**PatternLayout**

|  |  |
| --- | --- |
| **[Слово конверсии](http://logback.qos.ch/manual/layouts.html" \l "conversionWord)** | **эффект** |
| **c** { *length* } **lo** { *length* } **logger** { *length* } | Выводит имя регистратора в начало события регистрации.  Это слово преобразования принимает целое число в качестве первого и единственного параметра. Алгоритм сокращения преобразователя сократит имя регистратора, обычно без значительной потери смысла. Установка значения параметра длины в ноль является исключением. Это заставит слово преобразования возвращать подстроку справа от крайней правой точки в имени регистратора. В следующей таблице приведены примеры алгоритма сокращения в действии.   |  |  |  | | --- | --- | --- | | **Conversion specifier** | **Logger name** | **Result** | | %logger | mainPackage.sub.sample.Bar | mainPackage.sub.sample.Bar | | %logger{0} | mainPackage.sub.sample.Bar | Bar | | %logger{5} | mainPackage.sub.sample.Bar | m.s.s.Bar | | %logger{10} | mainPackage.sub.sample.Bar | m.s.s.Bar | | %logger{15} | mainPackage.sub.sample.Bar | m.s.sample.Bar | | %logger{16} | mainPackage.sub.sample.Bar | m.sub.sample.Bar | | %logger{26} | mainPackage.sub.sample.Bar | mainPackage.sub.sample.Bar |   Обратите внимание, что самый правый сегмент в имени регистратора никогда не сокращается, даже если его длина больше, чем параметр *длины* . Другие сегменты могут быть сокращены до максимум одного символа, но никогда не удаляются. |
| **C** { *length* } **class** { *length* } | Выводит полное имя класса вызывающего абонента, отправляющего запрос на регистрацию.  Как и слово преобразования *% logger* выше, это преобразование принимает целое число в качестве опции для сокращения имени класса. Ноль имеет особое значение и приведет к тому, что простое имя класса будет напечатано без префикса имени пакета. По умолчанию имя класса печатается полностью.  Генерация информации о классе вызывающего абонента не особенно быстра. Таким образом, его использование следует избегать, если скорость выполнения не является проблемой. |
| **contextName** **cn** | Выводит имя контекста регистратора, к которому подключен регистратор в источнике события. |
| **d**{pattern} **date**{pattern} **d**{pattern, timezone} **date**{pattern, timezone} | Используется для вывода даты события регистрации. Слово преобразования даты допускает строку шаблона в качестве параметра. Синтаксис шаблона совместим с форматом, принятым [java.text.SimpleDateFormat](https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html).  Вы можете указать строку *«ISO8601»* для формата даты ISO8601. Обратите внимание, что слово преобразования даты% по умолчанию соответствует [формату даты ISO 8601](http://en.wikipedia.org/wiki/ISO_8601) при отсутствии параметра шаблона.  Вот несколько примеров значений параметров. Они предполагают, что фактическая дата - пятница, 20 октября 2006 года, и что автор вернулся к работе над этим документом сразу после обеда.   |  |  | | --- | --- | |  |  | | **Conversion Pattern** | **Result** | | %d | 2006-10-20 14:06:49,812 | | %date | 2006-10-20 14:06:49,812 | | %date{ISO8601} | 2006-10-20 14:06:49,812 | | %date{HH:mm:ss.SSS} | 14:06:49.812 |   Второй параметр указывает часовой пояс. Например,% date {HH: mm: ss.SSS, Australia / Perth} будет печатать время в часовом поясе Перта, Австралия, самого изолированного города в мире. Обратите внимание, что при отсутствии параметра часового пояса используется часовой пояс по умолчанию для хост-платформы Java. Если указанный идентификатор часового пояса неизвестен или написан с ошибкой, часовой пояс GMT считается предполагаемым в соответствии со спецификацией метода [TimeZone.getTimeZone (String)](http://docs.oracle.com/javase/6/docs/api/java/util/TimeZone.html" \l "getTimeZone(java.lang.String)) .  **РАСПРОСТРАНЕННАЯ ОШИБКА** Учитывая, что запятая ',' интерпретируется как разделитель параметров, шаблон HH:mm:ss,SSSбудет интерпретироваться как шаблон HM:mm:ssи часовой пояс SSS. Если вы хотите включить запятую в шаблон даты, просто заключите шаблон в кавычки. Например,% date { **"** ЧЧ: мм: сс, ССС **"** }. |
| |  |  | | --- | --- | | **F / file** |  | | Выводит имя файла исходного файла Java, в который был отправлен запрос на регистрацию.  Генерация файловой информации не особенно быстра. Таким образом, его использование следует избегать, если скорость выполнения не является проблемой. |
| **caller{depth}** **caller{depthStart..depthEnd}** **caller{depth, evaluator-1, ... evaluator-n}** **caller{depthStart..depthEnd, evaluator-1, ... evaluator-n}** | Выводит информацию о местоположении вызывающего абонента, который сгенерировал событие регистрации.  Информация о местоположении зависит от реализации JVM, но обычно состоит из полного имени вызывающего метода, за которым следуют источник вызывающего, имя файла и номер строки в скобках.  Целое число может быть добавлено к опциям спецификатора преобразования *вызывающего абонента* для настройки глубины отображаемой информации.  Например, **% caller {2}** будет отображать следующую выдержку:  0 [main] DEBUG - logging statement  Caller+0 at mainPackage.sub.sample.Bar.sampleMethodName(Bar.java:22)  Caller+1 at mainPackage.sub.sample.Bar.createLoggingRequest(Bar.java:17)  И **% caller {3}** будет отображать этот другой отрывок:  16 [main] DEBUG - logging statement  Caller+0 at mainPackage.sub.sample.Bar.sampleMethodName(Bar.java:22)  Caller+1 at mainPackage.sub.sample.Bar.createLoggingRequest(Bar.java:17)  Caller+2 at mainPackage.ConfigTester.main(ConfigTester.java:38)  Спецификатор диапазона может быть добавлен к опциям спецификатора преобразования *вызывающего абонента* для настройки диапазона глубины отображаемой информации.  Например, **% caller {1..2}** будет отображать следующую выдержку:  0 [main] DEBUG - logging statement  Caller+0 at mainPackage.sub.sample.Bar.createLoggingRequest(Bar.java:17)  Это слово преобразования также может использовать оценщики для проверки событий регистрации по заданному критерию перед вычислением данных вызывающего абонента. Например, используя **% caller {3, CALLER\_DISPLAY\_EVAL}** отобразит три строки трассировки стека, только если оценщик с именем *CALLER\_DISPLAY\_EVAL* вернет **положительный** ответ.  Оценщики описаны ниже. |
| **L / line** | Выводит номер строки, с которой был отправлен запрос на регистрацию.  Генерация информации о номере строки не особенно быстра. Таким образом, его использование следует избегать, если скорость выполнения не является проблемой. |
| **m / msg / message** | Выводит предоставленное приложением сообщение, связанное с событием регистрации. |
| **M / method** | Выводит имя метода, по которому был отправлен запрос на регистрацию.  Генерация имени метода не особенно быстрая. Таким образом, его использование следует избегать, если скорость выполнения не является проблемой. |
| **n** | Выводит зависимый от платформы символ или символы разделителя строк.  Это слово преобразования предлагает практически такую ​​же производительность, как и использование непереносимых строк разделителя строк, таких как "\ n" или "\ r \ n". Таким образом, это предпочтительный способ указания разделителя строк. |
| **p / le / level** | Выводит уровень события регистрации. |
| **r / relative** | Выводит количество миллисекунд, прошедших с момента запуска приложения до создания события регистрации. |
| **t / thread** | Выводит имя потока, который сгенерировал событие регистрации. |
| **X** { *key: -defaultVal* } **mdc** { *key: -defaultVal* } | Выводит MDC (сопоставленный диагностический контекст), связанный с потоком, который сгенерировал событие регистрации.  Если за **словом** преобразования **mdc** следует ключ в фигурных скобках, как в **% mdc {userid}** , то будет **выведено** значение MDC, соответствующее ключу 'userid'. Если значение равно нулю, то [значение по умолчанию](http://logback.qos.ch/manual/configuration.html#defaultValuesForVariables) указывается после **: -** оператор вывода. Если значение по умолчанию не указано, выводится пустая строка.  Если ключ не указан, весь контент MDC будет выведен в формате «key1 = val1, key2 = val2».  См. [Главу о MDC](http://logback.qos.ch/manual/mdc.html) для более подробной информации по этому вопросу. |
| **ex**{depth} **exception**{depth} **throwable**{depth}  **ex**{depth, evaluator-1, ..., evaluator-n} **exception**{depth, evaluator-1, ..., evaluator-n} **throwable**{depth, evaluator-1, ..., evaluator-n} | Выводит трассировку стека исключения, связанного с событием регистрации, если оно есть. По умолчанию выводится полная трассировка стека.  *Throwable* слово преобразования может сопровождаться одним из следующих вариантов:   * *short* : печатает первую строку трассировки стека * *full* : печатает полную трассировку стека * Любое целое число: печатает указанное количество строк трассировки стека   Вот некоторые примеры:   |  |  | | --- | --- | | **Conversion Pattern** | **Result** | | %ex | mainPackage.foo.bar.TestException: Houston we have a problem  at mainPackage.foo.bar.TestThrower.fire(TestThrower.java:22)  at mainPackage.foo.bar.TestThrower.readyToLaunch(TestThrower.java:17)  at mainPackage.ExceptionLauncher.main(ExceptionLauncher.java:38) | | %ex{short} | mainPackage.foo.bar.TestException: Houston we have a problem  at mainPackage.foo.bar.TestThrower.fire(TestThrower.java:22) | | %ex{full} | mainPackage.foo.bar.TestException: Houston we have a problem  at mainPackage.foo.bar.TestThrower.fire(TestThrower.java:22)  at mainPackage.foo.bar.TestThrower.readyToLaunch(TestThrower.java:17)  at mainPackage.ExceptionLauncher.main(ExceptionLauncher.java:38) | | %ex{2} | mainPackage.foo.bar.TestException: Houston we have a problem  at mainPackage.foo.bar.TestThrower.fire(TestThrower.java:22)  at mainPackage.foo.bar.TestThrower.readyToLaunch(TestThrower.java:17 |   Это слово преобразования также может использовать оценщики для проверки событий регистрации по заданному критерию перед созданием выходных данных. Например, использование **% ex {full, EX\_DISPLAY\_EVAL}** отобразит полную трассировку стека исключения, только если оценщик с именем *EX\_DISPLAY\_EVAL* возвращает **отрицательный** ответ. Оценщики описаны ниже в этом документе.  Если вы не укажете% throwable или другое связанное с throwable слово преобразования в шаблоне преобразования, PatternLayoutавтоматически добавит его в качестве последнего слова преобразования, учитывая важность информации трассировки стека. Слово преобразования $ nopex можно заменить на% throwable, если вы не хотите, чтобы отображалась информация трассировки стека. Смотрите также слово преобразования% nopex. |
| **xEx**{depth} **xException**{depth} **xThrowable**{depth}  **xEx**{depth, evaluator-1, ..., evaluator-n} **xException**{depth, evaluator-1, ..., evaluator-n} **xThrowable**{depth, evaluator-1, ..., evaluator-n} | То же самое, что и слово преобразования% throwable выше с добавлением информации об упаковке класса.  В конце каждого фрейма стека исключения будет добавлена ​​строка, состоящая из файла JAR, содержащего соответствующий класс, за которым следует «Реализация-версия», как указано в манифесте этого JAR-файла. Эта инновационная техника была [первоначально предложена Джеймсом Страчаном](http://macstrac.blogspot.com/2008/09/better-stack-traces-in-java-with-log4j.html) . Если информация недостоверна, то данным упаковки класса будет предшествовать тильда, то есть символ «~».  Вот пример:  java.lang.NullPointerException  at com.xyz.Wombat(Wombat.java:57) **~[wombat-1.3.jar:1.3]**  at com.xyz.Wombat(Wombat.java:76) ~[wombat-1.3.jar:1.3]  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:1.5.0\_06]  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39) ~[na:1.5.0\_06]  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25) ~[na:1.5.0\_06]  at java.lang.reflect.Method.invoke(Method.java:585) ~[na:1.5.0\_06]  at org.junit.internal.runners.TestMethod.invoke(TestMethod.java:59) [junit-4.4.jar:na]  at org.junit.internal.runners.MethodRoadie.runTestMethod(MethodRoadie.java:98) [junit-4.4.jar:na]  ...etc  Logback делает все возможное, чтобы обеспечить правильность отображаемой информации об упаковке классов, даже в сколь угодно сложных иерархиях загрузчиков классов. Однако, когда он не может гарантировать абсолютную правильность информации, он префиксирует данные тильдой, то есть символом «~». Таким образом, теоретически возможно, чтобы информация об упаковке печатного класса отличалась от информации об упаковке реального класса. Итак, в приведенном выше примере, учитывая, что данным упаковки для класса Wombat предшествует тильда, вполне возможно, что в действительности правильные данные упаковки существуют [wombat.jar: 1.7].  Обратите внимание, что с учетом его потенциальной стоимости вычисление данных [**упаковки по умолчанию отключено**](http://logback.qos.ch/manual/configuration.html#packagingData) . Когда вычисление данных упаковки включено, PatternLayoutавтоматически будет использоваться суффикс% xThrowable вместо суффикса% throwable в конце строки шаблона.  [Отзывы пользователей](http://jira.qos.ch/browse/LBCLASSIC-212) показывают, что Netbeans задыхается от информации об упаковке. |
| **nopex** **nopexception** | Хотя он *притворяется,* что обрабатывает данные трассировки стека, это слово преобразования не выводит никаких данных, таким образом, фактически игнорируя исключения.  Слово преобразования% nopex позволяет пользователю переопределить PatternLayoutвнутренний механизм безопасности, который автоматически добавляет ключевое слово преобразования% xThrowable в отсутствие других исключений обработки слова преобразования. |
| **marker** | Выводит маркер, связанный с запросом регистратора.  Если маркер содержит дочерние маркеры, конвертер отображает имена родителей, а также имена детей в соответствии с форматом, показанным ниже.  *parentName [child1, child2]* |
| **property{key}** | Выводит значение, связанное со свойством с именем *key* . Соответствующие документы о том, как определить ион под названием [определить переменные](http://logback.qos.ch/manual/configuration.html#variableSubstitution) и [переменные области](http://logback.qos.ch/manual/configuration.html#scopes) . Если *ключ* не является свойством контекста логгера, *ключ* будет найден в свойствах системы.  Для *ключа* нет значения по умолчанию . Если оно опущено, возвращаемое значение будет «Property\_HAS\_NO\_KEY», объясняя условие ошибки. |
| **replace(p){r, t}** | Заменяет вхождения 'r', регулярное выражение, с его заменой 't' в строке, созданной подшаблоном 'p'. Например, «% replace (% msg) {'\ s', ''}» удалит все пробелы, содержащиеся в сообщении о событии.  Шаблон «p» может быть произвольно сложным и, в частности, может содержать несколько ключевых слов преобразования. Например, «% replace (% logger% msg) {'\.', '/'}» Заменит все точки в регистраторе или сообщении о событии косой чертой. |
| **rEx**{depth} **rootException**{depth}  **rEx**{depth, evaluator-1, ..., evaluator-n} **rootException**{depth, evaluator-1, ..., evaluator-n} | Выводит трассировку стека исключения, связанного с событием регистрации, если оно есть. Первопричина будет выводиться первой вместо стандартной «первопричины последней». Вот пример вывода (отредактированный для пробела):  java.lang.NullPointerException  at com.xyz.Wombat(Wombat.java:57) ~[wombat-1.3.jar:1.3]  at com.xyz.Wombat(Wombat.java:76) ~[wombat-1.3.jar:1.3]  Wrapped by: org.springframework.BeanCreationException: Error creating bean with name 'wombat':  at org.springframework.AbstractBeanFactory.getBean(AbstractBeanFactory.java:248) [spring-2.0.jar:2.0]  at org.springframework.AbstractBeanFactory.getBean(AbstractBeanFactory.java:170) [spring-2.0.jar:2.0]  at org.apache.catalina.StandardContext.listenerStart(StandardContext.java:3934) [tomcat-6.0.26.jar:6.0.26]  Преобразователь% rootException допускает те же дополнительные параметры, что и описанный выше преобразователь% xException, включая глубину и оценщики. Он также выводит информацию об упаковке. Короче говоря,% rootException очень похож на% xException, только порядок вывода исключений обратный.  Томаш Нуркевич, автор конвертера% rootException, документирует свой вклад в блоге, озаглавленном [«Первичная причина регистрации исключений»](http://nurkiewicz.blogspot.com/2011/09/logging-exceptions-root-cause-first.html) . |

#### Символ% имеет особое значение

Учитывая, что в контексте шаблонов преобразования знак процента имеет особое значение, чтобы включить его в качестве литерала, его необходимо экранировать обратной косой чертой, например «% d% p **\** %% m% n».

#### Ограничения на литералы сразу после конверсионных слов

В большинстве случаев литералы, естественно, содержат пробелы или другие разделительные символы, чтобы их не путать со словами преобразования. Например, шаблон «% level [% thread] -% message% n» содержит строковые литералы " ["и "] - ". Однако, если символ, который может быть частью идентификатора Java, следует сразу за словом преобразования, синтаксический анализатор шаблона logback будет обманут, думая, что литерал является частью слова преобразования. Например, шаблон «% date **% nHello»**"будет интерпретироваться как два слова преобразования% date и% nHello, и так как% nHello не является известным словом преобразования, обратный вызов выведет% PARSER\_ERROR [nHello] для% nHello. Если вы хотите, чтобы строковый литерал" Hello "немедленно разделял% n и Hello, передать пустой список аргументов% n. Например, «% date **% n {}** Hello» будет интерпретироваться как% date, за которым следует% n, за которым следует литерал «Hello».

**Скобки специальные**

В logback скобки в строке шаблона обрабатываются как токены группировки. Таким образом, можно группировать подшаблон и применять директивы форматирования к этому подшаблу. Начиная с версии 0.9.27, logback поддерживает составные слова преобразования, такие как [% replace,](http://logback.qos.ch/manual/layouts.html#replace) которые могут преобразовывать подшаблоны .

Например, шаблон

**% -30 (** % d {ЧЧ: мм: ss.SSS} [% thread] **)** % -5level% logger {32} -% msg% n

сгруппирует вывод, сгенерированный подшаблоном «% d {HH: mm: ss.SSS} [% thread]», чтобы он дополнялся справа, если он меньше 30 символов.

Если без группировки вывод был

13:09:30 [main] DEBUG c.q.logback.demo.ContextListener - Classload hashcode is 13995234

13:09:30 [main] DEBUG c.q.logback.demo.ContextListener - Initializing for ServletContext

13:09:30 [main] DEBUG c.q.logback.demo.ContextListener - Trying platform Mbean server

13:09:30 [pool-1-thread-1] INFO ch.qos.logback.demo.LoggingTask - Howdydy-diddly-ho - 0

13:09:38 [btpool0-7] INFO c.q.l.demo.lottery.LotteryAction - Number: 50 was tried.

13:09:40 [btpool0-7] INFO c.q.l.d.prime.NumberCruncherImpl - Beginning to factor.

13:09:40 [btpool0-7] DEBUG c.q.l.d.prime.NumberCruncherImpl - Trying 2 as a factor.

13:09:40 [btpool0-7] INFO c.q.l.d.prime.NumberCruncherImpl - Found factor 2

с группировкой "% -30 ()" это будет

13:09:30 [main] DEBUG c.q.logback.demo.ContextListener - Classload hashcode is 13995234

13:09:30 [main] DEBUG c.q.logback.demo.ContextListener - Initializing for ServletContext

13:09:30 [main] DEBUG c.q.logback.demo.ContextListener - Trying platform Mbean server

13:09:30 [pool-1-thread-1] INFO ch.qos.logback.demo.LoggingTask - Howdydy-diddly-ho - 0

13:09:38 [btpool0-7] INFO c.q.l.demo.lottery.LotteryAction - Number: 50 was tried.

13:09:40 [btpool0-7] INFO c.q.l.d.prime.NumberCruncherImpl - Beginning to factor.

13:09:40 [btpool0-7] DEBUG c.q.l.d.prime.NumberCruncherImpl - Trying 2 as a factor.

13:09:40 [btpool0-7] INFO c.q.l.d.prime.NumberCruncherImpl - Found factor 2

Последняя форма более удобна для чтения.

Если вам нужно трактовать символ круглой скобки как литерал, его необходимо экранировать, указав перед каждой круглой скобкой обратную косую черту. Например , **\ (** % d {ЧЧ: мм: сс.ССС} [% нить] **\)** .

**раскраска**

Группировка по круглым [скобкам,](http://logback.qos.ch/manual/layouts.html#Parentheses) как объяснено выше, позволяет раскрасить подшаблоны. Начиная с версии 1.0.5, PatternLayout распознает «% black», «% red», «% green», «% yellow», «% blue», «% magenta», «% cyan», «% white», «% серый ","% boldRed ","% boldGreen ","% boldYellow ","% boldBlue ","% boldMagenta ""% boldCyan ","% boldWhite "и"% highlight "в качестве слов преобразования. Эти слова преобразования предназначены, чтобы содержать подшаблон. Любой подшаблон, заключенный в раскрашивающее слово, будет выводиться указанным цветом.

Ниже приведен файл конфигурации, иллюстрирующий раскраску. Обратите внимание на спецификатор преобразования **% cyan**, содержащий "% logger {15}". Это выведет имя регистратора, сокращенное до 15 символов, на голубой. Спецификатор преобразования **% highlight** печатает свой подшаблон жирным красным цветом для событий уровня ERROR, красным для WARN, синим для INFO и цветом по умолчанию для других уровней.

*Пример: Подсветка уровней (logback-examples / src / main / resources / chapters /*layout / selected.xml*)*

<configuration debug="true">  
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
    **<withJansi>true</withJansi>**  
    <encoder>  
      <pattern>[%thread] **%highlight(%-5level)** **%cyan(%logger{15})** - %msg %n</pattern>  
    </encoder>  
  </appender>  
  <root level="DEBUG">  
    <appender-ref ref="STDOUT" />  
  </root>  
</configuration>

Значение withJansitrue разрешает интерпретацию кода цвета ANSI библиотекой Jansi, которая прозрачно отфильтровывает escape-последовательности ANSI, если базовый терминал несовместим. Это самый безопасный выбор для кроссплатформенного развертывания, но для него требуется org.fusesource.jansi: jansi: 1.17 или выше на пути к классам. Обратите внимание, что основанные на Unix операционные системы, такие как Linux и Mac OS X, изначально поддерживают цветовые коды ANSI и обычно не требуют включения библиотеки Jansi, но это безопасно. В Windows, однако, для включения Jansi рекомендуется использовать интерпретацию цветового кода в командной строке DOS, в противном случае существует риск отправки управляющих последовательностей ANSI, которые они не могут интерпретировать.

Вот соответствующий вывод:

[main] WARN c.l.TrivialMain - a warning message 0

[main] DEBUG c.l.TrivialMain - hello world number1

[main] DEBUG c.l.TrivialMain - hello world number2

[main] INFO c.l.TrivialMain - hello world number3

[main] DEBUG c.l.TrivialMain - hello world number4

[main] WARN c.l.TrivialMain - a warning message 5

[main] ERROR c.l.TrivialMain - Finish off with fireworks

Требуется очень мало строк кода для создания конверсионного слова. В разделе, озаглавленном « [Создание пользовательского спецификатора преобразования»,](http://logback.qos.ch/manual/layouts.html#customConversionSpecifier) обсуждаются шаги, необходимые для регистрации слова преобразования в файле конфигурации.

## Оценщики (Evaluators)

Как упоминалось выше, списки опций пригодятся, когда спецификатор преобразования должен вести себя динамически на основе одного или нескольких объектов EventEvaluator. Объекты EventEvaluator отвечают за определение того, соответствует ли данное событие ведения журнала критериям оценщика.

Давайте рассмотрим пример с EventEvaluator. Следующий файл конфигурации выводит события регистрации на консоль, отображая дату, поток, уровень, сообщения и данные вызывающего абонента. Принимая во внимание, что извлечение данных вызывающего абонента из события регистрации является дорогостоящей стороной, мы будем делать это только тогда, когда запрос регистрации поступает из определенного регистратора, и когда сообщение содержит определенную строку. Таким образом, мы гарантируем, что только определенные запросы регистрации будут генерировать и отображать информацию о вызывающем абоненте. В других случаях, когда данные вызывающего абонента являются излишними, мы не будем снижать производительность приложения.

Оценщики и, в частности, *выражения оценки* представлены в отдельном [разделе главы о фильтрах,](http://logback.qos.ch/manual/filters.html#evalutatorFilter) которые вы ДОЛЖНЫ прочитать, если хотите использовать оценщики каким-либо осмысленным образом. Также обратите внимание, что приведенные ниже примеры неявно основаны на JaninoEventEvaluator, для которого требуется библиотека Janino. Пожалуйста, смотрите [соответствующий раздел](http://logback.qos.ch/setup.html#janino) документа настройки.

*Пример: пример использования EventEvaluators (logback-examples / src / main / resources / chapters / layouts / callerEvaluatorConfig.xml)*

<configuration>  
  **<evaluator name="DISP\_CALLER\_EVAL">  
    <expression>logger.contains("chapters.layouts") &amp;&amp; \  
      message.contains("who calls thee")</expression>  
  </evaluator>**  
  
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">   
    <encoder>  
      <pattern>  
        %-4relative [%thread] %-5level - %msg%n**%caller{2, DISP\_CALLER\_EVAL}**  
      </pattern>  
    </encoder>  
  </appender>  
  
  <root level="DEBUG">   
    <appender-ref ref="STDOUT" />   
  </root>  
</configuration>

Вышеупомянутое оценочное выражение сопоставляет события, которые исходят от регистратора с именем, содержащим строку «chapters.layouts», а сообщение содержит строку «кто вам звонит». Из-за правил кодирования XML символ & не может быть записан как есть, и его необходимо экранировать как & amp ;.

Следующий класс использует некоторые характеристики, упомянутые в файле конфигурации выше.

*Пример: пример использования EventEvaluators*[*(logback-examples / src / main / java / chapters / layouts / CallerEvaluatorExample.java)*](http://logback.qos.ch/xref/chapters/layouts/CallerEvaluatorExample.html)

package **chapters.layouts**;  
  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
import ch.qos.logback.classic.LoggerContext;  
import ch.qos.logback.classic.joran.JoranConfigurator;  
import ch.qos.logback.core.joran.spi.JoranException;  
import ch.qos.logback.core.util.StatusPrinter;  
  
public class CallerEvaluatorExample {  
  
  public static void main(String[] args)  {  
    Logger logger = LoggerFactory.getLogger(CallerEvaluatorExample.class);  
    LoggerContext lc = (LoggerContext) LoggerFactory.getILoggerFactory();  
  
    try {  
      JoranConfigurator configurator = new JoranConfigurator();  
      configurator.setContext(lc);  
      configurator.doConfigure(args[0]);  
    } catch (JoranException je) {  
      // StatusPrinter will handle this  
    }  
    StatusPrinter.printInCaseOfErrorsOrWarnings(lc);  
  
    for (int i = 0; i < 5; i++) {  
      if (i == 3) {  
        logger.debug(**"who calls thee**?");  
      } else {  
        logger.debug("I know me " + i);  
      }  
    }  
  }  
}

Вышеуказанное приложение не делает ничего особенного. Выдается пять запросов на регистрацию, третий посылает сообщение "кто тебя зовет?"

Команда

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

будет давать

0 [main] DEBUG - I know me 0

0 [main] DEBUG - I know me 1

0 [main] DEBUG - I know me 2

0 [main] DEBUG - who calls thee?

Caller+0 at chapters.layouts.CallerEvaluatorExample.main(CallerEvaluatorExample.java:28)

0 [main] DEBUG - I know me 4

Когда выдается запрос на регистрацию, оценивается соответствующее событие регистрации. Только третье событие регистрации соответствует критериям оценки, в результате чего отображаются данные вызывающего абонента. Для других событий регистрации критерии оценки не совпадают, и данные вызывающего абонента не выводятся.

Можно изменить выражение, чтобы соответствовать сценарию реального мира. Например, можно объединить имя регистратора и уровень запроса. Таким образом, для запросов регистрации уровня *WARN* и выше, исходящих из чувствительной части приложения, например модуля финансовых транзакций, будут отображаться данные вызывающего абонента.

**Важно:** Со словом преобразования вызывающего абонента данные вызывающего абонента выводятся, когда выражение оценивается как true.

Давайте рассмотрим в другой ситуации. Когда исключения включаются в запрос регистрации, их трассировка стека также выводится. Тем не менее, может потребоваться подавить трассировку стека для некоторых конкретных исключений.

Код Java, показанный ниже, создает три запроса журнала, каждый с исключением. Второе исключение отличается от других: оно содержит строку «не отображать это» и относится к типу chapters.layouts.TestException. Как его команды сообщения, давайте теперь предотвратим печать второго исключения.

*Пример: пример использования EventEvaluators*[*(logback-examples / src / main / java / chapters / layouts / ExceptionEvaluatorExample.java)*](http://logback.qos.ch/xref/chapters/layouts/ExceptionEvaluatorExample.html)

package chapters.layouts;  
  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
import ch.qos.logback.classic.LoggerContext;  
import ch.qos.logback.classic.joran.JoranConfigurator;  
import ch.qos.logback.core.joran.spi.JoranException;  
import ch.qos.logback.core.util.StatusPrinter;  
  
public class ExceptionEvaluatorExample {  
  
  public static void main(String[] args) {  
    Logger logger = LoggerFactory.getLogger(ExceptionEvaluatorExample.class);  
    LoggerContext lc = (LoggerContext) LoggerFactory.getILoggerFactory();  
  
    try {  
      JoranConfigurator configurator = new JoranConfigurator();  
      configurator.setContext(lc);  
      lc.reset();  
      configurator.doConfigure(args[0]);  
    } catch (JoranException je) {  
       // StatusPrinter will handle this  
    }  
    StatusPrinter.printInCaseOfErrorsOrWarnings(lc);  
  
    for (int i = 0; i < 3; i++) {  
      if (i == 1) {  
        logger.debug("logging statement " + i, new TestException(  
            "do not display this"));  
      } else {  
        logger.debug("logging statement " + i, new Exception("display"));  
      }  
    }  
  }  
}

В следующем файле конфигурации выражение оценки соответствует событиям, содержащим бросаемый тип chapters.layouts.TextException, именно тот тип исключений, которые мы хотим подавить.

*Пример: пример использования EventEvaluators (logback-examples / src / main / resources / chapters / layouts / exceptionEvaluatorConfig.xml)*

<configuration>  
  
  **<evaluator name="DISPLAY\_EX\_EVAL">  
    <expression>throwable != null &amp;&amp; throwable instanceof  \  
      chapters.layouts.TestException</expression>  
  </evaluator>**  
          
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
    <encoder>  
      <pattern>%msg%n**%ex{full, DISPLAY\_EX\_EVAL}**</pattern>  
    </encoder>  
  </appender>  
  
  <root level="debug">  
    <appender-ref ref="STDOUT" />  
  </root>  
</configuration>

При такой конфигурации каждый раз, когда экземпляр *исключения chapters.layouts.TestException* включается в запрос регистрации, трассировка стека будет подавлена.

Запуск команды

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

будет давать

оператор ведения журнала 0 java.lang.Exception: отображение в chapters.layouts.ExceptionEvaluatorExample.main (ExceptionEvaluatorExample.java:43) [logback-examples-0.9.19.jar: na] оператор ведения журнала 1 оператор ведения журнала 2 java.lang.Exception: отобразить в chapters.layouts.ExceptionEvaluatorExample.main (ExceptionEvaluatorExample.java:43) [logback-examples-0.9.19.jar: na]

Обратите внимание, что второй оператор журнала не имеет трассировки стека. Мы эффективно подавили трассировку стека для TextException. Текст в квадратных скобках в конце каждой строки трассировки стека содержит [информацию](http://logback.qos.ch/manual/layouts.html#xThrowable) об [упаковке, которая](http://logback.qos.ch/manual/layouts.html#xThrowable) обсуждалась ранее.

**ПРИМЕЧАНИЕ.** При использовании спецификатора преобразования ***% ex*** трассировка стека отображается, когда *выражение оценивается как****ложное****.*

**Создание пользовательского спецификатора конверсии**

До этого момента мы представили встроенные слова преобразования в PatternLayout. Но также можно добавить конверсионные слова вашего собственного изготовления.

Создание пользовательского спецификатора преобразования состоит из двух этапов.

**Шаг 1**

Во-первых, вы должны расширить ClassicConverter класс. Объекты ClassicConverter отвечают за извлечение информации из экземпляров ILoggingEvent и создание строки. Например, LoggerConverter, конвертер, лежащий в основе слова преобразования% logger, извлекает имя регистратора из ILoggingEvent и возвращает его в виде строки. Это может привести к сокращению имени регистратора в процессе.

Вот конвертер клиента, который возвращает время, прошедшее с момента его создания в наносекундах:

*Пример: пример конвертера*[*(src / main / java / chapters / layouts / MySampleConverter.java)*](http://logback.qos.ch/xref/chapters/layouts/MySampleConverter.html)

public class MySampleConverter extends ClassicConverter {  
  
  long start = System.nanoTime();  
  
  **@Override**  
  **public String convert(ILoggingEvent event) {**  
    **long nowInNanos = System.nanoTime();**  
    **return Long.toString(nowInNanos-start);**  
  **}**  
}

Эта реализация довольно проста. MySampleConverterКласс расширяет ClassicConverterи реализует convertметод , который возвращает количество наносекунд , прошедших с момента его создания.

**Шаг 2**

На втором этапе мы должны сообщить logback о новом Converter. Для этого нам нужно объявить новое слово преобразования в файле конфигурации, как показано ниже:

*Пример: Пример конвертера примеров (src / main / java / chapters / layouts / mySampleConverterConfig.xml)*

<configuration>  
  
  **<conversionRule conversionWord="nanos"   
                  converterClass="chapters.layouts.MySampleConverter" />**  
          
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
    <encoder>  
      <pattern>**%-6nanos** [%thread] - %msg%n</pattern>  
    </encoder>  
  </appender>  
  
  <root level="DEBUG">  
    <appender-ref ref="STDOUT" />  
  </root>  
</configuration>

Как только новое слово преобразования было объявлено в файле конфигурации, мы можем ссылаться на него в PatternLayoutшаблоне, как и любое другое слово преобразования.

Команда:

java chapters.layouts.SampleLogging src / main / java / chapters / layouts / mySampleConverterConfig.xml

должен привести к выводу, похожему на:

4868695 [main] DEBUG - все идет хорошо

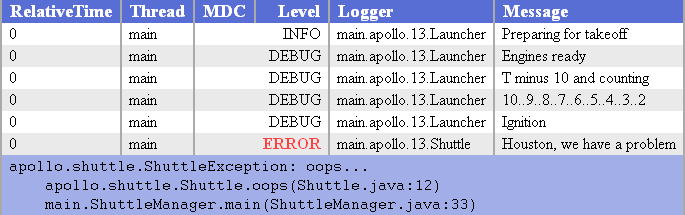
5758748 [main] ОШИБКА - может быть, не совсем ...

Читатель может захотеть взглянуть на другие реализации конвертера, такие как MDCConverter, чтобы узнать о более сложных режимах, таких как обработка опций. Для создания собственных цветовых схем взгляните на HighlightingCompositeConverter.

**HTMLLayout**

[HTMLLayout](http://logback.qos.ch/xref/ch/qos/logback/classic/html/HTMLLayout.html) (как включено в logback-classic) создает журналы в формате HTML. HTMLLayoutвыводит события регистрации в HTML-таблицу, где каждая строка таблицы соответствует событию регистрации.

Вот пример вывода, созданного с HTMLLayout использованием таблицы стилей CSS по умолчанию:



Содержимое столбцов таблицы указывается с помощью шаблона преобразования. Смотрите [PatternLayout](http://logback.qos.ch/manual/layouts.html" \l "ClassicPatternLayout)документацию по шаблонам конверсии. Таким образом, вы имеете полный контроль над содержимым и форматом таблицы. Вы можете выбрать и отобразить любую комбинацию преобразователей, PatternLayoutо которой известно.

Одно примечательное исключение в использовании PatternLayoutс HTMLLayout- это то, что спецификаторы преобразования не должны разделяться пробелами или, в более общем смысле, литеральным текстом. Каждый спецификатор, найденный в шаблоне, приведет к отдельному столбцу. Аналогичным образом будет создан отдельный столбец для каждого блока литерального текста, найденного в шаблоне, что может привести к бесполезной трате ценных ресурсов на ваш экран.

Вот простой, но функциональный файл конфигурации, иллюстрирующий использование HTMLLayout.

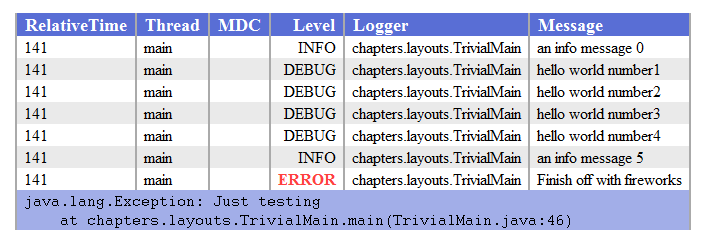
*Пример: Пример HTMLLayout (src / main / java / chapters / layouts / htmlLayoutConfig1.xml)*Просмотреть как .groovy

<configuration debug="true">  
  <appender name="FILE" class="ch.qos.logback.core.FileAppender">  
    <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">  
      <layout class="ch.qos.logback.classic.html.HTMLLayout">  
        **<pattern>%relative%thread%mdc%level%logger%msg</pattern>**  
      </layout>  
    </encoder>  
    <file>test.html</file>  
  </appender>  
  
  <root level="DEBUG">  
    <appender-ref ref="FILE" />  
  </root>  
</configuration>

Приложение [TrivialMain](http://logback.qos.ch/xref/chapters/layouts/TrivialMain.html) регистрирует несколько сообщений, заканчивающихся исключением. Команда:

java chapters.layouts.TrivialMain src / main / java / chapters / layouts / htmlLayoutConfig1.xml

создаст файл *test.html* в текущей папке. Содержимое *test.html* должно быть примерно таким:



**Следы стека**

Если вы используете слово преобразования *% em* для отображения трассировки стека, будет создан столбец таблицы для отображения трассировки стека. В большинстве случаев колонка будет пустой, тратя впустую экранную недвижимость. Кроме того, печать трассировки стека в отдельном столбце не дает очень удобочитаемых результатов. К счастью, слово преобразования *% ex* - не единственный способ отображения трассировки стека.

Лучшее решение доступно через реализации интерфейса IThrowableRenderer. Такая реализация может быть назначена HTMLLayout для управления отображаемыми данными, связанными с исключениями. По умолчанию каждому экземпляру HTMLLayout назначается DefaultThrowableRenderer. Он записывает исключения в новую строку таблицы вместе с трассировкой стека в легко читаемой форме, как показано на рисунке выше.

Если по какой-то причине вы все еще хотите использовать шаблон % ex, то вы можете указать NOPThrowableRenderer в файле конфигурации, чтобы отключить отображение отдельной строки для трассировки стека. Мы не имеем ни малейшего представления, почему вы хотели бы сделать это, но если бы вы хотели, вы могли бы.

**CSS**

Представление созданного HTML-кода HTMLLayout контролируется с помощью каскадной таблицы стилей (CSS). При отсутствии конкретных инструкций, по HTMLLayoutумолчанию будет использоваться его внутренний CSS. Тем не менее, вы можете указать HTMLLayoutвнешний CSS-файл. Для этой цели cssBuilderэлемент может быть вложен в <layout>элемент, как показано ниже.

<layout class="ch.qos.logback.classic.html.HTMLLayout">  
  <pattern>%relative...%msg</pattern>  
  <cssBuilder class="ch.qos.logback.classic.html.UrlCssBuilder">  
    <!-- url where the css file is located -->  
    <url>http://...</url>  
  </cssBuilder>   
</layout>

HTMLLayoutЧасто используется в сочетании с SMTPAppenderтак , что исходящая почта приятно отформатирован в HTML.

**Log4j XMLLayout**

[XMLLayout](http://logback.qos.ch/xref/ch/qos/logback/classic/log4j/XMLLayout.html) (часть logback-classic) генерирует выходные данные в формате, совместимом с log4j.dtd, для взаимодействия с такими инструментами, как [Chainsaw](http://logging.apache.org/chainsaw/index.html) и [Vigilog,](http://vigilog.sourceforge.net/) способными обрабатывать файлы, созданные [XMLLayout из log4j](http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/xml/XMLLayout.html) .

Как и исходный XMLLayout в log4j версии 1.2.15, XMLLayout в logback-classic принимает два логических свойства, locationInfo и properties . Если для locationInfo задано значение true, информация о местоположении (данные вызывающего абонента) включается в каждое событие. Установка свойств в true позволяет включить информацию MDC. Оба параметра по умолчанию имеют значение false.

Вот пример конфигурации

*Пример: Пример Log4jXMLLayout (src / main / java / chapters / layouts / log4jXMLLayout.xml)*

<configuration>  
  <appender name="FILE" class="ch.qos.logback.core.FileAppender">  
    <file>test.xml</file>  
    <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">  
      <layout class="ch.qos.logback.classic.log4j.XMLLayout">  
        <locationInfo>true</locationInfo>  
      </layout>  
    </encoder>   
  </appender>   
  
  <root level="DEBUG">  
    <appender-ref ref="FILE" />  
  </root>  
</configuration>

# Logback access

Большинство макетов с доступом к обратному входу являются простой адаптацией классических макетов с обратным доступом. Модули logback-classic и logback-access удовлетворяют различные потребности, но в целом предлагают сопоставимую функциональность.

**Написание собственного макета**

Написание пользовательского макета для доступа к обратному входу почти идентично написанию своего макета-брата в logback-classic.

.