Python 日志库 logging 的理解和实践经验

本文从 Python logging 库的基础概念入手，理解 logging 库的执行流程，以及可能忽视的一些细节。

日志级别

logging 库预置了 5 个错误级别，还有一个 NOTSET 级别，作为 logger 的默认值。

```
CRITICAL = 50
ERROR = 40
WARNING = 30
INFO = 20
DEBUG = 10
NOTSET = 0
```

logging 库也支持自定义错误级别，通过上面的源码可以看到，在不同级别的错误中间预留了 10 个数字的位置，方便我们在预置错误级别的基础上添加更细致的错误级别。

```
import logging


logging.addLevelName(31, 'SERIOUS WARNING')
logger = logging.getLogger('log')
logger.warn('warn info')
logger.log(logging.getLevelName('SERIOUS_WARNING'), 'serious warn')
```

例如添加一个 SERIOUS WARNING 类型的错误，值为 31，就可以用 log 方法输出该级别的错误。

也可以覆盖 logging 预置的错误级别，例如将 WARNING 修改为 SERIOUS WARNING。

```
logging.addLevelName(30, 'SERIOUS WARNING')
logger = logging.getLogger('log')
print(logging.getLevelName(30))  # SERIOUS WARNING
```

LogRecord、Formatter

logging 库中的每一条 log 都以 LogRecord 的形式存在，当调用 logger 打印 log 时候，都会有一条 LogRecord 被自动创建出来，LogRecord 中包含了大量的和该条日志相关的属性，也包含用户传入的 message。

属性名	格式	描述
asctime	%(asctime)s	以可读格式表示的日志创建时间
created	%(created)f	通过 time.time() 函数获取的日志创建时间
filename	%(filename)s	pathname 中的文件名称部分
funcName	%(funcName)s	日志输出位置的函数名称
levelname	%(levelname)s	字符串形式的日志级别
levelno	%(levelno)s	数字形式的日志级别
lineno	%(lineno)d	输出日志的源码行号
message	%(message)s	用户传入的经过格式化的日志内容
module	%(module)s	filename 中的模块名部分
msecs	%(msecs)d	日志创建时间的毫秒部分
name	%(name)s	logger 的名称
pathname	%(pathname)s	源码的路径
process	%(process)d	进程 ID
processName	%(processName)s	进程名
relativeCreated	%(relativeCreated)d	相对于 logging 模块加载时间的毫秒数
thread	%(thread)d	线程 ID
threadName	%(threadName)s	线程名

```
logger = logging.getLogger('log')
logger.warning('a warning message')  # a warning message
```

执行上述代码，会发现，logger 并没有输出列表中列出的 LogRecord 的各种属性，只有 message 内容。因为 LogRecord 只是承载每条日志内容和属性的对象，在一条 log 产生的时候就被创建了，而日志的输出格式则是在被输出时才确定，由 Formatter 来控制。Formatter 负责将一条 log（以 LogRecord 对象的形式存在）转换为可读的字符串，默认情况下，格式是`%(message)s`，所以当没有指定 Formatter 时，只输出用户传入的内容。

Logger、Handler、Filter

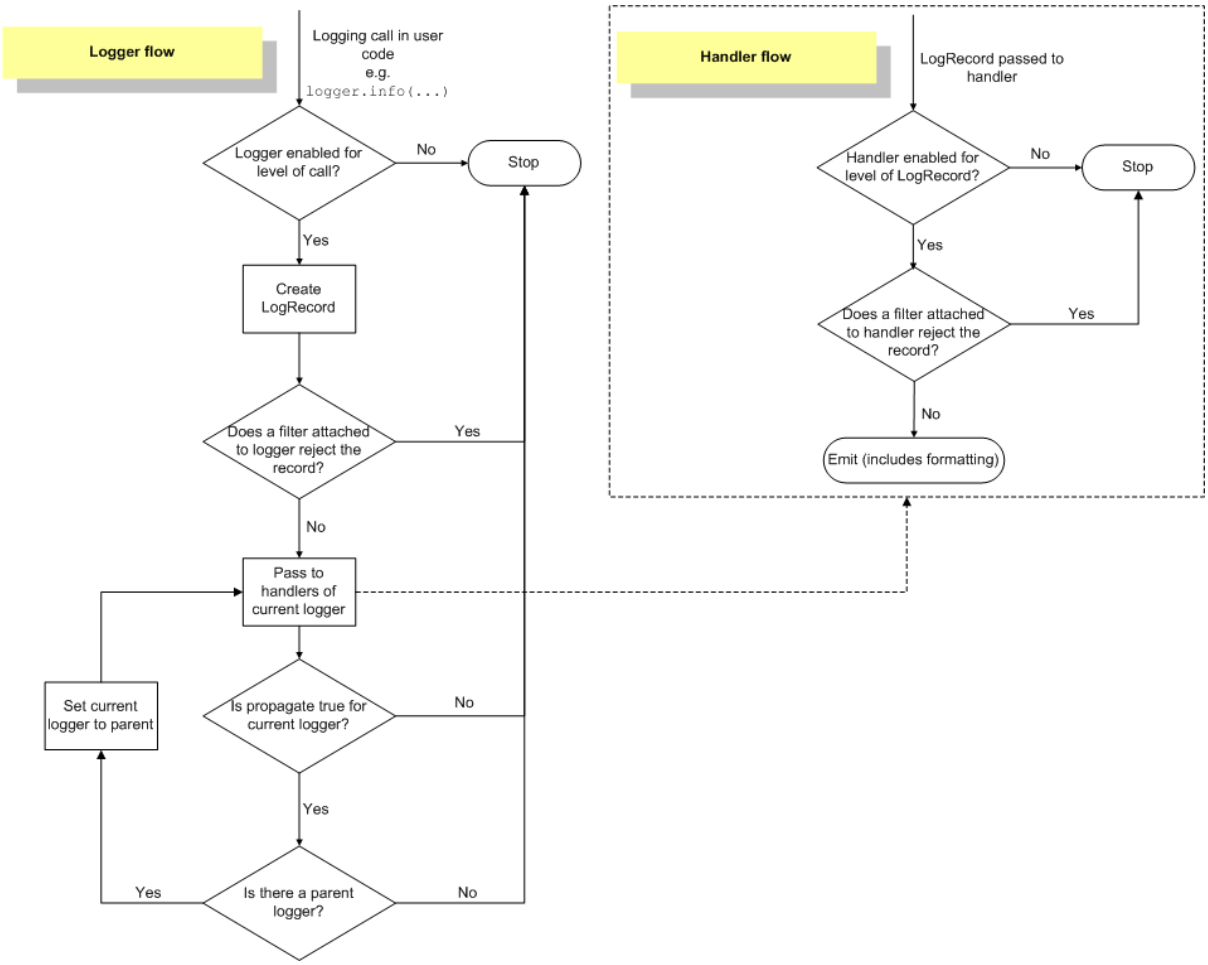
Logger 对象是 logging 库中最为常用的对象，Logger 对象的作用主要有三个：

1. 为应用暴露出 info、warning、error 等方法，应用可以通过这些方法创建对应级别的日志。
2. 根据 Filter 和日志级别的设置，来决定哪些日志可以被传入给下一个流程处理，
3. 将日志传递到所有相关的 Handler 中。

同时 Logger 对象还可以继承，一个 Logger 可以把 LogRecord 传递给父级的 Logger。

Handler 负责将日志写入到最终的归宿，可能文件、电子邮件、内存、队列... 由于一个 Logger 可以有多个 Handler，所以每个 Handler 都可以设置接收日志的级别和 Filter，换句话说，不同级别的日志可以输出到不同的归宿。

Python 官方文档提供了 logging 处理日志的逻辑流程图。



这里我们可能会有一个疑问，为 Logger 和 Handler 设置日志级别已经可以表示处理哪些日志，不处理哪些日志，为什么还需要 Filter 呢?

相比于日志级别，Filter 可定制性更丰富，可以在 Logger 和 Handler 上实现颗粒度更细的控制。例如希望只记录长度大于 10 的日志，可以用如下的代码实现。

```
class CustomFilter(logging.Filterer):
    def filter(self, record):
        return len(record.msg) > 10


logger = logging.getLogger('log')
filter = CustomFilter()
logger.addFilter(filter)
logger.warning('a warning message')  # a warning message
logger.warning('a warn')
logger.warning('another warning message')  # another warning message
```

长度小于 10 的第二条 log 并不会被输出。

实践中的一些经验

在使用 Python logging 库的过程中，我们发现了一些容易容易忽视的细节，这些细节可能会导致一些预期之外的情况，在此做以总结。

Logger 的继承链

Logger 对象是有一条继承链的，使用 logging.getLogger() 方法获取 logger 时，获取的是 root logger。如果为 getLogger 方法传入参数，获取的是子 logger。

```
root_logger = logging.getLogger()
sub_logger = logging.getLogger('log')
print(sub_logger.parent == root_logger)  # True
```

logging 的官方文档中推荐使用 __name__ 作为 getLogger 的参数，__name__ 是 module 的路径名，例如在 utils.log 包中使用 logging.getLogger(__name__) 相当于执行 logging.getLogger('base.db') ，这样就创建了一个名为 base.db 的 logger，这个 db 包的 logger 继承自 root logger。

如果我们在 base 中也创建一个 logger，logger.getLogger('base') ，这时候，base logger 也继承自 root logger，但是 db logger 的继承顺序则被修改成了继承自 base logger。

```
root_logger = logging.getLogger()
db_logger = logging.getLogger('base.db')
print(db_logger.name)  # base.db
print(db_logger.parent.name)  # root
base_logger = logging.getLogger('base')  # base
print(db_logger.name)  # base.db
print(db_logger.parent.name)  # base
```

换句话说我们可以通过 xxx.xxx 的形式获取任何一级的 logger，但是这些中间层的 logger

并不一定是存在的。

Logger 奇葩的默认行为

```
root_logger = logging.getLogger()
root_logger.info('root info')
```

执行上面的代码，会发现没有任何输出，但是如果打一个 warning 级别的 log，是可以输出的。

```
root_logger = logging.getLogger()
sub_logger = logging.getLogger('sub')

print(root_logger.level)  # 30 = WARNING
print(sub_logger.level)  # 0 = NOTSET
```

打印一下 root logger 的默认级别，会发现 30 对应的是 WARNING，也就是说，只有比 WARNING 高的级别才会被输出出来，而 INFO 对应值是 20， 比 WARNING 低，所以默认情况下 root logger 将不会接受 INFO 级别的错误。

但是只有 root logger 的默认级别是 WARNING，其他 logger 的默认级别 是 NOTSET = 0。

```
root_logger = logging.getLogger()
sub_logger = logging.getLogger('sub')

root_logger.info('root info')
sub_logger.info('sub info')
```

执行上面代码，会发现依然没有任何输出，既然其他 logger 的默认级别是 NOTSET，为什么比 NOTSET 高的 INFO 还是不会输出呢?

当一个 logger 的 level 被设置为 NOTSET 时，如果有父 logger，会将 log 传递给父 logger 处理，只有在 logger 是 root logger 或 propagate 属性设置为 False 时，才会由自己处理。接下来再修改一下上面的代码。

```
sub_logger = logging.getLogger('sub')
sub_logger.propagate = False

sub_logger.info('sub info')
sub_logger.warning('sub warn')  # sub warn
```

上面的代码中禁用了 logger 的传递功能，所以 logger 会自己处理错误，但是 INFO 级别的日志依然没有被输出，如果输出一下 sub_logger.handlers 属性，会发现默认情况下 logger 并没有任何的 handlers，这能解释为什么无法输出日志，但是下一行代码输出了 WARNING 级别的日志，显然又是和这个猜测违背的，原因到底是什么呢?

跟踪源码会发现，当一个 logger 需要自己处理日志时且没有任何一个 handler 时，会尝试使用 lastResort 属性所存储的 handler 来处理。

文档中是这样定义 lastResort 的。

> A “handler of last resort” is available through this attribute. This is a StreamHandler writing to sys.stderr with a level of WARNING, and is used to handle logging events in the absence of any logging configuration. The end result is to just print the message to sys.stderr. This replaces the earlier error message saying that “no handlers could be found for logger XYZ”. If you need the earlier behaviour for some reason, lastResort can be set to None.

所以，当一个 logger 没有任何 handler 的时候，依然能输出 WARNING 及以上级别的日志。