Rails 和 Django 的深度技术对比

我想以一个免责声明来开始下面的内容。我使用Django开发网站已经有三年了，众所周知，我喜欢Django。我已经写了一个开源的应用程序（app），并且我已经将补丁发送到了Django.然而，我以尽可能以公正的态度写了这篇文章，这篇文章对这个框架有称赞，也有批评。

6个月以前我在大学用Ruby on Rails做了一个项目而且一直做到现在。我做地第一件事就是仔细地学习了这两个框架并对它们进行了比较，但是我记得当时我很泄气的。当我寻找这些问题（比如说：”对于这两者来说，数据库的迁移是如何操作的？“、”它们的语法有什么区别？“、”用户认证是如何做的“）的答案时，我站在一个较高的角度比较了两者，发现它们大部分是非常肤浅的。下面的评论将会回答这些问题并且比较每个web框架是如何操作模型、控制器、视图、测试的。

简要介绍

两个框架都是为了更快的开发web应用程序和更好的组织代码这两个需求应运而生的. 它们都遵循 MVC 原则, 这意味着域（模型层）的建模，应用程序的展现（视图层）以及用户交互（控制层）三者之间都是相互分开的. 附带说明一下， Django 实际上只考虑了让框架做控制层的工作，因此 Django 自己声称它是一个模型-模板-视图（model-template-view）框架. Django 的模板可以被理解为视图，而视图则可以看做是MVC典型场景中的控制层. 本文中我将都是用标准的MVC术语.

Ruby on Rails

Ruby on Rails (RoR) 是一个用 Ruby 写就的web开发框架，并且Ruby“famous”也经常被认为是归功于它的. Rails 着重强调了约定大于配置和测试这两个方面. Rails 的约定大于配置（CoC）意味着几乎没有配置文件, 只有实现约定好的目录结构和命名规则. 它的每一处都藏着很多小魔法: 自动引入, 自动向视图层传递控制器实体，一大堆诸如模板名称这样的东西都是框架能自动推断出来的. 这也就意味着开发者只需要去指定应用程序中没有约定的部分, 结果就是干净简短的代码了.

Django

Django 是一个用 Python 写成的web开发框架，并以吉他手 Django Reinhardt 命名. Django 出现的动机在于 "产品部密集的最后期限和开发了它的有经验的Web开发者他们的严格要求". Django 遵循的规则是 明确的说明要比深晦的隐喻要好 (这是一条核心的 Python 原则), 结果就是即使对框架不熟悉的人，代码都是非常具有可读性的. 项目中的Django是围绕app组织的. 每一个app都有其自己的模型，控制器，视图以及测试设置，从而像一个小项目一样. Django 项目基本上就是一个小app的集合, 每一个app都负责一个特定的子系统.

模型（Model）

让我们先从看看每个框架怎样处理MVC原则开始. 模型描述了数据看起来是什么样子的，并且还包含了业务逻辑.

创建模型

Rails 通过在终端中运行一个命令来创建模型.

```
rails generate model Product name:string quantity_in_stock:integer
                             category:references
```

该命令会自动生成一次迁移和一个空的模型文件，看起来像下面这样:

```
class Product < ActiveRecord::Base

end
```

由于我有Django的技术背景, 令我很生气的一个事实就是我不能只通过模型文件就了解到一个模型有哪些字段. 我了解到Rails基本上只是将模型文件用于业务逻辑，而把模型长什么样存到了一个叫做schemas.rb 的文件中. 这个文件会在每次有迁移运行时被自动更新. 如果我们看看该文件，我们可以看到我们的 Product 模型长什么样子.

```
create_table "products", :force => true do |t|
  t.string   "name",
  t.integer  "quantity_in_stock",
  t.integer  "category_id",
  t.datetime "created_at", :null => false
  t.datetime "updated_at", :null => false
end
```

在这个模型中你可以看到两个额外的属性. created_at 和 updated_at 是两个会被自动添加到Rails中的每个模型中的属性.

在 Django 中，模型被定义到了一个叫做models.py的文件中. 同样的 Product 模型看起来也许会像下面这样

```
class Product(models.Model):
    name = models.CharField()
    quantity_in_stock = models.IntegerField()
    category = models.ForeignKey('Category')
    created_at = models.DateTimeField(auto_now_add=True) # set when it's created
    updated_at = models.DateTimeField(auto_now=True) # set every time it's updated
```

注意，我们必须在Django中明确的（也就是自己手动的）添加 created_at 和 updated_at 属性. 我们也要通过auto_now_add 和 auto_now 两个参数告诉 Django 这些属性的行为是如何定义的.

模型（Model）字段默认值和外键

Rails将默认允许字段为空。你可以在上面的例子中看到，我们创建的三个字段都被允许为空。引用字段类别也将既不创建索引，也不创建一个外键约束。这意味着引用完整性是无法保证的。Django的字段默认值是完全相反的。没有字段是被允许为空的，除非明确地设置。Django的ForeignKey将自动创建一个外键约束和索引。尽管Rails这里的制定可能是出于性能的担忧，但我会站在Django这边，我相信这个制定可以避免（意外）糟糕的设计和意想不到的情况。举例来说，在我们的项目中以前有一个学生没有意识到他创建的所有字段都被允许空（null）作为默认值。一段时间后，我们发现我们的一些表包含的数据是毫无意义的，如一个使用null作为标题的轮询。由于Rails不添加外键，在我们的例子中，我们可以删除一个持续引用其他产品的类别，这些产品将会有无效引用。一种选择是使用一个第三方应用程序，增加对自动创建外键的支持。

迁移（Migrations）

迁移允许数据库的模式(schema)在创建之后可以再次更改(实际上，在Rails中所有的内容都使用迁移，即使是创建)。我不得不敬佩Rails长期以来支持这个特性。通过使用Rails的生成器(generator)即可完成工作。

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

这会向Product模型(model)添加一个名为part_number的新字段(field)。

然而Django只能通过名为South的第三方库来支持迁移。我感觉South的方式更加简洁和实用。上面对应的迁移工作可以直接编辑Product模型的定义并添加新的字段

```
class Product(models.Model):
    ... # 旧字段
    part_number = models.CharField()
```

然后调用

```
$ python manage.py schemamigration products --auto
```

South会自动识别到一个新增字段添加到Product模型并创建迁移文件。随后会调用下面命令完成同步(synced)

```
$ python manage.py migrate products
```

Django最终在它的最新版本(1.7) 将South整合进来并支持迁移。

执行查询

感谢对象关系映射(object-relation mapping),你不需要在任何框架中写一行SQL语句。感谢Ruby表达式,你能够很优雅的写出范围搜索查询(range query)。.

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

这会查询到昨天创建的 `Clients` 。Python不支持像1.day这种极其可读和简洁的语法，也不支持 `..` 范围操作符。但是，有时在写Rails时我感觉像是又在写预声明(prepared statement)一样。比如为了选择所有的某一字段大于某个值的行，你不得不像下面这样

```
Model.where('field >= ?', value)
```

Django完成的方式不是太好，但以我的观点，却更加简介。在Django对应的代码如下：

```
Model.objects.filter(field__gt=value)
```

控制器(Controller)

控制器的工作就是利用请求返回准确的应答。网络应用程序典型工作是支持添加，编辑，删除和显示具体的资源，而RoR的便捷表现在使开发控制器的工作简单而贴心。控制器被拆分为若干个方法(method)，每个方法代表指定的动作(action)(show代表请求某个资源，new代表显示创建资源的表单,create代表从new接收POST的数据并真正的创建资源)。控制器的实例变量(以@为前缀)会自动被传递给视图(view)，Rails从方法名称就会识别应该把哪个模板(template)作为视图。

```
class ProductsController < ApplicationController
  # 自动渲染views/products/show.html.erb
  def show
```

```
    # params是包含请求变量的ruby hash
    # 实例变量会自动被传递给视图
    @product = Product.find(params[:id])
  end

  # 返回空的product，渲染views/products/new.html.erb
  def new
    @product = Product.new
  end

  # 接收用户提交的POST数据。多数来至于在'new'视图中表单
  def create
    @product = Product.new(params[:product])
    if @product.save
      redirect_to @product
    else
      # 重写渲染create.html.erb的默认行为
      render "new"
    end
  end
end
```

Django使用两种不同的方式实现控制器。你可以使用一个方法来实现每个动作，与Rails做法非常相似，或者你可以为每个控制器动作创建一个类。 Django没有区分new和create方法，资源的创建和空资源的创建发生在同一个控制器中。也没有便捷的方法命名你的视图。视图变量需要从控制器显式的传递，而使用的模板文件也需要显式的设置。

```
# django通常称 'show' 方法为'detail'
# product_id 参数由route传递过来
def detail(request, product_id):
    p = Product.objects.get(pk=product_id) # pk 表示主键

    # 使用传递的第三个参数作为内容渲染detail.html
    return render(request, 'products/detail.html', {'product': p})

def create(request):
    # 检查表单是否提交
    if request.method == 'POST':
        # 类似于RoR的 'create' 动作
        form = ProductForm(request.POST) # 绑定于POST数据的表单
        if form.is_valid(): # 所有的验证通过
            new_product = form.save()
            return HttpResponseRedirect(new_product.get_absolute_url())
    else:
        # 类似于RoR的 'new' 动作
        form = ProductForm() # 空的表单

    return render(request, 'products/create.html', { 'form': form })
```

在以上Django的例子中代码数量与RoR相比很明显。Django似乎也注意到这个问题，于是利用继承和mixin开发出了第二种实现控制器的方法。第二种方法称为基于类的视图(class-based views) (注意，

Django称这个控制器为view)，并且在Django 1.5中引入以提高代码重用。很多常用的动作都存在可被用来继承的类，比如对资源的显示，列表，创建和更新等，这大大简化了代码开发。重复的工作比如指定将被使用的视图文件名称，获取对象并向view传递该对象等工作也会被自动完成。上面相同的例子使用这种方式只有四行代码。

```
# 假设route传递了名为 'pk' 的参数，包含对象的 id 并使用该id获得对象。
# 自动渲染视图 /products/product_detail.html
# 并将product作为上下文(context)变量传递给该视图
class ProductDetail(DetailView):
    model = Product

# 为给定的模型生成表单。如果得到POST数据
# 自动验证表单并创建资源。
# 自动渲染视图 /products/product_create.html
# 并将表单作为上下文变量传递给视图
class ProductCreate(CreateView):
    model = Product
```

当控制器比较简单时，使用基于类的视图(class-based views)通常是最好的选择，因为代码会变得紧密，具有可读性。但是，取决于你的控制器的不标准(non-standard)程度，可能会需要重写很多函数来得到想要的功能。常遇到的情况就是程序员想向视图传递更多的变量，这时可以重写get_context_data函数来完成。你是不是想按照当前对象(模型实例)的特定的字段来渲染不同的模板？你只好重写render_to_response函数。你想不想改变获得对象的方式(默认是使用主键字段pk)？你只好重写get_object。例如，如果我们想要通过产品名称选择产品而不是id，也要把类似的产品传递给我们的视图，代码就有可能像这样：

```
class ProductDetail(DetailView):
    model = Product

    def get_object(self, queryset=None):
        return get_object_or_404(Product, key=self.kwargs.get('name'))

    def get_context_data(self, **kwargs):
        # 先调用基类函数获取上下文
        context = super(ProductDetail, self).get_context_data(**kwargs)

        # 在相关产品(product)中添加
        context['related_products'] = self.get_object().related_products
        return context
```

视图

Rails 视图使用 内置的Ruby 模板系统，它可以让你在你的模板里面编写任意的Ruby代码. 这就意味着它非常强大和快速, 而非常强大的同时就意味着非常大的责任. 你不得不非常小心的不去把表现层同任何其它类型的逻辑混在一起. 这里我需要再次提到涉及一位学生的例子. 一位新同学加入了我们的RoR项目，并且在学习一项新特性. 代码审查的时间到了. 我们首先从控制器开始，第一件令我吃惊的事情是他写的控制器里面代码非常少. 我转而很快去看看他写的视图，看到了大块混着HTML的ruby代码. 诚然，Rails并不会嫌弃缺乏经验的程序员，但我的观点是框架可以帮助开发者避免一些坏的实践. 例如 Django 就有一个非常简洁的 模板语言. 你可以进行if判断以及通过for循环进行迭代，但是没有方法选择没有从控制器传入的对象，因为它并不会执行任意的Python表达式. 这是一个我认为可以敦促开发者方向正确的设计决定. 这能让我们项目中的新手找到组织他们代码的正确方式.

资源: CSS, Javascript 以及 图片

Rails 有一个很不错的内置 资源管道. Rails 的资源管道具有对JavaScript和CSS文件进行串联、最小化和压缩的能力. 不仅仅如此，它也还支持诸如 Coffeescript, Sass 和 ERB 等其它语言. Django 对资源的支持同Rails相比就显得相形见绌了，它把要麻烦都丢给了开发者去处理. Django 唯一提供的就是所谓的 静态文件, 这基本上就只是从每个应用程序那里将所有的静态文件集合到一个位置. 有一个叫做 django_compressor 的第三方app提供了一种类似于Rails的资源管道的解决方案.

表单(Forms)

网络应用中的表单是用户输入(input)的界面。在Rails中的表单包含在视图中直接使用的帮助方法(method)。

```
<%= form_tag("/contact", method: "post") do %>
  <%= label_tag(:subject, "Subject:") %>
  <%= text_field_tag(:subject) %>
  <%= label_tag(:message, "Message:") %>
  <%= text_field_tag(:message) %>
  <%= label_tag(:subject, "Sender:") %>
  <%= text_field_tag(:sender) %>
  <%= label_tag(:subject, "CC myself:") %>
  <%= check_box_tag(:sender) %>
  <%= submit_tag("Search") %>
<% end %>
```

像subject,message这样的输入字段可以在控制器中通过ruby哈希 (类似字典的结构)params来读取，比如params[:subject]和params[:message]。Django通过另一种方式抽象了表单概念。表单封装了字段并包含验证规则。它们看起来像是模型。

```
class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField()
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False)
```

Django会将CharField解析为对应HTML元素的文本输入框，将BooleanField解析为单选框。你可以按照自己的意愿使用 widget 字段更换为其他输入元素。Django的表单会在控制器中实例化。

```
def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            return HttpResponseRedirect('/thanks/') # POST之后重定向
    else:
        form = ContactForm() # An unbound form

    return render(request, 'contact.html', { 'form': form })
```

Django会自动添加验证信息。默认情况下所有的字段都是必须的，除非特意定义(比如cc_myself)。使用上面的代码片段，如果表单验证失败，带有错误信息的表单会自动重新显示，已经输入的内容也会显示。下面的代码在视图中显示显示了一个表单。

```
<form action="/contact/" method="post">
{{ form.as_p }} <!-- 生成类似于rails的表单 -->
<input type="submit" value="Submit" />
</form>
```

URL 和 Route

Route 是将特定的URL匹配到指定控制器的工作。Rails建立REST网络服务非常轻松，而route以HTTP的行为(verbs)来表达。

```
get '/products/:id', to: 'products#show'
```

以上的例子显示，向/products/any_id发起的GET请求会自动route到products控制器和show动作(action)。感谢惯例优先原则(convention-over-configuration)，对于建立包含所有动作(create，show，index等等)的控制器的这种常见任务，RoR建立了一种快速声明所有常用route的方法，叫做resources。如果你依照Rails的惯例(convention)命名了控制器的方法时这会很方便。

```
# automatically maps GET /products/:id to products#show
#                    GET /products to products#index
#                    POST /products to products#create
#                    DELETE /products/:id to products#destroy
#                    etc.
resources :products
```

Django不是通过HTTP的行为来决定route。而是使用更复杂的使用正则表达式来匹配URL和对应的控制器。

```
urlpatterns = patterns('',
    # 在products控制器中匹配具体方法

    url(r'^products/(?P\d+)/$', products.views.DetailView.as_view(), name='detail'),
    # 匹配index方法就获得了主页
    url(r'^products/$', products.views.IndexView.as_view(), name='index'),
    url(r'^products/create/$', products.views.CreateView.as_view(), name='create'),
    url(r'^products/(?P\d+)/delete/$', products.views.DeleteView.as_view(), name='delete'),
)
```

由于使用了正则表达式，框架会自动使用单纯的验证。请求/products/test/会因匹配不到任何route而返回404，因为test不是正确的数字。不同的哲学思想在这里又一次出现。Django在命名控制器动作方面确实方便一些，以至于Django就没有像Rails的resource那样方便的助手，而且每个route必须显式的定义。这将导致每个控制器需要若干个route规则。

测试

在Rails中测试很轻松，与Django比较起来更需要着重强调。

Fixture

两个框架以相似的方式都支持fixture（示例数据）。我却给Rails更高的评价，因为它更实用，能从文件的名称得知你在使用哪个模板。Rails使用YAML格式的fixture，这是人类可读的数据序列化格式。

```
# users.yml (Rails当前知道我们在使用user的fixtures)
john:
  name: John Smith
  birthday: 1989-04-17
  profession: Blacksmith

bob:
  name: Bob Costa
  birthday: 1973-08-10
  profession: Surfer
```

所有的fixture会自动加载而且在测试中可以作为本地变量来访问。

```
users(:john).name # John Smith
```

Django也支持YAML格式的fixture但是开发人员更倾向于使用JSON格式。

```
[
  {
    "model": "auth.user",
    "fields": {
      "name": "John Smith",
      "birthday": "1989-04-17",
      "profession": "Blacksmith",
    }
  },
  {
    "model": "auth.user",
    "fields": {
      "name": "Bob Costa",
      "birthday": "1973-08-10",
      "profession": "Surfer",
    }
  }
]
```

这没什么吸引力，注意它有多啰嗦，因为你必须显式的定义它属于哪个模板，然后在 `fields` 下面列出每个字段。

测试模板

在单元测试模板时两种框架的方式基本一致。使用一组不同类型的断言来进行确定，比如 `assert_equal` , `assert_not_equal` , `assert_nil` , `assert_raises` 等等。

```
class AnimalTest < ActiveSupport::TestCase
  test "Animals that can speak are correctly identified" do
     assert_equal animals(:lion).speak(), 'The lion says "roar"'
     assert_equal animals(:cat).speak(), 'The cat says "meow"'
  end
end
```

类似功能的代码在Django非常相似。

```
class AnimalTestCase(TestCase):
    def test_animals_can_speak(self):
        """Animals that can speak are correctly identified"""
        # no way of directly accessing the fixtures, so we have to
        # manually select the objects
        lion = Animal.objects.get(name="lion")
        cat = Animal.objects.get(name="cat")
        self.assertEqual(lion.speak(), 'The lion says "roar"')
        self.assertEqual(cat.speak(), 'The cat says "meow"')
```

测试控制器（controller）

Rails又因为它魅力而更胜一筹。Rails 使用类名称来决定哪个控制器正在被测试，而测试某个特定动作（action）就像调用 `http_verb :action_name` 一样简单。我们看一下例子。

```
class UsersControllerTest < ActionController::TestCase
  test "should get index" do
    get :index # 向index 动作发起GET请求
    assert_response :success # 请求返回200
    # assigns是包含所有实例变量的hash
    assert_not_nil assigns(:users)
  end
end
```

上面的代码很容易理解正在发生什么。第一行测试模拟了向 `User` 控制器的 `index` 动作发起一个请求。第二行随后检查请求是否成功（返回代码200-299）。 `assigns` 是一个hash，包含了传递到视图（view）的实例变量。所以第三行检查是否存在名为 `users` 的实例变量并且值不是 `nil` 。

也有一些类似于 `assert_difference` 这样方便的断言帮助方法。

```
  # assert_difference检查被测试的数字在开始和结束之间是否更改
  assert_difference('Post.count') do
    # 创建post
    post :create, post: {title: 'Some title'}
  end
```

在Django中测试控制器可以通过使用一个叫 `Client` 类来完成，它扮演着虚拟浏览器(dummy web browser)的角色。下面是Django中对应的代码。

```
class UsersTest(unittest.TestCase):
    def setUp(self):
        self.client = Client()

    def test_index(self):
        """ should get index """
        response = self.client.get(reverse('users:index'))
        self.assertEqual(response.status_code, 200)
        self.assertIsNotNone(response.context['users'])
```

首先我们必须在测试设置时初始化 `Client` 。 `test_index` 的第一行模拟了向 `Users` 控制器的 `index` 动作申请了一个 `GET` 请求。 `reverse` 查找对应index动作的URL。注意代码是如此冗余并且没有类似于 `assert_response :success` 的帮助方法。 `response.context` 包含我们传递给视图的变量。

很显然Rails的magic是相当有帮助的。Rails/Ruby同时也拥有很多第三方app，比如factory_girl，RSpec，Mocha，Cucumber，这使得编写测试是一种乐趣。

工具和其他特征

依赖性管理（Dependency management）

两种框架都有出色的依赖性管理工具。Rails使用 Bundler 来读取Gemfile文件并跟踪文件中对应ruby应用程序运行所依赖的gem。

```
gem 'nokogiri'
gem 'rails', '3.0.0.beta3'
gem 'rack', '>=1.0'
gem 'thin', '~>1.1'
```

简单的在Gemfile文件中添加一个新行即可完成增加依赖（ Dependency）。通过简单调用如下命令即可安装所有需要的gem：

```
bundle install
```

Django强烈推荐使用 virtualenv 来隔离Python环境。 pip 则用来管理python包。单独的python包的安装可以通过以下命令完成：

```
pip install django-debug-toolbar
```

而项目依赖文件可以通过以下集中起来：

```
pip freeze > requirements.txt
```

管理命令

基本上每个项目完成时都有相同的管理工作要做，比如预编译文件（precompiling assets），清理记录（log）等等。Rails使用Rake来管理这些任务。Rake非常灵活而且将开发任务变得简单，特别是依赖于其他任务的。

```
desc "吃掉食物。在吃之前需要烹饪(Cooks)和设置表格(table)。"
task eat: [:cook, :set_the_table] do
  # 在吃掉美味的食物之前, :cook和:set_the_table需要做完
  # 吃的哪部分代码可以写在这里
end
```

Rake的任务可以有前提条件（prerequisite）。上面称为 `eat` 的任务，在执行之前必须运行任务 `cook` 和任务 `set_the_table` 。Rake也支持命名空间（namespace），能将相同的任务结合成组（group）来完成。执行任务只需简单的调用任务的名称：

```
rake eat
```

Django管理命令就没那么灵活而且不支持前提条件和命名空间。虽然任务最终也会完成，但不是很出色。

```
class Command(BaseCommand):
    help = '吃掉食物'

    def handle(self, *args, **options):
        call_command('cook') # 这里是如何在代码中调用管理命令
        set_the_table() # 但子任务需要是常规的python函数
        # 这里是吃的那些代码
```

如果我们将上面内容保存到 `eat.py` 中，我们可以如下调用它：

```
python manage.py eat
```

国际化和本地化

在Rails中国际化有些偏弱。在文件夹config/locales，翻译字符串在文件中作为ruby哈希来定义。

```
# config/locales/en.yml
en: # the language identifier
  greet_username: "Hello, %{user}!" # translation_key: "value"

# config/locales/pt.yml
pt:
  greet_username: "Olá, %{user}!"
```

通过函数t进行翻译。函数第一个变量是决定哪个字符串需要使用的key（比如greet_username）。Rails会自动选择正确的语言。

```
t('greet_username', user: "Bill") # Hi, Bill or Olá, Bill
```

我发现处理本地语言文件中key名字和手动注册这些内容很繁琐。Django将其打包进非常便捷的gettext。翻译也是通过一个帮助函数完成（ugettext），但是这次的key是不需要翻译的字符串本身。

```
ugettext('Hi, %(user)s.') % {'user': 'Bill'} # Hi, Bill 或者 Olá, Bill
```

Django会检查所有的源代码并调用以下命令自动收集将要翻译的字符串：

```
django-admin.py makemessages -a
```

上面的命令执行后会为每一种你想要的翻译的语言生成一个文件。文件内容可能像这样：

```
# locale/pt_BR/LC_MESSAGES/django.po
msgid "Hi, %(user)s." # key
msgstr "Olá, %(user)s" # 值 (翻译)
```

注意到我已经在msgstr填充了翻译内容（那些本来是空的）。一旦翻译完成，必须对它们进行编译。

```
django-admin.py compilemessages
```

这种本地化项目的方法实际上更实用，因为不需要考虑key的名字，在需要的时候也不需要现查找。

【译注】即无需自定义key，django会将整句话作为key值代入

用户授权

不得不说当我知道RoR（Ruby on Rails）没有打包任何形式的用户授权时多少有点震惊。我想不出任何不需要授权和用户管理的项目。在这方面最受欢迎的gem是devise，毫无疑问也是Rails上最受欢迎的，在Github上有Rails一半的得分。

尽管Django从最开始就将授权框架打包进来，但直到一年之前这种授权方式的灵活性才有所改善，就是当版本1.5发布并带来可配置的用户模型（user model）。之前，你会被强制要求使用Django的方式定义用户，而不能任意更改字段或者添加字段（field）。如今这不再是问题，你可以用自己定义的用户模型代替原有模型

第三方库

这里没什么好说的。这篇文章里已经提到了很多二者可使用的第三方库，而且都拥有太多的app。Django Packages是个非常好的网站，可以用来搜索Django的App。不过还未发现Rails有类似的网站。

社区

虽然我没有更具体的数据来证明，但我相当确定Rails的社区更大一些。在Github上RoR拥有Django两倍的得分。在Stackoverflow上标记为Rails的问题也有两倍之多。而且似乎RoR比Django有更多的工作（在Stackoverflow职业中241对58）。Rails很庞大而且有非常多迷人的资源来供学习，比如Rails Casts和Rails for Zombies。Django拥有Getting Started with Django但是没有可比性。我知道Django使用The Django Book，但是已经落后若干年了。不要以为我说错了，尽管有很多Django团体而且如果你遇到问题，你很容易通过google找到答案，但Django就是没有Rails庞大。

结论

Ruby on Rails和Django在网络开发方面都是非常出色的框架。在开发模块化的简洁的代码，在减少开发时间。我已经离不开ORM框架，模板引擎和会话管理系统。那么问题是我如何选择呢？

选择任何一个都不会错。我的建议通常就是两者都使用并选出你最合适的。最终的决定会取决于你倾向于哪种语言或者哪种原则：惯例优先原则（convention-over-configuration，CoC）还是显式优先隐式原则（explicit is better than implicit）。使用CoC可以自动加载（import），控制器实例会自动传递给视图以及便捷的编写测试。使用显式优先隐式，会明确知道代码在做什么，即使对那些不熟悉框架的人。

从我个人经验来看我更喜欢Django。我喜欢Python的明确（explicitness），喜欢Django的表单以及此框架更有防御性（有限的模板语言，在model字段中 `null` 默认不可用）。但我也知道更多人离开了Rails的魔法和它优秀的测试环境是没法活的。

本文地址：https://www.oschina.net/translate/rails-vs-django-an-in-depth-technical-comparison
原文地址：https://bernardopires.com/2014/03/rails-vs-django-an-in-depth-technical-comparison/
