### Зачем нужны связи?

Зачем нам нужны связи между моделями? Затем, что они позволяют сделать код для обычных операций проще и легче. Например, рассмотрим простое приложение на Rails, которое включает модель для покупателей и модель для заказов. Каждый покупатель может иметь много заказов. Без связей объявление модели будет выглядеть так:

|  |
| --- |
| class Customer < ActiveRecord::Base  end    class Order < ActiveRecord::Base  end |

Теперь, допустим, мы хотим добавить новый заказ для существующего покупателя. Нам нужно сделать так:

|  |
| --- |
| @order = Order.create(order\_date: Time.now, customer\_id: @customer.id) |

Или, допустим, удалим покупателя и убедимся, что все его заказы также будут удалены:

|  |
| --- |
| @orders = Order.where(customer\_id: @customer.id)  @orders.each do |order|    order.destroy  end  @customer.destroy |

Со связями Active Record можно упростить эти и другие операции, декларативно сказав Rails, что имеется соединение между двумя моделями. Вот пересмотренный код для создания покупателей и заказов:

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_many :orders, dependent: :destroy  end    class Order < ActiveRecord::Base    belongs\_to :customer  end |

С этими изменениями создание нового заказа для определенного покупателя проще:

|  |
| --- |
| @order = @customer.orders.create(order\_date: Time.now) |

Удаление покупателя и всех его заказов намного проще:

|  |
| --- |
| @customer.destroy |

Чтобы узнать больше о различных типах связей, читайте следующий раздел руководства. Затем следуют некоторые полезные советы по работе со связями, а затем полное описание методов и опций для связей в Rails.

### Типы связей

В Rails связи - это соединения между двумя моделями Active Record. Связи реализовываются с использованием макро-вызовов (macro-style calls), и таким образом вы можете декларативно добавлять возможности для своих моделей. Например, объявляя, что одна модель принадлежит (belongs\_to) другой, вы указываете Rails сохранять информацию о первичном-внешнем ключах между экземплярами двух моделей, а также получаете несколько полезных методов, добавленных в модель. Rails поддерживает шесть типов связей:

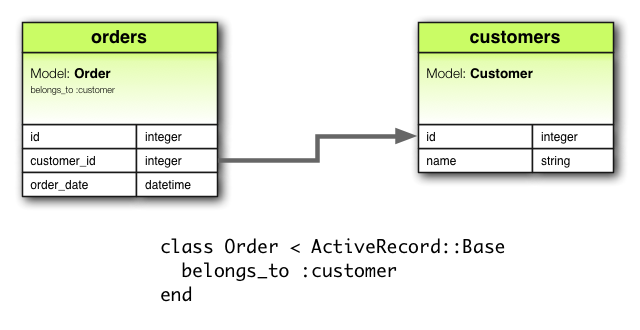
* belongs\_to
* has\_one
* has\_many
* has\_many :through
* has\_one :through
* has\_and\_belongs\_to\_many

После прочтения всего этого руководства, вы научитесь объявлять и использовать различные формы связей. Но сначала следует быстро ознакомиться с ситуациями, когда применим каждый тип связи.

#### Связь belongs\_to

Связь belongs\_to устанавливает соединение один-к-одному с другой моделью, когда один экземпляр объявляющей модели "принадлежит" одному экземпляру другой модели. Например, если в приложении есть покупатели и заказы, и один заказ может быть связан только с одним покупателем, нужно объявить модель order следующим образом:

|  |
| --- |
| class Order < ActiveRecord::Base    belongs\_to :customer  end |



связи belongs\_to обязаны использовать единственное число. Если использовать множественное число в вышеприведенном примере для связиcustomer в модели Order, вам будет сообщено "uninitialized constant Order::Customers". Это так, потому что Rails автоматически получает имя класса из имени связи. Если в имени связи неправильно использовано число, то получаемый класс также будет неправильного числа.

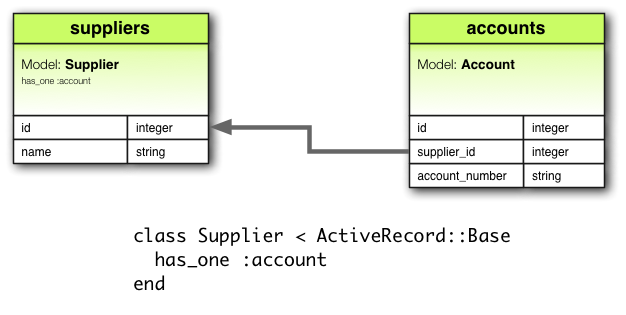
Соответствующая миграция может выглядеть так:

|  |
| --- |
| class CreateOrders < ActiveRecord::Migration    def change      create\_table :customers do |t|        t.string :name        t.timestamps      end        create\_table :orders do |t|        t.belongs\_to :customer        t.datetime :order\_date        t.timestamps      end    end  end |

#### Связь has\_one

Связь has\_one также устанавливает соединение один-к-одному с другой моделью, но в несколько ином смысле (и с другими последствиями). Эта связь показывает, что каждый экземпляр модели содержит или обладает одним экземпляром другой модели. Например, если каждый поставщик имеет только один аккаунт, можете объявить модель supplier подобно этому:

|  |
| --- |
| class Supplier < ActiveRecord::Base    has\_one :account  end |



Соответствующая миграция может выглядеть так:

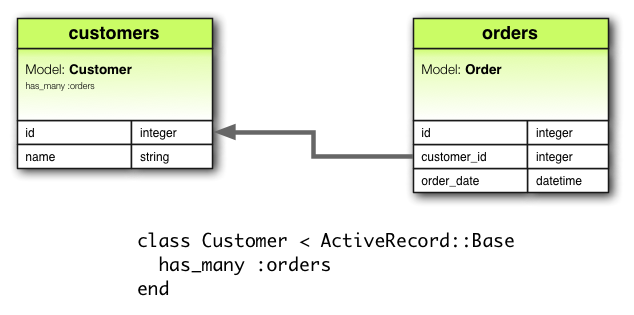
|  |
| --- |
| class CreateSuppliers < ActiveRecord::Migration    def change      create\_table :suppliers do |t|        t.string :name        t.timestamps      end        create\_table :accounts do |t|        t.belongs\_to :supplier        t.string :account\_number        t.timestamps      end    end  end |

#### Связь has\_many

Связь has\_many указывает на соединение один-ко-многим с другой моделью. Эта связь часто бывает на "другой стороне" связи belongs\_to. Эта связь указывает на то, что каждый экземпляр модели имеет ноль или более экземпляров другой модели. Например, в приложении, содержащем покупателей и заказы, модель customer может быть объявлена следующим образом:

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_many :orders  end |

Имя другой модели указывается во множественном числе при объявлении связи has\_many.



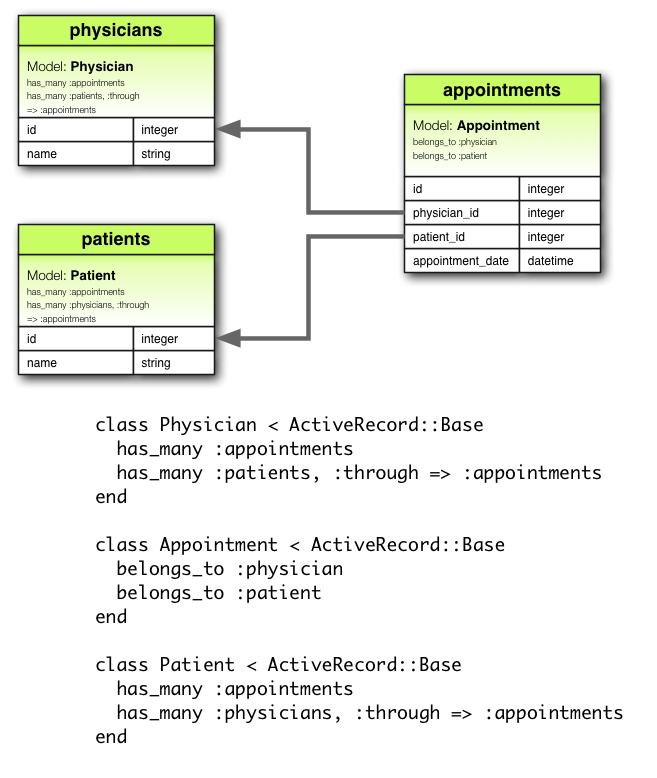
Соответствующая миграция может выглядеть так:

|  |
| --- |
| class CreateCustomers < ActiveRecord::Migration    def change      create\_table :customers do |t|        t.string :name        t.timestamps      end        create\_table :orders do |t|        t.belongs\_to :customer        t.datetime :order\_date        t.timestamps      end    end  end |

#### Связь has\_many :through

Связь has\_many :through часто используется для настройки соединения многие-ко-многим с другой моделью. Эта связь указывает, что объявляющая модель может соответствовать нулю или более экземплярам другой модели через третью модель. Например, рассмотрим поликлинику, где пациентам (patients) дают направления (appointments) к врачам (physicians). Соответствующие объявления связей будут выглядеть следующим образом:

|  |
| --- |
| class Physician < ActiveRecord::Base    has\_many :appointments    has\_many :patients, through: :appointments  end    class Appointment < ActiveRecord::Base    belongs\_to :physician    belongs\_to :patient  end    class Patient < ActiveRecord::Base    has\_many :appointments    has\_many :physicians, through: :appointments  end |



Соответствующая миграция может выглядеть так:

|  |
| --- |
| class CreateAppointments < ActiveRecord::Migration    def change      create\_table :physicians do |t|        t.string :name        t.timestamps      end        create\_table :patients do |t|        t.string :name        t.timestamps      end        create\_table :appointments do |t|        t.belongs\_to :physician        t.belongs\_to :patient        t.datetime :appointment\_date        t.timestamps      end    end  end |

Коллекция соединительных моделей может управляться с помощью API. Например, если вы присвоите:

|  |
| --- |
| physician.patients = patients |

будет создана новая соединительная модель для вновь связанных объектов, и если некоторые из них закончатся, их строки будут удалены.

Автоматическое удаление соединительных моделей прямое, ни один из колбэков на уничтожение не включается.

Связь has\_many :through также полезна для настройки "ярлыков" через вложенные связи has\_many. Например, если документ имеет много секций, а секция имеет много параграфов, иногда хочется получить просто коллекцию всех параграфов в документе. Это можно настроить следующим образом:

|  |
| --- |
| class Document < ActiveRecord::Base    has\_many :sections    has\_many :paragraphs, through: :sections  end    class Section < ActiveRecord::Base    belongs\_to :document    has\_many :paragraphs  end    class Paragraph < ActiveRecord::Base    belongs\_to :section  end |

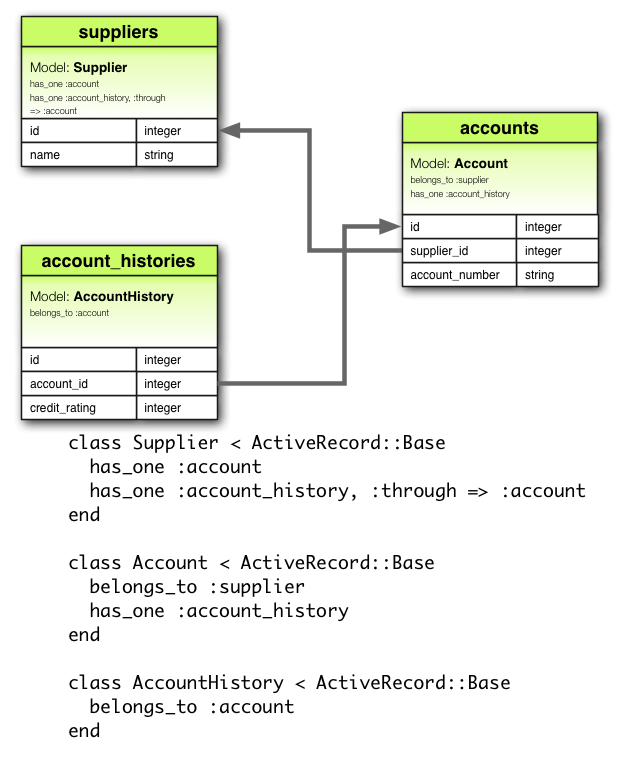
С определенным through: :sections Rails теперь понимает:

|  |
| --- |
| @document.paragraphs |

#### Связь has\_one :through

Связь has\_one :through настраивает соединение один-к-одному с другой моделью. Эта связь показывает, что объявляющая модель может быть связана с одним экземпляром другой модели через третью модель. Например, если каждый поставщик имеет один аккаунт, и каждый аккаунт связан с одной историей аккаунта, тогда модели могут выглядеть так:

|  |
| --- |
| class Supplier < ActiveRecord::Base    has\_one :account    has\_one :account\_history, through: :account  end    class Account < ActiveRecord::Base    belongs\_to :supplier    has\_one :account\_history  end    class AccountHistory < ActiveRecord::Base    belongs\_to :account  end |



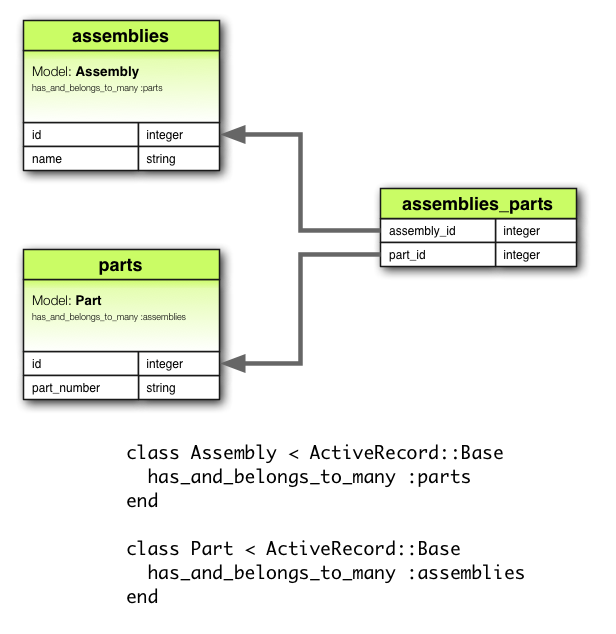
Соответствующая миграция может выглядеть так:

|  |
| --- |
| class CreateAccountHistories < ActiveRecord::Migration    def change      create\_table :suppliers do |t|        t.string :name        t.timestamps      end        create\_table :accounts do |t|        t.belongs\_to :supplier        t.string :account\_number        t.timestamps      end        create\_table :account\_histories do |t|        t.belongs\_to :account        t.integer :credit\_rating        t.timestamps      end    end  end |

#### Связь has\_and\_belongs\_to\_many

Связь has\_and\_belongs\_to\_many создает прямое соединение многие-ко-многим с другой моделью, без промежуточной модели. Например, если ваше приложение включает узлы (assemblies) и детали (parts), где каждый узел имеет много деталей, и каждая деталь встречается во многих узлах, модели можно объявить таким образом:

|  |
| --- |
| class Assembly < ActiveRecord::Base    has\_and\_belongs\_to\_many :parts  end    class Part < ActiveRecord::Base    has\_and\_belongs\_to\_many :assemblies  end |



Соответствующая миграция может выглядеть так:

|  |
| --- |
| class CreateAssembliesAndParts < ActiveRecord::Migration    def change      create\_table :assemblies do |t|        t.string :name        t.timestamps      end        create\_table :parts do |t|        t.string :part\_number        t.timestamps      end        create\_table :assemblies\_parts do |t|        t.belongs\_to :assembly        t.belongs\_to :part      end    end  end |

#### Выбор между belongs\_to и has\_one

Если хотите настроить отношение один-к-одному между двумя моделями, необходимо добавить belongs\_to к одной и has\_one к другой. Как узнать что к какой?

Различие в том, где помещен внешний ключ (он должен быть в таблице для класса, объявляющего связь belongs\_to), но вы также должны думать о реальном значении данных. Отношение has\_one говорит, что что-то принадлежит вам - то есть что что-то указывает на вас. Например, больше смысла в том, что поставщик владеет аккаунтом, чем в том, что аккаунт владеет поставщиком. Это означает, что правильные отношения подобны этому:

|  |
| --- |
| class Supplier < ActiveRecord::Base    has\_one :account  end    class Account < ActiveRecord::Base    belongs\_to :supplier  end |

Соответствующая миграция может выглядеть так:

|  |
| --- |
| class CreateSuppliers < ActiveRecord::Migration    def change      create\_table :suppliers do |t|        t.string  :name        t.timestamps      end        create\_table :accounts do |t|        t.integer :supplier\_id        t.string  :account\_number        t.timestamps      end    end  end |

Использование t.integer :supplier\_id указывает имя внешнего ключа очевидно и явно. В современных версиях Rails можно абстрагироваться от деталей реализации используя t.references :supplier.

#### Выбор между has\_many :through и has\_and\_belongs\_to\_many

Rails предлагает два разных способа объявления отношения многие-ко-многим между моделями. Простейший способ - использовать has\_and\_belongs\_to\_many, который позволяет создать связь напрямую:

|  |
| --- |
| class Assembly < ActiveRecord::Base    has\_and\_belongs\_to\_many :parts  end    class Part < ActiveRecord::Base    has\_and\_belongs\_to\_many :assemblies  end |

Второй способ объявить отношение многие-ко-многим - использование has\_many :through. Это осуществляет связь не напрямую, а через соединяющую модель:

|  |
| --- |
| class Assembly < ActiveRecord::Base    has\_many :manifests    has\_many :parts, through: :manifests  end    class Manifest < ActiveRecord::Base    belongs\_to :assembly    belongs\_to :part  end    class Part < ActiveRecord::Base    has\_many :manifests    has\_many :assemblies, through: :manifests  end |

Простейший признак того, что нужно настраивать отношение has\_many :through - если необходимо работать с моделью отношений как с независимым объектом. Если вам не нужно ничего делать с моделью отношений, проще настроить связь has\_and\_belongs\_to\_many (хотя нужно не забыть создать соединяющую таблицу в базе данных).

Вы должны использовать has\_many :through, если нужны валидации, колбэки или дополнительные атрибуты для соединительной модели.

#### Полиморфные связи

Полиморфные связи - это немного более "навороченный" вид связей. С полиморфными связями модель может принадлежать более чем одной модели, на одиночной связи. Например, имеется модель изображения, которая принадлежит или модели работника, или модели продукта. Вот как это объявляется:

|  |
| --- |
| class Picture < ActiveRecord::Base    belongs\_to :imageable, polymorphic: true  end    class Employee < ActiveRecord::Base    has\_many :pictures, as: :imageable  end    class Product < ActiveRecord::Base    has\_many :pictures, as: :imageable  end |

Можете считать полиморфное объявление belongs\_to как настройку интерфейса, который может использовать любая другая модель. Из экземпляра модели Employee можно получить коллекцию изображений: @employee.pictures.

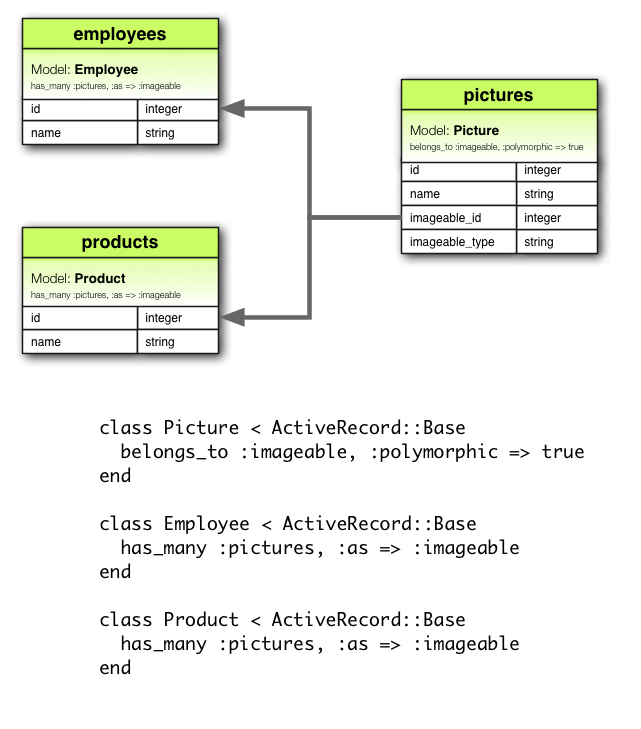
Подобным образом можно получить @product.pictures.

Если имеется экземпляр модели Picture, можно получить его родителя посредством @picture.imageable. Чтобы это работало, необходимо объявить столбец внешнего ключа и столбец типа в модели, объявляющей полиморфный интерфейс:

|  |
| --- |
| class CreatePictures < ActiveRecord::Migration    def change      create\_table :pictures do |t|        t.string  :name        t.integer :imageable\_id        t.string  :imageable\_type        t.timestamps      end    end  end |

Эта миграция может быть упрощена при использовании формы t.references:

|  |
| --- |
| class CreatePictures < ActiveRecord::Migration    def change      create\_table :pictures do |t|        t.string :name        t.references :imageable, polymorphic: true        t.timestamps      end    end  end |



#### Самоприсоединение

При разработке модели данных иногда находится модель, которая может иметь отношение сама к себе. Например, мы хотим хранить всех работников в одной модели базы данных, но нам нужно отслеживать отношения начальник-подчиненный. Эта ситуация может быть смоделирована с помощью самоприсоединяемых связей:

|  |
| --- |
| class Employee < ActiveRecord::Base    has\_many :subordinates, class\_name: "Employee",                            foreign\_key: "manager\_id"      belongs\_to :manager, class\_name: "Employee"  end |

С такой настройкой, вы можете получить @employee.subordinates и @employee.manager.

### Полезные советы и предупреждения

Вот некоторые вещи, которые необходимо знать для эффективного использования связей Active Record в Вашем приложении на Rails:

* Управление кэшированием
* Предотвращение коллизий имен
* Обновление схемы
* Управление областью видимости связей
* Двусторонние связи

#### Управление кэшированием

Все методы связи построены вокруг кэширования, которое хранит результаты последних запросов доступными для будущих операций. Кэш является общим для разных методов. Например:

|  |
| --- |
| customer.orders                 # получаем заказы из базы данных  customer.orders.size            # используем кэшированную копию заказов  customer.orders.empty?          # используем кэшированную копию заказов |

Но что если вы хотите перезагрузить кэш, так как данные могли быть изменены другой частью приложения? Всего лишь передайте true в вызов связи:

|  |
| --- |
| customer.orders                 # получаем заказы из базы данных  customer.orders.size            # используем кэшированную копию заказов  customer.orders(true).empty?    # отказываемся от кэшированной копии заказов                                  # и снова обращаемся к базе данных |

#### Предотвращение коллизий имен

Вы не свободны в выборе любого имени для своих связей. Поскольку создание связи добавляет метод с таким именем в модель, будет плохой идеей дать связи имя, уже используемое как метод экземпляра ActiveRecord::Base. Метод связи тогда переопределит базовый метод, и что-нибудь перестанет работать. Например, attributes или connection плохие имена для связей.

#### Обновление схемы

Связи очень полезные, но не волшебные. Вы ответственны за содержание вашей схемы базы данных в соответствии со связями. На практике это означает две вещи, в зависимости от того, какой тип связей создаете. Для связей belongs\_to нужно создать внешние ключи, а для связей has\_and\_belongs\_to\_many нужно создать подходящую соединительную таблицу.

##### Создание внешних ключей для связей belongs\_to

Когда объявляете связь belongs\_to, нужно создать внешние ключи, при необходимости. Например, рассмотрим эту модель:

|  |
| --- |
| class Order < ActiveRecord::Base    belongs\_to :customer  end |

Это объявление нуждается в создании подходящего внешнего ключа в таблице orders:

|  |
| --- |
| class CreateOrders < ActiveRecord::Migration    def change      create\_table :orders do |t|        t.datetime :order\_date        t.string   :order\_number        t.integer  :customer\_id      end    end  end |

Если создаете связь после того, как уже создали модель, лежащую в основе, необходимо не забыть создать миграцию add\_column для предоставления необходимого внешнего ключа.

##### Создание соединительных таблиц для связей has\_and\_belongs\_to\_many

Если вы создали связь has\_and\_belongs\_to\_many, необходимо обязательно создать соединительную таблицу. Если имя соединительной таблицы явно не указано с использованием опции :join\_table, Active Record создает имя, используя алфавитный порядок имен классов. Поэтому соединение между моделями customer и order по умолчанию даст значение имени таблицы "customers\_orders", так как "c" идет перед "o" в алфавитном порядке.

Приоритет между именами модели рассчитывается с использованием оператора < для String. Это означает, что если строки имеют разную длину. и в своей короткой части они равны, тогда более длинная строка рассматривается как большая, по сравнению с короткой. Например, кто-то ожидает, что таблицы "paper\_boxes" и "papers" создадут соединительную таблицу "papers\_paper\_boxes" поскольку имя "paper\_boxes" длинее, но фактически будет сгенерирована таблица с именем "paper\_boxes\_papers" (поскольку знак подчеркивания "\_" лексикографически меньше, чем "s" в обычной кодировке).

Какое бы ни было имя, вы должны вручную сгенерировать соединительную таблицу в соответствующей миграции. Например, рассмотрим эти связи:

|  |
| --- |
| class Assembly < ActiveRecord::Base    has\_and\_belongs\_to\_many :parts  end    class Part < ActiveRecord::Base    has\_and\_belongs\_to\_many :assemblies  end |

Теперь нужно написать миграцию для создания таблицы assemblies\_parts. Эта таблица должна быть создана без первичного ключа:

|  |
| --- |
| class CreateAssembliesPartsJoinTable < ActiveRecord::Migration    def change      create\_table :assemblies\_parts, id: false do |t|        t.integer :assembly\_id        t.integer :part\_id      end    end  end |

Мы передаем id: false в create\_table, так как эта таблица не представляет модель. Это необходимо, чтобы связь работала правильно. Если вы видите странное поведение в связи has\_and\_belongs\_to\_many, например, искаженные ID моделей, или исключения в связи с конфликтом ID, скорее всего вы забыли убрать первичный ключ.

#### Управление областью видимости связей

По умолчанию связи ищут объекты только в пределах области видимости текущего модуля. Это важно, когда вы объявляете модели Active Record внутри модуля. Например:

|  |
| --- |
| module MyApplication    module Business      class Supplier < ActiveRecord::Base         has\_one :account      end        class Account < ActiveRecord::Base         belongs\_to :supplier      end    end  end |

Это будет работать, так как оба класса Supplier и Account определены в пределах одной области видимости. Но нижеследующее не будет работать, потому что Supplier и Account определены в разных областях видимости:

|  |
| --- |
| module MyApplication    module Business      class Supplier < ActiveRecord::Base         has\_one :account      end    end      module Billing      class Account < ActiveRecord::Base         belongs\_to :supplier      end    end  end |

Для связи модели с моделью в другом пространстве имен, необходимо указать полное имя класса в объявлении связи:

|  |
| --- |
| module MyApplication    module Business      class Supplier < ActiveRecord::Base         has\_one :account,          class\_name: "MyApplication::Billing::Account"      end    end      module Billing      class Account < ActiveRecord::Base         belongs\_to :supplier,          class\_name: "MyApplication::Business::Supplier"      end    end  end |

#### Двусторонние связи

Для связей нормально работать в двух направлениях, затребовав объявление в двух различных моделях:

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_many :orders  end    class Order < ActiveRecord::Base    belongs\_to :customer  end |

По умолчанию, Active Record не знает о зависимости между этими двумя связями. Это может привести к двум несинхронизированным копиям объекта:

|  |
| --- |
| c = Customer.first  o = c.orders.first  c.first\_name == o.customer.first\_name # => true  c.first\_name = 'Manny'  c.first\_name == o.customer.first\_name # => false |

Это произошло потому, что c и o.customer это два разных представления в памяти одних и тех же данных, и ни одно из них автоматически не обновляется при изменении другого. Active Record предоставляет опцию :inverse\_of, чтобы вы могли его проинформировать об этих зависимостях:

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_many :orders, inverse\_of: :customer  end    class Order < ActiveRecord::Base    belongs\_to :customer, inverse\_of: :orders  end |

С этими изменениями Active Record загрузит только одну копию объекта customer, предотвратив несоответствия и сделав приложение более эффективным:

|  |
| --- |
| c = Customer.first  o = c.orders.first  c.first\_name == o.customer.first\_name # => true  c.first\_name = 'Manny'  c.first\_name == o.customer.first\_name # => true |

Имеется несколько ограничений в поддержке inverse\_of:

* Они не работают со связями :through.
* Они не работают со связями :polymorphic.
* Они не работают со связями :as.
* Для связей belongs\_to противоположные связи has\_many игнорируются.

Каждая связь попытается автоматически найти противоположную связь и установить опцию :inverse\_of эвристически (основываясь на имени связи). Поддерживается большинство связей со стандартными именами. Однако, связям, содержащим следующие опции, противоположности не будут установлены автоматически:

* :conditions
* :through
* :polymorphic
* :foreign\_key

### Подробная информация по связи belongs\_to

Связь belongs\_to создает соответствие один-к-одному с другой моделью. В терминах базы данных эта связь сообщает, что этот класс содержит внешний ключ. Если внешний ключ содержит другой класс, вместо этого следует использовать has\_one.

#### Методы, добавляемые belongs\_to

Когда объявляете связь belongs\_to, объявляющий класс автоматически получает четыре метода, относящихся к связи:

* association(force\_reload = false)
* association=(associate)
* build\_association(attributes = {})
* create\_association(attributes = {})
* create\_association!(attributes = {})

Во всех четырех методах association заменяется символом, переданным как первый аргумент в belongs\_to. Например, имеем объявление:

|  |
| --- |
| class Order < ActiveRecord::Base    belongs\_to :customer  end |

Каждый экземпляр модели order будет иметь эти методы:

|  |
| --- |
| customer  customer=  build\_customer  create\_customer  create\_customer! |

Когда устанавливаете новую связь has\_one или belongs\_to, следует использовать префикс build\_ для построения связи, в отличие от метода association.build, используемый для связей has\_many или has\_and\_belongs\_to\_many. Чтобы создать связь, используйте префикс create\_.

##### association(force\_reload = false)`

Метод association возвращает связанный объект, если он есть. Если объекта нет, возвращает nil.

|  |
| --- |
| @customer = @order.customer |

Если связанный объект уже был получен из базы данных для этого объекта, возвращается кэшированная версия. Чтобы переопределить это поведение (и заставить прочитать из базы данных), передайте true как аргумент force\_reload.

##### association=(associate)

Метод association= привязывает связанный объект к этому объекту. Фактически это означает извлечение первичного ключа из связанного объекта и присвоение его значения внешнему ключу.

|  |
| --- |
| @order.customer = @customer |

##### build\_association(attributes = {})

Метод build\_association возвращает новый объект связанного типа. Этот объект будет экземпляром с переданными атрибутами, будет установлена связь с внешним ключом этого объекта, но связанный объект пока не будет сохранен.

|  |
| --- |
| @customer = @order.build\_customer(customer\_number: 123,                                    customer\_name: "John Doe") |

##### create\_association(attributes = {})

Метод create\_association возвращает новый объект связанного типа. Этот объект будет экземпляром с переданными атрибутами, будет установлена связь с внешним ключом этого объекта, и, если он пройдет валидации, определенные в связанной модели, связанный объект будет сохранен.

|  |
| --- |
| @customer = @order.create\_customer(customer\_number: 123,                                     customer\_name: "John Doe") |

##### create\_association!(attributes = {})

Работает так же, как и вышеприведенный create\_association, но вызывает ActiveRecord::RecordInvalid, если запись невалидна.

#### Опции для belongs\_to

Хотя Rails использует разумные значения по умолчанию, работающие во многих ситуациях, бывают случаи, когда хочется изменить поведение связи belongs\_to. Такая настройка легко выполнима с помощью передачи опций и блоков со скоупом при создании связи. Например, эта связь использует две такие опции:

|  |
| --- |
| class Order < ActiveRecord::Base    belongs\_to :customer, dependent: :destroy,      counter\_cache: true  end |

Связь belongs\_to поддерживает эти опции:

* :autosave
* :class\_name
* :counter\_cache
* :dependent
* :foreign\_key
* :inverse\_of
* :polymorphic
* :touch
* :validate

##### :autosave

Если установить опцию :autosave в true, Rails сохранит любые загруженные члены и уничтожит члены, помеченные для уничтожения, всякий раз, когда вы сохраните родительский объект.

##### :class\_name

Если имя другой модели не может быть получено из имени связи, можете использовать опцию :class\_name для предоставления имени модели. Например, если заказ принадлежит покупателю, но фактическое имя модели, содержащей покупателей Patron, можете установить это следующим образом:

|  |
| --- |
| class Order < ActiveRecord::Base    belongs\_to :customer, class\_name: "Patron"  end |

##### :counter\_cache

Опция :counter\_cache может быть использована, чтобы сделать поиск количества принадлежацих объектов более эффективным. Рассмотрим эти модели:

|  |
| --- |
| class Order < ActiveRecord::Base    belongs\_to :customer  end  class Customer < ActiveRecord::Base    has\_many :orders  end |

С этими объявлениями запрос значения @customer.orders.size требует обращения к базе данных для выполнения запроса COUNT(\*). Чтобы этого избежать, можете добавить кэш счетчика в принадлежащую модель:

|  |
| --- |
| class Order < ActiveRecord::Base    belongs\_to :customer, counter\_cache: true  end  class Customer < ActiveRecord::Base    has\_many :orders  end |

С этим объявлением, Rails будет хранить в кэше актуальное значение и затем возвращать это значение в ответ на метод size.

Хотя опция :counter\_cache определяется в модели, включающей определение belongs\_to, фактический столбец должен быть добавлен в связанную модель. В вышеописанном случае, необходимо добавить столбец, названный orders\_count в модель Customer. Имя столбца по умолчанию можно переопределить, если вы этого желаете:

|  |
| --- |
| class Order < ActiveRecord::Base    belongs\_to :customer, counter\_cache: :count\_of\_orders  end  class Customer < ActiveRecord::Base    has\_many :orders  end |

Столбцы кэша счетчика добавляются в список атрибутов модели только для чтения посредством attr\_readonly.

##### :dependent

Если установить опцию :dependent как :destroy, тогда удаление этого объекта вызовет метод destroy у связанного объекта, для удаление того объекта. Если установить опцию :dependent как :delete, тогда удаление этого объекта удалит связанный объект без вызова его метода destroy. Если установить опцию :dependent как :restrict, тогда попытка удалить этот объект приведет к ActiveRecord::DeleteRestrictionError, если существуют какие-либо связанные объекты.

Не следует определять эту опцию в связи belongs\_to, которая соединена со связью has\_many в другом классе. Это приведет к "битым" связям в записях вашей базы данных.

##### :foreign\_key

По соглашению Rails предполагает, что столбец, используемый для хранения внешнего ключа в этой модели, имеет имя модели с добавленным суффиксом \_id. Опция :foreign\_key позволяет установить имя внешнего ключа явно:

|  |
| --- |
| class Order < ActiveRecord::Base    belongs\_to :customer, class\_name: "Patron",                          foreign\_key: "patron\_id"  end |

В любом случае, Rails не создаст столбцы внешнего ключа за вас. Вам необходимо явно определить их в своих миграциях.

##### :inverse\_of

Опция :inverse\_of определяет имя связи has\_many или has\_one, являющейся противополжностью для этой связи. Не работает в комбинации с опциями :polymorphic.

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_many :orders, inverse\_of: :customer  end    class Order < ActiveRecord::Base    belongs\_to :customer, inverse\_of: :orders  end |

##### :polymorphic

Передача true для опции :polymorphic показывает, что это полиморфная связь. Полиморфные связи подробно рассматривались [ранее](http://rusrails.ru/active-record-associations#polymorphic-associations).

##### :touch

Если установите опцию :touch в :true, то временные метки updated\_at или updated\_on на связанном объекте будут установлены в текущее время всякий раз, когда этот объект будет сохранен или уничтожен:

|  |
| --- |
| class Order < ActiveRecord::Base    belongs\_to :customer, touch: true  end    class Customer < ActiveRecord::Base    has\_many :orders  end |

В этом случае, сохранение или уничтожение заказа обновит временную метку на связанном покупателе. Также можно определить конкретный атрибут временной метки для обновления:

|  |
| --- |
| class Order < ActiveRecord::Base    belongs\_to :customer, touch: :orders\_updated\_at  end |

##### :validate

Если установите опцию :validate в true, тогда связанные объекты будут проходить валидацию всякий раз, когда вы сохраняете этот объект. По умолчанию она равна false: связанные объекты не проходят валидацию, когда этот объект сохраняется.

#### Скоупы для belongs\_to

Иногда хочется настроить запрос, используемый belongs\_to. Такая настройка может быть достигнута с помощью блока скоупа. Например:

|  |
| --- |
| class Order < ActiveRecord::Base    belongs\_to :customer, -> { where active: true },                          dependent: :destroy  end |

Внутри блока скоупа можно использовать любые стандартные "методы запросов":/active-record-query-interface. Далее обсудим следующие из них:

* where
* includes
* readonly
* select

##### where

Метод where позволяет определить условия, которым должен отвечать связанный объект.

|  |
| --- |
| class Order < ActiveRecord::Base    belongs\_to :customer, -> { where active: true }  end |

##### includes

Метод includes можно использовать для определения связей второго порядка, которые должны быть лениво загружены при использовании этой связи. Например, рассмотрим эти модели:

|  |
| --- |
| class LineItem < ActiveRecord::Base    belongs\_to :order  end    class Order < ActiveRecord::Base    belongs\_to :customer    has\_many :line\_items  end    class Customer < ActiveRecord::Base    has\_many :orders  end |

Если вы часто получаете покупателей непосредственно из элементов заказа (@line\_item.order.customer), то можно улучшить эффективность кода, включив покупателей в связь между заказом и его элементами:

|  |
| --- |
| class LineItem < ActiveRecord::Base    belongs\_to :order, -> { includes :customer }  end    class Order < ActiveRecord::Base    belongs\_to :customer    has\_many :line\_items  end    class Customer < ActiveRecord::Base    has\_many :orders  end |

Нет необходимости в использовании includes для ближайших связей - то есть, если есть Order belongs\_to :customer, то customer автоматически лениво загружается при необходимости.

##### readonly

При использовании readonly, связанный объект будет только для чтения при получении через связь.

##### select

Метод select позволяет переопределить SQL выражение SELECT, используемое для получения данных о связанном объекте. По умолчанию Rails получает все столбцы.

При использовании метода select на связи belongs\_to, следует также установить опцию :foreign\_key для гарантии правильных результатов.

#### Существуют ли связанные объекты?

Можно увидеть, существует ли какой-либо связанный объект, при использовании метода association.nil?:

|  |
| --- |
| if @order.customer.nil?    @msg = "No customer found for this order"  end |

#### Когда сохраняются объекты?

Присвоение связи belongs\_to не приводит к автоматическому сохранению ни самого объекта, ни связанного объекта.

### Подробная информация по связи has\_one

Связь has\_one создает соответствие один-к-одному с другой моделью. В терминах базы данных эта связь сообщает, что другой класс содержит внешний ключ. Если этот класс содержит внешний ключ, следует использовать belongs\_to.

#### Методы, добавляемые has\_one

Когда объявляете связь has\_one, объявляющий класс автоматически получает четыре метода, относящихся к связи: \* association(force\_reload = false) \* association=(associate) \* build\_association(attributes = {}) \* create\_association(attributes = {}) \* create\_association!(attributes = {})

Во всех этих методах association заменяется на символ, переданный как первый аргумент в has\_one. Например, имеем объявление:

|  |
| --- |
| class Supplier < ActiveRecord::Base    has\_one :account  end |

Каждый экземпляр модели Supplier будет иметь эти методы:

|  |
| --- |
| account  account=  build\_account  create\_account  create\_account! |

При устанавлении новой связи has\_one или belongs\_to, следует использовать префикс build\_ для построения связи, в отличие от метода association.build, используемого для связей has\_many или has\_and\_belongs\_to\_many. Чтобы создать связь, используйте префикс create\_.

##### association(force\_reload = false)

Метод association возвращает связанный объект, если таковой имеется. Если связанный объект не найден, возвращает nil.

|  |
| --- |
| @account = @supplier.account |

Если связанный объект уже был получен из базы данных для этого объекта, возвращается кэшированная версия. Чтобы переопределить это поведение (и заставить прочитать из базы данных), передайте true как аргумент force\_reload.

##### association=(associate)

Метод association= привязывает связанный объект к этому объекту. Фактически это означает извлечение первичного ключа этого объекта и присвоение его значения внешнему ключу связанного объекта.

|  |
| --- |
| @supplier.account = @account |

##### build\_association(attributes = {})

Метод build\_association возвращает новый объект связанного типа. Этот объект будет экземпляром с переданными атрибутами, и будет установлена связь через внешний ключ, но связанный объект не будет пока сохранен.

|  |
| --- |
| @account = @supplier.build\_account(terms: "Net 30") |

##### create\_association(attributes = {})

Метод create\_association возвращает новый объект связанного типа. Этот объект будет экземпляром с переданными атрибутами, будет установлена связь через внешний ключ, и, если он пройдет валидации, определенные в связанной модели, связанный объект будет сохранен

|  |
| --- |
| @account = @supplier.create\_account(terms: "Net 30") |

##### create\_association!(attributes = {})

Работает так же, как и вышеприведенный create\_association, но вызывает ActiveRecord::RecordInvalid, если запись невалидна.

#### Опции для has\_one

Хотя Rails использует разумные значения по умолчанию, работающие во многих ситуациях, бывают случаи, когда хочется изменить поведение связи has\_one. Такая настройка легко выполнима с помощью передачи опции при создании связи. Например, эта связь использует две такие опции:

|  |
| --- |
| class Supplier < ActiveRecord::Base    has\_one :account, class\_name: "Billing", dependent: :nullify  end |

Связь has\_one поддерживает эти опции:

* :as
* :autosave
* :class\_name
* :dependent
* :foreign\_key
* :inverse\_of
* :primary\_key
* :source
* :source\_type
* :through
* :validate

##### :as

Установка опции :as показывает, что это полиморфная связь. Полиморфные связи подробно рассматривались [ранее](http://rusrails.ru/active-record-associations#polymorphic-associations).

##### :autosave

Если установить опцию :autosave в true, это сохранит любые загруженные члены и уничтожит члены, помеченные для уничтожения, всякий раз, когда вы сохраните родительский объект.

##### :class\_name

Если имя другой модели не может быть образовано из имени связи, можете использовать опцию :class\_name для предоставления имени модели. Например, если поставщик имеет аккаунт, но фактическое имя модели, содержащей аккаунты, это Billing, можете установить это следующим образом:

|  |
| --- |
| class Supplier < ActiveRecord::Base    has\_one :account, class\_name: "Billing"  end |

##### :dependent

Управляет тем, что произойдет со связанным объектом, когда его владелец будет уничтожен:

* :destroy приведет к тому, что связанный объект также будет уничтожен
* :delete приведет к тому, что связанный объект будет удален из базы данных напрямую (таким образом не будут выполнены колбэки)
* :nullify приведет к тому, что внешний ключ будет установлен NULL. Колбэки не запускаются.
* :restrict\_with\_exception приведет к вызову исключения, если есть связанный объект
* :restrict\_with\_error приведет к ошибке, добавляемой к владельцу, если есть связанный объект

Нельзя устанавливать или уставлять опцию :nullify для свзей, имеющих ограничение NOT NULL. Если не установить dependent для уничтожения таких связей, вы не сможете изменить связанный объект, акт как внешнему ключу изначально связанного объекта будет назначено недопустимое значение NULL.

##### :foreign\_key

По соглашению Rails предполагает, что столбец, используемый для хранения внешнего ключа в этой модели, имеет имя модели с добавленным суффиксом \_id. Опция :foreign\_key позволяет установить имя внешнего ключа явно:

|  |
| --- |
| class Supplier < ActiveRecord::Base    has\_one :account, foreign\_key: "supp\_id"  end |

В любом случае, Rails не создаст столбцы внешнего ключа за вас. Вам необходимо явно определить их в своих миграциях.

##### :inverse\_of

Опция :inverse\_of определяет имя связи belongs\_to, являющейся обратной для этой связи. Не работает в комбинации с опциями :through или :as.

|  |
| --- |
| class Supplier < ActiveRecord::Base    has\_one :account, inverse\_of: :supplier  end    class Account < ActiveRecord::Base    belongs\_to :supplier, inverse\_of: :account  end |

##### :primary\_key

По соглашению, Rails предполагает, что столбец, используемый для хранения первичного ключа, это id. Вы можете переопределить это и явно определить первичный ключ с помощью опции :primary\_key.

##### :source

Опция :source определяет имя источника связи для связи has\_one :through.

##### :source\_type

Опция :source\_type определяет тип источника связи для связи has\_one :through, который действует при полиморфной связи.

##### :through

Опция :through определяет соединительную модель, через которую выполняется запрос. Связи has\_one :through подробно рассматривались [ранее](http://rusrails.ru/active-record-associations#the-has-one-through-association).

##### :validate

Если установите опцию :validate в true, тогда связанные объекты будут проходить валидацию всякий раз, когда вы сохраняете этот объект. По умолчанию она равна false: связанные объекты не проходят валидацию, когда этот объект сохраняется.

#### Скоупы для has\_one

Иногда хочется настроить запрос, используемый has\_one. Такая настройка может быть достигнута с помощью блока скоупа. Например:

|  |
| --- |
| class Supplier < ActiveRecord::Base    has\_one :account, -> { where active: true }  end |

Внутри блока скоупа можно использовать любые стандартные "методы запросов":/active-record-query-interface. Далее обсудим следующие из них:

* where
* includes
* readonly
* select

##### where

Метод where позволяет определить условия, которым должен отвечать связанный объект.

|  |
| --- |
| class Supplier < ActiveRecord::Base    has\_one :account, -> { where "confirmed = 1" }  end |

##### includes

Метод includes позволяет определить связи второго порядка, которые должны быть лениво загружены при использовании этой связи. Например, рассмотрим жти модели:

|  |
| --- |
| class Supplier < ActiveRecord::Base    has\_one :account  end    class Account < ActiveRecord::Base    belongs\_to :supplier    belongs\_to :representative  end    class Representative < ActiveRecord::Base    has\_many :accounts  end |

Если вы часто получаете representatives непосредственно из suppliers (@supplier.account.representative), то можно улучшить эффективность кода, включив representatives в связь между suppliers и accounts:

|  |
| --- |
| class Supplier < ActiveRecord::Base    has\_one :account, -> { includes :representative }  end    class Account < ActiveRecord::Base    belongs\_to :supplier    belongs\_to :representative  end    class Representative < ActiveRecord::Base    has\_many :accounts  end |

##### readonly

При использовании readonly, связанный объект будет только для чтения при получении через связь.

##### select

Метод select позволяет переопределить SQL выражение SELECT, используемое для получения данных о связанном объекте. По умолчанию Rails получает все столбцы.

#### Существуют ли связанные объекты?

Можно увидеть, существует ли какой-либо связанный объект, при использовании метода association.nil?:

|  |
| --- |
| if @supplier.account.nil?    @msg = "No account found for this supplier"  end |

#### Когда сохраняются объекты?

Когда вы назначаете объект связью has\_one, этот объект автоматически сохраняется (для того, чтобы обновить его внешний ключ). Кроме того, любой заменяемый объект также автоматически сохраняется, поскольку его внешний ключ также изменяется.

Если одно из этих сохранений проваливается из-за ошибок валидации, тогда выражение назначения возвращает false, и само назначение отменяется.

Если родительский объект (который объявляет связь has\_one) является несохраненным (то есть new\_record? возвращает true), тогда дочерние объекты не сохраняются. Они сохранятся автоматически, когда сохранится родительский объект.

Если вы хотите назначить объект связью has\_one без сохранения объекта, используйте метод association.build.

### Подробная информация по связи has\_many

Связь has\_many создает отношение один-ко-многим с другой моделью. В терминах базы данных эта связь говорит, что другой класс будет иметь внешний ключ, относящийся к экземплярам этого класса.

#### Добавляемые методы

Когда объявляете связь has\_many, объявляющий класс автоматически получает 13 методов, относящихся к связи: \* collection(force\_reload = false) \* collection<<(object, ...) \* collection.delete(object, ...) \* collection.destroy(object, ...) \* collection=objects \* collection\_singular\_ids \* collection\_singular\_ids=ids \* collection.clear \* collection.empty? \* collection.size \* collection.find(...) \* collection.where(...) \* collection.exists?(...) \* collection.build(attributes = {}, ...) \* collection.create(attributes = {}) \* collection.create!(attributes = {})

Во всех этих методах collection заменяется символом, переданным как первый аргумент в has\_many, и collection\_singular заменяется версией в единственном числе этого символа. Например, имеем объявление:

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_many :orders  end |

Каждый экземпляр модели customer будет иметь эти методы:

|  |
| --- |
| orders(force\_reload = false)  orders<<(object, ...)  orders.delete(object, ...)  orders.destroy(object, ...)  orders=objects  order\_ids  order\_ids=ids  orders.clear  orders.empty?  orders.size  orders.find(...)  orders.where(...)  orders.exists?(...)  orders.build(attributes = {}, ...)  orders.create(attributes = {})  orders.create!(attributes = {}) |

##### collection(force\_reload = false)

Метод collection возвращает массив всех связанных объектов. Если нет связанных объектов, он возвращает пустой массив.

|  |
| --- |
| @orders = @customer.orders |

##### collection<<(object, ...)

Метод collection<< добавляет один или более объектов в коллекцию, устанавливая их внешние ключи равными первичному ключу вызывающей модели.

|  |
| --- |
| @customer.orders << @order1 |

##### collection.delete(object, ...)

Метод collection.delete убирает один или более объектов из коллекции, установив их внешние ключи в NULL.

|  |
| --- |
| @customer.orders.delete(@order1) |

Объекты будут в дополнение уничтожены, если связаны с dependent: :destroy, и удалены, если они связаны с dependent: :delete\_all.

##### collection.destroy(object, ...)

Метод collection.destroy убирает один или более объектов из коллекции, выполняя destroy для каждого объекта.

|  |
| --- |
| @customer.orders.destroy(@order1) |

Объекты будут всегда удаляться из базы данных, игнорируя опцию :dependent.

##### collection=objects

Метод collection= делает коллекцию содержащей только представленные объекты, добавляя и удаляя по мере необходимости.

##### collection\_singular\_ids

Метод collection\_singular\_ids возвращает массив id объектов в коллекции.

|  |
| --- |
| @order\_ids = @customer.order\_ids |

##### collection\_singular\_ids=ids

Метод collection\_singular\_ids= делает коллекцию содержащей только объекты, идентифицированные представленными значениями первичного ключа, добавляя и удаляя по мере необходимости.

##### collection.clear

Метод collection.clear убирает каждый объект из коллекции. Это уничтожает связанные объекты, если они связаны с dependent: :destroy, удаляет их непосредственно из базы данных, если dependent: :delete\_all, и в противном случае устанавливает их внешние ключи в NULL.

##### collection.empty?

Метод collection.empty? возвращает true, если коллекция не содержит каких-либо связанных объектов.

|  |
| --- |
| <% if @customer.orders.empty? %>    No Orders Found  <% end %> |

##### collection.size

Метод collection.size возвращает количество объектов в коллекции.

|  |
| --- |
| @order\_count = @customer.orders.size |

##### collection.find(...)

Метод collection.find ищет объекты в коллекции. Он использует тот же синтаксис и опции, что и ActiveRecord::Base.find.

|  |
| --- |
| @open\_orders = @customer.orders.find(1) |

##### collection.where(...)

Метод collection.where ищет объекты в коллекции, основываясь на переданных условиях, но объекты загружаются лениво, что означает, что база данных запрашивается только когда происходит доступ к объекту(-там).

|  |
| --- |
| @open\_orders = @customer.orders.where(open: true) # Пока нет запроса  @open\_order = @open\_orders.first # Теперь база данных будет запрошена |

##### collection.exists?(...)

Метод collection.exists? проверяет, существует ли в коллекции объект, отвечающий представленным условиям. Он использует тот же синтаксис и опции, что и ActiveRecord::Base.exists?.

##### collection.build(attributes = {}, ...)

Метод collection.build возвращает один или более объектов связанного типа. Эти объекты будут экземплярами с переданными атрибутами, будет создана ссылка через их внешние ключи, но связанные объекты не будут пока сохранены.

|  |
| --- |
| @order = @customer.orders.build(order\_date: Time.now,                                  order\_number: "A12345") |

##### collection.create(attributes = {})

Метод collection.create возвращает новый объект связанного типа. Этот объект будет экземпляром с переданными атрибутами, будет создана ссылка через его внешний ключ, и, если он пройдет валидации, определенные в связанной модели, связанный объект будет сохранен

|  |
| --- |
| @order = @customer.orders.create(order\_date: Time.now,                                   order\_number: "A12345") |

##### collection.create!(attributes = {})

Работает так же, как вышеприведенный collection.create, но вызывает ActiveRecord::RecordInvalid, если запись невалидна.

#### Опции для has\_many

Хотя Rails использует разумные значения по умолчанию, работающие во многих ситуациях, бывают случаи, когда хочется изменить поведение связи has\_many. Такая настройка легко выполнима с помощью передачи опций при создании связи. Например, эта связь использует две такие опции:

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_many :orders, dependent: :delete\_all, validate: :false  end |

Связь has\_many поддерживает эти опции:

* :as
* :autosave
* :class\_name
* :dependent
* :foreign\_key
* :inverse\_of
* :primary\_key
* :source
* :source\_type
* :through
* :validate

##### :as

Установка опции :as показывает, что это полиморфная связь. Полиморфные связи подробно рассматривались [ранее](http://rusrails.ru/active-record-associations#polymorphic-associations).

##### :autosave

Если установить опцию :autosave в true, Rails сохранит любые загруженные члены и уничтожит члены, помеченные для уничтожения, всякий раз, когда вы сохраняете родительский объект.

##### :class\_name

Если имя другой модели не может быть произведено из имени связи, можете использовать опцию :class\_name для предоставления имени модели. Например, если покупатель имеет много заказов, но фактическое имя модели, содержащей заказы это Transaction, можете установить это следующим образом:

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_many :orders, class\_name: "Transaction"  end |

##### :dependent

Управляет тем, что произойдет со связанными объектами, когда его владелец будет уничтожен:

* :destroy приведет к тому, что связанные объекты также будут уничтожены
* :delete приведет к тому, что связанные объекты будут удалены из базы данных напрямую (таким образом не будут выполнены колбэки)
* :nullify приведет к тому, что внешние ключи будет установлен NULL. Колбэки не запускаются.
* :restrict\_with\_exception приведет к вызову исключения, если есть какой-нибудь связанный объект
* :restrict\_with\_error приведет к ошибке, добавляемой к владельцу, если есть какой-нибудь связанный объект

Эта опция игнорируется при использовании на связи опции :through.

##### :foreign\_key

По соглашению Rails предполагает, что столбец, используемый для хранения внешнего ключа в этой модели, имеет имя модели с добавленным суффиксом \_id. Опция :foreign\_key позволяет установить имя внешнего ключа явно:

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_many :orders, foreign\_key: "cust\_id"  end |

В любом случае, Rails не создаст столбцы внешнего ключа за вас. Вам необходимо явно определить их в своих миграциях.

##### :inverse\_of

Опция :inverse\_of определяет имя связи belongs\_to, являющейся обратной для этой связи. Не работает в комбинации с опциями :through или :as.

|  |
| --- |
| class Supplier < ActiveRecord::Base    has\_many :orders, inverse\_of: :customer  end    class Account < ActiveRecord::Base    belongs\_to :customer, inverse\_of: :orders  end |

##### :primary\_key

По соглашению, Rails предполагает, что столбец, используемый для хранения первичного ключа, это id. Вы можете переопределить это и явно определить первичный ключ с помощью опции :primary\_key.

Допустим, в таблице users есть id в качестве primary\_key, но также имеется столбец guid. А также имеется требование, что таблица todos должна содержать значение столбца guid, а не значение id. Это достигается следующим образом

|  |
| --- |
| class User < ActiveRecord::Base    has\_many :todos, primary\_key: :guid  end |

Теперь, если выполнить @user.todos.create, то в запись @todo значение user\_id будет таким же, как значение guid в @user.

##### :source

Опция :source oпределяет имя источника связи для связи has\_many :through. Эту опцию нужно использовать, только если имя источника связи не может быть автоматически выведено из имени связи.

##### :source\_type

Опция :source\_type определяет тип источника связи для связи has\_many :through, который действует при полиморфной связи.

##### :through

Опция :through определяет соединительную модель, через которую выполняется запрос. Связи has\_many :through предоставляют способ осуществления отношений многие-ко-многим, как обсуждалось [ранее в этом руководстве](http://rusrails.ru/active-record-associations#the-has-many-through-association).

##### :validate

Если установите опцию :validate в false, тогда связанные объекты не будут проходить валидацию всякий раз, когда вы сохраняете этот объект. По умолчанию она равна true: связанные объекты проходят валидацию, когда этот объект сохраняется.

#### Скоупы для has\_many

Иногда хочется настроить запрос, используемый has\_many. Такая настройка может быть достигнута с помощью блока скоупа. Например:

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_many :orders, -> { where processed: true }  end |

Внутри блока скоупа можно использовать любые стандартные [методы запросов](http://rusrails.ru/active-record-query-interface). Далее обсудим следующие из них:

* where
* extending
* group
* includes
* limit
* offset
* order
* readonly
* select
* uniq

##### where

Метод where позволяет определить условия, которым должен отвечать связанный объект.

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_many :confirmed\_orders, -> { where "confirmed = 1" },      class\_name: "Order"  end |

Также можно задать условия хэшем:

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_many :confirmed\_orders, -> { where confirmed: true },                                class\_name: "Order"  end |

При использовании опции where хэшем, при создание записи через эту связь будет автоматически применен скоуп с использованием хэша. В этом случае при использовании @customer.confirmed\_orders.create или @customer.confirmed\_orders.build будут созданы заказы, в которых столбец confirmed будет иметь значение true.

##### extending

Метод extending определяет именнованый модуль для расширения прокси связи. Расширения связей подробно обсуждаются [позже в этом руководстве](http://rusrails.ru/active-record-associations#association-callbacks-and-extensions).

##### group

Метод group доставляет имя атрибута, по которому группируется результирующий набор, используя выражение GROUP BY в поисковом SQL.

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_many :line\_items, -> { group 'orders.id' },                          through: :orders  end |

##### includes

Можете использовать метод include для определения связей второго порядка, которые должны быть нетерпеливо загружены, когда эта связь используется. Например, рассмотрим эти модели:

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_many :orders  end    class Order < ActiveRecord::Base    belongs\_to :customer    has\_many :line\_items  end    class LineItem < ActiveRecord::Base    belongs\_to :order  end |

Если вы часто получаете позиции прямо из покупателей (@customer.orders.line\_items), тогда можете сделать свой код более эффективным, включив позиции в связь от покупателей к заказам:

|  |
| --- |
| class Customer < ActiveRecord::Base    as\_many :orders, -> { includes :line\_items }  end    class Order < ActiveRecord::Base    belongs\_to :customer    has\_many :line\_items  end    class LineItem < ActiveRecord::Base    belongs\_to :order  end |

##### limit

Метод limit позволяет ограничить общее количество объектов, которые будут выбраны через связь.

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_many :recent\_orders,      -> { order('order\_date desc').limit(100) },      class\_name: "Order"  end |

##### offset

Метод offset позволяет определить начальное смещение для выбора объектов через связь. Например, -> { offset(11) } пропустит первые 11 записей.

##### order

Метод order предписывает порядок, в котором связанные объекты будут получены (в синтаксисе SQL, используемом в условии ORDER BY).

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_many :orders, -> { order "date\_confirmed DESC" }  end |

##### readonly

При использовании метода :readonly, связанные объекты будут доступны только для чтения, когда получены посредством связи.

##### select

Метод select позволяет переопределить SQL условие SELECT, которое используется для получения данных о связанном объекте. По умолчанию Rails получает все столбцы.

Если укажете свой собственный select, не забудьте включить столбцы первичного ключа и внешнего ключа в связанной модели. Если так не сделать, Rails выдаст ошибку.

##### distinct

Используйте метод distinct, чтобы убирать дубликаты из коллекции. Это полезно в сочетании с опцией :through.

|  |
| --- |
| class Person < ActiveRecord::Base    has\_many :readings    has\_many :posts, through: :readings  end    person = Person.create(name: 'John')  post   = Post.create(name: 'a1')  person.posts << post  person.posts << post  person.posts.inspect # => [#<Post id: 5, name: "a1">, #<Post id: 5, name: "a1">]  Reading.all.inspect  # => [#<Reading id: 12, person\_id: 5, post\_id: 5>, #<Reading id: 13, person\_id: 5, post\_id: 5>] |

В вышеописанной задаче два reading, и person.posts выявляет их оба, даже хотя эти записи указывают на один и тот же post.

Давайте установим :distinct:

|  |
| --- |
| class Person    has\_many :readings    has\_many :posts, -> { distinct }, through: :readings  end    person = Person.create(name: 'Honda')  post   = Post.create(name: 'a1')  person.posts << post  person.posts << post  person.posts.inspect # => [#<Post id: 7, name: "a1">]  Reading.all.inspect  # => [#<Reading id: 16, person\_id: 7, post\_id: 7>, #<Reading id: 17, person\_id: 7, post\_id: 7>] |

В вышеописанной задаче все еще два reading. Однако person.posts показывает только один post, поскольку коллекция загружает только уникальные записи.

Если вы хотите быть уверенными, что после вставки все записи сохраненной связи различны (и, таким образом, убедиться, что при росмотре связи никогда не будет дублирующихся записей), следует добавить уникальный индекс для самой таблицы. Например, если таблица называется person\_posts, и вы хотите убедиться, что все публикации уникальны, следует добавить в миграции:

|  |
| --- |
| add\_index :person\_posts, :post, unique: true |

Отметьте, что проверка уникальности при использовании чего-то, наподобие include?, это субъект гонки условий. Не пытайтесь использовать include? для соблюдения уникальности в связи. Используя вышеприведенный пример с публикацией, нижеследующий код вызовет гонку, поскольку несколько пользователей могут использовать его одновременно:

|  |
| --- |
| person.posts << post unless person.posts.include?(post) |

#### Когда сохраняются объекты?

Когда вы назначаете объект связью has\_many, этот объект автоматически сохраняется (для того, чтобы обновить его внешний ключ). Если назначаете несколько объектов в одном выражении, они все будут сохранены.

Если одно из этих сохранений проваливается из-за ошибок валидации, тогда выражение назначения возвращает false, и само назначение отменяется.

Если родительский объект (который объявляет связь has\_many) является несохраненным (то есть new\_record? возвращает true) тогда дочерние объекты не сохраняются при добавлении. Все несохраненные члены связи сохранятся автоматически, когда сохранится родительский объект.

Если вы хотите назначить объект связью has\_many без сохранения объекта, используйте метод collection.build.

### Подробная информация по связи has\_and\_belongs\_to\_many

Связь has\_and\_belongs\_to\_many создает отношение один-ко-многим с другой моделью. В терминах базы данных это связывает два класса через промежуточную соединительную таблицу, которая включает внешние ключи, относящиеся к каждому классу.

#### Добавляемые методы

Когда объявляете связь has\_and\_belongs\_to\_many, объявляющий класс автоматически получает 14 методов, относящихся к связи:

* collection(force\_reload = false)
* collection<<(object, ...)
* collection.delete(object, ...)
* collection.destroy(object, ...)
* collection=objects
* collection\_singular\_ids
* collection\_singular\_ids=ids
* collection.clear
* collection.empty?
* collection.size
* collection.find(...)
* collection.where(...)
* collection.exists?(...)
* collection.build(attributes = {})
* collection.create(attributes = {})
* collection.create!(attributes = {})

Во всех этих методах collection заменяется символом, переданным как первый аргумент в has\_and\_belongs\_to\_many, а collection\_singular заменяется версией в единственном числе этого символа. Например, имеем объявление:

|  |
| --- |
| class Part < ActiveRecord::Base    has\_and\_belongs\_to\_many :assemblies  end |

Каждый экземпляр модели part будет иметь эти методы:

|  |
| --- |
| assemblies(force\_reload = false)  assemblies<<(object, ...)  assemblies.delete(object, ...)  assemblies.destroy(object, ...)  assemblies=objects  assembly\_ids  assembly\_ids=ids  assemblies.clear  assemblies.empty?  assemblies.size  assemblies.find(...)  assemblies.where(...)  assemblies.exists?(...)  assemblies.build(attributes = {}, ...)  assemblies.create(attributes = {})  assemblies.create!(attributes = {}) |

##### Дополнительные методы столбцов

Если соединительная таблица для связи has\_and\_belongs\_to\_many имеет дополнительные столбцы, кроме двух внешних ключей, эти столбцы будут добавлены как атрибуты к записям, получаемым посредством связи. Записи, возвращаемые с дополнительными атрибутами, будут всегда только для чтения, поскольку Rails не может сохранить значения этих атрибутов.

Использование дополнительных атрибутов в соединительной таблице в связи has\_and\_belongs\_to\_many устарело. Если требуется этот тип сложного поведения таблицы, соединяющей две модели в отношениях многие-ко-многим, следует использовать связь has\_many :through вместо has\_and\_belongs\_to\_many.

##### collection(force\_reload = false)

Метод collection возвращает массив всех связанных объектов. Если нет связанных объектов, он возвращает пустой массив.

|  |
| --- |
| @assemblies = @part.assemblies |

##### collection<<(object, ...)

Метод collection&lt;&lt; добавляет один или более объектов в коллекцию, создавая записи в соединительной таблице.

|  |
| --- |
| @part.assemblies << @assembly1 |

Этот метод - просто синоним к collection.concat и collection.push.

##### collection.delete(object, ...)

Метод collection.delete убирает один или более объектов из коллекции, удаляя записи в соединительной таблице. Это не уничтожает объекты.

|  |
| --- |
| @part.assemblies.delete(@assembly1) |

Это не запустит колбэки на соединительных записях.

###### collection.destroy(object, ...)

Метод collection.destroy убирает один или более объектов из коллекции. запуская destroy на каждой записи в соединительной таблице, включая запуск колбэков. Это не уничтожает объекты.

|  |
| --- |
| @part.assemblies.destroy(@assembly1) |

##### collection=objects

Метод collection= делает коллекцию содержащей только представленные объекты, добавляя и удаляя по мере необходимости.

##### collection\_singular\_ids

Метод collection\_singular\_ids возвращает массив id объектов в коллекции.

|  |
| --- |
| @assembly\_ids = @part.assembly\_ids |

##### collection\_singular\_ids=ids

Метод collection\_singular\_ids= делает коллекцию содержащей только объекты, идентифицированные представленными значениями первичного ключа, добавляя и удаляя по мере необходимости.

##### collection.clear

Метод collection.clear убирает каждый объект из коллекции, удаляя строки из соединительной таблицы. Это не уничтожает связанные объекты.

##### collection.empty?

Метод collection.empty? возвращает true, если коллекция не содержит каких-либо связанных объектов.

|  |
| --- |
| <% if @part.assemblies.empty? %>    This part is not used in any assemblies  <% end %> |

##### collection.size

Метод collection.size возвращает количество объектов в коллекции.

|  |
| --- |
| @assembly\_count = @part.assemblies.size |

##### collection.find(...)

Метод collection.find ищет объекты в коллекции. Он использует тот же синтаксис и опции, что и ActiveRecord::Base.find. Он также добавляет дополнительное условие, что объект должен быть в коллекции.

|  |
| --- |
| @assembly = @part.assemblies.find(1) |

##### collection.where(...)

Метод collection.where ищет объекты в коллекции, основываясь на переданных условиях, но объекты загружаются лениво, что означает, что база данных запрашивается только когда происходит доступ к объекту(-там). Он также добавляет дополнительное условие, что объект должен быть в коллекции.

|  |
| --- |
| @new\_assemblies = @part.assemblies.where("created\_at > ?", 2.days.ago) |

##### collection.exists?(...)

Метод collection.exists? проверяет, существует ли в коллекции объект, отвечающий представленным условиям. Он использует тот же синтаксис и опции, что и ActiveRecord::Base.exists?.

##### collection.build(attributes = {})

Метод collection.build возвращает один или более объектов связанного типа. Эти объекты будут экземплярами с переданными атрибутами, и будет создана связь через соединительную таблицу, но связанный объект не будет пока сохранен.

|  |
| --- |
| @assembly = @part.assemblies.build({assembly\_name: "Transmission housing"}) |

##### collection.create(attributes = {})

Метод collection.create возвращает один или более объектов связанного типа. Эти объекты будут экземплярами с переданными атрибутами, будет создана связь через соединительную таблицу, и, если он пройдет валидации, определенные в связанной модели, связанный объект будет сохранен.

|  |
| --- |
| @assembly = @part.assemblies.create({assembly\_name: "Transmission housing"}) |

##### collection.create!(attributes = {})

Работает так же, как вышеприведенный collection.create, но вызывает ActiveRecord::RecordInvalid, если запись невалидна.

#### Опции для has\_and\_belongs\_to\_many

Хотя Rails использует разумные значения по умолчанию, работающие во многих ситуациях, бывают случаи, когда хочется изменить поведение связи has\_and\_belongs\_to\_many. Такая настройка легко выполнима с помощью передачи опции при создании связи. Например, эта связь использует две такие опции:

|  |
| --- |
| class Parts < ActiveRecord::Base    has\_and\_belongs\_to\_many :assemblies, autosave: true,                                         readonly: true  end |

Связь has\_and\_belongs\_to\_many поддерживает эти опции:

* :association\_foreign\_key
* :autosave
* :class\_name
* :foreign\_key
* :join\_table
* :validate
* :readonly

##### :association\_foreign\_key

По соглашению Rails предполагает, что столбец в соединительной таблице, используемый для хранения внешнего ключа, указываемого на другую модель, является именем этой модели с добавленным суффиксом \_id. Опция :association\_foreign\_key позволяет установить имя внешнего ключа явно:

Опции :foreign\_key и :association\_foreign\_key полезны при настройке самоприсоединения многие-ко-многим. Например:

|  |
| --- |
| class User < ActiveRecord::Base    has\_and\_belongs\_to\_many :friends,        class\_name: "User",        foreign\_key: "this\_user\_id",        association\_foreign\_key: "other\_user\_id"  end |

##### :autosave

Если установить опцию :autosave в true, Rails сохранит любые загруженные члены и уничтожит члены, помеченные для уничтожения, всякий раз, когда Вы сохраните родительский объектt.

##### :class\_name

Если имя другой модели не может быть произведено из имени связи, можете использовать опцию :class\_name для предоставления имени модели. Например, если часть имеет много узлов, но фактическое имя модели, содержащей узлы - это Gadget, можете установить это следующим образом:

|  |
| --- |
| class Parts < ActiveRecord::Base    has\_and\_belongs\_to\_many :assemblies, class\_name: "Gadget"  end |

##### :foreign\_key

По соглашению Rails предполагает, что столбец в соединительной таблице, используемый для хранения внешнего ключа, указываемого на эту модель, имеет имя модели с добавленным суффиксом \_id. Опция :foreign\_key позволяет установить имя внешнего ключа явно:

|  |
| --- |
| class User < ActiveRecord::Base    has\_and\_belongs\_to\_many :friends,        class\_name: "User",        foreign\_key: "this\_user\_id",        association\_foreign\_key: "other\_user\_id"  end |

##### :join\_table

Если имя соединительной таблицы по умолчанию, основанное на алфавитном порядке, - это не то, что вам нужно, используйте опцию :join\_table, чтобы переопределить его.

##### :validate

Если установите опцию :validate в false, тогда связанные объекты не будут проходить валидацию всякий раз, когда вы сохраняете этот объект. По умолчанию она равна true: связанные объекты проходят валидацию, когда этот объект сохраняется.

#### Скоупы для has\_and\_belongs\_to\_many

Иногда хочется настроить запрос, используемый has\_many. Такая настройка может быть достигнута с помощью блока скоупа. Например:

|  |
| --- |
| class Parts < ActiveRecord::Base    has\_and\_belongs\_to\_many :assemblies, -> { where active: true }  end |

Внутри блока скоупа можно использовать любые стандартные [методы запросов](http://rusrails.ru/active-record-query-interface). Далее обсудим следующие из них:

* where
* extending
* group
* includes
* limit
* offset
* order
* readonly
* select
* uniq

##### where

Метод where позволяет определить условия, которым должен отвечать связанный объект.

|  |
| --- |
| class Parts < ActiveRecord::Base    has\_and\_belongs\_to\_many :assemblies,      -> { where "factory = 'Seattle'" }  end |

Также можно задать условия хэшем:

|  |
| --- |
| class Parts < ActiveRecord::Base    has\_and\_belongs\_to\_many :assemblies,      -> { where factory: 'Seattle' }  end |

При использовании опции where хэшем, при создание записи через эту связь будет автоматически применен скоуп с использованием хэша. В этом случае при использовании @parts.assemblies.create или @parts.assemblies.build будут созданы заказы, в которых столбец factory будет иметь значение Seattle.

##### extending

Метод extending определяет именнованый модуль для расширения прокси связи. Расширения связей подробно обсуждаются [позже в этом руководстве](http://rusrails.ru/active-record-associations#association-callbacks-and-extensions).

##### group

Метод group доставляет имя атрибута, по которому группируется результирующий набор, используя выражение GROUP BY в поисковом SQL.

|  |
| --- |
| class Parts < ActiveRecord::Base    has\_and\_belongs\_to\_many :assemblies, -> { group "factory" }  end |

##### includes

Можете использовать метод include для определения связей второго порядка, которые должны быть нетерпеливо загружены, когда эта связь используется.

##### limit

Метод limit позволяет ограничить общее количество объектов, которые будут выбраны через связь.

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_and\_belongs\_to\_many :assemblies,      -> { order("created\_at DESC").limit(50) }  end |

##### offset

Метод offset позволяет определить начальное смещение для выбора объектов через связь. Например, -> { offset(11) } пропустит первые 11 записей.

##### order

Метод order предписывает порядок, в котором связанные объекты будут получены (в синтаксисе SQL, используемом в условии ORDER BY).

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_and\_belongs\_to\_many :assemblies,      -> { order "assembly\_name ASC" }  end |

##### readonly

При использовании метода :readonly, связанные объекты будут доступны только для чтения, когда получены посредством связи.

##### select

Метод select позволяет переопределить SQL условие SELECT, которое используется для получения данных о связанном объекте. По умолчанию Rails получает все столбцы.

##### uniq

Используйте метод uniq, чтобы убирать дубликаты из коллекции. Это полезно в сочетании с опцией :through.

#### Когда сохраняются объекты?

Когда вы назначаете объект связью has\_and\_belongs\_to\_many этот объект автоматически сохраняется (в порядке обновления соединительной таблицы). Если назначаете несколько объектов в одном выражении, они все будут сохранены.

Если одно из этих сохранений проваливается из-за ошибок валидации, тогда выражение назначения возвращает false, a само назначение отменяется.

Если родительский объект (который объявляет связь has\_and\_belongs\_to\_many) является несохраненным (то есть new\_record? возвращает true) тогда дочерние объекты не сохраняются при добавлении. Все несохраненные члены связи сохранятся автоматически, когда сохранится родительский объект.

Если вы хотите назначить объект связью has\_and\_belongs\_to\_many без сохранения объекта, используйте метод collection.build.

### Подробная информация по колбэкам и расширениям связи

#### Колбэки связи

Обычно колбэки прицепляются к жизненному циклу объектов Active Record, позволяя Вам работать с этими объектами в различных точках. Например, можете использовать колбэк :before\_save, чтобы вызвать что-то перед тем, как объект будет сохранен.

Колбэки связи похожи на обычные колбэки, но они включаются событиями в жизненном цикле коллекции. Доступны четыре колбэка связи:

* before\_add
* after\_add
* before\_remove
* after\_remove

Колбэки связи объявляются с помощью добавления опций в объявление связи. Например:

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_many :orders, before\_add: :check\_credit\_limit      def check\_credit\_limit(order)      ...    end  end |

Rails передает добавляемый или удаляемый объект в колбэк.

Можете помещать колбэки в очередь на отдельное событие, передав их как массив:

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_many :orders,      before\_add: [:check\_credit\_limit, :calculate\_shipping\_charges]      def check\_credit\_limit(order)      ...    end      def calculate\_shipping\_charges(order)      ...    end  end |

Если колбэк before\_add вызывает исключение, объект не будет добавлен в коллекцию. Подобным образом, если колбэк before\_remove вызывает исключение, объект не убирается из коллекции.

#### Расширения связи

Вы не ограничены функциональностью, которую Rails автоматически встраивает в выданные по связи объекты. Можете расширять эти объекты через анонимные модули, добавления новых методов поиска, создания и иных методов. Например:

|  |
| --- |
| class Customer < ActiveRecord::Base    has\_many :orders do      def find\_by\_order\_prefix(order\_number)        find\_by(region\_id: order\_number[0..2])      end    end  end |

Если имеется расширение, которое должно быть распространено на несколько связей, можете использовать именнованный модуль расширения. Например:

|  |
| --- |
| module FindRecentExtension    def find\_recent      where("created\_at > ?", 5.days.ago)    end  end    class Customer < ActiveRecord::Base    has\_many :orders, -> { extending FindRecentExtension }  end    class Supplier < ActiveRecord::Base    has\_many :deliveries, -> { extending FindRecentExtension }  end |

Расширения могут ссылаться на внутренние методы выданных по связи объектов, используя следующие три атрибута аксессора proxy\_association:

* proxy\_association.owner возвращает объект, в котором объявлена связь.
* proxy\_association.reflection возвращает объект reflection, описывающий связь.
* proxy\_association.target возвращает связанный объект для belongs\_to или has\_one, или коллекцию связанных объектов для has\_many или has\_and\_belongs\_to\_many.