Python Web框架Django课程

Django第7天

缓存

有些操作处理过程需要消耗大量时间，而且在一定时间段内，该操作的处理结果是不变的，那么可以将该操作结果，保存下来，短时间内在此访问时，直接读取缓存的结果而不需要在此执行耗时的处理

当使用缓存之后，访问流程将会发生改变

请求－－>处理函数－－>访问缓存－－>第一次缓存不存在－－>执行实际的操作－－>操作结果保存到缓存中
请求－－>处理函数－－>访问缓存－－>缓存存在－－>直接放回缓存的数据

缓存存储方式：
文件
数据库
内存数据库
本地内存

Django中默认缓存方式是使用数据库进行缓存。Django的缓存系统提供了很好的扩展接口，可以自行扩展缓存系统

配置django数据库方式缓存

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache',
        'LOCATION': 'my_cache_table',缓存数据库表名称
    }
}
```

python manage.py createcachetable 此命令将会在数据库中创建名称为my_cache_table的表

缓存参数：
TIMEOUT: 默认的缓存过期时间,这个值默认是300秒,如果设置为None表示缓存永不过期
OPTIONS: 不同缓存库使用的参数
KEY_PREFIX:cache是键值对形式存储的,key_prefix自动添加到左右键的前面

使用

```
from django.views.decorators.cache import cache_page

@cache_page(60)
def cache_page(request):
      #耗时的操作
```

```
    return HttpResponse('结果')
```

django-redis的使用
安装django-redis ：**pip installdjango-redis**
配置

```
CACHES = {
    "default": {
        "BACKEND": "django_redis.cache.RedisCache",
        "LOCATION": "redis://127.0.0.1:6379/1",
        "OPTIONS": {
            "CLIENT_CLASS": "django_redis.client.DefaultClient",
            "PASSWORD": "redis_password"
        }
    }
}
```

日志

日志可以用来追中系统在运行时,所发生的事件,通过分析日志，可以了解系统的健康状况，调试系统的bug等。
日志记录的内容，分为不同级别。

1. DEBUG 最详细的日志信息，典型应用场景是问题诊断
2. INFO 信息详细程度仅次于DEBUG，记录一些提示信息
3. WARNNING 系统运行出现了异常
4. ERROR 系统出现了严重问题,导致某些功能不正常
5. CRITICAL 系统发生了严重错误,导致程序不能运行

DEBUG<INFO<WARNNING<ERROR<CRITICAL

记录日志的注意点:
记录日志需要额外的系统资源，比如文件类型的日志需要消耗I/O。所以生产环境不应当记录DEBUG、INFO级别的信息。

日志4大组件
loggers:提供应用程序代码直接使用的接口,包含多个logger,主要用来获取logger对象,每一个logger通常包含多个handlers
handlers:用于将日志信息进行处理，例如:保存到文件当中
filters:过滤器，提供更细粒度的日志过滤功能，用于决定哪些日志记录将会被输出formmater:日志格式
formmaters:用于控制日志信息的最终输出格式

python中使用logging模块来记录日志

```
import logging
logging.debug
logging.info
```

logging.warn

日志格式化

asctime	%(asctime)s	日志事件发生的时间--人类可读时间，如：2003-07-08 16:49:45,896
created	%(created)f	日志事件发生的时间--时间戳，就是当时调用time.time()函数返回的值
relativeCreated	%(relativeCreated)d	日志事件发生的时间相对于logging模块加载时间的相对毫秒数（目前还不知道干嘛用的）
msecs	%(msecs)d	日志事件发生事件的毫秒部分
levelname	%(levelname)s	该日志记录的文字形式的日志级别（'DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL'）
levelno	%(levelno)s	该日志记录的数字形式的日志级别（10, 20, 30, 40, 50）
name	%(name)s	所使用的日志器名称，默认是'root'，因为默认使用的是 rootLogger
message	%(message)s	日志记录的文本内容，通过计算得到的
pathname	%(pathname)s	调用日志记录函数的源码文件的全路径
filename	%(filename)s	pathname的文件名部分，包含文件后缀
module	%(module)s	filename的名称部分，不包含后缀
lineno	%(lineno)d	调用日志记录函数的源代码所在的行号
funcName	%(funcName)s	调用日志记录函数的函数名
process	%(process)d	进程ID
processName	%(processName)s	进程名称，Python 3.1新增
thread	%(thread)d	线程ID
threadName	%(thread)s	线程名称

loggers配置

```
loggers:{
    'logger1':{
        'level':最低级别的日志,低于此级别,被忽略
        'handlers':[handler1,handler2],
    }
}
handlers:{
    'handler1':{
        'level': 最低级别的日志,低于此级别,被忽略,
        'class': 日志处理类,
        'formatter':日志格式化参数
    }
}
```

常用handler:

保存到文件

```
'file': {
            'level': 'WARN',
            'class': 'logging.handlers.TimedRotatingFileHandler',
            'formatter': 格式化名,
            'filename':文件名,
            # 每天一个新文件
            'when': 'D',
            'filters':['过滤器',]
}
```

打印到终端

```
'console': {
            'level': 'INFO',
            'class': 'logging.StreamHandler',
            'formatter': 'verbose'
        },
```

formatters:{
'formater1':{
'format':'[%(asctime)s](%(levelname)s)<%(name)s.%(funcName)s>{%(process)d/%(thread)d} : %(message)s'
},
}

日志处理过程:
获取logger—>logger判断日志是否在logger的level级别上—>不在就忽略|如果在—>日志传递给handler—>handler判断日志是否level级别上—>不在就忽略

Django使用python内置的logging来实现日志系统
Django中配置日志

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': True,
    'formatters': {
        'verbose': {
            'format': '[%(asctime)s](%(levelname)s)<%(name)s.%(funcName)s> : %(message)s'
        },
    },
    # 过滤器
    'filters': {},
```

```
    'handlers': {
        'console': {
            'level': 'INFO',
            'class': 'logging.StreamHandler',
            'formatter': 'verbose'
        },
        'file': {
            'level': 'WARN',
            'class': 'logging.handlers.TimedRotatingFileHandler',
            'formatter': 'verbose',
            'filename': 'log/webservice.log',
            # 每天一个新文件
            'when': 'D',
        },
    },
    'loggers': {
        'customer': {
            'handlers': ['file','console'],
            'level': 'DEBUG',
        },
    }
}
```

信号

用于对复杂的操作进行解耦。系统中事件之间通常有一定的关系，一旦某些特定事件发生，那么其他事件也会跟着发生。

传统编程中:
特定事件处理函数中调用其他事件处理函数

特定事件处理函数
special_fun1:

1. 调用other_fun1
2. 调用other_fun2
3. 调用other_fun3
4. 调用other_fun....

special_fun2:

1. 调用other_fun1
2. 调用other_fun3

3. 调用other_fun6

代码冗余,难于管理

信号系统

信号的三元素:
1. 接收者
2. 发送者
3. 信号

信号系统构建过程:
1. 定义信号
2. 注册信号处理函数
3. 发送信号

django中自带的信号

```
Model signals

    pre_init                    # django的modal执行其构造方法前，自动触发

    post_init                   # django的modal执行其构造方法后，自动触发
    pre_save                    # django的modal对象保存前，自动触发
    post_save                   # django的modal对象保存后，自动触发
    pre_delete                  # django的modal对象删除前，自动触发
    post_delete                 # django的modal对象删除后，自动触发
    m2m_changed                 # django的modal中使用m2m字段操作第三张表（add,remove,clear）前
后，自动触发

    class_prepared              # 程序启动时，检测已注册的app中modal类，对于每一个类，自动触发

Management signals

    pre_migrate                 # 执行migrate命令前，自动触发

    post_migrate                # 执行migrate命令后，自动触发
Request/response signals

    request_started             # 请求到来前，自动触发

    request_finished            # 请求结束后，自动触发
    got_request_exception       # 请求异常后，自动触发
Test signals

    setting_changed             # 使用test测试修改配置文件时，自动触发

    template_rendered           # 使用test测试渲染模板时，自动触发
Database Wrappers

    connection_created          # 创建数据库连接时，自动触发
```

将自定义函数绑定到django中自带的信号上:

定义回调函数

def call_back(sender,**kwargs):

print(sender,kwargs)

绑定处理函数

```
from django.db.models.signals import pre_save
pre_save.connect(call_back)
```

自定义信号

```
from django.dispatch import Signal
test_signal=Signal(providing_args=['para1','para2'])
```

自定义处理函数

```
def signal_call_back(sender,**kwargs):
    print(sender,kwargs)
```

注册信号处理函数:

```
test_signal.connect(signal_call_back)
```

发送信号:

```
test_signal.send('province_page',para1='para11',para2='para22')
```