HTMLParser Frame

$Hzy9819$

2018 年 5 月 5 日

1 Demand

1.1 Structure Mode

file, string.

1.2 Function

extract :

1.title(tag: h1 ∼ h6);

2.text(tag: p, b, code, em, font, i...);

3.url(tag: href);

...

1.3 Demand

1.use string, fstream, stringstream.

2*.use Regular Expression.

3*.Tree structure.

4*.Reference to http://htmlparser.sourceforge.net/

2 Frame

Is a good idea to use tree structure to reserve html codes.

2.1 Members

2.1.1 Parser

```
class Parser {
private:
    Node Head, Body;
    Tag HTML;
public:
    Parser() {}
    ~Parser() ;
    Parser(Node hd, Node bd, Tag T) : Head(hd), Body(bd), HTML(T) {}
    Parser(stringstream & ss) ;
    Parser(const char * f) {
        using namespace std;
        fstream in(f);
        stringstream ss;
        ss << in.rdbuf();
        *this = Parser{ss};
    }
    Parser(std::string s) {
        std::stringstream ss;
        ss.str(s);
        *this = Parser{ss};
    }
    void ShowHead(long long mode);
    void ShowBody(long long mode);
    void clear();
};
```

In fact we can change file and string into a stream,

so that we only need A stream's structure function .

2.1.2 Node

```
class Node {
private:
        Tag Mtag ;
        vector< Node > SubNode ;
        string Text ;
public:
        Node() {}
        ~Node() ;
        Node(Tag T, vector< Node > * SN, string tx) {
                ...
        }
        void SetTag(Tag T) ;
        void AddSubNode(Node p) ;
        void AddText(string s);
        Tag CheckTag() ;

        Node FindTag(string s);
        void ShowTag(long long mode) ;
        void ShowText() ;
        bool CheckType(long long mode) ;
        void ShowSubNode(long long mode);
        void clear();
};
```

use bitmask express one kind of mode avoid too much port.

And the mode config will be shown in global-setting.

2.1.3 Tag

```
class Tag {
private:
        string TagName;
        long long TagType;
        vector< Attribute > ATT;
public:
        Tag() {}
        ~Tag() ;
        void SetTagType(string s);
        Tag(string Typ, vector< Attribute > * att) {
                ...
        }
        string CheckTagName() ;
        long long ShowType();
        void ShowTagName();
        void ShowAttribute(long long mode);

        void AddAttribute(Attribute att);
        void clear();
};
```

divide Tag into different types is convenience to extract different kind of information.

2.1.4 Attribute

```
struct Attribute {
    std::string Type, Value;
    Attribute() {}
    Attribute(std::string tp, std::string val) : Type(tp), Value(val) {}
    void Show(long long mode) ;
    void clear() ;
};
```

maybe is no use ? Some information can be hidden in the attributions

2.2 Detail

2.2.1 structure

We use a stream to input the website(html code), it's easy to divide the code into tag and texts. So that we just need to find each $'<'$ and $'>'$ to find a tag. But it's also a matter to know a tag's scope(some tag has a $</>$ to stop it with some not), it forces us to list those special tags who need a $</>$ to stop in the global-setting.

When it comes to a tag, we also need to analyze the tag-name and select one or more tag-type(title, text, url...) to standard it. Also, attribute should not be left.

Then things become easy, we just need find tag $->$ analyze tag $->$ input text $->$ find next tag.

```
Tag Build_Tag(std::string s) {
    using namespace std;
    int n = s.size() - 1, p = 1;
    string Name, typ, val;
    bool SetName = 0;
    Tag T;
    while(p < n) {
        if(s[p] == Space) {
            do {
                ++p;
            }while(p < n && s[p] == Space);
            continue;
        }
        if(!SetName) {
            while(p < n && s[p] != Space) Name += s[p++];
            Name = Tolower(Name);
            T.SetTagType(Name);
            SetName = 1;
        }
        else {
            typ = val = Empty;
            while(p < n && s[p] != Space && s[p] != EqualMark)
                 typ += s[p++];
            if(s[p] == EqualMark) {
                do {
                    ++p;
                }while(p < n && s[p] == Space);
                if(s[p] != QuateMark)
                    while(p < n && s[p] != Space) val += s[p++];
                else {
                    do {
                        val += s[p++];
                    }while(p < n && s[p] != QuateMark);
                    if(p < n) val += s[p++];
                }
            }
            Attribute att(typ, val);
            T.AddAttribute(att);
        }
    }
    return T;
}
```

```
std::string Build_Node(Node & n, std::stringstream & st,
     std::string tagname, std::string Lef) {
    using namespace std;
    string temp = Empty, tmp = Lef;
    size_t p1, p2;
    bool b1 = 0, b2 = 0;
    do {
        temp = temp + Spacestring + tmp;

        if(!b1) p1 = tmp.find(LeftMark, 0);
        if(!b2) p2 = tmp.find(RightMark, 0);
        if(!b1 && (p1 != string::npos))
            b1 = 1;
        if(!b2 && (p2 != string::npos))
            b2 = 1;

        if(b1 && b2) {
            p1 = temp.find(LeftMark, 0);
            p2 = temp.find(RightMark, 0);
            if(p1 > p2) {
                tmp = temp.substr(0, p2);
                n.AddText(tmp);
                tmp = temp.substr(p2 + 1);
                temp = Empty;
                b1 = b2 = 0;
                continue;
            }

            tmp = temp.substr(0, p1);
            n.AddText(tmp);

            tmp = temp.substr(p1, p2 − p1 + 1);
            Tag T = Build_Tag(tmp);
            tmp = temp.substr(p2 + 1);
            temp = Empty;

            b1 = b2 = 0;
            if(AssMark + tagname == T.CheckTagName())
                return tmp;
            if(IgnoreMark == T.CheckTagName().at(0)
            || AssMark == T.CheckTagName().at(0))
                continue;


            Node sn;
            sn.SetTag(T);


            if(CheckBanner(T.CheckTagName())) {
                do {
                    temp = temp + Spacestring + tmp;
                    if(Tolower(tmp).find(AssMark + T.CheckTagName(), 0)
                != string::npos) {

                        string tp = Empty;
                        tp += LeftMark,
                tp += AssMark,
                    tp += T.CheckTagName() + RightMark;

                        size_t p = Tolower(temp).find(tp, 0);
                        tmp = temp.substr(0, p);
                        sn.AddText(tmp);

                        while(temp[p] != RightMark) p++;
                        tmp = temp.substr(p + 1);
                        temp = Empty;
                        break;
                    }
                    if(!(st >> tmp)) break;
                }while(1);
            }
            else if(CheckAss(T.CheckTagName()))
                tmp = Build_Node(sn, st, T.CheckTagName(), tmp) ;
            n.AddSubNode(sn);
            continue;
        }
        if(!(st >> tmp)) break;
    }while(1);
    return temp;
}
```

2.2.2 About Bitmask

It's a huge work to write a function for extracting different kind of text, so we classify different tag to different kind of text, and use Bitmask to represent it. This method can lower code length effective and easier to use. But it also forces us to define some of constant in global-setting to represent different kind of text. And enumerate tags to classify them is also a slavery.

Here is a sample(no need to regard it as a standard):

```
const LL NoneType = 0;
const LL Title = (1LL << 0);
const LL Text = (1LL << 1);
const LL Url = (1LL << 2);
const LL Img = (1LL << 3);
const LL Script = (1LL << 4);
const LL Button = (1LL << 5);
const LL Table = (1LL << 6);
const LL Video = (1LL << 7);
const LL audio = (1LL << 8);
const LL ALL = (1LL << 62) + ((1LL << 62) − 1);

const LL Att_Type = (1LL << 0);
const LL Att_Value = (1LL << 1);
...
```

2.2.3 About Banner

We need to find each <> in the html code, but in some text(like javascript) we also can find <>. That may be a mistake when we deal with it using the method above. So that we do need to skip this kind of text.

```
const std::string Banner = % script code ruby %;
```