正确、优美、迅速

——从设计XML验证器中学到的经验

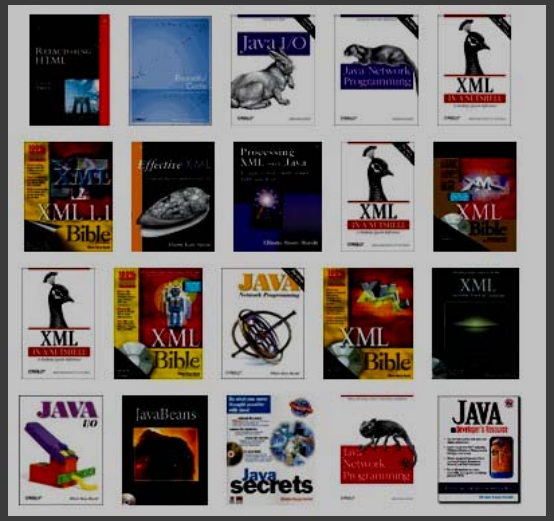
00801075　　李宣成

http://www.elharo.com/

Elliotte Rusty Harold

新奥尔良人，纽约科技大学副教授，主要讲授Java、XML以及面向对象编程，Internet方面国际知名的作家、程序员和教育家，Java创始人之一。

他的网站
Café au Lait： （http://www.cafeaulait.org）
是互联网上最流行的独立Java网站之一；



他的另一个网站
Café con Leche： (http://www.cafeconleche.org)
则是最流行的XML站点之一。

他编写了二十余本书籍，包括《Java I/O》，《Java Network Programming》和《XML in a Nutshell》等等

铺垫

XML：可扩展标记语言，为通用标记语言的一种，是一种简单易用的数据存储语言。与HTML相比，更为严谨简洁，且效能更佳

例：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
    <book catalog="Programming">
    <title>Beautiful Code</title>
    <author>Leading Programmers</author>
    <year>2010</year>
    <price>99.0</price>
    </book>
```

铺垫

JDOM、XOM：两者都是XML的API，使得程序员可以通过Java来访问、操作和输出XML数据

DTD：文档类型定义，一个XML文档可以引用外部的DTD，也可以自身在开头使用!DOCTYPE、 !ELEMENT、!ATTLIST等操作来定义DTD，它声明了XML文档的元素，并描述了文档的格式。一个XML文档可以携带一个DTD来测试它是否为有效的XML文档

例：

```
<?xml version="1.0"?>
    <!DOCTYPE note [
    <!ELEMENT note
(to,from,heading,body)>
    <!ELEMENT to (#PCDATA)>
    <!ELEMENT from (#PCDATA)>
    <!ELEMENT heading
(#PCDATA)>
    <!ELEMENT body (#PCDATA)>
    ]>
```

对应于这个DTD的定义，合法的XML文档有如下的格式：

```
 <note>
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget me this
weekend</body>
    </note>
```

铺垫

Namespace:即文中所说的命名空间，它的使用是为了区分不同XML应用中同名的元素和属性或者在应用中归类相关的元素和属性，让程序能容易地识别它们。

命名空间在操作上是将每个元素和属性加上一个前置字，每个前置字都对应到一个URI。附属于同一个URI的元素和属性，会属于同一个命名空间，标准URI可用来识别不同XML应用的元素。命名空间中的元素和属性之名称会包含一个冒号，在它之前的所有东西，称作前置字。而之后的所有东西，称作局部名字

```
<?xml version=“1.0” encoding=“ISO-8859-1”
standalone=“yes”?>
<catalog>
   <book>
      <title> Beautiful Code </title>
   </book>
   <paint>
      <pt:title> 111</pt:title>
   </paint>
</catalog>
```

确保XML文档正确性所需要的两个冗余的数据检查：

1、输入验证，验证XML文档的结构是否符合规范以及　与其所携带的DTD所匹配

2、输出验证，使用DOM、JDOM等工具构造XML文档时检查传递给API的字符串在XML中的合法性

XML名字的合法性约束：

- 文档中不允许出现交迭的元素（由于JDOM的机制不需考虑）
- 元素名、属性名或者处理指令的名字必须是合法的XML名字
- 局部名字不能包含冒号
- 属性的命名空间不能与其父元素或者兄弟属性的明明空间冲突
- 每一个Unicode的代理字符出现在一个代理对中
- 处理指令数据不包含双字节字符串?>
- ……

例5-1．分析XML名字的BNF语法

BaseChar ::= [#x0041-#x005A] | [#x0061-#x007A] | [#x00C0-#x00D6]
NameChar ::= Letter | Digit | '.' | '-' | '_' | ':' | CombiningChar | Extender
Name ::= (Letter | '_' | ':') (NameChar)*
Letter ::= BaseChar | Ideographic
Ideographic ::= [#x4E00-#x9FA5] | #x3007 | [#x3021-#x3029]
Digit ::= [#x0030-#x0039] | [#x0660-#x0669] | [#x06F0-#x06F9]
| [#x0966-#x096F] | [#x09E6-#x09EF] | [#x0A66-#x0A6F]
| [#x0AE6-#x0AEF] | [#x0B66-#x0B6F] | [#x0BE7-#x0BEF]
| [#x0C66-#x0C6F] | [#x0CE6-#x0CEF] | [#x0D66-#x0D6F]
| [#x0E50-#x0E59] | [#x0ED0-#x0ED9] | [#x0F20-#x0F29]
Extender ::= #x00B7 | #x02D0 | #x02D1 | #x0387 | #x0640 | #x0E46 | #x0EC6
| #x3005 | [#x3031-#x3035] | [#x309D-#x309E] | [#x30FC-#x30FE]
| [#x00D8-#x00F6] | [#x00F8-#x00FF] | [#x0100-#x0131]
| [#x0134-#x013E]...
CombiningChar ::= [#x0300-#x0345] | [#x0360-#x0361] | [#x0483-#x0486]
| [#x0591-#x05A1] | [#x05A3-#x05B9] | [#x05BB-#x05BD] | #x05BF
| [#x05C1-#x05C2] | #x05C4 | [#x064B-#x0652] | #x0670
| [#x06D6-#x06DC]...

BNF语法：

- “”中的字代表字符本身
- < >中的为必选项。
- [ ] 中的为可选项。
- { } 中的为可重复0至无数次的项。
- | 表示左右两边任选一项。
- ::= 是“被定义为”的意思。

版本一：简单的实现

第一个版本的名字字符验证

```
private static String checkXMLName(String name) {
// Cannot be empty or null
if ((name == null) || (name.length() == 0) || (name.trim( ).equals(""))) {
return "XML names cannot be null or empty";
}
// Cannot start with a number
char first = name.charAt(0);
if (Character.isDigit(first)) {
return "XML names cannot begin with a number.";
}
// Cannot start with a $
if (first == '$') {
return "XML names cannot begin with a dollar sign ($).";
}
// Cannot start with a _
if (first == '-') {
return "XML names cannot begin with a hyphen (-).";
}
// Ensure valid content
for (int i=0, len = name.length( ); i<len; i++) {
char c = name.charAt(i);
if ((!Character.isLetterOrDigit(c))
&& (c != '-')
&& (c != '$')
&& (c != '_')) {
return c + " is not allowed in XML names.";
}
}
// We got here, so everything is OK
return null;
}
```

版本一的缺陷：

它使用Java自带的判定函数来判断字符是否合法。由此导致两个问题：

- Java中这些函数的定义并没有完全遵照XML中关于字母和数字的定义。即Java把一些XML中非法的字母视为合法，或者正好相反。
- Java中的验证规则在版本更新时会有所变化，而XML的验证规则没有变化

另外，这个版本允许在名字中包含冒号，这不利于维持命名空间的良好形式

P.S. XML中不合法的字符可以将其转化为实体使用，如：

< → <
> → >
& → &
&pos; → ‘
" → “

* XML标准1.1中建议文档作者使用在自然语言中有意义的字词作为XML名称，并避免在名称中使用符号字符或空白字符。注意：冒号（：）、连字符（-）、句号（.）、下划线（_）和圆点（·）是明确允许的。详请参见：http://www.w3china.org/translation/xml1p1CR20021015_cn.htm

```
private static String checkXMLName(String name) {
// Cannot be empty or null
if ((name == null) || (name.length( ) == 0)
|| (name.trim( ).equals(""))) {
return "XML names cannot be null or empty";
}
// Cannot start with a number
char first = name.charAt(0);
if (!isXMLNameStartCharacter(first)) {
return "XML names cannot begin with the character \""
+
first + "\"";
}
// Ensure valid content
for (int i=0, len = name.length( ); i<len; i++) {
char c = name.charAt(i);
if (!isXMLNameCharacter(c)) {
return "XML names cannot contain the character \"" + c
+ "\"";
}
}
// We got here, so everything is OK
return null;
}
```

版本二：模拟BNF语法

以上程序中调用的几个函数：

```
public static boolean
isXMLNameCharacter(char c) {
return (isXMLLetter(c) || isXMLDigit(c) ||
c == '.' || c == '-'
|| c == '_' || c == ':' ||
isXMLCombiningChar(c)
|| isXMLExtender(c));
}
public static boolean
isXMLNameStartCharacter(char c) {
return (isXMLLetter(c) || c == '_' || c ==':');
}
```

```
public static boolean isXMLDigit(char c) {
if (c >= 0x0030 && c <= 0x0039) return true;
if (c >= 0x0660 && c <= 0x0669) return true;
if (c >= 0x06F0 && c <= 0x06F9) return true;
if (c >= 0x0966 && c <= 0x096F) return true;
if (c >= 0x09E6 && c <= 0x09EF) return true;
if (c >= 0x0A66 && c <= 0x0A6F) return true;
if (c >= 0x0AE6 && c <= 0x0AEF) return true;
if (c >= 0x0B66 && c <= 0x0B6F) return true;
if (c >= 0x0BE7 && c <= 0x0BEF) return true;
if (c >= 0x0C66 && c <= 0x0C6F) return true;
if (c >= 0x0CE6 && c <= 0x0CEF) return true;
if (c >= 0x0D66 && c <= 0x0D6F) return true;
if (c >= 0x0E50 && c <= 0x0E59) return true;
if (c >= 0x0ED0 && c <= 0x0ED9) return true;
if (c >= 0x0F20 && c <= 0x0F29) return true;
return false;
}
```

版本三

```
public static boolean isXMLDigit(char c) {
    if (c < 0x0030) return false; if (c <= 0x0039) return true;
    if (c < 0x0660) return false; if (c <= 0x0669) return true;
    if (c < 0x06F0) return false; if (c <= 0x06F9) return true;
    if (c < 0x0966) return false; if (c <= 0x096F) return true;
    if (c < 0x09E6) return false; if (c <= 0x09EF) return true;
    if (c < 0x0A66) return false; if (c <= 0x0A6F) return true;
    if (c < 0x0AE6) return false; if (c <= 0x0AEF) return true;
    if (c < 0x0B66) return false; if (c <= 0x0B6F) return true;
    if (c < 0x0BE7) return false; if (c <= 0x0BEF) return true;
    if (c < 0x0C66) return false; if (c <= 0x0C6F) return true;
    if (c < 0x0CE6) return false; if (c <= 0x0CEF) return true;
    if (c < 0x0D66) return false; if (c <= 0x0D6F) return true;
    if (c < 0x0E50) return false; if (c <= 0x0E59) return true;
    if (c < 0x0ED0) return false; if (c <= 0x0ED9) return true;
    if (c < 0x0F20) return false; if (c <= 0x0F29) return true;

    return false;
}
```

同时识别合法和非法字符

第二次优化：避免重复验证

```
private Element( ) {}
static Element build(String name, String uri, String localName) {
Element result = new Element( );
String prefix = "";
int colon = name.indexOf(':');
if (colon >= 0) {
prefix = name.substring(0, colon);
}
result.prefix = prefix;
result.localName = localName;
// We do need to verify the URI here because parsers are
// allowing relative URIs which XOM forbids, for reasons
// of canonical XML if nothing else. But we only have to verify
// that it's an absolute base URI. I don't have to verify
// no conflicts.
if (! "".equals(uri)) Verifier.checkAbsoluteURIReference(uri);
result.URI = uri;
return result;
}
```

复杂度为O(1)的优化

验证16进制字符的switch语句：

```
switch(c) {
case '0': return true;
case '1': return true;
case '2': return true;
……
case 'd': return true;
case 'e': return true;
case 'f': return true;
}
return false;
}
```

大型switch语句

直接查表，按位储存信息

查表法具体实现

保存并复制二进制查找表：

```
<target name="compile-core"
depends="prepare, compile-jaxen"
description="Compile the source
code">
<javac srcdir="${build.src}"
destdir="${build.dest}">
<classpath
refid="compile.class.path"/>
</javac>
<copy
file="${build.src}/nu/xom/characters.dat"
tofile="${build.dest}/nu/xom/characters.dat"/>
</target>
```

装载二进制查找表：

```
private static byte[] flags = null;
static {
ClassLoader loader = Verifier.class.getClassLoader( );
if (loader != null) loadFlags(loader);
// If that didn't work, try a different ClassLoader
if (flags == null) {
loader = Thread.currentThread().getContextClassLoader( );
loadFlags(loader);
}
}
private static void loadFlags(ClassLoader loader) {
DataInputStream in = null;
try {
InputStream raw = loader.getResourceAsStream("nu/xom/characters.d
if (raw == null) {
throw new RuntimeException("Broken XOM installation: "
+ "could not load nu/xom/characters.dat");
}
in = new DataInputStream(raw);
flags = new byte[65536];
in.readFully(flags);
}
catch (IOException ex) {
throw new RuntimeException("Broken XOM installation: "
+ "could not load nu/xom/characters.dat");
}
finally {
try {
if (in != null) in.close( );
}
catch (IOException ex) {
// no big deal
}
}
}
```

查表法具体实现

使用查表法验证名字：

```
private static void loadFlags(ClassLoader loader) {
DataInputStream in = null;
try {
InputStream raw =
loader.getResourceAsStream("nu/xom/characters.dat");
if (raw == null) {
throw new RuntimeException("Broken XOM installation: "
+ "could not load nu/xom/characters.dat");
}
in = new DataInputStream(raw);
flags = new byte[65536];
in.readFully(flags);
}
catch (IOException ex) {
throw new RuntimeException("Broken XOM installation: "
+ "could not load nu/xom/characters.dat");
}
finally {
try {
if (in != null) in.close( );
}
catch (IOException ex) {
// no big deal
}}}
```

最后的优化：缓存

在缓存中记录命名空间的URI

```
private final static class URICache {
private final static int LOAD = 6;
private String[] cache = new String[LOAD];
private int position = 0;
synchronized boolean contains(String s) {
for (int i = 0; i < LOAD; i++) {
// Here I'm assuming the namespace URIs are interned.
// This is commonly but not always true. This won't
// break if they haven't been. Using equals( ) instead
// of == is faster when the namespace URIs haven't
been
// interned but slower if they have.
if (s == cache[i]) {
return true;
}
}
return false;
}
synchronized void put(String s) {
cache[position] = s;
position++;
if (position == LOAD) position = 0;
}
}
```