Python之路(第十七篇)logging模块

一、logging模块

（一）、日志相关概念

日志是一种可以追踪某些软件运行时所发生事件的方法。软件开发人员可以向他们的代码中调用日志记录相关的方法来表明发生了某些事情。一个事件可以用一个可包含可选变量数据的消息来描述。此外，事件也有重要性的概念，这个重要性也可以被称为严重性级别（level）。

1、日志的作用

通过log的分析，可以方便用户了解系统或软件、应用的运行情况；如果你的应用log足够丰富，也可以分析以往用户的操作行为、类型喜好、地域分布或其他更多信息；如果一个应用的log同时也分了多个级别，那么可以很轻易地分析得到该应用的健康状况，及时发现问题并快速定位、解决问题，补救损失。

简单来讲就是，我们通过记录和分析日志可以了解一个系统或软件程序运行情况是否正常，也可以在应用程序出现故障时快速定位问题。比如，做运维的同学，在接收到报警或各种问题反馈后，进行问题排查时通常都会先去看各种日志，大部分问题都可以在日志中找到答案。再比如，做开发的同学，可以通过IDE控制台上输出的各种日志进行程序调试。对于运维老司机或者有经验的开发人员，可以快速的通过日志定位到问题的根源。可见，日志的重要性不可小觑。日志的作用可以简单总结为以下3点：

- 程序调试
- 了解软件程序运行情况，是否正常
- 软件程序运行故障分析与问题定位

如果应用的日志信息足够详细和丰富，还可以用来做用户行为分析，如：分析用户的操作行为、类型洗好、地域分布以及其它更多的信息，由此可以实现改进业务、提高商业利益。

2、日志的等级

我们先来思考下下面的两个问题：

- 作为开发人员，在开发一个应用程序时需要什么日志信息？在应用程序正式上线后需要什么日志信息？
- 作为应用运维人员，在部署开发环境时需要什么日志信息？在部署生产环境时需要什么日志信息？

在软件开发阶段或部署开发环境时，为了尽可能详细的查看应用程序的运行状态来保证上线后的稳定性，我们可能需要把该应用程序所有的运行日志全部记录下来进行分析，这是非常耗费机器性能的。当应用程序正式发布或在生产环境部署应用程序时，我们通常只需要记录应用程序的异常信息、错误信息等，这样既可以减小服务器的I/O压力，也可以避免我们在排查故障时被淹没在日志的海洋里。那么，怎样才能在不改动应用程序代码的情况下实现在不同的环境记录不同详细程度的日志呢？这就是日志等级的作用了，我们通过配置文件指定我们需要的日志等级就可以了。

不同的应用程序所定义的日志等级可能会有所差别，分的详细点的会包含以下几个等级：

- DEBUG
- INFO
- NOTICE
- WARNING
- ERROR
- CRITICAL
- ALERT
- EMERGENCY

级别	何时使用
DEBUG	详细信息，典型地调试问题时会感兴趣。 详细的debug信息。
INFO	证明事情按预期工作。 关键事件。
WARNING	表明发生了一些意外，或者不久的将来会发生问题（如'磁盘满了'）。软件还是在正常工作。

导航

博客园
首页
新随笔
联系
订阅XML
管理

公告

昵称：Nicholas-
园龄：1年4个月
粉丝：48
关注：0
+加关注

< 2019年7月 >

日	一	二	三	四	五	六
30	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	1	2	3
4	5	6	7	8	9	10

级别	何时使用
ERROR	由于更严重的问题，软件已不能执行一些功能了。 一般错误消息。
CRITICAL	严重错误，表明软件已不能继续运行了。
NOTICE	不是错误，但是可能需要处理。普通但是重要的事件。
ALERT	需要立即修复，例如系统数据库损坏。
EMERGENCY	紧急情况，系统不可用（例如系统崩溃），一般会通知所有用户。

3、日志字段信息与日志格式

一条日志信息对应的是一个事件的发生，而一个事件通常需要包括以下几个内容：

- 事件发生时间
- 事件发生位置
- 事件的严重程度--日志级别
- 事件内容

上面这些都是一条日志记录中可能包含的字段信息，当然还可以包括一些其他信息，如进程ID、进程名称、线程ID、线程名称等。日志格式就是用来定义一条日志记录中包含那些字段的，且日志格式通常都是可以自定义的。

4、日志功能的实现

几乎所有开发语言都会内置日志相关功能，或者会有比较优秀的第三方库来提供日志操作功能，比如：log4j，log4php等。它们功能强大、使用简单。Python自身也提供了一个用于记录日志的标准库模块--logging。

（二）logging模块

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。

1、 logging模块的日志级别

logging模块默认定义了以下几个日志等级，它允许开发人员自定义其他日志级别，但是这是不被推荐的，尤其是在开发供别人使用的库时，因为这会导致日志级别的混乱。

日志等级（level）	描述
DEBUG	最详细的日志信息，典型应用场景是 问题诊断
INFO	信息详细程度仅次于DEBUG，通常只记录关键节点信息，用于确认一切都是按照我们预期的那样进行工作
WARNING	当某些不期望的事情发生时记录的信息（如，磁盘可用空间较低），但是此时应用程序还是正常运行的
ERROR	由于一个更严重的问题导致某些功能不能正常运行时记录的信息
CRITICAL	当发生严重错误，导致应用程序不能继续运行时记录的信息

开发应用程序或部署开发环境时，可以使用DEBUG或INFO级别的日志获取尽可能详细的日志信息来进行开发或部署调试；

应用上线或部署生产环境时，应该使用WARNING或ERROR或CRITICAL级别的日志来降低机器的I/O压力和提高获取错误日志信息的效率。日志级别的指定通常都是在应用程序的配置文件中进行指定的。

说明：

- 上面列表中的日志等级是从上到下依次升高的，即：DEBUG < INFO < WARNING < ERROR < CRITICAL，而日志的信息量是依次减少的；

2018年7月 (3)
2018年6月 (6)
2018年5月 (10)
2018年4月 (4)
2018年3月 (12)

- 当为某个应用程序指定一个日志级别后，应用程序会记录所有日志级别大于或等于指定日志级别的日志信息，而不是仅仅记录指定级别的日志信息，nginx、php等应用程序以及这里的python的logging模块都是这样的。同样，logging模块也可以指定日志记录器的日志级别，只有级别大于或等于该指定日志级别的日志记录才会被输出，小于该等级的日志记录将会被丢弃。

2、logging模块的使用方式介绍

logging模块提供了两种记录日志的方式：

- 第一种方式是使用logging提供的模块级别的函数
- 第二种方式是使用Logging日志系统的四大组件

其实，logging所提供的模块级别的日志记录函数也是对logging日志系统相关类的封装而已。

logging模块定义的模块级别的常用函数

函数	说明
logging.debug(msg, *args, **kwargs)	创建一条严重级别为DEBUG的日志记录
logging.info(msg, *args, **kwargs)	创建一条严重级别为INFO的日志记录
logging.warning(msg, *args, **kwargs)	创建一条严重级别为WARNING的日志记录
logging.error(msg, *args, **kwargs)	创建一条严重级别为ERROR的日志记录
logging.critical(msg, *args, **kwargs)	创建一条严重级别为CRITICAL的日志记录
logging.log(level, *args, **kwargs)	创建一条严重级别为level的日志记录
logging.basicConfig(**kwargs)	对root logger进行一次性配置

其中`logging.basicConfig(**kwargs)`函数用于指定“要记录的日志级别”、“日志格式”、“日志输出位置”、“日志文件的打开模式”等信息，其他几个都是用于记录各个级别日志的函数。

3、第一种使用方式：简单配置

```
import logging
logging.debug("debug_msg")
logging.info("info_msg")
logging.warning("warning_msg")
logging.error("error_msg")
logging.critical("critical_msg")
```

输出结果

```
WARNING:root:warning_msg
ERROR:root:error_msg
CRITICAL:root:critical_msg
```

默认情况下Python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）

默认输出格式为

```
默认的日志格式为日志级别：Logger名称：用户输出消息
```

这里可以用 logging.basicConfig()函数调整日志级别、输出格式等

简单的例子

```
import logging
```

```
logging.basicConfig(level=logging.DEBUG,
                    format="%(asctime)s %(name)s %(levelname)s %(message)s",
                    datefmt = '%Y-%m-%d  %H:%M:%S %a'    #注意月份和天数不要搞乱了，这里的格式化符
                    )
logging.debug("msg1")
logging.info("msg2")
logging.warning("msg3")
logging.error("msg4")
logging.critical("msg5")
```

输出结果

```
2018-05-09  23:37:49 Wed root DEBUG msg1
2018-05-09  23:37:49 Wed root INFO msg2
2018-05-09  23:37:49 Wed root WARNING msg3
2018-05-09  23:37:49 Wed root ERROR msg4
2018-05-09  23:37:49 Wed root CRITICAL msg5
```

logging.basicConfig()函数包含参数说明

参数名称	描述
filename	指定日志输出目标文件的文件名（可以写文件名也可以写文件的完整的绝对路径，写文件名日志放执行文件目录下，写完整路径按照完整路径生成日志文件），指定该后日志信心就不会被输出到控制台了
filemode	指定日志文件的打开模式，默认为'a'。需要注意的是，该选项要在filename指定时才有效
format	指定日志格式字符串，即指定日志输出时所包含的字段信息以及它们的顺序。logging模块定义的格式字段下面会列出。
datefmt	指定日期/时间格式。需要注意的是，该选项要在format中包含时间字段%(asctime)s时才有效
level	指定日志器的日志级别
stream	指定日志输出目标stream，如sys.stdout、sys.stderr以及网络stream。需要说明的是，stream和filename不能同时提供，否则会引发 ValueError异常
style	Python 3.2中新添加的配置项。指定format格式字符串的风格，可取值为'%'、'{'和'$'，默认为'%'
handlers	Python 3.3中新添加的配置项。该选项如果被指定，它应该是一个创建了多个Handler的可迭代对象，这些handler将会被添加到root logger。需要说明的是：filena stream和handlers这三个配置项只能有一个存在，不能同时出现2个或3个，否则会引发ValueError异常。

logging模块中定义好的可以用于format格式字符串说明

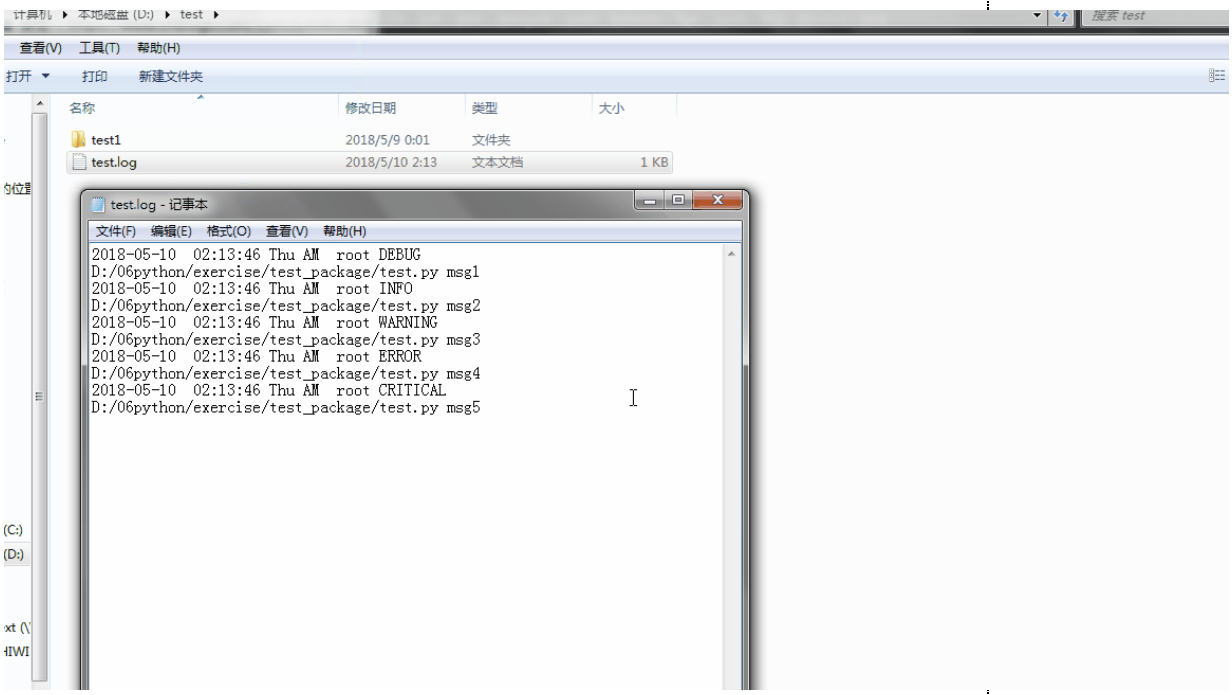
字段/属性名称	使用格式	描述
asctime	%(asctime)s	将日志的时间构造成可读的形式，默认情况下是'2016-02-08 12:00:00,123'精确到毫秒
name	%(name)s	所使用的日志器名称，默认是'root'，因为默认使用的是 rootLogger
filename	%(filename)s	调用日志输出函数的模块的文件名； pathname的文件名部分，包含文件后缀
funcName	%(funcName)s	由哪个function发出的log， 调用日志输出函数的函数名
levelname	%(levelname)s	日志的最终等级（被filter修改后的）

字段/属性名称	使用格式	描述
message	%(message)s	日志信息， 日志记录的文本内容
lineno	%(lineno)d	当前日志的行号， 调用日志输出函数的语句所在的代码行
levelno	%(levelno)s	该日志记录的数字形式的日志级别（10, 20, 30, 40, 50）
pathname	%(pathname)s	完整路径 ，调用日志输出函数的模块的完整路径名，可能没有
process	%(process)s	当前进程， 进程ID。可能没有
processName	%(processName)s	进程名称，Python 3.1新增
thread	%(thread)s	当前线程， 线程ID。可能没有
threadName	%(thread)s	线程名称
module	%(module)s	调用日志输出函数的模块名， filename的名称部分，不包含后缀即不包含文件后缀的文件名
created	%(created)f	当前时间，用UNIX标准的表示时间的浮点数表示； 日志事件发生的时间--时间戳，就是当时调用time.time()函数返回的值
relativeCreated	%(relativeCreated)d	输出日志信息时的，自Logger创建以 来的毫秒数； 日志事件发生的时间相对于logging模块加载时间的相对毫秒数
msecs	%(msecs)d	日志事件发生事件的毫秒部分。logging.basicConfig()中用了参数datefmt，将会去掉asctime中产生的毫秒部分，可以用这个加上

升级版日志例子

```
import logging
LOG_FORMAT = "%(asctime)s %(name)s %(levelname)s %(pathname)s %(message)s "#配置输出日志格式
DATE_FORMAT = '%Y-%m-%d  %H:%M:%S %a ' #配置输出时间的格式，注意月份和天数不要搞乱了
logging.basicConfig(level=logging.DEBUG,
                    format=LOG_FORMAT,
                    datefmt = DATE_FORMAT ,
                    filename=r"d:\test\test.log" #有了filename参数就不会直接输出显示到控制台，而是
                    )
logging.debug("msg1")
logging.info("msg2")
logging.warning("msg3")
logging.error("msg4")
logging.critical("msg5")
```

输出结果



日志在d:\test目录下，日志具体内容

```
2018-05-10 02:13:46 Thu AM root DEBUG D:/06python/exercise/test_package/test.py msg1
2018-05-10 02:13:46 Thu AM root INFO D:/06python/exercise/test_package/test.py msg2
2018-05-10 02:13:46 Thu AM root WARNING D:/06python/exercise/test_package/test.py msg3
2018-05-10 02:13:46 Thu AM root ERROR D:/06python/exercise/test_package/test.py msg4
2018-05-10 02:13:46 Thu AM root CRITICAL D:/06python/exercise/test_package/test.py msg5
```

说明

- logging.basicConfig()函数是一个一次性的简单配置工具使，也就是说只有在第一次调用该函数时会起作用，后续再次调用该函数时完全不会产生任何操作的，多次调用的设置并不是累加操作。
- 日志器（Logger）是有层级关系的，上面调用的logging模块级别的函数所使用的日志器是RootLogger类的实例，其名称为'root'，它是处于日志器层级关系最顶层的日志器，且该实例是以单例模式存在的。
- 如果要记录的日志中包含变量数据，可使用一个格式字符串作为这个事件的描述消息（logging.debug、logging.info等函数的第一个参数），然后将变量数据作为第二个参数*args的值进行传递，

  如:

  ```
  logging.warning('%s is %d years old.', 'Tom', 10)，
  ```

  输出内容为

  ```
  WARNING:root:Tom is 10 years old.
  ```

- logging.debug(), logging.info()等方法的定义中，除了msg和args参数外，还有一个**kwargs参数。它们支持3个关键字参数: exc_info, stack_info, extra，下面对这几个关键字参数作个说明。关于exc_info, stack_info, extra关键词参数的说明：见参考资料1。（了解）

4、第二种使用方式：日志流处理流程

上面简单配置的方法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。

第二种是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）。

logging日志模块四大组件

在介绍logging模块的日志流处理流程之前，我们先来介绍下logging模块的四大组件：

组件名称	对应类名	功能描述
日志器	Logger	提供了应用程序可一直使用的接口
处理器	Handler	将logger创建的日志记录发送到合适的目的输出
过滤器	Filter	提供了更细粒度的控制工具来决定输出哪条日志记录，丢弃哪条日志记录
格式器	Formatter	决定日志记录的最终输出格式

logging模块就是通过这些组件来完成日志处理的，上面所使用的logging模块级别的函数也是通过这些组件对应的类来实现的。

这些组件之间的关系描述：

- 日志器（logger）需要通过处理器（handler）将日志信息输出到目标位置，如：文件、sys.stdout、网络等；
- 不同的处理器（handler）可以将日志输出到不同的位置；
- 日志器（logger）可以设置多个处理器（handler）将同一条日志记录输出到不同的位置；
- 每个处理器（handler）都可以设置自己的过滤器（filter）实现日志过滤，从而只保留感兴趣的日志；
- 每个处理器（handler）都可以设置自己的格式器（formatter）实现同一条日志以不同的格式输出到不同的地方。

简单点说就是：日志器（logger）是入口，真正干活儿的是处理器（handler），处理器（handler）还可以通过过滤器（filter）和格式器（formatter）对要输出的日志内容做过滤和格式化等处理操作。

logging日志模块相关类及其常用方法介绍

与logging四大组件相关的类：Logger, Handler, Filter, Formatter。

Logger类

Logger对象有3个任务要做：

- 1）向应用程序代码暴露几个方法，使应用程序可以在运行时记录日志消息；
- 2）基于日志严重等级（默认的过滤设施）或filter对象来决定要对哪些日志进行后续处理；
- 3）将日志消息传送给所有感兴趣的日志handlers。

Logger对象最常用的方法分为两类：配置方法 和 消息发送方法

最常用的配置方法如下：

方法	描述
Logger.setLevel()	设置日志器将会处理的日志消息的最低严重级别
Logger.addHandler() 和 Logger.removeHandler()	为该logger对象添加 和 移除一个handler对象
Logger.addFilter() 和 Logger.removeFilter()	为该logger对象添加 和 移除一个filter对象

logger对象配置完成后，可以使用下面的方法来创建日志记录：

方法	描述
Logger.debug(), Logger.info(), Logger.warning(), Logger.error(), Logger.critical()	创建一个与它们的方法名对应等级的日志记录
Logger.exception()	创建一个类似于Logger.error()的日志消息

方法	描述
Logger.log()	需要获取一个明确的日志level参数来创建一个日志记录

一个Logger对象呢？一种方式是通过Logger类的实例化方法创建一个Logger类的实例，但是我们通常都是用第二种方式--logging.getLogger()方法。

logging.getLogger()方法有一个可选参数name，该参数表示将要返回的日志器的名称标识，如果不提供该参数，则其值为'root'。若以相同的name参数值多次调用getLogger()方法，将会返回指向同一个logger对象的引用。

多次使用注意不能创建多个logger,否则会出现重复输出日志现象。

关于logger的层级结构与有效等级的说明：

- logger的名称是一个以'.'分割的层级结构，每个'.'后面的logger都是'.'前面的logger的children，例如，有一个名称为 foo 的logger，其它名称分别为 foo.bar, foo.bar.baz 和 foo.bam都是 foo 的后代。
- logger有一个"有效等级（effective level）"的概念。如果一个logger上没有被明确设置一个level，那么该logger就是使用它parent的level;如果它的parent也没有明确设置level则继续向上查找parent的parent的有效level，依次类推，直到找到个一个明确设置了level的祖先为止。需要说明的是，root logger总是会有一个明确的level设置（默认为 WARNING）。当决定是否去处理一个已发生的事件时，logger的有效等级将会被用来决定是否将该事件传递给该logger的handlers进行处理。
- child loggers在完成对日志消息的处理后，默认会将日志消息传递给与它们的祖先loggers相关的handlers。因此，我们不必为一个应用程序中所使用的所有loggers定义和配置handlers，只需要为一个顶层的logger配置handlers，然后按照需要创建child loggers就可足够了。我们也可以通过将一个logger的propagate属性设置为False来关闭这种传递机制。

Handler类

Handler对象的作用是（基于日志消息的level）将消息分发到handler指定的位置（文件、网络、邮件等）。Logger对象可以通过addHandler()方法为自己添加0个或者更多个handler对象。比如，一个应用程序可能想要实现以下几个日志需求：

- 1）把所有日志都发送到一个日志文件中；
- 2）把所有严重级别大于等于error的日志发送到stdout（标准输出）；
- 3）把所有严重级别为critical的日志发送到一个email邮件地址。这种场景就需要3个不同的handlers，每个handler复杂发送一个特定严重级别的日志到一个特定的位置。

```
Handler.setLevel(lel):指定被处理的信息级别，低于lel级别的信息将被忽略
Handler.setFormatter()：给这个handler选择一个格式
Handler.addFilter(filt)、Handler.removeFilter(filt)：新增或删除一个filter对象
```

需要说明的是，应用程序代码不应该直接实例化和使用Handler实例。因为Handler是一个基类，它只定义了素有handlers都应该有的接口，同时提供了一些子类可以直接使用或覆盖的默认行为。下面是一些常用的Handler：

Handler	描述
logging.StreamHandler	将日志消息发送到输出到Stream，如std.out, std.err或任何file-like对象。
logging.FileHandler	将日志消息发送到磁盘文件，默认情况下文件大小会无限增长
logging.handlers.RotatingFileHandler	将日志消息发送到磁盘文件，并支持日志文件按大小切割
logging.hanlders.TimedRotatingFileHandler	将日志消息发送到磁盘文件，并支持日志文件按时间切割
logging.handlers.HTTPHandler	将日志消息以GET或POST的方式发送给一个HTTP服务器
logging.handlers.SMTPHandler	将日志消息发送给一个指定的email地址
logging.NullHandler	该Handler实例会忽略error messages，通常被想使用logging的library开发者使用来避免'No handlers could be found for logger XXX'信息的出现。

Formater类

Formater对象用于配置日志信息的最终顺序、结构和内容。与logging.Handler基类不同的是，应用代码可以直接实例化Formatter类。另外，如果你的应用程序需要一些特殊的处理行为，也可以实现一个Formatter的子类来完成。

Formatter类的构造方法定义如下：

```
logging.Formatter.__init__(fmt=None, datefmt=None, style='%')
```

可见，该构造方法接收3个可选参数：

- fmt：指定消息格式化字符串，如果不指定该参数则默认使用message的原始值
- datefmt：指定日期格式字符串，如果不指定该参数则默认使用"%Y-%m-%d %H:%M:%S"
- style：Python 3.2新增的参数，可取值为 '%', '{'和 '$'，如果不指定该参数则默认使用'%'

一般直接用logging.Formatter（fmt, datefmt）

Filter类（暂时了解）

Filter可以被Handler和Logger用来做比level更细粒度的、更复杂的过滤功能。Filter是一个过滤器基类，它只允许某个logger层级下的日志事件通过过滤。该类定义如下：

```
class logging.Filter(name='')
    filter(record)
```

比如，一个filter实例化时传递的name参数值为'A.B'，那么该filter实例将只允许名称为类似如下规则的loggers产生的日志记录通过过滤：'A.B'，'A.B.C'，'A.B.C.D'，'A.B.D'，而名称为'A.BB', 'B.A.B'的loggers产生的日志则会被过滤掉。如果name的值为空字符串，则允许所有的日志事件通过过滤。

filter方法用于具体控制传递的record记录是否能通过过滤，如果该方法返回值为0表示不能通过过滤，返回值为非0表示可以通过过滤。

> **说明：**
>
> - 如果有需要，也可以在filter(record)方法内部改变该record，比如添加、删除或修改一些属性。
> - 我们还可以通过filter做一些统计工作，比如可以计算下被一个特殊的logger或handler所处理的record数量等。

日志流处理简要流程

1、创建一个logger

2、设置下logger的日志的等级

3、创建合适的Handler(FileHandler要有路径)

4、设置下每个Handler的日志等级

5、创建下日志的格式

6、向Handler中添加上面创建的格式

7、将上面创建的Handler添加到logger中

8、打印输出logger.debug\logger.info\logger.warning\logger.error\logger.critical

例子

```
import logging

#创建logger，如果参数为空则返回root logger
logger = logging.getLogger("nick")
logger.setLevel(logging.DEBUG)  #设置logger日志等级

```

```
#创建handler
fh = logging.FileHandler("test.log",encoding="utf-8")
ch = logging.StreamHandler()

#设置输出日志格式
formatter = logging.Formatter(
    fmt="%(asctime)s %(name)s %(filename)s %(message)s",
    datefmt="%Y/%m/%d %X"
    )

#注意 logging.Formatter的大小写

#为handler指定输出格式，注意大小写
fh.setFormatter(formatter)
ch.setFormatter(formatter)

#为logger添加的日志处理器
logger.addHandler(fh)
logger.addHandler(ch)

#输出不同级别的log
logger.warning("泰拳警告")
logger.info("提示")
logger.error("错误")
```

python logging 重复写日志问题

用Python的logging模块记录日志时，可能会遇到重复记录日志的问题，第一条记录写一次，第二条记录写两次，第三条记录写三次

原因：没有移除handler 解决：在日志记录完之后removeHandler

例子

```
def log(msg):
    #创建logger，如果参数为空则返回root logger
    logger = logging.getLogger("nick")
    logger.setLevel(logging.DEBUG)  #设置logger日志等级

    #创建handler
    fh = logging.FileHandler("test.log",encoding="utf-8")
    ch = logging.StreamHandler()

    #设置输出日志格式
    formatter = logging.Formatter(
        fmt="%(asctime)s %(name)s %(filename)s %(message)s",
        datefmt="%Y/%m/%d %X"
        )

    #为handler指定输出格式
    fh.setFormatter(formatter)
    ch.setFormatter(formatter)

    #为logger添加的日志处理器
    logger.addHandler(fh)
    logger.addHandler(ch)

    # 输出不同级别的log
    logger.info(msg)

log("泰拳警告")
log("提示")
log("错误")
```

输出结果

```
2018/05/10 20:06:18 nick test.py 泰拳警告
2018/05/10 20:06:18 nick test.py 提示
2018/05/10 20:06:18 nick test.py 提示
2018/05/10 20:06:18 nick test.py 错误
2018/05/10 20:06:18 nick test.py 错误
2018/05/10 20:06:18 nick test.py 错误
```

分析：可以看到输出结果有重复打印

原因：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。

解决方案1

添加removeHandler语句

```
import logging
def log(msg):
    #创建logger，如果参数为空则返回root logger
    logger = logging.getLogger("nick")
    logger.setLevel(logging.DEBUG)  #设置logger日志等级

    #创建handler
    fh = logging.FileHandler("test.log",encoding="utf-8")
    ch = logging.StreamHandler()

    #设置输出日志格式
    formatter = logging.Formatter(
        fmt="%(asctime)s %(name)s %(filename)s %(message)s",
        datefmt="%Y/%m/%d %X"
        )

    #为handler指定输出格式
    fh.setFormatter(formatter)
    ch.setFormatter(formatter)

    #为logger添加的日志处理器
    logger.addHandler(fh)
    logger.addHandler(ch)

    # 输出不同级别的log
    logger.info(msg)

    #解决方案1，添加removeHandler语句，每次用完之后移除Handler
    logger.removeHandler(fh)
    logger.removeHandler(ch)


log("泰拳警告")
log("提示")
log("错误")
```

解决方案2

在log方法里做判断，如果这个logger已有handler，则不再添加handler。

```
import logging
def log(msg):
    #创建logger，如果参数为空则返回root logger
    logger = logging.getLogger("nick")
    logger.setLevel(logging.DEBUG)  #设置logger日志等级

    #解决方案2：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:
        #创建handler
        fh = logging.FileHandler("test.log",encoding="utf-8")
        ch = logging.StreamHandler()
```

```
        #设置输出日志格式
        formatter = logging.Formatter(
            fmt="%(asctime)s %(name)s %(filename)s %(message)s",
            datefmt="%Y/%m/%d %X"
            )

        #为handler指定输出格式
        fh.setFormatter(formatter)
        ch.setFormatter(formatter)

        #为logger添加的日志处理器
        logger.addHandler(fh)
        logger.addHandler(ch)

    # 输出不同级别的log
    logger.info(msg)


log("泰拳警告")
log("提示")
log("错误")
```

logger调用方式例子

```
import logging
def log():
    #创建logger，如果参数为空则返回root logger
    logger = logging.getLogger("nick")
    logger.setLevel(logging.DEBUG)  #设置logger日志等级

    #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:
        #创建handler
        fh = logging.FileHandler("test.log",encoding="utf-8")
        ch = logging.StreamHandler()

        #设置输出日志格式
        formatter = logging.Formatter(
            fmt="%(asctime)s %(name)s %(filename)s %(message)s",
            datefmt="%Y/%m/%d %X"
            )

        #为handler指定输出格式
        fh.setFormatter(formatter)
        ch.setFormatter(formatter)

        #为logger添加的日志处理器
        logger.addHandler(fh)
        logger.addHandler(ch)

    return logger #直接返回logger

logger = log()
logger.warning("泰拳警告")
logger.info("提示")
logger.error("错误")
logger.debug("查错")
```

参考链接

[1]https://www.cnblogs.com/yyds/p/6901864.html

[2]https://blog.csdn.net/huilan_same/article/details/51858817

分类: Python

Nicholas-
关注 - 0
粉丝 - 48
+加关注

4 0

« 上一篇：Python之路(第十五篇)sys模块、json模块、pickle模块、shelve模块
» 下一篇：Python之路(第十六篇)xml模块、datetime模块

posted on 2018-05-10 20:51 Nicholas- 阅读(11500) 评论(0) 编辑 收藏

刷新评论 刷新页面 返回顶部

相关博文：
· 模块之logging，shelve，sys模块
· python-25logging日志模块之一
· python标准模块--logging
· logging模块
· Python logging模块

最新新闻：
· 今年人类已提前耗尽自然资源年度"预算"
· 微信支付将推全新刷脸支付产品：青蛙Pro
· 魅族黄章深夜发文：雷军做小米不是因为我"不舍股份"
· 舔一口冰淇淋后突然头很痛？喝口热水压压惊就好了
· 小红书的"种草"困境
» 更多新闻...