Chapter 3: Logback configuration

In symbols one observes an advantage in discovery which is greatest when they express the exact nature of a thing briefly and, as it were, picture it; then indeed the labor of thought is wonderfully diminished.

当他们简单地表达事物的确切性质时，这是最伟大的。

—GOTTFRIED WILHELM LEIBNIZ

Authors: Ceki Gülcü, Sébastien Pennec, Carl Harris

We start by presenting ways for configuring logback, with many example configuration scripts. Joran, the configuration framework upon which logback relies will be presented in a later chapter.

Configuration in logback

日志请求插入到应用程序代码需要大量的规划与努力。

Inserting log requests into the application code requires a fair amount of planning and effort. Observation shows that approximately four percent of code is dedicated to logging. Consequently, even a moderately sized application will contain thousands of logging statements embedded within its code. Given their number, we need tools to manage these log statements.

Logback can be configured either programmatically or with a configuration script expressed in XML or Groovy format. By the way, existing log4j users can convert their *log4j.properties* files to *logback.xml* using our PropertiesTranslator web-application.

Let us begin by discussing the initialization steps that logback follows to try to configure itself:

1. Logback tries to find a file called *logback.groovy* in the classpath.
2. If no such file is found, logback tries to find a file called *logback-test.xml* in the classpath.
3. If no such file is found, it checks for the file *logback.xml* in the classpath..
4. If no such file is found, and the executing JVM has the ServiceLoader (JDK 6 and above) the ServiceLoader will be used to resolve an implementation of `com.qos.logback.classic.spi.Configurator`. The first implementation found will be used. See ServiceLoader documentation for more details.
5. If none of the above succeeds, logback configures itself automatically using the `BasicConfigurator` which will cause logging output to be directed to the console.

The fourth and last step is meant to provide a default (but very basic) logging functionality in the absence of a configuration file.

If you are using Maven and if you place the *logback-test.xml* under the *src/test/resources* folder, Maven will ensure that it won't be included in the artifact produced. Thus, you can use a different configuration file, namely *logback-test.xml* during testing, and another file, namely, *logback.xml*, in production. The same principle applies by analogy for Ant.

Automatically configuring logback

自动地配置logback

The simplest way to configure logback is by letting logback fall back to its default configuration. Let us give a taste of how this is done in an imaginary application called `MyApp1`.

Example: Simple example of `BasicConfigurator` usage (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.");
  }
}
```

This class defines a static logger variable. It then instantiates a `Foo` object. The `Foo` class is listed below:

Example: Small class doing logging (logback-examples/src/main/java/chapters/configuration/Foo.java)

```
package chapters.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!");
  }
}
```

In order to run the examples in this chapter, you need to make sure that certain jar files are present on the class path. Please refer to the setup page for further details.

Assuming the configuration files *logback-test.xml* or *logback.xml* are not present, logback will default to invoking `BasicConfigurator` which will set up a minimal configuration. This minimal configuration consists of a `ConsoleAppender` attached to the root logger. The output is formatted using a `PatternLayoutEncoder` set to the pattern *%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n*. Moreover, by default the root logger is assigned the `DEBUG` level.

Thus, the output of the command *java chapters.configuration.MyApp1* should be similar to:

```
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.
```

The `MyApp1` application links to logback via calls to `org.slf4j.LoggerFactory` and `org.slf4j.Logger` classes, retrieve the loggers it wishes to use, and chugs on. Note that the only dependencies of the `Foo` class on logback are through `org.slf4j.LoggerFactory` and `org.slf4j.Logger` imports. Except code that configures logback (if such code exists) client code does not need to depend on logback. Since SLF4J permits the use of any logging framework under its abstraction layer, it is easy to migrate large bodies of code from one logging framework to another.

Except code that configures logback (if such code exists) client code does not need to depend on logback. Applications that use logback as their logging framework will have a compile-time dependency on SLF4J but not logback.

Automatic configuration with *logback-test.xml* or *logback.xml*

As mentioned earlier, logback will try to configure itself using the files *logback-test.xml* or *logback.xml* if found on the class path. Here is a configuration file equivalent to the one established by `BasicConfigurator` we've just seen.

Example: Basic configuration file (logback-examples/src/main/java/chapters/configuration/sample0.xml) 基本的配置文件

View as .groovy

```
<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>
```

After you have renamed *sample0.xml* as *logback.xml* (or *logback-test.xml*) place it into a directory accessible from the class path. Running the *MyApp1* application should give identical results to its previous run.

自动打印"警告或错误"的状态信息

Automatic printing of status messages in case of warning or errors

If warnings or errors occur during the parsing of the configuration file, logback will automatically print its internal status data on the console. Note that to avoid duplication, automatic status printing is disabled if the user explicitly registers a status listener (defined below).

If warning or errors occur during the parsing of the configuration file, logback will automatically print its internal status messages on the console.

In the absence of warnings or errors, if you still wish to inspect logback's internal status, then you can instruct logback to print status data by invoking the `print()` of the `StatusPrinter` class. The *MyApp2* application shown below is identical to *MyApp1* except for the addition of two lines of code for printing internal status data.

Example: Print logback's internal status information (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);
    ...
  }
```

If everything goes well, you should see the following output on the console

```
17:44:58,578 |-INFO in ch.qos.logback.classic.LoggerContext[default] - Found resource [log
17:44:58,671 |-INFO in ch.qos.logback.classic.joran.action.ConfigurationAction - debug att
17:44:58,671 |-INFO in ch.qos.logback.core.joran.action.AppenderAction - About to instanti
17:44:58,687 |-INFO in ch.qos.logback.core.joran.action.AppenderAction - Naming appender a
17:44:58,812 |-INFO in ch.qos.logback.core.joran.action.AppenderAction - Popping appender
17:44:58,812 |-INFO in ch.qos.logback.classic.joran.action.LevelAction - root level set to
17:44:58,812 |-INFO in ch.qos.logback.core.joran.action.AppenderRefAction - Attaching appe

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.
```

At the end of this output, you can recognize the lines that were printed in the previous example. You should also notice the logback's internal messages, a.k.a. `Status` objects, which allow convenient access to logback's internal state.

Instead of invoking `StatusPrinter` programmatically from your code, you can instruct the configuration file to dump status data, even in the absence of errors. To achieve this, you need to set the *debug* attribute of the *configuration* element, i.e. the top-most element in the configuration file, as shown below. Please note that this *debug* attribute relates only to the status data. It does *not* affect logback's configuration otherwise, in particular with respect to logger levels. (If you are asking, no, the root logger will *not* be set to `DEBUG`.)

Example: Basic configuration file using debug mode (logback-examples/src/main/java/chapters/configuration/sample1.xml)

View as .groovy

```
<configuration debug="true">

  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <!-- encoders are  by default assigned 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>
```

Setting the `debug` attribute within the <configuration> element will output status information, assuming that:

1. the configuration file is found
2. the configuration file is well-formed XML.

If any of these two conditions is not fulfilled, Joran cannot interpret the *debug* attribute since the configuration file cannot be read. If the configuration file is found but is malformed, then logback will detect the error condition and automatically print its internal status on the console. However, if the configuration file cannot be found, logback will not automatically print its status data, since this is not necessarily an error condition. Programmatically invoking `StatusPrinter.print()` as shown in the *MyApp2* application above ensures that status information is printed in every case.

FORCING STATUS OUTPUT In the absence of status messages, tracking down a rogue *logback.xml* configuration file can be difficult, especially in production where the application source cannot be easily modified. To help identify the location of a rogue configuration file, you can set a `StatusListener` via the "logback.statusListenerClass" system property (defined below) to force output of status messages. The "logback.statusListenerClass" system property can also be used to silence output automatically generated in case of errors.

Specifying the location of the default configuration file as a system property

You may specify the location of the default configuration file with a system property named `"logback.configurationFile"`. The value of this property can be a URL, a resource on the class path or a path to a file external to the application.

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

Note that the file extension must be ".xml" or ".groovy". Other extensions are ignored. Explicitly registering a status listener may help debugging issues locating the configuration file.

Automatically reloading configuration file upon modification

变更后自动重加载配置文件

If instructed to do so, logback-classic will scan for changes in its configuration file and automatically reconfigure itself when the configuration file changes. In order to instruct logback-classic to scan for changes in its configuration file and to automatically re-configure itself set the *scan* attribute of the `<configuration>` element to true, as shown next.

Logback-classic can scan for changes in its configuration file and automatically reconfigure itself when the configuration file changes.

Example: Scanning for changes in configuration file and automatic re-configuration (logback-examples/src/main/java/chapters/configuration/scan1.xml)

View as .groovy

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

By default, the configuration file will be scanned for changes once every minute. You can specify a different scanning period by setting the *scanPeriod* attribute of the `<configuration>` element. Values can be specified in units of milliseconds, seconds, minutes or hours. Here is an example:

Example: Specifying a different scanning period (logback-examples/src/main/java/chapters/configuration/scan2.xml)

View as .groovy

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

NOTE If no unit of time is specified, then the unit of time is assumed to be milliseconds, which is usually inappropriate. If you change the default scanning period, do not forget to specify a time unit.

Behind the scenes, when you set the scan attribute to true, a `TurboFilter` called ReconfigureOnChangeFilter will be installed. TurboFilters are described in a later chapter. As a consequence, scanning is done "in-thread", that is any time a printing method of logger is invoked. For example, for a logger named `myLogger`, when you write "myLogger.debug("hello");", and if the scan attribute is set to true, then `ReconfigureOnChangeFilter` will be invoked. Moreover, the said filter will be invoked even if `myLogger` is disabled for the DEBUG level.

Given that `ReconfigureOnChangeFilter` is invoked every time *any* logger is invoked, regardless of logger level, `ReconfigureOnChangeFilter` is absolutely performance critical. So much so that in fact, the check whether the scan period has elapsed or not, is too costly in itself. In order to improve performance, `ReconfigureOnChangeFilter` is in reality "alive" only once every *N* logging operations. Depending on how often your application logs, the value of *N* can be modified on the fly by logback. By default N is 16, although it can go as high as 2^16 (= 65536) for CPU-intensive applications.

When a configuration file changes, it will be automatically reloaded but only after several logger invocations and after a delay determined by the scanning period.

In short, when a configuration file changes, it will be automatically reloaded but only after several logger invocations *and* after a delay determined by the scanning period.

Invoking `JoranConfigurator` directly

Logback relies on a configuration library called Joran, part of logback-core. Logback's default configuration mechanism invokes `JoranConfigurator` on the default configuration file it finds on the class path. If you wish to override logback's default configuration mechanism for whatever reason, you can do so by invoking `JoranConfigurator` directly. The next application, *MyApp3*, invokes JoranConfigurator on a configuration file passed as a parameter.

Example: Invoking `JoranConfigurator` directly (logback-examples/src/main/java/chapters/configuration/MyApp3.java)

```
package chapters.configuration;
```

```
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 context = (LoggerContext) LoggerFactory.getILoggerFactory();

    try {
      JoranConfigurator configurator = new JoranConfigurator();
      configurator.setContext(context);
      // Call context.reset() to clear any previous configuration, e.g. default
      // configuration. For multi-step configuration, omit calling context.reset().
      context.reset();
      configurator.doConfigure(args[0]);
    } catch (JoranException je) {
      // StatusPrinter will handle this
    }
    StatusPrinter.printInCaseOfErrorsOrWarnings(context);

    logger.info("Entering application.");

    Foo foo = new Foo();
    foo.doIt();
    logger.info("Exiting application.");
  }
}
```

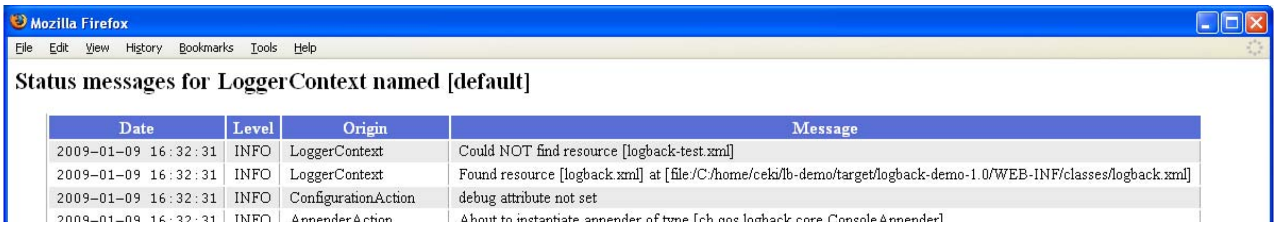
This application fetches the `LoggerContext` currently in effect, creates a new `JoranConfigurator`, sets the context on which it will operate, resets the logger context, and then finally asks the configurator to configure the context using the configuration file passed as a parameter to the application. Internal status data is printed in case of warnings or errors. Note that for multi-step configuration, `context.reset()` invocation should be omitted.

Viewing status messages

Logback collects its internal status data in a `StatusManager` object, accessible via the `LoggerContext`.

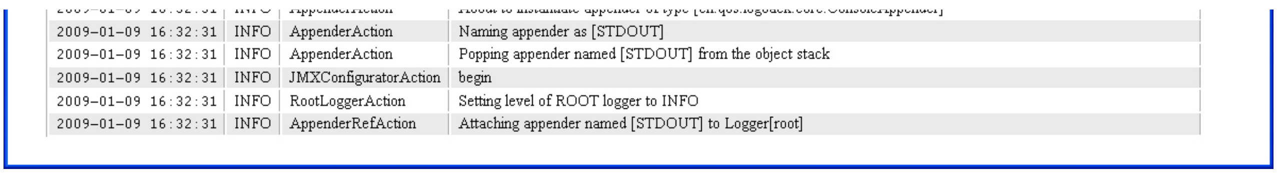
Given a `StatusManager` you can access all the status data associated with a logback context. To keep memory usage at reasonable levels, the default `StatusManager` implementation stores the status messages in two separate parts: the header part and the tail part. The header part stores the first *H* status messages whereas the tail part stores the last *T* messages. At present time *H*=*T*=150, although these values may change in future releases.

Logback-classic ships with a servlet called ViewStatusMessagesServlet. This servlet prints the contents of the `StatusManager` associated with the current `LoggerContext` as an HTML table. Here is sample output.



Status messages for LoggerContext named [default]

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]

To add this servlet to your web-application, add the following lines to its *WEB-INF/web.xml* file.

```
  <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>
```

The `ViewStatusMessages` servlet will be viewable at the URL `http://host/yourWebapp/lbClassicStatus`

Listening to status messages

You may also attach a `StatusListener` to a `StatusManager` so that you can take immediate action in response to status messages, especially to messages occurring after logback configuration. Registering a status listener is a convenient way to supervise logback's internal state without human intervention.

Logback ships with a `StatusListener` implementation called `OnConsoleStatusListener` which, as its name indicates, prints all *new* incoming status messages on the console.

Here is sample code to register an `OnConsoleStatusListener` instance with the StatusManager.

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

Note that the registered status listener will only receive status events subsequent to its registration. It will not receive prior messages. Thus, it is usually a good idea to place status listener registration directives at top of the configuration file before other directives.

It is also possible to register one or more status listeners within a configuration file. Here is an example.

Example: Registering a status listener (logback-examples/src/main/java/chapters/configuration/onConsoleStatusListener.xml)

View as .groovy

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

  ... the rest of the configuration file
</configuration>
```

"logback.statusListenerClass" system property

One may also register a status listener by setting the "logback.statusListenerClass" Java system property to the name of the listener class you wish to register. For example,

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

Logback ships with several status listener implementations. OnConsoleStatusListener prints incoming status messages on the console, i.e. on System.out. OnErrorConsoleStatusListener prints incoming status messages on System.err. NopStatusListener drops incoming status messages.

Note that automatic status printing (in case of errors) is disabled if any status listener is registered during configuration and in particular if the user specifies a status listener via the "logback.statusListenerClass" system. Thus, by setting `NopStatusListener` as a status listener, you can silence internal status printing altogether.

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

Stopping logback-classic

In order to release the resources used by logback-classic, it is always a good idea to stop the logback context. Stopping the context will close all appenders attached to loggers defined by the context and stop any active threads.

```
import org.sflf4j.LoggerFactory;
import ch.qos.logback.classic.LoggerContext;
...

// assume SLF4J is bound to logback-classic in the current environment
LoggerContext loggerContext = (LoggerContext) LoggerFactory.getILoggerFactory();
loggerContext.stop();
```

In web-applications the above code could be invoked from within the contextDestroyed method of `ServletContextListener` in order to stop logback-classic and release resources.

Stopping logback-classic via a shutdown hook 通过“关闭钩子”来停止logback

Installing a JVM shutdown hook is a convenient way for shutting down logback and releasing associated resources.

```
<configuration debug="true">
   <!-- in the absence of the class attribute, assume
   ch.qos.logback.core.hook.DelayingShutdownHook -->
   <shutdownHook/>
  ....
</configuration>
```

The default shutdown hook, namely DelayingShutdownHook, can delay shutdown by a user specified duration. Note that you may install a shutdown hook of your own making by setting the *class* attribute to correspond to your shutdown hook's class name.

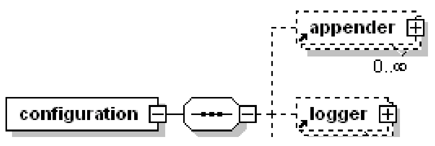
.

Configuration file syntax 配置文件语法

As you have seen thus far in the manual with plenty of examples still to follow, logback allows you to redefine logging behavior without needing to recompile your code. Indeed, you can easily configure logback so as to disable logging for certain parts of your application, or direct output to a UNIX Syslog daemon, to a database, to a log visualizer, or forward logging events to a remote logback server, which would log according to local server policy, for example by forwarding the log event to a second logback server.

The remainder of this section presents the syntax of configuration files.

As will be demonstrated over and over, the syntax of logback configuration files is extremely flexible. As such, it is not possible to specify the allowed syntax with a DTD file or an XML schema. Nevertheless, the very basic structure of the configuration file can be described as, `<configuration>` element, containing zero or more `<appender>` elements, followed by zero or more `<logger>` elements, followed by at most one `<root>` element. The following diagram illustrates this basic structure.



Case sensitivity of tag names

If you are unsure which case to use for a given tag name, just follow the camelCase convention which is almost always the correct convention.

Since logback version 0.9.17, tag names pertaining to explicit rules are case insensitive. For example, `<logger>`, `<Logger>` and `<LOGGER>` are valid configuration elements and will be interpreted in the same way. Note that XML well-formedness rules still apply, if you open a tag as `<xyz>` you must close it as `</xyz>`, `</XyZ>` will not work. As for implicit rules, tag names are case sensitive except for the first letter. Thus, `<xyz>` and `<Xyz>` are equivalent but not `<xYz>`. Implicit rules usually follow the camelCase convention, common in the Java world. Since it is not easy to tell when a tag is associated with an explicit action and when it is associated with an implicit action, it is not trivial to say whether an XML tag is case-sensitive or insensitive with respect to the first letter. If you are unsure which case to use for a given tag name, just follow the camelCase convention which is almost always the correct convention.

Configuring loggers, or the `<logger>` element 配置“日志记录器”

At this point you should have at least some understanding of level inheritance and the basic selection rule. Otherwise, and unless you are an Egyptologist, logback configuration will be no more meaningful to you than are hieroglyphics.

A logger is configured using the `<logger>` element. A `<logger>` element takes exactly one mandatory *`name`* attribute, an optional *`level`* attribute, and an optional *`additivity`* attribute, admitting the values *true* or *false*. The value of the *`level`* attribute admitting one of the case-insensitive string values TRACE, DEBUG, INFO, WARN, ERROR, ALL or OFF. The special case-insensitive value *INHERITED*, or its synonym *NULL*, will force the level of the logger to be inherited from higher up in the hierarchy. This comes in handy if you set the level of a logger and later decide that it should inherit its level.

Note that unlike log4j, logback-classic does *not* close nor remove any previously referenced appenders when configuring a given logger.

The `<logger>` element may contain zero or more `<appender-ref>` elements; each appender thus referenced is added to the named logger. Note that unlike log4j, logback-classic does *not* close nor remove any previously referenced appenders when configuring a given logger.

Configuring the root logger, or the `<root>` element

配置“根日志记录器”

The `<root>` element configures the root logger. It supports a single attribute, namely the *`level`* attribute. It does not allow any other attributes because the additivity flag does not apply to the root logger. Moreover, since the root logger is already named as "ROOT", it does not allow a name attribute either. The value of the level attribute can be one of the case-insensitive strings TRACE, DEBUG, INFO, WARN, ERROR, ALL or OFF. Note that the level of the root logger cannot be set to INHERITED or NULL.

Note that unlike log4j, logback-classic does *not* close nor remove any previously referenced appenders when configuring the root logger.

Similarly to the `<logger>` element, the `<root>` element may contain zero or more `<appender-ref>` elements; each appender thus referenced is added to the root logger. Note that unlike log4j, logback-classic does *not* close nor remove any previously referenced appenders when configuring the root logger.

Example 示例

Setting the level of a logger or root logger is as simple as declaring it and setting its level, as the next example illustrates. Suppose we are no longer interested in seeing any DEBUG messages from any component belonging to the "chapters.configuration" package. The following configuration file shows how to achieve that.

Example: Setting the level of a logger (logback-examples/src/main/java/chapters/configuration

/sample2.xml)

View as .groovy

```
<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>

  <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>
```

When the above configuration file is given as argument to the *MyApp3* application, it will yield the following output:

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

Note that the message of level DEBUG generated by the "chapters.configuration.Foo" logger has been suppressed. See also the Foo class.

You can configure the levels of as many loggers as you wish. In the next configuration file, we set the level of the *chapters.configuration* logger to INFO but at the same time set the level of the *chapters.configuration.Foo* logger to `DEBUG`.

Example: Setting the level of multiple loggers (logback-examples/src/main/java/chapters/configuration/sample3.xml)

View as .groovy

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

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

</configuration>
```

Running `MyApp3` with this configuration file will result in the following output on the console:

```
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.
```

The table below list the loggers and their levels, after `JoranConfigurator` has configured logback with the *sample3.xml* configuration file.

Logger name	Assigned Level	Effective Level
root	DEBUG	DEBUG
chapters.configuration	INFO	INFO
chapters.configuration.MyApp3	null	INFO
chapters.configuration.Foo	DEBUG	DEBUG

It follows that the two logging statements of level `INFO` in the `MyApp3` class as well as the DEBUG messages in `Foo.doIt()` are all enabled. Note that the level of the root logger is always set to a non-null value, DEBUG by default.

Let us note that the basic-selection rule depends on the effective level of the logger being invoked, not the level of the logger where appenders are attached. Logback will first determine whether a logging statement is enabled or not, and if enabled, it will invoke the appenders found in the logger hierarchy, regardless of their level. The configuration file *sample4.xml* is a case in point:

Example: Logger level sample (logback-examples/src/main/java/chapters/configuration/sample4.xml)

View as .groovy

```
<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" level="INFO" />

  <!-- turn OFF all logging (children can override) -->
  <root level="OFF">
    <appender-ref ref="STDOUT" />
  </root>

</configuration>
```

The following table lists the loggers and their levels after applying the *sample4.xml* configuration file.

Logger name	Assigned Level	Effective Level
root	OFF	OFF
chapters.configuration	INFO	INFO
chapters.configuration.MyApp3	null	INFO
chapters.configuration.Foo	null	INFO

The ConsoleAppender named *STDOUT*, the only configured appender in *sample4.xml*, is attached to the root logger whose level is set to `OFF`. However, running *MyApp3* with configuration script *sample4.xml* will yield:

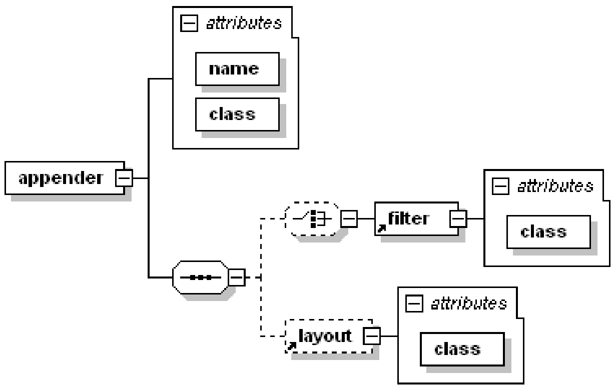
```
17:52:23.609 [main] INFO chapters.configuration.MyApp3 - Entering application.
```

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

Thus, the level of the root logger has no apparent effect because the loggers in `chapters.configuration.MyApp3` and `chapters.configuration.Foo` classes are all enabled for the `INFO` level. As a side note, the *chapters.configuration* logger exists by virtue of its declaration in the configuration file - even if the Java source code does not directly refer to it.

Configuring Appenders   配置"输出端"

An appender is configured with the `<appender>` element, which takes two mandatory attributes *`name`* and *`class`*. The *`name`* attribute specifies the name of the appender whereas the *`class`* attribute specifies the fully qualified name of the appender class to instantiate. The `<appender>` element may contain zero or one `<layout>` elements, zero or more `<encoder>` elements and zero or more `<filter>` elements. Apart from these three common elements, `<appender>` elements may contain any number of elements corresponding to JavaBean properties of the appender class. Seamlessly supporting any property of a given logback component is one of the major strengths of Joran as discussed in a later chapter. The following diagram illustrates the common structure. Note that support for properties is not visible.



The `<layout>` element takes a mandatory class attribute specifying the fully qualified name of the layout class to instantiate. As with the `<appender>` element, `<layout>` may contain other elements corresponding to properties of the layout instance. Since it's such a common case, if the layout class is `PatternLayout`, then the class attribute can be omitted as specified by default class mapping rules.

The `<encoder>` element takes a mandatory class attribute specifying the fully qualified name of the encoder class to instantiate. Since it's such a common case, if the encoder class is `PatternLayoutEncoder`, then the class attribute can be omitted as specified by default class mapping rules.

Logging to multiple appenders is as easy as defining the various appenders and referencing them in a logger, as the next configuration file illustrates:

Example: Multiple loggers (logback-examples/src/main/java/chapters/configuration/multiple.xml)

View as .groovy

```
<configuration>

  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <file>myApp.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>

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

These configuration scripts define two appenders called *FILE* and *STDOUT*. The *FILE* appender logs to a file called *myApp.log*. The encoder for this appender is a `PatternLayoutEncoder` that outputs the date, level, thread name, logger name, file name and line number where the log request is located, the message and line separator character(s). The second appender called `STDOUT` outputs to the console. The encoder for this appender outputs only the message string followed by a line separator.

The appenders are attached to the root logger by referencing them by name within an *appender-ref* element. Note that each appender has its own encoder. Encoders are usually not designed to be shared by multiple appenders. The same is true for layouts. As such, logback configuration files do not provide any syntactical means for sharing encoders or layouts.

Appenders accumulate

"输出端"累积

By default, **appenders are cumulative**: a logger will log to the appenders attached to itself (if any) as well as all the appenders attached to its ancestors. Thus, attaching the same appender to multiple loggers will cause logging output to be duplicated.

Example: Duplicate appender (logback-examples/src/main/java/chapters/configuration/duplicate.xml)

View as .groovy

```
<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>
```

Running `MyApp3` with *duplicate.xml* will yield the following output:

```
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.
```

Notice the duplicated output. The appender named *STDOUT* is attached to two loggers, to root and to *chapters.configuration*. Since the root logger is the ancestor of all loggers and *chapters.configuration* is the parent of both *chapters.configuration.MyApp3* and *chapters.configuration.Foo*, each logging request made with these two loggers will be output twice, once because *STDOUT* is attached to *chapters.configuration* and once because it is attached to *root*.

Appender additivity is not intended as a trap for new users. It is quite a convenient logback feature. For instance, you can configure logging such that log messages appear on the console (for all loggers in the system) while messages only from some specific set of loggers flow into a specific appender.

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

View as .groovy

```
<configuration>

  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <file>myApp.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">
    <appender-ref ref="FILE" />
  </logger>

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

In this example, the console appender will log all the messages (for all loggers in the system) whereas only logging requests originating from the *chapters.configuration* logger and its children will go into the *myApp.log* file.

Overriding the default cumulative behaviour

In case the default cumulative behavior turns out to be unsuitable for your needs, you can override it by setting the additivity flag to false. Thus, a branch in your logger tree may direct output to a set of appenders different from those of the rest of the tree.

Example: Additivity flag (logback-examples/src/main/java/chapters/configuration /additivityFlag.xml)

View as .groovy

```
<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>
```

This example, the appender named *FILE* is attached to the *chapters.configuration.Foo* logger. Moreover, the *chapters.configuration.Foo* logger has its additivity flag set to false such that its logging output will be sent to the appender named *FILE* but not to any appender attached higher in the hierarchy. Other loggers remain oblivious to the additivity setting of the *chapters.configuration.Foo* logger. Running the `MyApp3` application with the *additivityFlag.xml* configuration file will output results on the console from the *chapters.configuration.MyApp3* logger. However, output from the *chapters.configuration.Foo* logger will appear in the *foo.log* file and only in that file.

Setting the context name 设置"上下文名称"

As mentioned in an earlier chapter, every logger is attached to a logger context. By default, the logger context is called "default". However, you can set a different name with the help of the `<contextName>` configuration directive. Note that once set, the logger context name cannot be changed. Setting the context name is a simple and straightforward method in order to distinguish between multiple applications logging to the same target.

Example: Set the context name and display it (logback-examples/src/main/java/chapters/configuration/contextName.xml)

View as .groovy

```
<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>
```

This last example illustrates naming of the logger context. Adding the contextName conversion word in layout's pattern will output the said name.

Variable substitution 变量替换

NOTE Earlier versions of this document used the term "property substitution" instead of the term "variable". Please consider both terms interchangeable although the latter term conveys a clearer meaning.

As in many scripting languages, logback configuration files support definition and substitution of variables. Variables can be defined within the configuration file itself, in an external file, in an external resource or even computed and defined on the fly.

Variable substitution can occur at any point in a configuration file where a value can be specified. The syntax of variable substitution is similar to that of Unix shells. The string between an opening *${* and closing *}* is interpreted as a reference to the *value* of the property. For property *aName*, the string "${aName}" will be replaced with the value held by the *aName* property.

Variable substitution can occur at any point in a configuration file where a value can be specified.

Variables have a scope (see below).

As they often come in handy, the HOSTNAME and CONTEXT_NAME variables are automatically defined and have context scope.

Defining variables 定义变量

Variables can be defined one at a time in the configuration file itself or loaded wholesale from an external properties file or an external resource. For historical reasons, the XML element for defining variables is `<property>` although in logback 1.0.7 and later the element `<variable>` can be used interchangeably.

The next example shows a variable declared at the beginning of the configuration file. It is then used further down the file to specify the location of the output file.

Example: Simple Variable substitution (logback-examples/src/main/java/chapters/configuration/variableSubstitution1.xml)

View as .groovy

```
<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>
```

The next example shows the use of a System property to achieve the same result. The property is not declared in the configuration file, thus logback will look for it in the System properties. Java system properties can be set on the command line as shown next:

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

Example: System Variable substitution (logback-examples/src/main/java/chapters/configuration/variableSubstitution2.xml)

View as .groovy

```
<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>
```

When multiple variables are needed, it may be more convenient to create a separate file that will contain all the variables. Here is how one can do such a setup.

Example: Variable substitution using a separate file (logback-examples/src/main/java/chapters/configuration/variableSubstitution3.xml)

View as .groovy

```
<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>

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

This configuration file contains a reference to a file named *variables1.properties*. The variables contained in that file will be read and then defined within local scope. Here is what the *variable.properties* file might look like.

Example: Variable file (logback-examples/src/main/java/chapters/configuration/variables1.properties)

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

You may also reference a resource on the class path instead of a file.

```
<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>
```

Scopes

作用域：本地、上下文、系统

A property can be defined for insertion in *local scope*, in *context scope*, or in *system scope*. Local scope is the default. Although it is possible to read variables from the OS environment, it is not possible to write into the OS environment.

LOCAL SCOPE A property with local scope exists from the point of its definition in a configuration file until the end of interpretation/execution of said configuration file. As a corollary, each time a configuration file is parsed and executed, variables in local scope are defined anew.

CONTEXT SCOPE A property with context scope is inserted into the context and lasts as long as the context or until it is cleared. Once defined, a property in context scope is part of the context. As such, it is available in all logging events, including those sent to remote hosts via serialization.

SYSTEM SCOPE A property with system scope is inserted into the JVM's system properties and lasts as long as the JVM or until it is cleared.

During substitution, properties are looked up in the local scope first, in the context scope second, in the system properties scope third, and in the OS environment fourth and last.

Properties are looked up in the the local scope first, in the context scope second, in the system properties scope third, and in the OS environment last.

The *scope* attribute of the `<property>` element, `<define>` element or the `<insertFromJNDI>` element can be used to set the scope of a property. The *scope* attribute admits "local", "context" and "system" strings as possible values. If not specified, the scope is always assumed to be "local".

Example: A variable defined in "context" scope (logback-examples/src/main/java/chapters /configuration/contextScopedVariable.xml)

View as .groovy

```
<configuration>

  <property scope="context" name="nodeId" value="firstNode" />

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

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

In the above example, given that the *nodeId* property is defined in the context scope, it will be available in every logging event, even those sent to remote hosts via serialization.

Default values for variables

Under certain circumstances, it may be desirable for a variable to have a default value if it is not declared or its value is null. As in the Bash shell, default values can be specified using the **":-"** operator. For example, assuming the variable named *aName* is not defined, `"${aName:-golden}"` will be interpreted as "golden".

Nested variables

Variable nesting is fully supported. Both the name, default-value and value definition of a variable can reference other variables.

value nesting

The value definition of a variable can contain references to other variables. Suppose you wish to use variables to specify not only the destination directory but also the file name, and combine those two variables in a third variable called "destination". The properties file shown below gives an example.

Example: Nested variable references (logback-examples/src/main/java/chapters/configuration /variables2.properties)

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

Note that in the properties file above, "destination" is composed from two other variables, namely "USER_HOME" and "fileName".

Example: Variable substitution using a separate file (logback-examples/src/main/java/chapters /configuration/variableSubstitution4.xml)

```
<configuration>

  <property file="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>
```

name nesting

When referencing a variable, the variable name may contain a reference to another variable. For example, if the variable named "userid" is assigned the value "alice", then "${${userid}.password}" references the variable with the name "alice.password".

default value nesting

The default value of a variable can reference a another variable. For example, assuming the variable 'id' is unassigned and the variable 'userid' is assigned the value "alice", then the expression "${id**:-**${userid}}" will return "alice".

HOSTNAME property

As it often comes in handy, the `HOSTNAME` property is defined automatically during configuration with context scope.

CONTEXT_NAME property

As its name indicates, the `CONTEXT_NAME` property corresponds to the name of the current logging context.

Setting a timestamp

The *timestamp* element can define a property according to current date and time. The *timestamp* element is explained in a subsequent chapter.

Defining properties on the fly

You may define properties dynamically using the `<define>` element. The define element takes two mandatory attributes: *name* and *class*. The *name* attribute designates the name of the property to set whereas the *class* attribute designates any class implementing the PropertyDefiner interface. The value returned by the `getPropertyValue()` method of the `PropertyDefiner` instance will be the value of the named property. You may also specify a scope for the named property by specifying a *scope* attribute.

Here is an example.

```
<configuration>

  <define name="rootLevel" class="a.class.implementing.PropertyDefiner">
    <shape>round</shape>
    <color>brown</color>
    <size>24</size>
  </define>

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

In the above example, shape, color and size are properties of "a.class.implementing.PropertyDefiner". As long as there is a setter for a given property in your implementation of the `PropertyDefiner` instance, logback will inject the appropriate values as specified in the configuration file.

At the present time, logback does ships with two fairly simple implementations of `PropertyDefiner`.

Implementation name	Description
FileExistsPropertyDefiner	Set the named variable to "true" if the file specified by `path` property exists, to "false" otherwise.
ResourceExistsPropertyDefiner	Set the named variable to "true" if the `resource` specified by the user is available on the class path, to "false" otherwise.

Conditional processing of configuration files 配置文件的条件处理

Developers often need to juggle between several logback configuration files targeting different environments such as development, testing and production. These configuration files have substantial parts in common differing only in a few places. To avoid duplication, logback supports conditional processing of configuration files with the help of `<if>`, `<then>` and `<else>` elements so that a single configuration file can adequately target several environments. Note that conditional processing requires the Janino library.

The general format for conditional statements is shown below.

```
   <!-- 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>
```

The condition is a Java expression in which only context properties or system properties are accessible. For a key passed as argument, the `property()` or its shorter equivalent `p()` methods return the String value of the property. For example, to access the value of a property with key "k", you would write `property("k")` or equivalently `p("k")`. If the property with key "k" is undefined, the property method will return the empty string and not null. This avoids the need to check for null values.

The `isDefined()` method can be used to check whether a property is defined. For example, to check whether the property "k" is defined you would write `isDefined("k")` Similarly, if you need to check whether a property is null, the `isNull()` method is provided. Example: `isNull("k")`.

```
<configuration debug="true">

  <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>
```

Conditional processing is supported *anywhere* within the `<configuration>` element. Nested if-then-else statements are also supported. However, XML syntax is awfully cumbersome and is ill suited as the foundation of a general purpose programming language. Consequently, too many conditionals will quickly render your configuration files incomprehensible to subsequent readers, including yourself.

Obtaining variables from JNDI

Under certain circumstances, you may want to make use of env-entries stored in JNDI. The `<insertFromJNDI>` configuration directive extracts an env-entry stored in JNDI and inserts the property in local scope with key specified by the *as* attribute. As all properties, it is possible to insert the new property into a different scope with the help of the *scope* attribute.

Example: Insert as properties env-entries obtained via JNDI (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 ${CONTEXT_NAME} %level %msg %logger{50}%n</pattern>
    </encoder>
  </appender>

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

In this last example, the "java:comp/env/appName" env-entry is inserted as the *appName* property. Note that the `<contextName>` directive sets the context name based on the value of the *appName* property inserted by the previous `<insertFromJNDI>` directive.

File inclusion 文件引入

Joran supports including parts of a configuration file from another file. This is done by declaring a `<include>` element, as shown below:

Example: File include (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>
```

The target file MUST have its elements nested inside an `<included>` element. For example, a `ConsoleAppender` could be declared as:

Example: File include (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>
```

Again, please note the mandatory `<included>` element.

The contents to include can be referenced as a file, as a resource, or as a URL.

- **As a file:**
  To include a file use the *`file`* attribute. You can use relative paths but note that the current directory is defined by the application and is not necessarily related to the path of the configuration file.

- **As a resource:**
  To include a resource, i.e a file found on the class path, use the *`resource`* attribute.

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

- **As a URL:**
  To include the contents of a URL use the *`url`* attribute.

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

If it cannot find the file to be included, logback will complain by printing a status message. In case the included file is optional, you can suppress the warning message by setting *`optional`* attribute to `true` in the `<include>` element.

```
<include optional="true" ..../>
```

Adding a context listener 增加一个“上下文监听器”

Instances of the LoggerContextListener interface listen to events pertaining to the lifecycle of a logger context.

`JMXConfigurator` is one implementation of the `LoggerContextListener` interface. It is described in a subsequent chapter.

LevelChangePropagator

As of version 0.9.25, logback-classic ships with LevelChangePropagator, an implementation of `LoggerContextListener` which propagates changes made to the level of any logback-classic logger onto the java.util.logging framework. Such propagation eliminates the performance impact of disabled log statements. Instances of LogRecord will be sent to logback (via SLF4J) only for enabled log statements. This makes it reasonable for real-world applications to use the jul-to-slf4j bridge.

The contextListener element can be used to install `LevelChangePropagator` as shown next.

```
<configuration debug="true">
  <contextListener class="ch.qos.logback.classic.jul.LevelChangePropagator"/>
  ....
</configuration>
```

Setting the resetJUL property of LevelChangePropagator will reset all previous level configurations of all j.u.l. loggers. However, previously installed handlers will be left untouched.

```
<configuration debug="true">
  <contextListener class="ch.qos.logback.classic.jul.LevelChangePropagator">
    <resetJUL>true</resetJUL>
  </contextListener>
  ....
</configuration>
```