

# 学习过程中你是否有过不去的坎!

一个bug是不是能让你卡好久不能解决，然后想放弃



编程是可以自学的，但是有一个陪你学习的老师一定可以事半功倍



楠哥和李老师  
全程陪跑学习Java



## 一对一调试 重点难点讲解



### 扫码添加 立享优惠



IT楠老师 (微信号: itnanls)

5年开发, 2年教学经验

曾参与大型呼叫中心系统、全国电网线路损耗治理平台等大型项目的研发工作

B站粉丝13w, 帮助上千名学生或粉丝成功就业



IT李老师 (微信号: itlils)

5年java开发, 2年大数据组长, 3年教学经验

擅长java实战项目, 大数据实战

从事多年Java软件开发及相关教育工作

精通Java开发技术, 熟悉大数据技术

曾参与多个大型银行短信服务开发

# 第一章 日志的概念

---

## 一、概述

日志文件是用于记录系统操作事件的文件集合，可分为事件日志和消息日志。具有处理历史数据、诊断问题的追踪以及理解系统的活动等重要作用。

在计算机中，日志文件是记录在操作系统或其他软件运行中发生的事件或在通信软件的不同用户之间的消息的文件。记录是保持日志的行为。在最简单的情况下，消息被写入单个日志文件。

## 二、日志的作用

- 调试

在Java项目调试时，查看栈信息可以方便地知道当前程序的运行状态，输出的日志便于记录程序在之前的运行结果。如果你大量使用 `System.out` 或者 `System.err`，这是一种最方便最有效的方法，但显得不够专业。

- 错误定位

不要以为项目能正确跑起来就可以高枕无忧，项目在运行一段时间后，可能由于数据问题，网络问题，内存问题等出现异常。这时日志可以帮助开发或者运维人员快速定位错误位置，提出解决方案。

- 数据分析

大数据的兴起，使得大量的日志分析成为可能，ELK也让日志分析门槛降低了很多。日志中蕴含了大量的用户数据，包括点击行为，兴趣偏好等，用户画像对于公司下一步的战略方向有一定指引作用。

## 三、接触过的日志

最简单的日志输出方式，我们每天都在使用：

```
1 | System.out.println("这个数的结果是: "+ num);
```

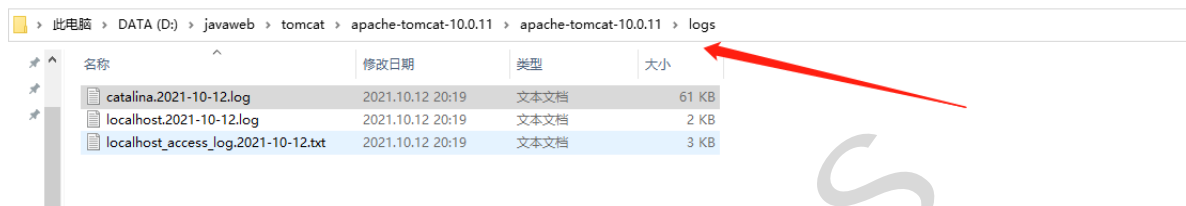
以及错误日志:

```
1 | System.err.println("此处发生了异常");
```

此类代码在程序的执行过程中没有什么实质的作用,但是却能打印一些中间变量,辅助我们调试和错误的排查。

日志系统我们也见过:

在tomcat中



当我们的程序无法启动或者运行过程中产生问题,会有所记录,比如我的catalina.log中查看,发现确实有错误信息,这能帮我们迅速定位:

```
Caused by: java.lang.IllegalArgumentException: 指定的主资源集 [D:\www\image] 无效
    at org.apache.catalina.webresources.StandardRoot.createMainResourceSet(StandardRoot.java:751)
    at org.apache.catalina.webresources.StandardRoot.startInternal(StandardRoot.java:708)
    at org.apache.catalina.util.LifecycleBase.start(LifecycleBase.java:183)
    ... 30 more
12-Oct-2021 20:16:19.819 信息 [main] org.apache.coyote.AbstractProtocol.pause 暂停ProtocolHandler["http-nio-8080"]
12-Oct-2021 20:16:19.819 信息 [main] org.apache.catalina.core.StandardService.stopInternal 正在停止服务[Catalina]
```

而我们的System.err只能做到控制台打印日志,所以我们需要更强大日志框架来处理:

## 四、主流日志框架

- 日志实现(具体干活的): JUL (java util logging)、logback、log4j、log4j2
- 日志门面(指定规则的): JCL (Jakarta Commons Logging)、slf4j (Simple Logging Facade for Java)

## 第二章 JUL日志框架

JUL全称Java util Logging是java原生的日志框架，使用时不需要另外引用第三方类库，相对其他日志框架使用方便，学习简单，能够在小型应用中灵活使用。

在JUL中有以下组件，我们先做了解，慢慢学习：

- Loggers：被称为记录器，应用程序通过获取Logger对象，调用其API来发布日志信息。Logger通常是应用程序访问日志系统的入口程序。
- Appenders：也被称为Handlers，每个Logger都会关联一组Handlers，Logger会将日志交给关联Handlers处理，由Handlers负责将日志做记录。Handlers在此是一个抽象，其具体的实现决定了日志记录的位置可以是控制台、文件、网络上的其他日志服务或操作系统日志等。
- Layouts：也被称为Formatters，它负责对日志事件中的数据进行转换和格式化。Layouts决定了数据在一条日志记录中的最终形式。
- Level：每条日志消息都有一个关联的日志级别。该级别粗略指导了日志消息的重要性和紧迫，我可以将Level和Loggers，Appenders做关联以便于我们过滤消息。
- Filters：过滤器，根据需要定制哪些信息会被记录，哪些信息会被放过。

**总结一下就是：**

用户使用Logger来进行日志记录，Logger持有若干个Handler，日志的输出操作是由Handler完成的。在Handler在输出日志前，会经过Filter的过滤，判断哪些日志级别过滤放行哪些拦截，Handler会将日志内容输出到指定位置（日志文件、控制台等）。Handler在输出日志时会使用Layout，将输出内容进行排版。

## 一、入门案例

```
1 public static void main(String[] args) {
2     Logger logger = Logger.getLogger("myLogger");
3     logger.info("信息");
4     logger.warning("警告信息");
5     logger.severe("严重信息");
6 }
```

```
0. (Program Files (x86)\jdk-11.0.4\windows-x86_bin\jdk-11
10月 21, 2021 11:15:05 上午 com.ydlclass.entity.User main
信息: 信息
10月 21, 2021 11:15:05 上午 com.ydlclass.entity.User main
警告: 警告信息
10月 21, 2021 11:15:05 上午 com.ydlclass.entity.User main
严重: 严重信息
```

## 二、日志的级别

jul中定义的日志级别，从上述例子中我们也看到使用info和warning打印出的日志有不同的前缀，通过给日志设置不同的级别可以清晰的从日志中区分出哪些是基本信息，哪些是调试信息，哪些是严重的异常。

(1) `java.util.logging.Level`中定义了日志的级别:

1. SEVERE (最高値)
2. WARNING
3. INFO (默认级别)
4. CONFIG
5. FINE
6. FINER
7. FINEST (最低値)

再例如：我们查看tomcat的日志，能明显的看到不同级别的日志，其实tomcat默认使用的就是JUL：

```

00 信息 [main] org.apache.catalina.startup.HostConfig.deployDirectory 把web 应用程序部署到目录 [D:\yd12\app]
02 信息 [main] org.apache.jasper.servlet.TldScanner.scanJars 至少有一个JAR被扫描用于TLD但尚未包含TLD。 关闭
09 警告 [main] org.apache.catalina.util.SessionIdGeneratorBase.createSecureRandom 使用[SHA1PRNG]创建会话: 不安全
10 信息 [main] org.apache.catalina.startup.HostConfig.deployDirectory Web应用程序目录 [D:\yd12\app] 的部署已完成
11 信息 [main] org.apache.catalina.startup.HostConfig.deployDirectory 把web 应用程序部署到目录 [D:\yd12\app]
15 信息 [main] org.apache.jasper.servlet.TldScanner.scanJars 至少有一个JAR被扫描用于TLD但尚未包含TLD。 关闭
17 信息 [main] org.apache.catalina.startup.HostConfig.deployDirectory Web应用程序目录 [D:\yd12\app] 的部署已完成
17 严重 [main] org.apache.catalina.core.ContainerBase.startInternal 子容器启动失败
it.ExecutionException: org.apache.catalina.LifecycleException: 子容器启动失败
ncurrent.FutureTask.report(FutureTask.java:122)

```



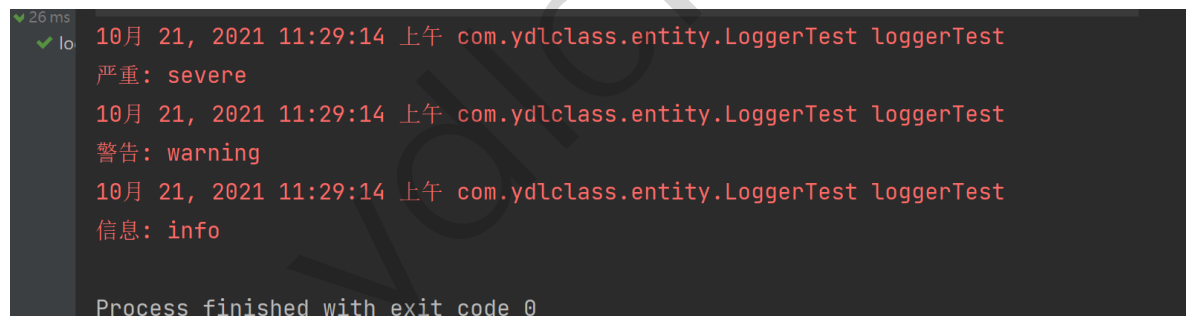
还有两个特殊的级别：

- OFF，可用来关闭日志记录。
- ALL，启用所有消息的日志记录。

虽然我们测试了7个日志级别，

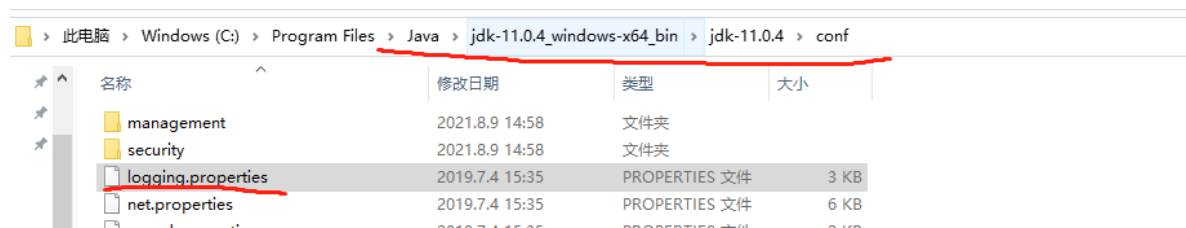
```
1 @Test
2 public void testLogger() {
3     Logger logger =
4     Logger.getLogger(LoggerTest.class.getName());
5     logger.severe("severe");
6     logger.warning("warning");
7     logger.info("info");
8     logger.config("config");
9     logger.fine("fine");
10    logger.finer("finer");
11    logger.finest("finest");
12 }
```

我们发现能够打印的只有三行，这是为什么呢？

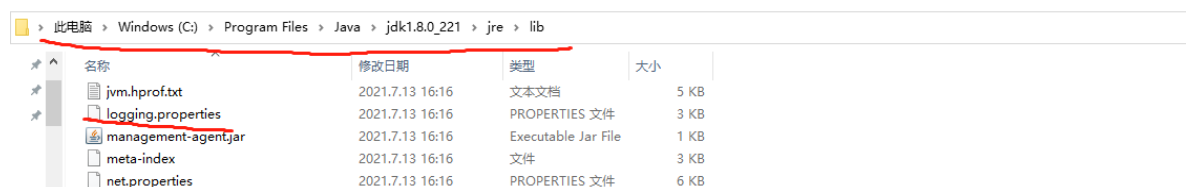


```
26 ms
10月 21, 2021 11:29:14 上午 com.ydlclass.entity.LoggerTest loggerTest
严重: severe
10月 21, 2021 11:29:14 上午 com.ydlclass.entity.LoggerTest loggerTest
警告: warning
10月 21, 2021 11:29:14 上午 com.ydlclass.entity.LoggerTest loggerTest
信息: info
Process finished with exit code 0
```

我们找一下这个文件，下图是jdk11的日志配置文件：



或者在jdk1.8中：



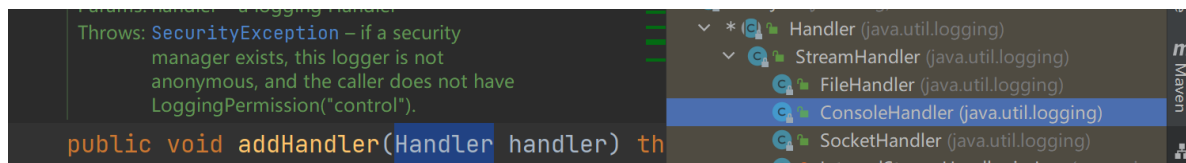
```
47 java.util.logging.ConsoleHandler.level = INFO
48 java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
```

就可以看到系统默认在控制台打印的日志级别了，系统配置我们暂且不动，一会我们独立创建配置文件完成修改。

但是我们可以简单的看看这个日志配置了哪些内容：

```
1  .level= INFO
2
3  #####
4  # Handler specific properties.
5  # Describes specific configuration info for Handlers.
6  #####
7
8  # default file output is in user's home directory.
9  java.util.logging.FileHandler.pattern = %h/java%u.log
10 java.util.logging.FileHandler.limit = 50000
11 java.util.logging.FileHandler.count = 1
12 # Default number of locks FileHandler can obtain
    synchronously.
13 # This specifies maximum number of attempts to obtain
    lock file by FileHandler
14 # implemented by incrementing the unique field %u as
    per FileHandler API documentation.
15 java.util.logging.FileHandler.maxLocks = 100
16 java.util.logging.FileHandler.formatter =
    java.util.logging.XMLFormatter
17
18 # Limit the message that are printed on the console to
    INFO and above.
19 java.util.logging.ConsoleHandler.level = INFO
20 java.util.logging.ConsoleHandler.formatter =
    java.util.logging.SimpleFormatter
```

在日志中我们发现了，貌似可以给这个日志对象添加各种handler就是处理器，比如ConsoleHandler专门处理控制台日志，FileHandler貌似可以处理文件，同时我们确实发现了他有这么一个方法：



### 三、日志配置

```
1  @Test
2  public void testLogConfig() throws Exception {
3      // 1.创建日志记录器对象
4      Logger logger =
5      Logger.getLogger("com.ydlclass.log.JULTest");
6      // 一、自定义日志级别
7      // a.关闭系统默认配置
8      logger.setUseParentHandlers(false);
9      // b.创建handler对象
10     ConsoleHandler consoleHandler = new
11     ConsoleHandler();
12     // c.创建formatter对象
13     SimpleFormatter simpleFormatter = new
14     SimpleFormatter();
15     // d.进行关联
16     consoleHandler.setFormatter(simpleFormatter);
17     logger.addHandler(consoleHandler);
18     // e.设置日志级别
19     logger.setLevel(Level.ALL);
20     consoleHandler.setLevel(Level.ALL);
21     // 二、输出到日志文件
22     FileHandler fileHandler = new
23     FileHandler("d:/logs/jul.log");
24     fileHandler.setFormatter(simpleFormatter);
25     logger.addHandler(fileHandler);
26     // 2.日志记录输出
27     logger.severe("severe");
28     logger.warning("warning");
29     logger.info("info");
30 }
```



```
26     logger.config("config");
27     logger.fine("fine");
28     logger.finer("finer");
29     logger.finest("finest");
30 }
```

再次查看结果：

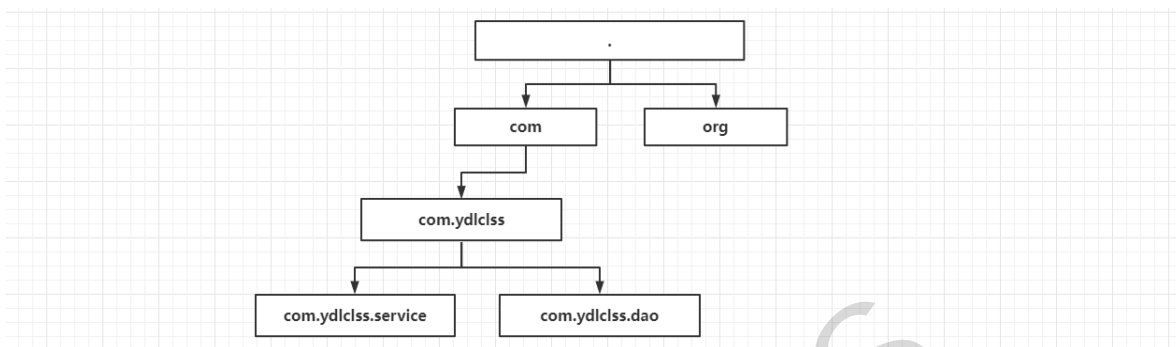
```
1  10月 21, 2021 11:50:01 上午
   com.ydlclass.entity.LoggerTest testConfig
2  严重: severe
3  10月 21, 2021 11:50:01 上午
   com.ydlclass.entity.LoggerTest testConfig
4  警告: warning
5  10月 21, 2021 11:50:01 上午
   com.ydlclass.entity.LoggerTest testConfig
6  信息: info
7  10月 21, 2021 11:50:01 上午
   com.ydlclass.entity.LoggerTest testConfig
8  配置: config
9  10月 21, 2021 11:50:01 上午
   com.ydlclass.entity.LoggerTest testConfig
10 详细: fine
11 10月 21, 2021 11:50:01 上午
   com.ydlclass.entity.LoggerTest testConfig
12 较详细: finer
13 10月 21, 2021 11:50:01 上午
   com.ydlclass.entity.LoggerTest testConfig
14 非常详细: finest
15
16 Process finished with exit code 0
```

文件中也输出了正确的结果：

```
10月 21, 2021 11:50:01 上午 com.ydlclass.entity.LoggerTest testConfig
严重: severe
10月 21, 2021 11:50:01 上午 com.ydlclass.entity.LoggerTest testConfig
警告: warning
10月 21, 2021 11:50:01 上午 com.ydlclass.entity.LoggerTest testConfig
信息: info
10月 21, 2021 11:50:01 上午 com.ydlclass.entity.LoggerTest testConfig
配置: config
10月 21, 2021 11:50:01 上午 com.ydlclass.entity.LoggerTest testConfig
详细: fine
10月 21, 2021 11:50:01 上午 com.ydlclass.entity.LoggerTest testConfig
较详细: finer
10月 21, 2021 11:50:01 上午 com.ydlclass.entity.LoggerTest testConfig
非常详细: finest
```

## 四、Logger之间的父子关系

JUL中Logger之间存在父子关系，这种父子关系通过树状结构存储，JUL在初始化时会创建一个顶层 RootLogger作为所有Logger父Logger，存储上作为树状结构的根节点。父子关系通过名称来关联。默认子Logger会继承父Logger的属性。



所有的logger实例都是由LoggerManager统一管理，不妨我们点进getLogger方法：

```
1 private static Logger demandLogger(String name, String
  resourceBundleName, Class<?> caller) {
2     LogManager manager = LogManager.getLogManager();
3     if (!SystemLoggerHelper.disableCallerCheck) {
4         if (isSystem(caller.getModule())) {
5             return manager.demandSystemLogger(name,
  resourceBundleName, caller);
6         }
7     }
8     return manager.demandLogger(name,
  resourceBundleName, caller);
9     // ends up calling new Logger(name,
  resourceBundleName, caller)
10    // iff the logger doesn't exist already
11 }
```

我们可以看到LogManager是单例的：

```
1 public static LogManager getLogger() {
2     if (manager != null) {
3         manager.ensureLogManagerInitialized();
4     }
5     return manager;
6 }
```

```
1 @Test
2 public void testLogParent() throws Exception {
3     Logger logger1 =
4     Logger.getLogger("com.ydlclass.service");
5     Logger logger2 = Logger.getLogger("com.ydlclass");
6     System.out.println("logger1 = " + logger1);
7     System.out.println("logger1.getParent() = " +
8     logger1.getParent());
9     System.out.println("logger2 = " + logger2);
10    System.out.println("logger2.getParent() = " +
11    logger2.getParent());
12    System.out.println(logger1.getParent() ==
13    logger2);
14 }
15
16 结果:
17 logger1 = java.util.logging.Logger@2b4bac49
18 logger1.getParent() = java.util.logging.Logger@fd07cbb
19 logger2 = java.util.logging.Logger@fd07cbb
20 logger2.getParent() =
21 java.util.logging.LogManager$RootLogger@3571b748
22 true
```

```
1 @Test
2 public void testLogParent() throws Exception {
3     Logger logger1 =
4     Logger.getLogger("com.ydlclass.service");
5     Logger logger2 = Logger.getLogger("com.ydlclass");
```

```
5 // 一、对logger2进行独立的配置
6 // 1.关闭系统默认配置
7 logger2.setUseParentHandlers(false);
8 // 2.创建handler对象
9 ConsoleHandler consoleHandler = new
ConsoleHandler();
10 // 3.创建formatter对象
11 SimpleFormatter simpleFormatter = new
SimpleFormatter();
12 // 4.进行关联
13 consoleHandler.setFormatter(simpleFormatter);
14 logger2.addHandler(consoleHandler);
15 // 5.设置日志级别
16 logger2.setLevel(Level.ALL);
17 consoleHandler.setLevel(Level.ALL);
18 // 测试logger1是否被logger2影响
19 logger1.severe("severe");
20 logger1.warning("warning");
21 logger1.info("info");
22 logger1.config("config");
23 logger1.fine("fine");
24 logger1.finer("finer");
25 logger1.finest("finest");
26 }
```

```
27
28
29 10月 21, 2021 12:45:15 下午
com.ydlclass.entity.LoggerTest testLogParent
30 严重: severe
31 10月 21, 2021 12:45:15 下午
com.ydlclass.entity.LoggerTest testLogParent
32 警告: warning
33 10月 21, 2021 12:45:15 下午
com.ydlclass.entity.LoggerTest testLogParent
34 信息: info
35 10月 21, 2021 12:45:15 下午
com.ydlclass.entity.LoggerTest testLogParent
36 配置: config
37 10月 21, 2021 12:45:15 下午
com.ydlclass.entity.LoggerTest testLogParent
38 详细: fine
```

```
39 10月 21, 2021 12:45:15 下午
    com.ydlclass.entity.LoggerTest testLogParent
40 较详细: finer
41 10月 21, 2021 12:45:15 下午
    com.ydlclass.entity.LoggerTest testLogParent
42 非常详细: finest
43
44 Process finished with exit code 0
45
```

## 五、日志格式化

我们可以独立的实现日志格式化的Formatter，而不使用SimpleFormatter，我们可以做如下处理，最后返回的结果我们可以随意拼写：

```
1 Formatter myFormatter = new Formatter(){
2     @Override
3     public String format(LogRecord record) {
4         return record.getLoggerName()+". "
5         +record.getSourceMethodName() + " " +
6         LocalDateTime.ofInstant(record.getInstant(),
7         ZoneId.systemDefault())+"\r\n"
8         +record.getLevel()+": "
9         +record.getMessage() + "\r\n";
10    }
11};
```

结果为：

```
com.ydlclass.service.testLogParent 2021-10-21T13:11:51.709722400
SEVERE: severe
com.ydlclass.service.testLogParent 2021-10-21T13:11:51.724682300
WARNING: warning
com.ydlclass.service.testLogParent 2021-10-21T13:11:51.724682300
INFO: info
```

当然我们参考一下SimpleFormatter的该方法的实现：

```
1 // format string for printing the log record
```

```
2 static String getLoggingProperty(String name) {
3     return
    LogManager.getLogManager().getProperty(name);
4 }
5
6 private final String format =
7
8     SurrogateLogger.getSimpleFormat(SimpleFormatter::getL
    oggingProperty);
9
10    ZonedDateTime zdt = ZonedDateTime.ofInstant(
11        record.getInstant(), ZoneId.systemDefault());
12    return String.format(format,
13                        zdt,
14                        source,
15                        record.getLoggerName(),
16                        record.getLevel().getLocalizedLevelName(),
17                        message,
18                        throwable);
```

这个写法貌似比我们的写法高级一点，所以我们必须好好学一下String的format方法了。

## 1、String的format方法

String类的format()方法用于创建格式化的字符串以及连接多个字符串对象。

format()方法有两种重载形式：



```
1 public static String format(String format, Object...  
  args) {  
2     return new Formatter().format(format,  
  args).toString();  
3 }  
4  
5 public static String format(Locale l, String format,  
  Object... args) {  
6     return new Formatter(l).format(format,  
  args).toString();  
7 }
```

在这个方法中我们可以定义字符串模板，然后使用类似填空的方式将模板格式化为我们想要的结果字符串：

```
1 String java = String.format("hello %s", "world");
```

得到的结果就是hello world，我们可以把第一个参数当做模板， %s当做填空题，后边的可变参数当做答案。

## 2、常用的转换符

当然不同数据类型需要不同转换符完成字符串的转换，以下是不同类型的转化符列表：

转换符	详细说明	示例
<b>%s</b>	<b>字符串类型</b>	<b>“喜欢请收藏”</b>
%c	字符类型	‘m’
%b	布尔类型	true
<b>%d</b>	<b>整数类型（十进制）</b>	<b>88</b>
%x	整数类型（十六进制）	FF
%o	整数类型（八进制）	77
<b>%f</b>	<b>浮点类型</b>	<b>8.888</b>
%a	十六进制浮点类型	FF.35AE
%e	指数类型	9.38e+5
<b>%n</b>	换行符	
%tx	日期与时间类型（x代表不同的日期与时间转换符）	后边详细说

小例子：

1	System.out.printf("过年了，%s今年%d岁了，今天收了%f元的压岁钱！",
2	"小明",5,88.88);
3	
4	结果：
5	过年了，小明今年5岁了，今天收了88.880000元的压岁钱！

这要比拼写字符串简单多了。

### 3、特殊符号

接下来我们看几个特殊字符的常用搭配，可以实现一些高级功能：

标志	说明	示例	结果
+	为正数或者负数添加符号，因为一般整数不会主动加符号	("%+d",15)	+15
0	数字前面补0，用于对齐	("%04d",99)	0099
空格	在整数之前添加指定数量的空格	("%4d",99)	99
,	以","对数字分组(常用显示金额)	("%,f",9999.99)	9,999.990000
(	使用括号包含负数	("%(f",-99.99)	(99.990000)

```
1 System.out.printf("过年了，%s今年%03d岁了，今天收了%,f元的压岁钱!",
2     "小明",5,8888.88);
3     结果
4     过年了，小明今年005岁了，今天收了8,888.880000元的压岁钱!
```

默认情况下，我们的可变参数是安装顺序依次替换，但是我想重复利用可变参数那该怎么处理呢？

我们可以采用 在转换符中加 `数字$` 完成匹配：

```
1 System.out.printf("%1$s %1$s %1$s","小明");
```

其中1\$就代表第一个参数，那么2\$就代表第二个参数了：

```
1 小明 小明 小明
```

#### 4、日期处理

第一个例子中有说到 %t x代表日期转换符 我也顺便列举下日期转换符

标志	说明	示例
c	包括全部日期和时间信息	周四 10月 21 14:52:10 GMT+08:00 2021
F	“年-月-日”格式	2021-10-21
D	“月/日/年”格式	10/21/21
r	“HH:MM:SS PM”格式（12时制）	02:53:20 下午
T	“HH:MM:SS”格式（24时制）	14:53:39
R	“HH:MM”格式（24时制）	14:53
b	月份本地化	10月
y	两位的年	21
Y	四位的年	2021
m	月	10
d	日	21
H	24小时制的时	14
I	12小时制的时	2
M	分	57
S	秒	46
s	秒为单位的时间戳	1634799527
p	上午还是下午	下午

我们可以使用以下三个类去进行格式化，其中可能存在不支持的情况，比如LocalDateTime不支持c：

```
1 System.out.printf("%tc",new Date());
2 System.out.printf("%tc",ZonedDateTime.now());
3 System.out.printf("%tF",LocalDateTime.now());
```

此时我们使用debug查看，默认情况下的format，我们不妨来读一读：

```
> format = "%1$tb %1$td, %1$tY %1$tL:%1$tM:%1$tS %1$Tp %2$s%n%4$s: %5$s%6$s%n"
```

```
1 10月 21, 2021 2:23:42 下午
  com.ydlclass.entity.LoggerTest testLogParent
2 警告: warning
```

## 六、配置文件

我们不妨看看一个文件处理器的源码是怎么读配置项的：

```
1 private void configure() {
2     LogManager manager =
LogManager.getLogManager();
3
4     String cname = getClass().getName();
5
6     pattern = manager.getStringProperty(cname +
".pattern", "%h/java%u.log");
7     limit = manager.getLongProperty(cname +
".limit", 0);
8     if (limit < 0) {
9         limit = 0;
10    }
11    count = manager.getIntProperty(cname +
".count", 1);
12    if (count <= 0) {
13        count = 1;
14    }
```

```

15         append = manager.getBooleanProperty(cname +
16         ".append", false);
17         setLevel(manager.getLevelProperty(cname +
18         ".level", Level.ALL));
19         setFilter(manager.getFilterProperty(cname +
20         ".filter", null));
21
22         setFormatter(manager.getFormatterProperty(cname +
23         ".formatter", new XMLFormatter()));
24         // Initialize maxLocks from the
25         logging.properties file.
26         // If invalid/no property is provided 100 will
27         be used as a default value.
28         maxLocks = manager.getIntProperty(cname +
29         ".maxLocks", MAX_LOCKS);
30         if(maxLocks <= 0) {
31             maxLocks = MAX_LOCKS;
32         }
33         try {
34
35             setEncoding(manager.getStringProperty(cname
36             + ".encoding", null));
37             } catch (Exception ex) {
38                 try {
39                     setEncoding(null);
40                 } catch (Exception ex2) {
41                     // doing a setEncoding with null
42                     should always work.
43                     // assert false;
44                 }
45             }
46         }
47     }

```

可以从以下源码中看到配置项：

```

1 public class FileHandler extends StreamHandler {
2     private MeteredStream meter;
3     private boolean append;
4     // 限制文件大小
5     private long limit;          // zero => no limit.
6     // 控制日志文件的数量

```



```

7     private int count;
8     // 日志文件的格式化方式
9     private String pattern;
10    private String lockFileName;
11    private FileChannel lockFileChannel;
12    private File files[];
13    private static final int MAX_LOCKS = 100;
14    // 可以理解为同时可以有多少个线程打开文件，源码中有介绍
15    private int maxLocks = MAX_LOCKS;
16    private static final Set<String> locks = new
    HashSet<>();
17 }

```

我们已经知道系统默认的配置文件的位置，那我们能不能自定义呢？当然可以了，我们从jdk中赋值一个配置文件过来：

```

1  .level= INFO
2
3  # default file output is in user's home directory.
4  java.util.logging.FileHandler.pattern = %h/java%u.log
5  java.util.logging.FileHandler.limit = 50000
6  java.util.logging.FileHandler.count = 1
7
8  # Default number of locks FileHandler can obtain
   synchronously.
9  # This specifies maximum number of attempts to obtain
   lock file by FileHandler
10 # implemented by incrementing the unique field %u as
   per FileHandler API documentation.
11 java.util.logging.FileHandler.maxLocks = 100
12 java.util.logging.FileHandler.formatter =
   java.util.logging.XMLFormatter
13
14 # Limit the message that are printed on the console to
   INFO and above.
15 java.util.logging.ConsoleHandler.level = INFO
16 java.util.logging.ConsoleHandler.formatter =
   java.util.logging.SimpleFormatter
17
18 # java.util.logging.SimpleFormatter.format=%4$s: %5$s
   [%1$tc]%n

```

```
1 | pattern = manager.getStringProperty(cname + ".pattern",
   | "%h/java%u.log");
```

```
1 | static File generate(String pat, int count, int
   | generation, int unique)
2 |         throws IOException
3 | {
4 |     Path path = Paths.get(pat);
5 |     Path result = null;
6 |     boolean sawg = false;
7 |     boolean sawu = false;
8 |     StringBuilder word = new StringBuilder();
9 |     Path prev = null;
10 |    for (Path elem : path) {
11 |        if (prev != null) {
12 |            prev =
13 |            prev.resolveSibling(word.toString());
14 |            result = result == null ? prev :
15 |            result.resolve(prev);
16 |        }
17 |        String pattern = elem.toString();
18 |        int ix = 0;
19 |        word.setLength(0);
20 |        while (ix < pattern.length()) {
21 |            char ch = pattern.charAt(ix);
22 |            ix++;
23 |            char ch2 = 0;
24 |            if (ix < pattern.length()) {
25 |                ch2 =
26 |                Character.toLowerCase(pattern.charAt(ix));
27 |            }
28 |            if (ch == '%') {
29 |                if (ch2 == 't') {
30 |                    String tmpDir =
31 |                    System.getProperty("java.io.tmpdir");
32 |                    if (tmpDir == null) {
```

```

29         tmpDir =
System.getProperty("user.home");
30     }
31     result = Paths.get(tmpDir);
32     ix++;
33     word.setLength(0);
34     continue;
35     } else if (ch2 == 'h') {
36         result =
Paths.get(System.getProperty("user.home"));
37         if
(jdk.internal.misc.VM.isSetUID()) {
38             // ok, we are in a set UID
program. For safety's sake
39             // we disallow attempts to
open files relative to %h.
40             throw new IOException("can't
use %h in set UID program");
41         }
42         ix++;
43         word.setLength(0);
44         continue;
45     } else if (ch2 == 'g') {
46         word = word.append(generation);
47         sawg = true;
48         ix++;
49         continue;
50     } else if (ch2 == 'u') {
51         word = word.append(unique);
52         sawu = true;
53         ix++;
54         continue;
55     } else if (ch2 == '%') {
56         word = word.append('%');
57         ix++;
58         continue;
59     }
60 }
61 word = word.append(ch);
62 }
63 prev = elem;

```

```

64     }
65
66     if (count > 1 && !sawg) {
67         word = word.append('.').append(generation);
68     }
69     if (unique > 0 && !sawu) {
70         word = word.append('.').append(unique);
71     }
72     if (word.length() > 0) {
73         String n = word.toString();
74         Path p = prev == null ? Paths.get(n) :
prev.resolveSibling(n);
75         result = result == null ? p :
result.resolve(p);
76     } else if (result == null) {
77         result = Paths.get("");
78     }
79
80     if (path.getRoot() == null) {
81         return result.toFile();
82     } else {
83         return
path.getRoot().resolve(result).toFile();
84     }
85 }

```

```

1 | System.out.println(System.getProperty("user.home") );

```

C:\Users\zn\java0.log

我们将拷贝的文件稍作修改：

```

1 | .level= INFO
2
3 | # default file output is in user's home directory.
4 | java.util.logging.FileHandler.pattern =
D:/log/java%u.log
5 | java.util.logging.FileHandler.limit = 50000
6 | java.util.logging.FileHandler.count = 1
7 | java.util.logging.FileHandler.maxLocks = 100

```

```
8 java.util.logging.FileHandler.formatter =
  java.util.logging.XMLFormatter
9
10 # Limit the message that are printed on the console to
    INFO and above.
11 java.util.logging.ConsoleHandler.level = INFO
12 java.util.logging.ConsoleHandler.formatter =
    java.util.logging.SimpleFormatter
13
14 # java.util.logging.SimpleFormatter.format=%4$s: %5$s
    [%1$tc]%n
```

```
1 @Test
2 public void testProperties() throws Exception {
3     // 读取自定义配置文件
4     InputStream in =
5
6         JULTest.class.getClassLoader().getResourceAsStream("l
    ogging.properties");
7     // 获取日志管理器对象
8     LogManager logManager =
9     LogManager.getLogManager();
10    // 通过日志管理器加载配置文件
11    logManager.readConfiguration(in);
12    Logger logger =
13    Logger.getLogger("com.ydlclass.log.JULTest");
14    logger.severe("severe");
15    logger.warning("warning");
16    logger.info("info");
17    logger.config("config");
18    logger.fine("fine");
19    logger.finer("finer");
20    logger.finest("finest");
21 }
```

配置文件:

```

1 handlers=
  java.util.logging.ConsoleHandler,java.util.logging.FileHandler
2 .level= INFO
3
4 java.util.logging.FileHandler.pattern =
  D:/logs/java%u.log
5 java.util.logging.FileHandler.limit = 50000
6 java.util.logging.FileHandler.count = 1
7 java.util.logging.FileHandler.maxLocks = 100
8 java.util.logging.FileHandler.formatter =
  java.util.logging.XMLFormatter
9
10 # Limit the message that are printed on the console to
  INFO and above.
11 java.util.logging.ConsoleHandler.level = INFO
12 java.util.logging.ConsoleHandler.formatter =
  java.util.logging.SimpleFormatter

```

文件中也出现了：



打开日志发现是xml，因为这里用的就是XMLFormatter：

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2021-10-21T08:15:16.998475800Z</date>
  <millis>1634804116998</millis>
  <nanos>475800</nanos>
  <sequence>0</sequence>
  <logger>com.ydlclass</logger>
  <level>SEVERE</level>
  <class>com.ydlclass.entity.LoggerTest</class>
  <method>testProperties</method>
  <thread>1</thread>
  <message>severe</message>
</record>

```



上边我们配置了两个handler给根Logger，我们还可以给其他的Logger做独立的配置：

```
1 handlers = java.util.logging.ConsoleHandler
2 .level = INFO
3 # 对这个logger独立配置
4 com.ydlclass.handlers = java.util.logging.FileHandler
5 com.ydlclass.level = ALL
6 com.ydlclass.useParentHandlers = false
7
8 # 修改了名字
9 java.util.logging.FileHandler.pattern = D:/logs/ydl-
  java%u.log
10 java.util.logging.FileHandler.limit = 50000
11 java.util.logging.FileHandler.count = 1
12 java.util.logging.FileHandler.maxLocks = 100
13 java.util.logging.FileHandler.formatter =
  java.util.logging.SimpleFormatter
14 # 文件使用追加方式
15 java.util.logging.FileHandler.append = true
16
17 # Limit the message that are printed on the console to
  INFO and above.
18 java.util.logging.ConsoleHandler.level = INFO
19 java.util.logging.ConsoleHandler.formatter =
  java.util.logging.SimpleFormatter
20
21 # 修改日志格式
22 java.util.logging.SimpleFormatter.format=%4$s: %5$s
  [%1$tc]%n
```

执行发现控制台没有内容，文件中有了，说明没有问题OK了：

名称	修改日期	类型	大小
java0.log	2021.10.21 16:08	文本文档	2 KB
ydl-java0.log	2021.10.21 16:15	文本文档	3 KB

日志出现以下内容：

```
严重: severe [周四 10月 21 16:18:24 GMT+08:00 2021]
警告: warning [周四 10月 21 16:18:24 GMT+08:00 2021]
信息: info [周四 10月 21 16:18:24 GMT+08:00 2021]
配置: config [周四 10月 21 16:18:24 GMT+08:00 2021]
详细: fine [周四 10月 21 16:18:24 GMT+08:00 2021]
较详细: finer [周四 10月 21 16:18:24 GMT+08:00 2021]
非常详细: finest [周四 10月 21 16:18:24 GMT+08:00 2021]
```

## 第三章 LOG4J 日志框架

Log4j是Apache下的一款开源的日志框架。官方网站: <http://logging.apache.org/log4j/1.2/>, 这是一款比较老的日志框架, 目前新的log4j2做了很大的改动, 任然有一些项目在使用log4j。

### 一、入门案例

1. 建立maven工程
2. 添加依赖

```
1  <dependencies>
2      <dependency>
3          <groupId>log4j</groupId>
4          <artifactId>log4j</artifactId>
5          <version>1.2.17</version>
6      </dependency>
7      <dependency>
8          <groupId>junit</groupId>
9          <artifactId>junit</artifactId>
10         <version>4.13.2</version>
11     </dependency>
12 </dependencies>
13
14 <build>
15     <plugins>
16         <plugin>
17
18             <groupId>org.apache.maven.plugins</groupId>
19             <artifactId>maven-compiler-
plugin</artifactId>
20             <version>3.8.1</version>
21             <configuration>
                <source>${maven.compiler.source}
            </source>
```

```

22         <target>${maven.compiler.target}
    </target>
23         <encoding>UTF-8</encoding>
24     </configuration>
25 </plugin>
26 </plugins>
27 </build>

```

### 3. java代码

```

1  @Test
2  public void testLogger() {
3      Logger logger = Logger.getLogger(Log4jTest.class);
4      // 日志记录输出
5      logger.info("hello log4j");
6      // 日志级别
7      logger.fatal("fatal"); // 严重错误，一般会造成系统崩溃
      和终止运行
8      logger.error("error"); // 错误信息，但不会影响系统运行
9      logger.warn("warn"); // 警告信息，可能会发生问题
10     logger.info("info"); // 程序运行信息，数据库的连接、网
      络、IO操作等
11     logger.debug("debug"); // 调试信息，一般在开发阶段使
      用，记录程序的变量、参数等
12     logger.trace("trace"); // 追踪信息，记录程序的所有流程
      信息
13 }

```

发现会有一些警告，JUL可以直接在控制台输出是因为他有默认的配置文  
件，而这个独立的第三方的日志框架却没有配置文件：

```

1  log4j:WARN No appenders could be found for logger
    (com.ydlclass.entity.Log4jTest).
2  log4j:WARN Please initialize the log4j system properly.
3  log4j:WARN See
    http://logging.apache.org/log4j/1.2/faq.html#noconfig
    for more info.

```

我们在执行代码之前，加上以下代码，他会初始化一个默认配置：

```

1  BasicConfigurator.configure();

```

结果：

```
1 0 [main] INFO com.ydlclass.entity.Log4jTest - hello
  log4j
2 1 [main] FATAL com.ydlclass.entity.Log4jTest - fatal
3 1 [main] ERROR com.ydlclass.entity.Log4jTest - error
4 1 [main] WARN com.ydlclass.entity.Log4jTest - warn
5 1 [main] INFO com.ydlclass.entity.Log4jTest - info
6 1 [main] DEBUG com.ydlclass.entity.Log4jTest - debug
```

从源码看，这一行代码给我们的RootLogger加入一个控制台的输出源，就和jul中的handler一样：

```
1 public static void configure() {
2     Logger root = Logger.getRootLogger();
3     root.addAppender(new ConsoleAppender(new
  PatternLayout("%r [%t] %p %c %x - %m%n")));
4 }
```

log4j定义了以下的日志的级别，和JUL的略有不同：

1. fatal 指出每个严重的错误事件将会导致应用程序的退出。
2. error 指出虽然发生错误事件，但仍然不影响系统的继续运行。
3. warn 表明会出现潜在的错误情形。
4. info 一般和在粗粒度级别上，强调应用程序的运行全程。
5. debug 一般用于细粒度级别上，对调试应用程序非常有帮助。
6. trace 是程序追踪，可以用于输出程序运行中的变量，显示执行的流程。

和JUL一样：还有两个特殊的级别：OFF，可用来关闭日志记录。ALL，启用所有消息的日志记录。

一般情况下，我们只使用4个级别，优先级从高到低为 ERROR > WARN > INFO > DEBUG。

## 二、组件讲解

Log4j 主要由 Loggers (日志记录器)、Appenders (输出端) 和 Layout (日志格式化器) 组成。其中 Loggers 控制日志的输出级别与日志是否输出；Appenders 指定日志的输出方式（输出到控制台、文件 等）；Layout 控制日志信息的输出格式。

## 1、Loggers

日志记录器：负责收集处理日志记录，实例的命名就是类“XX”的 full qualified name（类的全限定名），Logger 的名字大小写敏感，其命名有继承机制：例如：name 为 com.ydlclass.service 的 logger 会继承 name 为 com.ydlclass 的 logger，和 JUL 一致。

Log4j 中有一个特殊的 logger 叫做“root”，他是所有 logger 的根，也就意味着其他所有的 logger 都会直接 或者间接地继承自 root。root logger 可以用 Logger.getRootLogger() 方法获取。JUL 是不是也有一个名为 `java.util.logging.Logger` 的根。

## 2、Appenders

Appender 和 JUL 的 Handler 很像，用来指定日志输出到哪个地方，可以同时指定日志的输出目的地。Log4j 常用的输出目的地 有以下几种：

输出端类型	作用
ConsoleAppender	将日志输出到控制台
FileAppender	将日志输出到文件中
DailyRollingFileAppender	将日志输出到一个日志文件，并且每天输出到一个新的文件
RollingFileAppender	将日志信息输出到一个日志文件，并且指定文件的尺寸，当文件大小达到指定尺寸时，会自动把文件改名，同时产生一个新的文件
JDBCAppender	把日志信息保存到数据库中

```

1 // 配置一个控制台输出源
2 ConsoleAppender consoleAppender = new
  ConsoleAppender();
3 consoleAppender.setName("yd1");
4 consoleAppender.setWriter(new PrintWriter(System.out));
5 logger.addAppender(consoleAppender);

```

### 3、Layouts

```

1 Layout layout = new Layout() {
2     @Override
3     public String format(LoggingEvent loggingEvent) {
4         return loggingEvent.getLoggerName() + " "
5             + loggingEvent.getMessage() + "\r\n";
6     }
7
8     @Override
9     public boolean ignoresThrowable() {
10        return false;
11    }
12
13    @Override
14    public void activateOptions() {
15
16    }
17 };

```

有一些默认的实现类：

```

1 Layout layout = new SimpleLayout();

```

```

"C:\Program Files\Java\jdk-11.0.4_windows-x64_bin\jdk-11.0.4\bin\java.exe"
INFO - hello log4j
FATAL - fatal
ERROR - error
WARN - warn

```

他的实现太简单了：



```

1 public String format(LoggingEvent event) {
2     sbuf.setLength(0);
3     sbuf.append(event.getLevel().toString());
4     sbuf.append(" - ");
5     sbuf.append(event.getRenderedMessage());
6     sbuf.append(LINE_SEP);
7     return sbuf.toString();
8 }

```

还有一个比较常用的Layout，就是PatternLayout这个实现类，能够根据特定的占位符进行转化，和JUL很像，但是又不一样，我们庖丁解牛研究一番，首先看他的构造器，构造器中如果传入一个pattern字符串，他会根据这个pattern创建一个链表，这个链表具体干什么咱们慢慢往后看：

```

1 public PatternLayout(String pattern) {
2     this.pattern = pattern;
3     head = createPatternParser((pattern == null) ?
    DEFAULT_CONVERSION_PATTERN :
4     pattern).parse();
5 }

```

将步骤拆解开来看，首先创建了一个解析器：

```

1 protected PatternParser createPatternParser(String
    pattern) {
2     return new PatternParser(pattern);
3 }

```

查看parse方法，这个方法比较复杂我们简化来看：

```

1 public PatternConverter parse() {
2     char c;
3     i = 0;
4     while(i < patternLength) {
5         ...此次省略了很多代码，但是可以从这个核心看出来
6         c = pattern.charAt(i++);
7         finalizeConverter(c);
8     }
9     return head;
10 }

```

而finalizeConverter做的工作大家就能看的很清楚了：

```

1 protected void finalizeConverter(char c) {
2     PatternConverter pc = null;
3     switch(c) {
4         case 'c':
5             pc = new
6             CategoryPatternConverter(formattingInfo,
7                                     extractPrecisionOption());
8             //LogLog.debug("CATEGORY converter.");
9             //formattingInfo.dump();
10            currentLiteral.setLength(0);
11            break;
12            //处理类名的转化器
13        case 'C':
14            pc = new
15            ClassNamePatternConverter(formattingInfo,
16                                    extractPrecisionOption());
17            //LogLog.debug("CLASS_NAME converter.");
18            //formattingInfo.dump();
19            currentLiteral.setLength(0);
20            break;
21            //处理时间的转化器
22        case 'd':
23            String dateFormatStr =
24            AbsoluteTimeDateFormat.ISO8601_DATE_FORMAT;
25            DateFormat df;
26            ...
27            pc = new DatePatternConverter(formattingInfo,
28                                          df);

```

```
25     currentLiteral.setLength(0);
26     break;
27     //输出日志时间发生的位置，包括类名、线程、及在代码中的行数
28     case 'F':
29         pc = new
LocationPatternConverter(formattingInfo,
30                             FILE_LOCATION_CONVERTER);
31         break;
32     case 'l':
33         pc = new
LocationPatternConverter(formattingInfo,
34                             FULL_LOCATION_CONVERTER);
35         currentLiteral.setLength(0);
36         break;
37     case 'L':
38         pc = new
LocationPatternConverter(formattingInfo,
39                             LINE_LOCATION_CONVERTER);
40         currentLiteral.setLength(0);
41         break;
42     case 'm':
43         pc = new BasicPatternConverter(formattingInfo,
MESSAGE_CONVERTER);
44         currentLiteral.setLength(0);
45         break;
46     case 'M':
47         pc = new
LocationPatternConverter(formattingInfo,
48                             METHOD_LOCATION_CONVERTER);
49         currentLiteral.setLength(0);
50         break;
51     case 'p':
52         pc = new BasicPatternConverter(formattingInfo,
LEVEL_CONVERTER);
53         currentLiteral.setLength(0);
54         break;
55     case 'r':
56         pc = new BasicPatternConverter(formattingInfo,
57                                     RELATIVE_TIME_CONVERTER);
58         currentLiteral.setLength(0);
59         break;
```

```

60     case 't':
61         pc = new BasicPatternConverter(formattingInfo,
        THREAD_CONVERTER);
62         currentLiteral.setLength(0);
63         break;
64     case 'x':
65         pc = new BasicPatternConverter(formattingInfo,
        NDC_CONVERTER);
66         //LogLog.debug("NDC converter.");
67         currentLiteral.setLength(0);
68         break;
69     case 'x':
70         String xopt = extractOption();
71         pc = new MDCPatternConverter(formattingInfo,
        xopt);
72         currentLiteral.setLength(0);
73         break;
74     default:
75         LogLog.error("Unexpected char [" + c + "] at
        position " + i
76             + " in conversion pattern.");
77         pc = new
        LiteralPatternConverter(currentLiteral.toString());
78         currentLiteral.setLength(0);
79     }
80
81     addConverter(pc);
82 }

```

下边就是一个典型的链表结构的构建了：

```

1  protected void addConverter(PatternConverter pc) {
2      currentLiteral.setLength(0);
3      // Add the pattern converter to the list.
4      addToList(pc);
5      // Next pattern is assumed to be a literal.
6      state = LITERAL_STATE;
7      // Reset formatting info
8      formattingInfo.reset();
9  }

```

```

1 private void addToList(PatternConverter pc) {
2     if(head == null) {
3         head = tail = pc;
4     } else {
5         tail.next = pc;
6         tail = pc;
7     }
8 }

```

构建完转化器链表之后，就是循环这个链表，一次处理对应的占位符了，他的核心的格式化的方法也是format方法，在format方法中是通过一个转化器链来完成转化的：

```

1 public String format(LoggingEvent event) {
2     // 在format方法中是通过一个转化器链来完成转化的
3     PatternConverter c = head;
4
5     while(c != null) {
6         // 这一句是核心，第一个参数是一个StringBuilder，第二个
        // 参数LoggingEvent
7         c.format(sbuf, event);
8         c = c.next;
9     }
10    return sbuf.toString();
11 }
12 }

```

这里就是通过一个pattern字符串，这个字符串可能长这个样子（%-d{yyyy-MM-dd HH:mm:ss} [%t:%r] -[%p] %m%n），使用createPatternParser().parse()方法构建一个处理器的链表，这个每个处理器处理一个占位符比如（%d）。

进入c.format()方法，我们会进入一个抽象类PatternConverter中的format方法，里边的核心就是如下代码：

```

1 public void format(StringBuffer sbuf, LoggingEvent e) {
2     // 核心就是这一句
3     String s = convert(e);
4 }

```

log4j 其实采用类似 C 语言的 printf 函数的打印格式格式化日志信息，源码已经看过了，具体的占位符及其含义如下：

- 1 | %m 输出代码中指定的日志信息
- 2 | %p 输出日志级别，及 DEBUG、INFO 等
- 3 | %n 换行符（windows平台的换行符为 "\n"，Unix 平台为 "\n"）
- 4 | %r 输出自应用启动到输出该 log 信息耗费的毫秒数
- 5 | %c 输出打印语句所属的类的全名
- 6 | %t 输出产生该日志的线程全名
- 7 | %d 输出服务器当前时间，默认为 ISO8601，也可以指定格式，如：  
%d{yyyy年MM月dd日HH:mm:ss}
- 8 | %l 输出日志时间发生的位置，包括类名、线程、及在代码中的行数。如：  
Test.main(Test.java:10)
- 9 | %F 输出日志消息产生时所在的文件名称
- 10 | %L 输出代码中的行号
- 11 | %% 输出一个 "%" 字符
- 12 | \* 可以在 % 与字符之间加上修饰符来控制最小宽度、最大宽度和文本的对齐方式。如：
- 13 | %5c 输出category名称，最小宽度是5，category<5，默认的情况下右对齐
- 14 | %-5c 输出category名称，最小宽度是5，category<5， "-"号指定左对齐，会有空格
- 15 | %.5c 输出category名称，最大宽度是5，category>5，就会将左边多出的字符截掉，<5不会有空格
- 16 | %20.30c category名称<20补空格，并且右对齐，>30字符，就从左边交远销出的字符截掉

举一个例子：

- 1 | %-d{yyyy-MM-dd HH:mm:ss} [%t:%r] -[%p] %m%n
- 2 | 打印：日期 [线程:毫秒数] - [日志级别] - 日志信息 换行

尝试写一个：

```
1 | @Test
2 | public void testLog(){
3 |     // 获取一个logger
4 |     Logger logger = Logger.getLogger(TestLog4j.class);
5 |     // 创建一个layout
6 |     Layout layout = new PatternLayout("%-d{yyyy-MM-dd
HH:mm:ss} [%t:%r] -[%p] %m%n");
```

```

7      // 创建一个输出源
8      ConsoleAppender appender = new ConsoleAppender();
9      appender.setLayout(layout);
10     appender.setWriter(new PrintWriter(System.out));
11     logger.addAppender(appender);
12     logger.warn("warning");
13 }
14
15 结果:
16 2021-10-21 21:31:05 [main:0] -[WARN] warning

```

### 配置一个jdbcAppender

```

1  JDBCAppender jdbcAppender = new JDBCAppender();
2  jdbcAppender.setDriver("com.mysql.cj.jdbc.Driver");
3  jdbcAppender.setURL("jdbc:mysql://localhost:3306/ydlclass?
   characterEncoding=utf8&useSSL=false&serverTimezone=UTC"
   );
4  jdbcAppender.setUser("root");
5  jdbcAppender.setPassword("root");
6  jdbcAppender.setSql("INSERT INTO
   log(project_name,create_date,level,category,file_name,t
   hread_name,line,all_category,message)
   values('ydlclass','%d{yyyy-MM-
   ddHH:mm:ss}','%p','%c','%F','%t','%L','%l','%m')");
7

```

### 数据表

```

1 CREATE TABLE `log` (
2     `log_id` int(11) NOT NULL AUTO_INCREMENT,
3     `project_name` varchar(255) DEFAULT NULL COMMENT
    '目项名',
4     `create_date` varchar(255) DEFAULT NULL COMMENT
    '创建时间',
5     `level` varchar(255) DEFAULT NULL COMMENT '优先级',
6     `category` varchar(255) DEFAULT NULL COMMENT '所在
    类的全名',
7     `file_name` varchar(255) DEFAULT NULL COMMENT '输出
    日志消息产生时所在的文件名称 ',
8     `thread_name` varchar(255) DEFAULT NULL COMMENT
    '日志事件的线程名',
9     `line` varchar(255) DEFAULT NULL COMMENT '号行',
10    `all_category` varchar(255) DEFAULT NULL COMMENT
    '日志事件的发生位置',
11    `message` varchar(4000) DEFAULT NULL COMMENT '输出
    代码中指定的消息',
12    PRIMARY KEY (`log_id`)
13 );

```

## 依赖

```

1 <dependency>
2     <groupId>mysql</groupId>
3     <artifactId>mysql-connector-java</artifactId>
4     <version>8.0.22</version>
5 </dependency>

```

## 三、配置

log4j不仅仅可以在控制台，文件文件中输出日志，甚至可以在数据库中，我们先使用配置的方式完成日志的输入：

```

1 #指定日志的输出级别与输出端
2 log4j.rootLogger=INFO,Console,ydl
3 # 控制台输出配置

```



```

4 log4j.appender.Console=org.apache.log4j.ConsoleAppender
5 log4j.appender.Console.layout=org.apache.log4j.Pattern
  Layout
6 log4j.appender.Console.layout.ConversionPattern=%d
  [%t] %-5p [%c] - %m%n
7 # 文件输出配置
8 log4j.appender.ydl =
  org.apache.log4j.DailyRollingFileAppender
9 #指定日志的输出路径
10 log4j.appender.ydl.File = D:/logs/ydl.log
11 log4j.appender.ydl.Append = true
12 #使用自定义日志格式化器
13 log4j.appender.ydl.layout =
  org.apache.log4j.PatternLayout
14 #指定日志的输出格式
15 log4j.appender.ydl.layout.ConversionPattern = %-
  d{yyyy-MM-dd HH:mm:ss} [%t:%r] -[%p] %m%n
16 #指定日志的文件编码
17 log4j.appender.ydl.encoding=UTF-8

```

有了这个配置文件我们些代码就简单了一些：

```

1 @Test
2 public void testConfig(){
3     // 获取一个logger
4     Logger logger = Logger.getLogger(TestLog4j.class);
5     logger.warn("warning");
6 }
7
8 结果：
9     2021-10-21 21:37:06,705 [main] WARN
    [com.ydlclass.TestLog4j] - warning

```

文件也有了：

名称	修改日期	类型	大小
 ydl.log	2021/10/21 21:37	文本文档	1 KB

内容：

我们查看了可是确实没有问题。

当然日志配置文件是什么时候读取的呢？每一个logger都是LogManager创建的，而LogManager有一个静态代码块帮助我们解析配置文件，细节就不需要了解了：

```
1 public class LogManager {
2
3     /**
4      * @deprecated This variable is for internal use
5      * only. It will
6      * become package protected in future versions.
7      * */
8     static public final String
9     DEFAULT_CONFIGURATION_FILE = "log4j.properties";
10
11     static final String DEFAULT_XML_CONFIGURATION_FILE =
12     "log4j.xml";
13
14     /**
15      * @deprecated This variable is for internal use
16      * only. It will
17      * become private in future versions.
18      * */
19     static final public String
20     DEFAULT_CONFIGURATION_KEY="log4j.configuration";
21
22     /**
23      * @deprecated This variable is for internal use
24      * only. It will
```

```

25     * become private in future versions.
26     */
27     public static final String DEFAULT_INIT_OVERRIDE_KEY
    =
28     "log4j.defaultInitOverride";
29
30
31     static private Object guard = null;
32     static private RepositorySelector
repositorySelector;
33
34     static {
35         // By default we use a DefaultRepositorySelector
which always returns 'h'.
36         Hierarchy h = new Hierarchy(new
RootLogger((Level) Level.DEBUG));
37         repositorySelector = new
DefaultRepositorySelector(h);
38
39         /** search for the properties file
log4j.properties in the CLASSPATH. */
40         String override
=OptionConverter.getSystemProperty(DEFAULT_INIT_OVERRI
DE_KEY,
41         null);
42
43         // if there is no default init override, then
get the resource
44         // specified by the user or the default config
file.
45         if(override == null ||
>false".equalsIgnoreCase(override)) {
46
47             String configurationOptionStr =
OptionConverter.getSystemProperty(
48                 DEFAULT_CONFIGURATION_KEY,
49                 null);
50

```

```

51         String configuratorClassName =
OptionConverter.getProperty(
52             CONFIGURATOR_CLASS_KEY,
53             null);
54
55         URL url = null;
56
57         // if the user has not specified the
log4j.configuration
58         // property, we search first for the file
"log4j.xml" and then
59         // "log4j.properties"
60         if(configurationOptionStr == null) {
61             url =
Loader.getResource(DEFAULT_XML_CONFIGURATION_FILE);
62             if(url == null) {
63                 url =
Loader.getResource(DEFAULT_CONFIGURATION_FILE);
64             }
65             } else {
66                 try {
67                     url = new
URL(configurationOptionStr);
68                 } catch (MalformedURLException ex) {
69                     // so, resource is not a URL:
70                     // attempt to get the resource from
the class path
71                     url =
Loader.getResource(configurationOptionStr);
72                 }
73             }
74
75         // If we have a non-null url, then delegate
the rest of the
76         // configuration to the
OptionConverter.selectAndConfigure
77         // method.
78         if(url != null) {
79             LogLog.debug("Using URL [" + url + "] for
automatic log4j configuration.");
80             try {

```

```

81     optionConverter.selectAndConfigure(url,
    configuratorClassName,
82     LogManager.getLoggerRepository());
83         } catch (NoClassDefFoundError e) {
84             LogLog.warn("Error during default
initialization", e);
85         }
86     } else {
87         LogLog.debug("Could not find resource:
["+configurationOptionStr+"].");
88     }
89     } else {
90         LogLog.debug("Default initialization of
overridden by " +
91             DEFAULT_INIT_OVERRIDE_KEY +
"property.");
92     }
93 }
94 }

```

还有更有意思的，我们可以直接添加一个数据源，讲日志输出到数据库中，就是一个和数据库链接的输出源而已：

加入一个数据库的日志输出源：

```

1 #mysql
2 log4j.appender.logDB=org.apache.log4j.jdbc.JDBCAppender
3 log4j.appender.logDB.layout=org.apache.log4j.PatternLayout
4 log4j.appender.logDB.Driver=com.mysql.cj.jdbc.Driver
5 log4j.appender.logDB.URL=jdbc:mysql://localhost:3306/ssm
6 log4j.appender.logDB.User=root
7 log4j.appender.logDB.Password=root
8 log4j.appender.logDB.Sql=INSERT INTO
  log(project_name,create_date,level,category,file_name,thread_name,line,all_category,message)
  values('ydlclass','%d{yyyy-MM-ddHH:mm:ss}','%p','%c','%F','%t','%L','%l','%m')

```

需要

```

1 CREATE TABLE `log` (
2   `log_id` int(11) NOT NULL AUTO_INCREMENT,
3   `project_name` varchar(255) DEFAULT NULL COMMENT '目项名',
4   `create_date` varchar(255) DEFAULT NULL COMMENT '创建时间',
5   `level` varchar(255) DEFAULT NULL COMMENT '优先级',
6   `category` varchar(255) DEFAULT NULL COMMENT '所在类的全名',
7   `file_name` varchar(255) DEFAULT NULL COMMENT '输出日志消息产生时所在的文件名称',
8   `thread_name` varchar(255) DEFAULT NULL COMMENT '日志事件的线程名',
9   `line` varchar(255) DEFAULT NULL COMMENT '号行',
10  `all_category` varchar(255) DEFAULT NULL COMMENT '日志事件的发生位置',
11  `message` varchar(4000) DEFAULT NULL COMMENT '输出代码中指定的消息',
12  PRIMARY KEY (`log_id`)
13 );
14

```

pom中添加驱动:

```

1 <dependency>
2     <groupId>mysql</groupId>
3     <artifactId>mysql-connector-java</artifactId>
4     <version>8.0.22</version>
5 </dependency>

```

再次执行：

发现除了控制台，文件，数据库中也有了日志了：

category	file_name	thread_name	line	all_cate...	mess
1 ydlclass...	TestLog4j.java	main	32	com.ydlclass...	warnir

## 四、自定义Logger

```

1 # RootLogger配置
2 log4j.rootLogger = trace,console
3 # 自定义Logger
4 log4j.logger.com.ydlclass= WARN,logDB
5 log4j.logger.org.apache = error

```

由此我们发现，我们可以很灵活的自定义，组装不同logger的实现，接下来我们写代码测试：

```

1 @Test
2 public void testDefineLogger() throws Exception {
3     Logger logger1 =
4     Logger.getLogger(Log4jTest.class);
5     logger1.fatal("fatal"); // 严重错误，一般会造成系统崩溃
6     // 和终止运行
7     logger1.error("error"); // 错误信息，但不会影响系统运行
8     logger1.warn("warn"); // 警告信息，可能会发生问题
9     logger1.info("info"); // 程序运行信息，数据库的连接、网
10    // 络、IO操作等
11    logger1.debug("debug"); // 调试信息，一般在开发阶段使
12    // 用，记录程序的变量、参数等
13    logger1.trace("trace"); // 追踪信息，记录程序的所有流程
14    // 信息

```

```
10 // 自定义 org.apache
11 Logger logger2 = Logger.getLogger(Logger.class);
12 logger2.fatal("fatal logger2"); // 严重错误，一般会造成系统崩溃和终止运行
13 logger2.error("error logger2"); // 错误信息，但不会影响系统运行
14 logger2.warn("warn logger2"); // 警告信息，可能会发生问题
15 logger2.info("info logger2"); // 程序运行信息，数据库的连接、网络、IO操作等
16 logger2.debug("debug logger2"); // 调试信息，一般在开发阶段使用，记录程序的变量、参数等
17 logger2.trace("trace logger2"); // 追踪信息，记录程序的所有流程信息
18 }
19
```

我们发现logger1的日志级别成了warn，并且在数据库中有了日志，logger2级别成了error，他们其实都继承了根logger的一些属性。

## 第四章 日志门面

当我们的系统变的复杂的之后，难免会集成其他的系统，不同的系统之间可能会使用不同的日志系统。那么在一个系统中，我们的日志框架可能会出现多个，会出现混乱，而且随着时间的发展，可能会出现新的效率更高的日志系统，如果我们想切换代价会非常的大。如果我们的日志系统能和jdbc一样，有一套自己的规范，其他实现均按照规范去实现，就能很灵活的使用日志框架了。

日志门面就是为了解决这个问题而出现的一种技术，日志门面是规范，其他的实现按照规范实现各自的日志框架即可，我们程序员基于日志门面编程即可。举个例子：日志门面就好比菜单，日志实现就好比厨师，我们去餐馆吃饭按照菜单点菜即可，厨师是谁其实不重要，但是有一个符合我口味的厨师当然会更好。

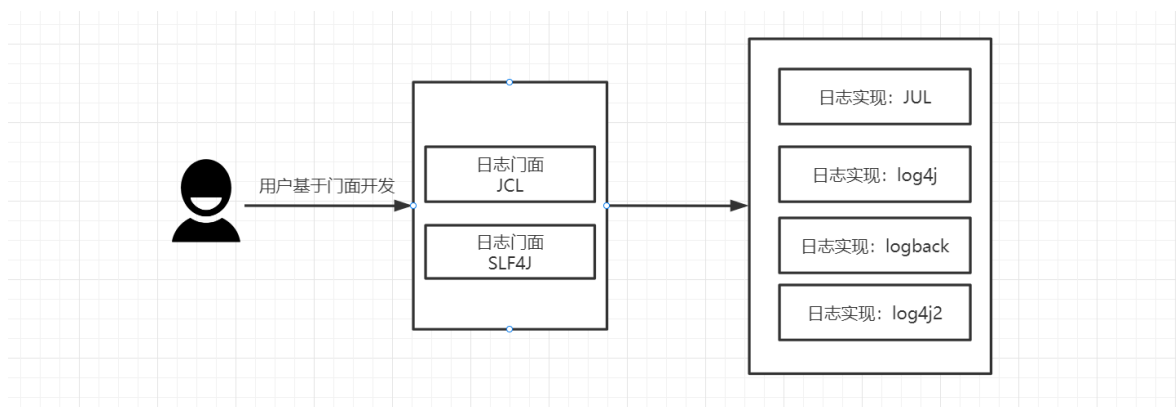
常见的日志门面：JCL、slf4j

常见的日志实现：JUL、log4j、logback、log4j2



日志框架出现的历史顺序：

log4j --> JUL --> JCL --> slf4j --> logback --> log4j2



## 一、SLF4J日志门面

简单日志门面(Simple Logging Facade For Java) SLF4J主要是为了给Java日志访问提供一套标准、规范的API框架，其主要意义在于提供接口，具体的实现可以交由其他日志框架，例如log4j和logback等。当然slf4j自己也提供了功能较为简单的实现，但是一般很少用到。对于一般的Java项目而言，日志框架会选择slf4j-api作为门面，配上具体的实现框架

(log4j、logback等)，中间使用桥接器完成桥接。官方网站：<https://www.slf4j.org/>

SLF4J是目前市面上最流行的日志门面。现在的项目中，基本上都是使用SLF4J作为我们的日志系统。

SLF4J日志门面主要提供两大功能：

1. 日志框架的绑定
2. 日志框架的桥接

### 1、阿里日志规约

1. 应用中不可直接使用日志系统（Log4j、Logback）中的API，而应依赖使用日志框架SLF4J中的API。使用门面模式的日志框架，有利于维护和各个类的日志处理方法统一。
2. 日志文件推荐至少保存15天，因为有些异常具备以“周”为频次发生的特点。
3. 应用中的扩展日志（如打点、临时监控、访问日志等）命名方式：  
appName\_logType\_logName.log。logType为日志类型，推荐分类有stats/monitor/visit 等；
4. logName为日志描述。这种命名的好处：通过文件名就可以知道日志文件属于哪个应用，哪种类型，有什么目的，这也有利于归类查找。
5. 对trace/debug/info级别的日志输出，必须使用条件输出形式或者占位符的方式。
6. 避免重复打印日志，否则会浪费磁盘空间。务必在日志配置文件中设置additivity=false。
7. 异常信息应该包括两类：案发现场信息和异常堆栈信息。如果不处理，那么通过关键字向上抛出。
8. 谨慎地记录日志。生产环境禁止输出debug日志；有选择地输出info日志；如果使用warn记录刚上线时的业务行为信息，一定要注意日志输出量的问题，避免吧服务器磁盘撑爆，并及时删除这些观察日志。
9. 可以使用warn日志级别记录用户输入参数错误的情况，避免当用户投诉时无所适从。

## 2、SLF4J实战

### (1) 添加依赖

```

1 <!--slf4j core 使用slf4j必須添加-->
2 <dependency>
3     <groupId>org.slf4j</groupId>
4     <artifactId>slf4j-api</artifactId>
5     <version>1.7.27</version>
6 </dependency>
7 <!--slf4j 自带的简单日志实现 -->
8 <dependency>
9     <groupId>org.slf4j</groupId>
10    <artifactId>slf4j-simple</artifactId>
11    <version>1.7.27</version>
12 </dependency>

```

## (2) 编写代码

```

1 public class TestSlf4j {
2
3     // 声明日志对象
4     public final static Logger LOGGER =
5         LoggerFactory.getLogger(TestSlf4j.class);
6
7     @Test
8     public void testSlfSimple() {
9         //打印日志信息
10        LOGGER.error("error");
11        LOGGER.warn("warn");
12        LOGGER.info("info");
13        LOGGER.debug("debug");
14        LOGGER.trace("trace");
15        // 使用占位符输出日志信息
16        String name = "lucy";
17        Integer age = 18;
18        LOGGER.info("{}今年{}岁了!", name, age);
19        // 将系统异常信息写入日志
20        try {
21            int i = 1 / 0;
22        } catch (Exception e) {
23            // e.printStackTrace();
24            LOGGER.info("出现异常: ", e);
25        }
26    }
27 }

```

slf4j支持占位符：

### 3、绑定其他日志的实现 (Binding)

如前所述，SLF4J支持各种日志框架。SLF4J发行版附带了几个称为“SLF4J绑定”的jar文件，每个绑定对应一个受支持的框架。

**使用slf4j的日志绑定流程：**

1. 添加slf4j-api的依赖
2. 使用slf4j的API在项目中进行统一的日志记录
3. 绑定具体的日志实现框架
  - a. 绑定已经实现了slf4j的日志框架,直接添加对应依赖
  - b. 绑定没有实现slf4j的日志框架,先添加日志的适配器,再添加实现类的依赖
4. slf4j有且仅有一个日志实现框架的绑定（如果出现多个默认使用第一个依赖日志实现）

绑定jul的实现

```
1 <dependency>
2     <groupId>org.slf4j</groupId>
3     <artifactId>slf4j-api</artifactId>
4     <version>1.7.27</version>
5 </dependency>
6 <dependency>
7     <groupId>org.slf4j</groupId>
8     <artifactId>slf4j-jdk14</artifactId>
9     <version>1.7.25</version>
10 </dependency>
```

绑定log4j的实现

```
1 <!--slf4j core 使用slf4j必須添加-->
2 <dependency>
3     <groupId>org.slf4j</groupId>
4     <artifactId>slf4j-api</artifactId>
5     <version>1.7.27</version>
6 </dependency>
7 <!-- log4j-->
8 <dependency>
9     <groupId>org.slf4j</groupId>
10    <artifactId>slf4j-log4j12</artifactId>
11    <version>1.7.27</version>
12 </dependency>
13 <dependency>
14    <groupId>log4j</groupId>
15    <artifactId>log4j</artifactId>
16    <version>1.2.17</version>
17 </dependency>
```

要切换日志框架，只需替换类路径上的slf4j绑定。例如，要从java.util.logging切换到log4j，只需将slf4j-jdk14-1.7.27.jar替换为slf4j-log4j12-1.7.27.jar即可。

SLF4J不依赖于任何特殊的类装载。实际上，每个SLF4J绑定在编译时都是硬连线的，以使用一个且只有一个特定的日志记录框架。例如，slf4j-log4j12-1.7.27.jar绑定在编译时绑定以使用log4j。

## 4、桥接旧的日志框架（Bridging）

通常，您依赖的某些组件依赖于SLF4J以外的日志记录API。您也可以假设这些组件在不久的将来不会切换到SLF4J。为了解决这种情况，SLF4J附带了几个桥接模块，这些模块将对log4j，JCL和java.util.logging API的调用重定向，就好像它们是对SLF4J API一样。

就是你还用log4j的api写代码，但是具体的实现给你抽离了，我们依赖了一个中间层，这个层其实是用旧的api操作slf4j，而不是操作具体的实现。

桥接解决的是项目中日志的遗留问题，当系统中存在之前的日志API，可以通过桥接转换到slf4j的实现

1. 先去除之前老的日志框架的依赖，必须去掉。
2. 添加SLF4J提供的桥接组件，这个组件就是模仿之前老的日志写了一套相同的api，只不过这个api是在调用slf4j的api。
3. 为项目添加SLF4J的具体实现。

迁移的方式：

```
1  <!-- 桥接的组件 -->
2  <dependency>
3      <groupId>org.slf4j</groupId>
4      <artifactId>log4j-over-slf4j</artifactId>
5      <version>1.7.27</version>
6  </dependency>
7
8  <dependency>
9      <groupId>org.slf4j</groupId>
10     <artifactId>slf4j-api</artifactId>
11     <version>1.7.27</version>
12 </dependency>
13
14 <dependency>
15     <groupId>org.slf4j</groupId>
16     <artifactId>slf4j-simple</artifactId>
17     <version>1.7.27</version>
18 </dependency>
```

SLF4J提供的桥接器：

```
1  <!-- log4j -->
2  <dependency>
3      <groupId>org.slf4j</groupId>
4      <artifactId>log4j-over-slf4j</artifactId>
5      <version>1.7.27</version>
6  </dependency>
7  <!-- jul -->
8  <dependency>
9      <groupId>org.slf4j</groupId>
```

```
10     <artifactId>jul-to-slf4j</artifactId>
11     <version>1.7.27</version>
12 </dependency>
13 <!--jcl -->
14 <dependency>
15     <groupId>org.slf4j</groupId>
16     <artifactId>jcl-over-slf4j</artifactId>
17     <version>1.7.27</version>
18 </dependency>
19
```

注意问题：

1. jcl-over-slf4j.jar和 slf4j-jcl.jar不能同时部署。前一个jar文件将导致JCL将日志系统的选择委托给 SLF4J，后一个jar文件将导致SLF4J将日志系统的选择委托给JCL，从而导致无限循环。
2. log4j-over-slf4j.jar和slf4j-log4j12.jar不能同时出现
3. jul-to-slf4j.jar和slf4j-jdk14.jar不能同时出现
4. 所有的桥接都只对Logger日志记录器对象有效，如果程序中调用了内部的配置类或者是 Appender,Filter等对象，将无法产生效果。

## 5、SLF4J原理解析

1. SLF4J通过LoggerFactory加载日志具体的实现对象。
2. LoggerFactory在初始化的过程中，会通过performInitialization()方法绑定具体的日志实现。
3. 在绑定具体实现的时候，通过类加载器，加载org.slf4j.impl/StaticLoggerBinder.class
4. 所以，只要是一个日志实现框架，在org.slf4j.impl包中提供一个自己的StaticLoggerBinder类，在其中提供具体日志实现的LoggerFactory就可以被SLF4J所加载

在slf4j中创建logger的方法是：

```

1 public static Logger getLogger(String name) {
2     ILoggerFactory iLoggerFactory =
3     getLoggerFactory();
4     return iLoggerFactory.getLogger(name);
5 }

```

继续进入查看，核心就是performInitialization();:

```

1 public static ILoggerFactory getLoggerFactory() {
2     if (INITIALIZATION_STATE == UNINITIALIZED) {
3         synchronized (LoggerFactory.class) {
4             if (INITIALIZATION_STATE == UNINITIALIZED)
5             {
6                 INITIALIZATION_STATE =
7                 ONGOING_INITIALIZATION;
8                 performInitialization();
9             }
10        }
11    }
12 }

```

继续进入查看，核心就是bind()，这个方法应该就能绑定日志实现了：

```

1 private final static void performInitialization() {
2     bind();
3     if (INITIALIZATION_STATE ==
4     SUCCESSFUL_INITIALIZATION) {
5         versionSanityCheck();
6     }
7 }

```

来到这里，看看绑定的方法：

```

1 private final static void bind() {
2     try {
3         ...
4         // 以下内容就绑定成功了
5         StaticLoggerBinder.getSingleton();
6         INITIALIZATION_STATE =
7         SUCCESSFUL_INITIALIZATION;
8     }
9 }

```



```

7      reportActualBinding(staticLoggerBinderPathSet);
8          fixSubstituteLoggers();
9          replayEvents();
10         // release all resources in SUBST_FACTORY
11         SUBST_FACTORY.clear();
12     } catch (NoClassDefFoundError ncde) {
13         String msg = ncde.getMessage();
14         if
15 (messageContainsOrgSlf4jImplStaticLoggerBinder(msg)) {
16             INITIALIZATION_STATE =
17 NOP_FALLBACK_INITIALIZATION;
18             Util.report("Failed to load class
19 \"org.slf4j.impl.StaticLoggerBinder\");
20             Util.report("Defaulting to no-
21 operation (NOP) logger implementation");
22             Util.report("See " +
23 NO_STATICLOGGERBINDER_URL + " for further details.");
24         } else {
25             failedBinding(ncde);
26             throw ncde;
27         }
28     } catch (java.lang.NoSuchMethodError nsme) {
29         String msg = nsme.getMessage();
30         if (msg != null &&
31 msg.contains("org.slf4j.impl.StaticLoggerBinder.getSin-
32 gleton())) {
33             INITIALIZATION_STATE =
34 FAILED_INITIALIZATION;
35             Util.report("slf4j-api 1.6.x (or
36 later) is incompatible with this binding.");
37             Util.report("Your binding is version
38 1.5.5 or earlier.");
39             Util.report("Upgrade your binding to
40 version 1.6.x.");
41         }
42         throw nsme;
43     } catch (Exception e) {
44         failedBinding(e);

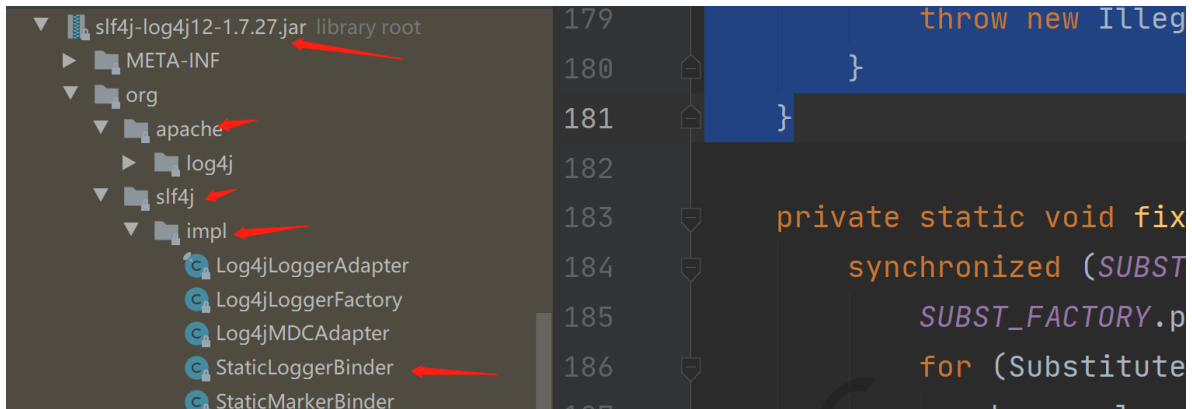
```

```

34         throw new
        IllegalStateException("Unexpected initialization
        failure", e);
35     }
36 }

```

每一个日志实现的中间包都有一个StaticLoggerBinder:



```

1  public class StaticLoggerBinder implements
    LoggerFactoryBinder {
2
3      /**
4       * The unique instance of this class.
5       *
6       */
7      private static final StaticLoggerBinder SINGLETON
8      = new StaticLoggerBinder();
9
10     /**
11      * Return the singleton of this class.
12      *
13      * @return the StaticLoggerBinder singleton
14      */
15     public static final StaticLoggerBinder
16     getSingleton() {
17         return SINGLETON;
18     }
19
20     /**
21      * Declare the version of the SLF4J API this
22      * implementation is compiled against.
23      *
24      * The value of this field is modified with each
25      * major release.

```

```

21     */
22     // to avoid constant folding by the compiler, this
    field must *not* be final
23     public static String REQUESTED_API_VERSION =
    "1.6.99"; // !final
24
25     private static final String loggerFactoryClassStr
    = Log4jLoggerFactory.class.getName();
26
27     /**
28      * The ILoggerFactory instance returned by the
    {@link #getLoggerFactory}
29      * method should always be the same object
30      */
31     private final ILoggerFactory loggerFactory;
32
33     private StaticLoggerBinder() {
34         loggerFactory = new Log4jLoggerFactory();
35         try {
36             @SuppressWarnings("unused")
37             Level level = Level.TRACE;
38         } catch (NoSuchFieldError nsfe) {
39             Util.report("This version of SLF4J
    requires log4j version 1.2.12 or later. See also
    http://www.slf4j.org/codes.html#log4j_version");
40         }
41     }
42
43     public ILoggerFactory getLoggerFactory() {
44         return loggerFactory;
45     }
46
47     public String getLoggerFactoryClassStr() {
48         return loggerFactoryClassStr;
49     }
50 }

```

## 二、JCL 日志门面

全称为Jakarta Commons Logging，是Apache提供的一个通用日志API。改日志门面的使用并不是很广泛。

它是为 "所有的Java日志实现"提供一个统一的接口，它自身也提供一个日志的实现，但是功能非常弱（SimpleLog）。所以一般不会单独使用它。他允许开发人员使用不同的具体日志实现工具: Log4j, Jdk 自带的日志 (JUL)

JCL 有两个基本的抽象类：Log(基本记录器)和LogFactory(负责创建Log实例)。

### 1、JCL入门

1. 建立maven工程
2. 添加依赖

```
1 <dependency>
2   <groupId>commons-logging</groupId>
3   <artifactId>commons-logging</artifactId>
4   <version>1.2</version>
5 </dependency>
```

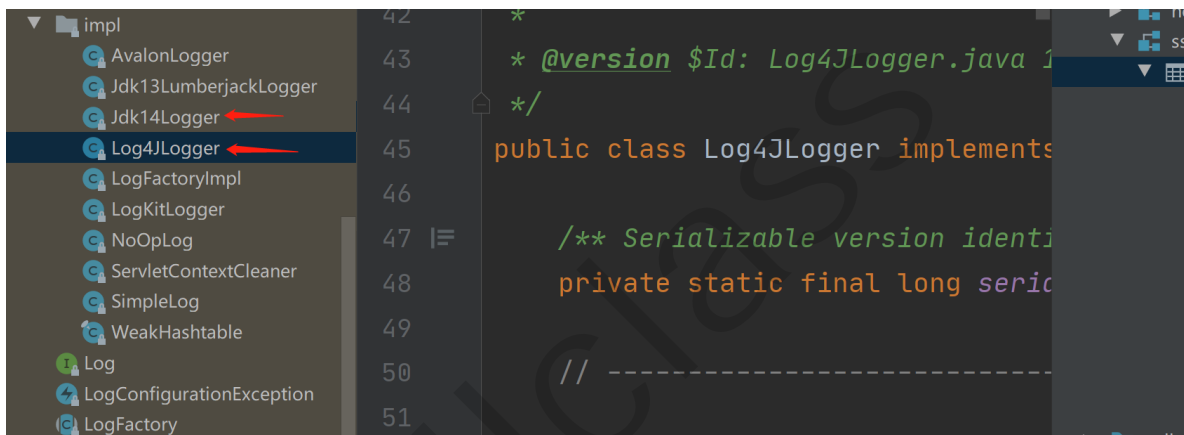
3. 入门代码

```
1 public class JULTest {
2     @Test
3     public void testQuick() throws Exception {
4         // 创建日志对象
5         Log log = LogFactory.getLog(JULTest.class);
6         // 日志记录输出
7         log.fatal("fatal");
8         log.error("error");
9         log.warn("warn");
10        log.info("info");
11        log.debug("debug");
12    }
13 }
```

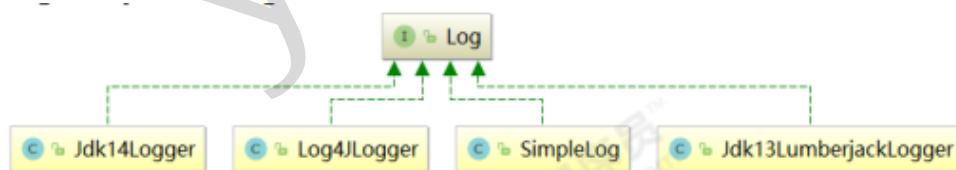
## 我们为什么要使用日志门面：

1. 面向接口开发，不再依赖具体的实现类。减少代码的耦合
2. 项目通过导入不同的日志实现类，可以灵活的切换日志框架
3. 统一API，方便开发者学习和使用
4. 统一配置便于项目日志的管理

## 2、JCL原理



### 1. 通过LogFactory动态加载Log实现类



### 2. 日志门面支持的日志实现数组

```

1 private static final String[] classesToDiscover =
2     new String[]
3     {"org.apache.commons.logging.impl.Log4JLogger",
4
5     "org.apache.commons.logging.impl.Jdk14Logger",
6
7     "org.apache.commons.logging.impl.Jdk13LumberjackLogger",
8
9     ,
10
11     "org.apache.commons.logging.impl.SimpleLog"};

```

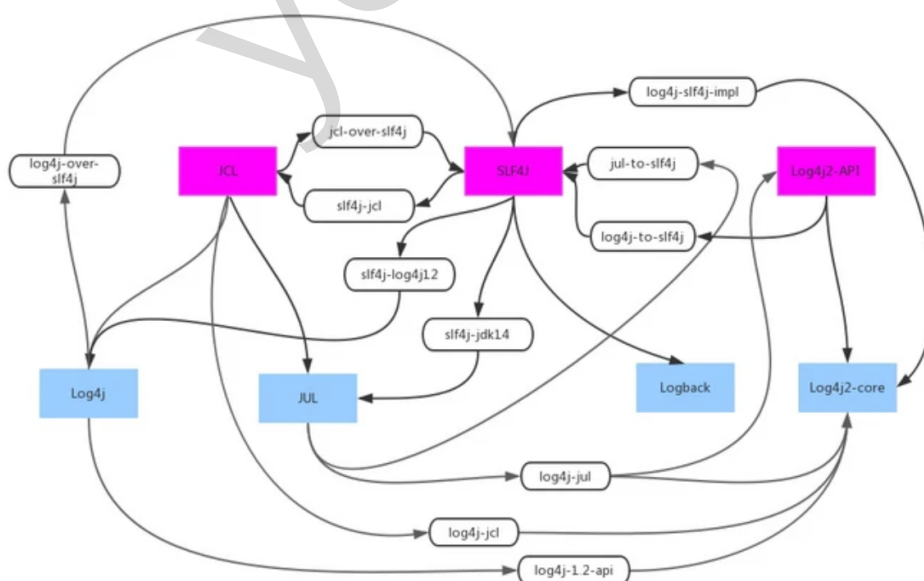
### 3. 获取具体的日志实现

```

1 for(int i = 0; i < classesToDiscover.length && result
2     == null; ++i) {
3     result =
4     this.createLogFromClass(classesToDiscover[i],
5                             logCategory,
6                             true);
7 }

```

### 3、日志生态图：



历史插曲：现在聊聊历史：[https://segmentfault.com/a/1190000021121882?utm\\_source=tag-newest](https://segmentfault.com/a/1190000021121882?utm_source=tag-newest)

## 第五章 Logback的使用

Logback是由log4j创始人设计的另一个开源日志组件，性能比log4j要好。

官方网站：<https://logback.qos.ch/index.html>

Logback主要分为三个模块：

- logback-core：其它两个模块的基础模块
- logback-classic：它是log4j的一个改良版本，同时它完整实现了slf4j API
- logback-access：访问模块与Servlet容器集成提供通过Http来访问日志的功能 后续的日志代码都是通过SLF4J日志门面搭建日志系统，所以在代码是没有区别，主要是通过修改配置文件和pom.xml依赖

### 一、logback入门

#### 1. 添加依赖

```
1 <dependency>
2     <groupId>ch.qos.logback</groupId>
3     <artifactId>logback-classic</artifactId>
4     <version>1.2.3</version>
5 </dependency>
```

#### 2. java代码

```
1 public class TestLogback {
2
3     private final static Logger logger =
4         LoggerFactory.getLogger(TestLog4j.class);
5
6     @Test
```

```

6      public void testLogback(){
7          //打印日志信息
8          logger.error("error");
9          logger.warn("warn");
10         logger.info("info");
11         logger.debug("debug");
12         logger.trace("trace");
13     }
14 }

```

其实我们发现即使项目中没有引入slf4j我们这里也是用的slf4j门面进行编程。

1、

```

package ch.qos.logback.classic;

import ...

public final class Logger implements org.slf4j.Logger, LocationAwareLogger, Append

```

2、从logback'的pom依赖中我们看到slf4j，依赖会进行传递



## 二、源码解析

### 1、spi机制

SPI全称Service Provider Interface，是Java提供的一套用来被第三方实现或者扩展的API，它可以用来启用框架扩展和替换组件。它是一种服务发现机制。它通过在ClassPath路径下的META-INF/services文件夹查找文件，自动加载文件里所定义的类。

主要是使用，java.util包下的ServiceLoader实现：



```
1 public static <S> ServiceLoader<S> load(Class<S>  
   service, ClassLoader loader)  
2 {  
3     return new ServiceLoader<>(service, loader);  
4 }
```

## 2、源码解析

源码看一下启动过程：

1、我们从日志工厂的常见看起，这里是slf4j的实现：

```
1 private final static Logger logger =  
   LoggerFactory.getLogger(TestLog4j.class);
```

核心方法只有一句：

```
1 public static Logger getLogger(Class<?> clazz) {  
2     Logger logger = getLogger(clazz.getName());  
3     ...中间的逻辑判断省略掉  
4     return logger;  
5 }
```

看一下getLogger方法，这里是先获取日志工厂，在从工厂中提取日志对象，我们不考虑日志对象，主要看看日志工厂的环境怎么初始化的：

```
1 public static Logger getLogger(String name) {  
2     ILoggerFactory iLoggerFactory =  
   getLoggerFactory();  
3     return iLoggerFactory.getLogger(name);  
4 }
```

日志工厂的创建方法：

```
1 public static ILoggerFactory getILoggerFactory() {  
2     ...去掉其他的代码，从这一行看。  
3     return  
4     StaticLoggerBinder.getSingleton().getLoggerFactory();  
5 }
```

这里就进入了，StaticLoggerBinder这个对象，这是日志实现用来和slf4j进行绑定的类，从此就进入日志实现中了。

StaticLoggerBinder.getSingleton()这里看到出来是一个单例，来到这个类当中，我们看到，直接返回了defaultLoggerContext

```
1 public ILoggerFactory getLoggerFactory() {  
2     if (!initialized) {  
3         return defaultLoggerContext;  
4     }  
5     ... 省略其他  
6  
7 }
```

这是个日志上下文，一定保存了我们的环境，配置内容一定在这个里边，那么哪里初始化他了呢，我们能想到的就是静态代码块了：

我们发现这个类中还真有：

```
1 static {  
2     SINGLETON.init();  
3 }
```

我们看到init()方法中，有一个autoConfig()，感觉就像在自动配置：

```

1 void init() {
2     try {
3         try {
4             new
ContextInitializer(defaultLoggerContext).autoConfig();
5         } catch (JoranException je) {
6             Util.report("Failed to auto configure
default logger context", je);
7         }
8         ...其他省略
9     }
10 }

```

默认配置：ContextInitializer类是初始化的关键：

自动配置是这么玩的，先找配置文件

```

1 public void autoConfig() throws JoranException {
2
3     StatusListenerConfigHelper.installIfAsked(loggerConte
xt);
4
5     // 这就是去找配置文件
6     URL url =
findURLofDefaultConfigurationFile(true);
7     if (url != null) {
8         // 解析配置
9         configureByResource(url);
10    } else {
11        // 没有找到文件，就去使用spi机制找一个配置类，这个
配置类是在web中用的
12        Configurator c =
EnvUtil.loadFromServiceLoader(Configurator.class);
13        if (c != null) {
14            try {
15                c.setContext(loggerContext);
16                c.configure(loggerContext);
17            } catch (Exception e) {
18                throw new
LogbackException(String.format("Failed to initialize
Configurator: %s using ServiceLoader", c != null ?
c.getClass()

```

```

17     .getCanonicalName() : "null"), e);
18         }
19     } else {
20         // 如果没有找到，就做基本的配置
21         BasicConfigurator basicConfigurator =
22     new BasicConfigurator();
23
24         basicConfigurator.setContext(loggerContext);
25
26         basicConfigurator.configure(loggerContext);
27     }
28 }
29 }

```

寻找配置文件的过程：

```

1  final public static String GROOVY_AUTOCONFIG_FILE =
2  "logback.groovy";
3  final public static String AUTOCONFIG_FILE =
4  "logback.xml";
5  final public static String TEST_AUTOCONFIG_FILE =
6  "logback-test.xml";
7
8  public URL findURLofDefaultConfigurationFile(boolean
9  updateStatus) {
10     ClassLoader myClassLoader =
11     Loader.getClassLoaderOfObject(this);
12     URL url =
13     findConfigFileURLFromSystemProperties(myClassLoader,
14     updateStatus);
15     if (url != null) {
16         return url;
17     }
18
19     url = getResource(TEST_AUTOCONFIG_FILE,
20     myClassLoader, updateStatus);
21     if (url != null) {
22         return url;
23     }
24 }

```

```

16
17     url = getResource(GROOVY_AUTOCONFIG_FILE,
myClassLoader, updateStatus);
18     if (url != null) {
19         return url;
20     }
21
22     return getResource(AUTOCONFIG_FILE, myClassLoader,
updateStatus);
23 }

```

```

1  public void configureByResource(URL url) throws
JoranException {
2      if (url == null) {
3          throw new IllegalArgumentException("URL
argument cannot be null");
4      }
5      final String urlString = url.toString();
6      if (urlString.endsWith("groovy")) {
7          if (EnvUtil.isGroovyAvailable()) {
8              // avoid directly referring to
GafferConfigurator so as to avoid
9              // loading groovy.lang.GroovyObject .
See also http://jira.qos.ch/browse/LBCLASSIC-214
10
GafferUtil.runGafferConfiguratorOn(loggerContext,
this, url);
11      } else {
12          StatusManager sm =
loggerContext.getStatusManager();
13          sm.add(new ErrorStatus("Groovy classes
are not available on the class path. ABORTING
INITIALIZATION.", loggerContext));
14      }
15      } else if (urlString.endsWith("xml")) {
16          JoranConfigurator configurator = new
JoranConfigurator();
17          configurator.setContext(loggerContext);

```

```

18         configurator.doConfigure(url);
19     } else {
20         throw new LogbackException("Unexpected
filename extension of file [" + url.toString() + "].
should be either .groovy or .xml");
21     }
22 }

```

基础配置的代码：

```

1 public class BasicConfigurator extends
ContextAwareBase implements Configurator {
2
3     public BasicConfigurator() {
4     }
5
6     public void configure(LoggerContext lc) {
7         addInfo("Setting up default configuration.");
8
9         ConsoleAppender<ILoggingEvent> ca = new
ConsoleAppender<ILoggingEvent>();
10        ca.setContext(lc);
11        ca.setName("console");
12        LayoutWrappingEncoder<ILoggingEvent> encoder =
new LayoutWrappingEncoder<ILoggingEvent>();
13        encoder.setContext(lc);
14
15        // same as
16        // PatternLayout layout = new PatternLayout();
17        // layout.setPattern("%d{HH:mm:ss.SSS}
[%thread] %-5level %logger{36} - %msg%n");
18        TTLLLayout layout = new TTLLLayout();
19
20        layout.setContext(lc);
21        layout.start();
22        encoder.setLayout(layout);
23
24        ca.setEncoder(encoder);
25        ca.start();
26

```

```

27
28         Logger rootLogger =
           lc.getLogger(Logger.ROOT_LOGGER_NAME);
29         rootLogger.addAppender(ca);
30     }
31 }

```

我们先不说配置的事情，从源码中我们可以看出有几种配置，因为有了我们先模仿BasicConfigurator写一个类，只做略微的改动：

```

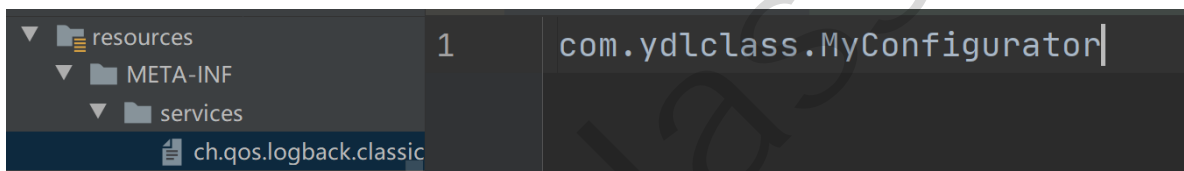
1  public class MyConfigurator extends ContextAwareBase
   implements Configurator {
2      public MyConfigurator() {
3      }
4
5      public void configure(LoggerContext lc) {
6          addInfo("Setting up default configuration.");
7
8          ConsoleAppender<ILoggingEvent> ca = new
           ConsoleAppender<ILoggingEvent>();
9          ca.setContext(lc);
10         ca.setName("console");
11         LayoutWrappingEncoder<ILoggingEvent> encoder =
           new LayoutWrappingEncoder<ILoggingEvent>();
12         encoder.setContext(lc);
13
14
15         // same as
16         // PatternLayout layout = new PatternLayout();
17         // layout.setPattern("%d{HH:mm:ss.SSS}
           [%thread] %-5level %logger{36} - %msg%n");
18         PatternLayout layout = new PatternLayout();
19         layout.setPattern("%d{HH:mm:ss} [%thread]
           %-5level %logger{36} - %msg%n");
20
21         layout.setContext(lc);
22         layout.start();
23         encoder.setLayout(layout);
24

```

```
25         ca.setEncoder(encoder);
26         ca.start();
27
28         Logger rootLogger =
29             lc.getLogger(Logger.ROOT_LOGGER_NAME);
30         rootLogger.addAppender(ca);
31     }
32 }
```

在resource中新建META-INF目录，下边在新建services文件夹，再新建一个名字我 `ch.qos.logback.classic.spi.Configurator` 的文件，

内容是： `com.ydlclass.MyConfigurator`



### 三、三大组件

- 1、appender，输出源，一个日志可以有多个输出源
- 2、encoder，一个appender有一个encoder，负责将一个event事件转换成一组byte数组，并将转换后的字节数据输出到文件中。

Encoder负责把事件转换为字节数组，并把字节数组写到合适的输出流。因此，encoder可以控制在什么时候、把什么样的字节数组写入到其所有者维护的输出流中。Encoder接口有两个实现类，LayoutWrappingEncoder与PatternLayoutEncoder。

注意：在logback 0.9.19 版之前没有 encoder。

在之前的版本里，多数 appender 依靠 layout 来把事件转换成字符串并用 `java.io.Writer` 把字符串输出。在之前的版本里，用户需要在 `FileAppender`里嵌入一个 `PatternLayout`。



3、layout，格式化数据将event事件转化为字符串，解析的过程

4、filter 过滤器

```
1 LevelFilter levelFilter = new LevelFilter();
2     levelFilter.setOnMatch(FilterReply.DENY);
3     levelFilter.setLevel(Level.WARN);
4     levelFilter.start();
5     ca.addFilter(levelFilter);
```

1. %-5level
2. %d{yyyy-MM-dd HH:mm:ss.SSS}日期
3. %c类的完整名称
4. %M为method
5. %L为行号
6. %thread线程名称
7. %m或者%msg为信息
8. %n换行

能看到logback的格式化信息

```
1 public class PatternLayout extends
   PatternLayoutBase<ILoggingEvent> {
2
3     public static final Map<String, String>
   defaultConverterMap = new HashMap<String, String>();
4     public static final String HEADER_PREFIX =
   "#logback.classic pattern: ";
5
6     static {
7
8         defaultConverterMap.putAll(Parser.DEFAULT_COMPOSITE_
   CONVERTER_MAP);
9
10        defaultConverterMap.put("d",
   DateConverter.class.getName());
```

```
10         defaultConverterMap.put("date",
DateConverter.class.getName());
11
12         defaultConverterMap.put("r",
RelativeTimeConverter.class.getName());
13         defaultConverterMap.put("relative",
RelativeTimeConverter.class.getName());
14
15         defaultConverterMap.put("level",
LevelConverter.class.getName());
16         defaultConverterMap.put("le",
LevelConverter.class.getName());
17         defaultConverterMap.put("p",
LevelConverter.class.getName());
18
19         defaultConverterMap.put("t",
ThreadConverter.class.getName());
20         defaultConverterMap.put("thread",
ThreadConverter.class.getName());
21
22         defaultConverterMap.put("lo",
LoggerConverter.class.getName());
23         defaultConverterMap.put("logger",
LoggerConverter.class.getName());
24         defaultConverterMap.put("c",
LoggerConverter.class.getName());
25
26         defaultConverterMap.put("m",
MessageConverter.class.getName());
27         defaultConverterMap.put("msg",
MessageConverter.class.getName());
28         defaultConverterMap.put("message",
MessageConverter.class.getName());
29
30         defaultConverterMap.put("C",
ClassOfCallerConverter.class.getName());
31         defaultConverterMap.put("class",
ClassOfCallerConverter.class.getName());
32
33         defaultConverterMap.put("M",
MethodOfCallerConverter.class.getName());
```

```
34         defaultConverterMap.put("method",
MethodOfCallerConverter.class.getName());
35
36         defaultConverterMap.put("L",
LineOfCallerConverter.class.getName());
37         defaultConverterMap.put("line",
LineOfCallerConverter.class.getName());
38
39         defaultConverterMap.put("F",
FileOfCallerConverter.class.getName());
40         defaultConverterMap.put("file",
FileOfCallerConverter.class.getName());
41
42         defaultConverterMap.put("x",
MDCCConverter.class.getName());
43         defaultConverterMap.put("mdc",
MDCCConverter.class.getName());
44
45         defaultConverterMap.put("ex",
ThrowableProxyConverter.class.getName());
46         defaultConverterMap.put("exception",
ThrowableProxyConverter.class.getName());
47         defaultConverterMap.put("rEx",
RootCauseFirstThrowableProxyConverter.class.getName()
);
48         defaultConverterMap.put("rootException",
RootCauseFirstThrowableProxyConverter.class.getName()
);
49         defaultConverterMap.put("throwable",
ThrowableProxyConverter.class.getName());
50
51         defaultConverterMap.put("xEx",
ExtendedThrowableProxyConverter.class.getName());
52         defaultConverterMap.put("xException",
ExtendedThrowableProxyConverter.class.getName());
53         defaultConverterMap.put("xThrowable",
ExtendedThrowableProxyConverter.class.getName());
54
55         defaultConverterMap.put("nopex",
NopThrowableInformationConverter.class.getName());
```

```
56         defaultConverterMap.put("nopexception",
NopThrowableInformationConverter.class.getName());
57
58         defaultConverterMap.put("cn",
ContextNameConverter.class.getName());
59         defaultConverterMap.put("contextName",
ContextNameConverter.class.getName());
60
61         defaultConverterMap.put("caller",
CallerDataConverter.class.getName());
62
63         defaultConverterMap.put("marker",
MarkerConverter.class.getName());
64
65         defaultConverterMap.put("property",
PropertyConverter.class.getName());
66
67         defaultConverterMap.put("n",
LineSeparatorConverter.class.getName());
68
69         defaultConverterMap.put("black",
BlackCompositeConverter.class.getName());
70         defaultConverterMap.put("red",
RedCompositeConverter.class.getName());
71         defaultConverterMap.put("green",
GreenCompositeConverter.class.getName());
72         defaultConverterMap.put("yellow",
YellowCompositeConverter.class.getName());
73         defaultConverterMap.put("blue",
BlueCompositeConverter.class.getName());
74         defaultConverterMap.put("magenta",
MagentaCompositeConverter.class.getName());
75         defaultConverterMap.put("cyan",
CyanCompositeConverter.class.getName());
76         defaultConverterMap.put("white",
WhiteCompositeConverter.class.getName());
77         defaultConverterMap.put("gray",
GrayCompositeConverter.class.getName());
78         defaultConverterMap.put("boldRed",
BoldRedCompositeConverter.class.getName());
```

```

79         defaultConverterMap.put("boldGreen",
    BoldGreenCompositeConverter.class.getName());
80         defaultConverterMap.put("boldYellow",
    BoldYellowCompositeConverter.class.getName());
81         defaultConverterMap.put("boldBlue",
    BoldBlueCompositeConverter.class.getName());
82         defaultConverterMap.put("boldMagenta",
    BoldMagentaCompositeConverter.class.getName());
83         defaultConverterMap.put("boldCyan",
    BoldCyanCompositeConverter.class.getName());
84         defaultConverterMap.put("boldwhite",
    BoldwhiteCompositeConverter.class.getName());
85         defaultConverterMap.put("highlight",
    HighlightingCompositeConverter.class.getName());
86
87         defaultConverterMap.put("lsn",
    LocalSequenceNumberConverter.class.getName());
88
89     }
90
91     public PatternLayout() {
92         this.postCompileProcessor = new
    EnsureExceptionHandling();
93     }
94
95     public Map<String, String>
    getDefaultConverterMap() {
96         return defaultConverterMap;
97     }
98
99     public String doLayout(ILoggingEvent event) {
100         if (!isStarted()) {
101             return CoreConstants.EMPTY_STRING;
102         }
103         return writeLoopOnConverters(event);
104     }
105
106     @Override
107     protected String getPresentationHeaderPrefix() {
108         return HEADER_PREFIX;
109     }

```

## 1 | OutputStreamAppender

```
1 protected void subAppend(E event) {
2     if (!isStarted()) {
3         return;
4     }
5     try {
6         // this step avoids LBCLASSIC-139
7         if (event instanceof
DeferredProcessingAware) {
8             ((DeferredProcessingAware)
event).prepareForDeferredProcessing();
9         }
10        // the synchronization prevents the
OutputStream from being closed while we
11        // are writing. It also prevents multiple
threads from entering the same
12        // converter. Converters assume that they
are in a synchronized block.
13        // lock.lock();
14
15        byte[] byteArray =
this.encoder.encode(event);
16        writeBytes(byteArray);
17
18    } catch (IOException ioe) {
19        // as soon as an exception occurs, move to
non-started state
20        // and add a single ErrorStatus to the SM.
21        this.started = false;
22        addStatus(new ErrorStatus("IO failure in
appender", this, ioe));
23    }
24 }
```

```
1 public byte[] encode(E event) {
2     String txt = layout.doLayout(event);
3     return convertToBytes(txt);
4 }
```

```
1 private void buildLoggingEventAndAppend(final String
    localFQCN, final Marker marker, final Level level,
    final String msg, final Object[] params,
2                                     final Throwable
    t) {
3     LoggingEvent le = new LoggingEvent(localFQCN, this,
    level, msg, t, params);
4     le.setMarker(marker);
5     callAppenders(le);
6 }
```

## 四、logback配置

Let us begin by discussing the initialization steps that logback follows to try to configure itself:

1. Logback tries to find a file called *logback-test.xml* [in the classpath](#).
2. If no such file is found, logback tries to find a file called *logback.groovy* [in the classpath](#).
3. If no such file is found, it checks for the file *logback.xml* [in the classpath](#)..
4. If no such file is found, [service-provider loading facility](#) (introduced in JDK 1.6) is used to resolve the implementation of `com.qos.logback.classic.spi.Configurator` interface by looking up the file *META-INF\services\ch.qos.logback.classic.spi.Configurator* in the class path.

Its contents should specify the fully qualified class name of the desired `Configurator` implementation.

5. If none of the above succeeds, logback configures itself automatically using the `BasicConfigurator` which will cause logging output to be directed to the console.

## 2. 基本配置信息

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <configuration>
3
4     <!-- 格式化输出: %d表示日期, %thread表示线程名, %-5level:
      级别从左显示5个字符宽度
5     %msg: 日志消息, %n是换行符-->
6     <property name="pattern" value="%d{yyyy-MM-dd
      HH:mm:ss.SSS} %c [%thread]
7                                     %-5level %msg%n"/>
8     <!--
9         Appender: 设置日志信息的去向, 常用的有以下几个
10        ch.qos.logback.core.ConsoleAppender (控制台)
11
12        ch.qos.logback.core.rolling.RollingFileAppender (文件
13        大小到达指定尺寸的时候产生一个新文件)
14        ch.qos.logback.core.FileAppender (文件)
15        -->
16     <appender name="console"
17     class="ch.qos.logback.core.ConsoleAppender">
18         <!-- 输出流对象 默认 System.out 改为 System.err -->
19         <target>System.err</target>
20         <!-- 日志格式配置 -->
21         <encoder
22         class="ch.qos.logback.classic.encoder.PatternLayoutEnc
23         oder">
24             <pattern>${pattern}</pattern>
25         </encoder>
26     </appender>
27     <!--
28         用来设置某一个包或者具体的某一个类的日志打印级别、以及指定
29         <appender>。
```



```

24         <logger>仅有一个name属性，一个可选的
    level和一个可选的additivity属性
25         name:
26         用来指定受此logger约束的某一个包或者具体的某一个类。
27         level:
28         用来设置打印级别，大小写无关：TRACE，DEBUG，INFO，
    WARN，ERROR，ALL 和
29         OFF，
30         如果未设置此属性，那么当前logger将会继承上级的级
    别。
31         additivity:
32         是否向上级logger传递打印信息。默认是true。
33         <logger>可以包含零个或多个<appender-ref>元素，
    标识这个appender将会添加到这个
34         logger
35         -->
36     <!--
37         也是<logger>元素，但是它是根logger。默认debug
38         level:用来设置打印级别，大小写无关：TRACE，
    DEBUG，INFO，WARN，ERROR，ALL
39         和 OFF，
40         <root>可以包含零个或多个<appender-ref>元
    素，标识这个appender将会添加到这个
41         logger。
42         -->
43     <root level="ALL">
44         <appender-ref ref="console"/>
45     </root>
46 </configuration>

```

### 3. FileAppender配置

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <configuration>
3     <!-- 自定义属性 可以通过${name}进行引用-->
4     <property name="pattern" value="[%-5level]
    %d{yyyy-MM-dd HH:mm:ss} %c %M
5                                     %L [%thread] %m
    %n"/>
6     <!--
7     日志输出格式:

```

```
8      %d{pattern}日期
9      %m或者%msg为信息
10     %M为method
11     %L为行号
12     %c类的完整名称
13     %thread线程名称
14     %n换行
15     %-5level
16     -->
17     <!-- 日志文件存放目录 -->
18     <property name="log_dir" value="d:/logs">
19 </property>
19     <!--控制台输出appender对象-->
20     <appender name="console"
21 class="ch.qos.logback.core.ConsoleAppender">
21         <!--输出流对象 默认 System.out 改为 System.err-->
22         <target>System.err</target>
23         <!--日志格式配置-->
24         <encoder
25
26 class="ch.qos.logback.classic.encoder.PatternLayoutEnc
27 oder">
26             <pattern>${pattern}</pattern>
27             </encoder>
28         </appender>
29     <!--日志文件输出appender对象-->
30     <appender name="file"
31 class="ch.qos.logback.core.FileAppender">
31         <!--日志格式配置-->
32         <encoder
33
34 class="ch.qos.logback.classic.encoder.PatternLayoutEnc
35 oder">
33             <pattern>${pattern}</pattern>
34             </encoder>
35             <!--日志输出路径-->
36             <file>${log_dir}/logback.log</file>
37         </appender>
38     <!-- 生成html格式appender对象 -->
39     <appender name="htmlFile"
40 class="ch.qos.logback.core.FileAppender">
40         <!--日志格式配置-->
```

```

41         <encoder
class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
42             <layout
class="ch.qos.logback.classic.html.HTMLLayout">
43                 <pattern>%level%d{yyyy-MM-dd
HH:mm:ss}%c%M%L%thread%m</pattern>
44             </layout>
45         </encoder>
46         <!-- 日志输出路径-->
47         <file>${log_dir}/logback.html</file>
48     </appender>
49     <!--RootLogger对象-->
50     <root level="all">
51         <appender-ref ref="console"/>
52         <appender-ref ref="file"/>
53         <appender-ref ref="htmlFile"/>
54     </root>
55 </configuration>

```

#### 4. RollingFileAppender配置

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <configuration>
3     <!-- 自定义属性 可以通过${name}进行引用-->
4     <property name="pattern" value="[%-5level]
%d{yyyy-MM-dd HH:mm:ss} %c %M
5                                     %L [%thread] %m
%n"/>
6     <!--
7     日志输出格式:
8     %d{pattern}日期
9     %m或者%msg为信息
10    %M为method
11    %L为行号
12    %c类的完整名称
13    %thread线程名称
14    %n换行
15    %-5level
16    -->
17    <!-- 日志文件存放目录 -->

```

```
18     <property name="log_dir" value="d:/logs">
19 </property>
19     <!--控制台输出appender对象-->
20     <appender name="console"
21 class="ch.qos.logback.core.ConsoleAppender">
21         <!--输出流对象 默认 System.out 改为 System.err-->
22         <target>System.err</target>
23         <!--日志格式配置-->
24         <encoder
25
26 class="ch.qos.logback.classic.encoder.PatternLayoutEnc
27 oder">
26             <pattern>${pattern}</pattern>
27             </encoder>
28         </appender>
29         <!-- 日志文件拆分和归档的appender对象-->
30         <appender name="rollFile"
31
32 class="ch.qos.logback.core.rolling.RollingFileAppende
33 r">
32             <!--日志格式配置-->
33             <encoder
34
35 class="ch.qos.logback.classic.encoder.PatternLayoutEnc
36 oder">
35                 <pattern>${pattern}</pattern>
36                 </encoder>
37                 <!--日志输出路径-->
38                 <file>${log_dir}/roll_logback.log</file>
39                 <!--指定日志文件拆分和压缩规则-->
40                 <rollingPolicy
41
42 class="ch.qos.logback.core.rolling.SizeAndTimeBasedRol
43 lingPolicy">
41                     <!--通过指定压缩文件名称，来确定分割文件方式-->
42
43                     <fileNamePattern>${log_dir}/rolling.%d{yyyy-
44 MM•dd}.log%i.gz</fileNamePattern>
43                     <!--文件拆分大小-->
44                     <maxFileSize>1MB</maxFileSize>
45                     </rollingPolicy>
46                 </appender>
```

```

47      <!--RootLogger对象-->
48      <root level="all">
49          <appender-ref ref="console"/>
50          <appender-ref ref="rollFile"/>
51      </root>
52 </configuration>
53

```

## 5. Filter和异步日志配置

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <configuration>
3      <!-- 自定义属性 可以通过${name}进行引用-->
4      <property name="pattern" value="[%-5level]
5          %d{yyyy-MM-dd HH:mm:ss} %c %M
6              %L [%thread] %m
7          %n"/>
8      <!--
9          日志输出格式:
10         %d{pattern}日期
11         %m或者%msg为信息
12         %M为method
13         %L为行号
14         %c类的完整名称
15         %thread线程名称
16         %n换行
17         %-5level
18         -->
19         <!-- 日志文件存放目录 -->
20         <property name="log_dir" value="d:/logs/">
21     </property>
22     <!--控制台输出appender对象-->
23     <appender name="console"
24         class="ch.qos.logback.core.ConsoleAppender">
25         <!--输出流对象 默认 System.out 改为 System.err-->
26         <target>System.err</target>
27         <!--日志格式配置-->
28         <encoder
29             class="ch.qos.logback.classic.encoder.PatternLayoutEnc
30             oder">

```

```
26         <pattern>${pattern}</pattern>
27     </encoder>
28 </appender>
29 <!-- 日志文件拆分和归档的appender对象-->
30 <appender name="rollFile"
31
32     class="ch.qos.logback.core.rolling.RollingFileAppende
33     r">
34         <!--日志格式配置-->
35         <encoder
36
37             <pattern>${pattern}</pattern>
38             </encoder>
39             <!--日志输出路径-->
40             <file>${log_dir}roll_logback.log</file>
41             <!--指定日志文件拆分和压缩规则-->
42             <rollingPolicy
43
44                 class="ch.qos.logback.core.rolling.SizeAndTimeBasedRol
45                 lingPolicy">
46                     <!--通过指定压缩文件名称，来确定分割文件方式-->
47
48                     <fileNamePattern>${log_dir}rolling.%d{yyyy-
49                     MM•dd}.log%i.gz</fileNamePattern>
50                     <!--文件拆分大小-->
51                     <maxFileSize>1MB</maxFileSize>
52                     </rollingPolicy>
53                     <!--filter配置-->
54                     <filter
55
56                         class="ch.qos.logback.classic.filter.LevelFilter">
57                             <!--设置拦截日志级别-->
58                             <level>error</level>
59                             <onMatch>ACCEPT</onMatch>
60                             <onMismatch>DENY</onMismatch>
61                         </filter>
62                     </appender>
63                     <!--异步日志-->
64                     <appender name="async"
65
66                         class="ch.qos.logback.classic.AsyncAppender">
```

```

57         <appender-ref ref="rollFile"/>
58     </appender>
59     <!--RootLogger对象-->
60     <root level="all">
61         <appender-ref ref="console"/>
62         <appender-ref ref="async"/>
63     </root>
64     <!--自定义logger additivity表示是否从 rootLogger继承配
        置-->
65     <logger name="com.ydlclass" level="debug"
        additivity="false">
66         <appender-ref ref="console"/>
67     </logger>
68 </configuration>

```

## 五、logback-access的使用

在server.xml里的标签下加上

```

1 <valve
  className="org.apache.catalina.valves.AccessLogValve"
  directory="logs" prefix="localhost_access_log."
  suffix=".txt" pattern="common" resolveHosts="false"/>

```

就可以了，下面咱们逐一分析各个参数。

className	想配置访问日志？这就必须得写成这样。
directory	这个东西是日志文件放置的目录，在tomcat下面有个logs文件夹，那里面是专门放置日志文件的，当然你也可以修改，我就给改成了D:\
prefix	这个是日志文件的名称前缀，我的日志名称为localhost_access_log.2007-09-22.txt，前面的前缀就是这个localhost_access_log
suffix	这就是后缀名啦，可以改成别的
pattern	这个是最主要的参数了，具体的咱们下面讲，这个参数的内容比较丰富。
resolveHosts	如果这个值是true的话，tomcat会将这个服务器IP地址通过DNS转换为主机名，如果是false，就直接写服务器IP地址啦

To use logback-access with Tomcat, after downloading the logback distribution, place the files *logback-core-1.3.0-alpha10.jar* and *logback-access-1.3.0-alpha10.jar* under \$TOMCAT\_HOME/lib/ directory, where \$TOMCAT\_HOME is the folder where you have installed Tomcat.

logback-access模块与Servlet容器（如Tomcat和Jetty）集成，以提供HTTP访问日志功能。我们可以使用logback-access模块来替换tomcat的访问日志。

1. 将logback-access.jar与logback-core.jar复制到\$TOMCAT\_HOME/lib/目录下
2. 修改\$TOMCAT\_HOME/conf/server.xml中的Host元素中添加：

```
1 <valve
  className="ch.qos.logback.access.tomcat.LogbackValve"/>
```

3. logback默认会在\$TOMCAT\_HOME/conf下查找文件 logback-access.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <configuration>
```





11	2021.10.26 10:51	文件夹	
access.ydl.log	2021.10.26 10:54	文本文档	3 KB

## 第六章 log4j2的使用

Apache Log4j2是对Log4j的升级版，参考了logback的一些优秀的设计，并且修复了一些问题，因此带来了一些重大的提升，主要有：

- 异常处理，在logback中，Appender中的异常不会被应用感知到，但是在log4j2中，提供了一些异常处理机制。
- 性能提升，log4j2相较于log4j和logback都具有明显的性能提升，后面会有官方测试的数据。
- 自动重载配置，参考了logback的设计，当然会提供自动刷新参数配置，最实用的就是我们在生产上可以动态的修改日志的级别而不需要重启应用。

官网：<https://logging.apache.org/log4j/2.x/>

### 一、Log4j2入门

目前已经有三个门面了，其实不管是哪里都是江湖，都想写一个门面，一统江湖，所以log4j2除了提供日志实现以外，也拥有一套自己的独立的门面。

目前市面上最主流的日志门面就是SLF4j，虽然Log4j2也是日志门面，因为它的日志实现功能非常强大，性能优越。所以大家一般还是将Log4j2看作是日志的实现，Slf4j + Log4j2应该是未来的大势所趋。

#### 1、使用log4j-api做门面

## (1) 添加依赖

```
1 <!-- Log4j2 门面API-->
2 <dependency>
3     <groupId>org.apache.logging.log4j</groupId>
4     <artifactId>log4j-api</artifactId>
5     <version>2.14.1</version>
6 </dependency>
7 <!-- Log4j2 日志实现 -->
8 <dependency>
9     <groupId>org.apache.logging.log4j</groupId>
10    <artifactId>log4j-core</artifactId>
11    <version>2.14.1</version>
12 </dependency>
```

## (2) JAVA代码

```
1 public class TestLog4j2 {
2
3     private static final Logger LOGGER =
4         LogManager.getLogger(TestLog4j2.class);
5
6     @Test
7     public void testLog(){
8         LOGGER.fatal("fatal");
9         LOGGER.error("error");
10        LOGGER.warn("warn");
11        LOGGER.info("info");
12        LOGGER.debug("debug");
13        LOGGER.trace("trace");
14    }
15 }
```

结果:

```
"C:\Program Files\Java\jdk-11.0.4_windows-x64_bin\jdk-11.0.4\bin\java.exe" -ea -Didea
15:18:50.722 [main] FATAL com.ydlclass.TestLog4j2 - fatal
15:18:50.724 [main] ERROR com.ydlclass.TestLog4j2 - error
```

## 2、使用slf4j做门面

使用slf4j作为日志的门面，使用log4j2作为日志的实现。

```
1  <!-- Log4j2 门面API-->
2  <dependency>
3      <groupId>org.apache.logging.log4j</groupId>
4      <artifactId>log4j-api</artifactId>
5      <version>2.14.1</version>
6  </dependency>
7  <!-- Log4j2 日志实现 -->
8  <dependency>
9      <groupId>org.apache.logging.log4j</groupId>
10     <artifactId>log4j-core</artifactId>
11     <version>2.14.1</version>
12 </dependency>
13 <!--使用slf4j作为日志的门面,使用log4j2来记录日志 -->
14 <dependency>
15     <groupId>org.slf4j</groupId>
16     <artifactId>slf4j-api</artifactId>
17     <version>1.7.30</version>
18 </dependency>
19 <!--为slf4j绑定日志实现 log4j2的适配器 -->
20 <dependency>
21     <groupId>org.apache.logging.log4j</groupId>
22     <artifactId>log4j-slf4j-impl</artifactId>
23     <version>2.12.1</version>
24 </dependency>
```

```
1 private static final org.slf4j.Logger LOG =  
    LoggerFactory.getLogger(TestLog4j2.class);  
2  
3 @Test  
4 public void testSlf4j(){  
5     LOG.error("error");  
6     LOG.warn("warn");  
7     LOG.debug("debug");  
8     LOG.info("info");  
9     LOG.trace("trace");  
10 }
```

结果：

```
✓ Tests passed: 1 of 1 test - 256 ms  
"C:\Program Files\Java\jdk-11.0.4_windows-x64_bin\jdk-11.0.4\bin\java.exe"  
15:23:47.643 [main] ERROR com.ydlclass.TestLog4j2 - error  
  
Process finished with exit code 0
```

我们看到log4j2的默认日志级别好像是error。

## 二、Log4j2配置

### 1、默认配置：

DefaultConfiguration类中提供的默认配置将设置，  
通过debug可以在LoggerContext类中发现

```
1 private volatile Configuration configuration = new  
    DefaultConfiguration();
```

可以看到默认的root日志的layout

```

    root = {LoggerConfig@1843} "root"
    appenderRefs = {ArrayList@1856} size = 0
    appenders = {AppenderControlArraySet@1857} "AppenderControlArraySet [appenderArray=[Lorg.apache.logging.log4j.core.config.AppenderControl;@58695725]"
    appenderArray = {AtomicReference@1865} "[Lorg.apache.logging.log4j.core.config.AppenderControl;@58695725]"
    value = {AppenderControl@1867}
    0 = {AppenderControl@1869} "org.apache.logging.log4j.core.config.AppenderControl@204bf81a[appender=DefaultConsole-1, a..."
    recursive = {ThreadLocal@1871}
    appender = {ConsoleAppender@1872} "DefaultConsole-1"
    target = {ConsoleAppender$Target$1@1877} "SYSTEM_OUT"
    immediateFlush = true
    manager = {OutputStreamManager@1878}
    name = "DefaultConsole-1"
    ignoreExceptions = true
    layout = {PatternLayout@1879} "%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n"
    handler = {DefaultHandler@1880} "DefaultHandler@1880"

```

我们也能看到他的日志级别:

```

    root = {LoggerConfig@1843} "root"
    appenderRefs = {ArrayList@1856} size = 0
    appenders = {AppenderControlArraySet@1857} "AppenderControlArraySet [appenderArray=[Lorg.apache.logging.log4j.core.config.AppenderControl;@58695725]"
    name = ""
    logEventFactory = {ReusableLogEventFactory@1859}
    level = {Level@1860} "ERROR"
    additive = true

```

我们能从默认配置类中看到一些默认的配置:

```

1  protected void setToDefault() {
2      // LOG4J2-1176 facilitate memory leak
   investigation
3      setName(DefaultConfiguration.DEFAULT_NAME + "@" +
   Integer.toHexString(hashCode()));
4      final Layout<? extends Serializable> layout =
   PatternLayout.newBuilder()
5
6      .withPattern(DefaultConfiguration.DEFAULT_PATTERN)
7      .withConfiguration(this)
8      .build();
9      final Appender appender =
   ConsoleAppender.createDefaultAppenderForLayout(layout)
10
11      ;
12      appender.start();
13      addAppender(appender);
14      final LoggerConfig rootLoggerConfig =
   getRootLogger();
15      rootLoggerConfig.addAppender(appender, null,
   null);
16
17      final Level defaultLevel = Level.ERROR;
18      final String levelName =
   PropertiesUtil.getProperties().getStringProperty(DefaultConfiguration.DEFAULT_LEVEL,
19

```

```

16         defaultLevel.name());
17     final Level level = Level.valueOf(levelName);
18     rootLoggerConfig.setLevel(level != null ? level :
        defaultLevel);
19 }

```

## 5 自定义配置文件位置

log4j2默认在classpath下查找配置文件，可以修改配置文件的位置。在非web项目中：

```

1 public static void main(String[] args) throws
  IOException {
2     File file = new File("D:/log4j2.xml");
3     BufferedInputStream in = new
  BufferedInputStream(new FileInputStream(file));
4     final ConfigurationSource source = new
  ConfigurationSource(in);
5     Configurator.initialize(null, source);
6
7     Logger logger = LogManager.getLogger("mylog");
8 }

```

如果是web项目，在web.xml中添加

```

1 <context-param>
2     <param-name>log4jConfiguration</param-name>
3     <param-value>/WEB-INF/conf/log4j2.xml</param-value>
4 </context-param>
5 <listener>
6     <listener-
  class>org.apache.logging.log4j.web.Log4jServletContextL
  istener</listener-class>
7 </listener>

```

log4j2默认加载classpath下的 log4j2.xml 文件中的配置。事实上log4j2可以通过 XML、JSON、YAML 或properties格式进行配置：

<https://logging.apache.org/log4j/2.x/manual/configuration.html>

如果找不到配置文件，Log4j 将提供默认配置。DefaultConfiguration 类中提供的默认配置将设置：

- %d{HH:mm:ss.SSS}，表示输出到毫秒的时间
- %t，输出当前线程名称
- %-5level，输出日志级别，-5表示左对齐并且固定输出5个字符，如果不足在右边补0
- %logger，输出logger名称，因为Root Logger没有名称，所以没有输出
- %msg，日志文本
- %n，换行

其他常用的占位符有：

- %F，输出所在的类文件名，如Client.java
- %L，输出行号
- %M，输出所在方法名
- %l，输出语句所在的行数, 包括类名、方法名、文件名、行数

```
1 private void reconfigure(final URI configURI) {
2     Object externalContext =
        externalMap.get(EXTERNAL_CONTEXT_KEY);
3     final ClassLoader cl =
        ClassLoader.class.isInstance(externalContext) ?
        (ClassLoader) externalContext : null;
4     LOGGER.debug("Reconfiguration started for
        context[name={}] at URI {} ({{}) with optional
        ClassLoader: {}",
5                 contextName, configURI, this, cl);
```



```

6      final Configuration instance =
ConfigurationFactory.getInstance().getConfiguration(this, contextName, configURI, cl);
7      if (instance == null) {
8          LOGGER.error("Reconfiguration failed: No
configuration found for '{}' at '{}' in '{}'",
contextName, configURI, cl);
9      } else {
10         setConfiguration(instance);
11         /*
12             * instance.start(); Configuration old =
setConfiguration(instance); updateLoggers(); if (old
!= null) {
13                 * old.stop(); }
14             */
15         final String location = configuration == null
? "?" :
String.valueOf(configuration.getConfigurationSource());
;
16         LOGGER.debug("Reconfiguration complete for
context[name={}] at URI {} ({{}) with optional
ClassLoader: {}",
17             contextName, location, this, cl);
18     }
19 }

```

1 | ConfigurationFactory

```

1  for (final ConfigurationFactory factory :
    getFactories()) {
2      final String[] types =
        factory.getSupportedTypes();
3      if (types != null) {
4          for (final String type : types) {
5              if (type.equals(ALL_TYPES)) {
6                  final Configuration config =
                    factory.getConfiguration(loggerContext, name,
                    configLocation);
7                      if (config != null) {
8                          return config;
9                      }
10             }
11         }
12     }
13 }

```

```

v factories = {Collections$UnmodifiableRandomAccessList@2081} size = 4
> 0 = {PropertiesConfigurationFactory@2083}
> 1 = {YamlConfigurationFactory@2084}
> 2 = {JsonConfigurationFactory@2085}
> 3 = {XmlConfigurationFactory@2086}

```

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <Configuration status="warn" monitorInterval="5">
3      <properties>
4          <property name="LOG_HOME">D:/logs</property>
5      </properties>
6      <Appenders>
7          <Console name="Console" target="SYSTEM_OUT">
8              <PatternLayout pattern="%d{HH:mm:ss.SSS}
9              [%t] [%-5level] %c{36}:%L -
10             -- %m%n" />
              </Console>

```

```

11         <File name="file"
fileName="${LOG_HOME}/myfile.log">
12             <PatternLayout pattern "[%d{yyyy-MM-dd
HH:mm:ss.SSS}] [%-5level] %l
13                                     %c{36} - %m%n" />
14         </File>
15         <RandomAccessFile name="accessFile"
fileName="${LOG_HOME}/myAcclog.log">
16             <PatternLayout pattern "[%d{yyyy-MM-dd
HH:mm:ss.SSS}] [%-5level] %l
17                                     %c{36} - %m%n" />
18         </RandomAccessFile>
19         <RollingFile name="rollingFile"
fileName="${LOG_HOME}/myrollog.log"
20
filePattern="D:/logs/${date:yyyy-MM-dd}/myrollog-
%d{yyyy•MM-dd-HH-mm}-%i.log">
21             <ThresholdFilter level="debug"
onMatch="ACCEPT" onMismatch="DENY" />
22             <PatternLayout pattern "[%d{yyyy-MM-dd
HH:mm:ss.SSS}] [%-5level] %l
23                                     %c{36} - %msg%n"
/>
24         <Policies>
25             <OnStartupTriggeringPolicy />
26             <SizeBasedTriggeringPolicy size="10
MB" />
27             <TimeBasedTriggeringPolicy />
28         </Policies>
29         <DefaultRolloverStrategy max="30" />
30     </RollingFile>
31     <RollingRandomAccessFile name="MyFile"
fileName="${LOG_HOME}/${FILE_NAME}.log"
32     filePattern="${LOG_HOME}/${date:yyyy-
MM}/${FILE_NAME}-%d{yyyy-MM-dd HH-mm}-%i.log">
33         <PatternLayout
34             pattern="%d{yyyy-MM-dd HH:mm:ss.SSS}
[%t] %-5level %logger{36} - %msg%n" />
35         <Policies>
36             <TimeBasedTriggeringPolicy
interval="1" />

```

```

38         <SizeBasedTriggeringPolicy size="10
    MB" />
39     </Policies>
40     <DefaultRolloverStrategy max="20" />
41 </RollingRandomAccessFile>
42 </Appenders>
43 <Loggers>
44     <Logger name="mylog" level="trace"
    additivity="false">
45         <AppenderRef ref="MyFile" />
46     </Logger>
47     <Root level="error">
48         <AppenderRef ref="Console" />
49     </Root>
50 </Loggers>
51 </Configuration>
52

```

注意根节点增加了一个monitorInterval属性，含义是每隔300秒重新读取配置文件，可以不重启应用的情况下修改配置，还是很好用的功能。

RollingRandomAccessFile的属性：

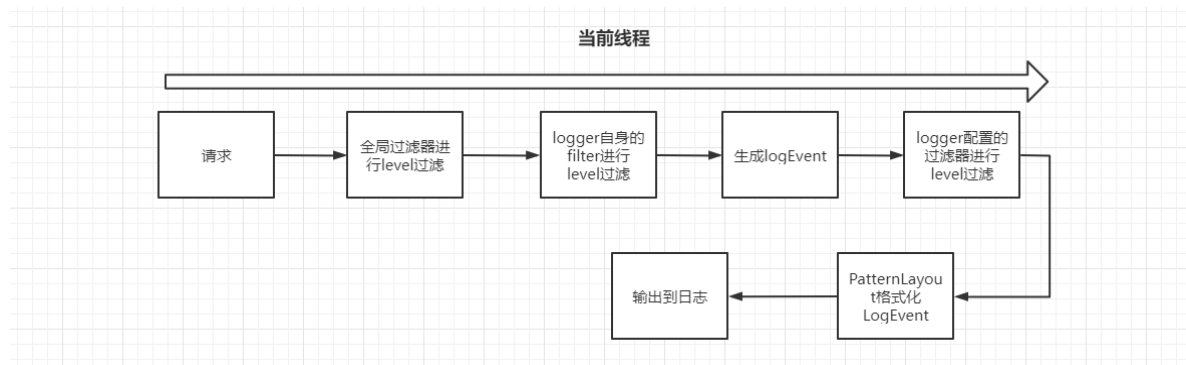
- fileName 指定当前日志文件的位置和文件名称
- filePattern 指定当发生Rolling时，文件的转移和重命名规则
- SizeBasedTriggeringPolicy 指定当文件体积大于size指定的值时，触发Rolling
- DefaultRolloverStrategy 指定最多保存的文件个数
- TimeBasedTriggeringPolicy 这个配置需要和filePattern结合使用，
- 注意filePattern中配置的文件重命名规则是\${FILE\_NAME}-%d{yyyy-MM-dd HH-mm}-%i，最小的时间粒度是mm，即分钟。
- TimeBasedTriggeringPolicy指定的size是1，结合起来就是每1分钟生成一个新文件。如果改成%d{yyyy-MM-dd HH}，最小粒度为小时，则每一个小时生成一个文件。

## 三、Log4j2异步日志

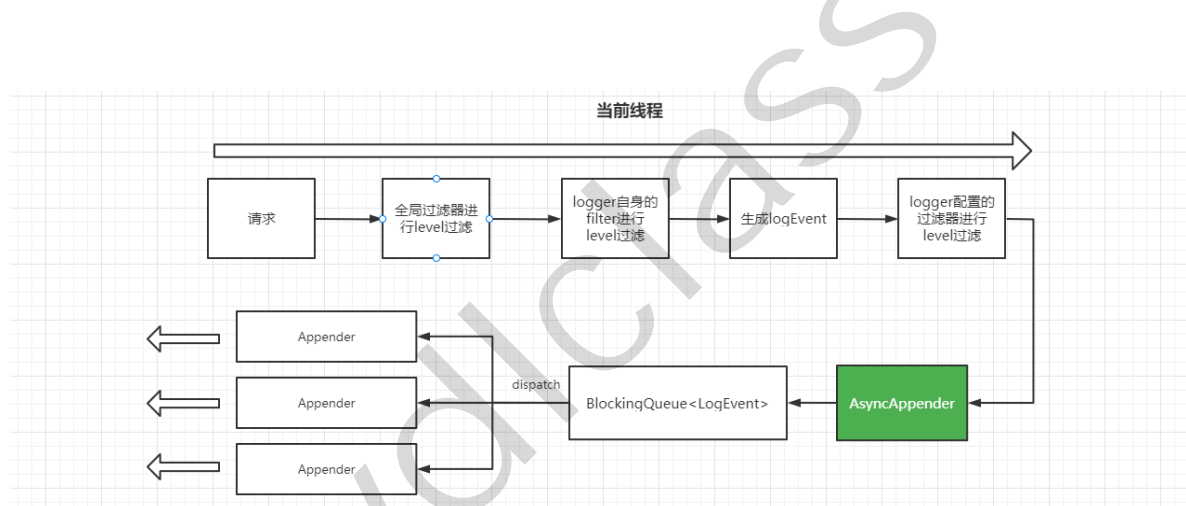
### 异步日志

log4j2最大的特点就是异步日志，其性能的提升主要也是从异步日志中受益，我们来看看如何使用 log4j2的异步日志。

- 同步日志



- 异步日志



```
public final class AsyncAppender extends AbstractAppender {  
  
    private static final int DEFAULT_QUEUE_SIZE = 1024;  
  
    private final BlockingQueue<LogEvent> queue;  
    private final int queueSize;  
}
```

Log4j2提供了两种实现日志的方式，一个是通过AsyncAppender，一个是通过AsyncLogger，分别对应 前面我们说的Appender组件和Logger组件。

注意：配置异步日志需要添加依赖

```
1 <!--异步日志依赖-->
2 <dependency>
3     <groupId>com.lmax</groupId>
4     <artifactId>disruptor</artifactId>
5     <version>3.3.4</version>
6 </dependency>
```

## 1. AsyncAppender方式

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Configuration status="warn">
3     <properties>
4         <property name="LOG_HOME">D:/logs</property>
5     </properties>
6     <Appenders>
7         <File name="file"
8             fileName="${LOG_HOME}/myfile.log">
9             <PatternLayout>
10                <Pattern>%d %p %c{1.} [%t]
11                %m%n</Pattern>
12            </PatternLayout>
13        </File>
14        <Async name="Async">
15            <AppenderRef ref="file"/>
16        </Async>
17    </Appenders>
18    <Loggers>
19        <Root level="error">
20            <AppenderRef ref="Async"/>
21        </Root>
22    </Loggers>
23 </Configuration>
```

## 2. AsyncLogger方式

AsyncLogger才是log4j2 的重头戏，也是官方推荐的异步方式。它可以使得调用Logger.log返回的 更快。你可以有两种选择：全局异步和混合异步。

- **全局异步**就是，所有的日志都异步的记录，在配置文件上不用做任何改动，只需要添加一个 log4j2.component.properties 配置；

```
1 Log4jContextSelector=org.apache.logging.log4j.core.async
  c.AsyncLoggerContextSelector
2
```

- **混合异步**就是，你可以在应用中同时使用同步日志和异步日志，这使得日志的配置方式更加灵活。

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Configuration status="WARN">
3     <properties>
4         <property name="LOG_HOME">D:/logs</property>
5     </properties>
6     <Appenders>
7         <File name="file"
8             fileName="${LOG_HOME}/myfile.log">
9             <PatternLayout>
10                <Pattern>%d %p %c{1.} [%t]
11                %m%n</Pattern>
12            </PatternLayout>
13        </File>
14        <Async name="Async">
15            <AppenderRef ref="file"/>
16        </Async>
17    </Appenders>
18    <Loggers>
19        <AsyncLogger name="com.ydlclass" level="trace"
20            includeLocation="false"
21            additivity="false">
22            <AppenderRef ref="file"/>
23        </AsyncLogger>
24        <Root level="info" includeLocation="true">
25            <AppenderRef ref="file"/>
26        </Root>
27    </Loggers>
28 </Configuration>
```

```
23         </Root>
24     </Loggers>
25 </Configuration>
26
```

如上配置： com.ydlclass 日志是异步的， root日志是同步的。

使用异步日志需要注意的问题：

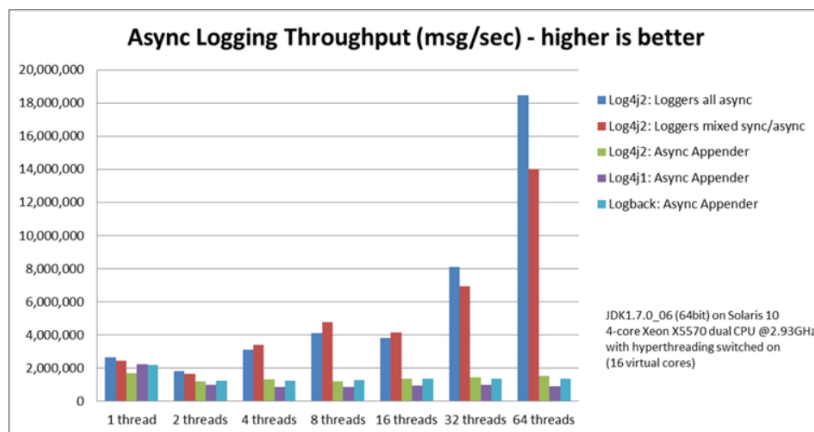
1. 如果使用异步日志， AsyncAppender、 AsyncLogger和全局日志， 不要同时出现。性能会和 AsyncAppender一致， 降至最低。
2. 设置includeLocation=false ， 打印位置信息会急剧降低异步日志的性能， 比同步日志还要慢。

```
1  for (int i = 0; i < 1000000; i++) {
2      LOGGER.fatal("fatal");
3
4  }
5  long end = System.currentTimeMillis();
6  System.out.println(end - start);
7
8
9  2970
```

### Log4j2的性能

log4j官网对其性能进行大肆宣扬，但是网上也有专业认识进行测试，log4j在大量日志的情况下有一定的优势，他确实是日后的选择。但是也不必纠结。





[Log4j - Performance \(apache.org\)](http://log4j.apache.org/Performance)

## 第七章：怎么打日志

### 基本格式

必须使用参数化信息的方式:

```
1 logger.debug("Processing trade with id:{{}} and symbol  
: {{}} ", id, symbol);
```

不要进行字符串拼接,那样会产生很多String对象, 占用空间, 影响性能。  
反例(不要这么做):

```
1 logger.debug("Processing trade with id: " + id + "  
symbol: " + symbol);
```

使用[]进行参数变量隔离, 如有参数变量, 应该写成如下写法:

```
1 logger.debug("Processing trade with id:{{}} and symbol  
: {{}} ", id, symbol);
```

这样的格式写法, 可读性更好, 对于排查问题更有帮助。不同级别的使用

### ERROR, 影响到程序正常运行、当前请求正常运行的异常情况:

- 打开配置文件失败

- 所有第三方对接的异常(包括第三方返回错误码)
- 所有影响功能使用的异常, 包括:SQLException和除了业务异常之外的所有异常(RuntimeException和Exception)
- 不应该出现的情况, 比如要使用阿里云传图片, 但是未响应
- 如果有Throwable信息, 需要记录完成的堆栈信息:

```
1 | log.error("获取用户[{}]的用户信息时出错",userName,e);
```

说明, 如果进行了抛出异常操作, 请不要记录error日志, 由最终处理方进行处理:

反例(不要这么做):

```
1 | try{
2 |     ....
3 | }catch(Exception ex){
4 |     String errorMessage=String.format("Error while
5 |     reading information of user [%s]",userName);
6 |     logger.error(errorMessage,ex);
7 |     throw new UserServiceException(errorMessage,ex);
8 | }
```

### **WARN, 不应该出现但是不影响程序、当前请求正常运行的异常情况:**

1. 有容错机制的时候出现的错误情况
2. 找不到配置文件, 但是系统能自动创建配置文件
3. 即将接近临界值的时候, 例如: 缓存池占用达到警告线, 业务异常的记录, 比如:用户锁定异常

### **INFO, 系统运行信息**

1. Service方法中对于系统/业务状态的变更
2. 主要逻辑中的分步骤: 1, 初始化什么 2、加载什么
3. 外部接口部分
4. 客户端请求参数(REST/WS)
5. 调用第三方时的调用参数和调用结果

6. 对于复杂的业务逻辑，需要进行日志打点，以及埋点记录，比如电商系统中的下订单逻辑，以及OrderAction操作(业务状态变更)。
7. 调用其他第三方服务时，所有的出参和入参是必须要记录的(因为你很难追溯第三方模块发生的问题)

#### 说明

并不是所有的service都进行出入口打点记录,单一、简单service是没有意义的(job除外,job需要记录开始和结束,)。反例(不要这么做):

```
1 public List listByBaseType(Integer baseTypeId) {
2     log.info("开始查询基地");
3     BaseExample ex=new BaseExample();
4     BaseExample.Criteria ctr = ex.createCriteria();
5     ctr.andIsDeleteEqualTo(IsDelete.USE.getValue());
6     Optionals.doIfPresent(baseTypeId,
7         ctr::andBaseTypeIdEqualTo);
8     log.info("查询基地结束");
9     return baseRepository.selectByExample(ex);
10 }
```

**DEBUG，可以填写所有的想知道的相关信息(但不代表可以随便写，debug信息要有意义,最好有相关参数)**

生产环境需要关闭DEBUG信息

如果在生产情况下需要开启DEBUG,需要使用开关进行管理，不能一直开启。

#### 说明

如果代码中出现以下代码，可以进行优化:

//1. 获取用户基本薪资

//2. 获取用户休假情况

//3. 计算用户应得薪资

```

1 logger.debug("开始获取员工[{}] [{}]年基本薪
   资",employee,year);
2 logger.debug("获取员工[{}] [{}]年的基本薪资为
   [{}]",employee,year,basicSalary);
3 logger.debug("开始获取员工[{}] [{}]年[{}]月休假情
   况",employee,year,month);
4 logger.debug("员工[{}][{}]年[{}]月年假/病假/事假为
   [{}]/[{}]/[{}]",employee,year,month,annualLeaveDays,sic
   kLeaveDays,noPayLeaveDays);
5 logger.debug("开始计算员工[{}][{}]年[{}]月应得薪
   资",employee,year,month);
6 logger.debug("员工[{}] [{}]年[{}]月应得薪资为
   [{}]",employee,year,month,actualSalary);

```

**TRACE**，特别详细的系统运行完成信息，业务代码中，不要使用。(除非有特殊用意，否则请使用DEBUG级别替代)

#### 规范示例说明

```

1 @Override
2 @Transactional
3 public void createUserAndBindMobile(@NotBlank String
   mobile, @NotNull User user) throws
   CreateConflictException{
4     boolean debug = log.isDebugEnabled();
5     if(debug){
6         log.debug("开始创建用户并绑定手机号. args[mobile=
           [{}],user=[{}]]", mobile, LogObjects.toString(user));
7     }
8     try {
9         user.setCreateTime(new Date());
10        user.setUpdateTime(new Date());
11        userRepository.insertSelective(user);
12        if(debug){
13            log.debug("创建用户信息成功. insertedUser=
              [{}]",LogObjects.toString(user));
14        }

```

```
15         UserMobileRelationship relationship = new
UserMobileRelationship();
16         relationship.setMobile(mobile);
17         relationship.setOpenId(user.getOpenId());
18         relationship.setCreateTime(new Date());
19         relationship.setUpdateTime(new Date());
20
        userMobileRelationshipRepository.insertOnDuplicateKey
        (relationship);
21         if(debug){
22             log.debug("绑定手机成功. relationship=
[{}]",LogObjects.toString(relationship));
23         }
24         log.info("创建用户并绑定手机号. userId=
[{}],openId=
[{}],mobile=[{}]",user.getId(),user.getOpenId(),mobile)
;           // 如果考虑安全, 手机号记得脱敏
25     }catch(DuplicateKeyException e){
26         log.info("创建用户并绑定手机号失败, 已存在相同的用户.
openId=[{}],mobile=[{}]",user.getOpenId(),mobile);
27         throw new CreateConflictException("创建用户发生冲
突, openid=[%s]",user.getOpenId());
28     }
29 }
```