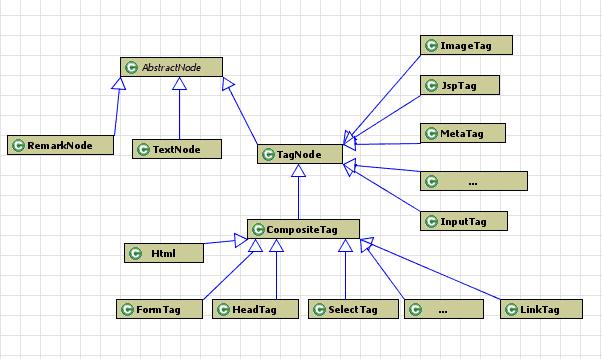
**Java解析HTML之HTMLParser**

HTMLParser具有小巧，快速的优点，缺点是相关文档比较少（英文的也少），很多功能需要自己摸索。对于初学者还是要费一些功夫的，而一旦上手以后，会发现HTMLParser的结构设计很巧妙，非常实用，基本你的各种需求都可以满足。  
    这里我根据自己这几个月来的经验，写了一点入门的东西，希望能对新学习HTMLParser的朋友们有所帮助。（不过当年高考本人语文只比及格高一分，所以文法方面的问题还希望大家多多担待）  
      
    HTMLParser的核心模块是org.htmlparser.Parser类，这个类实际完成了对于HTML页面的分析工作。这个类有下面几个构造函数：  
    public Parser ();  
    public Parser (Lexer lexer, ParserFeedback fb);  
   public Parser (URLConnection connection, ParserFeedback fb) throws ParserException;  
    public Parser (String resource, ParserFeedback feedback) throws ParserException;  
   public Parser (String resource) throws ParserException;  
    public Parser (Lexer lexer);  
    public Parser (URLConnection connection) throws ParserException;  
    和一个静态类 public static Parser createParser (String html, String charset);  
  
    对于大多数使用者来说，使用最多的是通过一个URLConnection或者一个保存有网页内容的字符串来初始化Parser，或者使用静态函数来生成一个Parser对象。ParserFeedback的代码很简单，是针对调试和跟踪分析过程的，一般不需要改变。而使用Lexer则是一个相对比较高级的话题，放到以后再讨论吧。  
    这里比较有趣的一点是，如果需要设置页面的编码方式的话，不使用Lexer就只有静态函数一个方法了。对于大多数中文页面来说，好像这是应该用得比较多的一个方法。  
  
   下面是初始化Parser的例子。

package com.baizeju.htmlparsertester;  
import java.io.BufferedReader;  
import java.io.InputStreamReader;  
import java.io.FileInputStream;  
import java.io.File;  
import java.net.HttpURLConnection;  
import java.net.URL;  
  
import org.htmlparser.visitors.TextExtractingVisitor;  
  
import org.htmlparser.Parser;  
  
/\*\*  
\* @author www.baizeju.com  
\*/  
public class Main {  
    private static String ENCODE = "GBK";  
    private static void message( String szMsg ) {  
        try{System.out.println(new String(szMsg.getBytes(ENCODE), System.getProperty("file.encoding"))); } catch(Exception e ){}  
    }  
    public static String openFile( String szFileName ) {  
        try {  
            BufferedReader bis = new BufferedReader(new InputStreamReader(new FileInputStream( new File(szFileName)), ENCODE) );  
            String szContent="";  
            String szTemp;  
              
            while ( (szTemp = bis.readLine()) != null) {  
                szContent+=szTemp+"\n";  
            }  
            bis.close();  
            return szContent;  
        }  
        catch( Exception e ) {  
            return "";  
        }  
    }  
      
   public static void main(String[] args) {  
          
        String szContent = openFile( "E:/My Sites/HTMLParserTester.html");  
          
        try{  
           **//Parser parser = Parser.createParser(szContent, ENCODE);  
            //Parser parser = new Parser( szContent );** Parser parser = new Parser( (HttpURLConnection) (new URL("http://127.0.0.1:8080/HTMLParserTester.html")).openConnection() );  
          
            TextExtractingVisitor visitor = new TextExtractingVisitor();  
            parser.visitAllNodesWith(visitor);

//获得了网页中的能容  
            String textInPage = visitor.getExtractedText();  
  
            message(textInPage);  
        }  
        catch( Exception e ) {              
        }  
    }  
}  
加重的部分测试了几种不同的初始化方法，后面的显示了结果。大家看到能Parser出内容就可以了，如何操作访问Parser的内容我们在后面讨论。

HTMLParser将解析过的信息保存为一个树的结构。Node是信息保存的数据类型基础。  
请看Node的定义：  
public interface Node extends Cloneable;  
  
**Node中包含的方法有几类：**  
**对于树型结构进行遍历的函数，这些函数最容易理解：**  
Node getParent ()：取得父节点  
NodeList getChildren ()：取得子节点的列表  
Node getFirstChild ()：取得第一个子节点  
Node getLastChild ()：取得最后一个子节点  
Node getPreviousSibling ()：取得前一个兄弟（不好意思，英文是兄弟姐妹，直译太麻烦而且不符合习惯，对不起女同胞了）  
Node getNextSibling ()：取得下一个兄弟节点  
**取得Node内容的函数：**  
String getText ()：取得文本  
String toPlainTextString()：取得纯文本信息。  
String toHtml () ：取得HTML信息（原始HTML）  
String toHtml (boolean verbatim)：取得HTML信息（原始HTML）  
String toString ()：取得字符串信息（原始HTML）  
Page getPage ()：取得这个Node对应的Page对象  
int getStartPosition ()：取得这个Node在HTML页面中的起始位置  
int getEndPosition ()：取得这个Node在HTML页面中的结束位置  
**用于Filter过滤的函数：**  
void collectInto (NodeList list, NodeFilter filter)：基于filter的条件对于这个节点进行过滤，符合条件的节点放到list中。  
**用于Visitor遍历的函数：**  
void accept (NodeVisitor visitor)：对这个Node应用visitor  
**用于修改内容的函数，这类用得比较少：**  
void setPage (Page page)：设置这个Node对应的Page对象  
void setText (String text)：设置文本  
void setChildren (NodeList children)：设置子节点列表  
**其他函数：**  
void doSemanticAction ()：执行这个Node对应的操作（只有少数Tag有对应的操作）  
Object clone ()：接口Clone的抽象函数。  
  
实际我们用HTMLParser最多的是处理HTML页面，Filter或Visitor相关的函数是必须的，然后第一类和第二类函数是用得最多的。第一类函数比较容易理解，下面用例子说明一下第二类函数。  
**下面是用于测试的HTML文件：**  
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
<head><meta http-equiv="Content-Type" content="text/html; charset=gb2312"><title>白泽居-www.baizeju.com</title></head>  
<html xmlns="http://www.w3.org/1999/xhtml">  
<body >  
<div id="top\_main">  
    <div id="logoindex">  
        <!--这是注释-->  
        白泽居-www.baizeju.com  
<a href="http://www.baizeju.com">白泽居-www.baizeju.com</a>  
    </div>  
    白泽居-www.baizeju.com  
</div>  
</body>  
</html>  
  
**测试代码：**  
/\*\*  
\* @author www.baizeju.com  
\*/  
  
package com.baizeju.htmlparsertester;  
import java.io.BufferedReader;  
import java.io.InputStreamReader;  
import java.io.FileInputStream;  
import java.io.File;  
import java.net.HttpURLConnection;  
import java.net.URL;  
  
import org.htmlparser.Node;  
import org.htmlparser.util.NodeIterator;  
import org.htmlparser.Parser;  
  
/\*\*  
\* @author www.baizeju.com  
\*/  
public class Main {  
    private static String ENCODE = "GBK";  
    private static void message( String szMsg ) {  
        try{ System.out.println(new String(szMsg.getBytes(ENCODE), System.getProperty("file.encoding"))); }    catch(Exception e ){}  
    }  
    public static String openFile( String szFileName ) {  
        try {  
            BufferedReader bis = new BufferedReader(new InputStreamReader(new FileInputStream( new File(szFileName)),    ENCODE) );  
            String szContent="";  
            String szTemp;  
              
            while ( (szTemp = bis.readLine()) != null) {  
                szContent+=szTemp+"\n";  
            }  
            bis.close();  
            return szContent;  
        }  
        catch( Exception e ) {  
            return "";  
        }  
    }  
      
   public static void main(String[] args) {  
          
        try{  
            Parser parser = new Parser( (HttpURLConnection) (new URL("http://127.0.0.1:8080/HTMLParserTester.html")).openConnection() );  
          
            for (NodeIterator i = parser.elements (); i.hasMoreNodes(); ) {  
                Node node = i.nextNode();  
                message("getText:"+node.getText());  
                message("getPlainText:"+node.toPlainTextString());  
                message("toHtml:"+node.toHtml());  
                message("toHtml(true):"+node.toHtml(true));  
                message("toHtml(false):"+node.toHtml(false));  
                message("toString:"+node.toString());  
                message("=================================================");  
            }              
        }  
        catch( Exception e ) {       
            System.out.println( "Exception:"+e );  
        }  
    }  
}  
  
**输出结果：**  
getText:!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"  
getPlainText:  
toHtml:<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
toHtml(true):<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
toHtml(false):<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
toString:Doctype Tag : !DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd; begins at : 0; ends at : 121  
=================================================  
getText:  
  
getPlainText:  
  
toHtml:  
  
toHtml(true):  
  
toHtml(false):  
  
toString:Txt (121[0,121],123[1,0]): \n  
=================================================  
getText:head  
getPlainText:白泽居-www.baizeju.com  
toHtml:<head><meta http-equiv="Content-Type" content="text/html; charset=gb2312"><title>白泽居-www.baizeju.com</title></head>  
toHtml(true):<head><meta http-equiv="Content-Type" content="text/html; charset=gb2312"><title>白泽居-www.baizeju.com</title></head>  
toHtml(false):<head><meta http-equiv="Content-Type" content="text/html; charset=gb2312"><title>白泽居-www.baizeju.com</title></head>  
toString:HEAD: Tag (123[1,0],129[1,6]): head  
Tag (129[1,6],197[1,74]): meta http-equiv="Content-Type" content="text/html; ...  
Tag (197[1,74],204[1,81]): title  
    Txt (204[1,81],223[1,100]): 白泽居-www.baizeju.com  
    End (223[1,100],231[1,108]): /title  
End (231[1,108],238[1,115]): /head  
  
=================================================  
getText:  
  
getPlainText:  
  
toHtml:  
  
toHtml(true):  
  
toHtml(false):  
  
toString:Txt (238[1,115],240[2,0]): \n  
=================================================  
getText:html xmlns="http://www.w3.org/1999/xhtml"  
getPlainText:  
  
  
          
                  
                白泽居-www.baizeju.com  
白泽居-www.baizeju.com  
          
        白泽居-www.baizeju.com  
  
  
  
toHtml:<html xmlns="http://www.w3.org/1999/xhtml">  
<body >  
<div id="top\_main">  
        <div id="logoindex">  
                <!--这是注释-->  
                白泽居-www.baizeju.com  
<a href="http://www.baizeju.com">白泽居-www.baizeju.com</a>  
        </div>  
        白泽居-www.baizeju.com  
</div>  
</body>  
</html>  
toHtml(true):<html xmlns="http://www.w3.org/1999/xhtml">  
<body >  
<div id="top\_main">  
        <div id="logoindex">  
                <!--这是注释-->  
                白泽居-www.baizeju.com  
<a href="http://www.baizeju.com">白泽居-www.baizeju.com</a>  
        </div>  
        白泽居-www.baizeju.com  
</div>  
</body>  
</html>  
toHtml(false):<html xmlns="http://www.w3.org/1999/xhtml">  
<body >  
<div id="top\_main">  
        <div id="logoindex">  
                <!--这是注释-->  
                白泽居-www.baizeju.com  
<a href="http://www.baizeju.com">白泽居-www.baizeju.com</a>  
        </div>  
        白泽居-www.baizeju.com  
</div>  
</body>  
</html>  
toString:Tag (240[2,0],283[2,43]): html xmlns="http://www.w3.org/1999/xhtml"  
Txt (283[2,43],285[3,0]): \n  
Tag (285[3,0],292[3,7]): body   
    Txt (292[3,7],294[4,0]): \n  
    Tag (294[4,0],313[4,19]): div id="top\_main"  
      Txt (313[4,19],316[5,1]): \n\t  
      Tag (316[5,1],336[5,21]): div id="logoindex"  
        Txt (336[5,21],340[6,2]): \n\t\t  
        Rem (340[6,2],351[6,13]): 这是注释  
        Txt (351[6,13],376[8,0]): \n\t\t白泽居-www.baizeju.com\n  
        Tag (376[8,0],409[8,33]): a href="http://www.baizeju.com"  
          Txt (409[8,33],428[8,52]): 白泽居-www.baizeju.com  
          End (428[8,52],432[8,56]): /a  
        Txt (432[8,56],435[9,1]): \n\t  
        End (435[9,1],441[9,7]): /div  
      Txt (441[9,7],465[11,0]): \n\t白泽居-www.baizeju.com\n  
      End (465[11,0],471[11,6]): /div  
    Txt (471[11,6],473[12,0]): \n  
    End (473[12,0],480[12,7]): /body  
Txt (480[12,7],482[13,0]): \n  
End (482[13,0],489[13,7]): /html  
  
=================================================  
  
  
对于第一个Node的内容，对应的就是第一行<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">，这个比较好理解。  
从这个输出结果中，也可以看出内容的树状结构。或者说是树林结构。在Page内容的第一层Tag，如DOCTYPE，head和html，分别形成了一个最高层的Node节点（很多人可能对第二个和第四个Node的内容有点奇怪。实际上这两个Node就是两个换行符号。HTMLParser把HTML页面内容中的所有换行，空格，Tab等都转换成了相应的Tag，所以就出现了这样的Node。虽然内容少但是级别高，呵呵）  
getPlainTextString是把用户可以看到的内容都包含了。有趣的有两点，一是<head>标签中的Title内容是在plainText中的，可能在标题中可见的也算可见吧。另外就是象前面说的，HTML内容中的换行符什么的，也都成了plainText，这个逻辑上好像有点问题。  
  
另外可能大家发现toHtml，toHtml(true)和toHtml(false)的结果没什么区别。实际也是这样的，如果跟踪HTMLParser的代码就可以发现，Node的子类是AbstractNode，其中实现了toHtml()的代码，直接调用toHtml(false)，而AbstractNode的三个子类RemarkNode，TagNode和TextNode中，toHtml(boolean verbatim)的实现中，都没有处理verbatim参数，所以三个函数的结果是一模一样的。如果你不需要实现你自己的什么特殊处理，简单使用toHtml就可以了。  
  
HTML的Node类继承关系如下图（这个是从别的文章Copy的）：

  
 AbstractNodes是Node的直接子类，也是一个抽象类。它的三个直接子类实现是RemarkNode，用于保存注释。在输出结果的toString部分中可以看到有一个"Rem (345[6,2],356[6,13]): 这是注释"，就是一个RemarkNode。TextNode也很简单，就是用户可见的文字信息。TagNode是最复杂的，包含了HTML语言中的所有标签，而且可以扩展（[扩展HTMLParser 对自定义标签的处理能力](http://www.baizeju.com/html/HTMLParser/200806/06-71.html)）。TagNode包含两类，一类是简单的Tag，实际就是不能包含其他Tag的标签，只能做叶子节点。另一类是CompositeTag，就是可以包含其他Tag，是分支节点

HTMLParser遍历了网页的内容以后，以树（森林）结构保存了结果。HTMLParser访问结果内容的方法有两种。使用Filter和使用Visitor。  
  
**（一）Filter类**  
顾名思义，Filter就是对于结果进行过滤，取得需要的内容。HTMLParser在org.htmlparser.filters包之内一共定义了16个不同的Filter，也可以分为几类。  
[**判断类Filter：**](http://www.baizeju.com/html/HTMLParser/200807/07-121.html#%E5%88%A4%E6%96%AD%E7%B1%BBFilter)  
TagNameFilter  
HasAttributeFilter  
HasChildFilter  
HasParentFilter  
HasSiblingFilter  
IsEqualFilter  
[**逻辑运算Filter：**](http://www.baizeju.com/html/HTMLParser/200807/07-121.html#%E9%80%BB%E8%BE%91%E8%BF%90%E7%AE%97Filter)  
AndFilter  
NotFilter  
OrFilter  
XorFilter  
[**其他Filter：**](http://www.baizeju.com/html/HTMLParser/200807/07-121.html#%E5%85%B6%E4%BB%96Filter)  
NodeClassFilter  
StringFilter  
LinkStringFilter  
LinkRegexFilter  
RegexFilter  
CssSelectorNodeFilter  
  
所有的Filter类都实现了org.htmlparser.NodeFilter接口。这个接口只有一个主要函数：  
boolean accept (Node node);  
各个子类分别实现这个函数，用于判断输入的Node是否符合这个Filter的过滤条件，如果符合，返回true，否则返回false。  
  
**（二）判断类Filter**  
**2.1 TagNameFilter**  
TabNameFilter是最容易理解的一个Filter，根据Tag的名字进行过滤。  
  
**下面是用于测试的HTML文件：**  
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
<head><meta http-equiv="Content-Type" content="text/html; charset=gb2312"><title>白泽居-www.baizeju.com</title>< /head>  
<html xmlns="http://www.w3.org/1999/xhtml">  
<body >  
<div id="top\_main">  
    <div id="logoindex">  
        <!--这是注释-->  
        白泽居-www.baizeju.com  
<a href="http://www.baizeju.com">白泽居-www.baizeju.com</a>  
    </div>  
    白泽居-www.baizeju.com  
</div>  
</body>  
</html>  
**测试代码：**（这里只列出了Main函数，全部代码请参考 [HTMLParser使用入门（2）- Node内容](http://www.baizeju.com/html/HTMLParser/200807/04-119.html)，自己添加import部分）  
public static void main(String[] args) {  
          
        try{  
            Parser parser = new Parser( (HttpURLConnection) (new URL("http://127.0.0.1:8080/HTMLParserTester.html")).openConnection() );  
          
            // 这里是控制测试的部分，后面的例子修改的就是这个地方。  
           ***NodeFilter filter = new TagNameFilter ("DIV");  
            NodeList nodes = parser.extractAllNodesThatMatch(filter);***  
              
            if(nodes!=null) {  
                for (int i = 0; i < nodes.size(); i++) {  
                    Node textnode = (Node) nodes.elementAt(i);  
                      
                    message("getText:"+textnode.getText());  
                    message("=================================================");  
                }  
            }              
        }  
        catch( Exception e ) {       
            e.printStackTrace();  
        }  
    }  
**输出结果：**getText:div id="top\_main"  
=================================================  
getText:div id="logoindex"  
=================================================可以看出文件中两个Div节点都被取出了。下面可以针对这两个DIV节点进行操作  
  
**2.2 HasChildFilter**  
下面让我们看看HasChildFilter。刚刚看到这个Filter的时候，我想当然地认为这个Filter返回的是有Child的Tag。直接初始化了一个  
NodeFilter filter = new HasChildFilter();  
结果调用NodeList nodes = parser.extractAllNodesThatMatch(filter);的时候HasChildFilter内部直接发生NullPointerException。读了一下HasChildFilter的代码，才发现，实际HasChildFilter是返回有符合条件的子节点的节点，需要另外一个Filter作为过滤子节点的参数。缺省的构造函数虽然可以初始化，但是由于子节点的Filter是null，所以使用的时候发生了Exception。从这点来看，HTMLParser的代码还有很多可以优化的的地方。呵呵。  
  
**修改代码：**  
NodeFilter innerFilter = new TagNameFilter ("DIV");  
NodeFilter filter = new HasChildFilter(innerFilter);  
NodeList nodes = parser.extractAllNodesThatMatch(filter);  
**输出结果：**  
getText:body   
=================================================  
getText:div id="top\_main"  
=================================================  
可以看到，输出的是两个有DIV子Tag的Tag节点。（body有子节点DIV "top\_main"，"top\_main"有子节点"logoindex"。  
  
注意HasChildFilter还有一个构造函数：  
public HasChildFilter (NodeFilter filter, boolean recursive)  
如果recursive是false，则只对第一级子节点进行过滤。比如前面的例子，body和top\_main都是在第一级的子节点里就有DIV节点，所以匹配上了。如果我们用下面的方法调用：  
NodeFilter filter = new HasChildFilter( innerFilter, true );  
**输出结果：**  
getText:html xmlns="http://www.w3.org/1999/xhtml"  
=================================================  
getText:body   
=================================================  
getText:div id="top\_main"  
=================================================  
可以看到输出结果中多了一个html xmlns="http://www.w3.org/1999/xhtml"，这个是整个HTML页面的节点（根节点），虽然这个节点下直接没有DIV节点，但是它的子节点body下面有DIV节点，所以它也被匹配上了。  
  
**2.3 HasAttributeFilter**  
HasAttributeFilter有3个构造函数：  
public HasAttributeFilter ();  
public HasAttributeFilter (String attribute);  
public HasAttributeFilter (String attribute, String value);  
这个Filter可以匹配出包含指定名字的属性，或者制定属性为指定值的节点。还是用例子说明比较容易。  
  
**调用方法1:**  
NodeFilter filter = new HasAttributeFilter();  
NodeList nodes = parser.extractAllNodesThatMatch(filter);  
**输出结果：**  
什么也没有输出。  
  
**调用方法2:**  
NodeFilter filter = new HasAttributeFilter( "id" );  
NodeList nodes = parser.extractAllNodesThatMatch(filter);  
**输出结果：**  
getText:div id="top\_main"  
=================================================  
getText:div id="logoindex"  
=================================================  
 **调用方法3:**  
NodeFilter filter = new HasAttributeFilter( "id", "logoindex" );  
NodeList nodes = parser.extractAllNodesThatMatch(filter);  
**输出结果：**  
getText:div id="logoindex"  
=================================================  
  
很简单吧。呵呵  
  
**2.4 其他判断列Filter**  
HasParentFilter和HasSiblingFilter的功能与HasChildFilter类似（也都有三个初始化方法），大家自己试一下就应该了解了。  
  
IsEqualFilter的构造函数参数是一个Node：  
public IsEqualFilter (Node node) {  
    mNode = node;  
}  
accept函数也很简单：  
public boolean accept (Node node)    {  
    return (mNode == node);  
}  
不需要过多说明了。  
  
  
**（三）逻辑运算Filter**  
前面介绍的都是简单的Filter，只能针对某种单一类型的条件进行过滤。HTMLParser支持对于简单类型的Filter进行组合，从而实现复杂的条件。原理和一般编程语言的逻辑运算是一样的。 **3.1 AndFilter**  
AndFilter可以把两种Filter进行组合，只有同时满足条件的Node才会被过滤。  
**测试代码：**  
NodeFilter filterID = new HasAttributeFilter( "id" );  
NodeFilter filterChild = new HasChildFilter(filterA);  
NodeFilter filter = new AndFilter(filterID, filterChild);  
**输出结果：**getText:div id="logoindex"  
================================================= **3.2 OrFilter**  
把前面的AndFilter换成OrFilter  
**测试代码：**  
NodeFilter filterID = new HasAttributeFilter( "id" );  
NodeFilter filterChild = new HasChildFilter(filterA);  
NodeFilter filter = new OrFilter(filterID, filterChild);  
**输出结果：**  
getText:div id="top\_main"  
=================================================  
getText:div id="logoindex"  
=================================================  
  
**3.3 NotFilter**  
把前面的AndFilter换成NotFilter  
**测试代码：**  
NodeFilter filterID = new HasAttributeFilter( "id" );  
NodeFilter filterChild = new HasChildFilter(filterA);  
NodeFilter filter = new NotFilter(new OrFilter(filterID, filterChild));  
**输出结果：**  
getText:!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"  
=================================================  
getText:  
  
=================================================  
getText:head  
=================================================  
getText:meta http-equiv="Content-Type" content="text/html; charset=gb2312"  
=================================================  
getText:title  
=================================================  
getText:白泽居-www.baizeju.com  
=================================================  
getText:/title  
=================================================  
getText:/head  
=================================================  
getText:  
  
=================================================  
getText:html xmlns="http://www.w3.org/1999/xhtml"  
=================================================  
getText:  
  
=================================================  
getText:body   
=================================================  
getText:  
  
=================================================  
getText:  
          
=================================================  
getText:  
                  
=================================================  
getText:这是注释  
=================================================  
getText:  
                白泽居-www.baizeju.com  
  
=================================================  
getText:a href="http://www.baizeju.com"  
=================================================  
getText:白泽居-www.baizeju.com  
=================================================  
getText:/a  
=================================================  
getText:  
          
=================================================  
getText:/div  
=================================================  
getText:  
        白泽居-www.baizeju.com  
  
=================================================  
getText:/div  
=================================================  
getText:  
  
=================================================  
getText:/body  
=================================================  
getText:  
  
=================================================  
getText:/html  
=================================================  
getText:  
  
=================================================  
  
除了前面3.2中输出的几个Tag，其余的Tag都在这里了。  
  
  
**3.4 XorFilter**  
把前面的AndFilter换成NotFilter  
**测试代码：**  
NodeFilter filterID = new HasAttributeFilter( "id" );  
NodeFilter filterChild = new HasChildFilter(filterA);  
NodeFilter filter = new XorFilter(filterID, filterChild);  
**输出结果：**  
getText:div id="top\_main"  
=================================================  
  
**（四）其他Filter：**  
**4.1 NodeClassFilter**  
这个Filter用于判断节点类型是否是某个特定的Node类型。在[HTMLParser使用入门（2）- Node内容](http://www.baizeju.com/html/HTMLParser/200807/04-119.html) 中我们已经了解了Node的不同类型，这个Filter就可以针对类型进行过滤。  
**测试代码：**  
           NodeFilter filter = new NodeClassFilter(RemarkNode.class);//注释的意思  
            NodeList nodes = parser.extractAllNodesThatMatch(filter);  
**输出结果：**  
getText:这是注释  
=================================================  
可以看到只有RemarkNode（注释）被输出了。  
  
**4.2 StringFilter**  
这个Filter用于过滤显示字符串中包含指定内容的Tag。注意是可显示的字符串，不可显示的字符串中的内容（例如注释，链接等等）不会被显示。  
**修改一下例子代码：**  
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
<head><meta http-equiv="Content-Type" content="text/html; charset=gb2312"><title>白泽居-title-www.baizeju.com</title></head>  
<html xmlns="http://www.w3.org/1999/xhtml">  
<body >  
<div id="top\_main">  
    <div id="logoindex">  
        <!--这是注释 白泽居-www.baizeju.com -->  
        白泽居-字符串1-www.baizeju.com  
<a href="http://www.baizeju.com">白泽居-链接文本-www.baizeju.com</a>  
    </div>  
    白泽居-字符串2-www.baizeju.com  
</div>  
</body>  
</html>  
 **测试代码：**  
           NodeFilter filter = new StringFilter("www.baizeju.com");  
            NodeList nodes = parser.extractAllNodesThatMatch(filter);  
**输出结果：**  
getText:白泽居-title-www.baizeju.com  
=================================================  
getText:  
                白泽居-字符串1-www.baizeju.com  
  
=================================================  
getText:白泽居-链接文本-www.baizeju.com  
=================================================  
getText:  
        白泽居-字符串2-www.baizeju.com  
  
=================================================  
可以看到包含title，两个内容字符串和链接的文本字符串的Tag都被输出了，但是注释和链接Tag本身没有输出。  
  
**4.3 LinkStringFilter**  
这个Filter用于判断链接中是否包含某个特定的字符串，可以用来过滤出指向某个特定网站的链接。  
**测试代码：**  
           NodeFilter filter = new LinkStringFilter("www.baizeju.com");  
            NodeList nodes = parser.extractAllNodesThatMatch(filter);  
**输出结果：**  
getText:a href="http://www.baizeju.com"  
=================================================  
  
**4.4 其他几个Filter**  
其他几个Filter也是根据字符串对不同的域进行判断，与前面这些的区别主要就是支持正则表达式。这个不在本文的讨论范围以内，大家可以自己实验一下。

HTMLParser遍历了网页的内容以后，以树（森林）结构保存了结果。HTMLParser访问结果内容的方法有两种。使用Filter和使用Visitor。  
下面介绍使用Visitor访问内容的方法。  
  
**4.1 NodeVisitor**  
从简单方面的理解，Filter是根据某种条件过滤取出需要的Node再进行处理。Visitor则是遍历内容树的每一个节点，对于符合条件的节点进行处理。实际的结果异曲同工，两种不同的方法可以达到相同的结果。  
下面是一个最常见的NodeVisitro的例子。  
**测试代码：**  
    public static void main(String[] args) {  
        try{  
            Parser parser = new Parser( (HttpURLConnection) (new URL("http://127.0.0.1:8080/HTMLParserTester.html")).openConnection() );  
  
            NodeVisitor visitor = new NodeVisitor( false, false ) {  
                public void visitTag(Tag tag) {  
                   message("This is Tag:"+tag.getText());  
                }  
                public void visitStringNode (Text string)    {  
                     message("This is Text:"+string);  
                }  
                public void visitRemarkNode (Remark remark) {  
                     message("This is Remark:"+remark.getText());  
                }  
                public void beginParsing () {  
                    message("beginParsing");  
                }  
                public void visitEndTag (Tag tag){  
                    message("visitEndTag:"+tag.getText());  
                }  
                public void finishedParsing () {  
                    message("finishedParsing");  
                }  
            };  
  
            parser.visitAllNodesWith(visitor);  
        }  
        catch( Exception e ) {       
            e.printStackTrace();  
        }  
    }  
**输出结果：**  
beginParsing  
This is Tag:!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"  
This is Text:Txt (121[0,121],123[1,0]): \n  
This is Text:Txt (244[1,121],246[2,0]): \n  
finishedParsing  
  
可以看到，开始遍历所以的节点以前，beginParsing先被调用，然后处理的是中间的Node，最后在结束遍历以前，finishParsing被调用。因为我设置的 recurseChildren和recurseSelf都是false，所以Visitor没有访问子节点也没有访问根节点的内容。中间输出的两个\n就是我们在[HTMLParser使用详解（1）- 初始化Parser](http://www.baizeju.com/html/HTMLParser/200807/04-118.html) 中讨论过的最高层的那两个换行。  
  
我们先把recurseSelf设置成true，看看会发生什么。  
NodeVisitor visitor = new NodeVisitor( false, **true**) {  
**输出结果：**beginParsing  
This is Tag:!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"  
This is Text:Txt (121[0,121],123[1,0]): \n  
This is Tag:head  
This is Text:Txt (244[1,121],246[2,0]): \n  
This is Tag:html xmlns="http://www.w3.org/1999/xhtml"  
finishedParsing可以看到，HTML页面的第一层节点都被调用了。我们再用下面的方法调用看看：  
NodeVisitor visitor = new NodeVisitor( **true, false**) {  
**输出结果：**beginParsing  
This is Tag:!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"  
This is Text:Txt (121[0,121],123[1,0]): \n  
This is Tag:meta http-equiv="Content-Type" content="text/html; charset=gb2312"  
This is Text:Txt (204[1,81],229[1,106]): 白泽居-title-www.baizeju.com  
visitEndTag:/title  
visitEndTag:/head  
This is Text:Txt (244[1,121],246[2,0]): \n  
This is Text:Txt (289[2,43],291[3,0]): \n  
This is Text:Txt (298[3,7],300[4,0]): \n  
This is Text:Txt (319[4,19],322[5,1]): \n\t  
This is Text:Txt (342[5,21],346[6,2]): \n\t\t  
This is Remark:这是注释 白泽居-www.baizeju.com   
This is Text:Txt (378[6,34],408[8,0]): \n\t\t白泽居-字符串1-www.baizeju.com\n  
This is Text:Txt (441[8,33],465[8,57]): 白泽居-链接文本-www.baizeju.com  
visitEndTag:/a  
This is Text:Txt (469[8,61],472[9,1]): \n\t  
visitEndTag:/div  
This is Text:Txt (478[9,7],507[11,0]): \n\t白泽居-字符串2-www.baizeju.com\n  
visitEndTag:/div  
This is Text:Txt (513[11,6],515[12,0]): \n  
visitEndTag:/body  
This is Text:Txt (522[12,7],524[13,0]): \n  
visitEndTag:/html  
finishedParsing  
可以看到，所有的子节点都出现了，除了刚刚例子里面的两个最上层节点This is Tag:head和This is Tag:html xmlns="http://www.w3.org/1999/xhtml"。  
 **想让它们都出来，只需要**NodeVisitor visitor = new NodeVisitor( **true, true**) {  
**输出结果：**  
beginParsing  
This is Tag:!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"  
This is Text:Txt (121[0,121],123[1,0]): \n  
This is Tag:head  
This is Tag:meta http-equiv="Content-Type" content="text/html; charset=gb2312"  
This is Tag:title  
This is Text:Txt (204[1,81],229[1,106]): 白泽居-title-www.baizeju.com  
visitEndTag:/title  
visitEndTag:/head  
This is Text:Txt (244[1,121],246[2,0]): \n  
This is Tag:html xmlns="http://www.w3.org/1999/xhtml"  
This is Text:Txt (289[2,43],291[3,0]): \n  
This is Tag:body   
This is Text:Txt (298[3,7],300[4,0]): \n  
This is Tag:div id="top\_main"  
This is Text:Txt (319[4,19],322[5,1]): \n\t  
This is Tag:div id="logoindex"  
This is Text:Txt (342[5,21],346[6,2]): \n\t\t  
This is Remark:这是注释 白泽居-www.baizeju.com   
This is Text:Txt (378[6,34],408[8,0]): \n\t\t白泽居-字符串1-www.baizeju.com\n  
This is Tag:a href="http://www.baizeju.com"  
This is Text:Txt (441[8,33],465[8,57]): 白泽居-链接文本-www.baizeju.com  
visitEndTag:/a  
This is Text:Txt (469[8,61],472[9,1]): \n\t  
visitEndTag:/div  
This is Text:Txt (478[9,7],507[11,0]): \n\t白泽居-字符串2-www.baizeju.com\n  
visitEndTag:/div  
This is Text:Txt (513[11,6],515[12,0]): \n  
visitEndTag:/body  
This is Text:Txt (522[12,7],524[13,0]): \n  
visitEndTag:/html  
finishedParsing  
哈哈，这下调用清楚了，大家在需要处理的地方增加自己的代码好了。 **4.2 其他Visitor**  
HTMLParser还定义了几个其他的Visitor。HtmlPage，NodeVisitor，ObjectFindingVisitor，StringFindingVisitor，TagFindingVisitor，TextExtractingVisitor，UrlModifyingVisitor，它们都是NodeVisitor的子类，实现了一些特定的功能。笔者个人的感觉是没什么用处，如果你需要什么特定的功能，还不如自己写一个，想在这些里面找到适合你需要的，化的时间可能更多。反正大家看看代码就发现，它们每个都没几行真正有效的代码。HTMLParser 是一个用来解析 HTML 文档的开放源码项目，它具有小巧、快速、使用简单的特点以及拥有强大的功能。对该项目还不了解的朋友可以参照 2004 年三月份我发表的文章--《[从HTML中攫取你所需的信息](http://www.ibm.com/developerWorks/cn/java/l-html-parser/)》，这篇文章介绍如何通过 HTMLParser来提取 HTML 文档中的文本数据以及提取出文档中的所有链接或者是图片等信息。

现在该项目的最新版本是 Integration Build 1.6，与之前版本的差别在于代码结构的调整、当然也有一些功能的提升以及 BugFix，同时对字符集的处理也更加自动了。比较遗憾的该项目并没有详尽的使用文档，你只能借助于它的 API 文档、一两个简单例子以及源码来熟悉它。

如果是 HTML 文档，那么用 HTMLParser 已经差不多可以满足你至少 90％ 的需求。一个 HTML 文档中可能出现的标签差不多在 HTMLParser 中都有对应的类，甚至包括一些动态的脚本标签，例如 <%...%> 这种 JSP 和 ASP用到的标签都有相应的 JspTag 对应。HTMLParser 的强大功能还体现在你可以修改每个标签的属性或者它所包含的文本内容并生成新的 HTML 文档，比如你可以文档中的链接地址偷偷的改成你自己的地址等等。关于 HTMLParser的强大功能，其实上一篇文章已经介绍很多，这里不再累赘，我们今天要讲的是另外一个用途--处理自定义标签。

首先我们先解释一下什么叫自定义标签，我把所有不是 HTML 脚本语言中定义的标签称之为自定义标签，比如可以是 <scriptlet>、<book> 等等，这是我们自己创造出来的标签。你可能会很奇怪，因为这些标签一旦用在HTML 文档中是没有任何效果的，那么我们换另外一个例子，假如你要解析的不是 HTML 文档，而是一个WML（Wireless Markup Lauguage）文档呢？WML 文档中的 card，anchor 等标签 HTMLParser 是没有现成的标签类来处理的。还有就是你同样可以用 HTMLParser 来处理 XML 文档，而 XML 文档中所有的标签都是你自己定义的。

为了使我们的例子更具有代表意义，接下来我们将给出一段代码用来解析出 WML 文档中的所有链接，了解WML 文档的人都知道，WML 文档中除了与 HTML 文档相同的链接写法外，还多了一种标签叫 <anchor>，例如在一个 WML 文档我们可以用下面两种方式来表示一个链接。

|  |
| --- |
| <a href="http://www.javayou.com?cat\_id=1">Java自由人</a>  或者：  <anchor>  Java自由人      <go href="http://www.javayou.com" method="get">          <postfield name="cat\_id" value="1"/>  </go>  </anchor> |

（更多的时候使用 anchor 的链接用来提交一个表单。） 如果我们还是使用 LinkTag 来遍历整个 WML 文档的话，那 Anchor 中的链接将会被我们所忽略掉。

下面我们先给出一个简单的例子，然后再叙述其中的道理。这个例子包含两个文件，一个是WML 的测试脚本文件 test.wml，另外一个是 Java 程序文件 HyperLinkTrace.java，内容如下：

|  |
| --- |
|  |

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | [**回页首**](http://www.ibm.com/developerworks/cn/opensource/os-htmlparser/#main) | |

1. test.wml

|  |
| --- |
| <?xml version="1.0"?>  <!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"  "http://www.wapforum.org/DTD/wml\_1.1.xml">  <wml>  <card title="Java自由人登录">  <p>   用户名:<input type="text" name="username" size="15"/>       密码:<input type="text" name="password" size="15"/>   <br/>   <anchor>现在登录    <go href="/wap/user.do" method="get">        <postfield name="name" value="$(username)"/>        <postfield name="password" value="$(password)"/>        <postfield name="eventSubmit\_Login" value="WML"/>    </go>   </anchor><br/>   <a href="/wap/index.vm">返回首页</a>  </p>  </card>  </wml> |

test.wml 中的粗体部分是我们需要提取出来的链接。

|  |
| --- |
|  |

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | [**回页首**](http://www.ibm.com/developerworks/cn/opensource/os-htmlparser/#main) | |

2. HyperLinkTrace.java

|  |
| --- |
| package demo.htmlparser;  import java.io.BufferedReader;  import java.io.File;  import java.io.FileReader;  import java.net.URL;  import org.htmlparser.Node;  import org.htmlparser.NodeFilter;  import org.htmlparser.Parser;  import org.htmlparser.PrototypicalNodeFactory;  import org.htmlparser.tags.CompositeTag;  import org.htmlparser.tags.LinkTag;  import org.htmlparser.util.NodeList;  /\*\*   \* 用来遍历WML文档中的所有超链接   \* @author Winter Lau   \*/  public class HyperLinkTrace {   public static void main(String[] args) throws Exception {    //初始化HTMLParser    Parser parser = new Parser();    parser.setEncoding("8859\_1");    parser.setInputHTML(getWmlContent());      //注册新的结点解析器    PrototypicalNodeFactory factory = new PrototypicalNodeFactory ();    factory.registerTag(new WmlGoTag ());    parser.setNodeFactory(factory);    //遍历符合条件的所有节点    NodeList nlist = parser.extractAllNodesThatMatch(lnkFilter);    for(int i=0;i<nlist.size();i++){     CompositeTag node = (CompositeTag)nlist.elementAt(i);     if(node instanceof LinkTag){      LinkTag link = (LinkTag)node;      System.out.println("LINK: \t" + link.getLink());     }     else if(node instanceof WmlGoTag){      WmlGoTag go = (WmlGoTag)node;      System.out.println("GO: \t" + go.getLink());     }    }   }   /\*\*    \* 获取测试的WML脚本内容    \* @return    \* @throws Exception    \*/   static String getWmlContent() throws Exception{    URL url = ParserTester.class.getResource("/demo/htmlparser/test.wml");    File f = new File(url.toURI());    BufferedReader in = new BufferedReader(new FileReader(f));    StringBuffer wml = new StringBuffer();    do{     String line = in.readLine();     if(line==null)      break;     if(wml.length()>0)      wml.append("\r\n");     wml.append(line);    }while(true);    return wml.toString();   }   /\*\*    \* 解析出所有的链接，包括行为<a>与<go>    \*/   static NodeFilter lnkFilter = new NodeFilter() {    public boolean accept(Node node) {     if(node instanceof WmlGoTag)      return true;     if(node instanceof LinkTag)      return true;     return false;    }   };     /\*\*    \* WML文档的GO标签解析器    \* @author Winter Lau    \*/   static class WmlGoTag extends CompositeTag {       private static final String[] mIds = new String[] {"GO"};       private static final String[] mEndTagEnders = new String[] {"ANCHOR"};       public String[] getIds (){           return (mIds);       }       public String[] getEnders (){           return (mIds);       }       public String[] getEndTagEnders (){           return (mEndTagEnders);       }         public String getLink(){        return super.getAttribute("href");       }         public String getMethod(){        return super.getAttribute("method");       }   }  } |

上面这段代码比较长，可以分成下面几部分来看：

1. getWmlContent方法： 该方法用来获取在同一个包中的test.wml脚本文件的内容并返回字符串。

2. 静态属性lnkFilter：这是一个NodeFilter的匿名类所构造的实例。该实例用来传递给HTMLParser告知需要提取哪些节点。在这个例子中我们仅需要提取链接标签以及我们自定义的一个GO标签。

3. 嵌套类WmlGoTag：这也是最为重要的一部分，这个类用来告诉HTMLParser如何去解析<go>这样一个节点。我们先看看下面这个HTMLParser的节点类层次图：

如上图所示，HTMLParser将一个文档分成三种节点分别是：Remark（注释）；Text（文本）；Tag（标签）。而标签又分成两种分别是简单标签（Tag）和复合标签（CompositeTag），像<img><br/>这种标签称为简单标签，因为标签不会再包含其它内容。而像<a href="xxxx">Home</a>这种类型的标签，因为标签会嵌套文本或者其他标签的称为复合标签，也就是对应着CompositeTag这个类。简单标签的实现类很简单，只需要扩展Tag类并覆盖getIds方法以返回标签的识别文本，例如<img>标签应该返回包含"img"字符串的数组，具体的代码可以参考HTMLParser自带的ImageTag标签类的实现。

从上图可清楚看出，复合标签事实上是对简单标签的扩展，HTMLParser在处理一个复合标签时需要知道该标签的起始标识以及结束标识，也就是我们在前面给出的源码中的两个方法getIds和getEnders，一般来讲，标签出现都是成对的，因此这两个方法一般返回相同的值。另外一个方法getEndTagEnders，这个方法用来返回父一级的标签名称，例如<tr>的父一级标签应该是<table>。这个方法的必要性在于HTML对格式的要求很不严格，在很多的HTML文档中的一些标签经常是有开始标识，但是没有结束标识，由于浏览器的超强适应能力使这种情况出现的很频繁，因此HTMLParser利用这个方法来辅助判断一个标签是否已经结束。由于WML文档的格式要求非常严格，因此上例源码中的getEndTagEnders方法事实上可有可无。

4. 入口方法main：该方法初始化HTMLParser并注册新的节点解析器，解析文档并打印运行结果。

最后我们编译并运行这个例子，便可以得到下面的运行结果：

|  |
| --- |
| GO:  /wap/user.do  LINK:  /wap/index.vm |

HTMLParser本身就是一个开放源码的项目，它对于HTML文档中出现的标签定义已经应有尽有，我们尽可以参考这些标签解析类的源码来学习如何实现一个标签的解析类，从而扩展出更丰富多彩的应用程序。