УДК 004.41

Использование продвинутых функций Git при разработке программного обеспечения

Хомоненко Анатолий Дмитриевич[1,2] — доктор технических наук, профессор, профессор кафедр «Информационные и вычислительные системы» и «Математическое и программное обеспечение». E-mail: khomon@mail.ru

Каратаев Евгений Николаевич[1] — магистрант 2-го курса направления 09.04.02 «Информационные системы и технологии». E-mail: zhenya-karat@yandex.ru

[1] Петербургский государственный университет путей сообщения Императора Александра I, Россия, Санкт-Петербург

[2] Военно-космическая академия имени А. Ф. Можайского, Россия, Санкт-Петербург

Для цитирования: Хомоненко А. Д., Каратаев Е. Н. Использование продвинутых функций Git при разработке программного обеспечения // Интеллектуальные технологии на транспорте. 2024. № 2 (38). С. 37–48. DOI: 10.20295/2413-2527-2024-238-37-48

Аннотация. *Рассматриваются ключевые аспекты использования продвинутых функций системы управления версиями Git при разработке программного обеспечения: модели ветвления, настройка git-конфигурации и методы исправления ошибок, полезные для разработчиков всех уровней.* ***Цель исследования:*** *демонстрация возможностей Git для повышения эффективности и гибкости разработки программного обеспечения. Рассмотрены вопросы интеграции продвинутых функций Git в процесс разработки программного обеспечения, представляющие интерес для профессионалов в области IT.* ***Практическая значимость:*** *обусловлена тем, что рассмотрены полезные параметры конфигурации, способствующие максимизации производительности и удобства использования Git. Приведены примеры команд и настроек системы Git, повышающие доступность и эффективность ее применения на практике.* ***Обсуждение:*** *через анализ технических и практических сторон показаны преимущества продвинутых функций Git и их вклад в улучшение процессов разработки. Показано, что использование продвинутых функций Git позволяет оптимизировать процессы разработки, повысить эффективность командных взаимодействий и обеспечить высокое качество кода.*

Ключевые слова: *Git, разработка программного обеспечения, управление версиями, Git-конфигурация, Git Flow, Trunk-Based Development.*

Введение

В современной разработке программного обеспечения системы управления версиями играют критическую роль, обеспечивая координацию работы команды, сохранность кода и возможность эффективного возвращения к предыдущим версиям продукта. Среди таких систем Git зарекомендовал себя как один из самых мощных и гибких инструментов в индустрии.

Цель настоящей статьи — исследовать и продемонстрировать, как в современном процессе разработки программного обеспечения применяются продвинутые возможности Git, которые могут быть использованы для повышения эффективности и гибкости в процессах разработки программного обеспечения.

Особое внимание уделяется различным моделям ветвления, в частности, Git Flow и Trunk-Based Development, которые предлагают структурированные подходы к организации работы с кодом,

способствуя повышению продуктивности и упрощению сотрудничества в командах разработчиков.

В дополнение в статье освещена значимость настройки Git-конфигурации и методологии, применяемые в этом процессе. Акцент сделан на том, как адекватно настроенные области видимости и конфигурационные файлы могут способствовать повышению эффективности использования системы. В заключительной части статьи рассмотрены полезные параметры конфигурации, способствующие максимизации производительности и удобства использования Git. Статья нацелена на разработчиков, стремящихся углубить свои знания и умения в работе с Git, раскрывая его потенциал в полной мере.

В статье показывается, как продвинутые функции Git интегрируется в современную разработку программного обеспечения, подчеркивая их роль в оптимизации процессов и повышении эффективности командных взаимодействий.

Обзор основных концепций Git

Git, разработанный Линусом Торвальдсом, является одной из наиболее широко используемых **систем контроля версий** в мире программирования. Эта система позволяет разработчикам отслеживать и управлять изменениями в коде, обеспечивая эффективную координацию работы в командах и сохранение истории проекта. В этом разделе подробно рассмотрены ключевые концепции Git, понимание которых необходимо для освоения продвинутых функций, описанных в последующих разделах [2].

Репозиторий — это основной элемент системы Git, представляющий собой центральное хранилище кода и его истории. Репозиторий содержит все версии файлов и информацию о их изменениях. В Git существуют локальные и удаленные репозитории, что позволяет разработчикам работать с кодом независимо, а затем синхронизировать изменения.

Коммиты — это «снимки» состояния репозитория в определенный момент времени. Каждый коммит содержит информацию об изменениях, авторе, дате и имеет уникальный идентификатор (hash). Через коммиты можно отслеживать историю изменений, возвращаться к предыдущим версиям и восстанавливать утерянные данные.

Ветвление и слияние. Ветвление позволяет разработчикам работать над разными задачами параллельно, не влияя на основной код проекта (master branch). Каждая ветка представляет собой отдельную линию разработки. Слияние (*merge*) — это процесс интеграции изменений из одной ветки в другую, позволяющий объединить разработки из разных веток.

Удаленные репозитории используются для обмена данными между участниками команды. Они служат централизованным хранилищем кода, к которому могут обращаться разработчики. Наиболее распространенным примером удаленного репозитория является GitHub. Команды `git push` и `git pull` позволяют синхронизировать локальные изменения с удаленным репозиторием.

Индексация и состояние файлов. Git отслеживает состояние файлов в репозитории, которые могут быть в трех состояниях: зафиксированные (*committed*), измененные (*modified*) и подготовленные к коммиту (*staged*). Индексация (*staging*) — это процесс добавления измененных файлов в промежуточное состояние перед фиксацией изменений коммитом.

Конфликты слияния возникают, когда изменения в одной ветке противоречат изменениям в другой. Git не может автоматически решить, какие изменения следует сохранить, поэтому требуется вмешательство разработчика для их разрешения [1–2].

Модели ветвления в Git

Модели ветвления в Git представляют собой структурированные подходы к управлению ветками, которые помогают координировать работу разработчиков, упорядочивать процесс интеграции изменений и поддерживать высокий уровень качества продукта. Без четко определенной модели ветвления команды разработчиков могут столкнуться с проблемами в управлении зависимостями, разрешении конфликтов и отслеживании изменений, что в конечном итоге может привести к снижению продуктивности и увеличению вероятности ошибок.

Существует несколько моделей ветвления, каждая из которых предлагает различные стратегии для управления изменениями и интеграции кода. Выбор конкретной модели зависит от множества факторов, включая размер команды, сложность проекта, частоту релизов и предпочтения команды. Примеры наиболее популярных моделей ветвления:

1. Git Flow: популярная модель, которая вводит строгую структуру ветвления, включая отдельные ветки для разработки, релизов, хотфиксов и функциональностей.

2. Trunk-Based Development: модель, акцентирующая внимание на короткоживущих ветках и частом слиянии с главной веткой (trunk), что способствует быстрой интеграции изменений и минимизации конфликтов.

3. Feature Branch Workflow: подход, при котором для каждой новой функции создается отдельная ветка, что позволяет изолированно разрабатывать и тестировать нововведения, прежде чем интегрировать их в основной код [4].

В статье рассматриваются две основные модели ветвления: Git Flow и Trunk-Based Development, так как Feature Branch Workflow схожа с моделью Trunk-Based Development.

- **Git Flow**

Git Flow выделяется как одна из наиболее структурированных и популярных моделей ветвления, разработанная и популяризированная Винсентом Дриссеном в 2010 году. Эта модель представляет собой точно определенный подход к ветвлению и слиянию, который способствует организации рабочего процесса разработки, особенно в командах среднего и большого размера.

Описание модели Git Flow

Основой Git Flow является деление рабочего процесса на несколько ключевых веток, каждая из которых служит определенной цели и имеет строгое назначение (рис. 1):

1. Основная ветка (master). В этой ветке содержится стабильная версия кода, готовая к выпуску. Изменения в эту ветку попадают только из ветки release или hotfix.

2. Ветка разработки (develop). Основная ветка для разработки, в которой происходит слияний изменений из различных веток функциональностей. После достижения определенного этапа развития проекта изменения из ветки develop могут быть перенесены в ветку release для последующего выпуска версии.

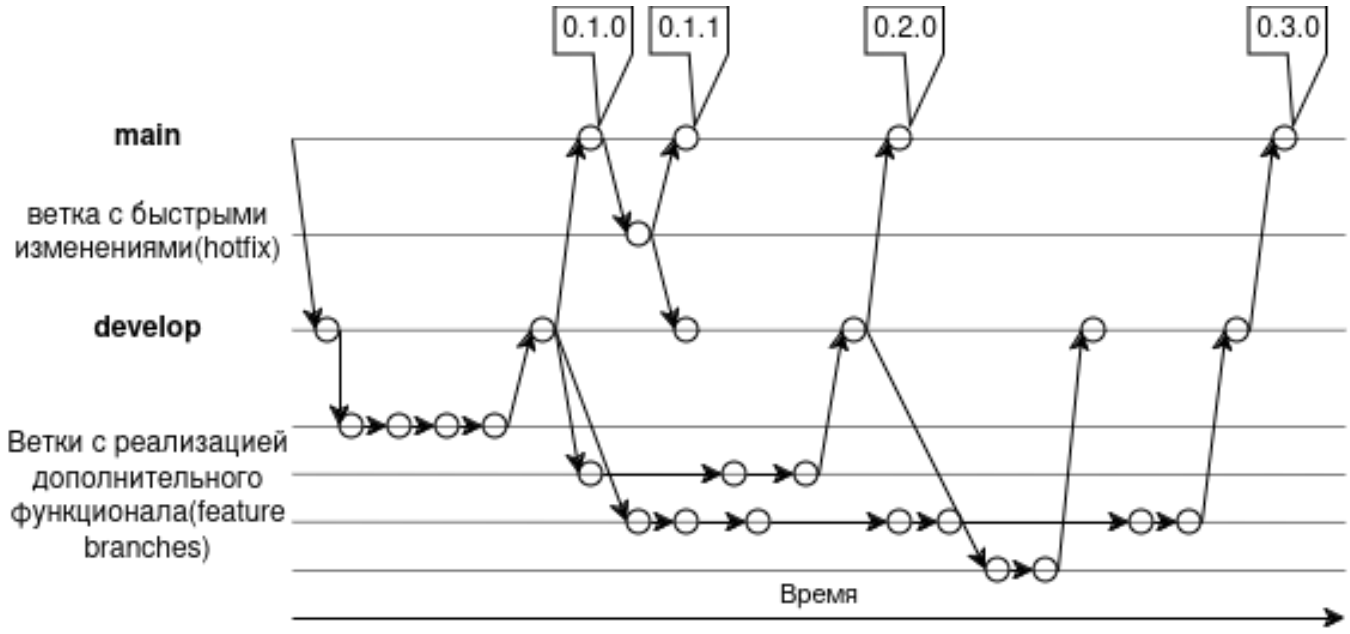

Рис. 1. Модель ветвления Git Flow

3. Ветки функциональностей (feature branches): Отдельные ветки, создаваемые для разработки новых функций или компонентов. После завершения работы функциональность интегрируется обратно в ветку develop.

4. Ветки выпуска (release branches). Эти ветки используются для подготовки новых версий продукта к выпуску. В них могут производиться последние исправления, документация и другие задачи, связанные с выпуском. После завершения изменения мерджатся как в master, так и в develop.

5. Ветки исправлений (hotfix branches). В случае обнаружения срочных ошибок в стабильной версии продукта создаются ветки hotfix, в которых происходит работа над исправлениями. После завершения исправлений, изменения интегрируются в master и develop, обеспечивая актуализацию кодовой базы [6].

Достоинства модели Git Flow

1. Параллельная разработка: возможность одновременной работы над различными функциями в отдельных ветках без влияния на основной код. Это ускоряет процесс разработки и позволяет разработчикам работать независимо.

2. Улучшенное тестирование: отделение новых разработок в отдельные ветки позволяет проводить тестирование изолированно от основного кода, что повышает качество тестирования и уменьшает вероятность ошибок в рабочей среде (production environment).

Недостатки модели Git Flow

1. Сложность.

Git Flow имеет относительно сложную структуру с множеством типов веток (feature, release, develop, hotfix, master), что может вызывать путаницу, особенно у новичков или в маленьких проектах, где такая структура может быть излишней.

2. Замедление разработки.

Постоянное переключение между ветками и необходимость соблюдения определенных правил могут замедлить процесс разработки, особенно в быстро меняющихся или маленьких проектах.

3. Избыточность.

Для маленьких или менее сложных проектов структура Git Flow может быть излишне сложной и тяжеловесной, что приводит к неоправданному увеличению работы с проектом [7].

Характеристика работы с моделью

Основные ветки. ***Master*** является основной веткой, в которой содержится стабильная версия кода, готовая к выпуску. Эта ветка служит конечной точкой для всех изменений, проходящих через процесс разработки и тестирования. ***Develop***, в свою очередь, представляет собой ветку для разработки, где собираются и интегрируются все изменения из вспомогательных веток, предварительно подготавливая их к переносу в основную ветку master.

Вспомогательные ветки (***feature branches***) создаются для разработки новых функциональностей. Они порождаются от ветки develop и после завершения работы над новой функцией сливаются обратно в нее. ***Release branches*** отвечают за подготовку новой версии продукта к выпуску. Они начинаются от ветки develop и позволяют вносить последние изменения перед слиянием в master и develop. ***Hotfix branches*** предназначены для оперативного устранения неполадок в уже выпущенных версиях. Создаются непосредственно от master и после внесения необходимых исправлений сливаются обратно в master и develop.

Процесс работы с Git Flow начинается с инициализации репозитория, что задает структуру веток. Далее в ходе разработки создаются ветки для новых функций (***feature***) от ***develop***, в которых и ведется вся работа. При подготовке к выпуску готовые и протестированные функции сливаются в ветку release, где происходит их окончательная доработка. После успешного тестирования и утверждения изменений ветка release интегрируется в master и помечается тегом с номером версии. В случае обнаружения срочных ошибок в произведенном релизе, применяются горячие исправления (***hotfix***), которые после устранения сливаются в master и develop, обеспечивая актуальность и стабильность кода [7].

- **Trunk-Based Development**

Trunk-Based Development (TBD) представляет собой подход к разработке программного обеспечения, который подчеркивает важность коротких циклов ветвления и слияния. Он ориентирован на упрощение процесса разработки путем минимизации продолжительности и сложности ветвлений, тем самым ускоряя слияние изменений и улучшая совместимость кода.

Описание TBD

Trunk-Based Development является методологией разработки программного обеспечения, при которой разработчики интегрируют свои изменения в одну центральную ветку в системе контроля версий, известную как trunk или main (рис. 2).

Основные принципы TBD включают следующие аспекты:

1. Централизация в одной ветке.

Все изменения напрямую интегрируются в главную ветку. Это снижает сложность управления множеством параллельных веток и уменьшает риск конфликтов при слиянии.

2. Частые коммиты.

Разработчики совершают коммиты в главную ветку как можно чаще, предпочтительно несколько раз в день. Это обеспечивает актуализацию кода у всех участников проекта и позволяет быстро выявлять и исправлять ошибки.

3. Минимальное время жизни веток.

Если создаются дополнительные ветки для специфических задач, их существование должно быть максимально коротким — от нескольких часов до пары дней. Эти ветки быстро интегрируются обратно в главную ветку, что поддерживает непрерывность и скорость разработки.

4. Контроль качества.

Несмотря на частоту коммитов, контроль качества не уступает по строгости. Автоматизированные тесты и проверки кода на предмет соответствия стандартам проводятся непрерывно, что гарантирует высокий уровень качества выпускаемого продукта.

5. Feature flags.

Для управления функционалом, который еще не готов к полноценному запуску, используются feature flags (флаги поведения или флаги функций). Это позволяет интегрировать код новых функций непосредственно в главную ветку, не ожидая их полного завершения, и активировать их в нужное время [8].

Достоинства и недостатки модели TBD

Ключевым достоинством модели Trunk-Based Development является повышение скорости внедрения новых возможностей и исправлений. Отказ от сложной ветвистой структуры позволяет команде быстрее реагировать на изменения требований, ускорять процесс непрерывной интеграции и развертывания.

К *достоинствам* этой модели ветвления можно отнести:

1. Ускоренный процесс разработки.

TBD позволяет разработчикам интегрировать изменения непосредственно в главную ветку, что минимизирует необходимость управления множеством параллельных веток. Это сокращает время для реализации нового функционала и уменьшает время выхода его в рабочую среду (production environment). Также уменьшение количества активных веток упрощает управление проектом.

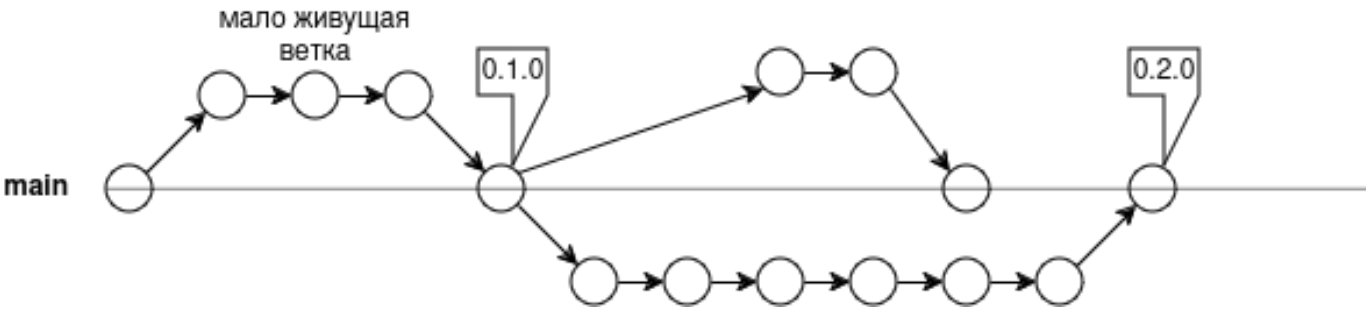

Рис. 2. Модель ветвления Trunk-Based Development

С малым количеством веток навигация и ориентирование в проекте становятся более интуитивными, особенно для новых участников команды.

2. Меньше конфликтов слияния.

Поскольку изменения вносятся часто и в малых объемах, вероятность возникновения сложных конфликтов слияния существенно уменьшается. Это способствует более простому процессу разработки и сокращает время, потраченное на разрешение конфликтов [9–11].

В то же время модель Trunk-Based Development имеет и некоторые *недостатки:*

1. Требования к дисциплине.

TBD требует от разработчиков высокой степени ответственности за вносимые изменения, поскольку любые ошибки сразу влияют на главную ветку. Это может увеличить количество проблем в рабочей ветке. Для эффективной работы с TBD необходима строгая система автоматизированного тестирования. Без надлежащего тестового покрытия, частое слияние изменений может привести к внедрению ошибок и нестабильности основного продукта.

2. Возможные проблемы с масштабированием.

В больших командах, где множество разработчиков одновременно работают над различными аспектами проекта, TBD может вызвать управленческие и технические трудности.

Процесс работы с моделью можно описать следующим образом:

1. Разработчики создают локальные ветки для реализации новых функций или исправления ошибок.

2. После завершения работы над изменениями, они производят слияние (***merge***) своей ветки в ***master***.

3. Автоматизированные системы сборки и тестирования проверяют интегрированные изменения на соответствие требованиям качества.

4. При успешном прохождении всех проверок, изменения считаются готовыми к релизу.

5. Новая версия продукта создается из среза ***master*** ветки [8].

Эффективность моделей ветвления

Внедрение моделей ветвления, таких как Git Flow и Trunk-Based Development, может повысить эффективность процессов разработки программного обеспечения. Структурированный подход к управлению ветками улучшают качества кода, ускоряет разработку новых версий продукта и упрощает взаимодействие в команде разработчиков.

Чтобы продемонстрировать преимущества использования моделей ветвления, проведен эксперимент в среднем проекте с командой из 10 разработчиков. В течение 6 месяцев команда не использовала модель ветвления. В следующие 6 месяцев, была внедрена модель Trunk-Based Development с коротким циклом ветвления.

В табл. 1 демонстрируются изменения в производительности и качестве работы команды до и после внедрения передовых практик ветвления.

Результаты эксперимента демонстрируют, что внедрение современных моделей ветвления, таких как Trunk-Based Development, повышает эффективность процессов разработки программного обеспечения. Структурированный подход к управлению ветками способствует ускорению выпуска новых версий продукта, повышению качества кода и улучшению сотрудничества в командах разработчиков.

Таблица 1

Результаты эксперимента

Параметр оценки	До введения	После введения	Изменение (%)
Время на разработку (дни)	45	38	-15.6%
Обнаруженные ошибки (шт.)	65	42	-35.4%
Время на исправление ошибок (часы)	120	84	-30.0%
Уровень сотрудничества (по шкале от 1 до 10)	5	8	+60.0%

Исправление действий в Git

В процессе разработки программного обеспечения с использованием распределенной системы управления версиями Git неизбежно возникают ситуации, когда разработчики совершают ошибочные действия. Эти ошибки могут негативно повлиять на процесс разработки и привести к нежелательным последствиям. Поэтому в статье рассмотрены наиболее распространенные ошибки, возникающих в процессе использования Git, и методов их решения.

Исправление ошибок в коммите

Еще одной из наиболее распространенных проблем, с которой сталкиваются пользователи Git, являются неправильные коммиты. Для решения данной задачи Git предлагает использовать команду `git commit --amend`, позволяющую вносить изменения в последний коммит без модификации его уникального хеша. Данная функциональность обеспечивает возможность быстрого исправления опечаток, неточностей в сообщении коммита, а также добавления упущенных файлов. При выполнении команды `git commit --amendd` автоматически открывается текстовый редактор, в котором разработчик может отредактировать содержимое последнего коммита, сохраняя при этом линейность истории репозитория [2, 12, 13].

Кроме того, в Git существует команда `git reset HEAD~1 --soft`, которая также может быть полезна при исправлении ошибок другого типа. Данная команда позволяет вернуть изменения в рабочую область, не затрагивая при этом историю коммитов. Это дает возможность разработчикам удалить нежелательные файлы или изменения, которые случайно попали в коммит, а затем создать новый, исправленный коммит [2, 12].

Исправление ошибок в ветке

Одним из распространенных сценариев является случайное внесение изменений в ветку master, вместо создания новой ветки для разработки функционала. В подобных ситуациях можно воспользоваться командой `git reset`, позволяющей откатить последние коммиты. Для решение данной проблемы нужно выполнить последовательно данные команды:

1. `Git branch future-brunch`. Создает новую ветку с названием future-brunch на основе текущей активной ветки, но без переключения на нее.

2. `Git reset HEAD~ --hard`. Сбрасывает состояние рабочего каталога и зону отслеживаемых изменений (staging area) к состоянию предпоследнего коммита в активной ветке.

3. `Git checkout future-brunch` переключает активную ветку на future-brunch [2, 12, 14].

Соответственно, все изменения перешли в ветку `future-brunch`, а в ветке `master` этих изменений нет. Также, если нужно откатить состояние на определенное количество коммитов назад, то перед `HEAD~` поставить число, указывающее, на сколько коммитов нужно вернуться. Например, для отката на три коммита назад используйте команду:

4. `Git reset HEAD~3 --hard`.

Другой распространенной ошибкой является некорректное название создаваемой ветки. В таком случае можно воспользоваться командой `git branch -m`, позволяющей переименовать существующую ветку. Например, команда `git branch -m future-brunch feature-branch` переименует ветку `future-brunch` в `feature-branch`. Если ветка уже была отправлена в удаленный репозиторий, потребуется также удалить старое название ветки с помощью `git push origin --delete future-brunch` и затем отправить переименованную ветку `git push origin feature-branch` [2].

Отмена действий в Git-репозитории

Когда после действий в Git-репозитории состояние стало еще хуже, чем было в начале. Для восстановления состояния используются такие инструменты, такие как `git reflog` и `git reset`.

`Git reflog` позволяет просмотреть журнал всех выполненных команд Git, включая перемещения между ветками, изменения указателя HEAD и другие операции. Используя эту информацию, разработчик может вернуться к любому предыдущему состоянию репозитория [2].

Используя эту информацию, разработчик может вернуться к любому предыдущему состоянию репозитория, воспользовавшись командой `git reset HEAD@{index}`, где `index` — это соответствующий индекс из вывода `git reflog` [2, 15].

Например, если в результате неверных действий в репозитории возникла нежелательная ситуация, разработчик может выполнить `git reflog`, чтобы увидеть историю изменений, и затем восстановить предыдущее, корректное состояние с помощью `git reset HEAD@{index}`. Данный подход позволяет эффективно «путешествовать во времени» и возвращаться к любым ранее существовавшим версиям проекта.

Следует отметить, что манипуляции с историей коммитов, такие как git reset, следует производить с особой осторожностью, поскольку они могут привести к потере данных, особенно если изменения уже были опубликованы в удаленном репозитории. В таких случаях рекомендуется использовать более безопасные методы, например, git revert, которые создают новый коммит, отменяющий предыдущие изменения [2].

Конфигурация Git

Несмотря на то что при первом знакомстве с системой контроля версий Git большинство пользователей ограничиваются только минимальной настройкой, необходимой для начала работы, углубленное понимание процесса конфигурации Git позволяет эффективно использовать данный инструмент в современной разработке.

Конфигурация Git представляет собой многогранный процесс, затрагивающий различные аспекты использования данной системы. От определения идентификационных данных пользователя до настройки стилей коммитов и интеграции со сторонними инструментами каждый из этих элементов вносит вклад в повышение удобства и эффективности работы разработчика.

Более того, возможность сохранения конфигурации Git в виде файлов способствует повышению переносимости среды разработки. Разработчики могут быстро восстановить свое рабочее окружение на новом компьютере или сервере, минимизируя затраты времени на повторную настройку.

Уровни конфигурации

Конфигурации Git могут быть заданы на трех уровнях: системном (system), глобальном (global) и локальном (local). Каждый уровень конфигурации имеет свою область действия и приоритет [1].

Системный уровень конфигурации затрагивает всех пользователей и все репозитории на данной машине. Настройки этого уровня применяются в случае, если не заданы конфигурации на глобальном или локальном уровнях. Для создания системной конфигурации используется команда `git config --system`. Файл системной конфигурации обычно находится по пути /etc/gitconfig.

Глобальный уровень конфигурации относится к конкретному пользователю и применяется ко всем репозиториям этого пользователя, если не переопределен на локальном уровне. Глобальные настройки удобны для задания персональных предпочтений, таких как имя пользователя, email и редактор по умолчанию. Для установки глобальной конфигурации используется команда `git config --global`. Файл глобальной конфигурации располагается в домашней директории пользователя: ~/.gitconfig или ~/.config/git/config.

Локальный уровень конфигурации имеет наивысший приоритет и применяется только к конкретному репозиторию. Эти настройки хранятся в файле .git/config внутри репозитория. Локальная конфигурация позволяет задать специфичные для репозитория параметры, такие как удаленные репозитории (remotes) и настройки ребейза. Для создания локальной конфигурации используется команда `git config --local` или просто `git config` без флагов, находясь внутри репозитория [16].

Приоритет уровней конфигурации следующий: локальный > глобальный > системный. То есть, если определенный параметр задан на нескольких уровнях, будет использовано значение с наивысшим приоритетом.

Файл конфигурации

Файл конфигурации представляет собой основное хранилище параметров, определяющих поведение Git во время выполнения различных операций. Формат конфигурационного файла основан на синтаксисе INI, состоящем из секций, обозначенных заголовками в квадратных скобках, и ключ-значных пар внутри этих секций. Пример:

```
[user]
    name = John Doe
    email = johndoe@example.com
[core]
    editor = vim
```

Типовые параметры конфигурации

Некоторые типовые параметры, которые можно установить в файле конфигурации Git [5]:

1. User.name и user.email

Эти параметры задают имя и email пользователя, которые будут использоваться при создании коммитов. Пример:

```
[user]
    name = John Doe
    email = johndoe@example.com
```

2. Core.Editor

Определяет текстовый редактор, который будет использоваться для редактирования сообщений коммитов и других текстовых данных. Пример:

```
[core]
    editor = vim
```

3. Credential.helper

Указывает способ хранения учетных данных для аутентификации при работе с удаленными репозиториями. `cache` указывает, что учетные данные сохраняются в памяти на определенный период времени (по умолчанию 15 минут). `store` указывает, что учетные данные сохраняются в файле на диске в открытом виде. `manager` указывает, что используется встроенный в операционную систему менеджер учетных данных. Также можно указать путь к внешней программе или скрипту, который будет отвечать за хранение и предоставление учетных данных. Пример:

```
[credential]
    helper = cache --timeout =3600
```

4. Core.autocrlf

Управляет автоматическим преобразованием символов окончания строк при чтении и записи файлов. При параметре `true` Git будет автоматически преобразовывать окончания строк из формата LF (Linux, macOS) в формат CRLF (Windows) при чтении файлов из репозитория и обратно при записи файлов в репозиторий. Это удобно для разработчиков, работающих в Windows. При параметре `false`: Git не будет выполнять никаких преобразований окончаний строк. Разработчики должны самостоятельно следить за согласованностью окончаний строк в файлах. При параметре `input`: Git будет преобразовывать окончания строк из формата CRLF в формат LF при записи файлов в репозиторий, но не будет выполнять преобразования при чтении файлов. Это удобно для разработчиков, работающих в Linux или macOS и сотрудничающих с разработчиками, использующими Windows. Пример:

```
[core]
    autocrlf = true
```

5. Core.whitespace

Настраивает обработку пробельных символов в файлах. Может включать опции для предупреждения или автоматического исправления нежелательных пробельных символов. Пример:

```
[core]
    whitespace = fix,-indent-with-
non-tab,trailing-space,cr-at-eol
```

Отметим, что локальные параметры переопределяют глобальные для конкретного репозитория. Многие GUI-клиенты Git также предоставляют интерфейсы для настройки этих параметров.

Заключение

Использование таких возможностей, как модели ветвления, исправление ошибок и настройка конфигурации, может значительно повысить эффективность и гибкость процессов разработки. Анализ моделей ветвления, таких как Git Flow и Trunk-Based Development, продемонстрировал их роль в структурировании работы с кодом и улучшении командного взаимодействия. Применение этих подходов позволяет оптимизировать процессы разработки и упростить сотрудничество между членами команды.

Правильное использование методов исправления ошибок в коммитах, ветках и отмены действий в git-репозитории помогает поддерживать целостность кода и обеспечивает возможность эффективного возврата к предыдущим версиям продукта. Адекватно настроенные области видимости и конфигурационные файлы способствуют повышению эффективности использования системы. Выделены полезные параметры конфигурации, которые могут значительно улучшить производительность и удобство работы с Git. Показано, что использование продвинутых функций Git играет важную роль в современной разработке программного обеспечения. Применение этих возможностей позволяет оптимизировать процессы разработки, повысить эффективность командных взаимодействий и обеспечить высокое качество кода.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Фишерман Л. В. Git. Практическое руководство. Управление и контроль версий в разработке программного обеспечения // Наука и техника, 2022. 304 с.

2. Chacon S., Straub B. Pro Git. Apress, 2014. Версия 2.1.104-2-g74b0d66, 06.09.2022. 538 с.

3. Shakikhanli U., Bilicki V. Optimizing branching strategies in mono and multi-repository environments: a comprehensive analysis // Computer Assisted Methods in Engineering and Science. 2024.

4. Гаспарян А. В., Тимошина Н. В. Совместная разработка по с использованием Git // ИТ-портал. 2017. № 1 (13) [Электронный ресурс]. URL: https://cyberleninka.ru/article/n/sovmestnaya-razrabotka-po-s-ispolzovaniem-git (дата обращения: 26.04.2024).

5. Вьюшкова М. В., Чернова С. В. Принцип работы системы контроля версий Git // Теория и практика современной науки. 2019. № 10 (52) [Электронный ресурс]. URL: https://cyberleninka.ru/article/n/printsip-raboty-sistemy-kontrolya-versiy-git (дата обращения: 26.04.2024).

6. Kummer D. Шпаргалка по git-flow. 2016 [Электронный ресурс]. URL: http://danielkummer.github.io/git-flow-cheatsheet/index.ru_RU.html (дата обращения: 26.04.2024).

7. Driessen V. A Successful git branching model. 2010 [Электронный ресурс]. URL: https://nvie.com/posts/a-successful-git-branching-model/ (дата обращения: 26.04.2024).

8. Trunk Based Development [Электронный ресурс]. URL: https://trunkbaseddevelopment.com/ (дата обращения: 26.04.2024).

9. Климентьев А. Пожалуйста, перестаньте рекомендовать Git Flow. 2020 [Электронный ресурс]. URL: https://habr.com/ru/companies/flant/articles/491320/ (дата обращения: 26.04.2024).

10. Александров А. Почему Trunk Based Development — лучшая модель ветвления. 2020 [Электронный ресурс]. URL: https://habr.com/ru/articles/519314/ (дата обращения: 26.04.2024).

11. Решетников С. Мой опыт перевода команд разработки на trunk-based development, 2024 [Электронный ресурс]. URL: https://habr.com/ru/articles/794246/ (дата обращения: 26.04.2024).

12. Beckham S. Git happens! 6 типичных ошибок Git и как их исправить. 2018 [Электронный ресурс]. URL: https://habr.com/ru/companies/flant/articles/419733/ (дата обращения: 26.04.2024).

13. Hexlet. Изменение последнего коммита [Электронный ресурс]. URL: https://ru.hexlet.io/courses/intro_to_git/lessons/git-commit-amend/theory_unit (дата обращения: 26.04.2024).

14. git scm. Git-git-branch Documentation. 2024 [Электронный ресурс]. URL: https://git-scm.com/docs/git-branch (дата обращения: 26.04.2024).

15. git scm. Git-git-reset Documentation. 2024 [Электронный ресурс]. URL: https://git-scm.com/docs/git-reset (дата обращения: 26.04.2024).

16. git scm. Git-git-config Documentation. 2024 [Электронный ресурс]. URL: https://git-scm.com/docs/git-config (дата обращения: 26.04.2024).

Дата поступления: 13.05.2024
Решение о публикации: 27.05.2024

Using Advanced Git Features in Software Development

Anatoly D. Khomonenko[1,2] — Doctor of Technical Sciences, Professor, Professor of the departments "Information and Computing Systems" and "Mathematical and Software". E-mail: khomon@mail.ru

Evgenij N. Karataev[1] — 2nd year undergraduate student of the direction 09.04.02 "Information systems and technologies". E-mail: zhenya-karat@yandex.ru

[1] Emperor Alexander I Petersburg State Transport University, Saint Petersburg, Russia
[2] A. F. Mozhaisky Military Space Academy, Saint Petersburg, Russia

For citation: Khomonenko A. D., Karataev E. N. Using advanced git features in software development // Intelligent technologies on transport. 2024. No. 2 (38). P. 37–48. (In Russian). DOI: 10.20295/2413-2527-2024-238-37-48

Abstract. *The key aspects of using the advanced features of the Git version control system in software development are considered: branching models, Git configuration settings and error correction methods useful for developers at all levels. The purpose of the study is to demonstrate the capabilities of Git to increase the efficiency and flexibility of software development. The issues of integrating advanced Git functions into the software development process, which are of interest to IT professionals, are considered. Practical significance: due to the fact that useful configuration parameters are considered that help maximize the performance and usability of Git. Examples of commands and settings of the Git system are given, which increase the accessibility and effectiveness of its application in practice. Discussion: through the analysis of technical and practical aspects, the advantages of advanced Git functions and their contribution to improving development processes are shown. It is shown that the use of advanced Git functions makes it possible to optimize development processes, increase the efficiency of team interactions and ensure high code quality.*

Keywords: *Git, software development, version control, Git configuration, Git Flow, Trunk-Based Development.*

REFERENCES

1. Fisherman L. V. Git. Prakticheskoe rukovodstvo. Upravlenie i kontrol' versij v razrabotke programmnogo obespechenija // Nauka i tehnika, 2022. 304 s. (In Russian)

2. Chacon S., Straub B. Pro Git. Apress, 2014. Versija 2.1.104-2-g74b0d66, 06.09.2022. 538 s.

3. Shakikhanli U., Bilicki V. Optimizing branching strategies in mono and multi-repository environments: a comprehensive analysis // Computer Assisted Methods in Engineering and Science. 2024.

4. Gasparjan A. V., Timoshina N. V. Sovmestnaja razrabotka po s ispol'zovaniem Git // IT-portal. 2017. № 1 (13) [Jelektronnyj resurs]. URL: https://cyberleninka.ru/article/n/sovmestnaya-razrabotka-po-s-ispolzovaniem-git (data obrashhenija: 26.04.2024). (In Russian)

5. V'jushkova M. V., Chernova S. V. Princip raboty sistemy kontrolja versij Git // Teorija i praktika sovremennoj nauki. 2019. № 10 (52) [Jelektronnyj resurs]. URL: https://cyberleninka.ru/article/n/printsip-raboty-sistemy-kontrolya-versiy-git (data obrashhenija: 26.04.2024). (In Russian)

6. Kummer D. Shpargalka po git-flow. 2016 [Jelektronnyj resurs]. URL: http://danielkummer.github.io/git-flow-cheatsheet/index.ru_RU.html (data obrashhenija: 26.04.2024). (In Russian)

7. Driessen V. A Successful git branching model. 2010 [Jelektronnyj resurs]. URL: https://nvie.com/posts/a-successful-git-branching-model/ (data obrashhenija: 26.04.2024).

8. Trunk Based Development [Jelektronnyj resurs]. URL: https://trunkbaseddevelopment.com/ (data obrashhenija:26.04.2024).

9. Kliment'ev A. Pozhalujsta, perestan'te rekomendovat' Git Flow. 2020 [Jelektronnyj resurs]. URL: https://habr.com/ru/companies/flant/articles/491320/ (data obrashhenija: 26.04.2024). (In Russian)

10. Aleksandrov A. Pochemu Trunk Based Development — luchshaja model' vetvlenija. 2020 [Jelektronnyj resurs]. URL: https://habr.com/ru/articles/519314/ (data obrashhenija: 26.04.2024). (In Russian)

11. Reshetnikov S. Moj opyt perevoda komand razrabotki na trunk-based development, 2024 [Jelektronnyj resurs]. URL: https://habr.com/ru/articles/794246/ (data obrashhenija: 26.04.2024). (In Russian)

12. Beckham S. Git happens! 6 tipichnyh oshibok Git i kak ih ispravit'. 2018 [Jelektronnyj resurs]. URL: https://habr.com/ru/companies/flant/articles/419733/ (data obrashhenija: 26.04.2024). (In Russian)

13. Hexlet. Izmenenie poslednego kommita [Jelektronnyj resurs]. URL: https://ru.hexlet.io/courses/intro_to_git/lessons/git-commit-amend/theory_unit (data obrashhenija: 26.04.2024). (In Russian)

14. git scm. Git-git-branch Documentation. 2024 [Jelektronnyj resurs]. URL: https://git-scm.com/docs/git-branch (data obrashhenija: 26.04.2024).

15. git scm. Git-git-reset Documentation. 2024 [Jelektronnyj resurs]. URL: https://git-scm.com/docs/git-reset (data obrashhenija: 26.04.2024).

16. git scm. Git-git-config Documentation. 2024 [Jelektronnyj resurs]. URL: https://git-scm.com/docs/git-config (data obrashhenija: 26.04.2024).

Received: 13.05.2024
Accepted: 27.05.2024