5 天搞定 Ruby on Rails

企业内训纲要

第 4 天：Model 培训

版本： V1

口号：快速迭代，不断完善

ruby 技术交流--南方群 95824005

ruby 技术交流--北方群 101388340

ruby 技术交流--东部群 236263776

ruby 技术交流--西部群 230015785

ruby 技术交流--中部群 104131248

文档，实例地址：

https://github.com/nienwoo/ruby-learn

1. 第 4 天目标

- 掌握 modelActive Record

2. 迁移

2.1. 基础

2.1.1. 特点

- 迁移使用一种统一、简单的方式，按照时间顺序修改数据库的模式
- 迁移使用 Ruby DSL 编写，因此不用手动编写 SQL 语句
- 迁移看做数据库的一个修订版本

你可以把每个迁移看做数据库的一个修订版本。

数据库中一开始什么也没有，各个迁移会添加或删除数据表、字段或记录。

Active Record 知道如何按照时间线更新数据库，不管数据库现在的模式如何，都能更新到最新结构。同时，Active Record 还会更新 db/schema.rb 文件，匹配最新的数据库结构。

2.1.2. 简单示例

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
```

```
    end
  end
end
```

这个迁移创建了一个名为 products 的表，

然后在表中创建字符串字段 name 和文本字段 description。

名为 id 的主键字段会被自动创建。

id 字段是所有 Active Record 模型的默认主键。

timestamps 方法创建两个字段：created_at 和 updated_at。

如果数据表中有这两个字段，Active Record 会负责操作。

2.2. rails 迁移指令

2.2.1. 创建表

如果迁移名是“CreateXXX”形式，后面跟着一串字段名和类型声明，迁移就会创建名为“XXX”的表，以及相应的字段。例如：

```
$ rails generate migration CreateProducts name:string part_number:string
```

生成的迁移如下：

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :name
      t.string :part_number
    end
  end
end
```

2.2.2. 增删字段

如果迁移的名字是“AddXXXToYYY”或者“RemoveXXXFromYYY”这种格式，而且后面跟着一个字段名和类型列表，那么迁移中会生成合适的 add_column 或 remove_column 语句。

```
$ rails generate migration AddPartNumberToProducts part_number:string
```

这个命令生成的迁移如下：

```
class AddPartNumberToProducts < ActiveRecord::Migration
  def change
    add_column :products, :part_number, :string
  end
end
```

迁移生成器不单只能创建一个字段，例如：

$ rails generate migration AddDetailsToProducts part_number:string price:decimal

生成的迁移如下：

```
class AddDetailsToProducts < ActiveRecord::Migration
  def change
    add_column :products, :part_number, :string
    add_column :products, :price, :decimal
  end
end
```

类似地，还可以生成删除字段的迁移：

$ rails generate migration RemovePartNumberFromProducts part_number:string

这个命令生成的迁移如下：

```
class RemovePartNumberFromProducts < ActiveRecord::Migration
  def change
    remove_column :products, :part_number, :string
  end
end
```

2.2.3. 创建索引

如果想为新建的字段创建添加索引，可以这么做：

```
$ rails generate migration AddPartNumberToProducts part_number:string:index
```

这个命令生成的迁移如下：

```
class AddPartNumberToProducts < ActiveRecord::Migration
  def change
    add_column :products, :part_number, :string
    add_index :products, :part_number
  end
end
```

2.2.4. 创建关联

在生成器中还可把字段类型设为 references（还可使用 belongs_to）。例如：

```
$ rails generate migration AddUserRefToProducts user:references
```

生成的迁移如下：

```
class AddUserRefToProducts < ActiveRecord::Migration
```

```
  def change
    add_reference :products, :user, index: true
  end
end
```

这个迁移会创建 user_id 字段，并建立索引。

如果迁移名中包含 JoinTable，生成器还会创建联合数据表：

```
rails g migration CreateJoinTableCustomerProduct customer product
```

生成的迁移如下：

```
class CreateJoinTableCustomerProduct < ActiveRecord::Migration
  def change
    create_join_table :customers, :products do |t|
      # t.index [:customer_id, :product_id]
      # t.index [:product_id, :customer_id]
    end
  end
end
```

2.2.5. 模型生成器

模型生成器和脚手架生成器会生成合适的迁移，创建模型。迁移中会包含创建所需数据表的

代码。如果在生成器中指定了字段，还会生成创建字段的代码。例如，运行下面的命令：

```
$ rails generate model Product name:string description:text
```

会生成如下的迁移：

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end
end
```

字段的名字和类型数量不限。

2.2.6. 类型修饰符

在字段类型后面，可以在花括号中添加选项。可用的修饰符如下：

limit：设置 string/text/binary/integer 类型字段的最大值；

precision：设置 decimal 类型字段的精度，即数字的位数；

scale：设置 decimal 类型字段小数点后的数字位数；

polymorphic：为 belongs_to 关联添加 type 字段；

null：是否允许该字段的值为 NULL；

例如，执行下面的命令：

$ rails generate migration AddDetailsToProducts 'price:decimal{5,2}' supplier:references{polymorphic}

生成的迁移如下：

```
class AddDetailsToProducts < ActiveRecord::Migration
  def change
    add_column :products, :price, :decimal, precision: 5, scale: 2
    add_reference :products, :supplier, polymorphic: true, index: true
  end
end
```

2.3. 迁移 DSL

2.3.1. 创建数据表

```
create_table :products do |t|
  t.string :name
```

```
end
```

这个迁移会创建 products 数据表，在数据表中创建 name 字段（后面会介绍，还会自动创建 id 字段）。

默认情况下，create_table 方法会创建名为 id 的主键。通过 :primary_key 选项可以修改主键名（修改后别忘了修改相应的模型）。如果不想生成主键，可以传入 id: false 选项。如果设置数据库的选项，可以在 :options 选择中使用 SQL。例如：

```
create_table :products, options: "ENGINE=BLACKHOLE" do |t|
  t.string :name, null: false
end
```

这样设置之后，会在创建数据表的 SQL 语句后面加上 ENGINE=BLACKHOLE。（MySQL 默认的选项是 ENGINE=InnoDB）

2.3.2. 创建 HABTM 联合数据表

create_join_table 方法用来创建 HABTM 联合数据表。典型的用例如下：

```
create_join_table :products, :categories
```

这段代码会创建一个名为 categories_products 的数据表，包含两个字段：category_id 和 product_id。这两个字段的 :null 选项默认情况都是 false，不过可在 :column_options 选项中设置。

```
create_join_table :products, :categories, column_options: {null: true}
```

这段代码会把 product_id 和 category_id 字段的 :null 选项设为 true。

如果想修改数据表的名字，可以传入 :table_name 选项。例如：

```
create_join_table :products, :categories, table_name: :categorization
```

创建的数据表名为 categorization。

create_join_table 还可接受代码库，用来创建索引（默认无索引）或其他字段。

```
create_join_table :products, :categories do |t|
  t.index :product_id
  t.index :category_id
end
```

2.3.3. 修改数据表

有一个和 create_table 类似地方法，名为 change_table，用来修改现有的数据表。其用法和 create_table 类似，不过传入块的参数知道更多技巧。例如：

```
change_table :products do |t|
```

```
    t.remove :description, :name
    t.string :part_number
    t.index :part_number
    t.rename :upccode, :upc_code
  end
```

这段代码删除了 description 和 name 字段，创建 part_number 字符串字段，并建立索引，最后重命名 upccode 字段。

2.3.4. change 方法

change 是迁移中最常用的方法，大多数情况下都能完成指定的操作，而且 Active Record 知道如何撤这些操作。目前，在 change 方法中只能使用下面的方法：

- add_column
- add_index
- add_reference
- add_timestamps
- create_table
- create_join_table
- drop_table（必须提供代码块）
- drop_join_table（必须提供代码块）
- remove_timestamps
- rename_column

rename_index

remove_reference

rename_table

只要在块中不使用 change、change_default 或 remove 方法，change_table 中的操作也是可逆的。

如果要使用任何其他方法，可以使用 reversible 方法，或者不定义 change 方法，而分别定义 up 和 down 方法。

2.3.5. reversible 方法

Active Record 可能不知如何撤销复杂的迁移操作，这时可以使用 reversible 方法指定运行迁移和撤销迁移时怎么操作。例如：

```
class ExampleMigration < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.references :category
    end

    reversible do |dir|
      dir.up do
        #add a foreign key
        execute <<-SQL
```

```
        ALTER TABLE products
          ADD CONSTRAINT fk_products_categories
          FOREIGN KEY (category_id)
          REFERENCES categories(id)
        SQL
      end
      dir.down do
        execute <<-SQL
          ALTER TABLE products
            DROP FOREIGN KEY fk_products_categories
        SQL
      end
    end

    add_column :users, :home_page_url, :string
    rename_column :users, :email, :email_address
  end
```

使用 reversible 方法还能确保操作按顺序执行。在上面的例子中，如果撤销迁移，down 代码块会在 home_page_url 字段删除后、products 数据表删除前运行。

有时，迁移的操作根本无法撤销，例如删除数据。这是，可以在 down 代码块中抛出 ActiveRecord::IrreversibleMigration 异常。如果有人尝试撤销迁移，会看到一个错误消息，告诉他无法撤销。

2.3.6. 使用 up 和 down 方法

在迁移中可以不用 change 方法，而用 up 和 down 方法。

up 方法定义要对数据库模式做哪些操作，down 方法用来撤销这些操作。也就是说，如果执行 up 后立即执行 down，数据库的模式应该没有任何变化。例如，在 up 中创建了数据表，在 down 方法中就要将其删除。撤销时最好按照添加的相反顺序进行。前一节中的 reversible 用法示例代码可以改成：

```
class ExampleMigration < ActiveRecord::Migration
  def up
    create_table :products do |t|
      t.references :category
    end

    # add a foreign key
    execute <<-SQL
      ALTER TABLE products
        ADD CONSTRAINT fk_products_categories
        FOREIGN KEY (category_id)
        REFERENCES categories(id)
    SQL

    add_column :users, :home_page_url, :string
```

```
    rename_column :users, :email, :email_address
  end

  def down
    rename_column :users, :email_address, :email
    remove_column :users, :home_page_url

    execute <<-SQL
      ALTER TABLE products
        DROP FOREIGN KEY fk_products_categories
    SQL

    drop_table :products
  end
end
```

如果迁移不可撤销，应该在 down 方法中抛出 ActiveRecord::IrreversibleMigration 异常。如果有人尝试撤销迁移，会看到一个错误消息，告诉他无法撤销。

2.3.7. 撤销之前的迁移

Active Record 提供了撤销迁移的功能，通过 revert 方法实现：

```
require_relative '2012121212_example_migration'
```

```
class FixupExampleMigration < ActiveRecord::Migration
  def change
    revert ExampleMigration

    create_table(:apples) do |t|
      t.string :variety
    end
  end
end
```

revert 方法还可接受一个块，定义撤销操作。revert 方法可用来撤销以前迁移的部分操作。例如，ExampleMigration 已经执行，但后来觉得最好还是序列化产品列表。那么，可以编写下面的代码：

```
class SerializeProductListMigration < ActiveRecord::Migration
  def change
    add_column :categories, :product_list

    reversible do |dir|
      dir.up do
        # transfer data from Products to Category#product_list
      end
      dir.down do
        # create Products from Category#product_list
```

```
  end
end

revert do
  # copy-pasted code from ExampleMigration
  create_table :products do |t|
    t.references :category
  end

  reversible do |dir|
    dir.up do
      #add a foreign key
      execute <<-SQL
        ALTER TABLE products
          ADD CONSTRAINT fk_products_categories
          FOREIGN KEY (category_id)
          REFERENCES categories(id)
      SQL
    end
    dir.down do
      execute <<-SQL
        ALTER TABLE products
          DROP FOREIGN KEY fk_products_categories
      SQL
```

```
        end
      end

      # The rest of the migration was ok
    end
  end
end
```

上面这个迁移也可以不用 revert 方法，不过步骤就多了：调换 create_table 和 reversible 的顺序，把 create_table 换成 drop_table，还要对调 up 和 down 中的代码。这些操作都可交给 revert 方法完成。

2.4. 迁移任务

rakedb:create 依照目前的 RAILS_ENV 环境建立数据库

rakedb:create:all 建立所有环境的数据库

rakedb:drop 依照目前的 RAILS_ENV 环境删除数据库

rakedb:drop:all 删除所有环境的数据库

rakedb:migrate 执行 Migration 动作

rakedb:rollback STEP=n 回复上 N 个 Migration 动作

rakedb:migrate:up VERSION=20080906120000 执行特定版本的 Migration

rakedb:migrate:down VERSION=20080906120000 回复特定版本的 Migration

rakedb:version 目前数据库的 Migration 版本

rakedb:seed 执行 db/seeds.rb 加载种子数据

如果需要指定 Rails 环境 ,例如 production ,可以输入 RAILS_ENV=production rake db:migrate

2.4.1. 执行迁移

Rails 提供了很多 Rake 任务，用来执行指定的迁移。

其中最常使用的是 rake db:migrate，执行还没执行的迁移中的 change 或 up 方法。

主要的执行过程如下：

step1：比较最高版本和当前版本， 如果当前版本不是最高版本，就执行所有的大于当前版本的迁移中的 chang 或者 up 方法，直到最高版本。

step2：将最高版本设置为当前版本。

如果没有未运行的迁移 ,直接退出。rake db:migrate 按照迁移文件名中时间戳顺序执行迁移。

如果指定了版本，Active Record 会运行该版本之前的所有迁移。版本就是迁移文件名前的数字部分。例如，要运行 20080906120000 这个迁移，可以执行下面的命令：

$ rake db:migrate VERSION=20080906120000

如果 20080906120000 比当前的版本高，上面的命令就会执行所有 20080906120000 之前（包括 20080906120000）的迁移中的 change 或 up 方法，但不会运行 20080906120000 之后的迁移。如果回滚迁移，则会执行 20080906120000 之前（不包括 20080906120000）的迁移中的 down 方法。

注意，执行 db:migrate 时还会执行 db:schema:dump，更新 db/schema.rb 文件，匹配数据库的结构。

2.4.2. 回滚

还有一个常用的操作时回滚到之前的迁移。例如，迁移代码写错了，想纠正。我们无须查找迁移的版本号，直接执行下面的命令即可：

```
$ rake db:rollback
```

这个命令会回滚上一次迁移，撤销 change 方法中的操作，或者执行 down 方法。如果想撤销多个迁移，可以使用 STEP 参数：

```
$ rake db:rollback STEP=3
```

这个命令会撤销前三次迁移。

db:migrate:redo 命令可以回滚上一次迁移，然后再次执行迁移。和 db:rollback 一样，如果想重做多次迁移，可以使用 STEP 参数。例如：

```
$ rake db:migrate:redo STEP=3
```

这些 Rake 任务的作用和 db:migrate 一样，只是用起来更方便，因为无需查找特定的迁移版本号。

2.4.3. 搭建数据库

rake db:setup 任务会创建数据库，加载模式，并填充种子数据。

2.4.4. 重建数据库

rake db:reset 任务会删除数据库，然后重建，等价于 rake db:drop db:setup。

这个任务和执行所有迁移的作用不同。rake db:reset 使用的是 schema.rb 文件中的内容。如果迁移无法回滚，rake db:reset 起不了作用。详细介绍参见“导出模式”一节。

2.4.5. 种子数据

有些人使用迁移把数据存入数据库：

```
class AddInitialProducts < ActiveRecord::Migration
  def up
    5.times do |i|
      Product.create(name: "Product ##{i}", description: "A product.")
    end
  end

  def down
    Product.delete_all
  end
end
```

Rails 提供了“种子”功能，可以把初始化数据存入数据库。这个功能用起来很简单，在 db/seeds.rb 文件中写一些 Ruby 代码，然后执行 rake db:seed 命令即可：

```
5.times do |i|
  Product.create(name: "Product ##{i}", description: "A product.")
end
```

3. 基本概念

3.1. OR Mapping 对象关系映射

- 无需直接编写 SQL 语句
- 不过度依赖特定的数据库种类，规避数据库差异
- 是 MVC 中的 M（模型），处理数据和业务逻辑
- 其他工作
- 管理 Model 之间的关联关系
- 数据验证
- 事务处理

3.2. Model 的 CRUD 四大操作

3.2.1. 创建

（1）Hash 创建

```
user = User.create(name: "David", occupation: "Code Artist")
```

User 模型中有两个属性，name 和 occupation。

reate 方法会创建一个新纪录，并存入数据库。

（2）创建后手动设置属性创建

使用 new 方法，可以实例化一个新对象，但不会保存：

```
user = User.new
user.name = "David"
user.occupation = "Code Artist"
```

不保存，调用 user.save 可以把记录存入数据库。

（3）使用块创建

```
user = User.new do |u|
   u.name = "David"
   u.occupation = "Code Artist"
end

user.save
```

3.2.2. 读取

```
# return a collection with all users
users = User.all
# return the first user
user = User.first
# return the first user named David
david = User.find_by(name: 'David')
```

```
# find all users named David who are Code Artists and sort by created_at
# in reverse chronological order
users = User.where(name: 'David', occupation: 'Code Artist').order('created_at DESC')
```

3.2.3. 更新

（1）设值方式更新

第一步：根据条件查询 model 对象，

第二步：改其属性

第三步：存入数据库。

```
user = User.find_by(name: 'David')
user.name = 'Dave'
user.save
```

（2）Hash 方式更新

使用 Hash，指定属性名和属性值，例如：

```
user = User.find_by(name: 'David')
user.update(name: 'Dave')
```

（3）批量更新

批量更新多个记录，可以使用类方法 update_all：

User.update_all "max_login_attempts = 3, must_change_password = 'true'"

3.2.4. 删除

得到 model 对象后还可以将其销毁，从数据库中删除。

```
user = User.find_by(name: 'David')

user.destroy
```

3.3. 基础命名约定

3.3.1. 外键

被参考的 model 名称_id 形式命名

例如 item_id，order_id。

创建模型关联后，Active Record 会查找这个字段；

3.3.2. 主键

Active Record 使用整数字段 id 作为表的主键。

使用 Active Record 迁移创建数据表时，会自动创建这个字段；

3.3.3. 可选的字段

created_at - 创建记录时，自动设为当前的时间戳；

updated_at - 更新记录时，自动设为当前的时间戳；

lock_version - 在模型中添加乐观锁定功能；

type - 让模型使用单表继承；

(association_name)_type - 多态关联的类型；

(table_name)_count - 缓存关联对象的数量。例如，posts 表中的 comments_count 字段，缓存每篇文章的评论数；

3.3.4. 文件名：下划线隔词，全部小写

3.3.5. 爆炸方法

empty!和 empty?方法

Ruby 的方法名可以用感叹号（爆炸方法）或者问号（断言方法）结尾。

爆炸方法通常会对接收者造成破坏，断言方法则根据某些条件返回 ture 或 false。

4. Model 生命周期

4.1. 回调

4.1.1. 方法回调

在使用回调之前，要先注册。回调方法的定义和普通的方法一样，然后使用类方法注册：

```
class User < ActiveRecord::Base
  validates :login, :email, presence: true

  before_validation :ensure_login_has_a_value

  protected
   def ensure_login_has_a_value
    if login.nil?
     self.login = email unless email.blank?
    end
   end
end
```

4.1.2. 代码块

这种类方法还可以接受一个代码块。如果操作可以使用一行代码表述，可以考虑使用代码块

形式。

```
class User < ActiveRecord::Base
  validates :login, :email, presence: true

  before_create do
    self.name = login.capitalize if name.blank?
  end
end
```

4.1.3. 事件触发

注册回调时可以指定只在对象生命周期的特定事件发生时执行：

```
class User < ActiveRecord::Base
  before_validation :normalize_name, on: :create

  # :on takes an array as well
  after_validation :set_location, on: [ :create, :update ]

  protected
    def normalize_name
      self.name = self.name.downcase.titleize
    end
```

```
    def set_location
      self.location = LocationService.query(self)
    end
  end
```

一般情况下，都把回调方法定义为受保护的方法或私有方法。如果定义成公共方法，回调就可以在模型外部调用，违背了对象封装原则。

4.2. 监控类型

4.2.1. 创建对象

before_validation

after_validation

before_save

around_save

before_create

around_create

after_create

after_save

4.2.2. 更新对象

before_validation

after_validation

before_save

around_save

before_update

around_update

after_update

after_save

创建和更新对象时都会触发 after_save，但不管注册的顺序，总在 after_create 和 after_update 之后执行。

4.2.3. 销毁对象

before_destroy

around_destroy

after_destroy

4.2.4. 其他监控

after_initialize 回调在 Active Record 对象初始化时执行，包括直接使用 new 方法初始化和从数据库中读取记录。

after_initialize 回调不用直接重定义 Active Record 的 initialize 方法。

after_find 回调在从数据库中读取记录时执行。如果同时注册了 after_find 和 after_initialize 回调，after_find 会先执行。

after_initialize 和 after_find 没有对应的 before_* 回调，但可以像其他回调一样注册。

```
class User < ActiveRecord::Base
  after_initialize do |user|
    puts "You have initialized an object!"
  end

  after_find do |user|

    puts "You have found an object!"
  end
end

>> User.new
You have initialized an object!
=> #<User id: nil>

>> User.first
You have found an object!
You have initialized an object!
=> #<User id: 1>
```

4.2.5. 关联监控

回调能在模型关联中使用，甚至可由关联定义。

假如一个用户发布了多篇文章，如果用户删除了，他发布的文章也应该删除。

下面我们在 Post 模型中注册一个 after_destroy 回调，应用到 User 模型上：

```
class User < ActiveRecord::Base
  has_many :posts, dependent: :destroy
end

class Post < ActiveRecord::Base
  after_destroy :log_destroy_action

  def log_destroy_action
    puts 'Post destroyed'
  end
end

>> user = User.first
=> #<User id: 1>
>> user.posts.create!
```

```
=> #<Post id: 1, user_id: 1>

>> user.destroy

Post destroyed

=> #<User id: 1>
```

4.2.6. 事务监控

还有两个回调会在数据库事务完成时触发：after_commit 和 after_rollback。这两个回调和 after_save 很像，只不过在数据库操作提交或回滚之前不会执行。如果模型要和数据库事务之外的系统交互，就可以使用这两个回调。

例如，在前面的例子中，PictureFile 模型中的记录删除后，还要删除相应的文件。如果执行 after_destroy 回调之后程序抛出了异常，事务就会回滚，文件会被删除，但模型的状态前后不一致。假设在下面的代码中，picture_file_2 是不合法的，那么调用 save! 方法会抛出异常。

```
PictureFile.transaction do
  picture_file_1.destroy
  picture_file_2.save!
end
```

使用 after_commit 回调可以解决这个问题。

```
class PictureFile < ActiveRecord::Base
  after_commit :delete_picture_file_from_disk, on: [:destroy]
```

```
    def delete_picture_file_from_disk
      if File.exist?(filepath)
        File.delete(filepath)
      end
    end
  end
```

4.3. 监控的触发与跳过

4.3.1. 触发监控

下面的方法会触发执行回调：

create

create!

decrement!

destroy

destroy!

destroy_all

increment!

save

save!

save(validate: false)

toggle!

update_attribute

update

update!

valid?

after_find 回调由以下查询方法触发执行：

all

first

find

find_by

find_by_*

find_by_*!

find_by_sql

last

after_initialize 回调在新对象初始化时触发执行。

4.3.2. 跳过监控

和数据验证一样，回调也可跳过，使用下列方法即可：

decrement

decrement_counter

delete

delete_all

increment

increment_counter

toggle

touch

update_column

update_columns

update_all

update_counters

使用这些方法是要特别留心，因为重要的业务逻辑可能在回调中完成。如果没弄懂回调的作用直接跳过，可能导致数据不合法。

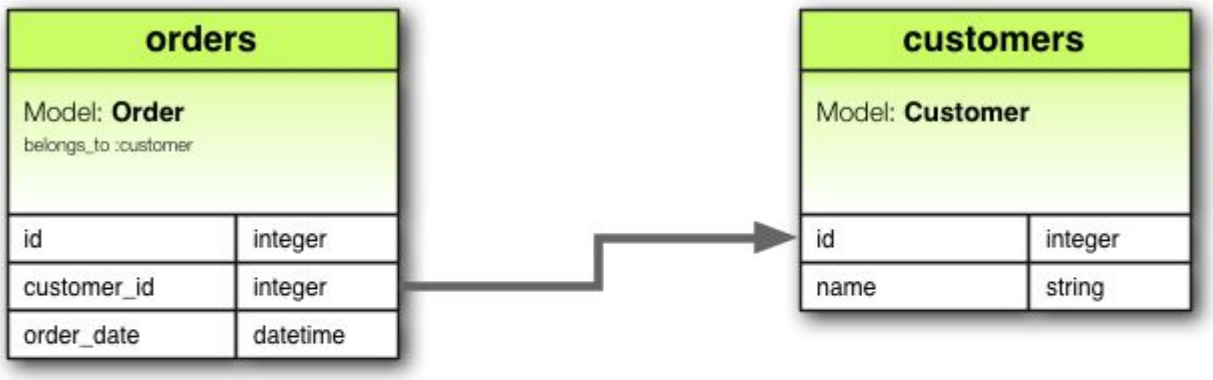
5. 关联

5.1. belongs_to

5.1.1. 场景：多端对一端

声明所在的模型实例属于另一个模型的实例

例如，如果程序中有顾客和订单两个模型，每个订单只能指定给一个顾客，就要这么声明订单模型：




```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

（1）添加的方法

声明 belongs_to 关联后，所在的类自动获得了五个和关联相关的方法：

association(force_reload = false)

association=(associate)

build_association(attributes = {})

create_association(attributes = {})

create_association!(attributes = {})

这五个方法中的 association 要替换成传入 belongs_to 方法的第一个参数。例如，如下的声明：

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

每个 Order 模型实例都获得了这些方法：

customer

customer=

build_customer

create_customer

create_customer!

在 has_one 和 belongs_to 关联中，必须使用 build_* 方法构建关联对象。association.build 方法是在 has_many 和 has_and_belongs_to_many 关联中使用的。创建关联对象要使用 create_* 方法。

①association(force_reload = false)

如果关联的对象存在，association 方法会返回关联对象。如果找不到关联对象，则返回 nil。

@customer = @order.customer

如果关联对象之前已经取回，会返回缓存版本。如果不想使用缓存版本，强制重新从数据库中读取，可以把 force_reload 参数设为 true。

②赋值

association=(associate)

association= 方法用来赋值关联的对象。

这个方法的底层操作是，从关联对象上读取主键，然后把值赋给该主键对应的对象。

@order.customer = @customer

③ build_association(attributes = {})

build_association 方法返回该关联类型的一个新对象。这个对象使用传入的属性初始化，和对象连接的外键会自动设置，但关联对象不会存入数据库。

```
@customer = @order.build_customer(customer_number: 123,
                                  customer_name: "John Doe")
```

④create_association(attributes = {})

```
create_association(attributes = {})
```

create_association 方法返回该关联类型的一个新对象。这个对象使用传入的属性初始化，和对象连接的外键会自动设置，只要能通过所有数据验证，就会把关联对象存入数据库。

```
@customer = @order.create_customer(customer_number: 123,
                                  customer_name: "John Doe")
```

⑤create_association!(attributes = {})

和 create_association 方法作用相同，但是如果记录不合法，会抛出 ActiveRecord::RecordInvalid 异常。

（2）选项

Rails 的默认设置足够智能，能满足常见需求。但有时还是需要定制 belongs_to 关联的行为。定制的方法很简单，声明关联时传入选项或者使用代码块即可。例如，下面的关联使用了两个选项：

```
class Order < ActiveRecord::Base
   belongs_to :customer, dependent: :destroy,
     counter_cache: true
end
```

belongs_to 关联支持以下选项：

:autosave

:class_name

:counter_cache

:dependent

:foreign_key

:inverse_of

:polymorphic

:touch

:validateautosave

①:autosave

如果把 :autosave 选项设为 true，保存父对象时，会自动保存所有子对象，并把标记为析构的子对象销毁。

②:class_name

如果另一个模型无法从关联的名字获取，可以使用 :class_name 选项指定模型名。例如，如果订单属于顾客，但表示顾客的模型是 Patron，就可以这样声明关联：

```
class Order < ActiveRecord::Base
  belongs_to :customer, class_name: "Patron"
end
```

③:counter_cache

:counter_cache 选项可以提高统计所属对象数量操作的效率。假如如下的模型：

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
class Customer < ActiveRecord::Base
  has_many :orders
end
```

这样声明关联后，如果想知道 @customer.orders.size 的结果，就要在数据库中执行 COUNT(*) 查询。如果不想执行这个查询，可以在声明 belongs_to 关联的模型中加入计数缓存功能：

```
class Order < ActiveRecord::Base
  belongs_to :customer, counter_cache: true
end
class Customer < ActiveRecord::Base
  has_many :orders
```

```
end
```

这样声明关联后，Rails 会及时更新缓存，调用 size 方法时返回缓存中的值。

虽然 :counter_cache 选项在声明 belongs_to 关联的模型中设置，但实际使用的字段要添加到关联的模型中。针对上面的例子，要把 orders_count 字段加入 Customer 模型。这个字段的默认名也是可以设置的：

```
class Order < ActiveRecord::Base
  belongs_to :customer, counter_cache: :count_of_orders
end
class Customer < ActiveRecord::Base
  has_many :orders
end
```

计数缓存字段通过 attr_readonly 方法加入关联模型的只读属性列表中。

④:dependent

:dependent 选项的值有两个：

:destroy：销毁对象时，也会在关联对象上调用 destroy 方法；

:delete：销毁对象时，关联的对象不会调用 destroy 方法，而是直接从数据库中删除；

在 belongs_to 关联和 has_many 关联配对时，不应该设置这个选项，否则会导致数据库中出现孤儿记录。

⑤:foreign_key

按照约定，用来存储外键的字段名是关联名后加 _id。:foreign_key 选项可以设置要使用的外键名：

```
class Order < ActiveRecord::Base
  belongs_to :customer, class_name: "Patron",
                foreign_key: "patron_id"
end
```

不管怎样，Rails 都不会自动创建外键字段，你要自己在迁移中创建。

⑥:inverse_of

:inverse_of 选项指定 belongs_to 关联另一端的 has_many 和 has_one 关联名。不能和 :polymorphic 选项一起使用。

```
class Customer < ActiveRecord::Base
  has_many :orders, inverse_of: :customer
end

class Order < ActiveRecord::Base
```

```
    belongs_to :customer, inverse_of: :orders
end
```

⑦:polymorphic

:polymorphic 选项为 true 时表明这是个多态关联。前文已经详细介绍过多态关联。

⑧:touch

如果把 :touch 选项设为 true，保存或销毁对象时，关联对象的 updated_at 或 updated_on 字段会自动设为当前时间戳。

```
class Order < ActiveRecord::Base
    belongs_to :customer, touch: true
end

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

在这个例子中，保存或销毁订单后，会更新关联的顾客中的时间戳。还可指定要更新哪个字段的时间戳：

```
class Order < ActiveRecord::Base
    belongs_to :customer, touch: :orders_updated_at
end
```

⑨:validate

如果把 :validate 选项设为 true，保存对象时，会同时验证关联对象。该选项的默认值是 false，保存对象时不验证关联对象。

5.1.2. 属性：拥有者为多端，单数形式

在 belongs_to 关联声明中必须使用单数形式。

如果在一端使用复数形式，程序会报错，提示未初始化常量 。 Rails 自动使用关联中的名字引用类名。如果关联中的名字错误的使用复数，引用的类也就变成了复数。

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

5.1.3. 迁移：外键在多端表

```
class CreateOrders < ActiveRecord::Migration
  def change
    create_table :customers do |t|
      t.string :name
      t.timestamps
    end
```

```
    create_table :orders do |t|
      t.belongs_to :customer
      t.datetime :order_date
      t.timestamps
    end
  end
end
```

5.2. has_one

5.2.1. 场景：多对一模拟一对一

```
class Supplier < ActiveRecord::Base
  has_one :account
end
```

has_one 关联建立两个模型之间的一对一关系。用数据库的行话说，这种关联的意思是外键在另一个类中。如果外键在这个类中，应该使用 belongs_to 关联。

（3）has_one 关联添加的方法

声明 has_one 关联后，声明所在的类自动获得了五个关联相关的方法：

```
association(force_reload = false)
```

association=(associate)

build_association(attributes = {})

create_association(attributes = {})

create_association!(attributes = {})

这五个方法中的 association 要替换成传入 has_one 方法的第一个参数。

例如，如下的声明：

```
class Supplier < ActiveRecord::Base
   has_one :account
end
```

每个 Supplier 模型实例都获得了这些方法：

account

account=

build_account

create_account

create_account!

在 has_one 和 belongs_to 关联中，必须使用 build_* 方法构建关联对象。association.build 方法是在 has_many 和 has_and_belongs_to_many 关联中使用的。

创建关联对象要使用 create_* 方法。

①association(force_reload = false)

如果关联的对象存在，association 方法会返回关联对象。如果找不到关联对象，则返回 nil。

@account = @supplier.account

如果关联对象之前已经取回，会返回缓存版本。如果不想使用缓存版本，强制重新从数据库中读取，可以把 force_reload 参数设为 true。

②association=(associate)

association= 方法用来赋值关联的对象。这个方法的底层操作是，从关联对象上读取主键，然后把值赋给该主键对应的关联对象。

@supplier.account = @account

③build_association(attributes = {})

build_association 方法返回该关联类型的一个新对象。这个对象使用传入的属性初始化，和对象连接的外键会自动设置，但关联对象不会存入数据库。

@account = @supplier.build_account(terms: "Net 30")

④create_association(attributes = {})

create_association 方法返回该关联类型的一个新对象。这个对象使用传入的属性初始化，和对象连接的外键会自动设置，只要能通过所有数据验证，就会把关联对象存入数据库。

```
@account = @supplier.create_account(terms: "Net 30")
```

⑤create_association!(attributes = {})

```
create_association!(attributes = {})
```

和 create_association 方法作用相同，但是如果记录不合法，会抛出 ActiveRecord::RecordInvalid 异常。

（4）has_one 方法的选项

Rails 的默认设置足够智能，能满足常见需求。但有时还是需要定制 has_one 关联的行为。定制的方法很简单，声明关联时传入选项即可。例如，下面的关联使用了两个选项：

```
class Supplier < ActiveRecord::Base
   has_one :account, class_name: "Billing", dependent: :nullify
end
```

has_one 关联支持以下选项：

```
:as
```

:autosave

:class_name

:dependent

:foreign_key

:inverse_of

:primary_key

:source

:source_type

:through

:validate

①:as

关联还有一种高级用法，“多态关联”。在多态关联中，在同一个关联中，模型可以属于其他多个模型。例如，图片模型可以属于雇员模型或者产品模型，模型的定义如下：

```
class Picture < ActiveRecord::Base
  belongs_to :imageable, polymorphic: true
end

class Employee < ActiveRecord::Base
  has_many :pictures, as: :imageable
end
```

```
class Product < ActiveRecord::Base
  has_many :pictures, as: :imageable
end
```

在 belongs_to 中指定使用多态，可以理解成创建了一个接口，可供任何一个模型使用。在 Employee 模型实例上，可以使用 @employee.pictures 获取图片集合。类似地，可使用 @product.pictures 获取产品的图片。

在 Picture 模型的实例上，可以使用 @picture.imageable 获取父对象。不过事先要在声明多态接口的模型中创建外键字段和类型字段：

```
class CreatePictures < ActiveRecord::Migration
  def change
    create_table :pictures do |t|
      t.string  :name
      t.integer :imageable_id
      t.string  :imageable_type
      t.timestamps
    end
  end
end
```

上面的迁移可以使用 t.references 简化：

```
class CreatePictures < ActiveRecord::Migration
  def change
    create_table :pictures do |t|
      t.string :name
      t.references :imageable, polymorphic: true
      t.timestamps
    end
  end
end
```

②:autosave

如果把 :autosave 选项设为 true，保存父对象时，会自动保存所有子对象，并把标记为析构的子对象销毁。

③:dependent

设置销毁拥有者时要怎么处理关联对象：

:destroy：也销毁关联对象；

:delete：直接把关联对象对数据库中删除，因此不会执行回调；

:nullify：把外键设为 NULL，不会执行回调；

:restrict_with_exception：有关联的对象时抛出异常；

:restrict_with_error：有关联的对象时，向拥有者添加一个错误；

如果在数据库层设置了 NOT NULL 约束，就不能使用 :nullify 选项。如果 :dependent 选项没有销毁关联，就无法修改关联对象，因为关联对象的外键设置为不接受 NULL。

④:foreign_key

按照约定，在另一个模型中用来存储外键的字段名是模型名后加 _id。:foreign_key 选项可以设置要使用的外键名：

```
class Supplier < ActiveRecord::Base
  has_one :account, foreign_key: "supp_id"
end
```

不管怎样，Rails 都不会自动创建外键字段，你要自己在迁移中创建。

⑤:inverse_of

:inverse_of 选项指定 has_one 关联另一端的 belongs_to 关联名。不能和 :through 或 :as 选项一起使用。

```
class Supplier < ActiveRecord::Base
  has_one :account, inverse_of: :supplier
end

class Account < ActiveRecord::Base
```

```
  belongs_to :supplier, inverse_of: :account
end
```

⑥:primary_key

按照约定，用来存储该模型主键的字段名 id。:primary_key 选项可以设置要使用的主键名。

⑦:source

:source 选项指定 has_one :through 关联的关联源名字。

⑧:source_type

:source_type 选项指定 has_one :through 关联中用来处理多态关联的关联源类型。

⑨:through

:through 选项指定用来执行查询的连接模型。前文详细介绍过 has_one :through 关联。

⑩:validate

如果把 :validate 选项设为 true，保存对象时，会同时验证关联对象。该选项的默认值是 false，保存对象时不验证关联对象。

⑪has_one 的作用域

有时可能需要定制 has_one 关联使用的查询方式，定制的查询可在作用域代码块中指定。例如：

```
class Supplier < ActiveRecord::Base
   has_one :account, -> { where active: true }
end
```

在作用域代码块中可以使用任何一个标准的查询方法。下面分别介绍这几个方法：

where

includes

readonly

select

1）where

where 方法指定关联对象必须满足的条件。

```
class Supplier < ActiveRecord::Base
   has_one :account, -> { where "confirmed = 1" }
end
```

2）includes

includes 方法指定使用关联时要按需加载的间接关联。例如，有如下的模型：

```
class Supplier < ActiveRecord::Base
   has_one :account
end
```

```
class Account < ActiveRecord::Base
  belongs_to :supplier
  belongs_to :representative
end

class Representative < ActiveRecord::Base
  has_many :accounts
end
```

如果经常要直接获取供应商代表（@supplier.account.representative），就可以把代表引入供应商和账户的关联中：

```
class Supplier < ActiveRecord::Base
  has_one :account, -> { includes :representative }
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  belongs_to :representative
end

class Representative < ActiveRecord::Base
  has_many :accounts
```

end

3）readonly

如果使用 readonly，通过关联获取的对象就是只读的。

4）select

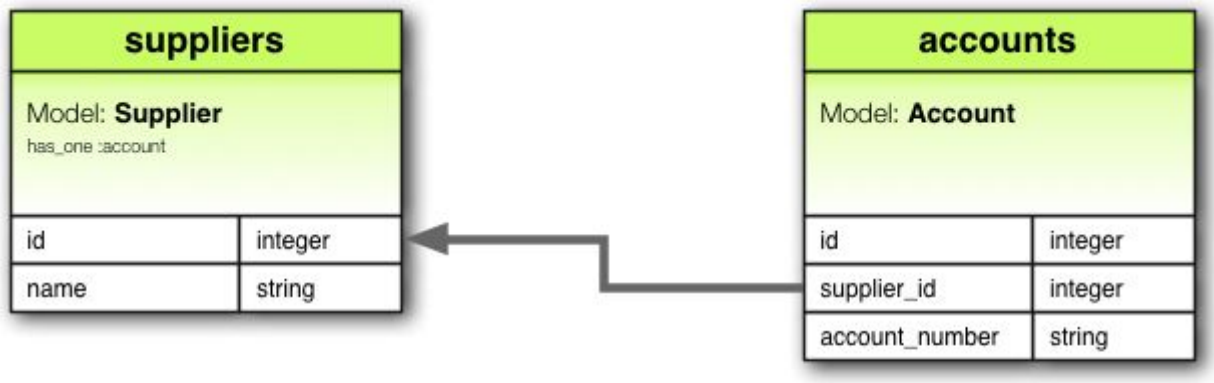
select 方法会覆盖获取关联对象使用的 SQL SELECT 子句。默认情况下，Rails 会读取所有字段。

⑫关联的对象是否存在

检查关联的对象是否存在可以使用 association.nil? 方法：

```
if @supplier.account.nil?
  @msg = "No account found for this supplier"
end
```

5.2.2. 属性：拥有者为一端，单数形式




```
class Supplier < ActiveRecord::Base
  has_one :account
end
```

5.2.3. 迁移：外键保持多端表

```
class CreateSuppliers < ActiveRecord::Migration
  def change
    create_table :suppliers do |t|
      t.string :name
      t.timestamps
    end

    create_table :accounts do |t|
      t.belongs_to :supplier
      t.string :account_number
      t.timestamps
    end
  end
end
```

或者

```
class CreateSuppliers < ActiveRecord::Migration
    def change
      create_table :suppliers do |t|
        t.string    :name
```

```
      t.timestamps
    end

    create_table :accounts do |t|
      t.integer :supplier_id
      t.string  :account_number
      t.timestamps
    end
  end
end
```

5.3. has_many

5.3.1. 场景：一对多

has_many 关联表示模型的实例有零个或多个另一个模型的实例

顾客有多个订单模型

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

has_many 关联建立两个模型之间的一对多关系。用数据库的行话说，这种关联的意思是外

键在另一个类中，指向这个类的实例。

（5）添加的方法

声明 has_many 关联后，声明所在的类自动获得了 16 个关联相关的方法：

collection(force_reload = false)

collection<<(object, ...)

collection.delete(object, ...)

collection.destroy(object, ...)

collection=objects

collection_singular_ids

collection_singular_ids=ids

collection.clear

collection.empty?

collection.size

collection.find(...)

collection.where(...)

collection.exists?(...)

collection.build(attributes = {}, ...)

collection.create(attributes = {})

collection.create!(attributes = {})

这些个方法中的 collection 要替换成传入 has_many 方法的第一个参数。collection_singular 要替换成第一个参数的单数形式。例如，如下的声明：

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

每个 Customer 模型实例都获得了这些方法：

```
orders(force_reload = false)
orders<<(object, ...)
orders.delete(object, ...)
orders.destroy(object, ...)
orders=objects
order_ids
order_ids=ids
orders.clear
orders.empty?
orders.size
orders.find(...)
orders.where(...)
orders.exists?(...)
orders.build(attributes = {}, ...)
orders.create(attributes = {})
orders.create!(attributes = {})
```

①collection(force_reload = false)

collection 方法返回一个数组，包含所有关联的对象。如果没有关联的对象，则返回空数组。

@orders = @customer.orders

②collection<<(object, ...)

collection<< 方法向关联对象数组中添加一个或多个对象，并把各所加对象的外键设为调用此方法的模型的主键。

@customer.orders << @order1

③collection.delete(object, ...)

collection.delete 方法从关联对象数组中删除一个或多个对象，并把删除的对象外键设为 NULL。

@customer.orders.delete(@order1)

如果关联设置了 dependent: :destroy，还会销毁关联对象；如果关联设置了 dependent: :delete_all，还会删除关联对象。

④collection.destroy(object, ...)

collection.destroy 方法在关联对象上调用 destroy 方法，从关联对象数组中删除一个或多个对象。

@customer.orders.destroy(@order1)

对象会从数据库中删除，忽略 :dependent 选项。

⑤collection=objects

collection= 让关联对象数组只包含指定的对象，根据需求会添加或删除对象。

⑥collection_singular_ids

collection_singular_ids 返回一个数组，包含关联对象数组中各对象的 ID。

@order_ids = @customer.order_ids

⑦collection_singular_ids=ids

collection_singular_ids= 方法让数组中只包含指定的主键，根据需要增删 ID。

⑧collection.clear

collection.clear 方法删除数组中的所有对象。如果关联中指定了 dependent: :destroy 选项，会销毁关联对象；如果关联中指定了 dependent: :delete_all 选项，会直接从数据库中删除对象，然后再把外键设为 NULL。

⑨collection.empty?

如果关联数组中没有关联对象，collection.empty? 方法返回 true。

```
<% if @customer.orders.empty? %>
  No Orders Found
<% end %>
```

⑩collection.size

collection.size 返回关联对象数组中的对象数量。

```
@order_count = @customer.orders.size
```

⑪collection.find(...)

collection.find 方法在关联对象数组中查找对象，句法和可用选项跟 ActiveRecord::Base.find 方法一样。

```
@open_orders = @customer.orders.find(1)
```

⑫collection.where(...)

collection.where 方法根据指定的条件在关联对象数组中查找对象，但会惰性加载对象，用到对象时才会执行查询。

```
@open_orders = @customer.orders.where(open: true) # No query yet
```

@open_order = @open_orders.first # Now the database will be queried

13 collection.exists?(...)

collection.exists? 方法根据指定的条件检查关联对象数组中是否有符合条件的对象，句法和可用选项跟 ActiveRecord::Base.exists? 方法一样。

14 collection.build(attributes = {}, ...)

collection.build 方法返回一个或多个此种关联类型的新对象。这些对象会使用传入的属性初始化，还会创建对应的外键，但不会保存关联对象。

```
@order = @customer.orders.build(order_date: Time.now,
                                 order_number: "A12345")
```

15 collection.create(attributes = {})

collection.create 方法返回一个此种关联类型的新对象。这个对象会使用传入的属性初始化，还会创建对应的外键，只要能通过所有数据验证，就会保存关联对象。

```
@order = @customer.orders.create(order_date: Time.now,
                                 order_number: "A12345")
```

16 **collection.create!(attributes = {})**

作用和 collection.create 相同，但如果记录不合法会抛出 ActiveRecord::RecordInvalid 异常。

（6）has_many 方法的选项

Rails 的默认设置足够智能，能满足常见需求。但有时还是需要定制 has_many 关联的行为。定制的方法很简单，声明关联时传入选项即可。例如，下面的关联使用了两个选项：

```
class Customer < ActiveRecord::Base
  has_many :orders, dependent: :delete_all, validate: :false
end
```

has_many 关联支持以下选项：

```
:as
:autosave
:class_name
:dependent
:foreign_key
:inverse_of
:primary_key
:source
:source_type
:through
```

:validate

①:as

:as 选项表明这是多态关联。前文已经详细介绍过多态关联。

②:autosave

如果把 :autosave 选项设为 true，保存父对象时，会自动保存所有子对象，并把标记为析构的子对象销毁。

③:class_name

如果另一个模型无法从关联的名字获取，可以使用 :class_name 选项指定模型名。例如，顾客有多个订单，但表示订单的模型是 Transaction，就可以这样声明关联：

```
class Customer < ActiveRecord::Base
  has_many :orders, class_name: "Transaction"
end
```

④:dependent

设置销毁拥有者时要怎么处理关联对象：

:destroy：也销毁所有关联的对象；

:delete_all：直接把所有关联对象对数据库中删除，因此不会执行回调；

:nullify：把外键设为 NULL，不会执行回调；

:restrict_with_exception：有关联的对象时抛出异常；

:restrict_with_error：有关联的对象时，向拥有者添加一个错误；

如果声明关联时指定了 :through 选项，会忽略这个选项。

⑤:foreign_key

按照约定，另一个模型中用来存储外键的字段名是模型名后加 _id。:foreign_key 选项可以设置要使用的外键名：

```
class Customer < ActiveRecord::Base
  has_many :orders, foreign_key: "cust_id"
end
```

不管怎样，Rails 都不会自动创建外键字段，你要自己在迁移中创建。

⑥:inverse_of

:inverse_of 选项指定 has_many 关联另一端的 belongs_to 关联名。不能和 :through 或 :as 选项一起使用。

```
class Customer < ActiveRecord::Base
```

```
  has_many :orders, inverse_of: :customer
end

class Order < ActiveRecord::Base
  belongs_to :customer, inverse_of: :orders
end
```

⑦:primary_key

按照约定，用来存储该模型主键的字段名 id。:primary_key 选项可以设置要使用的主键名。

假设 users 表的主键是 id ,但还有一个 guid 字段。根据要求 ,todos 表中应该使用 guid 字段，而不是 id 字段。这种需求可以这么实现：

```
class User < ActiveRecord::Base
  has_many :todos, primary_key: :guid
end
```

如果执行 @user.todos.create 创建新的待办事项，那么 @todo.user_id 就是 guid 字段中的值。

⑧:source

:source 选项指定 has_many :through 关联的关联源名字。只有无法从关联名种解出关联源的名字时才需要设置这个选项。

⑨:source_type

:source_type 选项指定 has_many :through 关联中用来处理多态关联的关联源类型。

⑩:through

:through 选项指定用来执行查询的连接模型。has_many :through 关联是实现多对多关联的一种方式，前文已经介绍过。

⑪:validate

如果把 :validate 选项设为 false，保存对象时，不会验证关联对象。该选项的默认值是 true，保存对象验证关联的对象。

（7）has_many 的作用域

有时可能需要定制 has_many 关联使用的查询方式，定制的查询可在作用域代码块中指定。例如：

```
class Customer < ActiveRecord::Base
  has_many :orders, -> { where processed: true }
end
```

在作用域代码块中可以使用任何一个标准的查询方法。下面分别介绍 where 这几个方法：

```
where
extending
```

group

includes

limit

offset

order

readonly

select

uniq

①where

where 方法指定关联对象必须满足的条件。

```
class Customer < ActiveRecord::Base
   has_many :confirmed_orders, -> { where "confirmed = 1" },
     class_name: "Order"
end
```

条件还可以使用 Hash 的形式指定：

```
class Customer < ActiveRecord::Base
   has_many :confirmed_orders, -> { where confirmed: true },
                        class_name: "Order"
end
```

如果 where 使用 Hash 形式，通过这个关联创建的记录会自动使用 Hash 中的作用域。针

对上面的例子，使用 @customer.confirmed_orders.create 或 @customer.confirmed_orders.build 创建订单时，会自动把 confirmed 字段的值设为 true。

②extending

extending 方法指定一个模块名，用来扩展关联代理。后文会详细介绍关联扩展。

③group

group 方法指定一个属性名，用在 SQL GROUP BY 子句中，分组查询结果。

```
class Customer < ActiveRecord::Base
  has_many :line_items, -> { group 'orders.id' },
                  through: :orders
end
```

④includes

includes 方法指定使用关联时要按需加载的间接关联。例如，有如下的模型：

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

class Order < ActiveRecord::Base
  belongs_to :customer
```

```
    has_many :line_items
end

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

如果经常要直接获取顾客购买的商品（@customer.orders.line_items），就可以把商品引入顾客和订单的关联中：

```
class Customer < ActiveRecord::Base
    has_many :orders, -> { includes :line_items }
end

class Order < ActiveRecord::Base
    belongs_to :customer
    has_many :line_items
end

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

⑤limit

limit 方法限制通过关联获取的对象数量。

```
class Customer < ActiveRecord::Base
   has_many :recent_orders,
    -> { order('order_date desc').limit(100) },
    class_name: "Order",
end
```

⑥offset

offset 方法指定通过关联获取对象时的偏移量。例如，-> { offset(11) } 会跳过前 11 个记录。

⑦order

order 方法指定获取关联对象时使用的排序方式，用于 SQL ORDER BY 子句。

```
class Customer < ActiveRecord::Base
   has_many :orders, -> { order "date_confirmed DESC" }
end
```

⑧distinct

使用 distinct 方法可以确保集合中没有重复的对象，和 :through 选项一起使用最有用。

```
class Person < ActiveRecord::Base
  has_many :readings
  has_many :posts, through: :readings
end

person = Person.create(name: 'John')
post   = Post.create(name: 'a1')
person.posts << post
person.posts << post
person.posts.inspect # => [#<Post id: 5, name: "a1">, #<Post id: 5, name: "a1">]
Reading.all.inspect   # => [#<Reading id: 12, person_id: 5, post_id: 5>, #<Reading id: 13, person_id: 5, post_id: 5>]
```

在上面的代码中，读者读了两篇文章，即使是同一篇文章，person.posts 也会返回两个对象。

下面我们加入 distinct 方法：

```
class Person
  has_many :readings
  has_many :posts, -> { distinct }, through: :readings
end

person = Person.create(name: 'Honda')
```

```
post     = Post.create(name: 'a1')

person.posts << post

person.posts << post

person.posts.inspect # => [#<Post id: 7, name: "a1">]

Reading.all.inspect     # => [#<Reading id: 16, person_id: 7, post_id: 7>, #<Reading id: 17, person_id: 7, post_id: 7>]
```

在这段代码中，读者还是读了两篇文章，但 person.posts 只返回一个对象，因为加载的集合已经去除了重复元素。

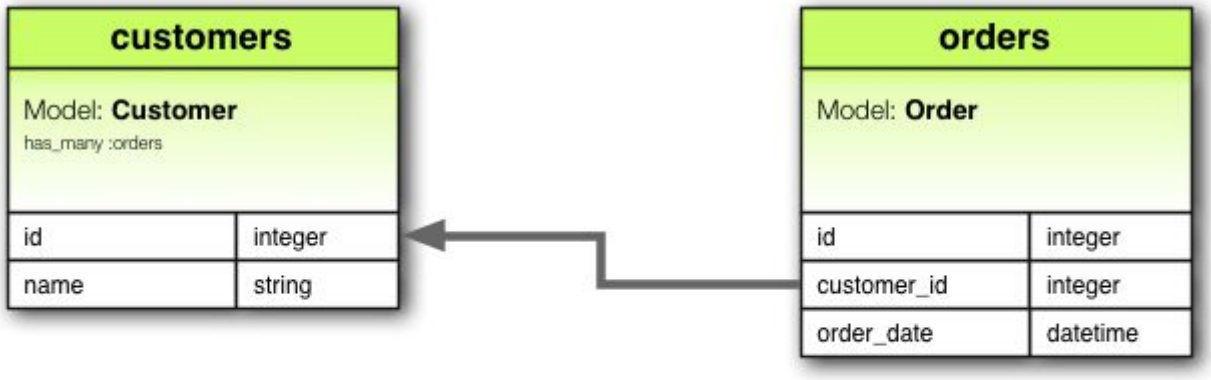
如果要确保只把不重复的记录写入关联模型的数据表(这样就不会从数据库中获取重复记录了），需要在数据表上添加唯一性索引。例如，数据表名为 person_posts，我们要保证其中所有的文章都没重复，可以在迁移中加入以下代码：

```
add_index :person_posts, :post, unique: true
```

注意，使用 include? 等方法检查唯一性可能导致条件竞争。不要使用 include? 确保关联的唯一性。还是以前面的文章模型为例，下面的代码会导致条件竞争，因为多个用户可能会同时执行这一操作：

```
person.posts << post unless person.posts.include?(post)
```

5.3.2. 属性：拥有者为一端，复数形式




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

5.3.3. 迁移：外键保持多端表

```
class CreateCustomers < ActiveRecord::Migration
  def change
    create_table :customers do |t|
      t.string :name
      t.timestamps
    end

    create_table :orders do |t|
      t.belongs_to :customer
      t.datetime :order_date
      t.timestamps
    end
```

```
    end
end
```

5.4. has_many :through

5.4.1. 场景：多对多

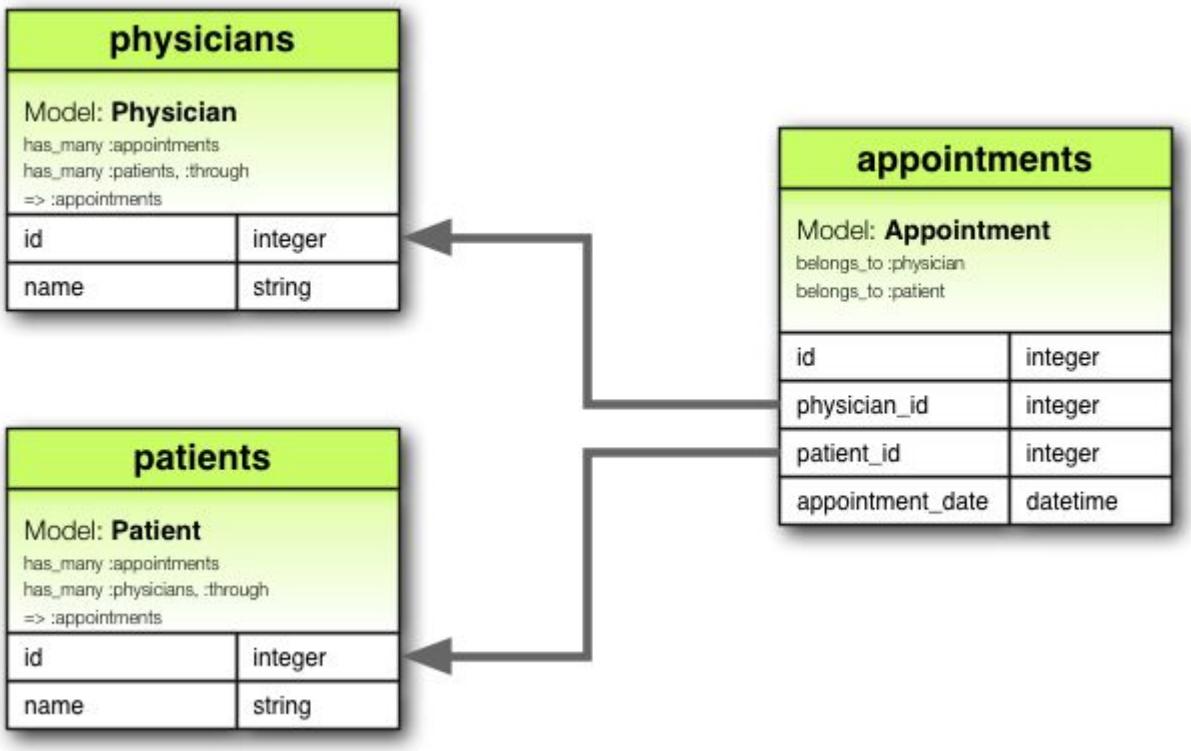
在看病过程中，病人要和医生预约时间。

```
class Physician < ActiveRecord::Base
    has_many :appointments
    has_many :patients, through: :appointments
end

class Appointment < ActiveRecord::Base
    belongs_to :physician
    belongs_to :patient
end

class Patient < ActiveRecord::Base
    has_many :appointments
    has_many :physicians, through: :appointments
end
```

5.4.2. 属性：拥有者复数形式；属于者单数形式




```
class Physician < ActiveRecord::Base
  has_many :appointments
  has_many :patients, :through => :appointments
end

class Appointment < ActiveRecord::Base
  belongs_to :physician
  belongs_to :patient
end

class Patient < ActiveRecord::Base
  has_many :appointments
  has_many :physicians, :through => :appointments
end
```

5.4.3. 迁移：外键保持在中间表

```
class CreateAppointments < ActiveRecord::Migration
  def change
    create_table :physicians do |t|
      t.string :name
      t.timestamps
    end

    create_table :patients do |t|
      t.string :name
      t.timestamps
    end

    create_table :appointments do |t|
      t.belongs_to :physician
      t.belongs_to :patient
      t.datetime :appointment_date
      t.timestamps
    end
  end
end
```

5.4.4. 两层一对多关系的简化

has_many :through 还可用来简化嵌套的 has_many 关联。例如，一个文档分为多个部分，每一部分又有多个段落，如果想使用简单的方式获取文档中的所有段落，可以这么做：

```
class Document < ActiveRecord::Base
  has_many :sections
  has_many :paragraphs, through: :sections
end

class Section < ActiveRecord::Base
  belongs_to :document
  has_many :paragraphs
end

class Paragraph < ActiveRecord::Base
  belongs_to :section
end
```

加上 through: :sections 后，Rails 就能理解这段代码：

```
@document.paragraphs
```

5.5. has_one :through

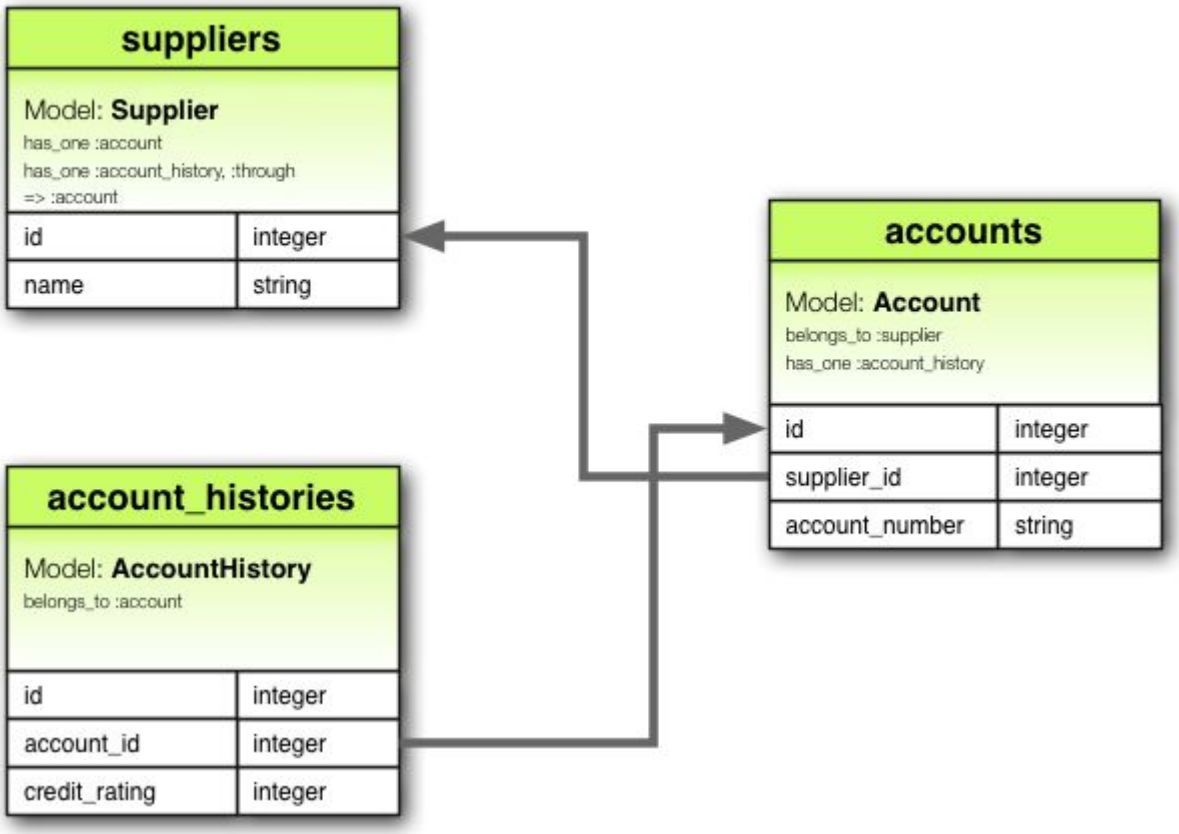
5.5.1. 场景：直接访问第三者

一个模型通过第二层模型，直接拥有第三层模型的实例。

```
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, through: :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```




```
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, :through => :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```

5.5.2. 迁移：在下层表

```
class CreateAccountHistories < ActiveRecord::Migration

  def change
```

```
    create_table :suppliers do |t|
      t.string :name
      t.timestamps
    end

    create_table :accounts do |t|
      t.belongs_to :supplier
      t.string :account_number
      t.timestamps
    end

    create_table :account_histories do |t|
      t.belongs_to :account
      t.integer :credit_rating
      t.timestamps
    end
  end
end
```

5.6. has_and_belongs_to_many

5.6.1. 场景：多对多

多对多关系，不借由第三个模型。

疑问：中间表是否要有？

例如，程序中有装配体和零件两个模型，每个装配体中有多个零件，每个零件又可用于多个装配体，这时可以按照下面的方式定义模型：

```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

has_and_belongs_to_many 关联建立两个模型之间的多对多关系。用数据库的行话说，这种关联的意思是有个连接数据表包含指向这两个类的外键。

（8）关联添加的方法

声明 has_and_belongs_to_many 关联后，声明所在的类自动获得了 16 个关联相关的方法：

```
collection(force_reload = false)
collection<<(object, ...)
collection.delete(object, ...)
collection.destroy(object, ...)
```

collection=objects

collection_singular_ids

collection_singular_ids=ids

collection.clear

collection.empty?

collection.size

collection.find(...)

collection.where(...)

collection.exists?(...)

collection.build(attributes = {})

collection.create(attributes = {})

collection.create!(attributes = {})

这些个方法中的 collection 要替换成传入 has_and_belongs_to_many 方法的第一个参数。collection_singular 要替换成第一个参数的单数形式。例如，如下的声明：

```
class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

每个 Part 模型实例都获得了这些方法：

assemblies(force_reload = false)

assemblies<<(object, ...)

assemblies.delete(object, ...)

```
assemblies.destroy(object, ...)
assemblies=objects
assembly_ids
assembly_ids=ids
assemblies.clear
assemblies.empty?
assemblies.size
assemblies.find(...)
assemblies.where(...)
assemblies.exists?(...)
assemblies.build(attributes = {}, ...)
assemblies.create(attributes = {})
assemblies.create!(attributes = {})
```

在 has_and_belongs_to_many 关联的连接数据表中使用其他字段的功能已经废弃。如果在多对多关联中需要使用这么复杂的数据表，可以用 has_many :through 关联代替 has_and_belongs_to_many 关联。

（9）作用域

有时可能需要定制 has_and_belongs_to_many 关联使用的查询方式，定制的查询可在作用域代码块中指定。例如：

```
class Parts < ActiveRecord::Base
```

```
  has_and_belongs_to_many :assemblies, -> { where active: true }
end
```

在作用域代码块中可以使用任何一个标准的查询方法。下面分别介绍这几个方法：

where

extending

group

includes

limit

offset

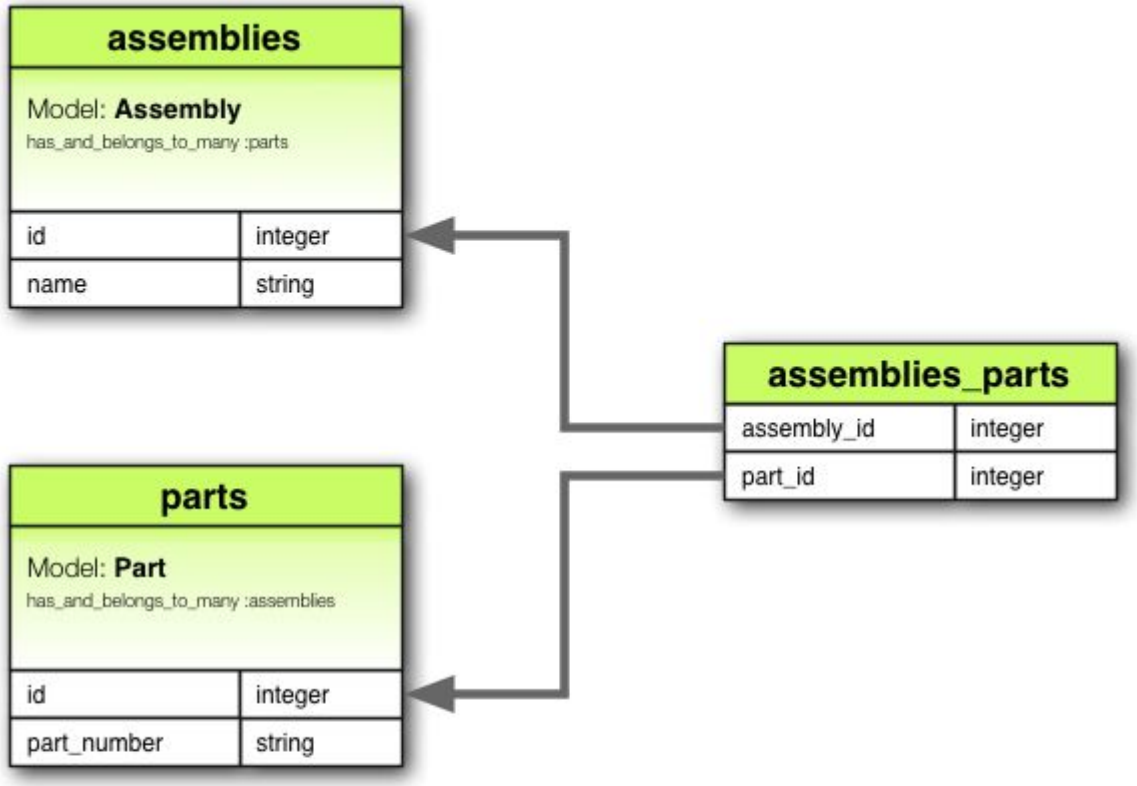
order

readonly

select

uniq

5.6.2. 迁移：关联表没有 id




```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

```
class CreateAssembliesAndParts < ActiveRecord::Migration
  def change
    create_table :assemblies do |t|
      t.string :name
```

```
    t.timestamps
  end

  create_table :parts do |t|
    t.string :part_number
    t.timestamps
  end

  create_table :assemblies_parts, id: false do |t|
    t.belongs_to :assembly
    t.belongs_to :part
  end
 end
end
```

5.6.3. 区分 has_many :through

根据经验，如果关联的第三个模型要作为独立实体使用，要用 has_many :through 关联；如果不需要使用第三个模型，用简单的 has_and_belongs_to_many 关联即可（不过要记得在数据库中创建连接数据表）。

如果需要做数据验证、回调，或者连接模型上要用到其他属性，此时就要使用 has_many :through 关联。

```
class Assembly < ActiveRecord::Base
  has_many :manifests
  has_many :parts, through: :manifests
end

class Manifest < ActiveRecord::Base
  belongs_to :assembly
  belongs_to :part
end

class Part < ActiveRecord::Base
  has_many :manifests
  has_many :assemblies, through: :manifests
end
```

5.7. 多态关联

5.7.1. 场景：分类型多对一

在多态关联中，在同一个关联中，模型可以属于其他多个模型。

图片模型可以属于雇员模型或者产品模型，模型的定义如下：

```
class Picture < ActiveRecord::Base
  belongs_to :imageable, polymorphic: true
end
```

```
class Employee < ActiveRecord::Base
  has_many :pictures, as: :imageable
end

class Product < ActiveRecord::Base
  has_many :pictures, as: :imageable
end
```

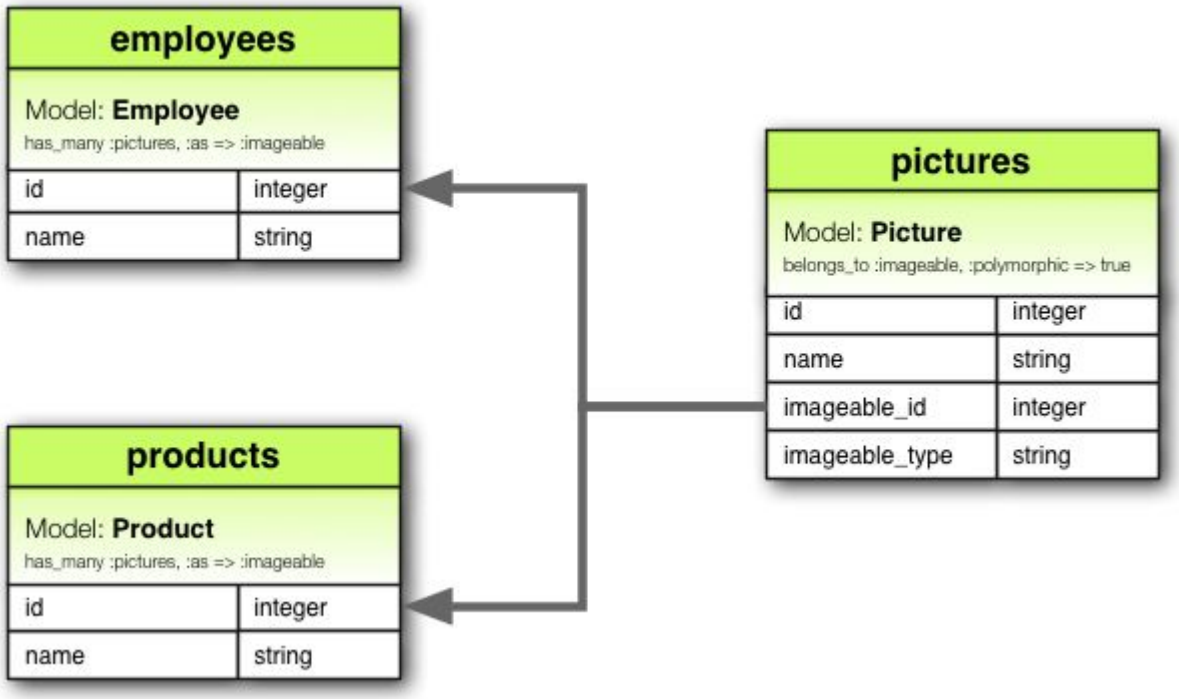
5.7.2. 迁移

```
class CreatePictures < ActiveRecord::Migration
 def change
   create_table :pictures do |t|
     t.string :name
     t.integer :imageable_id
     t.string :imageable_type
     t.timestamps
   end
 end
end
```

或者

```
class CreatePictures < ActiveRecord::Migration
```

```
def change
  create_table :pictures do |t|
    t.string :name
    t.references :imageable, polymorphic: true
    t.timestamps
  end
```




```
class Picture < ActiveRecord::Base
  belongs_to :imageable, :polymorphic => true
end

class Employee < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end

class Product < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end
```

end

end

5.8. 自连接

5.8.1. 场景： 关系在同一个表

在一个数据表中保存所有雇员的信息，但要建立经理和下属之间的关系。这种情况可以使用自连接关联解决

```
class Employee < ActiveRecord::Base
  has_many :subordinates, class_name: "Employee",
                  foreign_key: "manager_id"

  belongs_to :manager, class_name: "Employee"
end
```

这样定义模型后，就可以使用 @employee.subordinates 和 @employee.manager 了。

5.8.2. 迁移

```
class CreateEmployees < ActiveRecord::Migration
  def change
    create_table :employees do |t|
      t.references :manager
      t.timestamps
    end
  end
```

```
end
```

5.9. 关联要点

5.9.1. 缓存控制

（10）缓存共享

关联添加的方法都会使用缓存，记录最近一次查询结果，以备后用。缓存还会在方法之间共享。例如：

```
customer.orders                 # retrieves orders from the database
customer.orders.size            # uses the cached copy of orders
customer.orders.empty?          # uses the cached copy of orders
```

（11）重载缓存

程序的其他部分会修改数据，那么应该怎么重载缓存呢？调用关联方法时传入 true 参数即可：

```
customer.orders                 # retrieves orders from the database
customer.orders.size            # uses the cached copy of orders
customer.orders(true).empty?    # discards the cached copy of orders
                                # and goes back to the database
```

5.9.2. 更新模式

更新模式

关联非常有用，但没什么魔法。关联对应的数据库模式需要你自己编写。不同的关联类型，要做的事也不同。对 belongs_to 关联来说，要创建外键；对 has_and_belongs_to_many 来说，要创建相应的连接数据表。

3.3.1 创建 belongs_to 关联所需的外键

声明 belongs_to 关联后，要创建相应的外键。例如，有下面这个模型：

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

这种关联需要在数据表中创建合适的外键：

```
class CreateOrders < ActiveRecord::Migration
  def change
    create_table :orders do |t|
      t.datetime :order_date
      t.string   :order_number
```

```
      t.integer    :customer_id
    end
  end
end
```

如果声明关联之前已经定义了模型，则要在迁移中使用 add_column 创建外键。

3.3.2 创建 has_and_belongs_to_many 关联所需的连接数据表

声明 has_and_belongs_to_many 关联后，必须手动创建连接数据表。除非在 :join_table 选项中指定了连接数据表的名字，否则 Active Record 会按照类名出现在字典中的顺序为数据表起名字。那么，顾客和订单模型使用的连接数据表默认名为“customers_orders”，因为在字典中，“c”在“o”前面。

模型名的顺序使用字符串的 < 操作符确定。所以，如果两个字符串的长度不同，比较最短长度时，两个字符串是相等的，但长字符串的排序比短字符串靠前。例如，你可能以为“"paper_boxes”和“papers”这两个表生成的连接表名为“papers_paper_boxes”，因为“paper_boxes”比“papers”长。其实生成的连接表名为“paper_boxes_papers”，因为在一般的编码方式中，“_”比“s”靠前。

不管名字是什么，你都要在迁移中手动创建连接数据表。例如下面的关联声明：

```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

需要在迁移中创建 assemblies_parts 数据表，而且该表无主键：

```
class CreateAssembliesPartsJoinTable < ActiveRecord::Migration
  def change
    create_table :assemblies_parts, id: false do |t|
      t.integer :assembly_id
      t.integer :part_id
    end
  end
end
```

我们把 id: false 选项传给 create_table 方法，因为这个表不对应模型。只有这样，关联才能正常建立。如果在使用 has_and_belongs_to_many 关联时遇到奇怪的表现，例如提示模型 ID 损坏，或 ID 冲突，有可能就是因为创建了主键。

5.9.3. 关联的作用域

默认情况下，关联只会查找当前模块作用域中的对象。如果在模块中定义 Active Record 模

型，知道这一点很重要。例如：

```
module MyApplication
  module Business
    class Supplier < ActiveRecord::Base
      has_one :account
    end

    class Account < ActiveRecord::Base
      belongs_to :supplier
    end
  end
end
```

上面的代码能正常运行，因为 Supplier 和 Account 在同一个作用域内。但下面这段代码就不行了，因为 Supplier 和 Account 在不同的作用域中：

```
module MyApplication
  module Business
    class Supplier < ActiveRecord::Base
      has_one :account
    end
  end
```

```
  module Billing
    class Account < ActiveRecord::Base
      belongs_to :supplier
    end
  end
end
```

要想让处在不同命名空间中的模型正常建立关联，声明关联时要指定完整的类名：

```
module MyApplication
  module Business
    class Supplier < ActiveRecord::Base
      has_one :account,
       class_name: "MyApplication::Billing::Account"
    end
  end

  module Billing
    class Account < ActiveRecord::Base
      belongs_to :supplier,
       class_name: "MyApplication::Business::Supplier"
    end
  end
end
```

5.9.4. 双向关联

（12）inverse_of

一般情况下，都要求能在关联的两端进行操作。例如，有下面的关联声明：

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

class Order < ActiveRecord::Base
  belongs_to :customer
end
```

默认情况下，Active Record 并不知道这个关联中两个模型之间的联系。可能导致同一对象的两个副本不同步：

```
c = Customer.first
o = c.orders.first
c.first_name == o.customer.first_name # => true
c.first_name = 'Manny'
c.first_name == o.customer.first_name # => false
```

之所以会发生这种情况，是因为 c 和 o.customer 在内存中是同一数据的两钟表示，修改其中一个并不会刷新另一个。Active Record 提供了 :inverse_of 选项，可以告知 Rails 两者之间的

关系：

```
class Customer < ActiveRecord::Base
  has_many :orders, inverse_of: :customer
end

class Order < ActiveRecord::Base
  belongs_to :customer, inverse_of: :orders
end
```

这么修改之后，Active Record 就只会加载一个顾客对象，避免数据的不一致性，提高程序的执行效率：

```
c = Customer.first
o = c.orders.first
c.first_name == o.customer.first_name # => true
c.first_name = 'Manny'
c.first_name == o.customer.first_name # => true
```

（13）inverse_of 有些限制

inverse_of 有些限制：

不能和 :through 选项同时使用；

不能和 :polymorphic 选项同时使用；

不能和 :as 选项同时使用；

在 belongs_to 关联中，会忽略 has_many 关联的 inverse_of 选项；

每种关联都会尝试自动找到关联的另一端，设置 :inverse_of 选项（根据关联的名字）。使用标准名字的关联都有这种功能。但是，如果在关联中设置了下面这些选项，将无法自动设置 :inverse_of

如下：

:conditions

:through

:polymorphic

:foreign_key

6. 查询

Active Record 会代你执行数据库查询，可以兼容大多数数据库（MySQL，PostgreSQL 和 SQLite 等）。不管使用哪种数据库，所用的 Active Record 方法都是一样的。

如果习惯使用 SQL 查询数据库，会发现在 Rails 中执行相同的查询有更好的方式。大多数情况下，在 Active Record 中无需直接使用 SQL。

文中的实例代码会用到下面一个或多个模型：

下面所有的模型除非有特别说明之外，都使用 id 做主键。

```
class Client < ActiveRecord::Base
  has_one :address
  has_many :orders
  has_and_belongs_to_many :roles
end

class Address < ActiveRecord::Base
  belongs_to :client
end
```

```
class Order < ActiveRecord::Base
  belongs_to :client, counter_cache: true
end

class Role < ActiveRecord::Base
  has_and_belongs_to_many :clients
end
```

6.1. 获取对象

Active Record 提供了很多查询方法，用来从数据库中获取对象。每个查询方法都接可接受参数，不用直接写 SQL 就能在数据库中执行指定的查询。

这些方法是：

find

create_with

distinct

eager_load

extending

from

group

having

includes

joins

limit

lock

none

offset

order

preload

readonly

references

reorder

reverse_order

select

uniq

where

上述所有方法都返回一个 ActiveRecord::Relation 实例。

Model.find(options) 方法执行的主要操作概括如下：

把指定的选项转换成等价的 SQL 查询语句；

执行 SQL 查询，从数据库中获取结果；

为每个查询结果实例化一个对应的模型对象；

如果有 after_find 回调，再执行 after_find 回调；

6.1.1. 获取单个对象

（1）find(primary_key)

使用 Model.find(primary_key) 方法可以获取指定主键对应的对象。例如：

```
# Find the client with primary key (id) 10.
client = Client.find(10)
# => #<Client id: 10, first_name: "Ryan">
```

和上述方法等价的 SQL 查询是：

```
SELECT * FROM clients WHERE (clients.id = 10) LIMIT 1
```

如果未找到匹配的记录，Model.find(primary_key) 会抛出 ActiveRecord::RecordNotFound 异常。

（2）take

Model.take 方法会获取一个记录，不考虑任何顺序。例如：

```
client = Client.take

# => #<Client id: 1, first_name: "Lifo">
```

和上述方法等价的 SQL 查询是：

```
SELECT * FROM clients LIMIT 1
```

如果没找到记录，Model.take 不会抛出异常，而是返回 nil。

获取的记录根据所用的数据库引擎会有所不同。

（3）take!

Model.take! 方法会获取一个记录，不考虑任何顺序。例如：

```
client = Client.take!

# => #<Client id: 1, first_name: "Lifo">
```

和上述方法等价的 SQL 查询是：

```
SELECT * FROM clients LIMIT 1
```

如果未找到匹配的记录，Model.take! 会抛出 ActiveRecord::RecordNotFound 异常。

(4) first

Model.first 获取按主键排序得到的第一个记录。例如：

```
client = Client.first

# => #<Client id: 1, first_name: "Lifo">
```

和上述方法等价的 SQL 查询是：

```
SELECT * FROM clients ORDER BY clients.id ASC LIMIT 1
```

Model.first 如果没找到匹配的记录，不会抛出异常，而是返回 nil。

(5) last

Model.last 获取按主键排序得到的最后一个记录。例如：

```
client = Client.last

# => #<Client id: 221, first_name: "Russel">
```

和上述方法等价的 SQL 查询是：

```
SELECT * FROM clients ORDER BY clients.id DESC LIMIT 1
```

Model.last 如果没找到匹配的记录，不会抛出异常，而是返回 nil。

（6）find_by

Model.find_by 获取满足条件的第一个记录。例如：

```
Client.find_by first_name: 'Lifo'
# => #<Client id: 1, first_name: "Lifo">

Client.find_by first_name: 'Jon'
# => nil
```

等价于：

```
Client.where(first_name: 'Lifo').take
```

6.1.2. 获取多个对象

（1）使用多个主键

Model.find(array_of_primary_key) 方法可接受一个由主键组成的数组，返回一个由主键对应记录组成的数组。例如：

```
# Find the clients with primary keys 1 and 10.
client = Client.find([1, 10]) # Or even Client.find(1, 10)
# => [#<Client id: 1, first_name: "Lifo">, #<Client id: 10, first_name: "Ryan">]
```

上述方法等价的 SQL 查询是：

```
SELECT * FROM clients WHERE (clients.id IN (1,10))
```

只要有一个主键的对应的记录未找到，Model.find(array_of_primary_key) 方法就会抛出 ActiveRecord::RecordNotFound 异常。

（2）take(limit)

Model.take(limit) 方法获取 limit 个记录，不考虑任何顺序：

```
Client.take(2)
# => [#<Client id: 1, first_name: "Lifo">,
      #<Client id: 2, first_name: "Raf">]
```

和上述方法等价的 SQL 查询是：

```
SELECT * FROM clients LIMIT 2
```

(3) first(limit)

Model.first(limit) 方法获取按主键排序的前 limit 个记录：

```
Client.first(2)
# => [#<Client id: 1, first_name: "Lifo">,
      #<Client id: 2, first_name: "Raf">]
```

和上述方法等价的 SQL 查询是：

```
SELECT * FROM clients ORDER BY id ASC LIMIT 2
```

(4) last(limit)

Model.last(limit) 方法获取按主键降序排列的前 limit 个记录：

```
Client.last(2)
# => [#<Client id: 10, first_name: "Ryan">,
      #<Client id: 9, first_name: "John">]
```

和上述方法等价的 SQL 查询是：

```
SELECT * FROM clients ORDER BY id DESC LIMIT 2
```

6.1.3. 批量获取多个对象

我们经常需要遍历由很多记录组成的集合，例如给大量用户发送邮件列表，或者导出数据。

我们可能会直接写出如下的代码：

```
# This is very inefficient when the users table has thousands of rows.
User.all.each do |user|
  NewsLetter.weekly_deliver(user)
end
```

但这种方法在数据表很大时就有点不现实了，因为 User.all.each 会一次读取整个数据表，一行记录创建一个模型对象，然后把整个模型对象数组存入内存。如果记录数非常多，可能会用完内存。

Rails 为了解决这种问题提供了两个方法，把记录分成几个批次，不占用过多内存。第一个方法是 find_each，获取一批记录，然后分别把每个记录传入代码块。第二个方法是 find_in_batches，获取一批记录，然后把整批记录作为数组传入代码块。

find_each 和 find_in_batches 方法的目的是分批处理无法一次载入内存的巨量记录。如果只

想遍历几千个记录，更推荐使用常规的查询方法。

（1）find_each

find_each 方法获取一批记录，然后分别把每个记录传入代码块。在下面的例子中，find_each 获取 1000 个记录，然后把每个记录传入代码块，直到所有记录都处理完为止：

```
User.find_each do |user|
  NewsLetter.weekly_deliver(user)
end
```

在 find_each 方法中可使用 find 方法的大多数选项，但不能使用 :order 和 :limit，因为这两个选项是保留给 find_each 内部使用的。

find_each 方法还可使用另外两个选项：:batch_size 和 :start。

:batch_size

:batch_size 选项指定在把各记录传入代码块之前，各批次获取的记录数量。例如，一个批次获取 5000 个记录：

```
User.find_each(batch_size: 5000) do |user|
```

```
    NewsLetter.weekly_deliver(user)
end

:start
```

默认情况下，按主键的升序方式获取记录，其中主键的类型必须是整数。如果不想用最小的 ID，可以使用 :start 选项指定批次的起始 ID。例如，前面的批量处理中断了，但保存了中断时的 ID，就可以使用这个选项继续处理。

例如，在有 5000 个记录的批次中，只向主键大于 2000 的用户发送邮件列表，可以这么做：

```
User.find_each(start: 2000, batch_size: 5000) do |user|
    NewsLetter.weekly_deliver(user)
end
```

还有一个例子是，使用多个 worker 处理同一个进程队列。如果需要每个 worker 处理 10000 个记录，就可以在每个 worker 中设置相应的 :start 选项。

（2）find_in_batches

find_in_batches 方法和 find_each 类似，都获取一批记录。二者的不同点是，find_in_batches 把整批记录作为一个数组传入代码块，而不是单独传入各记录。在下面的例子中，会把 1000 个单据一次性传入代码块，让代码块后面的程序处理剩下的单据：

```
# Give add_invoices an array of 1000 invoices at a time
Invoice.find_in_batches(include: :invoice_lines) do |invoices|
  export.add_invoices(invoices)
end
```

:include 选项可以让指定的关联和模型一同加载。

1.3.2.1 find_in_batches 方法的选项

find_in_batches 方法和 find_each 方法一样，可以使用 :batch_size 和 :start 选项，还可使用常规的 find 方法中的大多数选项，但不能使用 :order 和 :limit 选项，因为这两个选项保留给 find_in_batches 方法内部使用。

6.2. 条件查询

where 方法用来指定限制获取记录的条件，用于 SQL 语句的 WHERE 子句。条件可使用字符串、数组或 Hash 指定。

6.2.1. 纯字符串条件

如果查询时要使用条件，可以直接指定。例如 Client.where("orders_count = '2'")，获取 orders_count 字段为 2 的客户记录。

使用纯字符串指定条件可能导致 SQL 注入漏洞。例如，Client.where("first_name LIKE '%#{params[:first_name]}%'")，这里的条件就不安全。推荐使用的条件指定方式是数组，请阅读下一节。

6.2.2. 数组条件

如果数字是在别处动态生成的话应该怎么处理呢？可用下面的查询：

```
Client.where("orders_count = ?", params[:orders])
```

Active Record 会先处理第一个元素中的条件，然后使用后续元素替换第一个元素中的问号（?）。

指定多个条件的方式如下：

```
Client.where("orders_count = ? AND locked = ?", params[:orders], false)
```

在这个例子中，第一个问号会替换成 params[:orders] 的值；第二个问号会替换成 false 在 SQL 中对应的值，具体的值视所用的适配器而定。

下面这种形式

```
Client.where("orders_count = ?", params[:orders])
```

要比这种形式好

```
Client.where("orders_count = #{params[:orders]}")
```

因为前者传入的参数更安全。直接在条件字符串中指定的条件会原封不动的传给数据库。也就是说，即使用户不怀好意，条件也会转义。如果这么做，整个数据库就处在一个危险境地，只要用户发现可以接触数据库，就能做任何想做的事。所以，千万别直接在条件字符串中使用参数。

关于 SQL 注入更详细的介绍，请阅读“Ruby on Rails 安全指南”

（3）条件中的占位符

除了使用问号占位之外，在数组条件中还可使用键值对 Hash 形式的占位符：

```
Client.where("created_at >= :start_date AND created_at <= :end_date",
    {start_date: params[:start_date], end_date: params[:end_date]})
```

如果条件中有很多参数，使用这种形式可读性更高。

6.2.3. Hash 条件

Active Record 还允许使用 Hash 条件，提高条件语句的可读性。使用 Hash 条件时，传入

Hash 的键是要设定条件的字段，值是要设定的条件。

在 Hash 条件中只能指定相等、范围、子集这三种条件。

（1）相等

Client.where(locked: true)

字段的名字还可使用字符串表示：

Client.where('locked' => true)

在 belongs_to 关联中，如果条件中的值是模型对象，可用关联键表示。这种条件指定方式也可用于多态关联。

Post.where(author: author)

Author.joins(:posts).where(posts: { author: author })

条件的值不能为 Symbol。例如，不能这么指定条件：Client.where(status: :active)。

（2）范围

Client.where(created_at: (Time.now.midnight - 1.day)..Time.now.midnight)

指定这个条件后，会使用 SQL BETWEEN 子句查询昨天创建的客户：

```
SELECT * FROM clients WHERE (clients.created_at BETWEEN '2008-12-21 00:00:00' AND '2008-12-22 00:00:00')
```

这段代码演示了数组条件的简写形式。

（3）子集

如果想使用 IN 子句查询记录，可以在 Hash 条件中使用数组：

```
Client.where(orders_count: [1,3,5])
```

上述代码生成的 SQL 语句如下：

```
SELECT * FROM clients WHERE (clients.orders_count IN (1,3,5))
```

6.2.4. NOT 条件

SQL NOT 查询可用 where.not 方法构建。

```
Post.where.not(author: author)
```

也即是说，这个查询首先调用没有参数的 where 方法，然后再调用 not 方法。

6.3. 排序与分组

6.3.1. 排序

要想按照特定的顺序从数据库中获取记录，可以使用 order 方法。

例如，想按照 created_at 的升序方式获取一些记录，可以这么做：

```
Client.order(:created_at)

# OR

Client.order("created_at")
```

还可使用 ASC 或 DESC 指定排序方式：

```
Client.order(created_at: :desc)

# OR

Client.order(created_at: :asc)

# OR

Client.order("created_at DESC")

# OR

Client.order("created_at ASC")
```

或者使用多个字段排序：

```
Client.order(orders_count: :asc, created_at: :desc)

# OR

Client.order(:orders_count, created_at: :desc)

# OR

Client.order("orders_count ASC, created_at DESC")

# OR

Client.order("orders_count ASC", "created_at DESC")
```

如果想在不同的上下文中多次调用 order，可以在前一个 order 后再调用一次：

```
Client.order("orders_count ASC").order("created_at DESC")

# SELECT * FROM clients ORDER BY orders_count ASC, created_at DESC
```

6.3.2. 查询指定字段

默认情况下，Model.find 使用 SELECT * 查询所有字段。

要查询部分字段，可使用 select 方法。

例如，只查询 viewable_by 和 locked 字段：

```
Client.select("viewable_by, locked")
```

上述查询使用的 SQL 语句如下：

```
SELECT viewable_by, locked FROM clients
```

使用时要注意，因为模型对象只会使用选择的字段初始化。如果字段不能初始化模型对象，会得到以下异常：

```
ActiveModel::MissingAttributeError: missing attribute: <attribute>
```

其中 <attribute> 是所查询的字段。id 字段不会抛出 ActiveRecord::MissingAttributeError 异常，所以在关联中使用时要注意，因为关联需要 id 字段才能正常使用。

如果查询时希望指定字段的同值记录只出现一次，可以使用 distinct 方法：

```
Client.select(:name).distinct
```

上述方法生成的 SQL 语句如下：

```
SELECT DISTINCT name FROM clients
```

查询后还可以删除唯一性限制：

```
query = Client.select(:name).distinct

# => Returns unique names


query.distinct(false)

# => Returns all names, even if there are duplicates
```

6.3.3. 限量和偏移

要想在 Model.find 方法中使用 SQL LIMIT 子句，可使用 limit 和 offset 方法。

limit 方法指定获取的记录数量，offset 方法指定在返回结果之前跳过多少个记录。例如：

```
Client.limit(5)
```

上述查询最大只会返回 5 各客户对象，因为没指定偏移，多以会返回数据表中的前 5 个记录。生成的 SQL 语句如下：

```
SELECT * FROM clients LIMIT 5
```

再加上 offset 方法：

```
Client.limit(5).offset(30)
```

这时会从第 31 个记录开始，返回最多 5 个客户对象。生成的 SQL 语句如下：

```
SELECT * FROM clients LIMIT 5 OFFSET 30
```

6.3.4. 分组

要想在查询时使用 SQL GROUP BY 子句，可以使用 group 方法。

例如，如果想获取一组订单的创建日期，可以这么做：

```
Order.select("date(created_at) as ordered_date, sum(price) as total_price").group("date(created_at)")
```

上述查询会只会为相同日期下的订单创建一个 Order 对象。

生成的 SQL 语句如下：

```
SELECT date(created_at) as ordered_date, sum(price) as total_price
FROM orders
GROUP BY date(created_at)
```

6.3.5. 分组筛选

SQL 使用 HAVING 子句指定 GROUP BY 分组的条件。在 Model.find 方法中可使用 :having 选项指定 HAVING 子句。

例如：

```
Order.select("date(created_at) as ordered_date, sum(price) as total_price").
  group("date(created_at)").having("sum(price) > ?", 100)
```

生成的 SQL 如下：

```
SELECT date(created_at) as ordered_date, sum(price) as total_price
FROM orders
GROUP BY date(created_at)
HAVING sum(price) > 100
```

这个查询只会为同一天下的订单创建一个 Order 对象，而且这一天的订单总额要大于 $100。

6.3.6. 条件覆盖

（1） unscope

如果要删除某个条件可使用 unscope 方法。例如：

```
Post.where('id > 10').limit(20).order('id asc').unscope(:order)
```

生成的 SQL 语句如下：

```
SELECT * FROM posts WHERE id > 10 LIMIT 20

# Original query without `unscope`
SELECT * FROM posts WHERE id > 10 ORDER BY id asc LIMIT 20
```

unscope 还可删除 WHERE 子句中的条件。例如：

```
Post.where(id: 10, trashed: false).unscope(where: :id)
# SELECT "posts".* FROM "posts" WHERE trashed = 0
```

unscope 还可影响合并后的查询：

```
Post.order('id asc').merge(Post.unscope(:order))

# SELECT "posts".* FROM "posts"
```

(2) only

查询条件还可使用 only 方法覆盖。例如：

```
Post.where('id > 10').limit(20).order('id desc').only(:order, :where)
```

执行的 SQL 语句如下：

```
SELECT * FROM posts WHERE id > 10 ORDER BY id DESC

# Original query without `only`

SELECT "posts".* FROM "posts" WHERE (id > 10) ORDER BY id desc LIMIT 20
```

(3) reorder

reorder 方法覆盖原来的 order 条件。例如：

```
class Post < ActiveRecord::Base
  ..
  ..
```

```
    has_many :comments, -> { order('posted_at DESC') }
end

Post.find(10).comments.reorder('name')
```

执行的 SQL 语句如下：

```
SELECT * FROM posts WHERE id = 10 ORDER BY name
```

没用 reorder 方法时执行的 SQL 语句如下：

```
SELECT * FROM posts WHERE id = 10 ORDER BY posted_at DESC
```

(4) reverse_order

reverse_order 方法翻转 ORDER 子句的条件。

```
Client.where("orders_count > 10").order(:name).reverse_order
```

执行的 SQL 语句如下：

```
SELECT * FROM clients WHERE orders_count > 10 ORDER BY name DESC
```

如果查询中没有使用 ORDER 子句，reverse_order 方法会按照主键的逆序查询：

```
Client.where("orders_count > 10").reverse_order
```

执行的 SQL 语句如下：

```
SELECT * FROM clients WHERE orders_count > 10 ORDER BY clients.id DESC
```

这个方法没有参数。

（5）rewhere

rewhere 方法覆盖前面的 where 条件。例如：

```
Post.where(trashed: true).rewhere(trashed: false)
```

执行的 SQL 语句如下：

```
SELECT * FROM posts WHERE `trashed` = 0
```

如果不使用 rewhere 方法，写成：

```
Post.where(trashed: true).where(trashed: false)
```

执行的 SQL 语句如下：

```
SELECT * FROM posts WHERE `trashed` = 1 AND `trashed` = 0
```

6.4. 特殊对象

6.4.1. 空关系

none 返回一个可链接的关系，没有相应的记录。none 方法返回对象的后续条件查询，得到的还是空关系。如果想以可链接的方式响应可能无返回结果的方法或者作用域，可使用 none 方法。

```
Post.none # returns an empty Relation and fires no queries.

# The visible_posts method below is expected to return a Relation.
@posts = current_user.visible_posts.where(name: params[:name])

def visible_posts
  case role
  when 'Country Manager'
```

```
      Post.where(country: country)
    when 'Reviewer'
      Post.published
    when 'Bad User'
      Post.none # => returning [] or nil breaks the caller code in this case
    end
  end
```

6.4.2. 只读对象

Active Record 提供了 readonly 方法，禁止修改获取的对象。试图修改只读记录的操作不会成功，而且会抛出 ActiveRecord::ReadOnlyRecord 异常。

```
client = Client.readonly.first
client.visits += 1
client.save
```

因为把 client 设为了只读对象，所以上述代码调用 client.save 方法修改 visits 的值时会抛出 ActiveRecord::ReadOnlyRecord 异常。

6.4.3. 更新时锁定记录

锁定可以避免更新记录时的条件竞争，也能保证原子更新。

Active Record 提供了两种锁定机制：

乐观锁定

悲观锁定

（1）乐观锁定

乐观锁定允许多个用户编辑同一个记录，假设数据发生冲突的可能性最小。Rails 会检查读取记录后是否有其他程序在修改这个记录。如果检测到有其他程序在修改，就会抛出 ActiveRecord::StaleObjectError 异常，忽略改动。

乐观锁定字段

为了使用乐观锁定，数据表中要有一个类型为整数的 lock_version 字段。每次更新记录时，Active Record 都会增加 lock_version 字段的值。如果更新请求中的 lock_version 字段值比数据库中的 lock_version 字段值小，会抛出 ActiveRecord::StaleObjectError 异常，更新失败。例如：

```
c1 = Client.find(1)
c2 = Client.find(1)

c1.first_name = "Michael"
c1.save
```

```
c2.name = "should fail"
c2.save # Raises an ActiveRecord::StaleObjectError
```

抛出异常后，你要负责处理冲突，可以回滚操作、合并操作或者使用其他业务逻辑处理。

乐观锁定可以使用 ActiveRecord::Base.lock_optimistically = false 关闭。

要想修改 lock_version 字段的名字，可以使用 ActiveRecord::Base 提供的 locking_column 类方法：

```
class Client < ActiveRecord::Base
  self.locking_column = :lock_client_column
end
```

（2）悲观锁定

悲观锁定使用底层数据库提供的锁定机制。

使用 lock 方法构建的关系在所选记录上生成一个“互斥锁”（exclusive lock）。

使用 lock 方法构建的关系一般都放入事务中，避免死锁。

例如：

```
Item.transaction do
  i = Item.lock.first
  i.name = 'Jones'
  i.save
end
```

在 MySQL 中，上述代码生成的 SQL 如下：

```
SQL (0.2ms)   BEGIN
Item Load (0.3ms)   SELECT * FROM `items` LIMIT 1 FOR UPDATE
Item Update (0.4ms)   UPDATE `items` SET `updated_at` = '2009-02-07 18:05:56', `name` = 'Jones' WHERE `id` = 1
SQL (0.8ms)   COMMIT
```

lock 方法还可以接受 SQL 语句，使用其他锁定类型。例如，MySQL 中有一个语句是 LOCK IN SHARE MODE，会锁定记录，但还是允许其他查询读取记录。要想使用这个语句，直接传入 lock 方法即可：

```
Item.transaction do
```

```
    i = Item.lock("LOCK IN SHARE MODE").find(1)
    i.increment!(:views)
end
```

如果已经创建了模型实例，可以在事务中加上这种锁定，如下所示：

```
item = Item.first
item.with_lock do
    # This block is called within a transaction,
    # item is already locked.
    item.increment!(:views)
end
```

6.5. 连接

6.5.1. 连接数据表

Active Record 提供了一个查询方法名为 joins，用来指定 SQL JOIN 子句。joins 方法的用法有很多种。

（1）使用字符串形式的 SQL 语句

在 joins 方法中可以直接使用 JOIN 子句的 SQL：

```
Client.joins('LEFT OUTER JOIN addresses ON addresses.client_id = clients.id')
```

生成的 SQL 语句如下：

```
SELECT clients.* FROM clients LEFT OUTER JOIN addresses ON addresses.client_id = clients.id
```

（2）使用数组或 Hash 指定具名关联

这种方法只用于 INNER JOIN。

使用 joins 方法时，可以使用声明关联时使用的关联名指定 JOIN 子句。

例如，假如按照如下方式定义 Category、Post、Comment、Guest 和 Tag 模型：

```
class Category < ActiveRecord::Base
  has_many :posts
end

class Post < ActiveRecord::Base
  belongs_to :category
  has_many :comments
  has_many :tags
```

```
end

class Comment < ActiveRecord::Base
   belongs_to :post
   has_one :guest
end

class Guest < ActiveRecord::Base
   belongs_to :comment
end

class Tag < ActiveRecord::Base
   belongs_to :post
end
```

下面各种用法能都使用 INNER JOIN 子句生成正确的连接查询：

①连接单个关联

```
Category.joins(:posts)
```

生成的 SQL 语句如下：

```
SELECT categories.* FROM categories
```

```
  INNER JOIN posts ON posts.category_id = categories.id
```

用人类语言表达，上述查询的意思是，“使用文章的分类创建分类对象”。注意，分类对象可能有重复，因为多篇文章可能属于同一分类。如果不想出现重复，可使用 Category.joins(:posts).uniq 方法。

②连接多个关联

```
Post.joins(:category, :comments)
```

生成的 SQL 语句如下：

```
SELECT posts.* FROM posts
  INNER JOIN categories ON posts.category_id = categories.id
  INNER JOIN comments ON comments.post_id = posts.id
```

用人类语言表达，上述查询的意思是，“返回指定分类且至少有一个评论的所有文章”。注意，如果文章有多个评论，同个文章对象会出现多次。

③连接一层嵌套关联

```
Post.joins(comments: :guest)
```

生成的 SQL 语句如下：

```
SELECT posts.* FROM posts
  INNER JOIN comments ON comments.post_id = posts.id
  INNER JOIN guests ON guests.comment_id = comments.id
```

用人类语言表达，上述查询的意思是，“返回有一个游客发布评论的所有文章”。

④连接多层嵌套关联

```
Category.joins(posts: [{ comments: :guest }, :tags])
```

生成的 SQL 语句如下：

```
SELECT categories.* FROM categories
  INNER JOIN posts ON posts.category_id = categories.id
  INNER JOIN comments ON comments.post_id = posts.id
  INNER JOIN guests ON guests.comment_id = comments.id
  INNER JOIN tags ON tags.post_id = posts.id
```

（3）指定用于连接数据表上的条件

作用在连接数据表上的条件可以使用数组和字符串指定。[Hash 形式的条件](#hash-conditions)使用的句法有点特殊：

```
time_range = (Time.now.midnight - 1.day)..Time.now.midnight
```

```
Client.joins(:orders).where('orders.created_at' => time_range)
```

还有一种更简洁的句法是使用嵌套 Hash：

```
time_range = (Time.now.midnight - 1.day)..Time.now.midnight
Client.joins(:orders).where(orders: { created_at: time_range })
```

上述查询会获取昨天下订单的所有客户对象，再次用到了 SQL BETWEEN 语句。

6.5.2. 按需加载关联

使用 Model.find 方法获取对象的关联记录时，按需加载机制会使用尽量少的查询次数。

N + 1 查询问题

假设有如下的代码，获取 10 个客户对象，并把客户的邮编打印出来

```
clients = Client.limit(10)

clients.each do |client|
  puts client.address.postcode
end
```

上述代码初看起来很好，但问题在于查询的总次数。上述代码总共会执行 1（获取 10 个客户记录）+ 10（分别获取 10 个客户的地址）= 11 次查询。

N + 1 查询的解决办法

在 Active Record 中可以进一步指定要加载的所有关联，调用 Model.find 方法是使用 includes 方法实现。使用 includes 后，Active Record 会使用尽可能少的查询次数加载所有指定的关联。

我们可以使用按需加载机制加载客户的地址，把 Client.limit(10) 改写成：

```
clients = Client.includes(:address).limit(10)

clients.each do |client|
    puts client.address.postcode
end
```

和前面的 11 次查询不同，上述代码只会执行 2 次查询：

```
SELECT * FROM clients LIMIT 10
SELECT addresses.* FROM addresses
```

```
WHERE (addresses.client_id IN (1,2,3,4,5,6,7,8,9,10))
```

（1）按需加载多个关联

调用 Model.find 方法时，使用 includes 方法可以一次加载任意数量的关联，加载的关联可以通过数组、Hash、嵌套 Hash 指定。

13.1.1 用数组指定多个关联

```
Post.includes(:category, :comments)
```

上述代码会加载所有文章，以及和每篇文章关联的分类和评论。

13.1.2 使用 Hash 指定嵌套关联

```
Category.includes(posts: [{ comments: :guest }, :tags]).find(1)
```

上述代码会获取 ID 为 1 的分类，按需加载所有关联的文章，文章的标签和评论，以及每个评论的 guest 关联。

（2）指定用于按需加载关联上的条件

虽然 Active Record 允许使用 joins 方法指定用于按需加载关联上的条件，但是推荐的做法是使用连接数据表。

如果非要这么做，可以按照常规方式使用 where 方法。

```
Post.includes(:comments).where("comments.visible" => true)
```

上述代码生成的查询中会包含 LEFT OUTER JOIN 子句，而 joins 方法生成的查询使用的是 INNER JOIN 子句。

```
SELECT "posts"."id" AS t0_r0, ... "comments"."updated_at" AS t1_r5 FROM "posts" LEFT OUTER JOIN "comments" ON "comments"."post_id" = "posts"."id" WHERE (comments.visible = 1)
```

如果没指定 where 条件，上述代码会生成两个查询语句。

如果像上面的代码一样使用 includes，即使所有文章都没有评论，也会加载所有文章。使用 joins 方法（INNER JOIN）时，必须满足连接条件，否则不会得到任何记录。

6.5.3. 作用域

作用域把常用的查询定义成方法，在关联对象或模型上调用。在作用域中可以使用前面介绍的所有方法，例如 where、joins 和 includes。所有作用域方法都会返回一个 ActiveRecord::Relation 对象，允许继续调用其他方法（例如另一个作用域方法）。

要想定义简单的作用域，可在类中调用 scope 方法，传入执行作用域时运行的代码：

```
class Post < ActiveRecord::Base
  scope :published, -> { where(published: true) }
end
```

上述方式和直接定义类方法的作用一样，使用哪种方式只是个人喜好：

```
class Post < ActiveRecord::Base
  def self.published
    where(published: true)
  end
end
```

作用域可以链在一起调用：

```
class Post < ActiveRecord::Base
  scope :published,               -> { where(published: true) }
  scope :published_and_commented, -> { published.where("comments_count > 0") }
end
```

可以在模型类上调用 published 作用域：

```
Post.published # => [published posts]
```

也可以在包含 Post 对象的关联上调用：

```
category = Category.first
category.posts.published # => [published posts belonging to this category]
```

（1）传入参数

作用域可接受参数：

```
class Post < ActiveRecord::Base
  scope :created_before, ->(time) { where("created_at < ?", time) }
end
```

作用域的调用方法和类方法一样：

```
Post.created_before(Time.zone.now)
```

不过这就和类方法的作用一样了。

```
class Post < ActiveRecord::Base
  def self.created_before(time)
    where("created_at < ?", time)
  end
end
```

如果作用域要接受参数，推荐直接使用类方法。有参数的作用域也可在关联对象上调用：

```
category.posts.created_before(time)
```

（2）合并作用域

和 where 方法一样，作用域也可通过 AND 合并查询条件：

```
class User < ActiveRecord::Base
  scope :active, -> { where state: 'active' }
  scope :inactive, -> { where state: 'inactive' }
end

User.active.inactive
# SELECT "users".* FROM "users" WHERE "users"."state" = 'active' AND "users"."state" =
```

'inactive'

作用域还可以 where 一起使用，生成的 SQL 语句会使用 AND 连接所有条件。

```
User.active.where(state: 'finished')

# SELECT "users".* FROM "users" WHERE "users"."state" = 'active' AND "users"."state" = 'finished'
```

如果不想让最后一个 WHERE 子句获得优先权，可以使用 Relation#merge 方法。

```
User.active.merge(User.inactive)

# SELECT "users".* FROM "users" WHERE "users"."state" = 'inactive'
```

使用作用域时要注意，default_scope 会添加到作用域和 where 方法指定的条件之前。

```
class User < ActiveRecord::Base
  default_scope { where state: 'pending' }
  scope :active, -> { where state: 'active' }
  scope :inactive, -> { where state: 'inactive' }
end

User.all

# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending'
```

```
User.active

# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending' AND "users"."state" = 'active'


User.where(state: 'inactive')

# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending' AND "users"."state" = 'inactive'
```

如上所示，default_scope 中的条件添加到了 active 和 where 之前。

（3）指定默认作用域

如果某个作用域要用在模型的所有查询中，可以在模型中使用 default_scope 方法指定。

```
class Client < ActiveRecord::Base

  default_scope { where("removed_at IS NULL") }

end
```

执行查询时使用的 SQL 语句如下：

```
SELECT * FROM clients WHERE removed_at IS NULL
```

如果默认作用域中的条件比较复杂，可以使用类方法的形式定义：

```
class Client < ActiveRecord::Base
  def self.default_scope
    # Should return an ActiveRecord::Relation.
  end
end
```

（4）删除所有作用域

如果基于某些原因想删除作用域，可以使用 unscoped 方法。如果模型中定义了default_scope，而在这个作用域中不需要使用，就可以使用 unscoped 方法。

```
Client.unscoped.load
```

unscoped 方法会删除所有作用域，在数据表中执行常规查询。

注意，不能在作用域后链式调用 unscoped，这时可以使用代码块形式的 unscoped 方法：

```
Client.unscoped {
  Client.created_before(Time.zone.now)
}
```

6.5.4. 其他查询

（1）动态查询方法

ctive Record 为数据表中的每个字段都提供了一个查询方法。例如，在 Client 模型中有个 first_name 字段，那么 Active Record 就会生成 find_by_first_name 方法。如果在 Client 模型中有个 locked 字段，就有一个 find_by_locked 方法。

在这些动态生成的查询方法后，可以加上感叹号（！），例如 Client.find_by_name!("Ryan")。此时，如果找不到记录就会抛出 ActiveRecord::RecordNotFound 异常。

如果想同时查询 first_name 和 locked 字段，可以用 and 把两个字段连接起来，获得所需的查询方法，例如 Client.find_by_first_name_and_locked("Ryan", true)。

（2）查找或构建新对象

某些动态查询方法在 Rails 4.0 中已经启用，会在 Rails 4.1 中删除。推荐的做法是使用 Active Record 作用域。废弃的方法可以在这个 gem 中查看：https://github.com/rails/activerecord-deprecated_finders。

我们经常需要在查询不到记录时创建一个新记录。这种需求可以使用 find_or_create_by 或 find_or_create_by! 方法实现。

①find_or_create_by

find_or_create_by 方法首先检查指定属性对应的记录是否存在，如果不存在就调用 create 方法。我们来看一个例子。

假设你想查找一个名为“Andy”的客户，如果这个客户不存在就新建。这个需求可以使用下面的代码完成：

```
Client.find_or_create_by(first_name: 'Andy')

# => #<Client id: 1, first_name: "Andy", orders_count: 0, locked: true, created_at: "2011-08-30 06:09:27", updated_at: "2011-08-30 06:09:27">
```

上述方法生成的 SQL 语句如下：

```
SELECT * FROM clients WHERE (clients.first_name = 'Andy') LIMIT 1

BEGIN

INSERT INTO clients (created_at, first_name, locked, orders_count, updated_at) VALUES ('2011-08-30 05:22:57', 'Andy', 1, NULL, '2011-08-30 05:22:57')

COMMIT
```

find_or_create_by 方法返回现有的记录或者新建的记录。在上面的例子中，名为“Andy”的客户不存在，所以会新建一个记录，然后将其返回。

新纪录可能没有存入数据库，这取决于是否能通过数据验证（就像 create 方法一样）。

假设创建新记录时，要把 locked 属性设为 false，但不想在查询中设置。例如，我们要查询一个名为“Andy”的客户，如果这个客户不存在就新建一个，而且 locked 属性为 false。

这种需求有两种实现方法。第一种，使用 create_with 方法：

```
Client.create_with(locked: false).find_or_create_by(first_name: 'Andy')
```

第二种，使用代码块：

```
Client.find_or_create_by(first_name: 'Andy') do |c|
  c.locked = false
end
```

代码块中的代码只会在创建客户之后执行。再次运行这段代码时，会忽略代码块中的代码。

②find_or_create_by!

还可使用 find_or_create_by! 方法，如果新纪录不合法，会抛出异常。本文不涉及数据验证，假设已经在 Client 模型中定义了下面的验证：

validates :orders_count, presence: true

如果创建新 Client 对象时没有指定 orders_count 属性的值，这个对象就是不合法的，会抛出以下异常：

```
Client.find_or_create_by!(first_name: 'Andy')
# => ActiveRecord::RecordInvalid: Validation failed: Orders count can't be blank
```

③find_or_initialize_by

find_or_initialize_by 方法和 find_or_create_by 的作用差不多，但不调用 create 方法，而是 new 方法。也就是说新建的模型实例在内存中，没有存入数据库。继续使用前面的例子，现在我们要查询的客户名为“Nick”：

```
nick = Client.find_or_initialize_by(first_name: 'Nick')
# => <Client id: nil, first_name: "Nick", orders_count: 0, locked: true, created_at: "2011-08-30 06:09:27", updated_at: "2011-08-30 06:09:27">

nick.persisted?
# => false

nick.new_record?
# => true
```

因为对象不会存入数据库，上述代码生成的 SQL 语句如下：

```
SELECT * FROM clients WHERE (clients.first_name = 'Nick') LIMIT 1
```

如果想把对象存入数据库，调用 save 方法即可：

```
nick.save
# => true
```

（3）使用 SQL 语句查询

如果想使用 SQL 语句查询数据表中的记录，可以使用 find_by_sql 方法。就算只找到一个记录，find_by_sql 方法也会返回一个由记录组成的数组。例如，可以运行下面的查询：

```
Client.find_by_sql("SELECT * FROM clients
   INNER JOIN orders ON clients.id = orders.client_id
   ORDER BY clients.created_at desc")
```

find_by_sql 方法提供了一种定制查询的简单方式。

①select_all

find_by_sql 方法有一个近亲，名为 connection#select_all。和 find_by_sql 一样，select_all 方

法会使用 SQL 语句查询数据库，获取记录，但不会初始化对象。select_all 返回的结果是一个由 Hash 组成的数组，每个 Hash 表示一个记录。

```
Client.connection.select_all("SELECT * FROM clients WHERE id = '1'")
```

②pluck

pluck 方法可以在模型对应的数据表中查询一个或多个字段，其参数是一组字段名，返回结果是由各字段的值组成的数组。

```
Client.where(active: true).pluck(:id)
# SELECT id FROM clients WHERE active = 1
# => [1, 2, 3]

Client.distinct.pluck(:role)
# SELECT DISTINCT role FROM clients
# => ['admin', 'member', 'guest']

Client.pluck(:id, :name)
# SELECT clients.id, clients.name FROM clients
# => [[1, 'David'], [2, 'Jeremy'], [3, 'Jose']]
```

如下的代码：

```
Client.select(:id).map { |c| c.id }
# or
Client.select(:id).map(&:id)
# or
Client.select(:id, :name).map { |c| [c.id, c.name] }
```

可用 pluck 方法实现：

```
Client.pluck(:id)
# or
Client.pluck(:id, :name)
```

和 select 方法不一样，pluck 直接把查询结果转换成 Ruby 数组，不生成 Active Record 对象，可以提升大型查询或常用查询的执行效率。但 pluck 方法不会使用重新定义的属性方法处理查询结果。例如：

```
class Client < ActiveRecord::Base
  def name
    "I am #{super}"
  end
end
```

```
Client.select(:name).map &:name

# => ["I am David", "I am Jeremy", "I am Jose"]


Client.pluck(:name)

# => ["David", "Jeremy", "Jose"]
```

而且，与 select 和其他 Relation 作用域不同的是，pluck 方法会直接执行查询，因此后面不能和其他作用域链在一起，但是可以链接到已经执行的作用域之后：

```
Client.pluck(:name).limit(1)

# => NoMethodError: undefined method `limit' for #<Array:0x007ff34d3ad6d8>


Client.limit(1).pluck(:name)

# => ["David"]
```

③ids

ids 方法可以直接获取数据表的主键。

```
Person.ids

# SELECT id FROM people


class Person < ActiveRecord::Base
```

```
    self.primary_key = "person_id"
end

Person.ids
# SELECT person_id FROM people
```

（4）检查对象是否存在

如果只想检查对象是否存在，可以使用 exists? 方法。这个方法使用的数据库查询和 find 方法一样，但不会返回对象或对象集合，而是返回 true 或 false。

```
Client.exists?(1)
```

exists? 方法可以接受多个值，但只要其中一个记录存在，就会返回 true。

```
Client.exists?(id: [1,2,3])
# or
Client.exists?(name: ['John', 'Sergei'])
```

在模型或关系上调用 exists? 方法时，可以不指定任何参数。

```
Client.where(first_name: 'Ryan').exists?
```

在上述代码中，只要有一个客户的 first_name 字段值为 'Ryan'，就会返回 true，否则返回 false。

```
Client.exists?
```

在上述代码中，如果 clients 表是空的，会返回 false，否则返回 true。

在模型或关系中检查存在性时还可使用 any? 和 many? 方法。

```
# via a model
Post.any?
Post.many?

# via a named scope
Post.recent.any?
Post.recent.many?

# via a relation
Post.where(published: true).any?
Post.where(published: true).many?

# via an association
```

```
Post.first.categories.any?

Post.first.categories.many?
```

6.6. 计算

这里先以 count 方法为例，所有的选项都可在后面各方法中使用。

所有计算型方法都可直接在模型上调用：

```
Client.count

# SELECT count(*) AS count_all FROM clients
```

或者在关系上调用：

```
Client.where(first_name: 'Ryan').count

# SELECT count(*) AS count_all FROM clients WHERE (first_name = 'Ryan')
```

执行复杂计算时还可使用各种查询方法：

```
Client.includes("orders").where(first_name: 'Ryan', orders: { status: 'received' }).count
```

上述代码执行的 SQL 语句如下：

```
SELECT count(DISTINCT clients.id) AS count_all FROM clients
  LEFT OUTER JOIN orders ON orders.client_id = client.id WHERE
  (clients.first_name = 'Ryan' AND orders.status = 'received')
```

6.6.1. 计数

如果想知道模型对应的数据表中有多少条记录，可以使用 Client.count 方法。如果想更精确的计算设定了 age 字段的记录数，可以使用 Client.count(:age)。

count 方法可用的选项如前所述。

6.6.2. 平均值

如果想查看某个字段的平均值，可以使用 average 方法。用法如下：

```
Client.average("orders_count")
```

这个方法会返回指定字段的平均值，得到的有可能是浮点数，例如 3.14159265。

average 方法可用的选项如前所述。

6.6.3. 最小值

如果想查看某个字段的最小值，可以使用 minimum 方法。用法如下：

```
Client.minimum("age")
```

minimum 方法可用的选项如前所述。

6.6.4. 最大值

如果想查看某个字段的最大值，可以使用 maximum 方法。用法如下：

```
Client.maximum("age")
```

maximum 方法可用的选项如前所述。

6.6.5. 求和

如果想查看所有记录中某个字段的总值，可以使用 sum 方法。用法如下：

```
Client.sum("orders_count")
```

sum 方法可用的选项如前所述。

6.6.6. 执行 EXPLAIN 命令

可以在关系执行的查询中执行 EXPLAIN 命令。例如：

```
User.where(id: 1).joins(:posts).explain
```

在 MySQL 中得到的输出如下：

EXPLAIN for: SELECT `users`.* FROM `users` INNER JOIN `posts` ON `posts`.`user_id` = `users`.`id` WHERE `users`.`id` = 1

```
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref   | rows | Extra       |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------------+
|  1 | SIMPLE      | users | const | PRIMARY       | PRIMARY | 4       | const |    1 |             |
|  1 | SIMPLE      | posts | ALL   | NULL          | NULL    | NULL    | NULL  |    1 | Using where |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------------+
```

2 rows in set (0.00 sec)

Active Record 会按照所用数据库 shell 的方式输出结果。所以，相同的查询在 PostgreSQL 中得到的输出如下：

EXPLAIN for: SELECT "users".* FROM "users" INNER JOIN "posts" ON "posts"."user_id" = "users"."id" WHERE "users"."id" = 1

```
                              QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..37.24 rows=8 width=0)
   Join Filter: (posts.user_id = users.id)
   ->  Index Scan using users_pkey on users  (cost=0.00..8.27 rows=1 width=4)
```

```
        Index Cond: (id = 1)
  ->  Seq Scan on posts  (cost=0.00..28.88 rows=8 width=4)
        Filter: (posts.user_id = 1)
(6 rows)
```

按需加载会触发多次查询，而且有些查询要用到之前查询的结果。鉴于此，explain 方法会真正执行查询，然后询问查询计划。例如：

```
User.where(id: 1).includes(:posts).explain
```

在 MySQL 中得到的输出如下：

```
EXPLAIN for: SELECT `users`.* FROM `users`  WHERE `users`.`id` = 1
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref   | rows | Extra |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+
|  1 | SIMPLE      | users | const | PRIMARY       | PRIMARY | 4       | const |    1 |       |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+
1 row in set (0.00 sec)

EXPLAIN for: SELECT `posts`.* FROM `posts`  WHERE `posts`.`user_id` IN (1)
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra       |
```

```
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
|  1 | SIMPLE      | posts | ALL  | NULL          | NULL | NULL    | NULL |    1 | Using where |
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
1 row in set (0.00 sec)
```

7. 数据验证

7.1. 简介

7.1.1. 示例

下面演示一个非常简单的数据验证：

```
class Person < ActiveRecord::Base
  validates :name, presence: true
end

Person.create(name: "John Doe").valid? # => true
Person.create(name: nil).valid? # => false
```

如上所示，如果 Person 的 name 属性值为空，验证就会将其视为不合法对象。创建的第二个 Person 对象不会存入数据库。

7.1.2. why

1.1.1.1 弥补数据库约束

数据库约束和“存储过程”无法兼容多种数据库，而且测试和维护较为困难。不过，如果其他程序也要使用这个数据库，最好在数据库层做些约束。数据库层的某些验证（例如在使用量很高的数据表中做唯一性验证）通过其他方式实现起来有点困难。

1.1.1.2 前台验证很容易跳过

客户端验证很有用，但单独使用时可靠性不高。如果使用 JavaScript 实现，用户在浏览器中禁用 JavaScript 后很容易跳过验证。客户端验证和其他验证方式结合使用，可以为用户提供实时反馈。

7.2. 验证基本环节

7.2.1. 验证相关方法

新建并保存记录会在数据库中执行 SQL INSERT 操作。更新现有的记录会在数据库上执行 SQL UPDATE 操作。一般情况下，数据验证发生在这些 SQL 操作执行之前。如果验证失败，对象会被标记为不合法，Active Record 不会向数据库发送 INSERT 或 UPDATE 指令。这样就可以避免把不合法的数据存入数据库。你可以选择在对象创建、保存或更新时执行哪些数据验证。

下列方法会做数据验证，如果验证失败就不会把对象存入数据库：

create

create!

save

save!

update

update!

爆炸方法（例如 save!）会在验证失败后抛出异常。验证失败后，非爆炸方法不会抛出异常，save 和 update 返回 false，create 返回对象本身。

7.2.2. 跳过验证相关方法

下列方法会跳过验证，不管验证是否通过都会把对象存入数据库，使用时要特别留意。

decrement!

decrement_counter

increment!

increment_counter

toggle!

touch

update_all

update_attribute

update_column

update_columns

update_counters

注意，使用 save 时如果传入 validate: false，也会跳过验证。使用时要特别留意。

save(validate: false)

7.2.3. valid? 和 invalid?

Rails 使用 valid? 方法检查对象是否合法。valid? 方法会触发数据验证，如果对象上没有错误，就返回 true，否则返回 false。前面我们已经用过了：

```
class Person < ActiveRecord::Base
  validates :name, presence: true
end

Person.create(name: "John Doe").valid? # => true
Person.create(name: nil).valid? # => false
```

Active Record 验证结束后，所有发现的错误都可以通过实例方法 errors.messages 获取，该方法返回一个错误集合。如果数据验证后，这个集合为空，则说明对象是合法的。

注意，使用 new 方法初始化对象时，即使不合法也不会报错，因为这时还没做数据验证。

```
class Person < ActiveRecord::Base
  validates :name, presence: true
end

>> p = Person.new
```

```
# => #<Person id: nil, name: nil>

>> p.errors.messages

# => {}


>> p.valid?

# => false

>> p.errors.messages

# => {name:["can't be blank"]}


>> p = Person.create

# => #<Person id: nil, name: nil>

>> p.errors.messages

# => {name:["can't be blank"]}


>> p.save

# => false


>> p.save!

# => ActiveRecord::RecordInvalid: Validation failed: Name can't be blank


>> Person.create!

# => ActiveRecord::RecordInvalid: Validation failed: Name can't be blank
```

invalid? 是 valid? 的逆测试，会触发数据验证，如果找到错误就返回 true，否则返回 false。

7. 2. 4. errors[]

要检查对象的某个属性是否合法，可以使用 errors[:attribute]。errors[:attribute] 中包含 :attribute 的所有错误。如果某个属性没有错误，就会返回空数组。

这个方法只在数据验证之后才能使用，因为它只是用来收集错误信息的，并不会触发验证。而且，和前面介绍的 ActiveRecord::Base#invalid? 方法不一样，因为 errors[:attribute] 不会验证整个对象，只检查对象的某个属性是否出错。

```
class Person < ActiveRecord::Base
  validates :name, presence: true
end

>> Person.new.errors[:name].any? # => false
>> Person.create.errors[:name].any? # => true
```

我们会在“处理验证错误”一节详细介绍验证错误。现在，我们来看一下 Rails 默认提供的数据验证帮助方法。

1.2 2 验证类型

7. 2. 5. acceptance

这个方法检查表单提交时，用户界面中的复选框是否被选中。

这个功能一般用来要求用户接受程序的服务条款，阅读一些文字，等等。这种验证只针对网页程序，不会存入数据库（如果没有对应的字段，该方法会创建一个虚拟属性）。

```
class Person < ActiveRecord::Base
  validates :terms_of_service, acceptance: true
end
```

这个帮助方法的默认错误消息是“must be accepted”。

这个方法可以指定 :accept 选项，决定可接受什么值。默认为“1”，很容易修改：

```
class Person < ActiveRecord::Base
  validates :terms_of_service, acceptance: { accept: 'yes' }
end
```

7. 2. 6. validates_associated

如果模型和其他模型有关联，也要验证关联的模型对象，可以使用这个方法。保存对象时，会在相关联的每个对象上调用 valid? 方法。

```
class Library < ActiveRecord::Base
  has_many :books
  validates_associated :books
end
```

这个帮助方法可用于所有关联类型。

不要在关联的两端都使用 validates_associated，这样会生成一个循环。

validates_associated 的默认错误消息是“is invalid”。注意，相关联的每个对象都有各自的 errors 集合，错误消息不会都集中在调用该方法的模型对象上。

7.2.7. confirmation

如果要检查两个文本字段的值是否完全相同，可以使用这个帮助方法。例如，确认 Email 地址或密码。这个帮助方法会创建一个虚拟属性，其名字为要验证的属性名后加 _confirmation。

```
class Person < ActiveRecord::Base
  validates :email, confirmation: true
end
```

在视图中可以这么写：

```
<%= text_field :person, :email %>
<%= text_field :person, :email_confirmation %>
```

只有 email_confirmation 的值不是 nil 时才会做这个验证。所以要为确认属性加上存在性验证（后文会介绍 presence 验证）。

```
class Person < ActiveRecord::Base
  validates :email, confirmation: true
  validates :email_confirmation, presence: true
end
```

这个帮助方法的默认错误消息是“doesn't match confirmation”。

7.2.8. exclusion

这个帮助方法检查属性的值是否不在指定的集合中。集合可以是任何一种可枚举的对象。

```
class Account < ActiveRecord::Base
  validates :subdomain, exclusion: { in: %w(www us ca jp),
    message: "%{value} is reserved." }
end
```

exclusion 方法要指定 :in 选项，设置哪些值不能作为属性的值。:in 选项有个别名 :with，作用相同。上面的例子设置了 :message 选项，演示如何获取属性的值。

默认的错误消息是“is reserved”。

7.2.9. format

这个帮助方法检查属性的值是否匹配 :with 选项指定的正则表达式。

```
class Product < ActiveRecord::Base
  validates :legacy_code, format: { with: /\A[a-zA-Z]+\z/,
    message: "only allows letters" }
end
```

默认的错误消息是“is invalid”。

7.2.10. inclusion

这个帮助方法检查属性的值是否在指定的集合中。集合可以是任何一种可枚举的对象。

```
class Coffee < ActiveRecord::Base
  validates :size, inclusion: { in: %w(small medium large),
    message: "%{value} is not a valid size" }
end
```

inclusion 方法要指定 :in 选项，设置可接受哪些值。:in 选项有个别名 :within，作用相同。上面的例子设置了 :message 选项，演示如何获取属性的值。

该方法的默认错误消息是“is not included in the list”。

7.2.11. length

这个帮助方法验证属性值的长度，有多个选项，可以使用不同的方法指定长度限制：

```
class Person < ActiveRecord::Base
  validates :name, length: { minimum: 2 }
  validates :bio, length: { maximum: 500 }
  validates :password, length: { in: 6..20 }
  validates :registration_number, length: { is: 6 }
end
```

可用的长度限制选项有：

:minimum：属性的值不能比指定的长度短；

:maximum：属性的值不能比指定的长度长；

:in（或 :within）：属性值的长度在指定值之间。该选项的值必须是一个范围；

:is：属性值的长度必须等于指定值；

默认的错误消息根据长度验证类型而有所不同，还是可以 :message 定制。定制消息时，可以使用 :wrong_length、:too_long 和 :too_short 选项，%{count} 表示长度限制的值。

```
class Person < ActiveRecord::Base
```

```
    validates :bio, length: { maximum: 1000,
      too_long: "%{count} characters is the maximum allowed" }
  end
```

这个帮助方法默认统计字符数，但可以使用 :tokenizer 选项设置其他的统计方式：

```
  class Essay < ActiveRecord::Base
    validates :content, length: {
      minimum: 300,
      maximum: 400,
      tokenizer: lambda { |str| str.scan(/\w+/) },
      too_short: "must have at least %{count} words",
      too_long: "must have at most %{count} words"
    }
  end
```

注意，默认的错误消息使用复数形式（例如，“is too short (minimum is %{count} characters”），所以如果长度限制是 minimum: 1，就要提供一个定制的消息，或者使用 presence: true 代替。:in 或 :within 的值比 1 小时，都要提供一个定制的消息，或者在 length 之前，调用 presence 方法。

7. 2. 12. numericality

这个帮助方法检查属性的值是否值包含数字。默认情况下，匹配的值是可选的正负符号后加整数或浮点数。如果只接受整数，可以把 :only_integer 选项设为 true。

如果 :only_integer 为 true，则使用下面的正则表达式验证属性的值。

/\A[+-]?\d+\Z/

否则，会尝试使用 Float 把值转换成数字。

注意上面的正则表达式允许最后出现换行符。

```
class Player < ActiveRecord::Base
  validates :points, numericality: true
  validates :games_played, numericality: { only_integer: true }
end
```

除了 :only_integer 之外，这个方法还可指定以下选项，限制可接受的值：

:greater_than ：属性值必须比指定的值大。该选项默认的错误消息是“must be greater than %{count}”；

:greater_than_or_equal_to：属性值必须大于或等于指定的值。该选项默认的错误消息是“must be greater than or equal to %{count}”；

:equal_to：属性值必须等于指定的值。该选项默认的错误消息是“must be equal to %{count}”；

:less_than :属性值必须比指定的值小。该选项默认的错误消息是“must be less than %{count}”；

:less_than_or_equal_to：属性值必须小于或等于指定的值。该选项默认的错误消息是“must be less than or equal to %{count}”；

:odd：如果设为 true，属性值必须是奇数。该选项默认的错误消息是“must be odd”；

:even：如果设为 true，属性值必须是偶数。该选项默认的错误消息是“must be even”；

默认的错误消息是“is not a number”。

7.2.13. presence

这个帮助方法检查指定的属性是否为非空值，调用 blank? 方法检查只是否为 nil 或空字符串，即空字符串或只包含空白的字符串。

```
class Person < ActiveRecord::Base
  validates :name, :login, :email, presence: true
end
```

如果要确保关联对象存在，需要测试关联的对象本身是够存在，而不是用来映射关联的外键。

```
class LineItem < ActiveRecord::Base
  belongs_to :order
  validates :order, presence: true
end
```

为了能验证关联的对象是否存在，要在关联中指定 :inverse_of 选项。

```
class Order < ActiveRecord::Base
  has_many :line_items, inverse_of: :order
end
```

如果验证 has_one 或 has_many 关联的对象是否存在，会在关联的对象上调用 blank? 和 marked_for_destruction? 方法。

因为 false.blank? 的返回值是 true，所以如果要验证布尔值字段是否存在要使用 validates :field_name, inclusion: { in: [true, false] }。

默认的错误消息是“can't be blank”。

7.3. 验证选项

7.3.1. :allow_nil

指定 :allow_nil 选项后，如果要验证的值为 nil 就会跳过验证。

```
class Coffee < ActiveRecord::Base
  validates :size, inclusion: { in: %w(small medium large),
    message: "%{value} is not a valid size" }, allow_nil: true
end
```

7.3.2. :allow_blank

:allow_blank 选项和 :allow_nil 选项类似。如果要验证的值为空（调用 blank? 方法，例如 nil 或空字符串），就会跳过验证。

```
class Topic < ActiveRecord::Base
  validates :title, length: { is: 5 }, allow_blank: true
end

Topic.create(title: "").valid?  # => true
Topic.create(title: nil).valid? # => true
```

7.3.3. :message

前面已经介绍过，如果验证失败，会把 :message 选项指定的字符串添加到 errors 集合中。如果没指定这个选项，Active Record 会使用各种验证帮助方法的默认错误消息。

7.3.4. :on

:on 选项指定什么时候做验证。所有内建的验证帮助方法默认都在保存时（新建记录或更新记录）做验证。如果想修改，可以使用 on: :create，指定只在创建记录时做验证；或者使用 on: :update，指定只在更新记录时做验证。

```
class Person < ActiveRecord::Base
```

```
    # it will be possible to update email with a duplicated value
    validates :email, uniqueness: true, on: :create

    # it will be possible to create the record with a non-numerical age
    validates :age, numericality: true, on: :update

    # the default (validates on both create and update)
    validates :name, presence: true
  end
```

7.4. 处理验证错误

7.4.1. 流程

除了前面介绍的 valid? 和 invalid? 方法之外，Rails 还提供了很多方法用来处理 errors 集合，以及查询对象的合法性。

下面介绍其中一些常用的方法。所有可用的方法请查阅 ActiveModel::Errors 的文档。

7.4.2. errors

ActiveModel::Errors 的实例包含所有的错误。其键是每个属性的名字，值是一个数组，包含错误消息字符串。

```
class Person < ActiveRecord::Base
  validates :name, presence: true, length: { minimum: 3 }
```

```
end

person = Person.new

person.valid? # => false

person.errors.messages

 # => {:name=>["can't be blank", "is too short (minimum is 3 characters)"]}

person = Person.new(name: "John Doe")

person.valid? # => true

person.errors.messages # => {}
```

1.2.1 errors[]

errors[] 用来获取某个属性上的错误消息，返回结果是一个由该属性所有错误消息字符串组成的数组，每个字符串表示一个错误消息。如果字段上没有错误，则返回空数组。

```
class Person < ActiveRecord::Base

   validates :name, presence: true, length: { minimum: 3 }

end

person = Person.new(name: "John Doe")

person.valid? # => true

person.errors[:name] # => []

person = Person.new(name: "JD")
```

```
person.valid? # => false

person.errors[:name] # => ["is too short (minimum is 3 characters)"]


person = Person.new

person.valid? # => false

person.errors[:name]

 # => ["can't be blank", "is too short (minimum is 3 characters)"]
```

1.2.2 errors.size

size 方法返回对象上错误消息的总数。

```
class Person < ActiveRecord::Base

   validates :name, presence: true, length: { minimum: 3 }

end


person = Person.new

person.valid? # => false

person.errors.size # => 2


person = Person.new(name: "Andrea", email: "andrea@example.com")

person.valid? # => true

person.errors.size # => 0
```

1.2.3 errors.clear

如果想清除 errors 集合中的所有错误消息，可以使用 clear 方法。当然了，在不合法的对象上调用 errors.clear 方法后，这个对象还是不合法的，虽然 errors 集合为空了，但下次调用 valid? 方法，或调用其他把对象存入数据库的方法时， 会再次进行验证。如果任何一个验证失败了，errors 集合中就再次出现值了。

```
class Person < ActiveRecord::Base
  validates :name, presence: true, length: { minimum: 3 }
end

person = Person.new
person.valid? # => false
person.errors[:name]
 # => ["can't be blank", "is too short (minimum is 3 characters)"]

person.errors.clear
person.errors.empty? # => true

p.save # => false

p.errors[:name]
# => ["can't be blank", "is too short (minimum is 3 characters)"]
```