Обзор библиотек логирования

Юрий Литвинов
yurii.litvinov@gmail.com

16.03.2017г

Логирование

- Отладочный вывод — дешёвая альтернатива отладке
  - Иногда быстрее вставить отладочную печать, чем проходить отладчиком
  - Иногда отладчик недоступен или бесполезен
    - Многопоточные и распределённые приложения
    - Встроенные системы
- Post-mortem-анализ
  - “Отладочный вывод” должен работать и на развёрнутой системе
  - И выводить не в консоль
  - И обеспечивать информацию о контексте
- Проблемы:
  - Scrolling blindness
  - Замедление приложения
- Примерно 4% кода типичных проектов связано с логированием

Apache Log4j 2, основные понятия

Logger — штука, которая может что-то куда-то выводить (на самом деле, производить логирующие события)

LoggerConfig — управляет поведением логгера

LogManager — создаёт, хранит и выдаёт по запросу логгеры

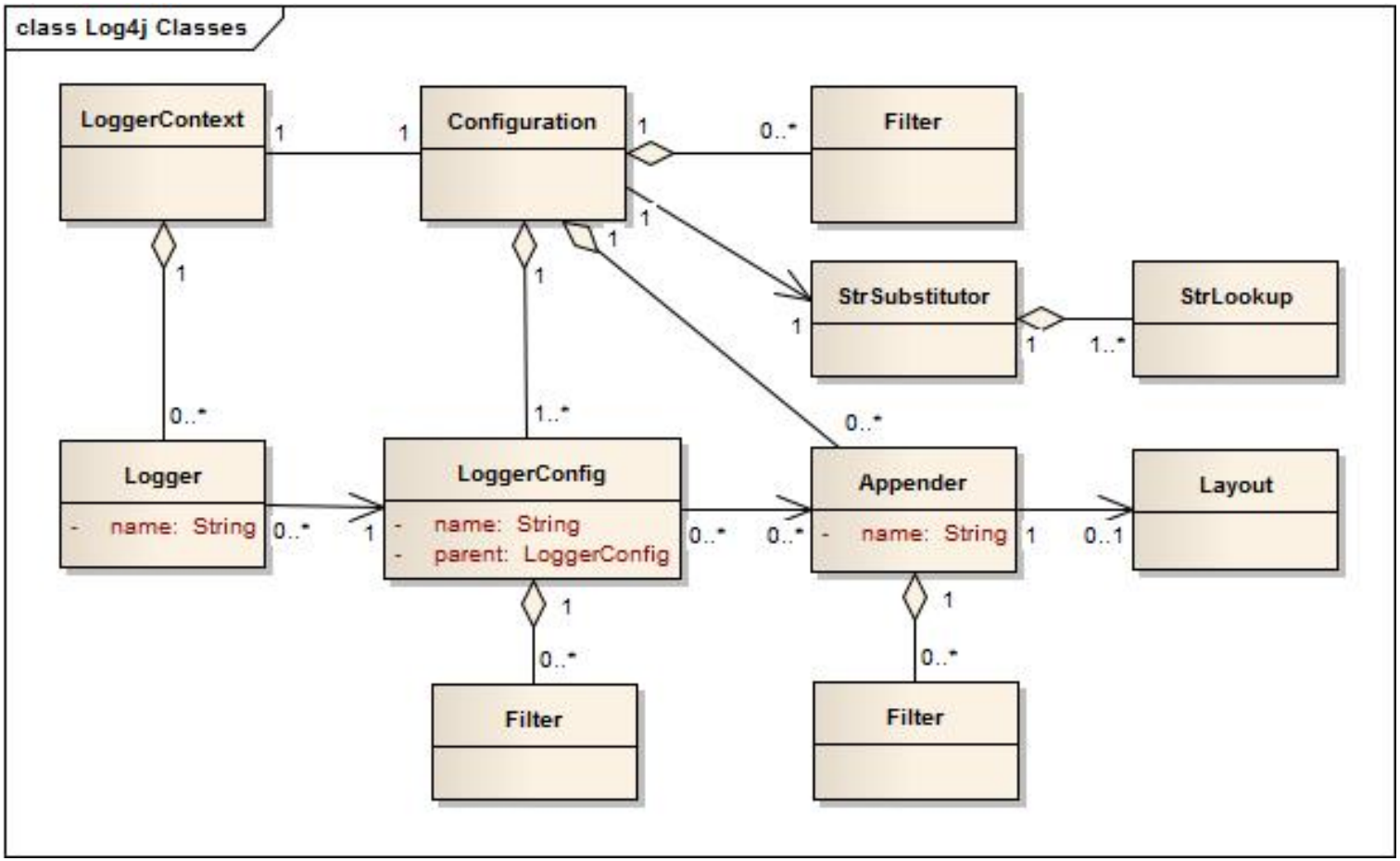
Filter — фильтрует логирующие события, говоря, надо или не надо их куда-то выводить

Appender — на самом деле выводит информацию куда-то (в файл, на консоль, в системный лог и т.д.)

Layout — говорит, в каком формате и какую информацию о событии следует выводить

Вся конфигурация — иерархическая!

Архитектура



Пример

```java
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class HelloWorld {
    private static final Logger logger
            = LogManager.getLogger("HelloWorld");
    public static void main(String[] args) {
        logger.info("Hello, World!");
    }
}
```

Пример конфигурации

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration monitorInterval="30">
 <Appenders>
  <Console name="Console" target="SYSTEM_OUT">
   <PatternLayout pattern=
     "%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
  </Console>
 </Appenders>
 <Loggers>
  <Root level="error">
   <AppenderRef ref="Console"/>
  </Root>
 </Loggers>
</Configuration>
```

Куда это писать

Log4j ищет конфигурации в следующих местах в следующем порядке:

- Системное свойство “log4j.configurationFile” (указывается при запуске опцией -D)
- log4j2-test.properties в classpath
- log4j2-test.yaml или log4j2-test.yml в classpath
- log4j2-test.json или log4j2-test.jsn в classpath
- log4j2-test.xml в classpath
- log4j2.properties в classpath
- log4j2.yaml или log4j2.yml в classpath
- log4j2.json или log4j2.jsn в classpath
- log4j2.xml в classpath
- Иначе используется DefaultConfiguration, которая выводит на консоль

Уровни и маркеры

Уровни логирования: **TRACE**, **DEBUG**, **INFO**, **WARN**, **ERROR**, **FATAL**, **OFF**

Маркеры — способ тонкой настройки информации, которую хочется выводить. Пример:

```
private static final Marker SQL_MARKER
        = MarkerManager.getMarker("SQL");

private static final Marker QUERY_MARKER
        = MarkerManager.getMarker("SQL_QUERY")
                .setParents(SQL_MARKER);
...
logger.debug(QUERY_MARKER, "SELECT * FROM {}", table);
```

Синтаксис конфигурации (1)

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
  <Properties>
    <Property name="name1">value</property>
    <Property name="name2" value="value2"/>
  </Properties>
  <Filter type="type" ... />
  <Appenders>
    <Appender type="type" name="name">
      <Filter type="type" ... />
    </Appender>
    ...
  </Appenders>
```

Синтаксис конфигурации (2)

```
<Loggers>
  <Logger name="name1">
    <Filter type="type" ... />
  </Logger>
  ...
  <Root level="level">
    <AppenderRef ref="name"/>
  </Root>
</Loggers>
</Configuration>
```

Appenders

Console — выводит в SYSTEM_OUT или SYSTEM_ERR

File — выводит в указанный файл

RollingFile — выводит в указанный файл, создавая новые файлы и удаляя старые при необходимости

- TriggeringPolicy — когда переходить к следующему файлу и что-то делать с предыдущими
  - При запуске, по времени, по размеру, по дате/часу
- RolloverStrategy — что делать с файлами
  - По шаблону (хитро), с указанием максимума хранимых файлов, кого удалять, сжатие логов

Ещё штук 20

Пример конфигурации

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration name="MyApp">
 <Appenders>
  <RollingFile name="RollingFile"
    fileName="logs/app.log"
    filePattern=
     "logs/$${date:yyyy-MM}/app-%d{MM-dd-yyyy}-%i.log.gz">
   <PatternLayout>
    <Pattern>%d %p %c{1.} [%t] %m%n</Pattern>
   </PatternLayout>
   <Policies>
    <TimeBasedTriggeringPolicy />
    <SizeBasedTriggeringPolicy size="250 MB"/>
   </Policies>
  </RollingFile>
 </Appenders>
 ...
</Configuration>
```

Patterns

c/logger	Имя логгера
C/class	Имя класса, который вывел сообщение
d/date	Дата и время
p/level	Уровень логирующего события (TRACE, INFO, ...)
t/thread	Имя потока, в котором произошло событие
m/message	Собственно, сообщение из программы
n	Перевод строки
marker	Полное имя маркера
L/line	Строка, где вызвали логгер
highlight	Штука, позволяющая управлять цветом вывода

Как это выглядит в коде

Неправильно:

```
if (logger.isDebugEnabled()) {
    logger.debug("Logging in user " + user.getName()
            + " with birthday " + user.getBirthdayCalendar());
}
```

Правильно:

```
logger.debug("Logging in user {} with birthday {}"
        , user.getName(), user.getBirthdayCalendar());
```

Длительные операции

Неправильно:

```
if (logger.isTraceEnabled()) {
    logger.trace("Some long-running operation returned {}",
            expensiveOperation());
}
```

Правильно:

```
logger.trace("Some long-running operation returned {}",
        () -> expensiveOperation());
```

Flow Tracing

```
public void setMessages(String[] messages) {
    logger.traceEntry(new JsonMessage(messages));
    this.messages = messages;
    logger.traceExit();
}

public String retrieveMessage() {
    logger.entry();
    String testMsg = getMessage(getKey());
    return logger.exit(testMsg);
}
```

ThreadContext

ThreadContext — Map со значениями, локальными для потока или для контекста, которые можно использовать в логах:

```
ThreadContext.put("id", UUID.randomUUID().toString());
ThreadContext.put("ipAddress", request.getRemoteAddr());
...
logger.debug("Message 1");
...
ThreadContext.clear();
```

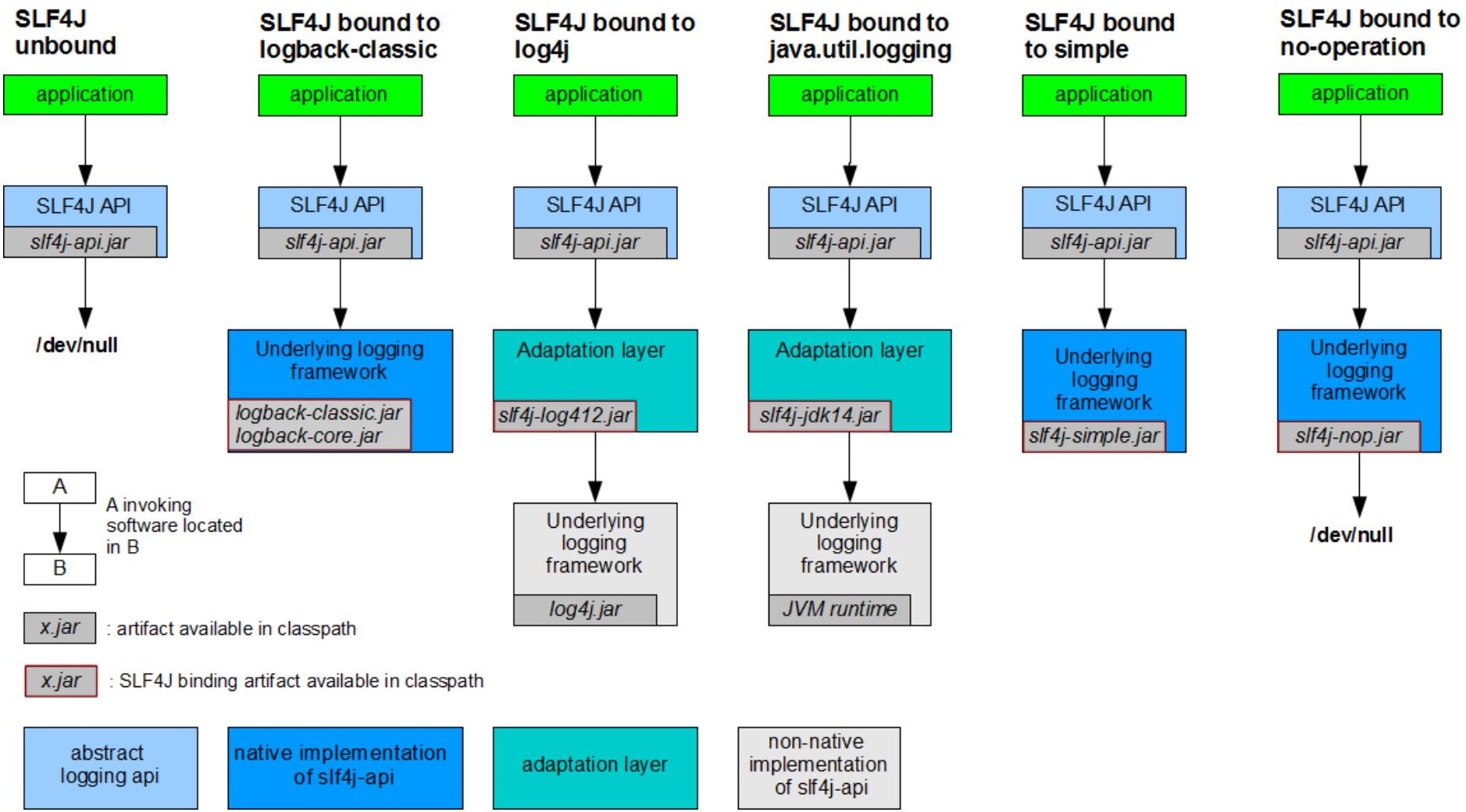
Шаблон **%X** включает в лог всё, **%X{key}** — только значение с заданным ключом

SLF4J

Simple Logging Facade for Java

- Фасад (на самом деле, прокси) для библиотек логирования
- Нужен, чтобы код не зависел от конкретной библиотеки логирования, а зависел только от легковесного фасада
- Фасад, в свою очередь, использует ту библиотеку, которую нашёл в CLASSPATH при запуске
- Работает очень быстро и позволяет не навязывать лишних зависимостей
  - Особенно полезно в библиотечном коде
  - Спасает от ситуации, когда есть несколько компонентов, каждый из которых хочет свою библиотеку логирования

Архитектура



Пример

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Wombat {
  private final Logger logger = LoggerFactory.getLogger(Wombat.class);
  private Integer t;
  private Integer oldT;

  public void setTemperature(Integer temperature) {
    oldT = t;
    t = temperature;

    logger.debug("Temperature set to {}. Old temperature was {}.", t, oldT);

    if(temperature.intValue() > 50) {
      logger.info("Temperature has risen above 50 degrees.");
    }
  }
}
```

SLF4J

- Умеет многое из того, что умеют “настоящие” библиотеки, так что можно просто выводить в лог, не задумываясь об API
  - Формирование строк через {}
  - Маркеры
- Чтобы всё работало, надо подключить:
  - slf4j-api — обязательно, и одно из:
  - slf4j-simple — бэкенд “из коробки”, умеет выводить в System.err
  - log4j-slf4j-impl — для использования Log4J в качестве бэкенда
- Не забываем конфигурационный файл Log4J, если используем как бэкенд его