python logging模块使用教程

简单使用

```
#!/usr/local/bin/python
# -*- coding:utf-8 -*-
import logging

logging.debug('debug message')
logging.info('info message')
logging.warn('warn message')
logging.error('error message')
logging.critical('critical message')
```

输出：

```
WARNING:root:warn message
ERROR:root:error message
CRITICAL:root:critical message
```

默认情况下，logging模块将日志打印到屏幕上(stdout)，日志级别为WARNING(即只有日志级别高于WARNING的日志信息才会输出)，日志格式如下图所示：

default_logging_format.png

问题来了

1. 日志级别等级及设置是怎样的?
2. 怎样设置日志的输出方式？比如输出到日志文件中?

简单配置

日志级别

级别	何时使用
DEBUG	详细信息，典型地调试问题时会感兴趣。
INFO	证明事情按预期工作。
WARNING	表明发生了一些意外，或者不久的将来会发生问题（如‘磁盘满了’）。软件还是在正常工作

	题（如“磁盘满了”）。软件还是在正常工作。
ERROR	由于更严重的问题，软件已不能执行一些功能了。
CRITICAL	严重错误，表明软件已不能继续运行了。

简单配置

```
#!/usr/local/bin/python
# -*- coding:utf-8 -*-
import logging

# 通过下面的方式进行简单配置输出方式与日志级别
logging.basicConfig(filename='logger.log', level=logging.INFO)

logging.debug('debug message')
logging.info('info message')
logging.warn('warn message')
logging.error('error message')
logging.critical('critical message')
```

输出：

标准输出(屏幕)未显示任何信息，发现当前工作目录下生成了logger.log，内容如下：

```
INFO:root:info message
WARNING:root:warn message
ERROR:root:error message
CRITICAL:root:critical message
```

因为通过*level=logging.INFO*设置日志级别为INFO，所以所有的日志信息均输出出来了。

问题又来了

1. 通过上述配置方法都可以配置那些信息？

在解决以上问题之前，需要先了解几个比较重要的概念，**Logger**，**Handler**，**Formatter**，**Filter**

几个重要的概念

- Logger 记录器，暴露了应用程序代码能直接使用的接口。
- Handler 处理器，将（记录器产生的）日志记录发送至合适的目的地。

- Filter 过滤器，提供了更好的粒度控制，它可以决定输出哪些日志记录。
- Formatter 格式化器，指明了最终输出中日志记录的布局。

Logger 记录器

Logger是一个树形层级结构，在使用接口debug，info，warn，error，critical之前必须创建Logger实例，即创建一个记录器，如果没有显式的进行创建，则默认创建一个root logger，并应用默认的日志级别(WARN)，处理器Handler(StreamHandler，即将日志信息打印输出在标准输出上)，和格式化器Formatter(默认的格式即为第一个简单使用程序中输出的格式)。

创建方法: `logger = logging.getLogger(logger_name)`

创建Logger实例后，可以使用以下方法进行日志级别设置，增加处理器Handler。

- logger.setLevel(logging.ERROR) # 设置日志级别为ERROR，即只有日志级别大于等于ERROR的日志才会输出
- logger.addHandler(handler_name) # 为Logger实例增加一个处理器
- logger.removeHandler(handler_name) # 为Logger实例删除一个处理器

Handler 处理器

Handler处理器类型有很多种，比较常用的有三个，**StreamHandler**，**FileHandler**，**NullHandler**，详情可以访问Python logging.handlers

创建StreamHandler之后，可以通过使用以下方法设置日志级别，设置格式化器Formatter，增加或删除过滤器Filter。

- ch.setLevel(logging.WARN) # 指定日志级别，低于WARN级别的日志将被忽略
- ch.setFormatter(formatter_name) # 设置一个格式化器formatter
- ch.addFilter(filter_name) # 增加一个过滤器，可以增加多个
- ch.removeFilter(filter_name) # 删除一个过滤器

StreamHandler

创建方法: `sh = logging.StreamHandler(stream=None)`

FileHandler

创建方法: `fh = logging.FileHandler(filename, mode='a', encoding=None, delay=False)`

NullHandler

NullHandler类位于核心logging包，不做任何的格式化或者输出。
本质上它是个“什么都不做”的handler，由库开发者使用。

Formatter 格式化器

使用Formatter对象设置日志信息最后的规则、结构和内容，默认的时间格式为%Y-%m-%d %H:%M:%S。

创建方法: `formatter = logging.Formatter(fmt=None, datefmt=None)`

其中，fmt是消息的格式化字符串，datefmt是日期字符串。如果不指明fmt，将使用'%(message)s'。如果不指明datefmt，将使用ISO8601日期格式。

Filter 过滤器

Handlers和Loggers可以使用Filters来完成比级别更复杂的过滤。Filter基类只允许特定Logger层次以下的事件。例如用‘A.B’初始化的Filter允许Logger‘A.B’，‘A.B.C’，‘A.B.C.D’，‘A.B.D’等记录的事件，logger‘A.BB’，‘B.A.B’ 等就不行。 如果用空字符串来初始化，所有的事件都接受。

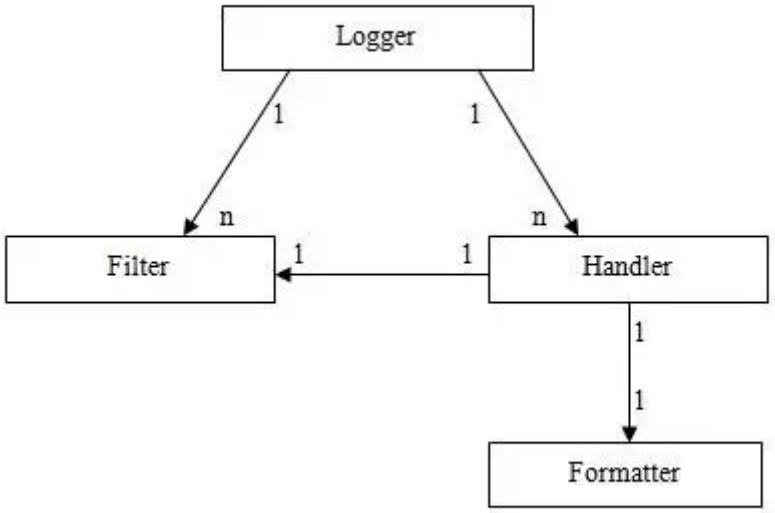
创建方法: `filter = logging.Filter(name='')`

以下是相关概念总结:

熟悉了这些概念之后，有另外一个比较重要的事情必须清楚，即**Logger是一个树形层级结构**;

Logger可以包含一个或多个Handler和Filter，即Logger与Handler或Fitler是一对多的关系;

一个Logger实例可以新增多个Handler，一个Handler可以新增多个格式化器或多个过滤器，而且日志级别将会继承。



element_relation.jpg

Logging工作流程

logging模块使用过程

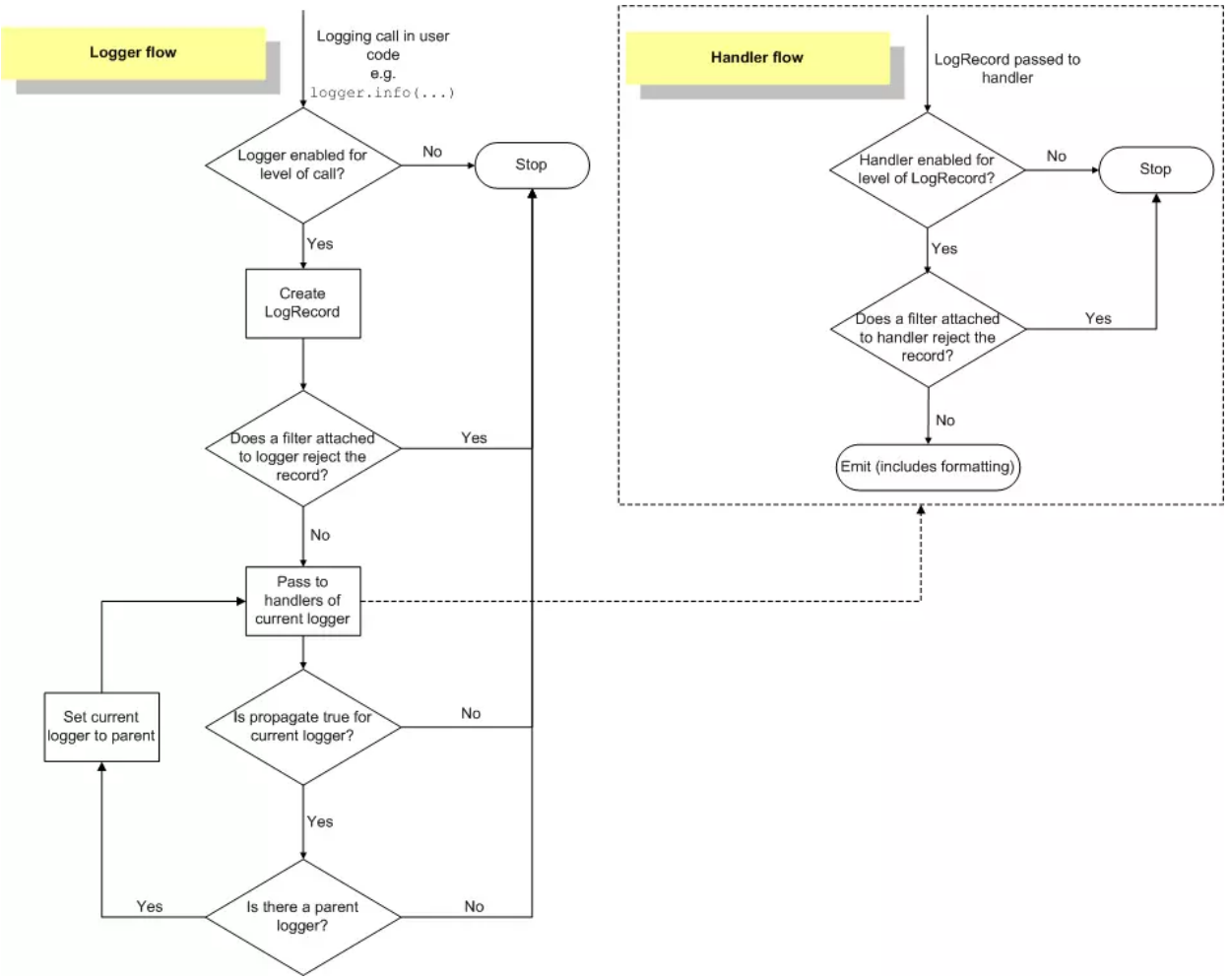
1. 第一次导入logging模块或使用reload函数重新导入logging模块，logging模块中的代码将被执行，这个过程中将产生logging日志系统的默认配置。

2. 自定义配置(可选)。logging标准模块支持三种配置方式: dictConfig，fileConfig，listen。其中，dictConfig是通过一个字典进行配置Logger，Handler，Filter，Formatter；fileConfig则是通过一个文件进行配置；而listen则监听一个网络端口，通过接收网络数据来进行配置。当然，除了以上集体化配置外，也可以直接调用Logger，Handler等对象中的方法在代码中来显式配置。

3. 使用logging模块的全局作用域中的getLogger函数来得到一个Logger对象实例(其参数即是一个字符串，表示Logger对象实例的名字，即通过该名字来得到相应的Logger对象实例)。

4. 使用Logger对象中的debug，info，error，warn，critical等方法记录日志信息。

logging模块处理流程



logging_flow.png

1. 判断日志的等级是否大于Logger对象的等级，如果大于，则往下执行，否则，流程结束。

2. 产生日志。第一步，判断是否有异常，如果有，则添加异常信息。第二步，处理日志记录方法(如debug，info等)中的占位符，即一般的字符串格式化处理。

3. 使用注册到Logger对象中的Filters进行过滤。如果有多个过滤器，则依次过滤；只要有一个过滤器返回假，则过滤结束，且该日志信息将丢弃，不再处理，而处理流程也至此结束。否则，处理流程往下执行。

4. 在当前Logger对象中查找Handlers，如果找不到任何Handler，则往上到该Logger对象的父Logger中查找；如果找到一个或多个Handler，则依次用Handler来处理日志信息。但在每个Handler处理日志信息过程中，会首先判断日志信息的等级是否大于该Handler的等级，如果大于，则往下执行(由Logger对象进入Handler对象中)，否则，处理流程结束。

5. 执行Handler对象中的filter方法，该方法会依次执行注册到该Handler对象中的Filter。如果有一个Filter判断该日志信息为假，则此后的所有Filter都不再执行，而直接将该日志信息丢弃，处理流程结束。

6. 使用Formatter类格式化最终的输出结果。 注：Formatter同上述第2步的字符串格式化不同，它会添加额外的信息，比如日志产生的时间，产生日志的源代码所在的源文件的路径等等。

7. 真正地输出日志信息(到网络，文件，终端，邮件等)。至于输出到哪个目的地，由Handler的种类来决定。

注：以上内容摘抄自第三条参考资料，内容略有改动，转载特此声明。

再看日志配置

配置方式

- 显式创建记录器Logger、处理器Handler和格式化器Formatter，并进行相关设置；
- 通过简单方式进行配置，使用basicConfig()函数直接进行配置；
- 通过配置文件进行配置，使用fileConfig()函数读取配置文件；
- 通过配置字典进行配置，使用dictConfig()函数读取配置信息；
- 通过网络进行配置，使用listen()函数进行网络配置。

basicConfig关键字参数

关键字	描述
filename	创建一个FileHandler，使用指定的文件名，而不是使用StreamHandler。
filemode	如果指明了文件名，指明打开文件的模式（如果没有指明filemode，默认为'a'）。
format	handler使用指明的格式化字符串。
datefmt	使用指明的日期 / 时间格式。
level	指明根logger的级别。
stream	使用指明的流来初始化StreamHandler。该参数与'filename'不兼容，如果两个都有，'stream'被忽略。

有用的format格式

格式	描述

格式	描述
%(levelno)s	打印日志级别的数值
%(levelname)s	打印日志级别名称
%(pathname)s	打印当前执行程序的路径
%(filename)s	打印当前执行程序名称
%(funcName)s	打印日志的当前函数
%(lineno)d	打印日志的当前行号
%(asctime)s	打印日志的时间
%(thread)d	打印线程id
%(threadName)s	打印线程名称
%(process)d	打印进程ID
%(message)s	打印日志信息

有用的datefmt格式

参考time.strftime

配置示例

显式配置

使用程序logger.py如下:

```
# -*- encoding:utf-8 -*-
import logging

# create logger
logger_name = "example"
logger = logging.getLogger(logger_name)
logger.setLevel(logging.DEBUG)

# create file handler
log_path = "./log.log"
fh = logging.FileHandler(log_path)
fh.setLevel(logging.WARN)

# create formatter
fmt = "%(asctime)-15s %(levelname)s %(filename)s %(lineno)d %(process)d
%(message)s"
datefmt = "%a %d %b %Y %H:%M:%S"
```

```
formatter = logging.Formatter(fmt, datefmt)

# add handler and formatter to logger
fh.setFormatter(formatter)
logger.addHandler(fh)

# print log info
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')```

##### 文件配置
配置文件logging.conf如下:
```[loggers]
keys=root,example01

[logger_root]
level=DEBUG
handlers=hand01,hand02

[logger_example01]
handlers=hand01,hand02
qualname=example01
propagate=0

[handlers]
keys=hand01,hand02

[handler_hand01]
class=StreamHandler
level=INFO
formatter=form02
args=(sys.stderr,)

[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form01
args=('log.log', 'a')

[formatters]
keys=form01,form02

[formatter_form01]
format=%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %
```
```

(message)s```

使用程序logger.py如下:
```#!/usr/bin/python
# -*- encoding:utf-8 -*-
import logging
import logging.config

logging.config.fileConfig("./logging.conf")

# create logger
logger_name = "example"
logger = logging.getLogger(logger_name)

logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')```

##### 字典配置
有兴趣的童靴可以使用```logging.config.dictConfig(config)```编写一个示例程序发给我，以提供给我进行完善本文。

##### 监听配置
有兴趣的童靴可以使用
```logging.config.listen(port=DEFAULT_LOGGING_CONFIG_PORT)```编写一个示例程序发给我，以提供给我进行完善本文。

更多详细内容参考[logging.config日志配置]
(http://python.usyiyi.cn/python_278/library/logging.config.html#module-logging.config)

### 参考资料
* [英文Python logging HOWTO]
(https://docs.python.org/2/howto/logging.html#logging-basic-tutorial)
* [中文Python 日志 HOWTO]
(http://python.usyiyi.cn/python_278/howto/logging.html#logging-basic-tutorial)
* [Python日志系统Logging](http://www.52ij.com/jishu/666.html)
* [logging模块学习笔记: basicConfig配置文件]
(http://www.cnblogs.com/bjdxy/archive/2013/04/12/3016820.html)
* 其他一些前辈博客相关文章

## Python之日志处理（logging模块）

# 本节内容

1. 日志相关概念
2. logging模块简介
3. 使用logging提供的模块级别的函数记录日志
4. logging模块日志流处理流程
5. 使用logging四大组件记录日志
6. 配置logging的几种方式
7. 向日志输出中添加上下文信息
8. 参考文档

# 一、日志相关概念

日志是一种可以追踪某些软件运行时所发生事件的方法。软件开发人员可以向他们的代码中调用日志记录相关的方法来表明发生了某些事情。一个事件可以用一个可包含可选变量数据的消息来描述。此外，事件也有重要性的概念，这个重要性也可以被称为严重性级别（level）。

## 1.日志的作用

通过log的分析，可以方便用户了解系统或软件、应用的运行情况；如果你的应用log足够丰富，也可以分析以往用户的操作行为、类型喜好、地域分布或其他更多信息；如果一个应用的log同时也分了多个级别，那么可以很轻易地分析得到该应用的健康状况，及时发现问题并快速定位、解决问题，补救损失。
简单来讲就是，我们通过记录和分析日志可以了解一个系统或软件程序运行情况是否正常，也可以在应用程序出现故障时快速定位问题。比如，做运维的同学，在接收到报警或各种问题反馈后，进行问题排查时通常都会先去看各种日志，大部分问题都可以在日志中找到答案。再比如，做开发的同学，可以通过IDE控制台上输出的各种日志进行程序调试。对于运维老司机或者有经验的开发人员，可以快速的通过日志定位到问题的根源。可见，日志的重要性不可小觑。日志的作用可以简单总结为以下3点：

- 程序调试
- 了解软件程序运行情况，是否正常
- 软件程序运行故障分析与问题定位

如果应用的日志信息足够详细和丰富，还可以用来做用户行为分析，如：分析用户的操作行为、类型洗好、地域分布以及其它更多的信息，由此可以实现改进业务、提高商业利益。

## 2.日志的等级

我们先来思考下下面的两个问题：

- 作为开发人员，在开发一个应用程序时需要什么日志信息？在应用程序正式上线后需要什么日志信息?
- 作为应用运维人员，在部署开发环境时需要什么日志信息？在部署生产环境时需要什么日志信息?

在软件开发阶段或部署开发环境时，为了尽可能详细的查看应用程序的运行状态来保证上线后的稳定性，我们可能需要把该应用程序所有的运行日志全部记录下来进行分析，这是非常耗费机器性能的。当应用程序正式发布或在生产环境部署应用程序时，我们通常只需要记录应用程序的异常信息、错误信息等，这样既可以减小服务器的I/O压力，也可以避免我们在排查故障时被淹没在日志的海洋里。那么，怎样才能在不改动应用程序代码的情况下实现在不同的环境记录不同详细程度的日志呢？这就是日志等级的作用了，我们通过配置文件指定我们需要的日志等级就可以了。

不同的应用程序所定义的日志等级可能会有所差别，分的详细点的会包含以下几个等级：

- DEBUG
- INFO
- NOTICE
- WARNING
- ERROR
- CRITICAL
- ALERT
- EMERGENCY

### 3.日志字段信息与日志格式

本节开始问题提到过，一条日志信息对应的是一个事件的发生，而一个事件通常需要包括以下几个内容：

- 事件发生时间
- 事件发生位置
- 事件的严重程度--日志级别
- 事件内容

上面这些都是一条日志记录中可能包含的字段信息，当然还可以包括一些其他信息，如进程ID、进程名称、线程ID、线程名称等。日志格式就是用来定义一条日志记录中包含那些字段的，且日志格式通常都是可以自定义的。

> ***说明：***
> 输出一条日志时，日志内容和日志级别是需要开发人员明确指定的。对于而其它字段信息，只需要是否显示在日志中就可以了。

### 4.日志功能的实现

几乎所有开发语言都会内置日志相关功能，或者会有比较优秀的第三方库来提供日志操作功能，比如：log4j，log4php等。它们功能强大、使用简单。Python自身也提供了一个用于记录日志的标准库模块--logging。

## 二、logging模块简介

logging模块定义的函数和类为应用程序和库的开发实现了一个灵活的事件日志系统。logging模块是Python的一个标准库模块，由标准库模块提供日志记录API的关键好处是所有Python模块都可以使用这个日志记录功能。所以，你的应用日志可以将你自己的日志信息与来自第三方模块的信息整合起来。

### 1. logging模块的日志级别

logging模块默认定义了以下几个日志等级，它允许开发人员自定义其他日志级别，但是这是不被推荐的，尤其是在开发供别人使用的库时，因为这会导致日志级别的混乱。

| 日志等级（level） | 描述 |
| --- | --- |
| DEBUG | 最详细的日志信息，典型应用场景是 问题诊断 |
| INFO | 信息详细程度仅次于DEBUG，通常只记录关键节点信息，用于确认一切都是按照我们预期的那样进行工作 |
| WARNING | 当某些不期望的事情发生时记录的信息（如，磁盘可用空间较低），但是此时应用程序还是正常运行的 |
| ERROR | 由于一个更严重的问题导致某些功能不能正常运行时记录的信息 |

| | |
|---|---|
| CRITICAL | 当发生严重错误，导致应用程序不能继续运行时记录的信息 |

开发应用程序或部署开发环境时，可以使用DEBUG或INFO级别的日志获取尽可能详细的日志信息来进行开发或部署调试；应用上线或部署生产环境时，应该使用WARNING或ERROR或CRITICAL级别的日志来降低机器的I/O压力和提高获取错误日志信息的效率。日志级别的指定通常都是在应用程序的配置文件中进行指定的。

***说明：***

- 上面列表中的日志等级是从上到下依次升高的，即：DEBUG < INFO < WARNING < ERROR < CRITICAL，而日志的信息量是依次减少的；
- 当为某个应用程序指定一个日志级别后，应用程序会记录所有日志级别大于或等于指定日志级别的日志信息，而不是仅仅记录指定级别的日志信息，nginx、php等应用程序以及这里要提高的python的logging模块都是这样的。同样，logging模块也可以指定日志记录器的日志级别，只有级别大于或等于该指定日志级别的日志记录才会被输出，小于该等级的日志记录将会被丢弃。

## 2. logging模块的使用方式介绍

logging模块提供了两种记录日志的方式：

- 第一种方式是使用logging提供的模块级别的函数
- 第二种方式是使用Logging日志系统的四大组件

其实，logging所提供的模块级别的日志记录函数也是对logging日志系统相关类的封装而已。

**logging模块定义的模块级别的常用函数**

| 函数 | 说明 |
|---|---|
| logging.debug(msg, *args, **kwargs) | 创建一条严重级别为DEBUG的日志记录 |
| logging.info(msg, *args, **kwargs) | 创建一条严重级别为INFO的日志记录 |
| logging.warning(msg, *args, **kwargs) | 创建一条严重级别为WARNING的日志记录 |
| logging.error(msg, *args, **kwargs) | 创建一条严重级别为ERROR的日志记录 |
| logging.critical(msg, *args, | 创建一条严重级别为CRITICAL |

| | |
|---|---|
| **kwargs) | 的日志记录 |
| logging.log(level, *args, **kwargs) | 创建一条严重级别为level的日志记录 |
| logging.basicConfig(**kwargs) | 对root logger进行一次性配置 |

其中`logging.basicConfig(**kwargs)`函数用于指定“要记录的日志级别”、“日志格式”、“日志输出位置”、“日志文件的打开模式”等信息，其他几个都是用于记录各个级别日志的函数。

**logging模块的四大组件**

| 组件 | 说明 |
|---|---|
| loggers | 提供应用程序代码直接使用的接口 |
| handlers | 用于将日志记录发送到指定的目的位置 |
| filters | 提供更细粒度的日志过滤功能，用于决定哪些日志记录将会被输出（其它的日志记录将会被忽略） |
| formatters | 用于控制日志信息的最终输出格式 |

***说明：*** logging模块提供的模块级别的那些函数实际上也是通过这几个组件的相关实现类来记录日志的，只是在创建这些类的实例时设置了一些默认值。

# 三、使用logging提供的模块级别的函数记录日志

回顾下前面提到的几个重要信息：

- 可以通过logging模块定义的模块级别的方法去完成简单的日志记录
- 只有级别大于或等于日志记录器指定级别的日志记录才会被输出，小于该级别的日志记录将会被丢弃。

## 1.最简单的日志输出

先来试着分别输出一条不同日志级别的日志记录：

```
import logging


logging.debug("This is a debug log.")
logging.info("This is a info log.")
logging.warning("This is a warning log.")
logging.error("This is a error log.")
logging.critical("This is a critical log.")
```

也可以这样写：

```
logging.log(logging.DEBUG, "This is a debug log.")
logging.log(logging.INFO, "This is a info log.")
logging.log(logging.WARNING, "This is a warning log.")
logging.log(logging.ERROR, "This is a error log.")
logging.log(logging.CRITICAL, "This is a critical log.")
```

输出结果：

```
WARNING:root:This is a warning log.
ERROR:root:This is a error log.
CRITICAL:root:This is a critical log.
```

2. 那么问题来了

问题1：为什么前面两条日志没有被打印出来?

这是因为logging模块提供的日志记录函数所使用的日志器设置的日志级别是`WARNING`，因此只有`WARNING`级别的日志记录以及大于它的`ERROR`和`CRITICAL`级别的日志记录被输出了，而小于它的`DEBUG`和`INFO`级别的日志记录被丢弃了。

问题2：打印出来的日志信息中各字段表示什么意思？为什么会这样输出?

上面输出结果中每行日志记录的各个字段含义分别是：

日志级别:日志器名称:日志内容

之所以会这样输出，是因为logging模块提供的日志记录函数所使用的日志器设置的日志格式默认是BASIC_FORMAT，其值为：

```
"%(levelname)s:%(name)s:%(message)s"
```

问题3：如果将日志记录输出到文件中，而不是打印到控制台?

因为在logging模块提供的日志记录函数所使用的日志器设置的处理器所指定的日志输出位置默认为：`sys.stderr`。

问题4：我是怎么知道这些的?

查看这些日志记录函数的实现代码，可以发现：当我们没有提供任何配置信息的时候，这些函数都会去调用`logging.basicConfig(**kwargs)`方法，且不会向该方法传递任何参数。继续查看`basicConfig()`方法的代码就可以找到上面这些问题的答案了。

问题5：怎么修改这些默认设置呢?

其实很简单，在我们调用上面这些日志记录函数之前，手动调用一下basicConfig()方法，把我们想设置的内容以参数的形式传递进去就可以了。

3. logging.basicConfig()函数说明

该方法用于为logging日志系统做一些基本配置，方法定义如下：

```
logging.basicConfig(**kwargs)
```

该函数可接收的关键字参数如下：

参数名称	描述
filename	指定日志输出目标文件的文件名，指定该设置项后日志信心就不会被输出到控制台了
filemode	指定日志文件的打开模式，默认为'a'。需要注意的是，该选项要在filename指定时才有效
format	指定日志格式字符串，即指定日志输出时所包含的字段信息以及它们的顺序。logging模块定义的格式字段下面会列出。
datefmt	指定日期/时间格式。需要注意的是，该选项要在format中包含时间字段%

	(asctime)s时才有效
level	指定日志器的日志级别
stream	指定日志输出目标stream，如sys.stdout、sys.stderr以及网络stream。需要说明的是，stream和filename不能同时提供，否则会引发`ValueError`异常
style	Python 3.2中新添加的配置项。指定format格式字符串的风格，可取值为'%'、'{'和'$'，默认为'%'
handlers	Python 3.3中新添加的配置项。该选项如果被指定，它应该是一个创建了多个Handler的可迭代对象，这些handler将会被添加到root logger。需要说明

	的是：filename、stream和handlers这三个配置项只能有一个存在，不能同时出现2个或3个，否则会引发ValueError异常。

4. logging模块定义的格式字符串字段

我们来列举一下logging模块中定义好的可以用于format格式字符串中字段有哪些：

字段/属性名称	使用格式	描述
asctime	%(asctime)s	日志事件发生的时间--人类可读时间，如：2003-07-08 16:49:45,896
created	%(created)f	日志事件发生的时间--时间戳，就是当时调用time.time()函数返回的值
relativeCreated	%(relativeCreated)d	日志事件发生的时间相对于logging模块加载时间的相对毫秒数（目前还不知道干嘛用的）
msecs	%(msecs)d	日志事件发生事件的毫秒部分

levelname	%(levelname)s	该日志记录的文字形式的日志级别（'DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL'）
levelno	%(levelno)s	该日志记录的数字形式的日志级别（10, 20, 30, 40, 50）
name	%(name)s	所使用的日志器名称，默认是'root'，因为默认使用的是rootLogger
message	%(message)s	日志记录的文本内容，通过`msg % args`计算得到的
pathname	%(pathname)s	调用日志记录函数的源码文件的全路径
filename	%(filename)s	pathname的文件名部分，包含文件后缀
module	%(module)s	filename的名称部分，不包含后缀
lineno	%(lineno)d	调用日志记录函数的源代码所在的行

		号
funcName	%(funcName)s	调用日志记录函数的函数名
process	%(process)d	进程ID
processName	%(processName)s	进程名称，Python 3.1新增
thread	%(thread)d	线程ID
threadName	%(thread)s	线程名称

5.经过配置的日志输出

先简单配置下日志器的日志级别

```
logging.basicConfig(level=logging.DEBUG)

logging.debug("This is a debug log.")
logging.info("This is a info log.")
logging.warning("This is a warning log.")
logging.error("This is a error log.")
logging.critical("This is a critical log.")
```

输出结果：

```
DEBUG:root:This is a debug log.
INFO:root:This is a info log.
WARNING:root:This is a warning log.
ERROR:root:This is a error log.
CRITICAL:root:This is a critical log.
```

所有等级的日志信息都被输出了，说明配置生效了。

在配置日志器日志级别的基础上，在配置下日志输出目标文件和日志格式

```
LOG_FORMAT = "%(asctime)s - %(levelname)s - %(message)s"
logging.basicConfig(filename='my.log', level=logging.DEBUG, format=LOG_FORMAT)

logging.debug("This is a debug log.")
logging.info("This is a info log.")
logging.warning("This is a warning log.")
logging.error("This is a error log.")
logging.critical("This is a critical log.")
```

此时会发现控制台中已经没有输出日志内容了，但是在python代码文件的相同目录下会生成一个名为'my.log'的日志文件，该文件中的内容为：

```
2017-05-08 14:29:53,783 - DEBUG - This is a debug log.
2017-05-08 14:29:53,784 - INFO - This is a info log.
2017-05-08 14:29:53,784 - WARNING - This is a warning log.
2017-05-08 14:29:53,784 - ERROR - This is a error log.
2017-05-08 14:29:53,784 - CRITICAL - This is a critical log.
```

在上面的基础上，我们再来设置下日期/时间格式

```
LOG_FORMAT = "%(asctime)s - %(levelname)s - %(message)s"
DATE_FORMAT = "%m/%d/%Y %H:%M:%S %p"

logging.basicConfig(filename='my.log', level=logging.DEBUG, format=LOG_FORMAT,
datefmt=DATE_FORMAT)

logging.debug("This is a debug log.")
logging.info("This is a info log.")
logging.warning("This is a warning log.")
logging.error("This is a error log.")
logging.critical("This is a critical log.")
```

此时会在my.log日志文件中看到如下输出内容:

```
05/08/2017 14:29:04 PM - DEBUG - This is a debug log.
05/08/2017 14:29:04 PM - INFO - This is a info log.
05/08/2017 14:29:04 PM - WARNING - This is a warning log.
05/08/2017 14:29:04 PM - ERROR - This is a error log.
05/08/2017 14:29:04 PM - CRITICAL - This is a critical log.
```

掌握了上面的内容之后，已经能够满足我们平时开发中需要的日志记录功能。

6. 其他说明

几个要说明的内容:

- `logging.basicConfig()`函数是一个一次性的简单配置工具使，也就是说只有在第一次调用该函数时会起作用，后续再次调用该函数时完全不会产生任何操作的，多次调用的设置并不是累加操作。
- 日志器（Logger）是有层级关系的，上面调用的logging模块级别的函数所使用的日志器是`RootLogger`类的实例，其名称为'root'，它是处于日志器层级关系最顶层的日志器，且该实例是以单例模式存在的。
- 如果要记录的日志中包含变量数据，可使用一个格式字符串作为这个事件的描述消息（logging.debug、logging.info等函数的第一个参数），然后将变量数据作为第二个参数*args的值进行传递，如:`logging.warning('%s is %d years old.', 'Tom', 10)`，输出内容为`WARNING:root:Tom is 10 years old.`
- logging.debug(), logging.info()等方法的定义中，除了msg和args参数外，还有一个**kwargs参数。它们支持3个关键字参数: `exc_info, stack_info, extra`，下面对这几个关键字参数作个说明。

关于exc_info, stack_info, extra关键词参数的说明:

- ***exc_info：*** 其值为布尔值，如果该参数的值设置为True，则会将异常异常信息添加到日志消息中。如果没有异常信息则添加None到日志信息中。
- ***stack_info：*** 其值也为布尔值，默认值为False。如果该参数的值设置为True，栈信息将会被添加到日志信息中。
- ***extra：*** 这是一个字典（dict）参数，它可以用来自定义消息格式中所包含的字段，但是它的key不能与logging模块定义的字段冲突。

一个例子：

在日志消息中添加exc_info和stack_info信息，并添加两个自定义的字端 ip和user

```
LOG_FORMAT = "%(asctime)s - %(levelname)s - %(user)s[%(ip)s] - %(message)s"
DATE_FORMAT = "%m/%d/%Y %H:%M:%S %p"

logging.basicConfig(format=LOG_FORMAT, datefmt=DATE_FORMAT)
logging.warning("Some one delete the log file.", exc_info=True, stack_info=True,
extra={'user': 'Tom', 'ip':'47.98.53.222'})
```

输出结果：

```
05/08/2017 16:35:00 PM - WARNING - Tom[47.98.53.222] - Some one delete the log
file.
NoneType
Stack (most recent call last):
  File "C:/Users/wader/PycharmProjects/LearnPython/day06/log.py", line 45, in
<module>
    logging.warning("Some one delete the log file.", exc_info=True,
stack_info=True, extra={'user': 'Tom', 'ip':'47.98.53.222'})
```

四、logging模块日志流处理流程

在介绍logging模块的高级用法之前，很有必要对logging模块所包含的重要组件以及其工作流程做个全面、简要的介绍，这有助于我们更好的理解我们所写的代码（将会触发什么样的操作）。

1. logging日志模块四大组件

在介绍logging模块的日志流处理流程之前，我们先来介绍下logging模块的四大组件：

组件名称	对应类名	功能描述
日志器	Logger	提供了应用程序可一直使用的接口
处理器	Handler	将logger创建的日志记录发送到合适的目的输出
		提供了更细粒度的控制工具

过滤器	Filter	来决定输出哪条日志记录，丢弃哪条日志记录
格式器	Formatter	决定日志记录的最终输出格式

logging模块就是通过这些组件来完成日志处理的，上面所使用的logging模块级别的函数也是通过这些组件对应的类来实现的。

这些组件之间的关系描述：

- 日志器（logger）需要通过处理器（handler）将日志信息输出到目标位置，如：文件、sys.stdout、网络等；
- 不同的处理器（handler）可以将日志输出到不同的位置；
- 日志器（logger）可以设置多个处理器（handler）将同一条日志记录输出到不同的位置；
- 每个处理器（handler）都可以设置自己的过滤器（filter）实现日志过滤，从而只保留感兴趣的日志；
- 每个处理器（handler）都可以设置自己的格式器（formatter）实现同一条日志以不同的格式输出到不同的地方。

简单点说就是：日志器（logger）是入口，真正干活儿的是处理器（handler），处理器（handler）还可以通过过滤器（filter）和格式器（formatter）对要输出的日志内容做过滤和格式化等处理操作。

2. logging日志模块相关类及其常用方法介绍

下面介绍下与logging四大组件相关的类：Logger, Handler, Filter, Formatter。

Logger类

Logger对象有3个任务要做：

- 1）向应用程序代码暴露几个方法，使应用程序可以在运行时记录日志消息；
- 2）基于日志严重等级（默认的过滤设施）或filter对象来决定要对哪些日志进行后续处理；
- 3）将日志消息传送给所有感兴趣的日志handlers。

Logger对象最常用的方法分为两类：配置方法 和 消息发送方法

最常用的配置方法如下：

方法	描述
Logger.setLevel()	设置日志器将会处理的日志消息的最低严重级别
Logger.addHandler() 和 Logger.removeHandler()	为该logger对象添加 和 移除一个handler对象
Logger.a	

Logger.addFilter()和Logger.removeFilter()	为该logger对象添加 和移除一个filter对象

关于Logger.setLevel()方法的说明:

内建等级中，级别最低的是DEBUG，级别最高的是CRITICAL。例如setLevel(logging.INFO)，此时函数参数为INFO，那么该logger将只会处理INFO、WARNING、ERROR和CRITICAL级别的日志，而DEBUG级别的消息将会被忽略/丢弃。

logger对象配置完成后，可以使用下面的方法来创建日志记录：

方法	描述
Logger.debug(), Logger.info(), Logger.warning(), Logger.error(), Logger.critical()	创建一个与它们的方法名对应等级的日志记录
Logger.exception()	创建一个类似于Logger.error()的日志消息
Logger.log()	需要获取一个明确的日志level参数来创建一个日志记录

说明:

- Logger.exception()与Logger.error()的区别在于：Logger.exception()将会输出堆栈追踪信息，另外通常只是在一个exception handler中调用该方法。
- Logger.log()与Logger.debug()、Logger.info()等方法相比，虽然需要多传一个level参数，显得不是那么方便，但是当需要记录自定义level的日志时还是需要该方法来完成。

那么，怎样得到一个Logger对象呢？一种方式是通过Logger类的实例化方法创建一个Logger类的实例，但是我们通常都是用第二种方式--logging.getLogger()方法。

logging.getLogger()方法有一个可选参数name，该参数表示将要返回的日志器的名称标识，如果不提供该参数，则其值为'root'。若以相同的name参数值多次调用getLogger()方法，将会返回指向同一个logger对象的引用。

关于logger的层级结构与有效等级的说明:

- logger的名称是一个以'.'分割的层级结构，每个'.'后面的logger都是'.'前面的logger的children，例如，有一个名称为 foo 的logger，其它名称分别为 foo.bar, foo.bar.baz 和

foo.bam都是 foo 的后代。

- logger有一个"有效等级（effective level）"的概念。如果一个logger上没有被明确设置一个level，那么该logger就是使用它parent的level;如果它的parent也没有明确设置level则继续向上查找parent的parent的有效level，依次类推，直到找到个一个明确设置了level的祖先为止。需要说明的是，root logger总是会有一个明确的level设置（默认为WARNING）。当决定是否去处理一个已发生的事件时，logger的有效等级将会被用来决定是否将该事件传递给该logger的handlers进行处理。
- child loggers在完成对日志消息的处理后，默认会将日志消息传递给与它们的祖先loggers相关的handlers。因此，我们不必为一个应用程序中所使用的所有loggers定义和配置handlers，只需要为一个顶层的logger配置handlers，然后按照需要创建child loggers就可足够了。我们也可以通过将一个logger的propagate属性设置为False来关闭这种传递机制。

Handler类

Handler对象的作用是（基于日志消息的level）将消息分发到handler指定的位置（文件、网络、邮件等）。Logger对象可以通过addHandler()方法为自己添加0个或者更多个handler对象。比如，一个应用程序可能想要实现以下几个日志需求：

- 1）把所有日志都发送到一个日志文件中；
- 2）把所有严重级别大于等于error的日志发送到stdout（标准输出）；
- 3）把所有严重级别为critical的日志发送到一个email邮件地址。

这种场景就需要3个不同的handlers，每个handler复杂发送一个特定严重级别的日志到一个特定的位置。

一个handler中只有非常少数的方法是需要应用开发人员去关心的。对于使用内建handler对象的应用开发人员来说，似乎唯一相关的handler方法就是下面这几个配置方法：

方法	描述
Handler.setLevel()	设置handler将会处理的日志消息的最低严重级别
Handler.setFormatter()	为handler设置一个格式器对象
Handler.addFilter()和Handler.removeFilter()	为handler添加 和 删除一个过滤器对象

需要说明的是，应用程序代码不应该直接实例化和使用Handler实例。因为Handler是一个基类，它只定义了素有handlers都应该有的接口，同时提供了一些子类可以直接使用或覆盖的默认行为。下面是一些常用的Handler：

Handler	描述

logging.StreamHandler	将日志消息发送到输出到Stream，如std.out, std.err或任何file-like对象。
logging.FileHandler	将日志消息发送到磁盘文件，默认情况下文件大小会无限增长
logging.handlers.RotatingFileHandler	将日志消息发送到磁盘文件，并支持日志文件按大小切割
logging.hanlders.TimedRotatingFileHandler	将日志消息发送到磁盘文件，并支持日志文件按时间切割
logging.handlers.HTTPHandler	将日志消息以GET或POST的方式发送给一个HTTP服务器
logging.handlers.SMTPHandler	将日志消息发送给一个指定的email地址
logging.NullHandler	该Handler实例会忽略error messages，通常被想使用logging的library开

	发者使用来避免'No handlers could be found for logger XXX'信息的出现。

Formater类

Formater对象用于配置日志信息的最终顺序、结构和内容。与logging.Handler基类不同的是，应用代码可以直接实例化Formatter类。另外，如果你的应用程序需要一些特殊的处理行为，也可以实现一个Formatter的子类来完成。

Formatter类的构造方法定义如下：

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

可见，该构造方法接收3个可选参数：

- fmt：指定消息格式化字符串，如果不指定该参数则默认使用message的原始值
- datefmt：指定日期格式字符串，如果不指定该参数则默认使用"%Y-%m-%d %H:%M:%S"
- style：Python 3.2新增的参数，可取值为 '%', '{'和 '$'，如果不指定该参数则默认使用'%'

Filter类

Filter可以被Handler和Logger用来做比level更细粒度的、更复杂的过滤功能。Filter是一个过滤器基类，它只允许某个logger层级下的日志事件通过过滤。该类定义如下：

```
class logging.Filter(name='')
    filter(record)
```

比如，一个filter实例化时传递的name参数值为'A.B'，那么该filter实例将只允许名称为类似如下规则的loggers产生的日志记录通过过滤：'A.B'，'A.B,C'，'A.B.C.D'，'A.B.D'，而名称为'A.BB', 'B.A.B'的loggers产生的日志则会被过滤掉。如果name的值为空字符串，则允许所有的日志事件通过过滤。

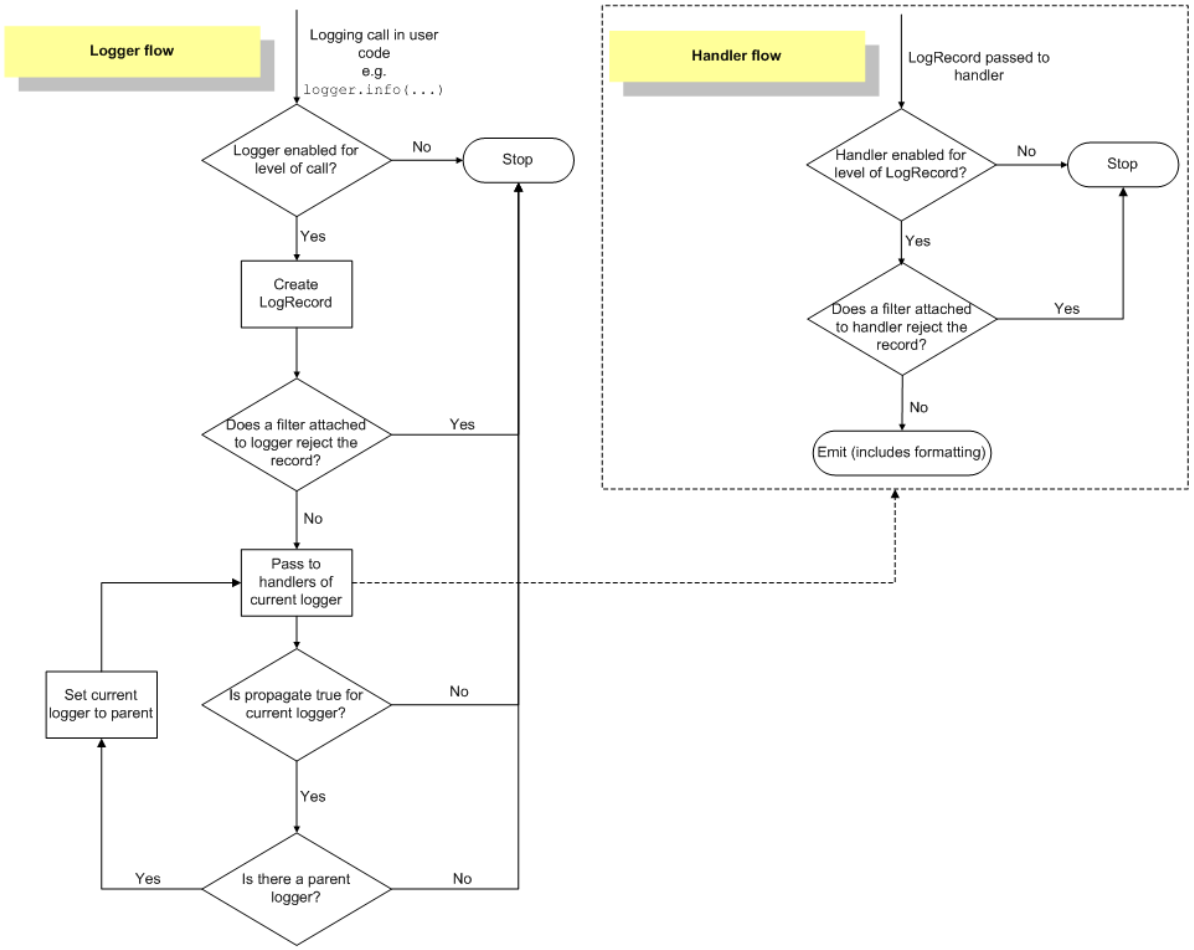
filter方法用于具体控制传递的record记录是否能通过过滤，如果该方法返回值为0表示不能通过过滤，返回值为非0表示可以通过过滤。

说明：

- 如果有需要，也可以在filter(record)方法内部改变该record，比如添加、删除或修改一些属性。
- 我们还可以通过filter做一些统计工作，比如可以计算下被一个特殊的logger或handler所处理的record数量等。

3. logging日志流处理流程

下面这个图描述了日志流的处理流程：



我们来描述下上面这个图的日志流处理流程：

- 1）（在用户代码中进行）日志记录函数调用，如：logger.info(...)，logger.debug(...)等；
- 2）判断要记录的日志级别是否满足日志器设置的级别要求（要记录的日志级别要大于或等于日志器设置的级别才算满足要求），如果不满足则该日志记录会被丢弃并终止后续的操作，如果满足则继续下一步操作；
- 3）根据日志记录函数调用时掺入的参数，创建一个日志记录（LogRecord类）对象；
- 4）判断日志记录器上设置的过滤器是否拒绝这条日志记录，如果日志记录器上的某个过滤器拒绝，则该日志记录会被丢弃并终止后续的操作，如果日志记录器上设置的过滤器不拒绝这条日志记录或者日志记录器上没有设置过滤器则继续下一步操作--将日志记录分别交给该日志器上添加的各个处理器；
- 5）判断要记录的日志级别是否满足处理器设置的级别要求（要记录的日志级别要大于或等于该处理器设置的日志级别才算满足要求），如果不满足记录将会被该处理器丢弃并终止后续的操作，如果满足则继续下一步操作；
- 6）判断该处理器上设置的过滤器是否拒绝这条日志记录，如果该处理器上的某个过滤器拒绝，则该日志记录会被当前处理器丢弃并终止后续的操作，如果当前处理器上设置的过滤器不拒绝这条日志记录或当前处理器上没有设置过滤器测继续下一步操作；
- 7）如果能到这一步，说明这条日志记录经过了层层关卡允许被输出了，此时当前处理器会根据自身被设置的格式器（如果没有设置则使用默认格式）将这条日志记录进行格式化，最后将格式化后的结果输出到指定位置（文件、网络、类文件的Stream等）；
- 8）如果日志器被设置了多个处理器的话，上面的第5-8步会执行多次；

- 9）这里才是完整流程的最后一步：判断该日志器输出的日志消息是否需要传递给上一级logger（之前提到过，日志器是有层级关系的）的处理器，如果propagate属性值为1则表示日志消息将会被输出到处理器指定的位置，同时还会被传递给parent日志器的handlers进行处理直到当前日志器的propagate属性为0停止，如果propagate值为0则表示不向parent日志器的handlers传递该消息，到此结束。

可见，一条日志信息要想被最终输出需要依次经过以下几次过滤：

- 日志器等级过滤；
- 日志器的过滤器过滤；
- 日志器的处理器等级过滤；
- 日志器的处理器的过滤器过滤；

需要说明的是： 关于上面第9个步骤，如果propagate值为1，那么日志消息会直接传递交给上一级logger的handlers进行处理，此时上一级logger的日志等级并不会对该日志消息进行等级过滤。

五、使用logging四大组件记录日志

现在，我们对logging模块的重要组件及整个日志流处理流程都应该有了一个比较全面的了解，下面我们来看一个例子。

1. 需求

现在有以下几个日志记录的需求：

- 1）要求将所有级别的所有日志都写入磁盘文件中
- 2）all.log文件中记录所有的日志信息，日志格式为：日期和时间 - 日志级别 - 日志信息
- 3）error.log文件中单独记录error及以上级别的日志信息，日志格式为：日期和时间 - 日志级别 - 文件名[:行号] - 日志信息
- 4）要求all.log在每天凌晨进行日志切割

2. 分析

- 1）要记录所有级别的日志，因此日志器的有效level需要设置为最低级别--DEBUG；
- 2）日志需要被发送到两个不同的目的地，因此需要为日志器设置两个handler；另外，两个目的地都是磁盘文件，因此这两个handler都是与FileHandler相关的；
- 3）all.log要求按照时间进行日志切割，因此他需要用logging.handlers.TimedRotatingFileHandler；而error.log没有要求日志切割，因此可以使用FileHandler；
- 4）两个日志文件的格式不同，因此需要对这两个handler分别设置格式器；

3. 代码实现

```
import logging
import logging.handlers
import datetime

logger = logging.getLogger('mylogger')
logger.setLevel(logging.DEBUG)

rf_handler = logging.handlers.TimedRotatingFileHandler('all.log',
when='midnight', interval=1, backupCount=7, atTime=datetime.time(0, 0, 0, 0))
```

```
rf_handler.setFormatter(logging.Formatter("%(asctime)s - %(levelname)s - %
(message)s"))

f_handler = logging.FileHandler('error.log')
f_handler.setLevel(logging.ERROR)
f_handler.setFormatter(logging.Formatter("%(asctime)s - %(levelname)s - %
(filename)s[:%(lineno)d] - %(message)s"))

logger.addHandler(rf_handler)
logger.addHandler(f_handler)

logger.debug('debug message')
logger.info('info message')
logger.warning('warning message')
logger.error('error message')
logger.critical('critical message')
```

all.log文件输出

```
2017-05-13 16:12:40,612 - DEBUG - debug message
2017-05-13 16:12:40,612 - INFO - info message
2017-05-13 16:12:40,612 - WARNING - warning message
2017-05-13 16:12:40,612 - ERROR - error message
2017-05-13 16:12:40,613 - CRITICAL - critical message
```

error.log文件输出

```
2017-05-13 16:12:40,612 - ERROR - log.py[:81] - error message
2017-05-13 16:12:40,613 - CRITICAL - log.py[:82] - critical message
```

六、配置logging的几种方式

作为开发者，我们可以通过以下3中方式来配置logging：

- 1）使用Python代码显式的创建loggers, handlers和formatters并分别调用它们的配置函数；
- 2）创建一个日志配置文件，然后使用`fileConfig()`函数来读取该文件的内容；
- 3）创建一个包含配置信息的dict，然后把它传递个`dictConfig()`函数；

具体说明请参考另一篇博文《python之配置日志的几种方式》

七、向日志输出中添加上下文信息

除了传递给日志记录函数的参数外，有时候我们还想在日志输出中包含一些额外的上下文信息。比如，在一个网络应用中，可能希望在日志中记录客户端的特定信息，如：远程客户端的IP地址和用户名。这里我们来介绍以下几种实现方式：

- 通过向日志记录函数传递一个`extra`参数引入上下文信息
- 使用LoggerAdapters引入上下文信息
- 使用Filters引入上下文信息

具体说明请参考另一篇博文《Python之向日志输出中添加上下文信息》
关于Python logging的更多高级用法，请参考文档<< Logging CookBook >>。

八、参考文档

- https://docs.python.org/3.5/howto/logging.html
- https://docs.python.org/3.5/library/logging.config.html
- https://docs.python.org/3.5/howto/logging-cookbook.html

python之配置日志的几种方式

作为开发者，我们可以通过以下3种方式来配置logging：

- 1）使用Python代码显式的创建loggers, handlers和formatters并分别调用它们的配置函数；
- 2）创建一个日志配置文件，然后使用`fileConfig()`函数来读取该文件的内容；
- 3）创建一个包含配置信息的dict，然后把它传递个`dictConfig()`函数；

需要说明的是，`logging.basicConfig()`也属于第一种方式，它只是对loggers, handlers和formatters的配置函数进行了封装。另外，第二种配置方式相对于第一种配置方式的优点在于，它将配置信息和代码进行了分离，这一方面降低了日志的维护成本，同时还使得非开发人员也能够去很容易地修改日志配置。

一、使用Python代码实现日志配置

代码如下：

```
# 创建一个日志器logger并设置其日志级别为DEBUG
logger = logging.getLogger('simple_logger')
logger.setLevel(logging.DEBUG)

# 创建一个流处理器handler并设置其日志级别为DEBUG
handler = logging.StreamHandler(sys.stdout)
handler.setLevel(logging.DEBUG)

# 创建一个格式器formatter并将其添加到处理器handler
formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s - %
(message)s")
handler.setFormatter(formatter)

# 为日志器logger添加上面创建的处理器handler
logger.addHandler(handler)

# 日志输出
logger.debug('debug message')
logger.info('info message')
```

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

运行输出：

```
2017-05-15 11:30:50,955 - simple_logger - DEBUG - debug message
2017-05-15 11:30:50,955 - simple_logger - INFO - info message
2017-05-15 11:30:50,955 - simple_logger - WARNING - warn message
2017-05-15 11:30:50,955 - simple_logger - ERROR - error message
2017-05-15 11:30:50,955 - simple_logger - CRITICAL - critical message
```

二、使用配置文件和fileConfig()函数实现日志配置

现在我们通过配置文件的方式来实现与上面同样的功能：

```
# 读取日志配置文件内容
logging.config.fileConfig('logging.conf')

# 创建一个日志器logger
logger = logging.getLogger('simpleExample')

# 日志输出
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

配置文件`logging.conf`内容如下：

```
[loggers]
keys=root,simpleExample

[handlers]
keys=fileHandler,consoleHandler

[formatters]
keys=simpleFormatter

[logger_root]
level=DEBUG
handlers=fileHandler

[logger_simpleExample]
level=DEBUG
handlers=consoleHandler
```

```
qualname=simpleExample
propagate=0

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

[handler_fileHandler]
class=FileHandler
args=('logging.log', 'a')
level=ERROR
formatter=simpleFormatter

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

运行输出：

```
2017-05-15 11:32:16,539 - simpleExample - DEBUG - debug message
2017-05-15 11:32:16,555 - simpleExample - INFO - info message
2017-05-15 11:32:16,555 - simpleExample - WARNING - warn message
2017-05-15 11:32:16,555 - simpleExample - ERROR - error message
2017-05-15 11:32:16,555 - simpleExample - CRITICAL - critical message
```

1. 关于`fileConfig()`函数的说明：

该函数实际上是对`configparser`模块的封装，关于`configparser`模块的介绍请参考≤。

函数定义：

该函数定义在loging.config模块下：

```
logging.config.fileConfig(fname, defaults=None, disable_existing_loggers=True)
```

参数：

- fname：表示配置文件的文件名或文件对象
- defaults：指定传给ConfigParser的默认值
- disable_existing_loggers：这是一个布尔型值，默认值为True（为了向后兼容）表示禁用已经存在的logger，除非它们或者它们的祖先明确的出现在日志配置中；如果值为False则对已存在的loggers保持启动状态。

2. 配置文件格式说明：

上面提到过，`fileConfig()`函数是对`ConfigParser/configparser`模块的封装，也就是说`fileConfig()`函数是基于`ConfigParser/configparser`模块来理解日志配置文件的。换句话说，`fileConfig()`函数所能理解的配置文件基础格式是与`ConfigParser/configparser`模块一致的，只是在此基础上对文件中包含的`section`和`option`做了一下规定和限制，比如：

- 1）配置文件中一定要包含loggers、handlers、formatters这些section，它们通过keys这个option来指定该配置文件中已经定义好的loggers、handlers和formatters，多个值之间用逗号分隔；另外loggers这个section中的keys一定要包含root这个值；
- 2）loggers、handlers、formatters中所指定的日志器、处理器和格式器都需要在下面以单独的section进行定义。seciton的命名规则为[logger_loggerName]、[formatter_formatterName]、[handler_handlerName]
- 3）定义logger的section必须指定level和handlers这两个option，level的可取值为DEBUG、INFO、WARNING、ERROR、CRITICAL、NOTSET，其中NOTSET表示所有级别的日志消息都要记录，包括用户定义级别；handlers的值是以逗号分隔的handler名字列表，这里出现的handler必须出现在[handlers]这个section中，并且相应的handler必须在配置文件中有对应的section定义；
- 4）对于非root logger来说，除了level和handlers这两个option之外，还需要一些额外的option，其中qualname是必须提供的option，它表示在logger层级中的名字，在应用代码中通过这个名字得到logger；propagate是可选项，其默认是为1，表示消息将会传递给高层次logger的handler，通常我们需要指定其值为0，这个可以看下下面的例子；另外，对于非root logger的level如果设置为NOTSET，系统将会查找高层次的logger来决定此logger的有效level。
- 5）定义handler的section中必须指定class和args这两个option，level和formatter为可选option；class表示用于创建handler的类名，args表示传递给class所指定的handler类初始化方法参数

，它必须是一个元组（tuple）的形式，即便只有一个参数值也需要是一个元组的形式；level与logger中的level一样，而formatter指定的是该处理器所使用的格式器，这里指定的格式器名称必须出现在formatters这个section中，且在配置文件中必须要有这个formatter的section定义；如果不指定formatter则该handler将会以消息本身作为日志消息进行记录，而不添加额外的时间、日志器名称等信息；

- 6）定义formatter的sectioin中的option都是可选的，其中包括format用于指定格式字符串，默认为消息字符串本身；datefmt用于指定asctime的时间格式，默认为'%Y-%m-%d %H:%M:%S'；class用于指定格式器类名，默认为logging.Formatter；

说明：

配置文件中的class指定类名时，该类名可以是相对于logging模块的相对值，如：FileHandler、handlers.TimeRotatingFileHandler；也可以是一个绝对路径值，通过普通的import机制来解析，如自定义的handler类mypackage.mymodule.MyHandler，但是mypackage需要在Python可用的导入路径中--sys.path。

3. 对于propagate属性的说明

实例1：

我们把logging.conf中simpleExample这个handler定义中的propagate属性值改为1，或者删除这个option（默认值就是1）：

```
[logger_simpleExample]
level=DEBUG
handlers=consoleHandler
```

```
qualname=simpleExample
propagate=1
```

现在来执行同样的代码：

```
# 读取日志配置文件内容
logging.config.fileConfig('logging.conf')

# 创建一个日志器logger
logger = logging.getLogger('simpleExample')

# 日志输出
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

我们会发现，除了在控制台有输出信息时候，在logging.log文件中也有内容输出：

```
2017-05-15 16:06:25,366 - simpleExample - ERROR - error message
2017-05-15 16:06:25,367 - simpleExample - CRITICAL - critical message
```

这说明simpleExample这个logger在处理完日志记录后，把日志记录传递给了上级的root logger再次做处理，所有才会有两个地方都有日志记录的输出。通常，我们都需要显示的指定propagate的值为0，防止日志记录向上层logger传递。

实例2：

现在，我们试着用一个没有在配置文件中定义的logger名称来获取logger：

```
# 读取日志配置文件内容
logging.config.fileConfig('logging.conf')

# 用一个没有在配置文件中定义的logger名称来创建一个日志器logger
logger = logging.getLogger('simpleExample1')

# 日志输出
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

运行程序后，我们会发现控制台没有任何输出，而logging.log文件中又多了两行输出：

```
2017-05-15 16:13:16,810 - simpleExample1 - ERROR - error message
2017-05-15 16:13:16,810 - simpleExample1 - CRITICAL - critical message
```

这是因为，当一个日志器没有被设置任何处理器是，系统会去查找该日志器的上层日志器上所设置的日志处理器来处理日志记录。simpleExample1在配置文件中没有被定义，因此`logging.getLogger(simpleExample1)`这行代码这是获取了一个logger实例，并没有给它设置任何处

理器，但是它的上级日志器--root logger在配置文件中有定义且设置了一个FileHandler处理器，simpleExample1处理器最终通过这个FileHandler处理器将日志记录输出到logging.log文件中了。

三、使用字典配置信息和dictConfig()函数实现日志配置

Python 3.2中引入的一种新的配置日志记录的方法--用字典来保存logging配置信息。这相对于上面所讲的基于配置文件来保存logging配置信息的方式来说，功能更加强大，也更加灵活，因为我们可把很多的数据转换成字典。比如，我们可以使用JSON格式的配置文件、YAML格式的配置文件，然后将它们填充到一个配置字典中；或者，我们也可以用Python代码构建这个配置字典，或者通过socket接收pickled序列化后的配置信息。总之，你可以使用你的应用程序可以操作的任何方法来构建这个配置字典。

这个例子中，我们将使用YAML格式来完成与上面同样的日志配置。

首先需要安装PyYAML模块：

```
pip install PyYAML
```

Python代码：

```
import logging
import logging.config
import yaml

with open('logging.yml', 'r') as f_conf:
    dict_conf = yaml.load(f_conf)
logging.config.dictConfig(dict_conf)

logger = logging.getLogger('simpleExample')
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

logging.yml配置文件的内容：

```
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
  console_err:
    class: logging.StreamHandler
    level: ERROR
```

```
    formatter: simple
    stream: ext://sys.stderr
loggers:
  simpleExample:
    level: DEBUG
    handlers: [console]
    propagate: yes
root:
  level: DEBUG
  handlers: [console_err]
```

输出结果：

```
2017-05-21 14:19:31,089 - simpleExample - DEBUG - debug message
2017-05-21 14:19:31,089 - simpleExample - INFO - info message
2017-05-21 14:19:31,089 - simpleExample - WARNING - warn message
2017-05-21 14:19:31,089 - simpleExample - ERROR - error message
2017-05-21 14:19:31,090 - simpleExample - CRITICAL - critical message
```

1. 关于`dictConfig()`函数的说明：

该函数实际上是对`configparser`模块的封装，关于`configparser`模块的介绍请参考≤。

函数定义：

该函数定义在loging.config模块下：

```
logging.config.dictConfig(config)
```

该函数可以从一个字典对象中获取日志配置信息，config参数就是这个字典对象。关于这个字典对象的内容规则会在下面进行描述。

2. 配置字典说明

无论是上面提到的配置文件，还是这里的配置字典，它们都要描述出日志配置所需要创建的各种对象以及这些对象之间的关联关系。比如，可以先创建一个名额为“simple”的格式器formatter；然后创建一个名为“console”的处理器handler，并指定该handler输出日志所使用的格式器为"simple"；然后再创建一个日志器logger，并指定它所使用的处理器为"console"。

传递给dictConfig()函数的字典对象只能包含下面这些keys，其中`version`是必须指定的key，其它key都是可选项：

key名称	描述
version	必选项，其值是一个整数值，表示配置格式的版本，当前唯一可用的值就是1
formatters	可选项，其值是一个字典对象，该字典对象每个元素的key为要定义的格式器名称，value为格式器的配置信息组成的dict，如format和datefmt
filters	可选项，其值是一个字典对象，该字典对象每个元素的key为要定义的过滤器名称，value为过滤器的配置信息组成的dict，如name
handlers	可选项，其值是一个字典对象，该字典对象每个元素的key为要定义的处理器名称，value为处理器的配置信息组成的dcit，如class、level、formatter和filters，其中class为必选项，其它为可选项；其他配置信息将会传递给class所指定的处理器类的构造函数，如下面的handlers定义示例中的stream、filename、maxBytes和backupCount等

loggers	可选项，其值是一个字典对象，该字典对象每个元素的key为要定义的日志器名称，value为日志器的配置信息组成的dcit，如level、handlers、filters 和 propagate（yes
root	可选项，这是root logger的配置信息，其值也是一个字典对象。除非在定义其它logger时明确指定propagate值为no，否则root logger定义的handlers都会被作用到其它logger上
incremental	可选项，默认值为False。该选项的意义在于，如果这里定义的对象已经存在，那么这里对这些对象的定义是否应用到已存在的对象上。值为False表示，已存在的对象将会被重新定义。
disable_existing_loggers	可选项，默认值为True。该选项用于指定是否禁用已存在的日志器loggers，如果incremental的值为True则该选项将会被忽略

handlers定义示例：

```
handlers:
  console:
    class : logging.StreamHandler
    formatter: brief
    level   : INFO
    filters: [allow_foo]
    stream  : ext://sys.stdout
  file:
    class : logging.handlers.RotatingFileHandler
    formatter: precise
    filename: logconfig.log
    maxBytes: 1024
    backupCount: 3
```

3. 关于外部对象的访问

需要说明的是，上面所使用的对象并不限于loggging模块所提供的对象，我们可以实现自己的formatter或handler类。另外，这些类的参数也许需要包含sys.stderr这样的外部对象。如果配置字典对象是使用Python代码构造的，可以直接使用sys.stdout、sys.stderr；但是当通过文本文件（如JSON、YAML格式的配置文件）提供配置时就会出现问题，因为在文本文件中，没有标准的方法来区分`sys.stderr`和字符串`'sys.stderr'`。为了区分它们，配置系统会在字符串值中查找特定的前缀，例如'ext://sys.stderr'中'ext://'会被移除，然后`import sys.stderr`。

- 参考文档：https://docs.python.org/3.5/library/logging.config.html

Python之向日志输出中添加上下文信息

除了传递给日志记录函数的参数（如msg）外，有时候我们还想在日志输出中包含一些额外的上下文信息。比如，在一个网络应用中，可能希望在日志中记录客户端的特定信息，如：远程客户端的IP地址和用户名。这里我们来介绍以下几种实现方式：

- 通过向日志记录函数传递一个extra参数引入上下文信息
- 使用LoggerAdapters引入上下文信息
- 使用Filters引入上下文信息

一、通过向日志记录函数传递一个extra参数引入上下文信息

前面我们提到过，可以通过向日志记录函数传递一个extra参数来实现向日志输出中添加额外的上下文信息，如：

```
import logging
import sys

fmt = logging.Formatter("%(asctime)s - %(name)s - %(ip)s - %(username)s - %(message)s")
h_console = logging.StreamHandler(sys.stdout)
h_console.setFormatter(fmt)
logger = logging.getLogger("myPro")
logger.setLevel(logging.DEBUG)
logger.addHandler(h_console)

extra_dict = {"ip": "113.208.78.29", "username": "Petter"}
logger.debug("User Login!", extra=extra_dict)

extra_dict = {"ip": "223.190.65.139", "username": "Jerry"}
logger.info("User Access!", extra=extra_dict)
```

输出：

```
2017-05-14 15:47:25,562 - myPro - 113.208.78.29 - Petter - User Login!
2017-05-14 15:47:25,562 - myPro - 223.190.65.139 - Jerry - User Access!
```

但是用这种方式来传递信息并不是那么方便，因为每次调用日志记录方法都要传递一个extra关键词参数。即便没有需要插入的上下文信息也是如此，因为该logger设置的formatter格式中指定的字段必须要存在。所以，我们推荐使用下面两种方式来实现上下文信息的引入。

> 也许可以尝试创建许多不同的Logger实例来解决上面存在的问题，但是这显然不是一个好的解决方案，因为这些Logger实例并不会进行垃圾回收。尽管这在实践中不是个问题，但是当Logger数量变得不可控将会非常难以管理。

二、使用LoggerAdapters引入上下文信息

使用`LoggerAdapter`类来传递上下文信息到日志事件的信息中是一个非常简单的方式，可以把它看做第一种实现方式的优化版--因为它为extra提供了一个默认值。这个类设计的类似于`Logger`，因此我们可以像使用Logger类的实例那样来调用`debug()`，`info()`，`warning()`,`error()`，`exception()`，`critical()`和`log()`方法。当创建一个LoggerAdapter的实例时，我们需要传递一个Logger实例和一个包含上下文信息的类字典对象给该类的实例构建方法。当调用LoggerAdapter实例的一个日志记录方法时，该方法会在对日志日志消息和字典对象进行处理后，调用构建该实例时传递给该实例的logger对象的同名的日志记录方法。下面是LoggerAdapter类中几个方法的定义：

```
class LoggerAdapter(object):
    """
```

```
    An adapter for loggers which makes it easier to specify contextual
    information in logging output.
    """

    def __init__(self, logger, extra):
        """
        Initialize the adapter with a logger and a dict-like object which
        provides contextual information. This constructor signature allows
        easy stacking of LoggerAdapters, if so desired.

        You can effectively pass keyword arguments as shown in the
        following example:

        adapter = LoggerAdapter(someLogger, dict(p1=v1, p2="v2"))
        """
        self.logger = logger
        self.extra = extra

    def process(self, msg, kwargs):
        """
        Process the logging message and keyword arguments passed in to
        a logging call to insert contextual information. You can either
        manipulate the message itself, the keyword args or both. Return
        the message and kwargs modified (or not) to suit your needs.

        Normally, you'll only need to override this one method in a
        LoggerAdapter subclass for your specific needs.
        """
        kwargs["extra"] = self.extra
        return msg, kwargs

    def debug(self, msg, *args, **kwargs):
        """
        Delegate a debug call to the underlying logger, after adding
        contextual information from this adapter instance.
        """
        msg, kwargs = self.process(msg, kwargs)
        self.logger.debug(msg, *args, **kwargs)
```

通过分析上面的代码可以得出以下结论：

- 上下文信息是在LoggerAdapter类的process()方法中被添加到日志记录的输出消息中的，如果要实现自定义需求只需要实现LoggerAdapter的子类并重写process()方法即可；

- process()方法的默认实现中，没有修改msg的值，只是为关键词参数插入了一个名为`extra`的 key，这个extra的值为传递给LoggerAdapter类构造方法的参数值；
- LoggerAdapter类构建方法所接收的extra参数，实际上就是为了满足logger的formatter格式要求所提供的默认上下文信息。

关于上面提到的第3个结论，我们来看个例子：

```
import logging
import sys


# 初始化一个要传递给LoggerAdapter构造方法的logger实例
fmt = logging.Formatter("%(asctime)s - %(name)s - %(ip)s - %(username)s - %
(message)s")
h_console = logging.StreamHandler(sys.stdout)
h_console.setFormatter(fmt)
init_logger = logging.getLogger("myPro")
init_logger.setLevel(logging.DEBUG)
init_logger.addHandler(h_console)

# 初始化一个要传递给LoggerAdapter构造方法的上下文字典对象
extra_dict = {"ip": "IP", "username": "USERNAME"}

# 获取一个LoggerAdapter类的实例
logger = logging.LoggerAdapter(init_logger, extra_dict)

# 应用中的日志记录方法调用
logger.info("User Login!")
logger.info("User Login!", extra={"ip": "113.208.78.29", "username": "Petter"})
logger.extra = {"ip": "113.208.78.29", "username": "Petter"}
logger.info("User Login!")
logger.info("User Login!")
```

输出结果：

```
# 使用extra默认值：{"ip": "IP", "username": "USERNAME"}
2017-05-14 17:23:15,148 - myPro - IP - USERNAME - User Login!

# info(msg, extra)方法中传递的extra方法没有覆盖默认值
2017-05-14 17:23:15,148 - myPro - IP - USERNAME - User Login!

# extra默认值被修改了
2017-05-14 17:23:15,148 - myPro - 113.208.78.29 - Petter - User Login!
2017-05-14 17:23:15,148 - myPro - 113.208.78.29 - Petter - User Login!
```

根据上面的程序输出结果，我们会发现一个问题：传递给LoggerAdapter类构造方法的extra参数值不能被LoggerAdapter实例的日志记录函数（如上面调用的info()方法）中的extra参数覆盖，只能通过修改LoggerAdapter实例的extra属性来修改默认值（如上面使用的logger.extra=xxx），但是这也就意味着默认值被修改了。

解决这个问题的思路应该是：实现一个LoggerAdapter的子类，重写process()方法。其中对于kwargs参数的操作应该是先判断其本身是否包含extra关键字，如果包含则不使用默认值进行替换；如果kwargs参数中不包含extra关键字则取默认值。来看具体实现：

```
import logging
import sys

class MyLoggerAdapter(logging.LoggerAdapter):

    def process(self, msg, kwargs):
        if 'extra' not in kwargs:
            kwargs["extra"] = self.extra
        return msg, kwargs

if __name__ == '__main__':
    # 初始化一个要传递给LoggerAdapter构造方法的logger实例
    fmt = logging.Formatter("%(asctime)s - %(name)s - %(ip)s - %(username)s - %
(message)s")
    h_console = logging.StreamHandler(sys.stdout)
    h_console.setFormatter(fmt)
    init_logger = logging.getLogger("myPro")
    init_logger.setLevel(logging.DEBUG)
    init_logger.addHandler(h_console)

    # 初始化一个要传递给LoggerAdapter构造方法的上下文字典对象
    extra_dict = {"ip": "IP", "username": "USERNAME"}

    # 获取一个自定义LoggerAdapter类的实例
    logger = MyLoggerAdapter(init_logger, extra_dict)

    # 应用中的日志记录方法调用
    logger.info("User Login!")
    logger.info("User Login!", extra={"ip": "113.208.78.29", "username":
"Petter"})
    logger.info("User Login!")
    logger.info("User Login!")
```

输出结果：

```
# 使用extra默认值: {"ip": "IP", "username": "USERNAME"}
2017-05-22 17:35:38,499 - myPro - IP - USERNAME - User Login!

# info(msg, extra)方法中传递的extra方法已覆盖默认值
2017-05-22 17:35:38,499 - myPro - 113.208.78.29 - Petter - User Login!

# extra默认值保持不变
2017-05-22 17:35:38,499 - myPro - IP - USERNAME - User Login!
2017-05-22 17:35:38,499 - myPro - IP - USERNAME - User Login!
```

OK！ 问题解决了。

其实，如果我们想不受formatter的限制，在日志输出中实现自由的字段插入，可以通过在自定义LoggerAdapter的子类的process()方法中将字典参数中的关键字信息拼接到日志事件的消息中。很明显，这些上下文中的字段信息在日志输出中的位置是有限制的。而使用'extra'的优势在于，这个类字典对象的值将被合并到这个LogRecord实例的__dict__中，这样就允许我们通过Formatter实例自定义日志输出的格式字符串。这虽然使得上下文信息中的字段信息在日志输出中的位置变得与内置字段一样灵活，但是前提是传递给构造器方法的这个类字典对象中的key必须是确定且明了的。

三、使用Filters引入上下文信息

另外，我们还可以使用自定义的`Filter.Filter`实例的方式，在filter(record)方法中修改传递过来的LogRecord实例，把要加入的上下文信息作为新的属性赋值给该实例，这样就可以通过指定formatter的字符串格式来输出这些上下文信息了。

我们模仿上面的实现，在传递个filter(record)方法的LogRecord实例中添加两个与当前网络请求相关的信息：ip和username。

```
import logging
from random import choice


class ContextFilter(logging.Filter):

        ip = 'IP'
        username = 'USER'

        def filter(self, record):
            record.ip = self.ip
            record.username = self.username
            return True

if __name__ == '__main__':
    levels = (logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR,
logging.CRITICAL)
    users = ['Tom', 'Jerry', 'Peter']
```

```
    ips = ['113.108.98.34', '219.238.78.91', '43.123.99.68']

    logging.basicConfig(level=logging.DEBUG,
                        format='%(asctime)-15s %(name)-5s %(levelname)-8s %
(ip)-15s %(username)-8s %(message)s')
    logger = logging.getLogger('myLogger')
    filter = ContextFilter()
    logger.addFilter(filter)
    logger.debug('A debug message')
    logger.info('An info message with %s', 'some parameters')

    for x in range(5):
        lvl = choice(levels)
        lvlname = logging.getLevelName(lvl)
        filter.ip = choice(ips)
        filter.username = choice(users)
        logger.log(lvl, 'A message at %s level with %d %s' , lvlname, 2,
'parameters')
```

输出结果：

```
2017-05-15 10:21:49,401 myLogger DEBUG    IP              USER     A debug
message
2017-05-15 10:21:49,401 myLogger INFO     IP              USER     An info
message with some parameters
2017-05-15 10:21:49,401 myLogger INFO     219.238.78.91   Tom      A message at
INFO level with 2 parameters
2017-05-15 10:21:49,401 myLogger INFO     219.238.78.91   Peter    A message at
INFO level with 2 parameters
2017-05-15 10:21:49,401 myLogger DEBUG    113.108.98.34   Jerry    A message at
DEBUG level with 2 parameters
2017-05-15 10:21:49,401 myLogger CRITICAL 43.123.99.68    Tom      A message at
CRITICAL level with 2 parameters
2017-05-15 10:21:49,401 myLogger INFO     43.123.99.68    Jerry    A message at
INFO level with 2 parameters
```

需要说明的是： 实际的网络应用程序中，可能还要考虑多线程并发时的线程安全问题，此时可以把连接信息或者自定义过滤器的实例通过threading.local保存到到一个threadlocal中。

- 参考文档：https://docs.python.org/3.5/howto/logging-cookbook.html#adding-contextual-information-to-your-logging-output