

webcppd 应用服务器 使用手册

admin@webcpp.net

2016 年 11 月 21 日

目录

1 概述	3
2 框架	3
3 安装	7
4 配置	8
4.1 核心配置	9
4.2 IP 黑名单配置	12
4.3 路由配置	12
5 INFO 组件	13
6 模板	16
6.1 面向编译	17
6.2 面向运行	17
7 缓存	20
8 根视图	22
9 分离前端要素	25
10 外部资源	28

插图

1	Poco 框架结构图	4
2	webcppd 安装与演示	9
3	webcppd 配置信息	16
4	分离前端要素演示	27

表格

1	常见 C/C++ Web 开发框架	3
---	-----------------------------	---

源代码

1	hello.hpp	3
2	hello.cpp	4
3	export.hpp	5
4	export.cpp	5
5	Makefile	5
6	webcppd.properties	9
7	info.hpp	13
8	info.cpp	14
9	info.html	17
10	mustache_info.hpp	18
11	mustache_info.cpp	19
12	cache_mustache_info.hpp	20
13	cache_mustache_info.cpp	21
14	root_view_info.hpp	23
15	root_view_info.cpp	24
16	info_css.html	25
17	demo.css	26
18	JS 要素	26
19	demo.js	27

摘要

webcppd 是一款用 C++ 语言写成的应用服务器软件。它被设计为 C++ 动态库容器，用来加载表现为 C++ 动态库的 Web 组件。通过从 Web 组件获得事先实现的类，webcppd 能够以优异的性能对 HTTP 请求做出合乎预期的响应。

1 概述

webcppd 是为了把 C++ 语言变成一种 Web 开发语言而准备的。利用 webcppd, C++ 语言能够轻松地与 HTML、JS、CSS 打成一片, 无障碍地融入到 Web 开发的实践当中。

因而, 你能够把 C++ 语言当作是更好、更快的 PHP 语言来对待。

项目	网址	语言
tntnet	http://www.tntnet.org/	C++
kore	https://kore.io/	C/C++
crow	https://github.com/ipkn/crow	C++
treefrog	http://www.treefrogframework.org/	C++
cppcms	http://cppcms.com	C++
nxweb	http://nxweb.org/	C
webcppd	https://github.com/webcpp/webcppd	C++

表 1: 常见 C/C++ Web 开发框架

表1列出了 7 种可使用 C/C++ 语言进行 Web 开发的框架, 包括 webcppd 在内。它们都具有自己的独特之处。有兴趣的读者可以关注它们, 根据自己的喜好并权衡多方面因素, 作出最合适的开发选择。

2 框架

webcppd 是用 C++ 语言实现的, 基于著名的 C++ 编程框架 Poco, 该框架的结构如图1。

因此, webcppd 约定的动态库写法, 必须是相容于 Poco 的写法。

webcppd 约定每一个被调用的类都必须是 `Poco::Net::HTTPRequestHandler` 的子类, 该子类必须实现 `handleRequest` 方法。例如将被调用的类是 `hello`:

源代码 1: hello.hpp

```
1 #ifndef HELLO_HPP
2 #define HELLO_HPP
3
4 #include <Poco/Net/HTTPRequestHandler.h>
5 #include <Poco/Net/HTTPServerRequest.h>
6 #include <Poco/Net/HTTPServerResponse.h>
7
8 namespace webcppd {
9
```

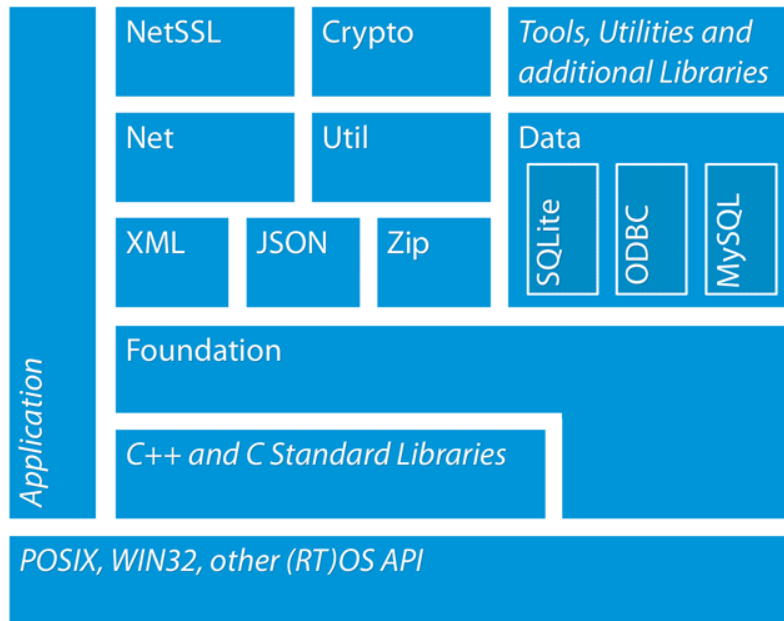


图 1: Poco 框架结构图

```

10     class hello : public Poco::Net::HTTPRequestHandler {
11     public:
12         virtual void handleRequest(Poco::Net::HTTPServerRequest& request,
13                                   Poco::Net::HTTPServerResponse& response);
14     };
15 }
16 #endif /* HELLO_HPP */

```

源代码 2: hello.cpp

```

1  #include "hello.hpp"
2
3  namespace webcppd {
4
5      void hello::handleRequest(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse&
6                               response) {
7          response.setContentType("text/plain;charset=utf-8");
8          response.send() << "Hello world.";
9      }
10 }

```

这是第一步。第二步是把该类按照 Poco 约定的方式包装到动态库中。为此，需要这样：

源代码 3: export.hpp

```
1  #ifndef EXPORT_HPP
2  #define EXPORT_HPP
3
4  #include "hello/hello.hpp"
5  #include "info/info.hpp"
6  #include "info/mustache_info.hpp"
7  #include "info/cache_mustache_info.hpp"
8  #include "root_view/root_view_info.hpp"
9  #include "captcha/captcha.hpp"
10 #include "qrcode/qrcode.hpp"
11 #include "uploadfile/uploadfile.hpp"
12 #include "mysql/mysql.hpp"
13
14 #endif /* EXPORT_HPP */
```

源代码 4: export.cpp

```
1  #include <Poco/ClassLibrary.h>
2  #include <Poco/Net/HTTPRequestHandler.h>
3
4  #include "export.hpp"
5
6  POCO_BEGIN_MANIFEST(Poco::Net::HTTPRequestHandler)
7
8  POCO_EXPORT_CLASS(webcppd::hello)
9  POCO_EXPORT_CLASS(webcppd::info)
10 POCO_EXPORT_CLASS(webcppd::mustache_info)
11 POCO_EXPORT_CLASS(webcppd::cache_mustache_info)
12 POCO_EXPORT_CLASS(webcppd::root_view_info)
13 POCO_EXPORT_CLASS(webcppd::captcha)
14 POCO_EXPORT_CLASS(webcppd::qrcode)
15 POCO_EXPORT_CLASS(webcppd::uploadfile)
16 POCO_EXPORT_CLASS(webcppd::mysql)
17
18 POCO_END_MANIFEST
```

源码准备好了，现在准备一个 Makefile：

源代码 5: Makefile

```
1  MOD=mod/demo.so
```

```

2
3 MODSRC=$(wildcard *.cpp \
4     hello/*.cpp \
5     info/*.cpp \
6     root_view/*.cpp \
7     captcha/*.cpp \
8     qrencode/*.cpp \
9     uploadfile/*.cpp \
10    mysql/*.cpp \
11    \)
12 MODOBJ=$(patsubst %.cpp,%.o,$(MODSRC))
13
14 CC=g++
15 CXXFLAGS+=-O3 -std=c++11 -fPIC -Wall -I./include `pkg-config --cflags opencv cryptopp`
16 LIBS+=-lPocoDataMySQL -lPocoData -lPocoJSON -lPocoNet -lPocoUtil -lPocoFoundation
17 LIBS+=`pkg-config --libs opencv cryptopp`
18 LIBS+=-lqrencode
19 LDFLAGS+=-shared
20
21
22 TPLDIR=/var/webcppd/tpl
23
24 all:$(MOD)
25
26 $(MOD):$(MODOBJ)
27     $(CC) -o $@ $^ $(CXXFLAGS) $(LIBS) $(LDFLAGS)
28
29 clean:
30     rm -f $(MODOBJ) $(MOD)
31
32
33 install:
34     install mod/*.so /var/webcppd/mod
35     install etc/route.conf /etc/webcppd
36     mkdir -pv /var/webcppd/www/assets
37     install assets/*.*/var/webcppd/www/assets
38     mkdir -pv /var/webcppd/www/demo
39     install www/*.*/var/webcppd/www/demo
40     mkdir -pv $(TPLDIR)/demo
41     install tpl/demo/info.html $(TPLDIR)/demo
42     install tpl/demo/info_css.html $(TPLDIR)/demo

```

然后 make,make install。如此，一个标准的 Web 组件 demo.so 就开发安装完成了。

3 安装

第2节的所有说明都假定读者已经把 webcppd 应用服务器安装到了自己的电脑中。现在要放弃 页3 这个假设，说明一下 webcppd 应用服务器的安装方法。

其实，webcppd 的安装方法非常简单，分为四步：

第一步 `yum install epel-release`

第二步 `yum install poco-devel`

第三步 `git clone https://github.com/webcpp/webcppd.git`

第四步 `make && sudo make install`

ubuntu 用户安装方法如下：

第一步 `git clone https://github.com/webcpp/webcppd.git`

第二步 `./ubuntu-install-depend.sh`

第三步 `make && sudo make install`

按照上面介绍完成操作之后，若无意外，webcppd 应用服务器已经安装成功了。安装说明如下：

参数配置 /etc/webcppd 目录为参数配置目录，下面包含三个文件

- webcppd.properties 服务器初始化配置
- ipdeny.conf 永久 IP 黑名单配置
- route.conf 路由配置

运行配置 /var/webcppd/(www|log|mod|tpl|cert) 目录为运行配置目录

- www 根目录，存放静态资源
- log 日志目录，存放日志文件
- mod 组件目录，存放 Web 组件即 C++ 动态库
- tpl 模板目录，存放模板文件
- cert 数字证书目录，存放开启 https 所需文件，其中的gencert.sh 脚本可以用于创建自签名的数字证书

pidfile /var/run/webcppd.pid

程序 /usr/local/bin/webcppd

程序符号链接 /usr/bin/webcppd

控制脚本 /usr/bin/webcppd-ctrl.sh 参数: start|stop|uninstall

systemd 控制脚本 /usr/bin/webcppd-service.sh 参数:

启动 start

停止 stop

重启 restart

状态 status

systemd 配置文件 /etc/systemd/system/webcppd.service

开发包头文件集 /usr/local/include/webcppd

既然已经安装成功，就立即体验下吧。

```
1  sudo systemctl start webcppd
2  #or
3  sudo service webcppd start
4  #or
5  sudo webcppd-service.sh start
6  $or
7  sudo webcppd-ctrl.sh start
```

推荐使用 `systemctl` 来控制 `webcppd`。如果需要 `webcppd` 开机自启动，那么需要运行：

```
1  sudo systemctl enable webcppd
```

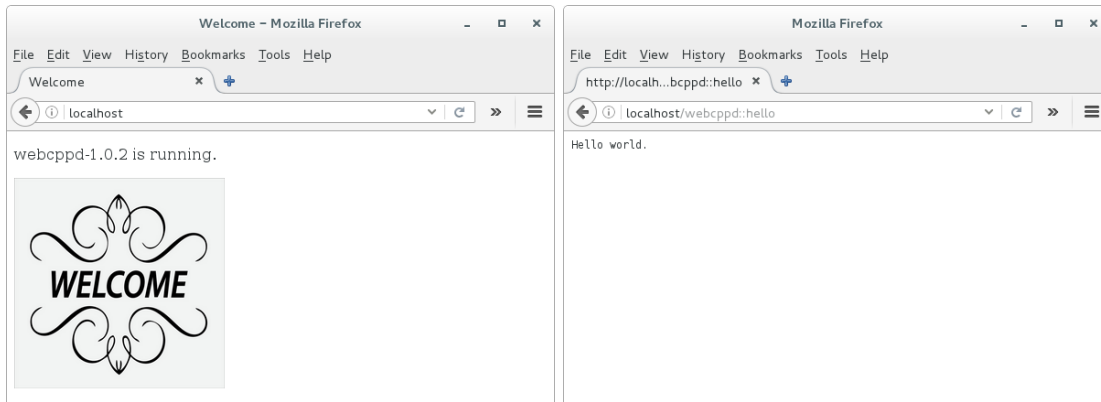
`webcppd` 默认开启 `https` 安全链接，监听 443 端口，用浏览器访问 `https://localhost` 即可。如果不需要安全链接，可通过 `http.enableSSL=false` 关闭安全链接。`http` 链接默认监听 80 端口，所以直接用浏览器访问 `http://localhost` 即可。如果出现如图2a的界面，那么恭喜你：`webcppd` 安装成功了！接着，用浏览器访问 `http://localhost/webcppd::hello`，如果正确返回如图2b，那么再次恭喜你：`demo.so` 组件安装成功了！

4 配置

`webcppd` 既然已经安装好，Web 组件 `demo.so` 也已经安装妥当，现在来看看 `webcppd` 的配置文件。

配置文件都安装在 `/etc/webcppd` 目录下：

- `webcppd.properties` 服务器初始化配置



(a) webcppd 欢迎界面

(b) demo 组件

图 2: webcppd 安装与演示

- ipdeny.conf 永久 IP 黑名单配置
- route.conf 路由配置

webcppd 启动时, 会读入 webcppd.properties 文件以初始化服务器。该文件指定了 ipdeny.conf 和 route.conf 的绝对路径。因此, 一旦服务器正常启动, 以上三个配置文件的所有设置均已生效, 不能修改, 除非修改配置后重启服务器。

4.1 核心配置

webcppd.properties 文件规定了 webcppd 的一切:

源代码 6: webcppd.properties

```

1 #####
2 # HTTP[S] server
3 #####
4
5 # enable https,default : true
6 ;http.enableSSL=true
7 ;http.certPrivateKeyFile=/var/webcppd/cert/server.key
8 ;http.certCertificateFile=/var/webcppd/cert/server.crt
9 ;http.certRootCertificateFile=/var/webcppd/cert/rootCA.pem
10 # client ssl check,default : false
11 ;http.certCheckClient=false
12
13 # enable hotlinking,default: true

```

```

14 ;http.enableHotlinking=true
15 # hotlinking regex match
16 ;http.matchHotlinking=.*
17
18 # bind ip,default: 127.0.0.1
19 ;http.ip = 127.0.0.1
20
21 # Listen port, http default: 80,https default:443
22 ;http.port = 80
23
24 # Maximum requests queue size, default: 1000
25 ;http.maxQueued = 1000
26
27 # Maximum working threads count, default: 1023
28 ;http.maxThreads = 1023
29
30 # Server Name,default:webcppd
31 ;http.serverName=webcppd
32 # Server version, default: webcppd/1.0.3
33 ;http.softwareVersion = webcppd/1.0.3
34
35 #keepAlive ,default true
36 ;http.keepAlive=true
37 #max keepAlive Requests number,default 0,the mean is unlimited
38 ;http.maxKeepAliveRequests=0
39 # keepalive timeout,default 60 second
40 ;http.keepAliveTimeout=60
41
42 # http connect timeout ;default 60 second
43 ;http.timeout=60
44
45
46 # handler Library directory,default: /var/webcppd/mod
47 ;http.libHandlerDir = /var/webcppd/mod
48
49 #ip access check,default
    :ipEnableCheck(false),ipDenyExpire(3600),ipMaxAccessCount(100),ipAccessInterval(30),in seconds
50 ;http.ipEnableCheck=false
51 ;http.ipDenyExpire = 3600
52 ;http.ipMaxAccessCount = 100
53 ;http.ipAccessInterval = 30
54
55 #default /etc/webcppd/ipdeny.conf
56 ;http.ipDenyFile=/etc/webcppd/ipdeny.conf
57

```

```

58 # docroot directory, default: /var/webcppd/www
59 # static file expires, default: 3600s
60 ;http.docroot = /var/webcppd/www
61 ;http.expires = 3600
62
63 #list static directory,default:false
64 ;http.enableIndex=false
65
66 # template directory ,default: /var/webcppd/tpl
67 ;http.tplDirectory = /var/webcppd/tpl
68
69 #upload setting,1mb
70 ;http.uploadMaxSize=1048576
71 ;http.uploadAllowType=image/png|image/jpeg|application/zip
72 ;http.uploadDirectory=/var/webcppd/www/upload
73
74 #logger directory,default:/var/webcppd/log
75 ;http.logDirectory=/var/webcppd/log
76 #logger file size,default:1 MB
77 ;http.logFileSize=1 M
78 #logger file Compress,default:true
79 ;http.logCompress=true
80 #maximum number of archived log files. default:10
81 ;http.logPurgeCount=10
82
83 # proxy server pass real ip,default:proxyUsed(false),proxyServerRealIpHeader(X-Real-IP)
84 ;http.proxyUsed=false
85 ;http.proxyServerRealIpHeader=X-Real-IP
86
87 #route configure
88 ;http.route=/etc/webcppd/route.conf
89
90 # secret-key
91 http.secretKey = a-&$bcDe#%@*#mGhk_A
92
93 # mysql configure
94 mysql.host=localhost
95 mysql.port=3306
96 mysql.user=root
97 mysql.password=123456
98 mysql.db=test
99 mysql.default-character-set=utf8
100 mysql.compress=true
101 mysql.auto-reconnect=true
102

```

```
103 # smtp configure
104 smtp.host=smtp.exmail.qq.com
105 smtp.port=465
106 smtp.user=admin@webcpp.net
107 smtp.password=123456
```

其中的每一项都具有默认值；当然，每一项都是可以自定义的。自定义的方法很简单：把行首的；分号去掉，设置需要的值即可。分号；表示该行取默认值。

需要强调的是，自定义配置必须按照默认值表示的值类型和格式来设置值，否则不能到达需要的效果。

4.2 IP 黑名单配置

核心配置源代码6中关于 IP 黑名单配置的项目有五个，分别是：

页9

```
http.ipEnableCheck=false

http.ipDenyExpire = 3600

http.ipMaxAccessCount = 100

http.ipAccessInterval = 30

http.ipDenyFile=/etc/webcppd/ipdeny.conf
```

webcppd 包含两种 IP 黑名单机制。第一种是动态黑名单机制，它能够自动检测活动的 IP 连接是否为恶意访问或者是否属于机器人行为。该机制通过前四项进行配置。默认情况下没有启用该机制¹，如 http.ipEnableCheck 设定的。要启用该机制，只要将 http.ipEnableCheck 的 false 改为 true 即可。http.ipDenyExpire 表示若活动 IP 被认为是不安全的，则将在接下来的 3600 秒内禁止该 IP 访问服务器。http.ipMaxAccessCount 和 http.ipAccessInterval 规定了发现不安全 IP 的具体方法，也就是：如果一个 IP 在 http.ipAccessInterval 秒以内访问服务器达到 http.ipMaxAccessCount 次，那么服务器就会认为该 IP 是不安全的。

第二种黑名单机制是静态的，由它规定的黑名单 IP 在服务器运行的整个生命期中都不能进行访问。它是由 http.ipDenyFile 指定的。默认情况下，/etc/webcppd/ipdeny.conf 是一个空白文件。如果有需要，逐行将不安全 IP 写入该文件并重启服务器即可。

4.3 路由配置

图2b表示 Web 组件 demo.so 安装妥当，可以通过<http://localhost/webcppd::hello> 来访问该组件。在该访问中，路径/webcppd::hello 即是组件 demo.so 中包含的 webcppd::hello 类。

¹作压力测试时务必禁用该机制。

但是显而易见，这样的 uri 肯定不能令人满意。

通过使用 webcppd 服务器的路由配置，可以很好地解决该问题。核心配置源代码6中关于路由配置的项目如下：

```
http.route=/etc/webcppd/route.conf
```

默认情况下，路由配置文件/etc/webcppd/route.conf 也是一个空白文件。要实现自己的路由规则，只需将相关规则按键值对的方式逐行写入即可。需要注意的是，键与值需要用逗号，或者分号；来分割表示，并且，键必须是正则表达式。比如：

```
1  ^/(hello|demo|test)/?,webcppd::hello
```

上面三条规则表示用路径/hello、/demo、/test 都可以访问类 webcppd::hello。

如果同一个键映射到多个不同的类，比如：

```
1  ^/a/?,webcppd::a
2  ^/a/?,webcppd::b
```

那么，实际上生效的是行号最小的那一条规则。在上面的例子里，有效的就是第一条规则，第二条将被忽略。

如果想服务器忽略某条路由规则，只需在该行行首添加 # 号即可。

修改路由配置后，别忘了重启服务器使之生效。

5 INFO 组件

总的来说，webcppd 应用服务器的配置信息分为三个类目。第一个类目是操作系统信息，第二个类目是服务器程序信息，第三个类目才是webcppd.properties 文件所设定的配置信息。现在，我们来写一个类webcppd::info。当我们用浏览器访问http://localhost/info 时，它能以表格的形式输出 webcppd 服务器运行时所有的配置信息。

源代码 7: info.hpp

```
1  #ifndef INFO_HPP
2  #define INFO_HPP
3
4  #include <Poco/Net/HTTPRequestHandler.h>
5  #include <Poco/Net/HTTPServerRequest.h>
6  #include <Poco/Net/HTTPServerResponse.h>
7
8  namespace webcppd {
9
10     class info : public Poco::Net::HTTPRequestHandler {
```

```

11     void handleRequest(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse&
12         response);
13 };
14 }
15
16 #endif /* INFO_HPP */

```

源代码 8: info.cpp

```

1  #include <Poco/Util/Application.h>
2  #include <Poco/Util/LayeredConfiguration.h>
3  #include <Poco/String.h>
4
5  #include "info.hpp"
6
7  namespace webcppd {
8
9      void info::handleRequest(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse&
10         response) {
11
12         std::string begin = "<html><head><title>webcppd 配置信息</title>%[0]s</head><body>",
13             tableHead = "<table class='gridtable'><caption>%[0]s
14                 配置信息</caption><tr><th>项目</th><th>取值</th></tr>",
15             tableBody,
16             tableFoot = "</table>",
17             end = "</body></html>",
18             style =
19                 "<style type='text/css'>\
20 body{\
21     width:900px;\
22     margin: 40px auto;\
23 }\
24 table.gridtable {\
25     font-family: verdana,arial,sans-serif;\
26     font-size:11px;\
27     color:#333333;\
28     border-width: 1px;\
29     border-color: #666666;\
30     border-collapse: collapse;\
31     margin-left:auto;\
32     margin-right:auto;\
33     margin-bottom:30px;\
34 }\

```

```

33 table.gridtable th {\
34     border-width: 1px;\
35     padding: 8px;\
36     border-style: solid;\
37     border-color: #666666;\
38     background-color: #dedede;\
39     width:300px;\
40 }\
41 table.gridtable td {\
42     border-width: 1px;\
43     padding: 8px;\
44     border-style: solid;\
45     border-color: #666666;\
46     background-color: #ffffff;\
47     width:300px;\
48 }\
49 table.gridtable caption{\
50     font-size:19px;\
51 }\
52 </style>";
53 Poco::Util::Application& app = Poco::Util::Application::instance();
54 Poco::Util::AbstractConfiguration::Keys rootKeys, configKeys, content;
55 app.config().keys(rootKeys);
56
57 for (auto& rootItem : rootKeys) {
58     app.config().keys(rootItem, configKeys);
59     for (auto& confItem : configKeys) {
60         tableBody.append("<tr><td>");
61         tableBody.append(confItem);
62         tableBody.append("</td><td>");
63         tableBody.append(app.config().getString(rootItem + "." + confItem, "none"));
64         tableBody.append("</td></tr>");
65     }
66     content.push_back(Poco::format(tableHead, rootItem) + tableBody + tableFoot);
67     tableBody.clear();
68     configKeys.clear();
69 }
70 for (auto &item : content) {
71     begin.append(item);
72 }
73 response.setContentType("text/html;charset=utf-8");
74 response.send() << Poco::format(begin.append(end), style);
75
76 }
77

```

源代码8的前 52 行设定了页面模板，其中第 16 至 52 行设定了页面样式。从第 53 至 72 行， 页14 是获取配置信息生成真实页面的“业务逻辑”部分。如无意外，`make && make install` 之后，即可重启服务器；之后，用浏览器访问`http://localhost/webcppd::info` 应该可获得如图3的界面。



图 3: webcppd 配置信息

现在，只需添加一行路由规则：

```
1 ^/info/?,webcppd::info
```

到`/etc/webcppd/route.conf` 中并重启服务器，就可以通过路径`http://localhost/info` 进行访问了。

6 模板

`webcppd` 应用服务器能够支持两种模板系统。

6.1 面向编译

webcppd 建立在 Poco 框架之上，故而支持该框架内置的模板系统 cpspc。这是一套把静态文件转换为 C++ 类，从而可以把静态文件转换为动态库 Web 组件的工具。

不推荐使用该系统。

6.2 面向运行

面向运行的模板系统更易于控制。

这里要推荐使用的模板系统是 **mustache**。关于这套模板系统的语法和用法，可参考一般性介绍²。该模板系统的 C++ 实现有不少，这里推荐两个：Mustache³和plustache⁴。前者只是一个头文件，更易于使用。

在第5节的源代码8中，已经出现了页面模板的概念。只不过，当时的页面模板还没有跟业务逻辑分离开来。也就是说，如果不使用面向运行的模板系统，将无法进行前、后端源代码的合理分离。现在，将要引进的 **mustache** 可以很好做到这一点。

下面将用Mustache来重新实现第5节实现的 INFO 组件。

首先页面模板分离出来：

源代码 9: info.html

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>webcppd 配置信息</title>
5     <meta charset="UTF-8">
6     <meta name="viewport" content="width=device-width, initial-scale=1.0">
7     <style type='text/css'>
8       body{
9         width:900px;
10        margin: 40px auto;
11      }
12      table.gridtable {
13        font-family: verdana,arial,sans-serif;
14        font-size:11px;
15        color:#333333;
16        border-width: 1px;
17        border-color: #666666;
18        border-collapse: collapse;
19        margin-left:auto;
```

²<https://mustache.github.io/mustache.5.html>

³<https://github.com/kainjow/Mustache>

⁴<https://github.com/mrtazz/plustache>

```

20         margin-right:auto;
21         margin-bottom:30px;
22     }
23     table.gridtable th {
24         border-width: 1px;
25         padding: 8px;
26         border-style: solid;
27         border-color: #666666;
28         background-color: #dedede;
29         width:300px;
30     }
31     table.gridtable td {
32         border-width: 1px;
33         padding: 8px;
34         border-style: solid;
35         border-color: #666666;
36         background-color: #ffffff;
37         width:300px;
38     }
39     table.gridtable caption{
40         font-size:19px;
41     }
42     </style>
43 </head>
44 <body>
45     {{#tableList}}
46     <table class="gridtable">
47         <caption>{{caption}}</caption>
48         <tr><th>项目</th><th>取值</th></tr>
49         {{#trList}}
50         <tr>
51             <td>{{key}}</td>
52             <td>{{value}}</td>
53         </tr>
54         {{/trList}}
55     </table>
56     {{/tableList}}
57 </body>
58 </html>

```

然后是 C++ 业务逻辑处理类:

源代码 10: mustache_info.hpp

```

1  #ifndef MUSTACHE_INFO_HPP

```

```

2  #define MUSTACHE_INFO_HPP
3
4
5  #include <Poco/Net/HTTPRequestHandler.h>
6  #include <Poco/Net/HTTPServerRequest.h>
7  #include <Poco/Net/HTTPServerResponse.h>
8
9  namespace webcppd {
10
11      class mustache_info : public Poco::Net::HTTPRequestHandler {
12          void handleRequest(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse&
              response);
13
14      };
15  }
16
17
18
19  #endif /* MUSTACHE_INFO_HPP */

```

源代码 11: mustache_info.cpp

```

1  #include <Poco/Util/Application.h>
2  #include <Poco/Util/LayeredConfiguration.h>
3  #include <Poco/String.h>
4  #include <Poco/StreamCopier.h>
5  #include <Poco/FileStream.h>
6
7  #include <webcppd/mustache.hpp>
8
9  #include "mustache_info.hpp"
10
11  namespace webcppd {
12
13      void mustache_info::handleRequest(Poco::Net::HTTPServerRequest& request,
          Poco::Net::HTTPServerResponse& response) {
14          Poco::Util::Application& app = Poco::Util::Application::instance();
15          Poco::Util::AbstractConfiguration::Keys rootKeys, configKeys;
16          app.config().keys(rootKeys);
17          Kainjow::Mustache::Data tableList(Kainjow::Mustache::Data::List());
18          for (auto& rootItem : rootKeys) {
19              Kainjow::Mustache::Data table, trList(Kainjow::Mustache::Data::List());
20              app.config().keys(rootItem, configKeys);
21              for (auto& confItem : configKeys) {

```

```

22         Kainjow::Mustache::Data kv;
23         kv.set("key", confItem);
24         kv.set("value", app.config().getString(rootItem + "." + confItem, "none"));
25         trList.push_back(kv);
26     }
27     table.set("caption", rootItem);
28     table.set("trList", trList);
29     tableList.push_back(table);
30     configKeys.clear();
31 }
32 Poco::FileInputStream IS(app.config().getString("http.tplDirectory", "/var/www/webcppd/tpl")
33     + "/demo/info.html");
34 std::string tpl;
35 Poco::StreamCopier::copyToString(IS, tpl);
36 Kainjow::Mustache engine(tpl);
37
38 response.setContentType("text/html;charset=utf-8");
39 response.setChunkedTransferEncoding(true);
40 engine.render({"tableList", tableList}, response.send());
41 }
42 }

```

源代码11相比于源代码8，已经非常简洁明晰。

页14

最后路由下：

```

1 ~/mustache_info/?,webcppd::mustache_info

```

7 缓存

从程序开发的角度来看模板技术，面向运行比面向编译要有优势；但是，从程序运行的角度来说，面向运行反而只能获得较低的运行速度。因此，使用面向运行的模板技术时，应该考虑使用缓存技术提升运行速度。

Poco 框架提供了非常好的缓存编程接口，可以非常方便地使用到 Web 组件的开发当中。

源代码 12: cache_mustache_info.hpp

```

1 #ifndef CACHE_MUSTACHE_INFO_HPP
2 #define CACHE_MUSTACHE_INFO_HPP
3
4 #include <Poco/Net/HTTPRequestHandler.h>
5 #include <Poco/Net/HTTPServerRequest.h>

```

```

6  #include <Poco/Net/HTTPServerResponse.h>
7  #include <Poco/ExpireCache.h>
8  #include <string>
9
10 namespace webcppd {
11
12     class cache_mustache_info : public Poco::Net::HTTPRequestHandler {
13     public:
14         void handleRequest(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse&
            response);
15     private:
16         static Poco::ExpireCache<std::string, std::string> cache;
17     };
18 }
19
20
21 #endif /* CACHE_MUSTACHE_INFO_HPP */

```

源代码 13: cache__mustache__info.cpp

```

1  #include <Poco/Util/Application.h>
2  #include <Poco/Util/LayeredConfiguration.h>
3  #include <Poco/String.h>
4  #include <Poco/StreamCopier.h>
5  #include <Poco/FileStream.h>
6
7  #include <webcppd/mustache.hpp>
8
9  #include "cache_mustache_info.hpp"
10
11 namespace webcppd {
12
13     Poco::ExpireCache<std::string, std::string> cache_mustache_info::cache(600000);
14
15     void cache_mustache_info::handleRequest(Poco::Net::HTTPServerRequest& request,
        Poco::Net::HTTPServerResponse& response) {
16         std::string cacheKey("cache_mustache_info");
17         if (cache_mustache_info::cache.has(cacheKey)) {
18             response.send() << *cache_mustache_info::cache.get(cacheKey);
19             return;
20         }
21         Poco::Util::Application& app = Poco::Util::Application::instance();
22         Poco::Util::AbstractConfiguration::Keys rootKeys, configKeys;
23         app.config().keys(rootKeys);

```

```

24     Kainjow::Mustache::Data tableList(Kainjow::Mustache::Data::List());
25     for (auto& rootItem : rootKeys) {
26         Kainjow::Mustache::Data table, trList(Kainjow::Mustache::Data::List());
27         app.config().keys(rootItem, configKeys);
28         for (auto& confItem : configKeys) {
29             Kainjow::Mustache::Data kv;
30             kv.set("key", confItem);
31             kv.set("value", app.config().getString(rootItem + "." + confItem, "none"));
32             trList.push_back(kv);
33         }
34         table.set("caption", rootItem);
35         table.set("trList", trList);
36         tableList.push_back(table);
37         configKeys.clear();
38     }
39     Poco::FileInputStream IS(app.config().getString("http.tplDirectory", "/var/www/webcppd/tpl")
40                             + "/demo/info.html");
41     std::string tpl;
42     Poco::StreamCopier::copyToString(IS, tpl);
43     Kainjow::Mustache engine(tpl);
44
45     response.setContentType("text/html;charset=utf-8");
46     response.setChunkedTransferEncoding(true);
47     cache_mustache_info::cache.add(cacheKey, engine.render({"tableList", tableList}));
48     response.send() << *cache_mustache_info::cache.get(cacheKey);
49 }
50 }

```

压力测试清楚地表明：使用缓存技术后，面向运行的模板技术不仅速度大幅度提升，而且要大大高于面向编译的模板技术。因此，为每一个 Web 组件提供一个缓存器是非常必要的，不管你是否需要真的使用它！

Poco 框架提供了多种缓存机制，源代码12中 `Poco::ExpireCache` 只是其中一种。究竟哪一种缓存器更适合业务逻辑，这是开发者有时候需要根据业务属性仔细斟酌的问题。

8 根视图

对 webcppd 应用服务器来说，每一个 Web 组件都是一个 `Poco::Net::HTTPRequestHandler` 子类。如第7节所述，每一个 Web 组件都应该有一个缓存器。因此，不妨构造基于 `Poco::Net::HTTPRequestHandler` 一个根视图类，为所有 Web 组件提供统一的缓存器。见开发包头文件 `/usr/local/include/webcppd/root_view.hpp`。如此，所有 Web 组件只要基于 `webcppd::root_view`

进行构造，就能轻易使用根缓存器和根会话管理器。

根视图略微改变了组件开发的接口，根据客户端请求方法把原来统一的 `handleRequest` 接口分为四个接口：

GET: `do_get`

POST: `do_post`

PUT: `do_put`

DELETE: `do_delete`

开发时，只需根据需要实现即可。

比如，第7节中实现的 `webcppd::cache_mustache_info`，可以用根视图重新实现如下。

源代码 14: `root_view_info.hpp`

```
1  #ifndef ROOT_VIEW_INFO_HPP
2  #define ROOT_VIEW_INFO_HPP
3
4  #include <webcppd/root_view.hpp>
5
6  namespace webcppd {
7
8      class root_view_info : public root_view {
9          void do_get(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse& response);
10
11          void do_post(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse& response) {
12              this->error(request, response);
13          }
14
15          void do_put(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse& response) {
16              this->error(request, response);
17          }
18
19          void do_delete(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse&
20                      response) {
21              this->error(request, response);
22          }
23
24
25      };
26  }
27
28  #endif /* ROOT_VIEW_INFO_HPP */
```

源代码 15: root_view_info.cpp

```
1  #include <Poco/Util/Application.h>
2  #include <Poco/Util/LayeredConfiguration.h>
3  #include <Poco/String.h>
4  #include <Poco/StreamCopier.h>
5  #include <Poco/FileStream.h>
6  #include <Poco/ExpireCache.h>
7
8  #include <webcppd/mustache.hpp>
9
10 #include "root_view_info.hpp"
11
12 namespace webcppd {
13
14     void root_view_info::do_get(Poco::Net::HTTPServerRequest& request,
15                                Poco::Net::HTTPServerResponse& response) {
16         std::string cacheKey("cache_mustache_info");
17         if (root_view_info::root_cache().has(cacheKey)) {
18             response.send() << *root_view_info::root_cache().get(cacheKey);
19             return;
20         }
21         Poco::Util::Application& app = Poco::Util::Application::instance();
22         Poco::Util::AbstractConfiguration::Keys rootKeys, configKeys;
23         app.config().keys(rootKeys);
24         Kainjow::Mustache::Data tableList(Kainjow::Mustache::Data::List());
25         for (auto& rootItem : rootKeys) {
26             Kainjow::Mustache::Data table, trList(Kainjow::Mustache::Data::List());
27             app.config().keys(rootItem, configKeys);
28             for (auto& confItem : configKeys) {
29                 Kainjow::Mustache::Data kv;
30                 kv.set("key", confItem);
31                 kv.set("value", app.config().getString(rootItem + "." + confItem, "none"));
32                 trList.push_back(kv);
33             }
34             table.set("caption", rootItem);
35             table.set("trList", trList);
36             tableList.push_back(table);
37             configKeys.clear();
38         }
39         Poco::FileInputStream IS(app.config().getString("http.tplDirectory", "/var/www/webcppd/tpl")
40                                + "/demo/info.html");
41         std::string tpl;
42         Poco::StreamCopier::copyToString(IS, tpl);
43         Kainjow::Mustache engine(tpl);
```



```

42
43     response.setContentType("text/html;charset=utf-8");
44     response.setChunkedTransferEncoding(true);
45     root_view_info::root_cache().add(cacheKey, engine.render({"tableList", tableList}));
46     response.send() << *root_view_info::root_cache().get(cacheKey);
47 }
48
49 }

```

9 分离前端要素

在第6节中，利用面向运行的模板技术，可以将后端业务逻辑与前端页面模板分离开来。这是非常重要的一步。已经表明，只要合理利用缓存器，就可以使得面向运行的模板技术在运行速度上大大超越面向编译的模板技术；与此同时，还能极大地提供开发的效率。

现在要考虑另一个问题。单纯从前端的角度来看，即使是面向运行的模板技术，也还有很大的提升空间。观察源代码9可知，样式表 CSS 元素是可以从整个页面模板中分离出来的。

页17

源代码 16: info_css.html

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>webcppd 配置信息</title>
5     <meta charset="UTF-8">
6     <meta name="viewport" content="width=device-width, initial-scale=1.0">
7     <link rel="stylesheet" type="text/css" href="/assets/demo.css"/>
8   </head>
9   <body>
10     {{#tableList}}
11     <table class="gridtable">
12       <caption>{{caption}}</caption>
13       <tr><th>项目</th><th>取值</th></tr>
14       {{#trList}}
15       <tr>
16         <td>{{key}}</td>
17         <td>{{value}}</td>
18       </tr>
19       {{/trList}}
20     </table>
21     {{/tableList}}
22     <script src="//cdn.bootcss.com/jquery/3.1.1/jquery.min.js"></script>
23     <script src='/assets/demo.js'></script>

```

```
24     </body>
25 </html>
```

源代码 17: demo.css

```
1  body{
2      width:900px;
3      margin: 40px auto;
4  }
5  table.gridtable {
6      font-family: verdana,arial,sans-serif;
7      font-size:11px;
8      color:#333333;
9      border-width: 1px;
10     border-color: #666666;
11     border-collapse: collapse;
12     margin-left:auto;
13     margin-right:auto;
14     margin-bottom:30px;
15 }
16 table.gridtable th {
17     border-width: 1px;
18     padding: 8px;
19     border-style: solid;
20     border-color: #666666;
21     background-color: #dedede;
22     width:300px;
23 }
24 table.gridtable td {
25     border-width: 1px;
26     padding: 8px;
27     border-style: solid;
28     border-color: #666666;
29     background-color: #ffffff;
30     width:300px;
31 }
32 table.gridtable caption{
33     font-size:19px;
34 }
```

前端要素的另一个大项是 JS。虽然页面模板9没有包含这一项，但是完全可以假定它包含 JS 要素。因此，这一项要素也需要分离出来。把下面两行代码放置在源代码9中 head 或者 body 结束标签之前。

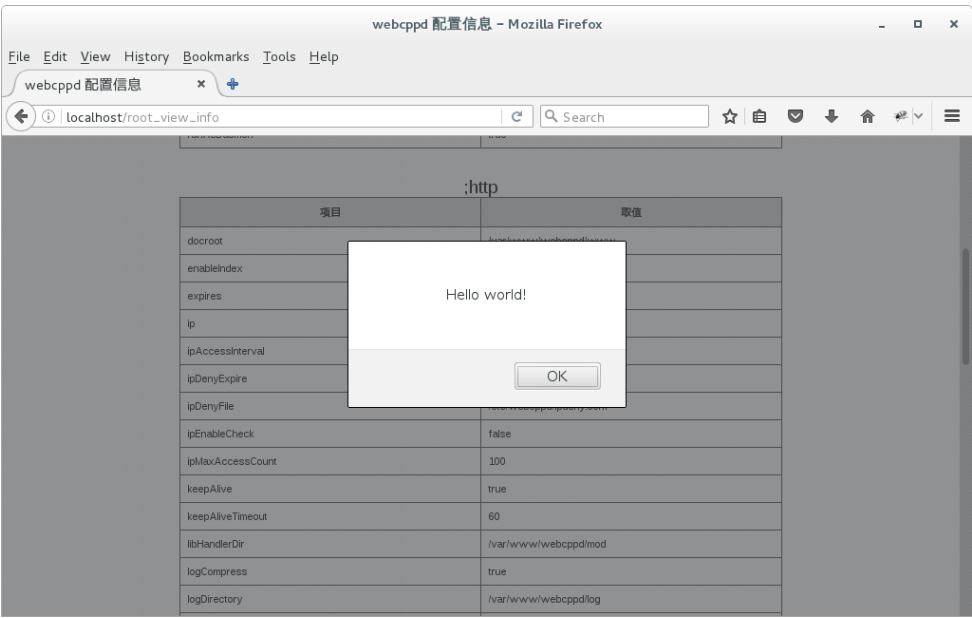


图 4: 分离前端要素演示

源代码 18: JS 要素

```
1 <script src="//cdn.bootcss.com/jquery/3.1.1/jquery.min.js"></script>
2 <script src='/assets/demo.js'></script>
```

源代码 19: demo.js

```
1 $(document).ready(function(){
2     alert('Hello world!');
3 });
```

这样一来，标准的、完整的前、后端分离就完成了⁵。只要把源代码15第 37 行中页面模板的默认值 页24 改为/demo/info_css.html，然后重新编译、安装 demo.so 组件并重启服务器：

```
1 make && sudo make install && sudo systemctl restart webcppd
```

就可以让该组件以预期的方式运行起来，如图4所示。

⁵源代码18第 1 行使用 jQuery 不过是最常见的做法，读者完全可以自行决定。

10 外部资源

外部资源不仅仅指后端 C/C++ 资源，也包括前端 HTML,JS,CSS 资源。付费资源当然很多，免费资源亦复不少。运用前文阐述的前、后端分离技术以及前端要素分离技术，webcppd 应用服务器可以很自然地把这些各司其职的资源統合在同一个 Web 组件当中。

外部后端 C/C++ 资源对于传统的 C/C++ 开发者来说，自然有着非常清晰的概念。怎样引入它们也不在话下。而对于外部前端资源如 HTML 模板和 JS,CSS 框架，传统的 C/C++ 开发者就不甚熟知了。像Bootstrap,jQuery和各种 Bootstrap 模板和 jQuery 插件，都是最为人津津乐道的前端资源。实际上，webcppd 应用服务器要求开发者对于前、后端资源都要有相当的了解。