slf4j,logback的配置和使用

# logback的介绍

Logback是由log4j创始人设计的又一个开源日志组件。logback当前分成三个模块：logback-core,logback- classic和logback-access。logback-core是其它两个模块的基础模块。logback-classic是log4j的一个改良版本。此外logback-classic完整实现SLF4J API使你可以很方便地更换成其它日志系统如log4j或JDK14 Logging。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。 Logback是要与SLF4J结合起来用两个组件的官方网站如下：

|  |
| --- |
| logback的官方网站： http://logback.qos.ch  SLF4J的官方网站：http://www.slf4j.org |
| #maven依赖  <dependency>  <groupId>ch.qos.logback</groupId>  <artifactId>logback-core</artifactId>  <version>1.1.8</version>  </dependency>  <dependency>  <groupId>ch.qos.logback</groupId>  <artifactId>logback-access</artifactId>  <version>1.1.8</version>  </dependency>  <dependency>  <groupId>ch.qos.logback</groupId>  <artifactId>logback-classic</artifactId>  <version>1.1.8</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>1.7.21</version>  </dependency> |

# logback取代log4j的理由：

Logback和log4j是非常相似的，如果你对log4j很熟悉，那对logback很快就会得心应手。下面列了logback相对于log4j的一些优点：

* 更快的实现 ：Logback的内核重写了，在一些关键执行路径上性能提升10倍以上。而且logback不仅性能提升，初始化内存加载也更小。
* 非常充分的测试 Logback经过了几年，数不清小时的测试。Logback的测试完全不同级别的。在作者的观点，这是简单重要的原因选择logback而不是log4j。
* Logback-classic非常自然实现了SLF4j：Logback-classic实现了 SLF4j。在使用SLF4j中，你都感觉不到logback-classic。而且因为logback-classic非常自然地实现了SLF4J， 所以切换到log4j或者其他，非常容易，只需要提供成另一个jar包就可以了，根本不需要去动那些通过SLF4JAPI实现的代码。
* 非常充分的文档 官方网站有两百多页的文档。
* 自动重新加载配置文件：当配置文件修改了，Logback-classic能自动重新加载配置文件。扫描过程快且安全，它并不需要另外创建一个扫描线程。这个技术充分保证了应用程序能跑得很欢在JEE环境里面。
* Lilith: Lilith是log事件的观察者，和log4j的chainsaw类似。而lilith还能处理大数量的log数据 。
* 谨慎的模式和非常友好的恢复 在谨慎模式下，多个FileAppender实例跑在多个JVM下，能够安全地写道同一个日志文件。RollingFileAppe  
  nder会有些限制。Logback的FileAppender和它的子类包括 Rolling  
  FileAppender能够非常友好地从I/O异常中恢复。
* 配置文件可以处理不同的情况 开发人员经常需要判断不同的Logback配置文件在不同的环境下（开发，测试，生产）。而这些配置文件仅仅只有一些很小的不同，可以通过,和来实现，这样一个配置文件就可以适应多个环境。
* Filters（过滤器） 有些时候，需要诊断一个问题，需要打出日志。在log4j，只有降低日志级别，不过这样会打出大量的日志，会影响应用性能。在Logback，你可以继续保持那个日志级别而除掉某种特殊情况，如alice这个用户登录，她的日志将打在DEBUG级别而其他用户可以继续打在WARN级别。要实现这个功能只需 加4行XML配置。可以参考MDCFIlter 。
* SiftingAppender（一个非常多功能的Appender） 它可以用来分割日志文件根据任何一个给定的运行参数。如，SiftingAppender能够区别日志事件跟进用户的Session，然后每个用户会有一个日志文件。
* 自动压缩已经打出来的log：RollingFileAppender在产生新文件的时候，会自动压缩已经打出来的日志文件。压缩是个异步过程，所以甚至对于大的日志文件，在压缩过程中应用不会受任何影响。
* 堆栈树带有包版本：Logback在打出堆栈树日志时，会带上包的数据。
* 自动去除旧的日志文件：通过设置TimeBasedRollingPolicy或者SizeAndTimeBasedFNATP的maxHistory属性，你可以控制已经产生日志文件的最大数量。如果设置maxHistory 12，那那些log文件超过12个月的都会被自动移除。

# Logback的配置介绍

## Logger、appender及layout

Logger作为日志的记录器，把它关联到应用的对应的context上后，主要用于存放日志对象，也可以定义日志类型、级别。

Appender主要用于指定日志输出的目的地，目的地可以是控制台，文件，远程套接字服务器, MySQL PostreSQL Oracle和其他数据库、 JMS和远程UNIX Syslog守护进程等。

Layout 负责把事件转换成字符串，格式化的日志信息的输出。

Logger、Appenders及layouts的关系：一个 logger 可以被关联多个 appender。方法 addAppender() 为指定的 logger 添加一个appender。对于 logger 的每个启用了的记录请求，都将被发送到 logger 里的全部 appender 及更高等级的 appender。换句话说，appender叠加性地继承了 logger 的层次等级。

Logger L的记录语句的输出会发送给 L及L祖先的全部 appender。如果 logger L的某个祖先 P设置叠加性标识为 false，那么，L的输出会发送给L 与 P之间（含P）的所有 appender，但不会发送给P的任何祖先的appender。

Logger 的叠加性默认为 true。如果希望定制输出格式。这时为 appender 关联一个 layout 即可。Layout负责根据用户意愿对记录请求进行格式化，appender 负责将格式化化后的输出发送到目的地。

|  |
| --- |
| 示例：叠加性标识  <configuration>  …………  <logger name="com.ttpod.chapters.configuration.Foo"additivity="false">  <appender-refref="FILE"/>  </logger>  <rootlevel="debug">  <appender-refref="STDOUT"/>  </root>  </configuration> |

## logger context

各个logger 都被关联到一个 LoggerContext，LoggerContext负责制造logger，也负责以树结构排列各logger。其他所有logger也通过org.slf4j.LoggerFactory 类的静态方法getLogger取得。 getLogger方法以 logger名称为参数。用同一名字调用LoggerFactory.getLogger 方法所得到的永远都是同一个logger对象的引用。

如果 logger的名称带上一个点号后是另外一个 logger的名称的前缀，那么，前者就被称为后者的祖先。如果logger与其后代 logger之间没有其他祖先，那么，前者就被称为子logger 之父。比如，名为"com.foo""的 logger 是名为"com.foo.Bar"之父。root logger 位于 logger 等级的最顶端，root logger 可以通过其名称取得，如下所示：

|  |
| --- |
| Logger rootLogger =  LoggerFactory.getLogger(org.slf4j.Logger.ROOT\_LOGGER\_NAME); |

## 有效级别及级别的继承

Logger 可以被分配级别。级别包括：TRACE、DEBUG、INFO、WARN 和 ERROR，定义于ch.qos.logback.classic.Level类。如果 logger没有被分配级别，那么它将从有被分配级别的最近的祖先那里继承级别。root logger 默认级别是 DEBUG。

## 打印方法与基本的选择规则

打印方法决定记录请求的级别。例如，如果 L 是一个 logger 实例，那么，语句 L.info("..")是一条级别为 INFO的记录语句。记录请求的级别在高于或等于其 logger 的有效级别时被称为被启用，否则，称为被禁用。记录请求级别为 p，其 logger的有效级别为 q，只有则当 p>=q时，该请求才会被执行。

该规则是 logback 的核心。级别排序为： TRACE < DEBUG < INFO < WARN < ERROR

## layout转换符

它和C语言的printf方法非常类似。格式转换由普通字符和转换字符组合而成。转换字符由%开始，紧跟着的是可选的格式修饰符和转换字符标示。使用%前缀的表示符号将被转换到实际的内容。如name, level, date, thread name.可用的转换符有：

|  |  |
| --- | --- |
| 转换符 | 描述 |
| c | 调用日志事件的所在记录器的名字，如一个logger的名字是my.test.bbb.ccc，调用的是WARN级别的日志输出，那么输出的是输出my.test.bbb.ccc，可以在其右边指定了精度，如%c{2}那么输出的是bbb.ccc |
| C | 调用日志事件的所在的类名，和c转换符一样，可以在右边指定宽度，如%C{2}输出%C{2} |
| d | 日志调用所发生的时间，日期格式在其后跟着的大括号内的格式指定如%d{yyyy-MM-dd HH:mm:ss}，我现在输出的结果是2011-07-11 21:05:22，推荐使用的是log4j本身提供的日期格式，如%d{ISO8601}，%d{ABSOLUTE}，%d{DATE} |
| F | 所处所在文件名，如上面说C转换符的例子，输出结果是LayoutTest.java |
| l | 是的日志事件发生的位置信息，这个和虚拟机的实现有点关系，一般境况下能得到类，方法，行数源文件等信息， |
| L | 只是输出触发日志事件代码所在的行号，性能损耗会小很多。 |
| m | 显示应用给日志提供的其他信息，如消息。logger.warn("Message 2");那么%m将得到的是Message 2 |
| M | 输出调用者所在的方法名 |
| n | 换行，和\r \r\n有相同功能，能识别系统的换行符，自动转换成\r或者\r\n，log4j推荐使用这个转换符，而不是\r或者\r\n |
| p | 输出调用的日志的级别，如我是调用logger.debug方法，那么级别就是debug |
| r | 输出自应用启动后第一次调用logger的日志输出方法，到输出该log信息耗费的毫秒数 |
| t | 输出所在线程的名字 |
| x | 输出产生的日志事件的线程的NDC（嵌套诊断上下文） |
| X | 输出与生成的日志事件的线程关联的MDC（映射诊断上下文）。X转换符括号之间放置了一个key，就像在％X {clientNumber}中的clientNumberkey 一样。在MDC correspondingvalue将被输出。 |
| % | 写上%%后将直接输出一个%符号 |

## layout格式修饰符

如给定的一个格式：%-5p [%t]: %m%n中，并没有明确的分隔转换字符和普通文本的字符存在。PatternLayout能自己区分普通文本和转换字符。其中%-5p是日志的调用级别。事件是左对齐的，5个字符宽度。

格式修饰符，放在%和转换符之间。 第一个可选的格式修饰符是左对齐(-)；第二个可选的格式修饰符是字段最小宽度。一个整数。表示输出的最小字符数。如果数据未达到指定最小大小，那么它将以左填充（默认）或者右填充方式（左对齐情况下只能使用右填充了）。用空格填充，直到达到最小宽度。如果大于指定最小宽度，不会被截断 。当然可以指定最大字符数，使用.符号加数字表示最大字符数。如果大于指定长度，多余的字符会被删除。它是从前面删除，而不是从后面删除的。如最大字符是8个，数据有10个字符，那么前面两个字符会被删除。

%20c 右对齐，最少20字符，没有左边用空格填充

%-20c 左对齐，最少20字符，没有右边用空格填充

%.30c 右对齐，最多30字符，超过左边的截取掉

%20.30c 右对齐，最少20字符，最多30字符，填充或截取规则略

%-20.30c 左对齐，最少20字符，最多30字符，填充或截取规则略

# Logback的默认配置

如果配置文件 logback-test.xml 和 logback.xml 都不存在，那么 logback 默认地会调用BasicConfigurator ，创建一个最小化配置。最小化配置由一个关联到根 logger 的ConsoleAppender 组成。输出用模式为%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n 的 PatternLayoutEncoder 进行格式化。root logger 默认级别是 DEBUG。

## Logback的配置文件

Logback 配置文件的语法非常灵活。正因为灵活，所以无法用 DTD 或 XML schema 进行定义。尽管如此，可以这样描述配置文件的基本结构：以<configuration>开头，后面有零个或多个<appender>元素，有零个或多个<logger>元素，有最多一个<root>元素。

## Logback默认配置的步骤

(1). 尝试在 classpath 下查找文件 logback-test.xml；

(2). 如果文件不存在，则查找文件 logback.xml；

(3). 如果两个文件都不存在，logback 用 Bas icConfigurator 自动对自己进行配置，这会导致记录输出到控制台。

## Logback.xml 文件

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <configuration>  <!--定义日志文件的存储地址 勿在 LogBack 的配置中使用相对路径-->  <property name="LOG\_HOME" value="c:/log" />  <!-- 控制台输出 -->  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  <!-- 日志输出编码 -->  <Encoding>UTF-8</Encoding>  <layout class="ch.qos.logback.classic.PatternLayout">  <!--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%msg：日志消息，%n是换行符-->  <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} - %msg%n  </pattern>  </layout>  </appender>  <!-- 按照每天生成日志文件 -->  <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">  <Encoding>UTF-8</Encoding>  <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  <!--日志文件输出的文件名-->  <FileNamePattern>${LOG\_HOME}/myApp.log.%d{yyyy-MM-dd}.log</FileNamePattern>  <MaxHistory>30</MaxHistory>  </rollingPolicy>  <layout class="ch.qos.logback.classic.PatternLayout">  <!--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%msg：日志消息，%n是换行符-->  <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} - %msg%n  </pattern>  </layout>  <!--日志文件最大的大小-->  <triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">  <MaxFileSize>10MB</MaxFileSize>  </triggeringPolicy>  </appender>  <!-- show parameters for hibernate sql 专为 Hibernate 定制 -->  <logger name="org.hibernate.type.descriptor.sql.BasicBinder" level="TRACE" />  <logger name="org.hibernate.type.descriptor.sql.BasicExtractor" level="DEBUG" />  <logger name="org.hibernate.SQL" level="DEBUG" />  <logger name="org.hibernate.engine.QueryParameters" level="DEBUG" />  <logger name="org.hibernate.engine.query.HQLQueryPlan" level="DEBUG" />    <!-- 日志输出级别 -->  <root level="INFO">  <appender-ref ref="STDOUT" />  <appender-ref ref="FILE" />  </root>    <!--日志异步到数据库 -->  <appender name="DB" class="ch.qos.logback.classic.db.DBAppender">  <!--日志异步到数据库 -->  <connectionSource class="ch.qos.logback.core.db.DriverManagerConnectionSource">  <!--连接池 -->  <dataSource class="com.mchange.v2.c3p0.ComboPooledDataSource">  <driverClass>com.mysql.jdbc.Driver</driverClass>  <url>jdbc:mysql://127.0.0.1:3306/databaseName</url>  <user>root</user>  <password>root</password>  </dataSource>  </connectionSource>  </appender> -->  </configuration> |

# 程序引用Logback

|  |
| --- |
| package com.stu.system.action;    import org.slf4j.Logger;  import org.slf4j.LoggerFactory;    public class BlogAction{  //定义一个全局的记录器，通过LoggerFactory获取  **private final static Logger logger = LoggerFactory.getLogger(BlogAction.class);**  /\*\*  \* @param args  \*/  public static void main(String[] args) {  logger.info("logback 成功了");  logger.error("logback 成功了");  }  } |

# 基于maven的slf4j+logback pom.xml配置

## maven依赖配置

|  |
| --- |
| <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>1.7.10</version>  </dependency>  <dependency>  <groupId>ch.qos.logback</groupId>  <artifactId>logback-classic</artifactId>  <version>1.1.2</version>  </dependency>  <dependency>  <groupId>ch.qos.logback</groupId>  <artifactId>logback-core</artifactId>  <version>1.1.2</version>  </dependency> |

## 在classpath目录下新建logback.xml配置文件

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <!--  scan：当此属性设置为true时，配置文件如果发生改变，将会被重新加载，默认值为true。  scanPeriod：设置监测配置文件是否有修改的时间间隔，如果没有给出时间单位，默认单位是毫秒当scan为true时，此属性生效。默认的时间间隔为1分钟。  debug：当此属性设置为true时，将打印出logback内部日志信息，实时查看logback运行状态。默认值为false。  -->  <configuration scan="false" scanPeriod="60 seconds" debug="false">  <!-- 定义日志的根目录 -->  <property name="LOG\_HOME" value="/app/log" />  <!-- 定义日志文件名称 -->  <property name="appName" value="netty"></property>  <!-- ch.qos.logback.core.ConsoleAppender 表示控制台输出 -->  <appender name="stdout" class="ch.qos.logback.core.ConsoleAppender">  <Encoding>UTF-8</Encoding>  <!--  日志输出格式：%d表示日期时间，%thread表示线程名，%-5level：级别从左显示5个字符宽度  %logger{50} 表示logger名字最长50个字符，否则按照句点分割。 %msg：日志消息，%n是换行符  -->  <layout class="ch.qos.logback.classic.PatternLayout">  <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} - %msg%n</pattern>  </layout>  </appender>    <!-- 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 -->  <appender name="appLogAppender" class="ch.qos.logback.core.rolling.RollingFileAppender">  <Encoding>UTF-8</Encoding>  <!-- 指定日志文件的名称 -->  <file>${LOG\_HOME}/${appName}.log</file>  <!--  当发生滚动时，决定 RollingFileAppender 的行为，涉及文件移动和重命名  TimeBasedRollingPolicy： 最常用的滚动策略，它根据时间来制定滚动策略，既负责滚动也负责出发滚动。  -->  <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  <!--  滚动时产生的文件的存放位置及文件名称 %d{yyyy-MM-dd}：按天进行日志滚动  %i：当文件大小超过maxFileSize时，按照i进行文件滚动  -->  <fileNamePattern>${LOG\_HOME}/${appName}-%d{yyyy-MM-dd}-%i.log</fileNamePattern>  <!--  可选节点，控制保留的归档文件的最大数量，超出数量就删除旧文件。假设设置每天滚动，  且maxHistory是365，则只保存最近365天的文件，删除之前的旧文件。注意，删除旧文件是，  那些为了归档而创建的目录也会被删除。  -->  <MaxHistory>365</MaxHistory>  <!--  当日志文件超过maxFileSize指定的大小是，根据上面提到的%i进行日志文件滚动 注意此处配置SizeBasedTriggeringPolicy是无法实现按文件大小进行滚动的，必须配置timeBasedFileNamingAndTriggeringPolicy  -->  <timeBasedFileNamingAndTriggeringPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">  <maxFileSize>100MB</maxFileSize>  </timeBasedFileNamingAndTriggeringPolicy>  </rollingPolicy>  <!--  日志输出格式：%d表示日期时间，%thread表示线程名，%-5level：级别从左显示5个字符宽度 %logger{50} 表示logger名字最长50个字符，否则按照句点分割。 %msg：日志消息，%n是换行符  -->  <layout class="ch.qos.logback.classic.PatternLayout">  <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [ %thread ] - [ %-5level ] [ %logger{50} : %line ] - %msg%n</pattern>  </layout>  </appender>    <!--  logger主要用于存放日志对象，也可以定义日志类型、级别  name：表示匹配的logger类型前缀，也就是包的前半部分  level：要记录的日志级别，包括 TRACE < DEBUG < INFO < WARN < ERROR  additivity：作用在于children-logger是否使用 rootLogger配置的appender进行输出，false：表示只用当前logger的appender-ref，true：表示当前logger的appender-ref和rootLogger的appender-ref都有效  -->  <!-- hibernate logger -->  <logger name="org.hibernate" level="error" />  <!-- Spring framework logger -->  <logger name="org.springframework" level="error" additivity="false"></logger>    <logger name="com.creditease" level="info" additivity="true">  <appender-ref ref="appLogAppender" />  </logger>    <!--  root与logger是父子关系，没有特别定义则默认为root，任何一个类只会和一个logger对应，  要么是定义的logger，要么是root，判断的关键在于找到这个logger，然后判断这个logger的appender和level。  -->  <root level="info">  <appender-ref ref="stdout" />  <appender-ref ref="appLogAppender" />  </root>  </configuration> |
| <configuration>  <contextName>standard-code</contextName>  <appender name="console" class="ch.qos.logback.core.ConsoleAppender">  <encoder>  <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %level %logger{55} - %msg%n  </pattern>  </encoder>  </appender>  <appender name="rollFile"  class="ch.qos.logback.core.rolling.RollingFileAppender">  <file>${catalina.base}/logs/3t-admin.log</file>  <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  <FileNamePattern>${catalina.base}/logs/3t-admin.log.%d{yyyy-MM-dd}.log  </FileNamePattern>  <maxHistory>30</maxHistory>  </rollingPolicy>  <encoder>  <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{35} -  %msg%n</pattern>  </encoder>  </appender>  <logger name="com.hengtiansoft" level="warn" />  <root level="error">  <appender-ref ref="console" />  <appender-ref ref="rollFile" level="error" />  </root>  </configuration> |