**一：根节点<configuration>包含的属性：**

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:

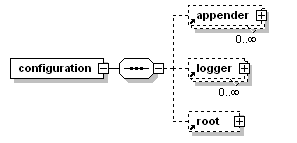
当此属性设置为true时，将打印出logback内部日志信息，实时查看logback运行状态。默认值为false。

例如：

**Xml代码**

1. **<configuration** scan="true" scanPeriod="60 seconds" debug="false"**>**
2. <!-- 其他配置省略-->
3. **</configuration>**

**二：根节点<configuration>的子节点：**



**2.1设置上下文名称：<contextName>**

每个logger都关联到logger上下文，默认上下文名称为“default”。但可以使用<contextName>设置成其他名字，用于区分不同应用程序的记录。一旦设置，不能修改。

**Xml代码**

1. **<configuration** scan="true" scanPeriod="60 seconds" debug="false"**>**
2. **<contextName>**myAppName**</contextName>**
3. <!-- 其他配置省略-->
4. **</configuration>**

**2.2设置变量：** **<property>**

用来定义变量值的标签，<property> 有两个属性，name和value；其中name的值是变量的名称，value的值时变量定义的值。通过<property>定义的值会被插入到logger上下文中。定义变量后，可以使“${}”来使用变量。

例如使用<property>定义上下文名称，然后在<contentName>设置logger上下文时使用。

**Xml代码**

1. **<configuration** scan="true" scanPeriod="60 seconds" debug="false"**>**
2. **<property** name="APP\_Name" value="myAppName" **/>**
3. **<contextName>**${APP\_Name}**</contextName>**
4. <!-- 其他配置省略-->
5. **</configuration>**

**2.3获取时间戳字符串：<timestamp>**

两个属性 key:标识此**<timestamp>** 的名字；datePattern：设置将当前时间（解析配置文件的时间）转换为字符串的模式，遵循java.txt.SimpleDateFormat的格式。

       例如将解析配置文件的时间作为上下文名称：

**Xml代码**

1. **<configuration** scan="true" scanPeriod="60 seconds" debug="false"**>**
2. **<timestamp** key="bySecond" datePattern="yyyyMMdd'T'HHmmss"**/>**
3. **<contextName>**${bySecond}**</contextName>**
4. <!-- 其他配置省略-->
5. **</configuration>**

**2.4设置loger：**

**<loger>**

用来设置某一个包或者具体的某一个类的日志打印级别、以及指定<appender>。<loger>仅有一个name属性，一个可选的level和一个可选的addtivity属性。

name:

用来指定受此loger约束的某一个包或者具体的某一个类。

level:

用来设置打印级别，大小写无关：TRACE, DEBUG, INFO, WARN, ERROR, ALL 和 OFF，还有一个特俗值INHERITED或者同义词NULL，代表强制执行上级的级别。

如果未设置此属性，那么当前loger将会继承上级的级别。

addtivity:

是否向上级loger传递打印信息。默认是true。

<loger>可以包含零个或多个<appender-ref>元素，标识这个appender将会添加到这个loger。

**<root>**

也是<loger>元素，但是它是根loger。只有一个level属性，应为已经被命名为"root".

level:

用来设置打印级别，大小写无关：TRACE, DEBUG, INFO, WARN, ERROR, ALL 和 OFF，不能设置为INHERITED或者同义词NULL。

默认是DEBUG。

<root>可以包含零个或多个<appender-ref>元素，标识这个appender将会添加到这个loger。

例如：

LogbackDemo.java类

**Java代码**

1. **package** logback;
3. **import** org.slf4j.Logger;
4. **import** org.slf4j.LoggerFactory;
6. **public** **class** LogbackDemo {
7. **private** **static** Logger log = LoggerFactory.getLogger(LogbackDemo.**class**);
8. **public** **static** **void** main(String[] args) {
9. log.trace("======trace");
10. log.debug("======debug");
11. log.info("======info");
12. log.warn("======warn");
13. log.error("======error");
14. }
15. }

logback.xml配置文件

**第1种：只配置root**

**Xml代码**

1. **<configuration>**
3. **<appender** name="STDOUT" class="ch.qos.logback.core.ConsoleAppender"**>**
4. <!-- encoder 默认配置为PatternLayoutEncoder -->
5. **<encoder>**
6. **<pattern>**%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n**</pattern>**
7. **</encoder>**
8. **</appender>**
10. **<root** level="INFO"**>**
11. **<appender-ref** ref="STDOUT" **/>**
12. **</root>**
14. **</configuration>**

 其中appender的配置表示打印到控制台(稍后详细讲解appender )；

<root level="INFO">将root的打印级别设置为“INFO”，指定了名字为“STDOUT”的appender。

当执行logback.LogbackDemo类的main方法时，root将级别为“INFO”及大于“INFO”的日志信息交给已经配置好的名为“STDOUT”的appender处理，“STDOUT”appender将信息打印到控制台；

**打印结果如下：**

**Xml代码**

1. 13:30:38.484 [main] INFO  logback.LogbackDemo - ======info
2. 13:30:38.500 [main] WARN  logback.LogbackDemo - ======warn
3. 13:30:38.500 [main] ERROR logback.LogbackDemo - ======error

**第2种：带有loger的配置，不指定级别，不指定appender，**

**Xml代码**

1. **<configuration>**
3. **<appender** name="STDOUT" class="ch.qos.logback.core.ConsoleAppender"**>**
4. <!-- encoder 默认配置为PatternLayoutEncoder -->
5. **<encoder>**
6. **<pattern>**%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n**</pattern>**
7. **</encoder>**
8. **</appender>**
10. <!-- logback为java中的包 -->
11. **<logger** name="logback"**/>**
13. **<root** level="DEBUG"**>**
14. **<appender-ref** ref="STDOUT" **/>**
15. **</root>**
17. **</configuration>**

 其中appender的配置表示打印到控制台(稍后详细讲解appender )；

<logger name="logback" />将控制logback包下的所有类的日志的打印，但是并没用设置打印级别，所以继承他的上级<root>的日志级别“DEBUG”；

没有设置addtivity，默认为true，将此loger的打印信息向上级传递；

没有设置appender，此loger本身不打印任何信息。

<root level="DEBUG">将root的打印级别设置为“DEBUG”，指定了名字为“STDOUT”的appender。

当执行logback.LogbackDemo类的main方法时，因为LogbackDemo 在包logback中，所以首先执行<logger name="logback" />，将级别为“DEBUG”及大于“DEBUG”的日志信息传递给root，本身并不打印；

root接到下级传递的信息，交给已经配置好的名为“STDOUT”的appender处理，“STDOUT”appender将信息打印到控制台；

**打印结果如下：**

**Xml代码**

1. 13:19:15.406 [main] DEBUG logback.LogbackDemo - ======debug
2. 13:19:15.406 [main] INFO  logback.LogbackDemo - ======info
3. 13:19:15.406 [main] WARN  logback.LogbackDemo - ======warn
4. 13:19:15.406 [main] ERROR logback.LogbackDemo - ======error

**第3种：带有多个loger的配置，指定级别，指定appender**

**Xml代码**

1. **<configuration>**
2. **<appender** name="STDOUT" class="ch.qos.logback.core.ConsoleAppender"**>**
3. <!-- encoder 默认配置为PatternLayoutEncoder -->
4. **<encoder>**
5. **<pattern>**%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n**</pattern>**
6. **</encoder>**
7. **</appender>**
9. <!-- logback为java中的包 -->
10. **<logger** name="logback"**/>**
11. <!--logback.LogbackDemo：类的全路径 -->
12. **<logger** name="logback.LogbackDemo" level="INFO" additivity="false"**>**
13. **<appender-ref** ref="STDOUT"**/>**
14. **</logger>**
16. **<root** level="ERROR"**>**
17. **<appender-ref** ref="STDOUT" **/>**
18. **</root>**
19. **</configuration>**

其中appender的配置表示打印到控制台(稍后详细讲解appender )；

<logger name="logback" />将控制logback包下的所有类的日志的打印，但是并没用设置打印级别，所以继承他的上级<root>的日志级别“DEBUG”；

没有设置addtivity，默认为true，将此loger的打印信息向上级传递；

没有设置appender，此loger本身不打印任何信息。

 <logger name="logback.LogbackDemo" level="INFO" additivity="false">控制logback.LogbackDemo类的日志打印，打印级别为“INFO”；

additivity属性为false，表示此loger的打印信息不再向上级传递，

指定了名字为“STDOUT”的appender。

<root level="DEBUG">将root的打印级别设置为“ERROR”，指定了名字为“STDOUT”的appender。

 当执行logback.LogbackDemo类的main方法时，先执行<logger name="logback.LogbackDemo" level="INFO" additivity="false">，将级别为“INFO”及大于“INFO”的日志信息交给此loger指定的名为“STDOUT”的appender处理，在控制台中打出日志，不再向次loger的上级 <logger name="logback"/> 传递打印信息；

<logger name="logback"/>未接到任何打印信息，当然也不会给它的上级root传递任何打印信息；

**打印结果如下：**

**Xml代码**

1. 14:05:35.937 [main] INFO  logback.LogbackDemo - ======info
2. 14:05:35.937 [main] WARN  logback.LogbackDemo - ======warn
3. 14:05:35.937 [main] ERROR logback.LogbackDemo - ======error

 如果将<logger name="logback.LogbackDemo" level="INFO" additivity="false">修改为 <logger name="logback.LogbackDemo" level="INFO" additivity="true">那打印结果将是什么呢？

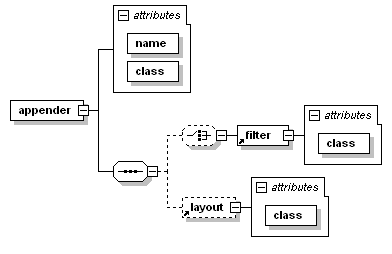
没错，日志打印了两次，想必大家都知道原因了，因为打印信息向上级传递，logger本身打印一次，root接到后又打印一次

**打印结果如下：**

**Xml代码**

1. 14:09:01.531 [main] INFO  logback.LogbackDemo - ======info
2. 14:09:01.531 [main] INFO  logback.LogbackDemo - ======info
3. 14:09:01.531 [main] WARN  logback.LogbackDemo - ======warn
4. 14:09:01.531 [main] WARN  logback.LogbackDemo - ======warn
5. 14:09:01.531 [main] ERROR logback.LogbackDemo - ======error
6. 14:09:01.531 [main] ERROR logback.LogbackDemo - ======error

**二 <appender>**



**<appender>：**

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

<appender>有两个必要属性name和class。name指定appender名称，class指定appender的全限定名。

**1.ConsoleAppender:**

把日志添加到控制台，有以下子节点：

<encoder>：对日志进行格式化。（具体参数稍后讲解 ）

<target>：字符串 *System.out*或者 *System.err* ，默认 *System.out* ；

例如：

**Xml代码**

1. **<configuration>**
3. **<appender** name="STDOUT" class="ch.qos.logback.core.ConsoleAppender"**>**
4. **<encoder>**
5. **<pattern>**%-4relative [%thread] %-5level %logger{35} - %msg %n**</pattern>**
6. **</encoder>**
7. **</appender>**
9. **<root** level="DEBUG"**>**
10. **<appender-ref** ref="STDOUT" **/>**
11. **</root>**
12. **</configuration>**

**2.FileAppender:**

把日志添加到文件，有以下子节点：

<file>：被写入的文件名，可以是相对目录，也可以是绝对目录，如果上级目录不存在会自动创建，没有默认值。

<append>：如果是 true，日志被追加到文件结尾，如果是 false，清空现存文件，默认是true。

<encoder>：对记录事件进行格式化。（具体参数稍后讲解 ）

<prudent>：如果是 true，日志会被安全的写入文件，即使其他的FileAppender也在向此文件做写入操作，效率低，默认是 false。

例如：

**Xml代码**

1. **<configuration>**
3. **<appender** name="FILE" class="ch.qos.logback.core.FileAppender"**>**
4. **<file>**testFile.log**</file>**
5. **<append>**true**</append>**
6. **<encoder>**
7. **<pattern>**%-4relative [%thread] %-5level %logger{35} - %msg%n**</pattern>**
8. **</encoder>**
9. **</appender>**
11. **<root** level="DEBUG"**>**
12. **<appender-ref** ref="FILE" **/>**
13. **</root>**
14. **</configuration>**

**3.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。有以下子节点：

<file>：被写入的文件名，可以是相对目录，也可以是绝对目录，如果上级目录不存在会自动创建，没有默认值。

<append>：如果是 true，日志被追加到文件结尾，如果是 false，清空现存文件，默认是true。

<encoder>：对记录事件进行格式化。（具体参数稍后讲解 ）

<rollingPolicy>:当发生滚动时，决定 **RollingFileAppender** 的行为，涉及文件移动和重命名。

<triggeringPolicy **>:**告知 **RollingFileAppender** 合适激活滚动。

<prudent>：当为true时，不支持FixedWindowRollingPolicy。支持TimeBasedRollingPolicy，但是有两个限制，1不支持也不允许文件压缩，2不能设置file属性，必须留空。

**rollingPolicy：**

**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。有以下子节点：

<fileNamePattern>:

必要节点，包含文件名及“%d”转换符， “%d”可以包含一个java.text.SimpleDateFormat指定的时间格式，如：%d{yyyy-MM}。如果直接使用 %d，默认格式是 yyyy-MM-dd。**RollingFileAppender** 的file字节点可有可无，通过设置file，可以为活动文件和归档文件指定不同位置，当前日志总是记录到file指定的文件（活动文件），活动文件的名字不会改变；如果没设置file，活动文件的名字会根据**fileNamePattern**的值，每隔一段时间改变一次。“/”或者“\”会被当做目录分隔符。

<maxHistory>:

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。假设设置每个月滚动，且<maxHistory>是6，则只保存最近6个月的文件，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。

**FixedWindowRollingPolicy：** 根据固定窗口算法重命名文件的滚动策略。有以下子节点：

<minIndex>:窗口索引最小值

<maxIndex>:窗口索引最大值，当用户指定的窗口过大时，会自动将窗口设置为12。

<fileNamePattern >:

必须包含“%i”例如，假设最小值和最大值分别为1和2，命名模式为 mylog%i.log,会产生归档文件mylog1.log和mylog2.log。还可以指定文件压缩选项，例如，mylog%i.log.gz 或者 没有log%i.log.zip

**triggeringPolicy:**

**SizeBasedTriggeringPolicy：** 查看当前活动文件的大小，如果超过指定大小会告知**RollingFileAppender**触发当前活动文件滚动。只有一个节点**:**

<maxFileSize>:这是活动文件的大小，默认值是10MB。

例如：每天生成一个日志文件，保存30天的日志文件。

**Java代码**

1. <configuration>
2. <appender name="FILE" **class**="ch.qos.logback.core.rolling.RollingFileAppender">
4. <rollingPolicy **class**="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
5. <fileNamePattern>logFile.%d{yyyy-MM-dd}.log</fileNamePattern>
6. <maxHistory>30</maxHistory>
7. </rollingPolicy>
9. <encoder>
10. <pattern>%-4relative [%thread] %-5level %logger{35} - %msg%n</pattern>
11. </encoder>
12. </appender>
14. <root level="DEBUG">
15. <appender-ref ref="FILE" />
16. </root>
17. </configuration>

  例如：按照固定窗口模式生成日志文件，当文件大于20MB时，生成新的日志文件。窗口大小是1到3，当保存了3个归档文件后，将覆盖最早的日志。

**Xml代码**

1. **<configuration>**
2. **<appender** name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender"**>**
3. **<file>**test.log**</file>**
5. **<rollingPolicy** class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy"**>**
6. **<fileNamePattern>**tests.%i.log.zip**</fileNamePattern>**
7. **<minIndex>**1**</minIndex>**
8. **<maxIndex>**3**</maxIndex>**
9. **</rollingPolicy>**
11. **<triggeringPolicy** class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy"**>**
12. **<maxFileSize>**5MB**</maxFileSize>**
13. **</triggeringPolicy>**
14. **<encoder>**
15. **<pattern>**%-4relative [%thread] %-5level %logger{35} - %msg%n**</pattern>**
16. **</encoder>**
17. **</appender>**
19. **<root** level="DEBUG"**>**
20. **<appender-ref** ref="FILE" **/>**
21. **</root>**
22. **</configuration>**

4.另外还有SocketAppender、SMTPAppender、DBAppender、SyslogAppender、SiftingAppender，并不常用，这些就不在这里讲解了，大家可以参考官方文档。当然大家可以编写自己的Appender。

**<encoder>：**

负责两件事，一是把日志信息转换成字节数组，二是把字节数组写入到输出流。

目前**PatternLayoutEncoder** 是唯一有用的且默认的**encoder** ，有一个<pattern>节点，用来设置日志的输入格式。使用“%”加“转换符”方式，如果要输出“%”，则必须用“\”对“\%”进行转义。

例如：

**Xml代码**

1. **<encoder>**
2. **<pattern>**%-4relative [%thread] %-5level %logger{35} - %msg%n**</pattern>**
3. **</encoder**