logging模块：

标准库里面的logging模块，在前面学习线程安全时曾用来解决print被打断的问题，这里会介绍logging模块的功能。

logging模块是线程安全的，不需要客户做任何特殊的工作。它通过使用线程锁实现了这一点; 有一个锁来序列化访问模块的共享数据，每个处理程序还创建一个锁来序列化访问其底层 I/O。

日志记录级别：

| **级别** | **数值** |
| --- | --- |
| CRITICAL | 50 |
| ERROR | 40 |
| WARNING | 30,默认 |
| INFO | 20 |
| DEBUG | 10 |
| NOTSET | 0 |

定义的记录级别越低，信息越多，级别越高，信息越少。

日志记录格式化字符串：

| **属性名** | **格式** | **描述** |
| --- | --- | --- |
| asctime | %(asctime)s | 易读的时间格式： 默认情况下是'2003-07-08 16：49：45,896'的形式（逗号之后的数字是毫秒部分的时间） |
| filename | %(filename)s | 路径名的文件名部分。 |
| funcName | %(funcName)s | 日志调用所在的函数名 |
| levelname | %(levelname)s | 消息的级别名称('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL'). |
| levelno | %(levelno)s | 对应数字格式的日志级别 (DEBUG, INFO, WARNING, ERROR,CRITICAL). |
| lineno | %(lineno)d | 发出日志记录调用的源码行号 (如果可用)。 |
| module | %(module)s | 所在的模块名(如test6.py模块则记录test6) |
| message | %(message)s | 记录的信息 |
| name | %(name)s | 调用的logger记录器的名称 |
| process | %(process)d | 进程ID |
| processName | %(processName)s | 进程名 |
| thread | %(thread)d | 线程ID |
| threadName | %(threadName)s | 线程名 |

使用basicConfig方法配置logging记录格式：

| **格式** | **描述** |
| --- | --- |
| filename | 指定使用指定的文件名而不是StreamHandler创建FileHandler。 |
| filemode | 指定打开文件的模式，如果指定了filename（如果文件模式未指定，则默认为'a'）。 |
| format | 为处理程序使用指定的格式字符串。 |
| datefmt | 使用指定的日期/时间格式。 |
| level | 将根记录器级别设置为指定的级别。 |
| handlers | 如果指定，这应该是一个已经创建的处理程序的迭代器添加到根记录器。任何尚未设置格式化程序的处理程序都将被分配在此函数中创建的默认格式化程序。 |

举例：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | import threading  import logging  FORMAT = "%(asctime)s %(thread)d %(message)s"  logging.basicConfig(level=logging.INFO,format=FORMAT)    def add(x,y):      logging.warning("{} {}".format(threading.enumerate(),x+y))    t = threading.Timer(1,add,args=(4,5))  t.start()    运行结果：  2017-12-17 15:40:34,226 123145367023616 [<\_MainThread(MainThread, stopped 4320629568)>, <Timer(Thread-1, started 123145367023616)>] 9 |

修改日期格式：

|  |  |
| --- | --- |
| 1  2  3 | DATEFMT ="[%Y-%m-%d %H:%M:%S]"  FORMAT = "%(asctime)s %(thread)d %(message)s"  logging.basicConfig(level=logging.INFO,format=FORMAT,datefmt=DATEFMT) |

　　[2017-12-17 15:45:18]

输出到文件：

|  |  |
| --- | --- |
| 1 | logging.basicConfig(level=logging.INFO,format=FORMAT,datefmt=DATEFMT,filename='class\_test.log') |

　　文件路径不指定，默认为当前模块路径。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import threading  import logging  DATEFMT ="[%Y-%m-%d %H:%M:%S]"  FORMAT = "%(asctime)s %(thread)d %(message)s"  logging.basicConfig(level=logging.INFO,format=FORMAT,datefmt=DATEFMT,filename='class\_test.log')    def add(x,y):      logging.warning("{} {}".format(threading.enumerate(),x+y))    t = threading.Timer(1,add,args=(4,5))  t.start()    输出结果会追加写入当前模块路径的class\_test.log文件：  [2017-12-17 15:50:13] 123145503244288 [<\_MainThread(MainThread, stopped 4320629568)>, <Timer(Thread-1, started 123145503244288)>] 9 |

 Logger类：

构造

使用工厂方法返回一个Logger实例。

logging.getLogger([name=None])

 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，是实例化一个对象。未指定name，返回Logger实例，名称是root，即根Logger。

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。

举例：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | import logging  DATEFMT ="[%Y-%m-%d %H:%M:%S]"  FORMAT = "%(asctime)s %(thread)d %(message)s"  logging.basicConfig(level=logging.INFO,format=FORMAT,datefmt=DATEFMT,filename='class\_test.log')    root = logging.getLogger()  print(root.name,type(root),root.parent,id(root))    logger = logging.getLogger(\_\_name\_\_)  print(logger.name, type(logger), id(logger), id((logger.parent)))    logger1 = logging.getLogger(\_\_name\_\_ + ".ok")  print(logger1.name, type(logger1), id(logger1), id((logger1.parent)))    print(logger1.parent,id(logger1.parent))    运行结果：  root <class 'logging.RootLogger'> None 4367575248  \_\_main\_\_ <class 'logging.Logger'> 4367575864 4367575248  \_\_main\_\_.ok <class 'logging.Logger'> 4367575920 4367575864  <logging.Logger object at 0x10453eb38> 4367575864 |

子child的级别设置，不影响父parent的级别：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import logging    FORMAT = "%(asctime)s %(thread)d %(message)s"  logging.basicConfig(level=logging.WARNING,format=FORMAT,datefmt="[%Y-%m-%d %H:%M:%S]")    root = logging.getLogger()  print(1,root,id(root)) #RootLogger,根Logger  root.info('my root') #低于定义的WARNING级别，所以不会记录    loga = logging.getLogger(\_\_name\_\_) #Logger继承自RootLogger  print(2,loga,id(loga),id(loga.parent))  print(3,loga.getEffectiveLevel()) #数值形式的有效级别    loga.warning('before')  loga.setLevel(28) #设置级别为28  print(4,loga.getEffectiveLevel())  loga.info('after')#  loga.warning('after1')    运行结果：  [2017-12-17 16:31:20] 4320629568 before  1 <logging.RootLogger object at 0x104534f28> 4367535912  2 <logging.Logger object at 0x1044ef630> 4367250992 4367535912  3 30  4 28  [2017-12-17 16:31:20] 4320629568 after1 |

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

　　StreamHandler #不指定使用sys.strerr

　　　　FileHandler #文件

　　　　\_StderrHandler #标准输出

　　NullHandler #什么都不做

level的继承：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | import logging    FORMAT = "%(asctime)s %(thread)d %(message)s"  logging.basicConfig(level=logging.INFO,format=FORMAT,datefmt="[%Y-%m-%d %H:%M:%S]")    root = logging.getLogger() #根Logger级别为INFO 20  print('root:',root.getEffectiveLevel())    log1 = logging.getLogger('s')  log1.setLevel(logging.ERROR) #级别为ERROR 40  print('log1:',log1.getEffectiveLevel())  log1.error('log1 error')    log2 = logging.getLogger('s.s1') #继承自log1 40，无法使用warning  log2.setLevel(logging.WARNING) #设置为WARNING 30，才可以使用warning  print('log2:',log2.getEffectiveLevel())  log2.warning('log2 warning')    运行结果：  [2017-12-17 16:52:22] 4320629568 log1 error  root: 20  log1: 40  [2017-12-17 16:52:22] 4320629568 log2 warning  log2: 30 |

　　logger实例，如果设置了level，就用它和信息的级别比较，否则，继承最近的祖先的level。

handler处理：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | import logging    FORMAT = "%(asctime)s %(thread)d %(message)s"  logging.basicConfig(level=logging.INFO,format=FORMAT,datefmt="[%Y-%m-%d %H:%M:%S]")    root = logging.getLogger()  print(1,root.getEffectiveLevel()) #RootLogger,根Logger    log1 = logging.getLogger('s')  print(2,log1.getEffectiveLevel())    h1 = logging.FileHandler('test.log')  h1.setLevel(logging.WARNING)  log1.addHandler(h1)  print(3,log1.getEffectiveLevel())    log2 = logging.getLogger('s.s2')  print(4,log2.getEffectiveLevel())    h2 = logging.FileHandler('test1.log')  h2.setLevel(logging.WARNING)  log1.addHandler(h2)  print(3,log1.getEffectiveLevel())      log2.warning('log2 info---')      运行结果：  1 20  [2017-12-17 19:02:53] 7956 log2 info---  2 20  3 20  4 20  3 20 |

　　test.log和test1.log最终都会记录一份"log2 info---"

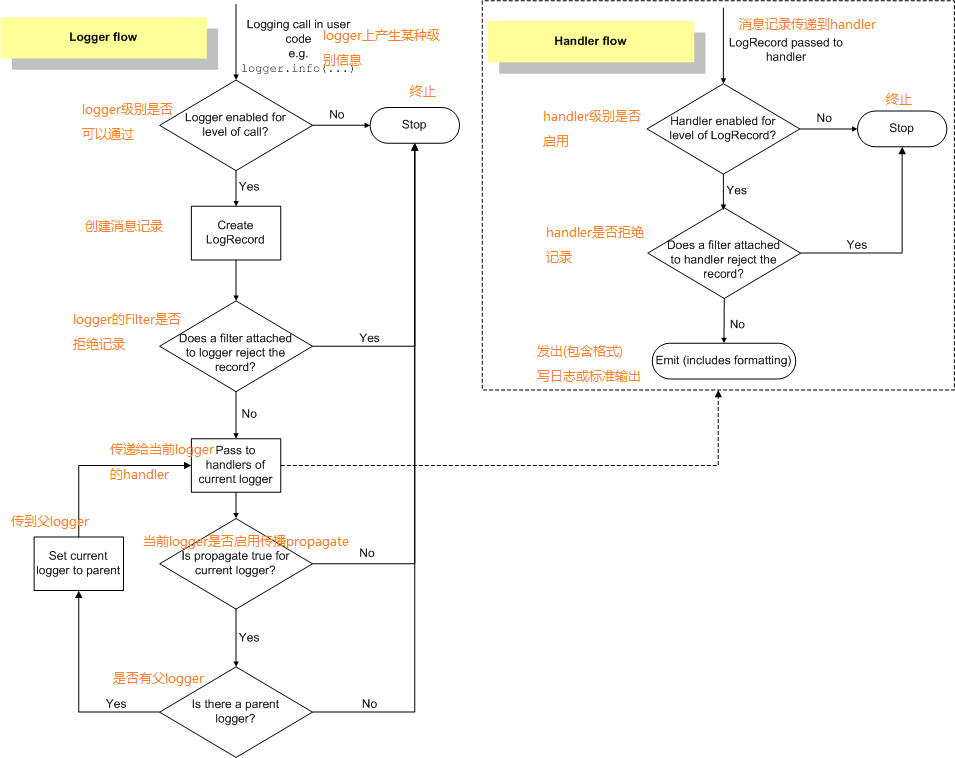
 同样，handler也可以设置使用logging.Formatter()设置格式和Logging.Filter()设置过滤器：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | import logging    FORMAT = "%(asctime)s %(thread)d %(message)s"  logging.basicConfig(level=logging.INFO,format=FORMAT,datefmt="[%Y-%m-%d %H:%M:%S]")    root = logging.getLogger()  print(1,root.getEffectiveLevel()) #RootLogger,根Logger    log1 = logging.getLogger('s')#模块化用\_\_module\_\_，函数化用\_\_name\_\_作为Logger名，Logger同名内存中也只有一个  print(2,log1.getEffectiveLevel())    h1 = logging.FileHandler('test.log')  h1.setLevel(logging.WARNING)  fmt1 = logging.Formatter('[%(asctime)s] %(thread)s %(threadName)s log1-handler1 %(message)s')  h1.setFormatter(fmt1) #重新个性化定义记录的格式化字符串  log1.addHandler(h1)  filter1 = logging.Filter('s') #过滤器 会记录s, s.s2的信息  log1.addFilter(filter1)  print(3,log1.getEffectiveLevel())    log2 = logging.getLogger('s.s2')  print(4,log2.getEffectiveLevel())    h2 = logging.FileHandler('test1.log')  h2.setLevel(logging.WARNING)  log1.addHandler(h2)  filter1 = logging.Filter('s.s2') #过滤器不会记录s.s2的消息，只会记录自己的消息  log1.addFilter(filter1)  print(3,log1.getEffectiveLevel())    log1.warning('log1 warning===')  log2.warning('log2 warning---')    运行结果：  test.log: #handler1记录了到了log1和log2的信息  [2017-12-17 19:43:12,654] 5872 MainThread log1-handler1 log1 warning===  [2017-12-17 19:43:12,654] 5872 MainThread log1-handler1 log2 warning---      test1.log:    #handler2只记录了它自己的信息  log2 warning--- |

loggerLevel --> FilterConditions --> HandlerLevel --> 父LoggerFilter --> 父LoggerHandler --> RootHandler --> 标准输出或记录到日志：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | import logging,datetime    FORMAT = "%(asctime)s %(thread)d %(message)s"  logging.basicConfig(level=logging.WARNING,format=FORMAT,datefmt="[%Y-%m-%d %H:%M:%S]")    "--------------root--------------------"  root = logging.getLogger()  print(1,root.getEffectiveLevel())    "--------------log1-------------------"  log1 = logging.getLogger('s')  log1.setLevel(logging.ERROR)  print(2,log1.getEffectiveLevel())    h1 = logging.FileHandler('h1.log')  h1.setLevel(logging.INFO)  log1.addHandler(h1)    "--------------log2-------------------"  log2 = logging.getLogger('s.s2')  log2.setLevel(logging.WARNING)  print(3,log2.getEffectiveLevel())    # h2 = logging.StreamHandler()  h2 = logging.FileHandler('h2.log')  h2.setLevel(logging.WARNING) # 最低40  f2 = logging.Filter('s.s3') # log2 = s.s2 ，  s, s.s2   s.s3  h2.addFilter(f2)  log2.addHandler(h2)      log2.warning('4,log2 warning -- {}'.format(datetime.datetime.now()))    运行结果：  1 30  2 40  3 30  [2017-12-19 14:23:43] 2508 4,log2 warning -- 2017-12-19 14:23:43.364928    #h1.log  4,log2 warning -- 2017-12-19 14:23:43.364928      #h2.log  空，没有写入 |

官网流程图：



官网的介绍流程图中第三步，在检查Filter过滤条件不满足时就直接终止。但在上面例子中，log2 --> h2 --> h2Filter --> h2Handler ，当log2传入的消息级别不满足h2Filter条件时，没有直接终止，而是继续向上传递给了父logger的Handler(h1)，满足h1级别，最后写入了h1.log。所以，官网上这一步不知道是不是正确的，但上面例子的实验结果却又证明官网的流程是错误的。

总结：

1. 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log2.info('log3 info')，如果log2定义的级别高于info级别，就不会又信息通过log2

2. 如果level没有设置，就用父logger的，如果父logger的level也没有设置，继续找父的父的，最终找到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

3.消息传递流程：

在某个logger上产生某种级别的信息，首先和logger的level检查，如果消息level低于logger的EffectiveLevl有效级别，消息丢弃，不会再向父logger传递该消息。如果通过(大于等于)检查后，则把消息交给logger所有的handler处理，每一个handler需要和自己level比较来决定是否处理。

如果没有一个handler，或者消息已经被handler处理过了，则需要通过本logger的propagate属性是否为True，Ture则把这个消息会继续发给父Logger，当前Logger的父Logger称为当前Logger，新的Logger的所有Handler继续处理消息。

4. logger实例初始的propagate属性为True，即允许想父logger传递消息

5. logger.basicConfig

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，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。