原文链接：https://blog.csdn.net/white\_ice/article/details/85065219

**springboot使用logback日志框架超详细教程**

1. 前言

项目中日志系统是必不可少的，目前**比较流行的日志框架有log4j、logback等**，可能大家还不知道，这两个框架的作者是同一个人，Logback旨在作为流行的log4j项目的后续版本，从而恢复log4j离开的位置。另外 **slf4j(Simple Logging Facade for Java) 则是一个日志门面框架，提供了日志系统中常用的接口，logback 和 log4j 则对slf4j 进行了实现。**我们本文将讲述如何在spring boot 中应用 logback+slf4j实现日志的记录。

1. 为什么使用logback

Logback 是log4j 框架的作者开发的新一代日志框架，它效率更高、能够适应诸多的运行环境，同时天然支持SLF4J。

Logback的定制性更加灵活，同时也是spring boot的内置日志框架。

1. 开始使用
2. 添加依赖：maven依赖中添加了spring-boot-starter-logging

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-logging</artifactId>

</dependency>

但是呢，**实际开发中我们不需要直接添加该依赖**，你会发现spring-boot-starter其中包含了 spring-boot-starter-logging，该依赖内容就是 Spring Boot 默认的日志框架 Logback+SLF4J。而 spring-boot-starter-web 包含了spring-boot-starte，所以**我们只需要引入web组件即可**：

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-web</artifactId>**

**</dependency>**

1. 修改全局配置文件

**默认情况下Spring Boot将日志输出到控制台，不会写到日志文件。**如果要编写除控制台输出之外的日志文件，则**需在application.properties中设置logging.file或logging.path属性**。

注：二者不能同时使用，如若同时使用，则只有logging.file生效

logging.file=文件名

logging.path=日志文件路径

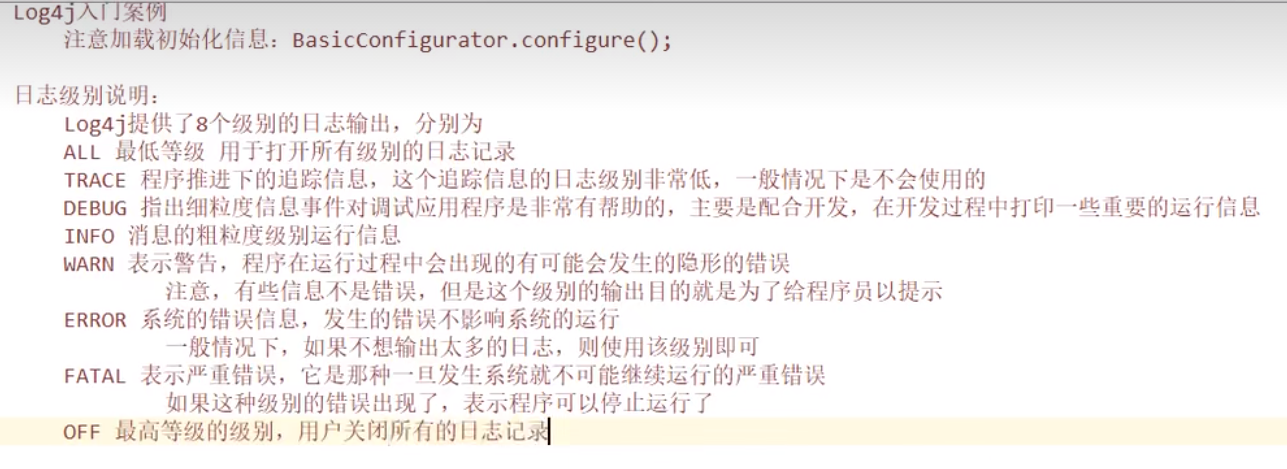
logging.level.包名=指定包下的日志级别

logging.pattern.console=日志打印规则

logging.file，设置文件，可以是绝对路径，也可以是相对路径。如：logging.file=my.log

logging.path，设置目录，会在该目录下创建spring.log文件，并写入日志内容，如：logging.path=/var/log

注：二者不能同时使用，如若同时使用，则只有logging.file生效，可以看到这种方式配置简单，但是能实现的功能也非常有限，如果想要更复杂的需求，就需要下面的定制化配置了。



1. logback-spring.xml详解

**Spring Boot官方推荐优先使用带有-spring的文件名作为你的日志配置**（如使用logback-spring.xml，而不是logback.xml），命名为logback-spring.xml的日志配置文件，将xml放至 src/main/resource下面。

也**可以使用自定义的名称，比如logback-config.xml，只需要在application.properties文件中使用logging.config=classpath:logback-config.xml指定即可。**

Logback基于三个主要类：Logger，Appender和Layout（记录器、附加器、布局）。 这三种类型的组件协同工作，使开发人员能够根据消息类型和级别记录消息，并在运行时控制这些消息的格式以及报告的位置。首先给出一个基本的xml配置如下：

<configuration>

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<!-- encoders are assigned the type

ch.qos.logback.classic.encoder.PatternLayoutEncoder by default

默认情况下，编码器被分配了  
ch.qos.logback.classic.encoder.PatternLayoutEncoder类型 -->

<encoder>

<pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>

</encoder>

</appender>

<logger name="chapters.configuration" level="INFO"/>

<!-- Strictly speaking, the level attribute is not necessary since the level of the root level is set to DEBUG by default.

严格来说，级别属性是不必要的，因为根级别的级别默认设置为DEBUG。-->

<root level="DEBUG">

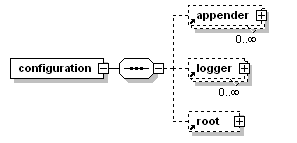
<appender-ref ref="STDOUT" />

</root>

</configuration>

1. <configuration>元素

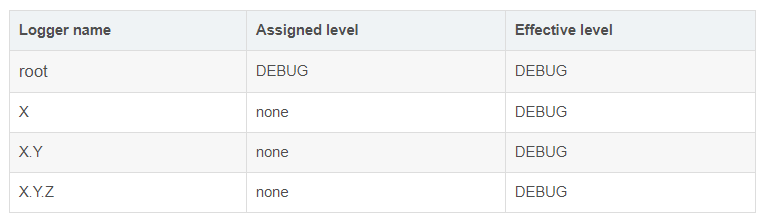
logback.xml配置文件的基本结构可以描述为<configuration>元素，包含零个或多个<appender>元素，后跟零个或多个<logger>元素，后跟最多一个<root>元素(也可以没有)。下图说明了这种基本结构：



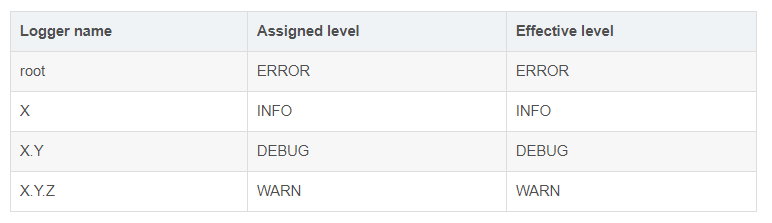
1. <logger>元素

<logger>元素只接受一个必需的name属性，一个可选的level属性和一个可选的additivity属性，允许值为true或false。 level属性的值允许一个不区分大小写的字符串值TRACE，DEBUG，INFO，WARN，ERROR，ALL或OFF。特殊于大小写不敏感的值INHERITED或其同义词NULL将强制记录器的级别从层次结构中的较高级别继承，<logger>元素可以包含零个或多个<appender-ref>元素; 这样引用的每个appender都被添加到指定的logger中，(注：additivity属性下面详说)，logger元素级别具有继承性。

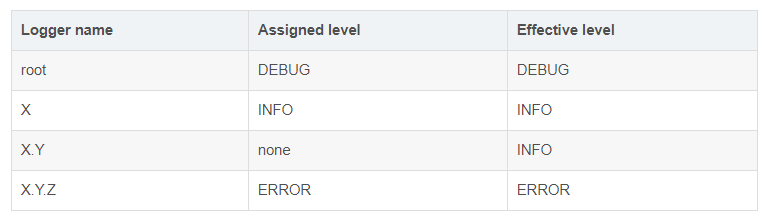
例1：示例中，仅为根记录器分配了级别。 此级别值DEBUG由其他记录器X，X.Y和X.Y.Z继承



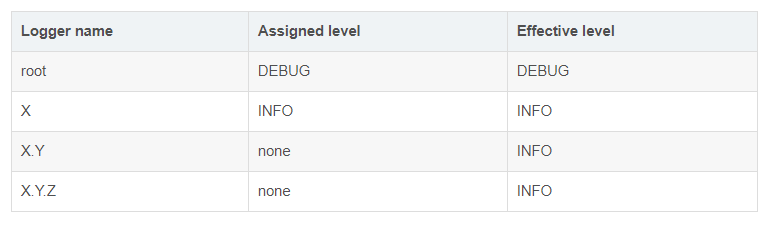
例2：所有记录器都有一个指定的级别值。 级别继承不起作用



例3：记录器root，X和X.Y.Z分别被分配了DEBUG，INFO和ERROR级别。 Logger X.Y从其父X继承其级别值。



例4：在示例4中，记录器root和X分别被分配了DEBUG和INFO级别。 记录器X.Y和X.Y.Z从其最近的父X继承其级别值，该父级具有指定的级别。

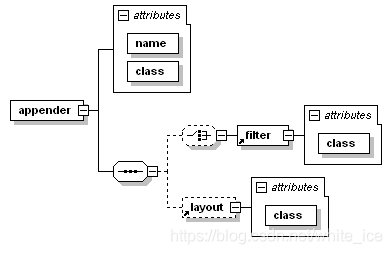


1. <root>元素

**<root>元素配置根记录器。 它支持单个属性，即level属性。 它不允许任何其他属性，**因为additivity标志不适用于根记录器。 此外，由于根记录器已被命名为“ROOT”，因此它也不允许使用name属性。 level属性的值可以是不区分大小写的字符串TRACE，DEBUG，INFO，WARN，ERROR，ALL或OFF之一。**<root>元素可以包含零个或多个<appender-ref>元素; 这样引用的每个appender都被添加到根记录器中**(注：additivity属性下面详说)。

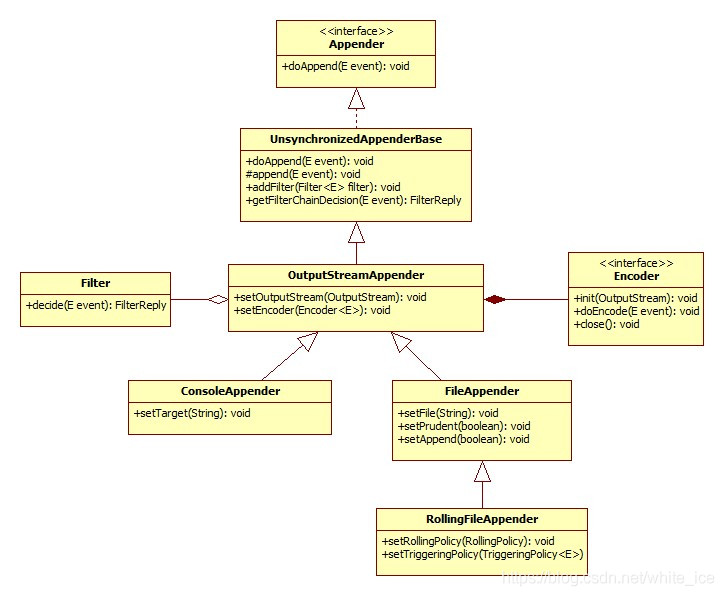
1. <appender>元素

appender使用<appender>元素配置，该元素采用两个必需属性name和class。 name属性指定appender的名称，而class属性指定要实例化的appender类的完全限定名称。 <appender>元素可以包含零个或一个<layout>元素，零个或多个<encoder>元素以及零个或多个<filter>元素，下图说明了常见的结构：



注意：在logback中，输出目标称为appender，addAppender方法将appender添加到给定的记录器logger。给定记录器的每个启用的日志记录请求都将转发到该记录器中的所有appender以及层次结构中较高的appender。换句话说，appender是从记录器层次结构中附加地继承的。例如，如果将控制台appender添加到根记录器，则所有启用的日志记录请求将至少在控制台上打印。如果另外将文件追加器添加到记录器（例如L），则对L和L的子项启用的记录请求将打印在文件和控制台上。通过将记录器的additivity标志设置为false，可以覆盖此默认行为，以便不再添加appender累积。

Appender是一个接口，它有许多子接口和实现类，具体如下图所示：



其中最重要的两个Appender为：ConsoleAppender 、RollingFileAppender。

1. ConsoleAppender

ConsoleAppender，如名称所示，将日志输出到控制台上。

1. RollingFileAppender

RollingFileAppender，是FileAppender的一个子类，扩展了FileAppender，具有翻转日志文件的功能。 例如，RollingFileAppender 可以记录到名为log.txt文件的文件，并且一旦满足某个条件，就将其日志记录目标更改为另一个文件。

有两个与RollingFileAppender交互的重要子组件。 第一个RollingFileAppender子组件，即 RollingPolicy 负责执行翻转所需的操作。 RollingFileAppender的第二个子组件，即 TriggeringPolicy 将确定是否以及何时发生翻转。 因此，RollingPolicy 负责什么和TriggeringPolicy 负责什么时候。

作为任何用途，RollingFileAppender 必须同时设置 RollingPolicy 和 TriggeringPolicy。 但是，如果其 RollingPolicy 也实现了TriggeringPolicy 接口，则只需要显式指定前者。

1. 滚动策略

TimeBasedRollingPolicy：可能是最受欢迎的滚动策略。 它根据时间定义翻转策略，例如按天或按月。 TimeBasedRollingPolicy承担滚动和触发所述翻转的责任。 实际上，TimeBasedTriggeringPolicy实现了RollingPolicy和TriggeringPolicy接口。

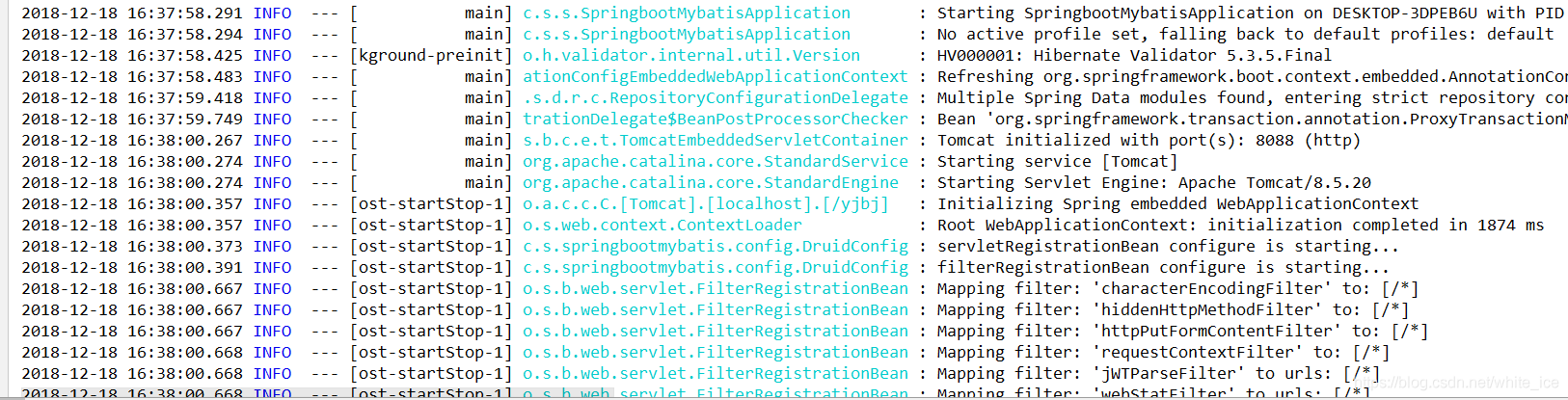
SizeAndTimeBasedRollingPolicy：有时您可能希望按日期归档文件，但同时限制每个日志文件的大小，特别是如果后处理工具对日志文件施加大小限制。 为了满足此要求，logback 提供了 SizeAndTimeBasedRollingPolicy ，它是TimeBasedRollingPolicy的一个子类，实现了基于时间和日志文件大小的翻滚策略。

1. <encoder>元素

encoder中最重要就是pattern属性，它负责控制输出日志的格式，这里给出一个我自己写的示例：

<pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %highlight(%-5level) --- [%15.15(%thread)] %cyan(%-40.40(%logger{40})) : %msg%n</pattern>

使用后的输出格式如下图所示



其中：%d{yyyy-MM-dd HH:mm:ss.SSS}：日期

%-5level：日志级别

%highlight()：颜色，info为蓝色，warn为浅红，error为加粗红，debug为黑色

%thread：打印日志的线程

%15.15():如果记录的线程字符长度小于15(第一个)则用空格在左侧补齐,如果字符长度大于15(第二个),则从开头开始截断多余的字符

%logger：日志输出的类名

%-40.40()：如果记录的logger字符长度小于40(第一个)则用空格在右侧补齐,如果字符长度大于40(第二个),则从开头开始截断多余的字符

%cyan：颜色

%msg：日志输出内容

%n：换行符

1. <filter>元素

filter中最重要的两个过滤器为：LevelFilter、ThresholdFilter。

LevelFilter 根据精确的级别匹配过滤事件。 如果事件的级别等于配置的级别，则筛选器接受或拒绝该事件，具体取决于onMatch和onMismatch属性的配置。 例如下面配置将只打印INFO级别的日志，其余的全部禁止打印输出：

<configuration>

<appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">

<filter class="ch.qos.logback.classic.filter.LevelFilter">

<level>INFO</level>

<onMatch>ACCEPT</onMatch>

<onMismatch>DENY</onMismatch>

</filter>

<encoder>

<pattern>

%-4relative [%thread] %-5level %logger{30} - %msg%n

</pattern>

</encoder>

</appender>

<root level="DEBUG">

<appender-ref ref="CONSOLE" />

</root>

</configuration>

ThresholdFilter 过滤低于指定阈值的事件。 对于等于或高于阈值的事件，ThresholdFilter将在调用其decision（）方法时响应NEUTRAL。 但是，将拒绝级别低于阈值的事件，例如下面的配置将拒绝所有低于INFO级别的日志，只输出INFO以及以上级别的日志：

<configuration>

<appender name="CONSOLE"

class="ch.qos.logback.core.ConsoleAppender">

<!-- deny all events with a level below INFO, that is TRACE and DEBUG -->

<filter class="ch.qos.logback.classic.filter.ThresholdFilter">

<level>INFO</level>

</filter>

<encoder>

<pattern>

%-4relative [%thread] %-5level %logger{30} - %msg%n

</pattern>

</encoder>

</appender>

<root level="DEBUG">

<appender-ref ref="CONSOLE" />

</root>

</configuration>

1. 详细的logback-spring.xml示例：

以上介绍了xml中重要的几个元素，下面将我配置的xml贴出来以供参考（实现了基于日期和大小翻滚的策略，以及经INFO和ERROR日志区分输出，还有规范日志输出格式等）：

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

<configuration debug="true">

<!-- appender是configuration的子节点，是负责写日志的组件。 -->

<!-- ConsoleAppender：把日志输出到控制台 -->

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<!-- 默认情况下，每个日志事件都会立即刷新到基础输出流。 这种默认方法更安全，因为如果应用程序在没有正确关闭appender的情况下退出，则日志事件不会丢失。

但是，为了显着增加日志记录吞吐量，您可能希望将immediateFlush属性设置为false -->

<!--<immediateFlush>true</immediateFlush>-->

<encoder>

<!-- %37():如果字符没有37个字符长度,则左侧用空格补齐 -->

<!-- %-37():如果字符没有37个字符长度,则右侧用空格补齐 -->

<!-- %15.15():如果记录的线程字符长度小于15(第一个)则用空格在左侧补齐,如果字符长度大于15(第二个),则从开头开始截断多余的字符 -->

<!-- %-40.40():如果记录的logger字符长度小于40(第一个)则用空格在右侧补齐,如果字符长度大于40(第二个),则从开头开始截断多余的字符 -->

<!-- %msg：日志打印详情 -->

<!-- %n:换行符 -->

<!-- %highlight():转换说明符以粗体红色显示其级别为ERROR的事件，红色为WARN，BLUE为INFO，以及其他级别的默认颜色。 -->

<pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %highlight(%-5level) --- [%15.15(%thread)] %cyan(%-40.40(%logger{40})) : %msg%n</pattern>

<!-- 控制台也要使用UTF-8，不要使用GBK，否则会中文乱码 -->

<charset>UTF-8</charset>

</encoder>

</appender>

<!-- info 日志-->

<!-- RollingFileAppender：滚动记录文件，先将日志记录到指定文件，当符合某个条件时，将日志记录到其他文件 -->

<!-- 以下的大概意思是：1.先按日期存日志，日期变了，将前一天的日志文件名重命名为XXX%日期%索引，新的日志仍然是project\_info.log -->

<!-- 2.如果日期没有发生变化，但是当前日志的文件大小超过10MB时，对当前日志进行分割 重命名-->

<appender name="info\_log" class="ch.qos.logback.core.rolling.RollingFileAppender">

<!--日志文件路径和名称-->

<File>logs/project\_info.log</File>

<!--是否追加到文件末尾,默认为true-->

<append>true</append>

<filter class="ch.qos.logback.classic.filter.LevelFilter">

<level>ERROR</level>

<onMatch>DENY</onMatch><!-- 如果命中ERROR就禁止这条日志 -->

<onMismatch>ACCEPT</onMismatch><!-- 如果没有命中就使用这条规则 -->

</filter>

<!--有两个与RollingFileAppender交互的重要子组件。 第一个RollingFileAppender子组件，即RollingPolicy:负责执行翻转所需的操作。

RollingFileAppender的第二个子组件，即TriggeringPolicy:将确定是否以及何时发生翻转。 因此，RollingPolicy负责什么和TriggeringPolicy负责什么时候.

作为任何用途，RollingFileAppender必须同时设置RollingPolicy和TriggeringPolicy,但是，如果其RollingPolicy也实现了TriggeringPolicy接口，则只需要显式指定前者。-->

<rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">

<!-- 日志文件的名字会根据fileNamePattern的值，每隔一段时间改变一次 -->

<!-- 文件名：logs/project\_info.2017-12-05.0.log -->

<!-- 注意：SizeAndTimeBasedRollingPolicy中 ％i和％d令牌都是强制性的，必须存在，要不会报错 -->

<fileNamePattern>logs/project\_info.%d.%i.log</fileNamePattern>

<!-- 每产生一个日志文件，该日志文件的保存期限为30天, ps:maxHistory的单位是根据fileNamePattern中的翻转策略自动推算出来的,例如上面选用了yyyy-MM-dd,则单位为天

如果上面选用了yyyy-MM,则单位为月,另外上面的单位默认为yyyy-MM-dd-->

<maxHistory>30</maxHistory>

<!-- 每个日志文件到10mb的时候开始切分，最多保留30天，但最大到20GB，哪怕没到30天也要删除多余的日志 -->

<totalSizeCap>20GB</totalSizeCap>

<!-- maxFileSize:这是活动文件的大小，默认值是10MB，测试时可改成5KB看效果 -->

<maxFileSize>10MB</maxFileSize>

</rollingPolicy>

<!--编码器-->

<encoder>

<!-- pattern节点，用来设置日志的输入格式 ps:日志文件中没有设置颜色,否则颜色部分会有ESC[0:39em等乱码-->

<pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %-5level --- [%15.15(%thread)] %-40.40(%logger{40}) : %msg%n</pattern>

<!-- 记录日志的编码:此处设置字符集 - -->

<charset>UTF-8</charset>

</encoder>

</appender>

<!-- error 日志-->

<!-- RollingFileAppender：滚动记录文件，先将日志记录到指定文件，当符合某个条件时，将日志记录到其他文件 -->

<!-- 以下的大概意思是：1.先按日期存日志，日期变了，将前一天的日志文件名重命名为XXX%日期%索引，新的日志仍然是project\_error.log -->

<!-- 2.如果日期没有发生变化，但是当前日志的文件大小超过10MB时，对当前日志进行分割 重命名-->

<appender name="error\_log" class="ch.qos.logback.core.rolling.RollingFileAppender">

<!--日志文件路径和名称-->

<File>logs/project\_error.log</File>

<!--是否追加到文件末尾,默认为true-->

<append>true</append>

<!-- ThresholdFilter过滤低于指定阈值的事件。 对于等于或高于阈值的事件，ThresholdFilter将在调用其decision（）方法时响应NEUTRAL。 但是，将拒绝级别低于阈值的事件 -->

<filter class="ch.qos.logback.classic.filter.ThresholdFilter">

<level>ERROR</level><!-- 低于ERROR级别的日志（debug,info）将被拒绝，等于或者高于ERROR的级别将相应NEUTRAL -->

</filter>

<!--有两个与RollingFileAppender交互的重要子组件。 第一个RollingFileAppender子组件，即RollingPolicy:负责执行翻转所需的操作。

RollingFileAppender的第二个子组件，即TriggeringPolicy:将确定是否以及何时发生翻转。 因此，RollingPolicy负责什么和TriggeringPolicy负责什么时候.

作为任何用途，RollingFileAppender必须同时设置RollingPolicy和TriggeringPolicy,但是，如果其RollingPolicy也实现了TriggeringPolicy接口，则只需要显式指定前者。-->

<rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">

<!-- 活动文件的名字会根据fileNamePattern的值，每隔一段时间改变一次 -->

<!-- 文件名：logs/project\_error.2017-12-05.0.log -->

<!-- 注意：SizeAndTimeBasedRollingPolicy中 ％i和％d令牌都是强制性的，必须存在，要不会报错 -->

<fileNamePattern>logs/project\_error.%d.%i.log</fileNamePattern>

<!-- 每产生一个日志文件，该日志文件的保存期限为30天, ps:maxHistory的单位是根据fileNamePattern中的翻转策略自动推算出来的,例如上面选用了yyyy-MM-dd,则单位为天

如果上面选用了yyyy-MM,则单位为月,另外上面的单位默认为yyyy-MM-dd-->

<maxHistory>30</maxHistory>

<!-- 每个日志文件到10mb的时候开始切分，最多保留30天，但最大到20GB，哪怕没到30天也要删除多余的日志 -->

<totalSizeCap>20GB</totalSizeCap>

<!-- maxFileSize:这是活动文件的大小，默认值是10MB，测试时可改成5KB看效果 -->

<maxFileSize>10MB</maxFileSize>

</rollingPolicy>

<!--编码器-->

<encoder>

<!-- pattern节点，用来设置日志的输入格式 ps:日志文件中没有设置颜色,否则颜色部分会有ESC[0:39em等乱码-->

<pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %-5level --- [%15.15(%thread)] %-40.40(%logger{40}) : %msg%n</pattern>

<!-- 记录日志的编码:此处设置字符集 - -->

<charset>UTF-8</charset>

</encoder>

</appender>

<!--给定记录器的每个启用的日志记录请求都将转发到该记录器中的所有appender以及层次结构中较高的appender（不用在意level值）。

换句话说，appender是从记录器层次结构中附加地继承的。

例如，如果将控制台appender添加到根记录器，则所有启用的日志记录请求将至少在控制台上打印。

如果另外将文件追加器添加到记录器（例如L），则对L和L'子项启用的记录请求将打印在文件和控制台上。

通过将记录器的additivity标志设置为false，可以覆盖此默认行为，以便不再添加appender累积-->

<!-- configuration中最多允许一个root，别的logger如果没有设置级别则从父级别root继承 -->

<root level="INFO">

<appender-ref ref="STDOUT" />

</root>

<!-- 指定项目中某个包，当有日志操作行为时的日志记录级别 -->

<!-- 级别依次为【从高到低】：FATAL > ERROR > WARN > INFO > DEBUG > TRACE -->

<logger name="com.sailing.springbootmybatis" level="INFO">

<appender-ref ref="info\_log" />

<appender-ref ref="error\_log" />

</logger>

<!-- 利用logback输入mybatis的sql日志，

注意：如果不加 additivity="false" 则此logger会将输出转发到自身以及祖先的logger中，就会出现日志文件中sql重复打印-->

<logger name="com.sailing.springbootmybatis.mapper" level="DEBUG" additivity="false">

<appender-ref ref="info\_log" />

<appender-ref ref="error\_log" />

</logger>

<!-- additivity=false代表禁止默认累计的行为，即com.atomikos中的日志只会记录到日志文件中，不会输出层次级别更高的任何appender-->

<logger name="com.atomikos" level="INFO" additivity="false">

<appender-ref ref="info\_log" />

<appender-ref ref="error\_log" />

</logger>

</configuration>

1. 附加内容

这里再说下log日志输出代码，一般有人可能在代码中使用如下方式输出：

Object entry = new SomeObject();

logger.debug("The entry is " + entry);

上面看起来没什么问题，但是会存在构造消息参数的成本，即将entry转换成字符串相加。并且无论是否记录消息，都是如此，即：那怕日志级别为INFO，也会执行括号里面的操作，但是日志不会输出，下面是优化后的写法：

if(logger.isDebugEnabled()) {

Object entry = new SomeObject();

logger.debug("The entry is " + entry);

}

首先对设置的日志级别进行了判断，如果为debug模式，才进行参数的构造，对第一种写法进行了改善。不过还有最好的写法，使用占位符：

Object entry = new SomeObject();

logger.debug("The entry is {}.", entry);

只有在评估是否记录之后，并且只有在决策是肯定的情况下，记录器实现才会格式化消息并将“{}”对替换为条目的字符串值。 换句话说，当禁用日志语句时，此表单不会产生参数构造的成本。

logback作者进行测试得出：第一种和第三种写法将产生完全相同的输出。 但是，在禁用日志记录语句的情况下，第三个变体将比第一个变体优于至少30倍。

如果有多个参数，写法如下：

logger.debug("The new entry is {}. It replaces {}.", entry, oldEntry);

如果需要传递三个或更多参数，则还可以使用Object []变体：

Object[] paramArray = {newVal, below, above};

logger.debug("Value {} was inserted between {} and {}.", paramArray);

记录日志的时候我们可能需要在文件中记录下异常的堆栈信息，经过测试，logger.error(e) 不会打印出堆栈信息，正确的写法是：

logger.error("程序异常, 详细信息:{}", e.getLocalizedMessage() , e);