二、使用配置文件和fileConfig()函数实现日志配置

现在我们通过配置文件的方式来实现与上面同样的功能：

```
# 读取日志配置文件内容
logging.config.fileConfig('logging.conf')

# 创建一个日志器logger
logger = logging.getLogger('simpleExample')

# 日志输出
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

配置文件 `logging.conf` 内容如下：

```
[loggers]
keys=root,simpleExample

[handlers]
keys=fileHandler,consoleHandler

[formatters]
keys=simpleFormatter

[logger_root]
level=DEBUG
handlers=fileHandler

[logger_simpleExample]
level=DEBUG
handlers=consoleHandler
qualname=simpleExample
propagate=0

[handler_consoleHandler]
class=StreamHandler
args=(sys.stdout,)
level=DEBUG
formatter=simpleFormatter

[handler_fileHandler]
class=FileHandler
args=('logging.log', 'a')
level=ERROR
formatter=simpleFormatter

[formatter_simpleFormatter]
format=%(asctime)s - %(name)s - %(levelname)s - %(message)s
datefmt=
```

三、使用字典配置信息和dictConfig()函数实现日志配置

Python 3.2中引入的一种新的配置日志记录的方法--用字典来保存logging配置信息。这相对于上面所讲的基于配置文件来保存能更加强大，也更加灵活，因为我们可把很多的数据转换成字典。比如，我们可以使用JSON格式的配置文件、YAML格式的配置字典中；或者，我们也可以用Python代码构建这个配置字典，或者通过socket接收pickled序列化后的配置信息。总之，你可任何方法来构建这个配置字典。

这个例子中，我们将使用YAML格式来完成与上面同样的日志配置。

首先需要安装PyYAML模块：

```
pip install PyYAML
```

Python代码：

```
import logging
import logging.config
import yaml

with open('logging.yml', 'r') as f_conf:
    dict_conf = yaml.load(f_conf)
logging.config.dictConfig(dict_conf)

logger = logging.getLogger('simpleExample')
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

logging.yml配置文件的内容：

```
version: 1
formatters:
  simple:
    format: '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
handlers:
  console:
    class: logging.StreamHandler
    level: DEBUG
    formatter: simple
    stream: ext://sys.stdout
  console_err:
    class: logging.StreamHandler
    level: ERROR
    formatter: simple
    stream: ext://sys.stderr
loggers:
  simpleExample:
    level: DEBUG
    handlers: [console]
    propagate: yes
root:
  level: DEBUG
  handlers: [console_err]
```

Django 中的 log

```
#LOGGING_DIR 日志文件存放目录
LOGGING_DIR = "logs"   # 日志存放路径
if not os.path.exists(LOGGING_DIR):
    os.mkdir(LOGGING_DIR)

import logging

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {  #格式化器
        'standard': {
            'format': '[%(levelname)s][%(asctime)s][%(filename)s][%(funcName)s][%(lineno)d] > %(message)s'
        },
        'simple': {
            'format': '[%(levelname)s]> %(message)s'
        },
    },
    'filters': {
        'require_debug_true': {
            '()': 'django.utils.log.RequireDebugTrue',
        },
    },
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'filters': ['require_debug_true'],
            'class': 'logging.StreamHandler',
            'formatter': 'simple'
        },
        'file_handler': {
             'level': 'INFO',
             'class': 'logging.handlers.TimedRotatingFileHandler',
             'filename': '%s/django.log' % LOGGING_DIR,  #具体日志文件的名字
             'formatter':'standard'
        }, # 用于文件输出
        'mail_admins': {
            'level': 'ERROR',
            'class': 'django.utils.log.AdminEmailHandler',
             'formatter':'standard'
        },
    },
    'loggers': {   #日志分配到哪个handlers中
        'mydjango': {
            'handlers': ['console','file_handler'],
            'level':'DEBUG',
            'propagate': True,
        },
        'django.request': {
            'handlers': ['mail_admins'],
            'level': 'ERROR',
            'propagate': False,
        },
      ####如果要将get,post请求同样写入到日志文件中，则这个触发器的名字必须交django,然后写到handler中
    }
}
```

```
'loggers': {
    'mydjango': {
        'handlers': ['console', 'file_handler'], #无论怎么样，都要写在一个列表中，单独的也是只写一个
        'level': 'INFO',
        'propagate': True, #参数解释，是否上传到上级文件，上级指的是上一级，例如debug上一级是info
    },
    'django.request': {
        'handlers': ['mail_admins', 'file_handler'],
        'level': 'INFO',
        'propagate': False,
    },
    'django': {
        'handlers': ['console',],
        'level': 'INFO',
        'propagate': False
    },
```

如果要将get,post请求写到日志中，名字必须叫django

在settings中写完配置之后，下面就要具体到相应的视图函数中了。**注意：这里我们在实施到具体函数或方法的时候需要手动捕获异常**

格式	描述
%(name)s	记录器的名称
%(levelno)s	数字形式的日志记录级别
%(levelname)s	日志记录级别的文本名称
%(filename)s	执行日志记录调用的源文件的文件名称
%(pathname)s	执行日志记录调用的源文件的路径名称
%(funcName)s	执行日志记录调用的函数名称
%(module)s	执行日志记录调用的模块名称
%(lineno)s	执行日志记录调用的行号
%(created)s	执行日志记录的时间
%(asctime)s	日期和时间
%(msecs)s	毫秒部分
%(thread)d	线程ID
%(threadName)s	线程名称
%(process)d	进程ID
%(message)s	记录的消息

民主

```
#这里我们模拟一个登陆来写

log = logging.getLogger('mydjango')  #这里的mydjango是settings中loggers里面对应的名字


class Login(View):
    def get(self,request):
        return render(request,'login.html')
    def post(self,request):
        user = request.POST.get('username')
        pwd = request.POST.get('password')
        try:
            s = 1/0
        except Exception as e:
            log.error(e)
```

后台 admin

使用站点管理

Django中默认集成了后台数据管理页面，通过简单的配置就可以实现模型后台的Web控制台。

管理界面通常是给系统管理员使用的，用来完成数据的输入，删除，查询等工作。

如果没有集成，自己在settings中的INSTALLED_APPS添加'django.contrib.admin'应用就好了

首先我们需要创建一个系统管理员

python manager.py createsuperuser

根据提示创建自己的管理员

管理界面设置

设置为中文

settings中LANGUAGE_CODE = 'zh-hans'

设置时间，时区

TIME_ZONE='Asia/Shanghai'

添加自己的数据模型

在admin.py中注册

admin.site.register(xxx)

端

- 用户端
- 公司自己的后台
- 商家端

后台管理

- 快速实现自己的后台

Django内置模型

- 用户
- 组

后台管理

- 快速实现自己的后台
- 内置了一个admin
- 也有第三方的
  - xadmin

个性化站点管理

如果你感觉默认的站点样式不能满足应用需求，则开发者可以通过继承Django定义的管理员数据模型，模板，站点类来开发出个性化的管理员站点。

注册的时候添加自己的管理类

创建管理类

```
class StudentAdmin(admin.ModelAdmin):
    # 规则
```

注册管理类

```
admin.site.register(Students,StudentAdmin)
```

个性化规则

显示规则

```
list_display    显示字段
list_filter     过滤字段
search_fields   搜索字段
list_per_page   分页，每页显示多少条数据
ordering_       排序规则
分组显示
        fieldsets = (
                ('班级',{fields:('sgrade')}),
                ('姓名',{fields:('sname')}),
            )
```

修改规则

```
fields          显示的字段
exclude         不显示的字段
```

布尔值定制显示

比如性别，在list_display的时候可以传递一个函数

```
def gender(self):
    if self.sgender:
        return '男'
    else:
        return '女'

list_display = ('sname', 'sage', gender)
# 设置显示的标题
gender.short_description = '性别'
```

```python
class StudentAdmin(admin.ModelAdmin):

    def sex(self):
        if self.s_sex:
            return '女'
        else:
            return '男'

    sex.short_description = '性别'

    list_display = 's_name', 's_age', sex

    fieldsets = (
        ('基本信息', {'fields': ('s_name', 's_age', 's_sex')}),
        ('可选信息', {'fields': ('s_height', 's_weight')}),
    )


admin.site.register(Student, StudentAdmin)
```

插入班级的时候，同时插入两个学生

创建班级的Admin

```
class GradeAdmin(admin.ModelAdmin):
    inlines = [StudentInfo]
```

创建学生信息

```
class StudentInfo(admin.TabularInline):
    model = Student
    extra = 2

admin.site.register(Grade,GradeAdmin)
```

```python
class StudentInfo(admin.TabularInline):
    extra = 3
    model = Student


class GradeAdmin(admin.ModelAdmin):
    list_display = 'g_name', 'g_postion'
    inlines = [StudentInfo]


admin.site.register(Grade, GradeAdmin)
```

这里 extra 表示必须创建三个学生才能创建一个班级

个性化定制

还有一种覆盖系统模板的方式，在
django/contrib/admin/templates/admin中，

将需要重新定制的文件复制出来，

在自己的工程中创建相对应的子目录，

在settings中注入模板路径，子模板继承自复制的模板

添加自己的代码

定制站点信息

在admin中继承自AdminSite

```
class MyAdminSite(admin.AdminSite):
    site_header = 'Rock小学堂'

site = MyAdminSite()
site.register(xxx)
```

最后在urls路由中修改admin的跳转

```
from App.admin import site

url(r'^admin/',site.urls)
```

常用定制属性

site_header:管理网页的页头部的标题

site_title:浏览窗口显示的页面名称

site_url:查看站点时的跳转，也就是主,默认 /

```
class MyAdminSite(admin.AdminSite):

    site_title = 'RockLearn'
    site_header = 'Rock'
    site_url = '/send/home/'
```