ACTIVERECORD

IT'S ALL ABOUT DATA 

開始之前...

- 回去繼續要多練習
- 補充資料
  - https://railsbook.tw/
  - 第 16 ~ 19 章

內容

- 資料表關連 (Relationshop)
- ORM 基本操作
- 資料驗證 (Validation)
- 回呼 (Callback)

ACTIVERECORD
慣例

» Model 名稱：大寫、單數

» Table 名稱：小寫、複數 (因為可以放很多資料?)

MODEL	TABLES
User	users
Post	posts
Category	categories

ACTIVERECORD
慣例

- 預設會有一個叫做 "id" 的流水編號欄位。
- timestamps
  - 自動建立 created_at 以及 updated_at 欄位。
  - 在資料新增或更新的時候自動寫入當下時間。

ACTIVERECORD
慣例

» 這個 Migration 檔會做什麼事? 會建立幾個欄位?

```
class CreateUsers < ActiveRecord::Migration[5.2]
  def change
    create_table :users do |t|
      t.string :name
      t.string :email

      t.timestamps
    end
  end
end
```

ACTIVERECORD
冷知識

- 怎麼知道英文的單、複數? 不規則變化?
  - `pluralize, singularize`
- 怎麼知道檔名跟類別之間的對照?
  - `underscore, camelize`

關連
SCENARIO (線上商店)

» Owner (資料表名稱：owners)

欄位名稱	資料型態	說明
name	string	姓名
email	string	Email
tel	string	電話

關連
SCENARIO (線上商店)

» Store (資料表名稱：stores)

欄位名稱	資料型態	說明
title	string	店名
address	string	地址
tel	string	電話
owner_id	integer	存放該商店的 Owner 的流水編號(id)

關連
SCENARIO (線上商店)

» Product (資料表名稱：products)

欄位名稱	資料型態	說明
name	string	商品名稱
description	text	描述
price	decimal	售價(為什麼不用 integer?)
is_available	boolean	是否銷售中?
store_id	integer	存放這筆商品所屬商店的流水編號(id)

關連
SCENARIO (線上商店)

» 一對一(1 : 1)

  » 一個店長(owner)只能開一間店(store)

» 一對多(1 : N)

  » 一間店(store)可以賣很多種的商品(products)

» 多對多(N : N)

  » 一間店(store)可以賣很多種的商品(products)，一種商品也可

關連
種類

- Rails 支援以下這六種關連種類：
  - has_one
  - has_many
  - belongs_to
  - has_one :through
  - has_many :through
  - has_and_belongs_to_many

HAS_ONE

關連
HAS_ONE

```ruby
class Owner < ApplicationRecord
  has_one :store
end
```

關連
HAS_ONE

» has_one 不是設定，它是一個類別方法。

» 設定 has_one :store 後，會動態產生幾個好用的方法：

  » store

  » store=

  » build_store

  » create_store

關連
HAS_ONE

```ruby
# 各別建立 Owner 跟 Store 實體
o1 = Owner.create(name: 'Sherly')
s1 = Store.new(name: 'Ruby Shop')

# 然後把 Store 指定給 Owner
o1.store = s1    # 這個 store= 方法就是 has_one 產生的
puts o1.store    # store 方法也是 has_one 產生的
```

關連
HAS_ONE

```
# 或是，直接從 Owner 的角度直接 create 或 build 商店

o1.create_store(name: 'Ruby Shop')
o1.build_store(name: 'Ruby Shop')
```

關連
HAS_ONE

» 問題

1. build 跟 create 的差別?

關連
HAS_ONE

- 練習
  - 建立 2 個 model (Owner 以及 Store)。
  - 設定 1 對 1 關係 (has_one、belongs_to)。
  - 各別建立 Owner 跟 Store 實體，然後把 Store 指定給 Owner。
  - 或是，直接從 Owner 的角度直接 build 或 create 商店。

BELONGS_TO

關連
BELONGS_TO

```ruby
class Store < ApplicationRecord
  belongs_to :owner
end
```

關連
BELONGS_TO

» 跟 has_one 一樣，belongs_to 也是類別方法。

» 設定 belongs_to :owner 後，會動態得到幾個好用的方法：

  » owner

  » owner=

關連
BELONGS_TO

» 問題：

1. has_one 跟 belongs_to 需要同時設定嗎?
2. 如果只設定 has_one 但沒有設定 belongs_to 的話會怎樣?
3. 到底誰該擁有誰、誰該屬於誰?
4. owner_id 一定要叫這個名字嗎?

HAS_MANY

關連
HAS_MANY

```ruby
class Store < ApplicationRecord
  has_many :products
end
```

關連
HAS_MANY

- has_many 不是設定，是類別方法。
- 設定 has_many :products 後，會動態產生幾個方法：
  - products
  - products=
  - build
  - create

關連
HAS_MANY

```ruby
# 各別建立物件
s1 = Store.first
p1 = Product.new(name: 'Ruby', price: 200)
p2 = Product.new(name: 'Rails', price: 150)

# 把 2 個 Product 物件丟給 Store 物件
s1.products = [p1, p2]  # 這個 products= 就是動態產生的

# 或是一次丟一個也可以
s1.products << p1
s1.products << p2
```

關連
HAS_MANY

```ruby
s1 = Store.first

# 或是，從 Store 的角度來建立 Product
s1.products.build(name: 'Ruby', price: 200)
s1.products.create(name: 'Ruby', price: 200)
```

關連
HAS_MANY

» 問題：

1. has_many 後面不用符號而改用字串會怎樣?

2. has_many 設定單數會怎樣?

關連
HAS_MANY

- 練習
  - 建立 2 個 model (Store 以及 Product)。
  - 設定 1 對多關係 (has_many、belongs_to)。
  - 各別建立 Product 物件，然後把 Product 指定給 Store。
  - 或是，從 Store 的角度來建立 Product。

HAS_MANY :THROUGH

關連
HAS_MANY :THROUGH

» 用起來的手感跟 has_many 差不多...

» 需要第三個資料表來儲存兩邊的資訊

» 第三個資料表通常僅存放兩邊的 id，並且 belongs_to 兩邊的 Model

關連
HAS_MANY :THROUGH

```ruby
# Store
class Store < ApplicationRecord
  has_many :ware_houses
  has_many :products, through: :ware_houses
end

# Product
class Product < ApplicationRecord
  has_many :ware_houses
  has_many :stores, through: :ware_houses
end
```

關連
HAS_MANY :THROUGH

» 用來存放關連的 WareHouse

```
# WareHouse
class WareHouse < ApplicationRecord
  belongs_to :store
  belongs_to :product
end
```

關連
HAS_MANY :THROUGH

» 練習

HABTM

關連
HABTM

```ruby
# Store
class Store < ApplicationRecord
  has_and_belongs_to_many :products
end

# Product
class Product < ApplicationRecord
  has_and_belongs_to_many :stores
end
```

ORM 基本操作

ACTIVERECORD
ORM 基本操作

» "new" 會建立一筆資料，但還不會存到資料庫裡。

» "create" 或 "create!" 會建立一筆資料，並直接寫入資料庫裡。

ACTIVERECORD
ORM 基本操作

» 問題：

1. create 跟 create! 方法有什麼不同?

ACTIVERECORD
CRUD 基本操作 (CREATE)

```ruby
b = Product.new(name: 'Ruby book', price: 350)
b.save    # 到這一步才存到資料庫裡

# 建立並同時存到 products 表格裡
Product.create(name: 'Ruby book', price:350)
```

ACTIVERECORD
CRUD 基本操作 (READ)

» first & last

» find_by(id: 1) v.s find(1)

» find_by_sql

» find_each

ACTIVERECORD
CRUD 基本操作 (READ)

```ruby
Product.first                # 找到第一筆資料
Product.last                 # 找到最後一筆資料
Product.find(1)              # 找到 id = 1 的資料
Product.find_by(id: 1)       # 找到 id = 1 的資料

# 直接使用 SQL 查詢
Product.find_by_sql("select * from products where id = 1")

# batch find
Product.find_each do | product |
  # ...
end
```

ACTIVERECORD
CRUD 基本操作 (READ)

» 問題

1. `find(1)` 跟 `find_by(id: 1)` 有什麼不一樣?

2. `Owner.find_by_name_and_email` 有這種神奇的寫法?

ACTIVERECORD
CRUD 基本操作 (READ)

» all

» select

» where

» order

» limit

ACTIVERECORD
CRUD 基本操作 (READ)

```ruby
Product.all                          # 找出所有產品
Product.select('name')               # 同上，但只選取 name 欄位
Product.where(name: 'Ruby')          # 找出所有 name 欄位的內容是 "Ruby" 的資料
Product.order('id DESC')             # 依照 id 大小反向排序
Product.order(id: :desc)             # 同上
Product.limit(5)                     # 只取出 5 筆資料
```

ACTIVERECORD
CRUD 基本操作 (READ)

» count

» average

» sum

» maximum & minimum

» 不要傻傻的用 ORM 抓出來轉迴圈再來計算！

ACTIVERECORD
CRUD 基本操作 (UPDATE)

» save

» update

» update_attributes

» update_all

ACTIVERECORD
CRUD 基本操作 (UPDATE)

```ruby
p1 = Product.find_by(id: 1)

# 儲存
p1.name = 'Ruby book'
p1.save

# 更新
p1.update(name: 'Ruby book')
p1.update_attributes(name: 'Ruby book')

# 全部更新(請小心使用!)
Product.update_all(name: 'Ruby book', price: 250)
```

ACTIVERECORD
CRUD 基本操作 (UPDATE)

» 問題

1. update 跟 update_attributes 有什麼不一樣?

ACTIVERECORD
CRUD 基本操作 (DELETE)

» delete

» destroy

» destroy_all(condition = nil)

ACTIVERECORD
CRUD 基本操作 (DELETE)

```ruby
p1 = Product.find_by(id: 1)

# 刪除
p1.destroy
p1.delete

# 刪除編號 1 號的商品
Product.delete(1)

# 刪除所有低於 10 元的商品
Product.destroy_all("price < 10")
```

ACTIVERECORD
CRUD 基本操作 (DELETE)

» 問題

1. delete 跟 destroy 有什麼不一樣?

2. destroy 是真的把資料刪除嗎?

ACTIVERECORD
ORM 基本操作

» increment & decrement

» toggle

```
my_order = Order.first
my_order.increment(:quantity)  # quantity 欄位的值 + 1
my_order.toggle(:is_online)    # 把原本的 true 變 false，false 變 true
```

ACTIVERECORD
SCOPE

» 把一群條件整理成一個 Scope

» 簡化使用時的邏輯

» 減少在 Controller 裡寫一堆的 Where 組合

```
class Product < ApplicationRecord
  scope :available, -> { where(is_available: true)}
  scope :price_over, ->(p) { where(["price > ?", p])}
end
```

ACTIVERECORD
SCOPE

» 用起來跟類別方法一樣

```
class ProductsController < ApplicationController
  def index
    @products = Product.available
  end
end
```

ACTIVERECORD
SCOPE

» default scope 及 unscope

» default scope 的副作用?

```ruby
class Product < ApplicationRecord
  default_scope { order('id DESC') }
  scope :available, -> { unscope(:order).where(is_available: true)}
end
```

ACTIVERECORD
類別方法 CLASS METHODS

» 除使用 scope 外，也可使用類別方法

```ruby
class Product < ApplicationRecord
  def self.available
    where(is_available: true)
  end
end
```

ACTIVERECORD
SCOPE

» 問題

1. scope 跟類別方法有什麼不一樣?

2. 什麼時候該用什麼寫法?

ACTIVERECORD
資料驗證 VALIDATION

» presence

» format

» uniqueness

» numericality

» length

» condition

ACTIVERECORD
資料驗證 VALIDATION

```ruby
class Product < ApplicationRecord
  validates :name, presence: true
end
```

ACTIVERECORD
資料驗證 VALIDATION

舊式寫法，可混搭使用：

```
class Product < ApplicationRecord
  validates_presence_of :name, :title, :price
end
```

ACTIVERECORD
自訂驗證器 VALIDATOR

```ruby
class Product < ApplicationRecord
  validate :name_validator

  private
  def name_validator
    unless name.starts_with? 'Ruby'
      errors[:name] << 'must begin with Ruby'
    end
  end
end
```

ACTIVERECORD
自訂驗證器 VALIDATOR

» 想寫出這樣的語法嗎?

```
class Product < ApplicationRecord
  validates :name, presence: true, begin_with_ruby: true
end
```

ACTIVERECORD
自訂驗證器 VALIDATOR

```ruby
class BeginWithRubyValidator < ActiveModel::EachValidator
  def validate_each(record, attribute, value)
    unless value.starts_with? 'Ruby'
      record.errors[attribute] << 'must begin with Ruby'
    end
  end
end

class Product < ApplicationRecord
  validates :name, presence: true, begin_with_ruby: true
end
```

ACTIVERECORD
資料驗證 VALIDATION

若資料未通過驗證...

```
>> p1 = Product.new   # 忘了寫產品名稱，但還沒寫入資料庫
>> p1.errors.any?     # 看看有沒錯誤...
=> false              # 這時候還沒發生錯誤
```

ACTIVERECORD
資料驗證 VALIDATION

若資料未通過驗證...

```
>> p1.validate                          # 開始進行驗證(還不需要存檔喔)
=> false                                # 得到 false 表示沒有驗證成功
>> p1.errors.any?                       # 看看有沒有錯誤...
=> true                                 # 這時候就有錯誤內容了
>> p1.errors.full_messages  # 使用 full_messages 可把資料印出來
=> ["Title can't be blank"]
```

ACTIVERECORD
資料驗證 VALIDATION

» 問題

1. 硬是要繞過驗證可以嗎?

2. 只要有驗證就可以保證資料正確嗎?

ACTIVERECORD
回呼 (CALLBACK)

- 資料存檔的流程會經過以下流程：
  - `save > valid > before_validation > validate > after_validate > before_save > before_create > create > after_create > after_save > after_commit`
- 可在這些流程執行的時候對資料做些事情...

ACTIVERECORD
回呼 (CALLBACK)

» 在資料存檔前對密碼進行加密

```
class User < ApplicationRecord
  before_save :encrypt_email

  private
  def encrypt_email
    require 'digest'
    self.email = Digest::MD5.hexdigest(email)
  end
end
```

ACTIVERECORD
回呼 (CALLBACK)

» 問題

1. 在上一頁的範例中，使用 `before_save` 會造成什麼問題?

2. `before_save` 跟 `before_create` 有什麼差別?

今天學到了...

- 資料表關連 (Relationshop)
- ORM 基本操作
- 資料驗證 (Validation)
- 回呼 (Callback)

聯絡
高見龍 (EDDIE)

» Blog: https://kaochenlong.com

» Facebook: http://www.facebook.com/eddiekao

» Twitter: https://twitter.com/eddiekao

» Email: eddie@5xruby.tw

» Mobile: +886-928-617-687