简介

Symfony2是一个基于PHP语言的Web开发框架，有着开发速度快、性能高等特点。但Symfony2的学习曲线也比较陡峭，没有经验的初学者往往需要一些练习才能掌握其特性。

本文通过一个快速开发寻人平台的实例向读者介绍Symfony2框架的一些核心功能和特点。通过阅读本文，你可以通过一些具体的例子了解Symfony2框架的优秀特性和技术特点，从而体会到使用Symfony2框架支持快速网站开发这一优势。

适合人群

- 本文适用于希望提高PHP语言的开发技术，或者对Symfony2框架有兴趣的读者。
- 本文也适用于系统架构师和各类技术决策者。

1.前言

在不久前的4月20日，中国四川省雅安地区发生了7.0级地震，累计受灾人数达到200多万。寻人平台在这样的情况下能够起到很大的帮助，而且，寻人平台越早上线，实用价值就越高。

Symfony2可以用来支持大型网站的建设，在中小型网站的快速搭建和开发上也有着非常好的支持。我借由这次撰文的机会，向大家具体地分享一下我是如何在3个小时内基于Symfony2开发出来一套支持PFIF[^1]格式的网站寻人平台的，希望读者能够对Symfony2的各个组件以及功能产生一些了解。

[^1]: People Finder Interchange Format（wiki）是一个被广泛使用的开放的数据结构及标准，灾难发生后可以用该标准在不同的组织或网站间交换寻人信息，帮助失去联系的人找到彼此。

2.Bundle的使用

Symfony2框架以及相关社区最大的特点之一就是支持Bundle。什么是Bundle呢？简单来说，Bundle就是一种“功能”的抽象。通过把一类具体的问题抽象成一个Bundle，可以把一个系统的逻辑进行切分：Bundle的开发者可以专注在某类问题的解决上，而Bundle的使用者则可以把工作的重心放在自己的业务逻辑上。

在互联网开发领域，存在着大量可以被抽象的功能。比如用户登录系统，比如新闻评论，比如JS/CSS文件的压缩和合并等等。举个具体的例子，比如用户登录系统，大部分项目对于用户系统的需求其实都是差不多的，但每次要开发新产品的时候，都多多少少会去重新造一整个或一部分用户

系统的轮子。而一个专门用来负责管理用户系统的Bundle的出现则会减轻这些项目的开发压力，提高项目质量的同时可以加快项目的整体开发速度。

Symfony2也支持Bundle。Symfony2的社区有大量由社区进行维护的Bundle，使用这些开源的Bundle可以让我们的项目直接拥有那部分Bundle所提供的功能。

以下列举了本项目中用到的一些第三方Bundle以及所对应负责的任务。

Bundle名	功能介绍	在项目中的职责
MopaBootstrapBundle	提供基于Bootstrap的页面结构和模板	提供页面的基本HTML架构，样式
NelmioApiDocBundle	自动生成API的文档及接口测试工具	生成API文档以及接口测试工具，并允许工程师及第三方调用者使用工具测试接口是否正常
JMSSerializerBundle	对象进行序列化工具	在接口中，将Doctrine2生成出来的Entity对象转换为Json格式

需要安装一个Bundle，通常只需要两步：

1. 使用composer安装这些Bundle
2. 对Symfony2进行配置，开启这些Bundle的支持并且做一些设置工作。

大部分Bundle通过以上两步就能够被集成进你的项目中，安装这些Bundle只需要修改一些配置文件并且运行一个系统命令即可。

3.数据库建表

Symfony2默认使用Doctrine2作为其ORM组件，而Doctrine2允许开发者通过定义一个普通的PHP类，再通过这个类生成相应的表结构（而不是像一些ORM会反过来做，先生成表结构才能生成类文件），所以我们可以通过熟悉的PHP语法来做建表这件事。而Doctrine2也支持Annotation，所以对于具体字段的定义我们就可以放在注释里，比如这个寻人项目中的Person表的定义文件Person.php是这样的：

```
```
<?php

namespace Scourgen\PersonFinderBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Person
```
```

```
 *
 * @ORM\Table()
 * @ORM\Entity(repositoryClass="Scourgen\PersonFinderBundle\Entity\PersonReposi
 */
class Person
{
    /**
     * @var integer
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @ORM\OneToMany(targetEntity="Note", mappedBy="person")
     **/
    private $person_records;

    /**
     * @ORM\OneToMany(targetEntity="Note", mappedBy="linked_person")
     **/
    private $linked_person_records;

    /**
     * @ORM\Column(type="datetime",nullable=true)
     */
    private $entry_date;

    /**
     * @ORM\Column(type="datetime",nullable=true)
     */
    private $expiry_date;

    /**
     * @ORM\Column(type="string", length=45,nullable=true)
     */
    private $author_name;

    /**
     * @ORM\Column(type="string", length=45,nullable=true)
     */
```

```
    private $author_email;

    ...
```

这是一个典型的php文件，我们使用了Annotation语法对每个字段的类型进行了定义。我们甚至可以通过Annotation语法定义表之间的外键关系（比如我们在上面的源代码中定义了person_records字段对Notes表的OneToMany关系，这种关系最终映射在数据库里就会体现成为两个表之间的外键）。

定义完成之后，我们就可以通过Doctrine2的一个命令去补全这个类文件的get和set方法：

```
php app/console doctrine:generate:entities
```

自动补全完毕之后，这个类就是一个可以使用的ORM对象了，你可以在项目的任何地方去实例化这个Person对象，然后通过setxxxx和getxxxx系列方法，像操作一个类一样去操作数据库里的一条记录。

但此时此刻，数据库本身缺还没有生成，我们可以通过下面这条命令把这些类生成相对应的数据库。

```
php app/console doctrine:schema:create
```

而此时MySQL里一个实际可用的数据库表就已经被生成出来了。

而当字段需要变更时，仅需要修改上面那个PHP类（person.php），然后运行doctrine2的update命令，Doctrine2会自动分析现有表结构和目标表结构的不同，然后生成相应的update schema语句并执行。

到此时为止，数据库定义工作就已经完成了，操作数据库需要的一些类也已经准备好，我们下面看一下如何快速把HTML页面结构搭建起来。

3.页面结构和layout

要开始进行业务逻辑开发之前，另外一件重要的事情就是先要把网站的HTML页面结构搭建起来。虽然业界也有一些成熟的框架，例如Bootstrap等，但由于这些前端框架都是单独的项目，在真实项

目的实用中总会有一些偏差，要做一些适配工作，也需要工程师把两个系统进行整合。而幸运的是，我们可以使用一个Bundle把Symfony2和Bootstrap整合在一起，这个Bundle叫做MopaBootstrapBundle。

我们看一个MopaBootstrapBundle自带的layout片段：

```
```
{% block body %}
    {% block navbar %}
    {{ mopa_bootstrap_navbar('frontendNavbar') }}
    {% endblock navbar %}

    {% block container %}
    <div class="{% block container_class %}container-fluid{% endblock container
        {% block header %}
        {% endblock header %}

            {% block page_header %}

                  <h1>{% block headline %}Mopa Bootstrap Bundle{% endblock head

            {% endblock page_header %}

            {% block flashes %}
            {% if app.session.flashbag.peekAll|length > 0 %}

                {{ session_flash() }}

            {% endif %}
            {% endblock flashes %}

            {% block content_row %}

                {% block content %}

                    {% block content_content %}
                    Hier könnte Ihre Werbung stehen ...
                    {% endblock content_content %}

```
```

```
                <div class="span3">
                    {% block content_sidebar %}
                    <h2>Sidebar</h2>
                    {% endblock content_sidebar %}
                </div>
                {% endblock content %}
            </div>
            {% endblock content_row %}
        </div>
```

可以看到，MopaBootstrapBundle已经帮我们做好了页面布局以及block的定位工作，我们只需要在页面中集成它提供的这个layout，然后再通过block复写，把特定的区块改成我们想要的样子，就能够很快速的完成一个页面的布局工作。

我通过两个步骤来具体解释一下这是如何做到的：

1.复写MopaBootstrapBundle自带的layout，实现全局统一的导航条及页脚等信息。

```
{% extends 'MopaBootstrapBundle::base.html.twig' %}

…

{% block header %}
    <div class="navbar">
        <div class="navbar-inner">
            <div class="container">
                <button type="button" class="btn btn-navbar" data-toggle="colla
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <div class="nav-collapse collapse">
                    <ul class="nav">
                        <li><a href="/">主页</a></li>
                        <li class="divider-vertical"></li>
                        <li{% if person_active is defined %} class="active"{% e
                        <li class="divider-vertical"></li>
                        <!--li{% if note_active is defined %} class="active"{%
                    </ul>
                </div>
```

```
            </div>
        </div>
    </div>
{% endblock header %}

```

通过上面的代码可以看到，通过继承并复写MopaBootstrapBundle的base.html.twig这个layout，我们重新定义了header这个block，所以其他页面都可以通过继承这个新的layout来显示公用的导航条。

如果对上述解释还不太明白的读者不妨这样想：一个页面的基本布局就是一个类，通过在这个页面布局定义block，等于是赋予了这个类许多的方法和属性。而一个具体的页面就是继承了这个类的一个实例，通过对所继承页面的block的重新定义，就相当于对这个类的方法和属性做了重新的定义。而这种页面布局上的继承和被继承关系是可以拥有无限多层的。这种继承页面的做法，给予了项目在页面布局上极大的灵活性。

而如果通过这种做法对页面布局层级的合理划分（比如全站级，频道级，栏目级，页面级这种典型的四层划分方法），每级都会有自己的页面定义文件，可以单独进行样式的变更和页面的修改，但又不彼此互相影响和冲突，整个项目的页面布局及管理也会层级清晰，开发起来也会非常方便和高效。

2.让我们看一下一个最终页面的源代码是怎样的：

```
```
{% extends 'ScourgenPersonFinderBundle::Layout.html.twig' %}

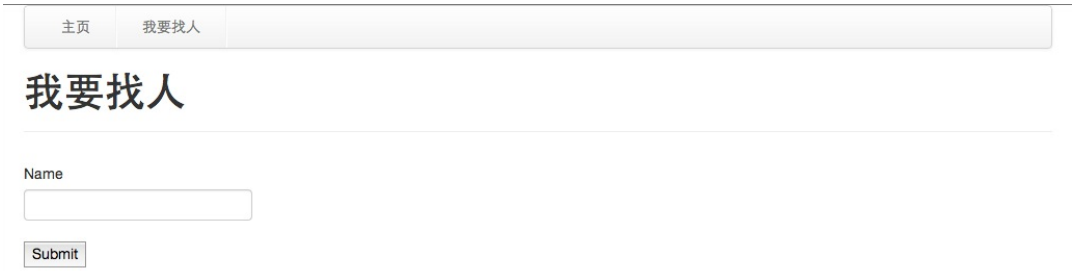
{% set seek_active=1 %}

{% block headline %}
    我要找人
{% endblock %}

{% block content_content %}
    <form action="{{ path('seek_search') }}" method="post" {{ form_enctype(form
        <fieldset>
            {{ form_rest(form) }}
            <input type="submit"/>
        </fieldset>
    </form>
{% endblock %}
```
```

```
```

我们最终得到页面应该是这个样子的：



5.表单验证及提交

细心的读者可能已经发现了，在上一节的最后一段代码中，我们用了几个form_开头的方法，把表单生成了出来，其实这就是Symfony2表单处理功能的强大之处：支持快速搭建表单系统。

我们都知道在应用开发过程当中，大部分工作都是在处理数据，而大部分数据又都是通过表单进行交互和维护的，而在一般情况下，表单处理会占据一个项目相当一部分开发时间。

有没有办法解决这个问题呢？答案当然是有的：

既然表单字段和数据库字段有一定的对应关系，那最理想的状态应该是有一个中间层能够根据数据库表结构自动生成表单系统，允许用户进行对数据库的CRUD操作。

而Symfony2就能够实现以上这点，我们通过一个例子来看看如何完成一个表单的开发。

我们在Controller中做如下的定义：

```
```
    public function indexAction(Request $request)
    {
        $person = new Person();

        $form = $this->createFormBuilder($person)
            ->add('fullname', 'text')
            ->add('description', 'textarea')
```
```

```
            ->getForm();

        return array('form' => $form->createView());
    }
```

通过这三段代码我们做了三件事情：

1. 声明了一个Person对象。
2. 将Person传入创建表单的方法，并且声明我们需要用到两个字段和对应的类型。
3. 我们将这个表单的view创建出来后返回给页面。

然后我们在页面样式文件里作如下定义：

```
<form action="{{ path('post_new_person') }}" method="post" {{ form_enctype(form
        {{ form_rest(form)  }}
        <button type="submit" class="btn">提交</button>
</form>
```

然后我们会得到如下的页面：

当然为了页面美观，我们也可以对这个表单进行一些调整：增加提示文字、表单报错信息的警告、优化一些样式等等。而即使完成了这些优化，最终代码其实也还是非常短：

```
{{ form_errors(form) }}
<form action="{{ path('post_new_person') }}" method="post" {{ form_enctype(form
    <fieldset>
        {{ form_row(form.fullname,{'label':'姓名','attr':{'placeholder':'例如：王
        <span class="help-block">您输入的姓名将成为其他人寻找的依据，请提供他的正式名
        {{ form_row(form.description,{'label':'描述','attr':{'placeholder':'例如
        <button type="submit" class="btn">提交</button>
        {{ form_rest(form)  }}
    </fieldset>
</form>
```

下面再来看一下如何实现表单处理的逻辑，我们对Controller进行一些变更：

```
    /**
```

```
     * @Route("/post_new_person",name="post_new_person")
     * @Template()
     */
    public function indexAction(Request $request)
    {
        $person = new Person();

        $form = $this->createFormBuilder($person)
            ->add('fullname', 'text')
            ->add('description', 'textarea')
            ->getForm();

        if ($request->isMethod('POST')) {
            $form->bind($request);
            if ($form->isValid()) {
                $person->setSourceDate(new \DateTime('now',new \DateTimeZone('U
                $em = $this->getDoctrine()->getManager();
                $em->persist($person);
                $em->flush();
            }
        } else {

        }
        return array('form' => $form->createView());
    }
```

在新增的几段代码中，做了如下的事情：

1. 判断这个请求是否是一个POST请求，如果是的话则进入表单处理逻辑。
2. 将表单和提交的数据进行绑定。
3. 判断表单是否验证通过。
4. 如果验证通过，则把提交的数据持久化到数据库里。

再对代码修改完之后，我们尝试操作一下页面，会发现这已经是一个完整可用的表单了，已经可以通过操作表单往数据库添加数据。这时的页面效果如下图所示：

那么到这个时候，一个完整的处理表单逻辑和相关的页面就已经被开发完毕了，我们总共只写了几十行代码而已。接下来我们依样画葫芦，把寻人平台中的其他表单和界面也都一一对应，应该很快就能完成。

6.API以及文档

对于一个寻人平台或任何一个成熟的系统来说，使用API进行数据的传递是一定需要的，不然就会让网站成为信息的孤岛。在这章里我将介绍如何使用Symfony2开发API接口，并且完成相应的文档和API测试工具。

我们假设需要这么一个API：允许用户通过HTTP协议，根据特定的PersonId获取某个Person的数据，下面看一下实现这些功能的代码是怎样的。

```
```
    /**
     * @Route("/get_person_by_person_id/{person_id}",requirements={"person_id"=
     * @Method("GET")
     * @ApiDoc(
     * resource=true,
     * description="get person by person_id",
     * filters={
     *   {"name"="person_id", "dataType"="integer"}
     * }
     *)
     */
```
```

```
    public function getPersonByPersonIdAction($person_id)
    {
        $odm = $this->getDoctrine()->getManager();
        $serializer = $this->container->get('serializer');

        $person=$odm->getRepository('ScourgenPersonFinderBundle:Person')->find(
        return new Response($serializer->serialize($person,'json'));
    }
```

通过这十几行代码，我们在Annotation里完成了以下功能：

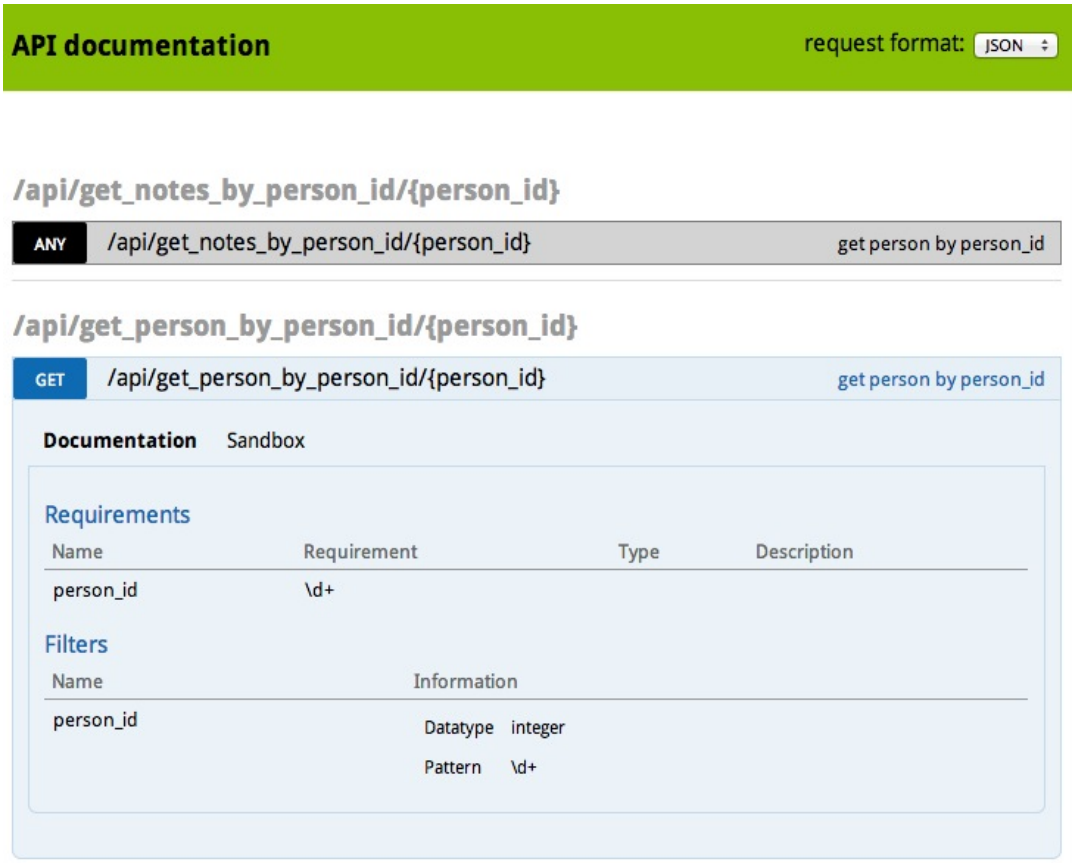
1. 定义了API的URL以及参数，并且限制了传递person_id的参数必须为数字
2. 限定了这个API只能够通过GET方式调用
3. 通过ApiDoc定义了这个API的一些使用条件和说明，以便之后可以对接口进行测试。

而在方法里我们则完成了以下功能：

1. 通过Doctrine2在数据库里找到id是$person_id的Person记录
2. 获取到Person记录后，通过调用serializer这个service，把Person转换为JSON格式。
3. 将JSON格式的数据返回给页面请求者。

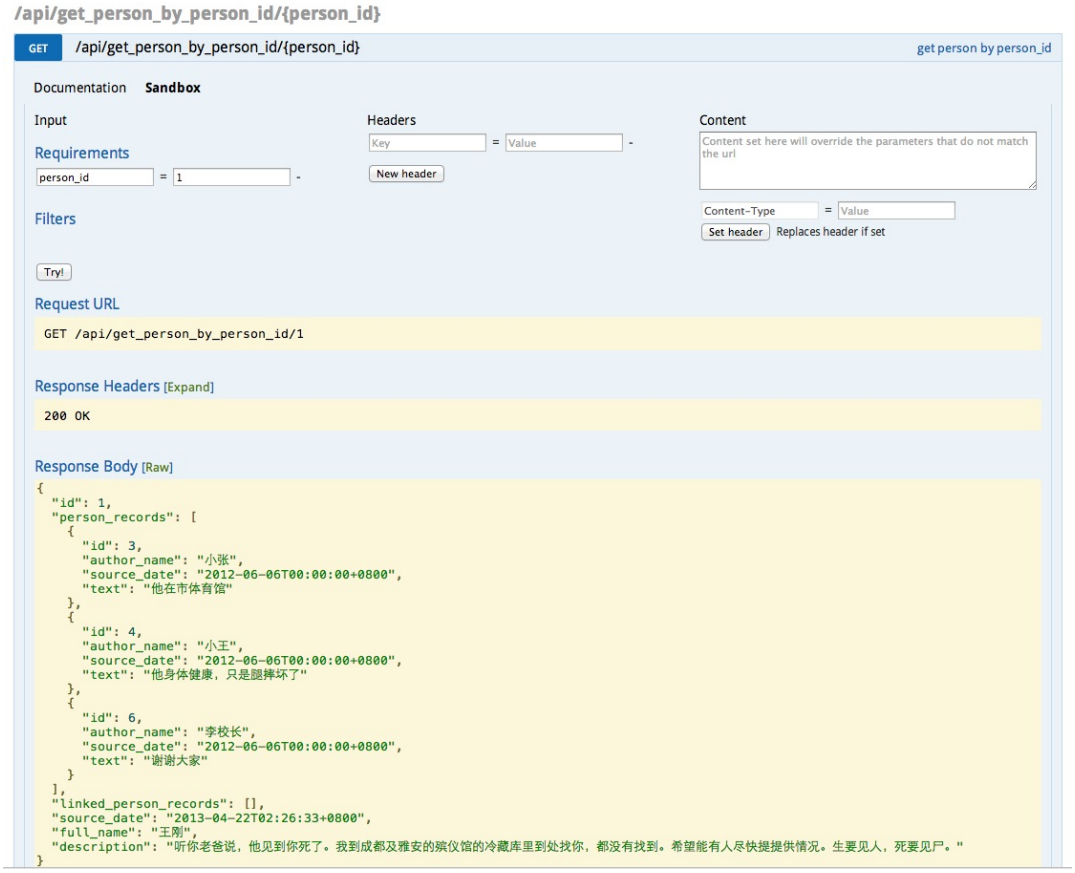
这样一个接口的开发就已经完成了，但接口毕竟不是一个页面，去测试一个接口需要配置各种参数，接口的返回值也需要一定的格式化才能够让人看得懂。所以在这里可以使用NelmioApiDocBundle去生成API的文档和测试工具。我们既然已经在上面的代码中定义了ApiDoc，那么就已经可以直接查看自动生成的API文档，并且使用测试工具了。

我们在浏览器中打开/apc/doc这个页面，即可看到自动生成的文档和测试工具，如下图所示：



在这个界面中显示了所有的API接口以及使用方式、参数定义等等，当然也包括我们刚才编写的/api/getpersonbypersonid这个接口。而文档的内容就是我们刚才在Annotation里定义的。

我们点击Sandbox，就可使用API测试工具对接口进行测试：



在上图中可以看到，我通过API测试工具模拟了一个API请求，并且这个页面工具会自动帮我把返回的JSON格式化，方便查看和调试。

当然读者也可以根据上文的例子依样画葫芦去编写其他的API接口，由于有了文档和工具的支持，即使比较复杂的接口开发起来也不会耗费太多的时间，所以API的开发也算是很快就完成了。

与此同时，整个网站就已经开发完成，一个满足基本需求、界面美观、支持API调用的寻人平台就可以投入使用了。

7.总结

在本项目的开发过程中，读者应该可以体会到Symfony2的这种支持快速建站的特性。使用Symfony2去开发业务逻辑非常复杂的大型网站也并不困难，我将在以后的文章中向读者一一做介绍。

本项目的源代码已经在Github上开源，有兴趣的朋友可以直接去Github上查看所有源代码，也可以克隆一个项目到本地把玩研究一下。