История логов в Java

JSP & Servlets
5 уровень, 1 лекция

ОТКРЫТА

2.1 Первый логгер – log4j

Как ты уже знаешь, история логов началась с `System.err.println()` – вывод записи в консоль. Его и сейчас активно используют при дебаге, например, Intellij IDEA с помощью него выводит сообщения об ошибках в консоль. Но никаких настроек у этого варианта нет, так что пойдем дальше.

Первый и самый популярный логгер назывался `Log4j`. Это было хорошее и гибко настраиваемое решение. В силу разных обстоятельств это решение так и не попало в JDK, чем очень расстроило все комьюнити.

Этот логгер не просто умел логировать, он был создан программистами для программистов и позволял им решать проблемы, которые постоянно возникали в связи с логированием.

Как вы уже знаете, логи пишутся в конечном итоге для того, чтобы какой-то человек их читал и пытался понять, что же произошло во время работы программы – что и когда пошло не так как ожидалось.

В `log4j` для этого были три вещи:

- **логирование подпакетов;**
- **множество appender'ов (результатов);**
- **горячая перезагрузка настроек.**

Во-первых, настройки `log4j` можно было прописать таким образом, чтобы включить логирование в одном пакете и выключить в другом. Например, можно было включить логирование в пакете `com.javarush.server`, но при этом выключить его в `com.javarush.server.payment`. Это позволяло быстро убрать из лога ненужную информацию.

Во-вторых, `log4j` позволял писать результаты логирования сразу в несколько лог-файлов. И вывод в каждый можно было настроить индивидуально. Например, в один файл можно было писать только информацию о серьезных ошибках, в другой – логи из определенного модуля, в а третий – логи за определенное время.

Каждый лог-файл, таким образом был настроен на определенный тип ожидаемых проблем. Это очень упрощало жизнь программистам, которым не доставляло удовольствия просматривать вручную гигабайтные файлы логов.

И наконец, в-третьих, `log4j` позволял изменить настройки лога прямо во время работы программы, не перезапуская ее. Это было очень удобно, когда нужно было подправить работу логов, чтобы найти дополнительную информацию по определенной ошибке.

Важно! Есть две версии лога `log4j`: **1.2.x** и **2.x.x**, которые несовместимы друг с другом.

Подключить логгер в проект можно с помощью кода:

```
1  <dependencies>
2    <dependency>
3      <groupId>org.apache.logging.log4j</groupId>
4      <artifactId>log4j-api</artifactId>
5      <version>2.17.2</version>
6    </dependency>
7
8    <dependency>
9      <groupId>org.apache.logging.log4j</groupId>
```

```
    <artifactId>log4j-core</artifactId>
    <version>2.17.2</version>
  </dependency>
</dependencies>
```

2.2 Первый официальный логгер – JUL: java.util.logging

После того как в Java-сообществе появился зоопарк логгеров, разработчики `JDK` решили сделать один стандартный логгер, которым бы пользовались все. Так появился логгер `JUL`: пакет `java.util.logging`.

Однако, при его разработке создатели логгера взяли за основу не `log4j`, а вариант логгера от IBM, что повлияло на его развитие. Хорошая новость – логгер `JUL` входит в состав `JDK`, плохая – им мало кто пользуется.

Мало того, что разработчики `JUL` сделали **«еще один универсальный стандарт»**, так они сделали для него свои собственные уровни логирования, которые отличались от принятых в то время у популярных логгеров.

И это было большой проблемой. Ведь продукты на `Java` зачастую собраны из большого количества библиотек и в каждой такой библиотеке был свой собственный логгер. Значит нужно было конфигурировать все логгеры которые есть в приложении.

Хотя сам по себе логгер довольно неплох. Создание логгера более-менее похожее. Для этого нужно сделать импорт:

```
java.util.logging.Logger log = java.util.logging.Logger.getLogger(LoggingJul.class.getName());
```

Имя класса специально передается для того, чтобы знать, откуда идет логирование.

Только с выходом разработчики решили важные проблемы, после чего `JUL` по-настоящему удобно использовать. До этого это был какой-то второразрядный логгер.

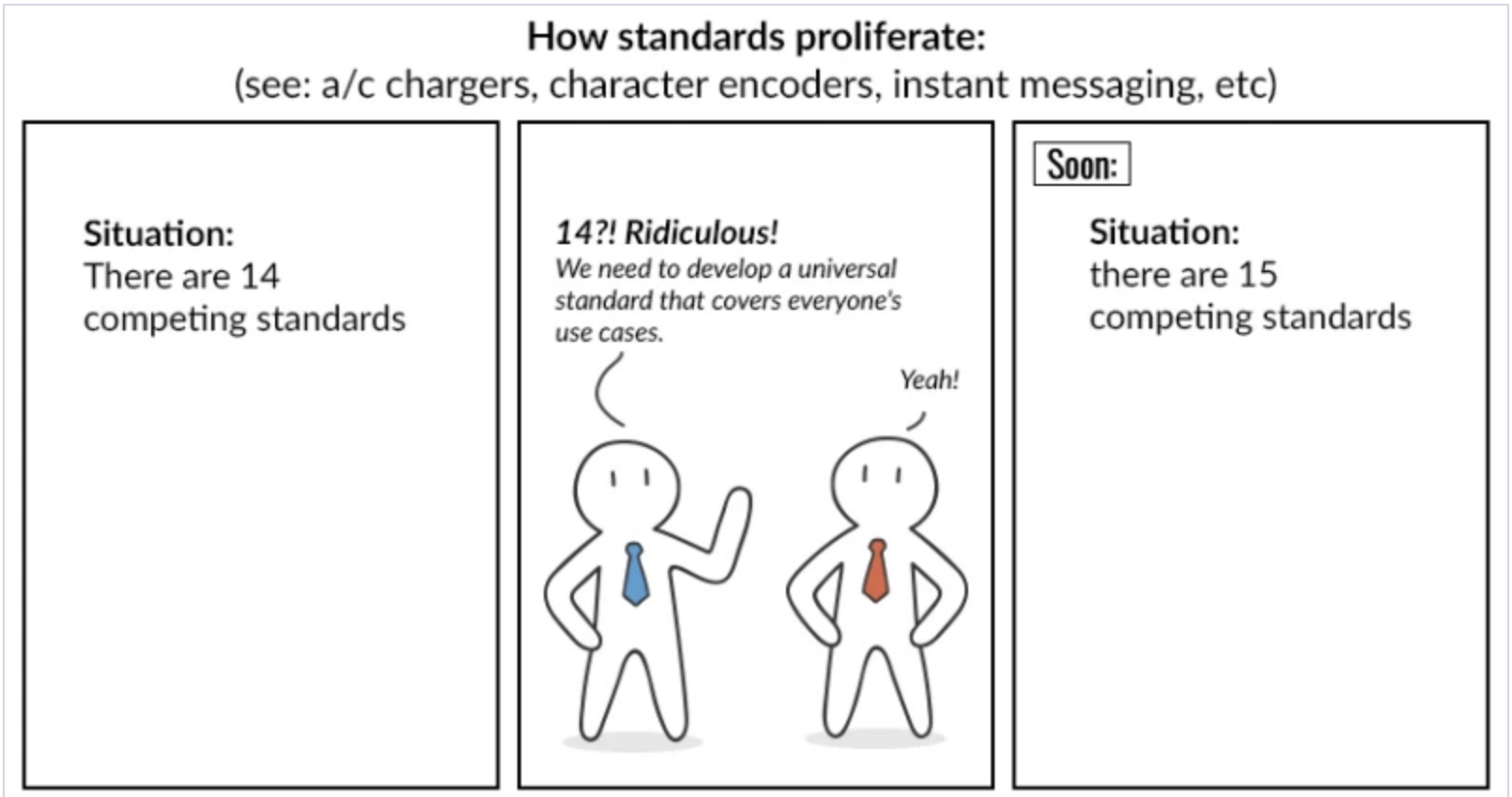
Также этот логгер поддерживает лямбда-выражения и ленивые вычисления. Начиная с `Java 8`, в него можно передавать `Supplier<String>`. Это помогает считать и создавать строку только в тот момент, когда это действительно нужно, а не каждый раз, как это было ранее.

Методы с аргументом `Supplier<String> msgSupplier` выглядят как показано ниже:

```
public void info(Supplier msgSupplier) {
   log(Level.INFO, msgSupplier);
}
```

2.3 Первый логгер-обертка – JCL: jakarta commons logging

Долгое время не было единого стандарта среди логгеров, `JUL` должен был стать таким, но он был хуже `log4j`, поэтому единого стандарта так и не появилось. Зато появился целый зоопарк логгеров, каждый из которых хотел стать тем самым.



Однако обычным Java-разработчикам не нравилось, что почти у каждой библиотеки есть свой собственный логгер и его нужно как-то по-особенному конфигурировать. Поэтому сообщество решило создать специальную обертку над другими логгерами – так появился `JCL: jakarta commons logging`

И опять-таки проект, который создавался, чтобы быть лидером, не стал им. Нельзя создать победителя, победителем можно только стать. Функциональность `JCL` была очень бедной и никто не хотел им пользоваться. Логгер, созданный, чтобы стать заменой всех логгеров, постигла такая же судьба, как и `JUL` – им не пользовались.

Хотя он был добавлен во многие библиотеки, выпускаемые сообществом Apache, но зоопарк логгеров только разрастался.

2.4 Первый последний логгер – Logback

Но и это еще не все. Разработчик `log4j` решил, что он самый умный (ну ведь его логгером пользовалось больше всего людей) и решил написать новый улучшенный логгер, который будет сочетать в себе плюсы `log4j` и других логгеров.

Новый логгер носил название `Logback`. Именно этот логгер должен был стать будущим единым логгером, которым бы пользовались все. В основе была та же идея, что и в `log4j`.

Подключить в проект этот логгер можно с помощью кода:

```
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.2.6</version>
</dependency>
```

Отличия были в том, что в `Logback`:

- улучшена производительность;
- добавлена нативная поддержка `slf4j`;
- расширена опция фильтрации.

Еще одним преимуществом этого логгера было то, что у него были очень хорошие настройки по умолчанию. И конфигурировать логгер нужно было, только если вы хотели что-то в них изменить. Также файл настроек был лучше адаптирован под корпоративный софт – все его конфигурации задавались в виде `xml/`.

По умолчанию `Logback` не требует каких-либо настроек и записывает все логи от уровня `DEBUG` и выше. Если вам нужно другое поведение, его можно настроить через `xml` конфигурацию:

```
<configuration>
    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file>app.log</file>
        <encoder>
            <pattern>%d{HH:mm:ss,SSS} %-5p [%c] - %m%n</pattern>
        </encoder>
    </appender>
    <logger name="org.hibernate.SQL" level="DEBUG" />
    <logger name="org.hibernate.type.descriptor.sql" level="TRACE" />
    <root level="info">
        <appender-ref ref="FILE" />
    </root>
</configuration>
```

2.5 Последний универсальный логгер – SLF4J: Simple Logging Facade for Java

Как же бывает долог путь поиска золотой середины…

В 2006 году один из создателей `log4j` вышел из проекта и решил еще раз попробовать создать универсальный логгер. Но на этот раз это был не новый логгер, а новый универсальный стандарт (обертка), который позволял взаимодействовать различным логгерам вместе.

Этот логгер назвали `slf4j – Simple Logging Facade for Java`, он был оберткой вокруг `log4j`, `JUL`, `common-loggins и logback`. Этот логгер решал реальную проблему – управление зоопарком логгеров, поэтому все стали сразу им пользоваться.

Мы героически решаем проблемы, которые сами себе и создаем. Как видим, прогресс дошел до того, что создали обертку над оберткой…

Сама обертка состоит из двух частей:

- `API`, который используется в приложениях;
- Реализации, которые добавляются в виде отдельных зависимостей для каждого логгера.

Подключить логгер в проект можно с помощью кода:

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.17.2</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.17.2</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-slf4j-impl</artifactId>
    <version>2.17.2</version>
</dependency>
```

Достаточно подключить правильную реализацию и все: весь проект будет работать с ней.

2.6 Оптимизация в slf4j

`Slf4j` поддерживает все новые функции, такие как форматирование строк для логирования. До этого была такая проблема. Допустим, ты хочешь вывести в лог сообщение:

```
log.debug("User " + user + " connected from " + request.getRemoteAddr());
```

С этим кодом есть проблема. Допустим, ваше приложение работает на `production` и не пишет в лог никакие `DEBUG-сообщения`, однако метод `log.debug()` все равно будет вызван, а при его вызове будут вызваны и такие методы:

- `user.toString();`
- `request.getRemoteAddr();`

Вызов этих методов замедляет приложение. Их вызов нужен только во время дебага, но они все равно вызываются.

С точки зрения логики, эту проблему нужно было решать в самой библиотеке логирования. И в первой версии log4j решение было придумано:

```
1  if (log.isDebugEnabled()) {
2      log.debug("User " + user + " connected from " + request.getRemoteAddr());
3  }
```

Вместо одной строки для лога теперь нужно было писать три. Что резко ухудшило читабельность кода, и понизило популярность `log4j`.

Логгер `slf4j` смог немного улучшить ситуацию, предложив умное логирование. Выглядело оно так:

```
log.debug("User {} connected from {}", user, request.getRemoteAddr());
```

где `{}` обозначают вставки аргументов, которые передаются в методе. То есть первая `{}` соответствует user, вторая `{}` — `request.getRemoteAddr()`.

Эти параметры будут конкатенировать в единое сообщение только в случае, если уровень логирования позволяет записывать в лог. Не идеально, но лучше, чем все остальные варианты.

После этого `SLF4J` стал быстро расти в популярности, на данный момент это лучшее решение.

Поэтому будем рассматривать логирование на примере связки `slf4j-log4j12`.

‹ Предыдущая лекция

Следующая лекция ›

почему l4j в прошедшем времени?

Ответить

0

Андраник Мамиконян Уровень 69, Москва, Россия

24 июня, 17:23

Только с выходом ? разработчики решили важные проблемы, после чего JUL по-настоящему удобно использовать. До этого это был какой-то второразрядный логгер.
Что-то пропущено на месте ?, по контексту не удалось догадаться(

Ответить

+10