实验 03 HTML CSS Selector

姓名：罗毅凡
学号：2024202715

2025 年 11 月 30 日

目录

1 需求分析

1. 通过**树数据结构**对 HTML DOM 层次结构进行建模。实现**常用 HTML CSS Selector 以及对应的获取文本、HTML、链接**等操作。

2. 开发测试或者**演示程序**，展示 HTML CSS Selector 的使用方法，验证正确性。程序执行的命令包括：

   - **read(file_name_or_url)**：从文件读入 Html
   - **query(selector)**：根据 selector 从全文档检索所有符合条件的 nodes，按顺序打印每个 node 的标签类型，id，class，同时将这些 nodes 保存，以供后续操作。
   - **Out[k].innerText**：根据上次 query 操作得到的 node list，找到第 k 个，输出他的 innerText
   - **Out[k].outerHTML**：根据上次 query 操作得到的 node list，找到第 k 个，输出他的 outerHTML
   - **Out[k].query(selector)**：根据上次 query 操作得到的 node list，找到第 k 个，再次调用选择器进行查询。按顺序打印每个 node 的标签类型，id，class，同时更新保存的 nodes
   - **Out[k].href**：根据上次 query 操作得到的 node list，找到第 k 个，输出他的 href（仅对 a 标签）

3. **额外功能实现：**

   - **xpath(path)**：支持 xpath 语法查找
   - **Out[k].xpath(path)**：根据上次 query 操作得到的 node list，找到第 k 个，进行 xpath 查找

4. **测试数据**

   - `example.html`：
     `query(.class1)`：`[div.class0.class1, span.class1, img.class1]` 等
   - `lab3_news.html`：
     `query(.holding.with-hover.weibo-logo)`：`[div.holding.with-hover.weibo-logo]` 等
   - `lab3_newslist.html`：
     `query(input~input)`：`[input#search_btn.submit]` 等

2 概要设计

- **为实现上述程序功能，应以树结构对 HTML DOM 层次结构进行建模，如何实现对选择字符 selector 的解析，遍历树结构进行选择，支持对已选择节点的后续操作，最后包装命令台交互**。

1. 树结构的数据类型为：

```
typedef struct Node{
    NodeType type;
    string tagname;
    string content;
    string id;
    vector<string> classlist;
    map<string,string> attributions;
    struct Node* parent;
    vector<Node*> children;
}Node;
```

2. 单查找内容数据结构为：

```
typedef struct one_query{
    string tag;
    string id;
    vector <string> classes;
}one_query;
```

3. 本程序包含四个模块：

   (a) **主程序模块**：

```
void main(){
    初始化;
    while(true){
        接受命令;
        处理命令;
    }
}
```

   (b) **文档读取模块**：读取本地文档或解析 url，构建 html 树结构

   (c) **选择模块**：解析 selector，遍历树结构，输出符合要求的节点信息

   (d) **节点后续操作模块**：对已选择节点进行后续操作

3 详细设计

3.1 read(file_or_url)

- **实现逻辑：**
  将整个 html 文件读取到字符串中，然后**解析这个字符串**来构造树，构造树结构时通过维护一个 node 指针 cur 来实现**层次结构的构建**，cur 指向当前读取到的 html 元素地址：
  - 读取到文本内容：构建一个文本节点作为 cur 的 child

- 读到元素开始：构建一个元素节点作为 cur 的 child，并更新 cur 为这个新节点指针
- 读到元素结束：更新 cur 为 cur->parent
- 特殊情况：读到自闭和标签、注释块等直接跳过

这样我们就完成动态构建了一个 html 层次结构树。

- **伪代码：**

```
Node* creat_dom(string path){
    content=读取文件(path)
    root=NewNode("document")
    cur=root  //当前节点指针

    while 遍历 content:
        //1.文本
        text=获取当前位置到下一个 '<' 之间的字符
        if text:
            cur.addChild(NewTextNode(text))

        //2.标签
        tagStr=获取 '<' 和 '>' 之间的内容
        if tagStr 以'/'开头:      //结束标签
            cur=cur.parent       //维护cur

        else if tagStr 是注释或声明:
            continue

        else:     //开始标签
            node=...  //创建新元素节点
            cur.addChild(node)

            if NOT isSelfClosing(node):
                cur=node        //维护cur

    return root.children[0]
```

3.2 query(selector)

- **实现逻辑：**
  我将选择条件分为两类：
  - 针对当前节点的选择条件
  - 对前后关系的选择条件，这个又分几种，这里不细说

  所以，这里要实现对复杂输入逻辑的选择，首先是处理 selector，区分两种条件，在依次遍历查找：

- 处理 selector 维护一个当前条件关系，然后循环选择并去重
- 根据当前的条件关系，选择所有符合条件的节点，再去重（这里逻辑比较粗暴，效率相对会低一些）

- **伪代码：**

```
vector<Node*> query(string selector,Node*html){
    selector_str=在selector的'>','+','~'两侧添加空格    //预处理selector
    tokens=split(selector_str, " ")

    sources=[htmlNode]   //上一轮匹配到的节点
    relation=" "   //维护条件关系

    for token in tokens:
        //更新关系
        if token is ">" OR "+" OR "~":
            relation=token
            continue

        rule=MakeQuery(token)
        targets=[]    //存储本轮匹配结果

        for node in sources:
            if relation==" ":
                found=FindDescendants(node, rule)
                targets.addAll(found)

            else if relation==">": //只看孩子节点
                for child in node.children:
                    if CheckNode(child, rule): targets.add(child)

            else if relation=="+" OR "~": //只看兄弟节点
                if node.parent is NULL: continue
                siblings=node.parent.children
                index=FindIndex(siblings, node)

                if relation=="+":  //只看第一个孩子
                    next_sib=FindFirstElementAfter(siblings, index)
                    if CheckNode(next_sib, rule): targets.add(next_sib)

                else if relation=="~": //所用后代节点
                    for k from index + 1 to end:
                        if CheckNode(siblings[k], rule): targets.add(
                            siblings[k])

        // 4. 准备下一轮
```

```
        sources=Unique(targets)   //去重
        if sources is empty: return []
        relation=" "

    return sources
}
```

3.3 xpath(path)

- **实现逻辑:**
  在前面 query 实现的基础上，去实现 xpath 有很多方式，我这里思路比较直接:
  - 把输入的 path **解析翻译**成 query 中的 selector，然后直接复用 query 来实现 xpath 查找

  具体的解析翻译逻辑:
  - 后代关系: 将 // 转换成
  - 直接子代关系: 将 / 转换成 >
  - 属性内容: 将 [@id='val'] 转换成 #val ，将 [@class='val'] 转换成 .val

- **伪代码:**

```
string xpath_to_css(string xpath) {
    FOR 遍历 xpath 中的每个字符(索引 i):
        IF 当前字符是 '/':
            IF 下一个字符也是 '/':
                css 追加 " "
                跳过下一个字符 (i++)
            ELSE:
                css 追加 " > "

        ELSE IF 当前字符是 '[':
            找到对应的右括号 ']' 的位置
            提取括号内的 content
            去除 content 中的多余空格

            从 content 中提取引号内的 value

            IF content 以 "@id" 开头:
                css 追加 "#" + value
            ELSE IF content 以 "@class" 开头:
                css 追加 "." + value
            ELSE:
                提取 key (去掉开头的 '@')
                css 追加 "[" + key + "='" + value + "']"
```

```
                将索引 i 移动到右括号的位置

            ELSE:
                css 直接追加当前字符
        RETURN css
}
```

3.4 Out[k]

- **实现逻辑：**根据 k 打印节点信息
- **代码实现：**

```
void print_text(Node* node){
    if(!node){
        cout<<endl;
        return;
    }else if(node->type==TEXT_NODE){
        cout<<node->content<<endl;
        return;
    }else{
        for(Node* child : node->children){
            print_text(child);
        }
    }
}
void print_outerhtml(Node* node){
    if (!node) return;
    if (node->type == ELEMENT_NODE) {
        cout<<"<"<<node->tagname;
        if (!node->id.empty())cout<<" id=\""<<node->id<<"\"";
        if(!node->attributions.empty()){
            for(auto const& [key,val]:node->attributions){
                cout<<" "<<key<<"=\""<<val<<"\"";
            }
        }
        cout<<">"<<endl;
    } else if (node->type == TEXT_NODE) {
        cout<<node->content<<endl;
    }
}
```

4 调试分析 & 代码优化历程

这里只列出主要的 coding 过程中遇到的问题，小的试错就不说了

1. 最开始对每种类型的 selector 设置不同的查找函数，最后放弃这个方案，主要有两点原因：
   - 代码量大且代码重复臃肿
   - 对混合查找不可行，逻辑会更复杂

   最后采用了对 selector 解析的方法，并维护一个 relation 实现对不同的关系的选择。
2. 去重是很重要的一步，最开始没有对结果去重，发现输出特别多重复，而且有错误的输出。

5 使用手册

与附录中使用手册相同

5.1 1. 概述

`HtmlSelectorTool` 是一个基于 C/C++ 实现的高效 HTML 解析和数据抽取工具。它将 HTML 文档结构建模为一棵树，并支持常用的 CSS 选择器（Selector）进行节点定位，以及方便地抽取节点的文本内容、完整 HTML 代码和链接信息。本手册将指导用户如何导入 HTML 文档、执行 CSS 查询以及对查询结果进行操作。

5.2 2. 核心功能

功能	描述
HTML 建模	将 HTML 文件或 URL 内容解析为内存中的树形数据结构（DOM）。
CSS 选择器	支持实现列表中的基本 CSS 选择器，快速定位目标节点。
数据抽取	支持抽取节点的内部文本、外部 HTML 代码以及链接（`href`）属性。
链式查询	支持对已选中的节点集合再次执行 CSS 查询，实现局部精确抽取。

5.3 3. 命令行交互操作

程序提供一个交互式的命令行界面，用户可以通过输入以下命令进行操作：

5.3.1 3.1 文档加载：read()

该命令用于加载 HTML 内容到程序中，构建 DOM 树。

命令格式	描述
`read(file_name)`	从本地文件加载 HTML 内容。
`read(url)`	从指定的 URL 地址下载并加载 HTML 内容。

示例:

```
> read(C:\Users\luo yifan\Desktop\GSAI-2th-mystudy\Data_structure\labs\lab3\实验03\examples\
文档加载成功。
```

5.3.2 3.2 全局查询：query()

该命令用于在当前加载的整个 HTML 文档中，根据给定的 CSS 选择器检索所有符合条件的节点。

命令格式	描述
`query(selector)`	执行查询，结果将存储为一个有序列表，供后续操作使用。

支持的基本 CSS 选择器 (Implemented Selectors):

选择器	例子	例子描述
.class	.intro	选择 class=”intro” 的所有元素。
.class1.class2	.name1.name2	选择 class 属性中同时有 name1 和 name2 的所有元素。
.class1 .class2	.name1 .name2	选择作为类名 name1 元素后代的所有类名 name2 元素。
#id	#firstname	选择 id=”firstname” 的元素。
*	*	选择所有元素。
element	p	选择所有 <p> 元素。
element.class	p.intro	选择 class=”intro” 的所有 <p> 元素。
element,element	div, p	选择所有 元素和所有 <p> 元素。
element element	div p	选择 元素内的所有 <p> 元素。
element>element	div > p	选择父元素是 的所有 <p> 元素。
element+element	div + p	选择紧跟 元素的首个 <p> 元素。
element1~element2	p ~ ul	选择前面有 <p> 元素的每个 元素。

查询结果输出: 程序将按文档顺序打印每个匹配节点的简要信息，并将其保存到 **结果列表** (`Out`) 中。

示例:

```
> query(.class2.class3)
[div.class2.class3, span.class3.class2, img.class2.class0.class3]
```

```
> query(#id4.class2)
[div#id4.class2]
```

5.3.3 3.3 XPath 查询：xpath()

该命令允许用户使用 XPath 语法在当前文档中进行灵活的节点查找和定位。

命令格式	描述
xpath(path)	解析 XPath 表达式，检索符合条件的所有节点。

支持的 XPath 语法 (Supported Syntax):

表达式	描述	示例
nodename	选取此节点的所有子节点。	body
/	从根节点选取（绝对路径）。	/body/div
//	从匹配选择的当前节点选择文档中的节点，而不考虑它们的位置（递归查找）。	//div
.	选取当前节点。	.
..	选取当前节点的父节点。	..
@	选取属性（常用于谓语条件中）。	//div[@id='id1']
*	通配符，匹配任何元素节点。	/body/*

示例:

```
> xpath(/body/div)
[div.class0.class1, div, div#id1, div, ...]
> xpath(//div[@id='id1'])
[div#id1]
```

5.3.4 3.4 结果列表操作：Out[k].operation

查询完成后，用户可以通过 Out[k] 访问结果列表中的第 k 个节点 (索引 k 从 0 开始)。

A. 获取内部文本：Out[k].innerText 返回节点及其所有后代节点的**纯文本内容**，去除所有 HTML 标签。

示例:

```
> Out[0].innerText
div#id4.class2
```

B. 获取完整 HTML：Out[k].outerHTML 返回节点自身的**完整 HTML 代码**，包括其自身标签及其所有子节点。

示例:

```
> Out[0].outerHTML
<div id="id4" class="class2" id="id4">
```

C. 获取链接属性：Out[k].href 仅适用于 <a> 标签节点。返回该节点的 href 属性值。
示例:

```
> Out[0].href
/a.html
```

D. 链式查询/二次查询：Out[k].query(selector) 以 Out[k] 节点为根，在其内部进行一次新的 CSS 查询。新的查询结果将**覆盖**原有的结果列表 (Out)。
示例 (在列表的第一个 div 节点内部查找所有的 p 标签):

```
> Out[0].query(*)
[p#id5.class5]
```

E. 链式 XPath 查询：Out[k].xpath(path) 以 Out[k] 节点为上下文环境（当前节点），执行 XPath 查询。支持使用 . (当前节点) 和 .. (父节点) 进行相对导航。
示例:

```
> Out[0].xpath(..)
[html]
> Out[0].xpath(.)
[body]
```

5.3.5 3.5 退出：q 或 quit 或 exit

6 功能测试报告

与附录中功能测试报告相同

6.1 1. 测试概述

本次测试旨在验证 HTML CSS Selector 工具的核心功能完整性与健壮性。测试范围覆盖了本地文件解析、网络 URL 资源加载、基础与复杂 CSS 选择器逻辑（包括层级、组合、兄弟选择器）、以及节点属性（innerText, outerHTML, href）的提取。

6.2 2. 测试资源

- **本地测试文件：** C:\Users\luo yifan\...\examples\example.html
- **网络测试地址：** https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/ (BeautifulSoup 中文文档)

6.3 3. 详细测试记录

6.3.1 3.1 文件加载与基础环境测试

ID	测试功能	输入命令	实际输出结果	状态
TC-01	**本地文件读取**	`read(...\example.html)`	文档加载成功。	**PASS**

6.3.2 3.2 CSS 选择器查询测试 (Query Functionality)

测试基于 `example.html` 的 DOM 结构。

ID	测试功能	输入命令	实际输出结果	结果
TC-02	**Class 选择器**	`query(.class1)`	`[div.class0.class1, ...]`	**PASS**
TC-03	**ID 选择器**	`query(#id1)`	`[div#id1]`	**PASS**
TC-04	**多 Class 组合**	`query(.class2.class3)`	`[div.class2.class3, ...]`	**PASS**
TC-05	**ID+Class 组合**	`query(#id4.class2)`	`[div#id4.class2]`	**PASS**
TC-06	**后代选择器**	`query(.class4 .class5)`	`[div.class5, p#id5.class5]`	**PASS**
TC-07	**复杂混合查询**	`query(div.class4 p.class5#id5)`	`[p#id5.class5]`	**PASS**
TC-08	**通用兄弟 (~)**	`query(div.class2.class3 ~ .class0)`	`[img.class2.class0.class3]`	**PASS**

6.3.3 3.3 结果集操作与数据抽取 (Extraction)

基于 `query(a)` 的结果集进行测试。

ID	测试功能	输入命令	实际输出结果	状态
TC-09	**Tag 选择器**	`query(a)`	`[a, a]`	**PASS**
TC-10	**innerText**	`Out[0].innerText`	`p#id5.class5`	**PASS**
TC-11	**outerHTML**	`Out[0].outerHTML`	`<a href="/a.html">`	**PASS**
TC-12	**href 属性**	`Out[0].href`	`/a.html`	**PASS**
TC-13	**链式查询**	`Out[0].query(*)`	`[p#id5.class5]`	**PASS**

6.3.4 3.4 网络加载与压力测试 (Network & Stress Test)

ID	测试功能	输入命令	实际输出结果	状态
TC-14	**URL 下载**	`read(https://...)`	检测到 URL... 加载成功。	**PASS**
TC-15	**真实 DOM 查询**	`query(div.line)`	返回包含 40+ 个节点的长列表	**PASS**
TC-16	**大数据抽取**	`Out[0].outerHTML`	`<div class="line">`	**PASS**

6.3.5 3.5 XPath 选择器查询测试 (XPath Query)

ID	测试功能	输入命令	实际输出结果	状态
TC-17	**绝对路径**	`xpath(/body/div)`	`[div.class0.class1, ...]`	**PASS**
TC-18	**全局递归**	`xpath(//div)`	`[div.class0.class1, ...]`	**PASS**
TC-19	**属性谓语**	`xpath(/body/div[@id='id1'])`	`[div#id1]`	**PASS**
TC-20	**通配符 (*)**	`xpath(/body/*)`	`[div.class0.class1, ...]`	**PASS**
TC-21	**名称查找**	`xpath(body)`	`[body]`	**PASS**
TC-22	**相对导航**	`Out[0].xpath(..)`	`[html]`	**PASS**

6.4 4. 测试结论与评价

1. **功能完备性：**程序完全覆盖了实验要求的所有功能点。
2. **交互体验：** `read`, `query`, `Out[k]` 的交互逻辑清晰流畅，且支持从结果集中进行**链式查询 (Nested Query)**，极大地增强了工具的灵活性。
3. **鲁棒性：**
   - 能够处理本地长路径文件。
   - **网络模块集成良好**，能够自动识别 URL，下载并解析 HTTPS 协议的真实网页。
   - 在解析包含大量节点的真实网页时，程序响应迅速，未出现崩溃或内存溢出。

最终结论：该 HTML CSS Selector 程序功能强大，运行稳定，**通过所有测试用例**，具备投入实际使用的能力。

7 附录

```
罗毅凡_2024202715_实验3 （总文件夹）
1   Selector_project   （项目工程文件）
        bin                （存放执行码）
            lab3_tree.exe
```

```
        src                 (存放源代码)
            lab3_tree.cpp
        .vscode             (存放工程配置)
            settings.json
2   实验报告.pdf
3   使用手册.pdf
4   功能测试报告.pdf
```