logging模块

日志级别

日志级别Level	数值
CRITICAL	50
ERROR	40
WARNING	30，**默认级别**
INFO	20
DEBUG	10
NOTSET	0

日志级别指的是产生日志的事件的严重程度。
设置一个级别后，严重程度低于设置值的日志消息将被忽略。

debug()、 info()、 warning()、 error() 和 critical()方法。

格式字符串

属性名	格式	描述
日志消息内容	%(message)s	The logged message, computed as msg % args. 当调用Formatter.format()时设置
asctime	%(asctime)s	创建LogRecord时的可读时间。默认情况下，它的格式为'2003-07-08 16:49:45,896'（逗号后面的数字是毫秒部分的时间）
函数名	%(funcName)s	日志调用所在的函数名
日志级别名称	%(levelname)s	消息的级别名称 'DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL'
日记级别数值	%(levelno)s	消息的级别数字，对应DEBUG, INFO, WARNING, ERROR, CRITICAL
行号	%(lineno)d	日志调用所在的源码行号
模块	%(module)s	模块（filename的名字部分）
进程ID	%(process)d	进程 ID
线程ID	%(thread)d	线程 ID
进程名称	%(processName)s	进程名
线程名称	%(threadName)s	线程名字

注意：funcName、threadName、processName都是小驼峰。

举例

默认级别

```
import logging

FORMAT = '%(asctime)-15s\tThread info: %(thread)d  %(threadName)s %(message)s'
logging.basicConfig(format=FORMAT)

logging.info('I am {}'.format(20)) # info不显示
logging.warning('I am {}'.format(20)) # warning默认级别
```

构建消息

```
import logging

FORMAT = '%(asctime)-15s\tThread info: %(thread)d  %(threadName)s %(message)s'
logging.basicConfig(format=FORMAT, level=logging.INFO)

logging.info('I am {}'.format(20)) # 单一字符串
logging.info('I am %d %s', 20, 'years old.') # c风格
```

上例是基本的使用方法，大多数时候，使用的是info，正常运行信息的输出

日志级别和格式字符串扩展的例子

```
FORMAT = '%(asctime)-15s\tThread info: %(thread)d  %(threadName)s %(message)s %(school)s'
logging.basicConfig(format=FORMAT, level=logging.WARNING)

d = {'school': 'magedu.com'}
logging.info('I am {}'.format(20), extra=d)
logging.warning('I am %s %s', 20, 'years old.', extra = d)
```

修改日期格式

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

输出到文件

```
import logging
logging.basicConfig(format='%(asctime)s %(message)s', filename='o:/test.log')
for _ in range(5):
    logging.warning('this event was logged.')
```

filename设置日志文件；filemode设置读写模式

Logger类

在logging模块中，顶层代码中有

```
root = RootLogger(WARNING) # 大约在1731行处。指定根Logger对象的默认级别。就在basicConfig函数上面
Logger.root = root # 为类Logger增加类属性root
```

logging模块加载的时候，会创建一个全局对象root，它是一个RootLogger实例，即root logger。根Logger对象的默认级别是WARNING。

RootLogger类继承自Logger类
类Logger初始化方法签名是(name, level=0)
类RootLogger初始化方法签名(level)，本质上调用的是`Logger.__init__(self, "root", WARNING)`

调用logging.basicConfig来调整级别，就是对这个根Logger的级别进行修改。

构造

Logger实例的构建，使用Logger类也行，但推荐getLogger函数。

```
Logger.manager = Manager(Logger.root) # 1733行，为Logger类注入一个manager类属性

def getLogger(name=None):
    """
    Return a logger with the specified name, creating it if necessary.
    If no name is specified, return the root logger.
    """
    if name:
        return Logger.manager.getLogger(name)
    else:
        return root # 没有指定名称，返回根logger
```

使用工厂方法返回一个Logger实例。
指定name，返回一个名称为name的Logger实例。如果再次使用相同的名字，返回同一个实例。背后使用一个字典保证同一个名称返回同一个Logger实例。
未指定name，返回根Logger实例。

```
import logging

a = logging.Logger('hello', 20)
b = logging.Logger('hello', 30)
print(a, id(a))
print(b, id(b))

c = logging.getLogger('hello')
print(c, id(c))
d = logging.getLogger('hello')
print(d, id(d))
```

层次结构

Logger是层次结构的，使用 . 点号分割，如'a'、'a.b'或'a.b.c.d'，a是a.b的父parent，a.b是a的子child。对于foo来说，名字为foo.bar、foo.bar.baz、foo.bam都是 foo的后代。

```
import logging

# 父子 层次关系
# 根logger
root = logging.root
print(root, id(root))
```

```
root = logging.getLogger()
print(root, id(root))
print(root.name, type(root), root.parent) # 根logger没有父

parent = logging.getLogger(__name__) # 模块级logger
print(parent.name, type(parent), id(parent.parent), id(parent))

child = logging.getLogger("{}{}".format(__name__, '.child')) # 子logger
print(child.name, type(child), id(child.parent), id(child))
```

Level级别设置

每一个logger实例都有级别

```
import logging

FORMAT = '%(asctime)-15s\tThread info: %(thread)d  %(threadName)s [%(message)s]'
logging.basicConfig(format=FORMAT, level=logging.INFO)

logger = logging.getLogger(__name__)
print(logger.name, type(logger), logger.level) # 新的logger实例的level是什么?
logger.info('1 info')
print(logger.getEffectiveLevel()) # 等效级别，从哪里来的

logger.setLevel(28) # 设置logger的level
print(logger.getEffectiveLevel(), logger.level)
logger.info('2 info')

logger.setLevel(42)
logger.warning('3 warning')
logger.error('4 error')
logger.critical('5 critical')

root = logging.getLogger() # 根logger
root.info('6 root info') # 不影响
```

每一个logger实例，都有一个等效的level。
logger对象可以在创建后动态的修改自己的level。
等效level决定着logger实例能输出什么级别信息。

Handler

Handler 控制日志信息的输出目的地，可以是控制台、文件。

- 可以单独设置level
- 可以单独设置格式
- 可以设置过滤器

Handler类层次

- Handler
  - StreamHandler # 不指定使用sys.stderr
    - FileHandler # 文件
    - _StderrHandler # 标准输出
  - NullHandler # 什么都不做

日志输出其实是Handler做的，也就是真正干活的是Handler。

在logging.basicConfig函数中，如下：

```
if handlers is None:
    filename = kwargs.pop("filename", None)
    mode = kwargs.pop("filemode", 'a')
    if filename:
        h = FileHandler(filename, mode)
    else:
        stream = kwargs.pop("stream", None)
        h = StreamHandler(stream)
    handlers = [h]
```

如果设置文件名，则为根Logger加一个输出到文件的FileHandler；如果没有设置文件名，则为根Logger加一个StreamHandler，默认输出到sys.stderr。
也就是说，根logger一定会至少有一个handler的。

思考
创建的Handler的初始的level是什么?

```
import logging

FORMAT = '%(asctime)s %(name)s %(message)s'
logging.basicConfig(format=FORMAT, level=logging.INFO)

logger = logging.getLogger('test')
print(logger.name, type(logger))

logger.info('line 1')

handler = logging.FileHandler('o:/l1.log','w') # 创建handler
logger.addHandler(handler) # 给logger对象绑定一个handler

# 注意看控制台，再看l1.log文件，对比差异
# 思考这是怎么打印的
logger.info('line 2')
```

Handler的初始的level是什么？是0

日志流

level的继承

```
import logging

logging.basicConfig(format="%(asctime)s %(name)s [%(message)s]")

log1 = logging.getLogger('s')
print(log1.level, log1.getEffectiveLevel())
log1.info('1 info')

log2 = logging.getLogger('s.s1')
print(log2.level, log2.getEffectiveLevel())
log2.info('2 info')
print('-' * 30)

log1.setLevel(20)
log1.info('3 info')
print(log1.level, log1.getEffectiveLevel())
print(log2.level, log2.getEffectiveLevel())
log2.info('4 info')

log2.setLevel(30)
print(log2.level, log2.getEffectiveLevel())
log2.info('5 info')
```

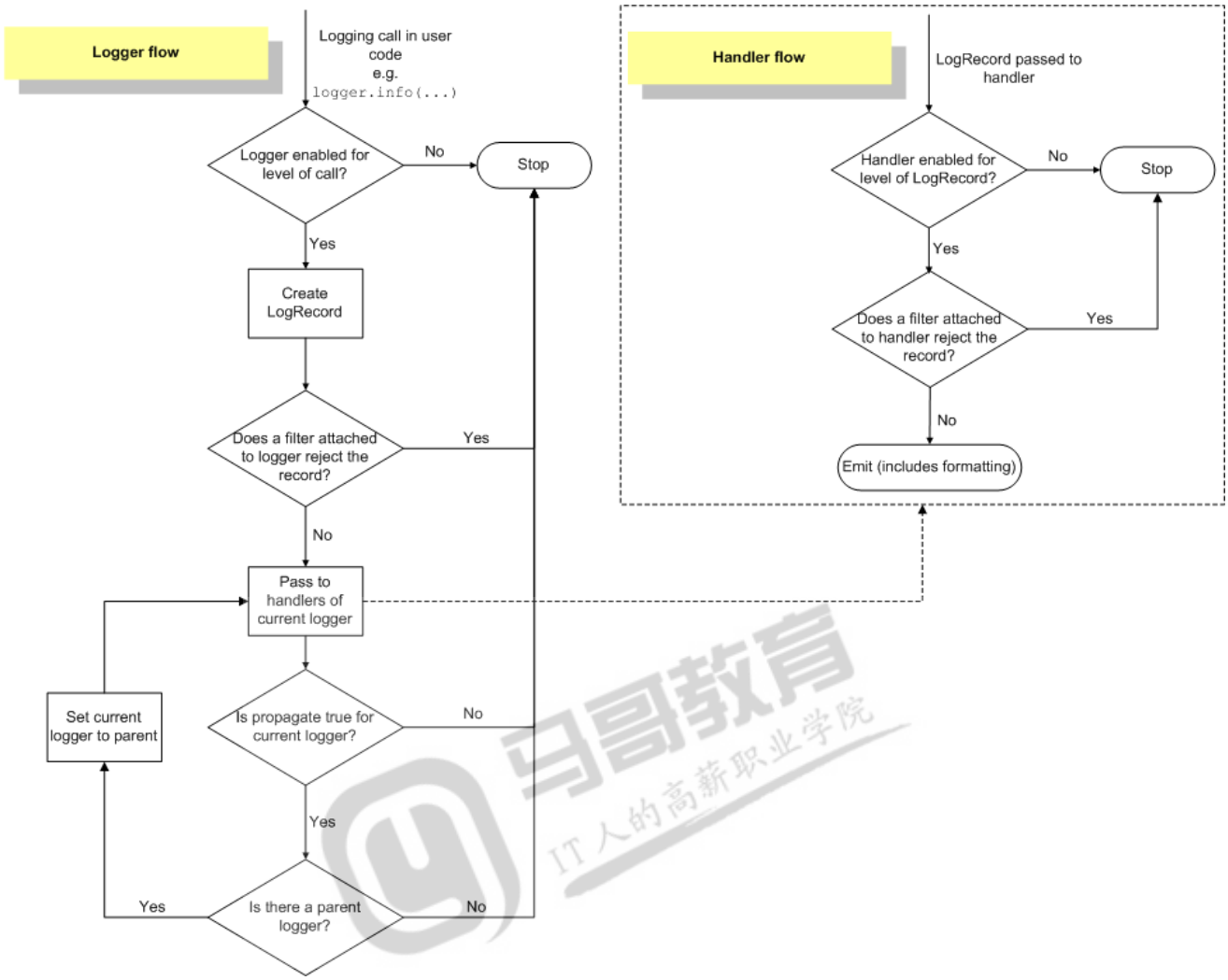
logger实例
如果不设置level，则初始level为0。
如果设置了level，就优先使用自己的level，否则，继承最近的祖先的level。
信息是否能够从该logger实例上输出，就是要看当前函数的level是否大于等于logger的有效level，否则输出不了。

继承关系及信息传递

- 每一个Logger实例的level如同入口，让水流进来，如果这个门槛太高，信息就进不来。例如log3.warning('log3')，如果log3定义的级别高，就不会有信息通过log3
- 如果level没有设置，就用父logger的，如果父logger的level没有设置，继续找父的父的，最终可以找到root上，如果root设置了就用它的，如果root没有设置，root的默认值是WARNING
- 消息传递流程
    1. 如果消息在某一个logger对象上产生，这个logger就是**当前logger**，首先消息level要和当前logger的EffectiveLevel比较，如果低于当前logger的EffectiveLevel，则流程结束；否则生成log记录。
    2. 日志记录会交给当前logger的所有handler处理，记录还要和每一个handler的级别分别比较，低的不处理，否则按照handler输出日志记录。
    3. **当前logger**的所有handler处理完后，就要看自己的propagate属性，如果是True表示向父logger传递这个日志记录，否则到此流程结束。
    4. 如果日志记录传递到了父logger，不需要和父logger的level比较，而是直接交给父的所有handler，父logger成为**当前logger**。重复2、3步骤，直到当前logger的父logger是None退出，也就是说当前logger最后一般是root logger（是否能到root logger要看中间的logger是否允许propagate）。
- logger实例初始的propagate属性为True，即允许向父logger传递消息
- logging.basicConfig函数
  如果root没有handler，就默认创建一个StreamHandler，如果设置了filename，就创建一个FileHandler。
  如果设置了format参数，就会用它生成一个Formatter对象，否则会生成缺省Formatter，并把这个formatter加入到刚才创建的handler上，然后把这些handler加入到root.handlers列表上。level是设置给

root logger的。
如果root.handlers列表不为空，logging.basicConfig的调用什么都不做。

官方日志流转图



参考logging.Logger类的callHandlers方法

实例

```
import logging
logging.basicConfig(level=logging.INFO, format="%(asctime)s %(name)-10s [%(message)s]")

# 根logger的操作
root = logging.getLogger()
root.setLevel(logging.ERROR)
print('root', root.handlers)
# 增加handler
import sys
h0 = logging.StreamHandler(sys.stdout)
h0.setLevel(logging.WARNING)
root.addHandler(h0)
print('root', root.handlers)

for h in root.handlers:
```

```
    print('root handler = {}, formatter = {}'.format(h,h.formatter._fmt if h.formatter else ''))

logging.error('root test~~~~~~~')

# logger s
log1 = logging.getLogger('s')
log1.setLevel(logging.ERROR)
h1 = logging.FileHandler('o:/test.log')
h1.setLevel(logging.WARNING)
log1.addHandler(h1)
print(log1.name, log1.handlers)
log1.warning('s test --------') # 能够打印?
print('-' * 30)

# logger s.s1
log2 = logging.getLogger('s.s1')
log2.setLevel(logging.CRITICAL)
h2 = logging.FileHandler('o:/test.log')
h2.setLevel(logging.WARNING)
log2.addHandler(h2)
print(log2.name, log2.handlers)
log2.error('s.s1 test =======')
print('-' * 30)

# logger s.s1.s2
log3 = logging.getLogger('s.s1.s2')
log3.setLevel(logging.INFO) # 级别最低
print(log3.getEffectiveLevel())
log3.info('s.s1.s2 test info ++++')
print(log3.name, log3.handlers)
```

Formatter

logging的Formatter类，它允许指定某个格式的字符串。如果提供None，那么'%(message)s'将会作为默认值。

```
import logging
logging.basicConfig(level=logging.INFO, format="%(asctime)s %(name)-10s [%(message)s]")

# 根logger的操作
root = logging.getLogger()
root.setLevel(logging.ERROR)
print('root', root.handlers)
# 增加handler
import sys
h0 = logging.StreamHandler(sys.stdout)
h0.setLevel(logging.WARNING)
root.addHandler(h0)
print('root', root.handlers)

for h in root.handlers:
    print('root handler = {}, formatter = {}'.format(h,h.formatter._fmt if h.formatter else ''))
```

```
# 为h0增加一个Formatter
fmt0 = logging.Formatter("**** %(message)s ****")
h0.setFormatter(fmt0)

logging.error('root test~~~~~~~')
```

Filter

Filter分为2种：

- 可以为handler增加过滤器，所以这种过滤器只影响某一个handler，不会影响整个处理流程
- 如果过滤器增加到logger上，就会影响流程

```
import logging
logging.basicConfig(level=logging.INFO, format="%(asctime)s %(name)-10s [%(message)s]")

# logger s
log1 = logging.getLogger('s')
log1.setLevel(logging.WARNING)
h1 = logging.StreamHandler()
h1.setLevel(logging.INFO)
f1 = logging.Formatter('log1-h1 %(message)s')
h1.setFormatter(f1)
log1.addHandler(h1)

# logger s.s1
log2 = logging.getLogger('s.s1')
#log2.setLevel(logging.CRITICAL)
print(log2.getEffectiveLevel()) # 继承父logger，就是s的level
h2 = logging.StreamHandler()
h2.setLevel(logging.INFO)
f2 = logging.Formatter('log2-h2 %(message)s')
h2.setFormatter(f2)

# 增加Filter
filter = logging.Filter('s') # 过滤器
h2.addFilter(filter) # h2上增加了该过滤器
print(filter.name)
log2.addHandler(h2)

log2.warning('log2 test warning string~~~~~')

# logger s.s2
log3 = logging.getLogger('s.s1.s2')
print(log3.name, log3.getEffectiveLevel())
log3.warning('log3 test warning string======')
```

消息log2的，它的名字是s.s1，因此过滤器名字设置为s或s.s1，消息就可以通过，但是如果是其他就不能通过，不设置过滤器名字，所有消息通过

过滤器核心就这一句，在logging.Filter类的filter方法中
`record.name.find(self.name, 0, self.nlen) != 0`
本质上就是等价于`record.name.startswith(filter.name)`