mysql数据库配置

实战项目采用mysql数据库，

准备工作

自行搭建mysql服务5.7

准备一个可以外部访问的账户，root也可以

创建数据库

不同于sqlite,数据选择mysql时 django不会帮你创建数据库，只会关联已有的数据库

```
CREATE DATABASE `db_name` DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci;
```

安装pymysqlclient模块

```
 pip install mysqlclient   #链接mysql数据库需要此模块,此模块安装不了的，参考附录的方法
```

对应的数据库参考配置如下：

项目/settings.py

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'course_autotp',  #数据库名称
        'USER': 'youruser',       #用户名
        'PASSWORD': 'youpsd',     #密码
        'HOST': '127.0.0.1',
        'PORT': 'yourport',
        'TEST':{
            'CHARSET': 'utf8',
            'COLLATION': 'utf8_general_ci',
        }
    }
}
```

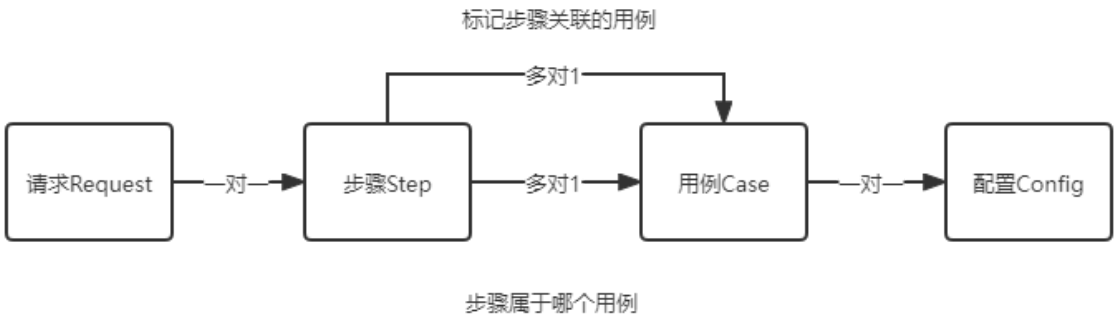
数据结构设计

1个web系统，最核心的当属数据库结构了，因为一切都围绕数据计算开始。

接口测试平台，最核心的除了接口运行框架就数数据结构了，因为我们需要将用例的数据持久化到数据库中，方便读取修改与存储。

如果我们要以httprunner为接口用例运行框架，则需要按照期yaml/json的数据结构来设计数据库。

经过设计，总结出了一套数据库模型关系图的总体结构



1对1关系

模型中用到了两个知识点：1对1关系和json字段，我们先看下1对1关系：

从概念上理解，1对1就是1个模型关联另一个模型，他们之间的关系是1对1

这里可以理解为模型关系的扩展。

一对一关联与多对一关联非常类似。若在模型中定义了 `OneToOneField`，该模型的实例只需通过其属性就能访问关联对象。

例如:

```
class EntryDetail(models.Model):
    entry = models.OneToOneField(Entry, on_delete=models.CASCADE)
    details = models.TextField()

ed = EntryDetail.objects.get(id=2)
ed.entry # Returns the related Entry object.
```

不同点在于“反向”查询。一对一关联所关联的对象也能访问 `Manager` 对象，但这个 `Manager` 仅代表一个对象，而不是对象的集合(QuerySet):

```
e = Entry.objects.get(id=2)
e.entrydetail # returns the related EntryDetail object
```

若未为关联关系指定对象，Django 会抛出 `DoesNotExist` 异常。

实例能通过为正向关联指定关联对象一样的方式指定给反向关联:

```
e.entrydetail = ed
```

模型代码设计

其中，request，step和config参考HR对应部分的字段，由于其中会出现很多嵌套字段，所以1层的字段就用json数据类型来代替。

```
class Config(models.Model):
    name = models.CharField('名称', max_length=128)
```

```
    base_url = models.CharField('IP/域名', max_length=512, null=True, blank=True)
 # 可Null,可空白
    variables = models.JSONField('变量', null=True)
    parameters = models.JSONField('参数', null=True)  # 用于参数化
    verify = models.BooleanField('https校验', default=False)
    export = models.JSONField('用例返回值', null=True)

    def __str__(self):
        return self.name


class Step(models.Model):
    # 同个模型中，两个字段关联同1个模型，必须指定related_name,且名字不能相同
    # 属于哪个用例
    belong_case = models.ForeignKey('Case', on_delete=models.CASCADE,
related_name='teststeps')
    # 引用哪条用例
    linked_case = models.ForeignKey('Case', on_delete=models.SET_NULL,
null=True, related_name='linked_steps')
    name = models.CharField('名称', max_length=128)
    variables = models.JSONField('变量', null=True)  #默认sqlite数据库不支持json字段
    extract = models.JSONField('请求返回值', null=True)
    validate = models.JSONField('校验项', null=True)
    setup_hooks = models.JSONField('初始化', null=True)
    teardown_hooks = models.JSONField('清除', null=True)

    def __str__(self):
        return self.name


# 请求
class Request(models.Model):
    method_choices = (  # method可选字段，二维元组
        (0, 'GET'),  # 参数1：保存在数据库中的值，参数2：对外显示的值
        (1, 'POST'),
        (2, 'PUT'),
        (3, 'DELETE'),
    )

    step = models.OneToOneField(Step, on_delete=models.CASCADE,
null=True,related_name='testrequest')
    method = models.SmallIntegerField('请求方法', choices=method_choices,
default=0)
    url = models.CharField('请求路径', default='/', max_length=1000)
    params = models.JSONField('url参数', null=True)
    headers = models.JSONField('请求头', null=True)
    cookies = models.JSONField('Cookies', null=True)
    data = models.JSONField('data参数', null=True)
    json = models.JSONField('json参数', null=True)

    def __str__(self):
        return self.url
```

此外，还需要创建case来统一管理接口用例

```
class Case(models.Model):
    config = models.OneToOneField(Config, on_delete=models.DO_NOTHING)
    suite = models.ForeignKey('Suite', on_delete=models.DO_NOTHING, null=True)
    file_path = models.CharField('用例文件路径', max_length=1000,
default='demo_case.json')

    def __str__(self):
        return self.config.name
```

同步数据库

```
python manage.py makemigrations
python manage.py migrate
```

反向查询概念解析

正向查询

在1对1，1对多，多对多关系中都存在反向查询，若想知道什么是反向查询先了解什么是正向查询

模型查询其关联的项目叫做正向查询（外键定义在模型）

例如：步骤查询其所在的用例

```
#test.py
class TestRelatedQuery(TestCase):
    def setUp(self) -> None:
        # 创建用例
        self.config = Config.objects.create(name='用例1')
        self.case = Case.objects.create(config=self.config)


    def test_step_case(self):
        step1 = Step.objects.create(case=self.case,name='步骤1')
        step2 = Step.objects.create(case=self.case,name='步骤2')
        print(step1.case) # 正向查询
        print(step2.case)  # 正向查询
```

反向查询

反过来，项目查询下面的模型叫做反向查询，通过反向查询的结果QuerySet

方式1：未指定related_name时

modelobj.field_set.all()

方式2：指定related_name时

modelobj.related_name.all()

案例：sqtp/tests.py

```
class TestRelatedQuery(TestCase):
    def setUp(self) -> None:
        # 创建用例
        self.config = Config.objects.create(name='用例1')
        self.suite_conf = Config.objects.create(name='套件')
        self.suite = Suite.objects.create(config=self.suite_conf)
        self.case = Case.objects.create(config=self.config,suite=self.suite)

    def test_step_case(self):
        step1 = Step.objects.create(case=self.case,name='步骤1')
        step2 = Step.objects.create(case=self.case,name='步骤2')
        print(step1.case) # 正向查询
        print(step2.case)  # 正向查询
        print(self.case.teststeps.all())  # 反向查询--外键字段指定了related_name
        print(self.suite.case_set.all())  # 反向查询--外键字段未指定related_name
```

1对1关系的反向查询

此时反向查询的结果不是QuerySet，而是数据对象

```
        print(self.config.case)    # 1对1关系反向查询
        print(self.suite_conf.suite)    # 1对1关系反向查询
```

Json字段操作解析

由于模型中大部分字段都是json类型存储，之前我们未接触过，所以需要对json字段的操作有一定了解

以Request(请求信息)模型为例，操作增删改查

打开django shell

```
python manage.py shell
```

新增

json字段的值直接传字符串，列表，字典即可， (json对应的python数据类型)

```
>>> req1=Request.objects.create(method=1,url='/example/demo',data=
{'name':'xiaoming','age':17,'addr':'nanjing'}
)
>>> req1.data
{'name': 'xiaoming', 'age': 17, 'addr': 'nanjing'}

>>> req2=Request.objects.create(method=1,url='/example/demo2',json='hello')
>>> req2.json
'hello'
>>> req3=Request.objects.create(method=1,url='/example/demo3',json=
['a','b',1,2])
>>> req3.json
['a', 'b', 1, 2]
```

修改

修改整体

```
>>> req2.json={'name': 'mike', 'age': 17, 'addr': 'nanjing'}
>>> req2.save()
>>> req2.json
{'name': 'mike', 'age': 17, 'addr': 'nanjing'}
```

修改局部

```
>>> req1.data['name']
'xiaoming'
>>> req1.data['name']='mike'
>>> req1.save()
>>> req1.data
{'name': 'mike', 'age': 17, 'addr': 'nanjing'}
```

删除

删除整体

```
>>> from django.db.models import  Value
>>> req2.json=Value('null')  # 存储为Json的null
```

删除局部

```
>>> req2.json.pop('name')
'mike'
>>> req2.save()
>>> req2.json
{'age': 17, 'addr': 'nanjing'}
```

查询

```
>>> Request.objects.filter(json__age=17)
<QuerySet [<Request: /example/demo2>]>
```

附录

除了mysqlclient，django操作mysql数据库依赖库还可以用 pymysql

mysql链接时，python版本过高，Django版本低导致连接有问题

1.使用pymysql

```
pip install pymysql
```

2.将Django 安装到最新
3.将pymysql 伪装成MySQLdb。
在主项目的**init**.py中写如下代码

```
import pymysql
pymysql.install_as_MySQLdb()
```

4.将base.py中的报错信息注释掉，如果有此处错误就做这一步，没有就忽略

将Django 安装到最新
将pymysql 伪装成MySQLdb。
在主项目的**init**.py中写如下代码

```
import pymysql
pymysql.install_as_MySQLdb()
```

4.将base.py中的报错信息注释掉，如果有此处错误就做这一步，没有就忽略

```
if version < (1, 3, 13):
    raise ImproperlyConfigured('mysqlclient 1.3.13 or newer is required; you have
%s.' % Database.__version__)
```

5.在数据库中创建对应数据库：

进入MySQL数据库

```
create database db_autotp;
```

6.在setting.py中配置数据库

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'db_autotp',
        'USER': 'root',
        'PASSWORD': 'devops',
        'HOST': '192.168.21.140',
        'PORT': '3306',
        'TEST':{
            'CHARSET': 'utf8',
            'COLLATION': 'utf8_general_ci',
        }
    }
}
```

7.生成迁移文件 :python manage.py makemigrations
再执行迁移文件：python manage.py migrate

8.使用pymysql连接数据库就成功了