# Mercurial: Полное руководство

Bryan O'Sullivan

# Содержание

# Список иллюстраций

# Список таблиц

# Предисловие

## 1. Технические детали

Несколько лет назад, когда я хотел объяснить человеку, почему распределенное управление версиями важно, тема была настолько новая, что почти не было литературы, которую можно было бы порекомендовать.

Хотя тогда я работал над самим Mercurial, я решил переключиться на написание этой книги, потому как считал, что это самый эффективный способ помочь новому программному обеспечению покорить широкую аудиторию. Также это отличный способ донести идею о том, что управление версиями по своей природе должно быть распределенным. Я опубликовал книгу в Сети под свободной лицензией по той же причине: чтобы ввести в курс дела всех желающих.

Я создавал это руководство в привычном ключе хорошей книги о программном обеспечении, которая рассказывает: «Что это такое?», «Почему это важно?», «Как это поможет мне?» и «Как это использовать?». В этой книге я пытаюсь ответить на эти вопросы как для систем распределённого контроля версий в целом, так и для Mercurial в частности.

## 2. Спасибо за поддержку Mercurial

Приобретая экземпляр этой книги, вы поддерживаете постоянное развитие и свободность Mercurial в частности, а также свободного программного обеспечения с открытым исходным кодом в целом. Я и O'Reilly Media безвозмездно передаём мои гонорары от продажи этой книги в Software Freedom Conservancy (http://www.softwarefreedom.org/), которая предоставляет административную и юридическую поддержку Mercurial, а также ряд других заметных и значимых открытых проектов свободного программного обеспечения.

## 3. Благодарности

Этой книги не существовало бы, если бы не помощь Мэтта Мэккалла, автора и руководителя проекта Mercurial. Также мне компетентно помогали сотни добровольных участников из разных уголков мира.

Спасибо моим детям, Чиан и Руари, всегда готовым помочь мне расслабиться и замечательно поиграть в детские игры. Я также хотел бы поблагодарить мою бывшую жену Шеннен за ее поддержку.

Мои коллеги и друзья оказывали помощь и поддержку во многих отношениях. Этот список людей очень неполон: Стивен Ган, Керин Риттер, Бонни Корвин, Джеймс Василе, Мэтт Норвуд, Эбен Моглен, Брэдли Кан, Роберт Уолш, Джереми Фитцхардиндж, Рэйчел Челмерс.

Я писал эту книгу открыто, размещая предварительные варианты глав на веб-сайте книги по мере их готовности. Читатели затем оставляли комментарии, используя разработанный мной веб-интерфейс. К тому времени, как я закончил писать книгу, более 100 человек представили свои замечания — это огромное количество, особенно с учетом того, что система комментариев стала работать всего за 2 месяца до окончания процесса написания книги.

Особенно хотелось бы отметить следующих людей, которые обеспечили более трети от общего количества замечаний. Я хотел бы поблагодарить их за помощь и усилия в написании настолько подробных отзывов:

Martin Geisler, Damien Cassou, Alexey Bakhirkin, Till Plewe, Dan Himes, Paul Sargent, Gokberk Hamurcu, Matthijs van der Vleuten, Michael Chermside, John Mulligan, Jordi Fita, Jon Parise.

Я также хотел бы отметить помощь многих людей, которые замечали ошибки и давали полезные советы по содержанию всей книги:

Jeremy W. Sherman, Brian Mearns, Vincent Furia, Iwan Luijks, Billy Edwards, Andreas Sliwka, Paweł Sołyga, Eric Hanchrow, Steve Nicolai, Michał Masłowski, Kevin Fitch, Johan Holmberg, Hal Wine, Volker Simonis, Thomas P Jakobsen, Ted Stresen-Reuter, Stephen Rasku, Raphael Das Gupta, Ned Batchelder, Lou Keeble, Li Linxiao, Kao Cardoso Félix, Joseph Wecker, Jon Prescot, Jon Maken, John Yeary, Jason Harris, Geoffrey Zheng, Fredrik Jonson, Ed Davies, David Zumbrunnen, David Mercer, David Cabana, Ben Karel, Alan Franzoni, Yousry Abdallah, Whitney Young, Vinay Sajip, Tom Towle, Tim Ottinger, Thomas Schraitle, Tero Saarni, Ted Mielczarek, Svetoslav Agafonkin, Shaun Rowland, Rocco Rutte, Polo-Francois Poli, Philip Jenvey, Petr Tesałék, Peter R. Annema, Paul Bonser, Olivier Scherler, Olivier

Fournier, Nick Parker, Nick Fabry, Nicholas Guarracino, Mike Driscoll, Mike Coleman, Mietek Bák, Michael Maloney, László Nagy, Kent Johnson, Julio Nobrega, Jord Fita, Jonathan March, Jonas Nockert, Jim Tittsler, Jeduan Cornejo Legorreta, Jan Larres, James Murphy, Henri Wiechers, Hagen Möbius, Gábor Farkas, Fabien Engels, Evert Rol, Evan Willms, Eduardo Felipe Castegnaro, Dennis Decker Jensen, Deniz Dogan, David Smith, Daed Lee, Christine Slotty, Charles Merriam, Guillaume Catto, Brian Dorsey, Bob Nystrom, Benoit Boissinot, Avi Rosenschein, Andrew Watts, Andrew Donkin, Alexey Rodriguez, Ahmed Chaudhary.

# 4. Соглашения, принятые в этой книге

В книге используются следующие соглашения:

*Курсив*
: Указывает на новые термины, URL ресурсы, адреса электронной почты, имена файлов и файловых расширений.

`Моноширинный шрифт`
: Используется для листингов программ, а также внутри текста, ссылающегося на элементы программы, такие, как, например, переменные или имена функций, базы данных, типы данных, переменные окружения, операторы и ключевые слова.

**`Моноширинный полужирный шрифт`**
: Указывает на команды или другой текст, которые пользователю при наборе следует в точности скопировать с оригинала.

***`Моноширинный шрифт с курсивом`***
: Указывает на текст, который должен быть заменен на пользовательские значения или значения, определяемые по контексту.

### Подсказка

Этот значок означает подсказку, замечание или общую заметку.

### Внимание

Этот значок означает предупреждение или предостережение.

# 5. Использование примеров кода

Эта книга сделана, чтобы помочь вам делать свою работу. В целом, вы можете использовать код в этой книге в своих программах и документах. Вам не нужно обращаться к нам за разрешением, если вы не воспроизводите значительные части кода. Например, написание программы, которая использует несколько фрагментов кода из этой книги, не требует разрешения. Продажа или распространение примеров с CD-ROM, прилагающемуся к книге O'Reilly требует разрешения. Вы не должны спрашивать разрешения, если отвечаете на вопрос, приводя цитаты из книги или примеров кода. Включение значительного количества примеров кода из этой книги в документацию вашего продукта требует разрешения.

Мы приветствуем, но не требуем ссылок на книгу. Ссылка на книгу обычно включает в себя название, автора, издателя, и ISBN. Например: "**Book Title** by Some Author. Copyright 2008 O'Reilly Media, Inc., 978-0-596-xxxx-x."

Если вы считаете, что использование вами примеров кода выходит за рамки разрешений, приведенных выше, не стесняйтесь обращаться к нам на `<permissions@oreilly.com>`.

# 6. Safari® Books Online

### Примечание

Когда вы видите значок Safari® Books Online на обложке вашей любимой технической книги, это означает, что книга доступна через O'Reilly Network Safari Bookshelf.

Safari предлагает решение, которое лучше, чем просто электронные книги. Это виртуальная библиотека, которая позволяет вам легко производить поиск по тысячам популярных технических книг, копировать примеры кода, скачивать главы и быстро находить ответы когда вам нужна наиболее точная и актуальная информация. Попробуйте воспользоваться её возможностями бесплатно на http://my.safaribooksonline.com [http://my.safaribooksonline.com/?portal=oreilly].

# 7. Как с нами связаться

С комментариями и вопросами по поводу этой книги обращайтесь к издателю:

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707 829-0104 (fax)

У нас есть веб-страница этой книги, где есть список опечаток, примеры и другая дополнительная информация. Вы можете получить доступ к этой странице по адресу:

http://www.oreilly.com/catalog/<catalog page>
*Don’t forget to update the <url> attribute, too.*

Чтобы прокомментировать или задать технические вопросы по этой книге, отправьте сообщение по электронной почте:

`<bookquestions@oreilly.com>`

Для получения дополнительной информации о наших книгах, конференциях, Resource Centers и O’Reilly Network, посмотрите наш веб-сайт:

http://www.oreilly.com

# Глава 1. Как мы сюда попали?

## 1.1. Зачем нужен контроль версий? Почему Mercurial?

Контроль ревизии представляет собой процесс управления несколькими версиями информации. В своей простейшей форме, это то, что многие люди делают вручную: каждый раз, когда вы изменяете файл, сохраняя его под новым именем, которое содержит номер, каждый из которых выше, чем число предшествующей версии.

Ручное управление несколькими версиями для одного файла — задача с возможностью ошибок, хотя программные средства для автоматизации этого процесса давно уже доступны. Первые автоматизированные средства контроля версий были созданы, чтобы помочь одному человеку управлять версиями одного файла. За последние несколько десятилетий, количество инструментов контроля версий значительно увеличилось. Теперь они управляют версиями многих файлов, и помогают нескольким людям работать вместе. Благодаря лучшим современным средствам контроля версий тысячи людей могут работать над общими проектами, содержащими сотни тысяч файлов.

Распределенные системы контроля версий появились относительно недавно, однако, эта область быстро развивается благодаря тяге людей к исследованию нового.

Я пишу книгу о распределенных системах контроля версий, потому что считаю, что это важная тема, которая заслуживает руководства. Я решил написать о Mercurial, потому что это простой инструмент, чтобы познакомится с этой областью, в то же время — это реальный инструмент, отработанный в сложных условиях, когда многие другие средства контроля версий ещё не проверены.

### 1.1.1. Зачем использовать систему контроля версий?

Есть ряд причин, по которым вы или ваша команда может использоваться автоматизированную систему контроля версий для вашего проекта.

- Она будет отслеживать историю и прогресс вашего проекта вместо вас. Для каждого изменения у вас будет запись о том, кто его сделал, почему он это сделал, когда он это сделал и в чем, собственно, заключалось изменение.
- Когда вы работаете с другими людьми, система контроля версий облегчает вам взаимодействие. Например, когда несколько человек делают потенциально конфликтные изменения одновременно, программа поможет вам определить и разрешить такие конфликты.
- Она может помочь вам восстановиться после ошибок. Если вы сделаете изменение, которое потом окажется ошибкой, вы сможете вернуться к более ранней версии одного или нескольких файлов. На самом деле, действительно хорошая система контроля версий даже поможет вам легко найти тот момент времени, когда проблема впервые появилась (см. Раздел 9.5, «Поиск источника ошибки»).
- Она поможет вам работать одновременно над несколькими версиями проекта и переключаться между ними.

Все эти причины одинаково важны (по крайней мере, в теории) независимо от того, работаете ли вы над проектом в одиночку или вместе с сотней других людей.

Ключевой вопрос относительно практичности системы контроля версий в таких разных средах («хакер-одиночка» и «огромная команда») в том, как преимущества соотносятся с «ценой» использования. Система, которую сложно понять или использовать, обойдётся дорого.

Проект для пятисот человек скорее всего развалится под собственным весом практически сразу, если в нём не используется система контроля версий и нет процесса разработки. В подобном случае даже говорить о цене использования контроля версий не имеет смысла, так как без неё провал практически неизбежен.

С другой стороны, проект «на коленке» для программиста-одиночки может показаться неудачным примером для использования системы контроля версий, потому как цена её использования наверняка будет сравнима с общей стоимостью всего проекта. Верно?

Mercurial уникален тем, что поддерживает оба уровня разработки. Вы можете изучить основы всего за несколько минут и, благодаря минимальным накладным расходам, вы с лёгкостию сможете применять управление версиями к наималейшим из проектов. Его простота означает, что вам не придется удерживать в голове множество маловразумительных концепций или последовательностей команд вместо того, чтобы реально работать. В то же время, высокая производительность Mercurial и его распределённая природа позволит вам безболезненно увеличивать масштаб для управления большими проектами.

Ни одна система контроля версий не поможет спасти скверно управляемый проект, но хорошие инструменты могут оказать значительное влияние на удобство работы над проектом.

### 1.1.2. Множество названий для контроля версий

Системы контроля версий используются в различных областях, так что их называют разными именами и сокращениями. Вот некоторые из наиболее распространенных вариаций с которыми вы можете столкнуться:

- Контроль ревизий (RCS, Revision control system)
- Конфигурационное управление (SCM, Software configuration management)
- Управление исходным кодом (Source code management)
- Контроль исходного кода (Source code control)
- Контроль версий (VCS, Version control system)

Некоторые люди утверждают, что на самом деле у этих терминов разные значения, но на практике они настолько сильно пересекаются, что нет общепринятого или хотя бы полезного способа разделить их.

## 1.2. О примерах в этой книге

В этой книге довольно необычные примеры. Каждый пример «живой» — фактически, это результат исполнения скрипта, который выполняет приведённые команды Mercurial. Каждый раз, когда книга собирается из исходников, все примеры автоматически исполняются и их текущие результаты сравниваются с ожидаемыми.

Плюс такого подхода в том, что примеры всегда верны. Они описывают поведение в точности той версии Mercurial, которая указана в начале книги. Если я обновлю версию Mercurial, которую я описываю, и вывод какой-нибудь команды изменится — сборка книги не выполнится.

У этого подхода есть маленький недостаток, заключающийся в том, что даты и время, которые вы видите в примерах, «слишком близки» друг к другу, чего бы не происходило, если бы те же самые команды вбивал человек. Там, где человек может набрать не больше одной команды за несколько секунд, с соответствующим образом разнесёнными «отпечатками времени» (timestamps), мои автоматизированные примеры исполняют много команд за одну секунду.

Например, несколько последовательных коммитов в примере могут выглядеть исполненными в одну и ту же секунду. Вы можете это увидеть на примере использования команды `bisect` в Раздел 9.5, «Поиск источника ошибки».

Так что когда вы читаете примеры, не придавайте слишком большого значения датам или времени, которые вы увидите в результатах исполнения команд. Однако, будьте уверены, что поведение команд, которое вы видите, в целом верно и воспроизводимо.

## 1.3. Тенденции в этой области

За последние четыре десятилетия в разработке и в использовании систем контроля версий — пока люди осваивали возможности своих инструментов и сталкивались с их ограничениями — можно проследить следующую тенденцию.

Первое поколение начинало с управления одиночными файлами на индивидуальных компьютерах. Хотя эти инструменты имели большое преимущество над специализированным ручным управлением версиями, их

блокирующая модель и зависимость от одного компьютера позволяли применять их лишь в тесных маленьких командах.

Второе поколение ослабило эти ограничения переходом на сетевые архитектуры и комплексное управление проектами. По мере роста проектов появлялись новые проблемы. Если клиенты нуждались в очень частом взаимодействии с серверами, масштабирование этих серверов становилось сложной задачей для больших проектов. Ненадёжное соединение с сетью могло вообще не давать удалённым пользователям общаться с сервером. Когда проекты с открытым кодом стали давать всем анонимный доступ только на чтение, люди, не имеющие достаточных прав для внесения изменений, обнаружили, что не могут использовать инструменты для взаимодействия с проектом естественным образом, так как не могут вносить свои правки.

Современное поколение инструментов контроля версий — по природе своей децентрализованное (peer-to-peer). Эти системы устранили зависимость от одного центрального сервера и позволили людям передавать данные контроля версий туда, где они действительно необходимы. Сотрудничество через Интернет, прежде ограниченное технологией, ныне стало вопросом выбора и согласия. Современные инструменты могут действовать и без сети, неограниченно и автономно, сетевое соединение требуется только при синхронизации изменений с другим репозиторием.

# 1.4. Некоторые из преимуществ распределенных систем контроля версий

Хотя инструменты распределённого контроля версий уже несколько лет так же надёжны и удобны, как и их аналоги предыдущего поколения, люди, использующие старые инструменты, ещё не осознали преимущества новых. Во многих отношениях распределённые инструменты блистают по сравнению с централизованными.

Для отдельного разработчика распределённые инструменты практически всегда намного быстрее централизованных. Этому есть простое объяснение: централизованная утилита для многих обыденных операций должна общаться по сети, поскольку большинство метаданных хранятся в единственном экземпляре на центральном сервере. Распределённый инструмент хранит все свои метаданные локально. При прочих равных условиях общение по сети увеличивает накладные расходы использования централизованного инструмента. Не недооценивайте значимость шустрого, быстро реагирующего инструмента: вам доведётся провести массу времени, взаимодействуя с системою контроля версий.

Распределённые инструменты безразличны к причудам вашей серверной инфраструктуры, опять же потому, что они создают дубликаты метаданных во множестве мест. Если вы используете централизованную систему, а ваш сервер воспламенится, то останется надеяться на то, что резервные копии надёжны, и что последнее создание их прошло успешно и не очень давно. С распределённым инструментом вам доступно множество резервных копий — на компьютере у каждого разработчика.

Надёжность вашего сетевого соединения будет влиять на распределённые системы значительно меньше, чем на централизованные. А использовать централизованную утилиту без сетевого соединения у вас даже не получится, за исключением нескольких сильно ограниченных команд. С распределённой системой отключение сетевого соединения во время работы вообще может пройти незамеченным. Единственное, что будет невозможным — запросы к репозиториям на других компьютерах, что происходит не так уж и часто по сравнению с другими операциями. Если вы состоите в группе разработчиков, находящихся на большом расстоянии друг от друга, это может быть значимым.

## 1.4.1. Преимущества для проектов с открытым исходным кодом

Если вы нашли открытый проект, над которым вам хотелось бы поработать, и проект использует распределённую систему контроля версий, вы находитесь на одной ступеньке с людьми, которые являются «ядром» проекта. Если они публикуют свои репозитории, вы можете незамедлительно копировать историю разработки, делать изменения и записывать их точно так же, как это делают полноправные участники проекта. Централизованную систему, напротив, придётся использовать в режиме «только чтение», если только кто-нибудь не даст вам достаточно прав для фиксирования изменений на центральном сервере. До тех пор у вас не будет никакой возможности фиксировать изменения и они будут под риском искажения каждый раз при обновлении рабочей копии репозитория.

#### 1.4.1.1. Ветвления — не проблема

Есть мнение, что распределённые системы контроля версий добавляют риска проектам с открытым исходным кодом, поскольку делают простым «ветвление» разработки проекта. Это случается, когда существуют разногласия во взглядах или отношениях между группами разработчиков, которые ведут к принятию ими решения о невозможности работать вместе. Тогда каждая сторона берёт более или менее полную копию исходного кода проекта и идёт в своём собственном направлении.

Иногда стороны решают объединиться и согласовать изменения. С централизованной системой процесс слияния изменений технически очень сложен и в основном должен быть произведён вручную. Вам придётся решать, чья история «выиграет», и каким-то образом прививать изменения другой команды. Обычно при этом теряется история изменений одной или обеих сторон.

Распределённые системы поступают с ветвлением очень просто — они объявляют его единственным путём развития проекта. Каждое изменение, которое вы делаете, потенциально является точкой ответвления. Силой такого подхода является то, что инструмент должен быть действительно хорош в объединении веток, потому что ветки крайне важны: они всё время создаются.

Если каждый кусочек работы, делаемой всеми, всегда оформляется в терминах ответвления и слияния, тогда то, что мир открытого ПО называет «ветвлением», становится исключительно социальной проблемой. Как бы то ни было, распределённые системы понижают вероятность ветвления:

- Они убирают социальное разделение, которое привнесли централизованные системы, между инсайдерами (теми, кто может вносить изменения) и аутсайдерами (теми, кто не может).
- Они упрощают воссоединение после социального ветвления, так как с точки зрения контроля версий это ничем не отличается от обычного слияния.

Некоторые люди сопротивляются использованию распределённого контроля версий, потому что хотят сохранить за собой строгий контроль над своими проектами, и думают, что централизованные системы дадут им его. Тем не менее, если вы придерживаетесь таких убеждений, и при этом разрешаете публичный доступ к своему CVS/Subversion репозиторию, то знайте, что существует множество инструментов, позволяющих вытащить полную историю вашего проекта (пусть даже и медленно) и пересоздать её в таком месте, которое вы не сможете контролировать. Таким образом получается, что ваш контроль в этом случае иллюзорен, и в то же время вы потеряли возможность гибко сотрудничать с теми людьми, которые почувствовали себя вынужденными продублировать вашу историю или ответвиться от неё.

### 1.4.2. Преимущества для коммерческих проектов

Команды многих коммерческих проектов зачастую разбросаны по всему земному шару. Территориально удалённые от главного сервера разработчики могут сталкиваться с такими проблемами, как замедленная реакция на выполнение команд и, возможно, перерывы в доступности системы. Коммерческие системы контроля версий предлагают решение данной проблемы в виде расширений для удалённой репликации данных. Такие расширения как правило довольно дороги и сложны для администрирования. Распределённые системы изначально лишены подобных недостатков. Более того, можно установить несколько проектных серверов, скажем один в каждом офисе, для сокращения объёма избыточного трафика между репозиториями через дорогие каналы связи.

Централизованные системы контроля версий как правило обладают ограниченной масштабируемостью. Падение дорогой централизованной системы под нагрузкой, вызванной одновременным обращением всего пары дюжин пользователей, не является чем-то необычным. Повторюсь, наиболее типичным решением проблемы будет дорогой и тяжелый механизм репликации. Так как нагрузка на главный сервер — даже если он единственный — для распределённого инструмента контроля версий во много раз меньше (потому что все данные реплицируются повсюду), один недорогой сервер может удовлетворять потребности гораздо более многочисленной команды разработчиков и для репликации с целью распределения нагрузки нужно лишь написать несложные скрипты.

Если некоторые члены вашей команды работают «в поле», разрешая проблемы на площадке заказчика, они также получают определённые преимущества от использования распределённой системы контроля версий. Инструмент позволит им строить кастомные сборки, пробовать различные исправления изолированно друг от друга, а также проводить эффективный поиск причины ошибки в среде заказчика с использованием истории версий. Всё это может быть выполнено без необходимости в постоянном соединении с внутренней сетью компании.

# 1.5. Почему следует остановить выбор на Mercurial?

Mercurial обладает уникальным набором свойств, позволяющим выбрать его в качестве наиболее подходящей системы контроля версий:

- Прост в изучении и использовании
- Легковесный
- Превосходно масштабируется
- Легко настраивается под конкретные нужды

Если вы обладаете опытом в использовании систем контроля версий, вам потребуется меньше пяти минут, чтобы начать работать с Mercurial. Если же вы новичок, процесс знакомства не должен занять больше десяти минут. Mercurial предоставляет единообразную и последовательную систему команд и функций, что позволяет руководствоваться небольшим набором общих правил вместо того, чтобы учить массу исключений.

В небольших проектах вы можете начать работу с Mercurial в считанные минуты. Создание новых веток и изменений, распространение изменений (как локально, так и по сети), операции с историей и статусом — всё это работает быстро. Mercurial старается быть незаметным и не путаться под вашими ногами, не требует от вас больших умственных усилий и совершает свои операции невероятно быстро.

Mercurial применяется не только в маленьких проектах, его используют и в проектах с сотнями и тысячами разработчиков, проектах, которые содержат десятки тысяч файлов и сотни мегабайт исходного кода.

Если вам не хватает базовой функциональности Mercurial, то её легко расширить. Mercurial хорошо подходит для задач скриптинга, его понятное устройство и реализация на языке Python позволяет легко добавлять новые возможности в виде расширений. Существует большое количество популярных и полезных расширений, охватывающих спектр задач от помощи в нахождении ошибок до улучшения производительности.

# 1.6. Сравнение Mercurial с другими системами контроля версий

Прежде чем вы продолжите чтение, вам следует уяснить, что этот раздел отражает мой опыт, интересы и (да, я осмелюсь сказать это) мои наклонности. Я использовал каждую из перечисленных ниже систем контроля версий в большинстве случаев в течение нескольких лет.

## 1.6.1. Subversion

Subversion — популярная система контроля версий, разработанная с целью заменить CVS. Subversion имеет централизованную клиент/серверную архитектуру.

Subversion и Mercurial имеют похожие команды для одних и тех же операций, так что если вы хорошо знаете одну из этих систем, вам легко будет научиться пользоваться другой. Обе системы портированы на все популярные операционные системы.

Subversion до версии 1.5 не имел нормальной поддержки слияния. На момент написания книги возможность отслеживания слияний являлась относительно новой, с присущими сложностями и ошибками [http://svnbook.red-bean.com/nightly/en/svn.branchmerge.advanced.html#svn.branchmerge.advanced.finalword].

В каждой операции, производительность которой я измерял, Mercurial обладает большей производительностью, чем Subversion. Скорость больше в 2-6 раз, когда речь идет о локальном репозитории Subversion 1.4.3 (самый быстрый метод доступа). При более реалистичном варианте использования — сетевой репозиторий, Subversion находится в существенно худшем положении. В силу того, что команды Subversion должны взаимодействовать с

сервером и при этом Subversion не имеет полезных средств репликации, производительность сервера и пропускная способность сети становятся узкими местами даже для некрупных проектов.

Кроме того, Subversion требует дополнительное дисковое пространство для того, чтобы избежать сетевых запросов при выполнении некоторых операций: поиск модифицированных файлов (`status`) и отображение изменений (`diff`). В результате рабочая копия Subversion такого же размера (а то и больше) как репозиторий Mercurial и рабочий каталог вместе взятые, хотя репозиторий Mercurial содержит полную историю проекта.

Subversion имеет широкую поддержку инструментария сторонних производителей. В этом отношении у Mercurial сейчас существенное отставание. Хотя разрыв сокращается, и некоторые GUI-утилиты для Mercurial превосходят свои аналоги для Subversion. Как и Mercurial, Subversion располагает отличным руководством пользователя.

Из-за того, что Subversion не хранит историю изменений на клиенте, она хорошо подходит для управления проектами, содержащими большое количество двоичных файлов. Если вы внесете в несжимаемый десятимегабайтный файл 50 изменений, то дисковое пространство, использованное Subversion останется неизменным. Пространство, используемое любой из распределенных систем контроля версий, будет быстро увеличиваться пропорционально количеству изменений, потому что различия между правками большие.

Кроме того, обычно трудно, а чаще даже невозможно слить разные версии двоичного файла. Subversion позволяет пользователю заблокировать файл, в результате пользователь на время получает эксклюзивные права на внесение изменений в него. Это может быть значительным преимуществом для проекта, в котором широко используются двоичные файлы.

Mercurial может импортировать историю изменений из репозитория Subversion. Возможен и обратный процесс. Это делает возможным прощупать почву и использовать Mercurial и Subversion одновременно, прежде чем решить, осуществлять переход или нет. Преобразование истории — пошаговый процесс, так что вы можете осуществить начальное преобразование, а потом вносить новые изменения.

## 1.6.2. Git

Git — распределенная система контроля версий, которая была разработана для управления исходным кодом ядра Linux. Как и в случае с Mercurial, на начальный дизайн системы оказал влияние Monotone.

Git предоставляет большой список команд, число которых в версии 1.5.0 достигает 139 уникальных единиц. Он имеет репутацию инструмента, сложного для изучения. В сравнении с Git, Mercurial делает упор на простоту.

Что касается производительности — Git очень быстр. В некоторых случаях он быстрее, чем Mercurial (по крайней мере под Linux), а в других быстрее оказывается Mercurial. Однако под Windows как производительность, так и общий уровень поддержки, во время написания этой книги у Git гораздо хуже, чем у Mercurial.

В то время как репозиторий Mercurial не требует операций по техническому обслуживанию, репозиторий Git требует частых ручных «перепаковок» собственных метаданных. Если этого не делать, производительность начинает падать, наряду с увеличением объёма занимаемого дискового пространства. Дисковый массив сервера, содержащего несколько Git репозиториев, по отношению к которым не выполняется строгое правило частой «перепаковки», рано или поздно забивается под завязку, в результате чего процесс ежедневного резервного копирования легко может занимать более 24 часов. Только что «запакованный» репозиторий Git занимает немного меньше места, чем репозиторий Mercurial, но объём не перепакованного репозитория будет на несколько порядков больше.

Ядро Git написано на языке C. Многие команды Git реализованы в виде Shell скриптов или скриптов на языке Perl и уровень качества данных скриптов сильно разнится. Я встречал несколько установок, в которых скрипты тупо продолжали выполнение, несмотря на наличие фатальных ошибок.

Mercurial предоставляет возможность импорта истории версий из репозитория Git.

## 1.6.3. CVS

CVS, наверное, самая широко распространённая система контроля версий в мире. Благодаря почтенному возрасту, а также бардаку, царящему внутри, он очень слабо поддерживается уже много лет.

CVS основан на централизованной, клиент-серверной архитектуре. Он не выполняет группировку файловых изменений в атомарные коммиты, тем самым позволяя людям легко «сломать билд»: один человек может успешно внести часть изменений в репозиторий, а затем оказаться заблокированным из-за необходимости выполнения слияния. Это приведёт к ситуации, когда остальные участники увидят только часть из тех изменений, которые они должны были увидеть. Данная особенность также влияет на то, как вы будете работать с историей изменений. Если вы хотите получить все изменения, которые один из членов команды внёс для решения определённой задачи, вам необходимо вручную исследовать описания и дату внесения изменений, произведённых для каждого затрагиваемого файла (если вы вообще знаете, какие файлы были затронуты).

CVS оперирует довольно запутанными понятиями веток и меток, которые я даже не буду пытаться описать в данной книге. Он не поддерживает переименование как файлов, так и папок, благодаря чему репозиторий может быть достаточно легко повреждён. Так как внутренние механизмы контроля целостности практически отсутствуют, зачастую даже невозможно точно утверждать, повреждён ли репозиторий, и если да, то каким образом. Таким образом я бы не стал рекомендовать CVS для использования в любом из существующих или новых проектов.

Mercurial предоставляет возможность импорта истории версий CVS. Тем не менее здесь есть несколько подводных камней, с которыми также сталкиваются любые другие инструменты иморта из CVS. Отсутствие атомарных изменений и версионирования иерархических данных файловой системы приводит к невозможности абсолютно точного реконструирования истории изменений CVS, поэтому в некоторых случаях используются допущения, а переименования обычно не отображаются. Так как множество задач по администрированию CVS должны выполняться вручную, что повышает риск ошибок, обычна ситуация, когда средство для импорта из CVS возвращает множество ошибок целостности репозитория (абсолютно нереальные даты изменения версий и файлы, которые остаются заблокированными на протяжении последнего десятка лет — это лишь пара из наименее интересных проблем, которые я могу вспомнить из собственного опыта).

Mercurial предоставляет возможность импорта истории версий из репозитория CVS.

### 1.6.4. Коммерческий инструментарий

Perforce основан на централизованной, клиент-серверной архитектуре, при этом данные не кэшируются на клиентской стороне. В отличие от современных средств контроля версий, Perforce требует от пользователя запуска специальной команды, информирующей сервер о каждом файле, который человек собирается редактировать.

Производительность Perforce вполне достаточна для небольших команд, но стремительно падает, если количество пользователей переваливает за пару дюжин. Умеренно большие установки Perforce требуют развёртывания прокси-серверов для распределения нагрузки, генерируемой пользователями.

### 1.6.5. Выбор системы контроля версий

За исключением CVS, все инструменты, перечисленные выше, имеют уникальные свойства, которые делают их подходящими для определённых стилей ведения проектов. Не существует инструмента, который мог бы быть использован в любой ситуации.

Например, Subversion является хорошим выбором для работы с часто изменяющимися бинарными файлами, благодаря его централизованной архитектуре и поддержке блокировок на уровне файлов.

Лично меня в Mercurial привлекает простота, производительность и хорошая поддержка процесса слияния — этот превосходный набор служит мне уже несколько лет.

## 1.7. Переход с других систем контроля версий на Mercurial

Mercurial поставляется с расширением под названием `convert`, которое пошагово импортирует историю изменений из некоторых систем контроля версий. Под словом «пошагово» я подразумеваю, что вы за один раз можете сконвертировать историю проекта до определенной даты, а позже запустить преобразование еще раз для получения изменений, произошедших после первичной конвертации.

Поддерживаются преобразование из следующих систем контроля версий:

- Subversion
- CVS
- Git
- Darcs

Кроме того, `convert` может экспортировать изменения из Mercurial в Subversion. Это позволяет использовать Subversion и Mercurial параллельно, без риска потери данных.

Команда **convert** проста в использовании. Просто укажите путь или URL исходного репозитория и имя целевого репозитория (необязательно), и она начнет работу. После первичного преобразования, запустите ту же самую команду для получения новых изменений.

# 1.8. Краткая история контроля версий

Самая известная из старых утилит контроля версий — SCCS (Source Code Control System, система контроля исходного кода), которую написал Марк Рочкайнд (Marc Rochkind) из Bell Labs, в начале 70-х. SCCS оперировала отдельными файлами и требовала, чтобы каждый человек, работающий над проектом, имел доступ к общему рабочему пространству, существовавшему в единственном экземпляре. Только один человек мог одновременно редактировать файл в один момент времени; конфликты доступа к файлам разрешались блокировками. Обычной ситуацией было забывание снятия блокировки после редактирования, что запрещало доступ к файлу другим людям без помощи администратора.

Вальтер Тичи (Walter Tichy) разработал свободную альтернативу SCCS в начале 1980-х; он назвал свою программу RCS (Revision Control System, система контроля ревизий). Подобно SCCS, RCS требовала от разработчиков как работы в едином разделяемом рабочем пространстве, так и блокировки файлов для предотвращения одновременного изменения файлов разными людьми.

Позднее, в 1980-х же годах, Дик Грюн (Dick Grune) использовал RCS как основу для набора shell-скриптов, изначально названных cmt, а позднее переименованных в CVS (Concurrent Versions System, система одновременных версий). Крупное нововведение CVS заключалось в том, что она позволяла разработчикам работать одновременно и, в некоторой степени, независимо в их личных рабочих пространствах. Этими-то пространствами и предотвратились постоянные наступания разработчиков друг другу на пятки, которое было обычным делом в SCCS и RCS. Каждый разработчик имел копию каждого файла проекта, разработчики могли модифицировать свои копии независимо. Им приходилось объединять собственные правки только перед отсылкою изменений в центральное хранилище.

Брайан Берлинер (Brian Berliner) взял первоначальные скрипты Грюна и переписал их на Си, выпустив в 1989 году код, который впоследствии развился в современную версию CVS. CVS в дальнейшем приобрела возможность работать по сети, обретя клиент-серверную архитектуру. Архитектура CVS является централизованной: только на сервере есть копия истории проекта. Клиентские рабочие копии содержали только экземпляры файлов последней версии и небольшие метаданные для определения местонахождения сервера. Система CVS достигла небывалого успеха: вероятно, она является самой широко используемой системой контроля версий в мире.

В начале 1990-х годов Sun Microsystems разработала раннюю распределённую систему контроля версий, называвшуюся TeamWare. Каждая рабочая копия TeamWare содержала полную копию истории изменений проекта. Понятие центрального репозитория в TeamWare отсутствовало как таковое. (Подобно CVS, использовавшей RCS для хранения истории, TeamWare использовала SCCS.)

Шли 1990-ые, росла осведомлённость о нескольких проблемах CVS. Система записывает одновременные изменения нескольких файлов раздельно, а не группирует их в одну логически атомарную операцию. Способ управления файловой иерархией не очень хорош: нетрудно устроить в репозитории беспорядок, переименовывая файлы и каталоги. Более того, исходные коды CVS непросто понимать и поддерживать, что сделало практически непреодолимым «болевой порог» исправления этих архитектурных проблем.

В 2001 году Джим Бланди (Jim Blandy) и Карл Фогель (Karl Fogel) — два разработчика, прежде работавшие над CVS — начали проект по её замене таким средством, которое имело бы архитектуру получше и код почище. Результат — Subversion — не отошёл от централизованной клиент-серверной модели CVS, но добавил атомарные коммиты нескольких файлов, лучшее управление пространствами имён и другие возможности, которые сделали Subversion более удобным средством работы, нежели CVS. Со времени выхода первой версии Subversion быстро обретал популярность.

Более или менее одновременно, Грейдон Хоар (Graydon Hoare) начал работать над амбициозной системой контроля версий, которую назвал Monotone. Эта система не только устраняет множество проблем внутреннего устройства CVS и имеет распределённую архитектуру, но и идёт далее нескольких прежних (и последующих) систем контроля версий в некоторых своих нововведениях. Monotone использует криптографические хеши в качестве идентификаторов и имеет неотъемлемое представление о «доверии» коду из различных источников.

Жизнь Mercurial началась в 2005 году. В то время как некоторые аспекты его архитектуры были созданы под влиянием Monotone, Mercurial сосредоточен на простоте использования, высокой производительности и масштабируемости до очень больших проектов.

# Глава 2. Экскурсия по Mercurial: основы

## 2.1. Установка Mercurial на вашем компьютере

Прекомпилированные пакеты Mercurial доступны для каждой популярной операционной системы. Это позволяет вам начать использование Mercurial на вашем компьютере немедленно.

### 2.1.1. Windows

Лучшей версией Mercurial для windows является TortoiseHg, который можно найти на сайте http://tortoisehg.org. Этот пакет не имеет внешних зависимостей, он «просто работает». Он позволяет использовать командную строку и графический пользовательский интерфейс.

### 2.1.2. Mac OS X

Ли Канти публикует инсталлятор mercurial для mac os x на http://mercurial.berkwood.com.

### 2.1.3. Linux

В силу того, что каждый дистрибутив Linux использует свои собственные менеджеры пакетов, а также стратегии и темпы разработки, трудно дать подробную инструкции по установке Mercurial. Версия Mercurial, которую вы получите очень зависит от того, насколько активен тот, кто занимается созданием пакетов для вашего дистрибутива.

Чтобы не усложнять процесс, я сфокусируюсь на установке Mercurial из командной строки в наиболее популярных дистрибутивах Linux. Большинство из них располагают менеджерами пакетов с графическим интерфейсом, что позволит вам установить Mercurial нажатием одной кнопки. Пакет для установки называется `mercurial`.

- Ubuntu и Debian:

```
apt-get install mercurial
```

- Fedora:

```
yum install mercurial
```

- OpenSUSE:

```
zypper install mercurial
```

- Gentoo:

```
emerge mercurial
```

### 2.1.4. Solaris

SunFreeWare, на http://www.sunfreeware.com , предоставляет готовые пакеты Mercurial.

## 2.2. Начало работы

Для начала выполним команду **hg version**, чтобы удостовериться, что Mercurial установлен правильно. Какая версия на самом деле — неважно, главное, что она вообще что-то выводит.

```
$ hg version
Mercurial Distributed SCM (version 2.0)
(see http://mercurial.selenic.com for more information)

Copyright (C) 2005-2011 Matt Mackall and others
This is free software; see the source for copying conditions. There is NO
```

```
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

## 2.2.1. Встроенная справка

У Mercurial есть встроенная справка. Очень сложно переоценить ее наличие, особенно когда ваша работа остановилась из-за того, что вы не можете вспомнить как выполнить какую-то команду. Если Вы застопорились, просто выполните **hg help**, и на экран выведется краткий список команд с описанием назначения каждой из них. Если же вы еще и укажите конкретную команду (см. ниже), выведется подробная информация именно о ней.

```
$ hg help init
hg init [-e CMD] [--remotecmd CMD] [DEST]

create a new repository in the given directory

    Initialize a new repository in the given directory. If the given directory
    does not exist, it will be created.

    If no directory is given, the current directory is used.

    It is possible to specify an "ssh://" URL as the destination. See "hg help
    urls" for more information.

    Returns 0 on success.

options:

 -e --ssh CMD       specify ssh command to use
    --remotecmd CMD specify hg command to run on the remote side
    --insecure      do not verify server certificate (ignoring web.cacerts
                    config)

use "hg -v help init" to show more info
```

Для большей детализации выполните **hg help `-v`**. Опция `-v` - сокращение от `--verbose`, заставит Mercurial выводить больше информации, чем обычно.

# 2.3. Работа с репозиторием

В Mercurial все происходит внутри репозитория. Репозиторий проекта содержит все файлы, которые «относятся» к проекту, а также историю изменений этих файлов.

В репозитории нет никакой магии, это просто каталог в файловой системе, который Mercurial обрабатывает особым образом. Вы можете переименовать или удалить репозиторий в любое время через командную строку или вашим собственным файловым менеджером.

## 2.3.1. Создание локальной копии репозитория

Копирование репозитория кое-чем отличается. Хотя вы можете скопировать репозиторий как обычный каталог, лучше использовать встроенную команду Mercurial. Она называется **hg clone**, потому что создает идентичную копию существующего репозитория.

```
$ hg clone http://hg.serpentine.com/tutorial/hello
destination directory: hello
requesting all changes
adding changesets
adding manifests
adding file changes
added 5 changesets with 5 changes to 2 files
updating to branch default
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Одно из преимуществ использования **hg clone** в том, что как мы видим выше, она позволяет клонировать репозитории по сети. Другим является то, что она запоминает, откуда мы его клонировали. Скоро мы убедимся, что это полезно, когда мы захотим принести новые изменения из другого репозитария.

Если клонирование прошло успешно, то у вас должен появится каталог под названием `hello`. В нем должны быть какие-то файлы.

```
$ ls -l
total 0
drwxr-x--- 3 slava slava 100 Feb  2 14:10 hello
$ ls hello
Makefile  hello.c
```

У файлов в нашем репозитории то же самое содержимое и история, как и в исходном.

Каждый репозиторий Mercurial полон, самодостаточнен и независим. Он содержит свою собственную копию файлов проекта и их историю. Склонированный репозиторий помнит, откуда он был склонирован, но не общается с тем репозиторием, да и ни с каким другим тоже, до тех пор пока вы ему не скажете.

Это означает, что вы можете свободно экспериментировать с вашим репозиторием. Это безопасно, потому что ваш репозиторий — «закрытая песочница», изменения в котором не повлияют ни на что, кроме него самого.

## 2.3.2. Что есть в репозитории?

Когда мы более пристально присмотримся к репозиторию, мы увидим, что он содержит каталог под названием `.hg`. Это место, где Mercurial хранит все метаданные репозитория.

```
$ cd hello
$ ls -a
.  ..  .hg  Makefile  hello.c
```

Содержание каталога `.hg` и его подкаталогов является собственностью Mercurial. Со всеми остальными файлами и каталогами в репозитории мы можем делать что угодно.

Строго говоря, каталог `.hg` — это и есть «настоящий» репозиторий, а все остальные файлы и каталоги рядом с ним называются рабочим каталогом. Разницу запомнить довольно просто — репозиторий содержит всю историю вашего проекта, в то время как рабочий каталог содержит слепок вашего проекта в определенной точке истории.

# 2.4. Путешествие по истории

Самое первое, что вы захотите сделать с новым, неизвестным репозиторием — изучить его историю. Команда **hg log** предназначена как раз для этого.

```
$ hg log
changeset:   4:2278160e78d4
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:16:53 2008 +0200
summary:     Trim comments.

changeset:   3:0272e0d5a517
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:08:02 2008 +0200
summary:     Get make to generate the final binary from a .o file.

changeset:   2:fef857204a0c
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:05:04 2008 +0200
summary:     Introduce a typo into hello.c.

changeset:   1:82e55d328c8c
user:        mpm@selenic.com
date:        Fri Aug 26 01:21:28 2005 -0700
summary:     Create a makefile

changeset:   0:0a04b987be5a
user:        mpm@selenic.com
date:        Fri Aug 26 01:20:50 2005 -0700
summary:     Create a standard "hello, world" program
```

По умолчанию, эта команда выводит краткую информацию о каждом изменении в проекте, которое было зафиксировано. В терминологии Mercurial мы называем эти зафиксированные события ревизией (**changeset**), потому что она может содержать изменения в различных файлах.

**hg log** выводит записи со следующими полями:

- `changeset` (ревизия). Она состоит из десятичного числа, двоеточия и строки шестнадцатеричных цифр. Это идентификаторы ревизии. Строка шестнадцатеричных цифр представляет собой уникальный идентификатор: та же строка шестнадцатеричных цифр будет обозначать этот набор изменений в каждой копии этого хранилища. Номер короче и проще напечатать, чем строку шестнадцатеричных цифр, но он не является уникальным: это же число в двух различных копиях репозитория могут иметь различные ревизии.

- `user` (пользователь). Идентификатор человека, создавшего ревизию. Там может находится все, что угодно, но чаще это имя человека и адрес электронной почты.

- `date`. Дата и время, когда была создана ревизия, а также часовой пояс, в котором она была создана. (Дата и время приведены относительно этого часового пояса, они указывают сколько времени было для того, кто создал ревизию)

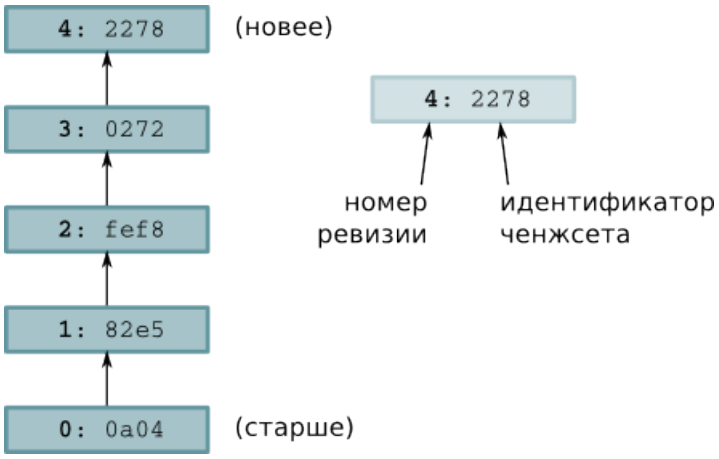
- `summary`. Первая строка комментария к ревизии, который оставил её автор.

- Некоторые наборы изменений, таких, как первая, имеют поле `tag` (тэг). Теги это еще один способ идентифицировать набор изменений, придав ему легкое для запоминания имя. (Тег с названием `tip` специальный: он всегда относится к новейшему изменению в репозитарии.)

По умолчанию **hg log** выводит очень общие сведения, с отсутствием множества деталей.

Рисунок 2.1, «Графическое представление истории репозитория `hello`» содержит графическое представление истории репозитория `hello`, что слегка облегчает понимание того, в каком направлении «развивается» его история. Мы будем возвращаться к этому рисунку в этой и следующей главах.

Рисунок **2.1.** Графическое представление истории репозитория **`hello`**




## 2.4.1. Изменения, ревизии и общение с другими людьми

Английский язык печально известен своей небрежностью, а компьютерная наука имеет обширную историю неразберихи в терминах (зачем один термин, если можно использовать четыре). В контроле версий есть множество слов и фраз, означающих одно и то же. Если речь идет об истории в Mercurial, вы увидите, что слово «changeset» (набор изменений), обычно сокращается до «change», или (при письме), до «cset», а иногда «changeset» называют «revision» (ревизия) или «rev».

Не важно, какое слово вы используете для концепции «ревизии», идентификатор, по которому вы ссылаетесь на «определенную ревизию» имеет гораздо большее значение. Вспомните, что `ревизия` в выводе команды **hg log** идентифицируется номером и шестнадцатеричной строкой.

- Номер ревизии удобен для записи, но действителен только в пределах своего репозитория.
- Шестнадцатеричная строка — постоянный, неизменный параметр, который всегда идентифицирует одну и ту же ревизию в каждой копии репозитория.

Это различие очень важно. Если вы отправите кому-нибудь письмо с упоминанием «ревизии 33», существует большая вероятность, что их ревизия 33 будет совсем другой. Причиной этого является то, что номер ревизии зависит от порядка, в котором ревизии попадают в репозиторий, и нет никакой гарантии, что одни и те же изменения произойдут в одинаковом порядке в различных репозиториях. Три изменения `A, Б и В` запросто могут появится в одном репозитории в порядке `0,1,2`, а в другом — `0,2,1`.

Mercurial использует номера ревизий исключительно для удобства. Если вам нужно обсудить с кем-то конкретную ревизию, или по какой-то другой причине ссылаться на нее (в багтрекере, например), используйте шестнадцатеричный идентификатор.

## 2.4.2. Просмотр определенных ревизий

Чтобы ограничить вывод команды **hg log** до одной ревизии, используйте опцию `-r` (или `--rev`). Вы можете использовать или номер ревизии или ее идентификатор, а также запросить ревизий столько, сколько вам захочется.

```
$ hg log -r 3
changeset:   3:0272e0d5a517
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:08:02 2008 +0200
summary:     Get make to generate the final binary from a .o file.

$ hg log -r 0272e0d5a517
changeset:   3:0272e0d5a517
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:08:02 2008 +0200
summary:     Get make to generate the final binary from a .o file.

$ hg log -r 1 -r 4
changeset:   1:82e55d328c8c
user:        mpm@selenic.com
date:        Fri Aug 26 01:21:28 2005 -0700
summary:     Create a makefile

changeset:   4:2278160e78d4
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:16:53 2008 +0200
summary:     Trim comments.
```

Если вы хотите увидеть историю нескольких ревизий, но не хотите просматривать их все, можете указать диапазон, как бы выражая мысль: «Мне нужны все ревизии от `А` до `Б` включительно».

```
$ hg log -r 2:4
changeset:   2:fef857204a0c
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:05:04 2008 +0200
summary:     Introduce a typo into hello.c.

changeset:   3:0272e0d5a517
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:08:02 2008 +0200
summary:     Get make to generate the final binary from a .o file.

changeset:   4:2278160e78d4
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:16:53 2008 +0200
summary:     Trim comments.
```

Mercurial учитывает порядок, в котором ревизии были указаны, так что **hg log -r 2:4** выводит ревизии 2,3 и 4. Тогда как **hg log -r 4:2** — 4,3 и 2.

### 2.4.3. Подробности

В то время как информация, которую выводит **hg log** полезна, если вы знаете что ищете, вам может понадобиться полное описание изменений или список измененных файлов, если вы хотите узнать та ли это ревизия, что вам нужна. Команда **hg log** с аргументом `-v` (`--verbose`) предоставит вам такую возможность.

```
$ hg log -v -r 3
changeset:   3:0272e0d5a517
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:08:02 2008 +0200
files:       Makefile
description:
Get make to generate the final binary from a .o file.


```

Если вы хотите видеть описание и то как изменялось содержимое, добавьте опцию `-p` (или `--patch`). Будет показываться содержание изменений в едином **diff** (если вы никогда не видели формат унифицированного diff раньше, см. Раздел 12.4, «Понимание патчей»).

```
$ hg log -v -p -r 2
changeset:   2:fef857204a0c
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:05:04 2008 +0200
files:       hello.c
description:
Introduce a typo into hello.c.


diff -r 82e55d328c8c -r fef857204a0c hello.c
--- a/hello.c Fri Aug 26 01:21:28 2005 -0700
+++ b/hello.c Sat Aug 16 22:05:04 2008 +0200
@@ -11,6 +11,6 @@

 int main(int argc, char **argv)
 {
- printf("hello, world!\n");
+ printf("hello, world!\");
   return 0;
 }

```

Опция `-p` очень полезна, поэтому следует её запомнить.

## 2.5. Об опциях команд

Давайте сделаем перерыв в изучении команд Mercurial и обсудим шаблоны работы с ними. Это будет вам полезно, когда мы продолжим наше турне.

Mercurial имеет простой и последовательный подход в работе с опциями, которые вы можете передавать командам. Это следует из соглашений по опциям, которые являются общими для современных Linux и Unix систем.

- Каждая опция имеет длинное имя. К примеру, как мы уже видели, команда **hg log** принимает параметр `--rev`.
- Также большинство опций имеют короткие имена. Вместо `--rev` можно использовать `-r`. Не все опции имеют короткие имена, потому как некоторые из них просто редко используются.
- Длинные опции начинаются с двух тире (`--rev`), а короткие начинаются с одного (`-r`).
- Имена опций и их применение в командах согласовано. Например, все команды, позволяющие указывать ID или номер ревизии принимают оба аргумента `-r` и `--rev`.
- При использовании коротких опций, вы можете печатать их вместе. Например, команда **hg log -v -p -r 2** может быть записана в виде **hg log -vpr2**.

В примерах в этой книге я использую короткую запись аргументов, а не длинную. Это всего лишь дело моего вкуса, так что не ищите в этом скрытого смысла.

Большинство выводящих какую-то информацию команд выдадут большее количество информации, если им передать опцию `-v` (или `--verbose`), или меньшее, если передать опцию `-q` (или `--quiet`).

### Вариант наименования опций

Почти всегда в командах Mercurial названия опций придерживаются единой концепции. Для примера, если команда работает с наборами изменений, вы всегда будете идентифицировать их опциями `--rev` или `-r`. Такая последовательность в использовании имён делает более лёгким запоминание опций различных команд.

# 2.6. Создание и анализ изменений

Так как мы уже умеем просматривать историю в Mercurial, можно заняться внесением изменений и их изучением.

Первое, что необходимо сделать — изолировать наш эксперимент в его собственном репозитории. Для этого используется команда **hg clone**, но так как у нас уже есть копия репозитория локально, то мы можем клонировать её вместо клонирования по сети. Это действие значительно быстрее, а кроме того, в большинстве случаев использует меньше места на диске[1].

```
$ cd ..
$ hg clone hello my-hello
updating to branch default
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd my-hello
```

Кстати говоря, хорошей идеей является хранение «нетронутой» копии удалённого репозитория, который можно использовать для создания временных клонов для получения «песочниц» для каждой задачи, над которой вам хочется поработать. Это позволяет вам работать над множеством задач одновременно, изолированных друг от друга до завершения и вашей готовности их интегрировать. Потому как локальные клоны настолько «дёшевы», что их создание и уничтожение в любое удобное время практически не несёт накладных расходов.

В нашем хранилище `my-hello`, мы имеем файл `hello.c` — классическую программу «Hello, World».

```
$ cat hello.c
/*
 * Placed in the public domain by Bryan O'Sullivan.  This program is
 * not covered by patents in the United States or other countries.
 */

#include <stdio.h>

int main(int argc, char **argv)
{
 printf("hello, world!\");
 return 0;
}
```

Давайте отредактируем этот файл чтоб он печатал на выходе вторую строку.

```
# ... edit edit edit ...
$ cat hello.c
/*
 * Placed in the public domain by Bryan O'Sullivan.  This program is
 * not covered by patents in the United States or other countries.
 */

#include <stdio.h>
```

[1] Экономия пространства возникает, когда источник и получатель репозитория находятся на одной файловой системе, в этом случае Mercurial будет использовать жесткие ссылки, чтобы использовать политику совместного использования копирование-при-записи для его внутренних метаданных. Если это объяснение ничего для вас не значит, не беспокойтесь: все происходит прозрачно и автоматически, и вам не нужно ничего понимать.

```
int main(int argc, char **argv)
{
 printf("hello, world!\");
 printf("hello again!\n");
 return 0;
}
```

Команда Mercurial **hg status** покажет, что Mercurial знает о файлах в репозитории.

```
$ ls
Makefile  hello.c
$ hg status
M hello.c
```

**hg status** выводит информацию не обо всех файлах, а только об изменённых: это строка, начинающаяся с буквы «M» для `hello.c`. Пока вы не укажете это специально, **hg status** не будет выводить информацию про файлы, которые не изменились.

«M» показывает, что Mercurial оповещен о модификации `hello.c`. Мы не уведомляли Mercurial о том, что изменили файл ни перед, ни после окончания работы, он способен самостоятельно находить изменения.

Иногда мало просто знать только о факте изменения файла, и мы предпочли бы знать, какие точно изменения были сделаны. Для этого используется команда **hg diff**.

```
$ hg diff
diff -r 2278160e78d4 hello.c
--- a/hello.c Sat Aug 16 22:16:53 2008 +0200
+++ b/hello.c Thu Feb 02 14:10:17 2012 +0000
@@ -8,5 +8,6 @@
 int main(int argc, char **argv)
 {
  printf("hello, world!\");
+ printf("hello again!\n");
  return 0;
 }
```

### Объяснение патчей

Напомню, что нужно посмотреть Раздел 12.4, «Понимание патчей», если вы не знаете, как читать вывод команды выше.

# 2.7. Запись изменений в новую ревизию

Мы можем изменять файлы, собирать и тестировать изменения и использовать команды **hg status**, **hg diff** для анализа изменений, пока мы не будем удовлетворены ими и не достигнем естественной точки, когда захочется записать проделанную работу в новую ревизию.

Команда **hg commit** позволяет создать новую ревизию. Для простоты, мы будем называть этот процесс «сделать коммит» или «закоммитить».

## 2.7.1. Установка имени пользователя

В первый раз выполнение команды **hg commit** может пройти неудачно. Mercurial записывает ваше имя и адрес в каждую ревизию, чтобы вы или другие пользователи могли связаться с автором каждого изменения. Mercurial пытается найти наиболее разумное имя пользователя для коммита. Поиск происходит в следующем порядке:

1. Если в команде **hg commit** вы указали опцию `-u`, с последующим именем пользователя, то оно будет обладать наивысшим приоритетом.

2. Если у вас установлена переменная окружения `HGUSER`, то следующей будет проверена она.

3. Если вы создали в своей домашней директории файл `.hgrc`, и в нём есть директива `username` — будет использована она. Чтобы узнать, как должен выглядеть содержимое этого файла, смотрите Раздел 2.7.1.1, «Создание файла конфигурации Mercurial».

4. Если у вас установлена переменная окружения `EMAIL`, то будет использована она.

5. Mercurial использует локальное имя пользователя и хоста в системе, чтобы создать конечное имя. Так как часто полученное имя малополезно, то будет выведено предупреждение.

Если все варианты поиска завершились неудачно, Mercurial выведет сообщение об ошибке и не позволит создать коммит, пока вы не укажете имя пользователя.

Переменная `HGUSER` и опция `-u` в команде **hg commit** должны использоватся для изменения стандартного способа выбора имени. Для нормальной эксплуатации наиболее простой и надежный путь: установить имя в файле `.hgrc`. Подробности смотри ниже.

### 2.7.1.1. Создание файла конфигурации Mercurial

Чтобы установить имя пользователя откройте свой любимый текстовый редактор и создайте файла `.hgrc` в домашней директории. Mercurial будет использовать этот файл для поиска персональных настроек. Первоначальное содержание этого файла должно выглядить примерно так.

#### «Домашняя директория» в **Windows**

Домашний каталог, на английской установке Windows это, как правило, папка `C:\Documents and Settings\<имя пользователя>`. Вы можете узнать точное название вашей домашней директории, открыв окно командной строки и выполнив следующую команду.

```
C:\> echo %UserProfile%
```

```
# This is a Mercurial configuration file.
[ui]
username = Firstname Lastname <email.address@example.net>
```

Строка «`[ui]`» обьявляет секцию конфигурационного файла. Вы можете прочитать «`username = ...`» как «установить значение переменной `username` в секции `ui`». Секции продолжаются до начала новых секций. Пустые строки и строки, начинаюшиеся с «`#`» игнорируются.

### 2.7.1.2. Выбор имени пользователя

Вы можете использовать любой текст в качестве значения `username`, так как эта информация предназначена для других людей, а не для интерпретации Mercurial'ом. В примере выше для этого использовалось распространенное соглашение: комбинация имени и адреса электронной почты.

#### Примечание

Встроенный веб-сервер Mercurial обфускцирует адреса электронной почты для затруднения работы утилит сбора адресов, которые используют спамеры. Это уменьшает вероятность того, что вы начнете получать больше спама, если опубликуете репозиторий Mercurial в сети.

## 2.7.2. Описание ревизии

Когда мы фиксируем изменения, Mercurial переводит нас в текстовый редактор, чтобы ввести комментарий, описывающее модификации, которые мы внесли в этом наборе изменений. Такое описание называется сообщением об изменениях (описанием изменений, описанием ревизии). Это будет записью для читателей о том, что мы сделали и почему, и будет выводиться при выполнении команды **hg log** после того, как мы закончим публикацию ревизии.

```
$ hg commit
```

Редактор, который откроется при выполнении команды **hg commit**, будет содержать пустую строку и несколько строк, начинающихся с «`HG:`».

```

This is where I type my commit comment.
```

```
HG: Enter commit message.  Lines beginning with 'HG:' are removed.
HG: --
HG: user: Bryan O'Sullivan <bos@serpentine.com>
HG: branch 'default'
HG: changed hello.c
```

Mercurial игнорирует строки, начинающиеся с «`HG:`». Он использует их только для того, чтобы сообщить нам, изменения в каких файлах он запишет. Редактирование или удаление этих строк ни на что не повлияет.

## 2.7.3. Написание хорошего сообщения к коммиту ревизии

Команда **hg log** по умолчанию выводит только первую строку описания изменений. Поэтому комментарий лучше написать так, чтобы первая строка была отделена. Вот хороший пример плохого комментария:

```
changeset:   73:584af0e231be
user:        Censored Person <censored.person@example.org>
date:        Tue Sep 26 21:37:07 2006 -0700
summary:     include buildmeister/commondefs. Add exports.
```

Для оставшейся части описания ревизии нет жестких правил. Сам Mercurial не обрабатывает и не заботится о содержимом сообщения об изменениях, хотя в вашем проекте могут быть правила, предписывающие определённое форматирование.

Моё личное предпочтение — короткие, но содержательные комментарии, которые сообщают мне то, чего я не могу выяснить при беглом взгляде на вывод команды **hg log --patch**.

Если мы выполним команду **hg commit** без каких-либо аргументов, запишутся все изменения, которые мы сделали, как сообщил **hg status** и **hg diff**.

### Сюрприз для пользователей **Subversion**

Если мы явно не укажем имена файлов для ревизии, **hg commit** как и любая другая команда Mercurial будет действовать для всего рабочего каталога репозитория. Учтите это, если вы переходите из мира Subversion или CVS, где подобная команда работает только для текущего каталога и его подкаталогов.

## 2.7.4. Отмена публикации ревизии

Если вы передумаете публиковать изменения во время редактирования комментария, просто выйдите из редактора без сохранения изменяемого файла. Это не вызовет изменений ни в репозитории, ни в рабочем каталоге.

## 2.7.5. Полюбуемся на наше творение

Закончив публикацию ревизии, мы можем воспользоваться командой **hg tip** для показа только что созданного набора изменений. Вывод этой команды похож на вывод команды **hg log**, но отображает только последнюю версию в репозитории.

```
$ hg tip -vp
changeset:   5:c3e3be994861
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:17 2012 +0000
files:       hello.c
description:
Added an extra line of output


diff -r 2278160e78d4 -r c3e3be994861 hello.c
--- a/hello.c Sat Aug 16 22:16:53 2008 +0200
+++ b/hello.c Thu Feb 02 14:10:17 2012 +0000
@@ -8,5 +8,6 @@
 int main(int argc, char **argv)
```

```
 {
   printf("hello, world!\");
+  printf("hello again!\n");
   return 0;
 }
```

Мы называем последнюю ревизию конечной (или верхней) ревизией или просто главной.

Кстати, команда **hg tip** принимает многие из опций команды **hg log**. Так, например, `-v` означает «более подробно», `-p` — «показать патч». Использование `-p` для показа патча является еще одним примером последовательного подхода к именованию опций.

# 2.8. Распространение изменений

Ранее мы упоминали, что репозитории в Mercurial самодостаточны. Это значит, что только что созданный набор изменений существует лишь в нашем репозитории `my-hello`. Давайте рассмотрим несколько способов, которыми мы можем распространить это изменение в другие репозитории.

## 2.8.1. Получение (вытягивание) изменений из другого репозитория

Для начала клонируем наш исходный репозиторий `hello`, в котором нет последнего изменения, только что нами опубликованного. Назовём наш временный репозиторий `hello-pull`.

```
$ cd ..
$ hg clone hello hello-pull
updating to branch default
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Мы будем использовать команду **hg pull**, для получения изменений из `my-hello` в `hello-pull`. Однако вытягивание вслепую неизвестных изменений в репозиторий может быть пугающей перспективой. Mercurial позволяет узнать с помощью команды **hg incoming**, какие изменения команда **hg pull** вытянет в репозиторий без реального применения изменений.

```
$ cd hello-pull
$ hg incoming ../my-hello
comparing with ../my-hello
searching for changes
changeset:   5:c3e3be994861
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:17 2012 +0000
summary:     Added an extra line of output
```

Получение изменений в репозиторий означает выполнение команды **hg pull** и указание ей, из какого репозитория следует вытягивать.

```
$ hg tip
changeset:   4:2278160e78d4
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:16:53 2008 +0200
summary:     Trim comments.

$ hg pull ../my-hello
pulling from ../my-hello
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
(run 'hg update' to get a working copy)
$ hg tip
```

```
changeset:   5:c3e3be994861
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:17 2012 +0000
summary:     Added an extra line of output
```

Как видно из вывода команды **hg tip** до и после, мы успешно вытянули изменения в наш репозиторий. Тем не менее, вытягивание изменений в Mercurial отделено от обновления рабочей директории. Остается один шаг до того, как мы увидим изменения в рабочем каталоге.

### Вытягивание конкретных изменений

Вполне возможно, что из-за задержки между выполнением команд **hg incoming** и **hg pull**, вы можете увидеть не все ревизии, которые будут добавлены из другого репозитория. Предположим, вы тянете изменения из репозитория где-то в сети. Пока вы смотрите вывод **hg incoming**, перед тем как вытащить изменения, кто-то может что-то совершить в удаленном репозитории. То есть можно вытянуть больше изменений, чем вы видели при использовании **hg incoming**.

Если вы хотите получить именно те изменения, которые были указаны в **hg incoming**, или вам требуется некоторое подмножество изменений, просто укажите id ревизии, которую хотите получить, например **hg pull -r7e95bb**.

## 2.8.2. Обновление рабочего каталога

До сих пор мы говорили о связи между репозиторием и его рабочим каталогом. Команда **hg pull**, которую мы выполнили в Раздел 2.8.1, «Получение (вытягивание) изменений из другого репозитория», вытянула изменения в наш репозиторий. Но если проверить, в рабочем каталоге нет ни следа этих изменений. Это потому, что **hg pull** не трогает (по умолчанию) рабочий каталог. Для применения изменений мы используем команду **hg update**.

```
$ grep printf hello.c
 printf("hello, world!\");
$ hg update tip
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ grep printf hello.c
 printf("hello, world!\");
 printf("hello again!\n");
```

Может показаться странным то, что **hg pull** не обновляет рабочий каталог автоматически. На самом деле этому есть веская причина: вы можете использовать **hg update**, для обновления рабочего каталога до состояния, в котором он был в любой ревизии в истории репозитория. Если бы вы обновили рабочий каталог до старой версии — например, чтобы отыскать причину ошибки, — а затем выполнили бы **hg pull**, которая обновляет рабочий каталог автоматически до новой версии, вы были бы ужасно недовольны этим.

Однако, так как вытягивание с последующим обновлением является распространенным явлением, Mercurial позволяет вам совместить их передачей команде **hg pull** ключа `-u`.

Если вы снова посмотрите на вывод команды **hg pull** в Раздел 2.8.1, «Получение (вытягивание) изменений из другого репозитория», когда мы выполнили её без `-u`, вы увидите, что в конце вывода было полезное напоминание о необходимости явного действия для обновления рабочего каталога.

Чтобы узнать, какая ревизия у рабочего каталога, используйте команду **hg parents**.

```
$ hg parents
changeset:   5:c3e3be994861
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:17 2012 +0000
summary:     Added an extra line of output
```

Если вы снова взглянете на Рисунок 2.1, «Графическое представление истории репозитория `hello`», вы увидите стрелки, соединяющие между собой каждую последующую ревизию. Вершина, из которой в каждом случае ведёт

стрелка, — родитель, а та вершина, куда стрелка ведёт, — потомок. Аналогично, у рабочего каталога есть родитель — это набор изменений, который содержится в данный момент в рабочем каталоге.

Чтобы обновить рабочий каталог до конкретной ревизии, передайте номер или идентификатор ревизии команде **hg update**.

```
$ hg update 2
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ hg parents
changeset:   2:fef857204a0c
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:05:04 2008 +0200
summary:     Introduce a typo into hello.c.

$ hg update
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ hg parents
changeset:   5:c3e3be994861
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:17 2012 +0000
summary:     Added an extra line of output
```

Если вы опустите явное указание ревизии, **hg update** обновит до верхней ревизии, как видно в примере выше при втором вызове **hg update**.

## 2.8.3. Передача (проталкивание) изменений в другой репозиторий

Mercurial позволяет передать изменения в другой репозиторий из репозитория, в котором мы в данный момент находимся. Как с примером **hg pull** выше, создадим временный репозиторий для передачи в него наших изменений.

```
$ cd ..
$ hg clone hello hello-push
updating to branch default
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Команда **hg outgoing** сообщает нам об изменениях, которые будут переданы в другой репозиторий.

```
$ cd my-hello
$ hg outgoing ../hello-push
comparing with ../hello-push
searching for changes
changeset:   5:c3e3be994861
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:17 2012 +0000
summary:     Added an extra line of output
```

И команда **hg push** вызывает настоящую передачу.

```
$ hg push ../hello-push
pushing to ../hello-push
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
```

Как и **hg pull**, команда **hg push** не обновляет рабочую директорию репозитория, в который передаются изменения. (В отличие от **hg pull**, у **hg push** нет ключа `-u`, который обновлял бы рабочую директорию другого репозитория). Эта асимметрия является преднамеренной: репозиторий в который мы передаём изменения может находиться на удаленном сервере и распределяться между несколькими людьми. Если бы нам пришлось обновить свой рабочий каталог в то время как кто-то работал в нём, его работа была бы нарушена.

Что же произдойдёт, если мы попробуем получить или передать изменения, а в принимающем репозитории они уже есть? Ничего особенного.

```
$ hg push ../hello-push
pushing to ../hello-push
searching for changes
no changes found
```

## 2.8.4. Размещение по умолчанию

Когда мы клонируем репозиторий, Mercurial записывает расположение репозитория из которого мы делали клон в файле `.hg/hgrc` нового репозитория. Если мы не будем указывать место для **hg pull** (откуда) или для **hg push** (куда), эти команды будут использовать это место в качестве репозитория по умолчанию. Команды **hg incoming** и **hg outgoing** делают то же самое.

Если вы откроете файл `.hg/hgrc` в текстовом редакторе, вы увидите содержимое похожее на следующее.

```
[paths]
default = http://www.selenic.com/repo/hg
```

Возможно — и часто полезно — чтобы путь по умолчанию для **hg push** и **hg outgoing**, отличался от пути для **hg pull** и **hg incoming**. Мы можем так сделать, добавив запись `default-push` в секцию `[paths]` файла `.hg/hgrc`.

```
[paths]
default = http://www.selenic.com/repo/hg
default-push = http://hg.example.com/hg
```

## 2.8.5. Распространение изменений по сети

Команды, которые мы затронули в нескольких предыдущих разделах, не ограничиваются работой с локальными репозиториями. Любая из них работает таким же способом и через сеть — просто укажите команде URL вместо локального пути.

```
$ hg outgoing http://hg.serpentine.com/tutorial/hello
comparing with http://hg.serpentine.com/tutorial/hello
searching for changes
changeset:   5:c3e3be994861
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:17 2012 +0000
summary:     Added an extra line of output

```

В следующем примере мы могли бы увидеть, какие изменения мы можем передать в удалённый репозиторий, но в репозитории, очевидно, не разрешён приём изменений от анонимных пользователей.

```
$ hg push http://hg.serpentine.com/tutorial/hello
pushing to http://hg.serpentine.com/tutorial/hello
searching for changes
remote: ssl required
```

# 2.9. Начало нового проекта

Начать новый проект так же просто, как и использовать уже существующий. Команда **hg init** создает новый, пустой репозиторий Mercurial.

```
$ hg init myproject
```

Это просто создаст репозиторий с именем `myproject` в текущем каталоге.

```
$ ls -l
total 8
-rw-r----- 1 slava slava 47 Feb  2 14:09 goodbye.c
-rw-r----- 1 slava slava 45 Feb  2 14:09 hello.c
drwxr-x--- 3 slava slava 60 Feb  2 14:09 myproject
```

Можно сказать, что `myproject` это репозиторий Mercurial, потому что он содержит каталог `.hg`.

```
$ ls -al myproject
total 0
drwxr-x--- 3 slava slava  60 Feb  2 14:09 .
drwx------ 3 slava slava 140 Feb  2 14:09 ..
drwxr-x--- 3 slava slava 100 Feb  2 14:09 .hg
```

Если мы хотим добавить существующие файлы в репозиторий, мы копируем их внутрь рабочей директории, и с помощью команды **hg add** сообщаем Mercurial, что нужно начинать за ними следить.

```
$ cd myproject
$ cp ../hello.c .
$ cp ../goodbye.c .
$ hg add
adding goodbye.c
adding hello.c
$ hg status
A goodbye.c
A hello.c
```

После того как мы убедились, что проект выглядит хорошо, мы публикуем наши изменения.

```
$ hg commit -m 'Initial commit'
```

Старт нового проекта с Mercurial займёт всего несколько мгновений, и это является частью его привлекательности. Контроль версий стал сейчас настолько простым, что мы можем его использовать даже в маленьких проектах, в которых, возможно, не использовали бы более сложные инструменты.

# Глава 3. Экскурсия по Mercurial: слияние результатов работы

К этому моменту мы рассмотрели клонирование репозитория, внесение изменений в репозиторий и получение или передачу изменений из одного репозитория в другой. Следующим нашим шагом будет слияние изменений из независимых репозиториев.

## 3.1. Слияние потоков работы

Слияние — это основная часть работы с инструментами распределённого контроля версий. Вот несколько случаев, когда возникает необходимость объединить работу:

- Элис и Боб имеют собственные репозитории проекта, над которым они вместе работают. Элис пофиксила ошибку в своем репозитории, а Боб добавил новую фичу в своём. Они хотят чтобы общий репозиторий содержал и багфикс и новую фичу.

- Синтия часто работает в одном проекте одновременно над несколькими разными задачами, каждая из которых изолирована в собственном репозитории. Работая таким образом, приходится частенько производить слияние одной части работы с другой.

Поскольку слияние очень частая операция, Mercurial имеет простые средства её осуществления. Давайте рассмотрим процесс слияния. Начнем с еще одного клонирования репозитория (заметили насколько часто мы это делаем?) и внесения изменений в него.

```
$ cd ..
$ hg clone hello my-new-hello
updating to branch default
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd my-new-hello
# Make some simple edits to hello.c.
$ my-text-editor hello.c
$ hg commit -m 'A new hello for a new day.'
```

Теперь у нас есть две копии `hello.c` с разным содержимым. История изменения этих двух репозиториев тоже различается, как показано на Рисунок 3.1, «Расхождение историй репозиториев `my-hello` и `my-new-hello`». Вот копия нашего файла из одного репозитория:

```
$ cat hello.c
/*
 * Placed in the public domain by Bryan O'Sullivan.  This program is
 * not covered by patents in the United States or other countries.
 */

#include <stdio.h>

int main(int argc, char **argv)
{
 printf("once more, hello.\n");
 printf("hello, world!\");
 printf("hello again!\n");
 return 0;
}
```

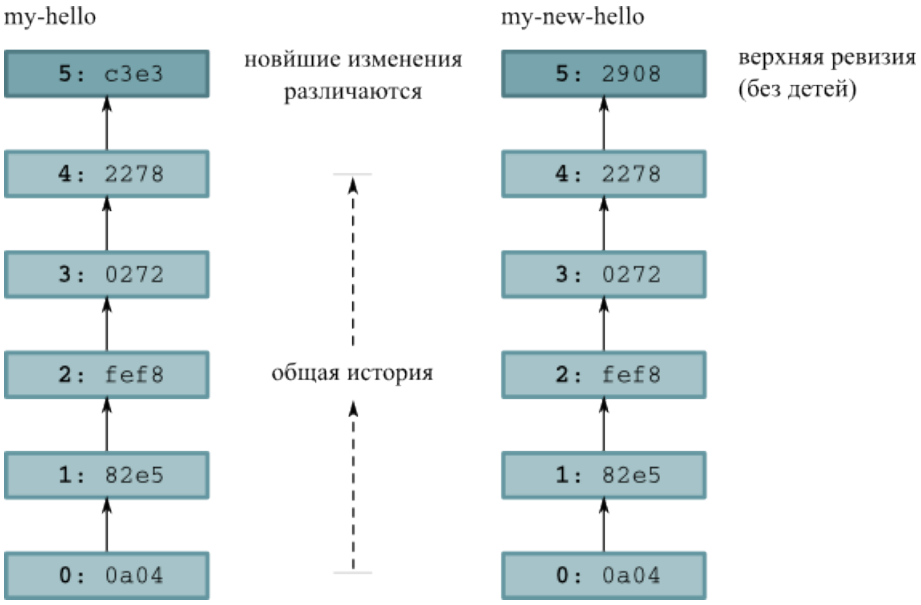
А здесь немного отличающаяся копия из другого хранилища:

```
$ cat ../my-hello/hello.c
/*
 * Placed in the public domain by Bryan O'Sullivan.  This program is
 * not covered by patents in the United States or other countries.
 */
```

```
#include <stdio.h>

int main(int argc, char **argv)
{
 printf("hello, world!\");
 printf("hello again!\n");
 return 0;
}
```

Рисунок **3.1.** Расхождение историй репозиториев `my-hello` и `my-new-hello`




Мы уже знаем, что получение изменений из репозитория `my-hello` не изменит состояния рабочего каталога.

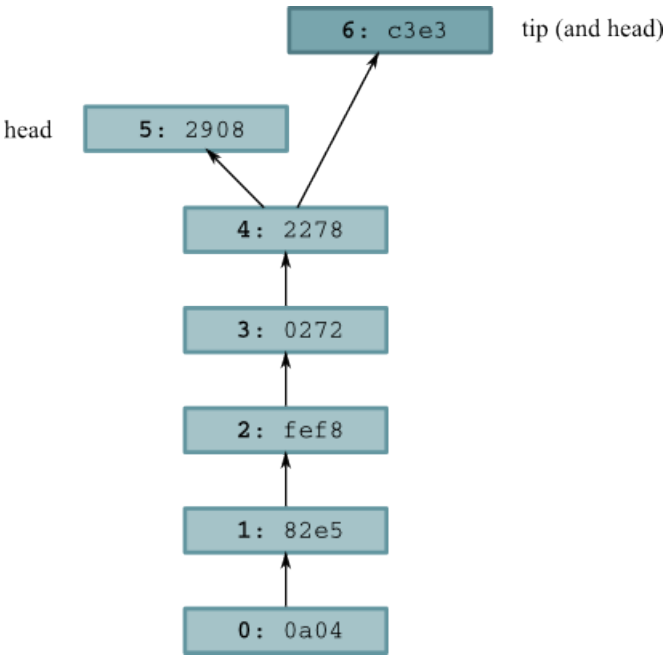
```
$ hg pull ../my-hello
pulling from ../my-hello
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
```

При этом команда **hg pull** говорит что-то о «головах» (heads), мол их стало на одну больше (+1 heads).

## 3.1.1. Головная ревизия

Напомним, что каждая ревизия в Mercurial, имеет родительскую ревизию. Если у ревизии есть родитель, мы называем её потомком. У головной (head) ревизии нет потомков. Главная (tip) ревизия тоже головная, потому что самая свежая ревизия в хранилище не может иметь потомков. Случаются моменты, когда репозиторий может содержать более одной головной ревизии.

Рисунок **3.2.** Содержимое хранилища **`my-new-hello`** после получения изменений из **`my-hello`**




На Рисунок 3.2, «Содержимое хранилища `my-new-hello` после получения изменений из `my-hello`» показан результат получения изменений из `my-hello` в `my-new-hello`. История, уже имеющаяся в `my-new-hello` не затронута, но добавлена новая ревизия. По сравнению с Рисунок 3.1, «Расхождение историй репозиториев `my-hello` и `my-new-hello`», видно, что **ID** ревизии остался прежним, а номер ревизии изменён. (Это, кстати, хороший пример, почему использование номеров ревизий при обсуждении наборов изменений не надёжно) Посмотреть головы в хранилище позволяет команда **hg heads**.

```
$ hg heads
changeset:   6:c3e3be994861
tag:         tip
parent:      4:2278160e78d4
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:17 2012 +0000
summary:     Added an extra line of output

changeset:   5:2908f9fcaad4
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:23 2012 +0000
summary:     A new hello for a new day.
```

## 3.1.2. Выполнение слияния

Что произойдёт, если мы попытаемся выполнить обычную команду **hg update** для обновления до новой головной ревизии?

```
$ hg update
abort: crosses branches (merge branches or update --check to force update)
```

Mercurial говорит нам, что команда **hg update** не выполняет слияния. Она думает, что мы ожидаем слияния, и не обновит рабочий каталог, если мы не принудим её к этому. (В данном случае, принудительное обновление с помощью **hg update -C** удалит все не сохраненные изменения в рабочем каталоге).

Для объединения двух голов мы воспользуемся командой **hg merge**.

```
$ hg merge
merging hello.c
```

```
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
```

Произошло слияние содержимого файла `hello.c`. Это привело к обновлению рабочего каталога — он теперь содержит изменения от обоих голов, что будет отражено в выводе **hg parents** и в содержании файла `hello.c`.

```
$ hg parents
changeset:   5:2908f9fcaad4
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:23 2012 +0000
summary:     A new hello for a new day.

changeset:   6:c3e3be994861
tag:         tip
parent:      4:2278160e78d4
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:17 2012 +0000
summary:     Added an extra line of output

$ cat hello.c
/*
 * Placed in the public domain by Bryan O'Sullivan.  This program is
 * not covered by patents in the United States or other countries.
 */

#include <stdio.h>

int main(int argc, char **argv)
{
 printf("once more, hello.\n");
 printf("hello, world!\");
 printf("hello again!\n");
 return 0;
}
```

## 3.1.3. Фиксация результатов слияния

Каждый раз, когда мы делаем слияние, **hg parents** показывает двух родителей пока мы не закрепим результат командой **hg commit**.

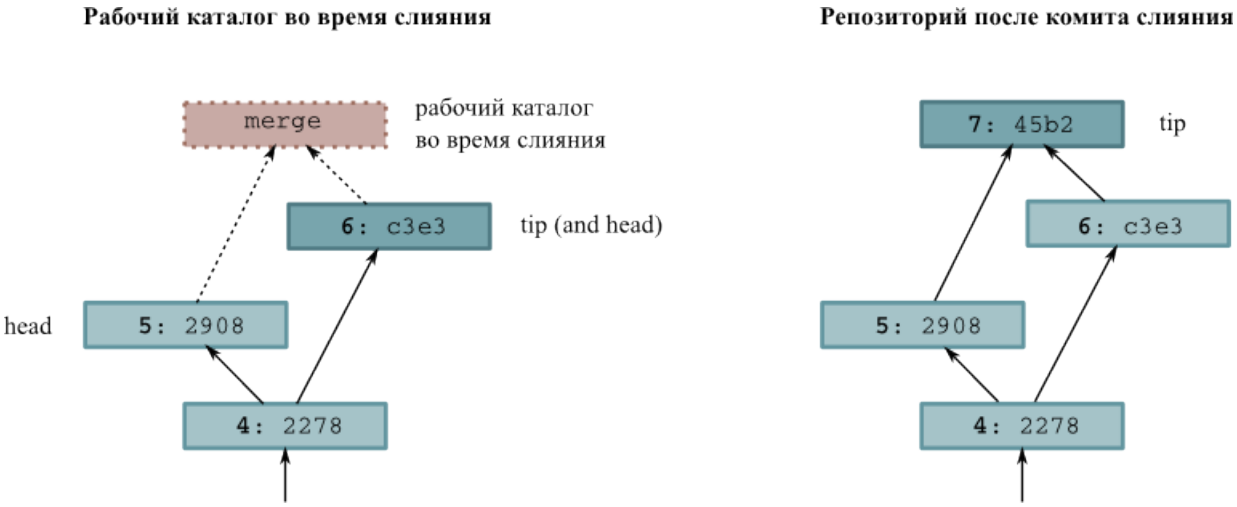
```
$ hg commit -m 'Merged changes'
```

Теперь у нас есть новая главная ревизия. Заметим, что обе бывшие головы теперь родители. Это те же ревизии, что раньше отображались командой **hg parents**.

```
$ hg tip
changeset:   7:45b2a02b6844
tag:         tip
parent:      5:2908f9fcaad4
parent:      6:c3e3be994861
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:24 2012 +0000
summary:     Merged changes
```

На Рисунок 3.3, «Рабочий каталог и репозиторий во время и после совершения слияния» вы можете увидеть представление того, что происходит с рабочим каталогом при слиянии, и как это влияет на хранилище, когда происходит коммит. Во время слияния, рабочий каталог состоял из двух родительских ревизий, и они стали родителями новой ревизии.

Рисунок **3.3.** Рабочий каталог и репозиторий во время и после совершения слияния




Иногда мы говорим о слиянии по сторонам: в левой части первый родитель, указанный в выводе **hg parents**, а в правой части — второй. Если до слияния рабочий каталог был таким как в ревизии 5, то ревизия будет с левосторонним слиянием.

# 3.2. Слияние конфликтующих изменений

Большая часть слияний проста, но иногда при слиянии возможны конфликты, когда участники изменили одинаковые части одного и того же файла. Если изменения не идентичны, то произойдет конфликт и вам придется решать, как согласовать изменения во что-то связное.

Рисунок **3.4.** Конфликт изменений в документе

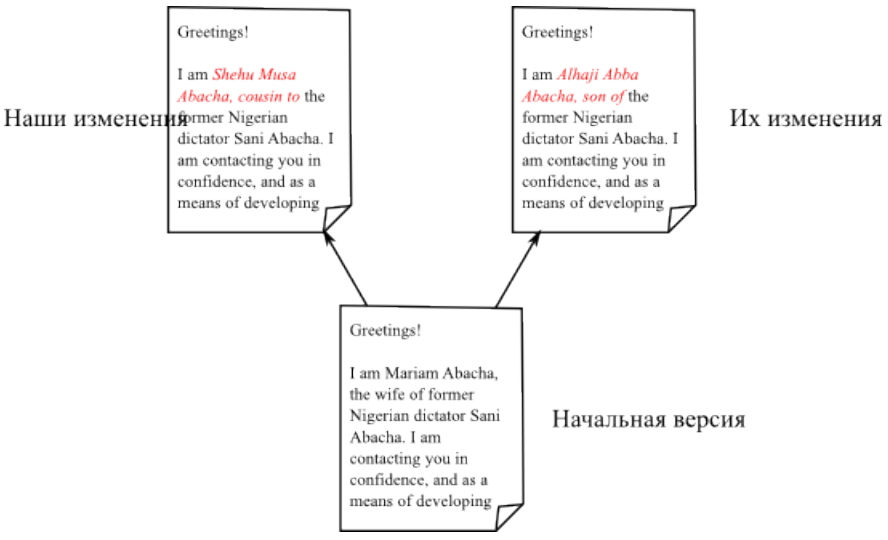



Рисунок 3.4, «Конфликт изменений в документе» показывает пример конфликта двух изменений в документе. Мы начали с одной версии файла, затем сделали несколько изменений, в то время, как кто-то другой также изменял этот текст. Наша задача в разрешении конфликта изменений — решить, как должен выглядеть окончательный вариант файла.

Mercurial не содержит встроенных средств обработки конфликтов. Вместо этого, он запускает внешнюю программу, обычно одну из графических утилит решения конфликтов. По умолчанию Mercurial пытается найти один из инструментов слияния, которые могут быть установлены в вашей системе. Вначале делается попытка слияния с помощью автоматических инструментов. Если это не удаётся (разрешить конфликт может только человек) или нет подходящего инструмента, сценарий пытается запустить один из графических инструментов.

Можно указать Mercurial использовать определённую программу, установив переменную окружения `HGMERGE` со значением имени необходимой программы.

## 3.2.1. Использование графического инструмента слияния

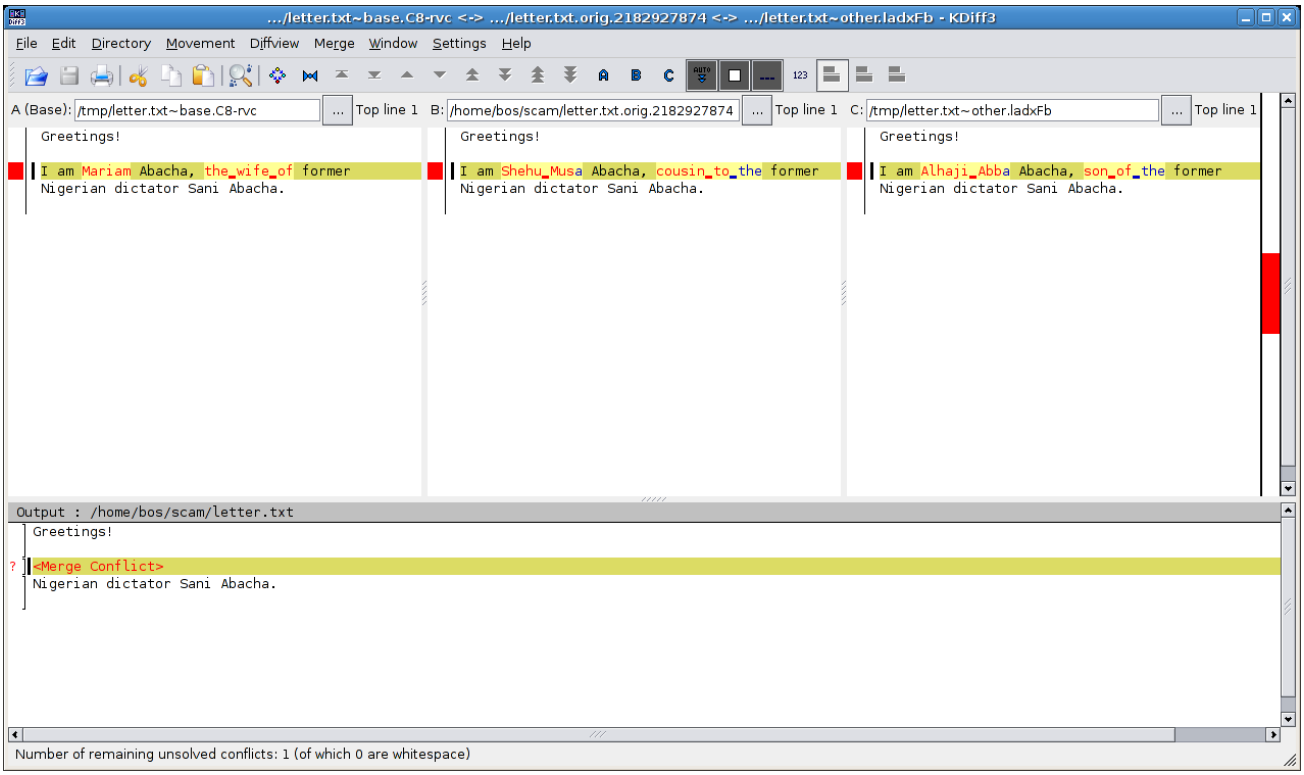
Мой любимый графический инструмент слияния это **kdiff3**, и его я буду использовать для описания возможностей, которые являются общими для графических инструментов слияния. На Рисунок 3.5, «Использование **kdiff3** для слияния версий файлов» показан снимок экрана **kdiff3** в работе. Выполняемое таким образом слияние называется тройственным (three-way), потому что есть три различные версии файла, интересующие нас. В инструменте сравнения верхняя часть окна поделена на три панели:

- Слева базовая версия файла, т.е. самая последняя версия, после которой произошло разделение на те две версии, которые мы пытаемся объединить.
- Посередине «наша» версия файла, содержащая наши изменения.
- Справа «их» версия файла, то есть версия из ревизии, с которой мы производим слияние.

На панели снизу располагается текущий результат слияния. Наша задача — заменить весь красный текст, означающий неразрешенные конфликты, на осмысленный результат слияния «нашей» и «их» версий файла.

Все четыре панели связаны друг с другом. Если мы начнем прокручивать любую из них по вертикали или по горизонтали, остальные панели последуют за нами и будут показывать соответствующие части файлов.

Рисунок **3.5.** Использование **kdiff3** для слияния версий файлов




Для каждого конфликтующего участка файла можно выбрать для разрешения конфликта любое сочетание текстов из базовой, нашей и их версий. Мы также можем вручную отредактировать результирующий файл в любое время, если требуются дополнительные изменения.

Существует множество инструментов слияния файлов, слишком много, чтобы их здесь описать. Они различаются доступностью для разных платформ и имеют свои слабые и сильные стороны. Большинство предназначены для слияния файлов, содержащих простой текст, но некоторые — для специализированных форматов (обычно XML).

## 3.2.2. Рабочий пример

В этом примере мы воспроизведем историю модификации файла с Рисунок 3.4, «Конфликт изменений в документе». Давайте начнем с создания пустого репозитория с базовой версией нашего документа.

```
$ cat > letter.txt <<EOF
> Greetings!
> I am Mariam Abacha, the wife of former
> Nigerian dictator Sani Abacha.
> EOF
$ hg add letter.txt
$ hg commit -m '419 scam, first draft'
```

Мы клонируем репозиторий и изменим файл.

```
$ cd ..
$ hg clone scam scam-cousin
updating to branch default
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd scam-cousin
$ cat > letter.txt <<EOF
> Greetings!
> I am Shehu Musa Abacha, cousin to the former
> Nigerian dictator Sani Abacha.
> EOF
$ hg commit -m '419 scam, with cousin'
```

Добавим еще одну копию и сымитируем, будто кто-то еще сделал изменение этого файла. Это намёк, что объединять свои же изменения — обычное дело, особенно когда вы разносите задачи по отдельным хранилищам, и вам нужно находить и разрешать конфликты между ними.

```
$ cd ..
$ hg clone scam scam-son
updating to branch default
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd scam-son
$ cat > letter.txt <<EOF
> Greetings!
> I am Alhaji Abba Abacha, son of the former
> Nigerian dictator Sani Abacha.
> EOF
$ hg commit -m '419 scam, with son'
```

Создав две разных версии файла, создадим окружение, в котором можно будет произвести наше объединение.

```
$ cd ..
$ hg clone scam-cousin scam-merge
updating to branch default
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd scam-merge
$ hg pull -u ../scam-son
pulling from ../scam-son
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
not updating: crosses branches (merge branches or update --check to force update)
```

В данном случае, я установил такое значение `HGMERGE`, чтобы Mercurial использовал консольную команду **merge**. Она встроена во многие Unix-подобные системы. Если вы выполняете этот пример на своём компьютере, то можете себя этим не утруждать. Вы просто передадите файл графическому инструменту для слияний, что гораздо предпочтительнее.

```
$ export HGMERGE=merge
$ hg merge
merging letter.txt
/bin/sh: merge: command not found
merging letter.txt failed!
0 files updated, 0 files merged, 0 files removed, 1 files unresolved
```

```
use 'hg resolve' to retry unresolved file merges or 'hg update -C .' to abandon
$ cat letter.txt
Greetings!
I am Shehu Musa Abacha, cousin to the former
Nigerian dictator Sani Abacha.
```

Так как **merge** не может самостоятельно выбрать правильное из противоречащих изменений, она оставляет маркеры слияния в файле с конфликтами, обозначая наши и их строки, содержащие противоречие.

Mercurial может определить, как завершилась **merge**, и если слияние не удалось, то он говорит, какие команды надо запустить, чтобы выполнить слияние по новой. Это может быть полезно, если мы запустили графическую утилиту объединения и вышли из нее, если что-то оказалось непонятно, или мы сделали ошибку.

Если автоматическое или ручное объединение не удалось, то ничто не мешает нам самим «поправить» пострадавшие файлы и закоммитить результаты слияния:

```
$ cat > letter.txt <<EOF
> Greetings!
> I am Bryan O'Sullivan, no relation of the former
> Nigerian dictator Sani Abacha.
> EOF
$ hg resolve -m letter.txt
$ hg commit -m 'Send me your money'
$ hg tip
changeset:   3:a31b9a7b1e74
tag:         tip
parent:      1:1504df74de25
parent:      2:203073c5900c
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:25 2012 +0000
summary:     Send me your money
```

### Где команда **hg resolve?**

Команда **hg resolve** была добавлена в Mercurial 1.1, выпущенный в декабре 2008 года. Если вы используете старую версию Mercurial (запустите **hg version**, чтобы узнать номер версии), эта команда вам недоступна. Если вы используете Mercurial версии ниже 1.1, вам следует подумать об обновлении прежде, чем пытаться решать сложные слияния.

## 3.3. Упрощение последовательности pull-merge-commit

Процесс слияния изменений, как говорилось выше, прост, но требует выполнения последовательности из трёх команд.

```
hg pull -u
hg merge
hg commit -m 'Merged remote changes'
```

В случае финального коммита вам также необходимо ввести комментарий, который в большинстве случаев — кусок неинтересного «стереотипного» текста.

Хорошо было бы, по возможности, сократить количество шагов. И действительно, Mercurial поставляется с расширением `fetch`, которое делает именно это.

Mercurial имеет гибкий механизм расширений, который позволяет расширять функциональность, оставляя ядро Mercurial небольшим и легким для использования. Некоторые расширения добавляют новые команды, которые вы можете использовать из командной строки, другие работают «за кулисами» — например, расширение, добавляющее возможности во встроенный в Mercurial сервер.

Расширение `fetch` добавляет новую команду **hg fetch**. По сути, это комбинация команд **hg pull -u**, **hg merge** и **hg commit**. Выполнение команды начинается с получения изменении из необходимого репозитория в текущий.

Если находятся изменения, добавляющие новую голову в репозиторий, то начинается слияние, затем, если слияние прошло успешно, происходит коммит результата с автоматической генерацией комментария. Если новых голов не было, расширение просто обновляет рабочую директорию.

Подключить расширение `fetch` просто: откройте в текстовом редакторе файл `.hgrc`, и добавьте в секцию `extensions` строку «`fetch=`», либо сначала создайте такую секцию.

```
[extensions]
fetch =
```

Обычно с правой стороны от «`=`» указывается местоположение расширения, но так как расширение `fetch` входит в стандартный пакет установки, Mercurial знает, где его искать.

# 3.4. Переименование, копирование и слияние

Во время жизни проекта мы будем часто изменять структуру своих файлов и каталогов. Это может быть такое простое изменение, как переименование файла, или же сложное, как перестройка всей иерархии файлов в рамках проекта.

Mercurial свободно поддерживает такого рода сложные изменения, при условии, что мы сообщаем ему о том, что делаем. Если мы хотим переименовать файл, мы должны использовать команду **hg rename**[1]. Команда переименует его, так что Mercurial будет знать что делать позже при слиянии.

Мы расскажем об использовании этих команд более подробно в Раздел 5.3, «Копирование файлов».

[1]Если вы пользователь Unix, вы будете рады узнать, что команда **hg rename** может быть сокращена, как **hg mv**.

# Глава 4. За кулисами

В отличии от многочисленных систем контроля версий концепция Mercurial достаточно проста, чтобы понять, как в действительности работает программа. Эти знания, безусловно, не являются необходимыми, но я считаю, что полезно иметь «мысленную модель» того, что происходит.

Понимание того, что происходит за кулисами, приводит к пониманию, что Mercurial тщательно спроектирован для безопасной и эффективной работы. И что не менее важно, если иметь представление о том, что делает программа, когда я выполняю какую-то задачу по контролю версий, меня не будет удивлять её поведение.

В этой главе мы сначала узнаем о внутреннем устройстве Mercurial, а затем обсудим интересные детали его реализации.

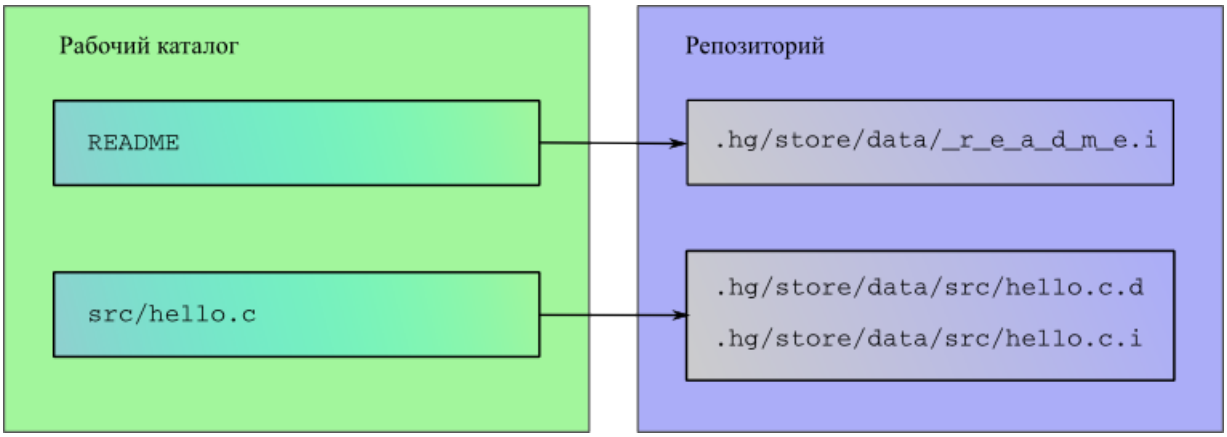
## 4.1. Запись истории в Mercurial

### 4.1.1. Отслеживание истории одного файла

Когда Mercurial отслеживает изменения файла, он сохраняет историю этого файла в объекте метаданных называемом **filelog**. Каждая запись в filelog содержит достаточно информации чтобы восстановить одну ревизию отслеженного файла. Filelog'и хранятся в виде файлов в папке `.hg/store/data`. Они содержат два вида информации: данные о ревизиях и индексы, помогающие Mercurial эффективно искать ревизии.

Для большого или имеющего длинную историю файла filelog хранится раздельно в файлах с данными (расширение «`.d`») и с индексом (расширение «`.i`»). Для маленьких файлов с небольшой историей ревизионные данные и индекс хранятся в едином файле с расширением «`.i`». Связь между файлом в рабочей директории и filelog'ом, который отслеживает его историю в хранилище, показана на Рисунок 4.1, «Связь между файлами в рабочей директории и filelog'ом в репозитории».

Рисунок **4.1.** Связь между файлами в рабочей директории и filelog'ом в репозитории




### 4.1.2. Управление отслеживаемыми файлами

Для сбора и хранения информации об отслеживаемых файлах Mercurial использует структуру называемую манифест. Каждый раздел в манифесте содержит информацию о файлах, входящих в один набор изменений. В разделе записано, какие файлы присутствуют в наборе изменений, ревизия каждого файла и некоторые другие части метаданных файла.

### 4.1.3. Запись информации о ревизиях

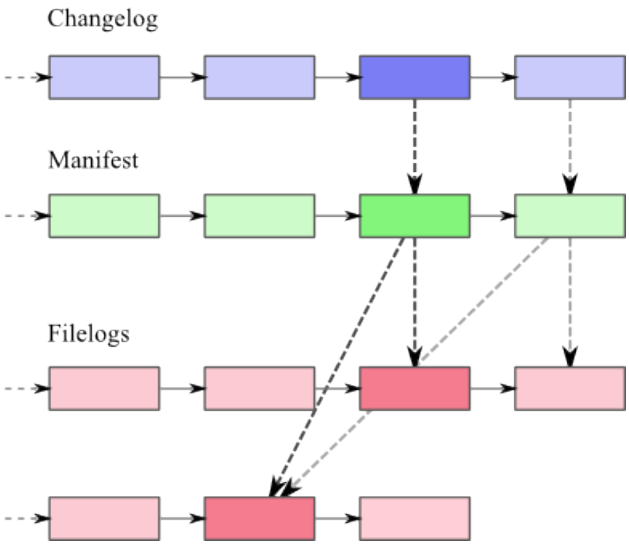
Журнал изменений содержит информацию о каждой ревизии. В каждую ревизию записывается, кто передал изменение, комментарий ревизии, другую связанную с этой ревизией информацию и манифест ревизии для дальнейшего использования

## 4.1.4. Зависимости между ревизиями

Внутри журнала изменений, манифеста или filelog'а каждая ревизия хранит указатель на своего непосредственного родителя (или на двух родителей, если эта ревизия получена при слиянии). Как я упоминал выше, есть также отношения между ревизиями через эти структуры, и они являются иерархическими по природе.

Каждому набору изменений в репозитории соответствует строго одна ревизия, хранящаяся в журнале изменений. Каждая ревизия в журнале изменений указывает на единственную ревизию в манифесте. Ревизия в манифесте указывает на единственную ревизию каждого filelog'а, отслеживающего, когда эта ревизия была создана. Эти взаимосвязи отражены на Рисунок 4.2, «Взаимосвязь метаданных».

Рисунок **4.2.** Взаимосвязь метаданных



Как показывает рисунок, между ревизиями в журнале изменений, манифесте и filelog'е нет отношений вида «один к одному». Если манифест не изменился между двумя наборами изменений, записи журнала изменений для этих наборов укажут на ту же самую ревизию манифеста. Если файл, который Mercurial отслеживает не изменился между двумя наборами, разделы для этого файла в двух ревизиях манифеста укажут на одну и ту же ревизию его filelog'а[1].

# 4.2. Безопасное и эффективное хранилище

Взаимосвязи журналов изменений, манифестов и filelog'ов обеспечены единой структурой, названной журнал ревизий (revlog).

## 4.2.1. Эффективное хранилище

Журнал ревизий эффективно хранит ревизии используя дельта-механизм. Вместо хранения полной копии файла для каждой ревизии он содержит изменения необходимые для преобразования из старой ревизии в новую. Для многих типов файлов данных эти дельты составляют менее одного процента от размера полной копии файла.

Некоторые устаревшие системы контроля ревизий могут работать только с дельтами текстовых файлов. Они должны хранить бинарные файлы как полные копии или перекодировать их в текстовое представление, но оба этих подхода слишком расточительны. Mercurial эффективно обращается с дельтами файлов с произвольным бинарным содержимым; ему не нужно рассматривать текст как нечто особое.

## 4.2.2. Безопасность работы

Mercurial только дописывает данные в конец revlog-файла. Он никогда не изменяет секцию файла после того, как файл был записан. Это более здравый и эффективный подход чем схемы, которые должны изменить или переписать данные.

Кроме того, каждое обращение Mercurial'а пишется как часть транзакции, которая может охватить много файлов. Транзакция является атомарной: или вся транзакция проходит, и все ее эффекты сразу видны читателям, или же вся транзакция не происходит. Эта гарантия атомарности означает, что, если Вы управляете двумя копиями Mercurial'а, где один читает данные и один пишет их, читатель никогда не будет видеть частично записанный результат, который мог бы запутать его.

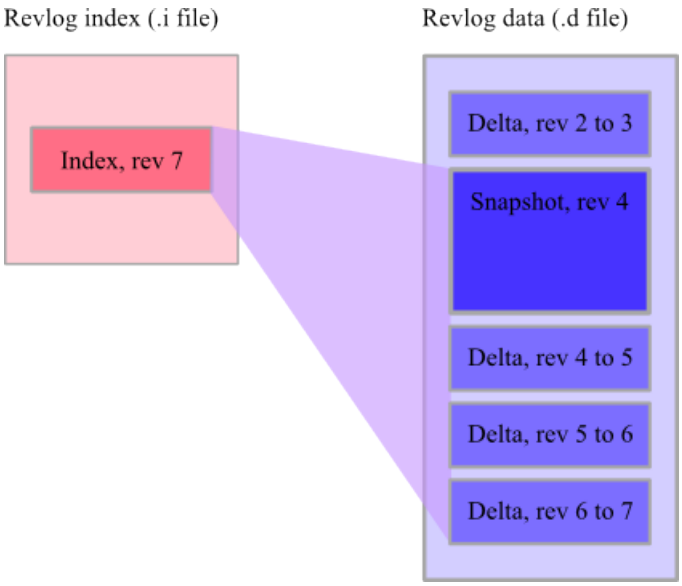
Тот факт что Mercurial оперирует только с файлами делает гарантированное выполнение транзакции более простым. Чем проще он выполняет подобные действия, тем вы можете более уверенными, что все выполнено корректно.

## 4.2.3. Быстрый поиск

Mercurial грамотно обходит скользкое место, обычное для всех более ранних систем контроля версий — проблему неэффективного восстановления версий. Большинство систем контроля версий хранят содержимое ревизий как возрастающие серии модификаций по сравнению с «моментальным снимком». Чтобы восстановить определенную ревизию, вам необходимо прочитать моментальный снимок, а затем все ревизии между ним и интересующий вас. Чем более длительную историю имеет файл, тем больше ревизий вам придется прочитать, следовательно, больше времени уйдет на восстановление отдельной ревизии.

[1] Возможно (хотя и необычно) манифест между двумя ревизиями останется неизменным, в этом случае запись лога изменений для ревизии будет указывать на ту же ревизию в манифесте.

Рисунок **4.3.** Моментальный снимок журнала изменений с возрастающими дельтами




Нововведение в Mercurial решает эту проблему просто, но эффективно. Когда суммарное количество информации в дельте по сравнению с последним снимком превышает заданный порог, Mercurial сохраняет новый снимок (сжатый, конечно), вместо следующей дельты. Это позволяет быстро восстановить любую ревизию файла. Такой подход работает настолько хорошо, что уже был скопирован несколькими другими системами контроля версий.

Рисунок 4.3, «Моментальный снимок журнала изменений с возрастающими дельтами» иллюстрирует идею. В разделе журнала изменений индексированного файла Mercurial сохраняет список разделов из файла с данными, который необходимо прочитать для восстановления определенной ревизии.

### 4.2.3.1. Отступление: влияние сжатия видео

Если вы знакомы со сжатием видео или когда-нибудь смотрели кабельное или спутниковое ТВ, то знаете, что в большинстве схем сжатия видео каждый кадр видео хранится как дельта от предыдущего кадра. Кроме того эти схемы используют сжатие с потерями для увеличения коэффициента сжатия, поэтому ошибки отображения накапливаются в зависимости от числа межкадровых дельт.

Mercurial использует эту идею для реконструкции ревизий из снапшотов и небольшого количества дельт.

## 4.2.4. Идентификация и надежная целостность

Вместе с дельтами или снимками информации, журнал изменений содержит криптографический хеш представленных данных. Это затрудняет подделку содержимого ревизии и облегчает обнаружение случайной порчи данных.

Хеши нужны не только для защиты от искажений — они используются и как идентификаторы ревизий. Хеши идентифицирующие наборы изменений, которые Вы видите как конечный пользователь, это хеши от ревизий журнала изменений. Хотя filelog'и и манифесты также используют хеши, Mercurial задействует их только во внутренних процессах.

Mercurial проверяет правильность хешей при восстановлении ревизий файлов и при перемещении изменений из другого репозитория. При обнаружении проблемы целостности, он пожалуется и остановит текущие действия.

В дополнение к эффективности поиска, использование Mercurial’ом периодических снимков делает его более устойчивым против частичного нарушения целостности данных. Если журнал ревизий частично поврежден из-за аппаратной или системной ошибки, часто возможно восстановить некоторые или большинство ревизий от неповрежденных секций журнала ревизий и перед, и после поврежденной секции. Это не было бы возможно с моделью хранения только дельты.

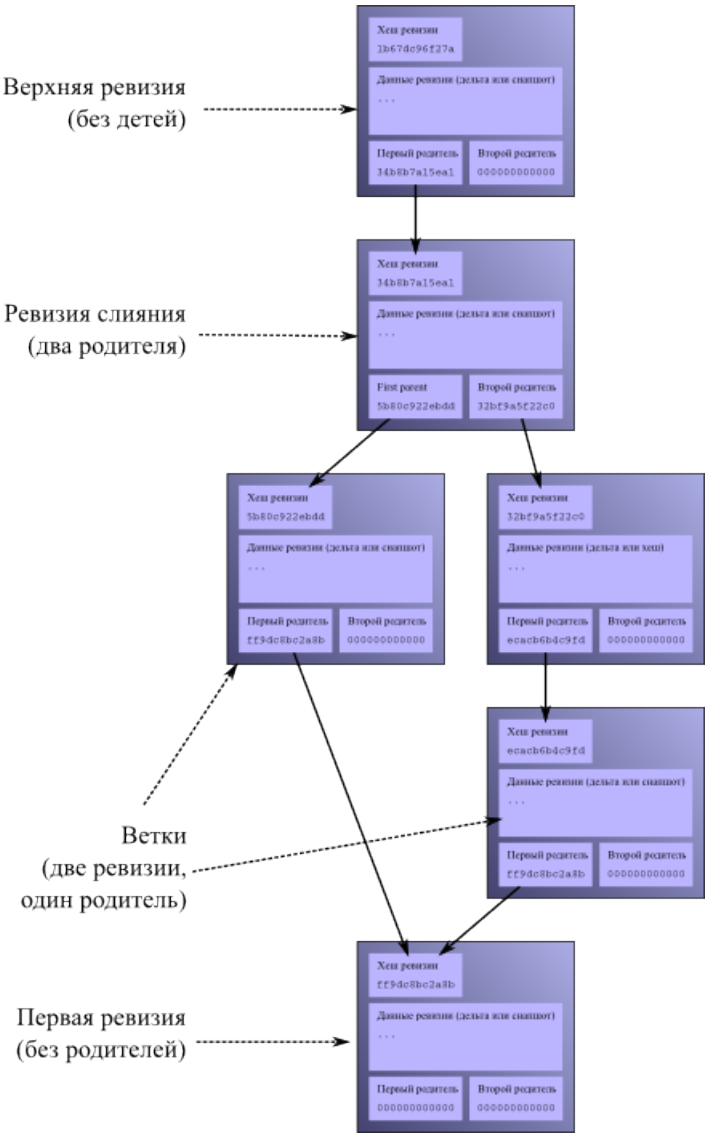
# 4.3. История ревизий, ветвление и слияние

Каждый раздел в журнале ревизий Mercurial точно соотносится со своей непосредственной ревизией-предком, обычно называемой родителем. Фактически, ревизия содержит место не для одного родителя, а для двух. Mercurial использует специальный хеш, называемый «нулевым идентификатором» для обозначения «нет здесь никакого родителя». Этот хеш — просто строка нулей.

На Рисунок 4.4, «Общий вид структуры журнала ревизий», вы можете видеть пример общего представления структуры журнала ревизий. Filelog`и, манифесты и журналы изменений имеют такую же структуру, они отличаются только видом данных, хранящихся в дельтах и снимках.

Первая ревизия в журнале (изображена внизу рисунка) имеет нулевые идентификаторы для обоих родителей. Для «нормальной» ревизии в слоте для одного родителя указан идентификатор ревизии-родителя, а второй слот содержит нулевой идентификатор, показывающий что у ревизии только один реальный родитель. Любые две ревизии с одинаковыми идентификаторами родителей называются ветвями. Ревизия, представляющая собой слияние между ветками имеет два нормальных идентификатора ревизий в родительских слотах.

Рисунок **4.4.** Общий вид структуры журнала ревизий



# 4.4. Рабочий каталог

В рабочем каталоге Mercurial хранит точную копию файлов из репозитория в соответствии с одной из ревизий.

Рабочий каталог «знает» какую из ревизий он содержит. Когда вы обновляете рабочий каталог до определенной ревизии, Mercurial просматривает соответствующую ревизию манифеста и находит там, какие файлы он отслеживал в то время, когда была создана ревизия, и какая из ревизий каждого из этих файлов была текущей. Потом он пересоздает копию каждого из этих файлов с содержимым на момент фиксации ревизии.

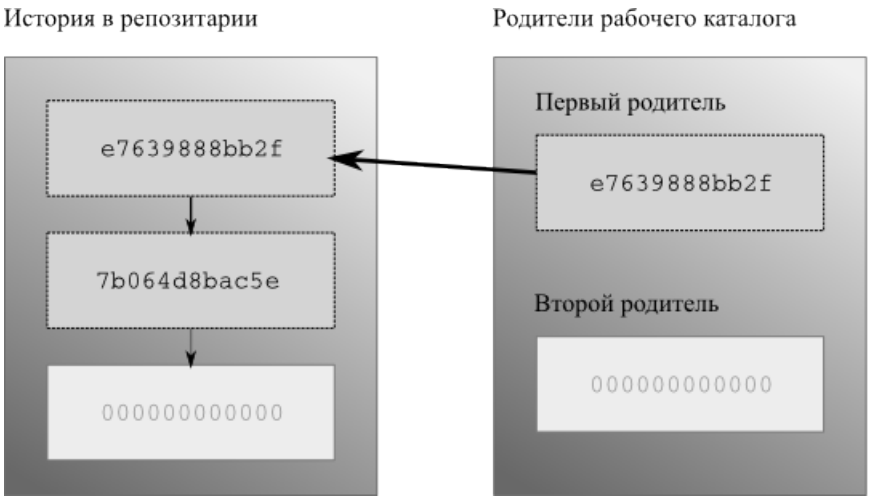
**Dirstate** это особая структура, которая содержит знания о рабочий каталоге Mercurial. Она содержится в файле с именем `.hg/dirstate` внутри хранилища. Dirstate детализирует ревизию которая содержит в рабочем каталоге и все файлы, которые Mercurial отслеживает в рабочей директории. Он также позволяет Mercurial быстро сообщает об измененных файлах, записывая их время вытягивания и размеры.

Так же, как и у ревизии в revlog есть место для двух родителей (обычная ревизия — с одним родителем, и результат слияния двух ревизий — с двумя родителями), у dirstate тоже есть слоты для двух родителей. Когда вы выполняете команду **hg update**, ревизия, которую вы обновляете сохраняется в слоте «первого родителя», а во второй слот помещается нулевое значение. Когда вы выполняете **hg merge** с другой ревизией, первый родитель остается неизменным, а вторым родителем становится ревизия с которой вы осуществляете слияние. Команда **hg parents** показывает родителей текущего состояния каталога.

## 4.4.1. Что происходит, когда вы фиксируете изменения (commit)

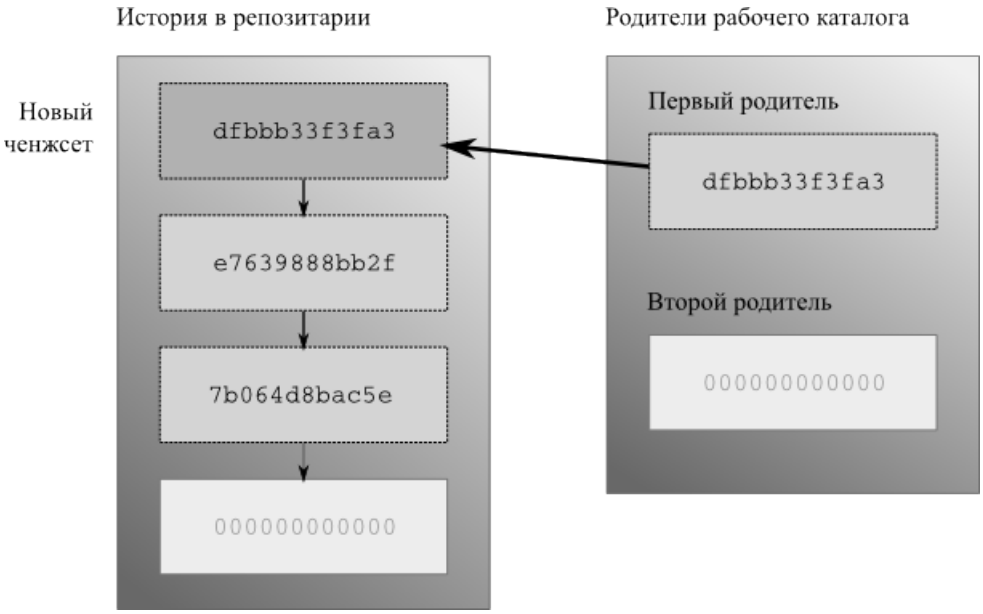
Dirstate хранит информацию о родительских ревизиях не только для своих целей. Mercurial сохраняет родительские ревизии состояния рабочего каталога как родителей новой ревизии во время коммита.

Рисунок **4.5.** Рабочий каталог может иметь две родительские ревизии




На Рисунок 4.5, «Рабочий каталог может иметь две родительские ревизии» показано нормальное состояние рабочего каталога, с одной ревизией в качестве родителя. Эта ревизия — окончание ветки (**tip**) — самая последняя ревизия в репозитории, у которой нет детей.

Рисунок **4.6.** После коммита у рабочего каталога появляются новые родители



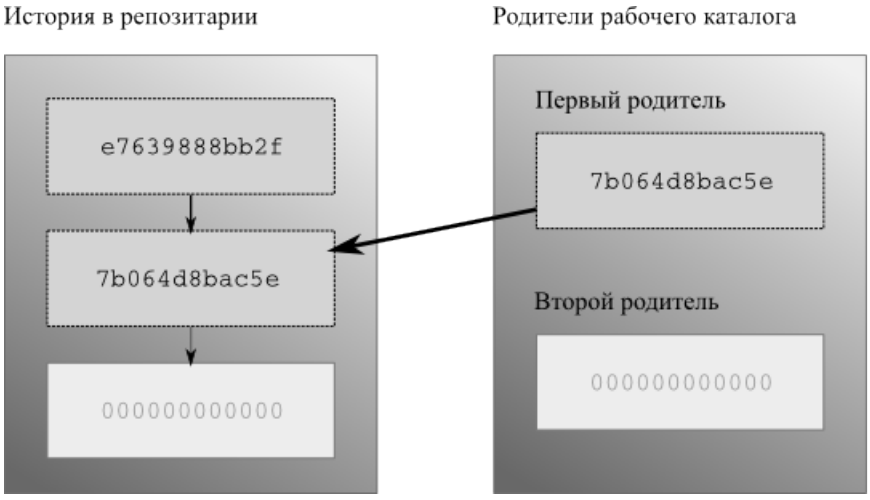

Полезно думать о рабочем каталоге, как о «ревизии, которую я сейчас зафиксирую». Любая операция с файлами (добавление, удаление, переименование, копирование), о которой вы сообщили Mercurial, будет отражена в этой ревизии, так же как и изменение в тех файлах, состояние которых он уже отслеживает. У новой ревизии будут те же родители, что у рабочего каталога.

После фиксации Mercurial обновит родителей рабочего каталога таким образом, что первым родителем будет идентификатор новой ревизии, а вторым — нулевой идентификатор. Это показано на Рисунок 4.6, «После коммита у рабочего каталога появляются новые родители» Mercurial не изменяет файлы в рабочем каталоге при фиксации изменений, он всего лишь изменяет dirstate, чтобы запомнить новых родителей.

## 4.4.2. Создание новой головы (head)

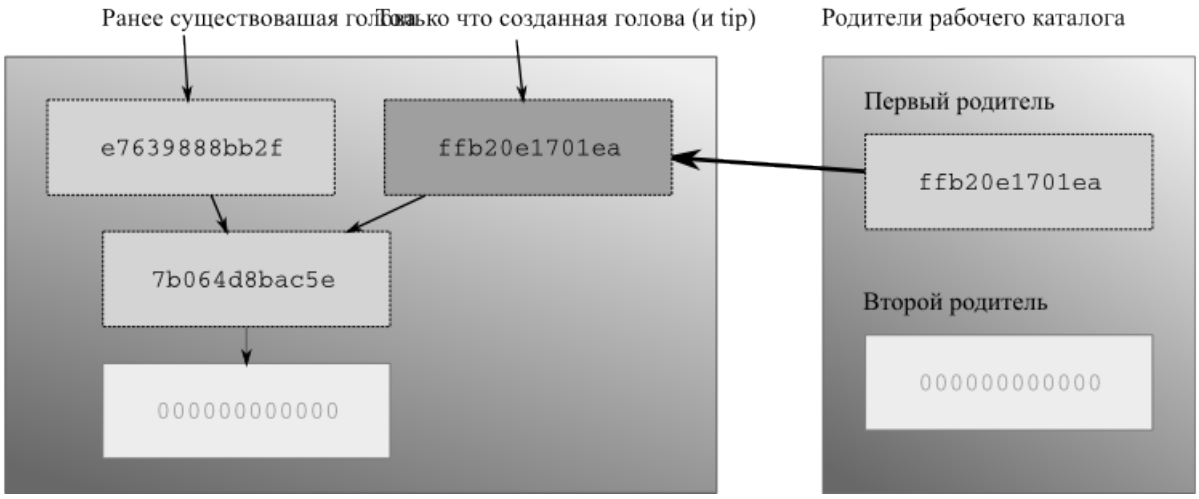
Абсолютно нормально обновлять текущий каталог до ревизии, которая не является последней. Например, вы хотите знать, как выглядел ваш проект в прошлый вторник, или вы просматриваете ревизии, чтобы найти ту, в которой появилась ошибка. В подобных случаях обычное дело обновить рабочий каталог до интересующей ревизии и потом просматривать содержимое файлов в нём в том состоянии, в каком они были во время фиксации ревизии. Результат этих действий показан на Рисунок 4.7, «Рабочий каталог, обновленный до ранней ревизии».

Рисунок **4.7.** Рабочий каталог, обновленный до ранней ревизии



Что произойдет, если мы сделаем изменения в то время, как рабочий каталог обновлен до более старой ревизии, а потом зафиксируем их? Mercurial поступит точно так, как я и говорил раньше. Родители рабочего каталога станут родителями новой ревизии. У новой ревизии не будет детей, так что она станет конечной ревизией. И с этого момента в репозитории будет две ревизии без детей, мы называем их головами (head). Вы можете посмотреть на то, что получилось на Рисунок 4.8, «После фиксации, сделанной в то время, как рабочий каталог был обновлен до ранней ревизии.».

Рисунок **4.8.** После фиксации, сделанной в то время, как рабочий каталог был обновлен до ранней ревизии.




## Примечание

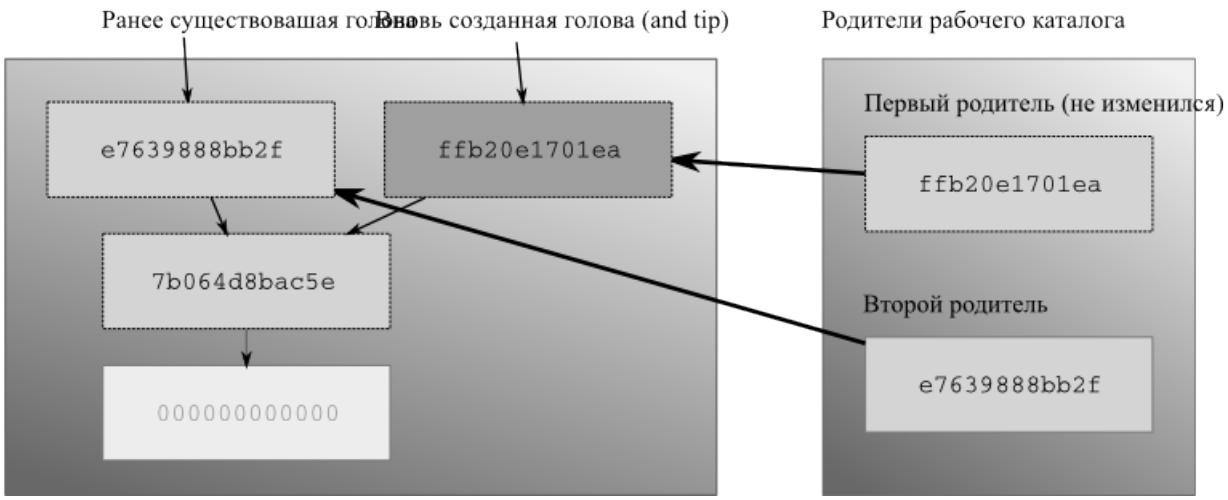
Если вы новичок в Mercurial, то вам следует помнить о распространенной «ошибке» — использовании команды **hg pull** без аргументов. По умолчанию, **hg pull** не обновляет рабочий каталог, так что вы получаете новые ревизии в репозиторий, но каталог соответствует той же ревизии, что и перед выполнением команды. Если вы сделаете изменения и зафиксируете их, то вы создадите новую голову, потому что рабочий каталог не соответствует ни одной концевой ревизии. Чтобы совместить операции вытягивания и последующего обновления, запустите команду **hg pull -u**.

Я написал «ошибка» в кавычках, потому что все что вам нужно сделать, чтобы исправить эту ситуацию — это выполнить **hg merge**, а потом **hg commit**. Другими словами, такая ситуация никогда не приводит к негативным последствиям, она просто приводит людей в замешательство. Позже мы обсудим пути обхода этой ситуации, а также почему Mercurial по умолчанию ведет себя таким образом.

## 4.4.3. Слияние изменений

Когда вы выполняете команду **hg merge**, Mercurial оставляет первого родителя рабочего каталога неизменным, а вторым родителем назначает ревизию, с которой вы осуществляете слияние (см. Рисунок 4.9, «Слияние двух голов»)

Рисунок **4.9.** Слияние двух голов




Mercurial также изменяет рабочий каталог, чтобы осуществить слияние файлов из двух ревизий. Немного упрощенно процесс слияния проходит следующим образом — для каждого файла в манифестах обеих ревизий:

- Если ни одна из ревизий не изменяла файл, то ничего с ним не делать
- Если одна ревизия изменила файл, а другая — нет, то создать измененную копию файла в рабочем каталоге
- Если одна ревизия удалила файл, а другая — нет (или тоже удалила его), то удалить файл из рабочего каталога
- Если одна ревизия удалила файл, а другая изменила его, то спросить пользователя что делать — оставить измененный файл или удалить его?
- Если обе ревизии изменили файл, то вызвать внешнюю программу для слияния, чтобы определить содержимое слитого файла. Эта операция может потребовать взаимодействия с пользователем.
- Если одна ревизия изменила файл, а другая переименовала или скопировала файл, то удостовериться, что изменения будут перенесены в файл с новым именем

На самом деле все сложнее — у слияния есть множество подводных камней, но вышеперечисленное — это основные решения, которые нужно принимать при слиянии. Как видите, большая часть из них полностью

автоматизирована, и таким образом, большая часть слияний завершается автоматически, без вмешательства пользователя для разрешения конфликтов.

Когда вы думаете о том, что произойдет, когда вы зафиксируете изменения после слияния, опять вспомните правило «рабочий каталог — это ревизия, которую я сейчас зафиксирую». После завершения команды **hg merge** у рабочего каталога будет два родителя, они же и станут родителями новой ревизии.

Mercurial позволяет выполнять несколько слияний, но вы должны фиксировать результаты каждого слияния после каждого слияния. Это необходимо потому, Mercurial отслеживает только два родителя для ревизии и рабочего каталога. Хотя было бы технически возможно объединить несколько ревизий сразу, Mercurial считает что проще этого избегать. С многоходовым слиянием, есть риск ввести пользователей в заблуждение, отвратительным урегулированием конфликтов, и страшный беспорядок при слияния будет расти.

### 4.4.4. Слияние и переименование

Удивительно ряд систем контроля версий практически не обращают внимания на изменении имени файла с течением времени. Так, например, это было общее, что если файл переименован то с одной стороны слияния изменения появятся, с другой стороны пропадут.

Mercurial записывает метаданные, когда вы говорите ему выполнить переименование или копирование. Он использует эти данные во время слияния, чтобы правильно делать слияние. Например, если я переименую файл, а вы правите без переименования, когда мы объединяем наши работы файл будет переименован и ваш изменения будут внесены.

## 4.5. Другие интересные дизайнерские решения

В предыдущих разделах, я попытался осветить некоторые из наиболее важных аспектов проектирования в Mercurial, чтобы показать, что он уделяет особое внимание надежности и производительности. Тем не менее, внимание к деталям, не останавливается на достигнутом. Есть целый ряд других аспектов построения Mercurial, которые мне лично кажутся интересными. Я подробно опишу некоторые из них, отдельно от «большого списка» пунктов выше, так что если вам интересно, вы можете получить более полное представление о наборе идей, которые используются в четко разработанной системе.

### 4.5.1. Умное сжатие

Когда нужно, Mercurial хранит и основные файлы и дельты в сжатой форме. Для этого он всегда пытается сжать файл или дельту, но хранит сжатую версию только в том случае, если она меньше оригинальной.

Это означает, что Mercurial «правильно» обрабатывает ситуацию, когда сохраняется файл, который уже сжат, например, `zip`-архив или JPEG-изображение. Когда подобные файлы пережимаются второй раз, они обычно больше оригинала, так что Mercurial сохраняет оригинальный `zip` или JPEG.

Дельты между ревизиями сжатых файлов тоже обычно больше, чем сами файлы, и тут Mercurial тоже поступает «правильно». Если он видит, что размер такой дельты превышает размер файла, то он сохраняет сам файл, что опять же дает экономию дискового пространства по сравнению хранением только дельты файлов.

#### 4.5.1.1. Сжатие при передаче по сети

При сохранении ревизий на диск, Mercurial использует алгоритм сжатия «deflate» (такой же, как и в популярном формате архивов `zip`), который сочетает хорошую скорость и приличный уровень сжатия. Однако, при передаче данных по сети, Mercurial распаковывает сжатые данные.

Если при передаче данных используется протокол HTTP, Mercurial переупаковывает весь поток данных целиком, используя алгоритм сжатия, который дает более высокую степень сжатия (алгоритм Burrows-Wheeler из архиватора `bzip2`). Комбинация из алгоритма и упаковки потока целиком (вместо сжатия каждой ревизии по отдельности) в значительной степени сокращает количество передаваемых данных, что в результате даёт лучшую производительность практически на любых видах сетевых подключений.

Если используется протокол **ssh**, то Mercurial не сжимает поток, потому что **ssh** делает это сам. Вы можете сказать, Mercurial, чтобы всегда использовать функцию сжатия в **ssh** редактированием файла `.hgrc` в вашем домашнем каталоге следующим образом.

```
[ui]
ssh = ssh -C
```

## 4.5.2. Порядок чтения/записи и атомартность

Добавление к файлам — ещё недостаточное условие, чтобы гарантировать, что читатель не увидит частичнозаписанные данные. Если Вы ещё раз посмотрите на Рисунок 4.2, «Взаимосвязь метаданных» ревизии в журнале изменений указывают на ревизии в манифесте, а ревизии в манифесте — на ревизии в filelog`ах. Эта иерархия является преднамеренной.

При записи транзакция начинается с записи данных filelog`а и манифеста, а в журнал изменений ничего не записывается до окончания работы с этими данными. Чтение же начинается с журнала изменений, потом читается данные манифеста, а потом — данные filelog`а.

С момента завершения записи в filelog и манифест и до записи в журнал изменений невозможно чтение ссылок на частично записанную ревизию манифеста из журнала изменений и на частичную ревизию filelog`а из манифеста.

## 4.5.3. Конкурентный доступ

Порядок чтения/записи и гарантии атомарности подразумевают, что Mercurial не нуждается в блокировке репозитория при чтении данных даже если паралельно с чтением происходит запись. Это оказывает большой эффект на масштабируемость; у Вас может быть произвольное число процессов Mercurial'а, безопасно читающих данные из репозитория одновременно, независимо от того записываются ли они него в или нет.

Неблокирующее чтение означает, что при использовании репозитория в многопользовательской системе вам не нужно наделять других локальных пользователей правами записи в ваш репозиторий для клонирования или вытягивания изменений из него; им будет достаточно прав чтения. (Это не общая черта среди систем контроля версий, так что не принимайте её, как очевидное! Большинство требует, чтобы читатели были в состоянии блокировать репозиторий для безопасного доступа к нему, а это требует прав записи по крайней мере для одного каталога, что, конечно, способствует всем видам противной и раздражающей безопасности и административным проблемам.)

Mercurial использует блокировки, чтобы гарантировать, что только один процесс может писать в репозиторий за один раз (механизм блокировки безопасен даже для файловых систем, которые известны своей враждебностью к блокировкам, такими как NFS). Если репозиторий будет блокирован, то программа записи будет ждать некоторое время, чтобы повторить попытку, если репозиторий будет разблокирован, но если репозиторий останется блокированным слишком долго, то процесс, пытающийся написать, будет остановлен по таймауту через некоторое время. Это означает, что Ваши ежедневные автоматизированные скрипты не будут застревать навсегда и накапливаться, если в системе незамеченный сбой, например. (Да, время ожидания настраивается, от нуля до бесконечности.)

### 4.5.3.1. Безопасный доступ к файлу dirstate

Как и в случае с ревизией данных, Mercurial не блокирует файл dirstate для чтения; файл блокируются только для записи. Чтобы избежать чтения частично записанной копии файла dirstate, Mercurial пишет в файл с уникальным именем в том же каталоге, что dirstate, а затем автоматически переименовывает временный файл в `dirstate`. Файл с именем `dirstate`, таким образом, гарантированно будет полным, а не частично сохраненным.

## 4.5.4. Предотвращение поиска секторов

Критичным для производительности Mercurial является предотвращение поиска сектора головкой диска, поскольку любой поиск требует гораздо больше накладных расходов, чем длительная операция чтения.

Вот почему, например, dirstate сохраняется в едином файле. Если бы этот файл сохранялся в каждом подкаталоге структуры, обрабатываемой Mercurial, операцию поиска пришлось бы делать по разу на каждый подкаталог. Вместо этого Mercurial читает единственный цельный файл dirstate за один шаг.

Mercurial также использует схему «копирование при записи», когда клонирует репозитарий в локальное хранилище. Вместо копирования каждого файла revlog из старого репозитария в новый создается «жесткая ссылка», которая является стенографическим способом сказать: «эти два имени указывают на один и тот же файл». Когда Mercurial готовится изменять один из файлов revlog, он проверяет, нет ли у этого файла нескольких имен. Если есть, — это значит, что больше одного репозитария используют данный файл, и Mercurial создает новую копию этого файла, приватную для данного репозитария.

Некоторые разработчики revision control отмечают, что эта идея по созданию полной частной копии файла не слишком эффективна с точки зрения использования свободного места. Хотя это и правда, хранение дешево, и этот метод дает наилучшую производительность вычислений на большинстве операционных систем. Альтернативные схемы, вероятнее всего, уменьшат производительность и увеличат сложность приложения, а обе эти характеристики весьма важны для «ощущений» от повседневного использования.

## 4.5.5. Другое содержимое dirstate

Поскольку Mercurial не требует от Вас сообщать ему, когда Вы изменили файл, он использует dirstate для сохранения некоторой дополнительной информации, помогающей ему эффективно определять, что файл изменен. Для каждого файла в рабочем каталоге сохранятся время последнего изменения, и размер файла в этот момент.

Когда вы явным образом производите операции **hg add**, **hg remove**, **hg rename** или **hg copy** с файлами, Mercurial обновляет dirstate и таким образом знает, что необходимо делать с этими файлами, когда Вы фиксируете изменения (commit).

Dirstate помогает Mercurial эффективно проверять статус файлов в репозитории.

- Когда Mercurial проверяет состояние файлов в рабочем каталоге, в первую очередь сравнивается время изменения файла и время записанное Mercurial в файл dirstate при последней фиксации файла. Если время последней модификации и время записанное при последней фиксации совпадают, то значит файл не должен был изменится, и Mercurial не проверяет этот файл дальше.

- Если размер файла изменился, файл должен был изменится. Если время модификации изменилось, а размер нет, только тогда Mercurial необходимо реально прочитать содержимое файла для того, чтоб посмотреть изменился ли он.

Сохранение времени модификации и размера поразительно уменьшает количество операций чтения, которые нужно совершать Mercurial при каждом запуске команд похожих на **hg status**. Такой результат приводит к значительному увеличению производительности.

# Глава 5. Повседневное использование Mercurial

## 5.1. Указание Mercurial, какие файлы необходимо отслеживать

Mercurial не работает с файлами в хранилище, пока вы не скажете ему, чтобы он управлял ими. Команда **hg status** покажет о каких файлах Mercurial не знает, он использует «?» для отображения таких файлов.

Чтобы сказать Mercurial отслеживать файлы, используйте команду **hg add**. После того, как вы добавили файл, запись в результатах **hg status** для этого файла изменится с «?» на «A».

```
$ hg init add-example
$ cd add-example
$ echo a > myfile.txt
$ hg status
? myfile.txt
$ hg add myfile.txt
$ hg status
A myfile.txt
$ hg commit -m 'Added one file'
$ hg status
```

После запуска **hg commit** файлы, которые вы добавили перед фиксацией не будут отображаться в выводе **hg status**. Дело в том, что **hg status** по умолчанию сообщает только о «интересных» файлах — о тех, что вы модифицировали, либо указали Mercurial'у сделать что-либо с ними. Если ваш репозиторий содержит тысячи файлов, то вам редко понадобится информация обо всех не измененных файлах, которые отслеживает Mercurial. (Вы можете узнать и о них; мы вернемся к этому позже.)

Когда вы добавляете файл, Mercurial сразу ничего с ним не делает. Только после фиксации Mercurial сделает снимок состояния файла. Он будет продолжать отслеживать изменения каждый раз после фиксации, до тех пор пока файл не будет удален.

### 5.1.1. Явное и неявное именование файлов

При выполнении любой команды, если вы не указываете имя файла, Mercurial понимает это как «Я собираюсь работать со всеми файлами в этом каталоге и его подкаталогах».

```
$ mkdir b
$ echo b > b/somefile.txt
$ echo c > b/source.cpp
$ mkdir b/d
$ echo d > b/d/test.h
$ hg add b
adding b/d/test.h
adding b/somefile.txt
adding b/source.cpp
$ hg commit -m 'Added all files in subdirectory'
```

Обратите внимание, что в данном примере в отличие от предыдущего Mercurial вывел имена добавленных файлов.

Когда, как в предыдущем случае, мы явно указываем имя файла добавляемого в командной строке. Mercurial делает предположение, что вы знаете, что делаете, и не выводит ничего.

Однако, когда мы подразумеваем файлы, указывая только имя каталога, Mercurial дополнительно выводит имя каждого обработанного файла. Это делает процесс более прозрачным и уменьшает вероятность незаметных неприятных сюрпризов. Такое поведение свойственно большинству команд Mercurial.

### 5.1.2. Mercurial отслеживает файлы, а не каталоги

Mercurial не отслеживает информацию о каталогах. Вместо этого он отслеживает путь к файлу. Перед созданием файла он создает недостающие каталоги пути. После удаления — удаляет все пустые каталоги, которые присутствовали в пути файла. Кажется в этом нет ничего особенного, но имеется незначительное следствие: в Mercurial невозможно содержать пустой каталог.

Пустые каталоги нужны не часто, и есть несколько вариантов обойти ограничение, чтобы достичь желаемого эффекта. Разработчики Mercurial полагали, что усложнения, которые потребуются для поддержки управления пустыми каталогами, не стоят незначительных преимуществ данной опции.

Если вам требуется пустой каталог в репозитории, есть несколько путей сделать это. Первый — создать каталог и добавить в него **hg add** скрытый («hidden») файл. В Unix-подобных системах любой файл, имя которого начинается с точки («.»), рассматривается как скрытый большинством команд и инструментами GUI. Этот метод представлен ниже.

```
$ hg init hidden-example
$ cd hidden-example
$ mkdir empty
$ touch empty/.hidden
$ hg add empty/.hidden
$ hg commit -m 'Manage an empty-looking directory'
$ ls empty
$ cd ..
$ hg clone hidden-example tmp
updating to branch default
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ ls tmp
empty
$ ls tmp/empty
```

Другой вариант удовлетворить потребность в пустой директории — просто создать необходимый каталог автоматически с помощью скриптов.

## 5.2. Как прекратить отслеживание файла

Когда файл больше не нужен в репозитории, используйте команду **hg remove**, которая удаляет файл и указывает Mercurial прекратить его отслеживание. Удаленный файл в выводе **hg status** отображается буквой «R».

```
$ hg init remove-example
$ cd remove-example
$ echo a > a
$ mkdir b
$ echo b > b/b
$ hg add a b
adding b/b
$ hg commit -m 'Small example for file removal'
$ hg remove a
$ hg status
R a
$ hg remove b
removing b/b
```

После выполнения **hg remove** над файлом Mercurial больше не отслеживает его изменения даже если вы пересоздадите файл с таким же именем в этом каталоге. Если вы создали одноименный файл и хотите, чтобы Mercurial отслеживал новый файл, просто выполните **hg add** с ним. Mercurial будет знать что вновь добавленный файл никак не связан со старым одноименным файлом.

### 5.2.1. Удаление файла не влияет на его историю

Важно понимать, что удаление файла, имеет только два результата.

- Удаляется текущая версия файла из рабочего каталога.

- Mercurial прекращает отслеживать изменения над файлом со следующей фиксации (commit'a)

Удаление файла в любом случае не меняет историю его изменений.

Если вы обновите рабочий каталог до версии, в которой удаленный файл еще отслеживался, то он появится в каталоге и будет содержать данные, которые в нем были на момент фиксации версии. Если вы обновите каталог до более поздней версии, где данный файл уже был удален, Mercurial снова удалит файл из каталога.

## 5.2.2. Отсутствующие файлы

Mercurial считает потерянными файлы, которые вы удалили не используя **hg remove**. Отсутствующие файлы в выводе **hg status** отображаются с «!». Команды Mercurial как правило ничего не сделают с потерянными файлами.

```
$ hg init missing-example
$ cd missing-example
$ echo a > a
$ hg add a
$ hg commit -m 'File about to be missing'
$ rm a
$ hg status
! a
```

Если в вашем репозитории есть файл, который **hg status** отображает как потерянный, и вы хотите его действительно удалить, вы можете это сделать командой **hg remove `--after`**.

```
$ hg remove --after a
$ hg status
R a
```

С другой стороны если вы случайно удалили файл, используйте команду **hg revert** filename чтобы его восстановить. Файл будет восстановлен в неизмененной форме.

```
$ hg revert a
$ cat a
a
$ hg status
```

## 5.2.3. Замечание: почему в Mercurial явно указывается удаление файла?

Возможно вы удивитесь, что Mercurial требует явно указывать удаление файла. Раньше при разработке Mercurial можно было удалять файл, когда угодно. Mercurial замечал отсутствие файла автоматически при следующем запуске **hg commit** и прекращал отслеживание файла. На практике выяснилось, что это приводит к случайному незаметному удалению файлов.

## 5.2.4. Полезное сокращение — добавление и удаление файлов в один прием

Mercurial предоставляет комбинированную команду **hg addremove**, которая добавляет неотслеживаемые файлы и помечает отсутствующие файлы как удаленные.

```
$ hg init addremove-example
$ cd addremove-example
$ echo a > a
$ echo b > b
$ hg addremove
adding a
adding b
```

Команда **hg commit** также имеет опцию `-A`, которая выполняет то же самое добавление-и-удаление, за которыми сразу же следует фиксация.

```
$ echo c > c
```

```
$ hg commit -A -m 'Commit with addremove'
adding c
```

# 5.3. Копирование файлов

Для создания копии файла Mercurial предоставляет команду **hg copy**. Когда вы копируете файл с помощью этой команды, Mercurial создает запись о том, что новый файл является копией исходного файла. Он использует такие скопированные файлы, когда вы объединяете свою работу с чьей-либо еще.

## 5.3.1. Поведение копии при слиянии

При объединении получается, что изменения «преследуют» копии. Чтобы лучше продемонстрировать эту мысль, рассмотрим пример. Для начала возьмем обычный скромный репозиторий с одним файлом.

```
$ hg init my-copy
$ cd my-copy
$ echo line > file
$ hg add file
$ hg commit -m 'Added a file'
```

Нам необходимо работать параллельно, и затем объединить. Поэтому давайте клонируем наш репозиторий.

```
$ cd ..
$ hg clone my-copy your-copy
updating to branch default
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Возвращаясь к исходному репозиторию, выполним команду **hg copy**, чтобы сделать копию первого созданного файла.

```
$ cd my-copy
$ hg copy file new-file
```

Если посмотрим вывод команды **hg status**, увидим, что скопированный файл отображается как обычный добавленный файл.

```
$ hg status
A new-file
```

Но если мы укажем опцию `-C` в команде **hg status**, в выводе будет еще одна строка: файл, копия которого была сделана.

```
$ hg status -C
A new-file
  file
$ hg commit -m 'Copied file'
```

Теперь, вернувшись к клонированному репозиторию, сделаем параллельные изменения. Добавим строку в исходный файл.

```
$ cd ../your-copy
$ echo 'new contents' >> file
$ hg commit -m 'Changed file'
```

Теперь мы имеем измененный файл `file` в этом репозитории. Когда мы подтягиваем изменения из первого репозитория и объединяем две последние ревизии (head), Mercurial переносит изменения, которые были сделаны в файле `file` в его копию `new-file`.

```
$ hg pull ../my-copy
pulling from ../my-copy
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
```

```
(run 'hg heads' to see heads, 'hg merge' to merge)
$ hg merge
merging file and new-file to new-file
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ cat new-file
line
new contents
```

## 5.3.2. Почему изменения следуют за копией?

Перенос изменений исходного файла в копии может показаться понятным лишь посвященным, но в большинстве случаев это крайне желательно.

Прежде всего напомним, что подобный перенос происходит только при объединении. И если вы делаете **hg copy** файла и последовательно изменяете оригинал в процессе работы, с копией ничего не происходит.

Во-вторых, надо понимать, что изменения будут вноситься в копию до тех пор пока репозиторий, из которого подтягиваются изменения, не знает о существовании копии.

Mercurial делает это по следующей причине. Представим, что я исправляю серьезный баг в исходнике, и фиксирую изменения. А в то же время вы решаете сделать **hg copy** файла в вашем репозитории, при этом вы ничего не знаете о баге и не видите последней фиксации, и вы начинаете колдовать со своей копией файла.

Когда вы подтянули и объединили мои изменения, и Mercurial не перенес изменения в копии, ваш исходник будет содержать баг, и пока вы не вспомните и не перенесете исправление вручную, баг останется не исправленным в вашей копии файла.

Благодаря автоматическому переносу зафиксированных изменений от исходного файла к копиям, Mercurial предотвращает подобные проблемы. Насколько мне известно, Mercurial единственная система контроля версий, которая подобным образом переносит изменения в копии.

Когда в истории изменений есть запись о появлении копии и последующего объединения, обычно не требуется последующий перенос изменений в исходном файле в копию, и поэтому Mercurial переносит только те изменения в копию, которые были до этой точки, а не дальнейшие.

## 5.3.3. Как сделать, чтобы изменения не преследовали копию

Если по каким-то причинам вы решили, что дело автоматического переноса изменений в копии не для Вас, то просто используйте обычную системную команду копирования (в Unix-подобных системах, это **cp**), а затем добавьте файл вручную с помощью **hg add**. Перед тем как это сделать, перечитайте Раздел 5.3.2, «Почему изменения следуют за копией?», и убедитесь, что в вашем специфическом случае это действительно не нужно.

## 5.3.4. Поведение команды hg copy

При выполнении команды **hg copy** Mercurial делает копии каждого исходного файла в таком виде, в каком на текущий момент он находится в рабочем каталоге. Это означает, что если вы вносите изменения в файл, а затем делаете его копию без предварительной фиксации изменений, то копия будет содержать изменения, внесенные до момента копирования. (Мне кажется такое поведение нелогичным, поэтому я обращаю внимание здесь.)

Действие команды **hg copy** подобно команде **cp** в Unix (для удобства вы можете использовать алиас **hg cp**). Последний аргумент — адрес назначения, все предыдущие — источники.

Если вы указываете единственный файл как источник и файл назначения не существует, будет создан новый файл с этим именем.

```
$ mkdir k
$ hg copy a k
$ ls k
```

```
a
```

Если адрес назначения указан каталог, Mercurial сделает копии источников в этот каталог

```
$ mkdir d
$ hg copy a b d
$ ls d
a  b
```

Копирование каталогов рекурсивно, и сохраняет структуру каталогов источника.

```
$ hg copy z e
copying z/a/c to e/a/c
```

Если и источник, и назначение — каталоги, то дерево каталогов источника будет также создано в каталоге назначения.

```
$ hg copy z d
copying z/a/c to d/z/a/c
```

Также как с командой **hg remove**, если вы создали копии вручную и хотите, чтобы Mercurial знал, что это копии, просто используйте опцию `--after` в команде **hg copy**.

```
$ cp a n
$ hg copy --after a n
```

# 5.4. Переименование файлов

Переименование файлов обычно используется чаще, чем копирование. Я рассмотрел команду **hg copy** до команды переименования, так как по существу Mercurial воспринимает переименование так же, как копирование. Следовательно, понимание того, как ведет себя Mercurial с копиями, объяснит, чего ожидать от переименования файлов.

При выполнении команды **hg rename** Mercurial копирует исходные файлы, затем удаляет их и помечает как удаленные.

```
$ hg rename a b
```

Команда **hg status** показывает, что новые файлы были добавлены, а файлы, с которых они были скопированы, удалены.

```
$ hg status
A b
R a
```

Так же как с результатами **hg copy** мы должны использовать опцию `-C` команды **hg status**, чтобы увидеть, что добавленный файл действительно отслеживается Mercurial, как исходный, уже удаленный, файл.

```
$ hg status -C
A b
  a
R a
```

Так же как и для команд **hg remove** и **hg copy**, вы можете указать Mercurial о переименовании после выполнения, используя опцию `--after`. В большинстве других случаев, поведение команды **hg rename** и ее поддерживаемых опций подобно поведению команды **hg copy**.

Если вы знакомы с командной строкой Unix, вы будете рады узнать, что команда **hg rename** может работать как **hg mv**.

## 5.4.1. Переименование файлов и объединение изменений

С тех пор как переименование в Mercurial реализовано как копирование и удаление, после переименования происходит такой же перенос изменений при объединении (merge), как и после копирования.

Если вы изменили файл и переименовали его, а затем мы объединяем наши изменения, мои модификации в файле с первоначальным именем будут перенесены в файл с новым именем. (Это кажется достаточно простым действием, однако не во всех системах контроля версий оно работает.)

Если по поводу следования изменений за копией вы возможно кивнете и скажете: «Да, это может быть полезно», то, что касается следования их при переименовании, абсолютно ясно, что это действительно важно. Без этой возможности изменения легко бы оставались осиротевшими при переименовании файлов.

## 5.4.2. Расходящиеся переименования и слияние

Расходящиеся переименования происходят, когда два разработчика берутся за один файл — назовем его `foo` в своих репозиториях.

```
$ hg clone orig anne
updating to branch default
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ hg clone orig bob
updating to branch default
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Анна переименовывает файл в `bar`.

```
$ cd anne
$ hg rename foo bar
$ hg ci -m 'Rename foo to bar'
```

Тем временем Bob переименовывает его в `quux`. (Помните, что **hg mv** является псевдонимом для **hg rename**.)

```
$ cd ../bob
$ hg mv foo quux
$ hg ci -m 'Rename foo to quux'
```

Я расцениваю это как конфликт, потому что каждый разработчик выразил свое мнение о том, как данный файл должен называться.

Как вы считаете, что должно произойти когда они объединят работу? На самом деле Mercurial всегда сохраняет оба имени при слиянии изменений, которые содержат расходящиеся переименования.

```
# See http://www.selenic.com/mercurial/bts/issue455
$ cd ../orig
$ hg pull -u ../anne
pulling from ../anne
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
1 files updated, 0 files merged, 1 files removed, 0 files unresolved
$ hg pull ../bob
pulling from ../bob
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
$ hg merge
note: possible conflict - foo was renamed multiple times to:
 bar
 quux
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ ls
bar  quux
```

Обратите внимание, что Mercurial предупредил о наличии расходящихся переименований, но оставил на ваше усмотрение, как с ними поступить после слияния.

### 5.4.3. Сходящиеся переименования и слияние

Другой вариант конфликта возникает, когда два человека переименовывают два разных файла и дают им одно и то же имя. В таком случае Mercurial выполняет стандартное слияние и предоставляет вам управлять им для нахождения подходящего решения.

### 5.4.4. Другие проблемы с именованием

У Mercurial есть давнишний баг, который дает ошибку при выполнении слияния, если с одной стороны имеется некоторый файл, а с другой стороны — каталог с таким же именем. Баг задокументирован как Mercurial issue 29 [http://www.selenic.com/mercurial/bts/issue29].

```
$ hg init issue29
$ cd issue29
$ echo a > a
$ hg ci -Ama
adding a
$ echo b > b
$ hg ci -Amb
adding b
$ hg up 0
0 files updated, 0 files merged, 1 files removed, 0 files unresolved
$ mkdir b
$ echo b > b/b
$ hg ci -Amc
adding b/b
created new head
$ hg merge
abort: Is a directory: /tmp/issue29iLXJ9A/issue29/b
```

## 5.5. Избавление от ошибок

У Mercurial есть несколько полезных команд для восстановления после некоторых распространенных ошибок.

Команда **hg revert** позволяет отменить изменения, сделанные в рабочем каталоге. Например, если вы случайно выполнили **hg add**, просто запустите **hg revert**, указав имя добавленного файла, и файл больше не будет считаться добавленным для отслеживания в Mercurial. **hg revert** можно использовать и для отмены от ошибочных изменений в файле.

Необходимо помнить, что команда **hg revert** действует только на изменения, которые не были фиксированы. Если вы зафиксировали изменение, но поняли, что произошла ошибка, вы по-прежнему можете ее исправить, хотя возможности будут более ограничены.

Дополнительная информация о команде **hg revert** и о том, что можно сделать с зафиксированными изменениями, приведена в Глава 9, *Поиск и исправление ваших ошибок*.

## 5.6. Работа со сложными слияниями

В сложных и крупных проектах, не редкость слияние двух ревизий является головной болью. Предположим, что существует большой исходный файл, который был сильно отредактирован каждой стороной слияния: это практически неминуемо приведет к конфликтам, некоторые из которых могут потребовать несколько попыток разобраться.

Давайте начнём с простого случая, чтоб увидеть как с этим бороться. Начнем с репозитория, содержащего один файл, и клонируем его дважды.

```
$ hg init conflict
$ cd conflict
$ echo first > myfile.txt
$ hg ci -A -m first
adding myfile.txt
$ cd ..
```

```
$ hg clone conflict left
updating to branch default
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ hg clone conflict right
updating to branch default
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

В одном клоне, будем изменять файл по одному пути.

```
$ cd left
$ echo left >> myfile.txt
$ hg ci -m left
```

В другом клоне, будем изменять файл по одному пути.

```
$ cd ../right
$ echo right >> myfile.txt
$ hg ci -m right
```

Далее вытянем каждую ревизию в наш первоначальный репозиторий.

```
$ cd ../conflict
$ hg pull -u ../left
pulling from ../left
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ hg pull -u ../right
pulling from ../right
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
not updating: crosses branches (merge branches or update --check to force update)
```

Мы думаем, что теперь в нашем репозитории находится две головных ревизии.

```
$ hg heads
changeset:   2:ee01e5caae49
tag:         tip
parent:      0:07ad45cd9c8f
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:09:37 2012 +0000
summary:     right

changeset:   1:0a7bacc4a7aa
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:09:37 2012 +0000
summary:     left
```

Обычно, если мы запустим **hg merge** в этот момент, он перенаправит нас в графический интерфейс, который позволит нам решать вручную противоречия в изменениях в файле `myfile.txt`. Однако, чтобы упростить эту задачу для представления здесь, мы хотели бы чтоб слияние провалилось. Вот один из способов, которым мы можем этого добиться.

```
$ export HGMERGE=false
```

Мы сообщили механизму слияния Mercurial выполнить команду **false** (которая, как мы и хотели, потерпит неудачу) если обнаружит, что слияние не может разрешится автоматически.

Если мы сейчас запустим **hg merge**, он должен остановится, и сообщить о провале.

```
$ hg merge
merging myfile.txt
merging myfile.txt failed!
```

```
0 files updated, 0 files merged, 0 files removed, 1 files unresolved
use 'hg resolve' to retry unresolved file merges or 'hg update -C .' to abandon
```

Даже если мы не заметим, что объединение не удалось, Mercurial не даст нам случайно зафиксировать результат неудачного слияния.

```
$ hg commit -m 'Attempt to commit a failed merge'
abort: unresolved merge conflicts (see hg help resolve)
```

Когда **hg commit** выполняется неудачно, как в этом случае, можно предположить, что мы используем неправильную команду **hg resolve**. Как обычно, **hg help resolve** выведет полезную справку.

### 5.6.1. Файл анализа состояний

Когда происходит слияние, большая часть файлов, как правило, остаётся без изменений. Для каждого файла, в котором Mercurial что-то делает, он отслеживает состояние файла.

- Разрешенные файлы, которые был успешно объединены, либо автоматически с помощью Mercurial или вручную с помощью человеческого вмешательства.
- Неразрешенные файлы — не были объединены успешно, необходимо уделить дополнительное внимание.

Если Mercurial видит любой файл в неразрешенных состояние после слияния, он будет считать, что слияние не увенчались успехом. К счастью, нам не запускать всё слияние с нуля.

Опция `--list` или `-l` команды **hg resolve** печатает состояние каждого объединяемого файла.

```
$ hg resolve -l
U myfile.txt
```

В выводе **hg resolve**, разрешённые файлы отмечены `R`, а неразрешённые файлы отмечены `U`. Если какие-либо файлы перечислены с `U`, мы знаем, что попытка зафиксировать результаты слияния не удастся.

### 5.6.2. Разрешение файлов при слиянии

Есть несколько вариантов, чтобы переместить файл из неразрешенного в разрешенное состояние. Самым распространенным является перезапуск **hg resolve**. Если мы будем пропускать отдельные имена файлов или каталогов, он будет повторять слияние неразрешенных файлов в этих местах. Мы также можем использовать опцию `--all` или `-a`, которая заставит повторить слияние для всех нерешенных файлов.

Mercurial также позволяет нам изменять состояние файла напрямую. Мы можем вручную пометить файл как решённый с помощью опции `--mark`, или, как нерешенный используя опцию `--unmark`. Это позволяет очистить особенно грязные сторонние дополнения, и следить за каждым файлом в прогрессе выполнения.

## 5.7. Более удобные diff-ы

Вывода команды **hg diff** по-умолчанию обратно совместимым с обычной командой **diff**, но имеет некоторые недостатки.

Рассмотрим случай, когда мы используем **hg rename** для переименования файла.

```
$ hg rename a b
$ hg diff
diff -r e1f8689c7be4 a
--- a/a Thu Feb 02 14:09:36 2012 +0000
+++ /dev/null Thu Jan 01 00:00:00 1970 +0000
@@ -1,1 +0,0 @@
-a
diff -r e1f8689c7be4 b
--- /dev/null Thu Jan 01 00:00:00 1970 +0000
+++ b/b Thu Feb 02 14:09:36 2012 +0000
@@ -0,0 +1,1 @@
+a
```

Вывод **hg diff** выше скрывает тот факт, что мы просто переименовали файл. Команда **hg diff** принимает опции `--git` или `-g`, чтобы использовать новый формат diff, который отображает такую информацию в более читаемом виде.

```
$ hg diff -g
diff --git a/a b/b
rename from a
rename to b
```

Эта функция также помогает в случаях, которые в противном случае могут ввести в заблуждение: файл, который, как представляется, измененился в соответствии с **hg status**, но для которого **hg diff** ничего не выводит. Такая ситуация может возникнуть, если мы изменили исполняемый файл.

```
$ chmod +x a
$ hg st
M a
$ hg diff
```

Обычная команда **diff** не обращает внимания на права доступа к файлу, поэтому **hg diff** ничего по умолчанию не выводит. Если мы запускаем ее с опцией `-g`, она расскажет нам, что происходит.

```
$ hg diff -g
diff --git a/a b/a
old mode 100644
new mode 100755
```

# 5.8. Какими файлами управлять, а каких избегать

Системы управления версиями, как правило, лучше всего в управляют текстовыми файлами, которые написаны людьми, например исходный код, где файлы не меняются от одного изменения к следующему. Некоторые централизованные систем контроля версий могут сносно иметь дело и с бинарными файлами, такими как растровые изображения.

Например, команда разработчиков игр, как правило, совмещать ее исходный код и все её двоичный данные (например, данные геометрии, текстуры, карты уровней) в системе контроля версий.

Так как, как правило, невозможно объединить два противоречивых изменения в двоичном файле, централизованные систем часто предоставляют механизм блокирования файла, что позволяет пользователю сказать: «Я единственный, кто может редактировать этот файл».

По сравнению с централизованной системой, распределенная система контроля версий меняет некоторые из факторов, которыми руководствуются принимая решение, какими файлами управлять и каким образом.

Например, распределенная система контроля версий не может, по своей природе, блокировать файлы. Таким образом, нет встроенного механизма защищающего 2 людей от конфликтующих изменений в двоичном файле. Если у вас есть команда, где несколько человек могут часто редактировать бинарный файл, тогда идея использовать Mercurial или любую другую распределенную систему контроля версий для управления этими файлами не будет хорошей.

При сохранении изменений файлов, как правило Mercurial сохраняет только различия между предыдущей и нынешней версиями файла. Для большинства текстовых файлов, это очень эффективно. Тем не менее, некоторые файлы (в частности, бинарные файлы) записываются таким образом, что даже незначительные изменения в логической структуре файла отражается в изменении во многих или большинстве байтов внутри файла. Например, сжатые файлы, особенно чувствительны к этому. Если различия между каждой последующей версией файла, всегда имеют больший размер, Mercurial не сможет хранить историю изменений файла очень эффективно. Это может повлиять на локальные ресурсы хранения и количество времени, необходимое для копирования хранилища.

Чтобы получить представление о том, каким образом это может повлиять на вас на практике, предположим, вы хотите использовать Mercurial для управления документами OpenOffice. OpenOffice хранит документы на диске как сжатый ZIP архив. Изменение даже одной буквы документа в OpenOffice, и приводит к изменению почти

каждого байта во всём файле, когда вы его сохраните. Теперь предположим, что файл 2 МБ в размере. Поскольку большая часть файла меняется каждый раз, когда вы сохраняете его, Mercurial придется хранить все 2 МБ файла каждый раз, когда вы фиксируете изменения, даже если с вашей точки зрения, меняется только несколько слов за раз. Частое редактирование файла, не дружественно для системы хранения Mercurial и приведёт к эффекту переполнения хранилища.

Еще хуже, если и вы, и кто-то изменяете документ OpenOffice, и не существует удобный способа объединить вашу работу. В самом деле, нет даже хорошего способа сказать, что изменилось между соответствующими ревизиями.

Есть, несколько четких рекомендаций относительно конкретных видов файлов, с которыми нужно быть очень осторожными.

- Файлы, которые являются очень большими и несжимаемыми, например, iso cd-rom образы, будет в силу огромных размеров очень медленно клонироваться по сети.

- Файлы, которые сильно меняются от одного изменения к следующему могут быть дорогостоящим для хранения, если вы их изменяете часто, и конфликты из-за одновременного изменения могут быть сопряжено с трудностями.

# 5.9. Резервные копии и мониторинг.

Mercurial поддерживает полную копию истории в каждом клоне, всем, кто использует Mercurial для совместной работы над проектом потенциально могут выступать в качестве источника резервных копий в случае катастрофы. Если центральное хранилище становится недоступным, можно построить просто заменить путем клонирования копии хранилища от другого разработчика и, получить любые изменения, которые, возможно, не видели другие.

Достаточно просто использовать Mercurial для оффлайновых резервных копий и удаленных зеркал. Настройка периодического задания (например, через команду **cron**) на удаленном сервере, чтобы вытаскивать изменения из вашего основного хранилища каждый час. Это будет немого сложнее в маловероятном случае, когда вы поддерживаете часто изменяемый набор мастер-репозиториев, в этом случае вам нужно сделать небольшой сценарий, чтобы обновлять список репозиториев для резервного копирования.

Если вы выполняете традиционное резервное копирование мастер-репозитория на ленту или жесткий диск, и вы хотите создать резервную копию хранилища с именем `myrepo`, **hg clone -U myrepo myrepo.bak** для создания клона `myrepo` перед началом резервного копирования. Опция `-U` не проверяет рабочий каталог после завершения клонирования, поскольку излишне и резервное копирование занимало бы больше времени.

Если вы затем используете `myrepo.bak` вместо `myrepo`, вам будет гарантирована возможность получить снимок репозитория, не боясь что какой-то разработчик вставит изменение в середине резервного копирования.

# Глава 6. Взаимодействие с людьми

Mercurial, как полностью децентрализованный инструмент, не навязывает никакой политики взаимодействия людей друг с другом. Однако, если вы новичок в работе с распределенным контролем версий, будет полезно иметь некоторые инструменты и примеры в голове, обдумывая возможные модели рабочего процесса.

## 6.1. Веб-интерфейс Mercurial

Mercurial имеет мощный веб-интерфейс, обеспечивающий несколько полезных возможностей.

В плане интерактивного использования интерфейс позволяет просматривать один или несколько репозиториев. Вы можете просматривать историю репозитория, изменения (комментарии и различия), а также содержимое каждого каталога и файла. Можно даже посмотреть на историю в графическом виде, который позволяет проследить зависимости между отдельными ревизиями и слияниями.

Также для использования человеком web-интерфейс обеспечивает RSS-канал для изменений в репозитарии. Это позволяет вам «подписаться» на репозитарий, используя вашу любимую программу для чтения новостей, и автоматически получать сообщения об активности в данном репозитарии, как только что-то произойдет. Я считаю эту возможность гораздо более удобной, чем модель подписки на почтовый список рассылки, с помощью которого будут рассылаться сообщения, поскольку это не требует дополнительной настройки со стороны владельца репозитария.

Web-интерфейс также позволяет удаленным пользователям клонировать репозиторий, получать с него изменения и в случае, если сервер настроен для внесения изменений, возвращать изменения обратно. Тунельный HTTP протокол Mercurial хорошо сжимает данные и это позволяет работать даже на низкоскоростных сетевых соединениях.

Простейший способ начать использовать Web-интерфейс — использовать ваш web-браузер для посещения существующего репозитария, например, такого, как основной репозитарий Mercurial, расположенный по адресу http://www.selenic.com/repo/hg.

Если вы заинтересованы обеспечить Web-интерфейс к своему собственному репозитарию, есть несколько хороших путей.

Самый простой и быстрый способ, чтобы начать работу в неформальной обстановке является использование команды **hg serve**, которая лучше всего подходит для кратковременных «лёгких» серверов. Смотрите Раздел 6.4, «Неофициальный обмен с помощью **hg serve**» чтобы узнать, как использовать эту команду.

Для долгоживущих репозитариев, которые вы хотели бы иметь постоянно доступным, есть несколько общественных хостинг сервисов. Некоторые из них бесплатны для проектов с открытым кодом, а другие платные коммерческие хостинги. Актуальный список доступен в http://www.selenic.com/mercurial/wiki/index.cgi/MercurialHosting.

Если вы предпочитаете использовать ваш собственный компьютер для репозитариев, Mercurial имеет встроенную поддержку нескольких популярных хостинг технологий, в первую очередь cgi (common gateway interface), а также wsgi (web services gateway interface). Смотрите Раздел 6.6, «Работа по HTTP с использованием CGI» для более подробной информации о конфигурации CGI и WCGI.

## 6.2. Модели сотрудничества

Если есть достаточно гибкий инструмент, принятие решений по поводу рабочего процесса становится задачей скорее социальной инженерии, чем технической. Mercurial накладывает немного ограничений на то, как Вы можете структурировать работу над проектом, таким образом, Вам и Вашей группе возможно создать модель, удовлетворяющую Вашим особым нуждам, и жить по этой модели.

## 6.2.1. Факторы, которые необходимо иметь в виду

Наиболее важным аспектом любой модели, который Вы должны иметь в виду, является то, как эта модель соответствует потребностям и возможностям людей, которые будут ее использовать. Это может показаться самоочевидным, но даже если это так, вы все равно не можете себе позволить забыть об этом хотябы на время.

Я как-то создал модель рабочего процесса, которая казалась, была идеальной для меня, но которая вызвала значительные потрясения и беспорядки в моей команде разработчиков. Несмотря на мои попытки объяснить, почему нам необходим набор различных ветвей репозитария, и каким образом изменения должны проходить между ними, несколько членов команды восстали. Несмотря на то, что они были умные люди, они не хотели обращать внимание на ограничения, в соответствии с которыми мы действовали, или столкнуться с последствиями таких ограничений в деталях той модели, которую я защищал.

Не заметайте поддающиеся предвидению социальные или технические проблемы под ковер. Какую бы схему Вы ни внедряли, вы должны планировать ошибки и проблемные сценарии. Рассмотрите добавление автоматизированных механизмов для предотвращения или быстрого восстановления после проблем, которые Вы можете предвидеть. Например, если Вы хотите иметь ветвь, в которой хранятся изменения не-для-релиза, Вы должны заранее хорошо подумать над тем, что кто-то может случайно добавить эти изменения в релиз. Вы могли бы избежать этой конкретной проблемы, написав перехватчик, который будет мешать изменениям, вносимым в неподходящую ветвь кода.

## 6.2.2. Неформальный подход

Я не стал бы рекомендовать этот подход, как универсальный, однако он крайне прост и отлично работает в некоторых нестандартных ситуациях.

К примеру, множество проектов представляет собой группу слабо взаимодействующих между собой участников, которые крайне редко встречаются лично. Некоторые группы стараются преодолеть возникающую в результате удаленной работы изоляцию, устраивая «спринты». Во время «спринта», участники собираются вместе в назначенном месте — конференц-зале компании или отеля — и проводят несколько дней в неотрывной разработке, разбираясь со сложными местами проектов.

«Спринт» или хакерский сбор в кафе — отличное место для применения команды **hg serve**, поскольку эта команда не требует никакой сложной серверной инфраструктуры. Вы можете приступить к использованию **hg serve** моментально, прочитав Раздел 6.4, «Неофициальный обмен с помощью **hg serve**». Вы можете просто сообщить соседу, что Вы запустили сервер, передать ему ссылку любым удобным способом, и у вас уже есть отличное средство для совместной работы. Ваш сосед может открыть полученный URL своим браузером и ознакомиться с внесенными Вами изменениями, он может воспользоваться сделанными Вами исправлениями, а может клонировать ветвь, содержащую новые возможности, и опробовать ее.

Одновременно положительной и отрицательной стороной такого варианта взаимодействия является то, что только те люди, которые знают о внесенных вами изменениях, могут их увидеть. Неформальный подход просто невозможно использовать в больших коллективах, поскольку каждый участник должен отслеживать изменения в **n** репозиториях, чтобы получить их.

## 6.2.3. Единый центральный репозиторий

Для маленьких проектов, мигрирующих с централизованных систем контроля версий, возможно самым легким путем будет использование одного центрального репозитория. Это наиболее частый «кирпич» для создания более сложных структур.

Каждый участник разработки начинает работу с создания локальной копии центрального репозитория. Он может получать изменения из него тогда, когда ему понадобится. В то же время некоторые (а возможно и все) разработчики имеют привилегии на добавление в репозиторий готовых к публикации изменений.

В рамках этого подхода также остается возможным обмен изменениями напрямую между разработчиками, без добавления их в центральный репозиторий. К примеру, я исправил ошибку, однако я не могу гарантировать, что будучи опубликованным в центральном репозитории, мое исправление не нарушит работу кода других

разработчиков, которые получат это исправление. Чтобы снизить риск возможного вреда, я могу попросить вас клонировать мой репозиторий в ваш собственный временный репозиторий, и проверить работоспособность. Это позволит нам избежать публикации потенциально небезопасных изменений до тех пор, пока они не пройдут небольшого тестирования.

Если команда хостит собственные репозитории по такому типу работы, разработчики обычно используют протокол **ssh** для безопасного добавления изменений в центральный репозиторий, как это описано в разделе Раздел 6.5, «Использование протокола Secure Shell (ssh)». Также, часто используется возможность публикации доступной только для чтения копии репозитория с помощью HTTP-сервера, используя CGI, как показано в Раздел 6.6, «Работа по HTTP с использованием CGI». Публикация с помощью HTTP удовлетворяет потребностям людей, которые не имеют доступа на запись, и которые хотят использовать web-браузеры для просмотра истории репозитория.

## 6.2.4. Хостинг центрального репозитория

Преимуществом общественных услуг хостинга, таких как Bitbucket [http://bitbucket.org/] является то, что они не только поддерживают неудобные детали конфигурации, такие как учетные записи пользователей, проверки подлинности и защищенные протоколы передачи, они обеспечивают дополнительную инфраструктуру, чтобы эта модель хорошо работала.

Например, хорошо организованный хостинг позволяет людям клонировать собственные копии репозитория за один клик. Это позволяет людям работать в разных местах и делиться своим изменениями, когда они готовы.

Кроме того, хороший сервис хостинга позволяет людям общаться друг с другом, например, говорить «Есть изменения готовые для просмотра в этом дереве».

## 6.2.5. Работа с несколькими ветвями

Работа над проектами более-менее значительного размера, как правило, идет сразу на нескольких фронтах. В течение жизненного цикла, проект переживает периодические официальные релизы. После этого релиз может на некоторое время после выпуска перейти в «режим поддержки» — когда в программное обеспечение вносятся только исправления ошибок, не добавляя новых возможностей. Параллельно с этими релизами, один или несколько будущих релизов находятся в разработке. Для обозначения подобных направлений в разработке, используется термин «ветвь».

Mercurial исключительно хорошо подходит для ведения нескольких похожих, но не одинаковых ветвей. Каждое «направление разработки» может храниться в своем собственном центральном репозитории, и вы можете добавлять изменения из одного в другой, когда появляется такая необходимость. Поскольку репозитории являются независимыми, нестабильные изменения в разрабатываемой ветви не повлияют на стабильную ветвь, покуда кто-нибудь не захочет объединить их.

Вот как это работает на практике: Допустим, у вас есть одна «главная ветвь» на центральном сервере.

```
$ hg init main
$ cd main
$ echo 'This is a boring feature.' > myfile
$ hg commit -A -m 'We have reached an important milestone!'
adding myfile
```

Остальные участники клонируют его, делают изменения, проверяют их, и добавляют в репозиторий.

Когда главная ветвь достигает состояния релиза, вы можете использовать команду **hg tag**, чтобы дать постоянное имя этой ревизии.

```
$ hg tag v1.0
$ hg tip
changeset:   1:1e1d75b92915
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:09:34 2012 +0000
summary:     Added tag v1.0 for changeset 6fc05bef0b12

$ hg tags
```

```
tip                                1:1e1d75b92915
v1.0                               0:6fc05bef0b12
```

Теперь, скажем, произошли изменения в главной ветви.

```
$ cd ../main
$ echo 'This is exciting and new!' >> myfile
$ hg commit -m 'Add a new feature'
$ cat myfile
This is a boring feature.
This is exciting and new!
```

Используя тег для пометки релиза, участник, клонирующий репозиторий, в любое время в последующем может воспользоваться командой **hg update** для получения точной копии рабочей папки по состоянию на момент релиза.

```
$ cd ..
$ hg clone -U main main-old
$ cd main-old
$ hg update v1.0
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cat myfile
This is a boring feature.
```

В дополнение к этому, сразу же после того, как основная ветка будет тегирована, кто-либо может клонировать основную ветку на сервере в новую «стабильную» ветку, также находящуюся на сервере.

```
$ cd ..
$ hg clone -rv1.0 main stable
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
updating to branch default
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Любой, кому нужно сделать изменения в стабильной ветке, может клонировать этот репозиторий, вполнить изменения, сделать коммит и передать изменения сюда.

```
$ hg clone stable stable-fix
updating to branch default
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd stable-fix
$ echo 'This is a fix to a boring feature.' > myfile
$ hg commit -m 'Fix a bug'
$ hg push
pushing to /tmp/branching5YEWM8/stable
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
```

Поскольку репозитории Mercurial независимы, и поскольку Mercurial не осуществляет автоматически изменения, стабильная и основная ветки изолированы друг от друга. Изменения, сделанные вами в основной ветке не «просачиваются» в стабильную ветку, и обратно.

Часто у вас будет возникать желание, что бы багфиксы из стабильной ветки применялись и к основной. Вместо простого переписывания исправления в основную ветку, вы можете просто выполнить pull и merge измениний стабильной ветки в основную. И Mercurial перенесет вам эти багфиксы.

```
$ cd ../main
$ hg pull ../stable
pulling from ../stable
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
$ hg merge
```

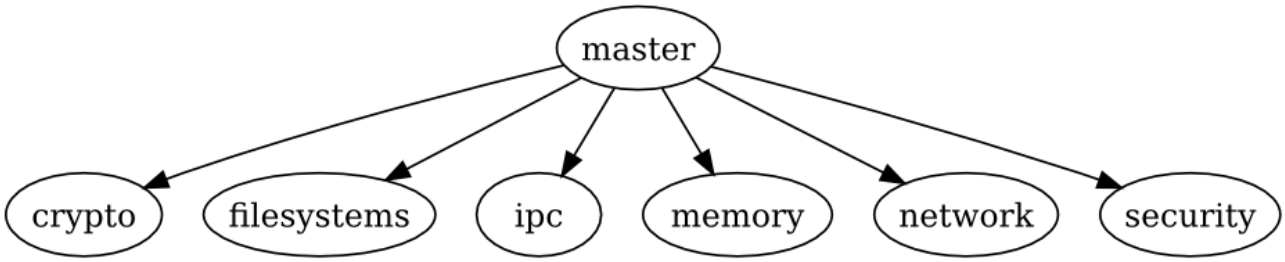
```
merging myfile
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ hg commit -m 'Bring in bugfix from stable branch'
$ cat myfile
This is a fix to a boring feature.
This is exciting and new!
```

Основная ветка все так же содержит изменения, которых нет в стабильной ветке, но она еще и содержит все исправления из стабильной ветки. А стабильная ветка продолжает оставаться незатронутой этими изменениями, так как изменения перемещаются только из стабильной ветви в остальные, а не иначе.

## 6.2.6. Ветви для новых функций

Для больших проектов эффективным способом управлять изменениями будет разбиение команды на несколько меньших групп. Каждая из которых будет использовать свою собственную ветку клонированную из единой «главной» ветки, используемой для всего проекта. Люди работающие над отдельными ветками обычно хорошо изолированны от изменений в других ветках.

Рисунок **6.1.** Ветви для новых функций




Когда отдельная функция приобретает удобоваримую форму, кто-то из команды, работавшей над данной функцией затягивает изменения из главной ветки в ветку функции и выполняет слияние, а затем заливает изменения назад в главную ветку.

## 6.2.7. Релиз по расписанию

Некоторые проекты организованы по принципу «поезда»: выпуск новой версии планируется каждые несколько месяцев, и все функции, которые завершены к «отправлению поезда», включаются в релиз.

Эта модель имеет много общего с моделью «ветви для новых функций». Отличие следующее: в случае, если функция не вошла в релиз (опоздала на поезд), один из членов команды-разработчика функции вытягивает вошедшие в релиз изменения в ветку функции и производит слияние, после чего команда продолжает работать «поверх» нового релиза и, таким образом, сможет включить функцию в новый релиз.

## 6.2.8. Модель ядра Linux

В разработке ядра Linux есть неглубокая иерархическая структура, окруженная очевидным облаком хаоса. Поскольку большинство разработчиков Linux используют **git**, распределенный инструмент управления версиями, подобный Mercurial, полезно описать рабочий процесс в этом окружении. Если вам понравятся идеи, подход легко переносится на другие инструменты.

В центре сообщества сидит Линус Торвальдс (Linus Torvalds), создатель Linux. Он публикует единственный исходный архив, который считается «авторитетным» текущим деревом всего сообщества разработчиков. Любой может клонировать дерево Линуса, но он очень разборчив в выборе деревьев с которых можно вливать изменения.

У Линуса есть несколько «доверенных лейтенантов». Как правило он помещает любые изменения, которые они издают, в большинстве случае даже не рассматривая их изменения. Некоторые из лейтенантов вообще являются «мейнтейнерами», отвечающими за отдельные подсистемы в пределах ядра. Если случайный разработчик ядра

хочет сделать изменения в подсистему, которые они хотят внести в дерево Линуса, они должны узнать кто является мейнтейнером подсистемы и попросить его внести изменения. Если мейнтейнер рассмотрит их изменения и согласиться их взять, то он передаст их Линусу должны образом.

У индивидуальных лейтенантов имеются свои собственные подходы к рассмотрению, принятию и публикации изменений, и для того чтобы решить когда передать их Линусу. В дополнение, существует несколько хорошо известных веток которые люди используют для различных целей. Например, некоторые люди обслуживают «stable» («стабильную ветку») репозитория с несколько устаревшей версией ядра, внося критические исправления если они необходимы. Некоторые мейнтейнеры публикуют несколько деревьев: одно для экспериментальных изменений, одно для изменений которые они собираются внести в стабильные и так далее. Другие просто публикуют отдельные деревья.

У этой модели есть две важные особенности. Первая это «только внесение». Вы должны спросить, убедить или попросить другого разработчика сделать изменения за вас, потому что практически нет веток, куда может помещать код одна персона, и не существует способа помещать код в ветки, контролируемые кем-то еще.

Вторая базируется на репутации и доверии. Если вы никому не известны, Линус вероятно проигнорирует ваши изменения и даже оставит вас без ответа. Но майнтейнер подсистемы вероятно рассмотрит их, если они пройдут его критерии пригодности. Чем более «хорошие» изменения вы вносите, тем более вероятно что они будут доверять вашему суждению и принимать ваши изменения. Если вы хорошо известны и поддерживаете долгое время какую-либо возможность в ветке, которые Линус еще не принял, люди с похожими интересами могут вносить ваши изменения регулярно, для того чтобы не отставать от вашей работы.

Репутация и признание не распространяется на другие подсистемы или людей. Если вы будете уважаемым, но специализированным разработчиком по файловым хранилищам, и попытаетесь исправить баг в сетевой подсистеме, это скорее всего, будет рассматриваться майнтейнером этой подсистемы как исправление от незнакомца.

Людям, пришедшим с более строгих проектов, и сравнивающих процесс разработки ядра Linux с места откуда они пришли, процесс разработки кажется полностью безумным. Разработка подчиняется прихотям людей; люди делают большие изменения всякий раз, как считают это нужным; и темп эволюционирования потрясает. Но все же Linux — это успешная и хорошо оцениваемая часть программного обеспечения.

## 6.2.9. Втягивающее против совместно-вносимого сотрудничества

Постоянным источником жара в open source сообществе является вопрос, которая из двух моделей разработки лучше: та, в которой люди всегда берут изменения у других, или та, в которой множество людей вносят изменения в общий репозиторий.

Обычно, покровители модели совместного вноса используют инструменты принуждающие использовать этот метод. Если вы используете инструмент централизованного контроля версий вроде Subversion, нет возможности выбрать модель: инструмент даёт вам только совместный внос, и если вы захотите что-то ещё, вам придётся прикрутить свой собственный метод сверху (как например наложение патча вручную).

Хорошая распределенная версионная система, такая как Mercurial, поддерживает обе модели. И вы и ваши коллеги можете организовывать совместную работу, основываясь на ваших нуждах и предпочтениях, а не так, как принуждают вас ваши инструменты.

## 6.2.10. Когда разработка сталкивается с управлением ветвлениями

Единожды создав несколько распределенных репозиториев и начав распространять изменения между локальным и общим репозиториями, вы с вашей командой начнете становиться связанными, но столкнетесь с немного другой проблемой: управлением направлениями, в которых ваша команда может разом двинуться. Хотя эта тема тесно связана с тем, как ваша команда взаимодействует, она довольно глубока, чтобы заслужить собственное рассмотрение в Глава 8, *Управление релизами и ветками*.

# 6.3. Техническая сторона совместного использования

Остаток от этой главы посвящен вопросу обмена данными с вашими сотрудниками.

# 6.4. Неофициальный обмен с помощью hg serve

Команда **hg serve** чудесно подходит для маленьких, тесно связанных и быстро продвигающихся групп (разработчиков). Она также предоставляет отличную возможность пощупать использование команд Mercurial через сеть.

Запустите **hg serve** внутри репозитория, и через секунду она (команда) поднимет специальный HTTP сервер, принимающий подключения от любого клиента и предоставляющий данные этого репозитория, пока вы не отключите его. Любой, кто знает URL только что запущенного сервера, и способный подключиться к вашему компьютеру через сеть, может использовать браузер или Mercurial для чтения дынных этого репозитория. URL запущенного экземпляра **hg serve** на ноутбуке вероятно будет выглядеть приблизительно как `http://my-laptop.local:8000/`.

Команда **hg serve** это не полноценный web сервер. Вы можете делать с ним только две вещи:

- Разрешать людям просматривать репозиторий через обычный web браузер
- Использовать родной протокол Mercurial, так что люди смогут сделать **hg clone** или **hg pull** с вашего репозитория.

На практике команда **hg serve** не позволяет удаленным пользователям модифицировать ваш репозиторий. Она предназначена для использования в рамках «только чтение».

Если вы начинаете знакомство с Mercurial, нет ничего чтобы препядствовало вам сделать **hg serve** чтобы дать общий доступ к вашему локальному репозиторию и использовать команды вроде **hg clone**, **hg incoming** и так далее, так, будто репозиторий находится удаленно. Это может помочь вам быстро ознакомиться с использованием команд для репозиториев, расположеных в сети.

## 6.4.1. Несколько деталей к размышлению

Поскольку **hg serve** предоставляет доступ на чтение всем клиентам без идентификации, эту команду следует использовать только в окружении, где вы либо не имеете причин для беспокойства, либо имеете возможность определять, кто может подключиться к вашему хранилищу и получить из него данные.

Команда **hg serve** ничего не знает о брандмауэрах, которые могут быть установлены в вашей системе или сети. Она не обнаруживает и не управляет вашими брандмауэрами. Если сторонний человек не может обратиться к запущенному экземпляру **hg serve**, второе, что нужно проверить (после того, как убедитесь, что он использует правильный URL) — это конфигурация брандмауэра.

По умолчанию, **hg serve** принимает входящие соединения на 8000 порт. Если другой процесс уже прослушивает порт, который вы хотите использовать, вы можете указать другой порт с помощью ключа `-p`.

Обычно, когда **hg serve** запущен, он ничего не выводит, что может быть немного странным. Если вы хотите убедиться, что он действительно работает и вы можете передать URL своим сотрудникам, запускайте его с ключом `-v`.

# 6.5. Использование протокола Secure Shell (ssh)

Используя протокол Secure Shell (`ssh`), Вы можете безопасно получать и записывать изменения через сетевое соединение. Для того, что бы воспользоваться этим, Вы должны немного изменить конфигурацию на стороне клиента или сервера.

Если Вы не знакомы с ssh, то это протокол, который позволяет Вам безопасно осуществлять соединение с другим компьютером. Чтобы использовать его с Mercurial, Вам нужно настроить одну или несколько пользовательских учетных записей на сервере так, чтобы удаленные пользователи могли входить в систему и исполнять команды.

(Если Вы знакомы с ssh, приведенный ниже материал наверняка покажется Вам элементарным.)

## 6.5.1. Как читать и записывать, используя ssh URL-ы

Обычно, ssh URL выглядит подобным образом:

```
ssh://bos@hg.serpentine.com:22/hg/hgbook
```

1. Часть «`ssh://`» указывает Mercurial, что нужно использовать ssh протокол.

2. Часть bos представляет собой имя пользователя для подключения к серверу. Вы можете не указывать эту часть, если имя пользователя на удалённом сервере совпадает с Вашим локальным именем пользователя.

3. «`hg.serpentine.com`» представляет собой имя хоста для подключения.

4. «`:22`» указывает номер порта для подключения к серверу. По умолчанию используется порт 22, поэтому Вам нужно указывать эту часть только если используется не стандартный 22 порт.

5. Остальная часть URL представляет собой локальный путь к хранилищу на серевере.

Существует множество неразберихи по прочтению ssh-URL, так как не существует стандарта его интерпретации. Некоторые программы ведут себя одним образом, другие — иначе, когда речь идет о таких путях. Это не идеальная ситуация, и она вряд ли изменится. Пожалуйста, прочтите внимательно следующие абзацы.

Mercurial рассматривает путь к хранилищу на сервере относительно домашней директории удаленного пользователя. Например, если домашняя директория пользователя `foo` на сервере `/home/foo`, тогда ssh-URL, который содержит компонент пути `bar` (`ssh://foo@myserver/bar`), в действительности ссылается на директорию `/home/foo/bar`.

Если Вы хотите указать путь относительно домашней директории другого пользователя, можете использовать путь, который начинается с тильды, за которой следует имя пользователя(назовем его `otheruser`), к примеру.

```
ssh://server/~otheruser/hg/repo
```

Если Вы действительно хотите указать абсолютный путь на сервере, укажите в начале два слэша, как в примере.

```
ssh://server//absolute/path
```

## 6.5.2. Выбор ssh-клиента для Вашей системы

Почти все Unix-подобные системы поставляются с предустановленным OpenSSH. Если Вы используете такую систему, запустите `which ssh`, что бы узнать, установлен ли **ssh** (он, как правило, находится в `/usr/bin`). В маловероятном случае, если его нету, обратитесь к документации на вашу систему, чтобы выяснить как его установить.

В windows, пакета tortoisehg поставляется в комплекте с версией отличной команде **plink** Симона Татама, и вам не нужно делать какой-либо дополнительной конфигурации.

## 6.5.3. Генерация криптографической пары (открытого и секретного ключей)

Чтобы не вводить пароли каждый раз при использовании ssh-клиента, рекомендую сгенерировать криптографическую пару.

### Криптографической ключи не являются обязательными

Mercurial ничего не знает об аутентификации или SSH-ключах. Вы можете, если хотите, проигнорировать этот раздел и следующий, пока не устанете от многократно ввода пароля SSH.

- На Unix-подобной операционной системы, можно сделать командой **ssh-keygen**.

  В windows, если вы используете TortoiseHg, вам, возможно, потребуется скачать команду с **puttygen** с веб-сайта PuTTY [http://www.chiark.greenend.org.uk/~sgtatham/putty] для генерации пары ключей. Смотрите документацию [http://the.earth.li/~sgtatham/putty/0.60/htmldoc/Chapter8.html#pubkey-puttygen] к **puttygen** об использовании этой команды.

При генерации криптографической пары весьма целесообразно защитить секретный ключ с помощью парольной фразы (единственный случай, когда Вам не захочется этого делать — это когда вы используете ssh протокол для автоматизации задач в защищённой сети)

Одной только генерации пары ключей не достаточно. Вам также необходимо добавить открытый ключ к множеству авторизованных ключей пользователя, который подключается удаленно. Для серверов, которые используют OpenSSH (подавляющее большинство), это означает, что нужно добавить публичный ключ к списку в файле под названием `authorized_keys` в директории `.ssh` пользователя.

На Unix-подобных системах, Ваш открытый ключ будет иметь расширение `.pub`. Если Вы используете **puttygen** на Windows, то Вы можете сохранить открытый ключ в файл или вставить из окна puttygen в файл `authorized_keys`.

## 6.5.4. Использование агента аутентификации

Аутентификационный агент — демон, который хранит парольную фразу в памяти (и она будет утеряна, когда Вы выйдете из системы). Ключевая фраза передаётся ssh-клиенту, если он запущен и опрашивает агента для получения парольной фразы. Если агент аутентификации не запущен, или не хранит необходимой парольной фразы, то пароль будет запрашиваться вский раз, когда Mercurial попробует установить соединение с сервером от Вашего имени (при отправке или получении Вами изменений).

Недостатком хранения парольных фраз в агенте является возможность их восстановления хорошо подготовленными злоумышленниками. Вам решать, насколько это приемлемый для Вас риск. С другой стороны, это избавляет от многократного повторного ввода.

- На Unix-подобных системах агент называется **ssh-agent**, который запускается автоматически когда вы входите в систему. Для того, что бы агент запомнил парольную фразу в хранилище, нужно использовать команду **ssh-add**.

- На Windows, если вы используете TortoiseHg, команда **pageant** выступает в качестве агента. Как и **puttygen**, вам необходимо скачать **pageant** [http://www.chiark.greenend.org.uk/%7Esgtatham/putty/download.html] с веб-сайта PuTTY и прочесть его документацию [http://the.earth.li/~sgtatham/putty/0.60/htmldoc/Chapter9.html#pageant]. Команда **pageant** добавляет иконку в системный трей, с помощью которой, вы можете управлять сохраненными парольными фразами.

## 6.5.5. Правильная настройка сервера.

Настройка ssh кропотливая, и если Вы новичек, то есть ряд вещей, которые могут пойти не так. Добавим сверху ещё и Mercurial, и появится ещё больше причин почесать затылок. Большинство потенциальных проблем возникают не на стороне клиента, а на стороне сервера. Хорошая новость заключается в том, что как только Вы получили рабочую конфигурацию, то, как правило, она будет работать неограниченное время.

Перед тем, как пытаться использовать Mercurial с доступом через ssh сервер, лучше для начала убедиться в том, что вы можете нормально использовать **ssh** или команду **putty** для доступа к серверу. Если вы столкнетесь с проблемами непосредственно при использовании этих команд, то наверняка Mercurial также не будет работать. Более того, он будет скрывать проблему. Когда Вы захотите настроить работу Mercurial по ssh, то сначала убедитесь, что подключения клиента по ssh работают, и только потом беспокойтесь о том, что это проблема Mercurial.

Первое, что нужно сделать, что бы убедиться в работоспособности серверной стороны — попробовать войти в систему с другой машины. Если вы не можете войти с помощью **ssh** или **putty**, полученное Вами сообщение об ошибке может дать несколько подсказок, что именно идет не так. Наиболее распространенными проблемами являются следующие:

- Если получаете сообщение «connection refused» — либо не запущен демон SSH на стороне сервера, либо он не доступен из-за конфигурации межсетевого экрана.
- Если Вы получаете сообщение об ошибке «no route to host» — Вы ввели неправильный адрес сервера или серьёзно заблокированы межсетевым экраном, который вообще не признаёт существование этого сервера.
- Если Вы получаете ошибку «permission denied» — Вы ошиблись при вводе логина на сервере, опечатались в ключевой фразе или пароле пользователя.

В итоге, если у Вас возникают проблемы подключения к даемону ssh сервера, первое в чем нужно убедиться — запущен ли он вообще. На многих системах он установлен, но по-умолчанию отключен. После того, как закончили этот шаг, Вы должны убедиться в том, что брандмауэр на стороне сервера разрешает входящие соединения к порту, на котором слушает ssh даемон (обычно 22). Не беспокойтесь о других возможных ошибках, пока не проверите эти две.

Если Вы используете агента аутентификации на клиентской стороне для хранения Ваших парольных фраз ключей, Вы можете подключаться к серверу без ввода парольной фразы или пароля. Если Вас спрашивают парольную фразу, есть несколько возможных причин:

- Возможно, Вы забыли использовать **ssh-add** или **pageant** для сохранения парольной фразы.
- Возможно, Вы ввели парольную фразу не для того ключа.

Если вас спрашивают пароль для аккаунта на сервере, есть несколько возможных причин:

- Права доступа к пользовательской домашней директории или поддиректории `.ssh` имеет лишние разрешения. В результате этого ssh даемон не доверяет такому `authorized_keys` файлу. Например, право на запись для группы в домашнюю или `.ssh` директорию часто является причиной такого поведения.
- Файл `authorized_keys` пользователя тоже может быть причиной проблем. Если кроме пользователя ещё кто-либо может в него писать, то ssh даемон не станет доверять такому файлу и не будет его использовать.

В идеальном случае, Вы должны успешно выполнить следующую команду и получить её вывод: текущую дату и время на сервере.

```
ssh myserver date
```

Если на Вашем сервере при входе выполняются скрипты, выводящие баннеры и прочий мусор, даже при запуске неинтерактивных сессий, вроде приведённой выше — Вы должны избавиться от них или настроить сервер, чтобы это безобразие появлялось только при интерактивном входе. Иначе этот мусор будет, по-крайней мере, загромождать вывод Mercurial. Также он создаёт потенциальные проблемы для удалённого выполнения команд Mercurial. Mercurial пытается определять и игнорировать баннеры в неинтерактивных **ssh** сессиях, но не всегда успешно. (Если Вы изменяли логин-скрипты на сервере, то чтобы убедититься в их работоспособности в интерактивной сессии — проверьте код, возвращаемый командой `tty -s`.)

Как только Вы проверите простой ssh-доступ на сервер, следующим шагом к победе будет проверка выполнения Mercurial на сервере. Следующая команда должна успешно исполниться:

```
ssh myserver hg version
```

Если вы получили ошибку вместо нормального вывода команды **hg version**, то это, как правило, из-за неустановленного Mercurial в `/usr/bin`. Не беспокойтесь, если это так, то вам не нужно этого делать. Но Вы должны проверить несколько возможных проблем.

- Mercurial вообще действительно установлен на сервере? Я знаю, что звучит банально, но это стоит проверить!

- Возможно в вашей командной оболочке неверно указаны пути поиска исполняемых файлов (как правило определяются переменной окружения `PATH`).

- Может быть переменная окружения `PATH` у вас правильно устанавливается только при интерактивном входе в систему. Это может случиться, если вы установили путь в неверном скрипте входа. Смотрите документацию по своей командной оболочке для подробностей.

- Переменная среды `PYTHONPATH` может потребоваться для определения пути к модулям Mercurial. Может быть она вообще не установлена; содержит неверное значение; или устанавливается только при интерактивной сессии оболочки.

Если Вы успешно выполнили **hg version** по ssh соединению, то всё готово. Ваш сервер и клиент настроены. Вы можете начинать использовать Mercurial для доступа к хранилищам этого пользователя на этом сервере. При возникновении проблем на этом этапе, попробуйте использовать опцию `--debug` для более полного представления о происходящем.

## 6.5.6. Использование сжатия по ssh

Mercurial не сжимает данные, передаваемые по ssh, поскольку ssh протокол сам может прозрачно это делать. Однако, по-умолчанию, ssh-клиенты не используют компрессию передаваемых данных.

При работе по сети, отличной от высокоскоростной LAN, (даже по беспроводной сети), использование сжатия значительно ускоряет выполнение сетевых операций Mercurial'a. Например, кто-то измерил, что сжатие уменьшило время на клонирование через WAN достаточно большого хранилища с 51 минуты до 17 минут.

Оба клиента, и **ssh**, и **plink**, понимают опцию `-C`, которая включает сжатие передаваемых данных. Вы легко можете разрешить компрессию по ssh для Mercurial, отредактировав ваш файл `~/.hgrc`. Вот как сделать это в обычном **ssh** на unix-подобных операционных системах, для примера.

```
[ui]
ssh = ssh -C
```

Если Вы используете **ssh**, то можете настроить постоянное использование сжатия при работе с Вашим сервером. Для этого поправьте файл `.ssh/config` (может не существовать), как приведено ниже.

```
Host hg
  Compression yes
  HostName hg.example.com
```

Здесь определён синоним, `hg`. При его использовании в командной строке **ssh** или в `ssh`-URL Mercurial'a, выполняется подключение к хосту `hg.example.com` с использованием сжатия. Это даёт Вам сразу и короткое имя для ввода, и включает компрессию, что хорошо и по-отдельности, и вместе.

# 6.6. Работа по HTTP с использованием CGI

Самый простой способ раздавать один или более репозиториев на постоянной основе, это использовать веб-сервер и встроенную реализацию CGI в Mercurial.

В зависимости от ваших целей, конфигурирование интерфейса CGI системы Mercurial может занять от нескольких мгновений до нескольких часов.

Мы начнем с самого простого примера и далее перейдем к более сложной конфигурации. Даже в обычных случаях вам непременно потребуется умение понимать и изменять конфигурацию вашего веб-сервера.

### Требуется высокая надёжность

Конфигурирование веб-сервера это сложная, кропотливая деятельность, сильно зависящая от всей системы. Я не могу дать указания для всех случаев, с которыми вы столкнетесь. Пожалуйста, будьте внимательны и рассудительны в следующей части работы. Будьте готовы совершить множество ошибок и провести бессоные ночи, читая логи ошибок сервера.

> Если у вас нет сильного желания настраивать конфигурацию, снова и снова, или насущной необходимости размещать собственные услуги, вы можете попробовать одну из публичных услуг хостинга, которые я упоминал ранее.

## 6.6.1. Список проверок конфигурации веб-сервера

Прежде чем вы продолжите, обязательно уделите пару минут для проверки некоторых настроек вашей системы.

1. Имеется ли в вашей системе установленный web-сервер? Mac OS X и разные дистрибутивы Linux поставляются с Apache, но во многих других операционных системах веб-серверы могут быть не установлены.

2. Если web-сервер установлен, запущен ли он? В большинстве систем, даже если он есть, он может быть отключен по умолчанию.

3. Позволяет ли конфигурация вашего сервера выполнять программы CGI в директории, где вы планируете это делать? Большинство серверов по умолчанию явно запрещают запуск CGI программ.

Если ваш web-сервер не установлен или у вас нет достаточного опыта конфигурирования Apache, вам следует использовать web-сервер `lighttpd` вместо Apache. Apache известен за причудливость и запутаность конфигурации. Хотя `lighttpd` менее функционален, чем Apache, большинство из его недостатков не относятся к обслуживанию хранилищ Mercurial. И с `lighttpd` несомненно намного легче начать работу, чем с Apache.

## 6.6.2. Базовая конфигурация CGI

Обычно в Unix-системах поддиректория web-контента пользователя имеет имя `public_html` и расположена в домашней директории. Файл с именем `foo` в этой директории будет доступен по ссылке `http://www.example.com/username/foo`.

Для начала работы найдите скрипт `hgweb.cgi`, который должен быть в вашей копии Mercurial. Если вы не можете быстро найти его локальную копию в вашей системе, просто скачайте его из основного хранилища Mercurial http://www.selenic.com/repo/hg/raw-file/tip/hgweb.cgi.

Вам следует скопировать данный скрипт в вашу директорию `public_html` и задать ему права на исполнение.

```
cp .../hgweb.cgi ~/public_html
chmod 755 ~/public_html/hgweb.cgi
```

Аргумент `755` команды **chmod** — установка прав на выполнение скрипта; он описывает, что скрипт может читаться и выполняться всеми пользователями, а права на запись у «группы» и «других» пользователей не установлены. Если бы Вы оставили всем права на запись, то модуль `suexec` сервера Apache скорее всего отказался бы выполнять скрипт. `suexec` так же требует, чтобы директория, в которой находится скрипт, не была доступна для записи другим пользователям.

```
chmod 755 ~/public_html
```

### 6.6.2.1. Где могут возникнуть проблемы?

Как только вы скопировали CGI скрипт, попробуйте перейти браузером по адресу `http://myhostname/~myuser/hgweb.cgi`, но не падайте духом перед неудачными попытками. Вероятней всего вы не сможете открыть ссылку, и на это есть множество причин. На самом деле вы спотыкаетесь на одной из ошибок, приведенных ниже, поэтому, пожалуйста, прочтите внимательно. Это те ошибки, с которыми я сталкивался в Fedora 7 с только что установленным Apache и пользовательским аккаунтом, который я специально создал для этого.

Ваш веб-сервер может иметь настройки, запрещающие пользовательский веб-контент. При использовании Apache найдите в его конфигурационном файле директиву `UserDir`. Если она отсутствует, то отображение пользовательских директорий запрещено. В противном случае следующее за `UserDir` строковое значение определяет имя директории в домашнем каталоге пользователя, контент из которой будет отдаваться Apache'м. Например, `public_html`.

Ваши настройки безопасности могут быть слишком ограничивающими. Web сервер должен иметь возможность доступа к вашей домашней директории и вложенные в `public_html` директории, а также файлам в них. Вот рецепт, чтобы помочь вам сделать соотвествующие настройки безопасности.

```
chmod 755 ~
find ~/public_html -type d -print0 | xargs -0r chmod 755
find ~/public_html -type f -print0 | xargs -0r chmod 644
```

Другая возможная проблема с правами состоит в том, что вы можете получить пустое окно, когда пытаетесь загрузить скрипт. В этом случае очень похоже что ваши настройки безопасности также слишком ограничивают выполнение подсистемой запуска скриптов Apache скриптов у которых, к примеру, стоят права записи для группы или всех пользователей.

Ваш web сервер может быть сконфигурирован с отключенными опциями выполнения CGI программ. Вот файл пользовательской конфигурации по умолчанию из моей ОС Fedora:

```
<Directory /home/*/public_html>
  AllowOverride FileInfo AuthConfig Limit
  Options MultiViews Indexes SymLinksIfOwnerMatch IncludesNoExec
  <Limit GET POST OPTIONS>
    Order allow,deny
    Allow from all
  </Limit>
  <LimitExcept GET POST OPTIONS>
    Order deny,allow Deny from all
  </LimitExcept>
</Directory>
```

Если вы обнаружите похожую на эту группу `Directory` в вашей конфигурации Apache, добавьте `ExecCGI` в конец директивы `Options`, если она отсуствует и перезапустите web сервер.

Если при открытии через web браузер Ваш Apache выдаёт исходный текст CGI скрипта, вместо результатов его выполнения, то Вам надо добавить или раскомментировать (если уже есть) такую директиву:

```
AddHandler cgi-script .cgi
```

Следующая возможная проблема, которую вам придется решить, может быть красочный вывод трассировки Python, который будет говорить о том что он не может импортировать связанный с `mercurial` модуль. Это уже прогресс! Теперь сервер пытается выполнить CGI программу. Это ошибка вероятно произойдет, если вы проведете инсталяцию Mercurial для себя, вместо версии установки в систему.Помните что ваш web сервер запускает ваши CGI программы без каких-либо переменных среды, которые присуствуют в интерактивном сеансе. Если случилась такая ошибка, отредактируйте ваш `hgweb.cgi` файл и следйуте указаниям как установить внутри него правильную переменную среды `PYTHONPATH`.

В конце-концов вы увидите другую трассировку Python, которая будет жаловаться что не может найти `/path/to/repository`. Отредактируйте ваш `hgweb.cgi` скрипт и замените в нем строку `/path/to/repository` на полный путь к репозиторию, который вы хотите отобразить в web сервере.

В этом месте, когда вы попробуете перезагрузить web страничку, вы должны увидеть простой HTML интерфейс истории вашего репозитория. Фуф!

## 6.6.2.2. Настройка lighttpd

Чтобы быть исчерпывающим в моих экспериментах, я попробовал настроить всё более популярный `lighttpd` веб сервер для обслуживания того же репозитория, который я выше описывал с Apache. Я уже преодолел все проблемы выделенные с Apache, многие из которых не связаны с сервером. В результате я справедливо убедился в правильности разрешений на мои файлы и директории, и в том, что мой скрипт `hgweb.cgi` был надлежаще исправлен.

Раз у меня работал Apache, я смог быстро заставить `lighttpd` обслуживать репозиторий (другими словами, даже если вы попробуете использовать `lighttpd`, вам придётся читать раздел про Apache). Для начала я отредактировал секцию `mod_access` в его конфигурационном файле, чтобы включить `mod_cgi` и

`mod_userdir`, которые по умолчанию были отключены в моей системе. Затем я добавил несколько строк в конце конфигурационного файла, чтобы настроить эти модули.

```
userdir.path = "public_html"
cgi.assign = (".cgi" => "" )
```

После этого `lighttpd` отлично запустился. Если бы я настраивал `lighttpd` до Apache, то почти наверняка столкнулся бы с теми же проблемами конфигурации системы, что и в Apache. Однако, я нашел, что `lighttpd` заметно проще конфигурировать чем Apache, несмотря на то, что я уже использовал Apache больше десяти лет, и это был мой первый опыт работы с `lighttpd`.

## 6.6.3. Настройка доступа к нескольким хранилищам с помощью одного CGI-скрипта

Скрипт `hgweb.cgi` позволяет Вам опубликовать одно хранилище, что является досадным ограничением. Если Вы хотите опубликовать больше одного, использовав несколько копий этого скрипта с разными именами без раздражения себя, лучшим выбором будет использовать скрипт `hgwebdir.cgi`.

Процедура настройки `hgwebdir.cgi` несколько сложнее чем `hgweb.cgi`. Для начала, Вы должны получить копию скрипта. Если у Вас его нет под рукой, можете загрузить из хранилища Mercurial по ссылке http://www.selenic.com/repo/hg/raw-file/tip/hgwebdir.cgi.

Вы должны скопировать этот скрипт в каталог `public_html` и убедиться в том, что он исполняемый.

```
cp .../hgwebdir.cgi ~/public_html
chmod 755 ~/public_html ~/public_html/hgwebdir.cgi
```

После базовой настроки попробуйте открыть `http://myhostname/~myuser/hgwebdir.cgi` в Вашем браузере. Должен отображаться пустой список хранилищ. Если Вы получаете пустое окно или сообщение об ошибке, попробуйте просмотреть список потенциальных проблем в Раздел 6.6.2.1, «Где могут возникнуть проблемы?».

Скрипт `hgwebdir.cgi` зависит от внешнего файла конфигурации. По-умолчанию он ищет файл `hgweb.config` в этом же каталоге. Вы должны создать его и сделать общедоступным. Формат этого файла похож на формат «ini» файлов в Windows, который распознается в Python модулем `ConfigParser` [web:configparser].

Самый простой способ настроить `hgwebdir.cgi` — использовать раздел, который называется `collections`. Это позволит автоматически опубликовать все хранилища в каталоге, который Вы укажите. Этот раздел должен выглядеть следующим образом:

```
[collections]
/my/root = /my/root
```

Mercurial воспринимает это так: смотрит имя директории с правой стороны от знака «=»; ищет репозитории внутри этой директории; и использует текст с левой стороны чтобы обрезать совпадающий текст в именах, фактически отображающихся в веб интерфейсе. Оставшаяся часть после обрезания пути называется «виртуальный путь».

В упомянутом выше примере, если мы имеем репозиторий, чей локальный путь `/my/root/this/repo`, CGI скрипт обрежет начальную часть `/my/root` в имени и опубликует репозиторий с виртуальным путём `this/repo`. Если базовый URL нашего CGI скрипта `http://myhostname/~myuser/hgwebdir.cgi`, тогда полный путь для этого репозитория будет `http://myhostname/~myuser/hgwebdir.cgi/this/repo`.

Если мы заменим `/my/root` с левой стороны этого примера на `/my`, тогда `hgwebdir.cgi` будет просто обрезать `/my` из имени репозитория и будет предоставлять виртуальный путь `root/this/repo` вместо `this/repo`.

Скрипт `hgwebdir.cgi` будет рекурсивно искать в каждой из перечисленных в секции `collections` конфигурационного файла директории, но `не` будет рекурсивно искать в найденых репозиториях.

Механизм `collections` позволяет легко публиковать множество репозиториев в манере «выстрелил и забыл». Вам нужно только один раз установить CGI скрипт и конфигурационный файл. Впоследствии вы можете публиковать и прятать репозиторий в любое время, просто помещая внутрь или убирая из дерктории, за которой наблюдает `hgwebdir.cgi`.

#### 6.6.3.1. Явное определение публикуемых репозиториев

В дополнение в механизму `collections`, скрипт `hgwebdir.cgi` позволяет вам публиковать специфичный лист репозиториев. Для того сделать это создайте секцию `paths` с содержимым подобным этому:

```
[paths]
repo1 = /my/path/to/some/repo
repo2 = /some/path/to/another
```

В этом случае, виртуальный путь (показывающаяся в URL часть) расположен слева в каждом определении, тогда как путь к репозиторию находиться справа. Обратите внимание, что нет никакой зависимости между выбранным вами виртуальным путём и положением ропозитория в вашей файловой системе.

Если вы хотите, то можете использовать `collections` и механизм `paths` одновременно в одном конфигурационном файле.

Остерегайтесь дублирования виртуальных путей

Если несколько репозиториев будут иметь один и тот же виртуальный путь, `hgwebdir.cgi` не будет сообщать об ошибке. Вместо этого он будет вести себя непредсказуемо.

### 6.6.4. Загрузка исходных архивов

Web интерфейс Mercurial позволяет пользователям скачивать архив любой ревизии. Этот архив будет содержать снимок рабочей директории, но не будет содержать копию данных репозитория.

По умолчанию эта возможность отключена. Если вы хотите включить её, вам нужно будет добавить элемент `allow_archive` в секцию `web` вашего `~/.hgrc` файла.

### 6.6.5. Опции настройки веб интерфейса

Web интерфейс Mercurial (команда **hg serve**, `hgweb.cgi` и скрипты `hgwebdir.cgi`) имеют несколько опций, которые вы можете настроить. Они располагаются в секции `web`.

- `allow_archive` Определяет который (если нужно) из методов загрузки архивов поддерживает Mercurial. Если вы включите эту опцию, пользователи веб интерфейса смогут скачать архив любой версии репозитория, который они просматривают. Чтобы включить возможность загрузки архивов, этот элемент должен содержать последовательность слов, перечисленных ниже.
  - `bz2`: Архив **tar**, сжатый по алгоритму `bzip2`. Имеет наилучшую степень сжатия, но использует больше процессорного времени на сервере.
  - `gz`: Архив **tar**, сжатый по алгоритму `gzip`.
  - `zip`: Архив **zip**, сжатый используя алгоритм LZW. Этот формат имеет наихудшее сжатие, но широко используется в мире Windows.

  Если у вас будет пустой список или не будет записи `allow_archive` вообще, то эта возможность будет отключена. Ниже дан пример как включить все три поддерживаемых формата.

  ```
  [web]
  allow_archive = bz2 gz zip
  ```

- `allowpull`: Boolean. Определяет позволяет ли web интерфейс удаленным пользователям использовать команды **hg pull** и **hg clone** через HTTP. Если эта опция установлена в `no` или `false`, то только «пользовательская» часть web интерфейса доступна.
- `contact`: String. Строка свободного формата (но предпочтительно короткая) отождествляющая человека или группу создавшую репозиторий. Она обычно содежит имя и email адрес человека или листа почтовой рассылки. Эту запись обычно располагают в собственном `.hg/hgrc` репозитория, но так же можно расположить в глобальном `~/.hgrc`, если каждый репозиторий имеет единственного мейнтейнера.

- `maxchanges`: Integer. Максимальное количество наборов изменений отображаемое на одной странице по умолчанию.
- `maxfiles`: Integer. Максимальное количество модифицированных файлов отображаемое на одной странице по умолчанию.
- `stripes`: Integer. Если web интерфейс отображает разноцветные «полосы», облегчающие визуальное восприятие выводимых строк, то значение этого параметра будет означать количество строк текста в каждой цветной строке.
- `style`: Управляет шаблоном Mercurial, использующимся для отображения web интерфейса. Mercurial поставляется с несколькими веб-шаблонами.
  - `coal` — чёрнобелый.
  - `gitweb` — эмулирует веб-интерфейс git-а
  - `monoblue` — используются синие и серый цвета.
  - `paper` — используется по умолчанию.
  - `spartan` — ранее использовался по умолчанию.

  Вы можете также определить свой шаблон; более детально эта возможность обсуждается в Глава 11, *Настройка вывода Mercurial*. Ниже вы можете видеть как включить стиль `gitweb`.

```
[web]
style = gitweb
```

- `templates`: Path. Path содержит имя каталога в котором ищутся файлы шаблонов. По умолчанию Mercurial ищет их в директории где установлен.

Если вы используете `hgwebdir.cgi`, вы можете поместить для удобства некоторые элементы конфигурации в секцию `web` файла `hgweb.config`, вместо файла `~/.hgrc`. Это элементы — `motd` и `style`.

### 6.6.5.1. Опции, специфичные для индивидуального репозитория

Некоторые элементы раздела `web` должны быть помещены в локальный файл `.hg/hgrc`, вместо помещения их в глобальный `~/.hgrc` для пользователя.

- `description`: String. Строка в свободной форме (предпочтительно небольшая), которая описывает сущность репозитория или его содержимое.
- `name`: String. Строка с именем репозитория для использования в web интерфейсе. Она имеет больший приоритет (переопределяет) нежели имя по умолчанию, которое является последним компонентом пути репозитория.

### 6.6.5.2. Опции, специфичные для команды hg serve

Некоторые элементы секции `web` файла `~/.hgrc` используются только с командой **hg serve**.

- `accesslog`: Path. Имя файла для аудита доступа. По умолчанию команда **hg serve** пишет свою информацию в stdout, а не в файл. Элементы лог файла пишутся в стандартном «объединенном» формате файлов, который используется почти всеми web серверами.
- `address`: String. Локальный адрес, который должен слушать web сервер в ожидании подключений к нему. По умолчанию сервер слушает все адреса.
- `errorlog`: Path. Имя файла в который будут выводиться сообщения об ошибках. По умолчанию команда **hg serve** записывает эту информацию в stdout, а не в файл.
- `ipv6`: Boolean. Определяет использовать или нет IPv6 протокол при работе сервера. По умолчанию IPv6 не используется.

- `port`: Integer. Определяет номер порта (число) который будет слушать web сервер. По умолчанию значение равно 8000.

### 6.6.5.3. Выбор правильного файла `~/.hgrc` для добавления элементов в секцию `web`.

Важно помнить, что веб сервера вроде Apache или `lighttpd` запускаются под пользовательским аккаунтом (UID-ом), который отличается от того, под которым работаете вы. Скрипты CGI, запускаемые вашим сервером, такие как `hgweb.cgi`, обычно запускаются под тем же пользователем (UID-ом).

Если вы добавляете элементы в секцию `web` вашего персонального `~/.hgrc`, то CGI скрипты не станут читать этот файл. Произведенные там настройки влияют только на поведение команды **hg serve**, когда вы ее используете. Чтобы заставить CGI скрипты видеть ваши параметры настроек создайте файл `~/.hgrc` в домашнем каталоге пользователя под кем запускается web сервис, или добавьте эти параметры к основному системному файлу `hgrc`.

## 6.7. Системный файл конфигурации

На Unix-подобных операционных системах совместного использования несколькими пользователями (например, сервер, на котором люди публикуют изменения), часто имеет смысл создавать некое глобальное поведение по умолчанию, например, какие темы использовать в веб-интерфейсах.

Если файл с именем `/etc/mercurial/hgrc` существует, Mercurial будет читать во время запуска и применять параметры конфигурации, которые находит в этом файле. Он также будет искать файлы с расширением `.rc` в каталоге `/etc/mercurial/hgrc.d`, и применять параметры конфигурации, которые находит в этих файлах.

### 6.7.1. Делаем Mercurial более доверенным.

Один из ситуации, в котором глобальный `hgrc` может быть полезен, если пользователи вытягивают изменения, принадлежащие другим пользователям. По умолчанию Mercurial не будет доверять большинству параметров конфигурации из файла `.hg/hgrc` в репозитории, который принадлежит другому пользователю. Если мы клонируем или вытягиваем изменения такого хранилища, Mercurial выведет предупреждение о том, что он не доверяет параметрам из `.hg/hgrc`.

Если все участники команды находятся в той или иной группе Unix и доверяют конфигурации друг друга, или мы хотим доверять конкретным пользователям, мы можем переопределить в Mercurial скептическое отношение по-умолчанию путем создания общесистемного файла `hgrc`, такого как:

```
# Save this as e.g. /etc/mercurial/hgrc.d/trust.rc
[trusted]
# Trust all entries in any hgrc file owned by the "editors" or
# "www-data" groups.
groups = editors, www-data

# Trust entries in hgrc files owned by the following users.
users = apache, bobo
```

# Глава 7. Имена файлов и шаблоны совпадений

Mercurial предоставляет механизмы, которые позволят вам работать с именами файлов единообразным и выразительным образом.

## 7.1. Простое именование файлов

Mercurial использует единообразную технику «скрытую под капотом» для обработки имен файлов. Каждая команда ведет себе одинаково относительно имен файлов. Путь по которому команды работают с именами файламов следующий.

Если вы явно указываете имена реальных файлов, Mercurial работает именно с теми файлами, которые вы ожидаете.

```
$ hg add COPYING README examples/simple.py
```

Когда вы указываете имя директории, Mercurial интерпретирует это как «сделай что либо со всеми файлами в этой директории и во вложенных директориях». Mercurial обрабатывает файлы и поддиректории в указанной директории в алфавитном порядке. Когда Mercurial дойдет до каталога, то он рекурсивно спустится в него и произведет теже действия что и выше, перед продолжением обработки в текущем каталоге.

```
$ hg status src
? src/main.py
? src/watcher/_watcher.c
? src/watcher/watcher.py
? src/xyzzy.txt
```

## 7.2. Запуск команд без указания имен файлов

Команды Mercurial, которые работают с именами файлов имеют полезные, заданные по умолчанию поведения, когда вы вызываете их без параметров имен файлов или шаблонов имен. Ожидаемое поведение зависит от того, что делает команда. Вот несколько небольших правил которые вы можете использовать, чтобы предсказать то, что вероятно сделает команда, если вы не укажите имена с которой ей работать.

- Большинство команд работают со всей рабочей директорией. Так например работает команда **hg add**.
- Если результат команды нельзя или очень сложно отменить, то она вынудит вас указать как минимум одно имя или шаблон (см. ниже). Например, это защитит вас от случайного удаления файлов запуском команды **hg remove** без аргументов.

Легко обойти стандартное поведение если оно вас не устраивает. Если команда как правило работает с целым рабочим каталогом, вы можете ссылаться на него только в текущем каталоге или его подкаталогах, давая ему имя «`.`».

```
$ cd src
$ hg add -n
adding ../MANIFEST.in
adding ../examples/performant.py
adding ../setup.py
adding main.py
adding watcher/_watcher.c
adding watcher/watcher.py
adding xyzzy.txt
$ hg add -n .
adding main.py
adding watcher/_watcher.c
adding watcher/watcher.py
adding xyzzy.txt
```

Как видно выше, некоторые команды обычно выводят относительные имена от корня репозитория, даже если вы выполняете их в подкаталоге. Этаже команда напечатает имена файлов относительно вашего подкаталога, если вы дадите ей явные имена. Давайте выполним **hg status** из подкаталога и заставим ее работать со всем рабочим каталогом, печатая имена файлов относительно нашего подкаталога, передав ей вывод команды **hg root**.

```
$ hg status
A COPYING
A README
A examples/simple.py
? MANIFEST.in
? examples/performant.py
? setup.py
? src/main.py
? src/watcher/_watcher.c
? src/watcher/watcher.py
? src/xyzzy.txt
$ hg status `hg root`
A ../COPYING
A ../README
A ../examples/simple.py
? ../MANIFEST.in
? ../examples/performant.py
? ../setup.py
? main.py
? watcher/_watcher.c
? watcher/watcher.py
? xyzzy.txt
```

# 7.3. Информация о том что произошло;

К примеру команда **hg add**, использованная в предыдущем разделе илюстрирует нечто большее, чем просто помощь по командам Mercurial. Если команда воздействует на файл, который вы явно не задали в командной строке, она обычно выведет имя файла, чтобы вас не удивляли изменения.

Это правило наименьшего удивления — если вы явно именуете файл в командной строке, нет никакого смысла в повторении его снова вам. Если же Mercurial работает с файлом неявно, потому что вы не указали имени, или директории, или шаблона (см. ниже), безопаснее сказать вам что произошло.

Для команд, которые ведут себя так есть простой способ заставить их замочать, используя опцию `-q`. Вы также можете заставить их печатать имена файлов, даже если они явно заданы вами, используя опцию `-v`.

# 7.4. Использование шаблонов для указания файлов

В дополнение к работе с именами файлов и директорий, Mercurial предоставляет вам возможность использования шаблонов. Шаблоны Mercurial достаточно выразительны.

В Unix-подобных системах (Linux, MaxOS, и т.д.), работу по сопоставлению имен файлов шаблонам обычно выполняет интерпретатор (shell). В этих системах вы должны явно указать Mercurial, что указанное имя является шаблоном. В системе Windows, интерпретатор не раскрывает шаблоны, так что Mercurial автоматически идентифицирует имена как шаблоны, и раскрывает их для вас.

Для указания шаблона вместо обычного имени в командной строке используется следующий синтаксис:

```
syntax:patternbody
```

Таким образом, шаблон идентифицируется короткой строкой, указывающей тип шаблона, с последующим двоеточием, и содержимым шаблона за ним.

Mercurial поддерживает два типа синтаксиса шаблонов. Наиболее часто употребляется синтаксис `glob`. Это тот же самый вид сопоставления, используемый интерпретатором Unix и также должен быть знаком пользователям командной строки Windows.

Когда Mercurial выполняет автоматическое сопоставление с шаблоном в Windows, она использует `glob` синтаксис. Вы также можете опустить префикс «`glob:`» в Windows, но более безопасно явно указать его.

Синтаксис `re` более могущественный. Он предполагает что вы указали регулярное выражение (regexps) как шаблон.

Между прочим, в примерах далее обратите внимание что я делаю все возможное обрамляя все мои шаблоны в кавычки, чтобы они не раскрывались интерпретатором командной строки прежде чем их обработает Mercurial.

## 7.4.1. `glob`-шаблоны в стиле shell

Этот раздел посвящен краткому обзору видов шаблонов, которые вы можете использовать в glob шаблонах.

Символ «`*`» (звездочка) соотвествует любой строке в пределах одного каталога.

```
$ hg add 'glob:*.py'
adding main.py
```

Шаблон «`**`» (две звездочки) соответствует любой строке включая подкаталоги. Это не стандартный шаблон Unix, но он используется в некоторых популярных интерпретаторах командной строки и очень полезен.

```
$ cd ..
$ hg status 'glob:**.py'
A examples/simple.py
A src/main.py
? examples/performant.py
? setup.py
? src/watcher/watcher.py
```

Шаблон «`?`» соотвествует любому одиночному символу.

```
$ hg status 'glob:**.?'
? src/watcher/_watcher.c
```

Символ «`[`» открывает класс символов. Этот шаблон соотвествует любому одиночному символу в указанном классе. Класс оканчивается символом «`]`». Класс может содержать диапазоны в форме «`a-f`», которые раскрываются в «`abcdef`».

```
$ hg status 'glob:**[nr-t]'
? MANIFEST.in
? src/xyzzy.txt
```

Если первый символ после «`[`» в классе символов является «`!`», это инвертирует класс, делая его соотвествующим любому символу не из класса.

Символ «`{`» начинает группу подшаблонов, где вся группа соответствует если любой из шаблонов в ней совпадает. Символ «`,`» разделяет подшаблоны, а символ «`}`» закрывает группу.

```
$ hg status 'glob:*.{in,py}'
? MANIFEST.in
? setup.py
```

### 7.4.1.1. Внимание!

Не забывайте что если вы хотите соответствия в каком-либо каталоге вы не должны использовать «`*<любой токен>`», т.к. «`*`» работает только с одной директорией. Вместо него используйте «`**`» (две звездочки). Ниже пример иллюстрирующий различия.

```
$ hg status 'glob:*.py'
? setup.py
$ hg status 'glob:**.py'
A examples/simple.py
A src/main.py
? examples/performant.py
? setup.py
```

```
? src/watcher/watcher.py
```

## 7.4.2. Шаблоны регулярных выражений

Mercurial поддерживает теже регулярные выражения, что и язык программирования Python (используется движок regexp из Python-а). Синтаксис базируется на регулярных выражениях Perl, которые являются наиболее популярных диалектом (например он используется в Java).

Я не стану обсуждать диалект regexp в Mercurial т.к. он не часто используется. В любом случае регулярные выражения Perl хорошо документированы на множестве веб сайтов и в большом количестве книг. Вместо этого я сфокусируюсь на некоторых вещах, которые вы должны знать если хотите использовать регулярные выражения в Mercurial.

Регулярные выражения применяются к полному пути файла, относительно корня репозитория. Другими словами, даже если вы уже в поддиректории `foo` и хотите работать с файлами в поддиректории вы должны начать паттерн с «`foo/`».

Стоит отметить одну вещь: если вы знакомы с регулярными выражениями Perl - регулярные выражения Mercurial применяются с начала строки. Таким образом, regexp ищет совпадения с начала строки и не ищет совпадения где-нибудь внутри строки. Для поиска везде в строке вы должны начинать ваше регулярное выражение с «`.*`».

# 7.5. Фильтрация файлов

Mercurial не только позволяет вам указывать разными способами нужные файлы, он также дает вам возможность отсеивать ненужные файлы. Команды, работающие с именами принимают две опции для фильтрования.

- `-I`, или `--include`, позволяет указать шаблон имен файлов, которые должны быть обработаны.
- `-X`, или `--exclude`, дает возможность указать шаблон имен для исключения из обработки.

Вы можете указывать многократно и ту и другую опции в командной строке и смешивать их так, как вам нужно. Mercurial интерпретирует шаблоны используя glob синтаксис по умолчанию (но вы можете использовать синтаксис regexp).

Вы можете считать опцию `-I` filter как «обрабатывать только те файлы, которые совпадают с этим фильтром».

```
$ hg status -I '*.in'
? MANIFEST.in
```

Опцию `-X` filter проще всего понять как «обрабатывать только те файлы, которые не попадают под фильтр».

```
$ hg status -X '**.py' src
? src/watcher/_watcher.c
? src/xyzzy.txt
```

# 7.6. Постоянное игнорирование ненужных файлов и директорий

Когда вы создаете новое хранилище, то вероятно что со временем оно будет расти содержать файлы, которые не должны управляется Mercurial, но вы же не хотите, чтобы они перечислялись при каждом запуске **hg status**. Например, файлы «сборки продукта», которые создаются как часть сборки, но которые не должны управляться системой контроля версий. Чаще всего файлы продуктов сборки, создаются программными инструментами, такими как компиляторы. В качестве другого примера, множество текстовых редакторов складывают в каталог блокировки файлов, временные рабочие файлы и файлы резервных копии, которые он также не имеет смысла в управлении.

Чтобы Mercurial постоянно игнорировал такие файлы, нужно создать файл с именем `.hgignore` в корневом каталоге вашего репозитория. Вы должны добавить его командой **hg add**, так чтобы он стал отслеживаемым вместе с остальным содержимым репозиторием, так как он вероятно будет полезным сотрудникам.

По умолчанию файл `.hgignore` должен содержать список регулярных выражений, по одному в каждой строке. Пустые строки пропускаются. Большинство людей предпочитают помечать файлы, которые они хотят игнорировать с использованием «glob» синтаксиса, который мы описали выше, такой типичный файл `.hgignore` будет начинаться с директивы:

```
syntax: glob
```

Это говорит Mercurial интерпретировать строки, указанные ниже, с использованием модели glob, а не регулярных выражений.

Вот типичный вид файла `.hgignore`.

```
syntax: glob
# This line is a comment, and will be skipped.
# Empty lines are skipped too.

# Backup files left behind by the Emacs editor.
*~

# Lock files used by the Emacs editor.
# Notice that the "#" character is quoted with a backslash.
# This prevents it from being interpreted as starting a comment.
.\#*

# Temporary files used by the vim editor.
.*.swp

# A hidden file created by the Mac OS X Finder.
.DS_Store
```

# 7.7. Регистрозависимость

Если вы работаете в гетерогенной среде разработки, которая содержит как Linux (или другие Unix) системы так и системы с Mac или Windows, вы должны помнить, что они относятся к регистру («N» и «n» ) имен файлов, по разному. Это вероятно не очень, повлияет на вас, и это легко исправляется, если это произойдет, но это может удивить вас, если вы не знаете об этом.

Операционные и файловые системы отличаются по способам работы с регистром символов в именах файлов и директорий. Существует три основных модели работы с именами.

- Полностью без учета регистра. Буквы в верхнем и нижнем регистрах считаются идентичными как при создании файла, так и при последующей работе с ним. Это поведение типично для файловых системы выросших из DOS.
- Сохранение регистра, без его учета в дальнейшем. Когда создается файл или каталог его имя сохраняется так как было введено. Оно также отображается операционной системой в сохраненном регистре. Когда производится работа с файлом, то регистр не учитывается. Это стандартное поведение в Windows и MaxOS. Например имена `foo` и `FoO` идентичны с этой точки зрения. Подобная обработка прописных и строчных букв как взаимозаменяемых также известна как сворачивание регистра.
- С учетом регистра. В этом случае имя файла существенно всегда. Имена `foo` и `FoO` идентифицируют разные файлы. Это стандартно для Linux и Unix систем и в них это используется по умолчанию.

В Unix-подобных системах возможно иметь любой или даже все вышеупомянутые способы обработки имен сразу. Например, если вы используете USB накопитель с файловой системой FAT32 в Linux, то имена файлов в этой файловой системе будут сохранять регистр, но при работе он не будет учитываться.

## 7.7.1. Безопасное и переносимое хранилище

Система хранилищ Mercurial регистронезависима. Она переводит имена файлов так, что они могут быть безопасно сохранены и на регистрозависимых, и на регистронезависимых файловых системах. Это означает, что вы можете использовать обычное копирование файлов, например, на USB-флешку и спокойно переносить на ней хранилище между компьютерами, использующими MacOS, Windows и Linux.

## 7.7.2. Определение конфликтов регистра символов

Когда операционная система работает в директории, Mercurial соблюдает политику именования принятую в файловой системе где расположена директория. Если файловая система позволяет сохранять регистр, но не учитывает его, Mercurial считает имена, различные по регистру, равнозначными.

Важным аспектом этого поведения является то что возможно сделать commit набора изменений в системе с учетом регистра в файловой системе (в Linux или Unix), которые вызовут проблемы у пользователей с другим типом файловой системы (Windows или MacOS). Если Linux-пользователь создаст файлы `myfile.c` и `MyFile.C`, оба будут успешно сохранены в репозитории и с ними можно будет работать на других Linux-машинах.

Если пользователь Windows или Mac зальет в свой репозиторий подобные изменения, то он не получит каких-то проблем, т.к. механизм хранения в репозитории Mercurial безопасен к регистру символов. В тоже время, как только он попытается сделать **hg update** в рабочем каталоге до этого набора изменений, или **hg merge** с ним, Mercurial определит конфликт имен файлов, которые файловая система трактует как идентичные и запретит обновление или слияние.

## 7.7.3. Исправление конфликта регистра символов

Если вы используете Windows или Mac в среде, где кто-то использует Linux или Unix, и Mercurial сообщает о проблемах с регистром, когда вы пытаетесь сделать **hg update** или **hg merge**, процедура исправления этого очень проста.

Просто найдите ближайшую машину с Linux или Unix, клонируйте на нее репозиторий и используйте команду **hg rename** для изменения имени на какое-то другое, так чтобы конфликта не было. Сохраните свои изменения (**hg commit**), сделайте **hg pull** или **hg push** в ваш Windows или MacOS репозитарий и **hg update** на ревизию без конфликта.

Список изменений с конфликтом регистра останется в истории вашего проекта и вы все еще будете не в состоянии выполнить **hg update** вашей рабочей директории к этому набору изменений в Windows или MacOS, но вы сможете продолжить разработку.

# Глава 8. Управление релизами и ветками

Mercurial предоставляет вам несколько механизмов для управления проектом, которые работают на множестве направлений одновременно. Для понимания этих механизмов сперва кратко рассмотрим довольно распространенные примеры структуры программных проектов.

Многие програмные проекты выпускают переодические «major» релизы, которые содержат некоторые новые существенные возможности. В тоже время они выпускают множество «minor» релизов. Они, как правило, идентичны основным релизам на которых они основаны, но исправляют ошибки.

В этой главе вы поговорим о том как вести учет вех проекта, таких как релизы. Затем мы продолжим разговор о рабочем процессе между двумя разными фазами проекта и увидим как Mercurial может помочь вам изолировать и управлять этим процессом.

## 8.1. Задание постоянного имени для ревизии

Как только вы решаете что хотели бы сделать определенную ревизию релизом — хорошая идея сделать идентифицирующую запись об этом. Это позволит вам позднее получить релиз в любое время (воспроизвести баг, портировать на новую платформу и т.д.).

```
$ hg init mytag
$ cd mytag
$ echo hello > myfile
$ hg commit -A -m 'Initial commit'
adding myfile
```

Mercurial дает вам возможность указать постоянное имя любой ревизии используя команду **hg tag.** Не удивительно что эти имена называют «тегами».

```
$ hg tag v1.0
```

Тег это ничто иное как «символическое имя» для ревизии. Теги существуют просто для вашего удобства, чтобы у вас был простой способ обратиться к ревизии. Mercurial никак не обрабатывает имена тегов, которые вы указываете. Mercurial не устанавливает ограничения на имена тегов, кроме некоторых, чтобы гарантировать что тег может быть проанализирован однозначно. Имя тега не может содержать ни один из следующих символов:

- Двоеточие (ASCII 58, «`:`»)
- Возврат каретки (ASCII 13, «`\r`»)
- Перевод строки (ASCII 10, «`\n`»)

Вы можете использовать **hg tags** для показа тегов в вашем репозитории. В выводе каждая тегированая ревизия идентифицируется сначала по имени, затем по номеру ревизии, и потом по уникальному хешу ревизии.

```
$ hg tags
tip                                1:2f120dd1b81c
v1.0                               0:2765f2245422
```

Обратите внимание, что в выводе **hg tags** показан тег `tip` (окончание ветки). Тег `tip` это специальный «плавающий» тег, который всегда указывает на новейшую ревизию в репозитории.

В выводе команды **hg tags** теги показаны в обратном (относительно номеров ревизий) порядке. Обычно это значит, что более новые теги будут показаны перед более старыми. Также это означает, что `tip` всегда будет показан первым в выводе **hg tags**.

Когда вы запускаете **hg log**, если она показывает номер ревизии, которая имеет ассоциацию с тегами, то печатаются и эти теги:

```
$ hg log
changeset:   1:2f120dd1b81c
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
```

```
date:        Thu Feb 02 14:10:07 2012 +0000
summary:     Added tag v1.0 for changeset 2765f2245422

changeset:   0:2765f2245422
tag:         v1.0
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:07 2012 +0000
summary:     Initial commit
```

Всегда, когда вам необходимо подставить идентификатор ревизии в команду Mercurial, можно подставлять имя соответствующего тега. Внутри себя Mercurial переводит имя тега в идентификатор ревизии.

```
$ echo goodbye > myfile2
$ hg commit -A -m 'Second commit'
adding myfile2
$ hg log -r v1.0
changeset:   0:2765f2245422
tag:         v1.0
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:07 2012 +0000
summary:     Initial commit
```

Для отдельной ревизии или всего репозитория не существует лимита на количество тегов, которые вы можете использовать. На практике это означает что «слишком много тегов» (число которое будет сильно изменяться от проекта к проекту) не очень хорошая идея, просто потому что теги, как предполагается, помогают вам находить ревизии. Если же у вас много тегов, простота их использования очень сильно уменьшается.

К примеру, если ваш проект будет создавать вехи каждые пару дней совершенно разумно помечать их тегами. Но если вы используете систему непрерывной интеграции, которая гарантирует чистоту сборки каждой ревизии, тегирование каждой успешной сборки внесет неясности. Взамен вы можете помечать тегом ревизии, чья сборка завершилась неудачей (при условии что они редки!), или просто не использовать теги для отслеживания процесса сборки.

Если вы хотите удалить более не используемый тег используйте команду **hg tag --remove.**

```
$ hg tag --remove v1.0
$ hg tags
tip                                3:7aa8dea37120
```

Вы также можете изменять тег в любое время, т.о. для идентифицирования разных ревизий выполните новую команду **hg tag.** Вы также можете использовать опцию `-f`, чтобы указать что вы действительно хотите изменить тег.

```
$ hg tag -r 1 v1.1
$ hg tags
tip                                4:e38b5c6c84f1
v1.1                               1:2f120dd1b81c
$ hg tag -r 2 v1.1
abort: tag 'v1.1' already exists (use -f to force)
$ hg tag -f -r 2 v1.1
$ hg tags
tip                                5:c224d1a9e578
v1.1                               2:36034fbfb874
```

В выводе все еще присутствуют старые записи тегов, но Mercurial не будет использовать их в дальнейшем.Таким образом нет никакой беды, если вы установили тег для неверной ревизии — все что нужно изменить неверный тег и скорректировать ревизию как только вы обнаружите ошибку.

Mercurial хранит теги как обычный файл с контролем ревизий в вашем репозитории. Если вы создаете любые теги, вы сможете найти их все в файле `.hgtags`. Когда вы запускаете **hg tag**, Mercurial модифицирует этот файл, а затем автоматически комитит изменения в нем. Это означает что на каждый запуск **hg tag** вы можете найти соответствующий набор изменений в выводе команды **hg log.**

```
$ hg tip
changeset:   5:c224d1a9e578
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:08 2012 +0000
```

```
summary:     Added tag v1.1 for changeset 36034fbfb874
```

### 8.1.1. Обработка конфликтов слияния тегов

Вы не должны особо заботиться о файле `.hgtags`, но иногда он дает о себе знать при слиянии изменений. Формат файла очень прост: он состоит из строк. Каждая строка начинается с хеша набора изменений, за который через пробел следует имя тега.

Если вы исправляете конфликт в файле `.hgtags`, есть одно замечание про его модификацию: когда Mercurial читает теги в репозитории, он никогда не читает рабочую копию файла `.hgtags`. Вместо этого он читает наиболее старшую ревизию этого файла.

Неудачное последствие такого дизайна состоит в том, что вы не можете проверить правильность вашего файла `.hgtags` после слияния до тех пор, пока вы не закомитите изменения. Таким образом, если вы решаете конфликт в `.hgtags` во время слияния убедитесь что вы запустили **hg tags** после слияния. Если эта команда найдет ошибку в файле `.hgtags`, она сообщит о месте ее возникновения. Эта информация поможет вам в исправлении ошибки и повторном комите. Вы должны запустить **hg tags** вновь, как только будете уверены в правильности файла.

### 8.1.2. Теги и клонирование

Возможно вы обратили внимание, что команда **hg clone** имеет опцию `-r`, которая позволяет клонировать точную копию репозитория как частичные наборы изменений. Новый клон не будет содержать истории проекта, которая появилась позже указанной вами ревизии. Это относится и к тегам, что может стать неожиданностью для неосторожных.

Вспомните что тег сохранен как ревизия в файле `.hgtags`, так что когда вы создаете тег, набору изменений в котором он записан нужно обратиться к старшему набору изменений. Когда вы запускаете команду **hg clone -r foo** для клонирования репозитория по имени тега `foo`, новый клон не будет содержать истории создания клона. Результатом этого станет то, что вы получите полное подмножество истории проекта в новом репозитории, а не тег, который вы, возможно, ожидали.

### 8.1.3. Когда тегов становится слишком много

Итак, так как теги в Mercurial подвержены версионности и связаны с историей проекта, любой кто работает с ними может видеть созданые вами теги. Именованая ревизия `4237e45506ee` будет доступна по простому тегу `v2.0.2`. Если же вы пытаетесь отыскать сложноуловимую ошибку, вы возможно захотите чтобы тег напоминал вам нечто вроде «Анна видела признаки ошибки в этой ревизии» (речь тут идет о «пометках на полях для себя», прим. переводчика).

Для подобных целей вы можете захотеть использовать локальные теги. Вы можете создать локальный тег используя опцию `-l` команды **hg tag**. Это позволит записать тег в файл `.hg/localtags`, вместо `.hgtags`. Версия файла `.hg/localtags` не отслеживается. Любые теги созданые с опцией `-l` остаются локальными для вашего репозитория.

## 8.2. Поток изменений — «большая картинка» против «маленькой»

Вернёмся к схеме, которую я нарисовал в начале главы и в то же время подумаем о проекте как о множественных конкурирующих кусках работы под разработкой.

Это может быть толчком для нового «основного» релиза; нового «небольшого» релиза с исправленными багами для последнего «основного» релиза; и неожиданного «горячего» изменения для старого релиза, который сейчас ещё поддерживается.

Обычно люди называют разные конкурирующие направления «ветками». Однако мы уже видели несколько раз, что Mercurial обрабатывает всю историю как серию «веток» и «слияний». На самом дела у нас есть 2 идеи, которые плохо связаны, но имеют одинаковые названия.

- «Большие картинки» веток представляют собой вариацию проекта; люди дают им имена и говорят о них.
- «Маленькие картинки» веток являются артефактами каждодневной разработки и слияния изменений. Они показывают, как код разрабатывался.

## 8.3. Управление ветками «больших картинок» в репозитории (хранилище)

Самый простой путь изолировать «большую картинку» веток в Mercurial это отдельное хранилище. Если у вас уже есть созданное общее хранилище — скажем, с именем `myproject` — которое достигло контрольной точки «1.0», вы можете начинать подготовку для будующих «основных» релизов сверху версии «1.0» добавляя ревизию, где вы готовы для релиза «1.0».

```
$ cd myproject
$ hg tag v1.0
```

Затем вы можете клонировать новое хранилище проекта в `myproject-1.0.1`.

```
$ cd ..
$ hg clone myproject myproject-1.0.1
updating to branch default
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

После этого, если кому-то будет нужно работать для устранения багов к предстоящему `1.0.1` релизу, то можно клонировать хранилище `myproject-1.0.1`, сделать нужные изменения и вернуть их обратно.

```
$ hg clone myproject-1.0.1 my-1.0.1-bugfix
updating to branch default
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd my-1.0.1-bugfix
$ echo 'I fixed a bug using only echo!' >> myfile
$ hg commit -m 'Important fix for 1.0.1'
$ hg push
pushing to /tmp/branch-repookUgeL/myproject-1.0.1
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
```

Тем временем разработчики следующего «большого» релиза могут продолжать работать, изолированные в хранилище `myproject`.

```
$ cd ..
$ hg clone myproject my-feature
updating to branch default
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd my-feature
$ echo 'This sure is an exciting new feature!' > mynewfile
$ hg commit -A -m 'New feature'
adding mynewfile
$ hg push
pushing to /tmp/branch-repookUgeL/myproject
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
```

## 8.4. Не повторяйте сами себя: слияния между «ветками»

Во многих случаях если вы должны исправить баг в «основной» ветке, есть большие шансы, что этот баг есть в ваших остальных ветках. Редкий разработчик хочет исправлять один и тот же баг несколько раз. Давайте

рассмотрим несколько вариантов, когда Mercurial может помочь вам исправить баги без дублирования вашей работы.

В простейшнм случае всё, что вам нужно сделать — это извлечь изменения с «основной» ветки в вашу локальную копию этой ветки.

```
$ cd ..
$ hg clone myproject myproject-merge
updating to branch default
3 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd myproject-merge
$ hg pull ../myproject-1.0.1
pulling from ../myproject-1.0.1
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
```

Затем вам нужно соединить заголовки 2-х веток и вернуться к «основной» ветке.

```
$ hg merge
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ hg commit -m 'Merge bugfix from 1.0.1 branch'
$ hg push
pushing to /tmp/branch-repookUgeL/myproject
searching for changes
adding changesets
adding manifests
adding file changes
added 2 changesets with 1 changes to 1 files
```

# 8.5. Наименование веток в одном репозитории(хранилище)

Во многих случаях изоляция веток в хранилищах — это хорошее решение. Это просто понять; и также сложно сделать ошибку. Это отношения один-к-одному между ветками, с которыми вы работаете и папками(директориями) в вашей системе. Тогда вы можете использовать нормальные(ничего не знающие о Mercurial) программы, чтобы работать с файлами в ветке/хранилище.

Если вы (как и ваши коллеги) «продвинутый пользователь», то вы можете рассматривать другой способ трактовки веток. Я уже упоминал разницу человеческого уровня между «маленькой картинкой» и «большой картинкой» веток. Пока Mercurial всё время работает с несколькими «маленькими картинками» в хранилище (например, когда вы отослали изменения, но ещё не соединили), Mercurial также может работать с несколькими ветками «больших картинок».

Ключ к работе в этом направлении в том, что Mercurial позволяет вам дать постоянное имя ветке. И всегда существует ветка, названная `default` (по-умолчанию). Даже перед тем, как вы переименуете ветку сами, вы можете найти историю ветки `default`, если поищите.

И как пример, когда вы запускаете команду **hg commit**, и вы попадаете в редактор, где вы можете ввести commit, посмотрите на верхнюю линию, которая содержит текст «`HG: branch default`». Это говорит вам о том, что ваш commit случится с веткой, названной `default`.

Чтобы начать работу с именованными ветками используйте команду **hg branches**. Эта команда покажет вам список именованных веток, которые есть в хранилище, расскажет, какая ветка есть изменение другой.

```
$ hg tip
changeset:   0:f9dffe1ca22e
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:09:29 2012 +0000
```

```
summary:     Initial commit

$ hg branches
default                        0:f9dffe1ca22e
```

Пока вы не создали ни одной именованной ветки, существует только одна, названная `default`.

Найти, какая «текущая» ветка сейчас, запустите команду **hg branch** без аргументов. Вы узнаете, какая ветка является «родительской» для «текущей».

```
$ hg branch
default
```

Чтобы создать новую ветку запустите команду **hg branch** снова. В этот раз с аргументом: именем ветки, которую вы создаёте.

```
$ hg branch foo
marked working directory as branch foo
$ hg branch
foo
```

После создания новой ветки вы спросите, что сделала команда **hg branch**? Что показывают команды **hg status** и **hg tip**?

```
$ hg status
$ hg tip
changeset:   0:f9dffe1ca22e
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:09:29 2012 +0000
summary:     Initial commit
```

Ничего не изменилось в рабочей папке и не было создано новой истории. Запуск команды **hg branch** не производит постоянного эффекта; она только говорит Mercurial, ветку с каким именем использовать для последующих фиксаций изменений (commit).

Когда вы подтверждаете изменения, Mercurial записывает имя ветки, которую вы изменяете. Один раз переключив ветку `default` на другую и подтвердив, вы увидите имя новой ветки в результатах **hg log**, **hg tip**, и других комманд, которые показывают эту информацию.

```
$ echo 'hello again' >> myfile
$ hg commit -m 'Second commit'
$ hg tip
changeset:   1:5dc3c3ccfa6d
branch:      foo
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:09:29 2012 +0000
summary:     Second commit
```

Команды как **hg log** напечатают имя ветки для каждого изменения, которое не в ветке `default`. Как результат, если вы никогда не именовали ветки — вы никогда не увидите эту информацию.

Один раз дав имя ветке и подтвердив изменение с этим именем каждое последующее изменение будет иметь то же имя ветки. Вы можете сменить имя ветки в любое время, используя команду **hg branch**.

```
$ hg branch
foo
$ hg branch bar
marked working directory as branch bar
$ echo new file > newfile
$ hg commit -A -m 'Third commit'
adding newfile
$ hg tip
changeset:   2:1536e3edee0d
branch:      bar
tag:         tip
```

```
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:09:30 2012 +0000
summary:     Third commit
```

На практике это то, что вы не должны делать часто, имена веток имеют тенденцию существовать долгое время(это не правило, лишь наблюдение).

# 8.6. Работа с несколькими поименованными ветками в хранилище.

Если у вас болльше чем 1 ветка с именем в хранилище, Mercurial будет помнить ветку вашей рабочей папки когда вы запустите такую команду, как **hg update** или **hg pull -u**. Это обновит рабочую папку с этой веткой. Обновить ветку с другим именем вы можете использовать опцию `-C` с командой **hg update**.

Посмотрим это на практике. Сперва вспомните, какая ветка сейчас «текущая» и какие ветки есть в нашем хранилище.

```
$ hg parents
changeset:   2:1536e3edee0d
branch:      bar
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:09:30 2012 +0000
summary:     Third commit

$ hg branches
bar                            2:1536e3edee0d
foo                            1:5dc3c3ccfa6d (inactive)
default                        0:f9dffe1ca22e (inactive)
```

Мы в `bar` ветке, но так же существует старая ветка **hg foo**.

Мы можем использовать **hg update** назад и вперед между `foo` и `bar` ветками без нужды использовать опцию `-C`, потому-что это всего лишь перемещение по линейной истории наших изменений.

```
$ hg update foo
0 files updated, 0 files merged, 1 files removed, 0 files unresolved
$ hg parents
changeset:   1:5dc3c3ccfa6d
branch:      foo
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:09:29 2012 +0000
summary:     Second commit

$ hg update bar
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ hg parents
changeset:   2:1536e3edee0d
branch:      bar
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:09:30 2012 +0000
summary:     Third commit
```

Если мы вернёмся к ветке `foo` и затем запустим **hg update**, мы останемся в `foo`, не перемещаясь к вершине `bar`.

```
$ hg update foo
0 files updated, 0 files merged, 1 files removed, 0 files unresolved
$ hg update
0 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Внесение нового изменения в ветку `foo` создаст новую голову.

```
$ echo something > somefile
```

```
$ hg commit -A -m 'New file'
adding somefile
$ hg heads
changeset:   3:e22fb67a4dba
branch:      foo
tag:         tip
parent:      1:5dc3c3ccfa6d
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:09:31 2012 +0000
summary:     New file

changeset:   2:1536e3edee0d
branch:      bar
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:09:30 2012 +0000
summary:     Third commit

changeset:   0:f9dffe1ca22e
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:09:29 2012 +0000
summary:     Initial commit
```

# 8.7. Имена веток и слияние

Как вам вероятно уже известно, слияния в Mercurial не симметричны. Давайте представим, что у нашего репозитория две головы: 17 и 23. Тогда если я запущу **hg update** для 17, и затем **hg merge** с 23, Mercurial запишет 17 в качестве первого родителя слияния, и 23 в качестве второго. Тогда как если я запущу **hg update** для 23 и затем **hg merge** с 17, это запишет 23 первым родителем, а 17 — вторым.

Это влияет на то, какое имя ветки выберет Mercurial для результата вашего слияния. После слияния, mercurial сохраняет имя ветки первого родителя, когда вы фиксируете результат слияния. Если имя ветки вашего первого родителя `foo`, и вы слили с `bar`, после слияния веткой по-прежнему будет `foo`.

Это необычно, если хранилище хранит несколько голов, с одним именем ветки. Скажем, я работаю с веткой `foo`, как и вы. Мы сохраняем разные изменения; я забираю ваши изменения; у меня сейчас головы, обе претендующие быть в ветке `foo`. Результатом слияния будет одна голова в ветке `foo`, как вы могли надеяться.

Но если я работаю с веткой `bar`, то результатом слияния bar и `foo` будет ветка `bar`.

```
$ hg branch
bar
$ hg merge foo
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ hg commit -m 'Merge'
$ hg tip
changeset:   4:2240616f6d14
branch:      bar
tag:         tip
parent:      2:1536e3edee0d
parent:      3:e22fb67a4dba
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:09:31 2012 +0000
summary:     Merge
```

Более конкретный пример: если я работаю с веткой `bleeding-edge` и хочу внести в неё последние фиксы из ветки `stable`, то Mercurial выберет имя «правильной» ветки (`bleeding-edge`), когда вытащу и объединю изменения из ветки `stable`.

# 8.8. Именованные ветки — это очень удобно.

Вам не стоит думать об именовании веток только когда несколько «устойчивых» веток находятся в одном хранилище. Это так же очень полезно даже для случая одна-ветка-на-хранилище.

В простом случае, если вы дадите имя каждой ветке, вы будете знать, какой ветке принадлежит фиксация. У вас будет больше контекста, когда вы попытаетесь проследить историю изменений.

Если вы работаете с общедоступным хранилищем, то вы можете задать `pretxnchangegroup` перехватчик, который будет блокировать все приходящие изменения с «неправильным» именем ветки. Это простой, но эффективный способ против случайного слияния изменений с «нестабильной» ветки в «стабильную». Так перехватчик может находится внутри общедоступного `/.hgrc` репозитория.

```
[hooks]
pretxnchangegroup.branch = hg heads --template '{branches} ' | grep mybranch
```

# Глава 9. Поиск и исправление ваших ошибок

Человек может ошибиться, но высококлассная система управления версиями берется исправить последствия. В этой главе мы расскажем о некоторых способах поиска проблем, закравшихся в ваш проект. Mercurial предоставляет высокоэффективные инструменты, позволяющие изолировать источник проблем и должным образом образом обработать.

## 9.1. Удаление локальной истории

### 9.1.1. Случайная фиксация

Не часто, но постоянно у меня возникает проблема, что печатаю быстрее, чем думаю, в результате, зафиксированные изменения либо не законченные, либо просто ошибочные. Стандартный для меня тип не завершенных изменений, в том ,что я создал новый исходник, но забыл добавить (**hg add**) его. «Просто ошибочные» изменения не так часты, но не менее досаждающие.

### 9.1.2. Откат транзакции

В разделе Раздел 4.2.2, «Безопасность работы» я упоминал, что Mercurial рассматривает каждую модификацию хранилища как транзакцию. Каждый раз, когда вы фиксируете изменения или подтягиваете изменения из другого хранилища, Mercurial запоминает, что вы сделали. Вы можете отменить, или откатить, только одно из этих действий с помощью команды **hg rollback**. (Смотрите раздел Раздел 9.1.4, «**hg rollback** бесполезен если изменения уже внесены.» о важном предупреждении о использовании данной команды.)

Приведу пример частой собственной ошибки: фиксация изменений, в которых я создал новый файл, но забыл выполнить **hg add** для него.

```
$ hg status
M a
$ echo b > b
$ hg commit -m 'Add file b'
```

Вывод команды **hg status** после фиксации подтверждает ошибку.

```
$ hg status
? b
$ hg tip
changeset:   1:e74fb1f34241
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:06 2012 +0000
summary:     Add file b

```

Были зафиксированы изменения для файла `a`, но не для нового файла `b`. Если бы я отправил эти изменения в общее с коллегами хранилище, то велик шанс того, что что-то в `a` ссылается на `b`, отсутствующий в репозиториях коллег после того, как они получат мои изменения. В таком случае я мог бы стать объектом некоторого негодования.

Однако, мне повезло — я заметил ошибку прежде, чем отослал изменения. Я воспользовался командой **hg rollback**, и Mercurial убрал последние измения.

```
$ hg rollback
repository tip rolled back to revision 0 (undo commit)
working directory now based on revision 0
$ hg tip
changeset:   0:f50222def92b
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:06 2012 +0000
```

```
summary:     First commit

$ hg status
M a
? b
```

Обратите внимание, что изменение отсутствует в истории хранилища, и репозиторий снова считает, что файл `a` в рабочей директории имеет не фиксированные модификации. Commit и rollback оставили рабочую директорию в том же состоянии, в котором она была перед commit; изменения полностью уничтожены. Теперь я могу безопасно выполнить **hg add** файл `b`, и снова запустить commit.

```
$ hg add b
$ hg commit -m 'Add file b, this time for real'
```

## 9.1.3. Ошибочное вытягивание

Стандартной методикой работы с Mercurial является поддержка разработки отдельных веток проекта в отдельных хранилищах. Ваша комманда разработчиков может содержать один репозиторий для релиза проекта версии «0.9», и другой, для других изменений, войдущих в версию «1.0».

На этом примере, можете представить себе неприятные последствия случайного вытягивания изменений из общего хранилища ветки «1.0» в ваш локальный репозиторий релиза «0.9». В худшем случае, вы заплатите за невнимательность втолкнув те самые изменения в общее «0.9» дерево, запутав остальных разработчиков. (Не волнуйтесь, к этому ужасному сценарию мы вернёмся позже). Впрочем, вероятнее всего, Вы заметите это сразу же, поскольку Mercurial покажет вам URL из которого происходит вытягивание. Или Вы обратите внимание на подозрительно большое число изменений в хранилище.

Для исключения этих, только что втянутых правок, отлично подходит команда **hg rollback**. Mercurial группирует все изменения, внесенные командой **hg pull**, в одну трансакцию, поэтому всё, что вам нужно для исправления ошибки — одна команда **hg rollback**.

## 9.1.4. hg rollback бесполезен если изменения уже внесены.

Ценность команды **hg rollback** падает до нуля, как только вы втолкнете изменения в другое хранилище. Конечно, эта команда исправит ошибку, но только в том хранилище к которому вы её применяете. Поскольку rollback уничтожает историю, нет способа остановить распространение ошибочного изменения между хранилищами.

Если Вы втолкнули изменение в общее хранилище, оно по сути «вырвалось на свободу» и вам придётся исправлять ошибку другим способом. Что будет, если вы втолкнули набор изменений куда-либо, после чего стерли их у себя, а потом втянули то что вы только что проталкивали? Набор изменений, от которого (по вашему мнению) вы избавились, опять появится в Вашем хранилище.

(Если вы абсолютно точно знаете, что те изменения, которые вы собираетесь откатить, самые свежие в том самом хранилище, и вы знаете, что никто еще не успел их вытянуть из этого хранилища, то вы можете стереть их и там тоже. Но вам действительно не стоит ожидать что это будет работать надежно. Рано или поздно, изменения наконец таки попадут в неподконтрольное вам хранилище и вернутся назад, чтобы укусить вас.)

## 9.1.5. Вы можете отменить только последнее изменение

Mercurial хранит в своем логе только одну транзакцию — ту, что была выполнена последней в данном хранилище. Это значит, что вы можете отменить только одну транзакцию. Если вы попытаетесь отменить еще одну, предшествующую последней, то вам не удастся этого сделать.

```
$ hg rollback
repository tip rolled back to revision 0 (undo commit)
working directory now based on revision 0
$ hg rollback
no rollback information available
```

Если вы произведете отмену транзакции, вы не сможете снова выполнять roll back до тех пор, пока не произведете другие операции commit или pull.

# 9.2. Отмена ошибочных изменений

Если вы изменили файл, но потом решили, что вообще не хотите его изменять, и если вы ещё не зафиксировали изменения, то команда **hg revert** — то, что вам нужно. Она смотрит на предыдущую ревизию рабочей директории и восстанавливает содержимое файла к состоянию из этой ревизии. (Это многоречивый способ сказать, что обычно эта команда отменяет ваши изменения.)

Давайте посмотрим, как работает команда **hg revert.** В этом маленьком примере мы начнем с изменения файла, о котором Mercurial уже знает.

```
$ cat file
original content
$ echo unwanted change >> file
$ hg diff file
diff -r af8f1f5e8d54 file
--- a/file Thu Feb 02 14:09:48 2012 +0000
+++ b/file Thu Feb 02 14:09:48 2012 +0000
@@ -1,1 +1,2 @@
 original content
+unwanted change
```

Нам не нужно это изменение, поэтому мы вызываем **hg revert** для этого файла.

```
$ hg status
M file
$ hg revert file
$ cat file
original content
```

Команда **hg revert** дополнительно заботится о безопасности данных, сохраняя наш измененный файл с расширением `.orig`.

```
$ hg status
? file.orig
$ cat file.orig
original content
unwanted change
```

### Будьте осторожны с `.orig` файлами

Крайне нежелательно, чтобы вы управляли `.orig`-файлами с помощью Mercurial, и даже чтобы вы задумывались о содержании этих файлов. На всякий случай, полезно помнить, что **hg revert** безоговорочно переписывает существующий `.orig`-файл. Например, если у вас уже есть файл с именем `foo.orig` когда вы откатываете `foo`, содержимое `foo.orig` будет перезаписано.

Ниже приведена сводка случаев, в которых может быть полезна команда **hg revert.** В следующей секции будет детальное описание каждого из них.

- Если вы изменяете файл, она восстановит его до немодифицированного состояния.
- Если вы используете **hg add**, она отменит «добавленное» состояние файла, но оставит сам файл неизменным.
- Если вы удаляете файл не сказав об этом Mercurial-у, она восстановит файл с его предыдущим содержанием.
- Если вы удаляете файл командой **hg remove**, она отменит «удаленное» состояние файла, и восстановит его немодифицированное содержимое.

## 9.2.1. Ошибки управления файлами

Команда **hg revert** полезна не только для только что измененных файлов. Она позволяет отменить результат любой команды управления файлами — **hg add**, **hg remove** и др.

Если вы использовали **hg add** для файла, но потом понимаете, что не хотите, чтобы Mercurial отслеживал его, используйте **hg revert** для отмены добавления. Не волнуйтесь: Mercurial не изменит файл ни в коем случае. Mercurial просто снимет пометку добавления файла.

```
$ echo oops > oops
$ hg add oops
$ hg status oops
A oops
$ hg revert oops
$ hg status
? oops
```

Аналогично, если Вы применили к файлу **hg remove**, то можете с помощью **hg revert** восстановить его с предыдущим содержимым.

```
$ hg remove file
$ hg status
R file
$ hg revert file
$ hg status
$ ls file
file
```

Это также сработает если файл был удалён «в ручную», без помощи Mercurial (Напомним, что в терминологии Mercurial такие файлы называются «missing»).

```
$ rm file
$ hg status
! file
$ hg revert file
$ ls file
file
```

Если вы откатываете операцию **hg copy**, целевой (скопированный) файл все равно остается в вашей рабочей директории, однако изменения в нем не отслеживаются. Поскольку операция копирования в любом случае не затрагивает исходный файл, Mercurial никак не изменяет исходный файл.

```
$ hg copy file new-file
$ hg revert new-file
$ hg status
? new-file
```

# 9.3. Работа с зафиксированными изменениями

Рассмотрим случай, когда вы зафиксировали изменение **a** и другое изменение **b** поверх него. Затем вы обнаружили, что изменение **a** было некорректным. Mercurial позволяет вам автоматически «вернуть» (back out) изменение целиком, и создаёт блоки, которые позволяют вам отменить часть изменения вручную.

Перед прочтением этой части руководства, вы должны четко представлять себе следующее: команда **hg backout** отменяет изменения добавляя новые записи к истории, но ни в коем случае не редактируя и не удаляя уже существующую в истории информацию. Эта утилита хорошо подходит для исправления небольших багов, но не для отмены больших изменений, приведших к серьезным проблемам. Чтобы разобраться с такими проблемами, смотрите секцию Раздел 9.4, «Изменения, которых быть не должно».

## 9.3.1. Отзыв набора измений

Команда **hg backout** позволяет вам автоматически «отменить» всю ревизию. Т.к. Меркуриал не позволяет изменять уже существующую историю, а только лишь добавлять в неё новые записи, данная команда не может просто удалить ревизию, которую вы хотите отменить. Вместо этого она создает новую ревизию, которая отражает состояние репозитория, если бы в него не была добавлена удаляемая ревизия.

Действия выполняемые командой **hg backout** на первый взгляд могут показаться несколько запутанными, поэтому продемонстрируем их на примере. Для начала создадим репозиторий с несколькими простыми изменениями.

```
$ hg init myrepo
$ cd myrepo
$ echo first change >> myfile
$ hg add myfile
$ hg commit -m 'first change'
$ echo second change >> myfile
```

```
$ hg commit -m 'second change'
```

В качестве единственного параметра команда **hg backout** принимает ID удаляемой ревизии. Обычно **hg backout** перебрасывает вас в текстовый редактор, где можно создать комментарий, объясняющий причину отмены изменений. В этом же примере мы добавили комментарий прямо в командной строке при помощи параметра `-m`.

## 9.3.2. Отзыв последней ревизии (tip)

Начнем с отзыва последней ревизии.

```
$ hg backout -m 'back out second change' tip
reverting myfile
changeset 2:9c57b44505fe backs out changeset 1:37eb97592c2d
$ cat myfile
first change
```

Как видите, в `myfile` второй строки уже нет. Взгляд на вывод команды **hg log** позволяет понять, что сделала **hg backout**.

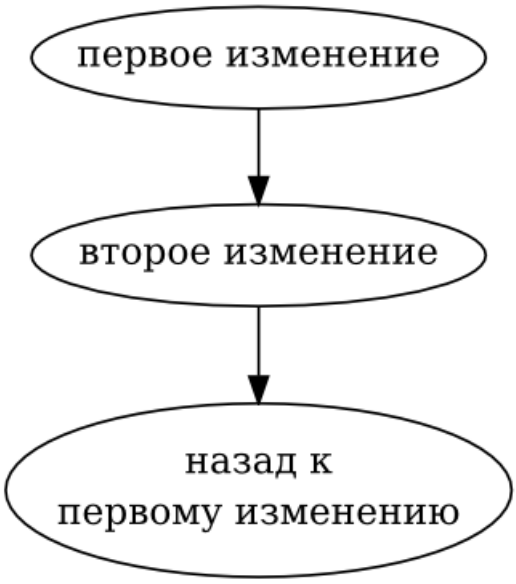
```
$ hg log --style compact
2[tip]   9c57b44505fe   2012-02-02 14:09 +0000   bos
  back out second change

1   37eb97592c2d   2012-02-02 14:09 +0000   bos
  second change

0   b48ef65237ab   2012-02-02 14:09 +0000   bos
  first change
```

Заметим, что новый набор изменений, который был создан командой **hg backout** является потомком отозванного набора. Это легко увидеть на рисунке Рисунок 9.1, «Отмена изменения используя команду **hg backout**», на котором показана история изменений в графическом виде. Как вы можете видеть, история изящная и прямолинейная.

Рисунок **9.1.** Отмена изменения используя команду **hg backout**




## 9.3.3. Отзыв ревизии, не являющейся последней

Если Вы хотите отозвать не последний набор изменений добавляйте к **hg backout** опцию `--merge`.

```
$ cd ..
$ hg clone -r1 myrepo non-tip-repo
adding changesets
```

```
adding manifests
adding file changes
added 2 changesets with 2 changes to 1 files
updating to branch default
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd non-tip-repo
```

Это позволяет делать отзыв любой фиксации за одно действие, что обычно просто и быстро.

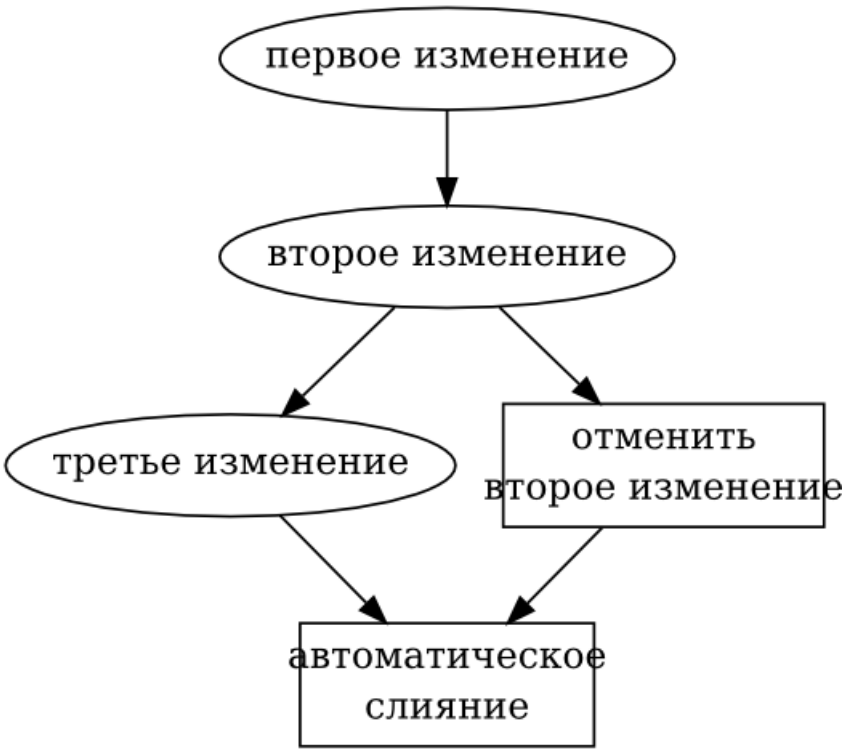
```
$ echo third change >> myfile
$ hg commit -m 'third change'
$ hg backout --merge -m 'back out second change' 1
reverting myfile
created new head
changeset 3:5d235a7062b8 backs out changeset 1:37eb97592c2d
merging with changeset 3:5d235a7062b8
merging myfile
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
```

Если Вы посмотрите на содержимое `myfile` после завершения отзЫва, то увидите, что присутствует первый и третий набор изменений. А второй отсутствует. (Примечание переводчика: пример несколько некорректен, т.к. получается конфликтное слияние, о чем уже написано в комментариях в оригинальном руководстве.)

```
$ cat myfile
first change
third change
```

На графическом представлении истории (рисунок Рисунок 9.2, «Автоматический возврат не последнего набора изменений с помощью команды **hg backout**» ) видим, что в этой ситуации, Mercurial так же фиксирует только одно изменение (прямоугольники — это узлы, которые Mercurial коммитит автоматически), но граф ревизий выглядит иначе. Перед тем, как Mercurial начнёт процесс отзыва, он запомнит текущего родителя рабочей директории. Затем он отзывает указанную ревизию и фиксирует этот отзыв. И наконец, он выполняет слияние с предыдущим родителем рабочей директории, но заметьте, что результат слияния не фиксируется**.** Репозиторий в настоящее время содержит 2 головы, и рабочий каталог в состоянии слияния.

Рисунок **9.2.** Автоматический возврат не последнего набора изменений с помощью команды **hg backout**



В результате вы оказываетесь «там же, где и были» лишь с небольшим увеличением истории, в которой отражён откат.

У вас может возникнуть вопрос, почему Mercurial не фиксирует результат слияния, которое он выполнил. Причина в осторожном поведении Mercurial: очевидно, что слияние имеет значительно большее поле для возможных ошибок, чем просто отмена эффекта последней ревизии, так что ваша работа будет безопаснее, если вы сперва проверите (и протестируете!) результат слияния, и только потом его зафиксируете.

#### 9.3.3.1. Всегда используйте опцию `--merge`

Фактически, поскольку опция `--merge` делает то что надо, независимо от того, является ли отменяемый набор изменений вершиной, или нет (т.е. не пытается зафиксировать возврат набора изменений, являющегося вершиной, поскольку в этом нет необходимости), вам следует всегда использовать эту опцию команды **hg backout**.

### 9.3.4. Получение бОльшего контроля над процессом возврата

До сих пор я рекомендовал вам всегда использовать опцию `--merge`, когда вы возвращаете изменение, однако команда **hg backout** позволяет вам выбрать, каким образом произвести фиксацию возврата изменения. Получение полного контроля над на тем, что происходит в процессе возврата — это то, что вам понадобится достаточно редко, однако полезно понимать, что именно команда **hg backout** делает в автоматическом режиме. Чтобы проиллюстрировать этот процесс, создадим копию репозитория, исключив возвращенные изменения, которые он содержит.

```
$ cd ..
$ hg clone -r1 myrepo newrepo
adding changesets
adding manifests
adding file changes
added 2 changesets with 2 changes to 1 files
updating to branch default
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd newrepo
```

Как и в предыдущем примере, Мы создадим третью ревизию, затем вернём в изначальное состояние, и посмотри что произошло.

```
$ echo third change >> myfile
$ hg commit -m 'third change'
$ hg backout -m 'back out second change' 1
reverting myfile
merging myfile
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
```

Наш новый набор изменений опять является потомком набора изменений, который мы возвращаем, а не того, который являлся вершиной. Команда **hg backout** вполне явно сообщает нам об этом.

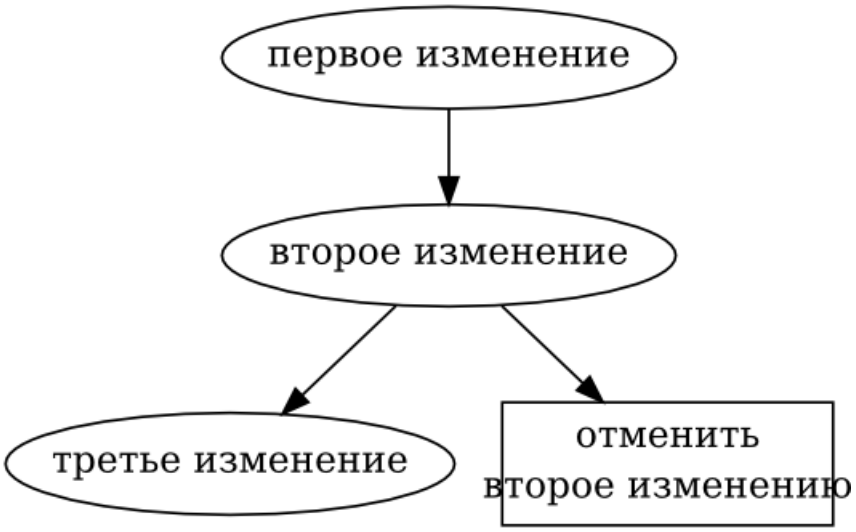
```
$ hg log --style compact
2[tip]   54d16b160c16   2012-02-02 14:09 +0000   bos
  third change

1   37eb97592c2d   2012-02-02 14:09 +0000   bos
  second change

0   b48ef65237ab   2012-02-02 14:09 +0000   bos
  first change
```

И снова чтобы увидеть что произошло, проще взглянуть на граф истории ревизий на рисунке Рисунок 9.3, «Возврат изменений с помощью команды **hg backout**». Он явно показывает, что когда мы использовали **hg backout** для возврата изменений, не являющихся вершиной, Меркуриал добавляет новый head в репозиторий (изменение, которое было зафиксировано, обозначено квадратом).

Рисунок **9.3.** Возврат изменений с помощью команды **hg backout**




После выполнения команды **hg backout**, новый «backout» набор изменений становится родителем рабочего каталога.

```
$ hg parents
changeset:   2:54d16b160c16
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:09:22 2012 +0000
summary:     third change
```

Теперь у нас есть два отдельных набора изменений.

```
$ hg heads
changeset:   2:54d16b160c16
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:09:22 2012 +0000
summary:     third change
```

Давайте подумаем о том, что мы ожидаем увидеть, в содержимом `myfile` сейчас. Первое изменение, должно присутствовать, потому что мы никогда не возвращали его. Второе изменение должно пропасть, как изменение, которое мы вернули. Поскольку граф истории показывает третье изменение в качестве отдельной головы, мы не ожидаем увидеть третье изменение в `myfile`.

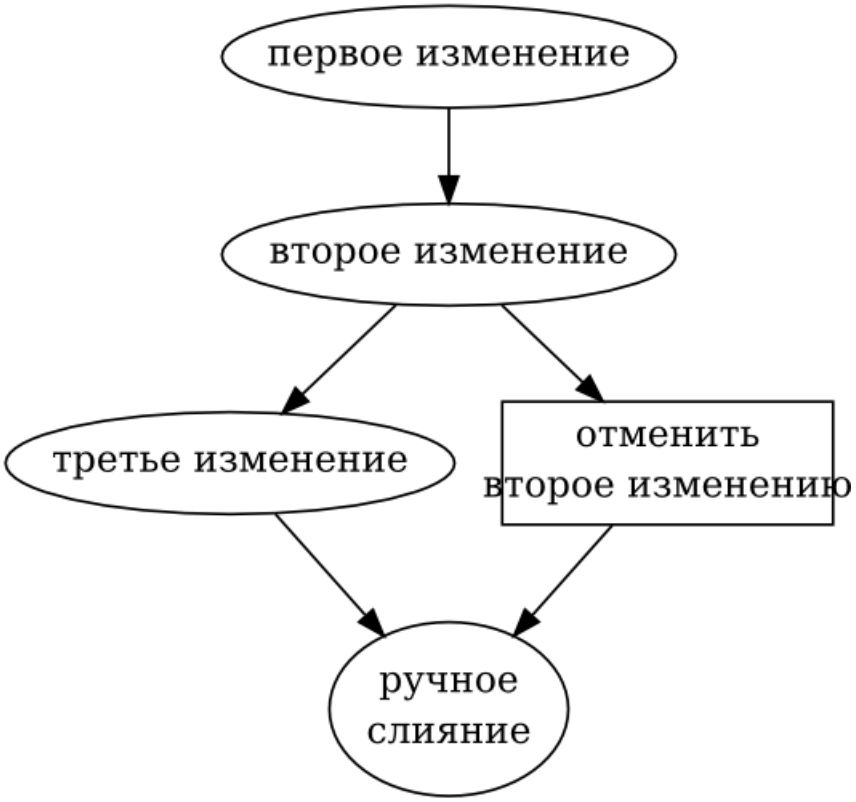
```
$ cat myfile
first change
```

Чтобы получить третье изменение обратно в файл, мы просто делаем нормальное слияние двух наших голов.

```
$ hg merge
abort: there is nothing to merge
$ hg commit -m 'merged backout with previous tip'
$ cat myfile
first change
```

После этого графическая история нашего хранилища показана на Рисунок 9.4, «Ручное слияние возвращённых изменений».

Рисунок **9.4.** Ручное слияние возвращённых изменений




## 9.3.5. Почему команда hg backout работает именно так

Вот краткое описание как работает команда **hg backout**.

1. Она гарантирует, что рабочий каталог будет «чистым», т.е., вывод команды **hg status** будет пустым.
2. Она запоминает текущего родителя рабочего каталога. Назовем эту ревизию оригинальной
3. Она не эквивалентна команде **hg update**, синхронизирующей текущий каталог с ревизией к которой вы хотите вернуться. Назовем эту ревизию откатываемой.
4. Она находит родителя ревизии, которую Вы хотите откатить. Назовем ее ревизия-родитель.
5. Для всех файлов, измененных в откатываемой ревизии, делается эквивалент команды **hg revert -r parent**, для восстановления их до содержимого которое они имели перед этой ревизией.
6. Результат сохраняется как новая ревизия. Эта ревизия имеет родителем откатываемую ревизию.
7. Если Вы указали ключ `--merge` в коммандной строке, делается слияние с оригинальной ревизией и результат слияния сохраняется как новая ревизия

Альтернативный способ реализации **hg backout** команда **hg export** применённую к откатываемому набору изменений как diff-у, а затем использовать опцию `--reverse` для команды **patch** для ликвидации последствий изменения без возни с рабочей директорией. Это звучит гораздо проще, но работает это не так хорошо.

Причина, по которой **hg backout** делает обновление, фиксацию, слияние, и ещё одну фиксацию используя механизм слияния в том, что это лучший шанс сделать работу хорошо, когда речь идет об сохранении всех изменения между ревизией которую мы откатываем, и текущей ревизией.

Если вы отказываясь ревизии, которая была 100 ревизий назад в истории вашего проекта, вероятность того, что команда **patch** сможет применить обратный diff аккуратно не велика, потому что вмешательство последующих

ревизий возможно «нарушило контекст», который использует **patch**, чтобы определить, может ли она применить патч (если это звучит для вас как бред, смотрите раздел Раздел 12.4, «Понимание патчей» для обсуждения строк команды **patch**). Кроме того, механизм слияние Mercurial будет работать с файлами и каталогами, который был переименованы, изменениями прав, и изменения в двоичных файлах, ни с одним из этих вариантов **patch** не может работать.

# 9.4. Изменения, которых быть не должно

Большую часть времени, команда **hg backout** именно то, что вам нужно, если вы хотите устранить последствия изменений. Она оставляет постоянную запись именно того, что вы сделали, как при фиксации оригинального набора изменений, так и после очистки.

В редких случаях, однако, вы можете обнаружить, что вы зафиксировали изменения, которые в действительности не должны присутствовать в общем репозитории. Например, было бы очень необычно, и обычно считается ошибкой, фиксация объектных файлов приложения, также как его исходных файлов. Объектные файлы не имеют почти никакой ценностью, и они большие, так что они увеличивают размер хранилища и количество времени, необходимое для клонирования или вытягивания изменении.

Прежде чем обсудить варианты, которые у вас есть, если вы зафиксировали «грязный пакет» изменений (выглядящий так плохо, что вы хотите выкинуть грязный пакет из головой), позвольте мы сначала обсудим некоторые подходы, которые, вероятно, не будут работать.

Mercurial рассматривает историю как накопление — каждое изменение создаётся поверх всех изменений, которые ей предшествовали — в общем вы не можете просто убрать внесённые изменения. Исключением является тот случай, когда вы только что совершили изменения, и не передали или вытянули их в другой репозиторий. Вот тогда можно смело использовать команду **hg rollback**, как я уже подробно описаны в разделе Раздел 9.1.2, «Откат транзакции».

После того как вы передали плохие изменения в другой репозиторий, вы все равно сможете использовать **hg rollback** в локальной копии изменения исчезают, но это не будет иметь последствий которых вы хотели. Изменения будут присутствовать в удаленном хранилище, поэтому они снова появится в вашем хранилище в следующий раз вы будете забирать из него изменения.

Если возникает такая ситуация, и вы знаете, в какие репозитории распространили свои плохие изменения, можно попробовать избавиться от изменении в каждом из этих репозиториев. Это, конечно, не является удовлетворительным решением: если вы пропустите один из репозиториев пока вы стираете, изменения находящиеся в «диком» мире, и может распространяться дальше.

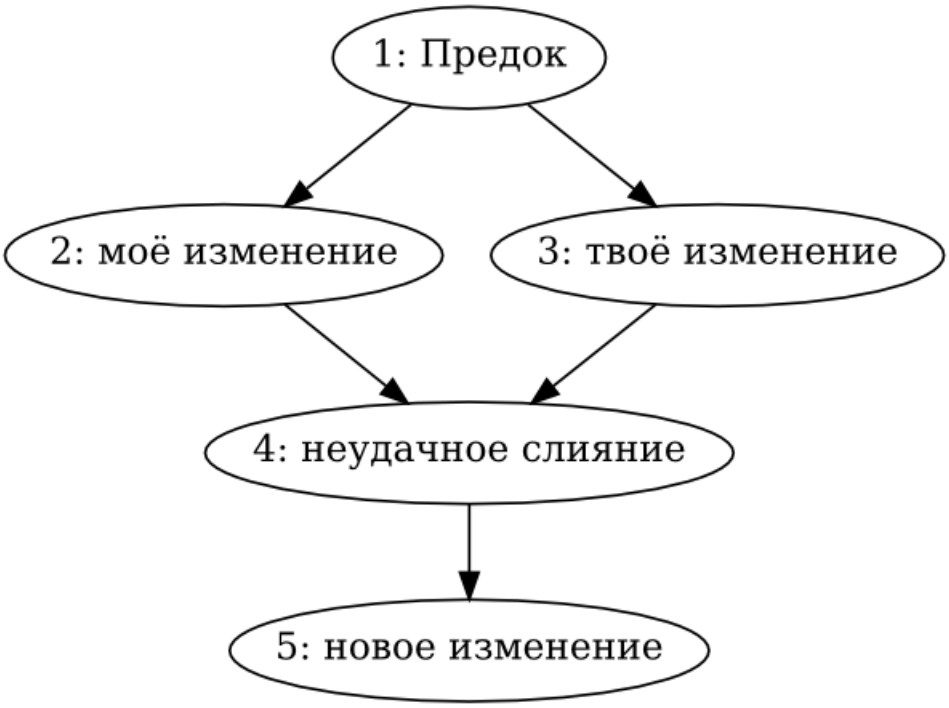
Если вы зафиксировали одно или несколько изменений после изменений, которые вы хотели бы удалить, ваши ревизии в будущем будут раздроблены. Mercurial не предоставляет способа «удалить дыру» в истории, оставив нетронутыми ревизии.

## 9.4.1. Откат слияния

После слияния зачастую сложно, это не неслыханно для слияния чтоб оно было плохим, но можно ошибочно зафиксировать его. Mercurial предоставляет гарантии против плохих слияний, отказываясь фиксировать неразрешенные файлы, но и человеческая изобретательность безгранична, и беспорядок все еще возможен беспорядок при слиянии и его фиксации.

Учитывая плохие слияния, которые были зафиксированы, как правило, лучший способ исправить его — просто попытаться исправить ущерб вручную. Полная катастрофа, которая не может быть легко исправлена своими руками должен происходить очень редко, но команда **hg backout** может помочь в упрощении очистки. Он предлагает — опцию `--parent`, которая позволяет определить, к какому из родителей возвращаться при откате слияния.

Рисунок **9.5.** Плохое слияние

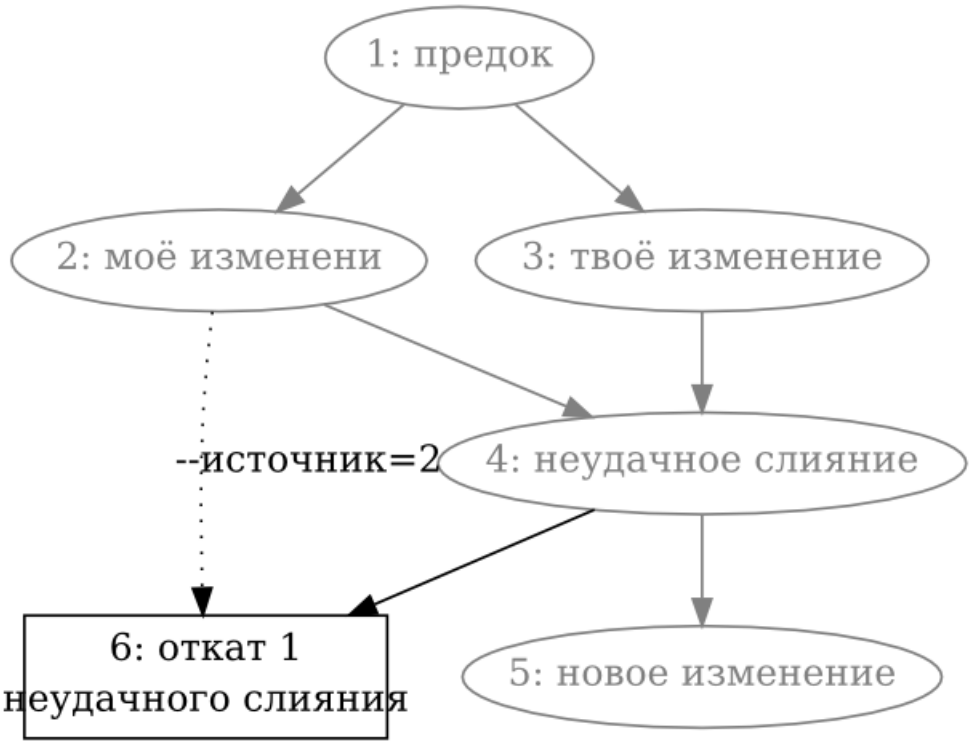



Предположим, мы имеем граф ревизий, как на рисунке Рисунок 9.5, «Плохое слияние». Мы бы хотели повторить это слияние ревизий 2 и 3.

Один из способов сделать это будет выглядеть следующим образом.

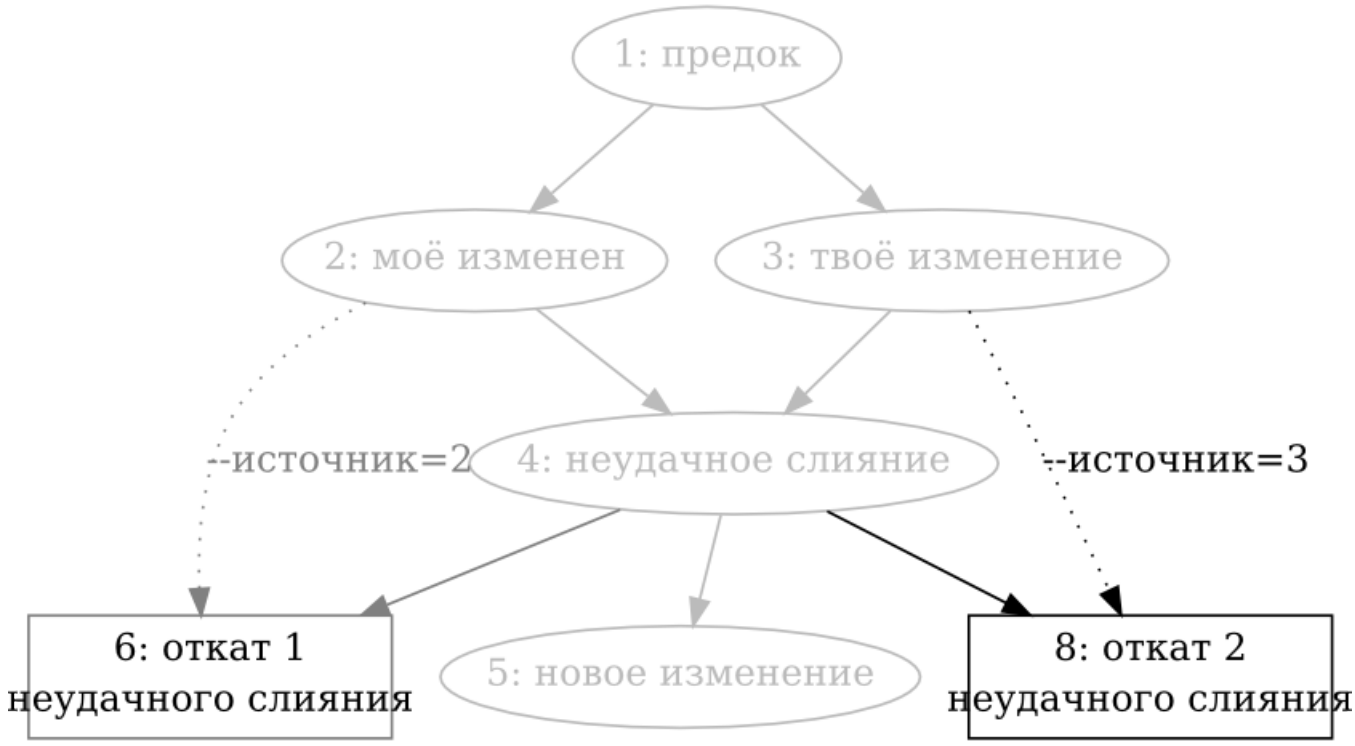
1. Вызываем **hg backout --rev=4 --parent=2**. Это говорит **hg backout** откатить ревизию 4, которая является плохим слиянием, а также при определяет какая ревизия выбрать родительской 2, одну из родителей слияния. Эффект может быть показан на рисунке Рисунок 9.6, «Откат слияния, в пользу одного из родителей».

Рисунок **9.6.** Откат слияния, в пользу одного из родителей



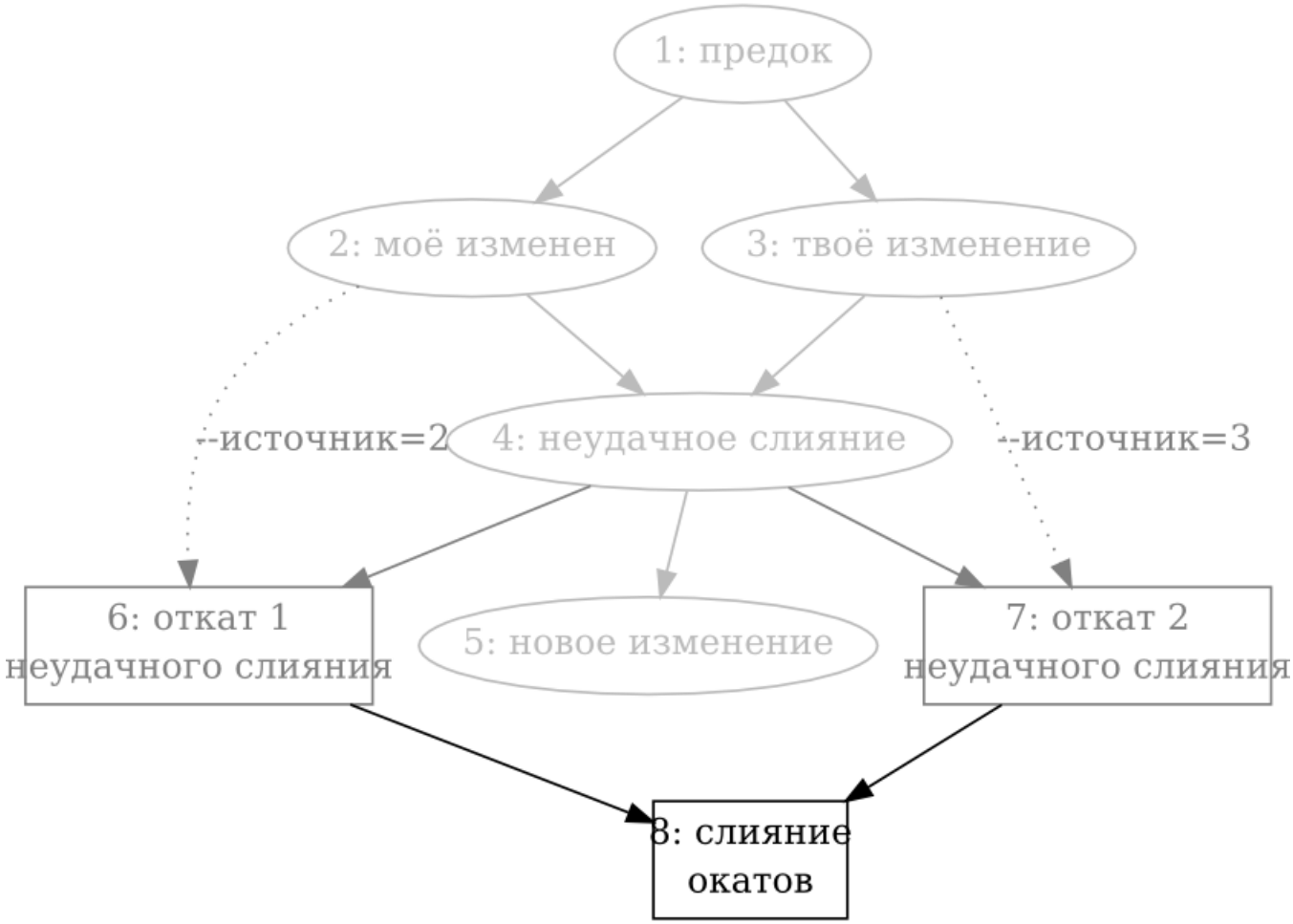

2. Вызываем **hg backout --rev=4 --parent=3**. Это говорит **hg backout** откатить ревизию 4, но на этот раз выбираем родителя 3, другого родителя слияния. В результате показано на рисунке Рисунок 9.7, «Поддержка отката слияния в пользу другого родителя», на котором в это время репозиторий содержит три головы.

Рисунок **9.7.** Поддержка отката слияния в пользу другого родителя



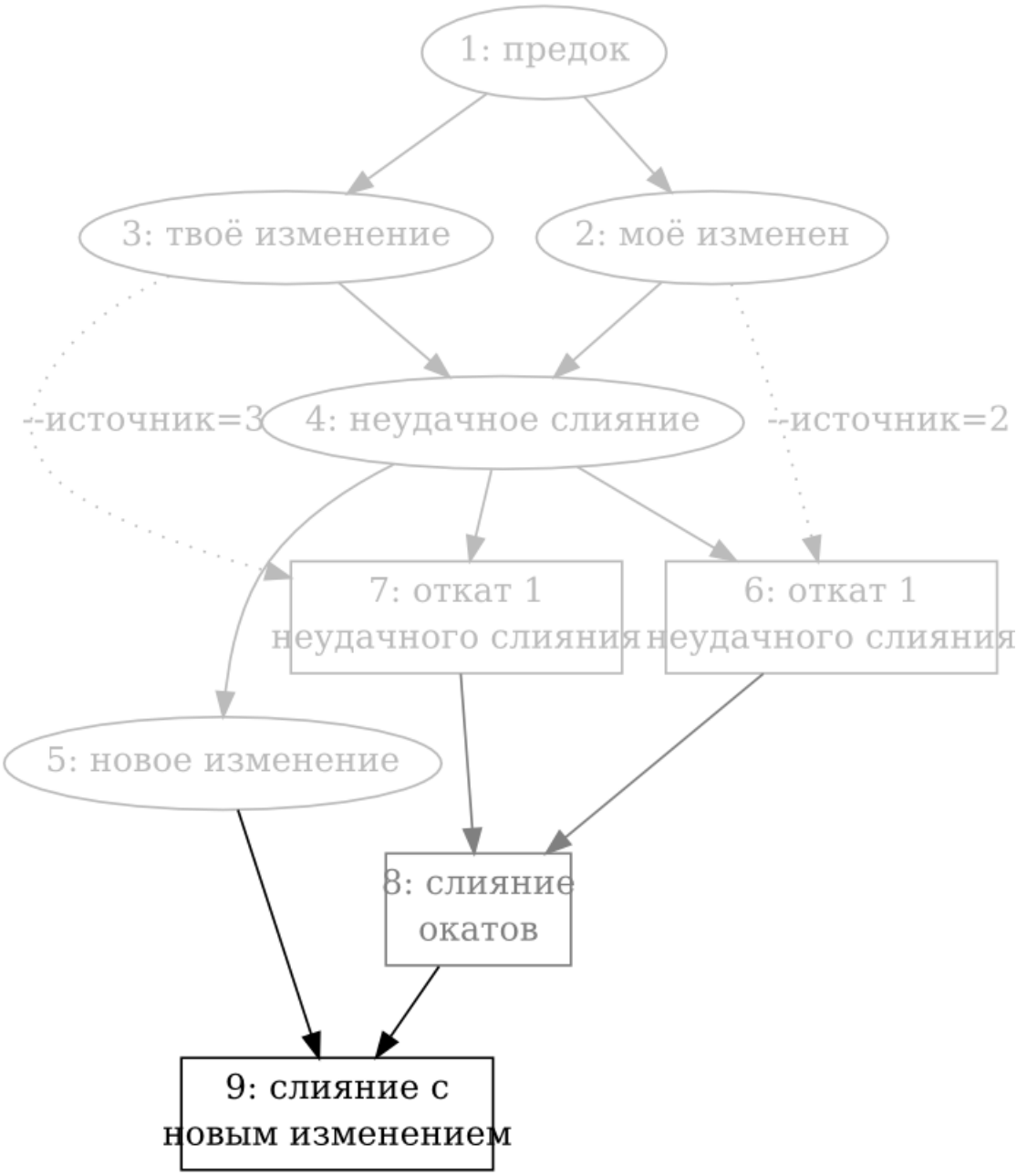
3. Повторяете плохое слияние путем объединения двух отозванных голов, что снижает количество голов в хранилище до двух, как это видно на рисунке Рисунок 9.8, «Слияние откатов».

Рисунок **9.8.** Слияние откатов




4. Слияние с фиксацией, того что было сделано после плохого слияния, как показано на рисунке Рисунок 9.9, «Слияние откатов».

Рисунок **9.9.** Слияние откатов




## 9.4.2. Защитите себя от «беглых» изменении

Если ты совершил некоторые изменения в локальном репозитории, и они были отправлены или вытянуты куда-то, это не всегда катастрофа. Вы можете защитить себя заранее в отношении некоторых классов плохих изменений. Это особенно удобно, если ваша команда обычно вытягивает изменения из центрального хранилища.

При настройке некоторых хуков в репозитории для проверки входящих ревизий (см. Глава 10, *Обработка событий в репозитории с помощью ловушек*), вы можете автоматически предотвращать некоторые виды плохих ревизий вынуждая всех отправлять изменения в центральное хранилище. При такой конфигурации на месте, некоторые виды плохих ревизий естественно, как правило, «отмирают», поскольку они не могут появится в центральном хранилище. А самое хорошее то, что это происходит без необходимости явного вмешательства.

Например, хук на входящие изменения проверяющий, что набор изменений может быть скомпилирован, может помешать людям нечаянно «Нарушить сборку».

### 9.4.3. Что делать с чувствительными изменениями, как избежать

Даже тщательно запущенный проект может пострадать от печального события, такого как фиксация и неконтролируемое распространение файла, который содержит важные пароли.

Если что-то подобное произойдет с вами, и информация, которая случайно распространяется, действительно чувствительна, ваш первый шаг должен быть для смягчения последствий утечки, не пытаясь контролировать утечку самостоятельно. Если вы еще не 100% уверены, что вы знаете точно, кто мог увидеть изменения, вы должны немедленно сменить пароли, отменить кредитную карту, или найти другой способ, чтобы убедиться, что информация, с которой произошла утечка, уже бесполезна. Иными словами, предполагайте, что изменение распространится далеко и широко, и что ничего вы не можете сделать.

Можно надеяться, что нет механизмов, которые можно использовать для выяснения того, кто видел изменения и удалить изменения сразу во всем мире, но есть веские причины, почему это не представляется возможным.

Mercurial не предоставляет аудита, кто вытянул изменения из репозитория, потому что, как правило, либо возможность записи такой информации невозможна или тривиально обманывается. В многопользовательской или сетевой среде, вы должны, таким образом, весьма скептически относится к себе, если вы думаете, что вы определили все места, чувствительные ревизии могут распространится. Не забывайте, что люди могут и будут отправлять разветвления по электронной почте, имеют программное обеспечение резервного копирования данных вне офиса, переносят репозиторий на usb-брелках, и найти другие пути совершенно невинные, чтобы посрамить ваши попытки отследить каждую копию проблемных ревизий.

Mercurial также не дает способ сделать файл или набор изменений которые навсегда исчезают из истории, потому что нет возможности реализовать в своей копии удаление, кто-то может легко изменить свою копию Mercurial игнорировав такие директивы. Кроме того, даже если Mercurial обеспечит возможности, запретить вытягивание файла «отмеченного как удалённый» ревизии не будут затронуты ею, равно как и сканерам, сохраняющими образ жесткого диска или других механизмов. В самом деле, не распределенная система контроля версий может сделать чтобы данные надежно исчезли. Предоставляя иллюзию такого контроля, и могут легко дать ложное чувство безопасности, и это хуже, чем не предоставлять его вовсе.

## 9.5. Поиск источника ошибки

Конечно, возможность откатить изменение, которое внесло ошибку, это хорошо, но сначала нужно узнать, какой именно набор изменений, нужно откатить. Mercurial предоставляет неоценимую команду **hg bisect**, которая поможет вам эффективно автоматизировать этот процесс.

Идея **hg bisect** в том, что ревизии вводили некоторые изменения в поведении, которые можно определить с помощью некоторых простых бинарных испытаний. Вы не знаете, какая часть кода внесла изменения, но вы знаете, как проверить его на наличие ошибок. Команда **hg bisect** использует ваш тест для прямого поиска ревизии, которая содержит код, вызывающий ошибку.

Вот несколько сценариев, которые помогут вам понять, как можно применить эту команду.

- Самая последняя версия программного обеспечения имеет ошибку, и Вы помните, что ее не было несколько недель назад, но не знаете, когда она появилась. Тут-то, ваш бинарный тест проверяет [ревизии] на наличие этой ошибки.
- Вы исправили ошибку в спешке, и теперь пришло время закрыть запись об ошибке в багтрекере вашей команды. Багтрекер данных требует ID ревизии, когда вы закрываете записи, но вы не помните, в какой ревизии Вы исправили ошибку. Снова, ваш бинарный тест проверяет на наличие ошибки.
- Ваше программное обеспечение работает правильно, но работает на 15% медленнее, чем в прошлый раз. Вы хотите знать, какие ревизии внесли уменьшение производительности. В этом случае ваш бинарный тест измеряет производительность вашего программного обеспечения, чтобы показать, что «быстрее» или «медленнее».

- Размеры компонент проекта, который Вы ведете внезапно раздулись, и вы подозреваете, что что-то изменилось во время построения проекта.

Из этих примеров должно быть ясно, что команда **hg bisect** полезна не только для нахождения источников ошибок. С ее помощью можно найти любое «непредвиденное свойство» кодов. То, которое не удаётся найти простым текстовым поиском, но можно отловить бинарным тестом.

Введем немного терминологии, просто чтобы понять, какая часть процесса поиска ложится на Вас и какая на Mercurial. Тест это то, что выбирает ревизию, когда вы запускаете **hg bisect**. Проверка — это то, что **hg bisect** запускает чтобы сказать, какая ревизия хороша. Наконец, мы будем использовать слово «bisect», и как существительное (бисектриса) и глагол (делить пополам), во фразе «поиск с использованием команды **hg bisect**».

Прямолинейный способ автоматизации процесса поиска — просто проверять каждый Changeset. Однако масштабы этого ненормальны. Если тестирование одной ревизии занимает десять минут то полный просмотр 10000 ревизий в вашем хранилище, потребует в среднем 35 дней. Даже если бы вы знали, что ошибка была внесена в одной из последних 500 ревизий и ограничили поиск этим, вы по-прежнему будете искать более чем 40 часов, чтобы найти эту ревизию.

**hg bisect** использует свои знания «формы» истории вашего проекта и чтобы выполнить поиск по времени пропорционально логарифму числа проверяемых ревизий (вид выполняемого поиска называется дихотомический поиск). При таком подходе, поиск по 10000 ревизий займет менее трех часов, даже при десяти минутах на одно испытание (поиск потребует около 14 проверок). Ограничьте поиск последними ста ревизиями, и это займет всего около часа (примерно семь тестов).

Команде известно о «ветвистом» характере истории проекта в Mercurial, поэтому у него нет проблем, связанных с ветвлениями, слияниями, или несколькими головами в репозитории. Он может обрезать все ветви истории одной проверкой, который, как она работает так эффективно.

## 9.5.1. Использование команды hg bisect

Вот пример **hg bisect** в действии.

### Примечание

В версии 0.9.5 и более ранних, **hg bisect** не является встроенной командой: она распространялась с Mercurial как расширение. В этом разделе описана встроенная команды, а не старое расширение.

Теперь давайте создадим репозитарий, так что мы сможем попробовать команду **hg bisect** отдельно.

```
$ hg init mybug
$ cd mybug
```

Мы будем моделировать проект, который содержит ошибку простым образом: создать незначительные изменения в цикле, и назначить одно конкретное изменение, которое будет иметь «ошибку». Этот цикл создает 35 ревизий, каждое добавление одного файла в хранилище. Мы представляем нашу «ошибку» с файлом, который содержит текст «У меня есть губы».

```
$ buggy_change=22
$ for (( i = 0; i < 35; i++ )); do
>   if [[ $i = $buggy_change ]]; then
>     echo 'i have a gub' > myfile$i
>     hg commit -q -A -m 'buggy changeset'
>   else
>     echo 'nothing to see here, move along' > myfile$i
>     hg commit -q -A -m 'normal changeset'
>   fi
> done
```

Следующее, что мы хотели бы сделать, это понять, как использовать команду **hg bisect**. Для этого мы можем использовать встроенная справочная механизмом Mercurial.

```
$ hg help bisect
```

```
hg bisect [-gbsr] [-U] [-c CMD] [REV]

subdivision search of changesets

    This command helps to find changesets which introduce problems. To use,
    mark the earliest changeset you know exhibits the problem as bad, then
    mark the latest changeset which is free from the problem as good. Bisect
    will update your working directory to a revision for testing (unless the
    -U/--noupdate option is specified). Once you have performed tests, mark
    the working directory as good or bad, and bisect will either update to
    another candidate changeset or announce that it has found the bad
    revision.

    As a shortcut, you can also use the revision argument to mark a revision
    as good or bad without checking it out first.

    If you supply a command, it will be used for automatic bisection. Its exit
    status will be used to mark revisions as good or bad: status 0 means good,
    125 means to skip the revision, 127 (command not found) will abort the
    bisection, and any other non-zero exit status means the revision is bad.

    Returns 0 on success.

options:

 -r --reset       reset bisect state
 -g --good        mark changeset good
 -b --bad         mark changeset bad
 -s --skip        skip testing changeset
 -e --extend      extend the bisect range
 -c --command CMD use command to check changeset state
 -U --noupdate    do not update to target

use "hg -v help bisect" to show more info
```

Команда **hg bisect** работает по шагам. Каждый шаг происходит следующим образом.

1. Вы запускаете Ваш бинарный тест.
   - Если тест успешен, Вы говорите об этом запуская команду **hg bisect --good**.
   - Если неуспешен, запускаете команду **hg bisect --bad**
2. Mercurial использует вашу информацию, чтобы решить, какая ревизия для тестирования следующая
3. Он обновляет рабочий каталог до этой ревизии и процесс повторяется сначала

Процесс заканчивается, когда **hg bisect** идентифицирует уникальный набор изменений, который знаменует собой точку, где Ваш тест перешол из «успешног» в «неуспешный».

Чтобы начать поиск, должны запустить команду **hg bisect --reset**.

```
$ hg bisect --reset
```

В нашем случае мы используем очень простой бинарный тест: мы проверим, содержит ли какой-то файл в хранилище строку «У меня есть жук». Если это так, эта ревизия содержит изменения, которые «вызвали ошибку». По соглашению, ревизия, которая обладает свойством «плохо», а не ту, которая не является «хорошим».

Чаще всего, ревизия с которой синхронизирован рабочий каталог (как правило, 'typ' — последняя голова), уже столкнулись с проблемой внесенной бажной правкой, поэтому мы помечаем его как «плохое».

```
$ hg bisect --bad
```

Нашей следующей задачей является назначить ревизию, про которую известно, что она не содержит искомую ошибку; Команда **hg bisect** ограничивает свой поиск между первой парой «хорошая — плохая» ревизиия. В нашем случае, мы знаем, что 10-я ревизия не имела ошибок. (Я расскажу несколько слов о выборе первой «хорошей» ревизии позднее).

```
$ hg bisect --good 10
Testing changeset 22:e9e43d57c12e (24 changesets remaining, ~4 tests)
0 files updated, 0 files merged, 12 files removed, 0 files unresolved
```

Обратите внимание, эта команда что-то выводит [на экран]

- Он рассказал нам, как много ревизий он должен рассматривать прежде чем он сможет определить, где введена одна ошибка, и как много тестов, потребуется.
- Он обновлил рабочий каталог до следующей ревизии для тестирования, и рассказал нам, какую ревизию будем тестировать.

Теперь запустим наш тест в рабочем каталоге. Мы используем команду **grep**, чтобы увидеть, находится ли наш «плохой» файл в рабочей директории. Если это так, эта ревизия является плохой, если нет — хорошей.

```
$ if grep -q 'i have a gub' *
> then
>   result=bad
> else
>   result=good
> fi
$ echo this revision is $result
this revision is bad
$ hg bisect --$result
Testing changeset 16:a20d4936611f (12 changesets remaining, ~3 tests)
0 files updated, 0 files merged, 6 files removed, 0 files unresolved
```

Этот тест выглядит идеальным кандидатом для автоматизации, так что давайте превратим его в функцию shell.

```
$ mytest() {
>   if grep -q 'i have a gub' *
>   then
>     result=bad
>   else
>     result=good
>   fi
>   echo this revision is $result
>   hg bisect --$result
> }
```

Теперь мы можем запустить весь тест одной командой `mytest`.

```
$ mytest
this revision is good
Testing changeset 19:d018a2afc0b4 (6 changesets remaining, ~2 tests)
3 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Теперь мы можем запустить весь шаг тестирования с одной командой, mytest.

```
$ mytest
this revision is good
Testing changeset 20:850a595df5d3 (3 changesets remaining, ~1 tests)
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ mytest
this revision is good
Testing changeset 21:a3b0fef2a78f (2 changesets remaining, ~1 tests)
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ mytest
this revision is good
The first bad revision is:
changeset:   22:e9e43d57c12e
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:09:26 2012 +0000
summary:     buggy changeset
```

Хотя у нас было 40 ревизий для поиска, **hg bisect** сумел найти «ошибку» с пяти испытаний. Поскольку количество тестов, для **hg bisect** растет логарифмически с числом ревизий для поиска, то преимущество перед «методом грубой силы» увеличивается с каждой добавленной ревизией.

### 9.5.2. Очистка после поиска

Когда вы закончили работать с **hg bisect**, вы можете использовать **hg bisect --reset** чтобы сбросить информацию, которую он использовал для проведения вашего поиска. Команда не использует много места, поэтому не страшно, если вы забыли запустить эту команду. Тем не менее, **hg bisect** не позволит вам начать новый поиск в этом хранилище, пока вы не сделаете **hg bisect --reset.**

```
$ hg bisect --reset
```

## 9.6. Советы для эффективного поиска ошибок

### 9.6.1. Давайте согласованный ввод

Команда **hg bisect** требует, чтобы вы правильно сообщали о результатах каждого теста, который вы выполняете. Если вы скажете, что это испытание не прошло, когда это действительно удалось, иногда можно обнаружить несоответствие. Если удалось определить несоответствия в Ваших отчетах, Mercurial сообщит Вам, что некоторая ревизия и хорошая и плохая. Однако, он не может делать это всегда, чаще он укажет на неправильную ревизию в качестве источника ошибки.

### 9.6.2. Автоматизируйте как можно больше

Когда я начал использовать команду **hg bisect,** я пытался несколько раз выполнить свои тесты вручную, в командной строке. С таким подходом я убил по крайней мере, первую половину дня. После нескольких попыток, я обнаружил, что я делал много ошибок, из-за которых мне приходилось перезапускать поиск несколько раз, чтобы наконец, получить правильные результаты.

Мои первоначальные проблемы с управлением командой **hg bisect** вручную происходили даже с простым поиском по малым репозиториям, а если проблема, которую вы ищете более тонкая, или количество тестов, которые **hg bisect** необходимо выполнить возрастает, вероятность ошибки оператора испортить поиск гораздо выше. Как только я начал автоматизировать мои тесты, я получал лучше результаты.

Ключ к автоматизированному тестированию является двойным:

- всегда проверяйте одни и те же симптомы, и
- всегда передавайте согласованный ввод для команды **hg bisect**

В моем учебном примере приведенном выше, команда **grep** тестирует симптом, и условие `if` принимает результат этой проверки и гарантирует, что мы всегда передаём тот же самое на вход команды **hg bisect.** Функция `mytest` связывается с этими тегами воспроизводимым образом, чтобы каждый тест являлся однородным и последовательным.

### 9.6.3. Проверка ваших результатов

Потому что вывод поиска **hg bisect** хороша лишь на входных данных которые вы ему передаёте, не принимайте этот отчёт с набором ревизий, как абсолютную истину. Самым простым способом для перекрестной проверки его отчёта, будет запуск вручную тестов на каждой из следующих ревизий:

- Ревизия, которая помечена первой плохой ревизией. Ваш тест должен сообщать о ней что она по прежнему плохая.
- Родителей, этой ревизии (каждого из родителей, если это слияние). Ваш тест должен сообщить об этих ревизиях как хороших.
- Потомка этой ревизии. Ваш тест должен сообщить об этой ревизии как о плохой.

## 9.6.4. Остерегайтесь интерференции ошибок

Вполне возможно, что ваш поиск одной ошибки может быть нарушен присутствием другой. Например, предположим, что в вашей программе ошибка в ревизии 100, и она работала правильно на ревизии 50. Кто-то неизвестный вам вставил различные ошибки в ревизию 60, и исправил их в ревизии 80. Это может исказить ваш результат одним из нескольких способов.

Вполне возможно, что эта совершенно другая ошибка «маскирует» вашу, и есть шанс, что эта ошибка проявит себя перед вашей ошибкой. Если вы не можете избежать этой другой ошибки (например, она не даёт вашему проекту собраться), и поэтому нельзя сказать есть ли ваша ошибка в конкретной ревизии, команда **hg bisect** не может помочь вам непосредственно. Вместо этого, вы можете пометить непроверяемую ревизию, запустив **hg bisect --skip**.

Другая проблема может возникнуть, если тест на наличие ошибка не является достаточно точным. Если вы проверяете условие «моя программа падает», то и ваша ошибка и сбой связанный с ошибкой, маскирующей вашу, будет выглядеть одинаково, и введёт в заблуждение **hg bisect**.

Еще одна ситуация, в которой полезно использовать **hg bisect --skip**, если вы не можете проверить ревизию, так как ваш проект был в сломан и, следовательно, нетестируем в той ревизии, возможно, что кто-то уже проверил в изменениях, которые приводили к провалу сборки проекта.

## 9.6.5. Опора вашего ленивого поиска

Выбрать первые «хорошую» и «плохую» ревизии, которые означают конечные точки поиска, зачастую нелегко, но тем не менее это приводит к небольшой дискуссии. С точки зрения **hg bisect**, «новейшая» ревизия условно обозначается как «плохая», а самая старшая ревизия как «хорошая».

Если вы не помните, когда была подходящая «хорошая» ревизия, о которой вы можете сказать **hg bisect**, вы можете сделать хуже, чем при тестировании ревизий наугад. Просто помните, для ликвидации ошибок, важны моменты когда ошибка появится не может(возможно, потому что программы с ошибкой ещё не было), и те моменты, когда одна проблема маскирует другую (как я говорил выше).

Даже если вы в конечном итоге «ранняя» ревизия будет за тысячу ревизий или месяцев истории, вы увидите добавление только нескольких тестов **hg bisect**, которые необходимо выполнить, благодаря логарифмическому поведению.

# Глава 10. Обработка событий в репозитории с помощью ловушек

Mercurial предлагает мощный механизм, позволяющий автоматизировать действия при возникновении в репозитории каких-либо событий. В некоторых случаях Вы даже можете управлять реакцией Mercurial на эти события.

Mercurial использует для этих действий название ловушка (hook). Ловушки в некоторых системах управления версиями называются «триггерами», но оба этих названия относятся к одной и той же идее.

## 10.1. Обзор ловушек Mercurial

Это краткий список ловушек, поддерживаемых Mercurial. Мы подробно опишем каждую из ловушек позднее, в разделе Раздел 10.7, «Информация для разработчиков ловушек».

Каждая из ловушек, описанная как «Контролирующая» имеет возможность определить, может ли быть продолжено действие. Если ловушка выполнилась успешно, то действие может продолжаться, если нет, действие либо не разрешается либо отменяется, в зависимости от ловушки.

- `changegroup`: Выполняется после группы ревизий, внесённых в хранилище извне.
- `commit`: Выполняется после создания новой ревизии в локальном хранилище.
- `incoming`: Однократно выполняется для каждого нового набора ревизии, внесённой в репозиторий извне. Примечание: отличается от `changegroup` тем, что выполняется однократно перед внесением группы ревизий.
- `outgoing`: Выполняется после передачи группы ревизий из этого репозитория.
- `prechangegroup`: Выполняемся перед началом приёма группы ревизии в репозиторий.
- `precommit`: Управляющая. Выполняется перед началом фиксации.
- `preoutgoing`: Управляющая. Выполняется перед началом передачи группы ревизий из этого репозитория.
- `pretag`: Управляющая. Выполняется перед созданием tag'а.
- `pretxnchangegroup`: Управляющая. Выполняется после группы ревизий, принятых в локальный репозиторий из других, но до окончательной обработки транзакции в репозитории, которая сделает изменения постоянными.
- `pretxncommit`: Управляющая. Выполняется после создания нового набора изменений в локальном хранилище, но перед их фиксацией.
- `preupdate`: Управляющая. Выполняется перед началом обновления или слияния рабочей директории.
- `tag`: Выполняется после создания tag'а.
- `update`: Выполняется после завершения обновления или слияния рабочей директории.

## 10.2. Ловушки и безопасность

### 10.2.1. Ловушки выполняются с Вашими привелегиями

При запуске Вами команд Mercurial в хранилище и команд, вызывающих выполнение ловушек, эти ловушки выполняются на Вашей системе, в Вашем аккаунте и с Вашим уровнем доступа. Ловушки частично состоят из исполняемого кода, так что относитесь к ним с достаточным подозрением. Не устанавливайте ловушку, если не знаете Кто и Зачем её написал и Что она делает.

В некоторых случаях Вы можете столкнуться с ловушками, которые Вы не устанавливали себе. Если вы работаете на незнакомой системе, Mercurial может выполнять ловушки, определённые в системном глобальном `~/.hgrc` файле.

Если вы работаете с репозиторием, принадлежащем другому пользователю, Mercurial может запускать ловушки, определённые в пользовательском репозитории, но он будет работать от «Вас». Например, если Вы выполняете **hg pull** из этого репозитория, а его `.hg/hgrc` определяет локальные `outgoing` ловушки, то ловушка будет работать под вашей учетной записью пользователя, даже если вы не владелец репозитория.

### Примечание

Это применимо только если Вы забираете изменения из репозитория в локальной или сетевой файловой системе. Если Вы забираете изменения по http или ssh, то любая `outgoing` ловушка будет запущена под аккаунтом, из-под которого выполняется серверный процесс, на сервере.

Для просмотра ловушек, определённых в репозитории, используйте команду **hg showconfig hooks**. Если Вы работаете в одном репозитории, но общаетесь с другим, которыми не владеете (например, используете **hg pull** или **hg incoming**), запомните что это ловушки другого хранилища, не Ваши, и Вы должны их проверять.

## 10.2.2. Ловушки не распространяются

В Mercurial ловушки не попадают под контроль ревизий и не распространяются, когда Вы клонируете репозиторий, или забираете изменения из репозитория. Причина этого проста: ловушка это произвольный кусок исполняевого кода. Она работает от Вашей учётной записи, с Вашим уровнем привелегий, на Вашей машине.

В любой распределенной системе контроля версий было бы крайне безрассудно реализовывать версионноконтролируемые ловушки, так как это даёт легкий способ компроментации аккаунтов пользователей системы управления версиями.

Mercurial не распространяет ловушки, и, при работе с другими людьми над совместным проектом, Вы не должны полагать, что другие используют те же ловушки Mercurial, что и Вы, или что они их правильно настроили. Вы должны документировать ловушки, если ожидаете их использование другими.

В корпоративной интрасети это несколько легче контролировать, так как Вы можете, например, создать «стандартную» установку Mercurial на файловой системе NFS и использовать общесистемный `~/.hgrc` файл для определения ловушек для всех пользователей. Тем не менее, здесь тоже есть ограничения; см. ниже.

## 10.2.3. Возможно переопределение ловушек

Mercurial позволяет изменить ловушку, переопределив её. Вы можете отключить ловушку, установив значение равным пустой строке, или по своему желанию изменить её поведение.

Вы должны понимать при распространении общесистемного или общесайтового `~/.hgrc` файла, определяющего какие-либо ловушки, что пользователи могут отключать или переопределять эти ловушки.

## 10.2.4. Обеспечение выполнения критических ловушек

Иногда Вам может понадобится соблюдение политики, и Вы не хотите, чтобы другие могли её обойти. Например, требование, что каждое изменение должно проходить строгий набор тестов. Определение этого требования с помощью ловушки в общесайтовом `~/.hgrc` не будет работать для удаленных пользователей с ноутбуками, и, конечно, локальные пользователи также могут её обойти, переопределив ловушку.

Вместо этого, Вы можете настроить политику использования Mercurial так, чтобы люди распространяли изменения через хорошо известный «канонический» сервер, который защищён и должным образом настроен.

Один из способов осуществления этого — сочетание социальной инженерии и технологии. Настройте ограниченный доступ к учетной записи: пользователи могут отправлять изменения по сети в репозиторий этой учетной записи, но не могут входить в аккаунт и запускать команды оболочки. В этом случае пользователи смогут отправлять изменения, содержащие любое любимое ими старьё.

Когда кто-то отправляет изменения на сервер, с которого все их забирают, то сервер проверит ревизию, прежде чем она будет принята как постоянная, и отвергнет его, если ревизия не пройдёт испытаний. Если человек только забирает изменения с этого фильтрующего сервера, то у него будет уверенность, что все им забираемые изменения были автоматически проверены.

# 10.3. Краткое руководство по использованию ловушек

Писать ловушки Mercurial легко. Давайте начнем с ловушки, которая запускается при окончании **hg commit**, и просто печатает хэш только что созданной ревизии. Ловушка вызывается `commit` (фиксацией).

Все ловушки работают по той же схеме, что и пример

```
$ hg init hook-test
$ cd hook-test
$ echo '[hooks]' >> .hg/hgrc
$ echo 'commit = echo committed $HG_NODE' >> .hg/hgrc
$ cat .hg/hgrc
[hooks]
commit = echo committed $HG_NODE
$ echo a > a
$ hg add a
$ hg commit -m 'testing commit hook'
committed df3f65052fb97482958fed9fde988a6550da98fc
```

Вы добавляете запись в разделе `hooks` вашего `~/.hgrc`. Слева находится название действия на которое он сработает; справа действие которое следует предпринять. Как вы видите, вы можете запустить произвольные команды оболочки в ловушке. Mercurial передаёт дополнительную информацию ловушке через переменные окружения (ищите `HG_NODE` в примере).

## 10.3.1. Выполнение нескольких действий на событие

Довольно часто вы хотелось бы задать более чем одну ловушку для определенного рода событий, как показано ниже.

```
$ echo 'commit.when = echo -n "date of commit: "; date' >> .hg/hgrc
$ echo a >> a
$ hg commit -m 'i have two hooks'
committed 95279e34c052b63a7ffa8778ae428644dd2071a6
date of commit: Thu Feb  2 14:09:53 GMT 2012
```

Mercurial позволяет сделать это путем добавления расширения к концу имени ловушки. Вы расширяете имя ловушки, добавляя к названию ловушки (после «`.`») произвольный текст. Например, Mercurial будет работать как с `commit.foo`, так и с `commit.bar` при событии фиксации (`commit`).

Чтобы задать четкий порядок выполнения, когда есть несколько ловушек, определенных для события, Mercurial сортирует ловушки в по расширению, и выполняет команды ловушки в этом порядке сортировки. В приведенном выше примере, он будет выполнять в следующем порядке `commit`, `commit.bar`, `commit.foo`.

Это хорошая идея использовать несколько описательных расширений при определении новых ловушек. Это поможет вам вспомнить для чего эта ловушка. Если ловушка не выполнилась, вы получите сообщение об ошибке, которое содержит имя и расширение ловушки, поэтому использование описательных расширений могло бы сразу дать вам намек на то, почему не сработала ловушка (смотрите раздел Раздел 10.3.2, «Управление возможностью выполнения действия» для примера).

## 10.3.2. Управление возможностью выполнения действия

В наших предыдущих примерах мы использовали ловушку `commit`, которая запускается после завершения фиксации. Это одна из нескольких ловушек Mercurial, которые выполняются после того, как действие произошло. Такие ловушки не имеют возможности влиять на само действие.

Mercurial определяет целый ряд событий, которые происходят до начала действия, или после его начала, но до его завершения. В ловушки, которые срабатывают при этих событиях, добавлена возможность выбора, продолжить ли выполнение этого действия, или прервать.

Ловушка `pretxncommit` запускается когда фиксация почти полностью завершена. Другими словами, метаданные, представляющие набор изменений были записаны на диск, но транзакция до сих пор не завершена. Ловушка `pretxncommit` имеет возможность решить, будет ли транзакция завершена или должен произойти откат.

Если ловушка `pretxncommit` завершает свою работу с кодом возврата 0, транзакция завершается; фиксация заканчивается, и выполняется ловушка `commit`. Если ловушка `pretxncommit` выходит с ненулевым кодом возврата, транзакция откатывается; метаданные, представляющие изменения стираются, а ловушка `commit` не выполняется.

```
$ cat check_bug_id
#!/bin/sh
# check that a commit comment mentions a numeric bug id
hg log -r $1 --template {desc} | grep -q "\<bug *[0-9]"
$ echo 'pretxncommit.bug_id_required = ./check_bug_id $HG_NODE' >> .hg/hgrc
$ echo a >> a
$ hg commit -m 'i am not mentioning a bug id'
transaction abort!
rollback completed
abort: pretxncommit.bug_id_required hook exited with status 1
$ hg commit -m 'i refer you to bug 666'
committed 13b236affeff106b31ae144e61b115cc36f3a63e
date of commit: Thu Feb  2 14:09:54 GMT 2012
```

Ловушка в приведенном выше примере проверяет, что комментарий при фиксации содержит id бага. Если он есть, фиксация завершается. Если нет, то происходит откат.

# 10.4. Написание собственных ловушек

Когда вы пишете ловушку, то может оказаться полезным для запускать Mercurial или с опцией `-v`, или параметром конфигурации `verbose` со значением «true». Когда вы это делаете, Mercurial выведет сообщение перед вызовом каждой ловушки.

## 10.4.1. Выбор того, как должна работать ваша ловушка

Вы можете написать ловушку как обычную программу, обычно скрипт на языке shell, или, как функции Python, которая выполняется в рамках процесса Mercurial.

Написание ловушки, как внешней программы имеет то преимущество, что не требует знаний о внутреннем строении Mercurial. Вы можете вызвать нормальную команду Mercurial для получения любой дополнительной информации. Недостаток в том, что внешние ловушки медленнее, чем ловушки внутри процесса.

Ловушка в процессе выполнения Python имеет полный доступ к Mercurial API, и не «создаёт» другой процесс, так что по своей сути быстрее, чем внешняя ловушка. Кроме того, легче получить больше информации, которая требуется ловушке с помощью API, чем выполнив команды Mercurial.

Если вы знакомы с Python, или требуются ловушки с высокой производительностью, использование ловушек в Python может быть хорошим выбором. Однако, если у вас есть простая ловушка и не нужно заботиться о производительности (вероятно, большинство ловушек), скрипт, это прекрасно.

## 10.4.2. Параметры ловушек

Mercurial вызывает каждую ловушку с набором четких параметров. В python, параметр передается в качестве аргументов функции ловушки. Для внешней программы, параметры передаются как переменные среды.

Написана ли ваша ловушка в python или скриптом, конкретные имена и значения параметров будут одинаковы. Логический параметр будет представлен как логическое значение в python, и как число 1 (для «true») или 0 (для «false»), в случае переменных среды для внешних ловушек. Если параметр ловушки называется `foo`, имя аргумента

ловушки в python будет также называться `foo`, в то время как переменная среды для внешних ловушек будет называться `HG_FOO`.

## 10.4.3. Возвращаемое значение ловушки и контроль за действием

Успешно выполнившаяся ловушка, должна выйти с кодом возврата 0, если она внешняя, или вернуть логическое «false», если внутрипроцессная. Отказ обозначается ненулевым кодом возврата из внешних ловушки, или внутри процесса ловушка возвращает логическое «true». Если в процессе выполнения ловушка генерирует исключение, ловушка считается выполнившейся неудачно.

Для контролирующей действие ловушки 0/false означает «разрешить», не-ноль/true/исключение означает «запретить».

## 10.4.4. Написание внешних ловушек

При определении внешней ловушки в вашем `~/.hgrc` и ловушка запускаются, её переменные передаются вашей оболочке, которая интерпретирует её. Это означает, что вы можете использовать обычные конструкции shell в теле ловушки.

Исполняемая ловушка всегда выполняется с текущей директорией установленой в корневую директорию репозитория.

Каждый параметр ловушки передается в качестве переменной окружения, имя в верхнем регистре с префиксом «`HG_`».

За исключением параметров ловушек, Mercurial не устанавливает и не изменяет переменные окружения при запуске ловушки. Это полезно помнить, если вы пишете ловушку на веб-сервере, который может быть запущен под целым рядом различных пользователей с различными наборами переменных окружения. В многопользовательской среде, вы не должны полагаться на переменные окружения, которые установлены в значения вашей среды, при тестировании ловушки.

## 10.4.5. Приказ Mercurial использовать внутренние ловушки

Синтаксис `~/.hgrc` для определения в внутренних ловушек несколько иной, чем для внешних ловушек. Значение ловушки должно начинаться с текста «`python:`», и продолжаться полностью определённым именем вызываемого объекта для использования в качестве значения ловушки.

Модуль, в котором находится ловушка автоматически импортируются при выполнении ловушки. До тех пор пока у вас есть имя модуля и правильный `PYTHONPATH`, это должно «просто работать».

Следующий `~/.hgrc` фрагмент иллюстрирует синтаксис и смысл понятий, которые мы только что описали.

```
[hooks]
commit.example = python:mymodule.submodule.myhook
```

Когда Mercurial запускает ловушку `commit.example`, он импортирует `mymodule.submodule`, ищет объект с именем `myhook`, и вызывает его.

## 10.4.6. Написание внутрипроцессных ловушек

Простейших внутрипроцессный хук ничего не делает, но иллюстрирует основную формат api хуков:

```
def myhook(ui, repo, **kwargs):
    pass
```

Первый аргумент ловушки в python всегда `ui` объекта. Второй представляет собой репозиторий объектов, используемый в данный момент, это всегда `localrepository`. После этих двух аргументов идут другие аргументы. Какие аргументы будут переданы зависит от ловушки, но ловушка может игнорировать аргументы она не заботится о помещении их в аргументы словари такие как `**kwargs` выше.

# 10.5. Несколько примеров ловушек

## 10.5.1. Проверка сообщений при фиксации

Трудно представить себе полезное сообщение при фиксации слишком коротким. Простая ловушка `pretxncommit` в приведенном ниже примере поможет предотвратить фиксацию ревизии с сообщением, которое составляет менее 10 байт.

```
$ cat .hg/hgrc
[hooks]
pretxncommit.msglen = test `hg tip --template {desc} | wc -c` -ge 10
$ echo a > a
$ hg add a
$ hg commit -A -m 'too short'
transaction abort!
rollback completed
abort: pretxncommit.msglen hook exited with status 1
$ hg commit -A -m 'long enough'
```

## 10.5.2. Проверка на конечные пробелы

Интересным эффект при использовании связанных с commit ловушек, чтобы помочь вам писать чистый код. Простой пример «чистого кода» является утверждение о том, что изменения не должны добавлять новые строки текста, которые содержат «конечные пробелы». Конечные пробелы это ряд пробелов и табуляций в конце строки текста. В большинстве случаев, конечные пробелы не нужны, представляют невидимый шум, но это иногда это проблема, и люди часто предпочитают, избавляться от них.

Вы можете использовать `precommit` или `pretxncommit` ловушку для проверки конечных пробелов. Если вы используете ловушку `precommit`, ловушка не будет знать, какие файлы вы фиксируете, так что придется проверять каждый измененный файл в репозитории на завершающие пробелы. Если вы хотите зафиксировать изменения в файле `foo`, но файл `bar` содержит конечные пробелы, проверка в ловушке `precommit` может предотвратить фиксацию `foo` из-за проблемы с `bar`. Это кажется не правильным.

Если вы выберите ловушку `pretxncommit`, проверка выполнится незадолго до завершения транзакции. Это позволит вам проверить проблемы только в тех файлах, которые фиксируются. Однако, если вы ввели сообщение при фиксации в интерактивном режиме и ловушка не выполнилась, то транзакция будет отозвана, вы должны будете заново ввести сообщение после того как исправите конечные пробелы и запустить **hg commit** еще раз.

```
$ cat .hg/hgrc
[hooks]
pretxncommit.whitespace = hg export tip | (! egrep -q '^\+.*[ \t]$')
$ echo 'a ' > a
$ hg commit -A -m 'test with trailing whitespace'
adding a
transaction abort!
rollback completed
abort: pretxncommit.whitespace hook exited with status 1
$ echo 'a' > a
$ hg commit -A -m 'drop trailing whitespace and try again'
```

В вышеприведенном примере мы используем ловушку `pretxncommit`, которая проверяет текст на лишние пробелы в конце строк. Несмотря на простой и короткий код ловушка не очень полезная, так как всего лишь прерывает выполнение со статусом ошибки при попытке добавления строки из файла с лишними пробелами — но при этом не говорит ничего о том, где именно можно найти ошибку. У ловушки есть еще одна особенность — она проверяет на лишние пробелы только изменённые в файле строки, полностью игнорируя неизменённые.

```
#!/usr/bin/env python
#
# save as .hg/check_whitespace.py and make executable

import re

def trailing_whitespace(difflines):
    #
```

```
    linenum, header = 0, False

    for line in difflines:
        if header:
            # remember the name of the file that this diff affects
            m = re.match(r'(?:---|\+\+\+) ([^\t]+)', line)
            if m and m.group(1) != '/dev/null':
                filename = m.group(1).split('/', 1)[-1]
            if line.startswith('+++ '):
                header = False
            continue
        if line.startswith('diff '):
            header = True
            continue
        # hunk header - save the line number
        m = re.match(r'@@ -\d+,\d+ \+(\d+),', line)
        if m:
            linenum = int(m.group(1))
            continue
        # hunk body - check for an added line with trailing whitespace
        m = re.match(r'\+.*\s$', line)
        if m:
            yield filename, linenum
        if line and line[0] in ' +':
            linenum += 1

if __name__ == '__main__':
    import os, sys

    added = 0
    for filename, linenum in trailing_whitespace(os.popen('hg export tip')):
        print >> sys.stderr, ('%s, line %d: trailing whitespace added' %
                              (filename, linenum))
        added += 1
    if added:
        # save the commit message so we don't need to retype it
        os.system('hg tip --template "{desc}" > .hg/commit.save')
        print >> sys.stderr, 'commit message saved to .hg/commit.save'
        sys.exit(1)
```

Выше версия гораздо более сложная, но и более полезная. Она анализирует унифицированный формат, чтобы увидеть, если добавились строки с конечными пробелами она печатает имя файла и номер строки, для каждого такого происшествия. Еще лучше, если изменение добавляет конечные пробелы, это ловушка сохраняет комментарии и печатает имя сохраненного файла перед выходом и говорит mercurial, откатить транзакцию, так что вы можете использовать опцию `-l filename` команды **hg commit** для повторного использования сохраненного сообщения фиксации, как только вы исправите проблемы.

```
$ cat .hg/hgrc
[hooks]
pretxncommit.whitespace = .hg/check_whitespace.py
$ echo 'a ' >> a
$ hg commit -A -m 'add new line with trailing whitespace'
a, line 2: trailing whitespace added
commit message saved to .hg/commit.save
transaction abort!
rollback completed
abort: pretxncommit.whitespace hook exited with status 1
$ sed -i 's, *$,,' a
$ hg commit -A -m 'trimmed trailing whitespace'
a, line 2: trailing whitespace added
commit message saved to .hg/commit.save
transaction abort!
rollback completed
abort: pretxncommit.whitespace hook exited with status 1
```

В качестве последнего отступления, к сведению в приведенном выше примере используется особенность редактора **sed** — редактирование на месте, чтобы избавиться от конечных пробелов в файле. Это краткое и достаточно полезное действие, и я буду ещё повторять его здесь (Использование **perl** тоже неплохая мера).

```
perl -pi -e 's,\s+$,,' filename
```

# 10.6. Встроенные ловушки

Mercurial поставляется с несколькими встроенными ловушками. Вы можете найти их в каталоге `hgext` дерева исходников Mercurial. Если вы используете бинарные пакеты Mercurial, ловушки будут находиться в подкаталоге `hgext`, каталога куда был установлен Mercurial.

## 10.6.1. `acl` — контроль доступа к частям репозитория

Расширение `acl` позволяет вам контролировать, какие удаленные пользователи могут отправлять ревизии в сетевой сервер. Вы можете защитить любую часть репозитория (включая весь репозиторий), так что конкретные удаленные пользователи могут отправлять ревизии, которые не влияют на охраняемую часть.

Этот модуль реализует контроль доступа на основе идентификационных данных пользователя, выполняющего отправку, а не того, кто фиксирует ревизию. Имеет смысл использовать этот ловушку, только если у вас есть заблокированный серверная среда, которая аутентифицирует удаленных пользователей, и вы хотите быть уверены, что только конкретные пользователи имеют возможность отправить изменения на этот сервер.

### 10.6.1.1. Конфигурация ловушки `acl`

Для того чтобы управлять входящими ревизиями, ловушка `acl` должна использоваться в качестве ловушки `pretxnchangegroup`. Это позволяет ей видеть, какие файлы были изменены в каждом входящей ревизии и отклонять группы ревизий, если они пытаются изменять «запрещенные» файлы. Пример:

```
[hooks]
pretxnchangegroup.acl = python:hgext.acl.hook
```

Расширение `acl` конфигурируется с помощью трех разделов.

Раздел `acl` имеет только одно поле, `sources`, в котором перечислены источники входящих ревизий, на которые ловушка обращает внимание. Вам обычно не нужно настраивать этот раздел.

- `serve`: контролирует входящие ревизии, которые поступают из удаленного репозитория по протоколу http либо ssh. Это значение по умолчанию для `sources`, и, как правило только это значение вам и нужно для этого элемента конфигурации.
- `pull`: контроль входящих ревизий, которые прибывают через вытягивание из локального репозитория.
- `push`: контроль входящих ревизий, которые прибывают через отправку из локального репозитория.
- `bundle`: Контроль входящих ревизий, которые прибывают из другого хранилища с помощью bundle-ов.

Раздел `acl.allow` контролирует пользователей, которым разрешено добавлять ревизии в репозиторий. Если этот раздел отсутствует, все пользователи, которые явно не запрещены будут разрешены. Если этот раздел присутствует, все пользователи, которые явно не разрешены будут запрещены (пустой раздел означает, что все пользователи не имеют доступа).

Раздел `acl.deny` определяет, какие пользователи не могут добавлять ревизии в репозиторий. Если этот раздел отсутствует или пуст, ни для одного нет запретов.

Синтаксис для разделов `acl.allow` и `acl.deny` одинаковы. Слева каждая записи glob-шаблон, который соответствует файлам или каталогам, по отношению к корню репозитория, справа, имя пользователя.

В следующем примере, пользователь `docwriter` может только добавить ревизии в подкаталоги каталога `docs` репозитория, а `intern` может вставлять изменения в любой файл или каталог, кроме `source/sensitive`.

```
[acl.allow]
docs/** = docwriter
[acl.deny]
source/sensitive/** = intern
```

### 10.6.1.2. Тестирование и поиск ошибок

Если вы хотите проверить ловушку `acl`, запустите его с включонным режимом отладки Mercurial. Поскольку вы, вероятно, будете запускать его на сервере, где это не удобно (или возможно только иногда) передайте опцию `--debug`, не забывайте, что вы можете разрешить отладку в файле `~/.hgrc`:

```
[ui]
debug = true
```

С включенной отладкой ловушка `acl` будет печатать достаточно информации, чтобы вы выяснили, почему разрешается или запрещается отправка от определенных пользователей.

## 10.6.2. `bugzilla` — интеграция с Bugzilla

Расширение `bugzilla` добавляет комментарии к ошибке Bugzilla, когда находит ссылку на id бага в комментарии фиксации. Вы можете установить эту ловушку на общем сервере, так что каждый раз, удаленный пользователь передавая изменения на этот сервер, будет запускать ловушку.

Расширение добавляет комментарий к ошибке, которая выглядит следующим образом (вы можете настроить содержание комментария — смотрите ниже):

```
Changeset aad8b264143a, made by Joe User
 <joe.user@domain.com> in the frobnitz repository, refers
 to this bug. For complete details, see
 http://hg.domain.com/frobnitz?cmd=changeset;node=aad8b264143a
 Changeset description: Fix bug 10483 by guarding against some
 NULL pointers
```

Значение этой ловушки в том, что она автоматизирует процесс обновления ошибка в любое время когда создаётся ревизия относящаяся к нему. Если вы настроите ловушку должным образом, людям будет легко просмотреть ревизию, которая ссылается на данный баг, прямо из комментариев bugzilla.

Вы можете использовать код в этой книге в качестве отправной точки для некоторых более экзотических рецептов интеграции с Bugzilla. Вот несколько возможностей:

- Требование, чтобы каждая ревизия помещаемая на сервер имела правильный id ошибки в комментарии фиксации. В этом случае, нужно настроить ловушку `pretxncommit`. Это позволило бы ловушке отклонять изменения, которые не содержат id ошибки.
- Разрешить входящие наборы изменений и автоматически изменять состояние ошибки, также как и добавлять комментарии. Например, ловушка может распознавать строку «исправлена ошибка 31337», как индикатор того, что следует обновлять состояние ошибки 31337 на «требует проверки».

### 10.6.2.1. Конфигурация ловушки `bugzilla`

Вы должны настроить эту ловушку в `~/.hgrc` сервера как ловушку `incoming`, например, следующим образом:

```
[hooks]
incoming.bugzilla = python:hgext.bugzilla.hook
```

Из-за специализированного характера этой ловушки, и потому, bugzilla не была задумана для такой интеграции, настройка этой ловушки является довольно сложным процессом.

Прежде чем начать, вы должны установить поддержку mysql для python на компьютеры, где вы будете устанавливаете ловушку. Если это не доступно в бинарном пакете для вашей системы, вы можете скачать его с [web:mysql-python].

Информация о конфигурации этой ловушке находится в разделе `bugzilla` вашего `~/.hgrc`.

- `version`: версия bugzilla, установленной на сервере. Схемы базы данных, которые использует bugzilla изменяются время от времени, так что ловушка должна знать, какие именно схемы использовать.
- `host`: имя хоста MySQL-сервера, который хранит ваши данные Bugzilla. База данных должна быть настроена так, чтобы принимать соединения от серверов на которых настроена ловушка `bugzilla`.

- `user`: имя пользователя, с которым нужно подключаться к серверу MySQL. База данных должна быть настроена так, чтобы этот пользователь имел возможность подключится с того компьютера, на котором настроена ловушка `bugzilla`. Этот пользователь должен иметь возможность читать и изменять таблицы Bugzilla. Значением по умолчанию для этого пункта является `bugs` — стандартное имя пользователя Bugzilla в базе данных MySQL.

- `password`: пароль пользователя MySQL, настроенного выше. Он сохраняется в виде обычного текста, так что вы должны убедиться, что неавторизованные пользователи не могут читать `~/.hgrc` файл, который хранит эту информацию.

- `db`: имя базы данных bugzilla на сервере MySQL. Значение по умолчанию этого пункта — `bugs`, стандартное имя базу данных Bugzilla для MySQL.

- `notify`: если вы хотите чтоб Bugzilla отправляла уведомление по электронной почте подписчикам после того, как ловушка добавила комментарий к ошибке, вам нужно чтобы ловушка выполнила команду, когда она обновит базу данных. Команда для запуска зависит от того, где вы установили bugzilla, но это, как правило, выглядит примерно так, если у вас есть bugzilla установлен в `/var/www/html/bugzilla`:

```
cd /var/www/html/bugzilla &&
       ./processmail %s nobody@nowhere.com
```

- Программа `processmail` Bugzilla получает ID ошибки (ловушка заменяет «`%s`» на ID ошибки) и адрес электронной почты. Также ожидается, чтоб иметь возможность писать в некоторые файлы в каталоге, в котором запускается. Если Bugzilla и эта ловушка установлены не на одном компьютере, вам нужно найти способ запустить `processmail` на сервере, где установлен Bugzilla.

## 10.6.2.2. Связывание имён тех кто фиксирует, с именами пользователей Bugzilla

По умолчанию, ловушка `bugzilla` пытается использовать адрес электронной почты коммиттера ревизии, как имя пользователя bugzilla, с которым обновляется ошибка. Если это не подходит к вашим потребностям, вы можете преобразовать mail-адрес коммиттера в имя пользователя Bugzilla используя раздел `usermap`.

Каждый элемент в разделе `usermap` содержит адрес электронной почты слева, и имя пользователя bugzilla справа.

```
[usermap]
jane.user@example.com = jane
```

Вы можете хранить `usermap` данные в обычном `~/.hgrc`, или сказать ловушке `bugzilla` читать информацию из внешнего файла `usermap`. В последнем случае вы можете хранить `usermap` данные сами по себе (например) в модифицируемом пользователями хранилища файле. Это даёт возможность пользователям сохранять свои записи `usermap`. Обычный `~/.hgrc` файл может выглядеть следующим образом:

```
# regular hgrc file refers to external usermap file
[bugzilla]
usermap = /home/hg/repos/userdata/bugzilla-usermap.conf
```

Тогда файл `usermap`, на который он ссылается может выглядеть следующим образом:

```
# bugzilla-usermap.conf - inside a hg repository
[usermap] stephanie@example.com = steph
```

## 10.6.2.3. Настройка текста, который будет добавлен в комментарий к ошибке

Вы можете настроить текст, который добавляет эта ловушка в качестве комментария, вы укажите его в виде шаблона Mercurial. Некоторые записи в `~/.hgrc` (в секция `bugzilla` ) управляют этим поведением.

- `strip`: число ведущих элементов пути к элементу от корня пути репозитория нужно убрать для построения url. Например, если репозитории на сервере, находится в `/home/hg/repos`, и у вас есть хранилище, которое находится в `/home/hg/repos/app/tests`, то установка `strip` в `4` даст путь `app/tests`. Ловушка делать этот путь, доступным при развёртывании шаблона, как `webroot`.

- `template`: текст шаблона для использования. В дополнение к обычными связанными с ревизией переменными, в этом шаблоне можно использовать `hgweb` (значение поля конфигурации `hgweb` выше) и `webroot` (путь построенный с как описано выше).

Кроме того, вы можете добавить элемент `baseurl` в секцию `web` вашего `~/.hgrc`. Ловушка `bugzilla` сделает его доступным при развёртывании имеющихся шаблонов в качестве основы строки для использования при построении url, который позволит пользователям просматривать из комментариев Bugzilla для просмотра ревизии. Например:

```
[web]
baseurl = http://hg.domain.com/
```

Вот примерный набор конфигурационной информации для ловушки `bugzilla`.

```
[bugzilla]
host = bugzilla.example.com
password = mypassword version = 2.16
# server-side repos live in /home/hg/repos, so strip 4 leading
# separators
strip = 4
hgweb = http://hg.example.com/
usermap = /home/hg/repos/notify/bugzilla.conf
template = Changeset {node|short}, made by {author} in the {webroot}
  repo, refers to this bug.\n
  For complete details, see
  {hgweb}{webroot}?cmd=changeset;node={node|short}\n
  Changeset description:\n
  \t{desc|tabindent}
```

## 10.6.2.4. Тестирование и поиск ошибок

Наиболее распространенные проблемы с настройкой ловушки `bugzilla` относятся к работе `processmail` сценария Bugzilla и преобразовании имен коммиттеров в имена пользователей.

Напомним, что в разделе Раздел 10.6.2.1, «Конфигурация ловушки `bugzilla`» выше, что пользователь, который работает с Mercurial на сервере тотже от кого будет выполнять сценарий `processmail`. Сценарий `processmail` иногда вызывает Bugzilla для записи файлов в каталоге конфигурации и файлов конфигурации Bugzilla, как правило, принадлежащих пользователю, от имени которого работает веб-сервер.

Вы можете заставить `processmail` запускаться от имени подходящего пользователя с помощью команды **sudo**. Вот пример записи в файле `sudoers`.

```
hg_user = (httpd_user)
NOPASSWD: /var/www/html/bugzilla/processmail-wrapper %s
```

Это позволяет пользователю `hg_user` запускать программе `processmail-wrapper` под пользователем `httpd_user`.

Косвенный вызов через сценарий обёртку необходим, потому что `processmail` ожидает, что будет запущен с текущим каталогом установленным туда куда вы установили Bugzilla; вы не можете определить, такого рода ограничение в файле `sudoers`. Содержание сценария обертки простое:

```
#!/bin/sh
cd `dirname $0` && ./processmail "$1" nobody@example.com
```

Не кажется важным, адрес электронной почты, который вы передаете `processmail`.

Если `usermap` не настроен правильно, пользователи увидят сообщение об ошибке от ловушки `bugzilla`, когда они передают изменения на сервер. Сообщение об ошибке будет выглядеть следующим образом:

```
cannot find bugzilla user id for john.q.public@example.com
```

Это означает, что адрес в коммиттера, `john.q.public@example.com`, не является правильным именем пользователя Bugzilla, и у него нет записи в `usermap`, который преобразует его в имя пользователя Bugzilla.

# 10.6.3. `notify` — отправка email оповещения

Хотя встроенный веб-сервер Mercurial предоставляет rss каналы изменений в каждом репозитории, многие люди предпочитают получать уведомления об изменениях по электронной почте. Ловушка `notify` позволяет посылать уведомления множеству адресов электронной почты, когда поступают ревизии, в которых заинтересованы абоненты.

Как и в ловушке `bugzilla`, ловушка `notify` управляется с помощью шаблонов, поэтому вы можете настроить содержание уведомлений, которые она посылает.

По умолчанию, ловушка `notify` включает в себя diff каждой ревизии, которые он посылает. Вы можете ограничить размер diff или отключить эту функцию полностью. Это полезно, позволяя абонентам просматривать изменения немедленно, а не нажатием ссылки в письме.

## 10.6.3.1. Настройка ловушки `notify`

Вы можете настроить ловушку `notify`, чтобы отправлять 1 сообщение электронной почты на 1 входящую ревизию или 1 письмо на группу ревизий (все те, которые прибыли в одном вытягивании или отправке).

```
[hooks]
# send one email per group of changes
changegroup.notify = python:hgext.notify.hook
# send one email per change
incoming.notify = python:hgext.notify.hook
```

Сведения о конфигурации этой ловушки находятся в секции `notify` файла `~/.hgrc`.

- `test`: По умолчанию эта ловушка не посылает электронную почту всем, вместо этого он выдает сообщение, которое было бы отправлено. Установить этот пункт в `false`, чтобы выключить отправку электронной почты. Причина того, что отправка электронной почты по умолчанию отключена, в том что нужно несколько попыток, чтобы настроить это расширение именно так, как вы хотите, и было бы дурным тоном спамить абонентов целым набором «неправильных» уведомлений при отладке конфигурации.

- `config`: путь к конфигурационному файлу, который содержит информацию о подписке. Он храниться отдельно от основного `~/.hgrc` так что вы можете сохранить его в собственном репозитории. Люди могут затем клонировать репозиторий, и обновлять подписки и отправлять изменения обратно на сервер.

- `strip`: число ведущих символов разделителей пути в строке пути репозитория, которые обрезаются для определения пути к репозиторию абонента. Например, если репозитории на сервере, находятся в `/home/hg/repos`, а предупреждаем мы о репозитории с именем `/home/hg/repos/shared/test`, установка `strip` в `4` заставит `notify` обрезать путь до `shared/test`, и это значение будет сравниваться с подпиской абонентов.

- `template`: текст шаблона который используется при отправке сообщений. Он определяет и содержимое заголовка сообщения и его тело.

- `maxdiff`: Максимальное количество строк diff, которые добавляются к концу сообщения. Если размер diff больше, чем это значение, они будут обрезаны. По умолчанию, 300. Установите в `0`, чтобы отправлять полный diff в сообщениях с уведомлением.

- `sources`: список источников ревизий за которыми ведётся наблюдение. Это позволяет ограничить отправку уведомлений, например только на те ревизии, которые удаленные пользователи вставили в этот репозиторий через сервер. Смотрите раздел Раздел 10.7.3.1, «Источники изменений» для того чтоб узнать какие источники вы можете здесь указать.

Если вы установите пункт `baseurl` в секции `web`, вы сможете использовать его в шаблоне, он будет доступен как `webroot`.

Это пример конфигурации ловушки `notify`.

```
[notify]
# really send email
test = false
```

```
# subscriber data lives in the notify repo
config = /home/hg/repos/notify/notify.conf
# repos live in /home/hg/repos on server, so strip 4 "/" chars
strip = 4
template = X-Hg-Repo: {webroot}\n
  Subject: {webroot}: {desc|firstline|strip}\n
  From: {author}
  \n\n
  changeset {node|short} in {root}
  \n\ndetails:
  {baseurl}{webroot}?cmd=changeset;node={node|short}
  description: {desc|tabindent|strip}

[web]
baseurl =
http://hg.example.com/
```

Он выдаст сообщение, выглядящее следующим образом:

```
X-Hg-Repo: tests/slave
Subject: tests/slave: Handle error case when slave has no buffers
Date: Wed,  2 Aug 2006 15:25:46 -0700 (PDT)

changeset 3cba9bfe74b5 in /home/hg/repos/tests/slave

details:
http://hg.example.com/tests/slave?cmd=changeset;node=3cba9bfe74b5

description: Handle error case when slave has no buffers

diffs (54 lines):
diff -r 9d95df7cf2ad -r 3cba9bfe74b5 include/tests.h
--- a/include/tests.h      Wed Aug 02 15:19:52 2006 -0700
+++ b/include/tests.h      Wed Aug 02 15:25:26 2006 -0700
@@ -212,6 +212,15 @@ static __inline__
void test_headers(void *h)
[...snip...]
```

### 10.6.3.2. Тестирование и поиск ошибок

Вы не должны забывать, что по умолчанию расширение `notify` не будут посылать никакие **e-mail**, до тех пор вы явно не сконфигурируете такое поведение, установив `test` в `false`. До тех пор пока вы это не сделаете, отсылаемое сообщение будет просто выводиться.

## 10.7. Информация для разработчиков ловушек

### 10.7.1. Выполнение внутрипроцессорых ловушек

Все внутрипроцессные ловушки вызываются с аргументами в следующей форме:

```
def myhook(ui, repo, **kwargs): pass
```

Здесь параметр `ui` это объект `ui`, параметр `repo` — объект `localrepository`. Имена и значения параметров `**kwargs` зависят от вызываемой ловушки и обладают общими чертами:

- Если параметр называется `node` или `parentN`, он будет содержать шестнадцатеричный ID набора изменений. Пустая строка используется для указания «null» (исходного) набора изменений, вместо строки нулей.
- Если параметр называется `url`, он содержит URL удаленного репозитория, если он может быть определен.
- Содержащие булевые значения параметры представлены в виде обектов `bool` Python-а.

Внутрепроцессные ловушки вызываются без смены рабочей директории процесса (в отличие от внешних ловушек, которые запускаются в корне репозитория). Они также не должны менять рабочую директорию процесса, в противном случае любые вызовы Mercurial API потерпят неудачу.

Если ловушка возвращает булевое значение «false», считается что вызов прошел успешно. Если возвращается «true» или генерируется исключение, считается что вызов завершился неудачей. Полезно думать о вызовах ловушек как «скажи мне, если ты потерпел неудачу».

Помните, что ID набора изменений передается в Python как шестнадцатеричная строка, а не как бинарный хеш, который использует API Mercurial. Для конвертации ее в бинарный вид используйте функцию `bin`.

## 10.7.2. Выполнение внешних ловушек

Внешние ловушки передаются командной оболочке пользователя, запустившего Mercurial. Доступны фичи командной оболочки, как, например, подстановка переменных и команды перенаправления. Ловушка запускается в корневой директории хранилища (в отличие от in-process ловушек, которые работают в той же директории из которой был запущен Mercurial).

Параметры ловушке передаются как переменные среды. Каждое имя переменной преобразуется в верхний регистр с префиксом «`HG_`». Например, если имя параметра «`node`», имя переменной среды, представляющее параметр, будет «`HG_NODE`».

Булевы параметры представляется в виде строки «`1`» для «истинно», «`0`» для «ложь». Если переменная окружения называется `HG_NODE`, `HG_PARENT1` или `HG_PARENT2`, она содержит ID набора изменений в виде шестнадцатеричной строки. Для представления «null changeset ID» используется пустая строка вместо строки нулей. Если переменная окружения называется `HG_URL`, она будет содержать адрес удалённого хранилища, если его можно определить.

Если ловушка завершается с кодом возврата, равным нулю, то это успешное завершениие. Если она завершилась с ненулевым статусом, то «всё» плохо.

## 10.7.3. Как определить, откуда пришли изменения

Ловушку, которая срабатывает при перемещении набора изменений между локальным хранилищем и каким-либо другим можно настроить на нахождение информации «о той стороне». Mercurial знает как набор изменений передаётся, и , во многих случаях, куда и откуда они передаются.

### 10.7.3.1. Источники изменений

Mercurial сообщает ловушкам что, или откуда, используется для передачи ревизий между репозиториями. Это информация обеспечивается Mercurial-ом в параметра python с именем `source`, или переменную окружения с именем `HG_SOURCE`.

- `serve`: ревизии передаются в или из удаленного репозитория по протоколу http или ssh.
- `pull`: ревизии передаются через pull из этого репозитория в другой.
- `push`: ревизии передаются через push из этого репозитория в другой.
- `bundle`: ревизии передаются в или из комплекта.

### 10.7.3.2. Откуда пришли ревизии — URL удалённого репозитория

Когда это возможно, Mercurial сообщает ловушкам положение «той стороны», при действиях, которые передают ревизии между репозиториями. Это обеспечивается Mercurial в параметре python с именем `url`, или переменной окружения с именем `HG_URL`.

Эта информация не всегда известна. Если ловушка вызывается в репозитории, работа с которым ведется по http или ssh, Mercurial не может сказать где расположен удаленный репозиторий, но может знать клиента, который подключен к нему. В этом случае URL будет представлен в одной из следующих форм:

- `remote:ssh:ip-address` — удаленный ssh клиент по указанному адресу.

- `remote:http:ip-address` — удаленный http клиент по указанному адресу. Если клиент использует SSL, то адрес будет в виде: `https:ip-address`.
- Пустой адрес — информация о удаленном клиенте недоступна.

# 10.8. Ловушки. Описание.

## 10.8.1. `changegroup` — после внесения внешних ревизий

Эта ловушка запускается после того как группа уже существующих ревизий добавляется в репозиторий, например, через **hg pull** или **hg unbundle**. Эта ловушка выполняется один раз за операцию, добавления одной или более ревизий. В отличие от ловушки `incoming`, которая выполняется один раз на каждую ревизию, независимо от того, прибыли ли ревизии в группе.

Возможное применение для этой ловушки включать автоматические сборку или тесты для добавленных ревизий, обновление базы данных багов, или уведомлять подписчиков о том, что в репозиторий содержит новые изменения.

Параметры ловушки:

- `node` — ID ревизии. ID первой ревизии в добавляемой группе ревизий. Все ревизии между этой и `tip`, включительно, добавляются отдельными командами **hg pull**, **hg push** или **hg unbundle**.
- `source` — Строка. Источник этих изменений. Смотрите раздел Раздел 10.7.3.1, «Источники изменений» для деталей.
- `url` — URL. Местанахождение удаленного репозитория, если оно известно. Смотрите раздел Раздел 10.7.3.2, «Откуда пришли ревизии — URL удалённого репозитория» для деталей.

Смотрите также: `incoming` (раздел Раздел 10.8.3, «`incoming` — после добавления одной удаленной ревизии»), `prechangegroup` (раздел Раздел 10.8.5, «`prechangegroup` — до начала добавления ревизий удалённого репозитория»), `pretxnchangegroup` (раздел Раздел 10.8.9, «`pretxnchangegroup` — перед завершением добавления ревизий удалённого репозитория»)

## 10.8.2. `commit`—после создания новой ревизии

Эта ловушка запускается после того, как создан новая ревизия

Параметры ловушки:

- `node` — ID ревизии. ID добавляемой ревизии.
- `parent1`: ID ревизии. ID ревизии первого родителя добавляемой ревизии.
- `parent2`: ID ревизии. ID ревизии второго родителя добавляемой ревизии.

Смотрите также: `precommit` (раздел Раздел 10.8.6, «`precommit` — перед фиксацией ревизии»), `pretxncommit` (раздел Раздел 10.8.10, «`pretxncommit` — перед завершением фиксации новой ревизии»)

## 10.8.3. `incoming` — после добавления одной удаленной ревизии

Эта ловушка запускается после добавления в репозиторий ранее существующей ревизии, например, с помощью **hg push**. Если группа ревизий добавляется одной операцией, то эта ловушка вызывается для каждой добавленной ревизии.

Вы можете использовать эту ловушку для тех же целей что и ловушку `changegroup` (см. раздел Раздел 10.8.1, «`changegroup` — после внесения внешних ревизий»); просто иногда более удобно выполнить один раз ловушку для группы ревизий, в то время как в другом случае удобнее выполнять ловушку один раз для ревизии.

Параметры этого хука:

- `node`: ID ревизии. ID вновь добавляемой ревизии.
- `source`: строка. Источник ревизий. Смотрите раздел Раздел 10.7.3.1, «Источники изменений» для деталей.
- `url`: URL. Местанахождение удаленного репозитория, если оно известно. Смотрите раздел Раздел 10.7.3.2, «Откуда пришли ревизии — URL удалённого репозитория» для деталей.

Смотрите также: `changegroup` (раздел Раздел 10.8.1, «`changegroup` — после внесения внешних ревизий») `prechangegroup` (раздел Раздел 10.8.5, «`prechangegroup` — до начала добавления ревизий удалённого репозитория»), `pretxnchangegroup` (раздел Раздел 10.8.9, «`pretxnchangegroup` — перед завершением добавления ревизий удалённого репозитория»)

## 10.8.4. `outgoing` — после распространения ревизии

Эта ловушка запускается после того как группа ревизий будет распространена за пределы этого репозитория, например при использовании команд **hg push** или **hg bundle**.

Одно из применений данного хука — это уведомление администраторов что изменения были вытянуты.

Параметры этого хука:

- `node`: ID ревизии. ID первой посланной ревизии в группе ревизий.
- `source`: строка. Источник операции (см. раздел Раздел 10.7.3.1, «Источники изменений»). Если удаленный клиент забирает изменения из этого репозитория, `source` будет равен `serve`. Если клиент, который забирает изменения из этого репозитория локальный, `source` будет равен `bundle`, `pull`, или `push`, в зависимости от операции выполняемой клиентом.
- `url`: URL. Местанахождение удаленного репозитория, если оно известно. Смотрите раздел Раздел 10.7.3.2, «Откуда пришли ревизии — URL удалённого репозитория» для деталей.

Смотрите также: `preoutgoing` (раздел Раздел 10.8.7, «`preoutgoing` — до начала передачи ревизий в другие репозитории»)

## 10.8.5. `prechangegroup` — до начала добавления ревизий удалённого репозитория

Эта контролирующая ловушка запускается до того как Mercurial начнет добавлять группы ревизий из другого репозитория.

Эта ловушка не содержит информации о добавляемой ревизии, потому что она запускается до разрешения передачи этой ревизии. Если эта ловушка возвращает ошибку, ревизия не передаётся.

Эту ловушку можно использовать для запрета внешним пользователям вносить изменения в репозиторий. Например так можно «заморозить» ветку на сервере, на время или постоянно, от пользователей в то время как администратор будет модифицировать репозиторий.

Параметры этого хука:

- `source`: строка. Источник ревизий. Смотрите раздел Раздел 10.7.3.1, «Источники изменений» для подробностей.
- `url`: URL. Местанахождение удаленного репозитория, если оно известно. Смотрите раздел Раздел 10.7.3.2, «Откуда пришли ревизии — URL удалённого репозитория» для деталей.

Смотрите также: `changegroup` (раздел Раздел 10.8.1, «`changegroup` — после внесения внешних ревизий»), `incoming` (раздел Раздел 10.8.3, «`incoming` — после добавления одной удаленной ревизии»),

`pretxnchangegroup` (раздел Раздел 10.8.9, «`pretxnchangegroup` — перед завершением добавления ревизий удалённого репозитория»)

## 10.8.6. `precommit` — перед фиксацией ревизии

Эта ловушка запускается до начала фиксации Mercurial'ом новой ревизии. Выполняется до того, как Mercurial сформирует какие-либо метаданные для передачи, такие как файлы, сообщения, или даты.

Одно из использований этой ловушки — запрет возможности фиксировать новые ревизии, позволяя в то же время входящие ревизии. Другой вариант использования состоит в том, чтобы запустить сборку или тест, и разрешить фиксацию ревизии только в том случае, если сборка или тесты прошли успешно.

Параметры этого хука:

- `parent1`: ID ревизии. ID ревизии первого родителя рабочей директории.
- `parent2`: ID ревизии. ID ревизии второго родителя рабочей директории.

Если фиксация завершится, то родители рабочей директории станут родителями новой ревизии.

Смотрите также: `commit` (раздел Раздел 10.8.2, «`commit`—после создания новой ревизии»), `pretxncommit` (раздел Раздел 10.8.10, «`pretxncommit` — перед завершением фиксации новой ревизии»)

## 10.8.7. `preoutgoing` — до начала передачи ревизий в другие репозитории

Этот ловушка вызывается до того как Mercurial узнает идентификатор ревизии, который будет передаваться.

Один из вариантов использования для этой ловушки — предотвращение передачи изменений в другой репозиторий.

Параметры ловушки:

- `source`: Строка. Источник операции с помощью которой пытаются получить данные из этого хранилища (см. раздел Раздел 10.7.3.1, «Источники изменений»). Смотрите документацию на параметр `source` ловушки `outgoing`, в разделе Раздел 10.8.4, «`outgoing` — после распространения ревизии», для информации о возможных значений этого параметра.
- `url`: URL. Местонахождение удаленного хранилища, если оно известно. См. раздел Раздел 10.7.3.2, «Откуда пришли ревизии — URL удалённого репозитория» для получения дополнительной информации.

Смотрите также: `outgoing` (раздел Раздел 10.8.4, «`outgoing` — после распространения ревизии»)

## 10.8.8. `pretag` — перед тегированием ревизии

Этот контролирующая ловушка запускается до создания тега. Если ловушка выполняется успешно, создание тега продолжается. Если ловушка не выполняется удачно, тег не создается.

Параметры ловушки:

- `local`: Логическая. Определяет является ли новый тег локальным для экземпляра репозитория (т.е. сохранен в `.hg/localtags`) или управляется Mercurial (сохранен в `.hgtags`)
- `node`: ID ревизии. ID ревизии, которая была тегирована.
- `tag`: Строка. Имя тега, который был создан.

Если тег должен быть создан под управлением ревизиями, ловушки `precommit` и `pretxncommit` (в разделе Раздел 10.8.2, «`commit`—после создания новой ревизии» и в разделе Раздел 10.8.10, «`pretxncommit` — перед завершением фиксации новой ревизии»), также будут запущены.

Смотрите также: `tag` (раздел Раздел 10.8.12, «`tag` — после создания метки ревизии»)

## 10.8.9. `pretxnchangegroup` — перед завершением добавления ревизий удалённого репозитория

Этот контрольная ловушка запускается перед завершением транзакции, которая управляет добавлением группы новых ревизий из другого репозитория. Если ловушка выполняется успешно, транзакция завершается, и все ревизии становятся постоянными в этом репозитории. Если ловушка срывается, транзакция откатывается, а данные ревизий стираются.

Эта ловушка можете получить доступ к метаданным, связанными с почти добавленными ревизиями, но она не должна ничего делать с этими данными. Она также не должна изменять рабочий каталог.

Когда запускается эта ловушка другие процессы Mercurial, имеющие доступ к этому репозиторию, они будут иметь возможность увидеть почти добавленные ревизии, как будто они постоянные. Это может привести к гонкам, если не принять меры,чтобы их избежать.

Эту ловушку можно использовать для автоматической проверки ревизий. Если ловушка не удалось, все ревизии будут «отклонены» и транзакция будет отклонена.

Параметры ловушки:

- `node`: ID ревизии. ID первой ревизии в добавляемой группе ревизий. Все ревизии между этой и `tip`, включительно, добавляются отдельными командами **hg pull**, **hg push** или **hg unbundle**.
- `source`: Строка. Источник этих изменений. Смотрите раздел Раздел 10.7.3.1, «Источники изменений» для деталей.
- `url`: URL. Местанахождение удаленного репозитория, если оно известно. Смотрите раздел Раздел 10.7.3.2, «Откуда пришли ревизии — URL удалённого репозитория» для деталей.

Смотрите также: `changegroup` (раздел Раздел 10.8.1, «`changegroup` — после внесения внешних ревизий»), `incoming` (раздел Раздел 10.8.3, «`incoming` — после добавления одной удаленной ревизии»), `prechangegroup` (раздел Раздел 10.8.5, «`prechangegroup` — до начала добавления ревизий удалённого репозитория»)

## 10.8.10. `pretxncommit` — перед завершением фиксации новой ревизии

Это управляющий хук запускается перед транзакцией, которая сделает новую фиксацию завершённой. Если хук отработал успешно, транзакция завершается и ревизия принимается в репозиторий. Если хук вернул ошибку — транзакция откатывается и принятые данные удаляются.

Этот хук может получить доступ к метаданным, связанным с почти новой ревизией, но он не должен ничего делать с этими постоянными данными. Он также не должн изменять рабочую директорию.

Пока выполняется этот хук, другие процессы Mercurial, обращающиеся к этому хранилищу, видят временное состояние хранилища как постоянное. Это может привести к ошибкам, если вы не предусмотрели мер предотвращения этого.

Параметры ловушки:

- `node`: ID ревизии. ID новой фиксируемой ревизии.
- `parent1`: ID ревизии. ID ревизии первого родителя добавляемой ревизии.
- `parent2`: ID ревизии. ID ревизии второго родителя добавляемой ревизии.

Смотрите также: `precommit` (раздел Раздел 10.8.6, «`precommit` — перед фиксацией ревизии»)

## 10.8.11. `preupdate` — перед обновлением или слиянием рабочей директории.

Эта управляющая ловушка, запускается перед началом обновления/слияния рабочей директории. Запускается только если внутренняя pre-update проверка Mercurial'a определила, что обновление/слияние возможны. Если ловушка возвращает успешный код возврата, обновление/слияние продолжается; иначе даже не начинается.

Параметры ловушки:

- `parent1`: ID ревизии. ID ревизии родителя, которой обновляется рабочая директория. Если производится слияние рабочей директории это действие не меняет этого родителя.
- `parent2`: ID ревизии. Устанавливается только когда производится слияние рабочей директории. ID ревизии, с которой производится слияние рабочей копии.

Смотрите также: `update` (раздел Раздел 10.8.13, «`update` — после обновления или слияния рабочей директории»)

## 10.8.12. `tag` — после создания метки ревизии

Ловушка выполняется после создания метки ревизии.

Параметры ловушки:

- `local`: Логическая. Определяет является ли новый тег локальным для экземпляра репозитория (т.е. сохранен в `.hg/localtags`) или управляется Mercurial (сохранен в `.hgtags`)
- `node`: ID ревизии. ID ревизии, которая была тегирована.
- `tag`: Строка. Имя тега, который был создан.

Если созданная метка попадает под контроль версий, то перед этим вызывается ловушка `commit` (раздел Раздел 10.8.2, «`commit`—после создания новой ревизии»)

Смотрите также: `pretag` (раздел Раздел 10.8.8, «`pretag` — перед тегированием ревизии»)

## 10.8.13. `update` — после обновления или слияния рабочей директории

Ловушка запускается после завершения обновления или объединения изменений. Поскольку слияние может не сработать (если внешняя команда **hgmerge** не выполнилась на конфликтных файлах), то эта ловушка сообщит — успешно ли обновление/слияние.

- `error`: Логическая. Показывает, успешно ли прошло обновление или слияние.
- `parent1`: ID ревизии. ID ревизии родителя который обновляется этой рабочей копией. Если производится слияние рабочей директории это действие не меняет этого родителя.
- `parent2`: ID ревизии. Устанавливается только когда производится слияние рабочей директории. ID ревизии, с которой производится слияние рабочей копии.

Смотрите также: `preupdate` (раздел Раздел 10.8.11, «`preupdate` — перед обновлением или слиянием рабочей директории.»).

# Глава 11. Настройка вывода Mercurial

Mercurial предоставляет мощный механизм позволяющий контролировать как будет выводиться информация. Механизм базируется на шаблонах. Вы можете использовать шаблоны для генерации специфичного вывода одной команды или настроить весь вывод встроенного web интерфейса.

## 11.1. Использование предустановленых стилей

Пакет Mercurial поставляется с некоторыми стилями вывода, которые вы можете использовать незамедлительно. Стиль — просто предустановленный шаблон, который кто-то написал и установил где-либо и который может найти Mercurial.

До того как мы рассмотрим встроенные стили, давайте взглянем на обычный вывод.

```
$ hg log -r1
changeset:   1:6f5ce6bc1898
tag:         mytag
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:09 2012 +0000
summary:     added line to end of <<hello>> file.
```

Это информативно, но занимает довольно много места — пять линий на набор изменений. Стиль `compact` уменьшает это до трех линий, представленных в разреженном виде.

```
$ hg log --style compact
3[tip]   c19f42cb5b95   2012-02-02 14:10 +0000   bos
  Added tag v0.1 for changeset b94147c22394

2[v0.1]   b94147c22394   2012-02-02 14:10 +0000   bos
  Added tag mytag for changeset 6f5ce6bc1898

1[mytag]   6f5ce6bc1898   2012-02-02 14:10 +0000   bos
  added line to end of <<hello>> file.

0   5b2b97feb76c   2012-02-02 14:10 +0000   bos
  added hello
```

Стиля `changelog` показывает выразительность и мощь шаблонного движка в Mercurial. Стиль следует принципам проекта GNU для отображения changelog[web:changelog].

```
$ hg log --style changelog
2012-02-02  Bryan O'Sullivan  <bos@serpentine.com>

 * .hgtags:
 Added tag v0.1 for changeset b94147c22394
 [c19f42cb5b95] [tip]

 * .hgtags:
 Added tag mytag for changeset 6f5ce6bc1898
 [b94147c22394] [v0.1]

 * goodbye, hello:
 added line to end of <<hello>> file.

 in addition, added a file with the helpful name (at least i hope
 that some might consider it so) of goodbye.
 [6f5ce6bc1898] [mytag]

 * hello:
 added hello
 [5b2b97feb76c]
```

Вы не будете шокированы, узнав, что стиль вывода Mercurial по умолчанию называется `default`.

### 11.1.1. Установка стиля по умолчанию

Вы можете изменить стиль вывода, который Mercurial будет использовать для каждой команды, редактируя файл `~/.hgrc`, указав стиль который вы предпочитаете использовать.

```
[ui]
style = compact
```

Если вы напишете свой стиль, вы можете использовать его, либо указав путь к файлу стиля или скопировав файл стиля в папку, в которой mercurial сможете его найти (как правило, шаблоны расположены в поддиректории `templates` того каталога, куда установлен Mercurial).

## 11.2. Команды, которые поддерживают стили и шаблоны

Все команды Mercurial «похожие на `log`» позволяют использовать стили и шаблоны: **hg incoming**, **hg log**, **hg outgoing**, и **hg tip**.

Когда я писал это руководство, только эти команды, поддерживали стили и шаблоны. Поскольку это наиболее важные команды, у которых должен настраиваться вывод, давление со стороны сообщества Mercurial по поводу возможности пользователю добавлять стили и шаблоны для других команд было небольшим.

## 11.3. Основы шаблонизации

В самом простом случае шаблон mercurial это фрагмент текста. Часть текста не меняется, в то время как другая часть развёртывается и заменяется новым текстом, в случае необходимости.

Для начала давайте посмотрим, как обычно Mercurial оформляет вывод.

```
$ hg log -r1
changeset:   1:6f5ce6bc1898
tag:         mytag
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:09 2012 +0000
summary:     added line to end of <<hello>> file.

```

Теперь, давайте выполним ту же команду, но с использованием шаблона для изменяющего данный вывод.

```
$ hg log -r1 --template 'i saw a changeset\n'
i saw a changeset
```

Приведенный выше пример иллюстрирует простейший шаблон, это просто статичный текст, напечатанный один раз для каждой ревизии. Опция `--template` команды **hg log** говорит Mercurial использовать данный текст в качестве шаблона при печати каждой ревизии.

Обратите внимание, что строки шаблона заканчиваются текстом «`\n`». Это управляющая последовательность, говорящая Mercurial печатать новую строку в конце каждого пункта шаблона. Если вы пропустите этот символ новой строки, Mercurial будет печатать все блоки на одной строке. Смотрите раздел Раздел 11.5, «Escape последовательности»; для более подробной информации о управляющих последовательностях.

Шаблон, который выводит фиксированную строку текста все время не очень полезен, давайте попробуем что-нибудь более сложное.

```
$ hg log --template 'i saw a changeset: {desc}\n'
i saw a changeset: Added tag v0.1 for changeset b94147c22394
i saw a changeset: Added tag mytag for changeset 6f5ce6bc1898
i saw a changeset: added line to end of <<hello>> file.

in addition, added a file with the helpful name (at least i hope that some might consider it so) of
 goodbye.
```

```
i saw a changeset: added hello
```

Как видите, строка «`{desc}`»> в шаблоне заменяется на выходе описанием каждой ревизии. Каждый раз, когда Mercurial читает, текст, заключенный в фигурные скобки («`{`» и «`}`»), он будет пытаться заменить фигурные скобки и текст подставляя все, что внутри. Для печати символа фигурной скобки, вы должны экранировать ее, как описано в разделе Раздел 11.5, «Escape последовательности».

# 11.4. Обычные ключевые слова шаблонов

Вы можете начать писать простые шаблоны, сразу же с помощью ключевых слов указанных ниже.

- `author`: string. Неизменяемый автор ревизии.
- `branches`: string. Название ветки, в которой была зафиксирована ревизия. Будет пустым, если имя ветки `default`.
- `date`: Дата. Дата, когда была зафиксирована ревизия. Эта дата не человекочитаема, надо пропустить её через фильтр, чтобы отобразить её соответствующим образом. Смотрите раздел Раздел 11.6, «Фильтрация ключевых слов, чтобы отобразить результат» для получения дополнительной информации об этих фильтрах. Дата выражается в виде пары чисел. Первое метка времени Unix UTC (в секундах с 1 января 1970); второе является смещением часового пояса коммиттера от UTC в секундах.
- `desc`: string. Текст описания ревизии.
- `files`: Список строк. Все файлы, измененные, добавленные или удаленные в этой ревизии.
- `file_adds`: Список строк. Файлы, добавляемые в этой ревизии.
- `file_dels`: Список строк. Файлы, удаляемые в этой ревизии.
- `node`: string. Идентификационный хеш ревизии, 40-символьная шестнадцатеричная строка.
- `parents`: Список строк. Родители ревизии.
- `rev`: Число. Номер ревизии локальный для репозитория.
- `tags`: Список строк. Любые теги, связанные с ревизией.

Несколько простых экспериментов покажет нам, чего ожидать, когда мы используем эти слова, вы можете увидеть результаты ниже.

```
$ hg log -r1 --template 'author: {author}\n'
author: Bryan O'Sullivan <bos@serpentine.com>
$ hg log -r1 --template 'desc:\n{desc}\n'
desc:
added line to end of <<hello>> file.

in addition, added a file with the helpful name (at least i hope that some might consider it so) of
 goodbye.
$ hg log -r1 --template 'files: {files}\n'
files: goodbye hello
$ hg log -r1 --template 'file_adds: {file_adds}\n'
file_adds: goodbye
$ hg log -r1 --template 'file_dels: {file_dels}\n'
file_dels:
$ hg log -r1 --template 'node: {node}\n'
node: 6f5ce6bc18989ecdb1e0ddf5d63d539787a66eb2
$ hg log -r1 --template 'parents: {parents}\n'
parents:
$ hg log -r1 --template 'rev: {rev}\n'
rev: 1
$ hg log -r1 --template 'tags: {tags}\n'
tags: mytag
```

Как мы уже отмечали выше, ключевое слово date выводит не человекочитаемый текст, поэтому мы должны обрабатывать его специально. Это связано с использованием фильтров, о которых смотрите в разделе Раздел 11.6, «Фильтрация ключевых слов, чтобы отобразить результат».

```
$ hg log -r1 --template 'date: {date}\n'
date: 1328191809.00
$ hg log -r1 --template 'date: {date|isodate}\n'
date: 2012-02-02 14:10 +0000
```

# 11.5. Escape последовательности

Движок шаблонов Mercurial распознаёт наиболее часто используемые управляющие последовательности в строках. Когда он видит символ обратной косой черты («\»), он читает следующий символ и заменяет 2 символа на один, как описано ниже.

- `\`: Обратная косая черта, «\», ASCII 134.
- `\n`: Перевод строки, ASCII 12.
- `\r`: Возврат каретки, ASCII 15.
- `\t`: Табуляция, ASCII 11.
- `\v`: Вертикальная табуляция, ASCII 13.
- `\{`: Открывающая фигурная скобка, «{», ASCII 173.
- `\}`: Закрывающая фигурная скобка, «}», ASCII 175.

Как указано выше, если вы хотите при развёртывании шаблона использовать символы «\», «{» или «}», вы должны экранировать их.

# 11.6. Фильтрация ключевых слов, чтобы отобразить результат

Некоторые из результатов расширения шаблона не просты в использовании. Mercurial позволяет указать необязательную цепочку фильтров, чтобы изменить результат расширения ключевого слова. Вы уже видели, общий фильтр, `isodate`, в действии выше, чтобы сделать дату читаемой.

Ниже приведен список наиболее часто используемых фильтров, которые поддерживает Mercurial. Некоторые фильтры могут быть применены к любому тексту, а другие могут быть использованы только в конкретных обстоятельствах. За именем каждого фильтра ниже сначала следует описание, где он может быть использован, а потом описание его действия.

- `addbreaks`: Любой текст. Добавляет тег XHTML «<br/>» перед концом каждой строки, кроме последней. Например, «`foo\nbar`» преобразуется в «`foo<br/>\nbar`».
- `age`: ключевое слово `date`. Формирует интервал времени который прошел со времени ревизии до текущего времени. Возвращает строку типа «`10 minutes`».
- `basename`: Любой текст, но наиболее полезна для работы с ключевым словом `files` и похожими. Применённый к пути файла, и верней имя файла. Например, «`foo/bar/baz`» преобразуется в «`baz`».
- `date`: ключевое слово `date`. Формирует дату в том же формате, что и команда Unix `date`, но с указанием часового пояса. Возвращает строку, например «`Mon Sep 04 15:13:13 2006 -0700`».
- `domain`: Любой текст, но наиболее полезно для ключевого слова `author`. Находит первую строку, которая выглядит как адрес электронной почты, и извлекает только доменную часть. Например, «`Bryan O'Sullivan <bos@serpentine.com>`» преобразуется в «`serpentine.com`».

- `email`: Любой текст, но наиболее полезно для ключевого слова `author`. Находит первую строку, которая выглядит как адрес электронной почты, и извлекает email. Например, «`Bryan O'Sullivan <bos@serpentine.com>`» преобразуется в «`bos@serpentine.com`».

- `escape`: Любой текст. Заменяет спецсимволы XML/XHTML «&», «<» и «>» на XML-entities.

- `fill68`: Любой текст. Разбивает текст чтоб он помещался в 68 колонок. Это полезно, сделать перед тем как передать текст на фильтр `tabindent`, если мы все еще хотим, чтобы он помещается в 80 колонок фиксированного шрифта окна.

- `fill76`: Любой текст. Разбивает текст чтоб он помещался в 76 колонок.

- `firstline`: Любой текст. Возвращает первую строку текста, без завершающих строк.

- `hgdate`: ключевое слово `date`. Генерирует дату, как пара читаемых чисел. Возвращает строку вида «`1157407993 25200`».

- `isodate`: ключевое слово `date`. Генерирует дату, как текстовую строку в формате ISO 8601. Возвращает строку вида «`2006-09-04 15:13:13 -0700`».

- `obfuscate`: Любой текст, но наиболее полезно для ключевого слова `author`. Возвращает текст преобразованный в последовательность XML entities. Это помогает победить некоторых, особенно глупых сканеров собирающих адреса для спам-ботов.

- `person`: Любой текст, но наиболее полезно для ключевого слова `author`. Возвращает текст перед адресом email. Например, «`Bryan O'Sullivan <bos@serpentine.com>`» преобразуется в «`Bryan O'Sullivan`».

- `rfc822date`: ключевое слово `date`. Генерирует дату, как текстовую строку в формате использующемся в заголовках email. Возвращает строку вида «`Mon, 04 Sep 2006 15:13:13 -0700`».

- `short`: Хеш ревизии. Возвращает короткую форму хеша ревизии. 12-символьную шестнадцатеричную строку.

- `shortdate`: ключевое слово `date`. Генерирует год, месяц и день даты. Возвращает строку вида «`2006-09-04`».

- `strip`: Любой текст. Удаляет все начальные и конечные пробелы из строки.

- `tabindent`: Любой текст. Возвращает текст в котором каждая строка, кроме первой, начинаются с символа табуляции.

- `urlescape`: Любой текст. Экранирует все символы, которые считаются «специальными» для url-анализаторов. Так, например, `foo bar` преобразуется в `foo%20bar`.

- `user`: Любой текст, но наиболее полезно для ключевого слова `author`. Возвращает часть «user» из email адреса. Например, «`Bryan O'Sullivan <bos@serpentine.com>`» преобразуется в «`bos`».

```
$ hg log -r1 --template '{author}\n'
Bryan O'Sullivan <bos@serpentine.com>
$ hg log -r1 --template '{author|domain}\n'
serpentine.com
$ hg log -r1 --template '{author|email}\n'
bos@serpentine.com
$ hg log -r1 --template '{author|obfuscate}\n' | cut -c-76
&#66;&#114;&#121;&#97;&#110;&#32;&#79;&#39;&#83;&#117;&#108;&#108;&#105;&#11
$ hg log -r1 --template '{author|person}\n'
Bryan O'Sullivan
$ hg log -r1 --template '{author|user}\n'
bos
$ hg log -r1 --template 'looks almost right, but actually garbage: {date}\n'
looks almost right, but actually garbage: 1328191809.00
$ hg log -r1 --template '{date|age}\n'
2 seconds ago
```

```
$ hg log -r1 --template '{date|date}\n'
Thu Feb 02 14:10:09 2012 +0000
$ hg log -r1 --template '{date|hgdate}\n'
1328191809 0
$ hg log -r1 --template '{date|isodate}\n'
2012-02-02 14:10 +0000
$ hg log -r1 --template '{date|rfc822date}\n'
Thu, 02 Feb 2012 14:10:09 +0000
$ hg log -r1 --template '{date|shortdate}\n'
2012-02-02
$ hg log -r1 --template '{desc}\n' | cut -c-76
added line to end of <<hello>> file.

in addition, added a file with the helpful name (at least i hope that some m
$ hg log -r1 --template '{desc|addbreaks}\n' | cut -c-76
added line to end of <<hello>> file.<br/>
<br/>
in addition, added a file with the helpful name (at least i hope that some m
$ hg log -r1 --template '{desc|escape}\n' | cut -c-76
added line to end of &lt;&lt;hello&gt;&gt; file.

in addition, added a file with the helpful name (at least i hope that some m
$ hg log -r1 --template '{desc|fill68}\n'
added line to end of <<hello>> file.

in addition, added a file with the helpful name (at least i hope
that some might consider it so) of goodbye.
$ hg log -r1 --template '{desc|fill76}\n'
added line to end of <<hello>> file.

in addition, added a file with the helpful name (at least i hope that some
might consider it so) of goodbye.
$ hg log -r1 --template '{desc|firstline}\n'
added line to end of <<hello>> file.
$ hg log -r1 --template '{desc|strip}\n' | cut -c-76
added line to end of <<hello>> file.

in addition, added a file with the helpful name (at least i hope that some m
$ hg log -r1 --template '{desc|tabindent}\n' | expand | cut -c-76
added line to end of <<hello>> file.

        in addition, added a file with the helpful name (at least i hope tha
$ hg log -r1 --template '{node}\n'
6f5ce6bc18989ecdb1e0ddf5d63d539787a66eb2
$ hg log -r1 --template '{node|short}\n'
6f5ce6bc1898
```

### Примечание

Если вы попробуете применить фильтр к фрагменту данных, который не может быть обработан, Mercurial напечатает исключение Python. Например, попытка запустить фильтр `isodate` для ключевого слова `desc`, будет не очень хорошей идеей.

## 11.6.1. Объединение фильтров

Легко объединять фильтры для получения нужного результата. Следующая цепочка фильтров приводит в порядок описание, убеждаемся что оно влезает в 68 столбцов, потом вставляем отступы в 8 символов (по крайней мере на unix-подобных системах, где табуляция условно занимает 8 символов).

```
$ hg log -r1 --template 'description:\n\t{desc|strip|fill68|tabindent}\n'
description:
 added line to end of <<hello>> file.

 in addition, added a file with the helpful name (at least i hope
 that some might consider it so) of goodbye.
```

Обратите внимание на использование «`\t`» (символ табуляции) в шаблоне, чтобы заставить первую строку отступить, это необходимо, поскольку `tabindent` вставляем отступы везде, кроме первой строки.

Имейте в виду, что порядок фильтров в цепочке является важным. Первый фильтр применяется к результату ключевого слова, второй к результату первого фильтра, и так далее. Например, использование `fill68|tabindent` даст очень разные результаты в сравнении с `tabindent|fill68`.

# 11.7. От шаблонов к стилям

Шаблон в командной строке предоставляет простой и быстрый способ для форматирования некоторого вывода. Шаблоны могут стать слишком многословным, хотя, и это полезно, чтобы иметь возможность указать имени шаблон. Файл стиля представляет собой шаблон с именем, хранящийся в файле.

Более того, использование файла стиля открывает силу движка шаблонов Mercurial таким образом, который не представляется возможным с помощью опции командной строки `--template`.

## 11.7.1. Простейшие файлы стилей

Наш простой файл стиля содержит всего одну строчку:

```
$ echo 'changeset = "rev: {rev}\n"' > rev
$ hg log -l1 --style ./rev
rev: 3
```

Это указывает Mercurial, «при печати ревизии, используй текст справа как шаблон»

## 11.7.2. Синтаксис файла стиля

Синтаксические правила файла стиля просты.

- Файл обрабатывается построчно.
- Начальные и конечные пробелы игнорируются
- Пустые строки пропускаются
- Если строка начинается с одного из символов «`#`» или «`;`», эта строка считается комментарием и пропускается как пустая.
- Строка начинается с ключевого слова. Оно должно начинаться с буквы или символа подчеркивания, и может в дальнейшем содержать любые алфавитно-цифровые символы и знак подчеркивания. (Ключевое слово должно удовлетворять следующему регулярному выражению `[A-Za-z_][A-Za-z0-9_]*`.)
- Следующий элемент должен быть символ «`=`», в окружении произвольного количества пробелов.
- Если остальная часть строки начинается и заканчивается с соответствующими кавычками (как одинарными, так и двойными), она рассматривается в качестве тела шаблона.
- Если остальная часть строки не начинается с кавычки, то она рассматривается как имя файла, содержимое этого файла будет читаться и использоваться в качестве тела шаблона.

# 11.8. Примеры файлов стиля

Чтобы показать, как писать стилевой файл, напишем несколько примеров. Вместо того, чтобы обеспечивать полный стилевой файл и проходить по нему, мы отобразим обычный процесс разработки стилевого файла, начиная с чего-то очень простого, и переходя через набор последовательно более полных примеров.

## 11.8.1. Определение ошибки в файле стиля

Если Mercurial сталкивается с проблемой в файле стиля, с которым вы работаете, он печатает краткие сообщения об ошибках на самом деле очень полезными, как только вы поймёте, что они означают.

```
$ cat broken.style
changeset =
```

Обратите внимание, что `broken.style` пытается определить ключевое слово `changeset`, но забывает дать какой-либо контент для него. Когда вы указываете Mercurial использовать этот файл стиля, он оперативно жалуется.

```
$ hg log -r1 --style broken.style
** unknown exception encountered, please report by visiting
**  http://mercurial.selenic.com/wiki/BugTracker
** Python 2.7.2 (default, Nov  3 2011, 15:14:27) [GCC 4.5.3]
** Mercurial Distributed SCM (version 2.0)
** Extensions loaded:
Traceback (most recent call last):
  File "/usr/bin/hg-2.7", line 38, in <module>
    mercurial.dispatch.run()
  File "/usr/lib64/python2.7/site-packages/mercurial/dispatch.py", line 27, in run
    sys.exit(dispatch(request(sys.argv[1:])))
  File "/usr/lib64/python2.7/site-packages/mercurial/dispatch.py", line 64, in dispatch
    return _runcatch(req)
  File "/usr/lib64/python2.7/site-packages/mercurial/dispatch.py", line 87, in _runcatch
    return _dispatch(req)
  File "/usr/lib64/python2.7/site-packages/mercurial/dispatch.py", line 684, in _dispatch
    cmdpats, cmdoptions)
  File "/usr/lib64/python2.7/site-packages/mercurial/dispatch.py", line 466, in runcommand
    ret = _runcommand(ui, options, cmd, d)
  File "/usr/lib64/python2.7/site-packages/mercurial/dispatch.py", line 738, in _runcommand
    return checkargs()
  File "/usr/lib64/python2.7/site-packages/mercurial/dispatch.py", line 692, in checkargs
    return cmdfunc()
  File "/usr/lib64/python2.7/site-packages/mercurial/dispatch.py", line 681, in <lambda>
    d = lambda: util.checksignature(func)(ui, *args, **cmdoptions)
  File "/usr/lib64/python2.7/site-packages/mercurial/util.py", line 454, in check
    return func(*args, **kwargs)
  File "/usr/lib64/python2.7/site-packages/mercurial/commands.py", line 3822, in log
    displayer = cmdutil.show_changeset(ui, repo, opts, True)
  File "/usr/lib64/python2.7/site-packages/mercurial/cmdutil.py", line 893, in show_changeset
    t = changeset_templater(ui, repo, patch, opts, mapfile, buffered)
  File "/usr/lib64/python2.7/site-packages/mercurial/cmdutil.py", line 762, in __init__
    cache=defaulttempl)
  File "/usr/lib64/python2.7/site-packages/mercurial/templater.py", line 298, in __init__
    if val[0] in "'\"":
IndexError: string index out of range
```

Это сообщение об ошибке выглядит устрашающе, но его не слишком трудно исследовать.

- Первым компонент является просто способом Mercurial сказать: «Я прерываю выполнение».

  ```
  ___abort___: broken.style:1: parse error
  ```

- Далее идёт название файла стиля который содержит ошибку

  ```
  abort: ___broken.style___:1: parse error
  ```

- После имени файла идёт номер строки, где произошла ошибка.

  ```
  abort: broken.style:___1___: parse error
  ```

- Наконец, описание того, что пошло не так.

  ```
  abort: broken.style:1: ___parse error___
  ```

- Описание проблемы, не всегда понятно (как в данном случае), но даже если оно загадочно, то проблема почти всегда тривиальна, посмотрите на указанную строку в файле стиля и посмотрите, что в ней неправильно.

## 11.8.2. Уникальный идентификатор репозитория

Если вы хотите, иметь возможность определить репозиторий Mercurial «довольно однозначно», используя короткую строку, в качестве идентификатора, вы можете использовать первую ревизию в репозитории.

```
$ hg log -r0 --template '{node}'
618e49b7b3328b9f52044186fc5089f37ebce576
```

Она может быть уникальной, и поэтому она является полезной во многих случаях. Есть несколько предостережений.

- Это не будет работать в совершенно пустом репозитории, потому что такие репозитории не имеют ревизии.
- Это не будет работать (крайне редко) в случае, когда репозиторий это слияние двух или более ранее независимых репозиториев, и вы до сих пор поддерживаете эти репозитории.

Вот некоторые варианты использования этого идентификатора:

- В качестве ключа в таблице базы данных, которая управляет хранилищами на сервере.
- В половине кортежа {**ID** репозитория, **ID** ревизии}. Сохранить эту информацию при запуске автоматической сборки и иных действиях, так что вы можете «повторить» сборку в дальнейшем, если это необходимо.

## 11.8.3. Просмотр файлов на нескольких строках

Предположим, мы хотим, чтобы список файлов изменившихся в ревизии, по одному на строке, с небольшой отступом перед каждым именем файла.

```
$ cat > multiline << EOF
> changeset = "Changed in {node|short}:\n{files}"
> file = "  {file}\n"
> EOF
$ hg log --style multiline
Changed in bb7c31cb2621:
  .bashrc
  .hgrc
  test.c
```

## 11.8.4. Вывод похожий на Subversion

Давайте попробуем подражать формату вывода который по умолчанию использует другой инструмент контроля версий, Subversion.

```
$ svn log -r9653
------------------------------------------------------------------------
r9653 | sean.hefty | 2006-09-27 14:39:55 -0700 (Wed, 27 Sep 2006) | 5 lines

On reporting a route error, also include the status for the error,
rather than indicating a status of 0 when an error has occurred.

Signed-off-by: Sean Hefty <sean.hefty@intel.com>

------------------------------------------------------------------------
```

Стиль с выводом похожим на Subversion является достаточно простым, легко скопировать и вставить кусок его вывода в файл, и заменить текст, подготовленный выше Subversion шаблонными значениями, которые мы хотели бы видеть при развёртывании.

```
$ cat svn.template
r{rev} | {author|user} | {date|isodate} ({date|rfc822date})

{desc|strip|fill76}

------------------------------------------------------------------------
```

Есть несколько мелочей, в котором этот шаблон отличается от вывода, производимого Subversion.

- Subversion печатает «читаемую» дату (как «`Wed, 27 Sep 2006`» в примере выше) в скобках. Шаблон движка Mercurial не дает возможности отображения даты в этом формате без печати времени и часового пояса.

- Мы подражать выводу Subversion вместе с «разделителем» строкой, заполненной символами «-» путем завершения шаблона такой строкой. Мы используем ключевое слово `header` шаблонного движка для печати разделительной линии в качестве первой строки вывода (см. ниже), что позволит достичь аналогичного вывода в Subversion.

- Вывод Subversion включает в себя в заголовке счётчик строк в сообщении фиксации. Мы не можем повторить это в Mercurial; шаблонный движок в настоящее время не обеспечивает фильтр, который подсчитывает количество строк сгенерированных шаблоном.

Мне потребовалось не более минуты или две работы, заменить точный текст из примера с выводом Subversion с некоторыми ключевыми словами и фильтрами используемыми в шаблоне выше. Файл стиля содержит ссылку на шаблон.

```
$ cat svn.style
header = '------------------------------------------------------------------------\n\n'
changeset = svn.template
```

Мы могли бы включить текст шаблона непосредственно в файл стиля, заключив его в кавычки и заменяя перевод строк на последовательность «`\n`», но это сделало бы файл стиля слишком трудночитаемым. Читаемость является хорошим руководством, когда вы пытаетесь решить, будет ли какой-нибудь текст принадлежит файлу стиля или файлу шаблона. Если файл стиля будет выглядеть слишком большим или непонятным если вы вставите шаблон точным фрагментом текста, поместите его лучше в шаблон.

# Глава 12. Управление изменениями с Mercurial Queues

## 12.1. Проблема управления патчами

Вот распространенная ситуация: вам нужно установить пакет программного обеспечения из исходных кодов, но вы обнаружили ошибку, которую вы должны исправить в исходниках, прежде чем начать работу с пакетом. Вы делаете изменения, забываете о пакете на некоторое время, и через несколько месяцев вам нужно провести обновление до новой версии пакета. Если новая версия пакета до сих пор с ошибкой, необходимо извлечь ваши исправления из дерева старых исходников и применить его на новой версии. Это утомительно, и тут легко сделать ошибку.

Это простой случай решения проблемы «управлением патчами». У вас есть «upstream» дерева исходных текстов, который вы не можете изменить, вы должны внести некоторые локальные изменения в части upstream дерева, и вы хотите иметь возможность сохранить эти изменения отдельно, так чтобы вы могли применить их в новых версиях upstream исходников.

Проблема управления патчами возникает во многих ситуациях. Вероятно, наиболее заметным является то, что пользователь проекта с открытым исходным кодом будет помогать сопровождающему проекта в исправлении ошибки или написанию новой функции только патчами.

Поставщикам операционных систем, включающих программное обеспечение с открытыми исходниками, часто требуется вносить изменения в пакеты которые они распространяют, чтобы они правильно установить их в своей среде.

Когда у вас есть несколько изменений, легко управлять одним патчем с использованием стандартных программ **diff** и **patch** (смотрите раздел Раздел 12.4, «Понимание патчей» для обсуждения этих инструментов). После того, как количество изменений растет, начинает иметь смысл сохранить патчи как отдельные «части работы», так чтобы, например, один патч содержал исправление только одной ошибки (патч может изменять несколько файлов, но он делает «только одну вещь»), и вы можете иметь несколько таких патчей для различных ошибок исправленых вами и локальными изменениями которые вам требуются. В этой ситуации, если вы предоставите патч, исправляющий ошибку сопровождающему пакет разработчику, и они включают ваши исправления в следующем релизе, вы можете просто удалить, один патч, при обновлении до новой версии.

Поддержка одного патча upstream дерева немного утомительно и чревато ошибками, но не трудно. Однако сложность задачи быстро растет с ростом числа исправлений которое вы должны поддерживать. Имея более чем небольшое количество патчей в руках, понимая, какие из них вы отправили разработчику и беспорядок будет впечатляющий.

К счастью, Mercurial включает в себя мощное расширение, Mercurial Queues (или проще «MQ»), которое упрощает задачу по массовому управлению патчами.

## 12.2. Предыстория Mercurial Queues

В конце 1990-х, некоторые разработчики ядра linux начали поддерживать «серии патчей», что изменило поведения ядра linux. Некоторые из этих серий были направлены на стабильность, некоторые добавляли фичи, у других был более спекулятивный характер.

Размер серии этих патчей быстро рос. В 2002 году Эндрю Мортон опубликовал некоторые скрипты, которые он использует для автоматизации задач управления своей очередью патчей. Эндрю был успешно использовал эти скрипты для управления сотней (иногда тысячей) патчей на ядро linux.

### 12.2.1. A patchwork quilt

В начале 2003 года Andreas Gruenbacher и Martin Quinson заимствовали подход сценариев Эндрю и опубликовали инструмент под названием «patchwork quilt» [web:quilt], или просто «quilt» (смотрите [gruenbacher:2005] для

документ с его описанием). Поскольку quilt существенно автоматизировало управление исправлениями, он быстро приобрел много последователей среди открытых разработчиков программного обеспечения.

Quilt управляет стеком патчей над деревом директорий. Для начала, вам говорите quilt управлять деревом каталогов, а также указываете, какими файлами вы хотите управлять, он хранит названия и содержание этих файлов. Чтобы исправить ошибку, вы создаете новый патч (с помощью одной команды), редактируете файлы которые вам нужно исправить, потом «обновляете» патч.

На шаге обновления вызывается quilt для сканирования каталогов, он обновляет патч со всеми изменениями, которые вы сделали. Вы можете создать один патч поверх первого, который будет отслеживать изменения, необходимые для изменения дерева «дерево с применённым первым патчем» к «дереву применяется второй патч».

Вы можете управлять тем, какие патчи применяются к дереву. Если «извлечь» патч, изменения, внесенные этим патчем исчезнут из дерева каталогов. Quilt помнит, какие патчи вы извлекли, хотя, вы можете так же «вставить» убраный патч еще раз, и дерево каталогов, будет восстановлено с содержащим изменения в патче. Самое главное, вы можете запустить команду «обновить» в любой момент, а верхний применяемый патч будет обновляться. Это означает, что вы можете в любое время изменить и какие патчи применяются и какие изменения внесены в эти патчи.

Quilt ничего не знает об инструментах контроля версий, так что работает одинаково над распакованными архивами или рабочей копией subversion.

### 12.2.2. От patchwork quilt до Mercurial Queues

В середине 2005 года, Крис Мейсон взял особенности quilt и написал расширение, которое он назвал Mercurial Queues, которое добавило похожее на quilt поведение к Mercurial.

Ключевой разницей между quilt и MQ является то, что quilt ничего не знает о системах управления версиями, а MQ интегрирована в mercurial. Каждый патч, который вы вставляете отображается как ревизия Mercurial. Извлекаете патч, и ревизия уходит.

Потому что quilt не зависит от инструментов контроля версий, он по-прежнему чрезвычайно полезен для программ, чтобы следить за ситуацией, когда вы не можете использовать Mercurial и MQ.

## 12.3. Огромное преимущество MQ

Я не могу переоценить значение того, что предлагает MQ объединяя патчи и контроль версий.

Одна из основных причин того, что патчи сохраняются в мире свободного программного обеспечения и открытого исходного кода, несмотря на наличие более мощных инструментов контроля версий — это гибкость которую они предлагают.

Традиционные инструменты контроля версий делают окончательную и необратимую записи всего, что вы делаете. Хотя это имеет большое значение, это также несколько удручающе. Если вы хотите выполнить дикий эксперимент, вы должны быть осторожны в том, как вы это делаете, или вы рискуете оставить ненужные или что еще хуже, вводящие в заблуждение или дестабилизирующие — следы вашей ошибки и просчета и запишете ошибку в постоянную регистрацию изменений.

MQ, же, объединяя распределенную систему контроля версий и патчи позволяет гораздо легче выделять вашу работу. Ваш патч находится на вершине общей истории изменений, и вы можете их скрыть или снова показать. Если вам не нравится патч, вы можете оставить его. Если патч не совсем такой, каким вы хотите его видеть, это просто исправить, столько раз, сколько нужно, пока вы приведёте его в форму которую вы хотите.

Для примера, интеграция патчей с контролем версий даёт понимание патчей и отладку последствий их взаимодействия с кодом чрезвычайно легче. Так как каждое применение патчей связанно с ревизией, вы можете прочитать **hg log** имя файла для просмотра ревизии и исправлений наложенных патчем. Вы можете использовать команду **hg bisect** для двоичного поиска по всем ревизиям и применять патчи, чтобы увидеть, где появилась ошибка

и где она была исправлена. Вы можете использовать команду **hg annotate** чтобы посмотреть, какие ревизии или патчи изменили данную строку исходного файла. И так далее.

# 12.4. Понимание патчей

Потому что MQ не скрывает свой патч-ориентированный характер, было бы полезно, понять, что такое патчи, а также немного о том, какие инструменты работают с ними.

Традиционная команда unix **diff** сравнивает два файла и выводит список различий между ними. Команда **patch** понимает эти различия, как изменения которые нужно внести в файл. Посмотрите ниже на простом примере из этих команд в действии.

```
$ echo 'this is my original thought' > oldfile
$ echo 'i have changed my mind' > newfile
$ diff -u oldfile newfile > tiny.patch
$ cat tiny.patch
--- oldfile 2012-02-02 14:09:55.119150885 +0000
+++ newfile 2012-02-02 14:09:55.119150885 +0000
@@ -1 +1 @@
-this is my original thought
+i have changed my mind
$ patch < tiny.patch
patching file oldfile
$ cat oldfile
i have changed my mind
```

Тип файла, который генерирует **diff** (а **patch** принимает в качестве входных данных), называется «patch» или «diff» нет никакой разницы между patch и diff. (Мы используем термин «patch», так как он чаще используется.)

Патч может начинаться с произвольного текста; команда **patch** игнорирует этот текст, но MQ использует его в качестве сообщения фиксации при создании ревизии. Чтобы найти начало содержания патча, **patch** ищет первую строку, которая начинается со строки «`diff -`».

MQ работает с унифицированным diff (**patch** может принять некоторые другие форматы просмотра, но MQ не умеет). Унифицированный формат содержит два вида заголовков. Заголовок файла описывает измененные файлы, он содержит имя файла для модификации. Когда **patch** видит новый заголовок файла, он ищет файл с таким именем, чтобы начать изменения.

После заголовка файла идет серия порций. Каждая порция начинается с заголовка, который определяет диапазон номеров строк в файле, которые изменяет данная порция. После заголовка, порция начинается и заканчивается через несколько (обычно три) не измененных строки текста из файла, которые называются контекстом для порции. Если имеется только небольшой разрыв между последовательными порциями, **diff** не печатает новый заголовок порции, он просто объединяет порции вместе, вставляя несколько строк из контекста между изменениями.

Каждая строка в контексте начинается с пробела. В порции строку, которая начинается с «`-`» означает «удалить эту строку», а строка, которая начинается с «`+`» означает «вставить эту строку» Например, изменившаяся строка представлена одним удалением и одной вставкой.

Мы вернемся к некоторым из наиболее тонких аспектов патчей позже (в разделе Раздел 12.6, «Более подробно о патчах»), но теперь вы должны иметь достаточно информации, для использования MQ прямо сейчас.

# 12.5. Начало работы с Mercurial Queues

Потому что MQ реализован в виде расширения, вы должны явно разрешить его, перед тем как использовать. (Вам не нужно ничего скачивать; MQ поставляется в стандартном пакете Mercurial.) Для того чтобы включить MQ, отредактируйте файл `~/.hgrc`, и добавьте следующие строки.

```
[extensions]
hgext.mq =
```

После включения расширения, будет доступен ряд новых команд. Чтобы убедиться, что расширение работает, вы можете использовать **hg help** для помощи по команде **qinit**.

```
$ hg help qinit
hg qinit [-c]

init a new queue repository (DEPRECATED)

    The queue repository is unversioned by default. If -c/--create-repo is
    specified, qinit will create a separate nested repository for patches
    (qinit -c may also be run later to convert an unversioned patch repository
    into a versioned one). You can use qcommit to commit changes to this queue
    repository.

    This command is deprecated. Without -c, it's implied by other relevant
    commands. With -c, use "hg init --mq" instead.

options:

 -c --create-repo create queue repository

use "hg -v help qinit" to show more info
```

Вы можете использовать MQ с любым Mercurial репозиторием, и его команды работают только с тем, что находится в хранилище. Чтобы начать работу, просто подготовьте репозиторий используя команду **qinit**.

```
$ hg init mq-sandbox
$ cd mq-sandbox
$ echo 'line 1' > file1
$ echo 'another line 1' > file2
$ hg add file1 file2
$ hg commit -m'first change'
$ hg qinit
```

Эта команда создает пустую папку с именем `.hg/patches`, где MQ будет хранить свои метаданных. Как и во многих командах Mercurial, команда **qinit** ничего не печатает, если всё проходит успешно.

## 12.5.1. Создание нового патча

Чтобы начать работу над новым патчем, используйте команду **qnew**. Эта команда имеет один аргумент, имя patch-а для создания.

MQ будет использовать в качестве фактических имена файлов в директории `.hg/patches`, как вы можете увидеть ниже.

```
$ hg tip
changeset:   0:bf7787dcd87d
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:01 2012 +0000
summary:     first change

$ hg qnew first.patch
$ hg tip
changeset:   1:322557d9d523
tag:         first.patch
tag:         qbase
tag:         qtip
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:01 2012 +0000
summary:     [mq]: first.patch

$ ls .hg/patches
first.patch  series  status
```

Кроме того, сейчас в директории `.hg/patches` есть два файла, `series` и `status`. Файл `series` содержит список всех исправлений, которые знает MQ для этого репозитория, с одним патчем в каждой строке. Mercurial использует файл `status` для внутренней бухгалтерии; он отслеживает все патчи, которые MQ применил в этом репозитории.

### Примечание

Вам может иногда понадобиться изменить файл `series` вручную, например, чтобы изменить последовательность, в которой применяются некоторые патчи. Тем не менее, вручную редактировать файл `status` почти всегда плохая идея, так как легко повредить данные MQ о происходящем.

После того как вы создали свой новый патч, вы можете редактировать файлы в рабочем каталоге, как вы это обычно делаете. Все нормальные команды mercurial, такие как **hg diff** и **hg annotate** работают так же, как раньше.

## 12.5.2. Обновление патча

Когда вы достигаете точки, где вы хотите сохранить свою работу, используйте команду **qrefresh** для обновления патча с которым вы работаете.

```
$ echo 'line 2' >> file1
$ hg diff
diff -r 322557d9d523 file1
--- a/file1 Thu Feb 02 14:10:01 2012 +0000
+++ b/file1 Thu Feb 02 14:10:02 2012 +0000
@@ -1,1 +1,2 @@
 line 1
+line 2
$ hg qrefresh
$ hg diff
$ hg tip --style=compact --patch
1[first.patch,qbase,qtip,tip]   919b9524323d   2012-02-02 14:10 +0000   bos
  [mq]: first.patch

diff -r bf7787dcd87d -r 919b9524323d file1
--- a/file1 Thu Feb 02 14:10:01 2012 +0000
+++ b/file1 Thu Feb 02 14:10:02 2012 +0000
@@ -1,1 +1,2 @@
 line 1
+line 2
```

Эта команда помещает сделанные вами изменения в рабочем каталоге на ваш патч, и обновляет свою соответствующую ревизию которая содержит эти изменения.

Вы можете запускать **qrefresh** так часто, как вам нравится, так что это хороший способ сохранять «контрольные точки» вашей работы. Обновляйте патч в подходящее время, попробуйте экспериментировать, и если эксперимент не получится, **hg revert** откатит ваши изменения обратно до последнего обновления.

```
$ echo 'line 3' >> file1
$ hg status
M file1
$ hg qrefresh
$ hg tip --style=compact --patch
1[first.patch,qbase,qtip,tip]   52117437a1b0   2012-02-02 14:10 +0000   bos
  [mq]: first.patch

diff -r bf7787dcd87d -r 52117437a1b0 file1
--- a/file1 Thu Feb 02 14:10:01 2012 +0000
+++ b/file1 Thu Feb 02 14:10:02 2012 +0000
@@ -1,1 +1,3 @@
 line 1
+line 2
+line 3
```

## 12.5.3. Укладка и отслеживания патчей

После того как вы закончили работу над патчем, или нужно работать с другим, вы можете использовать команду **qnew** еще раз для создания новых патчей. Mercurial сделает патч верхним из ваших существующих патчей.

```
$ hg qnew second.patch
```

```
$ hg log --style=compact --limit=2
2[qtip,second.patch,tip]   d535b10fd01c   2012-02-02 14:10 +0000   bos
  [mq]: second.patch

1[first.patch,qbase]   52117437a1b0   2012-02-02 14:10 +0000   bos
  [mq]: first.patch

$ echo 'line 4' >> file1
$ hg qrefresh
$ hg tip --style=compact --patch
2[qtip,second.patch,tip]   e933fc1784a2   2012-02-02 14:10 +0000   bos
  [mq]: second.patch

diff -r 52117437a1b0 -r e933fc1784a2 file1
--- a/file1 Thu Feb 02 14:10:02 2012 +0000
+++ b/file1 Thu Feb 02 14:10:03 2012 +0000
@@ -1,3 +1,4 @@
 line 1
 line 2
 line 3
+line 4

$ hg annotate file1
0: line 1
1: line 2
1: line 3
2: line 4
```

Обратите внимание, что патч содержит изменения в нашем предыдущем патче в качестве части своей контекста (вы можете увидеть это более четко в выводе **hg annotate**).

Пока, за исключением **qnew** и **qrefresh**, мы осторожно использовали только обычные команды Mercurial. Тем не менее, MQ предоставляет множество команд, которые проще использовать, когда вы думаете о патчах, как показано на рисунке ниже.

```
$ hg qseries
first.patch
second.patch
$ hg qapplied
first.patch
second.patch
```

- Команда **qseries** перечисляет все патчи, о которых знает MQ в этом репозитории, от старых к новым (созданным совсем недавно).

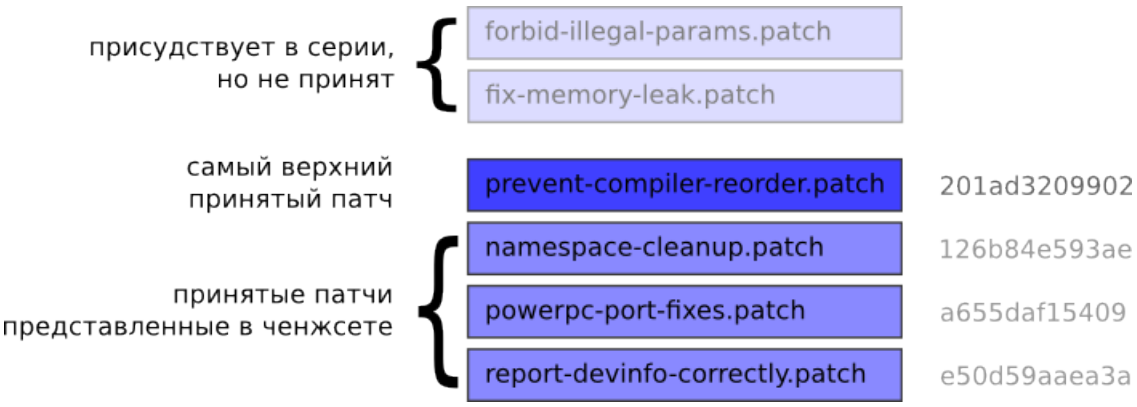
- Команда **qapplied** перечисляет все исправления, которые MQ применил в этом репозитории, опять от старых к новым.

## 12.5.4. Манипуляция стеком патчей

Предыдущее обсуждение предполагает, что должна быть разница между «знанием» и «применением» патчей и она есть. MQ может управлять патчами применёнными без него к репозиторию.

Применять патч соответствующей ревизии в репозитории, и последствия патча и ревизия видны в рабочей директории. Вы можете отменить применение патча использованием команды **qpop**. MQ до сих пор знает о, или управляет, извлеченными патчами, но патч уже не имеет соответствующего набора изменений в репозитории, и рабочий каталог не содержит изменения сделанные патчем. Рисунок 12.1, «Применение и отмена патчей в стеке патчей MQ» иллюстрирует разницу между применёнными и отслеживаемыми патчами.

Рисунок **12.1.** Применение и отмена патчей в стеке патчей **MQ**




Вы можете повторно применить отменённый или извлечённый патч используя команду **qpush**. Она создает новую ревизию соответствующую патчу и изменения патча в очередной раз станут присутствовать в рабочей директории. Ниже приведены примеры **qpop** и **qpush** в действии.

```
$ hg qapplied
first.patch
second.patch
$ hg qpop
popping second.patch
now at: first.patch
$ hg qseries
first.patch
second.patch
$ hg qapplied
first.patch
$ cat file1
line 1
line 2
line 3
```

Обратите внимание, что как только мы извлекли патч или два патча, вывод **qseries** остается неизменным, в то время как у **qapplied** изменился.

## 12.5.5. Вставка и извлечение нескольких патчей

Хотя по умолчанию **qpush** и **qpop** работают над одним патчем за раз, вы можете вставить и извлечь много патчей в один проход. Опция `-a` команды **qpush** приводит к вставке всех неприменённых патчей, а опция `-a` для **qpop** приводит к извлечению всех применённых патчей. (Для некоторых других способов вставки и извлечения многих патчей, смотрите раздел Раздел 12.8, «Получение максимальной производительности от MQ» ниже).

```
$ hg qpush -a
applying second.patch
now at: second.patch
$ cat file1
line 1
line 2
line 3
line 4
```

## 12.5.6. Безопасные проверки и их основа

Некоторые команды MQ проверяют рабочий каталог, прежде чем что-то делать, и не работают, если они находят какие-либо изменения. Они делают это, чтобы вы не потеряли изменения, которые вы уже сделали, но еще не включили в патч. Пример ниже иллюстрирует это, команда **qnew** не будет создавать новый патч, если есть изменения, вызванных в этом случае **hg add** для `file3`.

```
$ echo 'file 3, line 1' >> file3
$ hg qnew add-file3.patch
$ hg qnew -f add-file3.patch
```

```
abort: patch "add-file3.patch" already exists
```

Команды, которые проверяют рабочий каталог, все принимают опцию «Я знаю, что я делаю», которая всегда имеет имя `-f`. Точное значение `-f` зависит от команды. Например, в **hg qnew `-f`** будет включать любые не сохранённые изменения в новый патч который она создает, но **hg qpop `-f`** отменит изменения во всех файлах, измененные в результате применения патчей, которые она извлекает. Не забудьте прочитать документацию по опции -f для нужной вам команды, прежде чем использовать её!

## 12.5.7. Работа с различными патчами сразу

Команда **qrefresh** всегда обновляет верхний применяемый патч. Это означает, что вы можете приостановить работу по одному патчу (обновив его), извлечь или поместить другой патч наверх, и работать с тем патчем некоторое время.

Вот пример, который показывает, как можно использовать эту способность. Скажем вы разрабатываете новую функцию, как два патча. Первым из них является изменение ядра вашего программного обеспечения, а второй — слой поверх первого — изменения пользовательского интерфейса, использующее код, который вы только что добавили к ядру. Если вы заметили ошибку в ядре в то время как вы работаете на исправлением UI, вы можете легко исправить ядро. Просто вызовите **qrefresh** для патча пользовательского интерфейса, чтобы сохранить незавершенные изменения и **qpop** вплоть до патча ядра. Исправьте ошибку ядра, запустите **qrefresh** для патча ядра и **qpush** поднявшись к патчу пользовательского интерфейса и работайте с того места, где вы остановились.

# 12.6. Более подробно о патчах

MQ использует команду GNU **patch** для применения патчей, так что полезно узнать некоторые более детальные аспекты, как работает **patch**, и о самих патчах.

## 12.6.1. The strip count

Если вы посмотрите на заголовок файла патча, вы заметите, что путь к файлу, как правило содержит дополнительные компоненты в начале, которых нет в настоящем пути. Это остаток пути который люди использовали для создания патчей (люди до сих пор это делают, но уже реже с появлением современных средств контроля версий).

Алиса распаковала архив, изменила свои файлы, а затем решила, что она хочет создать патч. Она переименовывает свой рабочий каталог, распаковывает архив еще раз (это и обусловило необходимость переименования), и использует опции `-r` и `-N` команды **diff** для рекурсивно создания патча между не измененным каталогом и измененным. Результатом будет то, что имя не измененного каталога будет в начале пути каждого файла в заголовке иметь имя левого каталога, а путь каталога измененного будет в начале пути иметь правый каталог.

Кто-то получает патч от Алисы, вряд ли будет иметь неизмененный и измененый каталоги с точно такими же именами, команда **patch** имеет опцию `-p` указывающую на количество ведущих компонентов пути удаляют при применении патча. Это число называется счётчик удаления.

Опция «`-p1`» означает «использовать счётчик удаления с первого элемента». Если **patch** видит имя файл `foo/bar/baz` в заголовке файла, он удаляет `foo` и попытаться применить патч к файлу с именем `bar/baz`. (Строго говоря, счётчик удаления относится к количеству разделителей пути (и компонентов, которые следуют за ними). Счётчик удаления в значении 1 преобразует `foo/bar` в `bar`, но `/foo/bar` (обратите внимание на дополнительный ведущий слэш ) в `foo/bar`.)

«Стандартный» счётчик удаления для патчей равен 1, почти все патчи содержат один ведущий компонент пути, который необходимо отрезать. Команда Mercurial **hg diff** генерирует путь в этой форме, и команда **hg import** и MQ ожидают патчей со счётчиком удаления равным 1.

Если вы получили патч от кого-то, и вы хотите добавить патч в свою очередь, и патч требует другого счётчика удаления, чем 1, вы не можете просто применить **qimport** к патчу, потому что **qimport** еще не имеет опции `-p` (смотрите заявку issue 311 [http://www.selenic.com/mercurial/bts/issue311]) . Лучше всего, чтобы запустить **qnew** для создания собственного патча, а затем использовать **patch -pN**, чтобы применить их исправления, а затем запустить **hg addremove** чтобу узнать какие файлы добавлены или удалены патчем, а затем выполнить **hg qrefresh**. Эта сложность может стать ненужной, смотрите issue 311 [http://www.selenic.com/mercurial/bts/issue311] для деталей.

## 12.6.2. Стратегия для применения патчей

Когда **patch** применяет блок, он пытается использовать последовательно несколько менее точные стратегии, чтобы попытаться применить блок. Этот метод уменьшения точности часто дает возможность применить патч, который был создан на старой версии файла, и применить его на новой версии этого файла.

Во-первых, **patch** ищет точное совпадение, где номера строки, контекст, и текст, который будет изменен совпадает точно. Если он не может сделать точное соответствие, то он пытается найти точное соответствие для контекста, без информации о номерах строк. Если это удаётся, он сообщает что блок был применен, но имеет смещение от первоначального номера строки.

Если поиск по контексту не удается, **patch** устраняет первую и последнюю строку контекста, и попытается найти сокращённый контекст. Если блок с ограниченным контекстом успешно применится, он выводит сообщение о том, что он применяет блок с фактором промаха (число после фактора промаха это коэффициент показывающий, сколько строк из контекста обрезано в патче, чтобы патч применился).

Если ни один из этих методов сработал, **patch** печатает сообщение о том, что блок отклонен. Он сохраняет отклоненные блоки (называя просто «rejects») в файле с тем же именем и расширением `.rej`. Он также сохраняет не измененную копию файла с расширением `.orig`; копия файла без расширения будет содержать изменения, из применённых блоков, которые применились чисто. Если у вас есть патч, который изменяет `foo` с 6 блоками, а один из них не применяются, вы получите: неизмененный `foo.orig`, `foo.rej` содержащие один блок, и `foo`, содержащий изменения, внесенные в пяти успешных блоках

## 12.6.3. Некоторые причуды из представления патчей

Есть несколько полезных фактов о том, как **patch** работает с файлами.

- Это уже должно быть очевидно, но **patch** не может справиться с бинарными файлами.
- Он также не заботится о бите исполняемости, создает новые файлы читаемыми, но не исполняемыми.
- **patch** считает удалённый файл, как различие между фалом который удалён и пустым файлом. Так что ваша идея «Я удалил этот файл» в патче выглядят как «каждая строка этого файла была удалена».
- Это относится и к добавленным файлам как к различию между пустым файлом и файлом который будет добавлен. Таким образом, в патче, ваша идея «я добавил этот файл» выглядит как «каждая строка этого файла был добавлена».
- Это относится и к переименованным файлам, как удаление старого файла, и добавление нового. Это означает, что переименованные файлы будут иметь большой след в блоке. (Заметим также, что mercurial в настоящее время не попытаться сделать вывод, когда файлы были переименованы или скопированы и исправлены.)
- **patch** не может представлять пустые файлы, так что вы не можете использовать патч для представления понятия «Я добавила пустой файл в дерево».

## 12.6.4. Остерегайтесь неточностей

Применение блоков со смещением, или фактором неточности, часто будет полностью успешным, такие неточные методы естественно оставляют открытой возможность повреждать исправленный файл. Большинство случаев обычно связаны с применением патча два раза, или на неверное место расположения файла. Если **patch** или **qpush** постоянно упоминает о смещении или факторе неточности, вы должны убедиться, что измененные файлы правильны.

Часто хорошая идея, обновлять патч, который применился со смещением или фактором неточности, обновление патча порождает новые контекстные связи, которые помогут применить патч чисто. Я говорю: «часто», а не «всегда», потому что иногда обновление патча сделает его не применяемым для различных ревизий основных файлов. В некоторых случаях, например, когда вы поддерживаете патч, который должен находится на вершине нескольких версий исходного дерева, это приемлемо иметь патч применяемый с некоторыми неточностями, если вы убедились в правильности результатов применения патча в таких случаях.

### 12.6.5. Обработка отказа

Если **qpush** не удается применить патч, он выведет сообщение об ошибке и завершит работу. Если он оставит `.rej` файлы, как правило, лучше устранить отклонения блоков перед добавлением других патчей или какой-либо дальнейшей работой.

Если ваш патч применялся чисто, и больше не может, потому что вы изменили исходный код, на котором основан ваши патч, Mercurial Queues может помочь, смотрите в разделе Раздел 12.9, «Обновление патчей когда исходный код измененился» для подробностей.

К сожалению, не так много методов для решения отклоненных блоков. Чаще всего, вам понадобится просмотреть `.rej` файл и отредактировать целевой файл, применяя отклоненный блок вручную.

Разработчик ядра Linux, Крис Мейсон (автор Mercurial Queues), пишет инструмент под названием **mpatch** ((http://oss.oracle.com/~mason/mpatch/), который принимает простой подход к автоматизации применения блоков отклонённых **patch**. Команда **mpatch** может помочь с 4-я распространенными причинами отклонения блока:

- Контекст, в середине блока изменился.
- Часть контекста блока отсутствуют в начале или в конце.
- Большой блок может применится лучше полностью или частично, если его разбить на мелкие куски.
- Блок удаляет строки с несколько иным содержанием, чем в настоящее время в файле.

Если вы используете **mpatch**, вы должны быть особенно осторожным, и проверять свои результаты, когда закончите. В самом деле, **mpatch** применяет метод двойной проверки вывода инструмента, автоматически передавая вас программе объединения, когда он сделает свою работу, так что вы можете проверить свою работу и закончить всё оставшееся слияние.

## 12.7. Подробнее о управление патчами

По мере того как вы познаёте MQ, вы пожелаете выполнять другие виды операций по управлению патчами.

### 12.7.1. Удаление нежелательных патчей

Если вы хотите, избавиться от патча, воспользуйтесь командой **hg qdelete**, чтобы удалить файл патча и удалить его из серии патчей. Если вы попытаетесь удалить патч, который все еще применяется, **hg qdelete** откажется.

```
$ hg init myrepo
$ cd myrepo
$ hg qinit
$ hg qnew bad.patch
$ echo a > a
$ hg add a
$ hg qrefresh
$ hg qdelete bad.patch
abort: cannot delete applied patch bad.patch
$ hg qpop
popping bad.patch
patch queue now empty
$ hg qdelete bad.patch
```

### 12.7.2. Преобразование в и из постоянных ревизий

Как только вы закончите работу над патчем и хотите превратить его в постоянную ревизию, используйте команду **hg qfinish**. При переходе к ревизии команда определяет патч, который вы хотите превратить в обычную ревизию, этот патч уже должен быть применен.

```
$ hg qnew good.patch
$ echo a > a
$ hg add a
```

```
$ hg qrefresh -m 'Good change'
$ hg qfinish tip
$ hg qapplied
$ hg tip --style=compact
0[tip]   ea28b7ac83fb   2012-02-02 14:09 +0000   bos
  Good change
```

Команда **hg qfinish** принимает опцию `--all` или `-a`, который преобразует все применённые патчи в обычные ревизии.

Кроме того, можно превратить существующую ревизию в патч, с помощью опции `-r` команды **hg qimport**.

```
$ hg qimport -r tip
$ hg qapplied
0.diff
```

Обратите внимание, что это имеет смысл только для преобразования набора изменений в патч, если вы не распространяете ревизии в любой другой репозитарий. ID импортируемого набора изменений будет изменяться каждый раз, когда вы обновляете патч, который делает Mercurial рассматривать его как не связанный с первоначальной ревизией, если вы его передадите в любой другой репозитарий.

# 12.8. Получение максимальной производительности от MQ

MQ очень эффективен при обработке большого количества исправлений. Я провел несколько экспериментов производительности в середине 2006 года для доклада, который я делал в 2006 на конференции EuroPython (на современном оборудовании, вы должны ожидать более высокую производительность, чем вы увидите ниже). Я использовал в качестве моего набора данных серии патчей linux 2.6.17-mm1, которая состоит из 1738 патчей. Я применил их на поверх репозитория linux ядра, содержащее всего 27472 ревизии между linux 2.6.12-rc2 и linux 2.6.17.

На моём старом, медленном ноутбуке, я выполнил **hg qpush `-a`** для всех 1738 патчей в 3,5 минуты, а **hg qpop `-a`** всего в 30 секунд. (На новом ноутбуке, время добавления всех патчей снизилось до 2 минут). Я выполнил **qrefresh** для одного большого патча (который состоял из 22779 строк изменений в 287 файлах) за 6,6 секунды.

Очевидно, что MQ хорошо приспособлен к эксплуатации на больших деревьях, но есть несколько трюков которые можно использовать для получения наилучшей производительности.

Прежде всего, попытайтесь выполнять «пакетные» операции вместе. Каждый раз, когда вы запускаете **qpush** или **qpop**, эти команды проверяют рабочую директорию один раз, чтобы убедиться, что вы не внесли каких нибудь изменений, а затем забыл запустить **qrefresh.** На небольшом дереве, время этой проверки невелико. Однако, на средних деревьях (содержащих десятки тысяч файлов), это может занять секунду или даже больше.

Команды **qpush** и **qpop** позволяют вставлять и извлекать несколько патчей сразу. Вы можете указать «патч назначения» который вы хотите применить последним. **qpush** с указанным назначением, то он будет вставлять патчи до этого патча в верх стека применения. **qpop** с назначением, MQ извлекает патчи до того как патч назначения не окажется на самом верху.

Вы можете определить патч назначения, используя либо название патча, или номер. Если вы используете числовую адресацию, патчи начинаются с нуля, это означает, что первый патч равен нулю, второй единице, и так далее.

# 12.9. Обновление патчей когда исходный код измененился

Как правило имея патч на верху стека вы не изменяете напрямую нижележащий код в репозитории. Если вы работаете над изменениями в коде третьих сторон, или над особенностью, которая занимает больше времени, чем скорость развития изменений кода внизу, вам часто нужно синхронизироваться с исходным кодом, и исправлять блоки в своих патчах, которые уже не применяются. Это называется перебазирование серии патчей.

Проще всего это сделать так: выполнить **hg qpop** `hg -a` для ваших патчей, затем **hg pull** ревизий из основного репозитория, и, наконец выполнить **hg qpush** `-a` ваших патчей снова. MQ будет останавливать вставку каждый раз, когда проходит через патч, который не применяется из-за конфликтов, что позволяет исправлять ваши конфликты, и выполнять **qrefresh** для пострадавших патчей, и продолжать вставку, пока не исправите весь стек.

Этот подход прост в использовании и работает хорошо, если вы не ожидаете изменений в исходный код, которые повлияют на эффективность ваших патчей. Если ваш стек патчей касается кода, который модифицируются часто или агрессивно в базовом репозитории, однако, исправление отклонённых блоков вручную быстро становится скучным.

Можно частично автоматизировать процесс перебазирования. Если ваши патчи применяются чисто к другим ревизиям из репозитория, MQ может использовать эту информацию, чтобы помочь вам в разрешении конфликтов между патчами и различными ревизиями.

Этот процесс немного сложнее.

1. Во-первых, выполняем **hg qpush -a** применяя все ваши патча к старшей ревизии, к которой вы знаете, что они применяются чисто.

2. Сохранить резервную копию исправляемого каталога с помощью команды **hg qsave** `hg -e hg -c`. Она выводит имя директории, в которой она сохранит патчи. Это сохранит исправления в каталоге называемом `.hg/patches.N`, где `N` является малым числом. Она также «сохраняет ревизию» поверх которой применяются патчи, и записывает состояние файлов `series` и `status`.

3. Используйте **hg pull** для внесения изменений в основной репозиторий. (Не запускать **hg pull -u**, смотрите ниже, почему.)

4. Обновитесь до новой верхней ревизии, используя **hg update** `-C`, переписываяизменения ваших патчей.

5. Объедините все патчи **hg qpush -m -a**. Опция `-m` **qpush** говорит MQ выполнить трехстороннее слияние, если патч не может применится.

В **hg qpush** `hg -m`, каждый патч в файле `series` применяется в обычном режиме. Если исправление применяется с неточностями или отвергается, MQ смотрит на вашу очередь **qsave**d, и также выполняет трехстороннее слияние с соответствующей ревизией. Для слияния Mercurial использует нормальный механизм слияния, поэтому оно может быть передано GUI-инструменту слияния, чтобы помочь вам в решении проблем.

Когда вы закончите разрешение эффектов патча, MQ обновит ваш патч, основанный на результате слияния.

В конце этого процесса, репозиторий будет иметь одну дополнительную голову от старой очереди патчей, а копия старой очереди патч будет лежать в `.hg/patches.N`. Вы можете удалить дополнительную голову с помощью **hg qpop -a -n patches.N** или **hg strip**. Вы можете удалить `.hg/patches.N` когда вы уверены, что вам она больше не нужна в качестве резервной копии.

# 12.10. Идентификация патчей

Команды MQ, которые работают с патчами позволяют ссылаться на патч либо используя его имя или номер. По имени достаточно очевидно; передать имя `foo.patch` команде **qpush**, например, и она будет вставлять и применять патчи до `foo.patch`.

Для сокращения, вы можете обратиться к патчу с использованием имени и цифрового смещения; `foo.patch-2` означает «второй патч перед `foo.patch`», а `bar.patch+4` означает «четвёртый патч после `bar.patch`».

Ссылка на патч по индексу не сильно отличается. Первый патч напечатаны в выходе **qseries** это нулевой патч нулю (да, это один из тех систем отсчёта, которые начинаются с нуля); второй патч 1, и так далее.

MQ также позволяет легко работать с патчами, как вы используете нормальные команды Mercurial команд. Каждая команда, которая принимает id ревизии также примет название применяемого патч. MQ увеличивает количество тегов в репозитории, как правило по одному для каждого применённого патча. Кроме того, специальные теги `qbase` и `qtip` определяют «самый нижний» и «самый верхний» применённые патчи, соответственно.

Эти дополнительное тегирование нормальными тегами Mercurial делает возможность внесения патчей еще более легким.

- Хотите отправить по email patchbomb с вашей последней серией патчей?

```
hg email qbase:qtip
```

  (Не знаете, что такое «patchbombing»? Смотрите раздел Раздел 14.4, «Отправить изменений по электронной почте с расширением `patchbomb`».)

- Нужно, увидеть все патчи начиная с `foo.patch` которые коснулись файлов в подкаталоге вашего дерева?

```
hg log -r foo.patch:qtip subdir
```

Потому что MQ делает имена патчей доступными для остальных команд Mercurial через обычный внутренний механизм тегов, вам не нужно вводить полное имя файла с патчем, если вы хотите идентифицировать его по имени.

Другим полезным следствием представления патчей именами тегов является то, что когда вы выполните команду **hg log**, она будет отображать название патча, как тег, просто как часть нормального вывода. Это позволяет визуально различать применённые патчи от лежащие в основе «обычных» ревизий. В следующем примере показано несколько простых команд Mercurial использующихся с применёнными патчами.

```
$ hg qapplied
first.patch
second.patch
$ hg log -r qbase:qtip
changeset:   1:8bd990ad127b
tag:         first.patch
tag:         qbase
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:09:58 2012 +0000
summary:     [mq]: first.patch

changeset:   2:ce57e98425d7
tag:         qtip
tag:         second.patch
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:09:59 2012 +0000
summary:     [mq]: second.patch

$ hg export second.patch
# HG changeset patch
# User Bryan O'Sullivan <bos@serpentine.com>
# Date 1328191799 0
# Node ID ce57e98425d79f974ee096cc6fdf4cc5e4b55918
# Parent  8bd990ad127bfbc3153a923091f86021aa7a7818
[mq]: second.patch

diff -r 8bd990ad127b -r ce57e98425d7 other.c
--- /dev/null Thu Jan 01 00:00:00 1970 +0000
+++ b/other.c Thu Feb 02 14:09:59 2012 +0000
@@ -0,0 +1,1 @@
+double u;
```

# 12.11. Полезные вещи, которые необходимо знать

Есть ряд аспектов использования MQ, которые не вписываются аккуратно в отдельные разделы, но их хорошо бы знать. Вот они, в одном месте.

- Обычно, когда вы применяете к патчу **qpop** и потом **qpush**, ревизия представляющая патч после извлечения/вставки будет иметь другой идентификатор, чем ревизия, которые представляла патч ранее. Смотрите раздел Раздел B.1.14, «**qpush** — вставляет патчи в стек» для информации о том, почему так происходит.

- Это не очень хорошая идея, использовать **hg merge** изменений из другой ветви с ревизией патча, по крайней мере, если вы хотите сохранить «патченость» этой ревизии и ревизий ниже её в стеке патчей. Если вы попытаетесь сделать это, всё пройдёт успешно, но MQ будет запутан.

# 12.12. Управление патчами в репозитории

Так как каталог `.hg/patches` находится вне рабочего каталога репозитория Mercurial, «основной» Mercurial репозиторий ничего не знает об управлении или наличие патчей.

Это открывает интересные возможности управления содержанием каталога патчей как отдельным Mercurial репозиторием внутри собственного репозитория. Это может быть полезным для работы. Например, вы можете работать над патчем некоторое время, выполнить **qrefresh**, потом **hg commit** для текущего состояния патча. Это позволяет вам «откатываться» на этоу версию патча позже.

Вы можете поделиться различными версиями одного и того же стека патчей между несколькими обычными репозиториями. Я использую это, когда я занимаюсь разработкой фишек ядра linux. У меня есть чистый экземпляр с исходными текстами ядра для нескольких архитектур процессоров, а также клонированные реозитории каждый, из который содержит патчи над, которыми я работаю. Когда я хочу проверить изменения на различных архитектурах, я вставляю свои текущие патчи для исправления м храни ревизий, связанные с этим деревом ядра, извлечение и вставка всех моих патчей сборкой и тестированием

Управление патчами в репозитории позволяет нескольким разработчикам работать над одной и той же серией патчей не сталкиваясь друг с другом, всегд поверх основной части базы исходников, которые они могут или не могут контролировать.

## 12.12.1. Поддержка MQ для репозитория патчей

MQ позволяет работать с директорией `.hg/patches` как с репозиторием, когда вы готовите репозиторий для работы с патчами используя **qinit**, вы можете передать опцию `hg -c` для создания каталога `.hg/patches` как репозиторий Mercurial.

### Примечание

Если вы забыли использовать опцию `hg -c`, вы можете просто пойти в каталог `.hg/patches` в любое время и выполнить **hg init**. Не забудьте добавить запись в `.hgignore` о файле `status`, хотя (**hg qinit** `hg -c` делает это за вас автоматически); Вы ведь действительно не хотите, отслеживать файл `status`.

Для удобства, если MQ замечает, что каталог `.hg/patches` репозиторий, он будет автоматически выполнять **hg add** каждый патч, который вы создаете или импортируете.

MQ обеспечивает короткую команду **qcommit**, запускающую **hg commit** в директории `.hg/patches`.

Наконец, для удобства управления директорией патчей, можно определить alias **mq** на системах Unix. Например, в системах linux с помощью оболочки **bash**, вы можете включить следующий фрагмент в файле `~/.bashrc`.

```
alias mq=`hg -R $(hg root)/.hg/patches'
```

После этого можно отдавать команды вида **mq pull** из главного репозитория.

## 12.12.2. Несколько вещей для отслеживания

MQ поддерживает для работы с репозиторием полные патчи ограниченного небольшими отношениями.

MQ не умеет автоматически обнаруживать изменения, внесенные в каталог патчей. Если вы выполнили **hg pull**, вручную отредактировали, или сделали **hg update** изменений для патчей или файлом `series`, вам придется выполнить **hg qpop** `-a` and then **hg qpush** `-a` в базовом хранилище, чтобы увидеть эти изменения там. Если вы забудете это сделать, вы можете спутать MQ планы применения патчей.

# 12.13. Инструменты сторонних разработчиков для работы с патчами

Когда вы поработаете с патчами некоторое время, вы обнаружите недостаток инструментов, которые помогут вам понять и управлять патчами, с которыми вы имеете дело.

Команда **diffstat** [web:diffstat] генерирует гистограммы изменений, внесенных с каждым файлом в патч. Она обеспечивает хороший способ «получить смысл» патча — какие файлы он затрагивает, и сколько изменений применит к каждому файлу и в целом. (Я считаю, что это хорошая идея использовать опцию `-p` **diffstat** для отображения статистики различий как нечто само собой разумеющееся, так как иначе он будет пытаться делать умные вещи с префиксами имен файлов, которые неизбежно запутают по крайней мере меня).

```
$ diffstat -p1 remove-redundant-null-checks.patch
bash: diffstat: command not found
$ filterdiff -i '*/video/*' remove-redundant-null-checks.patch
bash: filterdiff: command not found
```

Пакет `patchutils` [web:patchutils] имеет неоценимое значение. Он предоставляет набор небольших утилит, которые следуют «Unix философии» каждая делает одну полезную вещь с патчем. Из команд `patchutils` я использую больше всего **filterdiff**, которая извлекает из подмножеств файла патча. Например, если патч, который изменяет сотни файлов в десятках каталогов, один вызов **filterdiff** может генерировать меньший патч, который затрагивает только файлы, имена которых совпадают с неким шаблоном. Смотрите раздел Раздел 13.9.2, «Просмотр истории патча» для других примеров.

# 12.14. Хорошие методы работы с патчами

Если вы работаете над серией патчей представляя их свободному программному обеспечению или проекту с открытым кодом, или серии, которую вы намерены рассматривать как последовательность регулярных ревизий которую вы закончили, вы можете использовать некоторые простые методы, чтобы сохранить вашу работу организованной.

Давайте вашим патчам описательные имена. Хорошее имя для патча может быть `rework-device-alloc.patch`, потому что оно будет сразу же давать вам подсказку на цель исправления. Длинные имена не должны быть проблемой, вы не будете вводить имена часто, а будете работать с такими командами, как **qapplied** и **qtop** снова и снова. Хорошее имя становится особенно важным, когда у вас есть несколько патчей над которыми вы работаете, или если вы жонглировали целым рядом различных задач, и ваши патчи только получили только часть вашего внимания.

Помните о том, над каким патч вы работаете. Используйте команду **qtop** и просматривайте текст ваших патчей чаще, например, с использованием **hg tip** `-p` — что быть уверенным в том, где вы находитесь. Я несколько раз работал, и запускал **qrefresh** не для того патча, которого хотел, и часто сложно перенести изменения в правильный патч после внесения их в неверный.

По этой причине, стоит потратить немного времени, чтобы узнать, как использовать некоторые из утилит сторонних разработчиков, которые я описал в разделе Раздел 12.13, «Инструменты сторонних разработчиков для работы с патчами», в частности, **diffstat** и **filterdiff**. Первый даст вам представление о том, какие изменения патч делает, а второй позволяет легко перемещать выбраные блоки из одного патча и в другой.

# 12.15. Поваренная книга MQ

## 12.15.1. Управление «тривиальными» патчами

Так как накладные расходы на добавление файлов в новый репозиторий Mercurial настолько низки, есть смысл, управлять патчами его путём, даже если вы просто хотите внести некоторые изменения в архив с исходными кодами, который вы скачивали.

Начните с загрузки и распаковки архива исходных кодов, и превратите его в репозиторий Mercurial.

```
$ download netplug-1.2.5.tar.bz2
$ tar jxf netplug-1.2.5.tar.bz2
$ cd netplug-1.2.5
$ hg init
$ hg commit -q --addremove --message netplug-1.2.5
$ cd ..
$ hg clone netplug-1.2.5 netplug
updating to branch default
18 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Продолжим создавать стек патчей и делать изменения

```
$ cd netplug
$ hg qinit
$ hg qnew -m 'fix build problem with gcc 4' build-fix.patch
$ perl -pi -e 's/int addr_len/socklen_t addr_len/' netlink.c
$ hg qrefresh
$ hg tip -p
changeset:   1:b51e7b2c8901
tag:         build-fix.patch
tag:         qbase
tag:         qtip
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Thu Feb 02 14:10:00 2012 +0000
summary:     fix build problem with gcc 4

diff -r fddb0fa5f203 -r b51e7b2c8901 netlink.c
--- a/netlink.c Thu Feb 02 14:09:59 2012 +0000
+++ b/netlink.c Thu Feb 02 14:10:00 2012 +0000
@@ -275,7 +275,7 @@
         exit(1);
     }

-    int addr_len = sizeof(addr);
+    socklen_t addr_len = sizeof(addr);

     if (getsockname(fd, (struct sockaddr *) &addr, &addr_len) == -1) {
         do_log(LOG_ERR, "Could not get socket details: %m");
```

Давайте предположим, что прошло несколько недель или месяцев, и ваш автор вашего пакета сообщит о выходе новой версии. Во-первых, вносим эти изменения в репозиторий.

```
$ hg qpop -a
popping build-fix.patch
patch queue now empty
$ cd ..
$ download netplug-1.2.8.tar.bz2
$ hg clone netplug-1.2.5 netplug-1.2.8
updating to branch default
18 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd netplug-1.2.8
$ hg locate -0 | xargs -0 rm
$ cd ..
$ tar jxf netplug-1.2.8.tar.bz2
$ cd netplug-1.2.8
$ hg commit --addremove --message netplug-1.2.8
```

С помощью конвеера, начинающегося с **hg locate** выше удаляем все файлы в рабочем каталоге, так что команда **hg commit** с опцией `--addremove` может сообщить, какие файлы были действительно удалены в новой версии исходников.

Наконец, применяем ваши патчи поверх нового дерева

```
$ cd ../netplug
$ hg pull ../netplug-1.2.8
pulling from ../netplug-1.2.8
searching for changes
adding changesets
adding manifests
```

```
adding file changes
added 1 changesets with 12 changes to 12 files
(run 'hg update' to get a working copy)
$ hg qpush -a
(working directory not at a head)
applying build-fix.patch
now at: build-fix.patch
```

## 12.15.2. Объединение целых патчей

MQ обеспечивает команду **qfold**, которая позволяет объединить целые патчи. Она «складывает» патчи имена которых вы указали, в порядке указанном вами, наверх применяемого патча, и объединяет их описания в конечное описание. Патчи должны быть неприменёнными, перед тем как вы их сложите.

Порядок, в котором вы складываете патчи важен. Если верхним применяется патч `foo`, и вы складываете **qfold** `bar` и `quux` в него, вы в конечном итоге с помощью патча, получите тот же эффект, как при применении сначала `foo`, потом `bar`, а затем `quux`.

## 12.15.3. Слияние части одного патча с другим

Слияние частей патча в другой сложнее, чем слияние целых патчей.

Если вы хотите переместить изменения некоторых файлов, вы можете использовать опции `-i` и `-x` команды **filterdiff** для выбора изменений из одного патча, присоедините ее выход к концу патча, в который хотите их влить. Вам, как правило, не требуется вносить изменения в патч измененный объединением. Вместо этого, MQ сообщит что некоторые блоки отвергнуты, когда вы попытаетесь сделать **qpush** (те блоки которые вы переместили в другой патч), и вы можете просто сделать **qrefresh** для удаления из патча дубликатов блоков.

Если у вас есть патч, который имеет несколько кусков изменяющих файл, и вы только хотите перенести часть из них, работа становится более грязным делом, но все равно можно частично автоматизировать его. Использование **lsdiff -nvv** печатает различные метаданные о патче.

```
$ lsdiff -nvv remove-redundant-null-checks.patch
bash: lsdiff: command not found
```

Это команда выводит три различных номера:

- (В первой колонке) номер файла для идентификации каждого файла модифицируемого в патче;
- (На следующей строке с отступом) номер строки в рамках изменяемого файла, где начинается блок, и
- (на той же линии) номер блока идентифицирующего блок.

Вам придется использовать некоторый визуальный осмотр, и чтение патча, чтобы идентифицировать файл и блок с номерами которые вы хотите, но вы можете передать их в **filterdiff** используя опции `--files` и `--hunks`, чтобы выбрать именно тот файл и блок который вы хотите извлечь.

После этого вы можете вставить этот блок в конец патч назначения и продолжать так как описано в разделе Раздел 12.15.2, «Объединение целых патчей».

# 12.16. Различия между quilt и MQ

Если вы уже знакомы с quilt, MQ обеспечивает аналогичный набор команд. Есть некоторые различия в том, как они работают.

Вы уже заметили, что для большинства команд quilt есть аналоги в MQ, только начинаются они с «`q`». Исключения в quilt команды `add` и `remove`, аналоги для них есть в обычном Mercurial команды **hg add** и **hg remove**. Нет эквивалента в MQ команды quilt `edit`.

# Глава 13. Расширенное использование Mercurial Queues

Хотя это легко просто использовать Mercurial Queues, используя минимум возможностей и некоторые менее часто используемые возможности MQ для работы в сложных условиях разработки.

В этой главе я буду использовать в качестве примера технику которую я использовал для управления процессом создания драйверов устройства infiniband для ядра linux. Драйвер в большом вопросе (по крайней мере как драйвер запускается), с 25000 строк кода, распространяемых через 35 исходных файлов. Он поддерживается небольшой группой разработчиков.

Хотя значительная часть материала в этой главе, специфична для linux, те же принципы применяются к любой кодовой базе, для которых вы не основной владелец, и в котором вам нужно сделать много.

## 13.1. Проблема множества целей

Ядро linux меняется очень быстро, и никогда не было внутренне стабильным; разработчики часто делают резкие изменения между версиями. Это означает, что версия драйвера, который хорошо работает с определенной версией ядра не будет даже правильно собираться, как правило, на какой-нибудь другой версией.

Чтобы разрабатывать драйвер, мы должны иметь различные версий ядра linux в виду.

- Первой целью является основная ветка разработки ядра linux. Разработка кода в этом случае частично совместима с другими разработчиками в сообществе ядра, которые делают «drive-by» модификации для драйверов по мере их развития и совершенствования подсистем ядра.
- Мы также поддерживали некоторое количество «backports» для старых версий ядра linux, в целях удовлетворения потребностей клиентов, которые работали под управлением более старых дистрибутивов linux, которые не имеют наших драйверов. (Чтобы портировать кусок кода, чтобы модифицировать его работу для старой версии целевого окружения, чем версия для которой был разработана.)
- Наконец, мы делаем выпусков программ по графику, не обязательно совпадающего с тем, который используется дистрибьюторами linux и разработчиками ядра, так что мы можем добавлять новые возможности для клиентов, не заставляя их обновлять всё их ядро и дистрибутив.

### 13.1.1. Соблазнительные подходы, которые работают не очень хорошо

Есть два «стандартных» способа поддерживать часть программы, которая должна работать в самых разных условиях.

Во-первых, поддерживать ряд ветвей, каждая предназначена для единственной цели. Проблема такого подхода заключается в том, что необходимо поддерживать железную дисциплину в потоке изменений между хранилищами. Новые функции или исправления ошибок должны начинать жизнь в «первозданном» репозитории, а затем проходить в каждый бекпортированый репозиторий. backport изменения, более ограничены в ветках они должны передаваться по наследству; портирование применённые изменения в ветку, к которой оно не принадлежит, вероятно, вызовет остановку компиляции драйвера.

Вторая заключается в сохранении единого дерева исходников наполненных условными выражениями, которые в свою очередь включают или выключают блоки кода в зависимости от поставленной цели. Поскольку эти «ifdefs» не допускаются в дереве ядра linux, ручной или автоматический процесс должен последовательно вырезать их и давать на выходе чистое дерево. Код основной разработки таким образом быстро становится крысиным гнездом условных блоков, которые трудно понимать и поддерживать.

Ни один из этих подходов не подходит хорошо для ситуации, когда вы используете не «свою» каноническую копию исходного дерева. В случае драйверов linux, которые распространяются со стандартным ядром, дерево Линуса

содержит копию кода, который будет рассматриваться в мире как канонический. Апстрим версия «моих» драйверов может быть изменена людьми, которых я не знаю, без меня, я даже узнаю об этом только после отображения изменений в дереве Линуса.

Эти подходы добавил слабое место, которое затрудняет создание хорошо сформированного патча для отправки в апстрим.

В принципе, Mercurial Queues кажется хорошим кандидатом для управления сценарием развития, таким как выше. Хотя это действительно так, MQ содержит несколько дополнительных функций, которые делают работу более приятной.

# 13.2. Условное применение патчей с защитой

Возможно, лучший способом поддерживать здраво так много задач является возможность выбирать конкретные патчи применяемые для той или иной ситуации. MQ обеспечивает функция называющуюся «стражи» (guards) (которая берет свое начало с командой quilt's `guards`), которая делает тоже самое. Для начала, давайте создадим простой репозиторий для экспериментов

```
$ hg qinit
$ hg qnew hello.patch
$ echo hello > hello
$ hg add hello
$ hg qrefresh
$ hg qnew goodbye.patch
$ echo goodbye > goodbye
$ hg add goodbye
$ hg qrefresh
```

Это дает нам крошечный репозиторий, содержащее два патча, не имеющие никаких зависимостей друг от друга, потому что они затрагивают разные файлы.

Идея применения условия, которое вы можете указать в «тэг» патча со стражем, который является простой текстовой строкой по вашему выбору, которая скажет MQ, выбрать конкретного стража используемого при применении исправлений. MQ потом либо применит, либо пропустить, охраняемый патч, в зависимости от охранника, которого вы выбрали.

Патч может иметь произвольное число охранников, и каждый из них положительный («применить этот патч, если выбран этот охранник») или отрицательным («пропустить этот патч, если выбран этот охранник»). Патч, без охранника всегда применяется.

# 13.3. Управление защитой патча

Команда **qguard** позволяет определить, какая охрана должна распространяться на патч, или отобразит охранников, которые уже действуют. Без аргументов, он отображает охранников на текущий верхний патч.

```
$ hg qguard
goodbye.patch: unguarded
```

Чтобы установить положительного охранника на патч, используйте префикс «`+`» перед именем стража.

```
$ hg qguard +foo
$ hg qguard
goodbye.patch: +foo
```

Чтобы установить положительного охранника на патч, используйте префикс «`-`» перед именем стража.

```
$ hg qguard -- hello.patch -quux
$ hg qguard hello.patch
hello.patch: -quux
```

Notice that we prefixed the arguments to the **hg qguard** command with a `--` here, so that Mercurial would not interpret the text `-quux` as an option.

### Setting vs. modifying

Команда **qguard** устанавливает охранников на патч, но не изменять их. Это означает, что при запуске **hg qguard +a +b** на патч, а затем **hg qguard +c** на тот же патч, единственным охранником, который будет установлен на него будет `+c`.

Mercurial сохраняет охранников магазинов в файле `series`; том виде, в котором они хранятся их легко понять и править руками. (Другими словами, вы не должны использовать **qguard** команду, если вы не хотите, достаточно просто редактировать файл `series`).

```
$ cat .hg/patches/series
hello.patch #-quux
goodbye.patch #+foo
```

# 13.4. Выбор используемых охранников

Команда **qselect** определяет, какие охранники активны в данный момент времени. Результатом этого является определение, какие патчи MQ будет применяться при следующем запуске **qpush**. Команда не имеет другого эффекта, в частности, она не делает ничего, с уже применёнными патчами.

Без аргументов **qselect** перечисляет охранников применённых в настоящее время, по одному на строку. Каждый аргумент рассматривается как имя применённого охранника.

```
$ hg qpop -a
popping goodbye.patch
popping hello.patch
patch queue now empty
$ hg qselect
no active guards
$ hg qselect foo
number of unguarded, unapplied patches has changed from 1 to 2
$ hg qselect
foo
```

Если вам интересно применённые охранники хранятся в файле `guards`.

```
$ cat .hg/patches/guards
foo
```

Эффект от выбранных охранников мы увидим когда запустим **qpush**.

```
$ hg qpush -a
applying hello.patch
applying goodbye.patch
now at: goodbye.patch
```

Охранник не может начинаться с символа «`+`» или «`-`». Имя охранника не должно содержать пробелов, но большинство других символов разрешены. Если вы пытаетесь использовать неверное имя сторожа, MQ будет ругаться:

```
$ hg qselect +foo
abort: guard '+foo' starts with invalid character: '+'
```

Изменение выбранных охранников изменяет применяемые патчи.

```
$ hg qselect quux
number of guarded, applied patches has changed from 0 to 2
$ hg qpop -a
popping goodbye.patch
popping hello.patch
patch queue now empty
$ hg qpush -a
skipping goodbye.patch - guarded by '+foo'
```

Вы можете увидеть в приведенном ниже примере, что негативные охранники имеют приоритет над положительными охранниками.

```
$ hg qselect foo bar
number of unguarded, unapplied patches has changed from 0 to 2
$ hg qpop -a
no patches applied
$ hg qpush -a
applying hello.patch
applying goodbye.patch
now at: goodbye.patch
```

# 13.5. Правила применения патчей в MQ

Правила, которые использует MQ при решении вопроса о применении патча состоят в следующем.

- Патч, который не имеет охраны применяется всегда.
- Если патч имеет негативного охранника, который соответствует любому выбранному охраннику, патч будет пропущен.
- Если патч имеет положительного охранника, который соответствует любому выбранному охраннику, патч будет применён.
- Если патч имеет положительного или отрицательного охранников, но никто из них не выбран, патч будет пропущен.

# 13.6. Обрезка рабочего окружения

В работе над драйвером о котором я говорил ранее, я не применять патчи для нормального дерева ядра linux. Вместо этого я использую репозиторий, который содержит только снимок из исходных файлов и заголовков, которые имеют отношение к разработке Infiniband. Этот репозиторий 1% от размера репозитория ядра, так легче работать.

Я тогда выбрал «основную» версию поверх которой применял патчи. Это снимок дерева ядра linux содержал выбранные мной ревизии. Когда я беру снимок, я записываю id ревизии из репозитория ядра в сообщении фиксации. Поскольку снимок сохраняет «форму» и содержание соответствующих разделов дерева ядра, и я могу применить мои патчи поверх и моего маленького репозитория, и нормальный дерева ядра.

Как правило, на вершине основного дерева, к которому применяются патчи, должен быть снимок совсем недавнего апстрим дерева. Это в наибольшей степени способствует разработке патчей, которые могут легко быть отправлены в апстрим с небольшим или вообще без изменений.

# 13.7. Разделение файла `series`

Я классифицировал патчи в файле `series` на несколько логических групп. Каждый раздел патчей начинается с блока комментариев, которые описывают цель исправлении, которые за ним следуют.

Последовательность групп патчей я поддерживаю следующую. Порядок этих групп является важным, и я расскажу, почему после того, как опишу группы.

- Группа «принято». Патчи, которые команда разработчиков представила управляющему подсистемы infiniband, и которые он принял, но которые ещё не применены к снимку на основе малого репозитория. Эти патчи доступны «только для чтение», присутствуют только в трансформированном дереве в таком же состоянии, как они представлены сопровождающему верхнего репозитория.
- Группа «доработки». Патчи, которые я принял, но вышележащий апстрим сопровождающий попросил изменить, до того как он их примет.
- Группа «ожидают решения». Патчи, которые я еще не представили вышестоящему сопровождающему, но они готовы к работе. Они будут на некоторое время доступны «только для чтения». Если верхний сопровождающий их примет, при представлении, я перемещу их окончательно в группы «принято». Если он попросит меня изменить какой-либо из них, я перемещу их сначала в группу «доработки».

- Группа «в процессе разработки». Патчи, которые активно развиваются, и не должны быть представлены еще нигде.
- Группа «backport». Патчи, которые адаптированные для дерева исходных текстов с более старыми версиями ядра.
- Группа «Не распространять». Патчи, которые почему-то никогда не должны быть представлены в апстриме. Например, один такой патч может изменить встроенный идентификатор драйвера, чтобы было легче различать, области, между версией драйвера вне дерева и версией распространяемой поставщиками.

Теперь вернемся к причинам сортировки групп патчей таким образом. Мы хотели бы, самые низкие патчи в стеке были бы максимально устойчивы, так чтоб не нужно было перерабатывать патчи выше в связи с изменениями в соответствующем контексте. Введение патчей, которые никогда не будут изменятся первыми в файле `series` служит именно этой цели.

Мы хотели бы также чтоб патчи, которые мы знаем, что нужно изменить должны применяться в верхней части дерева исходных текстов, которая напоминает апстрим дерево, насколько это возможно. Именно поэтому мы постоянно принимаем патчи близко к верху.

В группах «backport» и «Не распространять» патчи находятся в конце файла `series`. backport патчи должны применяться поверх всех других исправлений, также патчи из группы «Не распространять» лучше держать от греха подальше.

# 13.8. Поддержка серии патчей

В своей работе я использую набор охранников, чтобы контролировать, какие патчи будут применяться.

- `accepted` защищает «принятые» патчи. Я включаю эту охрану большую часть времени. Когда я применяю патчи поверх дерева, где эти патчи уже есть, я отключаю эти патчи, и следующие за ними применяются чисто.
- Патчи, которые «завершены», но пока не отправлены, не имеют охраны. Если я применяю стек патчей к копию апстрим дерева, мне не нужно включать какого либо из охранников для получения достаточно безопасного исходного дерева.
- Те, патчи, которые требуют доработки, прежде чем вновь отправится охраняются с помощью `rework`.
- Для тех патчей, которые находятся в стадии разработки, я использую `devel`.
- backport патчи имеют различных охранников, по одному для каждой версии ядра, к которому они относятся. Например, backports патча у кода ядра 2.6.9 будет охранник `2.6.9`.

Такое разнообразие охранников дает мне значительную гибкость в определении вида дерева исходников, к которому я хочу в итоге применить патчи. В большинстве случаев, выбор соответствующих охранников автоматизирован в процессе сборки, но я могу вручную настроить охранников для использования в менее общих обстоятельствах.

## 13.8.1. Искусство писать backport патчи

Использование MQ, для написания backport патчей простой процесс. Все что нужно сделать в патче, это изменить фрагмент кода, который использует функции ядра которой нет в старой версии ядра, так что драйвер продолжает работать правильно в соответствии с этой старой версией.

Полезная цель при написании хорошего backport патча это сделать код выглядящим так, будто он была написан для старой версии ядра, на которую вы ориентируетесь. Чем менее навязчив патч, тем легче будет его понять и поддерживать. Если вы пишете коллекцию backport патчей, чтобы избежать эффекта «крысиного гнезда» — большого количества `#ifdef`s (блоков исходного кода, которые будут использоваться только условно) в коде, не вводите зависимые от версии `#ifdef`s в патчи. Вместо этого, напишите несколько патчей, каждый из которых дает безусловные изменения, а также контролируйте их примеением с помощью охранников.

Есть две причины выделить backport патчи в отдельную группу, подальше от «обычных» патчей, последствия которых они модифицируют. Во-первых, их смешение делает более сложным в использовании инструменты такие, как расширение `patchbomb` для автоматизации процесса передачи патчей сопровождающим апстрима. Во-

вторых, портированое исправление может нарушить контекст, в котором применяются обычные патчи, что делает невозможным применение очередного патча чисто, если ранее применился backport патч.

# 13.9. Полезные советы для разработки с MQ

## 13.9.1. Организация патчей в каталогах

Если вы работаете на крупный проект с MQ, не трудно накопить большое количество патчей. Например, у меня есть репозиторий патчей, который содержит более 250 патчей.

Если вы можете сгруппировать эти патчи на отдельные логические категории, вы можете, если хотите хранить их в разных каталогах; MQ не имеет проблем с именами патчей, которые содержат разделители пути.

## 13.9.2. Просмотр истории патча

Если вы разрабатываете набор патчей в течение длительного времени, хорошая идея сохранить их в репозитории, о чем говорится в разделе Раздел 12.12, «Управление патчами в репозитории». Если вы это сделаете, вы быстро обнаружите, что использование команды **hg diff**, чтобы взглянуть на историю изменений патча обреченно на провал. Это отчасти потому, что вы смотрите на вторую производную от реального кода (список различий различий), но также и потому, MQ добавляет шуму в этот процесс, меняя временные метки и имена каталогов при обновлении патча.

Тем не менее, вы можете использовать расширение `extdiff`, которое поставляется вместе с Mercurial, для просмотра различий двух версий патча в читабельном виде. Для этого вам понадобится сторонний пакет называющийся `patchutils` [web:patchutils]. Он дает команду **interdiff**, которая показывает разницу между двумя diff-ами как diff. Используется на двух версиях одного и того же патча, она выдаёт diff, который представляет собой различия с первой по вторую версии.

Вы можете включить расширение `extdiff` обычным способом, путем добавления строки в раздел `extensions` вашего `~/.hgrc`.

```
[extensions]
extdiff =
```

Команда **interdiff** ожидает два аргумента — два названия файлов, но расширение `extdiff` передаёт в запускаемую программу пару каталогов, каждый из которых может содержать произвольное количество файлов. Таким образом, потребуется небольшая программа, которая будет запускать **interdiff** к каждой паре файлов в этих двух каталогах. Эта программа существует и называется `hg-interdiff` в каталоге `examples` репозитория исходного кода, прилагаемый к этой книге.

Запустить программу `hg-interdiff` в в вашей консоли, вы можете запустить его следующим образом, внутри каталога патчей MQ:

```
hg extdiff -p hg-interdiff -r A:B my-change.patch
```

Так как вы, наверное, хотите, чтобы использовать эту многословную команду часто, вы можете получить `hgext` чтобы сделать её доступной как обычную Mercurial команду, опять же, редактированием `~/.hgrc`.

```
[extdiff]
cmd.interdiff = hg-interdiff
```

Эта строка приказывает `hgext` сделать команду `interdiff` доступной, поэтому теперь вы можете сократить предыдущую команду **extdiff** до чего-то чуть более простого.

```
hg interdiff -r A:B my-change.patch
```

### Примечание

Команда **interdiff** хорошо работает только если лежащее в основе патчей остаются теми же. Если при создании патча, изменить основные файлы, а затем сгенерировать патч, **interdiff** не сможет произвести полезный вывод.

Расширение `extdiff` полезно для более просто улучшение отображения патча MQ. Чтобы узнать больше о ней, перейдите в раздел Раздел 14.2, «Гибкая поддержка diff с расширением `extdiff`».

# Глава 14. Добавление функциональности с помощью расширений.

Хотя ядро mercurial достаточно полно с точки зрения функциональности, но оно намеренно лишено замысловатых ухищрений. Такой подход, сохранять простоту, делает программное обеспечение легко поддерживаемым для сопровождающих и для пользователей.

Тем не менее, Mercurial не ящик с неизменяемым набором команд: вы можете добавлять новые функции к нему в качестве расширений (иногда их называют плагины). Мы уже обсуждали некоторые из этих расширений в предыдущих главах.

- В разделе Раздел 3.3, «Упрощение последовательности pull-merge-commit» включает расширение `fetch`, оно объединяет вытягивание изменений и слияние их с локальными изменениями в единую команду, **fetch.**
- В Глава 10, *Обработка событий в репозитории с помощью ловушек*, мы рассмотрели несколько расширений, которые полезны для связанных с ловушками функции: `acl` добавляет списки контроля доступа; `bugzilla` добавляет интеграцию с bugzilla системы слежения за ошибками; и `notify` направляет электронные уведомления о новых изменениях.
- Mercurial Queues расширения для управления патчами так бесценно, что заслуживает 2 глав и приложения о себе. Глава 12, *Управление изменениями с Mercurial Queues* охватывает основы; Глава 13, *Расширенное использование Mercurial Queues* рассматривает сложные вопросы, и Приложение B, *Справочник Mercurial Queues* описывает детали каждой команды.

В этой главе мы рассмотрим некоторые другие расширения, которые доступны для Mercurial, а также кратко остановимся на некоторых из механизмов о которых вам нужно знать, если вы хотите написать собственное расширение.

- В разделе Раздел 14.1, «Улучшение производительности с расширением `inotify`», мы обсудим возможность огромного прироста производительности с использованием расширения `inotify`.

## 14.1. Улучшение производительности с расширением `inotify`

Заинтересованы ли вы в чтоб, некоторые наиболее распространенные операции Mercurial запускались в 100 раз быстрее? Читайте дальше!

Mercurial имеет высокую производительность при нормальных обстоятельствах. Например, когда вы запускаете команду **hg status**, Mercurial сканирует почти каждую директорию и файл в репозитории, чтобы отображать статус файла. Многие другие команды Mercurial должны делать ту же работу за кулисами, например, команда **hg diff** использует механизм статусов, чтобы избежать дорогостоящих операций сравнения файлов, которые очевидно не изменились.

Потому что получение статуса файла имеет решающее значение для хорошей производительности, авторы Mercurial максимально оптимизировали код. Однако, не удастся избежать того, что при запуске **hg status**, Mercurial необходимо выполнить по крайней мере один дорогой системный вызов для каждого управляемого файла определяя, изменился ли он с момента последней проверки Mercurial. При достаточно большом хранилище, это может занимать длительное время.

Чтобы увидеть эффект от этого, я создал репозиторий, содержащий 150000 управляемых файлов. Я потратил на запуск **hg status** 10 секунд, даже если ни один из этих файлов не был изменён.

Многие современные операционные системы содержат уведомления о событиях файла. Если программа регистрируется в соответствующей службе, операционная система будет уведомлять его каждый раз когда

интересующие нас файлы создаются, изменяются или удаляются. На linux системах, компонентов ядра, отвечающий за это называется `inotify`.

Расширение Mercurial `inotify` общается с компонентом ядра `inotify` для оптимизации команды **hg status**. Расширение состоит из двух компонентов. Демон сидит в фоновом режиме и получает уведомления от подсистемы `inotify`. Он также ожидает соединения от обычной команды Mercurial. Расширение изменяет поведение Mercurial так, что вместо сканирования файловой системы, он отправляет запрос демону. Так как демон имеет полную информации о состоянии хранилища, он может ответить результатом мгновенно, что избавляет от необходимости проверять каждый каталог и файл в репозитории.

Напомним, 10 секунд, я потратил в простом Mercurial на запуск **hg status** на репозиторий со 150000 файлов. С включенным расширением `inotify`, время сократилось до 0,1 секунды, в сто раз быстрее.

Перед тем как продолжить, пожалуйста, обратите внимание на некоторые предостережения.

- Расширение `inotify` является linux-специфичным. Потому что использует интерфейс `inotify` ядра linux напрямую, оно не работает на других операционных системах.
- Он должно работать в любом дистрибутиве linux, который был выпущен после начала 2005 года. Старые дистрибутивы могут иметь ядро, без поддержки `inotify`, или версии `glibc`, которая не имеет необходимой поддержки интерфейса.
- Не все файловые системы, пригодные для использования с расширением `inotify`. Сетевые файловые системы, такие как nfs являются не позволяют, например, особенно если вы работаете с Mercurial на нескольких системах, монтирующих одну сетевую файловую систему. Подсистема `inotify` ядра не может узнать об изменениях, внесенных в другой системе. Большинство локальных файловых систем (например, ext3, xfs, reiserfs) должны работать нормально.

The `inotify` extension is shipped with Mercurial since 1.0. All you need to do to enable the `inotify` extension is add an entry to your `~/.hgrc`.

```
[extensions] inotify =
```

Когда расширение `inotify` включено, Mercurial будет автоматически и прозрачно стартовать статусного демона при первом запуске команды, которой необходим статус репозитория. Он запускает один демон на хранилище.

Статусный демон запускается молча, и работает в фоновом режиме. Если вы посмотрите на список запущенных процессов после того как вы включите расширение `inotify` и запустите несколько команд в разных репозиториях, вы увидите несколько процессов `hg`, ожидающих обновлений из ядра и запросов от Mercurial.

Первый раз, когда вы выполните команду Mercurial в репозитории, в котором включено расширение `inotify`, он будет работать с примерно такой же производительностью, как обычная команда Mercurial. Это потому, что статус демону необходимо выполнять обычное сканирование статуса, с тем чтобы получить исходное состояние, на которое позднее применяются обновления ядра. Тем не менее, каждая последующая команда, которая делает любые проверки статуса должна выполнятся заметно быстрее на репозитории даже несмотря на довольно скромные размеры. А еще лучше, чем больше ваш репозиторий, тем больше преимущество в производительности вы увидите. Демон `inotify` делает операции со статусом практически мгновенными на репозиториях любых размеров!

Если вы хотите, вы можете запустить демона вручную используя команду **inserve**. Она дает вам немного более тонкий контроль над тем, как демон должен работать. Эта команда, конечно, будет доступна только при включенном расширении `inotify`.

Когда вы используете расширение `inotify`, вы не должны заметить никакой разницы в поведении Mercurial, с единственным исключением, связанные со статусом команды работают гораздо быстрее, чем раньше. Вы можете совершенно точно ожидать, что команды не будут печатать различные результаты; они не должны давать разные результаты. Если любая из этих ситуаций происходит, пожалуйста, сообщите об ошибке.

# 14.2. Гибкая поддержка diff с расширением `extdiff`

Встроенная команда **hg diff** Mercurial выводит текст унифицированного diff-а.

```
$ hg diff
diff -r 415b05dedf77 myfile
--- a/myfile Thu Feb 02 14:09:50 2012 +0000
+++ b/myfile Thu Feb 02 14:09:50 2012 +0000
@@ -1,1 +1,2 @@
 The first line.
+The second line.
```

Если вы хотели бы использовать внешний инструмент, чтобы увидеть изменения, вы можете использовать расширение `extdiff`. Это позволит вам использовать, например, графический инструмент для сравнения.

Расширение `extdiff` идёт в комплекте с Mercurial, так что его легко установить. В разделе `extensions` вашего `~/.hgrc`, просто добавьте однострочную запись для включения расширения.

```
[extensions]
extdiff =
```

Оно добавляет команду **extdiff**, которая по умолчанию использует команду **diff** вашей системы, чтобы создать унифицированный diff в том же виде, что и встроенная команда **hg diff**.

```
$ hg extdiff
--- /tmp/extdiff.fhLF3h/a.415b05dedf77/myfile 2012-02-02 14:09:50.983150885 +0000
+++ /tmp/extdiffAi1Ydv/a/myfile 2012-02-02 14:09:50.775150885 +0000
@@ -1 +1,2 @@
 The first line.
+The second line.
```

Результат не будет таким же, как с помощью встроенной команды в **hg diff**, так как вывод **diff** изменяется от одной системы к другой, даже если используются те же опции.

Как «`делается снимок`» строк вывода представленного выше, команда **extdiff** работает путем создания двух снимков дерева исходников. Первый снимок исходная ревизия, вторая целевая ревизия или рабочий каталог. Команда **extdiff** генерирует эти снимки во временную директорию, передает имя каждого каталога внешней программе просмотра изменений, а затем удаляет временную директорию. Для повышения эффективности, снимки содержат только каталоги и файлы, которые изменились между двумя ревизиями.

Имя каталога снимка имеет то же имя, как базовое имя вашего репозитория. Если ваш репозиторий находится в `/quux/bar/foo`, то имя каждого каталога снимков будет `foo`. Каждое имя каталога снимка имеет id набора изменений в конце, в случае необходимости. Если снимок ревизии `a631aca1083f`, каталог будет называться `foo.a631aca1083f`. Снимок рабочего каталога не будет иметь id ревизии в конце, поэтому будет называться просто `foo` как в этом примере. Чтобы увидеть, как это выглядит на практике, еще раз посмотрим на пример **extdiff** выше. Обратите внимание, что сравнивается имя каталога снимков встроенное в его заголовке.

Команда **extdiff** принимает две важных опций. Опция `hg -p` позволяет выбирать программу для просмотра различий, вместо **diff**. С опцией `hg -o`, вы можете изменить параметры, которые **extdiff** передаёт в программу (по умолчанию это опции «`-Npru`», которые имеют смысл только если вы работаете с **diff**). В других случаях команда **extdiff** действует подобно встроенной команде **hg diff** можно использовать те же имена опций, синтаксис и аргументы для указания ревизий, которые вы хотите, и так далее.

Например, вот как запустить обычную системную команду **diff**, получим сгенерированные контекстные различия (с использованием опции `-c`) вместо унифицировнных различий, и пять строк контекста, вместо 3 по умолчанию (передаём `5` в качестве аргумента опции `-C`).

```
$ hg extdiff -o -NprcC5
*** /tmp/extdiff.0DjGtx/a.415b05dedf77/myfile Thu Feb  2 14:09:51 2012
--- /tmp/extdiffAi1Ydv/a/myfile Thu Feb  2 14:09:50 2012
***************
*** 1 ****
--- 1,2 ----
  The first line.
+ The second line.
```

Запуск визуального инструмента различий так же легко. Вот как можно запустить программу **kdiff3**.

```
hg extdiff -p kdiff3 -o
```

Если команда просмотра diff не может справиться с каталогами, вы можете легко обойти это с небольшим скриптом. В качестве примера таких сценариев в действии с расширением `mq` и командой **interdiff** смотрите в разделе Раздел 13.9.2, «Просмотр истории патча».

### 14.2.1. Определение псевдонимов команд

Может быть трудным запомнить параметры для обеих команд **extdiff** и команды просмотра которую вы используете, то расширение `extdiff` позволяет определить новые команды, которые будут ссылаться на вашу программу с заданными опциями.

Все, что вам нужно сделать, это отредактировать файл `~/.hgrc`, а также добавить раздел с именем `extdiff`. Внутри этого раздела, вы можете указать несколько команд. Вот как можно добавить команду `kdiff3`. Как только вы определите её, вы сможете ввести «`hg kdiff3`» и `extdiff` расширение будет запускать **kdiff3**.

```
[extdiff]
cmd.kdiff3 =
```

Если вы оставите правую часть определения пустой, как выше, `extdiff` модуль использует имя команды, которую Вы определили в качестве имени внешней программы для запуска. Однако эти имена не должны быть одинаковыми. Здесь мы определим команду под названием «`hg wibble`», которая запускает **kdiff3**.

```
[extdiff]
 cmd.wibble = kdiff3
```

Вы можете также задать параметры по умолчанию, которые вы хотите передать вашей программе просмотра diff. Используя префикс «`opts.`», а затем имя команды, для которой применяются параметры. Этот пример определяет команду «`hg vimdiff`», которая запускает расширение `DirDiff` редактора **vim**.

```
[extdiff]
 cmd.vimdiff = vim
opts.vimdiff = -f '+next' '+execute "DirDiff" argv(0) argv(1)'
```

## 14.3. cherrypicking изменений используя расширение `transplant`

Необходимо побеседовать с Бренданом об этом.

## 14.4. Отправить изменений по электронной почте с расширением `patchbomb`

Многие проекты поддерживают культуру «Обзора изменений», в которой люди отправляют их модификаций в список рассылки, чтобы другие могли прочитать и прокомментировать, прежде чем они зафиксируют окончательный вариант в общем репозитории. В некоторых проектах, есть люди, которые выступают в качестве хранителей, они применяются изменения со стороны других людей в репозиторий, в который другие не имеют доступа.

Mercurial позволяет легко отправлять по электронной почте ревизии для просмотра или применения, с помощью расширения `patchbomb`. Расширение называется так потому, ревизии отправляются в формате патчей, и обычно отправляется одна ревизия в сообщении. Отправка большого количества изменений по электронной почте, таким образом, подобна «бомбардировке» почтового ящика получателя, поэтому и «patchbomb».

Как обычно, базовая конфигурация расширения `patchbomb` занимает всего одну или две строки в файле `~/.hgrc`.

```
[extensions]
patchbomb =
```

После того как вы включите расширение, вам станет доступна новая команда, названная **email**.

Безопасный и лучший способ для вызова команды **email**, всегда запускать сначала с опцией `hg -n`. Она покажет вам, что команда будет отправлять, фактически не пересылая ничего. После того как вы бросите быстрый взгляд на изменения и убедитесь, что вы отправляете всё правильно, вы можете запустить эту же команду без опции `hg -n`.

Команда **email** понимает такой же синтаксис указания ревизии как и любая другая команда Mercurial. Например, эта команда будет посылать каждую ревизию между 7 и `tip` включительно.

```
hg email -n 7:tip
```

Вы также можете указать репозиторий для сравнения. Если вы предоставляете репозиторий, но не ревизию, команда **email** отправить все изменения в локальном репозитории, которые не представлены в удаленном репозитории. Если вы дополнительно укажите ревизию или название ветки (последняя с использованием опции `hg -b`), письма будут содержать эти ревизии.

Это совершенно безопасно выполнить команду **email** без указания имен людей, которым вы хотите отправить документ: если вы это сделаете, вам будет выдано приглашение указать их в интерактивном режиме. (Если вы используете linux или unix-подобную операционную систему, вы должны иметь расширенные в `readline`-стиле возможности редактирования при вводе этих заголовков, что тоже полезно.)

Когда вы отправляете только одну ревизию, команда **email** по умолчанию будет использовать первую строку ревизии в качестве заголовка единственного отправленного сообщения.

Если вы отправляете несколько ревизий, команда **email**, как правило, отправляет сообщение на каждое изменение. Вместе с предисловием серии с вступительным сообщением, в котором вы должны описать цель набора изменений, которые вы посылаете.

## 14.4.1. Изменение поведения patchbomb

Не каждый проект имеет одинаковую конвенцию, для отправки ревизий по электронной почте; `patchbomb` расширение пытается приспособиться под различные варианты с помощью параметров командной строки.

- Вы можете написать вступительное сообщение в командной строке, используя опцию `hg -s`. Она принимает один аргумент, текст который используется в заголовке.
- Чтобы изменить адрес электронной почты, с которого отправляются сообщения, используйте опцию `hg -f`. Она принимает один аргумент, адрес электронной почты.
- По-умолчанию, отправляется унифицированный diff (см. раздел Раздел 12.4, «Понимание патчей» для описания формата), один на сообщение. Вы можете отправить бинарный пакет вместо этого с использованием опции `hg -b`.
- Унифицированному diff обычно предшествуют метаданные заголовка. Вы можете пропустить его, и отправить без всяких украшений diff используя опцию `hg --plain`.
- Различия, как правило, отправляются «встроенными» в тело письма, как и описание патча. Это делает их просмотр простым для наибольшего числа читателей, цитирования и ответов на части diff-а, так как некоторые почтовые клиенты, цитируют только первую часть MIME тела в сообщении. Если вы хотите, отправлять описание и различия в отдельных частях тела, используйте опцию `hg -a`.
- Вместо отправки почты, вы можете записать их в папку формата `mbox` с помощью опции `hg -m`. Эта опция принимает один аргумент, имя файла для записи.
- Если вы хотели бы добавить сводку **diffstat**-формата для каждого патча, и одну во вступительное сообщение, используя опцию `hg -d`. Команда **diffstat** отображает таблицу, содержащую имя каждого файла в патчах, количество строк затронутого, и гистограмму, показывающую, как каждый файл будет изменен. Это дает читателям качественный взгляд на комплекс патчей.

# Приложение A. Переход на Mercurial

Простой способ прощупать почву с новым инструментом контроля версий, это поэкспериментировать с переключением существующих проектов, а не начинать новый проект с нуля.

В этом приложении мы обсудим, как импортировать историю проекта в Mercurial, и на что обратить внимание, если вы привыкли к другой системе контроля версий.

## A.1. Импорт истории из другой системы

Mercurial поставляется с расширением называемым `convert`, которое может импортировать историю проекта из наиболее популярных систем контроля версий. В то время когда эта книга была написана, оно могло импортировать историю из следующих систем:

- Subversion
- CVS
- git
- Darcs
- Bazaar
- Monotone
- GNU Arch
- Mercurial

(Чтобы понять, почему Mercurial поддерживает в качестве источника самого себя, смотрите в разделе Раздел A.1.3, «Очистка дерева».)

Вы можете включить расширение в обычном порядке, отредактировав файл `~/.hgrc`.

```
[extensions]
convert =
```

Это сделает доступной команду **hg convert**. Команда проста в использовании. Например, эта команда будет импортировать историю Subversion для Nose unit testing framework в Mercurial.

```
$ hg convert http://python-nose.googlecode.com/svn/trunk
```

Расширение `convert` действует поэтапно. Иными словами, после того как вы выполните **hg convert** первый раз, запуская его снова вы будете импортировать любые новые ревизии, совершенные после первого запуска. Инкрементные преобразования будут работать только если вы запустите **hg convert** в том же репозитории Mercurial, который вы использовали, потому что расширение `convert` сохраняет некоторые частные метаданные не под контролем системы контроля версий, в файле с именем `.hg/shamap` внутри целевого репозитория.

Если вы хотите начать делать изменения, используя Mercurial, то лучше клонировать дерево, в котором вы будете проводить преобразование и оставить оригинальное дерево для последующий инкрементальных преобразований. Это самый безопасный способ позволяет вытягивать и объединять будущие изменения из исходной системы контроля версий в своём новом активном репозитории Mercurial.

### A.1.1. Конвертирование нескольких ветвей

Команда **hg convert** приведенная выше преобразует только историю ветки `trunk` репозитория Subversion. Если мы вместо этого используем URL `http://python-nose.googlecode.com/svn`, Mercurial автоматически обнаружит `trunk`, `tags` и `branches`, которые обычно используют проекты Subversion, и он будет импортировать каждый как отдельную ветвь Mercurial.

По умолчанию, каждая ветка Subversion импортируется в Mercurial с названием ветки. После завершения преобразования, можно получить список имен активных веток в Mercurial репозитории с помощью **hg branches -a**. Если вы предпочитаете импортировать ветви Subversion без названия, используйте опцию `--config convert.hg.usebranchnames=false` команды **hg convert**.

Как только вы преобразовывали свое дерево, если вы хотите следовать обычной для Mercurial практике работы в одном дереве, содержащем одну ветку, можно клонировать одну ветку используя **hg clone -r mybranchname**.

## A.1.2. Связь имён пользователей

Некоторые средства контроля версий сохраняют только короткие имена пользователей при фиксации, и они могут трудно интерпретироваться. Обычно в Mercurial сохраняется имя коммиттера и адрес электронной почты, который является гораздо более полезным для разговора с ними после факта фиксации.

Если вы преобразовываете дерево из системы контроля версий, которая использует короткие имена, можно сопоставить эти имена с длинными эквивалентами, передавая опцию `--authors` команде **hg convert**. Этот параметр принимает имя файла, который должен содержать записи следующего вида.

```
arist = Aristotle <aristotle@phil.example.gr>
soc = Socrates <socrates@phil.example.gr>
```

Всякий раз, когда `convert` встретит ревизию с именем пользователя `arist` в исходном репозитории, он будет использовать имя `Aristotle <aristotle@phil.example.gr>` в преобразованной для Mercurial ревизии. Если совпадения не найдется, то имя используется дословно.

## A.1.3. Очистка дерева

Не все проекты имеют чистую историю. Там может быть каталог, который никогда не должен были проверятся, слишком большие файлы, или целая иерархия, которая должна быть переработана.

Расширение `convert` поддерживает идею «карты файлов», которая позволяет реорганизовать файлы и каталоги в проекте, при импорте истории проекта. Это полезно не только при импорте истории из других систем контроля версий, но также и для того чтобы подрезать или реорганизовать дерево Mercurial.

Чтобы указать карту файлов, используйте опцию `--filemap` и укажите имя файла. Карта файлов содержит строки в следующем формате.

```
# This is a comment.
# Empty lines are ignored.

include path/to/file

exclude path/to/file

rename from/some/path to/some/other/place
```

Директива `include` указывает файл или все файлы в каталоге, которые будут включены в целевой репозиторий. Она также исключает любые другие файлы и директории не включенные явно. Директива `exclude` указывает файлы или директории, которые будут исключены, а другие прямо не упоминаемые должны быть включены.

Чтобы переместить файлы или каталог из одного места в другое, используйте директиву `rename`. Если вам необходимо переместить файл или каталог из подкаталога в корневой каталог репозитория, используйте `.` в качестве второго аргумента директивы `rename`.

## A.1.4. Улучшение эффективности преобразования Subversion

Часто требуется несколько попыток, прежде чем будет получена идеальное сочетание карты пользователей, карты файлов и других параметров конвертации. Преобразование репозитория subversion через протокол доступа похожий на `ssh` или `http` может протекать в тысячи раз медленнее, чем Mercurial реально способен работать, из-за задержек в сети. Это делает подбор идеальных настроек преобразования очень тяжелой.

Команда **svnsync** [http://svn.collab.net/repos/svn/trunk/notes/svnsync.txt] может значительно ускорить преобразование репозитория Subversion. Она создаёт зеркало только для чтения для репозитория Subversion. Идея заключается в создании локального зеркала вашего Subversion дерева, а затем преобразовании зеркала в Mercurial репозиторий.

Предположим, мы хотим преобразовать репозиторий Subversion для популярного проекта Memcached в Mercurial дерево. Во-первых, мы создаем локальный репозиторий Subversion.

```
$ svnadmin create memcached-mirror
```

Далее, мы установим ловушку Subversion необходимую для **svnsync**.

```
$ echo '#!/bin/sh' > memcached-mirror/hooks/pre-revprop-change
$ chmod +x memcached-mirror/hooks/pre-revprop-change
```

Затем мы инициализируем **svnsync** в этом репозитории.

```
$ svnsync --init file://`pwd`/memcached-mirror \
  http://code.sixapart.com/svn/memcached
```

Наш следующий шаг — начинаем процесс зеркалирования **svnsync**.

```
$ svnsync sync file://`pwd`/memcached-mirror
```

Наконец, мы импортируем истории нашего локального зеркала Subversion в Mercurial.

```
$ hg convert memcached-mirror
```

Мы можем использовать этот процесс инкрементно, если репозиторий Subversion по-прежнему используется. Мы запускаем **svnsync** для вытягивания новых изменений в наше зеркало, а затем запускаем **hg convert** для импорта их в наше дерево Mercurial.

Есть два преимущества двухступенчатого импорта с **svnsync**. Во-первых, используется более эффективная синхронизация кода Subversion по сети, чем при **hg convert**, так как меньше данных передаётся по сети. Во-вторых, импорт из локального дерева Subversion настолько быстр, что вы можете изменять ваши установки неоднократно без ожидания сетевых процессов каждый раз.

# A.2. Переход из Subversion

Subversion в настоящее время является самой популярной системой контроля версий с открытым исходным кодом. Хотя есть много различий между Mercurial и Subversion, что делает переход от Subversion на Mercurial не особенно трудным. Обе имеют схожие наборы команд и в целом единый интерфейс.

## A.2.1. Философские различия

Основное различие между Subversion и Mercurial, конечно то, что Subversion является централизованной, в то время как Mercurial распределённой. Mercurial хранит всю историю проекта на локальном диске, сеть нужна только если вы хотите явно общаться с другим репозиторием. В отличие от Subversion, которая очень мало информации хранит локально, и таким образом, клиент должен связываться со своим сервером для большинства распространенных операций.

Subversion более или менее может обходится без четко определенного понятия ветви: какая-то часть пространства имён сервера объявляется веткой и является предметом конвенции, с программным ограничением доступа. Mercurial обрабатывает репозиторий в качестве единой ветви управления.

### A.2.1.1. Набор команд

Поскольку Subversion не знает, что части ее пространства имён в действительности ветви, то относится к большинству команд, как запросу работы над любым каталогом в котором вы сейчас находитесь. Например, если вы запускаете **svn log**, вы получите историю той части дерева, в которой вы находитесь, а не дерева в целом.

Команды Mercurial ведут себя по-другому, по умолчанию они работают со всем репозиторием. Запустите **hg log** и он расскажет вам историю всего дерева, в какой бы части рабочей директории вы не находились в данный момент. Если вы хотите увидеть историю одного файла или каталога, просто укажите его имя, например, **hg log src**.

Из моего собственного опыта, это различие в поведение по умолчанию, скорее всего запутает вас, если у вас есть необходимость часто переключаться между двумя инструментами.

### A.2.1.2. Многопользовательская эксплуатация и безопасность

В Subversion, это нормально (хотя и несколько неодобрительно) для нескольких пользователей совместно работать в одной ветке. Если Алиса и Боб работают вместе, и Алиса совершает какое-либо изменений в их общей ветке, Бобу необходимо обновить свою рабочую копию прежде чем он сможет фиксировать свои изменения. Так как в это время он не имеет постоянной записи своих изменений, он может повредить или потерять их в ходе или после обновления.

Mercurial использует модель фиксация-потом-слияние. Боб фиксирует свои изменения локально перед вытягиванием изменений, или отправкой их на сервер, который он разделяет с Алисой. Если Алиса отправит ее изменения перед тем как Боб попытается отправить свои, он не сможет отправить свои изменения пока вытянет ее, не выполнит слияние с ними, и не зафиксирует результат слияния. Если он делает ошибку в процессе слияния, он все еще может вернуться к ревизии, в которой записаны его изменения.

Стоит отметить, что это общие методы работы с этими инструментами. Subversion поддерживает безопасную модели работы-в-своей-собственной-ветке, но это достаточно громоздко на практике и не будет широко использоваться. Mercurial может поддерживать менее безопасный режим, позволяющий вытягивать изменения и объединять их поверх незафиксированных изменений, но это считается в высшей степени необычным.

### A.2.1.3. Публикация против локальных изменений

Команда Subversion **svn commit** немедленно публикует изменения на сервер, где их могут видеть все, кто имеет доступ на чтение.

С Mercurial фиксация всегда локальна, и должна быть опубликована через команду **hg push** позднее.

Каждый подход имеет свои преимущества и недостатки. Модель Subversion означает, что изменения публикуются и, следовательно могут быть пересмотрены и использованы, немедленно. С другой стороны, это означает, что пользователь должен иметь доступ к фиксации в репозиторий для того, чтобы использовать программное обеспечение в обычном порядке, а доступ к фиксации не легко выдается в большинстве проектов с открытым кодом.

Подход Mercurial позволяет любому, кто может клонировать репозиторий вносить изменения без необходимости каких-либо разрешений, и они могут публиковать свои изменения и продолжать участвовать так как они считают нужными. Различие между фиксацией и публикацией открывает возможность фиксировать изменения на своём ноутбуке и на несколько дней забыть, чтобы потом отправить их, что в редких случаях сотрудник может временно застрять.

## A.2.2. Краткий справочник

Таблица **A.1.** Команды **Subversion** и их эквиваленты в **Mercurial**

| **Subversion** | **Mercurial** | Примечание |
|---|---|---|
| **svn add** | **hg add** | |
| **svn blame** | **hg annotate** | |
| **svn cat** | **hg cat** | |
| **svn checkout** | **hg clone** | |
| **svn cleanup** | n/a | Очистка не требуется |
| **svn commit** | **hg commit**; **hg push** | **hg push** публикует после фиксации |
| **svn copy** | **hg clone** | Создание новой ветки |

| **Subversion** | **Mercurial** | Примечание |
|---|---|---|
| **svn copy** | **hg copy** | Копирование файлов и директорий |
| **svn delete** (**svn remove**) | **hg remove** | |
| **svn diff** | **hg diff** | |
| **svn export** | **hg archive** | |
| **svn help** | **hg help** | |
| **svn import** | **hg addremove**; **hg commit** | |
| **svn info** | **hg parents** | Показывает какая ревизия сейчас вытянута |
| **svn info** | **hg showconfig paths.parent** | Показывает вытянутый URL |
| **svn list** | **hg manifest** | |
| **svn log** | **hg log** | |
| **svn merge** | **hg merge** | |
| **svn mkdir** | n/a | Mercurial не отслеживает каталоги |
| **svn move** (**svn rename**) | **hg rename** | |
| **svn resolved** | **hg resolve -m** | |
| **svn revert** | **hg revert** | |
| **svn status** | **hg status** | |
| **svn update** | **hg pull -u** | |

# A.3. Полезные советы для новичков

В некоторых системах контроля версий, печать diff для одного зафиксированной ревизии может быть мучительным. Например, в Subversion, чтобы посмотреть, что изменилось в редакции 104654, вы должны ввести **svn diff -r104653:104654**. Mercurial устраняет необходимость указания ID-ревизии дважды в общем случае. Для простого просмотра используйте **hg export 104654**. Для сообщения лога и последующего diff **hg log -r104654 -p**.

При запуске **hg status** без каких-либо аргументов, он выдает статус всего дерева, с путями относительно корня репозитория. Это делает сложным копирование имени файла из вывода **hg status** в командную строку. Если вы укажете имя файла или каталога команде **hg status**, она будет печатать пути относительно вашего текущего местоположения вместо этого. Таким образом, чтобы получить статус всего дерева с помощью **hg status**, с путями, которые считаются относительно текущего каталога, а не корня репозитория, укажите вывод команды **hg root** как параметр для команды **hg status**. Вы легко можете сделать это следующим образом на Unix-подобной операционной системе:

```
$ hg status `hg root`
```

# Приложение B. Справочник Mercurial Queues

## B.1. Справочник команд MQ

Для просмотра команд предоставляемых mq, используйте команду **hg help mq**.

### B.1.1. qapplied — печатает применённые патчи

Команда **qapplied** выводит стек текущих применённых патчей. Патчи печатаются в порядке от старых к новым, так что последний патч в списке будет «верхним» патчем.

### B.1.2. qcommit — фиксирует изменения в репозитории очереди

Команда **qcommit** фиксирует замеченные изменения в репозитории `.hg/patches`. Эта команда работает только если директория `.hg/patches` является репозиторием, например, вы создали каталог с помощью **hg qinit `-c`** или запустив **hg init** в директории после запуска **qinit**.

Эта команда представляет собой сокращение от **hg commit --cwd .hg/patches**.

### B.1.3. qdelete — удалить патч из файла `series`

Команда **qdelete** удаляет запись о патче из файла `series` в каталоге `.hg/patches`. Это не извлечение патча, если патч уже применён. По умолчанию, это не приводит к удалению файла патча, для этого используйте опцию `-f`.

Опции:

- `-f`: Удалить файл патча.

### B.1.4. qdiff — печатает diff для верхнего применяемого патча

Команда **qdiff** выводит список различий верхнего применяемого патча. Это эквивалентно **hg diff -r-2:-1**.

### B.1.5. qfinish — перемещает применённые патчи в историю репозитория

Команда **hg qfinish** преобразует указанные примененные патчи в постоянную ревизию, перемещая их из-под контроля MQ, чтобы они преобразовались в нормальную историю репозитория.

### B.1.6. qfold — слияние («свёртка»), нескольких патчей в один

Команда **qfold** объединяет несколько исправлений в верхний применяемый патч, так что верхний применяемый патч делает объединение всех изменений в патчах запроса.

Патчи для свёртки не должны быть применены; **qfold** выйдет с ошибкой, если какой-либо патч применён. Порядок, в котором складываются патчи является значимым; **hg qfold a b** означает «применить текущий верхний патч, затем `a`, потом `b`».

Комментарии сворачиваемых патчей добавляются к комментариям патча назначения, при этом каждый блок комментариев разделен символами трёх звездочек («`***`»). Использование опции `-e` редактирует сообщение фиксации для комбинированных патчей/ревизий после завершен свёртки.

Опции:

- `-e`: Изменить сообщение фиксации и описание патча для недавно свёрнутого патча.
- `-l`: использовать содержимое заданного файла в качестве нового сообщения фиксации и описания свёрнутого патча.
- `-m`: Используйте данный текст в качестве нового сообщения фиксации и описание патча для сложенного патча.

## B.1.7. qheader — отображает заголовки/описание патча

Команда **qheader** печатает заголовок или описание, из патча. По умолчанию, она печатает заголовок применяемого верхнего патча. С учетом аргументов, оно выводит заголовок именованного патча.

## B.1.8. qimport — импорт сторонних патчей в очередь

Команда **qimport** добавляет запись для внешнего патча в файле `series`, а также копирует патч в каталог `.hg/patches`. Она добавляет запись сразу же после верхнего примененного патча, но не вставляет патч.

Если каталог `.hg/patches` репозиторий, **qimport** автоматически выполняет **hg add** импортируемого патча.

## B.1.9. qinit — подготовить хранилище для работы с MQ

Команда **qinit** готовит репозиторий для работы с MQ. Она создает каталог с именем `.hg/patches`.

Опции:

- `-c`: Создает `.hg/patches`, как репозиторий под своим собственным контролем. Также создается файл `.hgignore`, для игнорирования файла `status`.

Если директория `.hg/patches` репозиторий, команды **qimport** и **qnew** автоматически запускают **hg add** для новых патчей.

## B.1.10. qnew — создание новых патчей

Команда **qnew** создаёт новый патч. Она принимает один обязательный аргумент, имя использующееся для файла патча. Вновь созданный патч создается по-умолчанию пустым. Он добавляется в файл `series` после текущего верхнего применённого патча, и сразу вставляется верхним патчем.

Если **qnew** находит измененные файлы в рабочем каталоге, она откажется от создания нового патча, если не используется опция `-f` (смотрите ниже). Такое поведение позволяет вызвать **qrefresh** для вашего верхнего применённого патча, перед тем как применить новый патч поверх него.

Опции:

- `-f`: Создаёт новый патч, если содержимое рабочего каталога изменено. Любые неразрешенные модификаций добавляются в только что созданный патч, так что после того как команда завершает свою работу, рабочий каталог больше не будут измененым.
- `-m`: Использование данного текста как сообщения фиксации. Этот текст будет храниться в начале файла заплатки, до данных патча.

## B.1.11. qnext — печатает имя следующего патча

Команда **qnext** печатает имя следующего патча в файле `series` после верхнего применяемого патча. Этот патч будет применён, если вы запустите **qpush**.

## B.1.12. qpop— извлекает патчи из стека

Команда **qpop** удаляет применённые патчи с вершины стека применённых патчей. По умолчанию, она удаляет только один патч.

Эта команда удаляет наборы изменений, которые представляют извлечённые патчи из репозитория, и обновляет рабочий каталог, чтобы устранить последствия патчей.

Эта команда принимает необязательный аргумент, который она использует в качестве имени или индекса патча для извлечения. Если имя передаётся, патчи будут применяться пока патч, имя которого было передано, не станет верхним. Если передаётся номер, **qpop** использует номер в качестве индекса записи в файле серий, начиная с нуля (пустые строки и строки, содержащие только комментарии не в учитываются). Она извлекает патчи пока выявленный патч не станет самым верхним применённым патчем.

Команда **qpop** не читает или пишет патчи или файл `series`. Таким образом, безопасно извлечь патч, который вы удалили из файла `series` или патч, который был переименован или удален полностью. В двух последних случаях, используйте имя патча, как это было, когда вы применили его.

По умолчанию, команда **qpop** не выполнится для любого патча, если рабочий каталог был изменен. Вы можете изменить это поведение, используя опцию `-f`, которая возвращает все изменения в рабочий каталог.

Опции:

- `-a`: Извлекает все применённые патчи. Это вернет репозиторий в состояние до установки патчей.
- `-f`: Принудительно восстановить любые изменения в рабочем каталоге, при их извлечении.
- `-n`: Извлекает патч из именованной очереди.

Команда **qpop** удаляет одну строку с конца файла `status` для каждого патча, который она извлекает.

## B.1.13. qprev — печатает имя предыдущего патча

Команда **qprev** печатает название патча в файле `series`, который находится перед верхним применённым патчем. Он станет самым верхним патчем применённым при запуске **qpop**.

## B.1.14. qpush — вставляет патчи в стек

Команда **qpush** добавляет патчи на верх стека применённых патчей. По умолчанию, она добавляет только один патч.

Эта команда создает новую ревизию для представления каждого применённого патча и обновляет рабочую директорию применяя патч.

По умолчанию данные используемые при создании ревизии, следующие:

- Текущая дата и часовой пояс коммитера. Так как эти данные используются для вычисления идентификатора ревизии, это означает, что если вы извлечёте (**qpop**) патч и добавите (**qpush**) его снова, ревизия, которые вы вставите будет иметь другой идентификатор, чем та ревизия которую вы удалили.
- Автор который используется по умолчанию при использовании команды **hg commit**.
- Сообщение фиксации — любой текст из файла патча, который находится перед первым заголовком diff. Если нет такого текста, по умолчанию используется сообщение, которое идентифицирует имя патча.

Если патч содержит заголовок Mercurial для патча, информация в заголовке патча перекрывает все эти умолчания.

Опции:

- `-a`: вставить все не применённые патчи из файла `series`, пока есть что вставлять.
- `-l`: Добавить название патча к концу сообщения фиксации.
- `-m`: Если патч не удастся применить аккуратно, использовать запись исправлений в другой сохранённой очереди, чтобы вычислить параметры трехстороннего слияния, а также выполнить трехстороннее слияние с использованием обычного механизма слияния Mercurial. Использовать результат слияния в качестве нового содержимого патча.
- `-n`: Использовать имя очереди, если слияние происходит во время вставки.

Команда **qpush** читает, но не изменяет, файл `series`. Он добавляет одну строку в файл **hg status** для каждого патча, который она вставляет.

## B.1.15. qrefresh — обновление верхнего применённого патча

Команда **qrefresh** обновляет верхний применённый патч. Она изменяет патч, удаляет старую ревизию, которая представляет патч, и создает новую ревизию для представления изменённого патча.

Команда **qrefresh** просматривает следующие изменения:

- Изменения в сообщении фиксации, то есть текст до первого заголовка различий в патче, отражены в новом наборе изменений, которая представляет собой патч.
- Изменения в отслеживаемых файлах в рабочем каталоге добавляются в патч.
- Изменения файлов отслеживаемых с помощью **hg add**, **hg copy**, **hg remove**, или **hg rename**. Добавляет файлы и копирует и переименовывает целевой патч, в то время как удаленные файлы и переименованные удаляются.

Даже если **qrefresh** обнаруживает отсутствие изменений, она по-прежнему переписывает ревизию, которая представляет собой патч. Это приводит к изменению идентификатора отличающегося от предыдущих ревизий, которые идентифицируют патч.

Опции:

- `-e`: Изменение сообщения фиксации и описания патча, используя текстовый редактор.
- `-m`: Изменение сообщения фиксации и описания патча, используя передаваемый текст.
- `-l`: Изменение сообщения фиксации и описания патча, используя текст из переданного файла.

## B.1.16. qrename — переименование патча

Команда qrename переименовывает патч и изменяет запись для патча в файле `series`.

С помощью единственного аргумента, **qrename** переименовывает верхний применённый патч. С двумя аргументами, переименовывает патч из первого аргумента ко второму.

## B.1.17. qseries — печатает записи серии патчей

Команда **qseries** выводит целый ряд патчей из файла `series`. Он печатает только имена патчей, а не пустые строки или комментарии. Он печатает в порядке от первого к последнему применённому патчу.

## B.1.18. qtop— печатает имя текущего патча

Команда **qtop** печатает имя верхнего примененного в настоящее время патча.

### B.1.19. qunapplied— печатает не применённые патчи

Команда **qunapplied** печатает имена патчей из файла `series`, которые пока еще не применены. Он выводит их в порядке от следующего патча, которые будет вставлен до последнего.

### B.1.20. hg strip — удаляет ревизию и потомков

Команда **hg strip** удаляет ревизию, и всех её потомков, из репозитория. Она отменяет последствия изменений удаленной из репозитория, и обновляет рабочую директорию на первого родителя удалённой ревизии.

Команда **hg strip** сохраняет резервные копии удаленных ревизий в пакете, так что они могут быть повторно применены, если по ошибке удалены.

Опции:

- `-b`: Сохранить несвязанные ревизии, смешанные с ревизиями в резервный комплект.
- `-f`: если ветвь имеет несколько голов, удаляет все головы.
- `-n`: Не сохранять резервный комплект.

## B.2. Справочник файлов MQ

### B.2.1. Файл `series`

Файл `series` содержит список имен из всех патчей, которые MQ может применить. Он представляется в виде списка имен, с одним именем в каждой строке. Начальные и конечные пробелы в строке игнорируются.

Строки могут содержать комментарии, которые начинается с символа «#» и продолжается до конца строки. Пустые строки и строки, содержащие только комментарии, игнорируются.

Вам часто придется изменять ряд файлов вручную, поэтому поддерживаются комментарии и пустые строки, упомянутые выше. Например, вы можете временно закомментировать патч, и **qpush** будет пропускать этот патч при применении исправлений. Также вы можете изменить последовательность применения патчей, изменяя их порядок в файле `series`.

Размещение файла `series` под контролем версий также поддерживается. Это хорошая идея, чтобы поставить все патчи, которые к ней относятся, под контроль версий. Если вы создаете каталог с помощью опции `-c` команды **qinit**, это будет сделано автоматически.

### B.2.2. Файл `status`.

Файл `status` содержит имена и хеши ревизий всех патчей, которые в настоящее время могут быть применены MQ. В отличие от файла `series`, этот файл не предназначен для редактирования. Вы не должны помещать этот файл под контроль версий, или изменять его в любом случае. Он используется MQ строго для внутреннего учета.

# Приложение C. Установка Mercurial из исходников

## C.1. На Unix-подобных системах

Если вы используете Unix-подобную операционную систему, которая имеет достаточно свежую версию python (версии 2.3 или более поздней), установка Mercurial из исходных текстов проста.

1. Скачайте последний архив исходников из http://www.selenic.com/mercurial/download.

2. Распакуйте архив исходников:

```
gzip -dc mercurial-MYVERSION.tar.gz | tar xf -
```

3. Зайдите в директорию с исходными текстами и запустите скрипт установки. Он соберёт Mercurial и установит его в вашем домашнем каталоге.

```
cd mercurial-MYVERSION
python setup.py install --force --home=$HOME
```

После завершения установки Mercurial будет в подкаталоге `bin` домашнего каталога. Не забудьте убедиться, что эта директория находится на пути поиска вашей оболочки.

Вам, вероятно, необходимо будет установить переменную окружения `PYTHONPATH` так чтобы Mercurial при запуске мог найти остальные пакеты Mercurial. Например, на моем ноутбуке, я установил ее в `/home/bos/lib/python`. Точный путь, который вам нужно использовать, зависит от того, как Python был собран для вашей системы, но должен легко выяснятся. Если вы не уверены, посмотрите вывод скрипта установки и увидите, куда был установлен `mercurial`.

## C.2. На Windows

Сборка и установка Mercurial на Windows требует разнообразных инструментов, достаточного количества технических знаний и значительного терпения. Я очень не рекомендую этот путь, если вы «обычный пользователь». Если вы не собираетесь взломать Mercurial, я настоятельно рекомендую использовать бинарный пакет.

Если вы намерены собирать Mercurial из исходников под Windows, используйте «трудный путь» описаный на вики Mercurial http://www.selenic.com/mercurial/wiki/index.cgi/WindowsInstall, и имейте ввиду, что этот процесс утомителен.

# Приложение D. Open Publication License

Версия 1.0, 8 Июня 1999

## D.1. Требования в обоих немодифицированной и модифицированной версии

Открытые публикации могут воспроизводиться и распространяться целиком или частично на любом физическом или электронном носителе при условии точного соблюдения условий настоящей лицензии, и при включении полного текста настоящей лицензии или ссылки на неё (с любыми допустимыми дополнительными ограничениями, наложенными автором(ами) и/или издателем).

Ссылка должна быть приведена в следующем форме

> Copyright (c) год by имя автора или разработчика. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v**X.Y** or later (the latest version is presently available at http://www.opencontent.org/openpub/).

Непосредственно после ссылки должны быть приведены дополнительные ограничения, наложенные автором(ами) и/или издателем документа (см. раздел Раздел D.6, «Дополнительные ограничения»).

Допускается коммерческое распространение материалов, распространяющихся на условии Лицензии открытых публикаций.

Любая публикация в форме обычной (бумажной) книги требует указания первых издателя и автора. Имена издателя и автора должны присутствовать на всех внешних поверхностях книги. На всех внешних поверхностях книги имя первого издателя должно быть набрано шрифтом не меньшего размера, чем шрифт, которым набран заголовок произведения, и должно быть согласовано с заголовком произведения в притяжательном падеже.

## D.2. Исключительное авторское право

Исключительное авторское право на любую открытую публикацию принадлежит её автору(ам) или разработчику(ам).

## D.3. Отношения, регулируемые лицензией

Следующие условия лицензии применяются ко всем произведениям, распространяющимся на условиях настоящей лицензии, если иное не оговорено в документе явно.

Объединение произведения, распространяющегося на условиях настоящей лицензии, или части такого произведения, с другими произведениями (в т.ч. программами) на одном носителе само по себе не приводит к применению условий лицензии к таким произведениям. Результат объединения должен содержать уведомление, указывающее на присутствие материалов, распространяющихся на условиях Лицензии открытых публикаций, и соответствующее уведомление об авторских правах.

Частичное действие. Если какая-либо часть настоящей лицензии в каком-либо правовом пространстве признаётся не имеющей юридической силы, все остальные части лицензии по-прежнему сохраняют силу.

Отсутствие гарантий. Открытые публикации лицензируются и распространяются «как есть», без каких-либо гарантий, явных или подразумеваемых, включая, но не ограничиваясь ими, подразумеваемые гарантии коммерческого успеха и пригодности в конкретных целях или гарантии отсутствия нарушений законодательства.

## D.4. Требования к модифицированным копиям

Все модифицированные версии документов, распространяющихся на условиях настоящей лицензии, включая переводы, антологии, компиляции и фрагменты документов, должны соответствовать следующим требованиям:

1. Модифицированная версия должна быть явно обозначена как модифицированная.

2. Лицо, произведшее модификацию, должно быть однозначно указано, модификация датирована.

3. Если это возможно, должны быть сохранены ссылки на первых автора и издателя в соответствии с общепринятой в научной среде практикой цитирования.

4. Должно быть указано расположение оригинального не модифицированного документа.

5. Имя (имена) первого автора(ов) произведения не может использоваться в качестве подтверждения принадлежности первому автору данной редакции произведения без его (их) явного разрешения.

# D.5. Рекомендации

Помимо обязательных требований, настоящая лицензия содержит просьбу и настоятельную рекомендацию к лицам, распространяющим произведения на её условиях:

1. Если вы распространяете открытые публикации на физическом носителе или компакт-диске, отправьте уведомление о вашем намерении авторам по крайней мере за 30 дней до передачи в тираж, чтобы они могли предоставить вам обновлённую версию публикации. В такое уведомление следует включить также описание модификаций, сделанных в документе, если таковые есть.

2. Все существенные модификации документа (включая удаление его части) должны быть либо явно отмечены в его тексте, либо описаны в приложении к документу.

3. Наконец, хотя лицензия и не обязывает вас делать это, хорошим тоном считается предоставить бесплатный авторский экземпляр произведения на физическом носителе или компакт-диске его автору(ам).

# D.6. Дополнительные ограничения

Авторы(ы) и/или издатель произведения, распространяющегося на условиях Лицензии открытых публикаций, могут наложить некоторые дополнительные ограничения на условия их распространения, добавив соответствующее требование к ссылке на лицензию или её копии. Эти ограничения рассматриваются как часть текста лицензии и должны быть включены в любые производные произведения вместе с текстом лицензии (или ссылкой на неё).

1. Запрещение распространение существенно модифицированных версий без явного разрешения автора(ов). «Существенно модифицированными» признаются изменения смыслового содержания документа и не признаются изменения форматирования текста или типографская корректура.

2. Для наложения этого ограничения добавьте к ссылке на лицензию или её копии предложение «Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder».

3. Запрещение публикации оригинального или производного произведения целиком или частично в форме обычной (бумажной) книги в коммерческих целях без предварительного разрешения правообладателя.

4. Для наложения этого ограничения добавьте к ссылке на лицензию или её копии предложение «Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.».