Python常见的内置模块详解

# os模块:提供对操作系统进行调用的接口

os.getcwd() 获取当前工作目录，即当前python脚本工作的目录路径

os.chdir("dirname") 改变当前脚本工作目录；相当于shell下cdos.curdir 返回当前目录: ('.')

os.pardir 获取当前目录的父目录字符串名：('..')

os.makedirs('dirname1/dirname2') 可生成多层递归目录

os.removedirs('dirname1') 若目录为空，则删除，并递归到上一级目录，如若也为空，则删除，依此类推

os.mkdir('dirname') 生成单级目录；相当于shell中mkdir dirname

os.rmdir('dirname') 删除单级空目录，若目录不为空则无法删除，报错；相当于shell中rmdir dirname

os.listdir('dirname') 列出指定目录下的所有文件和子目录，包括隐藏文件，并以列表方式打印

os.remove() 删除一个文件

os.rename("oldname","newname") 重命名文件/目录

os.stat('path/filename') 获取文件/目录信息

os.sep 输出操作系统特定的路径分隔符，win下为"\\",Linux下为"/"

os.linesep 输出当前平台使用的行终止符，win下为"\t\n",Linux下为"\n"

os.pathsep 输出用于分割文件路径的字符串

os.name 输出字符串指示当前使用平台。win->'nt'; Linux->'posix'

os.system("bash command") 运行shell命令，直接显示

os.environ 获取系统环境变量

os.path.abspath(path) 返回path规范化的绝对路径

os.path.split(path) 将path分割成目录和文件名二元组返回

os.path.dirname(path) 返回path的目录。其实就是os.path.split(path)的第一个元素

os.path.basename(path) 返回path最后的文件名。如何path以／或\结尾，那么就会返回空值。即os.path.split(path)的第二个元素

os.path.exists(path) 如果path存在，返回True；如果path不存在，返回False

os.path.isabs(path) 如果path是绝对路径，返回True

os.path.isfile(path) 如果path是一个存在的文件，返回True。否则返回False

os.path.isdir(path) 如果path是一个存在的目录，则返回True。否则返回False

os.path.join(path1[, path2[, ...]]) 将多个路径组合后返回，第一个绝对路径之前的参数将被忽略

os.path.getatime(path) 返回path所指向的文件或者目录的最后存取时间

os.path.getmtime(path) 返回path所指向的文件或者目录的最后修改时间

1.2 sys:用于提供对解释器相关的操作

sys.argv 命令行参数List，第一个元素是程序本身路径

sys.exit(n) 退出程序，正常退出时exit(0)

sys.version 获取Python解释程序的版本信息

sys.maxint 最大的Int值

sys.path 返回模块的搜索路径，初始化时使用PYTHONPATH环境变量的值

sys.platform 返回操作系统平台名称

sys.stdout.write('please:')

val = sys.stdin.readline()[:-1]

# 1.3 hashlib

用于加密相关的操作，代替了md5模块和sha模块，主要提供 SHA1, SHA224, SHA256, SHA384, SHA512 ，MD5 算法

import md5

hash = md5.new()

hash.update('admin')

print hash.hexdigest()

import sha

hash = sha.new()

hash.update('admin')

print hash.hexdigest()

|  |
| --- |
| hablib加密的实例:  import hashlib  # ######## md5 ########  hash = hashlib.md5()  hash.update('admin')  print hash.hexdigest()  # ######## sha1 ########  hash = hashlib.sha1()  hash.update('admin')  print hash.hexdigest()  # ######## sha256 ########  hash = hashlib.sha256()  hash.update('admin')  print hash.hexdigest()  # ######## sha384 ########  hash = hashlib.sha384()  hash.update('admin')  print hash.hexdigest()  # ######## sha512 ########  hash = hashlib.sha512()  hash.update('admin')  print hash.hexdigest() |

以上加密算法虽然依然非常厉害，但时候存在缺陷，即：通过撞库可以反解。所以，有必要对加密算法中添加自定义key再来做加密。

|  |
| --- |
| import hashlib  # ######## md5 ########  hash = hashlib.md5('898oaFs09f')  hash.update('admin')  print hash.hexdigest()  还不够吊？python 还有一个 hmac 模块，它内部对我们创建 key 和 内容 再进行处理然后再加密  import hmac  h = hmac.new('wueiqi')  h.update('hellowo')  print h.hexdigest() |

# 1.4 logging:用于便捷记录日志且线程安全的模块

# 1.4.1 简单应用

|  |
| --- |
| import logging  logging.debug('debug message')  logging.info('info message')  logging.warning('warning message')  logging.error('error message')  logging.critical('critical message') |

输出：

WARNING:root:warning message

ERROR:root:error message

CRITICAL:root:critical message

可见，默认情况下[Python](http://lib.csdn.net/base/11" \o "undefined" \t "https://www.cnblogs.com/yuanchenqi/articles/_blank)的logging模块将日志打印到了标准输出中，且只显示了大于等于WARNING级别的日志，这说明默认的日志级别设置为WARNING（日志级别等级CRITICAL > ERROR > WARNING > INFO > DEBUG > NOTSET），默认的日志格式为日志级别：Logger名称：用户输出消息。

## 1.4.2 灵活配置日志级别，日志格式，输出位置

|  |
| --- |
| import logging  logging.basicConfig(level=logging.DEBUG,  format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message)s',  datefmt='%a, %d %b %Y %H:%M:%S',  filename='/tmp/test.log',  filemode='w')    logging.debug('debug message')  logging.info('info message')  logging.warning('warning message')  logging.error('error message')  logging.critical('critical message') |

查看输出：  
cat /tmp/test.log   
Mon, 05 May 2014 16:29:53 test\_logging.py[line:9] DEBUG debug message  
Mon, 05 May 2014 16:29:53 test\_logging.py[line:10] INFO info message  
Mon, 05 May 2014 16:29:53 test\_logging.py[line:11] WARNING warning message  
Mon, 05 May 2014 16:29:53 test\_logging.py[line:12] ERROR error message  
Mon, 05 May 2014 16:29:53 test\_logging.py[line:13] CRITICAL critical message

可见在logging.basicConfig()函数中可通过具体参数来更改logging模块默认行为，可用参数有

filename：用指定的文件名创建FiledHandler（后边会具体讲解handler的概念），这样日志会被存储在指定的文件中。  
filemode：文件打开方式，在指定了filename时使用这个参数，默认值为“a”还可指定为“w”。  
format：指定handler使用的日志显示格式。   
datefmt：指定日期时间格式。   
level：设置rootlogger（后边会讲解具体概念）的日志级别   
stream：用指定的stream创建StreamHandler。可以指定输出到sys.stderr,sys.stdout或者文件(f=open('test.log','w'))，默认为sys.stderr。若同时列出了filename和stream两个参数，则stream参数会被忽略。

format参数中可能用到的格式化串：

|  |
| --- |
| %(name)s :Logger的名字  %(levelno)s :数字形式的日志级别  %(levelname)s :文本形式的日志级别  %(pathname)s :调用日志输出函数的模块的完整路径名，可能没有  %(filename)s :调用日志输出函数的模块的文件名  %(module)s :调用日志输出函数的模块名  %(funcName)s :调用日志输出函数的函数名  %(lineno)d :调用日志输出函数的语句所在的代码行  %(created)f :当前时间，用UNIX标准的表示时间的浮 点数表示  %(relativeCreated)d :输出日志信息时的，自Logger创建以 来的毫秒数  %(asctime)s :字符串形式的当前时间。默认格式是 “2003-07-08 16:49:45,896”。逗号后面的是毫秒  %(thread)d :线程ID。可能没有  %(threadName)s :线程名。可能没有  %(process)d :进程ID。可能没有  %(message)s :用户输出的消息 |

## 1.4.3 logger对象

上述几个例子中我们了解到了logging.debug()、logging.info()、logging.warning()、logging.error()、logging.critical()（分别用以记录不同级别的日志信息），logging.basicConfig()（用默认日志格式（Formatter）为日志系统建立一个默认的流处理器（StreamHandler），设置基础配置（如日志级别等）并加到root logger（根Logger）中）这几个logging模块级别的函数，另外还有一个模块级别的函数是logging.getLogger([name])（返回一个logger对象，如果没有指定名字将返回root logger）

|  |
| --- |
| 先看一个最简单的过程：  import logging  logger = logging.getLogger()# 创建一个handler，用于写入日志文件  fh = logging.FileHandler('test.log')  # 再创建一个handler，用于输出到控制台  ch = logging.StreamHandler()  formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')  fh.setFormatter(formatter)  ch.setFormatter(formatter)  logger.addHandler(fh) #logger对象可以添加多个fh和ch对象  logger.addHandler(ch)  logger.debug('logger debug message')  logger.info('logger info message')  logger.warning('logger warning message')  logger.error('logger error message')  logger.critical('logger critical message') |

先简单介绍一下，logging库提供了多个组件：Logger、Handler、Filter、Formatter。Logger对象提供应用程序可直接使用的接口，Handler发送日志到适当的目的地，Filter提供了过滤日志信息的方法，Formatter指定日志显示格式。

(1)Logger是一个树形层级结构，输出信息之前都要获得一个Logger（如果没有显示的获取则自动创建并使用root Logger，如第一个例子所示）。

logger = logging.getLogger()返回一个默认的Logger也即root Logger，并应用默认的日志级别、Handler和Formatter设置。

当然也可以通过Logger.setLevel(lel)指定最低的日志级别，可用的日志级别有logging.DEBUG、logging.INFO、 logging.WARNING、logging.ERROR、logging.CRITICAL。

Logger.debug()、Logger.info()、Logger.warning()、Logger.error()、Logger.critical()输出不同级别的日志，只有日志等级大于或等于设置的日志级别的日志才会被输出。

|  |
| --- |
| logger.debug('logger debug message')  logger.info('logger info message')  logger.warning('logger warning message')  logger.error('logger error message')  logger.critical('logger critical message') |

只输出了

2014-05-06 12:54:43,222 - root - WARNING - logger warning message

2014-05-06 12:54:43,223 - root - ERROR - logger error message

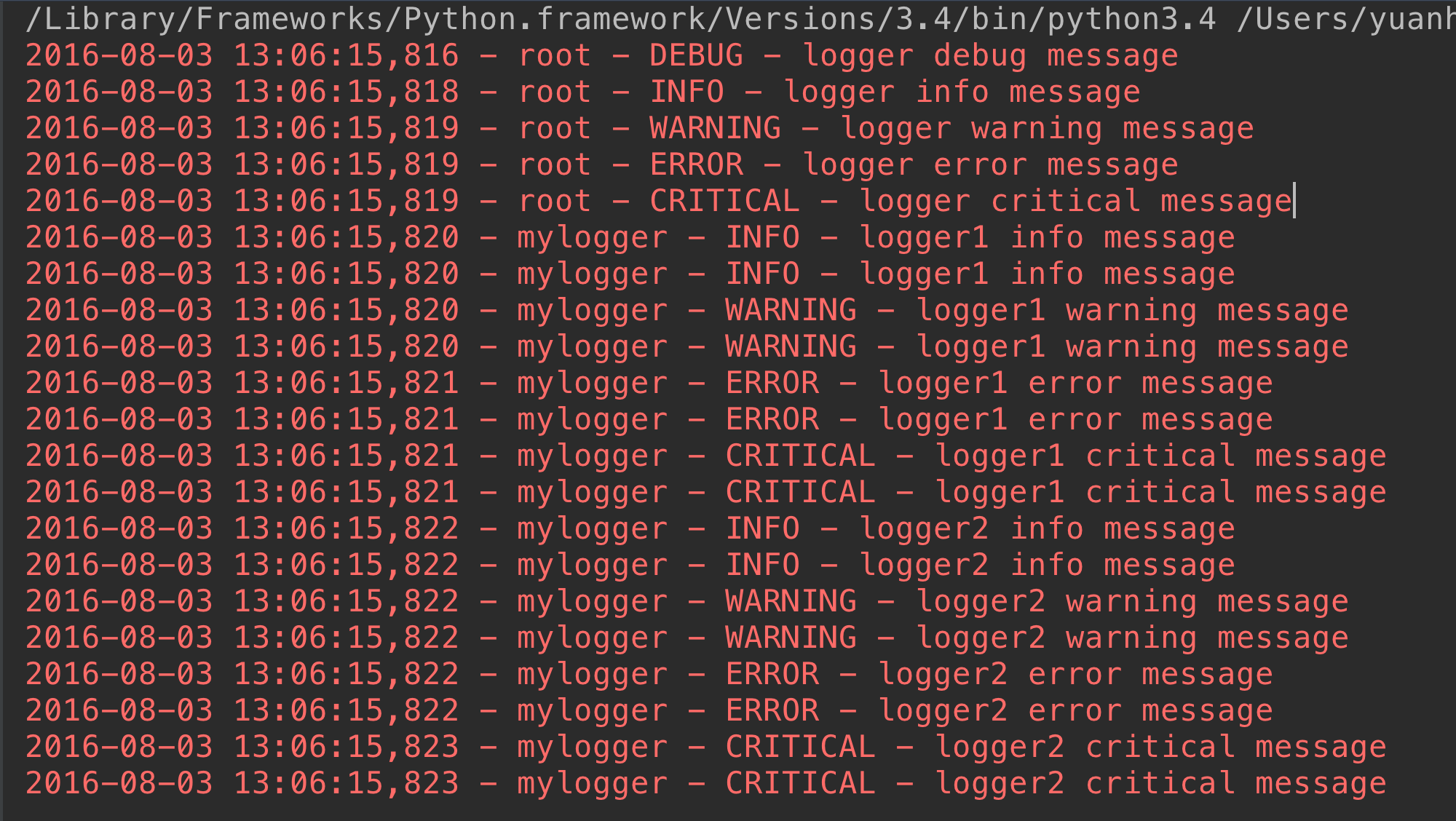
2014-05-06 12:54:43,224 - root - CRITICAL - logger critical message

从这个输出可以看出logger = logging.getLogger()返回的Logger名为root。这里没有用logger.setLevel(logging.Debug)显示的为logger设置日志级别，所以使用默认的日志级别WARNIING，故结果只输出了大于等于WARNIING级别的信息。

1. 如果我们再创建两个logger对象：

|  |
| --- |
| ##################################################  logger1 = logging.getLogger('mylogger')  logger1.setLevel(logging.DEBUG)  logger2 = logging.getLogger('mylogger')  logger2.setLevel(logging.INFO)  logger1.addHandler(fh)  logger1.addHandler(ch)  logger2.addHandler(fh)  logger2.addHandler(ch)  logger1.debug('logger1 debug message')  logger1.info('logger1 info message')  logger1.warning('logger1 warning message')  logger1.error('logger1 error message')  logger1.critical('logger1 critical message')    logger2.debug('logger2 debug message')  logger2.info('logger2 info message')  logger2.warning('logger2 warning message')  logger2.error('logger2 error message')  logger2.critical('logger2 critical message') |

结果：



这里有两个个问题：

<1>我们明明通过logger1.setLevel(logging.DEBUG)将logger1的日志级别设置为了DEBUG，为何显示的时候没有显示出DEBUG级别的日志信息，而是从INFO级别的日志开始显示呢？

原来logger1和logger2对应的是同一个Logger实例，只要logging.getLogger（name）中名称参数name相同则返回的Logger实例就是同一个，且仅有一个，也即name与Logger实例一一对应。在logger2实例中通过logger2.setLevel(logging.INFO)设置mylogger的日志级别为logging.INFO，所以最后logger1的输出遵从了后来设置的日志级别。

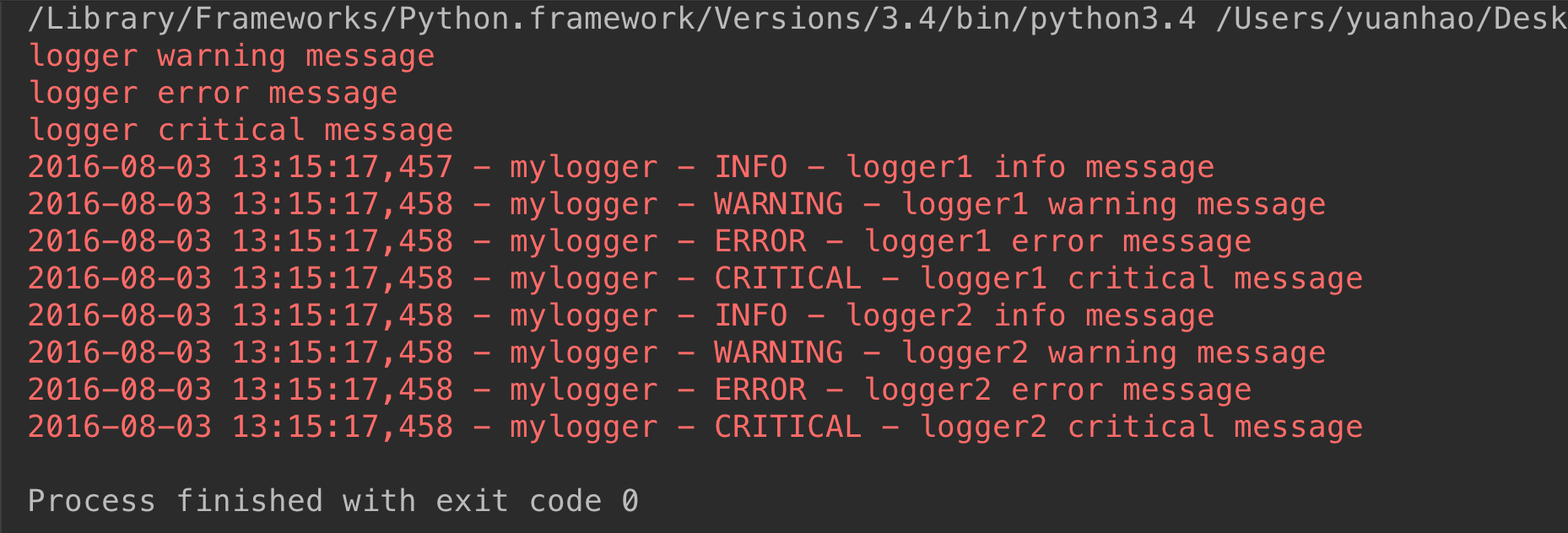
<2>为什么logger1、logger2对应的每个输出分别显示两次?

这是因为我们通过logger = logging.getLogger()显示的创建了root Logger，而logger1 = logging.getLogger('mylogger')创建了root Logger的孩子(root.)mylogger,logger2同样。而孩子,孙子，重孙……既会将消息分发给他的handler进行处理也会传递给所有的祖先Logger处理。

ok,那么现在我们把

# logger.addHandler(fh)

# logger.addHandler(ch) 注释掉，我们再来看效果：



因为我们注释了logger对象显示的位置，所以才用了默认方式，即标准输出方式。因为它的父级没有设置文件显示方式，所以在这里只打印了一次。

孩子,孙子，重孙……可逐层继承来自祖先的日志级别、Handler、Filter设置，也可以通过Logger.setLevel(lel)、Logger.addHandler(hdlr)、Logger.removeHandler(hdlr)、Logger.addFilter(filt)、Logger.removeFilter(filt)。设置自己特别的日志级别、Handler、Filter。若不设置则使用继承来的值。

## 1.4.4 FilterFilter

限制只有满足过滤规则的日志才会输出。

比如我们定义了filter = logging.Filter('a.b.c'),并将这个Filter添加到了一个Handler上，则使用该Handler的Logger中只有名字带a.b.c前缀的Logger才能输出其日志。

filter = logging.Filter('mylogger')

logger.addFilter(filter)

这是只对logger这个对象进行筛选

如果想对所有的对象进行筛选，则：

filter = logging.Filter('mylogger')

fh.addFilter(filter)

ch.addFilter(filter)

这样，所有添加fh或者ch的logger对象都会进行筛选。

完整代码1：

|  |
| --- |
| import logging  logger = logging.getLogger()  # 创建一个handler，用于写入日志文件  fh = logging.FileHandler('test.log')  # 再创建一个handler，用于输出到控制台  ch = logging.StreamHandler()  formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')  fh.setFormatter(formatter)  ch.setFormatter(formatter)  # 定义一个filter  filter = logging.Filter('mylogger')  fh.addFilter(filter)  ch.addFilter(filter)  # logger.addFilter(filter)  logger.addHandler(fh)  logger.addHandler(ch)  logger.setLevel(logging.DEBUG)  logger.debug('logger debug message')  logger.info('logger info message')  logger.warning('logger warning message')  logger.error('logger error message')  logger.critical('logger critical message')  ##################################################  logger1 = logging.getLogger('mylogger')  logger1.setLevel(logging.DEBUG)  logger2 = logging.getLogger('mylogger')  logger2.setLevel(logging.INFO)  logger1.addHandler(fh)  logger1.addHandler(ch)  logger2.addHandler(fh)  logger2.addHandler(ch)  logger1.debug('logger1 debug message')  logger1.info('logger1 info message')  logger1.warning('logger1 warning message')  logger1.error('logger1 error message')  logger1.critical('logger1 critical message')  logger2.debug('logger2 debug message')  logger2.info('logger2 info message')  logger2.warning('logger2 warning message')  logger2.error('logger2 error message')  logger2.critical('logger2 critical message') |

完整代码2

|  |
| --- |
| #coding:utf-8  import logging    # 创建一个logger  logger = logging.getLogger()    logger1 = logging.getLogger('mylogger')  logger1.setLevel(logging.DEBUG)    logger2 = logging.getLogger('mylogger')  logger2.setLevel(logging.INFO)    logger3 = logging.getLogger('mylogger.child1')  logger3.setLevel(logging.WARNING)    logger4 = logging.getLogger('mylogger.child1.child2')  logger4.setLevel(logging.DEBUG)    logger5 = logging.getLogger('mylogger.child1.child2.child3')  logger5.setLevel(logging.DEBUG)    # 创建一个handler，用于写入日志文件  fh = logging.FileHandler('/tmp/test.log')    # 再创建一个handler，用于输出到控制台  ch = logging.StreamHandler()    # 定义handler的输出格式formatter  formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')  fh.setFormatter(formatter)  ch.setFormatter(formatter)    #定义一个filter  #filter = logging.Filter('mylogger.child1.child2')  #fh.addFilter(filter)    # 给logger添加handler  #logger.addFilter(filter)  logger.addHandler(fh)  logger.addHandler(ch)    #logger1.addFilter(filter)  logger1.addHandler(fh)  logger1.addHandler(ch)    logger2.addHandler(fh)  logger2.addHandler(ch)    #logger3.addFilter(filter)  logger3.addHandler(fh)  logger3.addHandler(ch)    #logger4.addFilter(filter)  logger4.addHandler(fh)  logger4.addHandler(ch)    logger5.addHandler(fh)  logger5.addHandler(ch)    # 记录一条日志  logger.debug('logger debug message')  logger.info('logger info message')  logger.warning('logger warning message')  logger.error('logger error message')  logger.critical('logger critical message')    logger1.debug('logger1 debug message')  logger1.info('logger1 info message')  logger1.warning('logger1 warning message')  logger1.error('logger1 error message')  logger1.critical('logger1 critical message')    logger2.debug('logger2 debug message')  logger2.info('logger2 info message')  logger2.warning('logger2 warning message')  logger2.error('logger2 error message')  logger2.critical('logger2 critical message')    logger3.debug('logger3 debug message')  logger3.info('logger3 info message')  logger3.warning('logger3 warning message')  logger3.error('logger3 error message')  logger3.critical('logger3 critical message')    logger4.debug('logger4 debug message')  logger4.info('logger4 info message')  logger4.warning('logger4 warning message')  logger4.error('logger4 error message')  logger4.critical('logger4 critical message')    logger5.debug('logger5 debug message')  logger5.info('logger5 info message')  logger5.warning('logger5 warning message')  logger5.error('logger5 error message')  logger5.critical('logger5 critical message') |

应用代码

|  |
| --- |
| import os  import time  import logging  from config import settings  def get\_logger(card\_num, struct\_time):  if struct\_time.tm\_mday < 23:  file\_name = "%s\_%s\_%d" %(struct\_time.tm\_year, struct\_time.tm\_mon, 22)  else:  file\_name = "%s\_%s\_%d" %(struct\_time.tm\_year, struct\_time.tm\_mon+1, 22)  file\_handler = logging.FileHandler(  os.path.join(settings.USER\_DIR\_FOLDER, card\_num, 'record', file\_name),encoding='utf-8')  fmt = logging.Formatter(fmt="%(asctime)s : %(message)s")  file\_handler.setFormatter(fmt)  logger1 = logging.Logger('user\_logger', level=logging.INFO)  logger1.addHandler(file\_handler)  return logger1 |

re模块(\* \* \* \* \*)

就其本质而言，正则表达式（或 RE）是一种小型的、高度专业化的编程语言，（在Python中）它内嵌在Python中，并通过 re 模块实现。正则表达式模式被编译成一系列的字节码，然后由用 C 编写的匹配引擎执行。

字符匹配（普通字符，元字符）：

1 普通字符：大多数字符和字母都会和自身匹配

>>> re.findall('alvin','yuanaleSxalexwupeiqi')

['alvin']

2 元字符：. ^ $ \* + ? { } [ ] | ( ) \

元字符之. ^ $ \* + ? { }

|  |
| --- |
| import re  ret=re.findall('a..in','helloalvin')  print(ret)#['alvin']  ret=re.findall('^a...n','alvinhelloawwwn')  print(ret)#['alvin']  ret=re.findall('a...n$','alvinhelloawwwn')  print(ret)#['awwwn']  ret=re.findall('a...n$','alvinhelloawwwn')  print(ret)#['awwwn']  ret=re.findall('abc\*','abcccc')#贪婪匹配[0,+oo]  print(ret)#['abcccc']  ret=re.findall('abc+','abccc')#[1,+oo]  print(ret)#['abccc']  ret=re.findall('abc?','abccc')#[0,1]  print(ret)#['abc']  ret=re.findall('abc{1,4}','abccc')  print(ret)#['abccc'] 贪婪匹配 |

注意：前面的\*,+,?等都是贪婪匹配，也就是尽可能匹配，后面加?号使其变成惰性匹配

|  |
| --- |
| ret=re.findall('abc\*?','abcccccc')  print(ret)#['ab'] |