# log4j2 实际使用详解

## 一、目录简介

基础部分
日志框架简单比较（slf4j、log4j、logback、log4j2 ）
log4j2 基础示例
log4j2 配置文件
实战部分
slf4j + log4j2 实际使用
重要的就是需要使用 Maven 代理，因为国外真的是太慢了！不然 pom.xml 里面直接填写 jar 自动下载可能够呛！
下面的只是参考，有点混乱，重在理解。

## 二、日志框架比较（slf4j、log4j、logback、log4j2 ）

日志接口(slf4j)
slf4j 是对所有日志框架制定的一种规范、标准、接口，并不是一个框架的具体的实现，因为接口并不能独立使用，需要和具体的日志框架实现配合使用（如 log4j、logback）

日志实现(log4j、logback、log4j2)

log4j 是 apache 实现的一个开源日志组件
logback 同样是由 log4j 的作者设计完成的，拥有更好的特性，用来取代 log4j 的一个日志框架，是 slf4j 的原生实现
Log4j2 是 log4j 1.x 和 logback 的改进版，据说采用了一些新技术（无锁异步、等等），使得日志的吞吐量、性能比 log4j 1.x 提高 10 倍，并解决了一些死锁的 bug，而且配置更加简单灵活，官网地址： http://logging.apache.org/log4j/2.x/manual/configuration.html
为什么需要日志接口，直接使用具体的实现不就行了吗？

接口用于定制规范，可以有多个实现，使用时是面向接口的（导入的包都是 slf4j 的包而不是具体某个日志框架中的包），即直接和接口交互，不直接使用实现，所以可以任意的更换实现而不用更改代码中的日志相关代码。

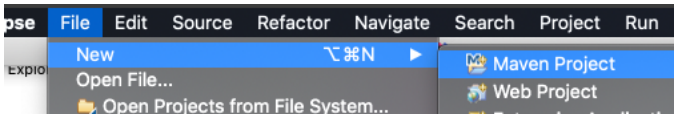
比如：slf4j 定义了一套日志接口，项目中使用的日志框架是 logback，开发中调用的所有接口都是 slf4j 的，不直接使用 logback，调用是 自己的工程调用 slf4j 的接口，slf4j 的接口去调用 logback 的实现，可以看到整个过程应用程序并没有直接使用 logback，当项目需要更换更加优秀的日志框架时（如 log4j2）只需要引入 Log4j2 的 jar 和 Log4j2 对应的配置文件即可，完全不用更改 Java 代码中的日志相关的代码 logger.info( “xxx” )，也不用修改日志相关的类的导入的包（import org.slf4j.Logger;
import org.slf4j.LoggerFactory;）

使用日志接口便于更换为其他日志框架。

log4j、logback、log4j2 都是一种日志具体实现框架，所以既可以单独使用也可以结合 slf4j 一起搭配使用）

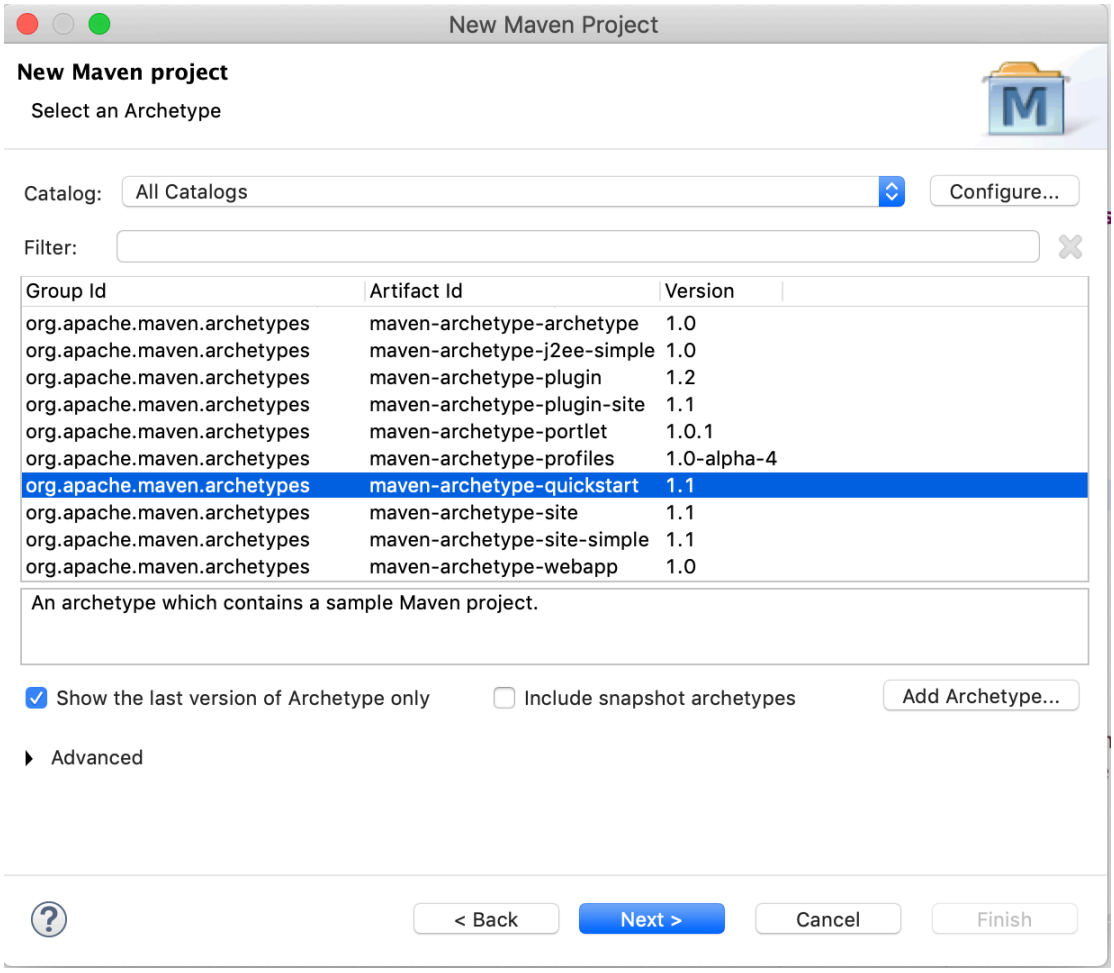
## 三、log4j2 基础示例

创建 maven ( web )项目，

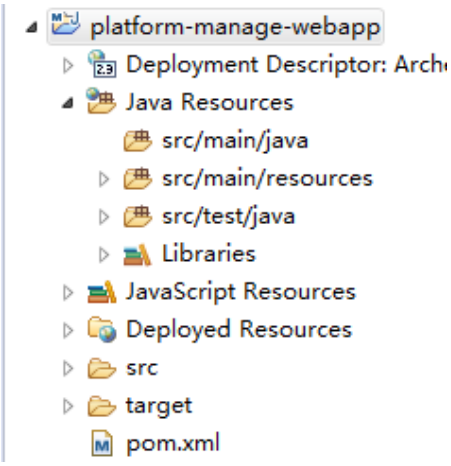



直接 Next ，

我自己测试就没有选择 webapp 而是直接默认的 quickstart 项目



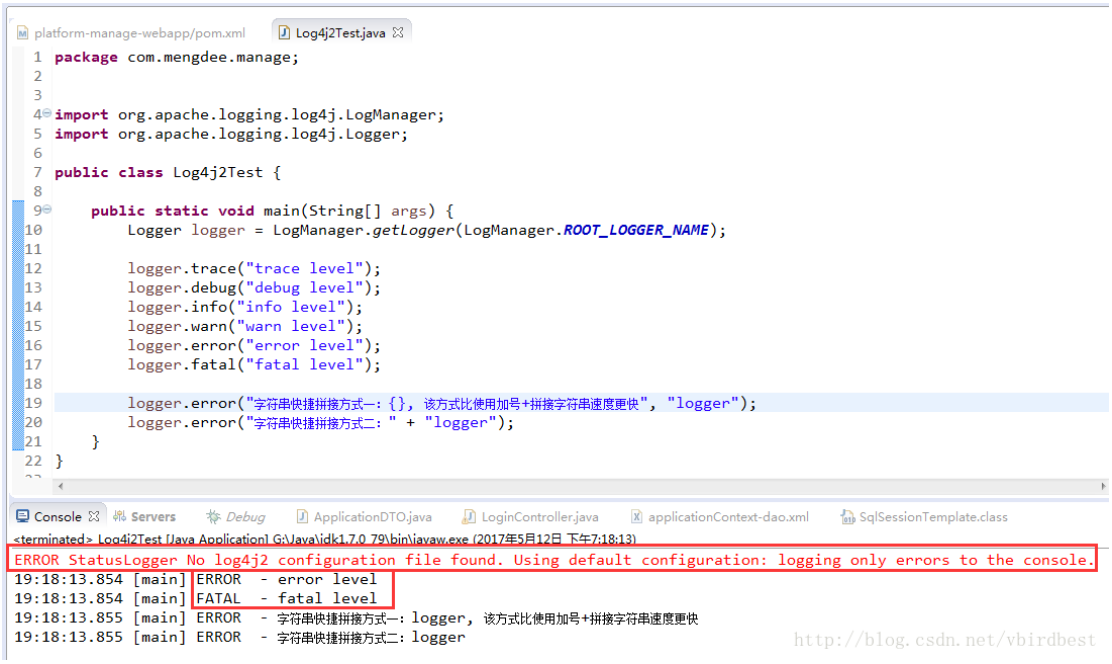

结构如下




配置 pom.xml,引入 log4j2 必要的依赖(log4j-api、log4j-core)

```
<properties>
      <junit.version>3.8.1</junit.version>
      <log4j.version>2.5</log4j.version>
</properties>

<!-- 使用 aliyun 镜像 -->
<repositories>
   <repository>
         <id>aliyun</id>
         <name>aliyun</name>
         <url>http://maven.aliyun.com/nexus/content/groups/public</url>
   </repository>
</repositories>

<dependencies>
      <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-api</artifactId>
            <version>${log4j.version}</version>
      </dependency>
      <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-core</artifactId>
            <version>${log4j.version}</version>
      </dependency>
</dependencies>
```

3、 使用 Main 方法简单测试




测试说明：

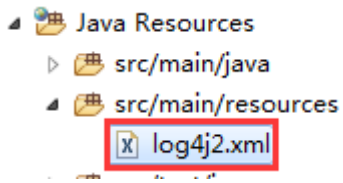
工程中只引入的 jar 并没有引入任何配置文件，在测试的时候可以看到有 ERROR 输出："ERROR StatusLogger No log4j2 configuration file found. Using default configuration: logging only errors to the console."

输出 logger 时可以看到只有 error 和 fatal 级别的被输出来，是因为没有配置文件就使用默认的，默认级别是 error，所以只有 error 和 fatal 输出来

引入的包是 log4j 本身的包（import org.apache.logging.log4j.LogManager）

## 四：log2j 配置文件详解

配置文件的格式和位置
配置文件的格式：log2j 配置文件可以是 xml 格式的，也可以是 json 格式的，
配置文件的位置：log4j2 默认会在 classpath 目录下寻找 log4j2.xml、log4j.json、log4j.jsn 等名称的文件，如果都没有找到，则会按默认配置输出，也就是输出到控制台，也可以对配置文件自定义位置（需要在 web.xml 中配置），一般放置在 src/main/resources 根目录下即可




纯 Java 方式：

```
public static void main(String[] args) throws IOException {
    File file = new File("D:/log4j2.xml");
    BufferedInputStream in = new BufferedInputStream(new FileInputStream(file));
    final ConfigurationSource source = new ConfigurationSource(in);
    Configurator.initialize(null, source);

    Logger logger = LogManager.getLogger("myLogger");
}
```

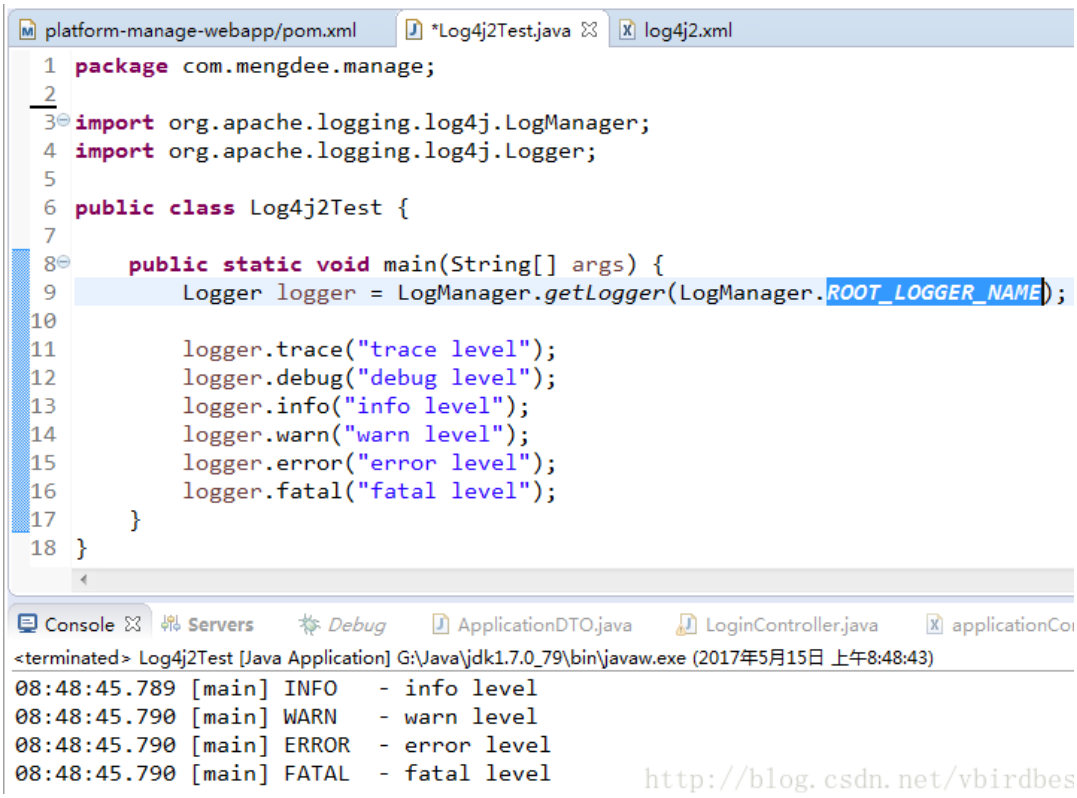
Web 工程方式：

```
<context-param>
    <param-name>log4jConfiguration</param-name>
    <param-value>/WEB-INF/conf/log4j2.xml</param-value>
</context-param>

<listener>
    <listener-class>org.apache.logging.log4j.web.Log4jServletContextListener</listener-class>
</listener>
```

示例一：简单配置（使用根控制器输出到控制台上）
log4j2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n" />
        </Console>
    </Appenders>

    <Loggers>
        <Root level="info">
            <AppenderRef ref="Console" />
        </Root>
    </Loggers>
</Configuration>
```

示例结果：




结果解释：
日志管理器获取的是根日志器 LogManager.getLogger(LogManager.ROOT_LOGGER_NAME); 对应的 log4j2.xml 中的 Loggers 节点下的 Root，因为该根日志器的 level= “info” ，所以输出的 info 级别以上的日志

## 示例二：File Logger

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n" />
        </Console>

        <File name="FileAppender" fileName="D:/logs/app.log">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n" />
        </File>

        <!-- 发现 Async 好像 PatternLayout 的输出格式配置的和输出的格式不一样，不
```

```
用异步就完全一样  -->
            <Async name="AsyncAppender">
            <AppenderRef ref="FileAppender"/>
            </Async>
        </Appenders>

        <Loggers>
            <Logger name="AsyncFileLogger" level="trace" additivity="true">
                <AppenderRef ref="AsyncAppender" />
            </Logger>
            <Root level="info">
                <AppenderRef ref="Console" />
            </Root>
        </Loggers>
</Configuration>



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

public class Log4j2Test {

        public static void main(String[] args) {
            Logger logger = LogManager.getLogger("AsyncFileLogger"); // Logger 的名称

            logger.trace("trace level");
            logger.debug("debug level");
            logger.info("info level");
            logger.warn("warn level");
            logger.error("error level");
            logger.fatal("fatal level");
        }
}
```

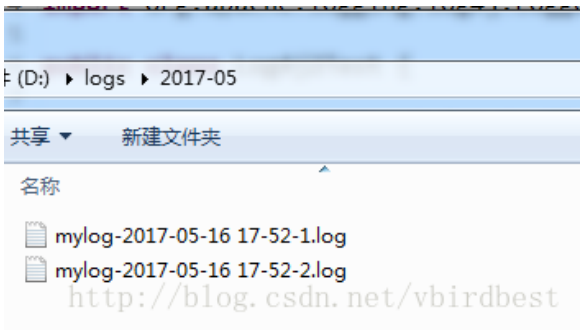
AsyncFileLogger 的 additivity 的值如果为 false 的话, 就不会在控制台上输出或者为该 Logger 再增加一个输出源 Consloe

```
<Logger name="AsyncFileLogger" level="trace" additivity="false">
        <AppenderRef ref="AsyncAppender" />
        <AppenderRef ref="Console" />
</Logger>
```

## 示例三： RollingRandomAccessFile

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <properties>
        <property name="LOG_HOME">D:/logs</property>
        <property name="FILE_NAME">mylog</property>
    </properties>


    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n" />
        </Console>

        <RollingRandomAccessFile name="RollingRandomAccessFile" fileName="${LOG_HOME}/${FILE_NAME}.log" filePattern="${LOG_HOME}/$${date:yyyy-MM}/${FILE_NAME}-%d{yyyy-MM-dd HH-mm}-%i.log">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
            <Policies>
                <TimeBasedTriggeringPolicy interval="1"/>
                <SizeBasedTriggeringPolicy size="10 MB"/>
            </Policies>
            <DefaultRolloverStrategy max="20"/>
        </RollingRandomAccessFile>

        <Async name="AsyncAppender">
            <AppenderRef ref="RollingRandomAccessFile"/>
        </Async>
    </Appenders>

    <Loggers>
        <Logger name="RollingRandomAccessFileLogger" level="info" additivity="false">
            <AppenderRef ref="AsyncAppender" />
            <AppenderRef ref="Console" />
        </Logger>
    </Loggers>
</Configuration>
```

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

public class Log4j2Test {

    public static void main(String[] args) {

        Logger logger = LogManager.getLogger("RollingRandomAccessFileLogger");
        for(int i = 0; i < 50000; i++) {
            logger.trace("trace level");
            logger.debug("debug level");
            logger.info("info level");
            logger.warn("warn level");
            logger.error("error level");
            logger.fatal("fatal level");
        }

        try {
            Thread.sleep(1000 * 61);
        } catch (InterruptedException e) {}

        logger.trace("trace level");
        logger.debug("debug level");
        logger.info("info level");
        logger.warn("warn level");
        logger.error("error level");
        logger.fatal("fatal level");
    }
}
```




RollingRandomAccessFile 会根据命名规则当文件满足一定大小时就会另起一个新的文件

# 五：log4j2 配置文件详解

log4j2.xml 文件的配置大致如下：

Configuration
properties
Appenders
Console
PatternLayout
File
RollingRandomAccessFile
Async
Loggers
Logger
Root
AppenderRef
Configuration：为根节点，有 status 和 monitorInterval 等多个属性

status 的值有 “trace”，“debug”，“info”，“warn”，“error” and “fatal”，用于控制 log4j2 日志框架本身的日志级别, 如果将 stratus 设置为较低的级别就会看到很多关于 log4j2 本身的日志，如加载 log4j2 配置文件的路径等信息
monitorInterval，含义是每隔多少秒重新读取配置文件，可以不重启应用的情况下修改配置
Appenders：输出源，用于定义日志输出的地方
log4j2 支持的输出源有很多，有控制台 Console、文件 File、RollingRandomAccessFile、MongoDB、Flume 等

Console：控制台输出源是将日志打印到控制台上，开发的时候一般都会配置，以便调试

File：文件输出源，用于将日志写入到指定的文件，需要配置输入到哪个位置（例如：D:/logs/mylog.log）

RollingRandomAccessFile: 该输出源也是写入到文件，不同的是比 File 更加强大，可以指定当文件达到一定大小（如 20MB）时，另起一个文件继续写入日志，另起一个文件就涉及到新文件的名字命名规则，因此需要配置文件命名规则
这种方式更加实用，因为你不可能一直往一个文件中写，如果一直写，文件过大，打开就会卡死，也不便于查找日志。

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
NoSql：MongoDb，输出到 MongDb 数据库中

Flume：输出到 Apache Flume（Flume 是 Cloudera 提供的一个高可用的，高可靠的，分布式的海量日志采集、聚合和传输的系统，Flume 支持在日志系统中定制各类数据发送方，用于收集数据；同时，Flume 提供对数据进行简单处理，并写到各种数据接受方（可定制）的能力。）

Async：异步，需要通过 AppenderRef 来指定要对哪种输出源进行异步（一般用于配置 RollingRandomAccessFile）

PatternLayout：控制台或文件输出源（Console、File、RollingRandomAccessFile）都必须包含一个 PatternLayout 节点，用于指定输出文件的格式（如 日志输出的时间 文件 方法 行数 等格式），例如 pattern=" %d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"

%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 Log4j2Test.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

## Loggers：日志器

日志器分根日志器 Root 和自定义日志器，当根据日志名字获取不到指定的日志器时就使用 Root 作为默认的日志器，自定义时需要指定每个 Logger 的名称 name（对于命名可以以包名作为日志的名字，不同的包配置不同的级别等），日志级别 level，相加性 additivity（是否继承下面配置的日志器），对于一般的日志器（如 Console、File、RollingRandomAccessFile）一般需要配置一个或多个输出源 AppenderRef；

每个 logger 可以指定一个 level（TRACE, DEBUG, INFO, WARN, ERROR, ALL or OFF），不指定时 level 默认为 ERROR

additivity 指定是否同时输出 log 到父类的 appender，缺省为 true。

```
<Logger name="rollingRandomAccessFileLogger" level="trace" additivity="true">
    <AppenderRef ref="RollingRandomAccessFile" />
</Logger>
```

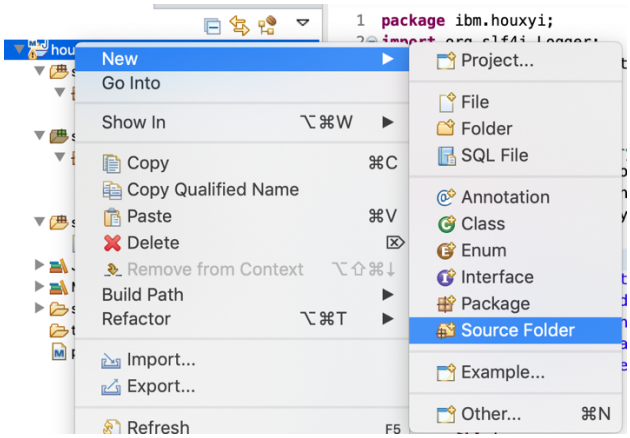
properties: 属性
使用来定义常量，以便在其他配置的时候引用，该配置是可选的，例如定义日志的存放位置 D:/logs
【实战部分】
引入 slf4j 和 log4j 需要的依赖

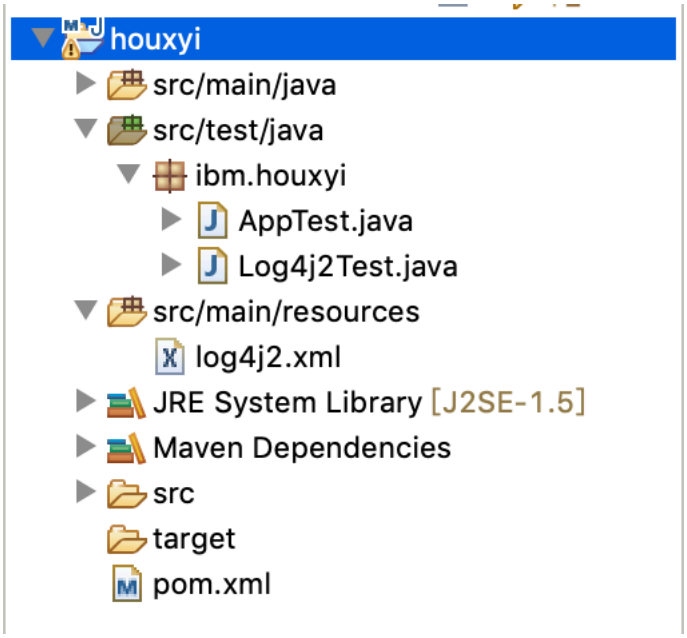
```
<properties>
    <junit.version>3.8.1</junit.version>
    <log4j.version>2.5</log4j.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junit.version}</version>
      <scope>test</scope>
    </dependency>

    <!-- slf4j + log4j2 begin -->
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.7.10</version>
    </dependency>

    <dependency> <!-- 桥接：告诉 Slf4j 使用 Log4j2 -->
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-slf4j-impl</artifactId>
        <version>2.2</version>
    </dependency>
    <dependency> <!-- 桥接：告诉 commons logging 使用 Log4j2 -->
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-jcl</artifactId>
        <version>2.2</version>
    </dependency>

    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
```

```
        <artifactId>log4j-api</artifactId>
        <version>${log4j.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>${log4j.version}</version>
    </dependency>
    <!-- log4j end-->
</dependencies>

<!-- 使用 aliyun 镜像 -->
<repositories>
    <repository>
        <id>aliyun</id>
        <name>aliyun</name>
        <url>http://maven.aliyun.com/nexus/content/groups/public</url>
    </repository>
</repositories>
```

## 2、配置 log4j2.xml

新建一个 resourceFolder 文件夹






```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <properties>
        <property name="LOG_HOME">.</property>
        <property name="FILE_NAME">mylog</property>
        <property name="log.sql.level">info</property>
    </properties>


    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %l - %msg%n" />
        </Console>

        <RollingRandomAccessFile name="RollingRandomAccessFile"
fileName="${LOG_HOME}/${FILE_NAME}.log" filePattern="${LOG_HOME}/$${date:yyyy-
MM}/${FILE_NAME}-%d{yyyy-MM-dd HH-mm}-%i.log">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %l - %msg%n"/>
            <Policies>
                <TimeBasedTriggeringPolicy interval="1"/>
                <SizeBasedTriggeringPolicy size="10 MB"/>
            </Policies>
            <DefaultRolloverStrategy max="20"/>
        </RollingRandomAccessFile>
    </Appenders>

    <Loggers>
```

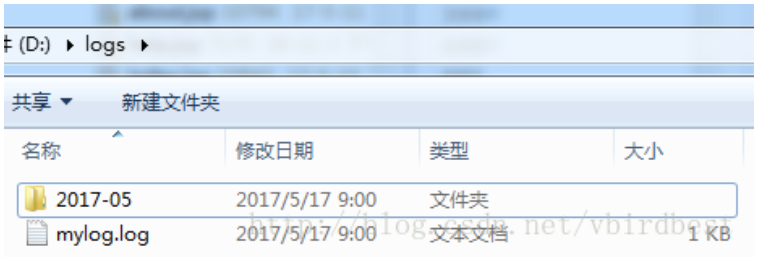
```
        <Root level="info">
            <AppenderRef ref="Console" />
            <AppenderRef ref="RollingRandomAccessFile" />
        </Root>
        <!—注意下面这个包名 -->
        <Logger name="com.mengdee.dao" level="${log.sql.level}" additivity="false">
             <AppenderRef ref="Console" />
        </Logger>
    </Loggers>
</Configuration>
```

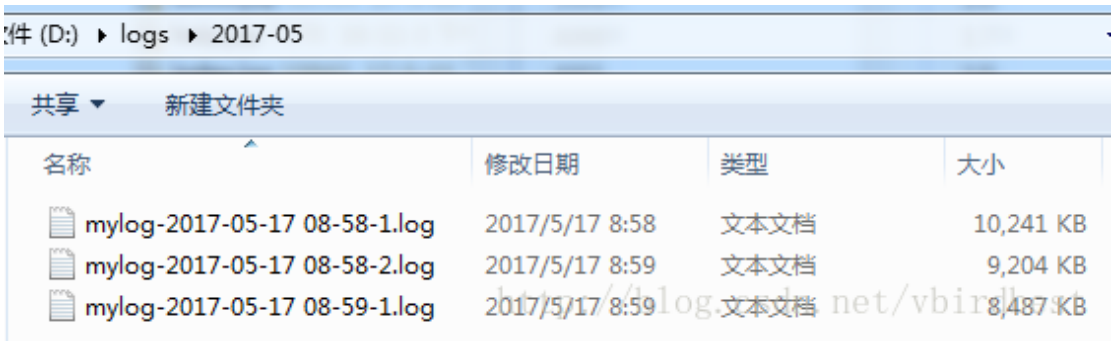
## 3、 Java

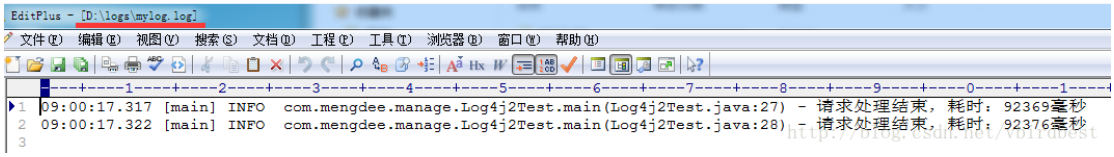
```
package com.mengdee.manage;


import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Log4j2Test {

    // Logger 和 LoggerFactory 导入的是 org.slf4j 包
    private final static Logger logger = LoggerFactory.getLogger(Log4j2Test.class);
    public static void main(String[] args) {
        long beginTime = System.currentTimeMillis();

        for(int i = 0; i < 10000; i++) {
            logger.trace("trace level");
            logger.debug("debug level");
            logger.info("info level");
            logger.warn("warn level");
            logger.error("error level");
        }

        try {
            Thread.sleep(1000 * 61);
        } catch (InterruptedException e) {}


        logger.info("请求处理结束, 耗时 :{}毫秒", (System.currentTimeMillis() - beginTime));
//第一种用法
        logger.info("请求处理结束，耗时：" + (System.currentTimeMillis() - beginTime)  +
"毫秒");      //第二种用法
```

```
    }
}
```

## 4、运行结果



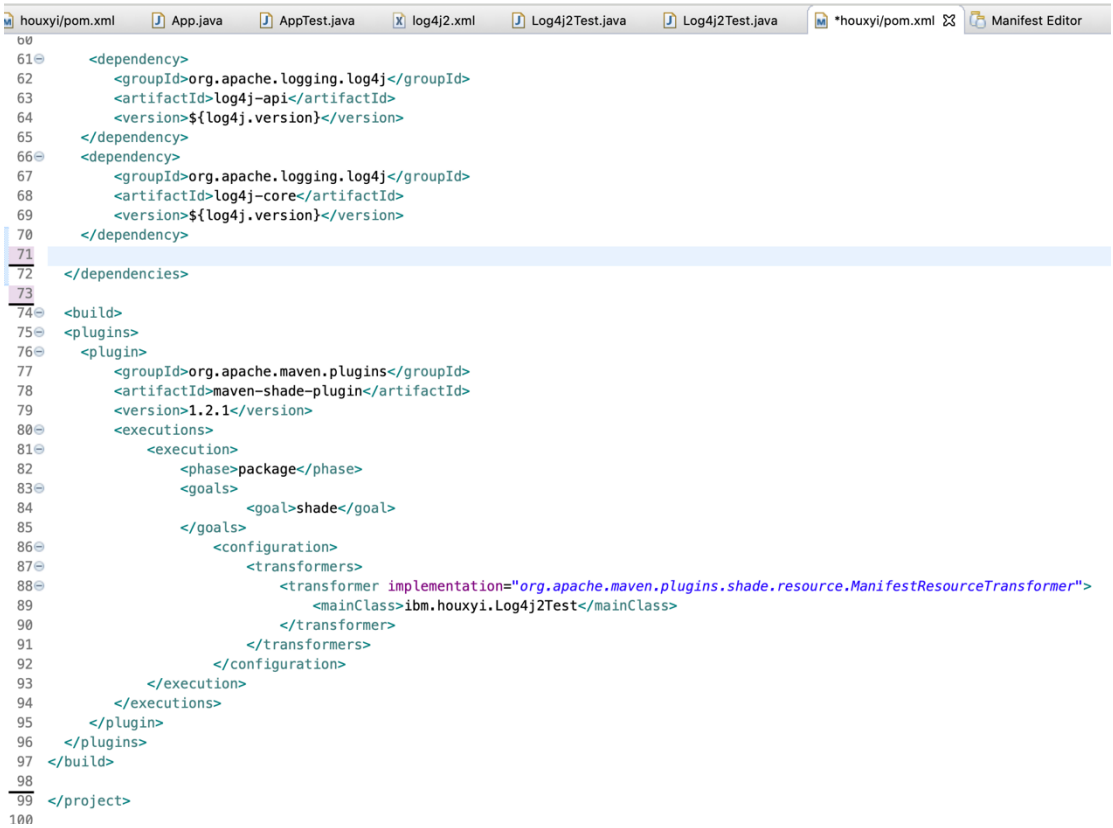



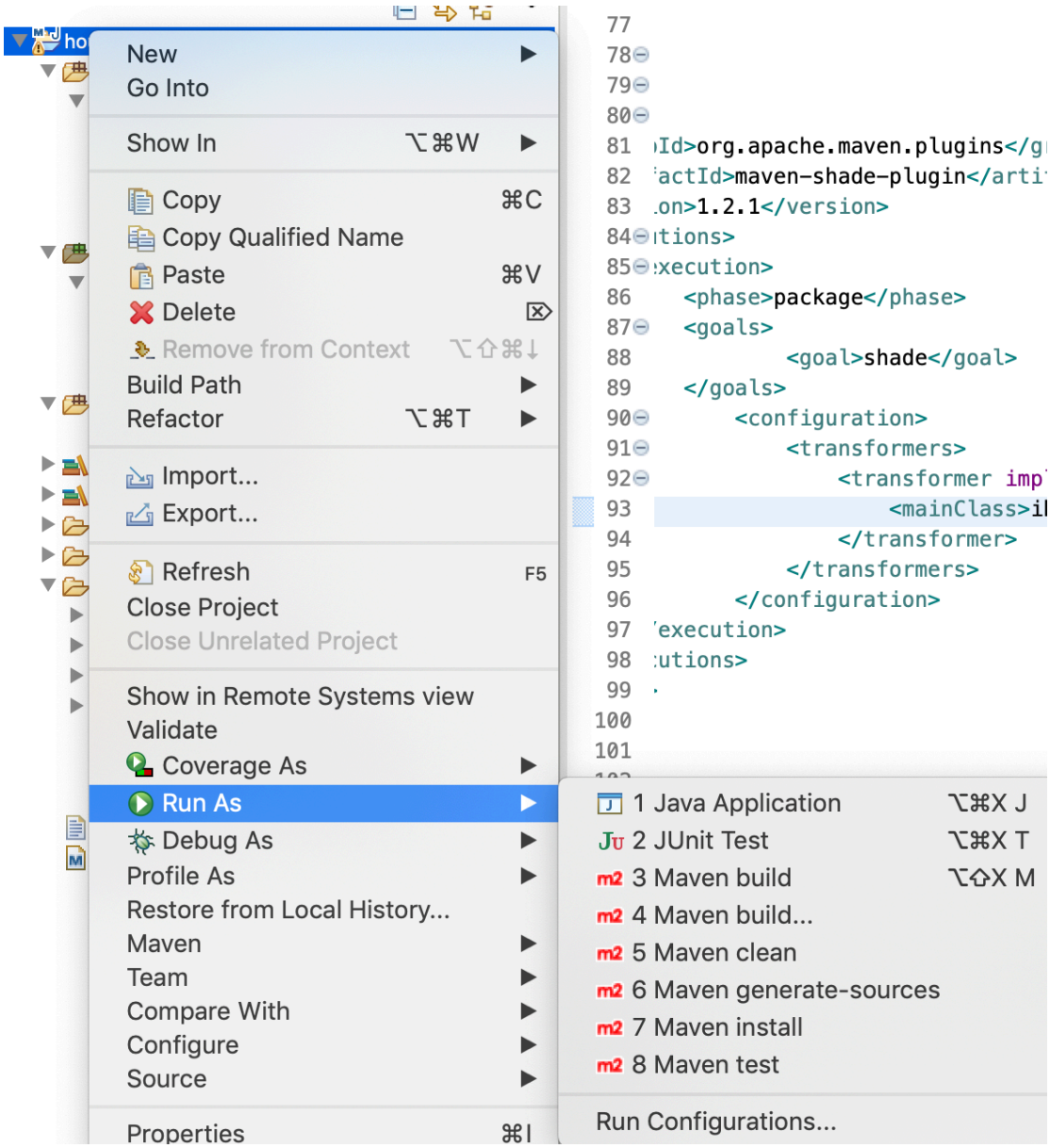

Maven 项目打包：
打包之前需要在 pom.xml 配置下主清单属性，否则 jar 运行的时候找不到程序入口：
参考：https://www.cnblogs.com/snaildev/p/8317896.html

```
<build>
<plugins>
  <plugin>
     <groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-shade-plugin</artifactId>
     <version>1.2.1</version>
     <executions>
        <execution>
           <phase>package</phase>
           <goals>
                 <goal>shade</goal>
```

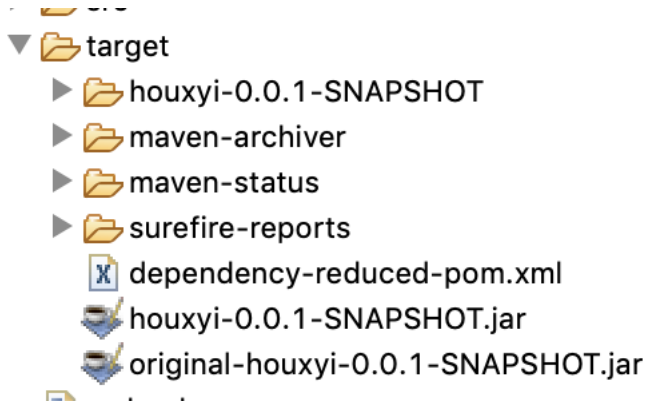
```
            </goals>
                <configuration>
                    <transformers>
                        <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTran
sformer">
                            <mainClass>ibm.houxyi.Log4j2Test</mainClass>
                        </transformer>
                    </transformers>
                </configuration>
            </execution>
        </executions>
    </plugin>
  </plugins>
</build>
```




项目右击菜单》Run as 》 Maven install




打包完成之后生成了一个 jar 包会在 target/下面



之后运行 jar 包：

```
wxydembp:target wxy$ java -jar houxyi-0.0.1-SNAPSHOT.jar
17:00:15.011 [main] INFO  ibm.houxyi.Log4j2Test.main(Log4j2Test.java:15) - info level
17:00:15.012 [main] WARN  ibm.houxyi.Log4j2Test.main(Log4j2Test.java:16) - warn level
17:00:15.012 [main] ERROR ibm.houxyi.Log4j2Test.main(Log4j2Test.java:17) - error level
17:00:15.013 [main] INFO  ibm.houxyi.Log4j2Test.main(Log4j2Test.java:15) - info level
17:00:15.013 [main] WARN  ibm.houxyi.Log4j2Test.main(Log4j2Test.java:16) - warn level
17:00:15.013 [main] ERROR ibm.houxyi.Log4j2Test.main(Log4j2Test.java:17) - error level
17:00:15.013 [main] INFO  ibm.houxyi.Log4j2Test.main(Log4j2Test.java:15) - info level
17:00:15.013 [main] WARN  ibm.houxyi.Log4j2Test.main(Log4j2Test.java:16) - warn level
17:00:15.014 [main] ERROR ibm.houxyi.Log4j2Test.main(Log4j2Test.java:17) - error level
17:00:15.014 [main] INFO  ibm.houxyi.Log4j2Test.main(Log4j2Test.java:15) - info level
17:00:15.014 [main] WARN  ibm.houxyi.Log4j2Test.main(Log4j2Test.java:16) - warn level
17:00:15.014 [main] ERROR ibm.houxyi.Log4j2Test.main(Log4j2Test.java:17) - error level
17:00:15.015 [main] INFO  ibm.houxyi.Log4j2Test.main(Log4j2Test.java:15) - info level
17:00:15.015 [main] WARN  ibm.houxyi.Log4j2Test.main(Log4j2Test.java:16) - warn level
17:00:15.015 [main] ERROR ibm.houxyi.Log4j2Test.main(Log4j2Test.java:17) - error level
17:00:15.016 [main] INFO  ibm.houxyi.Log4j2Test.main(Log4j2Test.java:15) - info level
17:00:15.016 [main] WARN  ibm.houxyi.Log4j2Test.main(Log4j2Test.java:16) - warn level
17:00:15.016 [main] ERROR ibm.houxyi.Log4j2Test.main(Log4j2Test.java:17) - error level
17:00:15.016 [main] INFO  ibm.houxyi.Log4j2Test.main(Log4j2Test.java:15) - info level
17:00:15.016 [main] WARN  ibm.houxyi.Log4j2Test.main(Log4j2Test.java:16) - warn level
```

参考：https://blog.csdn.net/vbirdbest/article/details/71751835