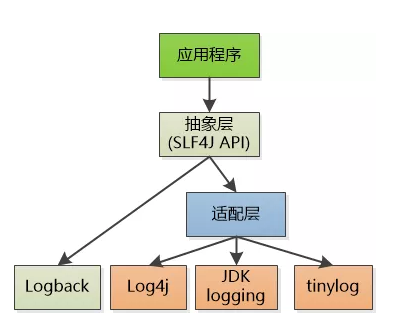
日志系统

logback中文网

<http://www.logback.cn/03%E7%AC%AC%E4%B8%89%E7%AB%A0logback%E7%9A%84%E9%85%8D%E7%BD%AE.html>



## 日志打印级别

如果一条的日志的打印级别大于 logger 的有效级别，该条日志才可以被打印出来。这条规则总结如下：

**基本选择规则**

日志的打印级别为 *p*，Logger 实例的级别为 *q*，如果 *p* >= *q*，则该条日志可以打印出来。

这条规则是 logbakc 的核心。各级别的排序为：**TRACE** < **DEBUG** < **INFO** < **WARN** < **ERROR**

## Logger 上下文

任何日志 API 的优势在于它能够禁止某些日志的输出，但是又不会妨碍另一些日志的输出。通过假定一个日志空间，这个空间包含所有可能的日志语句，这些日志语句根据开发人员设定的标准来进行分类。在 logback-classic 中，分类是 logger 的一部分，每一个 logger 都依附在 LoggerContext 上，它负责产生 logger，并且通过一个树状的层级结构来进行管理。

一个 Logger 被当作为一个实体，它们的命名是大小写敏感的，并且遵循一下规则：

命名层次结构

如果一个 logger 的名字加上一个 . 作为另一个 logger 名字的前缀，那么该 logger 就是另一个 logger 的祖先。如果一个 logger 与另一个 logger 之间没有其它的 logger ，则该 logger 就是另一个 logger 的父级。

例如：名为 com.foo 的 logger 是名为 com.foo.Bar 的 logger 的父级。名为 java 的 logger 是名为 java.util 的父级，是名为 java.util.Vector 的祖先。

root logger 作为 logger 层次结构的最高层。它是一个特殊的 logger，因为它是每一个层次结构的一部分。每一个 logger 都可以通过它的名字去获取。

如果一个给定的 logger 没有指定一个层级，那么它就会继承离它最近的一个祖先的层级。更正式的说法是：

对于一个给定的名为 *L* 的 logger，它的有效层级为从自身一直回溯到 root logger，直到找到第一个不为空的层级作为自己的层级。

为了确保所有的 logger 都有一个层级，root logger 会有一个默认层级 --- DEBUG

## Appender 输出目的地

有选择的启用或者禁用日志的输出只是 logger 的一部分功能。logback 允许日志在多个地方进行输出。站在 logback 的角度来说，输出目的地叫做 appender。appender 包括console、file、remote socket server、MySQL、PostgreSQL、Oracle 或者其它的数据库、JMS、remote UNIX Syslog daemons 中。

一个 logger 可以有多个 appender。

logger 通过 addAppender 方法来新增一个 appender。对于给定的 logger，每一个允许输出的日志都会被转发到该 logger 的所有 appender 中去。换句话说，appender 从 logger 的层级结构中去继承叠加性。例如：如果 root logger 添加了一个 console appender，所有允许输出的日志至少会在控制台打印出来。如果再给一个叫做 ***L*** 的 logger 添加了一个 file appender，那么 ***L*** 以及 ***L*** 的子级 logger 都可以在文件和控制台打印日志。可以通过设置 additivity = false 来改写默认的设置，这样 appender 将不再具有叠加性。

appender 的叠加性规则如下：

appender 的叠加性

logger *L* 的日志输出语句会遍历 *L* 和它的子级中所有的 appender。这就是所谓的 appender 叠加性（appender additivity）

如果 *L* 的子级 logger 为 *P*，且 *P* 设置了 additivity = false，那么 *L* 的日志会在 *L* 所有 的 appender 包括 *P* 本身的 appender 中输出，但是不会在 *P* 的子级 appender 中输出。

logger 默认设置 additivity = true。

## Layout 日志输出格式

通常，用户既想自定义日志的输出地，也想自定义日志的输出格式。通过给 appender 添加一个 *layout* 可以做到。layout 的作用是将日志格式化，而 appender 的作用是将格式化后的日志输出到指定的目的地。**PatternLayout** 能够根据用户指定的格式来格式化日志，类似于 C 语言的 printf 函数。

例：PatternLayout 通过格式化串 "%-4relative [%thread] %-5level %logger{32} - %msg%n" 会将日志格式化成如下结果：

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

第一个参数表示程序启动以来的耗时，单位为毫秒。第二个参数表示当前的线程号。第三个参数表示当前日志的级别。第四个参数是 logger 的名字。“-” 之后是具体的日志信息。

### 参数化日志

考虑到 logback-classic 实现了 SLF4J 的 Logger 接口，一些打印方法可以接收多个传参。这些打印方法的变体主要是为了提高性能以及减少对代码可读性的影响。

为了避免构建参数带来的损耗，可以在日志记录之前做一个判断，如下：

if(logger.isDebugEnabled()) {

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

}

### 使用{}格式化

Object entry = new SomeObject();

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

只有在需要打印 debug 信息的时候，才会去格式化日志信息，将 '{}' 替换成 entry 的字符串形式。也就是说在这种情况下，如果禁止了日志的打印，也不会有构建参数上的性能消耗。

使用两个参数的例子如下：

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

如果需要使用三个或三个以上的参数，可以采用如下的形式：

Object[] paramArray = {newVal, below, above};

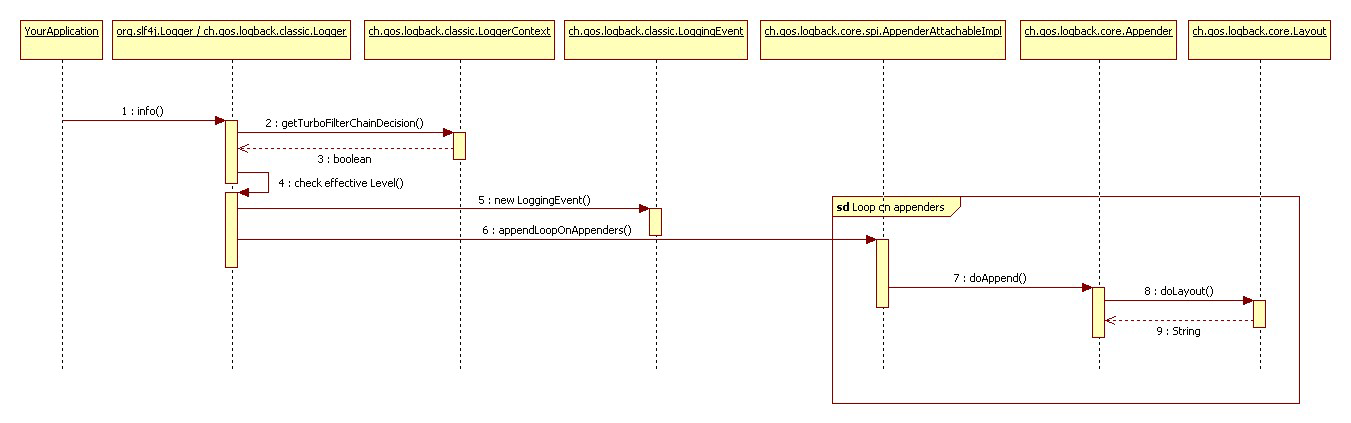
logger.debug("Value {} was inserted between {} and {}.", paramArray);

构建参数的成本取决于参数的大小，为了避免不必要的性能损耗，可以使用 SLF4J's 的参数化构建：

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

这种形式不会有构建参数的成本在里面。与上一个例子做比较，这个的速度会更快。只有当日志信息传递给了附加的 appender 时才会被格式化，而且格式化日志信息的组件也是被优化过的。

## logback 执行步骤的 UML 图



## logback的初始化步骤

以下是 logback 的初始化步骤：

1. logback 会在类路径下寻找名为 logback-test.xml 的文件。
2. 如果没有找到，logback 会继续寻找名为 logback.groovy 的文件。
3. 如果没有找到，logback 会继续寻找名为 logback.xml 的文件。
4. 如果没有找到，将会通过 JDK 提供的 [ServiceLoader](https://docs.oracle.com/javase/6/docs/api/java/util/ServiceLoader.html) 工具在类路径下寻找文件 *META-INFO/services/ch.qos.logback.classic.spi.Configurator*，该文件的内容为实现了 [Configurator](https://logback.qos.ch/xref/ch/qos/logback/classic/spi/Configurator.html) 接口的实现类的全限定类名。
5. 如果以上都没有成功，logback 会通过 [BasicConfigurator](https://logback.qos.ch/xref/ch/qos/logback/classic/BasicConfigurator.html) 为自己进行配置，并且日志将会全部在控制台打印出来。

最后一步的目的是为了保证在所有的配置文件都没有被找到的情况下，提供一个默认的（但是是非常基础的）配置。

## 当配置文件更改时，自动加载

为了让 logback 能够在配置文件改变的时候自动去扫描，需要在 <configuration> 标签上添加 scan=true 属性。

*Example*

<configuration scan="true">

...

</configuration>

默认情况下，一分钟扫描一次配置文件，看是否有更改。通过 <configuration> 标签上的 scanPeriod 属性可以指定扫描周期。扫描周期的时间单位可以是毫秒、秒、分钟或者小时。

*Example*：

<configuration scan="true" scanPeriod="30 seconds"

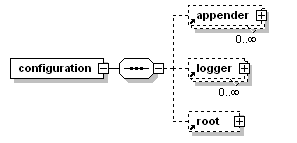
...

</configuration>

注意：如果没有指定时间单位，则默认为毫秒。

## 配置文件的语法

logback 的配置文件非常的灵活，不需要指定 DTD 或者 xml 文件需要的语法。但是，最基本的结构为 <configuration> 元素，包含 0 或多个 <appender> 元素，其后跟 0 或多个 <logger> 元素，其后再跟最多只能存在一个的 <root> 元素。基本结构图如下：



### 屏蔽组件输出

如果我们不想看到属于 "chapters.configuration" 组件中任何的 DEBUG 信息。

*Example*：sample2.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" level="INFO" />

<root level="DEBUG">

<appender-ref ref="STDOUT" />

</root>

</configuration>

### 配置 appender

appender 通过 <appender> 元素进行配置，需要两个强制的属性 *name* 与 *class*。*name* 属性用来指定 appender 的名字，*class* 属性需要指定类的全限定名用于实例化。<appender> 元素可以包含 0 或一个 <layout> 元素，0 或多个 <encoder> 元素，0 或多个 <filter> 元素。除了这些公共的元素之外，<appender> 元素可以包含任意与 appender 类的 JavaBean 属性相一致的元素。

通过多个 appender 输出日志就像定义多个 appender 以及将它们关联到 logger 上一样简单。

*Example*：multiple.xml

<?xml version="1.0" encoding="UTF-8"?>

<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>

这个配置文件定义了两个 appender：*FILE* 和 *STDOUT*。*FILE* appender 将日志输出到 *myApp.log* 文件。encoder 通过 PatternLayoutEncoder 输出日期、日志等级、线程名、logger 的名字、可以定位日志来源的文件以及所在行、具体的日志信息以及行分隔符。第二个 appender 是 STDOUT，将日志输出到控制台。它的 encoder 仅仅输出日志信息以及行分隔符。

appender 通过 *appender-ref* 元素附加到 root logger 上。每一个 appender 都有自己 encoder。encoder 通常不会设计成给所有的 appender 共享。对于 layout 也是如此。因此，logback 不会提供任何共享 encoder 和 layout 的语法。

### 重复使用 appender

在默认的情况下，appender 是可以重复使用的：logger 可以通过附加到本身的 appender 输出日志，同样的也可以附加到起祖先的身上，并输出日志。因此，如果同一个 appender 附加到多个 logger 身上，那么就导致日志重复打印。

*Example*：duplicate.xml

<?xml version="1.0" encoding="UTF-8"?>

<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" />

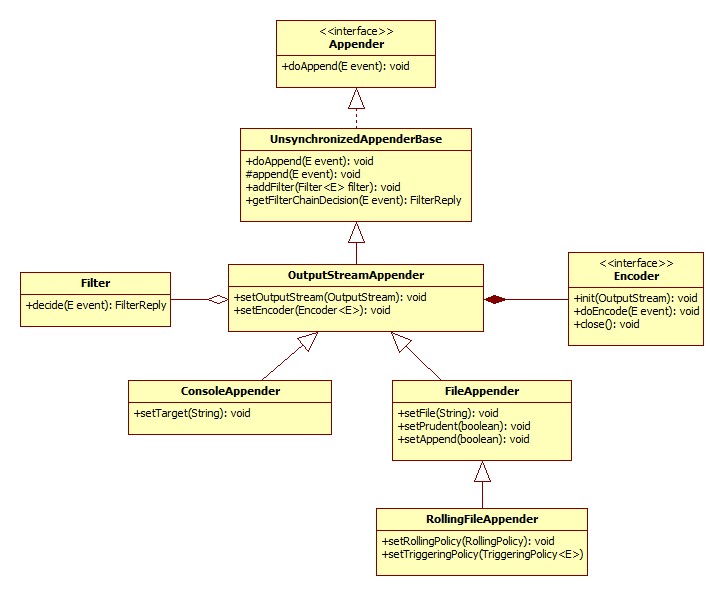
</root>

</configuration>

注意日志重复输出了，因为 appender *STDOUT* 附加到了两个 logger 身上：root 以及 *chapters.configuration*。因为 root logger 是所有 logger 的祖先，*chapters.configuration* 是 *chapters.configuration.MyApp3* 以及 *chapters.configuraion.Foo* 的父级。每一次日志请求都会被打印两次，一次是通过 *STDOUT*，一次是通过 *root*。

如果默认的累积行为对你来说不适合，你可以设置 additivity = false。

## Appender类型



### ConsoleAppender

[ConsoleAppender](https://logback.qos.ch/xref/ch/qos/logback/core/ConsoleAppender.html) 就跟名字显示的一样，是将日志事件附加到控制台，跟进一步说就是通过 *System.out* 或者 *System.err* 来进行输出。默认通过前者。

### FileAppender

[FileAppender](https://logback.qos.ch/xref/ch/qos/logback/core/FileAppender.html) 是 OutputStreamAppender 的子类，将日志事件输出到文件中。通过 file 来指定目标文件。如果该文件存在，根据 append 的值，要么将日志追加到文件中，要么该文件被截断。

立即刷新 默认情况下，每一个日志事件都会被立即刷新到底层的输出流。默认方法更加的安全，因为日志事件在你的应用没有正确关闭 appender 的情况下不会丢失。但是，要想显著的增加日志的吞吐率，你可以将 immediateFlush 设置为 false。

### RollingFileAppender

[RollingFileAppender](https://logback.qos.ch/xref/ch/qos/logback/core/rolling/RollingFileAppender.html) 继承自FileAppender，具有轮转日志文件的功能。例如，RollingFileAppender 将日志输出到 log.txt 文件，在满足了特定的条件之后，将日志输出到另外一个文件。

与 RollingFileAppender 进行交互的有两个重要的子组件。第一个是 RollingPolicy，它负责日志轮转的功能。另一个是 TriggeringPolicy，它负责日志轮转的时机。所以 RollingPolicy 负责发生什么，TriggeringPolicy 负责什么时候发生。

为了让 RollingFileAppender 生效，必须同时设置 RollingPolicy 与 TriggeringPolicy。但是，如果 RollingPolicy 也实现了 TriggeringPolicy 接口，那么只需要设置前一个就好了。

### SocketAppender and SSLSocketAppender

到目前为止，所介绍的 appender 只能将日志输出到本地资源。相反的是，[SocketAppender](https://logback.qos.ch/xref/ch/qos/logback/classic/net/SocketAppender.html) 被设计成可以将 ILoggingEvent 实例序列化再传输到远端机器。当使用 SocketAppender 时，日志事件将以明文发送，使用 [SSLSocketAppender](https://logback.qos.ch/xref/ch/qos/logback/classic/net/SSLSocketAppender.html) 时，日志事件将通过安全的通道传输。

#### 日志服务器的选择

logback classic 给服务器提供了两个选择接收来自 SocketAppender 与 SSLSocketAppender 的日志事件。

* ServerSocketReceiver 与跟它相对应的具有 SSL 功能的 SSLServerSocketReceiver 都是接收组件。可以通过配置应用中的 logback.xml 文件来接收远程 socket appender 的日志事件。查看[第十四章](https://logback.qos.ch/manual/receivers.html)获取更多的信息。
* SimpleSocketServer 与具有 SSL 功能的 SimpleSSLSocketServer 都提供了一个简单的 java 应用程序，该应用程序被设计成可配置，并且可以在命令行界面运行。这些应用仅仅等待来自 SocketAppender 或 SSLSocketAppender 的日志事件，每个被接收的日志事件按照本地服务器策略进行打印。下面给出一个简单的例子。

#### ServerSocketAppender and SSLServerSocketAppender

### SMTPAppender 邮件

[SMTPAppender](https://logback.qos.ch/xref/ch/qos/logback/classic/net/SMTPAppender.html) 收集日志事件到一个或多个固定大小的缓冲区，当用户指定的事件发生时，将从缓冲区中取出适当的内容进行发送。SMTP 邮件是异步发送的。默认情况下，当日志的级别为 ERROR 时，邮件发送将会被触发。而且默认的情况下，所有事件都使用同一个缓冲区。

### DBAppender

[DBAppender](https://logback.qos.ch/xref/ch/qos/logback/classic/db/DBAppender.html) 以一种独立于 JAVA 语言的方式将日志事件插入到三张数据库表中。

这三张表分别为：logging\_event, logging\_event\_property 与 logging\_event\_exception。在使用 DBAppender 之前，它们必须存在。logback 自带 SQL 脚本来创建表。这些脚本在 logback-classic/src/main/java/ch/qos/logback/classic/db/script 文件夹下。每一种最流行的数据库都有一个对应的脚本。如果没有你指定的数据库脚本，参考已经存在的例子，可以很简单的写一个。如果你把他们发给我们，我们很乐意在将来的版本中发布这些脚本。

### SyslogAppender

syslog 协议非常的简单：syslog 发送者将信息发送给 syslog 接收者。接收者通常叫做 syslog 守护线程 或者 syslog 服务器。logback 可以把消息发送给远程的 syslog 守护线程。通过 [SyslogAppender](https://logback.qos.ch/xref/ch/qos/logback/classic/net/SyslogAppender.html) 可以实现。

### SiftingAppender

如名字所示，SiftingAppender 根据给定的运行时属性分离或者过滤日志。例如，SiftingAppender 可以根据用户的 session 分离日志，因此不同的用户的日志会有不同的日志文件，一个用户一个日志文件。

### AsyncAppender

AsyncAppender 异步的打印 [ILoggingEvent](https://logback.qos.ch/apidocs/ch/qos/logback/classic/spi/ILoggingEvent.html)。它仅仅是作为一个事件调度器的存在，因此必须调用其它的 appender 来完成操作。

默认满了 80% 会丢弃数据 AsyncAppender 使用 [BlockingQueue](http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/BlockingQueue.html) 来缓存日志时间。AsyncAppender 会创建一个工作线程去队列的头部获取数据，并将日志事件调度给附加再 AsyncAppender 上的 appender。在默认的情况下，在队列被占用了 80% 的情况下，AsyncAppender 会丢弃掉级别为 TRACE，DEBUG，INFO 的日志事件。这个策略虽然会丢失掉日志，但是对性能有利。

### 编写你自己的 Appender

通过继承 AppenderBase 可以编写你自己的 appender。它支持处理过滤器，状态信息，以及其它大多数 appender 共享的功能。子类仅仅只需要实现 append(Object eventObject) 方法。

接下来列出来的 CountingConsoleAppender，限制了输出到控制台的日志事件的数量。当日志事件的数量达到上限时，它会退出。它使用 PatternLayoutEncoder 来格式化日志事件，还可以接收一个 limit 的参数。因此，除了 append(Object eventObject) 方法之后，还需要一些其它的方法。正如下面展示的，这些参数都是通过 logback 多种配置机制来自动处理的。

## encoder

encoder 将日志事件转换为字节数组，同时将字节数组写入到一个 OutputStream 中。encoder 在 logback 0.9.19 版本引进。

PatternLayoutEncoder 是目前真正唯一有用的 encoder。它仅仅包裹了一个 PatternLayout 就完成了大部分的工作。因此，除了不必要的复杂性，encoder 似乎不会有太多的用处。但是，我们希望一个全新的更加强大的 encoder 来改变这种印象。