学习目标

1·学习java日志体系及日志工具的演进

2·了解日志采集、处理、分析等各阶段的常用中间件

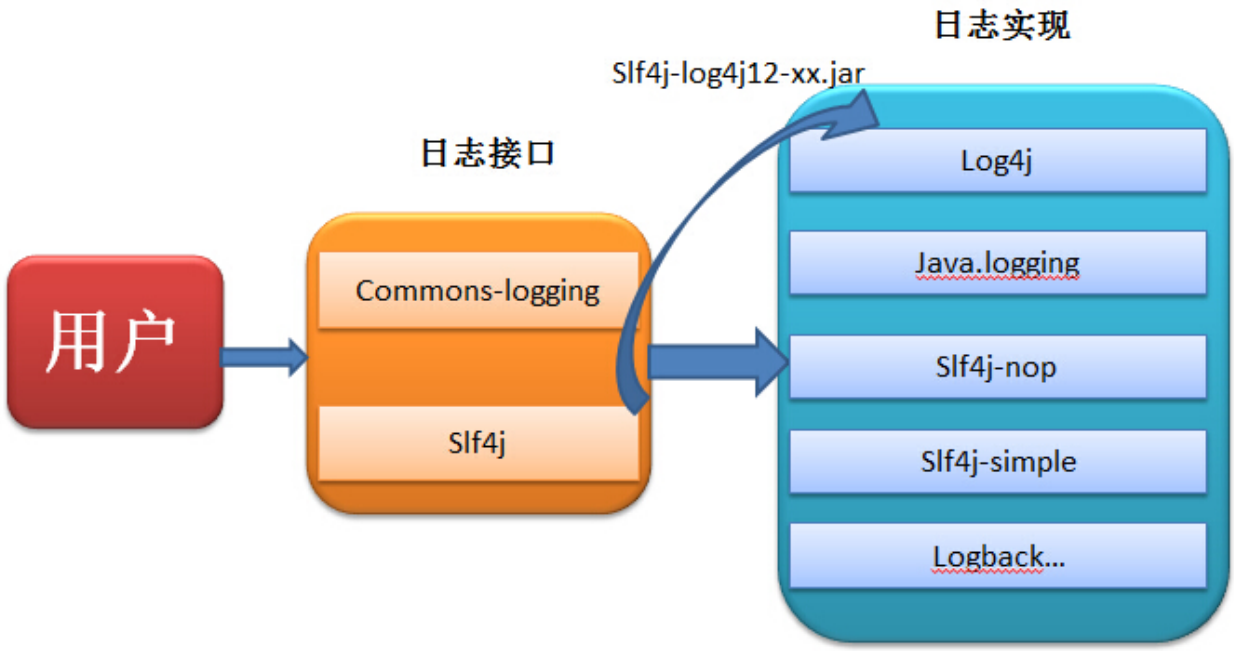
3·学会搭建完整的elk日志平台

4·学习日志打点，切面，日志文件等输出手段

5·项目实战，完成一访问日志链路的跟踪

1、java日志体系

1.1 体系概述



1.1.1 日志接口

- JCL：Apache基金会所属的项目，是一套Java日志接口，之前叫Jakarta Commons Logging，后更名为Commons Logging，简称JCL
- SLF4J：Simple Logging Facade for Java，缩写Slf4j，是一套简易Java日志门面，只提供相关接口，和其他日志工具之间需要桥接

1.1.2 日志实现

- JUL：JDK中的日志工具，也称为jdklog、jdk-logging，自Java1.4以来sun的官方提供。
- Log4j：隶属于Apache基金会的一套日志框架，现已不再维护
- Log4j2：Log4j的升级版本，与Log4j变化很大，不兼容
- Logback：一个具体的日志实现框架，和Slf4j是同一个作者，性能很好

1.2 发展历程

1.2.1 上古时代

在JDK 1.3及以前，Java打日志依赖System.out.println(), System.err.println()或者e.printStackTrace()，Debug日志被写到STDOUT流，错误日志被写到STDERR流。这样打日志有一个非常大的缺陷，非常机械，无法定制，且日志粒度不够细分。

代码：

```
System.out.println("123");
System.err.println("456");
```

1.2.2 开创先驱

于是，Ceki Gülcü 于2001年发布了Log4j，并将其捐献给了Apache软件基金会，成为Apache 基金会的顶级项目。后来衍生支持C, C++, C#, Perl, Python, Ruby等语言。 Log4j 在设计上非常优秀，它定义的Logger、Appender、Level等概念对后续的 Java Log 框架有深远的影响，如今的很多日志框架基本沿用了这种思想。Log4j 的性能是个问题，在Logback 和 Log4j2 出来之后，2015年9月，Apache软件基金会宣布，Log4j不再维护，建议所有相关项目升级到Log4j2

pom：

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
<dependency>
    <groupId>log4j</groupId>
    <artifactId>apache-log4j-extras</artifactId>
    <version>1.2.17</version>
</dependency>
```

配置：

```
log4j.rootLogger=debug

#console
```

```
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern= log4j:[%d{yyyy-MM-dd HH:mm:ss
a}]:%p %l%m%n

#dailyfile
log4j.appender.dailyfile=org.apache.log4j.DailyRollingFileAppender
log4j.appender.dailyfile.DatePattern='_'yyyy-MM-dd'.log'
log4j.appender.dailyfile.File=./log4j.log
log4j.appender.dailyfile.Append=true
log4j.appender.dailyfile.Threshold=INFO
log4j.appender.dailyfile.layout=org.apache.log4j.PatternLayout
log4j.appender.dailyfile.layout.ConversionPattern=log4j:[%d{yyyy-MM-dd HH:mm:ss
a}] [Thread: %t][ Class:%c >> Method: %l ]%n%p:%m%n
```

代码：

```
import org.apache.log4j.Logger;

Logger logger = Logger.getLogger(Demo.class);
logger.info("xxxx");
```

1.2.3 搞事情的JUL

sun公司对于log4j的出现内心隐隐表示嫉妒。于是在jdk1.4版本后，开始搞事情，增加了一个包为java.util.logging，简称为JUL，用以对抗log4j ，但是却给开发造成了麻烦。相互引用的项目之间可能使用了不同的日志框架，经常将代码搞得一片混乱。

代码：

```
import java.util.logging.Logger;

Logger loggger = Logger.getLogger(Demo.class.getName());
logger.finest("xxxx");
```

配置路径：

```
$JAVA_HOME/jre/lib/logging.properties
```

JUL功能远不如log4j完善，自带的Handlers有限，性能和可用性上也一般，JUL在Java1.5以后才有所提升。

1.2.4 JCL应运而生

从上面可以看出，JUL的api与log4j是完全不同的（参数只接受string）。由于日志系统互相没有关联，彼此没有约定，不同人的代码使用不同日志，替换和统一也就变成了比较棘手的一件事。假如你的应用使用log4j，然后项目引用了一个其他团队的库，他们使用了JUL，你的应用就得使用两个日志系统了，然后其他团队又使用了simplelog……这个时候如果要调整日志的输出级别，用于跟踪某个信息，简直就是一场灾难。

那这个状况该如何解决呢？答案就是进行抽象，抽象出一个接口层，对每个日志实现都适配或者转接，这样这些提供给别人的库都直接使用抽象层即可 ，以后调用的时候，就调用这些接口。（面向接口思想）

于是，JCL(Jakarta Commons Logging)应运而生，也就是commons-logging-xx.jar组件。JCL 只提供 log 接口，具体的实现则在运行时动态寻找。这样一来组件开发者只需要针对 JCL 接口开发，而调用组件的应用程序则可以在运行时搭配自己喜好的日志实践工具。

那接口下真实的日志是谁呢？参考下图：

JCL会在ClassLoader中进行查找，如果能找到Log4j 则默认使用log4j 实现，如果没有则使用JUL(jdk自带的) 实现，再没有则使用JCL内部提供的SimpleLog 实现。（代码验证）

pom：

```
<dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
    <version>1.2</version>
</dependency>
```

代码：

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

Log log =LogFactory.getLog(Demo.class);
log.info('xxx');
```

JCL缺点也很明显，一是效率较低，二是容易引发混乱，三是JCL的机制有很大的可能会引发内存泄露。

同时，JCL的书写存在一个不太优雅的地方，典型的场景如下：

假如要输出一条debug日志，而一般情况下，生产环境 log 级别都会设到 info 或者以上，那这条 log 是不会被输出的。于是，在代码里就出现了

```
logger.debug("this is a debug info , message :" + msg);
```

这个有什么问题呢？虽然生产不会打出日志，但是这其中都会做一个字符串连接操作，然后生成一个新的字符串。如果这条语句在循环或者被调用很多次的函数中，就会多做很多无用的字符串连接，影响性能。

所以，JCL推荐的写法如下：

```
if (logger.isDebugEnabled()) {
    logger.debug("this is a debug info , message :" + msg);
}
```

虽然解决了问题，但是这个代码实在看上去不怎么舒服...

1.2.5 再起波澜

于是，针对以上情况，log4j的作者再次出手，他觉得JCL不好用，自己又写了一个新的接口api，就是slf4j，并且为了追求更极致的性能，新增了一套日志的实现，就是logback，一时间烽烟又起.....

坐标：

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.30</version>
</dependency>
```

```
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-core</artifactId>
    <version>1.2.3</version>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.2.3</version>
</dependency>
```

配置：

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

<configuration scan="true" scanPeriod="60 seconds" debug="false">

    <property name="logPattern" value="logback:[ %-5level] [%date{yyyy-MM-dd
HH:mm:ss.SSS}] %logger{96} [%line] [%thread]- %msg%n"></property>
```

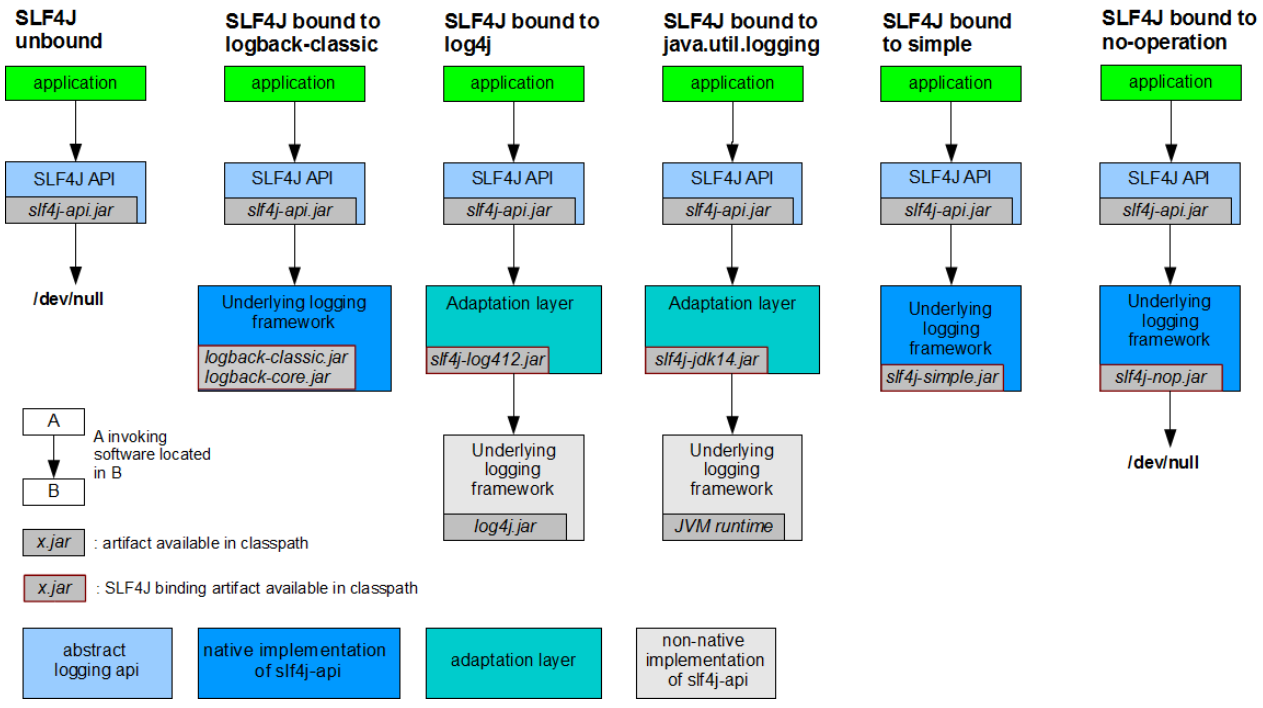
```
    <!-- 控制台的标准输出 -->
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <charset>UTF-8</charset>
            <pattern>${logPattern}</pattern>
        </encoder>
    </appender>

    <root level="DEBUG">
        <appender-ref ref="STDOUT"/>
    </root>

</configuration>
```

logback-core 提供基础抽象，logback-classic 提供日志实现，并且直接就是基于Slf4j API。所以slf4j配合logback来完成日志时，不需要像其他的日志框架一样提供适配器。

slf4j本身并没有实际的日志输出能力，它底层还是需要去调用具体的日志框架API，也就是它需要跟具体的日志框架结合使用。由于具体日志框架比较多，而且互相也大都不兼容，日志门面接口要想实现与任意日志框架结合就需要额外对应的桥接器。



有了新的slf4j后，上面的字符串拼接问题，被以下代码所取代，而logback也提供了更高级的特性，如异步 logger，Filter等。

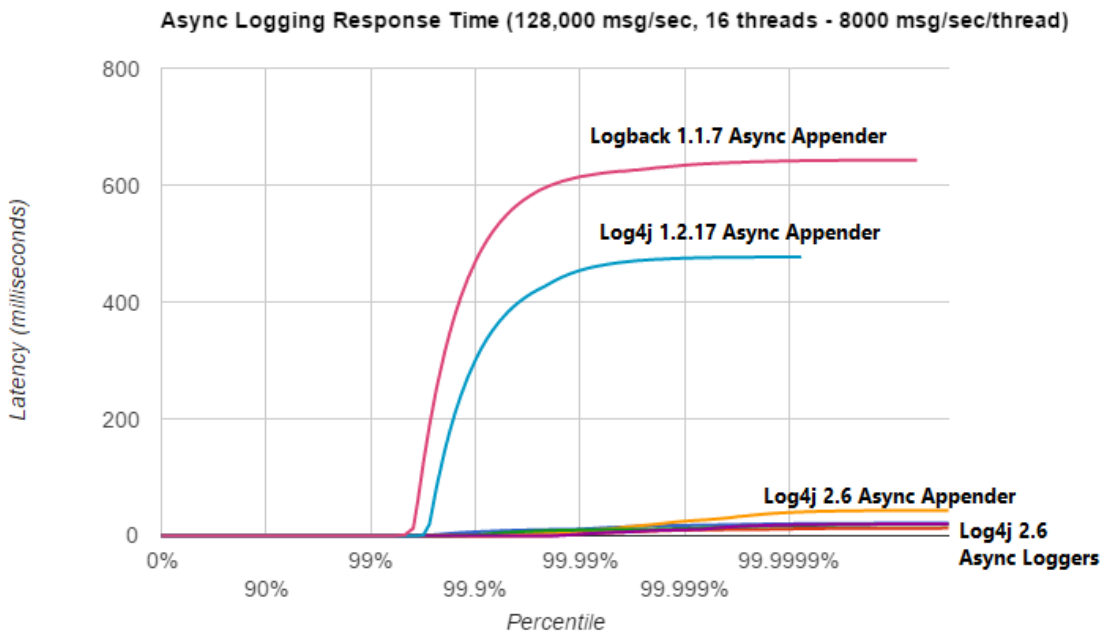
```
logger.debug("this is a debug info , message : {}", msg);
```

事情结束了吗？没有，log4j的粉丝们并不开心，于是下一代诞生了.....

1.2.6 再度青春

前面提到，log4j由apache宣布，2015年后，不再维护。推荐大家升级到log4j2，虽然log4j2沿袭了log4j的思想，然而log4j2和log4j完全是两码事，并不兼容。

log4j2以性能著称，它比其前身Log4j 1.x提供了重大改进，同时类比logback，它提供了Logback中可用的许多改进，同时修复了Logback架构中的一些固有问题。功能上，它有着和Logback相同的基本操作，同时又有自己独特的部分，比如：插件式结构、配置文件优化、异步日志等。



pom：

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.13.0</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.13.0</version>
</dependency>
<dependency>
    <groupId>com.lmax</groupId>
    <artifactId>disruptor</artifactId>
    <version>3.4.1</version>
</dependency>
```

代码：

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

Logger logger = LogManager.getLogger(Demo.class);

logger.debug("debug Msg");
```

配置：

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

<configuration status="info" monitorInterval="30">
    <Properties>
        <Property name="pattern">log4j2:[%-5p]:%d{YYYY-MM-dd HH:mm:ss} [%t]
%c{1}:%L - %msg%n</Property>
    </Properties>

    <appenders>
        <!--console :控制台输出的配置-->
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="${pattern}"/>
        </Console>
    </appenders>

    <loggers>
        <logger name="org.springframework" level="INFO"></logger>
        <root level="info">
            <appender-ref ref="Console"/>
        </root>
    </loggers>
</configuration>
```

到log4j2，轰轰烈烈的java log战役基本就结束了。下面的章节，我们将进入日志工具的详细配置阶段。

1.3 配置讲解

1.3.1 概述

1）日志级别

一个完整的日志组件都要具备日志级别的概念，每种日志组件级别定义不同，日常编码最经常用到的主流分级如下（由低到高）：

- trace：路径跟踪
- debug：一般用于日常调式

- info：打印重要信息
- warn：给出警告
- error：出现错误或问题

每个日志组件的具体级别划分稍有不同，参考下文各章节。

2）日志组件

- appender：日志输出目的地，负责日志的输出 （输出到什么 地方）
- logger：日志记录器，负责收集处理日志记录 （如何处理日志）
- layout：日志格式化，负责对输出的日志格式化（以什么形式展现）

1.3.2 jul

1）配置文件：

默认情况下配置文件路径为$JAVAHOME\jre\lib\logging.properties

可以指定配置文件：

```
static {
    System.setProperty("java.util.logging.config.file",

 Demo.class.getClassLoader().getResource("logging.properties").getPath());
}
```

代码实践：配置文件位置，日志输出级别，如何输出低于INFO级别的信息

2）级别：

- SEVERE（最高值）
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST（最低值）
- OFF，关闭日志。
- ALL，启用所有日志。

3）处理器：

- StreamHandler：日志记录写入OutputStream。
- ConsoleHandler：日志记录写入System.err。
- FileHandler：日志记录写入单个文件或一组滚动日志文件。

- SocketHandler：日志记录写入远程TCP端口的处理程序。
- MemoryHandler：缓冲内存中日志记录。

4）格式化：

- SimpleFormatter：格式化为简短的日志记录摘要。
- XMLFormatter：格式化为详细的XML结构信息。
- 可自定输出格式，继承抽象类java.util.logging.Formatter即可。

5) 代码实践：

- 如何修改输出级别?

1.3.3 log4j

1）配置文件：

- 启动时，默认会寻找source folder下的log4j.xml
- 若没有，会寻找log4j.properties

2）级别：

- FATAL（最高）
- ERROR
- WARN
- INFO
- DEBUG （最低）
- OFF，关闭日志。
- ALL，启用所有日志。

3）处理器：

- org.apache.log4j.ConsoleAppender（控制台）
- org.apache.log4j.FileAppender（文件）
- org.apache.log4j.DailyRollingFileAppender（每天产生一个日志文件）
- org.apache.log4j.RollingFileAppender（文件大小到达指定尺寸的时候产生一个新的文件）
- org.apache.log4j.WriterAppender（将日志信息以流格式发送到任意指定的地方）

4）格式化：

- org.apache.log4j.HTMLLayout（以HTML表格形式布局）
- org.apache.log4j.PatternLayout（可以灵活地指定布局模式）
- org.apache.log4j.SimpleLayout（包含日志信息的级别和信息字符串）
- org.apache.log4j.TTCCLayout（包含日志产生的时间、线程、类别等等信息）

5) 代码实践：

- Appender：Console，DailyRollingFile
- Layout：Pattern

```
log4j.rootLogger=debug,stdout,dailyfile

#console
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern= [%d{yyyy-MM-dd HH:mm:ss a}]:%p
%l%m%n

#dailyfile
log4j.appender.dailyfile=org.apache.log4j.DailyRollingFileAppender
log4j.appender.dailyfile.DatePattern='_'yyyy-MM-dd'.log'
log4j.appender.dailyfile.File=./log4j.log
log4j.appender.dailyfile.Append=true
log4j.appender.dailyfile.Threshold=INFO
log4j.appender.dailyfile.layout=org.apache.log4j.PatternLayout
log4j.appender.dailyfile.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss a}
[Thread: %t][ Class:%c >> Method: %l ]%n%p:%m%n
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration debug="true"
xmlns:log4j='http://jakarta.apache.org/log4j/'>

    <appender name="CONSOLE" class="org.apache.log4j.ConsoleAppender">
        <param name="Target" value="System.out"/>
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern"
                   value=""/>
        </layout>
        <filter class="org.apache.log4j.varia.LevelRangeFilter">
            <param name="LevelMin" value="DEBUG"/>
            <param name="LevelMax" value="DEBUG"/>
        </filter>
    </appender>
    <appender name="DAILYROLLINGFILE"
class="org.apache.log4j.DailyRollingFileAppender">
        <param name="File" value="log4j.log"/>
        <param name="DatePattern" value="yyyy-MM-dd"/>
        <param name="Append" value="true"/>
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern"
                   value="%d{yyyy-MM-dd HH:mm:ss a} [Thread: %t][ Class:%c
Method: %l ]%n%p:%m%n"/>
        </layout>
```

```
    </appender>


    <root>
        <appender-ref ref="CONSOLE"/>
        <appender-ref ref="DAILYROLLINGFILE"/>
    </root>
</log4j:configuration>
```

语法	说明
%c	a.b.c
%c{2}	b.c
%20c	（若名字空间长度小于20，则左边用空格填充）
%-20c	（若名字空间长度小于20，则右边用空格填充）
%.30c	（若名字空间长度超过30，截去多余字符）
%20.30c	（若名字空间长度小于20，则左边用空格填充；若名字空间长度超过30，截去多余字符）
%-20.30c	（若名字空间长度小于20，则右边用空格填充；若名字空间长度超过30，截去多余字符）
%C	org.apache.xyz.SomeClass
%C{1}	SomeClass
%d{yyyy/MM/dd HH:mm:ss,SSS}	2000/10/12 11:22:33,117
%d{ABSOLUTE}	11:22:33,117
%d{DATE}	12 Oct 2000 11:22:33,117
%d{ISO8601}	2000-10-12 11:22:33,117
%F	MyClass.java
%l	MyClass.main(MyClass.java:123)
%L	123
%m	This is a message for debug.
%M	main
%n	Windows平台下表示rn，UNIX平台下表示n
%p	INFO
%r	1215
%t	MyClass
%%	%

1.3.4 logback

http://logback.qos.ch/manual/index.html

1）配置文件：

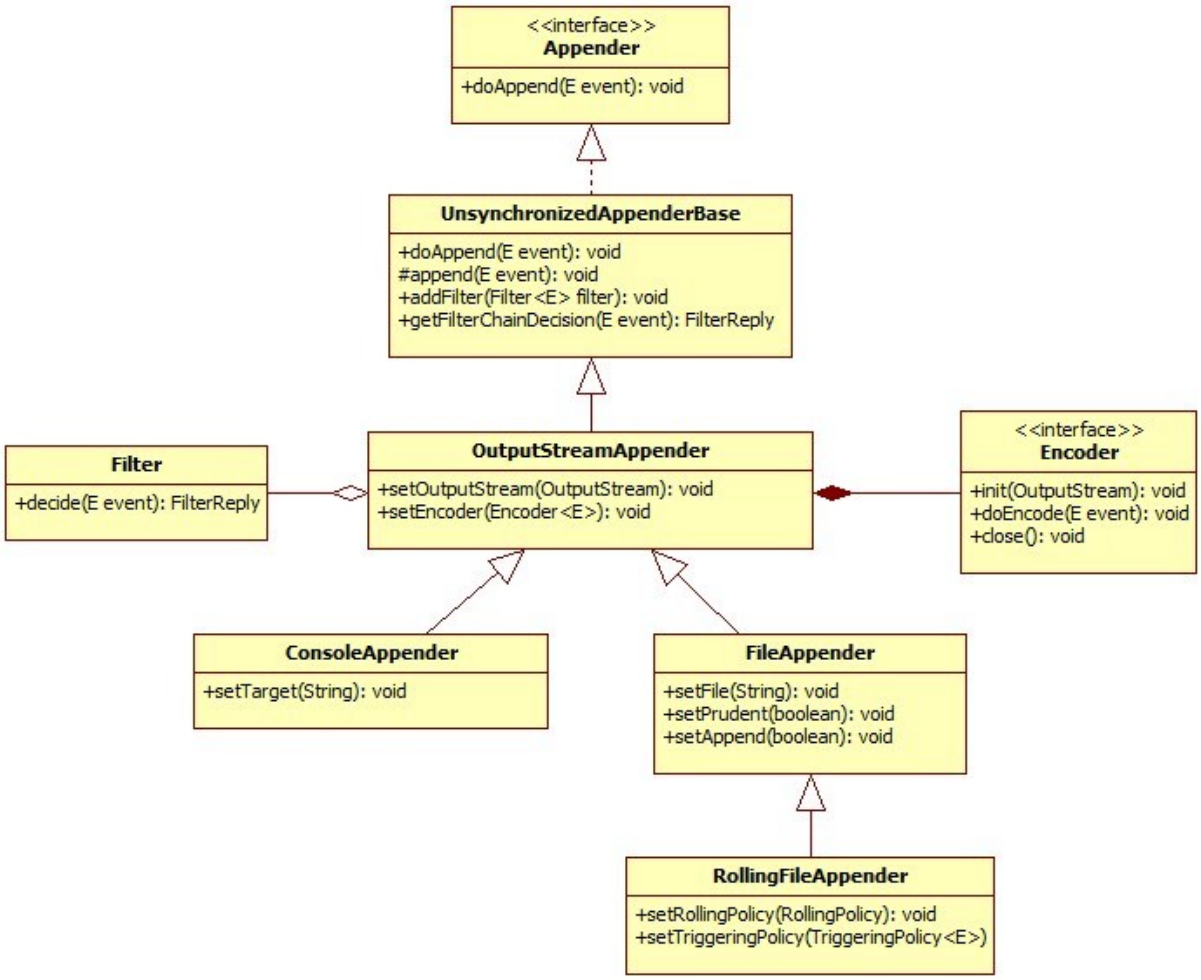
- Logback tries to find a file called `logback-test.xml` in the classpath.
- If no such file is found, logback tries to find a file called `logback.groovy` in the classpath.
- If no such file is found, it checks for the file `logback.xml` in the classpath..
- If no such file is found, service-provider 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.
- 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）级别：

- 日志打印级别 ALL > TRACE > FATAL > DEBUG > INFO > WARN > ERROR > OFF

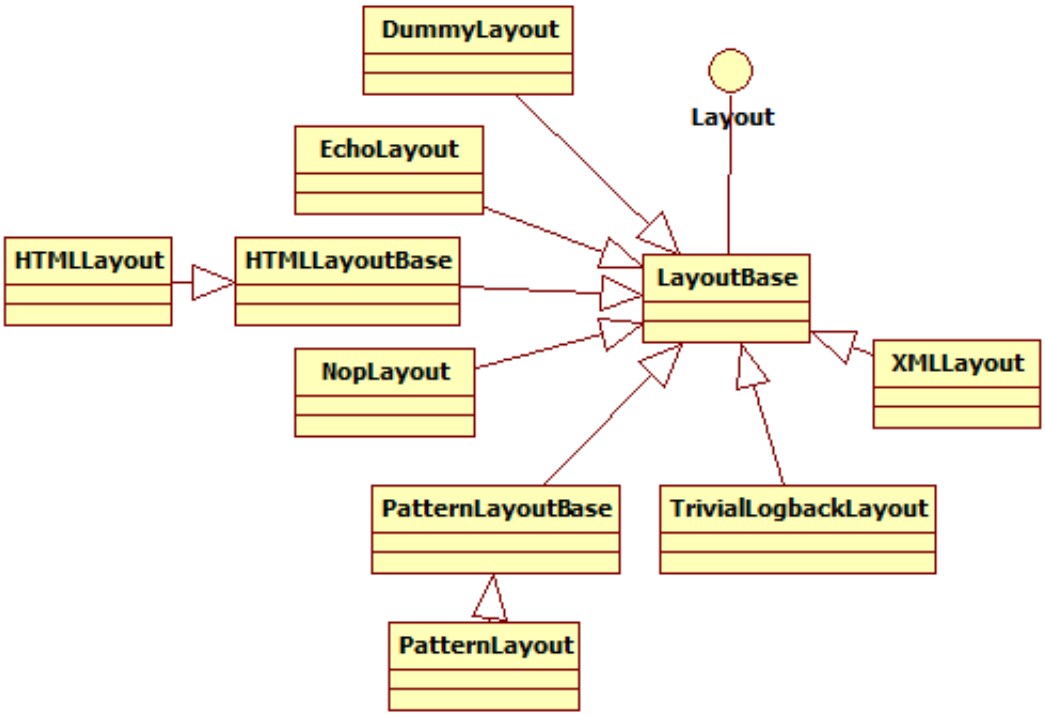
3）处理器：

http://logback.qos.ch/manual/appenders.html



4）格式化：

http://logback.qos.ch/manual/layouts.html



5) 代码实战：

- Appender：Console，Rollingfile
- Layout：Xml，Pattern，Html , 自定义

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration scan="true" scanPeriod="3 seconds" debug="false">
    <!-- lOGGER  PATTERN 根据个人喜好选择匹配  -->
    <property name="logPattern" value="logback:[ %-5level] [%date{yyyy-MM-dd
HH:mm:ss.SSS}] %logger{96} [%line] [%thread]- %msg%n"></property>

    <!-- 动态日志级别 -->
    <jmxConfigurator/>

    <!-- 控制台的标准输出 -->
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <charset>UTF-8</charset>
            <pattern>${logPattern}</pattern>
        </encoder>
    </appender>

    <!-- 滚动文件  -->
    <appender name="ROLLING_FILE"
 class="ch.qos.logback.core.rolling.RollingFileAppender">
        <filter class="ch.qos.logback.classic.filter.LevelFilter">
            <level>DEBUG</level>
            <onMatch>ACCEPT</onMatch>
```

```
            <onMismatch>DENY</onMismatch>
        </filter>
        <file>./logback.log</file>
        <rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>./logback.log.%d{yyyy-MM-dd}.zip</fileNamePattern>
            <!-- 最大保存时间 -->
            <maxHistory>2</maxHistory>
        </rollingPolicy>
        <encoder>
            <pattern>${logPattern}</pattern>
        </encoder>
    </appender>

    <!-- DB  -->
    <appender name="DB" class="ch.qos.logback.classic.db.DBAppender">
        <connectionSource
class="ch.qos.logback.core.db.DriverManagerConnectionSource">
            <driverClass>com.mysql.jdbc.Driver</driverClass>
            <url>jdbc:mysql://172.17.0.203:3306/log?useSSL=false</url>
            <user>root</user>
            <password>root</password>
        </connectionSource>
    </appender>

    <!-- ASYNC_LOG  -->
    <appender name="ASYNC_LOG" class="ch.qos.logback.classic.AsyncAppender">
        <!-- 不丢失日志.默认的,如果队列的80%已满,则会丢弃TRACT、DEBUG、INFO级别的日志 -
->
        <discardingThreshold>0</discardingThreshold>
        <!-- 更改默认的队列的深度,该值会影响性能.默认值为256 -->
        <queueSize>3</queueSize>
        <appender-ref ref="STDOUT"/>
    </appender>


    <!-- 日志的记录级别 -->
    <!-- 在定义后引用APPENDER -->
    <root level="DEBUG">
        <!--  控制台  -->
        <appender-ref ref="STDOUT"/>
        <!--  ROLLING_FILE  -->
        <appender-ref ref="ROLLING_FILE"/>
        <!-- ASYNC_LOG -->
        <appender-ref ref="ASYNC_LOG"/>
    </root>

</configuration>
```

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
        <!--<charset>UTF-8</charset>-->
        <!--<pattern>${logPattern}</pattern>-->

        <!--<layout class="com.itheima.logback.MySampleLayout" />-->

        <layout class="ch.qos.logback.classic.html.HTMLLayout">
            <pattern>%relative%thread%mdc%level%logger%msg</pattern>
        </layout>

        <!--<layout class="ch.qos.logback.classic.log4j.XMLLayout">-->
            <!--<locationInfo>false</locationInfo>-->
        <!--</layout>-->
    </encoder>
</appender>
```

1.3.5 jcl

1）配置文件：

- 首先在classpath下寻找commons-logging.properties文件。如果找到，则使用其中定义的Log实现类；如果找不到，则在查找是否已定义系统环境变量org.apache.commons.logging.Log，找到则使用其定义的Log实现类；
- 查看classpath中是否有Log4j的包，如果发现，则自动使用Log4j作为日志实现类；
- 否则，使用JDK自身的日志实现类（JDK1.4以后才有日志实现类）；
- 否则，使用commons-logging自己提供的一个简单的日志实现类SimpleLog；

2）级别：

- jcl有5个级别：trace < debug < info < warn < error

3）代码实战：

- 日志查找顺序： log4j-jul-slog
- commons-logging.properties配置

```
#指定日志对象:
#org.apache.commons.logging.Log=org.apache.commons.logging.impl.SimpleLog
#org.apache.commons.logging.Log=org.apache.commons.logging.impl.Jdk14Logger
org.apache.commons.logging.Log=org.apache.commons.logging.impl.Log4JLogger

#指定日志工厂:
org.apache.commons.logging.LogFactory =
org.apache.commons.logging.impl.LogFactoryImpl
```

1.3.6 log4j2

1）配置文件：

- Log4j will inspect the `log4j.configurationFile` system property and, if set, will attempt to load the configuration using the `ConfigurationFactory` that matches the file extension.
- If no system property is set the YAML ConfigurationFactory will look for `log4j2-test.yaml` or `log4j2-test.yml` in the classpath.
- If no such file is found the JSON ConfigurationFactory will look for `log4j2-test.json` or `log4j2-test.jsn` in the classpath.
- If no such file is found the XML ConfigurationFactory will look for `log4j2-test.xml` in the classpath.
- If a test file cannot be located the YAML ConfigurationFactory will look for `log4j2.yaml` or `log4j2.yml` on the classpath.
- If a YAML file cannot be located the JSON ConfigurationFactory will look for `log4j2.json` or `log4j2.jsn` on the classpath.
- If a JSON file cannot be located the XML ConfigurationFactory will try to locate `log4j2.xml` on the classpath.
- If no configuration file could be located the `DefaultConfiguration` will be used. This will cause logging output to go to the console.

2）级别：

- 从低到高为：ALL < TRACE < DEBUG < INFO < WARN < ERROR < FATAL < OFF

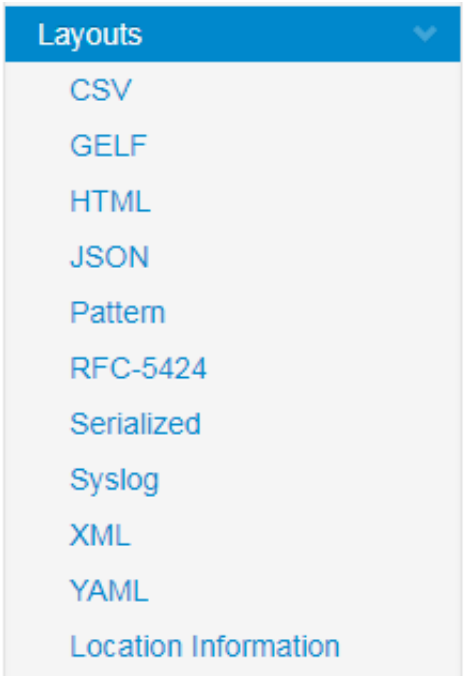
3）处理器：

http://logging.apache.org/log4j/2.x/manual/appenders.html

- FileAppender　　普通地输出到本地文件
- KafkaAppender 输出到kafka队列
- FlumeAppender　　将几个不同源的日志汇集、集中到一处。
- JMSQueueAppender，JMSTopicAppender　　与JMS相关的日志输出
- RewriteAppender　　对日志事件进行掩码或注入信息
- RollingFileAppender　　对日志文件进行封存（详细）
- RoutingAppender　　在输出地之间进行筛选路由
- SMTPAppender　　将LogEvent发送到指定邮件列表
- SocketAppender　　将LogEvent以普通格式发送到远程主机
- SyslogAppender　　将LogEvent以RFC 5424格式发送到远程主机
- AsynchAppender　　将一个LogEvent异步地写入多个不同输出地
- ConsoleAppender　　将LogEvent输出到命令行
- FailoverAppender　　维护一个队列，系统将尝试向队列中的Appender依次输出LogEvent，直到有一个成功为止

4）格式化：

http://logging.apache.org/log4j/2.x/manual/layouts.html



5）代码实战：

```
<?xml version="1.0" encoding="UTF-8"?>
<!--日志级别以及优先级排序: OFF > FATAL > ERROR > WARN > INFO > DEBUG > TRACE > ALL -->
<!--status="WARN" :用于设置log4j2自身内部日志的信息输出级别，默认是OFF-->
<!--monitorInterval="30"  :间隔秒数,自动检测配置文件的变更和重新配置本身-->
<configuration status="info" monitorInterval="30">
    <Properties>
        <!--自定义一些常量，之后使用${变量名}引用-->
        <Property name="pattern">log4j2:[%-5p]:%d{YYYY-MM-dd HH:mm:ss} [%t] %c{1}:%L - %msg%n</Property>
    </Properties>
    <!--appenders:定义输出内容,输出格式,输出方式,日志保存策略等,常用其下三种标签[console,File,RollingFile]-->
    <appenders>
        <!--console :控制台输出的配置-->
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="${pattern}"/>
        </Console>
        <!--File :同步输出日志到本地文件-->
        <!--append="false" :根据其下日志策略,每次清空文件重新输入日志,可用于测试-->
        <File name="File" fileName="./log4j2-file.log" append="false">
            <PatternLayout pattern="${pattern}"/>
        </File>
        <RollingFile name="RollingFile" fileName="./log4j2-rollingfile.log"
                     filePattern="./$${date:yyyy-MM}/log4j2-%d{yyyy-MM-dd}-%i.log">
            <!--ThresholdFilter :日志输出过滤-->
```

```
            <!--level="info" :日志级别,onMatch="ACCEPT" :级别在info之上则接
受,onMismatch="DENY" :级别在info之下则拒绝-->
            <ThresholdFilter level="debug" onMatch="ACCEPT" onMismatch="DENY"/>
            <PatternLayout pattern="${pattern}"/>
            <!-- Policies :日志滚动策略-->
            <Policies>
                <!-- TimeBasedTriggeringPolicy :时间滚动策略,
                默认0点产生新的文件,
                interval="6" : 自定义文件滚动时间间隔,每隔6小时产生新文件,
                modulate="true" : 产生文件是否以0点偏移时间,即6点,12点,18点,0点-->
                <TimeBasedTriggeringPolicy interval="6" modulate="true"/>
                <!-- SizeBasedTriggeringPolicy :文件大小滚动策略-->
                <SizeBasedTriggeringPolicy size="1 MB"/>
            </Policies>
            <!-- DefaultRolloverStrategy属性如不设置，则默认为最多同一文件夹下7个文
件，这里设置了20 -->
            <DefaultRolloverStrategy max="20"/>
        </RollingFile>

    </appenders>
    <!--然后定义logger，只有定义了logger并引入的appender，appender才会生效-->
    <loggers>
        <!--过滤掉spring和mybatis的一些无用的DEBUG信息-->
        <!--Logger节点用来单独指定日志的形式，name为包路径,比如要为org.springframework
包下所有日志指定为INFO级别等。 -->
        <logger name="org.springframework" level="INFO"></logger>
        <logger name="org.mybatis" level="INFO"></logger>

        <!--AsyncLogger :异步日志,LOG4J有三种日志模式,全异步日志,混合模式,同步日志,性能
从高到底,线程越多效率越高,也可以避免日志卡死线程情况发生-->
        <!--additivity="false" : additivity设置事件是否在root logger输出，为了避免重
复输出，可以在Logger 标签下设置additivity为”false”-->
        <AsyncLogger name="AsyncLogger" level="trace" includeLocation="true"
additivity="true">
            <appender-ref ref="Console"/>
        </AsyncLogger>

        <logger name="Kafka" additivity="false" level="debug">
            <appender-ref ref="Kafka"/>
            <appender-ref ref="Console"/>
        </logger>

        <!-- Root节点用来指定项目的根日志，如果没有单独指定Logger，那么就会默认使用该Root
日志输出 -->
        <root level="info">
            <appender-ref ref="Console"/>
            <!--<appender-ref ref="File"/>-->
            <!--<appender-ref ref="RollingFile"/>-->
            <!--<appender-ref ref="Kafka"/>-->
```

```
        </root>
    </loggers>
</configuration>
```

1.3.7 slf4j

1）配置文件：

具体日志输出内容取决于生效的具体日志实现

2）级别：

slf4j日志级别有五种：ERROR、WARN、INFO、DEBUG、TRACE，级别从高到底

3）slf日志实现：

```
<!--slf转其他日志-->
<!--slf - jcl-->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-jcl</artifactId>
    <version>1.7.30</version>
</dependency>

<!-- slf - log4j -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.30</version>
</dependency>

<!-- slf4j - log4j2 -->
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-slf4j-impl</artifactId>
    <version>2.13.0</version>
</dependency>

<!--slf - jul-->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-jdk14</artifactId>
    <version>1.7.30</version>
</dependency>

<!--slf - simplelog-->
<dependency>
```

```
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.7.30</version>
</dependency>

<!--slf - logback-->
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-core</artifactId>
    <version>1.2.3</version>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.2.3</version>
</dependency>
```

问题：多个桥接器的话，slf4j怎么处理呢?

使用在class path中较早出现的那个，如在maven中，会使用在pom.xml中定义较靠前的桥接器（代码验证）

小知识：

桥接器会传递依赖到对应的下游日志组件，比如slf4j-log4j12会附带log4j的jar包依赖（代码验证）

4）其他日志转slf

```
<!--其他日志转slf-->
<!--jul - slf-->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>jul-to-slf4j</artifactId>
    <version>1.7.30</version>
</dependency>

<!--jcl - slf-->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>jcl-over-slf4j</artifactId>
    <version>1.7.30</version>
</dependency>

<!--log4j - slf-->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>log4j-over-slf4j</artifactId>
```

```
    <version>1.7.30</version>
</dependency>

<!--log4j2 - slf-->
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-to-slf4j</artifactId>
    <version>2.13.0</version>
</dependency>
```

5）slf4j日志环

```
    <!-- 演示实例：slf4j - log4j - slf4j -->
  <!--log4j-->
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>apache-log4j-extras</artifactId>
        <version>1.2.17</version>
    </dependency>

  <!--slf4j-->
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.7.30</version>
    </dependency>

    <!--log4j - slf-->
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>log4j-over-slf4j</artifactId>
        <version>1.7.30</version>
    </dependency>

    <!-- slf - log4j -->
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.7.30</version>
        <exclusions>
            <exclusion>
                <groupId>*</groupId>
```

```
                <artifactId>*</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
```

依赖展示：

报错结果：

```
SLF4J: Detected both log4j-over-slf4j.jar AND bound slf4j-log4j12.jar on the
class path, preempting StackOverflowError.
SLF4J: See also http://www.slf4j.org/codes.html#log4jDelegationLoop for more
details.
java.lang.ExceptionInInitializerError
  at org.slf4j.impl.StaticLoggerBinder.<init>(StaticLoggerBinder.java:72)
  at org.slf4j.impl.StaticLoggerBinder.<clinit>(StaticLoggerBinder.java:45)
  at org.slf4j.LoggerFactory.bind(LoggerFactory.java:150)
  at org.slf4j.LoggerFactory.performInitialization(LoggerFactory.java:124)
  at org.slf4j.LoggerFactory.getILoggerFactory(LoggerFactory.java:417)
  at org.slf4j.LoggerFactory.getLogger(LoggerFactory.java:362)
  at org.slf4j.LoggerFactory.getLogger(LoggerFactory.java:388)
  at com.itheima.slf4j.Demo.<clinit>(Demo.java:8)
Caused by: java.lang.IllegalStateException: Detected both log4j-over-slf4j.jar
AND bound slf4j-log4j12.jar on the class path, preempting StackOverflowError.
See also http://www.slf4j.org/codes.html#log4jDelegationLoop for more details.
  at org.slf4j.impl.Log4jLoggerFactory.<clinit>(Log4jLoggerFactory.java:54)
  ... 8 more
Exception in thread "main"
Process finished with exit code 1
```

经验：使用slf桥接和绑定的时候要留心，不要自相矛盾形成环

1.4日志建议

1.4.1 门面约束

使用门面，而不是具体实现

使用 Log Facade 可以方便的切换具体的日志实现。而且，如果依赖多个项目，使用了不同的Log Facade，还可以方便的通过 Adapter 转接到同一个实现上。如果依赖项目直接使用了多个不同的日志实现，会非常糟糕。

经验之谈：日志门面，一般现在推荐使用 Log4j-API 或者 SLF4j，不推荐继续使用 JCL。

1.4.2 单一原则

只添加一个日志实现

项目中应该只使用一个具体的 Log Implementation，如果在依赖的项目中，使用的 Log Facade不支持当前 Log Implementation，就添加合适的桥接器。

经验之谈：jul性能一般，log4j性能也有问题而且不再维护，建议使用 Logback 或者Log4j2。

1.4.3 依赖约束

日志实现坐标应该设置为optional并使用runtime scope

在项目中，Log Implementation的依赖强烈建议设置为runtime scope，并且设置为optional。例如项目中使用了 SLF4J 作为 Log Facade，然后想使用 Log4j2 作为 Implementation，那么使用 maven 添加依赖的时候这样设置:

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>${log4j.version}</version>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-slf4j-impl</artifactId>
    <version>${log4j.version}</version>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
```

设为optional，依赖不会传递，这样如果你是个lib项目，然后别的项目使用了你这个lib，不会被引入不想要的Log Implementation 依赖；

Scope设置为runtime，是为了防止开发人员在项目中直接使用Log Implementation中的类，强制约束开发人员使用Facade接口。

1.4.4 避免传递

尽量用exclusion排除依赖的第三方库中的日志坐标

同上一个话题，第三方库的开发者却未必会把具体的日志实现或者桥接器的依赖设置为optional，然后你的项目就会被迫传递引入这些依赖，而这些日志实现未必是你想要的，比如他依赖了Log4j，你想使用Logback，这时就很尴尬。另外，如果不同的第三方依赖使用了不同的桥接器和Log实现，极有可能会形成环。

这种情况下，推荐的处理方法，是使用exclude来排除所有的这些Log实现和桥接器的依赖，只保留第三方库里面对Log Facade的依赖。

实例：依赖jstorm会引入Logback和log4j-over-slf4j，如果你在自己的项目中使用Log4j或其他Log实现的话，就需要加上exclusion:

```
<dependency>
    <groupId>com.alibaba.jstorm</groupId>
    <artifactId>jstorm-core</artifactId>
    <version>2.1.1</version>
    <exclusions>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>log4j-over-slf4j</artifactId>
        </exclusion>
        <exclusion>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

1.4.5 注意写法

避免为不会输出的log买单

Log库都可以灵活的设置输出级别，所以每一条程序中的log，都是有可能不会被输出的。这时候要注意不要额外的付出代价。实例如下：

```
logger.debug("this is debug: " + message);
logger.debug("this is json msg: {}", toJson(message));
```

前面讲到，第一条的字符串拼接，即使日志级别高于debug不会打印，依然会做字符串连接操作；第二条虽然用了SLF4J/Log4j2 中的懒求值方式，但是toJson()这个函数却是总会被调用并且开销更大。推荐的写法如下:

```
// SLF4J/LOG4J2
logger.debug("this is debug:{}", message);

// LOG4J2
logger.debug("this is json msg: {}", () -> toJson(message));

// SLF4J/LOG4J2
if (logger.isDebugEnabled()) {
    logger.debug("this is debug: " + message);
}
```

1.4.6 减少分析

输出的日志中尽量不要使用行号，函数名等信息

原因是，为了获取语句所在的函数名，或者行号，log库的实现都是获取当前的stacktrace，然后分析取出这些信息，而获取stacktrace的代价是很昂贵的。如果有很多的日志输出，就会占用大量的CPU。在没有特殊需要的情况下，建议不要在日志中输出这些这些字段。

1.4.7 精简至上

log中尽量不要输出稀奇古怪的字符，这是个习惯和约束问题。有的同学习惯用这种语句:

```
logger.debug("=====================================: {}",message);
```

输出了大量无关字符，虽然自己一时痛快，但是如果所有人都这样做的话，那log输出就没法看了！正确的做法是日志只输出必要信息，如果要过滤，后期使用grep来筛选，只查自己关心的日志。