



kentavr009 5 августа 2020 в 20:48

8 продвинутых возможностей модуля logging в Python, которые вы не должны пропустить

Python*, Программирование*, Отладка*

[Перевод](#)

Автор оригинала: Xiaoxu Gao

Понимайте свою программу без ущерба для производительности

Журналирование — это очень важная часть разработки ПО. Оно помогает разработчикам лучше понимать выполнение программы и судить о дефектах и непредвиденных сбоях. Журнальное сообщение может хранить информацию наподобие текущего статуса программы или того, в каком месте она выполняется. Если происходит ошибка, то разработчики могут быстро найти строку кода, которая вызвала проблему, и действовать с учетом этого.

Python предоставляет довольно мощный и гибкий встроенный модуль logging со множеством возможностей. В этой статье я хочу поделиться восемью продвинутыми возможностями, которые будут полезны при разработке ПО.

Основы модуля logging

Прежде чем приступить к рассмотрению продвинутых возможностей, давайте убедимся, что у нас есть базовое понимание модуля **logging**.

Логгер

Экземпляр, который мы создаем для генерации записей, называют логгером. Он инстанцируется через **logger = logging.getLogger(name)**. Лучшая практика — это использовать **name** в качестве имени логгера. **name** включает в себя имя пакета и имя модуля. Имя будет появляться в журнальном сообщении, что поможет разработчикам быстро находить, где оно было сгенерировано.

Форматировщик и обработчик

У любого логгера есть несколько конфигураций, которые могут быть модифицированы. Более продвинутые конфигурации мы обсудим позже, а наиболее ходовые — это **форматировщик** (прим. пер.: formatter) и **обработчик** (прим. пер.: handler).

Форматировщик устанавливает структуру журнального сообщения. Каждое журнальное сообщение — это объект класса **LogRecord** с **несколькими атрибутами** (имя модуля — один из них). Когда мы определяем форматировщик, мы можем решить, как должно выглядеть журнальное сообщение вместе с этими атрибутами и, возможно, с атрибутами, созданными нами. Стандартный форматировщик выглядит так:

```
серьезность ошибки:имя логгера:сообщение
# например: WARNING:root:Программа стартует!
```

Кастомизированный форматировщик может выглядеть так:

```
"%(asctime)s - [(levelname)s] - %(name)s - (%(filename)s).%(funcName)s(%(lineno)d) - %(mes
# 2020-07-26 23:37:15,374 - [INFO] - __main__ - (main.py).main(18) - Программа стартует!
```

Обработчик задает целевое местонахождение журнальных сообщений. Журнальное сообщение может быть отправлено в более чем одно место. Собственно говоря, модуль logging предоставляет довольно много стандартных обработчиков. Самые популярные — FileHandler, который отправляет записи в файл, и StreamHandler, который отправляет записи в потоки, такие как sys.stderr или sys.stdout. Экземпляр логгера поддерживает ноль или более обработчиков. Если никакие обработчики не определены, тогда он будет отправлять записи в sys.stderr. Если определен более чем один обработчик, тогда целевое местонахождение журнального сообщения зависит от его уровня и от уровня обработчика.

Например, у меня есть `FileHandler` с уровнем `WARNING` (прим. пер.: предупреждение) и `StreamHandler` с уровнем `INFO` (прим. пер.: информация). Если я напишу в журнал сообщение об ошибке, то оно будет отправлено как в `sys.stdout`, так и в файл журнала.

Например:

В этом примере мы создали `main.py`, `package1.py`, и `app_logger.py`. Модуль `app_logger.py` содержит функцию `get_logger`, которая возвращает экземпляр логгера. Экземпляр логгера включает в себя кастомный форматировщик и два обработчика: `StreamHandler` с уровнем `INFO` и `FileHandler` с уровнем `WARNING`. Важно установить базовый уровень в `INFO` или `DEBUG` (**уровень журналирования по умолчанию — `WARNING`**), в противном случае любые записи журнала по уровню ниже, чем `WARNING`, будут отфильтрованы. И `main.py`, и `package1.py`, используют `get_logger`, чтобы создавать свои собственные логгеры.

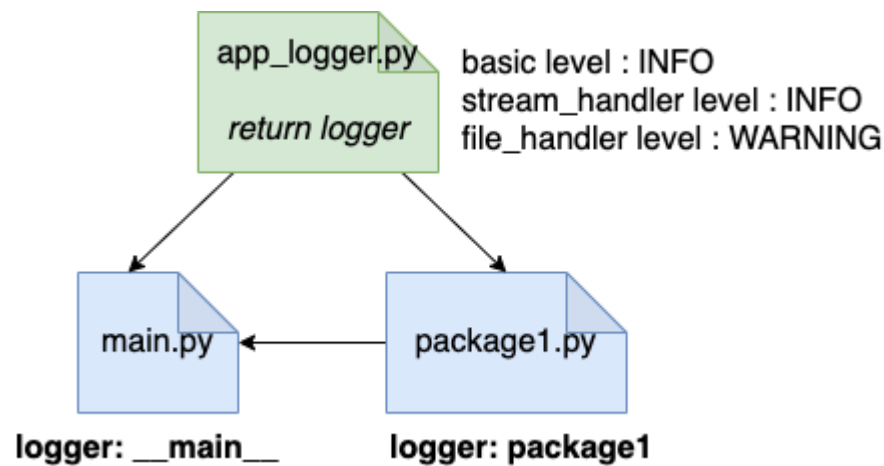


Диаграмма Xiaoxu Gao

```
# app_logger.py
import logging

_log_format = f"%(asctime)s - [% (levelname)s] - %(name)s - (%(filename)s).% (funcName)s(%(lin

def get_file_handler():
    file_handler = logging.FileHandler("x.log")
    file_handler.setLevel(logging.WARNING)
    file_handler.setFormatter(logging.Formatter(_log_format))
    return file_handler

def get_stream_handler():
    stream_handler = logging.StreamHandler()
    stream_handler.setLevel(logging.INFO)
    stream_handler.setFormatter(logging.Formatter(_log_format))
    return stream_handler

def get_logger(name):
    logger = logging.getLogger(name)
    logger.setLevel(logging.INFO)
```

```

logger.addHandler(get_file_handler())
logger.addHandler(get_stream_handler())
return logger

# package1.py
import app_logger

logger = app_logger.get_logger(__name__)

def process(msg):
    logger.info("Перед процессом")
    print(msg)
    logger.info("После процесса")

# main.py
import package1
import app_logger

logger = app_logger.get_logger(__name__)

def main():
    logger.info("Программа стартует")
    package1.process(msg="сообщение")
    logger.warning("Это должно появиться как в консоли, так и в файле журнала")
    logger.info("Программа завершила работу")

if __name__ == "__main__":
    main()

# 2020-07-25 21:06:06,657 - [INFO] - __main__ - (main.py).main(8) - Программа стартует
# 2020-07-25 21:06:06,658 - [INFO] - package1 - (package1.py).process(7) - Перед процессом
# сообщение
# 2020-07-25 21:06:06,658 - [INFO] - package1 - (package1.py).process(9) - После процесса
# 2020-07-25 21:06:06,658 - [WARNING] - __main__ - (main.py).main(10) - Это должно появиться
# 2020-07-25 21:06:06,658 - [INFO] - __main__ - (main.py).main(11) - Программа завершила раб

# cat x.log
# 2020-07-25 21:06:06,658 - [WARNING] - __main__ - (main.py).main(10) - Это должно появиться

```

basic-logging.py

Записи с уровнем INFO отправляются как в консольный вывод (sys.stdout), так и в файл журнала, а записи с уровнем WARNING пишутся только в файл журнала. Если вы можете полностью понять, что и почему происходит в этом примере, то мы готовы приступить к более продвинутым возможностям.

1. Создавайте заданные пользователем атрибуты объектов класса LogRecord, используя класс LoggerAdapter

Как я упоминал ранее, у LogRecord есть несколько **атрибутов**. Разработчики могут выбрать наиболее важные атрибуты и использовать в форматировщике. Помимо того, модуль logging также предоставляет возможность добавить в LogRecord определенные пользователем атрибуты. Один из способов сделать это — использовать **LoggerAdapter**. Когда вы создаете адаптер, вы передаете ему экземпляр логгера и свои атрибуты (в словаре). Этот класс предоставляет тот же интерфейс, что и класс **Logger**, поэтому вы все еще можете вызывать методы наподобие **logger.info**.

Новый атрибут с фиксированным значением

Если вы хотите иметь что-то вроде атрибута с фиксированным значением в журнальном сообщении, например имя приложения, то вы можете использовать стандартный класс **LoggerAdapter** и получать значение атрибута при создании логгера (прим. пер.: вероятно, получать откуда-либо, чтобы затем передать конструктору). Не забывайте добавлять этот атрибут в форматировщик. Местоположение атрибута вы можете выбрать по своему усмотрению. В следующем коде я добавляю атрибут **app**, значение которого определяется, когда я создаю логгер.

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - [%(levelname)s] - %(app)s - %(name)s - (%(filename)s).%(funcName)s
)

logger = logging.getLogger(__name__)
logger = logging.LoggerAdapter(logger, {"app": "тестовое приложение"})
logger.info("Программа стартует")
logger.info("Программа завершила работу")

# 2020-07-25 21:36:20,709 - [INFO] - тестовое приложение - __main__ - (main.py).main(8) - Пр
# 2020-07-25 21:36:20,709 - [INFO] - тестовое приложение - __main__ - (main.py).main(11) - П
```

logging_adapter_fixed_value.py

Новый атрибут с динамическим значением

В других ситуациях вам, возможно, понадобятся динамические атрибуты, например что-то вроде динамического идентификатора. В таком случае вы можете расширить базовый класс **LoggerAdapter** и создать свой собственный. Метод **process()** — то место, где дополнительные атрибуты добавляются к журнальному сообщению. В коде ниже я добавляю динамический атрибут

id, который может быть разным в каждом журнальном сообщении. В этом случае вам не нужно добавлять атрибут в форматировщик.

```
import logging

class CustomAdapter(logging.LoggerAdapter):
    def process(self, msg, kwargs):
        my_context = kwargs.pop('id', self.extra['id'])
        return '%s] %s' % (my_context, msg), kwargs

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - [%(levelname)s] - %(name)s - (%(filename)s).%(funcName)s(%(lineno)
)

logger = logging.getLogger(__name__)
logger = CustomAdapter(logger, {"id": None})

logger.info('ID предоставлен', id='1234')
logger.info('ID предоставлен', id='5678')
logger.info('Отсутствует информация об ID')

# 2020-07-25 22:12:06,715 - [INFO] - __main__ - (main.py).<module>(38) - [1234] ID предостав
# 2020-07-25 22:21:31,568 - [INFO] - __main__ - (main.py).<module>(39) - [5678] ID предостав
# 2020-07-25 22:12:06,715 - [INFO] - __main__ - (main.py).<module>(39) - [None] Отсутствует
```

logging_adapter_dynamic_value.py

2. Создавайте определенные пользователем атрибуты объектов класса LogRecord, используя класс Filter

Другой способ добавления определенных пользователем атрибутов — использование кастомного Filter. Фильтры предоставляют дополнительную логику для определения того, какие журнальные сообщения выводить. Это шаг после проверки базового уровня журналирования, но до передачи журнального сообщения обработчикам. В дополнение к определению, должно ли журнальное сообщение двигаться дальше, мы также можем вставить новые атрибуты в методе **filter()**.

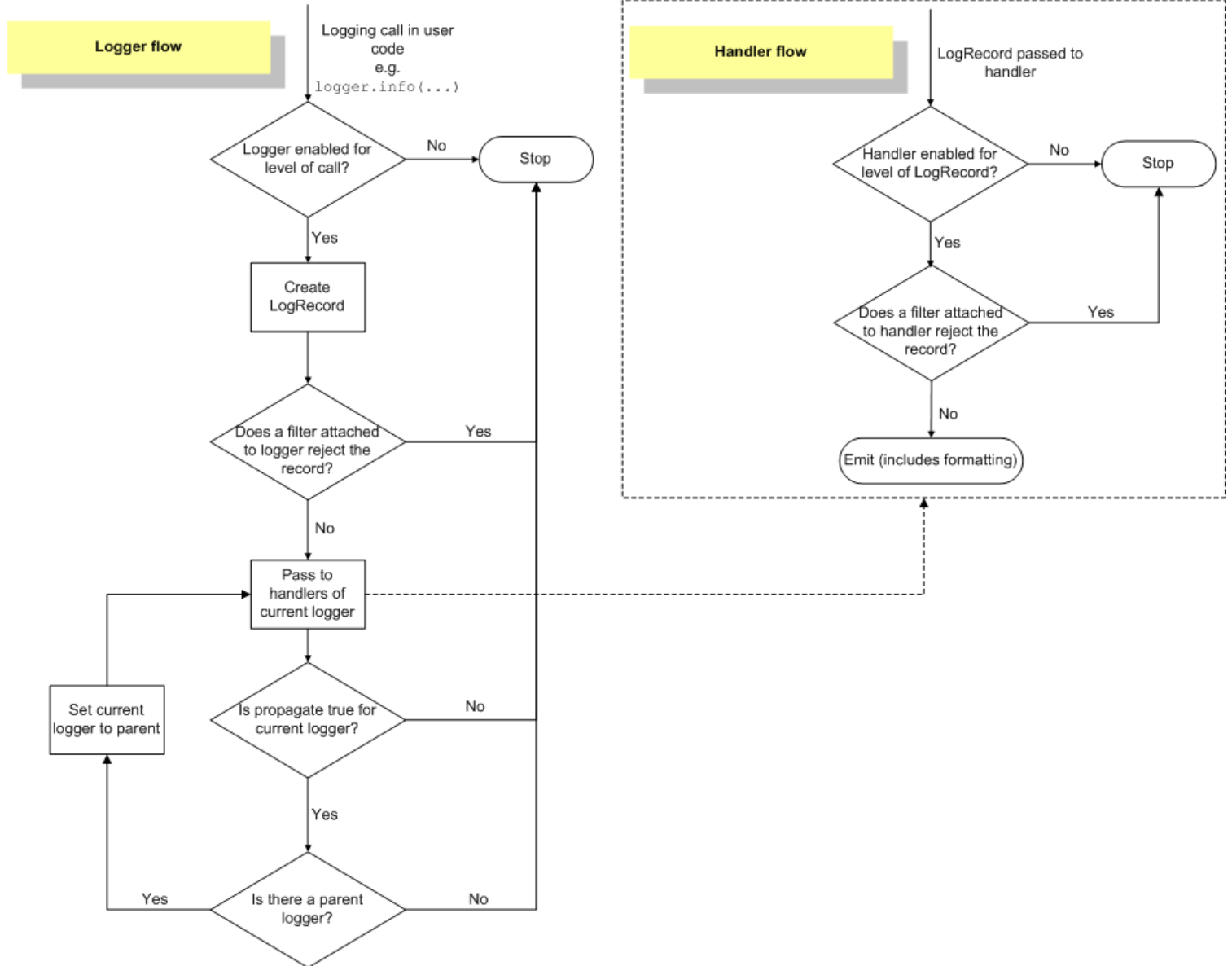


Диаграмма из официальной документации Python

В этом примере мы добавляем новый атрибут **color** (прим. пер.: цвет) в методе **filter()**, значение которого определяется на основе имени уровня в журнальном сообщении. В этом случае имя атрибута снова должно быть добавлено в форматировщик.

```

import logging

class CustomFilter(logging.Filter):

    COLOR = {
        "DEBUG": "GREEN",
        "INFO": "GREEN",
        "WARNING": "YELLOW",
        "ERROR": "RED",
        "CRITICAL": "RED",
    }

    def filter(self, record):
        record.color = CustomFilter.COLOR[record.levelname]
        return True
  
```

```

logging.basicConfig(
    level=logging.DEBUG,
    format="%(asctime)s - [%(levelname)s] - [%(color)s] - %(name)s - (%(filename)s).%(funcNa
)

logger = logging.getLogger(__name__)
logger.addFilter(CustomFilter())

logger.debug("сообщение для отладки, цвет – зеленый")
logger.info("информационное сообщение, цвет – зеленый")
logger.warning("предупреждающее сообщение, цвет – желтый")
logger.error("сообщение об ошибке, цвет – красный")
logger.critical("сообщение о критической ошибке, цвет – красный")

# 2020-07-25 22:45:17,178 - [DEBUG] - [GREEN] - __main__ - (main.py).<module>(52) - сообщени
# 2020-07-25 22:45:17,179 - [INFO] - [GREEN] - __main__ - (main.py).<module>(53) - информаци
# 2020-07-25 22:45:17,179 - [WARNING] - [YELLOW] - __main__ - (main.py).<module>(54) - преду
# 2020-07-25 22:45:17,179 - [ERROR] - [RED] - __main__ - (main.py).<module>(55) - сообщение
# 2020-07-25 22:45:17,179 - [CRITICAL] - [RED] - __main__ - (main.py).<module>(56) - сообщен

```

logging_filter_dynamic_attributes.py

3. Многопоточность с модулем logging

Модуль logging на самом деле реализован потокобезопасным способом, поэтому нам не нужны дополнительные усилия. Код ниже показывает, что MainThread и WorkThread разделяют один и тот же экземпляр логгера без проблемы состояния гонки. Также есть встроенный атрибут threadName для форматировщика.

```

import logging
import threading

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - [%(levelname)s] - [%(threadName)s] - %(name)s - (%(filename)s).%(f
)
logger = logging.getLogger(__name__)

def worker():
    for i in range(5):
        logger.info(f"журнальное сообщение {i} из рабочего потока")

thread = threading.Thread(target=worker)
thread.start()

for i in range(5):

```



```
logger.info(f"журнальное сообщение {i} из главного потока")
```

```
thread.join()
```

```
# 2020-07-26 15:33:21,078 - [INFO] - [Thread-1] - __main__ - (main.py).worker(62) - 0-ое жур
# 2020-07-26 15:33:21,078 - [INFO] - [Thread-1] - __main__ - (main.py).worker(62) - 1-ое жур
# 2020-07-26 15:33:21,078 - [INFO] - [Thread-1] - __main__ - (main.py).worker(62) - 2-ое жур
# 2020-07-26 15:33:21,078 - [INFO] - [Thread-1] - __main__ - (main.py).worker(62) - 3-ое жур
# 2020-07-26 15:33:21,078 - [INFO] - [Thread-1] - __main__ - (main.py).worker(62) - 4-ое жур
# 2020-07-26 15:33:21,078 - [INFO] - [MainThread] - __main__ - (main.py).<module>(69) - 0-ое
# 2020-07-26 15:33:21,078 - [INFO] - [MainThread] - __main__ - (main.py).<module>(69) - 1-ое
# 2020-07-26 15:33:21,078 - [INFO] - [MainThread] - __main__ - (main.py).<module>(69) - 2-ое
# 2020-07-26 15:33:21,078 - [INFO] - [MainThread] - __main__ - (main.py).<module>(69) - 3-ое
# 2020-07-26 15:33:21,078 - [INFO] - [MainThread] - __main__ - (main.py).<module>(69) - 4-ое
```

logging_multi_threading.py

Под капотом модуль logging использует **threading.RLock()** практически везде. Отличия между **RLock** от **Lock**:

1. **Lock** может быть получен только один раз и больше не может быть получен до тех пор, пока он не будет освобожден. С другой стороны, **RLock** может быть получен неоднократно до своего освобождения, но он должен быть освобожден столько же раз.
2. **Lock** может быть освобожден любым потоком, а **RLock** — только тем потоком, который его удерживает.

Любой обработчик, который наследуется от класса **Handler**, обладает методом **handle()**, предназначенным для генерации записей. Ниже представлен блок кода метода **Handler.handle()**. Как видите, обработчик получит и освободит блокировку до и после генерации записи соответственно. Метод **emit()** может быть реализован по-разному в разных обработчиках.

```
def handle(self, record):
    """
    При определенных условиях генерирует журнальную запись.
    Генерация зависит от фильтров, которые могут быть добавлены в обработчик.
    Оборачивает настоящую генерацию записи получением/освобождением
    блокировки потока ввода-вывода. Возвращает значение, свидетельствующее о том, пропустил
    генерацию.
    """
    rv = self.filter(record)
    if rv:
```

```
self.acquire()
try:
    self.emit(record)
finally:
    self.release()

return rv
```

logging_handle.py

4. Многопроцессная обработка с модулем logging — QueueHandler

Несмотря на то, что модуль logging потокобезопасен, он не процессобезопасен. Если вы хотите, чтобы несколько процессов вели запись в один и тот же файл журнала, то вы должны вручную позаботиться о доступе к вашему файлу. В соответствии с [учебником по logging](#), есть несколько вариантов.

QueueHandler + «процесс-потребитель»

Один из вариантов — использование **QueueHandler**. Идея заключается в том, чтобы создать экземпляр класса **multiprocessing.Queue** и поделить его между любым количеством процессов. В примере ниже у нас есть 2 «процесса-производителя», которые отправляют записи журнала в очередь и «процесс-потребитель», читающий записи из очереди и пишущий их в файл журнала.

У записей журнала в очереди будут, вероятно, разные уровни, так что в функции **log_processor** мы используем **logger.log(record.levelno, record.msg)** вместо **logger.info()** или **logger.warning()**. В конце (прим. пер.: функции main) мы отправляем сигнал, чтобы позволить процессу **log_processor** остановиться. Деление экземпляра очереди между множеством процессов или потоков — не новшество, но модуль logging как бы помогает нам справиться с этой ситуацией.

```
import os
from logging.handlers import QueueHandler

formatter = "%(asctime)s - [%(levelname)s] - %(name)s - (%(filename)s).%(funcName)s(%(lineno)s)\n"
logging.basicConfig(level=logging.INFO, format=formatter)

def log_processor(queue):
    logger = logging.getLogger(__name__)
    file_handler = logging.FileHandler("a.log")
    file_handler.setFormatter(logging.Formatter(formatter))
    logger.addHandler(file_handler)

    while True:
        try:
            record = queue.get()
            if record is None:
```

```

        break
    logger.log(record.levelno, record.msg)
except Exception as e:
    pass

def log_producer(queue):
    pid = os.getpid()
    logger = logging.getLogger(__name__)
    queue_handler = QueueHandler(queue)
    logger.addHandler(QueueHandler(queue))

    logger.info(f"уровень INFO - производитель {pid}")
    logger.warning(f"уровень WARNING - производитель {pid}")

def main():
    queue = multiprocessing.Queue(-1)
    listener = multiprocessing.Process(target=log_processor, args=(queue,))
    listener.start()
    workers = []
    for i in range(2):
        worker = multiprocessing.Process(target=log_producer, args=(queue,))
        workers.append(worker)
        worker.start()
    for w in workers:
        w.join()
    queue.put_nowait(None)
    listener.join()

if __name__ == '__main__':
    main()

# >> cat a.log
# 2020-07-26 18:38:10,525 - [INFO] - __mp_main__ - (main.py).log_processor(118) - уровень IN
# 2020-07-26 18:38:10,525 - [WARNING] - __mp_main__ - (main.py).log_processor(118) - уровень
# 2020-07-26 18:38:10,530 - [INFO] - __mp_main__ - (main.py).log_processor(118) - уровень IN
# 2020-07-26 18:38:10,530 - [WARNING] - __mp_main__ - (main.py).log_processor(118) - уровень

```

logging_queue_handler.py

QueueHandler + QueueListener

В модуле **logging.handlers** есть особый класс с именем **QueueListener**. Этот класс создает экземпляр слушателя с очередью журнальных сообщений и списком обработчиков для обработки записей журнала. **QueueListener** может заменить процесс, который мы создали в предыдущем примере, поместив его в переменную **listener**. При этом будет использовано меньше кода.

```

def log_producer(queue):
    pid = os.getpid()
    logger = logging.getLogger(__name__)
    logger.addHandler(QueueHandler(queue))

    logger.info(f"уровень INFO - производитель {pid}")
    logger.warning(f"уровень WARNING - производитель {pid}")

def main():
    queue = multiprocessing.Queue(-1)
    file_handler = logging.FileHandler("b.log")
    file_handler.setFormatter(logging.Formatter(formatter))
    queue_listener = QueueListener(queue, file_handler)
    queue_listener.start()

    workers = []
    for i in range(2):
        worker = multiprocessing.Process(target=log_producer, args=(queue,))
        workers.append(worker)
        worker.start()
    for w in workers:
        w.join()
    queue_listener.stop()

if __name__ == '__main__':
    main()

# >> cat b.log
# 2020-07-26 20:15:58,656 - [INFO] - __mp_main__ - (main.py).log_producer(130) - уровень INF
# 2020-07-26 20:15:58,656 - [WARNING] - __mp_main__ - (main.py).log_producer(131) - уровень
# 2020-07-26 20:15:58,662 - [INFO] - __mp_main__ - (main.py).log_producer(130) - уровень INF
# 2020-07-26 20:15:58,662 - [WARNING] - __mp_main__ - (main.py).log_producer(131) - уровень

```

logging_queue_listener.py

SocketHandler

Другое решение, предлагаемое учебником, — отправлять записи из нескольких процессов в `SocketHandler` и иметь отдельный процесс, который реализует сокет-сервер, читающий записи и отправляющий их в место назначения. В этих источниках есть довольно подробная реализация.

Все эти решения, в основном, следуют одному принципу: отправлять записи из разных процессов в централизованное место — либо в очередь, либо на удаленный сервер. Получатель на другой стороне ответственен за их запись в места назначения.

5. По умолчанию не генерируйте какие-либо журнальные записи библиотеки — `NullHandler`

На данный момент мы упомянули несколько обработчиков, реализованных модулем `logging`. Другой полезный встроенный обработчик — **`NullHandler`**. В реализации **`NullHandler`** практически ничего нет. Тем не менее, он помогает разработчикам отделить библиотечные записи журнала от записей приложения.

Ниже приведена реализация обработчика **`NullHandler`**.

```
class NullHandler(Handler):
    """
    Этот обработчик ничего не делает. Предполагается использовать его, чтобы избежать
    разового предупреждения "Для логгера XXX не найдено никаких обработчиков". Это
    важно для кода библиотеки, который может содержать код для журналирования событий. Если
    библиотеки не настроил конфигурацию журналирования, то может быть
    создано разовое предупреждение; чтобы избежать этого, разработчику библиотеку нужно прос
    NullHandler и добавить его в логгер верхнего уровня модуля библиотеки или
    пакета.
    """

    def handle(self, record):
        """Заглушка."""

    def emit(self, record):
        """Заглушка."""

    def createLock(self):
        self.lock = None
```

`logging_nullhandler.py`

Почему нам нужно отделять записи библиотеки от записей приложения?

По словам автора модуля `logging`, Vinay Sajip:

Сторонняя библиотека, которая использует **`logging`**, по умолчанию не должна выбрасывать вывод журналирования, так как он может быть не нужен разработчику/пользователю приложения, которое использует библиотеку.

Наилучшая практика — по умолчанию не генерировать библиотечные записи журнала и давать пользователю библиотеки возможность решить, хочет ли он получать и обрабатывать эти записи в приложении.

В роли разработчика библиотеки нам нужна только одна строка кода внутри **init.py**, чтобы добавить **NullHandler**. Во вложенных пакетах и модулях логгеры остаются прежними. Когда мы устанавливаем этот пакет в наше приложение через **pip install**, мы по умолчанию не увидим библиотечные записи журнала.

```
# package/
# |
# └─ __init__.py
# └─ module1.py

# __init__.py
import logging
logging.getLogger(__name__).addHandler(logging.NullHandler())

# module1.py
logger = logging.getLogger(__name__)
logger.info("это информационное журнальное сообщение")
```

logging_nullhandler_example.py

Чтобы сделать эти записи видимыми, нужно добавить обработчики в логгер библиотеки в своем приложении.

```
# ваше приложение
logging.getLogger("package").addHandler(logging.StreamHandler())
```

Если библиотека не использует **NullHandler**, но вы хотите отключить записи из библиотеки, то можете установить **logging.getLogger("package").propagate = False**. Если **propagate** установлен в **False**, то записи журнала не будут передаваться обработчикам.

6. Делайте ротацию своих файлов журнала — RotatingFileHandler, TimedRotatingFileHandler

RotatingFileHandler поддерживает ротацию файлов журнала, основанную на максимальном размере файла. Здесь должны быть определены два параметра: **maxBytes** и **backupCount**. Параметр **maxBytes** сообщает обработчику, когда делать ротацию журнала. Параметр **backupCount** — количество файлов журнала. У каждого «продолженного» файла журнала есть суффикс «.1», «.2» в конце имени файла. Если текущее журнальное сообщение вот-вот позволит файлу журнала превысить максимальный размер, то обработчик закроет текущий файл и откроет следующий.

Вот этот пример очень похож на пример из учебника. Должно получиться 6 файлов журнала.

```

import logging
from logging.handlers import RotatingFileHandler

def create_rotating_log(path):
    logger = logging.getLogger(__name__)
    logger.setLevel(logging.INFO)
    handler = RotatingFileHandler(path, maxBytes=20, backupCount=5)
    logger.addHandler(handler)

    for i in range(100):
        logger.info("Это тестовая строка-запись в журнале %s" % i)

if __name__ == "__main__":
    log_file = "test.log"
    create_rotating_log(log_file)

```

logging_file_rotation.py

Другой обработчик для ротации файлов — **TimeRotatingFileHandler**, который позволяет разработчикам создавать ротационные журналы, основываясь на истекшем времени. Условия времени включают: секунду, минуту, час, день, день недели (0=Понедельник) и полночь (журнал продлевается в полночь).

В следующем примере мы делаем ротацию файла журнала каждую секунду с пятью резервными файлами. В каждом резервном файле есть временная метка в качестве суффикса.

```

import logging
import time
from logging.handlers import TimedRotatingFileHandler

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)

handler = TimedRotatingFileHandler(
    "timed_test.log", when="s", interval=1, backupCount=5
)
logger.addHandler(handler)
for i in range(6):
    logger.info(i)
    time.sleep(1)

```

time_file_rotation.py

7. Исключения в процессе журналирования

Зачастую мы используем **logger.error()** или **logger.exception()** при обработке исключений. Но что если сам логгер генерирует исключение? Что случится с программой? Ну, зависит от обстоятельств.

Ошибка логгера обрабатывается, когда обработчик вызывает метод **emit()**. Это означает, что любое исключение, связанное с форматированием или записью, перехватывается обработчиком, а не поднимается. Если конкретнее, метод **handleError()** будет выводить трассировку в **stderr**, и программа продолжится. Если у вас есть кастомный обработчик, наследуемый от класса **Handler**, то вы можете реализовать свой собственный **handleError()**.

```
def emit(self, record):
    """
    Генерирует запись.
    Если форматировщик указан, он используется для форматирования записи.
    Затем запись пишется в поток с завершающим символом переноса строки. Если
    предоставлена информация об исключении, то она форматируется с использованием
    traceback.print_exception и добавляется в конец потока. Если у потока
    есть атрибут 'encoding', он используется для определения того, как делать
    вывод в поток.
    """
    try:
        msg = self.format(record)
        stream = self.stream
        # issue 35046: merged two stream.writes into one.
        stream.write(msg + self.terminator)
        self.flush()
    except RecursionError: # See issue 36272
        raise
    except Exception:
        self.handleError(record)
```

logging_emit.py

В этом примере во втором журнальном сообщении слишком много аргументов. Поэтому в консольном выводе мы получили трассировку, и выполнение программы все еще могло быть продолжено.

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - [%(levelname)s] - %(name)s - %(filename)s.%(funcName)s(%(lineno)s)
)

logger = logging.getLogger(__name__)
```



```
logger.info("Программа стартует")
logger.info("У этого сообщения %s слишком много аргументов", "msg", "other")
logger.info("Программа завершила работу")

# 2020-07-26 23:37:15,373 - [INFO] - __main__ - (main.py).main(16) - Программа стартует
# --- Logging error ---
# Traceback (most recent call last):
#   File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/logging/__init__.p
#     msg = self.format(record)
#   File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/logging/__init__.p
#     return fmt.format(record)
#   File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/logging/__init__.p
#     record.message = record.getMessage()
#   File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/logging/__init__.p
#     msg = msg % self.args
# TypeError: not all arguments converted during string formatting (прим. пер.: не все аргуме
# Call stack:
#   File "/Users/ie15qx/repo/compare_xml_files/source/main.py", line 22, in <module>
#     main()
#   File "/Users/ie15qx/repo/compare_xml_files/source/main.py", line 17, in main
#     logger.info("У этого сообщения %s слишком много аргументов", "msg", "other")
# Message: 'У этого сообщения %s слишком много аргументов'
# Arguments: ('msg', 'other')
# 2020-07-26 23:37:15,374 - [INFO] - __main__ - (main.py).main(18) - Программа завершила ра
```

logging_error_handle.py

Однако, если исключение произошло за пределами `emit()`, то оно может быть поднято, и программа остановится. Например, в коде ниже мы добавляем дополнительный атрибут `id` в `logger.info` без его обработки в `LoggerAdapter`. Эта ошибка не обработана и приводит к остановке программы.

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - [%(levelname)s] - %(app)s - %(name)s - (%(filename)s).%(funcName)s
")

logger = logging.getLogger(__name__)
logger = logging.LoggerAdapter(logger, {"app": "тестовое приложение"})
logger.info("Программа стартует", id="123")
logger.info("Программа завершила работу")

# source/main.py
# Traceback (most recent call last):
```

```
# File "/Users/ie15qx/repo/compare_xml_files/source/main.py", line 10, in <module>
#     logger.info("Программа стартует", id="123")
# File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/logging/__init__.p
#     self.log(INFO, msg, *args, **kwargs)
# File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/logging/__init__.p
#     self.logger.log(level, msg, *args, **kwargs)
# File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/logging/__init__.p
#     self._log(level, msg, args, **kwargs)
# TypeError: _log() got an unexpected keyword argument 'id' (прим. пер.: _log() получил непр
```

logging_exception_raise.py

8. Три разных способа конфигурирования своего логгера

Последний пункт, которым я хотел поделиться, — о конфигурировании своего логгера. Есть три способа конфигурирования логгера.

используйте код

Самый простой вариант — использовать код для конфигурирования своего логгера, так же, как во всех примерах, что мы видели ранее в этой статье. Но недостаток этого варианта в том, что любая модификация (прим. пер.: конфигурации) требует внесения изменений в исходном коде.

use dictConfig

Второй вариант — записывать конфигурацию в словарь и использовать **logging.config.dictConfig**, чтобы читать ее. Вы также можете сохранить словарь в JSON-файл и читать оттуда. Плюс в том, что этот файл может быть загружен как внешняя конфигурация, но он может способствовать появлению ошибок из-за своей структуры.

```
import logging
import logging.config

LOGGING_CONFIG = {
    'version': 1,
    'disable_existing_loggers': True,
    'formatters': {
        'standard': {
            'format': '%(asctime)s [%(levelname)s] %(name)s: %(message)s'
        },
    },
    'handlers': {
        'default': {
            'level': 'INFO',
            'formatter': 'standard',
```

```

        'class': 'logging.StreamHandler',
        'stream': 'ext://sys.stdout',
    },
},
'loggers': {
    '': { # root logger
        'handlers': ['default'],
        'level': 'DEBUG',
        'propagate': True,
    }
},
}
logging.config.dictConfig(LOGGING_CONFIG)

if __name__ == "__main__":
    log = logging.getLogger(__name__)
    log.info("hello world")

# 2020-07-28 21:44:09,966 [INFO] __main__: hello world

```

logging_configure_json.py

используйте fileConfig

И последний, но не менее важный, третий вариант — использовать **logging.config.fileConfig**. Конфигурация записывается в отдельный файл формата **.ini**.

```

# logging_config.ini
# [loggers]
# keys=root

# [handlers]
# keys=stream_handler

# [formatters]
# keys=formatter

# [logger_root]
# level=DEBUG
# handlers=stream_handler

# [handler_stream_handler]
# class=StreamHandler
# level=DEBUG
# formatter=formatter
# args=(sys.stderr,)

```

```
# [formatter_formatter]
# format=%(asctime)s %(name)-12s %(levelname)-8s %(message)s

import logging
from logging.config import fileConfig

fileConfig('logging_config.ini')
logger = logging.getLogger()
logger.debug('hello world')
```

logging_configure_file.py

Есть возможность обновлять конфигурацию во время выполнения программы через сервер конфигурации. Учебник показывает пример как со стороны клиента, так и со стороны сервера. Конфигурация обновляется посредством подключения через сокет, а на стороне клиента мы используем **c = logging.config.listen(PORT) c.start()**, чтобы получать самую новую конфигурацию.

Надеюсь, эти советы и приемы, связанные с модулем logging, могут помочь вам создать вокруг вашего приложения хороший фреймворк для журналирования без ущерба для производительности. Если у вас есть что-нибудь, чем можно поделиться, пожалуйста, оставьте комментарий ниже!

Теги: Python, журналирование, logging, threading, multiprocessing

Хабы: Python, Программирование, Отладка

Редакторский дайджест

Присылаем лучшие статьи раз в месяц

Электронпочта



11

Карма

0

Рейтинг

Alex Gerasimchuk @kentavr009

Marketing Specialist

 Комментарии 6

ПОХОЖИЕ ПУБЛИКАЦИИ

Несколько советов по организации Python-приложения на сервере

♦ +15

👁 28K

📖 198

💬 79 +79

14 июля 2014 в 17:34

Эффективная многопоточность в Python

♦ +23

👁 73K

📖 291

💬 12 +12

17 декабря 2013 в 14:59

Детализированный мониторинг запросов к Apache при помощи Python и Munin

♦ +2

👁 8.1K

📖 36

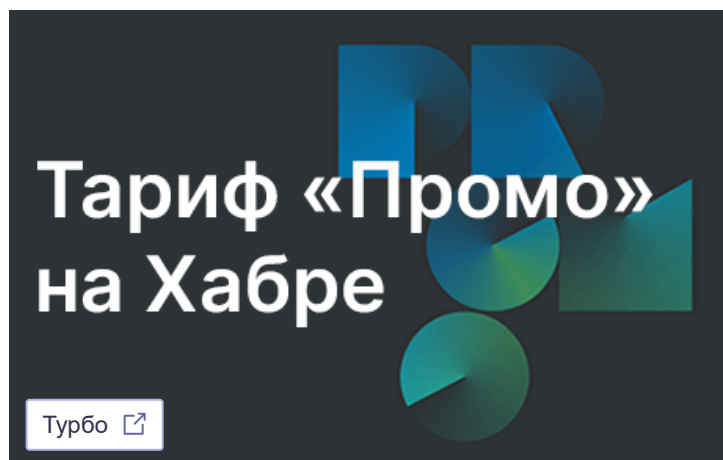
💬 0

МИНУТОЧКУ ВНИМАНИЯ

Разместить



Всем Хабром выбираем аугментации для Технотекста



Хабраблоги для маленьких, но гордых компаний

ЗАКАЗЫ

Повторить программу для определения свободного парковочного места

999 руб./за проект · 2 отклика · 43 просмотра

Переработать скрипт Pyrogram

1500 руб./за проект · 5 откликов · 45 просмотров

Создать telegram бота для беттинга

100000 руб./за проект · 12 откликов · 80 просмотров

Python. Приведение к стандартам и общим правилам оформления

5000 руб./за проект · 15 откликов · 96 просмотров

Чтение по губам с использованием нейросети

15000 руб./за проект · 7 откликов · 89 просмотров

ЛУЧШИЕ ПУБЛИКАЦИИ ЗА СУТКИ

вчера в 11:54

Как кудрявый пухляш сломал крипту: самая полная история краха биржи FTX

 +109

 24K

 24

 49 +49

вчера в 12:03

Что нам стоит UAV построить, нарисуем – будем жить. Часть 1. Про поликремний

 +57

 3.6K

 13

 18 +18

вчера в 23:51

Чини опять свою Теслу сам, тыжпрограммист

 +53

 8.7K

 27

 25 +25

вчера в 16:12

КРНР спустя 2 года

 +34

 3.8K

 16

 8 +8

вчера в 16:00

Аналоговая симуляция громкоговорителя Marshall

 +26

 2.1K

 14

 11 +11

Расчехляйте клавиатуры: открыт приём заявок на Технотекст 2022

Событие 

Ваш аккаунт

- Войти
- Регистрация

Разделы

- Публикации
- Новости
- Хабы
- Компании
- Авторы
- Песочница

Информация

- Устройство сайта
- Для авторов
- Для компаний
- Документы
- Соглашение
- Конфиденциальность

Услуги

- Корпоративный блог
- Медийная реклама
- Нативные проекты
- Образовательные программы
- Стартапам



[Настройка языка](#)

[Техническая поддержка](#)

[Вернуться на старую версию](#)

ЧИТАЮТ СЕЙЧАС



Учёные обнаружили новую комбинацию веществ, убивающую раковые клетки

 25K  14 +14

Чини опять свою Теслу сам, тыжпрограммист

 8.7K  25 +25

Почему после курсов по программированию вы никому не нужны. Как найти первую работу в IT

 2.1K  6 +6

Как случайность позволяла группе подростков обманывать пользователей Telegram в течении 3мес, с общим ущербом в \$1.5 млн

 18K  72 +72

Минцифры разрабатывает программу по возвращению уехавших IT-специалистов

 8.1K  86 +86

Как мы делаем тюнинг Spark-приложений без участия человека. Детали с примерами кода

Турбо

ИСТОРИИ

Китайские учёные планируют провести на станции «Тяньгун» репродуктивный эксперимент над обезьянами, чтобы понять, как организм адаптируется к условиям в космосе.



Космический телескоп «Хаббл» запечатлел раскидистые рукава спиральной галактики NGC 7038, находящейся на расстоянии около 220 млн световых лет от Земли.



Hover сертифицирует свою модель беспилотного аэротакси. Первые коммерческие полёты должны стартовать в 2025 году.



Хабр Карьера • Дайджест
События для зйчаров и рекрутеров в IT в ноябре 2022



Хабр Карьера • Кем работать в IT
Кем работать в IT: Специалист по ИБ



О своем опыте рассказала Кристина Жезлова, Специалист центра мониторинга информационной безопасности в IT-компании «Лощия»



Не работай через силу

Статьи о продуктивности, рабочем дне и психологии



Макак собираются размножить в космосе

«Хаббл» сделал новый снимок галактики

Hover регистрирует воздушное такси

Дайджест событий для HR в IT в ноябре

Кем работать в IT: Специалист по ИБ

Работа не в утробе жизни

РАБОТА

Data Scientist
130 вакансий

Python разработчик
137 вакансий

Django разработчик
38 вакансий

Все вакансии