日志操作手册

發🄵 *3.10.0a0*

Guido van Rossum
and the Python development team

5 月 20, 2020

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

多次调用 `logging.getLogger('someLogger')` 时会返回对同一个 logger 对象的引用。这不仅是在同一个模块中，在其他模块调用也是如此，只要是在同一个 Python 解释器进程中。是应该引用同一个对象，此外，应用程序也可以在一个模块中定义和配置父 logger，而在单独的模块中创建（但不配置）子 logger，对子 logger 的所有调用都将传给父 logger。这里是主模块:

```
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)
```

(下页继续)

(繼續上一頁)

```
# 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()')
```

这里是辅助模块:

```
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
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()
```

(下页继续)

(繼續上一頁)

```
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 在多线程中使用日志

在多个线程中记录日志并不需要特殊处理，以下示例展示了如何在主线程（起始线程）和其他线程中记录:

```
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
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
```

(下页继续)

(繼續上一頁)

```
4513 MainThread Hello from main
4518 Thread-1 Hi from myfunc
```

这表明不同线程的日志像期望的那样穿插输出，当然更多的线程也会像这样输出。

3 使用多个日志处理器和多种格式化

日志记录器是普通的 Python 对象。`addHandler()` 方法没有限制可以添加的日志处理器数量。有时候，应用程序需要将严重类的消息记录在一个文本文件，而将错误类或其他等级的消息输出在控制台中。要进行这样的设定，只需多配置几个日志处理器即可，在应用程序代码中的日志记录调用可以保持不变。以下是对之前的基于模块配置的示例的略微修改:

```
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' 代码并不关心是否有多个日志处理器。示例中所做的改变只是添加和配置了一个新的名为 *fh* 的日志处理器。

在编写和测试应用程序时，创建能过滤不同等级消息的日志处理器是很有用的。不要去使用 `print` 去调试，而是使用 `logger.debug`: 它不像打印语句需要在调试结束后注释或删除掉，你可以把它们保留在源码中并不输出。当需要再次调试时，只需要改变日志记录器或处理器的过滤等级即可。

4 在多个地方记录日志

假设有这样一种情况，你需要将日志按不同的格式和不同的情况存储在控制台和文件中。比如说想把日志等级为 DEBUG 或更高的消息记录于文件中，而把那些等级为 INFO 或更高的消息输出在控制台。而且记录在文件中的消息格式需要包含时间戳，打印在控制台的不需要。以下示例展示了如何做到:

```
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.
```

而在文件中会看到像这样

```
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.
```

正如你所看到的，DEBUG 级别的消息只展示在文件中，而其他消息两个地方都会输出。

这个示例只演示了在控制台和文件中去记录日志，但你也可以自由组合任意数量的日志处理器。

5 日志服务器配置示例

以下是在一个模块中使用日志服务器配置的示例:

```
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()
```

然后如下的脚本，它接收文件名做为命令行参数，并将该文件以二进制编码的方式传给服务器，做为新的日志服务器配置:

```
#!/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 处理日志处理器的阻塞

有时候需要让日志处理程序在不阻塞当前正在记录线程的情况下完成工作。这在 Web 应用程序中很常见，当然也会在其他场景中出现。

一个常见的缓慢行为是 SMTPHandler: 由于开发者无法控制的多种原因（例如，性能不佳的邮件或网络基础架构），发送电子邮件可能需要很长时间。其实几乎所有基于网络的处理程序都可能造成阻塞：即便是 SocketHandler 也可能在底层进行 DNS 查询，这太慢了（这个查询会深入至套接字代码，位于 Python 层之下，这是不受开发者控制的）。

一种解决方案是分成两部分去处理。第一部分，针对那些对性能有要求的关键线程的日志记录附加一个 QueueHandler。日志记录器只需简单写入队列，该队列可以设置一个足够大的容量甚至不设置容量上限。通常写入队列是一个快速的操作，即使可能需要在代码中去捕获例如 queue.Full 等异常。如果你是一名处理关键线程的开发者，请务必记录这些信息 (包括建议只为日志处理器附加 QueueHandlers) 以便于其他开发者使用你的代码。

解决方案的另一部分是 QueueListener，它被设计用来作为 QueueHandler 的对应。QueueListener 非常简单：向其传入一个队列和一些处理句柄，它会启动一个内部线程来监听从 QueueHandlers (或任何其他可用的 LogRecords 源) 发送过来的 LogRecords 队列。LogRecords 会从队列中被移除，并被传递给句柄进行处理。

使用一个单独的类 QueueListener 优点是可以使用同一个实例去服务于多个 “QueueHandlers“。这样会更节省资源，否则每个处理程序都占用一个线程没有任何益处。

以下是使用了这样两个类的示例 (省略了导入语句):

```
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 总是把从队列中接收的每个消息都传给它初始化的日志处理程序。(这是因为它会假设过滤级别总是在队列的另一侧去设置的。) 从 Python 3.5 开始，可以通过在监听器构造函数中添加一个参数 respect_handler_level=True 改变这种情况。当这样设置时，监听器会比较每条消息的等级和日志处理器中设置的等级，只把需要传递的消息传给对应的日志处理器。

7 通过网络发送和接收日志

如果你想在网络上发送日志，并在接收端处理它们。一个简单的方式是通过附加一个 SocketHandler 的实例在发送端的根日志处理器中:

```
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，你可以使用 socketserver 模块设置一个接收器。这里是一个基础示例:

```
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()
```

首先运行服务端，然后是客户端。在客户端，没有什么内容会打印在控制台中；在服务端，你应该会看到如下内容：

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

请注意，在某些情况下序列化会存在一些安全。如果这影响到你，那么你可以通过覆盖 makePickle() 方法，使用自己的实现来解决，并调整上述脚本也使用覆盖后的序列化方法。

8 在日志记录中添加上下文信息

有时，除了传递给日志记录器调用的参数外，我们还希望日志记录中包含上下文信息。例如，有一个网络应用，可能需要记录一些特殊的客户端信息在日志中（比如客户端的用户名、IP 地址等）。虽然你可以通过设置额外的参数去达到这个目的，但这种方式不一定方便。或者你可能想到在每个连接的基础上创建一个 Logger 的实例，但这些实例是不会被垃圾回收的，这在练习中也许不是问题，但当 Logger 的实例数量取决于你应用程序中想记录的细致程度时，如果 Logger 的实例数量不受限制的话，将会变得难以管理。

8.1 使用日志适配器传递上下文信息

一个传递上下文信息和日志事件信息的简单办法是使用类 LoggerAdapter。这个类设计的像 Logger，所以可以直接调用 debug()、info()、warning()、error()、exception()、critical() 和 log()。这些方法在对应的 Logger 中使用相同的签名，所以可以交替使用两种类型的实例。

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。

8.2 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)
```

(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 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。

You could also write your own handler which uses the `Lock` class from the `multiprocessing` module to serialize access to the file from your processes. The existing `FileHandler` and subclasses do not make use of

multiprocessing at present, though they may do so in the future. Note that at present, the multiprocessing module does not provide working lock functionality on all platforms (see 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
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)
```

(下页继续)

(繼續上一頁)

```
            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',
    '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()
```

(下页继续)

(繼續上一頁)

```
    queue.put_nowait(None)
    listener.join()

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

上面脚本的一个变种，仍然在主进程中记录日志，但使用一个单独的线程:

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

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


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

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

(下页继续)

(繼續上一頁)

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

9.1 Using concurrent.futures.ProcessPoolExecutor

If you want to use `concurrent.futures.ProcessPoolExecutor` to start your worker processes, you need to create the queue slightly differently. Instead of

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

you should use

```
queue = multiprocessing.Manager().Queue(-1)  # also works with the examples above
```

and you can then replace the worker creation from this:

```
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()
```

to this (remembering to first import `concurrent.futures`):

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

10 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)
```

(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: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。

11 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
>>>
```

Logging calls (`logger.debug()`, `logger.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 actual 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.

There is, however, a way that you can use {}- and $- formatting to construct your individual log messages. Recall that for a message you can use an arbitrary object as a message format string, and that the logging package will call `str()` on that object to get the actual format string. Consider the following two classes:

```
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)
```

Either of these can be used in place of a format string, to allow {}- or $-formatting to be used to build the actual "message" part which appears in the formatted log output in place of "%(message)s" or "{message}" or "$message". It's a little unwieldy to use the class names whenever you want to log something, but it's quite palatable if you use an alias such as __ (double underscore --- not to be confused with _, the single underscore used as a synonym/alias for `gettext.gettext()` or its brethren).

The above classes are not included in Python, though they're easy enough to copy and paste into your own code. They can be used as follows (assuming that they're declared in a module called `wherever`):

```
>>> 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
```

(下页继续)

(繼續上一頁)

```
>>>
```

While the above examples use `print()` to show how the formatting works, you would of course use `logger.debug()` or similar to actually log using this approach.

One thing to note is that you pay no significant performance penalty with this approach: the actual formatting happens not when you make the logging call, but when (and if) the logged message is actually about to be output to a log by a handler. So the only slightly unusual thing which might trip you up is that the parentheses go around the format string and the arguments, not just the format string. That's because the __ notation is just syntax sugar for a constructor call to one of the XXXMessage classes.

If you prefer, you can use a `LoggerAdapter` to achieve a similar effect to the above, as in the following example:

```
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(StyleAdapter, self).__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()
```

The above script should log the message `Hello, world!` when run with Python 3.2 or later.

12 Customizing `LogRecord`

Every logging event is represented by a `LogRecord` instance. When an event is logged and not filtered out by a logger's level, a `LogRecord` is created, populated with information about the event and then passed to the handlers for that logger (and its ancestors, up to and including the logger where further propagation up the hierarchy is disabled). Before Python 3.2, there were only two places where this creation was done:

- `Logger.makeRecord()`, which is called in the normal process of logging an event. This invoked `LogRecord` directly to create an instance.
- `makeLogRecord()`, which is called with a dictionary containing attributes to be added to the LogRecord. This is typically invoked when a suitable dictionary has been received over the network (e.g. in pickle form via a

SocketHandler, or in JSON form via an HTTPHandler).

This has usually meant that if you need to do anything special with a LogRecord, you've had to do one of the following.

- Create your own Logger subclass, which overrides Logger.makeRecord(), and set it using setLoggerClass() before any loggers that you care about are instantiated.
- Add a Filter to a logger or handler, which does the necessary special manipulation you need when its filter() method is called.

The first approach would be a little unwieldy in the scenario where (say) several different libraries wanted to do different things. Each would attempt to set its own Logger subclass, and the one which did this last would win.

The second approach works reasonably well for many cases, but does not allow you to e.g. use a specialized subclass of LogRecord. Library developers can set a suitable filter on their loggers, but they would have to remember to do this every time they introduced a new logger (which they would do simply by adding new packages or modules and doing

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

at module level). It's probably one too many things to think about. Developers could also add the filter to a NullHandler attached to their top-level logger, but this would not be invoked if an application developer attached a handler to a lower-level library logger --- so output from that handler would not reflect the intentions of the library developer.

In Python 3.2 and later, LogRecord creation is done through a factory, which you can specify. The factory is just a callable you can set with setLogRecordFactory(), and interrogate with getLogRecordFactory(). The factory is invoked with the same signature as the LogRecord constructor, as LogRecord is the default setting for the factory.

This approach allows a custom factory to control all aspects of LogRecord creation. For example, you could return a subclass, or just add some additional attributes to the record once created, using a pattern similar to this:

```
old_factory = logging.getLogRecordFactory()

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

logging.setLogRecordFactory(record_factory)
```

This pattern allows different libraries to chain factories together, and as long as they don't overwrite each other's attributes or unintentionally overwrite the attributes provided as standard, there should be no surprises. However, it should be borne in mind that each link in the chain adds run-time overhead to all logging operations, and the technique should only be used when the use of a Filter does not provide the desired result.

13 Subclassing QueueHandler - a ZeroMQ example

You can use a QueueHandler subclass to send messages to other kinds of queues, for example a ZeroMQ 'publish' socket. In the example below,the socket is created separately and passed to the handler (as its '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
```

(下页继续)

(繼續上一頁)

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


handler = ZeroMQSocketHandler(sock)
```

Of course there are other ways of organizing this, for example passing in the data needed by the handler to create the 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()
```

14 Subclassing QueueListener - a ZeroMQ example

You can also subclass `QueueListener` to get messages from other kinds of queues, for example a ZeroMQ 'subscribe' socket. Here's an example:

```
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)
```

也參考:

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` 日志记录模块附带的有用处理程序。

A basic logging tutorial

A more advanced logging tutorial

15 An example dictionary-based configuration

Below is an example of a logging configuration dictionary - it's taken from the documentation on the Django project. This dictionary is passed to `dictConfig()` to put the configuration into effect:

```
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']
        }
    }
}
```

For more information about this configuration, you can see the relevant section of the Django documentation.

16 Using a rotator and namer to customize log rotation processing

An example of how you can define a namer and rotator is given in the following snippet, which shows zlib-based compression of the log file:

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

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

These are not ”true” .gz files, as they are bare compressed data, with no ”container” such as you’d find in an actual gzip file. This snippet is just for illustration purposes.

17 A more elaborate multiprocessing example

The following working example shows how logging can be used with multiprocessing using configuration files. The configurations are fairly simple, but serve to illustrate how more complex ones could be implemented in a real multiprocessing scenario.

In the example, the main process spawns a listener process and some worker processes. Each of the main process, the listener and the workers have three separate configurations (the workers all share the same configuration). We can see logging in the main process, how the workers log to a QueueHandler and how the listener implements a QueueListener and a more complex logging configuration, and arranges to dispatch events received via the queue to the handlers specified in the configuration. Note that these configurations are purely illustrative, but you should be able to adapt this example to your own scenario.

Here’s the script - the docstrings and the comments hopefully explain how it works:

```
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
```

(下页继续)

(繼續上一頁)

```
    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.

    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,
```

(下页继续)

(繼續上一頁)

```
                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
    # you want.
```

(下页继续)

(繼續上一頁)

```
    # 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)
```

(下页继续)

(繼續上一頁)

```
    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.
    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 Inserting a BOM into messages sent to a SysLogHandler

RFC 5424 requires that a Unicode message be sent to a syslog daemon as a set of bytes which have the following structure: an optional pure-ASCII component, followed by a UTF-8 Byte Order Mark (BOM), followed by Unicode encoded using UTF-8. (See the **relevant section of the specification**.)

In Python 3.1, code was added to `SysLogHandler` to insert a BOM into the message, but unfortunately, it was implemented incorrectly, with the BOM appearing at the beginning of the message and hence not allowing any pure-ASCII component to appear before it.

As this behaviour is broken, the incorrect BOM insertion code is being removed from Python 3.2.4 and later. However, it is not being replaced, and if you want to produce RFC 5424-compliant messages which include a BOM, an optional pure-ASCII sequence before it and arbitrary Unicode after it, encoded using UTF-8, then you need to do the following:

1. Attach a `Formatter` instance to your `SysLogHandler` instance, with a format string such as:

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

   The Unicode code point U+FEFF, when encoded using UTF-8, will be encoded as a UTF-8 BOM -- the byte-string `b'\xef\xbb\xbf'`.

2. Replace the ASCII section with whatever placeholders you like, but make sure that the data that appears in there after substitution is always ASCII (that way, it will remain unchanged after UTF-8 encoding).

3. Replace the Unicode section with whatever placeholders you like; if the data which appears there after substitution contains characters outside the ASCII range, that's fine -- it will be encoded using UTF-8.

The formatted message *will* be encoded using UTF-8 encoding by `SysLogHandler`. If you follow the above rules, you should be able to produce **RFC 5424**-compliant messages. If you don't, logging may not complain, but your messages will not be RFC 5424-compliant, and your syslog daemon may complain.

19 Implementing structured logging

Although most logging messages are intended for reading by humans, and thus not readily machine-parseable, there might be circumstances where you want to output messages in a structured format which *is* capable of being parsed by a program (without needing complex regular expressions to parse the log message). This is straightforward to achieve using the logging package. There are a number of ways in which this could be achieved, but the following is a simple approach which uses JSON to serialise the event in a machine-parseable manner:

```
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))
```

If the above script is run, it prints:

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

Note that the order of items might be different according to the version of Python used.

If you need more specialised processing, you can use a custom JSON encoder, as in the following complete example:

```
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(Encoder, self).default(o)
```

(下页继续)

(繼續上一頁)

```
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()
```

When the above script is run, it prints:

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

Note that the order of items might be different according to the version of Python used.

20 Customizing handlers with `dictConfig()`

There are times when you want to customize logging handlers in particular ways, and if you use `dictConfig()` you may be able to do this without subclassing. As an example, consider that you may want to set the ownership of a log file. On POSIX, this is easily done using `shutil.chown()`, but the file handlers in the stdlib don't offer built-in support. You can customize handler creation using a plain function such as:

```
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)
```

You can then specify, in a logging configuration passed to `dictConfig()`, that a logging handler be created by calling this function:

```
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.
```

(下页继续)

(繼續上一頁)

```
            '()': 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',
    },
}
```

In this example I am setting the ownership using the `pulse` user and group, just for the purposes of illustration. Putting it together into a working script, `chowntest.py`:

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

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

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

(下页继续)

(繼續上一頁)

```
    },
}

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

To run this, you will probably need to run as `root`:

```
$ 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
```

Note that this example uses Python 3.3 because that's where `shutil.chown()` makes an appearance. This approach should work with any Python version that supports `dictConfig()` - namely, Python 2.7, 3.2 or later. With pre-3.3 versions, you would need to implement the actual ownership change using e.g. `os.chown()`.

In practice, the handler-creating function may be in a utility module somewhere in your project. Instead of the line in the configuration:

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

you could use e.g.:

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

where `project.util` can be replaced with the actual name of the package where the function resides. In the above working script, using `'ext://__main__.owned_file_handler'` should work. Here, the actual callable is resolved by `dictConfig()` from the `ext://` specification.

This example hopefully also points the way to how you could implement other types of file change - e.g. setting specific POSIX permission bits - in the same way, using `os.chmod()`.

Of course, the approach could also be extended to types of handler other than a `FileHandler` - for example, one of the rotating file handlers, or a different type of handler altogether.

21 Using particular formatting styles throughout your application

In Python 3.2, the `Formatter` gained a `style` keyword parameter which, while defaulting to `%` for backward compatibility, allowed the specification of `{` or `$` to support the formatting approaches supported by `str.format()` and `string.Template`. Note that this governs the formatting of logging messages for final output to logs, and is completely orthogonal to how an individual logging message is constructed.

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 no changing this while preserving backward compatibility, since all logging calls which are out there in existing code will be using %-format strings.

There have been suggestions to associate format styles with specific loggers, but that approach also runs into backward compatibility problems because any existing code could be using a given logger name and using %-formatting.

For logging to work interoperably between any third-party libraries and your code, decisions about formatting need to be made at the level of the individual logging call. This opens up a couple of ways in which alternative formatting styles can be accommodated.

21.1 Using LogRecord factories

In Python 3.2, along with the `Formatter` changes mentioned above, the logging package gained the ability to allow users to set their own `LogRecord` subclasses, using the `setLogRecordFactory()` function. You can use this to set your own subclass of `LogRecord`, which does the Right Thing by overriding the `getMessage()` method. The base class implementation of this method is where the `msg % args` formatting happens, and where you can substitute your alternate formatting; however, you should be careful to support all formatting styles and allow %-formatting as the default, to ensure interoperability with other code. Care should also be taken to call `str(self.msg)`, just as the base implementation does.

Refer to the reference documentation on `setLogRecordFactory()` and `LogRecord` for more information.

21.2 Using custom message objects

There is another, perhaps simpler way that you can use {}- and $- formatting to construct your individual log messages. You may recall (from arbitrary-object-messages) that when logging you can use an arbitrary object as a message format string, and that the logging package will call `str()` on that object to get the actual format string. Consider the following two classes:

```
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)
```

Either of these can be used in place of a format string, to allow {}- or $-formatting to be used to build the actual "message" part which appears in the formatted log output in place of "%(message)s" or "{message}" or "$message". If you find it a little unwieldy to use the class names whenever you want to log something, you can make it more palatable if you use an alias such as `M` or _ for the message (or perhaps __, if you are using _ for localization).

Examples of this approach are given below. Firstly, formatting with `str.format()`:

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

(下页继续)

(繼續上一頁)

```
>>> p.y = 0.5
>>> print(__('Message with coordinates: ({point.x:.2f}, {point.y:.2f})', point=p))
Message with coordinates: (0.50, 0.50)
```

Secondly, formatting with `string.Template`:

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

One thing to note is that you pay no significant performance penalty with this approach: the actual formatting happens not when you make the logging call, but when (and if) the logged message is actually about to be output to a log by a handler. So the only slightly unusual thing which might trip you up is that the parentheses go around the format string and the arguments, not just the format string. That's because the __ notation is just syntax sugar for a constructor call to one of the `XXXMessage` classes shown above.

22 Configuring filters with `dictConfig()`

You *can* configure filters using `dictConfig()`, though it might not be obvious at first glance how to do it (hence this recipe). Since `Filter` is the only filter class included in the standard library, and it is unlikely to cater to many requirements (it's only there as a base class), you will typically need to define your own `Filter` subclass with an overridden `filter()` method. To do this, specify the `()` key in the configuration dictionary for the filter, specifying a callable which will be used to create the filter (a class is the most obvious, but you can provide any callable which returns a `Filter` instance). Here is a complete example:

```
import logging
import logging.config
import sys

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

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

LOGGING = {
    'version': 1,
    'filters': {
        'myfilter': {
            '()': MyFilter,
            'param': 'noshow',
        }
    },
    'handlers': {
        'console': {
```

(下页继续)

(繼續上一頁)

```
            'class': 'logging.StreamHandler',
            'filters': ['myfilter']
        }
    },
    'root': {
        'level': 'DEBUG',
        'handlers': ['console']
    },
}

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

This example shows how you can pass configuration data to the callable which constructs the instance, in the form of keyword parameters. When run, the above script will print:

```
changed: hello
```

which shows that the filter is working as configured.

A couple of extra points to note:

- If you can't refer to the callable directly in the configuration (e.g. if it lives in a different module, and you can't import it directly where the configuration dictionary is), you can use the form `ext://...` as described in logging-config-dict-externalobj. For example, you could have used the text `'ext://__main__.MyFilter'` instead of `MyFilter` in the above example.
- As well as for filters, this technique can also be used to configure custom handlers and formatters. See logging-config-dict-userdef for more information on how logging supports using user-defined objects in its configuration, and see the other cookbook recipe *Customizing handlers with dictConfig()* above.

23 Customized exception formatting

There might be times when you want to do customized exception formatting - for argument's sake, let's say you want exactly one line per logged event, even when exception information is present. You can do this with a custom formatter class, as shown in the following example:

```
import logging

class OneLineExceptionFormatter(logging.Formatter):
    def formatException(self, exc_info):
        """
        Format an exception so that it prints on a single line.
        """
        result = super(OneLineExceptionFormatter, self).formatException(exc_info)
        return repr(result)  # or format into one line however you want to

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

(下页继续)

(繼續上一頁)

```
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()
```

When run, this produces a file with exactly two lines:

```
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'|
```

While the above treatment is simplistic, it points the way to how exception information can be formatted to your liking. The `traceback` module may be helpful for more specialized needs.

24 Speaking logging messages

There might be situations when it is desirable to have logging messages rendered in an audible rather than a visible format. This is easy to do if you have text-to-speech (TTS) functionality available in your system, even if it doesn't have a Python binding. Most TTS systems have a command line program you can run, and this can be invoked from a handler using `subprocess`. It's assumed here that TTS command line programs won't expect to interact with users or take a long time to complete, and that the frequency of logged messages will be not so high as to swamp the user with messages, and that it's acceptable to have the messages spoken one at a time rather than concurrently, The example implementation below waits for one message to be spoken before the next is processed, and this might cause other handlers to be kept waiting. Here is a short example showing the approach, which assumes that the `espeak` TTS package is available:

```
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)
```

(下页继续)

(繼續上一頁)

```
        # 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())
```

When run, this script should say ”Hello” and then ”Goodbye” in a female voice.

The above approach can, of course, be adapted to other TTS systems and even other systems altogether which can process messages via external programs run from a command line.

25 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。

The example script has a simple function, `foo`, which just cycles through all the logging levels, writing to `sys.stderr` to say what level it’s about to log at, and then actually logging a message at that level. You can pass a parameter to `foo` which, if true, will log at ERROR and CRITICAL levels - otherwise, it only logs at DEBUG, INFO and WARNING levels.

The script just arranges to decorate `foo` with a decorator which will do the conditional logging that’s required. The decorator takes a logger as a parameter and attaches a memory handler for the duration of the call to the decorated function. The decorator can be additionally parameterised using a target handler, a level at which flushing should occur, and a capacity for the buffer (number of records buffered). These default to a `StreamHandler` which writes to `sys.stderr`, `logging.ERROR` and `100` respectively.

Here’s the script:

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

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

(下页继续)

(繼續上一頁)

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

运行此脚本时，应看到以下输出：

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

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

你当然可以使用传统的装饰方法:

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

26 通过配置使用 UTC (GMT) 格式化时间

有时候，你希望使用 UTC 来格式化时间，这可以使用一个类来完成，例如 'UTCFormatter'，如下所示：

```
import logging
import time

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

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

```
import logging
import logging.config
import time

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

(下页继续)

(繼續上一頁)

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

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

脚本会运行输出类似下面的内容:

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

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

27 使用上下文管理器进行选择性记录

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等级以及增加日志句柄:

```
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
```

(下页继续)

(繼續上一頁)

```
    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句柄属性，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句柄时，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 标准错误传输到/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概括，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。

28 A CLI application starter template

Here's an example which shows how you can:

- Use a logging level based on command-line arguments
- Dispatch to multiple subcommands in separate files, all logging at the same level in a consistent way
- Make use of simple, minimal configuration

Suppose we have a command-line application whose job is to stop, start or restart some services. This could be organised for the purposes of illustration as a file `app.py` that is the main script for the application, with individual commands implemented in `start.py`, `stop.py` and `restart.py`. Suppose further that we want to control the verbosity of the application via a command-line argument, defaulting to `logging.INFO`. Here's one way that `app.py` could be written:

```
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')
```

(下页继续)

(繼續上一頁)

```
    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())
```

And the `start`, `stop` and `restart` commands can be implemented in separate modules, like so for starting:

```
# 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)
```

and thus for stopping:

```
# 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)
```

and similarly for restarting:

```
# restart.py
import logging

logger = logging.getLogger(__name__)
```

(下页继续)

(繼續上一頁)

```
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)
```

If we run this application with the default log level, we get output like this:

```
$ 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.
```

The first word is the logging level, and the second word is the module or package name of the place where the event was logged.

If we change the logging level, then we can change the information sent to the log. For example, if we want more information:

```
$ 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.
```

And if we want less:

```
$ 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
```

In this case, the commands don't print anything to the console, since nothing at `WARNING` level or above is logged by them.

29 A Qt GUI for logging

A question that comes up from time to time is about how to log to a GUI application. The Qt framework is a popular cross-platform UI framework with Python bindings using PySide2 or PyQt5 libraries.

The following example shows how to log to a Qt GUI. This introduces a simple `QtHandler` class which takes a callable, which should be a slot in the main thread that does GUI updates. A worker thread is also created to show how you can log to the GUI from both the UI itself (via a button for manual logging) as well as a worker thread doing work in the background (here, just logging messages at random levels with random short delays in between).

The worker thread is implemented using Qt's `QThread` class rather than the `threading` module, as there are circumstances where one has to use `QThread`, which offers better integration with other `Qt` components.

The code should work with recent releases of either `PySide2` or `PyQt5`. You should be able to adapt the approach to earlier versions of Qt. Please refer to the comments in the code snippet for more detailed information.

```
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.
#
class QtHandler(logging.Handler):
    def __init__(self, slotfunc, *args, **kwargs):
        super(QtHandler, self).__init__(*args, **kwargs)
        self.signaller = Signaller()
```

(下页继续)

(繼續上一頁)

```
        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:
#
# * 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
```

(下页继续)

(繼續上一頁)

```
# * 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(Window, self).__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))

    def start_thread(self):
        self.worker = Worker()
        self.worker_thread = QtCore.QThread()
        self.worker.setObjectName('Worker')
```

(下页继续)

(繼續上一頁)

```
        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()
```

索引

R