日志模块----logging模块

2019-08-14 23:07 风e逸 阅读(97) 评论(0) 编辑 收藏 举报

在**logging.basicConfig()**函数中可通过具体参数来更改logging模块默认行为，可用参数有:

filename：用指定的文件名创建FiledHandler（后边会具体讲解handler的概念），这样日志会被存储在指定的文件中。

filemode：文件打开方式，在指定了filename时使用这个参数，默认值为“a”还可指定为“w”。

format：指定handler使用的日志显示格式。

datefmt：指定日期时间格式。

level：设置rootlogger（后边会讲解具体概念）的日志级别

stream：用指定的stream创建StreamHandler。可以指定输出到sys.stderr,sys.stdout或者文件(f=open('test.log','w'))，默认为sys.stderr。若同时列出了filename和stream两个参数，则stream参数会被忽略。

```
import logging

logging.basicConfig(
    level=logging.DEBUG, # 修改默认，让loggin从debug等级开始显示
    filename="logging.log" ,# 修改储存位置，储存到文件logging.log中
    filemode="w" # 默认是添加模式，更改为替换模式
)

# logging下的五个等级，相当于预警等级，默认从warning开始显示
logging.debug('b')
logging.info('a')
logging.warning('s')
logging.error('ss')
logging.critical('ssss')

输出结果:
创建了一个文件，名为logging.log,内容为
DEBUG:root:b
INFO:root:a
WARNING:root:s
ERROR:root:ss
```

```
CRITICAL:root:sss
```

format参数中的格式化串：

%(**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**

——

logger模块

Logger是一个树形层级结构，输出信息之前都要获得一个Logger（如果没有显示的获取则自动创建并使用root Logger）

```
import logging
logger = logging.getLogger() # 设置logger为可得到其他功能的函数，默认为root用户
# logger.setLevel("DEBUG") # 设置从debug级别开始显示
logger.setLevel(logging.DEBUG) # 这种写法与上一种功能相同

fm = logging.FileHandler("logger.log") # fm表示将日志内容输出到文件
cm = logging.StreamHandler() # cm 表示将日志输出到屏幕
```

```
fh = logging.Formatter("%(asctime)s %(message)s") # 设置输出格式为fh，输出时间和内容
fm.setFormatter(fh) # 将格式应用于fm
cm.setFormatter(fh) # 将格式应用于cm

logger.addHandler(fm) # 将fm的功能附给logger
logger.addHandler(cm) # 将cm的功能附给logger

logger.debug("b")
logger.info("a")
logger.warning("s")
logger.error("ss")
logger.critical("sss")
```

如果我们再创建两个logger对象：

```
##################################################
logger1 = logging.getLogger('mylogger')
logger1.setLevel(logging.DEBUG)

logger2 = logging.getLogger('mylogger')
logger2.setLevel(logging.INFO)
<br>fh = logging.FileHandler("logger_tree_test")<br>ch = logging.StreamHandler()<br>
logger1.addHandler(fh)
logger1.addHandler(ch)

logger2.addHandler(fh)
logger2.addHandler(ch)

logger1.debug('logger1 debug message')
logger1.info('logger1 info message')
logger1.warning('logger1 warning message')
logger1.error('logger1 error message')
logger1.critical('logger1 critical message')

logger2.debug('logger2 debug message')
logger2.info('logger2 info message')
```

```
logger2.warning('logger2 warning message')
logger2.error('logger2 error message')
logger2.critical('logger2 critical message')<br>输出结果：
```

logger1 info message

logger1 warning message

logger1 error message

logger1 critical message

logger2 info message

logger2 warning message

logger2 error message

logger2 critical message

logger1和logger2对应的是同一个Logger实例，只要logging.getLogger（name）中名称参数name相同则返回的Logger实例就是同一个，且仅有一个，也即name与Logger实例一一对应。在logger2实例中通过logger2.setLevel(logging.INFO)设置mylogger的日志级别为logging.INFO，所以最后logger1的输出遵从了后来设置的日志级别。

如果父辈logger与子辈logger1,logger2同时存在并运行，那么子辈每个命令将输出两次，若还有孙辈，则孙辈每个命令输出三次。这是因为我们通过logger = logging.getLogger()显示的创建了root Logger，而logger1 = logging.getLogger('mylogger')创建了root Logger的孩子(root.)mylogger,logger2同样。而孩子,孙子，重孙……既会将消息分发给他的handler进行处理也会传递给所有的祖先Logger处理。

——

Introspect Python logging with logging_tree

Date: 13 April 2012
Tags: python, computing

It is frustrating that Python's `logging` module cannot display the tangled tree of configured loggers that often result from combining your own application code with several libraries and frameworks. So I have released a new Python package named logging_tree, which I announced last month during the PyCon 2012 closing lightning talks. My package displays the current `logging` tree to help debugging, and its output looks something like this:

```
<--""
   Level WARNING
   Handler Stream <open file '<stderr>', mode 'w' at ...>
   |
   o<--[cherrypy]
       |
       o<--"cherrypy.access"
       |   Level INFO
       |   Handler Stream <open file '<stdout>', mode 'w' ...>
       |
       o<--"cherrypy.error"
           Level INFO
           Handler Stream <open file '<stderr>', mode 'w' ...>
```

The configuration shown by this tree, it turns out, causes a bug. This diagram helped me fix a real-life application for Atlanta startup Rover Apps, who generously let me open-source `logging_tree` after I wrote the first version while helping them fix this bug.

In this post I am going to reproduce the problem using a simple 10-line CherryPy application, and then show how I used this `logging_tree` diagram to craft a solution. But, first, you need to know three things about the Python `logging` module — and I would like to thank Marius Gedminas for his recent post about logging levels that helped me correct something I had misunderstood.

- When you call a logger's `log()` method, or an equivalent helper like `error()` or `debug()`, your message has only one opportunity to be discarded: it is compared against that logger's `level` attribute and thrown away if it is not at least that important. But if it passes this one test, then the message will be submitted not only to the logger's own handlers, but also to the handlers on all of the logger's parents — so a message accepted by logger `'a.b.c'` will also be submitted to `'a.b'`, `'a'`, and the root logger `''` (whose name is the empty string). The `level` attribute of a parent logger *is completely ignored* — it applies only to messages submitted directly to that logger!
- Once a message passes this test against `logger.level` and is accepted, the only thing that can stop it from being submitted to the handlers of every parent logger is for it to encounter a false `propagate` attribute along the way. After a message is submitted to a logger's handlers, the logger's `propagate` attribute is consulted to see whether the message should jump to the parent logger. The first false `propagate` encountered on the way up the tree kills the message, and no further propagation takes place.
- Confusingly, each handler can also choose to do its own individual filtering by a `level` to decide which messages get output by that handler. Handlers can be configured with complex filters to determine which messages they are willing to output, and can even hand messages off to further handlers.

Some libraries also define custom filters and handlers. While `logging_tree` will at least print their names, it probably will not know anything else about them, so you will have to read their source code to learn more about how they work and behave.

The problem

We wanted to add some logging to our CherryPy application, so — as a simple first step — we grabbed the root logger `''` and tried writing a message:

```
import cherrypy
import logging

log = logging.getLogger('')

class HelloWorld:
    def index(self):
        log.error('Test message')
        return 'Hello world!'
    index.exposed = True

if __name__ == '__main__':
    cherrypy.quickstart(HelloWorld())
```

The result was disappointing. The site displayed `Hello world!` in the browser and produced a standard Apache log message, but our own log message was nowhere to be seen. I knew enough about `logging` to know that a root handler was necessary if I wanted output, so I tried adding one:

```
...
if __name__ == '__main__':
    log.addHandler(logging.StreamHandler())
    cherrypy.quickstart(HelloWorld())
```

The result was rather bizarre. While our `Test message` appeared in the output, the server now produced *two* copies of every CherryPy log message.

It was at this point that, tired of trying to guess how the logging tree looked by searching our entire code base for calls to the `logging` package — and searching the code of our third-party libraries, like CherryPy — I instead wrote the first version of `logging_tree()` so that I could really see what was going on. You can install it quite simply, and it supports any version of Python from 2.3 through 3.2:

```
pip install logging_tree
```

Invoking it from your application can be as simple as:

```
import logging_tree
logging_tree.printout()
```

These two lines, added to my `index()` method, printed the entire logger configuration to my terminal the next time I reloaded the page.

Interpreting a tree

This CherryPy application actually produces a more complicated logging situation than in the simplified tree used for illustration at the top of this post. Here is what `logging_tree` really prints out — in a few places I have used ellipses to keep lines within the margins of my blog, but this time no lines are omitted:

```
<--""
   Level WARNING
   Handler Stream <open file '<stderr>', mode 'w' at ...>
   |
   o<--[cherrypy]
       |
       o<--"cherrypy.access"
       |   Level INFO
       |   Handler <cherrypy._cplogging.NullHandler object at 0x...>
       |   Handler Stream <open file '<stdout>', mode 'w' at ...>
       |   |
       |   o<--"cherrypy.access.166457196"
       |       Level INFO
       |       Handler <cherrypy._cplogging.NullHandler object at 0x...>
       |
       o<--"cherrypy.error"
```

```
            Level INFO
            Handler <cherrypy._cplogging.NullHandler object at 0x...>
            Handler Stream <open file '<stdout>', mode 'w' at ...>
            |
            o<--"cherrypy.error.166457196"
                Level INFO
                Handler <cherrypy._cplogging.NullHandler object at 0x...>
```

Loggers that have been created through actual calls to `getLogger()` are displayed with their names in double quotes. When a logger only exists by implication, but has never actually been named in a `getLogger()` call — like the `[cherrypy]` node — then its name is shown in square brackets.

Each logger displays its own `Level` that, as discussed above, is only consulted when a message is submitted directly to that logger using one of its methods like `log()` or `error()`.

Propagation is turned on for all of these loggers, as shown by the `<--` arrows that connect each logger to its parent.

You can see that this tree includes both built-in handlers and also some custom ones defined in the CherryPy framework. The `logging_tree` package tries to introspect the built-in handlers to give you more information about them — here, it displays the particular output files to which the stream handlers will be printing — but for the CherryPy loggers it has no choice but to simply print their `repr()` and hope that you can make sense of them.

Solving the problem

Thanks to this diagram, the problem is now clear: because propagation is turned on, CherryPy logging messages get printed by their own handlers and *also* by the new handler we have installed at the root. You can see this by imagining a new `cherrypy.access` error message and following the propagation arrows that will take it from its logger-of-origin up to the root, where our own handler is installed.

We can see, in fact, that CherryPy creates several loggers to receive its Apache-style logging messages, and goes ahead and suits those loggers up with handlers that write the messages to the correct files. Since I ran this application in debug mode from the command line, these handlers are directed at `<stdout>` instead of actual log files.

The solution to my problem was to simply turn off propagation, since CherryPy's handlers were already taking care of its messages:

```
logging.getLogger('cherrypy').propagate = False
```

The `logging_tree` package makes it very clear when propagation has been turned off, both by removing the `<--` arrow from next to the logger's name, and also with a `Propagate OFF` message. So this is how the tree looked following the fix:

```
<--""
   Level WARNING
   Handler Stream <open file '<stderr>', mode 'w' at ...>
   |
   o   "cherrypy"
       Propagate OFF
       |
       ...
```

I hope that `logging_tree` proves useful for many more Python programmers as we all wrestle with logging misbehavior! The package is available both on the Python Package Index and is also available as a project on GitHub if you want to contribute.