日志操作手册

發🄵 *3.9.6*

Guido van Rossum
and the Python development team

8 月 30, 2021

Python Software Foundation
Email: docs@python.org

Contents

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

本页包含了许多日志记录相关的概念，这些概念在过去一直被认为很有用。

1 在多个模块中使用日志

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

```
import logging
import auxiliary_module

# create logger with 'spam_application'
logger = logging.getLogger('spam_application')
logger.setLevel(logging.DEBUG)
```

(下页继续)

(繼續上一頁)

```
# 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 -
   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
```

(下页继续)

(繼續上一頁)

```
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()
```

运行结果会像如下这样：

```
   0 Thread-1 Hi from myfunc
   3 MainThread Hello from main
 505 Thread-1 Hi from myfunc
 755 MainThread Hello from main
1007 Thread-1 Hi from myfunc
1507 MainThread Hello from main
1508 Thread-1 Hi from myfunc
2010 Thread-1 Hi from myfunc
2258 MainThread Hello from main
2512 Thread-1 Hi from myfunc
```

(下页继续)

(繼續上一頁)

```
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：

```
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='/temp/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。

5 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')
```

6 处理日志 handler 的阻塞

有时需让日志 handler 不要阻塞当前的线程。这在 Web 应用程序中比较常见，当然在其他场景中也会发生。

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!
```

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 。

7 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)
```

(下页继续)

(繼續上一頁)

```
            while len(chunk) < slen:
                chunk = chunk + self.connection.recv(slen - len(chunk))
            obj = self.unPickle(chunk)
            record = logging.makeLogRecord(obj)
            self.handleLogRecord(record)

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

    def handleLogRecord(self, record):
        # 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()
```

(下页继续)

(繼續上一頁)

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

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

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

请注意，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 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专属的信息（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。

8.1 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。

若要创建 `LoggerAdapter` 的实例，需传入一个 `Logger` 的实例和一个包含了上下文信息的字典类对象。当在 `LoggerAdapter` 实例上调用某个日志方法时，会将调用委托给传入构造函数的底层 `Logger` 实例，并在调用中传入上下文信息。以下是 `LoggerAdapter` 的一段代码：

```
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()` 方法用于将上下文信息添加到日志的输出中去。其入参为日志调用的信息和关键字参数，并将（可能）修改后的入参值传回，以供底层的日志对象调用。该方法的默认实现代码不会对信息做改动，而只是在关键字参数中增加一个键为“extra”的字段，其值即为传入构造函数的字典类对象。当然，若是在调用时传入了一个名为“extra”的关键字参数，则传入值会被悄无声息地覆盖掉。

使用“extra”的好处，就是字典类对象中的值会被并入 `LogRecord` 实例的 `__dict__` 中，这样就能利用 `Formatter` 实例使用自定义字符串了，Formatter 知道该如何使用字典类对象的键。若要用到其他方法，比如想在信息字符串中预置或追加上下文信息，只需继承 `LoggerAdapter` 并覆盖 `process()` 方法即可完成处理。下面给出一个简单的示例：

```
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
```

用法可如下所示：

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

这样，只要是该 Adapter 处理的任何日志事件，消息前都会加上 “some_conn_id“ 的值。

利用非字典对象传入上下文信息

传给 LoggerAdapter 的不一定要是真正的字典对象，也可以传入一个实现了 __getitem__ 和 “__iter__“ 方法的类实例，类似要写入日志的字典对象。若要动态生成值（而字典中的值应为常量），这就会很有用。

8.2 利用 Filter 传递上下文信息

还可以利用用户定义类 Filter 在日志输出中添加上下文信息。Filter 实例可以修改传入的 LogRecords，包括添加额外的属性，然后可以采用合适的格式化字符串对这些属性进行输出，必要时还可采用自定义类 Formatter。

例如在某 web 应用程序中，正在处理的请求（或至少是当前关注的部分），可存储于线程本地 (threading.local) 变量中，然后从 “Filter“ 中去访问，把请求中的一些信息添加进去，比如在 LogRecord 中写入远程 IP 地址和远程用户名，可利用以上 LoggerAdapter 示例中的 “ip” 和 “user” 属性名。这时可采用与上例相同的格式化字符串来得到类似的输出结果。以下是一段示例代码：

```
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)
```

(下页继续)

(繼續上一頁)

```
    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
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
```

9 在单个文件中记录多个进程的日志

尽管日志对象是线程安全的，也确实支持将一个进程中的多个线程日志记入单个文件，但不支持将多进程日志记入单个文件，因为 Python 没有标准方案来实现由多个进程串行访问单个文件。若要将多个进程的日志记入单个文件，一种方案是让所有进程都用一个 `SocketHandler` 处理日志，然后用一个单独的进程实现套接字服务器，一边从套接字读取数据一边向文件中写入日志。(当然可在某个现有进程中指定一个线程来执行此项功能。) 此部分 更详细地介绍了这种做法，包含了一个套接字接收器，可以此为起点建立适合自己应用程序的代码。

编写自己的 handler 也是可以的，可利用 `multiprocessing` 模块中的 `Lock` 类实现多个进程串行访问文件。现有的 `FileHandler` 及其子类目前没有用到 `multiprocessing`，或许将来有可能会吧。请注意，目前

multiprocessing模块并未在所有平台上都提供可用的同步锁功能(参见 https://bugs.python.org/issue3770)。

或者，还可以利用 Queue 和 QueueHandler 将所有的日志事件发送给自己的多进程应用中的某个进程。以下例程演示了这种做法，这里有一个单独的监听进程负责监听其他进程发来的日志事件，并根据自己的日志配置记入日志。尽管本例程只演示了一种实现方案（比如可能想用单独的监听线程而非进程——实现方式其实类似），但确实可以为应用程序的监听进程和其他进程采用不同的配置，作为满足特定需求代码的基础：

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

(下页继续)

(繼續上一頁)

```
# 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',
    '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()
```

(下页继续)

(繼續上一頁)

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

上述代码可做个变化，在主进程中用单独的线程记录日志：

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

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


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

if __name__ == '__main__':
    q = Queue()
    d = {
        'version': 1,
        'formatters': {
            'detailed': {
                'class': 'logging.Formatter',
                'format': '%(asctime)s %(name)-15s %(levelname)-8s %(processName)-10s
↪%(message)s'
            }
        },
        'handlers': {
            'console': {
                'class': 'logging.StreamHandler',
                'level': 'INFO',
            },
            'file': {
                'class': 'logging.FileHandler',
                'filename': 'mplog.log',
```

(下页继续)

(繼續上一頁)

```
                'mode': 'w',
                'formatter': 'detailed',
            },
            'foofile': {
                'class': 'logging.FileHandler',
                'filename': 'mplog-foo.log',
                'mode': 'w',
                'formatter': 'detailed',
            },
            'errors': {
                'class': 'logging.FileHandler',
                'filename': 'mplog-errors.log',
                'mode': 'w',
                'level': 'ERROR',
                'formatter': 'detailed',
            },
        },
        'loggers': {
            'foo': {
                'handlers': ['foofile']
            }
        },
        'root': {
            'level': 'DEBUG',
            'handlers': ['console', 'file', 'errors']
        },
    }
    workers = []
    for i in range(5):
        wp = Process(target=worker_process, name='worker %d' % (i + 1), args=(q,))
        workers.append(wp)
        wp.start()
    logging.config.dictConfig(d)
    lp = threading.Thread(target=logger_thread, args=(q,))
    lp.start()
    # 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()
```

这段改过的代码展示了如何为某日志对象应用指定配置——比如 `foo` 日志对象有个特别的 handler，将 `foo` 子系统的所有事件保存至文件 `mplog-foo.log` 中。主进程的日志机制将会用到这段代码（即便日志事件是在其他的工作进程中产生的），将信息定向输出到指定的地方。

9.1 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)
```

10 利用日志文件轮换机制

有时需要让日志文件增长到指定大小，然后打开一个新文件并记入日志。或许还需要只保留一定数量的日志文件，当创建了指定数量的文件后，就轮换使用这些文件，以便让文件数量和大小都维持上限。logging 包为这种使用模式提供了 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)
```

(下页继续)

(繼續上一頁)

```
# 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 `logging_rotatingfile_example.out`，每次达到大小限制时，都会用后缀 `.1` 重新命名。已有的备份文件全都会被重命名，将后缀递增（如 `.1` 变为 `.2`），而 `.6` 文件则会被删除。

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。

11 日志的其他格式

当日志模块刚加入 Python 标准库时，想要格式化输出带有可变内容的日志信息，只有一种%-f 方法。后来 Python 又有了两种格式化方法：`string.Template`（Python 2.4 加入）和 `str.format()`（Python 2.6 加入）。

从 Python 3.2 开始，日志模块为后加入的两种格式化方式提供了更多支持。`Formatter` 得以增强，可以接受名为 `style` 的可选关键字参数。其默认值为 `'%'`，其他还可以是 `'{'` 和 `'{TX-PL-LABEL}#x27;`，对应于另外两种格式化的样式。如您所料，默认保持向下兼容，而通过显式指定样式参数，能够设置用于 `str.format()` 或 `string.Template` 的格式串。下面是个控制台会话示例，演示一下功能:

```
>>> 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，最终输出到日志的信息格式与某一条信息的构造方式完全独立。单条信息仍然可以采用%-f 格式，如下所示:

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

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

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

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

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

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

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

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

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

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

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

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():
    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。

12 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。

13 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)
```

(下页继续)

(繼續上一頁)

```
        socket.bind(uri)
        super().__init__(socket)

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

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

14 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:

模块 `logging` 日志记录模块的 API 参考。

模块 `logging.config` 日志记录模块的配置 API 。

模块 `logging.handlers` 日志记录模块中的常用 handler。

日志操作基础教程

日志操作的高级教程

15 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'
```

(下页继续)

(繼續上一頁)

```
        },
    },
    '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 。

16 利用 rotator 和 namer 自定义日志轮换操作

以下代码给出了定义 namer 和 rotator 的示例，其中演示了基于 zlib 的日志文件压缩过程：

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

def rotator(source, dest):
    with open(source, "rb") as sf:
```

(下页继续)

(繼續上一頁)

```
        data = sf.read()
        compressed = zlib.compress(data, 9)
        with open(dest, "wb") as df:
            df.write(compressed)
    os.remove(source)

rh = logging.handlers.RotatingFileHandler(...)
rh.rotator = rotator
rh.namer = namer
```

这些不是“真正的”.gz 文件，因为他们只是纯压缩数据，缺少真正 gzip 文件中的“容器”。此段代码只是用于演示。

17 更详细的多进程日志示例

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。

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

以下是代码——但愿文档字符串和注释能有助于理解其工作原理：

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

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

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

        if logger.isEnabledFor(record.levelno):
            # 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)
```

(下页继续)

(繼續上一頁)

```
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.

    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)
```

(下页继续)

(繼續上一頁)

```
        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
    # 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'
            }
```

(下页继续)

(繼續上一頁)

```
        },
        '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 = []
    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.
```

(下页继续)

(繼續上一頁)

```
    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()
```

18 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. ASCII 部分可替换为任意字符，但要保证替换后的数据一定是 ASCII 码（这样在 UTF-8 编码后就会维持不变）。

3. Unicode 部分可替换为任意字符；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。

19 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
```

(下页继续)

(繼續上一頁)

```
    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：

```
from __future__ import unicode_literals

import json
import logging

# This next bit is to ensure the script runs unchanged on 2.x and 3.x
try:
    unicode
except NameError:
    unicode = str

class Encoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, set):
            return tuple(o)
        elif isinstance(o, unicode):
            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。

20 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:
```

(下页继续)

(繼續上一頁)

```
        if not os.path.exists(filename):
            open(filename, 'a').close()
        shutil.chown(filename, *owner)
    return logging.FileHandler(filename, mode, encoding)

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': '%(asctime)s %(levelname)s %(name)s %(message)s'
        },
    },
    'handlers': {
        'file':{
            # 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。

21 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。

日志函数（debug(), info() 等）只会读取位置参数获取日志信息本身，而关键字参数仅用于确定日志函数的工作选项（比如关键字参数 exc_info 表示应将跟踪信息记入日志，关键字参数 extra 则给出了需加入日志的额外上下文信息）。所以不能直接使用 str.format() 或 string.Template 这种语法进行日志调用，因为日志包在内部使用%-f 格式来合并格式串和可变参数。因为尚需保持向下兼容，这一点不会改变，已有代码中的所有日志调用都将采用%-f 格式串。

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

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

21.1 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。

21.2 自定义日志信息对象的使用

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
```

(下页继续)

(繼續上一頁)

```
    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。

22 用 dictConfig() 配置过滤器

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):
```

(下页继续)

(繼續上一頁)

```
    def __init__(self, param=None):
        self.param = param

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

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

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

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

```
changed: hello
```

这说明过滤器按照配置的参数生效了。

需要额外注意的地方：

- 如果在配置中无法直接引用可调用对象（比如位于不同的模块中，并且不能在配置字典所在的位置直接导入），则可以采用 `ext://...` 的形式，正如 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。

23 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')
    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。

24 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')

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，甚至可以利用命令行运行的外部程序来处理日志信息。

25 缓冲日志信息并按条件输出

有时可能需要将日志信息写入临时位置，仅在特定情况下才进行输出。比如在函数中记录调试事件，若函数执行完成且没有错误，收集到的调试信息就无需输出，以免造成日志混乱，但在出现错误时则需要输出全部的调试和错误信息。

以下例子展示了如何在日志函数上使用装饰器，以实现上述功能。这里用到了 `logging.handlers.MemoryHandler`，将日志事件缓存下来，直至发生某些状况时，缓存的事件才会被送出（`flushed`）——传给另一个 handler（`target`）进行处理。默认情况下，`MemoryHandler` 在缓冲区满时，或者事件级别大于等于指定阈值的，会将数据刷出。若需自定义刷新行为，可以利用定制 `MemoryHandler` 子类来实现。

以下例程包含一个简单的 `foo` 函数，它只是循环运行于全部日志级别上，写入 `sys.stderr`，输出信息包括日志级别和日志消息。可为 `foo` 传入一个参数，true 则在 ERROR 和 CRITICAL 级别记录日志，否则只在 DEBUG、INFO 和 WARNING 级别记录日志。

这里只是为 foo 加了个装饰器，进行有条件的日志记录。该装饰器以日志对象为参数，并在调用被装饰函数期间增加一个内存处理 handler。该装饰器的参数还可以加上目标 handler、日志级别和缓冲区的容量（可缓存的日志条数量）。这些参数的默认值分别是写入 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()
                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__':
```

(下页继续)

(繼續上一頁)

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

运行此脚本时，应看到以下输出：

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

由上可见，只有发生 ERROR 以上级别的事件时才会实际输出日志，但之前发生的低等级事件还是会被记入日志。

当然采用传统的装饰方法也是可以的：

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

26 通过配置将时间格式化为 UTC（GMT）格式

有时需要将时间格式化为 UTC 格式，可以用类似 *UTCFormatter* 的类来完成，如下所示：

```
import logging
import time

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

然后自己的代码中即可采用 `UTCFormatter` 了，而不用 `Formatter` 了。若要通过配置来实现，可以用 `dictConfig()` API 来完成，以下完整示例中将会演示这种方案：

```
import logging
import logging.config
import time

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

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

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

脚本会运行输出类似下面的内容:

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

以上将时间格式化为本地时间和 UTC 两种形式，每种形式对应一个 handler。

27 用上下文管理器选择性记录日志

有时需要临时更改日志配置，并在执行某些操作后还原配置。这时，上下文管理器正是最明显的方案，可以实现日志上下文的保存和恢复。以下是一个上下文管理器的简单示例，只在上下文管理器的作用域内更改日志等级和增加日志 handler：

```
import logging
import sys

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

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

    def __exit__(self, et, ev, tb):
        if self.level is not None:
            self.logger.setLevel(self.old_level)
        if self.handler:
            self.logger.removeHandler(self.handler)
        if self.handler and self.close:
            self.handler.close()
        # implicit return of None => don't swallow exceptions
```

如果给出了 level 值，则在上下文管理器 with 语句覆盖的作用域内，日志记录级别将设置为 level 值。如果给出了 handler 值，则在进入作用域时将会将其加入，离开时将会移除。如果后续用不到该 handler 了，则可在离开作用域时让上下文管理器将其关闭。

为了演示上述过程，可加入以下代码：

```
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.')
```

最开始日志级别设为 INFO，因此会显示 #1 信息，而 #2 信息则没有出现。接下来的 with 代码块中，暂时将日志级别变为 DEBUG，于是 #3 信息出现了。在离开该代码块后，日志记录器的级别恢复为 INFO，从而 #4 信息没有出现。在下一个 with 代码块中，再次将日志级别设为 DEBUG，并且加入一个写入 sys.stdout

的日志 handler。因此，#5 信息在控制台出现了两次（分别由 stderr 和 stdout 显示）。在 with 语句执行完毕后，状态复原，因此显示了 #6 信息（类似 #1），而 #7 信息没有出现（类似 #2）。

运行结果如下：

```
$ 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.
```

但若将 stderr 重定向到 /dev/null，再次运行一下，写入 stdout 的将只有以下信息：

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

再把 stdout 重定向到 /dev/null，结果将如下所示：

```
$ 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.
```

这符合预期，输出到 stdout 的 #5 信息不会显示出来。

当然，上述做法可以推广到其他应用，比如临时加入日志过滤器。请注意，上述代码对 Python 2 和 3 均适用。

28 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，同一级别的日志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')
```

(下页继续)

(繼續上一頁)

```
    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):
    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:]
```

(下页继续)

(繼續上一頁)

```
    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。

29 用作日志的 Qt GUI 程序

如何将日志写入某个 GUI 应用程序，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.
#
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.
#
```

(下页继续)

(繼續上一頁)

```
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():
            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:
```

(下页继续)

(繼續上一頁)

```
#
# * 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()
        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))
```

(下页继续)

(繼續上一頁)

```
    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__':
    main()
```

30 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。

30.1 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。
- 删除文件（例如在轮换日志文件时）会静默失败，因为有另一个引用指向这个文件。这可能导致混乱并浪费调试时间——日志项最后会出现在意想不到的地方，或者干脆丢失。

请用*在单个文件中记录多个进程的日志*中介绍的技术来避免上述问题。

30.2 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）。

30.3 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。

30.4 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。

索引

R