Поваренная книга логирования

Выпуск 3.8.8

Гвидо ван Россум
и команда разработчиков Python

августа 12, 2021

https://Digitology.tech
Email: tweakit@bk.ru

Содержание

1 Использование логирования в нескольких модулях 2

2 Логирование из нескольких потоков 4

3 Несколько обработчиков и форматтеров 5

4 Логирование для нескольких назначений 6

5 Пример сервера конфигурации 7

6 Работа с блокирующими обработчиками 8

7 Отправка и получение событий журналирования по сети 9

8 Добавление контекстной информации к выходным данным журнала 12
- 8.1 Использование LoggerAdapters для передачи контекстной информации . . . 12
- 8.2 Использование фильтров для передачи контекстной информации . . . 13

9 Запись в один файл из нескольких процессов 15
- 9.1 Использование concurrent.futures.ProcessPoolExecutor . . . 20

10 Использование ротации файлов 20

11 Использование альтернативных стилей форматирования 21

12 Настройка `LogRecord` 24

13 Создание подкласса QueueHandler — пример ZeroMQ 25

14 Создание подкласса QueueListener — пример ZeroMQ 26

15 Пример конфигурации на основе словаря 27

16 Использование ротатора и именователя для настройки обработки ротации журналов 28

17 Более сложный пример многопроцессорной обработки **28**

18 Вставка BOM в сообщения, отправленные в SysLogHandler **33**

19 Реализация структурированного журналирования **34**

20 Настройка обработчиков с помощью `dictConfig()` **35**

21 Использование определенных стилей форматирования во всём приложении **38**
21.1 Использование фабрик LogRecord 38
21.2 Использование настраиваемых объектов сообщений 38

22 Настройка фильтров с помощью `dictConfig()` **40**

23 Настраиваемое форматирование исключений **41**

24 Озвучивание журналируемых сообщений **42**

25 Буферизация сообщений журнала и их условный вывод **43**

26 Форматирование времени с использованием UTC (GMT) через конфигурацию **45**

27 Использование диспетчера контекста для выборочного логирования **47**

28 Стартовый шаблон CLI приложения **48**

29 Графический интерфейс Qt для логирования **51**

Алфавитный указатель **57**

Автор Винай Саджип

Данный документ содержит ряд рецептов, связанных с журналированием, которые были признаны полезными.

1 Использование логирования в нескольких модулях

Несколько вызовов `logging.getLogger('someLogger')` возвращают ссылку на один и тот же объект логгера. Это верно не только для одного модуля, но и для разных модулей, если они находятся в одном процессе интерпретатора Python. Это верно для ссылок на один и тот же объект; кроме того, код приложения может определять и настраивать родительский логгер в одном модуле и создавать (но не настраивать) дочерний логгер в отдельном модуле, и все вызовы логгера дочернему элементу будут передаваться родительскому. Далее пример основного модуля:

```
import logging
import auxiliary_module

# создать логгер 'spam_application'
logger = logging.getLogger('spam_application')
logger.setLevel(logging.DEBUG)
# создать обработчик файлов, который журналирует даже отладочные сообщения
fh = logging.FileHandler('spam.log')
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
fh.setLevel(logging.DEBUG)
# создать обработчик консоли с более высоким уровнем журналирования
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# создать форматтер и добавить его в обработчики
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s -
↪%(message)s')
fh.setFormatter(formatter)
ch.setFormatter(formatter)
# добавить обработчики в логгер
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()')
```

Вспомогательный модуль:

```
import logging

# создать логгер
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"')
```

Результат выглядит так:

```
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
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

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

2 Логирование из нескольких потоков

Логирование из нескольких потоков не требует особых усилий. В следующем примере показано журналирование из основного (начального) потока и другого потока:

```
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()
```

При запуске скрипт должен напечатать что-то вроде следующего:

```
   0 Thread-1 Hi from myfunc
   3 MainThread Hello from main
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

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

Это показывает, как и следовало ожидать, перемежающийся вывод журнала . Конечно, этот подход работает для бОльшего количества потоков, чем здесь показано.

3 Несколько обработчиков и форматтеров

Логеры — простые объекты Python. Метод `addHandler()` не имеет минимальной или максимальной квоты на количество добавляемых обработчиков. Иногда приложению будет полезно записывать все сообщения всех уровней серьезности в текстовый файл, одновременно записывая ошибки или выше в консоль. Чтобы это настроить, просто настройте соответствующие обработчики. Вызовы журналирования в коде приложения останутся неизменными. Вот небольшая модификация предыдущего простого примера конфигурации на основе модулей:

```
import logging

logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)
# создать файловый обработчик, который журналирует даже отладочные сообщения
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# создать консольный обработчик с более высоким уровнем журналирования
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# создать форматтер и добавить его в обработчики
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s -
↪%(message)s')
ch.setFormatter(formatter)
fh.setFormatter(formatter)
# добавить обработчики в логгер
logger.addHandler(ch)
logger.addHandler(fh)

# код "приложения"
logger.debug('debug message')
logger.info('info message')
logger.warning('warn message')
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
logger.error('error message')
logger.critical('critical message')
```

Обратите внимание, что код «приложения» не заботится о нескольких обработчиках. Всё, что изменилось — это добавление и настройка нового обработчика с именем *fh*.

Возможность создавать новые обработчики с фильтрами более высокого или низкого уровня серьезности может быть очень полезной при написании и тестировании приложения. Вместо использования множества операторов `print` для отладки используйте `logger.debug`: в отличие от операторов print, которые вам придется удалить или закомментировать позже, операторы logger.debug могут оставаться нетронутыми в исходном коде и оставаться бездействующими до тех пор, пока они вам снова не понадобятся. В то время единственное изменение, которое должно произойти — это изменить уровень серьезности логгера и/или обработчика для отладки.

4 Логирование для нескольких назначений

Допустим, вы хотите логировать в консоль и файл с разными форматами сообщений и при разных обстоятельствах. Допустим, вы хотите логировать сообщения с уровнями DEBUG и выше в файл, а сообщения с уровнем INFO и выше — в консоль. Предположим также, что файл должен содержать временные метки, а сообщения консоли — нет. Вот как этого добиться:

```
import logging

# настроить журналирование в файл, см. предыдущий раздел для более подробной
↪информации
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')
# определить обработчик, который записывает сообщения INFO или выше в sys.
↪stderr
console = logging.StreamHandler()
console.setLevel(logging.INFO)
# установить формат, который проще для использования консоли
formatter = logging.Formatter('%(name)-12s: %(levelname)-8s %(message)s')
# сказать обработчику использовать этот формат
console.setFormatter(formatter)
# добавить обработчик в корневой логгер
logging.getLogger('').addHandler(console)

# Теперь мы можем логировать в корневой логгер или любой другой логгер.
↪Сначала корневой ...
logging.info('Jackdaws love my big sphinx of quartz.')

# Теперь определим пару других логгеров, которые могут представлять области в
↪вашем
# приложении:

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

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
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.')
```

Когда вы запустите скрипт, в консоли вы увидите:

```
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.
```

и в файле вы увидите что-то вроде:

```
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.
```

Как видите, сообщение DEBUG отображается только в файле. Остальные сообщения отправляются обоим адресатам.

В этом примере используются консоль и файловые обработчики, но вы можете использовать любое количество и любую комбинацию обработчиков по вашему выбору.

5 Пример сервера конфигурации

Далее пример модуля, использующего сервер конфигурации журналирования:

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

# прочитать исходный файл конфигурации
logging.config.fileConfig('logging.conf')

# создать и запустить прослушиватель на порту 9999
t = logging.config.listen(9999)
t.start()

logger = logging.getLogger('simpleExample')

try:
    # цикл через протоколирование вызовов, чтобы увидеть разницу
    # создание новой конфигурации до тех пор, пока не будет нажата Ctrl+C
    while True:
        logger.debug('debug message')
        logger.info('info message')
        logger.warning('warn message')
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
        logger.error('error message')
        logger.critical('critical message')
        time.sleep(5)
except KeyboardInterrupt:
    # очистить
    logging.config.stopListening()
    t.join()
```

А вот сценарий, который принимает имя файла и отправляет этот файл на сервер, которому должным образом предшествует длина в двоичной кодировке, в качестве новой конфигурации логгирования:

```
#!/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 Работа с блокирующими обработчиками

Иногда вам нужно заставить обработчики журналирования выполнять свою работу, не блокируя поток, из которого вы выполняете логирование. Это обычное дело в веб- приложениях, хотя, конечно, также происходит и в других сценариях.

Распространенным виновником, демонстрирующим вялое поведение, является `SMTPHandler`: отправка электронных писем может занять много времени по ряду причин, не зависящих от разработчика (например, плохая работа почты или сетевой инфраструктуры). Но почти любой сетевой обработчик может блокировать: даже операция `SocketHandler` может выполнять DNS-запрос под капотом, который слишком медленный (и этот запрос может находиться глубоко в коде библиотеки сокетов, ниже уровня Python и вне вашего контроля).

Одно из решений — использовать двухэтапный подход. Для первой части подключается только `QueueHandler` к тем логгерам, к которым осуществляется доступ из потоков, критичных к производительности. Они просто записывают в свою очередь, размер которой может быть достаточно большой, или инициализироваться без ограничения их размера сверху. Запись в очередь обычно принимается быстро, хотя вам, вероятно, потребуется перехватить исключение `queue.Full` в качестве меры предосторожности в вашем коде. Если вы разработчик библиотеки, у которого в коде есть потоки, критичные к производительности, обязательно задокументируйте это (вместе с предложением присоединить к вашим логгерам только `QueueHandlers`) в интересах других разработчиков, которые будут использовать ваш код.

Вторая часть решения — `QueueListener`, разработанный как аналог `QueueHandler`. `QueueListener` очень прост: он передаёт очередь и некоторые обработчики и запускает внутренний поток, который прослушивает свою очередь на предмет LogRecords, отправленных из `QueueHandlers` (или любого другого

источника `LogRecords`, если на то пошло). `LogRecords` удаляются из очереди и передаются обработчикам для обработки.

Преимущество наличия отдельного класса `QueueListener` заключается в том, что вы можете использовать один и тот же экземпляр для обслуживания нескольких `QueueHandlers`. Это более дружественно к ресурсам, чем, скажем, наличие потоковых версий существующих классов обработчиков, которые потребляли бы один поток на обработчик без особой выгоды.

Ниже приводится пример использования этих двух классов (импорт пропущен):

```
que = queue.Queue(-1)  # нет ограничений на размер
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()
# Вывод журнала будет отображать поток, который был сгенерирован
# событие (основной поток), а не внутренний
# поток, который отслеживает внутреннюю очередь.
root.warning('Look out!')
listener.stop()
```

который при запуске будет производить:

```
MainThread: Look out!
```

Изменено в версии 3.5: До Python 3.5 `QueueListener` всегда передавал каждое сообщение, полученное из очереди, каждому обработчику, которым он был инициализирован. (Это произошло потому, что предполагалось, что фильтрация уровней была полностью выполнена на другой стороне, где очередь заполнена.) Начиная с версии 3.5, это поведение можно изменить, передав ключевой аргумент `respect_handler_level=True` конструктору слушателя. Когда это будет сделано, слушатель сравнивает уровень каждого сообщения с уровнем обработчика и передаёт сообщение обработчику только в том случае, если это необходимо.

7 Отправка и получение событий журналирования по сети

Допустим, вы хотите отправлять события журналирования по сети и обрабатывать их на принимающей стороне. Простой способ сделать это — присоединить экземпляр `SocketHandler` к корневому логгеру на отправляющей стороне:

```
import logging, logging.handlers

rootLogger = logging.getLogger('')
rootLogger.setLevel(logging.DEBUG)
socketHandler = logging.handlers.SocketHandler('localhost',
                    logging.handlers.DEFAULT_TCP_LOGGING_PORT)
# не беспокойтесь о форматере, так как обработчик сокета отправляет событие
↪как
# неформатированный пикл
rootLogger.addHandler(socketHandler)
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
# Теперь мы можем логировать в корневой логгер или любой другой логгер.
↪Сначала корневой...
logging.info('Jackdaws love my big sphinx of quartz.')

# Теперь определите пару других логгеров, которые могут представлять области
↪в вашем
# приложении:

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.')
```

На принимающей стороне вы можете настроить приёмник с помощью модуля `socketserver`. Далее базовый рабочий пример:

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


class LogRecordStreamHandler(socketserver.StreamRequestHandler):
    """Обработчик для запроса потокового журналирования.

    В основном запись логируется с использованием любой локальной политики
    логирования.
    """

    def handle(self):
        """
        Обработка нескольких запросов - каждый из которых должен иметь 4-
↪байтную длину,
        после чего следует запись LogRecord в формате пикл. Регистрирует
↪запись в
        соответствии с локальной политикой.
        """
        while True:
            chunk = self.connection.recv(4)
            if len(chunk) < 4:
                break
            slen = struct.unpack('>L', chunk)[0]
            chunk = self.connection.recv(slen)
            while len(chunk) < slen:
                chunk = chunk + self.connection.recv(slen - len(chunk))
            obj = self.unPickle(chunk)
            record = logging.makeLogRecord(obj)
            self.handleLogRecord(record)
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

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

    def handleLogRecord(self, record):
        # если имя определено, мы используем именованный логгер, а не тот,
        # реализуется записью.
        if self.server.logname is not None:
            name = self.server.logname
        else:
            name = record.name
        logger = logging.getLogger(name)
        # Нотабене. КАЖДАЯ запись записывается в журнал. Это потому что
↪Logger.handle
        # обычно вызывается ПОСЛЕ фильтрацией на уровне журнала. Если нужно
        # выполнить фильтрацию, сделайте это на стороне клиента, чтобы
↪сохранить расходы
        # циклы и пропускной способности сети!
        logger.handle(record)

class LogRecordSocketReceiver(socketserver.ThreadingTCPServer):
    """
    Простой основанный на сокетах приёмник TCP логирования, подходящий для
↪тестирования.
    """

    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...')
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
    tcpserver.serve_until_stopped()

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

Сначала запустите сервер, а затем клиент. На стороне клиента на консоли ничего не печатается; на стороне сервера вы должны увидеть что-то вроде:

```
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.
```

Обратите внимание, что в некоторых сценариях есть некоторые проблемы с безопасностью. Если это важно для вас, вы можете использовать альтернативную схему сериализации, переопределив метод `makePickle()` и реализовав там свою альтернативу, а также адаптируя приведенный выше сценарий для использования альтернативной сериализации.

8 Добавление контекстной информации к выходным данным журнала

Иногда вам нужно, чтобы вывод журнала содержал контекстную информацию в дополнение к параметрам, передаваемым в вызов логгера. Например, в сетевом приложении может быть желательно логировать в журнале информацию о клиенте (например, имя пользователя или IP-адрес удаленного клиента). Хотя для этого можно использовать *extra* параметр, не всегда удобно передавать информацию таким образом. Хотя может возникнуть соблазн создать экземпляры `Logger` для каждого соединения, это не лучшая идея, поскольку эти экземпляры не собираются сборщиком мусора. Хотя на практике это не проблема, когда количество экземпляров `Logger` зависит от уровня детализации, который вы хотите использовать при ведении журнала приложения, может быть трудно управлять, если количество экземпляров `Logger` станет фактически неограниченным.

8.1 Использование LoggerAdapters для передачи контекстной информации

Простой способ передать контекстную информацию для вывода вместе с информацией о событиях в журнале — это использовать класс `LoggerAdapter`. Этот класс выглядит как `Logger`, поэтому вы можете вызвать `debug()`, `info()`, `warning()`, `error()`, `exception()`, `critical()` и `log()`. Эти методы содержат те же сигнатуры, что и их аналоги в `Logger`, поэтому вы можете использовать два типа экземпляров как взаимозаменяемые.

Когда вы создаете экземпляр `LoggerAdapter`, вы передаёте ему экземпляр `Logger` и dict-подобный объект, который содержит вашу контекстную информацию. Когда вы вызываете один из методов журналирования в экземпляре `LoggerAdapter`, он делегирует вызов базовому экземпляру `Logger`, переданному его конструктору, и организует передачу контекстной информации в делегированном вызове. Вот отрывок из кода `LoggerAdapter`:

```
def debug(self, msg, /, *args, **kwargs):
    """
    Делегировать вызов отладки базовому журналу после добавления контекстной
    информации с сущности адаптера.
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
    """
    msg, kwargs = self.process(msg, kwargs)
    self.logger.debug(msg, *args, **kwargs)
```

Метод `process()` для `LoggerAdapter` — это то место, где контекстная информация добавляется к выходным данным журнала. Он передал аргументы сообщения и ключевые аргументы вызова logging и возвращает (потенциально) модифицированные версии этих аргументов для использования в вызове в базового логгера. Реализация этого метода по умолчанию оставляет сообщение в покое, но вставляет «extra» ключ в ключевом аргументе, значением которого является dict-подобный объект, переданный конструктору. Конечно, если вы передали «extra» ключевой аргумент при вызове адаптера, он будет автоматически перезаписан.

Преимущество использования «extra» состоит в том, что значения в объекте, подобном dict, объединяются в __dict__ экземпляра `LogRecord`, что позволяет использовать настраиваемые строки с экземплярами `Formatter`, которые знают о ключах объекта, подобного dict. Если вам нужен другой метод, например если вы хотите добавить или добавить контекстную информацию к строке сообщения, вам просто нужно создать подкласс `LoggerAdapter` и переопределить `process()`, чтобы сделать то, что вам нужно. Вот простой пример:

```
class CustomAdapter(logging.LoggerAdapter):
    """
    В этом примере адаптер ожидает, что переданный объект подобный dict
↪содержит ключ
    "connid", значение которого в скобках добавляется к лог сообщению.
    """
    def process(self, msg, kwargs):
        return '[%s] %s' % (self.extra['connid'], msg), kwargs
```

который вы можете использовать вот так:

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

Тогда все события, которые вы регистрируете в адаптере, будут иметь значение `some_conn_id`, добавленное к сообщениям журнала.

Использование объектов, отличных от dicts, для передачи контекстной информации

Вам не нужно передавать фактический dict в `LoggerAdapter` — вы можете передать экземпляр класса, который реализует `__getitem__` и `__iter__`, чтобы он выглядел как dict для логгирования. Это было бы полезно, если вы хотите динамически генерировать значения (тогда как значения в dict будут постоянными).

8.2 Использование фильтров для передачи контекстной информации

Вы также можете добавить контекстную информацию в вывод журнала, используя определяемый пользователем `Filter`. Экземплярам `Filter` разрешено изменять переданный им `LogRecords`, включая добавление дополнительных атрибутов, которые затем могут выводиться с использованием подходящей строки формата или, если необходимо, пользовательского `Formatter`.

Например, в веб-приложении обрабатываемый запрос (или, по крайней мере, его интересующие части) можно сохранить в переменной threadlocal (`threading.local`), а затем получить к нему доступ из `Filter`, чтобы добавить, скажем, информацию из запроса, например, удаленный IP-адрес и имя пользователя удаленного пользователя на `LogRecord`, используя имена атрибутов ip и user, как в примере

`LoggerAdapter` выше. В этом случае можно использовать ту же строку формата для получения вывода, аналогичного показанному выше. Пример сценария:

```
import logging
from random import choice

class ContextFilter(logging.Filter):
    """
    Фильтр, который вводит контекстную информацию в журнал.

    Вместо того, чтобы использовать фактическую контекстуальную информацию, мы
    просто используем случайные данные в этой демонстрации.
    """

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

    def filter(self, record):

        record.ip = choice(ContextFilter.IPS)
        record.user = choice(ContextFilter.USERS)
        return True

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

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

который при запуске производит что-то вроде:

```
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
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
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 Запись в один файл из нескольких процессов

Хотя журналирование является потокобезопасным и *поддерживается* Логирование в один файл из нескольких потоков в одном процессе, журналирование в один файл из *нескольких процессов не* поддерживается, поскольку нет стандартного способа сериализации доступа к одному файлу из нескольких процессов в Python. Если вам нужно логировать в один файл из нескольких процессов, один из способов сделать это — сделать так, чтобы все процессы логировали в `SocketHandler` и имели отдельный процесс, который реализует сервер сокетов, который читает из сокета и записывает в файл. (Если вы предпочитаете, вы можете выделить один поток в одном из существующих процессов для выполнения этой функции.) *Секция* документирует этот подход более подробно и включает рабочий приёмник сокетов, который можно использовать в качестве отправной точки для адаптации к собственному усмотрению приложений.

Вы также можете написать свой собственный обработчик, который использует класс `Lock` из модуля `multiprocessing` для сериализации доступа к файлу из ваших процессов. Существующие `FileHandler` и подклассы не используют `multiprocessing` в настоящее время, хотя они могут использовать это в будущем. Обратите внимание, что в настоящее время модуль `multiprocessing` не обеспечивает функциональность рабочей блокировки на всех платформах (см. https://bugs.python.org/issue3770).

В качестве альтернативы вы можете использовать `Queue` и `QueueHandler` для отправки всех событий логировании в один из процессов в многопроцессорном приложении. В следующем примере сценария показано, как это можно сделать; в этом примере отдельный процесс прослушивателя прослушивает события, отправленные другими процессами, и регистрирует их в соответствии со своей собственной конфигурацией логирования. Хотя пример демонстрирует только один способ сделать это (например, вы можете использовать поток слушателя, а не отдельный процесс слушателя — реализация будет аналогичной), он позволяет использовать совершенно разные конфигурации логирования для слушателя и других процессов в вашем приложении и могут быть использованы в качестве основы для кода, отвечающего вашим собственным требованиям:

```
# Импорт потребуется в собственном коде
import logging
import logging.handlers
import multiprocessing

# Следующие две строки импорта только для этой демонстрации
from random import choice, random
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
import time

#
# Поскольку требуется определить конфигурации логирование для прослушивателя и
# работников, функции прослушивателя и рабочего процесса принимают параметр
# конфигуратора, который является вызываемым для настройки логирование для этого
# процесса. Эти функции также передаются в очередь, которую они используют для
# связи.
#
# На практике вы можете настроить листенер, как хотите, но обратите
# внимание, что в этом простом примере листенер не применяет уровень или
# логику фильтрации к полученным записям. На практике, вероятно, потребуется
# использовать эту логику в рабочих процессах, чтобы избежать отправки событий,
# которые будут отфильтрованы между процессами.
#
# Размер повернутых файлов становится небольшим, что позволяет легко видеть
# результаты.
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)

# Цикл верхнего уровня процесса прослушивателя: дождаться событий логирования
# (LogRecords) в очереди и обработать их. Выйти, когда получено None для
# LogRecord.
def listener_process(queue, configurer):
    configurer()
    while True:
        try:
            record = queue.get()
            if record is None:  # Мы посылаем это как страж, чтобы сказать слушателю, чтобы он вышел.
                break
            logger = logging.getLogger(record.name)
            logger.handle(record)  # Не применена логика уровня или фильтра - просто сделайте это!
        except Exception:
            import sys, traceback
            print('Whoops! Problem:', file=sys.stderr)
            traceback.print_exc(file=sys.stderr)

# Массивы используются для случайного выбора в этой демонстрации

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

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

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

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

# Настройка рабочего выполняется в начале запуска рабочего процесса. Обратите
# внимание, что на Windows вы не можете полагаться на семантику fork, таким
# образом, каждый процесс будет управлять кодом настройки логирования, когда
↪он
# стартует.
def worker_configurer(queue):
    h = logging.handlers.QueueHandler(queue)  # Нужен только один обработчик
    root = logging.getLogger()
    root.addHandler(h)
    # отправить все сообщения, для демонстрации; другой уровень или логика
↪фильтра не
    # применяются.
    root.setLevel(logging.DEBUG)

# Цикл верхнего уровня рабочего процесса, который просто регистрирует десять
# событий со случайными промежуточными задержками перед завершением. Печатные
# сообщения просто, чтобы вы знали, что он что-то делает!
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)

# Вот где организуется демонстрация. Создаётся очередь, создаётся и
↪запускается
# прослушиватель. Порождается десять работников и их запуск, далее ожидание,
# пока они закончат, а затем отправляется в очередь сообщение None для
↪указания слушателю
# завершить работу.
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)
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
        worker.start()
    for w in workers:
        w.join()
    queue.put_nowait(None)
    listener.join()

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

Вариант приведенного выше сценария сохраняет журналирование в основном процессе в отдельном потоке:

```
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'
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
            }
        },
        'handlers': {
            'console': {
                'class': 'logging.StreamHandler',
                'level': 'INFO',
            },
            'file': {
                'class': 'logging.FileHandler',
                'filename': 'mplog.log',
                'mode': 'w',
                'formatter': 'detailed',
            },
            'foofile': {
                'class': 'logging.FileHandler',
                'filename': 'mplog-foo.log',
                'mode': 'w',
                'formatter': 'detailed',
            },
            'errors': {
                'class': 'logging.FileHandler',
                'filename': 'mplog-errors.log',
                'mode': 'w',
                'level': 'ERROR',
                'formatter': 'detailed',
            },
        },
        'loggers': {
            'foo': {
                'handlers': ['foofile']
            }
        },
        'root': {
            'level': 'DEBUG',
            'handlers': ['console', 'file', 'errors']
        },
    }
    workers = []
    for i in range(5):
        wp = Process(target=worker_process, name='worker %d' % (i + 1),
↪args=(q,))
        workers.append(wp)
        wp.start()
    logging.config.dictConfig(d)
    lp = threading.Thread(target=logger_thread, args=(q,))
    lp.start()
    # В этот момент основной процесс мог бы сделать какую-то собственную
↪полезную
    # работу, как только он будет сделан, он может ждать, когда воркеры
↪завершатся...
    for wp in workers:
        wp.join()
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
    # И теперь сказать также завершиться логированию в потоке
    q.put(None)
    lp.join()
```

Этот вариант показывает, как применить конфигурацию для определенных логгеров — например, у логгера `foo` есть специальный обработчик, который хранит все события подсистемы `foo` в файле `mplog-foo.log`. Будет использоваться механизмом логирования в основном процессе (даже если журналируемые события генерируются в рабочих процессах) для направления сообщений в соответствующие места назначения.

9.1 Использование concurrent.futures.ProcessPoolExecutor

Если вы хотите использовать `concurrent.futures.ProcessPoolExecutor` для запуска ваших рабочих процессов, вам нужно создать очередь несколько иначе. Вместо

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

вы должны использовать

```
queue = multiprocessing.Manager().Queue(-1)  # также работает с приведенными
↪выше примерами
```

и затем вы можете заменить создание рабочего из него:

```
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()
```

к этому (не забудьте сначала импортировать `concurrent.futures`):

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

10 Использование ротации файлов

Иногда вам нужно, чтобы файл журнала увеличился до определенного размера, затем открыть новый файл и журналировать в него. Возможно, вы захотите сохранить определенное количество этих файлов, и когда такое количество файлов будет создано, ротировать файлы так, чтобы количество файлов и размер файлов оставались фиксированными. Для этого шаблона использования пакет logging предоставляет `RotatingFileHandler`:

```
import glob
import logging
import logging.handlers

```

(продолжение на следующей странице)