009 в 15:16

Когда проблему с доступом к бранчам курил, нашел продвинутый аналог gitosis — gitolite

Отличия от gitosis: github.com/sitaramc/gitolite/blob/pu/doc/3-faq-tips-etc.mkd#diff

ivanych 22 ноября 2009 в 16:06

Спасибо, надо будет почитать.

ivanych 22 ноября 2009 в 16:05

Да, можно решить хуками. Но я тут рассуждаю так:

По большому счету, у каждого разработчика должен быть свой собственный репозиторий. Ну, представьте себе, что все, кому не лень, будут коммитить репозиторий ядра Линукса. Это ж бардак будет.

Все, кому не лень, должны коммитить в свои репозитории, а затем уж предлагать управляющему главного репозитория принять их правки.

Поэтому я сразу разделяю главный репозиторий и репозитории отдельных разработчиков. Правда, я все-таки делаю упрощение и репозитории отдельны разработчиков объединяю в один.

ghisguth 22 ноября 2009 в 17:58

Согласен, я лишь хотел заметить, что запретить коммитить в отдельно взятый бранч в одном репозитории возможно.
А в ядре, насколько я знаю, в основном пулят, а не пушат изменения.
В принципе если работа происходит в офисе и у суперразработчика есть доступ на все машины по ssh — то можно пулить в свой репозиторий прямо с персональных репозиториев;)
Но если в комманде есть редиска, который ходит на работу с ноутбуком — то всё становится тяжелее;)

romel 22 ноября 2009 в 13:09

Спасибо, статья вразумительная, однако с git не знаком совершенно. Возник такой, скорее всего, глупый вопрос: где девелоперы пишут код? :-)

ivanych 22 ноября 2009 в 16:07

Девелоперы вытягивают копию главного репозитория к себе. В свой домашний каталог, или вообще на свой собственный комп. Куда угодно, где им удоб работать. Там и пишут код. А потом отправляют то, что написали, в рабочий репозиторий.

allter 22 ноября 2009 в 22:34

Хочу дополнить ivanych`а, что для разработки web-приложений очень помогают отдельные виртуальные машины с Linux: если что, то можно быстро дать готовую и настроенную среду новому разработчику.

smind 22 ноября 2009 в 13:39

спасибо за статью, через недельку расскажу как происходит командная разработка с использованием git в проекте Midnight Commander…

ksi 22 ноября 2009 в 14:49

Тоесть вы сами проверяете все изменения разработчиков? У вас нет отдела тестирования и continious integration? Я вот все думаю как заставить Hudson др такой кучей веток.

ivanych 22 ноября 2009 в 16:08

Нету у меня отдела тестирования…

(рыдает)

ghisguth 22 ноября 2009 в 18:10

Вот интересно как автор решает проблемы с бинарными файлами в репозитории (если они возникают конечно).
Поясню, что имею ввиду: хранить бинарники в vcs не очень хорошая идея, но иногда нет выхода. К примеру это может быть файл с ресурсами для флешки, картинки или схожие ресурсы у которых нету текстовых исходников.
Так вот, если разработка svn-like — то проблемы особо нет — ктото предупреждает остальных, что файл будет им изменен, меняет, заливает и освобождае ресурс. Но вот в случае если в мастер принимаются патчи и мёржатся одним человеком — пока он не смержит изменения — никто не может менять файл.
Я вижу пока только 2 варианта — не хранить файлы в vcs, но тогда дополнительно надо иметь доступ к хранилищу этих файлов (samba, NFS), а также у ни будет испстории. Или хранить в отдельном репозитории с svn-like подходом.

ivanych 22 ноября 2009 в 21:06

Промахнулся с ответом. Комментарий ниже.

mx2000 23 ноября 2009 в 05:40

Товарищ описывает ситуацию следующего характера:

1. Разработчик (дизайнер) A пуллит бинарный файл F1 (например, картинку)
2. Разработчик (дизайнер) B пуллит бинарный файл F1
3. Разработчик A коммитит измененный файл F1/A в ветку B1
4. Разработчик B коммитит измененный файл F1/B в ветку B2
5. Вам нужны оба изменения на продакшне.

Ваши действия?

ivanych 23 ноября 2009 в 09:34

Понял.

Не знаю, как тут быть. У меня как-то не возникает таких ситуаций, поэтому четкой позиции у меня по этому поводу нет. Как столкнусь — напишу д статью:)

ivanych 22 ноября 2009 в 21:05

Честно говоря, не понял сути описанной Вами проблемы.

У меня есть бинарные файлы. Вообще проект имеет, в основном, веб-интерфейс, но некоторая часть интерфейса, требовательная к графическим возможно реализована в виде исполняемого windows-приложения. Исходные коды приложения хранятся в репозитории, это понятно. Где хранить готовый дистрибути тоже долго думал и решил хранить там же, в репозитории. Смысл в том, что в этом случае pull на продакшне автоматически деплоит дистрибутив на серве которого дистрибутив уже могут скачать все пользователи.

Впрочем, тут рецепта дать не могу, у меня бинарные файлы меняются очень редко и проблем как-то возникает. Вероятно, есть методы хранения более тол

mkevac 23 ноября 2009 в 00:29 +

Спасибо за рассказ. Только вы немного не поняли смысл децентрализации репозиториев. По вашей схеме вы с таким же успехом могли бы использовать sv иже с ними.

Разница будет как минимум в том, что не девелоперы будут куда-то коммитить свой код, а вы будете pull-ить их код к себе в репозиторий.

mkevac 23 ноября 2009 в 00:33

Тут можно спорить, конечно. Во всяком случае большое спасибо за подробное описание того, как у вас происходит разработка. Таких статей действител нехватает. Плюс в карму :-)

ivanych 23 ноября 2009 в 00:48

Вы о чем? Что не так?

vitalyk 23 ноября 2009 в 00:37 +

хорошая статья, но у меня есть пару замечаний:

* «git branch -D» исползуется толко в исключительных случаях т.к. он позволяет потерять данные. в 99% случаев лучше использовать «git branch -d» — ета к вернёт ошибки при попытке удаления бранча который не «замерджен» в текущий.
* также из моего опыта если в команде работают толковые люди то настолько тотальный контроль не обязателен. у нас любой может «мерджить» в master
* под gitosis обычно используют юзер git а не gituser :)
* также предложенный порядок ведения бранчей отличается от «канонического» с точностью до наоборот :)
по идее «master» (или trunk) это место куда сливают всё самое последнее (и куда все могут мерджить),
а как раз для стабилизации используют staging и production бранчи.

ivanych 23 ноября 2009 в 01:08

* Нет, "-D" я указал осознанно. Именно с той целью, чтобы не отвлекаться на чтение уведомлений о «неродственности» текущего и удаляемого бранчей. Разработчики никогда не сливают свои бранчи с актуальным, им расхождение удаляемого бранча с актуальным роли не играет. А я всегда в итоге удаля присланный бранч, не зависимо от того, слил я его с актуальной веткой, или нет, поэтому мне сообщения от флага "-d" тоже не интересны.

* Я и говорю — тотал контроль не главное, главное — возможность отправлять правки в продакшн по одной, не дожидаясь одобрения заказчиком всего правок, попавшего в master.

* это я для того, чтобы понятно назвать githost. Согласитесь, название хоста host — не очень говорящее:)

* это Вы про другую схему рассказываете. Я именно отказался от такой схемы, когда все валят все в master. Моя схема более универсальна и позволяет хочется, все-таки валить все в master на рабочем репозитории.

 **Aquary** 23 ноября 2009 в 04:56

Ну вот, наконец-то внятная грамотная статья от поклонника git, где описан правильный процесс контроля версий в рамках командной разработки. А то как статья — так одни восторги, эпитеты и простое перечисление команд из мануала. Здесь же показан пример грамотной CM-политики для команды. Причем, политики, а не разрозненых сведений.

Одним словом, респект.

В моей SCM-копилке есть несколько ссылок на политики ведения групповой разработки — как с использованием классических подходов, так и agile-метод привязки к инструментам). Теперь вот есть куда отдельно указать любителям DVCS.

 **СТраННое** 24 ноября 2009 в 09:34

спасибо, хорошая статья

 **apollonin** 27 января 2013 в 00:49

Хоть статья уже и старая, но всё же рискну поднять вопрос.
В данный момент мы у себя в проекте используем очень похожую схему разработки. Но проблема в том, что у нас очень много программистов и процесс проведения веток на актуальную версию (продакшн) занимает очень много времени.
Может быть есть какие-то системы автоматизации мержа многих веток на мастер, пуш в мастер, а потом пулл на мастере и пр?

 **ivanych** 27 января 2013 в 15:31

Средств подобной автоматизации не знаю, как-то не возникало такой необходимости. Но так вот сходу я бы сказал, что это задача не техническая, а организационная. Я бы выделил несколько ответственных программистов, которым поручил бы сборку крупных частей проекта из мелких веток. А потом крупные ветки сливал бы в мастер.

 **apollonin** 27 января 2013 в 15:35

Та вот как раз и проблема в том, что в этом случае люди будут заниматься рутинной и «хомячковой» работой по сливанию веток в мастер. Это занима много времени, которые можно было бы потратить на реализацию программистских задач.

 **ivanych** 27 января 2013 в 17:46

Но Вы не можете поступить иначе. Какая бы ни была автоматизация, но кто-то должен принять ответственность за принятие кода в продакшен. Э «хомячковая» работа, наоборот, это едва ли не более важная работа, чем собственно написание кода.

Если ответственному человеку приходится выполнять много однотипных действий, то их, конечно, следует автоматизировать. Но принимать реше все-равно кто-то должен.

 **apollonin** 27 января 2013 в 17:50

Да-да, я понимаю. Речь идёт о моменте, когда задача оттестирована и уже на 100% готова в продакшн. И нужно её довести как можно меньшим человекозатратами. Приходится, к примеру, 2 раза в день выводить по по 15-20 веток в продакшн. И вот проводящему это совсем не в кайф, тр столько времени.

 **ivanych** 27 января 2013 в 20:18

Каждая задача на 100% оттестирована? И возможность взаимных конфликтов всех 15-20 веток протестирована?

Тогда не очень понятна проблема. Даже если ответственный делает это руками, мне представляется несложным написать скрипт в пять стр который в качестве аргументов будет принимать имена веток (а имена веток я рекомендую называть номерами тикетов) и выполнять всю ру работу типа мержей и прочих пулов.

Если же тестированием у вас занимается автоматизированная система, то на выходе у нее уже должна быть готовая ветка для продакшена, всеми протестированными и слитыми ветками.

 **pesh1983** 19 июня 2015 в 11:19 *(комментарий был изменён)*

Пост порядком запылился) Но все же мне интересно. Вы пишете, что нужно 2 репо. Я сейчас работаю с проектом на bitbucket, и там можно настроить огран на коммиты в конкретную ветку. Например, сделать так, чтобы в эту ветку могли отправлять коммиты (и удалять ее тоже) только отдельные пользователи. знаю, это допилили сами разработчики bitbucket или такой функционал уже встроен в гит, но в этом случае держать 2 репо бесмысленно. Работа ведется с ветками так же, как у Вас с 2 репо.
Хотя, конечно, могу предположить, что это фича именно bitbucket, иначе бы вы не писали про 2 репо)

 **ivanych** 19 июня 2015 в 12:17

Пост был написан, когда Гитхаб еще только-только появился, а Гитлаб ещё вообще не появился. В тех условиях работать более чем с одним репозитори весьма заморочно, а настраивать один репозиторий для доступа было невозможно или тоже сложно. Поэтому работа с двумя репозиториями была неким компромиссом.

Сейчас, когда есть Гитхаб, Гитлаб и тот же Битбакет, описанная в статье работа с двумя репозиториями представляется морально устаревшей. Правиль путь сейчас — использование произвольного количества репозиториев, по количеству разработчиков.

Так что, да, пост порядком запылился:) Но спасибо, вы навели меня на мысль написать новую версию статьи, с учетом текущих реалий.

pesh1983 19 июня 2015 в 14:07

Не совсем понимаю, чем работа с отдельными репозиториями отличается от работы с отдельными ветками в одном репозитории, объясните пожалуй разницу. Единственный плюс (а в некоторых случаях скорее минус), который я вижу — изолирование разрабов друг от друга.

ivanych 19 июня 2015 в 14:21

> изолирование разрабов друг от друга.

Это и есть главная цель. Чтобы каждый разраб мог в своей песочнице творить всё, что угодно, не оказывая никакого влияния на других разрабов и более, на главный репозиторий (апстрим), из которого деплоится продакшен.

Есть и дополнительные плюшки — например, свой репозиторий можно утащить с собой в лес и там разрабатывать в тишине и медитации.

pesh1983 19 июня 2015 в 14:26

А как таким образом решается вопрос с код ревью или, например, совместной работой над каким-то кодом? Простой пример. Есть некая фича, достаточно массивная. Делим на отдельные части, даем каждому разрабу отдельную часть. Когда части закончены, нужно выполнить их подкл друг к другу. Как показывает опыт, очень часто бывает, что нужно допиливать напильником уже вместе, а не по отдельности. И вот тут может о пригодиться код другого разработчика. Если репо один, таких проблем не возникает. А как быть в этом случае, если у каждого разраба свой ре

ivanych 19 июня 2015 в 14:34

Происходит обмен кодом с другими репозиториями. В том и суть распределенной системы — не огородить всех неприступной стеной, а сдел сеть, с удобным доступом друг к другу.

Простите, я не готов в рамках комментария это описывать, это Вам лучше почитать в целом о Гите и пулл-реквестах.

pesh1983 19 июня 2015 в 14:35

Про пулл-реквесты я в курсе, только не знал, что их можно выполнять в другие репо. В целом, я понял, в чем профит от такого подхода, с

ivanych 19 июня 2015 в 14:25

Касательно веток — у каждого разраба в *своем* репозитории могут быть ветки с каким угодно названием. Если все разрабы будут работать в одно репозитории, неминуемо будет конфликт имен — Вася назвал ветку «check» и Петя тоже так назвал — и привет, иди ругайся, кто тут главный.

Разнесение по разным репозиториям — это, в частности, разнесение по разным пространствам имен.

pesh1983 19 июня 2015 в 14:28 *(комментарий был изменён)*

У нас такая проблема решается просто дописыванием в начале имени и фамилии разраба, например, вот так i_ivanov_название_ветки. Пока пр нет. Хотя проект небольшой, возможно поэтому особых проблем пока не испытываем.

ivanych 19 июня 2015 в 14:35

Это костыль, добавление префикса какбэ и намекает Вам, что нужны разные пространства имен:)

vitek 5 февраля 2017 в 10:25

Для этого как раз удобен подход, продвигаемый гитлабом, где ветка начинается с номера задачи — это и решает вопрос уникальности имён случае ветки на задачу каждая ветка и так будет называться по-разному), и добавляет простоту переключения на ветку вводом одного толь номера задачи и автодополнением по табу.

Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.

+10 3,5k 51 3

Инопланетная болтовня: на космической вечеринке разум будет слышно лучше всего GT

+23 7,6k 21 47

Астрономы открыли самую маленькую звезду за все время наблюдений GT

+25 8,5k 13 13