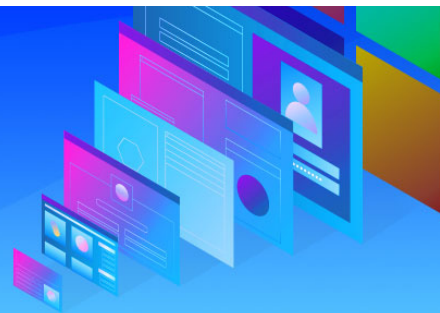


[КАК СТАТЬ АВТОРОМ](#)[Под холодный шёпот звёзд мы сделали опрос про Golang](#)

Лучшее предложение на рынке
VPS хостинг с Windows от **523** ₽/месяц

Лицензия на ОС включена в стоимость



2271.81
Рейтинг

RUVDS.com

VDS/VPS-хостинг. Скидка 10% по коду **HABR10**



ru_vds 12 января 2022 в 18:00

30 команд Git, необходимых для освоения интерфейса командной строки Git

Блог компании RUVDS.com, Системное администрирование*, Git*, GitHub*

[Перевод](#)

Автор оригинала: Tara Prasad Routray

**30 основных команд,
которые сделают
из вас
мастера Git**

TARA PRASAD ROUTRAY



Git — самая популярная в мире распределённая система контроля версий. Линус Торвальдс, разработчик ядра ОС Linux, создал этот инструмент ещё в 2005 году, а сегодня

Git активно поддерживается как проект с открытым исходным кодом. Огромное количество открытых и коммерческих проектов используют Git для контроля версий.

В данной статье перечисляются самые основные команды, которые следует знать разработчику, чтобы освоить управление репозиториями GitHub на высоком уровне. Ознакомиться с ними будет полезно как новичкам, так и опытным разработчикам.

30 основных команд, которые сделают из вас мастера Git

1. Как задать имя пользователя и адрес электронной почты
2. Кэширование учётных данных
3. Инициализация репозитория
4. Добавление отдельных файлов или всех файлов в область подготовленных файлов
5. Проверка статуса репозитория
6. Внесение изменений однострочным сообщением или через редактор
7. Просмотр истории коммитов с изменениями
8. Просмотр заданного коммита
9. Просмотр изменений до коммита
10. Удаление отслеживаемых файлов из текущего рабочего дерева
11. Переименование файлов
12. Отмена подготовленных и неподготовленных изменений
13. Изменение последнего коммита
14. Откат последнего коммита
15. Откат заданного коммита
16. Создание новой ветки и переход в неё
17. Просмотр списка веток
18. Удаление ветки
19. Слияние двух веток
20. Отображение журнала фиксации в виде графика для текущей или всех веток
21. Прекращение слияния при конфликте
22. Добавление удалённого репозитория
23. Просмотр удалённых URL-адресов
24. Получение дополнительных сведений об удалённом репозитории
25. Отправка изменений в удалённый репозиторий

26. Получение изменений из удалённого репозитория
27. Слияние удалённого репозитория с локальным
28. Отправка новой ветки в удалённый репозиторий
29. Удаление удалённой ветки
30. Использование перебазирования

1. Как задать имя пользователя и адрес электронной почты

Имя пользователя нужно, чтобы привязывать коммиты к вашему имени. Это не то же самое, что имя пользователя учётной записи GitHub, с помощью которого выполняется вход в профиль на GitHub. Задать или изменить имя пользователя можно с помощью команды `git config`. Новое имя будет автоматически отображаться в последующих коммитах, отправленных на GitHub через командную строку. Если хотите скрыть своё реальное имя, можно использовать в качестве имени пользователя Git произвольный набор символов.

```
git config --global user.name "Tara Routray"
```

Кроме того, командой `git config` можно изменять адрес электронной почты, привязанный к вашим коммитам Git. Новый адрес электронной почты будет автоматически отображаться во всех дальнейших коммитах, поданных на GitHub через командную строку.

```
git config --global user.email "dev@tararoutray.com"
```

2. Кэширование учётных данных

Кэшировать учётные данные можно с помощью параметра `config` с флагом `--global`. Так вы избавитесь от необходимости вручную вводить имя пользователя и пароль при создании нового коммита.

```
git config --global credential.helper cache
```

3. Инициализация репозитория

Создать пустой репозиторий Git или вновь инициализировать существующий можно параметром `init`. При инициализации он создаст скрытую папку. В ней содержатся все объекты и ссылки, которые Git использует и создаёт в истории работы над проектом.

```
git init
```

4. Добавление отдельных файлов или всех файлов в область подготовленных файлов

Добавить отдельный файл в область подготовленных файлов можно параметром `add` с указанием имени файла. Просто замените `somefile.js` на актуальное имя.

```
git add somefile.js
```

Кроме того, можно добавить все файлы и папки в эту область, предоставив wildcard `.` вместо имени файла:

```
git add .
```

5. Проверка статуса репозитория

Просмотреть статус нужного репозитория можно по ключевому слову `status`: его действие распространяется на подготовленные, неподготовленные и неотслеживаемые файлы.

```
git status
```

6. Внесение изменений однострочным сообщением или через редактор

При создании коммита в репозитории можно добавить однострочное сообщение с помощью параметра `commit` с флагом `-m`. Само сообщение вводится непосредственно

после флага, в кавычках.

```
git commit -m "Your short summary about the commit"
```

Также можно открыть текстовый редактор в терминале для написания полного сообщения коммита. Оно может состоять из нескольких строк текста, в котором подробно характеризуются изменения, внесённые в репозиторий.

```
git commit
```

7. Просмотр истории коммитов с изменениями

Просматривать изменения, внесённые в репозиторий, можно с помощью параметра `log`. Он отображает список последних коммитов в порядке выполнения. Кроме того, добавив флаг `-p`, вы можете подробно изучить изменения, внесённые в каждый файл.

```
git log -p
```

8. Просмотр заданного коммита

Просмотреть полный список изменений, внесённых конкретным коммитом, можно с помощью параметра `show`, указав идентификатор или хеш коммита. Значение хеша уникально для каждого коммита, созданного в вашем репозитории.

```
git show 1af17e73721dbe0c40011b82ed4bb1a7dbe3ce29
```

Также можно использовать сокращённый хеш.

```
git show 1af17e
```

9. Просмотр изменений до коммита

Можно просматривать список изменений, внесённых в репозиторий, используя параметр `diff`. По умолчанию отображаются только изменения, не подготовленные для фиксации.

```
git diff
```

Для просмотра подготовленных изменений необходимо добавить флаг `--staged`.

```
git diff --staged
```

Также можно указать имя файла как параметр и просмотреть изменения, внесённые только в этот файл.

```
git diff somefile.js
```

10. Удаление отслеживаемых файлов из текущего рабочего дерева

Удалять файлы из текущего рабочего дерева можно с помощью параметра `rm`. При этом файлы удаляются и из индекса.

```
git rm dirname/somefile.js
```

Можно также использовать маски файлов (например `*.js`) для удаления всех файлов, соответствующих критерию.

```
git rm dirname/*.html
```

11. Переименование файлов

Переименовать файл или папку можно параметром `mv`. Для него указывается источник `source` и назначение `destination`. Источник — реально существующий файл или папка, а назначение — существующая папка.

```
git mv dir1/somefile.js dir2
```

При выполнении команды файл или папка, указанные как источник, будут перемещены в папку назначения. Индекс будет обновлён соответственно, но изменения нужно записать.

12. Отмена подготовленных и неподготовленных изменений

Восстановить файлы рабочего дерева, не подготовленные к коммиту, можно параметром `checkout`. Для проведения операции требуется указать путь к файлу. Если путь не указан, параметр `git checkout` изменит указатель HEAD, чтобы задать указанную ветку как текущую.

```
git checkout somefile.js
```

Восстановить подготовленный файл рабочего дерева можно параметром `reset`. Потребуется указать путь к файлу, чтобы убрать его из области подготовленных файлов. При этом не будет производиться откат никаких изменений или модификаций — однако файл перейдёт в категорию не подготовленных к коммиту.

```
git reset HEAD somefile.js
```

Если нужно выполнить это действие для всех подготовленных файлов, путь к ним указывать не надо.

```
git reset HEAD
```

13. Изменение последнего коммита

Внести изменения в последний коммит можно параметром `commit` с флагом `--amend`. Например, вы записали изменения, внесённые в ряд файлов, и поняли, что допустили ошибку в сообщении коммита. В этом случае можете воспользоваться указанной командой, чтобы отредактировать сообщение предыдущего коммита, не изменяя его снимок.

```
git commit --amend -m "Updated message for the previous commit"
```

Также можно вносить изменения в файлы, отправленные ранее. Например, вы изменили несколько файлов в ряде папок и хотите их записать как единый снимок, но забыли добавить в коммит одну из папок. Чтобы исправить такую ошибку, достаточно подготовить для фиксации остальные файлы и папки и создать коммит с флагами `--amend` и `--no-edit`.

```
git add dir1
git commit

# Here you forgot to add dir2 to commit, you can execute the
following command to amend the other files and folders.

git add dir2
git commit --amend --no-edit
```

Флаг `--no-edit` позволит внести в коммит поправку без изменения сообщения коммита. В этом случае итоговый коммит заменит неполный, а выглядеть это будет так, как будто мы отправили изменения ко всем файлам в нужных папках как единый снимок.

Внимание! Не изменяйте публичные коммиты.

С помощью `amend` прекрасно исправляются локальные коммиты, а исправления можно передать в общий репозиторий. Однако изменять коммиты, уже доступные другим пользователям, не следует. Помните, что изменённые коммиты являются совершенно новыми, а предыдущий коммит уже не будет доступен в текущей ветке. Последствия будут такими же, как при отмене изменений публичного снимка.

14. Откат последнего коммита

Откатить последний коммит можно с помощью параметра `revert`. Создастся новый коммит, содержащий обратные преобразования относительно предыдущего, и добавится к истории текущей ветки.

```
git revert HEAD
```


Разница между revert и reset

Команда `git revert` отменяет изменения, записанные только одним коммитом. Она не откатывает проект к более раннему состоянию, удаляя все последующие коммиты, как это делает команда `git reset`.

У команды `revert` есть два крупных преимущества по сравнению с `reset`. Во-первых, она не меняет историю проекта и производит операцию, безопасную для коммитов. Во-вторых, её объектом выступает конкретный коммит, созданный в любой момент истории, а `git reset` всегда берёт за точку отсчёта текущий коммит. К примеру, если нужно отменить старый коммит с помощью `git reset`, придётся удалить все коммиты, поданные после целевого, а затем выполнить их повторно. Следовательно, команда `git revert` — гораздо более удобный и безопасный способ отмены изменений.

15. Откат заданного коммита

Откатить проект до заданного коммита можно с помощью параметра `revert` и идентификатора коммита. Создастся новый коммит — копия коммита с предоставленным идентификатором — и добавится к истории текущей ветки.

```
git revert 1af17e
```

16. Создание новой ветки и переход в неё

Создать новую ветку можно с помощью параметра `branch`, указав имя ветки.

```
git branch new_branch_name
```

Но Git не переключится на неё автоматически. Для автоматического перехода нужно добавить флаг `-b` и параметр `checkout`.

```
git checkout -b new_branch_name
```

17. Просмотр списка веток

Можно просматривать полный список веток, используя параметр `branch`. Команда отобразит все ветки, отметит текущую звёздочкой (*) и выделит её цветом.

```
git branch
```

Также можно вывести список удалённых веток с помощью флага `-a`.

```
git branch -a
```

18. Удаление ветки

Удалить ветку можно параметром `branch` с добавлением флага `-d` и указанием имени ветки. Если вы завершили работу над веткой и объединили её с основной, можно её удалить без потери истории. Однако, если выполнить команду удаления до слияния — в результате появится сообщение об ошибке. Этот защитный механизм предотвращает потерю доступа к файлам.

```
git branch -d existing_branch_name
```

Для принудительного удаления ветки используется флаг `-D` с заглавной буквой. В этом случае ветка будет удалена независимо от текущего статуса, без предупреждений.

```
git branch -D existing_branch_name
```

Вышеуказанные команды удаляют только локальную копию ветки. В удалённом репозитории она может сохраниться. Если хотите стереть удалённую ветку, выполните следующую команду:

```
git push origin --delete existing_branch_name
```

19. Слияние двух веток

Объединить две ветки можно параметром `merge` с указанием имени ветки. Команда объединит указанную ветку с основной.

```
git merge existing_branch_name
```

Если надо выполнить коммит слияния, выполните команду `git merge` с флагом `--no-ff`.

```
git merge --no-ff existing_branch_name
```

Указанная команда объединит заданную ветку с основной и произведёт коммит слияния. Это необходимо для фиксации всех слияний в вашем репозитории.

20. Отображение журнала фиксации в виде графика для текущей или всех веток

Просмотреть историю коммитов в виде графика для текущей ветки можно с помощью параметра `log` и флагов `--graph` `--oneline` `--decorate`. Опция `--graph` выведет график в формате ASCII, отражающий структуру ветвления истории коммитов. В связке с флагами `--oneline` и `--decorate`, этот флаг упрощает понимание того, к какой ветке относится каждый коммит.

```
git log --graph --oneline --decorate
```

Для просмотра истории коммитов по всем веткам используется флаг `--all`.

```
git log --all --graph --oneline --decorate
```

21. Прекращение слияния при конфликте

Прервать слияние в случае конфликта можно параметром `merge` с флагом `--abort`. Он позволяет остановить процесс слияния и вернуть состояние, с которого этот процесс был начат.

```
git merge --abort
```

Также при конфликте слияния можно использовать параметр `reset`, чтобы восстановить конфликтующие файлы до стабильного состояния.

```
git reset
```

22. Добавление удалённого репозитория

Добавить удалённый репозиторий можно параметром `remote add`, указав `shortname` и `url` требуемого репозитория.

```
git remote add awesomeapp https://github.com/someurl..
```

23. Просмотр удалённых URL-адресов

Просматривать удалённые URL-адреса можно параметром `remote` с флагом `-v`. Этот параметр отображает удалённые подключения к другим репозиториям.

```
git remote -v
```

Такая команда открывает доступ к интерфейсу управления удалёнными записями, которые хранятся в файле `.git/config` репозитория.

24. Получение дополнительных сведений об удалённом репозитории

Получить подробные сведения об удалённом репозитории можно с помощью параметра `remote show` с указанием имени репозитория — например, `origin`.

```
git remote show origin
```

Эта команда отображает список веток, связанных с удалённым репозиторием, а также

рабочих станций, подключённых для получения и отправки файлов.

25. Отправка изменений в удалённый репозиторий

Отправлять изменения в удалённый репозиторий можно параметром `push` с указанием имени репозитория и ветки.

```
git push origin main
```

Эта команда передаёт локальные изменения в центральный репозиторий, где с ними могут ознакомиться другие участники проекта.

26. Получение изменений из удалённого репозитория

Для загрузки изменений из удалённого репозитория используется параметр `pull`. Он скачивает копию текущей ветки с указанного удалённого репозитория и объединяет её с локальной копией.

```
git pull
```

Также можно просмотреть подробные сведения о загруженных файлах с помощью флага `--verbose`.

```
git pull --verbose
```

27. Слияние удалённого репозитория с локальным

Слияние удалённого репозитория с локальным выполняется параметром `merge` с указанием имени удалённого репозитория.

```
git merge origin
```

28. Отправка новой ветки в удалённый репозиторий

Передать новую ветку в удалённый репозиторий можно параметром `push` с флагом `-u`, указав имя репозитория и имя ветки.

```
git push -u origin new_branch
```

29. Удаление удалённой ветки

Чтобы избавиться от удалённой ветки, используйте параметр `push` с флагом `--delete`, указав имя удалённого репозитория и имя ветки.

```
git push --delete origin existing_branch
```

30. Использование перебазирования

Для доступа к этой функции используйте параметр `rebase` с указанием имени ветки. Перебазирование — это процесс объединения или перемещения последовательности коммитов на новый родительский снимок.

```
git rebase branch_name
```

Эта команда изменит основу ветки с одного коммита на другой, как если бы вы начали ветку с другого коммита. В Git это достигается за счёт создания новых коммитов и применения их к указанному базовому коммиту. Необходимо понимать, что, хотя ветка и выглядит такой же, она состоит из совершенно новых коммитов.

■ Спасибо за внимание!

Вы узнали 30 основных команд интерфейса командной строки Git. Теперь, при условии регулярной практики, у вас есть всё необходимое, чтобы достичь мастерства в управлении репозиториями GitHub.

Habrahabr10

Промокод для скидки 10% на виртуальные серверы RUVDS

Теги: ruvds_перевод, git, github, linux

Хабы: Блог компании RUVDS.com, Системное администрирование, Git, GitHub

◆ +54

👁 186K

📖 737

💬 62 +62



Редакторский дайджест



Присылаем лучшие статьи раз в месяц

Электронпочта



RUVDS.com

VDS/VPS-хостинг. Скидка 10% по коду **HABR10**

Telegram ВКонтакте Twitter



302

474.3

Карма

Рейтинг

@ru_vds

Пользователь

Комментарии 62



👤 **FDsagizi**
12.01.2022 в 19:17

Не подскажите, есть ли альтернатива ГИТ, чтобы у папок были права доступа ? И удобно для разработчиков игр.

◆ +1

Ответить



👤 **mayorovp**
12.01.2022 в 19:30

Если доступ нужно только на запись регулировать — можете побить основной репозиторий на сабмодули.

◆ 0

Ответить



👤 **Worky**
12.01.2022 в 19:32

Gitea

**-1**

Ответить

**zorn-v**

12.01.2022 в 22:35



чтобы у папок были права доступа ?

Нет такого. Делайте мультирепу.

**+1**

Ответить



НЛО прилетело и опубликовало эту надпись здесь

**zorn-v**

17.01.2022 в 01:47



Try free ?

Т.е. ваш код у непонятных левых типов ?

**0**

Ответить



НЛО прилетело и опубликовало эту надпись здесь

**idimus**

13.01.2022 в 18:15



Разработчики игр на больших проектах используют платный Helix (от компании Perforce). Он оптимизирован как раз для больших команд, где в моно репозитории хранится всё. Но там несколько другая философия, чем у гита.

**0**

Ответить

**byko3y**

13.01.2022 в 19:19



SVN

**+2**

Ответить



НЛО прилетело и опубликовало эту надпись здесь

**toxa82**

12.01.2022 в 20:07



Пункт 15 про revert неверный. Revert делает коммит с изменениями обратными только указанному коммиту, а не всему от головы до этого коммита. И если делать реверт мерж-коммиту, то нужно еще указать родителя в параметре -m (обычно это -m1).

Откат до заданного коммита это `git reset --hard`, но учтите что это потеря данных и изменение истории которое можно запустить только с параметром `--force`, а если вы с репозиторием работаете не один, то им тоже нужно будет провести некоторые манипуляции чтоб удалить эти коммиты тоже (и вполне возможно они вас за это возненавидят).

Пункт 30 немного объясню что делает команда: Например вы создали несколько коммитов в своей ветке `branch`. Находясь в своей ветке и выполнив команду `git rebase master`, гит находит общего предка веток `branch` и `master`, убирает ваши коммиты до общего коммита, обновляет вашу ветку `branch` до состояния `master`, и после этого применяет ваши коммиты сверху. Т.е. вы обновляете свою ветку относительно мастера и ваши коммиты будут сверху. Общий результат (в файлах) будет такой же как и если намержить ветку `master` на вашу, разница будет только в истории коммитов. При `merge` ваши коммиты будут до мержа с мастером, при `rebase` ваши коммиты будут сверху.

◆ +16 Ответить



○  **GaDzik**
12.01.2022 в 21:55

Спасибо. Как раз хотел повторить что то в этом духе).

◆ +1 Ответить



○  **Artem_zin**
12.01.2022 в 22:06

>2. Кэширование учётных данных

Кэшировать учётные данные можно с помощью параметра `config` с флагом `--global`. Так вы избавитесь от необходимости вручную вводить имя пользователя и пароль при создании нового коммита.

Пароль от чего? При коммите не нужно вводить пароль, разве что при криптоподписи коммита, что в основном не делают.

Учетные данные могут потребоваться для операций с `remote` репозиторием (`push/pull/fetch`), но их лучше делать с SSH ключем, в таком случае за кэширование его пароля отвечают всякие `ssh-agent` или средства ОС.

Этот пункт выдает делитанское отношение с гитом у автора оригинала статьи или я чего-то не понимаю.

>27. Слияние удалённого репозитория с локальным

Слияние удалённого репозитория с локальным выполняется параметром `merge` с указанием имени удалённого репозитория.

Тут стоит написать, что слияние будет с локальным слепком удаленного репозитория и перед этим стоит сделать ``git fetch remote-name`` чтобы этот слепок обновить,

>28. Отправка новой ветки в удалённый репозиторий

Я бы упомянул крайне полезную настройку стратегии наименования и трекинга веток при push, чтобы не указывать каждый раз название ветки, а запустить и создать такую же как текущая. ``git config push.default current`` и затем можно делать просто ``git push``, удаленная ветка создастся.

 +4 Ответить

**MishaRash**

14.01.2022 в 04:15



Согласен с большинством замечаний по поводу кэширования учётных данных. Но SSH-ключи поддерживаются не везде, например, в overleaf через git доступ только через логин-пароль.

 0 Ответить

**checkpoint**

13.01.2022 в 01:20



Ктонибудь, напишите в конце концов толковую статью про git, БЕЗ перечня команд.

В интернете есть миллион и одна "инструкция" по командам git, но нет почти ни одной статьи об идеологии с описанием того, что такое commit, branch, merge, pull request, на кой хрен всё это нужно и как этим пользоваться. Под выражением "как этим пользоваться" я НЕ имею в виду конкретную команду (её можно легко найти man-е или при выводе help-a). Я имею в виду то, какие принципы работы приследуются, в какой последовательности, что такое конфликт и как его разрешать. Короче, одни сухие перепечатки и выжимки из мануала.

Это я к чему. Вот, скажем, появился у нас в команде новый трудяга который ранее не то, что про git, вообще не слышал про репозитории и системы ведения версий, не понимает всей серьезности этого действия (да, таких все еще массово выпускают наши ВУЗы). Кому-то из команды придется потратить не один час на лекцию про git и все что с ним связано, совместно проделать несколько частых операций, показать как разрешать конфликты и т.д. И только после всего этого выдать ссылку на шпаргалку типа этой статьи. Я понимаю, что живое общение в команде - залог успеха, но опытному разработчику порой очень сложно структурировать весь объем знаний по какой-то тематике, в итоге лекция превращается в сумбур, новый работяга выходит с неё с выпученными глазами и с еще более запутанным представлением о мире. Некоторые - с заявлением на увольнение. ;-)

 +4 Ответить

**sakontwist**

13.01.2022 в 02:00



Ну вот же <https://git-scm.com/book/ru/v2>

 +4 Ответить

**checkpoint**

13.01.2022 в 08:41



Про книгу Git Pro я конечно в курсе, но не знал что есть перевод на русский, спасибо. Надо бы из неё какую-то выжимку сделать.



0

Ответить

**deely**

13.01.2022 в 14:09



Мне очень-очень помог вот этот курс уроков. Все просто, бесплатно, и по шагам.

<https://www.atlassian.com/git/tutorials>

Или же на русском:

<https://www.atlassian.com/ru/git/tutorials>



0

Ответить

**akazakou**

13.01.2022 в 01:28

Забыли "git rebase -i commitid" для объединения нескольких коммитов в 1. Очень удобно для людей с недержанием "git commit" :)



0

Ответить

**horror_x**

13.01.2022 в 02:20



Вообще хорошая практика причёсывать свои бранчи перед мержем. Мусорные коммиты никому не нужны, особенно всякие «fix» или «WIP» — ну никому не интересно как ты при разработке фичи свои же опечатки и баги правил или функциональность до рабочего состояния допиливал.

Ещё есть порочная практика — подмерживать изменения из апстримов в свои бранчи вместо rebase. Потом смотришь на эту паутину в истории и глаза начинают кровоточить. Ничем не лучше мусорных коммитов, а то и хуже.

Печально, что за чистотой истории мало кто следит на практике. Или из крайности в крайность бросаются — объединяют все коммиты в один при мерже, например.



0

Ответить

**Layan**

13.01.2022 в 20:29



У нас для сохранения чистой истории используется Squash Commit в master из ветки, где работал разработчик. Получается довольно удобно, с чистой историей. И не нужно обучать каждого джуна тому, как нужно структурировать коммиты внутри своей ветки. Главное правильно описать название и описание к pull request, т.к. они и будут текстом финального коммита.



+2

Ответить



**horror_x**

13.01.2022 в 21:03

Объединять все изменения в один коммит это другая крайность. Половина преимуществ использования Git теряется — ни найти конкретное изменение, ни точно откатить, ни черри-пикнуть. Вы же по сути отказались от возможностей merge.

Не понимаю, в чём тут удобство, оперировать изменениями размера «реализация фичи». Тем более, что у разных фич разные масштабы. Да и многие инструменты позволяют оперировать на уровне pull/merge request'ов, если уж надо именно фичами ворочать.

Я всегда стараюсь по крайней мере при коммитах производить декомпозицию задачи, коммитить отдельными подзадачами, чтобы изменения были максимально атомарны. При этом хорошей практикой считаю по возможности сохранять работоспособность в каждом коммите. Никогда заранее не знаешь, к какой точке и по какому поводу придётся вернуться.

**+2**

Ответить

**Layan**

13.01.2022 в 21:44

Дело у том, что отдельная ветка — полностью законченный функционал. И когда она завершена и протестирована — откат может быть только полный этого функционала.

В случае с большими фичами — это уже релизная ветка, в которую точно также будут сливаться мелкие фичи, и она уже будет вливаться в основную ветку через merge.

Я тоже при разработке мелких opensource пакетов в личном GitHub стараюсь максимально разбивать коммиты, даже если делал все за один раз, а в конце делаю кучу коммитов.

Однако, при работе в команде гораздо удобнее показал себя вариант с именно таким разбиением на фичи. Даже если в будущем потребуется посмотреть, как фича реализовывалась — достаточно посмотреть в веб интерфейсе нужный Pull Request, история там будет сохранена.

**0**

Ответить

**horror_x**

13.01.2022 в 22:01

И когда она завершена и протестирована — откат может быть только полный этого функционала.

Это весьма ограниченный опыт, да и правки как правило вносятся и в уже существующий код. Возможно, такова специфика конкретно вашего процесса, но в общем случае на практике встречается и множество других сценариев.

Например, переиспользование конкретных решений, которые являются лишь частью какой-то фичи. Или точечный откат одного или нескольких изменений фичи. Не говоря уж о постоянной необходимости просматривать конкретные изменения и понимать,

почему сделано именно так. В случае обобщённого коммита эта информация просто теряется, придётся каждый раз тратить время и вникать более детально.

Даже если в будущем потребуется посмотреть, как фича реализовывалась — достаточно посмотреть в веб интерфейсе нужный Pull Request, история там будет сохранена.

Ну т.е. ровно то, что предоставляет нативный механизм при merge. Тут есть минус хотя бы в том, что нельзя использовать привычные конкретному разработчику инструменты, включая сам Git. Да и вопрос в том, хранит ли реально этот веб-интерфейс историю или через пару недель она канет в лету после очередного git gc.



+1

Ответить

**noodles**

15.01.2022 в 15:54



Печально, что за чистотой истории мало кто следит на практике.

Всегда интересовал этот культ чистой истории коммитов. На практике. Чем чистая история лучше нечистой истории с кучей fix/merge коммитов? Какие реальные случаи.. на практике.. были, чтоб вот чистая история как-то реально помогла.

То что там всё чисто, и понятна история развития проекта.. это ясно. Я тоже за всё хорошее, и против всего плохого. Но на практике.. есть ли кому какое дело до чистой истории коммитов?

"Потом смотришь на эту паутину в истории и глаза начинают кровоточить."

Так можно просто не смотреть в эту историю. Зачем туда смотреть?)

Git - как источник истины и как хранилище - ок. Но как чтение истории проекта - зачем? кому оно надо? Пойди почитай релиз-ноутсы если на то пошло)

В небольших командах и так понятно кто какой кусок делает - и где чей залёт можно выяснить простым совещанием. В огромных командах, мне кажется, тем более всем плевать на историю проекта в контексте чтения коммитов в git-е.

Я без наездов, просто действительно культ чистой истории коммитов мне непонятен.. как-будто её кто-то читает.. ейбогу.)

История - это то что уже прошло, здесь и сейчас она не нужна, искать виноватых смысла нет, двигайте проект вперёд)



+2

Ответить

**horror_x**

15.01.2022 в 19:18



Какие реальные случаи... на практике... были, чтоб вот чистая история как-то реально помогла.

Вы когда-нибудь пользовались `git blame`? С мусорными коммитами часто невозможно сразу определить смысл конкретного изменения без последующего углублённого анализа истории или самого кода.

Делали cherry-pick конкретных изменений в ветке? С мусорными коммитами вам придётся анализировать ещё и каждый из них, чтобы захватить с собой и все последующие фиксы. А потом с большой вероятностью откатывать изменения, не относящиеся к нужной функциональности (скорей всего в виде конфликтов).

Переделывали часть функциональности какого-нибудь компонента? Если относящиеся к этому коммиты разбавлены мусором, придётся гораздо глубже изучать код и потратить гораздо больше времени, чтобы самостоятельно выделить все необходимые места, которые можно было бы определить по истории.

Откатывали какие-нибудь конкретные изменения из ветки? При мусоре приходится буквально хирургией заниматься, выделяя только нужное. Иногда для этого тоже нужно глубоко вникнуть в код, вплоть до полного понимания вообще всех принципов работы компонента. Особенно это «полезно», если с ним приходится работать в первый и последний раз.

Ссылались на какой-то конкретный опыт реализации чего-то или заимствовали готовые решения? С мусором будьте готовы потратить дополнительное время на поиск коммита, где нужный код в нужном вам состоянии.

И ещё множество сценариев, которые **заранее не предугадаешь**.

Но как чтение истории проекта — зачем? кому оно надо?

VCS даёт множество преимуществ при работе с кодом, почему бы ими не пользоваться? Я могу оперировать не только строками кода, но и наборами изменений, объединённых общим смыслом. Постоянно приходится что-то подсматривать, делать diff, изучать путь какого-то изменения, заимствовать, анализировать, искать. Да полно сценариев, у меня в IDE история постоянно открыта наряду с кодом.



Ответить



buldezir

13.01.2022 в 10:38




на гитхабе и гитлабе при мерже пулл-реквеста эта штука есть как опция (squash), так что можно самому заранее ничего не делать, наоборот лучше чтобы в ПРе была видна история разработки фичи.

так что штука полезная, но по факту руками ее никто никогда не делает.



Ответить



 **niyaho8778**
13.01.2022 в 11:04

```
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/
git push -u origin main
```

Почему github предлагает сначала сделать комит , потом создать новую ветку , а потом запустить ? Не логичней разве сначала создать ветку а потом начать комитить?

 0 Ответить



 **psydvl**
13.01.2022 в 14:54

-М не создает новую ветку, а переименовывает текущую (Move, двигает с предыдущего имени на новое), причем игнорируя то что main уже может существовать

Лучше использовать -m

 0 Ответить



 **Borz**
13.01.2022 в 18:34

так git только инициировали - там нет ещё веток даже кроме текущей

 0 Ответить



 **psydvl**
13.01.2022 в 22:17

Ничего подобного: ветка сразу создается с именем master или тем что указано в `init.defaultBranch` вот в мануале

Плюс сейчас только что выполнил git init

```
$ git init
hint: Using 'master' as the name for the initial branch. This default branch
name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint: git config --global init.defaultBranch
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
```

```
hint: git branch -m
Initialized empty Git repository in /home/nixi/work/Projects/test/.git/

$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

0 Ответить

 **Borz**
13.01.2022 в 23:04

правильно, после init есть только одна ветка - текущая. И там "main уже может существовать" не может быть, если не был использован defaultBranch=main

0 Ответить

 **Borz**
13.01.2022 в 18:33

по умолчанию создаётся ветка master (не хотят пока в main менять default branch) и в `git branch -M main` предлагается переименовать её в main

но это можно сразу двумя другими способами:

1. Сразу при инициировании git: `git init -b main`
2. Прописать глобально: `git config --global init.defaultBranch main` и потом уже вызывать `git init`

0 Ответить

 **Konvergent**
13.01.2022 в 13:22

Благодарю за хорошую шпоргалку.

Предлагаю поправить checkout на switch. Чтобы актуальность статьи посвежее была.

+1 Ответить

 **serjJS**
13.01.2022 в 13:38

После нескольких упрекааний в свой адрес, что забыл, мол пробелы убрать где не нужно. Начал пользоваться следующим алгоритмом.

1. Перед коммитом пишу git diff и проверяю где есть изменения (еще подойдет git status)
2. Потом с помощью команды git restore "file name" возвращаю файл к исходному значению

Таким образом коммичу только то что должно попасть в реквест. Избегая лишних пробелов и т.п в других файлах

 0 Ответить **slonopotamus**
13.01.2022 в 17:14

Но зачем вы лишние пробелы вообще говоря пишете?

 0 Ответить **psydvl**
13.01.2022 в 14:59 

Две самые важные команды забыли:)

`git gui` - позволяет работать с текущим рабочим деревом

`gitk` - позволяет работать с историей коммитов и сразу строит `git log --graph...`

 +1 Ответить **Vesh**
13.01.2022 в 20:09

`tig`, как `gitk`, но в терминале

 +1 Ответить **psydvl**
13.01.2022 в 22:20

`tig` отдельно ставить нужно, а `gitk` обычно сразу из коробки


 0 Ответить **static_cast**
13.01.2022 в 18:28

А по мне, важные пропущенные команды это

`git stash`

`git cherry-pick`

К тому же они очень полезны для новичков.

 +3 Ответить **prm_prg**
13.01.2022 в 20:09

А как же `git difftool -d`? Без нее ни одного коммита не делаю

 0 Ответить **byko3y**

 13.01.2022 в 21:15 

Ждем статьи "300 команд, которые сделают вас мастером bash". Восьмидесятые прошли, но для некоторых они остались в душе навсегда. Тулза, которая должна была быть незаметной, как калькулятор, стала идолом, к которому разработчики вынуждены обращаться мольбы по несколько раз в день. Согласитесь, что вы не найдете в интернете гайдов "как правильно передать параметры функции cos". Но как чекаутнуть произвольный коммит и, самое главное, как потом вернуть репозиторий в чистое состояние без необходимости удалять всю локальную копию и клонировать репу заново -- этим интернет завален. `git branch -D master; git checkout --track origin/master` -- о господи, это же так очевидно, почему я сразу не догадался?

 -1 Ответить   **mayorovp**
14.01.2022 в 03:47 

Но ведь... это и правда очевидно (хотя я бы написал `git checkout origin/master -B master`).

Это какое-то принципиальное нежелание понимать git или что?

 0 Ответить   **byko3y**
14.01.2022 в 04:51 

это и правда очевидно (хотя я бы написал `git checkout origin/master -B master`).

Что и требовалось доказать.

 -2 Ответить   **mayorovp**
14.01.2022 в 12:30 

Что именно требовалось доказать? Раскройте вашу мысль.

 0 Ответить   **byko3y**
14.01.2022 в 19:44 

Например, тот факт, что две заметно разные и при этом совсем не тривиальные команды дают один и тот же результат. Это если коротко. Если длинно, то я могу накатать целую статью. Как правило, короткого ответа хватает всем, кто хотя бы пару месяцев поработал с альтернативными VCS и знает, что ими можно пользоваться почти не читая документацию. Потому что VCS служит мне, а не я служу VCS. Интерфейс командной строки Git написан так, что нужно либо детально знать внутреннюю архитектуру кода и структур данных, либо заучить запредельное число готовых команд (сильно больше, чем упомянуто в этой статье), чтобы достаточно комфортно с Git работать. Благо, VS Code лично меня спасает от большей части базовых команд, но в сложных случаях все равно приходится лезть в консольку.



0

Ответить

**mayorovp**

14.01.2022 в 20:57



Но команды-то тривиальные и я бы не сказал что они какие-то заметно разные. Тут не надо знать внутреннюю структуру досконально, тут достаточно знать что "ветка" в гите является просто именованным ярлыком для номера коммита.

Кстати, а как эта задача решается в других VCS?



0

Ответить

**byko3y**

15.01.2022 в 05:31



Но команды-то тривиальные и я бы не сказал что они какие-то заметно разные.

`git checkout --` это как раз одна из самых безумных команд `git`, потому что одновременно применяется для подготовки незакоммиченных изменений в рабочем каталоге, для стирания изменений в рабочем каталоге, и для операций над ветками. Чтобы понять, как работает `checkout`, мне нужно понимать: что есть некий "индекс", который не коммит и не рабочий каталог, что есть "ветки", которые на самом деле не ветки, а головы ветки ("достаточно знать что "ветка" в гите является просто именованным ярлыком для номера коммита ", да), потому что ветки не могут прыгать по всему репозиторию и не могут схлопываться в один коммит; понимать, что такое "оторванная башка" и почему она оторванная.

Ты уже по крупницам из меня выгнываешь статью. Прямыми провальными решениями при создании `git` был именно выбор такого понятия "ветки" и применение индекса вместо реализации удобного интерфейса формирования коммита без внесения третьей сущности в контроль версий. Косвенными провальными решениями была ориентация на интенсивное ветвление, что в том числе усложнило опции команд работы с ветками, при том, что по итогу от этих веток все равно отказываются либо в пользу патчсетов и ревью, либо в пользу прямого формирования плоской истории изменений через какой-нибудь `git pull --rebase --autostash`` -- не так давно `git` получил эту фичу, которая в других VCS была из коробки изначально.



0

Ответить

**mayorovp**

15.01.2022 в 12:30



Вы не ответили на второй вопрос. Как эта задача решается в других VCS?

Интересует именно что аналог `git checkout origin/master -B master`

`git checkout` — это как раз одна из самых безумных команд `git`, потому что одновременно применяется для подготовки незакоммиченных изменений в рабочем каталоге, для стирания изменений в рабочем каталоге, и для операций над ветками.

git checkout применяется исключительно для обновления рабочей копии, все остальные применения — это побочные эффекты для удобства.

0 Ответить

 **byko3y**
15.01.2022 в 21:55

Вы не ответили на второй вопрос. Как эта задача решается в других VCS?

Интересует именно что аналог `git checkout origin/master -B master`

Это провокационный вопрос плана "какая же у вас шморгалка" -- эта команда уже жёстко протекает внутренностями git, которых, ожидаемо, просто нет в других VCS. В том же Mercurial нет такой замороченной системы веток с трекингом, потому аналогичной команды просто не существует, а существует акцент на плоской истории через грязную рабочую копию, расширение strip, или расширение очередей патчей.

В Mercurial есть даже слизанное с гита расширение rebase, но по факту малопопулярное потому, что rebase решает правильную задачу неправильным образом -- мне, как правило, никогда не нужно условно необратимо переместить ветку с одного места в другое, да еще и потенциально переписывая историю коммитов на публичной репе. Вместо этого есть чёткое деление на рабочие правки и рабочие патчи, не принадлежащие истории, и саму историю, которую публична и не подлежит изменению, которую я строго дописываю в конец на основе рабочих правок, и потому не выбью стул из-под ног человека, работающего с опубликованной веткой. Git по итогу пришел к тому же, но окольными путями и бессмысленными усложнениями операций с ветками.

git checkout применяется исключительно для обновления рабочей копии, все остальные применения — это побочные эффекты для удобства.

Удобства? Это как ковбои носили револьверы снятыми с предохранителя "для удобства", и стреляли себе в ногу? "Ну а что, вдруг кому-то понадобится срочно выстрелить себе в ногу?" -- так, что ли? Кажется, о чем-то таком думал Торвальдс, создавая изначальным прототип Git.

-1 Ответить

 **mayorovp**
17.01.2022 в 11:48

Но это же вы придумали специально для гита задачу, которая в других СКВ не возникает, и жалуетесь что есть аж целых два способа её решить.

Удобства? Это как ковбои носили револьверы снятыми с предохранителя "для удобства", и стреляли себе в ногу?

А почему создание ветки вы приравниваете к выстрелу в ногу?

 0 Ответить  **byko3y**
18.01.2022 в 10:19

Но это же вы придумали специально для гита задачу, которая в других СКВ не возникает, и жалуетесь что есть аж целых два способа её решить.

А Git научился в patch queue? Я понимаю, что сообщество уже напильило вокруг Git костылей под это дело, но можно было сделать сразу хорошо.

А почему создание ветки вы приравниваете к выстрелу в ногу?

Не создание ветки, а стирание изменение в рабочем каталоге, которое умеет делать checkout. Тот же checkout, который применяется для подготовки правок к коммиту в этом же рабочем каталоге.

 -1 Ответить  **mayorovp**
18.01.2022 в 12:50

Ну да, если указать конкретный файл, он будет перезаписан. А что ещё должна делать команда checkout?

Тот же checkout, который применяется для подготовки правок к коммиту в этом же рабочем каталоге.

Ни разу не применял checkout для подготовки правок к коммиту. Мы какие-то разные гиты обсуждаем?

 0 Ответить  **byko3y**
18.01.2022 в 14:08

Ну да, если указать конкретный файл, он будет перезаписан. А что ещё должна делать команда checkout?

Вот именно, одна и та же команда используется в роли "уничтожить локальные правки" и в роли "сохранить локальные правки после синхронизации". Но по факту это настолько опасно и заморочено, что я (и не только я) использую стэш и автостэш вместо какого-нибудь `git checkout --merge` или `git rebase`. Всё, что я до сих пор пытался сказать -- это что git взял за фундамент патологичный рабочий процесс, и только с годами и тысячами простреленных ног к гиту наконец прикостылили и даже внедрили в ядро рабочий процесс здорового человека, например, `git pull --rebase --autostash`.

Ни разу не применял checkout для подготовки правок к коммиту. Мы какие-то разные гиты обсуждаем?

Вот это самое "ни разу не применял" -- это одно из проявлений патологии, а именно -- избегание грязного рабочего каталога и коммиты каждые 5 минут. Что есть бессмысленно, потому что никого не волнуют эти незаконченные правки, они все равно будут сквошнуты и ребейзнуты -- именно последнее и есть настоящий рабочий процесс, а не плетение макаронного монстра в истории публичного репозитория.



-1

Ответить

**mayorovp**

18.01.2022 в 15:26



Вот это самое "ни разу не применял" — это одно из проявлений патологии, а именно — избегание грязного рабочего каталога и коммиты каждые 5 минут.

Не вижу связи.



0

Ответить

**byko3y**

19.01.2022 в 04:31



git checkout нужен на грязном рабочем каталоге для слияния локальных правок с последними правками в публичной репе. Примерно то, для чего делается "git pull --rebase --autostash", только для перехода к конкретной ревизии. То, для чего в SVN есть "svn update", а в Mercurial есть "hg update". Паталогичная же культура git заключается в том, что от разработчиков требуют держать все локальные правки в виде коммитов, из-за чего отпадает необходимость в "git checkout" на грязном рабочем каталоге, потому что рабочий каталог всегда чистый.

Но у чистого рабочего каталога есть своя цена -- теперь вместо того, чтобы применять локальные правки к какой угодно ревизии, я должен постоянно переписывать историю локальных веток, то есть, merge-rebase-squash-cherry pick, я должен упиваться кишками git вместо того, чтобы просто заниматься разработкой.



-1

Ответить

**mayorovp**

19.01.2022 в 12:40



Приведу аналогию: сохранения в редакторе. Вы сохраняетесь каждые полчаса (или вообще автоматически) или оставляете несохранённые изменения неделями? Знаю тех, кто использует последний вариант, были где-то рассказы на Хабре о чудесных никогда не вылетающих редакторах и сверхнадёжных ноутбуках — но это не означает, что привычка сохраняться паталогическая и обусловлена ненадёжными IDE.

Вот и с частыми коммитами в гите так же: это просто бэкап на случай ошибок рефакторинга. Но гит сам по себе их не требует.

И да, я всё ещё не вижу связи между практикой частых сохранений и неприменением `git checkout` для подготовки коммита.

То, для чего в SVN есть `"svn update"`, а в Mercurial есть `"hg update"`.

Вы по одному файлу обновляете в SVN и Mercurial, или всё-таки сразу всё дерево? Если первый вариант — то зачем? Если второй — то чем это от `checkout` отличается?



0

Ответить

**byko3y**

22.01.2022 в 10:09



Вот и с частыми коммитами в гите так же: это просто бэкап на случай ошибок рефакторинга. Но гит сам по себе их не требует.

Если я делаю какие-то крупномасштабные изменения, то, естественно, я сохраняю старую версию, с Git или без Git, разговор ведь не об этом совсем, такие события происходят нечасто, чтобы под них прогибать всю модель разработки.

И да, я всё ещё не вижу связи между практикой частых сохранений и неприменением `git checkout` для подготовки коммита.

Не частых сохранений, а частых коммитов. `checkout/update` нужен для переноса правок на другую ревизию, эдакий аналог `rebase`, но для незакоммиченных правок. Другой более близкий аналог в самом Git -- это `stash`.

>То, для чего в SVN есть `"svn update"`, а в Mercurial есть `"hg update"`.

Вы по одному файлу обновляете в SVN и Mercurial, или всё-таки сразу всё дерево? Если первый вариант — то зачем? Если второй — то чем это от `checkout` отличается?

Это и есть `checkout`, что я и пытался объяснить.



0

Ответить

**jt3k**

14.01.2022 в 02:59

Весьма посредственный перевод, в некоторых местах он не раскрывает тему а только запутывает неподготовленного читателя. Лучше бы собственную статью написали с упрощением понимания в виде примеров, а не демонстрировали слив гугл переводчику в виде искажений и многозначности донесения смыслов русским языком.

Вот пример:

Оригинал:

Second, it is able to target a specific commit at any point in the history, whereas git reset can only work backwards from the current commit.

Ваш перевод:

Во-вторых, её объектом выступает конкретный коммит, созданный в любой момент истории, а git reset всегда берёт за точку отсчёта текущий коммит.

Перевод гугла:

Во-вторых, он может нацеливаться на конкретный коммит в любой момент истории, тогда как git reset может работать только в обратном направлении от текущего коммита.

О каких объектах, какой точке отсчёта вы пишете?

Мне кажется из-за таких переводчиков наши локализаторы в кинопрокате выдают порой очень далёкие от оригинала названия фильмов, и даже переводы текста.

Я человек знакомый с темой поста, но если бы прочитал этот перевод до того как узнал о гите, многие места бы не понял.

Спасибо за внимание.



+2

Ответить



nickolaym

14.01.2022 в 05:07



Надёргали команд по своему вкусу. Возможно, это то, с чем сталкиваетесь именно вы в именно вашем рабочем процессе. Но это не "30 необходимых" для всех и каждого.

Где команды `clone`, `pull --ff`, `rebase --continue`, и прочая, и прочая?

Почему я эти привёл? - А почему вы привели те?

Короче, даже как шпаргалка - ваша статья на двоечку с плюсом.

А впрочем, понятно. Автор статьи - индус. Ему можно. А вот редактору, который выбирал, что переводить и что на хабр вываливать - незачёт.



+5

Ответить



varthon86

17.01.2022 в 11:27

Отличный ресурс для изучения и практики git: https://learngitbranching.js.org/?locale=ru_RU



0

Ответить



sergey-kuznetsov

26.05.2022 в 13:22

Куча вредных советов, а не статья. Новичкам эти заклинания ничем не помогут, а только запутают. Особенно с таким переводом.

0 Ответить



Только полноправные пользователи могут оставлять комментарии. [Войдите](#), пожалуйста.

Публикации

ЛУЧШИЕ ЗА СУТКИ ПОХОЖИЕ



MaFrance351 вчера в 19:01

Сам себе сотовый оператор

Тutorial

+206

24K

222

75 +75



engine9 вчера в 14:00

Электронный конструктор, не бьющий током

Тutorial

+93

9.2K

175

46 +46



Tolsedum вчера в 18:00

Доверяй, но проверяй

Мнение

+30

5.5K

38

6 +6



Ingirov сегодня в 02:27

Горе от совершенства: как избавиться от дурного перфекционизма. Личный опыт и лайфхаки

Из песочницы

 +29

 4.9K

 50

 8 +8




SLY_G вчера в 18:59

Новая эра астрономии: гигантские лазеры

Перевод

 +24

 3.2K

 12

 0

ИНФОРМАЦИЯ

Сайт	ruvds.com
Дата регистрации	18 марта 2016
Дата основания	27 июля 2015
Численность	11–30 человек
Местоположение	Россия
Представитель	ruvds

ССЫЛКИ

VPS / VDS сервер от 130 рублей в месяц.
ruvds.com

Дата-центры RUVDS в Москве, Санкт-Петербурге, Казани, Екатеринбурге, Новосибирске, Лондоне, Франкфурте, Цюрихе, Амстердаме
ruvds.com

Помощь и вопросы
ruvds.com

Партнерская программа RUVDS
ruvds.com

VPS (CPU 1x2ГГц, RAM 512Mb, SSD 10 Gb) — 190 рублей в месяц
ruvds.com

VPS Windows от 523 рублей в месяц. Бесплатный тестовый период 3 дня.
ruvds.com

VDS в Цюрихе. Дата-центр TIER III — швейцарское качество по низкой цене.
ruvds.com

Антивирусная защита виртуального сервера. Легкий агент для VPS.

ruvds.com

VPS в Лондоне. Дата-центр TIER III — английская точность за рубли.

ruvds.com

VPS с видеокартой на мощных серверах 3,4ГГц

ruvds.com

ПРИЛОЖЕНИЯ

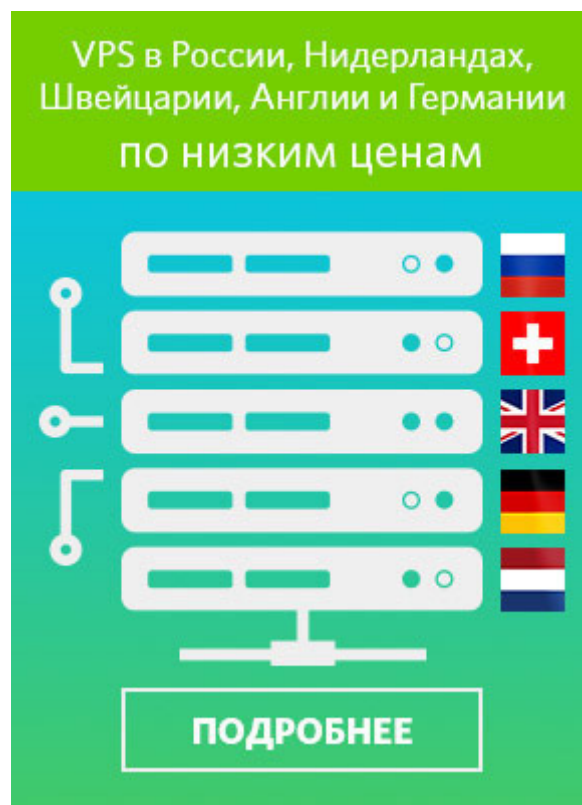


RUVDS Client

Приложение для мониторинга и управления виртуальными серверами RUVDS с мобильных устройств.

Android iOS

ВИДЖЕТ



БЛОГ НА ХАБРЕ

сегодня в 14:00

Почему форумы продолжают жить



107



0

вчера в 18:00

Доверяй, но проверяй

5.5K 6 +6

вчера в 14:00

Электронный конструктор, не бьющий током

9.2K 46 +46

9 января в 18:00

Создаём и настраиваем собственную CDN

5.9K 5 +5

9 января в 14:00

Консоль SSH на WebAssembly внутри браузера: как это сделано

6.9K 10 +10

Ваш аккаунт	Разделы	Информация	Услуги
Войти	Публикации	Устройство сайта	Корпоративный блог
Регистрация	Новости	Для авторов	Медийная реклама
	Хабы	Для компаний	Нативные проекты
	Компании	Документы	Образовательные
	Авторы	Соглашение	программы
	Песочница	Конфиденциальность	Стартапам
			Мегапроекты



Настройка языка

Техническая поддержка

[Вернуться на старую версию](#)

© 2006–2023, Habr