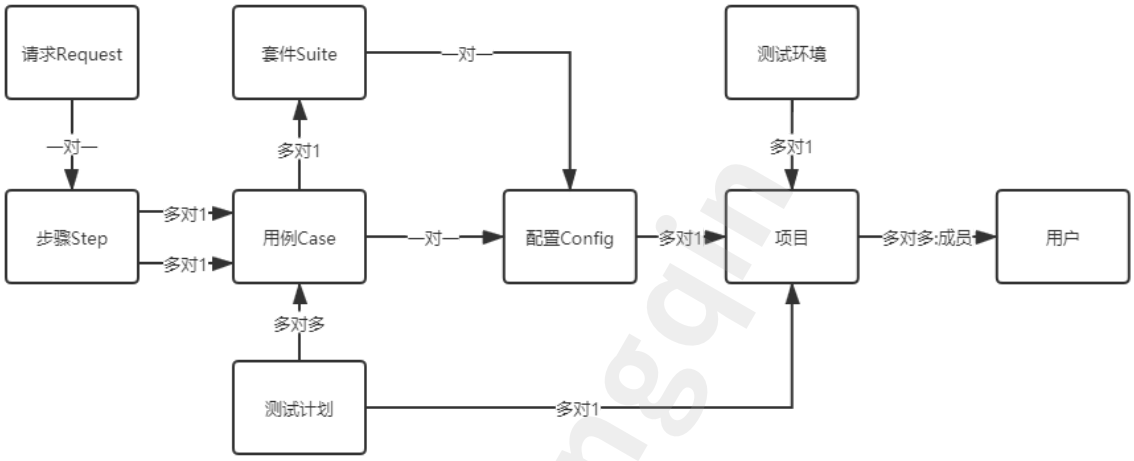
项目管理

到目前为止，我们都是针对核心功能进行的开发工作，要把平台做成一个用户可以使用的程度还需要些额外的功能，比如项目管理，用户管理等。

首先考虑需要增加的数据模型，项目管理需要项目，模块，和测试环境等相关数据，同时原有模型需要增加一些额外的通用字段，如增加时间，更新时间等。

模型关联关系

首先确立模型的关联关系



用例和项目之间的关联可以直接通过config，因为config是套件和用例的扩展，相当于用例和套件本身。

模块的包管理

在我们使用`python manage.py startapp xxx`命令创建新的应用时，Django会自动帮我们建立一个应用的基本文件组织结构，其中就包括一个`models.py`文件。通常，我们把当前应用的模型都编写在这个文件里，但是如果你的模型很多，那么将单独的`models.py`文件分割成一些独立的文件是个更好的做法。

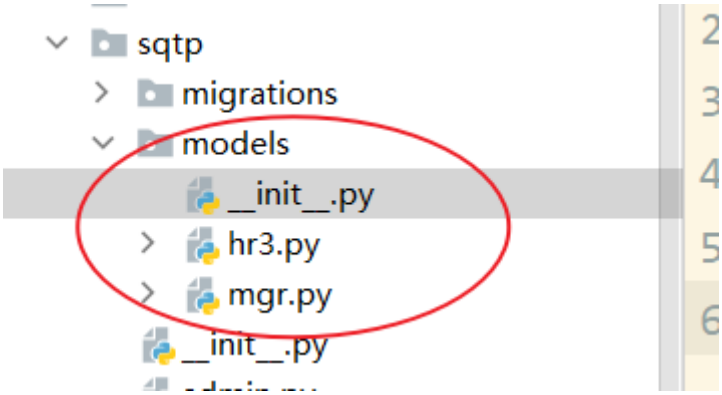
首先，我们需要在应用中新建一个叫做`models`的包，再在包下创建一个`__init__.py`文件，这样才能确立包的身份。然后将`models.py`文件中的模型分割到一些`.py`文件中，比如`plan.py`和`case.py`，然后删除`models.py`文件。最后在`__init__.py`文件中导入所有的模型。如下例所示：

```
#  sqtp/models/__init__.py

from .hr3 import Config,Suite,Case,Step,Request
```

要显式明确地导入每一个模型，而不要使用`from .models import *`的方式，这样不会混淆命名空间，让代码更可读，更容易被分析工具使用。

最终的模型结构



其中hr3.py存放所有核心数据模型

mgr.py存放所有项目管理相关模型

mgr.py中我们暂时定义项目和环境数据模型

```
"""
@author: haiwen
@date: 2021/6/16
@file: mgr.py
"""

from django.db import models


class Project(models.Model):
    proj_status = (
        ('developing', '开发中'),
        ('operating', '维护中'),
        ('stable', '稳定运行')
    )

    # 名称
    name = models.CharField(max_length=32, unique=True, verbose_name='项目名称')
    # 状态
    status = models.CharField(choices=proj_status, max_length=32,
default='stable', verbose_name='项目状态')
    # 版本
    version = models.CharField(max_length=32, default='v1.0', verbose_name='版
本')

    class Meta:
        verbose_name = '项目表'


class Environment(models.Model):
    # 服务器类型选项
    service_type = (
        (0, 'web服务器'),
        (1, '数据库服务器'),
    )
    # 服务器操作系统选项
    service_os = (
        (0, 'window'),
        (1, 'linux'),
    )
```

```
    # 服务器状态选项
    service_status = (
        (0, 'active'),
        (1, 'disable'),
    )

    project = models.ForeignKey(Project, on_delete=models.CASCADE,
verbose_name='所属项目')
    # ip--使用djangoORM提供的一个叫GenericIPAddressField专门储存IP类型的字段
    ip = models.GenericIPAddressField(default='127.0.0.1', verbose_name='ip地址')
    port = models.SmallIntegerField(default=80, verbose_name='端口号')
    # 服务器类型
    category = models.SmallIntegerField(default=0, choices=service_type,
verbose_name='服务器类型')
    # 操作系统
    os = models.SmallIntegerField(default=0, choices=service_os, verbose_name='服
务器操作系统')
    # 状态
    status = models.SmallIntegerField(default=0, choices=service_status,
verbose_name='服务器状态')

    class Meta:
        verbose_name = '测试环境表'
```

模型的继承

一般在实际项目中，数据模型除了业务字段以外，还需要有一些通用字段，如创建时间，更新时间，创建者，更新者等，这些字段如果在每个模型都定义的话，冗余度很高，而且维护起来不方便。

此时，我们可以用一个抽象模型类来存放这些字段，然后其他模型继承该抽象模型类即可。

新建 models/base.py，定义一个抽象模型CommonInfo

```
from django.db import models


# 公共抽象模型
class CommonInfo(models.Model):
    # 创建
    create_time = models.DateTimeField(auto_now_add=True, verbose_name='创建时间')
 # auto_now_add 第1次创建数据时自动添加当前时间
    # 更新
    update_time = models.DateTimeField(auto_now=True, verbose_name='创建时间',
null=True)  # auto_now 每次更新数据时自动添加当前时间
    # 描述--文本
    desc = models.TextField(null=True, blank=True, verbose_name='描述')

    def __str__(self):
        # 检查当前对象有没有name属性
        # 有name就返回name,没有就返回desc
        if hasattr(self, 'name'):
            return self.name
        return self.desc

    class Meta:
        abstract = True  # 定义为抽象表，不会创建数据库表
```

```
        # 默认使用id排序
        ordering = ['id']
```

由于在元类中定义了abstract = True ， 抽象模型在同步数据库的时候并不会创建表，子类只会继承其字段和方法

另外元类除了abstract = True不会继承，其他都会继承，若想把子类也设置为抽象模型，必须显示在元类中设置abstract = True。

子类继承抽象父类：

```
# mgr.py
class Project(CommonInfo):
    ...

class Environment(CommonInfo):
    ...
```

```
# hr3.py
from django.db import models


# Create your models here.
from .mgr import Project
from .base import CommonInfo

class Config(CommonInfo):
    ...

class Step(CommonInfo):
    ...


# 请求
class Request(CommonInfo):
    ...

class Case(CommonInfo):
    ...


class Suite(CommonInfo):
    ...
```

包文件增加导入

```
#  sqtp/models/__init__.py

from .hr3 import Config,Suite,Case,Step,Request
from .mgr import Project,Environment
```

此时同步数据库

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

抽象基类的Meta数据：

如果子类没有声明自己的Meta类，那么它将自动继承抽象基类的Meta类。

如果子类要设置自己的Meta属性，则需要扩展基类的Meta：

```
from django.db import models

class CommonInfo(models.Model):
    # ...
    class Meta:
        abstract = True
        ordering = ['name']

class Student(CommonInfo):
    # ...
    class Meta(CommonInfo.Meta):   # 注意这里有个继承关系
        db_table = 'student_info'
```

这里有几点要特别说明：

- 抽象基类中有的元数据，子模型没有的话，直接继承；
- 抽象基类中有的元数据，子模型也有的话，直接覆盖；
- 子模型可以额外添加元数据；
- 抽象基类中的`abstract=True`这个元数据不会被继承。也就是说如果想让一个抽象基类的子模型，同样成为一个抽象基类，那你必须显式的在该子模型的Meta中同样声明一个`abstract = True`；
- 有一些元数据对抽象基类无效，比如`db_table`，首先是抽象基类本身不会创建数据表，其次它的所有子类也不会按照这个元数据来设置表名。
- 由于Python继承的工作机制，如果子类继承了多个抽象基类，则默认情况下仅继承第一个列出的基类的 Meta 选项。如果要从多个抽象基类中继承 Meta 选项，必须显式地声明 Meta 继承。例如：

```
from django.db import models

class CommonInfo(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:
        abstract = True
        ordering = ['name']

class Unmanaged(models.Model):
    class Meta:
        abstract = True
        managed = False

class Student(CommonInfo, Unmanaged):
    home_group = models.CharField(max_length=5)

    class Meta(CommonInfo.Meta, Unmanaged.Meta):
        pass
```

警惕related_name和related_query_name参数

如果在你的抽象基类中存在ForeignKey或者ManyToManyField字段，并且使用了`related_name`或者`related_query_name`参数，那么一定要小心了。因为按照默认规则，每一个子类都将拥有同样的字段，这显然会导致错误。为了解决这个问题，当你在抽象基类中使用`related_name`或者`related_query_name`参数时，它们两者的值中应该包含`%(app_label)s`和`%(class)s`部分：

- `%(class)s`用字段所属子类的小写名替换
- `%(app_label)s`用子类所属app的小写名替换

例如，对于`common/models.py`模块：

```
from django.db import models

class Base(models.Model):
    m2m = models.ManyToManyField(
    OtherModel,
    related_name="%(app_label)s_%(class)s_related",
    related_query_name="%(app_label)s_%(class)ss",
    )

    class Meta:
        abstract = True

class ChildA(Base):
    pass

class ChildB(Base):
    pass
```

对于另外一个应用中的`rare/models.py`：

```
from common.models import Base

class ChildB(Base):
    pass
```

对于上面的继承关系：

- `common.ChildA.m2m`字段的`reverse name`（反向关系名）应该是`common_childa_related`；`reverse query name`(反向查询名)应该是`common_childas`。
- `common.ChildB.m2m`字段的反向关系名应该是`common_childb_related`；反向查询名应该是`common_childbs`。
- `rare.ChildB.m2m`字段的反向关系名应该是`rare_childb_related`；反向查询名应该是`rare_childbs`。

当然，如果你不设置`related_name`或者`related_query_name`参数，这些问题就不存在了。

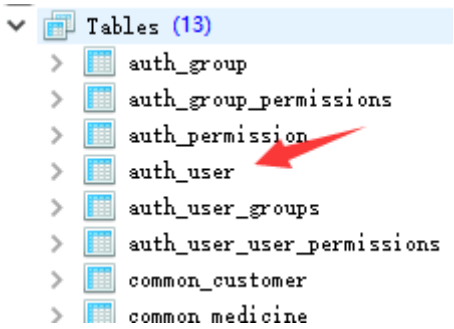
用户管理

后面我们在开发登录功能的时候需要数据库来保存用户登录信息和状态，

Django中有个内置app 名为`django.contrib.auth`，缺省包含在项目Installed App设置中。

这个app 的 models 定义中包含了一张 用户表，名为 `auth_user` 。

当我们执行 migrate 创建数据库表时，根据，就会为我们创建 用户表 `auth_user`，如下所示



`django.contrib.auth` 这个app 已经 **为我们做好了登录验证功能。**

自定义用户模型

Django 内置模块 contrib.auth 帮我们预置了一张用户表 `user` ，包含的字段有：

id	INT	11
password	VARCHAR	128
last_login	DATETIME	6
is_superuser	TINYINT	1
username	VARCHAR	150
first_name	VARCHAR	150
last_name	VARCHAR	150
email	VARCHAR	254
is_staff	TINYINT	1
is_active	TINYINT	1
date_joined	DATETIME	6

如果以上字段足够我们在项目中使用，那么我们直接使用默认User模型即可

但是如果他不符合我们的需求，我们要重新定义，怎么办?

比如，这个项目的用户表需要新增一些字段，像 真实姓名、手机号、用户类型等。

千万不要去改 Django 内置模块 contrib.auth.models 里面的类定义。（想一想，为什么？）

（如果改动源码，那么你的项目就依赖当前库，如果更新了库或者环境移植重写装库，那你原来的代码就失效了）

一种推荐的方式是：通过继承 `django.contrib.auth.models` 里面的 `AbstractUser` 类的方式

在你项目文件的 models 里面进行如下定义：

```
from django.contrib.auth.models import AbstractUser
from django.db import models

class User(AbstractUser):
    USER_TYPE=(
        (0,'开发'),
        (1,'测试'),
        (2,'运维'),
        (3,'项目经理'),
    )
    realname = models.CharField('真实姓名',max_length=30)
    phone = models.CharField('手机
号',max_length=11,unique=True,null=True,blank=True)
    user_type = models.SmallIntegerField('用户类型 ',choices=USER_TYPE,default=1)
```

这样就在原来的 contrib.auth 里面的 user表的基础上

- 新增了 usertype、realname、phone、 这些字段

Django自定义验证

然后，你需要告诉Django，使用这个表作为 系统的 user表。

就是 在 settings.py 中，添加如下设置

```
AUTH_USER_MODEL = 'sqtp.User'
```

其中 `sqtp` 为你的 User 定义 所在的 django app 名称

一定要确保 这个 sqtp 在你的 INSTALLED_APPS 里面设置了。

接下来同步数据库

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

可能会遇到

```
The field admin.LogEntry.user was declared with a lazy reference to 'sqtp.user',
but app 'sqtp' doesn't provide model 'use r'.
```

这个错误是由于原有的用户模型被依赖，使用自定义会改变依赖关系，这个改动并不能自动完成，需要手动修复你的架构，将数据从旧的用户表移出，并有可能需要手动执行一些迁移操作。

由于 Django 对可交换模型的动态依赖特性的限制，`AUTH_USER_MODEL` 所引用的模型必须在其应用的第一次迁移中创建（通常称为 `0001_initial`）；否则，你会出现依赖问题。

通俗点的就是，**1.删除migrations目录所有文件，2.删除数据库并重写创建。**然后再执行迁移动作就可以了。

引用User模型

用户模型创建好之后，其他模型需要关联的部分就可以加上了。

首先是Project，增加项目管理员和成员，由于关联了同一个模型，所以不要忘了设置反向查询名related_name

如果我们直接引用User模型,那么以后我们的项目改成不同的用户模型

时就无法自动切到`AUTH_USER_MODEL`配置的用户模型，代码的可复用性就大打折扣了。因此应该直接引用`AUTH_USER_MODEL`配置的用户模型，方法如下

```
from django.conf import settings
from .base import CommonInfo


class Project(CommonInfo):
    proj_status = (
        (0, 'developing',),
        (1, 'operating',),
        (2, 'stable',)
    )
    # 管理员
    admin = models.ForeignKey(settings.AUTH_USER_MODEL,
on_delete=models.DO_NOTHING, null=True,related_name='project')
    # 成员
    members = models.ManyToManyField(settings.AUTH_USER_MODEL,
related_name='projects')
    ...
```

接下来是通用表中的创建者和更新者字段

```
from django.db import models

from django.conf import settings

# 公共抽象模型
class CommonInfo(models.Model):
    # 创建者
    create_by =
models.ForeignKey(settings.AUTH_USER_MODEL,on_delete=models.SET_NULL,null=True,v
erbose_name='创建者', related_name='create_by')
    # 更新者
    update_by =
models.ForeignKey(settings.AUTH_USER_MODEL,on_delete=models.SET_NULL,null=True,v
erbose_name='更新者', related_name='update_by')
    ...
```

同步数据库时，出现错误

```
HINT: Add or change a related_name argument to the definition for
'sqtp.Project.update_by' or 'sqtp.Environment.update_by'.
sqtp.Project.update_by: (fields.E305) Reverse query name for
'sqtp.Project.update_by' clashes with reverse query name for 'sqtp.Enviro
nment.update_by'.
```

这时由于父类的字段被子类继承，都使用了相同的反向查询名，者显然是不行的，所以利用`%(app_label)s`和`%(class)s`让子类继承的字段可以自由替换反向查询名。

```
class CommonInfo(models.Model):
    # 创建者
    create_by = models.ForeignKey(settings.AUTH_USER_MODEL,
on_delete=models.SET_NULL, null=True, verbose_name='创建者',
                                  related_name='%(app_label)s_%(class)s_create')
    # 更新者
    update_by = models.ForeignKey(settings.AUTH_USER_MODEL,
on_delete=models.SET_NULL, null=True, verbose_name='更新者',
                                  related_name='%(app_label)s_%(class)s_update')
```

同步数据库

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

视图开发

采用REST框架开发项目管理和用户管理

创建序列器

serializers.py

```
# 项目
class ProjectSerializer(serializers.ModelSerializer):
    class Meta:
        model = Project
        fields = '__all__'

# 环境
class EnvironmentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Environment
        fields = '__all__'

# 用户
class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = '__all__'
```

创建视图

```
# 项目
class ProjectViewSet(viewsets.ModelViewSet):
    queryset = Project.objects.all()
    serializer_class = ProjectSerializer

# 环境
class EnvironmentViewSet(viewsets.ModelViewSet):
    queryset = Environment.objects.all()
    serializer_class = EnvironmentSerializer
```

```python
# 用户
@api_view(['GET'])
def user_list(request):
    queryset = User.objects.all()
    serializer = UserSerializer(queryset, many=True)
    return Response(serializer.data)


@api_view(['GET'])
def user_detail(request, _id):
    try:
        req_obj = User.objects.get(pk=_id)
    except User.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)  # status提供常用http状态码

    serializer = UserSerializer(req_obj)
    return Response(serializer.data)
```

创建路由

```python
# 用户
urlpatterns = [
    path('',include(router.urls)),  # 引用相关路由
    path('custom/',views.customer_api),
    path('users/',views.user_list),
    path('users/<int:_id>',views.user_detail),
]
```

由于前端页面目前还在开发中，大家可以用在线接口文档（http://127.0.0.1:8081/swagger/ ）测试接口的增删改查