

## 13 | 日志：日志记录真没你想象的那么简单

2020-04-07 朱晔

Java 业务开发常见错误 100 例

[进入课程 >](#)



讲述：王少泽

时长 22:28 大小 20.59M



你好，我是朱晔。今天，我和你分享的是，记录日志可能会踩的坑。

一些同学可能要说了，记录日志还不简单，无非是几个常用的 API 方法，比如 debug、info、warn、error；但我就见过不少坑都是记录日志引起的，容易出错主要在于三个方面：

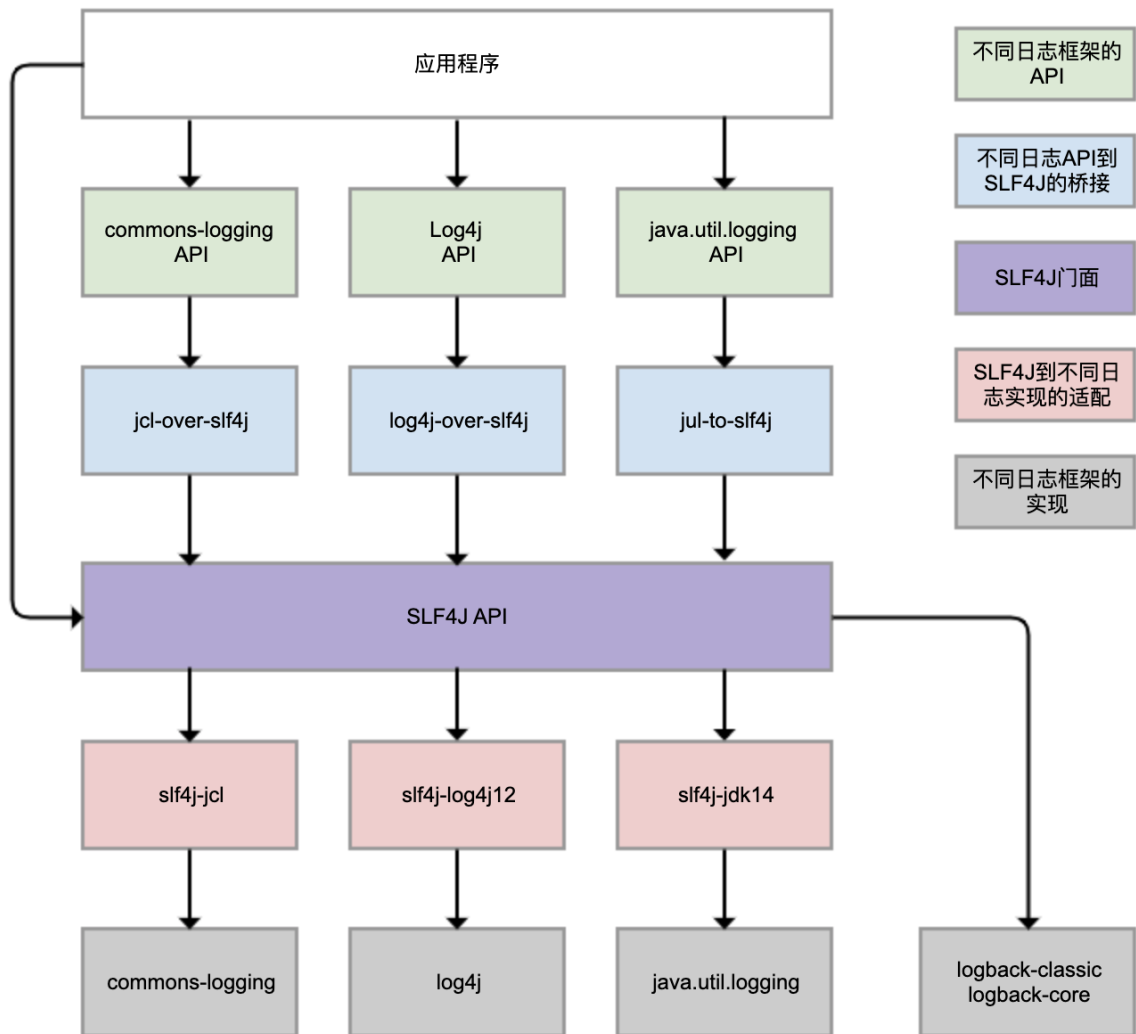
日志框架众多，不同的类库可能会使用不同的日志框架，如何兼容是一个问题。

配置复杂且容易出错。日志配置文件通常很复杂，因此有些开发同学会从其他项目或网络上复制一份配置文件，但却不知道如何修改，甚至是胡乱修改，造成很多问题。比如，重复记录日志的问题、同步日志的性能问题、异步记录的错误配置问题。



日志记录本身就有些误区，比如没考虑到日志内容获取的代价、胡乱使用日志级别等。

Logback、Log4j、Log4j2、commons-logging、JDK 自带的 java.util.logging 等，都是 Java 体系的日志框架，确实非常多。而不同的类库，还可能选择使用不同的日志框架。这样一来，日志的统一管理就变得非常困难。为了解决这个问题，就有了 SLF4J（Simple Logging Facade For Java），如下图所示：



SLF4J 实现了三种功能：

一是提供了统一的日志门面 API，即图中紫色部分，实现了中立的日志记录 API。

二是桥接功能，即图中蓝色部分，用来把各种日志框架的 API（图中绿色部分）桥接到 SLF4J API。这样一来，即便你的程序中使用了各种日志 API 记录日志，最终都可以桥接到 SLF4J 门面 API。

三是适配功能，即图中红色部分，可以实现 SLF4J API 和实际日志框架（图中灰色部分）的绑定。SLF4J 只是日志标准，我们还是需要一个实际的日志框架。日志框架本身没有实现 SLF4J API，所以需要有一个前置转换。Logback 就是按照 SLF4J API 标准实现的，因此不需要绑定模块做转换。

需要理清楚的是，虽然我们可以使用 log4j-over-slf4j 来实现 Log4j 桥接到 SLF4J，也可以使用 slf4j-log4j12 实现 SLF4J 适配到 Log4j，也把它们画到了一列，但是它不能同时使用它们，否则就会产生死循环。jcl 和 jul 也是同样的道理。

虽然图中有 4 个灰色的日志实现框架，但我看到的业务系统使用最广泛的是 Logback 和 Log4j，它们是同一人开发的。Logback 可以认为是 Log4j 的改进版本，我更推荐使用。所以，关于日志框架配置的案例，我都会围绕 Logback 展开。

Spring Boot 是目前最流行的 Java 框架，它的日志框架也用的是 Logback。那，为什么我们没有手动引入 Logback 的包，就可以直接使用 Logback 了呢？

查看 Spring Boot 的 Maven 依赖树，可以发现 spring-boot-starter 模块依赖了 spring-boot-starter-logging 模块，而 spring-boot-starter-logging 模块又帮我们自动引入了 logback-classic（包含了 SLF4J 和 Logback 日志框架）和 SLF4J 的一些适配器。其中，log4j-to-slf4j 用于实现 Log4j2 API 到 SLF4J 的桥接，jul-to-slf4j 则是实现 java.util.logging API 到 SLF4J 的桥接：


```
--- maven-dependency-plugin:3.1.1:tree (default-cli) @ common-mistakes ---
org.geekbang.time:common-mistakes:jar:0.0.1-SNAPSHOT
+- org.springframework.boot:spring-boot-starter-web:jar:2.2.1.RELEASE:compile
| +- org.springframework.boot:spring-boot-starter:jar:2.2.1.RELEASE:compile
| | +- org.springframework.boot:spring-boot:jar:2.2.1.RELEASE:compile
| | +- org.springframework.boot:spring-boot-autoconfigure:jar:2.2.1.RELEASE:compile
| | +- org.springframework.boot:spring-boot-starter-logging:jar:2.2.1.RELEASE:compile
| | | +- ch.qos.logback:logback-classic:jar:1.2.3:compile
| | | | \- ch.qos.logback:logback-core:jar:1.2.3:compile
| | | +- org.apache.logging.log4j:log4j-to-slf4j:jar:2.12.1:compile
| | | | \- org.apache.logging.log4j:log4j-api:jar:2.12.1:compile
| | | \- org.slf4j:jul-to-slf4j:jar:1.7.29:compile
```

接下来，我就用几个实际的案例和你说说日志配置和记录这两大问题，顺便以 Logback 为例复习一下常见的日志配置。

## 为什么我的日志会重复记录？

日志重复记录在业务上非常常见，不但给查看日志和统计工作带来不必要的麻烦，还会增加磁盘和日志收集系统的负担。接下来，我和你分享两个重复记录的案例，同时帮助你梳理 Logback 配置的基本结构。

**第一个案例是，logger 配置继承关系导致日志重复记录。**首先，定义一个方法实现 debug、info、warn 和 error 四种日志的记录：

 复制代码

```
1 @Log4j2
2 @RequestMapping("logging")
3 @RestController
4 public class LoggingController {
5     @GetMapping("log")
6     public void log() {
7         log.debug("debug");
8         log.info("info");
9         log.warn("warn");
10        log.error("error");
11    }
12 }
```

然后，使用下面的 Logback 配置：

第 11 和 12 行设置了全局的日志级别为 INFO，日志输出使用 CONSOLE Appender。

第 3 到 7 行，首先将 CONSOLE Appender 定义为 ConsoleAppender，也就是把日志输出到控制台（System.out/System.err）；然后通过 PatternLayout 定义了日志的输出格式。关于格式化字符串的各种使用方式，你可以进一步查阅 [官方文档](#)。

第 8 到 10 行实现了一个 Logger 配置，将应用包的日志级别设置为 DEBUG、日志输出同样使用 CONSOLE Appender。

 复制代码

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <configuration>
3     <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
4         <layout class="ch.qos.logback.classic.PatternLayout">
5             <pattern>[%d{yyyy-MM-dd HH:mm:ss.SSS}] [%thread] [%-5level] [%logger] %n
6         </layout>
7     </appender>
8     <logger name="org.geekbang.time.commonmistakes.logging" level="DEBUG">
9         <appender-ref ref="CONSOLE"/>
10    </logger>
11 </configuration>
```

```

10     </logger>
11     <root level="INFO">
12         <appender-ref ref="CONSOLE"/>
13     </root>
14 </configuration>

```

这段配置看起来没啥问题，但执行方法后出现了日志重复记录的问题：

```

[2020-01-25 15:54:52.155] [http-nio-45678-exec-1] [DEBUG] [o.g.t.c.logging.LoggingController:55] - debug
[2020-01-25 15:54:52.155] [http-nio-45678-exec-1] [DEBUG] [o.g.t.c.logging.LoggingController:55] - debug
[2020-01-25 15:54:52.156] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController:56] - info
[2020-01-25 15:54:52.156] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController:56] - info
[2020-01-25 15:54:52.156] [http-nio-45678-exec-1] [WARN ] [o.g.t.c.logging.LoggingController:57] - warn
[2020-01-25 15:54:52.156] [http-nio-45678-exec-1] [WARN ] [o.g.t.c.logging.LoggingController:57] - warn
[2020-01-25 15:54:52.156] [http-nio-45678-exec-1] [ERROR] [o.g.t.c.logging.LoggingController:58] - error
[2020-01-25 15:54:52.156] [http-nio-45678-exec-1] [ERROR] [o.g.t.c.logging.LoggingController:58] - error

```

从配置文件的第 9 和 12 行可以看到，CONSOLE 这个 Appender 同时挂载到了两个 Logger 上，一个是我们定义的，一个是，由于我们定义的继承自，所以同一条日志既会通过 logger 记录，也会发送到 root 记录，因此应用 package 下的日志出现了重复记录。

后来我了解到，这个同学如此配置的初衷是实现自定义的 logger 配置，让应用内的日志暂时开启 DEBUG 级别的日志记录。其实，他完全不需要重复挂载 Appender，去掉下挂载的 Appender 即可：

复制代码

```

1 <logger name="org.geekbang.time.commonmistakes.logging" level="DEBUG"/>

```

如果自定义的需要把日志输出到不同的 Appender，比如将应用的日志输出到文件 app.log、把其他框架的日志输出到控制台，可以设置的 additivity 属性为 false，这样就不会继承的 Appender 了：

复制代码

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <configuration>
3     <appender name="FILE" class="ch.qos.logback.core.FileAppender">
4         <file>app.log</file>
5         <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
6             <pattern>[%d{yyyy-MM-dd HH:mm:ss.SSS}] [%thread] [%-5level] [%logger] %n</pattern>
7         </encoder>
8     </appender>
9     <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">

```

```


10     <layout class="ch.qos.logback.classic.PatternLayout">
11         <pattern>[%d{yyyy-MM-dd HH:mm:ss.SSS}] [%thread] [%-5level] [%logger{
12     </layout>
13 </appender>
14     <logger name="org.geekbang.time.commonmistakes.logging" level="DEBUG" addi
15         <appender-ref ref="FILE" />
16     </logger>
17 <root level="INFO">
18     <appender-ref ref="CONSOLE" />
19 </root>
20 </configuration>

```

## 第二个案例是，错误配置 LevelFilter 造成日志重复记录。

一般互联网公司都会使用 ELK 三件套来统一收集日志，有一次我们发现 Kibana 上展示的日志有部分重复，一直怀疑是 Logstash 配置错误，但最后发现还是 Logback 的配置错误引起的。

这个项目的日志是这样配置的：在记录日志到控制台的同时，把日志记录按照不同的级别记录到两个文件中：

 复制代码

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <configuration>
3     <property name="logDir" value="./logs" />
4     <property name="app.name" value="common-mistakes" />
5     <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
6         <layout class="ch.qos.logback.classic.PatternLayout">
7             <pattern>[%d{yyyy-MM-dd HH:mm:ss.SSS}] [%thread] [%-5level] [%logger{
8         </layout>
9     </appender>
10    <appender name="INFO_FILE" class="ch.qos.logback.core.FileAppender">
11        <File>${logDir}/${app.name}_info.log</File>
12        <filter class="ch.qos.logback.classic.filter.LevelFilter">
13            <level>INFO</level>
14        </filter>
15        <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
16            <pattern>[%d{yyyy-MM-dd HH:mm:ss.SSS}] [%thread] [%-5level] [%logger{
17            <charset>UTF-8</charset>
18        </encoder>
19    </appender>
20    <appender name="ERROR_FILE" class="ch.qos.logback.core.FileAppender
21 ">
22        <File>${logDir}/${app.name}_error.log</File>
23        <filter class="ch.qos.logback.classic.filter.ThresholdFilter">

```



```

24         <level>WARN</level>
25     </filter>
26     <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
27         <pattern>[%d{yyyy-MM-dd HH:mm:ss.SSS}] [%thread] [%-5level] [%logger{
28             <charset>UTF-8</charset>
29         </encoder>
30     </appender>
31     <root level="INFO">
32         <appender-ref ref="CONSOLE" />
33         <appender-ref ref="INFO_FILE"/>
34         <appender-ref ref="ERROR_FILE"/>
35     </root>
36 </configuration>

```

这个配置文件比较长，我带着你一段一段地看：

第 31 到 35 行定义的 root 引用了三个 Appender。

第 5 到 9 行是第一个 ConsoleAppender，用于把所有日志输出到控制台。

第 10 到 19 行定义了一个 FileAppender，用于记录文件日志，并定义了文件名、记录日志的格式和编码等信息。最关键的是，第 12 到 14 行定义的 LevelFilter 过滤日志，将过滤级别设置为 INFO，目的是希望 \_info.log 文件中可以记录 INFO 级别的日志。

第 20 到 30 行定义了一个类似的 FileAppender，并使用 ThresholdFilter 来过滤日志，过滤级别设置为 WARN，目的是把 WARN 以上级别的日志记录到另一个 \_error.log 文件中。

运行一下测试程序：

```

→ common-mistakes git:(master) ✗ tail -3 logs/common-mistakes_info.log
[2020-01-25 20:20:48.850] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController:56] - info
[2020-01-25 20:20:48.851] [http-nio-45678-exec-1] [WARN ] [o.g.t.c.logging.LoggingController:57] - warn
[2020-01-25 20:20:48.851] [http-nio-45678-exec-1] [ERROR] [o.g.t.c.logging.LoggingController:58] - error
→ common-mistakes git:(master) ✗ tail -2 logs/common-mistakes_error.log
[2020-01-25 20:20:48.851] [http-nio-45678-exec-1] [WARN ] [o.g.t.c.logging.LoggingController:57] - warn
[2020-01-25 20:20:48.851] [http-nio-45678-exec-1] [ERROR] [o.g.t.c.logging.LoggingController:58] - error

```

可以看到，\_info.log 中包含了 INFO、WARN 和 ERROR 三个级别的日志，不符合我们的预期；error.log 包含了 WARN 和 ERROR 两个级别的日志。因此，造成了日志的重复收集。

你可能会问，这么明显的日志重复为什么没有及时发现？一些公司使用自动化的 ELK 方案收集日志，日志会同时输出到控制台和文件，开发人员在本机测试时不太会关心文件中记录的日志，而在测试和生产环境又因为开发人员没有服务器访问权限，所以原始日志文件中的重复问题并不容易发现。

为了分析日志重复的原因，我们来复习一下 ThresholdFilter 和 LevelFilter 的配置方式。

分析 ThresholdFilter 的源码发现，当日志级别大于等于配置的级别时返回 NEUTRAL，继续调用过滤器链上的下一个过滤器；否则，返回 DENY 直接拒绝记录日志：

 复制代码

```
1 public class ThresholdFilter extends Filter<ILoggingEvent> {
2     public FilterReply decide(ILoggingEvent event) {
3         if (!isStarted()) {
4             return FilterReply.NEUTRAL;
5         }
6
7         if (event.getLevel().isGreaterOrEqual(level)) {
8             return FilterReply.NEUTRAL;
9         } else {
10            return FilterReply.DENY;
11        }
12    }
13 }
```

在这个案例中，把 ThresholdFilter 设置为 WARN，可以记录 WARN 和 ERROR 级别的日志。

LevelFilter 用来比较日志级别，然后进行相应处理：如果匹配就调用 onMatch 定义的处理方式，默认是交给下一个过滤器处理（AbstractMatcherFilter 基类中定义的默认值）；否则，调用 onMismatch 定义的处理方式，默认也是交给下一个过滤器处理。

 复制代码

```
1 public class LevelFilter extends AbstractMatcherFilter<ILoggingEvent> {
2     public FilterReply decide(ILoggingEvent event) {
3         if (!isStarted()) {
4             return FilterReply.NEUTRAL;
5         }
6
7     }
```




```

8         if (event.getLevel().equals(level)) {
9             return onMatch;
10        } else {
11            return onMismatch;
12        }
13    }
14 }
15
16 public abstract class AbstractMatcherFilter<E> extends Filter<E> {
17     protected FilterReply onMatch = FilterReply.NEUTRAL;
18     protected FilterReply onMismatch = FilterReply.NEUTRAL;
19 }

```

和 ThresholdFilter 不同的是，LevelFilter 仅仅配置 level 是无法真正起作用的。**由于没有配置 onMatch 和 onMismatch 属性，所以相当于这个过滤器是无用的，导致 INFO 以上级别的日志都记录了。**

定位到问题后，修改方式就很明显了：配置 LevelFilter 的 onMatch 属性为 ACCEPT，表示接收 INFO 级别的日志；配置 onMismatch 属性为 DENY，表示除了 INFO 级别都不记录：

 复制代码

```

1 <appender name="INFO_FILE" class="ch.qos.logback.core.FileAppender">
2   <File>${logDir}/${app.name}_info.log</File>
3   <filter class="ch.qos.logback.classic.filter.LevelFilter">
4     <level>INFO</level>
5     <onMatch>ACCEPT</onMatch>
6     <onMismatch>DENY</onMismatch>
7   </filter>
8   ...
9 </appender>

```

这样修改后，\_info.log 文件中只会有 INFO 级别的日志，不会出现日志重复的问题了。


## 使用异步日志改善性能的坑

掌握了把日志输出到文件中的方法后，我们接下来面临的问题是，如何避免日志记录成为应用的性能瓶颈。这可以帮助我们解决，磁盘（比如机械磁盘）IO 性能较差、日志量又很大的情况下，如何记录日志的问题。

我们先来测试一下，记录日志的性能问题，定义如下的日志配置，一共有两个 Appender：

FILE 是一个 FileAppender，用于记录所有的日志；

CONSOLE 是一个 ConsoleAppender，用于记录带有 time 标记的日志。

 复制代码


```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <configuration>
3     <appender name="FILE" class="ch.qos.logback.core.FileAppender">
4         <file>app.log</file>
5         <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
6             <pattern>[%d{yyyy-MM-dd HH:mm:ss.SSS}] [%thread] [%-5level] [%logger] %n</pattern>
7         </encoder>
8     </appender>
9     <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
10        <layout class="ch.qos.logback.classic.PatternLayout">
11            <pattern>[%d{yyyy-MM-dd HH:mm:ss.SSS}] [%thread] [%-5level] [%logger] %n</pattern>
12        </layout>
13        <filter class="ch.qos.logback.core.filter.EvaluatorFilter">
14            <evaluator class="ch.qos.logback.classic.boolex.OnMarkerEvaluator">
15                <marker>time</marker>
16            </evaluator>
17            <onMismatch>DENY</onMismatch>
18            <onMatch>ACCEPT</onMatch>
19        </filter>
20    </appender>
21    <root level="INFO">
22        <appender-ref ref="FILE"/>
23        <appender-ref ref="CONSOLE"/>
24    </root>
25 </configuration>
```

不知道你有没有注意到，这段代码中有个 EvaluatorFilter（求值过滤器），用于判断日志是否符合某个条件。

在后续测试代码中，我们会把大量日志输出到文件中，日志文件会非常大，如果性能测试结果也混在其中的话，就很难找到那条日志。所以，这里我们使用 EvaluatorFilter 对日志按照标记进行过滤，并将过滤出的日志单独输出到控制台上。在这个案例中，我们给输出测试结果的那条日志上做了 time 标记。

配合使用标记和 `EvaluatorFilter`，实现日志的按标签过滤，是一个不错的小技巧。

如下测试代码中，实现了记录指定次数的大日志，每条日志包含 1MB 字节的模拟数据，最后记录一条以 `time` 为标记的方法执行耗时日志：

 复制代码


```
1 @GetMapping("performance")
2 public void performance(@RequestParam(name = "count", defaultValue = "1000") i
3     long begin = System.currentTimeMillis();
4     String payload = IntStream.rangeClosed(1, 1000000)
5         .mapToObj(__ -> "a")
6         .collect(Collectors.joining("")) + UUID.randomUUID().toString();
7     IntStream.rangeClosed(1, count).forEach(i -> log.info("{} {}", i, payload));
8     Marker timeMarker = MarkerFactory.getMarker("time");
9     log.info(timeMarker, "took {} ms", System.currentTimeMillis() - begin);
10 }
```

执行程序后可以看到，记录 1000 次日志和 10000 次日志的调用耗时，分别是 6.3 秒和 44.5 秒：

```
[2020-01-25 21:20:44.638] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController:76] - took 6320 ms
[2020-01-25 21:21:32.251] [http-nio-45678-exec-2] [INFO ] [o.g.t.c.logging.LoggingController:76] - took 44525 ms
```

对于只记录文件日志的代码来说，这个耗时挺长的。为了分析其中原因，我们需要分析下 `FileAppender` 的源码。

`FileAppender` 继承自 `OutputStreamAppender`，查看 `OutputStreamAppender` 源码的第 30 到 33 行发现，在追加日志的时候，是直接把日志写入 `OutputStream` 中，属于同步记录日志：

 复制代码

```
1 public class OutputStreamAppender<E> extends UnsynchronizedAppenderBase<E> {
2     private OutputStream outputStream;
3     boolean immediateFlush = true;
4     @Override
5     protected void append(E eventObject) {
6         if (!isStarted()) {
7             return;
8         }
9         subAppend(eventObject);
10    }
11 }
```

```

12     protected void subAppend(E event) {
13         if (!isStarted()) {
14             return;
15         }
16         try {
17             //编码LoggingEvent
18             byte[] byteArray = this.encoder.encode(event);
19             //写字节流
20             writeBytes(byteArray);
21         } catch (IOException ioe) {
22             ...
23         }
24     }
25
26     private void writeBytes(byte[] byteArray) throws IOException {
27         if(byteArray == null || byteArray.length == 0)
28             return;
29
30         lock.lock();
31         try {
32             //这个OutputStream其实是一个ResilientFileOutputStream, 其内部使用的是带缓冲
33             this.outputStream.write(byteArray);
34             if (immediateFlush) {
35                 this.outputStream.flush(); //刷入OS
36             }
37         } finally {
38             lock.unlock();
39         }
40     }
41 }

```

分析到这里，我们就明白为什么日志大量写入时会耗时这么久了。那，有没有办法实现大量日志写入时，不会过多影响业务逻辑执行耗时，影响吞吐量呢？

办法当然有了，使用 Logback 提供的 AsyncAppender 即可实现异步的日志记录。AsyncAppender 类似装饰模式，也就是在不改变类原有基本功能的情况下为其增添新功能。这样，我们就可以把 AsyncAppender 附加在其他的 Appender 上，将其变为异步的。

定义一个异步 Appender ASYNCFILE，包装之前的同步文件日志记录的 FileAppender，就可以实现异步记录日志到文件：

 复制代码

```
1 <appender name="ASYNCFILE" class="ch.qos.logback.classic.AsyncAppender">
```

```
2     <appender-ref ref="FILE"/>
3 </appender>
4 <root level="INFO">
5     <appender-ref ref="ASYNCFILE"/>
6     <appender-ref ref="CONSOLE"/>
7 </root>
```

测试一下可以发现，记录 1000 次日志和 10000 次日志的调用耗时，分别是 735 毫秒和 668 毫秒：

```
[2020-01-25 21:46:04.539] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController:76] - took 735 ms
[2020-01-25 21:46:07.483] [http-nio-45678-exec-2] [INFO ] [o.g.t.c.logging.LoggingController:76] - took 668 ms
```

性能居然这么好，你觉得其中有什么问题吗？异步日志真的如此神奇和万能吗？当然不是，因为这样并没有记录下所有日志。**我之前就遇到过很多关于 AsyncAppender 异步日志的坑，这些坑可以归结为三类：**

记录异步日志撑爆内存；

记录异步日志出现日志丢失；


记录异步日志出现阻塞。

为了解释这三种坑，我来模拟一个慢日志记录场景：首先，自定义一个继承自 ConsoleAppender 的 MySlowAppender，作为记录到控制台的输出器，写入日志时休眠 1 秒。

```
1 public class MySlowAppender extends ConsoleAppender {
2     @Override
3     protected void subAppend(Object event) {
4         try {
5             // 模拟慢日志
6             TimeUnit.MILLISECONDS.sleep(1);
7         } catch (InterruptedException e) {
8             e.printStackTrace();
9         }
10        super.subAppend(event);
11    }
12 }
```

 复制代码

然后，在配置文件中使用 AsyncAppender，将 MySlowAppender 包装为异步日志记录：

 复制代码

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <configuration>
3 <appender name="CONSOLE" class="org.geekbang.time.commonmistakes.logging.async
4     <layout class="ch.qos.logback.classic.PatternLayout">
5         <pattern>[%d{yyyy-MM-dd HH:mm:ss.SSS}] [%thread] [%-5level] [%logg
6     </layout>
7 </appender>
8 <appender name="ASYNC" class="ch.qos.logback.classic.AsyncAppender">
9     <appender-ref ref="CONSOLE" />
10 </appender>
11 <root level="INFO">
12     <appender-ref ref="ASYNC" />
13 </root>
14 </configuration>
```

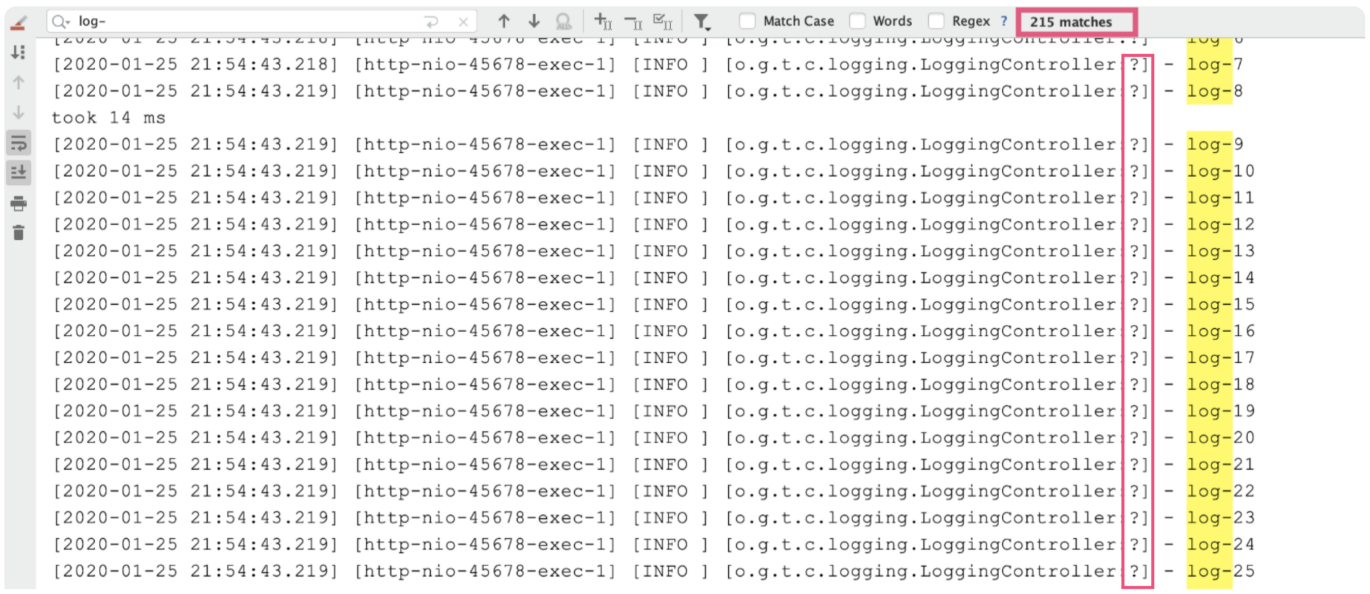
定义一段测试代码，循环记录一定次数的日志，最后输出方法执行耗时：

 复制代码

```
1 @GetMapping("manylog")
2 public void manylog(@RequestParam(name = "count", defaultValue = "1000") int c
3     long begin = System.currentTimeMillis();
4     IntStream.rangeClosed(1, count).forEach(i -> log.info("log-{}", i));
5     System.out.println("took " + (System.currentTimeMillis() - begin) + " ms")
6 }
```

执行方法后发现，耗时很短但出现了日志丢失：我们要记录 1000 条日志，最终控制台只能搜索到 215 条日志，而且日志的行号变为了一个问号。





```
log-
[2020-01-25 21:54:43.219] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController?] - log-7
[2020-01-25 21:54:43.219] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController?] - log-8
took 14 ms
[2020-01-25 21:54:43.219] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController?] - log-9
[2020-01-25 21:54:43.219] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController?] - log-10
[2020-01-25 21:54:43.219] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController?] - log-11
[2020-01-25 21:54:43.219] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController?] - log-12
[2020-01-25 21:54:43.219] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController?] - log-13
[2020-01-25 21:54:43.219] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController?] - log-14
[2020-01-25 21:54:43.219] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController?] - log-15
[2020-01-25 21:54:43.219] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController?] - log-16
[2020-01-25 21:54:43.219] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController?] - log-17
[2020-01-25 21:54:43.219] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController?] - log-18
[2020-01-25 21:54:43.219] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController?] - log-19
[2020-01-25 21:54:43.219] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController?] - log-20
[2020-01-25 21:54:43.219] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController?] - log-21
[2020-01-25 21:54:43.219] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController?] - log-22
[2020-01-25 21:54:43.219] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController?] - log-23
[2020-01-25 21:54:43.219] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController?] - log-24
[2020-01-25 21:54:43.219] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.logging.LoggingController?] - log-25
```


出现这个问题的原因在于，AsyncAppender 提供了一些配置参数，而我们没用对。我们结合相关源码分析一下：

includeCallerData 用于控制是否收集调用方数据，默认是 false，此时方法行号、方法名等信息将不能显示（源码第 2 行以及 7 到 11 行）。

queueSize 用于控制阻塞队列大小，使用的 ArrayBlockingQueue 阻塞队列（源码第 15 到 17 行），默认大小是 256，即内存中最多保存 256 条日志。

discardingThreshold 是控制丢弃日志的阈值，主要是防止队列满后阻塞。默认情况下，队列剩余量低于队列长度的 20%，就会丢弃 TRACE、DEBUG 和 INFO 级别的日志。（参见源码第 3 到 6 行、18 到 19 行、26 到 27 行、33 到 34 行、40 到 42 行）

neverBlock 用于控制队列满的时候，加入的数据是否直接丢弃，不会阻塞等待，默认是 false（源码第 44 到 68 行）。这里需要注意一下 offer 方法和 put 方法的区别，当队列满的时候 offer 方法不阻塞，而 put 方法会阻塞；neverBlock 为 true 时，使用 offer 方法。

 复制代码

```
1 public class AsyncAppender extends AsyncAppenderBase<ILoggingEvent> {
2     boolean includeCallerData = false;//是否收集调用方数据
3     protected boolean isDiscardable(ILoggingEvent event) {
4         Level level = event.getLevel();
5         return level.toInt() <= Level.INFO_INT;//丢弃<=INFO级别的日志
6     }
7     protected void preprocess(ILoggingEvent eventObject) {
8         eventObject.prepareForDeferredProcessing();
9         if (includeCallerData)
10             eventObject.getCallerData();
```

```

11     }
12 }
13 public class AsyncAppenderBase<E> extends UnsyncronizedAppenderBase<E> implem
14
15     BlockingQueue<E> blockingQueue; //异步日志的关键，阻塞队列
16     public static final int DEFAULT_QUEUE_SIZE = 256; //默认队列大小
17     int queueSize = DEFAULT_QUEUE_SIZE;
18     static final int UNDEFINED = -1;
19     int discardingThreshold = UNDEFINED;
20     boolean neverBlock = false; //控制队列满的时候加入数据时是否直接丢弃，不会阻塞等待
21
22     @Override
23     public void start() {
24         ...
25         blockingQueue = new ArrayBlockingQueue<E>(queueSize);
26         if (discardingThreshold == UNDEFINED)
27             discardingThreshold = queueSize / 5; //默认丢弃阈值是队列剩余量低于队列长
28         ...
29     }
30
31     @Override
32     protected void append(E eventObject) {
33         if (isQueueBelowDiscardingThreshold() && isDiscardable(eventObject)) {
34             return;
35         }
36         preprocess(eventObject);
37         put(eventObject);
38     }
39
40     private boolean isQueueBelowDiscardingThreshold() {
41         return (blockingQueue.remainingCapacity() < discardingThreshold);
42     }
43
44     private void put(E eventObject) {
45         if (neverBlock) { //根据neverBlock决定使用不阻塞的offer还是阻塞的put方法
46             blockingQueue.offer(eventObject);
47         } else {
48             putUninterruptibly(eventObject);
49         }
50     }
51     //以阻塞方式添加数据到队列
52     private void putUninterruptibly(E eventObject) {
53         boolean interrupted = false;
54         try {
55             while (true) {
56                 try {
57                     blockingQueue.put(eventObject);
58                     break;
59                 } catch (InterruptedException e) {
60                     interrupted = true;
61                 }
62             }

```

```
63         } finally {  
64             if (interrupted) {  
65                 Thread.currentThread().interrupt();  
66             }  
67         }  
68     }  
69 }
```

看到默认队列大小为 256，达到 80% 容量后开始丢弃  $\leq$  INFO 级别的日志后，我们就可以理解日志中为什么只有 215 条 INFO 日志了。

我们可以继续分析下异步记录日志出现坑的原因。

queueSize 设置得特别大，就可能会导致 OOM。

queueSize 设置得比较小（默认值就非常小），且 discardingThreshold 设置为大于 0 的值（或者为默认值），队列剩余容量少于 discardingThreshold 的配置就会丢弃  $\leq$  INFO 的日志。这里的坑点有两个。一是，因为 discardingThreshold 的存在，设置 queueSize 时容易踩坑。比如，本例中最大日志并发是 1000，即便设置 queueSize 为 1000 同样会导致日志丢失。二是，discardingThreshold 参数容易有歧义，它不是百分比，而是日志条数。对于总容量 10000 的队列，如果希望队列剩余容量少于 1000 条的时候丢弃，需要配置为 1000。

neverBlock 默认为 false，意味着总可能会出现阻塞。如果 discardingThreshold 为 0，那么队列满时再有日志写入就会阻塞；如果 discardingThreshold 不为 0，也只会丢弃  $\leq$  INFO 级别的日志，那么出现大量错误日志时，还是会阻塞程序。

可以看出 queueSize、discardingThreshold 和 neverBlock 这三个参数息息相关，务必按需进行设置和取舍，到底是性能为先，还是数据不丢为先：

如果考虑绝对性能为先，那就设置 neverBlock 为 true，永不阻塞。

如果考虑绝对不丢数据为先，那就设置 discardingThreshold 为 0，即使是  $\leq$  INFO 的级别日志也不会丢，但最好把 queueSize 设置大一点，毕竟默认的 queueSize 显然太小，太容易阻塞。

如果希望兼顾两者，可以丢弃不重要的日志，把 queueSize 设置大一点，再设置一个合理的 discardingThreshold。

以上就是日志配置最常见的两个误区了。接下来，我们再看一个日志记录本身的误区。

## 使用日志占位符就不需要进行日志级别判断了？

不知道你有没有听人说过：SLF4J 的{}占位符语法，到真正记录日志时才会获取实际参数，因此解决了日志数据获取的性能问题。你觉得，这种说法对吗？

为了验证这个问题，我们写一段测试代码：有一个 slowString 方法，返回结果耗时 1 秒：

 复制代码


```
1 private String slowString(String s) {
2     System.out.println("slowString called via " + s);
3     try {
4         TimeUnit.SECONDS.sleep(1);
5     } catch (InterruptedException e) {
6     }
7     return "OK";
8 }
```

如果我们记录 DEBUG 日志，并设置只记录  $\geq$  INFO 级别的日志，程序是否也会耗时 1 秒呢？我们使用三种方法来测试：

拼接字符串方式记录 slowString；

使用占位符方式记录 slowString；

先判断日志级别是否启用 DEBUG。

 复制代码

```
1 Stopwatch stopWatch = new Stopwatch();
2 stopWatch.start("debug1");
3 log.debug("debug1:" + slowString("debug1"));
4 stopWatch.stop();
5 stopWatch.start("debug2");
6 log.debug("debug2:{}", slowString("debug2"));
7 stopWatch.stop();
8 stopWatch.start("debug3");
9 if (log.isDebugEnabled())
10     log.debug("debug3:{}", slowString("debug3"));
11 stopWatch.stop();
```


可以看到，前两种方式都调用了 `slowString` 方法，所以耗时都是 1 秒：

```
slowString called via debug1
slowString called via debug2
[2020-01-26 11:59:06.955] [http
-----
ns          %      Task name
-----
1004678839  050%   debug1
1003741887  050%   debug2
000002164   000%   debug3
```

使用占位符方式记录 `slowString` 的方式，同样需要耗时 1 秒，是因为这种方式虽然允许我们传入 `Object`，不用拼接字符串，但也只是延迟（如果日志不记录那么就是省去）了日志参数对象 `toString()` 和字符串拼接的耗时。


在这个案例中，除非事先判断日志级别，否则必然会调用 `slowString` 方法。**回到之前提的问题，使用 {} 占位符语法不能通过延迟参数值获取，来解决日志数据获取的性能问题。**

除了事先判断日志级别，我们还可以通过 `lambda` 表达式进行延迟参数内容获取。但，SLF4J 的 API 还不支持 `lambda`，因此需要使用 Log4j2 日志 API，把 Lombok 的 `@Slf4j` 注解替换为 `@Log4j2` 注解，这样就可以提供一个 `lambda` 表达式作为提供参数数据的方法：

 复制代码

```
1 @Log4j2
2 public class LoggingController {
3     ...
4     log.debug("debug4:{}", ()->slowString("debug4"));
```

像这样调用 debug 方法，签名是 Supplier<?>，参数会延迟到真正需要记录日志时再获取：

 复制代码

```
1 void debug(String message, Supplier<?>... paramSuppliers);
2
3 public void logIfEnabled(final String fqcn, final Level level, final Marker ma
4     final Supplier<?>... paramSuppliers) {
5     if (isEnabled(level, marker, message)) {
6         logMessage(fqcn, level, marker, message, paramSuppliers);
7     }
8 }
9 protected void logMessage(final String fqcn, final Level level, final Marker m
10     final Supplier<?>... paramSuppliers) {
11     final Message msg = messageFactory.newMessage(message, LambdaUtil.getAll(p
12     logMessageSafely(fqcn, level, marker, msg, msg.getThrowable());
13 }
```

修改后再次运行测试，可以看到这次 debug4 并不会调用 slowString 方法：

```
-----
ns           %           Task name
-----
1003296610   050%    debug1
1003422079   050%    debug2
000002657    000%    debug3
000007714    000%    debug4
```

其实，我们只是换成了 Log4j2 API，真正的日志记录还是走的 Logback 框架。没错，这就是 SLF4J 适配的一个好处。

## 重点回顾

我将记录日志的坑，总结为框架使用配置和记录本身两个方面。



Java 的日志框架众多，SLF4J 实现了这些框架记录日志的统一。在使用 SLF4J 时，我们需要理清其桥接 API 和绑定这两个模块。如果程序启动时出现 SLF4J 的错误提示，那很可能是配置出现了问题，可以使用 Maven 的 `dependency:tree` 命令梳理依赖关系。

Logback 是 Java 最常用的日志框架，其配置比较复杂，你可以参考官方文档中关于 Appender、Layout、Filter 的配置，切记不要随意从其他地方复制别人的配置，避免出现错误或与当前需求不符。

使用异步日志解决性能问题，是用空间换时间。但空间毕竟有限，当空间满了之后，我们要考虑是阻塞等待，还是丢弃日志。如果更希望不丢弃重要日志，那么选择阻塞等待；如果更希望程序不要因为日志记录而阻塞，那么就需要丢弃日志。

最后，我强调的是，日志框架提供的参数化日志记录方式不能完全取代日志级别的判断。如果你的日志量很大，获取日志参数代价也很大，就要进行相应日志级别的判断，避免不记录日志也要花费时间获取日志参数的问题。

今天用到的代码，我都放在了 GitHub 上，你可以点击 [🔗 这个链接](#) 查看。

## 思考与讨论

1. 在第一小节的案例中，我们把 INFO 级别的日志存放到 `_info.log` 中，把 WARN 和 ERROR 级别的日志存放到 `_error.log` 中。如果现在要把 INFO 和 WARN 级别的日志存放到 `_info.log` 中，把 ERROR 日志存放到 `_error.log` 中，应该如何配置 Logback 呢？
2. 生产级项目的文件日志肯定需要按时间和日期进行分割和归档处理，以避免单个文件太大，同时保留一定天数的历史日志，你知道如何配置吗？可以在 [🔗 官方文档](#) 找到答案。

针对日志记录和配置，你还遇到过其他坑吗？我是朱晔，欢迎在评论区与我留言分享，也欢迎你把这篇文章分享给你的朋友或同事，一起交流。

点击参与 

## 进入朱晔老师「读者群」带你攻克 Java 业务开发常见错误



添加Java班长，报名入群



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 12 | 异常处理：别让自己在出问题的时候变为瞎子

下一篇 加餐1 | 带你吃透课程中Java 8的那些重要知识点（上）

### 精选留言 (7)

 写留言



**jiarupc**

2020-04-07

对于我们这些小公司，日志的性能不是问题，日志也没踩过坑。

最大的问题是，日志记录不合理。我工作过两家公司。

一家公司日志记录很随意，想起来才记录，日志根本没啥用，全凭经验找问题。...

展开 



 3



**Monday**

2020-04-07

@Log4j2 @Slf4j 两个注解使用的区别是什么，前者使用了Log4j的框架记录日志，后者使用了默认的Logback框架吗？

展开 ▾

作者回复: @Log4j

Creates private static final org.apache.log4j.Logger log = org.apache.log4j.Logger.getLogger(LogExample.class);

@Log4j2

Creates private static final org.apache.logging.log4j.Logger log = org.apache.logging.log4j.LogManager.getLogger(LogExample.class);

@Slf4j

Creates private static final org.slf4j.Logger log = org.slf4j.LoggerFactory.getLogger(LogExample.class);

@Log4j2和@Slf4j只是日志记录API，和实际日志记录框架没有关系，可以再看看文章一开始说的那段有关SLF4J结构的内容



**Darren**

2020-04-08

我们在线上的日志基本遇到的问题也不多，最多就是日志消费不及时问题，目前通过filebeat采集写入kafka，strom消费，写入es聚合，然后前端展示；现在有延迟问题，正在切flink。

回答下问题：

第一个问题采用了表达式； ...

展开 ▾



**Husiun**

2020-04-07

课后问题1.过滤级别可以多个用逗号隔开2.日志归档，通过设置不同的滚动policy策略将日志存档,当然logback比log4j更好的是可以自动删除过期日志；还望老师指点--



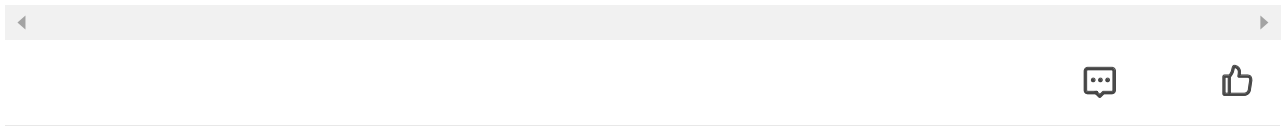
**Husiun**

2020-04-07

老师还有一点我补充一下，springboot默认使用starter日志依赖logback的时候，日志配置文件名应以-spring结尾，才会默认加入其上下文环境中。

展开 ▾

作者回复: 是的, 相关信息参考 <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-logback-extensions>



**王飞洋**

2020-04-07

通过源码发现问题, 解决问题

展开 ∨



**hellojd**

2020-04-07

赞, 日志是性能的隐形杀手, 学问也不小。佩服老师的工匠精神。

