# 日志专题手册

**作者:** Vinay Sajip <vinay_sajip at red-dove dot com>

本页面包含多个与日志相关的专题，历史证明它们是很有用的。教程和参考信息的链接另见 其他资源。

## 在多模块中使用日志

无论对 `logging.getLogger('someLogger')` 进行多少次调用，都会返回同一个 logger 对象的引用。不仅在同一个模块内如此，只要是在同一个 Python 解释器进程中，跨模块调用也是一样。同样是引用同一个对象，应用程序也可以在一个模块中定义和配置一个父 logger，而在另一个单独的模块中创建（但不配置）子 logger，对于子 logger 的所有调用都会传给父 logger。以下是主模块：

```
import logging
import auxiliary_module

# 创建 'spam_application' 日志记录器
logger = logging.getLogger('spam_application')
logger.setLevel(logging.DEBUG)
# 创建可记录调试消息的文件处理器
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# 创建具有更高日志层级的控制台处理器
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# 创建格式化器并将其添加到处理器
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)
fh.setFormatter(formatter)
ch.setFormatter(formatter)
# 将处理器添加到日志记录器
logger.addHandler(fh)
logger.addHandler(ch)

logger.info('creating an instance of auxiliary_module.Auxiliary')
a = auxiliary_module.Auxiliary()
logger.info('created an instance of auxiliary_module.Auxiliary')
logger.info('calling auxiliary_module.Auxiliary.do_something')
a.do_something()
logger.info('finished auxiliary_module.Auxiliary.do_something')
logger.info('calling auxiliary_module.some_function()')
auxiliary_module.some_function()
logger.info('done with auxiliary_module.some_function()')
```

以下是辅助模块：

```
import logging

# 创建日志记录器
module_logger = logging.getLogger('spam_application.auxiliary')
```

```
class Auxiliary:
    def __init__(self):
        self.logger = logging.getLogger('spam_application.auxiliary.Auxiliary')
        self.logger.info('creating an instance of Auxiliary')

    def do_something(self):
        self.logger.info('doing something')
        a = 1 + 1
        self.logger.info('done doing something')

def some_function():
    module_logger.info('received a call to "some_function"')
```

输出结果会像这样:

```
2005-03-23 23:47:11,663 - spam_application - INFO -
   creating an instance of auxiliary_module.Auxiliary
2005-03-23 23:47:11,665 - spam_application.auxiliary.Auxiliary - INFO -
   creating an instance of Auxiliary
2005-03-23 23:47:11,665 - spam_application - INFO -
   created an instance of auxiliary_module.Auxiliary
2005-03-23 23:47:11,668 - spam_application - INFO -
   calling auxiliary_module.Auxiliary.do_something
2005-03-23 23:47:11,668 - spam_application.auxiliary.Auxiliary - INFO -
   doing something
2005-03-23 23:47:11,669 - spam_application.auxiliary.Auxiliary - INFO -
   done doing something
2005-03-23 23:47:11,670 - spam_application - INFO -
   finished auxiliary_module.Auxiliary.do_something
2005-03-23 23:47:11,671 - spam_application - INFO -
   calling auxiliary_module.some_function()
2005-03-23 23:47:11,672 - spam_application.auxiliary - INFO -
   received a call to 'some_function'
2005-03-23 23:47:11,673 - spam_application - INFO -
   done with auxiliary_module.some_function()
```

## 在多个线程中记录日志

多线程记录日志并不需要特殊处理，以下示例演示了在主线程（起始线程）和其他线程中记录日志的过程:

```
import logging
import threading
import time

def worker(arg):
    while not arg['stop']:
        logging.debug('Hi from myfunc')
        time.sleep(0.5)

def main():
    logging.basicConfig(level=logging.DEBUG, format='%(relativeCreated)6d %(thread
    info = {'stop': False}
    thread = threading.Thread(target=worker, args=(info,))
    thread.start()
    while True:
```

```
        try:
            logging.debug('Hello from main')
            time.sleep(0.75)
        except KeyboardInterrupt:
            info['stop'] = True
            break
    thread.join()

if __name__ == '__main__':
    main()
```

脚本会运行输出类似下面的内容:

```
   0 Thread-1 Hi from myfunc
   3 MainThread Hello from main
 505 Thread-1 Hi from myfunc
 755 MainThread Hello from main
1007 Thread-1 Hi from myfunc
1507 MainThread Hello from main
1508 Thread-1 Hi from myfunc
2010 Thread-1 Hi from myfunc
2258 MainThread Hello from main
2512 Thread-1 Hi from myfunc
3009 MainThread Hello from main
3013 Thread-1 Hi from myfunc
3515 Thread-1 Hi from myfunc
3761 MainThread Hello from main
4017 Thread-1 Hi from myfunc
4513 MainThread Hello from main
4518 Thread-1 Hi from myfunc
```

以上如期显示了不同线程的日志是交替输出的。当然更多的线程也会如此。

## 多个 handler 和多种 formatter

日志是个普通的 Python 对象。 `addHandler()` 方法可加入不限数量的日志 handler。有时候，应用程序需把严重错误信息记入文本文件，而将一般错误或其他级别的信息输出到控制台。若要进行这样的设定，只需多配置几个日志 handler 即可，应用程序的日志调用代码可以保持不变。下面对之前的分模块日志示例略做修改:

```
import logging

logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)
# 创建可记录调试消息的文件处理器
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# 创建具有更高日志层级的控制台处理器
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# 创建格式化器并将其添加到处理器
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)
ch.setFormatter(formatter)
fh.setFormatter(formatter)
# 将处理器添加到日志记录器
logger.addHandler(ch)
```

```
logger.addHandler(fh)

# '应用程序' 代码
logger.debug('debug message')
logger.info('info message')
logger.warning('warn message')
logger.error('error message')
logger.critical('critical message')
```

需要注意的是，“应用程序”内的代码并不关心是否存在多个日志 handler。示例中所做的改变，只是新加入并配置了一个名为 *fh* 的 handler。

在编写和测试应用程序时，若能创建日志 handler 对不同严重级别的日志信息进行过滤，这将十分有用。调试时无需用多条 `print` 语句，而是采用 `logger.debug`：print 语句以后还得注释或删掉，而 logger.debug 语句可以原样留在源码中保持静默。当需要再次调试时，只要改变日志对象或 handler 的严重级别即可。

## 在多个地方记录日志

假定要根据不同的情况将日志以不同的格式写入控制台和文件。比如把 DEBUG 以上级别的日志信息写于文件，并且把 INFO 以上的日志信息输出到控制台。再假设日志文件需要包含时间戳，控制台信息则不需要。以下演示了做法：

```
import logging

# 设置日志记录到文件 —— 参阅前一节了解详情
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s %(name)-12s %(levelname)-8s %(message)s',
                    datefmt='%m-%d %H:%M',
                    filename='/tmp/myapp.log',
                    filemode='w')
# 定义一个将 INFO 或更高层级消息写到 sys.stderr 的处理器
console = logging.StreamHandler()
console.setLevel(logging.INFO)
# 设置一个适用于控制台的更简单格式
formatter = logging.Formatter('%(name)-12s: %(levelname)-8s %(message)s')
# 告诉处理器使用此格式
console.setFormatter(formatter)
# 将处理器添加到根日志记录器
logging.getLogger('').addHandler(console)

# 现在我们可以写入根记录器或任何其他记录器。 首先是根记录器...
logging.info('Jackdaws love my big sphinx of quartz.')

# 现在，定义几个可以代表你的应用程序中不同组成部分的
# 其他日志记录器

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')

logger1.debug('Quick zephyrs blow, vexing daft Jim.')
logger1.info('How quickly daft jumping zebras vex.')
logger2.warning('Jail zesty vixen who grabbed pay from quack.')
logger2.error('The five boxing wizards jump quickly.')
```

当运行后，你会看到控制台如下所示

```
root        : INFO     Jackdaws love my big sphinx of quartz.
myapp.area1 : INFO     How quickly daft jumping zebras vex.
myapp.area2 : WARNING  Jail zesty vixen who grabbed pay from quack.
myapp.area2 : ERROR    The five boxing wizards jump quickly.
```

而日志文件将如下所示：

```
10-22 22:19 root         INFO     Jackdaws love my big sphinx of quartz.
10-22 22:19 myapp.area1  DEBUG    Quick zephyrs blow, vexing daft Jim.
10-22 22:19 myapp.area1  INFO     How quickly daft jumping zebras vex.
10-22 22:19 myapp.area2  WARNING  Jail zesty vixen who grabbed pay from quack.
10-22 22:19 myapp.area2  ERROR    The five boxing wizards jump quickly.
```

如您所见，DEBUG 级别的日志信息只出现在了文件中，而其他信息则两个地方都会输出。

上述示例只用到了控制台和文件 handler，当然还可以自由组合任意数量的日志 handler。

请注意上面选择的日志文件名 `/tmp/myapp.log` 表示在 POSIX 系统上使用临时文件的标准位置。 在 Windows 上，你可能需要为日志选择不同的目录名称 —— 只要确保该目录存在并且你有在其中创建和更新文件的权限。

## 自定义处理级别

有时，你想要做的可能略微不同于处理器中标准的级别处理方式，即某个界限以上的所有级别都会被处理器所处理。 要做到这一点，你需要使用过滤器。 让我们来看一个假设你想要执行如下安排的场景:

- 将严重级别为 `INFO` 和 `WARNING` 的消息发送到 `sys.stdout`
- 将严重级别为 `ERROR` 及以上的消息发送到 `sys.stderr`
- 将严重级别为 `DEBUG` 及以上的消息发送到文件 `app.log`

假定你使用以下 JSON 来配置日志记录:

```
{
    "version": 1,
    "disable_existing_loggers": false,
    "formatters": {
        "simple": {
            "format": "%(levelname)-8s - %(message)s"
        }
    },
    "handlers": {
        "stdout": {
            "class": "logging.StreamHandler",
            "level": "INFO",
            "formatter": "simple",
            "stream": "ext://sys.stdout"
        },
        "stderr": {
            "class": "logging.StreamHandler",
            "level": "ERROR",
```

```
            "formatter": "simple",
            "stream": "ext://sys.stderr"
        },
        "file": {
            "class": "logging.FileHandler",
            "formatter": "simple",
            "filename": "app.log",
            "mode": "w"
        }
    },
    "root": {
        "level": "DEBUG",
        "handlers": [
            "stderr",
            "stdout",
            "file"
        ]
    }
}
```

这个配置 *几乎* 能做到我们想要的，但是除了 `sys.stdout` 在 `INFO` 和 `WARNING` 消息之外会只显示严重程度 `ERROR` 及以上的消息。 为了防止这种情况，我们可以设置一个排除掉这些消息的过滤器并将其添加到相应的处理器中。 这可以通过添加一个平行于 `formatters` 和 `handlers` 的 `filters` 节来配置：

```
{
    "filters": {
        "warnings_and_below": {
            "()" : "__main__.filter_maker",
            "level": "WARNING"
        }
    }
}
```

并修改 `stdout` 处理器上的节来添加它:

```
{
    "stdout": {
        "class": "logging.StreamHandler",
        "level": "INFO",
        "formatter": "simple",
        "stream": "ext://sys.stdout",
        "filters": ["warnings_and_below"]
    }
}
```

过滤器就是一个函数，因此我们可以定义 `filter_maker` (工厂函数) 如下:

```
def filter_maker(level):
    level = getattr(logging, level)

    def filter(record):
        return record.levelno <= level

    return filter
```

此函数将传入的字符串参数转换为数字级别，并返回一个仅在传入等于或低于指定数字级别的级别时返回 `True` 的函数。 请注意在这个示例中我是将 `filter_maker` 定义在一个从命令行运行的测试脚本 `main.py` 中，因此其所属模块将为 `__main__` —— 即在过滤器配置中写作 `__main__.filter_maker`。 如果你在不同的模块中定义它则需要加以修改。

在添加该过滤器后，我们就可以运行 `main.py`，完整代码如下:

```
import json
import logging
import logging.config

CONFIG = '''
{
    "version": 1,
    "disable_existing_loggers": false,
    "formatters": {
        "simple": {
            "format": "%(levelname)-8s - %(message)s"
        }
    },
    "filters": {
        "warnings_and_below": {
            "()" : "__main__.filter_maker",
            "level": "WARNING"
        }
    },
    "handlers": {
        "stdout": {
            "class": "logging.StreamHandler",
            "level": "INFO",
            "formatter": "simple",
            "stream": "ext://sys.stdout",
            "filters": ["warnings_and_below"]
        },
        "stderr": {
            "class": "logging.StreamHandler",
            "level": "ERROR",
            "formatter": "simple",
            "stream": "ext://sys.stderr"
        },
        "file": {
            "class": "logging.FileHandler",
            "formatter": "simple",
            "filename": "app.log",
            "mode": "w"
        }
    },
    "root": {
        "level": "DEBUG",
        "handlers": [
            "stderr",
            "stdout",
            "file"
        ]
    }
}
'''
```

```
def filter_maker(level):
    level = getattr(logging, level)

    def filter(record):
        return record.levelno <= level

    return filter

logging.config.dictConfig(json.loads(CONFIG))
logging.debug('A DEBUG message')
logging.info('An INFO message')
logging.warning('A WARNING message')
logging.error('An ERROR message')
logging.critical('A CRITICAL message')
```

使用这样的命令运行它之后:

```
python main.py 2>stderr.log >stdout.log
```

我们可以看到结果是符合预期的:

```
$ more *.log
::::::::::::::
app.log
::::::::::::::
DEBUG    - A DEBUG message
INFO     - An INFO message
WARNING  - A WARNING message
ERROR    - An ERROR message
CRITICAL - A CRITICAL message
::::::::::::::
stderr.log
::::::::::::::
ERROR    - An ERROR message
CRITICAL - A CRITICAL message
::::::::::::::
stdout.log
::::::::::::::
INFO     - An INFO message
WARNING  - A WARNING message
```

## 日志配置服务器示例

以下是一个用到了日志配置服务器的模块示例:

```
import logging
import logging.config
import time
import os

# 读取初始配置文件
logging.config.fileConfig('logging.conf')

# 在 9999 端口上创建并启动监听器
t = logging.config.listen(9999)
t.start()
```

```
logger = logging.getLogger('simpleExample')

try:
    # 循环遍历日志记录调用以查看
    # 新配置进行的修改，直到按下 Ctrl+C
    while True:
        logger.debug('debug message')
        logger.info('info message')
        logger.warning('warn message')
        logger.error('error message')
        logger.critical('critical message')
        time.sleep(5)
except KeyboardInterrupt:
    # 清理
    logging.config.stopListening()
    t.join()
```

以下脚本将接受文件名作为参数，然后将此文件发送到服务器，前面加上文件的二进制编码长度，做为新的日志配置：

```
#!/usr/bin/env python
import socket, sys, struct

with open(sys.argv[1], 'rb') as f:
    data_to_send = f.read()

HOST = 'localhost'
PORT = 9999
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print('connecting...')
s.connect((HOST, PORT))
print('sending config...')
s.send(struct.pack('>L', len(data_to_send)))
s.send(data_to_send)
s.close()
print('complete')
```

## 处理日志 handler 的阻塞

有时你必须让日志记录处理程序的运行不会阻塞你要记录日志的线程。 这在 Web 应用程序中是很常见，当然在其他场景中也可能发生。

有一种原因往往会让程序表现迟钝，这就是 `SMTPHandler`：由于很多因素是开发人员无法控制的（例如邮件或网络基础设施的性能不佳），发送电子邮件可能需要很长时间。不过几乎所有网络 handler 都可能会发生阻塞：即使是 `SocketHandler` 操作也可能在后台执行 DNS 查询，而这种查询实在太慢了（并且 DNS 查询还可能在很底层的套接字库代码中，位于 Python 层之下，超出了可控范围）。

有一种解决方案是分成两部分实现。第一部分，针对那些对性能有要求的关键线程，只为日志对象连接一个 `QueueHandler`。日志对象只需简单地写入队列即可，可为队列设置足够大的容量，或者可以在初始化时不设置容量上限。尽管为以防万一，可能需要在代码中捕获 `queue.Full` 异常,不过队列写入操作通常会很快得以处理。如果要开发库代码，包含性能要求较高的线程，为了让使用该库的开发人员受益，请务必在开发文档中进行标明（包括建议仅连接 `QueueHandlers` ）。

解决方案的另一部分就是 `QueueListener`，它被设计为 `QueueHandler` 的对应部分。`QueueListener` 非常简单：传入一个队列和一些 handler，并启动一个内部线程，用于侦听 `QueueHandlers`（或其他 `LogRecords` 源）发送的 LogRecord 队列。`LogRecords` 会从队列中移除并传给 handler 处理。

`QueueListener` 作为单独的类，好处就是可以用同一个实例为多个 `QueueHandlers` 服务。这比把现有 handler 类线程化更加资源友好，后者会每个 handler 会占用一个线程，却没有特别的好处。

以下是这两个类的运用示例（省略了 import 语句）：

```
que = queue.Queue(-1)  # 对大小没有限制
queue_handler = QueueHandler(que)
handler = logging.StreamHandler()
listener = QueueListener(que, handler)
root = logging.getLogger()
root.addHandler(queue_handler)
formatter = logging.Formatter('%(threadName)s: %(message)s')
handler.setFormatter(formatter)
listener.start()
# 日志输出将显示生成事件的线程（主线程）
# 而不是监控内部队列的内部线程。这也正是
# 你所希望的。
root.warning('Look out!')
listener.stop()
```

在运行后会产生:

```
MainThread: Look out!
```

**备注:** 虽然前面的讨论没有专门提及异步代码，但需要注意当在异步代码中记录日志时，网络甚至文件处理器都可能会导致问题（阻塞事件循环）因为某些日志记录是在 `asyncio` 内部完成的。如果在应用程序中使用了任何异步代码，最好的做法是使用上面的日志记录方式，这样任何阻塞式代码都将只在 `QueueListener` 线程中运行。

> *在 3.5 版本发生变更:* 在 Python 3.5 之前，`QueueListener` 总会把由队列接收到的每条信息都传递给已初始化的每个处理程序。（因为这里假定级别过滤操作已在写入队列时完成了。）从 3.5 版开始，可以修改这种处理方式，只要将关键字参数 `respect_handler_level=True` 传给侦听器的构造函数即可。这样侦听器将会把每条信息的级别与 handler 的级别进行比较，只在适配时才会将信息传给 handler 。

> *在 3.14 版本发生变更:* `QueueListener` 可通过 `with` 语句来启动（和停止）。 例如:
>
> ```
> with QueueListener(que, handler) as listener:
>     # 该队列监听器将会在进入
>     # 'with' 代码块时自动启动。
>     pass
> # 该队列监听器将在退出
> # 'with' 代码块时自动停止。
> ```

## 通过网络收发日志事件

假定现在要通过网络发送日志事件，并在接收端进行处理。有一种简单的方案，就是在发送端的根日志对象连接一个 SocketHandler 实例：

```
import logging, logging.handlers

rootLogger = logging.getLogger('')
rootLogger.setLevel(logging.DEBUG)
socketHandler = logging.handlers.SocketHandler('localhost',
                    logging.handlers.DEFAULT_TCP_LOGGING_PORT)
# 不必设置格式化器，因为套接字处理器会将事件以未格式化的
# pickle 形式发送
rootLogger.addHandler(socketHandler)

# 现在我们可以写入根记录器或任何其他记录器。 首先是根记录器...
logging.info('Jackdaws love my big sphinx of quartz.')

# 现在定义几个可以代表你的应用程序中不同组成部分的
# 其他日志记录器：

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')

logger1.debug('Quick zephyrs blow, vexing daft Jim.')
logger1.info('How quickly daft jumping zebras vex.')
logger2.warning('Jail zesty vixen who grabbed pay from quack.')
logger2.error('The five boxing wizards jump quickly.')
```

在接收端，可以用 socketserver 模块设置一个接收器。简要示例如下：

```
import pickle
import logging
import logging.handlers
import socketserver
import struct


class LogRecordStreamHandler(socketserver.StreamRequestHandler):
    """Handler for a streaming logging request.

    This basically logs the record using whatever logging policy is
    configured locally.
    """

    def handle(self):
        """
        Handle multiple requests - each expected to be a 4-byte length,
        followed by the LogRecord in pickle format. Logs the record
        according to whatever policy is configured locally.
        """
        while True:
            chunk = self.connection.recv(4)
            if len(chunk) < 4:
                break
            slen = struct.unpack('>L', chunk)[0]
            chunk = self.connection.recv(slen)
            while len(chunk) < slen:
                chunk = chunk + self.connection.recv(slen - len(chunk))
            obj = self.unPickle(chunk)
```

```
            record = logging.makeLogRecord(obj)
            self.handleLogRecord(record)

    def unPickle(self, data):
        return pickle.loads(data)

    def handleLogRecord(self, record):
        # 如果指定了名称，我们将使用指定的记录器而不是
        # record 原本使用的。
        if self.server.logname is not None:
            name = self.server.logname
        else:
            name = record.name
        logger = logging.getLogger(name)
        # 注意每条记录都会被写入。 这是因为 Logger.handle
        # 通常会在记录器层级过滤之后被调用。 如果你希望
        # 进行过滤，请在客户端结束时进行以避免浪费循环
        # 并节省网络带宽！
        logger.handle(record)

class LogRecordSocketReceiver(socketserver.ThreadingTCPServer):
    """
    Simple TCP socket-based logging receiver suitable for testing.
    """

    allow_reuse_address = True

    def __init__(self, host='localhost',
                 port=logging.handlers.DEFAULT_TCP_LOGGING_PORT,
                 handler=LogRecordStreamHandler):
        socketserver.ThreadingTCPServer.__init__(self, (host, port), handler)
        self.abort = 0
        self.timeout = 1
        self.logname = None

    def serve_until_stopped(self):
        import select
        abort = 0
        while not abort:
            rd, wr, ex = select.select([self.socket.fileno()],
                                       [], [],
                                       self.timeout)
            if rd:
                self.handle_request()
            abort = self.abort

def main():
    logging.basicConfig(
        format='%(relativeCreated)5d %(name)-15s %(levelname)-8s %(message)s')
    tcpserver = LogRecordSocketReceiver()
    print('About to start TCP server...')
    tcpserver.serve_until_stopped()

if __name__ == '__main__':
    main()
```

先运行服务端，再运行客户端。客户端控制台不会显示什么信息；在服务端应该会看到如下内容：

```
About to start TCP server...
   59 root            INFO     Jackdaws love my big sphinx of quartz.
   59 myapp.area1     DEBUG    Quick zephyrs blow, vexing daft Jim.
   69 myapp.area1     INFO     How quickly daft jumping zebras vex.
   69 myapp.area2     WARNING  Jail zesty vixen who grabbed pay from quack.
   69 myapp.area2     ERROR    The five boxing wizards jump quickly.
```

请注意在某些情况下 pickle 会存在一些安全问题。 如果这些问题对你有影响，你可以换用自己的替代序列化方案，只要重写 `makePickle()` 方法并在其中实现你的替代方案，并调整上述脚本以使用这个替代方案。

### 在生产中运行日志套接字侦听器

要在生产环境中运行日志记录监听器，你可能需要使用一个进程管理工具如 Supervisor。 这个 Gist 提供了使用 Supervisor 来运行上述功能的基本框架文件。 它由以下文件组成:

| 文件 | 目的 |
| --- | --- |
| `prepare.sh` | 用于准备针对测试的环境的 Bash 脚本 |
| `supervisor.conf` | Supervisor 配置文件，其中有用于侦听器和多进程 Web 应用程序的条目 |
| `ensure_app.sh` | 用于确保 Supervisor 在使用上述配置运行的 Bash 脚本 |
| `log_listener.py` | 接收日志事件并将其记录到文件中的套接字监听器 |
| `main.py` | 一个通过连接到监听器的套接字来执行日志记录的简单 Web 应用程序 |
| `webapp.json` | 一个针对 Web 应用程序的 JSON 配置文件 |
| `client.py` | 使用 Web 应用程序的 Python 脚本 |

该 Web 应用程序使用了 Gunicorn，这个流行的 Web 应用服务器可启动多个工作进程来处理请求。 这个示例设置演示了多个工作进程是如何写入相同的日志文件而不会相互冲突的 --- 它们都通过套接字监听器进程操作。

要测试这些文件，请在 POSIX 环境中执行以下操作:

1. 使用 Download ZIP 按钮将 此 Gist 下载为 ZIP 归档文件。
2. 将上述文件从归档解压缩到一个初始目录中。
3. 在初始目录中，运行 `bash prepare.sh` 完成准备工作。 这将创建一个 `run` 子目录来包含 Supervisor 相关文件和日志文件，以及一个 `venv` 子目录来包含安装了 `bottle`, `gunicorn` 和 `supervisor` 的虚拟环境。
4. 运行 `bash ensure_app.sh` 以确保 Supervisor 正在使用上述配置运行。
5. 运行 `venv/bin/python client.py` 来使用 Web 应用程序，这将使得记录被写入到日志中。
6. 检查 `run` 子目录中的日志文件。 你应当看到匹配模式为 `app.log*` 的文件中最新的日志记录行。 它们不会有任何特定的顺序，因为它们是由不同的工作进程以不确定的方式并发地处理的。

7. 你可以通过运行 `venv/bin/supervisorctl -c supervisor.conf shutdown` 来关闭监听器和 Web 应用程序。

你可能需要在配置的端口与你的测试环境中其他程序发生意外冲突的情况下调整配置文件。

默认配置使用一个 9020 端口上的 TCP 套接字。 你可以通过以下方式改用 Unix 域套接字代替 TCP 套接字：

1. 在 `listener.json` 中，添加一个 `socket` 键并设为你想使用的域套接字路径。 如果存在该键，监听器就将监听相应的域套接字而不是 TCP 套接字 (`port` 键将被忽略)。
2. 在 `webapp.json` 中，修改套接字处理器配置字典以使 `host` 值为该域套接字的路径，并将 `port` 值设为 `null`。

## 在自己的输出日志中添加上下文信息

有时，除了调用日志对象时传入的参数之外，还希望日志输出中能包含上下文信息。 比如在网络应用程序中，可能需要在日志中记录某客户端的信息（如远程客户端的用户名或 IP 地址）。 这虽然可以用 *extra* 参数实现，但传递起来并不总是很方便。 虽然为每个网络连接都创建 `Logger` 实例貌似不错，但并不是个好主意，因为这些实例不会被垃圾回收。 虽然在实践中不是问题，但当 `Logger` 实例的数量取决于应用程序要采用的日志粒度时，如果 `Logger` 实例的数量实际上是无限的，则有可能难以管理。

### 利用 LoggerAdapter 传递上下文信息

要传递上下文信息和日志事件信息，有一种简单方案是利用 `LoggerAdapter` 类。这个类设计得类似 `Logger`，所以可以直接调用 `debug()`、`info()`、 `warning()`、 `error()`、`exception()`、`critical()` 和 `log()`。这些方法的签名与 `Logger` 对应的方法相同，所以这两类实例可以交换使用。

当你创建一个 `LoggerAdapter` 的实例时，你会传入一个 `Logger` 的实例和一个包含了上下文信息的字典对象。当你调用一个 `LoggerAdapter` 实例的方法时，它会把调用委托给内部的 `Logger` 的实例，并为其整理相关的上下文信息。这是 `LoggerAdapter` 的一个代码片段:

```
def debug(self, msg, /, *args, **kwargs):
    """
    在添加来自这个适配器实例的上下文信息之后，
    将调试调用委托给下层的日志记录器。
    """
    msg, kwargs = self.process(msg, kwargs)
    self.logger.debug(msg, *args, **kwargs)
```

`LoggerAdapter` 的 `process()` 方法是将上下文信息添加到日志的输出中。 它传入日志消息和日志调用的关键字参数，并传回（隐式的）这些修改后的内容去调用底层的日志记录器。此方法的默认参数只是一个消息字段，但留有一个 'extra' 的字段作为关键字参数传给构造器。当然，如果你在调用适配器时传入了一个 'extra' 字段的参数，它会被静默覆盖。

使用 'extra' 的优点是这些键值对会被传入 `LogRecord` 实例的 __dict__ 中，让你通过 `Formatter` 的实例直接使用定制的字符串，实例能找到这个字典类对象的键。 如果你需要一个其他的方法，比如

说，想要在消息字符串前后增加上下文信息，你只需要创建一个 `LoggerAdapter` 的子类，并覆盖它的 `process()` 方法来做你想做的事情，以下是一个简单的示例:

```
class CustomAdapter(logging.LoggerAdapter):
    """
    This example adapter expects the passed in dict-like object to have a
    'connid' key, whose value in brackets is prepended to the log message.
    """
    def process(self, msg, kwargs):
        return '[%s] %s' % (self.extra['connid'], msg), kwargs
```

你可以这样使用:

```
logger = logging.getLogger(__name__)
adapter = CustomAdapter(logger, {'connid': some_conn_id})
```

然后，你记录在适配器中的任何事件消息前将添加 `some_conn_id` 的值。

### 使用除字典之外的其它对象传递上下文信息

你不需要将一个实际的字典传递给 `LoggerAdapter`-你可以传入一个实现了 `__getitem__` 和 `__iter__` 的类的实例，这样它就像是一个字典。这对于你想动态生成值（而字典中的值往往是常量）将很有帮助。

### 使用过滤器传递上下文信息

你也可以使用一个用户定义的类 `Filter` 在日志输出中添加上下文信息。`Filter` 的实例是被允许修改传入的 `LogRecords`，包括添加其他的属性，然后可以使用合适的格式化字符串输出，或者可以使用一个自定义的类 `Formatter`。

例如，在一个web应用程序中，正在处理的请求（或者至少是请求的一部分），可以存储在一个线程本地 (`threading.local`) 变量中，然后从 `Filter` 中去访问。请求中的信息，如IP地址和用户名将被存储在 `LogRecord` 中，使用上例 `LoggerAdapter` 中的 'ip' 和 'user' 属性名。在这种情况下，可以使用相同的格式化字符串来得到上例中类似的输出结果。这是一段示例代码:

```
import logging
from random import choice

class ContextFilter(logging.Filter):
    """
    This is a filter which injects contextual information into the log.

    Rather than use actual contextual information, we just use random
    data in this demo.
    """

    USERS = ['jim', 'fred', 'sheila']
    IPS = ['123.231.231.123', '127.0.0.1', '192.168.0.1']

    def filter(self, record):

        record.ip = choice(ContextFilter.IPS)
        record.user = choice(ContextFilter.USERS)
```

```
        return True

if __name__ == '__main__':
    levels = (logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR, logging
    logging.basicConfig(level=logging.DEBUG,
                        format='%(asctime)-15s %(name)-5s %(levelname)-8s IP: %(ip
    a1 = logging.getLogger('a.b.c')
    a2 = logging.getLogger('d.e.f')

    f = ContextFilter()
    a1.addFilter(f)
    a2.addFilter(f)
    a1.debug('A debug message')
    a1.info('An info message with %s', 'some parameters')
    for x in range(10):
        lvl = choice(levels)
        lvlname = logging.getLevelName(lvl)
        a2.log(lvl, 'A message at %s level with %d %s', lvlname, 2, 'parameters')
```

在运行时，产生如下内容:

```
2010-09-06 22:38:15,292 a.b.c DEBUG    IP: 123.231.231.123 User: fred     A debug
2010-09-06 22:38:15,300 a.b.c INFO     IP: 192.168.0.1     User: sheila   An info
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 127.0.0.1       User: sheila   A messag
2010-09-06 22:38:15,300 d.e.f ERROR    IP: 127.0.0.1       User: jim      A messag
2010-09-06 22:38:15,300 d.e.f DEBUG    IP: 127.0.0.1       User: sheila   A messag
2010-09-06 22:38:15,300 d.e.f ERROR    IP: 123.231.231.123 User: fred     A messag
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 192.168.0.1     User: jim      A messag
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 127.0.0.1       User: sheila   A messag
2010-09-06 22:38:15,300 d.e.f DEBUG    IP: 192.168.0.1     User: jim      A messag
2010-09-06 22:38:15,301 d.e.f ERROR    IP: 127.0.0.1       User: sheila   A messag
2010-09-06 22:38:15,301 d.e.f DEBUG    IP: 123.231.231.123 User: fred     A messag
2010-09-06 22:38:15,301 d.e.f INFO     IP: 123.231.231.123 User: fred     A messag
```

## contextvars 的使用

自 Python 3.7 起，contextvars 模块提供了同时适用于 threading 和 asyncio 处理需求的上下文本地存储。 因此这种存储类型通常要比线程本地存储更好。 下面的例子演示了在多线程环境中日志如何用上下文信息来填充内容，例如 Web 应用程序所处理的请求属性。

出于说明的目的，比方说你有几个不同的 Web 应用程序，彼此都保持独立状态但运行在同一个 Python 进程中并且它们共同使用了某个库。 这些应用程序要如何拥有各自的日志记录，其中来自这个库的日志消息（以及其他请求处理代码）会发到对应的应用程序的日志文件，同时在日志中包括额外的上下文信息如客户端 IP、HTTP 请求方法和客户端用户名呢?

让我们假定这个库可以通过以下代码来模拟:

```
# webapplib.py
import logging
import time

logger = logging.getLogger(__name__)

def useful():
    # 一条从库中记录的代表性事件
```

```
    logger.debug('Hello from webapplib!')
    # 休眠一下以便其他线程能够运行
    time.sleep(0.01)
```

我们可以通过两个简单的类 `Request` 和 `WebApp` 来模拟多个 Web 应用程序。 它们模拟了真正的多线程 Web 应用程序是如何工作的 —— 每个请求均由单独的线程来处理:

```
# main.py
import argparse
from contextvars import ContextVar
import logging
import os
from random import choice
import threading
import webapplib

logger = logging.getLogger(__name__)
root = logging.getLogger()
root.setLevel(logging.DEBUG)

class Request:
    """
    A simple dummy request class which just holds dummy HTTP request method,
    client IP address and client username
    """
    def __init__(self, method, ip, user):
        self.method = method
        self.ip = ip
        self.user = user

# 将在模拟中使用的一组假请求 — 我们将从这个列表随机选取。
# 请注意所有 GET 请求都来自 192.168.2.XXX 地址，
# 而 POST 请求都来自 192.16.3.XXX 地址。
# 在这些样例请求中有三个用户。

REQUESTS = [
    Request('GET', '192.168.2.20', 'jim'),
    Request('POST', '192.168.3.20', 'fred'),
    Request('GET', '192.168.2.21', 'sheila'),
    Request('POST', '192.168.3.21', 'jim'),
    Request('GET', '192.168.2.22', 'fred'),
    Request('POST', '192.168.3.22', 'sheila'),
]

# 请注意格式字符串包括了对请求上下文信息的引用
# 如 HTTP 方法，客户端 IP 和用户名

formatter = logging.Formatter('%(threadName)-11s %(appName)s %(name)-9s %(user)-6s

# 创建我们的上下文变量。 它们将在开始处理请求时被填充，
# 并将在处理时发生的日志记录中被使用。

ctx_request = ContextVar('request')
ctx_appname = ContextVar('appname')

class InjectingFilter(logging.Filter):
    """
    A filter which injects context-specific information into logs and ensures
```

```
    that only information for a specific webapp is included in its log
    """
    def __init__(self, app):
        self.app = app

    def filter(self, record):
        request = ctx_request.get()
        record.method = request.method
        record.ip = request.ip
        record.user = request.user
        record.appName = appName = ctx_appname.get()
        return appName == self.app.name

class WebApp:
    """
    A dummy web application class which has its own handler and filter for a
    webapp-specific log.
    """
    def __init__(self, name):
        self.name = name
        handler = logging.FileHandler(name + '.log', 'w')
        f = InjectingFilter(self)
        handler.setFormatter(formatter)
        handler.addFilter(f)
        root.addHandler(handler)
        self.num_requests = 0

    def process_request(self, request):
        """
        This is the dummy method for processing a request. It's called on a
        different thread for every request. We store the context information into
        the context vars before doing anything else.
        """
        ctx_request.set(request)
        ctx_appname.set(self.name)
        self.num_requests += 1
        logger.debug('Request processing started')
        webapplib.useful()
        logger.debug('Request processing finished')

def main():
    fn = os.path.splitext(os.path.basename(__file__))[0]
    adhf = argparse.ArgumentDefaultsHelpFormatter
    ap = argparse.ArgumentParser(formatter_class=adhf, prog=fn,
                                 description='Simulate a couple of web '
                                             'applications handling some '
                                             'requests, showing how request '
                                             'context can be used to '
                                             'populate logs')
    aa = ap.add_argument
    aa('--count', '-c', type=int, default=100, help='How many requests to simulate
    options = ap.parse_args()

    # 创建假 Web 应用并将其放在列表中以便我们
    # 用于随机选取
    app1 = WebApp('app1')
    app2 = WebApp('app2')
    apps = [app1, app2]
    threads = []
    # 添加一个将捕获所有事件的通用处理器
```

```
    handler = logging.FileHandler('app.log', 'w')
    handler.setFormatter(formatter)
    root.addHandler(handler)

    # 生成调用来处理请求
    for i in range(options.count):
        try:
            # Pick an app at random and a request for it to process
            app = choice(apps)
            request = choice(REQUESTS)
            # Process the request in its own thread
            t = threading.Thread(target=app.process_request, args=(request,))
            threads.append(t)
            t.start()
        except KeyboardInterrupt:
            break

    # 等待线程终结
    for t in threads:
        t.join()

    for app in apps:
        print('%s processed %s requests' % (app.name, app.num_requests))

if __name__ == '__main__':
    main()
```

如果你运行上面的代码，你将会发现约有半数请求是发给 `app1.log` 而其余的则是发给 `app2.log`，并且所有请求都会被记录至 `app.log`。 每个 Web 应用专属的日志将只包含该 Web 应用的日志条目，请求信息也将以一致的方式显示在日志里（即每个模拟请求中的信息将总是在一个日志行中一起显示）。 如下面的 shell 输出所示:

```
~/logging-contextual-webapp$ python main.py
app1 processed 51 requests
app2 processed 49 requests
~/logging-contextual-webapp$ wc -l *.log
  153 app1.log
  147 app2.log
  300 app.log
  600 total
~/logging-contextual-webapp$ head -3 app1.log
Thread-3 (process_request) app1 __main__  jim    192.168.3.21 POST Request process
Thread-3 (process_request) app1 webapplib jim    192.168.3.21 POST Hello from weba
Thread-5 (process_request) app1 __main__  jim    192.168.3.21 POST Request process
~/logging-contextual-webapp$ head -3 app2.log
Thread-1 (process_request) app2 __main__  sheila 192.168.2.21 GET  Request process
Thread-1 (process_request) app2 webapplib sheila 192.168.2.21 GET  Hello from weba
Thread-2 (process_request) app2 __main__  jim    192.168.2.20 GET  Request process
~/logging-contextual-webapp$ head app.log
Thread-1 (process_request) app2 __main__  sheila 192.168.2.21 GET  Request process
Thread-1 (process_request) app2 webapplib sheila 192.168.2.21 GET  Hello from weba
Thread-2 (process_request) app2 __main__  jim    192.168.2.20 GET  Request process
Thread-3 (process_request) app1 __main__  jim    192.168.3.21 POST Request process
Thread-2 (process_request) app2 webapplib jim    192.168.2.20 GET  Hello from weba
Thread-3 (process_request) app1 webapplib jim    192.168.3.21 POST Hello from weba
Thread-4 (process_request) app2 __main__  fred   192.168.2.22 GET  Request process
Thread-5 (process_request) app1 __main__  jim    192.168.3.21 POST Request process
Thread-4 (process_request) app2 webapplib fred   192.168.2.22 GET  Hello from weba
```

```
Thread-6 (process_request) app1 __main__  jim    192.168.3.21 POST Request process
~/logging-contextual-webapp$ grep app1 app1.log | wc -l
153
~/logging-contextual-webapp$ grep app2 app2.log | wc -l
147
~/logging-contextual-webapp$ grep app1 app.log | wc -l
153
~/logging-contextual-webapp$ grep app2 app.log | wc -l
147
```

## 在处理器中传递上下文信息

每个 Handler 都有自己的过滤器链。 如果你想向一个 LogRecord 添加上下文信息而不使其泄露给其它处理器，你可以使用一个返回新 LogRecord 而不是原地修改它的过滤器，如下面的脚本所示:

```
import copy
import logging

def filter(record: logging.LogRecord):
    record = copy.copy(record)
    record.user = 'jim'
    return record

if __name__ == '__main__':
    logger = logging.getLogger()
    logger.setLevel(logging.INFO)
    handler = logging.StreamHandler()
    formatter = logging.Formatter('%(message)s from %(user)-8s')
    handler.setFormatter(formatter)
    handler.addFilter(filter)
    logger.addHandler(handler)

    logger.info('A log message')
```

## 从多个进程记录至单个文件

尽管 logging 是线程安全的，将单个进程中的多个线程日志记录至单个文件也 *是* 受支持的，但将 *多个进程* 中的日志记录至单个文件则 *不是* 受支持的，因为在 Python 中并没有在多个进程中实现对单个文件访问的序列化的标准方案。 如果你需要将多个进程中的日志记录至单个文件，有一个方案是让所有进程都将日志记录至一个 SocketHandler，然后用一个实现了套接字服务器的单独进程一边从套接字中读取一边将日志记录至文件。 （如果愿意的话，你可以在一个现有进程中专门开一个线程来执行此项功能。） 这一部分 文档对此方式有更详细的介绍，并包含一个可用的套接字接收器，你自己的应用可以在此基础上进行适配。

你也可以编写你自己的处理器，让其使用 multiprocessing 模块中的 Lock 类来顺序访问你的多个进程中的文件。 标准库的 FileHandler 及其子类均未使用 multiprocessing。

或者，你也可以使用 Queue 和 QueueHandler 将所有的日志事件发送至你的多进程应用的一个进程中。 以下示例脚本演示了如何执行此操作。 在示例中，一个单独的监听进程负责监听其他进程的日志事件，并根据自己的配置记录。 尽管示例只演示了这种方法（例如你可能希望使用单独的监听线

程而非监听进程 —— 它们的实现是类似的），但你也可以在应用程序的监听进程和其他进程使用不同的配置，它可以作为满足你特定需求的一个基础:

```
# 你将在自己的代码中需要这些导入
import logging
import logging.handlers
import multiprocessing

# 以下两行导入仅针对本演示
from random import choice, random
import time

#
# 因为你会希望为监听进程和工作进程定义日志记录配置，
# 这些进程函数将接受一个可调用对象作为 configurer 形参
# 用于为进程配置日志记录。 这些函数还将接受一个队列，
# 供它们在通信中使用。
#
# 实际上，你可以根据你的需要任意配置监听进程，但请注意在
# 该简单示例中监听进程没有对收到的记录应用层级或过滤逻辑。
# 在实践中，你可能会希望在工作进程中执行此逻辑，以避免发送
# 将会在进程间被过滤掉的事件。
#
# 轮转文件的尺寸被设置为很小以便你能方便地查看结果。
def listener_configurer():
    root = logging.getLogger()
    h = logging.handlers.RotatingFileHandler('mptest.log', 'a', 300, 10)
    f = logging.Formatter('%(asctime)s %(processName)-10s %(name)s %(levelname)-8s
    h.setFormatter(f)
    root.addHandler(h)

# 这是监听进程的最高层级循环：等待队列中的日志记录事件
# (LogRecords) 并处理它们，当在接受 LogRecord 时收到 None
# 则退出。
def listener_process(queue, configurer):
    configurer()
    while True:
        try:
            record = queue.get()
            if record is None:  # 我们发送该值以通知监听进程退出。
                break
            logger = logging.getLogger(record.name)
            logger.handle(record)  # 未应用层级或过滤逻辑 —— 直接做！          except
            import sys, traceback
            print('Whoops! Problem:', file=sys.stderr)
            traceback.print_exc(file=sys.stderr)

# 用于在本演示中随机选取的数组

LEVELS = [logging.DEBUG, logging.INFO, logging.WARNING,
          logging.ERROR, logging.CRITICAL]

LOGGERS = ['a.b.c', 'd.e.f']

MESSAGES = [
    'Random message #1',
    'Random message #2',
    'Random message #3',
]
```

```
# 工作进程配置在工作进程开始运行时完成。
# 请注意在 Windows 上不能依赖 fork 语义，因此每个进程
# 将在启动时运行日志记录配置代码。
def worker_configurer(queue):
    h = logging.handlers.QueueHandler(queue)  # 只需要一个处理器
    root = logging.getLogger()
    root.addHandler(h)
    # 发送所有消息，用于演示；未应用其他层级或过滤逻辑。
    root.setLevel(logging.DEBUG)

# 这是工作进程的最高层级循环，它将在结束前以随机间隔
# 记录十个事件。
# 打印消息只是让你知道它正在做一些事情！
def worker_process(queue, configurer):
    configurer(queue)
    name = multiprocessing.current_process().name
    print('Worker started: %s' % name)
    for i in range(10):
        time.sleep(random())
        logger = logging.getLogger(choice(LOGGERS))
        level = choice(LEVELS)
        message = choice(MESSAGES)
        logger.log(level, message)
    print('Worker finished: %s' % name)

# 以下是演示整合各个组件的地方。 创建队列，创建并启动
# 监听进程，创建十个工作进程并启动它们，等待它们结束，
# 然后向队列发送 None 以通知监听进程退出。
def main():
    queue = multiprocessing.Queue(-1)
    listener = multiprocessing.Process(target=listener_process,
                                       args=(queue, listener_configurer))
    listener.start()
    workers = []
    for i in range(10):
        worker = multiprocessing.Process(target=worker_process,
                                         args=(queue, worker_configurer))
        workers.append(worker)
        worker.start()
    for w in workers:
        w.join()
    queue.put_nowait(None)
    listener.join()

if __name__ == '__main__':
    main()
```

上面脚本的一个变种，仍然在主进程中记录日志，但使用一个单独的线程:

```
import logging
import logging.config
import logging.handlers
from multiprocessing import Process, Queue
import random
import threading
import time

def logger_thread(q):
```

```
    while True:
        record = q.get()
        if record is None:
            break
        logger = logging.getLogger(record.name)
        logger.handle(record)


def worker_process(q):
    qh = logging.handlers.QueueHandler(q)
    root = logging.getLogger()
    root.setLevel(logging.DEBUG)
    root.addHandler(qh)
    levels = [logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR,
              logging.CRITICAL]
    loggers = ['foo', 'foo.bar', 'foo.bar.baz',
               'spam', 'spam.ham', 'spam.ham.eggs']
    for i in range(100):
        lvl = random.choice(levels)
        logger = logging.getLogger(random.choice(loggers))
        logger.log(lvl, 'Message no. %d', i)

if __name__ == '__main__':
    q = Queue()
    d = {
        'version': 1,
        'formatters': {
            'detailed': {
                'class': 'logging.Formatter',
                'format': '%(asctime)s %(name)-15s %(levelname)-8s %(processName)-
            }
        },
        'handlers': {
            'console': {
                'class': 'logging.StreamHandler',
                'level': 'INFO',
            },
            'file': {
                'class': 'logging.FileHandler',
                'filename': 'mplog.log',
                'mode': 'w',
                'formatter': 'detailed',
            },
            'foofile': {
                'class': 'logging.FileHandler',
                'filename': 'mplog-foo.log',
                'mode': 'w',
                'formatter': 'detailed',
            },
            'errors': {
                'class': 'logging.FileHandler',
                'filename': 'mplog-errors.log',
                'mode': 'w',
                'level': 'ERROR',
                'formatter': 'detailed',
            },
        },
        'loggers': {
            'foo': {
                'handlers': ['foofile']
```

```
            }
        },
        'root': {
            'level': 'DEBUG',
            'handlers': ['console', 'file', 'errors']
        },
    }
    workers = []
    for i in range(5):
        wp = Process(target=worker_process, name='worker %d' % (i + 1), args=(q,))
        workers.append(wp)
        wp.start()
    logging.config.dictConfig(d)
    lp = threading.Thread(target=logger_thread, args=(q,))
    lp.start()
    # 在这里，主进程可以执行某些对它自己有用的工作
    # 当其完成后，即可等待工作进程终结...
    for wp in workers:
        wp.join()
    # 现在再通知日志记录线程结束
    q.put(None)
    lp.join()
```

这段变种的代码展示了如何使用特定的日志记录配置 - 例如 `foo` 记录器使用了特殊的处理程序，将 `foo` 子系统中所有的事件记录至一个文件 `mplog-foo.log` 。在主进程（即使是在工作进程中产生的日志事件）的日志记录机制中将直接使用恰当的配置。

### concurrent.futures.ProcessPoolExecutor 的用法

若要利用 concurrent.futures.ProcessPoolExecutor 启动工作进程，创建队列的方式应稍有不同。不能是：

```
queue = multiprocessing.Queue(-1)
```

而应是：

```
queue = multiprocessing.Manager().Queue(-1)  # 同样适用于上面的例子
```

然后就可以将以下工作进程的创建过程：

```
workers = []
for i in range(10):
    worker = multiprocessing.Process(target=worker_process,
                                     args=(queue, worker_configurer))
    workers.append(worker)
    worker.start()
for w in workers:
    w.join()
```

改为 (记得要先导入 concurrent.futures):

```
with concurrent.futures.ProcessPoolExecutor(max_workers=10) as executor:
    for i in range(10):
        executor.submit(worker_process, queue, worker_configurer)
```

### 使用 Gunicorn 和 uWSGI 来部署 Web 应用程序

当使用 Gunicorn 或 uWSGI (或其他类似工具) 来部署 Web 应用时，会创建多个工作进程来处理客户端请求。 在这种环境下，要避免在你的 Web 应用中直接创建基于文件的处理器。 而应改为使用一个 `SocketHandler` 将来自 Web 应用的日志发送到在单独进程中运行的监听器。 这可以通过使用一个进程管理工具例如 Supervisor 来进行设置 —— 请参阅 Running a logging socket listener in production 了解详情。

## 轮换日志文件

有时您会希望让日志文件增长到一定大小，然后打开一个新的接着记录日志。 您可能希望只保留一定数量的日志文件，当创建文件达到指定数量后将会轮换文件，从而使文件数量和文件大小都保持在一定范围之内。 对于这种使用模式，日志包提供了一个 `RotatingFileHandler`:

```
import glob
import logging
import logging.handlers

LOG_FILENAME = 'logging_rotatingfile_example.out'

# 使用我们想要的输出层级设置特定的日志记录器
my_logger = logging.getLogger('MyLogger')
my_logger.setLevel(logging.DEBUG)

# 将日志消息处理器添加到日志记录器
handler = logging.handlers.RotatingFileHandler(
              LOG_FILENAME, maxBytes=20, backupCount=5)

my_logger.addHandler(handler)

# 记录一些消息
for i in range(20):
    my_logger.debug('i = %d' % i)

# 查看创建了哪些文件
logfiles = glob.glob('%s*' % LOG_FILENAME)

for filename in logfiles:
    print(filename)
```

结果应该是6个单独的文件，每个文件都包含了应用程序的部分历史日志:

```
logging_rotatingfile_example.out
logging_rotatingfile_example.out.1
logging_rotatingfile_example.out.2
logging_rotatingfile_example.out.3
logging_rotatingfile_example.out.4
logging_rotatingfile_example.out.5
```

最新的文件始终是 `logging_rotatingfile_example.out`，每次到达大小限制时，都会使用后缀 `.1` 重命名。每个现有的备份文件都会被重命名并增加其后缀（例如 `.1` 变为 `.2` ），而 `.6` 文件会被删除掉。

显然，这个例子将日志长度设置得太小，这是一个极端的例子。 你可能希望将 *maxBytes* 设置为一个合适的值。

## 使用其他日志格式化方式

当日志模块被添加至 Python 标准库时，只有一种格式化消息内容的方法即 %-formatting。 在那之后，Python 又增加了两种格式化方法: `string.Template` (在 Python 2.4 中新增) 和 `str.format()` (在 Python 2.6 中新增)。

日志（从 3.2 开始）为这两种格式化方式提供了更多支持。`Formatter` 类可以添加一个额外的可选关键字参数 `style`。它的默认值是 `'%'`，其他的值 `'{'` 和 `'$'` 也支持，对应了其他两种格式化样式。其保持了向后兼容（如您所愿），但通过显示指定样式参数，你可以指定格式化字符串的方式是使用 `str.format()` 或 `string.Template`。 这里是一个控制台会话的示例，展示了这些方式:

```
>>> import logging
>>> root = logging.getLogger()
>>> root.setLevel(logging.DEBUG)
>>> handler = logging.StreamHandler()
>>> bf = logging.Formatter('{asctime} {name} {levelname:8s} {message}',
...                        style='{')
>>> handler.setFormatter(bf)
>>> root.addHandler(handler)
>>> logger = logging.getLogger('foo.bar')
>>> logger.debug('This is a DEBUG message')
2010-10-28 15:11:55,341 foo.bar DEBUG    This is a DEBUG message
>>> logger.critical('This is a CRITICAL message')
2010-10-28 15:12:11,526 foo.bar CRITICAL This is a CRITICAL message
>>> df = logging.Formatter('$asctime $name ${levelname} $message',
...                        style='$')
>>> handler.setFormatter(df)
>>> logger.debug('This is a DEBUG message')
2010-10-28 15:13:06,924 foo.bar DEBUG This is a DEBUG message
>>> logger.critical('This is a CRITICAL message')
2010-10-28 15:13:11,494 foo.bar CRITICAL This is a CRITICAL message
>>>
```

请注意最终输出到日志的消息格式完全独立于单条日志消息的构造方式。 它仍然可以使用 %-formatting，如下所示:

```
>>> logger.error('This is an%s %s %s', 'other,', 'ERROR,', 'message')
2010-10-28 15:19:29,833 foo.bar ERROR This is another, ERROR, message
>>>
```

日志调用（`logger.debug()` 、`logger.info()` 等）接受的位置参数只会用于日志信息本身，而关键字参数仅用于日志调用的可选处理参数（如关键字参数 `exc_info` 表示应记录跟踪信息， `extra` 则标识了需要加入日志的额外上下文信息）。所以不能直接用 `str.format()` 或 `string.Template` 语法进行日志调用，因为日志包在内部使用 %-f 格式来合并格式串和参数变量。在保持向下兼容性时，这一点不会改变，因为已有代码中的所有日志调用都会使用%-f 格式串。

还有一种方法可以构建自己的日志信息，就是利用 {}- 和 $- 格式。回想一下，任意对象都可用为日志信息的格式串，日志包将会调用该对象的 `str()` 方法，以获取最终的格式串。不妨看下一下两个

类：

```
class BraceMessage:
    def __init__(self, fmt, /, *args, **kwargs):
        self.fmt = fmt
        self.args = args
        self.kwargs = kwargs

    def __str__(self):
        return self.fmt.format(*self.args, **self.kwargs)

class DollarMessage:
    def __init__(self, fmt, /, **kwargs):
        self.fmt = fmt
        self.kwargs = kwargs

    def __str__(self):
        from string import Template
        return Template(self.fmt).substitute(**self.kwargs)
```

上述两个类均可代替格式串，使得能用 {}- 或 $-formatting 构建最终的“日志信息”部分，这些信息将出现在格式化后的日志输出中，替换 %(message)s 或“{message}”或“$message”。每次写入日志时都要使用类名，有点不大实用，但如果用上 __ 之类的别名就相当合适了（双下划线 --- 不要与 _ 混淆，单下划线用作 gettext.gettext() 或相关函数的同义词/别名 ）。

Python 并没有上述两个类，当然复制粘贴到自己的代码中也很容易。用法可如下所示（假定在名为 `wherever` 的模块中声明）：

```
>>> from wherever import BraceMessage as __
>>> print(__('Message with {0} {name}', 2, name='placeholders'))
Message with 2 placeholders
>>> class Point: pass
...
>>> p = Point()
>>> p.x = 0.5
>>> p.y = 0.5
>>> print(__('Message with coordinates: ({point.x:.2f}, {point.y:.2f})',
...       point=p))
Message with coordinates: (0.50, 0.50)
>>> from wherever import DollarMessage as __
>>> print(__('Message with $num $what', num=2, what='placeholders'))
Message with 2 placeholders
>>>
```

上述示例用了 `print()` 演示格式化输出的过程，实际记录日志时当然会用类似 `logger.debug()` 的方法来应用。

需要注意的是使用这种方式不会对性能造成明显影响：实际的格式化工作不是在日志记录调用时发生的，而是在（如果）处理器即将把日志消息输出到日志时发生的。 因此，唯一可能令人困惑的不寻常之处在于包裹在格式字符串和参数外面的圆括号，而不仅仅是格式字符串。 这是因为 __ 标记只是对 *XXX*`Message` 类的构造器的调用的语法糖。

只要愿意，上述类似的效果即可用 `LoggerAdapter` 实现，如下例所示：

```
import logging

class Message:
    def __init__(self, fmt, args):
        self.fmt = fmt
        self.args = args

    def __str__(self):
        return self.fmt.format(*self.args)

class StyleAdapter(logging.LoggerAdapter):
    def log(self, level, msg, /, *args, stacklevel=1, **kwargs):
        if self.isEnabledFor(level):
            msg, kwargs = self.process(msg, kwargs)
            self.logger.log(level, Message(msg, args), **kwargs,
                            stacklevel=stacklevel+1)

logger = StyleAdapter(logging.getLogger(__name__))

def main():
    logger.debug('Hello, {}', 'world!')

if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG)
    main()
```

在用 Python 3.8 以上版本运行时上述脚本应该会将消息 `Hello, world!` 写入日志。

## 自定义 LogRecord

每条日志事件都由一个 `LogRecord` 实例表示。当某事件要记入日志并且没有被某级别过滤掉时，就会创建一个 `LogRecord` 对象，并将有关事件的信息填入，传给该日志对象的 handler（及其祖先，直至对象禁止向上传播为止）。在 Python 3.2 之前，只有两个地方会进行事件的创建：

- `Logger.makeRecord()`，在事件正常记入日志的过程中调用。这会直接调用 `LogRecord` 来创建一个实例。
- `makeLogRecord()`，调用时会带上一个字典参数，其中存放着要加入 LogRecord 的属性。这通常在通过网络接收到合适的字典时调用（如通过 `SocketHandler` 以 pickle 形式，或通过 `HTTPHandler` 以 JSON 形式）。

于是这意味着若要对 `LogRecord` 进行定制，必须进行下述某种操作。

- 创建 `Logger` 自定义子类，重写 `Logger.makeRecord()`，并在实例化所需日志对象之前用 `setLoggerClass()` 进行设置。
- 为日志对象添加 `Filter` 或 handler，当其 `filter()` 方法被调用时，会执行必要的定制操作。

比如说在有多个不同库要完成不同操作的场景下，第一种方式会有点笨拙。 每次都要尝试设置自己的 `Logger` 子类，而起作用的是最后一次尝试。

第二种方式在多数情况下效果都比较良好，但不允许你使用特殊化的 `LogRecord` 子类。 库开发者可以为他们的日志记录器设置合适的过滤器，但他们应当要记得每次引入新的日志记录器时都需如此（他们只需通过添加新的包或模块并执行以下操作即可）：

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

或许这样要顾及太多事情。开发人员还可以将过滤器附加到其顶级日志对象的 `NullHandler` 中，但如果应用程序开发人员将 handler 附加到较底层库的日志对象，则不会调用该过滤器 --- 所以 handler 输出的内容不会符合库开发人员的预期。

在 Python 3.2 以上版本中，`LogRecord` 的创建是通过工厂对象完成的，工厂对象可以指定。工厂对象只是一个可调用对象，可以用 `setLogRecordFactory()` 进行设置，并用 `getLogRecordFactory()` 进行查询。工厂对象的调用参数与 `LogRecord` 的构造函数相同，因为 `LogRecord` 是工厂对象的默认设置。

这种方式可以让自定义工厂对象完全控制 LogRecord 的创建过程。比如可以返回一个子类，或者在创建的日志对象中加入一些额外的属性，使用方式如下所示：

```
old_factory = logging.getLogRecordFactory()

def record_factory(*args, **kwargs):
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdecafbad
    return record

logging.setLogRecordFactory(record_factory)
```

这种模式允许不同的库将多个工厂对象链在一起，只要不会覆盖彼此的属性或标准属性，就不会出现意外。但应记住，工厂链中的每个节点都会增加日志操作的运行开销，本技术仅在采用 `Filter` 无法达到目标时才应使用。

## 子类化 QueueHandler 和 QueueListener - ZeroMQ 示例

### 子类 QueueHandler

你可以使用 `QueueHandler` 子类将消息发送给其他类型的队列，比如 ZeroMQ 'publish' 套接字。 在以下示例中，套接字将单独创建并传给处理器 (作为它的 'queue'):

```
import zmq   # 使用 pyzmq，这是 ZeroMQ 的 Python 绑定
import json  # 用于可移植地对记录进行序列化

ctx = zmq.Context()
sock = zmq.Socket(ctx, zmq.PUB)  # 或 zmq.PUSH，或其他适当的值
sock.bind('tcp://*:5556')        # 或任何值

class ZeroMQSocketHandler(QueueHandler):
    def enqueue(self, record):
        self.queue.send_json(record.__dict__)


handler = ZeroMQSocketHandler(sock)
```

当然还有其他方案，比如通过 hander 传入所需数据，以创建 socket:

```
class ZeroMQSocketHandler(QueueHandler):
    def __init__(self, uri, socktype=zmq.PUB, ctx=None):
```

```
        self.ctx = ctx or zmq.Context()
        socket = zmq.Socket(self.ctx, socktype)
        socket.bind(uri)
        super().__init__(socket)

    def enqueue(self, record):
        self.queue.send_json(record.__dict__)

    def close(self):
        self.queue.close()
```

子类 QueueListener

你还可以子类化 QueueListener 来从其他类型的队列中获取消息，比如从 ZeroMQ 'subscribe' 套接字。 下面是一个例子:

```
class ZeroMQSocketListener(QueueListener):
    def __init__(self, uri, /, *handlers, **kwargs):
        self.ctx = kwargs.get('ctx') or zmq.Context()
        socket = zmq.Socket(self.ctx, zmq.SUB)
        socket.setsockopt_string(zmq.SUBSCRIBE, '')  # 全部预订
        socket.connect(uri)
        super().__init__(socket, *handlers, **kwargs)

    def dequeue(self):
        msg = self.queue.recv_json()
        return logging.makeLogRecord(msg)
```

## 子类化 QueueHandler 和 QueueListener - pynng 示例

通过与上一节类似的方式，我们可以使用 pynng 来实现监听器和处理器，这个包是针对 NNG 的 Python 绑定，它被确定为 ZeroMQ 的精神后继者。 以下代码片段被用作演示 -- 你可以在安装了 `pynng` 的环境中测试它们。 为增加变化，我们先编写监听器。

子类 QueueListener

```
# listener.py
import json
import logging
import logging.handlers

import pynng

DEFAULT_ADDR = "tcp://localhost:13232"

interrupted = False

class NNGSocketListener(logging.handlers.QueueListener):

    def __init__(self, uri, /, *handlers, **kwargs):
        # 设置超时以允许中断，并打开一个
        # 订阅方套接字
        socket = pynng.Sub0(listen=uri, recv_timeout=500)
        # b'' 订阅将匹配所有主题
        topics = kwargs.pop('topics', None) or b''
        socket.subscribe(topics)
```

```
        # 我们将套接字视为一个队列
        super().__init__(socket, *handlers, **kwargs)

    def dequeue(self, block):
        data = None
        # 在未被打断且未从套接字接收数据时
        # 保持循环
        while not interrupted:
            try:
                data = self.queue.recv(block=block)
                break
            except pynng.Timeout:
                pass
            except pynng.Closed:  # 会在你按下 Ctrl-C 时发生
                break
        if data is None:
            return None
        # 获取从发布方发送的日志记录事件
        event = json.loads(data.decode('utf-8'))
        return logging.makeLogRecord(event)

    def enqueue_sentinel(self):
        # 在本实现中未被使用，因为套接字并不是
        # 真正的队列
        pass

logging.getLogger('pynng').propagate = False
listener = NNGSocketListener(DEFAULT_ADDR, logging.StreamHandler(), topics=b'')
listener.start()
print('Press Ctrl-C to stop.')
try:
    while True:
        pass
except KeyboardInterrupt:
    interrupted = True
finally:
    listener.stop()
```

子类 QueueHandler

```
# sender.py
import json
import logging
import logging.handlers
import time
import random

import pynng

DEFAULT_ADDR = "tcp://localhost:13232"

class NNGSocketHandler(logging.handlers.QueueHandler):

    def __init__(self, uri):
        socket = pynng.Pub0(dial=uri, send_timeout=500)
        super().__init__(socket)

    def enqueue(self, record):
        # Send the record as UTF-8 encoded JSON
```

```
        d = dict(record.__dict__)
        data = json.dumps(d)
        self.queue.send(data.encode('utf-8'))

    def close(self):
        self.queue.close()

logging.getLogger('pynng').propagate = False
handler = NNGSocketHandler(DEFAULT_ADDR)
# 确保进程 ID 在输出内容中
logging.basicConfig(level=logging.DEBUG,
                    handlers=[logging.StreamHandler(), handler],
                    format='%(levelname)-8s %(name)10s %(process)6s %(message)s')
levels = (logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR,
          logging.CRITICAL)
logger_names = ('myapp', 'myapp.lib1', 'myapp.lib2')
msgno = 1
while True:
    # 随机地选择日志记录器和层级并记录日志
    level = random.choice(levels)
    logger = logging.getLogger(random.choice(logger_names))
    logger.log(level, 'Message no. %5d' % msgno)
    msgno += 1
    delay = random.random() * 2 + 0.5
    time.sleep(delay)
```

你可以在不同的命令行 shell 中运行上面两个代码片段。 如果我们在一个 shell 中运行监听器并在两个不同的 shell 中运行发送器，我们将看到如下的结果。 在第一个发送器 shell 中：

```
$ python sender.py
DEBUG         myapp    613 Message no.     1
WARNING  myapp.lib2    613 Message no.     2
CRITICAL myapp.lib2    613 Message no.     3
WARNING  myapp.lib2    613 Message no.     4
CRITICAL myapp.lib1    613 Message no.     5
DEBUG         myapp    613 Message no.     6
CRITICAL myapp.lib1    613 Message no.     7
INFO     myapp.lib1    613 Message no.     8
(下略)
```

在第二个发送器 shell 中：

```
$ python sender.py
INFO     myapp.lib2    657 Message no.     1
CRITICAL myapp.lib2    657 Message no.     2
CRITICAL      myapp    657 Message no.     3
CRITICAL myapp.lib1    657 Message no.     4
INFO     myapp.lib1    657 Message no.     5
WARNING  myapp.lib2    657 Message no.     6
CRITICAL      myapp    657 Message no.     7
DEBUG    myapp.lib1    657 Message no.     8
(下略)
```

在监听器 shell 中：

```
$ python listener.py
Press Ctrl-C to stop.
```

```
DEBUG         myapp    613 Message no.     1
WARNING  myapp.lib2    613 Message no.     2
INFO     myapp.lib2    657 Message no.     1
CRITICAL myapp.lib2    613 Message no.     3
CRITICAL myapp.lib2    657 Message no.     2
CRITICAL      myapp    657 Message no.     3
WARNING  myapp.lib2    613 Message no.     4
CRITICAL myapp.lib1    613 Message no.     5
CRITICAL myapp.lib1    657 Message no.     4
INFO     myapp.lib1    657 Message no.     5
DEBUG         myapp    613 Message no.     6
WARNING  myapp.lib2    657 Message no.     6
CRITICAL      myapp    657 Message no.     7
CRITICAL myapp.lib1    613 Message no.     7
INFO     myapp.lib1    613 Message no.     8
DEBUG    myapp.lib1    657 Message no.     8
(下略)
```

如你所见，来自两个发送器进程的日志记录会在监听器的输出中交错出现。

## 基于字典进行日志配置的示例

以下是日志配置字典的一个示例——它取自 Django 项目的`文档 <https://docs.djangoproject.com/en/stable/topics/logging/#configuring-logging>`_。此字典将被传给 `dictConfig()` 以使配置生效:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'verbose': {
            'format': '{levelname} {asctime} {module} {process:d} {thread:d} {mess
            'style': '{',
        },
        'simple': {
            'format': '{levelname} {message}',
            'style': '{',
        },
    },
    'filters': {
        'special': {
            '()': 'project.logging.SpecialFilter',
            'foo': 'bar',
        },
    },
    'handlers': {
        'console': {
            'level': 'INFO',
            'class': 'logging.StreamHandler',
            'formatter': 'simple',
        },
        'mail_admins': {
            'level': 'ERROR',
            'class': 'django.utils.log.AdminEmailHandler',
            'filters': ['special']
        }
    },
```

```
    'loggers': {
        'django': {
            'handlers': ['console'],
            'propagate': True,
        },
        'django.request': {
            'handlers': ['mail_admins'],
            'level': 'ERROR',
            'propagate': False,
        },
        'myproject.custom': {
            'handlers': ['console', 'mail_admins'],
            'level': 'INFO',
            'filters': ['special']
        }
    }
}
```

有关本配置的更多信息，请参阅 Django 文档的 有关章节 。

## 利用 rotator 和 namer 自定义日志轮换操作

下面的可运行代码给出了你可以怎样定义命名器和轮换器的例子，其中演示了日志文件的 gzip 压缩过程:

```
import gzip
import logging
import logging.handlers
import os
import shutil

def namer(name):
    return name + ".gz"

def rotator(source, dest):
    with open(source, 'rb') as f_in:
        with gzip.open(dest, 'wb') as f_out:
            shutil.copyfileobj(f_in, f_out)
    os.remove(source)


rh = logging.handlers.RotatingFileHandler('rotated.log', maxBytes=128, backupCount
rh.rotator = rotator
rh.namer = namer

root = logging.getLogger()
root.setLevel(logging.INFO)
root.addHandler(rh)
f = logging.Formatter('%(asctime)s %(message)s')
rh.setFormatter(f)
for i in range(1000):
    root.info(f'Message no. {i + 1}')
```

运行此脚本后，你将看到六个新文件，其中五个是已压缩的:

```
$ ls rotated.log*
rotated.log       rotated.log.2.gz  rotated.log.4.gz
```

```
rotated.log.1.gz  rotated.log.3.gz  rotated.log.5.gz
$ zcat rotated.log.1.gz
2023-01-20 02:28:17,767 Message no. 996
2023-01-20 02:28:17,767 Message no. 997
2023-01-20 02:28:17,767 Message no. 998
```

## 更加详细的多道处理示例

以下可运行的示例显示了如何利用配置文件在多进程中应用日志。这些配置相当简单，但足以说明如何在真实的多进程场景中实现较为复杂的配置。

上述示例中，主进程产生一个侦听器进程和一些工作进程。每个主进程、侦听器进程和工作进程都有三种独立的日志配置（工作进程共享同一套配置）。大家可以看到主进程的日志记录过程、工作线程向 QueueHandler 写入日志的过程，以及侦听器实现 QueueListener 和较为复杂的日志配置，如何将由队列接收到的事件分发给配置指定的 handler。请注意，这些配置纯粹用于演示，但应该能调整代码以适用于自己的场景。

以下是代码——但愿文档字符串和注释能有助于理解其工作原理：

```
import logging
import logging.config
import logging.handlers
from multiprocessing import Process, Queue, Event, current_process
import os
import random
import time

class MyHandler:
    """
    A simple handler for logging events. It runs in the listener process and
    dispatches events to loggers based on the name in the received record,
    which then get dispatched, by the logging system, to the handlers
    configured for those loggers.
    """

    def handle(self, record):
        if record.name == "root":
            logger = logging.getLogger()
        else:
            logger = logging.getLogger(record.name)

        if logger.isEnabledFor(record.levelno):
            # 进程名称经过变换以演示是由监听器来执行
            # 记录日志到文件和控制台
            record.processName = '%s (for %s)' % (current_process().name, record.p
            logger.handle(record)

def listener_process(q, stop_event, config):
    """
    This could be done in the main process, but is just done in a separate
    process for illustrative purposes.

    This initialises logging according to the specified configuration,
    starts the listener and waits for the main process to signal completion
    via the event. The listener is then stopped, and the process exits.
    """
```

```
    logging.config.dictConfig(config)
    listener = logging.handlers.QueueListener(q, MyHandler())
    listener.start()
    if os.name == 'posix':
        # 在 POSIX 系统上，setup 日志记录器将会在
        # 父进程中完成配置，但应当在 dictConfig 调用
        # 之后即已被禁用。
        # 在 Windows 上，由于不会使用 fork，setup 日志记录器
        # 将不会在子进程中退出，因此它将被创建并且显示消息
        # —— 对应 "if posix" 子句。
        logger = logging.getLogger('setup')
        logger.critical('Should not appear, because of disabled logger ...')
    stop_event.wait()
    listener.stop()

def worker_process(config):
    """
    A number of these are spawned for the purpose of illustration. In
    practice, they could be a heterogeneous bunch of processes rather than
    ones which are identical to each other.

    This initialises logging according to the specified configuration,
    and logs a hundred messages with random levels to randomly selected
    loggers.

    A small sleep is added to allow other processes a chance to run. This
    is not strictly needed, but it mixes the output from the different
    processes a bit more than if it's left out.
    """
    logging.config.dictConfig(config)
    levels = [logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR,
              logging.CRITICAL]
    loggers = ['foo', 'foo.bar', 'foo.bar.baz',
               'spam', 'spam.ham', 'spam.ham.eggs']
    if os.name == 'posix':
        # 在 POSIX 系统上，setup 日志记录器将会在
        # 父进程中完成配置，但应当在 dictConfig 调用
        # 之后已被禁用。
        # 在 Windows 上，由于不会使用 fork，setup 日志记录器
        # 将不会在子进程中退出，因此它将被创建并且显示消息
        # —— 对应 "if posix" 子句。
        logger = logging.getLogger('setup')
        logger.critical('Should not appear, because of disabled logger ...')
    for i in range(100):
        lvl = random.choice(levels)
        logger = logging.getLogger(random.choice(loggers))
        logger.log(lvl, 'Message no. %d', i)
        time.sleep(0.01)

def main():
    q = Queue()
    # 主进程将获得一个打印到控制台的简单配置。
    config_initial = {
        'version': 1,
        'handlers': {
            'console': {
                'class': 'logging.StreamHandler',
                'level': 'INFO'
            }
        },
```

```
        'root': {
            'handlers': ['console'],
            'level': 'DEBUG'
        }
    }
    # 工作进程配置就是一个附加到根日志记录器的 QueueHandler，
    # 它允许所有消息被发送至队列 。
    # 我们禁用现有的日志记录器以禁用在父进程中使用的 "setup"
    # 日志记录器。 这在 POSIX 中是必需的因为日志记录器将会在
    # fork() 之后出现在子进程中。
    config_worker = {
        'version': 1,
        'disable_existing_loggers': True,
        'handlers': {
            'queue': {
                'class': 'logging.handlers.QueueHandler',
                'queue': q
            }
        },
        'root': {
            'handlers': ['queue'],
            'level': 'DEBUG'
        }
    }
    # 监听器进程配置显示可以使用日志记录配置的
    # 完整适应性以便以你希望的方式将事件分发给
    # 处理器。
    # 我们禁用现有的日志记录器以禁用在父进程中使用的
    # "setup" 日志记录器。 这在 POSIX 中是必需的因为
    # 日志记录器将会在 fork() 之后出现在子进程中。
    config_listener = {
        'version': 1,
        'disable_existing_loggers': True,
        'formatters': {
            'detailed': {
                'class': 'logging.Formatter',
                'format': '%(asctime)s %(name)-15s %(levelname)-8s %(processName)-
            },
            'simple': {
                'class': 'logging.Formatter',
                'format': '%(name)-15s %(levelname)-8s %(processName)-10s %(messag
            }
        },
        'handlers': {
            'console': {
                'class': 'logging.StreamHandler',
                'formatter': 'simple',
                'level': 'INFO'
            },
            'file': {
                'class': 'logging.FileHandler',
                'filename': 'mplog.log',
                'mode': 'w',
                'formatter': 'detailed'
            },
            'foofile': {
                'class': 'logging.FileHandler',
                'filename': 'mplog-foo.log',
                'mode': 'w',
                'formatter': 'detailed'
```

```
            },
            'errors': {
                'class': 'logging.FileHandler',
                'filename': 'mplog-errors.log',
                'mode': 'w',
                'formatter': 'detailed',
                'level': 'ERROR'
            }
        },
        'loggers': {
            'foo': {
                'handlers': ['foofile']
            }
        },
        'root': {
            'handlers': ['console', 'file', 'errors'],
            'level': 'DEBUG'
        }
    }
    # 记录一些初始事件，以便显示父进程中的日志记录
    # 工作正常。
    logging.config.dictConfig(config_initial)
    logger = logging.getLogger('setup')
    logger.info('About to create workers ...')
    workers = []
    for i in range(5):
        wp = Process(target=worker_process, name='worker %d' % (i + 1),
                     args=(config_worker,))
        workers.append(wp)
        wp.start()
        logger.info('Started worker: %s', wp.name)
    logger.info('About to create listener ...')
    stop_event = Event()
    lp = Process(target=listener_process, name='listener',
                 args=(q, stop_event, config_listener))
    lp.start()
    logger.info('Started listener')
    # 我们现在要等待工作进程完成其工作。
    for wp in workers:
        wp.join()
    # 工作进程全部结束，现在可以停止监听。
    # 父进程中的日志记录仍然正常进行。
    logger.info('Telling listener to stop ...')
    stop_event.set()
    lp.join()
    logger.info('All done.')

if __name__ == '__main__':
    main()
```

## 在发送给 SysLogHandler 的信息中插入一个 BOM。

RFC 5424 要求，Unicode 信息应采用字节流形式发送到系统 syslog 守护程序，字节流结构如下所示：可选的纯 ASCII部分，后跟 UTF-8 字节序标记（BOM），然后是采用 UTF-8 编码的 Unicode。（参见 相关规范 。）

在 Python 3.1 的 `SysLogHandler` 中，已加入了在日志信息中插入 BOM 的代码，但不幸的是，代码并不正确，BOM 出现在了日志信息的开头，因此在它之前就不允许出现纯 ASCII 内容了。

由于无法正常工作， Python 3.2.4 以上版本已删除了出错的插入 BOM 代码。但已有版本的代码不会被替换，若要生成与 **RFC 5424** 兼容的日志信息，包括一个 BOM 符，前面有可选的纯 ASCII 字节流，后面为 UTF-8 编码的任意 Unicode，那么 需要执行以下操作：

1. 为 `SysLogHandler` 实例串上一个 `Formatter` 实例，格式串可如下：

   ```
   'ASCII section\ufeffUnicode section'
   ```

   用 UTF-8 编码时，Unicode 码位 U+FEFF 将会编码为 UTF-8 BOM——字节串 `b'\xef\xbb\xbf'` 。

2. 用任意占位符替换 ASCII 部分，但要保证替换之后的数据一定是 ASCII 码（这样在 UTF-8 编码后就会维持不变）。

3. 用任意占位符替换 Unicode 部分；如果替换后的数据包含超出 ASCII 范围的字符，没问题——他们将用 UTF-8 进行编码。

`SysLogHandler` *将* 对格式化后的日志信息进行 UTF-8 编码。如果遵循上述规则，应能生成符合 **RFC 5424** 的日志信息。否则，日志记录过程可能不会有什么反馈，但日志信息将不与 RFC 5424 兼容，syslog 守护程序可能会有出错反应。

## 结构化日志的实现代码

大多数日志信息是供人阅读的，所以机器解析起来并不容易，但某些时候可能希望以结构化的格式输出，以 *能够* 被程序解析（无需用到复杂的正则表达式）。这可以直接用 logging 包实现。实现方式有很多，以下是一种比较简单的方案，利用 JSON 以机器可解析的方式对事件信息进行序列化：

```
import json
import logging

class StructuredMessage:
    def __init__(self, message, /, **kwargs):
        self.message = message
        self.kwargs = kwargs

    def __str__(self):
        return '%s >>> %s' % (self.message, json.dumps(self.kwargs))

_ = StructuredMessage   # 可选项，用于提升可读性

logging.basicConfig(level=logging.INFO, format='%(message)s')
logging.info(_('message 1', foo='bar', bar='baz', num=123, fnum=123.456))
```

上述代码运行后的结果是：

```
message 1 >>> {"fnum": 123.456, "num": 123, "bar": "baz", "foo": "bar"}
```

请注意，根据 Python 版本的不同，各项数据的输出顺序可能会不一样。

若需进行更为定制化的处理，可以使用自定义 JSON 编码对象，下面给出完整示例：

```
import json
import logging


class Encoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, set):
            return tuple(o)
        elif isinstance(o, str):
            return o.encode('unicode_escape').decode('ascii')
        return super().default(o)

class StructuredMessage:
    def __init__(self, message, /, **kwargs):
        self.message = message
        self.kwargs = kwargs

    def __str__(self):
        s = Encoder().encode(self.kwargs)
        return '%s >>> %s' % (self.message, s)

_ = StructuredMessage   # 可选项，用于提升可读性

def main():
    logging.basicConfig(level=logging.INFO, format='%(message)s')
    logging.info(_('message 1', set_value={1, 2, 3}, snowman='\u2603'))

if __name__ == '__main__':
    main()
```

上述代码运行后的结果是：

```
message 1 >>> {"snowman": "\u2603", "set_value": [1, 2, 3]}
```

请注意，根据 Python 版本的不同，各项数据的输出顺序可能会不一样。

## 利用 dictConfig() 自定义 handler

有时需要以特定方式自定义日志 handler，如果采用 dictConfig()，可能无需生成子类就可以做到。比如要设置日志文件的所有权。在 POSIX 上，可以利用 shutil.chown() 轻松完成，但 stdlib 中的文件 handler 并不提供内置支持。于是可以用普通函数自定义 handler 的创建，例如：

```
def owned_file_handler(filename, mode='a', encoding=None, owner=None):
    if owner:
        if not os.path.exists(filename):
            open(filename, 'a').close()
        shutil.chown(filename, *owner)
    return logging.FileHandler(filename, mode, encoding)
```

然后，你可以在传给 dictConfig() 的日志配置中指定通过调用此函数来创建日志处理程序：

```
LOGGING = {
    'version': 1,
```

```
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': '%(asctime)s %(levelname)s %(name)s %(message)s'
        },
    },
    'handlers': {
        'file':{
            # 下面的值将从该字典中弹出并被用来
            # 创建处理器，设置处理器的层级及其
            # 格式化器：
            '()': owned_file_handler,
            'level':'DEBUG',
            'formatter': 'default',
            # 下面的值将以关键字参数形式传给
            # 处理器调用方的可调用对象。
            'owner': ['pulse', 'pulse'],
            'filename': 'chowntest.log',
            'mode': 'w',
            'encoding': 'utf-8',
        },
    },
    'root': {
        'handlers': ['file'],
        'level': 'DEBUG',
    },
}
```

出于演示目的，以下示例设置用户和用户组为 `pulse`。代码置于一个可运行的脚本文件 `chowntest.py` 中：

```
import logging, logging.config, os, shutil

def owned_file_handler(filename, mode='a', encoding=None, owner=None):
    if owner:
        if not os.path.exists(filename):
            open(filename, 'a').close()
        shutil.chown(filename, *owner)
    return logging.FileHandler(filename, mode, encoding)

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': '%(asctime)s %(levelname)s %(name)s %(message)s'
        },
    },
    'handlers': {
        'file':{
            # 下面的值将从此字典中被弹出并被用来
            # 创建处理器、设置处理器的层级
            # 及其格式化器。
            '()': owned_file_handler,
            'level':'DEBUG',
            'formatter': 'default',
            # 下面的值将以关键字参数形式传给处理器的
            # 创建方可调用对象。
```

```
            'owner': ['pulse', 'pulse'],
            'filename': 'chowntest.log',
            'mode': 'w',
            'encoding': 'utf-8',
        },
    },
    'root': {
        'handlers': ['file'],
        'level': 'DEBUG',
    },
}

logging.config.dictConfig(LOGGING)
logger = logging.getLogger('mylogger')
logger.debug('A debug message')
```

可能需要 `root` 权限才能运行：

```
$ sudo python3.3 chowntest.py
$ cat chowntest.log
2013-11-05 09:34:51,128 DEBUG mylogger A debug message
$ ls -l chowntest.log
-rw-r--r-- 1 pulse pulse 55 2013-11-05 09:34 chowntest.log
```

请注意此示例用的是 Python 3.3，因为 `shutil.chown()` 是从此版本开始出现的。 此方式应当适用于任何支持 `dictConfig()` 的 Python 版本 —— 例如 Python 2.7, 3.2 或更新的版本。 对于 3.3 之前的版本，你应当使用 `os.chown()` 之类的函数来实现实际的所有权修改。

实际应用中，handler 的创建函数可能位于项目的工具模块中。以下配置：

```
'()': owned_file_handler,
```

应使用：

```
'()': 'ext://project.util.owned_file_handler',
```

这里的 `project.util` 可以换成函数所在包的实际名称。 在上述的可用脚本中，应该可以使用 `'ext://__main__.owned_file_handler'`。 在这里，实际的可调用对象是由 `dictConfig()` 从 `ext://` 说明中解析出来的。

上述示例还指明了其他的文件修改类型的实现方案 —— 比如同样利用 `os.chmod()` 设置 POSIX 访问权限位。

当然，以上做法也可以扩展到 `FileHandler` 之外的其他类型的 handler ——比如某个轮换文件 handler，或类型完全不同的其他 handler。

## 生效于整个应用程序的格式化样式

在 Python 3.2 中，`Formatter` 增加了一个 `style` 关键字形参，它默认为 `%` 以便向下兼容，但是允许采用 `{` 或 `$` 来支持 `str.format()` 和 `string.Template` 所支持的格式化方式。 请注意此形参控制着用用于最终输出到日志的日志消息格式，并且与单独日志消息的构造方式完全无关。

日志调用 (debug(), info() 等) 只接受包含实际日志消息自身的位置参数，而关键字参数仅用于确定如何处理日志调用的选项 (例如 exc_info 关键字参数表示应将回溯信息记入日志，而 extra 关键字参数则指定要添加到日志的额外上下文信息)。 所以你不能直接使用 str.format() 或 string.Template 语法来直接执行日志调用，因为 logging 包在内部是使用 % 格式符来合并格式字符串和可变参数的。 这一点不应被改变以保持向下兼容性，因为现有代码中所有的日志调用都将使用 % 格式化字符串。

有人建议将格式化样式与特定的日志对象进行关联，但其实也会遇到向下兼容的问题，因为已有代码可能用到了某日志对象并采用了 %-f 格式串。

为了让第三方库和自编代码都能够交互使用日志功能，需要决定在单次日志记录调用级别采用什么格式。于是就出现了其他几种格式化样式方案。

### LogRecord 工厂的用法

在 Python 3.2 中，伴随着 Formatter 的上述变化，logging 包增加了允许用户使用 setLogRecordFactory() 函数来。设置自己的 LogRecord 子类的功能。 你可以使用此功能来设置自己的 LogRecord 子类，它会通过重写 getMessage() 方法来完成适当的操作。 msg % args 格式化是在此方法的基类实现中进行的，你可以在那里用你自己的格式化操作来替换；但是，你应当注意要支持全部的格式化样式并允许将 %-formatting 作为默认样式，以确保与其他代码进行配合。 还应当注意调用 str(self.msg)，正如基类实现所做的一样。

更多信息请参阅 setLogRecordFactory() 和 LogRecord 的参考文档。

### 自定义信息对象的使用

另一种方案可能更为简单，可以利用 {}- 和 $- 格式构建自己的日志消息。大家或许还记得（来自 使用任意对象作为消息），可以用任意对象作为日志信息的格式串，日志包将调用该对象上 str() 获取实际的格式串。看下以下两个类：

```
class BraceMessage:
    def __init__(self, fmt, /, *args, **kwargs):
        self.fmt = fmt
        self.args = args
        self.kwargs = kwargs

    def __str__(self):
        return self.fmt.format(*self.args, **self.kwargs)

class DollarMessage:
    def __init__(self, fmt, /, **kwargs):
        self.fmt = fmt
        self.kwargs = kwargs

    def __str__(self):
        from string import Template
        return Template(self.fmt).substitute(**self.kwargs)
```

以上两个类均都可用于替代格式串，以便用 {}- 或 $-formatting 构建实际的“日志信息”部分，此部分将出现在格式化后的日志输出中，替换 %(message)s 、“{message}”或“$message”。每次要写入日志

时都使用类名，如果觉得使用不便，可以采用 `M` 或 `_` 之类的别名（如果将 `_` 用于本地化操作，则可用 `__`）。

下面给出示例。 首先用 `str.format()` 进行格式化：

```
>>> __ = BraceMessage
>>> print(__('Message with {0} {1}', 2, 'placeholders'))
Message with 2 placeholders
>>> class Point: pass
...
>>> p = Point()
>>> p.x = 0.5
>>> p.y = 0.5
>>> print(__('Message with coordinates: ({point.x:.2f}, {point.y:.2f})', point=p))
Message with coordinates: (0.50, 0.50)
```

然后，用 `string.Template` 格式化：

```
>>> __ = DollarMessage
>>> print(__('Message with $num $what', num=2, what='placeholders'))
Message with 2 placeholders
>>>
```

需要注意的是使用这种方式不会对性能造成明显影响：实际的格式化工作不是在日志记录调用时发生的，而是在（如果）处理器即将把日志消息输出到日志时发生的。 因此，唯一可能令人困惑的不寻常之处在于包裹在格式字符串和参数外面的圆括号，而不仅仅是格式字符串。 这是因为 _ 符号只是对上面显示的 *XXX*`Message` 类的构造器的调用的语法糖。

## 利用 `dictConfig()` 定义过滤器

用 `dictConfig()` *可以* 对日志过滤器进行设置，尽管乍一看做法并不明显（所以才需要本秘籍）。由于 `Filter` 是标准库中唯一的日志过滤器类，不太可能满足众多的要求（它只是作为基类存在），通常需要定义自己的 `Filter` 子类，并重写 `filter()` 方法。为此，请在过滤器的配置字典中设置 `()` 键，指定要用于创建过滤器的可调用对象（最明显可用的就是给出一个类，但也可以提供任何一个可调用对象，只要能返回 `Filter` 实例即可）。下面是一个完整的例子：

```
import logging
import logging.config
import sys

class MyFilter(logging.Filter):
    def __init__(self, param=None):
        self.param = param

    def filter(self, record):
        if self.param is None:
            allow = True
        else:
            allow = self.param not in record.msg
        if allow:
            record.msg = 'changed: ' + record.msg
        return allow
```

```
LOGGING = {
    'version': 1,
    'filters': {
        'myfilter': {
            '()': MyFilter,
            'param': 'noshow',
        }
    },
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
            'filters': ['myfilter']
        }
    },
    'root': {
        'level': 'DEBUG',
        'handlers': ['console']
    },
}

if __name__ == '__main__':
    logging.config.dictConfig(LOGGING)
    logging.debug('hello')
    logging.debug('hello - noshow')
```

以上示例展示了将配置数据传给构造实例的可调用对象，形式是关键字参数。运行后将会输出：

```
changed: hello
```

这说明过滤器按照配置的参数生效了。

需要额外注意的地方：

- 如果在配置中无法直接引用可调用对象（比如位于不同的模块中，并且不能在配置字典所在的位置直接导入），则可以采用 `ext://...` 的形式，正如 访问外部对象 所述。例如，在上述示例中可以使用文本 `'ext://__main__.MyFilter'` 而不是 `MyFilter` 对象。
- 与过滤器一样，上述技术还可用于配置自定义 handler 和格式化对象。有关如何在日志配置中使用用户自定义对象的更多信息，请参阅 用户定义对象，以及上述 利用 dictConfig() 自定义 handler 的其他指南。

## 异常信息的自定义格式化

有时可能需要设置自定义的异常信息格式——考虑到会用到参数，假定要让每条日志事件只占一行，即便存在异常信息也一样。这可以用自定义格式化类来实现，如下所示：

```
import logging

class OneLineExceptionFormatter(logging.Formatter):
    def formatException(self, exc_info):
        """
        Format an exception so that it prints on a single line.
        """
        result = super().formatException(exc_info)
        return repr(result)  # 或格式化为任何你想要的单行内容
```

```
    def format(self, record):
        s = super().format(record)
        if record.exc_text:
            s = s.replace('\n', '') + '|'
        return s

def configure_logging():
    fh = logging.FileHandler('output.txt', 'w')
    f = OneLineExceptionFormatter('%(asctime)s|%(levelname)s|%(message)s|',
                                  '%d/%m/%Y %H:%M:%S')
    fh.setFormatter(f)
    root = logging.getLogger()
    root.setLevel(logging.DEBUG)
    root.addHandler(fh)

def main():
    configure_logging()
    logging.info('Sample message')
    try:
        x = 1 / 0
    except ZeroDivisionError as e:
        logging.exception('ZeroDivisionError: %s', e)

if __name__ == '__main__':
    main()
```

运行后将会生成只有两行信息的文件：

```
28/01/2015 07:21:23|INFO|Sample message|
28/01/2015 07:21:23|ERROR|ZeroDivisionError: division by zero|'Traceback (most rec
```

虽然上述处理方式很简单，但也给出了根据喜好对异常信息进行格式化输出的方案。或许 `traceback` 模块能满足更专门的需求。

## 语音播报日志信息

有时可能需要以声音的形式呈现日志消息。如果系统自带了文本转语音 （TTS）功能，即便没与 Python 关联也很容易做到。大多数 TTS 系统都有一个可运行的命令行程序，在 handler 中可以用 `subprocess` 进行调用。这里假定 TTS 命令行程序不会与用户交互，或需要很长时间才会执行完毕，写入日志的信息也不会多到影响用户查看，并且可以接受每次播报一条信息，以下示例实现了等一条信息播完再处理下一条，可能会导致其他 handler 的等待。这个简短示例仅供演示，假定 `espeak` TTS 包已就绪：

```
import logging
import subprocess
import sys

class TTSHandler(logging.Handler):
    def emit(self, record):
        msg = self.format(record)
        # 以女性的英语语音慢速地说话
        cmd = ['espeak', '-s150', '-ven+f3', msg]
        p = subprocess.Popen(cmd, stdout=subprocess.PIPE,
                             stderr=subprocess.STDOUT)
```

```
        # 等待程序结束
        p.communicate()

def configure_logging():
    h = TTSHandler()
    root = logging.getLogger()
    root.addHandler(h)
    # 默认格式化器简单地返回消息
    root.setLevel(logging.DEBUG)

def main():
    logging.info('Hello')
    logging.debug('Goodbye')

if __name__ == '__main__':
    configure_logging()
    sys.exit(main())
```

运行后将会以女声播报“Hello”和“Goodbye”。

当然，上述方案也适用于其他 TTS 系统，甚至可以通过利用命令行运行的外部程序来处理消息。

## 缓冲日志消息并有条件地输出它们

在某些情况下，你可能希望在临时区域中记录日志消息，并且只在发生某种特定的情况下才输出它们。 例如，你可能希望起始在函数中记录调试事件，如果函数执行完成且没有错误，你不希望输出收集的调试信息以避免造成日志混乱，但如果出现错误，那么你希望所有调试以及错误消息被输出。

下面是一个示例，展示如何在你的日志记录函数上使用装饰器以实现这一功能。该示例使用 `logging.handlers.MemoryHandler` ，它允许缓冲已记录的事件直到某些条件发生，缓冲的事件才会被刷新（`flushed`） - 传递给另一个处理程序（`target` handler）进行处理。 默认情况下，`MemoryHandler` 在其缓冲区被填满时被刷新，或者看到一个级别大于或等于指定阈值的事件。 如果想要自定义刷新行为，你可以通过更专业的 `MemoryHandler` 子类来使用这个秘诀。

这个示例脚本有一个简单的函数 `foo` ，它只是在所有的日志级别中循环运行，写到 `sys.stderr` ，说明它要记录在哪个级别上，然后在这个级别上实际记录一个消息。你可以给 `foo` 传递一个参数，如果为 true ，它将在ERROR和CRITICAL级别记录，否则，它只在DEBUG、INFO和WARNING级别记录。

脚本只是使用了一个装饰器来装饰 `foo`，这个装饰器将记录执行所需的条件。装饰器使用一个记录器作为参数，并在调用被装饰的函数期间附加一个内存处理程序。装饰器可以使用目标处理程序、记录级别和缓冲区的容量（缓冲记录的数量）来附加参数。这些参数分别默认为写入 `sys.stderr` 的 `StreamHandler` ， `logging.ERROR` 和 `100`。

以下是脚本：

```
import logging
from logging.handlers import MemoryHandler
import sys

logger = logging.getLogger(__name__)
```

```
logger.addHandler(logging.NullHandler())

def log_if_errors(logger, target_handler=None, flush_level=None, capacity=None):
    if target_handler is None:
        target_handler = logging.StreamHandler()
    if flush_level is None:
        flush_level = logging.ERROR
    if capacity is None:
        capacity = 100
    handler = MemoryHandler(capacity, flushLevel=flush_level, target=target_handle

    def decorator(fn):
        def wrapper(*args, **kwargs):
            logger.addHandler(handler)
            try:
                return fn(*args, **kwargs)
            except Exception:
                logger.exception('call failed')
                raise
            finally:
                super(MemoryHandler, handler).flush()
                logger.removeHandler(handler)
        return wrapper

    return decorator

def write_line(s):
    sys.stderr.write('%s\n' % s)

def foo(fail=False):
    write_line('about to log at DEBUG ...')
    logger.debug('Actually logged at DEBUG')
    write_line('about to log at INFO ...')
    logger.info('Actually logged at INFO')
    write_line('about to log at WARNING ...')
    logger.warning('Actually logged at WARNING')
    if fail:
        write_line('about to log at ERROR ...')
        logger.error('Actually logged at ERROR')
        write_line('about to log at CRITICAL ...')
        logger.critical('Actually logged at CRITICAL')
    return fail

decorated_foo = log_if_errors(logger)(foo)

if __name__ == '__main__':
    logger.setLevel(logging.DEBUG)
    write_line('Calling undecorated foo with False')
    assert not foo(False)
    write_line('Calling undecorated foo with True')
    assert foo(True)
    write_line('Calling decorated foo with False')
    assert not decorated_foo(False)
    write_line('Calling decorated foo with True')
    assert decorated_foo(True)
```

运行此脚本时，应看到以下输出：

```
Calling undecorated foo with False
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
Calling undecorated foo with True
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
about to log at ERROR ...
about to log at CRITICAL ...
Calling decorated foo with False
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
Calling decorated foo with True
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
about to log at ERROR ...
Actually logged at DEBUG
Actually logged at INFO
Actually logged at WARNING
Actually logged at ERROR
about to log at CRITICAL ...
Actually logged at CRITICAL
```

如你所见，实际日志记录输出仅在消息等级为ERROR或更高的事件时发生，但在这种情况下，任何之前较低消息等级的事件还会被记录。

你当然可以使用传统的装饰方法:

```
@log_if_errors(logger)
def foo(fail=False):
    ...
```

## 将日志消息发送至电子邮件，附带缓存支持

为演示如何通过电子邮件发送日志消息，让每封电子邮件发送指定数量的日志消息，你可以子类化 `BufferingHandler`。 对于下面的例子，你可以继续调整以适合你自己的特定需求，它提供了简单的测试代码来允许你附带命令行参数运行该脚本来指定你需要通过 SMTP 发送的内容。 （请附带 `-h` 参数运行已下载的脚本来查看必须的和可选的参数。）

```
import logging
import logging.handlers
import smtplib

class BufferingSMTPHandler(logging.handlers.BufferingHandler):
    def __init__(self, mailhost, port, username, password, fromaddr, toaddrs,
                 subject, capacity):
        logging.handlers.BufferingHandler.__init__(self, capacity)
        self.mailhost = mailhost
        self.mailport = port
        self.username = username
        self.password = password
        self.fromaddr = fromaddr
```

```
        if isinstance(toaddrs, str):
            toaddrs = [toaddrs]
        self.toaddrs = toaddrs
        self.subject = subject
        self.setFormatter(logging.Formatter("%(asctime)s %(levelname)-5s %(message

    def flush(self):
        if len(self.buffer) > 0:
            try:
                smtp = smtplib.SMTP(self.mailhost, self.mailport)
                smtp.starttls()
                smtp.login(self.username, self.password)
                msg = "From: %s\r\nTo: %s\r\nSubject: %s\r\n\r\n" % (self.fromaddr
                for record in self.buffer:
                    s = self.format(record)
                    msg = msg + s + "\r\n"
                smtp.sendmail(self.fromaddr, self.toaddrs, msg)
                smtp.quit()
            except Exception:
                if logging.raiseExceptions:
                    raise
            self.buffer = []

if __name__ == '__main__':
    import argparse

    ap = argparse.ArgumentParser()
    aa = ap.add_argument
    aa('host', metavar='HOST', help='SMTP server')
    aa('--port', '-p', type=int, default=587, help='SMTP port')
    aa('user', metavar='USER', help='SMTP username')
    aa('password', metavar='PASSWORD', help='SMTP password')
    aa('to', metavar='TO', help='Addressee for emails')
    aa('sender', metavar='SENDER', help='Sender email address')
    aa('--subject', '-s',
       default='Test Logging email from Python logging module (buffering)',
       help='Subject of email')
    options = ap.parse_args()
    logger = logging.getLogger()
    logger.setLevel(logging.DEBUG)
    h = BufferingSMTPHandler(options.host, options.port, options.user,
                             options.password, options.sender,
                             options.to, options.subject, 10)
    logger.addHandler(h)
    for i in range(102):
        logger.info("Info index = %d", i)
    h.flush()
    h.close()
```

如果你运行此脚本并且你的 SMTP 服务器已正确设置，你将发现它会向你指定的地址发出十一封电子邮件。 前十封邮件每封各有十条日志消息，第十一封将有两条消息。 如脚本所指定的总计为 102 条消息。

## 通过配置使用UTC (GMT) 格式化时间

有时你会想要使用 UTC 时间格式，这可以用 `UTCFormatter` 这样的类来实现，如下所示:

```
import logging
import time

class UTCFormatter(logging.Formatter):
    converter = time.gmtime
```

然后你可以在你的代码中使用 `UTCFormatter`，而不是 `Formatter`。 如果你想通过配置来实现这一功能，你可以使用 `dictConfig()` API 来完成，该方法在以下完整示例中展示:

```
import logging
import logging.config
import time

class UTCFormatter(logging.Formatter):
    converter = time.gmtime

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'utc': {
            '()': UTCFormatter,
            'format': '%(asctime)s %(message)s',
        },
        'local': {
            'format': '%(asctime)s %(message)s',
        }
    },
    'handlers': {
        'console1': {
            'class': 'logging.StreamHandler',
            'formatter': 'utc',
        },
        'console2': {
            'class': 'logging.StreamHandler',
            'formatter': 'local',
        },
    },
    'root': {
        'handlers': ['console1', 'console2'],
   }
}

if __name__ == '__main__':
    logging.config.dictConfig(LOGGING)
    logging.warning('The local time is %s', time.asctime())
```

脚本会运行输出类似下面的内容:

```
2015-10-17 12:53:29,501 The local time is Sat Oct 17 13:53:29 2015
2015-10-17 13:53:29,501 The local time is Sat Oct 17 13:53:29 2015
```

展示了如何将时间格式化为本地时间和UTC两种形式，其中每种形式对应一个日志处理器 。

## 使用上下文管理器的可选的日志记录

有时候，我们需要暂时更改日志配置，并在执行某些操作后将其还原。为此，上下文管理器是实现保存和恢复日志上下文的最明显的方式。这是一个关于上下文管理器的简单例子，它允许你在上下文管理器的作用域内更改日志记录等级以及增加日志处理器：

```
import logging
import sys

class LoggingContext:
    def __init__(self, logger, level=None, handler=None, close=True):
        self.logger = logger
        self.level = level
        self.handler = handler
        self.close = close

    def __enter__(self):
        if self.level is not None:
            self.old_level = self.logger.level
            self.logger.setLevel(self.level)
        if self.handler:
            self.logger.addHandler(self.handler)

    def __exit__(self, et, ev, tb):
        if self.level is not None:
            self.logger.setLevel(self.old_level)
        if self.handler:
            self.logger.removeHandler(self.handler)
        if self.handler and self.close:
            self.handler.close()
        # 隐式地返回 None => 不捕获异常
```

如果指定上下文管理器的日志记录等级属性，则在上下文管理器的with语句所涵盖的代码中，日志记录器的记录等级将临时设置为上下文管理器所配置的日志记录等级。 如果指定上下文管理的日志处理器属性，则该句柄在进入上下文管理器的上下文时添加到记录器中，并在退出时被删除。 如果你再也不需要该日志处理器时，你可以让上下文管理器在退出上下文管理器的上下文时关闭它。

为了说明它是如何工作的，我们可以在上面添加以下代码块:

```
if __name__ == '__main__':
    logger = logging.getLogger('foo')
    logger.addHandler(logging.StreamHandler())
    logger.setLevel(logging.INFO)
    logger.info('1. This should appear just once on stderr.')
    logger.debug('2. This should not appear.')
    with LoggingContext(logger, level=logging.DEBUG):
        logger.debug('3. This should appear once on stderr.')
    logger.debug('4. This should not appear.')
    h = logging.StreamHandler(sys.stdout)
    with LoggingContext(logger, level=logging.DEBUG, handler=h, close=True):
        logger.debug('5. This should appear twice - once on stderr and once on std
    logger.info('6. This should appear just once on stderr.')
    logger.debug('7. This should not appear.')
```

我们最初设置日志记录器的消息等级为 `INFO` ，因此消息#1出现，消息#2没有出现。在接下来的 `with` 代码块中我们暂时将消息等级变更为 `DEBUG` ，从而消息 #3 出现。在这一代码块退出后，日志记录器的消息等级恢复为 `INFO` ，从而消息 #4 没有出现。在下一个 `with` 代码块中，我们再一次将

设置消息等级设置为 `DEBUG` ，同时添加一个将消息写入 `sys.stdout` 的日志处理器。因此，消息#5在控制台出现两次 (分别通过 `stderr` 和 `stdout` )。在 `with` 语句完成后，状态与之前一样，因此消息 #6 出现（类似消息 #1），而消息 #7 没有出现（类似消息 #2）。

如果我们运行生成的脚本，结果如下：

```
$ python logctx.py
1. This should appear just once on stderr.
3. This should appear once on stderr.
5. This should appear twice - once on stderr and once on stdout.
5. This should appear twice - once on stderr and once on stdout.
6. This should appear just once on stderr.
```

我们将 `stderr` 标准错误重定向到 `/dev/null` ，我再次运行生成的脚步，唯一被写入 `stdout` 标准输出的消息，即我们所能看见的消息，如下：

```
$ python logctx.py 2>/dev/null
5. This should appear twice - once on stderr and once on stdout.
```

再一次，将 `stdout` 标准输出重定向到 `/dev/null`，我获得如下结果：

```
$ python logctx.py >/dev/null
1. This should appear just once on stderr.
3. This should appear once on stderr.
5. This should appear twice - once on stderr and once on stdout.
6. This should appear just once on stderr.
```

在这种情况下，与预期一致，打印到 `stdout` 标准输出的消息＃5不会出现。

当然，这里描述的方法可以被推广，例如临时附加日志记录过滤器。 请注意，上面的代码适用于Python 2以及Python 3。

## 命令行日志应用起步

下面的示例提供了如下功能：

- 根据命令行参数确定日志级别
- 在单独的文件中分发多条子命令，同一级别的子命令均以一致的方式记录。
- 最简单的配置用法

假定有一个命令行应用程序，用于停止、启动或重新启动某些服务。为了便于演示，不妨将 `app.py` 作为应用程序的主代码文件，并在 `start.py` 、 `stop.py` 和 `restart.py` 中实现单独的命令。再假定要通过命令行参数控制应用程序的日志粒度，默认为 `logging.INFO` 。以下是 `app.py` 的一个示例：

```
import argparse
import importlib
import logging
import os
import sys
```

```
def main(args=None):
    scriptname = os.path.basename(__file__)
    parser = argparse.ArgumentParser(scriptname)
    levels = ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL')
    parser.add_argument('--log-level', default='INFO', choices=levels)
    subparsers = parser.add_subparsers(dest='command',
                                       help='Available commands:')
    start_cmd = subparsers.add_parser('start', help='Start a service')
    start_cmd.add_argument('name', metavar='NAME',
                           help='Name of service to start')
    stop_cmd = subparsers.add_parser('stop',
                                     help='Stop one or more services')
    stop_cmd.add_argument('names', metavar='NAME', nargs='+',
                          help='Name of service to stop')
    restart_cmd = subparsers.add_parser('restart',
                                        help='Restart one or more services')
    restart_cmd.add_argument('names', metavar='NAME', nargs='+',
                             help='Name of service to restart')
    options = parser.parse_args()
    # 分发命令的代码可以全都放在此文件中。 只是出于演示目的，
    # 我们将在单独的模块中实现每个命令。
    try:
        mod = importlib.import_module(options.command)
        cmd = getattr(mod, 'command')
    except (ImportError, AttributeError):
        print('Unable to find the code for command \'%s\'' % options.command)
        return 1
    # 这里可以做得更为灵活并从文件或目录加载配置
    logging.basicConfig(level=options.log_level,
                        format='%(levelname)s %(name)s %(message)s')
    cmd(options)

if __name__ == '__main__':
    sys.exit(main())
```

`start`、`stop` 和 `restart` 命令可以在单独的模块中实现，启动命令的代码可如下：

```
# start.py
import logging

logger = logging.getLogger(__name__)

def command(options):
    logger.debug('About to start %s', options.name)
    # 在此进行实际的命令处理 ...
    logger.info('Started the \'%s\' service.', options.name)
```

然后是停止命令的代码：

```
# stop.py
import logging

logger = logging.getLogger(__name__)

def command(options):
    n = len(options.names)
    if n == 1:
        plural = ''
```

```
        services = '\'%s\'' % options.names[0]
    else:
        plural = 's'
        services = ', '.join('\'%s\'' % name for name in options.names)
        i = services.rfind(', ')
        services = services[:i] + ' and ' + services[i + 2:]
    logger.debug('About to stop %s', services)
    # 在此进行实际的命令处理 ...
    logger.info('Stopped the %s service%s.', services, plural)
```

重启命令类似：

```
# restart.py
import logging

logger = logging.getLogger(__name__)

def command(options):
    n = len(options.names)
    if n == 1:
        plural = ''
        services = '\'%s\'' % options.names[0]
    else:
        plural = 's'
        services = ', '.join('\'%s\'' % name for name in options.names)
        i = services.rfind(', ')
        services = services[:i] + ' and ' + services[i + 2:]
    logger.debug('About to restart %s', services)
    # 在此进行实际的命令处理 ...
    logger.info('Restarted the %s service%s.', services, plural)
```

如果以默认日志级别运行该程序，会得到以下结果：

```
$ python app.py start foo
INFO start Started the 'foo' service.

$ python app.py stop foo bar
INFO stop Stopped the 'foo' and 'bar' services.

$ python app.py restart foo bar baz
INFO restart Restarted the 'foo', 'bar' and 'baz' services.
```

第一个单词是日志级别，第二个单词是日志事件所在的模块或包的名称。

如果修改了日志级别，发送给日志的信息就能得以改变。如要显示更多信息，则可：

```
$ python app.py --log-level DEBUG start foo
DEBUG start About to start foo
INFO start Started the 'foo' service.

$ python app.py --log-level DEBUG stop foo bar
DEBUG stop About to stop 'foo' and 'bar'
INFO stop Stopped the 'foo' and 'bar' services.

$ python app.py --log-level DEBUG restart foo bar baz
```

```
DEBUG restart About to restart 'foo', 'bar' and 'baz'
INFO restart Restarted the 'foo', 'bar' and 'baz' services.
```

若要显示的信息少一些，则：

```
$ python app.py --log-level WARNING start foo
$ python app.py --log-level WARNING stop foo bar
$ python app.py --log-level WARNING restart foo bar baz
```

这里的命令不会向控制台输出任何信息，因为没有记录 `WARNING` 以上级别的日志。

## Qt GUI 日志示例

一个时常被提出的问题是 GUI 应用程序要如何记录日志。 Qt 框架是一个流行的跨平台 UI 框架，它具有使用 PySide2 或 PyQt5 库的 Python 绑定。

下面的例子演示了将日志写入 Qt GUI 程序的过程。这里引入了一个简单的 `QtHandler` 类，参数是一个可调用对象，其应为嵌入主线程某个“槽位”中运行的，因为GUI 的更新由主线程完成。这里还创建了一个工作线程，以便演示由 UI（通过人工点击日志按钮）和后台工作线程（此处只是记录级别和时间间隔均随机生成的日志信息）将日志写入 GUI 的过程。

该工作线程是用 Qt 的 `QThread` 类实现的，而不是 `threading` 模块，因为某些情况下只能采用 `QThread`，它与其他 `Qt` 组件的集成性更好一些。

此代码应当适用于最新的 `PySide6`, `PyQt6`, `PySide2` 或 `PyQt5` 发布版。 你也可以将此做法适配到更早的 Qt 版本。 请参阅代码片段中的注释来获取更详细的信息。

```
import datetime
import logging
import random
import sys
import time

# 处理不同 Qt 包之间的微小差异
try:
    from PySide6 import QtCore, QtGui, QtWidgets
    Signal = QtCore.Signal
    Slot = QtCore.Slot
except ImportError:
    try:
        from PyQt6 import QtCore, QtGui, QtWidgets
        Signal = QtCore.pyqtSignal
        Slot = QtCore.pyqtSlot
    except ImportError:
        try:
            from PySide2 import QtCore, QtGui, QtWidgets
            Signal = QtCore.Signal
            Slot = QtCore.Slot
        except ImportError:
            from PyQt5 import QtCore, QtGui, QtWidgets
            Signal = QtCore.pyqtSignal
            Slot = QtCore.pyqtSlot

logger = logging.getLogger(__name__)
```

```
#
# 信号需要被包含在 QObject 或其子类中以便能够正确地
# 初始化
#
class Signaller(QtCore.QObject):
    signal = Signal(str, logging.LogRecord)

#
# 输出到 Qt GUI 应当仅发生在主线程中。 因此，本处理器
# 被设计为接受一个已经设置好运行主线程的槽位函数。
# 在本示例中，该函数接受一个已格式化的日志消息字符串
# 参数，以及生成它的日志记录。 已格式化的字符串只是
# 出于方便考虑 —— 你也可以在槽位函数本身以任意方式
# 格式化字符串供输出。
#
# 你可以指定槽位函数执行任何你想要的 GUI 更新。 处理器
# 并不知道或关心特定的 UI 元素。
#
class QtHandler(logging.Handler):
    def __init__(self, slotfunc, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.signaller = Signaller()
        self.signaller.signal.connect(slotfunc)

    def emit(self, record):
        s = self.format(record)
        self.signaller.signal.emit(s, record)

#
# 本示例使用 QThreads，这意味着在 Python 层级中的线程
# 将为像 "Dummy-1" 的名称。 下面的函数将获得当前线程的
# Qt 名称。
#
def ctname():
    return QtCore.QThread.currentThread().objectName()


#
# 用于生成随机的日志记录层级。
#
LEVELS = (logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR,
          logging.CRITICAL)

#
# 这个工作类代表在一个独立于主线程的线程中完成的工作。
# 该线程开始执行工作的方式是通过按下一个连接到工作类中
# 槽位的按钮。
#
# 因为 LogRecord 中默认的 threadName 值没有什么用处。
# 我们增加了一个包含通过上述方式计算的 QThread 的
# qThreadName，并在一个“额外”字典中传递该值并使用它
# 将 LogRecord 更新为 QThread 名称。
#
# 这个示例工作类将顺序地输出消息，并以数秒的随机延时
# 进行间隔。
#
class Worker(QtCore.QObject):
    @Slot()
```

```
    def start(self):
        extra = {'qThreadName': ctname() }
        logger.debug('Started work', extra=extra)
        i = 1
        # 让线程运行直到被中断。 这允许合理并且清晰的
        # 线程终结。
        while not QtCore.QThread.currentThread().isInterruptionRequested():
            delay = 0.5 + random.random() * 2
            time.sleep(delay)
            try:
                if random.random() < 0.1:
                    raise ValueError('Exception raised: %d' % i)
                else:
                    level = random.choice(LEVELS)
                    logger.log(level, 'Message after delay of %3.1f: %d', delay, i
            except ValueError as e:
                logger.exception('Failed: %s', e, extra=extra)
            i += 1

#
# 为本专题指南示例实现一个简单的 UI。 其中包含:
#
# * 一个只读的文本编辑窗口用以显示已格式化的日志消息
# * 一个按钮用以开始工作并在单独线程中记录日志内容
# * 一个按钮用以以记录来自主线程的日志内容
# * 一个按钮用以清空日志窗口
#
class Window(QtWidgets.QWidget):

    COLORS = {
        logging.DEBUG: 'black',
        logging.INFO: 'blue',
        logging.WARNING: 'orange',
        logging.ERROR: 'red',
        logging.CRITICAL: 'purple',
    }

    def __init__(self, app):
        super().__init__()
        self.app = app
        self.textedit = te = QtWidgets.QPlainTextEdit(self)
        # 设置系统平台所使用的默认等宽字体
        f = QtGui.QFont('nosuchfont')
        if hasattr(f, 'Monospace'):
            f.setStyleHint(f.Monospace)
        else:
            f.setStyleHint(f.StyleHint.Monospace)  # 针对 Qt6
        te.setFont(f)
        te.setReadOnly(True)
        PB = QtWidgets.QPushButton
        self.work_button = PB('Start background work', self)
        self.log_button = PB('Log a message at a random level', self)
        self.clear_button = PB('Clear log window', self)
        self.handler = h = QtHandler(self.update_status)
        # 记得在格式字符串中使用 qThreadName 而非 threadName。
        fs = '%(asctime)s %(qThreadName)-12s %(levelname)-8s %(message)s'
        formatter = logging.Formatter(fs)
        h.setFormatter(formatter)
        logger.addHandler(h)
        # 设置当退出时终结 QThread
```

```
        app.aboutToQuit.connect(self.force_quit)

        # 对所有控件进行布局
        layout = QtWidgets.QVBoxLayout(self)
        layout.addWidget(te)
        layout.addWidget(self.work_button)
        layout.addWidget(self.log_button)
        layout.addWidget(self.clear_button)
        self.setFixedSize(900, 400)

        # 连接非工作槽位和信号
        self.log_button.clicked.connect(self.manual_update)
        self.clear_button.clicked.connect(self.clear_display)

        # 启动一个新工作线程并为其连接槽位
        self.start_thread()
        self.work_button.clicked.connect(self.worker.start)
        # 一旦启动，该按钮应当被禁用
        self.work_button.clicked.connect(lambda : self.work_button.setEnabled(Fals

    def start_thread(self):
        self.worker = Worker()
        self.worker_thread = QtCore.QThread()
        self.worker.setObjectName('Worker')
        self.worker_thread.setObjectName('WorkerThread')  # 针对 qThreadName
        self.worker.moveToThread(self.worker_thread)
        # 这将在工作线程中启动一个事件循环
        self.worker_thread.start()

    def kill_thread(self):
        # 告知工作线程停止运行，然后告知它退出并等待
        # 后续发生的事件
        self.worker_thread.requestInterruption()
        if self.worker_thread.isRunning():
            self.worker_thread.quit()
            self.worker_thread.wait()
        else:
            print('worker has already exited.')

    def force_quit(self):
        # 当窗口被关闭时使用
        if self.worker_thread.isRunning():
            self.kill_thread()

    # 下面的函数更新 UI 并在主线程中运行因为槽位是在
    # 那里设置的

    @Slot(str, logging.LogRecord)
    def update_status(self, status, record):
        color = self.COLORS.get(record.levelno, 'black')
        s = '<pre><font color="%s">%s</font></pre>' % (color, status)
        self.textedit.appendHtml(s)

    @Slot()
    def manual_update(self):
        # 此函数使用传入的已格式化消息，但也会使用来自
        # 记录的信息根据其严重程度（层级）以适当的颜色
        # 格式化消息。
        level = random.choice(LEVELS)
        extra = {'qThreadName': ctname() }
```

```
        logger.log(level, 'Manually logged!', extra=extra)

    @Slot()
    def clear_display(self):
        self.textedit.clear()


def main():
    QtCore.QThread.currentThread().setObjectName('MainThread')
    logging.getLogger().setLevel(logging.DEBUG)
    app = QtWidgets.QApplication(sys.argv)
    example = Window(app)
    example.show()
    if hasattr(app, 'exec'):
        rc = app.exec()
    else:
        rc = app.exec_()
    sys.exit(rc)

if __name__=='__main__':
    main()
```

## 将日志记录到带有 RFC5424 支持的 syslog

虽然 **RFC 5424** 在 2009 年就已发布，但大多数 syslog 服务器都默认被配置为使用更旧的 **RFC 3164**，它发布于 2001 年。 当 `logging` 在 2003 年被加入 Python 时，它支持了当时（唯一存在的）较早版本的协议。 自从 RFC5424 发布后，因为它还未被广泛部署到 syslog 服务器上，因此 `SysLogHandler` 的功能也没有被更新。

RFC 5424 包括一些有用的特性例如对结构化数据的支持等，如果你想要能够将日志记录到带有该协议支持的 syslog 服务器上，你可以使用一个看起来像是这样的子类化处理器来实现:

```
import datetime
import logging.handlers
import re
import socket
import time

class SysLogHandler5424(logging.handlers.SysLogHandler):

    tz_offset = re.compile(r'([+-]\d{2})(\d{2})$')
    escaped = re.compile(r'([\]"\\])')

    def __init__(self, *args, **kwargs):
        self.msgid = kwargs.pop('msgid', None)
        self.appname = kwargs.pop('appname', None)
        super().__init__(*args, **kwargs)

    def format(self, record):
        version = 1
        asctime = datetime.datetime.fromtimestamp(record.created).isoformat()
        m = self.tz_offset.match(time.strftime('%z'))
        has_offset = False
        if m and time.timezone:
            hrs, mins = m.groups()
            if int(hrs) or int(mins):
                has_offset = True
```

```
        if not has_offset:
            asctime += 'Z'
        else:
            asctime += f'{hrs}:{mins}'
        try:
            hostname = socket.gethostname()
        except Exception:
            hostname = '-'
        appname = self.appname or '-'
        procid = record.process
        msgid = '-'
        msg = super().format(record)
        sdata = '-'
        if hasattr(record, 'structured_data'):
            sd = record.structured_data
            # 这应当是一个字典，其中的键为 SD-ID 而值则为
            # 将 PARAM-NAME 映射到 PARAM-VALUE 的字典
            # （请参阅 RFC 了解其含义）
            # 这里没有错误检查 — 它只是作为演示，你可以
            # 调整此代码以在生产环境中使用
            parts = []

            def replacer(m):
                g = m.groups()
                return '\\' + g[0]

            for sdid, dv in sd.items():
                part = f'[{sdid}'
                for k, v in dv.items():
                    s = str(v)
                    s = self.escaped.sub(replacer, s)
                    part += f' {k}="{s}"'
                part += ']'
                parts.append(part)
            sdata = ''.join(parts)
        return f'{version} {asctime} {hostname} {appname} {procid} {msgid} {sdata}
```

你需要熟悉 RFC 5424 才能完全理解上面的代码，你还可能会有稍加变化的的需求（例如你要如何将结构化数据记入日志）。 不管怎样，上面的代码应当根据你的特定需求来灵活调整。 通过上面的处理器，你可以使用类似这样的代码来传入结构化数据:

```
sd = {
    'foo@12345': {'bar': 'baz', 'baz': 'bozz', 'fizz': r'buzz'},
    'foo@54321': {'rab': 'baz', 'zab': 'bozz', 'zzif': r'buzz'}
}
extra = {'structured_data': sd}
i = 1
logger.debug('Message %d', i, extra=extra)
```

## 如何将日志记录器作为输出流

有时，你需要通过接口访问某个预期要写入到文件型对象第三方 API，但你希望将 API 的输出重定向到一个日志记录器。 你可以使用一个以文件类 API 来包装日志记录器的类。 下面是一个演示这样的类的简短脚本:

```
import logging

class LoggerWriter:
    def __init__(self, logger, level):
        self.logger = logger
        self.level = level

    def write(self, message):
        if message != '\n':  # 避免打印空白行，如果你希望如此
            self.logger.log(self.level, message)

    def flush(self):
        # 实际上不做任何事，但对文件型对象来说应当提供
        # —— 因此根据你的情况作为可选项
        pass

    def close(self):
        # 实际上不做任何事，但对文件型对象来说应当提供
        # —— 因此根据你的情况作为可选项。 你可能会希望
        # 设置一个旗标以便后续的写入调用引发异常
        pass

def main():
    logging.basicConfig(level=logging.DEBUG)
    logger = logging.getLogger('demo')
    info_fp = LoggerWriter(logger, logging.INFO)
    debug_fp = LoggerWriter(logger, logging.DEBUG)
    print('An INFO message', file=info_fp)
    print('A DEBUG message', file=debug_fp)

if __name__ == "__main__":
    main()
```

当此脚本运行时，它将打印

```
INFO:demo:An INFO message
DEBUG:demo:A DEBUG message
```

你还可以使用 `LoggerWriter` 通过下面这样的做法来重定向 `sys.stdout` 和 `sys.stderr`:

```
import sys

sys.stdout = LoggerWriter(logger, logging.INFO)
sys.stderr = LoggerWriter(logger, logging.WARNING)
```

你应当在根据需要配置日志记录 *之后* 再这样做。 在上面的例子中，`basicConfig()` 调用执行了此操作（在 `sys.stderr` 被一个 `LoggerWriter` 实例覆盖 *之前* 使用它的值）。 然后，你将得到这样的结果:

```
>>> print('Foo')
INFO:demo:Foo
>>> print('Bar', file=sys.stderr)
WARNING:demo:Bar
>>>
```

当然，上面的例子是按照 basicConfig() 所使用的格式来显示输出的，但你也可以在配置日志记录时使用其他的格式。

请注意当使用上面的预置方案时，你将在一定程度上受到你所拦截的写入调用的缓冲和顺序的控制。 例如，在使用上述 `LoggerWriter` 的定义的情况下，如果你使用代码段

```
sys.stderr = LoggerWriter(logger, logging.WARNING)
1 / 0
```

则运行该脚本的结果为

```
WARNING:demo:Traceback (most recent call last):

WARNING:demo:  File "/home/runner/cookbook-loggerwriter/test.py", line 53, in <mod

WARNING:demo:
WARNING:demo:main()
WARNING:demo:  File "/home/runner/cookbook-loggerwriter/test.py", line 49, in main

WARNING:demo:
WARNING:demo:1 / 0
WARNING:demo:ZeroDivisionError
WARNING:demo::
WARNING:demo:division by zero
```

如你所见，这个输出并不很理想。 那是因为下层的写入 `sys.stderr` 的代码会执行多次写入，每次都将产生一条单独的日志记录行（例如，上面的最后三行）。 要避免这个问题，你需要使用缓冲并且只在看到换行符时才输出日志记录行。 让我们使用一个更好些的 `LoggerWriter` 实现：

```
class BufferingLoggerWriter(LoggerWriter):
    def __init__(self, logger, level):
        super().__init__(logger, level)
        self.buffer = ''

    def write(self, message):
        if '\n' not in message:
            self.buffer += message
        else:
            parts = message.split('\n')
            if self.buffer:
                s = self.buffer + parts.pop(0)
                self.logger.log(self.level, s)
            self.buffer = parts.pop()
            for part in parts:
                self.logger.log(self.level, part)
```

这段代码对内容进行了缓冲直至遇到换行符，然后将完整的行写入日志记录。 通过这种方式，你将得到更好的输出:

```
WARNING:demo:Traceback (most recent call last):
WARNING:demo:  File "/home/runner/cookbook-loggerwriter/main.py", line 55, in <mod
WARNING:demo:    main()
WARNING:demo:  File "/home/runner/cookbook-loggerwriter/main.py", line 52, in main
```

```
WARNING:demo:    1/0
WARNING:demo:ZeroDivisionError: division by zero
```

## 如何统一地处理日志记录输出中的换行符

通常，被记录的消息（输出到控制台或文件）是由单行文本组成的。 不过，在某些时候也会需要处理具有多行的消息 —— 不论是因为日志格式字符串包含换行符，还是因为被记录的数据包含换行符。 如果你想以统一的方式处理此类消息，使得被记录的消息中的每一行格式看起来保持一致就像它是被单独记录的，你可以使用处理器混入类做到这一点，如下面的代码片段所示：

```
# 假定这是在一个模块的 mymixins.py 中
import copy

class MultilineMixin:
    def emit(self, record):
        s = record.getMessage()
        if '\n' not in s:
            super().emit(record)
        else:
            lines = s.splitlines()
            rec = copy.copy(record)
            rec.args = None
            for line in lines:
                rec.msg = line
                super().emit(rec)
```

你可以像下面的脚本一样使用该混入类：

```
import logging

from mymixins import MultilineMixin

logger = logging.getLogger(__name__)

class StreamHandler(MultilineMixin, logging.StreamHandler):
    pass

if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG, format='%(asctime)s %(levelname)-9s %
                        handlers = [StreamHandler()])
    logger.debug('Single line')
    logger.debug('Multiple lines:\nfool me once ...')
    logger.debug('Another single line')
    logger.debug('Multiple lines:\n%s', 'fool me ...\ncan\'t get fooled again')
```

在运行时，这个脚本将打印如下内容：

```
2025-07-02 13:54:47,234 DEBUG     Single line
2025-07-02 13:54:47,234 DEBUG     Multiple lines:
2025-07-02 13:54:47,234 DEBUG     fool me once ...
2025-07-02 13:54:47,234 DEBUG     Another single line
2025-07-02 13:54:47,234 DEBUG     Multiple lines:
2025-07-02 13:54:47,234 DEBUG     fool me ...
2025-07-02 13:54:47,234 DEBUG     can't get fooled again
```

另一方面，如果你担心 日志注入，你可以使用对换行符进行转义的格式化器，就像下面的例子：

```
import logging

logger = logging.getLogger(__name__)

class EscapingFormatter(logging.Formatter):
    def format(self, record):
        s = super().format(record)
        return s.replace('\n', r'\n')

if __name__ == '__main__':
    h = logging.StreamHandler()
    h.setFormatter(EscapingFormatter('%(asctime)s %(levelname)-9s %(message)s'))
    logging.basicConfig(level=logging.DEBUG, handlers = [h])
    logger.debug('Single line')
    logger.debug('Multiple lines:\nfool me once ...')
    logger.debug('Another single line')
    logger.debug('Multiple lines:\n%s', 'fool me ...\ncan\'t get fooled again')
```

当然，你可以使用任何对你来说最合适的转义方案。 在运行时，这个脚本将会产生这样的输出：

```
2025-07-09 06:47:33,783 DEBUG     Single line
2025-07-09 06:47:33,783 DEBUG     Multiple lines:\nfool me once ...
2025-07-09 06:47:33,783 DEBUG     Another single line
2025-07-09 06:47:33,783 DEBUG     Multiple lines:\nfool me ...\ncan't get fooled a
```

转义行为不能是标准库默认设置，因为它会破坏向下兼容性。

## 理应避免的用法

前几节虽介绍了几种方案，描述了可能需要处理的操作，但值得一提的是，有些用法是 *没有好处* 的，大多数情况下应该避免使用。下面几节的顺序不分先后。

### 多次打开同一个日志文件

因会导致 "文件被其他进程占用 "错误，所以在 Windows 中一般无法多次打开同一个文件。但在 POSIX 平台中，多次打开同一个文件不会报任何错误。这种操作可能是意外发生的，比如：

- 多次添加指向同一文件的 handler（比如通过复制/粘贴，或忘记修改）。
- 打开两个貌似不同（文件名不一样）的文件，但一个是另一个的符号链接，所以其实是同一个文件。
- 进程 fork，然后父进程和子进程都有对同一文件的引用。 例如，这可能是通过使用 `multiprocessing` 模块实现的。

在大多数情况下，多次打开同一个文件 *貌似* 一切正常，但实际会导致很多问题。

- 由于多个线程或进程会尝试写入同一个文件，日志输出可能会出现乱码。尽管日志对象可以防止多个线程同时使用同一个 handler 实例，但如果两个不同的线程使用不同的 handler 实例同时写入文件，而这两个 handler 又恰好指向同一个文件，那么就失去了这种防护。

- 尝试删除文件（例如在轮换日志文件时）会静默地失败，因为存在另一个指向它的引用。 这可能导致混乱并浪费调试时间 —— 日志条目会出现在意想不到的地方，或者完全丢失。 或者会有应当移除的文件仍然保持存在，文件还会在已经设置了基于文件大小的轮换的情况下仍然增长到预料之外的大小。

请用 从多个进程记录至单个文件 中介绍的技术来避免上述问题。

### 将日志对象用作属性或传递参数

虽然特殊情况下可能有必要如此，但一般来说没有意义，因为日志是单实例对象。代码总是可以通过 `logging.getLogger(name)` 用名称访问一个已有的日志对象实例，因此将实例作为参数来传递，或作为属性留存，都是毫无意义的。请注意，在其他语言中，如 Java 和 C#，日志对象通常是静态类属性。但在 Python 中是没有意义的，因为软件拆分的单位是模块（而不是类）。

### 为库中的日志记录器添加 `NullHandler` 以外的处理器

通过添加 handler、formatter 和 filter 来配置日志，这是应用程序开发人员的责任，而不是库开发人员该做的。如果正在维护一个库，请确保不要向任何日志对象添加 `NullHandler` 实例以外的 handler。

### 创建大量的日志对象

日志是单实例对象，在代码执行过程中不会被释放，因此创建大量的日志对象会占用很多内存，而这些内存又不能被释放。与其为每个文件或网络连接创建一个日志，还不如利用 已有机制 将上下文信息传给自定义日志对象，并将创建的日志对象限制在应用程序内的指定区域（通常是模块，但偶尔会再精细些）使用。

## 其他资源

**参见:**

**模块** `logging`
日志记录模块的 API 参考。

`logging.config` **模块**
日志记录模块的配置 API 。

`logging.handlers` **模块**
日志记录模块附带的有用处理器。

基础教程

进阶教程