Forms

> `django`提供了一整套健全的机制来帮助我们自动创建对应`HTML`中的表单
>
> 开发者可以方便的使用已经设定好的一系列字段进行表单的设计
>
> 可以在某个`app`下面新建一个`forms.py`文件，在这个文件编写`django`自带表单类的编写
>
> 比如像下面这样

```
from django import forms
class TestForm(forms.Form):
    name = forms.CharField(label='名字:',max_length=100)
```

> 在这个表单类中，设置了一个`CharField`字段，并且具有`label`标签值为`name`
>
> 此外在`<input>`标签处还会设置`maxlength=100`的属性
>
> `django`在接收到这样表单内的数据时，还将验证数据的长度
>
> 实例化该类，然后打印出来查看效果

```
<tr><th><label for="id_name">名字:</label></th><td><input type="text" name="name"
maxlength="100" required id="id_name" /></td></tr>
```

> 在渲染后的结果中不包含提交的按钮，以及外层的form标签，还需要我们自己手动在模板页面中进行添加

> `form`表单实例的使用也非常简单，直接在模板页面处将表单实例以模板变量形式传递赋值即可

```
# views.py
def index(request):
    form = forms.TestForm()
    return render(request,'index.html',locals())
```

```
<!-- index.html -->
<form action="/" method="POST">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="提交">
</form>
```

is_valid

> 每一个`form`类的实例都具有一个`is_valid()`方法，验证表单内的字段是否合法，并将表单中合法的的数据将放到表单中的`cleaned_data`属性中
>
> 如果全部数据都没有问题，那么该函数将会返回`True`，返回的合法数据。结果是一个字典的数据类型

```
form = TestFrom()
if form.is_valid():
    data = form.cleaned_data
```

```
def post_test(request):
    if request.method == "POST":
        form = TestForm(request.POST)
        if form.is_valid():
            name = form.cleaned_data.get('name')
            return HttpResponse('OK')
    else:
        form = TestForm()
    return render(request, "xxx.html",{"form":form})
```

> 在视图函数中，当用户以post形式提交数据，此时将post数据与表单类进行关联
>
> 使用`post`数据做为类实例化的参数，这种操作也叫作**绑定数据到表单**
>
> 如果用户在表单中填写张三并提交，那么绑定数据之后的表单实例像是这样

```
<label for="id_name">名字:</label>
<input id="id_name" maxlength="100" name="name" type="text" value="张三" />
```

> `input`标签中的`value`值为用户`post`所提交的数据
>
> 如果绑定数据的表单实例经过`is_valid`函数校验并通过，那么正确的数据将存储在`cleaned_data`中，`cleaned_data`中的数据同时也是处理好的`Python`数据类型，比如这里为一个字典数据类型
>
> 接下来在视图函数中可以直接通过字典的操作方式来获取到用户在对应表单标签中所填写的数据

表单字段类型

> 所有表单字段`Field`的子类均带有默认参数`require`

BooleanField

> 控件：`CheckboxInput`
>
> 复选框：`<input type='checkbox' ...>`

> 空值：`False`
>
> `Python`：`True\False`
>
> **错误键**：`required`

CharField

> 控件：`TextInput`
>
> 文本输入：`<input type="text" ...>`
>
> 空值：空字符串

> `Python`：`str`
>
> **错误键**：`max_length`、`min_length`、`required`

ChoiceField

> 控件：`Select`
>
> 选择框：`<select><option ...>...</select>`
>
> 空值：空字符串
>
> Python：`Unicode str`
>
> 必选参数：`choices`，该参数为一个二元组组成的可迭代对象，二元组中的第一个值为获取到的数据，第二个值为表单中展示的内容。
>
> **错误键**：`required`、`invalid_choice`

```
class TestForm(forms.Form):
    choices = (
        ('0','男'),
        ('1','女'),
    )
    gender = forms.ChoiceField(choices=choices)
```

DateField

> 控件：`DateInput`
>
> 日期以普通的文本框输入：`<input type='text' ...>`
>
> 空值：`None`
>
> Python：`datetime.date`
>
> 验证是否为一个指定日期格式的字符串
>
> **错误键**：`required`、`invalid`
>
> 可选参数：`input_formats`，一个时间格式化字符串，用来将表单中的数据转换为`datetime.date`对象
>
> 可选参数格式参考如下：

```
'%Y-%m-%d', # '2006-10-25'
'%m/%d/%Y', # '10/25/2006'
'%m/%d/%y'  # '10/25/06'
```

DateTimeField

> 控件：`DateTimeInput`
>
> 日期/时间以普通的文本框输入：<input type='text' ...>
>
> 空值：None

> Python：`datetime.datetime`
>
> 验证是否为一个指定日期格式的字符串
>
> 可选参数：`input_formats`，一个时间格式化字符串，用来将表单中的数据转换为`datetime.datetime`对象
>
> **错误键**：`required`、`invalid`

DecimalField

> 控件：当`Field.localize`是`False`时为`NumberInput`，否则为`TextInput`
>
> `NumberInput`文本输入：`<input type="number" ...>`
>
> `TextInput`文本输入：`<input type="text" ...>`
>
> 空值：`None`
>
> Python：`decimal`
>
> 验证给定值是否为一个十进制数字
>
> 可选参数：`max_value`、`min_value`控制大小值范围
>
> `max_digits`：值允许的最大位数（小数点之前和之后的数字总共的位数，前导的零将被删除）
>
> `decimal_places`：允许的最大小数位
>
> **错误键**：`required`，`invalid`，`max_value`，`min_value`，`max_digits`，`max_decimal_places` max_whole_digits

EmailField

> 控件：文本输入：`<input type="email" ...>`
>
> 空值：空字符串
>
> Python：`Unicode str`
>
> 使用正则验证给定的值是否为一个合法的邮件地址
>
> 可选参数：`max_length`与`min_length`，限定邮件地址字符串大小长度。
>
> **错误键**：`required`、`invalid`

FileField

> 控件：`ClearableFileInput`
>
> 文件上传输入：`<input type='file' ...>`
>
> 空值：`None`
>
> Python：`UploadedFile`
>
> 验证非空的文件数据绑定到表单
>
> 使用该字段时，在使用表单实例获取上传文件数据时，表单标签中需要具备`enctype="multipart/form-data"`属性，此外还需要绑定文件数据在表单上

```
form = TestForm(request.POST,request.FILES)
```

FloatField

> 控件：当`Field.localize`是`False`时为`NumberInput`，否则为`TextInput`
>
> `NumberInput`文本输入：`<input type="number" ...>`
>
> `TextInput`文本输入：`<input type="text" ...>`
>
> 空值：`None`
>
> Python：`Float`
>
> 验证给出的值是一个浮点数，对比`float`函数
>
> 可选参数：`max_value`、`min_value`限定大小值范围
>
> **错误键**：`required`， `invalid`， `max_value`， `min_value`

ImageField

> 控件：`ClearableFileInput`
>
> 文件上传输入：`<input type='file' ...>`
>
> 空值：`None`
>
> Python：`UploadedFile`
>
> 验证文件数据并且检验是否是一个可以被pillow所解释的图像
>
> 使用该字段，需要安装`pillow`模块。
>
> **错误键**：`required`，`invalid`，`missing`，`empty`，`invalid_image`

IntergerField

> 控件：当`Field.localize`是`False`时为`NumberInput`，否则为`TextInput`
>
> `NumberInput`文本输入：`<input type="number" ...>`
>
> `TextInput`文本输入：`<input type="text" ...>`
>
> 空值：`None`
>
> Python：`int`
>
> 验证给定的值是否是一个整数
>
> 可选参数：`max_value`、`min_value`限定大小值范围
>
> **错误键**：`required`，`invalid`，`max_value`，`min_value`

GenericIPAddressField

> 控件：`TextInput`
>
> 文本输入：`<input type="text" ...>`

> 空值：空字符串
>
> Python：`Unicode str`
>
> 可选参数
>
> `protocol`：默认值为`both`，可选`IPv4`或`IPv6`。
>
> **错误键**：`required`，`invalid`

MultipleChoiceField

> 控件：`SelectMultiple`
>
> `<select multiple='multiple'>...</select>`
>
> 空值：一个空列表
>
> Python：`list`
>
> 验证表单中的值是否存在于选择列表中，对比`ChoiceField`，该字段支持多选
>
> 必选参数：`choices`，与`ChoiceField`类似，接收一个二元组可迭代对象
>
> **错误键**：`required`，`invalid_choice`，`invalid_list`

RegexField

> 控件：`TextInput`
>
> 文本输入：`<input type="text" ...>`
>
> 空值：空字符串
>
> Python：`Unicode str`
>
> 验证表单中值与某个正则表达式匹配
>
> 必选参数：`regex`，字符串或编译的正则表达式
>
> 可选参数：`max_length`、`min_length`
>
> **错误键**：`required`，`invalid`

SlugField

> 控件：`TextInput`
>
> 文本输入：`<input type="text" ...>`
>
> 空值：空字符串
>
> Python：`Unicode`对象
>
> 验证给定的值为**字母**、**数字**、**下划线**及**连字符**组成
>
> **错误键**：`required`，`invalid`

URLField

> 控件：`TextInput`
>
> 文本输入：`<input type="text" ...>`
>
> 空值：空字符串
>
> Python：`Unicode`对象
>
> 验证给定值是一个有效的`URL`
>
> 可选参数：`max_length`、`min_length`
>
> **错误键**：`required`，`invalid`

TimeField

> 控件：`TextInput`
>
> 文本输入：`<input type="text" ...>`
>
> 空值：`None`
>
> Python：`datetime.time`
>
> 验证给定值是否为一个给定格式的时间字符串
>
> 可选参数：`input_formats`，控制表单输入的格式

表单属性

- `required`：

  > 表单字段为必填值，当传递数据为一个空值，不管是空字符串还是`None`
  >
  > 在表单验证时，将引发`ValidationError`异常，这个异常将会在表单上展示错误信息

- `label`

  > 指定当前字段的`label`标签值，字段默认`Label`为字段名所有下换线转换为空格
  >
  > 且一个字母大写生成

- `label_suffix`

  > 修改label提示字符串的追加符号，默认表单类实例化过程会自动在`label`属性后加`:`

- `initial`

  > 字段的初始值。不能将初始值直接作为参数传入，会造成直接验证表单数据而报错。

  ```
  form = forms.TestForm(initial={'name':'Bob'})
  ```

- `widget`

  > 表单字段渲染时使用的`Widget`类，如果不想使用默认的表单类型，通过该参数指明所需表单控件
  >
  > 可以使用类似的表单类型，在下面会有详细的介绍。

- `help_text`

> 指定字段的描述文本，该文本一般会紧挨着字段显示

表单控件：widget

> 默认`django`会为每一个表单字段设置默认的`HTML`控件
>
> 控件用来渲染`HTML`中输入元素与提取提交的原始数据
>
> 如果你希望使用一个不同的控件`Widget`，可以为字段设置`widget`参数

```
from django import forms
class CommentForm(forms.Form):
    comment = forms.CharField(widget=forms.Textarea)
#修改CharField默认控件TextInput为Textarea
```

> 此外，我们还可以为字段的`Widget`设置额外的属性
>
> 比如一些之后在`HTML`渲染时候将会使用到的标签`class`值等等
>
> 只需要在`widget`参数部分使用`attrs`形参指定即可，该参数设置这个字段控件的对应`HTML`属性

```
name = forms.CharField(
    max_length=5,
    widget=forms.TextInput(attrs={'class':'green'})
)
```

> 还可以使用日期控件覆盖默认日期控件

```
YEARS = ('2016', '2017', '2018')
MONTHS = {
    1:'一月', 2:'二月', 3:'三月', 4:'四月',
5:'五月', 6:'六月', 7:'七月', 8:'八月',
9:'九月', 10:'十月', 11:'十一月', 12:'十二月'
}
birth_year = forms.DateField(widget=froms.SelectDateWidget(years=YEARS,months=MONTHS))
```

文本输入控件

- `TextInput`
  > 文本输入：`<input type="text" ...>`
- `NumberInput`
  > 文本输入：`<input type="number" ...>`
- `EmailInput`
  > 文本输入：`<input type="email" ...>`
- `URLInput`
  > 文本输入：`<input type="url" ...>`

- `PasswordInput`

  > 密码输入：`<input type='password' ...>`

- `HiddenInput`

  > 隐藏输入：`<input type='hidden' ...>`

- `DateInput`

  > 日期以普通的文本框输入：`<input type='text' ...>`
  >
  > 可选参数：`format`，时间的字符串格式

- `DateTimeInput`

  > 日期/时间以普通的文本框输入：`<input type='text' ...>`
  >
  > 可选参数：`format`，时间的字符串格式

- `TimeInput`

  > 时间以普通的文本框输入：`<input type='text' ...>`
  >
  > 可选参数：`format`，时间的字符串格式

- `Textarea`

  > 文本区域：`<textarea>...</textarea>`

选择和复选框

- `CheckboxInput`

  > 复选框：`<input type='checkbox' ...>`
  >
  > 可选参数：`check_test`
  >
  > 这个参数接收一个**函数对象**，函数对象的参数为当前`CheckboxInput`的值，函数对象如果返回`True`，该控件在字段渲染时自动勾上。

```
comment = forms.CharField(widget=forms.CheckboxInput(check_test=lambda *arg: True))
```

- `Select`

  > 单选框：`<select><option ...>...</select>`
  >
  > 可选参数：`choices`，与字段设置相同，但是会被字段设置所覆盖。

- `NullBooleanSelect`

  > 单选框：选项为`Unknown`、`Yes`和`No`，`Unknown`也代表`False`。

- `SelectMultiple`

  > 多选框：`<select multiple='multiple'>...</select>`

- `RadioSelect`

  > 单选框，与`select`类似，但是会将选择渲染为一个**单选按钮列表**

- `CheckboxSelectMultiple`

  > 多选框：与`SelectMultiple`类似，但是会渲染为一个**复选框列表**

复合控件

- `SelectDateWidget`

  > 封装了三个`Widget`，分别用于年、月、日
  >
  > 可选参数：可以来指定日期表单的选择
  >
  > `years`：一个列表或元组的序列数据类型，用来确定年的选择。
  >
  > `months`：一个字典数据类型，字典的key值为月份数字，从1开始，value值为在表单中渲染展示的字符串，比如

```
MONTHS = {
    1:'一月', 2:'二月', 3:'三月', 4:'四月',
    5:'五月', 6:'六月', 7:'七月', 8:'八月',
    9:'九月', 10:'十月', 11:'十一月', 12:'十二月'
}
```

表单API

> 表单类的实例，只有两种，一种是绑定了数据的，一种是未绑定的。都可以渲染成为`html`

- `Form.is_valid()`

  > 对于绑定了数据的表单，进行验证并返回一个数据是否合法的布尔值
  >
  > 并在**所有数据**有效时将数据放入`cleaned_data`中

- `Form.is_bound()`

  > 区分绑定表单和未绑定表单，当表单类绑定数据时，返回`True`

- `Form.errors`

  > 当验证发生错误时的错误信息的字典，字典`key`值为字段名称，`value`为报错信息列表，可能有多个报错
  >
  > 表单的数据将会在调用`is_valid`时或访问`errors`属性时验证
  >
  > 并且验证过程只会调用一次，不论访问`errors`和调用`is_valid`多少次

```
class TestForm(forms.Form):
    name = forms.CharField(max_length=5,)
    email = forms.EmailField(required=True)
    def clean_name(self):
        cleaned_data = super(TestForm,self).clean()
        if self.cleaned_data.get('name') == '小红':
            raise forms.ValidationError("不允许小红")
        return cleaned_data
```

```
>>> f = forms.TestForm({'name':'小红','emali':'123'})
>>> a.errors
{'name': ['不允许小红'], 'email': ['This field is required.']}
```

- `Form.errors.as_data`

  > 返回报错信息的字典，映射字段报错信息到一个`ValidationError`实例

```
>>> f.errors.as_data()
{
    'name': [ValidationError(['不允许小红'])],
    'email': [ValidationError(['This field is required.'])]
}
```

- `Form.errors.as_json(escape_html=False)`

  > 以`json`格式返回错误信息

```
>>> a.errors.as_json()
'{
    "name": [{"message": "\\u4e0d\\u5141\\u8bb8\\u5c0f\\u7ea2", "code": ""}],
    "email": [{"message": "This field is required.", "code": "required"}]
}'
```

- `Form.initial`

  > 声明当前表单类的默认数据，参数为一个字典数据类型
  >
  > `key`对应需要填充默认数据的表单字段，`value`值为实际数据

```
class TestForm(forms.Form):
    name = forms.CharField(max_length=5,initial='Jack',)
```

```
>>> f = TestForm(initial={'name':'Bob'})
>>> print(f)
<tr><th>
<label for="id_name">Name:</label></th><td>
<input id="id_name" maxlength="5"
name="name" type="text" value="Bob" /></td></tr>
<tr><th>
```

- `Form.has_changed()`

  > 检查表单当前的数据是否与默认值不同

```
>>> f = TestForm(data={'name':'Jack'},initial={'name':'Bob'})
>>> f.has_changed()
True
```

- `Form.cleaned_data`

> 在对绑定数据的表单实例进行`is_valid`验证之后，如果数据无误
>
> 那么返回的数据将保存在`cleaned_data`中
>
> 如果有部分数据没有经过验证，那么`cleaned_data`中也会保留合法的字段
>
> 并且，`在cleaned_data`属性中获取到的数据，只包含表单类中含有的字段

```
class TestForm(forms.Form):
    name = forms.CharField(max_length=5,)
    email = forms.EmailField(required=True)
    active = forms.BooleanField()
```

```
>>> data = {
...     'name':'Jack',
...     'email':'111',
...     'active':True,
... }
>>>
>>> f = TestForm(data=data)
>>> f.is_valid()
False
>>> f.cleaned_data
{ 'name':'Jack', 'active': True}
```

- `Form.as_p()`

> 将表单渲染为一系列的`<p>`标签，每个标签内含一个字段

```
class TestFrom(forms.Form):
    name = forms.CharField(max_length=5)
```

```
>>> f = TestForm()
>>> print(f.as_p())
<p><label for="id_name">Name:</label> <input id="id_name" maxlength="5" name="name"
type="text" /></p>
```

- `Form.as_ul()`

> 渲染表单为一系列的`<li>`标签，并且不包含`<ul>`标签，可以自行指定`<ul>`的`HTML`属性

```
>>> print(f.as_ul())
<li><label for="id_name">Name:</label> <input id="id_name" maxlength="5" name="name"
type="text" /></li>
```

- `Form.as_table()`

> 渲染表单为`<tr><th>`标签

```
>>> print(f.as_table())
<tr><th><label for="id_name">Name:</label></th><td><input id="id_name" maxlength="5"
name="name" type="text" /></td></tr>
```

> 配置表单元素的`HTML id`值与默认自带的`label`标签
>
> 通过表单类进行渲染时，默认会包含以下属性

- 表单元素的`HTML id`属性
- 辅助的`label`标签

> 有些时候，想要设置自定义HTML id值或者取消label标签，可以使用如下内置函数

- `Form.auto_id=True`

  > 修改对应渲染表单属性
  >
  > 当`auto_id`值为`False`时，表单类的渲染将不会包含`<label>`以及`id`属性

```
>>> f = TestForm(auto_id=False)
>>> print(f)
<tr><th>Name:</th><td><input maxlength="5" name="name" type="text" /></td></tr>
```

模板中表单实例属性

> 模板页面接收到的`form`表单实例支持循环遍历访问

```
{% for field in form %}
        {{ field }}
{% endfor %}
```

> 其中`for`迭代访问之后的每一个表单字段又支持如下操作

- `{{ field.label }}`：字段的label，例如Email address。
- `{{ field.label_tag }}`：包含在HTML `<label>`标签中的字段值。
- `{{ field.id_for_label }}`：这个字段的ID值。
- `{{ field.value }}`：字段的值
- `{{ field.html_name }}`：该字段的标签中name属性使用的值。
- `{{ field.help_text }}`：该字段的帮助文档。
- `{{ field.errors }}`：字段的验证错误信息，字段标签会在

  属性中。
  - `{{ field.is_hidden }}`：如果该字段为隐藏字段，返回True。反之返回False。
  - `{{ field.field }}`：获取当前字段实例，可以用该属性来访问字段实例的属性

    ```
    {{ field.field.max_length }}
    ```

与模型类关联的表单

> 除了以上我们自定义表单类来进行表单的初始化
>
> `django`还提供了另外一种表单类的创建方法，可以通过与模型关联来构建表单
>
> 这种办法可以更加省时省力，直接使用模型类中已经定义好的字段来进行表单字段的生成

```
class TestTable(modes.Model):
    name = models.CharField(max_length=10)

class TestTableForm(forms.ModelForm):
    class Meta:
        model = TestTable
        fields = ['name']
```

> 生成的表单实例将具备模型类中的字段，表单生成的字段顺序也与模型类中的定义顺序相同
>
> `fields`属性用来显示的设置所有需要在表单中处理的字段
>
> 也可以直接为该字段设置`fields = '__all__'`来使用所有模型类中的字段作为未来的表单字段

- **注意**:
  - 如果模型类中字段定义了`blank=True`，那么对应关联的表单类中字段会默认具有`require=False`的属性
  - 模型类中字段的`verbose_name`属性对应关联表单类字段的Label属性
  - 如果模型类字段中设置了`choices`值，那么对应关联表单字段的`widget`将会设置为`select`

> 当然，除了根据关联模型类来创建表单类，还可以在关联表单类中选择性的覆盖某些字段的设置
>
> 比如使用表单类`Meta`元类中的`widgets`属性可以以字典形式设置对应字段的控件

```
class TestTableForm(forms.ModelForm):
    name = forms.URLField()
    class Meta:
        model = TestTable
        fields = ['name']
        widgets = {
            'name':forms.Textarea(attrs={'class':'green'})
        }
```

> 除此之外，还可以指定`labels`、`help_texts`和`error_messages`等信息

```
class TestTableForm(forms.ModelForm):
    name = forms.URLField()
    class Meta:
        model = TestTable
        fields = ['name']
        labels = {
            'name':'您的名字'
        }
        help_texts = {
            'name':'请输入您的名字'
        }
        error_messages = {
            'name':{
```

```
            'required':'你必须填写这个名字',
            'max_length':'你的名字太长了'
        }
    }
```

与模型关联的表单验证

> 表单的验证在我们调用`is_valid`函数时执行，也可以通过访问`errors`属性或调用`full_clean`函数
>
> 验证的出错会引发`ValidationError`异常，该异常会向表单传达一个错误信息

> 验证的步骤主要分为两步，表单验证，如果关联了模型，则还会进行模型验证

- **表单字段的验证**分为以下过程
    1. 字段`to_python`，这个方法将字段的值根据字段的类型转换为Python中的数据类型，如果不能转换则引发`ValidationError`异常
    2. 字段的`clean`函数，该函数用来运行对应的验证器，根据顺序执行`to_python`，`validate`特异性验证，以及`run_validators`（用于将错误信息汇总）验证，如果有任何验证过程引发了`ValidationError`异常，验证都将停止。其余通过验证的字段数据插入到表单的cleaned_data字典中
    3. 表单中的字段`clean`函数，这个验证用于完成特定属性，与表单字段类型无关；比如我们经常需要验证用户输入的字段值不能为小红，那么可以编写字段的`clean`函数，函数命名为`clean_<fields_name>`，`fileds_name`为字段名

        ```
        class TestTableForm(forms.ModelForm):
        def clean_name(self):
            name = self.cleaned_data.get('name')
            if name == '小红':
                raise forms.ValidationError('不允许小红')
            return name
        ```

    4. 表单的`clean`函数，这个方法进行表单中多个字段值的联合验证，验证之后的数据返回为`cleaned_data`，可以通过重写该函数来提供的额外验证方法，并且为了维持`clean`方法的验证行为，在代码中，表单类需要调用父类的`clean`方法

        ```
        def clean(self):
            cleaned_data = super(TestTableForm,self).clean()
            name = cleaned_data.get('name')
            if '1' in name:
                cleaned_data['name'] = name.replace('1','一')
            return cleaned_data
        ```

> 最后总结的来说：
>
> 一个表单在验证时，首先验证每一个字段，接着调用字段的`clean_fields`函数，最后使用表单类的`clean`函数进行验证

> 如果表单与模型关联，那么现在还有第二步验证，模型的验证

- **模型的验证**为如下过程

1. 验证关联模型的字段及相关属性：`Model.clean_fields(exclud=None)`，该方法将验证模型的所有字段属性，如果有字段验证错误，引发`ValidationError`异常
2. 验证模型的完整性：`Model.clean(exclude=None)`，可以对模型做整体的检验，如果想要自己验证模型中通过属性校验的数据，可以在模型类中重新定义这个函数

```
from django.core.exceptions import ValidationError
class TestTable(models.Model):
    name = models.CharField(max_length=10,verbose_name='名字',unique=True)
    def clean(self):
        if '$' in self.name:
            raise ValidationError('无法使用$符号')
```

3. 验证模型的唯一性：`Model.validate_unique(exclude=None)`，如果模型中所有唯一约束性，比如使用类似`unique`属性，会校验表单中的值是否唯一

> 并且，除了通过绑定模型的表单实例`is_valid`函数可以来进行以上的验证过程，如果想自己控制验证
>
> 可以直接使用模型的`full_clean(exclude=None,validate_unique=True)`方法进行以上三个步骤的验证

与模型关联的表单保存

> 与模型关联的表单，在校验成功之后，表单实例可以直接通过`save`函数来进行表单数据的保存数据库

```
def form_test(request):
    if request.method == "POST":
        form = TestTableForm(request.POST,request.FILES)
        if form.is_valid():
            form.save()
    return HttpResponse('OK:%s' % value)
```

> 该函数也支持在模型类中进行重写，但是要切记使用父类的save方法，确保数据可以正确存储到数据库中

```
def save(self, *args, **kwargs):
    if self.name == 'abc':
        return False#不做存储
    else:
        super(TestTable,self).save(*args, **kwargs)
```