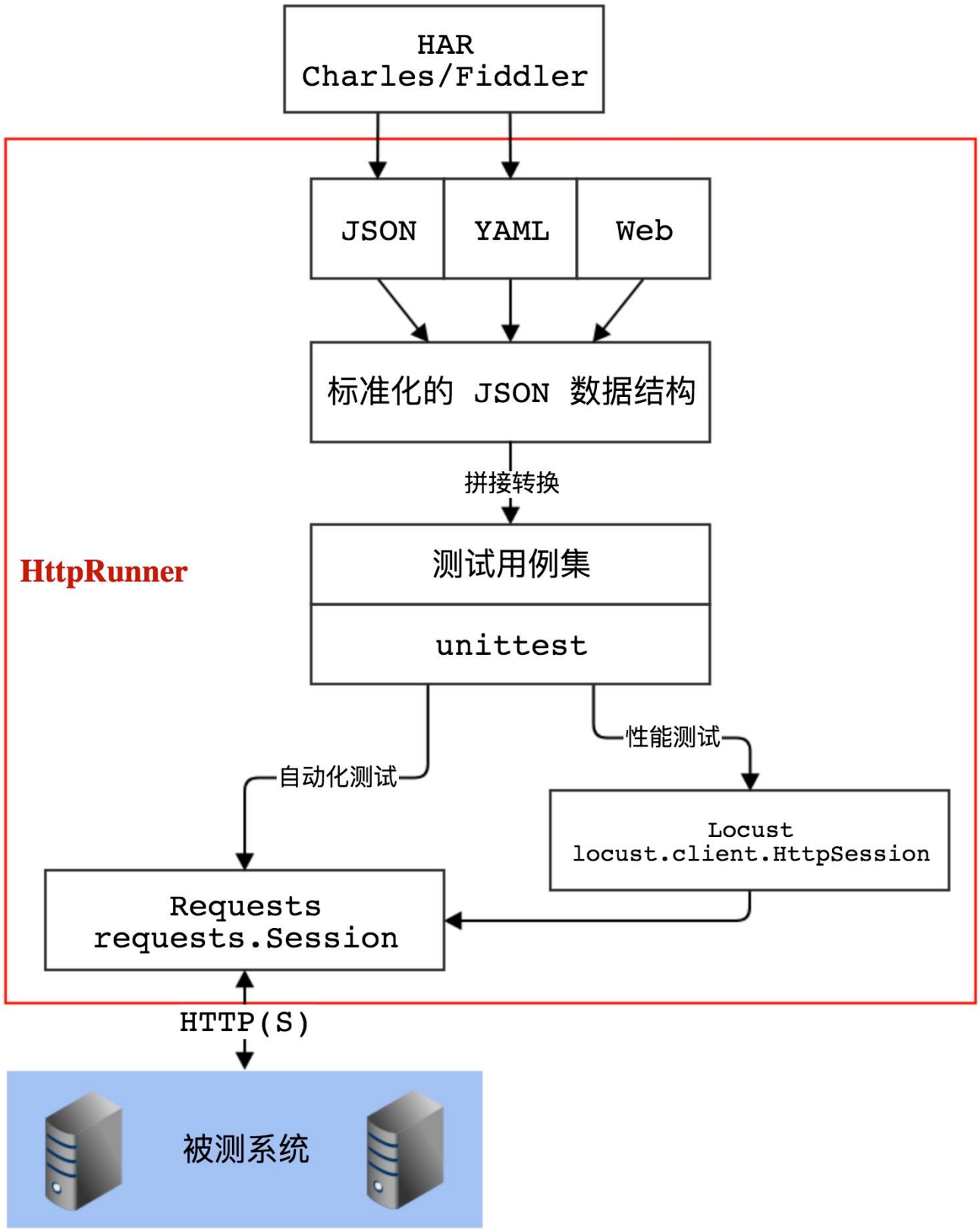
HttpRunner

简介

HttpRunner 是一款面向 HTTP(S) 协议的通用测试框架，只需编写维护一份 YAML/JSON 脚本，即可实现自动化测试、性能测试、线上监控、持续集成等多种测试需求。

- 项目地址：https://github.com/HttpRunner/HttpRunner
- 中文手册：http://cn.httprunner.org

框架流程


HAR
Charles/Fiddler
JSON
YAML
Web
标准化的 JSON 数据结构
拼接转换
HttpRunner
测试用例集
unittest
性能测试
自动化测试
Locust
locust.client.HttpSession
Requests
requests.Session
HTTP(S)
被测系统

核心特性

- 继承 Requests 的全部特性，轻松实现 HTTP(S) 的各种测试需求
- 测试用例与代码分离，采用 YAML/JSON 的形式描述测试场景，保障测试用例具备可维护性
- 测试用例支持分层机制，充分实现测试用例的复用
- 测试用例支持参数化和数据驱动机制
- 使用 skip 机制实现对测试用例的分组执行控制
- 测试请求支持完善的 hook 机制
- 支持热加载机制，在文本测试用例中轻松实现复杂的动态计算逻辑
- 基于 HAR 实现接口录制和用例生成功能（har2case）
- 结合 Locust 框架，无需额外的工作即可实现分布式性能测试
- 执行方式采用 CLI 调用，可与 Jenkins 等持续集成工具完美结合
- 测试结果统计报告简洁清晰，附带详尽统计信息和日志记录
- 具有可扩展性，便于扩展实现 Web 平台化（HttpRunnerManager）

下载安装

使用 pip 命令进行安装

```
pip install httprunner
```

安装后校验是否安装成功，可以使用如下命令进行校验

```
hrun -V
1.4.2

har2case -V
0.1.8
```

若版本号正常显示，则说明安装正常。

入门使用

测试场景

- 测试接口：http://httpbin.org/get
- 接口类型：GET

用例设计

HttpRunner 的测试用例支持两种文件格式：YAML 和 JSON。这里以 YAML 为例。

test_httpbin.yml

```
- config:
    name: httpbin api test
    request:
        base_url: http://www.httpbin.org
- test:
    name: get request
    request:
        url: /get
        method: GET
    validate:
        - eq: [status_code,200]
```

- `config`：作为整个测试用例集的全局配置项
- `test`：对应单个测试用例
- `name` 这个 test 的名字
- `request` 这个 test 具体发送 http 请求的各种信息，如下:

- `url` 请求的路径（若 config 中有定义 base_url，则完整路径是用 base_url + url）
- `method` 请求方法 POST, GET 等等
- `validate` 完成请求后，所要进行的验证内容. 所有验证内容均通过该 test 才算通过,否则失败.

相关资料

- yaml 教程
- HttpRunner 用例结构

运行测试

使用 `hrun` 执行测试，如下所示：

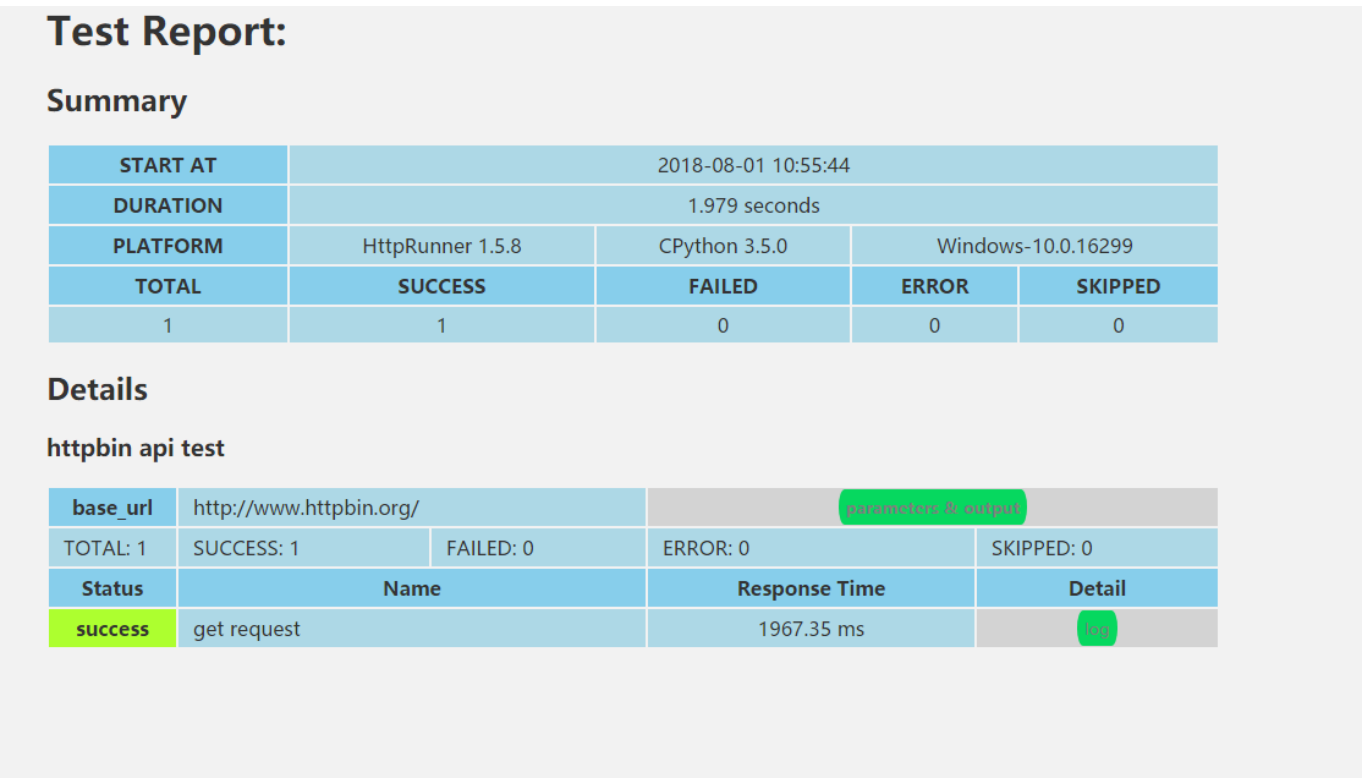
```
C:\Users\Shuqing>hrun D:\api_test\HttpRunner_test\test_httpbin.yml
get request
INFO     GET /get
INFO     status_code: 200, response_time(ms): 1967.35 ms, response_length: 273 bytes
INFO     start to validate.
.

----------------------------------------------------------------------
Ran 1 test in 1.976s

OK
INFO     Start to render Html report ...
INFO     Generated Html report: C:\Users\Shuqing\reports\1533092144.html
```

查看测试报告

打开 html 报告如下：



Test Report:

Summary

START AT	2018-08-01 10:55:44			
DURATION	1.979 seconds			
PLATFORM	HttpRunner 1.5.8	CPython 3.5.0	Windows-10.0.16299	
TOTAL	SUCCESS	FAILED	ERROR	SKIPPED
1	1	0	0	0

Details

httpbin api test

base_url	http://www.httpbin.org/		parameters & output	
TOTAL: 1	SUCCESS: 1	FAILED: 0	ERROR: 0	SKIPPED: 0
Status	Name		Response Time	Detail
success	get request		1967.35 ms	log

HttpRunnerManager

简介

HttpRunnerManager 是基于 `HttpRunner` 的接口自动化测试平台,该工具是对 `HttpRunner` 的包装和 Web 图形化，另外还增加了一些新概念(项目/模块)用来组织用例。 如果对 yaml 语法格式不熟悉，以及对于 httprunner 命令不熟悉的可以使用该平台执行接口自动化测试。

项目地址：https://github.com/HttpRunner/HttpRunnerManager

核心特性

- 项目管理：新增项目、列表展示及相关操作，支持用例批量上传(标准化的 HttpRunner json 和 yaml 用例脚本)
- 模块管理：为项目新增模块，用例和配置都归属于 module，module 和 project 支持同步和异步方式

- 用例管理：分为添加 config 与 test 子功能，config 定义全部变量和 request 等相关信息 request 可以为公共参数和请求头，也可定义全部变量
- 场景管理：可以动态加载可引用的用例，跨项目、跨模快，依赖用例列表支持拖拽排序和删除
- 运行方式：可单个 test，单个 module，单个 project，也可选择多个批量运行，支持自定义测试计划，运行时可以灵活选择配置和环境，
- 分布执行：单个用例和批量执行结果会直接在前端展示，模块和项目执行可选择为同步或者异步方式，
- 环境管理：可添加运行环境，运行用例时可以一键切换环境
- 报告查看：所有异步执行的用例均可在线查看报告，可自主命名，为空默认时间戳保存，
- 定时任务：可设置定时任务，遵循 `crontab` 表达式，可在线开启、关闭，完毕后支持邮件通知
- 持续集成：jenkins 对接，开发中。。。

下载安装

1. 安装 mysql 数据库服务端(推荐 5.7+),并设置为 utf-8 编码，创建相应 `HttpRunnerManager` 数据库，设置好相应用户名、密码，启动 mysql。
2. 将 HttpRunnerManager 下载下来，解压放在任意盘符位置，例如我放在 D 盘根目录，并重命名为 `HttpRunnerManager`

环境配置

HttpRunnerManager 支持分布式执行，模块和项目执行可选择为同步或者异步方式，因此需要安装相关依赖工具。

erlang

Erlang 是一种通用的面向并发的编程语言，它由瑞典电信设备制造商爱立信所辖的 CS-Lab 开发，目的是创造一种可以应对大规模并发活动的编程语言和运行环境。

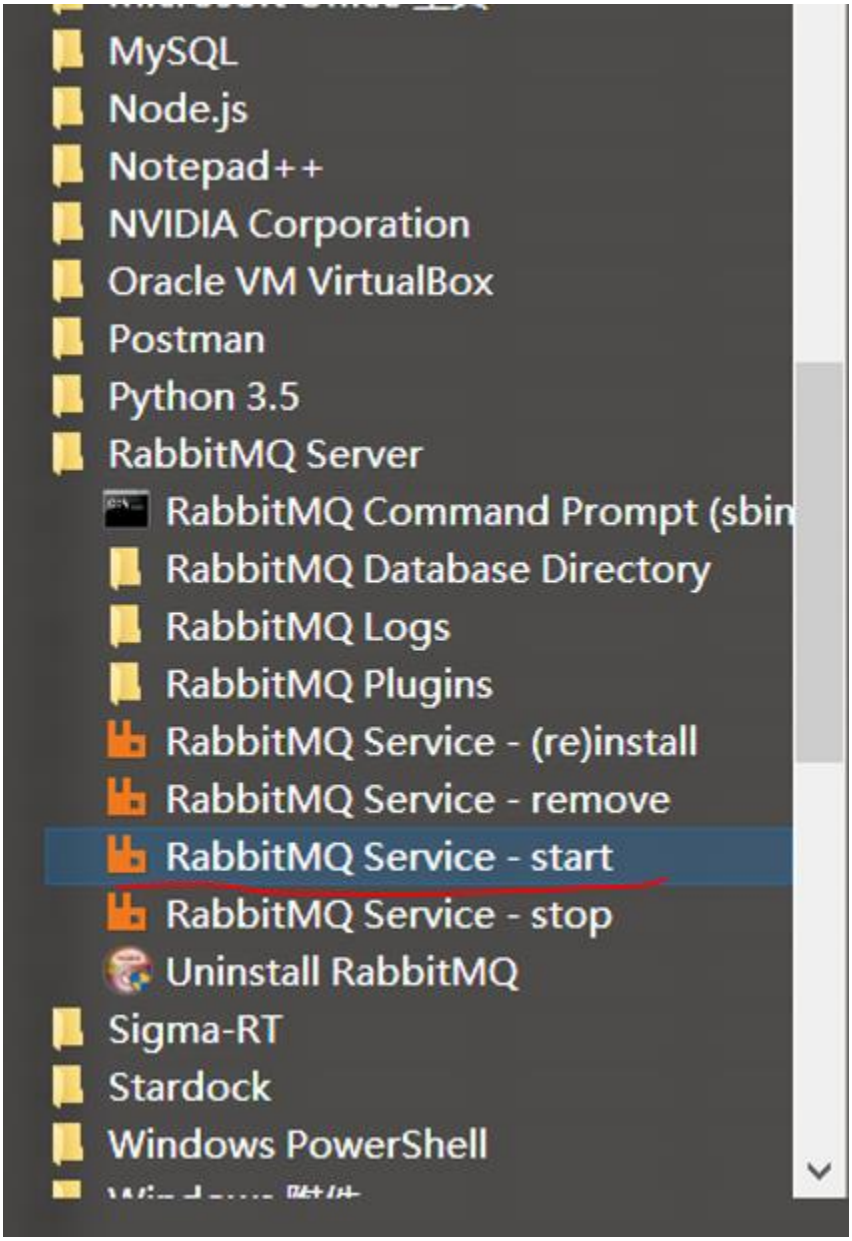
下载地址：http://www.erlang.org/downloads

Rabbitmq

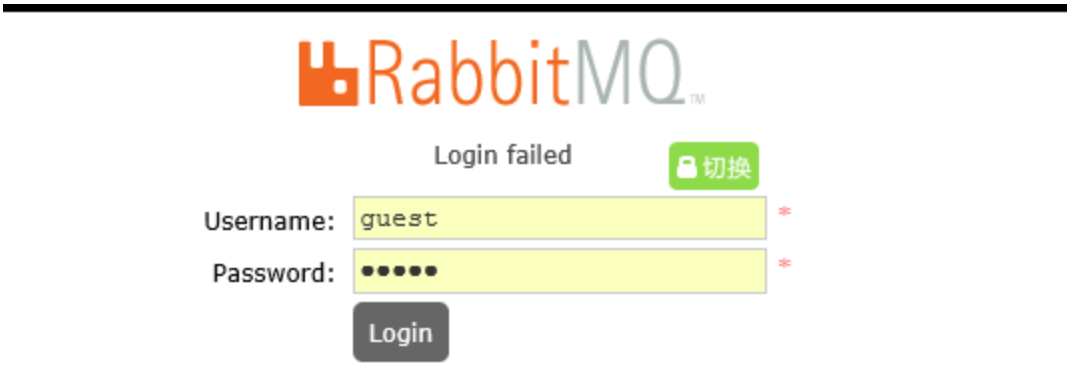
RabbitMQ 是一个由 Erlang 语言开发的 AMQP(高级消息队列协议)的开源实现。它支持多个消息传递协议。RabbitMQ 可以部署在分布式和联合配置中，以满足高规模、高可用性的需求，另外安装 rabbitmq 需要先安装 `erlang`。

下载地址：http://www.rabbitmq.com/download.html 下载后双击 `rabbitmq-server-3.7.7.exe` 文件进行安装。

安装完成后如下图如所示，选中 `RabbitMQ Service -start` 然后以管理员身份运行。



可以通过访问 http://localhost:15672 进行测试，默认的登陆账号为：guest，密码为：guest。



相关资料：[RabbitMQ Windows 环境安装官方手册]

数据库配置

打开 HttpRunnerManager 项目的 setting.py 文件，进行如下配置

```
if DEBUG:
    DATABASES = {
        'default': {
            'ENGINE': 'django.db.backends.mysql',
            'NAME': 'HttpRunnerManager',  # 新建数据库名
            'USER': 'root',  # 数据库登录名
            'PASSWORD': '',  # 数据库登录密码
            'HOST': '127.0.0.1',  # 数据库所在服务器 ip 地址
            'PORT': '3306',  # 监听端口 默认 3306 即可
        }
    }
    STATICFILES_DIRS = (
        os.path.join(BASE_DIR, 'static'),  # 静态文件额外目录
    )
else:
    DATABASES = {
        'default': {
            'ENGINE': 'django.db.backends.mysql',
            'NAME': 'HttpRunnerManager',  # 新建数据库名
            'USER': 'root',  # 数据库登录名
            'PASSWORD': '',  # 数据库登录密码
            'HOST': '127.0.0.1',  # 数据库所在服务器 ip 地址
            'PORT': '3306',  # 监听端口 默认 3306 即可
        }
    }
    STATIC_ROOT = os.path.join(BASE_DIR, 'static')
```

work 配置

修改 work 配置

```
djcelery.setup_loader()
CELERY_ENABLE_UTC = True
CELERY_TIMEZONE = 'Asia/Shanghai'
BROKER_URL = 'amqp://guest:guest@127.0.0.1:5672//' if DEBUG else 'amqp://guest:guest@127.0.0.1:5672//'
CELERYBEAT_SCHEDULER = 'djcelery.schedulers.DatabaseScheduler'
```

```
CELERY_RESULT_BACKEND = 'djcelery.backends.database:DatabaseBackend'
CELERY_ACCEPT_CONTENT = ['application/json']
CELERY_TASK_SERIALIZER = 'json'
CELERY_RESULT_SERIALIZER = 'json'

CELERY_TASK_RESULT_EXPIRES = 7200  # celery 任务执行结果的超时时间,
CELERYD_CONCURRENCY = 1 if DEBUG else 10 # celery worker 的并发数 也是命令行-c 指定的数目 根据服务器配置实际更改 一般 25 即可
CELERYD_MAX_TASKS_PER_CHILD = 100  # 每个worker 执行了多少任务就会死掉，我建议数量可以大一些，比如 200


EMAIL_SEND_USERNAME = 'xxxx@163.com'  # 定时任务报告发送邮箱，支持 163,qq,sina,企业 qq 邮箱等，注意需要开通 smtp 服务
EMAIL_SEND_PASSWORD = 'XXX'     # 邮箱密码
```

安装依赖库文件

打开 cmd 命令窗口，切换到 HttpRunnerManager 目录，然后执行下面命令，自动安装需要的依赖库文件。

```
pip install -r requirements.txt
```

数据库迁移

```
python manage.py makemigrations ApiManager #生成数据迁移脚本
python manage.py migrate  #应用到 db 生成数据表
```

创建超级用户，用户后台管理数据库，并按提示输入相应用户名，密码，邮箱。

```
python manage.py createsuperuser
```

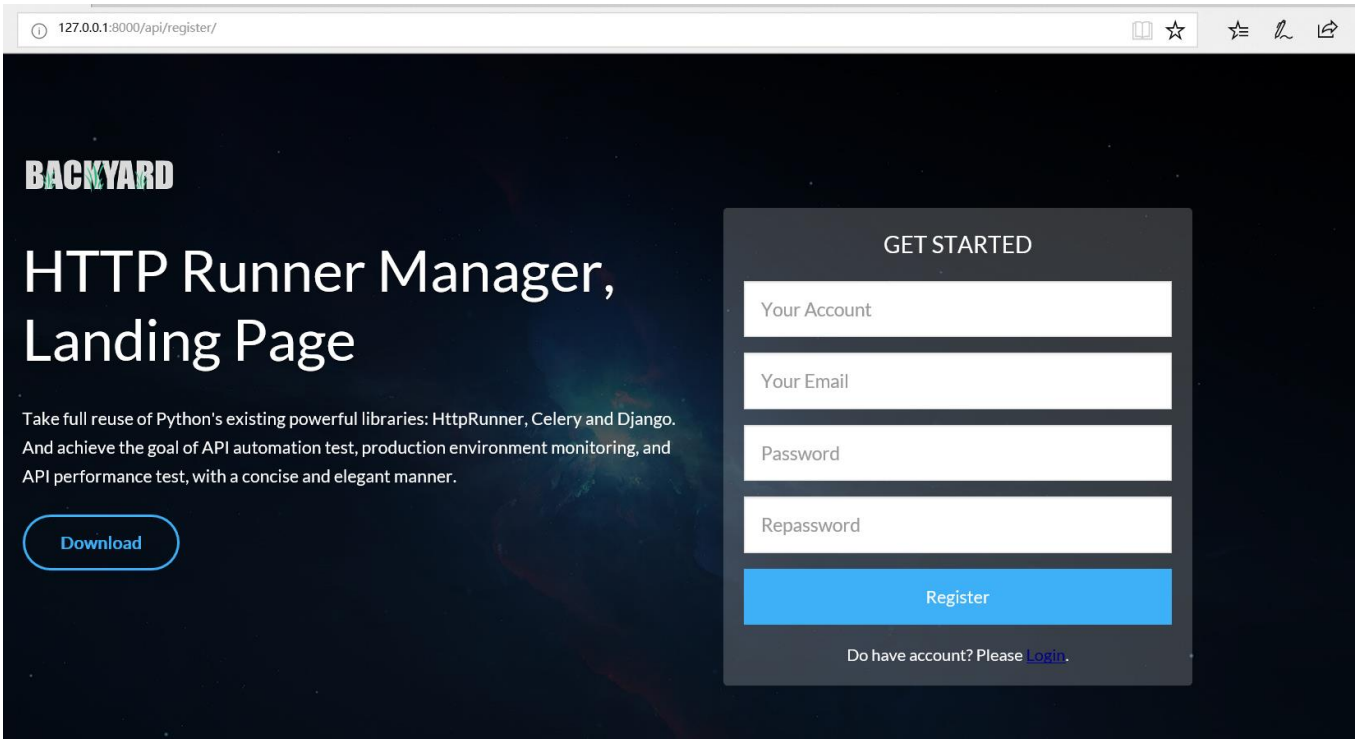
启动服务

输入下面命令启动服务

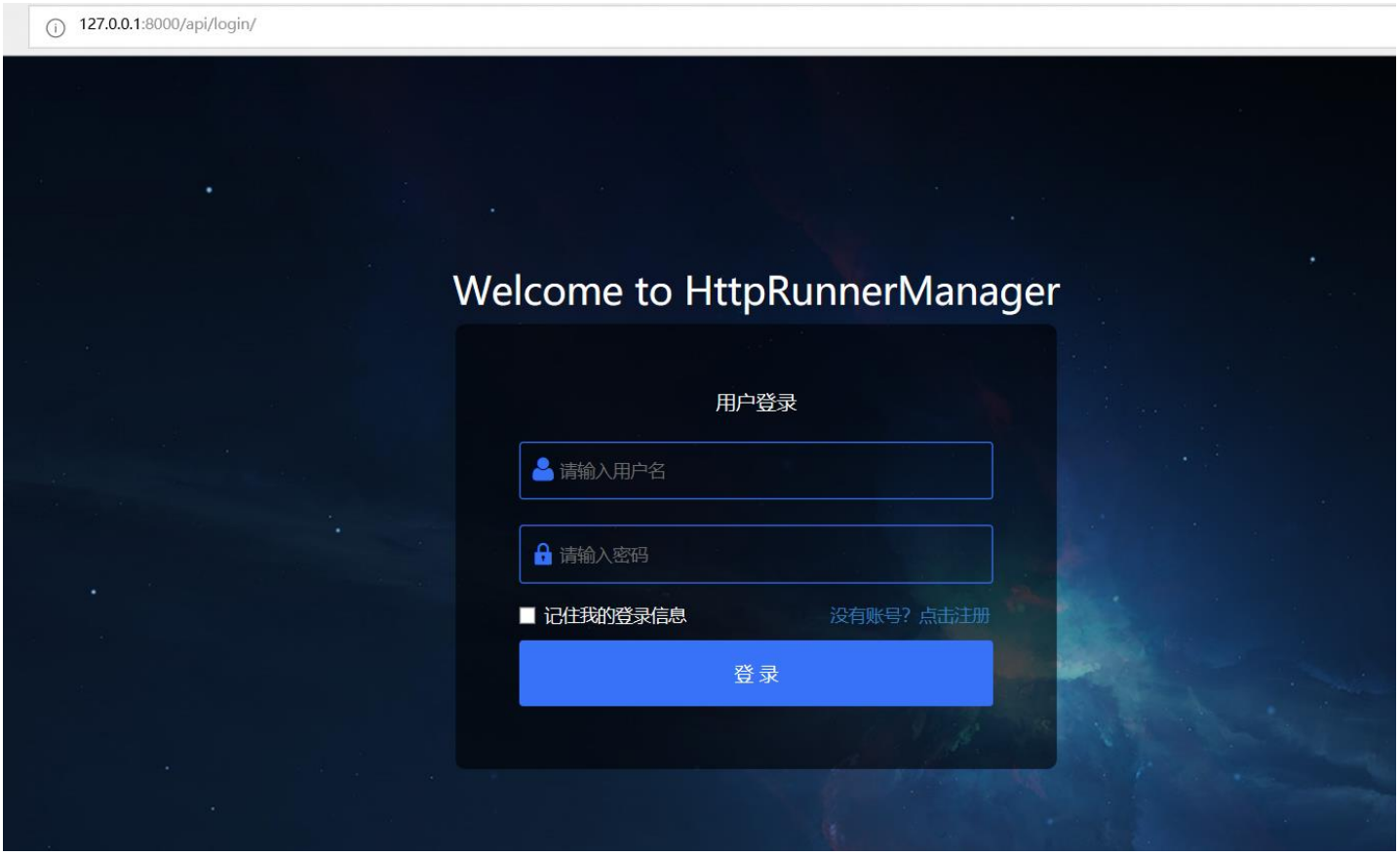
```
python manage.py runserver
```

服务启动成功之后，打开如下地址，可以进入到不同的页面。

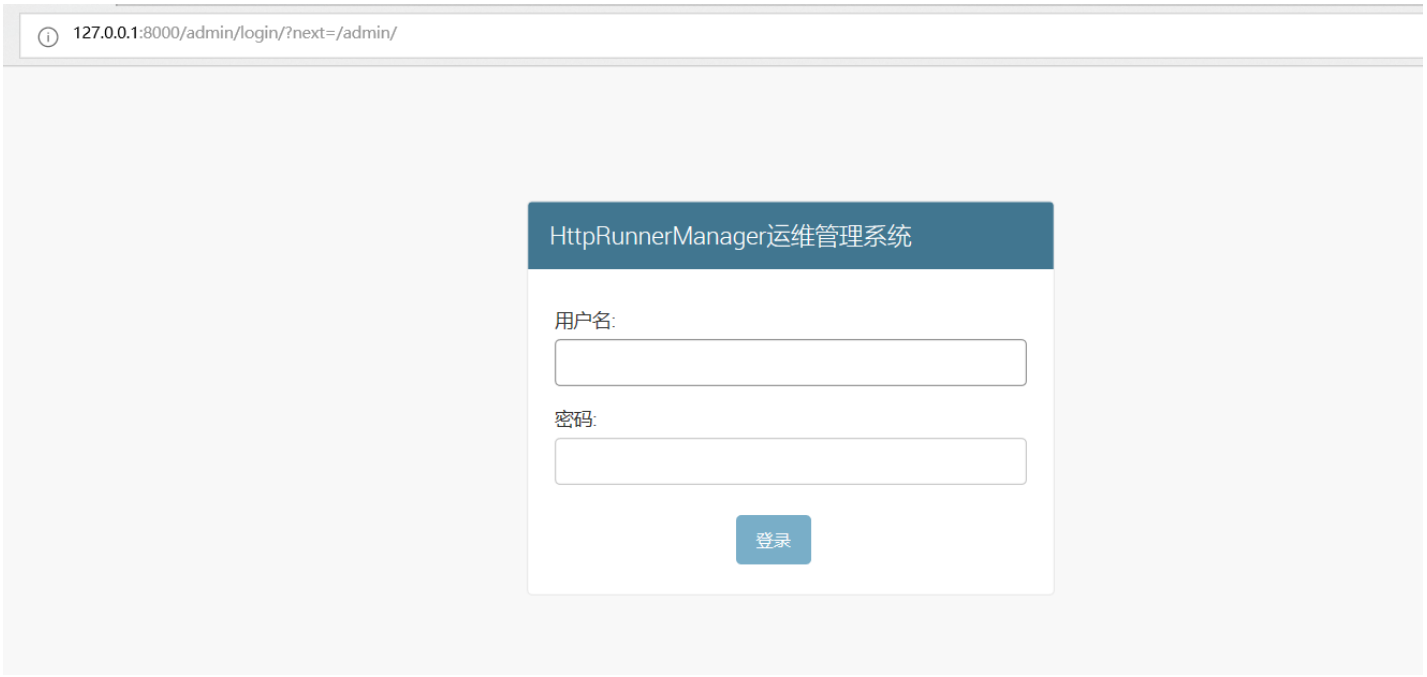
- 注册: http://127.0.0.1:8000/api/register/



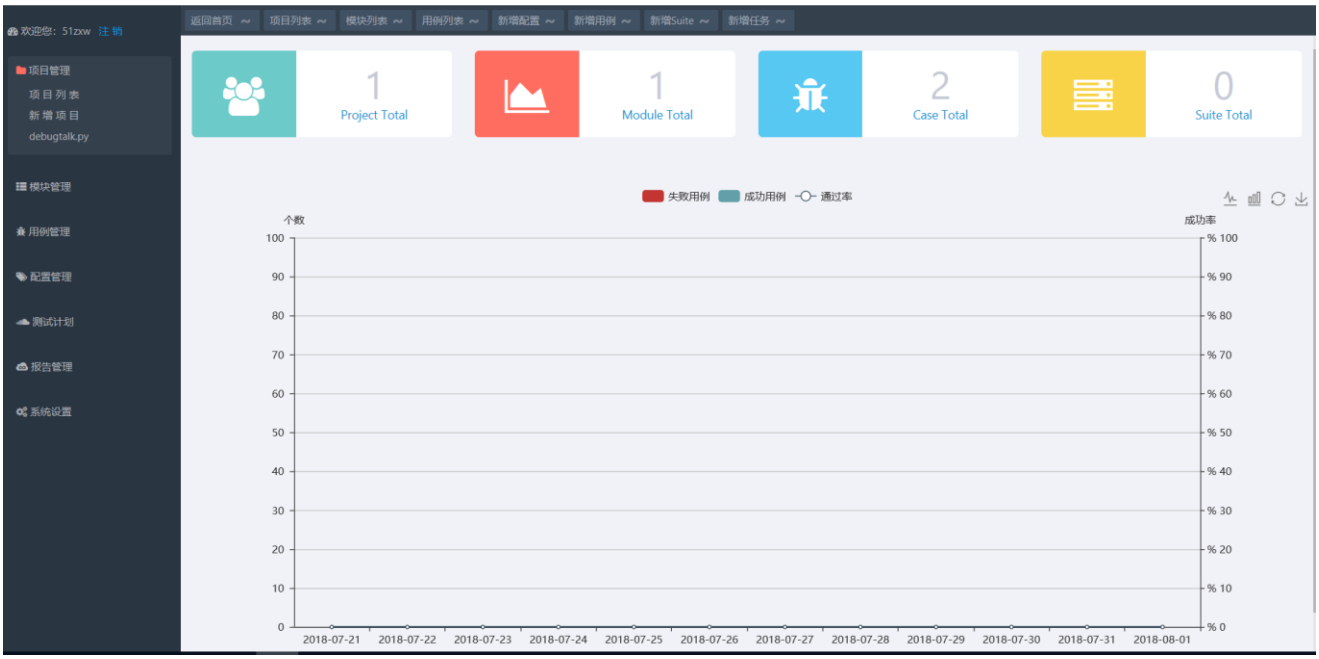
- 登录: http://127.0.0.1:8000/api/login/



- 后台数据库管理：http://127.0.0.1:8000/admin/



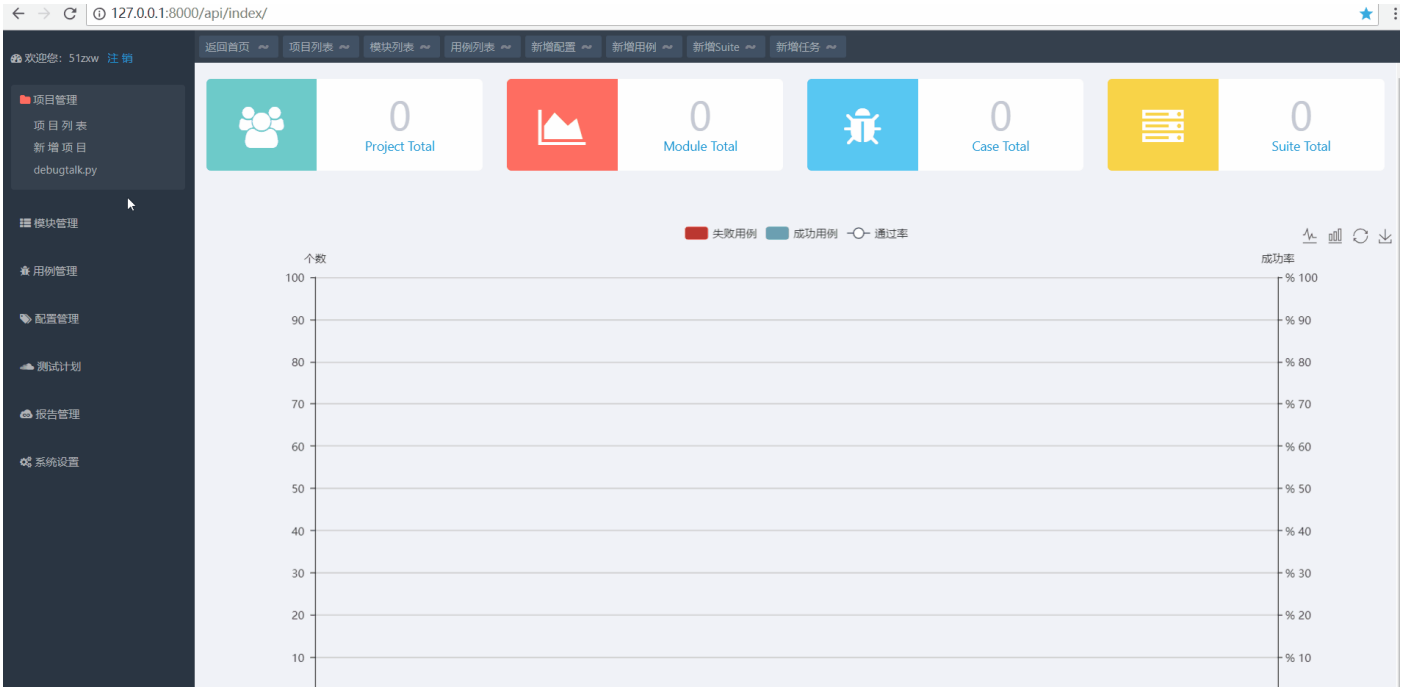
注册登录之后就可以看到平台的界面，接下来就可以创建接口测试的项目和用例了。



HttpRunnerManage 入门使用

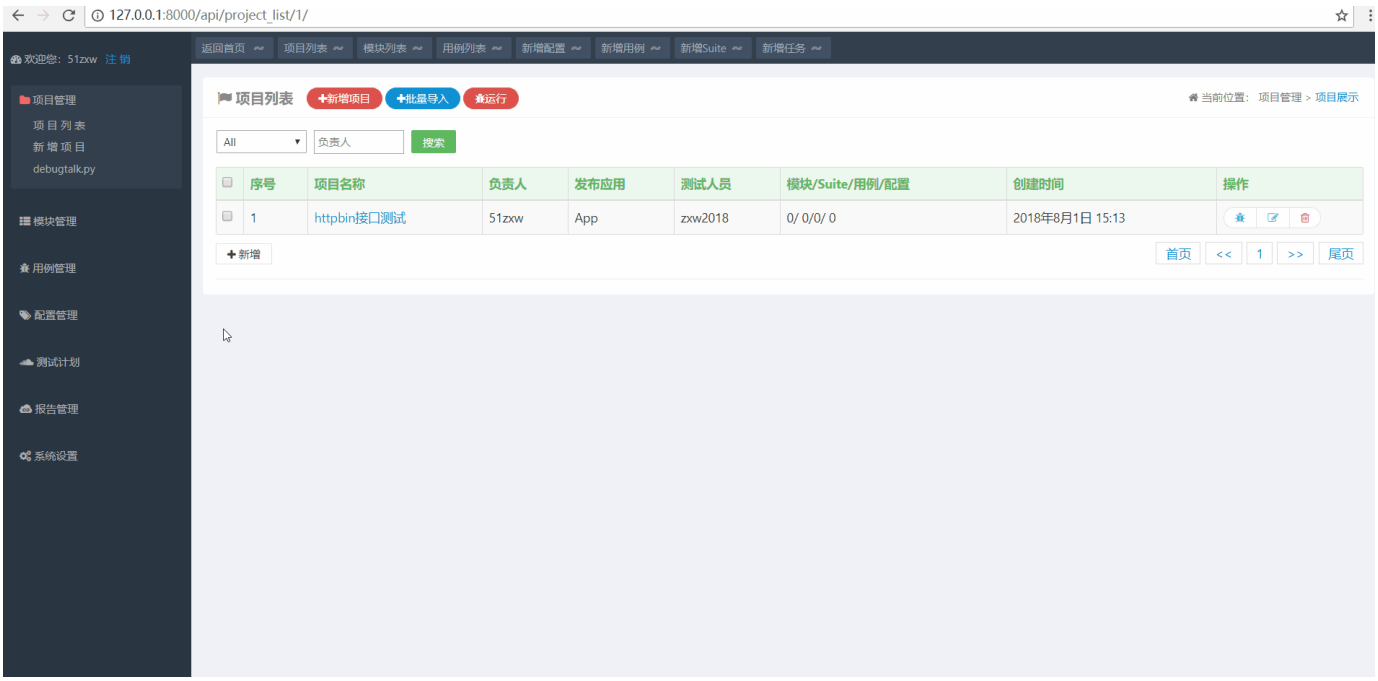
创建项目

在首页点击左侧菜单栏**新增项目**，然后输入项目相关信息。我们接下来将会以 [httpbin](#) 里面的接口来进行测试,所以项目名称命名为:**httpbin 接口测试**



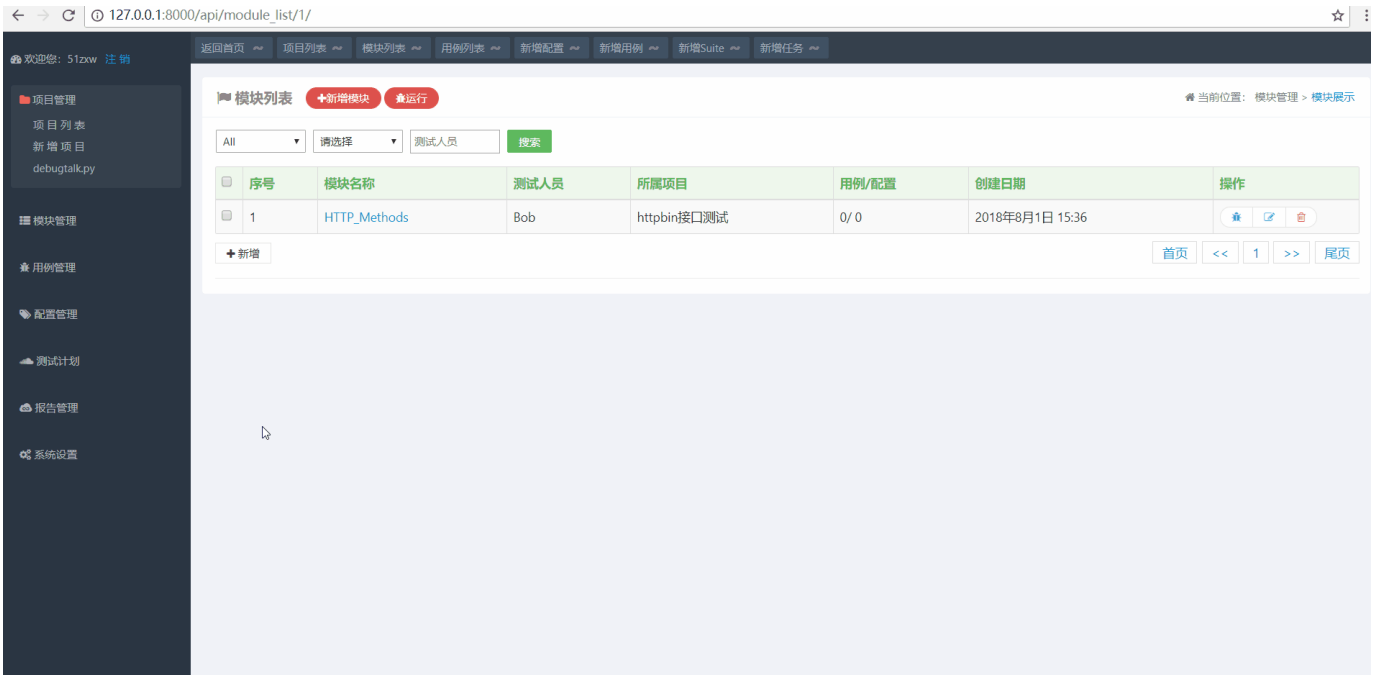
创建模块

一个项目会一般分为多个功能模块，我们可以创建不同模块，然后基于不同模块创建测试用例。 在左侧菜单选择**模块管理** 然后点击新增模块，接下来**输入模块**信息。 这里我们创建一个模块：`HTTP_Methods`



创建环境

在接口测试过程中，我们有时需要设置 `base_url` 来提高用例编写执行效率，我们可以在**系统设置**中的**运行环境**来创建。例如我们创建一个 base_url 操作过程如下图所示：

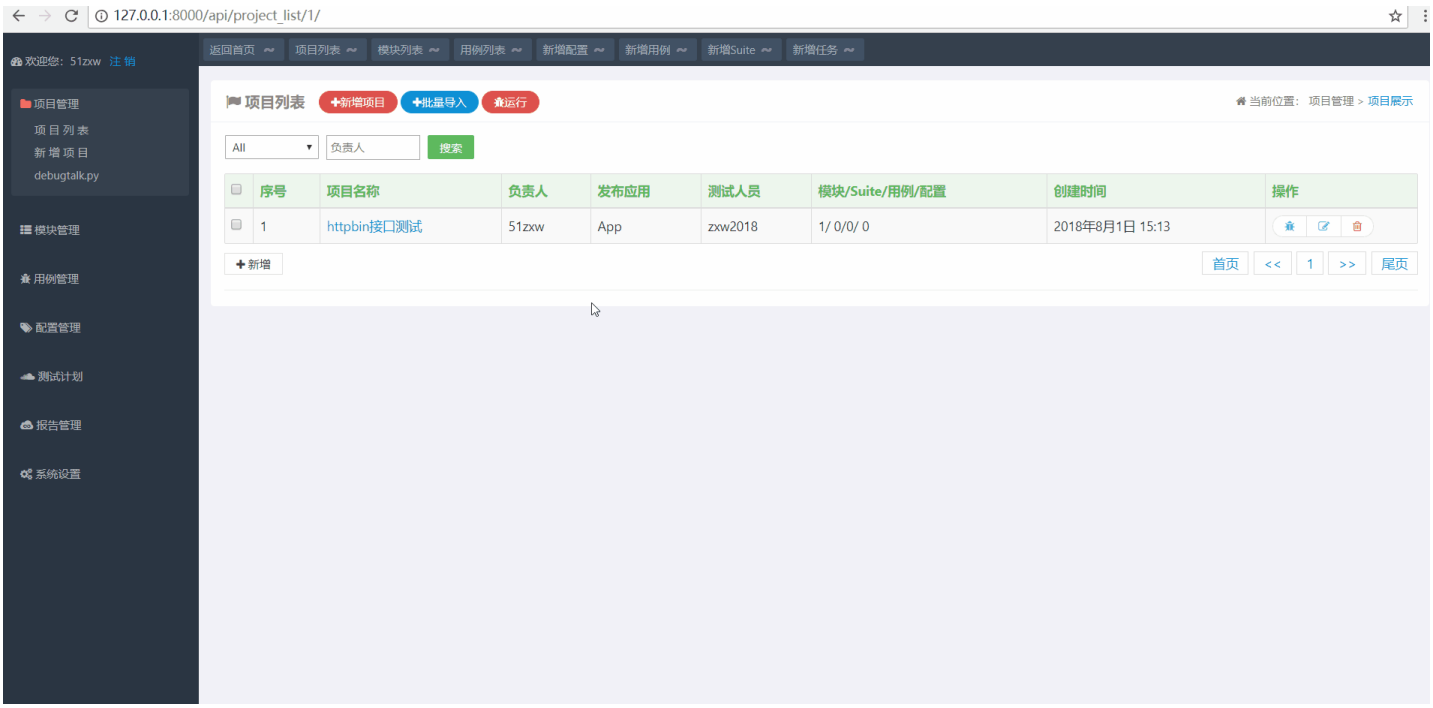


创建用例

这里以下面接口为例创建用例：

```
http://www.httpbin.org/get #请求方式为 GET
```

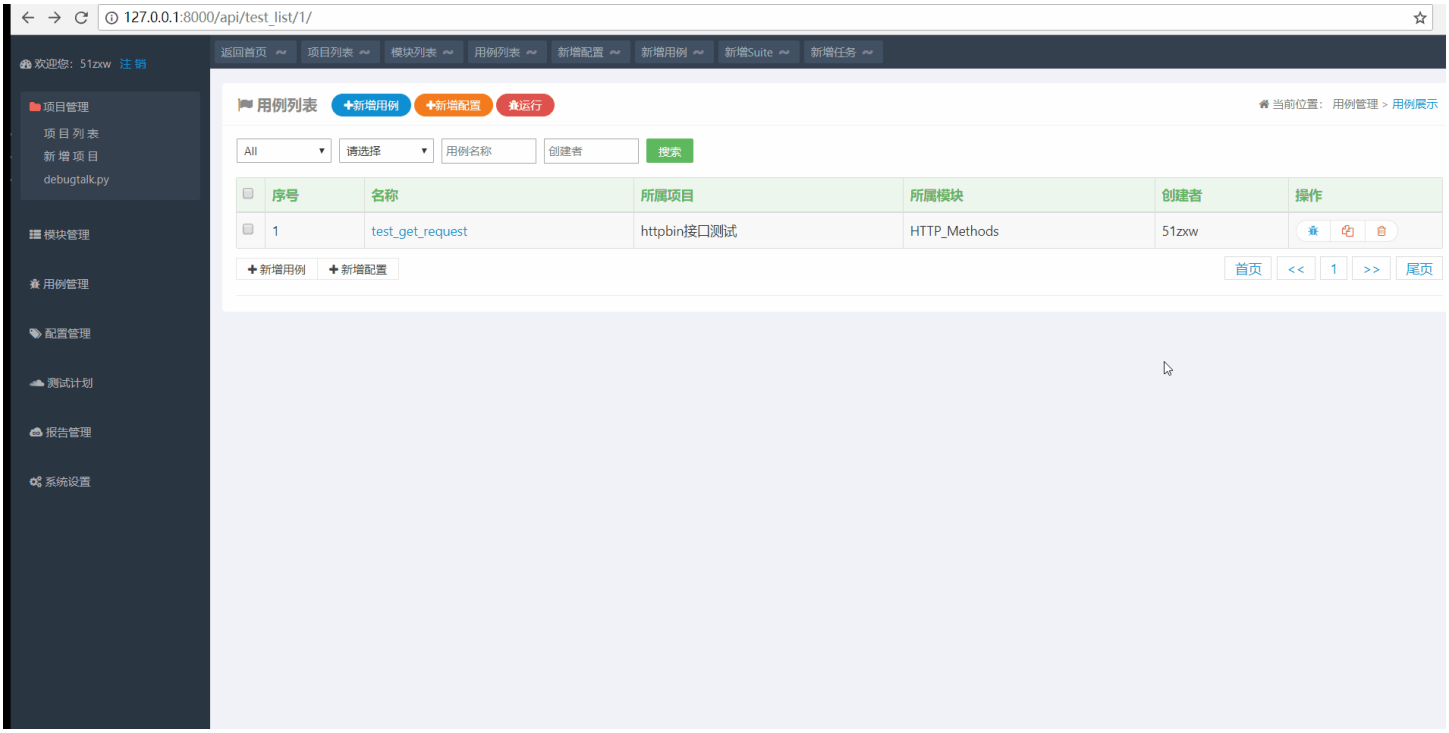
点击顶部快捷入口**新增用例** 然后在用例编辑窗口切换到 request 来编辑用例，操作步骤如下：



如上图演示所示，用例名称为 test_get_request 用例要归属到项目和具体的模块

运行测试

如下图所示，点击用例测试的运行图标，然后选择运行环境即可执行用例，执行完成之后会自动生成测试报告，可以查看运行的结果。

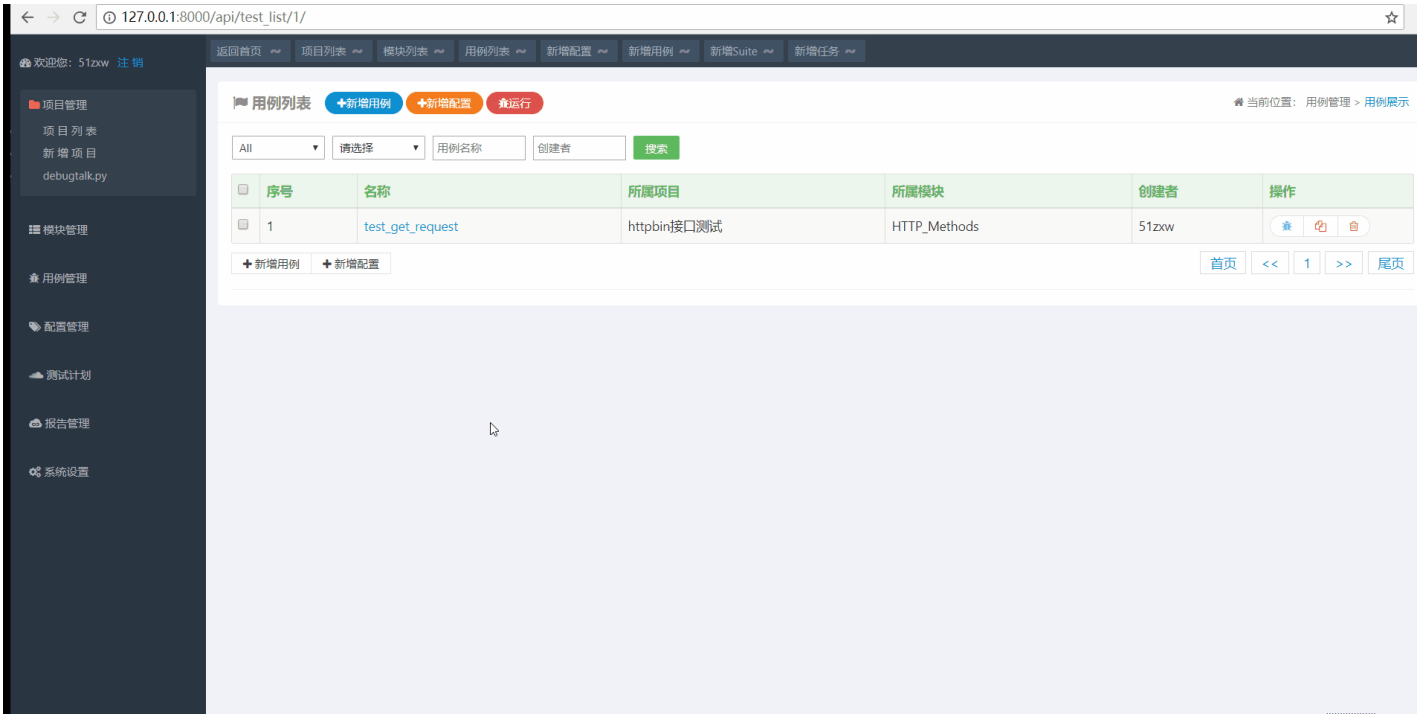


以上我们就完成了单个接口的简单测试，接下来我们要进一步完善用例。

用例配置

header 设置

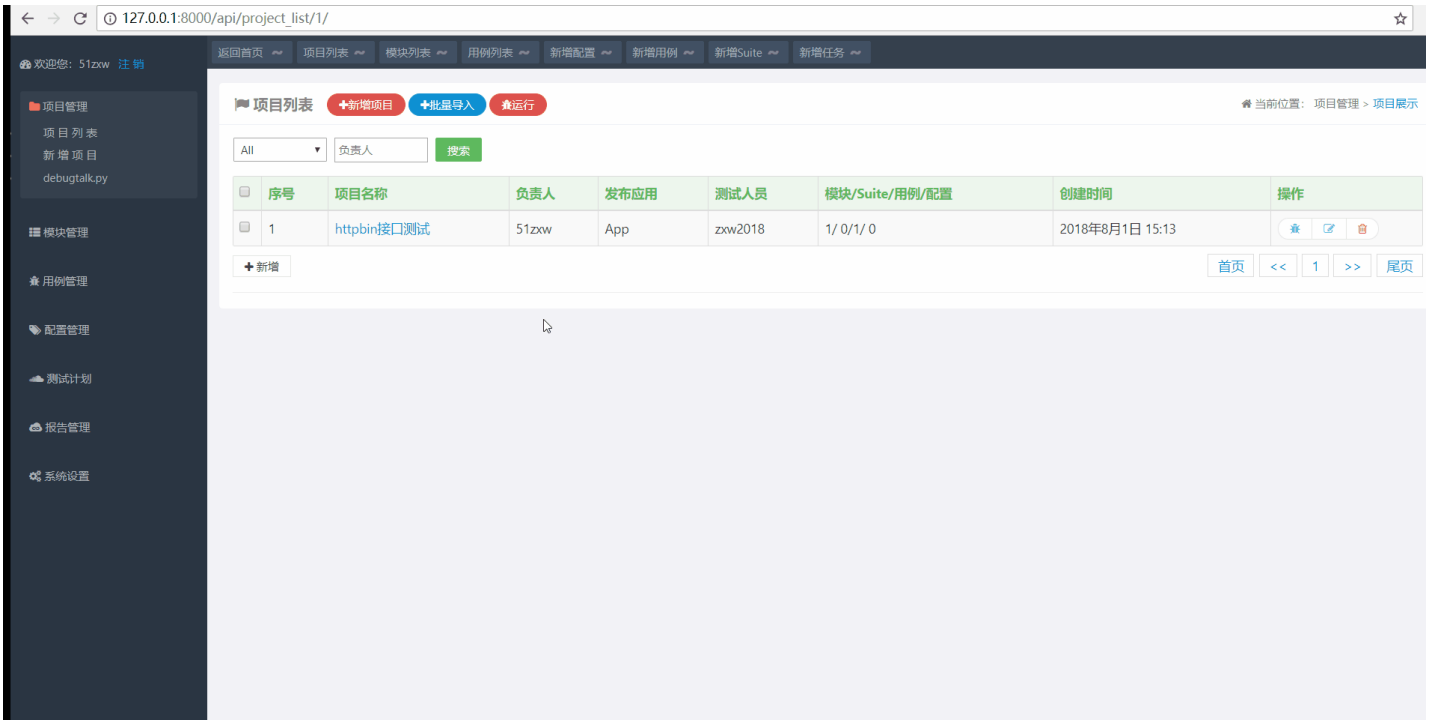
如果想自定义 header 信息，则可以在用例编辑界面点击 `add headers`，然后配置 header 信息。操作如下图所示:



请求参数

URL 参数

在 GET 请求中，经常会有在 URL 中的参数，也就是 `Query String Parameters` 比如在用例 `test_get_request` 增加一个参数 `user=51zxw` 操作过程如下：



body 参数

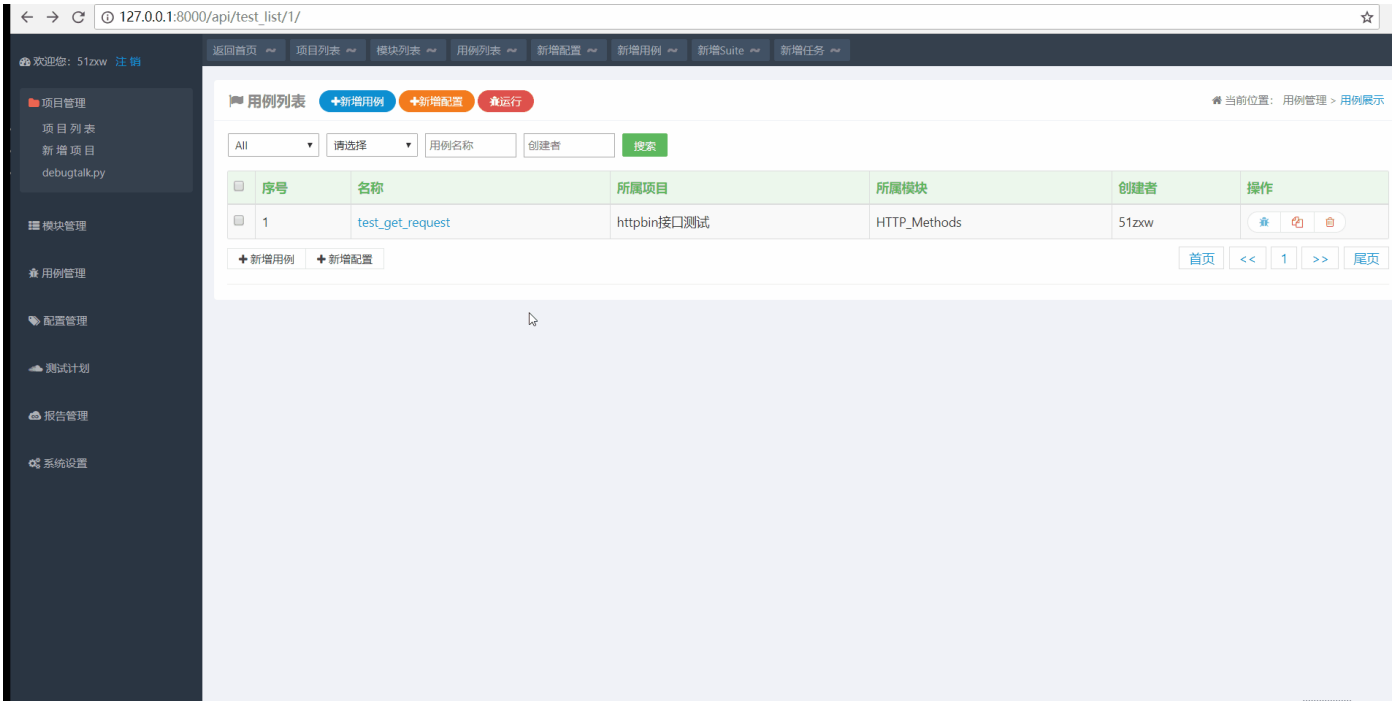
在 Post 请求中，请求参数一般放在请求体 `Request body` 中,HttpRunner 支持 `form-data` 和 `json` 两种数据格式来传递参数。

这里我们的测试接口如下：请求类型为 `POST`

```
http://httpbin.org/post
```

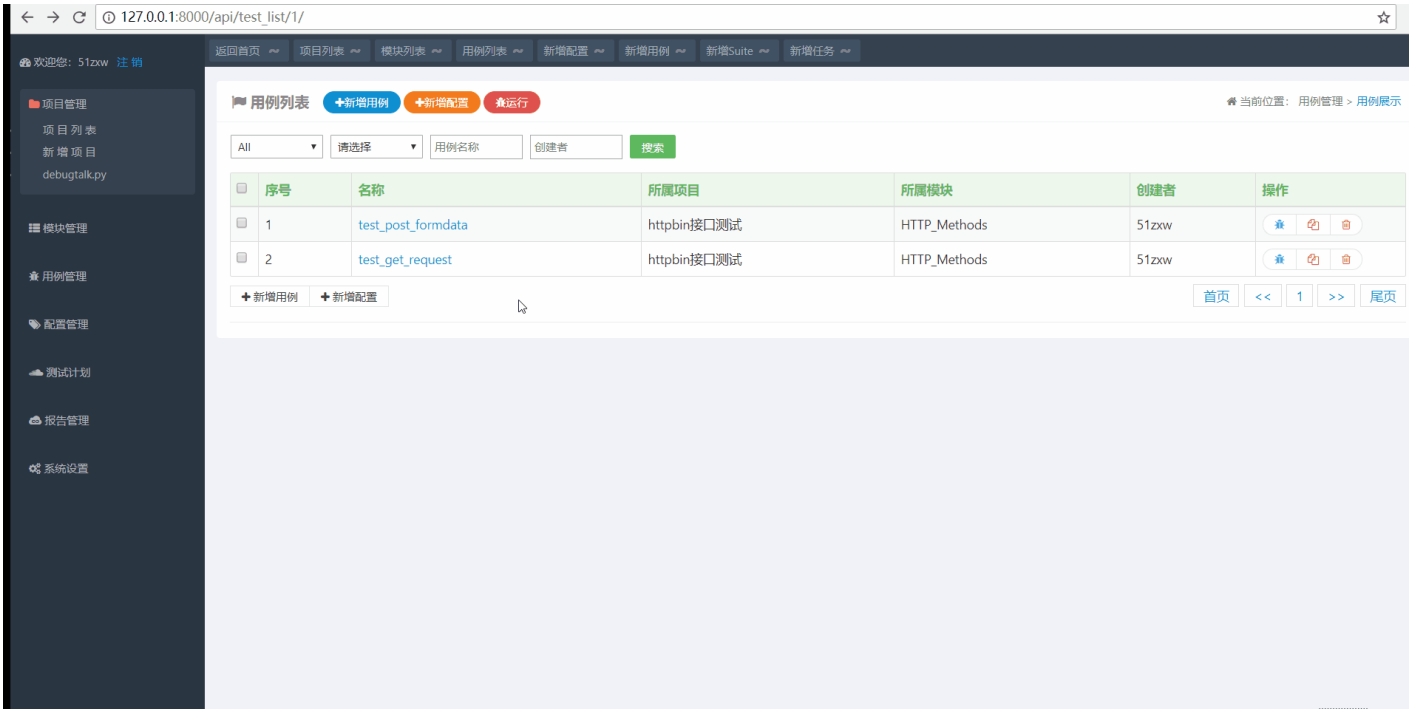
form-data

首先创建用例名称为 `test_post_formdata` 用例编辑界面 `Type` 选择 `data`,然后点击 `add data` 按钮，在下面表单中输入参数名称和值即可。操作过程如下：



json

传递 Json 参数与 form-data 方法类似，选择 Type 时选 Json 创建用例 test_post_jsondata 操作流程如下：

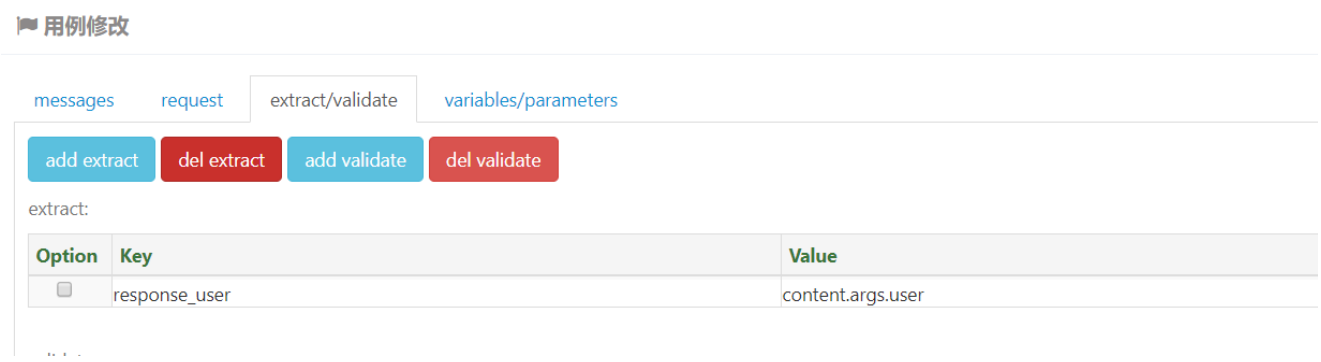


获取返回结果

HttpRunnerManager 提供了 extract 功能来从返回结果中提取我们需要的内容。例如前面用例 test_get_request 执行之后返回结果如下：


```
{
  "args": {
    "user": "51zxw"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Connection": "close",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.18.4",
    "User-Headers": "zxw2018"
  },
  "origin": "110.52.4.234",
  "url": "http://httpbin.org/get?user=51zxw"
}
```

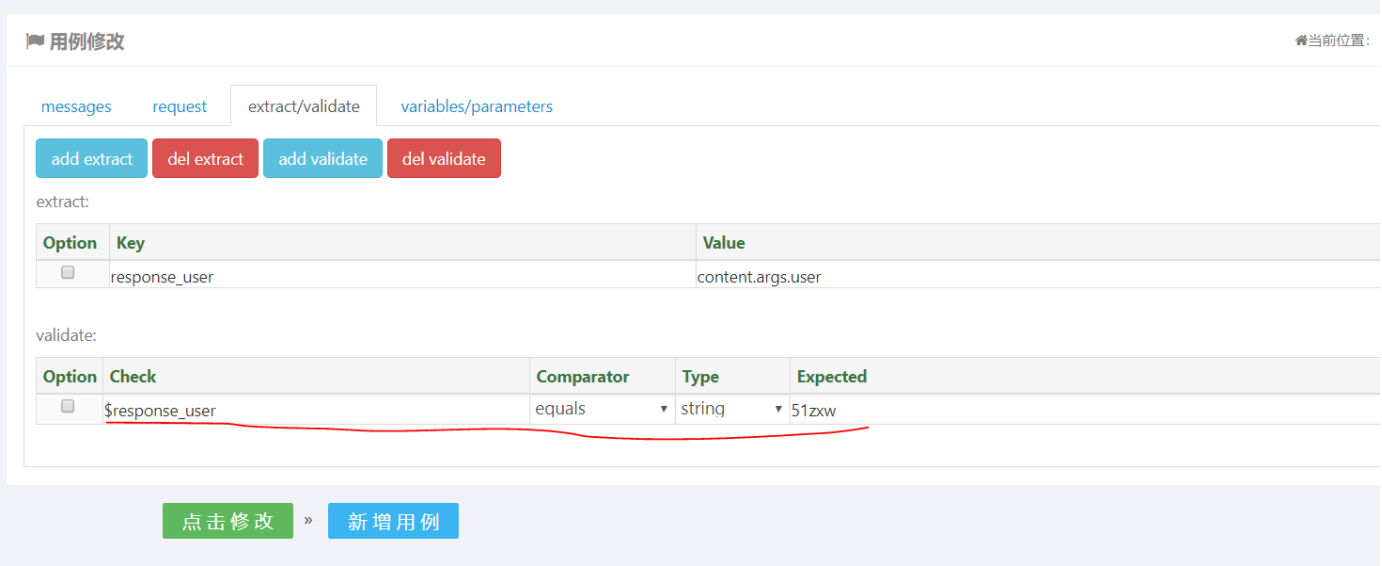

如果我们想提取 user 值放在一个变量里面，那么可以使用 extract 来提取。在用例编辑界面点击 extract/validate 标签，然后点击 add extract 按钮，进行如下配置即可：



其中 Key 的值 response_user 就是将返回值存储的变量名，content.args.user 表示从返回内容中提取 args 属性中的 user 值。下一步再断言设置中，我们可以验证是否获取正确。

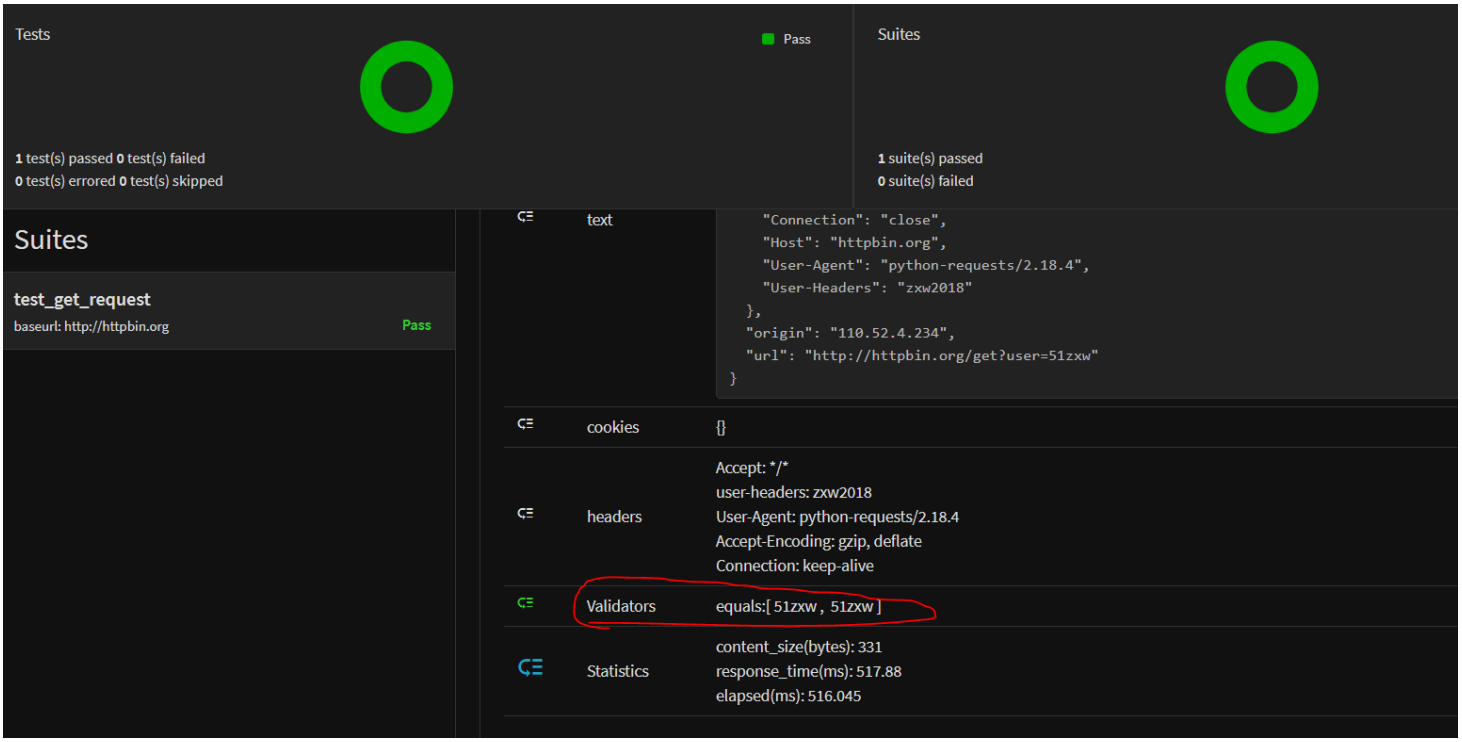
断言设置

结合前面提取返回值的内容，我们设置断言验证返回的 user 值是否和我们预期的一样，首先点击 add validate 然后可以进行如下设置：



上面的$response_user 表示引用我们之前设置的获取返回值的变量，Comparator 表示匹配规则，匹配规则有很多可以点击下拉菜单查看 Expected 值表示我们的期望值。

执行用例之后，我们可以看到在测试报告中，断言验证是通过的。



当然如果还想添加其他断言规则，就继续点击 `add validate` 例如设置验证响应状态码为 `200` 可以进行如下设置

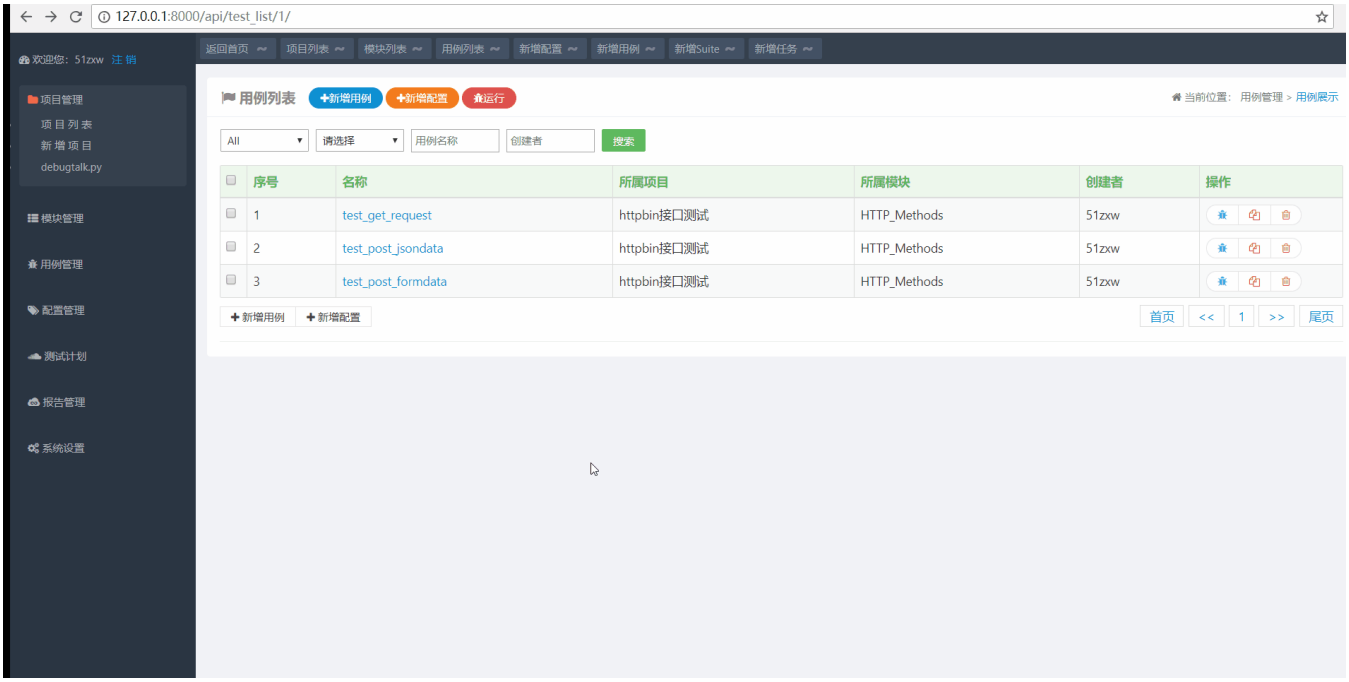
注意：`200` 数值类型为 `int`

用例组合

有时候我们想把一些单个的接口按照指定顺序组合成为一个业务逻辑模块。比如用户模块，把用户注册、用户登录，用户退出几个用例封装成一个业务逻辑模块，从而形成接分层测试。

实践案例：

将之前的三个用例 `test_get_request`,`test_post_formdata`、 `test_post_jsondata` 封装为一个用例 `test_method_group` 然后执行 `test_method_group`。具体操作如下：



从图中我们可以看出新建一个用例 `test_method_group`，然后可以自由添加单个接口用例，并且可以自由调整用例执行顺序，以及删减用例。

注意：新建组装的用例时，只能选择单请求的用例进行拼接，不可选择已组装过的用例。例如再新建一个用例，把 `test_method_group` 放进去是不行的。 只能选择包含"不包含别的用例的用例"。

测试计划

测试套件

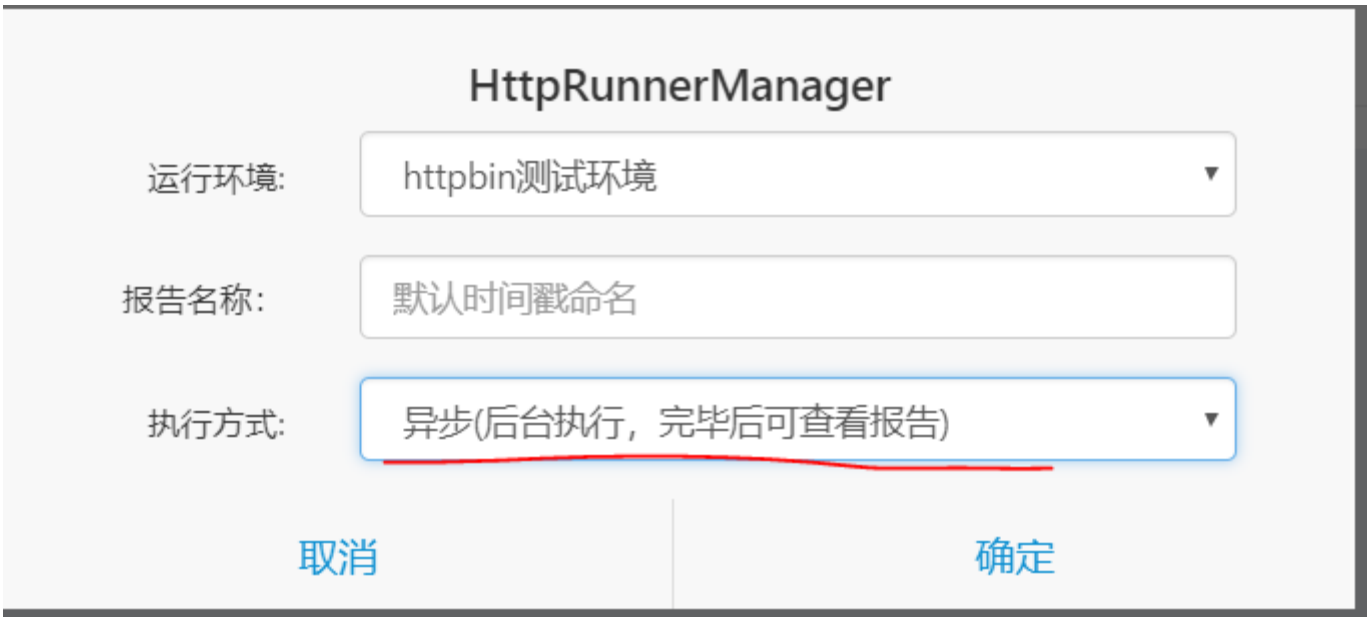
测试套件(test suite)和我们上面讲的组合用例类似，我们可以把单个用例按照业务逻辑进行组合运行，和组合用例不同的是：测试套件可以包含组合用例，但是组合用例不能包含组合用例。 测试套件是对测试用例的更高一层的封装。

实践案例

创建测试用例集 `suite_test_methods` 包含测试用例 `test_get_request` 和组合用例 `test_method_group` 然后执行查看结果。 操作过程如下：

从上图中我们可以看到创建的测试套件成功执行，加载的测试套件也可以任意调整执行顺序。相关的数据配置会自动从用例的配置中读取，无需再单独配置参数。

上面我们选择的是同步执行方式，我们可以选择异步执行方式，可以在后台执行，然后生成测试报告。



在异步执行之前我们需要先启动支持异步的相关服务：

1.启动 RabbitMQ Server

2.进入到 HttpRunnerManager 目录，启动 worker

```
python manage.py celery -A HttpRunnerManager worker --loglevel=info
```

3.启动任务监控后台

```
celery flower
```

celery 简介

Celery 是一个异步任务队列/基于分布式消息传递的作业队列。它侧重于实时操作，但对调度支持也很好。Celery 用于生产系统每天处理数以百万计的任务。Celery 是用 Python 编写的，但该协议可以在任何语言实现。它也可以与其他语言通过 webhooks 实现。

执行完成之后我们可以在【报告管理】——【查看报告】中看到生成的测试报告。

扩展资料：[Python 并行分布式框架 Celery]

定时任务

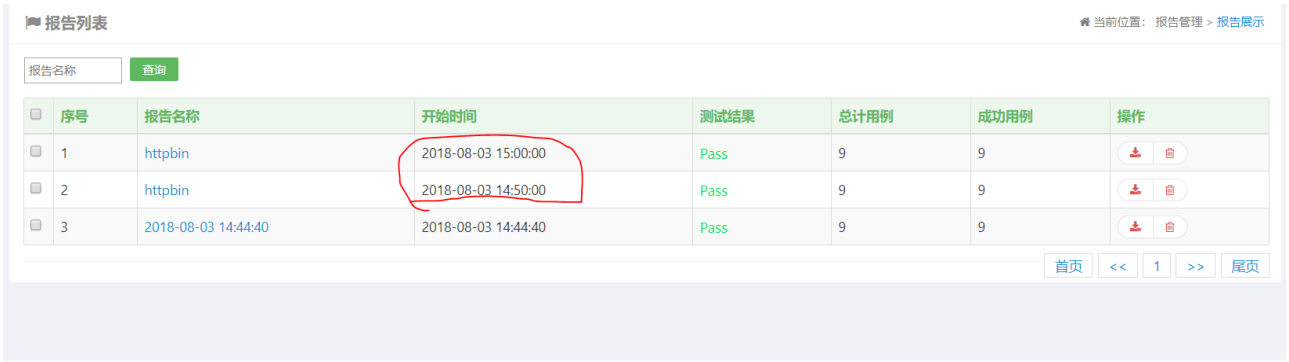
测试套件还支持定时任务，这样方便进行回归测试。 定时任务需要启动定时任务监听器，具体如下：

```
python manage.py celery beat --loglevel=info
```

平台界面设置步骤为：点击菜单栏左侧【测试计划】-【定时任务】 进入到【系统设置】界面，可以进行如下设置：

上面定时配置 `*/10 * * * *` 表示每 10 分钟执行一次，使用的是 `crontab` 表达式

执行结果如下图所示，可以看到是每隔 10 分钟执行一次。



crontab 格式

通过 crontab 命令，我们可以在固定的间隔时间执行指定的系统指令或 shell script 脚本。时间间隔的单位可以是分钟、小时、日、月、周及以上的任意组合。

分 时 日 月 星期

- 第 1 列分钟 0～59
- 第 2 列小时 0～23（0 表示子夜）
- 第 3 列日 1～31
- 第 4 列月 1～12
- 第 5 列星期 0～7（0 和 7 表示星期天）

crontab 设置案例

每隔 1 分钟执行一次

```
* * * * *
```

每 30 分钟运行一次：

```
*/30 * * * *
```

每隔 1 小时执行一次

```
* */1 * * *
```

每周一到周五早上八点运行

```
* 8 * * 1-5
```

相关资料：crontab 表达式

配置管理

前面我们用例的各个参数都是配置在各自用例之中的，如果参数有变化则需要打开对应请求用例来修改，当用例数量较多时，这样操作会比较低效。那么该如何有效解决这个问题呢?

使用 HttpRunnerManager 的**配置管理**工具就可以比较好的解决这个问题。我们可以将一些公共的变量，参数、方法、请求头信息都存储在一个配置模块中，这样维护和使用就非常便利。

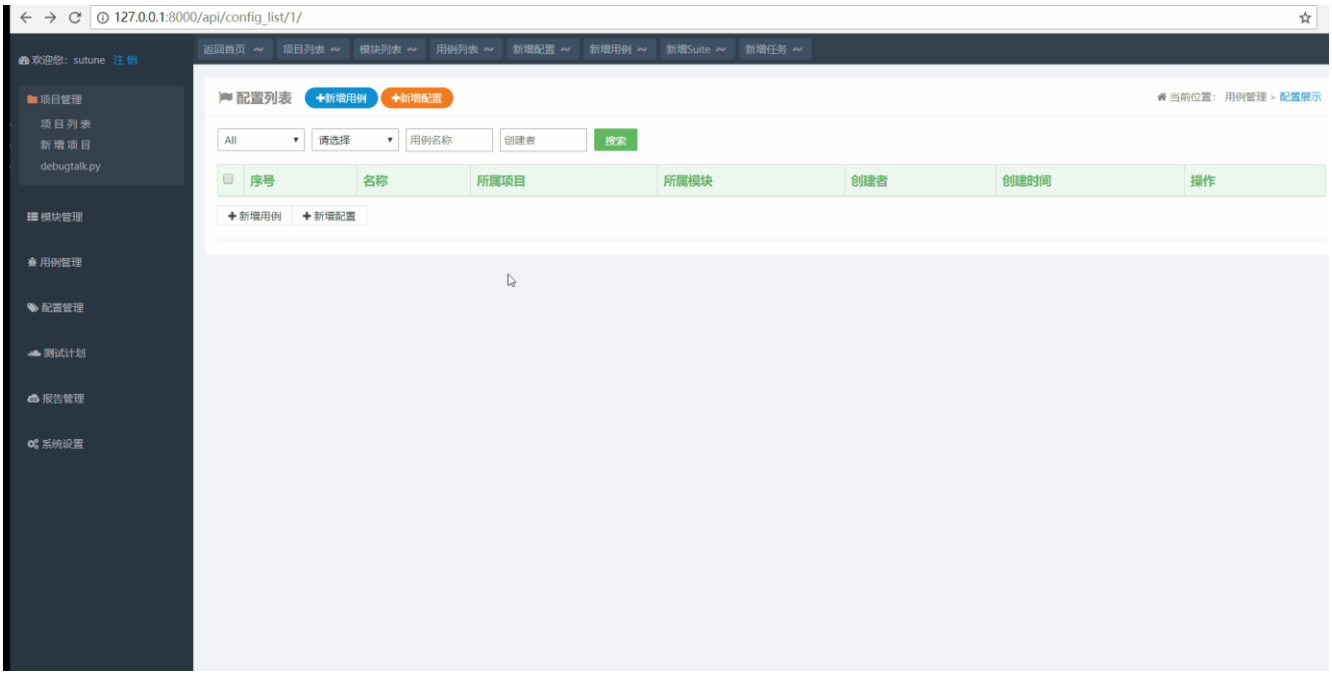
支持的配置类型包括以下几类：

- `Variables` 变量类型
- `Parameters` 参数类型
- `Hooks` 方法类型
- `Request` 请求类型

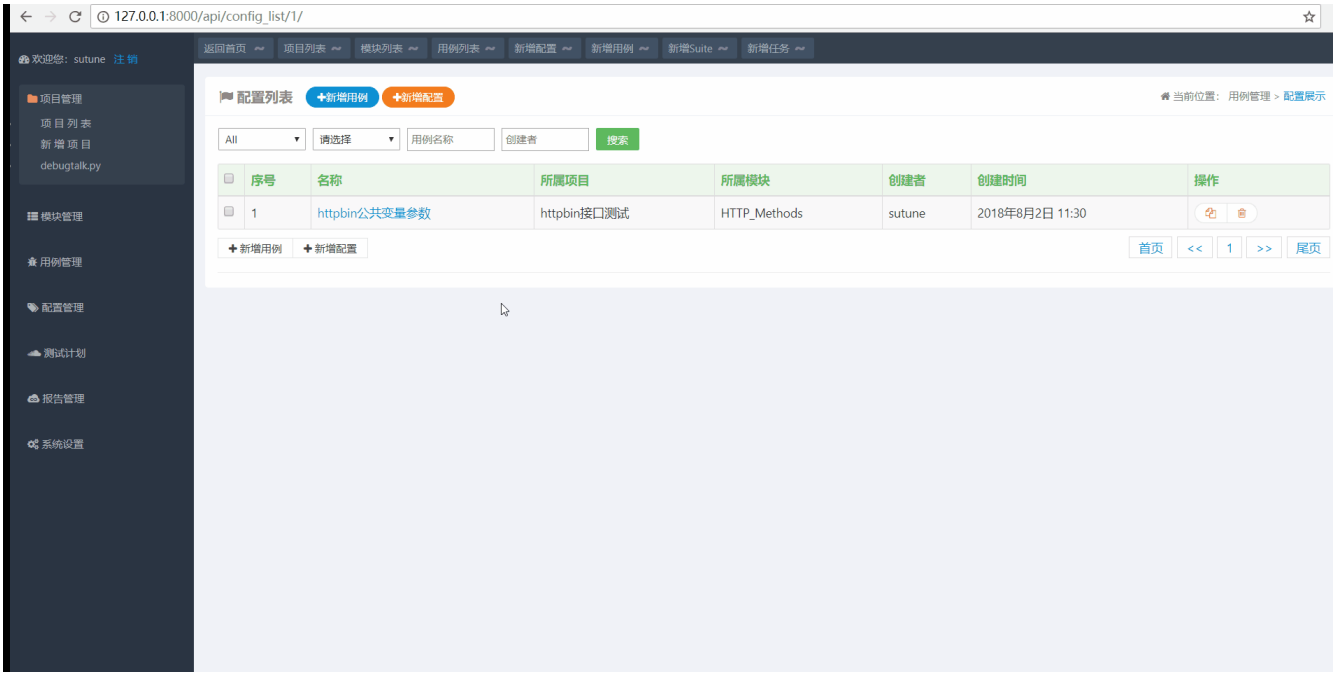
点击菜单栏左侧的**配置管理** 然后点击**新增配置** 或者直接顶部**新增配置**快捷入口，就可以创建配置。 这里演示 `Variables` 和 `Parameters` 配置

variable 配置

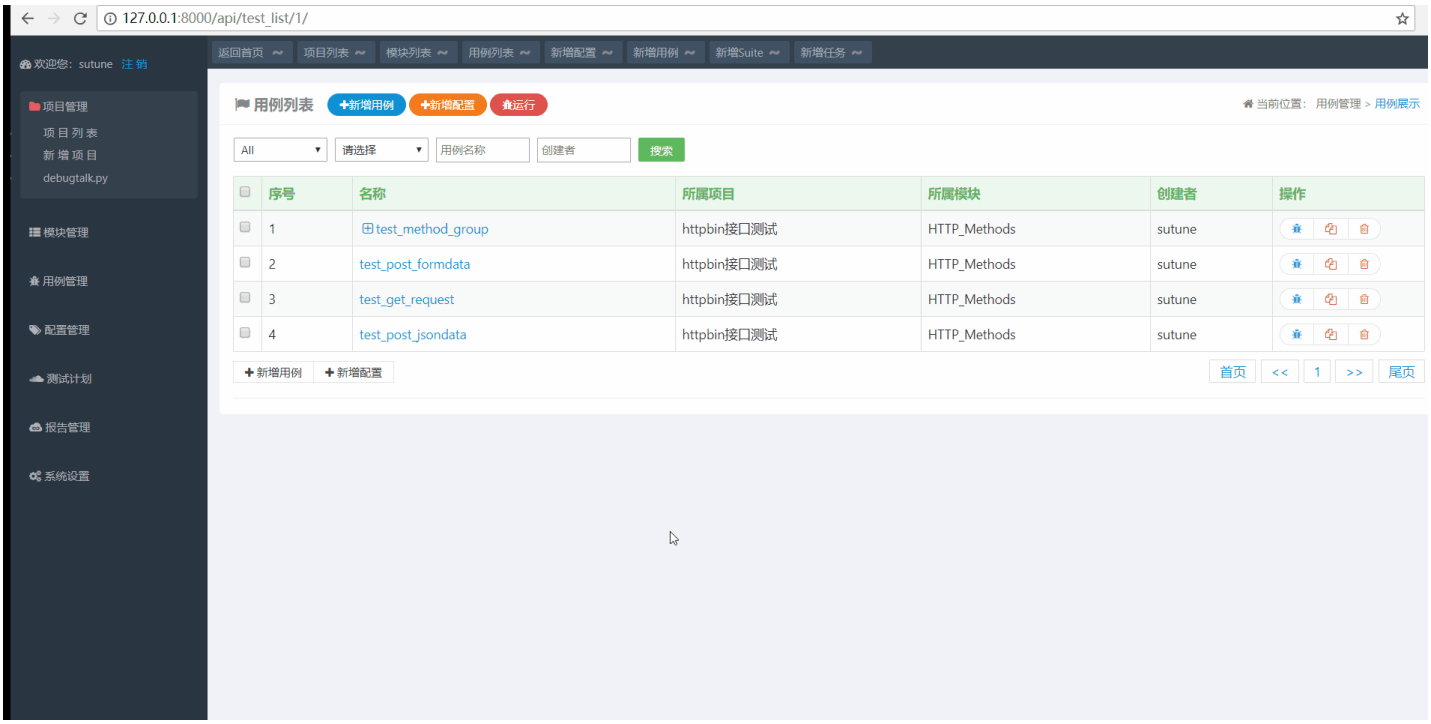
如下图所示：创建公共变量 `com_user` 值设为 `zxw2018` 操作过程如下：



变量创建完成之后,接下来要引用这个公共变量，在用例 test_get_request 和 test_post_formdata 分别添加公共变量 com_user,配置如下：



最后开始引用配置，在用例 test_method_group 中引用该配置，然后运行即可。操作过程如下：



从上图可以看出，我们设置的配置变量已经在用例生效，下次想修改变量值，只需要修改配置就可以了。

注意：

当用例 1 关联了配置(config)1，用例 2 关联了配置(config)2 时，现在新建一个用例 3，将用例 1 和用例 2 都组合进用例 3 中，并且用例 3 关联配置(config)3，此时执行用例 3，配置(config)1 和 配置(config)2 会被使用么?

答案是不会，执行用例 3 时，只有配置(config)3 会被使用. 在执行的过程中，用例 1 和用例 2 所要引用的变量，都只会去配置(config)3 中找。

Parameters 配置

对用例中的某些请求参数，有时想要测试多种输入情况，可以为每种情况都编写独立的用例或配置. 但是这样会比较麻烦，另一种方式是将原来的 `variables` 变量定义改为 `Parameters` 参数列表来定义. 这个列表中定义需要传入的参数. 然后使用这些参数分别执行一次，也就是参数化。

案例 1——单个参数设置

针对用例 `test_get_request` 设置参数 `para_user` 分别取值为`['zxw2016','zxw2017','zxw2018']`针对这三个不同的参数来进行接口测试。

配置编辑

messages | variables/parameters/hooks | request

add variables | del variables | add param | del param | add hooks | del hooks

Variables:

Option	Key	Type	Value
☐	com_user	string	zxw2018

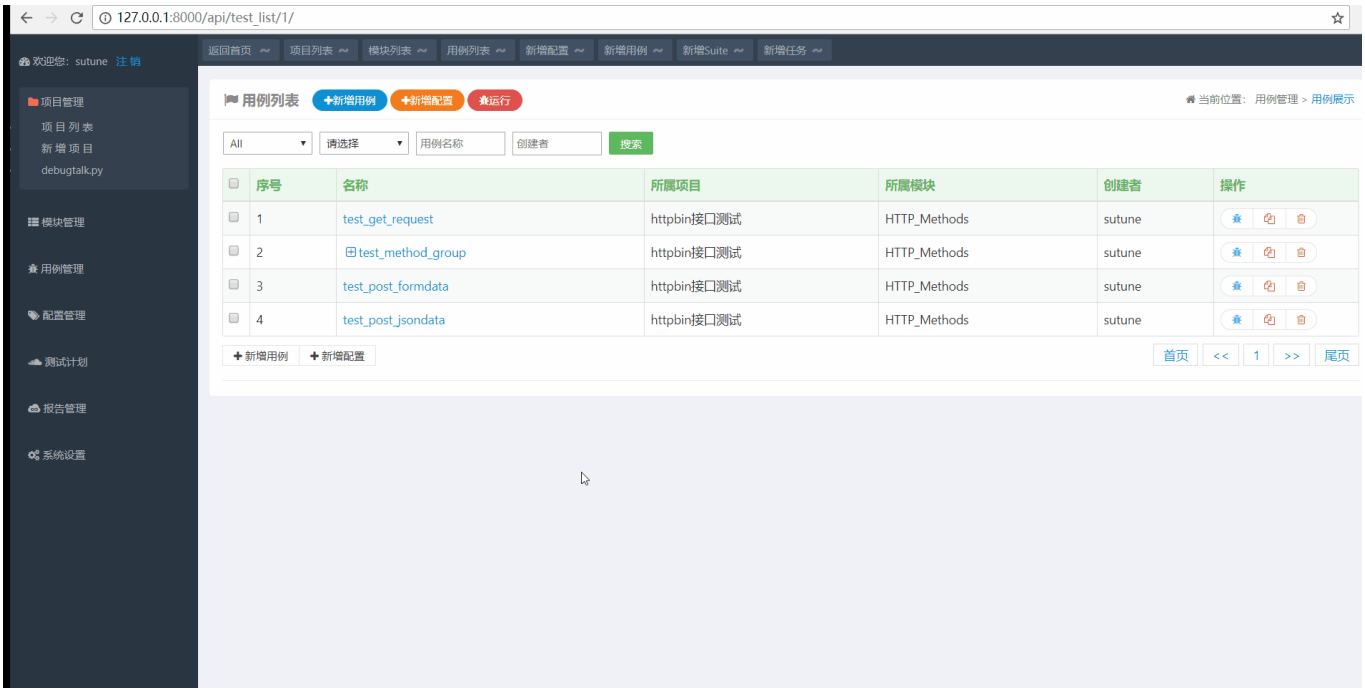
parameters:

Option	Key	Value
☐	para_user	['zxw2016', 'zxw2017', 'zxw2018']

hooks:

Option	setup_hooks	teardown_hooks

配置好之后，接下来点击用例 `test_get_request` 然后添加参数配置，操作如下所示：



从上图我们可以看出，用例遍历执行了我们设定的三个参数。

案例 2——多个参数

对于同时存在多个参数列表，则需要对其排列组合的每一种情况都执行一次,也就是笛卡尔乘积

例如有 xx 和 yy 两组参数,对应的变量分别如下

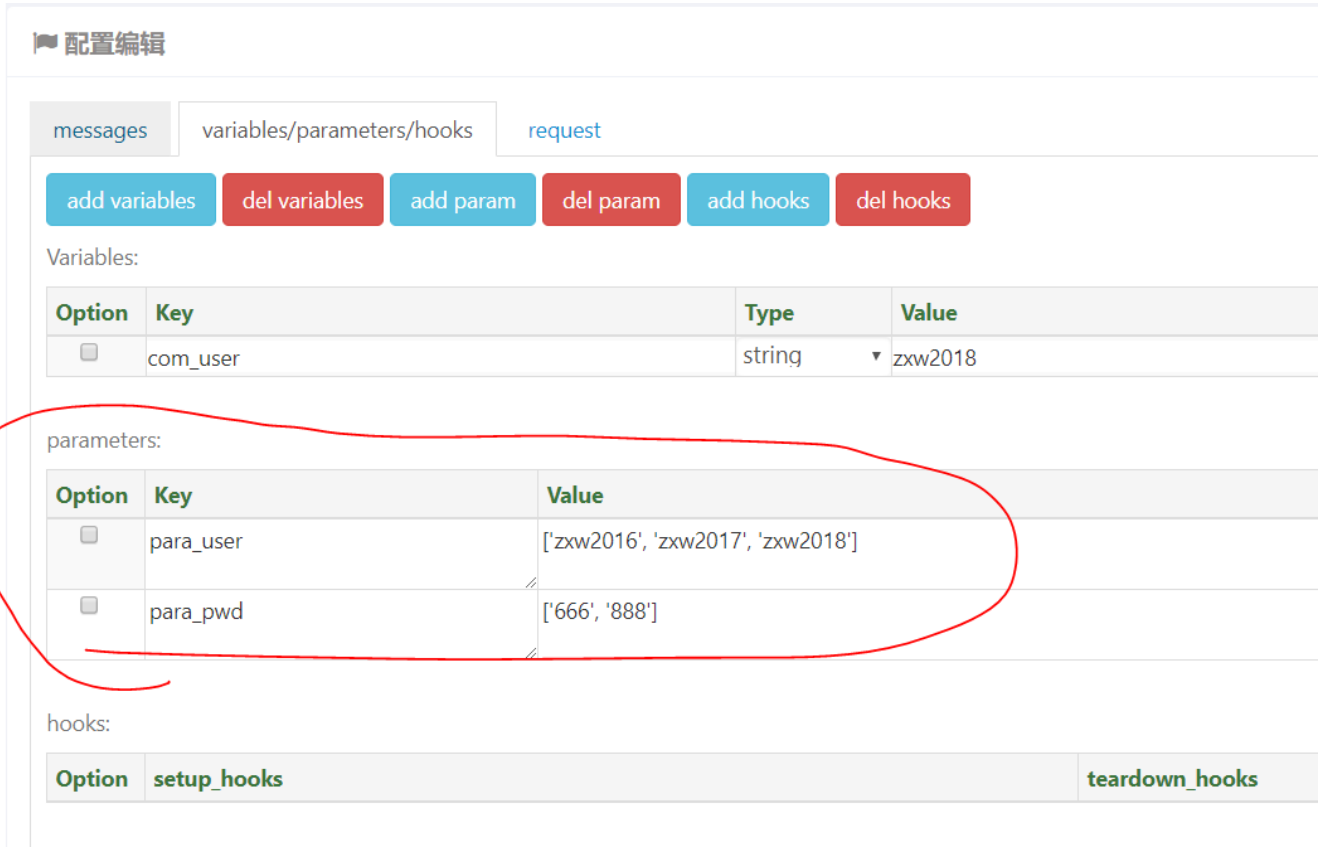
```
xx : [xxvalue1, xxvalue2]
yy : [yyvalue1, yyvalue2, yyvalue3]
```

那么最终会按以下六种情况各执行一次用例:

```
xx=xxvalue1, yy=yyvalue1
xx=xxvalue1, yy=yyvalue2
xx=xxvalue1, yy=yyvalue3
xx=xxvalue2, yy=yyvalue1
xx=xxvalue2, yy=yyvalue2
xx=xxvalue2, yy=yyvalue3
```

所以，当对多个变量使用 parameters 参数列表时，特别需要事先考虑清楚用例将会被循环的次数，避免不必要的测试时间的消耗.

例如在配置表中我们再增加一个参数 para_pwd 取值为['666', '888']



然后在用例中引用变量 para_pwd

用例修改

messages | request | extract/validate | variables/parameters

URL /get | Method GET | Type params | add params | del params | add headers | del headers

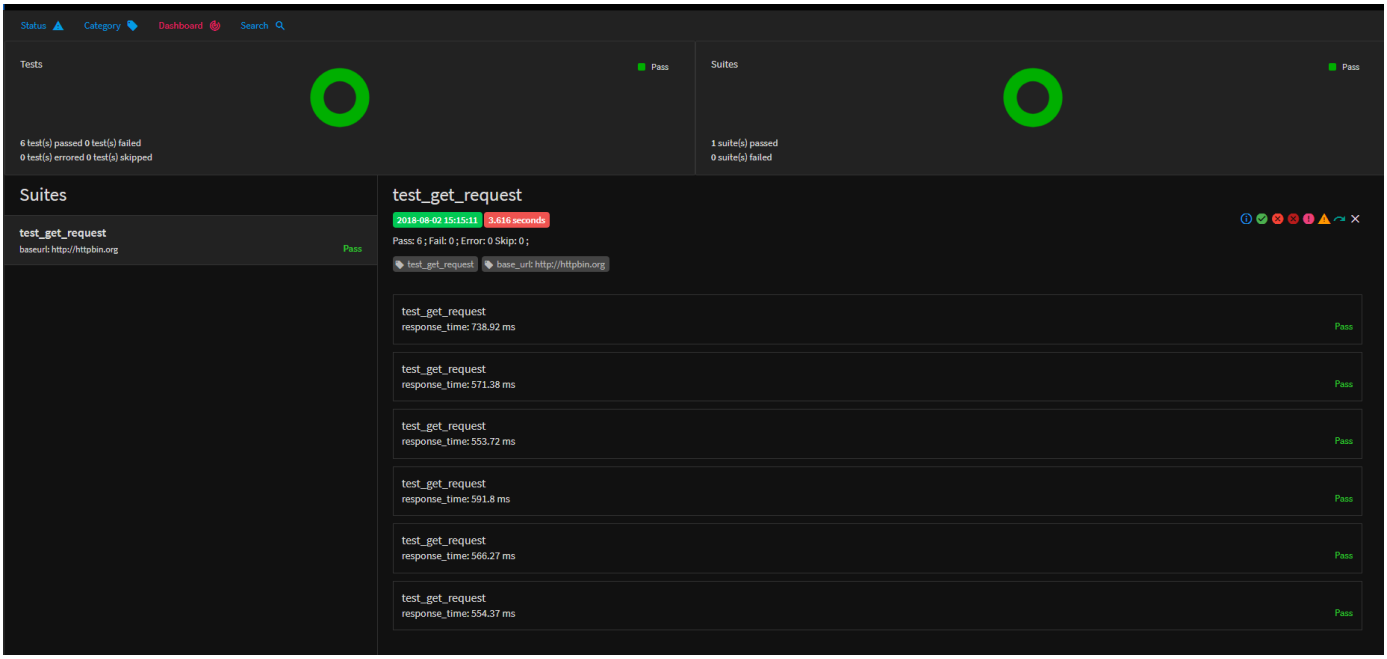
request:

Option	Key	Type	Value
	para_user	string	$para_user
	para_pwd	string	$para_pwd
	com_user	string	$com_user
	user	string	51zxw

headers:

Option	Key	Value

最后执行的结果为 6 次。



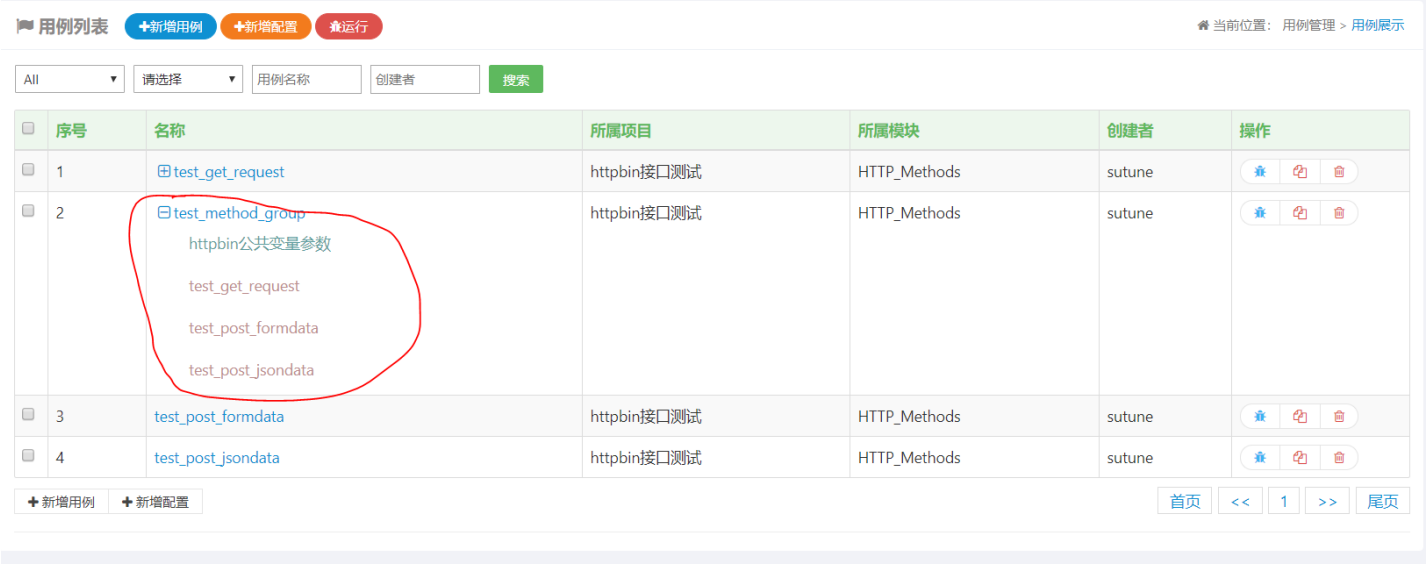
参数组合为：

```
zxw2016,666
zxw2016,888
zxw2017,666
zxw2017,888
zxw2018,666
zxw2018,888
```

配置参数的作用域

当组合用例中的单个用例引用了配置参数，那么该组合用例下所有的用例都会按照参数组合来执行对应的次数。

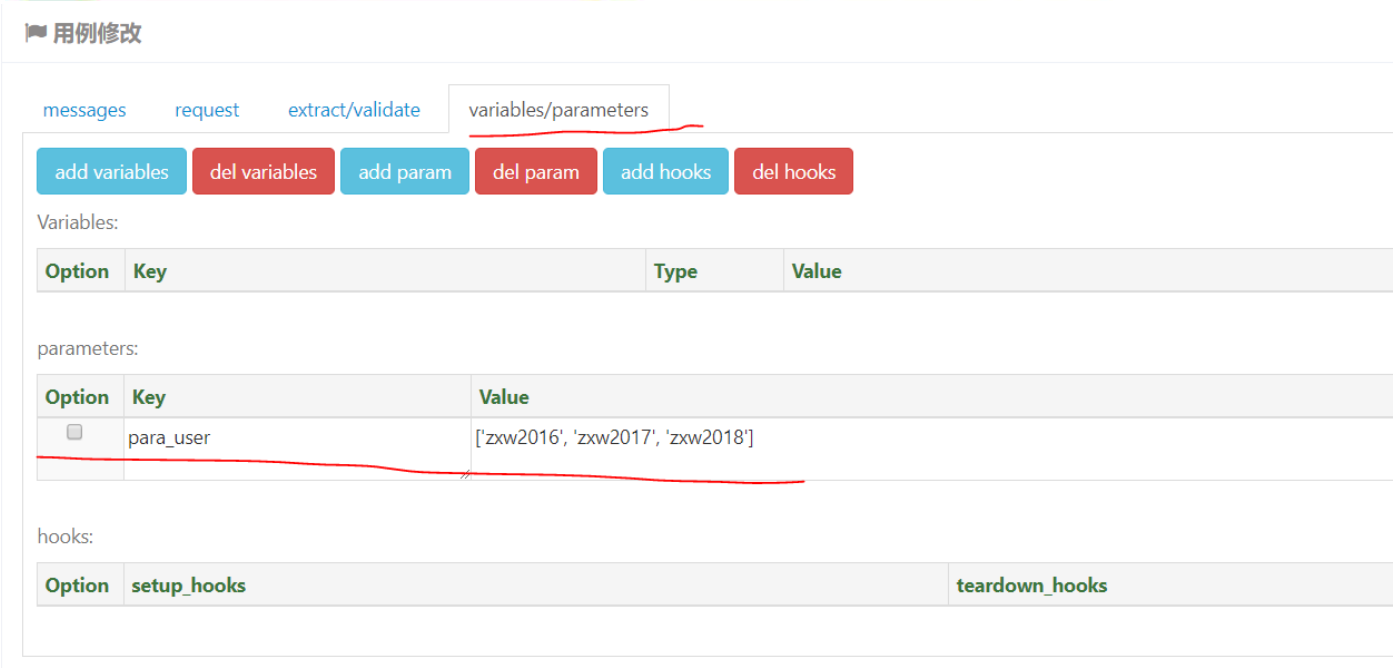
例如在 test_method_group 中用例 test_get_request 引用了配置参数,那么所有的用例都会执行 6 次(上面案例中 para_user 和 para_pwd 的笛卡儿积)，总共为 3*6=18 次。



这样带来一个问题也就是有些没有设置参数的用例也会重复运行，这对整体运行效率会有比较大的影响，

如果只想针对指定的用例遍历设定的参数，那么需要将参数配置在用例中，而不是在公共配置模块中。

例如在用例 test_get_request 配置参数 para_user 取值为['zxw2016', 'zxw2017', 'zxw2018'] 同时删除之前配置模块的对应的参数，以及引用的参数。



然后执行用例集 `test_method_group`，那么 `test_get_request` 会根据设置的三个不同参数运行 3 次，其他两个接口运行 1 次。

键值对参数

对于参数间有对应关系的(如用户名:密码)可以按如下方式定义，这样就避免了无效的排列组合:

```
xx-yy: [[xxvalue1, yyvaule1],  [xxvalue2,yyvalue2]]
```

组合结果:

```
xx=xxvalue1, yy=yyvalue1
xx=xxvalue2, yy=yyvalue2
```

在使用多个账户/密码进行测试时，常用这种方式.

例如在用例 `test_post_formdata` 中配置如下参数

```
#用户名密码组合
51zxw2016,666
51zxw2018,888
```

在用例中进行如下配置

用例修改

messages | request | extract/validate | variables/parameters

add variables | del variables | add param | del param | add hooks | del hooks

Variables:

Option	Key	Type	Value

parameters:

Option	Key	Value
☐	para_user-para_passwd	[['51zxw2016', '666'], ['51zxw2018', '888']]

用例修改

messages | request | extract/validate | variables/parameters

URL /post | Method POST | Type data | add data | del data | add headers | del headers

request:

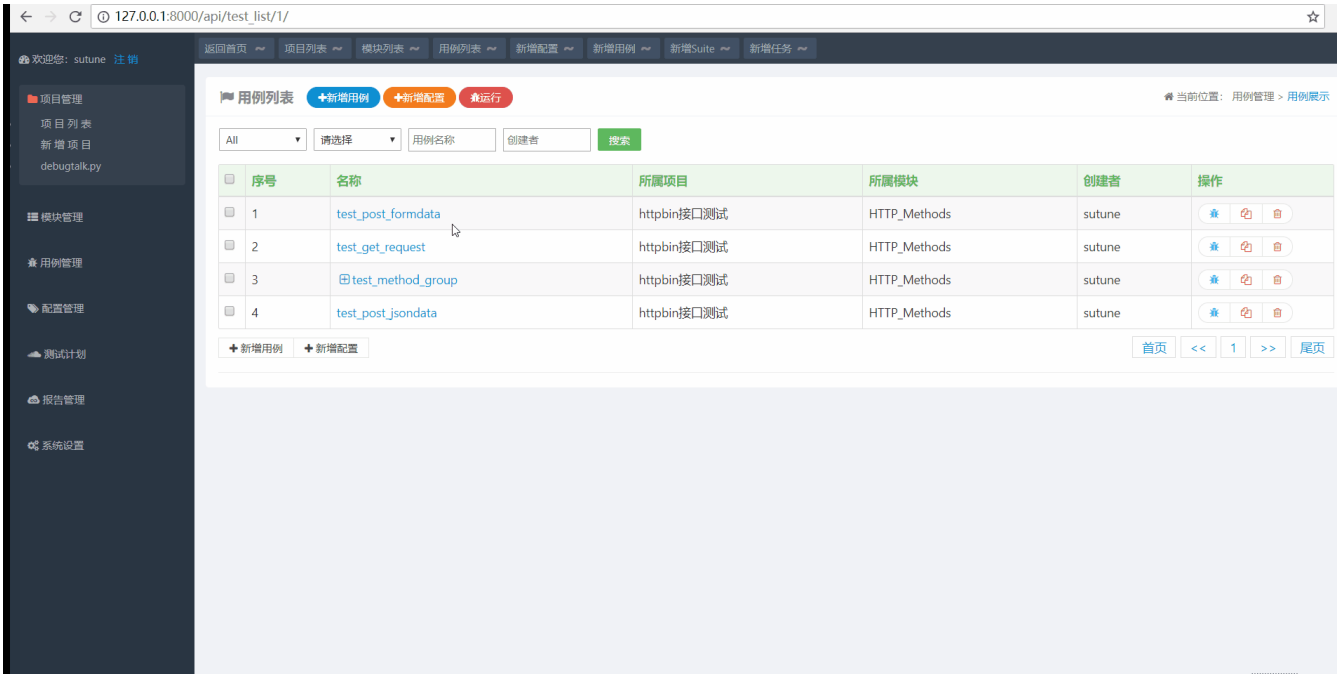
Option	Key	Type	Value
☐	passwd	string	$para_passwd
☐	user	string	$para_user

headers:

Option	Key	Value

点击修改 » 新增用例

运行测试用例：



从运行的报告中我们可以看到，参数按照我们设定的运行。

HttpRunnerManager 进阶应用

自定义辅助函数

在一些比较特殊的接口测试过程中，有时需要做一些比较复杂的逻辑处理，比如加解密。针对这些接口我们需要编写一些辅助函数来完成测试。

案例：接口参数 md5 加密

测试登录接口:`http://httpbin.org/post`，参数除了 `user` 和 `passwd` 还需要一个 `sign` 参数 `sign` 生成规则为用户名+密码然后进行 `md5` 加密。

在项目管理菜单栏中点击 `debugtalk.py` 然后编辑如下代码：

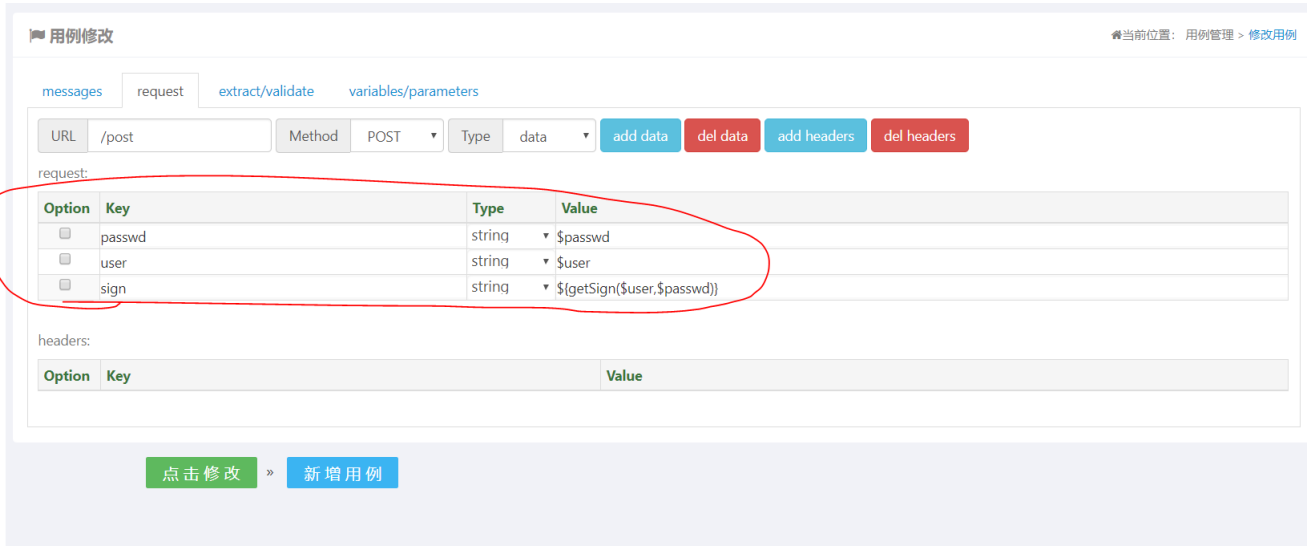
```
# debugtalk.py

import hashlib
```

```
def getSign(user,passwd):
    str=user+passwd
    md5=hashlib.md5()
    md5.update(str.encode(encoding='utf-8'))
    sign=md5.hexdigest()
    return sign
```

上面代码代表根据用户名密码生成 md5 摘要信息，并返回结果。

创建用例 test_getSign 配置如下：



用例修改

messages | request | extract/validate | variables/parameters

add variables | del variables | add param | del param | add hooks | del hooks

Variables:

Option	Key	Type	Value
☐	user	string	51zxw
☐	passwd	string	8888

parameters:

Option	Key	Value

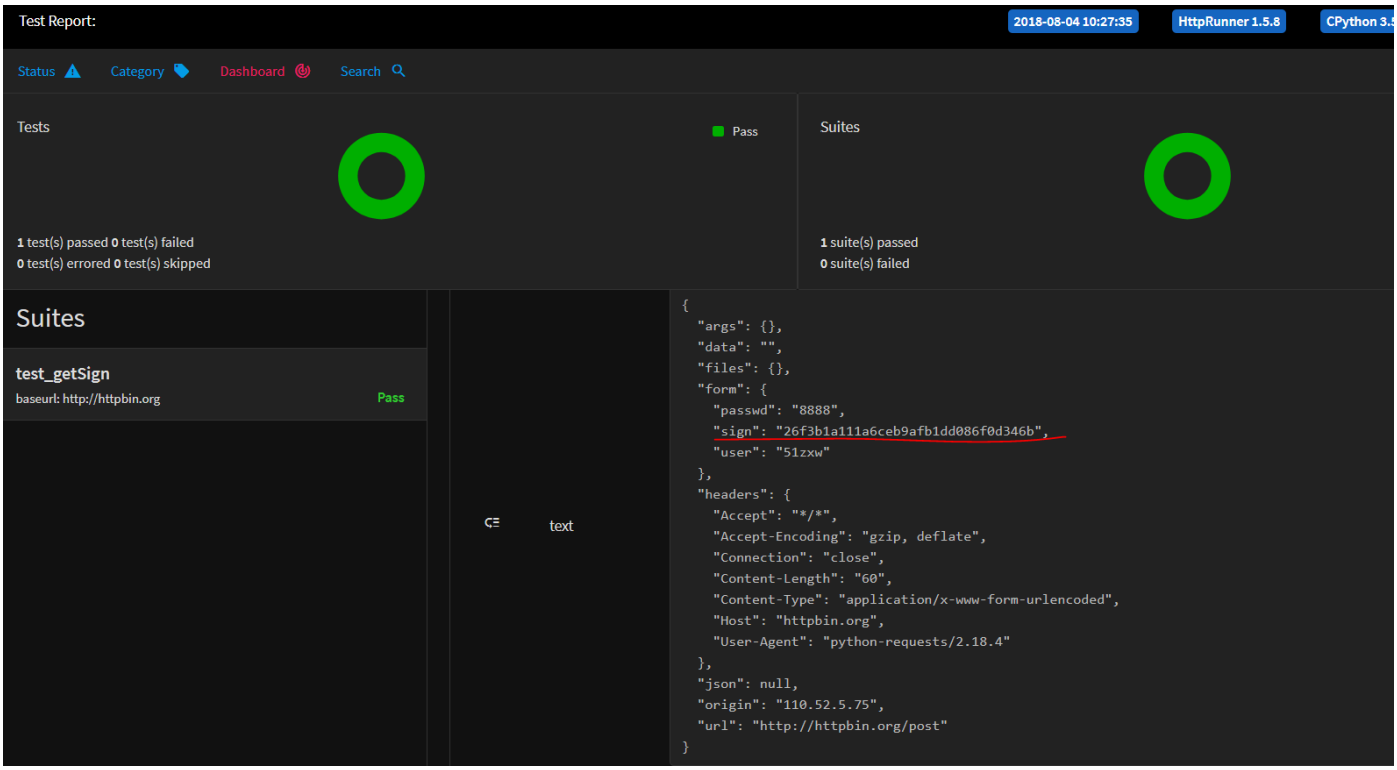
hooks:

Option	setup_hooks	teardown_hooks

点击修改 » 新增用例

从上面的配置可以看出，方法引用格式为：`${function($para)}`

运行结果如下：



从上面的报告可以看出，`sign` 的值参数为 `md5` 算法生成的结果。

Hook

有些接口测试前后需要进行一些特殊的处理，比如初始化操作或者执行完成之后等待操作。 相当于 unittest 中的 setUp 和 tearDown 方法，HttpRunner 也支持类似于这样的方法。其中 Hook 功能就支持这样的操作。

- setup_hooks：在 HTTP 请求发送前执行 hook 函数，主要用于准备工作；也可以实现对请求的 request 内容进行预处理。
- teardown_hooks：在 HTTP 请求发送后执行 hook 函数，主要用于测试后的清理工作；也可以实现对响应的 response 进行修改，例如进行加解密等处理。

实践案例

设置接口请求之后如果响应状态码为 200 就等待 0.1s 否则就按照设定的时间等待。

在 debugtalk.py 创建辅助函数 sleep() 定义如下：

```
import time

def sleep(response,t):
    if response.status_code==200:
        time.sleep(0.1)
    else:
        time.sleep(t)
```

然后在用例 test_get_request 中进行设置${time($response,2)}即可。

任务监控

任务监控可以查看节点的状况（包括处理的队列信息等）和 task 的执行情况

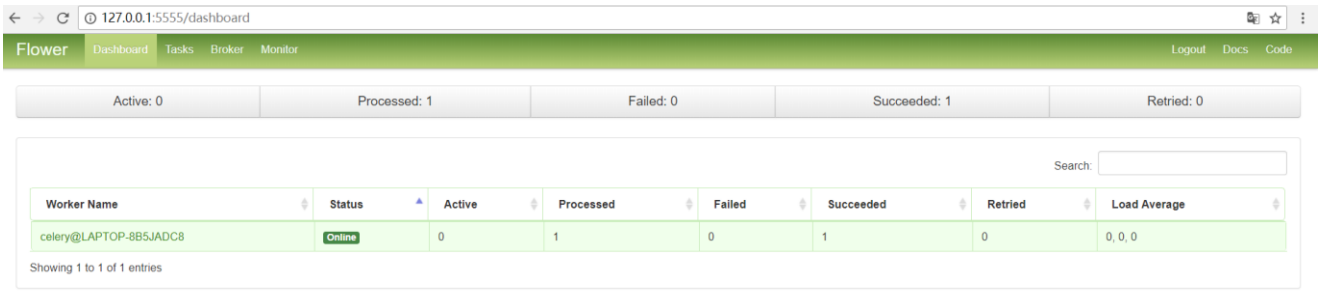
Tips：这里需要做一个小小的修改，因为作者把地址配置按照他自己的运行环境写死了。修改方式为：打开文件 D:\HttpRunnerManager\templates\base.html

将 192.168.91.45 修改为：127.0.0.1

```
<!-- <li><a href="http://192.168.91.45:5555/dashboard">任 务 监 控</a></li>-->
```

```
<li><a href="http://127.0.0.1:5555/dashboard">任 务 监 控</a></li>
```

设置好之后打开即可看到如下页面：



线上部署

当前系统是部署在本地环境，如果想部署到线上环境，那么可以参考：Django 快速部署简约版 v3.0。

参考文档

- http://cn.httprunner.org/testcase/structure/
- https://www.cnblogs.com/junrong624/p/4121656.html