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

15

Active Support

Active Support est un ensemble de bibliothèques utilisées par tous les composants de Rails. La plupart des éléments qui s'y trouvent sont destinés à l'usage interne de Rails. Cependant, Active Support apporte certaines extensions à des classes standards de Ruby qui sont tout à fait intéressantes à utiliser dans nos applications. Dans cette section nous allons rapidement énumérer celles qui sont le plus fréquemment utilisées.

Nous terminerons en jetant un bref coup d'œil sur la façon dont Ruby et Rails manipulent les (chaînes de) caractères Unicode et permettent ainsi la gestion de texte international dans vos sites Web.

Extensions générales

Comme nous le verrons en détaillant Ajax à partir du chapitre 23, il s'avère quelquefois utile de pouvoir convertir des objets Ruby dans une forme neutre afin de pouvoir communiquer avec un programme distant (le plus souvent du JavaScript dans le navigateur de l'utilisateur). Pour cela, Rails étend les objets Ruby à l'aide de deux méthodes, `to_json()` et `to_yaml()`, qui convertissent en utilisant soit la notation d'objet JavaScript (JavaScript Object Notation : JSON) soit YAML, qui est la même notation que celle utilisée dans la configuration de Rails et les fichiers de garniture.

```
# Pour les besoins de la démo, crée une structure Ruby à deux attributs
Rating = Struct.new(:name, :ratings)
rating = Rating.new("Rails", [ 10, 10, 9.5, 10 ])
# et sérialise un objet de cette structure de deux manières...
```

```
puts rating.to_json      #=> ["Rails", [10, 10, 9.5, 10]]
puts rating.to_yaml      #=> --- !ruby/struct:Rating
                             name: Rails
                             ratings:
                             - 10
                             - 10
                             - 9.5
                             - 10
```

De plus, tous les objets Active Record, ainsi que tous les tableaux associatifs bénéficient de la méthode `to_xml()` que nous avons abordée au chapitre 12, section *Autogénération du XML*.

Afin de faciliter la vérification de la présence de contenu dans un objet, Rails étend à tous les objets Ruby la méthode `blank?()` qui, quand l'objet est `nil` ou `false`, retourne toujours `true` (une chaîne contenant seulement des espaces est considérée comme vide).

```
puts [ ].blank?          #=> true
puts { 1 => 2}.blank?    #=> false
puts "   cat ".blank?    #=> false
puts "".blank?           #=> true
puts "   ".blank?        #=> true
puts nil.blank?          #=> true
```

Pourquoi l'extension des classes de base ne conduit-elle pas à l'Apocalypse ?

La crainte qu'entraîne la première rencontre d'une expression comme `5 months + 30 minutes` est en général assez rapidement suivie d'un état de panique : si chacun pouvait changer à sa guise la façon dont fonctionnent les entiers, cela ne conduirait-il pas rapidement à un sac de nœuds inextricable ? Si on le faisait en permanence, oui. Mais comme on ne le fait pas, ce n'est pas le cas !

N'allez pas imaginer que Active Support n'est qu'une collection d'extensions aléatoires au langage Ruby, invitant chacun et son voisin à modifier la classe `String` par une fonctionnalité maison de leur choix. Pensez-le comme un dialecte de Ruby, parlé universellement par tous les développeurs Rails. Comme Active Support fait partie intégrante de Rails, vous pouvez être sûr du fait que `5.months` fonctionnera dans toute application Rails. Active Support élimine le risque de se retrouver face à un millier de dialectes individualisés de Ruby.

Active Support nous offre le meilleur des mondes possibles en matière d'extension de langage : la standardisation contextuelle.

Énumérations et tableaux

Nos applications Web utilisant abondamment les collections d'objets, Rails enrichit également les capacités du module `Enumerable` de Ruby.

La méthode `group_by()` partitionne une collection en jeux de valeurs. Ceci est rendu possible par l'appel d'un bloc sur chaque élément de la collection, qui utilise les résultats retournés comme clés de hachage. Ce qui donne au final un tableau associatif dont chaque clé pointe vers un tableau d'éléments issus de la collection originale et partageant cette clé. L'exemple suivant explique comment éclater un ensemble d'articles (*posts* en anglais) en les triant par auteur.

```
groups = posts.group_by {|post| post.author_id}
```

La variable `groups` consistera en un tableau associatif dont les clés seront les identifiants des auteurs et dont les valeurs associées à ces clés seront des tableaux contenant les articles écrits par les auteurs en question.

On pourrait aussi bien l'écrire :

```
groups = posts.group_by {|post| post.author}
```

Les groupes résultants seront les mêmes dans les deux cas. Dans le second cependant, ce sont les noms des objets de la classe `Author` qui seront utilisés comme clés de hachage ce qui signifie que les objets auteur seront extraits de la base de données pour chaque message. La meilleure manière d'effectuer cette opération dépend de votre application.

Rails étend le module `Enumerable` de deux autres méthodes. La méthode `index_by()` convertit une collection en un tableau associatif dans lequel les valeurs de la collection d'origine seront conservées. La clé de hachage référençant chaque valeur étant déterminée par le passage d'un paramètre au bloc.

```
us_states = State.find(:all)
state_lookup = us_states.index_by {|state| state.short_name}
```

La méthode `sum()` effectue la somme des valeurs d'une collection en passant chacun de ses éléments par un bloc totalisant les résultats renvoyés. Ce bloc considère que la valeur initiale par défaut de l'accumulateur est à zéro mais vous pouvez le modifier en passant un paramètre à la méthode `sum()`.

```
total_orders = Order.find(:all).sum {|order| order.value }
```

Rails ajoute encore quelques méthodes très pratiques aux tableaux :

```
puts [ "ant", "bat", "cat"].to_sentence  #=> "ant, bat, and cat"
puts [ "ant", "bat", "cat"].to_sentence(:connector => "and not forgetting")
                                        #=> "ant, bat, and not forgetting cat"
puts [ "ant", "bat", "cat"].to_sentence(:skip_last_comma => true)
```

```
                                        #=> "ant, bat and cat"
[1,2,3,4,5,6,7].in_groups_of(3) {|slice| puts slice.inspect}
                                        #=> [1, 2, 3]
                                            [4, 5, 6]
                                            [7, nil, nil]
[1,2,3,4,5,6,7].in_groups_of(3, "X") {|slice| puts slice.inspect}
                                        #=> [1, 2, 3]
                                            [4, 5, 6]
                                            [7, "X", "X"]
```

Extensions de la classe String

Les débutants en Ruby sont souvent surpris qu'un index du type `string[2]` dans une chaîne de caractères retourne un entier et non une chaîne d'un caractère. Rails ajoute quelques méthodes d'assistance aux manipulations de chaînes, leur donnant un comportement plus naturel.

```
string = "Now is the time"
puts string.at(2)       #=> "w"
puts string.from(8)     #=> "he time"
puts string.to(8)       #=> "Now is th"
puts string.first       #=> "N"
puts string.first(3)    #=> "Now"
puts string.last        #=> "e"
puts string.last(4)     #=> "time"
puts string.starts_with?("No")  #=> true
puts string.ends_with?("ME")    #=> false
count = Hash.new(0)
string.each_char {|ch| count[ch] += 1}
puts count.inspect      #=> {" "=>3, "w"=>1, "m"=>1, "N"=>1, "o"=>1,
                             "e"=>2, "h"=>1, "s"=>1, "t"=>2, "i"=>2}
```

Active Support ajoute de nouvelles méthodes aux objets chaînes de caractères pour faciliter la conversion des noms du singulier au pluriel, des lettres minuscules en casse mixte, etc. Parmi celles-ci, deux sont plus particulièrement utiles dans les applications.

```
puts "cat".pluralize            #=> cats
puts "cats".pluralize           #=> cats
puts "erratum".pluralize        #=> errata
puts "cats".singularize         #=> cat
```

```
puts "errata".singularize        #=> erratum
puts "first_name".humanize       #=> "First name"
puts "now is the time".titleize  #=> "Now Is The Time"
```

Écrire vos règles d'inflexions

Rails s'accompagne d'une série de règles conséquentes destinées à la pluralisation des mots anglais mais il ne connaît pas (encore) toutes les formes de singulier irrégulières. Par exemple, si vous écrivez une application agricole et que vous concevez une table `geese` (*oies* en anglais), Rails pourrait bien ne pas la trouver automatiquement.

```
depot> ruby script/console
Loading development environment.
>> "goose".pluralize
=> "gooses"
```

Cela donne *gooses* , une forme verbale improbable mais pas un nom au pluriel.

Mais comme partout dans Rails, si vous n'aimez pas les valeurs par défaut, vous pouvez les modifier. Changer une inflexion automatique est très simple. En bas du fichier *`environment.rb`* situé sous *`config`*, vous trouverez une section commentée pour la configuration du module d'inflexion. Il vous permet de définir de nouvelles règles de mise au singulier ou au pluriel des mots. On peut lui indiquer :

- La forme plurielle d'un mot ou d'une classe de mots en fonction de leur forme singulière.
- La forme singulière d'un mot ou d'une classe de mots en fonction de leur forme plurielle.
- Les mots ayant des pluriels irréguliers.
- Les mots n'ayant pas de forme plurielle.

Notre exemple `goose/geese` présente un pluriel irrégulier, nous allons donc informer l'inflecteur sur la façon de traiter ce cas particulier.

```
Inflector.inflections do |inflect|
  inflect.irregular "goose", "geese"
end
```

Maintenant, Rails est au courant.

```
depot> ruby script/console
Loading development environment.
>> "goose".pluralize               #=> "geese"
>> "geese".singularize             #=> "goose"
```

C'est peut être surprenant mais la définition d'une forme plurielle irrégulière entraîne son extension à tous les mots ayant la même terminaison.

```
>> "canadagoose".pluralize       #=> "canadageese"
>> "wildgeese".singularize       #=> "wildgoose"
```

Pour des familles de pluriel, vous devez définir des règles pour les formes au singulier et au pluriel. Par exemple, le pluriel de *father-in-law* est *fathers-in-law*, *mother-in-law* devient *mothers-in-law*, etc. Il est possible d'indiquer à Rails ce qu'il doit faire à l'aide d'expressions régulières. Dans ce cas, il faut lui indiquer comment obtenir le pluriel à partir du singulier mais également l'inverse.

```
Inflector.inflections do |inflect|
  inflect.plural(/-in-law$/, "s-in-law")
  inflect.singular(/s-in-law$/, "-in-law")
end
>> "sister-in-law".pluralize       #=> "sisters-in-law"
>> "brothers-in-law".singularize  #=> "brother-in-law"
```

Certains mots ne sont pas dénombrables et n'ont donc pas de forme plurielle. Il est nécessaire de le préciser à l'inflecteur à l'aide de la méthode `uncountable()`.

```
Inflector.inflections do |inflect|
  inflect.uncountable("air", "information", "water")
end
>> "water".pluralize         #=> "water"
>> "water".singularize       #=> "water"
```

Dans une application Rails, ces changements s'appliquent dans le fichier *`environment.rb`* du répertoire *`config`*.

Une remarque importante : si vous nommez les classes de vos modèles en français, le passage à la forme plurielle fonctionnera correctement dans la plupart des cas puisque le français, comme l'anglais, ajoutent un « s » à la fin des formes plurielles des noms. Cependant, prenez bien garde que dans sa version actuelle Rails ne connaît pas les règles de passage au pluriel pour les cas irréguliers du français. Ainsi, si la classe de votre modèle s'appelle `Journal`, la table correspondante dans la base de données devra s'appeler `journals` et non pas `journaux`, sauf si le terme en question à fait l'objet d'une nouvelle règle d'inflection comme expliqué précédemment. Dans sa version standard, Rails ne connaît que les formes irrégulières de l'anglais.

Extensions aux nombres

Les nombres entiers (la classe `Fixnum`) gagnent deux méthodes d'instance pour les tests de parité `even?` (pair) et `odd?` (impair). Il est également possible d'obtenir la forme ordinale d'un entier par l'utilisation de la méthode `ordinalize()`.

```
puts 3.ordinalize     #=> "3rd"
puts 321.ordinalize   #=> "321st"
```

Tous les objets numériques profitent d'une nouvelle série de méthodes pour la conversion d'unité. Singulier et pluriel sont supportés.

```
puts 20.bytes       #=> 20
puts 20.kilobytes   #=> 20480
puts 20.megabytes   #=> 20971520
puts 20.gigabytes   #=> 21474836480
puts 20.terabytes   #=> 21990232555520
puts 20.petabytes   #=> 22517998136852480
puts 1.exabyte      #=> 1152921504606846976
```

Il en existe d'autres pour les unités de temps qui convertissent le receveur du message (le nombre de départ) en un nombre équivalent de secondes. Les méthodes `months()` (mois) et `years()` (années) sont des approximations : les mois sont supposés avoir 30 jours et les années 365. Toutefois, la classe `Time` a elle-même été étendue par des méthodes permettant d'obtenir des dates relativement précises (voir les explications à la section suivante). Les formes singulières et plurielles sont ici aussi prises en charge.

```
puts 20.seconds    #=> 20
puts 20.minutes    #=> 1200
puts 20.hours      #=> 72000
puts 20.days       #=> 1728000
puts 20.weeks      #=> 12096000
puts 20.fortnights #=> 24192000
puts 20.months     #=> 51840000
puts 20.years      #=> 630720000
```

Vous pouvez également calculer des temps relatifs à la date courante (`Time.now`) en utilisant les méthodes `ago()` (avant) et `from_now()` (après) (ou leurs noms d'alias respectifs `until()` et `since()`).

```
puts Time.now            #=> Thu May 18 23:29:14 CDT 2006
puts 20.minutes.ago      #=> Thu May 18 23:09:14 CDT 2006
puts 20.hours.from_now   #=> Fri May 19 19:29:14 CDT 2006
```

```
puts 20.weeks.from_now   #=> Thu Oct 05 23:29:14 CDT 2006
puts 20.months.ago       #=> Sat Sep 25 23:29:16 CDT 2004
puts 20.minutes.until("2006-12-25 12:00:00".to_time)
                         #=> Mon Dec 25 11:40:00 UTC 2006
puts 20.minutes.since("2006-12-25 12:00:00".to_time)
                         #=> Mon Dec 25 12:20:00 UTC 2006
```

Pratique, non ? Et ce n'est pas fini...

Extensions des classes Time et Date

La classe `Time` se voit complétée de quelques méthodes très utiles permettant de calculer une date relativement à une autre ou d'en formater les chaînes. Plusieurs de ces méthodes ont des alias : consultez la documentation de l'API pour plus de détails.

```
now =  Time.now
puts now                          #=> Thu May 18 23:36:10 CDT 2006
puts now.to_date                  #=> 2006-05-18
puts now.to_s                     #=> Thu May 18 23:36:10 CDT 2006
puts now.to_s(:short)             #=> 18 May 23:36
puts now.to_s(:long)              #=> May 18, 2006 23:36
puts now.to_s(:db)                #=> 2006-05-18 23:36:10
puts now.to_s(:rfc822)            #=> Thu, 18 May 2006 23:36:10 -0500
puts now.ago(3600)                #=> Thu May 18 22:36:10 CDT 2006
puts now.at_beginning_of_day      #=> Thu May 18 00:00:00 CDT 2006
puts now.at_beginning_of_month    #=> Mon May 01 00:00:00 CDT 2006
puts now.at_beginning_of_week     #=> Mon May 15 00:00:00 CDT 2006
puts now.at_beginning_of_quarter  #=> Sat Apr 01 00:00:00 CST 2006
puts now.at_beginning_of_year     #=> Sun Jan 01 00:00:00 CST 2006
puts now.at_midnight              #=> Thu May 18 00:00:00 CDT 2006
puts now.change(:hour => 13)      #=> Thu May 18 13:00:00 CDT 2006
puts now.last_month               #=> Tue Apr 18 23:36:10 CDT 2006
puts now.last_year                #=> Wed May 18 23:36:10 CDT 2005
puts now.midnight                 #=> Thu May 18 00:00:00 CDT 2006
puts now.monday                   #=> Mon May 15 00:00:00 CDT 2006
puts now.months_ago(2)            #=> Sat Mar 18 23:36:10 CST 2006
puts now.months_since(2)          #=> Tue Jul 18 23:36:10 CDT 2006
puts now.next_week                #=> Mon May 22 00:00:00 CDT 2006
puts now.next_year                #=> Fri May 18 23:36:10 CDT 2007
puts now.seconds_since_midnight   #=> 84970.423472
puts now.since(7200)              #=> Fri May 19 01:36:10 CDT 2006
puts now.tomorrow                 #=> Fri May 19 23:36:10 CDT 2006
```

```
puts now.years_ago(2)            #=> Tue May 18 23:36:10 CDT 2004
puts now.years_since(2)          #=> Sun May 18 23:36:10 CDT 2008
puts now.yesterday               #=> Wed May 17 23:36:10 CDT 2006
puts now.advance(:days => 30)    #=> Sat Jun 17 23:36:10 CDT 2006
puts Time.days_in_month(2)       #=> 28
puts Time.days_in_month(2, 2000) #=> 29
```

Les objets `Date` gagnent également quelques méthodes utiles.

```
date = Date.today
puts date.to_s                   #=> "2006-05-18"
puts date.to_time                #=> Thu May 18 00:00:00 CDT 2006
puts date.to_s(:short)           #=> "18 May"
puts date.to_s(:long)            #=> "May 18, 2006"
puts date.to_s(:db)              #=> "2006-05-18"
```

La dernière de celles-ci convertit une date en une chaîne de caractères acceptable par la base de données utilisée par défaut dans votre application. Vous aurez peut-être remarqué que la classe `Time` possède une extension similaire destinée au formatage des données temporelles dans un format de base de données spécifique.

Vous pouvez ajouter vos propres extensions de formatage temporel ou de date. Par exemple, votre application nécessitera peut être l'affichage de dates sous une forme ordinale (le nombre de jours dans une année). Les librairies `Date` et `Time` de Ruby supportent toutes les deux la méthode `strftime()` vous permettant d'utiliser un formatage comme :

```
>> d = Date.today
=> #<Date: 4907769/2,0,2299161>
>> d.to_s
=> "2006-05-29"
>> d.strftime("%y-%j")
=> "06-149"
```

Mais il se peut qu'au lieu de ceci, vous souhaitiez encapsuler ce formatage par une extension de la méthode de date `to_s`. Dans votre fichier *`environment.rb`*, ajoutez une ligne comme celle-ci.

```
ActiveSupport::CoreExtensions::Date::Conversions::DATE_FORMATS.merge!(
  :ordinal => "%Y-%j"
)
```

Vous pouvez maintenant écrire :

```
any_date.to_s(:ordinal)     #=>  "2006-149"
```

Il est également possible d'étendre la classe `Time` dans le même sens.

```
ActiveSupport::CoreExtensions::Time::Conversions::DATE_FORMATS.merge!(
  :chatty => "It's %I:%M%p on %A, %B %d, %Y"
)
Time.now.to_s(:chatty)   #=> "It's 12:49PM on Monday, May 29, 2006"
```

Il existe de plus deux méthodes très utiles qui sont des extensions de la classe `String`. Les méthodes `to_time()` et `to_date()` retournent respectivement des objets `Time` ou `Date`.

```
puts "2006-12-25 12:34:56".to_time   #=> Mon Dec 25 12:34:56 UTC 2006
puts "2006-12-25 12:34:56".to_date   #=> 2006-12-25
```

Active Support inclut également une classe `TimeZone` dont les objets encapsulent les noms et décalages des fuseaux horaires. Cette classe contient une liste des fuseaux horaires mondiaux. Consultez Active Support RDoc pour plus de détails.

Une extension des symboles Ruby

Cette section décrit une fonctionnalité avancée de Ruby que vous pouvez passer sans problème lors des premières lectures.

Nous utilisons très souvent des itérations, en fait partout où un bloc invoque une méthode sur l'argument qu'on lui passe. C'est ce que nous avons fait précédemment dans les exemples de `group_by` et `index_by`.

```
groups = posts.group_by {|post| post.author_id}
```

Rails utilise une notation abrégée pour cela et nous aurions pu écrire ce code :

```
groups = posts.group_by(&:author_id)
```

De la même manière, le code

```
us_states = State.find(:all)
state_lookup = us_states.index_by {|state| state.short_name}
```

aurait pu s'écrire :

```
us_states = State.find(:all)
state_lookup = us_states.index_by(&:short_name)
```

Mais comment fonctionne ce sortilège ? Il repose sur le fait que la notation `&` pour le passage de paramètre s'attend à trouver un objet `Proc`. S'il n'est pas trouvé, Ruby essaye de convertir ce qu'il peut en invoquant sa méthode `to_proc()`. Dans notre exemple, nous lui passons un symbole (`:author_id`) pour lequel Rails a confortablement défini une méthode `to_proc()` dans la classe `Symbol`. Comprendre comment cela fonctionne est laissé en exercice au lecteur. En voici toutefois l'implémentation :

```
class Symbol
  def to_proc
    Proc.new { |obj, *args| obj.send(self, *args) }
  end
end
```

with_options()

De nombreuses méthodes de Rails utilisent un tableau associatif d'options en dernier paramètre. Vous serez parfois obligé d'utiliser ces méthodes à répétition, et sur plusieurs lignes d'affilée dès lors que vos appels auront une ou plusieurs options en commun. Par exemple, vous pourriez définir des routes.

```
ActionController::Routing::Routes.draw do |map|
  map.connect "/shop/summary",   :controller => "store",
                                 :action => "summary"
  map.connect "/titles/buy/:id", :controller => "store",
                                 :action => "add_to_cart"
  map.connect "/cart",           :controller => "store",
                                 :action => "display_cart"
end
```

La méthode `with_options()` vous permet de résumer la définition de cette option commune à une ligne.

```
ActionController::Routing::Routes.draw do |map|
  map.with_options(:controller => "store") do |store_map|
    store_map.connect "/shop/summary", :action => "summary"
    store_map.connect "/titles/buy/:id", :action => "add_to_cart"
```

```
    store_map.connect "/cart", :action => "display_cart"
  end
end
```

Dans cet exemple, `store_map` se comporte comme un objet `map`, mais l'option `controller =>store` sera ajoutée à sa liste d'options chaque fois qu'il est appelé.

La méthode `with_options` peut être utilisée avec n'importe quel appel d'API dans la mesure où le dernier paramètre est un tableau associatif.

Support Unicode

Autrefois, les caractères étaient représentés par des séquences de 6, 7 ou 8 bits. Chaque constructeur informatique choisissait son format parmi ces séquences et décidait des représentations de caractères. Peu à peu, des standards commencèrent à émerger et des encodages tels que ASCII et EBCDIC devinrent populaires. Toutefois, même avec ces standards, vous ne pouviez jamais être sûr qu'un schéma de bit donné correspondrait à un caractère précis : le caractère ASCII sur 7 bits 0b0100011 donnait # sur les terminaux américains et § sur les terminaux britanniques. Les bricolages, comme les pages de code, qui superposent plusieurs couches de caractères sur un schéma de bit, résolvaient les problèmes localement mais les démultipliaient globalement.

Dans le même temps, il apparut rapidement que le schéma 8 bits n'était tout simplement pas suffisant à l'encodage des caractères de nombreux langages. La conférence Unicode fut créée dans le but de répondre à ce problème[1].

Unicode définit un certain nombre de schémas d'encodage qui autorisent jusqu'à 32 bits par représentation de caractère. Unicode est généralement enregistré en utilisant l'un des trois schémas d'encodage possible. Dans l'un d'entre eux, UTF-32, chaque caractère (techniquement un point de code) est représenté par une valeur 32-bit. Dans les deux autres (UTF-16 et UTF-8), les caractères sont représentés par une ou plusieurs valeurs 16-bit ou 8-bit. Quand Rails enregistre des chaînes de caractères en Unicode, il utilise UTF-8.

Le langage Ruby, sous-jacent à Rails, est originaire du Japon et il se trouve que, historiquement, les programmeurs japonais ont eu des problèmes avec l'encodage de leur langage en Unicode. Ce qui signifie que, même si Ruby supporte les chaînes de caractères encodées en Unicode, il ne les supporte pas vraiment dans ses librairies. Par exemple, la représentation UTF-8 du « ü » correspond à la séquence sur deux octets : c3 bc (la notation hexadécimale permet de montrer les valeurs binaires). Mais si vous passez à Ruby une chaîne contenant le caractère ü, ses librairies seront incapables de deviner que deux octets sont utilisés pour la représentation d'un seul caractère.

1. http://www.unicode.org

```
dave> irb
irb(main):001:0> name = "Günter"
=> "G\303\274nter"
irb(main):002:0> name.length
=> 7
```

Si le prénom Günter affiche six caractères, sa représentation compte sept octets, et c'est d'ailleurs le nombre que Ruby renvoie.

Rails 1.2 apporte une solution partielle à ce problème. Toutefois, ne s'agissant pas d'un remplacement des librairies Ruby, il reste des endroits où des choses inattendues peuvent survenir. Mais malgré cela, la nouvelle librairie Multibyte de Rails, ajoutée à Active Support en septembre 2006, fait un pas considérable en avant vers l'intégration facile du traitement Unicode dans les applications Rails.

Plutôt que de remplacer les méthodes de chaînes de caractères des librairies de Ruby par des versions reconnaissant Unicode, la librairie Multibyte définit une nouvelle classe, baptisée `Chars`, pourvue de méthodes identiques à celles de la classe `String`, sauf qu'elles sont au courant de l'encodage sous-jacent aux chaînes.

La règle d'utilisation des chaînes Multibyte est simple : chaque fois que vous souhaitez travailler avec des chaînes encodées en UTF-8, convertissez premièrement ces chaînes en objets `Chars`. La librairie propose la méthode `chars()` à toutes les chaînes pour faciliter le travail.

Expérimentons ceci par l'intermédiaire de `script/console`.

```
1   dave> script/console
-   Loading development environment.
-   >> name = "G\303\274nter"
-   => "Günter"
5   >> name.length
-   => 7
-   >> name.chars.length
-   => 6
-   >> name.reverse
10  => "retn\274?G"
-   >> name.chars.reverse
-   => #<ActiveSupport::Multibyte::Chars:0x2c4cdf4 @string="retnüG">
```

Nous commençons donc par affecter une chaîne contenant des caractères UTF-8 à la variable `name`.

À la ligne 5, nous demandons à Ruby la longueur de la chaîne et il retourne 7, le nombre d'octets correspondant à la représentation. Puis, à la ligne 7, nous utilisons la méthode `chars` pour créer un objet `Chars` qui encapsule la chaîne précédente. En demandant sa longueur à ce nouvel objet, nous obtenons 6, le nombre de caractères de la chaîne.

De même, l'inversion de la chaîne initiale ne produit qu'un résultat incompréhensible en renversant simplement l'ordre des bits alors que d'un autre côté, l'inversion de l'objet `Chars` produit le résultat escompté.

En théorie, toutes les librairies internes de Rails sont maintenant à l'épreuve d'Unicode. Cela signifie, par exemple, que `validates_length_of` (valide la longueur de) vérifiera correctement la longueur des chaînes UTF-8 pour peu que le support UTF-8 soit activé dans votre application.

Toutefois, une gestion correcte des chaînes de caractères encodées n'est pas suffisante pour s'assurer que votre application fonctionne avec des caractères Unicode. Vous devrez vous assurer que, sur l'intégralité du chemin parcouru du navigateur à la base de données, tous s'entendent sur un encodage commun. Explorons ce point en écrivant une application très simple permettant la construction d'une liste de noms.

Exemple d'application utilisant Unicode

Vous allez donc écrire une application simple permettant d'afficher une liste de noms sur une page. Un champ d'entrée texte sur cette même page vous permettra d'ajouter de nouveaux noms à la liste. La liste complète des noms sera enregistrée dans une table de base de données.

Créons tout d'abord une application Rails.

```
dave> rails namelist
dave> cd namelist
namelist> ruby script/server
```

Vous devez maintenant créer votre base de données. Toutefois, il faut également s'assurer que le jeu de caractères par défaut pour cette base de données sera bien UTF-8. La manière d'y parvenir dépend de la base de données. Voici ce qu'il faut faire pour une base MySQL[1].

```
namelist>  mysql -u root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 85 to server version: 5.0.22
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> create database namelist_development character set utf8;
Query OK, 1 row affected (0.00 sec)
```

1. Normalement, nous devrions utiliser **`mysqladmin`** pour créer les bases de données. Toutefois, l'option **`--default-character-set`** semble ne pas fonctionner correctement.

La ligne 5 indique à la base de données quel encodage de caractères utiliser. De façon plus surprenante, nous devons également indiquer à chaque connexion MySQL l'encodage qu'il devra utiliser. Cette option d'encodage se précise dans le fichier *database.yml* (nous ne verrons ici que la section développement : vous devrez répéter l'opération pour les bases **test** et **production**).

code/e1/namelist/config/database.yml

```
development:
  adapter: mysql
  database: namelist_development
  username: root
  password:
  host: localhost
  encoding: utf8
```

Nous allons maintenant créer un modèle pour nos noms,

```
namelist> script/generate model person
```

puis créer une `Migration`.

code/e1/namelist/db/migrate/001_create_people.rb

```
class CreatePeople < ActiveRecord::Migration
  def self.up
    create_table :people do |t|
      t.column :name, :string
    end
  end
  def self.down
    drop_table :people
  end
end
```

Comme nous avons réglé le jeu de caractères par défaut de la base de données sur UTF, nous n'aurons rien à ajouter de spécial dans le fichier migration. Si nous n'avions pas pu régler cette option au niveau de la base de données, il aurait fallu le faire maintenant, table par table dans la migration, comme dans cet exemple :

```
create_table :people, :options => 'default charset=utf8' do |t|
  t.column :name, :string
end
```

Cependant, cette opération rend la migration spécifique à MySQL et il en résulte que les options de tables ne seront pas répercutées dans la base de données de test tant que vous n'aurez pas modifié le schéma par défaut de *environment.rb* à `:sql`. Ce procédé fastidieux vous suggère qu'il est préférable de choisir le jeu de caractères au niveau de la base de données ; c'est la meilleure façon de procéder.

Nous allons maintenant écrire notre contrôleur et notre vue. Le contrôleur sera très simple car il ne contiendra qu'une action.

code/e1/namelist/app/controllers/people_controller.rb

```
class PeopleController < ApplicationController
  def index
    @person = Person.new(params[:person])
    @person.save! if request.post?
    @people = Person.find(:all)
  end
end
```

Comme nous avons rendu la base de données compatible avec Unicode, il ne nous reste plus qu'à faire de même du côté du navigateur.

Depuis Rails 1.2, l'en-tête type de contenu (*content-type header*) par défaut est :

```
Content-Type: text/html; charset=UTF-8
```

Toutefois, pour en être certains, nous allons également ajouter une balise `<meta>` dans l'en-tête de page pour forcer UTF-8. Ce qui signifie qu'un utilisateur sauvant la page dans un fichier local pourra la relire correctement plus tard. Notre fichier de mise en page correspondra à ceci :

code/e1/namelist/app/views/layouts/people.rhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8"></meta>
    <title>My Name List</title>
  </head>
  <body>
    <%= yield :layout %>
  </body>
</html>
```

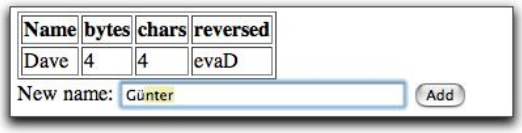
Dans la vue index, nous allons afficher la liste complète des noms contenus dans la base de données et proposer un formulaire simplifié pour permettre aux gens d'entrer de nouveaux noms. Dans la liste, nous afficherons le nom, sa taille en octets ainsi que sa taille en caractères, et pour finir, en exagérant un peu, nous renverserons le nom.

code/e1/namelist/app/views/people/index.rhtml

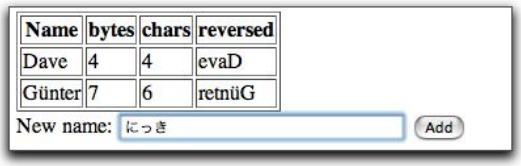
```
<table border="1">
  <tr>
    <th>Name</th><th>bytes</th><th>chars</th><th>reversed</th>
  </tr>
  <% for person in @people %>
    <tr>
      <td><%= h(person.name) %>
      <td><%= person.name.length %></td>
      <td><%= person.name.chars.length %></td>
      <td><%= h(person.name.chars.reverse) %></td>
    </tr>
  <% end %>
</table>
<% form_for :person do |form| %>
  New name: <%= form.text_field :name %>
  <%= submit_tag "Add" %>
<% end %>
```

Si vous pointez votre navigateur sur votre contrôleur `people`, vous découvrez une table vide. Il est maintenant temps de taper Dave dans le champ de formulaire.

En pressant sur le bouton Add, nous voyons que la chaîne « Dave » contient à la fois quatre octets et quatre caractères : les caractères ASCII normaux valent 1 octet en UTF-8.

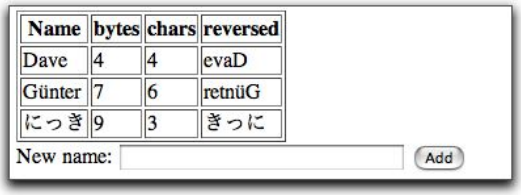


En pressant sur Add, après avoir entré « Günter », ce que nous voyons est différent.



Parce que le caractère « ü » demande deux octets pour une représentation en UTF-8, nous voyons ici que la chaîne a une longueur de sept octets et une longueur de six caractères. Vous remarquerez que la forme inversée de la chaîne s'affiche correctement.

Pour finir, nous allons ajouter du texte japonais.



À présent, la disparité entre la longueur en octets et la longueur en caractères est encore plus grande. Néanmoins, la chaîne se renverse toujours correctement, caractère après caractère.