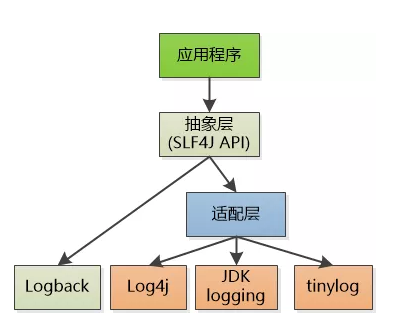
日志系统

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};

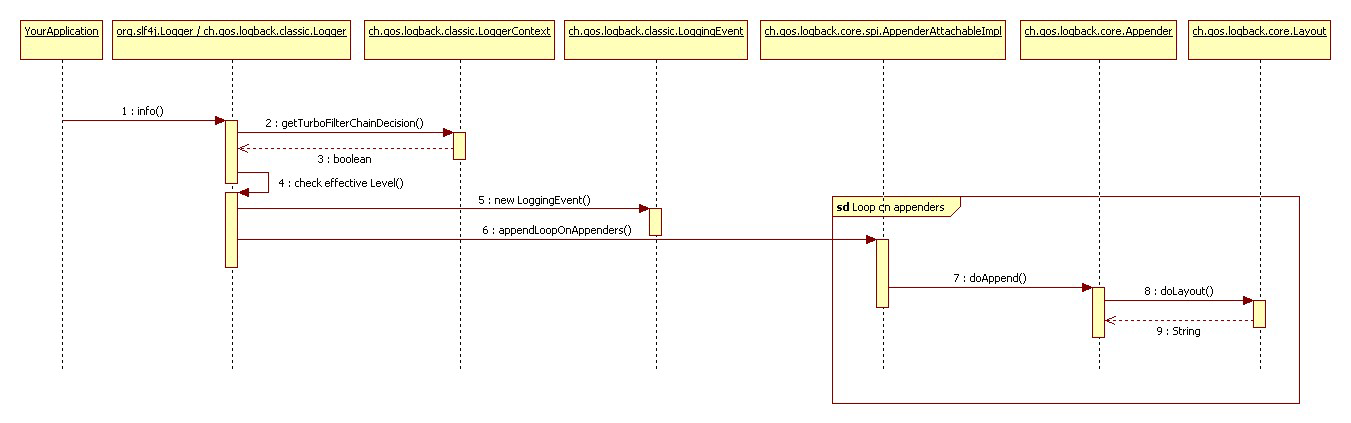
logger.debug("Value {} was inserted between {} and {}.", paramArray);

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。