本科毕业论文（设计）

题　　目：基于 Python 的在线评测系统的设计与开发

学　　院：数据科学与软件工程学院

专　　业：软件工程

姓　　名：李扬

指导教师：陈宇

2016 年 5 月 15 日

<table>
<tr><td>论文（设计）题目</td><td colspan="3">基于 Python 的在线评测系统的设计与开发</td></tr>
<tr><td>学院</td><td>数据科学与软件工程学院</td><td>专业</td><td>软件工程</td></tr>
<tr><td>学生姓名</td><td>李扬</td><td>学号</td><td>201240703104</td></tr>
<tr><td rowspan="2">选题来源</td><td colspan="3">教师科研课题（ ）；生产、工程或社会实践课题（ ）
学生自拟课题（ √ ）；师生共同拟定课题（ ）</td></tr>
<tr><td colspan="3">大学生创新创业训练项目（ ）；学科竞赛（ ）</td></tr>
<tr><td colspan="4">研究内容或设计方案：
OJ 系统的设计与开发</td></tr>
<tr><td colspan="4">研究方法与技术路线：
从 OJ 系统的设计入手，讨论了从系统架构设计到各个模块的核心内容，逐步的实现一个完整功能的 OJ。</td></tr>
<tr><td colspan="4">进度安排：
3.1 - 3.10 架构设计
3.10 - 4.10 前后端基本功能完成
4.10 - 4.20 判题沙箱设计与开发
4.20 - 5.1 细节优化与 bug 修复
5.1 - 5.20 撰写论文</td></tr>
<tr><td colspan="4">论文起止时间：
2016 年 3 月 1 日 - 2016 年 5 月 20 日</td></tr>
<tr><td colspan="2">学生（签名）：
指导老师（签名）：
2016 年 5 月 1 日</td><td colspan="2">教研室意见：）
教研室主任（签名）：
年 月 日</td></tr>
</table>

Design and Development of Online Judge System Based on Python

郑重声明

本人呈交的学位论文 (设计) 是在指导教师的指导下，独立进行研究工作所取得的成果，所有数据、图片资料真实可靠。除文中已经注明引用的内容外, 本学位论文 (设计) 的研究成果不包含他人享有著作权的内容。对本论文 (设计) 所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确的方式标明。本学位论文 (设计) 的知识产权归属于青岛大学。

本人签名:　　日期:

摘 要

Online Judge（在线评测系统，以下简称 OJ）起源于 ACM 国际大学生程序设计竞赛（ACM/ICPC），初衷是训练 ACM 选手。用户登录系统提交相关题目的源代码后，系统就会进行编译、运行和评测，然后返回结果。这对于计算机专业的算法、数据结构等课程来说也是一个训练学习的绝佳平台。

从 OJ 系统的需求分析入手，讨论了从系统架构设计到具体功能开发以及部分遇到的问题和挑战，逐步的实现一个完整功能的 OJ，包括基于 Python 和 Django 的 Web 程序开发，基于 avalon 的复杂单页面网页的构建，数据库和缓存的性能调优，基于 seccomp 的沙盒安全机制的设计，基于 Docker 的部署运维等。

关键词 在线评测系统 ACM 竞赛 沙箱 Python 单页面程序

Abstract

Online Judge (OJ) originated from ACM International Collegiate Programming Contest(ACM/ICPC), which is intended to train ACM players. After logging into the system, user can submit solutions of the problems, OJ system will compile and run the code and then return the result. Meanwhile, for courses like Algorithm and Data Structures, OJ is a great platform to learn and practice skills.

The paper will start with the requirement analysis of OJ system, and then we will discuss topics ranging from system architecture design, specific module development to issues and challenges we solved, including Web server development based on Python and Django, database and cache related performance optimization, complex single page application based on avalon, sandbox security mechanism based on seccomp.

Keywords OnlineJudge ACMContest Sandbox Python SinglePageApplication

目录

前言

本文从 OJ 系统的需求分析入手，讨论了从系统架构设计到具体功能开发以及部分遇到的问题和挑战，逐步的实现一个完整功能的 OJ，共分为六章，各章内容安排如下：

第一章：绪论。从国内外现有的 OJ 系统的发展情况入手，分析了目前 OJ 系统存在的缺陷，并提出了自己的改进意见。

第二章：系统架构与模块。本章纵览全局，介绍了 OJ 使用的平台和技术、系统架构与组成和主要的功能模块。

第三章：Web 后端实现。从后端整体架构到用户模块、题目模块、比赛模块、判题调度模块和 Virtual Judge 模块逐一介绍，还讨论了 OJ 系统安全性的话题。

第四章：Web 前端实现。OJ 系统中前端页面主要分为两部分，用户前台和 admin 后台，两部分应用场景有着明显的差异，需要根据实际情况来选择，这一章对这两部分使用的技术进行了分析。

第五章：判题沙箱的设计。判题沙箱是保证 OJ 系统安全的重要措施之一，本章分析了沙箱要实现的功能，并逐一实现，同时讨论了开发过程中遇到的部分沙箱绕过的问题。

第六章：系统测试与部署。介绍了 OJ 开发中借助 Docker 进行系统测试和部署的方法和经验。

第一章　绪论

1.1　背景

ACM/ICPC 是世界上公认的规模最大、水平最高的国际大学生程序设计竞赛项目，其目的旨在使大学生运用计算机来充分展示自分析问题和解决问题的能力。该竞赛一直受到国际各大知名大学和知名软件公司的的重视。

在 ACM/ICPC 集训队日常训练中，OJ 起到了非常重要的作用。集训队员可以在 OJ 上挑选各种题目进行训练，然后可以在 OJ 上举办比赛，培养参加 ACM 队员的团结合作能力。

1.1.1　国内外的 OJ

在 ACM/ICPC 几十年的发展过程中，出现过很多知名的 OJ，例如：

- 北京大学 Online Judge http://poj.org/
- 杭州电子科技大学 Online Jusge http://acm.hdu.edu.cn/
- UVaOJ https://uva.onlinejudge.org/

其中还有些学校将自己的 OJ 开源，供其他的学校免费试用，例如：

- 华中科技大学 Online Judge https://github.com/zhblue/hustoj
- DMOJ https://github.com/DMOJ/judge
- GoOnlineJudge https://github.com/ZJGSU-Open-Source/GoOnlineJudge

1.1.2　项目的意义

上文提到有很多大学开放的 OJ，但是我们在使用的时候还是遇到了一些问题，包括系统界面简陋、操作复杂、没有题目添加权限、举办比赛需要申请、申请流程繁琐等等。在进行开源 OJ 二次开发的时候也有很多问题，主要是软件文档不全、代码质量偏低、部分设计和架构上存在明显的问题。

比如某开源 OJ，使用 daemon 进程轮询数据库来获取新的提交，这样在已有大量数据记录的情况下可能会给数据库造成较大的压力，新提交的评测也无法立即进行评测，必须等到下一次轮询。同时判题沙箱部分和调度部分耦合，造成 Web 服务器和判题服务器分离困难。

还有的 OJ 存在安全漏洞，比如沙箱绕过、SQL 注入 [1]、XSS[2]、CSRF 和越权等，这些都可能威胁 OJ 的正常运行和数据安全。

所以有必要开发一个新的 OJ 系统，从根本上解决这个问题，为大家提供一个稳定可用的环境。同时本系统在设计上就兼顾了校内教学和考试平台，老师可以在上面进行日常的布置作业和考试等。

第二章　系统架构与模块

2.1 使用的平台和技术

- Linux

  Linux 是高性能的开源操作系统，在服务器环境上有着非常广泛的应用，很多开源的程序和平台都是基于 Linux 的。

  Linux 是 OJ 的的开发和部署平台，虽然 Python 可以跨平台，但是在 Windows 上兼容性并不好，同时因为很多系统 API 的不一致，部分运行库并没有兼容 Windows 或者 Windows 上运行效率比较低。

- Python 和 Django

  Python 是一种面向对象的脚本语言，语法简洁易读，具有丰富和强大的类库，而且和 C/C++ 语言进行混合编程非常方便。

  Django 是本项目使用的 Web 框架，使用 Python 语言编写的，是最著名的 Python Web 框架。使用 Django 可以让你比较小的代价来构建和维护高质量的 Web 应用。

- MySQL 和 Redis

  MySQL 是关系型数据库，是 OJ 中数据存储的主力。因为它开源、性能高等特点，已经成为最流行的数据库，广泛的用在互联网上各种网站上。

  Redis 是内存数据库，主要用于 Key-Value 类型数据的存储，存取速度非常快。担当队列和缓存的角色。

- Docker

  Docker 是一个开源的应用容器引擎，让开发者可以打包应用和依赖到一个可移植的容器中，然后发布到不同的机器上。OJ 用 Docker 来创建测试和部署环境，大大简化了部署过程。

- Bootstrap

  Bootstrap 是来自 Twitter 的前端框架，包括 HTML、CSS 和 JavaScript 框架，提供排版、表单、按钮、导航等各种组件。它简洁灵活，使得 Web 开发更加快捷。

- avalon

  avalon 是 OJ 使用的前端 MVVM 框架。在传统前端开发中，很多动态的效果需要直接进行 DOM 操作和拼接 HTML 操作，这是造成 JavaScript 代码无法维护的元凶之一，使用 MVVM 框架可以不再手动的去操作 DOM，简化了操作，而且可以提高性能。类似的框架还有 Vue.js 和 Angular JS。

- MVC 和 RESTful API

  MVC 是一种软件架构模式，分为三部分：Model、View 和 Controller。Django 就是一个 MVC 的框架，数据库定义在 Model 中，View 是业务逻辑，Controller 就是框架本身。Django 将不同的 URL 和 View 进行绑定，然后传递到 View 中指定的函数进行处理。

  前端 JavaScript 和后端的 Python 代码使用 RESTful API 进行通信，数据格式是 JSON。这样实现了复杂页面的前后端分离。

2.2 系统架构与组成

OJ 系统后端由 MySQL 数据库、Redis、异步队列、判题服务器、测试用例同步模块组成，其中，OJ 系统将用户提交的代码和判题信息存储在一个单独的 MySQL 数据库中，用户信息和题目等在另外一个数据库中。

多数情况下，系统只需要与 MySQL 数据库进行交互，进行增删改查，然后渲染前端模板显示，比较复杂的流程是判题流程。用户提交代码后，写入数据库，创建判题任务并交给异步队列去处理，立即返回前端提交 id，前端定时使用提交 id 来获取提交状态并提示用户。异步队列根据当前系统设置调度选择一台判题服务器，调用 RPC 接口发送用户代码、题目信息等数据，判题服务器编译运行用户代码并返回给 Web 服务器运行结果然后更新数据库。系统还可能进行的操作包括：更新题目计数器、更新用户做题状态、更新比赛排名、更新用户排名和部分缓存的主动失效等。

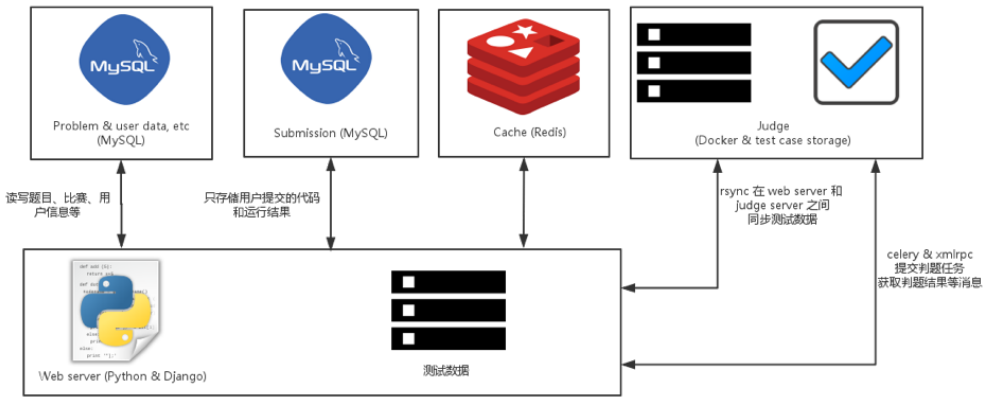

图 2.1: OJ 后端架构图

Web 前端使用了 Bootstrap 作为前端 UI 框架，使用了 avalon 作为 MVVM 框架。由于 JS 数量非常多，依赖关系也很复杂，所以还使用了 require.js 进行 JavaScript 模块化，使用了 r.js 进行静态文件的压缩和合并。

2.3 功能模块

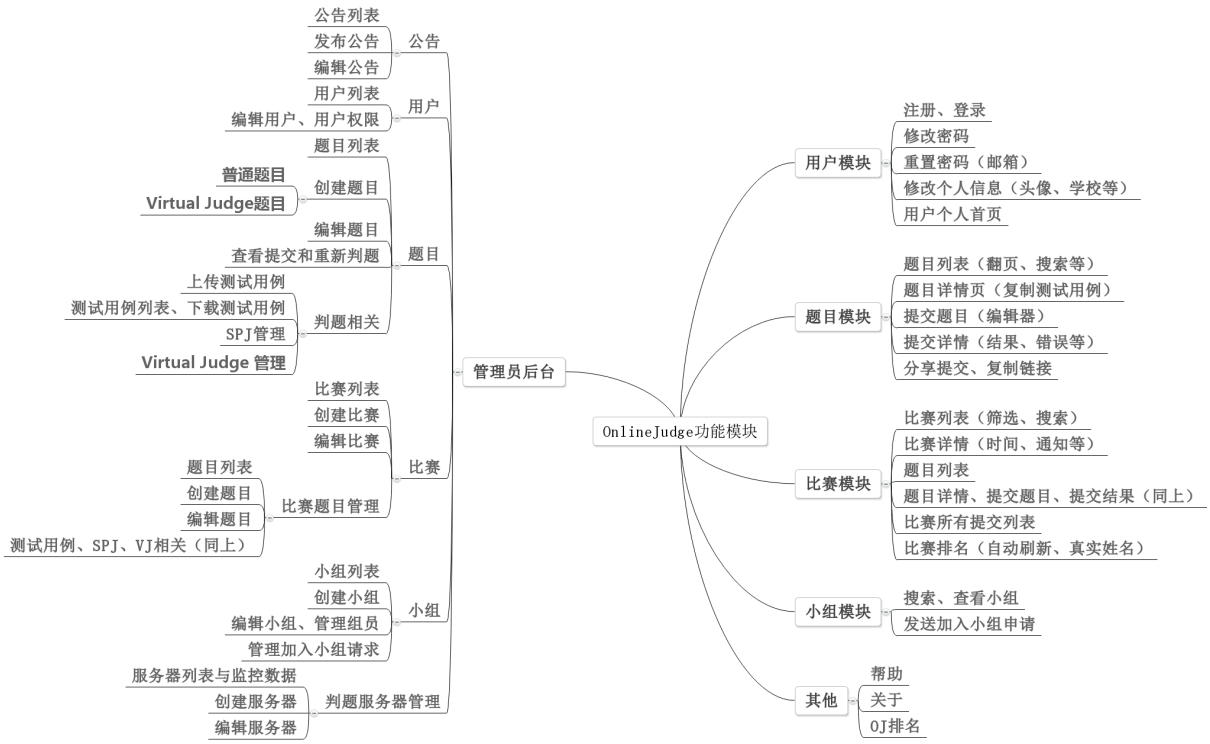

图 2.2: OJ 功能模块

第三章　Web 后端实现

OJ 的后端是基于 Python 和 Django 的，数据库使用 MySQL，缓存及队列使用 Redis 和 Celery。

3.1　Django 框架模块组成

3.1.1　Model 数据库模型

Model 是数据模型，是 Django ORM 的重要组成部分。使用 ORM 可以大大简化数据库操作，提高开发效率，同时避免 SQL 注入等安全问题的发生。一般情况下一个 Model 映射到数据库中的一张表中，Model 的每一个属性就是表中的每一个字段。比如说

```
class User(models.Model):
    username = models.CharField(max_length=30, unique=True)
    password = models.CharField(max_length=50)
    email = models.EmailField(max_length=254, blank=True, null=True)
```

就会生成如下所示创建表的 SQL 语句。

```
CREATE TABLE user(
    id INT PRIMARY KEY,
    username VARCHAR(30) UNIQUE NOT NULL,
    password VARCHAR(50) NOT NULL,
    email VARCHAR(254)
)
```

如果需要创建用户名为 foo 的用户，可以使用 `User.objects.create(username="foo", password="hidden", email="foo@gmail.com")`。如果需要查询用户名 foo 的用户，可以使用 `user = User.objects.get(username="foo")`，Django 就会自动的将其转换为 SQL 语句 `SELECT id, username, password, email FROM user WHERE username="foo";`，然后就可以访问 `user` 对象的属性来获取具体字段的值。

如果需要修改用户信息，可以直接给属性赋值，然后调用 `saeve` 方法即可。比如

```
user.emai="foo@example.com"
user.save()
```

Django 就可以生成对应的 SQL update 语句。

3.1.2 View 视图处理函数

View 视图处理函数就是业务逻辑所在了，负责处理各种请求、进行数据校验、与数据库进行交互和调用第三方 API 等。

Django 的 View 处理函数都是与 URL 绑定的，符合该 URL 规则的请求就会进入对应的 View 函数处理。比如

```
url(r'^problem/(?P<problem_id>\d+)/$', "problem.views.problem_page")


def problem_page(request, problem_id):
    try:
        problem = Problem.objects.get(id=problem_id, visible=True)
    except Problem.DoesNotExist:
        return error_page(request, u"题目不存在")
    return render(request, "oj/problem/problem.html", {"problem": problem})
```

View 视图处理函数一般接收 `request` 和部分 URL 中匹配的值为参数，需要返回一个 `HTTPResponse` 对象。

3.2 部分模块功能实现

下面将以部分逻辑比较复杂的模块为例，详细分析 OJ 后端的开发过程。

3.2.1 用户登录

用户登录是几乎所有网站必备的基础功能。用户需要在页面上提交自己的用户名和密码，然后后端进行验证，如果验证通过，就设置 Cookies，否则提示用户名或密码错误。

```
class UserLoginAPIView(APIView):
    def post(self, request):
        """
        用户登录json api接口
        ---
        request_serializer: UserLoginSerializer
        """
        serializer = UserLoginSerializer(data=request.data)
        if serializer.is_valid():
            data = serializer.data
        user = auth.authenticate(username=data["username"], password=data["password"])
```

```
            # 用户名或密码错误的话 返回None
            if user:
                auth.login(request, user)
                return success_response(u"登录成功")
            else:
                return error_response(u"用户名或密码错误")
        else:
            return serializer_invalid_response(serializer)
```

这里使用了 Django Rest Framework 的 APIView，可以简化通过 AJAX 和 API 通信的 Web 程序的开发，这里只实现了 POST 函数，如果不是 POST 方法请求 API 就会收到错误响应。

serializer 是用来校验数据的，写法和 Model 类似，需要规定字段和对应的数据类型等，UserLoginSerializer 的实现代码是

```
class UserLoginSerializer(serializers.Serializer):
    username = serializers.CharField(max_length=30)
    password = serializers.CharField(max_length=30)
```

如果缺少字段或者字段的数据不符合要求,就会返回一个 serializer_invalid_response(serializ 的响应，这是一个封装后的函数，实现代码是

```
def serializer_invalid_response(serializer):
    for k, v in serializer.errors.iteritems():
        return error_response(k + " : " + v[0])
```

通过 serializer.errors 属性可以得到错误详情，然后返回前端，提示用户。

如果数据校验通过,就会使用用户名和密码作为参数,调用 Django 的 auth.authenticate 方法进行验证。auth.authenticate 实现也比较简单，首先根据配置文件选择密码的哈希函数计算密码的哈希值，然后使用用户名和哈希值去数据库查询，如果查询到记录，说明用户名密码正确，返回对应的 User 对象。

因为 HTTP 协议是一个无状态的协议，服务器端需要通过 Cookies 来将请求和登录的用户进行对应，这个对应关系是保存在服务器端的，有时也称为 Session，这个过程就是 auth.login(request, user) 实现的。

Django 生成的数据库中有 django_session 表,表中有三个字段,session_key、session_data 和 expire_date。首先 Django 会在这张表中插入一条记录，简化的 demo 如下。

```
DjangoSession.objects.create(session_key=random_string(), session_data=json.dumps({"user
```

然后使用 `set-cookie` HTTP 头给浏览器设置一个 Cookie，比如 `set-cookie:sessionid=9400de64wpd` `expires=Wed, 18-May-2016 13:13:34 GMT; httponly; Max-Age=1209600; Path=/`，key 是 `sessionid`，value 是数据库里的 `session_key`，还包括过期时间等信息。

这样每次用户请求的时候，取到 Cookies 中 `sessionid` 的值，再去表中查询就可以得到当前用户的 ID 了。

3.2.2 重置密码

重置密码功能的设计关乎用户用户账户的安全，已经有很多案例说明重置密码功能设计缺陷带来的安全问题 [3]。

用户提交重置密码请求后，首先会查询邮箱是否存在，如果存在，就会生成 32 位长度的随机 token 和 30 分钟的过期时间。然后调用异步队列，向该邮箱发送重置密码链接，响应前端重置邮件已发出。

用户收到邮件之后，点击重置链接，后端会先验证该链接的有效性，包括是否存在和是否过期，如果验证通过就会返回重置密码页面。用户填写新密码提交后，会再次验证 token 的有效性，如果验证通过，就会查询该 token 对应的用户，然后更新密码。

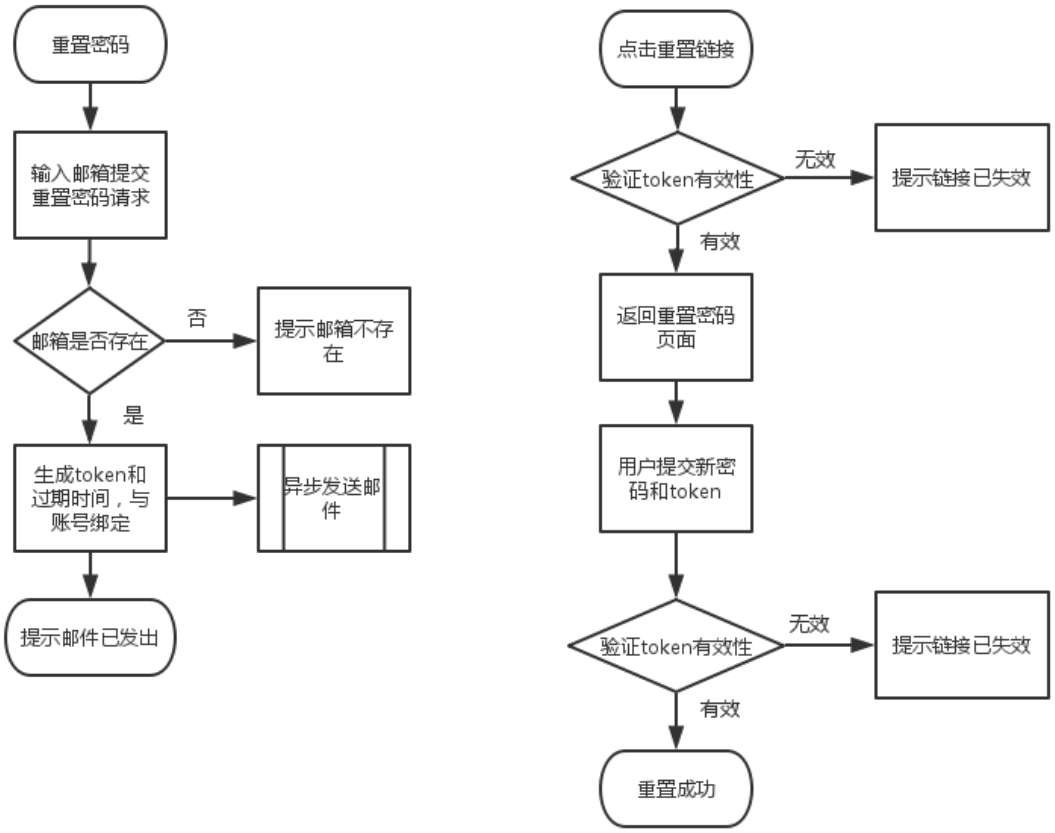

图 3.1: 密码重置流程

3.2.3 比赛排名

比赛排名的 Model 是

```
class ContestRank(models.Model):
    user = models.ForeignKey(User)
    contest = models.ForeignKey(Contest)
    total_submission_number = models.IntegerField(default=0)
    total_ac_number = models.IntegerField(default=0)
    total_time = models.IntegerField(default=0)
    submission_info = JSONField(default={})
```

可以看到，记录了具体题目提交信息的字段保存的是 JSON 数据，格式是 `{$problem_id: {"is_ac": True/False, "ac_time": $ac_time, "error_number":$error_number, "is_first_ac": True/False}}` 这样可以简化数据库设计的难度，让 SQL 数据库按照 NoSQL 来用。这样每一个用户就是一条记录，也就是排名上的一行。只要根据 `contest_id` 查询一次数据库后，就可以得到生成页面的全部数据。

鉴于比赛参与人数可能很多，排名数据偏大，如果每次重新生成，可能会遇到性能问题，我们在这里使用了 Redis 作为缓存，在 HUSTOJ 等系统中也有类似的用法 [4]。使用 `${contest_id}_rank_cache` 作为缓存的 key。每次生成页面的时候，先使用缓存 key 去 Redis 中取数据，如果没有取到，就去 MySQL 中查询，同时生成 Redis 缓存。每次有新的提交的时候，异步队列需要主动的清除 Redis 缓存，避免页面数据过期。

3.3 判题模块

3.3.1 选择判题服务器

在 OJ 使用人数较多的情况下，需要使用多台服务器进行判题。选择哪一台服务器，怎么均衡服务器负载，怎么与服务器通信、怎么得到服务器判题结果就是我们要面对的问题了。

管理员可以在 admin 后台配置多台判题服务器，每台服务器都有自己的最大判题实例数量 `max_instance_number`、当前使用的判题实例数量 `used_instance_number` 和负载 `workload` 三个指标，每次分配给该服务器一个判题任务，当前已经使用的实例数加一，服务器的负载 workload 就会修改为 $\frac{used\_instance\_number}{max\_instance\_number} \times 100\ \%$，判题结束的时候，当前已经使用的实例数减一，负载值也对应修改。分配给该服务器同时运行的任务不会超过该服务器的最大判题实例的值。

3.3.2 RPC 通信

XML-RPC 是一个远程过程调用（remote procedure call，RPC) 的分布式计算协议，通过 XML 将调用远程机器上的函数封装，一般使用 HTTP 协议作为通信协议。

在远程机器上启动 XML-RPC Server，注册 `JudgeInstanceRunner` 类，供远程调用。

```
server = AsyncXMLRPCServer(('0.0.0.0', 8080), SimpleXMLRPCRequestHandler)
server.register_instance(JudgeInstanceRunner())
server.serve_forever()
```

由上一部分选择好服务器后，指定服务器的 IP 和端口就可以运行代码了。

```
s = TimeoutServerProxy("http://" + judge_server.ip +
                               ":" + str(judge_server.port), timeout=30)
data = s.run(judge_server.token, self.submission.id, self.submission.language,
             self.submission.code, self.time_limit, self.memory_limit,
             self.test_case_id, self.spj, self.spj_language,
             self.spj_code, self.spj_version)
```

这样在判题服务器上就会按照参数配置，编译运行相关的代码，并返回判题结果。

3.3.3 判题结束后的各种动作

服务器判题结束后，还有一系列的动作需要执行，包括：

- 更新题目提交数量和 AC 数量计数器
- 更新用户做题状态计数器，用于题目前面显示对应的符号
- 更新用户做题数量和 AC 数量计数器
- 比赛排名缓存的刷新

3.4 Virtual Judge 的实现

3.4.1 简介

Virtual Judge 是一种特殊的 OJ 系统。与其他 OJ 系统不同的是，Virtual Judge 系统本身并没有任何测试数据，而是通过在其他 OJ 系统中的机器人账号进行测试并抓取测试结果。因此可以在只有题目而没有测试数据的前提下建立竞赛。[5]

3.4.2 插件式爬虫

对于每个 OJ，具体的处理方法和过程都是不同的，但是最终要实现的功能都是统一的，比如登录用户、获取指定 URL 的题目、提交题目和获取指定的提交结果。在这种情况下，我们可以让每个 OJ 的爬虫都继承一个父类，需要实现父类中所有的公开方法，如果子类没有实现该方法，调用的时候就会引发 `NotImplementedError` 异常。

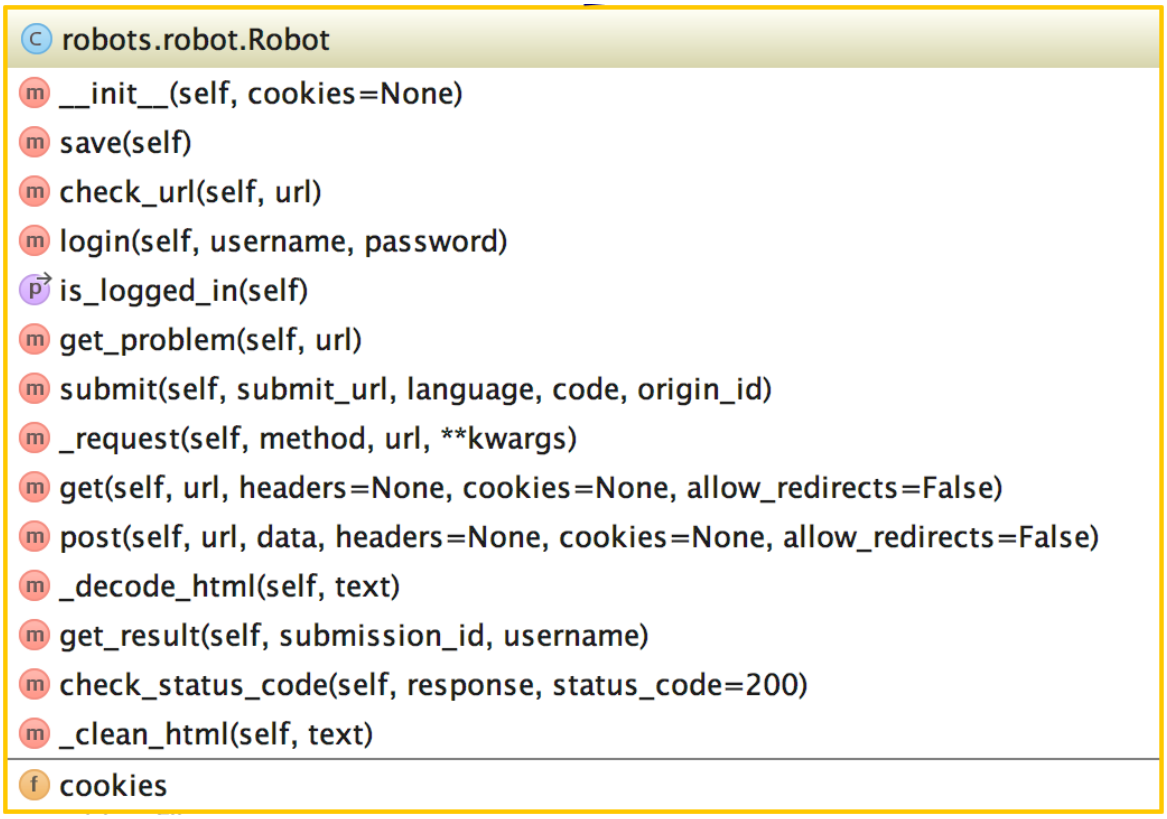

图 3.2: 爬虫父类的 UML

部分函数的作用：

- `save`：返回登陆后 Cookies 和 token 信息
- `check_url`：检查是否是本 OJ 题目的 URL
- `login`：使用给定的用户名密码登录 OJ
- `get_problem`：获取指定 URL 的题目信息
- `submit`：提交题目
- `get` 和 `post`：HTTP 协议请求网络
- `get_result`：获取提交结果

3.4.3 工作流程

以获取杭州电子科技大学 OJ 题目详情为例，具体实现的逻辑是：

```
def get_problem(self, url):
    if not self.check_url(url):
        raise RequestFailed("Invaild Hduoj url")
```

```
    regex = {"title": r"<h1 style='color:#1A5CC8'>(.*)</h1>",
             "time_limit": r"Time Limit:\s*[\d]*/([\d]*)\s*MS",
             "memory_limit": r"Memory Limit:\s*[\d]*/([\d]*)\s*K",
         "description": r"Problem Description</div>\s*<div class=panel_content>([\s\S]*?)</
         "input_description": r"Input</div>\s*<div class=panel_content>([\s\S]*?)</div>",
         "output_description": r"Output</div>\s*<div class=panel_content>([\s\S]*?)</div>",
         "hint": r"Hint(?:[\s\S]*?Hint[\s\S]*?</i>|</i>\s*</div>)([\s\S]*?)</div>",
             "spj": r"<font color=red>Special Judge</font>",
         "samples": r'Courier New,Courier,monospace;">([\s\S]*?)(?:<div|</div>)'}
    problem_id = re.compile(r"\d{4}").search(url).group()
    data = self._regex_page(url, regex)
    data["id"] = problem_id
    data["submit_url"] = "http://acm.hdu.edu.cn/submit.php?action=submit"
    return data
```

以提交题目并获取提交结果为例，分析爬虫提交题目的的工作逻辑。此步骤涉及到多个异步任务。

首先根据请求数据中的题目 id 获取给题目对应的 OJ 的具体信息, 在 VJ 数据库中创建该提交的记录，查询当前是否有空闲的爬虫用户用于提交。

如果没有空闲的爬虫用户，该提交就会被放入一个等待队列中，返回提交信息，否则将会修改该用户的标志位，设置为被占用。

然后会根据 OJ 的设置和该用户的登录信息实例化一个对应的爬虫，传递所需参数，创建 `submit_dispatcher` 异步任务，得到 `task_id`，更新到数据库，返回调用者提交信息。这时，爬虫在后端异步的运行。

而 `submit_dispatcher` 异步任务会去创建真正的提交异步任务 `submit`，同时还会设置该异步任务的回调函数为 `submit_waiting_submission` 和 `update_submission`，然后将该 `task_id` 更新到数据库中。

`submit` 方法会调用爬虫的 `submit` 方法去提交代码，接着循环请求判题结果，直到得到结果或者超过最多尝试次数，最终返回结果。

等到 `submit` 执行完毕的时候，两个回调函数会被调用，功能分别是提交等待队列中的题目和更新该次提交的信息。如果等待队列中没有提交，就会释放该爬虫用户，否则直接使用该用户继续提交题目，回到本流程开头。

3.5 安全问题

3.5.1 用户密码存储

现代网站已经不再使用明文密码存储的方式了，而是改用只在数据库中存储明文密码的哈希。常见的哈希算法有 MD5 和 SHA，可以从任何一段字符串计算出一段固定长度的哈希值。这样当用户输入密码时，直接将该密码代入算法得出哈希值，再与存储的哈希值对比，相同则允许登录。用户注册时也是直接存储密码的哈希值，而不是明文密码。这样就防止数据库泄露的时候导致用户明文密码泄露。

但是随着计算能力大幅提高，MD5 和 SHA1 算法已经不再推荐使用了，这些算法被碰撞的几率已经越来越大了。本系统中存储密码使用的 Django 默认哈希方法，SHA256 加 salt 后使用 PBKDF2 算法处理，该算法计算消耗的 CPU 时间非常多，能大大减慢暴力破解速度。

3.5.2 XSS

XSS 漏洞跨站脚本攻击，是 Web 程序中常见的漏洞。其原理是攻击者向有 XSS 漏洞的网站中传入恶意的 HTML 和 JavaScript 代码，当其它用户浏览该网站时，JavaScript 代码会自动执行，从而达到攻击的目的。比如盗取用户 Cookie、重定向到其它网站等。

Django 模板默认对模板输出全部转义，避免了绝大多数的 XSS 问题。但是有一些输出是不能转义的，比如公告的内容、题目的说明等，它们是使用富文本编辑器进行编辑的，如果进行过滤，文字的样式就无法显示了。所以我们在这里需要过滤危险的 JavaScript 和 HTML 属性等，使用了 https://github.com/phith0n/python-xss-filter 作为过滤器。

3.5.3 SQL 注入

SQL 注入攻击，是 Web 开发中最常见的一种安全漏洞。可以用它来从数据库获取敏感信息，甚至有可能获取数据库乃至系统用户最高权限。而造成 SQL 注入的原因是因为程序没有有效过滤用户的输入，使攻击者成功的向服务器提交恶意的 SQL 查询代码，程序在接收后错误的将攻击者的输入作为查询语句的一部分执行，导致原始的查询逻辑被改变，额外的执行了攻击者的代码。

OJ 系统全部使用 Django 的 ORM 进行数据库查询，而 ORM 在进行查询的时候使用了 SQL 参数绑定，所以不存在 SQL 注入的风险。

3.5.4 CSRF

CSRF 可以伪造受信任用户的请求来利用受信任的网站。攻击者可以在第三方网站 hacker.com 设置代码，用户在浏览这些网页的时候向 victim.com 跨域发送数据，浏览器根据同源策略，带上了 victim.com 的 cookies，导致请求被伪造，可能造成很大的危害，比如发送微博、添加管理员、修改密码等。

OJ 系统采取的措施是需要修改服务器状态的请求全部增加 token 验证, token 是在 Cookies 中读取的，而根据同源策略，hacker.com 是无法读取 victim.com 的 Cookies 的，这样就无法伪造 token。后端服务器只要对比 Cookies 中的 token 和请求头中的 token 是否一致就可以了。

3.5.5 沙箱安全

沙箱安全是整体开发中的重点和难点，本文将在第 5 章中详细分析。

3.5.6 其他安全问题

上面提到的几个安全问题是出现频率比较高的，危害比较大的。还有一些其他的安全问题没有单独列出，比如密码暴力破解、恶意提交请求等。

针对密码暴力破解，增加了管理员密码的两步验证。管理员在登录的时候，必须同时输入手机动态密码 app 上显示的验证码。针对恶意注册，增加了验证码。

针对使用脚本进行大量恶意提交的问题，使用 Token Bucket 机制增加了频率限制，同时为一定的并发请求预留了空间。当用户触发频率限制的时候，将会得到提交失败的提示和需要等待的时间。

第四章　Web 前端实现

OJ 系统中前端页面主要分为两部分，用户前台和 admin 后台，两部分应用场景有着明显的差异。用户前台，逻辑简单，主要以展示为主，需要兼顾搜索引擎的抓取；admin 后台，界面逻辑非常复杂，有大量的列表页、动态验证项目、动态变化的表单，不需要考虑搜索引擎的抓取问题。需要进行实际的分析，选择不同的开发技术。

4.1　admin 后台的 SPA 页面实现

4.1.1　JavaScript 模块化

页面功能的复杂化往往伴随的是 JavaScript 代码的数量的爆炸，传统的使用 `script` 标签直接加载的方案会遇到很多问题，比如无法确定模块依赖顺序、文件合并复杂等。由于历史原因，JavaScript 并未提供一种原生的、语言级别的模块化组织模式，而是将模块化的方法交由开发者来实现。因此，出现了很多种 JavaScript 模块化的实现方式，比如，CommonJS Modules、AMD 等。我们使用了 AMD 的模块化方案，由 require.js 充当加载器。

```
define("bsAlert", ["jquery", "bootstrap"], function($){
     function bsAlert(content){
         // TODO
     }
    return bsAlert;
});
```

`bsAlert` 是模块的名字，`jquery` 和 `bootstrap` 是模块的依赖，加载 `bsAlert` 模块的时候，require.js 会自动去加载它的依赖模块。

使用 `require` 函数引用这个模块，获得函数引用。

```
require(["bsAlert"], function (bsAlert) {
    bsAlert("foobar");
})
```

同时，由于已经确定模块间依赖关系，r.js 可以将模块级别和页面级别的 JavaScript 合并成一个文件，这样就减少了页面上的网络请求次数，提高了性能和用户体验。以 admin 后台添加题目为例：

表 4.1: 合并 JavaScript 前后网络请求统计

是否合并	网络请求次数	网络请求数据量
否	43	1.3M
是	15	844K

4.1.2 MVVM 框架

在传统前端开发中，大量的动态效果需要直接进行 DOM 操作和拼接 HTML 操作，这是造成 JavaScript 代码无法维护的元凶之一，使用 MVVM 框架可以不再手动操作 DOM，简化了操作，而且可以提高性能。OJ 系统使用的是 avalon.js 框架，类似的框架还有 Vue.js 和 Angular JS。

所有前端代码彻底分成两部分，视图的处理通过绑定实现，业务逻辑则集中在 VM 对象中处理。我们只要操作 VM 的数据，它就自动的同步到视图。demo 如下：

```
<script>
var vm = avalon.define({
    $id: "demo",
    name: "foobar"
})
</script>

<body ms-controller="demo">
    <input ms-duplex="name">
    <p>Hello,{{name}}!</p>
</body>
```

这样页面加载的时候，文本框和”hello” 后面中就会显示”foobar”，然后随着用户修改文本框中的文字，页面上”hello” 后面的文字也会自动改变。

4.1.3 Web 组件

前端开发的时候经常会遇到一些重复的功能和逻辑，比如翻页模块、上传模块、富文本编辑器等，将底层的 UI 组件与自己的业务逻辑联系在一起然后复用才能提高开发效率，增强代码的可维护性。admin 后台有多个封装的 Web 组件，以翻页 pager 组件为例。

```
avalon.component("ms:pager", {
    $template: "页数: {{ currentPage }}/{{ totalPage }} " +
    "<button ms-class=\"{{ currentPage==1?'btn disabled':'btn' }}
    \"ms-click=\"_getPrevPage\">上一页</button> " +
    " <button ms-class=\"{{ currentPage==totalPage?'btn disabled':'btn' }}
    \" ms-click=\"_getNextPage\">下一页</button>",
    currentPage: 1,
    totalPage: 1,
    _getPrevPage: _interface,
```

```
    _getNextPage: _interface,
    $init: function (vm, el) {
        vm._getPrevPage = function () {
            if (vm.currentPage > 1) {
                vm.currentPage--;
                vm.getPage(vm.currentPage);
            }
        };
        vm._getNextPage = function () {
            if (vm.currentPage < vm.totalPage) {
                vm.currentPage++;
                vm.getPage(vm.currentPage);
            }
        };
    }
```

在页面上只要使用

```
<ms:pager \$id="userPager" config="pager"></ms:pager>
```

就可以得到有完整功能的翻页按钮，不需要每个页面都去重复代码，而且不需要关心内部实现。

4.1.4 前后端分离

我们在传统前端开发过程中有很多痛点：

- 前端开发依赖后端，一般需要后端功能完成后，由后端向前端模板传递变量完成渲染，才能进行开发和测试，而且前端可能需要搭建后端开发环境。或者前端先完成一个 demo 页面，然后由后端工程师转换为实际的框架模板。
- 前端模板中有大量的后端语言逻辑，导致模板维护性变差。
- 前端模板数据更新需要刷新页面，而整个页面的完全加载是对带宽和性能的浪费。

为了提升开发效率，前后端分离的需求越来越被重视，后端负责业务/数据接口，前端负责展现/交互逻辑。

4.1.5 admin 后台整体框架

由于 admin 后台是一个 SPA 页面，页面无刷新，URL 不会变化，所以需要使用 URL 中的 hash 存储页面路由信息。比如`/admin/#announcement/announcement`、`/admin/#user/user_list` 等，JavaScript 监听 hash 变化事件，修改 `vm.templateUrl`，加载不同的 HTML 模板。

```
<div ms-include-src="templateUrl" data-include-replace="true"></div>
```

图 4.1: admin 首页

4.1.6 用户管理页面的实现

本节将以后台用户管理页面为例，解释 admin 模块前端开发的流程。

用户管理页面主要分为用户翻页列表和用户编辑两部分。对于用户列表，在 VM 中存储一个数组，然后在前端模板进行循环。

```
var vm = avalon.define({
    $id: "userList",
    userList: []
}
```

HTML 模板的部分代码

```
<tr ms-repeat="userList">
    <td>{{ el.id }}</td>
    <td>{{ el.username }}</td>
    <td>{{ el.create_time|date("yyyy-MM-dd HH:mm:ss")}}</td>
    <td>{{ el.real_name }}</td>
    <td>{{ el.email }}</td>
</tr>
```

页面加载的时候，JavaScript 通过 ajax 去后端查询分页用户列表，然后赋值给 vm.userList，这时 HTML 模板中就会自动显示为列表项目。

```
function getPage(page) {
    var url = "/api/admin/user/?paging=true&page=" + page + "&page_size=10";
    $.ajax({
        beforeSend: csrfTokenHeader,
        url: url,
        dataType: "json",
        method: "get",
        success: function (data) {
            if (!data.code) {
                vm.userList = data.data.results;
                avalon.vmodels.userPager.totalPage = data.data.total_page;
            }
            else {
                bsAlert(data.data);
            }
        }
    });
}
```

图 4.2: admin 用户列表

编辑用户的时候，需要 VM 中增加几个同步变量，比如 name、email 等，然后与 input 双向绑定。以用户名为例。

```
<div class="form-group col-md-4"><label>用户名</label>
    <input type="text" class="form-control" ms-duplex="username"
```

```
          data-minlength="3" data-minlength-error="用户名不得少于3位" required>
    <div class="help-block with-errors"></div>
</div>
```

vm.username 的修改将导致页面显示的修改，用户输入用户名将导致 vm.username 的修改。

最后通过 AJAX 请求后端更新用户信息

```
$("#edit-user-form").validator()
.on('submit', function (e) {
    if (!e.isDefaultPrevented()) {
        var data = {
            username: vm.username,
            email: vm.email,
            // 省略部分属性
        };
        $.ajax({
            url: "/api/admin/user/",
            data: data,
            dataType: "json",
            method: "put",
            success: function (data) {
                if (!data.code) {
                    bsAlert("编辑成功! ");
                    getPage(1);
                } else {
                    bsAlert(data.data);
                }
            }
        });
        return false;
    }
});
```

图 4.3: admin 编辑用户

在更加复杂的页面上也是类似的思路，只是使用了更多的绑定、Web 组件和更加复杂的业务逻辑和动态效果。

4.2 用户前台的后端渲染页面

和前后端分离然后前端渲染页面相比较，后端渲染要简单的多，也是最传统的做法。因为 Django 是 MVC 的架构，我们 View 中的逻辑去 Model 中查询，然后传递给模板就可以了。

以题目详情页为例：

```
def problem_page(request, problem_id):
    """
    前台题目详情页
    """
    try:
        problem = Problem.objects.get(id=problem_id, visible=True)
    except Problem.DoesNotExist:
        return error_page(request, u"题目不存在")
    return render(request, "oj/problem/problem.html", {"problem": problem})
```

部分模板的代码

```
<div class="problem-section">
    <label class="problem-label">描述</label>
    <div class="problem-detail">{{ problem.description|safe }}</div>
</div>
<div class="problem-section">
    <label class="problem-label">输入</label>
    <p class="problem-detail">{{ problem.input_description }}</p>
```

```
</div>
{% endfor %}
```

第五章　判题沙箱的设计

5.1　总体分析

判题沙箱是保证 OJ 系统安全的重要措施之一，如果没有沙箱，直接在服务器上运行用户的代码是存在严重安全风险 [6] 的，而且也无法去控制或获取代码的运行时间、占用内存大小和运行结果等运行数据。

沙箱要实现的功能：

- 获取进程运行实际时间和 CPU 时间，超时则结束进程
- 获取代码运行占用的内存，超内存则结束进程
- 获取代码运行结果和返回值
- 阻止代码执行一切危险的动作

5.1.1　进程实际运行时间和 CPU 时间

一般情况下，一个进程实际运行时间与占用的 CPU 时间并不相等。在单核处理器上，任何时刻只有一个进程在真正的运行，其余的进程都在等待。操作系统会按照一定的算法进行调度，每个进程轮换的执行。这就导致进程实际运行时间必然大于 CPU 时间。所以说，一个进程运行的耗时还是需要统计 CPU 时间，实际运行时间受到系统负载的影响比较大。

5.1.2　进程内存占用

Linux 下一个进程实际的内存占用的计算是比较复杂的，因为一个进程在运行的时候，其占用的内存并不是仅仅代码中变量占用的内存和动态分配的内存相加，还包括动态库占用、声明占用但未分配等等很多影响因素。

Linux 上的 `ps aux` 命令，可以查看进程的内存占用，

```
virusdefender@vm:~$ ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  3.2  0.1  33900  4440 ?        Ss   11:20   0:02 /sbin/init
```

需要注意的是 RSS 和 VSZ 两列。简单来说 RSS 就是进程实际占用的物理内存，VSZ 是进程的虚拟内存，也就是进程还没有使用但是未来可能会分配的内存大小。如果把所有进程的 RSS 加起来，可能会超过实际的物理内存，那是因为动态库的共享内存在每个进程中都是重复计算的，而实际只需要占用一份内存空间。

5.1.3 进程的状态和返回值

进程结束的时候一般都有返回值的，比如

```
#include <stdlib.h>
#include <string.h>
int main()
{
    int *p = (int *)malloc(100000);
    if(p == NULL){
        exit(1);
    }
    memset(p, 0, 100000);
    return 0;
}
```

就会在分配内存成功的时候返回 0，分配失败的时候返回 1。

Linux 上有一种 signal 的机制来进程进程间通信，进程可以自定义信号处理函数或者忽略信号，比如

```
#include <stdlib.h>
#include <string.h>
int main()
{
    memset(NULL, 0, 100000);
    return 0;
}
```

就会提示段错误，并且因为程序中没有捕获 `SEGFAULT` 的 signal，导致进程被终止。我们可以通过这种机制来获取和控制进程状态。

5.1.4 OJ 上要避免的危险操作

为了确保 OJ 运行环境的安全，沙箱需要控制用户代码可以执行的动作，常见的危险行为有：

- 使用汇编直接调用系统调用
- 调用系统原生 API，比如 `remove("/etc/passwd")`
- 执行系统命令，比如 `system("rm -rf /")`

- 恶意占用大量系统资源，比如 `while(1)fork()`

- 网络通信相关，可能泄露服务器上敏感信息

- 造成编译器出现故障，比如 `#include</dev/random>`，比如产生大量编译错误日志等

5.2 资源限制的具体实现

5.2.1 setitimer 限制进程实际运行时间和 CPU 时间

Linux 中 setitimer 函数可以提供高精度定时器，用于定时执行某个动作，其中具体实现了三种定时器，当定时器计时结束时，系统就会给进程发送一个信号。

需要关心的两个计数器分别是 ITIMER_REAL，进程实际运行时间计数器，结束的时候发送 SIGALRM 信号；ITIMER_VIRTUAL，进程 CPU 时间计数器，结束的时候发送 SIGVTALRM 信号。我们设置好定时器之后，如果捕获到了对应的信号，说明当前进程运行超时。

具体实现代码如下

```
int set_timer(int sec, int ms, int is_cpu_time) {
    struct itimerval time_val;
    time_val.it_interval.tv_sec = time_val.it_interval.tv_usec = 0;
    time_val.it_value.tv_sec = sec;
    time_val.it_value.tv_usec = ms * 1000;
    if (setitimer(is_cpu_time?ITIMER_VIRTUAL:ITIMER_REAL, &time_val, NULL)) {
        LOG_FATAL("setitimer failed, errno %d", errno);
        return SETITIMER_FAILED;
    }
    return SUCCESS;
}
```

但是有一点是需要注意的，setitimer 不能限制子进程的 CPU 和实际运行时间 [7]。在部分只限制资源占用而不启用沙箱的场景下，这可能导致资源限制失效。

5.2.2 setrlimit 限制进程 CPU 时间和内存占用

Linux 中 setrlimit 函数可以用来限制进程的资源占用,其中支持 RLIMIT_CPU、RLIMIT_AS 等参数，同时子进程会继承父进程的设置。RLIMIT_CPU 也可以控制进程 CPU 时间，但是只能精确到秒，所以并没有直接使用这个，而是设置为 CPU 时间向上取整的值，用于限制子进程的 CPU 时间占用；RLIMIT_AS 是限制进程最大内存地址空间，超过这个地址空间的将不能分配成功，影响 brk、mmap、mremap 等系统调用。

具体实现代码如下

```
if (setrlimit(RLIMIT_AS, &memory_limit)) {
    LOG_FATAL("setrlimit failed, errno: %d", errno);
    ERROR(SETRLIMIT_FAILED);
}

cpu_time_rlimit.rlim_cur = cpu_time_rlimit.rlim_max =
                           (config->max_cpu_time + 1000) / 1000;
if (setrlimit(RLIMIT_CPU, &cpu_time_rlimit) == -1) {
    LOG_FATAL(log_fp, "setrlimit cpu time failed, errno: %d", errno);
    ERROR(log_fp, SETRLIMIT_FAILED);
}
```

5.2.3 重定向进程输入输出

Linux 系统中，每个进程都有 stdin、stdout 和 stderr 这 3 种标准输入输出，它们是程序最通用的输入输出方式。但是 OJ 系统中输入和输出通常都是保存在文件中的，我们需要将文件重定向到标准输入输出上。

dup2 函数可以复制文件描述符，两个文件描述符就会完全一致，可以交换使用，这样的话，我们将输入输出文件的文件描述符复制一份给 stdin、stdout 和 stderr 就实现了输入输出重定向了。使用了下面的代码实现

```
if (dup2(fileno(fopen(config->in_file, "r")), 0) == -1) {
     LOG_FATAL("dup2 stdin failed, errno: %d", errno);
    ERROR(DUP2_FAILED);
}
// write stdout to out file
if (dup2(fileno(fopen(config->out_file, "w")), 1) == -1) {
    LOG_FATAL("dup2 stdout failed, errno: %d", errno);
    ERROR(DUP2_FAILED);
}

// redirect stderr to stdout
if (dup2(fileno(stdout), fileno(stderr)) == -1) {
    LOG_FATAL("dup2 stderr failed, errno: %d", errno);
     ERROR(DUP2_FAILED);
}
```

5.2.4 获取进程状态和资源占用情况

Linux 中 wait* 系列函数是用来阻塞并等待进程状态改变的，可以用在父进程中监听子进程状态。我们使用的是 wait4 函数，函数原型是

```
pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);
```

status 参数是用来保存进程状态，比如信号、返回值等，在不同的位上保存；rusage 参数用来保存进程资源占用情况，比如 CPU 时间、内存等等。

```
if (wait4(pid, &status, 0, &resource_usage) == -1) {
    LOG_FATAL("wait4 failed");
    result->flag = SYSTEM_ERROR;
    return;
}
```

在调用 wait4 函数之后，我们可以通过两个宏定义获取返回值和信号。

```
result->exit_status = WEXITSTATUS(status);
result->signal = WIFSIGNALED(status)
```

获取 CPU 时间和内存。由于操作系统内存机制非常复杂而且 OJ 系统对内存限制要求的精确度不高，直接取 max_rss 作为进程的内存占用的值。

```
result->cpu_time = (int) (resource_usage.ru_utime.tv_sec * 1000 +
                  resource_usage.ru_utime.tv_usec / 1000 +
                  resource_usage.ru_stime.tv_sec * 1000 +
                  resource_usage.ru_stime.tv_usec / 1000);
result->memory = resource_usage.ru_maxrss;
```

5.3 沙箱安全机制的具体实现

沙箱安全机制的实现是重点和难点。操作系统提供了一种标准的服务来让程序员实现对底层硬件和服务的控制，叫做系统调用。当一个程序需要作系统调用的时候，它将相关参数放进系统调用相关的寄存器，然后调用 0x80 中断，程序将参数和系统调用号交给内核，内核来完成系统调用的执行。所以沙箱实现的底层都是系统调用的过滤，目前常用的有 ptrace 和 seccomp[8, 9, 10, 11]。

5.3.1 ptrace 和 seccomp 的对比

ptrace 提供了父进程观察和控制子进程执行的能力，并允许父进程检查和替换子进程的内核镜像 (包括寄存器) 的值。其基本原理是：当使用了 ptrace 跟踪后，所有发送给被跟踪的

子进程的信号 (除 SIGKILL)，都会被转发给父进程，而子进程则会被阻塞，这时子进程的状态就会被系统标注为 TASK_TRACED。而父进程收到信号后，就可以对停止下来的子进程进行检查和修改，然后让子进程继续运行。

ptrace 在很多 OJ 上都有应用，但是不可否认的是 ptrace 存在一个重大缺点：严重影响进程运行的性能 [12]，因为每次系统调用就要进行两次上下文切换，从子进程到父进程，然后父进程到子进程。OJ 上题目很多都需要大量的输入和输出，会产生大量的系统调用，导致代码运行时间加长。

安全计算模式 seccomp（Secure Computing Mode）是自 Linux 2.6.10 之后引入到 kernel 的特性。一切都在内核中完成，不需要额外的上下文切换，所以不会造成性能问题。目前在 Docker 和 Chrome 中广泛使用。使用 seccomp，可以定义系统调用白名单和黑名单，可以定义出现非法系统调用时候的动作，比如结束进程或者使进程调用失败。

https://github.com/lodevil/Lo-runner 是使用 ptrace 实现的沙箱，C++ 语言进行 900 万次 cin 和 cout 输入输出，和不使用沙箱对比，CPU 时间消耗增加了近 200%，实际运行时间增加了近 500%。而使用 seccomp，CPU 时间和实际运行时间几乎不变。[13, 14]

表 5.1: 各种沙箱实现对比

沙箱	CPU 时间（ms）	实际运行时间（ms）
ptrace	37047	75967
seccomp	13430	13504
不使用	13076	13163

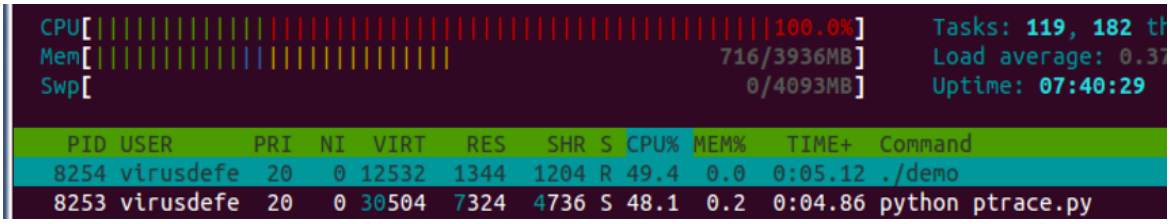

图 5.1: ptrace 父进程也消耗大量 CPU 资源

测试环境：Ubuntu 14.04，虚拟化单核，主机 MacBook Pro Intel(R) Core(TM) i5-4278U CPU @ 2.60GHz。

seccomp 是一种侵入式策略，需要在代码执行之前加载，而不是 ptrace 那样在父进程中全部控制。开发过程中设计过几种加载 seccomp 的方法，但是都逐渐的发现存在绕过，最终方案是在 execve 之前加载，并且对 execve 这个本该是黑名单的系统调用进行了参数过滤，下面章节是详细的分析。

5.3.2 动态库使用 LD_PRELOAD 加载的绕过

在编写代码的时候一般是从 main 函数开始的，但是实际进程执行的时候并不是，在 main 函数执行之前还有一些初始化函数，比如 start 和 __libc_start_main 函数 [15]。如果可以 hook 这些函数就可以在用户代码执行之前执行加载 seccomp 的代码了，很明显可以使用动态库来覆盖 __libc_start_main，加载 seccomp 策略，然后模拟正常执行的去调用 main 函数，示例如下

```
typedef int (*main_t)(int, char **, char **);
int __libc_start_main(main_t main, int argc,
    char *__unbounded *__unbounded ubp_av,
    ElfW(auxv_t) *__unbounded auxvec,
    __typeof (main) init,
    void (*fini) (void),
    void (*rtld_fini) (void), void *__unbounded
    stack_end)
{
    int i;
    ssize_t len;
    void *libc;
    int (*libc_start_main)(main_t main,
        int,
        char *__unbounded *__unbounded,
        ElfW(auxv_t) *,
        __typeof (main),
        void (*fini) (void),
        void (*rtld_fini) (void),
        void *__unbounded stack_end);
    libc = dlopen("libc.so.6", RTLD_LOCAL  | RTLD_LAZY);
    if (!libc) {
        exit(1);
    }

    libc_start_main = dlsym(libc, "__libc_start_main");
    if (!libc_start_main) {
        exit(2);
```

```
}
// TODO: load seccomp rules
    return ((*libc_start_main)(main, argc, ubp_av, auxvec,
                 init, fini, rtld_fini, stack_end));
}
```

上述代码由 https://github.com/quark-zju/lrun 修改而来,完整代码和编译选项可见 GitHub。GitHub 上另外一个项目 https://github.com/daveho/EasySandbox 也使用了类似的方法。并且 EasySandbox 的说明 [16] 中也提到了这种实现可能的绕过:如果用户程序自定义了 `__libc_start_main` 函数，动态库中的 `__libc_start_main` 就不会执行，但是这个需要编译选项`-nostdlib` 的支持，如果没有这个编译选项，就会出现编译错误。

将上述代码加载 seccomp 的部分删除，补充 main 函数，在 Ubuntu 14.04 上使用 gcc 4.8.4 进行编译并没有出现编译错误,并且执行的是确实是用户的而不是沙箱的 `__libc_start_main` 函数，导致沙箱被绕过。所以这种方案只能放弃，如果没有这个问题，确实是一个很好的做法，不需要修改任何代码，只要加载一个动态库，后续的进程行为都被我们控制了。

5.3.3 同名库函数覆盖导致的绕过

为了避免 `__libc_start_main` 函数被覆盖,可以将沙箱代码和用户代码一起编译,这样用户代码中如果也定义了 `__libc_start_main` 就会导致编译错误,实际验证也是这样的。但是用户代码中也可以覆盖一些头文件中的函数为空函数，导致沙箱失效，比如 `seccomp_rule_add`、`seccomp_release` 函数等。

```
#include <stdio.h>
#include <seccomp.h>

int seccomp_rule_add(scmp_filter_ctx ctx, uint32_t action,
                     int syscall, unsigned int arg_cnt, ...) {
    printf("This is a fake seccomp_rule_add\n");
    return 0;
}
int seccomp_load(scmp_filter_ctx ctx) {
    return 0;
}
void seccomp_release(scmp_filter_ctx ctx) {}
int main()
{
```

```
    return 0;
}
```

`gcc main.c sandbox.c -ldl -lseccomp -o user_code && ./user_code` 编译运行发现输出了 This is a fake seccomp_rule_add。

5.3.4 在 execve 之前加载 seccomp

在 `execve` 之前加载 seccomp 看起来是没有问题的，但是 seccomp 策略是白名单类型的，要这样做的话必须将 `execve` 也加入白名单，而这确确实实是一个危险函数，也可以直接用来执行系统命令，所以也是一个潜在的绕过，我们可以使用 seccomp 的参数过滤来过滤 `execve` 的参数，可执行文件路径只能是确定的。例如

```
#include <stdio.h>
#include <unistd.h>
#include <seccomp.h>

int main() {
  char file_name[30] = "/bin/ls";
  char dangerous_file_name[30] = "/bin/rm";
  char *argv[] = {"/", NULL};
  char *env[] = {NULL};
  scmp_filter_ctx ctx;
  ctx = seccomp_init(SCMP_ACT_ALLOW);
  seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(execve), 1,
                        SCMP_A0(SCMP_CMP_NE, file_name));
  seccomp_load(ctx);
  execve(file_name, argv, env);
  return 0;
}
```

如果把 file_name 换成 dangerous_file_name，就会提示 bad system call。这样就解决了 execve 可能带来的绕过。

5.3.5 系统调用白名单

确定了加载 seccomp 的方法之后，只需要确定哪些系统调用是安全的就可以了。在 Linux 系统下，使用 strace 命令可以获取进程运行所有的系统调用，在抽样分析了 OJ 上提交过的代码之后，得到了以下系统调用白名单。

表 5.2: 系统调用白名单

系统调用	说明
open	打开一个文件
read	读取文件
write	写文件
close	关闭文件
fstat	检测文件状态
mmap munmap mprotect arch_ptctl brk	内存分配 相关
access	检查文件权限
exit_group	结束进程的所有线程

和 `execve` 一样，`write` 系统调用也需要过滤参数。当 `write` 的参数为 1 和 2 的时候，代表 `stdout` 和 `strerr`，其余的参数值代表某个文件的文件描述符，如果不限制 `write` 的参数，进程就可以写任意文件了。

使用了下面的代码实现

```
// only fd 0 1 2 are allowed
if (seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 1,
                          SCMP_A0(SCMP_CMP_LE, 2))) {
    LOG_FATAL("load dup2 rule failed");
    ERROR(LOAD_SECCOMP_FAILED);
}
```

5.3.6 用户权限控制

Linux 上用户分为 root 用户和非 root 用户。root 用户拥有最高权限，而 nobody 用户权限非常低，可以使用 nobody 用户来运行风险级别比较高的进程，比如各种 Web 服务器软件。沙箱也使用了这个用户来运行用户的代码，一旦沙箱机制被攻破，还可以通过 Linux 用户权限控制避免更大的危害。Linux 中提供了 `setuid` 函数，可以以 uid 用户身份运行，但是需要 root 权限来调用。

使用了下面的代码实现

```
if (setgid(NOBODY_GID)) {
    LOG_FATAL("setgid failed, errno: %d", errno);
    ERROR(SET_GID_FAILED);
```

```
}
if (setuid(NOBODY_UID)) {
    LOG_FATAL("setuid failed, errno: %d", errno);
    ERROR(SET_UID_FAILED);
}
```

5.3.7 编译器安全

这是一个容易被忽视的方面，目前已知的主要有以下几种。

一是引用某些可以无限输出的文件，比如 `#include</dev/random>`，编译器会一直读取，导致卡死。

二是让编译器产生大量的错误信息，比如下面一段代码，可以让 g++ 编译器产生数 G 的错误日志。

```
int main()
{
    struct x struct z<x(x(x(x(x(x(x(x(x(x(x(x(x(y,x(y><y*,x(y*w>v<y*,w,x{}
    return 0;
}
```

处理方法就是编译器运行的时候也要控制 CPU 时间和实际运行时间，同时使用编译器参数`-fmax-errors=N` 来控制最大错误数量。

三是 C++ 的模板元编程，部分代码是编译期执行的，可以构造出让编译器产生大量计算的代码。

和第二个问题一样，实际大量的去占用资源的并不是编译器进程，而是编译器的子进程，在单纯使用 `setitimer` 方法的时候并无法控制子进程，所以还需要 `setrlimit` 的辅助。

四是引用一些敏感文件可能导致信息泄露，比如 `#include</etc/shadow/>`，会在编译错误的信息中泄露文件开头的内容。

5.3.8 小结

我们主要是从系统调用的层面来保证安全性的，而 Linux 还有很多其他的特性可以用来保证安全，比如 `chroot`、`cgroups`、`user_namespace` 等。还可以将代码放入 Docker 或者虚拟机中运行，使用 Docker 或者虚拟机的隔离性进一步提高安全性。

在开发过程中还遇到了很多细节问题，如不同操作系统系统调用的不同，多线程竞态条件的问题等等。实现一个安全的判题沙箱是很困难的，需要对 Linux API 有深入的了解，有时还需要找源码或者反汇编查看一些底层的实现。

第六章　系统测试与部署

6.1　使用持续集成构建代码与测试环境

在系统开发过程中，为了保证代码质量，使用了 travis-ci 持续集成测试，流程如下：

- 绑定 Github 和 travis-ci 项目，项目中创建相关的配置文件。
- 每次向 Github push 代码或者发起新的 Merge Request 的时候，Github 都会向 travis-ci 发送一个消息，这样 travis-ci 就可以拉取代码进行集成测试了。
- 测试成功或者失败会向邮箱发送邮件，便于了解最新的状态。

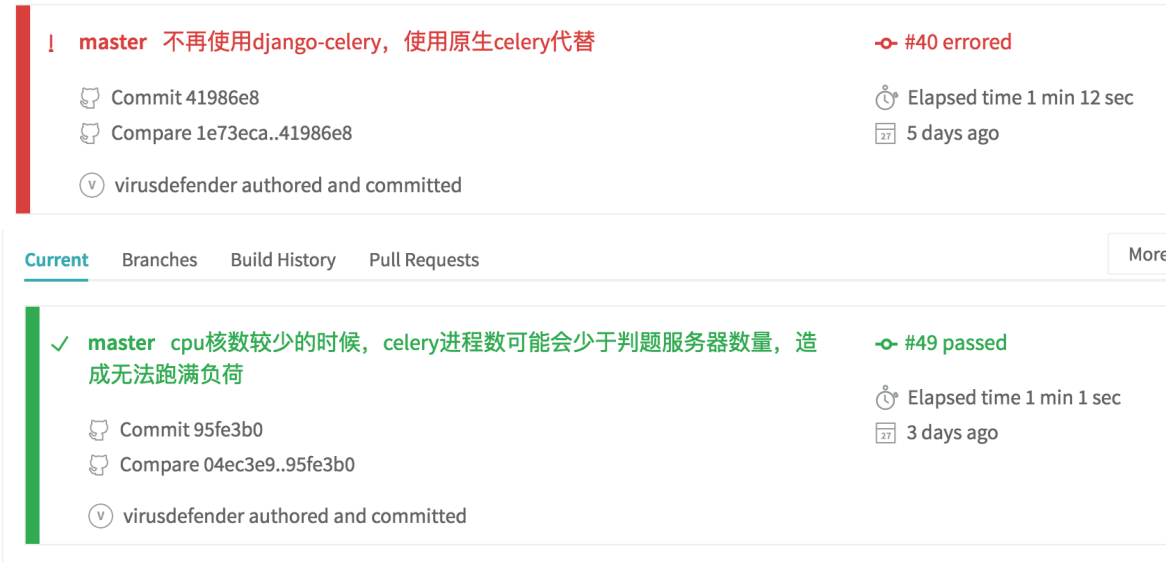

图 6.1: CI 测试失败和通过的界面

如果测试通过，自动化部署的机器人就会拉取最新的代码，通过实现设定的脚本，自动化的更新测试环境, 包括构建新的 Docker 镜像、应用数据库修改和重启 Web Server 等。

6.2　使用 Docker 简化部署难度

Docker 是革命性的产品，大大简化了互联网产品的部署测试难度，它具有如下几个方面的优势：

- 使用 Docker，开发人员可以使用镜像来快速构建一套标准的开发环境，然后可以直接使用相同环境来部署代码。
- 更高效的资源利用。Docker 容器的运行不需要额外的虚拟化管理程序支持，它是内核级的虚拟化，和虚拟机啊相比可以实现更高的性能。

本系统只需要通过 Dockerfile 全自动构建 Docker 镜像,然后简单配置 docker-compose.yml 文件，包括目录映射、部分环境变量和端口映射等，就可以直接运行了。相比传统部署方式，隔离了服务器环境与系统的运行环境，大大降低了部署难度。

Docker 在运行的时候，如果没有配置目录映射，数据都是保存在 Docker 容器里面的，如果 Docker 容器被删除，数据就会丢失，所以需要使用 `volumes` 指令将容器内的数据映射到服务器上存储。

参考文献

[1] *HUSTOJ sql 注入漏洞*. URL: https://github.com/zhblue/hustoj/commit/8c6bf01603e4acd4d0d1036cd01a0f96

[2] *HUSTOJ XSS*. URL: https://github.com/zhblue/hustoj/releases/tag/16.05.

[3] 任意密码重置漏洞 *site:wooyun.org*. URL: https://www.google.com.hk/search?site=&source=hp&q=%E4%BB%BB%E6%84%8F%E5%AF%86%E7%A0%81%E9%87%8D%E7%BD%AE+site:wooyun.org&oq=%E4%BB%BB%E6%84%8F%E5%AF%86%E7%A0%81%E9%87%8D%E7%BD%AE+site:wooyun.org.

[4] 张浩斌. "基于开放式云平台的开源在线评测系统设计与实现". In: 计算机科学 39.11 (2012), pp. 339–343.

[5] 在线判题系统. URL: https://zh.wikipedia.org/wiki/%E5%9C%A8%E7%BA%BF%E5%88%A4%E9%A2%98%E7%B3%BB%E7%BB%9F.

[6] Mr .LZH. 看我如何免笔试获取多家大型企业校招面试资格（赛码校招笔试系统命令执行分析）. URL: http://www.wooyun.org/bugs/wooyun-2015-0141839.

[7] *getitimer(2) - Linux manual page*. URL: http://man7.org/linux/man-pages/man2/setitimer.2.html.

[8] Taesoo Kim and Nickolai Zeldovich. "Practical and Effective Sandboxing for Non-root Users." In: *USENIX Annual Technical Conference*. 2013, pp. 139–144.

[9] Chao Yi, Su Feng, and Zhi Gong. "A Comparison of Sandbox Technologies Used in Online Judge Systems". In: *Applied Mechanics and Materials*. Vol. 490. Trans Tech Publ. 2014, pp. 1201–1204.

[10] Martin Mareš and Bernard Blackham. "A new contest sandbox". In: *Olympiads in Informatics* 6 (2012), pp. 100–109.

[11] Jedrzej Jabłonski and Marcin Pawłowski. "Secure sandboxing solution for GNU/Linux". In: (2011).

[12] Bruce Merry. "Performance analysis of sandboxes for reactive tasks". In: *Olympiads in Informatics* 4 (2010), pp. 87–94.

[13] 李扬. 本论文所有的测试代码和数据. URL: https://github.com/virusdefender/UndergraduateThesis.

[14] qduoj dev team. *qduoj 开源判题沙箱*. URL: https://github.com/QingdaoU/Judger.

[15] 亚嵌教育. *main 函数和启动例程*. URL: https://akaedu.github.io/book/ch19s02.html.

[16] daveho. *Using EasySandbox*. URL: https://github.com/daveho/EasySandbox/blob/master/README.md#using-easysandbox.

谢 辞

时光荏苒，岁月如梭，写下这篇致谢的时候，大学生活就要接近尾声了。还能在青大有最后一个多的时光，虽有激动，但心中更多的是不舍。

感谢信息工程学院（现计算机学院科学技术学院）创新实验室的李建波老师和青大 ACM 队的杜祥军老师，在创新实验室和 ACM 队的近三年时间，是我快速成长的三年，同时实验室的同学们也给予了非常多的帮助，让我能一直坚定的走下去，真的很高兴遇到大家。

感谢参与部分模块开发和改进孙小文、王涛、孙晓冬和江学磊同学，感谢给该毕业设计提出过问题和建议的朋友们。

感谢指导老师陈宇，在本论文的写作过程中提出了非常多的建设性意见。