一、序列化关系字段

关系字段用于表示模型之间的关联。在Django中存在 `ForeignKey` 、 `ManyToManyField` 和 `OneToOneField` 三种正向关系，以及反向关联和自定义关联。

注意: 在DRF框架，关系型字段的代码虽然定义在 `relations.py` 模块中，但我们通常从 `serializers` 模块中导入关系型字段，也就是 `from rest_framework import serializers` ，然后使用 `serializers.<FieldName>` 的方式引用。

当继承 `ModelSerializer` 类的时候，包括关系型字段在内的所有字段会自动生成。我们可以查看这些字段。使用 `python manage.py shell` 命令，进入Django的shell环境，然后按下面的操作查看字段信息：

```
>>> from myapp.serializers import AccountSerializer
>>> serializer = AccountSerializer()
>>> print(repr(serializer))
AccountSerializer():
    id = IntegerField(label='ID', read_only=True)
    name = CharField(allow_blank=True, max_length=100, required=False)
    owner = PrimaryKeyRelatedField(queryset=User.objects.all())
```

二、API参考

为了解释不同类型的关系字段，我们使用下面的一组模型作为例子。Album是唱片，Track是每张唱片里的某首歌曲。

```
class Album(models.Model):  # 唱片
    album_name = models.CharField(max_length=100)   # 唱片的名字
    artist = models.CharField(max_length=100)       # 艺术家

class Track(models.Model):  # 歌曲
    # 通过外键关联某张唱片
    album = models.ForeignKey(Album, related_name='tracks', on_delete=models.CASCADE)
    order = models.IntegerField()  # 顺序
    title = models.CharField(max_length=100) # 歌曲名
    duration = models.IntegerField()  # 时间长度

    class Meta:
        unique_together = ('album', 'order')  # 每首歌在唱片中的顺序必须唯一
```

```
        ordering = ['order']  # 按序号排序

    def __str__(self):
        return '%d: %s' % (self.order, self.title)
```

StringRelatedField

`StringRelatedField` 使用对象的 `__str__` 方法来表示关联的对象。这个字段其实也就是将关联对象的字符串表示形式的信息拿来，放到自己的序列化类中，供API视图使用并渲染，然后传递给前端。

例如下面的序列化类：

```
class AlbumSerializer(serializers.ModelSerializer):
    tracks = serializers.StringRelatedField(many=True)  # 额外增加了一个字段，注意many参数

    class Meta:
        model = Album
        fields = ('album_name', 'artist', 'tracks')
```

上面的操作，会获得下面的结果，注意'tracks'键：

```
{
    'album_name': 'Things We Lost In The Fire',
    'artist': 'Low',
    'tracks': [
        '1: Sunflower',
        '2: Whitetail',
        '3: Dinosaur Act',
        ...
    ]
}
```

注意：该字段是只读的！

StringRelatedField的参数：

- `many` - 如果关联的是一个复数数量的对象，必须将此参数设置为 `True` 。

PrimaryKeyRelatedField

`PrimaryKeyRelatedField` 使用关联对象的主键id值来表示对象。

例如:

```
class AlbumSerializer(serializers.ModelSerializer):
    tracks = serializers.PrimaryKeyRelatedField(many=True, read_only=True) # 一定要注意参数

    class Meta:
        model = Album
        fields = ('album_name', 'artist', 'tracks')
```

序列化的结果如下，注意那些数字:

```
{
    'album_name': 'Undun',
    'artist': 'The Roots',
    'tracks': [
        89,
        90,
        91,
        ...
    ]
}
```

默认情况下，这种字段关联方式是可读可写的。可以通过添加 `read_only=True` 标识，变成只读!

PrimaryKeyRelatedField参数:

- `queryset` - 当对字段的输入数据进行验证的时候，用于查询模型实例的查询集。必须显式的提供这个参数，或者设置 `read_only=True` 。
- `many` - 如果关联的是一个复数数量的对象，必须将此参数设置为 `True` 。
- `allow_null` - 默认为False。如果设置为 `True` ，在可空的关联上，该字段将可以接收 `None` 或空字符串。
- `pk_field` - 指定序列化/反序列化过程中，使用的主键字段类型。例如，`pk_field=UUIDField(format='hex')` 将序列化一个UUID主键值。

HyperlinkedRelatedField

`HyperlinkedRelatedField` 使用关联对象的超链接形式来标识关联对象。也就是说，不使用字符串，也不用主键id数字，而是用一个可以点击跳转的url地址来表示关联的对象。

例如：

```
class AlbumSerializer(serializers.ModelSerializer):
    tracks = serializers.HyperlinkedRelatedField(
        many=True,
        read_only=True,
        view_name='track-detail'
    )  # 注意参数

    class Meta:
        model = Album
        fields = ('album_name', 'artist', 'tracks')
```

序列化的结果;

```
{
    'album_name': 'Graceland',
    'artist': 'Paul Simon',
    'tracks': [
        'http://www.example.com/api/tracks/45/',
        'http://www.example.com/api/tracks/46/',
        'http://www.example.com/api/tracks/47/',
        ...
    ]
}
```

默认情况下，这种字段关联方式是可读可写的。可以通过添加 `read_only=True` 标识，变成只读!

注意: 这种字段是为那些将自己映射到某个URL的对象设计的。对应的URL接收 `lookup_field` 和 `lookup_url_kwarg` 参数。

这种关系字段适合包含单独主键或者slug参数的URLs。

如果你需要更复杂的超链接表示形式，参考下面的自定义关系字段章节。

参数:

- `view_name` - 处理关联对象URL的视图。如果你使用的是标准的路由类，它必须是一个 `<modelname>-detail` 格式的字符串。此参数必填!

- `queryset` - 当对字段的输入数据进行验证的时候，用于查询模型实例的查询集。必须显式的提供这个参数，或者设置 `read_only=True` 。
- `many` - 如果关联的是一个复数数量的对象，必须将此参数设置为 `True` 。
- `allow_null` -默认为False。如果设置为 `True` ，在可空的关联上，该字段将可以接收 `None` 或空字符串。
- `lookup_field` - 该参数的默认值为 `'pk'` ，表示用主键id在视图种查找关联的对象，一般我们不需要修改这个参数。它必须和对应视图的URL关键字参数一致，你这里如果改了，在视图中也必须跟着改。
- `lookup_url_kwarg` - 指定上面参数值的名字，一般使用 `lookup_field` 。大多数情况下，我们不用修改这个参数，默认就好，除非你需要自定义一大堆，不要给自己挖坑。
- `format` - 如果URL种使用了格式后缀，超链接字段将使用同样的格式后缀，除非使用这个 `format` 参数另外指定后缀形式。

SlugRelatedField

`SlugRelatedField` 使用某个指定的字段的值作为关联对象的表示形式。比如拿对象的名字、或者邮箱、或者昵称、或者地址等等。

例如:

```
class AlbumSerializer(serializers.ModelSerializer):
    tracks = serializers.SlugRelatedField(
        many=True,
        read_only=True,
        slug_field='title'
     )

    class Meta:
        model = Album
        fields = ('album_name', 'artist', 'tracks')
```

序列化的结果

```
{
    'album_name': 'Dear John',
    'artist': 'Loney Dear',
    'tracks': [
        'Airport Surroundings',
        'Everything Turns to You',
        'I Was Only Going Out',
        ...
    ]
}
```

默认情况下，这种字段关联方式是可读可写的。可以通过添加 `read_only=True` 标识，变成只读！

如果 `SlugRelatedField` 作为一个可读写的字段，那么你必须确保对应的模型种的字段必须是 `unique=True` ，否则你用什么来区分是哪个对象？

参数:

- `slug_field` - 指定用来表示关联对象的字段，该参数必填。
- `queryset` - 同上
- `many` - 同上
- `allow_null` - 同上

HyperlinkedIdentityField

这种字段可以用作身份关联，使用较少。

```
class AlbumSerializer(serializers.HyperlinkedModelSerializer): # 注意继承的类变了
    track_listing = serializers.HyperlinkedIdentityField(view_name='track-list')

    class Meta:
        model = Album
        fields = ('album_name', 'artist', 'track_listing')
```

序列化结果：

```
{
    'album_name': 'The Eraser',
    'artist': 'Thom Yorke',
    'track_listing': 'http://www.example.com/api/track_list/12/',
}
```

这种字段只读!

参数:

- `view_name` - 同前。必填。
- `lookup_field` - 同前
- `lookup_url_kwarg` - 同前
- `format` - 同前

三、嵌套关联

可以将序列化类作为字段，来表示嵌套关联。

如果关联的是一个复数数量的对象，必须将 `many` 参数设置为 `True` 。

例如:

```
class TrackSerializer(serializers.ModelSerializer):
    class Meta:
        model = Track
        fields = ('order', 'title', 'duration')

class AlbumSerializer(serializers.ModelSerializer):
    tracks = TrackSerializer(many=True, read_only=True)

    class Meta:
        model = Album
        fields = ('album_name', 'artist', 'tracks')
```

序列化的过程例子如下:

```
>>> album = Album.objects.create(album_name="The Grey Album",
artist='Danger Mouse')
>>> Track.objects.create(album=album, order=1, title='Public Service
Announcement', duration=245)
<Track: Track object>
>>> Track.objects.create(album=album, order=2, title='What More Can I Say',
duration=264)
```

```
<Track: Track object>
>>> Track.objects.create(album=album, order=3, title='Encore', duration=159)
<Track: Track object>
>>> serializer = AlbumSerializer(instance=album)
>>> serializer.data
{
    'album_name': 'The Grey Album',
    'artist': 'Danger Mouse',
    'tracks': [
        {'order': 1, 'title': 'Public Service Announcement', 'duration': 245},
        {'order': 2, 'title': 'What More Can I Say', 'duration': 264},
        {'order': 3, 'title': 'Encore', 'duration': 159},
        ...
    ],
}
```

默认情况下，嵌套关联是只读的！如果你想设置可读写的嵌套关联，你必须自己实现 `create()` 与/或 `update()` 方法，显式地指定如何保存子关系。例如：

```
class TrackSerializer(serializers.ModelSerializer):
    class Meta:
        model = Track
        fields = ('order', 'title', 'duration')

class AlbumSerializer(serializers.ModelSerializer):
    tracks = TrackSerializer(many=True)

    class Meta:
        model = Album
        fields = ('album_name', 'artist', 'tracks')

    def create(self, validated_data):
        tracks_data = validated_data.pop('tracks')
        album = Album.objects.create(**validated_data)
        for track_data in tracks_data:
            Track.objects.create(album=album, **track_data)
        return album

>>> data = {
    'album_name': 'The Grey Album',
    'artist': 'Danger Mouse',
    'tracks': [
```

```
        {'order': 1, 'title': 'Public Service Announcement', 'duration': 245},
        {'order': 2, 'title': 'What More Can I Say', 'duration': 264},
        {'order': 3, 'title': 'Encore', 'duration': 159},
    ],
}
>>> serializer = AlbumSerializer(data=data)
>>> serializer.is_valid()
True
>>> serializer.save()
<Album: Album object>
```

四、自定义关系类型字段

在很少的情况下，如果现有的关系样式都不适合所需要的表示，那么你可以实现一个完全自定义的关系字段，该字段准确描述了应该如何从模型实例生成输出表示。

要自定义关系类型字段，你必须先继承 `RelatedField` 类，然后实现 `.to_representation(self, value)` 方法。此方法将字段的目标作为“value”参数，并应返回对象序列化后的表示形式。“value”参数通常是模型实例。

如果你想实现可读写的关系类型字段，你还需要实现 `.to_internal_value(self, data)` 方法.

如果想提供一个基于 `context` 的动态查询集，你需要覆盖 `.get_queryset(self)` 方法。

例如:

```
import time

class TrackListingField(serializers.RelatedField):
    def to_representation(self, value):  # 看这里
        duration = time.strftime('%M:%S', time.gmtime(value.duration))
        return 'Track %d: %s (%s)' % (value.order, value.name, duration)

class AlbumSerializer(serializers.ModelSerializer):
    tracks = TrackListingField(many=True)

    class Meta:
        model = Album
        fields = ('album_name', 'artist', 'tracks')
```

自定义字段的序列化结果：

```
{
    'album_name': 'Sometimes I Wish We Were an Eagle',
    'artist': 'Bill Callahan',
    'tracks': [
        'Track 1: Jim Cain (04:39)',
        'Track 2: Eid Ma Clack Shaw (04:19)',
        'Track 3: The Wind and the Dove (04:34)',
        ...
    ]
}
```

五、自定义超链接字段

首先需要继承HyperlinkedRelatedField`类，然后根据需要，选择性地覆写下面两个方法：

- **get_url(self, obj, view_name, request, format)**

此方法用于对象实例和它的URL表示之间的映射。

- **get_object(self, view_name, view_args, view_kwargs)**

这个方法的返回值必须和匹配的URL conf参数一致。

下面假设我们有一个顾客对象的URL，并接受两个参数：

```
/api/<organization_slug>/customers/<customer_pk>/
```

对于上面这种url，普通的超链接字段无法实现，因为超链接字段只能处理url中只有一个参数的情况，而上面有两个参数。只能自定义超链接字段，然后自己写代码处理了：

```
from rest_framework import serializers
from rest_framework.reverse import reverse

class CustomerHyperlink(serializers.HyperlinkedRelatedField):
    # 这里把下面两个变量作为类属性进行处理，这样就不需要传递参数了
    view_name = 'customer-detail'
    queryset = Customer.objects.all()

    def get_url(self, obj, view_name, request, format):
        url_kwargs = {
            'organization_slug': obj.organization.slug,
```

```
            'customer_pk': obj.pk
        }
        return reverse(view_name, kwargs=url_kwargs, request=request,
    format=format)

    def get_object(self, view_name, view_args, view_kwargs):
        lookup_kwargs = {
           'organization__slug': view_kwargs['organization_slug'],
           'pk': view_kwargs['customer_pk']
        }
        return self.get_queryset().get(**lookup_kwargs)
```

如果你想将上面的自定义超链接字段和通用视图一起使用，那么你还需要在视图中重写 `.get_object` 方法，保持两者的一致。

六、注意事项

- `queryset` 参数

“queryset”参数仅对**可写**关系字段是必需的，在这种情况下，它用于执行从基本用户输入映射到模型实例的查找。

- 自定义HTML 显示

当在HTML的可浏览API中，如果要显示包含 `choices` 属性的字段时，默认使用对象的 `__str__` 方法，也就是字符串表示形式。

如果要自定义这个形式，重写 `RelatedField` 的 `display_value()` 方法。该方法接收一个实例作为参数，并应当返回一个合适的表示形式，例如：

```
class TrackPrimaryKeyRelatedField(serializers.PrimaryKeyRelatedField):
    def display_value(self, instance):
        return 'Track: %s' % (instance.title)
```

- 反向关联

请注意，在 继承 `ModelSerializer` 和 `HyperlinkedModelSerializer` 类时，不会自动生成反向关联，你必须显式地添加反向关联字段。例如：

```
class AlbumSerializer(serializers.ModelSerializer):
    class Meta:
        fields = ('tracks', ...)
```

一般情况下，我们会为反向关联设置一个 `related_name` 参数，作为反向关联时的字段名，例如：

```
class Track(models.Model):
    album = models.ForeignKey(Album, related_name='tracks', on_delete=models.CASCADE)
    ...
```

如果没有指定，那么你只能使用Django原生给我们提供的关联名，也就是 `modelname_set`：

```
class AlbumSerializer(serializers.ModelSerializer):
    class Meta:
        fields = ('track_set', ...)
```

For more information see **the Django documentation on generic relations**.

- 使用中间模型的ManyToManyFields

默认情况下，关联到一个使用了中间模型的ManyToManyFields的字段，需要设为只读，因此请确保为字段设置了 `read_only=True` 属性。