

[首页](#) [资讯](#) [精华](#) [论坛](#) [问答](#) [博客](#) [专栏](#) [群组](#) [更多 ▼](#)

[您还未登录!](#) [登录](#) [注册](#)

## yuanlanxiaup

- [博客](#)
- [微博](#)
- [相册](#)
- [收藏](#)
- [留言](#)
- [关于我](#)



[C++ Html解析器-HtmlCxx用户手册和源代码解析](#)

# HtmlCxx用户手册

中科院计算所网络数据科学与工程研究中心

信息抽取小组

[gengyun@sohu.com](mailto:gengyun@sohu.com)

## 1.1 简介

HtmlCxx是一款简洁的，非验证式的，用C++编写的css1和html解析器。和其他的几款Html解析器相比，它具有以下的几个特点：

使用由KasperPeeters编写的强大的tree.h库文件，可以实现类似STL的DOM树遍历和导航。

可以通过解析后生成的树，逐字节地重新生成原始文档。

打包好的Css解析器。

额外的属性解析功能

看似很像C++代码的C++代码(其实已不再是C++了)

原始文档中的tags/elements的偏移值都存储在DOM树的节点当中。

Htmlcxx的解析策略其实是尝试模仿mozilla firefox(<http://www.mozilla.org>)的模式。因此你应当尝试去解析那些由firefox所生成的文档。然而不同于firefox浏览器，htmlcxx并不会将一些原本不存在的东西加入到所生成的文档当中去。因此，在将生成树进行序列化的时候，能够完全地还原和原始Byte大小一样的HTML文档。

## 1.2 快速上手

下面我们通过一个简单的小例子来让大家对如何使用HtmlCxx进行开发有一个快速，直观地了解。

### 1. 安装环境

操作系统: Ubuntu 10.10 (32-bit Linux)

编译环境: GCC 4.4.4

HtmlCxx版本: 0.85

## 2. 源代码下载

各个版本的HtmlCxx可以从著名的开源代码网站SourceForge上下载。由于HtmlCxx已经于2005年停止更新，我们采用的是其最后更新的版本0.85。

除此之外，使用Ubuntu的用户还可以直接在命令行下使用以下命令很方便地进行安装：

```
sudo apt-get install htmlcxx
```

安装好后，可以到usr/include/htmlcxx目录下进行查看，其中包含了一些可以使用的功能的.h文件。

## 3. 编写小例子

在随意一个文件夹下新建一个文件，如Temp.cc，之后打开该文件，并输入如下代码：

```
#include <string>

#include <iostream>

#include <sstream>

#include <htmlcxx/html/ParserDom.h>

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

using namespace std;

using namespace htmlcxx;

int main()

{

    //解析一段Html代码

    string html = "<html><body>hey</body></html>";

    HTML::ParserDom parser;

    tree<HTML::Node> dom = parser.parseTree(html);

    //输出整棵DOM树

    cout<< dom << endl;

    //输出树中所有的超链接节点

    tree<HTML::Node>::iterator it = dom.begin();

    tree<HTML::Node>::iterator end = dom.end();

    for(; it != end; ++it)

    {

        if(strcasecmp(it->tagName().c_str(), "A") == 0)

        {
```

```
it->parseAttributes();

cout << it->attribute("href").second << endl;

}

}

//输出所有的文本节点

it= dom.begin();

end= dom.end();

for(; it != end; ++it)

{

if((!it->isTag()) && (!it->isComment()))

{

cout << it->text();

}

}

cout << endl;

}
```

将代码编写好保存后，打开控制台进入Temp.cc的目录下，输入以下的命令使用gcc将其编译：

```
gcc -o Temp Temp.cc -Iusr/include/htmlcxx-lhtmlcxx
```

请注意上面这段代码中的第一个-I是大写的i，第二个-I是小写的L。

编译生成可执行文件Temp，直接输入./Temp运行，程序如果正常运行则应当出现如下的结果：



至此我们的htmlcxx已经可以成功地使用和运行了！

## 二.源代码分析

### 2.1 简介

代码分为html和css两个部分，我们主要分析html部分。比较复杂的是tree.h文件，里面不但包含了对树的结构定义，而且包含了许多遍历算法。该文件的说明可以在<http://tree.phy-sci.com/documentation.html>上找到。在学习HtmlCxx的源代码的时候，要注意到的一点就是不仅应学习它的原理，更因为它引用了开源的树结构代码tree.h，优秀的编码风格贯穿始终，代码十分工整和简洁，广泛适用C++的template机制使其可扩展性非常强，为我们学习C++ 语言编程提供了良好的示范教材。

由于其注释代码内部较少，下面这段源代码解析主要为方便大家理解。

### 2.2 ParserDom.h(cc)

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，下面简要介绍一下这两种方式的异同。

### DOM方式：

DOM(DocumentObject Model 文档对象模型)方式，从名字上来看就是将XML格式的文档读至内存，并为每个节点建立一个对象，之后我们可以通过操作对象的方式来对节点进行操作，即是把XML的节点映射到了内存中的一系列对象之上。

该方法的优点是在建立好模型后的查询和修改速度较快。

该方法的缺点是在处理之前需要将整个XML文档全部读入内存进行解析，并根据XML树生成一个对象模型，当文档很大时，DOM方式就会突现出其庞大的特性，一个300KB的XML文档可以导致RAM或者虚拟内存中3000000KB的DOM树模型。

### SAX方式：

SAX(Simple API for XML)是一种基于事件(Event)的XML处理模式，该模式和DOM相反，不需要将整个XML树读入内存后再进行处理，而是将XML文档作为一个输入“流”来处理，在处理该流的时候会触发很多事件，如SAX解析器在处理文档第一个字符的时候，发现文档的开始，就会触发Start document事件，用户可以重载这个事件函数来对该事件进行相应处理。又如当遇到新的节点时，则会触发Start element事件，此时我们可以对该节点名进行一些判断，并作出相应的处理等。

该方法的优点是不用等待所有数据都被处理后再开始分析，且不需要将整个文档都装入内存中，这对大文档来说是个巨大的优点。一般来说SAX要比DOM方式快很多。

该方法的缺点是由于在处理过程中没有储存任何数据，因此在处理过程中不能够对数据流进行后移，或者回溯等操作。

在熟悉了这两种方式之后，我们可以看到在ParserDom.h文件中include了ParserSax.h文件，这是由于在HtmlCxx中，DOM方式是基于SAX方式之上的，关于这点我们之后再说。

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。

```
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String字符串来解析树
    const tree<Node> &getTree() //返回(解析好的)数据成员mHtmlTree

    {return mHtmlTree; }

protected:
    //声明了一系列虚函数，用于之后的重载

    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; //数据成员，用于存放解析好的树

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

};
```

在ParseDom类之后我们还可以见到有如下代码：

```
std::ostream& operator<<(std::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。

### 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
}
```

### 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(); //首先清空mHtmlTree

    tree<HTML::Node>::iterator top = mHtmlTree.begin(); //之后申请一个top游标，指向mHtmlTree树的开始节点

    HTML::Node lambda_node; //申请一个新节点

    lambda_node.offset(0); //初始化moffset值为0，具体请查阅node.h和node.cc文件

    lambda_node.length(0);

    lambda_node.isTag(true);

    lambda_node.isComment(false);

    mCurrentState = mHtmlTree.insert(top, lambda_node); //将这个节点插入到树中去成为根节点，注意是插入到top游标节点的之前，即previous sibling，具体请查阅tree.h中的insert方法。

}
```

### endParsing()

该方法主要作用是停止当前的解析，并记录下已经解析的长度。

```
void ParserDom::endParsing()
{
    tree<HTML::Node>::iteratortop = mHtmlTree.begin(); //获取根节点
    top->length(mCurrentOffset); //将根节点的length数据成员设置为当年已解析的距离
}
```

### foundComment()

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)
{
    //Addchild 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游标的位置
    mHtmlTree.append_child(mCurrentState,node);
}
```

### foundText()

该方法的主要作用和前一个方法雷同，从函数名的字面解释来讲应当是添加文本。

```
void ParserDom::foundComment(Node node)
{
    //Addchild 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游标的位置
    mHtmlTree.append_child(mCurrentState,node);
}
```

### foundTag(Node node, 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)
```

```
{

//appendto current tree node

tree<HTML::Node>::iteratormnext_state;

next_state= mHtmlTree.append_child(mCurrentState, node);

mCurrentState= next_state;

}

else

{

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

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

//matches

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

tree<HTML::Node>::iteratori = mCurrentState;

boolfound_open = false;

while(i != mHtmlTree.begin())

{

#ifdefDEBUG

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());

boolequal;

constchar *open = i->tagName().c_str();

constchar *close = node.tagName().c_str();

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

if(equal)

{

DEBUGP("Foundmatching tag %s\n", i->tagName().c_str());

//Closingtag closes this tag

//Setlength to full range between the opening tag and

//closingtag

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)

{

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

//nodesthat 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("Unmatchedtag %s\n", node.text().c_str());

//Treat as comment

node.isTag(false);

node.isComment(true);

mHtmlTree.append_child(mCurrentState,node);

}

}

}
```



这段算法的思路很简单，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vector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。

本方法主要是提供给ParseSax.tcc中的解析方法所调用。

**operator<<(ostream &stream, const tree<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)
```

```
{
    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;

}

```

代码很简单，就是利用了前序的游标`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。

## 2.3 ParserSax.h(.cc)

前面我们介绍过了Sax方式和DOM方式的区别，实际上在HtmlCxx中，这两种方式是结合使用的，即首先用SAX方式对流进行解析，解析出的结果采用DOM方式保存到一个树型的`tree<Node>`数据结构中。

首先我们还是来看一下ParserSax.h头文件中对于数据成员和函数的声明：

```

class ParserSax
{
public:
    ParserSax(): mpLiteral(0), mCdata(false) {} //构造方法，初始化两个成员变量的值
    virtual ~ParserSax() {} //析构方法，空

    /**Parse the html code */
    void parse(const std::string &html); //解析一个string格式的html文档

    template<typename _Iterator>
    void parse(_Iterator begin, _Iterator end); //使用两个类型相同的形参格式的解析器模板

protected:
    //Redefine this if you want to do some initialization before
    //the parsing

    virtual void beginParsing() {} //空，可进行扩展

    //以下几个虚函数，都在ParseDom.cc中进行了实现
    virtual void foundTag(Node node, bool isEnd) {}

    virtual void foundText(Node node) {}

    virtual void foundComment(Node node) {}

    virtual void endParsing() {}

    //以下是几个模板，大部分都在ParseSax.tcc文件中实现
    template<typename _Iterator>
    void parse(_Iterator &begin, _Iterator &end,
        std::forward_iterator_tag);

    template<typename _Iterator>

```

```

voidparseHtmlTag(_Iterator b, _Iterator c);

template<typename _Iterator>

voidparseContent(_Iterator b, _Iterator c);

template<typename _Iterator>

voidparseComment(_Iterator b, _Iterator c);

template<typename _Iterator>

_IteratorskipHtmlTag(_Iterator ptr, _Iterator end);

template<typename _Iterator>

_IteratorskipHtmlComment(_Iterator ptr, _Iterator end);

//几个数据成员

unsignedlong mCurrentOffset; //当前正在解析的偏移位置

constchar *mpLiteral; //具体作用需要到ParseSax.tcc中才显现出来

boolmCdata; //同上

};

```

打开ParserSax.cc可以看到，里面空荡荡地只有一个Parse(const std::string &html)方法

```

void htmlcxx::HTML::ParserSax::parse(conststd::string &html)

{

// std::cerr<< "Parsing string" << std::endl;

parse(html.c_str(),html.c_str() + html.length());

}

```

该方法实际上直接调用了另一个template<typename \_Iterator> void parse(\_Iterator begin, \_Iterator end)方法，而这个方法和一些其他主要方法都是在ParseSax.tcc中进行实现。

## 2.4 ParseSax.tcc

注意.tcc文件和一般的.cc文件有所不同，应当是基于turbo c++的，因为turbo c++是很多年没用过的东西了，具体我也不是很懂有什么区别。

ParseSax.tcc这个文件中主要实现了对html文档的文本流的解析，并将其转化为一个个的节点，并存储到Html树中去。其实现了一整套的Html解析流程，值得我们学习。该流程主要针对的是FireFox生成的Html格式来进行的解析。并且使用parseContent(), parseTag(), parseComment()对一段内容作为内容，标签，注释等进行不同的处理。

如果读者曾经写过类似的Html解析器，那么在看这段代码的时候会非常清晰，即使没有写过也没关系，因为这段代码的逻辑很清晰，可以借鉴其作为对Html解析器的标准来进行学习。

其基本思路就是，按序遍历整个字符串，遇到'<'则认为其是tag标签的起始处，在tag中遇到'/'则认为该标签是结束型tag标签，遇到'>'则认为其是tag标签的结束处，遇到'!'则认为其是注释字段的起始或结束处。之后运行相应的parseContent(),parseTag(),parseComment()进行解析即可。

**parse(\_Iterator &begin, \_Iterator&end, std::forward\_iterator\_tag)**

由于这段代码较长，我们将其分解后进行讲解。

```
voidhtmlcxx::HTML::ParserSax::parse(_Iterator &begin, _Iterator &end,std::forward_iterator_tag)
{
typedef _Iterator iterator;

// std::cerr<< "Parsing forward_iterator" << std::endl;

mCdata= false;

mpLiteral= 0;

mCurrentOffset= 0;

this->beginParsing(); //调用beginParsing()进行初始化

// DEBUGP("Parsedtext\n");
```

上面这段代码进行了一些成员变量的初始化，并且调用了beginParsing()进行相应的初始化。

```
while(begin != end)
{
*begin;// This is for the multi_pass to release the buffer
```

这个while循环是最外层的循环，其循环的依据是begin和end两个游标，这两个游标始终指向了待解析的文本串的起始端和结束端。

```
while(c != end)
{
//For some tags, the text inside it is considered literal and is
//only closed for its </TAG> counterpart
```

上面这个while循环是次外层的循环，该循环每次结束都即为解析出一个节点。

```
while(mpLiteral)
{
// DEBUGP("Treatingliteral %s\n", mpLiteral);
```

在这个循环的判断条件中，mpLiteral默认初始值为0，在运行后面的ParseHtmlTag()方法时候会更改它的值，如果不为零，则说明当前的tagName应当为以下这个数组的str属性其中之一。

```
static

struct literal_tag {
```

```

intlen;

constchar* str;

intis_cdata;

}

literal_mode_elem[] =

{

{6,"script", 1},

{5,"style", 1},

{3,"xmp", 1},

{9,"plaintext", 1},

{8,"textarea", 0},

{0,0, 0}

};

```

其意义就是说，当遇到这些特殊节点的时候，需要进入该循环进行一些特别的处理，即应当着手向后寻找该tagName对应的</tagName>标签，其间遇到的所有内容都应当作为普通文本进行处理。

我们继续分析代码：

```

// DEBUGP("Treatingliteral %s\n", mpLiteral);

while(c != end && *c != '<') ++c;//向后不断查找，直到找到了文件末尾或'<'才停止，此刻说明解析完毕或者解析到了某个tag标签

if(c == end) { //如果成立说明解析完毕

if(c != begin) this->parseContent(begin, c); //此时最后一个解析的标签末尾和文档末尾之间可能还有一段内容需要解析

gotoDONE; //转到结束阶段

}

iteratorend_text(c); //申请一个新游标end_text，初始值赋入c

++c; //将c游标前进一步

```

此处申请的end\_text，主要是为了在后面的代码中记录下当前'<'符号的位置。

```

if(*c == '/') //判断'<'后面紧跟的是否为'/'符号

{

++c; //前进一步，指向tagName的起始位置

constchar *l= mpLiteral;

while(*l && ::tolower(*c) == *l) //逐字节将mpLiteral和tagName进行比较

```

```
{
++c;
++l;
}
```

跳出最后一个while循环时，c和l应当指向的是mpLiteral和tagName从起始位置开始的最小公共子串的结束位置，如果匹配成功则l应当指向字符串的最后一个位置+1，即为空。此处需要注意的是，Mozilla浏览器一旦遇到/plaintext就会停止解析，因此这里需要对此情况作出判断。不过作者此处貌似没有写得很完善，还期待将来做出修改。

```
//FIXME: Mozilla stops when it sees a /plaintext. Check
//other browsers and decide what to do

if(*l && strcmp(mpLiteral, "plaintext"))
{
//matched all and is not tag plaintext
//说明不是plaintext标签，且此时指针为空
while(isspace(*c)) ++c; //跳过空格字符
if(*c == '>') //直到找到标签结束位置
{
++c;

if(begin != end_text)
this->parseContent(begin,end_text); //处理这段标签之前的那段“文本”
mpLiteral= 0; //更改循环进入条件
c= end_text; //修改游标值
begin= c; //修改游标值
break; //跳出循环，此时说明已经找到了对应tagName的</tagName>标签
}
}
```

除此之外，后面的一段代码还考虑到了在寻找过程中遇到注释的情况，也需要做出相应处理。

```
elseif(*c == '!')
{
// we may find a comment and we should support it
iteratore(c);
++e;
```

```

if(e != end && *e == '-' && ++e != end && *e == '-')
{
// DEBUGP("Parsingcomment\n");

++e;

c= this->skipHtmlComment(e, end);

}

}

}

```

上面这段代码很简单，就是对遇到注释后的进行处理，直接调用skipHtmlComment()方法就可以很方便地进行处理了，关于skipHtmlComment()方法的详细介绍我们会在后面给出。

到这里为止while(mpLiteral)循环的代码就结束了，下面的代码则是解析一般tag标签的代码。

```

if(*c == '<') //判断是否是一个标签的开始
{
    iterator d(c); //申请一个新游标d，初始为这个'<'的位置
    ++d; //将其后移一位
    if(d != end) //如果不是文档的结尾
    {

```

此时d所指向的应当是'<'之后的那个字符，现在要通过对这个字符进行判断来进行不同的处理，一共分四种情况进行处理：普通字符，'/'符号，'!'符号，'?'或'%'符号。分别对应tagName，结束tag标签，注释字段，以及一些特殊字段如<?xml 或 <%VBSCRIPT。

首先是解析普通起始型标签的：

```

if(isalpha(*d)) //如果d指向的是普通字符
{
//beginning of tag 说明是tag标签的起始

if(begin != c) //如果标签之前还有一段文本
this->parseContent(begin,c); //则对其进行处理

// DEBUGP("Parsingbeginning of tag\n");

d= this->skipHtmlTag(d, end); //找到Tag标签的结束位置

this->parseHtmlTag(c,d); //对Tag标签的起始位置和结束位置之间的这段标签内容适用parseHtmltag(X,X)进行处理。

//continue from the end of the tag

c= d; //将游标移动到标签之后的位置

begin= c;

```

```
break; //成功解析出一个节点，跳出循环
```

```
}
```

其次是解析普通结束型标签的：

```
if(*d == '/')
```

```
{
```

```
if(begin != c) //如果标签之前还有一段文本
```

```
this->parseContent(begin,c); //则对其进行处理
```

```
iteratore(d); //申请一个新标签e，初始化为d
```

```
++e; //将其后移一位
```

```
if(e != end && isalpha(*e))
```

```
{
```

```
//说明此时e指向的是结尾型标签的起始处
```

```
d= this->skipHtmlTag(d, end); //寻找标签结束位置
```

```
this->parseHtmlTag(c,d); //对这段位置之间的标签进行处理
```

```
}
```

```
else
```

```
{
```

```
//不是一个标准的结束型标签，和Mozilla采用一样的处理方式
```

```
d= this->skipHtmlTag(d, end);
```

```
this->parseComment(c,d);
```

```
}
```

```
//将游标移动到tag标签之后继续
```

```
c= d;
```

```
begin= c;
```

```
break; //解析出一个节点，跳出循环
```

```
}
```

之后是处理注释型标签的，即“<!--”型标签：

```
if(*d == '!')
```

```
{
```



```
//说明是注释标签

if(begin != c) //如果此标签之前还有一段文本
this->parseContent(begin,c); //则对其进行处理
iteratore(d); //申请一个游标e，初始化为d
++e; //将e后移一位

if(e != end && *e == '-' && ++e != end && *e == '-')
{
// DEBUGP("Parsingcomment\n");

++e;

d = this->skipHtmlComment(e, end); //查找到标签结尾
}

else
{
d = this->skipHtmlTag(d, end); //也可能注释直到文档结束
}

this->parseComment(c,d); //将这段文本作为注释进行处理

//移动游标至标签结尾处

c = d;

begin = c;

break; //成功解析出一个标签，跳出循环
}
```

最后这段代码是处理一些诸如<?xml 或 <%VBSCRIPT之类的特殊标签的：

```
if(*d == '?' || *d == '%')
{
//这段代码很简单

if(begin != c) //如果标签之前还有一段文本
this->parseContent(begin,c); //则对其进行处理

d = this->skipHtmlTag(d, end); //找到标签末尾

this->parseComment(c,d); //对这段标签进行处理

//移动游标至末尾

c = d;

begin = c;
```

```
break;//成功解析出标签，跳出循环

}
```

之后还有一点结束代码，以及将游标c移位的代码。

```
}

}

}

c++; //每次while循环后都需要后移游标c

}

//在所有标签都处理完后，可能在文档末尾还有一些文本要进行处理

if(begin != c)

{

this->parseContent(begin,c); //处理这段文本

begin= c;

}

}

DONE: //作为GOTO代码的位置，即为解析结束后执行的收尾工作

this->endParsing();

return;

}
```

至此，Sax方式的html流解析主体代码基本就分析完毕了。下面我们来分析一下上面代码中分别用到的对注释，标签，内容进行解析的几个方法parseComment，parseHtmlTag，parseContent等，这些方法的原理和实现方法都很简单，读者可一目了然。

### parseComment(\_Iteratorb, \_Iterator c)

这段代码的形参为两个游标，他们分别指示了要处理的文本的起始位置和结束位置，并将这段文本作为注释来进行处理。

```
template <typename _Iterator>

voidhtmlcxx::HTML::ParserSax::parseComment(_Iterator b, _Iterator c)

{

// DEBUGP("Creatingcomment node %s\n", std::string(b, c).c_str());

//申请一个新的Node节点

htmlcxx::HTML::Nodecom_node;

//FIXME:set_tagname shouldn't be needed, but first I must check
```

```
//legacycode
```

```
//申请一个string类型，用游标b,c之间的这段文本为其初始化赋值
```

```
std::stringcomment(b, c);
```

```
//以下是利用各种变量为代表该Node的一些属性的数据成员进行复制
```

```
//具体方法的细节我们在后面分析Node.h(cc)的时候再进行解释
```

```
com_node.tagName(comment); //tag名
```

```
com_node.text(comment); //文本
```

```
com_node.offset(mCurrentOffset); //在文档中的偏移值
```

```
com_node.length((unsignedint)comment.length()); //节点长度
```

```
com_node.isTag(false); //是否是标签
```

```
com_node.isComment(true); //是否是注释
```

```
mCurrentOffset+= com_node.length(); //将当前解析位置后移，移动到该节点的结束位置
```

```
//Call callback method
```

```
//回调方法，将该节点插入已有的Html树中
```

```
this->foundComment(com_node);
```

```
}
```

### **parseContent(\_Iteratorb, \_Iterator c)**

这段代码的形参为两个游标，他们分别指示了要处理的文本的起始位置和结束位置，并将这段文本作为文本内容来进行处理。

```
template <typename _Iterator>
```

```
voidhtmlcxx::HTML::ParserSax::parseContent(_ Iterator b, _Iterator c)
```

```
{
```

```
// DEBUGP("Creatingtext node %s\n", (std::string(b, c)).c_str());
```

```
//申请一个新的Node节点
```

```
htmlcxx::HTML::Nodetxt_node;
```

```
//FIXME:set_tagname shouldn't be needed, but first I must check
```

```
//legacycode
```

```
//申请一个string类型，用游标b,c之间的这段文本为其初始化赋值
```

```
std::stringtext(b, c);
```

```
//以下是利用各种变量为代表该Node的一些属性的数据成员进行复制
```

```
//具体方法的细节我们在后面分析Node.h(cc)的时候再进行解释
```

```

txt_node.tagName(text); //tag名

txt_node.text(text); //文本

txt_node.offset(mCurrentOffset); //在文档中的偏移值

txt_node.length((unsignedint)text.length()); //节点长度

txt_node.isTag(false); //是否是标签

txt_node.isComment(false); //是否是注释

mCurrentOffset+= txt_node.length(); //将当前解析位置后移，移动到该节点的结束位置

//Call callback method

//回调方法，将该节点插入已有的Html树中

this->foundText(txt_node);

}

```

### parseHtmlTag(\_Iteratorb, \_Iterator c)

这段代码的形参为两个游标，他们分别指示了要处理的文本的起始位置和结束位置，并将这段文本作为Html标签来进行处理。

我们分三段来看这段代码，首先看第一段：

```

template <typename _Iterator>

voidhtmlcxx::HTML::ParserSax::parseHtmlTag(_Iterator b, _Iterator c)

{

    _Iteratorname_begin(b); //申请一个新的游标name_begin游标，初始为b

    ++name_begin; //将name_begin后移一位

    boolis_end_tag = (*name_begin == '/'); //判断name_begin是否是'/'，如果是则说明是结束型tag标签

    if(is_end_tag) ++name_begin; //如果是结束型的tag标签，则再将name_begin前进一位，跳过'/'符号

    _Iteratorname_end(name_begin); //申请一个新的name_end游标，记录一下name_begin标签。

    while(name_end != c && isalnum(*name_end)) //将name_end后移，直到遇到了文档末尾或者普通字符为止。

    {

        ++name_end;

    }

    std::stringname(name_begin, name_end); //取name_begin和name_end之间的这段文本认为是tagName并赋值给一个string型的变量name

```

之后，我们要对这个Name做一些判断，判断其是否是特殊类型的tag标签，即是否是前面literal\_mode\_clem[]数组中的某个标签。

```

if(!is_end_tag) //当然首先得判断这不是一个结束型的标签

{

```

```
std::string::size_type tag_len = name.length();

for(int i = 0; literal_mode_elem[i].len; ++i) //循环进行查找

{

    if(tag_len == literal_mode_elem[i].len) //进行比较

    {

        #ifdef(WIN32) && !defined(__MINGW32__) //这段就是根据系统采用不同的string比较函数

        if(!_stricmp(name.c_str(), literal_mode_elem[i].str))

        #else

        if(!strcasecmp(name.c_str(), literal_mode_elem[i].str))

        #endif

        {

            mpLiteral= literal_mode_elem[i].str; //如果找到，则将其赋值为mpLiteral

            break; //并退出循环

        }

    }

}
```

之后，我们就可以将其作为Node节点来做一些存储的工作了。

```
htmlcxx::HTML::Nodetag_node;

//bynow, length is just the size of the tag

//注意到，目前这还只是起始节点，因此长度只是<tag>本身的长度

std::stringtext(b, c);

tag_node.length(static_cast<unsignedint>(text.length())); //节点长度

tag_node.tagName(name); //tag名

tag_node.text(text); //节点文本

tag_node.offset(mCurrentOffset); //在文档中的偏移位置

tag_node.isTag(true); //是否是tag标签

tag_node.isComment(false); //是否是注释

mCurrentOffset+= tag_node.length(); //将当前的解析位置后移

this->foundTag(tag_node,is_end_tag); //调用方法将其加入到已有的Html树中

}
```

这个文件中的最后还有几个小方法，作用和思路都很简单，主要都是为上面几个方法进行服务的，我们简单地作一下介绍。

### **find\_next\_quote(\_Iteratorc, \_Iterator end, char quote)**

本方法主要是从游标c处开始，直到游标end之间，寻找到第一个quote字符的位置并返回其游标，未找到则返回end的位置。

```
template <typename _Iterator>
static inline
_Iterator find_next_quote(_Iterator c, _Iterator end, char quote)
{
    // std::cerr<< "generic find" << std::endl;

    while(c != end && *c != quote) ++c;

    return c;
}
```

### **\*find\_next\_quote(constchar \*c, const char \*end, char quote)**

本方法主要是从字符指针\*c开始，直到字符指针\*end之间，寻找到第一个quote字符的位置并返回指向其的字符指针，未找到则返回\*end指针。

```
template <>
inline
const char *find_next_quote(const char *c, const char *end, char quote)
{
    // std::cerr<< "fast find" << std::endl;

    constchar *d = reinterpret_cast<const char*>(memchr(c, quote, end - c));

    if(d) return d;

    else return end;
}
```

### **skipHtmlTag(\_Iteratorc, \_Iterator end)**

从游标c的位置开始，直到游标end的位置之间，寻找到该标签所对应的结束'>'符号的位置，并以游标形式返回。其中要注意对tag标签内的属性做了另外的处理，即能够识别并跳过属性中的'>'符号，防止出错。

```
template <typename _Iterator>
_Iteratorhtmlcxx::HTML::ParserSax::skipHtmlTag(_Iterator c, _Iterator end)
{
    while(c != end && *c != '>')
```

```
{  
  
if(*c != '=')  
  
{  
  
++c;  
  
}  
  
else  
  
{// found an attribute  
  
++c;  
  
while(c != end && isspace(*c)) ++c;  
  
if(c == end) break;  
  
if(*c == '"' || *c == '\')  
  
{  
  
_Iteratorsave(c);  
  
charquote = *c++;  
  
c= find_next_quote(c, end, quote);  
  
// while(c != end && *c != quote) ++c;  
  
// c= static_cast<char*>(memchr(c, quote, end - c));  
  
if(c != end)  
  
{  
  
++c;  
  
}  
  
else  
  
{  
  
c= save;  
  
++c;  
  
}  
  
// DEBUGP("Quotes:%s\n", std::string(save, c).c_str());  
  
}  
  
}  
  
}  
  
if(c != end) ++c;  
  
returnc;  
  
}
```

## 2.5 Node.h(.cc)

Node.h里定义了树的每个节点的数据类型，重载了一些运算符使其方便进行一些逻辑运算，并提供了根据节点内文本分析和存储节点属性的parseAttributes()方法。我们在使用HtmlCxx进行html解析的时候，经常会对Node数据类型进行操作以及用Node进行输出，因此理解其的数据结构十分重要。

首先我们来看Node.h头文件的定义，其数据成员全部声明为protected类型：

protected:

std::stringmText; //节点的内部文本

std::stringmClosingText;

unsignedint mOffset; //节点在原文档中的偏移值

unsignedint mLength; //节点的字符长度的

std::stringmTagName; //节点的Tag名

std::map<std::string,std::string> mAttributes; //存放属性的二维数组

boolmIsHtmlTag; //节点是否是HtmlTag标签

boolmComment; //节点是否是注释

再看它的成员函数，全部为public。

public:

Node(){} //构造函数

//Node(constNode &rhs); //uses default

~Node(){} //析构函数

inlinevoid text(const std::string& text) { this->mText = text; }

inlineconst std::string& text() const { return this->mText; }

inlinevoid closingText(const std::string &text) { this->mClosingText = text; }

inlineconst std::string& closingText() const { return mClosingText; }

inlinevoid offset(unsigned int offset) { this->mOffset = offset; }

inlineunsigned int offset() const { return this->mOffset; }

inlinevoid length(unsigned int length) { this->mLength = length; }

inlineunsigned int length() const { return this->mLength; }

inlinevoid tagName(const std::string& tagname) { this->mTagName = tagname; }

inlineconst std::string& tagName() const { return this->mTagName; }

boolisTag() const { return this->mIsHtmlTag; }



```
void isTag(bool is_html_tag){ this->mIsHtmlTag = is_html_tag; }

bool isComment() const { return this->mComment; }

void isComment(bool comment){ this->mComment = comment; }
```

以上所有函数都是对各个数据成员的简单的赋值取值函数，十分简单就不加以解释了。

主要了解一下下面这个函数

```
std::pair<bool, std::string> attribute(const std::string &attr) const
{
    std::map<std::string, std::string>::const_iterator i = this->mAttributes.find(attr);
    if(i != this->mAttributes.end()) {
        return make_pair(true, i->second);
    } else {
        return make_pair(false, std::string());
    }
}
```

该函数主要功能是根据参数string &attr，查找到mAttributes[]中的某个项，并且返回该项的第二个成员值，即根据属性的名称查找该属性的值。返回的形式为一个make\_pair数据类型，如果不为最后一个属性，则为make\_pair(true, 属性值)，而如果是最后一个属性则返回一个make\_pair(false, 空字符串)。

之后还有一些关于操作符重载的声明：

```
operator std::string() const;

std::ostream& operator<<(std::ostream &stream) const;

const std::map<std::string, std::string> & attributes() const { return this->mAttributes; }

void parseAttributes();

bool operator==(const Node &rhs) const;
```

下面我们来看Node.cc文件中的内容。

### parseAttributes()

由于这段代码较长，因此我们也分段对其进行解析。这段代码默认在mText中已经存放了节点的文本数据。并且通过对该文本数据进行解析，将所有诸如<.....ID = “abc”...>之类的属性信息识别并储存起来。

```
if(!(this->isTag())) return; //判断是否是Tag标签节点，不是则返回

constchar *end;

constchar *ptr = mText.c_str(); //ptr作为mText的*char类型指针

if(ptr = strchr(ptr, '<')) == 0) return; //将ptr指向字符串中首先出现的<位置，并判断该位置是否合法，如果不合法则返
```

回。

```
++ptr; //将ptr后移一个字节
```

```
//Skip initial blankspace
```

```
while(isspace(*ptr)) ++ptr; //跳过接下来所有遇到的空格字符
```

```
//Skip tagname
```

```
if(!isalpha(*ptr)) return; //判断是否是普通字符，不是则返回
```

```
while(!isspace(*ptr)) ++ptr; //不断后移ptr指针，直到遇到空格字符
```

```
//Skip blankspace after tagname
```

```
while(isspace(*ptr)) ++ptr; //跳过接下来所有遇到的空格字符
```

下面进入一个循环，该循环将一直不断向后解析该tag标签，直到遇到了’>’字符或者指针为空才退出。

```
while(*ptr && *ptr != '>')
```

```
{
```

```
stringkey, val;
```

```
//跳过所有无法识别的字节
```

```
while(*ptr && !isalnum(*ptr) && !isspace(*ptr)) ++ptr;
```

```
//跳过所有的空格字符
```

```
while(isspace(*ptr)) ++ptr;
```

```
end= ptr;
```

```
//将end指向ptr之后第一个不为普通字符或’-’字符的位置
```

```
while(isalnum(*end) || *end == '-') ++end;
```

```
//将ptr至end之间的这段文本变为小写字符赋值给key
```

```
key.assign(end- ptr, '\0');
```

```
transform(ptr,end, key.begin(), ::tolower);
```

```
ptr= end;
```

```
//此时key中应当存放的是属性名
```

```
//跳过所有的空格字符
```

```
while(isspace(*ptr)) ++ptr;
```

下面是解析并获取属性的值

```
while(*ptr && *ptr != '>')
```

```
{
```

```
stringkey, val;

//跳过所有无法识别的字节

while(*ptr && !isalnum(*ptr) && !isspace(*ptr)) ++ptr;

//跳过所有的空格字符

while(isspace(*ptr)) ++ptr;

end= ptr;

//将end指向ptr之后第一个不为普通字符或'-'字符的位置

while(isalnum(*end) || *end == '-') ++end;

//将ptr至end之间的这段文本变为小写字符赋值给key

key.assign(end- ptr, '\0'); //注意要加上'\0'

transform(ptr,end, key.begin(), ::tolower);

ptr= end;

//此时key中应当存放的是属性名

//跳过所有的空格字符

while(isspace(*ptr)) ++ptr;

if(*ptr == '=') //如果后面跟的是'='符号，说明后面就是属性值

{

++ptr; //将ptr后移一位

while(isspace(*ptr)) ++ptr; //跳过所有的空格字符

if(*ptr == '"' || *ptr == '\') //判断属性值是否被引号括起

{

charquote = *ptr;

// fprintf(stderr, "Trying to find quote: %c\n", quote);

constchar *end = strchr(ptr + 1, quote);

if(end == 0)

{

//b= mText.find_first_of(">", a+1);

constchar *end1, *end2;

end1= strchr(ptr + 1, ' ');

end2= strchr(ptr + 1, '>');

if(end1 && end1 < end2) end = end1;

elseend = end2;

if(end == 0) return;
```

```
}

constchar *begin = ptr + 1;

while(isspace(*begin) && begin < end) ++begin;

constchar *trimmed_end = end - 1;

while(isspace(*trimmed_end) && trimmed_end >= begin) --trimmed_end;

val.assign(begin,trimmed_end + 1);

ptr= end + 1;

}

else //否则直接获取其值

{

end= ptr;

while(*end && !isspace(*end) && *end != '>') end++;

val.assign(ptr,end);

ptr= end;

}

// fprintf(stderr,"%s = %s\n", key.c_str(), val.c_str());

mAttributes.insert(make_pair(key,val));

}

else//否则说明格式有错误，并将该key对应的值按空字符串插入结果集

{

// fprintf(stderr,"D: %s\n", key.c_str());

mAttributes.insert(make_pair(key,string()));

}

}

}
```

之后，还有几个关于操作符重载的代码，很好理解，这里仅仅将其列出来就不详细说明了。

```
bool Node::operator==(const Node &n)const //判断两个Node是否都为tag标签，且tag名是否相等

{

if(!isTag() || !n.isTag()) return false;

return!(strcasecmp(tagName().c_str(), n.tagName().c_str()));

}
```

```
Node::operator string() const { //根据是否是tag标签，返回tagName或者text

if(isTag()) return this->tagName();

return this->text();

}

ostream &Node::operator<<(ostream&stream) const { //按对象输出Node

stream<< (string)(*this);

return stream;

}
```

## 三.总结

对HtmlCxx进行学习，可以很好地掌握标准Html解析的流程，以及解析器的内部结构，并掌握C++编程时的一些好习惯和系统组织方式。遗憾的是HtmlCxx在2005年之后就不再更新，且仅仅支持Mozilla的Html标准。它对Html的良好性要求比较高，对于一些含有错误的Html文档，在解析时可能会出现一些意想不到的错误。


在编程过程中，我们可以针对具体的需求，很方便地对其的内部数据结构进行调整，如为节点新加一些属性等。



目前的文档中没有包含tree.h的代码说明，因为它属于第三方代码，希望了解的可以去<http://tree.phisci.com/documentation.html>了解具体的情况。后续有机会还会专门写一篇关于这个tree.h的文档。

如果对此文档中的代码解析有什么问题，或者发现了什么问题，请您发送邮件到我的邮箱[gengyun@sohu.com](mailto:gengyun@sohu.com)。

免费下载《技术指标精解》学会五大技术指标，轻松判断市场趋势 移动平均线、MACD、相对强弱指标... [www.Forex.cn](http://www.Forex.cn)

抗漲省電首選 日立變頻冷氣 精打細算做環保!8/16前購買能源效率 1、2級冷氣機每台政府補助2000元 [www.taiwan-hitac.com](http://www.taiwan-hitac.com)

全新ARM嵌入式cAndroid系統移植 linux程式、Embedded系統 Driver撰寫整合 採Cortex-A8開發板搭配  Google 提供的广告!

分享到:  

桌面管理工具: [fences](#) | [Android的文本编解码工具类](#)

- 2011-10-21 11:02
- 浏览 10
- [评论\(0\)](#)
- [相关推荐](#)

评论

发表评论



[您还没有登录,请您登录后再发表评论](#)



yuanlanxiaup

- 浏览: 19317 次
-  我现在离线

最近访客 [更多访客>>](#)



[hiems](#)



[向左走向右走](#)



[ziyu2012](#)



[zgb061](#)

#### 文章分类

- [全部博客 \(1107\)](#)

#### 社区版块

- [我的资讯](#) (0)
- [我的论坛](#) (0)
- [我的问答](#) (0)

#### 存档分类

- [2012-04](#) (1)
- [2012-01](#) (8)
- [2011-12](#) (104)
- [更多存档...](#)

#### 评论排行榜

- [分享下曾经做的一个JS小游戏——《Battle C ...](#)

#### 最新评论

- [ruishen](#) : ...  
[分享下曾经做的一个JS小游戏——《Battle City》](#)
- [ansjsun](#) : 牛人和小霸王一模一样哦  
[分享下曾经做的一个JS小游戏——《Battle City》](#)

---

声明：ITeye文章版权属于作者，受法律保护。没有作者书面许可不得转载。若作者同意转载，必须以超链接形式标明文章原始出处和作者。

© 2003-2011 ITeye.com. All rights reserved. [ 京ICP证110151号 京公网安备110105010620 ]