Logger

Интерфейс: org.slf4j.Logger

Описание

Абстрактный интерфейс для логирования сообщений, часть абстрактного API для логирования SLF4J, без привязки к конкретной реализации этого API.

Подключение библиотеки SLF4J

В составе JDK есть собственнный пакет логирования (`java.util.logging`), но у него есть недостаток: он привязан к одной конкретной реализации логирования, к той, которая содержится в самом JDK. Разработчики приложений могут предпочесть ей стороннюю библиотеку логирования, обладающую расширенными возможностями; для разработчиков же библиотек, использующих логирование, нежелательно привязываться к конкретной реализации логирования, а желательно оставлять выбор реализации за разработчиком приложения, использующего эту библиотеку.

SLF4J — это де факто стандарт для абстрактного API логирования. Библиотеки, использующие логирование, добавляют в зависимости крошечную библиотеку `slf4j-api`, содержащую только интерфейсы самого API логирования. Приложения же при развёртывании поставляют две библиотеки: само API и одну из его реализаций.

Реализаций SLF4J API существует несколько. Для учебных примеров нам вполне подойдёт простая реализация `slf4j-simple`, которая просто записывает все сообщения логирования в один файл или в консоль (по умолчанию — `System.err`). Для сеьёзных проектов имеет смысл посмотреть на одну из промышленных библиотек логирования. Например, `Logback` — эталонная реализация SLF4J API от тех же разработчиков — это мощная библиотека с гибкими средствами конфигурации, форматирования и направления сообщений логирования в консоль, файлы, по почте или в определённые пользователем приёмники сообщений.

На первых порах нам достаточно будет скачать дистрибутив SLF4J с сайта slf4j.org. В архиве содержится множество JAR-файлов, но нам из них понадобятся только два: `slf4j-api-1.x.x.jar` (собственно API) и `slf4j-simple-1.x.x.jar` (простая реализация). Чтобы подключить их к проекту Eclipse, скопируйте из в папку проекта, выделите оба файла и в их контекстном меню выберите *Build Path → Add to Build Path*.

Важно! Если подключить к проекту только библиотеку `slf4j-api` и не подключить никакую реализацию, то при создании первого логгера SLF4J выдаст не очень вразумительное предупреждение.

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further
```

```
details.
```

Имеется в виду, что при отсутствии в classpath какой-либо другой реализации выбирается реализация, не выполняющая никаких операций (no-operation) и просто игнорирующая все передаваемые логгеру сообщения.

Пять уровней логирования

В SLF4J API каждое сообщение логирования принадлежит к одному из пяти уровней, перечисленных в перечислимом типе Level. Их интерпретация отдаётся на откуп разработчику, но есть сложившиеся соглашения о том, какой уровень для каких случаев использовать.

Эти пять уровней соответствуют пяти из семи уровней (Level) стандартного пакета `java.util.logging`. Реализация `slf4j-jdk14`, оборачивающая SLF4J API поверх `java.util.logging`, отображает их стандартным образом. Два оставшихся уровня — `CONFIG` и `FINER` — недоступны через SLF4J API.

`ERROR` (ошибка)
: Обычно используется для сообщений о фатальных ошибках, при которых выполнение текущей операции аварийно обрывается. Также на этом уровне обычно логируются исключения. В реализации `slf4j-jdk14` отображается в уровень `SEVERE`.

`WARN` (предупреждение)
: Обычно используется для предупреждений, после которых выполнение текущей операции продолжается, но может привести к неожиданным для пользователя результатам. В реализации `slf4j-jdk14` отображается в уровень `WARNING`.

`INFO` (информация)
: Обычно используется для сообщений, представляющих интерфейс для конечного пользователя. В реализации `slf4j-jdk14` отображается в уровень `INFO`.

`DEBUG` (отладка)
: Обычно используется для отладочных сообщений, представляющих интерфейс только для разработчика. В реализации `slf4j-jdk14` отображается в уровень `FINE`.

`TRACE` (трассировка)
: Обычно используется для массового потока отладочных сообщений, позволяющих анализировать выполнение программы по шагам — например, сообщений о входе в метод и выходе из него. В реализации `slf4j-jdk14` отображается в уровень `FINEST`.

Получение объекта-логгера

Для получения объекта типа `Logger` с определённым именем используются семейство фабричных методов `LoggerFactory.getLogger`:

```
static Logger getLogger(String name)
```

```
static Logger getLogger(Class<?> clazz)
```

В случае вызова метода `getLogger` с параметром `Class` возвращается логгер с именем, равным полному имени переданного класса. Например, в следующем коде

```
package com.mycompany;

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

public class MyClass {
    private static final Logger log = LoggerFactory.getLogger(MyClass.class);
```

```
}
```

имя логгера, присвоенного полю `log`, будет равно `com.mycompany.MyClass`.

Показанный код является стандартной идиомой для создания логгеров: каждый класс содержит в себе собственную ссылку на используемый им логгер в `private static final`-поле.

> **Важно!** Будьте внимательны при импорте типа `Logger`. Убедитесь, что вы импортируете тип `Logger` из библиотеки SLF4J (`org.slf4j.Logger`), а не из стандартной библиотеки (`java.util.logging.Logger`).

Логирование сообщений

Для каждого уровня логирования в интерфейсе `Logger` есть одноимённый метод для логирования сообщения на этом уровне:

`void error(String format, Object... arguments)`

`void warn(String format, Object... arguments)`

`void info(String format, Object... arguments)`

`void debug(String format, Object... arguments)`

`void trace(String format, Object... arguments)`

Каждый метод принимает сообщение и необязательный список аргументов форматирования — идиома, похожая на `printf`, но использующая для форматирования другой синтаксис. Строковое представление каждого аргумента форматирования подставляется в строку вместо соответствующего по счёту вхождения пары фигурных скобок `{}`:

```
log.error("Index out of bounds (index: {}, size: {})", -1, 2);
// Index out of bounds (index: -1, size: 2)
```

Если пар фигурных скобок больше, чем аргументов форматирования, лишние пары не заменяются:

```
log.info("Я люблю {} и {}", "яблоки");
// Я люблю яблоки и {}
```

Наконец, каждый аргумент форматирования (как и в `printf` при использовании `%s`) преобразуется в строку с помощью `String.valueOf`, поэтому значения `null` подставляются в виде строки `"null"`, а для остальных объектов вызывается `toString`:

```
log.info("Сегодня {}", LocalDate.now());
// Сегодня 2016-05-19

log.info("Нулевая ссылка отображается как {}", null);
// Нулевая ссылка отображается как null
```

Зачем в SLF4J используется свой собственный формат сообщений, когда есть `String.format`? Дело в том, что SLF4J API был спроектирован до выхода Java 5, то есть до появления `String.format`. Кроме того, логирование критично к производительности, и такой простой формат сообщений значительно быстрее, чем полные возможности `String.format`. Но, конечно, ничто не запрещает использовать свои собственные средства форматирования вместо встроенных:

```
log.info(String.format("%04d", 42));
// 0042
```

Другая причина состоит в том, что форматирование сообщений в SLF4J осуществляется лениво, только в том случае, если соответствующий уровень логирования включён в конфигурации логирования. Это позволяет избежать лишнего форматирования строк, которые на самом деле никуда не будут выведены.

Оптимизация логирования с дорогими операциями

Представим себе, что сообщение логирования выводит результат какой-нибудь дорогой операции, которая может замедлить работу приложения. Конечно, использование стандартных средств форматирования SLF4J позволит избежать лишнего слияния строк и строковых представлений объектов, но сами объекты при этом будут созданы:

```
// Неоптимальное логирование с дорогими операциями - не делайте так!
List<String> words = getWords();
log.debug("Words in sorted order: {}",
        words.stream().sorted().collect(Collectors.joining(", ")));
```

В нашем случае, даже если уровень `DEBUG` отключён в настройках логирования, операции сортировки и слияния списка слов всё равно будут выполнены — вхолостую. Имеет смысл выполнять их только в случае, если соответствующий уровень логирования включён. Для такой проверки в классе `Logger` имеется пять методов, по одному на каждый уровень:

`boolean isErrorEnabled()`

`boolean isWarnEnabled()`

`boolean isInfoEnabled()`

`boolean isDebugEnabled()`

`boolean isTraceEnabled()`

Пользуясь этим средством, мы могли бы оптимизировать свой код:

```
List<String> words = getWords();

if (log.isDebugEnabled()) {
    log.debug("Words in sorted order: {}",
            words.stream().sorted().collect(Collectors.joining(", ")));
}
```

Логирование исключений

Для каждого из пяти уровней есть ещё одна форма методов логирования, принимающая вместо списка аргументов форматирования один параметр типа `Throwable` (хотя на практике, как правило, исключения логируются только на уровне `ERROR`):

`void error(String msg, Throwable t)`

`void warn(String msg, Throwable t)`

`void info(String msg, Throwable t)`

`void debug(String msg, Throwable t)`

`void trace(String msg, Throwable t)`

Эти методы не поддерживают форматирование сообщений (хотя для него всегда можно использовать сторонние средства форматирования, например, `String.format`). Эта форма методов логирования записывает в лог сначала сообщение, переданное первым параметром, а затем — сообщение и трассировку стека для переданного исключения.

Например, следующий код

```
public class ExceptionLogging {
    private static final Logger log =
            LoggerFactory.getLogger(ExceptionLogging.class);

    public static void main(String[] args) {
        try {
            Files.size(Paths.get("/invalid/invalid"));
        } catch (final IOException e) {
            log.error("Не удалось получить размер файла", e);
        }
    }

}
```

запишет в лог сообщение наподобие этого:

```
[main] ERROR ExceptionLogging - Не удалось получить размер файла
java.nio.file.NoSuchFileException: /invalid/invalid
    at sun.nio.fs.UnixException.translateToIOException(UnixException.java:86)
    at sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:102)
    at sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:107)
    at sun.nio.fs.UnixFileAttributeViews$Basic.readAttributes(UnixFileAttributeViews.j
ava:55)
    at sun.nio.fs.UnixFileSystemProvider.readAttributes(UnixFileSystemProvider.java:14
4)
    at sun.nio.fs.LinuxFileSystemProvider.readAttributes(LinuxFileSystemProvider.java:
99)
    at java.nio.file.Files.readAttributes(Files.java:1737)
    at java.nio.file.Files.size(Files.java:2332)
    at ExceptionLogging.main(ExceptionLogging.java:15)
```