Модель. ActiveRecord. База данных. Миграции. Ассоциации. Валидация

Active Record это M в [MVC](http://rusrails.ru/getting_started.html#the-mvc-architecture) - модель - которая является слоем в системе, ответственным за представление бизнес-логики и данных. Active Record упрощает создание и использование бизнес-объектов, данные которых требуют постоянного хранения в базе данных. Сама по себе эта реализация паттерна Active Record является описанием системы ORM (Object Relational Mapping)

**Паттерн Active Record**

[Active Record был описан Martin Fowler](http://www.martinfowler.com/eaaCatalog/activeRecord.html) в его книге *Patterns of Enterprise Application Architecture*. В Active Record объекты содержат и сохраненные данные, и поведение, которое работает с этими данными. Active Record исходит из мнения, что обеспечение логики доступа к данным как части объекта покажет пользователям этого объекта то, как читать и писать в базу данных

**Object Relational Mapping (ORM)**

Object-Relational Mapping, обычно упоминающееся как аббревиатура ORM, это техника, соединяющая сложные объекты приложения с таблицами в системе управляения реляционными базами данных. С использованием ORM, свойства и взаимоотношения этих объектов приложения могут быть с легкостью сохранены и получены из базы данных без непосредственного написания выражений SQL, и, в итоге, с меньшим суммарным кодом для доступа в базу данных.

**Active Record это фреймворк ORM**

Active Record предоставляет нам несколько механизмов, наиболее важными из которых является способности для:

* Представления моделей и их данных
* Представления связей между этими моделями
* Представления иерархий наследования с помощью связанных моделей
* Валидации моделей до того, как они будут сохранены в базу данных
* Выполнения операций с базой данных в объктно-ориентированном стиле.

**Соглашения над конфигурацией в Active Record**

При написании приложения с использованием других языков программирования или фреймворков часто требуется писать много конфигурационного кода. В частности, это справедливо для фреймворков ORM. Однако, если следовать соглашениям, принятым Rails, вам придется написать совсем немного конфигураций (а иногда совсем не придется) при создании моделей Active Record. Идея в том, что в большинстве случаев вы настраиваете свои приложения одинаковым образом, и этот способ должен быть способом по умолчанию. Таким образом, явная конфигурация потребуется только тогда, когда вы не следуете соглашениям по какой-то причине.

**Соглашения по именованию**

По умолчанию Active Record использует некоторые соглашения по именованию чтобы узнать, как должна быть создана связь между моделями и таблицами базы данных. Rails образует множественное число для имен класса, чтобы найти соответствующую таблицу базы данных. Так, для класса Book следует создать таблицу базы данных с именем **books**. Механизмы образования множественного числа Rails очень мощные, они способны образовывать множественное (и единственное) число как для правильных, так и для неправильных слов. При использовании имен класса, созданных из двух и более слов, имя класса модели должно следовать соглашениям Ruby, используя форму CamelCase, тогда как имя таблицы должно содержать слова, разделенные знаком подчеркивания. Примеры:

* Таблица базы данных - Множественная форма со словами, разделенными знаком подчеркивания (т.е., book\_clubs)
* Класс модели - Единственное число с первой прописной буквой в каждом слове (т.е., BookClub)

|  |  |
| --- | --- |
| **Модель / Класс** | **Таблица / Схема** |
| Post | posts |
| LineItem | line\_items |
| Deer | deer |
| Mouse | mice |
| Person | people |

**Соглашения схемы**

Active Record использует соглашения о именовании для столбцов в таблицах базы данных, зависящих от назначения этих столбцов.

* **Внешние ключи** - Эти поля должны именоваться по образцу singularized\_table\_name\_id (т.е., item\_id, order\_id). Это поля, которые ищет Active Record при создании связей между вашими моделями.
* **Первичные ключи** - По умолчанию Active Record использует числовой столбец с именем id как первичный ключ таблицы. Этот столбец будет автоматически создан при использовании [миграций Rails](http://rusrails.ru/rails-database-migrations) для создания таблиц.

Также имеются некоторые опциональные имена столбцов, добавляющие дополнительные особенности для экземпляров Active Record:

* created\_at - Автоматически будут установлены текущие дата и время при изначальном создании записи.
* updated\_at - Автоматически будут установлены текущие дата и время всякий раз, когда обновляется запись.
* lock\_version - Добавляет [оптимистичную блокировку](http://api.rubyonrails.org/classes/ActiveRecord/Locking.html) к модели.
* type - Указывает, что модель использует [Single Table Inheritance](http://api.rubyonrails.org/classes/ActiveRecord/Base.html)
* (association\_name)\_type - Хранит тип для [полиморфных связей](http://rusrails.ru/active-record-associations#polymorphic-associations).
* (table\_name)\_count - Используется для кэширования количества принадлежных по связи объектов. Например, столбец comments\_count в классе Post, у которого может быть несколько связанных экземпляров Comment, закэширует количество существующих комментариев для каждой публикации.

Хотя эти имена столбцов опциональны, фактически они зарезервированы Active Record. Избегайте зарезервированных ключевых слов, если вы не желаете дополнительной функциональности. Например, type - это зарезервированное слово для определения таблицы, использующей Single Table Inheritance (STI). Если вы не используете STI, попытайтесь использовать аналогичное слово, такое как "context", которое также может аккуратно описать данные, которые вы моделируете.

**Создание моделей Active Record**

Создавать модели Active Record очень просто. Все, что необходимо сделать, - это создать подкласс ActiveRecord::Base, и готово:

|  |
| --- |
| class Product < ActiveRecord::Base  end |

Это создаст модель Product, связав ее с таблицей products в базе данных. Сделав так, также появится способность связать столбцы каждой строки этой таблицы с атрибутами экземпляров вашей модели. Допустим, что таблица products была создана с использованием такого выражения SQL:

|  |
| --- |
| CREATE TABLE products (     id int(11) NOT NULL auto\_increment,     name varchar(255),     PRIMARY KEY  (id)  ); |

Следуя вышеуказанной схеме, можно будет писать подобный код:

|  |
| --- |
| p = Product.new  p.name = "Some Book"  puts p.name # "Some Book" |

**Переопределение соглашений о именовании**

Но что, если вы следуете другому соглашению по именованию или используете новое приложение Rails со старой базой данных? Не проблема, можно просто переопределить соглашения по умолчанию.

Можно использовать метод ActiveRecord::Base.table\_name= для указания имени таблицы, которая должна быть использована:

|  |
| --- |
| class Product < ActiveRecord::Base    self.table\_name = "PRODUCT"  end |

Если так сделать, нужно вручную определить имя класса, содержащего фикстуры (class\_name.yml), используя метод set\_fixture\_class в определении теста:

|  |
| --- |
| class FunnyJoke < ActiveSupport::TestCase    set\_fixture\_class funny\_jokes: Joke    fixtures :funny\_jokes    ...  end |

Также возможно переопределить столбец, который должен быть использован как первичный ключ таблицы, с помощью метода ActiveRecord::Base.primary\_key=:

|  |
| --- |
| class Product < ActiveRecord::Base    self.primary\_key = "product\_id"  end |

**CRUD: Чтение и запись данных**

CRUD это сокращение для четерех глаголов, используемых для описания операций с данными: **C**reate (создать), **R**ead (прочесть), **U**pdate (обновить) и **D**elete (удалить). Active Record автоматически создает методы, позволяющие приложению читать и воздействовать на данные, хранимые в своих таблицах.

**Создание**

Объекты Active Record могут быть созданы из хэша, блока или из вручную указанных после создания атрибутов. Метод new возвратит новый объект, в то время как create возвратит объект и сохранит его в базу данных.

Например, для модели User с атрибутами name и occupation, вызов метода create создаст и сохранит новую запись в базу данных:

|  |
| --- |
| user = User.create(name: "David", occupation: "Code Artist") |

Используя метод new, объект может быть инициализирован без сохранения:

|  |
| --- |
| user = User.new  user.name = "David"  user.occupation = "Code Artist" |

Вызов user.save передаст запись в базу данных.

Наконец, если представлен блок и create, и new передадут новый объект в этот блок для инициализации:

|  |
| --- |
| user = User.new do |u|    u.name = "David"    u.occupation = "Code Artist"  end |

**Чтение**

Active Record предоставляет богатый API для доступа к данным в базе данных. Ниже несколько примеров различных методов доступа к данным, представленных Active Record.

|  |
| --- |
| # возвратит коллекцию со всеми пользователями  users = User.all |
| # возвратит первого пользователя  user = User.first |

|  |
| --- |
| # возвратит первого пользователя с именем David  david = User.find\_by(name: 'David') |
| # найдет всех пользователей с именем David, которые Code Artists, и сортирует их по created\_at в обратном хронологическом порядке  users = User.where(name: 'David', occupation: 'Code Artist').order('created\_at DESC') |

Подробно о запросах в моделях Active Record можно узнать в руководстве [Интерфейс запросов Active Record](http://rusrails.ru/active-record-query-interface).

**Обновление**

Как только объект Active Record будет получен, его атрибуты могут быть изменены, и он может быть сохранен в базу данных.

|  |
| --- |
| user = User.find\_by(name: 'David')  user.name = 'Dave'  user.save |

Сокращенным вариантом для этого является использование хэша с атрибутами, связанными с желаемыми значениями, таким образом:

|  |
| --- |
| user = User.find\_by(name: 'David')  user.update(name: 'Dave') |

Это наиболее полезно, когда необходимо обновить несколько атрибутов за раз. Если, с другой стороны, необходимо обновить несколько записей за раз, полезен метод класса update\_all:

|  |
| --- |
| User.update\_all "max\_login\_attempts = 3, must\_change\_password = 'true'" |

**Удаление**

Более того, после получения, объект Active Record может быть уничтожен, что уберет его из базы данных.

|  |
| --- |
| user = User.find\_by(name: 'David')  user.destroy |

**Валидации**

Active Record позволяет проверять состояние модели до того, как она будет записана в базу данных. Имеется несколько методов, которые могут быть использованы для проверки ваших моделей и валидации, что значение атрибута не пустое, уникальное (не существующее в базе данных), отвечает определенному формату, и многие другие.

Валидация - это очень важный вопрос, который нужно рассмотреть при сохранении в базу данных, поэтому методы create, save и update учитывают ее при запуске: они возвращают false, когда валидация проваливается, и фактически они не выполняют каких-либо операций с базой данных. Каждый из этих методов имеет пару с восклицательным знаком (create!, save! и update!), которые строже в том, что они вызывают исключение ActiveRecord::RecordInvalid если валидация провалится. Краткий пример:

|  |
| --- |
| class User < ActiveRecord::Base    validates :name, presence: true  end    User.create  # => false  User.create! # => ActiveRecord::RecordInvalid: Validation failed: Name can't be blank |

**Обзор валидаций**

Вот пример очень простой валидации:

|  |
| --- |
| class Person < ActiveRecord::Base    validates :name, presence: true  end    Person.create(name: "John Doe").valid? # => true  Person.create(name: nil).valid? # => false |

Как видите, наша валидация позволяет узнать, что наш Person не валиден без атрибута name. Второй Person не будет сохранен в базу данных.

Прежде чем погрузиться в подробности, давайте поговорим о том, как валидации вписываются в общую картину приложения.

**Зачем использовать валидации?**

Валидации используются, чтобы быть уверенными, что только валидные данные сохраняются в вашу базу данных. Например, для вашего приложения может быть важно, что каждый пользователь предоставил валидный электронный и почтовый адреса. Валидации на уровне модели - наилучший способ убедиться, что в базу данных будут сохранены только валидные данные. Они не зависят от базы данных, не могут быть обойдены конечными пользователями и удобны в тестировании и обслуживании. Rails представляет простоту в обслуживании, представляет встроенные хелперы для общих нужд, а также позволяет создавать свои собственные методы валидации.

Есть несколько способов валидации данных, прежде чем они будут сохранены в вашу базу данных, включая ограничения, встроенные в базу данных, валидации на клиентской части, валидации на уровне контроллера и валидации на уровне модели. Вкратце о плюсах и минусах:

* Ограничения базы данных и/или хранимые процедуры делают механизмы валидации зависимыми от базы данных, что делает тестирование и поддержку более трудными. Однако, если ваша база данных используется другими приложениями, валидация на уровне базы данных может безопасно обрабатывать некоторые вещи (такие как уникальность в нагруженных таблицах), которые затруднительно выполнять по другому.
* Валидации на клиентской части могут быть очень полезны, но в целом ненадежны, если используются в одиночку. Если они используют JavaScript, они могут быть пропущены, если JavaScript отключен в клиентском браузере. Однако, если этот способ комбинировать с другими, валидации на клиентской части могут быть удобным способам предоставить пользователям немедленную обратную связь при использовании вашего сайта.
* Валидации на уровне контроллера заманчиво делать, но это часто приводит к громоздкости и трудности тестирования и поддержки. Во всех случаях, когда это возможно, держите свои контроллеры 'тощими', тогда с вашим приложением будет приятно работать в долгосрочной перспективе.

Выбирайте их под свои определенные специфичные задачи. Общее мнение команды Rails состоит в том, что валидации на уровне модели - наиболее подходящая во многих случаях.

**Когда происходит валидация?**

Есть два типа объектов Active Record: те, которые соответствуют строке в вашей базе данных, и те, которые нет. Когда создаете новый объект, например, используя метод new, этот объект еще не привязан к базе данных. Как только вы вызовете save. этот объект будет сохранен в подходящую таблицу базы данных. Active Record использует метод экземпляра new\_record? для определения, есть ли уже объект в базе данных или нет. Рассмотрим следующий простой класс Active Record:

|  |
| --- |
| class Person < ActiveRecord::Base  end |

Можно увидеть, как он работает, взглянув на результат rails console:

|  |
| --- |
| $ rails console  >> p = Person.new(name: "John Doe")  => #<Person id: nil, name: "John Doe", created\_at: nil, updated\_at: nil>  >> p.new\_record?  => true  >> p.save  => true  >> p.new\_record?  => false |

Создание и сохранение новой записи посылает операцию SQL INSERT базе данных. Обновление существующей записи вместо этого посылает операцию SQL UPDATE. Валидации обычно запускаются до того, как эти команды посылаются базе данных. Если любая из валидаций проваливается, объект помечается как недействительный и Active Record не выполняет операцию INSERT или UPDATE. Это помогает избежать хранения невалидного объекта в базе данных. Можно выбирать запуск специфичных валидаций, когда объект создается, сохраняется или обновляется.

Есть разные методы изменения состояния объекта в базе данных. Некоторые методы вызывают валидации, некоторые нет. Это означает, что возможно сохранить в базу данных объект с недействительным статусом, если вы будете не внимательны.

Следующие методы вызывают валидацию, и сохраняют объект в базу данных только если он валиден:

* create
* create!
* save
* save!
* update
* update!

Версии с восклицательным знаком (т.е. save!) вызывают исключение, если запись недействительна. Невосклицательные версии не вызывают, save и update возвращают false, create возвращает объект.

**Пропуск валидаций**

Следующие методы пропускают валидации, и сохраняют объект в базу данных, независимо от его валидности. Их нужно использовать осторожно.

* decrement!
* decrement\_counter
* increment!
* increment\_counter
* toggle!
* touch
* update\_all
* update\_attribute
* update\_column
* update\_columns
* update\_counters

Заметьте, что save также имеет способность пропустить валидации, если как передать validate: false как аргумент. Этот способ нужно использовать осторожно.

* save(validate: false)

**valid? или invalid?**

Чтобы определить, валиден объект или нет, Rails использует метод valid?. Вы также можете его использовать для себя. valid? вызывает ваши валидации и возвращает true, если ни одной ошибки не было найдено у объекта, иначе false.

|  |
| --- |
| class Person < ActiveRecord::Base    validates :name, presence: true  end    Person.create(name: "John Doe").valid? # => true  Person.create(name: nil).valid? # => false |

После того, как Active Record выполнит валидации, все найденные ошибки будут доступны в методе экземпляра errors.messages, возвращающем коллекцию ошибок. По определению объект валиден, если эта коллекция будет пуста после запуска валидаций.

Заметьте, что объект, созданный с помощью new не сообщает об ошибках, даже если технически невалиден, поскольку валидации не запускаются при использовании new.

|  |
| --- |
| class Person < ActiveRecord::Base    validates :name, presence: true  end    >> p = Person.new  #=> #<Person id: nil, name: nil>  >> p.errors.messages  #=> {}    >> p.valid?  #=> false  >> p.errors.messages  #=> {name:["can't be blank"]}    >> p = Person.create  #=> #<Person id: nil, name: nil>  >> p.errors.messages  #=> {name:["can't be blank"]}    >> p.save  #=> false    >> p.save!  #=> ActiveRecord::RecordInvalid: Validation failed: Name can't be blank    >> Person.create!  #=> ActiveRecord::RecordInvalid: Validation failed: Name can't be blank |

invalid? это просто антипод valid?. Он запускает ваши валидации, возвращая true, если для объекта были добавлены ошибки, и false в противном случае.

**errors[]**

Чтобы проверить, является или нет конкретный атрибут объекта валидным, можно использовать errors[:attribute], который возвращает массив со всеми ошибками атрибута, когда нет ошибок по определенному атрибуту, возвращается пустой массив.

Этот метод полезен только *после того*, как валидации были запущены, так как он всего лишь исследует коллекцию errors, но сам не вызывает валидации. Он отличается от метода ActiveRecord::Base#invalid?, описанного выше, тем, что не проверяет валидность объекта в целом. Он всего лишь проверяет, какие ошибки были найдены для отдельного атрибута объекта.

|  |
| --- |
| class Person < ActiveRecord::Base    validates :name, presence: true  end    >> Person.new.errors[:name].any? # => false  >> Person.create.errors[:name].any? # => true |

Мы рассмотрим ошибки валидации подробнее в разделе [Работаем с ошибками валидации](http://rusrails.ru/active-record-validations#working-with-validation-errors). А сейчас обратимся к встроенным валидационным хелперам, предоставленным Rails по умолчанию.

**Валидационные хелперы**

Active Record предлагает множество предопределенных валидационных хелперов, которые Вы можете использовать прямо внутри Ваших определений класса. Эти хелперы предоставляют общие правила валидации. Каждый раз, когда валидация проваливается, сообщение об ошибке добавляется в коллекцию errors объекта, и это сообщение связывается с атрибутом, который подлежал валидации.

Каждый хелпер принимает произвольное количество имен атрибутов, поэтому в одной строке кода можно добавить валидации одинакового вида для нескольких аттрибутов.

Они все принимают опции :on и :message, которые определяют, когда валидация должна быть запущена, и какое сообщение должно быть добавлено в коллекцию errors, если она провалится. Опция :on принимает одно из значений :create или :update. Для каждого валидационного хелпера есть свое сообщение об ошибке по умолчанию. Эти сообщения используются, если не определена опция :message. Давайте рассмотрим каждый из доступных хелперов.

**acceptance**

Этот метод проверяет, что чекбокс в пользовательском интерфейсе был нажат, когда форма была подтверждена. Обычно используется, когда пользователю нужно согласиться с условиями использования Вашего приложения, подтвердить прочтение некоторого текста или выполнить любое подобное действие. Валидация очень специфична для веб приложений, и ее принятие не нужно записывать куда-либо в базу данных (если у вас нет поля для него, хелпер всего лишь создаст виртуальный аттрибут).

|  |
| --- |
| class Person < ActiveRecord::Base    validates :terms\_of\_service, acceptance: true  end |

Для этого хелпера сообщение об ошибке по умолчанию следующее *"must be accepted"*.

Он может получать опцию :accept, которая определяет значение, которое должно считаться принятым. По умолчанию это "1", но его можно изменить.

|  |
| --- |
| class Person < ActiveRecord::Base    validates :terms\_of\_service, acceptance: { accept: 'yes' }  end |

**validates\_associated**

Этот хелпер можно использовать, когда у вашей модели есть связи с другими моделями, и их также нужно проверить на валидность. Когда вы пытаетесь сохранить свой объект, будет вызван метод valid? для каждого из связанных объектов.

|  |
| --- |
| class Library < ActiveRecord::Base    has\_many :books    validates\_associated :books  end |

Эта валидация работает со всеми типами связей.

Не используйте validates\_associated на обоих концах ваших связей, они будут вызывать друг друга в бесконечном цикле.

Для validates\_associated сообщение об ошибке по умолчанию следующее *"is invalid"*. Заметьте, что каждый связанный объект имеет свою собственную коллекцию errors; ошибки не добавляются к вызывающей модели.

**confirmation**

Этот хелпер можно использовать, если у вас есть два текстовых поля, из которых нужно получить полностью идентичное содержание. Например, вы хотите подтверждение адреса электронной почты или пароля. Эта валидация создает виртуальный атрибут, имя которого равно имени подтверждаемого поля с добавлением "\_confirmation".

|  |
| --- |
| class Person < ActiveRecord::Base    validates :email, confirmation: true  end |

В вашем шаблоне вьюхи нужно использовать что-то вроде этого

<%= text\_field :person, :email %>

<%= text\_field :person, :email\_confirmation %>

Эта проверка выполняется, только если email\_confirmation не равно nil. Чтобы требовать подтверждение, нужно добавить еще проверку на существование проверяемого аттрибута (мы рассмотрим presence чуть позже):

class Person < ActiveRecord::Base

validates :email, confirmation: true

validates :email\_confirmation, presence: true

end

По умолчанию сообщение об ошибке для этого хелпера такое *"doesn't match confirmation"*.

**exclusion**

Этот хелпер проводит валидацию того, что значения аттрибутов не включены в указанный набор. Фактически, этот набор может быть любым перечисляемым объектом.

class Account < ActiveRecord::Base

validates :subdomain, exclusion: { in: %w(www us ca jp),

message: "Subdomain %{value} is reserved." }

end

Хелпер exclusion имеет опцию :in, которая получает набор значений, которые не должны приниматься проверяемыми аттрибутами. Опция :in имеет псевдоним :within, который используется для тех же целей. Этот пример использует опцию :message, чтобы показать вам, как можно включать значение аттрибута.

Значение сообщения об ошибке по умолчанию "\_is reserved\_".

**format**

Этот хелпер проводит валидацию значений аттрибутов, тестируя их на соответствие указанному регулярному выражению, которое определяется с помощью опции :with.

class Product < ActiveRecord::Base

validates :legacy\_code, format: { with: /\A[a-zA-Z]+\z/,

message: "only allows letters" }

end

Значение сообщения об ошибке по умолчанию "\_is invalid\_".

**inclusion**

Этот хелпер проводит валидацию значений аттрибутов на включение в указанный набор. Фактически этот набор может быть любым перечисляемым объектом.

class Coffee < ActiveRecord::Base

validates :size, inclusion: { in: %w(small medium large),

message: "%{value} is not a valid size" }

end

Хелпер inclusion имеет опцию :in, которая получает набор значений, которые должны быть приняты. Опция :in имеет псевдоним :within, который используется для тех же целей. Предыдущий пример использует опцию :message, чтобы показать вам, как можно включать значение аттрибута.

Значение сообщения об ошибке по умолчанию для этого хелпера такое "\_is not included in the list\_".

**length**

Этот хелпер проводит валидацию длины значений аттрибутов. Он предлагает ряд опций, с помощью которых вы можете определить ограничения по длине разными способами:

class Person < ActiveRecord::Base

validates :name, length: { minimum: 2 }

validates :bio, length: { maximum: 500 }

validates :password, length: { in: 6..20 }

validates :registration\_number, length: { is: 6 }

end

Возможные опции ограничения длины такие:

* :minimum - аттрибут не может быть меньше определенной длины.
* :maximum - аттрибут не может быть больше определенной длины.
* :in (или :within) - длина аттрибута должна находиться в указанном интервале. Значение этой опции должно быть интервалом.
* :is - длина аттрибута должна быть равной указанному значению.

Значение сообщения об ошибке по умолчанию зависит от типа выполняемой валидации длины. Можно переопределить эти сообщения, используя опции :wrong\_length, :too\_long и :too\_short, и %{count} как место для вставки числа, соответствующего длине используемого ограничения. Можете использовать опцию :message для определения сообщения об ошибке.

class Person < ActiveRecord::Base

validates :bio, length: { maximum: 1000,

too\_long: "%{count} characters is the maximum allowed" }

end

По умолчанию этот хелпер считает символы, но вы можете разбить значение иным способом используя опцию :tokenizer:

class Essay < ActiveRecord::Base

validates :content, length: {

minimum: 300,

maximum: 400,

tokenizer: lambda { |str| str.scan(/\w+/) },

too\_short: "must have at least %{count} words",

too\_long: "must have at most %{count} words"

}

end

Отметьте, что сообщения об ошибке по умолчанию во множественном числе (т.е., "is too short (minimum is %{count} characters)"). По этой причине, когда :minimum равно 1, следует предоставить собственное сообщение или использовать вместо него presence: true. Когда :in или :within имеют как нижнюю границу 1, следует или предоставить собственное сообщение, или вызвать presence перед length.

Хелпер size это псевдоним для length.

**numericality**

Этот хелпер проводит валидацию того, что ваши аттрибуты имеют только числовые значения. По умолчанию, этому будет соответствовать возможный знак первым символом, и следующее за ним целочисленное или с плавающей запятой число. Чтобы определить, что допустимы только целочисленные значения, установите :only\_integer в true.

Если установить :only\_integer в true, тогда будет использоваться регулярное выражение

/\A[+-]?\d+\Z/

для проведения валидации значения атрибута. В противном случае, он будет пытаться конвертировать значение в число, используя Float.

Отметьте, что вышеописанное регулярное выражение позволяет завершающий символ перевода строки

class Player < ActiveRecord::Base

validates :points, numericality: true

validates :games\_played, numericality: { only\_integer: true }

end

Кроме :only\_integer, хелпер validates\_numericality\_of также принимает следующие опции для добавления ограничений к приемлемым значениям:

* :greater\_than - определяет, что значение должно быть больше, чем значение опции. По умолчанию сообщение об ошибке для этой опции такое *"must be greater than %{count}"*.
* :greater\_than\_or\_equal\_to - определяет, что значение должно быть больше или равно значению опции. По умолчанию сообщение об ошибке для этой опции такое *"must be greater than or equal to %{count}"*.
* :equal\_to - определяет, что значение должно быть равно значению опции. По умолчанию сообщение об ошибке для этой опции такое *"must be equal to %{count}"*.
* :less\_than - определяет, что значение должно быть меньше, чем значение опции. По умолчанию сообщение об ошибке для этой опции такое *"must be less than %{count}"*.
* :less\_than\_or\_equal\_to - определяет, что значение должно быть меньше или равно значению опции. По умолчанию сообщение об ошибке для этой опции такое *"must be less than or equal to %{count}"*.
* :odd - определяет, что значение должно быть нечетным, если установлено true. По умолчанию сообщение об ошибке для этой опции такое *"must be odd"*.
* :even - определяет, что значение должно быть четным, если установлено true. По умолчанию сообщение об ошибке для этой опции такое *"must be even"*.

По умолчанию сообщение об ошибке *"is not a number"*.

**presence**

Этот хелпер проводит валидацию того, что определенные аттрибуты не пустые. Он использует метод blank? для проверки того, является ли значение или nil, или пустой строкой (это строка, которая или пуста, или содержит пробелы).

class Person < ActiveRecord::Base

validates :name, :login, :email, presence: true

end

Если хотите быть уверенным, что связь существует, нужно проверить, существует ли внешний ключ, используемый для связи, но не сам связанный объект.

class LineItem < ActiveRecord::Base

belongs\_to :order

validates :order, presence: true

end

Для того, чтобы проверять связанные записи, чье присутствие необходимо, нужно определить опцию :inverse\_of для связи:

class Order < ActiveRecord::Base

has\_many :line\_items, inverse\_of: :order

end

При проведении валидации существования объекта, связанного отношением has\_one или has\_many, будет проверено, что объект ни blank?, ни marked\_for\_destruction?.

Так как false.blank? это true, если хотите провести валидацию существования булева поля, нужно использовать validates :field\_name, inclusion: { in: [true, false] }.

По умолчанию сообщение об ошибке *"can't be empty"*.

**absence**

Этот хелпер проверяет, что указанные атрибуты отсутствуют. Он использует метод present? для проверки, что значение является либо nil, либо пустой строкой (то есть либо нулевой длины, либо состоящей из пробелов).

class Person < ActiveRecord::Base

validates :name, :login, :email, absence: true

end

Если хотите убедиться, что отсутствует связь, неободимо проверить, что отсутствует сам связанный объект, а не внешний ключ, используемый для связи.

class LineItem < ActiveRecord::Base

belongs\_to :order

validates :order, absence: true

end

Чтобы проверять связанные объекты, отсутствие которых требуется, для связи необходимо указать опцию :inverse\_of:

class Order < ActiveRecord::Base

has\_many :line\_items, inverse\_of: :order

end

Если проверяете отсутствие объекта, связанного отношением has\_one или has\_many, он проверит, что объект и не present?, и не marked\_for\_destruction?.

Поскольку false.present? является false, если хотите проверить отсутствие булева поля, следует использовать validates :field\_name, exclusion: { in: [true, false] }.

По умолчанию сообщение об ошибке *"must be blank"*.

**uniqueness**

Этот хелпер проводит валидацию того, что значение атрибута уникально, перед тем, как объект будет сохранен. Он не создает условие уникальности в базе данных, следовательно, может произойти так, что два разных подключения к базе данных создадут две записи с одинаковым значением для столбца, который вы подразумеваете уникальным. Чтобы этого избежать, нужно создать индекс unique в вашей базе данных.

class Account < ActiveRecord::Base

validates :email, uniqueness: true

end

Валидация производится путем SQL запроса в таблицу модели, поиска существующей записи с тем же значением аттрибута.

Имеется опция :scope, которую можно использовать для определения других аттрибутов, используемых для ограничения проверки уникальности:

class Holiday < ActiveRecord::Base

validates :name, uniqueness: { scope: :year,

message: "should happen once per year" }

end

Также имеется опция :case\_sensitive, которой можно определить, будет ли ограничение уникальности чувствительно к регистру или нет. Опция по умолчанию равна true.

class Person < ActiveRecord::Base

validates :name, uniqueness: { case\_sensitive: false }

end

Отметьте, что некоторые базы данных настроены на выполнение чувствительного к регистру поиска в любом случае.

По умолчанию сообщение об ошибке *"has already been taken"*.

**validates\_with**

Этот хелпер передает запись в отдельный класс для валидации.

class Person < ActiveRecord::Base

validates\_with GoodnessValidator

end

class GoodnessValidator < ActiveModel::Validator

def validate(record)

if record.first\_name == "Evil"

record.errors[:base] << "This person is evil"

end

end

end

Ошибки, добавляемые в record.errors[:base] относятся к состоянию записи в целом, а не к определенному аттрибуту.

Хелпер validates\_with принимает класс или список классов для использования в валидации. Для validates\_with нет сообщения об ошибке по умолчанию. Следует вручную добавлять ошибки в коллекцию errors записи в классе валидатора.

Для применения метода validate, необходимо иметь определенным параметр record, который является записью, проходящей валидацию.

Подобно всем другим валидациям, validates\_with принимает опции :if, :unless и :on. Если передадите любые другие опции, они будут переданы в класс валидатора как options:

class Person < ActiveRecord::Base

validates\_with GoodnessValidator, fields: [:first\_name, :last\_name]

end

class GoodnessValidator < ActiveModel::Validator

def validate(record)

if options[:fields].any?{|field| record.send(field) == "Evil" }

record.errors[:base] << "This person is evil"

end

end

end

Отметьте, что валидатор будет инициализирован **только один раз** на протяжении всего жизненного цикла приложения, а не при каждом запуске валидации, поэтому будьте аккуратнее с использованием переменных экземпляра в нем.

Если ваш валидатор настолько сложный, что вы хотите использовать переменные экземпляра, вместо него проще использовать обычные объекты Ruby:

class Person < ActiveRecord::Base

validate do |person|

GoodnessValidator.new(person).validate

end

end

class GoodnessValidator

def initialize(person)

@person = person

end

def validate

if some\_complex\_condition\_involving\_ivars\_and\_private\_methods?

@person.errors[:base] << "This person is evil"

end

end

# ...

end

**validates\_each**

Этот хелпер помогает провести валидацию атрибутов с помощью блока кода. Он не имеет предопределенной валидационной функции. Вы должны создать ее, используя блок, и каждый атрибут, указанный в validates\_each, будет протестирован в нем. В следующем примере нам не нужны имена и фамилии, начинающиеся с маленькой буквы.

class Person < ActiveRecord::Base

validates\_each :name, :surname do |record, attr, value|

record.errors.add(attr, 'must start with upper case') if value =~ /\A[a-z]/

end

end

Блок получает запись, имя атрибута и значение атрибута. Вы можете делать что угодно для проверки валидности данных внутри блока. Если валидация проваливается, следует добавить сообщение об ошибке в модель, которое делает ее невалидной.

**Общие опции валидаций**

Есть несколько общих опций валидаций:

**:allow\_nil**

Опция :allow\_nil пропускает валидацию, когда проверяемое значение равно nil.

class Coffee < ActiveRecord::Base

validates :size, inclusion: { in: %w(small medium large),

message: "%{value} is not a valid size" }, allow\_nil: true

end

**:allow\_blank**

Опция :allow\_blank подобна опции :allow\_nil. Эта опция пропускает валидацию, если значение аттрибута blank?, например nil или пустая строка.

class Topic < ActiveRecord::Base

validates :title, length: { is: 5 }, allow\_blank: true

end

Topic.create(title: "").valid? # => true

Topic.create(title: nil).valid? # => true

**:message**

Как мы уже видели, опция :message позволяет определить сообщение, которое будет добавлено в коллекцию errors, когда валидация проваливается. Если эта опция не используется, Active Record будет использовать соответственные сообщение об ошибках по умолчанию для каждого валидационного хелпера.

**:on**

Опция :on позволяет определить, когда должна произойти валидация. Стандартное поведение для всех встроенных валидационных хелперов это запускаться при сохранении (и когда создается новая запись, и когда она обновляется). Если хотите изменить это, используйте on: :create, для запуска валидации только когда создается новая запись, или on: :update, для запуска валидации когда запись обновляется.

class Person < ActiveRecord::Base

# будет возможно обновить email с дублирующим значением

validates :email, uniqueness: true, on: :create

# будет возможно создать запись с нечисловым возрастом

validates :age, numericality: true, on: :update

# по умолчанию (проверяет и при создании, и при обновлении)

validates :name, presence: true

end

**Строгие валидации**

Также можно определить валидации строгими, чтобы они вызывали ActiveModel::StrictValidationFailed, когда объект невалиден.

class Person < ActiveRecord::Base

validates :name, presence: { strict: true }

end

Person.new.valid? # => ActiveModel::StrictValidationFailed: Name can't be blank

Также возможно передать собственное исключение в опцию :strict

class Person < ActiveRecord::Base

validates :token, presence: true, uniqueness: true, strict: TokenGenerationException

end

Person.new.valid? # => TokenGenerationException: Token can't be blank

**Условная валидация**

Иногда имеет смысл проводить валидацию объекта только при выполнении заданного условия. Это можно сделать, используя опции :if и :unless, которые принимают символ, строку, Proc или массив. Опцию :if можно использовать, если вы хотите определить, когда валидация **должна** произойти. Если вы хотите определить, когда валидация **не должна** произойти, воспользуйтесь опцией :unless.

**Использование символа с :if и :unless**

Вы можете связать опции :if и :unless с символом, соответствующим имени метода, который будет вызван перед валидацией. Это наиболее часто используемый вариант.

class Order < ActiveRecord::Base

validates :card\_number, presence: true, if: :paid\_with\_card?

def paid\_with\_card?

payment\_type == "card"

end

end

**Использование строки с :if и :unless**

Также можно использовать строку, которая будет вычислена с использованием eval, и должна содержать валидный код Ruby. Этот вариант следует использовать, если строка содержит действительно короткое условие.

class Person < ActiveRecord::Base

validates :surname, presence: true, if: "name.nil?"

end

**Использование Proc с :if и :unless**

Наконец, можно связать :if и :unless с объектом Proc, который будет вызван. Использование объекта Proc дает возможность написать встроенное условие вместо отдельного метода. Этот вариант лучше всего подходит для однострочного кода.

class Account < ActiveRecord::Base

validates :password, confirmation: true,

unless: Proc.new { |a| a.password.blank? }

end

**Группировка условных валидаций**

Иногда полезно иметь несколько валидаций с одним условием, это легко достигается с использованием with\_options.

class User < ActiveRecord::Base

with\_options if: :is\_admin? do |admin|

admin.validates :password, length: { minimum: 10 }

admin.validates :email, presence: true

end

end

Во все валидации внутри with\_options будет автоматически передано условие if: :is\_admin?.

**Объединение условий валидации**

С другой стороны, может использоваться массив, когда несколько условий определяют, должна ли произойти валидация. Более того, в одной и той же влидации можно применить и :if:, и :unless.

class Computer < ActiveRecord::Base

validates :mouse, presence: true,

if: ["market.retail?", :desktop?]

unless: Proc.new { |c| c.trackpad.present? }

end

Валидация выполнится только тогда, когда все условия :if, и ни одно из условий :unless будут вычислены со значением true.

**Выполнение собственных валидаций**

Когда встроенных валидационных хелперов недостаточно для ваших нужд, можете написать свои собственные валидаторы или методы валидации.

**Собственные валидаторы**

Собственные валидаторы это классы, расширяющие ActiveModel::Validator. Эти классы должны реализовать метод validate, принимающий запись как аргумент и выполняющий валидацию на ней. Собственный валидатор вызывается с использованием метода validates\_with.

class MyValidator < ActiveModel::Validator

def validate(record)

if record.name.starts\_with? ‘X’

record.errors[:name] << ‘Need a name starting with X please!’

end

end

end

class Person

include ActiveModel::Validations

validates\_with MyValidator

end

Простейшим способом добавить собственные валидаторы для валидации отдельных атрибутов является наследуемость от ActiveModel::EachValidator. В этом случае класс собственного валидатора должен реализовать метод validate\_each, принимающий три аргумента: запись, атрибут и значение, соответствующее экземпляру, соответственно атрибут тот, который будет проверяться и значение в переданном экземпляре:

class EmailValidator < ActiveModel::EachValidator

def validate\_each(record, attribute, value)

unless value =~ /\A([^@\s]+)@((?:[-a-z0-9]+\.)`[a-z]{2,})\z/i

record.errors[attribute] << (options[:message] || "is not an email")

end

end

end

class Person < ActiveRecord::Base

validates :email, presence: true, email: true

end

Как показано в примере, можно объединять стандартные валидации со своими произвольными валидаторами.

**Собственные методы**

Также возможно создать методы, проверяющие состояние ваших моделей и добавляющие сообщения в коллекцию errors, когда они невалидны. Затем эти методы следует зарегистрировать, используя метод класса validate, передав символьные имена валидационных методов.

Можно передать более одного символа для каждого метода класса, и соответствующие валидации будут запущены в том порядке, в котором они зарегистрированы.

class Invoice < ActiveRecord::Base

validate :expiration\_date\_cannot\_be\_in\_the\_past,

:discount\_cannot\_be\_greater\_than\_total\_value

def expiration\_date\_cannot\_be\_in\_the\_past

if expiration\_date.present? && expiration\_date < Date.today

errors.add(:expiration\_date, "can't be in the past")

end

end

def discount\_cannot\_be\_greater\_than\_total\_value

errors.add(:discount, "can't be greater than total value") if

discount > total\_value

end

end

По умолчанию такие валидации будут выполнены каждый раз при вызове valid?. Также возможно контролировать, когда выполнять собственные валидации, передав опцию :on в метод validate, с ключами: :create или :update.

class Invoice < ActiveRecord::Base

validate :active\_customer, on: :create

def active\_customer

errors.add(:customer\_id, "is not active") unless customer.active?

end

end

**Работаем с ошибками валидации**

В дополнение к методам valid? и invalid?, раскрытым ранее, Rails предоставляет ряд методов для работы с коллекцией errors и исследования валидности объектов.

Предлагаем список наиболее часто используемых методов. Если хотите увидеть список всех доступных методов, обратитесь к документации по ActiveModel::Errors.

**errors**

Возвращает экземпляр класса ActiveModel::Errors, содержащий все ошибки. Каждый ключ это имя атрибута и значение это массив строк со всеми ошибками.

class Person < ActiveRecord::Base

validates :name, presence: true, length: { minimum: 3 }

end

person = Person.new

person.valid? # => false

person.errors.messages

# => {:name=>["can't be blank", "is too short (minimum is 3 characters)"]}

person = Person.new(name: "John Doe")

person.valid? # => true

person.errors.messages # => []

**errors[]**

errors[] используется, когда вы хотите проверить сообщения об ошибке для определенного атрибута. Он возвращает массив строк со всеми сообщениями об ошибке для заданного атрибута, каждая строка с одним сообщением об ошибке. Если нет ошибок, относящихся к атрибуту, возвратится пустой массив.

class Person < ActiveRecord::Base

validates :name, presence: true, length: { minimum: 3 }

end

person = Person.new(name: "John Doe")

person.valid? # => true

person.errors[:name] # => []

person = Person.new(name: "JD")

person.valid? # => false

person.errors[:name] # => ["is too short (minimum is 3 characters)"]

person = Person.new

person.valid? # => false

person.errors[:name]

# => ["can't be blank", "is too short (minimum is 3 characters)"]

**errors.add**

Метод add позволяет вручную добавлять сообщения, которые относятся к определенным атрибутам. Можно использовать методы errors.full\_messages или errors.to\_a для просмотра сообщения в форме, в которой они отображаются пользователю. Эти определенные сообщения получают предшествующим (и с прописной буквы) имя аттрибута. add получает имя атрибута, к которому вы хотите добавить сообщение, и само сообщение.

class Person < ActiveRecord::Base

def a\_method\_used\_for\_validation\_purposes

errors.add(:name, "cannot contain the characters !@#%\*()\_-+=")

end

end

person = Person.create(name: "!@#")

person.errors[:name]

# => ["cannot contain the characters !@#%\*()\_-+="]

person.errors.full\_messages

# => ["Name cannot contain the characters !@#%\*()\_-+="]

Другой способ использования заключается в установлении []=

class Person < ActiveRecord::Base

def a\_method\_used\_for\_validation\_purposes

errors[:name] = "cannot contain the characters !@#%\*()\_-+="

end

end

person = Person.create(name: "!@#")

person.errors[:name]

# => ["cannot contain the characters !@#%\*()\_-+="]

person.errors.to\_a

# => ["Name cannot contain the characters !@#%\*()\_-+="]

**errors[:base]**

Можете добавлять сообщения об ошибках, которые относятся к состоянию объекта в целом, а не к отдельному аттрибуту. Этот метод можно использовать, если вы хотите сказать, что объект невалиден, независимо от значений его атрибутов. Поскольку errors[:base] массив, можете просто добавить строку к нему, и она будет использована как сообщение об ошибке.

class Person < ActiveRecord::Base

def a\_method\_used\_for\_validation\_purposes

errors[:base] << "This person is invalid because ..."

end

end

**errors.clear**

Метод clear используется, когда вы намеренно хотите очистить все сообщения в коллекции errors. Естественно, вызов errors.clear для невалидного объекта фактически не сделает его валидным: сейчас коллекция errors будет пуста, но в следующий раз, когда вы вызовете valid? или любой метод, который пытается сохранить этот объект в базу данных, валидации выполнятся снова. Если любая из валидаций провалится, коллекция errors будет заполнена снова.

class Person < ActiveRecord::Base

validates :name, presence: true, length: { minimum: 3 }

end

person = Person.new

person.valid? # => false

person.errors[:name]

# => ["can't be blank", "is too short (minimum is 3 characters)"]

person.errors.clear

person.errors.empty? # => true

p.save # => false

p.errors[:name]

# => ["can't be blank", "is too short (minimum is 3 characters)"]

**errors.size**

Метод size возвращает количество сообщений об ошибке для объекта.

class Person < ActiveRecord::Base

validates :name, presence: true, length: { minimum: 3 }

end

person = Person.new

person.valid? # => false

person.errors.size # => 2

person = Person.new(name: "Andrea", email: "andrea@example.com")

person.valid? # => true

person.errors.size # => 0

**Колбэки**

Колбэки Active Record разрешают присоединить код к определенным событиям в жизненном цикле ваших моделей. Это позволяет добавить поведение модели, прозрачно запустив код, когда эти события произойдут, например, когда вы создадите новую запись, обновите его, удалите его и так далее.

**Жизненный цикл объекта**

В результате обычных операций приложения на Rails, объекты могут быть созданы, обновлены и уничтожены. Active Record дает возможность вмешаться в этот жизненный цикл объекта, таким образом, вы можете контролировать свое приложение и его данные.

Валидации позволяют вам быть уверенными, что только валидные данные хранятся в вашей базе данных. Колбэки позволяют вам переключать логику до или после изменения состояния объекта.

**Обзор колбэков**

Колбэки это методы, которые вызываются в определенные моменты жизненного цикла объекта. С колбэками возможно написать код, который будет запущен, когда объект Active Record создается, сохраняется, обновляется, удаляется, проходит валидацию или загружается из базы данных.

**Регистрация колбэков**

Для того, чтобы использовать доступные колбэки, их нужно зарегистрировать. Можно реализовать колбэки как обычные методы, а затем использовать макро-методы класса для их регистрации как колбэков.

|  |
| --- |
| class User < ActiveRecord::Base    validates :login, :email, presence: true      before\_validation :ensure\_login\_has\_a\_value      protected    def ensure\_login\_has\_a\_value      if login.nil?        self.login = email unless email.blank?      end    end  end |

Макро-методы класса также могут получать блок. Их можно использовать, если код внутри блока такой короткий, что помещается в одну строку.

|  |
| --- |
| class User < ActiveRecord::Base    validates :login, :email, presence: true      before\_create do      self.name = login.capitalize if name.blank?    end  end |

Колбэки также могут быть зарегистрированы на выполнение при определенных событиях жизненного цикла:

|  |
| --- |
| class User < ActiveRecord::Base    before\_validation :normalize\_name, on: :create      # :on также принимает массив    after\_validation :set\_location, on: [ :create, :update ]      protected    def normalize\_name      self.name = self.name.downcase.titleize    end      def set\_location      self.location = LocationService.query(self)    end  end |

Считается хорошей практикой объявлять методы колбэков как protected или private. Если их оставить public, они могут быть вызваны извне модели и нарушить принципы инкапсуляции объекта.

**Доступные колбэки**

Вот список всех доступных колбэков Active Record, перечисленных в том порядке, в котором они вызываются в течение соответственных операций:

**Создание объекта**

* before\_validation
* after\_validation
* before\_save
* around\_save
* before\_create
* around\_create
* after\_create
* after\_save

**Обновление объекта**

* before\_validation
* after\_validation
* before\_save
* around\_save
* before\_update
* around\_update
* after\_update
* after\_save

**Уничтожение объекта**

* before\_destroy
* around\_destroy
* after\_destroy

after\_save запускается и при создании, и при обновлении, но всегда *после* более специфичных колбэков after\_create и after\_update, не зависимо от порядка, в котором запускаются макро-вызовы.

**after\_initialize и after\_find**

Колбэк after\_initialize вызывается всякий раз, когда возникает экземпляр объекта Active Record, или непосредственно при использовании new, или когда запись загружается из базы данных. Он необходим, чтобы избежать необходимости непосредственно переопределять метод Active Record initialize.

Колбэк after\_find будет вызван всякий раз, когда Active Record загружает запись из базы данных. after\_find вызывается перед after\_initialize, если они оба определены.

У колбэков after\_initialize и after\_find нет пары before\_\*, но они могут быть зарегистрированы подобно другим колбэкам Active Record.

|  |
| --- |
| class User < ActiveRecord::Base    after\_initialize do |user|      puts "You have initialized an object!"    end      after\_find do |user|      puts "You have found an object!"    end  end    >> User.new  You have initialized an object!  => #<User id: nil>    >> User.first  You have found an object!  You have initialized an object!  => #<User id: 1> |

**Запуск колбэков**

Следующие методы запускают колбэки:

* create
* create!
* decrement!
* destroy
* destroy!
* destroy\_all
* increment!
* save
* save!
* save(validate: false)
* toggle!
* update\_attribute
* update
* update!
* valid?

Дополнительно, колбэк after\_find запускается следующими поисковыми методами:

* all
* first
* find
* find\_by
* find\_by\_\*
* find\_by\_\*!
* find\_by\_sql
* last

Колбэк after\_initialize запускается всякий раз, когда инициализируется новый объект класса.

Методы find\_by\_\* и find\_by\_\*! это динамические методы поиска, создаваемые автоматически для каждого атрибута.

**Пропуск колбэков**

Подобно валидациям, также возможно пропустить колбэки, используя следующие методы.

* decrement
* decrement\_counter
* delete
* delete\_all
* increment
* increment\_counter
* toggle
* touch
* update\_column
* update\_columns
* update\_all
* update\_counters

Однако, эти методы нужно использовать осторожно, поскольку важные бизнес-правила и логика приложения могут содержаться в колбэках. Пропуск их без понимания возможных последствий может привести к невалидным данным.

**Прерывание выполнения**

Как только вы зарегистрировали новые колбэки в своих моделях, они будут поставлены в очередь на выполнение. Эта очередь включает все валидации вашей модели, зарегистрированные колбэки и операции с базой данных для выполнения.

Вся цепочка колбэков упаковывается в операцию. Если любой метод *before* колбэков возвращает false или вызывает исключение, выполняемая цепочка прерывается и запускается ROLLBACK; Колбэки *after* могут достичь этого, только вызвав исключение.

Вызов произвольного исключения может прервать код, который предполагает, что save и тому подобное не будут провалены подобным образом. Исключение ActiveRecord::Rollback чуть точнее сообщает Active Record, что происходит откат. Он подхватывается изнутри, но не перевызывает исключение.

**Колбэки для отношений**

Колбэки работают с отношениями между моделями, и даже могут быть определены ими. Представим пример, где пользователь имеет много публикаций. Публикации пользователя должны быть уничтожены, если уничтожается пользователь. Давайте добавим колбэк after\_destroy в модель User через ее отношения с моделью Post.

|  |
| --- |
| class User < ActiveRecord::Base    has\_many :posts, dependent: :destroy  end    class Post < ActiveRecord::Base    after\_destroy :log\_destroy\_action      def log\_destroy\_action      puts 'Post destroyed'    end  end    >> user = User.first  => #<User id: 1>  >> user.posts.create!  => #<Post id: 1, user\_id: 1>  >> user.destroy  Post destroyed  => #<User id: 1> |

**Условные колбэки**

Как и в валидациях, возможно сделать вызов метода колбэка условным от удовлетворения заданного условия. Это осуществляется при использовании опций :if и :unless, которые могут принимать символ, строку, Proc или массив. Опцию :if следует использовать для определения, при каких условиях колбэк *должен* быть вызван. Если вы хотите определить условия, при которых колбэк *не должен* быть вызван, используйте опцию :unless.

**Использование :if и :unless с символом**

Опции :if и :unless можно связать с символом, соответствующим имени метода условия, который будет вызван непосредственно перед вызовом колбэка. При использовании опции :if, колбэк не будет выполнен, если метод условия возвратит false; при использовании опции :unless, колбэк не будет выполнен, если метод условия возвратит true. Это самый распространенный вариант. При использовании такой формы регистрации, также возможно зарегистрировать несколько различных условий, которые будут вызваны для проверки, должен ли запуститься колбэк.

|  |
| --- |
| class Order < ActiveRecord::Base    before\_save :normalize\_card\_number, if: :paid\_with\_card?  end |

**Использование :if и :unless со строкой**

Также возможно использование строки, которая будет вычислена с помощью eval, и, следовательно, должна содержать валидный код Ruby. Этот вариант следует использовать только тогда, когда строка представляет действительно короткое условие.

|  |
| --- |
| class Order < ActiveRecord::Base    before\_save :normalize\_card\_number, if: "paid\_with\_card?"  end |

**Использование :if и :unless с Proc**

Наконец, можно связать :if и :unless с объектом Proc. Этот вариант более всего подходит при написании коротких методов, обычно в одну строку.

|  |
| --- |
| class Order < ActiveRecord::Base    before\_save :normalize\_card\_number,      if: Proc.new { |order| order.paid\_with\_card? }  end |

**Составные условия для колбэков**

При написании условных колбэков, возможно смешивание :if и :unless в одном объявлении колбэка.

|  |
| --- |
| class Comment < ActiveRecord::Base    after\_create :send\_email\_to\_author, if: :author\_wants\_emails?,      unless: Proc.new { |comment| comment.post.ignore\_comments? }  end |

**Классы колбэков**

Иногда написанные вами методы колбэков достаточно полезны для повторного использования в других моделях. Active Record делает возможным создавать классы, включающие методы колбэка, так, что становится очень легко использовать их повторно.

Вот пример, где создается класс с колбэком after\_destroy для модели PictureFile:

|  |
| --- |
| class PictureFileCallbacks    def after\_destroy(picture\_file)      if File.exists?(picture\_file.filepath)        File.delete(picture\_file.filepath)      end    end  end |

При объявлении внутри класса, как выше, методы колбэка получают объект модели как параметр. Теперь можем использовать класс коллбэка в модели:

|  |
| --- |
| class PictureFile < ActiveRecord::Base    after\_destroy PictureFileCallbacks.new  end |

Заметьте, что нам нужно создать экземпляр нового объекта PictureFileCallbacks, после того, как объявили наш колбэк как отдельный метод. Это особенно полезно, если колбэки используют состояние экземпляря объекта. Часто, однако, более подходящим является иметь его как метод класса.

|  |
| --- |
| class PictureFileCallbacks    def self.after\_destroy(picture\_file)      if File.exists?(picture\_file.filepath)        File.delete(picture\_file.filepath)      end    end  end |

Если метод колбэка объявляется таким образом, нет необходимости создавать экземпляр объекта PictureFileCallbacks.

|  |
| --- |
| class PictureFile < ActiveRecord::Base    after\_destroy PictureFileCallbacks  end |

Внутри своего колбэк-класса можно создать сколько угодно колбэков.

**Транзакционные колбэки**

Имеются два дополнительных колбэка, которые включаются по завершению транзакции базы данных: after\_commit и after\_rollback. Эти колбэки очень покожи на колбэк after\_save, за исключением того, что они не запускаются пока изменения в базе данных не будут подтверждены или обращены. Они наиболее полезны, когда вашим моделям active record необходимо взаимодействовать с внешними системами, не являющимися частью транзакции базы данных.

Рассмотрим, к примеру, предыдущий пример, где модели PictureFile необходимо удалить файл после того, как запись уничтожена. Если что-либо вызовет исключение после того, как был вызван колбэк after\_destroy, и транзакция откатывается, файл будет удален и модель останется в противоречивом состоянии. Например, предположим, что picture\_file\_2 в следующем коде не валидна, и метод save! вызовет ошибку.

|  |
| --- |
| PictureFile.transaction do    picture\_file\_1.destroy    picture\_file\_2.save!  end |

Используя колбэк after\_commit, можно учесть этот случай.

|  |
| --- |
| class PictureFile < ActiveRecord::Base    after\_commit :delete\_picture\_file\_from\_disk, on: [:destroy]      def delete\_picture\_file\_from\_disk      if File.exist?(filepath)        File.delete(filepath)      end    end  end |

опция :on определяет, когда будет запущен колбэк. Если не предоставить опцию :on, колбэк будет запущен для каждого действия.

Колбэки after\_commit и after\_rollback гарантируют, что будут вызваны для всех созданных, обновленных или удаленных моделей внутри блока транзакции. Если какое-либо исключение вызовется в одном из этих колбэков, они будут проигнорированы, чтобы не препятствовать другим колбэкам. По сути, если код вашего колбэка может вызвать исключение, нужно для него вызвать rescue, и обработать его нужным образом в колбэке.

**Миграции**

Rails представляет DSL для управления схемой базы данных, называемый миграциями. Миграции хранятся в файлах, запускаемых для любой базы данных, которую поддерживает Active Record, с использованием rake. Вот миграция, создающая таблицу:

|  |
| --- |
| class CreatePublications < ActiveRecord::Migration    def change      create\_table :publications do |t|        t.string :title        t.text :description        t.references :publication\_type        t.integer :publisher\_id        t.string :publisher\_type        t.boolean :single\_issue          t.timestamps      end      add\_index :publications, :publication\_type\_id    end  end |

Rails отслеживает, какие файлы переданы в базу данных, и представляет особенность отката. Чтобы фактически создать таблицу, нужно запустить rake db:migrate, а чтобы ее откатить rake db:rollback.

Отметьте, что вышеприведенный код не зависит от базы данных: он выполнится в MySQL, postgresql, Oracle и иных.

**Обзор миграций**

Миграции - это удобный способ изменять схему вашей базы данных всё время неизменным и простым образом. Они используют Ruby DSL. Поэтому вам не нужно писать SQL вручную, позволяя вашей схеме быть независимой от базы данных.

Каждую миграцию можно рассматривать как новую 'версию' базы данных. Схема изначально ничего не содержит, а каждая миграция изменяет ее, добавляя или убирая таблицы, столбцы или записи. Active Record знает, как обновлять вашу схему со временем, перенося ее из определенной точки в прошлом в последнюю версию. Active Record также обновляет ваш файл db/schema.rb, чтобы он соответствовал текущей структуре вашей базы данных.

Вот пример миграции:

|  |
| --- |
| class CreateProducts < ActiveRecord::Migration    def change      create\_table :products do |t|        t.string :name        t.text :description          t.timestamps      end    end  end |

Эта миграция добавляет таблицу products со строковым столбцом name и текстовым столбцом description. Первичный ключ, названный id, также будет неявно добавлен по умолчанию, так как это первичный ключ по умолчанию для всех моделей Active Record. Макрос timestamps добавляет два столбца, created\_at и updated\_at. Эти специальные столбцы автоматически управляются Active Record, если существуют.

Отметьте, что мы определили изменение, которое мы хотим, чтобы произошло при движении вперед во времени. До запуска этой миграции таблицы нет. После - таблица будет существовать. Active Record также знает, как обратить эту миграцию: если мы откатываем эту миграцию, он удалит таблицу.

В базах данных, поддерживающих транзакции с выражениями, изменяющими схему, миграции оборачиваются в транзакцию. Если база данных это не поддерживает, и миграция проваливается, части, которые прошли успешно, не будут откаченны назад. Вам нужно произвести откат вручную.

Некоторые запросы не могут быть запущены в транзакции. Если ваш адаптер поддерживает транзакции DDL, можно использовать disable\_ddl\_transaction! для их отключения для отдельной миграции.

Если хотите миграцию для чего-то, что Active Record не знает, как обратить, вы можете использовать reversible:

|  |
| --- |
| class ChangeProductsPrice < ActiveRecord::Migration    def change      reversible do |dir|        change\_table :products do |t|          dir.up   { t.change :price, :string }          dir.down { t.change :price, :integer }        end      end    end  end |

С другой стороны, можно использовать up и down вместо change:

|  |
| --- |
| class ChangeProductsPrice < ActiveRecord::Migration    def up      change\_table :products do |t|        t.change :price, :string      end    end      def down      change\_table :products do |t|        t.change :price, :integer      end    end  end |

**Создание миграции**

**Создание автономной миграции**

Миграции хранятся как файлы в директории db/migrate, один файл на каждый класс. Имя файла имеет вид YYYYMMDDHHMMSS\_create\_products.rb, это означает, что временная метка UTC идентифицирует миграцию, затем идет знак подчеркивания, затем идет имя миграции, где слова разделены подчеркиваниями. Имя класса миграции содержит буквенную часть названия файла, но уже в формате CamelCase (т.е. слова пишутся слитно, каждое слово начинается с большой буквы). Например, 20080906120000\_create\_products.rb должен определять класс CreateProducts, а 20080906120001\_add\_details\_to\_products.rb должен определять AddDetailsToProducts. Rails использует эту метку, чтобы определить, какая миграция должна быть запущена и в каком порядке, так что если вы копируете миграции из другого приложения или генерируете файл сами, будьте более бдительны.

Конечно, вычисление временных меток не забавно, поэтому Active Record предоставляет генератор для управления этим:

|  |
| --- |
| $ rails generate migration AddPartNumberToProducts |

Это создат пустую, но правильно названную миграцию:

|  |
| --- |
| class AddPartNumberToProducts < ActiveRecord::Migration    def change    end  end |

Если имя миграции имеет форму "AddXXXToYYY" или "RemoveXXXFromYYY" и далее следует перечень имен столбцов и их типов, то в миграции будут созданы соответствующие выражения add\_column и remove\_column.

|  |
| --- |
| $ rails generate migration AddPartNumberToProducts part\_number:string |

создаст

|  |
| --- |
| class AddPartNumberToProducts < ActiveRecord::Migration    def change      add\_column :products, :part\_number, :string    end  end |

Если вы хотите добавить индекс на новый столбец, вы можете сделать это так

|  |
| --- |
| $ rails generate migration AddPartNumberToProducts part\_number:string:index |

создаст

|  |
| --- |
| class AddPartNumberToProducts < ActiveRecord::Migration    def change      add\_column :products, :part\_number, :string      add\_index :products, :part\_number    end  end |

Точно так же, вы можете сгенерировать миграцию для удаления столбца из командной строки:

|  |
| --- |
| $ rails generate migration RemovePartNumberFromProducts part\_number:string |

создаст

|  |
| --- |
| class RemovePartNumberFromProducts < ActiveRecord::Migration    def change      remove\_column :products, :part\_number, :string    end  end |

Вы не ограничены одним создаваемым столбцом. Например

|  |
| --- |
| $ rails generate migration AddDetailsToProducts part\_number:string price:decimal |

создаст

|  |
| --- |
| class AddDetailsToProducts < ActiveRecord::Migration    def change      add\_column :products, :part\_number, :string      add\_column :products, :price, :decimal    end  end |

Если имя миграции имеет форму "CreateXXX" и затем следует список имен и типов столбцов, то будет сгенерирована миграция, создающая таблицу XXX с перечисленными столбцами. Например:

|  |
| --- |
| $ rails generate migration CreateProducts name:string part\_number:string |

создаст

|  |
| --- |
| class CreateProducts < ActiveRecord::Migration    def change      create\_table :products do |t|        t.string :name        t.string :part\_number      end    end  end |

Как всегда, то, что было сгенерировано, является всего лишь стартовой точкой. Вы можете добавлять и убирать строки, как считаете нужным, отредактировав файл db/migrate/YYYYMMDDHHMMSS\_add\_details\_to\_products.rb.

Также генератор принимает такой тип столбца, как references (или его псевдоним belongs\_to). Например

|  |
| --- |
| $ rails generate migration AddUserRefToProducts user:references |

создаст

|  |
| --- |
| class AddUserRefToProducts < ActiveRecord::Migration    def change      add\_reference :products, :user, index: true    end  end |

Эта миграция создаст столбец user\_id и соответствующий индекс.

Существует также генератор, который будет производить объединение таблиц, если JoinTable является частью названия.

Например

|  |
| --- |
| $ rails generate migration CreateJoinTableCustomerProduct customer product |

Сгенерирует следующую миграцию:

|  |
| --- |
| class CreateJoinTableCustomerProduct < ActiveRecord::Migration    def change      create\_join\_table :customers, :products do |t|        # t.index [:customer\_id, :product\_id]        # t.index [:product\_id, :customer\_id]      end    end  end |

**Генераторы модели**

Генераторы модели и скаффолда создадут миграции, подходящие для создания новой модели. Миграция будет содержать инструкции для создания соответствующей таблицы. Если вы сообщите Rails, какие столбцы вы хотите, то выражения для добавления этих столбцов также будут созданы. Например, запуск

|  |
| --- |
| $ rails generate model Product name:string description:text |

создаст миграцию, которая выглядит так

|  |
| --- |
| class CreateProducts < ActiveRecord::Migration    def change      create\_table :products do |t|        t.string :name        t.text :description          t.timestamps      end    end  end |

Можно определить сколько угодно пар имя\_столбца/тип.

**Поддерживаемые модификаторы типа**

Также можно определить некоторые опции сразу после типа поля в фигурных скобках. Можно использовать следующие модификаторы:

* limit Устанавливает максимальный размер полей string/text/binary/integer
* precision Определяет точность для полей decimal
* scale Определяет масштаб для полей decimal
* polymorphic Добавляет столбец type для связей belongs\_to

К примеру, запуск

|  |
| --- |
| $ rails generate migration AddDetailsToProducts price:decimal{5,2} supplier:references{polymorphic} |

создат миграцию, которая выглядит как эта:

|  |
| --- |
| class AddDetailsToProducts < ActiveRecord::Migration    def change      add\_column :products, :price, precision: 5, scale: 2      add\_reference :products, :supplier, polymorphic: true, index: true    end  end |

**Написание миграции**

Как только вы создали свою миграцию, используя один из генераторов, пришло время поработать!

**Создание таблицы**

Метод create\_table один из самых фундаментальных, но в большинстве случаев, он будет создан для вас генератором модели или скаффолда. Обычное использование такое

|  |
| --- |
| create\_table :products do |t|    t.string :name  end |

Это создаст таблицу products со столбцом name (и, как обсуждалось выше, подразумеваемым столбцом id).

По умолчанию create\_table создаст первичный ключ, названный id. Вы можете изменить имя первичного ключа с помощью опции :primary\_key (не забудьте также обновить соответствующую модель), или, если вы вообще не хотите первичный ключ, можно указать опцию id: false. Если нужно передать базе данных специфичные опции, вы можете поместить фрагмент SQL в опцию :options. Например,

|  |
| --- |
| create\_table :products, options: "ENGINE=BLACKHOLE" do |t|    t.string :name, null: false  end |

добавит ENGINE=BLACKHOLE к SQL выражению, используемому для создания таблицы (при использовании MySQL по умолчанию передается ENGINE=InnoDB).

**Создание соединительной таблицы**

Миграционный метод create\_join\_table создает соединительную страницу HABTM. Обычное использование будет таким:

|  |
| --- |
| create\_join\_table :products, :categories |

что создаст таблицу categories\_products с двумя столбцами по имени category\_id и product\_id. У этих столбцов есть опция :null, установленная в false по умолчанию. Это может быть переопределено опцией :column\_options.

|  |
| --- |
| create\_join\_table :products, :categories, column\_options: {null: true} |

создаст product\_id и category\_id с опцией :null как true.

Если хотите изменить имя таблицы, используйте опцию :table\_name. Например,

|  |
| --- |
| create\_join\_table :products, :categories, table\_name: :categorization |

создаст таблицу categorization.

По умолчанию create\_join\_table создаст два столбца без опций, но можно определить эти опции с использованием опции :column\_options. Например,

|  |
| --- |
| create\_join\_table :products, :categories, column\_options: {null: true} |

создаст product\_id и category\_id с опцией :null равной true.

create\_join\_table также принимает блок, который можно использовать для добавления индексов (которые по умолчанию не создаются) или дополнительных столбцов:

|  |
| --- |
| create\_join\_table :products, :categories do |t|    t.index :product\_id    t.index :category\_id  end |

**Изменение таблиц**

Близкий родственник create\_table это change\_table, используемый для изменения существующих таблиц. Он используется подобно create\_table, но у объекта, передаваемого в блок, больше методов. Например

|  |
| --- |
| change\_table :products do |t|    t.remove :description, :name    t.string :part\_number    t.index :part\_number    t.rename :upccode, :upc\_code  end |

удаляет столбцы description и name, создает строковый столбец part\_number и добавляет индекс на него. Наконец, он переименовывает столбец upccode.

**Когда хелперов недостаточно**

Если хелперов, предоставленных Active Record, недостаточно, можно использовать метод execute для запуска произвольного SQL:

|  |
| --- |
| Products.connection.execute('UPDATE `products` SET `price`=`free` WHERE 1') |

Больше подробностей и примеров отдельных методов содержится в документации по API. В частности, документация для [ActiveRecord::ConnectionAdapters::SchemaStatements](http://api.rubyonrails.org/classes/ActiveRecord/ConnectionAdapters/SchemaStatements.html) (который обеспечивает методы, доступные в методах up, down и change), [ActiveRecord::ConnectionAdapters::TableDefinition](http://api.rubyonrails.org/classes/ActiveRecord/ConnectionAdapters/TableDefinition.html) (который обеспечивает методы, доступные у объекта, переданного методом create\_table) и [ActiveRecord::ConnectionAdapters::Table](http://api.rubyonrails.org/classes/ActiveRecord/ConnectionAdapters/Table.html) (который обеспечивает методы, доступные у объекта, переданного методом change\_table).

**Использование метода change**

Метод change это основной метод написания миграций. Он работает в большинстве случаев, когда Active Record знает, как обратить миграцию автоматически. На текущий момент метод change поддерживает только эти определения миграции:

* add\_column
* add\_index
* add\_reference
* add\_timestamps
* create\_table
* create\_join\_table
* drop\_table (необходимо указать блок)
* drop\_join\_table (необходимо указать блок)
* remove\_timestamps
* rename\_column
* rename\_index
* remove\_reference
* rename\_table

change\_table так же является обратимым, пока блок не вызывает change, change\_default или remove.

Если вы нуждаетесь в использовании иных методов, следует использовать reversible или писать методы up и down вместо метода change.

**Использование reversible**

Комплексная миграция может включать процессы, которые Active Record не знает как обратить. Вы можете использовать reversible, чтобы указать что делать когда миграция требует отката. Например,

|  |
| --- |
| class ExampleMigration < ActiveRecord::Migration    def change      create\_table :products do |t|        t.references :category      end        reversible do |dir|        dir.up do          # добавим внешний ключ          execute <<-SQL            ALTER TABLE products              ADD CONSTRAINT fk\_products\_categories              FOREIGN KEY (category\_id)              REFERENCES categories(id)          SQL        end        dir.down do          execute <<-SQL            ALTER TABLE products              DROP FOREIGN KEY fk\_products\_categories          SQL        end      end        add\_column :users, :home\_page\_url, :string      rename\_column :users, :email, :email\_address    end |

Использование reversible гарантирует что инструкции выполнятся в правильном порядке. Если предыдущий пример миграции откатывается, down блок начнёт выполнятся после того как столбец home\_page\_url будет удалён и перед перед тем как произойдёт удаление таблицы products.

Иногда миграция будет делать то, что просто необратимо; например, она может уничтожить некоторые данные. В таких случаях, вы можете вызвать ActiveRecord::IrreversibleMigration в вашем down блоке. Если кто-либо попытается отменить вашу миграцию, будет отображена ошибка, что это не может быть выполнено.

**Использование методов up/down**

Вы так же можете использовать старый стиль миграций используя up и down методы, вместо change Метод up должен описывать изменения, которые вы хотите внести в вашу схему, а метод down вашей миграции должен обращать изменения, внесенные методом up. Другими словами, схема базы данных должна остаться неизменной после выполнения up, а затем down. Например, если вы создали таблицу в методе up, ее следует удалить в методе down. Разумно производить отмену изменений в полностью противоположном порядке тому, в котором они сделаны в методе up. Например в сравнении с кодом reversible следующий код будет эквивалентным

|  |
| --- |
| class ExampleMigration < ActiveRecord::Migration    def up      create\_table :products do |t|        t.references :category      end        #добавляем внешний ключ      execute <<-SQL        ALTER TABLE products          ADD CONSTRAINT fk\_products\_categories          FOREIGN KEY (category\_id)          REFERENCES categories(id)      SQL        add\_column :users, :home\_page\_url, :string      rename\_column :users, :email, :email\_address    end      def down      rename\_column :users, :email\_address, :email      remove\_column :users, :home\_page\_url        execute <<-SQL        ALTER TABLE products          DROP FOREIGN KEY fk\_products\_categories      SQL        drop\_table :products    end  end |

Если ваша миграция не обратима вам следует вызвать ActiveRecord::IrreversibleMigration из вашего метода down. Если кто-либо попытается отменить вашу миграцию, будет отображена ошибка, что это не может быть выполнено.

**Возвращение к предыдущим миграциям**

Вы можете использовать возможность Active Record откатить миграции используя revert метод:

|  |
| --- |
| require\_relative '2012121212\_example\_migration'    class FixupExampleMigration < ActiveRecord::Migration    def change      revert ExampleMigration        create\_table(:apples) do |t|        t.string :variety      end    end  end |

Метод revert так же может принимает блок. Это может быть полезным для отката выбранной части предыдущих миграций. Для примера, давайте представим что ExampleMigration закомичена и уже поздно решать, хорошо ли было бы сериализовать список продуктов или нет. Она может быть написана так:

|  |
| --- |
| class SerializeProductListMigration < ActiveRecord::Migration    def change      add\_column :categories, :product\_list        reversible do |dir|        dir.up do          # transfer data from Products to Category#product\_list        end        dir.down do          # create Products from Category#product\_list        end      end        revert do        # copy-pasted code from ExampleMigration        create\_table :products do |t|          t.references :category        end          reversible do |dir|          dir.up do            #add a foreign key            execute <<-SQL              ALTER TABLE products                ADD CONSTRAINT fk\_products\_categories                FOREIGN KEY (category\_id)                REFERENCES categories(id)            SQL          end          dir.down do            execute <<-SQL              ALTER TABLE products                DROP FOREIGN KEY fk\_products\_categories            SQL          end        end          # The rest of the migration was ok      end    end  end |

Подобная миграция так же может быть написана без использования revert, но это бы привело к ещё нескольким шагам: изменение порядка(следования) create table и reversible, замена create\_table на drop\_table и в конечном итоге изменение up down наоборот. Обо всём этом уже позаботился revert.

**Запуск миграций**

Rails предоставляет ряд задач rake для запуска определенных наборов миграций.

Самая первая команда Rake, относящаяся к миграциям, которую вы будете использовать, это rake db:migrate. В своей основной форме она всего лишь запускает метод change или up для всех миграций, которые еще не были запущены. Если таких миграций нет, она выходит. Она запустит эти миграции в порядке, основанном на дате миграции.

Заметьте, что запуск db:migrate также вызывает задачу db:schema:dump, которая обновляет ваш файл db/schema.rb в соответствии со структурой вашей базы данных.

Если вы определите целевую версию, Active Record запустит требуемые миграции (методы up, down или change), пока не достигнет требуемой версии. Версия это числовой префикс у файла миграции. Например, чтобы мигрировать к версии 20080906120000, запустите

|  |
| --- |
| $ rake db:migrate VERSION=20080906120000 |

Если версия 20080906120000 больше текущей версии (т.е. миграция вперед) это запустит метод change (или up) для всех миграций до и включая 20080906120000, но не запустит какие-либо поздние миграции. Если миграция назад, это запустит метод down для всех миграций до, но не включая, 20080906120000.

**Откат**

Обычная задача это откатить последнюю миграцию. Например, вы сделали ошибку и хотите исправить ее. Можно отследить версию предыдущей миграции и произвести миграцию до нее, но можно поступить проще, запустив

|  |
| --- |
| $ rake db:rollback |

Это вернёт ситуацию к последней миграции, или обратив метод change, или запустив метод down. Если нужно отменить несколько миграций, можно указать параметр STEP:

|  |
| --- |
| $ rake db:rollback STEP=3 |

произойдёт откат на 3 последних миграции.

Задача db:migrate:redo это ярлык для выполнения отката, а затем снова запуска миграции. Так же, как и с задачей db:rollback можно указать параметр STEP, если нужно работать более чем с одной версией, например

|  |
| --- |
| $ rake db:migrate:redo STEP=3 |

Ни одна из этих команд Rake не может сделать ничего такого, чего нельзя было бы сделать с db:migrate. Они просто более удобны, так как вам не нужно явно указывать версию миграции, к которой нужно мигрировать.

**Установка базы даных**

Задача rake db:setup создаст базу данных, загрузит схему и инициализирует ее с помощью данных seed.

**Сброс базы данных**

Задача db:reset удалит базу данных и установит ее заново. Функционально это эквивалентно rake db:drop db:setup.

Это не то же самое, что запуск всех миграций. Оно использует только текущее содержимое файла schema.rb. Если миграция не может быть откачена, 'rake db:reset' может не помочь вам. Подробнее об экспорте схемы смотрите "Экспорт схемы":/rails-database-migrations/schema-dumping-and-you.

**Запуск определенных миграций**

Если вам нужно запустить определенную миграцию (up или down), задачи db:migrate:up и db:migrate:down сделают это. Просто определите подходящий вариант и у соответствующей миграции будет вызван метод change, up или down, например

|  |
| --- |
| $ rake db:migrate:up VERSION=20080906120000 |

запустит метод up у миграции 20080906120000. Эта задача сперва проверит, была ли миграция уже выполнена, и ничего делать не будет, если Active Record считает, что она уже была запущена.

**Запуск миграций в различных средах**

По умолчанию запуск rake db:migrate запустится в окружении development. Для запуска миграций в другом окружении, его можно указать, используя переменную среды RAILS\_ENV при запуске команды. Например, для запуска миграций в среде test, следует запустить:

|  |
| --- |
| $ rake db:migrate RAILS\_ENV=test |

**Изменение вывод результата запущенных миграций**

По умолчанию миграции говорят нам только то, что они делают, и сколько времени это заняло. Миграция, создающая таблицу и добавляющая индекс, выдаст что-то наподобие этого

|  |
| --- |
| ==  CreateProducts: migrating =================================================  -- create\_table(:products)     -> 0.0028s  ==  CreateProducts: migrated (0.0028s) ======================================== |

Некоторые методы в миграциях позволяют вам все это контролировать:

|  |  |
| --- | --- |
| **Метод** | **Назначение** |
| suppress\_messages | Принимает блок как аргумент и запрещает любой вывод, сгенерированный этим блоком. |
| say | Принимает сообщение как аргумент и выводит его как есть. Может быть передан второй булевый аргумент для указания, нужен отступ или нет. |
| say\_with\_time | Выводит текст вместе с продолжительностью выполнения блока. Если блок возвращает число, предполагается, что это количество затронутых строк. |

Например, эта миграция

|  |
| --- |
| class CreateProducts < ActiveRecord::Migration    def change      suppress\_messages do        create\_table :products do |t|          t.string :name          t.text :description          t.timestamps        end      end        say "Created a table"        suppress\_messages {add\_index :products, :name}      say "and an index!", true        say\_with\_time 'Waiting for a while' do        sleep 10        250      end    end  end |

сгенерирует следующий результат

|  |
| --- |
| ==  CreateProducts: migrating =================================================  -- Created a table     -> and an index!  -- Waiting for a while     -> 10.0013s     -> 250 rows  ==  CreateProducts: migrated (10.0054s) ======================================= |

Если хотите, чтобы Active Record ничего не выводил, запуск rake db:migrate VERBOSE=false запретит любой вывод.

**Изменение существующих миграций**

Периодически вы будете делать ошибки при написании миграции. Если вы уже запустили миграцию, вы не сможете просто отредактировать миграцию и запустить ее снова: Rails посчитает, что он уже выполнял миграцию, и ничего не сделает при запуске rake db:migrate. Вы должны откатить миграцию (например, с помощью rake db:rollback), отредактировать миграцию и затем запустить rake db:migrate для запуска исправленной версии.

В целом, редактирование существующих миграций не хорошая идея. Вы создадите дополнительную работу себе и своим коллегам, и вызовете море головной боли, если существующая версия миграции уже была запущена в production. Вместо этого, следует написать новую миграцию, выполняющую требуемые изменения. Редактирование только что созданной миграции, которая еще не была закомичена в систему контроля версий (или, хотя бы, не ушла дальше вашей рабочей машины) относительно безвредно.

Метод revert может быть очень полезным при написании новой миграции для возвращения предыдущей в целом или какой то части (смотрите [Возвращение к предыдущим миграциям](http://rusrails.ru/rails-database-migrations#writing-a-migration)

**Использование моделей в ваших миграциях**

При создании или обновлении данных зачастую хочется использовать одну из ваших моделей. Ведь они же существуют, чтобы облегчить доступ к лежащим в их основе данным. Это осуществимо, но с некоторыми предостережениями.

Например, проблемы происходят, когда модель использует столбцы базы данных, которые (1) в текущий момент отсутствуют в базе данных и (2) будут созданы в этой или последующих миграциях.

Рассмотрим пример, когда Алиса и Боб работают над одним и тем же участком кода, содержащим модель Product

Боб ушел в отпуск.

Алиса создала миграцию для таблицы products, добавляющую новый столбец, и инициализировала ее.

|  |
| --- |
| # db/migrate/20100513121110\_add\_flag\_to\_product.rb    class AddFlagToProduct < ActiveRecord::Migration    def change      add\_column :products, :flag, :boolean      reversible do |dir|        dir.up { Product.update\_all flag: false }      end    end  end |

Она также добавила в модели Product валидацию на новый столбец:

|  |
| --- |
| # app/models/product.rb    class Product < ActiveRecord::Base    validates :flag, inclusion: { in: [true, false] }  end |

Алиса добавила вторую миграцию, добавляющую другой столбец в таблице products, и инициализировала ее.

|  |
| --- |
| # db/migrate/20100515121110\_add\_fuzz\_to\_product.rb    class AddFuzzToProduct < ActiveRecord::Migration    def change      add\_column :products, :fuzz, :string      reversible do |dir|        dir.up { Product.update\_all fuzz: 'fuzzy' }      end    end  end |

Она также добавила в модели Product валидацию на новый столбец:

|  |
| --- |
| # app/models/product.rb    class Product < ActiveRecord::Base    validates :flag, inclusion: { in: [true, false] }    validates :fuzz, presence: true  end |

Обе миграции работают для Алисы.

Боб вернулся с отпуска, и:

* Обновил исходники - содержащие обе миграции и последнюю версию модели Product.
* Запустил невыполненные миграции с помощью rake db:migrate, включая обновляющие модель Product.

Миграции не выполнятся, так как при попытке сохранения модели, она попытается валидировать второй добавленный столбец, отсутствующий в базе данных на момент запуска *первой* миграции.

|  |
| --- |
| rake aborted!  An error has occurred, this and all later migrations canceled:    undefined method `fuzz' for #<Product:0x000001049b14a0> |

Это исправляется путем создания локальной модели внутри миграции. Это предохраняет Rails от запуска валидаций, поэтому миграции проходят.

При использовании локальной модели неплохо бы вызвать Product.reset\_column\_information для обновления кэша Active Record для модели Product до обновления данных в базе данных.

Если бы Алиса сделала бы так, проблем бы не было:

|  |
| --- |
| # db/migrate/20100513121110\_add\_flag\_to\_product.rb    class AddFlagToProduct < ActiveRecord::Migration    class Product < ActiveRecord::Base    end      def change      add\_column :products, :flag, :boolean      Product.reset\_column\_information      reversible do |dir|        dir.up { Product.update\_all flag: false }      end    end  end |
| # db/migrate/20100515121110\_add\_fuzz\_to\_product.rb    class AddFuzzToProduct < ActiveRecord::Migration    class Product < ActiveRecord::Base    end      def change      add\_column :products, :fuzz, :string      Product.reset\_column\_information      reversible do |dir|        dir.up { Product.update\_all fuzz: 'fuzzy' }      end    end  end |

Имеется несколько способов, при которых вышеприведенные примеры могут сработать плохо.

Например, представим, что Алиса создала миграцию, избирательно обновляющую поле description для определенных продуктов. Она запускает миграцию, комитит код, и начинает работать над следующей задачей, которая добавляет новый столбец fuzz в таблицу продуктов.

Она создает две миграции для этой новой задачи, одна из которых добавляет новый столбец, а вторая избирательно обновляет столбец fuzz, основываясь на других атрибутах продукта.

Эти миграции прекрасно запускаются, но когда Боб возвращается из отпуска вызывает rake db:migrate для запуска всех невыполненных миграций, он получает неуловимый баг: Описания имеют значения по умолчанию, столбец fuzz присутствует, но fuzz равно nil для всех продуктов.

Решением снова является использование Product.reset\_column\_information до обращения к модели Product в миграции, чтобы убедиться в знании Active Record о текущей структуре таблицы до манипуляции с данными в этих записях.