Chapter 2: Architecture

All true classification is genealogical.

—CHARLES DARWIN, *The Origin of Species*

学习一门课程，而不应用这些信息到具体的问题

It is difficult, if not impossible, for anyone to learn a subject purely by reading about it, without applying the information to specific problems and thereby forcing himself to think about what has been read. Furthermore, we all learn best the things that we have discovered ourselves.

我们都学最好的东西

—DONALD KNUTH, *The Art of Computer Programming*

《计算机程序设计艺术》

Authors: Ceki Gülcü, Sébastien Pennec, Carl Harris

Logback's architecture Logback的架构

Logback's basic architecture is sufficiently generic so as to apply under different circumstances. At the present time, logback is divided into three modules, logback-core, logback-classic and logback-access.

The *core* module lays the groundwork for the other two modules. The *classic* module extends *core*. The classic module corresponds to a significantly improved version of log4j. Logback-classic natively implements the SLF4J API so that you can readily switch back and forth between logback and other logging systems such as log4j or java.util.logging (JUL) introduced in JDK 1.4. The third module called *access* integrates with Servlet containers to provide HTTP-access log functionality. A separate document covers access module documentation.

In the remainder of this document, we will write "logback" to refer to the logback-classic module.

Logger, Appenders and Layouts 日志记录器、输出端和日志格式化器

三个主要的类：

Logback is built upon three main classes: `Logger`, `Appender` and `Layout`. These three types of components work together to enable developers to log messages according to message type and level, and to control at runtime how these messages are formatted and where they are reported.

The `Logger` class is part of the logback-classic module. On the other hand, the `Appender` and `Layout` interfaces are part of logback-core. As a general-purpose module, logback-core has no notion of loggers.

Logger context 日志上下文

The first and foremost advantage of any logging API over plain `System.out.println` resides in its ability to disable certain log statements while allowing others to print unhindered. This capability assumes that the logging space, that is, the space of all possible logging statements, is categorized according to some developer-chosen criteria. In logback-classic, this categorization is an inherent part of loggers. Every single logger is attached to a `LoggerContext` which is responsible for manufacturing loggers as well as arranging them in a tree like hierarchy.

Loggers are named entities. Their names are case-sensitive and they follow the hierarchical naming rule: 层次命名规则：

> **Named Hierarchy** 如果一个日志名称是后代日志名称的前缀，那么它则是其它日志的祖先。
>
> A logger is said to be an ancestor of another logger if its name followed by a dot is a

> prefix of the descendant logger name. A logger is said to be a parent of a child logger if there are no ancestors between itself and the descendant logger.

For example, the logger named `"com.foo"` is a parent of the logger named `"com.foo.Bar"`. Similarly, `"java"` is a parent of `"java.util"` and an ancestor of `"java.util.Vector"`. This naming scheme should be familiar to most developers.

根记录器驻留在记录器层次结构的顶部。

The root logger resides at the top of the logger hierarchy. It is exceptional in that it is part of every hierarchy at its inception. Like every logger, it can be retrieved by its name, as follows:

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

All other loggers are also retrieved with the class static `getLogger` method found in the org.slf4j.LoggerFactory class. This method takes the name of the desired logger as a parameter. Some of the basic methods in the `Logger` interface are listed below.

```
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);
}
```

Effective Level aka Level Inheritance 有效的级别、级别继承

Loggers may be assigned levels. The set of possible levels (TRACE, DEBUG, INFO, WARN and ERROR) are defined in the `ch.qos.logback.classic.Level` class. Note that in logback, the `Level` class is final and cannot be sub-classed, as a much more flexible approach exists in the form of `Marker` objects.

If a given logger is not assigned a level, then it inherits one from its closest ancestor with an assigned level. More formally:

> The effective level for a given logger *L*, is equal to the first non-null level in its hierarchy, starting at *L* itself and proceeding upwards in the hierarchy towards the root logger.

To ensure that all loggers can eventually inherit a level, the root logger always has an assigned level. By default, this level is DEBUG.

Below are four examples with various assigned level values and the resulting effective (inherited) levels according to the level inheritance rule.

Example 1

Logger name	Assigned level	Effective level
root	DEBUG	DEBUG
X	none	DEBUG
X.Y	none	DEBUG
X.Y.Z	none	DEBUG

In example 1 above, only the root logger is assigned a level. This level value, `DEBUG`, is inherited by the other loggers `X`, `X.Y` and `X.Y.Z`

Example 2

Logger name	Assigned level	Effective level
root	ERROR	ERROR
X	INFO	INFO
X.Y	DEBUG	DEBUG
X.Y.Z	WARN	WARN

In example 2 above, all loggers have an assigned level value. Level inheritance does not come into play.

Example 3

Logger name	Assigned level	Effective level
root	DEBUG	DEBUG
X	INFO	INFO
X.Y	none	INFO
X.Y.Z	ERROR	ERROR

In example 3 above, the loggers `root`, `X` and `X.Y.Z` are assigned the levels `DEBUG`, `INFO` and `ERROR` respectively. Logger `X.Y` inherits its level value from its parent `X`.

Example 4

Logger name	Assigned level	Effective level
root	DEBUG	DEBUG
X	INFO	INFO
X.Y	none	INFO
X.Y.Z	none	INFO

In example 4 above, the loggers `root` and `X` and are assigned the levels `DEBUG` and `INFO` respectively. The loggers `X.Y` and `X.Y.Z` inherit their level value from their nearest parent `X`, which has an assigned level.

Printing methods and the basic selection rule 打印方法和基本的选择规则

根据定义，打印方法决定日志记录请求的级别。

By definition, the printing method determines the level of a logging request. For example, if `L` is a logger instance, then the statement `L.info("..")` is a logging statement of level INFO.

如果日志记录请求的级别高于或等于日志记录器的有效级别，那么日志记录功能就开启。

A logging request is said to be *enabled* if its level is higher than or equal to the effective level of its logger. Otherwise, the request is said to be *disabled*. As described previously, a logger without an assigned level will inherit one from its nearest ancestor. This rule is summarized below.

> **Basic Selection Rule** 基本的选择规则
>
> A log request of level *p* issued to a logger having an effective level *q*, is enabled if *p >= q*.

This rule is at the heart of logback. It assumes that levels are ordered as follows:
`TRACE < DEBUG < INFO < WARN < ERROR`.

In a more graphic way, here is how the selection rule works. In the following table, the vertical header shows the level of the logging request, designated by *p*, while the horizontal header shows effective level of the logger, designated by *q*. The intersection of the rows (level request) and columns (effective level) is the boolean resulting from the basic selection rule.

level of request *p*	effective level *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**

Here is an example of the basic selection rule.

```
import ch.qos.logback.classic.Level;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
....

// get a logger instance named "com.foo". Let us further assume that the
// logger is of type  ch.qos.logback.classic.Logger so that we can
// set its level
ch.qos.logback.classic.Logger logger =
        (ch.qos.logback.classic.Logger) LoggerFactory.getLogger("com.foo");
//set its Level to INFO. The setLevel() method requires a logback logger
logger.setLevel(Level. INFO);

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

// This request is enabled, because WARN >= INFO
logger.warn("Low fuel level.");

// This request is disabled, because DEBUG < INFO.
logger.debug("Starting search for nearest gas station.");

// The logger instance barlogger, named "com.foo.Bar",
// will inherit its level from the logger named
// "com.foo" Thus, the following request is enabled
// because INFO >= INFO.
barlogger.info("Located nearest gas station.");

// This request is disabled, because DEBUG < INFO.
barlogger.debug("Exiting gas station search");
```

Retrieving Loggers 可检索的日志记录器

Calling the `LoggerFactory.getLogger` method with the same name will always return a reference to the exact same logger object.

For example, in 单例模式

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

`x` and `y` refer to *exactly* the same logger object.

Thus, it is possible to configure a logger and then to retrieve the same instance somewhere else in the code without passing around references. In fundamental contradiction to biological parenthood, where parents always precede their children, logback loggers can be created and configured in any order. In particular, a "parent" logger will find and link to its descendants even if it is instantiated

after them.

logback环境的配置通常在应用初始化时完成。
首选方法是通过读取配置文件，这种方法将在稍后讨论。

Configuration of the logback environment is typically done at application initialization. The preferred way is by reading a configuration file. This approach will be discussed shortly.

Logback makes it easy to name loggers by *software component*. This can be accomplished by instantiating a logger in each class, with the logger name equal to the fully qualified name of the class. This is a useful and straightforward method of defining loggers. As the log output bears the name of the generating logger, this naming strategy makes it easy to identify the origin of a log message. However, this is only one possible, albeit common, strategy for naming loggers. Logback does not restrict the possible set of loggers. As a developer, you are free to name loggers as you wish.

Nevertheless, naming loggers after the class where they are located seems to be the best general strategy known so far.

Appenders and Layouts

输出端和日志格式化器

可选择的开关功能的日志记录请求

The ability to selectively enable or disable logging requests based on their logger is only part of the picture. Logback allows logging requests to print to multiple destinations. In logback speak, an output destination is called an appender. Currently, appenders exist for the console, files, remote socket servers, to MySQL, PostgreSQL, Oracle and other databases, JMS, and remote UNIX Syslog daemons.

More than one appender can be attached to a logger.

The `addAppender` method adds an appender to a given logger. Each enabled logging request for a given logger will be forwarded to all the appenders in that logger as well as the appenders higher in the hierarchy. In other words, appenders are inherited additively from the logger hierarchy. For example, if a console appender is added to the root logger, then all enabled logging requests will at least print on the console. If in addition a file appender is added to a logger, say *L*, then enabled logging requests for *L* and *L*'s children will print on a file *and* on the console. It is possible to override this default behavior so that appender accumulation is no longer additive by setting the additivity flag of a logger to false.

The rules governing appender additivity are summarized below.

输出端的可添加性

Appender Additivity

The output of a log statement of logger *L* will go to all the appenders in *L* and its ancestors. This is the meaning of the term "appender additivity".

However, if an ancestor of logger *L*, say *P*, has the additivity flag set to false, then *L*'s output will be directed to all the appenders in *L* and its ancestors up to and including *P* but not the appenders in any of the ancestors of *P*.

Loggers have their additivity flag set to true by default.

The table below shows an example:

Logger Name	Attached Appenders	Additivity Flag	Output Targets	Comment
root	A1	not applicable	A1	Since the root logger stands at the top of the logger hierarchy, the additivity flag does not apply to it.
x	A-x1, A-x2	true	A1, A-x1, A-x2	Appenders of "x" and of root.

x.y	none	true	A1, A-x1, A-x2	Appenders of "x" and of root.
x.y.z	A-xyz1	true	A1, A-x1, A-x2, A-xyz1	Appenders of "x.y.z", "x" and of root.
security	A-sec	false	A-sec	No appender accumulation since the additivity flag is set to `false`. Only appender A-sec will be used.
security.access	none	true	A-sec	Only appenders of "security" because the additivity flag in "security" is set to `false`.

More often than not, users wish to customize not only the output destination but also the output format. This is accomplished by associating a *layout* with an appender. The layout is responsible for formatting the logging request according to the user's wishes, whereas an appender takes care of sending the formatted output to its destination. The `PatternLayout`, part of the standard logback distribution, lets the user specify the output format according to conversion patterns similar to the C language `printf` function.

For example, the PatternLayout with the conversion pattern "%-4relative [%thread] %-5level %logger{32} - %msg%n" will output something akin to:

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

程序消耗的毫秒数

The first field is the number of milliseconds elapsed since the start of the program. The second field is the thread making the log request. The third field is the level of the log request. The fourth field is the name of the logger associated with the log request. The text after the '-' is the message of the request.

Parameterized logging 参数化日志

实现Logger接口

Given that loggers in logback-classic implement the SLF4J's Logger interface, certain printing methods admit more than one parameter. These printing method variants are mainly intended to improve performance while minimizing the impact on the readability of the code.

这些打印方法的变体主要是为了提高性能，同时最小化对代码可读性的影响。

For some Logger `logger`, writing,

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

构建消息参数的成本：将变量转化为字符串，并连接中间字符串。

incurs the cost of constructing the message parameter, that is converting both integer `i` and `entry[i]` to a String, and concatenating intermediate strings. This is regardless of whether the message will be logged or not.

One possible way to avoid the cost of parameter construction is by surrounding the log statement with a test. Here is an example.

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

This way you will not incur the cost of parameter construction if debugging is disabled for `logger`. On the other hand, if the logger is enabled for the DEBUG level, you will incur the cost of evaluating whether the logger is enabled or not, twice: once in `debugEnabled` and once in `debug`. In practice, this overhead is insignificant because evaluating a logger takes less than 1% of the time it takes to actually log a request.

Better alternative 更好地选择

There exists a convenient alternative based on message formats. Assuming `entry` is an object, you can write: 基于消息格式的一种方便选择

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

仅在评判是否需要记录日志后，才会格式化消息并使用参数替换'{}'。
Only after evaluating whether to log or not, and only if the decision is positive, will the logger implementation format the message and replace the '{}' pair with the string value of `entry`. In other words, this form does not incur the cost of parameter construction when the log statement is disabled.

The following two lines will yield the exact same output. However, in case of a *disabled* logging statement, the second variant will outperform the first variant by a factor of at least 30.

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

A two argument variant is also available. For example, you can write:

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

If three or more arguments need to be passed, an `Object[]` variant is also available. For example, you can write:

```
Object[] paramArray = {newVal, below, above};
logger.debug("Value {} was inserted between {} and {}.", paramArray);
```

A peek under the hood 引擎盖下的一眼

After we have introduced the essential logback components, we are now ready to describe the steps that the logback framework takes when the user invokes a logger's printing method. Let us now analyze the steps logback takes when the user invokes the `info()` method of a logger named *com.wombat*.

1. Get the filter chain decision 1. 获取"过滤链"的决策

If it exists, the `TurboFilter` chain is invoked. Turbo filters can set a context-wide threshold, or filter out certain events based on information such as `Marker`, `Level`, `Logger`, message, or the `Throwable` that are associated with each logging request. If the reply of the filter chain is `FilterReply.DENY`, then the logging request is dropped. If it is `FilterReply.NEUTRAL`, then we continue with the next step, i.e. step 2. In case the reply is `FilterReply.ACCEPT`, we skip the next and directly jump to step 3.

2. Apply the basic selection rule 2. 应用"基本的选择规则"

At this step, logback compares the effective level of the logger with the level of the request. If the logging request is disabled according to this test, then logback will drop the request without further processing. Otherwise, it proceeds to the next step.

3. Create a `LoggingEvent` object 3. 创建一个"日志记录事件"对象

If the request survived the previous filters, logback will create a `ch.qos.logback.classic.LoggingEvent` object containing all the relevant parameters of the request, such as the logger of the request, the request level, the message itself, the exception that might have been passed along with the request, the current time, the current thread, various data about the class that issued the logging

request and the `MDC`. Note that some of these fields are initialized lazily, that is only when they are actually needed. The `MDC` is used to decorate the logging request with additional contextual information. MDC is discussed in a subsequent chapter.

4. Invoking appenders 4. 调用“输出端”

After the creation of a `LoggingEvent` object, logback will invoke the `doAppend()` methods of all the applicable appenders, that is, the appenders inherited from the logger context.

All appenders shipped with the logback distribution extend the `AppenderBase` abstract class that implements the `doAppend` method in a synchronized block ensuring thread-safety. The `doAppend()` method of `AppenderBase` also invokes custom filters attached to the appender, if any such filters exist. Custom filters, which can be dynamically attached to any appender, are presented in a separate chapter.

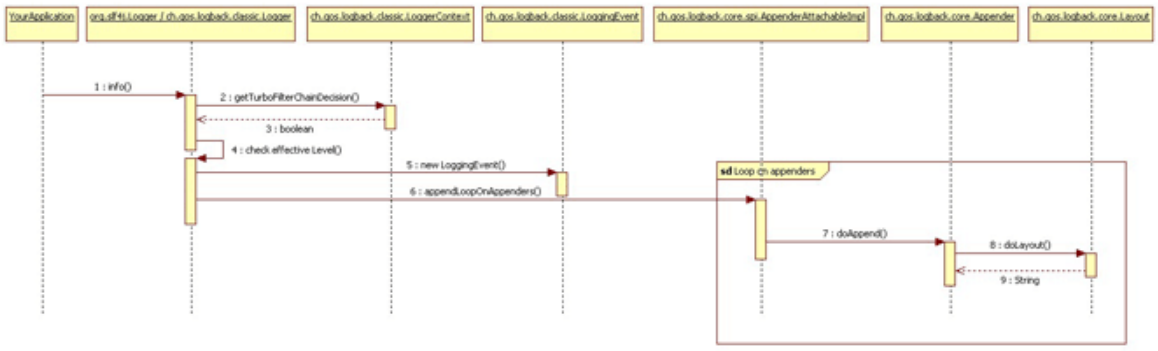
5. Formatting the output 5. 格式化输出

It is responsibility of the invoked appender to format the logging event. However, some (but not all) appenders delegate the task of formatting the logging event to a layout. A layout formats the `LoggingEvent` instance and returns the result as a String. Note that some appenders, such as the `SocketAppender`, do not transform the logging event into a string but serialize it instead. Consequently, they do not have nor require a layout.

6. Sending out the `LoggingEvent` 6. 发送“日志记录事件”

After the logging event is fully formatted it is sent to its destination by each appender.

Here is a sequence UML diagram to show how everything works. You might want to click on the image to display its bigger version.



Performance 性能

One of the often-cited arguments against logging is its computational cost. This is a legitimate concern as even moderately-sized applications can generate thousands of log requests. Much of our development effort is spent measuring and tweaking logback's performance. Independently of these efforts, the user should still be aware of the following performance issues.

1. Logging performance when logging is turned off entirely 1. 当日志记录被完全关闭时

You can turn off logging entirely by setting the level of the root logger to `Level.OFF`, the highest possible level. When logging is turned off entirely, the cost of a log request consists of a method invocation plus an integer comparison. On a 3.2Ghz Pentium D machine this cost is typically around 20 nanoseconds.

However, any method invocation involves the "hidden" cost of parameter construction. For

example, for some logger *x* writing,

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

incurs the cost of constructing the message parameter, i.e. converting both integer `i` and `entry[i]` to a string, and concatenating intermediate strings, regardless of whether the message will be logged or not.

参数构造的成本也很高，并依赖于参数的大小

The cost of parameter construction can be quite high and depends on the size of the parameters involved. To avoid the cost of parameter construction you can take advantage of SLF4J's parameterized logging:

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

This variant will not incur the cost of parameter construction. Compared to the previous call to the `debug()` method, it will be faster by a wide margin. The message will be formatted only if the logging request is to be sent to attached appenders. Moreover, the component that formats messages is highly optimized.

Notwithstanding the above placing log statements in tight loops, i.e. very frequently invoked code, is a lose-lose proposal, likely to result in degraded performance. Logging in tight loops will slow down your application even if logging is turned off, and if logging is turned on, will generate massive (and hence useless) output.

2. The performance of deciding whether to log or not to log when logging is turned on.

2. 决定是否记录日志

In logback, there is no need to walk the logger hierarchy. A logger knows its effective level (that is, its level, once level inheritance has been taken into consideration) when it is created. Should the level of a parent logger be changed, then all child loggers are contacted to take notice of the change. Thus, before accepting or denying a request based on the effective level, the logger can make a quasi-instantaneous decision, without needing to consult its ancestors.

做出一个准实时的决定

3. Actual logging (formatting and writing to the output device)

3. 实际的日志记录（格式化和写到输出终端）

This is the cost of formatting the log output and sending it to its target destination. Here again, a serious effort was made to make layouts (formatters) perform as quickly as possible. The same is true for appenders. The typical cost of actually logging is about 9 to 12 microseconds when logging to a file on the local machine. It goes up to several milliseconds when logging to a database on a remote server.

logback最重要的设计目标之一是其执行速度，这个仅次于可靠性。

Although feature-rich, one of the foremost design goals of logback was speed of execution, a requirement which is second only to reliability. Some logback components have been rewritten several times to improve performance.