Ruby on Rails

Dave Thomas
David Heinemeier Hansson

© Groupe Eyrolles, 2006,

ISBN : 2-212-11746-9.

EYROLLES

15

Active Record en profondeur

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]

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.

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 faisons dans l'exemple qui suit, basé sur une nouvelle table fille (appelée `children`) et une table parent.

Fichier 15.1

```
create table parents (
  id              int           not null auto_increment,
  primary key (id)
);
create table children (
  id              int           not null auto_increment,
  parent_id       int            not null,
  name            varchar(20),
  position        int,
  constraint fk_parent   foreign key (parent_id) references parents(id),
  primary key (id)
);
```

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 sûr que le tableau retourné est dans le bon ordre.

Fichier 15.2

```
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.

© 2006 Groupe Eyrolles

Fichier 15.2

```
parent = Parent.new
%w{ Un Deux Trois Quatre}.each do |name|
  parent.children.create(:name => name)
end
parent.save
```

Écrivons une méthode toute simple pour examiner le contenu de la liste.

Fichier 15.2

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

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

Fichier 15.2

```
display_children(parent)         #=> Un, Deux, Trois, Quatre
puts parent.children[0].first?   #=> true
two = parent.children[1]
puts two.lower_item.name         #=> Trois
puts two.higher_item.name        #=> Un
parent.children[0].move_lower
parent.reload
display_children(parent)         #=> Deux, Un, Trois, Quatre
parent.children[2].move_to_top
parent.reload
display_children(parent)         #=> Trois, Deux, Un, Quatre
parent.children[2].destroy
parent.reload
display_children(parent)         #=> Trois, Deux, Quatre
```

Vous remarquerez que nous avons dû appeler la méthode `reload()` sur l'objet parent. Les divers appels aux méthodes `move_xxx` modifient les enfants dans la base de données, mais comme ces méthodes opèrent directement sur les objets enfants, l'objet parent n'aura pas connaissance des changements.

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.

© 2006 Groupe Eyrolles

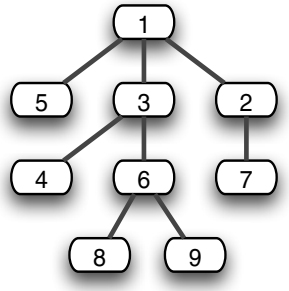
Les nouveaux enfants sont automatiquement ajoutés à la fin de liste. Quand un enregistrement enfant est détruit, les enfants qui suivent sont déplacés d'un rang vers le haut pour remplir l'emplacement vide.

Acts As Tree (agit comme un arbre)

Active Record permet d'organiser les enregistrements d'une table en une structure hiérarchique ou 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 15-1.

Figure 15-1
Représenter un arbre dans une table en utilisant des liens vers les nœuds parents



catégories

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

Pour montrer 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.

Fichier 15.1

```
create table categories (
  id              int           not null auto_increment,
  name            varchar(100)  not null,
  parent_id       int,
```

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`.

© 2006 Groupe Eyrolles

Fichier 15.3

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

Le but recherché avec ces structures (liste, arbre, etc.) est souvent de pouvoir les présenter à l'utilisateur selon la même forme. Et c'est très logiquement que vous allez offrir à vos utilisateurs la possibilité de manipuler les objets de la structure, comme déplacer des catégories dans l'arbre. Dans le code cela se traduit par des appels très simples basés sur l'attribut `children`.

Fichier 15.3

```
root        = Category.create(:name => "Livres")
fiction     = root.children.create(:name => "Fiction")
non_fiction = root.children.create(:name => "Non Fiction")
non_fiction.children.create(:name => "Informatique")
non_fiction.children.create(:name => "Science")
non_fiction.children.create(:name => "Histoire de l'art")
fiction.children.create(:name => "Polar")
fiction.children.create(:name => "Roman")
fiction.children.create(:name => "Science Fiction")
```

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

Fichier 15.3

```
display_children(root)             # Fiction, Non Fiction
sub_category = root.children.first
puts sub_category.children.size    #=> 3
display_children(sub_category)     #=> Polar, Roman, Science Fiction
non_fiction = root.children.find(:first, :conditions => "name = 'Non Fiction'")
display_children(non_fiction)      #=> Histoire de l'art, Informatique, Science
puts non_fiction.parent.name       #=> Livres
```

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",
```

© 2006 Groupe Eyrolles

```
            :foreign_key => "parent_id",
            :order       => "name",
            :dependent   => true
end
```

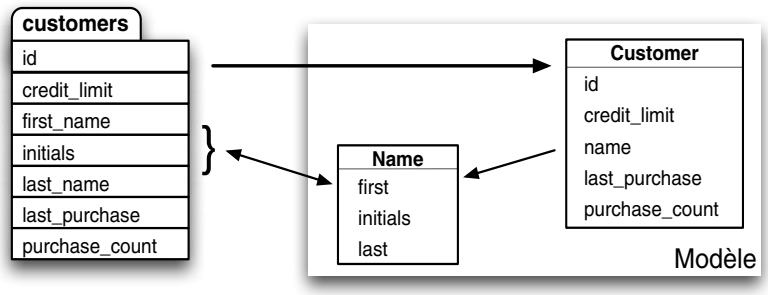
Si vous avez besoin d'optimiser les performances de `children.size`, vous pouvez mettre en place un cache de compteur (exactement comme pour `has_many`). Il suffit d'ajouter l'option `:counter_cache => true` à la déclaration `acts_as_tree` et d'ajouter une colonne `children_count` à la table.

Agrégation

Les colonnes d'une base de données ont un nombre limité de type : entiers, chaînes de caractères, dates, etc. Nos application sont, elles, beaucoup plus riches : nous définissons des classes pour améliorer le niveau d'abstraction de notre code. Il serait vraiment très intéressant de pouvoir utiliser les colonnes de la base de données pour représenter nos abstractions de haut niveau de la même manière que nous encapsulons les données des enregistrements dans les objets d'un modèle.

Ainsi, une table contenant des données client fournit entre autres des colonnes qui concernent le nom du client (prénom, nom, initiales et peut-être un surnom). Dans notre programme nous aimerions pouvoir encapsuler toutes les colonnes relatives à l'identité du client (et uniquement celles-là) dans une classe `Name` ; les trois colonnes seraient ainsi mises en correspondance avec un objet Ruby, contenu dans le modèle client comme les autres colonnes. Et bien sûr lorsque nous sauvegardons un enregistrement client au complet, nous voudrions que les données concernant le nom du client soient automatiquement extraites de l'objet `Name` et sauvegardées dans la base de données elles aussi.

Figure 15-2



C'est ce qu'on appelle l'*agrégation* (ou bien pour certains, la *composition,* selon que vous regardez le processus de haut en bas ou de bas en haut). Et, bien sûr, Rails vous permet de faire

© 2006 Groupe Eyrolles

cela très facilement. Il suffit de définir une classe pour vos données agrégées et d'ajouter une déclaration à la classe du modèle indiquant quelles colonnes de la base de données doivent être mises en correspondance avec la classe.

La classe en charge des données agrégées (ici la classe `Name`) doit satisfaire deux critères. Tout d'abord, elle doit offrir un constructeur qui accepte les données telles qu'elles apparaissent dans les colonnes de la base de données, un paramètre par colonne. Deuxièmement, elle doit fournir des attributs, un par colonne, qui renvoient les valeurs de ces colonnes telles qu'elles apparaissent dans la base de données. Tant que vous respectez ces deux impératifs, vous êtes libres de stocker les données comme vous le souhaitez en interne dans votre classe.

Dans notre exemple de nom de client, nous allons définir une classe qui contient les trois composantes du nom comme des variables d'instance ainsi qu'une méthode `to_s()` qui transforme les informations en une chaîne de caractères contenant le nom complet du client.

Fichier 15.4

```
class Name
  attr_reader :first, :initials, :last
  def initialize(first, initials, last)
    @first = first
    @initials = initials
    @last = last
  end
  def to_s
    [ @first, @initials, @last ].compact.join(" ")
  end
end
```

Nous devons maintenant informer notre classe de modèle `Customer` que les trois colonnes `first_name`, `initials` et `last_name` doivent être mises en correspondance avec les objets `Name`. C'est ce que fait la déclaration `composed_of`.

Bien que `composed_of` puisse être appelée avec un seul paramètre, il est plus utile de commencer par la description de la forme complète de la déclaration et de montrer quelles sont les valeurs par défaut des paramètres non spécifiés.

`composed_of` *:attr_name*, `:class_name =>` *SomeClass*, `:mapping =>` *mapping*

Le paramètre *attr_name* spécifie le nom donné à l'attribut composite dans la classe modèle. Si la classe client est définie comme suit :

```
class Customer < ActiveRecord::Base
  composed_of :name, ...
end
```

© 2006 Groupe Eyrolles

nous pouvons accéder à l'attribut composite en utilisant l'attribut `name` de l'objet de type `Customer`.

```
customer = Customer.find(123)
puts customer.name.first
```

L'option `:class_name` spécifie le nom de la classe qui contient les données composites. La valeur peut être soit le nom d'une classe ou bien une chaîne ou un symbole du nom de classe. Dans notre cas, la classe en question s'appelle `Name`, et le code ressemble donc à :

```
class Customer < ActiveRecord::Base
  composed_of :name, :class_name => Name, ...
end
```

Si le nom de la classe est simplement la forme à casse mixte du nom d'attribut (comme dans notre exemple), il peut être omis.

Le paramètre `:mapping` indique à Active Record la correspondance entre les colonnes de la table et les paramètres du constructeur de l'objet composite. Le paramètre `:mapping` est soit un tableau à deux éléments ou un tableau de tableaux à deux éléments. Le premier élément de ces tableaux à deux éléments est le nom de la colonne et le second, le nom de la méthode d'accès de l'attribut composite correspondant. L'ordre dans lequel apparaissent les éléments dans le paramètre de mise en correspondance définit l'ordre dans lequel le contenu des colonnes de la base de données est passé en paramètre au constructeur de la classe composite (la méthode `initialize()`). La figure 15-3 montre le fonctionnement du paramètre de mise en correspondance. Si cette option est omise, Active Record en déduit que les colonnes et les attributs de l'objet composite portent les mêmes noms que les attributs du modèle.

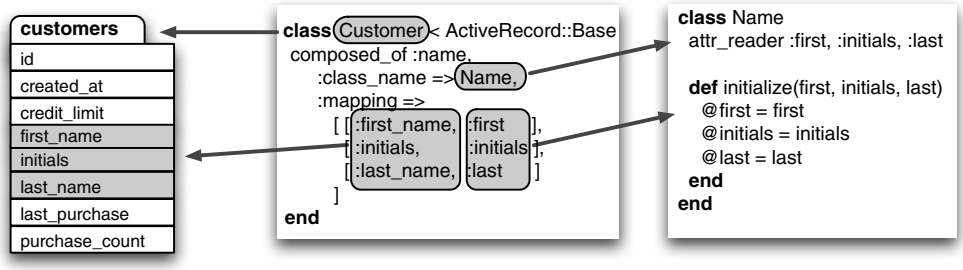


Figure 15-3
Liens de correspondance des attributs composites, les tables et classes

© 2006 Groupe Eyrolles

En ce qui concerne notre classe `Name`, nous devons faire correspondre 3 colonnes dans l'objet composite. La définition de la table `customers` est la suivante :

Fichier 15.1

```
create table customers (
  id              int           not null auto_increment,
  created_at      datetime      not null,
  credit_limit    decimal(10,2) default 100.0,
  first_name      varchar(50),
  initials        varchar(20),
  last_name       varchar(50),
  last_purchase   datetime,
  purchase_count  int           default 0,
  primary key (id)
);
```

Les colonnes `first_name`, `initials` et `last_name` doivent être mises en correspondance avec les attributs `first`, `initials` et `last` dans la classe `Name`[1]. Pour spécifier ces correspondances à Active Record nous utilisons la déclaration suivante :

Fichier 15.4

```
class Customer < ActiveRecord::Base
  composed_of :name,
              :class_name => Name,
              :mapping =>
                 [ # base de données    ruby
                   [ :first_name,  :first ],
                   [ :initials,    :initials ],
                   [ :last_name,   :last ]
                 ]
end
```

La description de ces options nous a pris du temps mais dans la pratique l'effort nécessaire à la création d'une agrégation est minime. Et une fois en place, ces agrégations sont faciles à utiliser : l'attribut composite d'un objet du modèle n'est rien d'autre que l'instance de la classe composite que vous avez définie.

Fichier 15.4

```
name = Name.new("Dwight", "D", "Eisenhower")
Customer.create(:credit_limit => 1000, :name => name)
customer = Customer.find(:first)
puts customer.name.first    #=> Dwight
puts customer.name.last     #=> Eisenhower
```

1. Dans une application réelle, il est préférable que le nom des attributs soient les mêmes que les noms des colonnes. Ici l'utilisation de noms différents n'est là que pour vous aider à comprendre comment est construite l'option `:mapping`.

© 2006 Groupe Eyrolles

```
puts customer.name.to_s     #=> Dwight D Eisenhower
customer.name = Name.new("Harry", nil, "Truman")
customer.save
```

Le code qui précède crée un nouvel enregistrement dans la table `customers` avec les colonnes `first_name`, `initials` et `last_name` initialisées à partir des valeurs des attributs `first`, `initials`, et `last` du nouvel objet `Name`. Le code extrait l'enregistrement de la base de données et accède aux champs via l'objet composite. Pour finir, il modifie l'enregistrement. Vous noterez au passage qu'on ne peut pas changer les champs d'un objet composite. La seule manière de le faire est de passer un nouvel objet.

L'objet composite ne doit pas nécessairement regrouper plusieurs colonnes de la base de données. Il est souvent utile de prendre une colonne unique et de la faire correspondre à un type autre qu'un entier, un flottant, une chaîne de caractères ou une date. La représentation de sommes d'argent est un cas classique : plutôt que de maintenir cette donnée dans un objet de type nombre flottant, il est possible de créer un objet `Money` qui possède certaines propriétés (comme la valeur arrondie) dont votre application a besoin.

Dans la page 219, nous avons vu comment utiliser la déclaration `serialize` afin de stocker des données structurées dans la base de données. On peut aussi le faire en utilisant la déclaration `composed_of`. Au lieu d'utiliser YAML pour sérier les données dans une seule colonne de la base de données, on peut effectuer notre propre sériation à l'aide d'un objet composite. À titre d'exemple, revoyons la façon dont nous avions stocké les cinq dernières commandes d'un client. Auparavant nous les avions stocké dans un tableau Ruby, lui même sérié dans la base de données comme une chaîne YAML. Enveloppons maintenant l'information dans un objet et faisons en sorte que cet objet sauve ses données dans son propre format. Dans le cas présent, nous allons sauver la liste des produits d'une commande comme une chaîne de caractères composée de la suite des produits séparés par une virgule.

Nous créons tout d'abord la classe `LastFive` pour encapsuler la liste. Puisque la base de données va stocker la liste des produits sous forme d'une simple chaîne de caractères, son constructeur doit aussi accepter une chaîne de caractères en paramètre et nous aurons aussi besoin d'un attribut qui retourne le contenu comme une chaîne de caractères. En interne nous stockerons cependant la liste dans un tableau Ruby.

Fichier 15.4

```
class LastFive

  attr_reader :list
  # Prend une chaîne contenant "a,b,c" et la
  # stocke sous la forme [ 'a', 'b', 'c' ]
  def initialize(list_as_string)
    @list = list_as_string.split(/,/)
  end
```

© 2006 Groupe Eyrolles

```
  # Retourne le contenu sous la forme
  # d'une chaîne dont les fragments sont séparés
  # par des virgules
  def last_five
    @list.join(',')
  end
end
```

Nous pouvons dire que notre classe `LastFive` encapsule la colonne `last_five` de la base de données.

Fichier 15.4

```
class Purchase < ActiveRecord::Base
  composed_of :last_five
end
```

Quand nous exécutons cet exemple, nous constatons que l'attribut `last_five` contient bien un tableau de valeurs.

Fichier 15.4

```
Purchase.create(:last_five => LastFive.new("3,4,5"))
purchase = Purchase.find(:first)
puts purchase.last_five.list[1]     #=>  4
```

Les objets composites sont des objets de valeur

Un *objet de valeur* (de l'anglais *value object*) est un objet dont l'état ne peut être modifié après sa création – il est gelé. La philosophie de l'agrégation dans Active Record suit le principe des objets de valeur : les objets composites sont des objets de valeur dont l'état interne ne peut être modifié.

Active Record n'a pas toujours les moyens d'imposer cette règle. Il est toujours possible, par exemple, d'utiliser la méthode `replace()` de la classe `String` pour modifier la valeur d'un des attributs d'un objet composite. Si vous le faites, Active Record ignorera le changement au moment de la sauvegarde de l'objet du modèle.

La bonne façon de modifier la valeur des colonnes associées à un attribut composite est de lui assigner un nouvel objet composite.

```
customer = Customer.find(123)
old_name = customer.name
customer.name = Name.new(old_name.first, old_name.initials, "Smith")
customer.save
```

© 2006 Groupe Eyrolles

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 à faire 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.

Dans le monde des bases de données relationnelle, 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 classe. 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 15-4.

La mise en œuvre de l'héritage à une table dans Active Record est très simple. Il suffit de définir la hiérarchie de classe 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 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 définition de la table qui est par ailleurs illustrée figure 15-4 :

Fichier 15.1

```
create table people (
  id              int           not null auto_increment,
  type            varchar(20)   not null,

  /* attributs communs */
  name            varchar(100)  not null,
  email           varchar(100)  not null,
  /* attributs pour type=Customer */
  balance         decimal(10,2),
  /* attributs pour type=Employee */
  reports_to      int,
  dept            int,
```

© 2006 Groupe Eyrolles

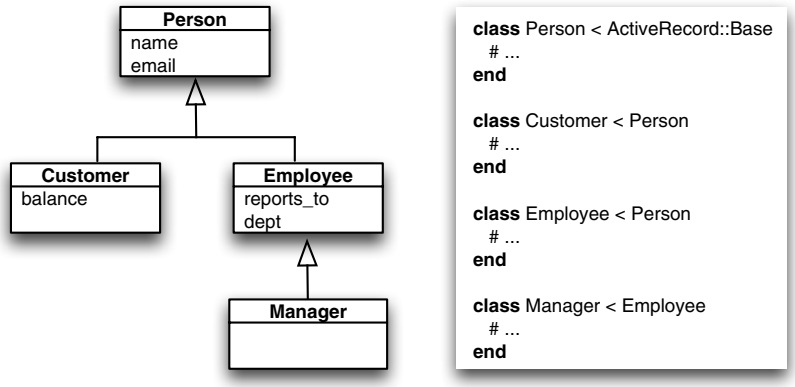
```
  /* attributes pour type=Manager */
  /* -- none -- */
  constraint fk_reports_to foreign key (reports_to) references people(id),
  primary key (id)
);
```

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

Fichier 15.5

```
class Person < ActiveRecord::Base
end
class Customer < Person
end
class Employee < Person
end
class Manager < Employee
end
```

Figure 15-4
Héritage à une table : une hiérarchie de quatre tables 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	-66.75		
7	Employee	Dino Dogg	dino@dig.prg		2	23

© 2006 Groupe Eyrolles

Créons maintenant quelques enregistrements et relisons-les.

Fichier 15.5

```
Manager.create(:name => 'Bob', :email => "bob@some.add",
               :dept => 12, :reports_to => nil)
Customer.create(:name => 'Sally', :email => "sally@other.add",
                :balance => 123.45)
person = Person.find(:first)
puts person.class    #=> Manager
puts person.name     #=> Bob
puts person.dept     #=> 12
person = Person.find_by_name("Sally")
puts person.class    #=> Customer
puts person.email    #=> sally@other.add
puts person.balance  #=> 123.45
```

Vous noterez que nous avons demandé à la classe de base, `Person`, de trouver des enregistrements particuliers 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 détermine le type de l'objet en examinant la valeur de la colonne `type` de l'enregistrement concerné.

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 puisqu'on se retrouverait avec deux colonnes baptisées de la même manière dans la table de la base de données.

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 bien se traduire par l'apparition de quelques messages d'erreurs 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'
```

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.

© 2006 Groupe Eyrolles

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

Mais l'héritage à une table n'est pas toujours la solution idéale. Il ne s'applique pas très bien par exemple dans des hiérarchies où les 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 en définissant une classe `ContentMetadata` que toutes les autres classes concrètes (représentant chacune un modèle) vont pouvoir référencer dans une relation `has_one()`.

Validation

Active Record est capable de valider le contenu de l'objet d'un modèle, notamment juste avant sa sauvegarde dans la base de données. Il est aussi possible de demander la validation de l'état courant d'un modèle par une instruction dans un programme.

Comme indiqué dans le précédent chapitre, Active Record fait la distinction entre les modèles dont les objets correspondent à un enregistrement dans la base de données et ceux qui n'en ont pas encore. Ces derniers sont appelés des *nouveaux enregistrements* (la méthode `new_record?()` retourne `true` pour ces objets). Lorsque vous appelez la méthode `save()`, Active Record exécute une requête SQL `insert` pour les nouveaux enregistrements et une requête SQL `update` pour les enregistrements existants.

Cette distinction se reflète dans le processus de validation de Active Record : il est en effet possible de spécifier certaines opérations de validation applicables à toutes les opérations de sauvegardes et d'autres spécifiques aux opérations de mise à jour et de création.

Au niveau le plus bas, la mise en œuvre de règles de validation se fait par les méthodes `validate()`, `validate_on_create()` et `validate_on_update()`. La méthode `validate()` est invoquée pour toutes les opérations de sauvegarde dans la base de données. Puis l'une des deux autres méthodes est invoquée selon que l'on crée un nouvel enregistrement ou bien qu'on le modifie.

Vous pouvez aussi valider l'objet d'un modèle à tout moment sans avoir à le sauvegarder dans la base de données. Il suffit d'appeler la méthode `valid?()` qui se chargera d'appeler les deux mêmes méthodes de validation que si vous aviez appelé `save()`.

Le code qui suit montre comment s'assurer que la colonne nom d'utilisateur a toujours une valeur valide et que le nom est unique pour tous les objets `User` nouvellement créés (nous verrons plus loin comment spécifier ce genre de contraintes plus simplement).

```
class User < ActiveRecord::Base
  def validate
    unless name && name =~ /^\w+$/
```

© 2006 Groupe Eyrolles

```
      errors.add(:name, "est manquant ou invalide")
    end
  end
  def validate_on_create
    if self.find_by_name(name)
      errors.add(:name, "est déjà utilisé")
    end
  end
end
```

Quand une méthode de validation échoue, elle ajoute un message à la liste des erreurs du modèle en utilisant la méthode `errors.add()`. Le premier paramètre est le nom de l'attribut mis en cause et le second est un message d'erreur. Si vous souhaitez ajouter un message d'erreur qui se rapporte à l'objet du modèle dans son ensemble, utilisez la méthode `add_to_base()` (remarquez dans ce code l'usage de la méthode `blank?()` qui retourne `true` si le receveur du message est égal à `nil` ou bien à une chaîne de caractères vide).

```
def validate
  if name.blank? && email.blank?
    errors.add_to_base("Vous devez spécifier un nom ou une adresse e-mail.")
  end
end
```

Comme vous le verrez page 389, Rails peut utiliser cette liste d'erreurs lors de l'affichage des formulaires HTML – les champs mis en cause seront alors automatiquement surlignés en rouge et il est facile d'ajouter une jolie boîte de présentation en tête de la page pour afficher les messages d'erreur correspondants.

Dans vos programmes vous pouvez accéder à la liste des erreurs liées à un attribut en utilisant la méthode `errors.on()` (alias `errors[:name]`) et vous pouvez effacer la liste des erreurs avec `errors.clear()`. Si vous lisez la documentation RDoc du module `ActiveRecord::Errors` vous découvrirez d'autres méthodes. La plupart d'entre elles ont été remplacées par des méthodes de validation de plus haut niveau.

Assistants de validation

Certaines règles de validation sont constamment utilisées : cet attribut ne doit pas être vide, celui-là doit se situer entre les valeurs 18 et 65, etc. Active Record offre un ensemble de méthodes d'assistance qui permettent d'ajouter ces règles à vos modèles. Ce sont des méthodes de classe dont le nom débute par `validates_`. Chaque méthode prend en paramètre une liste d'attributs suivie par un tableau associatif d'options de configuration de la règle de validation.

© 2006 Groupe Eyrolles

Ainsi, il est possible de réécrire les règles de validation précédentes comme suit :

```
class User < ActiveRecord::Base
  validates_format_of :name,
                      :with    => /^\w+$/,
                      :message => "est manquant ou invalide"
  validates_uniqueness_of :name,
                          :on      => :create,
                          :message => "est déjà utilisé"
        end
```

La majorité des méthodes `validates_xxx` acceptent les options `:on` et `:message`. La première indique dans quels cas la validation doit avoir lieu et prend comme valeur de paramètres `:save` (la valeur par défaut), `:create`, ou `:update`. La seconde peut être utilisée pour remplacer le message d'erreur généré par Rails.

Quand la validation échoue, l'assistant ajoute un objet erreur à l'objet du modèle Active Record. Cet objet est associé au champ en cours de validation. Après la validation, vous pouvez consulter la liste d'erreurs par le biais de l'attribut `errors`. Quand Active Record est utilisé comme module d'une application Rails, la validation est souvent effectuée en deux étapes :

1. Le contrôleur essaye d'abord de sauver l'objet Active Record, retourne `false` si l'opération de validation échoue et affiche de nouveau le formulaire avec les données incriminées.
2. La vue utilise la méthode `error_messages_for()` pour afficher la liste des erreurs de l'objet du modèle et l'utilisateur a alors la possibilité de corriger les valeurs invalides.

Les interactions entre formulaires et modèles sont traitées page 389.

Voici une liste des assistants de validation utilisables sur les objets d'un modèle.

validates_acceptance_of

Vérifie qu'une case à cocher a été cochée.

```
validates_acceptance_of attr... [ options... ]
```

De nombreux formulaires utilisent des cases à cocher pour lesquelles il faut simplement vérifier qu'elles ont bien été cochées (cas de l'approbation d'une licence ou de conditions de vente par exemple). Dans ce cas, il faut vérifier que l'attribut retourné par le formulaire HTML est égal à la chaîne de caractères *1*. L'attribut lui-même ne doit pas forcément être conservé dans la base de données bien que vous puissiez le faire si vous le souhaitez.

© 2006 Groupe Eyrolles

```
class Order < ActiveRecord::Base
  validates_acceptance_of  :terms,
                           :message => "Veuillez accepter les conditions avant de
                                        poursuivre"
end
```

Options :

:message	**text**	La valeur par défaut est « must be accepted » (doit être acceptée).
:on		:save, :create, ou :update.

validates_associated

Effectue une validation des objets associés.

```
validates_associated name... [ options... ]
```

Valide l'attribut spécifié qui doit être l'objet d'un modèle. Pour chaque attribut dont la validation échoue, un message d'erreur est ajouté à la liste d'erreurs de cet attribut (et non du modèle).

Faites attention de ne pas inclure un appel validates_associated() dans des modèles qui se référencent l'un l'autre. En effet, le premier essaierait de valider le second qui, à son tour, tenterait de valider le premier, etc. Jusqu'à ce que la pile des appels de Ruby déborde.

```
class Order < ActiveRecord::Base
  has_many   :line_items
  belongs_to :user
  validates_associated :line_items,
                       :message => "sont en désordre"
  validates_associated :user
end
```

Options :

:message	**text**	La valeur par défaut est « is invalid ».
:on		:save, :create, ou :update.

validates_confirmation_of

Vérifie qu'un champ et sa confirmation ont le même contenu.

```
validates_confirmation_of attr... [ options... ]
```

© 2006 Groupe Eyrolles

De nombreux formulaires nécessitent de saisir la même information à deux reprises pour s'assurer que l'utilisateur ne s'est pas trompé. Si vous utilisez la convention de nommage qui consiste à nommer le second champ comme le premier en le faisant suivre du suffixe `_confirmation`, vous pouvez utiliser `validates_confirmation_of()` pour vérifier que les deux champs ont bien la même valeur. Le second champ n'a pas besoin d'être stocké dans la base de données.

Ainsi, une vue peut contenir :

```
<%= password_field "user", "password" %><br />
<%= password_field "user", "password_confirmation" %><br />
```

Dans le modèle `User`, vous pouvez alors valider que les deux mots de passe sont identiques de cette manière :

```
class User < ActiveRecord::Base
  validates_confirmation_of :password
end
```

Options :

`:message text`	La valeur par défaut est « doesn't match confirmation ».
`:on`	`:save`, `:create`, ou `:update`.

validates_each

Vérifie un ou plusieurs attributs par bloc.

```
validates_each attr... [ options... ] { |model, attr, value| ... }
```

Invoque le bloc de code pour chaque attribut (en omettant ceux dont la valeur est `nil` si `:allow_nil` est vrai). Passe le modèle à valider, le nom de l'attribut et sa valeur. Comme le montre l'exemple suivant, le bloc de code doit renseigner la liste des erreurs du modèle si une validation échoue.

```
class User < ActiveRecord::Base
  validates_each :name, :email do |model, attr, value|
    if value =~ /groucho|harpo|chico/i
      model.errors.add(attr, "Vous n'êtes pas sérieux, #{value}")
    end
  end
end
```

© 2006 Groupe Eyrolles

Options :

:allow_nil boolean	Si :allow_nil est vrai, les attributs de valeur nil ne seront pas passés au bloc de code. Par défaut, ils le sont.
:on	:save, :create, ou :update.

validates_exclusion_of

Vérifie que des attributs ne font pas partie d'un ensemble de valeurs.

```
validates_exclusion_of attr..., :in => enum [ options... ]
```

Valide qu'aucun des attributs ne figure dans l'énumérateur enum passé en paramètre (en Ruby tout objet supportant le prédicat include?() est un énumerateur).

```
class User < ActiveRecord::Base
  validates_exclusion_of :genre,
                         :in => %w{ polka twostep foxtrot },
                         :message => "pas de musique de zazou ici!"
  validates_exclusion_of :age,
                         :in => 13..19,
                         :message => "ne peut pas être un ado"
            end
```

Options :

:allow_nil enum	n'est pas vérifié si un attribut est à nil et que l'option :allow_nil est vraie.
:in (or :within)enumerable	Un objet énumérable.
:message text	La valeur par défaut est « is not included in the list ».
:on	:save, :create, ou :update.

validates_format_of

Vérifie des attributs suivant un modèle.

```
validates_format_of attr..., :with => regexp [ options... ]
```

Valide chaque attribut en comparant sa valeur avec l'expression régulière regexp.

```
class User < ActiveRecord::Base
  validates_format_of :length, :with => /^\d+(in|cm)/
end
```

© 2006 Groupe Eyrolles

Options :

:message text	La valeur par défaut est « is invalid ».
:on	:save, :create, ou :update.
:with	L'expression régulière à utiliser pour valider les attributs.

validates_inclusion_of

Vérifie que des attributs appartiennent à un ensemble de valeurs.

```
validates_inclusion_of attr..., :in => enum [ options... ]
```

Vérifie que la valeur de chaque attribut figure dans l'énumérateur `enum` (en Ruby tout objet supportant le prédicat `include?()` est un énumérateur).

```
class User < ActiveRecord::Base
  validates_inclusion_of :gender,
                         :in => %w{ male female },
                         :message => "should be 'male' or 'female'"
  validates_inclusion_of :age,
                         :in => 0..130,
                         :message => "should be between 0 and 130"
end
```

Options :

:allow_nil	enum n'est pas vérifié si un attribut est à nil et que l'option :allow_nil est vraie.
:in (or :within)enumerable	Un objet énumérable.
:message text	La valeur par défaut est « is not included in the list ».
:on	:save, :create, ou :update.

validates_length_of

Vérifie la taille des attributs.

```
validates_length_of attr..., [ options... ]
```

Vérifie que la taille de chacun des attributs satisfait à certaines contraintes : avoir au moins une certaine taille, au plus une certaine taille, se situer entre deux tailles ou avoir une taille exacte. Au lieu d'une seule option :message, cet assistant de validation en autorise plusieurs selon le type d'erreur ; il est néanmoins possible de continuer à utiliser l'option :message. Quelle que soit l'option, les tailles ne peuvent pas être négatives.

© 2006 Groupe Eyrolles

```
class User < ActiveRecord::Base
  validates_length_of :name,     :maximum => 50
  validates_length_of :password, :in => 6..20
  validates_length_of :address,  :minimum => 10,
                      :message => "paraît un peu court"
end
```

Options :

:in (or :within)range	La taille de la valeur doit figurer dans `range`.
:is integer	La taille doit être égale à `integer` caractères.
:minimum integer	La valeur ne peut être inférieure à `integer` caractères.
:maximum integer	La valeur ne peut être supérieure à `integer` caractères.
:message text	Le message par défaut dépend du test effectué. Votre propre message peut utiliser la chaîne de caractères `%d qui sera remplacée par le minimum, le maximum ou la taille exacte requise.`
:on	`:save`, `:create`, ou `:update`.
:too_long text	Un synonyme pour `:message` quand `:maximum` est utilisé.
:too_short text	Un synonyme pour `:message` quand `:minimum` est utilisé.
:wrong_length text	Un synonyme pour `:message` quand `:is` est utilisé.

validates_numericality_of

Vérifie que les attributs sont des nombres valides.

```
validates_numericality_of attr... [ options... ]
```

Vérifie que chacun des attributs est un nombre valide. Avec l'option `:only_integer`, les attributs doivent commencer par un signe + ou – optionnel suivi de un ou plusieurs chiffres. Sans cette option (ou si l'option est fausse), n'importe quel nombre flottant accepté par la méthode `Float()` de Ruby est autorisé.

```
class User < ActiveRecord::Base
  validates_numericality_of :height_in_meters
  validates_numericality_of :age, :only_integer => true
end
```

Options :

:message text	La valeur par défaut est « is not a number ».
:on	`:save`, `:create`, ou `:update`.

© 2006 Groupe Eyrolles

`:only_integer`	Si vrai, les attributs ne peuvent contenir que les caractères + ou – suivi de chiffres (un entier en somme).

validates_presence_of

Vérifie que les attributs ne sont pas vides.

```
validates_presence_of attr... [ options... ]
```

Vérifie que chacun des attributs n'est ni égal à `nil` ni vide.

```
class User < ActiveRecord::Base
  validates_presence_of :name, :address
end
```

Options :

`:message text`	La valeur par défaut est « can't be empty ».
`:on`	`:save`, `:create`, ou `:update`.

validates_uniqueness_of

Vérifie que les attributs sont uniques.

```
validates_uniqueness_of attr... [ options... ]
```

Pour chaque attribut, vérifie qu'aucun autre enregistrement dans la base de données n'utilise la même valeur pour la colonne spécifiée. Quand l'objet du modèle provient d'un enregistrement existant, il est ignoré durant le test. Le paramètre facultatif `:scope` permet de limiter les enregistrements testés à ceux dont la valeur de la colonne `:scope` est identique à celle de l'enregistrement testé.

Cet exemple s'assure que les noms des utilisateurs sont uniques dans la base de données.

```
class User < ActiveRecord::Base
  validates_uniqueness_of :name
end
```

Celui-ci vérifie que les noms des utilisateurs sont uniques au sein d'un groupe.

```
class User < ActiveRecord::Base
  validates_uniqueness_of :name, :scope => "group_id"
end
```

© 2006 Groupe Eyrolles

Options :

`:message`	`text`	La valeur par défaut est « has already been taken ».
`:on`		`:save`, `:create`, ou `:update`.
`:scope`	`attr`	Limite la validation aux enregistrements dont la valeur de la colonne spécifiée est la même que celle de l'enregistrement testé.

Procédures de rappel (callbacks)

Active Record contrôle le cycle de vie des objets d'un modèle : il les crée, vérifie leur conformité après modification, les sauvegarde, les met à jour et les regarde disparaître. Par le biais des procédures de rappel, Active Record permet au programmeur d'intervenir tout au long de ce cycle de vie. Nous pouvons faire en sorte que notre propre code soit invoqué par Active Record à n'importe quel moment critique de la vie de l'objet. C'est ainsi qu'il est possible d'effectuer des validations complexes, de mettre en correspondance des valeurs de colonnes à leur entrée ou leur sortie de la base de données et même d'empêcher certaines opérations d'aboutir.

Nous avons déjà utilisé cette fonctionnalité par le passé. Lorsque nous avons ajouté le code de maintenance des utilisateurs à notre application Dépôt, il a fallu s'assurer que l'utilisateur magique Dave ne pouvait être détruit de la base de données. Cette vérification avait été faite par le biais d'une procédure de rappel de la classe `User` comme suit :

```
class User < ActiveRecord::Base
  before_destroy :dont_destroy_dave
  def dont_destroy_dave
    raise "Impossible de détruire dave" if name == 'dave'
  end
end
```

L'appel à `before_destroy` enregistre la méthode `dont_destroy_dave()` comme une procédure de rappel à invoquer avant que tout objet de la classe `User` ne soit détruit. S'il y a tentative de destruction de l'utilisateur Dave, cette méthode lève une exception et la colonne n'est pas détruite.

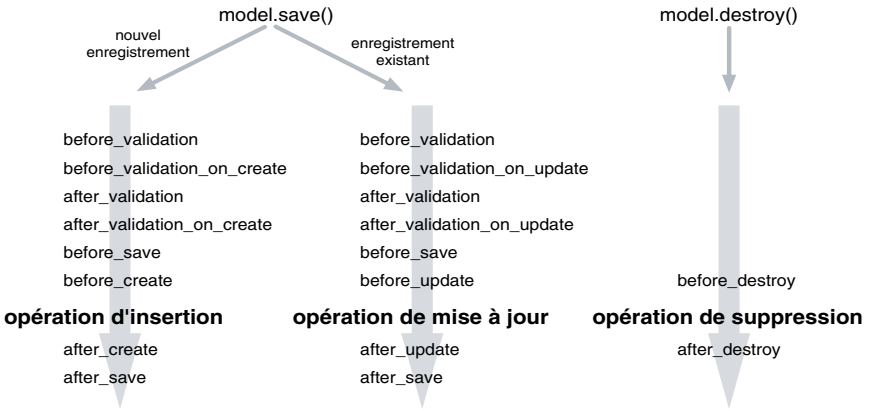
Active Record définit seize procédures de rappel. Quatorze d'entre elles forment des paires avant/après et enserrent certaines opérations effectuées sur les objets Active Record. Ainsi la procédure de rappel `before_destroy` est invoquée juste avant l'appel à la méthode `destroy()` et `after_destroy` juste après. Les deux exceptions sont `after_find` et `after_initialize`, qui n'ont aucune procédure `before_`*xxx* correspondantes. Ces deux procédures de rappel diffèrent également par d'autres points comme nous le verrons plus loin.

La figure 15-5 montre les sept paires de procédure de rappel qui enserrent les opérations de création, de modification et de destruction des objets d'un modèle.

© 2006 Groupe Eyrolles

En plus de ces quatorze procédures de rappel, `after_find` est invoquée après une opération find et `after_initialize` après toute nouvelle création de l'objet d'un modèle.

Figure 15-5
Séquence des procédures de rappel de Active Record



Pour que votre code soit exécuté par une procédure de rappel, vous devez écrire une méthode et l'associer avec la procédure de rappel adéquate.

Il y a deux façons de mettre en œuvre une procédure de rappel.

La première consiste à définir directement la procédure de rappel dans votre classe. Si vous souhaitez intercepter le cycle de vie juste avant la sauvegarde d'un objet, vous pouvez écrire :

```
class Order < ActiveRecord::Base
# ..
  def before_save
    self.payment_due ||= Time.now + 30.days
  end
end
```

Pourquoi after_find et after_initialize sont-ils des cas particuliers ?

Rails utilise les capacités de réflexion du langage Ruby pour déterminer si certaines procédures de rappel doivent être invoquées. Quand on effectue de vraies opérations dans une base de données, le coût engendré est normalement insignifiant par rapport au temps d'interaction avec la base de données. Cependant, une seule instruction `select` sur la base de données peut retourner des centaines d'enregistrements et les deux procédures de rappel seront invoquées pour chacun d'eux. Il en résulte un ralentissement considérable des traitements mais, dans ce cas particulier, l'équipe de développement de Rails a décidé que la cohérence devait l'emporter sur la performance.

© 2006 Groupe Eyrolles

La deuxième façon de faire consiste à déclarer un gestionnaire de procédures de rappel. Ce gestionnaire peut être défini soit sous la forme d'une méthode, soit sous la forme d'un bloc de code[1]. Pour associer un gestionnaire avec un événement particulier, on utilise des méthodes de classe portant le nom de l'événement. Pour associer une méthode, déclarez-la `protected` ou `private` et spécifiez son nom sous la forme d'un symbole dans la déclaration du gestionnaire. Pour utiliser un bloc de code, ajoutez-le simplement après la déclaration. Ce bloc recevra l'objet du modèle en paramètre lors de son exécution.

```
class Order < ActiveRecord::Base
  before_validation :normalize_credit_card_number
  after_create do |order|
    logger.info "Order #{order.id} created"
  end
  protected
  def normalize_credit_card_number
    self.cc_number.gsub!(/-\w/, '')
  end
end
```

Vous pouvez définir plusieurs gestionnaires pour un même événement. Ils seront en général invoqués dans l'ordre où ils ont été associés sauf lorsque l'un d'eux renvoie la valeur `false` (et ce doit être la vraie valeur `false`), auquel cas la chaîne des procédures de rappel est interrompue avant la fin.

Pour des raisons d'optimisation de performance, la seule façon de définir une procédure de rappel pour les événements `after_find` et `after_initialize` est de les définir sous la forme de méthode. Si vous essayez de les déclarer suivant la deuxième façon, il seront tout simplement ignorés par Rails.

Enregistrements marqueurs de temps

Un des usages potentiels des procédures de rappel `before_create` et `before_update` consiste à horodater les enregistrements.

```
class Order < ActiveRecord::Base
  def before_create
    self.order_created ||= Time.now
  end
  def before_update
```

1. Un gestionnaire peut aussi être une chaîne de caractères contenant du code à évaluer mais cette forme n'est aujourd'hui plus recommandée.

© 2006 Groupe Eyrolles

```
    self.order_modified = Time.now
  end
end
```

Toutefois, Active Record peut vous épargner cette peine. Si vous définissez dans vos tables une colonne baptisée `created_at` ou `created_on`, elle est automatiquement renseignée avec l'heure et la date de création de l'enregistrement. De la même manière, une colonne portant le nom `updated_at` ou `updated_on` est renseignée avec l'heure de la dernière modification. Par défaut, ces marqueurs de temps sont stockés en heure locale. Pour les stocker sous forme UTC (aussi connu sous le nom GMT), il suffit d'inclure la ligne de code suivante dans votre code (soit dans le code lui-même pour une application Active Record, soit dans un fichier d'environnement pour une application Rails).

```
ActiveRecord::Base.default_timezone = :utc
```

Pour désactiver complètement cette fonctionnalité, écrivez :

```
ActiveRecord::Base.record_timestamps = false
```

Objets procédure de rappel

Plutôt que de définir les procédures de rappel directement dans la classe du modèle, il est possible de créer des classes de gestionnaires destinées à encapsuler toutes les procédures de rappel. Ces gestionnaires présentent l'avantage d'être utilisables par plusieurs modèles. Une classe de gestionnaires est une classe ordinaire qui définit des méthodes portant les noms des procédures de rappel (`before_save()`, `after_create()`, etc.). Les fichiers source de ces classes doivent être placés dans le répertoire *`app/models`*.

Dans le modèle objet qui utilise les gestionnaires, il suffit ensuite de créer une instance de cette classe de gestionnaires et de passer cette instance aux divers appels de déclaration. Quelques exemples rendront les choses plus faciles à comprendre.

Si notre application utilise les cartes de paiement électroniques en plusieurs endroits, il est probablement souhaitable de partager la méthode `normalize_credit_card_number()`. Pour ce faire, nous plaçons cette méthode dans sa classe propre et la rebaptisons du nom de l'événement que nous souhaitons intercepter. Cette méthode reçoit un seul paramètre, à savoir l'objet du modèle qui a généré la procédure de rappel.

```
class CreditCardCallbacks
  # Normaliser le numéro de carte de crédit
  def before_validation(model)
```

© 2006 Groupe Eyrolles

```
    model.cc_number.gsub!(/-\w/, '')
  end
end
```

Nous pouvons maintenant invoquer cette procédure de rappel partagée dans nos classes Active Record.

```
class Order < ActiveRecord::Base
  before_validation CreditCardCallbacks.new
  # ...
end
class Subscription < ActiveRecord::Base
  before_validation CreditCardCallbacks.new
  # ...
end
```

Dans cet exemple, la classe de gestionnaires suppose que le numéro de la carte de crédit se trouve dans l'attribut du modèle `cc_number` ; ce qui implique que les classes `Order` et `Subscription` utilisent le même nom d'attribut. Mais nous pouvons généraliser cette idée et rendre le gestionnaire moins dépendant des détails d'implémentation de la classe qui l'utilise.

C'est ainsi que nous pouvons créer un gestionnaire de cryptage et de décryptage totalement générique dont le rôle est de crypter certains champs nommés avant qu'ils ne soient enregistrés dans la base de données et de les décrypter lorsqu'ils sont relus. Du même coup ce gestionnaire devient utilisable depuis n'importe quel modèle.

Le gestionnaire doit crypter[1] un ensemble d'attributs d'un modèle juste avant que les données du modèle ne soient enregistrées dans la base de données. étant donné que notre application doit aussi accéder aux valeurs en clair de ces données, le gestionnaire décrypte à nouveau les données après la sauvegarde et il les décrypte aussi lors de la lecture d'un nouvel enregistrement. Tout cela signifie que nous devons gérer les événements `before_save`, `after_save` et `after_find`. Puisque nous devons décrypter l'enregistrement de la base de données après sa sauvegarde et lors de sa relecture, nous pouvons économiser du code en créant un alias de la méthode `after_find()` vers `after_save()` (ce qui signifie que la même méthode porte deux noms).

Fichier 15.6

```
class Encrypter
  # On nous passe une liste d'attributs qui devraient
  # être stockés de façon cryptée dans la bdd
  def initialize(attrs_to_manage)
    @attrs_to_manage = attrs_to_manage
```

1. Notre exemple utilise un cryptage des plus simples. Vous pouvez l'améliorer avant de réellement utiliser cette classe.

© 2006 Groupe Eyrolles

```
  end
  # Avant de sauvegarder ou de mettre à jour, crypter les
  # champs avec la méthode du décalage approuvée par la
  # DGSE
  def before_save(model)
    @attrs_to_manage.each do |field|
      model[field].tr!("a-z", "b-za")
    end
  end
  # Après la sauvegarde, les décrypter à nouveau
  def after_save(model)
    @attrs_to_manage.each do |field|
      model[field].tr!("b-za", "a-z")
    end
  end
  # Faire la même chose après avoir trouvé un enregistrement existant
  alias_method :after_find, :after_save
end
```

La classe `Encrypter` peut maintenant être invoquée depuis notre modèle de commandes (classe `Order`).

```
require "encrypter"
class Order < ActiveRecord::Base
  encrypter = Encrypter.new(:name, :email)
  before_save encrypter
  after_save  encrypter
  after_find  encrypter
  protected
  def after_find
  end
end
```

Nous créons un nouvel objet `Encrypter` et l'associons aux événements `before_save`, `after_save`, et `after_find`. Ainsi, avant toute sauvegarde d'une commande, la méthode `before_save()` du gestionnaire de cryptage sera invoquée et toutes les autres méthodes le seront aussi au moment opportun.

Mais pourquoi définir une méthode `after_find()` vide ? Nous avons dit plus haut que pour des raisons de performance, `after_find` et `after_initialize` sont traitées de façon particulière. Entre autres, Active Record n'appellera pas `after_find` à moins qu'il voit une méthode

© 2006 Groupe Eyrolles

du même nom dans la classe du modèle. Il faut donc définir une classe fictive pour que le traitement `after_find` ait lieu.

Tout cela est très bien mais toutes les classes des modèles qui veulent utiliser notre gestionnaire de cryptage vont devoir introduire à peu près 8 lignes de code supplémentaires comme pour la classe `Order`. On peut mieux faire. Nous pouvons en effet définir une méthode qui va se charger de cette besogne et que chaque modèle pourra solliciter. C'est pourquoi nous la plaçons dans la classe `ActiveRecord::Base`.

Fichier 15.6

```
class ActiveRecord::Base
  def self.encrypt(*attr_names)
    encrypter = Encrypter.new(attr_names)

    before_save encrypter
    after_save  encrypter
    after_find  encrypter
    define_method(:after_find) { }
  end
end
```

Avec cette nouvelle méthode, il est possible d'ajouter un gestionnaire de cryptage à n'importe quel attribut d'une classe de modèle en utilisant un seul appel.

Fichier 15.6

```
class Order < ActiveRecord::Base
  encrypt(:name, :email)
end
```

Expérimentons tout cela avec le petit programme suivant :

Fichier 15.6

```
o = Order.new
o.name = "Dave Thomas"
o.address = "123 The Street"
o.email   = "dave@pragprog.com"
o.save
puts o.name
o = Order.find(o.id)
puts o.name
```

Sur la console, nous voyons le nom de notre client en clair dans l'objet du modèle.

```
ar> ruby encrypt.rb
Dave Thomas
Dave Thomas
```

© 2006 Groupe Eyrolles

Dans la base de données, cependant, le nom et l'adresse e-mail sont masqués par l'algorithme de cryptage.

```
ar>  mysql -urailsuser -prailspw railsdb
mysql> select * from orders;
+----+-------------+-------------------+----------------+----------+-------------+
| id | name        | email             | address        | pay_type | when_shipped |
+----+-------------+-------------------+----------------+----------+-------------+
|  1 | Dbwf Tipnbt | ebwf@qsbhqsph.dpn | 123 The Street |          |         NULL |
+----+-------------+-------------------+----------------+----------+-------------+
1 row in set (0.00 sec)
```

Observateurs

Les procédures de rappel constituent une belle technique mais dans certains cas elles amènent la classe d'un modèle à prendre des responsabilités qui ne sont pas vraiment les siennes. Par exemple, page 294, nous avons créé une procédure de rappel qui génère un message dans le journal de Rails à chaque création d'une commande. Cette fonctionnalité ne fait pas vraiment partie de la classe `Order` mais nous l'avons placé là parce que les procédures de rappel s'y trouvent aussi.

Les *observateurs* de Active Record permettent de dépasser cette limitation. Un observateur est capable de s'attacher à une classe d'un modèle de façon totalement transparente, c'est-à-dire qu'il s'enregistre comme une procédure de rappel auprès de la classe mais sans nécessiter de changement dans le code de la classe elle-même. Voici le précédent exemple d'écriture dans le journal réécrit à l'aide d'un observateur.

Fichier 15.7

```
class OrderObserver < ActiveRecord::Observer
  def after_save(an_order)
    an_order.logger.info("Commande #{an_order.id} créée")
  end
end
OrderObserver.instance
```

Quand `ActiveRecord::Observer` est sous-classé, il regarde le nom de la nouvelle classe, supprime le suffixe `Observer` à la fin, et prend ce qui reste comme nom de la classe à observer. Dans notre exemple, nous avons appelé la classe d'observation `OrderObserver` et elle va donc s'accrocher automatiquement à la classe `Order`.

Parfois cette convention de nommage ne fonctionne pas. Si c'est le cas, la classe d'observation peut mentionner explicitement le ou les modèles à observer en utilisant la méthode `observe()`.

© 2006 Groupe Eyrolles

Fichier 15.7

```
class AuditObserver < ActiveRecord::Observer
  observe Order, Payment, Refund
  def after_save(model)
    model.logger.info("#{model.class.name} #{model.id} créé")
  end
end
AuditObserver.instance
```

Dans les deux exemples qui précèdent nous avons dû créer une instance de l'observateur, la définition de la classe ne suffisant pas à activer cet observateur. Pour les applications Active Record pures, vous devez appeler la méthode `instance()` durant l'initialisation. Dans une application Rails, vous utiliserez la directive `observer` dans la classe `ApplicationController`, comme nous le verrons page 307.

Par convention, les fichiers source des observateurs résident dans *`app/models`*.

D'une certaine façon, les observateurs apportent à Rails la plupart des avantages de la programmation orientée aspect de première génération des langages tels que Java. Ils vous permettent d'injecter des comportements particuliers au sein d'une classe de modèles sans changer son code.

Attributs avancés

Dans l'introduction de Active Record, nous avons mentionné qu'un objet Active Record possède des attributs qui correspondent aux noms des colonnes dans la table de la base de données. Nous avions aussi signalé au passage que cette description n'était pas tout à fait exacte. Voici pourquoi.

Quand Active Record utilise un modèle pour la première fois, il rend visite à la base de données et détermine quelles sont les colonnes utilisées par la table correspondant au modèle. À partir de là, Active Record construit un ensemble d'objets de type `Column`. Ces objets sont accessibles via la méthode de classe `columns()`, et l'objet `Column` correspondant à une colonne donnée de la table peut être récupéré en utilisant la méthode `columns_hash()`. Les objets `Column` fournissent le nom, le type et la valeur par défaut définis dans la base de données.

Quand Active Record lit des informations dans la base de données, il construit une requête SQL `select` qui, après exécution, retourne un nombre variable d'enregistrements. Active Record construit un nouvel objet du modèle pour chacun d'eux en chargeant les données brutes dans un tableau associatif, qu'il appelle les données d'*attributs*. Chaque entrée du tableau correspond à un résultat de la requête de départ, la valeur de la clé utilisée par le tableau étant le nom de l'élément dans le jeu de résultats.

La plupart du temps nous utilisons un finder standard de Active Record pour extraire les données de la base de données. Ces méthodes renvoient la totalité des colonnes de la table

© 2006 Groupe Eyrolles

pour chacun des enregistrements sélectionnés. En conséquence, le tableau associatif des attributs contient une entrée pour chaque colonne, où le nom de la colonne représente la clé et la valeur ainsi que les données de la colonne.

```
result = LineItem.find(:first)
p result.attributes
{"order_id"=>13,
 "quantity"=>1,
 "product_id"=>27,
 "id"=>34,
 "unit_price"=>29.95}
```

Habituellement, on n'accède pas aux données via le tableau associatif des attributs mais par les méthodes d'attributs.

```
result = LineItem.find(:first)
p result.quantity         #=>  1
p result.unit_price       #=>  29.95
```

Mais que se passe-t-il si nous exécutons une requête qui renvoie des noms de colonnes qui ne correspondent pas aux colonnes de la table ? Par exemple, dans la requête qui suit, une colonne (la seconde) est renvoyée qui ne correspond à aucun attribut.

```
select quantity, quantity*unit_price from line_items;
```

Si nous exécutons cette requête manuellement, le résultat suivant apparaît :

```
mysql> select quantity, quantity*unit_price from line_items;
+----------+--------------------+
| quantity | quantity*unit_price |
+----------+--------------------+
|        1 |              29.95 |
|        2 |              59.90 |
|        1 |              44.95 |
         :                  :
```

Notez que dans la liste des résultats les noms des colonnes reflètent ceux utilisés dans la requête `select`. Ces entités sont utilisées par Active Record pour nommer les entrées du tableau associatif contenant les résultats. C'est ce que nous pouvons vérifier en exécutant la même requête depuis Active Record avec la méthode `find_by_sql()` et en regardant le tableau des résultats.

© 2006 Groupe Eyrolles

```
result = LineItem.find_by_sql("select quantity, quantity*unit_price " +
                              "from line_items")
p result[0].attributes
```

Les informations affichées montrent bien que les noms des colonnes sont utilisés comme clés dans le tableau associatif.

```
{"quantity*unit_price"=>"29.95",
 "quantity"=>1}
```

Vous remarquerez au passage que la colonne calculée est renvoyée sous la forme d'une chaîne de caractères. Active Record connaît le type des colonnes de notre table mais de nombreux moteurs de base de données ne renvoient pas le type des colonnes calculées. MySQL en fait partie et Active Record laisse donc le résultat sous forme d'une chaîne de caractères. Si nous avions utilisé Oracle, nous aurions reçu un résultat de type `Float` car le pilote OCI d'Oracle fournit l'information de type pour toutes les colonnes d'un jeu de résultats.

Ce n'est pas vraiment pratique d'accéder aux attributs calculés en utilisant la clé `quantity*price` et c'est pourquoi on utilise normalement la directive `as` dans la requête SQL pour donner un nom plus parlant à la colonne calculée.

```
result = LineItem.find_by_sql("select quantity,
                                      quantity*unit_price as total_price " +
                              "  from line_items")
p result[0].attributes
```

Ce qui produit :

```
{"total_price"=>"29.95",
 "quantity"=>1}
```

L'attribut `total_price` est sans aucun doute plus facile à utiliser.

```
result.each do |line_item|
  puts "Item-ligne #{line_item.id}:  #{line_item.total_price}"
end
```

Rappelez-vous, cependant, que les valeurs de ces colonnes calculées sont stockées dans les attributs d'un tableau associatif sous la forme de chaînes de caractères. Vous aurez donc un résultat assez inattendu si vous tentez quelque chose comme :

© 2006 Groupe Eyrolles

```
TAX_RATE = 0.07
# ...
sales_tax = line_item.total_price * TAX_RATE
```

Vous serez sûrement étonnés d'apprendre que, dans cet exemple, `sales_tax` reçoit finalement une chaîne de caractères vide. C'est en fait tout à fait normal : la variable `total_price` est elle-même une chaîne de caractères et l'opérateur `*` agit alors comme un duplicateur de contenu. Comme `TAX_RATE` est inférieur à 1, le contenu est dupliqué zéro fois donnant ainsi une chaîne vide.

Cependant, tout n'est pas perdu. Il est possible de redéfinir les méthodes d'accès aux attributs et d'exécuter les conversions de type nécessaires pour les champs calculés.

```
class LineItem < ActiveRecord::Base
  def total_price
    Float(read_attribute("total_price"))
  end
end
```

Notez que nous avons accédé à la valeur interne de l'attribut en utilisant la méthode `read_attribute()`, plutôt que d'aller directement le chercher dans le tableau associatif des attributs. La méthode `read_attribute()` connaît les types des colonnes de la base de données (y compris les colonnes contenant des données Ruby sériées) et se charge des conversions de type nécessaires. Ce n'est pas spécialement utile dans notre exemple mais ça le deviendra lorsque nous regarderons comment fournir des colonnes de façade.

Colonnes de façade

Il peut arriver que nous héritions d'un schéma de base de données dont certaines colonnes offrent un format peu pratique à manipuler. Il serait utile de pouvoir habiller ces colonnes pour qu'elles présentent un autre visage à l'application tout en restant inchangées dans la base de données.

Il s'avère que Rails nous permet de le faire en redéfinissant les méthodes d'accès aux attributs fournies par Active Record. Imaginons, par exemple, que notre application utilise une table existante `product_data`, une table si vieille que les dimensions des produits stockés dans cette table sont mesurées en cubits[1]. Dans notre application, on préférerait évidemment avoir affaire à des centimètres. Nous allons donc définir des méthodes d'accès aux attributs qui effectuent les opérations de conversion nécessaires.

1. Un *cubit* est défini comme la longueur allant du coude au bout du doigt le plus long de la main. Dans la mesure où cette mesure est très subjective, les égyptiens avaient établi le cubit royal comme standard. Ils possédaient même un corps royal standard, avec un cubit étalon gravé dans le marbre (http://www.ncsli.org/misc/cubit.cfm).

© 2006 Groupe Eyrolles

```
class ProductData < ActiveRecord::Base
  CUBITS_TO_CM = 46
  def length
    read_attribute("length") * CUBITS_TO_CM
  end
  def length=(centimetres)
    write_attribute("length", Float(centimetres) / CUBITS_TO_CM)
  end
end
```

Divers

Cette section traite de divers points relatifs à Active Record qui n'ont pas trouvé leur place ailleurs.

Identité des objets

Les objets d'un modèle redéfinissent les méthodes standards `id()` et `hash()` de Ruby pour pointer sur la clé primaire d'un modèle. Cela signifie que les objets d'un modèle munis d'identifiants valides peuvent être utilisés comme clé de hachage. A contrario, cela implique aussi que des objets d'un modèle qui n'ont pas encore été sauvegardés dans la base de données ne peuvent pas être utilisés comme clés de hachage puisqu'ils n'ont pas d'identifiant valide.

Deux objets d'un modèle sont considérés comme égaux (en utilisant ==) s'ils émanent tous deux de la même classe et que leurs clés primaires sont identiques. Cela signifie que les objets d'un modèle qui n'ont pas été sauvegardés peuvent s'avérer égaux bien qu'ils aient des données d'attributs différentes. Si vous vous retrouvez à comparer des objets d'un modèle non sauvegardés (ce qui n'est pas si fréquent), il se peut que vous deviez redéfinir la méthode ==.

Utiliser la connexion en direct

Les requêtes SQL peuvent être exécutées en utilisant le pilote de la connexion à la base de données de Active Record. Cela peut être utile dans les rares cas où vous devez interagir avec la base de données en dehors du contexte d'un modèle Active Record.

Au niveau le plus bas, vous pouvez appeler `execute()` pour lancer une requête SQL. La valeur de retour dépend du pilote de base de données utilisé. Pour MySQL, par exemple, la valeur retournée est une instance de la classe `Mysql::Result`. Si vous avez réellement besoin de travailler à un niveau aussi bas, il est fortement recommandé de lire les détails de cette méthode directement dans le code de Active Record. Heureusement, vous ne devriez pas avoir à le faire très souvent, car le pilote de la base de données de Active Record fournit une abstraction de plus haut niveau.

© 2006 Groupe Eyrolles

La méthode `select_all()` exécute une requête et retourne un tableau de tableaux associatifs d'attributs correspondant aux résultats.

```
res = Order.connexion.select_all("select id, "+
                                 "       quantity*unit_price as total " +
                                 "  from line_items")
p res
```

Voici ce qu'affiche ce code :

```
 [{"total"=>"29.95", "id"=>"91"},
  {"total"=>"59.90", "id"=>"92"},
  {"total"=>"44.95", "id"=>"93"}]
```

La méthode `select_one()` retourne un tableau associatif unique, qui correspond au premier enregistrement de la liste des résultats.

Consultez la documentation RDoc de la classe `AbstractAdapter` pour une liste complète des méthodes de bas niveau disponibles dans Active Record.

Le cas de l'identifiant manquant

Il existe un danger potentiel à utiliser votre propre requête de recherche SQL pour extraire des enregistrements de la base de données.

Comme vous le savez, Active Record utilise une colonne `id` pour identifier de façon unique les enregistrements provenant de la base de données. Si vous ne récupérez pas la colonne id parmi les données de votre requête `find_by_sql()`, il vous sera impossible de sauvegarder les modifications des objets correspondants dans la base de données. Malheureusement, Active Record va tout de même essayer et échouer en silence. À titre d'exemple, le code suivant ne met pas à jour la base de données et ne se plaint de rien :

```
result = LineItem.find_by_sql("select quantity from line_items")
result.each do |li|
  li.quantity += 2
  li.save
end
```

Peut-être Active Record détectera-t-il un jour l'absence de l'attribut id et lèvera-t-il une exception en conséquence. En attendant, la morale est claire : récupérez toujours la colonne qui représente la clé primaire si vous avez l'intention de sauvegarder un objet Active Record dans la base de données. En fait, à moins que vous ayez une bonne raison de ne pas le faire, il est toujours plus sûr de lancer vos propres recherches avec une requête SQL `select *` qui ramènera toutes les colonnes de la table y compris la clé primaire.

© 2006 Groupe Eyrolles

Noms des colonnes magiques

Au fil des deux derniers chapitres, nous avons rencontré un certain nombre de colonnes qui jouent un rôle particulier pour Active Record. En voici un résumé.

`created_at, created_on, updated_at, updated_on`

Colonne automatiquement mise à jour avec l'heure (pour la forme (`_at`) ou la date (pour la forme `_on`) de création ou de dernière modification de l'enregistrement (page 295).

`lock_version`

Rails incrémente automatiquement le numéro de version d'un enregistrement et procède à un verrouillage optimiste si une table contient une colonne `lock_version` (page 237).

`type`

Utilisée par l'héritage à une table pour conserver la classe de l'objet stocké dans l'enregistrement (page 280).

`id`

Nom par défaut de la clé primaire d'une table (page 221).

`xxx_id`

Nom par défaut de la clé étrangère référençant la table portant le nom `xxx` dans sa forme plurielle (page 240).

`xxx_count`

Maintient un cache du compteur des enregistrements de la table fille `xxx` (page 260).

`position`

Codifie la position d'un enregistrement si le comportement `acts_as_list` est utilisé (page 270).

`parent_id`

Contient une référence à l'identifiant du parent de cet enregistrement lors de l'utilisation du comportement `acts_as_tree` (page 270).

© 2006 Groupe Eyrolles