**Глава 7: Фильтры**

В предыдущих главах было представлено [основное правило выбора](http://logback.qos.ch/manual/architecture.html#basic_selection) , которое лежит в основе logback-classic. В этой главе будут представлены дополнительные методы фильтрации.

Logback фильтры основаны на троичной логике, позволяющей собирать их или объединять в цепочки для создания сколь угодно сложной политики фильтрации. Они во многом вдохновлены iptables Linux.

Чтобы запустить примеры из этой главы, вам нужно убедиться, что в пути к классам присутствуют определенные файлы jar. Пожалуйста, обратитесь к [странице настройки](http://logback.qos.ch/setup.html) для получения дополнительной информации.

**В logback-classic**

Logback-classic предлагает два типа фильтров: обычные и турбо-фильтры.

**Обычные фильтры**

Регулярные классические фильтры logback расширяют абстрактный класс Filter, который по существу состоит из единственного метода solve (), принимающего экземпляр ILoggingEvent в качестве параметра.

Фильтры организованы в виде упорядоченного списка и основаны на троичной логике. decide(ILoggingEvent event)метод каждого фильтра вызывается последовательно. Этот метод возвращает один из [FilterReply](http://logback.qos.ch/xref/ch/qos/logback/core/spi/FilterReply.html) значений перечисления, т.е. один из: DENY \ NEUTRAL \ ACCEPT. Если значение, возвращаемое функцией decide(), равно DENY, то событие журнала немедленно удаляется, не обращаясь к оставшимся фильтрам. Если возвращаемое значение равно NEUTRAL, тогда используется следующий фильтр в списке. Если больше нет фильтров, к которым можно обратиться, тогда событие регистрации обрабатывается нормально. Если возвращаемое значение равно ACCEPT, тогда событие регистрации обрабатывается немедленно, пропуская вызов оставшихся фильтров.

В logback-classic фильтры могут быть добавлены к Appender экземплярам. Добавив один или несколько фильтров к приложению, вы можете фильтровать события по произвольным критериям, таким как содержимое сообщения журнала, содержимое MDC, время дня или любая другая часть события регистрации.

**Реализация собственного фильтра**

Создать свой собственный фильтр легко. Все, что вам нужно сделать, это расширить  абстрактный класс Filter и реализовать decide()метод.

Класс SampleFilter, показанный ниже, предоставляет пример. Его decide метод возвращает ACCEPT для регистрации событий, содержащих строку «sample» в своем поле сообщения. Для других событий возвращается значение NEUTRAL.

*Пример: базовый пользовательский фильтр ( [logback-examples / src / main / java / chapters / filters / SampleFilter.java](http://logback.qos.ch/xref/chapters/filters/SampleFilter.html) )*

package chapters.filters;  
  
import ch.qos.logback.classic.spi.ILoggingEvent;  
import ch.qos.logback.core.filter.Filter;  
import ch.qos.logback.core.spi.FilterReply;  
  
public class SampleFilter extends Filter<ILoggingEvent> {  
  
  @Override  
  public FilterReply decide(ILoggingEvent event) {      
    if (event.getMessage().contains("sample")) {  
      return FilterReply.ACCEPT;  
    } else {  
      return FilterReply.NEUTRAL;  
    }  
  }  
}

Файлы конфигурации, показанные ниже, присоединяют SampleFilter к ConsoleAppender.

*Пример: конфигурация SampleFilter (logback-examples / src / main / resources / chapters / filters / SampleFilterConfig.xml)*

<configuration>  
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
  
    **<filter class="chapters.filters.SampleFilter" />**  
  
    <encoder>  
      <pattern>  
        %-4relative [%thread] %-5level %logger - %msg%n  
      </pattern>  
    </encoder>  
  </appender>  
          
  <root>  
    <appender-ref ref="STDOUT" />  
  </root>  
</configuration>

С помощью Джорана, структура конфигурации logback, указание свойств или подкомпонентов для фильтров также легко. После добавления соответствующего метода установки в класс фильтра укажите значение свойства в элементе xml, названном в честь свойства, вложив его в <filter> элемент.

Часто требуемая логика фильтра состоит из двух ортогональных частей, теста на совпадение / несоответствие и ответа в зависимости от совпадения / несовпадения. Например, для данного теста, например, сообщение равно «foobar», один фильтр может ответить ACCEPT при совпадении и NEUTRAL при несовпадении, а другой фильтр может ответить NEUTRAL при совпадении и DENY при несовпадении.

Принимая во внимание эту ортогональность, logback поставляется с классом AbstractMatcherFilter, который предоставляет полезный каркас для указания соответствующего ответа при совпадении и при несовпадении, с помощью двух свойств, названных OnMatch и OnMismatch. Большинство регулярных фильтров, включенных в logback, являются производными от AbstractMatcherFilter.

**LevelFilter**

[LevelFilter](http://logback.qos.ch/xref/ch/qos/logback/classic/filter/LevelFilter.html) фильтрует события на основе точного соответствия уровня.

Если уровень события равен настроенному уровню, фильтр принимает или отклоняет событие в зависимости от конфигурации свойств onMatch и onMismatch . Вот пример файла конфигурации.

*Пример: Пример конфигурации LevelFilter (logback-examples / src / main / resources / chapters / filters / levelFilterConfig.xml)*

<configuration>  
  <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">  
    **<filter class="ch.qos.logback.classic.filter.LevelFilter">  
      <level>INFO</level>  
      <onMatch>ACCEPT</onMatch>  
      <onMismatch>DENY</onMismatch>  
    </filter>**  
    <encoder>  
      <pattern>  
        %-4relative [%thread] %-5level %logger{30} - %msg%n  
      </pattern>  
    </encoder>  
  </appender>  
  <root level="DEBUG">  
    <appender-ref ref="CONSOLE" />  
  </root>  
</configuration>

**ThresholdFilter**

ThresholdFilter отбрасывает события ниже указанного порога.

Для событий уровня, равного или превышающего пороговое значение, ThresholdFilter ответит НЕЙТРАЛЬНО, когда вызывается его метод decide (). Однако события с уровнем ниже порогового значения будут отклонены. Вот пример файла конфигурации.

*Пример: Пример конфигурации ThresholdFilter (logback-examples / src / main / resources / chapters / filters / thresholdFilterConfig.xml)*

<configuration>  
  <appender name="CONSOLE"  
    class="ch.qos.logback.core.ConsoleAppender">  
    <!-- deny all events with a level below INFO, that is TRACE and DEBUG -->  
    **<filter class="ch.qos.logback.classic.filter.ThresholdFilter">  
      <level>INFO</level>  
    </filter>**  
    <encoder>  
      <pattern>  
        %-4relative [%thread] %-5level %logger{30} - %msg%n  
      </pattern>  
    </encoder>  
  </appender>  
  <root level="DEBUG">  
    <appender-ref ref="CONSOLE" />  
  </root>  
</configuration>

**EvaluatorFilter**

EvaluatorFilter - это универсальный фильтр, инкапсулирующий EventEvaluator. Как следует из названия, EventEvaluator оценивает, удовлетворены ли заданные критерии для данного события. В случае совпадения и несоответствия хост-объект EvaluatorFilter возвращает значение, указанное в свойствах onMatch или onMismatch соответственно.

Обратите внимание, что EventEvaluator это абстрактный класс. Вы можете реализовать свою собственную логику оценки событий с помощью подклассов EventEvaluator.

**GEventEvaluator**

[GEventEvaluator](http://logback.qos.ch/xref/ch/qos/logback/classic/boolex/GEventEvaluator.html) - это конкретная [EventEvaluator](http://logback.qos.ch/xref/ch/qos/logback/core/boolex/EventEvaluator.html) реализация, использующая произвольные логические выражения на языке Groovy в качестве критериев оценки. Мы называем такие выражения логического языка Groovy как «выражения оценки groovy». Groovy оценочные выражения обеспечивают беспрецедентную гибкость в фильтрации событий. GEventEvaluator требует Groovy runtime. Пожалуйста, обратитесь к [соответствующему разделу](http://logback.qos.ch/setup.html#groovy) документа установки о добавлении среды исполнения Groovy в путь к классам.

Оценочные выражения составляются на лету во время интерпретации файла конфигурации. Как пользователь, вам не нужно беспокоиться о реальной сантехники. Тем не менее, вы несете ответственность за правильность выражения на языке Groovy.

Выражение оценки действует для текущего события регистрации. Logback автоматически вставляет текущее событие регистрации типа [ILoggingEvent](http://logback.qos.ch/apidocs/ch/qos/logback/classic/spi/ILoggingEvent.html) как переменную, называемую « *событием* », а также его сокращение, называемое « *е* ». Переменные TRACE, DEBUG, INFO, WARN и ERROR также экспортируются в область выражения. Таким образом, "event.level == DEBUG" и "e.level == DEBUG" являются эквивалентными и допустимыми выражениями groovy, которые будут возвращаться trueтолько в том случае, если уровень текущего события регистрации идентичен уровню DEBUG. Для других операторов сравнения на уровнях поле уровня должно быть преобразовано в целое число с toInt()оператором.

Вот более полный пример.

<configuration>  
      
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
    **<filter class="ch.qos.logback.core.filter.EvaluatorFilter">        
      <evaluator class="ch.qos.logback.classic.boolex.GEventEvaluator">   
        <expression>  
           e.level.toInt() >= WARN.toInt() &amp;&amp;  <!-- Stands for && in XML -->  
           !(e.mdc?.get("req.userAgent") =~ /Googlebot|msnbot|Yahoo/ )  
        </expression>  
      </evaluator>  
      <OnMismatch>DENY</OnMismatch>  
      <OnMatch>NEUTRAL</OnMatch>  
    </filter>**  
    <encoder>  
      <pattern>  
        %-4relative [%thread] %-5level %logger - %msg%n  
      </pattern>  
    </encoder>  
  </appender>  
  
  <root level="DEBUG">  
    <appender-ref ref="STDOUT" />  
  </root>  
</configuration>

Приведенный выше фильтр пропустит события уровня WARN и выше на консоль, если ошибка не генерируется веб-сканерами, связанными с Google, MSN или Yahoo. Это делается путем проверки, содержит ли MDC, связанный с событием, значение для «req.userAgent», совпадающее с /Googlebot|msnbot|Yahoo/регулярным выражением. Обратите внимание, что поскольку карта MDC может быть нулевой, мы также используем [оператор безопасной разыменования](http://groovy.codehaus.org/Null+Object+Pattern) Groovy , то есть?. оператор. Эквивалентная логика была бы намного длиннее, если бы она была выражена в Java.

Если вам интересно, как идентификатор пользовательского агента был вставлен в MDC под ключом 'req.userAgent', следует упомянуть, что logback поставляется с фильтром сервлета, названным [MDCInsertingServletFilter](http://logback.qos.ch/manual/mdc.html" \l "mis) специально для этой цели. Это будет описано в следующей главе.

**JaninoEventEvaluator**

Logback-classic поставляется с другой конкретной EventEvaluator реализацией, называемой [JaninoEventEvaluator, которая](http://logback.qos.ch/xref/ch/qos/logback/classic/boolex/JaninoEventEvaluator.html) **принимает произвольный блок языка Java, возвращая логическое значение в качестве критерия оценки**. Мы называем такие булевские выражения языка Java « *оценочными выражениями* ». Выражения оценки обеспечивают большую гибкость в фильтрации событий.

JaninoEventEvaluator требует [библиотеки Janino](http://docs.codehaus.org/display/JANINO/Home) . Пожалуйста, смотрите [соответствующий раздел](http://logback.qos.ch/setup.html#janino) документа настройки. По сравнению с JaninoEventEvaluator, GEventEvaluator благодаря языку Groovy, он значительно удобнее в использовании, но JaninoEventEvaluator обычно будет работать (намного) быстрее для эквивалентных выражений.

Оценочные выражения составляются на лету во время интерпретации файла конфигурации. Как пользователь, вам не нужно беспокоиться о реальной сантехники. Однако вы обязаны убедиться, что выражение языка Java возвращает логическое значение, т. е. что оно имеет значение true или false.

Выражение оценки оценивается по текущему событию регистрации. Logback-classic автоматически экспортирует различные поля события регистрации в виде переменных, доступных из выражения оценки. Чувствительные к регистру имена этих экспортируемых переменных перечислены ниже.

|  |  |  |
| --- | --- | --- |
| **имя** | **Тип** | **Описание** |
| event | LoggingEvent | Необработанное logging событие, связанное с logging запросом. Все следующие переменные также доступны из события. Например, event.getMessage() возвращает то же значение String, что и переменная *сообщения,* описанная далее. |
| message | String | Необработанное сообщение logging запроса. Для некоторого логгера **l**, когда вы пишете l.info ("Hello {}", name); где имени присваивается значение «Алиса», тогда сообщение - это«Hello {}». |
| formattedMessage | String | Отформатированное сообщение в logging запросе. Для некоторого логгера **l** , когда вы пишете l.info ("Hello {}", name); где имени присваивается значение «Алиса», тогда форматированное сообщение - это«Hello Алиса». |
| logger | String | Название регистратора. |
| loggerContext | [LoggerContextVO](http://logback.qos.ch/xref/ch/qos/logback/classic/spi/LoggerContextVO.html) | Ограниченное представление (объект-значение) контекста средства ведения журнала, которому принадлежит событие ведения журнала. |
| level | int | Значение int, соответствующее уровню. Для облегчения создания выражений, включающих уровни, также доступны значения по умолчанию *DEBUG* , *INFO* , *WARN* и *ERROR* . Таким образом, использование *уровня> INFO* является правильным выражением. |
| timeStamp | long | Отметка времени, соответствующая созданию события регистрации. |
| marker | Marker | MarkerОбъект , связанный с запросом регистрации. Обратите внимание, что маркер может быть нулевым, и вы должны проверить это условие, чтобы избежать NullPointerException. |
| MDC | Map | Карта, содержащая все значения MDC на момент создания события регистрации. Доступ к значению можно получить с помощью следующего выражения: ***mdc.get ("myKey")*** . Начиная с классической версии 0.9.30, переменная 'mdc' никогда не будет нулевой.  java.util.Map тип непараметризированный потому Janino не поддерживает генерики. Отсюда следует, что тип, возвращаемый функцией mdc.get (), это Object, а не String. Чтобы вызвать String методы для возвращаемого значения, оно должно быть приведено как String. Например, ((String) mdc.get("k")).contains("val"). |
| Throwable | java.lang.Throwable | Если с событием не связано исключение, тогда значение переменной throwable будет нулевым. К сожалению, «throwable» не выдерживает сериализации. Таким образом, в удаленных системах его значение всегда будет нулевым. Для выражений, не зависящих от местоположения, используйте throwableProxy переменную, описанную ниже. |
| throwableProxy | [IThrowableProxy](http://logback.qos.ch/xref/ch/qos/logback/classic/spi/IThrowableProxy.html) | Прокси для исключения, связанного с событием регистрации. Если с событием не связано исключение, тогда значение переменной throwableProxy будет нулевым. В отличие от «throwable», когда исключение связано с событием, значение «throwableProxy» будет ненулевым даже в удаленных системах, то есть даже после сериализации. |

Вот конкретный пример.

*Пример: базовое использование оценщика событий (logback-examples / src / main / resources / chapters / filters / basicEventEvaluator.xml)*

<configuration>  
  
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
    **<filter class="ch.qos.logback.core.filter.EvaluatorFilter">        
      <evaluator> <!-- defaults to type ch.qos.logback.classic.boolex.JaninoEventEvaluator -->  
        <expression>return message.contains("billing");</expression>  
      </evaluator>  
      <OnMismatch>NEUTRAL</OnMismatch>  
      <OnMatch>DENY</OnMatch>  
    </filter>**  
    <encoder>  
      <pattern>  
        %-4relative [%thread] %-5level %logger - %msg%n  
      </pattern>  
    </encoder>  
  </appender>  
  
  <root level="INFO">  
    <appender-ref ref="STDOUT" />  
  </root>  
</configuration>

Жирная часть в приведенном выше файле конфигурации добавляет EvaluatorFilter к ConsoleAppender. Оценщик типа JaninoEventEvaluator затем вводится в EvaluatorFilter. При отсутствии атрибута *класса* в <evaluator>элементе, указанном пользователем, Джоран выведет тип по умолчанию JaninoEventEvaluator для оценщика. Это один из [немногих случаев,](http://logback.qos.ch/manual/onJoran.html#defaultClassMapping) когда Джоран неявно определяет тип компонента.

Элемент *выражения* соответствует только что обсужденному оценочному выражению. Выражение “return message.contains("billing");” возвращает логическое значение. Обратите внимание, что переменная *message* автоматически экспортируется JaninoEventEvaluator.

Учитывая, что для свойства OnMismatch установлено значение NEUTRAL, а для свойства OnMatch установлено значение DENY, этот фильтр оценщика удалит все события регистрации, сообщение которых содержит строку "billing".

Приложение FilterEvents выдает десять logging запросов, пронумерованных от 0 до 9. Давайте сначала запустим класс FilterEvents без каких-либо фильтров:

java chapters.filters.FilterEvents src / main / java / chapters / filters / basicConfiguration.xml

Все запросы будут отображаться, как показано ниже:

0 [main] INFO chapters.filters.FilterEvents - logging statement 0

0 [main] INFO chapters.filters.FilterEvents - logging statement 1

0 [main] INFO chapters.filters.FilterEvents - logging statement 2

0 [main] DEBUG chapters.filters.FilterEvents - logging statement 3

0 [main] INFO chapters.filters.FilterEvents - logging statement 4

0 [main] INFO chapters.filters.FilterEvents - logging statement 5

0 [main] ERROR chapters.filters.FilterEvents - **billing statement 6**

0 [main] INFO chapters.filters.FilterEvents - logging statement 7

0 [main] INFO chapters.filters.FilterEvents - logging statement 8

0 [main] INFO chapters.filters.FilterEvents - logging statement 9

Предположим, что мы хотим избавиться от "billing statement". Перечисленный выше файл конфигурации *basicEventEvaluator.xml* фильтрует сообщения, содержащие строку «billing», которая является именно желаемым результатом.

*Работает* с *basicEventEvaluator.xml* :

java chapters.filters.FilterEvents src / main / java / chapters / filters / basicEventEvaluator.xml

мы получаем:

0 [main] INFO chapters.filters.FilterEvents - logging statement 0

0 [main] INFO chapters.filters.FilterEvents - logging statement 1

0 [main] INFO chapters.filters.FilterEvents - logging statement 2

0 [main] DEBUG chapters.filters.FilterEvents - logging statement 3

0 [main] INFO chapters.filters.FilterEvents - logging statement 4

0 [main] INFO chapters.filters.FilterEvents - logging statement 5

0 [main] INFO chapters.filters.FilterEvents - logging statement 7

0 [main] INFO chapters.filters.FilterEvents - logging statement 8

0 [main] INFO chapters.filters.FilterEvents - logging statement 9

Выражения оценки могут быть блоками Java. Например, следующее является допустимым выражением.

<evaluator>  
  <expression>  
    if(logger.startsWith("org.apache.http"))  
      return true;  
  
    if(mdc == null || mdc.get("entity") == null)  
      return false;  
  
    String payee = (String) mdc.get("entity");  
  
    if(logger.equals("org.apache.http.wire") &amp;&amp; <!-- & encoded as &amp; -->  
        payee.contains("someSpecialValue") &amp;&amp;  
        !message.contains("someSecret")) {  
      return true;  
    }  
  
    return false;  
  </expression>  
</evaluator>

**Matchers**

Хотя сопоставление с образцом можно выполнить, вызвав метод [match ()](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html#matches%28java.lang.String%29) в String классе, это требует затрат на компиляцию совершенно нового Pattern объекта при каждом вызове фильтра. Чтобы устранить эти издержки, вы можете заранее определить один или несколько объектов [Matcher](http://logback.qos.ch/xref/ch/qos/logback/core/boolex/Matcher.html) . Как только определитель определен, на него можно неоднократно ссылаться по имени в выражении оценщика.

Пример должен прояснить вопрос:

*Пример: определение совпадений в оценщике событий (logback-examples / src / main / resources / chapters / filters*/valuatorWithMatcher.xml*)*

<configuration debug="true">  
  
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
    <filter class="ch.qos.logback.core.filter.EvaluatorFilter">  
      <evaluator>          
        **<matcher>  
          <Name>ooo</Name>  
          <!-- filter out ooo numbered statements -->  
          <regex>statement [13579]</regex> //удалит сообщения со “statement 1 \ 3\5\7\9”  
        </matcher>  
          
        <expression>ooo.matches(formattedMessage)</expression>**  
      </evaluator>  
      <OnMismatch>NEUTRAL</OnMismatch>  
      <OnMatch>DENY</OnMatch>  
    </filter>  
    <encoder>  
      <pattern>%-4relative [%thread] %-5level %logger - %msg%n</pattern>  
    </encoder>  
  </appender>  
  
  <root level="DEBUG">  
    <appender-ref ref="STDOUT" />  
  </root>  
</configuration>

Выполнение с *evaluatorWithMatcher.xml* :

java chapters.filters.FilterEvents src / main / java / chapters / filters /valuatorWithMatcher.xml

мы получаем:

260 [main] INFO chapters.filters.FilterEvents - logging statement 0

264 [main] INFO chapters.filters.FilterEvents - logging statement 2

264 [main] INFO chapters.filters.FilterEvents - logging statement 4

266 [main] ERROR chapters.filters.FilterEvents - billing statement 6

266 [main] INFO chapters.filters.FilterEvents - logging statement 8

Если вам нужно определить дополнительные соответствия, вы можете сделать это, добавив дополнительные <matcher>элементы.

**TurboFilters**

Все объекты TurboFilter расширяют абстрактный класс TurboFilter. Как и обычные фильтры, они используют троичную логику, чтобы вернуть свою оценку события регистрации.

В целом они работают так же, как ранее упомянутые фильтры. Тем не менее, есть два основных отличия Filter и TurboFilter объектов.

TurboFilter объекты привязаны к контексту регистрации. Следовательно, они вызываются не только при использовании данного приложения, но и каждый раз, когда выдается запрос на регистрацию. Их область применения шире, чем у прикрепленных к аппендеру фильтров.

Более того, они вызываются до создания объекта LoggingEvent. TurboFilter объекты не требуют создания экземпляра события регистрации, чтобы отфильтровать запрос регистрации. Таким образом, турбо-фильтры предназначены для высокопроизводительной фильтрации событий регистрации еще до того, как события будут созданы.

**Реализация собственного TurboFilter**

Чтобы создать свой собственный TurboFilter компонент, просто расширьте абстрактный класс TurboFilter. Как и ранее, при реализации настраиваемого объекта фильтра при разработке настраиваемого объекта TurboFilter запрашивается только тот, который реализует decide()метод. В следующем примере мы создаем немного более сложный фильтр:

*Пример: базовый пользовательский TurboFilter( [logback-examples / src / main / java / chapters / filters / SampleTurboFilter.java](http://logback.qos.ch/xref/chapters/filters/SampleTurboFilter.html) )*

package chapters.filters;  
  
import org.slf4j.Marker;  
import org.slf4j.MarkerFactory;  
  
import ch.qos.logback.classic.Level;  
import ch.qos.logback.classic.Logger;  
import ch.qos.logback.classic.turbo.TurboFilter;  
import ch.qos.logback.core.spi.FilterReply;  
  
public class SampleTurboFilter extends TurboFilter {  
  
  String marker;  
  Marker markerToAccept;  
  
  @Override  
  public FilterReply decide(Marker marker, Logger logger, Level level,  
      String format, Object[] params, Throwable t) {  
  
    if (!isStarted()) {  
      return FilterReply.NEUTRAL;  
    }  
  
    if ((markerToAccept.equals(marker))) {  
      return FilterReply.ACCEPT;  
    } else {  
      return FilterReply.NEUTRAL;  
    }  
  }  
  
  public String getMarker() {  
    return marker;  
  }  
  
  public void setMarker(String markerStr) {  
    this.marker = markerStr;  
  }  
  
  @Override  
  public void start() {  
    if (marker != null && marker.trim().length() > 0) {  
      markerToAccept = MarkerFactory.getMarker(marker);  
      super.start();   
    }  
  }  
}

TurboFilter выше принимает события , которые содержат определенный маркер. Если указанный маркер не найден, то фильтр передает ответственность следующему фильтру в цепочке.

Чтобы обеспечить большую гибкость, маркер, который будет проверен, может быть указан в файле конфигурации, следовательно, методы getter и setter. Мы также реализовали start() метод, чтобы убедиться, что опция была указана в процессе настройки.

Вот пример конфигурации, которая использует наш недавно созданный TurboFilter.

*Пример: базовая пользовательская TurboFilterконфигурация (logback-examples / src / main / resources / chapters / filters / sampleTurboFilterConfig.xml)*

<configuration>  
  **<turboFilter class="chapters.filters.SampleTurboFilter">  
    <Marker>sample</Marker>  
  </turboFilter>**  
  
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
    <encoder>  
      <pattern>  
        %-4relative [%thread] %-5level %logger - %msg%n  
      </pattern>  
    </encoder>  
  </appender>  
  
  <root>  
    <appender-ref ref="STDOUT" />  
  </root>  
</configuration>

Logback classic корабли с несколькими TurboFilter классами готовы к использованию. В [MDCFilter](http://logback.qos.ch/xref/ch/qos/logback/classic/turbo/MDCFilter.html) проверяет наличие заданного значения в МОМ , тогда как [DynamicThresholdFilter](http://logback.qos.ch/apidocs/ch/qos/logback/classic/turbo/DynamicThresholdFilter.html) позволяет фильтрации на основе порогового MDC ассоциаций ключа / уровня. С другой стороны, [MarkerFilter](http://logback.qos.ch/xref/ch/qos/logback/classic/turbo/MarkerFilter.html) проверяет наличие определенного маркера, связанного с запросом на регистрацию.

Вот пример конфигурации с использованием обоих MDCFilterи MarkerFilter.

*Пример: MDCFilterи MarkerFilter конфигурация (logback-examples / src / main / resources / chapters / filters / turboFilters.xml)*

<configuration>  
  
  <turboFilter class="ch.qos.logback.classic.turbo.MDCFilter">  
    <MDCKey>username</MDCKey>  
    <Value>sebastien</Value>  
    <OnMatch>ACCEPT</OnMatch>  
  </turboFilter>  
          
  <turboFilter class="ch.qos.logback.classic.turbo.MarkerFilter">  
    <Marker>billing</Marker>  
    <OnMatch>DENY</OnMatch>  
  </turboFilter>  
  
  <appender name="console" class="ch.qos.logback.core.ConsoleAppender">  
    <encoder>  
      <pattern>%date [%thread] %-5level %logger - %msg%n</pattern>  
    </encoder>  
  </appender>  
  
  <root level="INFO">  
    <appender-ref ref="console" />  
  </root>    
</configuration>

Вы можете увидеть эту конфигурацию в действии, выполнив следующую команду:

java chapters.filters.FilterEvents src / main / java / chapters / filters / turboFilters.xml

Как мы видели ранее, [FilterEvents](http://logback.qos.ch/xref/chapters/filters/FilterEvents.html) приложение выдает 10 запросов на регистрацию, пронумерованных от 0 до 9. За исключением запросов 3 и 6, все запросы имеют уровень *INFO* , того же уровня, что и тот, который назначен корневому регистратору. Третий запрос выдается на уровне *DEBUG* , который ниже эффективного уровня. Однако, поскольку ключ MDC «username» установлен в «sebastien» непосредственно перед 3-м запросом и удален сразу после него, он MDCFilter принимает этот запрос (и только этот запрос). Шестой запрос, выданный на уровне *ОШИБКА* , помечается как «биллинг». Как таковой, он запрещен MarkerFilter (второй турбо-фильтр в конфигурации).

Таким образом, вывод FilterEventsприложения, настроенного с *помощью* файла *turboFilters.xml,* показанного выше:

2006-12-04 15:17:22,859 [main] INFO chapters.filters.FilterEvents - logging statement 0

2006-12-04 15:17:22,875 [main] INFO chapters.filters.FilterEvents - logging statement 1

2006-12-04 15:17:22,875 [main] INFO chapters.filters.FilterEvents - logging statement 2

2006-12-04 15:17:22,875 [main] DEBUG chapters.filters.FilterEvents - logging statement 3

2006-12-04 15:17:22,875 [main] INFO chapters.filters.FilterEvents - logging statement 4

2006-12-04 15:17:22,875 [main] INFO chapters.filters.FilterEvents - logging statement 5

2006-12-04 15:17:22,875 [main] INFO chapters.filters.FilterEvents - logging statement 7

2006-12-04 15:17:22,875 [main] INFO chapters.filters.FilterEvents - logging statement 8

2006-12-04 15:17:22,875 [main] INFO chapters.filters.FilterEvents - logging statement 9

Видно, что третий запрос, который не должен отображаться, если мы следовали только за общим уровнем *INFO* , все равно появляется, потому что он соответствует первым TurboFilter требованиям и был принят.

С другой стороны, должен отображаться 6-й запрос, то есть запрос уровня *ERROR* . Но он удовлетворил второго, TurboFilter чья опция OnMatch установлена ​​в *DENY* . Таким образом, 6-й запрос не отображался.

**DuplicateMessageFilter**

DuplicateMessageFilter заслуживает отдельной презентации. Этот фильтр обнаруживает повторяющиеся сообщения и, после определенного количества повторений, удаляет повторные сообщения.

Чтобы обнаружить повторение, этот фильтр использует простое равенство строк между сообщениями. Он не обнаруживает сообщения, которые очень похожи, меняются только на несколько символов. Например, если вы напишите:

logger.debug("Hello "+name0);  
logger.debug("Hello "+name1);

Предполагая, что name0 и name1 имеют разные значения, два сообщения «Hello» будут считаться несвязанными. В зависимости от требований пользователя будущие выпуски могут проверять сходство строк, устраняя повторения похожих, но не идентичных сообщений.

Обратите внимание, что в случае параметризованной регистрации принимается во внимание только необработанное сообщение. Например, в следующих двух запросах необработанные сообщения, то есть «Hello {}.», являются идентичными и, таким образом, рассматриваются как повторения.

logger.debug("Hello {}.", name0);  
logger.debug("Hello {}.", name1);

Количество разрешенных повторений может быть указано свойством AllowedRepetitions. Например, если для свойства установлено значение 1, второе и последующие вхождения одного и того же сообщения будут удалены. Аналогично, если для свойства установлено значение 2, то 3-е и последующие вхождения одного и того же сообщения будут отброшены. По умолчанию для свойства AllowedRepetitions установлено значение 5.

Для обнаружения повторений этот фильтр должен хранить ссылки на старые сообщения во внутреннем кэше. Размер этого кэша определяется свойством CacheSize . По умолчанию это значение равно 100.

*Пример: DuplicateMessageFilter конфигурация (logback-examples / src / main / resources / chapters / filters / duplicateMessage.xml)*

<configuration>  
  
  **<turboFilter class="ch.qos.logback.classic.turbo.DuplicateMessageFilter"/>**  
  
  <appender name="console" class="ch.qos.logback.core.ConsoleAppender">  
    <encoder>  
      <pattern>%date [%thread] %-5level %logger - %msg%n</pattern>  
    </encoder>  
  </appender>  
  
  <root level="INFO">  
    <appender-ref ref="console" />  
  </root>    
</configuration>

Таким образом, вывод для FilterEventsприложения, настроенного с *duplicateMessage.xml* :

2008-12-19 15:04:26,156 [main] INFO chapters.filters.FilterEvents - logging statement 0

2008-12-19 15:04:26,156 [main] INFO chapters.filters.FilterEvents - logging statement 1

2008-12-19 15:04:26,156 [main] INFO chapters.filters.FilterEvents - logging statement 2

2008-12-19 15:04:26,156 [main] INFO chapters.filters.FilterEvents - logging statement 4

2008-12-19 15:04:26,156 [main] INFO chapters.filters.FilterEvents - logging statement 5

2008-12-19 15:04:26,171 [main] ERROR chapters.filters.FilterEvents - billing statement 6

"logging statement 0" является первым *появлением* сообщения «logging statement {}». «logging statement 1» - это первое *повторение* , logging statement 2 - это второе повторение. Интересно, что «logging statement 3» уровня DEBUG является *третьим* повторением, хотя впоследствии он отбрасывается в силу [основного правила выбора](http://logback.qos.ch/manual/architecture.html#basic_selection) . Это можно объяснить тем фактом, что турбо-фильтры вызываются раньше других типов фильтров, включая основное правило выбора. Таким образом,DuplicateMessageFilter рассматривает «logging statement 3» как повторение, не обращая внимания на тот факт, что он будет добавлен вниз в цепочке обработки. «logging statement 4» является четвертым повторением, а «logging statement 5» - пятым. Утверждения 6 и более поздние отбрасываются, потому что только 5 повторений разрешены по умолчанию.

# In logback-access

Logback-access предлагает большинство функций, доступных в logback-classic. В частности, Filter объекты доступны и работают так же, как их классические аналоги logback, с одним заметным отличием. Вместо LoggingEvent экземпляров фильтры обратного доступа действуют на [AccessEvent](http://logback.qos.ch/xref/ch/qos/logback/access/spi/AccessEvent.html) экземпляры. В настоящее время логбэк-доступ поставляется с ограниченным количеством фильтров, описанных ниже. Если вы хотите предложить дополнительные фильтры, пожалуйста, свяжитесь со списком рассылки logback-dev.

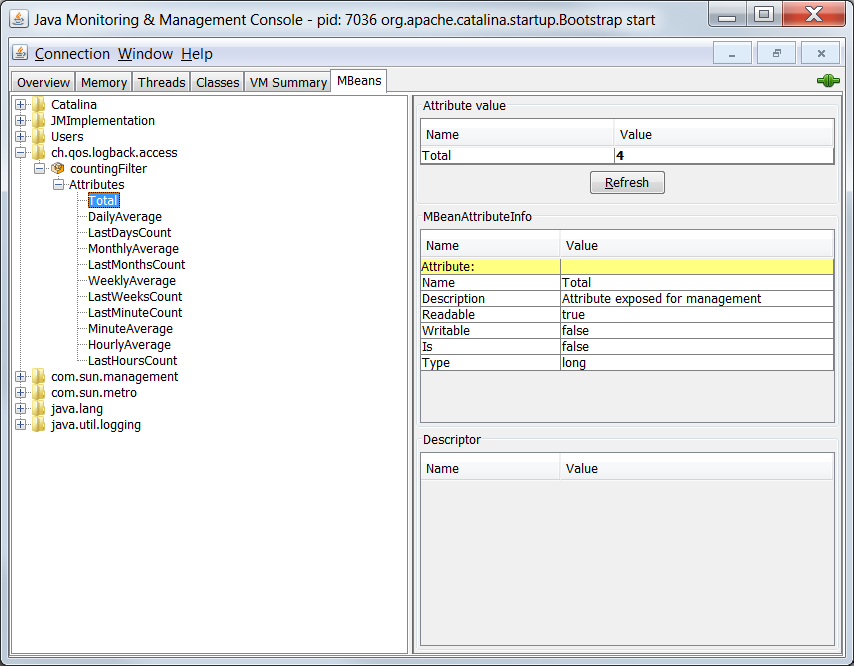
**CountingFilter**

С помощью [CountingFilter](http://logback.qos.ch/manual/xref/ch/qos/logback/access/filter/CountingFilter.html) класса logback-access можно предоставить статистические данные о доступе к веб-серверу. После инициализации CountingFilter регистрируется как MBean на сервере JMX платформы. Затем вы можете запросить этот MBean для получения статистических данных, например, средних значений по минутам, часам, дням, неделям или месяцам. Другие статистические данные, такие как количество за предыдущую неделю, день, час или месяц, а также общее количество также доступны.

Следующий *файл* конфигурации *logback-access.xml* объявляет CountingFilter.

<configuration>  
  <statusListener class="ch.qos.logback.core.status.OnConsoleStatusListener" />  
  
  **<filter class="ch.qos.logback.access.filter.CountingFilter">  
    <name>countingFilter</name>  
  </filter>**  
  
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
    <encoder>  
      <pattern>%h %l %u %t \"%r\" %s %b</pattern>  
    </encoder>  
  </appender>  
  
  <appender-ref ref="STDOUT" />  
</configuration>

Вы можете просмотреть различные статистические данные CountingFilter, хранящиеся на сервере JMX вашей платформы через jconsoleприложение.



**EvaluatorFilter**

[EvaluatorFilter](http://logback.qos.ch/xref/ch/qos/logback/core/filter/EvaluatorFilter.html) это общий фильтр инкапсулирования EventEvaluator. Как следует из названия, оценка оценивает, удовлетворяются ли заданные критерии для данного события. При совпадении и при несовпадении хостинг будет возвращать значение, указанное в свойствах onMatch или onMismatch соответственно. Обратите внимание, что ранее обсуждалось в контексте logback-classic ( [см. Выше](http://logback.qos.ch/manual/filters.html" \l "evalutatorFilter) ). Настоящий текст является в основном повторением предыдущего обсуждения.[EventEvaluator](http://logback.qos.ch/xref/ch/qos/logback/core/boolex/EventEvaluator.html)EvaluatorFilterEvaluatorFilter

Обратите внимание, что EventEvaluatorэто абстрактный класс. Вы можете реализовать свою собственную логику оценки событий с помощью подклассов EventEvaluator. Доступ к обратному доступу поставляется с конкретной реализацией [JaninoEventEvaluator](http://logback.qos.ch/xref/ch/qos/logback/access/boolex/JaninoEventEvaluator.html) . В качестве критериев оценки используются произвольные логические выражения языка Java. Мы называем такие блоки языка Java « *оценочными выражениями* ». Выражения оценки обеспечивают большую гибкость в фильтрации событий. JaninoEventEvaluatorтребует [библиотеки Янино](http://docs.codehaus.org/display/JANINO/Home) . Пожалуйста, смотрите [соответствующий раздел](http://logback.qos.ch/setup.html#janino) документа настройки.

Оценочные выражения составляются на лету во время интерпретации файла конфигурации. Как пользователь, вам не нужно беспокоиться о реальной сантехники. Однако вы обязаны убедиться, что выражение языка Java возвращает логическое значение, т. Е. Что оно имеет значение true или false.

Выражение оценки оценивается для текущего события доступа. Logback-access автоматически экспортирует текущий AccessEventэкземпляр под именем переменной **event**. Вы можете прочитать различные данные, связанные с запросом HTTP, а также ответ HTTP через eventпеременную. Пожалуйста, обратитесь к [AccessEvent исходному коду класса](http://logback.qos.ch/xref/ch/qos/logback/access/spi/AccessEvent.html) для точного списка.

Следующий файл конфигурации logback-access иллюстрирует фильтрацию по коду ответа HTTP [404 (Not Found)](http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.4.5) . Каждый запрос в результате 404 будет напечатан на консоли.

*Пример: Access Evaluator (logback-examples / src / main / resources / chapters / filters / accessEventEvaluator.xml)*

<configuration>  
  <statusListener class="ch.qos.logback.core.status.OnConsoleStatusListener" />  
  
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
    **<filter class="ch.qos.logback.core.filter.EvaluatorFilter">  
      <evaluator>  
        <expression>event.getStatusCode() == 404</expression>  
      </evaluator>  
      <onMismatch>DENY</onMismatch>  
    </filter>**  
   <encoder><pattern>%h %l %u %t %r %s %b</pattern></encoder>  
  </appender>  
  
  <appender-ref ref="STDOUT" />  
</configuration>

В следующем примере мы по-прежнему регистрируем запросы, которые приводят к 404 ошибкам, кроме тех запросов, которые запрашивают файлы CSS.

*Пример 6.10: Access Evaluator (logback-examples / src / main / resources / chapters / filters / accessEventEvaluator2.xml)*

<configuration>  
  <statusListener class="ch.qos.logback.core.status.OnConsoleStatusListener" />  
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
    <filter class="ch.qos.logback.core.filter.EvaluatorFilter">  
      <evaluator name="Eval404">  
        <expression>  
         **(event.getStatusCode() == 404)**  
           **&amp;&amp;**  <!-- ampersand characters need to be escaped -->  
         **!(event.getRequestURI().contains(".css"))**  
        </expression>  
      </evaluator>  
      <onMismatch>DENY</onMismatch>  
    </filter>  
  
   <encoder><pattern>%h %l %u %t %r %s %b</pattern></encoder>  
  </appender>  
  
  <appender-ref ref="STDOUT" />  
</configuration>