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

#### 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;

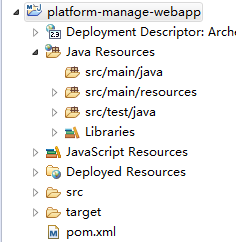
import org.slf4j.LoggerFactory;）

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, 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

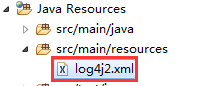
引入的包是log4j本身的包（import org.apache.logging.log4j.LogManager）

#### 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">

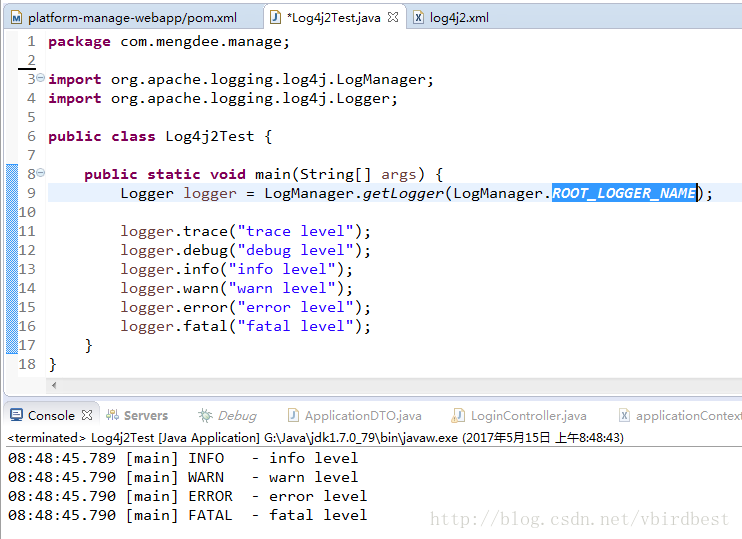
<AppenderRef ref="Console" />

</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");

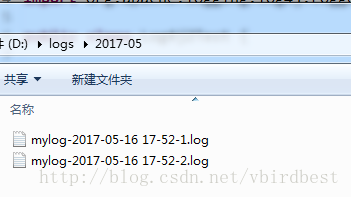
logger.warn("warn level");

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log2.xml

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

<Configuration status="WARN">

<properties>

<property name="LOG\_HOME">D:/logs</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>

<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 < 100000; 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) {}

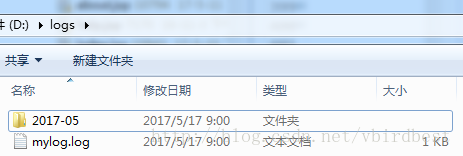
logger.info("请求处理结束，耗时：{}毫秒", (System.currentTimeMillis() - beginTime)); //第一种用法

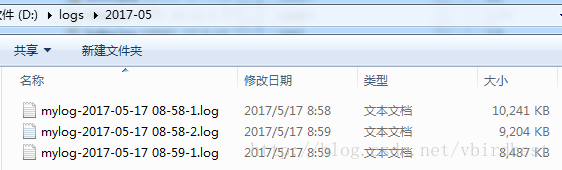
logger.info("请求处理结束，耗时：" + (System.currentTimeMillis() - beginTime) + "毫秒"); //第二种用法

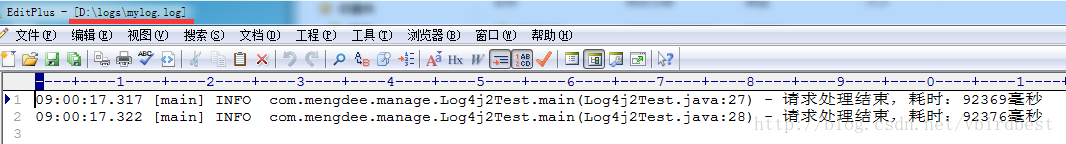
}

}

##### 4、运行结果







持续完善中。。。

---------------------

作者：vbirdbest

来源：CSDN

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

版权声明：本文为博主原创文章，转载请附上博文链接！