

# Logback 手册

## 中文版

文档版本：0.1

发布日期：2010 年 4 月 23 日

原作者：Ceki Gülcü、Sébastien Penne

中文版译者：陈华

联系方式：[clinker@163.com](mailto:clinker@163.com)

---

# 目录

目录 .....	I
译者声明.....	1
发布记录.....	1
1. 介绍.....	2
1.1. 什么是 logback .....	2
1.2. 第一步 .....	2
1.2.1. 必要条件.....	2
1.3. 构建 logback.....	5
2. 体系结构.....	6
2.1. logback 的体系结构 .....	6
2.2. Logger、Appender 和 Layout .....	6
2.2.1. Logger 上下文 .....	6
2.2.2. 有效级别（Level）即级别继承 .....	7
2.2.3. 打印方法和基本选择规则 .....	9
2.2.4. 获取 Logger.....	10
2.2.5. Appender 和 Layout.....	11
2.2.6. 参数化记录 .....	13
2.2.7. 更好的替代方法 .....	13
2.2.8. 工作原理.....	14
2.2.9. 性能.....	15
3. 配置.....	17
3.1. Logback 里的配置 .....	17
3.2. 自动配置.....	17
3.3. 用 logback-test.xml 或 logback.xml 自动配置 .....	19
3.4. 自动打印警告和错误消息 .....	19
3.5. 把默认配置文件的位置作为系统属性进行指定.....	21
3.6. 配置文件修改后自动重新加载 .....	22
3.7. 直接调用 JoranConfigurator .....	22

3.8.	查看状态消息.....	24
3.9.	监听状态消息.....	25
3.10.	配置文件语法.....	26
3.10.1.	标记名大小写敏感性.....	26
3.10.2.	配置 logger，或<logger>元素.....	26
3.10.3.	配置根 logger，或<root>元素.....	27
3.10.4.	示例.....	27
3.10.5.	配置 Appender.....	30
3.10.6.	Appender 累积.....	32
3.10.7.	覆盖默认的累积行为.....	34
3.10.8.	设置上下文名称.....	35
3.10.9.	变量替换.....	36
4.	Appender.....	44
4.1.	什么是 Appender.....	44
4.2.	AppenderBase.....	45
4.3.	Logback-core.....	46
4.3.1.	OutputStreamAppender.....	46
4.3.2.	ConsoleAppender.....	47
4.3.3.	FileAppender.....	48
4.3.4.	RollingFileAppender.....	51
4.3.5.	TimeBasedRollingPolicy.....	54
4.3.6.	触发策略概述.....	60
4.4.	Logback Classic.....	61
4.4.1.	SocketAppender.....	61
4.4.2.	JMSAppenderBase.....	64
4.4.3.	SMTPAppender.....	70
4.4.4.	DBAppender.....	78
4.4.5.	SyslogAppender.....	84
4.4.6.	SiftingAppender.....	85
4.4.7.	自定义 Appender.....	87

4.5.	Logback Access.....	89
4.5.1.	SocketAppender .....	89
4.5.2.	SMTPAppender .....	89
4.5.3.	DBAppender .....	90
4.5.4.	SiftingAppender .....	92
5.	Encoder.....	94
5.1.	什么是 encoder.....	94
5.2.	Encoder 接口.....	94
5.3.	LayoutWrappingEncoder .....	95
5.4.	PatternLayoutEncoder .....	96
6.	排版 (Layout) .....	97
6.1.	什么是 layout .....	97
6.2.	Logback-classic.....	97
6.2.1.	自定义 layout .....	97
6.2.2.	PatternLayout.....	101
6.2.3.	转换符说明 .....	103
6.2.4.	格式修饰符 .....	107
6.2.5.	圆括号的特殊含义.....	108
6.2.6.	求值式 (Evaluator) .....	110
6.2.7.	创建自定义格式转换符.....	115
6.2.8.	HTMLLayout .....	117
6.3.	Logback access.....	119
6.3.1.	自定义 layout .....	119
6.3.2.	PatternLayout.....	119
6.3.3.	HTMLLayout .....	122
7.	过滤器 (Filter) .....	123
7.1.	在 logback-classic 里 .....	123
7.1.1.	常规过滤器 .....	123
7.1.2.	TurboFilters .....	132
7.1.3.	重复消息过滤器 (DuplicateMessageFilter) .....	135

7.2.	在 logback-access 里.....	137
7.2.1.	过滤器 .....	137
8.	映射诊断环境（Mapped Diagnostic Context） .....	139
8.1.	高级用法.....	141
8.2.	自动访问 MDC.....	148
8.3.	MDC 和受管线程 .....	150
8.3.1.	MDCInsertingServletFilter .....	150
9.	记录隔离.....	152
9.1.	最简易的方法.....	152
9.2.	上下文选择器（Context Selector） .....	152
9.2.1.	ContextJNDISelector .....	153
9.2.2.	在应用程序里设置 JNDI 变量 .....	153
9.2.3.	为 Tomcat 配置 ContextJNDISelector.....	154
9.3.	在共享类库里使用静态引用.....	155
10.	JMX 配置器 .....	159
10.1.	使用 JMX 配置器 .....	159
10.2.	避免内存泄露.....	161
10.3.	多个应用程序里的 JMXConfigurator.....	161
10.4.	支持 JMX.....	162
10.4.1.	Jetty 启用 JMX（在 JDK 1.5 和 JDK1.6 上通过测试） .....	163
10.4.2.	Jetty 启用 MX4J（在 JDK 1.5 和 JDK1.6 上通过测试） .....	164
10.4.3.	Tomcat 启用 JMX（在 JDK 1.5 和 JDK1.6 上通过测试） .....	166
10.4.4.	Tomcat 启用 MX4J（在 JDK 1.5 和 JDK1.6 上通过测试） .....	166
11.	Joran.....	168
11.1.	历史回顾.....	168
11.2.	SAX 还是 DOM? .....	169
11.3.	模式（Pattern） .....	169
11.4.	动作（Action） .....	169
11.5.	RuleStore.....	170
11.6.	解释上下文（Interpretation context） .....	171

---

11.7.	Hello world.....	171
11.8.	合作动作.....	172
11.9.	隐式动作（Implicit actions）.....	173
11.10.	实践中的隐式动作.....	175
11.10.1.	默认类映射.....	176
11.10.2.	属性集合.....	176
11.10.3.	动态新规则.....	176
12.	从 log4j 迁移.....	178
12.1.	迁移 log4j 的 layout.....	178
12.2.	迁移 log4j 的 appender.....	179

## 译者声明

原文档版权说明: <http://creativecommons.org/licenses/by-nc-sa/2.5/>

原文档地址: <http://logback.qos.ch/manual/index.html>

本人遵守原文档之版权规定。

翻译此文档的目的是仅为方便本人学习和阅读。

在引用、转载本文档的部分全部内容时, 请标注译者信息, 包括姓名和电子邮件地址。

如果您阅读本文档, 即表示已经完全接受并遵守上述声明。由于您违反上述声明而造成的一切后果, 由您承担, 本人不承担任何责任。

译文处理了原手册的部分文字错误。

译文根据 logback 发行包的 actual 代码和文档对原手册进行了部分修正。

翻译本文时, logback 版本是 0.9.20, slf4j 版本是 1.5.11。

## 发布记录

版本	日期	作者	说明
0.1	2010-04-23	陈华	

# 1. 介绍

## 1.1. 什么是 logback

Logback 为取代 log4j 而生。

Logback 由 log4j 的创立者 Ceki Gülcü 设计。以十多年设计工业级记录系统的经验为基础，所创建的 logback 比现有任何记录系统更快、占用资源更少，有时差距非常大。

Logback 提供独特而实用的特性，比如 Marker、参数化记录语句、条件化堆栈跟踪和强大的事件过滤功能。以上列出的仅仅是 logback 实用特性的一小部分。

对于自身的错误报告，logback 依赖状态（Status）对象，状态对象极大地简化了故障查找。你也许想在上下文中使用状态对象而不是记录。

Logback-core 附带了 Joran，Joran 是个强大的、通用的配置系统，你可以在自己的项目里使用 Joran 以获得巨大的作用。

## 1.2. 第一步

### 1.2.1. 必要条件

Logback-classic 依赖 slf4j-api.jar 和 logback-core.jar。

现在让我们开始体验 logback。

示例 1.1：记录基本模版

(logback-examples/src/main/java/chapters/introduction/HelloWorld1.java)

```
package chapters.introduction;

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

public class HelloWorld1 {

    public static void main(String[] args) {

        Logger logger = LoggerFactory
            .getLogger("chapters.introduction.HelloWorld1");
```



```
        logger.debug("Hello world.");
    }
}
```

HelloWorld1 类导入了 SLF4J API 定义的 Logger 类和 LoggerFactory 类，更明确地说是定义在 org.slf4j 包里的两个类。

main()方法的第一行里，调用 LoggerFactory 类的静态方法 getLogger 取得一个 Logger 实例，将该实例赋值给变量 logger。这个 logger 被命名为“chapters.introduction.HelloWorld1”。main 方法继续调用这个 logger 的 debug 方法并传递参数“Hello world”。我们称之为 main 方法包含了一条消息是“Hello world”、级别是 DEBUG 的记录语句。

注意上面的例子并没有引用任何 logback 的类。多数情况下，只要涉及到记录，你只需要引用 SLF4J 的类。因此在绝大多数情况下，你的类只导入 SLF4J 的 API，基本可以忽略 logback 的存在。

运行示例程序：

```
java chapters.introduction.HelloWorld1
```

运行后会在控制台输出下面的一行文字。得益于 logback 提供了默认配置策略，当没有发现默认配置文件时，logback 会为根（root）logger 添加一个 ConsoleAppender。

```
20:49:07.962 [main] DEBUG chapters.introduction.HelloWorld1 - Hello world.
```

Logback 可以通过内置的状态系统来报告其内部状态。通过 StatusManager 组件可以访问 logback 生命期内发生的重要事件。目前，我们调用 StatusPrinter 类的 print()方法来打印 logback 的内部状态。

示例 1.2：打印 Logger 状态

(logback-examples/src/main/java/chapters/introduction/HelloWorld2.java)

```
package chapters.introduction;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import ch.qos.logback.classic.LoggerContext;
import ch.qos.logback.core.util.StatusPrinter;

public class HelloWorld2 {

    public static void main(String[] args) {
        Logger logger = LoggerFactory
            .getLogger("chapters.introduction.HelloWorld2");
```

```
logger.debug("Hello world.");

// print internal state
LoggerContext lc = (LoggerContext)
LoggerFactory.getILoggerFactory();
    StatusPrinter.print(lc);
}
}
```

运行后输出如下：

```
12:49:22.203 [main] DEBUG chapters.introduction.HelloWorld2 - Hello world.
12:49:22.078 |-INFO in ch.qos.logback.classic.LoggerContext[default] - Could NOT find
resource [logback-test.xml]
12:49:22.093 |-INFO in ch.qos.logback.classic.LoggerContext[default] - Could NOT find
resource [logback.xml]
12:49:22.093 |-INFO in ch.qos.logback.classic.LoggerContext[default] - Setting up default
configuration.
```

Logback 说它没有找到配置文件 `logback-test.xml` 和 `logback.xml`（稍后解释），于是用默认策略进行配置，即用一个基本的 `ConsoleAppender`。`Appender` 类可被视为输出目的地的。`Appender` 包含许多不同类型的目的地，包括控制台、文件、Syslog、TCP 套接字、JMS 和其他。用户可以很容易地自定义 `Appender`。

当发生错误时，logback 将自动在控制台上打印其内部状态。

之前的两个示例相当简单，大型程序里真实记录志情况也不会有太大区别。记录系统的基本模式不会改变，可能改变的是配置过程。也许你想按照自己的需要来定制或配置 logback，之后的章节会讨论配置 logback。

在上面的例子里，我们调用 `StatusPrinter.print()` 方法来打印 logback 的内部状态。在诊断与 logback 相关的问题时，logback 的内部状态信息会非常有用。

在应用程序里启用记录的三个必需步骤如下：

1. 配置 logback 环境。方法有繁有简，稍后讨论。
2. 在每个需要执行记录的类里，调用 `org.slf4j.LoggerFactory` 类的 `getLogger()` 方法获取一个 `Logger` 实例，以当前类名或类本身作为参数。

3. 调用取得的 `logger` 实例的打印方法，即 `debug()`、`info()`、`warn()`和 `error()`，把记录输出到配置里的各 `appender`。

## 1.3. 构建 logback

Logback 使用 Maven2 进行构建。

安装 Maven2 后，解压 logback 发行包，在解压后的目录下执行 `mvn package` 命令，就可以构建整个 logback 项目，包括各个模块。Maven 会自动下载所需外部类库。

Logback 发行包包含完整的源代码，你可以修改源代码，创建自己的版本。你还可以发布修改过的版本，前提是遵守 LGPL 或 EPL。

Logback 在以下 JDK 进行过构建和测试。

JDK	Operating System
Sun JDK 1.5.0.06	Windows XP
Sun JDK 1.5.0.08	Linux 64bit AMD
WebLogic JRockit 1.5.0.14	Linux 64bit AMD
IBM JDK 1.6.0.1	Linux 64bit AMD
Sun JDK 1.6.0.16 (64 bit)	Windows 7 (64 bit)

## 2. 体系结构

### 2.1. logback 的体系结构

Logback 的基本结构充分通用，可应用于各种不同环境。目前，logback 分为三个模块：Core、Classic 和 Access。

Core 模块是其他两个模块的基础。Classic 模块扩展了 core 模块。Classic 模块相当于 log4j 的显著改进版。Logback-classic 直接实现了 SLF4J API，因此你可以在 logback 与其他记录系统如 log4j 和 java.util.logging (JUL) 之间轻松互相切换。Access 模块与 Servlet 容器集成，提供 HTTP 访问记录功能。本文不讲述 access 模块。

本文中，“logback”代表 logback-classic 模块。

### 2.2. Logger、Appender 和 Layout

Logback 建立于三个主要类之上：Logger、Appender 和 Layout。这三种组件协同工作，使开发者可以按照消息类型和级别来记录消息，还可以在程序运行期内控制消息的输出格式和输出目的地。

Logger 类是 logback-classic 模块的一部分，而 Appender 和 Layout 接口来自 logback-core。作为一个多用途模块，logback-core 不包含任何 logger。

#### 2.2.1. Logger 上下文

任何比 System.out.println 高级的记录 API 的第一个也是最重要的优点便是能够在禁用特定记录语句的同时却不妨碍输出其他语句。这种能力源自记录隔离（space）——即所有各种记录语句的隔离——是根据开发者选择的条件而进行分类的。在 logback-classic 里，这种分类是 logger 固有的。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。

Logger 是命名了的实体。它们的名字大小写敏感且遵从下面的层次化的命名规则：

**命名层次：**

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。同理，“java”是“java.util”之父，也是“java.util.Vector”的祖先。

根 **logger** 位于 **logger** 等级的最顶端，它的特别之处是它是每个层次等级的共同始祖。如同其他各 **logger**，根 **logger** 可以通过其名称取得，如下所示：

```
Logger rootLogger = LoggerFactory.getLogger(org.slf4j.Logger.ROOT_LOGGER_NAME);
```

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。`Logger` 接口的部分基本方法列举如下：

```
package org.slf4j;

public interface Logger {

    // Printing methods:
    public void trace(String message);
    public void debug(String message);
    public void info(String message);
    public void warn(String message);
    public void error(String message);
}
```

### 2.2.2. 有效级别（Level）即级别继承

`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。注意在 `logback` 里，`Level` 类是 `final` 的，不能被继承，`Marker` 对象提供了更灵活的方法。

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。更正式地说：

**logger** `L` 的有效级别等于其层次等级里的第一个非 `null` 级别，顺序是从 `L` 开始，向上直至根 **logger**。

为确保所有 logger 都能够最终继承一个级别，根 logger 总是有级别，默认情况下，这个级别是 DEBUG。

下面的四个例子包含各种分配级别值和根据级别继承规则得出的最终有效(继承)级别。

● 例 1

Logger 名	分配级别	有效级别
root	DEBUG	DEBUG
X	none	DEBUG
X.Y	none	DEBUG
X.Y.Z	none	DEBUG

例 1 里，仅根 logger 被分配了级别。级别值 DEBUG 被其他 logger X、X.Y 和 X.Y.Z 继承。

● 例 2

Logger 名	分配级别	有效级别
root	ERROR	ERROR
X	INFO	INFO
X.Y	DEBUG	DEBUG
X.Y.Z	WARN	WARN

例 2 里，所有 logger 都被分配了级别。级别继承不发挥作用。

● 例 3

Logger 名	分配级别	有效级别
root	DEBUG	DEBUG
X	INFO	INFO
X.Y	none	INFO
X.Y.Z	ERROR	ERROR

例 3 里，根 logger、X 和 X.Y.Z 分别被分配了 DEBUG、INFO 和 ERROR 级别。X.Y 从其父 X 继承级别。

● 例 4

Logger 名	分配级别	有效级别
----------	------	------

root	DEBUG	DEBUG
X	INFO	INFO
X.Y	none	INFO
X.Y.Z	none	none

例 4 里，根 logger 和 X 分别被分配了 DEBUG 和 INFO 级别。X.Y 和 X.Y.Z 从其最近的父 X 继承级别，因为 X 被分配了级别。

### 2.2.3. 打印方法和基本选择规则

根据定义，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。如前所述，没有被分配级别的 logger 将从其最近的祖先继承级别。该规则总结如下：

#### 基本选择规则

记录请求级别为  $p$ ，其 logger 的有效级别为  $q$ ，只有当  $p \geq q$  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。

下表显示了选择规则是如何工作的。行头是记录请求的级别  $p$ 。列头是 logger 的有效级别  $q$ 。行（请求级别）与列（有效级别）的交叉部分是按照基本选择规则得出的布尔值。

请求级别 $p$	有效级别 $q$					
	TRACE	DEBUG	INFO	WARN	ERROR	OFF
TRACE	YES	NO	NO	NO	NO	NO
DEBUG	YES	YES	NO	NO	NO	NO
INFO	YES	YES	YES	NO	NO	NO
WARN	YES	YES	YES	YES	NO	NO
ERROR	YES	YES	YES	YES	YES	NO
OFF	YES	NO	NO	NO	NO	NO

下面是基本选择规则的例子。

```
// 取得名为"com.foo"的 logger 实例
Logger logger = LoggerFactory.getLogger("com.foo");

// 设其级别为 INFO
logger.setLevel(Level.INFO);

Logger barlogger = LoggerFactory.getLogger("com.foo.Bar");

// 该请求有效, 因为 WARN >= INFO
logger.warn("Low fuel level.");

// 该请求无效, 因为 DEBUG < INFO.
logger.debug("Starting search for nearest gas station.");

// 名为"com.foo.Bar"的 logger 实例 barlogger, 从"com.foo"继承级别
// 因此下面的请求有效, 因为 INFO >= INFO.
barlogger.info("Located nearest gas station.");

// 该请求无效, 因为 DEBUG < INFO.
barlogger.debug("Exiting gas station search");
```

译者注: 上例的 `logger.setLevel(Level.INFO)` 无效。org.slf4j.Logger 没有 `setLevel()` 方法, `ch.qos.logback.classic.Logger` 有此方法。

## 2.2.4. 获取 Logger

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。

例如,

```
Logger x = LoggerFactory.getLogger("wombat");
Logger y = LoggerFactory.getLogger("wombat");
```

x 和 y 指向同一个 logger 对象。

因此, 可以配置一个 logger, 然后从其他地方取得同一个实例, 不需要到处传递引用。生物学里的父母总是先于其孩子, 而 logback 不同, 它可以以任何顺序创建和配置 logger。特别的是, 即使“父”logger 是在其后代初始化之后才初始化的, 它仍将查找并链接到其后



代们。

通常是在程序初始化时对 logback 环境进行配置。推荐用读配置文件类进行配置。稍后会讲这种方法。

Logback 简化了 logger 命名，方法是在每个类里初始化 logger，以类的全限定名作为 logger 名。这种定义 logger 的方法即有用又直观。由于记录输出里包含 logger 名，这种命名方法很容易确定记录消息来源。Logback 不限制 logger 名，你可以随意命名 logger。

然而，目前已知最好的策略是以 logger 所在类的名字作为 logger 名称。

## 2.2.5. Appender 和 Layout

有选择性地启用或禁用记录请求仅仅是 logback 功能的冰山一角。Logback 允许打印记录请求到多个目的地。在 logback 里，一个输出目的地称为一个 appender。目前有控制台、文件、远程套接字服务器、MySQL、PostgreSQL、Oracle 和其他数据库、JMS 和远程 UNIX Syslog 守护进程等多种 appender。

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。例如，如果根 logger 有一个控制台 appender，那么所有启用了的请求都至少会被打印到控制台。如果 logger L 有额外的文件 appender，那么，L 和 L 后代的所有启用了的请求都将同时打印到控制台和文件。设置 logger 的 `additivity` 为 `false`，则可以取消这种默认的 appender 累积行为。

控制 appender 叠加性的规则总结如下。

### Appender 叠加性

Logger L 的记录语句的输出会发送给 L 及其祖先的全部 appender。这就是“appender 叠加性”的含义。

然而，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。

示例：

Logger 名	关联的 Appender	叠加性标识	输出目标	说明
root	A1	不可用	A1	叠加性标识不适用于根 logger
x	A-x1, A-x2	true	A1, A-x1, A-x2	根和 x
x.y	none	true	A1, A-x1, A-x2	根和 x
x.y.z	A-xyz1	true	A1, A-x1, A-x2, A-xyz1	根、x.y.z 和 x
security	A-sec	false	A-sec	因为叠加性标识为 false，所以 appender 不累积。只有 A-sec
security.access	none	true	A-sec	只有 security，因为 security 的叠加性标识为 false。

有些用户希望不仅可以定制输出目的地，还可以定制输出格式。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 负责将格式化后的输出发送到目的地。PatternLayout 是标准 logback 发行包的一部分，允许用户按照类似于 C 语言的 printf 函数的转换模式设置输出格式。

例如，转换模式"%-4relative [%thread] %-5level %logger{32} - %msg%n"在 PatternLayout 里会输出形如：

```
176 [main] DEBUG manual.architecture.HelloWorld2 - Hello world.
```

第一个字段是自程序启动以来的逝去时间，单位是毫秒。

第二个地段发出记录请求的线程。

第三个字段是记录请求的级别。

第四个字段是与记录请求关联的 `logger` 的名称。

“-”之后是请求的消息文字。

### 2.2.6. 参数化记录

因为 `logback-classic` 里的 `logger` 实现了 `SLF4J` 的 `Logger` 接口，某些打印方法可接受多个参数。这些不同的打印方法主要是为了提高性能的同时尽量不影响代码可读性。

对于某个 `Logger`，下面的代码

```
logger.debug("Entry number: " + i + " is " + String.valueOf(entry[i]));
```

在构造消息参数时有性能消耗，即把整数 `i` 和 `entry[i]` 都转换为字符串时，还有连接多个字符串时。不管消息会不会被记录，都会造成上述消耗。

一个可行的办法是用测试语句包围记录语句以避免上述消耗，比如，

```
if(logger.isDebugEnabled()) {  
    logger.debug("Entry number: " + i + " is " + String.valueOf(entry[i]));  
}
```

当 `logger` 的 `debug` 级别被禁用时，这个方法可以避免参数构造带来的性能消耗。另一方面，如果 `logger` 的 `DEBUG` 级别被启用，那么会导致两次评估 `logger` 是否被启用：一次是 `isDebugEnabled` 方法，一次是 `debug` 方法。在实践中，这种额外开销无关紧要，因为评估 `logger` 所消耗的时间不足实际记录请求所用时间的 1%。

### 2.2.7. 更好的替代方法

还有一种基于消息格式的方便的替代方法。假设 `entry` 是一个 `object`，你可以编写：

```
Object entry = new SomeObject();  
logger.debug("The entry is {}", entry);
```

在评估是否作记录后，仅当需要作记录时，`logger` 才会格式化消息，用 `entry` 的字符串值替换“{}”。换句话说，当记录语句被禁用时，这种方法不会产生参数构造所带来的性能消耗。

## 2.2.8. 工作原理

介绍过 logback 的核心组件后，下面描述 logback 框架在用户调用 logger 的打印方法时所做的事情。在本例中，用户调用名为 `com.wombat` 的 logger 的 `info()` 方法。

### 1. 取得过滤链（filter chain）的判定结果

如果 TurboFilter 链存在，它将被调用。Turbo filters 能够设置一个上下文范围内的临界值，这个临界值或者表示过滤某些与信息有关（比如 Marker、级别、Logger、消息）的特定事件，或者表示与每个记录请求相关联的 Throwable。如果过滤链的结果是 `FilterReply.DENY`，则记录请求被抛弃。如果结果是 `FilterReply.NEUTRAL`，则继续下一步，也就是第二步。如果结果是 `FilterReply.ACCEPT`，则忽略过第二步，进入第三步。

### 2. 应用基本选择规则

Logback 对 logger 的有效级别与请求的级别进行比较。如果比较的结果是记录请求被禁用，logback 会直接抛弃请求，不做任何进一步处理。否则，继续下一步。

### 3. 创建 LoggingEvent 对象

记录请求到了这一步后，logback 会创建一个 `ch.qos.logback.classic.LoggingEvent` 对象，该对象包含所有与请求相关的参数，比如请求用的 logger、请求级别、消息、请求携带的异常、当前时间、当前线程、执行记录请求的类的各种数据，还有 MDC。注意有些成员是延迟初始化的，只有当它们真正被使用时才会被初始化。MDC 用来为记录请求添加额外的上下文信息。之后的章节会讨论 MDC。

### 4. 调用 appender

创建了 LoggingEvent 对象后，logback 将调用所有可用 appender 的 `doAppend()` 方法，这就是说，appender 继承 logger 的上下文。

所有 appender 都继承 AppenderBase 抽象类，AppenderBase 在一个同步块里实现了 `doAppend` 方以确保线程安全。AppenderBase 的 `doAppend()` 方法也调用 appender 关联的自定义过滤器，如果它们存在的话。自定义过滤器能被动态地关联到任何 appender，另有章节专门讲述它。

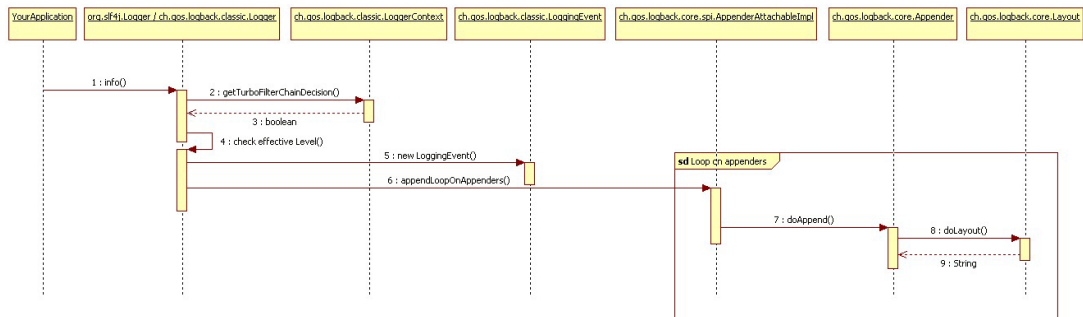
### 5. 格式化输出

那些被调用了的 appender 负责对记录事件（LoggingEvent）进行格式化。然而，有些但不是全部 appender 把格式化记录事件的工作委托给 layout。Layout 对 LoggingEvent 实例进行格式化，然后把结果以字符串的形式返回。注意有些 appender，比如 SocketAppender，把记录事件进行序列化而不是转换成字符串，所以它们不需要也没有 layout。

### 6. 发送记录事件（LoggingEvent）

记录事件被格式化后，被各个 appender 发送到各自的目的地。

下图是整个流程的 UML 图。



## 2.2.9. 性能

一个关于记录的常见争论是它的计算代价。这种关心很合理，因为即使是中等大小的应用程序也会生成数以千计的记录请求。人们花了很多精力来测算和调整记录性能。尽管如此，用户还是需要注意下面的性能问题。

### 1. 记录被彻底关闭时的记录性能

你可以将根 logger 的级别设为最高级的 Level.OFF，就可以彻底关闭记录。当记录被彻底关闭时，记录请求的消耗包括一次方法调用和一次整数比较。在 CPU 为 3.2Ghz 的 Pentium D 电脑上，一般需要 20 纳秒。

但是，任何方法调用都会涉及“隐藏的”参数构造消耗，例如，对于 logger x，

```
x.debug("Entry number: " + i + "is " + entry[i]);
```

把整数 i 和 entry[i] 都转换为字符串和连接各字符串会造成消息参数构造消耗，不管消息是否被记录。

参数构造消耗可以变得非常高，同时也跟参数大小有关。利用 SLF4J 的参数化记录可以避免这种消耗。

```
x.debug("Entry number: {} is {}", i, entry[i]);
```

这种方式不会造成参数构造消耗。与前面的 debug() 方法相比，这种方法快得多。只有当请求在被发送给 appender 时，消息才会被格式化。在格式化的时候，负责格式化消息的组件性能很高，不会对整个过程造成负面影响。格式化 1 个和 3 个参数分别需要 2 和 4 微妙。

请注意，无论如何，应当避免在紧密循环里或者非常频繁地调用记录语句，因为很

可能降低性能。即使记录被禁用，在紧密循环里作记录仍然会拖慢应用程序，如果记录被启用，就会产生大量（也是无用的）输出。

## 2. 当记录启用时，判断是否进行记录的性能

在 logback 中，logger 在被创建时就明确地知道其有效级别（已经考虑了级别继承）。当父 logger 的级别改变时，所有子 logger 都会得知这个改变。因此，在根据有效级别去接受或拒绝请求之前，logger 能够作出准即时判断，不需要咨询其祖先。

## 3. 实际记录（格式化和写入输出设备）

性能消耗包括格式化输出和发送到目的地。我们努力使 layout（formatter）和 appender 都尽可能地快。记录到本地机器的文件里的耗时一般大约在 9 至 12 微秒。如果目的地是远程服务器上的数据库时，会增加早几个毫秒。

尽管功能丰富，logback 最首要的一项设计目标就是执行速度，重要程度仅排在可靠性之后。为提高性能，logback 的一些组件已经被多次重写。

## 3. 配置

在第一部分，我们将介绍配置 logback 的各种方法，给出了很多配置脚本例子。在第二部分，我们将介绍 Joran，它是一个通用配置框架，你可以在自己的项目里使用 Joran。

### 3.1. Logback 里的配置

把记录请求插入程序代码需要相当多的计划和努力。有观察显示大约 4% 的代码是记录。所以即使是一个中等规模的应用程序也会包含数以千计的记录语句。考虑到数量庞大，我们需要使用工具来管理记录语句。

Logback 可以通过编程式配置，或用 XML 格式的配置文件进行配置。

Logback 采取下面的步骤进行自我配置：

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 BasicConfigurator 自动对自己进行配置，这会导致记录输出到控制台。

第三步也是最后一步是为了在缺少配置文件时提供默认（但基本的）记录功能。

### 3.2. 自动配置

最简单的配置方法就是使用默认配置。

BasicConfigurator 用法的简单例子：

((logback-examples/src/main/java/chapters/configuration/MyApp1.java))

```
package manual.configuration;

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

public class MyApp1 {
    final static Logger logger = LoggerFactory.getLogger(MyApp1.class);

    public static void main(String[] args) {
        logger.info("Entering application.");
    }
}
```

```
    Foo foo = new Foo();
    foo.doIt();

    logger.info("Exiting application.");
}
}
```

该类定义了一个静态变量 `logger`，然后实例化一个 `Foo` 对象。`Foo` 类如下 ((logback-examples/src/main/java/chapters/configuration/Foo.java)):

```
package manual.configuration;

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

public class Foo {
    static final Logger logger = LoggerFactory.getLogger(Foo.class);

    public void doIt() {
        logger.debug("Did it again!");
    }
}
```

假设配置文件 `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。还有，根 `logger` 默认级别是 `DEBUG`。

因此，`chapters.configuration.MyApp1` 运行后的输出应当类似于：

```
16:06:09.031 [main] INFO chapters.configuration.MyApp1 - Entering application.
16:06:09.046 [main] DEBUG chapters.configuration.Foo - Did it again!
16:06:09.046 [main] INFO chapters.configuration.MyApp1 - Exiting application.
```

`MyApp1` 程序通过调用 `org.slf4j.LoggerFactory` 类和 `org.slf4j.Logger` 类连接到 `logback`，取得想要的 `logger`，然后继续。注意 `Foo` 类对 `logback` 唯一的依赖是通过引入 `org.slf4j.LoggerFactory` 和 `org.slf4j.Logger`。



### 3.3. 用 logback-test.xml 或 logback.xml 自动配置

前面提到过，如果 classpath 里有 logback-test.xml 或 logback.xml，logback 会试图用它进行自我配置。下面的配置文件与刚才的 BasicConfigurator 等效。

示例：基本配置文件(logback-examples/src/main/java/chapters/configuration/sample0.xml)

```
<configuration>

  <appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
    <!--
        encoders are assigned the type
        ch.qos.logback.classic.encoder.PatternLayoutEncoder by
default
    -->
    <encoder>
        <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36}
- %msg%n</pattern>
    </encoder>
  </appender>

  <root level="debug">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

把 sample0.xml 重命名为 logback.xml 或 logback-test.xml，放到 classpath 里，运行后会和上例的输出几乎一样。

### 3.4. 自动打印警告和错误消息

当解析配置文件有警告或出错时，logback 会在控制台上自动打印状态数据。如果没有警告或错误，你还是想检查 logback 的内部状态的话，可以调用 StatusPrinter 的 print()方法。

MyApp2 程序等价于 MyApp1，只是多了两行打印内部状态数据的代码。

示例：打印 logback 的内部状态信息

(logback-examples/src/main/java/chapters/configuration/MyApp2.java)

```
public static void main(String[] args) {
    // assume SLF4J is bound to logback in the current environment
    LoggerContext lc = (LoggerContext)
```

```
LoggerFactory.getILoggerFactory();  
    // print logback's internal status  
    StatusPrinter.print(lc);  
  
    logger.info("Entering application.");  
    Foo foo = new Foo();  
    foo.doIt();  
    logger.info("Exiting application.");  
}
```

如果一切顺利，控制台上会输出如下：

```
17:44:58,578 |-INFO in ch.qos.logback.classic.LoggerContext[default] - Found resource  
[logback-test.xml]  
17:44:58,671 |-INFO in ch.qos.logback.classic.joran.action.ConfigurationAction - debug  
attribute not set  
17:44:58,671 |-INFO in ch.qos.logback.core.joran.action.AppenderAction - About to  
instantiate appender of type [ch.qos.logback.core.ConsoleAppender]  
17:44:58,687 |-INFO in ch.qos.logback.core.joran.action.AppenderAction - Naming appender  
as [STDOUT]  
17:44:58,812 |-INFO in ch.qos.logback.core.joran.action.AppenderAction - Popping  
appender named [STDOUT] from the object stack  
17:44:58,812 |-INFO in ch.qos.logback.classic.joran.action.LevelAction - root level set to  
DEBUG  
17:44:58,812 |-INFO in ch.qos.logback.core.joran.action.AppenderRefAction - Attaching  
appender named [STDOUT] to Logger[root]  
  
17:44:58.828 [main] INFO chapters.configuration.MyApp2 - Entering application.  
17:44:58.828 [main] DEBUG chapters.configuration.Foo - Did it again!  
17:44:58.828 [main] INFO chapters.configuration.MyApp2 - Exiting application.
```

在输出的最后面，你可以看到上例输出的内容。你也应当注意到 logback 的内部消息，也就是 Status 对象，它可以方便地访问 logback 的内部状态。

可以不用从代码里调用 StatusPrinter，而是在配置文件里进行相关配置，即使没有出现错误。方法是，设置 configuration 元素的 debug 属性为 true。请注意 debug 属性只与状态数据有关，它不影响 logback 的配置，更不会影响记录级别。

示例：debug 模式的基本配置

(logback-examples/src/main/java/chapters/configuration/sample1.xml)

```
<configuration debug="true">

  <appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
    <!--
        encoders are assigned by default the type
        ch.qos.logback.classic.encoder.PatternLayoutEncoder
    -->
    <encoder>
        <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36}
- %msg%n
        </pattern>
    </encoder>
  </appender>

  <root level="debug">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

把 configuration 元素的 debug 属性设为 true 后，会输出状态信息，但是前提是：

1. 找到了配置文件；
2. 配置文件是格式化良好的 XML。

如果其中任一条件未满足，Joran 就会因为配置文件不可读而无法读取 debug 属性。如果找到了配置文件，但却不是格式化良好的，那么 logback 会检测出错误并把内部状态打印到控制台。然而，如果找不到配置文件，由于这不是个严重的错误，logback 不会自动打印状态数据。使用程式的主动调用 `StatusPrinter.print()` 可以确保始终打印状态信息，如 MyApp2。

### 3.5. 把默认配置文件的位置作为系统属性进行指定

设置名为 `logback.configurationFile` 的系统属性，把默认配置文件的位置作为属性值，这种方法也可以。属性值即配置文件位置可以是 URL、classpath 里的一个资源，或者是程序外部的文件路径。

```
java -Dlogback.configurationFile=/path/to/config.xml chapters.configuration.MyApp1
```

### 3.6. 配置文件修改后自动重新加载

如果设置成自动重新加载，logback-classic 会扫描配置文件里的变化，并且当发生变化后进行重新配置。设置方法是设 configuration 元素的 scan 属性为 true。

示例：扫描配置文件的变化并自动重新配置

(logback-examples/src/main/java/chapters/configuration/scan1.xml)

```
<configuration scan="true">
...
</configuration>
```

默认情况下，每隔一分钟扫描一次。configuration 元素的 scanPeriod 属性控制扫描周期，其值可以带时间单位，包括：milliseconds、seconds、minutes 和 hours。

示例：指定不同的扫描周期

(logback-examples/src/main/java/chapters/configuration/scan2.xml)

```
<configuration scan="true" scanPeriod="30 seconds">
...
</configuration>
```

如果没写明时间单位，则默认为毫秒。

内部实现是这样的，当设置扫描属性为 true 时，会安装一个叫 ReconfigureOnChangeFilter 的 TurboFilter。每次调用 logger 的打印方法时，都会进行扫描。比如，当名为 myLogger 的 logger 执行“myLogger.debug("hello");”时，如果 scan 属性为 true，则 ReconfigureOnChangeFilter 会被调用。而且，即使 myLogger 的 debug 级别被禁用了，仍然会调用上述过滤器。

考虑到在任何 logger 在每次被调用时都要调用 ReconfigureOnChangeFilter，这个过滤器的性能就变得十分关键了。为提高性能，不会在每个 logger 被调用时去检查是否需要扫描，而是每隔 16 次记录操作进行一次检查。简言之，当配置文件改变后，它会被延时重新加载，延时时间由扫描间隔时间和一些 logger 调用所决定。

### 3.7. 直接调用 JoranConfigurator

Logback 依赖 Joran，Joran 是 logback-core 的一部分，是个配置类库。Logback 的默认配

置机制是调用 `JoranConfigurator` 对 `classpath` 上的默认配置文件进行处理。不管出于什么理由，如果你想重新实现 `logback` 的默认配置机制的话，你可以直接调用 `JoranConfigurator`。下面的程序 `MyApp3` 就调用了 `JoranConfigurator` 对作为参数传入的配置文件进行处理。

示例：直接调用 `JoranConfigurator`

(`logback-examples/src/main/java/chapters/configuration/MyApp3.java`)

```
package chapters.configuration;

/**
 * Demonstrates programmatic invocation of Joran.
 *
 */
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import ch.qos.logback.classic.LoggerContext;
import ch.qos.logback.classic.joran.JoranConfigurator;
import ch.qos.logback.core.joran.spi.JoranException;
import ch.qos.logback.core.util.StatusPrinter;

public class MyApp3 {
    final static Logger logger = LoggerFactory.getLogger(MyApp3.class);

    public static void main(String[] args) {
        // assume SLF4J is bound to logback in the current environment
        LoggerContext lc = (LoggerContext)
        LoggerFactory.getILoggerFactory();

        try {
            JoranConfigurator configurator = new JoranConfigurator();
            configurator.setContext(lc);
            // the context was probably already configured by default
            // configuration rules
            lc.reset();
            configurator.doConfigure(args[0]);
        } catch (JoranException je) {
            // StatusPrinter will handle this
        }
        StatusPrinter.printInCaseOfErrorsOrWarnings(lc);

        logger.info("Entering application.");

        Foo foo = new Foo();
    }
}
```

```

    foo.doIt();
    logger.info("Exiting application.");
}
}

```

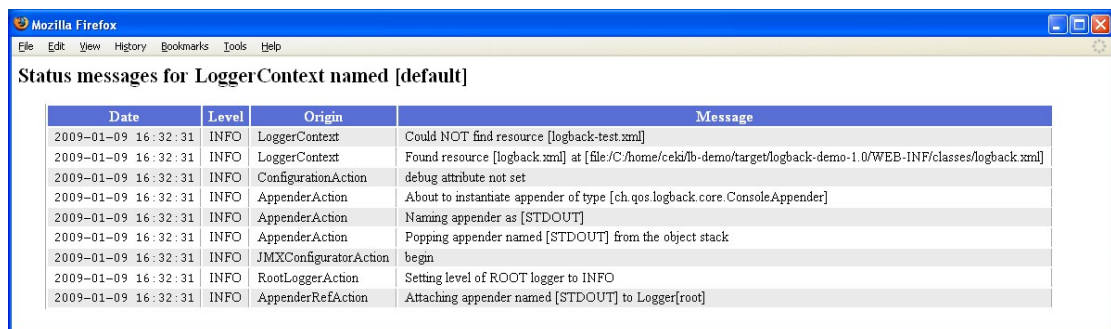
本程序直接取得 `LoggerContext`，创建新 `JoranConfigurator` 并设置它要操作的上下文，重置 `logger` 上下文，最后要求配置器用参数中的配置文件对上下文进行配置。同时打印了内部状态数据。

### 3.8. 查看状态消息

Logback 把内部数据放在一个 `StatusManager` 对象里，并通过 `LoggerContext` 访问。

`StatusManager` 通过 `logback` 上下文来访问所有数据对象。为把内存占用保持在合理的范围内，默认的 `StatusManager` 实现将状态消息按头和尾两部分存储。头部存储开始的 `H` 条状态消息，尾部存储后面的 `T` 条消息。现在的 `H=T=150`，将来或许会改变。

Logback-classic 带了一个叫 `ViewStatusMessagesServlet` 的 `Servlet`，它以 HTML 表格的格式打印与当前 `LoggerContext` 关联的 `StatusManager` 的内容。示例如下。



Date	Level	Origin	Message
2009-01-09 16:32:31	INFO	LoggerContext	Could NOT find resource [logback-test.xml]
2009-01-09 16:32:31	INFO	LoggerContext	Found resource [logback.xml] at [file/C:/home/ceki/lb-demo/target/logback-demo-1.0/WEB-INF/classes/logback.xml]
2009-01-09 16:32:31	INFO	ConfigurationAction	debug attribute not set
2009-01-09 16:32:31	INFO	AppenderAction	About to instantiate appender of type [ch.qos.logback.core.ConsoleAppender]
2009-01-09 16:32:31	INFO	AppenderAction	Naming appender as [STDOUT]
2009-01-09 16:32:31	INFO	AppenderAction	Popping appender named [STDOUT] from the object stack
2009-01-09 16:32:31	INFO	JMXConfiguratorAction	begin
2009-01-09 16:32:31	INFO	RootLoggerAction	Setting level of ROOT logger to INFO
2009-01-09 16:32:31	INFO	AppenderRefAction	Attaching appender named [STDOUT] to Logger[root]

要加到自己的 web 应用程序里，可以在 `WEB-INF/web.xml` 里添加如下内容：

```

<servlet>
  <servlet-name>ViewStatusMessages</servlet-name>
  <servlet-class>ch.qos.logback.classic.ViewStatusMessagesServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>ViewStatusMessages</servlet-name>
  <url-pattern>/lbClassicStatus</url-pattern>
</servlet-mapping>

```

```
</servlet-mapping>
```

访问地址是 `http://host/yourWebapp/lbClassicStatus`。

### 3.9. 监听状态消息

你也可以为 `StatusManager` 附加一个 `StatusListener`，这样就能立即对状态消息作出响应，尤其对那些 `logback` 配置完成之后的消息。注册一个状态监听器可以方便地实现对 `logback` 内部状态的无人监管。

`Logback` 带了一个叫 `OnConsoleStatusListener` 的 `StatusListener` 实现，可以把状态消息打印到控制台。

下例演示了如何为 `StatusManager` 注册一个 `OnConsoleStatusListener` 实例。

```
LoggerContext lc = (LoggerContext) LoggerFactory.getILoggerFactory();
StatusManager statusManager = lc.getStatusManager();
OnConsoleStatusListener onConsoleListener = new
OnConsoleStatusListener();
statusManager.add(onConsoleListener);
```

注意注册了的状态监听器只会接收被注册之后的状态消息，不会注册之前的消息。

也可以在配置文件里注册一个或多个状态监听器。如下面的例子。

示例：注册状态监听器

(`logback-examples/src/main/java/chapters/configuration/onConsoleStatusListener.xml`)

```
<configuration>

  <statusListener
class="ch.qos.logback.core.status.OnConsoleStatusListener" />
... the rest of the configuration file
</configuration>
```

还可以通过设置 Java 系统属性 “`logback.statusListenerClass`” 注册状态监听器，例如，

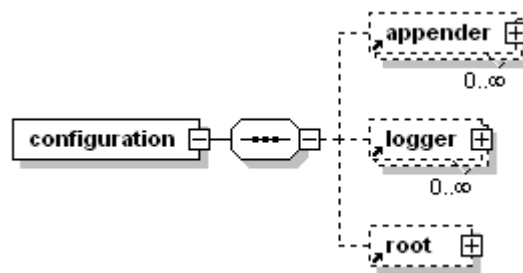
```
java -Dlogback.statusListenerClass=ch.qos.logback.core.status.OnConsoleStatusListener ...
```

## 3.10. 配置文件语法

到目前为止，正如你已经在这份手册里看到的不少例子，logback 允许你重新定义记录行为而不必重新编译你的代码。实际上，你可以轻易地配置 logback，比如禁用程序里某些地方的记录功能，或者直接输出到一个 UNIX 系统守护进程、数据库、日志查看器，或把记录事件发送到远程 logback 服务器，远程 logback 服务器按照其本地策略进行记录，比如把记录时间发送到第二个 logback 服务器。

本节剩余部分介绍配置文件的语法。

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。如下图所示：



### 3.10.1. 标记名大小写敏感性

从 logback 0.9.17 版起，标记名不区分大小写。比如，<logger>、<Logger>和<LOGGER>都是合法元素且表示同一个意思。按照隐式规则，标记名除了首字母外要区分大小写。因此，<xyz>与<Xyz>等价，但不等价于<xYz>。隐式规则一般遵循 Java 世界里常用的驼峰命名规则。因为很难确定一个标记什么时候与显式动作相关，什么时候又与隐式动作相关，所以很难说 XML 标记是否是大小写敏感。如果你不确定标记名的大小写，就用驼峰命名法，基本不会错。

### 3.10.2. 配置 logger，或<logger>元素

Logger 是用<logger>元素配置的。<logger>元素有且仅有一个 name 属性、一个可选的 level 属性和一个可选的 additivity 属性。

Level 属性的值大小写无关，其值为下面其中一个字符串：TRACE、DEBUG、INFO、WARN、ERROR、ALL 和 OFF。还可以是一个特殊的字符串“INHERITED”或其同义词



“NULL”，表示强制继承上级的级别。

<logger>元素可以包含零个或多个<appender-ref>元素，表示这个 appender 会被添加到该 logger。强调一下，每个用<logger>元素声明的 logger，首先会移除所有 appender，然后才添加引用了的 appender，所以如果 logger 没有引用任何 appender，就会失去所有 appender。

### 3.10.3. 配置根 logger，或<root>元素

<root>元素配置根 logger。该元素有一个 level 属性。没有 name 属性，因为已经被命名为“ROOT”。

Level 属性的值大小写无关，其值为下面其中一个字符串：TRACE、DEBUG、INFO、WARN、ERROR、ALL 和 OFF。注意不能设置为“INHERITED” 或“NULL”。

<logger>元素可以包含零个或多个<appender-ref>元素。与<logger>元素类似，声明<root>元素后，会先关闭然后移除全部当前 appender，只引用声明了的 appender。如果 root 元素没有引用任何 appender，就会失去所有 appender。

### 3.10.4. 示例

假设我们不想看到“chapters.configuration”包里的任何组件的任何 DEBUG 信息，可以设置如下：

示例：设置 logger 级别

(logback-examples/src/main/java/chapters/configuration/sample2.xml)

```
<configuration>

  <appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
    <!--
        encoders are assigned by default the type
        ch.qos.logback.classic.encoder.PatternLayoutEncoder
    -->
    <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.      -->
<root level="DEBUG">
  <appender-ref ref="STDOUT" />
</root>

</configuration>
```

对于 MyApp3 应用上述配置文件后，输出如下：

```
17:34:07.578 [main] INFO  chapters.configuration.MyApp3 - Entering application.
17:34:07.578 [main] INFO  chapters.configuration.MyApp3 - Exiting application.
```

注意由名为“chapters.configuration.Foo”的 logger 生成的 DEBUG 级别的信息都被屏蔽了。

你可以为任意数量的 logger 设置级别。下面的配置文件里，我们为 logger “chapters.configuration” 设置级别为 INFO，同时设置 logger “chapters.configuration.Foo” 级别为 DEBUG。

示例：设置多个 logger 的级别

```
<configuration>

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

    <!-- encoders are assigned by default the type
         ch.qos.logback.classic.encoder.PatternLayoutEncoder -->
    <encoder>
      <pattern>
        %d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
      </pattern>
    </encoder>
  </appender>

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

  <logger name="chapters.configuration.Foo" level="DEBUG" />

  <root level="DEBUG">
    <appender-ref ref="STDOUT" />
```

```
</root>

</configuration>
```

对于 MyApp3 应用上述配置文件后，输出如下：

```
17:39:27.593 [main] INFO  chapters.configuration.MyApp3 - Entering application.
17:39:27.593 [main] DEBUG chapters.configuration.Foo - Did it again!
17:39:27.593 [main] INFO  chapters.configuration.MyApp3 - Exiting application.
```

经 JoranConfigurator 用 sample3.xml 对 logback 进行配置后，各 logger 和各自级别如下表所示：

Logger 名	分配级别	有效级别
root	DEBUG	DEBUG
chapters.configuration	INFO	INFO
chapters.configuration.MyApp3	null	INFO
chapters.configuration.Foo	DEBUG	DEBUG

MyApp3 类的 INFO 级别的记录语句和 Foo.doIt() 的 DEBUG 消息都被启用。注意根 logger 总是设为非 null 值，默认为 DEBUG。

注意，基本选择规则依赖于被调用的 logger 的有效级别，而不是 appender 所关联的 logger 的级别。Logback 首先判断记录语句是否被启用，如果启用，则调用 logger 等级里的 appender，无视 logger 的级别。配置文件 sample4.xml 演示了这一点。

示例：logger 级别(logback-examples/src/main/java/chapters/configuration/sample4.xml)

```
<configuration>

  <appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
    <!--
      encoders are assigned by default the type
      ch.qos.logback.classic.encoder.PatternLayoutEncoder
    -->
    <encoder>
      <pattern>
        %d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
      </pattern>
```

```

    </encoder>
  </appender>

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

  <root level="OFF">
    <appender-ref ref="STDOUT" />
  </root>

</configuration>

```

用 sample4.xml 对 logback 进行配置后，各 logger 和各自级别如下表所示：

Logger 名	分配级别	有效级别
root	OFF	OFF
chapters.configuration	INFO	INFO
chapters.configuration.MyApp3	null	INFO
chapters.configuration.Foo	null	INFO

配置里唯一的 appender“STDOUT”，被关联到级别为 OFF 的根 logger，但是运行 MyApp3 后会输出：

```

17:52:23.609 [main] INFO chapters.configuration.MyApp3 - Entering application.
17:52:23.609 [main] INFO chapters.configuration.MyApp3 - Exiting application.

```

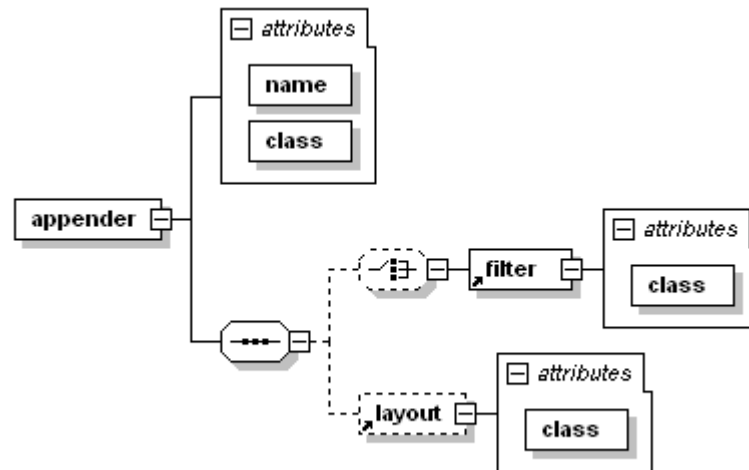
根 logger 的级别不起任何作用，是因为 chapters.configuration.MyApp3 里的 logger 和 chapters.configuration.Foo 类的 INFO 级别都是启用的。附注：logger “chapters.configuration” 即使没有任何 Java 代码直接引用它也会存在，全因为在配置文件里声明了它。

### 3.10.5. 配置 Appender

Appender 用<appender>元素配置，该元素必要属性 name 和 class。

name 属性指定 appender 的名称，class 属性指定 appender 类的全限定名。

<appender>元素可以包含零个或多个<layout>元素、零个或多个<encoder>元素和零个或多个<filter>元素。除了这三个常用元素之外，还可以包含 appender 类的任意数量的 javabean 属性。下图演示了常用结构，注意对 javabean 属性的支持在图中不可见。



<layout>元素的 `class` 属性是必要的，表示将被实例化的 `layout` 类的全限定名。和 <appender>元素一样，<layout>元素可以包含 `layout` 的 `javaBean` 属性。因为太常用了，所以当 `layout` 是 `PatternLayout` 时，可以省略 `class` 属性。

<encoder>元素 `class` 属性是必要的，表示将被实例化的 `encoder` 类的全限定名。因为太常用了，所以当 `encoder` 是 `PatternLayoutEncoder` 时，可以省略 `class` 属性。

记录输出到多个 `appender` 很简单，先定义各种 `appender`，然后在 `logger` 里进行引用，就行了。如下面的配置文件所示：

示例：多个 `logger` (logback-examples/src/main/java/chapters/configuration/multiple.xml)

```

<configuration>
  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <file>myApp.log</file>
    <!--
      encoders are assigned by default the type
      ch.qos.logback.classic.encoder.PatternLayoutEncoder
    -->
    <encoder>
      <pattern>%date %level [%thread] %logger{10}
[%file:%line] %msg%n
      </pattern>
    </encoder>
  </appender>

  <appender name="STDOUT"

```

```
class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
    <pattern>%msg%n</pattern>
  </encoder>
</appender>

<root level="debug">
  <appender-ref ref="FILE" />
  <appender-ref ref="STDOUT" />
</root>
</configuration>
```

该配置文件定义了两个 appender，分别是“FILE”和“STDOUT”。

“FILE”这个 appender 把记录输出到文件“myapp.log”，它的 encoder 是 PatternLayoutEncoder，输出了日期、级别、线程名、logger 名、文件名及记录请求的行号、消息和行分隔符。

“STDOUT”这个 appender 把记录输出到控制台，它的 encoder 只是输出消息和行分隔符。

注意每个 appender 都有自己的 encoder。Encoder 通常不能被多个 appender 共享，layout 也是。所以，logback 的配置文件里没有共享 encoder 或 layout 的语法。

### 3.10.6. Appender 累积

默认情况下，appender 是可累积的：logger 会把记录输出到它自身的 appender 和它所有祖先的 appender。因此，把同一 appender 关联到多个 logger 会导致重复输出。

示例：重复的 appender

(logback-examples/src/main/java/chapters/configuration/duplicate.xml)

```
<configuration>

  <appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>
        %d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
      </pattern>
    </encoder>
  </appender>
```

```
<logger name="chapters.configuration">
  <appender-ref ref="STDOUT" />
</logger>

<root level="debug">
  <appender-ref ref="STDOUT" />
</root>
</configuration>
```

用 duplicate.xml 配置 MyApp3 后输出如下:

```
14:25:36.343 [main] INFO chapters.configuration.MyApp3 - Entering application.
14:25:36.343 [main] INFO chapters.configuration.MyApp3 - Entering application.
14:25:36.359 [main] DEBUG chapters.configuration.Foo - Did it again!
14:25:36.359 [main] DEBUG chapters.configuration.Foo - Did it again!
14:25:36.359 [main] INFO chapters.configuration.MyApp3 - Exiting application.
14:25:36.359 [main] INFO chapters.configuration.MyApp3 - Exiting application.
```

看到重复输出了吧? 名为 STDOUT 的 appender 被关联给两个 logger, 分别是根和 chapters.configuration。由于根 logger 是所有 logger 的祖先, chapters.configuration 是 chapters.configuration.MyApp3 和 chapters.configuration.Foo 之父, 所以这两个 logger 的记录请求会被输出两次。一次是因为 STDOUT 被关联到 chapters.configuration, 一次是因为被关联到根 logger。

Appender 的叠加性对新手来说并不是陷阱, 反而是非常方便的。举例来说, 你可以让某些系统里所有 logger 的记录信息出现在控制台, 却让某些特定 logger 的记录信息发到一个特定的 appender。

示例: 多个 appender(logback-examples/src/main/java/chapters/configuration/restricted.xml)

```
<configuration>

  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <file>myApp.log</file>
    <!--
      encoders are assigned by default the type
      ch.qos.logback.classic.encoder.PatternLayoutEncoder
    -->
    <encoder>
      <pattern>%date %level [%thread] %logger{10}
```

```
[%file:%line] %msg%n
    </pattern>
  </encoder>
</appender>

<appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
    <pattern>%msg%n</pattern>
  </encoder>
</appender>

<logger name="chapters.configuration">
  <appender-ref ref="FILE" />
</logger>

<root level="debug">
  <appender-ref ref="STDOUT" />
</root>
</configuration>
```

本例中，控制台 appender 会输出所有消息（出自系统里的所有 logger），但是只有来自 chapters.configuration 这个 logger 的消息会输出到文件 myApp.log。

### 3.10.7. 覆盖默认的累积行为

如果你觉得默认的累积行为不合适，可以设置叠加性标识为 false 以关闭它。这样的话，logger 树里的某个分支可以输出到与其他 logger 不同的 appender。

示例：叠加性标识

(logback-examples/src/main/java/chapters/configuration/additivityFlag.xml)

```
<configuration>

  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <file>foo.log</file>
    <encoder>
      <Pattern>
        %date %level [%thread] %logger{10} [%file : %line] %msg%n
      </Pattern>
    </encoder>
  </appender>
```



```
<appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
    <Pattern>%msg%n</Pattern>
  </encoder>
</appender>

<logger name="chapters.configuration.Foo" additivity="false">
  <appender-ref ref="FILE" />
</logger>

<root level="debug">
  <appender-ref ref="STDOUT" />
</root>
</configuration>
```

本例中，logger “chapters.configuration.Foo” 关联 appender “FILE”，它的叠加性标记为 false，这样它的记录输出仅会被发送到 appender “FILE”，不会被发送到更高 logger 等级关联的 appender。其他 logger 不受此影响。

用 additivityFlag.xml 配置 MyApp3，运行后，控制台上由输出由 “chapters.configuration.MyApp3” 产生的记录。而 logger “chapters.configuration.Foo” 将且仅仅将输出到文件 foo.log。

### 3.10.8. 设置上下文名称

每个 logger 都关联到 logger 上下文。默认情况下，logger 上下文名为 “default”。但是你可以借助配置指令 <contextName> 设置成其他名字。注意一旦设置 logger 上下文名称后，不能再改。设置上下文名称后，可以方便地区分来自不同应用程序的记录。

示例：设置上下文名称并显示它

(logback-examples/src/main/java/chapters/configuration/contextName.xml)

```
<configuration>

  <contextName>myAppName</contextName>

  <appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <Pattern>%d %contextName [%t] %level %logger{36}
- %msg%n</Pattern>
    </encoder>
```

```
</appender>

<root level="debug">
  <appender-ref ref="STDOUT" />
</root>

</configuration>
```

本例演示了 logger 上下文的命名方法：在 layout 模式里添加 “%contextName” 就会输出上下文的名称。

### 3.10.9. 变量替换

原则上，指定变量的地方就能够发生变量替换。变量替换的语法与 Unix shell 中的变量替换相似。位于 “\${” 与 “}” 之间的字符串是键（key），取代键的值可以在同一配置文件里指定，也可以在外部文件或通过系统属性进行指定。例如，如果设系统属性 “java.home.dir” 为 “/home/xyz”，那么每次当 \${java.home.dir} 出现时都会被解释为 “/home/xyz”。Logback 自动定义了一个常用变量 “\${HOSTNAME}”。

#### 3.10.9.1. 属性被插入 logger 上下文

注意通过 <property> 元素定义的值实际上会被插入 logger 上下文。换句话说，这些值变成了 logger 上下文的属性。所以，它们对所有记录事件都可用，包括通过序列化方式被发送到远程主机的记录事件。

下面的例子在配置文件的开头声明了一个变量又名替换属性，它代表输出文件的位置，然后在后面的配置文件里使用它。

示例：简单变量替换

(logback-examples/src/main/java/chapters/configuration/variableSubstitution1.xml)

```
<configuration>

  <property name="USER_HOME" value="/home/sebastien" />

  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <file>${USER_HOME}/myApp.log</file>
    <encoder>
      <pattern>%msg%n</pattern>
    </encoder>
  </appender>
```

```
<root level="debug">
  <appender-ref ref="FILE" />
</root>
</configuration>
```

下一个例子用系统属性实现了同样的功能。属性没有在配置文件里声明，因此 logback 会从系统属性里找。Java 系统属性用下面的命令行进行设置：

```
java -DUSER_HOME="/home/sebastien" MyApp2
```

示例：系统变量替换

(logback-examples/src/main/java/chapters/configuration/variableSubstitution2.xml)

```
<configuration>

  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <file>${USER_HOME}/myApp.log</file>
    <encoder>
      <pattern>%msg%n</pattern>
    </encoder>
  </appender>

  <root level="debug">
    <appender-ref ref="FILE" />
  </root>
</configuration>
```

当需要很多变量时，更方便的做法是在一个单独的文件里声明所有变量，如下例所示。

示例：文件变量替换

(logback-examples/src/main/java/chapters/configuration/variableSubstitution3.xml)

```
<configuration>

  <property
    file="src/main/java/chapters/configuration/variables1.properties" />

  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <file>${USER_HOME}/myApp.log</file>
    <encoder>
      <pattern>%msg%n</pattern>
    </encoder>
  </appender>
</configuration>
```

```
</appender>

<root level="debug">
  <appender-ref ref="FILE" />
</root>
</configuration>
```

这个配置文件包含对文件 “variables1.properties” 的引用，该文件里的变量会被读入 logback 配置文件的上下文里。

文件 “variables1.properties” 内容类似于：

示例：变量文件

(logback-examples/src/main/java/chapters/configuration/variables1.properties)

```
USER_HOME=/home/sebastien
```

还可以不引用文件，而是引用 class path 上的资源。

```
<configuration>

  <property resource="resource1.properties" />

  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <file>${USER_HOME}/myApp.log</file>
    <encoder>
      <pattern>%msg%n</pattern>
    </encoder>
  </appender>

  <root level="debug">
    <appender-ref ref="FILE" />
  </root>
</configuration>
```

### 3.10.9.2. 嵌套变量替换

Logback 支持嵌套变量替换。这里的嵌套是指变量的值里包含对其他变量的引用。假设你希望用变量指定目的地目录和文件名，然后用一个变量 “destination” 组合这两个变量，如下面所示。

示例：嵌套变量替换

(logback-examples/src/main/java/chapters/configuration/variables2.properties)

```
USER_HOME=/home/sebastien
fileName=myApp.log
destination=${USER_HOME}/${fileName}
```

注意在上面的属性文件里,“destination”由另外两个变量“USER\_HOME”和“fileName”组合而成。

示例: 文件变量替换

(logback-examples/src/main/java/chapters/configuration/variableSubstitution4.xml)

```
<configuration>

  <property
file="src/main/java/chapters/configuration/variables2.properties" />

  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <file>${destination}</file>
    <encoder>
      <pattern>%msg%n</pattern>
    </encoder>
  </appender>

  <root level="debug">
    <appender-ref ref="FILE" />
  </root>
</configuration>
```

### 3.10.9.3. 变量的默认替换值

在某些特定情况下,最好给变量一个默认值,以免变量未被声明或值为 null。Bash shell 用 “:-” 指定默认值。例如,假设 “aKey” 未被声明,那么 “\${aKey:-golden}” 将被解释为 “golden”。

### 3.10.9.4. HOSTNAME 属性

HOSTNAME 属性因为很常用,所以在配置过程中被自动定义。

### 3.10.9.5. 设置时间戳

元素 timestamp 可以定义表示一个当前日期和时间的属性。

### 3.10.9.6. 在运行中定义属性

可以配置文件里用`<define>`元素定义属性。`<define>`元素有两个必要属性：`name` 和 `class`。`name` 属性代表属性的名称，`class` 属性代表 `PropertyDefiner` 接口的任意实现。`PropertyDefiner` 接口的 `getPropertyValue()`方法返回的值就是属性值。

如下例。

```
<configuration>

  <define name="rootLevel"
class="a.class.implementing.PropertyDefiner">
    <aProperty>of a.class.implementing.PropertyDefiner</aProperty>
  </define>

  <root level="${rootLevel}" />
</configuration>
```

Logback 目前还没有提供 `PropertyDefiner` 接口的具体实现，只是为动态定义属性提供了一种方法。

### 3.10.9.7. 配置文件里的条件化处理

开发者经常需要针对不同的环境在不同的配置文件里换来换去，比如开发、测试和生产环境。这些配置文件大同小异。为避免重复劳动，logback 支持在配置文件里进行条件化处理，用`<if>`、`<then>`和`<else>`这些元素可以让一个配置文件适用于多个环境。

条件语句一般格式如下。

```
<configuration>

  <!-- if-then form -->
  <if condition="some conditional expression">
    <then>
      ...
    </then>
  </if>

  <!-- if-then-else form -->
  <if condition="some conditional expression">
    <then>
      ...
    </then>
    <else>
      ...
    </else>
  </if>
```

```
        </else>
      </if>

</configuration>
```

其中“condition”是 java 表达式，只允许访问上下文属性和系统属性。对于作为参数传入的键，property()方法或其等价的 p()方法将返回属性的字符串值。例如，想访问属性键为“k”的值，你可以用 property("k")或等价的 p("k")。如果键为“k”的属性未被定义，property 方法将返回空字符串而不是 null，这样避免了检查 null 值。

下一个例子里，ConsoleAppender 被关联到根 logger，但是前提条件是 HOSTNAME 属性的值是“torino”。注意名为“FILE”的 FileAppender 在任何情况下都被关联到根 logger。

```
<configuration>

  <if condition='property("HOSTNAME").contains("torino") '>
    <then>
      <appender name="CON"
class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
          <pattern>%d %-5level %logger{35} - %msg %n</pattern>
        </encoder>
      </appender>
      <root>
        <appender-ref ref="CON" />
      </root>
    </then>
  </if>

  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <file>${randomOutputDir}/conditional.log</file>
    <encoder>
      <pattern>%d %-5level %logger{35} - %msg %n</pattern>
    </encoder>
  </appender>

  <root level="ERROR">
    <appender-ref ref="FILE" />
  </root>
</configuration>
```

<configuration>元素之内的任何地方都支持条件化处理。也支持嵌套的 if-then-else 语句。然而，加入过多的条件语句会导致 XML 文件非常难读。

### 3.10.9.8. 从 JNDI 获取变量

在某些特定情况下,你也许利用 JNDI 里存储的 env 项,<insertFromJNDI>指令会从 JNDI 里取得 env 项,然后用 as 属性把它们作为变量。

示例: 通过 JNDI 取得 env 项并作为属性插入

(logback-examples/src/main/java/chapters/configuration/insertFromJNDI.xml)

```
<configuration>
  <insertFromJNDI env-entry-name="java:comp/env/appName"
    as="appName" />
  <contextName>${appName}</contextName>

  <appender name="CONSOLE"
class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d %contextName %level %msg %logger{50}%n</pattern>
    </encoder>
  </appender>

  <root level="DEBUG">
    <appender-ref ref="CONSOLE" />
  </root>
</configuration>
```

本例中, env 项 “java:comp/env/appName” 被作为 “appName” 属性插入到配置文件。注意先是<insertFromJNDI>指令插入的 “appName” 属性, 然后<contextName>指令把 “appName” 的值设成 “contextName”。

### 3.10.9.9. 文件包含

Joran 支持在配置文件里包含其他文件。方法是声明<include>元素, 如下所示:

示例: 文件包含

(logback-examples/src/main/java/chapters/configuration/containingConfig.xml)

```
<configuration>
  <include
file="src/main/java/chapters/configuration/includedConfig.xml" />

  <root level="DEBUG">
    <appender-ref ref="includedConsole" />
  </root>
```



```
</configuration>
```

被包含的文件必须把它的元素嵌套在<included>元素里。例如，可以这样声明 ConsoleAppender:

示例：文件包含

(logback-examples/src/main/java/chapters/configuration/includedConfig.xml)

```
<included>
  <appender name="includedConsole"
    class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>"%d - %m%n"</pattern>
    </encoder>
  </appender>
</included>
```

请再次注意<included>元素是必需的。

被包含的内容可以是文件、资源或 URL。

- 作为文件

用“file”属性包含一个文件。可以用相对路径，但是需要注意，当前目录是由应用程序决定的，与配置文件的路径必要的联系。

- 作为资源

用“resource”属性包含一个资源，也就是在 class path 上的文件。

```
<include resource="includedConfig.xml" />
```

- 作为 URL

用“url”属性包括一个 URL。

```
<include url="http://some.host.com/includedConfig.xml" />
```

## 4. Appender

### 4.1. 什么是 Appender

Appender 是负责写记录事件的组件。Appender 必须实现接口“ch.qos.logback.core.Appender”。该接口的重要方法总结如下：

```
package ch.qos.logback.core;

import ch.qos.logback.core.spi.ContextAware;
import ch.qos.logback.core.spi.FilterAttachable;
import ch.qos.logback.core.spi.Lifecycle;

public interface Appender<E> extends Lifecycle, ContextAware,
    FilterAttachable {

    public String getName();

    public void setName(String name);

    void doAppend(E event);

}
```

Appender 接口里的多数方法都是 getter 和 setting。值得注意的是 doAppend() 方法，它唯一的参数是类型 E 的对象。类型 E 的实际类型视 logback 模块的不同而不同。在 logback-classic 模块里，E 可能是“ILoggingEvent”类型；在 logback-access 模块里，E 可能是“AccessEvent”类型。doAppend() 方法也许是 logback 框架里最重要的方法，它负责以适当的格式将记录事件输出到合适的设备。

Appender 是被命名的实体。因为有名字，所以能被引用。Appender 接口扩展了 FilterAttachable 接口，因此 appender 实例可被关联一个或多个过滤器。

Appender 是最终负责输出记录事件的组件。然而，它们可以把实际格式化的任务委托给 Layout 或 Encoder 对象。每个 layout/encoder 都关联到一个且仅一个 appender。有些 appender 有内置的或固定的事件格式，因此它们不需要也没有 layout/encoder。例如，SocketAppender 在发送记录事件之前只是简单地对其进行序列化。

## 4.2. AppenderBase

类 `ch.qos.logback.core.AppenderBase` 是实现了 `Appender` 接口的抽象类。`AppenderBase` 提供所有 `appender` 共享的基本功能，比如设置/获取名字的方法，其活动状态和过滤器。`AppenderBase` 是 `logback` 里所有 `appender` 的超类。尽管是抽象类，`AppenderBase` 实际上实现了 `Appender` 接口的 `doAppend()` 方法。代码更能说明问题：

```
public synchronized void doAppend(E eventObject) {

    // prevent re-entry.
    if (guard) {
        return;
    }

    try {
        guard = true;

        if (!this.started) {
            if (statusRepeatCount++ < ALLOWED_REPEATS) {
                addStatus(new WarnStatus(
                    "Attempted to append to non started appender ["
                        + name + "].", this));
            }
            return;
        }

        if (getFilterChainDecision(eventObject) == FilterReply.DENY)
        {
            return;
        }

        // ok, we now invoke derived class' implementation of append
        this.append(eventObject);

    } catch (Exception e) {
        if (exceptionCount++ < ALLOWED_REPEATS) {
            addError("Appender [" + name + "] failed to append.", e);
        }
    } finally {
        guard = false;
    }
}
```

这里的 `doAppend()` 方法的实现是同步的，确保不同线程对同一个 `appender` 的记录是线程安全的。

这里进行的同步并不总是合适的，logback 带了与 `AppenderBase` 非常相似的类 `ch.qos.logback.core.UnsynchronizedAppenderBase`，之后会讨论它。

`doAppend()` 方法做的第一件事就是检查“guard”是否为 `true`。如果是，则立即退出方法。如果未设置“guard”，紧接下来的语句就把它设为 `true`。“guard”确保 `doAppend()` 方法不会递归调用自己。

之后的语句里，我们检查“started”字段是否为 `true`。如果不是，`doAppend()` 会发出一条警告信息然后返回。换句话说，`appender` 一旦关闭后，就无法再向它写入。`Appender` 对象实现 `LifeCycle` 接口，意味着它们实现了 `start()`、`stop()` 和 `isStarted()` 方法。对 `appender` 的所有属性都设值后，Joran 调用 `start()` 方法让 `appender` 激活自己的属性。`Appender` 也许会启动失败，比如因为找不到某些属性，或者因为不同属性之间互相引用。例如，假设文件创建是截断模式，那么，`FileAppender` 在 `Append` 选项没确定之前不能作用于 `File` 选项。这种显式激活步骤确保 `appender` 在属性值被确定之后才能使用属性。

如果 `appender` 不能启动或者已经被停止，则会通过 logback 的内部状态管理系统发出一条警告消息。尝试几次后，为了避免内部状态系统被同一条警告消息所淹没，`doAppend()` 方法将停止发出这些警告消息。

接着的 `if` 语句检查关联的过滤器的结果。根据过滤器链的判断结果，事件被拒绝或接受。如果过滤器链没有结果，则事件被默认接受。

`doAppend()` 方法然后调用派生类的 `append()` 方法，此方法真正把事件增加到合适的设备。

最后，释放 `guard`，允许下一个 `doAppend()` 调用。

在本手册中，术语“选项 (option)”或“属性 (property)”表示用 `JavaBeans` 内省机制通过 `setter` 和 `getter` 方法动态引用的属性 (attribute)。

## 4.3. Logback-core

Logback-core 是其他 logback 模块的基础。一般来说，logback-core 里的模块需要一些细微的定制。不过在下面的几节里，我们将讲述几个直接可以使用的 `appender`。

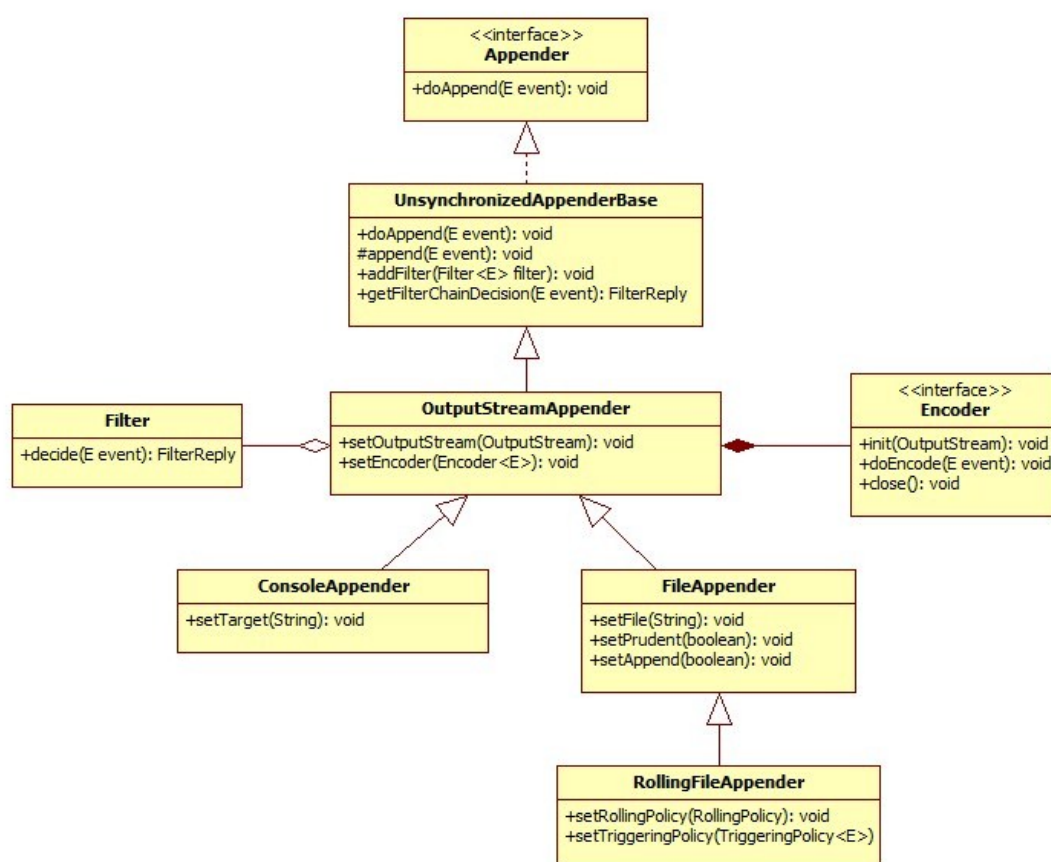
### 4.3.1. OutputStreamAppender

`OutputStreamAppender` 把事件添加到 `java.io.OutputStream`。该类提供其他 `appender` 所需

的基本服务。用户通常不直接实例化 `OutputStreamAppender` 对象。由于 `java.io.OutputStream` 一般无法被方便地映射到字符串，所以无法在配置文件里指定目标 `OutputStream` 对象。简而言之，你不能在配置文件里配置 `OutputStreamAppender`。但这不是说 `OutputStreamAppender` 没有配置属性。它的属性如下。

属性名	类型	描述
encoder	Encoder	决定把事件写入到底层 <code>OutputStreamAppender</code> 的方式。

`OutputStreamAppender` 是另外三个 appender 的超类：`ConsoleAppender`、`FileAppender` 及其直接子类 `RollingFileAppender`。下图演示了 `OutputStreamAppender` 和其子类的类图。



### 4.3.2. ConsoleAppender

`ConsoleAppender` 把事件添加到控制台，更准确地说是 `System.out` 或 `System.err`，默认为前者。`ConsoleAppender` 按照用户指定的 `encoder` 对事件进行格式化。`System.out` 和 `System.err` 都是 `java.io.PrintStream` 类型，因此，它们被包裹在有缓冲 I/O 操作的 `OutputStreamWriter` 里。

属性名	类型	描述
encoder	Encoder	参见 OutputStreamAppender 属性
target	String	字符串 “System.out ” 或 “System.err ”。默认为 “System.out”。

下面是使用 ConsoleAppender 的配置示例。

示例：ConsoleAppender 配置

(logback-examples/src/main/java/chapters/appenders/conf/logback-Console.xml)

```
<configuration>

  <appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
    <!--
      encoders are assigned the type
      ch.qos.logback.classic.encoder.PatternLayoutEncoder by
default
    -->
    <encoder>
      <pattern>%-4relative [%thread] %-5level - %msg%n</pattern>
    </encoder>
  </appender>

  <root level="DEBUG">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

运行：

```
java                                chapters.appenders.ConfigurationTester
src/main/java/chapters/appenders/conf/logback-Console.xml
```

4.3.3. FileAppender

FileAppender 是 OutputStreamAppender 的子类，把记录事件添加到文件。目标文件通过 File 选项指定。如果文件已经存在，则根据 Append 属性追加或清空文件。

属性名	类型	描述
append	boolean	如果 true，事件被追加到现存文件尾部。 如果 false，清空现存文件。 默认为 true。

encoder	Encoder	参见 OutputStreamAppender 属性
file	String	<p>被写入的文件名。如果文件不存在，则创建之。</p> <p>没有默认值。</p> <p>如果文件的父目录不存在，FileAppender 会自动创建各级不存在的目录。</p>
prudent	boolean	<p>在 prudent 模式下，FileAppender 将安全地写入指定文件，即使其他 FileAppender 实例运行在不同 JVM 上，比如运行在不同主机上。</p> <p>prudent 默认值为 false。</p> <p>prudent 模式意味着 Append 属性自动设为 true。</p> <p>prudent 模式写记录事件时，大约消耗非 prudent 模式的三倍。在一台“普通”的 PC 上，向本地硬盘写文件，写一条记录事件，非 prudent 模式需要 10 微妙，prudent 模式需要 30 微妙。也就是非 prudent 模式每秒记录 100000 条事件，prudent 模式每秒 33000 条。</p> <p>不同 JVM 写入同一个文件时，prudent 模式高效地排列 I/O 操作。所以，由于访问文件的 JVM 的数量增加，导致每次 I/O 操作都会有延迟。只要 I/O 操作的总数大约是 20 次记录请求/秒，就可以忽略对性能的影响。每秒 100 次或等多次 I/O 操作会影响性能，此时应当避免 prudent 模式。</p> <p>prudent 模式可以与 RollingFileAppender 联合使用，但有些限制。</p>

下面是 FileAppender 的配置文件例子。

示例：FileAppender 配置

(logback-examples/src/main/java/chapters/appenders/conf/logback-fileAppender.xml)

```
<configuration>

  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <file>testFile.log</file>
    <append>true</append>

    <!--
      encoders are assigned the type
      ch.qos.logback.classic.encoder.PatternLayoutEncoder by
default
    -->
```

```
<encoder>
  <pattern>%-4relative [%thread] %-5level %logger{35} - %msg%n
</pattern>
</encoder>
</appender>

<root level="DEBUG">
  <appender-ref ref="FILE" />
</root>
</configuration>
```

运行:

```
java                                chapters.appenders.ConfigurationTester
src/main/java/chapters.appenders/conf/logback-fileAppender.xml
```

#### 4.3.3.1. 唯一文件名（用时间戳）

在程序开发期，或对于短生命周期程序如批量程序，可以在个新程序启动时创建新记录文件。用<timestamp>元素可以轻易做到。举例如下。

示例：用时间戳实现名称唯一的 FileAppender

(logback-examples/src/main/java/chapters.appenders/conf/logback-timestamp.xml)

```
<configuration>

  <!--
    Insert the current time formatted as "yyyyMMdd'T'HHmmss" under the
key
    "bySecond" into the logger context. This value will be available
to
    all subsequent configuration elements.
  -->
  <timestamp key="bySecond" datePattern="yyyyMMdd'T'HHmmss" />

  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <!--
      use the previously created timestamp to create a uniquely named
log
      file
    -->
    <file>log-{bySecond}.txt</file>
    <encoder>
      <pattern>%logger{35} - %msg%n</pattern>
```



```
</encoder>
</appender>

<root level="DEBUG">
  <appender-ref ref="FILE" />
</root>
</configuration>
```

timestamp 元素有两个属性：key 和 datePattern。属性 key 是变量名，对余下的配置元素可用。属性 datePattern 表示把当前时间（解析配置文件的时间）转换成字符串时使用的日期模式，遵从 java.text.SimpleDateFormat 里的约定。

运行：

```
java                                chapters.appenders.ConfigurationTester
src/main/java/chapters/appenders/conf/logback-timestamp.xml
```

#### 4.3.4. RollingFileAppender

RollingFileAppender 继承 FileAppender，能够滚动记录文件。例如，RollingFileAppender 能先记录到文件 “log.txt”，然后当符合某个条件时，变成记录到其他文件。

RollingFileAppender 有两个与之互动的重要子组件。第一个是 RollingPolicy，负责滚动。第二个是 TriggeringPolicy，决定是否以及何时进行滚动。所以，RollingPolicy 负责 “什么”，TriggeringPolicy 负责 “何时”。

要想 RollingFileAppender 起作用，必须同时设置 RollingPolicy 和 TriggeringPolicy。不过，如果 RollingPolicy 也实现了 TriggeringPolicy 接口，那么只需要设置 RollingPolicy。

下面是 RollingFileAppender 的可用属性。

属性名	类型	描述
file	String	参加 FileAppender 属性
append	boolean	参加 FileAppender 属性
encoder	Encoder	参见 OutputStreamAppender 属性
rollingPolicy	RollingPolicy	当发生滚动时，决定 RollingFileAppender 的行为。
triggeringPolicy	TriggeringPolicy	告知 RollingFileAppender 何时激活滚动。
prudent	boolean	prudent 模式下不支持 FixedWindowRollingPolicy。 RollingFileAppender 支持 prudent 与

		<p>TimeBasedRollingPolicy 的联合使用，但有两个限制：</p> <ol style="list-style-type: none"><li>1. 在 <code>prudent</code> 模式，不支持也不允许文件压缩（不能在一个 JVM 压缩文件时，让另一个 JVM 写文件）。</li><li>2. 不能设置 <code>FileAppender</code> 的 <code>file</code> 属性，必须留空。实际上，多数操作系统不允许当一个进程打开文件时又重命名该文件。</li></ol> <p>参加 <code>FileAppender</code> 属性。</p>
--	--	---

#### 4.3.4.1. 滚动策略概述

`RollingPolicy` 负责滚动步骤，涉及文件移动和重命名。

`RollingPolicy` 接口如下：

```
package ch.qos.logback.core.rolling;

import ch.qos.logback.core.FileAppender;
import ch.qos.logback.core.rolling.helper.CompressionMode;
import ch.qos.logback.core.spi.Lifecycle;

public interface RollingPolicy extends Lifecycle {

    public void rollover() throws RolloverFailure;
    public String getActiveFileName();
    public CompressionMode getCompressionMode();
    public void setParent(FileAppender appender);
}
```

`rollover` 方法完成对当前记录文件的归档工作。`getActiveFileName()` 方法计算当前记录文件（写实时记录的地方）的文件名。如 `getCompressionMode()` 方法名所示，`RollingPolicy` 也负责决定压缩模式。最后，`RollingPolicy` 通过 `setParent()` 方法得到一个对其父的引用。

#### 4.3.4.2. FixedWindowRollingPolicy

当发生滚动时，`FixedWindowRollingPolicy` 根据如下固定窗口（`window`）算法重命名文件。

选项“`fileNamePattern`”代表归档（滚动）记录文件的文件名模式。该选项是必需的，且必需在模式的某处包含标志“`%i`”。

下面是 `FixedWindowRollingPolicy` 的可用属性。

属性名	类型	描述
minIndex	int	窗口索引的最小值
maxIndex	int	窗口索引的最大值
fileNamePattern	String	例如,对于最小值和最大值分别是 1 和 3 的文件名模式 “MyLogFile%i.log”,会产生归档文件 MyLogFile1.log、MyLogFile2.log 和 MyLogFile3.log。  该属性还可以指定文件压缩选项。例如 “MyLogFile%i.log.zip” 表示归档文件必须用 zip 格式进行压缩;还支持 “.gz” 格式。

由于固定窗口滚动策略需要的文件重命名操作与窗口大小一样多,所以强烈建议不要使用太大的窗口大小。当用户指定过大的窗口大小时,当前的代码会自动将窗口大小设为 12。

让我们来看固定窗口滚动策略的一个更具体的例子。假设“minIndex”是 1,“maxIndex”是 3,“fileNamePatter”是 “foo%i.log”。

滚动文件数量	活动输出目标	归档记录文件	描述
0	foo.log	-	还没发生滚动,记录到初始文件。
1	foo.log	foo1.log	第 1 次滚动。foo.log 被重命名为 foo1.log。创建新 foo.log 并成为活动输出目标。
2	foo.log	foo1.log, foo2.log	第 2 次滚动。foo1.log 被重命名为 foo2.log。foo.log 被重命名为 foo1.log。创建新 foo.log 并成为活动输出目标。
3	foo.log	foo1.log, foo2.log, foo3.log	第 3 次滚动。foo2.log 被重命名为 foo1.log。foo1.log 被重命名为 foo2.log。foo.log 被重命名为 foo1.log。创建新 foo.log 并成为活动输出目标。
4	foo.log	foo1.log, foo2.log, foo3.log	此时及此后,发生滚动时会先删除 foo3.log。其他文件按照上面的步骤被重命名。此时及此后,将有 3 个归档记录文件和 1 个活动记录文件。

下面的配置文件演示了 RollingFileAppender 和 FixedWindowRollingPolicy。注意 “file” 选项是必需的,即使它与 “fileNamePattern” 选项有部分重叠信息。

示例: 使用 FixedWindowRollingPolicy 的 RollingFileAppender 的示例配置  
(logback-examples/src/main/java/chapters/appenders/conf/logback-RollingFixedWindow.xml)

```
<configuration>
  <appender name="FILE"
```

```
class="ch.qos.logback.core.rolling.RollingFileAppender">

<file>testFile.log</file>
<rollingPolicy
class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy">
    <fileNamePattern>testFile.%i.log.zip</fileNamePattern>
    <minIndex>1</minIndex>
    <maxIndex>3</maxIndex>
</rollingPolicy>

    <triggeringPolicy

class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
        <maxFileSize>5MB</maxFileSize>
    </triggeringPolicy>
    <encoder>
        <pattern>%-4relative [%thread] %-5level %logger{35} - %msg%n
        </pattern>
    </encoder>
</appender>

<root level="DEBUG">
    <appender-ref ref="FILE" />
</root>
</configuration>
```

### 4.3.5. TimeBasedRollingPolicy

TimeBasedRollingPolicy 或许是最受流行的滚动策略。它根据时间来制定滚动策略，例如根据日或月。TimeBasedRollingPolicy 既负责滚动也负责触发滚动。实际上，TimeBasedRollingPolicy 同时实现了 RollingPolicy 接口和 TriggeringPolicy 接口。

TimeBasedRollingPolicy 有两个属性：必需的“fileNamePattern”和可选的“maxHistory”。

属性名	类型	描述
fileNamePattern	String	必需。定义滚动（归档）记录文件的名字。其值应当包含文件名及“%d”格式转换符。“%d”可以包含一个 java.text.SimpleDateFormat 指定的日期时间模式。如果没有指定日期时间模式，则默认为 yyyy-MM-dd。RollingFileAppender（TimeBasedRollingPolicy 之父）的“file”选项可有可无。

		<p>通过设置 “file” 属性，可以为活动文件和归档文件指定不同的位置。当前记录总是被指向由 “file” 属性指定的文件。如果有 “file” 属性，则活动文件的名字不会改变。而如果没有 “file” 属性，则活动文件的名字会根据 “fileNamePattern” 的值每隔一段时间就重新计算一次。下面的例子会解释这点。</p> <p>“ %d{} ” 里的日期时间模式遵循 java.text.SimpleDateFormat 的约定。在 fileNamePattern 或日期时间模式里的前置 “/” 或后置 “\” 会被当作目录分隔符。</p>
maxHistory	int	<p>控制被保留的归档文件的最大数量，超出数量就删除旧文件。例如，假设每月滚动，且 maxHistory 是 6，则只保留最近 6 个月的归档文件，删除之前的文件。注意当删除旧归档文件时，那些为了归档而创建的目录也会被删除。</p>

下面是 fileNamePattern 的部分值及其作用。

file Name Pattern	滚动计划	示例
/wombat/foo.%d	每天滚动（在午夜）。因为%d没有指定日期时间格式，所以使用默认的yyyy-MM-dd，即每天滚动。	<p>未设置 file 属性：在 2006 年 11 月 23 日，记录会输出到文件/wombat/foo.2006-11-23。在午夜及之后的 24 日，输出到文件/wombat/foo.2006-11-24。</p> <p>设置 file 属性为 /wombat/foo.txt：活动记录文件总是 /wombat/foo.txt。在 2006 年 11 月 23 日，记录会输出到文件/wombat/foo.txt。在午夜，/wombat/foo.txt 被重命名为 /wombat/foo.2006-11-23，创建新的/wombat/foo.txt，之后的 24 日，输出到文件 /wombat/foo.txt。</p>
/wombat/%d{yyyy/MM}/foo.txt	每月初滚动	<p>未设置 file 属性：在 2006 年 10 月，记录会输出到文件 /wombat/2006/10/foo.txt。在 10 月 31 日午夜及 11 月，输出到文件</p>

		<p>/wombat/2006/11/foo.txt。</p> <p>设置 file 属性为 /wombat/foo.txt：活动记录文件总是 /wombat/foo.txt。在 2006 年 10 月，记录会输出到文件/wombat/foo.txt。在 10 月 31 日午夜， /wombat/foo.txt 被重命名为 /wombat/2006/10/foo.txt, 创建新的/wombat/foo.txt，之后的 1 个月，输出到文件 /wombat/foo.txt。在 11 月 30 日的午夜， /wombat/foo.txt 被重命名为 /wombat/2006/11/foo.txt, 依此类推。</p>
/wombat/foo.%d{yyyy-ww}.log	每周初滚动。 注意每周的第一天是星期几取决于地区（locale）	同前，只是滚动发生在每周的开头。
/wombat/foo.%d{yyyy-MM-dd_HH}.log	每小时滚动	同前，只是滚动发生在每小时的开头。
/wombat/foo.%d{yyyy-MM-dd_HH-mm}.log	每分钟滚动	同前，只是滚动发生在每分钟的开头。

所有 “\” 和 “/” 都被解释为目录分隔符。任何需要的目录都会被创建。所以你可以轻松地把记录文件放到不同的目录。

正如 FixedWindowRollingPolicy， TimeBasedRollingPolicy 支持自动压缩文件。如果 “fileNamePattern” 选项以 “.gz” 或 “.zip” 结尾，就表示需要压缩。

属性名	类型	描述
/wombat/foo.%d.gz	每天滚动(在午夜)，归档文件被自动压缩为 GZIP 文件。	<p>未设置 file 属性：在 2006 年 11 月 23 日，记录会输出到文件 /wombat/foo.2006-11-23。在午夜， /wombat/foo.txt 被压缩为/wombat/foo.2009-11-23.gz, 之后的 24 日，输出到文件/wombat/foo.2006-11-24。</p> <p>设置 file 属性为/wombat/foo.txt：活动记录文件总是</p>

		/wombat/foo.txt。在 2006 年 11 月 23 日，记录会输出到文件 /wombat/foo.txt。在午夜，在午夜，/wombat/foo.txt 被压缩为/wombat/foo.2009-11-23.gz，创建新的/wombat/foo.txt，之后的 24 日，输出到文件 /wombat/foo.txt。
--	--	---

属性“fileNamePattern”有两个目的。一是，通过学习模式，logback 计算滚动周期。二是，计算每个归档文件的名称。注意，可以为两种不同的模式指定同一个周期。模式 yyyy-MM 和 yyyy@MM 都表示每月滚动，但它们的归档文件名不同。

通过设置“file”属性，你可以为活动记录文件和归档记录文件指定不同的位置。记录输出会被指向“file”属性指定的文件，所以活动文件的名称不会改变。然而，如果没有“file”属性，则活动文件的名称会根据“fileNamePattern”的值每隔一段时间就重新计算一次。

属性“maxHistory”控制被保留的归档文件的最大数量，超出数量就删除旧文件。例如，假设每月滚动，且 maxHistory 是 6，则只保留最近 6 个月的归档文件，删除之前的文件。注意当删除旧归档文件时，那些为了归档而创建的目录也会被删除。

出于某些技术原因，滚动不是时钟驱动，而是按照记录事件的到达时间。比如，在 2002 年 3 月 8 日，假设“fileNamePattern”是“yyyy-MM-dd”（每日滚动），则午夜过后的第一个记录事件会触发滚动。如果，比如说直到 0 点 23 分 47 秒之前都没有记录事件，那么滚动发生的实际时间是 3 月 9 日 0 点 23 分 47 秒，而不是 0 点 0 分。因此，根据事件到达的频率，滚动或许会被延时触发。不过，在某个时期内产生的所有记录事件都被输出到划分该时期的正确的文件，从这个角度看，虽然有延迟，滚动算法仍然是正确的。

下面是 RollingFileAppender 和 TimeBasedRollingPolicy 合作的例子。  
示例：使用 TimeBasedRollingPolicy 的 RollingFileAppender 的配置例子

(logback-examples/src/main/java/chapters/appenders/conf/logback-RollingTimeBased.xml)

```
<configuration>

  <appender name="FILE"
    class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>logFile.log</file>
    <rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
      <!-- daily rollover -->

    <fileNamePattern>logFile.%d{yyyy-MM-dd}.log</fileNamePattern>
```

```
        <!-- keep 30 days worth of history -->
        <maxHistory>30</maxHistory>
    </rollingPolicy>

    <encoder>
        <pattern>%-4relative [%thread] %-5level %logger{35} - %msg%n
        </pattern>
    </encoder>
</appender>

<root level="DEBUG">
    <appender-ref ref="FILE" />
</root>
</configuration>
```

下面是在 `prudent` 模式下，使用 `TimeBasedRollingPolicy` 的 `RollingFileAppender` 的配置例子。

示例：使用 `TimeBasedRollingPolicy` 的 `RollingFileAppender` 的配置例子

(`logback-examples/src/main/java/chapters/appenders/conf/logback-PrudentTimeBasedRolling.xml`)

```
<configuration>
  <appender name="FILE"
    class="ch.qos.logback.core.rolling.RollingFileAppender">

    <!-- Support multiple-JVMs writing to the same log file -->
    <prudent>true</prudent>
    <rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

      <fileNamePattern>logFile.%d{yyyy-MM-dd}.log</fileNamePattern>
      <maxHistory>30</maxHistory>
    </rollingPolicy>

    <encoder>
      <pattern>%-4relative [%thread] %-5level %logger{35} - %msg%n
      </pattern>
    </encoder>
  </appender>

  <root level="DEBUG">
```



```
<appender-ref ref="FILE" />
</root>
</configuration>
```

#### 4.3.5.1. 基于大小和时间的归档

有时你也许想按照日期进行归档的同时限制每个记录文件的大小，特别是当后处理工具对记录文件大小有限制时。Logback 为此提供了 `SizeAndTimeBasedFNATP`，它是 `TimeBasedRollingPolicy` 的子组件，FNATP 代表“文件命名和触发策略”。

下面的例子演示了基于大小和时间的记录文件归档。

示例：SizeAndTimeBasedFNATP 配置

(logback-examples/src/main/java/chapters/appenders/conf/logback-sizeAndTime.xml)

```
<configuration>

  <appender name="ROLLING"
    class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>mylog.txt</file>
    <rollingPolicy
      class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <!-- rollover daily -->

        <fileNamePattern>mylog-%d{yyyy-MM-dd}.%i.txt</fileNamePattern>
        <timeBasedFileNamingAndTriggeringPolicy
          class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
            <!-- or whenever the file size reaches 100MB -->
            <maxFileSize>100MB</maxFileSize>
          </timeBasedFileNamingAndTriggeringPolicy>
        </rollingPolicy>
    <encoder>
      <pattern>%msg%n</pattern>
    </encoder>
  </appender>

  <root level="debug">
    <appender-ref ref="ROLLING" />
  </root>

</configuration>
```

注意“%d”后面的“%i”。在当前时间周期结束之前，每当当前记录文件达到“maxFileSize”时，就会用递增索引归档，索引从 0 开始。

基于大小和时间的归档支持删除旧归档文件。你需要用“maxHistory”属性指定要保留的周期的数量。当程序停止并重启时，记录会从正确的位置继续，即当前周期的最大索引。

### 4.3.6. 触发策略概述

TriggeringPolicy 负责指示 RollingFileAppender 在什么时候滚动。

TriggeringPolicy 接口只有一个方法。

```
package ch.qos.logback.core.rolling;

import java.io.File;

import ch.qos.logback.core.spi.Lifecycle;

public interface TriggeringPolicy<E> extends Lifecycle {
    public boolean isTriggeringEvent(final File activeFile, final E
event);
}
```

isTriggeringEvent()方法有两个参数，一个是归档文件，一个是当前正被处理的记录事件。该方法的具体实现根据这两个参数来决定是否进行滚动。

使用最广泛的触发策略是 TimeBasedRollingPolicy，它也是一个滚动策略，上节已讲过。

#### 4.3.6.1. SizeBasedTriggeringPolicy

查看当前活动文件的大小。如果超过指定大小，SizeBasedTriggeringPolicy 会告诉 RollingFileAppender 去触发当前活动文件的滚动。

SizeBasedTriggeringPolicy 只有一个参数：maxFileSize，默认值是 10MB。

根据数字后面不同的后缀，“maxFileSize”可以是 bytes、KB、MB 或 GB。比如 5000000, 5000KB、5MB 和 2GB 都是合法值，且前三个等价。

下面是 RollingFileAppender 与 SizeBasedTriggeringPolicy 合作的配置例子，当记录文件的大小超过 5MB 后，会触发滚动。

示例：使用 SizeBasedTriggeringPolicy 的 RollingFileAppender 的配置

(logback-examples/src/main/java/chapters/appenders/conf/logback-RollingSizeBased.xml)

```
<configuration>

  <appender name="FILE"
    class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>test.log</file>
    <rollingPolicy
class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy">
      <fileNamePattern>test.%i.log.zip</fileNamePattern>
      <minIndex>1</minIndex>
      <maxIndex>3</maxIndex>
    </rollingPolicy>

    <triggeringPolicy
class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
      <maxFileSize>5MB</maxFileSize>
    </triggeringPolicy>
    <encoder>
      <pattern>%-4relative [%thread] %-5level %logger{35} - %msg%n
    </pattern>
    </encoder>
  </appender>

  <root level="DEBUG">
    <appender-ref ref="FILE" />
  </root>
</configuration>
```

## 4.4. Logback Classic



在 logback-core 里的记录事件都是泛型，而在 logback-classic 里，记录事件都是 ILoggingEvent 的实例。Logback-classic 仅仅是一个专门处理 ILoggingEvent 实例的管道。

### 4.4.1. SocketAppender

到目前为止，我们讨论的 appender 都只能输出到本地资源。而 SocketAppender 被设计为通过序列化 ILoggingEvent 实例把记录输出到远程实体。被序列化的事件的真实类型是

LoggingEventVO，它实现了 ILoggingEvent 接口。尽管如此，就记录事件而言，远程记录仍然是无损的。在接收和反序列化后，事件像是从本地产生的一样被记录。运行在不同机器上的多个 SocketAppender 实例可以把各自的记录输出到一个格式固定的中央记录服务器。SocketAppender 不关联 layout，因为它把序列化的事件发送到远程服务器。SocketAppender 运作在 TCP 层上，TCP 层提供可靠、有序、流量控制的端到端的八进制流。因此，如果远程服务器可访问，则记录事件最终会到达那里。否则，如果远程服务器关机或不可访问，那么记录事件会被抛弃。当远程服务器恢复可用时，会透明地继续传输事件。这种透明的重新连接是由一个连接器（connector）线程执行的，它定时尝试连接服务器。

记录事件被本地 TCP 实现自动地缓冲。这意味着如果服务器连接很慢，但快于客户端生成事件的速度，那么客户端不会受网络连接慢的影响。但是如果网络连接慢于生成事件的速度，那么客户端只能按网络速度执行。特别是在服务器当机这种极端情况下，客户端最终会被阻塞。而当网络连接恢复，但服务器当机时，客户端不会被阻塞，此时记录事件还是会因为服务器不可用而丢失。

如果连接器线程仍然存在，即使 SocketAppender 不再关联到任何 logger，也不会被垃圾回收。连接器线程只在当与服务器之间没有连接时存在。为避免这个垃圾回收问题，你应当显式地关闭 SocketAppender。会创建/销毁很多 SocketAppender 实例的长期运行的程序应当注意这个垃圾回收问题。多数其他程序可以安全地忽略这个问题。如果宿主 JVM 在 SocketAppender 关闭之前退出了，不管 SocketAppender 被显式关闭还是交给垃圾回收，都有可能管道（pipe）里剩有一些未被传输的数据，这些数据会丢失。为避免数据丢失，一般在退出程序之前显式地调用 SocketAppender 的 close()方法或调用 LoggerContext 的 stop()方法就可以了。

远程服务器由“RemoteHost”属性和“Port”属性标识。SocketAppender 的属性见下表。

属性名	类型	描述
IncludeCallerData	boolean	如果 true，则调用者的数据对远程服务器可用。 默认为 false。
Port	int	远程服务器的端口。
ReconnectionDelay	int	正整数，表示连接失败后，重连服务器的时间间隔，单位毫秒。  默认值是 30000 即 30 秒。设为 0 表示关闭重连功能。 注意，当成功连接到服务器后，不会再用连接器线程。
RemoteHost	String	服务器的主机名。

Logback 标准发行包包含一个简单的记录服务器程序“ch.qos.logback.classic.net.SimpleSocketServer”，可以支持多个 SocketAppender 客户端。它

等待来自 `SocketAppender` 客户端的记录事件，接收到事件后，按照本地服务器的记录策略进行记录。它有两个参数：端口和配置文件，端口是监听端口，配置文件是 XML 格式的配置脚本。

假设你在 `logback-examples/` 目录，运行：

```
java ch.qos.logback.classic.net.SimpleSocketServer 6000
src/main/java/chapters/appenders/socket/server1.xml
```

6000 是监听端口。server1.xml 是配置脚本，为根 logger 添加了一个 `ConsoleAppender` 和 `RollingFileAppender`。启动 `SimpleSocketServer` 后，你可以从使用了 `SocketAppender` 的多个客户端发送记录事件。本例涉及两个客户端：`chapters.appenders.SocketClient1` 和 `chapters.appenders.SocketClient2`。两个客户端都等待用户在控制台键入一行文字。键入的文字被封装在 `debug` 级别的记录事件里，然后发送到远程服务器。两个客户端的不同之处在于 `SocketAppender` 的配置，`SocketClient1` 在代码里配置，而 `SocketClient2` 用配置文件配置。

假设 `SimpleSocketServer` 运行在本机，用下面的命令连接到它：

```
java chapters.appenders.socket.SocketClient1 localhost 6000
```

在控制台输入的每一行都会出现在 `SimpleSocketServer` 的控制台。如果停止并重启 `SimpleSocketServer`，客户端会透明地重新连接到新的服务器，但是在断开连接期间生成的事件都被简单而不可恢复地抛弃。

与 `SocketClient1` 不同，`SocketClient2` 用 XML 格式的配置进行配置。下面的配置文件 `client1.xml` 创建一个 `SocketAppender` 并把它关联到根 logger。

示例：SocketAppender 配置

(`logback-examples/src/main/java/chapters/appenders/socket/client1.xml`)

```
<configuration>

  <appender name="SOCKET"
class="ch.qos.logback.classic.net.SocketAppender">
    <RemoteHost>${host}</RemoteHost>
    <Port>${port}</Port>
    <ReconnectionDelay>10000</ReconnectionDelay>
    <IncludeCallerData>${includeCallerData}</IncludeCallerData>
  </appender>

  <root level="debug">
    <appender-ref ref="SOCKET" />
  </root>

</configuration>
```

注意在上面的配置文件里，属性 RemoteHost、Port 和 IncludeCallerData 的值都不是直接给出的，而是用变量。这些变量的值可以作为系统属性来指定：

```
java          -Dhost=localhost          -Dport=6000          -DincludeCallerData=false
chapters.appenders.socket.SocketClient2 src/main/java/chapters/appenders/socket/client1.xml
```

该命令的结果与前面的 SocketClient1 相似。

我们再次强调记录事件的序列化是无损的。反序列化的记录事件与其他任何记录事件承载一样的信息。反序列化的事件被当作从本地产生的一样进行处理，除了序列化记录事件在默认情况下不包含调用者的数据。下面的例子演示了这点。首先，启动 SimpleSocketServer：

```
java          ch.qos.logback.classic.net.SimpleSocketServer          6000
src/main/java/chapters/appenders/socket/server2.xml
```

配置文件 server2.xml 创建一个 ConsoleAppender，在输出其他信息的同时也输出调用者的文件名和行号。如果像前面一样用配置文件 client1.xml 运行 SocketClient2，你会注意到服务器端的输出包含两个问号而不是调用者的文件名和行号：

```
2006-11-06 17:37:30,968 DEBUG [Thread-0] [?:?] chapters.appenders.socket.SocketClient2 - Hi
```

通过设置 SocketAppender 的 IncludeCallerData 属性为 true，就可以改变输出内容：

```
java          -Dhost=localhost          -Dport=6000          -DincludeCallerData=true
chapters.appenders.socket.SocketClient2 src/main/java/chapters/appenders/socket/client1.xml
```

反序列化事件与本地生成的事件按照同样的方式进行处理，甚至还能被发送到第二台服务器作进一步处理。你也许想设置两台服务器，第一台作为隧道将接收到的事件发送给第二台我放弃。

#### 4.4.2. JMSAppenderBase

JMSAppenderBase 类与 SocketAppender 完成相同概念的任务，只是基于 JMS API 而不是 TCP 套接字。JMS 即 Java 消息系统 API 为面向消息的中间件 (MOM) 产品提供抽象层。JMS 的一个关键架构概念就是解耦消息生产者和消息消费者。发送者不需要等待接收者处理完消息，有消息时接收者才会消费消息。这样的消息称为是异步传递的。同样重要的是，消费者和生产者都可以随意从 JMS 频道 (channel) 上增加或移除。消息生产者和消息消费者可以透明地独立变化，彼此无视对方。

JMS 规范提供两种消息模型：发布-订阅和点对点队列。Logback 用 JMSTopicAppender

支持前者，用 `JMSQueueAppender` 支持后者，两个 `appender` 都继承 `JMSAppenderBase` 类，都把序列化的事件发布到用户指定的主题(topic)或队列 (queue)。

一个或多个 `JMSTopicSink` 或 `JMSQueueSink` 程序可以注册一个 JMS 服务器并消费序列化的事件。`JMS appender` 生成的事件的消费者不一定只是 `JMSTopicSink` 或 `JMSQueueSink` 程序。只要程序或 `MessageDrivenBean` 能够订阅主题或队列，并且能够消费序列化的记录事件消息，就可以作为事件消费者。在 `JMSTopicSink` 或 `JMSQueueSink` 模型基础上可以迅速建立其他消费者。

下面是 `JMSAppenderBase` 的属性。

属性名	类型	描述
<code>InitialContextFactoryName</code>	<code>String</code>	初始 JNDI 上下文工厂的类名。如果有 <code>jndi.properties</code> ，或 <code>JMSAppenderBase</code> 的子类运行在应用程序服务器里，就不需要设置该属性。 如果设置了该属性，也应当设置 <code>ProviderURL</code> 属性。
<code>ProviderURL</code>	<code>String</code>	JNDI 服务提供商的配置信息。属性值应当包含一个 URL 字符串，比如 <code>ldap://somehost:389</code> 。 只有当指定了 <code>InitialContextFactoryName</code> 时，才有效；否则忽略该属性。
<code>URLPkgPrefixes</code>	<code>String</code>	包含当加载 URL 上下文工厂时所用到的包名前缀列表，以冒号分隔。 对 JBoss 来说是： <code>org.jboss.naming.org.jnp.interfaces</code> 。 Weblogic 不需要该属性。 只有当指定了 <code>InitialContextFactoryName</code> 时，才有效；否则忽略该属性。
<code>SecurityPrincipalName</code>	<code>String</code>	访问 JNDI 命名空间时用到。 只有当指定了 <code>InitialContextFactoryName</code> 时，才有效；否则忽略该属性。
<code>SecurityCredentials</code>	<code>String</code>	访问 JNDI 命名空间时用到。 只有当指定了 <code>InitialContextFactoryName</code> 时，才有效；否则忽略该属性。
<code>UserName</code>	<code>String</code>	创建主题或队列连接时的用户名。
<code>Password</code>	<code>String</code>	创建主题或队列连接时的密码。

JMS 主题、队列和连接工厂都是受管对象，通过 JNDI API 获取。这也意味着必须获取

JNDI 上下文。获取 JNDI 有两种常见方法。如果一个名为“jndi.properties”的文件对 JNDI API 可用，那么 JNDI API 会用文件里的信息来获取 JNDI 上下文。要取得初始上下文，一个简单的调用：

```
InitialContext jndiContext = new InitialContext();
```

调用无参数的 `InitialContext()` 构造方法也可用于 EJB。实际上，EJB 规范规定应用程序服务器为每个 EJB 提供一个环境命名上下文（ENC）。

在第二种方法里，要指定一些预定义属性。这些预定义属性被传递给 `InitialContext` 构造方法，用于连接到命名服务提供商。例如，要连接到命名服务提供商“ActiveMQ”，应该这样写：

```
Properties env = new Properties();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "org.apache.activemq.jndi.ActiveMQInitialContextFactory");
env.put(Context.PROVIDER_URL, "tcp://hostname:61616");
Context ctx = new InitialContext(env);
```

“hostname”是运行 ActiveMQ 服务器的主机。

其他 JNDI 提供商显然需要不同的值。之前提到过，初始 JNDI 上下文可以通过在 EJB 内调用无参数的 `InitialContext()` 方法而获得。只有当客户端运行在其他 JVM 上时，才需要考虑 `jndi.properties` 文件，或在调用参数为 `Properties`（即 `HashTable`）的 `InitialContext` 构造方法之前设置其他值。

#### 4.4.2.1. JMS appenders 备注

用 JMS 传送同一个信息包比用原始 TCP 套接字要慢。那些吹嘘其消息平台的性能的 JMS 厂商有意忽略这个朴素的事实。对存储和发送有担保的消息系统需要付出代价。作为对代价增长的回报，JMS 消息系统提供对发送者和接收者的解耦。只要 JMS 服务器可到达，消息最终会到达目的地。不过，如果 JMS 服务器当机或不可到达呢？

根据 JMS 规范，生产者可以把消息标记为持久的或非持久的。作为客户端的一种操作，持久传递模式指示 JMS 服务器把消息记录到稳定的存储器，消息不回因 JMS 服务器崩溃而丢失。JMS appender 不设置持久化标记，因为根据 JMS 规范，传递模式被视为一项受管属性。

一旦当消息到达 JMS 服务器时，服务器承担把消息传递到目的地的责任，客户端与此无关。假如 JMS 服务器不可到达呢？JMS API 为处理这种情况提供了 `ExceptionListener` 接口。当客户端检测到与 JMS 服务器失去连接时，就调用 `ExceptionListener` 的 `onException()` 方法。当被告知这个问题时，客户端可以尝试重新建立连接。根据 JMS 规范之 4.3.8 节，服



务器应当在告知客户端之前，尝试解决连接问题。JMS appender 不实现 `ExceptionListener` 接口。

#### 4.4.2.2. JMSTopicAppender

JMSTopicAppender 作为消息生产者，发布和订阅主题。

最重要的方法 `doAppend()` 如下：

```
public void append(ILoggingEvent event) {
    if (!isStarted()) {
        return;
    }

    try {
        ObjectMessage msg = topicSession.createObjectMessage();
        Serializable so = pst.transform(event);
        msg.setObject(event);
        topicPublisher.publish(msg);
        successiveFailureCount = 0;
    } catch (Exception e) {
        successiveFailureCount++;
        if (successiveFailureCount > SUCCESSIVE_FAILURE_LIMIT) {
            stop();
        }
        addError("Could not publish message in JMSTopicAppender [" +
name
                + "].", e);
    }
}
```

`isStarted()` 方法允许 appender 检查是否满足前提条件，特别是是否有一个有效的、打开的 `TopicConnection` 和 `TopicSession`。如果条件不满足，`append()` 方法返回，不执行任何工作。如果条件满足，`append()` 方法将发布记录事件，实现方法是从 `TopicSession` 取得一个 `javax.jms.ObjectMessage` 并把输入参数里的记录事件设置给它作为它的承载物（payload）。一旦消息的承载物被设置，消息就被发布。注意 `PreSerializationTransformer` 把 `ILoggingEvent` 转换成了可序列化的对象。只有序列化的对象才能被 `ObjectMessage` 传输。

总之，JMSTopicAppender 广播的消息是由用户指定的 JMS 主题所承载的一个序列化的 `LoggingEvent`。这些事件可被 JMSTopicSink 或类似消费者所处理。根据 JMS 规范，JMS 服务器将调用正式注册和订阅了的 `javax.jms.MessageListener` 对象的 `onMessage()` 方法。JMSTopicSink 的 `onMessage()` 实现如下：

```

public void onMessage(javax.jms.Message message) {
    ILoggingEvent event;
    try {
        if (message instanceof ObjectMessage) {
            ObjectMessage objectMessage = (ObjectMessage) message;
            event = (ILoggingEvent) objectMessage.getObject();
            Logger log = (Logger) LoggerFactory.getLogger(event
                .getLoggerName());
            log.callAppenders(event);
        } else {
            logger.warn("Received message is of type "
                + message.getJMSType()
                + ", was expecting ObjectMessage.");
        }
    } catch (JMSEException jmse) {
        logger.error("Exception thrown while processing incoming
message.", jmse);
    }
}
}

```

onMessage()先取得记录事件的承载物，接着取得与记录事件的 logger 同名的 logger，调用取得的 logger 的 callAppenders()方法，这样事件就如同是在本地生成的一样被记录。SimpleSocketServer 使用的 SocketNode 类处理记录事件的方式本质上与此相同。

JMSTopicAppender 的特有属性如下。

属性名	类型	描述
TopicConnectionFactoryBindingName	String	必选。主题工厂的名称。没有默认值。
TopicBindingName	String	必选。主题名称。没有默认值。

JMSTopicAppender 的配置非常直观。

示例：JMSTopicAppender 配置

(logback-examples/src/main/java/chapters/appenders/conf/logback-JMSTopic.xml)

```

<configuration>

    <appender name="Topic"
class="ch.qos.logback.classic.net.JMSTopicAppender">
        <InitialContextFactoryName>
            org.apache.activemq.jndi.ActiveMQInitialContextFactory
        </InitialContextFactoryName>
        <ProviderURL>tcp://localhost:61616</ProviderURL>
    </appender>

```

```
<TopicConnectionFactoryBindingName>
    ConnectionFactory
</TopicConnectionFactoryBindingName>
<TopicBindingName>MyTopic</TopicBindingName>
</appender>

<root level="debug">
    <appender-ref ref="Topic" />
</root>
</configuration>
```

#### 4.4.2.3. JMSQueueAppender

JMSQueueAppender 是点对点队列的消息生产者。

工作方式与 JMSTopicAppender 非常相似。

JMSQueueAppender 的特有属性如下。

属性名	类型	描述
QueueConnectionFactoryBindingName	String	必选。队列工厂的名称。没有默认值。
QueueBindingName	String	必选。队列名称。没有默认值。

JMSQueueAppender 的典型配置与 JMSTopicAppender 的配置非常相似。

示例：JMSQueueAppender 配置

(logback-examples/src/main/java/chapters/appenders/conf/logback-JMSQueue.xml)

```
<configuration>

    <appender name="Queue"
class="ch.qos.logback.classic.net.JMSQueueAppender">
        <InitialContextFactoryName>
            org.apache.activemq.jndi.ActiveMQInitialContextFactory
        </InitialContextFactoryName>
        <ProviderURL>tcp://localhost:61616</ProviderURL>
        <QueueConnectionFactoryBindingName>
            ConnectionFactory
        </QueueConnectionFactoryBindingName>
        <QueueBindingName>MyQueue</QueueBindingName>
    </appender>

    <root level="debug">
```

```
<appender-ref ref="Queue" />
</root>
</configuration>
```

### 4.4.3. SMTPAppender



SMTPAppender 在固定大小的缓冲里积累记录时间，当用户指定的事件发生后，就通过 email 发出这些事件。默认情况下，email 发送是由级别为 ERROR 或更高级别的记录事件触发的。

SMTPAppender 的属性如下。

属性名	类型	描述
SMTPHost	String	必选。SMTP 服务器的主机名。
SMTPPort	int	SMTP 服务器的监听端口。默认为 25。
To	String	接收者的 email 地址。多个接收者用多个<To>元素。
From	String	发送者的 email 地址。
Subject	String	email 的主题。可以是任何符合 PatternLayout 的字符串。 主题是对触发 email 的记录事件应用模式后的字符串。 假设该选项设为 “Log: %logger - %msg”，触发事件的 logger 名为 “com.foo.Bar”，消息是 “Hello world”，则被发送的 email 的主题是 “Log: com.foo.Bar - Hello World”。 默认为 “%logger{20} - %m”。
BufferSize	int	正整数。表示循环缓冲里的记录事件的数量。当达到 BufferSize 后，从缓冲里删除旧事件并添加新事件。默认为 512。
Evaluator	String	通过创建<EventEvaluator/>元素声明该选项。 可以通过 “class” 属性指定 Evaluator 所用的类名。 如果未指定该选项，则 SMTPAppender 被分配一个 OnErrorEvaluator 实例，当遇到级别为 ERROR 或更高级别的事件后，触发 email 传输。 Logback 带有其他 Evaluator：OnMarkerEvaluator 和强大的 JaninoEventEvaluator。
Username	String	email 用户名。默认为 null。
Password	String	email 密码。默认为 null。
STARTTLS	boolean	如果为 true，则 appender 将执行 STARTTLS 命令（如果服务器支持该命令），将连接切换到 SSL。注意连接最初是未加密的。

		默认为 false。
SSL	boolean	如果设为 true，则 appender 将打开与服务器之间的 SSL 连接。 默认为 false。
CharsetEncoding	String	email 正文的字符集编码。 默认为 “UTF-8”。

SMTPAppender 依赖 JavaMail API，在 JavaMail API 1.4 版下通过测试。

下面的演示程序 chapters.appenders.mail.Email，生成大量记录消息，然后生成一个错误消息。该程序有两个参数，第一个是需要生成的记录消息的数量，第二个是 logback 配置文件。该程序生成的最后一条记录事件，即 ERROR 级别的事件，将触发 email 的发送。

下面是该程序的一个配置文件例子。

示例：SMTPAppender 示例配置

(logback-examples/src/main/java/chapters/appenders/mail/mail1.xml)

```
<configuration>

  <appender name="EMAIL"
class="ch.qos.logback.classic.net.SMTPAppender">
    <SMTPHost>ADDRESS-OF-YOUR-SMTP-HOST</SMTPHost>
    <To>EMAIL-DESTINATION</To>
    <To>ANOTHER_EMAIL_DESTINATION</To> <!-- additional destinations
are possible -->
    <From>SENDER-EMAIL</From>
    <Subject>TESTING: %logger{20} - %m</Subject>
    <layout class="ch.qos.logback.classic.PatternLayout">
      <Pattern>%date %-5level %logger - %message%n</Pattern>
    </layout>
  </appender>

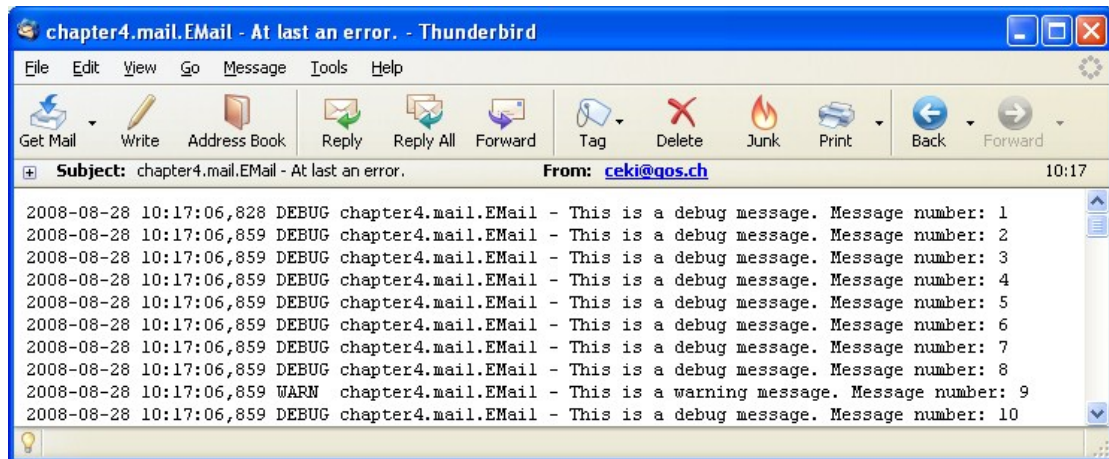
  <root level="debug">
    <appender-ref ref="EMAIL" />
  </root>
</configuration>
```

在运行该程序前，必须先设置 “SMTPHost”、“To” 和 “From” 属性。设置完后，运行：

```
java chapters.appenders.mail.Email 300 src/main/java/chapters/appenders/mail/mail1.xml
```

接收者应该会收到一封 email，包含 300 条被 PatternLayout 格式化的记录事件。下图该

email 在 Mozilla Thunderbird 里的显示。



下一个配置文件 mail2.xml 里，“SMTPHost”、“To”和“From”属性是用变量设置的，下面是 mail2.xml 里的相关内容。

```
<appender name="EMAIL"
class="ch.qos.logback.classic.net.SMTPAppender">
  <SMTPHost>${smtpHost}</SMTPHost>
  <To>${to}</To>
  <From>${from}</From>
  <layout class="ch.qos.logback.classic.html.HTMLLayout" />
</appender>
```

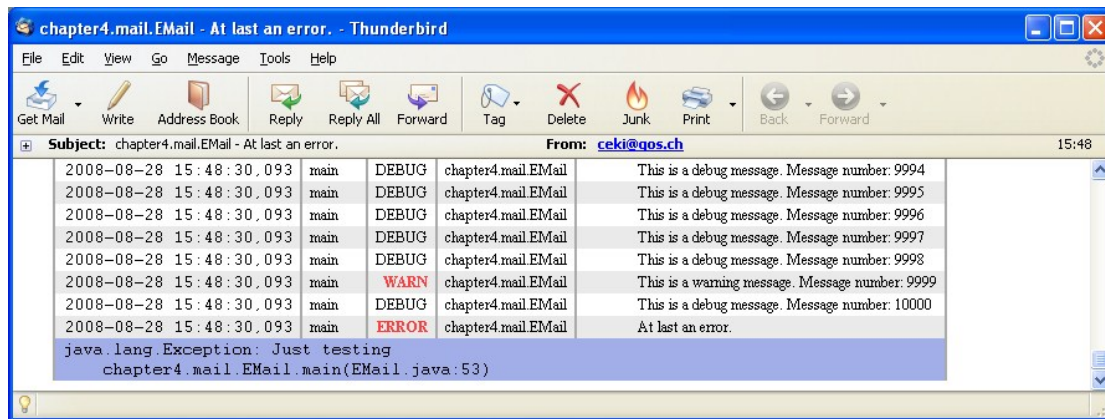
可以在命令行传递所需参数：

```
java -Dfrom=source@xyz.com -Dto=recipient@xyz.com -DsmtpHost=some_smtp_host
chapters.appenders.mail.Email 10000 src/main/java/chapters/appenders/mail/mail2.xml
```

别忘了用自己的值。

注意在上一个例子里，PatternLayout 被 HTMLLayout 替换，HTMLLayout 用 HTML 表格对记录进行格式化。你可以更改表格的列、列的顺序和 CSS，详情参考 HTMLLayout。

由于循环缓冲（BufferSize）默认为 512，所以接收者收到的邮件包含 512 条格式化良好的、放在 HTML 表格里的事件。注意运行 chapters.appenders.mail.Email 后会生成 10000 条记录，但只有最后的 512 条记录会被 email 收纳。



很多电子邮件客户端比如 Mozilla Thunderbird、Eudora 或 MS Outlook，都很好地支持 HTML 邮件里的 CSS。不过，有时客户端们会自动把 HTML 降级为普通文本。例如，要想在 Thunderbird 里查看 HTML 格式的邮件，必须设置“View→Message Body As→Original HTML”。Yahoo!Mail 支持 HTML 邮件，尤其很好地支持 CSS。GMail 支持基本的 HTML 表格而忽略 HTML 里的 CSS，虽然它支持内联（inline）CSS，但是内联 CSS 会使输出结果变得冗长，HTMLLayout 不使用内联 CSS。

#### 4.4.3.1. 事件的触发

如果未指定选项“Evaluator”，则 SMTPAppender 被默认分配一个 OnErrorEvaluator（ch.qos.logback.classic.boolex.OnErrorEvaluator）实例，当遇到级别为 ERROR 或更高级别的事件后，触发 email 传输。虽然遇到错误就触发 email 传输很合理，但也可以通过提供 EventEvaluator 接口的不同实现来覆盖此默认行为。

SMTPAppender 对每个进来的事件都调用 evaluate()方法进行评估，为的是检查事件是应该触发 email 传输还是放入循环缓冲。评估结果为 true 时，就发送 email。SMTPAppender 包含且仅包含一个求值器（evaluator）对象，该对象可以管理自己的内部状态。下面的 CounterBasedEvaluator 类演示了这点，它实现了 EventEvaluator 接口，每收到 1024 条事件就触发一次 email 传输。

示例：EventEvaluator 的一个实现，每收到 1024 条事件就评估为 true

(logback-examples/src/main/java/chapters/appenders/mail/CounterBasedEvaluator.java)

```
package chapters.appenders.mail;

import ch.qos.logback.core.boolex.EvaluationException;
import ch.qos.logback.core.boolex.EventEvaluator;
import ch.qos.logback.core.spi.ContextAwareBase;

public class CounterBasedEvaluator extends ContextAwareBase implements
```

```
EventEvaluator {

    static int LIMIT = 1024;
    int counter = 0;
    String name;
    boolean started;

    public boolean evaluate(Object event) throws NullPointerException,
        EvaluationException {
        counter++;

        if (counter == LIMIT) {
            counter = 0;

            return true;
        } else {
            return false;
        }
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public boolean isStarted() {
        return started;
    }

    public void start() {
        started = true;
    }

    public void stop() {
        started = false;
    }
}
```

注意该类继承 `ContextAwareBase` 且实现 `EventEvaluator`。这样做是让用户专注于 `EventEvaluator` 的核心功能，让基类提供通用功能。



设置“Evaluator”选项会让 SMTPAppender 使用自定义求值器。下面的配置文件为根 logger 关联了一个 SMTPAppender。这个 SMTPAppender 的缓冲大小是 2048，使用上面的 CounterBasedEvaluator 作为其求值器。

示例：自定义求值器和缓冲大小的 SMTPAppender

(logback-examples/src/main/java/chapters/appenders/mail/mail3.xml)

```
<configuration>
  <appender name="EMAIL"
class="ch.qos.logback.classic.net.SMTPAppender">
    <Evaluator
class="chapters.appenders.mail.CounterBasedEvaluator" />
    <BufferSize>1050</BufferSize>
    <SMTPHost>${smtpHost}</SMTPHost>
    <To>${to}</To>
    <From>${from}</From>
    <layout class="ch.qos.logback.classic.html.HTMLLayout" />
  </appender>

  <root>
    <level value="debug" />
    <appender-ref ref="EMAIL" />
  </root>
</configuration>
```

#### 4.4.3.2. 基于标记的触发

根据 ERROR 级别触发 email 可能会导致产生太多 email。Logback 提供了另外的触发策略：OnMarkerEvaluator（ch.qos.logback.classic.boolex.OnMarkerEvaluator）。本质上，只有当事件包含一个用户指定的标记时，才会触发 email。例子如下。

程序 Marked\_Email(chapters.appenders.mail.Email) 包含一些记录语句，部分为 ERROR 级别。注意下面的语句包含一个标记。

```
Marker notifyAdminMarker = MarkerFactory.getMarker("NOTIFY_ADMIN");
logger.error(notifyAdminMarker,
    "This is a serious an error requiring the admin' attention",
    new Exception("Just testing"));
```

下面的配置会导致仅当出现 NOTIFY\_ADMIN 或 TRANSACTION\_FAILURE 时才会触发 email。

示例：使用 OnMarkerEvaluator 的 SMTPAppender

(logback-examples/src/main/java/chapters/appenders/mail/mailWithMarker.xml)

```
<configuration>
  <appender name="EMAIL"
class="ch.qos.logback.classic.net.SMTPAppender">
    <evaluator
class="ch.qos.logback.classic.boolex.OnMarkerEvaluator">
      <marker>NOTIFY_ADMIN</marker>
      <!-- you specify add as many markers as you want -->
      <marker>ANOTHER_MARKER</marker>
    </evaluator>
    <BufferSize>1050</BufferSize>
    <SMTPHost>${smtpHost}</SMTPHost>
    <To>${to}</To>
    <From>${from}</From>
    <layout class="ch.qos.logback.classic.html.HTMLLayout" />
  </appender>

  <root>
    <level value="debug" />
    <appender-ref ref="EMAIL" />
  </root>
</configuration>
```

运行:

```
java -Dfrom=source@xyz.com -Dto=recipient@xyz.com -DsmtpHost=some_smtp_host
chapters.appenders.mail.Marked_EMail
src/main/java/chapters/appenders/mail/mailWithMarker.xml
```

#### 4.4.3.3. 验证/STARTTLS/SSL

SMTPAppender 支持使用明文用户密码的验证, 包含 STARTTLS 和 SSL 协议。注意 STARTTLS 与 SSL 的区别是: STARTTLS 的连接一开始不是加密的, 只有当客户端执行“STARTTLS”命令后(如果服务器支持), 才把连接切换到 SSL; 而 SSL 模式里的连接一开始就是加密的。

#### 4.4.3.4. 针对 Gmail (SSL) 的 SMTPAppender 配置

下例演示了如何配置 SMTPAppender, 以支持使用 SSL 协议的 Gmail。

示例: 支持使用 SSL 的 Gmail 的 SMTPAppender

(logback-examples/src/main/java/chapters/appenders/mail/gmailSSL.xml)

```
<configuration>
```

```

    <appender name="EMAIL"
class="ch.qos.logback.classic.net.SMTPAppender">
      <SMTPHost>smtp.gmail.com</SMTPHost>
      <SMTPPort>465</SMTPPort>
      <SSL>true</SSL>
      <Username>YOUR_USERNAME@gmail.com</Username>
      <Password>YOUR_GMAIL_PASSWORD</Password>

      <To>EMAIL-DESTINATION</To>
      <To>ANOTHER_EMAIL_DESTINATION</To> <!-- additional destinations
are possible -->
      <From>YOUR_USERNAME@gmail.com</From>
      <Subject>TESTING: %logger{20} - %m</Subject>
      <layout class="ch.qos.logback.classic.PatternLayout">
        <Pattern>%date %-5level %logger - %message%n</Pattern>
      </layout>
    </appender>

    <root level="debug">
      <appender-ref ref="EMAIL" />
    </root>
</configuration>

```

#### 4.4.3.5. 针对 Gmail (STARTTLS) 的 SMTPAppender

下例演示了如何配置 SMTPAppender，以支持使用 STARTTLS 协议的 Gmail。

示例：支持使用 STARTTLS 的 Gmail 的 SMTPAppender

(logback-examples/src/main/java/chapters/appenders/mail/gmailSTARTTLS.xml)

```

<configuration>
  <appender name="EMAIL"
class="ch.qos.logback.classic.net.SMTPAppender">
    <SMTPHost>smtp.gmail.com</SMTPHost>
    <SMTPPort>587</SMTPPort>
    <STARTTLS>true</STARTTLS>
    <Username>YOUR_USERNAME@gmail.com</Username>
    <Password>YOUR_GMAIL_PASSWORD</Password>

    <To>EMAIL-DESTINATION</To>
    <To>ANOTHER_EMAIL_DESTINATION</To> <!-- additional destinations
are possible -->
    <From>YOUR_USERNAME@gmail.com</From>

```

```
<Subject>TESTING: %logger{20} - %m</Subject>
<layout class="ch.qos.logback.classic.PatternLayout">
  <Pattern>%date %-5level %logger - %message%n</Pattern>
</layout>
</appender>

<root level="debug">
  <appender-ref ref="EMAIL" />
</root>
</configuration>
```

4.4.4. DBAppender



DBAppender 把记录事件写入数据库的三张表。

三张表分别是 logging\_event、logging\_event\_property 和 logging\_event\_exception。在使用 DBAppender 之前，这三张表必须已经被创建。Logback 提供创建这些表的 SQL 脚本，位于 logback-classic/src/main/java/ch/qos/logback/classic/db/dialect 目录，对各个流行数据库分别有一个脚本。如果没有你想用的数据库的脚本，那就按照现有的脚本自己写吧。

如果你的 JDBC 驱动支持 JDBC 3.0 规范引入的 getGeneratedKeys 方法，那么就不需要额外的步骤，当然你得先创建上面的三张表和进行常规的 logback 配置。

如果不支持 getGeneratedKeys 方法，你必须为你的数据库提供合适的 SQL 方言（SQLDialect）。目前，我们有 PostgreSQL、MySQL、Oracle 和 MS SQL Server 方言。

下表总结了数据库类型及其对 getGeneratedKeys() 的支持。

关系型数据库	测试版本	测试 JDBC 驱动版本	支持 getGeneratedKeys()
DB2	未测试	未测试	未知
HSQL	1.8.0.7	-	不
Microsoft SQL Server	2005	2.0.1008.2 (sqljdbc.jar)	是
MySQL	5.0.22		是
PostgreSQL			不
Oracle	10g	10.2.0.1 (ojdbc14.jar)	是（10.2.0.1）

实验显示，在“标准”PC 上，向数据库写入一条记录耗时大约 10 毫秒。如果使用了连接池，则降到约 1 毫秒。大多数 JDBC 驱动都支持连接池。

有多种方法让 logback 使用 DBAppender，取决于连接数据的工具和数据库本身。配置 DBAppender 的关键问题是设置 ConnectionSource 对象，稍后会讲。

一旦 logback 被正确配置，记录事件就被发送到指定的数据库。前面讲过，logback 用三张表存储记录事件数据。

表 logging\_event 包含下列字段：

字段名	类型	描述
timestamp	big int	创建记录事件的时间戳。
formatted_message	text	经 org.slf4j.impl.MessageFormatter 格式化后，被添加到记录事件的消息。
logger_name	varchar	执行记录请求的 logger。
level_string	varchar	记录事件的级别。
reference_flag	smallint	用于标识那些关联了异常或 MDCproperty 值的记录事件。其值由 ch.qos.logback.classic.db.DBHelper 计算。包含 MDC 或上下文属性的记录事件的 reference_flag 是 1。包含异常的 reference_flag 是 2。两者都包含的 reference_flag 是 3。
caller_filename	varchar	执行记录请求的文件名。
caller_class	varchar	执行记录请求的类。
caller_method	varchar	执行记录请求的方法。
caller_line	char	执行记录请求的行号。
event_id	int	记录事件的数据库 ID。

表 logging\_event\_property 用于存储包含在 MDC 或上下文里的键和值，字段如下：

字段名	类型	描述
event_id	int	记录事件的数据库 ID。
mapped_key	varchar	MDC 属性的键。
mapped_value	varchar	MDC 属性的值。

表 logging\_event\_exception 包含下列字段：

字段名	类型	描述
event_id	int	记录事件的数据库 ID。

i	smallint	完整的堆栈跟踪里的行索引。
trace_line	varchar	对应的行。

下面的截图展示了 DBAppender 的工作，数据库是 MySQL。

表 logging\_event:

timestamp	formatted_message	logger_name	level_string	thread_name	reference_flag	caller_filename	caller_class	caller_method	caller_line	event_id
1162926561498	Preparing for takeoff	main.apollo.thirteen.Launcher	INFO	main	0	ShuttleManager.java	main.ShuttleManager	main	23	31
1162926561576	Engines ready	main.apollo.thirteen.Launcher	DEBUG	main	0	ShuttleManager.java	main.ShuttleManager	main	24	32
1162926561638	T minus 10 and counting	main.apollo.thirteen.Launcher	DEBUG	main	0	ShuttleManager.java	main.ShuttleManager	main	25	33
1162926561701	10. 9. 8. 7. 6. 5. 4. 3. 2	main.apollo.thirteen.Launcher	DEBUG	main	1	ShuttleManager.java	main.ShuttleManager	main	27	34
1162926561748	Ignition	main.apollo.thirteen.Launcher	DEBUG	main	1	ShuttleManager.java	main.ShuttleManager	main	28	35
1162926561794	Counting height	main.apollo.thirteen.Shuttle	DEBUG	main	0	Shuttle.java	apollo.shuttle.Shuttle	<init>	13	36
1162926561941	Dropping reactors	main.apollo.thirteen.Shuttle	DEBUG	main	0	Shuttle.java	apollo.shuttle.Shuttle	<init>	14	37
1162926561973	We're in space, Houston!	main.apollo.thirteen.Shuttle	DEBUG	main	0	Shuttle.java	apollo.shuttle.Shuttle	<init>	15	38
1162926561904	Houston, we have a problem	main.apollo.thirteen.Shuttle	ERROR	main	3	Shuttle.java	apollo.shuttle.Shuttle	oops	20	39

表 logging\_event\_exception:

event_id	mapped_key	mapped_value
34	statusMessage	Everything's fine
35	statusMessage	Everything's fine
39	commanderSays	Hey, I told you not to push that button!

表 logging\_event\_property:

event_id	mapped_key	mapped_value
34	statusMessage	Everything's fine
35	statusMessage	Everything's fine
39	commanderSays	Hey, I told you not to push that button!

#### 4.4.4.1. ConnectionSource

对于 logback 里需要使用 `java.sql.Connection` 的类，`ConnectionSource` 接口提供了一种可插拔地、透明地获取 JDBC 连接的方法。现在有三种具体实现：`DataSourceConnectionSource`、`DriverManagerConnectionSource` 和 `JNDIConnectionSource`。

第一个例子用了 `DriverManagerConnectionSource` 和 MySQL 数据库。

示例：DBAppender 配置

(logback-examples/src/main/java/chapters/appenders/db/append-toMySQL-with-driverManager.xml)

```
<configuration>

  <appender name="DB" class="ch.qos.logback.classic.db.DBAppender">
```

```
<connectionSource
  class="ch.qos.logback.core.db.DriverManagerConnectionSource">
  <driverClass>com.mysql.jdbc.Driver</driverClass>
  <url>jdbc:mysql://host_name:3306/database_name</url>
  <user>username</user>
  <password>password</password>
</connectionSource>
</appender>

<root level="debug">
  <appender-ref ref="DB" />
</root>
</configuration>
```

`DriverManagerConnectionSource` 是 `ConnectionSource` 的一个实现，以传统的 JDBC 方式从连接 URL 里取得连接。

注意 `DriverManagerConnectionSource` 会在每次调用 `getConnection()` 方法时建立一个新连接。建议使用内置连接池的 JDBC 驱动，或者自己实现带有连接池的 `ConnectionSource`。

用 `DataSource` 连接数据库与上面类似。下面的配置里使用 `DataSourceConnectionSource`，以推荐的方式从 `javax.sql.DataSource` 取得连接。

示例：DBAppender 配置

(logback-examples/src/main/java/chapters/appenders/db/append-with-datasource.xml)

```
<configuration>

  <appender name="DB" class="ch.qos.logback.classic.db.DBAppender">
    <connectionSource
      class="ch.qos.logback.core.db.DataSourceConnectionSource">

        <dataSource class="${dataSourceClass}">
          <!--
            Joran cannot substitute variables that are not
            attribute values.
            Therefore, we cannot declare the next parameter like
            the others.
          -->
          <param name="${url-key:-url}" value="${url}" />
          <serverName>${serverName}</serverName>
          <databaseName>${databaseName}</databaseName>
        </dataSource>
      </connectionSource>
    </appender>
  </configuration>
```

```
<user>${user}</user>
<password>${password}</password>
</connectionSource>
</appender>

<root level="debug">
  <appender-ref ref="DB" />
</root>
</configuration>
```

注意在这个例子里,我们大量使用了变量。当 logback 和其他框架需要共享连接信息时,把信息集中放到一个配置里会很方便。

ConnectionSource 的第三个实现是 JNDIConnectionSource, 从 JNDI 变量里取得 javax.sql.DataSource 并用它取得 java.sql.Connection。JNDIConnectionSource 主要设计为用于 J2EE 应用服务器内部或应用服务器的客户端, 假如应用服务器支持原创访问 javax.sql.DataSource。这种方式可以利用连接池和应用服务器提供的其他东西。

```
<connectionSource
class="ch.qos.logback.core.db.JNDIConnectionSource">
  <param name="jndiLocation" value="jdbc/MySQLDS" />
  <param name="username" value="myUser" />
  <param name="password" value="myPassword" />
</connectionSource>
```

该类用无参数构造方法取得一个 javax.naming.InitialContext, 如果运行在 J2EE 环境内部, 通常没问题。当运行在 J2EE 环境外时, 请按照你的 JNDI 提供商的文档提供一份 jndi.properties 文件。

#### 4.4.4.2. 连接池

记录事件有时创建得很快。为了来得及插入数据库, 推荐为 DBAppender 使用连接池。

实验显示 DBAppender 用了连接池后, 性能得到巨大提升。

下面的配置文件把记录事插入 MySQL 数据库, 没有用连接池。

示例: 没有连接池的 DBAppender 配置

(logback-examples/src/main/java/chapters/appenders/db/append-toMySQL-with-datasource.xml)

```
<configuration>

  <appender name="DB" class="ch.qos.logback.classic.db.DBAppender">
```



```

        <connectionSource
class="ch.qos.logback.core.db.DataSourceConnectionSource">
        <dataSource
class="com.mysql.jdbc.jdbc2.optional.MysqlDataSource">
            <serverName>${serverName}</serverName>
            <port>${port}</port>
            <databaseName>${dbName}</databaseName>
            <user>${user}</user>
            <password>${pass}</password>
        </dataSource>
    </connectionSource>
</appender>

<root level="debug">
    <appender-ref ref="DB" />
</root>

</configuration>

```

用这个配置文件，向 MySQL 数据库发送 500 条记录事件耗时 5 秒，即每个请求 10 毫秒。当处理大型应用时，这个数字不可接受。

为 DBAppender 使用连接池需要用外部类库。下面的例子用了 c3p0 (<http://sourceforge.net/projects/c3p0>)。

示例：用了连接池的 DBAppender

(logback-examples/src/main/java/chapters/appenders/db/append-toMySQL-with-datasource-and-pooling.xml)

```

<configuration>

    <appender name="DB" class="ch.qos.logback.classic.db.DBAppender">
        <connectionSource
class="ch.qos.logback.core.db.DataSourceConnectionSource">
            <dataSource
class="com.mchange.v2.c3p0.ComboPooledDataSource">
                <driverClass>com.mysql.jdbc.Driver</driverClass>

                <jdbcUrl>jdbc:mysql://localhost:3306/logbackdb</jdbcUrl>
                <user>logback</user>
                <password>logback</password>
            </dataSource>
        </connectionSource>
    </appender>

```

```
</appender>

<root level="debug">
  <appender-ref ref="DB" />
</root>

</configuration>
```

用这个配置, 向 MySQL 数据库发送 500 条记录事件耗时 0..5 秒, 大约每个请求 1 毫秒, 性能提升了 10 倍。

4.4.5. SyslogAppender

syslog 协议很简单: syslog 发送者发送一小段信息到 syslog 服务器。接收者一般称为“syslog 守护进程”或“syslog 服务器”。Logback 能向远程 syslog 守护进程发送消息。实现办法是 SyslogAppender。

下面是 SyslogAppender 的属性。

属性名	类型	描述
SyslogHost	String	syslog 服务器的主机名。
Port	String	syslog 服务器的监听端口。默认为 514
Facility	String	用于标识消息的来源。 只允许下列值, 大小写无关: KERN、USER、MAIL、DAEMON、AUTH、SYSLOG、LPR、NEWS、UUCP、CRON、AUTHPRIV、FTP、NTP、AUDIT、ALERT、CLOCK、LOCAL0、LOCAL1、LOCAL2、LOCAL3、LOCAL4、LOCAL5、LOCAL6、LOCAL7。
SuffixPattern	String	表示发向 syslog 服务器的消息中的非标准部分的格式。默认为 “[%thread] %logger %msg”。任何可用于 PatternLayout 的值都可用。

记录事件在 syslog 里的严重程度是从记录事件的级别转换而来的。DEBUG 级别转换为 7, INFO 是 6, WRAN 是 4, ERROR 是 3。

由于 syslog 请求的格式遵循相当严格的规则, 所以 SyslogAppender 没有用到任何 layout。但是, 用 “SuffixPattern” 选项能显示用户所希望看到的任何信息。

下面是 SyslogAppender 的一个例子。

示例: SyslogAppender 配置

([logback-examples/src/main/java/chapters/appenders/conf/logback-syslog.xml](#))

```
<configuration>

  <appender name="SYSLOG"
class="ch.qos.logback.classic.net.SyslogAppender">
    <syslogHost>${syslogHost}</syslogHost>
    <facility>${facility}</facility>
    <suffixPattern>%-4relative [%thread] %-5level
- %msg</suffixPattern>
  </appender>

  <root level="DEBUG">
    <appender-ref ref="SYSLOG" />
  </root>
</configuration>
```

测试这个配置时，请先保证远程 syslog 守护进程可以接收外部请求。经验显示，默认情况下，syslog 守护进程通常拒绝来自网络连接的请求。



#### 4.4.6. SiftingAppender

如其名，SiftingAppender 能按照给定的运行时属性，对记录进行分离或筛选。例如，SiftingAppender 能根据用户会话对记录事件进行分离，这样，每个用户生成的记录会进入不同的记录文件，一个用户一个文件。

SiftingAppender 内置并且管理多个 appender，这些 appender 是按照区别值(discriminating value)而动态构建的。在配置文件里的 SiftingAppender 定义的内部，指定构建好的 appender。默认情况下，SiftingAppender 用 MDC 的键/值对作为鉴别器 (discriminator)。

配置 logback 后，SiftExample 程序 (chapters.appenders.sift.SiftExample) 记录一条消息，说明程序已经开始，接着设置 MDC 键 “userid” 的值为 “Alice” 并且记录一条消息。下面是关键代码：

```
logger.debug("Application started");
MDC.put("userid", "Alice");
logger.debug("Alice says hello");
```

下面的配置文件演示了 SiftingAppender 的用法。

示例：SiftingAppender 配置

(logback-examples/src/main/java/chapters/appenders/sift/byUserId.xml)

```
<configuration>
  <appender name="SIFT"
```

```
class="ch.qos.logback.classic.sift.SiftingAppender">
  <discriminator>
    <Key>userid</Key>
    <DefaultValue>unknown</DefaultValue>
  </discriminator>
  <sift>
    <appender name="FILE-`${userid}"
class="ch.qos.logback.core.FileAppender">
      <File>${userid}.log</File>
      <Append>false</Append>
      <layout class="ch.qos.logback.classic.PatternLayout">
        <Pattern>%d [%thread] %level %mdc %logger{35}
- %msg%n</Pattern>
      </layout>
    </appender>
  </sift>
</appender>

<root level="DEBUG">
  <appender-ref ref="SIFT" />
</root>
</configuration>
```

如果没有指定 class 属性，则默认为“MDCBasedDiscriminator”（ch.qos.logback.classic.sift.MDCBasedDiscriminator），它用“Key”属性关联的 MDC 值作为鉴别器。如果值为 null，则使用“DefaultValue”关联的值。

SiftingAppender 是唯一能够引用和配置嵌套 appender 的 appender。在上面的例子里，在 SiftingAppender 里有嵌套的 FileAppender 实例，每个实例都用 MDC 键“userid”所关联的值作标识。每当 MDC 键“userid”关联一个新值时，就会从头创建一个新的 FileAppender 实例。SiftingAppender 监视被它创建的 appender，如果 appender 在 30 分钟内没被使用，则会被自动关闭和丢弃。

仅拥有不同的 appender 实例是不够得，每个实例还必须输出到不同的资源。为实现这种差异，在嵌套 appender（上面的 FileAppender）里，传递给鉴别器的键（上面的“userid”）变成了变量。所以，这个变量可用来区分嵌套 appender 所使用的实际资源。

用上面的配置文件“byUserId.xml”运行 SiftExample 程序，将产生两个记录文件：

unknown.log 和 Alice.log。

### 4.4.7. 自定义 Appender

继承 AppenderBase 就可以轻松地写自己的 appender。AppenderBase 处理过滤器、状态信息和大多数 appender 共享的其他功能。派生类只需要实现一个方法：append(Object eventObject)。

下面的 CountingConsoleAppender，在控制台输出有数量限制的事件。如果到达限制值就不输出任何东西。它用 Layout 来对事件进行格式化，还有一个参数“limit”，因此需要增加一些方法。

示例：CountingConsoleAppender

(logback-examples/src/main/java/chapters/appenders/CountingConsoleAppender.java)

```
package chapters.appenders;

import java.io.IOException;

import ch.qos.logback.classic.encoder.PatternLayoutEncoder;
import ch.qos.logback.classic.spi.ILoggingEvent;
import ch.qos.logback.core.AppenderBase;

public class CountingConsoleAppender extends
AppenderBase<ILoggingEvent> {
    static int DEFAULT_LIMIT = 10;
    int counter = 0;
    int limit = DEFAULT_LIMIT;

    PatternLayoutEncoder encoder;

    public CountingConsoleAppender() {
    }

    public void setLimit(int limit) {
        this.limit = limit;
    }

    public int getLimit() {
        return limit;
    }
}
```

```
@Override
public void start() {
    if (this.encoder == null) {
        addError("No layout set for the appender named [" + name + "].");
        return;
    }

    try {
        encoder.init(System.out);
    } catch (IOException e) {
    }

    super.start();
}

public void append(ILoggingEvent event) {
    if (counter >= limit) {
        return;
    }
    // output the events as formatted by our layout
    try {
        this.encoder.doEncode(event);
    } catch (IOException e) {
    }

    // prepare for next event
    counter++;
}

public PatternLayoutEncoder getEncoder() {
    return encoder;
}

public void setEncoder(PatternLayoutEncoder encoder) {
    this.encoder = encoder;
}
}
```

start()方法检查是否有 encoder，如果没有 encoder，则不能启动并输出错误消息。

这个自定义 appender 阐述了两点：

- 所有遵循 JavaBeans 的 setter/getter 之约定属性，都能被透明地处理。在 logback 配置期间自动被调用的 start()方法，有责任核实 appender 的各个属性是一致的。

- `AppenderBase.doAppend()` 调用其派生类的 `append()` 方法。实际输出操作发生在 `append()` 方法，特别是，`appender` 是在这个方法里调用其 `layout/encoder` 完成格式化的。

配置 `CountingConsoleAppender` 和配置其他任何 `appender` 一样。参见配置例子“`logback-examples/src/main/java/chapters/appenders/countingConsole.xml`”。

## 4.5. Logback Access

Logback-classic 里的大多数 `appender` 在 `logback-access` 都有其等价物，工作方式本质上也一样。

### 4.5.1. SocketAppender

`SocketAppender` 被设计为把序列化的 `AccessEvent` 对象通过网络传输到远程实体。就访问事件远程而言，远程记录是无损的。被接收端反向序列化后，事件能像从本地产生的一样被记录。

`SocketAppender` 的属性与 `logback-classic` 里的 `SocketAppender` 一样。

### 4.5.2. SMTPAppender

`SMTPAppender` 的工作方式与 `logback-classic` 里的一样。不过，属性“`evaluator`”略有不同。默认情况下，`SMTPAppender` 使用 `URLEvaluator` 对象。`URLEvaluator` 包含一个检查当前请求的 URL 的 URL 列表。当 `URLEvaluator` 里的页面被访问时，`SMTPAppender` 就发送 email。

下面是 `logback-access` 环境下的 `SMTPAppender` 配置文件例子。

示例：SMTPAppender 配置

(`logback-examples/src/main/java/chapters/appenders/conf/access/logback-smtp.xml`)

```
<configuration>

  <appender name="SMTP"
    class="ch.qos.logback.access.net.SMTPAppender">
    <layout class="ch.qos.logback.access.html.HTMLLayout">
      <Pattern>%h%l%u%t%r%s%b</Pattern>
    </layout>
  </appender>
</configuration>
```

```
</layout>

<b>
    <Evaluator class="ch.qos.logback.access.net.URLEvaluator">
        <URL>url1.jsp</URL>
        <URL>directory/url2.html</URL>
    </Evaluator>
</b>
<From>sender_email@host.com</From>
<SMTPHost>mail.domain.com</SMTPHost>
<To>recipient_email@host.com</To>
</appender>

<appender-ref ref="SMTP" />
</configuration>
```

这种触发 email 的方法允许用户选择在某个流程里的重要页面，例如，当某个重要页面被访问时，email 就会被发送，email 包含被访问的页面和用户想放在 email 里的任何其他信息。

### 4.5.3. DBAppender

DBAppender 用于把访问事件插入数据库。

DBAppender 用到两张表：access\_event 和 access\_event\_header。使用 DBAppender 前，必须先创建这两张表。Logback 提供创建这些表的 SQL 脚本，位于 logback-access/src/main/java/ch/qos/logback/access/db/dialect 目录，对各个流行数据库分别有一个脚本。如果没有你想用的数据库的脚本，那就按照现有的脚本自己写吧。

表 access\_event 包含的字段：

字段名	类型	描述
timestamp	big int	创建访问事件的时间戳。
requestURI	varchar	请求的 URI。
requestURL	varchar	请求的 URL。包括请求方法、请求 URI 和请求协议。
remoteHost	varchar	远程主机名。
remoteUser	varchar	远程用户名。
remoteAddr	varchar	远程 IP 地址。
protocol	varchar	请求协议，如 HTTP 或 HTTPS。
method	varchar	请求方法，一般是 GET 或 POST。



serverName	varchar	执行请求的服务器名。
event_id	int	访问事件的数据库 ID。

表 access\_event 包含的字段:

字段名	类型	描述
event_id	int	访问事件的数据库 ID。
header_key	varchar	消息头名称, 比如 “User-Agent”。
header_value	varchar	消息头值, 比如 “Mozilla/5.0 (Windows; U; Windows NT 5.1; fr; rv:1.8.1) Gecko/20061010 Firefox/2.0”。

logback-classic 里的 DBAppender 的全部属性都适用于 logback-access 的 DBAppender, 后者多了一个属性:

字段名	类型	描述
insertHeaders	boolean	是否让 DBAppender 把所有请求的头信息都放进数据库。

下面是使用 DBAppender 的配置例子。

示例: DBAppender 配置

(logback-examples/src/main/java/chapters/appenders/conf/access/logback-DB.xml)

```
<configuration>

  <appender name="DB" class="ch.qos.logback.access.db.DBAppender">
    <connectionSource
      class="ch.qos.logback.core.db.DriverManagerConnectionSource">
        <driverClass>com.mysql.jdbc.Driver</driverClass>
        <url>jdbc:mysql://localhost:3306/logbackdb</url>
        <user>logback</user>
        <password>logback</password>
      </connectionSource>
      <insertHeaders>true</insertHeaders>
    </appender>

    <appender-ref ref="DB" />
  </configuration>
```

### 4.5.4. SiftingAppender

logback-access 的 SiftingAppender 与 logback-classic 里的非常相似。主要的区别是，logback-access 的默认鉴别器（discriminator）AccessEventDiscriminator 不是基于 MDC 的。如其名，AccessEventDiscriminator 使用 AccessEvent 的指定字段作为选择嵌套 appender 的基础。如果字段值为 null，则使用 “DefaultValue” 属性的值。

AccessEvent 的指定字段可以是下列字符串之一：COOKIE、REQUEST\_ATTRIBUTE、SESSION\_ATTRIBUTE、REMOTE\_ADDRESS、LOCAL\_PORT、REQUEST\_URI。注意前三个字段需要同时指定 “AdditionalKey” 属性。

下面是配置文件例子。

示例：SiftingAppender 配置

(logback-examples/src/main/java/chapters/appenders/conf/sift/access-siftingFile.xml)

```
<configuration>

  <appender name="SIFTING"
class="ch.qos.logback.access.sift.SiftingAppender">
    <Discriminator>
      <Key>id</Key>
      <FieldName>SESSION_ATTRIBUTE</FieldName>
      <AdditionalKey>username</AdditionalKey>
      <DefaultValue>NA</DefaultValue>
    </Discriminator>
    <sift>
      <appender name="${id}"
class="ch.qos.logback.core.FileAppender">
        <File>byUser/${id}.log</File>
        <layout class="ch.qos.logback.access.PatternLayout">
          <Pattern>%h %l %u %t \"%r\" %s %b</Pattern>
        </layout>
      </appender>
    </sift>
  </appender>

  <appender-ref ref="SIFTING" />
</configuration>
```

在上面的配置里，SiftingAppender 嵌套了 FileAppender 实例。键 “id” 作为变量，对嵌套的 FileAppender 实例可用。默认的鉴别器 AccessEventDiscriminator，会在每个 AccessEvent 里查找 “username” 会话属性，如果找不到，则用默认值 “NA”。因此，假设会话属性 “username”

包含每个已登录用户的用户名，则在当前目录的 `byUser/` 目录下，每个用户的访问记录都分别记录在以用户名命名的文件。

## 5. Encoder

### 5.1. 什么是 encoder

Encoder 负责两件事，一是把事件转换为字节数组，二是把字节数组写入输出流。在 logback 0.9.19 版之前没有 encoder。在之前的版本里，多数 appender 依靠 layout 来把事件转换成字符串并用 `java.io.Writer` 把字符串输出。在之前的版本里，用户需要在 `FileAppender` 里嵌入一个 `PatternLayout`。而从 0.9.19 版开始，`FileAppender` 和其子类使用 encoder，不接受 layout。

为什么变了？

Layout，之后的章节会讲到，只负责把事件转换为字符串。此外，因为 layout 不能控制事件何时被写出，所以不能成批地聚集事件。相比之下，encoder 不但可以完全控制待写出的字节的格式，而且可以控制字节何时及是否被写出。

目前，`PatternLayoutEncoder` 是唯一有用的 encoder，它基本上是封装了 `PatternLayout`，让 `PatternLayout` 负责大多数工作。因此，似乎 encoder 并没有带来多少好东西，反而只有不需要的复杂性。然而，我们希望当新的、强大的 encoder 到来时，这种印象会改变。

### 5.2. Encoder 接口

Encoder 负责把事件转换为字节数组并把字节数组写出到合适的输出流。因此，encoder 可以完全控制在什么时候、把什么样的字节写入到由其拥有者 appender 维护的输出流。下面是 Encoder 接口：

```
package ch.qos.logback.core.encoder;

import java.io.IOException;
import java.io.OutputStream;

import ch.qos.logback.core.spi.ContextAware;
import ch.qos.logback.core.spi.Lifecycle;

public interface Encoder<E> extends ContextAware, Lifecycle {

    /**
     * This method is called when the owning appender starts or
     * whenever output needs to be directed to a new OutputStream,
     * for instance as a result of a rollover.
     */
}
```

```
    */
    void init(OutputStream os) throws IOException;

    /**
     * Encode and write an event to the appropriate {@link OutputStream}.
     * Implementations are free to differ writing out of the encoded
     * event and instead write in batches.
     */
    void doEncode(E event) throws IOException;

    /**
     * This method is called prior to the closing of the underlying
     * {@link OutputStream}. Implementations MUST not close the underlying
     * {@link OutputStream} which is the responsibility of the
     * owning appender.
     */
    void close() throws IOException;
}
```

Encoder 接口由很少的方法组成，但这些方法却可以做许多有用的事情。

### 5.3. LayoutWrappingEncoder

在 logback 0.9.19 版之前，appender 依赖 layout 提供输出格式的灵活性。因为有大量现存代码是基于 layout 接口的，所以我们需要想办法让 encoder 与 layout 实现互操作。LayoutWrappingEncoder 连接了 encoder 和 layout，它实现 encoder 接口，并且包裹了一个 layout，layout 负责把事件转换成字符串。

下面是 LayoutWrappingEncoder 类的代码片段，阐述了如何把工作委托给 layout 实例。

```
package ch.qos.logback.core.encoder;

import java.io.IOException;
import java.nio.charset.Charset;

import ch.qos.logback.core.Layout;

public class LayoutWrappingEncoder<E> extends EncoderBase<E> {

    protected Layout<E> layout;
    private Charset charset;
```

```
public void doEncode(E event) throws IOException {
    String txt = layout.doLayout(event);
    outputStream.write(convertToBytes(txt));
    outputStream.flush();
}

private byte[] convertToBytes(String s) {
    if (charset == null) {
        return s.getBytes();
    } else {
        return s.getBytes(charset);
    }
}
}
```

doEncode()方法首先让被包裹的 layout 把传入的事件转换成字符串，再根据用户选择的字符集编码把字符串转换成字节，然后把自己写入其拥有者 appender 指定的输出流，输出流被立即冲出（flush）。

## 5.4. PatternLayoutEncoder

既然 PatternLayout 是最常用的 layout，logback 便提供了 PatternLayoutEncoder，它扩展了 LayoutWrappingEncoder，且仅使用 PatternLayout。

从 logback 0.9.19 版起，FileAppender 或其子类在只要用到 PatternLayout 时，都必须换成 PatternLayoutEncoder。

## 6. 排版 (Layout)

### 6.1. 什么是 layout

Layout 负责把事件转换成字符串。Layout 接口的 `format()` 方法的参数是代表任何类型的事件，返回字符串。Layout 接口的概要如下：

```
public interface Layout<E> extends ContextAware, Lifecycle {

    String doLayout(E event);
    String getFileHeader();
    String getPresentationHeader();
    String getFileFooter();
    String getPresentationFooter();
    String getContentType();
}
```

接口很简单却足够完成很多格式化需求。

### 6.2. Logback-classic

Logback-classic 只处理 `ch.qos.logback.classic.spi.ILoggingEvent` 类型的事件。

#### 6.2.1. 自定义 layout

让我们实现一个简单却可工作的 layout，打印内容包括：自程序启动以来逝去的时间、记录事件的级别、包含在方括号里的调用者线程的名字、logger 名、连字符、事件消息和换行。

输出类似于：

```
10489 DEBUG [main] com.marsupial.Pouch - Hello world.
```

下面是一个例子。

示例：Layout 示例实现

(`logback-examples/src/main/java/chapters/layouts/MySampleLayout.java`)

```
package chapters.layouts;

import ch.qos.logback.classic.spi.ILoggingEvent;
import ch.qos.logback.core.CoreConstants;
```

```
import ch.qos.logback.core.LayoutBase;

public class MySampleLayout extends LayoutBase<ILoggingEvent> {

    public String doLayout(ILoggingEvent event) {
        StringBuffer sbuf = new StringBuffer(128);
        sbuf.append(event.getTimestamp()
            - event.getLoggerContextVO().getBirthTime());
        sbuf.append(" ");
        sbuf.append(event.getLevel());
        sbuf.append(" [");
        sbuf.append(event.getThreadName());
        sbuf.append("] ");
        sbuf.append(event.getLoggerName());
        sbuf.append(" - ");
        sbuf.append(event.getFormattedMessage());
        sbuf.append(CoreConstants.LINE_SEPARATOR);
        return sbuf.toString();
    }
}
```

MySampleLayout 继承 LayoutBase。LayoutBase 类管理对所有 layout 实例通用的状态，比如 layout 是启动还是停止、header、footer 和 content type 数据。LayoutBase 类允许开发者在自己的 layout 里实现具体的格式化方式。LayoutBase 类是泛型的，MySampleLayout 继承 LayoutBase<ILoggingEvent>。

MySampleLayout 类里唯一的方法 doLayout(ILoggingEvent event)，一开始先初始化一个 StringBuffer，接着添加 event 参数的各种字段，然后把 StringBuffer 转换成 String，最后返回这个 String。

在上面的例子里，doLayout 方法忽略了 event 参数里的任何异常。在真实世界里，你或许想把异常的内容也打印出来。

#### 6.2.1.1. 配置自定义 layout

配置自定义 layout 与配置其他 layout 是一样的。FileAppender 和其子类需要一个 encoder。为了满足这个需求，我们把包裹了 MySampleLayout 的 LayoutWrappingEncoder 实例传递给 FileAppender。下面是配置文件：

示例：MySampleLayout 的配置

(logback-examples/src/main/java/chapters/layouts/sampleLayoutConfig.xml)

```
<configuration debug="true">
```



```
<appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
  <encoder
class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
  <layout class="chapters.layouts.MySampleLayout" />
  </encoder>
</appender>

<root level="DEBUG">
  <appender-ref ref="STDOUT" />
</root>
</configuration>
```

示例程序 `chapters.layouts.SampleLogging` 输出一条 `debug` 信息, 然后输出一条 `error` 消息。

运行:

```
java chapters.layouts.SampleLogging src/main/java/chapters/layouts/sampleLayoutConfig.xml
```

输出:

```
0 DEBUG [main] chapters.layouts.SampleLogging - Everything's going well
0 ERROR [main] chapters.layouts.SampleLogging - maybe not quite...
```

很简单吧。

如何为 `layout` 增加选项? 为 `layout` 或任何 `logback` 的其他组件添加属性非常简单: 声明一个属性及 `setter` 方法接即可。 `MySampleLayout2` 类包含两个属性。第一个是为输出添加的前缀。第二个是用于选择是否显示记录请求的线程名。

下面是 `MySampleLayout2` 类:

```
package chapters.layouts;

import ch.qos.logback.classic.spi.ILoggingEvent;
import ch.qos.logback.core.CoreConstants;
import ch.qos.logback.core.LayoutBase;

public class MySampleLayout2 extends LayoutBase<ILoggingEvent> {

    String prefix = null;
    boolean printThreadName = true;

    public void setPrefix(String prefix) {
        this.prefix = prefix;
    }
}
```

```
}

public void setPrintThreadName(boolean printThreadName) {
    this.printThreadName = printThreadName;
}

public String doLayout(ILoggingEvent event) {
    StringBuffer sbuf = new StringBuffer(128);
    if (prefix != null) {
        sbuf.append(prefix + ": ");
    }
    sbuf.append(event.getTimestamp()
        - event.getLoggerContextVO().getBirthTime());
    sbuf.append(" ");
    sbuf.append(event.getLevel());
    if (printThreadName) {
        sbuf.append(" [");
        sbuf.append(event.getThreadName());
        sbuf.append("] ");
    } else {
        sbuf.append(" ");
    }
    sbuf.append(event.getLoggerName());
    sbuf.append(" - ");
    sbuf.append(event.getFormattedMessage());
    sbuf.append(CoreConstants.LINE_SEPARATOR);
    return sbuf.toString();
}
}
```

在配置里启用属性只需要有对应的 `setter` 方法即可。下面是 `MySampleLayout2` 所用的配置文件。

```
<configuration>

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

        <encoder
class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
            <layout class="chapters.layouts.MySampleLayout2">
                <prefix>MyPrefix</prefix>
                <printThreadName>false</printThreadName>
            </layout>
        </encoder>
    </appender>
</configuration>
```

```
        </encoder>
    </appender>

    <root level="debug">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

### 6.2.2. PatternLayout

与所有 Layout 一样，PatternLayout 的参数是一个记录时间并返回一个字符串，但是被返回的字符串可以通过 PatternLayout 的转换模式进行任意定制。

PatternLayout 的转换模式与 C 语言里的 printf() 的转换模式很接近。转换模式是由文本和格式控制表达式（称为格式转换符（conversion specifier））组成。你可以在格式转换符内插入任意文本。每个格式转换符以 “%” 开头，接着是可选的格式修饰符（format modifier）、一个转换符（conversion word）和放在括号里的其他可选参数。转换符控制待转换的数据字段，比如 logger 名、级别、日期或线程名。格式修饰符控制字段的宽度、填充和左右对齐方式。

已经讲过几次了，FileAppender 及其子类需要一个 encoder。所以，当用于 FileAppender 或其子类时，PatternLayout 必须被包裹在一个 encoder 里。由于 FileAppender 与 PatternLayout 的组合用法太常用了，所以 logback 提供了一个名为 “PatternLayoutEncoder” 的 encoder，它唯一的设计目标就是包裹一个 PatternLayout 实例，以便 PatternLayout 能被看作一个 encoder。下面的例子里，用编程方式为 ConsoleAppender 配置了一个 PatternLayoutEncoder：

示例：PatternLayout 用法

(logback-examples/src/main/java/chapters/layouts/PatternSample.java)

```
package chapters.layouts;

import org.slf4j.LoggerFactory;

import ch.qos.logback.classic.Logger;
import ch.qos.logback.classic.LoggerContext;
import ch.qos.logback.classic.encoder.PatternLayoutEncoder;
import ch.qos.logback.classic.spi.ILoggingEvent;
import ch.qos.logback.core.ConsoleAppender;

public class PatternSample {
```

```
static public void main(String[] args) throws Exception {
    Logger rootLogger = (Logger) LoggerFactory.getLogger("root");
    LoggerContext loggerContext = rootLogger.getLoggerContext();
    loggerContext.reset();

    PatternLayoutEncoder encoder = new PatternLayoutEncoder();
    encoder.setContext(loggerContext);
    encoder.setPattern("%-5level [%thread]: %message%n");
    encoder.start();

    ConsoleAppender<ILoggingEvent> appender = new
ConsoleAppender<ILoggingEvent>();
    appender.setContext(loggerContext);
    appender.setEncoder(encoder);
    appender.start();

    rootLogger.addAppender(appender);

    rootLogger.debug("Message 1");
    rootLogger.warn("Message 2");
}
}
```

上例中，转换模式是“%-5level [%thread]: %message%n”，运行：

```
java java chapters.layouts.PatternSample
```

输出：

```
DEBUG [main]: Message 1
WARN  [main]: Message 2
```

注意在转换模式“%-5level [%thread]: %message%n”里，文本文字与转换符之间没有分隔符。当解析转换模式时，PatternLayout 有能力区分文本文字（空格、方括号、冒号）和转换符。在上面的例子里，转换符“%-5level”表示记录事件的级别应该左对齐且宽为 5 个字符。

在 PatternLayout 里，圆括号表示对转换模式进行分组。“(”和“)”有特殊含义，如果想把它们作为文本文字，就必须进行转义。圆括号的特殊含义会在下面讲到。

前面说过，某些格式转换符可以包含放在花括号（{}）可选参数，比如“%logger{10}”，其中，“logger”是转换符，“10”是选项。更多选项会在下面讲到。

6.2.3. 转换符说明

下表列出了转换符及其选项。当单元格里同时列出多个转换符时，它们就是同义词。

转换符	作用																								
c{length} lo{length} logger{length}	<p>输出源记录事件的 logger 名。</p> <p>可以有一个整数型的参数，功能是缩短 logger 名。设为“0”表示只输出 logger 名里最右边的点号之后的字符串。下表是缩写算法例子。</p> <table><tr><th>格式转换符</th><th>logger 名</th><th>结果</th></tr><tr><td>%logger</td><td>mainPackage.sub.sample.Bar</td><td>mainPackage.sub.sample.Bar</td></tr><tr><td>%logger{0}</td><td>mainPackage.sub.sample.Bar</td><td>Bar</td></tr><tr><td>%logger{5}</td><td>mainPackage.sub.sample.Bar</td><td>m.s.s.Bar</td></tr><tr><td>%logger{10}</td><td>mainPackage.sub.sample.Bar</td><td>m.s.s.Bar</td></tr><tr><td>%logger{15}</td><td>mainPackage.sub.sample.Bar</td><td>m.s.sample.Bar</td></tr><tr><td>%logger{16}</td><td>mainPackage.sub.sample.Bar</td><td>m.sub.sample.Bar</td></tr><tr><td>%logger{26}</td><td>mainPackage.sub.sample.Bar</td><td>mainPackage.sub.sample.Bar</td></tr></table> <p>注意最右边的 logger 名永远不被省略，即使它的长度超过了“length”选项。logger 名里的其他片段可以被缩短为至少 1 个字符，但永远不会消失。</p>	格式转换符	logger 名	结果	%logger	mainPackage.sub.sample.Bar	mainPackage.sub.sample.Bar	%logger{0}	mainPackage.sub.sample.Bar	Bar	%logger{5}	mainPackage.sub.sample.Bar	m.s.s.Bar	%logger{10}	mainPackage.sub.sample.Bar	m.s.s.Bar	%logger{15}	mainPackage.sub.sample.Bar	m.s.sample.Bar	%logger{16}	mainPackage.sub.sample.Bar	m.sub.sample.Bar	%logger{26}	mainPackage.sub.sample.Bar	mainPackage.sub.sample.Bar
格式转换符	logger 名	结果																							
%logger	mainPackage.sub.sample.Bar	mainPackage.sub.sample.Bar																							
%logger{0}	mainPackage.sub.sample.Bar	Bar																							
%logger{5}	mainPackage.sub.sample.Bar	m.s.s.Bar																							
%logger{10}	mainPackage.sub.sample.Bar	m.s.s.Bar																							
%logger{15}	mainPackage.sub.sample.Bar	m.s.sample.Bar																							
%logger{16}	mainPackage.sub.sample.Bar	m.sub.sample.Bar																							
%logger{26}	mainPackage.sub.sample.Bar	mainPackage.sub.sample.Bar																							
C{length} class{length}	<p>输出执行记录请求的调用者的全限定类名。</p> <p>和上面的“%logger”一样，也有“length”属性，表示缩短类名。“length”为“0”表示不输出包名。默认输出类的全限定名。</p> <p>输出调用者的类信息并不很快，所以尽量避免使用，除非执行速度不造成任何问题。</p>																								
contextName cn	输出事件源头关联的 logger 的 logger 上下文的名称。																								
d{pattern} date{pattern}	<p>输出记录事件的日期。该转换符有可选的模式字符串选项。模式语法与 java.text.SimpleDateFormat 的格式兼容。</p> <p>可以为 ISO8061 日期格式指定字符串“ISO8601”。如果没有模式选项，则默认为 ISO 8601 日期格式。</p> <p>下面是一些选项值的例子。</p> <table><tr><th>格式转换符</th><th>结果</th></tr><tr><td>%d</td><td>2006-10-20 14:06:49,812</td></tr></table>	格式转换符	结果	%d	2006-10-20 14:06:49,812																				
格式转换符	结果																								
%d	2006-10-20 14:06:49,812																								

	<code>%date</code>	2006-10-20 14:06:49,812
	<code>%date{ISO8601}</code>	2006-10-20 14:06:49,812
	<code>%date{HH:mm:ss.SSS}</code>	14:06:49.812
	<code>%date{dd MMM yyyy ;HH:mm:ss.SSS}</code>	20 oct. 2006;14:06:49.812
	<p>该转换符还可以有第二个选项：时区。因此，“<code>date{HH:mm:ss.SSS,Australia/Perth}</code>”会输出澳大利亚珀斯城所在时区的时间。</p> <p>由于逗号“,”是选项分隔符，所以“<code>[HH:mm:ss,SSS]</code>”会输出“SSS”时区的时间，但是“SSS”时区并不存在，因此会用默认的 GMT 时区输出时间。如果想在日期模式里使用逗号，可以用引号包含之，例如<code>%date{"HH:mm:ss,SSS"}</code>。</p>	
F file	<p>输出执行记录请求的 Java 源文件的文件名。</p> <p>输出文件信息并不很快，所以尽量避免使用，除非执行速度不造成任何问题。</p>	
caller{depth} caller{depth, evaluator-1,... evaluator-n}	<p>输出生成记录事件的调用者的位置信息。</p> <p>位置信息依赖 JVM 实现，但通常由调用方法的全限定名、放在括号里的文件名和行号。</p> <p>该转换符可以有一个整数选项，表示显示信息的深度。</p> <p>例如，<code>%caller{2}</code>会输出：</p> <pre> 0    [main] DEBUG - logging statement Caller+0    at mainPackage.sub.sample.Bar.sampleMethodName(Bar.java:22) Caller+1    at mainPackage.sub.sample.Bar.createLoggingRequest(Bar.java:17) </pre> <p><code>%caller{3}</code>会输出：</p> <pre> 16   [main] DEBUG - logging statement Caller+0    at mainPackage.sub.sample.Bar.sampleMethodName(Bar.java:22) Caller+1    at mainPackage.sub.sample.Bar.createLoggingRequest(Bar.java:17) Caller+2    at mainPackage.ConfigTester.main(ConfigTester.java:38) </pre> <p>在创建输出之前，该转换符可以用求值式来测试是否满足给定的条件。例如，只有求值式“<code>CALLER_DISPLAY_EVAL</code>”返回 <code>true</code> 时，“<code>%caller{3,CALLER_DISPLAY_EVAL}</code>”才会输出三行堆栈跟踪。</p>	
L line	<p>输出执行记录请求的行号。</p> <p>输出行号并不很快，所以尽量避免使用，除非执行速度不造成任何问题。</p>	

m msg message	输出与记录事件相关联的应用程序提供的消息。						
M method	输出执行记录请求的方法名。 输出方法名并不很快，所以尽量避免使用，除非执行速度不造成任何问题。						
n	输出与平台相关的行分隔符。 该转换符与不可移植的行分隔符如“\n”或“\r\n”的性能几乎一样。所以该换行符是指定行分隔符的首选方式。						
p le level	输出记录事件的级别。						
r relative	输出从程序启动到创建记录事件的逝去时间，单位毫秒。						
t thread	输出产生记录事件的线程名。						
X{key} mdc {key}	输出与产生记录事件的线程相关联的 MDC。 如果该转换符后有放在花括号里的 key，比如%mdc {clientNumber}，则输出该 key 对应的值。 如果没有指定 key，则输出 MDC 的全部内容，格式是“key1=val1, key2=val2”。						
ex {length} exception {length} throwable {length}  ex {length, evaluator-1, ..., evaluator-n}  exception {length, evaluator-1, ..., evaluator-n}  throwable {length, evaluator-1, ..., evaluator-n}	<p>输出与记录事件相关联的堆栈跟踪，如果有的话。默认输出全部堆栈跟踪。</p> <p>“throw”转换符可跟下面选项之一：</p> <ul style="list-style-type: none"> <li>● short: 打印堆栈跟踪的第一行；</li> <li>● full: 打印全部堆栈跟踪；</li> <li>● 任意整数: 堆栈跟踪的行数。</li> </ul> <p>示例：</p> <table border="1"> <thead> <tr> <th>转换模式</th><th>结果</th></tr> </thead> <tbody> <tr> <td>%ex</td><td>mainPackage.foo.bar.TestException: Houston we have a problem at mainPackage.foo.bar.TestThrower.fire(TestThrower.java:22) at mainPackage.foo.bar.TestThrower.readyToLaunch(TestThrower.java:17) at mainPackage.ExceptionLauncher.main(ExceptionLauncher.java:38)</td></tr> <tr> <td>%ex{short}</td><td>mainPackage.foo.bar.TestException: Houston we have a problem</td></tr> </tbody> </table>	转换模式	结果	%ex	mainPackage.foo.bar.TestException: Houston we have a problem at mainPackage.foo.bar.TestThrower.fire(TestThrower.java:22) at mainPackage.foo.bar.TestThrower.readyToLaunch(TestThrower.java:17) at mainPackage.ExceptionLauncher.main(ExceptionLauncher.java:38)	%ex{short}	mainPackage.foo.bar.TestException: Houston we have a problem
转换模式	结果						
%ex	mainPackage.foo.bar.TestException: Houston we have a problem at mainPackage.foo.bar.TestThrower.fire(TestThrower.java:22) at mainPackage.foo.bar.TestThrower.readyToLaunch(TestThrower.java:17) at mainPackage.ExceptionLauncher.main(ExceptionLauncher.java:38)						
%ex{short}	mainPackage.foo.bar.TestException: Houston we have a problem						

	<pre> at mainPackage.foo.bar.TestThrower.fire(TestThrower.java:22) %ex{full} mainPackage.foo.bar.TestException: Houston we have a problem            at mainPackage.foo.bar.TestThrower.fire(TestThrower.java:22)            at mainPackage.foo.bar.TestThrower.readyToLaunch(TestThrower.java:17)            at mainPackage.ExceptionLauncher.main(ExceptionLauncher.java:38) %ex{2}    mainPackage.foo.bar.TestException: Houston we have a problem            at mainPackage.foo.bar.TestThrower.fire(TestThrower.java:22)            at mainPackage.foo.bar.TestThrower.readyToLaunch(TestThrower.java:17) </pre> <p>在创建输出之前，该转换符可以用求值式来测试是否满足给定的条件。例如，只有求值式“EX_DISPLAY_EVAL”返回 false 时，“%ex{full, EX_DISPLAY_EVAL}”才会输出全部堆栈跟踪。</p>
<pre> xEx{length} xException{length} xThrowable{length}  xEx{length, evaluator-1, ..., evaluator-n} xException{length, evaluator-1, ..., evaluator-n} xThrowable{length, evaluator-1, ..., evaluator-n} </pre>	<p>与上面的转换符“%throwable”一样，只是多了包信息。</p> <p>如果没有指定“%xThrowable”或其他与 throwable 有关的转换符，则 PatternLayout 会根据堆栈跟踪信息的重要性，自动把它作为最后面的转换符。如果不想显示堆栈跟踪信息，则可以用转换符“\$nopex”替换“%xThrowable”。</p> <p>堆栈跟踪的每帧后面，会添加包名和在包 jar 里找到的“Implementation-Version”。如果信息不确定，则包名前会添加波浪符“~”。</p> <p>示例：</p> <pre> java.lang.NullPointerException   at com.xyz.Wombat(Wombat.java:57) ~[wombat-1.3.jar:1.3]   at com.xyz.Wombat(Wombat.java:76) ~[wombat-1.3.jar:1.3]   at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:1.5.0_06]   at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl. java:39) ~[na:1.5.0_06]   at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcces sorImpl.java:25) ~[na:1.5.0_06]   at java.lang.reflect.Method.invoke(Method.java:585) ~[na:1.5.0_06]   at org.junit.internal.runners.TestMethod.invoke(TestMethod.java:59) [junit-4.4.jar:na]   at org.junit.internal.runners.MethodRoadie.runTestMethod(MethodRoadie.java </pre>



	<div>:98) [junit-4.4.jar:na] ...etc</div> <p>Logback 尽力确保包信息的正确性，即使是在非常复杂的类加载器层次里。但是，当不能保证信息正确性时，会在信息前添加“~”。因此理论上可以区分打印出的包信息与真实包信息。在上例中，由于 Wombat 类的包信息前有“~”，所以真实的包信息可能会是[wombat.jar:1.7]。</p>
nopex nopexception	<p>表示不输出任何堆栈跟踪，因此可以高效地忽略异常。</p> <p>如果没有指定“%xThrowable”或其他与 throwable 有关的转换符，则 PatternLayout 会自动把“%xThrowable”作为最后面的转换符，但本转换符可以覆盖这种默认行为，从而不显示堆栈跟踪信息。</p>
marker	<p>输出与记录请求相关联的 marker。</p> <p>如果 marker 包含子 marker，则按照下面的格式输出父、子 marker 的名称：parentName [ child1, child2 ]</p>
property{key}	<p>输出名为“key”上下文属性的值。如果“key”不是 logger 上下文的属性，则从系统属性里查找。</p> <p>“key”没有默认值，如果忽略之，则返回错误提示“Property_HAS_NO_KEY”。</p>

由于在转换模式上下文里，百分号“%”有特殊含义，所以如果想把“%”作为普通文本文字，则必须用“\”对它进行转义，如“d%p\%%m%n”。

6.2.4. 格式修饰符

默认情况下，相关信息会按原格式输出。但是，在格式修饰符的帮助下，就可以为每个字段指定最小、最大宽度，以及对齐方式。

可选的格式修饰符位于百分号与转换符之间。

第一个可选的格式修饰符是左对齐标志，符号是减号“-”。接着是可选的最小宽度修饰符，符号是表示输出的最少字符的十进制数字。如果字符数小于最小宽度，则左填充或右填充。默认是左填充（即右对齐）。填充符是空格。如果字符数大于最小宽度，则扩张到字符的宽度。字符永远不会被截断。

最大宽度修饰符能够改变上面的行为，符号是点号“.”后加数字。如果字符数大于最大宽度，则从前面截断字符。例如，如果最大宽度是“8”，字符有 10 个，则前两个字符会被抛弃。C 语言的 printf 函数是从字符尾部进行截断。

在点号“.”后加上减号“-”表示从尾部截断。例如，最大宽度是“8”，字符有 10 个，

则最后两个字符会被抛弃。

下面是格式修饰符的各种例子。

格式修饰符	左对齐	最小宽度	最大宽度	备注
%20logger	false	20	无	如果 logger 名少于 20 个字符则左填充空格。
%-20logger	true	20	无	如果 logger 名少于 20 个字符则右填充空格。
%.30logger	NA	无	30	如果 logger 名多于 30 个字符则从前截断。
%20.30logger	false	20	30	如果 logger 名少于 20 个字符则左填充空格。 同时，如果 logger 名多于 30 个字符则从前截断。
%-20.30logger	true	20	30	如果 logger 名少于 20 个字符则右填充空格 同时，如果 logger 名多于 30 个字符则从前截断。
%. -30logger	NA	无	30	如果 logger 名多于 30 个字符则从后截断。

下表是格式修饰符截断的例子。请注意方括号“[]”不是输出的一部分，只是用于界定输出的宽度。

格式修饰符	logger 名	结果
[%20.20logger]	main.Name	[ main.Name]
[%-20.20logger]	main.Name	[main.Name ]
[%10.10logger]	main.foo.foo.bar.Name	[o.bar.Name]
[%10.-10logger]	main.foo.foo.bar.Name	[main.foo.f]

#### 6.2.4.1. 用一个字符输出级别

可以不打印级别的全名，而是用 T、D、W、I 和 E，分别对应 TRACE、DEBUG、WARN、INFO 和 ERROR。你既可以写一个自定义转换器，或简单地用上面的格式修饰符把级别缩短为一个字符，如“%.1level”。

#### 6.2.5. 圆括号的特殊含义

在 logback 里，圆括号被视为编组标记。因此可以将一个子模式进行编组，然后对这个编组应用格式化指令。

例如，模式：

```
%-30(%d{HH:mm:ss.SSS} [%thread]) %-5level %logger{32} - %msg%n
```

将对子模式 “%d{HH:mm:ss.SSS} [%thread]” 产生的输出进行编组，结果是，如果少于 30 个字符就右填充。

如果没有编组，则输出：

```
13:09:30 [main] DEBUG c.q.logback.demo.ContextListener - Classload hashCode is 13995234
13:09:30 [main] DEBUG c.q.logback.demo.ContextListener - Initializing for ServletContext
13:09:30 [main] DEBUG c.q.logback.demo.ContextListener - Trying platform Mbean server
13:09:30 [pool-1-thread-1] INFO ch.qos.logback.demo.LoggingTask - Howdydy-diddly-ho - 0
13:09:38 [btpool0-7] INFO c.q.l.demo.lottery.LotteryAction - Number: 50 was tried.
13:09:40 [btpool0-7] INFO c.q.l.d.prime.NumberCruncherImpl - Beginning to factor.
13:09:40 [btpool0-7] DEBUG c.q.l.d.prime.NumberCruncherImpl - Trying 2 as a factor.
13:09:40 [btpool0-7] INFO c.q.l.d.prime.NumberCruncherImpl - Found factor 2
```

有了 “%-30()” 编组后，输出：

```
13:09:30 [main]          DEBUG c.q.logback.demo.ContextListener - Classload hashCode is
13995234
13:09:30 [main]          DEBUG c.q.logback.demo.ContextListener - Initializing for
ServletContext
13:09:30 [main]          DEBUG c.q.logback.demo.ContextListener - Trying platform
Mbean server
13:09:30 [pool-1-thread-1] INFO ch.qos.logback.demo.LoggingTask - Howdydy-diddly-ho - 0
13:09:38 [btpool0-7]     INFO c.q.l.demo.lottery.LotteryAction - Number: 50 was tried.
13:09:40 [btpool0-7]     INFO c.q.l.d.prime.NumberCruncherImpl - Beginning to factor.
13:09:40 [btpool0-7]     DEBUG c.q.l.d.prime.NumberCruncherImpl - Trying 2 as a factor.
13:09:40 [btpool0-7]     INFO c.q.l.d.prime.NumberCruncherImpl - Found factor 2
```

后者更适宜阅读，尤其当句子很长时。

如果要把圆括号作为普通文本文字，则用前置“\”进行转义，比如“\(%d{HH:mm:ss.SSS} [%thread])\”。严格地说，只有闭合圆括号（“)”）才需要转义。所以“(%d [%thread])\”等价于“\(%d [%thread])\”。但是因为很难记清楚哪个圆括号需要转义，所以你可以对两个圆括号都进行转义。

### 6.2.5.1. 选项

格式修饰符后可以跟选项。选项总是在花括号里声明。我们已经见过选项的一些用法，比如与 MDC 转换符联合使用的：`%mdc{someKey}`。

格式转换符可以有多个选项。例如，我们很快会讲到一个使用求值式的格式转换符，可以在选项列表里添加求值式：

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
  <layout class="ch.qos.logback.classic.PatternLayout">
    <param name="Pattern"
      value="%-4relative [%thread] %-5level - %msg%n \
        %caller{2, DISP_CALLER_EVAL, OTHER_EVAL_NAME, THIRD_EVAL_NAME}"
    />
  </layout>
</appender>
```

### 6.2.6. 求值式 (Evaluator)

上面讲过，当格式转换符需要根据一个或多个 `EventEvaluator` 对象 (`ch.qos.logback.core.boolex.EventEvaluator`) 而有动态的行为时，选项列表就派得上用场了。`EventEvaluator` 对象负责决定给定的记录事件是否匹配求值式的条件。

我们来看一个调用 `EventEvaluator` 的例子。下面的配置文件向控制台输出记录事件，显示日期、线程、级别、消息和调用者数据。鉴于提取记录事件的调用者数据很昂贵，我们仅当记录事件源自指定 `logger`、且消息包含指定字符串时才这样做，因此我们确定只有特定的记录请求才能产生和输出它们的调用者信息。其他情况下，我们不输出调用者信息。

求值器和求值表达式详见第 7 章过滤器。下面的例子隐式地基于 `JaninoEventEvaluator`，依赖 `Janino` 类库 (<http://docs.codehaus.org/display/JANINO/Home>)。

示例：EventEvaluators 用法

(`logback-examples/src/main/java/chapters/layouts/callerEvaluatorConfig.xml`)

```
<configuration>

  <evaluator name="DISPLAY_CALLER_EVAL">
    <expression>
      logger.contains("chapters.layouts") & &
      message.contains("who calls thee")
    </expression>
  </evaluator>
```

```
<appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
    <pattern>%-4relative [%thread] %-5level - %msg%n%caller{2,
      DISPLAY_CALLER_EVAL}</pattern>
  </encoder>
</appender>

<root level="DEBUG">
  <appender-ref ref="STDOUT" />
</root>
</configuration>
```

上面的求值表达式匹配的事件必须满足两个条件：产生事件的 `logger` 的名字包含字符串 “chapters.layouts”，消息包含字符串 “who calls thee”。按照 XML 语法，“&” 被转义为 “&amp;”。

下面的类使用了上面的配置文件里的某些特性。

示例：EventEvaluators 用法

(logback-examples/src/main/java/chapters/layouts/CallerEvaluatorExample.java)

```
package chapters.layouts;

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

import ch.qos.logback.classic.LoggerContext;
import ch.qos.logback.classic.joran.JoranConfigurator;
import ch.qos.logback.core.joran.spi.JoranException;
import ch.qos.logback.core.util.StatusPrinter;

public class CallerEvaluatorExample {

  public static void main(String[] args) {
    Logger logger =
      LoggerFactory.getLogger(CallerEvaluatorExample.class);
    LoggerContext lc = (LoggerContext)
      LoggerFactory.getILoggerFactory();

    try {
      JoranConfigurator configurator = new JoranConfigurator();
      configurator.setContext(lc);
      lc.reset();
    }
  }
}
```

```
        configurator.doConfigure(args[0]);
    } catch (JoranException je) {
        // StatusPrinter will handle this
    }
    StatusPrinter.printInCaseOfErrorsOrWarnings(lc);

    for (int i = 0; i < 5; i++) {
        if (i == 3) {
            logger.debug("who calls thee?");
        } else {
            logger.debug("I know me " + i);
        }
    }
}
```

该程序执行 5 条记录请求，第三条消息包含 “who calls thee?”。

执行：

```
java                                     chapters.layouts.CallerEvaluatorExample
src/main/java/chapters/layouts/callerEvaluatorConfig.xml
```

输出：

```
0    [main] DEBUG - I know me 0
0    [main] DEBUG - I know me 1
0    [main] DEBUG - I know me 2
0    [main] DEBUG - who calls thee?
Caller+0    at chapters.layouts.CallerEvaluatorExample.main(CallerEvaluatorExample.java:28)
0    [main] DEBUG - I know me 4
```

当记录请求被执行时，对应的记录事件就被评估。只有第三个记录事件匹配了评估条件，所以显示其调用者数据。其他的记录事件不能匹配评估条件，所以不打印调用者数据。

**重要：**使用 “caller” 转换符时，只有当求值表达式为 `true` 时，才会输出调用者信息。

让我们换个角度考虑。当记录请求里有异常时，会输出异常的堆栈跟踪。但是，你也许想不输出堆栈跟踪。

下面的 `java` 代码创建了三条记录请求，每个都有异常。第二个异常与其他两个不同之

处在于：包含字符串 “do not display this”、类型是 `chapters.layouts.TestException`。让我们屏蔽第二个异常。

示例：EventEvaluators 用法

(logback-examples/src/main/java/chapters/layouts/ExceptionEvaluatorExample.java)

```
package chapters.layouts;

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

import ch.qos.logback.classic.LoggerContext;
import ch.qos.logback.classic.joran.JoranConfigurator;
import ch.qos.logback.core.joran.spi.JoranException;
import ch.qos.logback.core.util.StatusPrinter;

public class ExceptionEvaluatorExample {

    public static void main(String[] args) {
        Logger logger = LoggerFactory
            .getLogger(ExceptionEvaluatorExample.class);
        LoggerContext lc = (LoggerContext)
        LoggerFactory.getILoggerFactory();

        try {
            JoranConfigurator configurator = new JoranConfigurator();
            configurator.setContext(lc);
            lc.reset();
            configurator.doConfigure(args[0]);
        } catch (JoranException je) {
            // StatusPrinter will handle this
        }
        StatusPrinter.printInCaseOfErrorsOrWarnings(lc);

        for (int i = 0; i < 3; i++) {
            if (i == 1) {
                logger.debug("logging statement " + i, new TestException(
                    "do not display this"));
            } else {
                logger
                    .debug("logging statement " + i, new Exception(
                        "display"));
            }
        }
    }
}
```

```
}  
}
```

下面的配置里, 求值表达式匹配那些包含类型 `chapters.layouts.TestException` 为 throwable 的事件。

示例: `EventEvaluators` 用法

(`logback-examples/src/main/java/chapters/layouts/exceptionEvaluatorConfig.xml`)

```
<configuration>  
  
  <evaluator name="DISPLAY_EX_EVAL">  
    <expression>throwable != null && throwable instanceof  
      chapters.layouts.TestException</expression>  
  </evaluator>  
  
  <appender name="STDOUT"  
class="ch.qos.logback.core.ConsoleAppender">  
    <encoder>  
      <pattern>%msg%n%xEx{full, DISPLAY_EX_EVAL}</pattern>  
    </encoder>  
  </appender>  
  
  <root level="DEBUG">  
    <appender-ref ref="STDOUT" />  
  </root>  
</configuration>
```

用了上面的配置后, 每个包含 “`chapters.layouts.TestException`” 实例的记录请求都不会打印堆栈跟踪。

运行:

```
java                                     chapters.layouts.ExceptionEvaluatorExample  
src/main/java/chapters/layouts/exceptionEvaluatorConfig.xml
```

输出:

```
logging statement 0  
java.lang.Exception: display  
    at chapters.layouts.ExceptionEvaluatorExample.main(ExceptionEvaluatorExample.java:43)  
[logback-examples-0.9.19.jar:na]  
logging statement 1  
logging statement 2  
java.lang.Exception: display
```



```
at chapters.layouts.ExceptionEvaluatorExample.main(ExceptionEvaluatorExample.java:43)
[logback-examples-0.9.19.jar:na]
```

注意第二条记录语句是如何没有堆栈跟踪的。我们有效地阻止输出 `TextException` 的堆栈跟踪。堆栈跟踪的每帧后面的方括号里的文字是前面讲过的包信息。

重要：使用 “%ex” 或 “%xEx” 转换符时，只有当求值表达式为 `false` 时，才会输出堆栈跟踪。

## 6.2.7. 创建自定义格式转换符

创建自定义格式转换符有两步。

首先，必须继承 `ClassicConverter` 类。`ClassicConverter` 对象负责从 `ILoggingEvent` 提取信息，并产生一个字符串。例如，`LoggerConverter`，它是处理 “%logger” 转换符的转换器，它从 `ILoggingEvent` 提取 `logger` 的名字并作为字符串返回。

假设我们的自定义 `ClassicConverter` 的功能是按照 ANSI 终端惯例为记录事件的级别进行着色，下面是一种可能的实现：

示例：样本转换器例子

(src/main/java/chapters/layouts/MySampleConverter.java)

```
package chapters.layouts;

import ch.qos.logback.classic.Level;
import ch.qos.logback.classic.pattern.ClassicConverter;
import ch.qos.logback.classic.spi.ILoggingEvent;

public class MySampleConverter extends ClassicConverter {

    private static final String END_COLOR = "\u001b[m";

    private static final String ERROR_COLOR = "\u001b[0;31m";
    private static final String WARN_COLOR = "\u001b[0;33m";

    @Override
    public String convert(ILoggingEvent event) {
        StringBuffer sbuf = new StringBuffer();
        sbuf.append(getColor(event.getLevel()));
        sbuf.append(event.getLevel());
        sbuf.append(END_COLOR);
        return sbuf.toString();
    }
}
```

```
/**
 * Returns the appropriate characters to change the color for the
 * specified
 * logging level.
 */
private String getColor(Level level) {
    switch (level.toInt()) {
        case Level.ERROR_INT:
            return ERROR_COLOR;
        case Level.WARN_INT:
            return WARN_COLOR;
        default:
            return "";
    }
}
```

这里的实现很直观。MySampleConverter 类继承 ClassicConverter，实现 convert 方法，convert 方法返回按照 ANSI 着色编码装饰后的字符串。

第二步，我们必须让 logback 知道这个新的 Converter。方法是在配置里声明新的转换符。

示例：样本转换器例子

(src/main/java/chapters/layouts/mySampleConverterConfig.xml)

```
<configuration>

    <conversionRule conversionWord="sample"
        converterClass="chapters.layouts.MySampleConverter" />

    <appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%-4relative [%thread] %sample - %msg%n</pattern>
        </encoder>
    </appender>

    <root level="DEBUG">
        <appender-ref ref="STDOUT" />
    </root>
```

```
</configuration>
```

新的转换符号在配置文件里被声明后，我们可以在 `PatternLayout` 模式里像引用任何其他转换符一样引用它。

由于 Windows 不支持 ANSI 终端编码，你可以在其他平台如 Linux 或 Mac 上执行看到效果。

执行：

```
java chapters.layouts.SampleLogging
src/main/java/chapters/layouts/mySampleConverterConfig.xml
```

输出：

```
0 [main] DEBUG - Everything's going well
3 [main] ERROR - maybe not quite...
```

请注意“ERROR”是红色的，也正是本例的目的。

## 6.2.8. HTMLLayout

HTMLLayout 以 HTML 表格的形式输出记录，表格的每行对应于一个记录事件。

下面是使用默认 CSS 的 HTMLLayout 的输出样本：

RelativeTime	Thread	MDC	Level	Logger	Message
0	main		INFO	main.apollo.13.Launcher	Preparing for takeoff
0	main		DEBUG	main.apollo.13.Launcher	Engines ready
0	main		DEBUG	main.apollo.13.Launcher	T minus 10 and counting
0	main		DEBUG	main.apollo.13.Launcher	10..9..8..7..6..5..4..3..2
0	main		DEBUG	main.apollo.13.Launcher	Ignition
0	main		ERROR	main.apollo.13.Shuttle	Houston, we have a problem
apollo.shuttle.ShuttleException: oops...					
apollo.shuttle.Shuttle.oops(Shuttle.java:12)					
main.ShuttleManager.main(ShuttleManager.java:33)					

表格的列是由格式转换符指定的，因此你可以完全控制表格的内容和格式。你可以选择和显示任何被 `PatternLayout` 所知的转换器的组合。

`PatternLayout` 与 `HTMLLayout` 一起使用的一个例外是，格式转换符不能用空格分隔，或更一般地说，不能被文本文字分隔。格式转换符里的每个转换符都会产生一个单独的列。同样地，转换符里的每块文本文字也会导致生成一个单独的列。

下面是演示了 `HTMLLayout` 的简单用法。

示例：HTMLLayout 例子

HTMLLayout Example (src/main/java/chapters/layouts/htmlLayoutConfig1.xml)

```
<configuration>
  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <encoder
class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
      <layout class="ch.qos.logback.classic.html.HTMLLayout">

        <pattern>%relative%thread%mdc%level%logger%msg</pattern>
      </layout>
    </encoder>
    <file>test.html</file>
  </appender>

  <root level="DEBUG">
    <appender-ref ref="FILE" />
  </root>
</configuration>
```

程序 TrivialMain（chapters.layouts.TrivialMain）记录了一些消息，最后记录一个异常。  
运行：

```
java chapters.layouts.TrivialMain src/main/java/chapters/layouts/htmlLayoutConfig1.xml
```

会在当前目录创建文件“test.html”，文件内容与下面类似：

RelativeTime	Thread	MDC	Level	Logger	Message
141	main		INFO	chapters.layouts.TrivialMain	an info message 0
141	main		DEBUG	chapters.layouts.TrivialMain	hello world number1
141	main		DEBUG	chapters.layouts.TrivialMain	hello world number2
141	main		DEBUG	chapters.layouts.TrivialMain	hello world number3
141	main		DEBUG	chapters.layouts.TrivialMain	hello world number4
141	main		INFO	chapters.layouts.TrivialMain	an info message 5
141	main		ERROR	chapters.layouts.TrivialMain	Finish off with fireworks
java.lang.Exception: Just testing at chapters.layouts.TrivialMain.main(TrivialMain.java:46)					

6.2.8.1. 堆栈跟踪

如果使用“%ex”来显示堆栈跟踪，则会创建一个表格列来显示堆栈跟踪。多数时候该列式空的，白白浪费屏幕空间。而且，在单独的列里打印堆栈跟踪不易读。幸运的是，“%ex”不是显示堆栈跟踪的唯一方法。

IThrowableRenderer 接口的具体实现提供了更好的解决方案。这样的实现可以被委派给 HTMLLayout，负责管理与异常有关的显示数据。默认情况下，DefaultThrowableRenderer 被委派给每个 HTMLLayout，它把异常及其堆栈跟踪写到新行，这就方便看了，如上图所示。

如果你坚持想用 “%ex” 模式，那么可以在配置文件里设置 `NOPTrowableRenderer`，就能禁止在单独的行里显示堆栈跟踪。

### 6.2.8.2. CSS

`HTMLLayout` 通过 CSS (Cascading Style Sheet) 展示生成的 HTML。`HTMLLayout` 有默认的内部 CSS。但你可以让 `HTMLLayout` 使用外部 CSS 文件，方法是用嵌套在 `<layout>` 元素里的 `cssBuilder` 元素，如下所示：

```
<layout class="ch.qos.logback.classic.html.HTMLLayout">
  <pattern>%relative...%msg</pattern>
  <cssBuilder class="ch.qos.logback.core.html.UrlCssBuilder">
    <!-- url where the css file is located -->
    <url>http://...</url>
  </cssBuilder>
</layout>
```

`HTMLLayout` 经常和 `SMTPAppender` 一起使用，让发出的 email 是好看的 HTML。

## 6.3. Logback access

`Logback-access` 的大多数 layout 都改变自 `logback-classic`。`Logback-access` 和 `logback-classic` 有各自的用途，但一般提供类似的功能。

### 6.3.1. 自定义 layout

为 `logback-access` 写自定义 layout 与在 `logback-classic` 里基本相同。

### 6.3.2. PatternLayout

`PatternLayout` 的配置方式与在 `logback-classic` 里的 `PatternLayout` 基本相同。但它提供了一些额外的格式转换符，适合记录那些只存在于 HTTP servlet 请求和 HTTP servlet 响应的信息。

下面是 `logback-access` 里的 `PatternLayout` 的额外格式转换符。当单元格里同时列出多个转换符时，它们就是同义词。

转换符	作用
a remoteIP	远程 IP 地址。
A localIP	本地 IP 地址。
b B byteSent	响应的 content length。
h clientHost	远程主机。
H protocol	请求协议。
I	远程 logger 名。在 logback-access 里，总返回 “-”。
reqParameter {paramName}	请求的参数。 花括号里是参数名。 %reqParameter {input_data} 显示对应的参数值。
i{header} header {header}	请求头。 花括号里是参数名。 %header {Referer} 显示请求的 “referer”。 如果未指定选项，则显示请求头里的全部信息。
m requestMethod	请求方法
r requestURL	请求的 URL
s statusCode	请求的状态码
t date	输出记录事件的日期。 该转换符可后跟一对花括号，里面是与 java.text.SimpleDateFormat 的格式兼容的日期时间模式。 ABSOLUTE、DATE 和 ISO8601 都有效。 例如, %d{HH:mm:ss,SSS}, %d{dd MMM yyyy ;HH:mm:ss,SSS} 或 %d{DATE}。如果没有指定日期格式，则默认为 ISO8601 格式。
u	远程用户。

user	
U requestURI	请求的 URI。
v server	服务器名。
localPort	本地端口。
reqAttribute{attributeName}	请求的属性。 花括号里是属性名。 %reqAttribute{SOME_ATTRIBUTE} 显示对应的属性值。
reqCookie{cookie}	请求的 cookie。 花括号里是 cookie 名。 %cookie{COOKIE_NAME} 显示对应的 cookie 值。
responseHeader{header}	响应的头。 花括号里是参数名。 %header{Referer} 显示响应的 “referer”。
requestContent	显示请求的内容，即请求的 <code>InputStream</code> 。该转换符与 <code>Servlet</code> 过滤器 <code>TeeFilter</code> ( <code>ch.qos.logback.access.servlet.TeeFilter</code> ) 联合使用，用 <code>TeeHttpServletRequest</code> ( <code>ch.qos.logback.access.servlet.TeeHttpServletRequest</code> ) 替换原始 <code>HttpServletRequest</code> ，允许多次访问请求的 <code>InputStream</code> 而不会丢失任何数据。
fullRequest	输出与请求相关的数据，包括所有头和请求内容。
responseContent	显示响应的内容，即响应的 <code>InputStream</code> 。该转换符与 <code>Servlet</code> 过滤器 <code>TeeFilter</code> ( <code>ch.qos.logback.access.servlet.TeeFilter</code> ) 联合使用，用 <code>TeeHttpServletResponse</code> ( <code>ch.qos.logback.access.servlet.TeeHttpServletResponse</code> ) 替换原始 <code>HttpServletResponse</code> ，允许多次访问请求的 <code>InputStream</code> 而不会丢失任何数据。
fullResponse	输出与响应相关的数据，包括所有头和响应内容。

Logback-access 的 `PatternLayout` 还有三个关键字，作为某种模式的快捷方式。

关键字	等价的转换模式
common 或 CLF	%h %l %u %t \"%r\" %s %b
combined	%h %l %u %t \"%r\" %s %b \"%i{Referer}\" \"%i{User-Agent}\"

关键字 “common” 相当于模式 “%h %l %u %t \"%r\" %s %b”，显示客户端主机、远程

logger 名、用户、日期、请求的 URL、状态码和响应的 content length。

关键字 “combined” 相当于模式 “%h %l %u %t \"%r\" %s %b \"%i{Referer}\" \"%i{User-Agent}\"”，开头和 “common” 模式类似，但多显示了两个请求头：“referer” 和 “user-agent”。

### 6.3.3. HTMLLayout

HTMLLayout 与 logback-classic 里的 HTMLLayout 相似。

默认情况下，PatternLayout 创建包含下列数据的表格：

- 远程 IP；
- 日期；
- 请求的 URL；
- 状态码；
- Content Length

下面是 logback-access 里的 HTMLLayout 输出样本：

RemoteHost	RemoteUser	Date	RequestURL	StatusCode	ContentLength
127.0.0.1	-	25/10/2006:18:40:14 +0200	POST /test/session/ HTTP/1.1	302	0
127.0.0.1	-	25/10/2006:18:40:14 +0200	GET /test/session/?R=0 HTTP/1.1	200	157
127.0.0.1	-	25/10/2006:18:40:14 +0200	POST /test/session/ HTTP/1.1	302	0
127.0.0.1	-	25/10/2006:18:40:14 +0200	GET /test/session/?R=1 HTTP/1.1	200	732
127.0.0.1	-	25/10/2006:18:40:19 +0200	POST /test/session/ HTTP/1.1	302	0
127.0.0.1	-	25/10/2006:18:40:19 +0200	GET /test/session/?R=2 HTTP/1.1	200	754
127.0.0.1	-	25/10/2006:18:40:20 +0200	POST /test/session/ HTTP/1.1	302	0
127.0.0.1	-	25/10/2006:18:40:20 +0200	GET /test/session/?R=3 HTTP/1.1	200	157
127.0.0.1	-	25/10/2006:18:40:40 +0200	GET /test/chat/chat.html HTTP/1.1	200	4566
127.0.0.1	-	25/10/2006:18:40:40 +0200	GET /test/js/default.js HTTP/1.1	200	670
127.0.0.1	-	25/10/2006:18:40:40 +0200	GET /test/js/prototype.js HTTP/1.1	200	47603
127.0.0.1	-	25/10/2006:18:40:40 +0200	GET /test/js/behaviour.js HTTP/1.1	200	7896
127.0.0.1	-	25/10/2006:18:40:40 +0200	GET /test/js/ajax.js HTTP/1.1	200	4705
127.0.0.1	-	25/10/2006:18:40:40 +0200	GET /test/chat/chat.js HTTP/1.1	200	3894
127.0.0.1	-	25/10/2006:18:40:40 +0200	GET /test/chat/chat.css HTTP/1.1	200	688
127.0.0.1	-	25/10/2006:18:40:40 +0200	GET /test/chat/?ajax=poll&message=poll&timeout=0&_ = HTTP/1.1	200	35
127.0.0.1	-	25/10/2006:18:40:41 +0200	GET /test/chat/?ajax=poll&message=poll&_ = HTTP/1.1	200	219
127.0.0.1	-	25/10/2006:18:40:41 +0200	POST /test/chat/ HTTP/1.1	200	113
127.0.0.1	-	25/10/2006:18:40:43 +0200	POST /test/chat/ HTTP/1.1	200	97
127.0.0.1	-	25/10/2006:18:40:43 +0200	GET /test/chat/?ajax=poll&message=poll&_ = HTTP/1.1	200	35
127.0.0.1	-	25/10/2006:18:40:53 +0200	GET /test/chat/?ajax=poll&message=poll&_ = HTTP/1.1	200	35
127.0.0.1	-	25/10/2006:18:41:03 +0200	GET /test/chat/?ajax=poll&message=poll&_ = HTTP/1.1	200	35
127.0.0.1	-	25/10/2006:18:41:13 +0200	GET /test/chat/?ajax=poll&message=poll&_ = HTTP/1.1	200	35
127.0.0.1	-	25/10/2006:18:41:23 +0200	GET /test/chat/?ajax=poll&message=poll&_ = HTTP/1.1	200	35
127.0.0.1	-	25/10/2006:18:41:33 +0200	GET /test/chat/?ajax=poll&message=poll&_ = HTTP/1.1	200	35
127.0.0.1	-	25/10/2006:18:41:35 +0200	GET /test/data.txt HTTP/1.1	200	25691
127.0.0.1	-	25/10/2006:18:41:43 +0200	GET /test/chat/?ajax=poll&message=poll&_ = HTTP/1.1	200	35
127.0.0.1	-	25/10/2006:18:41:48 +0200	GET /javadoc/ HTTP/1.1	302	0
127.0.0.1	-	25/10/2006:18:41:48 +0200	GET /javadoc/index.html HTTP/1.1	200	1411
127.0.0.1	-	25/10/2006:18:41:48 +0200	GET /javadoc/overview-frame.html HTTP/1.1	200	7764
127.0.0.1	-	25/10/2006:18:41:48 +0200	GET /javadoc/overview-summary.html HTTP/1.1	200	14465
127.0.0.1	-	25/10/2006:18:41:48 +0200	GET /javadoc/allclasses-frame.html HTTP/1.1	200	55049
127.0.0.1	-	25/10/2006:18:41:48 +0200	GET /javadoc/styleSheet.css HTTP/1.1	200	1202
127.0.0.1	-	25/10/2006:18:41:48 +0200	GET /javadoc/styleSheet.css HTTP/1.1	200	1202
127.0.0.1	-	25/10/2006:18:41:48 +0200	GET /javadoc/styleSheet.css HTTP/1.1	200	1202



## 7. 过滤器 (Filter)

Logback 的过滤器基于三值逻辑 (ternary logic)，允许把它们组装或成链，从而组成任意的复合过滤策略。过滤器很大程度上受到 Linux 的 iptables 启发。

译者注：这里的所谓三值逻辑是说，过滤器的返回值只能是 ACCEPT、DENY 和 NEUTRAL 的其中一个。

### 7.1. 在 logback-classic 里

Logback-classic 提供两种类型的过滤器：常规过滤器和 TurboFilter 过滤器。

#### 7.1.1. 常规过滤器

Logback-classic 的常规过滤器继承 Filter 抽象类，该类的核心方法 decide() 的参数是一个 ILoggingEvent 实例。

过滤器按照有序列表进行组织，基于三值逻辑。各个过滤器的 decide(ILoggingEvent event) 方法按照顺序被调用，该方法返回 FilterReply 的一个枚举值，即 DENY、NEUTRAL 和 ACCEPT 其中之一。如果返回 DENY，那么记录事件立即被抛弃，不再经过剩余过滤器。如果返回 NEUTRAL，那么有序列表里的下一个过滤器会接着处理记录事件。如果返回 ACCEPT，那么记录事件被立即处理，不再经过剩余过滤器。

在 logback-classic 里，过滤器可以被添加到 Appender 实例。为 Appender 添加一个或多个过滤器后，你可以用任意条件对事件行过滤，比如记录消息的内容、MDC 的内容、日期里的时间或者记录事件的任何其他部分。

##### 7.1.1.1. 自定义过滤器

创建自己的过滤器很容易。只需要继承 Filter 抽象类并实现 decide() 方法。

如下面的 SampleFilter 类，当记录事件的消息字段包含字符串 “sample” 时，它的 decide 方法返回 ACCEPT，其他情况下返回 NEUTRAL。

示例：基本自定义过滤器

(logback-examples/src/main/java/chapters/filters/SampleFilter.java)

```
package chapters.filters;

import ch.qos.logback.classic.spi.ILoggingEvent;
import ch.qos.logback.core.filter.Filter;
import ch.qos.logback.core.spi.FilterReply;
```

```
public class SampleFilter extends Filter<ILoggingEvent> {

    @Override
    public FilterReply decide(ILoggingEvent event) {
        if (event.getMessage() != null &&
            event.getMessage().contains("sample")) {
            return FilterReply.ACCEPT;
        } else {
            return FilterReply.NEUTRAL;
        }
    }
}
```

上例使用的配置文件如下：

示例：SampleFilter 配置

(logback-examples/src/main/java/chapters/filters/SampleFilterConfig.xml)

```
<configuration>

    <appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
        <filter class="chapters.filters.SampleFilter" />

        <encoder>
            <pattern>
                %-4relative [%thread] %-5level %logger - %msg%n
            </pattern>
        </encoder>
    </appender>

    <root>
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

借助于 Joran——logback 的配置框架，很容易为过滤器指定属性或子组件：为过滤器类添加相应的 setter 方法后，在<filter>元素下添加“property”元素，然后在 property 里设置属性的值。

过滤器逻辑经常由两个正交部分组成：一次匹配/不匹配测试和对匹配/不匹配的响应。例如，假设在一次测试中，消息是“foobar”，一个过滤器可能在匹配时响应“ACCEPT”，在不匹配时响应“NEUTRAL”，另一个过滤器可能在匹配时响应“NEUTRAL”，在不匹配

时响应 “DENY”。

考虑到这种正交性, logback 带了 `AbstractMatcherFilter` 类, 该类提供了有用的基础框架, 它有两个属性 `OnMatch` 和 `OnMismatch`, 分别指定在匹配和不匹配时的响应。Logback 里的多数常规过滤器都派生自 `AbstractMatcherFilter`。

#### 7.1.1.2. 级别过滤器 (LevelFilter)

`LevelFilter` 根据记录级别对记录事件进行过滤。如果事件的级别等于配置的级别, 过滤器会根据 `onMatch` 和 `onMismatch` 属性接受或拒绝事件。下面是个配置文件例子。

示例: `LevelFilter` 示例配置

(logback-examples/src/main/java/chapters/filters/levelFilterConfig.xml)

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

<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>
```

#### 7.1.1.3. 临界值过滤器 (ThresholdFilter)

`ThresholdFilter` 过滤掉低于指定临界值的事件。当记录的级别等于或高于临界值时, `ThresholdFilter` 的 `decide()` 方法会返回 `NEUTRAL`; 当记录级别低于临界值时, 事件会被拒绝。下面是个配置文件例子。

示例: `ThresholdFilter` 示例配置

(logback-examples/src/main/java/chapters/filters/thresholdFilterConfig.xml)

```
<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>
```

7.1.1.4. 求值过滤器 (EvaluatorFilter)

EvaluatorFilter 封装了 EventEvaluator (ch.qos.logback.core.boolex.EventEvaluator)，评估是否符合指定的条件。

求值过滤器 EvaluatorFilter 是抽象类，你可以通过实现 EventEvaluator 接口定义自己的求值逻辑。不过，logback-classic 提供了 EventEvaluator 的一个具体实现类 JaninoEventEvaluator，它以任意 java 布尔值表达式作为求值条件。我们把这种 java 布尔值表达式称为“求值表达式”。求值表达式让事件过滤达到了前所未有的灵活。JaninoEventEvaluator 依赖 Janino library (<http://docs.codehaus.org/display/JANINO/Home>)。

求值表达式在配置文件的解释过程中被动态编译。作为用户，你只需要保证 java 表达式返回的是布尔值。

由于求值表达式作用于当前记录事件，logback 会自动向求值表达式暴露记录事件的各种字段。下面列出了记录事件的字段名，大小写敏感：

名称	类型	描述
----	----	----

event	LoggingEvent	与记录请求相关联的原始记录事件。下面所有的变量都来自 event。例如，event.getMessage()与返回下面的“message”相同的字符串。
message	String	记录请求的原始消息。设有 logger l, “name”的值时“Alice”，则对于 <code>l.info("Hello {}", name);</code> “Hello {}”就是消息。
formattedMessage	String	记录请求里的被格式化了的消息。设有 logger l, “name”的值时“Alice”，则对于 <code>l.info("Hello {}", name);</code> “Hello Alice”就是被格式化了的消息。
logger	String	logger 名。
loggerContext	LoggerContextVO	记录事件所属于的 logger 上下文的受限（值对象）视图。
level	int	级别对应的整数值。支持默认的 DEBUG、INFO、WARN 和 ERROR。所以“level > INFO”是正确的表达式。
timeStamp	long	创建记录事件的时间戳。
marker	Marker	与记录请求相关联的 Marker 对象。注意“marker”可能为 null，所以你负责确保它不能为 null。
mdc	Map	包含创建记录事件期间的 MDC 的所有值的 map。访问值的方法是： <code>mdc.get("myKey")</code> 。 <code>mdc.get()</code> 返回 Object 而不是 String，要想对返回值调用 String 的方法，必须强制转换为 String，例如： <code>((String) mdc.get("k")).contains("val")</code> 。
throwable	java.lang.Throwable	如果没有异常与事件关联，则变量“throwable”为 null。不幸的是，“throwable”不能被序列化。所以在远程系统上，其值永远为 null。对于与位置无关的表达式，请用下面的变量“throwableProxy”。
throwableProxy	IThrowableProxy	与记录事件关联的异常的代理。如果没有异常与事件关联，则变量“throwableProxy”为 null。与“throwable”相比，当异常被关联到事件时，“throwableProxy”的值在远程系统上不会为 null，即使是在被序列化之后。

示例：事件求值式（event evaluator）基本用法

(logback-examples/src/main/java/chapters/filters/basicEventEvaluator.xml)

```
<configuration debug="true">

  <appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
    <filter class="ch.qos.logback.core.filter.EvaluatorFilter">
      <evaluator>
        <expression>message.contains("billing") </expression>
      </evaluator>
      <OnMismatch>NEUTRAL</OnMismatch>
      <OnMatch>DENY</OnMatch>
    </filter>
    <layout>
      <pattern>%-4relative [%thread] %-5level %logger
- %msg%n</pattern>
    </layout>
  </appender>

  <root level="DEBUG">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

粗体部分向 ConsoleAppender 添加了 EvaluatorFilter。EvaluatorFilter 被注射了一个类型为 JaninoEventEvaluator 的求值式。如果<evaluator>元素没有 class 属性，那么 joran 会使用默认类型，即 JaninoEventEvaluator。

<expression>元素对应于上面讨论的求值表达式。表达式 “message.contains("billing")” 返回一个布尔值。注意变量 “message” 被 JaninoEventEvaluator 自动导出。

因为 OnMatch 属性设为 NEUTRAL，OnMismatch 属性设为 DENY，求值过滤器将抛弃消息里包含字符串 “billing” 的所有记录事件。

程序 “FilterEvents” 执行了十次记录请求，从 0 到 9 编号，源代码如下：

```
package chapters.filters;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.slf4j.MDC;
import org.slf4j.Marker;
import org.slf4j.MarkerFactory;

import ch.qos.logback.classic.LoggerContext;
import ch.qos.logback.classic.joran.JoranConfigurator;
```

```
import ch.qos.logback.core.joran.spi.JoranException;

public class FilterEvents {

    public static void main(String[] args) throws InterruptedException
    {
        if (args.length == 0) {
            System.out
                .println("A configuration file must be passed as a
parameter.");
            return;
        }

        Logger logger = (Logger)
LoggerFactory.getLogger(FilterEvents.class);
        LoggerContext lc = (LoggerContext)
LoggerFactory.getILoggerFactory();

        try {
            JoranConfigurator configurator = new JoranConfigurator();
            configurator.setContext(lc);
            lc.reset();
            configurator.doConfigure(args[0]);
        } catch (JoranException je) {
            je.printStackTrace();
        }

        for (int i = 0; i < 10; i++) {
            if (i == 3) {
                MDC.put("username", "sebastien");
                logger.debug("logging statement {}", i);
                MDC.remove("username");
            } else if (i == 6) {
                Marker billing = MarkerFactory.getMarker("billing");
                logger.error(billing, "billing statement {}", i);
            } else {
                logger.info("logging statement {}", i);
            }
        }
    }
}
```

在没有任何过滤器时，运行该程序：

```
java chapters.filters.FilterEvents src/main/java/chapters/filters/basicConfiguration.xml
```

所有请求都被显示：

```
0 [main] INFO chapters.filters.FilterEvents - logging statement 0
0 [main] INFO chapters.filters.FilterEvents - logging statement 1
0 [main] INFO chapters.filters.FilterEvents - logging statement 2
0 [main] DEBUG chapters.filters.FilterEvents - logging statement 3
0 [main] INFO chapters.filters.FilterEvents - logging statement 4
0 [main] INFO chapters.filters.FilterEvents - logging statement 5
0 [main] ERROR chapters.filters.FilterEvents - billing statement 6
0 [main] INFO chapters.filters.FilterEvents - logging statement 7
0 [main] INFO chapters.filters.FilterEvents - logging statement 8
0 [main] INFO chapters.filters.FilterEvents - logging statement 9
```

假如我们想去掉“billing statement”，可以用上面的配置文件“basicEventEvaluator.xml”，运行：

```
java chapters.filters.FilterEvents src/main/java/chapters/filters/basicEventEvaluator.xml
```

输出：

```
0 [main] INFO chapters.filters.FilterEvents - logging statement 0
0 [main] INFO chapters.filters.FilterEvents - logging statement 1
0 [main] INFO chapters.filters.FilterEvents - logging statement 2
0 [main] DEBUG chapters.filters.FilterEvents - logging statement 3
0 [main] INFO chapters.filters.FilterEvents - logging statement 4
0 [main] INFO chapters.filters.FilterEvents - logging statement 5
0 [main] INFO chapters.filters.FilterEvents - logging statement 7
0 [main] INFO chapters.filters.FilterEvents - logging statement 8
0 [main] INFO chapters.filters.FilterEvents - logging statement 9
```

#### 7.1.1.4.1. 匹配器 (Matchers)

尽管能通过调用 `String` 类的 `matches()` 方法进行模式匹配，但这会导致每次调用过滤器时都会创建一个全新的 `Pattern` 对象。为消除这种开销，你可以预先定义一个或多个 `Matcher` 对象。一旦定义 `matcher` 后，就可以在求值表达式里重复引用它。

示例：在事件求值式里定义匹配器

(logback-examples/src/main/java/chapters/filters/evaluatorWithMatcher.xml)



```
<configuration debug="true">

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

        <matcher>
          <Name>odd</Name>
          <!-- 过滤掉序号为奇数的语句-->
          <regex>statement [13579]</regex>
        </matcher>

        <expression>odd.matches(formattedMessage)</expression>
      </evaluator>
      <OnMismatch>NEUTRAL</OnMismatch>
      <OnMatch>DENY</OnMatch>
    </filter>
    <layout>
      <pattern>%-4relative [%thread] %-5level %logger
- %msg%n</pattern>
    </layout>
  </appender>

  <root level="DEBUG">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

运行：

```
java chapters.filters.FilterEvents src/main/java/chapters/filters/evaluatorWithMatcher.xml
```

输出：

```
260 [main] INFO chapters.filters.FilterEvents - logging statement 0
264 [main] INFO chapters.filters.FilterEvents - logging statement 2
264 [main] INFO chapters.filters.FilterEvents - logging statement 4
266 [main] ERROR chapters.filters.FilterEvents - billing statement 6
266 [main] INFO chapters.filters.FilterEvents - logging statement 8
```

添加多个<matcher>元素，就等于定义更多匹配器。

## 7.1.2. TurboFilters

所有 TurboFilter 对象均继承自 TurboFilter 抽象类。像常规过滤器一样，它们用三值逻辑返回对记录事件的求值结果。

整体上说，它们与前面提到过的过滤器的工作原理非常相似。但是，与 Filter 对象相比，TurboFilter 有两个主要区别。

TurboFilter 对象被绑定到记录上下文。因此，不仅当指定的 appender 被使用时会调用 TurboFilter，而且每次执行记录请求时也会调用它们。它们的作用范围（scope）比关联到 appender 的过滤器大。

更重要的是，它们在 LoggingEvent 创建之前被调用。TurboFilter 对象过滤记录请求时不需要先初始化记录事件。TurboFilter 是用来高性能地过滤记录事件的，设置可以发生在记录事件被创建之前。

### 7.1.2.1. 自定义 TurboFilter

继承 TurboFilter 抽象类就创建了一个自定义 TurboFilter 组件。和常规过滤器一样，只需要实现 decide() 方法。下面的例子里，我们创建了一个稍微复杂的过滤器：

示例：基本自定义 TurboFilter

(logback-examples/src/main/java/chapters/filters/SampleTurboFilter.java)

```
package chapters.filters;

import org.slf4j.Marker;
import org.slf4j.MarkerFactory;

import ch.qos.logback.classic.Level;
import ch.qos.logback.classic.Logger;
import ch.qos.logback.classic.turbo.TurboFilter;
import ch.qos.logback.core.spi.FilterReply;

public class SampleTurboFilter extends TurboFilter {

    String marker;
    Marker markerToAccept;

    @Override
    public FilterReply decide(Marker marker, Logger logger, Level level,
        String format, Object[] params, Throwable t) {

        if (!isStarted()) {
            return FilterReply.NEUTRAL;
        }
    }
}
```

```
    }

    if ((markerToAccept.equals(marker))) {
        return FilterReply.ACCEPT;
    } else {
        return FilterReply.NEUTRAL;
    }
}

public String getMarker() {
    return marker;
}

public void setMarker(String markerStr) {
    this.marker = markerStr;
}

@Override
public void start() {
    if (marker != null && marker.trim().length() > 0) {
        markerToAccept = MarkerFactory.getMarker(marker);
        super.start();
    }
}
}
```

上面的 TurboFilter 接受包含特定 marker 的事件。如果上述 marker 未找到，则把任务交给过滤器链里的下一个过滤器。

为了灵活性，可以在配置文件里指定 marker，所以才有 getter 和 setting 方法。我们也实现了 start() 方法，检查在配置过程中是否指定了 marker 选项。

下面的配置文件使用了刚才创建的 TurboFilter。

示例：基本自定义 TurboFilter 配置

(logback-examples/src/main/java/chapters/filters/sampleTurboFilterConfig.xml)

```
<configuration>

  <turboFilter class="chapters.filters.SampleTurboFilter">
    <Marker>sample</Marker>
  </turboFilter>

</configuration>
```

```
<appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
    <layout class="ch.qos.logback.classic.PatternLayout">
        <pattern>%-4relative [%thread] %-5level %logger
- %msg%n</pattern>
    </layout>
</appender>

<root>
    <appender-ref ref="STDOUT" />
</root>
</configuration>
```

Logback-classic 带了一些可用的 TurboFilter 实现类。MDCFilter 类检查在 MDC 里是否存在一个给定值。DynamicThresholdFilter 类根据 MDC 的键与级别临界值之间的关联关系进行过滤。MarkerFilter 检查与记录事件相关联的某个特定 marker 是否存在。

下面的例子用到了 MDCFilter 和 MarkerFilter。

示例：MDCFilter 和 MarkerFilter 配置

(logback-examples/src/main/java/chapters/filters/turboFilters.xml)

```
<configuration>

    <turboFilter class="ch.qos.logback.classic.turbo.MDCFilter">
        <MDCKey>username</MDCKey>
        <Value>sebastien</Value>
        <OnMatch>ACCEPT</OnMatch>
    </turboFilter>

    <turboFilter class="ch.qos.logback.classic.turbo.MarkerFilter">
        <Marker>billing</Marker>
        <OnMatch>DENY</OnMatch>
    </turboFilter>

    <appender name="console"
class="ch.qos.logback.core.ConsoleAppender">
        <layout class="ch.qos.logback.classic.PatternLayout">
            <Pattern>%date [%thread] %-5level %logger - %msg%n</Pattern>
        </layout>
    </appender>

    <root level="info">
```

```
<appender-ref ref="console" />
</root>
</configuration>
```

执行：

```
java chapters.filters.FilterEvents src/main/java/chapters/filters/turboFilters.xml
```

回想一下，这个“FilterEvents”程序执行 10 次记录请求，从 0 到 9 编号。除了第 3 个和第 6 个请求，其他请求都是 INFO 级别，即与根 logger 同一个级别。第 3 个请求是 DEBUG 级别，比有效级别低。但是，因为 MDC 键“username”在第 3 个请求之前被设置为“sebastien”，而之后该键又被移除，所以 MDCFilter 接受了请求（仅该请求）。第 6 个请求是 ERROR 级别，被标记为“billing”，因此被 MarkerFilter（配置文件里的第二个 turbo filter）拒绝。

“FilterEvents”用配置“turboFilters.xml”运行后，输出如下：

```
2006-12-04 15:17:22,859 [main] INFO   chapters.filters.FilterEvents - logging statement 0
2006-12-04 15:17:22,875 [main] INFO   chapters.filters.FilterEvents - logging statement 1
2006-12-04 15:17:22,875 [main] INFO   chapters.filters.FilterEvents - logging statement 2
2006-12-04 15:17:22,875 [main] DEBUG chapters.filters.FilterEvents - logging statement 3
2006-12-04 15:17:22,875 [main] INFO   chapters.filters.FilterEvents - logging statement 4
2006-12-04 15:17:22,875 [main] INFO   chapters.filters.FilterEvents - logging statement 5
2006-12-04 15:17:22,875 [main] INFO   chapters.filters.FilterEvents - logging statement 7
2006-12-04 15:17:22,875 [main] INFO   chapters.filters.FilterEvents - logging statement 8
2006-12-04 15:17:22,875 [main] INFO   chapters.filters.FilterEvents - logging statement 9
```

可以看到显示了第 3 个请求，如果只是按照总体的 INFO 级别，它就不会显示，之所以显示，是因为它匹配了第一个 TurboFilter 的要求并且被接受了。

另一方面，第 6 个请求的级别是 ERROR，本来应该显示，但它满足了第二个 TurboFilter 的条件即 OnMatch 选项设为“DENY”，所以被拒绝而未显示。

### 7.1.3. 重复消息过滤器（DuplicateMessageFilter）

我们单独讲 DuplicateMessageFilter。DuplicateMessageFilter 检测重复的消息，如果重复次数超过某个值，就抛弃重复的消息。

为了检测重复消息，DuplicateMessageFilter 对消息采用简单的字符串相等性比较。不检测消息是否相似，即使只有个别字符不同也不检测。

注意，考虑到参数化记录的情况，仅仅比较原始消息。比如下面的例子里，两句的原始

消息都是 “Hello {}”，所以它们被认为是重复的。

```
logger.debug("Hello {}", name0);  
logger.debug("Hello {}", name1);
```

通过 “AllowedRepetitions” 属性指定允许的重复次数。例如，如果设为 1，则同一条消息之后的第 2 条会被抛弃。同样的，如果设为 2，则同一条消息之后的第 3 条会被抛弃。默认情况下，AllowedRepetitions 设为 5。

为了检测重复消息，DuplicateMessageFilter 需要在内部缓存里保持对旧消息的引用。缓存大小由 CacheSize 属性决定。默认情况下，CacheSize 是 100。

示例：DuplicateMessageFilter 配置

(logback-examples/src/main/java/chapters/filters/duplicateMessage.xml)

```
<configuration>  
  
  <turboFilter  
    class="ch.qos.logback.classic.turbo.DuplicateMessageFilter" />  
  
  <appender name="console"  
    class="ch.qos.logback.core.ConsoleAppender">  
    <layout class="ch.qos.logback.classic.PatternLayout">  
      <Pattern>%date [%thread] %-5level %logger - %msg%n</Pattern>  
    </layout>  
  </appender>  
  
  <root level="info">  
    <appender-ref ref="console" />  
  </root>  
</configuration>
```

输出：

```
2008-12-19 15:04:26,156 [main] INFO  chapters.filters.FilterEvents - logging statement 0  
2008-12-19 15:04:26,156 [main] INFO  chapters.filters.FilterEvents - logging statement 1  
2008-12-19 15:04:26,156 [main] INFO  chapters.filters.FilterEvents - logging statement 2  
2008-12-19 15:04:26,156 [main] INFO  chapters.filters.FilterEvents - logging statement 4  
2008-12-19 15:04:26,156 [main] INFO  chapters.filters.FilterEvents - logging statement 5  
2008-12-19 15:04:26,171 [main] ERROR chapters.filters.FilterEvents - billing statement 6
```

消息 “logging statement {}” 第一次出现是 “logging statement 0”，“logging statement 1”

是第 1 次重复,“logging statement 2”是第 2 次重复。有趣的是,“logging statement 3”是 DEBUG 级别,也是第 3 次重复,虽然它随后会依照基本选择规则被抛弃。这也用事实说明 TurboFilter 是在其他过滤器包括基本选择规则之前被调用的。DuplicateMessageFilter 无视“logging statement 3”将在过滤器链里被抛弃的事实,仍然把它算做第 3 次重复。“logging statement 4”是第 4 次重复,“logging statement 5”是第 5 次。从第 6 条语句起都被抛弃,因为默认只允许 5 次重复。

## 7.2. 在 logback-access 里

Logback-access 提供 logback-classic 的大部分特性。有 Filter 对象,且工作方式与 logback-classic 里的对应物一样,处理 logback-access 里的记录事件实现: AccessEvent。因此,logback-access 里的自定义过滤器与 logback-classic 里的自定义过滤器严格遵从同样的规则,唯一的区别是参数里的事件类型不同。另一方面,logback-access 支持 TurboFilter。

### 7.2.1. 过滤器

Logback-access 支持 EvaluatorFilter 对象,也是在配置文件的<evaluator>里设置 java 表达式。不过,对表达式暴露的隐式变量列表不同。表达式仅仅能使用变量名是“event”的 AccessEvent 对象。但是 logback-access 里的求值式已经足够强大,变量“event”可以访问请求和响应里的所有组件。

下面的配置例子确保任何 404 错误都会被记录。

示例: Logback-access 求值式

(logback-examples/src/main/java/chapters/filters/accessEventEvaluator.xml)

```
<configuration>

  <appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
    <filter class="ch.qos.logback.core.filter.EvaluatorFilter">
      <evaluator name="myEval">
        <expression>event.getStatusCode() == 404</expression>
      </evaluator>
      <OnMismatch>NEUTRAL</OnMismatch>
      <OnMatch>ACCEPT</OnMatch>
    </filter>
    <layout class="ch.qos.logback.access.PatternLayout">
      <pattern>
        %h %l %u %t %r %s %b
      </pattern>
    </layout>
  </appender>
</configuration>
```

```
    </layout>
  </appender>

  <appender-ref ref="STDOUT" />
</configuration>
```

稍微增加点复杂度，我们记录 404 错误，但不记录对访问 CSS 资源的 404 错误。如例中所示：

示例：Logback-access 求值式

(logback-examples/src/main/java/chapters/filters/accessEventEvaluator2.xml)

```
<configuration>

  <appender name="STDOUT"
    class="ch.qos.logback.core.ConsoleAppender">
    <filter class="ch.qos.logback.core.filter.EvaluatorFilter">
      <evaluator name="Eval404">
        <expression>event.getStatusCode() == 404</expression>
      </evaluator>
      <OnMismatch>NEUTRAL</OnMismatch>
      <OnMatch>ACCEPT</OnMatch>
    </filter>
    <filter class="ch.qos.logback.core.filter.EvaluatorFilter">
      <evaluator name="EvalCSS">

        <expression>event.getRequestURI().contains("css")</expression>
      </evaluator>
      <OnMismatch>NEUTRAL</OnMismatch>
      <OnMatch>DENY</OnMatch>
    </filter>
    <layout class="ch.qos.logback.access.PatternLayout">
      <pattern>
        %h %l %u %t %r %s %b
      </pattern>
    </layout>
  </appender>

  <appender-ref ref="STDOUT" />
</configuration>
```



## 8. 映射诊断环境（Mapped Diagnostic Context）

Logback 的设计目标之一是审查和调试复杂的分布式应用程序。真实世界的多数分布式系统需要同时处理多个客户端。在一个典型的多线程方式实现的分布式系统里，不同的线程处理不同的客户端。区分不同客户端的记录输出的一个可行的但不好的方法是为每个客户端都创建新的、独立的 **logger**。这种技术使 **logger** 的数量增多且大大增加了管理开销。

一个轻量的技术是为客户端的每个记录请求添加唯一戳（uniquely stamp）。Logback 在 SLF4J 里使用了这种技术的一种变体：映射诊断环境（MDC）。

为了给每个请求添加唯一戳，用户把环境（context）信息放进 MDC。MDC 类的重要部分如下，更多方法请参阅 MDC 的 javadocs 文档：

```
package org.slf4j;

public class MDC {
    // Put a context value as identified by key
    // into the current thread's context map.
    public static void put(String key, String val);

    // Get the context identified by the key parameter.
    public static String get(String key);

    // Remove the the context identified by the key parameter.
    public static void remove(String key);

    // Clear all entries in the MDC.
    public static void clear();
}
```

MDC 类只有静态方法，开发者可以把信息放进一个诊断环境，之后用其他 **logback** 组件获取这些信息。MDC 是基于每个线程进行管理的。子线程自动继承其父的映射诊断环境的一个副本。典型地，当开始为新的客户端请求服务时，开发者会向 MDC 里插入恰当的环境信息，比如客户端 id、客户端 IP 地址、请求参数等等。Logback 组件会自动在每个记录条目里包含这些信息。

下面的程序 `impleMDC` 演示了这一基本原则。

示例：MDC 基本用法

(`logback-examples/src/main/java/chapters/mdc/SimpleMDC.java`)

```
package chapters.mdc;

import java.net.URL;
```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.slf4j.MDC;

import ch.qos.logback.classic.LoggerContext;
import ch.qos.logback.classic.encoder.PatternLayoutEncoder;
import ch.qos.logback.classic.joran.JoranConfigurator;
import ch.qos.logback.classic.spi.ILoggingEvent;
import ch.qos.logback.core.ConsoleAppender;
import ch.qos.logback.core.joran.spi.JoranException;
import ch.qos.logback.core.util.Loader;
import ch.qos.logback.core.util.StatusPrinter;

public class SimpleMDC {
    static public void main(String[] args) throws Exception {
        // You can put values in the MDC at any time. Before anything else
        // we put the first name
        MDC.put("first", "Dorothy");
        [省略]
        Logger logger = LoggerFactory.getLogger(SimpleMDC.class);
        // We now put the last name
        MDC.put("last", "Parker");

        // The most beautiful two words in the English language according
        // to Dorothy Parker:
        logger.info("Check enclosed.");
        logger.debug("The most beautiful two words in English.");

        MDC.put("first", "Richard");
        MDC.put("last", "Nixon");
        logger.info("I am not a crook.");
        logger.info("Attributed to the former US president. 17 Nov
1973.");
    }

    [省略]
}
```

`main()`方法先在 MDC 里关联键“first”和值“Dorothy”。你可以在 MDC 里放入任何数量的键/值对。同名的键会覆盖旧键的值。代码接着配置 logback。

为了简洁，我们忽略了用配置文件 `simpleMDC.xml` 对 logback 进行配置的代码，下面是

配置文件里的相关部分：

```
<appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
  <layout>
    <Pattern>%X{first} %X{last} - %m%n</Pattern>
  </layout>
</appender>
```

注意“%X”在 `PatternLayout` 里的用法。“%X”用了两次，一次是为键“first”，一次是为键“last”。代码先取得对应于“`SimpleMDC.class`”的 `logger`，接着为键“last”关联一个值“Parker”，然后用不同的消息两次调用 `logger`，最后把 MDC 设为其他值并执行几条记录请求。

运行 `SimpleMDC`，输出：

```
Dorothy Parker - Check enclosed.
Dorothy Parker - The most beautiful two words in English.
Richard Nixon - I am not a crook.
Richard Nixon - Attributed to the former US president. 17 Nov 1973.
```

`SimpleMDC` 程序演示了 `logback` 的 `layout` 如何自动输出 MDC 信息。而且，放在 MDC 里的信息能被 `logger` 多次使用。

## 8.1. 高级用法

映射诊断环境在客户端-服务器模式下最有奇效。典型情况是，多个客户端被服务器的多个线程所处理。虽然 MDC 里的方法都是静态的，但 MDC 是基于每个线程进行管理的，这允许服务器的每个线程都有自己独立的 MDC 戳。MDC 的操作，比如 `put()` 和 `get()`，只作用于当前线程和当前线程的子线程，不影响其他线程里的 MDC。由于 MDC 的信息是基于每个线程进行管理的，因此每个线程都有自己的 MDC 副本。所以在使用 MDC 时，开发者不需要操心线程安全或同步，因为 MDC 透明地、安全地处理了这些问题。

下面的例子更高级点，演示了 MDC 如何用于客户端-服务器模式。服务器端实现 `NumberCruncher` 接口。`NumberCruncher` 接口只包含一个方法 `factor()`。使用 RMI 技术，客户端调用服务器程序的 `factor()` 方法来获得整数的因数。

示例：service 接口

(`logback-examples/src/main/java/chapters/mdc/NumberCruncher.java`)

```
package chapters.mdc;
```

```
import java.rmi.Remote;
import java.rmi.RemoteException;

/**
 * NumberCruncher factors positive integers.
 */
public interface NumberCruncher extends Remote {
    /**
     * Factor a positive integer <code>number</code> and return its
     * <em>distinct</em> factor's as an integer array.
     */
    int[] factor(int number) throws RemoteException;
}
```

下面的 NumberCruncherServer 程序，实现 NumberCruncher 接口，main()方法在本机导出一个 RMI Registry，接收来自默认端口（1099）的请求。

示例：服务器端

(logback-examples/src/main/java/chapters/mdc/NumberCruncherServer.java)

```
package chapters.mdc;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;

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

import ch.qos.logback.classic.LoggerContext;
import ch.qos.logback.classic.joran.JoranConfigurator;
import ch.qos.logback.core.joran.spi.JoranException;

/**
 * A simple NumberCruncher implementation that logs its progress when
 * factoring
 * numbers. The purpose of the whole exercise is to show the use of mapped
 * diagnostic contexts in order to distinguish the log output from
```

```
different
 * client requests.
 * */
public class NumberCruncherServer extends UnicastRemoteObject implements
    NumberCruncher {

    private static final long serialVersionUID = 1L;

    static Logger logger =
        LoggerFactory.getLogger(NumberCruncherServer.class);

    public NumberCruncherServer() throws RemoteException {
    }

    public int[] factor(int number) throws RemoteException {
        // The client's host is an important source of information.
        try {
            MDC.put("client", NumberCruncherServer.getClientHost());
        } catch (java.rmi.server.ServerNotActiveException e) {
            logger.warn("Caught unexpected ServerNotActiveException.",
e);
        }

        // The information contained within the request is another source
        // of distinctive information. It might reveal the users name,
        // date of request, request ID etc. In servlet type environments,
        // useful information is contained in the HttpRequest or in the
        // HttpSession.
        MDC.put("number", String.valueOf(number));

        logger.info("Beginning to factor.");

        if (number <= 0) {
            throw new IllegalArgumentException(number
                + " is not a positive integer.");
        } else if (number == 1) {
            return new int[] { 1 };
        }

        Vector<Integer> factors = new Vector<Integer>();
        int n = number;

        for (int i = 2; (i <= n) && ((i * i) <= number); i++) {
            // It is bad practice to place log requests within tight loops.

```

```
// It is done here to show interleaved log output from
// different requests.
logger.debug("Trying " + i + " as a factor.");

if ((n % i) == 0) {
    logger.info("Found factor " + i);
    factors.addElement(new Integer(i));

    do {
        n /= i;
    } while ((n % i) == 0);
}

// Placing artificial delays in tight loops will also lead to
// sub-optimal results. :-)
delay(100);
}

if (n != 1) {
    logger.info("Found factor " + n);
    factors.addElement(new Integer(n));
}

int len = factors.size();

int[] result = new int[len];

for (int i = 0; i < len; i++) {
    result[i] = ((Integer) factors.elementAt(i)).intValue();
}

// clean up
MDC.remove("client");
MDC.remove("number");

return result;
}

static void usage(String msg) {
    System.err.println(msg);
    System.err
        .println("Usage: java chapters.mdc.NumberCruncherServer
configFile\n"
               + "    where configFile is a logback configuration
```

```
file.");
    System.exit(1);
}

public static void delay(int millis) {
    try {
        Thread.sleep(millis);
    } catch (InterruptedException e) {
    }
}

public static void main(String[] args) {
    if (args.length != 1) {
        usage("Wrong number of arguments.");
    }

    String configFile = args[0];

    if (configFile.endsWith(".xml")) {
        try {
            LoggerContext lc = (LoggerContext) LoggerFactory
                .getILoggerFactory();
            JoranConfigurator configurator = new JoranConfigurator();
            configurator.setContext(lc);
            lc.reset();
            configurator.doConfigure(args[0]);
        } catch (JoranException je) {
            je.printStackTrace();
        }
    }

    NumberCruncherServer ncs;

    try {
        ncs = new NumberCruncherServer();
        logger.info("Creating registry.");

        Registry registry = LocateRegistry
            .createRegistry(Registry.REGISTRY_PORT);
        registry.rebind("Factor", ncs);
        logger.info("NumberCruncherServer bound and ready.");
    } catch (Exception e) {
        logger.error("Could not bind NumberCruncherServer.", e);
    }
}
```

```
        return;  
    }  
}  
}
```

`factor(int number)`方法的实现特别重要，先把客户端主机名放进 MDC 的键“client”，把客户端请求的整数放进 MDC 键“number”，计算完整数的因数后，把结果返回给客户端。在返回结果之前，调用 MDC 的 `remove()`方法删除键“client”和键“number”的值。正常情况下，`add()`操作应该要有对应的 `remove()`操作，否则 MDC 将保持某些键的旧值。我们建议尽可能在 `finally` 块里执行 `remove()`操作。

启动服务器：

```
java chapters.mdc.NumberCruncherServer src/main/java/chapters/mdc/mdc1.xml
```

`mdc1.xml` 配置文件如下。

示例：配置文件(`logback-examples/src/main/java/chapters/mdc/mdc1.xml`)

```
<configuration>  
  
    <appender name="CONSOLE"  
class="ch.qos.logback.core.ConsoleAppender">  
        <layout class="ch.qos.logback.classic.PatternLayout">  
            <Pattern>%-4r [%thread] %-5level C:%X{client} N:%X{number}  
- %msg%n  
        </Pattern>  
    </layout>  
    </appender>  
  
    <root>  
        <level value="debug" />  
        <appender-ref ref="CONSOLE" />  
    </root>  
</configuration>
```

注意 Pattern 选项里的格式转换符“%X”的用法。

启动客户端：

```
java chapters.mdc.NumberCruncherClient hostname
```

把“hostname”改成服务器 `NumberCruncherServer` 的主机名。



运行多个客户端，一个客户端请求 129 的因数，稍后另一个客户端请求 71 的因数，服务器输出如下：

```
70984 [RMI TCP Connection(4)-192.168.1.6] INFO   C:orion N:129 - Beginning to factor.
70984 [RMI TCP Connection(4)-192.168.1.6] DEBUG C:orion N:129 - Trying 2 as a factor.
71093 [RMI TCP Connection(4)-192.168.1.6] DEBUG C:orion N:129 - Trying 3 as a factor.
71093 [RMI TCP Connection(4)-192.168.1.6] INFO   C:orion N:129 - Found factor 3
71187 [RMI TCP Connection(4)-192.168.1.6] DEBUG C:orion N:129 - Trying 4 as a factor.
71297 [RMI TCP Connection(4)-192.168.1.6] DEBUG C:orion N:129 - Trying 5 as a factor.
71390 [RMI TCP Connection(4)-192.168.1.6] DEBUG C:orion N:129 - Trying 6 as a factor.
71453 [RMI TCP Connection(5)-192.168.1.6] INFO   C:orion N:71 - Beginning to factor.
71453 [RMI TCP Connection(5)-192.168.1.6] DEBUG C:orion N:71 - Trying 2 as a factor.
71484 [RMI TCP Connection(4)-192.168.1.6] DEBUG C:orion N:129 - Trying 7 as a factor.
71547 [RMI TCP Connection(5)-192.168.1.6] DEBUG C:orion N:71 - Trying 3 as a factor.
71593 [RMI TCP Connection(4)-192.168.1.6] DEBUG C:orion N:129 - Trying 8 as a factor.
71656 [RMI TCP Connection(5)-192.168.1.6] DEBUG C:orion N:71 - Trying 4 as a factor.
71687 [RMI TCP Connection(4)-192.168.1.6] DEBUG C:orion N:129 - Trying 9 as a factor.
71750 [RMI TCP Connection(5)-192.168.1.6] DEBUG C:orion N:71 - Trying 5 as a factor.
71797 [RMI TCP Connection(4)-192.168.1.6] DEBUG C:orion N:129 - Trying 10 as a factor.
71859 [RMI TCP Connection(5)-192.168.1.6] DEBUG C:orion N:71 - Trying 6 as a factor.
71890 [RMI TCP Connection(4)-192.168.1.6] DEBUG C:orion N:129 - Trying 11 as a factor.
71953 [RMI TCP Connection(5)-192.168.1.6] DEBUG C:orion N:71 - Trying 7 as a factor.
72000 [RMI TCP Connection(4)-192.168.1.6] INFO   C:orion N:129 - Found factor 43
72062 [RMI TCP Connection(5)-192.168.1.6] DEBUG C:orion N:71 - Trying 8 as a factor.
72156 [RMI TCP Connection(5)-192.168.1.6] INFO   C:orion N:71 - Found factor 71
```

从上可见，这些客户端的主机名是“orion”。即使客户端请求几乎同时来自不同线程，服务器仍然可以通过 MDC 区分记录请求属于哪个客户端。注意例子里与“number”即被分解的数字相关联的印戳。

留心的读者或许已经观察到线程名或许也可以用于区分各个请求。当服务器端循环使用线程时，用线程名会导致混乱，因为难以确定每个请求的边界，边界是指当指定线程处理完一个请求并开始处理下一个请求时。因为 MDC 是被程序开发者管理的，所以 MDC 印戳没有这个问题。

## 8.2. 自动访问 MDC

我们已经看到，在处理多个客户端时，MDC 非常有用。在一个管理用户验证的 web 程序里，可以在 MDC 里设置用户名，然后在用户注销登录时从 MDC 删除用户名。不幸的是，上面的方法不是始终可靠的。由于 MDC 是基于每个线程对数据进行管理的，所以服务器循环使用线程时会导致错误地使用 MDC 里的信息。

在处理请求时，为保证 MDC 里的信息始终正确，一个可行的方法是，在处理流程的开头存储用户名，然后再这个处理流程的尾部移除用户名。这种情况可以用一个 servlet 过滤器。

在 servlet 过滤器的 `doFilter()` 方法里，我们可以从请求（或 cookie）取得相关的用户数据，然后存储在 MDC。后续的其他过滤器和 servlet 将自动从先前存储的 MDC 里受益。最终，当我们的 servlet 过滤器重新得到控制权后，就有机会清除 MDC 数据。

下面是这种过滤器的一个实现。

示例：用户 servlet 过滤器

(logback-examples/src/main/java/chapters/mdc/UserServletFilter.java)

```
package chapters.mdc;

import java.io.IOException;
import java.security.Principal;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;

import org.slf4j.MDC;

public class UserServletFilter implements Filter {

    private final String USER_KEY = "username";

    public void destroy() {
    }

    public void doFilter(ServletRequest request, ServletResponse
response,
```

```
FilterChain chain) throws IOException, ServletException {

    boolean successfulRegistration = false;
    HttpServletRequest req = (HttpServletRequest) request;
    Principal principal = req.getUserPrincipal();
    // Please note that we also could have used a cookie to
    // retrieve the user name

    if (principal != null) {
        String username = principal.getName();
        successfulRegistration = registerUsername(username);
    }

    try {
        chain.doFilter(request, response);
    } finally {
        if (successfulRegistration) {
            MDC.remove(USER_KEY);
        }
    }
}

public void init(FilterConfig arg0) throws ServletException {

    /**
     * Register the user in the MDC under USER_KEY.
     *
     * @param username
     * @return true id the user can be successfully registered
     */
    private boolean registerUsername(String username) {
        if (username != null && username.trim().length() > 0) {
            MDC.put(USER_KEY, username);
            return true;
        }
        return false;
    }
}
```

当该过滤器的 `doFilter()` 方法被调用时，先在请求里查找 `java.security.Principal` 对象，该对象包含当前授权用户的用户名，如果找到了用户名，就注册在 MDC 里。

过滤器链完成后，该过滤器从 MDC 移除用户信息。

上面的这种方法设置的 MDC 数据只在请求期和处理 MDC 的线程里有效，对其他线程无效。

## 8.3. MDC 和受管线程

当用 `java.util.concurrent.Executors` 来管理线程时，初始线程的 MDC 副本不是总能被工作线程所继承。例如，`newCachedThreadPool()`方法创建一个 `ThreadPoolExecutor`，像其他线程池代码一样，它创建线程的逻辑很复杂。

在这种情况下，建议在把 task 提交给 executor 之前，在初始（主）线程上调用 `MDC.getCopyOfContextMap()`。当 task 运行时，它的第一个动作就应该是调用 `MDC.setContextMapValues()`，为新的 Executor 受管线程关联初始 MDC 值的副本。

### 8.3.1. MDCInsertingServletFilter

在 web 程序里，经常需要取得 http 请求的主机名、请求 uri 和 user-agent。`MDCInsertingServletFilter` 把这些数据放入 MDC，所用的键如下：

MDC 键	MDC 值
<code>req.remoteHost</code>	与 <code>getRemoteHost()</code> 返回的一样
<code>req.xForwardedFor</code>	头 “X-Forwarded-For” 的值
<code>req.requestURI</code>	与 <code>getRequestURI()</code> 返回的一样
<code>req.requestURL</code>	与 <code>getRequestURL()</code> 返回的一样
<code>req.queryString</code>	与 <code>getQueryString()</code> 返回的一样
<code>req.userAgent</code>	头 “User-Agent” 的值

安装 `MDCInsertingServletFilter` 方法是在 web 程序的 `web.xml` 里添加：

```
<filter>
  <filter-name>MDCInsertingServletFilter</filter-name>
  <filter-class>
    ch.qos.logback.classic.helpers.MDCInsertingServletFilter
  </filter-class>
</filter>
<filter-mapping>
  <filter-name>MDCInsertingServletFilter</filter-name>
```

```
<url-pattern>/*</url-pattern>  
</filter-mapping>
```

如果你的 web 程序里有多个过滤器，请把 `MDCInsertingServletFilter` 放在第一个。

安装后 `MDCInsertingServletFilter` 后，MDC 的各个键的值会被转换符 “%X” 输出。例如，为了在一行里打印主机名和请求 URI，下一行打印日期和消息，可以设置 `PatternLayout` 的 layout 为：

```
%X{req.remoteHost} %X{req.requestURI}%n%d - %m%n
```

## 9. 记录隔离

本章讲述如何在运行于同一个 web 或 EJB 容器里的多个应用程序实现记录分离。在本章的余下部分，术语“application”即表示 web 应用程序，也表示 J2EE 应用程序，两种意思可以互换。在记录隔离的环境里，每个应用程序使用互不相同的记录环境，因此一个应用程序的 logback 的配置与其他应用程序的配置互不干涉。用专业术语讲就是，每个 web 应用程序拥有为自己保留的与众不同的 LoggerContext 副本（copy）。在 logback 里，每个 logger 对象都是由 LoggerContext 制造的，只要 logger 对象还存活于内存，它就始终关联到 LoggerContext。记录隔离的另一个场景是应用程序与其容器之间的记录隔离。

### 9.1. 最简易的方法

假设你的容器支持优先加载应用程序的类，那么在每个应用程序里都放入 slf4j 和 logback 的 jar 文件就能实现记录隔离。对于 web 应用程序，只需要把 slf4j 和 logback 的 jar 文件放到 web 应用程序的 WEB-INF/lib 目录即可，配置文件 logback.xml 放在 WEB-INF/classes 目录下。

得益于容器的类加载隔离机制，每个 web 应用程序将从它自己的 logback.xml 加载自己的 LoggerContext。

够简单吧。

但是，也有例外。有时你被迫把 SLF4J 和 logback 文件放到一个供所有应用程序共享的地方，一个典型情形是某个共享类库用到 SLF4J。在这种情况下，所有应用程序将共享同一个记录环境。还有其他各种共享 SLF4J 和 logback 的情形，导致通过类加载隔离实现记录隔离的方法失效。希望还没彻底破灭，请往下读。

### 9.2. 上下文选择器（ContextSelector）

Logback 提供一种机制，加载到内存的 SLF4J 和 logback 的一个实例就可以提供多个 logger 上下文。当写下：

```
Logger logger = LoggerFactory.getLogger("foo");
```

LoggerFactory 类的 getLogger() 方法会要求 SLF4J 绑定一个 ILoggerFactory。当 SLF4J 绑定到 logback 时，返回 ILoggerFactory 的任务交给了 ContextSelector 实例。注意 ContextSelector 的所有实现总是返回 LoggerContext 实例。LoggerContext 类实现了 ILoggerFactory 接口，换句话说，context selector 有能力根据自己的标准返回任何它认为合

适的 `LoggerContext` 实例。

默认情况下,logback 用 `DefaultContextSelector` 进行绑定,始终返回同一个 `LoggerContext`,称为默认 logger 上下文。

你可以通过设置系统属性 “logback.ContextSelector” 来指定不同的上下文选择器。假设你想把上下文选择器指定给 `myPackage.myContextSelector` 类实例,可以这样指定系统属性:

```
-Dlogback.ContextSelector=myPackage.myContextSelector
```

上下文选择器需要实现 `ContextSelector` 接口,至少实现一个只含 `LoggerContext` 参数的构造方法。

### 9.2.1. ContextJNDISelector

Logback-classic 有一个叫 “ContextJNDISelector” 的选择器,它基于 JNDI 查找到的数据进行 logger 上下文选择。这种方法利用了 J2EE 规范规定的 JNDI 数据隔离机制。因此,同一个环境变量可在不同的应用程序里设为不同的值。换句话说,在不同的应用程序里调用 `LoggerFactory.getLogger()` 方法会返回关联于不同 logger 上下文的 logger,即使内存里只有一个被所有应用程序共享的 `LoggerFactory` 类。对你来说就是记录隔离。

为启用 `ContextJNDISelector`,需要设置系统属性 “logback.ContextSelector”:

```
-Dlogback.ContextSelector=JNDI
```

注意这里的 “JNDI” 是 “ch.qos.logback.classic.selector.ContextJNDISelector” 的简写。

### 9.2.2. 在应用程序里设置 JNDI 变量

需要为各个应用程序命名记录上下文。对于一个 web 应用程序,在 `web.xml` 文件里指定 JNDI 项。如果你的应用程序叫 “kenobi”,可以在 kenobi 的 `web.xml` 文件里文件如下内容:

```
<env-entry>
  <env-entry-name>logback/context-name</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>kenobi</env-entry-value>
</env-entry>
```

假设已经启用了 `ContextJNDISelector`,则会用名为 “kenobi” 的 logger 上下文负责 Kenobi 的记录。而且,按照约定,名为 “kenobi” 的 logger 上下文用当前上下文类加载器查找被视为资源 (resource) 的配置文件 “logback-kenobi.xml”,从而进行初始化。对 web 应用程序 kenobi, `logback-kenobi.xml` 应当被放在 `WEB-INF/classes` 目录。

你也可以不遵守约定，自己指定一个不同的配置文件，方法是设置 JNDI 变量“logback/configuration-resource”。例如，对 web 应用程序 kenobi，如果你想用“aFolder/my\_config.xml”而不是约定中的“logback-kenobi.xml”，可以在 web.xml 里添加如下内容：

```
<env-entry>
  <env-entry-name>logback/configuration-resource</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>aFolder/my_config.xml</env-entry-value>
</env-entry>
```

文件“my\_config.xml”应当放在“WEB-INF/classes/aFolder/”目录。重点是记住是用当前线程的上下文类加载器把配置文件作为 java 资源进行查找的。

### 9.2.3. 为 Tomcat 配置 ContextJNDISelector

首先，把 logback 的 jar 文件（logback-classic-0.9.20.jar、logback-core-0.9.20.jar 和 slf4j-api-1.5.11.jar）防盜 Tomcat 的全局（共享）类目录。对于 Tomcat 6.x 是“\$TOMCAT\_HOME/lib/”。

设置“logback.ContextSelector”系统属性的方法是修改“\$TOMCAT\_HOME/bin”目录下的在 catalina.sh 脚本（对于 UNIX）、catalina.bat（对于 Windows）文件，添加：

```
JAVA_OPTS="$JAVA_OPTS -Dlogback.ContextSelector=JNDI"
```

#### 9.2.3.1. 热部署应用程序

当 web 应用程序重启或关闭时，我们强烈建议关闭使用中的 LoggerContext，从而可被正常地垃圾回收。Logback 带了叫“ContextDetachingSCL”的 ServletContextListener，专门用来分离那些被关联到旧 web 应用程序实例的 ContextSelector 实例。安装方法是在 web.xml 里添加：

```
<listener>
  <listener-class>ch.qos.logback.classic.selector.servlet.ContextDe
tachingSCL</listener-class>
</listener>
```

#### 9.2.3.2. 更好的性能

当 ContextJNDISelector 处于活动状态时，每次获取 logger 时都要查找 JNDI，这会引引起性能下降，尤其是在使用非静态 logger 引用时。Logback 带里一个 servlet 过滤器叫



“LoggerContextFilter”，专门用于避免查找 JNDI 带来的消耗。安装方法是在 web.xml 里添加：

```
<filter>
  <filter-name>LoggerContextFilter</filter-name>
  <filter-class>ch.qos.logback.classic.selector.servlet.LoggerContextFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>LoggerContextFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

在每个 HTTP 请求的开头，LoggerContextFilter 会获取与应用程序关联的 logger 上下文，然后把上下文放到一个 ThreadLocal 变量。ContextJNDISelector 将首先检查是否有上述 ThreadLocal 变量，如果有，则跳过 JNDI 查找。注意在 HTTP 请求的最后，这个 ThreadLocal 变量被置为 null。安装 LoggerContextFilter 后极大地改进了获取 logger 的性能。

把这个 ThreadLocal 变量置为 null 可以让 web 应用程序在重启或停止时能够被执行垃圾回收。

### 9.3. 在共享类库里使用静态引用

当 SLF4J 和 logback 被所有应用程序共享时，ContextJNDISelector 很好地实现了记录分离。当 ContextJNDISelector 处于活动状态时，每次调用 LoggerFactory.getLogger() 所返回的 logger 属于与调用者/当前应用程序相关联的 logger 上下文。

引用 logger 的常用做法是通过静态引用。例如，

```
public class Foo {
  static Logger logger = LoggerFactory.getLogger(Foo.class);
  ...
}
```

静态 logger 引用对内存和 CPU 来说都高效。一个类的所有实例均使用同一个 logger 引用。而且，当类被加载到内存时，logger 实例仅仅被获取一次。如果宿主类属于某个应用程序，假设是 kenobi，那么静态 logger 将被 ContextJNDISelector 关联到 kenobi 的 logger 上下文。与此相似，如果宿主类属于某个其他应用程序，假设是 yoda，那么对它的静态引用将被 ContextJNDISelector 关联到 yoda 的 logger 上下文。

如果一个类，假设是 Mustafar，属于被 kenobi 和 yoda 所共享的类库，那么只要 Mustafar 有非静态 logger，对 LoggerFactory.getLogger() 的每一次调用都会返回一次属于与调用者/当

前应用程序相关联的 `logger` 上下文。但是如果 Mustafar 有一个静态 `logger` 引用，那么这个 `logger` 就会被关联到第一个调用它的应用程序的 `logger` 上下文。因此，`ContextJNDISelector` 不能为这种使用了静态 `logger` 引用的共享类提供记录隔离。这种情况是条死路。

透明地、完美地解决这个问题的唯一方法也许是在 `logger` 内部引入另外一个间接级别，以便每个 `logger` 外壳 (shell) 把工作委托给与适当的上下文所关联的 `logger`。这个方法非常难以实现，也会导致明显增加计算量。我们不打算这么做。

当然，只要简单地把共享类移到 web 应用程序内部即不共享，就可以轻易解决所谓的“共享类的静态 `logger`”问题。如果无法不共享，我们还可以使用 `SiftingAppender` 实现记录隔离，它用 JNDI 数据作为隔离条件。

Logback 带了一个鉴别器 `JNDIBasedContextDiscriminator`，它返回经 `ContextJNDISelector` 计算的当前 `logger` 上下文的名称。`SiftingAppender` 与 `JNDIBasedContextDiscriminator` 的组合可以为每个 web 应用程序创造各自的 `appender`。

```
<configuration>

  <statusListener
class="ch.qos.logback.core.status.OnConsoleStatusListener" />

  <appender name="SIFT"
class="ch.qos.logback.classic.sift.SiftingAppender">
    <discriminator

      class="ch.qos.logback.classic.sift.JNDIBasedContextDiscriminator"
    >
      <defaultValue>unknown</defaultValue>
    </discriminator>
    <sift>
      <appender name="FILE-${contextName}"
class="ch.qos.logback.core.FileAppender">
        <file>${contextName}.log</file>
        <encoder>
          <pattern>%-50(%level %logger{35}) cn=%contextName
- %msg%n</pattern>
        </encoder>
      </appender>
    </sift>
  </appender>

  <root level="DEBUG">
    <appender-ref ref="SIFT" />
  </root>
```

```
</configuration>
```

如果 `kenobi` 和 `yoda` 是 web 应用程序, 那么上面的配置会把 `yoda` 的记录输出到 `yoda.log`, 把 `kenobi` 的记录输出到 `kenobi.log`。这些公平的记录是由共享类引用的静态 `logger` 生成的。

`logback-starwars` 项目 (<http://github.com/ceki/logback-starwars>) 演示了上面的技术, 你可以参考它。

上述方法解决了记录隔离问题但相当复杂。需要正确安装 `ContextJNDISelector`, 同时要求用 `SiftingAppender` 包裹 `appender`。

注意每个记录上下文可以同一个或不同的配置文件进行配置。你自己做主。让所有上下文都是用同一个配置文件比较简单, 因为只需要维护一个文件。为每个应用程序使用不同的配置文件不易维护但却更灵活。

所有问题都解决了? 我们可以宣告胜利然后回家了吗? 嗯, 还没。

假设 web 应用程序 `yoda` 在 `kenobi` 之前初始化。为了初始化 `yoda`, 访问 `http://localhost:port/yoda/servlet` 会调用 `YodaServlet`, 这个 `servlet` 只是说 “hello”, 接着在调用 `Mustafar` 类的 `foo()` 方法前记录消息, `foo()` 方法只是简单地记录消息并返回。

在 `YodaServlet` 被调用之后, `yoda.log` 文件应该包含:

```
DEBUG ch.qos.starwars.yoda.YodaServlet          cn=yoda - in doGet()
DEBUG ch.qos.starwars.shared.Mustafar           cn=yoda - in foo()
```

注意这两个 `log` 项都关联到名为 “`yoda`” 的上下文。从此时直到服务器停止, `logger` “`ch.qos.starwars.shared.Mustafar`” 也被关联到 “`yoda`” 上下文直至服务器停止。

访问 `http://localhost:port/kenobi/servlet` 会在 `kenobi.log` 里输出:

```
DEBUG ch.qos.starwars.kenobi.KenobiServlet      cn=kenobi - in doGet()
DEBUG ch.qos.starwars.shared.Mustafar           cn=yoda - in foo()
```

注意虽然 `logger` “`ch.qos.starwars.shared.Mustafar`” 输出到 `kenobi.log`, 但仍然关联到 “`yoda`”。所以, 两个不同的记录上下文都输出到同一个文件, 本例中视 “`kenobi.log`”。

每个上下文都引用嵌套在 `SiftingAppender` 里的 `FileAppender` 实例, 输出到同一个文件。尽管似乎如我们所愿实现了记录隔离, 但是除非启用 “`prudent`” 模式, 否则 `FileAppender` 不能安全地写入同一个文件, 目标文件也会被破坏。

下面是启用了 `prudent` 模式的配置文件:

```
<configuration>
```

```
<statusListener
class="ch.qos.logback.core.status.OnConsoleStatusListener" />

<appender name="SIFT"
class="ch.qos.logback.classic.sift.SiftingAppender">
  <discriminator
    class="ch.qos.logback.classic.sift.JNDIBasedContextDiscriminator"
  >
    <defaultValue>unknown</defaultValue>
  </discriminator>
  <sift>
    <appender name="FILE-${contextName}"
class="ch.qos.logback.core.FileAppender">
      <file>${contextName}.log</file>
      <prudent>true</prudent>
      <encoder>
        <pattern>%-50(%level %logger{35}) cn=%contextName
- %msg%n</pattern>
      </encoder>
    </appender>
  </sift>
</appender>

<root level="DEBUG">
  <appender-ref ref="SIFT" />
</root>
</configuration>
```

如果你越读越困惑，请联系专业的技术支持 (<http://www.qos.ch/shop/products/professionalSupport>)。

## 10. JMX 配置器

JMX 配置器允许通过 JMX 配置 logback。它让你可以用默认配置文件、指定的文件或 URL 来重新配置 logback，还可以列出 logger、修改 logger 级别。

### 10.1. 使用 JMX 配置器

如果你的服务器运行在 JDK 1.6 或更高版本，那么你可以直接运行 jconsole 程序，然后连接到你的服务器的 MBeanServer。如果 JDK 版本低，请读本章内的“[支持 JMX](#)”。

一行配置就能启用 JMXConfigurator，如下：

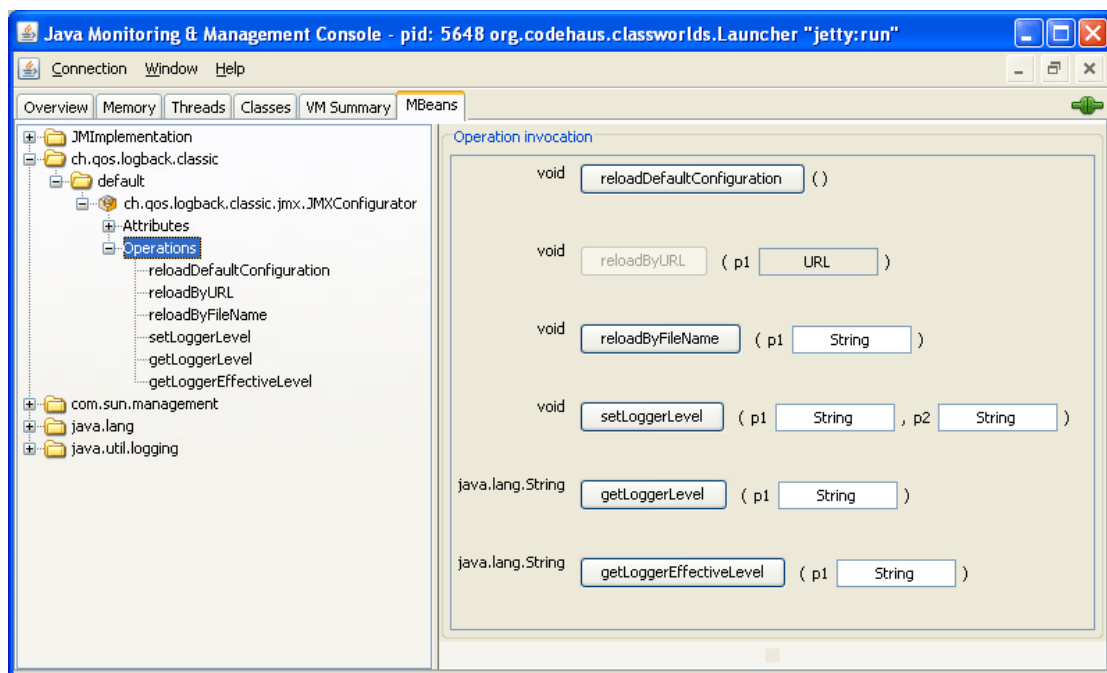
```
<configuration>
  <jmxConfigurator />

  <appender name="console"
class="ch.qos.logback.classic.ConsoleAppender">
    <layout class="ch.qos.logback.classic.PatternLayout">
      <Pattern>%date [%thread] %-5level %logger{25}
- %msg%n</Pattern>
    </layout>
  </appender>

  <root level="debug">
    <appender-ref ref="console" />
  </root>
</configuration>
```

用 jconsole 连接到你的服务器后，在 MBean 面板上，在“ch.qos.logback.classic.jmx.Configurator”目录下，你应该能看到一些操作选项，如下图所示：

jconsole 里的 JMXConfigurator 截图：

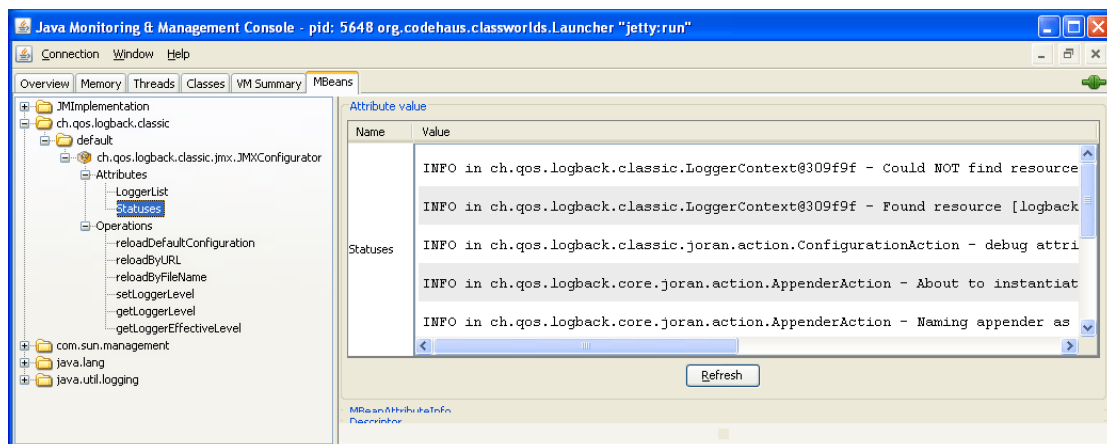


你可以

- 用默认配置文件重新加载 logback 配置。
- 用指定的 URL 重新加载配置。
- 用指定的文件重新加载配置。
- 设置 logger 的级别。欲设置为 null，则输入字符串“null”。
- 取得 logger 的级别。返回值可以为 null。
- 取得 logger 的有效级别。

JMXConfigurator 的属性显示了现存 logger 的列表和状态列表。

状态列表有助于诊断 logback 的内部状态。



## 10.2. 避免内存泄露

如果你的应用程序部署于 web 服务器或应用程序服务器，注册 JMXConfigurator 实例时会从系统的类加载器往你的应用程序里创建一个引用，目的是防止在你的应用程序停止或重新部署时被垃圾回收，但是这导致了严重的内存泄露。

因此，除非你的应用程序是独立运行的 java 程序，否则你必须从 JVM 的 Mbean 服务器上卸载 JMXConfigurator。调用 LoggerContext 的 reset() 方法会自动卸载任何 JMXConfigurator 实例。重置 logger 上下文的一个好地方是 javax.servlet.ServletContextListener 的 contextDestroyed() 方法，实例代码如下：

```
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

import org.slf4j.LoggerFactory;
import ch.qos.logback.classic.LoggerContext;

public class MyContextListener implements ServletContextListener {

    public void contextDestroyed(ServletContextEvent sce) {
        LoggerContext lc = (LoggerContext)
LoggerFactory.getILoggerFactory();
        lc.stop();
    }

    public void contextInitialized(ServletContextEvent sce) {
    }

}
```

## 10.3. 多个应用程序里的 JMXConfigurator

如果在同一个服务器里部署了多个应用程序，假设你没有覆盖默认的上下文选择器，假设 logback-\*.jar 和 slf4j-api.jar 放在每个 web 应用程序的 WEB-INF/lib 目录下，此时，默认情况下每个 JMXConfigurator 实例都会用同一个名字注册，即“ch.qos.logback.classic:Name=default,Type=ch.qos.logback.classic.jmx.JMXConfigurator”。换句话说，默认情况下，与每个 web 应用程序相关联的 JMXConfigurator 各个实例会起冲突。

为避免冲突，你可以简单地应用程序设置记录[上下文名称](#)（第 3 章第 10.8 节），这样

JMXConfigurator 将自动使用你设置的名字。

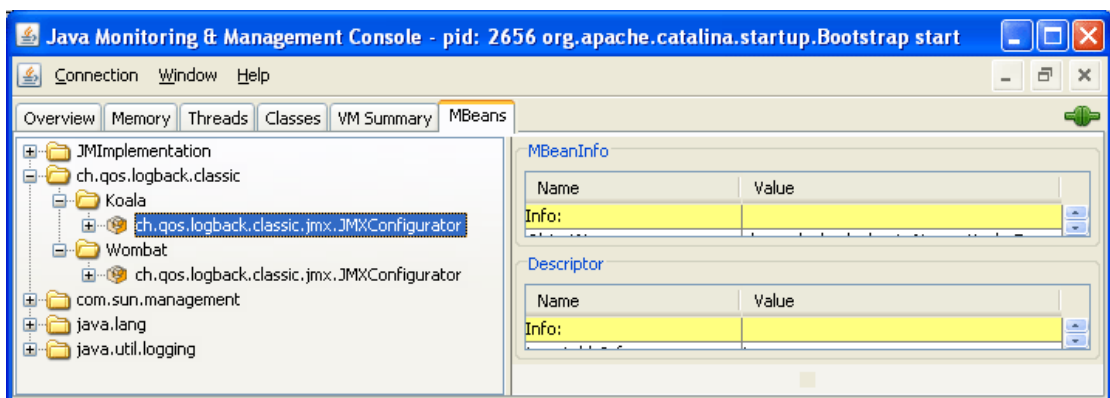
例如，如果你部署了两个 web 应用程序“Koala”和“Wombat”，那么你可以在 Koala 的配置文件里这样写：

```
<configuration>
  <contextName>Koala</contextName>
  <jmxConfigurator />
  ...
</configuration>
```

同时在 Wombat 配置文件里这样写：

```
<configuration>
  <contextName>Wombat</contextName>
  <jmxConfigurator />
  ...
</configuration>
```

在 jconsole 的 MBeans 面板里，可以看到两个不同的 JMXConfigurator 实例：



可以用<jmxConfigurator>元素的“objectName”属性控制 JMXConfigurator 在 MBeans 服务器里显示的名称。

## 10.4. 支持 JMX

如果你的服务器运行于 JDK 1.6 或更高版本，则默认支持 JMX。

对于低版本 JVM，我们建议你查阅你的 web 服务器的有关文档。Tomcat 和 Jetty 都有相关的文档。在本文中，我们简单地描述 Tomcat 和 Jetty 下的必要配置步骤。



### 10.4.1. Jetty 启用 JMX（在 JDK1.5 和 JDK1.6 上通过测试）

以下内容已经在 JDK 1.5 和 JDK1.6 上通过测试。在 JDK 1.6 或更高版本上，已经默认启用 JMX，你可以——但没必要——遵循下面的步骤。

在 JDK 1.5 上，为 Jetty 增加 JMX 支持需要在配置文件 “\$JETTY\_HOME/etc/jetty.xml” 里添加如下内容：

```
<Call id="MBeanServer" class="java.lang.management.ManagementFactory"
      name="getPlatformMBeanServer" />

<Get id="Container" name="container">
  <Call name="addEventListener">
    <Arg>
      <New class="org.mortbay.management.MBeanContainer">
        <Arg>
          <Ref id="MBeanServer" />
        </Arg>
        <Call name="start" />
      </New>
    </Arg>
  </Call>
</Get>
```

如果你想通过 jconsole 访问 Jetty 暴露出的 MBeans，就需要在设置 java 系统属性 “com.sun.management.jmxremote” 之后启动 Jetty。

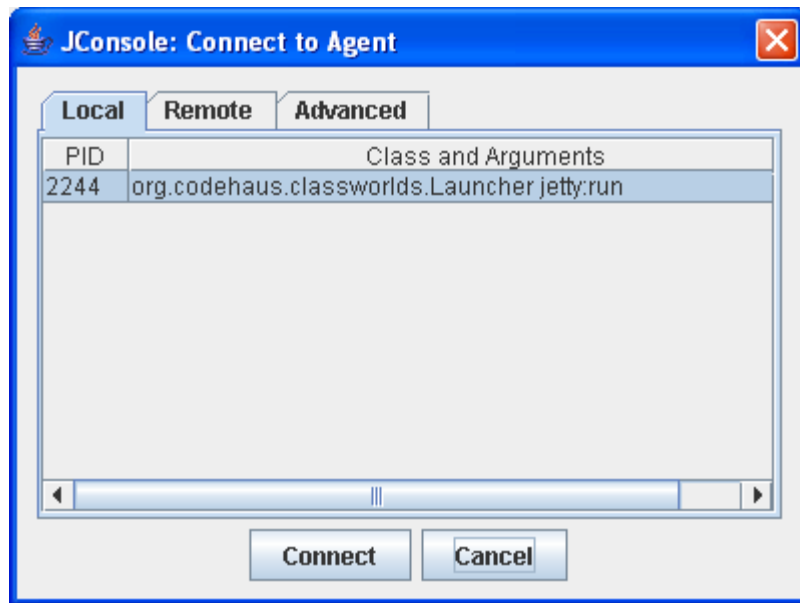
对于 Jetty 的独立运行版，启动命令是：

```
java -Dcom.sun.management.jmxremote -jar start.jar [config files]
```

如果你想把 Jetty 作为 Maven 插件启动，就需要通过 “MAVEN\_OPTS” 环境变量设置 “com.sun.management.jmxremote” 系统属性：

```
MAVEN_OPTS="-Dcom.sun.management.jmxremote"
mvn jetty:run
```

现在可以通过 jconsole 访问 Jetty 和 logback 暴露的 MBeans 了。



连接后，你可以访问如上面截图中所示的 JMXConfigurator。

### 10.4.2. Jetty 启用 MX4J（在 JDK 1.5 和 JDK 1.6 上通过测试）

如果你想通过 MX4J 的 HTTP 接口访问 JMXConfigurator，就需要在 Jetty 的配置文件“\$JETTY\_HOME/etc/jetty.xml”里添加如下内容：

```
<Call id="MBeanServer" class="java.lang.management.ManagementFactory"
  name="getPlatformMBeanServer" />

<Get id="Container" name="container">
  <Call name="addEventListener">
    <Arg>
      <New class="org.mortbay.management.MBeanContainer">
        <Arg>
          <Ref id="MBeanServer" />
        </Arg>
        <Set name="managementPort">8082</Set>
        <Call name="start" />
      </New>
    </Arg>
  </Call>
</Get>
```

需要把 mx4j-tools.jar 加入 Jetty 的 class path。

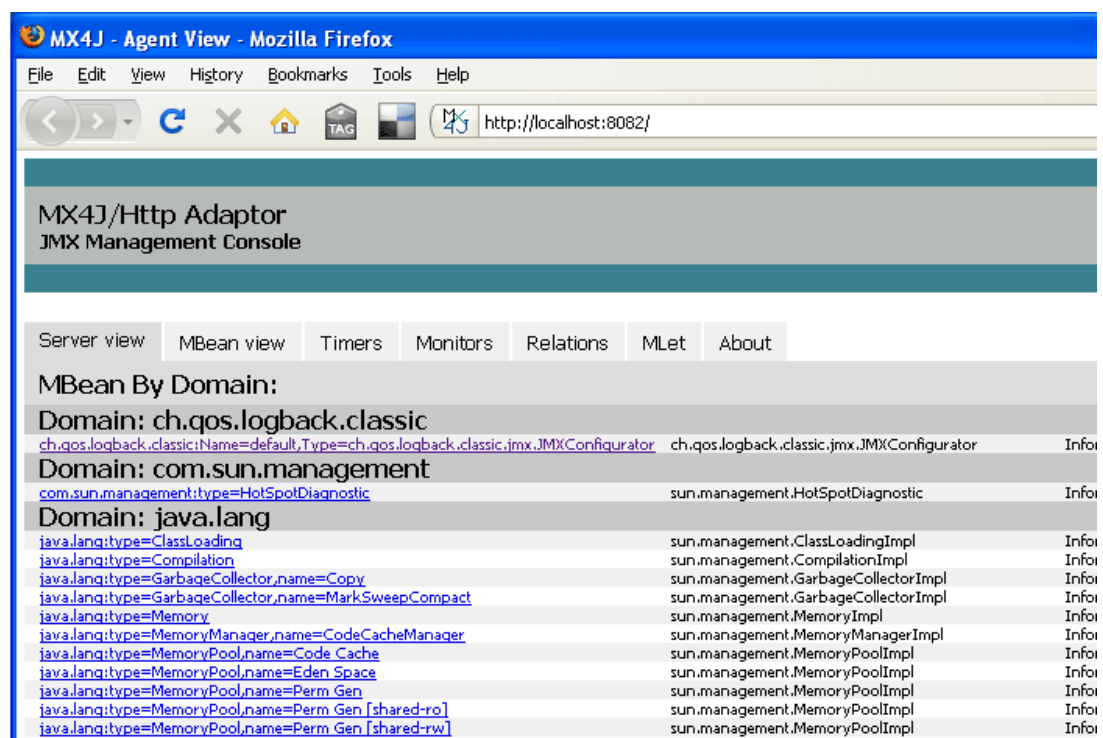
如果把 Jetty 作为 Maven 插件运行，则需要把 mx4j-tools.jar 作为一项依赖。

```
<plugin>
  <groupId>org.mortbay.jetty</groupId>
  <artifactId>maven-jetty-plugin</artifactId>
  <configuration>
    <jettyConfig>path/to/jetty.xml</jettyConfig>
    ...
  </configuration>
  <dependencies>
    <dependency>
      <groupId>mx4j</groupId>
      <artifactId>mx4j-tools</artifactId>
      <version>3.0.1</version>
    </dependency>
  </dependencies>
</plugin>
```

当 Jetty 按照上面的配置启动后，可以在下面的 URL 访问 JMXConfigurator（搜索“ch.qos.logback.classic”）：

<http://localhost:8082/>

下面是 MX4J 接口的截图：

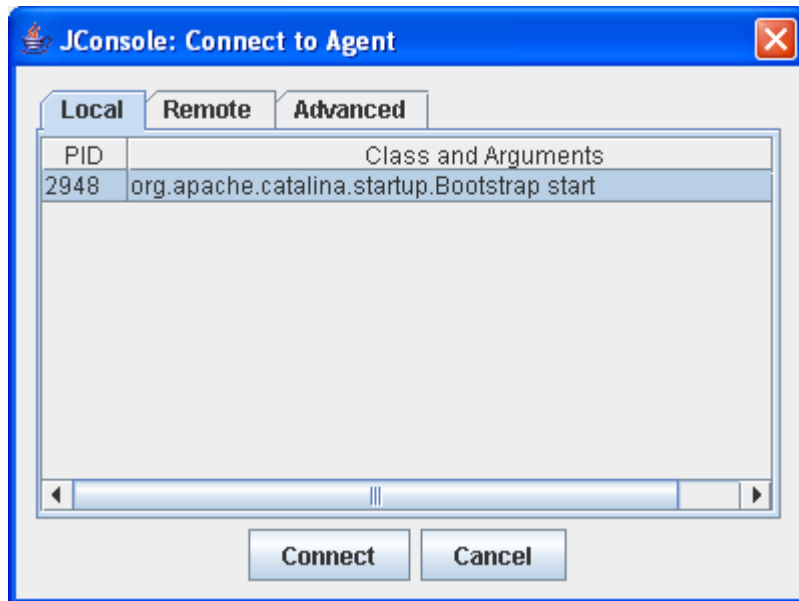


### 10.4.3. Tomcat 启用 JMX（在 JDK1.5 和 JDK1.6 上通过测试）

在 JDK 1.6 或更高版本上，已经默认启用 JMX，你可以——但没必要——遵循下面的步骤。在 JDK 1.5 上，为 Tomcat 增加 JMX 支持需要在脚本文件“\$TOMCAT\_HOME/bin/catalina.bat(或.sh)”里添加如下内容：

```
CATALINA_OPTS="-Dcom.sun.management.jmxremote"
```

用上面的选项启动后，就可以通过 jconsole 访问 Jetty 和 logback 暴露的 MBeans 了。



连接后，你可以访问如上面截图中所示的 JMXConfigurator。

### 10.4.4. Tomcat 启用 MX4J（在 JDK1.5 和 JDK1.6 上通过测试）

如果你想通过 MX4J 的 HTTP 接口访问 JMXConfigurator，则需要按下面的步骤：

把 “mx4j-tools.jar” 放到 “\$TOMCAT\_HOME/bin/” 目录，再在“\$TOMCAT\_HOME/bin/catalina.bat(或.sh)”里添加如下内容：

```
<!-- at the beginning of the file -->
CATALINA_OPTS="-Dcom.sun.management.jmxremote"

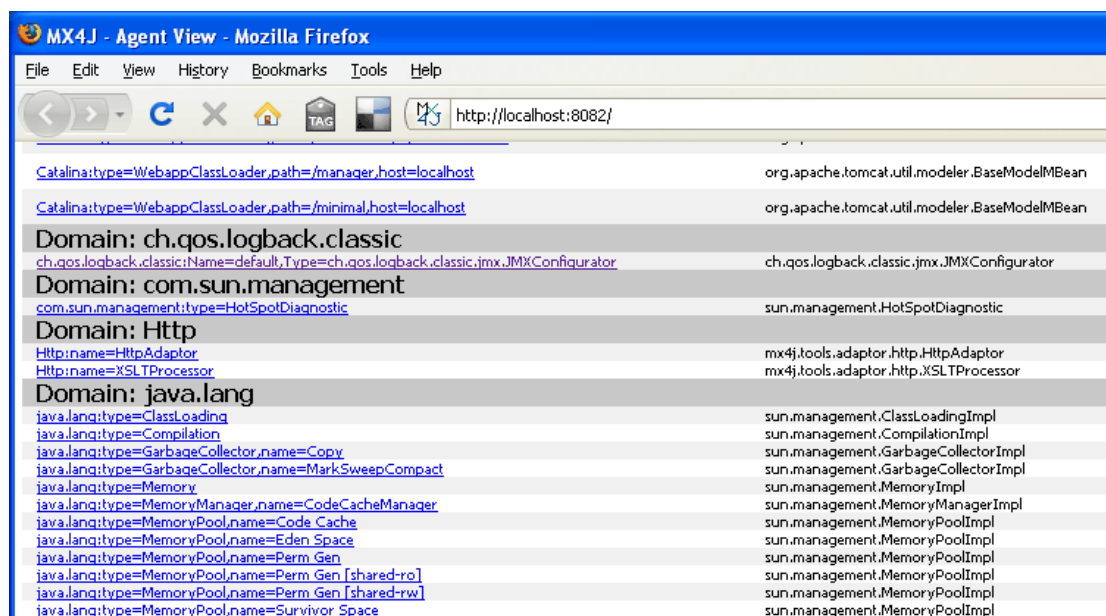
<!-- in the "Add on extra jar files to CLASSPATH" section -->
CLASSPATH="$CLASSPATH": "$CATALINA_HOME"/bin/mx4j-tools.jar
```

最后，在“\$TOMCAT\_HOME/conf/server.xml”文件里声明一个新 Connector：

```
<Connector port="0" handler.list="mx" mx.enabled="true"
  mx.httpHost="localhost" mx.httpPort="8082" protocol="AJP/1.3" />
```

按照上面的配置启动后，可以在下面的 URL 访问 JMXConfigurator（搜索“ch.qos.logback.classic”）：

下面是 MX4J 接口的截图：



## 11.Joran

Logback 依赖 Joran。Joran 是个成熟、灵活和强大的配置框架。Logback 模块的许多能力都只能在 Joran 帮助下实现。本章专注于 Joran 的基本设计和突出特性。

Joran 实际上是个通用的配置系统，可以在记录系统之外使用。强调一下，logback-core 模块完全与 logger 无关，秉承这一点，本章中的大多数例子都与 logger、appender 或 layout 无关。

本章的例子在 LOGBACK\_HOME/logback-examples/src/main/java/chapters/onJoran/ 目录。

安装 joran 的方法是把 logback-core.jar 加到 class path。

### 11.1. 历史回顾

反射 (reflection) 是 Java 语言的强大特性，使软件系统的声明式配置成为可能。例如，EJB 的很多重要属性是在 ejb.xml 里配置的。虽然 EJB 是用 Java 编写的，但很多属性是在 ejb.xml 文件里指定的。同样，logback 的设置可以在配置文件里指定，用 XML 格式。注解 (annotation) 在 JDK 1.5 中引入，大量应用于 EJB 3.0，用于取代之之前放在 XML 文件里的指令。Joran 也利用了注解，但使用范围比较小。由于 logback 的配置数据天生是动态 (与 EJB 相比) 的，所以 Joran 对注解的使用非常有限。

在 log4j——logback 的祖先——里，DOMConfigurator 类出现在 log4j 1.2.x 版及之后的版本中，也能够解析 XML 格式的配置文件。每当配置文件有变动时，DOMConfigurator 强迫我们这些开发者去调整代码，修改后的代码必须再次编译和部署。同样重要的是，DOMConfigurator 的代码包含对子元素的循环处理，里面到处都有 if/else 语句。无法回避那些代码的冗余和重复。commons-digester 项目 (<http://jakarta.apache.org/commons/digester/>) 向我们展示了用模式匹配规则处理 XML 文件的可行性。在解析期，digester 会对匹配了指定模式的内容应用规则 (rule)。Rule 类通常很小很专业。这样的结果是 digester 相当容易理解和维护。

有了开发 DOMConfigurator 的经验，我们开始开发 Joran，一个在 logback 中使用的强大的配置框架。Joran 深受 commons-digester 项目的启发。Joran 中的术语与 digester 略有不同。在 commons-digester 里，rule 可被视为一个 pattern 和一个 rule 的组合，如 Digester.addRule(String pattern, Rule rule) 方法所示。我们认为没有必要让 rule 不递归地而是另有它意地组成它自己。在 Joran 里，rule 由一个 pattern 和一个 action 组成。当模式产生匹配时，就调用 action。Pattern 与 action 的关系是 Joran 的核心。毫不夸张地说，用简单的模式就能处理相当复杂的需求，用精确匹配和通配符匹配可以进行精确地处理。

## 11.2. SAX 还是 DOM?

由于 SAX API 的基于事件的架构，基于 SAX 的工具不能轻易地处理向前的引用，即引用在当前元素之后定义的元素。有循环引用的元素也是个问题。更一般地说，DOM API 允许用户在所有元素里进行搜索和往前跳。

这种额外的灵活性最初让我们选择了 DOM API 作为 Joran 的底层解析 API。经过一些试验，很快表明，当解释规则是用 `pattern` 和 `action` 表达时，在解析 DOM 树的过程中跳到远处的元素没有意义。Joran 得到的元素只需要按照 XML 文件里连续的、深度优先的顺序。

Joran 先是用 DOM 实现，然而，作者换成了 SAX，以便得到只有 SAX API 才提供的元素位置信息。位置信息让 Joran 能在错误发生时显示准确的行号和列号，这对分析问题非常有用。

由于其高度动态的本性，Joran API 没有打算用于分析包含数以千计的元素巨大的 XML 文档。

## 11.3. 模式 (Pattern)

Joran 的模式本质上是字符串。有两种模式：精确模式和通配符模式。

模式 “a/b” 可以匹配嵌套在顶层元素 `<a>` 里的 `<b>` 元素，不会匹配其他模式。这就是精确模式。

通配符模式用来匹配前缀和后缀。例如，“\*/a” 模式匹配任何后缀是 “a” 的元素，也就是 XML 文档里的任意 `<a>` 元素，而不是嵌套在 `<a>` 里的元素。“a/\*” 模式匹配任何前缀是 `<a>` 的元素，也就是嵌套在 `<a>` 里的任何元素。

## 11.4. 动作 (Action)

如上所述，Joran 的解析规则包含与模式的关系。动作继承 `Action` 类，包含下面两个抽象方法，略去了其他方法。

```
package ch.qos.logback.core.joran.action;

import org.xml.sax.Attributes;
import ch.qos.logback.core.joran.spi.ExecutionContext;
```

```
public abstract class Action {  
    /**  
     * Called when the parser encounters an element matching a  
     * {@link ch.qos.logback.core.joran.spi.Pattern Pattern}.  
     */  
    public abstract void begin(InterpretationContext ic, String name,  
        Attributes attributes) throws ActionException;  
  
    /**  
     * Called to pass the body (as text) contained within an element.  
     */  
    public void body(InterpretationContext ic, String body)  
        throws ActionException {  
        // NOP  
    }  
  
    /**  
     * Called when the parser encounters an endElement event matching a  
     * {@link  
     *   ch.qos.logback.core.joran.spi.Pattern Pattern}.  
     */  
    public abstract void end(InterpretationContext ic, String name)  
        throws ActionException;  
}
```

因此，每个动作必须实现 `begin()` 方法和 `end()` 方法。是否实现 `body()` 方法是可选的，因为 `Action` 不提供任何操作。

## 11.5. RuleStore

之前提到过，按照匹配的模式而调用动作是 `Joran` 的一个核心概念。一个规则（`rule`）就是一个与模式的关联关系加一个动作。规则存放在 `RuleStore`。

上面说过，`Joran` 构建于 `SAX API` 之上。当解析 XML 文档时，每个元素的开始、体（`body`）和结束都产生对应的事件。当 `Joran` 配置器接收到这些事件时，就会在 `rule store` 里查找与当前模式所对应的动作。例如，对嵌套在顶级元素“`A`”里的 `B` 元素，它的开始事件、体事件和结束事件的当前模式是“`A/B`”。当前模式是一种数据结构，由 `Joran` 在接收和处理 `SAX` 事件时自动维护。

当一些规则匹配了当前模式时，精确匹配会覆盖后缀匹配，后缀匹配覆盖前缀匹配。详



情请参考 SimpleRuleStore 类（ch.qos.logback.core.joran.spi.SimpleRuleStore）。

## 11.6. 解释上下文（Interpretation context）

为了让多个动作合作，begin()和 end()方法的第一个参数是解释上下文。解释上下文包括一个对象堆栈、对象 map、错误列表和一个对调用了动作的 Joran 解析器的引用。详情请参考 InterpretationContext 类（ch.qos.logback.core.joran.spi. InterpretationContext）。

## 11.7. Hello world

本章的第一个例子演示了使用 Joran 所需要的最基本的代码。每当 HelloWorldAction(chapters.onJoran.helloWorld. HelloWorldAction)的 begin()方法被调用时就向控制台打印“Hello world”。XML 文件的解析工作由一个配置器完成。针对本章的目的，我们开发了一个非常简单的配置器“SimpleConfigurator”(chapters.onJoran. SimpleConfigurator)。HelloWorld 程序（chapters.onJoran.helloWorld. HelloWorld）把各部分组合到了一起：

- 创建包含规则的 map 和一个 Context；
- 把模式“hello-world”与对应的 HelloWorldAction 实例相关联，从而创建一个解析规则；
- 创建一个 SimpleConfigurator，参数是前面的 rule map；
- 调用配置器的 doConfigure 方法，把指定的 XML 文件作为参数；
- 最后一步，打印 context 里的状态消息。

hello.xml 文件包含一个<hello-world>元素，没有嵌套任何其他元素。请在 logback-examples/src/main/java/chapters/onJoran/helloWorld/目录下查阅。

以 hello.xml 做配置文件，运行 HelloWorld 程序后，会在控制台打印“Hello World”。

```
java chapters.onJoran.helloWorld.HelloWorld
src/main/java/chapters/onJoran/helloWorld/hello.xml
```

强烈鼓励你折腾这个例子，比如往 rule store 加新规则、修改 XML 文件（hello.xml）、添加新动作。

## 11.8. 合作动作

logback-examples/src/main/java/chapters/onJoran/calculator/目录下包含一些 Action，它们通过普通的对象栈进行共同合作来完成简单的计算。

文件 calculator1.xml 包扩一个 computation 元素，其中嵌套了一个 literal 元素，内容如下：

示例：第一个计算器例子

(logback-examples/src/main/java/chapters/onJoran/calculator/calculator1.xml)

```
<computation name="total">
  <literal value="3" />
</computation>
```

在 Calculator1 程序里，我们声明了各种解析规则（模式和动作），这些规则共同合作，根据 XML 文档的内容计算出结果。

运行参数是 calculator1.xml 的 Calculator1 程序：

```
java chapters.onJoran.calculator.Calculator1
src/main/java/chapters/onJoran/calculator/calculator1.xml
```

输出：

```
The computation named [total] resulted in the value 3
```

对 calculator1.xml 的解析涉及下面的步骤：

- 对应于<computation>元素的开始事件翻译为当前模式“/computation”。由于在 Calculator1 程序里，模式“/computation”关联于“ComputationAction1”实例，所以调用 ComputationAction1 实例的 begin()方法。
- 对应于<literal>元素的开始事件翻译为当前模式“/computation/literal”。由于模式“/computation/literal”关联于“LiteralAction”实例，所以调用 LiteralAction 实例的 begin()方法。
- 对应于<literal>元素的结束事件会触发调用同一个 LiteralAction 实例的 end()方法。
- 对应于<computation>元素的结束事件会触发调用同一个 ComputationAction1 实例的 end()方法。

有趣的是动作如何进行合作的。LiteralAction 读取 literal 的值，然后放入由

InterpretationContext 维护的对象栈。完成后,任何其他动作都可以 pop 出该值进行读写操作。本例中, ComputationAction1 类的 end()方法从栈中 pop 出 literal 的值然后打印出来。

下个例子 calculator2.xml 稍微复杂一点,同时也更有趣。

示例: Calculator 配置文件

(logback-examples/src/main/java/chapters/onJoran/calculator/calculator2.xml)

```
<computation name="toto">
  <literal value="7" />
  <literal value="3" />
  <add />
  <literal value="3" />
  <multiply />
</computation>
```

在 calculator1.xml 里,作为对<literal>元素的响应,LiteralAction 实例会在解释上下文(interpretation context)的对象栈的顶端 push 一个整数。而在本例中,即 calculator2.xml 中,有两个整数 7 和 3。作为对<add>元素的响应,AddAction 会 pop 刚才的两个整数,计算它们的和,然后把计算结果即 10 (= 7 + 3) push 到解释上下文的对象栈的顶端。下一个 literal 元素将把整数 3 push 到栈的顶端。作为对<multiply>元素的响应,MultiplyAction 会 pop 刚才的两个整数,即 10 和 3 计算它们的乘积,然后把计算结果即 30 push 到对象栈的顶端。最后,作为对<computation>的结束事件的响应,ComputationAction1 会打印对象栈顶端的值。因此,运行:

```
java                                     chapters.onJoran.calculator.Calculator1
src/main/java/chapters/onJoran/calculator/calculator2.xml
```

会输出:

```
The computation named [toto] resulted in the value 30
```

## 11.9. 隐式动作 (Implicit actions)

到目前为止,我们定义的规则都称为显式动作,因为对当前元素的模式/动作关联都能在 rule store 找到。然而,在高度可扩展的系统里,组件的数量和类型都非常多,因此为每个模式都关联显式动作会非常冗长乏味。

同时,即使是在高度可扩展的系统里,仍然可以观察到重复出现的规则。假设我们可以

识别这些重复规则，我们就可以处理在 logback 编译期内还是未知的包含子组件的组件。例如，Apache Ant 能够处理在编译期内包含未知标记的任务，方法是检查方法名以“add”开头的组件，比如 addFile 或 addClassPath。当 Ant 遇到嵌入在 task 里的标记时，就简单地实例化与 task 类的 add 方法签名相匹配的对象，然后把该对象附加给其父对象。

Joran 以隐式对象的方式的支持类似功能。Joran 维持一个隐式动作列表，如果没有显式模式能匹配当前模式时，就应用隐式动作。然而，应用隐式动作并不总是适当的。在执行隐式动作前，Joran 询问给定的隐式动作是否适用于当前情况。只有当动作回答“是”时，Joran 配置器才会调用隐式动作。注意，这个额外的步骤让对给定的情况使用多个或零个动作可能。

你可以创建并注册一个自定义隐式动作，如下例所示，位于 logback-examples/src/main/java/chapters/onJoran/implicit 目录：

PrintMe 程序为模式 “\*/foo” 关联了一个 NOPAction 动作实例，即所有名为“foo”的元素。如其名字所示，NOPAction 的 begin 和 end 方法都为空。PrintMe 程序还在其隐式动作列表里注册了一个 PrintMeImplicitAction 动作。PrintMeImplicitAction 动作适用于所有属性“printme”为“true”的元素。请参阅 PrintMeImplicitAction 的 isApplicable() 方法。PrintMeImplicitAction 的 begin() 方法把当前元素的名称打印到控制台。

XML 文档 implicit1.xml 用于演示隐式动作是如何工作的。

示例：隐式动作用法

(logback-examples/src/main/java/chapters/onJoran/implicit/implicit1.xml)

```
<foo>
  <xyz printme="true">
    <abc printme="true" />
  </xyz>

  <xyz />

  <foo printme="true" />
</foo>
```

运行：

```
java chapters.onJoran.implicit.PrintMe src/main/java/chapters/onJoran/implicit/implicit1.xml
```

输出：

```
Element [xyz] asked to be printed.
Element [abc] asked to be printed.
```

```
20:33:43,750 |-ERROR in c.q.l.c.joran.spi.Interpreter@10:9 - no applicable action for [xyz],
current pattern is [[foo][xyz]]
```

因为 NOPAction 实例显式地关联 “\*/foo” 模式，所以 NOPAction 在<foo>元素上调用 begin() 和 end() 方法。对任何<foo>元素都不会调用 PrintMeImplicitAction。对于其他元素，由于没有匹配任何隐式动作，所以调用 PrintMeImplicitAction 的 isApplicable() 方法。只有对于那些包含属性 “printme” 且其属性值为 “true” 的元素，即第一个<xyz>元素——但不是第二个<xyz>元素——和<abc>元素，isApplicable() 方法才返回 “true”。第 10 行上的第二个<xyz>元素没有可用的动作，所以会生成一个内部错误消息。这个错误消息会被 PrintMe 程序的最后一条语句 “StatusPrinter.print” 打印出来。这就解释了上面的输出结果。

## 11.10. 实践中的隐式动作

Logback-classic 和 logback-access 各自的 Joran 配置器只包含两个隐式动作：NestedBasicPropertyIA 和 NestedComplexPropertyIA。

NestedBasicPropertyIA 适用于属性类型是基本类型或基本类型在 java.lang 包里的等价对象类型、枚举类型或任何符合 “valueOf” 约定的类型。上述属性称为基本或简单属性。一个类如果包含静态方法 valueOf，方法参数是 java.lang.String 类型且返回自身类型的实例，这样的类就是符合 “valueOf” 约定的。目前，Level 类 (ch.qos.logback.classic.Level)、Duration(ch.qos.logback.core.util.Duration) 类和 FileSize(ch.qos.logback.core.util.FileSize) 类符合此约定。

NestedComplexPropertyIA 适用场合是：NestedBasicPropertyIA 不适用，并且，对象栈顶端的对象含有与当前元素名相等的属性，该属性有 setter 或 adder 方法。注意此类属性可以依次包含其他组件。因此这类属性称为复合属性。当出现复合属性时，NestedComplexPropertyIA 将为嵌套的组件实例化对应的类，并通过父组件（位于对象栈顶端）的 setter/adder 方法和嵌套元素的名称把实例化的类附加到父组件。对应的类通过当前（嵌套的）元素的 “class” 属性指定。如果没有 “class” 属性，可以根据下列步骤推断类名：

1. 有内部规则将指定类与父对象的属性进行关联；
2. setter 方法的 @DefaultClass 属性指定了一个类；
3. setter 方法的参数类型是一个有 public 构造方法的具体类。

### 11.10.1. 默认类映射

在 `logback-classic` 里，有一些内部规则把父类名、父属性名映射到默认类。默认类如下：

父类	属性名	默认嵌套类
<code>ch.qos.logback.core.AppenderBase</code>	<code>encoder</code>	<code>ch.qos.logback.classic.encoder.PatternLayoutEncoder</code>
<code>ch.qos.logback.core.UnsynchronizedAppenderBase</code>	<code>encoder</code>	<code>ch.qos.logback.classic.encoder.PatternLayoutEncoder</code>
<code>ch.qos.logback.core.AppenderBase</code>	<code>layout</code>	<code>ch.qos.logback.classic.PatternLayout</code>
<code>ch.qos.logback.core.UnsynchronizedAppenderBase</code>	<code>layout</code>	<code>ch.qos.logback.classic.PatternLayout</code>
<code>ch.qos.logback.core.filter.EvaluatorFilter</code>	<code>evaluator</code>	<code>ch.qos.logback.classic.boolex.JaninoEventEvaluator</code>

该列表在将来可能会改变。最新规则请参阅 `logback-classic` 的 `JoranConfigurator` 类 (`ch.qos.logback.classic.joran.JoranConfigurator`) 的 `addDefaultNestedComponentRegistryRules()` 方法。

### 11.10.2. 属性集合

Logback 的隐式动作不但支持单个简单属性或单个复合属性，还支持属性集合，集合里的属性可以是简单的也可以是复合的。复合属性用 “`adder`” 方法指定，而不是 “`setter`” 方法。

### 11.10.3. 动态新规则

Joran 的 `NewRuleAction` 动作支持在解析 XML 文档的过程中让 Joran 配置器学习新规则。示例代码见 `logback-examples/src/main/java/chapters/onJoran/newRule/` 目录，其中的 `NewRuleCalculator` 程序使用了两条规则，一条规则处理最顶端的元素，第二条规则学习新规则。下面是 `NewRuleCalculator` 的相关代码：

```
ruleMap.put(new Pattern("*/computation"), new ComputationAction1());
ruleMap.put(new Pattern("/computation/new-rule"), new
NewRuleAction());
```

`NewRuleAction` 是 `logback-core` 的一部分，工作原理与其他动作基本相同，有 `begin()` 和 `end()` 方法，每当解析器找到 “`new-rule`” 元素时被调用。当被调用时，`begin()` 方法查找 “`pattern`” 和 “`actionClass`” 属性，然后实例化对应的动作类，把模式/动作的关联关系作为一条新规则

加入到 Joran 的 rule store。

下面是如何在 xml 文件里声明新规则：

```
<new-rule pattern="*/computation/literal"
actionClass="chapters.onJoran.calculator.LiteralAction" />
```

使用这种 “new-rule” 声明后，我们可以改造 NewRuleCalculator，让它和 Calculator1 表现一致。

示例：使用动态新规则的配置文件

(logback-examples/src/main/java/chapters/onJoran/newrule/new-rule.xml)

```
<computation name="toto">
  <new-rule pattern="*/computation/literal"
actionClass="chapters.onJoran.calculator.LiteralAction" />
  <new-rule pattern="*/computation/add"
actionClass="chapters.onJoran.calculator.AddAction" />
  <new-rule pattern="*/computation/multiply"
actionClass="chapters.onJoran.calculator.MultiplyAction" />

  <computation>
    <literal value="7" />
    <literal value="3" />
    <add />
  </computation>

  <literal value="3" />
  <multiply />
</computation>
```

运行：

```
java          java          chapters.onJoran.newRule.NewRuleCalculator
src/main/java/chapters/onJoran/newRule/new-rule.xml
```

输出：

```
The computation named [toto] resulted in the value 30
```

该输出与 “合作动作” 里的 calculator2.xml 相同。

## 12. 从 log4j 迁移

本章讲述如何把自定义的 log4j 组件如 appender 或 layout 迁移到 logback-classic。

如果软件只调用了 log4j 的客户端 API，即 org.apache.log4j.Logger 包里的 Logger 或 Category 类，就能用 SLF4J 迁移工具 (<http://www.slf4j.org/migrator.html>) 自动迁移到 SLF4J。配置文件 log4j.properties 可以用 <http://logback.qos.ch/translator/> 转换到等价的 logback 配置文件。

从大方面来看，log4j 和 Logback-classic 联系紧密。核心组件，比如 logger、appender 和 layout，两者都有且功能相同。类似地，最重要的内部数据结构——LoggingEvent，两者都有也类似，但具体实现不同。最明显的区别是 logback-classic 里的 LoggingEvent 实现了 ILoggingEvent 接口。把 log4j 组件迁移到 logback-classic 的大多数工作都与 LoggingEvent 在两者内的不同实现有关，其余工作不多。

### 12.1. 迁移 log4j 的 layout

假设有 TrivialLog4jLayout，把记录事件的消息作为格式化后的消息返回。代码如下：

```
package chapters.migrationFromLog4j;

import org.apache.log4j.Layout;
import org.apache.log4j.spi.LoggingEvent;

public class TrivialLog4jLayout extends Layout {

    public void activateOptions() {
        // there are no options to activate
    }

    public String format(LoggingEvent loggingEvent) {
        return loggingEvent.getRenderedMessage();
    }

    public boolean ignoresThrowable() {
        return true;
    }
}
```



等价的 logback-classic 代码如下：

```
package chapters.migrationFromLog4j;

import ch.qos.logback.classic.spi.ILoggingEvent;
import ch.qos.logback.core.LayoutBase;

public class TrivialLogbackLayout extends LayoutBase<ILoggingEvent> {

    public String doLayout(ILoggingEvent loggingEvent) {
        return loggingEvent.getMessage();
    }
}
```

可见，在 logback-classic 的 layout 里，负责格式化的方法是 doLayout()，而在 log4j 里是 format()。Log4j 里的 ignoresThrowable() 方法在 logback-classic 里不再存在。注意 logback-classic 的 layout 必须继承 LayoutBase<ILoggingEvent> 类。

activateOptions() 方法值得多讲几句。在 log4j 里，一个 layout 会让 log4j 配置器调用其 activateOptions() 方法，PropertyConfigurator 或 DOMConfigurator 会在 layout 的所有选项都设置好之后，紧接着就调用 activateOptions()。因此，layout 就有机会对选项进行检查，然后再初始化自身。

在 logback-classic 里，layout 必须实现 LifeCycle 接口，该接口包含 start() 方法，该方法等价于 log4j 的 activateOptions() 方法。

## 12.2. 迁移 log4j 的 appender

迁移 appender 与迁移 layout 相似。以一个简单的 TrivialLog4jLayout 为例，它把自己返回的字符串写到控制台。

```
package chapters.migrationFromLog4j;

import org.apache.log4j.AppenderSkeleton;
import org.apache.log4j.spi.LoggingEvent;

public class TrivialLog4jAppender extends AppenderSkeleton {

    protected void append(LoggingEvent loggingevent) {
        String s = this.layout.format(loggingevent);
        System.out.println(s);
    }
}
```

```
public void close() {  
    // nothing to do  
}  
  
public boolean requiresLayout() {  
    return true;  
}  
}
```

等价的 logback-classic 的 TrivialLogbackAppender 如下:

```
package chapters.migrationFromLog4j;  
  
import ch.qos.logback.classic.spi.ILoggingEvent;  
import ch.qos.logback.core.AppenderBase;  
  
public class TrivialLogbackAppender extends AppenderBase<ILoggingEvent>  
{  
  
    @Override  
    public void start() {  
        if (this.layout == null) {  
            addError("No layout set for the appender named [" + name + "].");  
            return;  
        }  
        super.start();  
    }  
  
    @Override  
    protected void append(ILoggingEvent loggingevent) {  
        // note that AppenderBase.doAppend will invoke this method only  
        if  
            // this appender was successfully started.  
  
        String s = this.layout.doLayout(loggingevent);  
        System.out.println(s);  
    }  
}
```

对比上面两个类, 你应该注意 `append()` 方法里的内容没变。 `requiresLayout()` 在 logback 里已经被移除。 Logback 的 `stop()` 方法与 log4j 的 `close()` 方法等价。 然而, logback-classic 里的 `AppenderBase` 类包含对 `stop()` 方法的空实现, 对上面的这个演示用的 appender 已经够用了。