
日志常用指引

发布 3.8.4rc1

Guido van Rossum
and the Python development team

七月 13, 2020

Python Software Foundation
Email: docs@python.org

Contents

1	日志基础教程	2
1.1	什么时候使用日志	2
1.2	一个简单的例子	3
1.3	记录日志到文件	3
1.4	从多个模块记录日志	4
1.5	记录变量数据	4
1.6	更改显示消息的格式	5
1.7	在消息中显示日期/时间	5
1.8	后续步骤	5
2	进阶日志教程	6
2.1	记录流程	6
2.2	记录器	7
2.3	处理程序	8
2.4	格式化程序	9
2.5	配置日志记录	9
2.6	如果没有提供配置会发生什么	12
2.7	配置库的日志记录	12
3	日志级别	13
3.1	自定义级别	13
4	有用的处理程序	14
5	记录日志中引发的异常	15
6	使用任意对象作为消息	15
7	优化	15
	索引	17

1 日志基础教程

日志是对软件执行时所发生事件的一种追踪方式。软件开发人员对他们的代码添加日志调用，借此来指示某事件的发生。一个事件通过一些包含变量数据的描述信息来描述（比如：每个事件发生时的数据都是不同的）。开发者还会区分事件的重要性，重要性也被称为 等级或 严重性

1.1 什么时候使用日志

对于简单的日志使用来说日志功能提供了一系列便利的函数。它们是 `debug()`，`info()`，`warning()`，`error()` 和 `critical()`。想要决定何时使用日志，请看下表，其中显示了对于每个通用任务集合来说最好的工具。

你想要执行的任务	此任务最好的工具
对于命令行或程序的应用，结果显示在控制台。	<code>print()</code>
在对程序的普通操作发生时提交事件报告 (比如：状态监控和错误调查)	<code>logging.info()</code> 函数 (当有诊断目的需要详细输出信息时使用 <code>logging.debug()</code> 函数)
提出一个警告信息基于一个特殊的运行时事件	<code>warnings.warn()</code> 位于代码库中，该事件是可以避免的，需要修改客户端应用以消除告警 <code>logging.warning()</code> 不需要修改客户端应用，但是该事件还是需要引起关注
对一个特殊的运行时事件报告错误	引发异常
报告错误而不引发异常 (如在长时间运行中的服务端进程的 错误处理)	<code>logging.error()</code> ， <code>logging.exception()</code> 或 <code>logging.critical()</code> 分别适用于特定的错误及应用领域

日志功能应以所追踪事件级别或严重性而定。各级别适用性如下（以严重性递增）：

级别	何时使用
DEBUG	细节信息，仅当诊断问题时适用。
INFO	确认程序按预期运行
WARNING	表明有已经或即将发生的意外（例如：磁盘空间不足）。程序仍按预期进行
ERROR	由于严重的问题，程序的某些功能已经不能正常执行
CRITICAL	严重的错误，表明程序已不能继续执行

默认的级别是“WARNING”，意味着只会追踪该级别及以上的事件，除非更改日志配置。

所追踪事件可以以不同形式处理。最简单的方式是输出到控制台。另一种常用的方式是写入磁盘文件。

1.2 一个简单的例子

一个非常简单的例子：

```
import logging
logging.warning('Watch out!') # will print a message to the console
logging.info('I told you so') # will not print anything
```

如果你在命令行中输入这些代码并运行，你将会看到：

```
WARNING:root:Watch out!
```

输出到命令行。INFO 消息并没有出现，因为默认级别是 WARNING。打印的信息包含事件的级别以及在日志调用中的对于事件的描述，例如“Watch out!”。暂时不用担心“root”部分：之后会作出解释。输出格式可按需要进行调整，格式化选项同样会在之后作出解释。

1.3 记录日志到文件

一种非常常见的情况是将日志事件记录到文件，让我们继续往下看。请确认启动新的 Python 解释器，不要在上一个环境中继续操作：

```
import logging
logging.basicConfig(filename='example.log', level=logging.DEBUG)
logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
```

现在，如果我们打开日志文件，我们应当能看到日志信息：

```
DEBUG:root:This message should go to the log file
INFO:root:So should this
WARNING:root:And this, too
```

该示例同样展示了如何设置日志追踪级别的阈值。该示例中，由于我们设置的阈值是 DEBUG，所有信息全部打印

如果你想从命令行设置日志级别，例如：

```
--log=INFO
```

并且在一些 *loglevel* 变量中你可以获得 `--log` 命令的参数，你可以使用：

```
getattr(logging, loglevel.upper())
```

通过 *level* 参数获得你将传递给 `basicConfig()` 的值。你需要对用户输入数据进行错误排查，可如下例：

```
# assuming loglevel is bound to the string value obtained from the
# command line argument. Convert to upper case to allow the user to
# specify --log=DEBUG or --log=debug
numeric_level = getattr(logging, loglevel.upper(), None)
if not isinstance(numeric_level, int):
    raise ValueError('Invalid log level: %s' % loglevel)
logging.basicConfig(level=numeric_level, ...)
```

对 `basicConfig()` 的调用应该在 `debug()`，`info()` 等的前面。因为它被设计为一次性的配置，只有第一次调用会进行操作，随后的调用不会产生有效操作。

如果多次运行上述脚本，则连续运行的消息将追加到文件 *example.log*。如果你希望每次运行重新开始，而不是记住先前运行的消息，则可以通过将上例中的调用更改为来指定 *filemode* 参数：

```
logging.basicConfig(filename='example.log', filemode='w', level=logging.DEBUG)
```

输出将与之前相同，但不再追加进日志文件，因此早期运行的消息将丢失。

1.4 从多个模块记录日志

如果你的程序包含多个模块，这里有一个如何组织日志记录的示例：

```
# myapp.py
import logging
import mylib

def main():
    logging.basicConfig(filename='myapp.log', level=logging.INFO)
    logging.info('Started')
    mylib.do_something()
    logging.info('Finished')

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

```
# mylib.py
import logging

def do_something():
    logging.info('Doing something')
```

如果你运行 *myapp.py*，你应该在 *myapp.log* 中看到：

```
INFO:root:Started
INFO:root:Doing something
INFO:root:Finished
```

这是你期待看到的。你可以使用 *mylib.py* 中的模式将此概括为多个模块。请注意，对于这种简单的使用模式，除了查看事件描述之外，你 cannot 通过查看日志文件来了解应用程序中消息的来源。如果要跟踪消息的位置，则需要参考教程级别以外的文档 - 请参阅[进阶日志教程](#)。

1.5 记录变量数据

要记录变量数据，请使用格式字符串作为事件描述消息，并将变量数据作为参数附加。例如：

```
import logging
logging.warning('%s before you %s', 'Look', 'leap!')
```

将显示：

```
WARNING:root:Look before you leap!
```

如你所见，将可变数据合并到事件描述消息中使用旧的 %-s 形式的字符串格式化。这是为了向后兼容：logging 包的出现时间早于较新的格式化选项例如 `str.format()` 和 `string.Template`。这些较新格式化选项是受支持的，但探索它们超出了本教程的范围：有关详细信息，请参阅 [formatting-styles](#)。

1.6 更改显示消息的格式

要更改用于显示消息的格式，你需要指定要使用的格式：

```
import logging
logging.basicConfig(format='%(levelname)s: %(message)s', level=logging.DEBUG)
logging.debug('This message should appear on the console')
logging.info('So should this')
logging.warning('And this, too')
```

这将输出：

```
DEBUG:This message should appear on the console
INFO:So should this
WARNING:And this, too
```

请注意，前面示例中出现的“root”已消失。对于可以出现在格式字符串中的全部内容，你可以参考以下文档 `logrecord-attributes`，但为了简单使用，你只需要 `levelname`（严重性），`message`（事件描述，包括可变数据），也许在事件发生时显示。这将在下一节中介绍。

1.7 在消息中显示日期/时间

要显示事件的日期和时间，你可以在格式字符串中放置 `%(asctime)s`

```
import logging
logging.basicConfig(format='%(asctime)s %(message)s')
logging.warning('is when this event was logged.')
```

应该打印这样的东西：

```
2010-12-12 11:41:42,612 is when this event was logged.
```

日期/时间显示的默认格式（如上所示）类似于 ISO8601 或 **RFC 3339**。如果你需要更多地控制日期/时间的格式，请为 `basicConfig` 提供 `datefmt` 参数，如下例所示：

```
import logging
logging.basicConfig(format='%(asctime)s %(message)s', datefmt='%m/%d/%Y %I:%M:%S %p')
logging.warning('is when this event was logged.')
```

这会显示如下内容：

```
12/12/2010 11:46:36 AM is when this event was logged.
```

`datefmt` 参数的格式与 `time.strftime()` 支持的格式相同。

1.8 后续步骤

基本教程到此结束。它应该足以让你启动并运行日志记录。日志包提供了更多功能，但为了充分利用它，你需要花更多的时间来阅读以下部分。如果你准备好了，可以拿一些你最喜欢的饮料然后继续。

如果你的日志记录需求很简单，那么使用上面的示例将日志记录合并到你自己的脚本中，如果你遇到问题或者不理解某些内容，请在 `comp.lang.python` Usenet 组上发布一个问题（在 <https://groups.google.com/forum/#!forum/comp.lang.python>），你应该在短时间内得到帮助。

还不够？你可以继续阅读接下来的几个部分，这些部分提供了比上面基本部分更高级或深入的教程。之后，你可以看一下 `logging-cookbook`。

2 进阶日志教程

日志库采用模块化方法，并提供几类组件：记录器、处理程序、过滤器和格式化程序。

- 记录器暴露了应用程序代码直接使用的接口。
- 处理器将日志记录（由记录器创建）发送到适当的目标。
- 过滤器提供了更精细的附加功能，用于确定要输出的日志记录。
- 格式器指定最终输出中日志记录的样式。

日志事件信息在 `LogRecord` 实例中的记录器、处理程序、过滤器和格式化程序之间传递。

通过调用 `Logger` 类（以下称为 *loggers*，记录器）的实例来执行日志记录。每个实例都有一个名称，它们在概念上以点（句点）作为分隔符排列在命名空间的层次结构中。例如，名为 `'scan'` 的记录器是记录器 `'scan.text'`，`'scan.html'` 和 `'scan.pdf'` 的父级。记录器名称可以是你想要的任何名称，并指示记录消息源自的应用程序区域。

在命名记录器时使用的一个好习惯是在每个使用日志记录的模块中使用模块级记录器，命名如下：

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

这意味着记录器名称跟踪包或模块的层次结构，并且直观地从记录器名称显示记录事件的位置。

记录器层次结构的根称为根记录器。这是函数 `debug()`、`info()`、`warning()`、`error()` 和 `critical()` 使用的记录器，它们只调用根记录器的同名方法。功能和方法具有相同的签名。根记录器的名称在记录的输出中打印为 `'root'`。

当然，可以将消息记录到不同的地方。软件包中的支持包含，用于将日志消息写入文件、HTTP GET/POST 位置、通过 SMTP 发送电子邮件、通用套接字、队列或特定于操作系统的日志记录机制（如 `syslog` 或 `Windows NT` 事件日志）。目标由 *handler* 类提供。如果你有任何内置处理程序类未满足的特殊要求，则可以创建自己的日志目标类。

默认情况下，没有为任何日志记录消息设置目标。你可以使用 `basicConfig()` 指定目标（例如控制台或文件），如教程示例中所示。如果你调用函数 `debug()`、`info()`、`warning()`、`error()` 和 `critical()`，他们将检查是否有设置目的地；如果没有设置，它们将在委托给根记录器进行实际的消息输出之前设置目标为控制台（`sys.stderr`）和默认格式的显示消息。

由 `basicConfig()` 设置的消息默认格式为：

```
severity:logger name:message
```

你可以通过使用 *format* 参数将格式字符串传递给 `basicConfig()` 来更改此设置。有关如何构造格式字符串的所有选项，请参阅 `formatter-objects`。

2.1 记录流程

记录器和处理程序中的日志事件信息流程如下图所示。



2.2 记录器

Logger 对象有三重任务。首先，它们向应用程序代码公开了几种方法，以便应用程序可以在运行时记录消息。其次，记录器对象根据严重性（默认过滤工具）或过滤器对象确定要处理的日志消息。第三，记录器对象将相关的日志消息传递给所有感兴趣的日志处理程序。

记录器对象上使用最广泛的方法分为两类：配置和消息发送。

这些是最常见的配置方法：

- `Logger.setLevel()` 指定记录器将处理的最低严重性日志消息，其中 `debug` 是最低内置严重性级别，`critical` 是最高内置严重性级别。例如，如果严重性级别为 `INFO`，则记录器将仅处理 `INFO`、`WARNING`、`ERROR` 和 `CRITICAL` 消息，并将忽略 `DEBUG` 消息。
- `Logger.addHandler()` 和 `Logger.removeHandler()` 从记录器对象中添加和删除处理程序对象。处理程序在以下内容中有更详细的介绍[处理程序](#)。
- `Logger.addFilter()` 和 `Logger.removeFilter()` 可以添加或移除记录器对象中的过滤器。`filter` 包含更多的过滤器细节。

你不需要始终在你创建的每个记录器上调用这些方法。请参阅本节的最后两段。

配置记录器对象后，以下方法将创建日志消息：

- `Logger.debug()`、`Logger.info()`、`Logger.warning()`、`Logger.error()` 和 `Logger.critical()` 都创建日志记录，包含消息和与其各自方法名称对应的级别。该消息实际上是一个格式化字符串，它可能包含标题字符串替换语法 `%s`、`%d`、`%f` 等等。其余参数是与消息中的替换字段对应的对象列表。关于 `**kwargs`，日志记录方法只关注 `exc_info` 的关键字，并用它来确定是否记录异常信息。
- `Logger.exception()` 创建与 `Logger.error()` 相似的日志信息。不同之处是，`Logger.exception()` 同时还记录当前的堆栈追踪。仅从异常处理程序调用此方法。
- `Logger.log()` 将日志级别作为显式参数。对于记录消息而言，这比使用上面列出的日志级别方便方法更加冗长，但这是自定义日志级别的方法。

`getLogger()` 返回对具有指定名称的记录器实例的引用（如果已提供），或者如果没有则返回 `root`。名称是以句点分隔的层次结构。多次调用 `getLogger()` 具有相同的名称将返回对同一记录器对象的引用。在分层列表中较低的记录器是列表中较高的记录器的子项。例如，给定一个名为 `foo` 的记录器，名称为 `foo.bar`、`foo.bar.baz` 和 `foo.bam` 的记录器都是 `foo` 子项。

记录器具有有效等级的概念。如果未在记录器上显式设置级别，则使用其父级别作为其有效级别。如果父级没有明确的级别设置，则检查其父级。依此类推，搜索所有上级元素，直到找到明确设置的级别。根记录器始终具有显式级别集（默认情况下为 `WARNING`）。在决定是否处理事件时，记录器的有效级别用于确定事件是否传递给记录器的处理程序。

子记录器将消息传播到与其上级记录器关联的处理程序。因此，不必为应用程序使用的所有记录器定义和配置处理程序。为顶级记录器配置处理程序并根据需要创建子记录器就足够了。（但是，你可以通过将记录器的 `propagate` 属性设置 `False` 来关闭传播。）

2.3 处理程序

`Handler` 对象负责将适当的日志消息（基于日志消息的严重性）分派给处理程序的指定目标。`Logger` 对象可以使用 `addHandler()` 方法向自己添加零个或多个处理程序对象。作为示例场景，应用程序可能希望将所有日志消息发送到日志文件，将错误或更高的所有日志消息发送到标准输出，以及将所有关键消息发送到一个邮件地址。此方案需要三个单独的处理程序，其中每个处理程序负责将特定严重性的消息发送到特定位置。

标准库包含很多处理程序类型（参见[有用的处理程序](#)）；教程主要使用 `StreamHandler` 和 `FileHandler`。

处理程序中很少有方法可供应用程序开发人员使用。与使用内置处理程序对象（即不创建自定义处理程序）的应用程序开发人员相关的唯一处理程序方法是以下配置方法：

- `setLevel()` 方法，就像在记录器对象中一样，指定将被分派到适当目标的最低严重性。为什么有两个 `setLevel()` 方法？记录器中设置的级别确定将传递给其处理程序的消息的严重性。每个处理程序中设置的级别确定处理程序将发送哪些消息。
- `setFormatter()` 选择一个该处理程序使用的 `Formatter` 对象。
- `addFilter()` 和 `removeFilter()` 分别在处理程序上配置和取消配置过滤器对象。

应用程序代码不应直接实例化并使用 `Handler` 的实例。相反，`Handler` 类是一个基类，它定义了所有处理程序应该具有的接口，并建立了子类可以使用（或覆盖）的一些默认行为。

2.4 格式化程序

格式化程序对象配置日志消息的最终顺序、结构和内容。与 `logging.Handler` 类不同，应用程序代码可以实例化格式化程序类，但如果应用程序需要特殊行为，则可能会对格式化程序进行子类化。构造函数有三个可选参数——消息格式字符串、日期格式字符串和样式指示符。

```
logging.Formatter.__init__(fmt=None, datefmt=None, style='%')
```

如果没有消息格式字符串，则默认使用原始消息。如果没有日期格式字符串，则默认日期格式为：

```
%Y-%m-%d %H:%M:%S
```

最后加上毫秒数。style 是 `%`、`'{'` 或 `$` 之一。如果未指定其中一个，则将使用 `'%'`。

如果 style 是 `'%'`，则消息格式字符串使用 `%(dictionary key)s` 样式字符串替换；可能的键值在 `logrecord-attributes` 中。如果样式为 `'{'`，则假定消息格式字符串与 `str.format()`（使用关键字参数）兼容，而如果样式为 `$`，则消息格式字符串应符合 `string.Template.substitute()`。

在 3.2 版更改：添加 style 形参。

以下消息格式字符串将以人类可读的格式记录时间、消息的严重性以及消息的内容，按此顺序：

```
'%(asctime)s - %(levelname)s - %(message)s'
```

格式化程序使用用户可配置的函数将记录的创建时间转换为元组。默认情况下，使用 `time.localtime()`；要为特定格式化程序实例更改此项，请将实例的 `converter` 属性设置为具有相同签名的函数 `time.localtime()` 或 `time.gmtime()`。要为所有格式化程序更改它，例如，如果你希望所有记录时间都以 GMT 显示，请在格式化程序类中设置 `converter` 属性（对于 GMT 显示，设置为 `time.gmtime`）。

2.5 配置日志记录

开发者可以通过三种方式配置日志记录：

1. 使用调用上面列出的配置方法的 Python 代码显式创建记录器、处理程序和格式化程序。
2. 创建日志配置文件并使用 `fileConfig()` 函数读取它。
3. 创建配置信息字典并将其传递给 `dictConfig()` 函数。

有关最后两个选项的参考文档，请参阅 `logging-config-api`。以下示例使用 Python 代码配置一个非常简单的记录器/一个控制台处理程序和一个简单的格式化程序：

```
import logging

# create logger
logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)

# create console handler and set level to debug
ch = logging.StreamHandler()
ch.setLevel(logging.DEBUG)

# create formatter
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

# add formatter to ch
ch.setFormatter(formatter)

# add ch to logger
```

(下页继续)

(续上页)

```
logger.addHandler(ch)

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warning('warn message')
logger.error('error message')
logger.critical('critical message')
```

从命令行运行此模块将生成以下输出：

```
$ python simple_logging_module.py
2005-03-19 15:10:26,618 - simple_example - DEBUG - debug message
2005-03-19 15:10:26,620 - simple_example - INFO - info message
2005-03-19 15:10:26,695 - simple_example - WARNING - warn message
2005-03-19 15:10:26,697 - simple_example - ERROR - error message
2005-03-19 15:10:26,773 - simple_example - CRITICAL - critical message
```

以下 Python 模块创建的记录器、处理程序和格式化程序几乎与上面列出的示例中的相同，唯一的区别是对象的名称：

```
import logging
import logging.config

logging.config.fileConfig('logging.conf')

# create logger
logger = logging.getLogger('simpleExample')

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warning('warn message')
logger.error('error message')
logger.critical('critical message')
```

这是 logging.conf 文件：

```
[loggers]
keys=root,simpleExample

[handlers]
keys=consoleHandler

[formatters]
keys=simpleFormatter

[logger_root]
level=DEBUG
handlers=consoleHandler

[logger_simpleExample]
level=DEBUG
handlers=consoleHandler
qualname=simpleExample
propagate=0
```

(下页继续)

(续上页)

```
[handler_consoleHandler]
class=StreamHandler
level=DEBUG
formatter=simpleFormatter
args=(sys.stdout,)

[formatter_simpleFormatter]
format=%(asctime)s - %(name)s - %(levelname)s - %(message)s
datefmt=
```

输出几乎与不基于配置文件的示例相同：

```
$ python simple_logging_config.py
2005-03-19 15:38:55,977 - simpleExample - DEBUG - debug message
2005-03-19 15:38:55,979 - simpleExample - INFO - info message
2005-03-19 15:38:56,054 - simpleExample - WARNING - warn message
2005-03-19 15:38:56,055 - simpleExample - ERROR - error message
2005-03-19 15:38:56,130 - simpleExample - CRITICAL - critical message
```

你可以看到配置文件方法比 Python 代码方法有一些优势，主要是配置和代码的分离以及非开发者轻松修改日志记录属性的能力。

警告： `fileConfig()` 函数接受一个默认参数 `disable_existing_loggers`，出于向后兼容的原因，默认为 `True`。这可能是你想要的，也可能不是你想要的，因为除非在配置中明确命名它们（或一个上级节点中），否则它将导致在 `fileConfig()` 调用之前存在的任何非 `root` 记录器被禁用。有关更多信息，请参阅参考文档，如果需要，请为此参数指定 `False`。

传递给 `dictConfig()` 的字典也可以用键 `disable_existing_loggers` 指定一个布尔值，如果没有在字典中明确指定，也默认被解释为 `True`。这会导致上面描述的记录器禁用行为，这可能不是你想要的——在这种情况下，明确地为键提供 `False` 值。

请注意，配置文件中引用的类名称需要相对于日志记录模块，或者可以使用常规导入机制解析的绝对值。因此，你可以使用 `WatchedFileHandler`（相对于日志记录模块）或 `mypackage.mymodule.MyHandler`（对于在 `mypackage` 包中定义的和模块 `mymodule`，其中 `mypackage` 在 Python 导入路径上可用）。

在 Python 3.2 中，引入了一种新的配置日志记录的方法，使用字典来保存配置信息。这提供了上述基于配置文件方法的功能的超集，并且是新应用程序和部署的推荐配置方法。因为 Python 字典用于保存配置信息，并且由于你可以使用不同的方式填充该字典，因此你有更多的配置选项。例如，你可以使用 JSON 格式的配置文件，或者如果你有权访问 YAML 处理功能，则可以使用 YAML 格式的文件来填充配置字典。当然，你可以在 Python 代码中构造字典，通过套接字以 `pickle` 形式接收它，或者使用对你的应用程序合理的任何方法。

以下是与上述相同配置的示例，采用 YAML 格式，用于新的基于字典的方法：

```
version: 1
formatters:
  simple:
    format: '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
handlers:
  console:
    class: logging.StreamHandler
    level: DEBUG
    formatter: simple
    stream: ext://sys.stdout
loggers:
```

(下页继续)

```

simpleExample:
    level: DEBUG
    handlers: [console]
    propagate: no
root:
    level: DEBUG
    handlers: [console]

```

有关使用字典进行日志记录的更多信息，请参阅 `logging-config-api`。

2.6 如果没有提供配置会发生什么

如果未提供日志记录配置，则可能出现需要输出日志记录事件但无法找到输出事件的处理程序的情况。在这些情况下，日志包的行为取决于 Python 版本。

对于 3.2 之前的 Python 版本，行为如下：

- 如果 `logging.raiseExceptions` 为 `False`（生产模式），则会以静默方式丢弃该事件。
- 如果 `logging.raiseExceptions` 为 `True`（开发模式），则会打印一条消息“无法找到记录器 X.Y.Z 的处理程序”。

在 Python 3.2 及更高版本中，行为如下：

- 事件使用“最后的处理程序”输出，存储在 `logging.lastResort` 中。这个内部处理程序与任何记录器都没有关联，它的作用类似于 `StreamHandler`，它将事件描述消息写入 `sys.stderr` 的当前值（因此服从任何可能的重定向影响）。没有对消息进行格式化——只打印裸事件描述消息。处理程序的级别设置为“警告”，因此将输出此级别和更高级别的所有事件。

要获得 3.2 之前的行为，`logging.lastResort` 可以设置为 `None`。

2.7 配置库的日志记录

在开发使用日志记录的库时，你应该注意记录库如何使用日志记录——例如，使用的记录器的名称。还需要考虑其日志记录配置。如果应用程序不使用日志记录，并且库代码进行日志记录调用，那么（如上一节所述）严重性为“WARNING”及更高级别的事件将打印到 `sys.stderr`。这被认为是最好的默认行为。

如果由于某种原因，你不希望没有任何日志记录配置的情况下打印这些消息，则可以将无操作处理程序附加到库的顶级记录器。这样可以避免打印消息，因为将始终为库的事件找到处理程序：它不会产生任何输出。如果库用户配置应用程序使用的日志记录，可能是配置将添加一些处理程序，如果级别已适当配置，则在库代码中进行的日志记录调用将正常地将输出发送给这些处理程序。

日志包中包含一个不做任何事情的处理程序：`NullHandler`（自 Python 3.1 起）。可以将此处理程序的实例添加到库使用的日志记录命名空间的顶级记录器中（如果你希望在没有日志记录配置的情况下阻止库的记录事件输出到 `sys.stderr`）。如果库 `foo` 的所有日志记录都是使用名称匹配 `'foo.x'`，`'foo.x.y'` 等的记录器完成的，那么代码：

```

import logging
logging.getLogger('foo').addHandler(logging.NullHandler())

```

应该有预计的效果。如果一个组织生成了许多库，则指定的记录器名称可以是“`orgname.foo`”而不仅仅是“`foo`”。

注解：强烈建议你 不要将 `NullHandler` 以外的任何处理程序添加到库的记录器中。这是因为处理程序的配置是使用你的库的应用程序开发人员的权利。应用程序开发人员了解他们的目标受众以及哪些处理程序最

适合他们的应用程序：如果你在“底层”添加处理程序，则可能会干扰他们执行单元测试和提供符合其要求的日志的能力。

3 日志级别

日志记录级别的数值在下表中给出。如果你想要定义自己的级别，并且需要它们具有相对于预定义级别的特定值，那么这些内容可能是你感兴趣的。如果你定义具有相同数值的级别，它将覆盖预定义的值；预定义的名称丢失。

级别	数值
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

级别也可以与记录器相关联，由开发人员设置或通过加载已保存的日志记录配置。在记录器上调用日志记录方法时，记录器会将其自己的级别与与方法调用关联的级别进行比较。如果记录器的级别高于方法调用的级别，则实际上不会生成任何记录消息。这是控制日志记录输出详细程度的基本机制。

记录消息被编码为 `LogRecord` 类的实例。当记录器决定实际记录事件时，从记录消息创建 `LogRecord` 实例。

记录消息通过使用 *handlers* 进行调度机制，它们是 `Handler` 类的子类的实例。处理程序负责确保记录的消息（以 `LogRecord` 的形式）最终位于特定位置（或一组位置），这对该消息的目标受众（例如最终用户、支持服务台员工、系统管理员、开发人员）。传递处理程序用于特定目标的 `LogRecord` 实例。每个记录器可以有零个、一个或多个与之关联的处理程序（通过 `Logger` 的 `addHandler()` 方法）。除了与记录器直接关联的任何处理程序之外，还调用与记录器的所有祖先相关联的所有处理程序来分派消息（除非记录器的 **propagate* 标志设置为 `false` 值，这将停止传递到上级处理程序）。

就像记录器一样，处理程序可以具有与它们相关联的级别。处理程序的级别作为过滤器，其方式与记录器级别相同。如果处理程序决定调度一个事件，则使用 `emit()` 方法将消息发送到其目标。大多数用户定义的 `Handler` 子类都需要重载 `emit()`。

3.1 自定义级别

定义你自己的级别是可能的，但不一定是必要的，因为现有级别是根据实践经验选择的。但是，如果你确信需要自定义级别，那么在执行此操作时应特别小心，如果你正在开发库，则定义自定义级别可能是一个非常糟糕的主意。这是因为如果多个库作者都定义了他们自己的自定义级别，那么使用开发人员很难控制和解释这些多个库的日志记录输出，因为给定的数值可能意味着不同的东西对于不同的库。

4 有用的处理程序

作为 Handler 基类的补充，提供了很多有用的子类：

1. `StreamHandler` 实例发送消息到流（类似文件对象）。
 2. `FileHandler` 实例将消息发送到硬盘文件。
 3. `BaseRotatingHandler` 是轮换日志文件的处理程序的基类。它并不应该直接实例化。而应该使用 `RotatingFileHandler` 或 `TimedRotatingFileHandler` 代替它。
 4. `RotatingFileHandler` 实例将消息发送到硬盘文件，支持最大日志文件大小和日志文件轮换。
 5. `TimedRotatingFileHandler` 实例将消息发送到硬盘文件，以特定的时间间隔轮换日志文件。
 6. `SocketHandler` 实例将消息发送到 **TCP/IP** 套接字。从 3.4 开始，也支持 **Unix** 域套接字。
 7. `DatagramHandler` 实例将消息发送到 **UDP** 套接字。从 3.4 开始，也支持 **Unix** 域套接字。
 8. `SMTPHandler` 实例将消息发送到指定的电子邮件地址。
 9. `SysLogHandler` 实例将消息发送到 **Unix syslog** 守护程序，可能在远程计算机上。
 10. `NTEventLogHandler` 实例将消息发送到 **Windows NT/2000/XP** 事件日志。
 11. `MemoryHandler` 实例将消息发送到内存中的缓冲区，只要满足特定条件，缓冲区就会刷新。
 12. `HTTPHandler` 实例使用 **GET** 或 **POST** 方法将消息发送到 **HTTP** 服务器。
 13. `WatchedFileHandler` 实例会监视他们要写入日志的文件。如果文件发生更改，则会关闭该文件并使用文件名重新打开。此处理程序仅在类 **Unix** 系统上有用；**Windows** 不支持依赖的基础机制。
 14. `QueueHandler` 实例将消息发送到队列，例如在 `queue` 或 `multiprocessing` 模块中实现的队列。
 15. `NullHandler` 实例对错误消息不执行任何操作。它们由想要使用日志记录的库开发人员使用，但是想要避免如果库用户没有配置日志记录，则显示“无法找到记录器 **XXX** 的消息处理器”消息的情况。有关更多信息，请参阅[配置库的日志记录](#)。
- 3.1 新版功能: `NullHandler` 类。
- 3.2 新版功能: `QueueHandler` 类。

`The NullHandler`、`StreamHandler` 和 `FileHandler` 类在核心日志包中定义。其他处理程序定义在 `logging.handlers` 中。（还有另一个子模块 `logging.config`，用于配置功能）

记录的消息被格式化为通过 `Formatter` 类的实例进行呈现。它们使用适合与 `%` 运算符一起使用的格式字符串和字典进行初始化。

对于批量格式化多个消息，可以使用 `BufferingFormatter` 的实例。除了格式字符串（应用于批处理中的每个消息）之外，还提供了标题和尾部格式字符串。

当基于记录器级别和处理程序级别的过滤不够时，可以将 `Filter` 的实例添加到 `Logger` 和 `Handler` 实例（通过它们的 `addFilter()` 方法）。在决定进一步处理消息之前，记录器和处理程序都会查询其所有过滤器以获得许可。如果任何过滤器返回 `false` 值，则不会进一步处理该消息。

基本 `Filter` 的功能允许按特定的记录器名称进行过滤。如果使用此功能，则允许通过过滤器发送到指定记录器及其子项的消息，并丢弃所有消息。

5 记录日志中引发的异常

日志包设计为忽略记录日志生产时发生的异常。这样，处理日志记录事件时发生的错误（例如日志记录错误配置、网络或其他类似错误）不会导致使用日志记录的应用程序过早终止。

`SystemExit` 和 `KeyboardInterrupt` 异常永远不会被忽略。在 `Handler` 子类的 `emit()` 方法中发生的其他异常被传递给它的 `handleError()` 方法。

`Handler` 中默认实现的 `handleError()` 检查是否设置了模块级变量 `raiseExceptions`。如果有设置，则会将跟踪打印到 `sys.stderr`。如果未设置，则忽略异常。

注解： `raiseExceptions` 默认值是 `True`。这是因为在开发期间，你通常希望收到任何发生异常的通知。建议你将 `raiseExceptions` 设置为 `False` 以供生产环境使用。

6 使用任意对象作为消息

在前面的部分和示例中，都假设记录事件时传递的消息是字符串。但是，这不是唯一的可能性。你可以将任意对象作为消息传递，并且当日志记录系统需要将其转换为字符串表示时，将调用其 `__str__()` 方法。实际上，如果你愿意，你可以完全避免计算字符串表示。例如，`SocketHandler` 通过 `pickle` 并网络发送来发出事件。

7 优化

消息参数的格式化将被推迟，直到无法避免。但是，计算传递给日志记录方法的参数也可能很消耗资源，如果记录器只是丢弃你的事件，你可能希望避免这样做。要决定做什么，可以调用 `isEnabledFor()` 方法，该方法接受一个 `level` 参数，如果记录器为该级别的调用创建了该事件，则返回 `true`。你可以写这样的代码：

```
if logger.isEnabledFor(logging.DEBUG):
    logger.debug('Message with %s, %s', expensive_func1(),
                expensive_func2())
```

因此，如果记录器的阈值设置在“DEBUG”以上，则永远不会调用 `expensive_func1()` 和 `expensive_func2()`。

注解： 在某些情况下，`isEnabledFor()` 本身可能比你想要的更消耗资源（例如，对于深度嵌套的记录器，其中显式级别仅在记录器层次结构中设置为高）。在这种情况下（或者如果你想避免在紧密循环中调用方法），你可以在本地或实例变量中将调用的结果缓存到 `isEnabledFor()`，并使用它而不是调用每次方法。只在日志记录配置在应用程序运行时动态更改（这不常见）时需重新计算这样的缓存值。

对于需要对收集的日志信息进行更精确控制的特定应用程序，还可以进行其他优化。以下是你可以执行的操作列表，以避免在你不需要的日志记录期间进行处理：

你不想收集的内容	如何避免收集它
有关调用来源的信息	将 <code>logging._srcfile</code> 设置为 <code>None</code> 。这避免了调用 <code>sys._getframe()</code> , 如果 PyPy 支持 Python 3.x , 这可能有助于加速 PyPy (无法加速使用 <code>sys._getframe()</code> 的代码) 等环境中的代码.
线程信息	将 <code>logging.logThreads</code> 置为 0 。
进程信息	将 <code>logging.logProcesses</code> 置为 0 。

另请注意, 核心日志记录模块仅包含基本处理程序。如果你不导入 `logging.handlers` 和 `logging.config` , 它们将不会占用任何内存。

参见:

模块 `logging` 日志记录模块的 API 参考。

`logging.config` 模块 日志记录模块的配置 API 。

`logging.handlers` 模块 日志记录模块附带的有用处理器。

日志操作手册

索引

非字母

`__init__()` (*logging.logging.Formatter* 方法), 9

R

RFC

RFC 3339, 5