日志操作手册

發𝔽 *3.10.11*

Guido van Rossum
and the Python development team

5 月 24, 2023

Python Software Foundation
Email: docs@python.org

Contents

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

This page contains a number of recipes related to logging, which have been found useful in the past. For links to tutorial and reference information, please see 其他资源.

1 在多个模块中记录日志

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

# create logger with 'spam_application'
logger = logging.getLogger('spam_application')
logger.setLevel(logging.DEBUG)
# create file handler which logs even debug messages
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s
↪')
fh.setFormatter(formatter)
ch.setFormatter(formatter)
# add the handlers to the logger
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

# create logger
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 -
```

（下页继续）

(繼續上一頁)

```
   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()
```

2 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
↪%(threadName)s %(message)s')
    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
```

(下页继续)

```
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。

3 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：

```python
import logging

logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)
# create file handler which logs even debug messages
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s
↪')
ch.setFormatter(formatter)
fh.setFormatter(formatter)
# add the handlers to logger
logger.addHandler(ch)
logger.addHandler(fh)

# 'application' code
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。

4 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

# set up logging to file - see previous section for more details
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')
# define a Handler which writes INFO messages or higher to the sys.stderr
console = logging.StreamHandler()
console.setLevel(logging.INFO)
# set a format which is simpler for console use
formatter = logging.Formatter('%(name)-12s: %(levelname)-8s %(message)s')
# tell the handler to use this format
console.setFormatter(formatter)
# add the handler to the root logger
logging.getLogger('').addHandler(console)

# Now, we can log to the root logger, or any other logger. First the root...
logging.info('Jackdaws love my big sphinx of quartz.')

# Now, define a couple of other loggers which might represent areas in your
# application:

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。

Note that the above choice of log filename `/tmp/myapp.log` implies use of a standard location for temporary files on POSIX systems. On Windows, you may need to choose a different directory name for the log - just ensure that the directory exists and that you have the permissions to create and update files in it.

5 Custom handling of levels

Sometimes, you might want to do something slightly different from the standard handling of levels in handlers, where all levels above a threshold get processed by a handler. To do this, you need to use filters. Let's look at a scenario where you want to arrange things as follows:

- Send messages of severity `INFO` and `WARNING` to `sys.stdout`
- Send messages of severity `ERROR` and above to `sys.stderr`
- Send messages of severity `DEBUG` and above to file `app.log`

Suppose you configure logging with the following JSON:

```
{
    "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"
        ]
    }
}
```

This configuration does *almost* what we want, except that `sys.stdout` would show messages of severity `ERROR` and above as well as `INFO` and `WARNING` messages. To prevent this, we can set up a filter which excludes those messages and add it to the relevant handler. This can be configured by adding a `filters` section parallel to `formatters` and `handlers`:

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

(下页继续)

(繼續上一頁)

```
    }
}
```

and changing the section on the `stdout` handler to add it:

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

A filter is just a function, so we can define the `filter_maker` (a factory function) as follows:

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

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

    return filter
```

This converts the string argument passed in to a numeric level, and returns a function which only returns `True` if the level of the passed in record is at or below the specified level. Note that in this example I have defined the `filter_maker` in a test script `main.py` that I run from the command line, so its module will be `__main__` - hence the `__main__.filter_maker` in the filter configuration. You will need to change that if you define it in a different module.

With the filter added, we can run `main.py`, which in full is:

```
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": {
```

(下页继续)

(繼續上一頁)

```
            "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')
```

And after running it like this:

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

We can see the results are as expected:

```
$ 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
::::::::::::::
```

(下页继续)

```
INFO      - An INFO message
WARNING   - A WARNING message
```

6 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

# read initial config file
logging.config.fileConfig('logging.conf')

# create and start listener on port 9999
t = logging.config.listen(9999)
t.start()

logger = logging.getLogger('simpleExample')

try:
    # loop through logging calls to see the difference
    # new configurations make, until Ctrl+C is pressed
    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:
    # cleanup
    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')
```

7 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)  # no limit on size
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()
# The log output will display the thread which generated
# the event (the main thread) rather than the internal
# thread which monitors the internal queue. This is what
# you want to happen.
root.warning('Look out!')
listener.stop()
```

在运行后会产生：

```
MainThread: Look out!
```

備註: Although the earlier discussion wasn't specifically talking about async code, but rather about slow logging handlers, it should be noted that when logging from async code, network and even file handlers could lead to problems (blocking the event loop) because some logging is done from asyncio internals. It might be best, if any async code is used in an application, to use the above approach for logging, so that any blocking code runs only in the QueueListener thread.

3.5 版更變: 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 。

8 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)
# don't bother with a formatter, since a socket handler sends the event as
# an unformatted pickle
rootLogger.addHandler(socketHandler)

# Now, we can log to the root logger, or any other logger. First the root...
logging.info('Jackdaws love my big sphinx of quartz.')

# Now, define a couple of other loggers which might represent areas in your
# application:

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)
```

(下页继续)

(繼續上一頁)

```
            self.handleLogRecord(record)

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

    def handleLogRecord(self, record):
        # if a name is specified, we use the named logger rather than the one
        # implied by the record.
        if self.server.logname is not None:
            name = self.server.logname
        else:
            name = record.name
        logger = logging.getLogger(name)
        # N.B. EVERY record gets logged. This is because Logger.handle
        # is normally called AFTER logger-level filtering. If you want
        # to do filtering, do it at the client end to save wasting
        # cycles and network bandwidth!
        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.
```

(下页继续)

```
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，某些时候 pickle 会存在一些安全问题。若有问题可换用自己的序列化方案，只要覆盖 `makePickle()` 方法即可，并调整上述脚本以采用自己的序列化方案。

8.1 在生产中运行日志套接字侦听器

To run a logging listener in production, you may need to use a process-management tool such as Supervisor. Here is a Gist which provides the bare-bones files to run the above functionality using Supervisor. It consists of the following files:

File	目的
`prepare.sh`	A Bash script to prepare the environment for testing
`supervisor.conf`	The Supervisor configuration file, which has entries for the listener and a multi-process web application
`ensure_app.sh`	A Bash script to ensure that Supervisor is running with the above configuration
`log_listener.py`	The socket listener program which receives log events and records them to a file
`main.py`	A simple web application which performs logging via a socket connected to the listener
`webapp.json`	A JSON configuration file for the web application
`client.py`	A Python script to exercise the web application

The web application uses Gunicorn, which is a popular web application server that starts multiple worker processes to handle requests. This example setup shows how the workers can write to the same log file without conflicting with one another --- they all go through the socket listener.

To test these files, do the following in a POSIX environment:

1. Download the Gist as a ZIP archive using the *Download ZIP* button.
2. Unzip the above files from the archive into a scratch directory.
3. In the scratch directory, run `bash prepare.sh` to get things ready. This creates a `run` subdirectory to contain Supervisor-related and log files, and a `venv` subdirectory to contain a virtual environment into which `bottle`, `gunicorn` and `supervisor` are installed.
4. Run `bash ensure_app.sh` to ensure that Supervisor is running with the above configuration.
5. Run `venv/bin/python client.py` to exercise the web application, which will lead to records being written to the log.
6. Inspect the log files in the `run` subdirectory. You should see the most recent log lines in files matching the pattern `app.log*`. They won't be in any particular order, since they have been handled concurrently by different worker processes in a non-deterministic way.
7. You can shut down the listener and the web application by running `venv/bin/supervisorctl -c supervisor.conf shutdown`.

You may need to tweak the configuration files in the unlikely event that the configured ports clash with something else in your test environment.

9 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。

9.1 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):
    """
    Delegate a debug call to the underlying logger, after adding
    contextual information from this adapter instance.
    """
    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。

9.2 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.
↪CRITICAL)
    logging.basicConfig(level=logging.DEBUG,
                        format='%(asctime)-15s %(name)-5s %(levelname)-8s IP:
↪%(ip)-15s User: %(user)-8s %(message)s')
    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␣
↪message
2010-09-06 22:38:15,300 a.b.c INFO     IP: 192.168.0.1     User: sheila   An info␣
↪message with some parameters
```

(下页继续)

(繼續上一頁)

```
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 127.0.0.1       User: sheila   A␣
↪message at CRITICAL level with 2 parameters
2010-09-06 22:38:15,300 d.e.f ERROR    IP: 127.0.0.1       User: jim      A␣
↪message at ERROR level with 2 parameters
2010-09-06 22:38:15,300 d.e.f DEBUG    IP: 127.0.0.1       User: sheila   A␣
↪message at DEBUG level with 2 parameters
2010-09-06 22:38:15,300 d.e.f ERROR    IP: 123.231.231.123 User: fred     A␣
↪message at ERROR level with 2 parameters
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 192.168.0.1     User: jim      A␣
↪message at CRITICAL level with 2 parameters
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 127.0.0.1       User: sheila   A␣
↪message at CRITICAL level with 2 parameters
2010-09-06 22:38:15,300 d.e.f DEBUG    IP: 192.168.0.1     User: jim      A␣
↪message at DEBUG level with 2 parameters
2010-09-06 22:38:15,301 d.e.f ERROR    IP: 127.0.0.1       User: sheila   A␣
↪message at ERROR level with 2 parameters
2010-09-06 22:38:15,301 d.e.f DEBUG    IP: 123.231.231.123 User: fred     A␣
↪message at DEBUG level with 2 parameters
2010-09-06 22:38:15,301 d.e.f INFO     IP: 123.231.231.123 User: fred     A␣
↪message at INFO level with 2 parameters
```

10 Use of contextvars

Since Python 3.7, the `contextvars` module has provided context-local storage which works for both `threading` and `asyncio` processing needs. This type of storage may thus be generally preferable to thread-locals. The following example shows how, in a multi-threaded environment, logs can populated with contextual information such as, for example, request attributes handled by web applications.

For the purposes of illustration, say that you have different web applications, each independent of the other but running in the same Python process and using a library common to them. How can each of these applications have their own log, where all logging messages from the library (and other request processing code) are directed to the appropriate application's log file, while including in the log additional contextual information such as client IP, HTTP request method and client username?

Let's assume that the library can be simulated by the following code:

```python
# webapplib.py
import logging
import time

logger = logging.getLogger(__name__)

def useful():
    # Just a representative event logged from the library
    logger.debug('Hello from webapplib!')
    # Just sleep for a bit so other threads get to run
    time.sleep(0.01)
```

We can simulate the multiple web applications by means of two simple classes, `Request` and `WebApp`. These simulate how real threaded web applications work - each request is handled by a thread:

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

(下页继续)

(繼續上一頁)

```
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

# A dummy set of requests which will be used in the simulation - we'll just pick
# from this list randomly. Note that all GET requests are from 192.168.2.XXX
# addresses, whereas POST requests are from 192.16.3.XXX addresses. Three users
# are represented in the sample requests.

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'),
]

# Note that the format string includes references to request context information
# such as HTTP method, client IP and username

formatter = logging.Formatter('%(threadName)-11s %(appName)s %(name)-9s %(user)-6s
↪%(ip)s %(method)-4s %(message)s')

# Create our context variables. These will be filled at the start of request
# processing, and used in the logging that happens during that processing

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
```

(下页继续)

(繼續上一頁)

```
    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()

    # Create the dummy webapps and put them in a list which we can use to select
    # from randomly
    app1 = WebApp('app1')
    app2 = WebApp('app2')
    apps = [app1, app2]
    threads = []
    # Add a common handler which will capture all events
    handler = logging.FileHandler('app.log', 'w')
    handler.setFormatter(formatter)
    root.addHandler(handler)

    # Generate calls to process requests
    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
```

(下页继续)

(繼續上一頁)

```
    # Wait for the threads to terminate
    for t in threads:
        t.join()

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

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

If you run the above, you should find that roughly half the requests go into `app1.log` and the rest into `app2.log`, and the all the requests are logged to `app.log`. Each webapp-specific log will contain only log entries for only that webapp, and the request information will be displayed consistently in the log (i.e. the information in each dummy request will always appear together in a log line). This is illustrated by the following shell output:

```
~/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␣
↪processing started
Thread-3 (process_request) app1 webapplib jim    192.168.3.21 POST Hello from␣
↪webapplib!
Thread-5 (process_request) app1 __main__  jim    192.168.3.21 POST Request␣
↪processing started
~/logging-contextual-webapp$ head -3 app2.log
Thread-1 (process_request) app2 __main__  sheila 192.168.2.21 GET  Request␣
↪processing started
Thread-1 (process_request) app2 webapplib sheila 192.168.2.21 GET  Hello from␣
↪webapplib!
Thread-2 (process_request) app2 __main__  jim    192.168.2.20 GET  Request␣
↪processing started
~/logging-contextual-webapp$ head app.log
Thread-1 (process_request) app2 __main__  sheila 192.168.2.21 GET  Request␣
↪processing started
Thread-1 (process_request) app2 webapplib sheila 192.168.2.21 GET  Hello from␣
↪webapplib!
Thread-2 (process_request) app2 __main__  jim    192.168.2.20 GET  Request␣
↪processing started
Thread-3 (process_request) app1 __main__  jim    192.168.3.21 POST Request␣
↪processing started
Thread-2 (process_request) app2 webapplib jim    192.168.2.20 GET  Hello from␣
↪webapplib!
Thread-3 (process_request) app1 webapplib jim    192.168.3.21 POST Hello from␣
↪webapplib!
Thread-4 (process_request) app2 __main__  fred   192.168.2.22 GET  Request␣
↪processing started
Thread-5 (process_request) app1 __main__  jim    192.168.3.21 POST Request␣
↪processing started
Thread-4 (process_request) app2 webapplib fred   192.168.2.22 GET  Hello from␣
↪webapplib!
Thread-6 (process_request) app1 __main__  jim    192.168.3.21 POST Request␣
↪processing started
~/logging-contextual-webapp$ grep app1 app1.log | wc -l
153
```

(下页继续)

(繼續上一頁)

```
~/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
```

11 Imparting contextual information in handlers

Each Handler has its own chain of filters. If you want to add contextual information to a LogRecord without leaking it to other handlers, you can use a filter that returns a new LogRecord instead of modifying it in-place, as shown in the following script:

```
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')
```

12 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程序，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。现有的 FileHandler 及其子类目前并不使用 multiprocessing，尽管它们将来可能会这样做。请注意在目前，multiprocessing 模块并未在所有平台上都提供可用的锁功能 (参见 https://bugs.python.org/issue3770)。

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:

```
# You'll need these imports in your own code
import logging
```

(下页继续)

(繼續上一頁)

```
import logging.handlers
import multiprocessing

# Next two import lines for this demo only
from random import choice, random
import time

#
# Because you'll want to define the logging configurations for listener and␣
↪workers, the
# listener and worker process functions take a configurer parameter which is a␣
↪callable
# for configuring logging for that process. These functions are also passed the␣
↪queue,
# which they use for communication.
#
# In practice, you can configure the listener however you want, but note that in␣
↪this
# simple example, the listener does not apply level or filter logic to received␣
↪records.
# In practice, you would probably want to do this logic in the worker processes,␣
↪to avoid
# sending events which would be filtered out between processes.
#
# The size of the rotated files is made small so you can see the results easily.
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
↪%(message)s')
    h.setFormatter(f)
    root.addHandler(h)

# This is the listener process top-level loop: wait for logging events
# (LogRecords)on the queue and handle them, quit when you get a None for a
# LogRecord.
def listener_process(queue, configurer):
    configurer()
    while True:
        try:
            record = queue.get()
            if record is None:  # We send this as a sentinel to tell the listener␣
↪to quit.
                break
            logger = logging.getLogger(record.name)
            logger.handle(record)  # No level or filter logic applied - just do it!
        except Exception:
            import sys, traceback
            print('Whoops! Problem:', file=sys.stderr)
            traceback.print_exc(file=sys.stderr)

# Arrays used for random selections in this demo

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',
```

(下页继续)

(繼續上一頁)

```
    'Random message #3',
]

# The worker configuration is done at the start of the worker process run.
# Note that on Windows you can't rely on fork semantics, so each process
# will run the logging configuration code when it starts.
def worker_configurer(queue):
    h = logging.handlers.QueueHandler(queue)  # Just the one handler needed
    root = logging.getLogger()
    root.addHandler(h)
    # send all messages, for demo; no other level or filter logic applied.
    root.setLevel(logging.DEBUG)

# This is the worker process top-level loop, which just logs ten events with
# random intervening delays before terminating.
# The print messages are just so you know it's doing something!
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)

# Here's where the demo gets orchestrated. Create the queue, create and start
# the listener, create ten workers and start them, wait for them to finish,
# then send a None to the queue to tell the listener to finish.
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):
```

(下页继续)

(繼續上一頁)

```
    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)-
↪10s %(message)s'
            }
        },
        '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']
```

(下页继续)

```
                }
            },
            '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()
        # At this point, the main process could do some useful work of its own
        # Once it's done that, it can wait for the workers to terminate...
        for wp in workers:
            wp.join()
        # And now tell the logging thread to finish up, too
        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。

12.1 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)  # also works with the examples above
```

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)
```

12.2 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句柄。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。

13 轮换日志文件

有时，你希望当日志文件不断记录增长至一定大小时，打开一个新的文件接着记录。你可能希望只保留一定数量的日志文件，当不断的创建文件到达该数量时，又覆盖掉最开始的文件形成循环。对于这种使用场景，日志包提供了 RotatingFileHandler:

```
import glob
import logging
import logging.handlers

LOG_FILENAME = 'logging_rotatingfile_example.out'

# Set up a specific logger with our desired output level
my_logger = logging.getLogger('MyLogger')
my_logger.setLevel(logging.DEBUG)

# Add the log message handler to the logger
handler = logging.handlers.RotatingFileHandler(
              LOG_FILENAME, maxBytes=20, backupCount=5)

my_logger.addHandler(handler)

# Log some messages
for i in range(20):
    my_logger.debug('i = %d' % i)

# See what files are created
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:file:*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。

14 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
```

（下页继续）

(繼續上一頁)

```
        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。

值得注意的是，上述做法对性能并没什么影响：格式化过程其实不是在日志记录调用时发生的，而是在日志信息即将由 handler 输出到日志时发生。因此，唯一可能让人困惑的稍不寻常的地方，就是包裹在格式串和参数外面的括号，而不是格式串。因为 __ 符号只是对 XXXMessage 类的构造函数调用的语法糖。

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 __init__(self, logger, extra=None):
        super().__init__(logger, extra or {})

    def log(self, level, msg, /, *args, **kwargs):
        if self.isEnabledFor(level):
            msg, kwargs = self.process(msg, kwargs)
            self.logger._log(level, Message(msg, args), (), **kwargs)

logger = StyleAdapter(logging.getLogger(__name__))

def main():
```

(下页继续)

```
    logger.debug('Hello, {}', 'world!')

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

在用 Python 3.2 以上版本运行时，上述代码应该会把 `Hello, world!` 写入日志。

15 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。

16 子类化 QueueHandler - ZeroMQ 示例

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句柄 (作为它的'queue'):

```
import zmq   # using pyzmq, the Python binding for ZeroMQ
import json  # for serializing records portably

ctx = zmq.Context()
sock = zmq.Socket(ctx, zmq.PUB)  # or zmq.PUSH, or other suitable value
sock.bind('tcp://*:5556')        # or wherever

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()
```

17 子类化 QueueListener ——ZeroMQ 示例

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, '')  # subscribe to everything
        socket.connect(uri)
        super().__init__(socket, *handlers, **kwargs)

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

也参考:

logging 模組 日志记录模块的 API 参考。

logging.config 模組 日志记录模块的配置 API 。

logging.handlers 模組 日志记录模块附带的有用处理器。

日志操作基础教程

18 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': True,
    'formatters': {
        'verbose': {
            'format': '%(levelname)s %(asctime)s %(module)s %(process)d %(thread)d
↪%(message)s'
        },
        'simple': {
            'format': '%(levelname)s %(message)s'
        },
    },
    'filters': {
        'special': {
            '()': 'project.logging.SpecialFilter',
            'foo': 'bar',
        }
    },
    'handlers': {
        'null': {
            'level':'DEBUG',
            'class':'django.utils.log.NullHandler',
        },
        'console':{
            'level':'DEBUG',
            'class':'logging.StreamHandler',
            'formatter': 'simple'
        },
        'mail_admins': {
            'level': 'ERROR',
            'class': 'django.utils.log.AdminEmailHandler',
            'filters': ['special']
        }
    },
    'loggers': {
        'django': {
            'handlers':['null'],
            'propagate': True,
            'level':'INFO',
        },
        '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 。

19 利用 rotator 和 namer 自定义日志轮换操作

An example of how you can define a namer and rotator is given in the following runnable script, which shows gzip compression of the log file:

```
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=5)
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}')
```

After running this, you will see six new files, five of which are compressed:

```
$ 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
```

20 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
```

（下页继续）

(繼續上一頁)

```python
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):
            # The process name is transformed just to show that it's the listener
            # doing the logging to files and console
            record.processName = '%s (for %s)' % (current_process().name, record.
↪processName)
            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':
        # On POSIX, the setup logger will have been configured in the
        # parent process, but should have been disabled following the
        # dictConfig call.
        # On Windows, since fork isn't used, the setup logger won't
        # exist in the child, so it would be created and the message
        # would appear - hence the "if posix" clause.
        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.
```

(下页继续)

(繼續上一頁)

```
    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':
        # On POSIX, the setup logger will have been configured in the
        # parent process, but should have been disabled following the
        # dictConfig call.
        # On Windows, since fork isn't used, the setup logger won't
        # exist in the child, so it would be created and the message
        # would appear - hence the "if posix" clause.
        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()
    # The main process gets a simple configuration which prints to the console.
    config_initial = {
        'version': 1,
        'handlers': {
            'console': {
                'class': 'logging.StreamHandler',
                'level': 'INFO'
            }
        },
        'root': {
            'handlers': ['console'],
            'level': 'DEBUG'
        }
    }
    # The worker process configuration is just a QueueHandler attached to the
    # root logger, which allows all messages to be sent to the queue.
    # We disable existing loggers to disable the "setup" logger used in the
    # parent process. This is needed on POSIX because the logger will
    # be there in the child following a fork().
    config_worker = {
        'version': 1,
        'disable_existing_loggers': True,
        'handlers': {
            'queue': {
                'class': 'logging.handlers.QueueHandler',
                'queue': q
            }
        },
        'root': {
            'handlers': ['queue'],
            'level': 'DEBUG'
        }
    }
    # The listener process configuration shows that the full flexibility of
    # logging configuration is available to dispatch events to handlers however
```

(下页继续)

(繼續上一頁)

```
    # you want.
    # We disable existing loggers to disable the "setup" logger used in the
    # parent process. This is needed on POSIX because the logger will
    # be there in the child following a fork().
    config_listener = {
        'version': 1,
        'disable_existing_loggers': True,
        'formatters': {
            'detailed': {
                'class': 'logging.Formatter',
                'format': '%(asctime)s %(name)-15s %(levelname)-8s %(processName)-
↪10s %(message)s'
            },
            'simple': {
                'class': 'logging.Formatter',
                'format': '%(name)-15s %(levelname)-8s %(processName)-10s
↪%(message)s'
            }
        },
        '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'
        }
    }
    # Log some initial events, just to show that logging in the parent works
    # normally.
    logging.config.dictConfig(config_initial)
    logger = logging.getLogger('setup')
    logger.info('About to create workers ...')
    workers = []
```

(下页继续)

```
    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')
    # We now hang around for the workers to finish their work.
    for wp in workers:
        wp.join()
    # Workers all done, listening can now stop.
    # Logging in the parent still works normally.
    logger.info('Telling listener to stop ...')
    stop_event.set()
    lp.join()
    logger.info('All done.')

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

21 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。

22 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   # optional, to improve readability

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   # optional, to improve readability

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。

23 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':{
            # The values below are popped from this dictionary and
            # used to create the handler, set the handler's level and
            # its formatter.
            '()': owned_file_handler,
            'level':'DEBUG',
            'formatter': 'default',
            # The values below are passed to the handler creator callable
            # as keyword arguments.
            '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)
```

(下页继续)

```
    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':{
            # The values below are popped from this dictionary and
            # used to create the handler, set the handler's level and
            # its formatter.
            '()': owned_file_handler,
            'level':'DEBUG',
            'formatter': 'default',
            # The values below are passed to the handler creator callable
            # as keyword arguments.
            '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。

24 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 {TX-PL-LABEL}#x60; 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。

Logging calls (debug(), info() etc.) only take positional parameters for the actual logging message itself, with keyword parameters used only for determining options for how to handle the logging call (e.g. the exc_info keyword parameter to indicate that traceback information should be logged, or the extra keyword parameter to indicate additional contextual information to be added to the log). So you cannot directly make logging calls using str.format() or string.Template syntax, because internally the logging package uses %-formatting to merge the format string and the variable arguments. There would be no changing this while preserving backward compatibility, since all logging calls which are out there in existing code will be using %-format strings.

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。

24.1 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，它会通过重载 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。

24.2 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 arbitrary-object-messages），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
>>>
```

值得注意的是，上述做法对性能并没什么影响：格式化过程其实不是在日志调用时发生的，而是在日志信息即将由 handler 输出到日志时发生。因此，唯一可能让人困惑的稍不寻常的地方，就是包裹在格式串和参数外面的括号，而不是格式串。因为 __ 符号只是对 XXXMessage 类的构造函数调用的语法糖。

25 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',
        }
```

（下页继续）

(繼續上一頁)

```
    },
    '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 logging-config-dict-externalobj 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 logging-config-dict-userdef，以及上述利用 *dictConfig()* 自定义 *handler* 的其他指南。

26 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)  # or format into one line however you want to

    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')
```

(下页继续)

(繼續上一頁)

```
    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: integer division or modulo by zero|
↪'Traceback (most recent call last):\n  File "logtest7.py", line 30, in main\n  ␣
↪x = 1 / 0\nZeroDivisionError: integer division or modulo by zero'|
```

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。

27 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)
        # Speak slowly in a female English voice
        cmd = ['espeak', '-s150', '-ven+f3', msg]
        p = subprocess.Popen(cmd, stdout=subprocess.PIPE,
                             stderr=subprocess.STDOUT)
        # wait for the program to finish
        p.communicate()

def configure_logging():
    h = TTSHandler()
    root = logging.getLogger()
    root.addHandler(h)
    # the default formatter just returns the message
    root.setLevel(logging.DEBUG)

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

(下页继续)

(繼續上一頁)

```
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。

28 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_
↪handler)

    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()
```

(下页继续)

(繼續上一頁)

```
                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 ...
```

(下页继续)

(繼續上一頁)

```
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):
    ...
```

29 Sending logging messages to email, with buffering

To illustrate how you can send log messages via email, so that a set number of messages are sent per email, you can subclass `BufferingHandler`. In the following example, which you can adapt to suit your specific needs, a simple test harness is provided which allows you to run the script with command line arguments specifying what you typically need to send things via SMTP. (Run the downloaded script with the `-h` argument to see the required and optional arguments.)

```
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)s"))

    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,
↪ ','.join(self.toaddrs), self.subject)
                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__':
```

(下页继续)

(繼續上一頁)

```
    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()
```

If you run this script and your SMTP server is correctly set up, you should find that it sends eleven emails to the addressee you specify. The first ten emails will each have ten log messages, and the eleventh will have two messages. That makes up 102 messages as specified in the script.

30 通过配置使用 UTC (GMT) 格式化时间

Sometimes you want to format times using UTC, which can be done using a class such as `UTCFormatter`, shown below:

```
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',
        },
```

(下页继续)

(繼續上一頁)

```
        '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。

31 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)
```

(下页继续)

```
        if self.handler and self.close:
            self.handler.close()
        # implicit return of None => don't swallow exceptions
```

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␣
↪stdout.')
    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。

32 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()
    # the code to dispatch commands could all be in this file. For the purposes
    # of illustration only, we implement each command in a separate module.
    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
    # Could get fancy here and load configuration from file or dictionary
    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):
```

（下页继续）

```
    logger.debug('About to start %s', options.name)
    # actually do the command processing here ...
    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)
    # actually do the command processing here ...
    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)
    # actually do the command processing here ...
    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。

33 Qt GUI 日志示例

GUI 应用程序如何记录日志，这是个常见的问题。Qt 框架是一个流行的跨平台 UI 框架，采用的是 PySide2 或 PyQt5 库。

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。

以下代码应能适用于最新版的 PySide2 或 PyQt5。对于低版本的 Qt 应该也能适用。更多详情，请参阅代码注释。

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

# Deal with minor differences between PySide2 and PyQt5
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__)


#
# Signals need to be contained in a QObject or subclass in order to be correctly
# initialized.
#
```

（下页继续）

(繼續上一頁)

```
class Signaller(QtCore.QObject):
    signal = Signal(str, logging.LogRecord)

#
# Output to a Qt GUI is only supposed to happen on the main thread. So, this
# handler is designed to take a slot function which is set up to run in the main
# thread. In this example, the function takes a string argument which is a
# formatted log message, and the log record which generated it. The formatted
# string is just a convenience - you could format a string for output any way
# you like in the slot function itself.
#
# You specify the slot function to do whatever GUI updates you want. The handler
# doesn't know or care about specific UI elements.
#
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)

#
# This example uses QThreads, which means that the threads at the Python level
# are named something like "Dummy-1". The function below gets the Qt name of the
# current thread.
#
def ctname():
    return QtCore.QThread.currentThread().objectName()


#
# Used to generate random levels for logging.
#
LEVELS = (logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR,
          logging.CRITICAL)

#
# This worker class represents work that is done in a thread separate to the
# main thread. The way the thread is kicked off to do work is via a button press
# that connects to a slot in the worker.
#
# Because the default threadName value in the LogRecord isn't much use, we add
# a qThreadName which contains the QThread name as computed above, and pass that
# value in an "extra" dictionary which is used to update the LogRecord with the
# QThread name.
#
# This example worker just outputs messages sequentially, interspersed with
# random delays of the order of a few seconds.
#
class Worker(QtCore.QObject):
    @Slot()
    def start(self):
        extra = {'qThreadName': ctname() }
        logger.debug('Started work', extra=extra)
        i = 1
        # Let the thread run until interrupted. This allows reasonably clean
        # thread termination.
        while not QtCore.QThread.currentThread().isInterruptionRequested():
```

(下页继续)

(繼續上一頁)

```
            delay = 0.5 + random.random() * 2
            time.sleep(delay)
            level = random.choice(LEVELS)
            logger.log(level, 'Message after delay of %3.1f: %d', delay, i,␣
↪extra=extra)
            i += 1

#
# Implement a simple UI for this cookbook example. This contains:
#
# * A read-only text edit window which holds formatted log messages
# * A button to start work and log stuff in a separate thread
# * A button to log something from the main thread
# * A button to clear the log window
#
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)
        # Set whatever the default monospace font is for the platform
        f = QtGui.QFont('nosuchfont')
        f.setStyleHint(f.Monospace)
        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)
        # Remember to use qThreadName rather than threadName in the format string.
        fs = '%(asctime)s %(qThreadName)-12s %(levelname)-8s %(message)s'
        formatter = logging.Formatter(fs)
        h.setFormatter(formatter)
        logger.addHandler(h)
        # Set up to terminate the QThread when we exit
        app.aboutToQuit.connect(self.force_quit)

        # Lay out all the widgets
        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)

        # Connect the non-worker slots and signals
        self.log_button.clicked.connect(self.manual_update)
        self.clear_button.clicked.connect(self.clear_display)

        # Start a new worker thread and connect the slots for the worker
        self.start_thread()
```

(下页继续)

(繼續上一頁)

```
        self.work_button.clicked.connect(self.worker.start)
        # Once started, the button should be disabled
        self.work_button.clicked.connect(lambda : self.work_button.
↪setEnabled(False))

    def start_thread(self):
        self.worker = Worker()
        self.worker_thread = QtCore.QThread()
        self.worker.setObjectName('Worker')
        self.worker_thread.setObjectName('WorkerThread')  # for qThreadName
        self.worker.moveToThread(self.worker_thread)
        # This will start an event loop in the worker thread
        self.worker_thread.start()

    def kill_thread(self):
        # Just tell the worker to stop, then tell it to quit and wait for that
        # to happen
        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):
        # For use when the window is closed
        if self.worker_thread.isRunning():
            self.kill_thread()

    # The functions below update the UI and run in the main thread because
    # that's where the slots are set up

    @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):
        # This function uses the formatted message passed in, but also uses
        # information from the record to format the message in an appropriate
        # color according to its severity (level).
        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()
    sys.exit(app.exec_())

if __name__=='__main__':
```

(下页继续)

(繼續上一頁)

```
    main()
```

34 将日志记录到带有 RFC5424 支持的 syslog

虽然 **RFC 5424** 诞生于 2009 年，但大多数 syslog 服务器都默认被配置为使用更旧的 **RFC 3164**，它诞生于 2001 年。当 `logging` 在 2003 年被加入到 Python 时，它支持了当时（唯一存在的）较早版本的协议。在 RFC5424 诞生后，它还没有被广泛部署到 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句柄来实现:

```
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
            # This should be a dict where the keys are SD-ID and the value is a
            # dict mapping PARAM-NAME to PARAM-VALUE (refer to the RFC for what␣
↪these
            # mean)
            # There's no error checking here - it's purely for illustration, and␣
↪you
            # can adapt this code for use in production environments
```

(下页继续)

(繼續上一頁)

```
            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}
↪{msg}'
```

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。通过上面的处理句柄，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)
```

35 How to treat a logger like an output stream

Sometimes, you need to interface to a third-party API which expects a file-like object to write to, but you want to direct the API's output to a logger. You can do this using a class which wraps a logger with a file-like API. Here's a short script illustrating such a class:

```
import logging

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

    def write(self, message):
        if message != '\n':  # avoid printing bare newlines, if you like
            self.logger.log(self.level, message)

    def flush(self):
        # doesn't actually do anything, but might be expected of a file-like
        # object - so optional depending on your situation
        pass

    def close(self):
        # doesn't actually do anything, but might be expected of a file-like
        # object - so optional depending on your situation. You might want
        # to set a flag so that later calls to write raise an exception
        pass
```

(下页继续)

(繼續上一頁)

```
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()
```

When this script is run, it prints

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

You could also use `LoggerWriter` to redirect `sys.stdout` and `sys.stderr` by doing something like this:

```
import sys

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

You should do this *after* configuring logging for your needs. In the above example, the `basicConfig()` call does this (using the `sys.stderr` value *before* it is overwritten by a `LoggerWriter` instance). Then, you'd get this kind of result:

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

Of course, these above examples show output according to the format used by `basicConfig()`, but you can use a different formatter when you configure logging.

Note that with the above scheme, you are somewhat at the mercy of buffering and the sequence of write calls which you are intercepting. For example, with the definition of `LoggerWriter` above, if you have the snippet

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

then running the script results in

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

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

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
```

As you can see, this output isn't ideal. That's because the underlying code which writes to `sys.stderr` makes mutiple writes, each of which results in a separate logged line (for example, the last three lines above). To get around this problem, you need to buffer things and only output log lines when newlines are seen. Let's use a slghtly better implementation of LoggerWriter:

```
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)
```

This just buffers up stuff until a newline is seen, and then logs complete lines. With this approach, you get better output:

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

36 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。

36.1 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。
- An attempt to delete a file (e.g. during file rotation) silently fails, because there is another reference pointing to it. This can lead to confusion and wasted debugging time - log entries end up in unexpected places, or are lost altogether. Or a file that was supposed to be moved remains in place, and grows in size unexpectedly despite size-based rotation being supposedly in place.

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。

36.2 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）。

36.3 给日志库代码添加 `NullHandler` 之外的其他 handler

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。

36.4 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。

37 其他资源

也参考:

`logging` 模組 日志记录模块的 API 参考。

`logging.config` 模組 日志记录模块的配置 API 。

`logging.handlers` 模組 日志记录模块附带的有用处理器。

基础教程

进阶教程

索引

R