开源 C++版本 CGI 库 CGICC 入门

易剑 2015/3/14

目录

目录.....1
1. 简介.....1
2. CGICC 组成.....1
3. CGI 输入处理子模块类结构.....2
3.1. Cgicc.....2
3.2. CgiEnvironment.....2
3.3. HTTPCookie.....2
3.4. CgiInput.....3
3.5. FormFile.....3
3.6. FormEntry.....3
4. CGI 输入处理子模块初始化流程.....3
5. 编译和安装 CGICC.....4
6. CGICC 使用示例.....5
6.1. 页面效果.....5
6.2. HTML 文件.....5
6.3. test.txt 文件.....6
6.4. CGI 文件.....6
6.5. 运行效果.....8
7. HTML 输出子模块类图.....10
7.1. HTTPContentHeader.....13
7.2. HTMLElement::render()函数.....13
8. 问题？.....16

1. 简介

CGICC 是一个 C++语言实现的开源 CGI 库，采用 LGPL 授权协议，使用较为简单。

CGICC 官网：http://www.gnu.org/software/cgicc/，截止 2015/3/14，CGICC 最新稳定版本为 3.2.16，下载地址是：http://ftp.gnu.org/gnu/cgicc/cgicc-3.2.16.tar.gz，最新更新时间为 2014/12/7（令人惊讶和欣慰的是作为古老的 CGI，CGICC 还在不断的更新）。

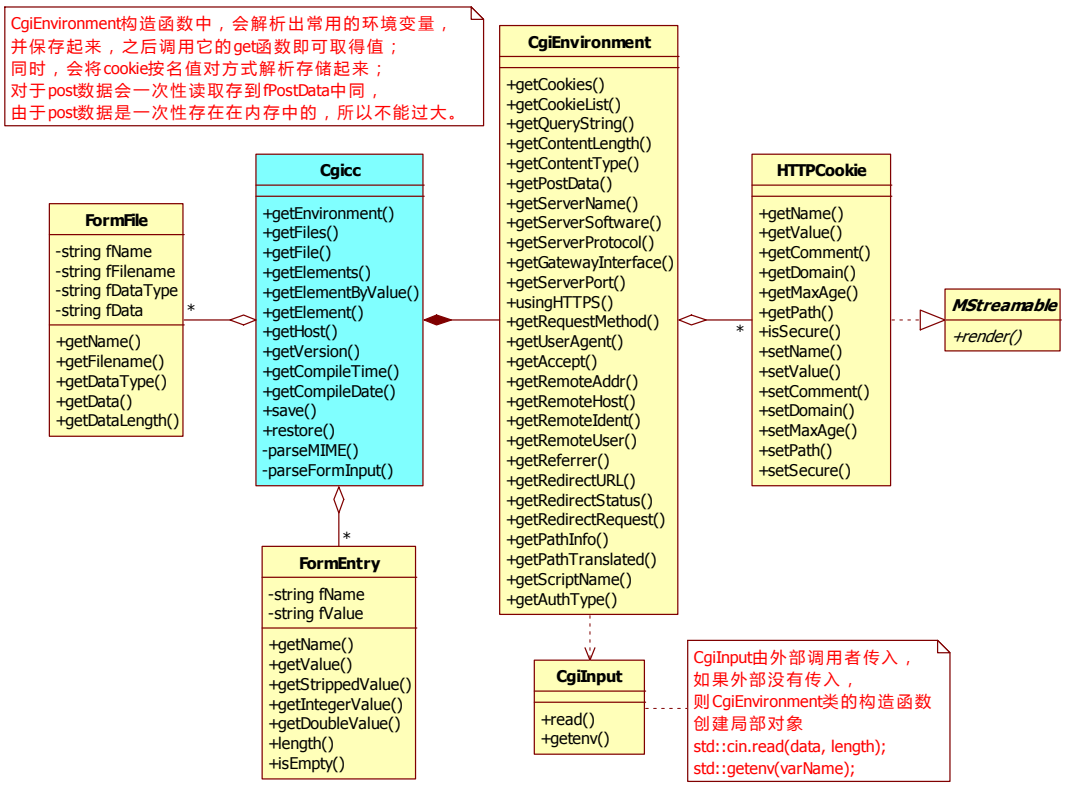
2. CGICC 组成

CGICC 由两大部分组成：

1) CGI 输入处理子模块
2) HTML 输出子模块

本文暂只介绍 CGI 输入处理子模块，对于 HTML 输出，推荐 Google 开源的 ctemplate（https://github.com/OlafvdSpek/**ctemplate**）。

3. CGI 输入处理子模块类结构



3.1. Cgicc

CGICC 的一类，通常直接在 CGI 的入口函数，如 main 函数中定义一个 CGICC 对象，然后即可使用 CGICC 提供的各种能力。

3.2. CgiEnvironment

提供 get 系列方法取各环境变量的值。

3.3. HTTPCookie

提花 get 系列方法取各 Cookie 的值，并支持 set 新增或修改 Cookie 值。

3.4. CgiInput

CgiEnvironment 内部类，仅供 CgiEnvironment 使用。

3.5. FormFile

提供访问 HTML 的 Form 中的被上传文件信息和数据接口。

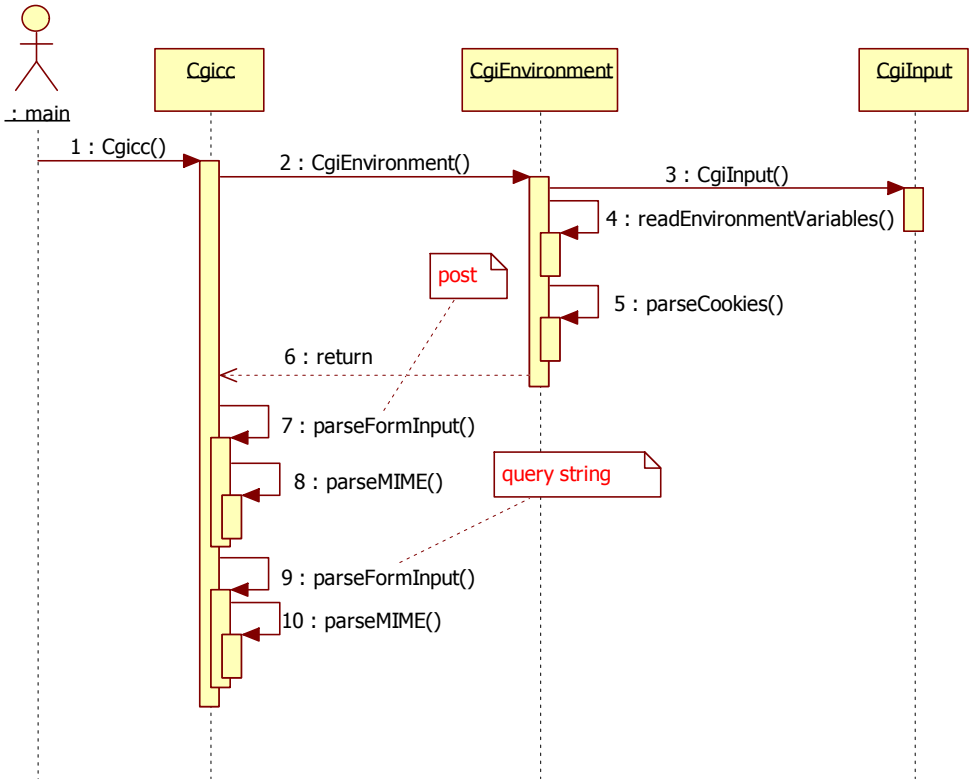
3.6. FormEntry

提供访问 HTML 的 Form 中的非被上传文件类的信息和数据接口。取 URL 参数值示例：

```
// http://127.0.0.1/?param_name=param_value
cgicc::form_iterator iter = cgi.getElement("param_name");
if (iter != cgi.getElements().end())
{
    std::string param_value = iter->getValue();
}

// 也可以这样做：
std::string param_value = cgi("param_name");

// 除此之外，FormEntry 还提供了直接取指定数据类型的参数值，如：getIntegerValue、getDoubleValue
```

4. CGI 输入处理子模块初始化流程

初始化流程是由 Cgicc 构造函数触发的，一般可在 CGI 的 main 函数中定义一个 Cgicc 对象：



5. 编译和安装 CGICC

详细编译步骤如下：

1) 将 CGICC 源代码包（本文下载的是 cgicc-3.2.16.tar.gz）上传到 Linux 某目录（本文将 CGICC 源代码包 cgicc-3.2.16.tar.gz 上传到/tmp 目录）；
2) 登录 Linux，并进入目录/tmp；
3) 解压 CGICC 源代码包 cgicc-3.2.16.tar.gz：tar xzf cgicc-3.2.16.tar.gz；
4) 解压后，会在/tmp 下产生一个子目录 cgicc-3.2.16，进入到这个子目录；
5) 然后执行 configure 命令（本文指定的安装目录为/usr/local/cgicc-3.2.16，可以根据需要设定为其它目录），以生成 Makefile 编译文件，如果要在共享库中使用 CGICC，请使用下列编译命令：

```
./configure --prefix=/usr/local/cgicc-3.2.16 CXXFLAGS=-fPIC LDFLAGS=-fPIC
```

否则，可按如下命令编译：

```
./configure --prefix=/usr/local/cgicc-3.2.16
```

在一些环境上，如果不带-fPIC 编译静态库，使用静态库时，就会报链接错误。

6) 执行 make 编译：make
7) 安装 CGICC 库：make install
8) 为/usr/local/cgicc-3.2.16 建立不带版本号的软链接：

```
ln -s /usr/local/cgicc-3.2.16 /usr/local/cgicc
```

至此，CGICC 库就安装好了！

6. CGICC 使用示例

6.1.页面效果

6.2.HTML 文件

页面效果对应的 HTML 文件内容如下（HTML 中的 id 一般是给前端如 js 使用的，而 name 通常是给服务端如 CGI 使用的）：

```
<html>
    <head>
       <title>upload</title>
    </head>

    <body>
        <p>upload:
        <div>
            <form action="/cgi-bin/upload.cgi" method="post" name="formname"
                  enctype="multipart/form-data">
                <input type="text" id="id1" name="name1" />
                <input type="text" id="id2" name="name2" />

                <p>
                <input type="file" id="fileid" name="filename" />
                <input type="submit" value="upload" id="upid" name="upname" />
            </form>
        </div>
    </body>
</html>
```

注意，上传文件时，Form 的 enctype 属性值必须被设定为 multipart/form-data。

6.3.test.txt 文件

test.txt 是一个被上传的文件，内容只有一行：**0123456789**。

6.4.CGI 文件

```
// 如果是 Exe 形式的 CGI，则使用如下语句编译：
// g++ -g -o upload.cgi upload.cpp -I/usr/local/cgicc/include /usr/local/cgicc/lib/libcgicc.a
// 如果是共享库（Windows 平台叫动态库）形式的 CGI，则使用如下语句编译：
// g++ -g -o upload.cgi upload.cpp -shared -fPIC -I/usr/local/cgicc/include /usr/local/cgicc/lib/libcgicc.a
#include <stdio.h>
#include <sstream>
#include "cgicc/Cgicc.h"
#include "cgicc/HTMLClasses.h"
#include "cgicc/HTTPHTMLHeader.h"

int main(int argc, char **argv)
{
    try
    {
        cgicc::Cgicc cgi;

        // Output the HTTP headers for an HTML document,
        // and the HTML 4.0 DTD info
        std::cout << cgicc::HTTPHTMLHeader()
                  << cgicc::HTMLDoctype(cgicc::HTMLDoctype::eStrict)
                  << std::endl;
        std::cout << cgicc::html().set("lang", "en").set("dir", "ltr")
                  << std::endl;

        // Set up the page's header and title.
        std::cout << cgicc::head() << std::endl;
        std::cout << cgicc::title() << "GNU cgicc v" << cgi.getVersion()
                  << cgicc::title() << std::endl;
        std::cout << cgicc::head() << std::endl;

        // Start the HTML body
        std::cout << cgicc::body() << std::endl;

        // Print out a message
        std::cout << cgicc::h1("Hello, world from GNU cgicc") << std::endl;
        const cgicc::CgiEnvironment& env = cgi.getEnvironment();
```

```
        std::cout << "<p>accept: " << env. getAccept() << std::endl;
        std::cout << "<p>user agent: " << env.getUserAgent() << std::endl;

        std::cout << "<p>cookie: " << std::endl;
        const std::vector<cgicc::HTTPCookie>& cookies = env.getCookieList();
        for (std::vector<cgicc::HTTPCookie>::size_type i=0; i<cookies.size(); ++i)
        {
            const cgicc::HTTPCookie& cookie = cookies[i];
            std::cout << "<br>    cookie[" << cookie.getName()
                      << "] = " << cookie.getValue() << std::endl;
        }

        std::cout << "<p>query string: " << env.getQueryString() << std::endl;
        std::cout << "<p>remote: " << env.getRemoteAddr() << ":" << env.getServerPort()
                  << std::endl;

        std::cout << "<p>form: " << std::endl;
        const std::vector<cgicc::FormEntry>& form_entries = cgi.getElements();
        for (std::vector<cgicc::FormEntry>::size_type i=0; i<form_entries.size(); ++i)
        {
            const cgicc::FormEntry& form_entry = form_entries[i];
            std::cout << "<br>    form["
                      << form_entry.getName() << "] = "
                      << form_entry.getValue() << std::endl;
        }

        //
        // 取被上传的文件信息
        //

        // 使用 getFile 取得指定的被上传文件信息
        cgicc::const_file_iterator file_iter = cgi.getFile("file");

        // 使用 getFiles 可以取得所有被上传文件信息
        if (file_iter == cgi.getFiles().end())
        {
            std::cout << "<p>file: " << cgi.getFiles().size() << std::endl;
        }
        else
        {
            const cgicc::FormFile& file = *file_iter;
            std::cout << "<p>file: " << std::endl;
            std::cout << "<br>    name: "
                      << file.getName() << std::endl;
```

```
            std::cout << "<br>    filename: "
                      << file.getFilename() << std::endl;
            std::cout << "<br>    type: "
                      << file.getDataType() << std::endl;
            std::cout << "<br>    size: "
                      << file.getDataLength() << std::endl;
            std::cout << "<br>    content: "
                      << file.getData() << std::endl;
        }

        // Close the document
        std::cout << cgicc::body() << cgicc::html();
    }
    catch(const std::exception& e)
    {
        // handle error condition
    }

    return 0;
}
```

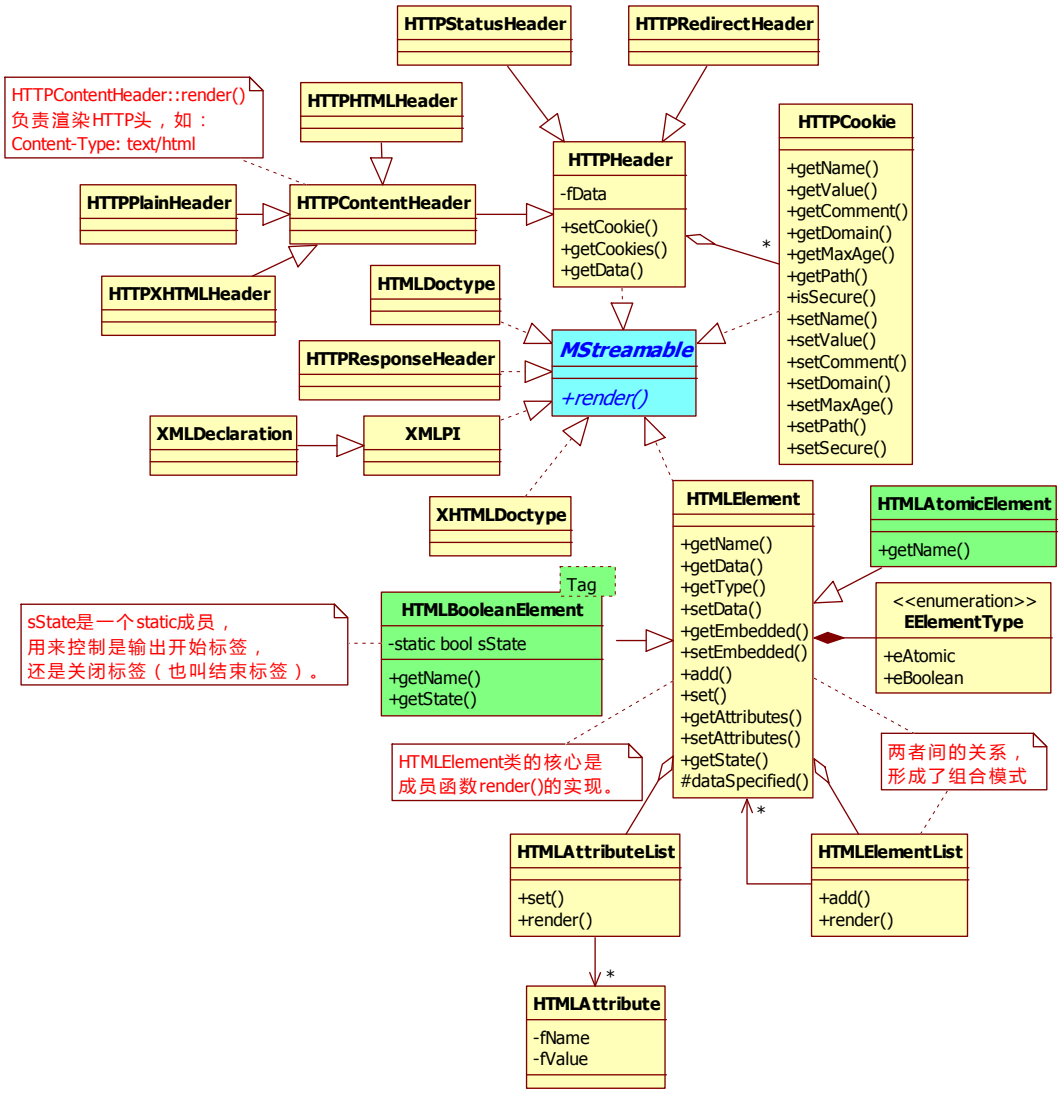
6.5.运行效果

点击 HTML 页面的 upload 按钮后，页面变成如下：

```
Hello, world from GNU cgicc

accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8

user agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.36 (KHTML, like Gecko)
Chrome/39.0.2172.95 Safari/537.36

cookie:
    cookie[pgv] = 445364884
    cookie[ku]                                                                          =
f0ab9e006c7f4d5a4e9b394fc44fafc8afd6df6d373f9ff5f2946047974daf0ef9b00c6a1d7c341b
    cookie[uid] = zhangshan
    cookie[post-shareto-guide] = 1
    cookie[si] = s4001534976
    cookie[info] = ssid=s9175124444
    cookie[pvid] = 6963827212
    cookie[code_user_name] = A5C9579BE8B7C0E0

query string:
```

```
remote: 120.16.82.66:80

form:
    form[name1] = abc
    form[name2] = xyz
    form[upname] = upload

file:
    name: filename
    filename: test.txt
    type: text/plain
    size: 10
    content: 0123456789
```

7. HTML 输出子模块类图



- **HTMLBooleanElement**

  注意 **sState** 是类 HTMLBooleanElement 的静态数据成员，sState 的数据类型为 bool。

- **标签**

  对于<html></html>，前者<html>叫开始标签，后者</html>叫关闭标签。

- **EElementType**

  枚举类型，定义了两个枚举值：eAtomic 和 **eBoolean**，eAtomic 对应的实现类为 **HTMLAtomicElement**，eBoolean 对应的实现为 **HTMLBooleanElement**。类似于 strong 类的为 eBoolean 类型，而类似于 hr、br 之类的为 eAtomic 类型。对于 eAtomic 类型的 HTML 标签，它没有对应的关闭标签（也叫结束标签），观察以下两组的差别：

```
<br />
<strong>This text is strong</strong>

<hr />
```

```
<p>This is some text</p>
```

可以看到 **br** 和 **hr** 均是 eAtomic 类型的标签，而 **strong** 和 **p** 均是 eBoolean 类型的标签。

■ **fEmbedded**

对于 **eBoolean** 类型的标签，在 HTMLElement::**render**()函数的实现中，会发现还区分是否有 fEmbedded，什么是有 fEmbedded 的 eBoolean 类型标签？

```
下面这行为无 fEmbedded 的 eBoolean 标签：



下段这段也是含 fEmbedded 的 eBoolean 的标签，“<title>CGICC</title>”为标签 head 的 fEmbedded 内容：
<head>
    <title>CGICC</title>
</head>
```

■ **fDataSpecified**

也是针对 eBoolean 类型标签的，同样在 HTMLElement::**render**()函数的实现中，会发现到差别（对应于对 HTMLElement::**dataSpecified**()的调用）。下列的 a 即为 fDataSpecified 类型的 eBoolean 标签，其中“一见的技术博客”为标签 a 的 Data：

```
<a href="http://aquester.cublog.cn">一见的技术博客</a>
```

■ **代码中的 html()究竟是啥？**

阅读示例代码，可能会有这样一个疑问：html()是如何被调用的？发现没法直接找到名叫 html 的函数。

```
cout << html().set("lang", "en").set("dir", "ltr") << endl;
cout << head() << endl;
cout << title() << "GNU cgicc v" << cgi.getVersion() << title() << endl;
cout << head() << endl;
cout << body() << endl;
cout << h1("Hello, world from GNU cgicc") << endl;
cout << body() << html();
```

上述调用中的 html()、head()、title()、h1()和 body()等，实际都是调用类 **HTMLBooleanElement** 的构造函数，演变成调用 **HTMLElement**::**render**(std::ostream& out)。

流函数的定义为：

```
std::ostream& cgicc::operator <<(std::ostream& out, const cgicc::MStreamable& obj)
{
    obj.render(out);
    return out;
}
```

从流函数的定义不难看出，实际上调用的是 **render**()。

在 HTMLClasses.h 文件中，定义了 html、body 等类（位于 cgicc 名字空间内），但是有些隐晦，直接看不出来：

```
105   BOOLEAN_ELEMENT (html,       "html");       // HTML document
106   BOOLEAN_ELEMENT (head,       "head");       // document head
107   BOOLEAN_ELEMENT (title,      "title");      // document title
108   ATOMIC_ELEMENT  (meta,       "meta");       // meta data
109   BOOLEAN_ELEMENT (style,      "style");      // style sheet
110   BOOLEAN_ELEMENT (body,       "body");       // document body
111   BOOLEAN_ELEMENT (div,        "div");        // block-level grouping
112   BOOLEAN_ELEMENT (span,       "span");       // inline grouping
113   BOOLEAN_ELEMENT (h1,         "h1");         // level 1 heading
114   BOOLEAN_ELEMENT (h2,         "h2");         // level 2 heading
```

翻译一下，以便容易看懂这个过程，先看相关的宏定义：

1) 宏 BOOLEAN_ELEMENT

```
#define BOOLEAN_ELEMENT(name, tag) \
    TAG(name, tag); typedef HTMLBooleanElement<name##Tag> name
```

2) 宏 TAG

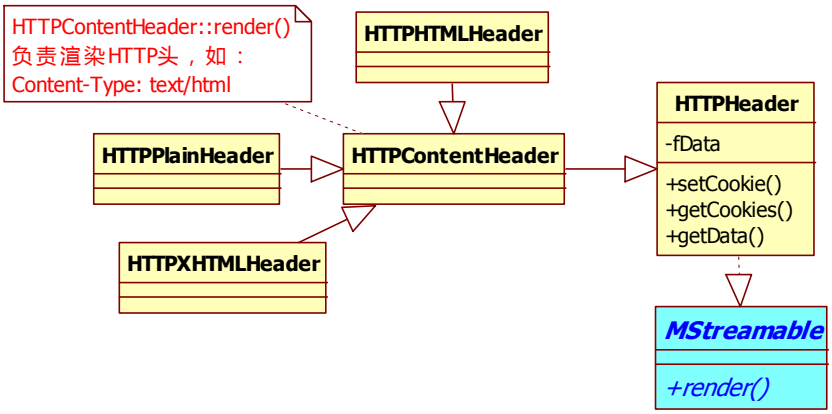
```
// 注意区分下面的 tag 和 Tag
#define TAG(name, tag) \
    class name##Tag \
    {
    public:
        inline static const char* getName()
        {
            return tag; // 注意不是 Tag，而是 tag
        }
    } // 注意，这里并没有加分号
```

现在来看 HTMLClasses.h 文件中定义的 **BOOLEAN_ELEMENT**(html, "html");，宏展开后，变成：

```
class htmlTag
{
public:
    inline static const char* getName()
    {
        return "html";
    }
};

typedef HTMLBooleanElement<htmlTag> html; // html 是不是就是一个类了？
```

html()怎么来的清楚了，还有一个疑问：对于：cout << html()，怎么知道是输出<html>还是</html>的？这个逻辑是在函数 HTMLElement::**render**(std::ostream& out)中完成的。

7.1.HTTPContentHeader



HTTPContentHeader 负责输出 HTTP 头中的“**Content-Type:**”，看它的渲染函数 reader()实现：

```
void cgicc::HTTPContentHeader::render(std::ostream& out) const
{
    out << "Content-Type: " << getData() << std::endl;

    std::vector<HTTPCookie>::const_iterator iter;
    for (iter = getCookies().begin(); iter != getCookies().end(); ++iter)
    {
        out << *iter << std::endl;
    }

    out << std::endl;
}
```

其中，子类 HTTP**HTML**Header 的 getData()返回“**text/html**”，子类 HTTP**Plain**Header 的 getData() 返回“**text/plain**”，子类 HTTP**XHTML**Header 的 getData() 返回“**application/xhtml+xml**”。

7.2.HTMLElement::render()函数

```
void cgicc::HTMLElement::render(std::ostream& out) const
{
    if (eBoolean == getType() && false == dataSpecified())
    {
        if (0 == fEmbedded) /* no embedded elements */
        {
            // 切换：用来控制是输入开始标签，还是关闭标签
```

```
        // HTMLBooleanElement::sState 为类静态数据成员，
        // swapState()的作用就是用来切换它的值。
        swapState();

        /* getState() == true ===> element is active */
        if (true == getState())
        {
            // 输出开始标签，
            out << '<' << getName();

            // 开始标签是可能包含属性的，
            // 如：<a href="http://aquester.cublog.cn">，
            // 这里的 href 即为标签<a>的属性
            if (0 != fAttributes)
            {
                out << ' '; // 属性间使用一个空格分开
                fAttributes->render(out);
            }

            out << '>';
        }
        else
        {
            // 输出关闭标签，如：</head>
            out << "</" << getName() << '>';
        }
    }
    else /* embedded elements present */
    {
        // 被嵌入的（embedded）的内容总是整体一次性输出，
        // 而不是区分其状态值 HTMLBooleanElement::sState
        out << '<' << getName();

        /* render attributes, if present */
        if (0 != fAttributes)
        {
            out << ' ';
            fAttributes->render(out);
        }

        out << '>';
        fEmbedded->render(out);

        // 输出关闭标签，如：</head>
```

```
            out << "</" << getName() << '>';
        }
    }
    else /* For non-boolean elements */
    {
        if (eAtomic == getType())
        {
            out << '<' << getName();
            if (0 != fAttributes)
            {
                out << ' ';
                fAttributes->render(out);
            }

            // eAtomic 类型的标签
            out << " />";
        }
        else
        {
            out << '<' << getName();
            if (0 != fAttributes)
            {
                out << ' ';
                fAttributes->render(out);
            }
            out << '>';

            if (0 != fEmbedded)
            {
                fEmbedded->render(out);
            }
            else
            {
                // 输出数据，
                // 如<a href="http://www.gnu.org/software/cgicc">CGICC</a>
                // 中的 CGICC
                out << getData();
            }

            // 输出关闭标签，如：</head>
            out << "</" << getName() << '>';
        }
    }
}
```

8. 问题？

1) 问题 1：怎么取得不在 CgiEnvironment 支持范围内的环境变量值？
答：可直接调用 C 库函数 getenv()取值。