Python实用教程系列-Loggingr日志模块

作为Python的实用教程之一，上篇推文，我们讲述了Python中的相关异常的知识，包括异常的种类以及异常的处理方式，以及使用实力的方式说明了一些常用异常的处理。这次，我们来讲述一下Python中的另一个实用的教程，日志相关的知识。

很多的小伙伴在平时写代码的时候，基本上使用的都是使用print()函数来进行一些日志的打印，使用print()来进行一些代码的调试，在较小的程序中这样做是可以的。但是在实际的工程项目中，我们需要保存程序运行的日志，以排查程序在某一个时候崩溃的具体原因，以便及时定位Bug进行抢救，而在Python中，logging就可以十分灵活的帮助我们记录程序运行的过程的中的一些信息。

那么日志的作用是什么呢？归纳起来日志的作用有三点：

1. 进行程序(代码)的调试
2. 程序运行过程中的问题定位和分析
3. 收集程序运行的情况

接下来我们就来介绍Python中的标准库Logging的相关知识：

1. **日志等级**

Logging中的日志等级如下：

|  |  |
| --- | --- |
| **等级(低到高)** | **对应描述** |
| DEBUG | 调试，记录最详细日志信息，在问题诊断时有用 |
| INFO | Information简称，记录一些设置的关键节点信息 |
| WARNING | 警告，记录不期望的事情发生时的信息（如，磁盘可用空间较低），程序仍正常 |
| ERROR | 错误，记录由于更严重的问题而导致软件无法执行某些功能时的信息 |
| CRITICAL | 严重，记录发生严重错误，程序本身可能无法继续运行时的信息 |

使用Python的小伙伴对WARNING和ERROR特别的熟悉，在一些模块版本过低的时候运行程序就会出现WARNING的警告。我们来看一些代码:

import logging  
import logging  
*# 设置打印日志的级别，level级别以上的日志会打印出  
# level=logging.DEBUG 、INFO 、WARNING、ERROR、CRITICAL*def log\_testing():

*# 此处进行Logging.basicConfig() 设置，后面设置无效*  
 logging.basicConfig(level=logging.ERROR)  
 logging.debug(**'debug，用来打印一些调试信息，级别最低'**)  
 logging.info(**'info，用来打印一些正常的操作信息'**)  
 logging.warning(**'waring，用来用来打印警告信息'**)  
 logging.error(**'error，一般用来打印一些错误信息'**)  
 logging.critical(**'critical，用来打印一些致命的错误信息，等级最高'**)

log\_testing()

当然了，上述的一些信息是输出到控制台的，下文我们将介绍日志输出到文件的方法。细心的小伙伴可能发现logging.basicConfig(level=logging.INFO) 这段代码，这段代码主要是用来配置日志的输出的，代码会将level以上级别日志输出，比如设置level=logging.ERROR。稍微总结一下就是：程序在运行的过程中会记录下大于或等于这个日志级别的日志信息，如果不设置level的值则打印大于等于 WARNING 级别的日志。

1. **Logging.basicConfig()**

logging.basicConfig()的参数除了日志等级的这个参数可以设置以外，还可以设置其他的参数，比如：

|  |  |
| --- | --- |
| **参数名称** | **参数作用说明** |
| filename | 指定保存日志的文件名 |
| filemode | 如果指明了文件名，指明打开文件的模式（如果没有指明 filemode，默认为 'a'） |
| format | 指定输出的格式和内容(下文会详解) |
| datefmt | 指定时间格式，同time.strftime() |
| level | 指定根日志记录器级别，默认为logging.WARNING |
| stream | 指定日志的输出流，可以指定输出到sys.stderr,sys.stdout或者文件，默认输出到sys.stderr，当stream和filename同时指定时，stream被忽略 |

上表提到日志的输出格式参数format,其控制着日志输出的一些格式：

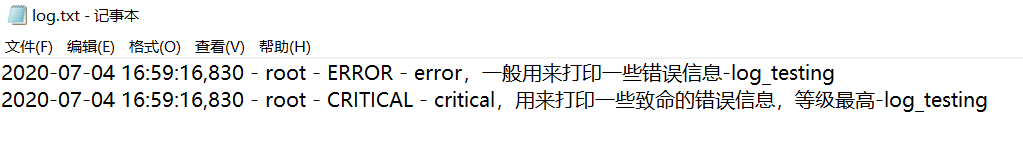
|  |  |
| --- | --- |
| **对应格式** | **格式描述** |
| %(levelno)s | 日志级别 |
| %(levelname)s | 日志级别名称 |
| %(pathname)s | 当前程序的路径 |
| %(filename)s | 当前程序名称 |
| %(funcName)s | 日志所属的当前函数 |
| %(lineno)d | 日志的当前行号 |
| %(asctime)s | 日志的时间 |
| %(thread)d | 线程的ID |
| %(threadName)s | 线程的名称 |
| %(process)d | 进程的ID |
| %(message)s | 日志的信息 |

我们来看看怎么使用：

import logging  
*# 设置打印日志的级别，level级别以上的日志会打印出  
# level=logging.DEBUG 、INFO 、WARNING、ERROR、CRITICAL*def log\_testing():  
 *# 此处进行Logging.basicConfig() 设置，后面设置无效* logging.basicConfig(filename=**'log.txt'**,  
 format = **'%(asctime)s - %(name)s - %(levelname)s - %(message)s-%(funcName)s'**,  
 level=logging.ERROR)  
 logging.debug(**'debug，用来打印一些调试信息，级别最低'**)  
 logging.info(**'info，用来打印一些正常的操作信息'**)  
 logging.warning(**'waring，用来用来打印警告信息'**)  
 logging.error(**'error，一般用来打印一些错误信息'**)  
 logging.critical(**'critical，用来打印一些致命的错误信息，等级最高'**)  
  
log\_testing()

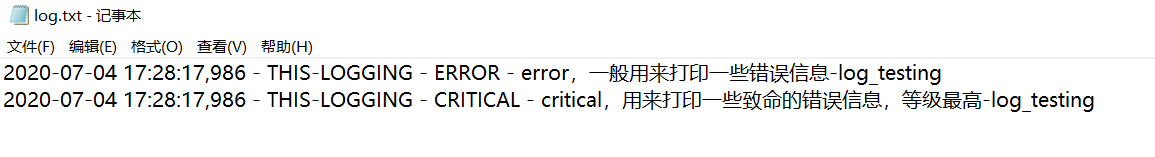
filemode也是一个常用的参数，不设的话默认为‘a’，即追加模式；也可以设为‘w’，那么每次写日志会覆盖之前的日志。

上述代码产生的日志文件截图如下：



上述日志详情中出现了一个root字眼，这是root日志器的意思，我们也可以使用logging.getLogger()函数来自定义一个日志器：

import logging  
def log\_testing():  
 selflogger = logging.getLogger(**'THIS-LOGGING'**)  
 logging.basicConfig(filename=**'log.txt'**,  
 format = **'%(asctime)s - %(name)s - %(levelname)s - %(message)s-%(funcName)s'**,  
 level=logging.ERROR)  
 selflogger.warning(**'waring，用来用来打印警告信息'**)  
 selflogger.error(**'error，一般用来打印一些错误信息'**)  
 selflogger.critical(**'critical，用来打印一些致命的错误信息，等级最高'**)  
  
log\_testing()



可以看到在日志结果文件中我，THIS-LOGGING代替了原来的root日志器了

1. **Logging组件**

Logging中除了以上的一些设置之外，logging还提供了几个组件（类）供我们实现一些特殊的功能，它们主要是：

|  |  |  |
| --- | --- | --- |
| **组件名称** | **对应类名** | **对应描述** |
| 日志器 | Logger | 提供了应用程序可以一直使用的接口 |
| 处理器 | Handler | 将创建的日志记录发送到合适的目的输出 |
| 过滤器 | Filter | 提供了更细粒度的控制工具来决定输出哪条日志记录，丢弃哪条日志记录 |
| 格式器 | Formatter | 决定日志记录的最终输出格式 |

**这些组件共同完成日志的配置和输出：**

* Logger需要通过handler将日志信息输出到目标位置，目标位置可以是sys.stdout和文件等。
* 不同的Handler可以将日志输出到不同的位置(不同的日志文件)。
* Logger可以设置多个handler将同一条日志记录输出到不同的位置。
* 每个Handler都可以设置自己的filter从而实现日志过滤，保留实际项目中需要的日志。
* formatter实现同一条日志以不同的格式输出到不同的地方。

**简单的说就是：**日志器作为入口，通过设置处理器的方式将日志输出，处理器再通过过滤器和格式器对日志进行相应的处理操作。

我们来简单的介绍一下这些组件中一些常用的方法：

**Logger类(**Logger = logging.getLogger()**)**

|  |  |
| --- | --- |
| **方法** | **方法描述** |
| Logger.setLevel() | 设置日志器将会处理的日志消息级别 |
| Logger.addHandler() | 添加一个handler对象 |
| Logger.removeHandler() | 移除一个handler对象 |
| Logger.addFilter() | 添加一个filter对象 |
| Logger.removeFilter() | 移除一个filter对象 |
| Logger.debug() | 设置DEBUG级别的日志记录 |
| Logger.info() | 设置INFO级别的日志记录 |
| Logger.warning() | 设置WARNING级别的日志记录 |
| Logger.error() | 设置ERROR级别的日志记录 |
| Logger.critical() | 设置CRITICAL级别的日志记录 |
| Logger.exception() | 输出堆栈追踪信息 |
| Logger.log() | 设置一个自定义的level参数来创建一个日志记录 |

**Handler类**

|  |  |
| --- | --- |
| **方法** | **方法描述** |
| Handler.setLevel() | 设置处理器将会处理的日志消息的最低严重级别 |
| Handler.setFormatter() | 为处理器设置一个格式器对象 |
| Handler.addFilter() | 为处理器添加一个过滤器对象 |
| Handler.removeFilter() | 为处理器删除一个过滤器对象 |
| logging.StreamHandler() | 将日志消息发送到输出到Stream，如std.out, std.err |
| logging.FileHandler() | 将日志消息发送到磁盘文件，默认情况文件大小会无限增长 |
| RotatingFileHandler() | 将日志消息发送到磁盘文件，支持日志文件按大小切割 |
| TimedRotatingFileHandler() | 将日志消息发送到磁盘文件，并支持日志文件按时间切割 |
| logging.handlers.HTTPHandler() | 将日志消息通过GET或POST的方式发送给一个HTTP服务器 |
| logging.handlers.SMTPHandler() | 将日志消息发送email地址 |

**Filter类**

看名字大家就知道这是一个过滤类，那么过滤什么呢？Filter可以被Handler和Logger用来做比之前设置的日志等级level更为细粒度的、更复杂的相关过滤功能。简单的说Filter是一个过滤器基类，它只允许某个logger层级下的日志事件通过过滤，保存下来。该类定义如下：

class logging.Filter(name='')

filter(record)

**过滤举例：**

例如用 ‘A.B’ 初始化的 Filter，那么其允许Logger ‘A.B’, ‘A.B.C’, ‘A.B.C.D’, ‘A.B.D’ 等日志记录的事件，logger‘A.BB’, ‘B.A.B’ 等就不满足过滤的条件。如果用空字符串来对Filter初始化，所有日志记录的事件都将不会被过滤。

**Formater类**

Formater类主要负责日志的格式化输出的。

可通过logging.basicConfig或logging.Formatter函数来配置日志输出内容。

如：formatter = logging.Formatter(fmt=None, datefmt=None)。

如果不指明 fmt，将默认使用 '%(message)s' ，如果不指明  datefmt，将默认使用 ISO8601 日期格式。

说了这么多，我们先看两个简单的例子：

*# 例1*

import logging  
from logging.handlers import RotatingFileHandler  
  
def logging\_fun():  
 *# 创建日志的记录等级设* logging.basicConfig(level=logging.DEBUG)  
 *# 创建日志记录器，指明日志保存的路径，每个日志文件的最大值，保存的日志文件个数上限* log\_handle = RotatingFileHandler(**"log.txt"**, maxBytes=1024 \* 1024, backupCount=5)  
 *# 创建日志记录的格式* formatter = logging.Formatter(**"format = '%(asctime)s - %(name)s - %(levelname)s - %(message)s-%(funcName)s',"**)

*# 为创建的日志记录器设置日志记录格式* log\_handle.setFormatter(formatter)  
 *# 为全局的日志工具对象添加日志记录器* logging.getLogger().addHandler(log\_handle)  
 logging.warning(**'用来用来打印警告信息'**)  
 logging.error(**'一般用来打印一些错误信息'**)  
 logging.critical(**'用来打印一些致命的错误信息，等级最高'**)  
  
logging\_fun()

*# 例2*

import logging  
def log\_file():  
 log\_file = **'testfun.log'** handler\_test = logging.FileHandler(log\_file) *# stdout to file* handler\_control = logging.StreamHandler() *# stdout to console* handler\_test.setLevel(**'ERROR'**) *# 设置ERROR级别* handler\_control.setLevel(**'INFO'**) *# 设置INFO级别* selfdef\_fmt = **'%(asctime)s - %(funcName)s - %(levelname)s - %(message)s'** formatter = logging.Formatter(selfdef\_fmt)  
 handler\_test.setFormatter(formatter)  
 handler\_control.setFormatter(formatter)  
  
 logger = logging.getLogger(**'updateSecurity'**)  
 logger.setLevel(**'DEBUG'**) *#设置了这个才会把debug以上的输出到控制台* logger.addHandler(handler\_test) *#添加handler* logger.addHandler(handler\_control)  
 logger.info(**'info,一般的信息输出'**)  
 logger.warning(**'waring，用来用来打印警告信息'**)  
 logger.error(**'error，一般用来打印一些错误信息'**)  
 logger.critical(**'critical，用来打印一些致命的错误信息，等级最高'**)  
  
log\_file()

上述代码使用了四个组件中的一些简单的方法，大家可以自己看一下，很简单。

1. **配置文件**

**.conf形式的配置文件**

我们知道在logging.basicConfig()中，是进行一些日志的配置的，如果每次都去改动代码，那将变得十分的麻烦，特别是在程序发布或者上线的时候。所以我们可以使用一个什么好的方法来规避这个问题呢？有的，那就是配置文件。

我们在文件loguser.conf中写入相关的信息，内容如下：

[loggers]  
keys=root,fileLogger,rotatingFileLogger  
  
[handlers]  
keys=consoleHandler,fileHandler,rotatingFileHandler  
  
[formatters]  
keys=simpleFormatter  
  
[logger\_root]  
level=INFO  
handlers=consoleHandler  
  
[logger\_fileLogger]  
level=INFO  
handlers=fileHandler  
qualname=fileLogger  
propagate=0  
  
[logger\_rotatingFileLogger]  
level=INFO  
handlers=consoleHandler,rotatingFileHandler  
qualname=rotatingFileLogger  
propagate=0  
  
[handler\_consoleHandler]  
class=StreamHandler  
level=INFO  
formatter=simpleFormatter  
args=(sys.stdout,)  
  
[handler\_fileHandler]  
class=FileHandler  
level=INFO  
formatter=simpleFormatter  
args=(**"logs/fileHandler\_test.log"**, **"a"**)  
  
[handler\_rotatingFileHandler]  
class=handlers.RotatingFileHandler  
level=WARNING  
formatter=simpleFormatter  
args=(**"logs/rotatingFileHandler.log"**, **"a"**, 10\*1024\*1024, 50)  
  
[formatter\_simpleFormatter]  
format=%(asctime)s - %(module)s - %(levelname)s -%(thread)d : %(message)s  
datefmt=%Y-%m-%d %H:%M:%S

那么使用这个配置文件的代码可以如下：

import logging  
import os  
from logging import config  
  
def log\_config():  
 *# '读取日志配置文件'* path = **r'C:\Users\TEN\Desktop\ Logg\log\_config.conf'** if os.path.exists(path):  
 with open(path,**"r"**,encoding = **'utf-8'**) as f:  
 logging.config.fileConfig(f)  
 *# 创建一个日志器logger* logger = logging.getLogger(name=**"fileLogger"**)  
 rotating\_logger = logging.getLogger(name=**"rotatingFileLogger"**)  
  
 logger.debug(**'debug'**)  
 logger.info(**'info'**)  
 logger.warning(**'warn'**)  
 logger.error(**'error'**)  
 logger.critical(**'critical'**)  
  
 rotating\_logger.debug(**'debug'**)  
 rotating\_logger.info(**'info'**)  
 rotating\_logger.warning(**'warn'**)  
 rotating\_logger.error(**'error'**)  
 rotating\_logger.critical(**'critical'**)  
  
log\_config()

具体的输出这里就不展开了，大家可以敲击代码试试。

细心的同学可能发现[logger\_fileLogger]和[logger\_rotatingFileLogger]中有一个属性值propagate，在我们的例子中propagate被设置为0的，实际上这个值默认为1，这两者有什么区别呢？

如果将logger\_fileLogger和logger\_rotatingFileLogger的属性值propagate设置为1，那么这些个logger在处理完日志记录后，把日志记录传递给了上级的logger\_root再次进行一些处理，这样的话我们设置的保存日志的文件和控制台这两个地方都有相关日志记录。基于此，我们将propagate的值设置为0，这样就可以避免日志记录向上层的日志器进行传递。

配置文件除了上述的.conf格式文件以外，我们还可以配置字典形式以及yaml形式的配置文件，接下来我们来看看yaml形式的配置文件该怎么做，个人比较喜欢这种模式：

**.yaml形式：**

**假设我配置的logtest.yaml的文件的内容如下：**

version: 1  
disable\_existing\_loggers: False  
*# formatters配置了日志输出时的样式  
# formatters定义了一组formatID,有不同的格式;*formatters:  
 brief:  
 format: **"%(asctime)s - %(message)s"**  
 simple:  
 format: **"%(asctime)s - [%(name)s] - [%(levelname)s] :%(levelno)s: %(message)s"**  
 datefmt: **'%F %T'**  
*# handlers配置了需要处理的日志信息,logging模块的handler只有streamhandler和filehandler*handlers:  
 console:  
 class : logging.StreamHandler  
 formatter: brief  
 level : DEBUG  
 stream : ext://sys.stdout  
 info\_file\_handler:  
 class : logging.FileHandler  
 formatter: simple  
 level: ERROR  
 filename: ./mylog/debug\_test.log  
 error\_file\_handler:  
 class: logging.handlers.RotatingFileHandler  
 level: ERROR  
 formatter: simple  
 filename: ./mylog/errors.log  
 maxBytes: 10485760 *# 10MB #1024\*1024\*10* backupCount: 50  
 encoding: utf8  
  
loggers:  
*# fileLogger, 就是在代码中通过logger = logging.getLogger("fileLogger")来获得该类型的logger* my\_testyaml:  
 level: DEBUG  
 handlers: [console, info\_file\_handler,error\_file\_handler]  
*# root为默认情况下的输出配置, 当logging.getLogger("fileLoggername")里面的fileLoggername没有传值的时候,  
# 就是用的这个默认的root,如logging.getLogger(\_\_name\_\_)或logging.getLogger()*root:  
 level: DEBUG  
 handlers: [console]

disable\_existing\_loggers：这是一个布尔型值的参数，默认值为True（为了向后兼容）表示禁用已经存在的logger；如果值为False则对已存在的loggers保持启动状态，这里设置为False。

那么使用这个logtest.yaml的代码的示例如下：

import logging  
import os  
from logging import config  
import yaml  
  
def use\_yaml\_config(default\_path=**'.\mylog\config.yaml'**, default\_level=logging.INFO):  
 path = default\_path  
 if os.path.exists(path):  
 with open(path, **'r'**, encoding = **'utf-8'**) as f:  
 config = yaml.load(stream=f, Loader=yaml.FullLoader)  
 logging.config.dictConfig(config)  
 else:  
 logging.basicConfig(level=default\_level)  
  
if **'\_\_main\_\_'** == \_\_name\_\_:  
   
 use\_yaml\_config(default\_path=**'.\mylog\logtest.yaml'**)  
 *# root的logger* root = logging.getLogger()  
 *# 子记录器的名字与配置文件中loggers字段内的保持一致  
 # loggers:  
 # my\_testyaml:  
 # level: DEBUG  
 # handlers: [console, info\_file\_handler,error\_file\_handler]* my\_testyaml = logging.getLogger(**"my\_testyaml"**)  
 print(**"rootlogger:"**, root.handlers)  
 print(**"selflogger"**, my\_testyaml.handlers)  
 *# 判断子记录器与根记录器的handler是否相同* print(root.handlers[0] == my\_testyaml.handlers[0])  
  
 my\_testyaml.info(**"INFO"**)  
 my\_testyaml.error(**'ERROR'**)  
 my\_testyaml.debug(**"rootDEBUG"**)  
  
 root.info(**"INFO"**)  
 root.error(**'ERROR'**)  
 root.debug(**"rootDEBUG"**)

大家可以试着自己运行一下，看看最后的输出是怎样的。

1. **自定义logger日志输出文件**

以上介绍的知识点，大家在平时或在实际的项目中使用基本上就足够了，下面我们简单的介绍一下怎么进行自定义logger进行日志的输出。

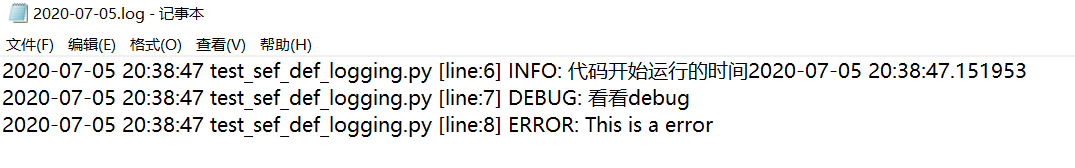
我们写一个sel\_def\_logger.py文件，文件中的定义一个类，内容如下：

import os  
import sys  
import logging  
from time import strftime  
*# 输出日志路径*PATH = os.path.abspath(**'.'**) + **'/logs/'***# 设置日志格式#和时间格式*FMT = **'%(asctime)s %(filename)s [line:%(lineno)d] %(levelname)s: %(message)s'**DATEFMT = **'%Y-%m-%d %H:%M:%S'**class MyLog(object):  
 def \_\_init\_\_(self):  
 self.logger = logging.getLogger()  
 self.formatter = logging.Formatter(fmt=FMT, datefmt=DATEFMT)  
 self.log\_filename = **'{0}{1}.log'**.format(PATH, strftime(**"%Y-%m-%d"**))  
  
 self.logger.addHandler(self.get\_file\_handler(self.log\_filename))  
 self.logger.addHandler(self.get\_console\_handler())  
 *# 设置日志的默认级别* self.logger.setLevel(logging.DEBUG)  
  
 *# 输出到文件handler的函数定义* def get\_file\_handler(self, filename):  
 filehandler = logging.FileHandler(filename, encoding=**"utf-8"**)  
 filehandler.setFormatter(self.formatter)  
 return filehandler  
  
 *# 输出到控制台handler的函数定义* def get\_console\_handler(self):  
 console\_handler = logging.StreamHandler(sys.stdout)  
 console\_handler.setFormatter(self.formatter)  
 return console\_handler

那么使用这个类的方法可以这样写：

import datetime  
from sef\_def\_logger import MyLog  
my\_logg = MyLog().logger  
my\_logg.info(**"代码开始运行的时间{}"**.format(datetime.datetime.now()))  
my\_logg.debug(**'看看debug'**)  
my\_logg.error(**'This is a error'**)

生成的结果在我们预先生成的文件夹logs下，有类似的时间命名的日志文件如：2020-07-05.log，日志内容如下：



到此python的日志模块logging就介绍结束了，内容还是非常多的，大家得慢慢学习慢慢体会，最重要的还是要在实际的项目中进行实战。

最后大家可以参考官方文档的知识点进行查缺补漏：

<https://docs.python.org/3.7/howto/logging-cookbook.html#logging-cookbook>

好了，我们下期再见！