Последнее изменение: 22 июня 2010г.

Ведение лога приложения

I like to log it, log it!

Суровая правда жизни

Если бы мы жили в мире, где обитают сферические программисты в вакууме – этой статьи бы не было. Ибо речь у нас пойдет о таком явлении, как ведение журнала действий приложения. В просторечии – лога [1] .

Для чего нужен лог? К сожалению, у несферических программистов приложений без ошибок не бывает. И чем сложнее приложение – тем больше в нем потенциально содержится ошибок. В общем-то это аксиома, но напомнить нелишне. И, что более важно, – чем сложнее приложение, тем изощреннее могут быть ошибки в нем. В том смысле, что произошедшая в данную секунду ошибка может быть вызвана событиями, произошедшими час назад. А то и день назад. Или же вообще – ошибок вроде как и нет, но система все равно ведет себя неадкеватно – на нажатие каждой кнопки реагирует секунд по 8-10, хотя еще два часа назад время реакции не превышало полсекунды.

А когда система в боевых условиях *регулярно* падает (совсем, насмерть!) без объявления войны, а в тестовых условиях такое поведение не воспроизводится никоим образом – хочется кого-нибудь убить. Чаще всего себя, ибо именно мне и предстоит решать эту проблему. Причем вчера.

Собственно, что объединяет все описаные случаи – во всех них сильно помогла бы информация о происходящих в системе событиях. Причем информация за определенный промежуток времени. Понимание этого факта и породила такое явление как ведение лога (журнала, протокола, – называйте как хотите) действий приложения.

Вот, что мы будем обсуждать:

Итак, приступим.

Типы логов и требования к ним

Принципиально логи нужны не только для отслеживания *программых* ошибок – точно так же можно отслеживать и ошибки (а то и целенаправленные действия) пользователей. Формально протокол действий приложения можно разделить на несколько частей. Если говорить об очень общем делении, то таких частей будет две – *что происходит в приложения с точки зрения бизнес-пользователя* и *что происходит в приложение с точки зрения разработчика (программиста)*. Если смотреть в несколько другой плоскости – можно поделить все действия на *системные (SYSTEM)* , *безопасности (SECURITY)* и *приложения (APPLICATION, BUSINESS)*. Так, например, классифицируются системные события в Windows семейства NT.

> Поясню на примерах. Пользователь входит в приложение, проверяется пароль. Это действие относится к безопасности. Дальше он запускает какой-нибудь модуль приложения, например, работу с заявками на кредиты. Это действие приложения (BUSINESS). Модуль при старте обращается к другому модулю за какими-то дополнительными данными, производит какие-либо еще телодвижения – это уже системные действия.

Из всего перечисленного нас – разработчиков – интересует прежде всего то, что происходит в системе с точки зрения именно разработчика. Во всяком случае обсуждать мы будем только это. Что же касается классификации происходящих событий – *SYSTEM/SECURITY/APPLICATION* – разработчикам приходится иметь дело со всеми ними. Ниже мы рассмотрим, как эти логи можно разделять для удобства анализа.

Дальше. Простой вопрос – а сколько информации нам нужно? Такой же простой ответ – чем больше, тем лучше! – не годится. Ибо чем больше мы выводим информации о происходящем в системе – тем больше процессорного времени тратится на эти бесполезные с точки зрения конечного пользователя действия, и тем меньше остается собственно на работу. У меня в практике был случай, когда интенсивный вывод в лог – ошиблись в конфигурации – замедлил работу системы более чем в 50 раз. Вот тут на самом деле есть противоречие – с точки зрения пользователя логов должно быть в минимальном объеме, а с точки зрения разработчика – в максимальном. Ну и если мечтать – идеальной была бы ситуация, когда количество выводимых данных можно было бы менять как нужно, и в тот момент, когда нужно.

Следующее очевидное требование – вывод только той информации, которая нам нужна, и в том виде, в котором она нам нужна. Т.е. необходима возможность управлять форматом вывода – что именно выводить, мы и так контролируем. Причем крайне желательно, чтобы такая настройка была отделена от приложения – чтобы ее можно было менять, не трогая программного кода.

Не будь требования об универсальности, все было бы намного проще. Думаю, вы сами прекрасно понимаете, что идея ведения лога не нова. И, соответственно, реализована она во многих имеющихся операционных системах. На всякий случай не говорю "во всех", хотя уверенность в этом почти стопроцентная. Правда, реализована эта идея сильно по-разному. В *NIX это сервис *syslog*, в Windows семейства NT – *NT Event Viewer*. В других системах есть свои механизмы. И далеко не всегда результат их работы удобен для использования.

В общем, как ни крути – необходима унифицированная, гибко настраиваемая система для ведения лога. Эта система должна быть отделена от приложения, конфигурироваться отдельно от него, в идеале – вообще на лету. И такие системы есть. Правда, с ними иногда возникает путаница, потому в следующей часты мы рассмотрим, что есть что.

Унифицированные системы логирования

В настоящий момент лично я знаю три[2] фреймворка для ведения лога – библиотека *Log4J*, пакет *java.util.logging* в JavaSE и библиотека *Logback*. Помимо них мы рассмотрим два, я бы так выразился, "зонтичных" фреймворка – *Apache Commons Logging* и *Simple Logging Facade for Java*. Эти фреймворки являются прослойками между системой логирования и самим приложением.

Итак, начнем по хронологии.

Log4J

Фреймворк *Apache Log4J* появился первым из всех рассматриваемых. Если мне не изменяет память, до версии 1.2.8 он был совместим с Java 1.1. Изначально этот фреймворк был хорошо архитектурно проработан, потому он быстро завоевал популярность. Возможно, определенную роль сыграл также факт, что других просто не было.

Далее мы *рассмотрим* этот фреймворк в деталях, потому более подробно описывать его сейчас не буду. Хочу только сказать, что этот фреймворк очень популярен и по сей день.

java.util.logging

Пакет *java.util.logging* появился в JavaSE в версии 1.4, в 2001 году. К этому моменту уже существовал *Log4J*, судя по всему, от него и отталкивались. Однако, как это очень часто бывало у Sun, перемудрили. Возможностей этот фреймворк давал меньше, чем *Log4J*, в использовании, по крайней мере на мой взгляд, был менее интуитивным. Тем не менее, у *java.util.logging* было большое преимущество – он был частью JavaSE. Не надо тащить с собой отдельную библиотеку, а иногда это критично.

Соответственно, в какой-то момент сложилась не очень приятная ситуация. В части приложений используется *Log4J*, в части – пакет *java.util.logging*. И все бы ничего, но возникает вопрос: а что делать разработчикам *библиотек*? Им тоже нужно использовать логирование. На какую из систем ориентироваться? API этих фреймворков различается, связывание происходит на этапе компиляции, просто так фреймворк логирования не заменить. Делать версию библиотеки под каждый? Это безумие.

Выход был найден достаточно простой. Раз у нас используются два разных фреймворка для одной цели – их надо унифицировать. Т.е. написать прослойку (фактически, адаптер), которая будет скрывать от разработчика, какой реально фреймворк он использует. Он будет звать прослойку, а она – конкретный фреймворк. Так появился *Apache Commons Logging*.

Apache Commons Logging

Фреймворк *Apache Commons Logging* предназначен для абстрагирования разработчика от конкретного фреймворка логирования. Он предоставляет некоторый унифицированный интерфейс, транслируя его вызовы в использование конкретных возможностей фреймворков. Ключевым тут является слово "*унифицированный*". Оно означает минимальное доступное покрытие, т.е. отсутствие, например, специфических возможностей *Log4J*. А они бывают ОЧЕНЬ полезны.

Commons Logging абстрагирует следующие фреймворки:*Log4J*, *java.util.logging*, *Avalon LogKit*, *Lumberjack* (см.*примечание 2*). Кроме того, тот же унифицированный интерфейс реализован в самом фреймворке в двух вариантах – т.н. *NOP*[3]-вариант, и простейший вывод в *System.err*.

Появившись, *Commons Logging* стал спасением для разработчиков библиотек общего назначания. В результате этого он используется необычайно широко. Однако у него есть ряд существенных недостатков, с которыми приходилось мириться.

1. Фреймворк является минимальным – не поддерживает очень полезных специфических возможностей абстрагируемых фреймворков.
2. Фреймворк никак не занимается инициализацией и конфигурированием конкретных фреймворков логирования – это остается на самом разработчике. Но это все равно лучше, чем было до него – при смене фреймворка логирования нужно изменить только код инициализации.
3. Самое неприятное – при некоторых условиях[4] у этого фреймворка есть проблемы с загрузчиком классов. В одном проекте нам пришлось отказаться от него и перейти на чистый *Log4J* именно по этой причине.

Если с первыми двумя неудобствами еще можно было как-то жить, то третье осложняло жизнь очень серьезно. В немалой степени из-за этого и появился следующий фреймворк – *SLF4J*.

Simple Logging Facade for Java

По своей сути *SLF4J* является тем же, чем и *Commons Logging* – абстрагирующим фреймворком логирования. Однако у него есть ряд важных отличий:

1. *SLF4J* является более продвинутым, нежели *Commons Logging* – набор поддерживаемых им возможностей конечных фреймворков шире. Если где-то конкретный конечный фреймфорк не поддерживает каких-либо возможностей – делается их иммитация.
2. *SLF4J* поддерживает (абстрагирует) большее количество фреймворков логирования, чем *Commons Logging* – это *java.util.logging*, *Log4J*, *Commons Logging*, *Logback* (он вообще является реализацией интерфейсов самого *SLF4J*). Есть две собственные реализации – *NOP* и *Simple*.
3. *SLF4J* является *приемником* для *java.util.logging*, *Log4J*, *Commons Logging* – т.е. его можно подключить "под" эти фреймворки, так, что вывод через них будет перенаправляться в *SLF4J*. Единственное ограничение – невозможность работы в схеме *Log4J >> SLF4J >> Log4J*.
4. Проблем с загрузчиком классов у него нет

Устроен *SLF4J* просто, в некоторой степени даже элегантно. Есть общая часть библиотеки, API. И есть несколько библиотек, каждая из которых реализует свою схему – *SLF4J >> Log4J*, *SLF4J >> Commons Logging*, *SLF4J >> java.util.logging*, *SLF4J >> NOP*, *SLF4J >> Simple*. Эти библиотеки подключаются простым помещением их в classpath. Сделано это следующим образом – каждая из дополнительных библиотек содержит класс с определенным именем (на самом деле классов несколько, но это неважно). Этот класс является, фактически, фабрикой для создания всех сущностей *SLF4J* на основе конкретного фреймворка. А следует из этого такой момент – *дополнительные библиотеки можно подключать только по одной*! Нет, можно и все, но задействована будет первая найденная в classpath-е.

В целом *SLF4J* является более удачным, нежели *Commons Logging*, потому в последнее время наблюдается тенденция перехода на него. Кстати, у этого фреймворка тот же автор, что и у *Log4J*.

При всех плюсах *SLF4J* один минус я все-таки знаю – он не дает изменять уровень детализации логирования для конкретного логгера во время исполнения (помните *мечту*?), хотя и *Log4J* и *java.util.logging* это поддерживают.

Ну и последний по порядку, но не по значению – *Logback*.

Logback

Logback концептуально является наследником *Log4J*. Что неудивительно, ибо автор у них один – вместе с *SLF4J*. И опять-таки неудивительно, что *Logback* при этом является реализацией интерфейсов *SLF4J*, т.е. максимально к нему приближен. Что немаловажно с точки зрения прежде всего производительности.

От *Log4J Logback* взял все хорошее, что там было. То есть – практически всё. По используемым понятиям они похожи до степени смешения. А когда читаешь документацию, понимаешь, что она местами вообще идентична. Однако в *Logback* добавлено несколько интересных возможностей, которые могут сильно облегчить жизнь. Как, например, зависимость уровня логирования от определенных параметров, что позволяет воспроизводить ошибки в режиме промышленной эксплуатации, выставляя уровень детального логирования не всему приложению, а только действиям, совершаемым указанным пользователем.

Подробно *Logback* я тоже рассматривать не буду – во-первых, потому что он сильно похож на *Log4J*, о котором мы будем говорить подробно, а во-вторых, потому что он еще, на мой взгляд, не достиг зрелости. Текущая его версия – 0.9.20. Однако по возможностям он весьма привлекателен, так что для текущих своих задач его непременно попробую, тем более что мы используем *SLF4J*, с которым *Logback* сочетается очень легко.

Краткое резюме

Подведем промежуточный итог. Хочу в очередной раз подчеркнуть – все нижесказанное является всего лишь моим мнением.

- *Apache Log4J* – хороший фреймворк для логирования, практически лишенный недостатков. Широко используется. Разработка находится, фактически, в замороженном[5] состоянии, производится только исправление ошибок.
- *java.util.logging* – фреймворк, являющийся частью JavaSE. По возможностям уступает *Log4J*. Тем не менее используется хотя бы потому, что всегда под рукой и не требует дополнительных библиотек.
- *Apache Commons Logging* – фреймворк, предназначенный для абстрагирования конкретного фреймворка ("под" ним может работать как *Log4J*, так и *java.util.logging*, а также несколько других). Имеет определенные проблемы с загрузчиком классов, что в определенных ситуациях затрудняет его использование.
- *SLF4J* – еще один абстрагирующий фреймворк, существенно более удачный, чем *Commons Logging*. Может работать в двух ипостасях – как общий интерфейс к лежащим ниже фреймворкам и как приемник соответствующего типа для фреймворков, расположенных "над" ним.
- *Logback* – молодой, но весьма интересный фреймворк, выросший из *Log4J*. Взял от родителя все преимущества, плюс еще добавил своих. Возможно, в будущем станет даже более привлекательным, чем *Log4J*.

Теперь о том, что и когда использовать. Если вы пишете библиотеку, которая будет (или хотя бы может быть) использована сторонними разработчиками – имеет смысл использовать *SLF4J*. Этот фреймворк не будет вас стеснять, а другим даст свободу выбора. *Commons Logging* я бы не рекомендовал использовать вообще.

Если же вы разрабатываете собственное приложение, имеет смысл использовать *Log4J*. Он обладает большими возможностями, нежели *java.util.logging*, и даст вам больше пространства для маневра в том случае, если вы его же будете использовать "под" *SLF4J*. Например, в *Log4J* во время исполнения вы можете получить по имени логгер и поменять ему уровень (вот тут есть *пример*), а *SLF4J* в своем API такой возможности не имеет. *Logback*, на мой взгляд, использовать в промышленном режиме пока рано, хотя покрутить в руках, безусловно, стоит.

Есть, правда, и тут один нюанс. Мы у себя в собственном приложении, которое ни разу не библиотека, используем тем не менее *SLF4J*. Дело в том, что у нас задействована куча разных библиотек, которые, в свою очередь, используют кто во что горазд. И для того, чтобы весь этот зоопарк требуемых логгеров свести в одну точку, мы и используем *SLF4J* в режиме, так сказать, обратного мостика – "под" логгерами. А ниже него – *Log4J*, который мы уже конфигурируем напрямую, используя все его возможности.

Общая часть о различных фреймворках закончена, перейдем к конкретике.

Использование Log4J

Apache Log4J является весьма зрелым фреймворком, с устоявшимися *принципами и понятиями*. Поддерживает несколько способов *конфигурации*. Позволяет *управлять* своим поведением во время исполнения. Все это, а также многое другое, будет рассмотрено далее.

Принципы и понятия

В основе библиотеки *Log4J* лежит три понятия – **логгер** (logger), **аппендер** (appender) и **компоновка** (layout). К сожалению, нормальных эквивалентных русских терминов не существует, прижились заимствованные английские.

И еще к вопросу о терминологии – порции выводимых данных в *Log4J* называются сообщениями.

Логгер

Логгер представляет собой объект класса `org.apache.log4j.Logger`, который используется для вывода данных и управления уровнем (детализацией) вывода. В текущей версии – 1.2.16 – *Log4J* поддерживает следующие уровни вывода, в порядке возрастания:

- TRACE
- DEBUG
- INFO
- WARN
- ERROR
- FATAL
- OFF

Установка логгеру определенного уровня означает следующее – сообщения, выводимые с этим или более высоким уровнем, попадут в лог. Сообщения, выводимые с уровнем *ниже* установленного в лог не попадут. И в этом заключается вся прелесть – можно вставлять в программный код вывод информации на различных уровнях (об ошибках – на уровне *ERROR*, о нормальном ходе выполнения – на уровне *INFO*, отладочную – на уровне *DEBUG*), а потом гибко регулировать, что именно будет выводиться. Как именно регулировать – мы рассмотрим дальше.

> Необходимо упомянуть еще о таком понятии как *категория* (`org.apache.log4j.Category`). Фактически это тот же логгер, я, честно сказать, не вижу разницы между ними, тем более что программно логгер наследует категорию. Рекомендуется использовать логгер, потому категорий я не касаюсь. Хотя в некоторых старых библиотеках типа *Hibernate* используются именно категории. Да и терминологически имя логгера чаще всего в документации называется категорией.

Другим важным свойством логгеров является то, что они организованы иерархично. Каждый логгер имеет имя, описывающее иерархию, к которой он принадлежит. Разделитель – точка. Принцип полностью аналогичен формированию имени пакета в Java.

Зачем это нужно. Дело в том, что установленный логгеру уровень вывода распространяется на все его дочерние логгеры, для которых явно не выставлен уровень. Проще это показать на примере.

Пусть у нас есть иерархия *ru.skipy.logger.test*. Это четыре логгера. Для начала назначим каждому свой уровень. Получается вот что:

Имя	Назначенный уровень	Эффективный уровень
ru	INFO	INFO

ru.skipy	WARN	WARN
ru.skipy.logger	DEBUG	DEBUG
ru.skipy.logger.test	ERROR	ERROR

Теперь *не будем* назначать уровень логгерам *ru.skipy* и *ru.skipy.logger.test*. Тогда уровни будут такими:

Имя	**Назначенный уровень**	**Эффективный уровень**
ru	INFO	INFO
ru.skipy	**нет**	**INFO**
ru.skipy.logger	DEBUG	DEBUG
ru.skipy.logger.test	**нет**	**DEBUG**

Возникает вопрос. А что будет, если не указать уровень для *ru*? Какой будет уровень?

Ответ простой. Дело в том, что помимо тех логгеров, которые создаются нами, есть еще один, корневой. У него нет имени, его можно получить с помощью специального метода. И именно этот логгер является родительским как для *ru*, так и для всех остальных на самом верхнем уровне. Именно его уровень они наследуют. Не установить этот уровень нельзя, это ошибка конфигурации.

Таким образом, реальная картинка на самом деле следующая:

Имя	**Назначенный уровень**	**Эффективный уровень**
root	INFO	INFO
ru	**нет**	**INFO**
ru.skipy	**нет**	**INFO**
ru.skipy.logger	DEBUG	DEBUG
ru.skipy.logger.test	**нет**	**DEBUG**

Такой подход дает большую гибкость – можно для всех на логгере *root* выставить требуемый уровень (чаще всего это *ERROR*), а для необходимых логгеров его менять, причем как в сторону понижения, так и в сторону повышения.

Переходим к следующему понятию, а именно –

Аппендер

Если логгер – это та точка, куда уходят сообщения в коде, то аппендер – это та точка, куда они приходят в конечном итоге. Например, файл. Или консоль. Хотя на самом деле список таких точек, поддерживаемых *Log4J*, намного шире:

- Консоль
- Файлы (несколько различных типов)
- JDBC
- Темы (topics) JMS
- NT Event Log
- SMTP
- Сокет
- Syslog
- Telnet
- Любой *java.io.Writer* или *java.io.OutputStream*

И это не говоря уже о том, что никто не мешает написать свой аппендер и благополучно его использовать. Да, все эти объекты реализуют интерфейс `org.apache.log4j.Appender`.

Логгеры связываются с аппендерами в соотношении "многие ко многим" – у одного логгера может быть несколько аппендеров, а к одному аппендеру может быть привязано несколько логгеров. Важно понимать, что **аппендеры наследуются от родительских логгеров**. Т.е., например, если к корневому (*root*) логгеру в конфигурации привязан аппендер *A1*, а к логгеру *ru.skipy* – *A2*, то вывод в логгер *ru.skipy* попадет в *A2* **и** *A1*, а вывод в *ru* – только в *A1*.

И еще один момент, который обычно вызывает сложности. Уровень логирования наследуется (или устанавливается) **независимо** от аппендера. Иначе говоря, если на логгере *root* сконфигурирован вывод в *A1* с уровнем *ERROR*, а на *ru.skipy* – в *A2* с уровнем *INFO*, то вывод в *ru.skipy* с уровнем *INFO* попадет и в *A2*, **и** в *A1*, несмотря на то, что в конфигурации прямо рядом с *A1* указан уровень *ERROR*. Такой поворот событий часто приводит в ступор – как же так, я же явно указываю для *root* уровень *ERROR*, почему же сюда сыпется всё, вплоть до *DEBUG*??? Как раз поэтому. Аппендер унаследовали, а уровень вывода переписали.

Существует возможность отказаться от наследования аппендеров. Для этого логгеру надо выставить свойство *additivity* в *false*, по умолчанию оно выставлено в *true*. Ну и, соответственно, всё вышесказанное в виде таблицы:

Имя логгера	**Назначенные аппендеры**	**Значение *additivity***	**Эффективные аппендеры**	**Комментарии**
root	A1		A1	Родительских аппендеров нет, *additivity* значения не имеет
ru	A2,A3	true	A1,A2,A3	Аппендеры родительского (корневого) логгера плюс собственные
ru.skipy	-	true	A1,A2,A3	*Все* аппендеры родительского логгера (включая унаследованные), собственных нет
ru.skipy.logger	A4	true	A1,A2,A3,A4	*Все* аппендеры родительского логгера (включая унаследованные) плюс собственные
info	A5	**false**	A5	Только собственные аппендеры – родительские не наследуются
info.skipy	-	true	A5	Только родительские аппендеры – от *ближайшего* родителя, собственных нет

Эта таблица вместе с *таблицей наследования уровней* сильно облегчают понимание происходящего. Вывод идет во *все* эффективные аппендеры на эффективном уровне и вне зависимости от того, какой уровень сконфигурирован для родителя, к которому привязан соответствующий аппендер.

* * *

Поговорим теперь об имеющихся типах аппендеров. Всех мы касаться совершенно точно не будем, это выходит за рамки обзорной статьи. Желающих углубиться отсылаю к документации по интерфейсу: *http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/Appender.html*. Мы же будем говорить о тех типах, которые используются чаще всего. Это консоль и файлы.

Начнем с `org.apache.log4j.ConsoleAppender`, как с более простого. Этот аппендер используется для вывода данных в консоль (STDOUT). Это удобно при отладке, но практически ничего не дает в случае, когда надо диагностировать ошибку. Дело в том, что некоторые сервера приложений, например, вывод в консоль просто глушат – перехватывают и никуда не выпускают.

У консольного аппендера, как и у многих других (на самом деле – у всех, унаследованных от *org.apache.log4j.WriterAppender*), есть свойство, позволяющее указать ему кодировку, в которой выводить данные. На всякий случай напоминаю, что в Windows консоль имеет кодировку *Cp866*. Как это свойство выставляется, мы увидим в разделе *Конфигурирование*.

Теперь перейдем к основным аппендерам, использующимся наиболее широко – файловым. Их есть несколько типов:

- `org.apache.log4j.FileAppender`
- `org.apache.log4j.RollingFileAppender`
- `org.apache.log4j.varia.ExternallyRolledFileAppender`
- `org.apache.log4j.DailyRollingFileAppender`

`org.apache.log4j.FileAppender`

Начнем опять с самого простого – `org.apache.log4j.FileAppender`. Как нетрудно доогадаться, этот аппендер добавляет данные в файл. До бесконечности. И в этом его существенный недостаток – файл размером в 500Мб просмотривать весьма неудобно. Да и с записью бывают проблемы. Потому этот аппендер сам по себе практически не используется. Он является базой для остальных, предоставляя общие средства работы с файлами. Поддерживает этот аппендер следующие свойства: *append* (дописывать существующий файл или каждый раз начинать заново), *bufferedIO* (буферизовать ли вывод в файл), *file* (имя файла). Ну и от *org.apache.log4j.WriterAppender* унаследовано свойство *encoding*.

`org.apache.log4j.RollingFileAppender`

Гораздо интереснее `org.apache.log4j.RollingFileAppender`. Этот аппендер позволяет ротировать файл по достижении определенного размера. "Ротировать" означает, что текущему файлу приписывается расширение ".0" и открывается следующий. По достижении им максимального размера – первому вместо расширения ".0" выставляется ".1", текущему – ".0", открывается следующий. И так далее. Максимальный размер файла и максимальный индекс, устанавливаемый сохраняемым предыдущим файлам, задаются свойствами *maximumFileSize* и *maxBackupIndex* соответственно. Если индекс должен быть превышен – файл не переименовывается, а удаляется. Таким образом мы всегда имеем не больше определенного количества файлов, каждый из которых не больше определенного объема. Гораздо более предсказуемая ситуация, чем с обычным *FileAppender*-ом. Собственно, этот тип аппендеров, наверное, самый используемый.

`org.apache.log4j.varia.ExternallyRolledFileAppender`

Дальше идет `org.apache.log4j.varia.ExternallyRolledFileAppender`. Любопытный аппендер, я бы сказал – вещь в себе. Он унаследован от *org.apache.log4j.RollingFileAppender*-а, соответственно, ведет себя так же и имеет те же свойства. Однако вдобавок к этому он еще слушает указанный ему порт. Если открыть соединение на этот порт и послать строку *"RollOver"* (через `DataOutputStream.writeUTF(java.lang.String)`) – аппендер выполняет внеочередную смену индексов. Мне сложно представить, зачем такое нужно, хотя может пригодиться.

`org.apache.log4j.DailyRollingFileAppender`

Следующий! `org.apache.log4j.DailyRollingFileAppender`. Очень полезный и функциональный аппендер. В отличии от *org.apache.log4j.RollingFileAppender*-а, ротирующего файл по достижении определенного размера, *org.apache.log4j.DailyRollingFileAppender* ротирует файл с определенной частотой. Она зависит от шаблона, указанного в конфигурации:

- `'.'yyyy-MM` – файл ротируется раз в месяц
- `'.'yyyy-ww` – файл ротируется раз в неделю
- `'.'yyyy-MM-dd` – файл ротируется раз в день
- `'.'yyyy-MM-dd-a` – файл ротируется раз в полдня
- `'.'yyyy-MM-dd-HH` – файл ротируется раз в час
- `'.'yyyy-MM-dd-HH-mm` – файл ротируется раз в минуту

При ротации к имени файла в конце приписываются текущие дата и время, отформатированные согласно указанному шаблону (с помощью класса `java.text.SimpleDateFormat`). В кавычках в начале шаблона указан символ, который будет использоваться как разделитель между значением даты/времени и именем файла. В принципе этот символ может быть практически любым, кроме ":" – он интерпретируется в имени файла как указатель имени сетевого протокола (всё, что стоит до ":").

Этот аппендер может быть весьма удобен в случае, когда у вас организована автоматическая архивация лога. Кроме того, наличие в имени файла временной метки делает его по определению уникальным – лог не потеряется, как это может произойти с обычным ротирующим аппендером.

Как конфигурируются все эти аппендеры, мы увидим ***в соответствующем разделе***. А пока переходим к третьему базовому понятию.

Компоновка

Мы уже знаем, что нужно использовать, чтобы вывести данные – логгер. Правда, пока не знаем, как – но это ненадолго. :) Мы знаем, что нужно использовать для конфигурирования точки назначения – аппендер. Однако мы еще не знаем, как сконфигурировать формат вывода данных. Пришла пора узнать и это.

Для конфигурирования формата вывода используются наследники класса `org.apache.log4j.Layout`:

- `org.apache.log4j.SimpleLayout`
- `org.apache.log4j.HTMLLayout`
- `org.apache.log4j.xml.XMLLayout`
- `org.apache.log4j.TTCCLayout`
- `org.apache.log4j.PatternLayout` / `org.apache.log4j.EnhancedPatternLayout`

У каждого из них свое предназначение и свои возможности.

`org.apache.log4j.SimpleLayout`

Наиболее простой вариант. На выходе дает ***уровень вывода*** и, собственно, сообщение. Т.е. следующий код –

```
logger.info("Some message");
```

– на выходе даст вот так отформатированную строку:

```
INFO - Some message
```

В принципе, для какого-то простейшего случая такая компоновка может и пригодиться. Однако необходимо заметить, *что вывод исключений в лог при использовании данной компоновки невозможен*.

`org.apache.log4j.HTMLLayout`

Более интересный вариант. Форматирует сообщения в виде HTML-таблицы. У этого компоновщика есть два свойства – *Title* и *LocationInfo*, задающие заголовок HTML-документа и режим вывода информации о точке, где сгенерировано сообщение (имя файла и номер строки в нем) соответственно. По умолчанию *LocationInfo* имеет значение *false*, т.к. **генерация информации о точке возникновения сообщения – крайне затратная процедура**.

Есть один момент, который необходимо принимать внимание при использовании этого компоновщика. Формат HTML требует корректного закрытия документа. А при генерации вывода, естественно, ни о каком закрытии речи не идет – мы непрерывно добавляем сообщения, т.е. строки в таблицу. Так вот, для того, чтобы получить корректный документ, необходимо *закрыть* компоновщик в конце работы, обычно при выходе из приложения. Делается это при помощи следующего вызова статического метода:

```
org.apache.log4j.LogManager.shutdown();
```

Фактически, это закрытие всей системы логирования – сброс всех кешированных данных, закрытие всех форматов и т.п.

Результат работы этого компоновщика на примере, описанном ***выше***, можно увидеть в ***этом документе***. Вызов логгера производится в методе `main` класса `ru.skipy.logging.tests.Log4JTest`, в строке 19.

Исключения данный компоновщик поддерживает. Необходимо также помнить, что при его использовании кодировка аппендеру должна быть выставлена в `UTF-8` или `UTF-16`, в противном случае данные при выводе могут быть повреждены. Хотя сам компоновщик кодировку в генерируемый HTML не добавляет...

`org.apache.log4j.xml.XMLLayout`

Этот компоновщик формирует сообщения в виде XML. В противовес *HTMLLayout* он **не генерирует корректного – well-formed – документа**. Результаты его работы должны быть вставлены в определенное XML-обрамление – только тогда получится корректный документ. Подробнее можно посмотреть вот тут: *http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/xml/XMLLayout.html*.

Ну и соответственно, результатом работы на примере, упомянутом ***выше***, будет вот такой фрагмент xml-документа:

```
<log4j:event logger="ru.skipy.logging.tests.Log4JTest" timestamp="1274084709955" level="INFO" thread="Main Thread">
<log4j:message><![CDATA[Some message]]></log4j:message>
<log4j:locationInfo class="ru.skipy.logging.tests.Log4JTest" method="main" file="Log4JTest.java" line="19"/>
</log4j:event>
```

Всё, что сказано о свойстве *LocationInfo* применительно к *HTMLLayout*, справедливо и тут. Исключения данный компоновщик также поддерживает. И точно так же аппендеру должна быть выставлена кодировка `UTF-8` или `UTF-16`.

`org.apache.log4j.TTCCLayout`

TTCC – сокращение от *Time-Thread-Category-Context*. Означает оно, что помимо, собственно, сообщения, в лог выводится информация о времени, потоке, категории (имени логгера) и ***вложенном диагностическом контексте*** (о них мы ***поговорим ниже***). У компоновщика есть булевские свойства *CategoryPrefixing*, *ContextPrinting* и *ThreadPrinting*, указывающие, выводить или нет категорию, контекст и имя потока, соответственно. По умолчанию все три свойства выставлены в *true*.

Еще одно свойство – *DateFormat*. Оно позволяет указать, в каком варианте будет выводиться время. Имеется пять предустановленных форматов (шаблоны указаны применительно к `java.text.SimpleDateFormat`):

Имя формата	Значение
NULL	Значение не выводится
RELATIVE	Число – количество миллисекунд с момента инициализации Log4J
ABSOLUTE	Время в формате "**`HH:mm:ss,SSS`**"
DATE	Дата и время в формате "**`dd MMM yyyy HH:mm:ss,SSS`**"
ISO8601	Дата и время в формате "**`yyyy-MM-dd HH:mm:ss,SSS`**"

Если значение не соответствует ни одному из предустановленных – это должен быть шаблон в формате, поддерживаемом `java.text.SimpleDateFormat`.

В следующей таблице содержатся результаты вывода в лог сообщения, приведенного ***выше***, в зависимости от указанного формата даты:

Формат даты	Сообщение в логе
NULL	`[Main Thread] INFO ru.skipy.logging.tests.Log4JTest - Some message`
RELATIVE	`15 [Main Thread] INFO ru.skipy.logging.tests.Log4JTest - Some message`
ABSOLUTE	`13:26:34,804 [Main Thread] INFO ru.skipy.logging.tests.Log4JTest - Some message`
DATE	`17 май 2010 13:26:34,804 [Main Thread] INFO ru.skipy.logging.tests.Log4JTest - Some message`
ISO8601	`2010-05-17 13:26:34,804 [Main Thread] INFO ru.skipy.logging.tests.Log4JTest - Some message`
dd.MM.yyyy	`17.05.2010 13:26:34.804 [Main Thread] INFO ru.skipy.logging.tests.Log4JTest - Some message`

HH:mm:ss.SSS	

Компоновщик *TTCCLayout* поддерживает вывод исключений. При программном использовании необходимо помнить, что этот компоновщик НЕ потоко-независимый (not thread-safe), так что необходимо создавать по экземпляру на каждый аппендер, к которому он привязывается.

`org.apache.log4j.PatternLayout` / `org.apache.log4j.EnhancedPatternLayout`

Наконец мы добрались до самого мощного и наиболее часто используемого компоновщика – *PatternLayout*. Он использует шаблонную строку для форматирования выводимого сообщения. Формат чем-то напоминает `printf` – тот же знак `'%'`, после которого (возможно) идет модификатор формата и дальше символ, обозначающий тип выводимых данных. Кроме таких служебных комбинаций в строку шаблона можно вставлять любые символы, что позволяет еще более гибко конфигурировать лог.

На самом деле, как вы можете заметить, в заголовке упомянут еще и такой компоновщик, как `org.apache.log4j.EnhancedPatternLayout`. Этот класс появился только в версии 1.2.16. В документации к нему написано, что он является улучшенной версией `org.apache.log4j.PatternLayout` и должен использоваться вместо старого него. Я не заметил каких-то особых улучшений, хотя изменения, безусловно, есть. Далее я буду указывать отличия там, где они появятся.

В следующей таблице приведены опции, обозначающие типы выводимых данных. Обратите внимание, что **они чувствительны к регистру** – 'c' означает совсем не то, что 'C'.

Опция	Значение, выводимое в лог
c	**Категория сообщения**. После символа категории в фигурных скобках может следовать указание – сколько частей имени категории выводить. Они отсчитываются с конца, что логично – это позволяет отсечь длинное имя пакета. Т.е., например, при имени категории *ru.skipy.logging.tests.Log4JTest* комбинация `%c{3}` приведет к выводу в лог *logging.tests.Log4JTest* (три части имени с конца). Если такого указания нет – имя выводится целиком. У **`org.apache.log4j.EnhancedPatternLayout`** есть еще несколько вариантов сокращения имени. Отрицательное значение в скобках означает "убрать указанное количество частей сначала". Т.е. при имени категории *ru.skipy.logging.tests.Log4JTest* комбинация `%c{-3}` приведет к выводу в лог *Log4JTest* – три первых части убраны. Если убрать надо б**о**льшее количество частей, чем присутствует – будет выведено все имя целиком. Еще один вариант сокращения – запись вида `%c{1.2.3.}`. Означает она, что от первой части остается одна буква, от второй – две, от третьей – три. На оставшиеся части распространяется последнее значение. Последняя часть имени выводится целиком. Т.е. из имени *ru.skipy.logging.tests.Log4JTest* форматом `%c{1.2.1.}` мы получим *r.sk.l.t.Log4JTest* – одна буква, две, далее опять одна. Можно задать еще и символ, которым будут замещаться убранные символы: `%c{1*.2#.1$}` даст результат *r*.sk#.l$.t$.Log4JTest*. При длинных именах категорий такой формат может оказаться удобным.
C	**Полное имя класса, в котором сгенерировано сообщение** Не путайте это имя с именем категории. В разделе *Использование в программном коде* мы увидим, что это две большие разницы. Имя категории может не совпадать с именем класса. После имени класса также может идти указание на то, сколько частей имени выводить – полностью аналогично опции `'%c'`. **Важно!** Генерация имени класса – достаточно медленная процедура. Стоит ее избегать при наличии такой возможности. И уж точно не стоит оставлять эту опцию в настройках в режиме промышленной эксплуатации. У **`org.apache.log4j.EnhancedPatternLayout`** есть все те же варианты сокращения имени класса, которые перечислены для категории.
d	**Дата и/или время** Выводит в лог текущие дату и/или время. В фигурных скобках после данной опции указывается формат даты – либо шаблон `java.text.SimpleDateFormat`, либо один из предустановленных – `DATE`, `ABSOLUTE` или `ISO8601`. Сравните этот набор с таблицей *выше* – поддерживаются *не все* форматы, перечисленные для *TTCCLayout*. Если указания на шаблон нет, используется формат `ISO8601`. Документация рекомендует использовать предопределенные форматы вместо собственных шаблонов – под них разработаны специальные классы для более оптимального форматирования, чем это делает `java.text.SimpleDateFormat`. У **`org.apache.log4j.EnhancedPatternLayout`** есть еще возможность в фигурных скобках указать временную зону, например, `{GMT+1}`. В этом случае дата будет выводиться в указанной временной зоне, вне зависимости от установленной на сервере.
F	**Имя файла, в котором было сгенерировано сообщение**. Не путайте с именем класса. В данном случае в лог выведется именно имя файла, в общем случае не совпадающее с именем класса, например, для любых внутренних классов. **Важно!** Генерация имени класса – **крайне** медленная процедура. Стоит ее избегать при наличии любой возможности. И категорически не рекомендуется оставлять эту опцию в настройках в режиме промышленной эксплуатации.
l	**Полная информация о точке генерации сообщения**. Содержит имя класса, имя метода, имя файла и строку, в которой было сгенерировано сообщение. Т.е. для нашего примера *выше* (напомню, что вызов логгера производится в методе `main` класса `ru.skipy.logging.tests.Log4JTest`, в строке 19), в лог будет выведена строка `ru.skipy.logging.tests.Log4JTest.main(Log4JTest.java:19)`. **Важно!** Фактически эта опция является аналогом следующей конструкции: `%C.%M(%F:%L)`. Генерация **каждой** из частей в этом наборе – **крайне** медленная процедура. Ну и вся комбинация, естественно, быстрой не будет. Поэтому опции `%l` необходимо категорически избегать в режиме промышленной эксплуатации. В то же время в процессе отладки она может оказать неоценимую помощь.
L	**Номер строки, в которой было сгенерировано сообщение**. Имеется в виду номер строки в файле. В принципе информация полезная, хотя часто можно обойтись и без нее. **Важно!** Генерация номера строки – **крайне** медленная процедура. Стоит ее избегать при наличии любой возможности. И категорически не рекомендуется оставлять эту опцию в настройках в режиме промышленной эксплуатации.
m	**Сообщение** То самое сообщение, которое передается в метод логгера. Ради чего, в основном, всё и затевается.
M	**Имя метода, в котором было сгенерировано сообщение**. **Важно!** Генерация имени метода – **крайне** медленная процедура. Стоит ее избегать при наличии любой возможности. И категорически не рекомендуется оставлять эту опцию в настройках в режиме промышленной эксплуатации.
n	**Перевод строки** Переводит в логе строку. Это необходимо, иначе все сообщения будут писаться в одну строку. Для чего нужна такая опция – конец строки, вообще-то, платформозависимый. Под Windows это '\r\n' (CRLF), под *NIX – '\n' (LF), на Mac-ах до недавнего времени был '\r' (CR), в MacOS X, возможно, стал как в *NIX. И стандартная для одной системы комбинация может быть неправильно интерпретирована в другой. В общем, для того, чтобы нормально читать логи в той операционной системе, где они ведутся, рекомендуется для перевода строки использовать именно опцию `%n`.
p	**Приоритет сообщения**. Выводит *уровень логирования* для сообщения.
r	**Количество миллисекунд с момента инициализации системы логирования**. Аналог *формата даты* `RELATIVE` компоновщика *TTCCLayout*. Может использоваться вместо даты, если есть такая необходимость.
t	**Имя потока**. Выводит имя потока, в котором сгенерировано сообщение. Эта информация бывает весьма полезна, особенно если не лениться и осмысленно именовать все порождаемые потоки.
x	**Вложенный диагностический контекст (NDC)** Выводит связанный с текущим потоком *вложенный диагностический контекст*.

`X`	**Ассоциативный диагностический контекст (MDC)**. Выводит связанный с текущим потоком *ассоциативный диагностический контекст*. После опции в фигурных скобках **должно** идти имя ключа, по которому выбирается значение из контекста: `%X{username}` – вывод из контекста имени пользователя, если оно там есть. У **`org.apache.log4j.EnhancedPatternLayout`** имя ключа необязательно. Если его нет – выводится всё содержимое контекста.
`%`	**Знак процента**. Поскольку знак '%' является частью формата, а необходимость в его выводе периодичеси присутстует, конструкция `'%%'` выводит в лог знак '%'.
`properties`	**Свойства, связанные с сообщением**. Эта опция специфична только для **`org.apache.log4j.EnhancedPatternLayout`**, у `org.apache.log4j.PatternLayout` ее нет. Выводит свойство по имени, указанному в фигурных скобках после опции. Если имя не указано – выводятся все свойства. Честно сказать, разницы между этой опцией и `%x` я не нашел. Более того, по коду свойства сообщения трактуются именно как MDC – название с использованием этого префикса, копирование из него и т.п. В общем, можно использовать то, что нравится.
`throwable`	**Информация об исключении**. Эта опция специфична только для **`org.apache.log4j.EnhancedPatternLayout`**, у `org.apache.log4j.PatternLayout` ее нет. Она позволяет вывести информацию об исключении, если оно было передано в метод логгера. В фигурных скобках указывается количество строк, которые надо выводить. `%throwable{1}` или `%throwable{short}` выведет одну строку – как правило, это имя класса исключения и текстовое сообщение. `%throwable{0}` или `%throwable{none}` вообще подавит вывод информации об исключении. Указание положительного числа означает количество строк, которое надо оставить с начала стека сообщения об ошибке, отрицательное – сколько убрать с конца. Если фигурные скобки отсутствуют – выводится полный стек. Точно так же логгер поступает и при отсутствии этой опции вообще.

С учетом баланса между требованиями производительности и объемом информации, которого достаточно для анализа логов, в промышленном режиме рекомендовано использование следующих опций: `%c, %d, %m, %n, %p, %t, %x, %X, %throwable, %%`. Остальные – `%C, %F, %l, %L, %M` – способны вызвать сильное падение производительности.

С опциями закончили. Но если вы думаете, что возможности по настройке компоновщика на этом заканчиваются – вы в иллюзии. Все только начинается. :)

Данный компоновщик поддерживает, кроме всего прочего, позиционное форматирование. Означает оно, что под каждую опцию можно выделить некоторое место – задать минимальный и максимальный размер значения, а также выравнивание, если значение меньше минимальной выделенной области. Модификаторы форматирования задаются между символом '%' и опцией. На примере опции `%c` рассмотрим действие модификаторов:

Модификатор	Выравнивание	Минимальная ширина	Максимальная ширина	Действие
`%10c`	вправо	10	нет	Отводит минимум 10 символов под имя категории, если длина значения меньше – выравнивает его по правому краю поля
`%-10c`	влево	10	нет	Отводит минимум 10 символов под имя категории, если длина значения меньше – выравнивает его по левому краю поля
`%.20c`	нет	нет	20	Отводит максимум 20 символов под имя категории, если длина значения больше – обрезает с начала, оставляя указаное количество символов. Поскольку длина значения не может быть меньше предопределенной, о выравнивании говорить не приходится.
`%10.20c`	вправо	10	20	Отводит минимум 10 и максимум 20 символов под имя категории, если длина значения меньше – выравнивает его по правому краю поля, если больше – обрезает с начала, оставляя 20 символов.
`%-10.20c`	влево	10	20	Отводит минимум 10 и максимум 20 символов под имя категории, если длина значения меньше – выравнивает его по левому краю поля, если больше – обрезает с начала, оставляя 20 символов.

Думаю, суть ясна. Теперь – для чего это нужно. Ну, ограничивать размер выводимой информации сверху – в приниципе правильно. Сообщение размером в 10000 символов не несет никакой практической ценности. Только загромождает лог, искать действительно важную информацию становится труднее. А вот зачем ограничивать размер снизу?

А это тоже просто. Представьте себе, что вы выводите в лог, скажем, дату, имя потока, приоритет, категорию и сообщение. Сравните два следующих фрагмента:

```
11:31:32,342 Thread-1 ERROR ru.skipy.tests.audit.LoadTest - Check in 344ms: GlobalID=2
11:31:32,358 Thread-17 WARN ru.skipy.tests.ServiceLoadTest - Check in 156ms: GlobalID=8
11:31:32,378 Thread-2 INFO ru.skipy.tests.trace.ServiceLoadTrace - Check in 328ms: GlobalID=3
11:31:35,358 Thread-44 DEBUG ru.skipy.tests.parallel.ext.ServiceParallelLoadTest - Check in 250ms: GlobalID=5
11:31:36,637 Thread-503 INFO ru.skipy.tests.ServiceLoadTest - Check in 219ms: GlobalID=6
11:31:37,846 Thread-59 INFO ru.skipy.tests.extract.Extractor - Check in 94ms: GlobalID=10
11:31:39,072 Thread-86 DEBUG ru.skipy.tests.ServiceLoadTest - Check in 188ms: GlobalID=7
11:31:41,309 Thread-10 INFO ru.skipy.tests.back.BackLoaderInfo - Check in 47ms: GlobalID=11
```

```
11:31:32,342 Thread-1   ERROR    ru.skipy.tests.audit.LoadTest - Check in 344ms: GlobalID=2
11:31:32,358 Thread-17  WARN    ru.skipy.tests.ServiceLoadTest - Check in 156ms: GlobalID=8
11:31:32,378 Thread-2   INFO  ipy.tests.trace.ServiceLoadTrace - Check in 328ms: GlobalID=3
11:31:35,358 Thread-44  DEBUG llel.ext.ServiceParallelLoadTest - Check in 250ms: GlobalID=5
11:31:36,637 Thread-503 INFO    ru.skipy.tests.ServiceLoadTest - Check in 219ms: GlobalID=6
11:31:37,846 Thread-59  INFO  ru.skipy.tests.extract.Extractor - Check in 94ms: GlobalID=10
11:31:39,072 Thread-86  DEBUG   ru.skipy.tests.ServiceLoadTest - Check in 188ms: GlobalID=7
11:31:41,309 Thread-10  INFO  .skipy.tests.back.BackLoaderInfo - Check in 47ms: GlobalID=11
```

Вопросы. В каком из двух вариантов сообщения читаются проще? Насколько быстро вы можете найти в строке начало сообщения в первом из двух вариантов? А во втором? Как быстро вы можете увидеть, какой категории сообщение?

На мой взгляд, разница видна невооруженным глазом. Второй вариант существенно удобнее читать. А отличается он от первого только тем, что для имени потока, приоритета и категории установлены минимальные ширины – 10, 5 и 32 символа соответственно. Максимальные установлены так же, в результате чего длинные категории не вылезают за отведенные им пределы, и сообщения, соответственно, все начинаются с одной и той же позиции. Более того, даже этот вариант можно несколько улучшшить. Мы знаем, что категория всегда начинается с *ru.skipy.tests*, и видеть в логе эту часть нам совершенно необязательно. Т.е. ее можно убрать (с помощью `%c{-3}`), тогда лог будет выглядеть еще красивее:

```
11:31:32,342 Thread-1   ERROR                audit.LoadTest - Check in 344ms: GlobalID=2
11:31:32,358 Thread-17  WARN               ServiceLoadTest - Check in 156ms: GlobalID=8
11:31:32,378 Thread-2   INFO         trace.ServiceLoadTrace - Check in 328ms: GlobalID=3
11:31:35,358 Thread-44  DEBUG    ext.ServiceParallelLoadTest - Check in 250ms: GlobalID=5
11:31:36,637 Thread-503 INFO               ServiceLoadTest - Check in 219ms: GlobalID=6
11:31:37,846 Thread-59  INFO             extract.Extractor - Check in 94ms: GlobalID=10
11:31:39,072 Thread-86  DEBUG              ServiceLoadTest - Check in 188ms: GlobalID=7
11:31:41,309 Thread-10  INFO           back.BackLoaderInfo - Check in 47ms: GlobalID=11
```

Теперь и категория читается легко.

Ну и последнее – пример шаблона, от которого я обычно отталкиваюсь в работе:

```
%d{ISO8601} [%-5p][%-16.16t][%32.32c] - %m%n
```

Расшифровка:

1. **Дата в формате `ISO8601`**. Когда логи большие, только времени будет недостаточно, нужна и дата. У формата `DATE` используется символьное имя месяца, потому ширина получается переменной.
2. **Приоритет** – минимально 5 символов (а больше и не бывает в *Log4J*), выравнивание влево.
3. **Имя потока** – 16 символов, выравнивание влево.
4. **Категория** – 32 символа, выравнивание вправо. В принципе, можно обрезать начальные части имени, но тогда надо быть уверенным, что в конкретный аппендер будут идти логи только от собственного кода, иначе есть риск не увидеть разницы между `ru.skipy.SQL` и `org.hibernate.SQL`.
5. **Сообщение**, после которого идет символ окончания строки.

Все части лога, кроме даты и сообщения, заключены в квадратные скобки. Это улучшает читаемость – визуально подчеркивает колонки. Да, я забыл упомянуть раньше – **в шаблон можно включать любые символы помимо, собственно, описателей элементов лога**. При необходимости в шаблон можно добавить и вывод *диагностических контекстов* (*MDC* и *NDC*).

И последняя часть в разделе о концепциях *Log4J* –

Диагностические контексты

Понятия диагностических контекстов я, честно сказать, до *Log4J* не встречал. Используются они редко, а зря, на мой взгляд. Ибо иногда крайне полезная вещь. Итак, что это такое и для чего это нужно.

Представьте себе веб-приложение. Множество пользователей, множество запросов. Информация идет в лог. И вот, у одного пользователя возникают проблемы. Система ведет себя странно. У других – все в порядке. Надо разобраться, что делает не так этот пользователь. Вопрос. Как отследить в логах именно его действия?

Log4J много чего умеет выводить самостоятельно – набор *опций* в том же `PatternLayout` достаточно велик – но он все-таки не всесилен. И всегда будет дополнительная, диагностическая информация, которую придется выводить нам самостоятельно. На первый взгляд сложностей это не вызывает. Ну получить имя пользователя, ну вставить в сообщение – что сложного?

А сложность тут в том, что для четкой идентификации имя пользователя надо добавлять в КАЖДОЕ сообщение. Т.е. его либо надо будет получать во всех точках, где ведется лог, что означает, что весь код должен знать о том, что он выполняется под каким-то пользователем, – бессмысленное знание для большей части кода, – либо протаскивать его туда как параметр, что означает ровно то же самое.

Да и это не даст нужного результата. Ну да, в *свои* сообщения вы имя пользователя добавите. А в сообщения *Hibernate*? А в сообщения остальных девяноста девяти библиотек?

Можно, конечно, отслеживать, в каком потоке начинается запрос, а потом ориентироваться по имени потока. Беда только в том, что потоки сервер переиспользует. И потеряться, где закончился прошлый запрос, и начался следующий, от другого пользователя, легче легкого. Поверьте на слово.

И это всего лишь один пример, а их намного больше. Необходимость во включении в лог дополнительной информации возникает довольно часто. А протаскивать эту информацию до точки генерации сообщения либо сложно (часто), либо неправильно (практически всегда), либо вообще нереально, в случаях с любыми библиотеками.

Что делать?

Вот тут-то как раз и появляется такое понятие как *диагностический контекст*. Как вы понимаете из названия, он представляет собой некий набор информации – контекст, – предназначенный для диагностики чего-либо.

Диагностических контекстов в *Log4J* два разных типа – *вложенные (Nested diagnostic context, NDC)* и *ассоциативные (Mapped diagnostic context, MDC)*. Разница между двумя этими контекстами во внутренней организации. Вложенный представляет собой стек. Ассоциативный – ассоциативный массив (map). Подробнее они будут рассмотренны в разделе об *использовании в программном коде*, здесь описаны только общие принципы.

Диагностический контекст привязывается к потоку, с помощью `java.lang.ThreadLocal`. Это обеспечивает его доступность в любой точке, где выводится в лог сообщение. Поскольку добавлением контекста в лог занимается *Log4J*, а не мы, – это возможно даже в тех точках, где мы создание сообщений не контролируем. Надо только добавить соответствующие опции в шаблон в *PatternLayout*. Также *вложенный* контекст выводится еще и *TTCCLayout*-ом.

* * *

Вот мы и добрались до практики. Начнем с конфигурирования.

Конфигурирование

Конфигурирование *Log4J* осуществляется двумя способами – через файл свойств и через xml-файл. Принято считать эти два способа равнозначными. У меня по этому поводу нет четкого мнения, сам я досконально этот вопрос не исследовал, а документация по самому *Log4J* местами противоречива (например, в описании класса `org.apache.log4j.PropertyConfigurator` сказано, что обработчики ошибок он не поддерживает, а в описании метода `doConfigure` этого же класса приведен формат конфигурации обработчиков ошибок). В любом случае за более чем 5 лет использования *Log4J* я ни разу не сталкивался с разницей в возможностях конфигурации. Потому для наших целей будем считать эти способы равноценными и выбирать, что именно использовать, исходя из собственных предпочтений.

Я лично предпочитаю использовать xml-формат. Во-первых, он, на мой взгляд, проще для понимания, прежде всего за счет структуры. Во-вторых, в отличие от файла свойств, тут все-таки есть DTD[6], т.е. – формат более строгий. Все примеры я, естественно, буду давать в обоих вариантах.

Как нетрудно догадаться, конфигурационные файлы называются *log4j.properties* и *log4j.xml*. При инициализации *Log4J* они ищутся в `classpath`, сначала xml-файл, потом properties-файл. Так что при наличии обоих рабочим будет именно xml.

Итак, простейшие конфигурации. В XML:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration debug="false" xmlns:log4j="http://jakarta.apache.org/log4j/">

    <appender name="ConsoleAppender" class="org.apache.log4j.ConsoleAppender">
        <param name="Encoding" value="Cp866"/>
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%d{ISO8601} [%-5p][%-16.16t][%32.32c] - %m%n" />
        </layout>
    </appender>

    <root>
        <priority value="ERROR"/>
        <appender-ref ref="ConsoleAppender" />
    </root>

</log4j:configuration>
```

И в виде свойств:

```
log4j.debug = false

log4j.rootLogger = ERROR, ConsoleAppender

log4j.appender.ConsoleAppender = org.apache.log4j.ConsoleAppender
log4j.appender.ConsoleAppender.encoding = Cp866
log4j.appender.ConsoleAppender.layout = org.apache.log4j.PatternLayout
log4j.appender.ConsoleAppender.layout.ConversionPattern = %d{ISO8601} [%-5p][%-16.16t][%32.32c] - %m%n
```

Что тут сделано:

- Создан аппендер с именем *ConsoleAppender* и кодировкой *Cp866*
- У созданного аппендера установлен компоновщик *PatternLayout* с шаблоном `%d{ISO8601} [%-5p][%-16.16t][%32.32c] - %m%n`
- Сконфигурирован корневой логгер, использующий созданный аппендер и имеющий уровень `ERROR`.

Комментарии:

- В случае файла свойств порядок указания свойств не важен – все равно они будут загружены в виде `java.util.Properties`, а там порядок не сохраняется
- В случае XML-конфигурации порядок важен – он диктуется DTD.
- В данном примере в явном виде указано, что внутренний лог *Log4J* отключен – свойство `log4j.debug` и атрибут `debug` у корневого элемента в XML имеют значение `false`. Иногда необходимо посмотреть на работу самого *Log4J*, в этом случае данное свойство просто выставляется в true.
- Формат имен свойств следующий: базовая часть `log4j.{logger|appender}.<имя логгера/аппендера>`, дальше могут следовать имена свойств, например, `layout`, дальше – имя свойства внутри свойства, например, `ConversionPattern`. Таким образом создается древесная структура. XML в этом отношении более естественнен – он изначально имеет древесную структуру и вложенность свойств в нем видна с первого взгляда.
- Как уже *упоминалось*, логгеры с аппендерами связаны отношением "многие ко многим", соответственно, у логгера может быть указано несколько аппендеров. В XML просто повторяется тег `appender-ref` с соответствующим именем аппендера (добавляем *FileAppender*):

  ```xml
  <root>
      <priority value="ERROR"/>
      <appender-ref ref="ConsoleAppender" />
      <appender-ref ref="FileAppender" />
  </root>
  ```

  В свойствах – аппендер указывается через запятую:

  ```
  log4j.rootLogger = ERROR, ConsoleAppender, FileAppender
  ```

- В данном примере установлена кодировка консоли – *Cp866*. Эта кодировка является стандартной только для Windows. В *NIX необходимо устанавливать кодировку, которая поддерживается системой. Чаще всего это *KOI8-R* или *UTF-8*. Последняя, на мой взгляд, предпочтительнее.

Теперь сконфигурируем что-нибудь посложнее. Консоль уберем, добавим вывод в несколько файлов, для разных категорий и с разными уровнями.

В XML:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration debug="false" xmlns:log4j="http://jakarta.apache.org/log4j/">

    <appender name="bulk" class="org.apache.log4j.DailyRollingFileAppender">
        <param name="File" value="bulk.log"/>
        <param name="Append" value="true"/>
        <param name="DatePattern" value="'.'yyyy-MM-dd'.log'"/>
        <param name="Encoding" value="UTF-8"/>
        <layout class="org.apache.log4j.EnhancedPatternLayout">
            <param name="ConversionPattern" value="%d{ISO8601} [%-5p][%-16.16t][%30c] - %m%n"/>
        </layout>
    </appender>

    <appender name="application" class="org.apache.log4j.RollingFileAppender">
        <param name="File" value="application.log"/>
        <param name="MaxFileSize" value="100MB"/>
        <param name="MaxBackupIndex" value="10"/>
        <param name="Encoding" value="UTF-8"/>
        <layout class="org.apache.log4j.EnhancedPatternLayout">
            <param name="ConversionPattern" value="%d{ISO8601} [%-5p][%-16.16t][%30c{-2}] - %m%n"/>
        </layout>
    </appender>

    <appender name="orm" class="org.apache.log4j.DailyRollingFileAppender">
        <param name="File" value="orm.log"/>
        <param name="DatePattern" value="'.'yyyy-MM-dd'.log'"/>
        <param name="Encoding" value="UTF-8"/>
        <layout class="org.apache.log4j.TTCCLayout">
            <param name="DateFormat" value="ABSOLUTE"/>
            <param name="ContextPrinting" value="false"/>
        </layout>
    </appender>

    <logger name="ru.skipy">
        <level value="INFO"/>
        <appender-ref ref="application"/>
    </logger>

    <logger name="org.hibernate" additivity="false">
        <level value="WARN"/>
        <appender-ref ref="orm"/>
    </logger>

    <root>
        <priority value="WARN"/>
        <appender-ref ref="bulk"/>
    </root>

</log4j:configuration>
```

В виде свойств:

```
log4j.debug=false

log4j.rootLogger=WARN, bulk

log4j.appender.bulk=org.apache.log4j.DailyRollingFileAppender
log4j.appender.bulk.file=bulk.log
log4j.appender.bulk.append=true
log4j.appender.bulk.datePattern='.'yyyy-MM-dd'.log'
log4j.appender.bulk.layout=org.apache.log4j.EnhancedPatternLayout
log4j.appender.bulk.layout.conversionPattern=%d{ISO8601} [%-5p][%-16.16t][%30c] - %m%n

log4j.appender.application=org.apache.log4j.RollingFileAppender
log4j.appender.application.file=application.log
log4j.appender.application.file.MaxBackupIndex=10
log4j.appender.application.file.MaxFileSize=100MB
log4j.appender.application.layout=org.apache.log4j.EnhancedPatternLayout
log4j.appender.application.layout.conversionPattern=%d{ISO8601} [%-5p][%-16.16t][%20c{-2}] - %m%n

log4j.appender.orm=org.apache.log4j.DailyRollingFileAppender
log4j.appender.orm.file=orm.log
log4j.appender.orm.datePattern='.'yyyy-MM-dd'.log'
log4j.appender.orm.layout=org.apache.log4j.TTCCLayout
log4j.appender.orm.layout.dateFormat=ABSOLUTE
log4j.appender.orm.layout.contextPrinting=false

log4j.logger.ru.skipy=INFO, application

log4j.logger.org.hibernate=WARN, orm

log4j.additivity.org.hibernate=false
```

Что тут сделано:

- Создано три аппендера – *bulk*, *application* и *orm*. Они выводят сообщения в разные файлы, в разных форматах. Например, у *application* при выводе обрезаются два начальных элемента имени категории.
- Сконфигурированы логгеры – *ru.skipy* выводится в *application*, *org.hibernate* – в *orm*. У *org.hibernate* флаг *additivity* установлен в `false`.
- Сконфигурирован корневой логгер, использующий аппендер *bulk* и имеющий уровень `WARN`.

Комментарии:

- Обратите внимание, как выставляется флаг *additivity* в случае файла свойств и в XML. Это как раз пример той неочевидности, из-за которой я предпочитаю XML – там принадлежность этого свойства не вызывает сомнений.

Перевожу данную конфигурацию на русский язык.

- Все сообщения из своего приложения – имя категории начинается на *ru.skipy* – я хочу выводить в файл *application.log*. Именно поэтому в аппендере я могу обрезать первые две части имени категории – они **всегда** будут *ru.skipy*, и мне незачем загромождать лог ненужной информацией. Раз в сутки я хочу создавать новый файл, для удобства анализа и архивации.
- Все сообщения от всего исполняемого кода я хочу выводить в *bulk.log*. Этот файл я буду обрезать по размеру. Сюда же будут попадать и сообщения от моего кода, т.к. аппендер унаследован от корня. Это, в общем-то, логично – помимо анализа работы собственно моего кода мне необходимо смотреть на все приложение в целом. Уровень всех сообщений я выставляю в `WARN` при конфигурации корневого логгера, он будет действовать на все сообщения, кроме моих (у них используется сконфигурированный для *ru.skipy* уровень `INFO`).
- Временами мне необходимо смотреть, что происходит при работе с базой – какие запросы генерируются, какие результаты возвращаются. При этом будет использоваться уровень `DEBUG`, т.е. информации будет очень много. Чтобы она не загромождала основной лог, я выставляю флаг *additivity* в `false`. По умолчанию уровень выставлен в `WARN`, что минимизирует временны*е* потери на лог.

* * *

Продолжать приводить примеры можно долго, но смысла нет. Общий принцип организации лога следующий. Путем рассуждений, подобных только что приведенным, решите:

1. какую информацию вы хотите видеть
2. как вы хотите группировать или разделять эту информацию
3. в каком формате вам нужна информация
4. как вы хотите архивировать информацию за прошлые периоды, и хотите ли вообще, какой максимальный объем логов вы готовы хранить единовременно

После того, как вы определитесь со всеми этими вопросами – можете писать конфигурацию.

И еще один вопрос, который иногда вполне закономерно возникает. Можно ли сконфигурировать вывод так, чтобы в один аппендер сообщения от определенного логгера выводились на одном уровне, а в другой на другом?

Можно. Вариантов два. Первый – параметр аппендера *Threshold*, который содержит минимальный уровень сообщений для этого аппендера. Конфигурируется он соответственно:

```
<appender name="bulk" class="org.apache.log4j.DailyRollingFileAppender">
    <param name="Threshold" value="WARN"/>
    <!-- остальная конфигурация -->
</appender>
```

```
log4j.appender.bulk.threshold=WARN
```

И второй вариант – использование фильтров. У аппендера можно создать фильтр (на самом деле, любую цепочку), который, в числе прочего, может фильтровать по уровню лога (`org.apache.log4j.varia.LevelRangeFilter`). Подробно останавливаться на фильтрах не буду, желающие могут посмотреть описание и примеры использования, например, *mym*. Пример конфигурирования:

```
<appender name="bulk" class="org.apache.log4j.DailyRollingFileAppender">
    <filter class="org.apache.log4j.varia.LevelRangeFilter">
        <param name="LevelMin" value="warn"/>
        <param name="LevelMax" value="fatal"/>
        <param name="AcceptOnMatch" value="true"/>
    </filter>
</appender>
```

```
log4j.appender.bulk.filter.a=org.apache.log4j.varia.LevelRangeFilter
log4j.appender.bulk.filter.a.LevelMin=WARN
log4j.appender.bulk.filter.a.LevelMax=FATAL
log4j.appender.bulk.filter.a.AcceptOnMatch=TRUE
```

Порядок фильтров для XML-конфигурации определяется порядком их описания (согласно DTD фильтры конфигурируются *после layout-ов*). В случае properties-конфигурации фильтры *сортируются по имени*.

Пожалуй, о конфигурации сказано достаточно. Основные аппендеры и компоновщики я описал, примеры привел. Соответственно, можно перейти к следующей части.

Использование в программном коде

Начнем с инициализации.

Инициализация *Log4J*

В простейших случаях об этом можно не заботиться. Как уже *упоминалось*, конфигурационные файлы ищутся в `classpath`, сначала XML-вариант, потом файл свойств. И в довольно большом количестве случаев этого достаточно.

Бывают, однако, исключения. Представьте себе, что у вас есть несколько веб-приложений. Каждое из них имеет свой загрузчик классов. А конфигурация лога у всех одинаковая. Вариантов два – либо класть файл конфигурации в каждый веб-архив – что особенно удобно, когда надо обновить конфигурацию лога! – либо держать его в какой-то одной точке, где можно его менять и откуда все приложения будут его читать.

На все такие случаи *Log4J* предоставляет средства для инициализации вручную. Для этого применяются классы `org.apache.log4j.PropertyConfigurator` и `org.apache.log4j.xml.DOMConfigurator`. Эти классы имеют несколько статических методов для инициализации, если быть точным, то их по пять (в версии 1.2.16). Четыре из них совпадают по сигнатурам:

- `configure(String configFilename)` – инициализирует *Log4J* на основании указанного имени файла
- `configure(java.net.URL configURL)` – инициализирует *Log4J* на основании указанного URL файла
- `configureAndWatch(String configFilename)` – инициализирует *Log4J* на основании указанного имени файла и отслеживает изменения, при которых файл перезагружается и происходит переинициализация. Период отслеживания изменений составляет 60 секунд (значение константы `org.apache.log4j.helpers.FileWatchdog.DEFAULT_DELAY`).
- `configureAndWatch(String configFilename, long delay)` – инициализирует *Log4J* на основании указанного имени файла и отслеживает изменения, при которых файл перезагружается и происходит переинициализация. Период отслеживания изменений задается вторым параметром в методе.

Последний метод специфичен для форматов данных. Он в обоих случаях называется `configure`, однако в `org.apache.log4j.PropertyConfigurator` он принимает параметр типа `java.util.Properties`, а в `org.apache.log4j.xml.DOMConfigurator` – типа `org.w3c.dom.Element`. Оба эти метода предназначены для создания конфигурации вручную, в программном коде.

Т.е. вся инициализация *Log4J* по большому счету сводится к одной единственной строчке:

```
DOMConfigurator.configure("./log4j.xml");
```

Советую также посмотреть документацию по классам *`org.apache.log4j.PropertyConfigurator`* и *`org.apache.log4j.xml.DOMConfigurator`*, ибо я не ставил целью ее копировать и некоторые моменты сознательно опустил.

Теперь об использовании логгеров в коде.

Использование логгеров

Инициализация логгера производится тривиально, тоже статическими методами:

- `Logger.getLogger(Class clazz)` – инициализация по классу. Наиболее распространенный вариант. Имя категории принимается равным полному имени класса, собственно, внутри вызывается второй метод инициализации с параметром `clazz.getName()`.
- `Logger.getLogger(String categoryName)` – более общий вариант. Возвращается логгер по указанному имени, которое в общем случае может не совпадать с именем класса. Этот метод может быть полезен, если есть необходимость в разных классах выводить сообщения в одной категории.

Как правило, логгеры получаются в каждом классе, где они нужны, по одному на класс:

```
private static final Logger logger = Logger.getLogger(MyClass.class);
```

Это позволяет наиболее эффективно конфигурировать уровень лога и разводить сообщения по разным файлам. Если же есть необходимость, например, все относящиеся к работе с БД сообщения конфирурировать синхронно, вне зависимости от того, где они генерируются – можно ввести одну категорию и получить под нее собственный логгер:

```
private static final Logger logger = Logger.getLogger(MyClass.class);
private static final Logger dbLogger = Logger.getLogger("ru.skipy.DB");
```

И в этом случае конфигурация для этой категории не будет пересекаться с конфигурацией для остального лога данного класса. Так сделано, например, в Hibernate, с категорией `org.hibernate.SQL`, которая используется для вывода логов, связанных с SQL, из любой точки библиотеки. Но такой подход достаточно редок.

> На всякий случай уточняю: имя категории совсем не обязательно должно начинаться с имени пакета – `ru.skipy`. Точно так же я мог указать в качестве имени `my_favorite_DB_category` или `my.favorite.DB.category` – все эти варианты будут работать идентично.

Использование логгера в коде тоже не вызывает сложностей – просто вызывается соответствующий метод:

```
logger.info("Starting mass rate charge calculation...");
```

В случае обработки исключения – вторым параметром передается оно:

```
try{
    // код, вызывающий исключение
}catch(RateCalculationException ex){
    logger.error("Error while calculating rate change!", ex);
}
```

Есть, однако, тонкость, связанная с использованием уровней DEBUG и TRACE. Дело тут в двух моментах:

1. Количество данных, выводимых на уровне DEBUG и, тем более, TRACE, достаточно велико. Причем выводятся, как правило, не только сообщения, но и множество параметров внутреннего состояния, в чем, собственно, и заключается смысл такого подробного лога:

   ```
   logger.debug("Starting rate charge calculation for account " + accountNum + " with parameters: rest=" + rest +
               ", percent=" + percent + " period=" + period);
   ```

   Тут параметров выводится всего четыре, в реальной жизни их могут быть десятки.
2. Вывод лога на уровне DEBUG и, тем более, TRACE, как правило, **запрещен**.

То есть что получается. Мы *сначала* формируем сообщение, а о том, что лог запрещен, узнаем только *потом*. Когда время на создание сообщения уже потрачено. Мы можем использовать StringBuilder, MessageFormat и что угодно еще – кардинально ситуацию это не изменит. Мы в любом случае тратим достаточно времени впустую. А сообщений на уровне DEBUG выводится много.

Так вот, чтобы этого избежать, в логгере существуют два метода – isDebugEnabled() и isTraceEnabled(). Они позволяют *сначала* проверить, имеет ли смысл вообще создавать сообщение, и только *потом* тратить на это время, когда понятно, что оно будет потрачено не зря. Т.е. фрагмент кода, приведенный выше, должен выглядеть так:

```
if (logger.isDebugEnabled()){
    logger.debug("Starting rate charge calculation for account " + accountNum + " with parameters: rest=" + rest +
                ", percent=" + percent + ", period=" + period);
}
```

Использование такой конструкции **крайне рекомендуется**. И то же самое относится к TRACE.

Больше, пожалуй тонких моментов я не встречал. Если знаете – пишите, с удовольствием включу в статью.

Переходим к следующему разделу.

NDC – Nested Diagnostic Context

За работу с *NDC* отвечает класс по имени org.apache.log4j.NDC. Использовать его очень просто. Для добавления информации в контекст вызывается статический метод NDC.push(String). Передаваемый параметр помещается на вершину стека, откуда его можно – и нужно! – удалить при помощи другого статического метода – NDC.pop(). Эти методы необходимо применять парно, чтобы не вызывать перекосов (накапливания или исчерпания стека).

Еще одно правило, исполнения которого требует *Log4J* – при завершении работы в потоке вызывать метод NDC.remove(). В принципе, ничего страшного не произойдет, если вы этого не сделаете, однако контекст останется присоединеным к потоку, и если вы "забудете" там данные – они попадут в следующий запрос, обрабатываемый этим потоком, запутав вас еще больше.

Таким образом, использование *NDC* выглядит примерно так (на примере doGet сервлета):

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException{
    NDC.push(getUserName());
    // тут выполняем какие-то действия
    doWork(); // вызываем метод
    // тут выполняем какие-то действия
    NDC.pop();
    NDC.remove();
}

private void doWork(){
    NDC.push(getPermissions());
    // тут выполняем какие-то действия
    NDC.pop();
}
```

Для вывода *NDC* в лог можно использовать либо *TTCCLayout*, либо *PatternLayout* и опцию %x. В логе в результате появится стек в виде строк, разделенных пробелами (использован формат %d{ISO8601} %p [%x] %m%n):

```
NDC.push("aaa");
NDC.push("bbb");
NDC.push("ccc");
logger.info("My message");
```

```
2010-05-27 22:52:15,909 INFO [aaa bbb ccc] My message
```

Позиционное форматирование на опцию %x действует. При этом строка, представляющая стек, обрезается с начала. Т.е. если ограничить в примере выше длину *NDC* 10-ю символами (%.10x), то в лог будет выведена строка aa bbb ccc. Таким образом, всегда видна вершина стека.

Желающие могут более подробно ознакомиться с *NDC* вот тут: *http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/NDC.html*. А мы переходим ко второму типу диагностического контекста –

MDC – Mapped Diagnostic Context

За работу с *MDC* отвечает класс по имени org.apache.log4j.MDC. Использовать его не сложнее, чем *NDC*. У *MDC* есть следующие статические методы:

- put(String,Object) – помещает в контекст объект под соответствующим именем.
- remove(String) – удаляет из контекста объект под соответствующим именем
- get(String) – получает из контекста объект под соответствующим именем (не удаляя)
- clear() – очищает контекст

Принципы использования *MDC* те же, что и у *NDC*. put и remove используются парами, перед завершением работы в потоке вызывается clear. Пример кода даже приводить не буду.

За вывод *MDC* в лог отвечает опция *%X*. На нее также распространяется *позиционное форматирование*, и строка обрезается тоже сначала. Однако, поскольку порядок элементов в строковом представлении для *MDC* не определен – предсказать, что именно будет видно в логе, невозможно. Формат вывода в лог следующий (используется шаблон %d{ISO8601} %p [%X] %m%n):

```
MDC.put("key1", "value1");
MDC.put("key2", "value2");
MDC.put("key3", "value3");
logger.info("My message");
```

```
2010-05-27 23:27:52,636 INFO [{{key3,value3}{key2,value2}{key1,value1}}] My message
```

Хочу еще раз *напомнить*, что %X работает только при использовании *EnhancedPatternLayout*, обычный *PatternLayout* требует указания ключа и не может вывести контекст целиком. Разумеется, *EnhancedPatternLayout* с ключами тоже умеет работать.

И последняя тема –

Управление во время исполнения

Эта часть посвящена исполнению озвученной в начале *мечты* – *"идеальной была бы ситуация, когда количество выводимых данных можно было бы менять как нужно, и в тот момент, когда нужно"*. При минимальных усилиях можно добиться и этого.

Отталкиваться будем от того, что каждому логгеру можно выставить уровень. В программном коде. Использовать для этого надо метод логгера `setLevel(org.apache.log4j.Level)`.

Метод есть – это хорошо, полдела сделано. Однако его еще надо как-то вызвать. Мне в этом отношении ближе всего использование технологии JMX – *Java Management Extension*. Подробно в нее погружаться не буду, из примера все понятно.

Я еще несколько упрощу задачу – буду считать, что имя категории, которой надо менять уровень, известно. В принципе, это не лишено смысла. Достаточно редко бывает необходимость изменить уровень произвольного логгера, чаще это запросы типа "а давайте-ка посмотрим, что у нас творится в работе с БД". А работа с БД – это набор логгеров. И проще где-то его прописать, чем заставлять конечного пользователя вводить все эти длинные и бесссмысленные на его взгляд имена.

В общем, тестовое приложение устроено так. Создается один management bean с функциями установки уровня – от *TRACE* до *FATAL*. Реализация этих функций устанавливает предопределенному логгеру этот уровень. Еще раз повторюсь, ничего не мешает передать имя логгера в приложение, если в этом возникнет необходимость. Его просто надо будет сделать параметром вызываемой функции.

Соединение с приложением и вызов функций можно выполнять из любого JMX-клиента. В Java Runtime Environment такой клиент тоже есть, это JConsole. Он позволяет подключиться к виртуальной машине и в числе прочего вызывать ее JMX-функции. Чем мы и воспользуемся.

Всего кода я тут приводить не буду, только отдельные фрагменты.

Определение интерфейса для JMX:

```
/**
 * MBean interface definition
 */
public static interface Log4JManagerMBean {

    /**
     * Setting log level to <code>TRACE</code>
     */
    public void setTraceLevel();

    // для остальных уровней аналогично
}
```

Реализация этого интерфейса:

```
/**
 * MBean implementation
 */
public static class Log4JManager implements Log4JManagerMBean{

    /**
     * Returns instance of the logger for category <code>ru.skipy.logging</code>
     *
     * @return logger instance
     */
    private Logger getLogger(){
      return LogManager.getLogger("ru.skipy.logging");
    }

    /**
     * Setting log level to <code>TRACE</code>
     */
    @Override
    public void setTraceLevel() {
      getLogger().setLevel(Level.TRACE);
    }

    // для остальных уровней аналогично
}
```

И регистрация созданного management bean-а:

```
public static final String OBJECT_NAME = "ru.skipy.logging.jmx:type=Log4J JMX Manager";

/**
 * Initializes MBean with the object name {@link #OBJECT_NAME}
 *
 * @throws Exception if any error occures
 */
private static void initMBean() throws Exception{
    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
    ObjectName name = new ObjectName(OBJECT_NAME);
    Log4JManager mbean = new Log4JManager();
    mbs.registerMBean(mbean, name);
}
```

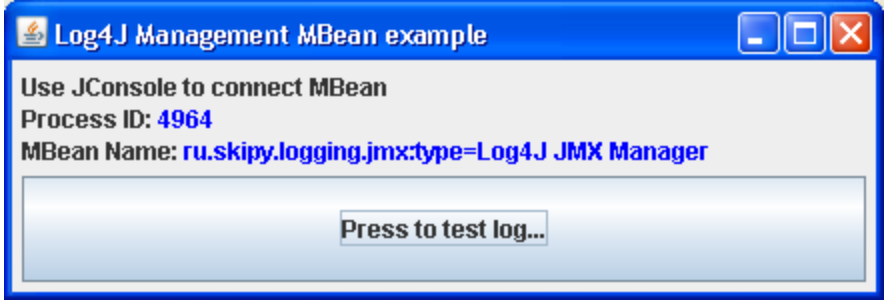
А дальше – можно выводить сообщения на всех уровнях и смотреть на результат.

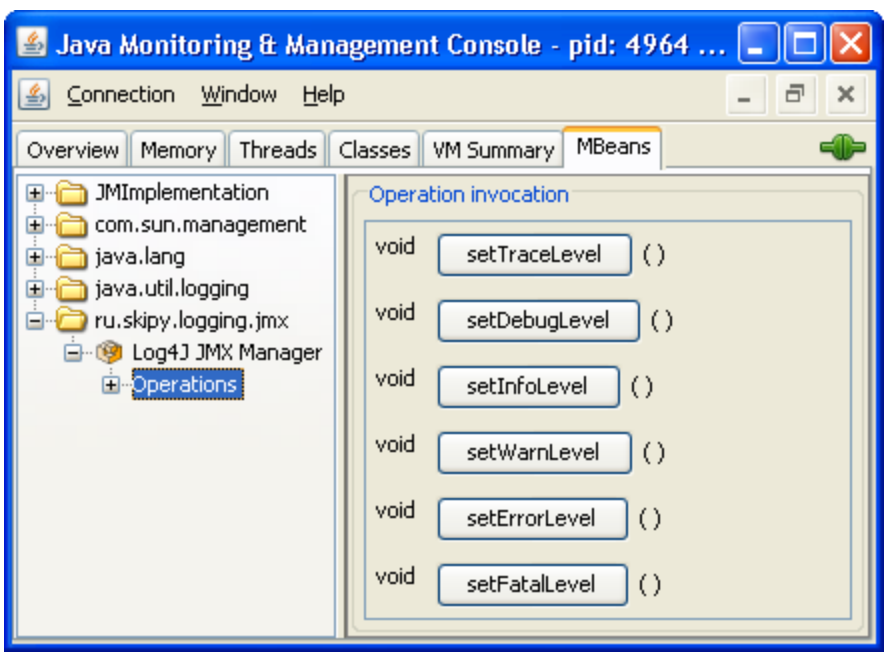
```
/**
 * Testing all log levels
 */
private void test() {
    System.out.println("Testing all levels:\n====================");
    System.out.println("Trace: ");
    if (logger.isTraceEnabled()){
        logger.trace("TRACE message");
    }
    System.out.println("Debug: ");
    if (logger.isDebugEnabled()){
        logger.debug("DEBUG message");
    }
    System.out.println("Info: ");
    logger.info("INFO message");
    System.out.println("Warn: ");
    logger.warn("WARN message");
    System.out.println("Error: ");
    logger.error("ERROR message");
    System.out.println("Fatal: ");
    logger.fatal("FATAL message");
    System.out.println("Done\n====================");
}
```

Собственно, больше ничего необычного тут нет. Полный код примера вместе с файлом сборки `ant` можно *скачать*. Для компиляции и запуска потребуется еще сама библиотека *log4j-1.2.16.jar*. Предпочтительно ее, конечно, скачать с сайта *Log4J* (*http://logging.apache.org/log4j/1.2/download.html*), если такой возможности нет – можно взять *мут*.

Краткая инструкция по запуску примера.

Скачиваем архив, распаковываем, собираем. Команда для запуска примера – `ant run`. Появится окно, которое сейчас находится слева. Обратите внимание на значение *Process ID* – оно необходимо для надежной идентификации процесса в JConsole.





Далее запускаем JConsole. Это приложение входит в поставку Java SDK, находится в директории `<JSDK_INSTALL_DIR>/bin`. Появляется *окно логина*. Находим в нем процесс с указанным ID, соединяемся (жмем кнопку Connect).

Последняя закладка JConsole – MBeans. Находим там management bean, соответствующий имени, которое мы указали в коде – *ru.skipy.logging.jmx:type=Log4J JMX Manager*. Открываем в дереве ветку *Operations* и видим все операции, которые мы определили для изменения уровня.

Снимок окна JConsole приведен слева. В его правой части виден список операций. Для выполнения соответствующей операции достаточно нажать на кнопку с ее именем. Собственно, в этом и заключается смысл примера – поочередно устанавливать различные уровни лога, нажимать *"Press to test log..."* в тестовом приложении и наблюдать, как изменится вывод в лог.

Ниже приведен пример вывода в консоль после нажатия кнопки в приложении в изначальном состоянии (сразу после запуска), и дальше – после установки уровней `TRACE` и `FATAL`. Изначально был установлен уровень `INFO`, соответственно, сообщения на уровнях `TRACE` и `DEBUG` видны не были. Далее установили уровень `TRACE` – стали видны все сообщения. После установки уровня `FATAL` видно только сообщение на этом уровне, остальные сообщения до консоли не дошли.

```
[java] Testing all levels:
[java] ====================
[java] Trace:
[java] Debug:
[java] Info:
[java] 2010-05-30 13:14:09,720 [INFO ] [AWT-EventQueue-0    ] [ru.skipy.logging.jmx.Main] - INFO message
[java] Warn:
[java] 2010-05-30 13:14:09,720 [WARN ] [AWT-EventQueue-0    ] [ru.skipy.logging.jmx.Main] - WARN message
[java] Error:
[java] 2010-05-30 13:14:09,720 [ERROR] [AWT-EventQueue-0    ] [ru.skipy.logging.jmx.Main] - ERROR message
[java] Fatal:
[java] 2010-05-30 13:14:09,720 [FATAL] [AWT-EventQueue-0    ] [ru.skipy.logging.jmx.Main] - FATAL message
[java] Done
[java] ====================
[java] Testing all levels:
[java] ====================
[java] Trace:
[java] 2010-05-30 13:14:17,798 [TRACE] [AWT-EventQueue-0    ] [ru.skipy.logging.jmx.Main] - TRACE message
[java] Debug:
[java] 2010-05-30 13:14:17,798 [DEBUG] [AWT-EventQueue-0    ] [ru.skipy.logging.jmx.Main] - DEBUG message
[java] Info:
[java] 2010-05-30 13:14:17,798 [INFO ] [AWT-EventQueue-0    ] [ru.skipy.logging.jmx.Main] - INFO message
[java] Warn:
[java] 2010-05-30 13:14:17,798 [WARN ] [AWT-EventQueue-0    ] [ru.skipy.logging.jmx.Main] - WARN message
[java] Error:
[java] 2010-05-30 13:14:17,798 [ERROR] [AWT-EventQueue-0    ] [ru.skipy.logging.jmx.Main] - ERROR message
[java] Fatal:
[java] 2010-05-30 13:14:17,798 [FATAL] [AWT-EventQueue-0    ] [ru.skipy.logging.jmx.Main] - FATAL message
[java] Done
[java] ====================
[java] Testing all levels:
[java] ====================
[java] Trace:
[java] Debug:
[java] Info:
[java] Warn:
[java] Error:
[java] Fatal:
[java] 2010-05-30 13:14:21,579 [FATAL] [AWT-EventQueue-0    ] [ru.skipy.logging.jmx.Main] - FATAL message
[java] Done
[java] ====================
```

Вот так, почти что тривиальный код позволяет эффективно управлять выводим в лог, не останавливая приложения. Более того, мы можем реализовать возможность указывать имя логгера, уровень вывода для которого мы хотим изменить. И тогда мы сможем установить вообще любой уровень для любого логгера. Гибкость максимальная, удобство обсуждаемо.

* * *

Послесловие

Вот, пожалуй, и все, что я хотел рассказать о логировании как явлении вообще и о *Log4J* в частности. Я не затронул довольно много тем – собственные обработчики ошибок, фильтры, преобразователи (renderers) объектов, переинициализацию всей системы логирования во время исполнения, использование системных и собственных свойств в файле свойств *Log4J*. Я не углублялся в другие фреймворки, хотя, например, *Logback* тоже весьма интересен, тем более, что он представляет собой развитие *Log4J*.

Я не сделал всего этого по той причине, что где-то все-таки надо провести логическую черту. Того, что я рассказал, должно с лихвой хватить для освоения *Log4J* в частности и понимания принципов логирования вообще. Если кому-то интересно – после этого материала он вполне сможет продолжить изучение самостоятельно. Я же на текущий момент считаю свою задачу выполненной.

Всем спасибо! Надеюсь, это было полезно.

P.S. Для вопросов, комментариев и обсуждения создана страница в блоге: *http://skipy-ru.livejournal.com/2980.html*.

---

1. Слово *log* в английском имеет множество значений, в том числе *протокол* или *журнал*. В русском успешно прижилось само слово *лог* и производные от него *логирование* и *логировать*. Хотя правильнее было бы говорить *журналирование (протоколирование)* и *вести журнал (протокол)* соответственно. В этой статье я буду употреблять прижившиеся термины.

2. На самом деле я их знаю больше, чем три. Есть еще *LogKit*, бывший частью проекта *Apache Avalon*, есть еще логгер в самом *Avalon*, который тоже является зонтичным и является уровнем абстракции для *LogKit*, *java.util.logging* и *Log4J*. Есть *Lumberjack* – фреймворк для логирования в Java версии до 1.4 И т.д. и т.п. На раннем этапе, когда ниша логгеров не была занята, разрабатывалось много разных фреймворков. Однако практически все они уже отмерли, потому я их даже не упоминаю.

3. *NOP* (*No Op*) – сокращение от*No Operation*. Такой логгер не осуществляет никакого вывода информации – все его методы имеют пустую реализацию. Нужен для того, чтобы минимизировать затраты на логирование, когда оно не нужно.

4. Вот, например, неплохая статья по этому поводу: *http://articles.qos.ch/classloader.html*.

5. Последняя версия – 1.2.16 – вышла 7 апреля 2010, 1.2.15 – 30 августа 2007, 1.2.14 – 18 сентября 2006. Информация из архива версий Log4J: *http://archive.apache.org/dist/logging/log4j/*.

6. Желающие могут посмотреть DTD тут: *http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/xml/doc-files/log4j.dtd*.

В начало