Статья опубликована в журнале *RSDN Magazine*, 2005, № 1, с. 14-19

Фильтрация строк с использованием автоматов

Александр Бабаев

Необходимость фильтрации строк

Строки используются очень часто. А применимо к Интернет-программированию можно сказать, что строки используются постоянно. Любой ответ сервера – это строка, запрос клиента – тоже строка. Работа с *XML*-файлами – это опять работа со строками, пускай и очень формализованная. Поэтому необходимо уметь быстро и эффективно обрабатывать строковые данные. Основная операция, которая используется – это конкатенация (слияние). Она реализована для всего, чего угодно и обычно очень прозрачна. Вторая же операция – это изменение строк. И тут мнения относительно того, что использовать, расходятся.

В статье предлагается *Pure Java API* для произвольной обработки строк. При этом показывается, как пользоваться такого рода библиотекой на конкретном примере разработанной автором библиотеки. Также сравнивается подход автора с классическим.

Стандартные методы фильтрации строк

Для начала вспомним, как происходит работа со строками в обычной программе. Используется несколько методов. Первый можно назвать классическим. В этом случае используются стандартные операции поиска, замены, конкатенации и удаления частей строки для получения результата. Такой метод оправдан для быстрого решения самых простых задач, но как только требуется реализовать что-нибудь более-менее сложное, мгновенно начинаются проблемы. Также этот способ совершенно не масштабируется и очень сложно изменяется.

Второй метод – использование регулярных выражений (регэкспов). Подробно рассматривать их не имеет смысла, есть отличная книга Дж. Фридла [1], в которой все подробно описано, в том числе и применимо к *Java*. Достоинства подхода заключаются в том, что у регулярных выражений очень компактная запись, оргомнейшие возможности и наличие «стандарта». То есть если вы научились использовать регулярные выражения в *Perl* или *PHP*, то ничего не стоит использовать их в *Java* (хотя все-равно приходися каждый раз выяснять нюансы реализации). Самый главный недостаток – сложность, которая произрастает из огромной мощности регулярных выражений. Простые регэкспы может понять даже начинающий программист, но более-менее сложные – начинающему уже не по зубам. Регэкспы же следующего вида (Листинг 1):

```
^[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-
\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-
\xff\n\015()]*)*\))[^\\\x80-
\xff\n\015()]*)*\)[\040\t]*)*(?:(?:[^(\040)<>@,;:".\\\[\]\000-
\037\x80-\xff]+(?![^(\040)<>@,;:".\\\[\]\000-\037\x80-
\xff])|"[^\\\x80-\xff\n\015"]*(?:\\[^\x80-\xff][^\\\x80-
\xff\n\015"]*)*")[\040\t]*(?:\([^\\\x80-
\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-
\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-
… … … … …
\xff])|\[(?:[^\\\x80-\xff\n\015\[\]]|\\[^\x80-
\xff])*\])[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-
\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-
\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*)*>)$
```

Листинг 1.Часть регулярного выражения, предназначенного для проверки корректности e-mail адреса, соответствия его RFC

не поймет никто даже при очень большом желании (на листинге представлена примерно восьмая часть регулярного выражения, предназначенного для проверки корректности e-mail адреса, соответствия его RFC). Хотя есть люди, которые «читают» регулярные выражения «с листа». Данный пример не совсем показателен в том смысле, что и программа, выполняющая аналогичную функцию, будет очень и очень сложна. Но есть и гораздо более простые задачи, которые, например, будут рассмотрены позже и в которых регулярные выражения дают аналогичный или худший результат при гораздо меньшем удобстве.

Другой немаловажный недостаток регулярных выражений в том, что мало кто понимает, как они работают. «Я пишу это, он делает то…» А как – это проблема тех, кто библиотеку разрабатывает. «Чукча не читатель, чукча писатель». В результате – ляпы, непонятные «глюки», и неправильно, некорректно работающий программный код. Также регэкспы ненастраиваемы. Для того чтобы поменять регэксп, нужно изменить его код и перекомпилировать регулярное выражение. Нельзя просто поменять значение одной переменной для того, чтобы немного изменить логику работы.

Фильтрация строк

После довольно длительного использования различного рода методов обработки строк постепенно появился вывод, что хочется совместить настраиваемость обычного класса и мощность регулярных выражений. А как базу для этого – использовать автоматы [2-4]. Рассмотрим такой подход на конкретном примере. Пускай необходимо обрабатывать строки записей в интернет-форуме. При этом требуется реализовать обработку следующих правил:

1. Все слова длинее некоторого количества символов *N* разбивать пробелами на отрезки, длинна которых меньше, либо равна *N*.
2. Если длина сообщения больше *M*, то оставлять только первые *M* символов.
3. Заменять три точки на символ многоточия.
4. Заменять два подряд идущих символа «минус», обрамленных пробелами, на «тире».
5. Заменять символы «"» на правильные кавычки в русском тексте – «елочки» и «лапки».
6. Заменять ссылки на интернет-ресурсы (*http://...*, *ftp://...*) на *HTML*-ссылки
7. Заменять *e-mail* адреса на *HTML*-ссылки. При этом адресом для упрощения считаем последовательность непробельных символов, которая содержит «@». Это не самое лучшее определение, но работающее достаточно часто.
8. Заменять комбинации символов, которые обозначают стандартные эмотиконы (смайлы) на соответствующие картинки.
9. «Обезвреживать код». То есть делать так, чтобы пользователь не мог в тексте сообщения ввести вредоносный *HTML*-код. Таким кодом традиционно считается любой кроме некоторых очень простых тегов <*b*>, <*i*>, <*u*> и аналогичных.
10. При условии соблюдения правила номер девять, дать пользователю возможность форматирования текста. А именно, выделения текста полужирным, наклонным начертанием, перечеркивание или подчеркивание текста, выделение цитаты и форматированного текста (кода программы, например).

Эти правила являются достаточно стандартными для практически любой системы, где используется работа с текстом. Существует множество вариантов их реализации. Самый распространенный – при помощи уже упоминавшихся регулярных выражений [5]. В данном случае строится по одному или несколько выражений на каждое правило, после чего они в определенном порядке применяются к строке. Выполнение каждого регулярного выражения – это один проход по строке, следовательно, таких прогонов будет огромное количество. Правда, большая часть из них будет пустая, но даже они занимают какое-то время.

Безусловно, возможно написать такое регулярное выражение, которое будет одно исполнять все правила сразу, но, боюсь, что его написание займет не один день, а малейшее изменение потребует очень серьезных усилий.

Возможен другой вариант, который и подводит непосредственно к автоматному методу работы. Те, кто более глубоко интересовался регэкспами, скажут, что автоматы и регэкспы – это одно и то же. Да, любой регэксп – это всего лишь короткая строковая запись автомата. Но обсуждение такого рода различий выходит далеко за рамки статьи.

Код, который обрабатывает строку, называется фильтром. Фильтр посимвольно перебирает строку, и для каждого символа проверяет, есть ли обработчик этого символа. Если есть – то передает управление ему. Иначе просто добавляет символ в выходной поток и переходит к следующему.

На рис. 1 представлен граф состояний автомата, который управляет работой фильтра.

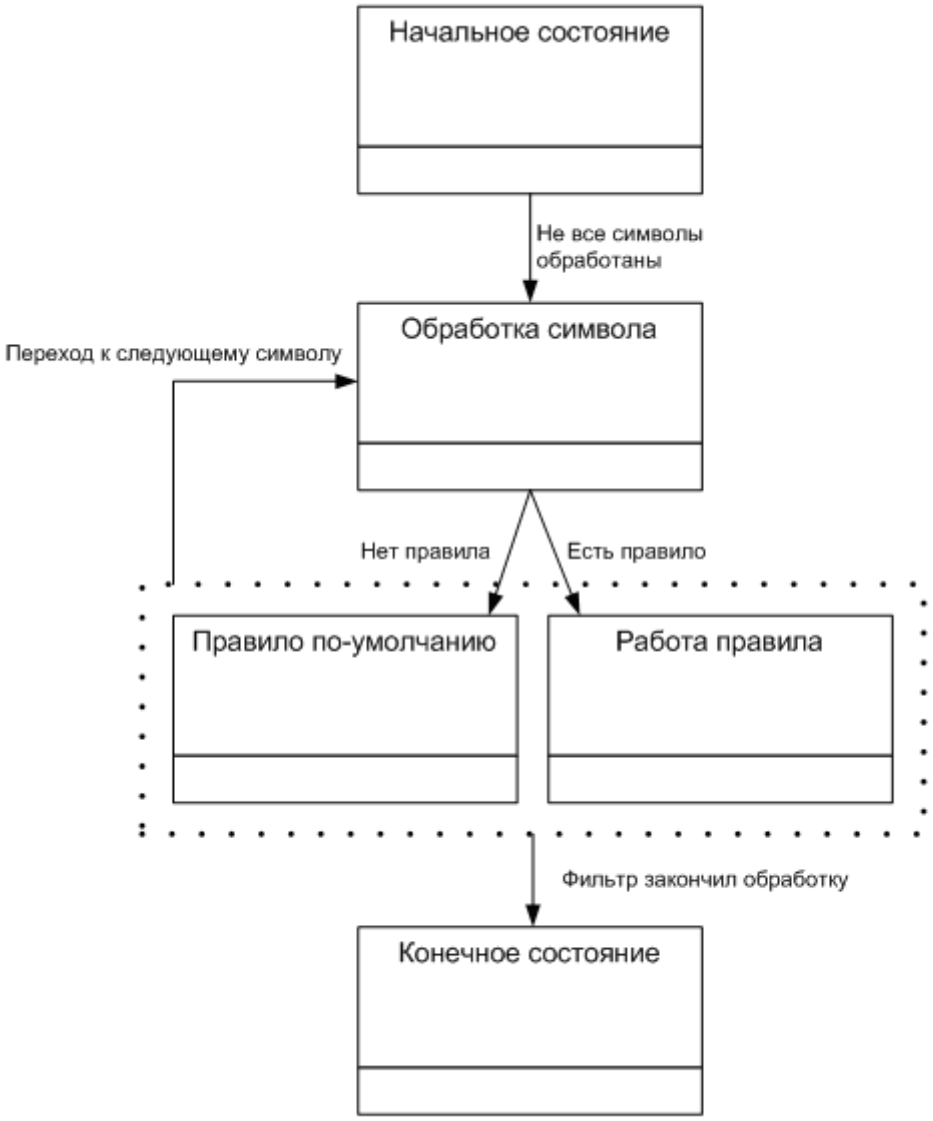

Рисунок 1. Граф состояний автомата, который управляет работой фильтра

Листинг 2 показывает, как этот автомат реализован в коде. Код не является чистым автоматом «по коду», но реализует именно автоматную логику.

```
public String process(String aString) throws FilterException {
    // что такое правила - будет объяснено чуть позже, тут они
    // инициализируются, потому что фильтр может быть использован
    // повторно
    initRules();

    // проверим, что на вход получена корректная строка
    if (aString == null || aString.length() == 0) {
        return "";
    }

    // Состояние "Начальное состояние"
    Source source = new Source(aString);
    Result result = new Result();

    // основной цикл длится, пока мы находимся «не в состоянии завершения»
    while (!result.getLastRuleResult().
                    equals(RuleResult.FILTER_FINISHED_PROCESSING)) {
        result.setLastRuleResult(RuleResult.CHAR_NOT_CHANGED);

        // Состояние "Обработка символа"

        // строка обработана полностью
        if (source.isStringFinished()) {
            break;
        }

        // перед каждой обработкой - происходит внутряняя инициализация
        // так же проверяется, что нет зацикливания
        try {
            source.prepare();
        } catch (FilterException e) {
            e.printStackTrace();
            if (e.getCanContinue().equals(FilterException.CONTINUABLE)) {
                source.addToPosition(1);
                continue;
            } else if (e.getCanContinue().equals(FilterException.FATAL)) {
                throw e;
            }
        }

        // прогоняем правила, соответствующие текущему символу

        if (rules.size() > 0) {
            // Состояние "Работа правила"
            Collection rules = getRulesCollection(
                    aSource.getCurrentCharacter());
            for (Iterator iterator = rules.iterator(); iterator.hasNext();) {
                IRule rule = (IRule) iterator.next();
                if (rule.isEnabled()) {
                    rule.process(aSource, aResult, this);
                }

                if (!aResult.getLastRuleResult().
                        equals(RuleResult.CHAR NOT CHANGED)) {
                    break;
                }
            }
        } else {
            // Состояние "Правило по-умолчанию"
            EMPTY_RULE.process(source, result, this);
        }

        // если ни одно правило не было применено, то
        // выполняем правило по умолчанию
        if (result.getLastRuleResult().
                        equals(RuleResult.CHAR_NOT_CHANGED)) {
            EMPTY_RULE.process(source, result, this);
        } else if (result.getLastRuleResult().
                        equals(RuleResult.FILTER FINISHED PROCESSING)) {
            // Переход в состояние "Конечное состояние"
            break;
        }
    }

    // Состояние "Конечное состояние"

    // В процессе работы фильтра в строку включаются тэги (основное
    // его предназначение - форматирование для вывода в HTML)
    // В результате ошибок и неаккуратностей некоторые теги могут быть
    // незакрыты. Следующий метод дополняет строку закрывающими
    // тегами в корректном порядке.
    result.appendEndAppendersInReverseOrder();

    return result.getResult();
}
```

Листинг 2. Реализация автомата

Теперь, когда понятно, как работает основной цикл программы, посмотрим на некоторые правила. Например, вот правило замены трех точек на специальный символ (на листинге 3 приведен только метод обработки символа, но не весь класс).

```
public class HellipRule extends AbstractRule {
    private static final char CHARACTER = '.';
    private static final Character INITIATOR = new Character(CHARACTER);

    public Character getInitiatorCharacter() {
        return INITIATOR;
    }

    public void process(Source aSource, Result aResult, IFilter aFilter) {
        // проверяем, что за текущей точкой будут еще две точки
        if (StringUtils.isSymbol(aSource.getSource(),
                                 aSource.getPosition() + 1, CHARACTER) &&
                StringUtils.isSymbol(aSource.getSource(),
                                 aSource.getPosition() + 2, CHARACTER)) {
            // в результат выводим нужную строку
            aResult.append("&hellip;");
            // выставляем состояние автомату, чтобы он
            // переходил к следующему символу
            aResult.setLastRuleResult(RuleResult.CHAR_FINISHED_PROCESSING);
            // перескакиваем обработку всех трех точек
            aSource.addToPosition(3);
        }
    }
}
```

Листинг 3. Реализация правила замены трех точек на «…»

На листинге 4 представлено правило общей обработки «двойных символов». Это правило, которое является базой для множества правил форматирования, при помощи которых можно выделять текст «жирным», «наклонным» и так далее, не прибегая к тегам *HTML*, но обрамляя нужные куски текста в «звездочки», «наклонные черты» и другие легко запоминающиеся символы.

```
    public void process(Source aSource, Result aResult, IFilter aFilter) {
        int nextPosition = aSource.getPosition() + 1;

        // проверяется, что следующий символ – такой же, как и предыдущий
        if (isSymbol(aSource.getSourceString(), nextPosition, getSymbol())) {
            // смотрим на текущее состояние правила, чтобы определить,
            // создавать открывающий или закрывающий тег
            if (getState().equals(DoubleCharacterState.STATE_OUT)) {
                // открывающий тег
                setState(DoubleCharacterState.STATE IN);
                aSource.addToPosition(2);
                aResult.append(getPrefix());

                // записываем в «строки окончаний» закрывающий тег, который
                // является парным к текущему – чтобы автоматически
                // закрыть тег в конце строки, если не окажется
                // парного к тому, который сейчас вставляется (1)
                aResult.addEndAppend(getPostfix());
                // устанавливаем состояние «окончания обработки символа»
                aResult.setLastRuleResult(RuleResult.CHAR_FINISHED_PROCESSING);
            } else if (getState().equals(DoubleCharacterState.STATE_IN)) {
                if (aResult.containsEndAppend(getPostfix())) {
                    setState(DoubleCharacterState.STATE OUT);
                    aSource.addToPosition(2);
                    aResult.append(getPostfix());
                    // удаляем закрывающий тег из «строк окончаний».
                    // Если бы мы его тут не удалили, после окончания
                    // обработки строки, он бы вставился автоматически.
                    aResult.removeEndAppend(getPostfix());
                    // устанавливаем состояние «окончания обработки символа»
                    aResult.
                       setLastRuleResult(RuleResult.CHAR_FINISHED_PROCESSING);
                }
            }
        }
    }
```

Листинг 4. Реализация правила обработки «двойных символов»

Структура библиотеки *JFilter*

Классы

На рисунке 2 представлена диаграмма классов для библиотеки, при этом большая часть классов правил убрана для того, чтобы повысить читабельность.

Рисунок 2. Диаграмма классов

Описание

В библиотеке JFilter есть несколько интерфейсов:

- *IFilter* (Листинг 5), который описывает сам фильтр.

```
public interface IFilter extends Serializable {
    /**
     * Обрабатывает строку, воззвращая в качестве результата строку
     * после фильтрации.
     * @param aSourceString - исходная строка
     * @return обработанная строка
     * @throws FilterException если произошла фатальная ошибка
     * (зацикливание)
     */
    public String process(String aSourceString) throws FilterException;

    /**
     * Выключает правило по его rule.getClass().getName()
     * @param aRuleClass - класс правила (что-то вроде IRule.class)
     */
    public void disableRuleByClassName(Class aRuleClass);

    /**
     * Включает правило по его rule.getClass().getName()
     * @param aRuleClass - класс правила (что-то вроде IRule.class)
     */
    public void enableRuleByClassName(Class aRuleClass);

    /**
     * Выключает все правила.
     */
    public void enableAllRules();

    /**
     * Включает все правила.
     */
    public void disableAllRules();

    /**
     * Добавляет правило в фильтр.
     * @param aRule правило, которое нужно добавить в фильтр.
     */
    void addRule(IRule aRule);
}
```

Листинг 5. Интерфейс фильтра

- *IRule* (Листинг 6), показывающий, какие методы должны быть у правила.

```
public interface IRule extends Serializable {
    /**
     * Метод, который вызывается перед обработкой произвольной строки.
     */
    public void initialize();

    /**
     * Инициализация параметров правила. Этот метод используется в
     * основном для установки параметров правила при загрузки
     * конфигурации фильтра из XML.
     * @param aParameters карта параметров.
     */
    void setParameters(Map aParameters) throws FilterException;

    /**
     * Включает или выключает правило.
     */
    public void setEnabled(boolean aEnabled);

    /**
     * @return true, если правило включено, false - иначе.
     */
    public boolean isEnabled();

    /**
     * @return символ, который является инициатором данного правила.
     */
    public Character getInitiatorCharacter();
    /**
     * Обрабатывает текущую строку при помощи правила.
     * @param aSource исходная строка, текущая позиция
     * @param aResult текущий результат обработки
     * @param aFilter текущий фильтр
     */
    public void process(Source aSource, Result aResult, IFilter aFilter);
}
```

Листинг 6. Интерфейс правила

- *IRuleGroup* (Листинг 7) – интерфейс работы с группой однотипных правил, как например, правила транслитерации.

```
public interface IRuleGroup {
    /**
     * Добавляет правила группы в указанный фильтр.
     * @param aFilter
     */
    public void addRules(IFilter aFilter);
    /**
     * Включает или выключает все правила, входящие в группу.
     */
    public void setEnabled(boolean aEnabled);
    /**
     * Возвращает true, если все правила группы включены в указанном фильтре
     * @param aFilter
     */
    public boolean isEnabled(IFilter aFilter);
    /**
     * Инициализация параметров группы. Этот метод используется в
     * основном для установки параметров правила при загрузки
     * конфигурации фильтра из XML.
     * @param aParameters карта параметров
     */
    public void setParameters(Map aParameters);
}
```

Листинг 7. Интерфейс группы правил

Для того, чтобы было проце работать с системой, написано несколько классов, помогающих при написании новых фильтров и правил. Такими классами являются *AbstractFilter*, *AbstractRule*. Первый – описывает все необходимые методы для работы стандартного фильтра. Поэтому для того, чтобы создать нужный фильтр – можно просто отнаследоваться от *AbstractFilter* и в конструкторе вызвать метод *addRule()*, добавив все необходимое в нужной последовательности (листинг 8).

```
public class WikiFilter extends AbstractFilter {
    public WikiFilter(int aMaxWordLength, int aMaxStringLength) {
        // замена < на &lt;
        addRule(new ReplaceLeftTagBracketRule());
        // замена & на &amp;
        addRule(new ReplaceAmpersandTagBracketRule());
        // правило экранирования – для возможности вывода спецсимволов
        addRule(new EkranRule());
        // правило замены http://... - ссылками
        addRule(new AnchoringRule(aMaxWordLength));
        … … …
        // правило разбивания длинных слов пробелами
        addRule(new BreakWordsRule(aMaxWordLength));
        // правило «обрезания» длинных строк
        addRule(new MaxLengthRule(aMaxStringLength));
    }
}
```

Листинг 8. Составление фильтра из отдельных правило

После этого обработка строки представляет данным фильтром собой тривиальную задачу:

```
String result =
     new WikiFilter(maxWordLength, maxStringLength).process(sourceString);
```

AbstractRule определяет методы *setEnabled()* и *isEnabled()*, поскольку они одинаковые для большинства фильтров.

```
public abstract class AbstractRule implements IRule {
    // это поле сделано ThreadLocal для того, чтобы можно было одним фильтром
    // обрабатывать несколько строк одновременно
    private ThreadLocal _enabledThreadLocal = new ThreadLocal() {
        protected Object initialValue() {
            // по-умолчанию правило включается
            return Boolean.TRUE;
        }
    };

    public void setParameters(Map aParameters) throws FilterException {
        // для многих правил этот метод не используется, поэтому делаем
        // его необязательным
    }

    public void initialize() {
        // также как и setParameters - делаем этот метод необязательным
    }

    public void setEnabled(boolean aEnabled) {
         enabledThreadLocal.set(Boolean.valueOf(aEnabled));
    }

    public boolean isEnabled() {
        return ((Boolean) _enabledThreadLocal.get()).equals(Boolean.TRUE);
    }
}
```

Естественно, есть возможность отключать некоторые правила непосредственно в процессе работы фильтра или между исполнениями метода *process()*. Можно также динамически изменять фильтр или настраивать его. При этом не требуется перекомпиляции кода или самого фильтра.

Также создан вспомогательный класс *CustomFilterFromXML*, который позволяет загрузить конфигурацию фильтра из *XML*-файла и автоматически ее обновляет при изменении *XML*-файла.

Применение

Метод фильтрации строк, представленный в данной статье применим не только в узкоспециализированной области работы с интернет-текстами. Создавая разные правила легко создавать строки, которые будут управлять, например поведением программы. Возможности неограниченны. Система, которая описана, называется *JFilter* и распространяется свободно, страничка в интернете: *http://blog.existence.ru/exception/.products.JFilter* Там можно найти как библиотеку в формате *jar*, так и исходные тексты с документацией. В приложении приведен код интерфейсов фильтра и правила, которые являются основными интерфейсами системы. *JFilter* используется в системе блогов *JDnevnik* [6] и еще в нескольких проектах.

Правила, входящие в поставку

Все стандартные правила написаны в стиле, допускающем одновременную обработку фильтром нескольких строк, благодаря чему их можно использовать в многопоточной среде.

Таблица 1. Стандартные правила

Стандартные правила (*jfilter.rules.general*)	*BreakWorksRule* – разбивает длинные слова пробелом.
	EkranRule – правило экранирования спецсимволов.
	MaxLengthRule – правило, которое ограничивает длину строк.
HTML-правила (*jfilter.rules.html*)	*AnchoringRule* – замена текста, начинающегося с *http://* (или аналогов) и заканчивающегося пробелом на *HTML*-ссылку
	MailRule – замена текста, заключенного в пробелы и содержащего символ «@» на *HTML*-ссылку.

Эмотиконы (*jfilter.rules.smiles*)	*PreSmiler* – замена стандартных обозначений эмотиконов («:-)», «;)» и так далее) на их стандартные обозначения для обработки SmilesRule'ом.
	SmilesRule – замена одиночных слов, заключенных в парные двоеточия «::» на *HTML*-картинки.
Транслитерация (*jfilter.rules.transliteration*)	*Lat2RusTransliterationRuleGroup* – транслитерация. Соответствует *ISO* и *ГОСТ* (то есть можно писать как по *ГОСТу*, так и обозначениями *ISO*. Пока конфликтов такого варианта я не обнаружил).
Типографика (*jfilter.rules.typografica*)	*AbbreviationsRule* – замена (c), (p), (r), (tm) на соответствующие *HTML*-символы.
	HellipRule – замена трех точек подряд на символ «многоточие».
	LongDashRule – Замена символа «минус», обрамленного пробелами на «тире».
	QuotesRule – замена «знака дюйма» на корректные кавычки для русского языка.
Вики-форматирование (*jfilter.rules.wacko*)	*AnchorRule* – создание ссылок в виде ((link)).
	BlockquoteRule – цитата (>>текст>>).
	BoldRule – выделение полужирным начертанием (**полужирный**).
	HeaderRule – заголовок (==заголовок==).
	ItalicRule – выделение курсивом (//курсив//).
	MonospaceRule – выделение моноширинным шрифтом (##моноширинный##).
	NoteRule – выделение замечания (span class="note").
	ParagraphRule – Работа с переводами строки (одиночный перевод строки - , два подряд - <p>).
	PreformattedRule – выделение предварительно отформатированного текста (%%уже форматрованный текст%%).
	ReplaceAmpersandTagBracketRule – замена символа "&" на соответствующий *HTML*-аналог (&).
	ReplaceLeftTagBracketRule – замена символа "<" на соответствующий *HTML*-аналог (<).
	SmallRule – выделение мелким шрифтом (++мелкий++).
	StrikeRule – выделение перечеркиванием (--перечеркнутый--).
	SubscriptRule – выделение нижним индексом (vvнижний индексvv).
	SuperscriptRule – выделение верхним индексом (^^верхний индекс^^).
	UnderlineRule – выделение подчеркиванием (__подчеркивание__).

Сравнение работы разных типов обработки строк

Таблица 2. Сравнение разных типов обработки строк

Свойство	Классический способ	Регулярные выражения	Фильтрация
Простота реализации	Просто	Очень сложно	Сложно
Возможность изменения	Для простых задач – очень большие, для сложных – очень сложные	Только вместе с перекомпиляцией выражения	Максимальные
Простота использования	Очень просто	Просто для тех, кто потратил много времени на обучение	Очень просто
Скорость работы	Быстро на простых вариантах, обычно медленно на сложных, сильно зависит от реализации	Быстро, если правильно использовать	Быстро

Заключение

Хочется отметить, что библиотека *JFilter* продолжает развиваться, на настоящий момент реализовано очень много, но далеко не все из того, что хотелось бы реализовать. В ближайшем будущем предполагается написание дополнительных правил для классической, «бумажной» типографики, которые помогали бы редакторам и верстальщикам самых обычных, бумажных книг/журналов/газет.

Литература

1. Фридл Дж. Mastering Regular Expressions. Питер, 2003 год
2. http://is.ifmo.ru Санкт-Петербургский государственный университет информационных технологий, механики и оптики, кафедра «Технологии программирования»
3. Бабаев А. Транслитерация и как правильно ее надо программировать // МИР ПК – ДИСК. 2003. №12. http://is.ifmo.ru/works/translit/
4. http://is.ifmo.ru/projects/bone/ Создание скелетной анимации на основе автоматного программирования
5. http://wackowiki.com/projects/WackoFormatter Библиотека для форматирования текста *WackoFormatter*
6. http://blog.existence.ru/exception/.products.JDnevnik Система блогов *JDnevnik*