python 日志 logging模块(详细解析)

1 基本使用

转自：https://www.cnblogs.com/wf-linux/archive/2018/08/01/9400354.html
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")
```

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 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，

```
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")
```

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 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
```

2.3 日志回滚

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")
```

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，

```
import logging
logger = logging.getLogger(__name__)
```

```
logger.setLevel(level = logging.INFO) 4 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")
```

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`，

```
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")
```

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，

```
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()
```

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，

```yaml
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，

```python
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()
```