module

ActiveRecord::Associations::ClassMethods

Ruby on Rails latest stable (v4.2.1) - 4 notes

- 1.0.0 (0)
- 1.1.1 (34)
- 1.1.6 (0)
- 1.2.0 (11)
- 1.2.6 (2)
- 2.0.0 (33)
- 2.0.3 (4)
- 2.1.0 (5)
- 2.2.1 (3)
- 2.3.2 (1)
- 2.3.8 (7)
- 3.0.0 (15)
- 3.0.5 (0)
- 3.0.9 (-8)
- 3.1.0 (38)
- 3.2.1 (3)
- 3.2.3 (0)
- 3.2.8 (0)
- 3.2.13 (0)
- 4.0.2 (4)
- 4.1.8 (8)
- 4.2.1 (2)
- What's this?

Related

- Instance methods (61)
- add_association_callbacks (<= v3.0.9)
- add_counter_cache_callbacks (<= v3.0.9)
- add_deprecated_api_for_has_... (<= v1.2.6)
- add_limited_ids_condition! (<= v2.3.8)
- add_multiple_associated_sav... (<= v2.2.1)
- add_multiple_associated_val... (<= v2.2.1)
- add_single_associated_save_... (<= v2.1.0)
- add_single_associated_valid... (<= v2.2.1)
- add_sti_conditions! (<= v1.0.0)
- add_touch_callbacks (<= v3.0.9)

- associate_identification (<= v1.0.0)
- association_accessor_methods (<= v3.0.9)
- association_constructor_method (<= v3.0.9)
- association_join (<= v1.0.0)
- belongs_to
- collection_accessor_methods (<= v3.0.9)
- collection_reader_method (<= v3.0.9)
- column_aliases (<= v2.3.8)
- conditions_tables (<= v2.3.8)
- condition_word (<= v2.3.8)
- configure_after_destroy_met... (<= v3.0.9)
- configure_dependency_for_be... (<= v3.0.9)
- configure_dependency_for_ha... (<= v3.0.9)
- configure_dependency_for_ha... (<= v3.0.9)
- construct_counter_sql_with_... (<= v1.1.6)
- construct_finder_sql_for_as... (<= v2.3.8)
- construct_finder_sql_with_i... (<= v2.3.8)
- count_with_associations (<= v1.1.6)
- create_belongs_to_reflection (<= v3.0.9)
- create_extension_module (<= v1.2.6)
- create_extension_modules (<= v3.0.9)
- create_has_and_belongs_to_m... (<= v3.0.9)
- create_has_many_reflection (<= v3.0.9)
- create_has_one_reflection (<= v3.0.9)
- create_has_one_through_refl... (<= v3.0.9)
- delete_all_has_many_depende... (<= v3.0.9)
- extract_record (<= v1.0.0)
- find_with_associations (<= v2.3.8)
- generate_primary_key_table (<= v1.0.0)
- generate_schema_abbreviations (<= v1.0.0)
- guard_against_missing_refle... (<= v1.0.0)
- guard_against_unlimitable_r... (<= v2.3.8)
- has_and_belongs_to_many
- has_many
- has_one
- include_eager_conditions? (<= v2.3.8)
- include_eager_order? (<= v2.3.8)
- include_eager_select? (<= v2.3.8)
- joined_tables (<= v2.3.8)
- join_table_name (<= v3.0.9)
- nullify_has_many_dependencies (<= v3.0.9)
- order_tables (<= v2.3.8)
- references_eager_loaded_tab... (<= v2.3.8)
- reflect_on_included_associa... (<= v2.3.8)
- require_association_class (<= v1.1.6)
- select_all_rows (<= v2.3.8)
- select_limited_ids_list (<= v2.3.8)
- selects_tables (<= v2.3.8)
- tables_in_hash (<= v2.3.8)
- tables_in_string (<= v2.3.8)
- using_limitable_reflections? (<= v2.3.8)
- Included modules
- Module
- Namespace children
- InnerJoinDependency (<= v2.3.8)
- JoinDependency (<= v3.0.9)

= private
= protected

Associations are a set of macro-like class methods for tying objects together through foreign keys. They express relationships like “Project has one Project Manager” or “Project belongs to a Portfolio”. Each macro adds a number of methods to the class which are specialized according to the collection or association symbol and the options hash. It works much the same way as Ruby’s own `attr*` methods.

```
class Project < ActiveRecord::Base
  belongs_to              :portfolio
  has_one                 :project_manager
  has_many                :milestones
```

```
  has_and_belongs_to_many :categories
end
```

The project class now has the following methods (and more) to ease the traversal and manipulation of its relationships:

- `Project#portfolio, Project#portfolio=(portfolio), Project#portfolio.nil?`
- `Project#project_manager, Project#project_manager=(project_manager), Project#project_manager.nil?,`
- `Project#milestones.empty?, Project#milestones.size, Project#milestones, Project#milestones<<(milestone), Project#milestones.delete(milestone), Project#milestones.destroy(milestone), Project#milestones.find(milestone_id), Project#milestones.build, Project#milestones.create`
- `Project#categories.empty?, Project#categories.size, Project#categories, Project#categories<<(category1), Project#categories.delete(category1), Project#categories.destroy(category1)`

A word of warning

Don't create associations that have the same name as instance methods of `ActiveRecord::Base`. Since the association adds a method with that name to its model, it will override the inherited method and break things. For instance, `attributes` and `connection` would be bad choices for association names.

Auto-generated methods

See also Instance Public methods below for more details.

Singular associations (one-to-one)

generated methods	belongs_to	belongs_to :polymorphic	has_one
other(force_reload=false)	X	X	X
other=(other)	X	X	X
build_other(attributes={})	X		X
create_other(attributes={})	X		X
create_other!(attributes={})	X		X

Collection associations (one-to-many / many-to-many)

generated methods	habtm	has_many	has_many :through
others(force_reload=false)	X	X	X
others=(other,other,...)	X	X	X
other_ids	X	X	X
other_ids=(id,id,...)	X	X	X
others<<	X	X	X
others.push	X	X	X
others.concat	X	X	X
others.build(attributes={})	X	X	X
others.create(attributes={})	X	X	X
others.create!(attributes={})	X	X	X
others.size	X	X	X
others.length	X	X	X
others.count	X	X	X
others.sum(*args)	X	X	X
others.empty?	X	X	X
others.clear	X	X	X
others.delete(other,other,...)	X	X	X
others.delete_all	X	X	X
others.destroy(other,other,...)	X	X	X
others.destroy_all	X	X	X
others.find(*args)	X	X	X
others.exists?	X	X	X
others.distinct	X	X	X
others.uniq	X	X	X
others.reset	X	X	X

Overriding generated methods

Association methods are generated in a module that is included into the model class, which allows you to easily override with your own methods and call the original generated method with `super`. For example:

```
class Car < ActiveRecord::Base
  belongs_to :owner
  belongs_to :old_owner
  def owner=(new_owner)
    self.old_owner = self.owner
    super
  end
end
```

If your model class is `Project`, the module is named `Project::GeneratedFeatureMethods`. The GeneratedFeatureMethods module is included in the model class immediately after the (anonymous) generated attributes methods module, meaning an association will override the methods for an attribute with the same name.

Cardinality and associations

Active Record associations can be used to describe one-to-one, one-to-many and many-to-many relationships between models. Each model uses an association to describe its role in the relation. The `belongs_to` association is always used in the model that has the foreign key.

One-to-one

Use `has_one` in the base, and `belongs_to` in the associated model.

```
class Employee < ActiveRecord::Base
  has_one :office
end
class Office < ActiveRecord::Base
  belongs_to :employee    # foreign key - employee_id
end
```

One-to-many

Use `has_many` in the base, and `belongs_to` in the associated model.

```
class Manager < ActiveRecord::Base
  has_many :employees
end
class Employee < ActiveRecord::Base
  belongs_to :manager     # foreign key - manager_id
end
```

Many-to-many

There are two ways to build a many-to-many relationship.

The first way uses a `has_many` association with the `:through` option and a join model, so there are two stages of associations.

```
class Assignment < ActiveRecord::Base
  belongs_to :programmer  # foreign key - programmer_id
  belongs_to :project     # foreign key - project_id
end
class Programmer < ActiveRecord::Base
  has_many :assignments
  has_many :projects, through: :assignments
end
class Project < ActiveRecord::Base
  has_many :assignments
  has_many :programmers, through: :assignments
end
```

For the second way, use `has_and_belongs_to_many` in both models. This requires a join table that has no corresponding model or primary key.

```
class Programmer < ActiveRecord::Base
  has_and_belongs_to_many :projects       # foreign keys in the join table
end
class Project < ActiveRecord::Base
  has_and_belongs_to_many :programmers    # foreign keys in the join table
end
```

Choosing which way to build a many-to-many relationship is not always simple. If you need to work with the relationship model as its own entity, use `has_many :through`. Use `has_and_belongs_to_many` when working with legacy schemas or when you never work directly with the relationship itself.

Is it a belongs_to or has_one association?

Both express a 1-1 relationship. The difference is mostly where to place the foreign key, which goes on the table for the class declaring the belongs_to relationship.

```
class User < ActiveRecord::Base
  # I reference an account.
  belongs_to :account
end

class Account < ActiveRecord::Base
  # One user references me.
  has_one :user
end
```

The tables for these classes could look something like:

```
CREATE TABLE users (
  id int(11) NOT NULL auto_increment,
  account_id int(11) default NULL,
  name varchar default NULL,
  PRIMARY KEY  (id)
)

CREATE TABLE accounts (
  id int(11) NOT NULL auto_increment,
  name varchar default NULL,
  PRIMARY KEY  (id)
)
```

Unsaved objects and associations

You can manipulate objects and associations before they are saved to the database, but there is some special behavior you should be aware of, mostly involving the saving of associated objects.

You can set the `:autosave` option on a `has_one`, `belongs_to`, `has_many`, or has_and_belongs_to_many association. Setting it to `true` will *always* save the members, whereas setting it to `false` will *never* save the members. More details about `:autosave` option is available at AutosaveAssociation.

One-to-one associations

- Assigning an object to a has_one association automatically saves that object and the object being replaced (if there is one), in order to update their foreign keys - except if the parent object is unsaved (`new_record? == true`).
- If either of these saves fail (due to one of the objects being invalid), an ActiveRecord::RecordNotSaved exception is raised and the assignment is cancelled.
- If you wish to assign an object to a has_one association without saving it, use the `build_association` method (documented below). The object being replaced will still be saved to update its foreign key.
- Assigning an object to a belongs_to association does not save the object, since the foreign key field belongs on the parent. It does not save the parent either.

Collections

- Adding an object to a collection (has_many or has_and_belongs_to_many) automatically saves that object, except if the parent object (the owner of the collection) is not yet stored in the database.
- If saving any of the objects being added to a collection (via `push` or similar) fails, then `push` returns `false`.
- If saving fails while replacing the collection (via `association=`), an ActiveRecord::RecordNotSaved exception is raised and the assignment is cancelled.
- You can add an object to a collection without automatically saving it by using the `collection.build` method (documented below).
- All unsaved (`new_record? == true`) members of the collection are automatically saved when the parent is saved.

Customizing the query

Associations are built from `Relations`, and you can use the `Relation` syntax to customize them. For example, to add a condition:

```
class Blog < ActiveRecord::Base
  has_many :published_posts, -> { where published: true }, class_name: 'Post'
end
```

Inside the `-> { ... }` block you can use all of the usual `Relation` methods.

Accessing the owner object

Sometimes it is useful to have access to the owner object when building the query. The owner is passed as a parameter to the block. For example, the following association would find all events that occur on the user's birthday:

```
class User < ActiveRecord::Base
  has_many :birthday_events, ->(user) { where starts_on: user.birthday }, class_name: 'Event'
end
```

Note: Joining, eager loading and preloading of these associations is not fully possible. These operations happen before instance creation and the scope will be called with a `nil` argument. This can lead to unexpected behavior and is deprecated.

Association callbacks

Similar to the normal callbacks that hook into the life cycle of an Active Record object, you can also define callbacks that get triggered when you add an object to or remove an object from an association collection.

```
class Project
  has_and_belongs_to_many :developers, after_add: :evaluate_velocity

  def evaluate_velocity(developer)
    ...
  end
end
```

It's possible to stack callbacks by passing them as an array. Example:

```
class Project
  has_and_belongs_to_many :developers,
                          after_add: [:evaluate_velocity, Proc.new { |p, d| p.shipping_date = Time.now}]
end
```

Possible callbacks are: `before_add`, `after_add`, `before_remove` and `after_remove`.

If any of the `before_add` callbacks throw an exception, the object will not be added to the collection.

Similarly, if any of the `before_remove` callbacks throw an exception, the object will not be removed from the collection.

Association extensions

The proxy objects that control the access to associations can be extended through anonymous modules. This is especially beneficial for adding new finders, creators, and other factory-type methods that are only used as part of this association.

```
class Account < ActiveRecord::Base
  has_many :people do
    def find_or_create_by_name(name)
      first_name, last_name = name.split(" ", 2)
      find_or_create_by(first_name: first_name, last_name: last_name)
    end
  end
end

person = Account.first.people.find_or_create_by_name("David Heinemeier Hansson")
person.first_name # => "David"
person.last_name  # => "Heinemeier Hansson"
```

If you need to share the same extensions between many associations, you can use a named extension module.

```
module FindOrCreateByNameExtension
  def find_or_create_by_name(name)
    first_name, last_name = name.split(" ", 2)
    find_or_create_by(first_name: first_name, last_name: last_name)
  end
end
```

```
class Account < ActiveRecord::Base
  has_many :people, -> { extending FindOrCreateByNameExtension }
end

class Company < ActiveRecord::Base
  has_many :people, -> { extending FindOrCreateByNameExtension }
end
```

Some extensions can only be made to work with knowledge of the association's internals. Extensions can access relevant state using the following methods (where `items` is the name of the association):

- `record.association(:items).owner` - Returns the object the association is part of.
- `record.association(:items).reflection` - Returns the reflection object that describes the association.
- `record.association(:items).target` - Returns the associated object for `belongs_to` and `has_one`, or the collection of associated objects for `has_many` and `has_and_belongs_to_many`.

However, inside the actual extension code, you will not have access to the `record` as above. In this case, you can access `proxy_association`. For example, `record.association(:items)` and `record.items.proxy_association` will return the same object, allowing you to make calls like `proxy_association.owner` inside association extensions.

Association Join Models

Has Many associations can be configured with the `:through` option to use an explicit join model to retrieve the data. This operates similarly to a `has_and_belongs_to_many` association. The advantage is that you're able to add validations, callbacks, and extra attributes on the join model. Consider the following schema:

```
class Author < ActiveRecord::Base
  has_many :authorships
  has_many :books, through: :authorships
end

class Authorship < ActiveRecord::Base
  belongs_to :author
  belongs_to :book
end

@author = Author.first
@author.authorships.collect { |a| a.book } # selects all books that the author's authorships belong to
@author.books                              # selects all books by using the Authorship join model
```

You can also go through a `has_many` association on the join model:

```
class Firm < ActiveRecord::Base
  has_many   :clients
  has_many   :invoices, through: :clients
end

class Client < ActiveRecord::Base
  belongs_to :firm
  has_many   :invoices
end

class Invoice < ActiveRecord::Base
  belongs_to :client
end

@firm = Firm.first
@firm.clients.flat_map { |c| c.invoices } # select all invoices for all clients of the firm
@firm.invoices                            # selects all invoices by going through the Client join model
```

Similarly you can go through a `has_one` association on the join model:

```
class Group < ActiveRecord::Base
  has_many   :users
  has_many   :avatars, through: :users
end

class User < ActiveRecord::Base
  belongs_to :group
  has_one    :avatar
end

class Avatar < ActiveRecord::Base
  belongs_to :user
```

```
end

@group = Group.first
@group.users.collect { |u| u.avatar }.compact # select all avatars for all users in the group
@group.avatars                                # selects all avatars by going through the User join model.
```

An important caveat with going through has_one or has_many associations on the join model is that these associations are **read-only**. For example, the following would not work following the previous example:

```
@group.avatars << Avatar.new   # this would work if User belonged_to Avatar rather than the other way around
@group.avatars.delete(@group.avatars.last)  # so would this
```

Setting Inverses

If you are using a belongs_to on the join model, it is a good idea to set the `:inverse_of` option on the `belongs_to`, which will mean that the following example works correctly (where `tags` is a has_many `:through` association):

```
@post = Post.first
@tag = @post.tags.build name: "ruby"
@tag.save
```

The last line ought to save the through record (a `Taggable`). This will only work if the `:inverse_of` is set:

```
class Taggable < ActiveRecord::Base
  belongs_to :post
  belongs_to :tag, inverse_of: :taggings
end
```

If you do not set the `:inverse_of` record, the association will do its best to match itself up with the correct inverse. Automatic inverse detection only works on `has_many`, `has_one`, and belongs_to associations.

Extra options on the associations, as defined in the `AssociationReflection::INVALID_AUTOMATIC_INVERSE_OPTIONS` constant, will also prevent the association's inverse from being found automatically.

The automatic guessing of the inverse association uses a heuristic based on the name of the class, so it may not work for all associations, especially the ones with non-standard names.

You can turn off the automatic detection of inverse associations by setting the `:inverse_of` option to `false` like so:

```
class Taggable < ActiveRecord::Base
  belongs_to :tag, inverse_of: false
end
```

Nested Associations

You can actually specify **any** association with the `:through` option, including an association which has a `:through` option itself. For example:

```
class Author < ActiveRecord::Base
  has_many :posts
  has_many :comments, through: :posts
  has_many :commenters, through: :comments
end

class Post < ActiveRecord::Base
  has_many :comments
end

class Comment < ActiveRecord::Base
  belongs_to :commenter
end

@author = Author.first
@author.commenters # => People who commented on posts written by the author
```

An equivalent way of setting up this association this would be:

```
class Author < ActiveRecord::Base
  has_many :posts
  has_many :commenters, through: :posts
end

class Post < ActiveRecord::Base
  has_many :comments
```

```
  has_many :commenters, through: :comments
end

class Comment < ActiveRecord::Base
  belongs_to :commenter
end
```

When using a nested association, you will not be able to modify the association because there is not enough information to know what modification to make. For example, if you tried to add a `Commenter` in the example above, there would be no way to tell how to set up the intermediate `Post` and `Comment` objects.

Polymorphic Associations

Polymorphic associations on models are not restricted on what types of models they can be associated with. Rather, they specify an interface that a `has_many` association must adhere to.

```
class Asset < ActiveRecord::Base
  belongs_to :attachable, polymorphic: true
end

class Post < ActiveRecord::Base
  has_many :assets, as: :attachable         # The :as option specifies the polymorphic interface to use.
end

@asset.attachable = @post
```

This works by using a type column in addition to a foreign key to specify the associated record. In the Asset example, you'd need an `attachable_id` integer column and an `attachable_type` string column.

Using polymorphic associations in combination with single table inheritance (STI) is a little tricky. In order for the associations to work as expected, ensure that you store the base model for the STI models in the type column of the polymorphic association. To continue with the asset example above, suppose there are guest posts and member posts that use the posts table for STI. In this case, there must be a `type` column in the posts table.

Note: The `attachable_type=` method is being called when assigning an `attachable`. The `class_name` of the `attachable` is passed as a String.

```
class Asset < ActiveRecord::Base
  belongs_to :attachable, polymorphic: true

  def attachable_type=(class_name)
     super(class_name.constantize.base_class.to_s)
  end
end

class Post < ActiveRecord::Base
  # because we store "Post" in attachable_type now dependent: :destroy will work
  has_many :assets, as: :attachable, dependent: :destroy
end

class GuestPost < Post
end

class MemberPost < Post
end
```

Caching

All of the methods are built on a simple caching principle that will keep the result of the last query around unless specifically instructed not to. The cache is even shared across methods to make it even cheaper to use the macro-added methods without worrying too much about performance at the first go.

```
project.milestones             # fetches milestones from the database
project.milestones.size        # uses the milestone cache
project.milestones.empty?      # uses the milestone cache
project.milestones(true).size  # fetches milestones from the database
project.milestones             # uses the milestone cache
```

Eager loading of associations

Eager loading is a way to find objects of a certain class and a number of named associations. It is one of the easiest ways to prevent the dreaded N+1 problem in which fetching 100 posts that each need to display their author triggers 101 database queries. Through the use of

eager loading, the number of queries will be reduced from 101 to 2.

```
class Post < ActiveRecord::Base
  belongs_to :author
  has_many   :comments
end
```

Consider the following loop using the class above:

```
Post.all.each do |post|
  puts "Post:            " + post.title
  puts "Written by:      " + post.author.name
  puts "Last comment on: " + post.comments.first.created_on
end
```

To iterate over these one hundred posts, we'll generate 201 database queries. Let's first just optimize it for retrieving the author:

```
Post.includes(:author).each do |post|
```

This references the name of the `belongs_to` association that also used the `:author` symbol. After loading the posts, find will collect the `author_id` from each one and load all the referenced authors with one query. Doing so will cut down the number of queries from 201 to 102.

We can improve upon the situation further by referencing both associations in the finder with:

```
Post.includes(:author, :comments).each do |post|
```

This will load all comments with a single query. This reduces the total number of queries to 3. In general, the number of queries will be 1 plus the number of associations named (except if some of the associations are polymorphic `belongs_to` - see below).

To include a deep hierarchy of associations, use a hash:

```
Post.includes(:author, { comments: { author: :gravatar } }).each do |post|
```

The above code will load all the comments and all of their associated authors and gravatars. You can mix and match any combination of symbols, arrays, and hashes to retrieve the associations you want to load.

All of this power shouldn't fool you into thinking that you can pull out huge amounts of data with no performance penalty just because you've reduced the number of queries. The database still needs to send all the data to Active Record and it still needs to be processed. So it's no catch-all for performance problems, but it's a great way to cut down on the number of queries in a situation as the one described above.

Since only one table is loaded at a time, conditions or orders cannot reference tables other than the main one. If this is the case, Active Record falls back to the previously used LEFT OUTER JOIN based strategy. For example:

```
Post.includes([:author, :comments]).where(['comments.approved = ?', true])
```

This will result in a single SQL query with joins along the lines of: `LEFT OUTER JOIN comments ON comments.post_id = posts.id` and `LEFT OUTER JOIN authors ON authors.id = posts.author_id`. Note that using conditions like this can have unintended consequences. In the above example posts with no approved comments are not returned at all, because the conditions apply to the SQL statement as a whole and not just to the association.

You must disambiguate column references for this fallback to happen, for example `order: "author.name DESC"` will work but `order: "name DESC"` will not.

If you want to load all posts (including posts with no approved comments) then write your own LEFT OUTER JOIN query using ON

```
Post.joins("LEFT OUTER JOIN comments ON comments.post_id = posts.id AND comments.approved = '1'")
```

In this case it is usually more natural to include an association which has conditions defined on it:

```
class Post < ActiveRecord::Base
  has_many :approved_comments, -> { where approved: true }, class_name: 'Comment'
end

Post.includes(:approved_comments)
```

This will load posts and eager load the `approved_comments` association, which contains only those comments that have been approved.

If you eager load an association with a specified `:limit` option, it will be ignored, returning all the associated objects:

```
class Picture < ActiveRecord::Base
  has_many :most_recent_comments, -> { order('id DESC').limit(10) }, class_name: 'Comment'
end

Picture.includes(:most_recent_comments).first.most_recent_comments # => returns all associated comments.
```

Eager loading is supported with polymorphic associations.

```
class Address < ActiveRecord::Base
  belongs_to :addressable, polymorphic: true
end
```

A call that tries to eager load the addressable model

```
Address.includes(:addressable)
```

This will execute one query to load the addresses and load the addressables with one query per addressable type. For example if all the addressables are either of class Person or Company then a total of 3 queries will be executed. The list of addressable types to load is determined on the back of the addresses loaded. This is not supported if Active Record has to fallback to the previous implementation of eager loading and will raise `ActiveRecord::EagerLoadPolymorphicError`. The reason is that the parent model's type is a column value so its corresponding table name cannot be put in the `FROM/JOIN` clauses of that query.

Table Aliasing

Active Record uses table aliasing in the case that a table is referenced multiple times in a join. If a table is referenced only once, the standard table name is used. The second time, the table is aliased as `#{reflection_name}_#{parent_table_name}`. Indexes are appended for any more successive uses of the table name.

```
Post.joins(:comments)
# => SELECT ... FROM posts INNER JOIN comments ON ...
Post.joins(:special_comments) # STI
# => SELECT ... FROM posts INNER JOIN comments ON ... AND comments.type = 'SpecialComment'
Post.joins(:comments, :special_comments) # special_comments is the reflection name, posts is the parent table name
# => SELECT ... FROM posts INNER JOIN comments ON ... INNER JOIN comments special_comments_posts
```

Acts as tree example:

```
TreeMixin.joins(:children)
# => SELECT ... FROM mixins INNER JOIN mixins childrens_mixins ...
TreeMixin.joins(children: :parent)
# => SELECT ... FROM mixins INNER JOIN mixins childrens_mixins ...
                            INNER JOIN parents_mixins ...
TreeMixin.joins(children: {parent: :children})
# => SELECT ... FROM mixins INNER JOIN mixins childrens_mixins ...
                            INNER JOIN parents_mixins ...
                            INNER JOIN mixins childrens_mixins_2
```

Has and Belongs to Many join tables use the same idea, but add a `_join` suffix:

```
Post.joins(:categories)
# => SELECT ... FROM posts INNER JOIN categories_posts ... INNER JOIN categories ...
Post.joins(categories: :posts)
# => SELECT ... FROM posts INNER JOIN categories_posts ... INNER JOIN categories ...
                           INNER JOIN categories_posts posts_categories_join INNER JOIN posts posts_categories
Post.joins(categories: {posts: :categories})
# => SELECT ... FROM posts INNER JOIN categories_posts ... INNER JOIN categories ...
                           INNER JOIN categories_posts posts_categories_join INNER JOIN posts posts_categories
                           INNER JOIN categories_posts categories_posts_join INNER JOIN categories categories_posts_2
```

If you wish to specify your own custom joins using `joins` method, those table names will take precedence over the eager associations:

```
Post.joins(:comments).joins("inner join comments ...")
# => SELECT ... FROM posts INNER JOIN comments_posts ON ... INNER JOIN comments ...
Post.joins(:comments, :special_comments).joins("inner join comments ...")
# => SELECT ... FROM posts INNER JOIN comments comments_posts ON ...
                           INNER JOIN comments special_comments_posts ...
                           INNER JOIN comments ...
```

Table aliases are automatically truncated according to the maximum length of table identifiers according to the specific database.

Modules

By default, associations will look for objects within the current module scope. Consider:

```
module MyApplication
  module Business
    class Firm < ActiveRecord::Base
      has_many :clients
    end
```

```
    class Client < ActiveRecord::Base; end
  end
end
```

When `Firm#clients` is called, it will in turn call `MyApplication::Business::Client.find_all_by_firm_id(firm.id)`. If you want to associate with a class in another module scope, this can be done by specifying the complete class name.

```
module MyApplication
  module Business
    class Firm < ActiveRecord::Base; end
  end

  module Billing
    class Account < ActiveRecord::Base
      belongs_to :firm, class_name: "MyApplication::Business::Firm"
    end
  end
end
```

Bi-directional associations

When you specify an association there is usually an association on the associated model that specifies the same relationship in reverse. For example, with the following models:

```
class Dungeon < ActiveRecord::Base
  has_many :traps
  has_one :evil_wizard
end

class Trap < ActiveRecord::Base
  belongs_to :dungeon
end

class EvilWizard < ActiveRecord::Base
  belongs_to :dungeon
end
```

The `traps` association on `Dungeon` and the `dungeon` association on `Trap` are the inverse of each other and the inverse of the `dungeon` association on `EvilWizard` is the `evil_wizard` association on `Dungeon` (and vice-versa). By default, Active Record doesn't know anything about these inverse relationships and so no object loading optimization is possible. For example:

```
d = Dungeon.first
t = d.traps.first
d.level == t.dungeon.level # => true
d.level = 10
d.level == t.dungeon.level # => false
```

The `Dungeon` instances `d` and `t.dungeon` in the above example refer to the same object data from the database, but are actually different in-memory copies of that data. Specifying the `:inverse_of` option on associations lets you tell Active Record about inverse relationships and it will optimise object loading. For example, if we changed our model definitions to:

```
class Dungeon < ActiveRecord::Base
  has_many :traps, inverse_of: :dungeon
  has_one :evil_wizard, inverse_of: :dungeon
end

class Trap < ActiveRecord::Base
  belongs_to :dungeon, inverse_of: :traps
end

class EvilWizard < ActiveRecord::Base
  belongs_to :dungeon, inverse_of: :evil_wizard
end
```

Then, from our code snippet above, `d` and `t.dungeon` are actually the same in-memory instance and our final `d.level == t.dungeon.level` will return `true`.

There are limitations to `:inverse_of` support:

- does not work with `:through` associations.
- does not work with `:polymorphic` associations.
- for `belongs_to` associations `has_many` inverse associations are ignored.

Deleting from associations

Dependent associations

`has_many`, `has_one` and `belongs_to` associations support the `:dependent` option. This allows you to specify that associated records should be deleted when the owner is deleted.

For example:

```
class Author
  has_many :posts, dependent: :destroy
end
Author.find(1).destroy # => Will destroy all of the author's posts, too
```

The `:dependent` option can have different values which specify how the deletion is done. For more information, see the documentation for this option on the different specific association types. When no option is given, the behavior is to do nothing with the associated records when destroying a record.

Note that `:dependent` is implemented using Rails' callback system, which works by processing callbacks in order. Therefore, other callbacks declared either before or after the `:dependent` option can affect what it does.

Delete or destroy?

`has_many` and `has_and_belongs_to_many` associations have the methods `destroy`, `delete`, `destroy_all` and `delete_all`.

For `has_and_belongs_to_many`, `delete` and `destroy` are the same: they cause the records in the join table to be removed.

For `has_many`, `destroy` and `destroy_all` will always call the `destroy` method of the record(s) being removed so that callbacks are run. However `delete` and `delete_all` will either do the deletion according to the strategy specified by the `:dependent` option, or if no `:dependent` option is given, then it will follow the default strategy. The default strategy is to do nothing (leave the foreign keys with the parent ids set), except for `has_many :through`, where the default strategy is `delete_all` (delete the join records, without running their callbacks).

There is also a `clear` method which is the same as `delete_all`, except that it returns the association rather than the records which have been deleted.

What gets deleted?

There is a potential pitfall here: `has_and_belongs_to_many` and `has_many :through` associations have records in join tables, as well as the associated records. So when we call one of these deletion methods, what exactly should be deleted?

The answer is that it is assumed that deletion on an association is about removing the *link* between the owner and the associated object(s), rather than necessarily the associated objects themselves. So with `has_and_belongs_to_many` and `has_many :through`, the join records will be deleted, but the associated records won't.

This makes sense if you think about it: if you were to call `post.tags.delete(Tag.find_by(name: 'food'))` you would want the 'food' tag to be unlinked from the post, rather than for the tag itself to be removed from the database.

However, there are examples where this strategy doesn't make sense. For example, suppose a person has many projects, and each project has many tasks. If we deleted one of a person's tasks, we would probably not want the project to be deleted. In this scenario, the delete method won't actually work: it can only be used if the association on the join model is a `belongs_to`. In other situations you are expected to perform operations directly on either the associated records or the `:through` association.

With a regular `has_many` there is no distinction between the "associated records" and the "link", so there is only one choice for what gets deleted.

With `has_and_belongs_to_many` and `has_many :through`, if you want to delete the associated records themselves, you can always do something along the lines of `person.tasks.each(&:destroy)`.

Type safety with `ActiveRecord::AssociationTypeMismatch`

If you attempt to assign an object to an association that doesn't match the inferred or specified `:class_name`, you'll get an `ActiveRecord::AssociationTypeMismatch`.

Options

All of the association macros can be specialized through options. This makes cases more complex than the simple and guessable ones possible.

Show files where this module is defined (1 file)
Register or log in to add new notes.

 netmaniac - April 23, 2009
1 thank

Using strings as association names

Beware, that using strings as association names, when giving Hash to :include will render errors:

The error occurred while evaluating nil.name

So, :include => [‘assoc1’, ‘assoc2’] will work, and :include => [{‘assoc1’ => ‘assoc3’}, ‘assoc2’] won’t. Use symbols:

Proper form

:include => [{:assoc1 => :assoc3}, ‘assoc2’]

 will - August 8, 2008
0 thanks

finder_sql

If you are using the finder_sql option, it is important to use single quotes if need to interpolate variables, such as the id of the record. Otherwise you will get the object_id of the class.

 adzdavies - July 26, 2010
0 thanks

Using strings as association names - beware of HashWithIndifferentAccess

If you merge a normal Hash into a HashWithIndifferentAccess, then the keys will convert to strings…

This will likely bite you if the merge is passed to AR find: as netmaniac said “Beware, that using strings as association names, when giving Hash to :include will render errors”.

Beware that params from your controller are HashWithIndifferentAccess like.

 vad4msiu - August 3, 2012
0 thanks

a misprint?

In section ‘Bi-directional associations’ an example:

d = Dungeon.first

t = d.traps.first

d.level == t.dungeon.level # => true

d.level = 10

d.level == t.dungeon.level # => false

Then use has_many associations, but lower than written ‘for belongs_to associations has_many inverse associations are ignored.’

- Welcome
- Register
- Projects
- Help

APIdock release: IRON STEVE (1.4)
If you have any comments, ideas or feedback, feel free to contact us at team@apidock.com