6

Работа с ActiveRecord

Объект, обертывающий строку таблицы или представления базы данных, инкапсулирует доступ к базе и добавляет к данным логику предметной области.

Мартин Фаулер,
«Архитектура корпоративных программных приложений»

Паттерн ActiveRecord, выявленный Мартином Фаулером в основополагающей книге *Patterns of Enterprise Architecture*[1], отображает один класс предметной области на одну таблицу базы данных, а один экземпляр класса – на одну строку таблицы. Хотя этот простой подход применим и не во всех случаях, он обеспечивает удобную среду для доступа к базе данных и сохранения в ней объектов.

Среда ActiveRecord в Rails включает механизмы для представления моделей и их взаимосвязей, операций CRUD (Create, Read, Update, Delete), сложных поисков, контроля данных, обратных вызовов и многого другого. Она опирается на принцип «примата соглашения над конфигурацией», поэтому проще всего ее применять, когда уже на этапе создания схемы новой базы данных вы следуете определенным соглашениям. Однако ActiveRecord предоставляет и средства конфигурирования, позволяющие адаптировать его к унаследованным базам данных, в которых соглашения Rails не применялись.

---

[1] Мартин Фаулер «Архитектура корпоративных программных приложений», Вильямс, 2007 год.

В основном докладе на конференции, посвященной рождению Rails, в 2006 году, Мартин Фаулер сказал, что в Ruby on Rails паттерн Active Record внедрен настолько глубоко, насколько никто не мог и предполагать. На этом примере показано, чего можно добиться, если всецело посвятить себя достижению идеала, в качестве которого в Rails выступает простота.

Основы

Для полноты начнем с изложения самых основ работы ActiveRecord. Первое, что нужно сделать при создании нового класса модели, – объявить его как подкласс `ActiveRecord::Base`, применив синтаксис расширения Ruby:

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

По принятому в ActiveRecord соглашению класс `Client` отображается на таблицу `clients`. О том, как Rails понимает, что такое множественное число, см. раздел «Приведение к множественному числу» ниже. По тому же соглашению, ActiveRecord ожидает, что первичным ключом таблицы будет колонка с именем `id`. Она должна иметь целочисленный тип, а сервер должен автоматически инкрементировать ключ при создании новой записи. Отметим, что в самом классе нет никаких упоминаний об имени таблицы, а также об именах и типах данных колонок.

Каждый экземпляр класса ActiveRecord обеспечивает доступ к данным одной строки соответствующей таблицы в объектно-ориентированном стиле. Колонки строки представляются в виде атрибутов объекта; при этом применяются простейшие преобразования типов (то есть типу varchar соответствует строка Ruby, типам даты/времени – даты Ruby и т. д.). Проверка наличия значения по умолчанию не производится. Типы атрибутов выводятся из определения колонок в таблице, ассоциированной с классом. Добавление, удаление и изменение самих атрибутов или их типов реализуется изменением описания таблицы в схеме базы данных.

Если сервер Rails запущен в *режиме разработки*, то изменения в схеме базы данных отражаются на объектах ActiveRecord немедленно, и это видно в веб-броузере. Если же изменения внесены в схему в режиме работы с консолью Rails, то они автоматически *не* подхватываются, но это можно сделать вручную, набрав на консоли команду `reload!`.

Путь Rails состоит в том, чтобы генерировать, а не писать трафаретный код. Поэтому вам почти никогда не придется ни создавать файл для своего класса модели, ни вводить его объявление. Гораздо проще воспользоваться для этой цели встроенным в Rails генератором моделей.

Например, позволим генератору моделей создать класс `Client` и посмотрим, какие в результате появятся файлы:

```
$ script/generate model client
      exists app/models/
      exists test/unit/
      exists test/fixtures/
      create app/models/client.rb
      create test/unit/client_test.rb
      create test/fixtures/clients.yml
      exists db/migrate
      create db/migrate/002_create_clients.rb
```

Файл, в котором находится новый класс модели, называется client.rb:

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

Просто и красиво. Посмотрим, что еще было создано. Файл **client_test.rb** содержит заготовку для автономных тестов:

```
require File.dirname(__FILE__) + '/../test_helper'

class ClientTest < Test::Unit::TestCase
  fixtures :clients

  # Заменить настоящими тестами.
  def test_truth
    assert true
  end
end
```

Комментарий предлагает заменить метод `test_truth` настоящими тестами. Но пока мы просто знакомимся со сгенерированным кодом, поэтому пойдем дальше. Отметим, что класс `ClientTest` ссылается на файл *фикстуры* (fixture) **clients.yml**:

```
# О фикстурах см. http://ar.rubyonrails.org/classes/Fixtures.html
one:
  id: 1
two:
  id: 2
```

Какие-то идентификаторы... Автономные тесты и фикстуры рассматриваются в главе 17 «Тестирование».

И наконец, имеется файл миграции с именем 002_create_clients.rb:

```
class CreateClients < ActiveRecord::Migration
  def self.up
    create_table :clients do |t|
      # t.column :name, :string
    end
  end
```

```
  def self.down
    drop_table :clients
  end
end
```

Механизм миграций в Rails позволяет создавать и развивать схему базы данных, без него вы не получили бы никаких моделей ActiveRecord (точнее, они были бы очень скучными). Коли так, рассмотрим миграции более подробно.

Говорит Кортенэ...

ActiveRecord – прекрасный пример «Золотого пути» Rails. Это означает, что, оставаясь в рамках наложенных ограничений, можно зайти очень далеко. Но стоит свернуть в сторону, и вы, скорее всего, завязнете в грязи. Золотой путь подразумевает соблюдение ряда соглашений, в частности, присваивание таблицам имен во множественном числе (users).

Разработчики, недавно открывшие для себя Rails, а также приверженцы конкурирующих веб-платформ ругаются, что их заставляют именовать таблицы определенным образом, на уровне базы данных нет никаких ограничений, обработка внешних ключей абсолютно неправильна, в системах масштаба предприятия первичные ключи должны быть составными, и т. д. и т. п.

Но перестаньте хныкать – все это не более чем умолчания, которые можно переопределить в одной строке кода или с помощью подключаемого модуля.

Миграции

Никуда не уйти от того факта, что со временем схема базы данных эволюционирует. Добавляются новые таблицы, изменяются имена колонок, что-то удаляется – в общем, вы понимаете, о чем я. Если разработчики не готовы строго соблюдать соглашения и придерживаться определенной дисциплины, то синхронизация схемы базы данных с кодом приложений традиционно становится очень трудоемкой задачей.

Миграции в Rails помогают развивать схему базы данных приложения (ее еще называют DDL-описанием[1]) без уничтожения и нового создания базы после каждого изменения. А это означает, что вы не теряете данные, появившиеся в процессе разработки. Быть может, иногда это

[1] DDL (Data Definition Language) – язык определения данных. – *Примеч. перев.*

и не так важно, но обычно очень удобно. При выполнении миграции достаточно описать изменения, необходимые для перехода от одной версии схемы к другой – следующей или *предыдущей*.

Создание миграций

Rails предлагает генератор для создания миграций. Вот текст справки по нему[1]:

```
$ script/generate migration
Порядок вызова: script/generate migration MigrationName [флаги]
Информация о Rails:
-v, --version           Вывести номер версии Rails и завершиться.
-h, --help              Вывести это сообщение и завершиться.
Общие параметры:
-p, --pretend           Выполнять без внесения изменений.
-f, --force             Перезаписывать существующие файлы.
-s, --skip              Пропускать существующие файлы.
-q, --quiet             Подавить нормальную печать.
-t, --backtrace         Отладка: выводить трассировку стека в случае ошибок.
-c, --svn               Модифицировать файлы в системе subversion.
                        (Примечание: команда svn должна находиться по одному
                        из просматриваемых путей.)
Описание:
    Генератор миграций создает заглушку для новой миграции базы данных.
    В качестве аргумента передается имя миграции. Имя может быть задано
    в ВерблюжьейНотации или с_подчерками.
    Генератор создает класс миграции в каталоге db/migrate, указывая в начале
    имени порядковый номер.
Пример:
    ./script/generate migration AddSslFlag
    Если уже существуют 4 миграции, то для миграция AddSslFlag будет создан
файл db/migrate/005_add_ssl_flag.rb.
```

Как видите, от вас требуется только задать понятное имя миграции в CamelCase[2], а генератор сделает все остальное. Напрямую мы вызываем генератор только при необходимости изменить атрибуты колонок в существующей таблице.

---

[1] Оригинальный генератор печатает справку на английском языке. Для удобства читателя она переведена. – *Примеч. перев.*

[2] Нотация CamelCase или camelCase (вариативность написания заглавных букв существенна) получила название ВерблюжьейНотации. Имена идентификаторов, состоящие из нескольких слов, записываются так, что слова не разделяются никакими знаками, но каждое слово начинается с заглавной буквы, например dateOfBirth или DateOfBirth. При этом первое слово может начинаться с заглавной или строчной буквы – в зависимости от соглашения. Визуально имеется ряд «горбов», как у верблюда. Отсюда и английское название. – *Примеч. перев.*

Как уже отмечалось выше, другие генераторы, в частности генератор моделей, тоже создают сценарии миграции, если только не указан флаг `--skip-migration`.

Именование миграций

Последовательность миграций определяется простой схемой нумерации, отраженной в именах файлов; генератор миграций формирует порядковые номера автоматически.

По соглашению, имя файла начинается с трехзначного номера версии (дополненного слева нулями), за которым следует имя класса миграции, отделенное знаком подчерка. (Примечание: имя файла *обязано* точно соответствовать имени класса, иначе процедура миграции закончится с ошибкой.)

Генератор миграций определяет порядковый номер следующей миграции, справляясь со специальной таблицей в базе данных, за которую отвечает Rails. Она называется `schema_info` и имеет очень простую структуру:

```
mysql> desc schema_info;
+---------+---------+------+-----+---------+-------+
| Field   | Type    | Null | Key | Default | Extra |
+---------+---------+------+-----+---------+-------+
| version | int(11) | YES  |     | NULL    |       |
+---------+---------+------+-----+---------+-------+
1 row in set (0.00 sec)
```

Таблица содержит всего одну колонку и одну строку, в которой хранится текущий номер миграции приложения.

Содержательная часть имени миграции оставлена на ваше усмотрение, но многие известные мне разработчики Rails стараются отражать в нем операцию над схемой (в простых случаях) или хотя бы намекнуть, для чего миграция предназначена (в более сложных).

Подвохи миграций

Если вы пишете свои программы для Rails в одиночку, то у схемы порядковой нумерации имен никаких подвохов нет, так что можете смело пропустить этот раздел. Проблемы начинаются, когда над одним проектом работают несколько программистов, особенно большие коллективы. Речь идет не о проблемах миграции API самой среды Rails – именно сопровождение изменяющейся схемы базы данных представляет собой сложную и пока не до конца решенную задачу.

Вот что пишет о проблемах миграции, с которыми приходилось сталкиваться, мой старый приятель по компании ThoughtWorks Джей Филдс (Jay Fields), не раз возглавлявший большие коллективы разработчиков на платформе Rails, в своем блоге:

Миграции – это прекрасно, но за них приходится расплачиваться. При работе в большом коллективе (а моя теперешняя команда состоит из 14 человек и продолжает расти) случаются конфликты миграций. Проблему можно отчасти решить с помощью договоренностей, но нет сомнений, что миграции могут стать узким местом. Кроме того, сам процесс создания миграции в большой команде может протекать болезненно. Прежде чем создавать новую миграцию, вы должны убедиться, что коллеги сохранили свои миграции в системе управления версиями. Далее наилучшее развитие событий – создать миграцию, которая ничего не изменяет, и сразу же поставить ее на учет. Это гарантирует, что вы не помешаете создавать миграции другим членам команды, однако не всегда можно ограничиться одним лишь добавлением таблицы. Если миграция изменяет структуру базы данных, то часто некоторые тесты перестают работать. Очевидно, не следует ставить миграцию на учет, пока все тесты не заработают нормально, но на это может потребоваться время. В течение всего этого времени больше никто не сможет вносить изменения в базу данных[1].

По-другому та же проблема проявляется, когда требуется внести изменения сразу в несколько ветвей программы. Это вообще оказывается кошмаром (дополнительную информацию на эту тему см. в комментариях к процитированной записи в блоге Джея).

К сожалению, у данной проблемы нет простого решения, разве что спроектировать базу от начала до конца еще до реализации приложения. Но на этом пути куча своих проблем (настолько много, что мы даже затрагивать их не будем). Могу лишь по собственному опыту сказать, что заранее продумать пусть грубую, но достаточно полную схему базы данных, а уж потом нырять с головой в кодирование весьма полезно.

Также весьма желательно известить коллег о том, что вы собираетесь приступить к созданию новой миграции, чтобы потом два человека не

Говорит Себастьян...

Мы применяем подключаемый к `svn` сценарий (идею подал Конор Хант), который контролирует добавление новых миграций в репозиторий и предотвращает появление одинаковых номеров.

Еще один подход – назначить человека, отвечающего за миграции. Тогда разработчики могли бы локально создавать и тестировать миграции, а потом отправлять их по электронной почте «координатору», который проверит результат и присвоит правильные номера.

[1] *http://jayfields.blogspot.com/2006/12/rails-migrations-with-large-team-part.html.*

Говорит Кортенэ...

Выгоды от хранения схемы базы данных в системе управления версиями намного перевешивают трудности, которые возникают, когда члены команды спонтанно изменяют схему. Хранение всех версий кода снимает неприятный вопрос: «Кто добавил это поле?».

Как обычно, на эту тему написано несколько подключаемых к Rails модулей. Один из них написал я сам и назвал IndependentMigrations. Проще говоря, он позволяет иметь несколько миграций с одним и тем же номером. Другие модули допускают идентификацию миграций по временному штампу. Подробнее о моем модуле и альтернативах можно прочитать по адресу *http://blog.caboo.se/articles/2007/3/27/independent-migrations-plugin*.

попытались поставить на учет миграцию с одним и тем же номером, что приведет к печальным последствиям.

Migration API

Но вернемся к самому Migration API. Вот как будет выглядеть созданный ранее файл 002_create_clients.rb после добавления определений четырех колонок в таблицу `clients`:

```
class CreateClients < ActiveRecord::Migration
  def self.up
    create_table :clients do |t|
      t.column :name, :string
      t.column :code, :string
      t.column :created_at, :datetime
      t.column :updated_at, :datetime
    end
  end

  def self.down
    drop_table :clients
  end
end
```

Как видно из этого примера, директивы миграции находятся в определениях двух методов класса: `self.up` и `self.down`. Если перейти в каталог проекта и набрать команду `rake db:migrate`, будет создана таблица `clients`. По ходу миграции Rails печатает информативные сообщения, чтобы было видно, что происходит:

```
$ rake db:migrate
(in /Users/obie/prorails/time_and_expenses)
```

```
== 2 CreateClients: migrating
=============================================
-- create_table(:clients)
   -> 0.0448s
== 2 CreateClients: migrated (0.0450s)
================================
```

Обычно выполняется только код метода `up`, но, если вы захотите *откатиться* к предыдущей версии схемы, то метод `down` опишет, что нужно сделать, чтобы отменить действия, произведенные в методе `up`.

Для отката необходимо выполнить то же самое задание `migrate`, но передать в качестве параметра номер версии, на которую необходимо откатиться: `rake db:migrate VERSION=1`.

create_table(name, options)

Методу `create_table` необходимы по меньшей мере имя таблицы и блок, содержащий определения колонок. Почему мы задаем идентификаторы символами, а не просто в виде строк? Работать будет и то, и другое, но для ввода символа нужно на одно нажатие меньше[1].

В методе `create_table` сделано серьезное и обычно оказывающееся истинным предположение: необходим автоинкрементный целочисленный первичный ключ. Именно поэтому вы не видите его объявления в списке колонок. Если это допущение не выполняется, придется передать методу `create_table` некоторые параметры в виде хеша.

Например, как определить простую связующую таблицу, в которой есть два внешних ключа, но ни одного первичного? Просто задайте параметр `:id` равным `false` – в виде булева значения, а не символа! Тогда миграция не будет автоматически генерировать первичный ключ:

```
create_table :ingredients_recipes, :id => false do |t|
  t.column :ingredient_id, :integer
  t.column :recipe_id, :integer
end
```

Если же вы хотите, чтобы колонка, содержащая первичный ключ, называлась не id, передайте в параметре `:id` какой-нибудь символ. Пусть, например, корпоративный стандарт требует, чтобы первичные ключи именовались с учетом объемлющей таблицы: *tablename_id*. Тогда ранее приведенный пример нужно записать так:

```
create_table :clients, :id => :clients_id do |t|
  t.column :name, :string
  t.column :code, :string
  t.column :created_at, :datetime
```

[1] Если вы находите, что взаимозаменяемость символов и строк в Rails несколько раздражает, то не одиноки.

```
    t.column :updated_at, :datetime
  end
```

Параметр `:force => true` говорит миграции, что нужно *предварительно удалить определяемую таблицу, если она существует*. Но будьте осторожны, поскольку при запуске в режиме эксплуатации это может привести к потере данных (чего вы, возможно, не хотели). Насколько я знаю, параметр `:force` наиболее полезен, когда нужно привести базу данных в известное состояние, но при повседневной работе он редко бывает нужен.

Параметр `:options` позволяет включить дополнительные инструкции в SQL-предложение CREATE и полезен для учета специфики конкретной СУБД. В зависимости от используемой СУБД можно задать, например, кодировку, схему сортировки, комментарии, минимальный и максимальный размер и многие другие свойства.

Параметр `:temporary => true` сообщает, что нужно создать таблицу, существующую только во время выполнения миграции. В сложных случаях это может оказаться полезным для переноса больших наборов данных из одной таблицы в другую, но вообще-то применяется нечасто.

> **Говорит Себастьян...**
>
> Малоизвестно, что можно удалять файлы из каталогов миграции (сохраняя самые свежие), чтобы размер каталога `db/migrate` оставался на приемлемом уровне. Можно, скажем, переместить старые миграции в каталог `db/archived_migrations` или сделать еще что-то в этом роде.
>
> Если вы хотите быть абсолютно уверены, что ваш код допускает развертывание с нуля, можете заменить миграцию с наименьшим номером миграцией «создать заново все», основанной на текущем содержимом файла `schema.rb`.

Определение колонок

Добавить в таблицу колонки можно либо с помощью метода `column` внутри блока, ассоциированного с предложением `create_table`, либо методом `add_column`. Второй метод отличается от первого только тем, что принимает в качестве первого аргумента имя таблицы, куда добавляется колонка.

```
create_table :clients do |t|
  t.column :name, :string
end

add_column :clients, :code, :string
add_column :clients, :created_at, :datetime
```

Первый (или второй) параметр – имя колонки, а второй (или третий) – ее тип. В стандарте SQL92 определены фундаментальные типы данных, но в каждой конкретной СУБД имеются свойственные только ей расширения стандарта.

Если вы знакомы с типами данных в СУБД, то предыдущий пример мог вызвать недоумение: почему колонка имеет тип `string`, хотя в базах данных такого типа нет, а есть типы `char` и `varchar`?

Отображение типов колонок

Причина, по которой для колонки базы данных объявлен тип `string`, заключается в том, что миграции в Rails по идее должны быть независимыми от СУБД. Вот почему можно (и я это делал) вести разработку на СУБД Postgres, а развертывать систему на Oracle.

Полное обсуждение вопроса о том, как выбирать правильные типы данных, выходит за рамки этой книги. Но полезно иметь под рукой справку от отображении обобщенных типов в миграциях на конкретные типы для различных СУБД. В табл. 6.1 такое отображение приведено для СУБД, которые наиболее часто встречаются в приложениях Rails.

Таблица 6.1. Отображение типов данных для СУБД, которые наиболее часто встречаются в приложениях Rails

Тип миграции Класс Ruby	MySQL	Postgres	SQLite	Oracle
:binary String	blob	bytea	blob	blob
:boolean Boolean	tinyint(1)	boolean	Boolean	number(1)
:date Date	date	date	date	date
:datetime Time	datetime	timestamp	datetime	date
:decimal BigDecimal	decimal	decimal	decimal	decimal
:float Float	float	float	float	number
:integer Fixnum	int(11)	integer	integer	number(38)
:string String	varchar(255)	character varying(255)	varchar(255)	varchar(255)
:text String	text	clob(32768)	text	clob
:time Time	time	time	time	date
:timestamp Time	datetime	timestamp	datetime	date

Для каждого класса-адаптера соединения существует хеш native_database_types, устанавливающий описанное в табл. 6.1 отображение. Если вас заинтересуют отображения для других СУБД, можете открыть код соответствующего адаптера и найти в нем вышеупомянутый хеш. Так, в классе SQLServerAdapter из файла sqlserver_adapter.rb хеш native_database_types выглядит следующим образом:

```
def native_database_types
  {
    :primary_key => "int NOT NULL IDENTITY(1, 1) PRIMARY KEY",
    :string      => { :name => "varchar", :limit => 255 },
    :text        => { :name => "text" },
    :integer     => { :name => "int" },
    :float       => { :name => "float", :limit => 8 },
    :decimal     => { :name => "decimal" },
    :datetime    => { :name => "datetime" },
    :timestamp   => { :name => "datetime" },
    :time        => { :name => "datetime" },
    :date        => { :name => "datetime" },
    :binary      => { :name => "image"},
    :boolean     => { :name => "bit"}
  }
end
```

Дополнительные характеристики колонок

Во многих случаях одного лишь указания типа данных недостаточно. Все объявления колонок принимают еще и следующие параметры:

```
:default => value
```

Задает значение по умолчанию, которое записывается в данную колонку вновь созданной строки. Явно указывать null необязательно, достаточно просто опустить этот параметр.

```
:limit => size
```

Задает размер для колонок типа string, text, binary и integer. Семантика зависит от конкретного типа данных. В общем случае ограничение на строковые типы относится к числу символов, а для других типов речь идет о количестве байтов, выделяемых в базе для хранения значения.

```
:null => true
```

Делает колонку обязательной, добавляя ограничение not null, которое проверяется на уровне СУБД.

Количество знаков в десятичной записи

Для колонок, объявленных как :decimal, можно задавать следующие характеристики:

```
:precision => number
```

Здесь `precision` (точность) – общее число цифр в десятичной записи числа.

```
:scale => number
```

Здесь `scale` (масштаб) – число цифр *справа* от десятичного знака. Например, для числа 123,45 точность равна 5, а масштаб – 2. Очевидно, масштаб не может быть больше точности.

Примечание

Для десятичных типов велика опасность потери данных при переносе с одной СУБД на другую Например, поскольку в Oracle и SQL Server принимаемая по умолчанию точность различается, в процессе переноса может происходить отбрасывание знаков и, как следствие, изменение числового значения. Поэтому разумно всегда задавать точность и масштаб явно.

Подводные камни при выборе типов колонок

Выбор типа колонки не всегда очевиден и зависит как от используемой СУБД, так и от требований, предъявляемых приложением:

- **:binary.** В зависимости от способа использования хранение в базе двоичных данных может сильно снизить производительность. Rails загружает объекты из базы данных целиком, поэтому присутствие больших двоичных атрибутов в часто употребляемых моделях заметно увеличивает нагрузку на сервер базы данных;
- **:boolean.** Булевы значения в разных СУБД хранятся по-разному. Иногда для представления true и false используются целые значения 1 и 0, а иногда – символы T и F. Rails прекрасно справляется с задачей отображения таких значений на «родные» для Ruby объекты `true` и `false`, поэтому задумываться о реальной схеме хранения не нужно. Прямое присваивание атрибутам значений 1 или F, в конкретном случае, может быть, и сработает, но такая практика считается антипаттерном;
- **:date**, **:datetime** и **:time**. Сохранение дат в СУБД, где нет встроенного типа даты, например в Microsoft SQL Server, может стать проблемой. Rails отображает тип `datetime` на класс Ruby `Time`, который не позволяет представить даты ранее 1 января 1970 года. Но ведь имеющийся в Ruby класс `DateTime` умеет работать с более ранними датами, так почему же он не используется в Rails? Дело в том, что класс `Time` реализован на C и потому работает очень быстро, тогда как `DateTime` написан на чистом Ruby и, следовательно, медленнее.

Чтобы заставить ActiveRecord отображать тип даты на `DateTime` вместо `Time`, поместите код из листинга 6.1 в какой-нибудь файл, находящийся в каталоге **lib/** и затребуйте его из сценария **config/environment.rb** с помощью `require`.

Листинг 6.1. Отображение даты на тип DateTime вместо Time

```
require 'date'
# Это необходимо сделать, потому что класс Time не поддерживает
# даты ранее 1970 года…

class ActiveRecord::ConnectionAdapters::Column
  def self.string_to_time(string)
    return string unless string.is_a?(String)
    time_array = ParseDate.parsedate(string)[0..5]
    begin
      Time.send(Base.default_timezone, *time_array)
    rescue
      DateTime.new(*time_array) rescue nil
    end
  end
end
```

- **:decimal.** В старых версиях Rails (до 1.2) тип `:decimal` с фиксированной точкой не поддерживался, поэтому во многих ранних приложениях Rails некорректно использовался тип `:float`. Числа с плавающей точкой по природе своей неточны, поэтому для большинства бизнес-приложений следует выбирать тип **:decimal**, а не **:float**;
- **:float.** Не пользуйтесь типом `:float` для хранения денежных сумм[1] и вообще любых данных, для которых необходима фиксированная точность. Поскольку числа с плавающей точкой дают хорошую аппроксимацию, простое хранение данных в таком формате, наверное, приемлемо. Проблемы начинаются, когда вы пытаетесь выполнять над числами математические действия или операции сравнения, поскольку внести таким образом ошибку в приложение до смешного просто, а найти ее ох как тяжело;
- **:integer** и **:string**. Есть не так уж много неприятностей, с которыми можно столкнуться при использовании целых и строковых типов. Это основные кирпичики любого приложения, и многие разработчики опускают задание размера, получая по умолчанию 11 цифр и 255 знаков соответственно.

Следует помнить, что при попытке сохранить значение, которое не помещается в отведенную для него колонку (для строк – по умолчанию 255 знаков), вы не получите никакого уведомления об ошибке. Строка будет просто молча обрезана. Убеждайтесь, что длина введенных пользователем данных не превосходит максимально допустимой.

- **:text.** Есть сообщения о том, что текстовые поля снижают производительность запросов настолько, что в сильно нагруженных приложениях это может превратиться в проблему. Если вам абсолютно необходимы текстовые данные в приложениях, для которых быстродействие критично, помещайте их в отдельную таблицу;

[1] По адресу *http://dist.leetsoft.com/api/money/* размещен рекомендуемый класс Money с открытым исходным текстом.

- **:timestamp.** В версии Rails 1.2 при создании новых записей ActiveRecord может не очень хорошо работать, когда значение колонки по умолчанию генерируется функцией, как для данных типа timestamp в Postgres. Проблема в том, что Rails не исключает такие колонки из предложения `insert`, как следовало бы, а задает для них «значение» `null`, что может приводить к игнорированию значения по умолчанию.

Нестандартные типы данных

Если в вашем приложении необходимы типы данных, специфичные для конкретной СУБД (например, тип `:double`, обеспечивающий более высокую точность, чем `:float`), включите в файл **config/environment.rb** директиву `config.active_record.schema_format = :sql`, чтобы заставить Rails сохранять информацию о схеме в «родном» для данной СУБД формате DDL, а не в виде кросс-платформенного кода на Ruby, записываемого в файл **schema.rb**.

«Магические» колонки с временными штампами

К колонкам типа timestamp Rails применяет *магию*, если они названы определенным образом. Active Record автоматически снабжает операции *создания* временным штампом, если в таблице есть колонка с именем `created_at` или `created_on`. То же самое относится к операциям *обновления*, если в таблице есть колонка с именем `updated_at` или `updated_on`.

Отметим, что в файле миграции тип колонок `created_at` и `updated_at` должен быть задан как `datetime`, а не `timestamp`.

Автоматическую проштамповку можно глобально отключить, задав в файле **config/environment.rb** следующую переменную:

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

За счет наследования данный код отключает временные штампы для всех моделей, но можно сделать это и избирательно в конкретной модели, если установить переменную `record_timestamps` в `false` только для нее. По умолчанию временные штампы выражены в местном поясном времени, но, если задать переменную `ActiveRecord::Base.default_timezone = :utc`, будет использовано время UTC.

Методы в стиле макросов

Большинство существенных классов, которые вы пишете, программируя приложение для Rails, сконфигурированы для вызова *в стиле макросов* (в определенных кругах это также называется *предметно-ориентированным языком*, или *DSL* – domain-specific language). Основная идея заключается в том, что в начале класса размещается максимально понятный блок кода, конфигурация которого сразу видна.

Для размещения вызовов в стиле макросов именно в начале файла есть веская причина. Эти методы *декларативно* сообщают Rails, как управлять экземплярами, выполнять контроль данных, делать обратные вызовы и взаимодействовать с другими моделями. Часто в таких методах используется *метапрограммирование*, то есть на этапе выполнения они добавляют в класс то или иное поведение в форме дополнительных переменных экземпляра или методов.

Объявление отношений

Рассмотрим, например, класс `Client`, в котором объявлены некоторые отношения. Не пугайтесь, если смысл этих объявлений вам не ясен; мы подробно поговорим об этом в главе 7 «Ассоциации в ActiveRecord». Сейчас я хочу лишь показать, что имею в виду, говоря о *стиле макросов*:

```
class Client < ActiveRecord::Base
  has_many :billing_codes
  has_many :billable_weeks
  has_many :timesheets, :through => :billable_weeks

end
```

Благодаря трем объявлениям `has_many` класс `Client` получает по меньшей мере три новых атрибута – прокси-объекты, позволяющие интерактивно манипулировать ассоциированными наборами.

Я припоминаю, как когда-то в первый раз обучал своего друга – опытного программиста на Java – основам Ruby и Rails. Несколько минут он пребывал в сильном замешательстве, а потом я буквально увидел, как в его голове зажглась лампочка, и он провозгласил: «О! Так это же методы!».

Ну конечно, это самые обычные вызовы методов в контексте объекта класса. Мы опустили скобки, чтобы подчеркнуть декларативность. Это не более чем вопрос стиля, но лично мне скобки в таком фрагменте кажутся неуместными:

```
class Client < ActiveRecord::Base
  has_many(:billing_codes)
  has_many(:billable_weeks)
  has_many(:timesheets, :through => :billable_weeks)
end
```

Когда интерпретатор Ruby загружает файл **client.rb**, он выполняет методы `has_many`, которые, еще раз подчеркну, определены как *методы класса* `ActiveRecord::Base`. Они выполняются в контексте *класса* `Client` и добавляют в него атрибуты, которые в дальнейшем становятся доступны *экземплярам* класса `Client`. Новичку такая модель программирования может показаться странной, но очень скоро она становится «вторым я» любого программиста Rails.

Примат соглашения над конфигурацией

Примат соглашения над конфигурацией – один из руководящих принципов Rails. Если вы следуете принятым в Rails соглашениям, то почти ничего не придется явно конфигурировать. И здесь мы наблюдаем разительный контраст с тем, сколько приходится конфигурировать для запуска даже простейшего приложения в других технологиях.

Дело не в том, что всякое новое приложение Rails уже создается с готовой *умалчиваемой* конфигурацией, отражающей применяемые соглашения. Нет – соглашения уже *впечатаны* в среду, жестко зашиты в ее поведение, и это-то подразумеваемое по умолчанию поведение вы и переопределяете с помощью явного конфигурирования, когда возникает необходимость.

Стоит также отметить, что, как правило, конфигурирование производится очень близко к тому, что конфигурируется. Объявления *ассоциаций*, *проверок* и *обратных вызовов* располагаются в начале большинства моделей ActiveRecord.

Подозреваю, что многие из нас впервые занялись конфигурированием (в противовес соглашению), чтобы изменить соответствие между именем класса и таблицы базы данных, поскольку по умолчанию Rails предполагает, что имя таблицы образуется как множественное число от имени класса. И, поскольку такое соглашение застает врасплох многих начинающих разработчиков Rails, рассмотрим эту тему, перед тем как двигаться дальше.

Приведение к множественному числу

В Rails есть класс `Inflector`, в обязанность которого входит преобразование строк (слов) из единственного числа в множественное, имен классов – в имена таблиц, имен классов с указанием модуля – в имена без модуля, имен классов – во внешние ключи и т. д. (некоторые операции имеют довольно смешные имена, например `dasherize`).

Принимаемые по умолчанию окончания для формирования единственного и множественного числа неисчисляемых имен существительных хранятся в файле **inflections.rb** в каталоге установки Rails. Как правило, класс `Inflector` успешно находит имя таблицы, образуемое приведением имени класса к множественному числу, но иногда случаются оплошности. Для многих новых пользователей Rails это становится первым камнем преткновения, но причин для паники нет. Можно заранее проверить, как `Inflector` будет реагировать на те или иные слова. Для этого понадобится лишь консоль Rails, которая, кстати, является одним из лучших инструментов при работе с Rails.

Чтобы запустить консоль, выполните из командной строки сценарий **script/console**, который находится в каталоге вашего проекта.

```
$ script/console
>> Inflector.pluralize "project"
=> "projects"
>> Inflector.pluralize "virus"
=> "viri"
>> Inflector.pluralize "pensum"
=> "pensums"
```

Как видите, `Inflector` достаточно умен – в качестве множественного числа от virus он правильно выбрал viri. Но, если вы знаете латынь, то, наверное, обратили внимание, что pensum во множественном числе на самом деле пишется как pensa. Понятно, что инфлектор латыни не обучен.

Однако вы *можете* расширить познания инфлектора одним из трех способов:

- добавить новое правило-образец
- описать исключение
- объявить, что некое слово не имеет множественного числа

Лучше всего делать это в файле **config/environment.rb**, где уже имеется прокомментированный пример:

```
Inflector.inflections do |inflect|
  inflect.plural /^(.*)um$/i, '\1a'
  inflect.singular /^(.*)a/i, '\1um'
  inflect.irregular 'album', 'albums'
  inflect.uncountable %w( valium )
end
```

Кстати, в версии Rails 1.2 инфлектор принимает слова, уже записанные *во множественном числе*, и... ничего с ними не делает, что, наверное, самое правильное. Прежние версии Rails вели себя в этом отношении не так разумно.

```
>> "territories".pluralize
=> "territories"
>> "queries".pluralize
=> "queries"
```

Если хотите посмотреть на длинный список существительных, которые `Inflector` правильно приводит к множественному числу, загляните в файл **activesupport/test/inflector_test.rb**. Я нашел в нем немало интересного, например:

```
"datum" => "data",
"medium" => "media",
"analysis" => "analyses"
```

Надо ли сообщать разработчикам ядра об ошибках в работе Inflector

Майкл Козярский (Michael Koziarski), один из разработчиков ядра Rails, пишет, что сообщать о проблемах с классом `Inflector` не следует: «Инфлектор практически заморожен; до выхода версии 1.0 мы добавляли в него много правил для исправления ошибок, чем привели в ярость тех, кто называл свои таблицы согласно старым вариантам. Если необходимо, добавляйте исключения в файл environment.rb самостоятельно».

Задание имен вручную

Разобравшись с инфлектором, вернемся к конфигурированию классов моделей ActiveRecord. Методы `set_table_name` и `set_primary_key` позволяют обойти соглашения Rails и явно задать имя таблицы и имя колонки, содержащей первичный ключ.

Предположим, к примеру (и только к примеру!), что я вынужден использовать для именования таблиц какое-то мерзкое соглашение, отличающееся от принятого в ActiveRecord. Тогда я мог бы поступить следующим образом:

```
class Client < ActiveRecord::Base
  set_table_name "CLIENT"
  set_primary_key "CLIENT_ID"
end
```

Методы `set_table_name` и `set_primary_key` позволяют выбрать для таблиц и первичных ключей произвольные имена, но тогда вы должны явно указать их в классе модели. Для одной модели это всего лишь пара лишних строчек, но в большом приложении оказывается ненужным усложнением, поэтому не делайте этого без острой необходимости.

Если вы не можете сами диктовать соглашения о выборе имен, например, когда схемами управляет отдельная группа администрирования базы данных, то выбора, наверное, нет. Но при наличии свободы действий лучше придерживаться соглашений Rails. Возможно, это покажется непривычным, зато позволит сэкономить время и избежать ненужных сложностей.

Унаследованные схемы именования

Если вы работаете с унаследованными схемами, может возникнуть искушение вставлять метод `set_table_name` повсюду, нужен он или не нужен. Но прежде чем у вас появится такая привычка, познакомьтесь