Python日志库logging总结

在部署项目时，不可能直接将所有的信息都输出到控制台中，我们可以将这些信息记录到日志文件中，这样不仅方便我们查看程序运行时的情况，也可以在项目出现故障时根据运行时产生的日志快速定位问题出现的位置。

1、日志级别

Python 标准库 logging 用作记录日志，默认分为六种日志级别（括号为级别对应的数值），NOTSET（0）、DEBUG（10）、INFO（20）、WARNING（30）、ERROR（40）、CRITICAL（50）。我们自定义日志级别时注意不要和默认的日志级别数值相同，logging 执行时输出大于等于设置的日志级别的日志信息，如设置日志级别是 INFO，则 INFO、WARNING、ERROR、CRITICAL 级别的日志都会输出。

2、logging 流程

官方的 logging 模块工作流程图如下：

从下图中我们可以看出看到这几种 Python 类型，**Logger**、**LogRecord**、**Filter**、**Handler**、**Formatter**。

类型说明：

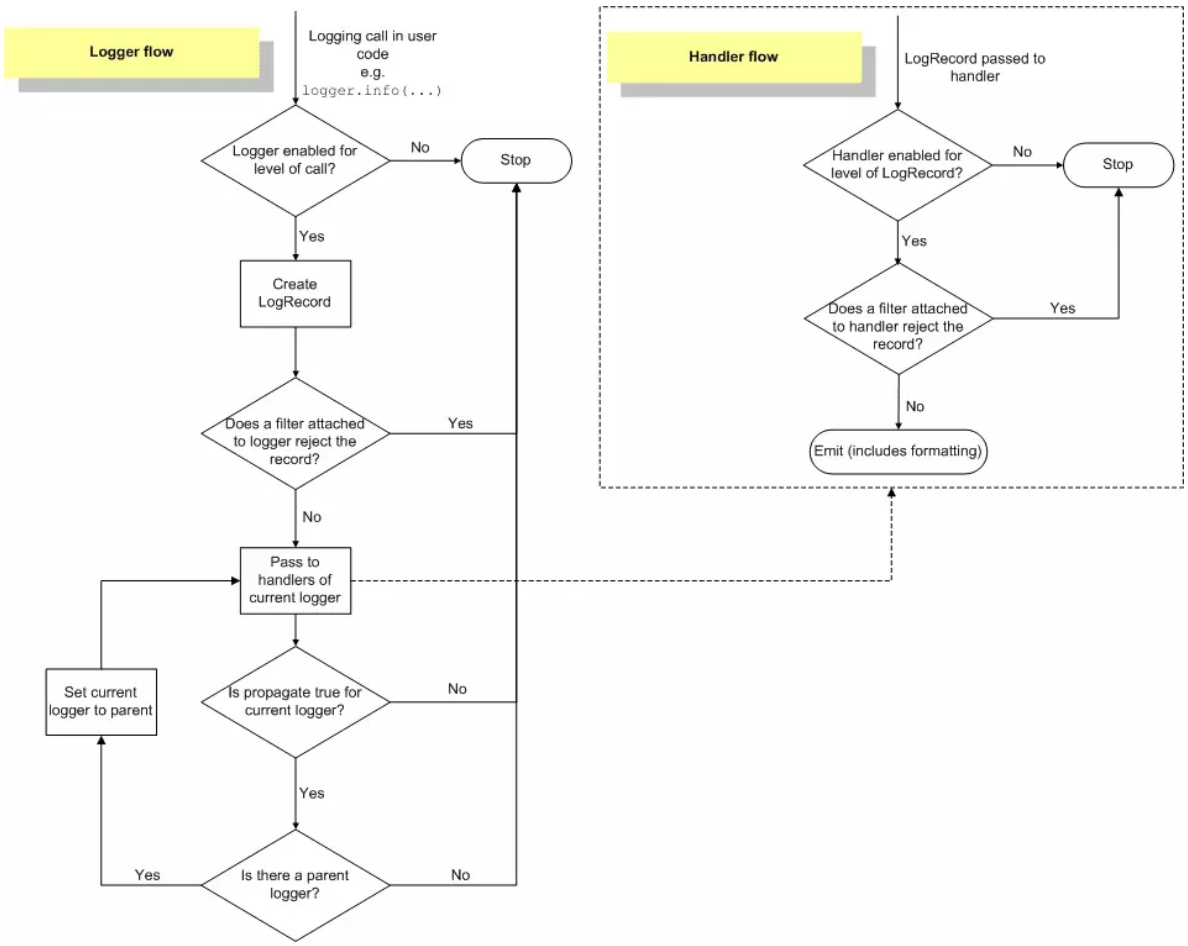
Logger：日志，暴露函数给应用程序，基于日志记录器和过滤器级别决定哪些日志有效。

LogRecord： 日志记录器，将日志传到相应的处理器处理。

Handler： 处理器, 将(日志记录器产生的)日志记录发送至合适的目的地。

Filter： 过滤器, 提供了更好的粒度控制,它可以决定输出哪些日志记录。

Formatter：格式化器, 指明了最终输出中日志记录的布局。



1. 判断 Logger 对象对于设置的级别是否可用，如果可用，则往下执行，否则，流程结束。
2. 创建 LogRecord 对象，如果注册到 Logger 对象中的 Filter 对象过滤后返回 False，则不记录日志，流程结束，否则，则向下执行。
3. LogRecord 对象将 Handler 对象传入当前的 Logger 对象，（图中的子流程）如果 Handler 对象的日志级别大于设置的日志级别，再判断注册到 Handler 对象中的 Filter 对象过滤后是否返回 True 而放行输出日志信息，否则不放行，流程结束。
4. 如果传入的 Handler 大于 Logger 中设置的级别，也即 Handler 有效，则往下执行，否则，流程结束。
5. 判断这个 Logger 对象是否还有父 Logger 对象，如果没有（代表当前 Logger 对象是最顶层的 Logger 对象 root Logger），流程结束。否则将 Logger 对象设置为它的父 Logger 对象，重复上面的 3、4 两步，输出父类 Logger 对象中的日志输出，直到是 root Logger 为止。

3、日志输出格式

日志的输出格式可以认为设置，默认格式为下图所示。

4、基本使用

logging 使用非常简单，使用 basicConfig() 方法就能满足基本的使用需要，如果方法没有传入参数，会根据默认的配置创建Logger 对象，默认的日志级别被设置为 **WARNING**，默认的日志输出格式如上图，该函数可选的参数如下表所示。

参数名称	参数描述
filename	日志输出到文件的文件名
filemode	文件模式，r[+]、w[+]、a[+]
format	日志输出的格式
datefat	日志附带日期时间的格式
style	格式占位符，默认为 "%" 和 “{}”
level	设置日志输出级别
stream	定义输出流，用来初始化 StreamHandler 对象，不能 filename 参数一起使用，否则会ValueError 异常
handles	定义处理器，用来创建 Handler 对象，不能和 filename 、stream 参数一起使用，否则也会抛出 ValueError 异常

示例代码如下：

```
import logging

logging.basicConfig()
logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')
```

输出结果如下：

```
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical message
```

传入常用的参数，示例代码如下（这里日志格式占位符中的变量放到后面介绍）：

```
import logging

logging.basicConfig(filename="test.log", filemode="w", format="%(asctime)s %
(name)s:%(levelname)s:%(message)s", datefmt="%d-%M-%Y %H:%M:%S",
level=logging.DEBUG)
logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')
```

生成的日志文件 test.log ， 内容如下：

```
13-10-18 21:10:32 root:DEBUG:This is a debug message
13-10-18 21:10:32 root:INFO:This is an info message
13-10-18 21:10:32 root:WARNING:This is a warning message
13-10-18 21:10:32 root:ERROR:This is an error message
13-10-18 21:10:32 root:CRITICAL:This is a critical message
```

但是当发生异常时，直接使用无参数的 debug()、info()、warning()、error()、critical() 方法并不能记录异常信息，需要设置 exc_info 参数为 True 才可以，或者使用 exception() 方法，还可以使用 log() 方法，但还要设置日志级别和 exc_info 参数。

```
import logging

logging.basicConfig(filename="test.log", filemode="w", format="%(asctime)s %
(name)s:%(levelname)s:%(message)s", datefmt="%d-%M-%Y %H:%M:%S",
level=logging.DEBUG)
a = 5
b = 0
try:
    c = a / b
except Exception as e:
    # 下面三种方式三选一，推荐使用第一种
    logging.exception("Exception occurred")
    logging.error("Exception occurred", exc_info=True)
    logging.log(level=logging.DEBUG, msg="Exception occurred", exc_info=True)
```

5、自定义 Logger

上面的基本使用可以让我们快速上手 logging 模块，但一般并不能满足实际使用，我们还需要自定义 Logger。

一个系统只有一个 Logger 对象，并且该对象不能被直接实例化，没错，这里用到了单例模式，获取 Logger 对象的方法为 **getLogger**。

注意：这里的单例模式并不是说只有一个 Logger 对象，而是指整个系统只有一个根 Logger 对象，Logger 对象在执行 info()、error() 等方法时实际上调用都是根 Logger 对象对应的 info()、error() 等方法。

我们可以创造多个 Logger 对象，但是真正输出日志的是根 Logger 对象。每个 Logger 对象都可以设置一个名字，如果设置 `logger = logging.getLogger(__name__)`，**name** 是 Python 中的一个特殊内置变量，他代表当前模块的名称（默认为 **main**）。则 Logger 对象的 name 为建议使用使用以点号作为分隔符的命名空间等级制度。

Logger 对象可以设置多个 Handler 对象和 Filter 对象，Handler 对象又可以设置 Formatter 对象。Formatter 对象用来设置具体的输出格式，常用变量格式如下表所示，所有参数见 Python(3.7)官方文档:

变量	格式	变量描述
asctime	%(asctime)s	将日志的时间构造成可读的形式，默认情况下是精确到毫秒，如 2018-10-13 23:24:57,832，可以额外指定 datefmt 参数来指定该变量的格式
name	%(name)	日志对象的名称
filename	%(filename)s	不包含路径的文件名
pathname	%(pathname)s	包含路径的文件名
funcName	%(funcName)s	日志记录所在的函数名
levelname	%(levelname)s	日志的级别名称
message	%(message)s	具体的日志信息
lineno	%(lineno)d	日志记录所在的行号
pathname	%(pathname)s	完整路径
process	%(process)d	当前进程ID
processName	%(processName)s	当前进程名称
thread	%(thread)d	当前线程ID
threadName	%threadName)s	当前线程名称

Logger 对象和 Handler 对象都可以设置级别，而默认 Logger 对象级别为 30 ，也即 WARNING，默认 Handler 对象级别为 0，也即 NOTSET。logging 模块这样设计是为了更好的灵活性，比如有时候我们既想在控制台中输出DEBUG 级别的日志，又想在文件中输出WARNING级别的日志。可以只设置一个最低级别的 Logger 对象，两个不同级别的 Handler 对象，示例代码如下：

```
import logging
import logging.handlers

logger = logging.getLogger("logger")

handler1 = logging.StreamHandler()
handler2 = logging.FileHandler(filename="test.log")

logger.setLevel(logging.DEBUG)
handler1.setLevel(logging.WARNING)
handler2.setLevel(logging.DEBUG)
```

```
formatter = logging.Formatter("%(asctime)s %(name)s %(levelname)s %(message)s")
handler1.setFormatter(formatter)
handler2.setFormatter(formatter)

logger.addHandler(handler1)
logger.addHandler(handler2)

# 分别为 10、30、30
# print(handler1.level)
# print(handler2.level)
# print(logger.level)

logger.debug('This is a customer debug message')
logger.info('This is an customer info message')
logger.warning('This is a customer warning message')
logger.error('This is an customer error message')
logger.critical('This is a customer critical message')
```

控制台输出结果为：

```
2018-10-13 23:24:57,832 logger WARNING This is a customer warning message
2018-10-13 23:24:57,832 logger ERROR This is an customer error message
2018-10-13 23:24:57,832 logger CRITICAL This is a customer critical message
```

文件中输出内容为：

```
2018-10-13 23:44:59,817 logger DEBUG This is a customer debug message
2018-10-13 23:44:59,817 logger INFO This is an customer info message
2018-10-13 23:44:59,817 logger WARNING This is a customer warning message
2018-10-13 23:44:59,817 logger ERROR This is an customer error message
2018-10-13 23:44:59,817 logger CRITICAL This is a customer critical message
```

创建了自定义的 Logger 对象，就不要在用 logging 中的日志输出方法了，这些方法使用的是默认配置的 Logger 对象，否则会输出的日志信息会重复。

```
import logging
import logging.handlers

logger = logging.getLogger("logger")
handler = logging.StreamHandler()
handler.setLevel(logging.DEBUG)
formatter = logging.Formatter("%(asctime)s %(name)s %(levelname)s %(message)s")
handler.setFormatter(formatter)
logger.addHandler(handler)
logger.debug('This is a customer debug message')
logging.info('This is an customer info message')
logger.warning('This is a customer warning message')
logger.error('This is an customer error message')
logger.critical('This is a customer critical message')
```

输出结果如下（可以看到日志信息被输出了两遍）：

```
2018-10-13 22:21:35,873 logger WARNING This is a customer warning message
WARNING:logger:This is a customer warning message
2018-10-13 22:21:35,873 logger ERROR This is an customer error message
ERROR:logger:This is an customer error message
2018-10-13 22:21:35,873 logger CRITICAL This is a customer critical message
CRITICAL:logger:This is a customer critical message
```

说明：在引入有日志输出的 python 文件时，如 import test.py，在满足大于当前设置的日志级别后就会输出导入文件中的日志。

6、Logger 配置

通过上面的例子，我们知道创建一个 Logger 对象所需的配置了，上面直接硬编码在程序中配置对象，配置还可以从字典类型的对象和配置文件获取。打开 logging.config Python 文件，可以看到其中的配置解析转换函数。

从字典中获取配置信息：

```
import logging.config

config = {
    'version': 1,
    'formatters': {
        'simple': {
            'format': '%(asctime)s - %(name)s - %(levelname)s - %(message)s',
        },
        # 其他的 formatter
    },
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
            'level': 'DEBUG',
            'formatter': 'simple'
        },
        'file': {
            'class': 'logging.FileHandler',
            'filename': 'logging.log',
            'level': 'DEBUG',
            'formatter': 'simple'
        },
        # 其他的 handler
    },
    'loggers':{
        'StreamLogger': {
            'handlers': ['console'],
            'level': 'DEBUG',
        },
        'FileLogger': {
            # 既有 console Handler，还有 file Handler
            'handlers': ['console', 'file'],
            'level': 'DEBUG',
        },
        # 其他的 Logger
    }
}

logging.config.dictConfig(config)
```

```
StreamLogger = logging.getLogger("StreamLogger")
FileLogger = logging.getLogger("FileLogger")
# 省略日志输出
```

从配置文件中获取配置信息：

常见的配置文件有 ini 格式、yaml 格式、JSON 格式，或者从网络中获取都是可以的，只要有相应的文件解析器解析配置即可，下面只展示了 ini 格式和 yaml 格式的配置。

test.ini 文件

```
[loggers]
keys=root,sampleLogger

[handlers]
keys=consoleHandler

[formatters]
keys=sampleFormatter

[logger_root]
level=DEBUG
handlers=consoleHandler

[logger_sampleLogger]
level=DEBUG
handlers=consoleHandler
qualname=sampleLogger
propagate=0

[handler_consoleHandler]
class=StreamHandler
level=DEBUG
formatter=sampleFormatter
args=(sys.stdout,)

[formatter_sampleFormatter]
format=%(asctime)s - %(name)s - %(levelname)s - %(message)s
```

testinit.py 文件

```
import logging.config

logging.config.fileConfig(fname='test.ini', disable_existing_loggers=False)
logger = logging.getLogger("sampleLogger")
# 省略日志输出
```

test.yaml 文件

```
version: 1
formatters:
  simple:
    format: '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
handlers:
  console:
    class: logging.StreamHandler
```

```
    level: DEBUG
    formatter: simple

loggers:
  simpleExample:
    level: DEBUG
    handlers: [console]
    propagate: no
root:
  level: DEBUG
  handlers: [console]
```

testyaml.py 文件

```
import logging.config
# 需要安装 pyymal 库
import yaml

with open('test.yaml', 'r') as f:
    config = yaml.safe_load(f.read())
    logging.config.dictConfig(config)

logger = logging.getLogger("sampleLogger")
# 省略日志输出
```

7、实战中的问题

1、中文乱码

上面的例子中日志输出都是英文内容，发现不了将日志输出到文件中会有中文乱码的问题，如何解决到这个问题呢？FileHandler 创建对象时可以设置文件编码，如果将文件编码设置为 “utf-8”（utf-8 和 utf8 等价），就可以解决中文乱码问题啦。一种方法是自定义 Logger 对象，需要写很多配置，另一种方法是使用默认配置方法 basicConfig()，传入 handlers 处理器列表对象，在其中的 handler 设置文件的编码。网上很多都是无效的方法，关键参考代码如下：

```
# 自定义 Logger 配置
handler = logging.FileHandler(filename="test.log", encoding="utf-8")
复制代码
# 使用默认的 Logger 配置
logging.basicConfig(handlers=[logging.FileHandler("test.log", encoding="utf-
8")], level=logging.DEBUG)
```

2、临时禁用日志输出

有时候我们又不想让日志输出，但在这后又想输出日志。如果我们打印信息用的是 print() 方法，那么就需要把所有的 print() 方法都注释掉，而使用了 logging 后，我们就有了一键开关闭日志的 "魔法"。一种方法是在使用默认配置时，给 logging.disabled() 方法传入禁用的日志级别，就可以禁止设置级别以下的日志输出了，另一种方法时在自定义 Logger 时，Logger 对象的 disable 属性设为 True，默认值是 False，也即不禁用。

```
logging.disable(logging.INFO)

logger.disabled = True
```

3、日志文件按照时间划分或者按照大小划分

如果将日志保存在一个文件中，那么时间一长，或者日志一多，单个日志文件就会很大，既不利于备份，也不利于查看。我们会想到能不能按照时间或者大小对日志文件进行划分呢？答案肯定是可以的，并且还很简单，logging 考虑到了我们这个需求。logging.handlers 文件中提供了 **TimedRotatingFileHandler** 和 **RotatingFileHandler** 类分别可以实现按时间和大小划分。打开这个 handles 文件，可以看到还有其他功能的 Handler 类，它们都继承自基类 **BaseRotatingHandler**。

```
# TimedRotatingFileHandler 类构造函数
def __init__(self, filename, when='h', interval=1, backupCount=0, encoding=None,
delay=False, utc=False, atTime=None):
# RotatingFileHandler 类的构造函数
def __init__(self, filename, mode='a', maxBytes=0, backupCount=0, encoding=None,
delay=False)
```

示例代码如下：

```
# 每隔 1000 Byte 划分一个日志文件，备份文件为 3 个
file_handler = logging.handlers.RotatingFileHandler("test.log", mode="w",
maxBytes=1000, backupCount=3, encoding="utf-8")
复制代码
# 每隔 1小时 划分一个日志文件，interval 是时间间隔，备份文件为 10 个
handler2 = logging.handlers.TimedRotatingFileHandler("test.log", when="H",
interval=1, backupCount=10)
```

Python 官网虽然说 logging 库是线程安全的，但在多进程、多线程、多进程多线程环境中仍然还有值得考虑的问题，比如，如何将日志按照进程（或线程）划分为不同的日志文件，也即一个进程（或线程）对应一个文件。由于本文篇幅有限，故不在这里做详细说明，只是起到引发读者思考的目的，这些问题我会在另一篇文章中讨论。

总结：Python logging 库设计的真的非常灵活，如果有特殊的需要还可以在这个基础的 logging 库上进行改进，创建新的 Handler 类解决实际开发中的问题。