Ruby on Rails

2e édition

Dave Thomas

David Heinemeier Hansson

Avec la collaboration de Leon Breedt, Mike Clark,
James Duncan Davidson, Justin Gehtland et Andreas Schwarz

ISBN : 978-2-212-12079-0

EYROLLES

18

Active Record : relations entre les tables

La plupart des applications utilisent plusieurs tables dans leur base de données et, habituellement, il existe des relations entre certaines de ces tables. Par exemple, une commande possède plusieurs items. Un item du panier référence un produit particulier. Le produit peut lui-même appartenir à plusieurs catégories et chaque catégorie peut être liée à différents produits.

Dans le schéma de base de données, ces relations sont exprimées par le biais de liens basés sur les clés primaires[1]. Si un item de la commande référence un produit, la table `line_items` comportera une colonne contenant les valeurs de la clé primaire de l'enregistrement correspondant de la table `products`. Dans le jargon des bases de données, on dit que la table `line_items` possède une *clé étrangère* qui référence la table `products`.

Mais toutes ces considérations sont d'assez bas niveau. Dans notre application, nous ne voulons avoir affaire qu'à des objets et leurs relations et pas à des enregistrements de base de données et aux clés des tables. Si une commande possède plusieurs items, nous aimerions pouvoir simplement itérer sur ces items et, si ceux-ci référencent des produits, nous souhaiterions écrire des choses aussi simples que :

```
price = line_item.product.price
```

plutôt que :

```
product_id = line_item.product_id
product    = Product.find(product_id)
price      = product.price
```

1. Il existe un autre style de relation entre objets d'un modèle, dans lequel un modèle est la sous-classe d'un autre. Nous en reparlerons plus loin à la section *Héritage à une table*.

Active Record vient une fois de plus à la rescousse ! La magie de la couche ORM de Active Record se charge en effet de convertir toutes les relations de bas niveau de type clé étrangère en relation de haut niveau sur les objets. Active Record gère les trois cas de base :

- Un enregistrement de la table A est associé avec zéro ou un enregistrement de la table B.
- Un enregistrement de la table A est associé avec un nombre quelconque d'enregistrements de la table B.
- Un nombre quelconque d'enregistrements de la table A sont associés avec un nombre quelconque d'enregistrements de la table B.

Nous devons cependant aider un peu Active Record pour ce qui est des relations entre tables : il est en effet impossible de déduire du schéma SQL la nature des relations entre les tables souhaitées par le développeur. Toutefois, l'aide à fournir à Active Record est vraiment minime.

Créer des clés étrangères

Comme indiqué plus haut, deux tables sont liées quand l'une d'elles contient une clé étrangère référençant la clé primaire de l'autre. Dans la migration suivante, la table `line_items` contient deux clés étrangères, une sur la table `products` et l'autre sur `orders`.

```
def self.up
   create_table :products do |t|
     t.column :title, :string
     # ...
   end
   create_table :orders do |t|
     t.column :name, :string
     # ...
   end
   create_table :line_items do |t|
     t.column :product_id, :integer
     t.column :order_id,   :integer
     t.column :quantity,   :integer,
     t.column :unit_price, :decimal, :precision => 8, :scale => 2
   end
end
```

Il est à noter que cette migration ne définit aucune contrainte de clés étrangères. Les relations entre tables n'interviennent que lorsque le développeur décide de peupler les colonnes `product_id` et `order_id` avec les valeurs des clés en provenance des tables `products` et `orders`. Vous pouvez également choisir de définir ces contraintes directement dans vos migrations (ce que je vous recommande personnellement de faire), bien que le support des clés étrangères par Rails ne l'exige pas.

En observant cette migration, on comprend pourquoi il est difficile pour Active Record de déterminer la nature des relations entre deux tables. Les clés étrangères sur les tables `orders` et `products` sont déclarées de façon identique. Pourtant la colonne `product_id` est utilisée pour associer un item de la commande avec exactement un produit, alors que la colonne `order_id` est utilisée pour associer plusieurs items du panier avec une seule commande. Autrement dit, un item du panier *est une partie* de la commande mais *référence* précisément un produit.

Cet exemple montre également les conventions de nommage adoptées par Active Record. Les clés étrangères doivent être baptisées d'après le nom de la classe de la table cible, nom converti en lettres minuscules et complété par le suffixe `_id`. Il est à noter que, dans ces conditions, le nom de la clé étrangère sera toujours différent du nom de la table référencée. Si vous utilisez un modèle Active Record nommé `Person`, il sera mis en correspondance avec la table `people` en base de données. Une clé étrangère, dans une autre table, référençant la table `people` portera le nom de colonne `person_id`.

Un autre type de relation concerne le cas où un certain nombre d'objets sont liés à un certain nombre d'autres (par exemple, des produits qui appartiennent à plusieurs catégories et des catégories qui peuvent contenir plusieurs produits). SQL veut que ce type de relation soit géré dans une troisième table appelée *table de jointure*. La table de jointure contient une clé étrangère vers chacune des tables concernées. Ainsi, chaque enregistrement de la table de jointure représente un lien entre les deux tables. Observons cette autre migration :

```
def self.up
  create_table :products do |t|
    t.column :title, :string
    # ...
  end
  create_table :categories do |t|
    t.column :name, :string
    # ...
  end
  create_table :categories_products, :id => false do |t|
    t.column :product_id,  :integer
    t.column :category_id, :integer
  end
        # Les index sont importants pour la performance
  add_index :categories_products, [:product_id, :category_id]
  add_index :categories_products, :category_id
end
```

Rails suppose que le nom de la table de jointure est composé du nom des deux tables jointes classés par ordre alphabétique. Ainsi, Rails s'attend à ce que la table de jointure des tables `categories` et

`products` porte le nom `categories_products`. Si vous utilisez un autre nom pour votre table de jointure, vous devrez ajouter une déclaration explicite permettant à Rails de la retrouver.

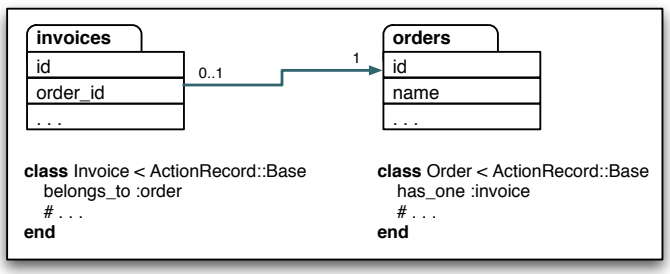
Notre table de jointure ne nécessite pas de colonne d'identifiant comme clé primaire car la combinaison des identifiants produit et catégorie est unique. C'est pour cette raison que nous empêchons la migration de l'ajouter automatiquement en lui spécifiant `:id =>false`. Nous créons ensuite deux index sur la table de jointure. Le premier, qui est un index composite, sert à deux choses. Il permet d'effectuer une recherche basée sur les deux colonnes de clés étrangères de la table de jointure et, pour la plupart des systèmes de base de données, il permettra la création d'un index autorisant une recherche rapide sur l'identifiant d'un produit. Le second index complète le paysage en permettant la recherche rapide sur l'identifiant de catégorie.

Spécifier les relations dans les modèles

Active Record supporte trois types de relation entre les tables : un-vers-un, un-vers-N et N-vers-N. La spécification de la relation à utiliser dans vos modèles se fait par l'intermédiaire des déclarations suivantes : `has_one`, `has_many`, `belongs_to` et `has_and_belongs_to_many`.

Association un-vers-un

Une association un-vers-un (ou plus exactement une relation un-vers-zéro-ou-un) s'implémente par l'utilisation d'une clé étrangère dans une colonne d'une table pour référencer au plus un enregistrement particulier d'une autre table. Une relation un-vers-un peut exister, par exemple, entre une commande et sa facture : pour chaque commande il y au plus une facture.

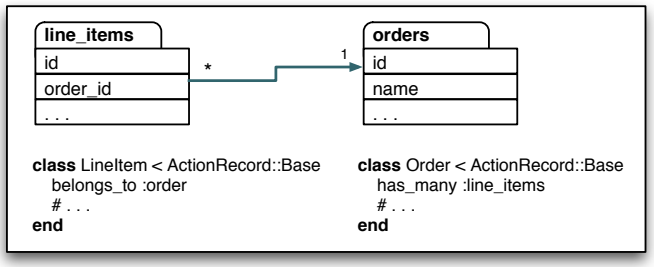


Comme montré dans l'exemple, cette relation est précisée en Rails par l'ajout d'une déclaration `has_one` dans le modèle `Order` et d'une déclaration `belongs_to` dans le modèle `Invoice`.

Une règle importante est ici illustrée : le modèle de la table contenant la clé étrangère contient *toujours* la déclaration `belongs_to`.

Association un-vers-N

Une association un-vers-N permet de représenter une collection d'objets. Par exemple, une commande peut avoir un nombre quelconque d'items associés. Dans la base de données, tous les enregistrements des items d'une commande particulière contiennent une clé étrangère qui se réfère à cette commande.

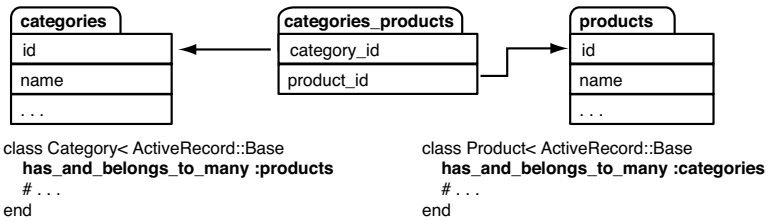


Dans Active Record, l'objet parent (celui qui contient la collection d'objets fils) utilise la déclaration `has_many` pour déclarer sa relation à la table fille, et cette même table utilise la déclaration `belongs_to` pour indiquer la relation à sa table parente. Dans notre exemple, il s'agit de la classe `LineItem` `belongs_to :order` (appartient à) et de la table `orders has_many :line_items` (a plusieurs).

Parce que les enregistrements des items d'une commande contiennent une clé étrangère référençant cette commande, vous remarquerez, à nouveau, que le modèle `LineItem` contient la déclaration `belongs_to`.

Association N-vers-N

Pour finir, nous pourrions catégoriser nos produits. Un produit peut appartenir à plusieurs catégories, et chaque catégorie peut contenir de multiples produits. C'est un exemple d'association N-vers-N, un peu comme s'il existait réciprocité d'appartenance de part et d'autre de la relation.



En Rails nous exprimons ceci par l'ajout de la déclaration `has_and_belongs_to_many` dans chacun des modèles (à partir de ce point, nous utiliserons l'abréviation `habtm` pour signifier cette déclaration).

Les associations N-vers-N sont symétriques : chacune des tables concernées déclare son association avec l'autre en utilisant `habtm`.

Dans la base de données, les associations N-vers-N sont implémentées par l'intermédiaire d'une table de jointure. Elle contient des paires de clés étrangères qui référencent les tables cibles et son nom est bâti par concaténation des noms des deux tables jointes classés par ordre alphabétique (et séparés par un tiret-bas « _ »). Dans l'exemple qui suit, on joint les tables `categories` et `products`, ce qui amène Active Record à rechercher dans la base de données une table de jointure dont le nom est `categories_products`.

Déclarations belongs_to et has_xxx

Les diverses déclarations relationnelles (`belongs_to`, `has_one`, etc.) font plus que spécifier les relations entre les tables. Elles ajoutent chacune au modèle tout un ensemble de méthodes facilitant la navigation entre les objets associés. Voyons maintenant ceci plus en détail (vous pouvez accéder directement au résumé de cette section dans la figure 18-3).

Déclaration belongs_to

`belongs_to()`déclare que la classe spécifiée établit une relation de parenté avec la classe qui contient la déclaration. Bien que le terme *belongs to* (appartient à) puisse ne pas être celui qui vient en premier à l'esprit pour qualifier ce type de relation, Active Record l'utilise pour signifier que la table qui contient la clé étrangère appartient à la table qu'elle référence. Si cela peut vous aider, vous pouvez penser au verbe référence quand vous tapez `belongs_to`.

Le nom de la classe parente est supposé être de la forme à casse mixte au singulier et basé sur le nom de l'attribut. La clé étrangère est la forme au singulier du nom de l'attribut complété du suffixe `_id`. Considérons le code suivant :

```
class LineItem < ActiveRecord::Base
  belongs_to :product
  belongs_to :invoice_item
end
```

Déclaration	Clé Étrangère	Classe Cible	Table Cible
belongs_to :product	product_id	Product	products
belongs_to :invoice_item	invoice_item_id	InvoiceItem	invoice_items

Active Record relie les lignes de commande aux objets de la classe `Product` et `InvoiceItem`. Le schéma sous-jacent utilise les clés `product_id` et `invoice_item_id` pour référencer les colonnes `id` des tables `products` et `invoice_items`, respectivement.

Vous pouvez remplacer ces comportements par défaut en passant en paramètre à `belongs_to()` un tableau associatif qui explicite le nom de l'association.

```
class LineItem < ActiveRecord::Base
  belongs_to :paid_order,
             :class_name  => "Order",
             :foreign_key => "order_id",
             :conditions  => "paid_on is not null"
end
```

Dans cet exemple, nous avons créé une association appelée `paid_order`, qui est une référence à la classe `Order` (et donc à la table `orders`). Le lien se fait via la clé étrangère `order_id`, mais il est en plus complété par une condition qui stipule que la commande ne sera référencée que si la colonne `paid_on` de la cible est non nulle. Dans le cas présent, notre association n'a pas de correspondance directe vers une seule colonne de la table `line_items`. Par ailleurs, `belongs_to` accepte d'autres options que nous traiterons ultérieurement en étudiant des sujets plus avancés.

La méthode `belongs_to()` fournit un certain nombre de méthodes d'instance pour gérer l'association. Le nom de ces méthodes débute par le nom de l'association. Prenons la classe `LineItem` pour exemple :

```
class LineItem < ActiveRecord::Base
  belongs_to :product
end
```

Dans ce cas, les méthodes suivantes seront disponibles pour les objets correspondant aux items de la commande ainsi que pour les produits auxquels ils appartiennent.

`product(force_reload=false)`

Retourne le produit associé ou `nil` s'il n'y en a pas. Le résultat est placé dans le cache, ce qui signifie qu'une requête similaire ne sollicitera pas la base de données, à moins que `true` ne soit passé en paramètre.

De façon plus générale, cette méthode est appelée comme s'il s'agissait d'un simple attribut d'un item de la commande.

```
li = LineItem.find(1)
puts "The product name is #{li.product.name}"
```

`product=`*obj*

Associe l'item de la commande avec le produit concerné en stockant, dans la clé étrangère de cet item de commande, la valeur de la clé primaire du produit. Si le produit n'a pas été sauvegardé, il le sera lors de la sauvegarde de l'item de la commande. Les clés seront reliées à ce moment-là.

`build_product(`*attributes={}*`)`

Construit un nouvel objet produit et l'initialise avec les attributs spécifiés. L'item de commande courant y sera relié. Le produit n'est pas sauvegardé dans la base de données.

`create_product(`*attributes={}*`)`

Construit un nouvel objet produit, lui relie l'item de commande courant et sauvegarde le produit dans la base de données.

Voyons maintenant quelques-unes de ces méthodes créées automatiquement en action. Étant donné les modèles suivants :

code/e1/ar/associations.rb

```
class Product < ActiveRecord::Base
  has_many :line_items
end
class LineItem < ActiveRecord::Base
  belongs_to :product
end
```

Et en considérant que la base de données contient déjà des produits et des commandes, lançons le code suivant.

code/e1/ar/associations.rb

```
item = LineItem.find(2)
# item.product est l'objet Product associé
puts "Le produit courant est #{item.product.id}"
puts item.product.title
item.product = Product.new(:title          => "Rails for Java Developers",
                           :description    => "...",
                           :image_url      => "http://....jpg",
                           :price          => 34.95,
                           :available_at   => Time.now)
item.save!
puts "Le nouveau produit est #{item.product.id}"
puts item.product.title
```

Si nous exécutons ce code (avec une connexion appropriée à la base de données), nous verrons s'afficher :

```
Le produit courant est 1
Programming Ruby
Le nouveau produit est 2
Rails for Java Developers
```

Nous utilisons les méthodes `product()` et `product=()` générées dans la classe `LineItem` pour accéder et mettre à jour l'objet produit associé à un objet item de la commande. En arrière-plan, Active Record a automatiquement sauvegardé le produit nouvellement créé et a lié un item de la commande à l'identifiant de ce nouveau produit.

Nous pourrions tout aussi bien utiliser la méthode auto-générée `create_product` pour la création d'un nouveau produit et son association avec notre item de commande courant.

code/e1/ar/associations.rb

```
item.create_product(:title          => "Rails Recipes",
                    :description    => "...",
                    :image_url      => "http://....jpg",
                    :price          => 32.95,
                    :available_at   => Time.now)
```

Nous avons utilisé `create_` plutôt que `build_`, et il n'est donc pas nécessaire de sauver le produit créé.

Déclaration has_one

`has_one` déclare qu'une classe donnée (par défaut de la forme à casse mixte et au singulier du nom de l'attribut) est une classe fille de la classe courante. Ceci signifie que la table correspondante à la classe fille contiendra une clé étrangère en référence à la classe contenant la déclaration. Le code suivant déclare la table `invoices` comme étant fille de la table `orders` :

```
class Order < ActiveRecord::Base
  has_one :invoice
end
```

Déclaration	Clé Étrangère	Classe Cible	Table Cible
has_one :invoice	order_id (dans la table invoices)	Invoice	invoices

La déclaration `has_one` définit le même jeu de méthodes dans l'objet d'un modèle que la déclaration `belongs_to`. Ainsi, étant donnée la définition précédente de la classe `Order`, nous pourrions écrire :

```
order = Order.new(... attributes ...)
invoice = Invoice.new(... attributes ...)
order.invoice = invoice
```

S'il n'existe pas d'enregistrement fils en relation avec un enregistrement parent, l'association `has_one` retournera `nil` (ce qui en Ruby est traité comme `false`). Ce qui permet d'écrire :

```
if order.invoice
  print_invoice(order.invoice)
end
```

S'il existe déjà un objet fils lors de l'assignation d'un nouvel objet avec une association `has_one`, l'objet fils déjà existant sera mis à jour avec suppression de la clé étrangère d'association à l'objet parent (la clé étrangère prendra la valeur `null`) comme exposé dans la figure 18-1.

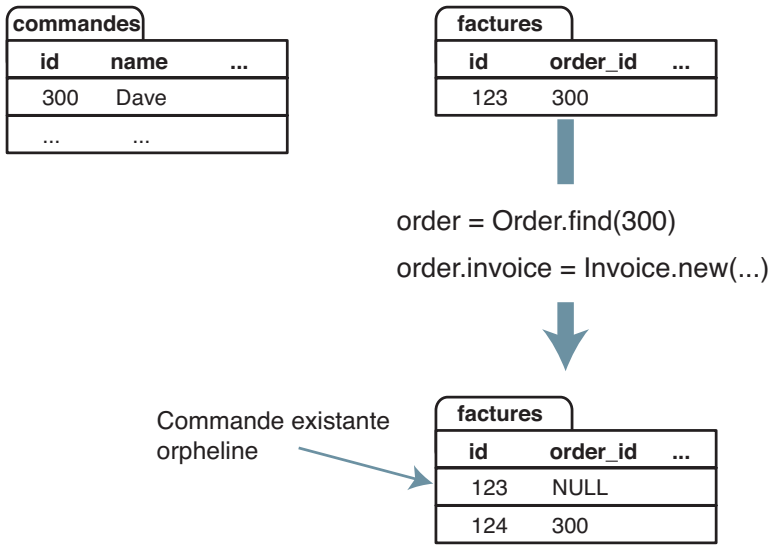


Figure 18-1
Ajout dans une relation has_one

Options de has_one

Vous pouvez modifier le comportement par défaut en passant un tableau associatif d'options à `has_one`. En plus des options `:class_name`, `:foreign_key` et `:conditions` vues dans `belongs_to()`, `has_one` possède d'autres options (abordées plus tard) mais l'une d'elle retiendra notre attention.

L'option `:dependent` indique à Active Record ce qu'il doit faire des enregistrements fils lors de la destruction d'un enregistrement dans la table parent. Il existe cinq valeurs possibles pour cette option.

`:dependent =>:destroy` *(ou* `true`*)*

L'enregistrement fils sera détruit en même temps que l'enregistrement parent.

`:dependent =>:nullify`

L'enregistrement fils deviendra orphelin dès l'instant où l'enregistrement parent sera détruit. La clé étrangère de l'orphelin prendra la valeur `null`.

`:dependent =>false` *(ou* `nil`*)*

L'enregistrement fils ne sera ni détruit ni mis à jour lors de la destruction de l'enregistrement parent. Si vous avez défini des contraintes de clés étrangères entre les tables filles et les tables parents, l'utilisation de cette option peut conduire à une violation de ces contraintes lors de la destruction de l'enregistrement parent.

L'option `:order` qui détermine la façon dont les enregistrements sont triés avant d'être retournés à l'appelant, est un peu étrange. Nous en reparlerons après avoir couvert la déclaration `has_many`.

Déclaration has_many

`has_many` définit un attribut qui se comporte comme une collection d'objets fils.

```
class Order < ActiveRecord::Base
  has_many :line_items
end
```

Déclaration	Clé Étrangère	Classe Cible	Table Cible
has_many :line_items	order_id (dans la table line_items)	LineItem	line_items

Cette collection d'objets est accessible sous la forme d'un tableau où il est possible de trouver un objet fils particulier ou d'en ajouter un nouveau. Ainsi, l'exemple de code qui suit ajoute un item à une commande.

```
order = Order.new
params[:products_to_buy].each do |prd_id, qty|
  product = Product.find(prd_id)
  order.line_items << LineItem.new(:product  => product,
                   :quantity => qty)
end
order.save
```

L'opérateur d'ajout (`<<()`) n'ajoute pas seulement un objet à une liste. Il fait aussi en sorte de lier l'item avec la commande, en positionnant la clé étrangère avec la valeur de l'identifiant de la commande, et s'assure que les items de la commande sont enregistrés dans la base de données, quand la commande est elle-même sauvegardée.

Il est possible d'itérer sur les enfants d'une relation `has_many` puisque l'attribut se comporte comme un tableau.

```
order = Order.find(123)
total = 0.0
order.line_items.each do |li|
  total += li.quantity * li.unit_price
end
```

Comme pour `has_one`, vous pouvez modifier le comportement par défaut de Active Record en fournissant un tableau associatif d'options à `has_many`. Les options `:class_name`, `:foreign_key` et `:conditions` fonctionnent de la même façon que pour la méthode `has_one`.

L'option `:dependent` accepte les valeurs `:destroy`, `:nullify` et `false`, qui ont la même signification qu'avec `has_one`, sauf qu'elles s'appliquent à tous les enregistrements fils. Toutefois, la version `has_many` de `:dependent` offre une valeur additionnelle, `:delete_all`. Comme pour l'option `:destroy`, celle ci élimine les enregistrements fils si un enregistrement parent est supprimé. Voyons en quoi ces deux options diffèrent.

`:dependent =>:destroy` fonctionne en parcourant la table fille et en appelant `destroy()` pour chaque enregistrement dont la clé étrangère référence la clé primaire de l'enregistrement détruit dans la table parente.

Cependant, si la table fille est utilisée uniquement par cette table parente (c'est-à-dire qu'elle n'a aucune autre dépendance) et qu'elle n'a pas de méthodes reliées aux points d'accrochage (hooks) utilisés lors de la destruction, vous pouvez utiliser la déclaration `:dependent`

=>:delete_all. Si cette option est utilisée, les enregistrements fils sont tous détruits en une seule requête SQL, ce qui est évidemment plus rapide.

Finalement, vous pouvez remplacer le SQL utilisé par Active Record pour récupérer et compter les enregistrements fils en positionnant les options :finder_sql et :counter_sql. Cela peut être utile dans les cas où l'ajout de quelques instructions dans la clause where par le biais de l'option :condition ne suffit pas. Vous pouvez, par exemple, créer une collection de tous les items de la commande pour un produit particulier.

```
class Order < ActiveRecord::Base
  has_many :rails_line_items,
           :class_name => "LineItem",
           :finder_sql => "select l.* from line_items l, products p " +
                          " where l.product_id = p.id " +
                          "   and p.title like '%rails%'"
end
```

L'option :counter_sql est utilisée pour remplacer la requête que Active Record utilise pour le comptage des enregistrements. Si :finder_sql est spécifiée et que :counter_sql ne l'est pas, Active Record fabrique le compteur SQL en remplaçant l'instruction select du finder SQL par select count(*).

L'option :order spécifie un fragment SQL qui permet d'établir l'ordre dans lequel les enregistrements sont classés dans la collection retournée par la base de données. Si vous voulez que les enregistrements vous soient retournés dans un ordre particulier, vous devez utiliser l'option :order. Le fragment SQL fourni est tout simplement celui qui apparaît après la clause order by d'une instruction select. Il consiste en une liste d'un ou plusieurs noms de colonne. La collection est d'abord triée sur la première colonne, puis sur la deuxième si les valeurs de la première sont identiques et ainsi de suite. Par défaut, le tri se fait par ordre croissant. Incluez le mot-clé DESC après un nom de colonne si vous souhaitez que le tri se fasse par ordre décroissant.

L'exemple de code suivant permet de trier les items d'une commande par ordre croissant de quantité.

```
class Order < ActiveRecord::Base
  has_many :line_items,
           :order => "quantity, unit_price DESC"
end
```

Si deux items de la commande ont la même quantité, celui possédant le prix unitaire le plus élevé apparaît en premier.

En présentant la déclaration `has_one`, plus haut dans ce chapitre, nous avons mentionné son l'option `:order`. Cela vous paraît peut-être étrange qu'un objet parent associé à un seul enfant dispose d'un moyen de spécifier un ordre de classement pour l'objet enfant.

Il s'avère que Active Record est en mesure de créer des relations `has_one` même quand elles n'existent pas dans la base de données. Ainsi, un client peut avoir plusieurs commandes et il s'agit bien d'une relation `has_many`. Mais l'association entre le client et sa commande la plus récente est unique et peut s'exprimer sous la forme d'une relation `has_one` combinée avec l'option `:order`.

```
class Customer < ActiveRecord::Base
  has_many :orders
  has_one  :most_recent_order,
     :class_name => 'Order',
     :order      => 'created_at DESC'
end
```

Cet extrait de code crée un nouvel attribut dans le modèle appelé `most_recent_order` (commande la plus récente). Il pointe sur la commande ayant le marqueur horaire `created_at` le plus récent. On peut donc utiliser cet attribut pour trouver la commande la plus récente d'un client.

```
cust = Customer.find_by_name("Dave Thomas")
puts "Dave last ordered on #{cust.most_recent_order.created_at}"
```

Tout ceci fonctionne parce que Active Record récupère les enregistrements de la relation `has_one` en utilisant l'instruction SQL suivante :

```
SELECT * FROM orders
WHERE  customer_id = ?
ORDER  BY created_at DESC
LIMIT  1
```

La clause `limit` indique qu'un seul enregistrement doit être retourné, satisfaisant ainsi la partie one de la déclaration `has_one`. La clause `order by`, quant à elle, s'assure que l'unique enregistrement renvoyé est bien le plus récent.

Nous couvrirons d'autres options supportées par `has_many` lorsque nous nous intéresserons à des aspects plus avancés de Active Record.

Les méthodes ajoutées par has_many()

À l'instar de `belongs_to` et `has_one`, `has_many` ajoute un certain nombre de méthodes liées aux attributs de la classe qui l'héberge. Là encore, les noms de ces méthodes commencent par le

nom de l'attribut auquel elles s'appliquent. Dans la description qui suit, nous présentons les méthodes qui sont ajoutées par la déclaration :

```
class Customer < ActiveRecord::Base
  has_many :orders
end
```

`orders(force_reload=false)`

Retourne un tableau des commandes associées à ce client (le tableau peut être vide s'il n'y a aucune commande associée). Le résultat est placé dans le cache ce qui signifie qu'une requête similaire ne sollicitera pas la base de données à moins que `true` ne soit passé en paramètre.

`orders<<`*commande*

Ajoute une commande (`order`)à la liste des commandes (`orders`) de ce client.

`orders.push(`*commande1, ...*`)`

Ajoute un ou plusieurs objets de type commande à la liste des commandes associées à ce client. `concat()` est un nom d'alias de la même méthode.

`orders.replace(`*commande1, ...*`)`

Remplace le jeu de commandes associées à ce client par un nouveau jeu d'enregistrements. Détecte les différences entre le jeu courant et le nouveau jeu de commandes en synchronisant les changements en base de données.

`orders.delete(`*commande1, ...*`)`

Détruit un ou plusieurs objets de type commande (`order`) de la liste des commandes associées à ce client. Si l'association est marquée avec `:dependent => :destroy` ou `:delete_all` , chaque commande (`order`) sera détruite dans la base de données. Dans les autres cas, les enfants ne seront pas détruits, mais leur clé étrangère sera simplement réinitialisée à `null`, détruisant ainsi leur association.

`orders.delete_all`

Appelle la méthode destructive `delete` sur chaque enregistrement fils.

`orders.destroy_all`

Appelle la méthode destructive `destroy` sur tous les enregistrements fils.

`orders.clear`

Dissocie toutes les commandes de ce client. Comme `delete()`, cette méthode efface l'association mais détruit les commandes correspondantes de la base de données, si l'option `:dependent` est positionnée à `true`.

`orders.find(`*options...*`)`

Utilise un appel à la méthode `find()` bien connue, mais limite les résultats uniquement aux commandes associées à ce client. Fonctionne avec les formes `id`, `:all` et `:first` de `find()`.

`orders.count(`*options...*`)`

Retourne le nombre d'enfants. Si vous utilisez un finder particulier ou un compteur SQL, ce SQL sera utilisé. Dans les autres cas, c'est la méthode `count` standard d'Active Record qui sera utilisée, contrainte aux enregistrements fils possédant la clé étrangère appropriée. Tous les paramètres optionnels de `count` peuvent être utilisés.

`orders.size`

Retourne la taille de la collection courante si vous avez déjà chargé l'association (en y ayant accédé). Dans le cas contraire, retourne un comptage par interrogation de la base de données. À la différence de `count`, la méthode `size` honore toutes les options `:limit` passées à `has_many` et n'utilise pas `finder_sql`.

`orders.length`

Force le rechargement de l'association et retourne sa taille.

`orders.empty?`

Équivaut à l'appel de la méthode `orders.size.zero?`.

`orders.sum(`*options...*`)`

Équivaut à l'appel de la méthode `sum` d'Active Record (documentée à la section *Obtenir des statistiques de colonne* du chapitre 17) sur les enregistrements de l'association. Notez bien que ceci fonctionne par l'emploi de fonctions SQL sur les enregistrements en base de données, et non par itération sur la collection chargée en mémoire.

`orders.uniq`

Retourne un tableau des enfants avec un identifiant unique.

`orders.build(`*attributs={}*`)`

Construit un nouvel objet de la classe `commande`, initialisé avec les attributs spécifiés et lié au client. Il n'est pas sauvé dans la base de données.

`orders.create(`*attributs={}*`)`

Construit et sauve un nouvel objet de la classe `commande`, initialisé avec les attributs spécifiés et lié au client.

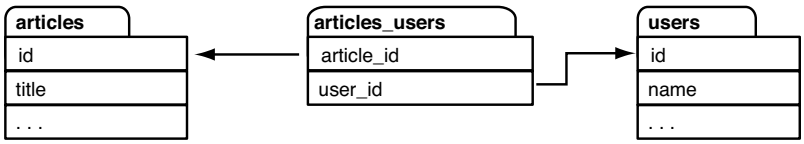
Oui, c'est un peu confus ...

Vous avez peut-être l'impression qu'il y a beaucoup de duplications (ou presque) au sein des méthodes ajoutées à votre classe Active Record par la déclaration `has_many`. Les différences entre, par exemple, `count`, `size` et `length`, ou entre `clear`, `destroy_all` et `delete_all`, sont subtiles. Ceci est largement dû à l'accumulation graduelle de fonctionnalités dans Active Record au fil du temps. Au fur et à mesure de l'apparition de nouvelles options, les méthodes existantes n'ont pas nécessairement été mises à jour. Mon sentiment est qu'à un moment donné, ceci sera résolu et les méthodes unifiées. Il est bon d'étudier la documentation en ligne de l'API Rails, parce que Rails pourrait bien subir des changements après la parution de cet ouvrage.

Déclaration has_and_belongs_to_many

`has_and_belongs_to_many` (ci-après abrégée en `habtm`), se comporte de façon très similaire à `has_many`. D'une part, `habtm` crée un attribut qui est, pour l'essentiel, une collection. Cet attribut offre les mêmes méthodes que `has_many`. D'autre part, `habtm` vous permet d'ajouter des informations à la table de jointure lorsque vous associez deux objets (bien que, comme nous allons le voir, cette capacité tombe peu à peu en désuétude).

Intéressons-nous, pour une fois, à autre chose qu'à notre application `Dépôt`. Supposons que nous utilisions Rails pour écrire un site communautaire où les utilisateurs peuvent lire des articles. On trouve dans cette application de nombreux utilisateurs, des articles en grand nombre et n'importe quel utilisateur peut lire n'importe quel article. Pour des raisons de suivi, nous aimerions connaître les lecteurs de chaque article ainsi que les articles consultés par chaque utilisateur. Nous sommes aussi intéressés par la date à laquelle un utilisateur a consulté tel ou tel article pour la dernière fois. Nous allons réalisé tout cela avec une simple table de jointure. En Rails, le nom de la table de jointure est la concaténation des noms des deux tables qui sont jointes, présentés dans l'ordre alphabétique.



Mettons en place nos deux classes de modèle de façon à ce qu'elles soient en relation via cette table de jointure.

```
class Article < ActiveRecord::Base
  has_and_belongs_to_many :users
  # ...
end
class User < ActiveRecord::Base
```

```
  has_and_belongs_to_many :articles
  # ...
end
```

Telles quelles, ces classes nous permettent déjà de lister tous les utilisateurs qui ont lu l'article 123 et tous les articles lus par l'utilisateur `pragdave`.

```
# Qui a lu l'article 123 ?
article = Article.find(123)
readers = article.users
# Qu'a lu Dave ?
dave = User.find_by_name("pragdave")
articles_that_dave_read = dave.articles
# Combien de fois chaque utilisateur a-t-il lu l'article 123 ?
counts = Article.find(123).users.count(:group => "users.name")
```

Quand l'application note que quelqu'un a lu un article, elle relie l'enregistrement utilisateur à l'enregistrement article en utilisant une méthode d'instance de la classe `User`.

```
class User < ActiveRecord::Base
  has_and_belongs_to_many :articles
  # cet utilisateur vient juste de lire l'article
  def just_read(article)
    articles << article
  end
  # ...
end
```

Qu'aurions nous fait si nous avions souhaité enregistrer davantage d'informations avec l'association entre l'utilisateur et l'article, comme la date de lecture ? Autrefois (fin 2005), nous aurions fait appel à la méthode `push_with_attributes()`, qui remplissait la même fonction de liaison des deux modèles que la méthode d'adjonction `<<` vue précédemment, mais qui ajoutait aussi les valeurs spécifiées dans les enregistrements de la table de jointure, enregistrements créés à chaque fois qu'un utilisateur lit un article.

Toutefois, `push_with_attributes` s'est dépréciée en faveur d'une technique de loin plus puissante, dans laquelle les modèles Active Record sont utilisés comme tables de jointure (rappelez-vous qu'avec `habtm`, la table de jointure n'est pas un objet Active Record). Nous discuterons de ce schéma dans la prochaine section.

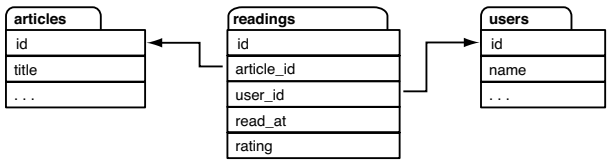
Comme pour d'autres méthodes de relation, `habtm` supporte une panoplie d'options qui modifie le comportement par défaut de Active Record. `:class_name`, `:foreign_key` et `:conditions` fonctionnent de la même façon que les autres méthodes `has_` (l'option

`:foreign_key` positionne le nom de la clé étrangère pour cette table dans la table de jointure). D'autre part, `habtm()` supporte les options qui permettent de remplacer le nom de la table de jointure, le nom des clés étrangères de cette table et le code SQL qui permet de trouver, d'insérer et de détruire les liens entre deux modèles. Référez-vous à la documentation de l'API pour plus d'informations.

Utiliser les modèles comme tables de jointure

La pensée Rails actuelle voudrait que les tables de jointure soit conservées aussi pures que possible. Elles ne devraient contenir qu'une paire de colonnes de clés étrangères. Dès lors que vous ressentez le besoin de rajouter davantage de données dans ce genre de table, vous ne faites rien d'autre que créer un nouveau modèle. La table de jointure glisse du statut de simple mécanisme de mise en relation à celui de participant à part entière dans la logique de votre application. Mais revenons à l'exemple précédent des articles et des utilisateurs.

Dans une implémentation simple de `habtm`, la table de jointure enregistre le fait qu'un article soit lu par un utilisateur. Les enregistrements de la table de jointure n'ont pas d'existence indépendante. Mais très rapidement, nous souhaitons augmenter les informations contenues dans cette table : nous voulons enregistrer la date de lecture de l'article et le nombre d'étoiles attribuées à l'article en fin de lecture. La table de jointure prend soudainement vie et mérite dès lors son propre modèle Active Record. Appelons-le `Reading` (lecture). Son schéma correspond à ceci :



En utilisant les possibilités de Rails que nous avons évoquées dans ce chapitre, nous pourrions construire le modèle suivant :

```
class Article < ActiveRecord::Base
  has_many :readings
end
class User < ActiveRecord::Base
  has_many :readings
end
class Reading < ActiveRecord::Base
  belongs_to :article
  belongs_to :user
end
```

Quand un utilisateur lit un article, nous pouvons l'enregistrer.

```
reading = Reading.new
reading.rating  = params[:rating]
reading.read_at = Time.now
reading.article = current_article
reading.user    = session[:user]
reading.save
```

Cependant, nous perdons ici quelque chose en comparaison avec la solution `habtm`. Nous ne pouvons plus aussi facilement retrouver les lecteurs d'un article ou déterminer les articles lus par chaque lecteur. C'est ici que l'option `:through` entre en jeu. Mettons à jour nos modèles `Article` et `User`.

```
class Article < ActiveRecord::Base
  has_many :readings
  has_many :users, :through => :readings
end
class User < ActiveRecord::Base
  has_many :readings
  has_many :articles, :through => :readings
end
```

L'option `:through` employée sur les deux nouvelles déclarations `has_many()` indiquent à Rails que la table `readings` peut être utilisée pour naviguer d'un article à un nombre d'utilisateurs ayant lu cet article. Nous pouvons maintenant écrire :

```
readers = an_article.users
```

En tâche de fond, Rails construit le code SQL nécessaire pour retourner tous les enregistrements d'utilisateurs référencés à partir de la table `readers` et dans laquelle les enregistrements `readers` référencent l'article original (Wooow !).

Le paramètre `:through` nomme l'association permettant de naviguer dans la classe de modèle original. De ce fait, lorsque nous disons :

```
class Article < ActiveRecord::Base
  has_many :readings
  has_many :users, :through => :readings
end
```

Le parameter `:through =>:readings` informe Active Record qu'il doit utiliser l'association `has_many :readings` pour trouver un modèle nommé `Reading`.

Le nom que nous avons donné à l'association (`:users` dans ce cas précis) indique ensuite à Active Record quel attribut utiliser pour rechercher les utilisateurs (`user_id`). Vous pouvez changer ceci en ajoutant un paramètre `:source` à la déclaration `has_many`. Par exemple, jusqu'ici, nous avons appelé *users* les personnes ayant lu un article, simplement parce que c'était le nom de l'association dans le modèle `Reading`. Il serait cependant facile de les appeler des *readers* (lecteurs) : il nous suffit de modifier le nom de l'association utilisée.

```
class Article < ActiveRecord::Base
  has_many :readings
  has_many :readers, :through => :readings, :source => :user
end
```

En fait, nous pouvons même aller encore plus loin. Il s'agit toujours d'une déclaration `has_many` et, en tant que telle, elle accepte tous les paramètres de `has_many`. Par exemple, nous pouvons créer une association qui retournera tous les utilisateurs qui ont noté nos articles avec quatre étoiles ou plus.

```
class Article < ActiveRecord::Base
  has_many :readings
  has_many :readers, :through => :readings, :source => :user
  has_many :happy_users, :through => :readings,  :source => :user,
         :conditions => 'readings.rating >= 4'
end
```

Suppression des doublons

Les collections retournées par `has_many :through` ne sont que le résultat de l'observation des relations de jointure sous-jacentes. De ce fait, si un utilisateur a lu un article particulier à trois reprises, interroger la liste des utilisateurs de cet article nous retournera trois copies du modèle utilisateur pour cette personne (parmi celles des autres lecteurs de l'article). Il existe deux façons d'éliminer les doublons.

Premièrement, vous pouvez ajouter le qualificateur `:unique =>true` à la déclaration `has_many`.

```
class Article < ActiveRecord::Base
  has_many :readings
  has_many :users, :through => :readings, :unique => true
end
```

Ceci est totalement implanté au sein de Active Record : un jeu complet d'enregistrements est retourné par la base de données et Active Record le traite en éliminant tous les objets dupliqués.

Il existe également un bricolage permettant le dédoublonnage dans la base de données en modifiant l'instruction `select` du SQL généré par Active Record par l'adjonction du

qualificateur `distinct`. Vous devez vous souvenir qu'il est nécessaire d'ajouter le nom de la table parce que la requête SQL contient une instruction `join`.

```
class Article < ActiveRecord::Base
  has_many :readings
  has_many :users, :through => :readings, :select => "distinct users.*"
end
```

Vous pouvez créer de nouvelles associations `:through` en utilisant la méthode `<<()` (qui a pour alias `push()`). Les deux extrémités de l'association doivent avoir été préalablement sauvegardées pour que ceci fonctionne.

```
class Article < ActiveRecord::Base
  has_many :readings
  has_many :users, :through => :readings
end
user = User.create(:name => "dave")
article = Article.create(:name => "Join Models")
          article.users << user
```

Il est enfin possible d'utiliser la méthode `create!()` pour créer un enregistrement à l'extrémité d'une association. Ce code est équivalent à l'exemple précédent :

```
article = Article.create(:name => "Join Models")
article.users.create!(:name => "dave")
```

Notez qu'il n'est pas possible de définir des attributs dans la table intermédiaire en utilisant cette approche.

Étendre les associations

Une association (`belongs_to`, `has_xxx`) vaut déclaration sur les relations entre vos objets de modèle. Assez fréquemment, il existe une logique métier additionnelle associée à cette déclaration particulière. Dans l'exemple précédent, nous avons défini une relation entre des articles et leurs lecteurs appelée `Reading`. Cette relation incorporait la notation des utilisateurs concernant l'article qu'ils venaient juste de lire. Pour un utilisateur donné, comment obtenir une liste de tous les articles qu'il a notés avec trois étoiles ou plus ? Quatre ou plus ? Et ainsi de suite...

Nous avons vu précédemment avec l'exemple de `happy_users` (page 383) une façon de construire de nouvelles associations pour lesquelles les jeux de résultats satisferont des critères additionnels. Toutefois, cette méthode était limitée : on ne pouvait pas paramétrer la requête et nous ne pouvions compter que sur l'appel statique de `:conditions` pour définir l'état de satisfaction du lecteur comme étant `happy`.

Une alternative serait d'ajouter dynamiquement, dans le code utilisant notre modèle, ses propres conditions à la requête.

```
user = User.find(some_id)
user.articles.find(:all, :conditions => [ 'rating >= ?', 3])
```

Cela fonctionne, mais casse sensiblement l'encapsulation. Nous voudrions vraiment conserver l'idée de pouvoir trouver les articles en se basant sur leur notation au sein de l'association des articles elle-même. Rails nous le permet par l'addition d'un bloc à chaque déclaration `has_many`. Toutes les méthodes définies dans ce bloc deviendront des méthodes de l'association elle-même.

Le code qui suit ajoute la méthode finder `rated_at_or_above` à l'association `articles` du modèle `user`.

```
class User < ActiveRecord::Base
  has_many :readings
  has_many :articles, :through => :readings do
    def rated_at_or_above(rating)
      find :all, :conditions => ['rating >= ?', rating]
    end
  end
end
```

Étant donné l'objet de modèle `user`, nous pouvons maintenant appeler cette méthode pour retrouver la liste des articles auxquels les lecteurs ont attribué de fortes notes.

```
user = User.find(some_id)
good_articles = user.articles.rated_at_or_above(4)
```

Bien que nous ayons illustré ici cette pratique par l'option `:through` de `has_many`, cette capacité à étendre une association avec vos propres méthodes s'appliquent à toutes les déclarations d'association.

Partager les extensions d'association

Vous aurez parfois besoin d'appliquer le même jeu d'extensions à plusieurs associations. Vous pouvez réaliser ceci en plaçant vos méthodes d'extension dans un module Ruby et en passant ce module aux déclarations associatives à l'aide du paramètre `:extend`.

```
has_many :articles, :extend => RatingFinder
```

Vous pouvez de plus étendre une association par de multiples modules en passant un tableau au paramètre `:extend`.

```
has_many :articles, :extend => [ RatingFinder, DateRangeFinder ]
```

Jointure de tables multiples

Les bases de données relationnelles nous permettent d'établir des jointures entre des tables : par exemple, un enregistrement de notre table `orders` est associé avec nombre d'enregistrements de la table `line_items`. Cette relation est définie statiquement. Cependant, c'est parfois peu pratique.

Vous pourriez contourner le problème par un code astucieux, mais par chance vous n'aurez pas à le faire. Rails fournit deux mécanismes pour l'habillage d'un modèle relationnel dans un modèle plus complexe orienté objet : *l'héritage à une table* et les *associations polymorphiques*. Nous les examinerons tour à tour.

Héritage à une table

Quand nous programmons avec des objets et des classes, nous utilisons parfois la notion d'héritage pour exprimer une relation entre nos différentes abstractions. Notre application peut ainsi avoir affaire à des personnes aux rôles variés : clients, employés, dirigeants, etc. Tous ces rôles ont des caractéristiques en commun et d'autres qui sont spécifiques à chacun d'eux. On peut modéliser cela en disant que la classe `Employee` et la classe `Customer` sont deux sous-classes de la classe `Person` et que `Manager` est elle-même une sous-classe de `Employee`. Les sous-classes *héritent* des propriétés et des responsabilités de leur classe parente[1].

Dans le monde des bases de données relationnelles, le concept d'héritage n'existe pas : les relations sont principalement exprimées en termes d'associations. Néanmoins, nous pourrions avoir besoin de stocker un modèle orienté objet dans une base de données relationnelle. Il existe de nombreuses façons de mettre l'un en correspondance avec l'autre. La plus simple s'appelle l'héritage à une table. Dans ce schéma, toutes les classes liées par une relation d'héritage sont mises en correspondance dans une seule table. Cette table contient une colonne pour chacun des attributs de la hiérarchie de classes. Une colonne supplémentaire, nommée `type`, identifie la classe à laquelle appartient chaque objet stocké dans un enregistrement de la table. C'est ce qu'illustre la figure 18-2.

La mise en œuvre de l'héritage à une table dans Active Record est très simple. Il suffit de définir la hiérarchie de classes souhaitée dans vos modèles et de vous assurer que la table qui correspond à la classe de base possède bien une colonne pour chacun des attributs utilisés par

1. Bien sûr, en programmation, l'héritage est une construction largement galvaudée. Avant de suivre ce chemin, demandez-vous si vous avez réellement besoin d'une relation de type *is-a* (est un). Par exemple, un employé peut aussi être un client, ce qui est plutôt difficile à modéliser sur la base d'une arborescence d'héritage statique. Dans ces cas-là, considérez les alternatives (comme les tags ou les taxonomies basées sur les rôles).

l'ensemble des classes de la hiérarchie. La table doit aussi inclure une colonne de type, qui sera utilisée pour définir la classe de l'objet stocké dans un enregistrement.

Lors de la définition de la table, rappelez-vous que les attributs de sous-classes ne seront présents que dans les enregistrements de la table qui correspondent à ces sous-classes. Par exemple, un client n'a pas d'attribut salaire. En conséquence, vous devez autoriser la valeur vide (`null`) pour les colonnes qui ne sont pas communes à toutes les classes. Voici la migration créant la table qui est par ailleurs illustrée figure 18-2.

code/e1/ar/sti.rb

```
create_table :people, :force => true do |t|
  t.column :type, :string
  # attributs communs
  t.column :name, :string
  t.column :email, :string
  # attributs pour la classe Customer
  t.column :balance, :decimal, :precision => 10, :scale => 2
  # attributs pour la classe Employee
  t.column :reports_to, :integer
  t.column :dept,       :integer
  # attributs pour la classe Manager
  # none
end
```

Nous pouvons définir notre hiérarchie d'objets modèle comme suit :

code/e1/ar/sti.rb

```
class Person < ActiveRecord::Base
end
class Customer < Person
end
class Employee < Person
  belongs_to :boss, :class_name => "Employee", :foreign_key => :reports_to
end
class Manager < Employee
end
```

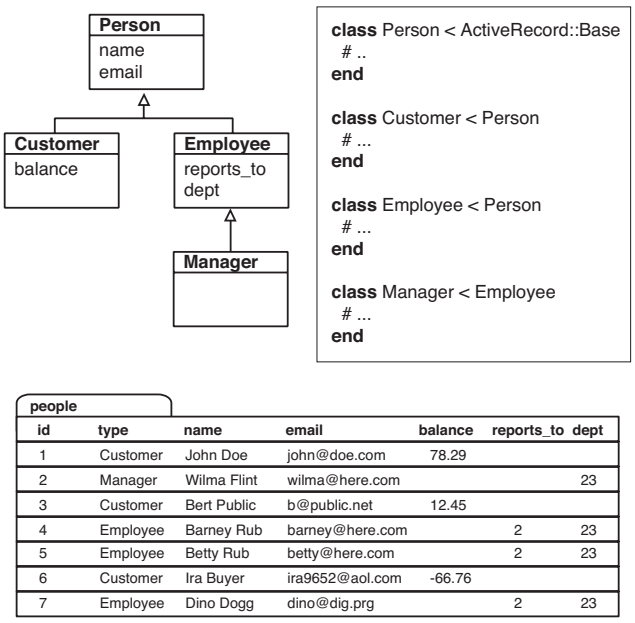


Figure 18-2
Héritage à une table : une hiérarchie de quatre classes mise en correspondance sur une seule table

people

id	type	name	email	balance	reports_to	dept
1	Customer	John Doe	john@doe.com	78.29		
2	Manager	Wilma Flint	wilma@here.com			23
3	Customer	Bert Public	b@public.net	12.45		
4	Employee	Barney Rub	barney@here.com		2	23
5	Employee	Betty Rub	betty@here.com		2	23
6	Customer	Ira Buyer	ira9652@aol.com	-66.76		
7	Employee	Dino Dogg	dino@dig.prg		2	23

Créons maintenant quelques enregistrements et relions-les.

code/e1/ar/sti.rb

```
Customer.create(:name => 'John Doe', :email => "john@doe.com",
                :balance => 78.29)

wilma = Manager.create(:name  => 'Wilma Flint', :email => "wilma@here.com",
                       :dept => 23)

Customer.create(:name => 'Bert Public', :email => "b@public.net",
                :balance => 12.45)

barney = Employee.new(:name => 'Barney Rub',  :email => "barney@here.com",
                      :dept => 23)
barney.boss = wilma
barney.save!
manager = Person.find_by_name("Wilma Flint")
puts manager.class    #=> Manager
puts manager.email    #=> wilma@here.com
puts manager.dept     #=> 23
customer = Person.find_by_name("Bert Public")
```

```
puts customer.class    #=> Customer
puts customer.email    #=> b@public.net
puts customer.balance  #=> 12.45
```

Vous noterez que nous avons demandé à la classe de base, `Person`, de trouver un enregistrement particulier et que c'est une instance de la classe `Manager` qui nous est retournée dans un cas, alors qu'il s'agit d'une instance de `Customer` dans le cas suivant. Active Record a déterminé le type de l'objet en examinant la valeur de la colonne `type` de l'enregistrement concerné.

Notez aussi la petite astuce que nous avons utilisée dans la classe `Employee`. Nous avons employé `belongs_to` pour la création d'un attribut `boss`. Cet attribut utilise la colonne `reports_to`, qui pointe sur la table `people`. C'est ce qui nous permet d'écrire `barney.boss = wilma`.

Il existe une contrainte assez évidente liée à l'héritage à une table : deux sous-classes ne peuvent avoir d'attributs portant le même nom et des types différents, parce que les deux attributs seraient mis en correspondance avec la même colonne du schéma sous-jacent.

Il en existe aussi une autre moins évidente. L'attribut `type` porte le même nom qu'une méthode du langage Ruby qui renvoie le type d'un objet. Par conséquent, accéder directement à l'attribut type de l'objet du modèle pour changer sa valeur pourrait se traduire par l'apparition de messages étranges de la part de Ruby. Il est donc recommandé d'y accéder implicitement en créant des objets de la classe ad hoc ou en utilisant la forme indexée pour accéder à l'attribut `type`. Comme ceci :

```
person[:type] = 'Manager'
```

Que se passe-t-il si je désire un héritage direct ?

L'héritage à une table est intelligent : il s'active automatiquement chaque fois que vous sous-classez une classe Active Record. Mais que se passe-t-il si vous désirez un héritage réel ? Souhaitez-vous définir un comportement qui sera partagé par un ensemble de classes Active Record par la définition d'une classe de base abstraite et d'un ensemble de sous-classes ?

La réponse consiste à définir une méthode de classe appelée `abstract_class?()` dans votre classe de base abstraite. La méthode devrait retourner `true`, ce qui a deux effets. Premièrement, Active Record ne tentera jamais de trouver une table en base de données correspondante à cette classe abstraite. Deuxièmement, toutes les sous-classes de cette classe seront considérées comme des classes Active Record indépendantes (chacune sera en correspondance avec sa propre table en base de données).

Bien entendu, la meilleure façon de parvenir à cela est probablement d'utiliser un module Ruby contenant la fonctionnalité partagée, et de mixer ce module entre les classes Active Record qui nécessitent ce comportement.

Avec l'héritage à une table, les sous-classes ne partageront-elles pas tous les attributs ?

Si, mais ce n'est pas aussi grave qu'il y paraît. Tant que les sous-tables sont assez similaires, vous pouvez ignorer sans risque l'attribut `reports_to` quand vous avez affaire à un client. Il vous suffit simplement de ne pas l'utiliser.

Ici, nous faisons clairement un compromis entre propreté du modèle objet, vitesse de traitement et facilité d'implémentation. En effet, sélectionner toutes les données à partir d'une seule table est bien plus rapide que d'avoir à faire une jointure entre les tables `people` et `customers` pour récupérer tous les attributs d'un client.

Mais l'héritage à une table n'est pas toujours la solution idéale. Il ne s'applique pas toujours très bien, comme dans des hiérarchies où les sous-classes ont peu d'attributs en commun. Par exemple, un système de gestion de contenu peut déclarer une classe de base `Content` et des sous-classes `Article`, `Image`, `Page`, etc. Et il y a toutes les chances que ces sous-classes soient très différentes, donnant ainsi naissance à une table comportant énormément de colonnes afin de couvrir tous les attributs de toutes les sous-classes. Dans cette situation, il est préférable d'utiliser les associations polymorphiques que nous allons décrire maintenant.

Associations polymorphiques

Un inconvénient majeur de l'héritage à une classe (STI pour *Single Table Inheritance*) est qu'il n'existe qu'une seule table sous-jacente contenant tous les attributs de toutes les sous-classes de notre arbre d'héritage. Nous pouvons pallier ce problème par l'emploi de la seconde forme d'agrégation hétérogène de Rails : les associations polymorphiques.

Les associations polymorphiques reposent sur le fait que les valeurs contenues dans une colonne de clés étrangères se résument à des entiers. Bien qu'il existe une convention selon laquelle une clé étrangère nommée `user_id` référence la colonne `id` de la table `users`, il n'existe pas de loi qui l'impose[1].

En informatique, le polymorphisme est un mécanisme qui permet l'abstraction essentielle de quelque chose sans considération pour l'implémentation sous-jacente, le type ou la forme (comme une interface). La méthode d'addition, par exemple, est polymorphique parce qu'elle fonctionne avec des entiers, des flottants et même des chaînes de caractères.

En Rails, une association polymorphique est une association qui lie des objets de différents types. Le postulat est que tous ces objets partagent des caractéristiques communes mais qu'ils auront des représentations différentes.

Pour rendre tout ceci un peu plus tangible, intéressons-nous à un système très simple de gestion de documents. Ceux-ci seront indexés dans un catalogue. Chaque entrée de catalogue contient un nom, la date d'acquisition, et une référence éventuelle à une ressource associée : un article, une image, un son, etc. Chacun des différents types de ressources correspond à une

1. Si vous précisez que votre base de données doit garantir les contraintes de clés étrangères, les associations polymorphiques ne fonctionneront pas.

table différente de la base de données ainsi qu'à un modèle Active Record qui lui est propre, bien que ce soient tous des documents et qu'ils soient tous catalogués.

Commençons par les trois tables contenant les trois types de ressources.

code/e1/ar/polymorphic.rb

```
create_table :articles, :force => true do |t|
  t.column :content, :text
end
create_table :sounds, :force => true do |t|
  t.column :content, :binary
end
create_table :images, :force => true do |t|
  t.column :content, :binary
end
```

Maintenant, pensons un peu aux trois modèles qui vont habiller les tables. Nous aimerions écrire quelque chose comme :

```
# CECI NE MARCHE PAS
class Article < ActiveRecord::Base
  has_one :catalog_entry
end
class Sound < ActiveRecord::Base
  has_one :catalog_entry
end
class Image < ActiveRecord::Base
  has_one :catalog_entry
end
```

Malheureusement, ceci ne peut pas fonctionner. Quand nous écrivons `has_one :catalog_entry` dans un modèle, cela signifie que la table `catalog_entries` possède une clé étrangère qui renvoie à notre table. Mais dans le cas présent, nous avons trois tables réclamant chacune de posséder une entrée dans le catalogue et nous ne pouvons probablement pas nous arranger de manière à ce que l'entrée de catalogue renvoie à ces trois tables...

... à moins d'utiliser les associations polymorphiques. L'astuce étant d'employer deux colonnes de `catalog_entry` pour la clé étrangère. La première contiendra l'identifiant de l'enregistrement cible, et la seconde indiquera à Active Record dans quel modèle de ressource se trouve cette clé. Si nous nommons `resource` la clé étrangère des entrées de catalogue, nous devrons créer deux colonnes, `resource_id` et `resource_type`. Voici la migration créant le catalogue dans sa totalité.

code/e1/ar/polymorphic.rb

```
create_table :catalog_entries, :force => true do |t|
  t.column :name, :string
  t.column :acquired_at, :datetime
  t.column :resource_id, :integer
  t.column :resource_type, :string
end
```

Nous pouvons maintenant créer le modèle Active Record d'une entrée de catalogue. Nous devons signaler à Active Record que nous sommes en train de construire une association polymorphique au travers de nos colonnes `resource_id` et `resource_type`.

code/e1/ar/polymorphic.rb

```
class CatalogEntry < ActiveRecord::Base
  belongs_to :resource, :polymorphic => true
end
```

Maintenant que la plomberie est en place, nous pouvons définir les versions finales de nos modèles Active Record des trois types de documents.

code/e1/ar/polymorphic.rb

```
class Article < ActiveRecord::Base
  has_one :catalog_entry, :as => :resource
end
class Sound < ActiveRecord::Base
  has_one :catalog_entry, :as => :resource
end
class Image < ActiveRecord::Base
  has_one :catalog_entry, :as => :resource
end
```

La clé est ici l'option `:as` de `has_one` qui spécifie que l'association entre une entrée de catalogue (`catalog_entry`) et les documents est polymorphique, et utilise l'attribut `resource` dans les entrées de catalogue. Testons ce code !

code/e1/ar/polymorphic.rb

```
a = Article.new(:content => "This is my new article")
```

```
c = CatalogEntry.new(:name => 'Article One', :acquired_at => Time.now)
c.resource = a
c.save!
```

Voyons ce qui s'est produit dans la base de données. A priori rien de spécial concernant l'article.

```
mysql> select * from articles;
+----+------------------------+
| id | content                |
+----+------------------------+
| 1  | This is my new article |
+----+------------------------+
1 row in set (0.00 sec)
```

L'entrée de catalogue possède une clé étrangère en référence à l'article et contient également le type d'objet Active Record auquel il se réfère, ici un `Article`.

```
mysql> select * from catalog_entries;
+----+-------------+---------------------+-------------+---------------+
| id | name        | acquired_at         | resource_id | resource_type |
+----+-------------+---------------------+-------------+---------------+
| 1  | Article One | 2006-07-18 16:48:29 | 1           | Article       |
+----+-------------+---------------------+-------------+---------------+
1 row in set (0.00 sec)
```

Nous pouvons accéder aux données par les deux extrémités de la relation.

code/e1/ar/polymorphic.rb

```
article = Article.find(1)
p article.catalog_entry.name  #=> "Article Un"
cat = CatalogEntry.find(1)
resource = cat.resource
p resource                    #=> #<Article:0x640d80 @attributes={"id"=>"1",
                              #     "content"=>"Voici mon nouvel article"}>
```

La partie astucieuse consiste ici dans la ligne `resource =cat.resource`. Nous récupérons la `ressource` de l'entrée du catalogue, et il nous est retourné un objet `Article`. Ce code a donc correctement déterminé la classe Active Record, lu la table appropriée en base de données (`articles`), et retourné la bonne classe d'objet.

Rendons ceci encore plus intéressant. Nettoyons d'abord notre base de données puis ajoutons-y des documents des trois types.

code/e1/ar/polymorphic.rb

```
c = CatalogEntry.new(:name => 'Article Un', :acquired_at => Time.now)
c.resource = Article.new(:content => "Voici mon nouvel article")
c.save!
c = CatalogEntry.new(:name => 'Image Une', :acquired_at => Time.now)
c.resource = Image.new(:content => "des données binaires")
c.save!
c = CatalogEntry.new(:name => 'Sound Un', :acquired_at => Time.now)
c.resource = Sound.new(:content => "des données binaires")
c.save!
```

Notre base de données est maintenant plus intéressante.

```
mysql> select * from articles;
+----+--------------------------+
| id | content                  |
+----+--------------------------+
| 1  | Voici mon nouvel article |
+----+--------------------------+
mysql> select * from images;
+----+----------------------+
| id | content              |
+----+----------------------+
| 1  | des données binaires |
+----+----------------------+
mysql> select * from sounds;
+----+----------------------+
| id | content              |
+----+----------------------+
| 1  | des données binaires |
+----+----------------------+
mysql> select * from catalog_entries;
+----+-------------+---------------------+-------------+---------------+
| id | name        | acquired_at         | resource_id | resource_type |
+----+-------------+---------------------+-------------+---------------+
| 1  | Article Un  | 2006-07-18 17:02:05 | 1           | Article       |
| 2  | Image Une   | 2006-07-18 17:02:05 | 1           | Image         |
| 3  | Son Un      | 2006-07-18 17:02:05 | 1           | Sound         |
+----+-------------+---------------------+-------------+---------------+
```

Notez comment les trois clés étrangères du catalogue possèdent chacune un identifiant de 1 : elles ne se distinguent plus que par leur colonne de type.

Nous allons maintenant retrouver les trois documents par itération sur le catalogue.

code/e1/ar/polymorphic.rb

```
CatalogEntry.find(:all).each do |c|
  puts "#{c.name}:  #{c.resource.class}"
end
```

Ce qui produit la sortie :

```
Article Un:  Article
Image Une:  Image
Son Un:  Sound
```

Figure 18-3
Méthodes créées par les déclarations d'association

	has_one :other	belongs_to :other
other(reload=false)	✓	✓
other=	✓	✓
create_other(...)	✓	✓
build_other(...)	✓	✓
replace	✓	✓
updated?		✓

	has_many :others	habtm :others
others	✓	✓
others=	✓	✓
other_ids=	✓	✓
others.<<	✓	✓
others.build(...)	✓	✓
others.clear(...)	✓	✓
others.concat(...)	✓	✓
others.count	✓	✓
others.create(...)	✓	✓
others.delete(...)	✓	✓
others.delete_all	✓	✓
others.destroy_all	✓	✓
others.empty?	✓	✓
others.find(...)	✓	✓
others.length	✓	✓
others.push(...)	✓	✓
others.replace(...)	✓	✓
others.reset	✓	✓
others.size	✓	✓
others.sum(...)	✓	✓
others.to_ary	✓	✓
others.uniq	✓	✓
push_with_attributes(...)		✓ [obsolète]

Jointure autoréférentielle

Il est possible pour un enregistrement d'une table d'en référencer un autre situé dans la même table. Par exemple, tous les employés dans une société peuvent avoir à la fois un supérieur hiérarchique et un mentor, tous deux également employés. Vous pouvez modéliser cela avec Rails en utilisant une classe `Employee` comme suit.

code/e1/ar/self_association.rb

```
class Employee < ActiveRecord::Base
  belongs_to :manager,
             :class_name  => "Employee",
             :foreign_key => "manager_id"
  belongs_to :mentor,
             :class_name  => "Employee",
             :foreign_key => "mentor_id"

  has_many   :mentored_employees,
             :class_name  => "Employee",
             :foreign_key => "mentor_id"
  has_many   :managed_employees,
             :class_name  => "Employee",
             :foreign_key => "manager_id"
end
```

Chargeons quelques données. Clem et Dawn ont chacun un supérieur et un mentor.

code/e1/ar/self_association.rb

```
Employee.delete_all
adam = Employee.create(:id => 1, :name => "Adam")
beth = Employee.create(:id => 2, :name => "Beth")
clem = Employee.new(:name => "Clem")
clem.manager = adam
clem.mentor  = beth
clem.save!
dawn = Employee.new(:name => "Dawn")
dawn.manager = adam
dawn.mentor  = clem
dawn.save!
```

Nous pouvons maintenant parcourir la relation et répondre à des questions comme qui est le mentor de X ? ou quels sont les subordonnés de Y ?

code/e1/ar/self_association.rb

```
p adam.managed_employees.map {|e| e.name}  # => [ "Clem", "Dawn" ]
p adam.mentored_employees                  # => []
p dawn.mentor.name                         # => "Clem"
```

Il est sans doute opportun de passer aussi en revue les relations *acts as*.

Acts As (agit comme)

Nous avons vu comment `has_one`, `has_many` et `has_and_belongs_to_many` nous permettent de représenter les associations typiques des bases de données relationnelles, telles que les associations un-vers-un, un-vers-N et N-vers-N. Il arrive aussi que nous ayons besoin de bâtir des structures plus avancées sur la base de ces relations simples.

Supposons qu'une commande comporte plusieurs items à facturer. Jusqu'à présent, nous avons utilisé `has_many` avec succès pour représenter ce type de relation. Mais la richesse de notre application allant croissante, il est possible que nous ayons à ajouter de nouveaux comportements à cette liste d'items, comme les placer dans un certain ordre ou déplacer un item d'un endroit à un autre dans la liste.

Nous souhaitons peut-être gérer les catégories de notre produit dans une structure de données arborescente où les catégories peuvent posséder des sous-catégories, qui elles-mêmes en possèdent d'autres, etc.

Active Record fournit justement ce genre de fonctionnalités en standard en s'appuyant lui-même sur les relations `has_`. On appelle ces nouvelles relations *acts as* (agit comme), car elles font en sorte que les objets des modèles se comportent comme quelque chose d'autre[1].

Acts As List (agit comme une liste)

Utilisez la déclaration `acts_as_list` dans un modèle enfant pour lui donner un comportement identique à celui d'une liste. Le modèle parent pourra alors parcourir les enfants l'un après l'autre, déplacer un objet enfant dans la liste ou l'enlever de la liste.

Les listes sont implémentées en assignant à chaque enfant un numéro de rang. Cela signifie que la table fille doit posséder une colonne pour garder trace de ce rang. Si nous appelons cette colonne `position`, Rails l'utilisera automatiquement, si le nom est différent, il faut le signaler à Rails. C'est ce que nous écrivons dans l'exemple qui suit, basé sur une nouvelle table fille (appelée `children`) et une table parent.

1. Rails est livré avec les extensions **`acts_as`** suivantes : **`acts_as_list`** (se comporte comme une liste), **`acts_as_tree`** (agit comme un arbre), et **`acts_as_nested_set`** (agit comme des ensembles imbriqués). J'ai choisi de documenter les deux premières car, juste avant la sortie du livre, de sérieuses anomalies sont apparues dans la variante ensemble imbriqué qui nous ont empêché de faire fonctionner notre code d'exemple.

code/e1/ar/acts_as_list.rb

```
create_table :parents, :force => true do |t|
end
create_table :children, :force => true do |t|
  t.column :parent_id, :integer
  t.column :name,      :string
  t.column :position,  :integer
end
```

Ensuite, nous allons écrire les classes des modèles. Notez que dans la classe `Parent` nous choisissons d'ordonner les objets enfants selon les valeurs de la colonne `position`. De cette façon nous sommes sûrs que le tableau retourné est dans le bon ordre.

code/e1/ar/acts_as_list.rb

```
class Parent < ActiveRecord::Base
  has_many :children, :order => :position
end
class Child < ActiveRecord::Base
  belongs_to :parent
  acts_as_list  :scope => :parent_id
end
```

Dans la classe `Child`, nous voyons la déclaration classique `belongs_to`, qui établit la connexion avec le modèle parent. Nous avons aussi une déclaration `acts_as_list` assortie d'une option `:scope`, qui indique à Rails que nous voulons une liste par parent. Sans cette option, Rails gérerait uniquement une liste globale pour toutes les entrées de la table enfant.

Mettons quelques données de test en place : quatre enfants, que nous appellerons un, deux, trois et quatre, sont créés pour un parent.

code/e1/ar/acts_as_list.rb

```
parent = Parent.new
%w{ One Two Three Four}.each do |name|
  parent.children.create(:name => name)
end
parent.save
```

écrivons une méthode pour examiner le contenu de la liste. Il y a une subtilité ici : notez que nous passons le paramètre `true` à l'association `children`, ce qui la forcera à se recharger à chaque fois que nous y accéderons. Nous effectuons cette opération car les diverses

méthodes `move_` mettent bien à jour les enregistrements fils dans la base de données, mais elles n'opèrent directement que sur les enfants. Les parents n'auront pas connaissance des modifications immédiatement. Le rechargement les force donc à se rafraîchir la mémoire.

code/e1/ar/acts_as_list.rb

```
def display_children(parent)
  puts parent.children(true).map {|child| child.name }.join(", ")
end
```

Et pour finir, manipulons cette liste. Les commentaires indiquent l'affichage attendu à l'exécution de `display_children()`.

code/e1/ar/acts_as_list.rb

```
display_children(parent)         #=> One, Two, Three, Four
puts parent.children[0].first?   #=> true
two = parent.children[1]
puts two.lower_item.name         #=> Three
puts two.higher_item.name        #=> One
parent.children[0].move_lower
display_children(parent)         #=> Two, One, Three, Four
parent.children[2].move_to_top
display_children(parent)         #=> Three, Two, One, Four
parent.children[2].destroy
display_children(parent)         #=> Three, Two, Four
```

La bibliothèque de manipulation des listes utilise la terminologie *lower* et *higher* pour désigner les positions relatives des éléments. Higher signifie plus près de la tête de liste et lower, plus près de la fin. Les méthodes `move_higher()`, `move_lower()`, `move_to_bottom()` et `move_to_top()` déplacent un élément particulier de la liste en ajustant automatiquement la position des autres éléments.

`higher_item()` et `lower_item()` renvoient respectivement les éléments suivant et précédant l'élément courant. `first?()` et `last?()` renvoient `true` si l'élément est respectivement en tête ou à la fin de la liste.

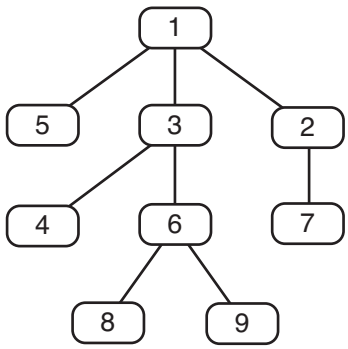
Les nouveaux enfants sont automatiquement ajoutés à la fin de la liste. Quand un enregistrement enfant est détruit, les enfants qui suivent sont déplacés d'un rang vers le haut pour remplir l'emplacement vide.

Acts As Tree (agit comme un arbre)

Active Record permet d'organiser les enregistrements d'une table en une structure hiérarchique ou un arbre. C'est très utile pour gérer des structures où les entrées possèdent des sous-entrées, les sous-entrées ayant elles-mêmes des sous-entrées, etc. Les listes de catégories ont souvent cette structure, tout comme les répertoires, les descriptions des permissions, etc.

Cette structure peut être activée en ajoutant une simple colonne (appelée `parent_id` par défaut) à la table. Cette colonne doit être une clé étrangère sur la même table afin de relier les nœuds de l'arbre à leur nœud parent. C'est ce qu'illustre la figure 18-4.

Figure 18-4
Représenter un arbre dans une table en utilisant des liens vers les nœuds parents



catégories

id	parent_id	rest_of_data
1	null	...
2	1	...
3	1	...
4	3	...
5	1	...
6	3	...
7	2	...
8	6	...
9	6	...

Pour illustrer le fonctionnement des arbres, créons une table de catégories, où chaque catégorie de plus haut niveau peut avoir des sous-catégories et chaque sous-catégorie peut elle-même posséder des sous-catégories. Remarquez la clé étrangère qui pointe sur sa propre table.

code/e1/ar/acts_as_tree.rb

```
create_table :categories, :force => true do |t|
  t.column :name, :string
  t.column :parent_id, :integer
end
```

Le modèle correspondant utilise la méthode `acts_as_tree` pour spécifier le comportement que doit adopter la structure. Le paramètre `:order` spécifie que lorsque nous parcourons les catégories filles d'un nœud particulier, elles sont rangées par ordre alphabétique sur la base de la colonne `name`.

code/e1/ar/acts_as_tree.rb

```
class Category < ActiveRecord::Base
  acts_as_tree  :order => "name"
end
```

Normalement, nous devrions disposer de quelques fonctionnalités destinées à l'utilisateur final afin de créer et de modifier la hiérarchie des catégories. Dans notre exemple, nous les créerons juste en écrivant du code. Notez tout de même comment nous manipulons les enfants de n'importe quel nœud en utilisant les attributs enfants.

code/e1/ar/acts_as_tree.rb

```
root        = Category.create(:name => "Books")
fiction     = root.children.create(:name => "Fiction")
non_fiction = root.children.create(:name => "Non Fiction")
non_fiction.children.create(:name => "Computers")
non_fiction.children.create(:name => "Science")
non_fiction.children.create(:name => "Art History")
fiction.children.create(:name => "Mystery")
fiction.children.create(:name => "Romance")
fiction.children.create(:name => "Science Fiction")
```

Maintenant que tout est en place, nous pouvons manipuler la structure arborescente. La même méthode `display_children()` que pour `acts as list` est appelée pour vérifier que les changements demandés ont bien été effectués.

code/e1/ar/acts_as_tree.rb

```
display_children(root)             # Fiction, Non Fiction
sub_category = root.children.first
puts sub_category.children.size    #=> 3
display_children(sub_category)     #=> Mystery, Romance, Science Fiction
non_fiction = root.children.find(:first, :conditions => "name = 'Non Fiction'")
display_children(non_fiction)      #=> Art History, Computers, Science
puts non_fiction.parent.name       #=> Books
```

Les diverses méthodes utilisées pour la manipulation de l'arbre vous sembleront familières : elles portent les mêmes noms que celles fournies par `has_many`. En fait, si vous avez le courage d'aller voir dans le code de Rails la façon dont `acts_as_tree` est implémentée, vous verrez que Rails ne fait qu'utiliser deux déclarations `belongs_to` et `has_many`, chacune pointant sur la même table. C'est exactement comme si on avait écrit :

```
class Category < ActiveRecord::Base
        belongs_to :parent,
                   :class_name  => "Category"
        has_many   :children,
                   :class_name  => "Category",
                   :foreign_key => "parent_id",
       :order       => "name",
       :dependent   => :destroy
end
```

Si vous avez besoin d'optimiser les performances de `children.size`, vous pouvez mettre en place un cache de compteur (exactement comme pour `has_many`). Il suffit d'ajouter l'option `:counter_cache => true` à la déclaration `acts_as_tree` et d'ajouter une colonne `categories_count` à la table.

Quand les objets sont sauvés

Revenons à nos tables `invoices` et `orders`.

code/e1/ar/one_to_one.rb

```
class Order < ActiveRecord::Base
  has_one :invoice
end
class Invoice < ActiveRecord::Base
  belongs_to :order
end
```

Il est possible d'associer une facture avec une commande par chaque extrémité de la relation : vous pouvez informer une commande qu'une facture lui est associée, et vous pouvez également avertir la facture qu'elle est associée à une commande. Les deux manières d'exprimer la relation sont presque équivalentes. La différence est dans la façon dont elles sauvent (ou ne sauvent pas) les objets dans la base de données. Si vous assignez un objet à une association `has_one` dans un objet existant, cet objet associé sera automatiquement sauvegardé.

```
order = Order.find(some_id)
an_invoice = Invoice.new(...)
order.invoice = an_invoice      # invoice est sauvegardé
```

Pourquoi les objets sont-ils sauvés ainsi dans les associations ?

Il peut paraître inconsistant que l'assignation d'une commande à une facture ne sauvegarde pas immédiatement l'association, alors que l'assignation inverse le fait. C'est en fait parce que la table `invoices` est la seule à détenir l'information sur l'état de cette relation, et partant de là, ce sont toujours les enregistrements `invoice` qui contiennent l'information. Lors de l'assignation d'une commande à une facture, vous pouvez facilement intégrer cette sauvegarde comme partie d'une mise à jour plus étendue des enregistrements `invoice`, qui pourrait également inclure la date de facturation, par exemple. C'est l'occasion de grouper sur une seule opération ce qui aurait demandé autrement deux mises à jour de la base de données. En ORM, la règle consiste généralement à limiter au maximum les appels à la base de données.

Quand un objet `order` se voit assigner une facture, il doit quand même effectuer une mise à jour de l'enregistrement `invoice`. Ainsi, il n'y a aucun bénéfice supplémentaire à retarder cette association jusqu'à ce que l'objet `order` soit sauvegardé. En fait, cela serait même très gourmand en code alors que Rails est entièrement tourné vers l'économie de code.

Si, en revanche, vous assignez un nouvel objet a une association `belongs_to`, il ne sera jamais sauvé automatiquement.

```
order = Order.new(...)
an_invoice.order = order    # order ne sera pas sauvé ici
an_invoice.save             # invoice et order sont sauvegardés
```

Pour finir, il y a un autre danger. Si l'enregistrement fils ne peut pas être sauvegardé (par exemple, parce que la validation aurait échoué), Active Record ne se plaindra pas : vous n'obtiendrez aucune indication sur le fait que l'enregistrement n'a pas été ajouté à la base de données. Pour cette raison, nous vous recommandons fortement d'utiliser le code suivant :

```
invoice = Invoice.new
# remplir la facture
invoice.save!
an_order.invoice = invoice
```

La méthode `save!()` lève une exception en cas d'échec, ainsi vous serez au moins mis au courant que quelque chose s'est mal passé pendant l'opération.

Sauvegarde et collections

Les règles concernant la sauvegarde des objets qui impliquent des collections (c'est-à-dire lorsque vous travaillez avec un modèle contenant une déclaration `has_many()` ou `has_and_belongs_to_many()`) sont typiquement les mêmes.

- Si l'objet parent existe en base de données, alors, l'ajout d'un objet enfant à une collection sauvegarde automatiquement cet enfant. Si le parent n'existe pas en base de données, alors l'enfant est maintenu en mémoire et sera sauvegardé quand le parent aura lui-même été sauvé.
- Si la sauvegarde d'un objet enfant échoue, la méthode utilisée pour ajouter cet enfant à la collection retourne `false`.

Comme pour `has_one`, assigner un objet du côté `belongs_to` d'une association n'entraîne pas sa sauvegarde.

Précharger des enregistrements fils

Normalement, Active Record diffère le chargement des enregistrements fils depuis la base de données jusqu'à ce que vous les référenciez. Ainsi, en nous inspirant de l'exemple fourni dans le document RDoc de Rails, supposons qu'une application de blog possède le modèle suivant :

```
class Post < ActiveRecord::Base
  belongs_to :author
  has_many   :comments, :order => 'created_on DESC'
end
```

Si nous itérons sur les messages postés sur le blog, en accédant à la fois à l'auteur du message et au commentaire, nous utiliserons une requête SQL pour retourner les n enregistrements de la table `posts`, puis n autres requêtes pour récupérer les enregistrements des tables `authors` et `comments`, soit un total de 2n+1 requêtes. Nous avons là, à coup sûr, un problème de performance en perspective...

```
for post in Post.find(:all)
  puts "Post:            #{post.title}"
  puts "Written by:      #{post.author.name}"
  puts "Last comment on: #{post.comments.first.created_on}"
end
```

Ce problème de performance peut parfois être résolu en utilisant l'option `:include` de la méthode `find()`. Cette option permet de spécifier les associations qui doivent être préchargées quand la méthode `find` est invoquée. Active Record effectue cette tâche de façon plutôt intelligente et extrait en une seule requête SQL les données de la table principale et des tables associées. Si nous avons 100 messages postés sur notre blog, le code suivant va éliminer 100 requêtes SQL par rapport à l'exemple précédent.

```
for post in Post.find(:all, :include => :author)
  puts "Post:            #{post.title}"
```

```
  puts "Written by:      #{post.author.name}"
  puts "Last comment on: #{post.comments.first.created_on}"
end
```

Et l'exemple qui suit ramènera toutes les données en une seule requête !

```
for post in Post.find(:all, :include => [:author, :comments])
  puts "Post:            #{post.title}"
  puts "Written by:      #{post.author.name}"
  puts "Last comment on: #{post.comments.first.created_on}"
end
```

Ce préchargement n'améliore pas les performances à chaque fois[1]. Sous le capot, le moteur de base de données opère une jointure sur toutes les tables concernées et peut donc finir par retourner beaucoup de données à convertir en objet Active Record. Et si votre application n'a pas besoin de toutes ces informations supplémentaires fournies par la jointure, vous n'y gagnerez peut-être rien. Vous pourriez aussi rencontrer des difficultés si la table parente contient énormément d'enregistrements. Dans ce cas, en comparaison du chargement individuel des données, la technique de préchargement consommera beaucoup plus de mémoire vive sur le serveur.

Si vous utilisez `:include`, vous devrez lever l'ambiguïté sur le nom des colonnes utilisées dans les autres paramètres de `find()` en les préfixant avec le nom de la table où ils se trouvent. Dans l'exemple suivant, la colonne `title`, mentionnée dans le paramètre de condition, doit être préfixée par le nom de table pour que la requête aboutisse.

```
for post in Post.find(:all, :conditions => "posts.title like '%ruby%'",
                      :include => [:author, :comments])
  #
...
end
```

Compteurs

La relation `has_many` définit un attribut contenant une collection d'objets. Il paraît raisonnable de pouvoir obtenir la taille de cette collection : combien y a-t-il d'items dans cette commande ? Et, de fait, un objet résultant d'une agrégation offre une méthode `size()` qui retourne précisément cette information. Cette méthode exécute une requête `select count(*)` sur la table fille, en comptant le nombre d'enregistrements dont la clé étrangère référence l'enregistrement concerné de la table parente.

1. En fait, il peut même ne pas fonctionner du tout ! Si votre base de données ne supporte pas les jointures externes à gauche (de l'anglais *left outer join*), vous ne pouvez pas utiliser cette possibilité. Les utilisateurs de Oracle 8, par exemple, devront passer à la version 9 pour bénéficier de cette fonction.

Cela fonctionne et c'est fiable. Toutefois, si vous écrivez une application qui compte fréquemment les enregistrements fils, il peut être très appréciable d'éviter la surcharge induite par ces multiples requêtes SQL. Active Record peut vous prêter main forte en utilisant la technique appelée *cache des compteurs*. Dans la déclaration `belongs_to` du modèle fils, vous pouvez demander à Active Record de maintenir le compte du nombre d'enregistrements fils associés aux enregistrements de la table parente. Ce compte sera alors maintenu automatiquement. Si vous ajoutez un enregistrement fils, le compteur est incrémenté, et si vous le détruisez, le compteur est décrémenté.

Pour activer cette fonctionnalité, vous devez procéder en deux étapes simples. D'abord, ajoutez l'option `:counter_cache` à la déclaration `belongs_to` de la table fille.

code/e1/ar/counters.rb

```
class LineItem < ActiveRecord::Base
  belongs_to :product, :counter_cache => true
end
```

Ensuite, dans la définition de la table parente (`products` dans cet exemple), vous devez ajouter une colonne de type `integer`, dont le nom est le même que la table fille suivi de `_count`.

code/e1/ar/counters.rb

```
create_table :products, :force => true do |t|
  t.column :title, :string
  t.column :description, :text
  # ...
  t.column :line_items_count, :integer, :default => 0
end
```

Il existe un point important dans cette définition de la table : la colonne doit être déclarée avec une valeur par défaut de zéro (ou alors vous devez la mettre à zéro vous-même dans le code de votre application quand l'enregistrement parent est créé). Si vous ne le faites pas, vous obtiendrez toujours une valeur de type `null` pour le compteur, quel que soit le nombre effectif d'enregistrements concernés.

Si vous suivez les deux étapes précédentes, vous verrez le compteur des enregistrements de la table parente suivre automatiquement le nombre d'enregistrements dans la table fille.

Il y a néanmoins un problème potentiel avec le cache des compteurs. Le compteur est maintenu par l'objet qui contient la collection et il est mis à jour automatiquement, uniquement si les nouvelles associations sont ajoutées via cet objet. Si vous créez de nouvelles associations directement au niveau de la table fille, le compteur ne sera pas mis à jour.

L'exemple qui suit montre précisément ce qu'il ne faut pas faire. Ici, les enregistrements fils sont liés aux parents à la main. Vous noterez que l'attribut `size()` (la taille de la collection) est faux tant que le rechargement de la collection n'a pas été forcé.

code/e1/ar/counters.rb

```
product = Product.create(:title => "Programming Ruby",
                         :description => " ... ")
line_item = LineItem.new
line_item.product = product
line_item.save
puts "In memory size = #{product.line_items.size}"             #=> 0
puts "Refreshed size = #{product.line_items(:refresh).size}"   #=> 1
```

La bonne approche consiste à ajouter l'enfant via le parent :

code/e1/ar/counters.rb

```
product = Product.create(:title => "Programming Ruby",
                         :description => " ... ")
product.line_items.create
puts "In memory size = #{product.line_items.size}"             #=> 1
puts "Refreshed size = #{product.line_items(:refresh).size}"   #=> 1
```