



Урок 6

Обзор средств разработки

Логирование. Тестирование с использованием JUnit. Класс Assert. Аннотации.

Логирование

[Logger](#)

[Appender](#)

[Layout](#)

[Преимущество Log4j2 перед System.out.println\(\)](#)

[Пример использования Log4j2](#)

Тестирование

[Дополнительные настройки аннотации @Test](#)

[Особенности аннотации @Test](#)

[Параметризованный запуск](#)

Практическое задание

Дополнительные материалы

Логирование

Вывод сообщений (в консоль, и не только) является неотъемлемой частью тестирования и отладки разрабатываемых приложений. Использование `System.out.println()` для логирования работы приложения имеет несколько недостатков: вывод сообщений производится только в консоль, и по завершению работы эта информация теряется; нет возможности сохранить выведенные сообщения. Если же разработчику приходится работать с серверной частью приложения, то он не может видеть что происходит в консоли сервера, и для отладки можно использовать только лог-файл.

Для вывода информации и сохранении журнала работы приложения используются фреймворки для логирования. Давайте посмотрим на один из таких фреймворков Apache log4j2 - быстрый, гибкий, потокобезопасный, с поддержкой интернационализации.

Log4j2 имеет включает в себя три основных компонента: Logger, Appender, Layout.

Logger

Logger предназначен для логирования сообщений. Для работы с ним необходимо в каждом классе создавать по одному отдельному объекту. В примере ниже `CurrentClass` это класс, в котором создается логгер.

```
private static final Logger logger = LogManager.getLogger(CurrentClass.class.getName());
```

Logger содержит в себе методы, которые позволяют выводит сообщения о работе нашего приложения. Перечислим эти методы: `debug()`, `error()`, `info()`, `fatal()`, `warn()`, `trace()`. Каждый из них имеет свой приоритет:

Trace < Debug < Info < Warn < Error < Fatal

Fatal имеет наивысший приоритет, а Trace - наименьший. Если указать логеру уровень Info, то он будет выводить сообщения только с уровнем info и выше (info, warn, error, fatal).

Appender

Appender осуществляет запись сообщения в "пункт назначения": консоль, файл, базу данных, и т.д. Appender это интерфейс Log4j2, а его реализациями являются: `ConsoleAppender`, `FileAppender`, `JDBCAppender`, `SocketAppender`, `SyslogAppender`, и еще очень много других классов.

Layout

Layout отвечает за форматирование выводимых сообщений, и в Log4j представлены: `PatternLayout`, `HtmlLayout`, `YAMLLayout`, `XmlLayout`, `JSONLayout`.

Преимущество Log4j2 перед System.out.println()

Любой фреймворк для логирования позволяет указывать уровень сообщения, который может быть использован для фильтрации сообщений. Фреймворк для логирования предоставляет улучшенные средства вывода сообщений и метаданных, что позволяет ускорить и упростить процесс отладки.

Пример использования Log4j2

Первым делом необходимо создать проект и подключить к нему библиотеку Log4J2.

```
package com.geekbrains;

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

public class MainApp {
    private static final Logger LOGGER = LogManager.getLogger(MainApp.class);

    public static void main(String[] args) {
        LOGGER.debug("Debug");
        LOGGER.info("Info");
        LOGGER.warn("Warn");
        LOGGER.error("Error");
        LOGGER.fatal("Fatal");
        LOGGER.info("String: {}. ", "Hello, World");
    }
}
```

Log4j2 настраивается через properties или xml файл. Таким образом, разработчик может менять настройки логирования без необходимости корректировать исходный код приложения. По-умолчанию Log4j2 ищет файл с названием log4j2.xml в CLASSPATH проекта. Пример такого файла:

```
<?xml version="1.0" encoding="UTF-8"?>

<Configuration xmlns="http://logging.apache.org/log4j/2.0/config">
    <Properties>
        <Property name="basePath">logs</Property>
    </Properties>

    <Appenders>
        <File name="FILE" fileName="${basePath}/logfile.log" append="true">
            <PatternLayout pattern="%-5p %d{yyyy-MM-dd HH:mm:ss} [%t] %C (%F:%L)
- %m%n" />
        </File>

        <Console name="STDOUT" target="SYSTEM_OUT">
            <PatternLayout pattern="%-5p %d{yyyy-MM-dd HH:mm:ss} [%t] %C (%F:%L)
- %m%n" />
        </Console>
    </Appenders>

    <Loggers>
        <Logger name="com.geekbrains" level="debug" />
        <Root level="info">
            <AppenderRef ref="STDOUT" />
            <AppenderRef ref="FILE" />
        </Root>
    </Loggers>
</Configuration>
```

```
</Configuration>
```

Давайте посмотрим что означают эти настройки:

- 1) Properties - мы задаем собственные свойства для данного файла настроек. В данном случае нам удобно иметь свойство basePath - путь к каталогу, в котором будут храниться лог-файлы;
- 2) Appenders - настройка одно или более Appender'ов. Первый (FILE) будет выводить сообщения в файл, второй (STDOUT) - в консоль. PatternLayout задает форматирование сообщений.
- 3) Loggers - содержит объявление и настройку одного, или нескольких логеров. При использовании Root мы указываем настройки безымянного стандартного логера, который получает все сообщения от нашего приложения. Для каждого логера указывает минимальный уровень логирования.
- 4) AppendersRef - показывает какие Appender'ы подключены к логерам. ref указывает на поле name Appender'a.

Правила оформления шаблона для вывода PatternLayout:

- 1) %-5p указывает на то что будет отпечатан уровень сообщения, и он должен занимать 5 символов;
- 2) %d{yyyy-MM-dd HH:mm:ss} задает форматирование даты;
- 3) %t отображение имени метода, из которого была выполнена запись;
- 4) %C указание полного имени класса;
- 5) %F:%L имя файла:номер строки, где был выполнен вызов метода для логирования;
- 6) %m само сообщение;
- 7) %n перенос каретки.

При использовании таких настроек и запуске примера, приведенного выше, получим вывод в консоль и файл logs/logfile.log.

```
DEBUG 2019-02-11 22:29:34 [main] com.geekbrains.MainApp (MainApp.java:10) -
Debug
INFO 2019-02-11 22:29:34 [main] com.geekbrains.MainApp (MainApp.java:11) - Info
WARN 2019-02-11 22:29:34 [main] com.geekbrains.MainApp (MainApp.java:12) - Warn
ERROR 2019-02-11 22:29:34 [main] com.geekbrains.MainApp (MainApp.java:13) -
Error
FATAL 2019-02-11 22:29:34 [main] com.geekbrains.MainApp (MainApp.java:14) -
Fatal
INFO 2019-02-11 22:29:34 [main] com.geekbrains.MainApp (MainApp.java:15) -
String: Hello, World.
```

Тестирование

Рассмотрим тестирование на примере JUnit – библиотеки для написания тестов на Java. Чтобы начать тестировать, надо написать класс, который мы собираемся проверять. Пусть в этой роли выступит калькулятор (Calculator), обладающий функциями сложения, вычитания, деления и умножения пар чисел.

Код калькулятора:

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

```

    public int sub(int a, int b) {
        return a - b;
    }

    public int mul(int a, int b) {
        return a * b;
    }

    public int div(int a, int b) {
        return a / b;
    }
}

```

Каждый тест – это отдельный *public void* метод внутри класса. Он помечен аннотацией **@Test**, чтобы модуль тестирования смог найти его в коде. В качестве модуля, запускающего тесты, выступает класс **BlockJUnit4ClassRunner**.

Напишем тест, который будет складывать два числа и проверять, верный ли результат:

```

public class CalcTest {
    private Calculator calculator;

    @Test
    public void testAdd() {
        calculator = new Calculator();
        Assert.assertEquals(4, calculator.add(2, 2));
    }
}

```

Сначала мы создали объект класса **Calculator**, чтобы можно было пользоваться его методами. Потом обратились к статическому методу **assertEquals()** класса **org.junit.Assert**. Этот метод принимает на вход два аргумента (ожидаемый и полученный результат). Если в ходе выполнения теста эти результаты совпали – тест считается пройденным, если нет – проваленным (failed).

Класс **org.junit.Assert** содержит в себе много статических методов-проверок, чтобы гибко составлять тесты. Как правило, каждый из них имеет множество перегрузок для работы с разными типами данных:

- **assertTrue()** – проверяет, что переданное условие истинно;
- **assertFalse()** – проверяет, что переданное условие ложно;
- **fail()** – метод для принудительного «провала» теста;
- **assertEquals()** - проверяет, что для объектов, переданных в качестве параметров, метод **equals** возвращает **true**;
- **assertNotEquals()** – проверяет, что **equals** для объектов возвращает **false**;
- **assertNull()** – проверяет, что переданный в качестве параметра объект равен **null**;
- **assertNotNull()** – проверяет, что переданный в качестве параметра объект не равен **null**.

Практически во все IDE встроены инструменты для работы с тестами. Для запуска тестов достаточно нажать правой кнопкой мыши на тестовый метод или тестовый класс и выбрать пункт Run.

Код для тестирования всех методов класса **Calculator**:

```
public class CalcTest {
    private Calculator calculator;

    @Test
    public void testAdd() {
        calculator = new Calculator();
        Assert.assertEquals(4, calculator.add(2, 2));
    }

    @Test
    public void testSub() {
        calculator = new Calculator();
        Assert.assertEquals(3, calculator.sub(5, 2));
    }

    @Test
    public void testMul() {
        calculator = new Calculator();
        Assert.assertEquals(9, calculator.mul(3, 3));
    }

    @Test
    public void testDiv() {
        calculator = new Calculator();
        Assert.assertEquals(1, calculator.div(2, 2));
    }
}
```

При вызове каждого теста мы создаем объект класса **Calculator**, то есть проводим начальную настройку теста. Удобно написать метод, который автоматически выполнялся перед каждым тестом. Для этого можно использовать аннотацию **@Before**.

```
public class CalcTest {
    private Calculator calculator;

    @Before
    public void startTest() {
        calculator = new Calculator();
    }

    @Test
    public void testAdd() {
        Assert.assertEquals(4, calculator.add(2, 2));
    }

    @Test
    public void testSub() {
```

```

        Assert.assertEquals(3, calculator.sub(5, 2));
    }

    @Test
    public void testMul() {
        Assert.assertEquals(9, calculator.mul(3, 3));
    }

    @Test
    public void testDiv() {
        Assert.assertEquals(1, calculator.div(2, 2));
    }
}

```

Теперь перед запуском каждого теста будет запускаться метод **startTest()** и создавать экземпляр класса **Calculator**.

Методы, помеченные аннотацией **@After**, будут выполняться после каждого теста. Используя эти аннотации, мы подготавливаем и завершаем тесты, а также делаем код более читаемым.

Аннотации **@BeforeClass** и **@AfterClass** применяются к методам, которые будут вызваны один раз до и после выполнения всех тестов. Их используют для однократной подготовки и очистки «глобальных» тестовых данных. В этих методах можно инициализировать и прерывать соединение с базой данных. Методы с этими аннотациями обязательно должны быть объявлены как **public static void**.

Дополнительные настройки аннотации @Test

Ранее написанный тест можно пропустить, используя аннотацию **@Ignore**. При запуске тест будет помечен, как пропущенный. В аннотации можно указать причину пропуска. В итоге тестовый метод будет выглядеть примерно так:

```

@Test
@Ignore("Деление пока тестировать не нужно")
public void testDiv() {
    Assert.assertEquals(1, calculator.div(2, 2));
}

```

Когда не нужно запускать все тесты из класса, можно добавить аннотацию перед именем класса:

```

@Ignore
public class CalcTest {
    // ...
}

```

Особенности аннотации @Test

Параметр **timeout** – задает количество миллисекунд, по истечении которых тест будет помечен как **failed**. Пример:

```

@Test(timeout = 10000)
public void doSomething() {

```

```
// выполняем тяжелую задачу
}
```

Параметр `expected` – можно указать класс `Exception`, и если тест не выдаст необходимую ошибку, то будет помечен как `failed`. Пример:

```
@Test(expected = ArithmeticException.class)
public void checkSomething() {
    // здесь должно появиться исключение
}
```

В этом случае, если в тесте не будет брошен `ArithmeticException`, тест «провалится».

Параметризованный запуск

Как запустить один и тот же тест, но с разными входными данными, при этом не создавая при этом практически одинаковых методов? Нужно добавить к тестовому классу аннотацию **@RunWith** и указать в качестве параметра значение **Parametrized.class**. Затем добавить метод с генерацией параметров с аннотацией **@Parametrized.Parameters**, сами параметры – в класс, создать конструктор.

Разберем эту последовательность шагов подробнее.

Аннотация **@RunWith(Parametrized.class)** указывает на то, что мы собираемся запустить параметризованный тест. Аннотация **@Parametrized.Parameters** указывает на метод, который занимается генерацией входных данных для тестов. Он должен возвращать коллекцию данных в виде массивов объектов, которые будут использованы в качестве параметров.

У нас есть метод, возвращающий коллекцию массивов объектов – данных для тестов. В коде методов тестов нужно использовать сгенерированные данные. Для этого применяем переменные класса, заполняемые через конструктор.

Вернемся к тестированию калькулятора. Теперь у нас несколько тестов, проверяющих только сложение:

```
public class CalcAddTest {
    private Calculator calculator;

    @Before
    public void init() {
        calculator = new Calculator();
    }

    @Test
    public void testAdd1() {
        Assert.assertEquals(0, calculator.add(0, 0));
    }

    public void testAdd2() {
        Assert.assertEquals(2, calculator.add(1, 1));
    }
}
```



```

public void testAdd3() {
    Assert.assertEquals(4, calculator.add(2, 2));
}

public void testAdd4() {
    Assert.assertEquals(10, calculator.add(5, 5));
}

public void testAdd5() {
    Assert.assertEquals(6, calculator.add(4, 2));
}

public void testAdd6() {
    Assert.assertEquals(5, calculator.add(1, 3));
}

public void testAdd7() {
    Assert.assertEquals(4, calculator.add(6, -2));
}

public void testAdd8() {
    Assert.assertEquals(4, calculator.add(-1, 5));
}
}

```

Здесь кода немного: 8 небольших методов. Но представим, что вариантов входных данных станет не 8, а несколько сотен. А если проверка будет осуществляться не одной строкой, а двумя-тремя? Или потребуется скорректировать код метода тестирования? В любом из этих случаев придется писать и исправлять обширный код.

Рассмотрим тот же пример, написанный с использованием параметризации:

```

@RunWith(Parameterized.class)
public class CalcMassAddTest {
    @Parameterized.Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {
            {0, 0, 0},
            {1, 1, 2},
            {2, 2, 4},
            {5, 5, 10},
            {4, 2, 6},
            {1, 3, 4},
            {6, -2, 4},
            {-1, 5, 4}
        });
    }

    private int a, b, c;

    public CalcMassAddTest(int a, int b, int c) {

```

```

        this.a = a;
        this.b = b;
        this.c = c;
    }

    private Calculator calculator;

    @Before
    public void init() {
        calculator = new Calculator();
    }

    @Test
    public void massTestAdd() {
        Assert.assertEquals(c, calculator.add(a, b));
    }
}

```

Код стал намного компактнее. Все входные данные располагаются в методе `data()`. Кроме того, эти данные можно заменить на чтение из файла или базы данных.

Практическое задание

1. Написать метод, которому в качестве аргумента передается не пустой одномерный целочисленный массив. Метод должен вернуть новый массив, который получен путем вытаскивания из исходного массива элементов, идущих после последней четверки. Входной массив должен содержать хотя бы одну четверку, иначе в методе необходимо выбросить `RuntimeException`.
Написать набор тестов для этого метода (по 3-4 варианта входных данных).
Вх: [1 2 4 4 2 3 4 1 7] -> вых: [1 7].
2. Написать метод, который проверяет состав массива из чисел 1 и 4. Если в нем нет хоть одной четверки или единицы, то метод вернет `false`; Написать набор тестов для этого метода (по 3-4 варианта входных данных).
[1 1 1 4 4 1 4 4] -> true
[1 1 1 1 1 1] -> false
[4 4 4 4] -> false
[1 4 4 1 1 4 3] -> false
3. * Добавить на серверную сторону сетевого чата логирование событий (сервер запущен, произошла ошибка, клиент подключился, клиент прислал сообщение/команду).

Дополнительные материалы

1. Кей С. Хорстманн, Гари Корнелл. Java. Библиотека профессионала. Том 1. Основы;
2. Стив Макконнелл. Совершенный код;
3. Брюс Эккель. Философия Java;
4. Герберт Шилдт. Java 8: Полное руководство.