中山大学软件学院 2012 级

软件设计文档

——公益活动管理系统 Philosopher

2015-7-18

项目小组成员

12330054 陈跃东

12330053 陈远宏

12330051 陈宇文

12330084 封伟城

12330293 田　　澍

目录

一． 软件需求概述

鉴于当前许多同学因为公益时不够而发愁，主要的原因是没有一个好的公益活动信息平台。而我们的“公益活动发布平台”是一个 B/S 架构的平台，使得公益活动组织可方可以发布和管理公益活动，个人用户可以通过我们的平台参加相关公益活动。这样既便利了公益活动组织方对于活动的宣传和组织，同时也让同学们有了一个公益活动信息的“集散地”，能够及时得到各种公益活动信息。

每个公益活动组织方的注册都需要进行申请，并且经过系统管理员的同意。每个发布的公益活动都需要有相关信息，例如活动名称，公益时数，报名方式等等。同时，对于希望参加活动的申请者，公益活动组织方可以对他们上传的申请资料进行审核。若通过审核，则可通过系统向他们发送活动参与通知；反之，也可以向他们发送申请失败的通知。当公益活动结束后，公益活动组织方根据每个活动参与者的勤情况，确认活动参与者是否成功完成活动。若成功完成，系统会自动给每个成功参与者的账户加上相应的公益时。

而对于每个个人用户，都有一个与自己相关的活动列表，每个活动有一个状态，；例如：“正在申请”、“被拒绝”、“成功参与”。用户登录后可以浏览并申请相应的公益活动，。如果申请被通过，用户可以收到活动的参与通知。当活动结束，如果用户成功参与，活动组织方将进行确认，这样用户的公益时数将会相应增加，同时用户也可以对活动进行反馈。当然每个学期开始，用户的公益时数会进行清零操作。

综上所述，我们想构建一个轻量级，且功能完备，使用便利的在校公益发布平台。平台最终的实现形式将会是，基于 PC 上的 B/S 架构的网站系统。

二． 技术选型说明

1. **版本控制工具与团队管理**：Git、Github

- 由于是多人协同开发，所以为了进行团队内代码版本的控制，我们需要一个版本控制工具，这里我们选择了 Git 来作为我们团队的版本控制工具，Git 是一个开源的分布式版本控制系统，用以有效、高速的处理从很小到非常大的项目版本管理。相比 SVN 来讲，它更适合团队开发，因为每个团队成员都可以保存一个独立的版本库，而 SVN 则只有一个中心版本库，且值得一提的是 Git 是 Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的版本控制软件，所以我们选择 Git 作为我们的版本控制工具
- 至于团队管理平台，我们选择 Github，GitHub 可以托管各种 git 库，并提供一个 web 界面，但与其它像 SourceForge 或 Google Code 这样的服务不同，GitHub 的独特卖点在于从另外一个项目进行分支的简易性。为一个项目贡献代码非常简单：

首先点击项目站点的“fork”的按 钮，然后将代码检出并将修改加入到刚才分出的代码库中，最后通过内建的“pull request”机制向项目负责人申请代码合并。Github 是全球最大的 IT 开源社区，许多著名的开源项目都在上做团队开发，例如 Ruby on Rails 和 redis 等。

2. 后端开发框架：Django on Python

- Python 作为一门脚本语言，虽然其执行效率上不及 C 与 C++，但其最大的优势就是 Python 有一个庞大的模块系统，能提供许多功能强大的模块，基本上你能想到的她都有，用 Python 进行开发可以大大缩短开发周期和代码量，这是 C 与 C++ 所不及的地方。所以为了缩短我们的开发周期，进行敏捷开发，我们选择 Python 作为我们的后端开发语言
- Django 诞生于新闻网站的环境中，所以它提供很多了特性（例如她的的管理后台），非常适合内容类的网站，而我们所要开发的系统就是一个内容类的网站，所以在功能特性上刚好能够匹配
- Django 是一个要求用户基于 MVC 设计模式进行开发的后端框架，易于整体设计，开发起来便捷而高效，这与我们构建一个功能完备的轻量级系统的想法不谋而合
- Django 有活跃的社区支持，作为一个成熟的后端框架 Django 有丰富的技术文档作为参考，所以我们开发起来在技术上就少了很多后顾之忧

3. 数据库：SQLite

在数据库的选择上我们就比较保守了，这里我们选择的是轻量级数据库 SQLite，原因主要是我们所要构建的是一个轻量级的在校公益平台发布系统，所以在数据处理的并发性以及数据容量上并不会有太高要求，一个轻量级的 SQLite 就足以满足我们的需要了，而且 SQLite 是 Django 自带的，有方便的数据操作接口，也省去了环境搭建所需的时间

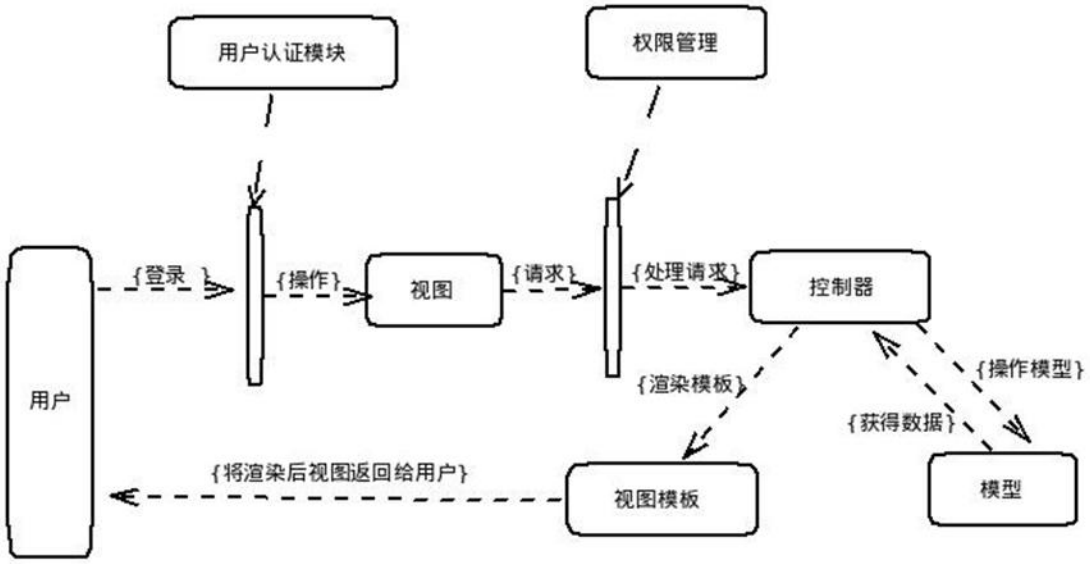
4. 前端框架：Bootstrap、JQuery

- 前端 CSS 框架我们选择 Bootstrap，她来自 Twitter，是目前最受欢迎的前端框架。Bootstrap 是基于 HTML、CSS、JAVASCRIPT 的，它简洁灵活，使得 Web 开发更加快捷。它由 Twitter 的设计师 Mark Otto 和 Jacob Thornton 合作开发，是一个 CSS/HTML 框架。Bootstrap 提供了优雅的 HTML 和 CSS 规范，它即是由动态 CSS 语言 Less 写成。Bootstrap 一经推出后颇受欢迎，一直是 GitHub 上的热门开源项目，包括 NASA 的 MSNBC（微软全国广播公司）的 Breaking News 都使用了该项目。国内一些移动开发者较为熟悉的框架，如 WeX5 前端开源框架等，也是基于 Bootstrap 源码进行性能优化而来。

- 为方便网站前端动态效果的实现，我们选择 Javasrcipt 框架 JQuery，她是继 prototype 之后又一个优秀的 Javascript 库。它是轻量级的 js 库 ，它兼容 CSS3，还兼容各种浏览器（IE 6.0+, FF 1.5+, Safari 2.0+, Opera 9.0+），jQuery2.0 及后续版本将不再支持 IE6/7/8 浏览器。jQuery 使用户能更方便地处理 HTML（标准通用标记语言下的一个应用）、events、实现动画效果，并且方便地为网站提供 AJAX 交互。jQuery 还有一个比较大的优势是，它的文档说明很全，而且各种应用也说得很详细，同时还有许多成熟的插件可供选择。jQuery 能够使用户的 html 页面保持代码和 html 内容分离，也就是说，不用再在 html 里面插入一堆 js 来调用命令了，只需要定义 id 即可。

三. 架构设计

本系统是基于B/S架构使用Python进行开发的，整体的框架可以说就是MVC框架，即分为 Model(模型)、View(视图)、Contrller(控制器)三个大的模块，整体框架如下图 ：



系统对用户的认证和认证后的权限进行了管理，用户操作视图页面，向控制器发出请求，加入权限允许，那么控制器处理相应请求，并操作模型获得相关数据，然后用获得的数据去渲染视图的模板，再将渲染后的视图回传给用户。

- 视图

一般将其称作前端，是系统与用户直接交互的环境，提供用户请求服务器的接口，视图模块提供的功能有：用户登录、用户注册、查看公益活动、申请公益活动、创建公益活动、更改个人信息、审查申请等。

- 控制器

可以说是整个系统的核心部分，即相当于大脑功能，它根据前端用户发来的请求，处理相应数据，并做出反馈，是一个中心枢纽。它有用户登录、用户注册、处理申请、创建公益活动等功能模块。

- 模型

负责操作数据库的模块，这里分为两个子模块： data access layer(dal) 和 web data layer(wdl)。其中 dal 定义了对数据库各个表的增、删、改、查这些基本操作，而 wdl 则是直接定义了每个数据类的数据操作方法，而这些方法是通过调用 dal 实现的。

四． 模块划分

Django 是一个基于 MVC 构造的框架。但是在 Django 中，控制器接受用户输入的部分由框架自行处理，所以 Django 里更关注的是模型（Model）、模板(Template)和视图（Views），称为 MTV 模式。所以，本项目大体上也分为这三大模块。

1. Model

模型，也就是数据存取层。在本项目中对应的文件为/philosopher/models.py，分为三个类 UserProfile(用户信息)，Activity(活动)，Application(申请表)。

另外，还使用 Django 框架自带的类 User 来做为本项目的用户类，优点有二：

- 符合开发规范，提高了代码的可读性
- 使用现成框架，节省开发时间

但是，考虑到 User 中的相关属性相对较少，直接修改底层代码又是非常冒险也不提倡的做法，所以就使用了 UserProfile 类来封装 User 并提供其他相关的属性。这样，当需要对用户添加属性时，只需要直接在 UserProfile 中添加一列就可以了，可扩展性较高。

2. View

视图，即业务逻辑层。存取模型及调取恰当模板的相关逻辑。模型与模板之间的桥梁。Django1.4 框架中，view 层中全部逻辑都是在一个文件 views.py 中实现的，但是，考虑到本项目中逻辑交互较多，全部存放在一个文件中将会降低代码的可读性以及可维护性，故本项目将 View 层细分为四大模块，结构如下：

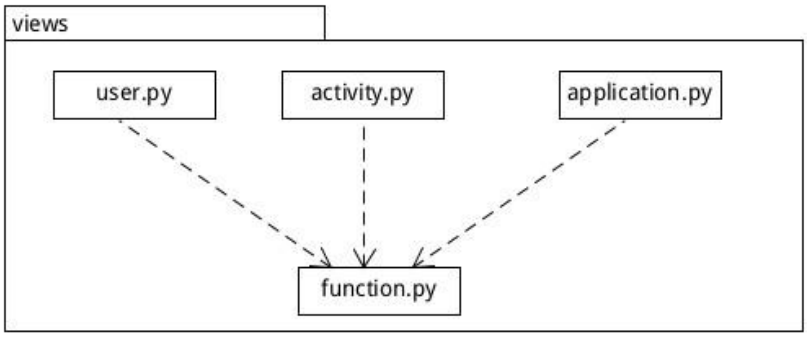
也就是将 views 文件拆分为一个文件夹，其中

funcion.py 封装一些基本操作；

user.py 封装与用户操作相关的操作，如登录，查看个人资料等，同时它也调用 function.py 来使用其基本函数；

其他的文件也封装各自对应的类的相关操作

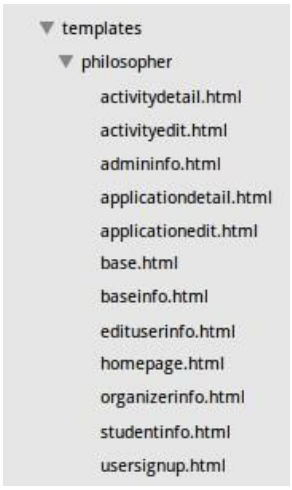
其调用关系图如下：

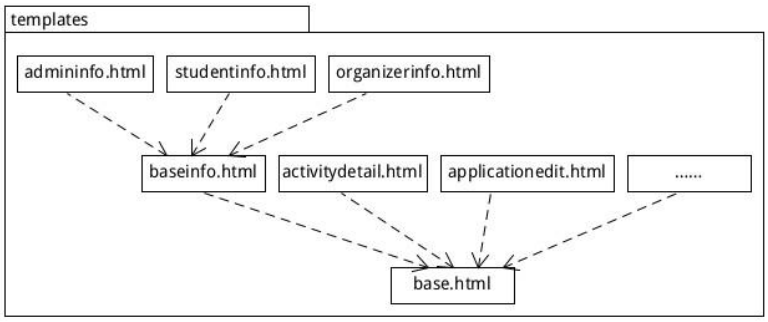


3. Template

模板，即表现层。对应于本项目的/templates/philosopher 文件夹，其使用的是 Django 提供的模板语言，接受后端传递的数据并渲染为 html 页面。本项目中该层的结构如下左图所示；可以理解为它是页面驱动的，基于每个页面提供一个相应的模板页，同时，

他们之间的关系如下右所示





4. 其他相关小模块

依据 Django 开发框架的设定，其他模块也都做了较好的分离。

- 表单模块

存放表单的相关信息，在本项目中对应为/philosopher/forms.py。

- 多媒体模块

存放用户上传的图片资料，在本项目中对应的结构有右所示

- 静态资源模块

存放如 css，jquery，font 等等与前端页面显示相关的资源，其结构如右所示。这样做，一方面可以将前后端合理的分离，方便开发人员的开发；另一方面，将前端相关的代码集中到一个模块，也提高了代码的可读性。

五． 技术实现

1. Structured programming

- SP 简介

结构化程式设计（英语：Structured programming），一种编程典范。它采用子程序、程式码区块(英语:block structures)、for 循环以及 while 循环等结构，来取代传统的 goto。希望借此来改善计算机程序的明晰性、品质以及开发时间，并且避免写出面条式代码。结构化程序设计提出的原则可以归纳为 32 个字：自顶向下，逐步细化；清晰第一，效率第二；书写规范，缩进格式；基本结构，组合而成。

- SP 实现说明

在本项目中，我们将频繁使用到的基本操作分离为子程序，并将其存放于 function.py 文件中，需要使用时直接调用函数即可。这样，使得代码清晰，也方便维护。

比如，项目中的几乎每一个页面都会用到验证用户身份这样一个功能，这样，项目中便将用户身份验证匹配的操作分离为子程序，如下：

```
def is_organizer(user):
        userprofile = user.get_profile()
        if userprofile.role == 'o':
                return True
        else:
                return False
def is_admin(user):
        userprofile = user.get_profile()
        if userprofile.role == 'a':
                return True
        else:
                return False
```

然后，在其他页面中将其 import 进来，如下

```
6   from philosopher.views.function import is_admin, is_organizer, is_student, is_power
```

最后，当需要匹配用户身份时，直接调用函数即可，如下

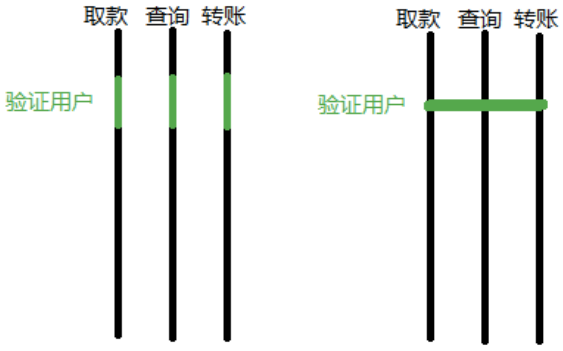
```
@login_required
def  UserInfo(request, userid):
        nowuser = get_object_or_404(User, pk = userid)
        nowuserprofile = nowuser.get_profile()
        isAdmin = is_admin(request.user)
        has_right = False
```

2. Aspect-Oriented Programming

- AOP 简介

AOP， 即面向切面编程，就是把多个流程中重复出现的业务抽取出来，封装好。例如有两个流程，都需要使用到验证功能，此时我们有两种选择，一种是单独的开发两个流程。另一种方法是先把验证功能开发出来，然后两个流程在到达验证业务的时候就接入到已经开发的验证模块。而这个把公用的模块单独出来，不需要重复开发同样的功能的开发思想就是 AOP 思想。以下这一幅截取出来的图片就能够很好的说明问题。

个人认为 AOP 与模块化其实有点相像，但又不完全类似。后者是把每一个功能块独立出来，要使用就调入这个接口。而前者是更抽象的一个概念，像 Model\view\Control 层一样，AOP 就是对特定重用功能再分一层，而一层中可能需要多个模块才能完成这一层的任务



从小而言，可以是重用代码的一种方式。而放大来说，AOP 就是一种软件的分层思想，特别用于多人协作的工程。

- AOP 实现说明

而在我们项目的开发过程中，由于需求和设计做得比较详尽，在开发前我们就确定了有

几个模块是会被几个 Use case 多次用到的，于是基于 AOP 的思想，我们就先开发了这些单独的模块，然后重用这些模块代码。

(1)Front Router 的配置，对 URL 识别分发到控制器

个人认为 URL 的识别分发可以算是一个 AOP 的实现，因为对于 url 请求和控制器调用，如果按照传统的模式，服务器会根据 url 到对应的路径请求文件，然后再从文件调用控制器，最后完成 Conrtrol 层的工作。而 URL 的分发就算是切片编程，在 Control 层与 View 层间再切出一层 Front Router，而这一层的工作就是对 URL 进行识别以及分发到控制器，省却流程中的 url 与控制器识别的步骤。以下是本项目 ActivityManager/urls.py 文件中关于 url 的配置。

```
urlpatterns = patterns('',
    url(r'^admin/', include(admin.site.urls)),

    url(r'^$', 'philosopher.views.activity.Homepage'),
    url(r'^philosopher/accounts/signup/$', 'philosopher.views.user.Signup', name="Signup"),
    url(r'^accounts/login/$', 'django.contrib.auth.views.login', name="Login"),
    url(r'^accounts/logout/$', 'django.contrib.auth.views.logout', {'next_page': '/'}, name="Logout"),
    url(r'^philosopher/accounts/info/(?P<userid>\d+)/$', 'philosopher.views.user.UserInfo', name="UserInfo"),
    url(r'^philosopher/accounts/editinfo/(?P<userid>\d+)/$', 'philosopher.views.user.EditInfo', name="EditUserInfo"),
    url(r'^philosopher/accounts/toggleblack/(?P<userid>\d+)/$', 'philosopher.views.user.ToggleBlack', name="ToggleBlack"),

    url(r'^philosopher/activity/detail/(?P<activityid>\d+)/$', 'philosopher.views.activity.ActivityDetail', name="ActivityDetail"),
    url(r'^philosopher/activity/add/$', 'philosopher.views.activity.AddActivity', name="AddActivity"),
```

(2)数据请求的拦截器，用于整体数据的添加

而数据请求的拦截器则是基于 MVC 模式的，由于 django 使用的是 View 层是使用 template 模板的，只有得到数据的导入，模板才能正常工作。而这一个数据请求拦截器则是为了对付一些每个页面都会使用到的数据，如项目中的右侧栏的排行数据资料。这一层在 View 层与数据请求间划分出来一层，用于添加整体的数据。使我们可以省却每一次取数据都要取一次通用数据的苦况。

实现的具体方法就是在 philosopher 应用中加入一个 context_processer.py 文件用于存放数据的操作函数。

```
def newTen(request):
        from philosopher.models import Activity
        newTen = Activity.objects.filter(publishdate__isnull = False).filter(atstatus=0).order_by('-publishdate')[:9]
        return {"newTen": newTen, }
```

然后，在 settings.py 文件中指明该文件的位置

```
# For using context processors
TEMPLATE_CONTEXT_PROCESSORS = (
    "philosopher.context_processors.newTen",
    "django.contrib.auth.context_processors.auth",
)
```

最后，在调用模板渲染时，加入 RequestContext

```
        return render_to_response('templates/philosopher/applicationdetail.html',
        {"application": application, "has_right":has_right, "activity":activity, "student":student,
        "isOwner":isOwner },
        context_instance = RequestContext(request))
```

这样，当每次将数据发送到 template 中时，调用上下文处理器时，便会自动加入 newTen 的数据，实现每个页面都可以显示前十排行榜

3. Object-oriented programming

- OOP 介绍

面向对象程序设计（OOP）是一种程序设计范型，同时也是一种程序开发的方法。具有相同数据和操作的对象可以归纳成类，对象是类的实例。OOP 将对象作为程序的基本单元，将程序和数据封装其中，以提高软件的重用性、灵活性和扩展性。

OOP 被理解为一种将程序分解为封装数据及相关操作的模块而进行的编程方式。有别于其它编程方式，OOP 中的与某数据类型相关的一系列操作都被有机地封装到该数据类型当中，而非散放于其外，因而 OOP 中的数据类型不仅有着状态，还有着相关的行为。一般 OOP 都具有以下几个基本理论：

类：定义了一件事物的抽象特点。

封装：用抽象的数据类型将数据和基于数据的操作封装在一起，数据被保护在抽象的数据类型内

继承：子类拥有父类的所有数据和操作。

多态：一个程序中同名的不同方法共存的情况。有两种形式的多态：重载和重写。

- OOP 实现说明

公益活动管理系统的编程设计采用了 python 的 web 框架 django，django 是由 python 编写而成的 web 框架，因此它具有面向对象的所有特点。公益活动管理系统采用面向对象编程技术，提高了软件的重用性、灵活性和扩展性。在设计中我们抽象出了现实生活公益活动管理过程中需要用到的类，将申请表，公益活动，用户等对象抽象为类，简化了复杂的问题，如下：

用户

```
class UserProfile(models.Model):
    ROLE_CHOICES = (
        ('s', u"学生"),
        ('o', u"组织者"),
        ('a', u"管理员"),
    )
    SEX_CHOICES = (
        ('f', u"女"),
        ('m', u"男"),
        ('o', u"其他"),
    )
    STATUS_CHOICES = (
        ('n', u'正常'),
        ('b', u'被拉黑'),
        ('w', u'待批准'),
    )
    user = models.OneToOneField(User)
    role = models.CharField(u'角色', max_length=1, choices=ROLE_CHOICES, default=ROLE_CHOICES[0][0])
    gender = models.CharField(u'性别', max_length=1, choices=SEX_CHOICES, default=SEX_CHOICES[0][0])
    status = models.CharField(u'状态', max_length=1, choices=STATUS_CHOICES, default=STATUS_CHOICES[0][0])
    schoolID = models.CharField(u'学号', max_length=20, default="000")
    phone = models.CharField(u'手机号码', max_length=25, default="000")
    attimes = models.IntegerField(u'公益时长', default = 0)
    intro = models.TextField(u'自我介绍', blank = True)
    avatar = models.ImageField(u'头像', upload_to = "image/UserAvatar", blank = True)
    def getAvatar(self):
        if self.avatar and hasattr(self.avatar, 'url'):
            return self.avatar.url
        else:
            return '/media/image/UserAvatar/default.jpg'
    def save(self, *args, **kwargs):
        if not self.pk:
            try:
                p = UserProfile.objects.get(user = self.user)
                self.pk = p.pk
            except UserProfile.DoesNotExist:
                pass

        super(UserProfile, self).save(*args, **kwargs)
```

```
    def __unicode__(self):
        return self.user.username

def create_user_profile(sender, instance, created, **kwargs):
    if created:
        UserProfile.objects.create(user=instance)

post_save.connect(create_user_profile, sender=User)
```

公益活动

```
class Activity(models.Model):
    atorganizer = models.ForeignKey(User)
    #0:pass; 1:reject; 2: wait
    atstatus = models.IntegerField(u'状态', default = 2)
    atname = models.CharField(u'活动名称', max_length=50)
    atcontent = models.TextField(u'活动内容')
    numlimit = models.IntegerField(u'人数限制', default=99999)
    athours = models.IntegerField(u'公益时长', default=0)
    feedback = models.TextField(u'备注')
    createdate = models.DateTimeField(u'创建日期', default = timezone.now, editable=False)
    publishdate = models.DateTimeField(u'发布日期', blank=True, null=True)
    applystart = models.DateTimeField(u'申请开始日期', default = timezone.now)
    applyend = models.DateTimeField(u'申请结束日期')
    doingstart = models.DateTimeField(u'活动开始日期')
    doingend = models.DateTimeField(u'活动结束日期')
    poster = models.ImageField(u'海报', upload_to = "image/ActivityPoster", blank = True)
    def getPoster(self):
        if self.poster and hasattr(self.poster, 'url'):
            return self.poster.url
        else:
            return '/media/image/ActivityPoster/default.jpg'
    def __unicode__(self):
        return self.atname
```

申请表

```
class Application(models.Model):
    GRADE_CHOICES = (
        ('a', '大一'),
        ('b', '大二'),
        ('c', '大三'),
        ('d', '大四'),
        ('e', '研究生'),
        ('f', '博士生'),
    )
    activity = models.ForeignKey(Activity)
    student = models.ForeignKey(User)
    activityid = models.IntegerField()
    #0:pass; 1:reject; 2: wait; 3: finish; 4: failed
    apstatus = models.IntegerField(u'状态', default = 2)
    grade = models.CharField(u'年级', max_length=1, choices=GRADE_CHOICES, default=GRADE_CHOICES[0][0])
    phone = models.CharField(u'手机号码', max_length=25)
    applyreason = models.TextField(u'申请原因')
    feedback = models.TextField(u'反馈')
    def __unicode__(self):
        return self.activity.atname
```

另一方面，公益活动管理系统将类中的数据进行封装。封装是通过限制只有特定类的对象可以访问这一特定类的成员，而它们通常利用接口实现消息的传入传出。这样可以保证数据的安全性，如下：

```
</div>
<div class="user-image col-md-2">
    <img src="{{ nowuserprofile.getAvatar }}" alt="{{ nowuser.username }}" title="{{ nowuser.username }}" />
    {% if has_right %}
        <a href="/philosopher/accounts/editinfo/{{ nowuser.id }}" class="btn btn-primary">修改个人信息</a>
    {% endif %}


</div>
```

前端页面只能通过 getAvatar 获取用户头像。

在公益活动管理系统中，我们还运用了继承的思想，提高了代码的可重用性，在用户类

中，我们继承了 django 自带的 user 类，并定义了系统中用户应该存在的属性。如：

```
)
user = models.OneToOneField(User)
role = models.CharField(u'角色', max_length=1, choices=ROLE_CHOICES, default=ROLE_CHOICES[0][0])
gender = models.CharField(u'性别', max_length=1, choices=SEX_CHOICES, default=SEX_CHOICES[0][0])
status = models.CharField(u'状态', max_length=1, choices=STATUS_CHOICES, default=STATUS_CHOICES[0][0])
schoolID = models.CharField(u'学号', max_length=20, default="000")
phone = models.CharField(u'手机号码', max_length=25, default="000")
attimes = models.IntegerField(u'公益时长', default = 0)
intro = models.TextField(u'自我介绍', blank = True)
avatar = models.ImageField(u'头像', upload_to = "image/UserAvatar", blank = True)
```

models.OneToOneField()函数实现了对 user 的继承。

4. MVC

- MVC 简介

MVC 全名是 Model View Controller，是模型(model)－视图(view)－控制器(controller)的缩写，一种软件设计典范，用一种业务逻辑、数据、界面显示分离的方法组织代码，将业务逻辑聚集到一个部件里面，在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑。MVC 分层有助于管理复杂的应用程序，因为您可以在一个时间内专门关注一个方面。MVC 主要分为：

Model（模型）：是应用程序中用于处理应用程序数据逻辑的部分。常模型对象负责在数据库中存取数据。

View（视图）：是应用程序中处理数据显示的部分。通常视图是依据模型数据创建的。

Controller（控制器）：是应用程序中处理用户交互的部分。通常控制器负责从视图读取数据，控制用户输入，并向模型发送数据。

- MVC 实现说明

公益活动管理系统的编程设计采用了 python 的 web 框架 django，Django 是一个基于 MVC 构造的框架。但是在 Django 中，控制器接受用户输入的部分由框架自行处理，所以 Django 里更关注的是模型、模板和视图。在该系统中

- models.py 文件主要用 Python 类来描述数据表。称为模型。在该文件中定义了用户，活动，申请表三个类。运用这些类，可以进行创建、检索、更新、删除 数据库中的记录而无需写一条又一条的 SQL 语句。
- views.py 文件包含了页面的业务逻辑。
- urls.py 指出了什么样的 URL 调用什么的视图。

template 包含了 html 模板，它描述了页面是如何设计的。

六． 网站测试

1. 功能测试

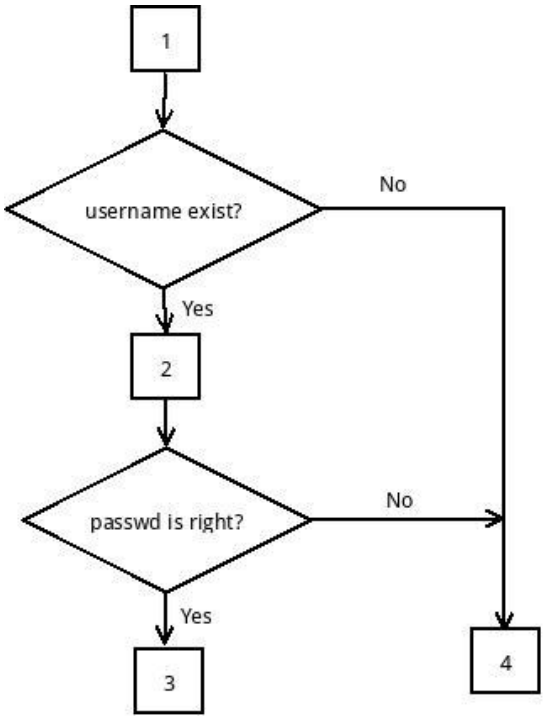
功能测试

由于一个 Web 程序由多种逻辑层组成——从 Http 层的 request 执行，到表单的确定和处理，再到模板的渲染，所以直接测试一个网站将会是一个比较复杂工作。但是， 通过利用 Django 测试框架以及齐全的配套工具，我们可以比较容易的模拟 request，插入测试数据，检查程序输出和一般情况下的测试。

在 Django 的测试框架中，测试有两种方式：Doctests 和 Unit Tests。本项目采用的是第二种方式，其具体的代码位于/philosopher/tests.py 文件中。以下谨描述本网站的用户登录以及用户注册两个用例的测试流程。

1. 用户登录

该用例的控制流图如下所示



测试用例：

- 用户名不存在
- 密码错误
- 正常登录

测试函数

基本环境设置，创建一个用户

```
12  class LoginTest(TestCase):
13          """docstring for LoginTest"""
14          def setUp(self):
15                  user = User.objects.create(username = "donald")
16                  user.set_password("123456")
17                  user.save()
```

- 用户名不存在

```
18          def test_login_fail_with_wrong_password(self):
19                  response = self.client.post('/accounts/login/', {"username": "donald", "password": "000000" })
20                  self.assertEqual(response.status_code, 200)
21                  self.assertContains(response, "username and password didn't match")
```

- 密码错误

```
22          def test_login_fail_with_wrong_username(self):
23                  response = self.client.post('/accounts/login/', {"username": "test", "password": "123456"})
24                  self.assertEqual(response.status_code, 200)
25                  self.assertContains(response, "username and password didn't match")
```

- 正常登录，由于登录成功会跳转到首页，所以返回码是 302

```
26          def  test_login_success(self):
27                  response = self.client.post('/accounts/login/', {"username": "donald", "password": "123456" })
28                  self.assertEqual(response.status_code, 302)
```

用例覆盖路径块分别为：

- {1, 4}

- {1, 2, 4}
- {1, 2, 3}

块覆盖率为：4/(4-0) * 100% = 100%

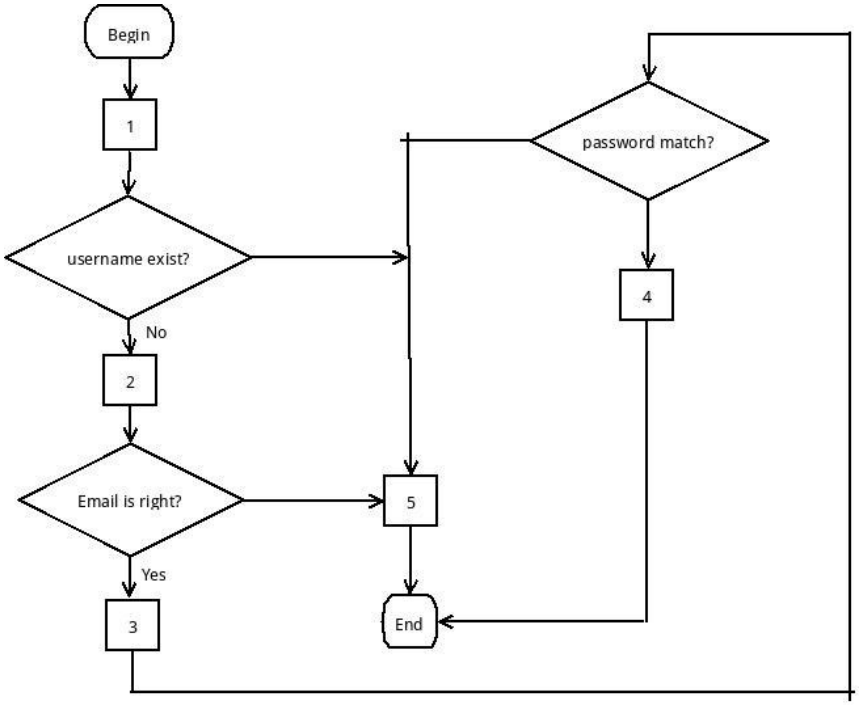
测试结果

```
donald@DonaldPC:~/Desktop/Philosopher$ python manage.py test philosopher
Creating test database for alias 'default'...
...
----------------------------------------------------------------------
Ran 3 tests in 0.273s

OK
Destroying test database for alias 'default'...
```

测试结果符合预期。

2. 用户注册

该用例的控制流图如下所示



测试用例：

- 用户名已存在
- 邮箱格式错误
- 两次输入密码不一致
- 正常注册

测试函数：

基本环境设置，创建一个用户

```
30  class SignUpTest(TestCase):
31          """docstring for SignUpTest"""
32          def setUp(self):
33                  user = User.objects.create(username = "donald")
34                  user.set_password("123456")
35                  user.save()
```

- 用户名已存在

```
36          def test_sign_up_with_exist(self):
37                  userinfo = {
38                          "user-username": "donald",
39                          "user-email": "123456@123.com",
40                          "user-password1": "123456",
41                          "user-password2": "123456",
42                          "userprofile-role": "s",
43                          "userprofile-gender": "f",
44                          "userprofile-schoolID": "1234",
45                          "userprofile-phone": "123456789",
46                  }
47                  response = self.client.post('/philosopher/accounts/signup/', userinfo)
48                  self.assertEqual(response.status_code, 200)
49                  self.assertContains(response, "A user with that username already exists.")
```

- 邮箱格式错误

```
50          def test_sign_up_with_wrong_email(self):
51                  userinfo = {
52                          "user-username": "test",
53                          "user-email": "wrongemail",
54                          "user-password1": "123456",
55                          "user-password2": "123456",
56                          "userprofile-role": "s",
57                          "userprofile-gender": "f",
58                          "userprofile-schoolID": "1234",
59                          "userprofile-phone": "123456789",
60                  }
61                  response = self.client.post('/philosopher/accounts/signup/', userinfo)
62                  self.assertEqual(response.status_code, 200)
63                  self.assertContains(response, "Enter a valid e-mail address.")
```

- 两次输入密码不一致

```
65          def test_sign_up_with_different_password(self):
66                  userinfo = {
67                          "user-username": "test",
68                          "user-email": "123456@123.com",
69                          "user-password1": "123456",
70                          "user-password2": "000000",
71                          "userprofile-role": "s",
72                          "userprofile-gender": "f",
73                          "userprofile-schoolID": "1234",
74                          "userprofile-phone": "123456789",
75                  }
76                  response = self.client.post('/philosopher/accounts/signup/', userinfo)
77                  self.assertEqual(response.status_code, 200)
78                  self.assertContains(response, "The two password fields didn&#39;t match.")
```

- 正常注册，由于注册成功后会自动登录并跳转到首页，所以返回码是 302

```
80          def test_sign_up_successfully(self):
81                  userinfo = {
82                          "user-username": "test",
83                          "user-email": "123456@123.com",
84                          "user-password1": "123456",
85                          "user-password2": "123456",
86                          "userprofile-role": "s",
87                          "userprofile-gender": "f",
88                          "userprofile-schoolID": "1234",
89                          "userprofile-phone": "123456789",
90                  }
91                  response = self.client.post('/philosopher/accounts/signup/', userinfo)
92                  self.assertEqual(response.status_code, 302)
```

用例覆盖路径块分别为：

- {1, 5}
- {1, 2, 5}
- {1, 2, 3, 4}
- {1, 2, 3, 5}

块覆盖率为：5/(5-0) * 100% = 100%

测试结果

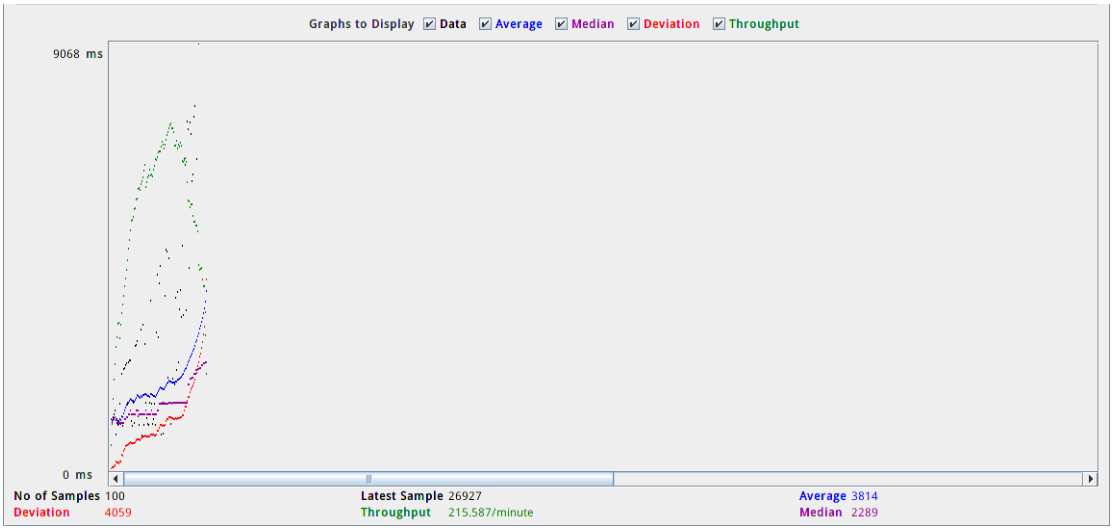
```
donald@DonaldPC:~/Desktop/Philosopher$ python manage.py test philosopher
Creating test database for alias 'default'...
....
----------------------------------------------------------------------
Ran 4 tests in 0.316s

OK
Destroying test database for alias 'default'...
```

测试结果符合预期。

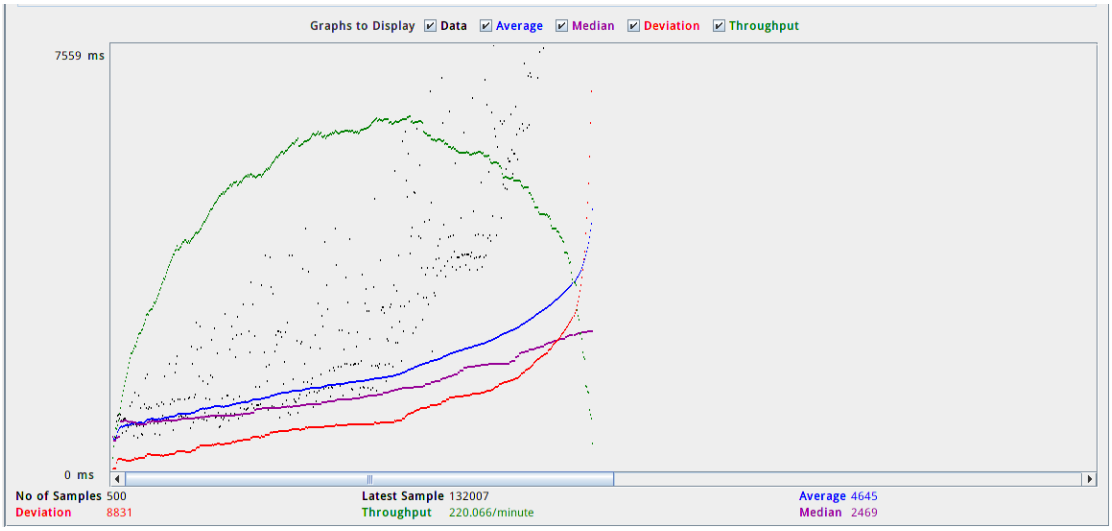
2. 负载和压力测试

本项目采用 JMeter 来对其进行负载/压力测试，创建的线程组的数目依次是 100，500，1000，1500，2000，2500，Ramp-Up period 为 5 秒，每个测试计划循环一次。测试计划存储在项目/JmeterTest/路径下，测试的结果及分析如下

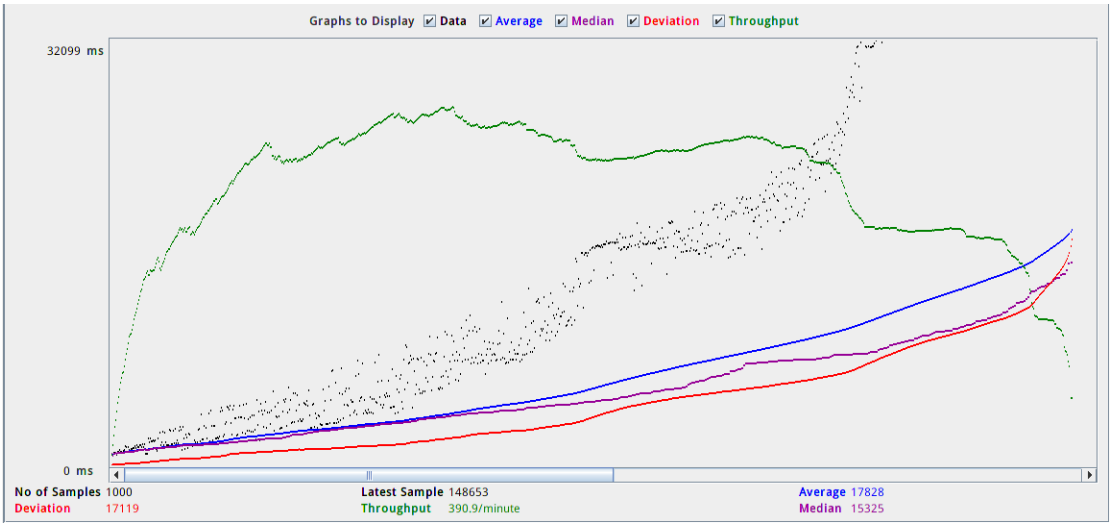
- 线程组为 100 的结果图



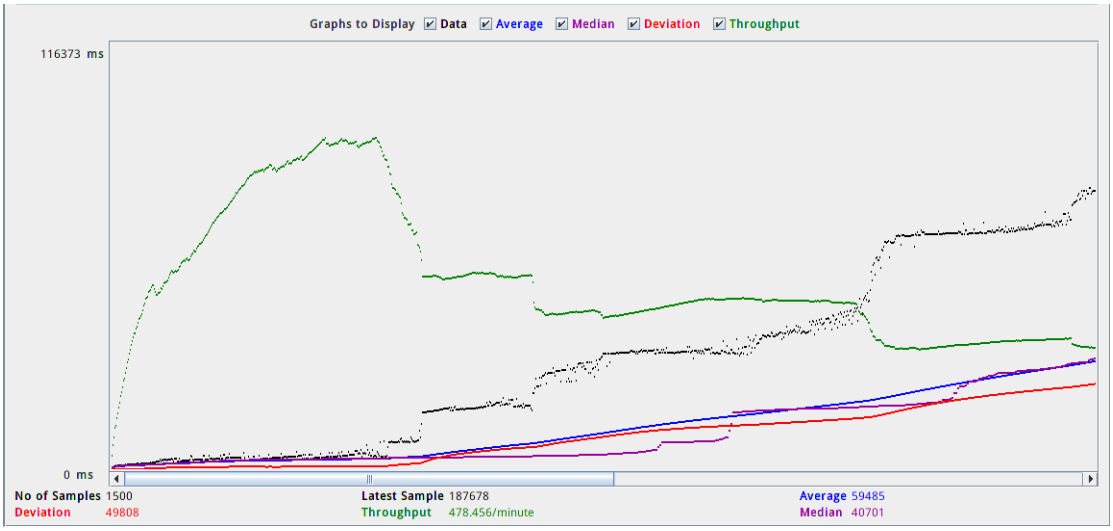
- 线程组为 500 的结果图



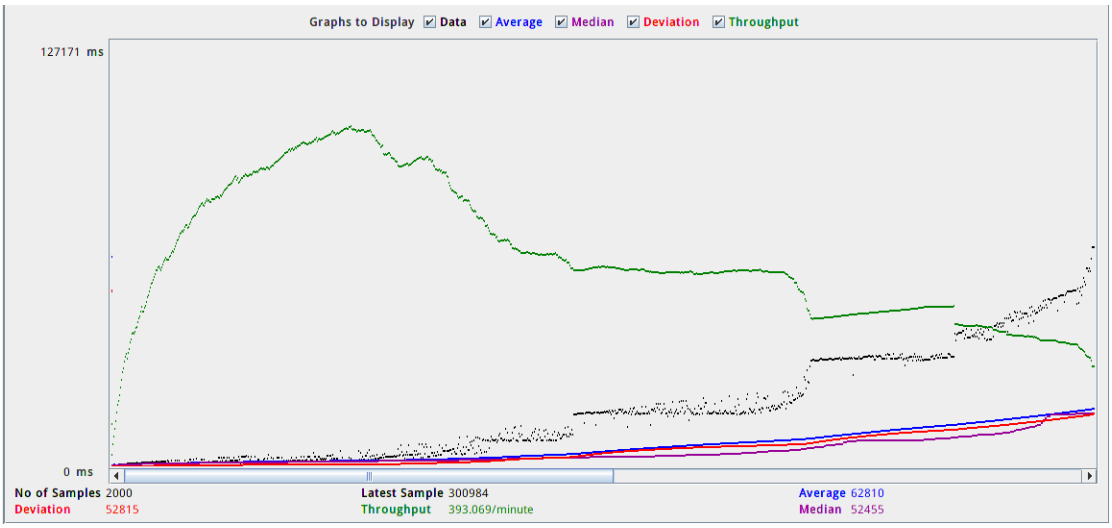
- 线程组为 1000 的结果图



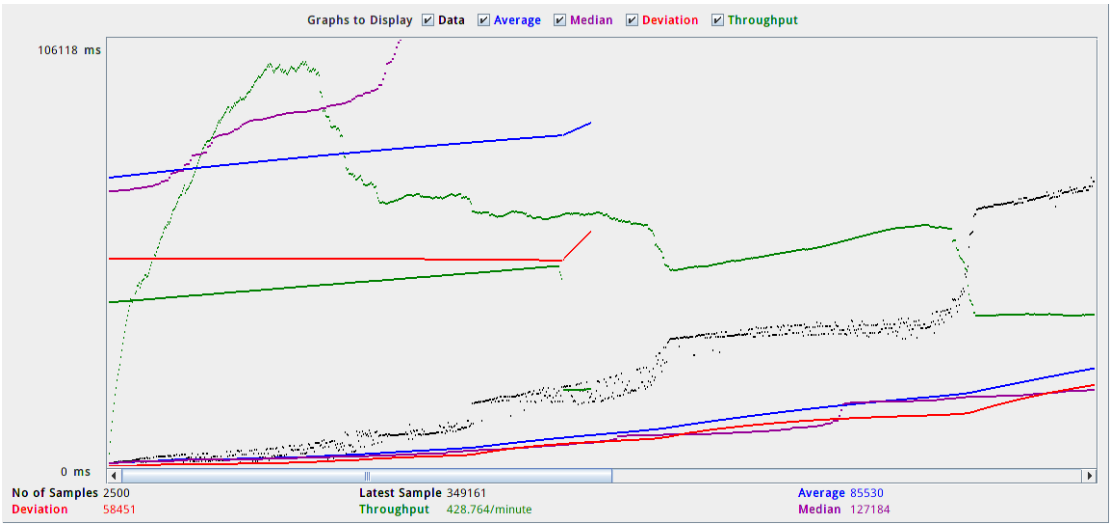
- 线程组为 1500 的结果图



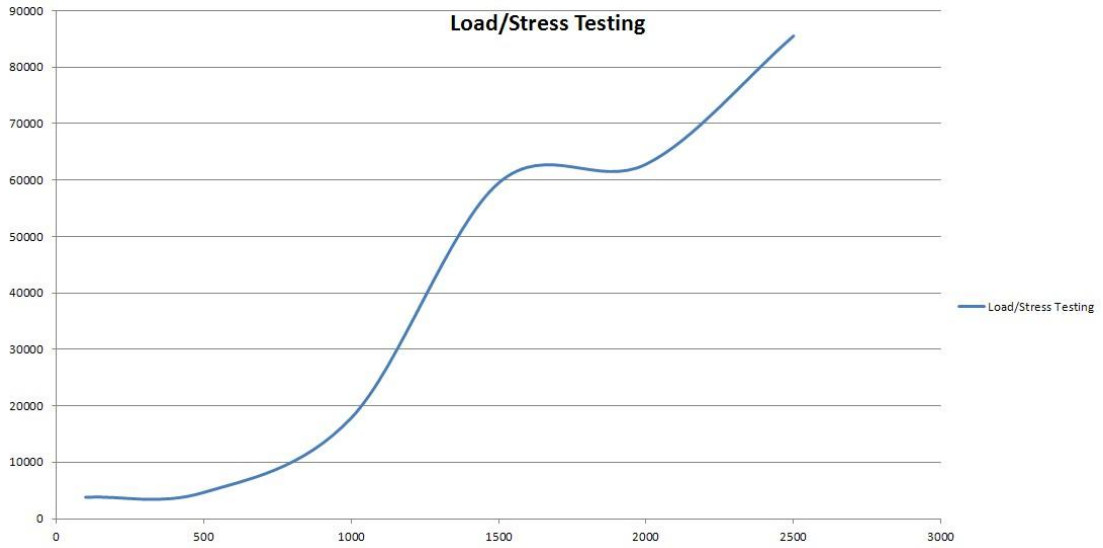
- 线程组为 2000 的结果图



- 线程组为 2500 的结果图



- X(线程数)-Y(平均访问时间/ms)的平滑线状图



分析：

从图形结果可知，吞吐量和平均时间一开始基本都是呈上升状态，后来又略有下降。而从整

体的线状图可以看出，访问的平均时间随着线程总数的增加而增加。当线程数达到 2500 时，平均时间增幅较大，其原因在于后面有近 100 个线程长期属于卡顿状态。可见，这时应该是达到了网站的最高的访问频率，所以网站同一时间能够承受的最高的访问量应该是 2500/5=500（Ramp-Up period 为 5 秒），基本满足预期目标。