**카테고리 보관물: logback**

**https://sonegy.wordpress.com/category/logback/**

[**logging framework logback**](https://sonegy.wordpress.com/2014/06/03/235/)

**logback이란**

logback은 SLF4J의 native 구현체 입니다. slf4j로 어플리케이션 로그를 남긴다면 logback을 선택하는게 가장 좋습니다. slf4j의 도움으로 연관 라이브러리들이 다른 logging framework를 쓰더라도 logback으로 통합할 수 있습니다.

logback 은 logback-core, logback-classic, logback-access의 3개의 모듈이 있습니다. core는 classic과 access의 공통라이브러리입니다. maven repository를 쓴다면 classic만 추가하면 관련 라이브러리가 추가 됩니다.

Maven pom.xml

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <dependencies>      <dependency>          <groupId>ch.qos.logback</groupId>          <artifactId>logback-classic</artifactId>          <version>1.1.2</version>      </dependency>  </dependencies> |

External Libraries  
Maven: ch.qos.logback:logback-classic:1.1.2  
Maven: ch.qos.logback:logback-core:1.1.2  
Maven: org.slf4j:slf4j-api:1.7.6

**전격 실행**

logback을 사용할수 있는 준비가 끝났습니다. 아래 코드를 바로 실행해봅시다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import ch.qos.logback.classic.LoggerContext;  import ch.qos.logback.core.util.StatusPrinter;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;    public class Tutorial {      private static final Logger logger = LoggerFactory.getLogger(Tutorial.class);        public static void main(String[] args) {          logger.trace("trace");          logger.debug("debug");          logger.info("info");          logger.warn("warn");          logger.error("error");      }  } |

**Logger, LoggerFactory**

Logger와 LoggerFactory는 SLF4J에 있는 interface와 implements입니다. 실제 어플리케이션상에 logger를 정의 할때 logback 관련 코드를 넣지는 않습니다. 다시 말해 logback과 dependency가 없는 코드를 구현하게 되면 차후 다른 logging framework로 교체하는게 가능하게 됩니다.

|  |  |
| --- | --- |
| 1 | Logger logger = LoggerFactory.getLogger(Tutorial.class); |

20:57:59.346 [main] DEBUG example.logback.Tutorial – debug  
20:57:59.350 [main] INFO example.logback.Tutorial – info  
20:57:59.350 [main] WARN example.logback.Tutorial – warn  
20:57:59.350 [main] ERROR example.logback.Tutorial – error

**logback.groovy, logback-test.xml, logback.xml**

logback.groovy, logback-test.xml, logback.xml resource가 없다면 디폴트 설정 이라는 정보를 출력합니다. 현재 로그상태를 알수 있습니다. 만약 모든 설졍이 root classpath에 등록이 되어 있다면. 우선순위 전략을 따랍니다.

1. logback.groovy 파일을 먼저 찾습니다.
2. 없다면 logback-test.xml 을 찾습니다.
3. 그래도 없다면 logback.xml을 찾습니다.
4. 모두 없다면 기본 설정 전략을 따릅니다. BasicConfiguration

maven 프로젝틀르 사용하는 경우에는 test/resources에 logback-test.xml을 두고 테스트가 실행될때는 logback-test.xml이 우선적용되도록 사용합니다. 어때요. 참 쉽죠?

**좀더 멋진 maven dependency 설정**

표준은 좋은것이죠. 사실 로그에 대한 표준이 없지만, SLF4J를 사용하게 된다면 다양한 logging framework을 선택적으로 사용할수 있습니다. SLF4J에만 코드가 의존적이 된다면 아주 좋죠. 그래서 사실 java compile에서는 logback이 필요 없습니다. 아래 pom.xml 설정에서 runtime 로 logback-classic 을 설정하면 logback-core도 같은 조건이 됩니다.

pom.xml

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | <dependencies>      <dependency>          <groupId>org.slf4j</groupId>          <artifactId>slf4j-api</artifactId>          <version>1.7.7</version>      </dependency>      <dependency>          <groupId>ch.qos.logback</groupId>          <artifactId>logback-classic</artifactId>          <version>1.1.2</version>          <exclusions>              <exclusion>                  <groupId>org.slf4j</groupId>                  <artifactId>slf4j-api</artifactId>              </exclusion>          </exclusions>          <scope>runtime</scope>       </dependency>  </dependencies> |

**Logger**

**Name**

Logger들은 이름 기반으로 생성이 됩니다. LoggerFactory.getLogger(“NAME”)로 Logger를 호출하면 “NAME”에 대한 딱하나를 instance를 반환합니다. 여러번 호출해도 같은 객체입니다. String 대신 .class로 클래스 정보를 넘겨주면 .getName()으로 클래스 이름을 사용하게 됩니다. 흔히 Class객체를 넘겨주어 결국 이름은 packageName + ClassName으로 구성이 됩니다. 흔히 Logger는 특정 패키지 이하로 재한을 두고 정의를 합니다.

그럼으로 우리가 만드는 어플리케이션에서의 Logger들은 . 으로 구분된 Hierarchy가 생기게 되는것 입니다. 일종의 상속 구조가 됩니다.코드로 예를 들어 봅시다.

|  |  |
| --- | --- |
| 1  2  3 | example.logback.level.Grandparents  example.logback.level.grandparents.Parents  example.logback.level.grandparents.parents.Children |

위처럼 3개의 Class Grandparents, Parents, Children를 위와 같은 구조로 생성을 하고 아래처럼 Logger를 class마다 생성하면 3개의 Logger가 full package+className으로 생성이 됩니다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | private static final Logger logger = LoggerFactory.getLogger(Grandparents.class);  private static final Logger logger = LoggerFactory.getLogger(Parents.class);  private static final Logger logger = LoggerFactory.getLogger(Children.class);    ...  // 각각 클래스마다 구현  public void run() {      logger.trace("trace");      logger.debug("debug");      logger.info("info");      logger.warn("warn");      logger.error("error");  } |

Logger들은 Tree Hierarchy 구조로 level을 적용 받을수 있습니다.

logback.xml 설정

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | <?xml version="1.0" encoding="UTF-8"?>  <configuration>      <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">          <layout class="ch.qos.logback.classic.PatternLayout">              <Pattern>%d{HH:mm} %-5level %logger{36} - %msg%n</Pattern>          </layout>      </appender>        <logger name="example.logback.level.grandparents" level="TRACE"/>      <logger name="example.logback.level.grandparents.parents.children" level="INFO"/>      <root level="DEBUG">          <appender-ref ref="STDOUT" />      </root>  </configuration> |

여기서 root는 root logger를 말합니다. logger name=””과 같다고 생각하면 됩니다.

|  |  |
| --- | --- |
| 1 | <logger name="example.logback.level.grandparents" level="TRACE"/> |

example.logback.level.grandparents 이하 모든 Logger들의 level은 TRACE라는 설정입니다.

**Level**

1. trace
2. debug
3. info
4. warn
5. error

위처럼 5단계의 level이 존재합니다. TRACE라고 선언을 하면 trace,debug,info,warn,error 모든 Level을 포함합니다. 그리고 INFO로 설정된다면 info.warn,error이 포함 됩니다.

|  |  |
| --- | --- |
| 1  2  3  4  5 | public static void main(String[] args) {      new Grandparents().run();      new Parents().run();      new Children().run();  } |

20:23 TRACE e.l.level.grandparents.Grandparents – trace  
20:23 DEBUG e.l.level.grandparents.Grandparents – debug  
20:23 INFO e.l.level.grandparents.Grandparents – info  
20:23 WARN e.l.level.grandparents.Grandparents – warn  
20:23 ERROR e.l.level.grandparents.Grandparents – error  
20:23 TRACE e.l.l.grandparents.parents.Parents – trace  
20:23 DEBUG e.l.l.grandparents.parents.Parents – debug  
20:23 INFO e.l.l.grandparents.parents.Parents – info  
20:23 WARN e.l.l.grandparents.parents.Parents – warn  
20:23 ERROR e.l.l.grandparents.parents.Parents – error  
20:23 INFO e.l.l.g.parents.children.Children – info  
20:23 WARN e.l.l.g.parents.children.Children – warn  
20:23 ERROR e.l.l.g.parents.children.Children – error

Logger Name의 Level설정은 Name마다 상속 적용됩니다. grandparents의 TRACE설정은 이하 모두 적용이 되고, children 이하 INFO로 재지정 됩니다.

**Appender**

**Event마다 Log를 기록하는 기능**은 Appender가 처리합니다. 그래서 Logger는 어떤 appender에 해당이 되어 처리 되는게 중요합니다. Appender를 설정하더래도 log출력에 해당되지 않으면 작동하지 않습니다.  
Appender는 출력될 형식을 직접 가지고 있지 않고, 해당 기능은 Layout과 Encoder에 위임을 합니다.

**ConsoleAppender**

ConsoleAppender 는 OutputStreamAppender를 상속합니다. **encoder, pattern**으로 PatternLayoutEncoder가 생성해서 Appender에 주입됩니다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">    <!-- encoders are assigned the type         ch.qos.logback.classic.encoder.PatternLayoutEncoder by default -->    <encoder>      <pattern>%-4relative [%thread] %-5level %logger{35} - %msg %n</pattern>    </encoder>  </appender> |

PatternLayoutEncoder는 pattern으로 받은 값을 이용해서 PatternLayout을 생성하고 PatternLayoutEncoder는 log message를 byte[]로 변환하는 기능도 포함합니다. 이로써 Appender는 Layout기능과 Encoder기능을 모두 가지게 됩니다. 이것을 이용해서 OutputStreamAppender는 byte[]를 OuputStream에 write하게 됩니다.  
<http://logback.qos.ch/manual/layouts.html#ClassicPatternLayout>

**FileAppender**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | <!-- Insert the current time formatted as "yyyyMMdd'T'HHmmss" under       the key "bySecond" into the logger context. This value will be       available to all subsequent configuration elements. -->  <timestamp key="bySecond" datePattern="yyyyMMdd'T'HHmmss"/>  <appender name="FILE" class="ch.qos.logback.core.FileAppender">    <file>log-${bySecond}.txt</file>    <append>true</append>    <!-- encoders are assigned the type         ch.qos.logback.classic.encoder.PatternLayoutEncoder by default -->    <encoder>      <pattern>%-4relative [%thread] %-5level %logger{35} - %msg%n</pattern>    </encoder>  </appender> |

로그가 저장될 file을 선언하고, ConsoleAppender처럼 encoder,pattern을 선언하게 되면, log event를 지정된 file에 저장할 수 있습니다. 이때 파일 포맷중 날짜형식은 java.text.SimpleDateFormat을 따릅니다.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <appender name="FILE" class="ch.qos.logback.core.FileAppender">  ...    <prudent>true</prudent>  ...  </appender> |

옵션중 prudent를 선언할수 있는데, file에 저장될때 lock을 생성해서 처리합니다. 서로 다른 java vm이 같은 파일을 가리킬때 사용합니다. 당연히 file lock이니 성능 저하 있습니다.

**RollingFileAppender**

log 가 많아지면 file 하나당 최대 용량 제한도 있고, 로그를 파악하기도 어렵습니다. 이때는 대부분 날짜 기준으로 file을 남깁니다. 따로 crontab으로 매일 file을 rename해서 처리할 수도 있지만, logback은 RollingFileAppender로 처리할 수 있습니다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | <appender name="ROLLING" class="ch.qos.logback.core.rolling.RollingFileAppender">    <file>mylog.txt</file>    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">      <!-- rollover daily -->      <fileNamePattern>mylog-%d{yyyy-MM-dd}.%i.txt</fileNamePattern>      <timeBasedFileNamingAndTriggeringPolicy            class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">        <!-- or whenever the file size reaches 100MB -->        <maxFileSize>100MB</maxFileSize>      </timeBasedFileNamingAndTriggeringPolicy>    </rollingPolicy>    <encoder>      <pattern>%msg%n</pattern>    </encoder>  </appender> |

rollingPolicy로 rolling 정책을 등록할 수 있습니다. fileNamePattern으로 file pattern을 선언하고, timeBasedFileNamingAndTriggeringPolicy로 파일마다 트리거를 걸어 파일 최대 용량을 설정하면 새로운 index이름으로 파일을 생성합니다.

FileSize는 끝문자열에 kb, mb, gb를 인식하고 대소문자 구분은 없습니다. whitespace는 모두 ignore처리합니다.

|  |  |
| --- | --- |
| 1 | <fileNamePattern>mylog-%d{yyyy-MM-dd}.%i.txt.zip</fileNamePattern> |

확장자에 .zip을 선언하면 새로운 file 이 생성될때 이전 파일은 .zip으로 압축을 할수 있습니다.

**EventEvaluator**

어느 프로젝트에서 SMTPAppender를 사용해서 에러가 발생하면 관리자에서 에러 리포팅을 하는 기능을 추가했습니다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <appender name="EMAIL" class="ch.qos.logback.classic.net.SMTPAppender">      <smtpHost>\*\*\*\*</smtpHost>      <smtpPort>25</smtpPort>      <to>\*\*\*\*</to>      <to>\*\*\*\*</to>      <from>\*\*\*\*</from>      <subject>PRODUCT-PCS: %logger{20} - %m</subject>      <layout class="ch.qos.logback.classic.PatternLayout">          <Pattern>%d{HH:mm} %-5level %logger{36} - %msg%n</Pattern>      </layout>  </appender> |

하지만 새벽에 장애가 발생하였고, 정상적으로 에러 리포팅을 했습니다. 다음 날 아침.  
메일은 11만통이 전송 되었습니다. 어느 메일서버가 사망했다는 전설이 내려옵니다.

간단하게 메일 전송이 너무 많을때 걸러주는 evaluator를 만들어 넣었습니다. 5분이내에 발송이 없었을때만 메일을 전송합니다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class TimeBasedEventEvaluator extends ContextAwareBase implements EventEvaluator {      private long beforeTime = 0;      @Setter      private long intervalTime = 1000 \* 60;        @Override      public boolean evaluate(Object event) throws NullPointerException, EvaluationException {          long current = System.currentTimeMillis();          long backupBeforeTime = this.beforeTime;            if (current - backupBeforeTime > intervalTime) {              this.beforeTime = current;              return true;          }          return false;      }  } |
| 1  2  3 | <appender name="EMAIL" class="ch.qos.logback.classic.net.SMTPAppender">      <evaluator class="xxxx.evaluator.TimeBasedEventEvaluator" />  ... |

**layout**

언급했던 layout pattern에 대해 정리합니다.

1. %logger{length}
   * Logger name의 이름을 축약할 수 있습니다. {length}는 최대 차릿수 입니다.
2. %thread
   * 현재 Thread name
3. %-5level
   * log level -5는 출력 고정폭 값
4. %msg
   * log message %message은 alias
5. %n
   * new line

자세한 설명은 아래를 참조합니다.  
<http://logback.qos.ch/manual/layouts.html>

**Configuration**

끝으로 logback.xml설정에서 빠진 부분을 적어봅니다.

**logback debug 모드**

아래 설정으로 logback이 구동될때 logback 상태를 확인할 수 있습니다.

|  |  |
| --- | --- |
| 1  2  3 | <configuration debug="true">  ...  </configuration> |

**additivity**

additivity의 default값은 true입니다. logger name이하 모두 적용이 되는데 additivity를 false로 설정하면 해당 name에만 logger가 적용됩니다.

|  |  |
| --- | --- |
| 1 | <logger name="XXX" level="DEBUG" additivity="false"/> |

**root**

logger들은 name으로 등록이 됩니다. 기본적으로 java package구조와 동일하게 적용하는데요. tree 구조이기때문에 최상단 root을 적용하면 모든 tree이하에 적용할 수 있습니다. 이대 라는 설정으로 logger를 대신합니다.

|  |  |
| --- | --- |
| 1  2  3 | <root level="DEBUG">      <appender-ref ref="STDOUT" />  </root> |

모든 대상에 STDOUT Appender를 적용하고 level이 DEBUG이하인것만 처리로 설정됩니다.

이상 끝.

관련소스는 아래 첨부합니다.  
<https://github.com/sonegy/how-to-use-logback>