

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：

- 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；
- 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；

logging 库采取了模块化的设计，提供了许多组件：记录器、处理器、过滤器和格式化器。

- Formatter 指明了最终输出中日志记录的布局。
- Logger 暴露了应用程序代码能直接使用的接口。
- Handler 将（记录器产生的）日志记录发送至合适的目的地。
- Filter 提供了更好的粒度控制，它可以决定输出哪些日志记录。

Loggers

Logger 对象要做三件事情。首先，它们向应用代码暴露了许多方法，这样应用可以在运行时记录消息。其次，记录器对象通过严重程度（默认的过滤设施）或者过滤器对象来决定哪些日志消息需要记录下来。第三，记录器对象将相关的日志消息传递给所有感兴趣的日志处理器。

常用的记录器对象的方法分为两类：配置和发送消息。

这些是最常用的配置方法：

`Logger.setLevel()` 指定 logger 将会处理的最低的安全等级日志信息，debug 是最低的内置安全等级，critical 是最高的内建安全等级。例如，如果严重程度为 INFO，记录器将只处理 INFO, WARNING, ERROR 和 CRITICAL 消息，DEBUG 消息被忽略。`Logger.addHandler()` 和 `Logger.removeHandler()` 从记录器对象中添加和删除处理程序对象。处理器详见 Handlers。

`Logger.addFilter()` 和 `Logger.removeFilter()` 从记录器对象添加和删除过滤器对象。

Handlers

处理程序对象负责将适当的日志消息（基于日志消息的严重性）分派到处理程序的指定目标。Logger 对象可以通过 `addHandler()` 方法增加零个或多个 handler 对象。举个例子，一个应用可以将所有的日志消息发送至日志文件，所有的错误级别 (error) 及以上的日志消息发送至标准输出，所有的严重级别 (critical) 日志消息发送至某个电子邮箱。在这个例子中需要三个独立的处理器，每一个负责将特定级别的消息发送至特定的位置。

常用的有4种：

- `logging.StreamHandler` -> 控制台输出
使用这个 Handler 可以向类似与 `sys.stdout` 或者 `sys.stderr` 的任何文件对象 (file object) 输出信息。
它的构造函数是：
`StreamHandler([strm])`
其中 `strm` 参数是一个文件对象。默认是 `sys.stderr`
- `logging.FileHandler` -> 文件输出 和 `StreamHandler` 类似，用于向一个文件输出日志信息。不过 `FileHandler` 会帮你打开这个文件。它的构造函数是：
`FileHandler(filename[,mode])`

`filename` 是文件名，必须指定一个文件名。

`mode` 是文件的打开方式。默认是 'a', 即添加到文件末尾。

- `logging.handlers.RotatingFileHandler` -> 按照大小自动分割日志文件，一旦达到指定的大小重新生成文件`

这个 `Handler` 类似于上面的 `FileHandler`, 但是它可以管理文件大小。当文件达到一定大小之后，它会自动将当前日志文件改名，然后创建一个新的同名日志文件继续输出。比如日志文件是 `chat.log`. 当 `chat.log` 达到指定的大小之后，`RotatingFileHandler` 自动把文件改名为 `chat.log.1`. 不过，如果 `chat.log.1` 已经存在，会先把 `chat.log.1` 重命名为 `chat.log.2` ... 最后重新创建 `chat.log`, 继续输出日志信息。它的构造函数是：

```
RotatingFileHandler( filename[, mode[, maxBytes[, backupCount]]])
```

其中 `filename` 和 `mode` 两个参数和 `FileHandler` 一样。

`maxBytes` 用于指定日志文件的最大文件大小。如果 `maxBytes` 为 0, 意味着日志文件可以无限大，这时上面描述的重命名过程就不会发生。`backupCount` 用于指定保留的备份文件的个数。比如，如果指定为 2, 当上面描述的重命名过程发生时，原有的 `chat.log.2` 并不会被更名，而是被删除。

- `logging.handlers.TimedRotatingFileHandler` -> 按照时间自动分割日志文件

这个 `Handler` 和 `RotatingFileHandler` 类似，不过，它没有通过判断文件大小来决定何时重新创建日志文件，而是间隔一定时间就自动创建新的日志文件。重命名的过程与 `RotatingFileHandler` 类似，不过新的文件不是附加数字，而是当前时间。它的构造函数是：

```
TimedRotatingFileHandler( filename [,when [,interval [,backupCount]]])
```

其中 `filename` 参数和 `backupCount` 参数和 `RotatingFileHandler` 具有相同的意义。

`interval` 是时间间隔。

`when` 参数是一个字符串。表示时间间隔的单位，不区分大小写。它有以下取值：

S 秒

M 分

H 小时

D 天

W 每星期（`interval==0`代表星期一）

midnight 每天凌晨

配置方法：

- `setLevel()` 方法和日志对象的一样，指明了将会分发日志的最低级别。为什么会有两个 `setLevel()` 方法？记录器的级别决定了消息是否要传递给处理器。每个处理器的级别决定了消息是否要分发。
- `setFormatter()` 为该处理器选择一个格式化器。
- `addFilter()` 和 `removeFilter()` 分别配置和取消配置处理程序上的过滤器对象。

Formatters

`Formatter` 对象设置日志信息最后的规则、结构和内容，默认的时间格式为 `%Y-%m-%d %H:%M:%S`, 下面是 `Formatter` 常用的一些信息

<code>%(name)s</code>	Logger 的名字
<code>%(levelno)s</code>	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

eg. 输出log到控制台以及将日志写入log文件。

保存2种类型的log，all.log 保存debug, info, warning, critical 信息，error.log则只保存error信息，同时按照时间自动分割日志文件。

```
import logging
from logging import handlers

class Logger(object):
    level_relations = {
        'debug':logging.DEBUG,
        'info':logging.INFO,
        'warning':logging.WARNING,
        'error':logging.ERROR,
        'crit':logging.CRITICAL
    }#日志级别关系映射

    def __init__(self,filename,level='info',when='D',backCount=3,fmt='%(asctime)s
- %(pathname)s[line:%(lineno)d] - %(levelname)s: %(message)s'):
        self.logger = logging.getLogger(filename)
        format_str = logging.Formatter(fmt)#设置日志格式
        self.logger.setLevel(self.level_relations.get(level))#设置日志级别
        sh = logging.StreamHandler()#往屏幕上输出
        sh.setFormatter(format_str) #设置屏幕上显示的格式
        th =
handlers.TimedRotatingFileHandler(filename=filename,when=when,backupCount=backCount,encoding='utf-8')#往文件里写入#指定间隔时间自动生成文件的处理器
        #实例化TimedRotatingFileHandler
        #interval是时间间隔，backupCount是备份文件的个数，如果超过这个个数，就会自动删
```

除，when是间隔的时间单位，单位有以下几种：

```
# S 秒
# M 分
# H 小时、
# D 天、
# W 每星期（interval==0时代表星期一）
# midnight 每天凌晨
th.setFormatter(format_str)#设置文件里写入的格式
self.logger.addHandler(sh) #把对象加到logger里
self.logger.addHandler(th)
if __name__ == '__main__':
    log = Logger('all.log', level='debug')
    log.logger.debug('debug')
    log.logger.info('info')
    log.logger.warning('警告')
    log.logger.error('报错')
    log.logger.critical('严重')
    Logger('error.log', level='error').logger.error('error')
```

logging 模块使用

基本使用

- logging 日志基本格式化

```
logging.basicConfig(format='%(asctime)s - %(pathname)s[line:%(lineno)d] - %
(levelname)s: %(message)s',
                    level=logging.DEBUG)
```

- logging 日志基本格式

```
logging.basicConfig(level=logging.DEBUG,#控制台打印的日志级别
                    filename='new.log',
                    filemode='a',##模式，有w和a，w就是写模式，每次都会重新写日志，覆盖
之前的日志
                    #a是追加模式，默认如果不写的话，就是追加模式
                    format=
'%(asctime)s - %(pathname)s[line:%(lineno)d] - %(levelname)s:
%(message)s'
                    #日志格式
                    )
```

- 基本方式

配置logging基本的设置，然后在控制台输出日志

```
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")
```

- 打印日志
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。

```
logging.debug('debug 信息')
logging.info('info 信息')
logging.warning('warning 信息')
logging.error('error 信息')
logging.critical('critical 信息')
```

这种打印日志需要将日志格式化放在首次调用的位置，一旦格式化执行执行后，其余格式化将不再执行
[参考](#)

如需要多个日志输出则需要使用动态输出日志 [参考](#)

基本参数

Formatters 定义了 Logger 记录的输出格式。
定义了最终 log 信息的内容格式，应用可以直接实例化 Foamatter 类。信息格式字符串用 %(<dictionary key>)s 风格的字符串做替换。
logging.basicConfig 函数各参数：

参数：	作用
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
-	-
%(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

参数:	作用
<code>%(funcName)s</code>	打印日志的当前函数
<code>%(lineno)d</code>	打印日志的当前行号
<code>%(asctime)s</code>	打印日志的时间
<code>%(thread)d</code>	打印线程ID
<code>%(threadName)s</code>	打印线程名称
<code>%(process)d</code>	打印进程ID
<code>%(message)s</code>	打印日志信息
-	-
<code>datefmt</code>	指定时间格式，同 <code>time.strftime()</code>
<code>level</code>	设置日志级别，默认为 <code>logging.WARNING</code>
<code>stream</code>	指定将日志的输出流，可以指定输出到 <code>sys.stderr</code> <code>sys.stdout</code> 或者文件，默认输出到 <code>sys.stderr</code> ，当 <code>stream</code> 和 <code>filename</code> 同时指定时， <code>stream</code> 被忽略

logging.logger & logging.handler

logger

Logger从来不直接实例化，经常通过logging模块级方法（Module-Level Function）`logging.getLogger(name)`来获得，其中如果`name`不给定就用`root`。

名字是以点号分割的命名方式命名的(a.b.c)。

对同一个名字的多调用`logging.getLogger()`方法会返回同一个logger对象。

这种命名方式里面，后面的loggers是前面logger的子logger，自动继承父loggers的log信息，正因为此,没有必要把一个应用的所有logger都配置一遍，只要把顶层的logger配置好了，然后子logger根据需要继承就行了。

logging.Logger对象扮演了三重角色:

- 首先,它暴露给应用几个方法以便应用可以在运行时写log.
- 其次,Logger对象按照log信息的严重程度或者根据filter对象来决定如何处理log信息(默认过滤功能).
- 最后,logger还负责把log信息传送给相关的handlers.

logger & handler 基本使用

典型的日志记录步骤:

- 创建 logger
- 创建 handler
- 定义 formatter
- 给 handler 添加 formatter
- 给 logger 添加 handler

代码:

```
import logging

# 1、创建一个logger
logger = logging.getLogger('mylogger')
logger.setLevel(logging.DEBUG)

# 2、创建一个handler，用于写入日志文件
fh = logging.FileHandler('test.log')
fh.setLevel(logging.DEBUG)

# 再创建一个handler，用于输出到控制台
ch = logging.StreamHandler()
ch.setLevel(logging.DEBUG)

# 3、定义handler的输出格式（formatter）
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

# 4、给handler添加formatter
fh.setFormatter(formatter)
ch.setFormatter(formatter)

# 5、给logger添加handler
logger.addHandler(fh)
logger.addHandler(ch)
```

参考

将日志写入文件

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

```
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")
```

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

```
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")
```

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

handler名称	位置	作用
HTTPHandler	logging.handlers.HTTPHandler	通过"GET"或者"POST"远程输出到HTTP服务器

日志回滚

即分割日志

使用RotatingFileHandler，可以实现日志回滚

```
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")
```

设置消息的等级

可以设置不同的消息等级，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

级别	数字值
FATAL	50
CRTTCAL	40
ERROR	30
WARNING	20
INFO	10
DEBUG	0

默认等级是 WAENING, 这意味着仅仅这个等级及以上的才会反馈信息, 除非 logging 模块被用来做其它事情。

捕获 traceback

```
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("Faield to open sklearn.txt from logger.error", exc_info = True)

logger.info("Finish")
```

也可以使用 `logger.exception(msg, _args)`, 它等价于 `logger.error(msg, exc_info = True, _args)`, 将 `logger.error("Faield to open sklearn.txt from logger.error", exc_info = True)` 替换为 `logger.exception("Failed to open sklearn.txt from logger.exception")`

多模块使用 logging

主模块 mainModule.py

```

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")

```

子模块 submodule.py

```

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")

```

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.

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。

通过 JSON 文件配置

JSON文件配置

```
{
  "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

```

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()

```

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

```
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()
```

遇到的问题

logging 重复写日志问题

用以上的方式，在多模块调用的时候，遇到了重复写日志的问题，第一条写一次、第二条写两次、第三条写三次..... 后找到原因及解决方案

- 原因：没有移除 handler

在某个模块中定义了 `logging.logger`, 之后每次进入此模块没有进行判断，都会再 `get` 一个 handler, 会和之前 `get` 的所有 handler 同时输出信息，所以就造成重复输出的问题。

就是你第二次调用 `log` 的时候，根据 `getLogger(name)` 里的 `name` 获取同一个 logger，而这个 logger 里已经有了第一次你添加的 handler，第二次调用又添加了一个 handler，所以，这个 logger 里有了两个同样的 handler，以此类推，调用几次就会有几个 handler。

我第一次输出的日志..... 打印了 2.7G 的 txt 文本，给 vscode 弄崩了都。

- 解决方案：
- 1. 每次创建不同 name 的 logger, 每次都是新 logger, 不会有添加多个 handler 的问题。（ps:这个方法太笨，不过我之前就是这么干的。。。）
- 2. 像上面一样每次记录完日志之后，调用 `removeHandler()` 把这个 logger 里的 handler 移除掉。
- 3. 在 `log` 方法里做判断，如果这个 logger 已有 handler, 则不再添加 handler.
- 4. 与方法2一样，不过把用 `pop` 把 logger 的 handler 列表中的 handler 移除。

方法3 & 方法4 代码示例：

方法3

```
import logging

def log(message):
    logger = logging.getLogger('testlog')

    # 这里进行判断，如果logger.handlers列表为空，则添加，否则，直接去写日志
    if not logger.handlers:
        streamhandler = logging.StreamHandler()
```

```
        streamhandler.setLevel(logging.ERROR)
        formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(name)s - %
(message)s')
        streamhandler.setFormatter(formatter)
        logger.addHandler(streamhandler)

    logger.error(message)

if __name__ == '__main__':
    log('hi')
    log('hi too')
    log('hi three')
```

方法4

```
import logging

def log(message):
    logger = logging.getLogger('testlog')

    streamhandler = logging.StreamHandler()
    streamhandler.setLevel(logging.ERROR)
    formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(name)s - %
(message)s')
    streamhandler.setFormatter(formatter)

    logger.addHandler(streamhandler)

    logger.error(message)

    # 用pop方法把logger.handlers列表中的handler移除，注意如果你add了多个handler，这里
    需多次pop，或者可以直接为handlers列表赋空值
    logger.handlers.pop()
    # logger.handler = []

if __name__ == '__main__':
    log('hi')
    log('hi too')
    log('hi three')
```

至此，困扰了几天的日志问题算是告一段落了，写这篇又花了些时间，完全是为了关掉 **Chrome** 上一堆开着的标签页哈哈哈哈哈

广州这两天也是开始凉了，前天冻的我颈椎又犯了大概，支着我这个破颈椎写完这篇

本来是用着有道云笔记写的，可惜昨天实在是受不了有道的 **Markdown** 了，在 **VsCode** 上写的第一篇，设好了云端备份，准备之后也该弄个博客了，不过没想好弄什么方式的哈哈哈哈哈，纠结啧啧啧

这里的问题是在把公司的测试用例和脚本从 **Python2** 迁到 **Python3** 的收尾时候遇到的。接下来就是把有道

的笔记都迁过来，以及新的笔记，哦还有养好脖子，好好学习

近期的任务大概是，把迁到 **Python3** 的用例在 **Linux** 上跑起来，在不影响现阶段用例运行环境的前提下，弄好了差不多一半吧大概

哈哈哈哈哈，随时都有删库跑路的风险，瑟瑟发抖

2018.11.21

Reference

[python logging模块](#)

[Python logger模块](#)

[Python + logging 输出到屏幕，将log日志写入文件](#)

[python logging 重复写日志问题](#)