Python中的logging模块

2016/11/27 · [基础知识](http://python.jobbole.com/category/basic/) · [logging](http://python.jobbole.com/tag/logging/)

分享到：

原文出处： [cicaday](https://segmentfault.com/a/1190000007581128)

最近修改了项目里的logging相关功能，用到了python标准库里的logging模块，在此做一些记录。主要是从官方文档和stackoverflow上查询到的一些内容。

* [官方文档](https://docs.python.org/2.7/library/logging.html)
* [技术博客](http://blog.csdn.net/balderfan/article/details/7644807)

**基本用法**

下面的代码展示了logging最基本的用法。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | # -\*- coding: utf-8 -\*-    import logging  import sys    # 获取logger实例，如果参数为空则返回root logger  logger = logging.getLogger("AppName")    # 指定logger输出格式  formatter = logging.Formatter('%(asctime)s %(levelname)-8s: %(message)s')    # 文件日志  file\_handler = logging.FileHandler("test.log")  file\_handler.setFormatter(formatter)  # 可以通过setFormatter指定输出格式    # 控制台日志  console\_handler = logging.StreamHandler(sys.stdout)  console\_handler.formatter = formatter  # 也可以直接给formatter赋值    # 为logger添加的日志处理器  logger.addHandler(file\_handler)  logger.addHandler(console\_handler)    # 指定日志的最低输出级别，默认为WARN级别  logger.setLevel(logging.INFO)    # 输出不同级别的log  logger.debug('this is debug info')  logger.info('this is information')  logger.warn('this is warning message')  logger.error('this is error message')  logger.fatal('this is fatal message, it is same as logger.critical')  logger.critical('this is critical message')    # 2016-10-08 21:59:19,493 INFO    : this is information  # 2016-10-08 21:59:19,493 WARNING : this is warning message  # 2016-10-08 21:59:19,493 ERROR   : this is error message  # 2016-10-08 21:59:19,493 CRITICAL: this is fatal message, it is same as logger.critical  # 2016-10-08 21:59:19,493 CRITICAL: this is critical message    # 移除一些日志处理器  logger.removeHandler(file\_handler) |

除了这些基本用法，还有一些常见的小技巧可以分享一下。

**格式化输出日志**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | # 格式化输出    service\_name = "Booking"  logger.error('%s service is down!' % service\_name)  # 使用python自带的字符串格式化，不推荐  logger.error('%s service is down!', service\_name)  # 使用logger的格式化，推荐  logger.error('%s service is %s!', service\_name, 'down')  # 多参数格式化  logger.error('{} service is {}'.format(service\_name, 'down')) # 使用format函数，推荐    # 2016-10-08 21:59:19,493 ERROR   : Booking service is down! |

**记录异常信息**

当你使用logging模块记录异常信息时，不需要传入该异常对象，只要你直接调用logger.error() 或者 logger.exception()就可以将当前异常记录下来。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | # 记录异常信息    try:      1 / 0  except:      # 等同于error级别，但是会额外记录当前抛出的异常堆栈信息      logger.exception('this is an exception message')    # 2016-10-08 21:59:19,493 ERROR   : this is an exception message  # Traceback (most recent call last):  #   File "D:/Git/py\_labs/demo/use\_logging.py", line 45, in  #     1 / 0  # ZeroDivisionError: integer division or modulo by zero |

**logging配置要点**

**GetLogger()方法**

这是最基本的入口，该方法参数可以为空，默认的logger名称是root，如果在同一个程序中一直都使用同名的logger，其实会拿到同一个实例，使用这个技巧就可以跨模块调用同样的logger来记录日志。

另外你也可以通过日志名称来区分同一程序的不同模块，比如这个例子。



|  |  |
| --- | --- |
| 1  2 | logger = logging.getLogger("App.UI")  logger = logging.getLogger("App.Service") |

**Formatter日志格式**

Formatter对象定义了log信息的结构和内容，构造时需要带两个参数：

* 一个是格式化的模板fmt，默认会包含最基本的level和 message信息
* 一个是格式化的时间样式datefmt，默认为 2003-07-08 16:49:45,896 (%Y-%m-%d %H:%M:%S)

fmt中允许使用的变量可以参考下表。

* **%(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

**SetLevel 日志级别**

Logging有如下级别: DEBUG，INFO，WARNING，ERROR，CRITICAL  
默认级别是WARNING，logging模块只会输出指定level以上的log。这样的好处, 就是在项目开发时debug用的log，在产品release阶段不用一一注释，只需要调整logger的级别就可以了，很方便。

**Handler 日志处理器**

最常用的是StreamHandler和FileHandler, Handler用于向不同的输出端打log。  
Logging包含很多handler, 可能用到的有下面几种

* **StreamHandler** instances send error messages to streams (file-like objects).
* **FileHandler** instances send error messages to disk files.
* **RotatingFileHandler** instances send error messages to disk files, with support for maximum log file sizes and log file rotation.
* **TimedRotatingFileHandler** instances send error messages to disk files, rotating the log file at certain timed intervals.
* **SocketHandler** instances send error messages to TCP/IP sockets.
* **DatagramHandler** instances send error messages to UDP sockets.
* **SMTPHandler** instances send error messages to a designated email address.

**Configuration 配置方法**

logging的配置大致有下面几种方式。

1. 通过代码进行完整配置，参考开头的例子，主要是通过getLogger方法实现。
2. 通过代码进行简单配置，下面有例子，主要是通过basicConfig方法实现。
3. 通过配置文件，下面有例子，主要是通过 logging.config.fileConfig(filepath)

**logging.basicConfig**

basicConfig()提供了非常便捷的方式让你配置logging模块并马上开始使用，可以参考下面的例子。具体可以配置的项目请查阅[官方文档](https://docs.python.org/2/library/logging.html#logging.basicConfig)。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | import logging    logging.basicConfig(filename='example.log',level=logging.DEBUG)  logging.debug('This message should go to the log file')    logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.DEBUG)  logging.debug('This message should appear on the console')    logging.basicConfig(format='%(asctime)s %(message)s', datefmt='%m/%d/%Y %I:%M:%S %p')  logging.warning('is when this event was logged.') |

备注： 其实你甚至可以什么都不配置直接使用默认值在控制台中打log，用这样的方式替换print语句对日后项目维护会有很大帮助。

**通过文件配置logging**

如果你希望通过配置文件来管理logging，可以参考这个[官方文档](https://docs.python.org/2/library/logging.config.html)。在log4net或者log4j中这是很常见的方式。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | # logging.conf  [loggers]  keys=root    [logger\_root]  level=DEBUG  handlers=consoleHandler  #,timedRotateFileHandler,errorTimedRotateFileHandler    #################################################  [handlers]  keys=consoleHandler,timedRotateFileHandler,errorTimedRotateFileHandler    [handler\_consoleHandler]  class=StreamHandler  level=DEBUG  formatter=simpleFormatter  args=(sys.stdout,)    [handler\_timedRotateFileHandler]  class=handlers.TimedRotatingFileHandler  level=DEBUG  formatter=simpleFormatter  args=('debug.log', 'H')    [handler\_errorTimedRotateFileHandler]  class=handlers.TimedRotatingFileHandler  level=WARN  formatter=simpleFormatter  args=('error.log', 'H')    #################################################  [formatters]  keys=simpleFormatter, multiLineFormatter    [formatter\_simpleFormatter]  format= %(levelname)s %(threadName)s %(asctime)s:   %(message)s  datefmt=%H:%M:%S    [formatter\_multiLineFormatter]  format= ------------------------- %(levelname)s -------------------------  Time:      %(asctime)s  Thread:    %(threadName)s  File:      %(filename)s(line %(lineno)d)  Message:  %(message)s    datefmt=%Y-%m-%d %H:%M:%S |

假设以上的配置文件放在和模块相同的目录，代码中的调用如下。



|  |  |
| --- | --- |
| 1  2  3  4 | import os  filepath = os.path.join(os.path.dirname(\_\_file\_\_), 'logging.conf')  logging.config.fileConfig(filepath)  return logging.getLogger() |

**日志重复输出的坑**

你有可能会看到你打的日志会重复显示多次，可能的原因有很多，但总结下来无非就一个，日志中使用了重复的handler。

**第一坑**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | import logging    logging.basicConfig(level=logging.DEBUG)    fmt = '%(levelname)s:%(message)s'  console\_handler = logging.StreamHandler()  console\_handler.setFormatter(logging.Formatter(fmt))  logging.getLogger().addHandler(console\_handler)    logging.info('hello!')    # INFO:root:hello!  # INFO:hello! |

上面这个例子出现了重复日志，因为在第3行调用basicConfig()方法时系统会默认创建一个handler，如果你再添加一个控制台handler时就会出现重复日志。

**第二坑**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | import logging    def get\_logger():      fmt = '%(levelname)s:%(message)s'      console\_handler = logging.StreamHandler()      console\_handler.setFormatter(logging.Formatter(fmt))      logger = logging.getLogger('App')      logger.setLevel(logging.INFO)      logger.addHandler(console\_handler)      return logger    def call\_me():      logger = get\_logger()      logger.info('hi')    call\_me()  call\_me()    # INFO:hi  # INFO:hi  # INFO:hi |

在这个例子里hi居然打印了三次，如果再调用一次call\_me()呢？我告诉你会打印6次。why? 因为你每次调用get\_logger()方法时都会给它加一个新的handler，你是自作自受。正常的做法应该是全局只配置logger一次。

**第三坑**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | import logging    def get\_logger():      fmt = '%(levelname)s: %(message)s'      console\_handler = logging.StreamHandler()      console\_handler.setFormatter(logging.Formatter(fmt))      logger = logging.getLogger('App')      logger.setLevel(logging.INFO)      logger.addHandler(console\_handler)      return logger    def foo():      logging.basicConfig(format='[%(name)s]: %(message)s')      logging.warn('some module use root logger')    def main():      logger = get\_logger()      logger.info('App start.')      foo()      logger.info('App shutdown.')    main()    # INFO: App start.  # [root]: some module use root logger  # INFO: App shutdown.  # [App]: App shutdown. |

为嘛最后的App shutdown打印了两次？所以在Stackoverflow上很多人都问，我应该怎么样把root logger关掉，root logger太坑爹坑妈了。只要你在程序中使用过root logger，那么默认你打印的所有日志都算它一份。上面的例子没有什么很好的办法，我建议你招到那个没有经过大脑就使用root logger的人，乱棍打死他或者开除他。

如果你真的想禁用root logger，有两个不是办法的办法：



|  |  |
| --- | --- |
| 1  2 | logging.getLogger().handlers = []  # 删除所有的handler  logging.getLogger().setLevel(logging.CRITICAL)  # 将它的级别设置到最高 |

**小结**

Python中的日志模块作为标准库的一部分，功能还是比较完善的。个人觉得上手简单，另外也支持比如过滤，文件锁等高级功能，能满足大多数项目需求。

不过切记，小心坑。

原文：http://www.cnblogs.com/dahu-daqing/p/7040764.html

**1 logging模块简介**

logging模块是Python内置的标准模块，主要用于输出运行日志，可以设置输出日志的等级、日志保存路径、日志文件回滚等；相比print，具备如下优点：

1. 可以通过设置不同的日志等级，在release版本中只输出重要信息，而不必显示大量的调试信息；
2. print将所有信息都输出到标准输出中，严重影响开发者从标准输出中查看其它数据；logging则可以由开发者决定将信息输出到什么地方，以及怎么输出；

**2 logging模块使用**

**2.1 基本使用**

配置logging基本的设置，然后在控制台输出日志，

[复制代码](javascript:void(0);)

复制代码

import logging

logging.basicConfig(level = logging.INFO,format = '%(asctime)s - %(name)s - %(levelname)s - %(message)s')

logger = logging.getLogger(\_\_name\_\_)

logger.info("Start print log")

logger.debug("Do something")

logger.warning("Something maybe fail.")

logger.info("Finish")

复制代码

[复制代码](javascript:void(0);)

运行时，控制台输出，

2016-10-09 19:11:19,434 - \_\_main\_\_ - INFO - Start print log

2016-10-09 19:11:19,434 - \_\_main\_\_ - WARNING - Something maybe fail.

2016-10-09 19:11:19,434 - \_\_main\_\_ - INFO - Finish

logging中可以选择很多消息级别，如debug、info、warning、error以及critical。通过赋予logger或者handler不同的级别，开发者就可以只输出错误信息到特定的记录文件，或者在调试时只记录调试信息。

例如，我们将logger的级别改为DEBUG，再观察一下输出结果，

logging.basicConfig(level = logging.DEBUG,format = '%(asctime)s - %(name)s - %(levelname)s - %(message)s')

控制台输出，可以发现，输出了debug的信息。

2016-10-09 19:12:08,289 - \_\_main\_\_ - INFO - Start print log

2016-10-09 19:12:08,289 - \_\_main\_\_ - DEBUG - Do something

2016-10-09 19:12:08,289 - \_\_main\_\_ - WARNING - Something maybe fail.

2016-10-09 19:12:08,289 - \_\_main\_\_ - INFO - Finish

logging.basicConfig函数各参数：

filename：指定日志文件名；

filemode：和file函数意义相同，指定日志文件的打开模式，'w'或者'a'；

format：指定输出的格式和内容，format可以输出很多有用的信息，

参数：作用

%(levelno)s：打印日志级别的数值

%(levelname)s：打印日志级别的名称

%(pathname)s：打印当前执行程序的路径，其实就是sys.argv[0]

%(filename)s：打印当前执行程序名

%(funcName)s：打印日志的当前函数

%(lineno)d：打印日志的当前行号

%(asctime)s：打印日志的时间

%(thread)d：打印线程ID

%(threadName)s：打印线程名称

%(process)d：打印进程ID

%(message)s：打印日志信息

datefmt：指定时间格式，同time.strftime()；

level：设置日志级别，默认为logging.WARNNING；

stream：指定将日志的输出流，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；

**2.2 将日志写入到文件**

**2.2.1 将日志写入到文件**

设置logging，创建一个FileHandler，并对输出消息的格式进行设置，将其添加到logger，然后将日志写入到指定的文件中，

[复制代码](javascript:void(0);)

复制代码

import logging

logger = logging.getLogger(\_\_name\_\_)

logger.setLevel(level = logging.INFO)

handler = logging.FileHandler("log.txt")

handler.setLevel(logging.INFO)

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

handler.setFormatter(formatter)

logger.addHandler(handler)

logger.info("Start print log")

logger.debug("Do something")

logger.warning("Something maybe fail.")

logger.info("Finish")

复制代码

[复制代码](javascript:void(0);)

log.txt中日志数据为，

2016-10-09 19:01:13,263 - \_\_main\_\_ - INFO - Start print log

2016-10-09 19:01:13,263 - \_\_main\_\_ - WARNING - Something maybe fail.

2016-10-09 19:01:13,263 - \_\_main\_\_ - INFO - Finish

**2.2.2 将日志同时输出到屏幕和日志文件**

logger中添加StreamHandler，可以将日志输出到屏幕上，

[复制代码](javascript:void(0);)

复制代码

import logging

logger = logging.getLogger(\_\_name\_\_)

logger.setLevel(level = logging.INFO)

handler = logging.FileHandler("log.txt")

handler.setLevel(logging.INFO)

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

handler.setFormatter(formatter)

console = logging.StreamHandler()

console.setLevel(logging.INFO)

logger.addHandler(handler)

logger.addHandler(console)

logger.info("Start print log")

logger.debug("Do something")

logger.warning("Something maybe fail.")

logger.info("Finish")

复制代码

[复制代码](javascript:void(0);)

可以在log.txt文件和控制台中看到，

2016-10-09 19:20:46,553 - \_\_main\_\_ - INFO - Start print log

2016-10-09 19:20:46,553 - \_\_main\_\_ - WARNING - Something maybe fail.

2016-10-09 19:20:46,553 - \_\_main\_\_ - INFO - Finish

可以发现，logging有一个日志处理的主对象，其他处理方式都是通过addHandler添加进去，logging中包含的handler主要有如下几种，

handler名称：位置；作用

StreamHandler：logging.StreamHandler；日志输出到流，可以是sys.stderr，sys.stdout或者文件

FileHandler：logging.FileHandler；日志输出到文件

BaseRotatingHandler：logging.handlers.BaseRotatingHandler；基本的日志回滚方式

RotatingHandler：logging.handlers.RotatingHandler；日志回滚方式，支持日志文件最大数量和日志文件回滚

TimeRotatingHandler：logging.handlers.TimeRotatingHandler；日志回滚方式，在一定时间区域内回滚日志文件

SocketHandler：logging.handlers.SocketHandler；远程输出日志到TCP/IP sockets

DatagramHandler：logging.handlers.DatagramHandler；远程输出日志到UDP sockets

SMTPHandler：logging.handlers.SMTPHandler；远程输出日志到邮件地址

SysLogHandler：logging.handlers.SysLogHandler；日志输出到syslog

NTEventLogHandler：logging.handlers.NTEventLogHandler；远程输出日志到Windows NT/2000/XP的事件日志

MemoryHandler：logging.handlers.MemoryHandler；日志输出到内存中的指定buffer

HTTPHandler：logging.handlers.HTTPHandler；通过"GET"或者"POST"远程输出到HTTP服务器

**2.2.3 日志回滚**

使用RotatingFileHandler，可以实现日志回滚，

[复制代码](javascript:void(0);)

复制代码

import logging

from logging.handlers import RotatingFileHandler

logger = logging.getLogger(\_\_name\_\_)

logger.setLevel(level = logging.INFO)

#定义一个RotatingFileHandler，最多备份3个日志文件，每个日志文件最大1K

rHandler = RotatingFileHandler("log.txt",maxBytes = 1\*1024,backupCount = 3)

rHandler.setLevel(logging.INFO)

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

rHandler.setFormatter(formatter)

console = logging.StreamHandler()

console.setLevel(logging.INFO)

console.setFormatter(formatter)

logger.addHandler(rHandler)

logger.addHandler(console)

logger.info("Start print log")

logger.debug("Do something")

logger.warning("Something maybe fail.")

logger.info("Finish")

复制代码

[复制代码](javascript:void(0);)

可以在工程目录中看到，备份的日志文件，

2016/10/09 19:36 732 log.txt

2016/10/09 19:36 967 log.txt.1

2016/10/09 19:36 985 log.txt.2

2016/10/09 19:36 976 log.txt.3

**2.3 设置消息的等级**

可以设置不同的日志等级，用于控制日志的输出，

日志等级：使用范围

FATAL：致命错误

CRITICAL：特别糟糕的事情，如内存耗尽、磁盘空间为空，一般很少使用

ERROR：发生错误时，如IO操作失败或者连接问题

WARNING：发生很重要的事件，但是并不是错误时，如用户登录密码错误

INFO：处理请求或者状态变化等日常事务

DEBUG：调试过程中使用DEBUG等级，如算法中每个循环的中间状态

**2.4 捕获traceback**

Python中的traceback模块被用于跟踪异常返回信息，可以在logging中记录下traceback，

代码，

[复制代码](javascript:void(0);)

复制代码

import logging

logger = logging.getLogger(\_\_name\_\_)

logger.setLevel(level = logging.INFO)

handler = logging.FileHandler("log.txt")

handler.setLevel(logging.INFO)

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

handler.setFormatter(formatter)

console = logging.StreamHandler()

console.setLevel(logging.INFO)

logger.addHandler(handler)

logger.addHandler(console)

logger.info("Start print log")

logger.debug("Do something")

logger.warning("Something maybe fail.")

try:

open("sklearn.txt","rb")

except (SystemExit,KeyboardInterrupt):

raise

except Exception:

logger.error("Faild to open sklearn.txt from logger.error",exc\_info = True)

logger.info("Finish")

复制代码

[复制代码](javascript:void(0);)

控制台和日志文件log.txt中输出，

Start print log

Something maybe fail.

Faild to open sklearn.txt from logger.error

Traceback (most recent call last):

File "G:\zhb7627\Code\Eclipse WorkSpace\PythonTest\test.py", line 23, in <module>

open("sklearn.txt","rb")

IOError: [Errno 2] No such file or directory: 'sklearn.txt'

Finish

也可以使用logger.exception(msg,\_args)，它等价于logger.error(msg,exc\_info = True,\_args)，

将

logger.error("Faild to open sklearn.txt from logger.error",exc\_info = True)

替换为，

logger.exception("Failed to open sklearn.txt from logger.exception")

控制台和日志文件log.txt中输出，

Start print log

Something maybe fail.

Failed to open sklearn.txt from logger.exception

Traceback (most recent call last):

File "G:\zhb7627\Code\Eclipse WorkSpace\PythonTest\test.py", line 23, in <module>

open("sklearn.txt","rb")

IOError: [Errno 2] No such file or directory: 'sklearn.txt'

Finish

**2.5 多模块使用logging**

主模块mainModule.py，

[复制代码](javascript:void(0);)

复制代码

import logging

import subModule

logger = logging.getLogger("mainModule")

logger.setLevel(level = logging.INFO)

handler = logging.FileHandler("log.txt")

handler.setLevel(logging.INFO)

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

handler.setFormatter(formatter)

console = logging.StreamHandler()

console.setLevel(logging.INFO)

console.setFormatter(formatter)

logger.addHandler(handler)

logger.addHandler(console)

logger.info("creating an instance of subModule.subModuleClass")

a = subModule.SubModuleClass()

logger.info("calling subModule.subModuleClass.doSomething")

a.doSomething()

logger.info("done with subModule.subModuleClass.doSomething")

logger.info("calling subModule.some\_function")

subModule.som\_function()

logger.info("done with subModule.some\_function")

复制代码

[复制代码](javascript:void(0);)

子模块subModule.py，

[复制代码](javascript:void(0);)

复制代码

import logging

module\_logger = logging.getLogger("mainModule.sub")

class SubModuleClass(object):

def \_\_init\_\_(self):

self.logger = logging.getLogger("mainModule.sub.module")

self.logger.info("creating an instance in SubModuleClass")

def doSomething(self):

self.logger.info("do something in SubModule")

a = []

a.append(1)

self.logger.debug("list a = " + str(a))

self.logger.info("finish something in SubModuleClass")

def som\_function():

module\_logger.info("call function some\_function")

复制代码

[复制代码](javascript:void(0);)

执行之后，在控制和日志文件log.txt中输出，

2016-10-09 20:25:42,276 - mainModule - INFO - creating an instance of subModule.subModuleClass

2016-10-09 20:25:42,279 - mainModule.sub.module - INFO - creating an instance in SubModuleClass

2016-10-09 20:25:42,279 - mainModule - INFO - calling subModule.subModuleClass.doSomething

2016-10-09 20:25:42,279 - mainModule.sub.module - INFO - do something in SubModule

2016-10-09 20:25:42,279 - mainModule.sub.module - INFO - finish something in SubModuleClass

2016-10-09 20:25:42,279 - mainModule - INFO - done with subModule.subModuleClass.doSomething

2016-10-09 20:25:42,279 - mainModule - INFO - calling subModule.some\_function

2016-10-09 20:25:42,279 - mainModule.sub - INFO - call function some\_function

2016-10-09 20:25:42,279 - mainModule - INFO - done with subModule.some\_function

首先在主模块定义了logger'mainModule'，并对它进行了配置，就可以在解释器进程里面的其他地方通过getLogger('mainModule')得到的对象都是一样的，不需要重新配置，可以直接使用。定义的该logger的子logger，都可以共享父logger的定义和配置，所谓的父子logger是通过命名来识别，任意以'mainModule'开头的logger都是它的子logger，例如'mainModule.sub'。

实际开发一个application，首先可以通过logging配置文件编写好这个application所对应的配置，可以生成一个根logger，如'PythonAPP'，然后在主函数中通过fileConfig加载logging配置，接着在application的其他地方、不同的模块中，可以使用根logger的子logger，如'PythonAPP.Core'，'PythonAPP.Web'来进行log，而不需要反复的定义和配置各个模块的logger。

**3 通过JSON或者YAML文件配置logging模块**

尽管可以在Python代码中配置logging，但是这样并不够灵活，最好的方法是使用一个配置文件来配置。在Python 2.7及以后的版本中，可以从字典中加载logging配置，也就意味着可以通过JSON或者YAML文件加载日志的配置。

**3.1 通过JSON文件配置**

JSON配置文件，

{

"version":1,

"disable\_existing\_loggers":false,

"formatters":{

"simple":{

"format":"%(asctime)s - %(name)s - %(levelname)s - %(message)s"

}

},

"handlers":{

"console":{

"class":"logging.StreamHandler",

"level":"DEBUG",

"formatter":"simple",

"stream":"ext://sys.stdout"

},

"info\_file\_handler":{

"class":"logging.handlers.RotatingFileHandler",

"level":"INFO",

"formatter":"simple",

"filename":"info.log",

"maxBytes":"10485760",

"backupCount":20,

"encoding":"utf8"

},

"error\_file\_handler":{

"class":"logging.handlers.RotatingFileHandler",

"level":"ERROR",

"formatter":"simple",

"filename":"errors.log",

"maxBytes":10485760,

"backupCount":20,

"encoding":"utf8"

}

},

"loggers":{

"my\_module":{

"level":"ERROR",

"handlers":["info\_file\_handler"],

"propagate":"no"

}

},

"root":{

"level":"INFO",

"handlers":["console","info\_file\_handler","error\_file\_handler"]

}

}

通过JSON加载配置文件，然后通过logging.dictConfig配置logging，

[复制代码](javascript:void(0);)

复制代码

import json

import logging.config

import os

def setup\_logging(default\_path = "logging.json",default\_level = logging.INFO,env\_key = "LOG\_CFG"):

path = default\_path

value = os.getenv(env\_key,None)

if value:

path = value

if os.path.exists(path):

with open(path,"r") as f:

config = json.load(f)

logging.config.dictConfig(config)

else:

logging.basicConfig(level = default\_level)

def func():

logging.info("start func")

logging.info("exec func")

logging.info("end func")

if \_\_name\_\_ == "\_\_main\_\_":

setup\_logging(default\_path = "logging.json")

func()

复制代码

[复制代码](javascript:void(0);)

**3.2 通过YAML文件配置**

通过YAML文件进行配置，比JSON看起来更加简介明了，

version: 1

disable\_existing\_loggers: False

formatters:

simple:

format: "%(asctime)s - %(name)s - %(levelname)s - %(message)s"

handlers:

console:

class: logging.StreamHandler

level: DEBUG

formatter: simple

stream: ext://sys.stdout

info\_file\_handler:

class: logging.handlers.RotatingFileHandler

level: INFO

formatter: simple

filename: info.log

maxBytes: 10485760

backupCount: 20

encoding: utf8

error\_file\_handler:

class: logging.handlers.RotatingFileHandler

level: ERROR

formatter: simple

filename: errors.log

maxBytes: 10485760

backupCount: 20

encoding: utf8

loggers:

my\_module:

level: ERROR

handlers: [info\_file\_handler]

propagate: no

root:

level: INFO

handlers: [console,info\_file\_handler,error\_file\_handler]

通过YAML加载配置文件，然后通过logging.dictConfig配置logging，

[复制代码](javascript:void(0);)

复制代码

import yaml

import logging.config

import os

def setup\_logging(default\_path = "logging.yaml",default\_level = logging.INFO,env\_key = "LOG\_CFG"):

path = default\_path

value = os.getenv(env\_key,None)

if value:

path = value

if os.path.exists(path):

with open(path,"r") as f:

config = yaml.load(f)

logging.config.dictConfig(config)

else:

logging.basicConfig(level = default\_level)

def func():

logging.info("start func")

logging.info("exec func")

logging.info("end func")

if \_\_name\_\_ == "\_\_main\_\_":

setup\_logging(default\_path = "logging.yaml")

func()

复制代码

[复制代码](javascript:void(0);)