**C++ html——ParserDom.h的解析**

我们首先来分析ParserDom.h这个文件。上面的例子中我们就用到了这个头文件来对Html文档进行解析。这里我们需要说明，一般的XML格式文档(包括Html文档)解析有两种方式，即DOM方式和SAX方式，下面我们就来详细的了解下DOM方式。

**1首先我们来看一下ParserDom.h中ParserDom类的几个方法和数据成员的声明。.**

#ifndef \_\_HTML\_PARSER\_DOM\_H\_\_

#define \_\_HTML\_PARSER\_DOM\_H\_\_

#include "ParserSax.h"

#include "tree.h"

namespace htmlcxx

{

namespace HTML

{

class ParserDom : public ParserSax

{

public:

ParserDom() {}//构造方法

~ParserDom() {}//析构方法

const tree<Node> &parseTree(const std::string &html);//通过字符串来解析树

const tree<Node> &getTree() { return mHtmlTree; }//返回解析树

protected://声明虚函数，用于重载

virtual void beginParsing();//开始解析

virtual void foundTag(Node node, bool isEnd);//寻找指定标签

virtual void foundText(Node node);//寻找文本

virtual void foundComment(Node node);//寻找注释文本

virtual void endParsing();//结束解析

tree<Node> mHtmlTree;//存放解析好的树

tree<Node>::iterator mCurrentState;//用于遍历树的迭代器，具体声明可以见tree.h

};

//重写操作符<< ,通过该操作符可以直接输出tree<HTML::Node> 类型的变量

std::ostream &operator<<(std::ostream &stream, const tree<HTML::Node> &tr);

} //namespace HTML

} //namespace htmlcxx

#endif

**2. 在ParseDom类之后我们还可以见到有如下代码：**

ostream &HTML::operator<<(ostream &stream, const tree<HTML::Node> &tr) //该段代码重写了<<操作符，使得可以该操作符可以直接输出tree<HTML::Node>类型的变量。

**3. parseTree(const std::string &html)**

该方法主要是调用ParserSax.h中的parse()方法(并最终经过一系列调用，调用到ParserSax.tcc中的parse()方法)对输入的一段string字符串形式的html文档进行解析。

const tree<HTML::Node>& ParserDom::parseTree(const std::string &html)

{

this->parse(html);//调用ParserSax.h中的parse方法对html字符串进行解析，并且将结果存入数据成员mHtmlTree

return this->getTree();//返回受保护的数据成员mHtmlTree

}

**4. beginParsing()**

该方法主要是为解析之前做一些初始化的工作，主要是在树的根节点之前插入一个新的节点作为新根节点。

void ParserDom::beginParsing()

{

mHtmlTree.clear();

tree<HTML::Node>::iterator top = mHtmlTree.begin();

HTML::Node lambda\_node;

lambda\_node.offset(0);

lambda\_node.length(0);

lambda\_node.isTag(true);

lambda\_node.isComment(false);

mCurrentState = mHtmlTree.insert(top,lambda\_node);

}

**5. endParsing()**

void ParserDom::endParsing()

{

tree<HTML::Node>::iterator top = mHtmlTree.begin();

top->length(mCurrentOffset);//mCurrentOffset是ParserSax类中的一个protected成员变量，记录当前解析长度

}

/\*

该方法的主要作用是在当前正在解析的节点(即mCurrentState游标所指向的节点)下将节点作为注释内容插入进去。

实际这个方法上和后面的foundText()方法完全一样，因为无论是Comment或Text都是作为Node添加进去的，

对于程序来讲没有本质上的区别。

\*/

void ParserDom::foundComment(Node node)

{

//Add child content node, but do not update current state

//在当前节点下添加一个新的节点node，但是不更新当前解析的进度，即不修改mCurrentState游标

mHtmlTree.append\_child(mCurrentState, node);

}

/\*

同ParserDom::foundComment(Node node)

\*/

void ParserDom::foundText(Node node)

{

//Add child content node, but do not update current state

//在当前节点下添加一个新的节点node，但是不更新当前解析的进度，即不修改mCurrentState游标

mHtmlTree.append\_child(mCurrentState, node);

}

**6. foundTag(Nodenode, bool isEnd)**

该方法的主要作用是添加一个新的标签节点，并且需要根据isEnd变量进行判断是起始标签如<div>或<a>等，还是结束标签如</div>或</a>等**。**

void ParserDom::foundTag(Node node, bool isEnd)

{

if (!isEnd)

{

//append to current tree node

tree<HTML::Node>::iterator next\_state;

next\_state = mHtmlTree.append\_child(mCurrentState, node);

mCurrentState = next\_state;

}

else

{

//Look if there is a pending open tag with that same name upwards

//If mCurrentState tag isn't matching tag, maybe a some of its parents

// matches

vector< tree<HTML::Node>::iterator > path;

tree<HTML::Node>::iterator i = mCurrentState;

bool found\_open = false;

while (i != mHtmlTree.begin())

{

#ifdef DEBUG

cerr << "comparing " << node.tagName() << " with " << i->tagName()<<endl<<":";

if (!i->tagName().length()) cerr << "Tag with no name at" << i->offset()<<";"<<i->offset()+i->length();

#endif

assert(i->isTag());

assert(i->tagName().length());

bool equal;

const char \*open = i->tagName().c\_str();

const char \*close = node.tagName().c\_str();

equal = !(strcmp(open,close));

if (equal)

{

//DEBUGP("Found matching tag %s\n", i->tagName().c\_str());

//Closing tag closes this tag

//Set length to full range between the opening tag and

//closing tag

i->length(node.offset() + node.length() - i->offset());

i->closingText(node.text());

mCurrentState = mHtmlTree.parent(i);

found\_open = true;

break;

}

else

{

path.push\_back(i);

}

i = mHtmlTree.parent(i);

}

if (found\_open)

{

//If match was upper in the tree, so we need to invalidate child

//nodes that were waiting for a close

for (unsigned int j = 0; j < path.size(); ++j)

{

// path[j]->length(node.offset() - path[j]->offset());

mHtmlTree.flatten(path[j]);

}

}

else

{

//DEBUGP("Unmatched tag %s\n", node.text().c\_str());

// Treat as comment

node.isTag(false);

node.isComment(true);

mHtmlTree.append\_child(mCurrentState, node);

}

}

}

这段算法首先根据isEnd判断要添加的节点是否是结束tag节点，如果不是则直接将该节点插入到当前节点下。否则的话则需要向前寻找其对应的起始tag节点，一直回溯找到树的根节点，若没有找到则报错。此时还需要考虑如下可能，即该起始tag节点和所插入的结束tag节点之间还可能有其他的tag节点，如插入的是**</div>**，而可能会有<div><a>…</a><div>…</div>**</div>**，此时第一个<div>才是其所对应的起始tag节点。为了解决这个问题，本函数里采用了一个将树路径以vetor存储的方式进行查找匹配。

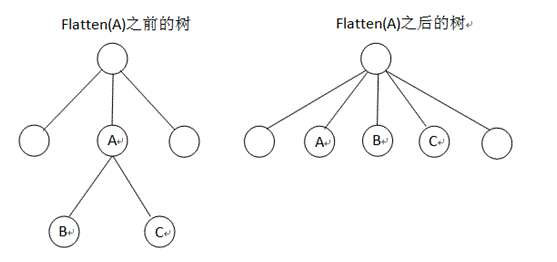
**1. 将所插入节点与当前节点的tagName进行比较。如果相同或者当前节点就是根节点，则前往第三步。**

**2. 将当前节点重置为当前节点的父节点，并将当前节点按顺序添加至path。回到第1步。**

**3. 检查是否找到tagName相同的节点，如果找到，我们需要调用flatten()方法调整树的结构，从而消除掉原本正在等待结束tag的起始tag节点。**

**4. 否则即为未找到，报错。**

为了方便大家理解，我们稍微介绍一下这里使用到的flatten()方法，这是一个很有意思的树操作方法，其效果如下图所示：



关于flatten()具体可以参看tree.h中的具体算法。在这个算法中，由于建树时的算法特性，需要在找到匹配的起始标签后，对path路径上的所有节点执行flatten算法，这样就可以将原本需要等待结束tag标签的起始标签放到合适的位置。具体原因可以去查看ParseSax.tcc中构建树节点的算法来理解。

**7. operator<<(ostream &stream, consttree<HTML::Node> &tr)**

本方法主要重写了<<操作符，使其可以直接输出tree<HTML::Node>类型的对象。

ostream &HTML::operator<<(ostream &stream, const tree<HTML::Node> &tr) //该段代码重写了<<操作符，使得可以该操作符可以直接输出tree<HTML::Node>类型的变量。

{

tree<HTML::Node>::pre\_order\_iterator it = tr.begin();

tree<HTML::Node>::pre\_order\_iterator end = tr.end();

int rootdepth = tr.depth(it);

stream << "-----" << endl;

unsigned int n = 0;

while ( it != end )

{

int cur\_depth = tr.depth(it);

for(int i=0; i < cur\_depth - rootdepth; ++i)

{

stream << " ";

}

stream << n << "@";

stream << "[" << it->offset() << "~";

stream << it->offset() + it->length() << "] ";

stream << (string)(\*it) << endl;

++it, ++n;

}

stream << "-----" << endl;

return stream;

}

利用了前序的游标pre\_order\_iterator循环遍历整棵树并且进行一些格式化的输出。关于pre\_order\_iterator的定义可以参考tree.h部分代码。