webcppd 应用服务器
使用手册

admin@webcpp.net

2016 年 10 月 30 日

目录

插图

表格

源代码

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
```

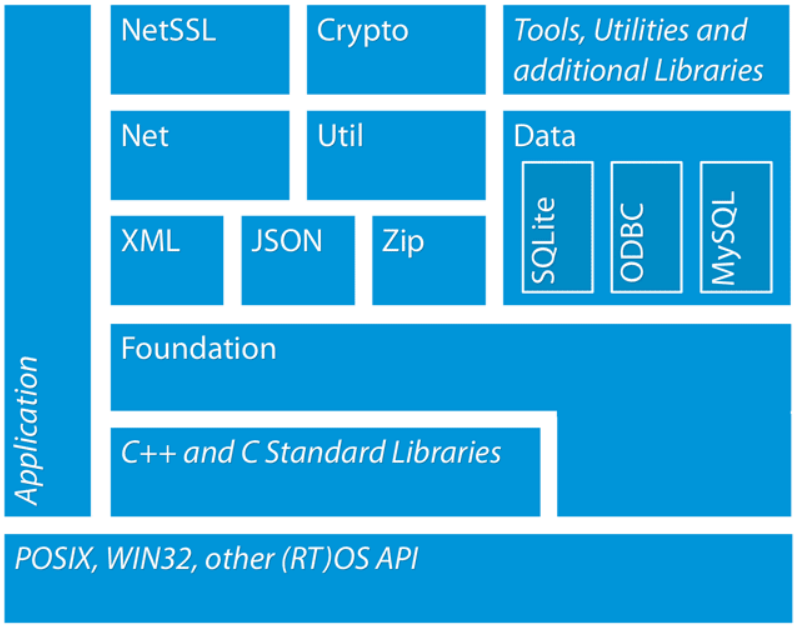


图 1: Poco 框架结构图

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

源代码 2: hello.cpp

```
1  #include "hello.hpp"
2
3  namespace webcppd {
4
```

```
    void hello::handleRequest(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse&
        response) {
        response.setContentType("text/plain;charset=utf-8");
        response.send() << "Hello world.";
    }
}
```

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

```
#ifndef EXPORT_HPP
#define EXPORT_HPP

#include "hello/hello.hpp"
#include "info/info.hpp"
#include "info/mustache_info.hpp"
#include "info/cache_mustache_info.hpp"
#include "root_view/root_view_info.hpp"

#endif /* EXPORT_HPP */
```

源代码 4: export.cpp

```
#include <Poco/ClassLibrary.h>
#include <Poco/Net/HTTPRequestHandler.h>

#include "export.hpp"

POCO_BEGIN_MANIFEST(Poco::Net::HTTPRequestHandler)

POCO_EXPORT_CLASS(webcppd::hello)
POCO_EXPORT_CLASS(webcppd::info)
POCO_EXPORT_CLASS(webcppd::mustache_info)
POCO_EXPORT_CLASS(webcppd::cache_mustache_info)
POCO_EXPORT_CLASS(webcppd::root_view_info)

POCO_END_MANIFEST
```

源码准备好了，现在准备一个 `Makefile`：

源代码 5: Makefile

```
MOD=mod/demo.so

```

```
MODSRC=$(wildcard *.cpp hello/*.cpp info/*.cpp root_view/*.cpp)
MODOBJ=$(patsubst %.cpp,%.o,$(MODSRC))

CC=g++
CXXFLAGS+=-O3 -std=c++11 -fPIC -Wall
LIBS+=-lPocoNet -lPocoUtil -lPocoFoundation
LDFLAGS+=-shared


TPLDIR=/var/www/webcppd/tpl

all:$(MOD)

$(MOD):$(MODOBJ)
    $(CC) $(CXXFLAGS) $(LIBS) $(LDFLAGS) -o $@ $^

clean:
    rm -f $(MODOBJ) $(MOD)


install:
    install mod/*.so /var/www/webcppd/mod
    install etc/route.conf /etc/webcppd
    mkdir -pv /var/www/webcppd/www/assets
    install assets/*.* /var/www/webcppd/www/assets
    mkdir -pv $(TPLDIR)/demo
    install tpl/demo/info.html $(TPLDIR)/demo
    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 && make install

按照上面四步完成操作之后，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/www/webcppd/(www|log|mod|tpl) 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

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

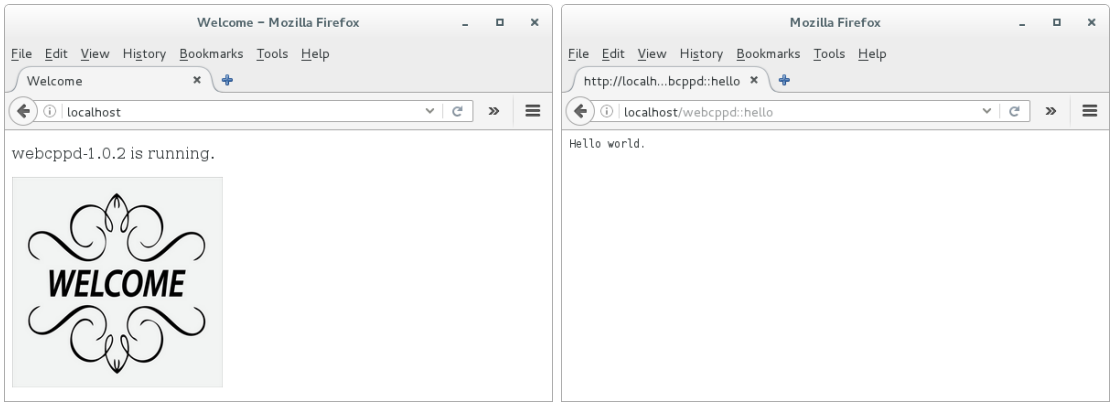
既然已经安装成功，就立即体验下吧。

```
sudo systemctl start webcppd
#or
sudo service webcppd start
#or
sudo webcppd-service.sh start
$or
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：

```
sudo systemctl enable webcppd
```

webcppd 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



(a) webcppd 欢迎界面　　(b) demo 组件

图 2: webcppd 安装与演示

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 目录下面：

- 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

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

```
##############
# HTTP server
##############

# bind ip,default: 127.0.0.1
;http.ip = 127.0.0.1

# Listen port, default: 80
;http.port = 80

# Maximum requests queue size, default: 1000
;http.maxQueued = 1000

# Maximum working threads count, default: 1023
;http.maxThreads = 1023

# Software version (Server HTTP header), default: webcppd/1.0.2
;http.softwareVersion = webcppd/1.0.2

#keepAlive ,default true
;http.keepAlive=true
#max keepAlive Requests number;default 0,the mean is unlimited
;http.maxKeepAliveRequests=0
# keepalive timeout;default 60 second
;http.keepAliveTimeout=60

# http connect timeout ;default 60 second
;http.timeout=60


# handler Library directory,default: /var/www/webcppd/mod
;http.libHandlerDir = /var/www/webcppd/mod

#ip access check,default
    :ipEnableCheck(false),ipDenyExpire(3600),ipMaxAccessCount(100),ipAccessInterval(30),in seconds
;http.ipEnableCheck=false
;http.ipDenyExpire = 3600
;http.ipMaxAccessCount = 100
;http.ipAccessInterval = 30

```

```
#default /etc/webcppd/ipdeny.conf
;http.ipDenyFile=/etc/webcppd/ipdeny.conf

# docroot directory, default: /var/www/webcppd/www
# static file expires, default: 3600s
;http.docroot = /var/www/webcppd/www
;http.expires = 3600

#list static directory,default:true
;http.enableIndex=true

# template directory ,default: /var/www/webcppd/tpl
;http.tplDirectory = /var/www/webcppd/tpl

#upload setting
;http.uploadMaxSize=51200
;http.uploadAllowType=image/png|image/jpeg|application/zip
;http.uploadDirectory=/var/www/webcppd/www/upload

#logger directory,default:/var/www/webcppd/log
;http.logDirectory=/var/www/webcppd/log
#logger file size,default:1 MB
;http.logFileSize=1 M
#logger file Compress,default:true
;http.logCompress=true
#maximum number of archived log files. default:10
;http.logPurgeCount=10

# proxy server pass real ip,default:proxyUsed(false),proxyServerRealIpHeader(X-Real-IP)
;http.proxyUsed=false
;http.proxyServerRealIpHeader=X-Real-IP

#route configure
;http.route=/etc/webcppd/route.conf

# secret-key : default
;http.secretKey = a-&$bcDe#%@*#mGhk_A
```

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，分别是第 35，36，37，38，41 行。 页9

第 35 行 `http.ipEnableCheck=false`

第 36 行 `http.ipDenyExpire = 3600`

第 37 行 `http.ipMaxAccessCount = 100`

第 38 行 `http.ipAccessInterval = 30`

第 41 行 `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]，如第 35 行所设定的。要启用该机制，只要将第 35 行的 false 改为 true 即可。第 36 行表示若活动 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。第 37，38 行规定了发现不安全 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。它是由第 41 行指定的。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。但是显而易见，这样的 uri 肯定不能令人满意。 页8

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是第 72 行。 页9

第 72 行 `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。比如:

[1]作压力测试时务必禁用该机制。

```
demo,webcppd::hello
test,webcppd::hello
hello;webcppd::hello
```

上面三条规则表示用路径/demo、/test、/hello 都可以访问类 webcppd::hello。如果觉得写三条太麻烦，也可以只写一条：

```
demo,test,hello,webcppd::hello
```

这种写法的要点是必须把值放在最后。至于风格符号用逗号还是分号，抑或混合使用，则完全是任意的。

另外，如果同一个键映射到多个不同的类，比如:

```
a1,webcppd::a
a2,webcppd::a
```

那么，实际上生效的是行号最大的那一条规则。在上面的例子里，有效的就是第二条规则，因为第一条被覆盖了。

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，第二个类目时服务器程序信息，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

```
#ifndef INFO_HPP
#define INFO_HPP

#include <Poco/Net/HTTPRequestHandler.h>
#include <Poco/Net/HTTPServerRequest.h>
#include <Poco/Net/HTTPServerResponse.h>

namespace webcppd {

   class info : public Poco::Net::HTTPRequestHandler {
      void handleRequest(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse&
          response);
```

```

    };
}

#endif /* INFO_HPP */
```

源代码 8: info.cpp

```
#include <Poco/Util/Application.h>
#include <Poco/Util/LayeredConfiguration.h>
#include <Poco/String.h>

#include "info.hpp"

namespace webcppd {

    void info::handleRequest(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse&
         response) {

        std::string begin = "<html><head><title>webcppd 配置信息</title>%[0]s</head><body>",
               tableHead = "<table class='gridtable'><caption>%[0]s
                    配置信息</caption><tr><th>项目</th><th>取值</th></tr>",
               tableBody,
               tableFoot = "</table>",
               end = "</body></html>",
               style =
               "<style type='text/css'>\
body{\
       width:900px;\
       margin: 40px auto;\
}\
table.gridtable {\
    font-family: verdana,arial,sans-serif;\
    font-size:11px;\
    color:#333333;\
    border-width: 1px;\
    border-color: #666666;\
    border-collapse: collapse;\
        margin-left:auto;\
        margin-right:auto;\
        margin-bottom:30px;\
}\
table.gridtable th {\
    border-width: 1px;\
```

```
    padding: 8px;\
    border-style: solid;\
    border-color: #666666;\
    background-color: #dedede;\
        width:300px;\
}\
table.gridtable td {\
    border-width: 1px;\
    padding: 8px;\
    border-style: solid;\
    border-color: #666666;\
    background-color: #ffffff;\
        width:300px;\
}\
table.gridtable caption{\
        font-size:19px;\
}\
</style>";
        Poco::Util::Application& app = Poco::Util::Application::instance();
        Poco::Util::AbstractConfiguration::Keys rootKeys, configKeys, content;
        app.config().keys(rootKeys);

        for (auto& rootItem : rootKeys) {
            app.config().keys(rootItem, configKeys);
            for (auto& confItem : configKeys) {
                tableBody.append("<tr><td>");
                tableBody.append(confItem);
                tableBody.append("</td><td>");
                tableBody.append(app.config().getString(rootItem + "." + confItem, "none"));
                tableBody.append("</td></tr>");
            }
            content.push_back(Poco::format(tableHead, rootItem) + tableBody + tableFoot);
            tableBody.clear();
            configKeys.clear();
        }
        for (auto &item : content) {
            begin.append(item);
        }
        response.setContentType("text/html;charset=utf-8");
        response.send() << Poco::format(begin.append(end), style);

    }

}
```

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13
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：Mustache[3]和plustache[4]。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。 页13

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

```
<!DOCTYPE html>
<html>
   <head>
      <title>webcppd 配置信息</title>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
      <style type='text/css'>
         body{
            width:900px;
            margin: 40px auto;
         }
         table.gridtable {
            font-family: verdana,arial,sans-serif;
            font-size:11px;
            color:#333333;
            border-width: 1px;
            border-color: #666666;
            border-collapse: collapse;
            margin-left:auto;
            margin-right:auto;
            margin-bottom:30px;
         }
         table.gridtable th {
            border-width: 1px;
            padding: 8px;
            border-style: solid;
```

[2]https://mustache.github.io/mustache.5.html

[3]https://github.com/kainjow/Mustache

[4]https://github.com/mrtazz/plustache

```
            border-color: #666666;
            background-color: #dedede;
            width:300px;
        }
        table.gridtable td {
            border-width: 1px;
            padding: 8px;
            border-style: solid;
            border-color: #666666;
            background-color: #ffffff;
            width:300px;
        }
        table.gridtable caption{
            font-size:19px;
        }
    </style>
</head>
<body>
    {{#tableList}}
    <table class="gridtable">
        <caption>{{caption}}</caption>
        <tr><th>项目</th><th>取值</th></tr>
        {{#trList}}
        <tr>
            <td>{{key}}</td>
            <td>{{value}}</td>
        </tr>
        {{/trList}}
    </table>
    {{/tableList}}
</body>
</html>
```

然后是 C++ 业务逻辑处理类:

源代码 10: mustache_info.hpp

```
#ifndef MUSTACHE_INFO_HPP
#define MUSTACHE_INFO_HPP


#include <Poco/Net/HTTPRequestHandler.h>
#include <Poco/Net/HTTPServerRequest.h>
#include <Poco/Net/HTTPServerResponse.h>

```

```
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 <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")
              + "/demo/info.html");
33        std::string tpl;
34        Poco::StreamCopier::copyToString(IS, tpl);
35        Kainjow::Mustache engine(tpl);
36
37        response.setContentType("text/html;charset=utf-8");
38        response.setChunkedTransferEncoding(true);
39        engine.render({"tableList", tableList}, response.send());
40    }
41
42 }
```

源代码11相比于源代码8，已经非常简洁明晰。 页13

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          void handleRequest(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse&
                response);
14      private:
15          static Poco::ExpireCache<std::string, std::string> cache;
16      };
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 <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);
```

```
            kv.set("value", app.config().getString(rootItem + "." + confItem, "none"));
            trList.push_back(kv);
        }
        table.set("caption", rootItem);
        table.set("trList", trList);
        tableList.push_back(table);
        configKeys.clear();
    }
    Poco::FileInputStream IS(app.config().getString("http.tplDirectory", "/var/www/webcppd/tpl")
        + "/demo/info.html");
    std::string tpl;
    Poco::StreamCopier::copyToString(IS, tpl);
    Kainjow::Mustache engine(tpl);

    response.setContentType("text/html;charset=utf-8");
    response.setChunkedTransferEncoding(true);
    cache_mustache_info::cache.add(cacheKey, engine.render({"tableList", tableList}));
    response.send() << *cache_mustache_info::cache.get(cacheKey);
  }

}
```

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。 页19

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。

源代码 14: root_view.hpp

```
#ifndef ROOT_VIEW_HPP
#define ROOT_VIEW_HPP

#include <Poco/Net/HTTPRequestHandler.h>
#include <Poco/Net/HTTPServerRequest.h>
#include <Poco/Net/HTTPServerResponse.h>
```

```
7  #include <Poco/ExpireCache.h>
8  #include <string>
9
10 namespace webcppd {
11
12     class root_view : public Poco::Net::HTTPRequestHandler {
13     public:
14
15         void handleRequest(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse&
               response);
16     protected:
17
18         virtual void do_get(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse&
               response) = 0;
19
20         virtual void do_post(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse&
               response) = 0;
21
22         virtual void do_put(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse&
               response) = 0;
23
24         virtual void do_delete(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse&
               response) = 0;
25
26         void error(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse& response) {
27            response.setStatusAndReason(Poco::Net::HTTPServerResponse::HTTP_FORBIDDEN);
28            response.send() << "";
29         }
30         static Poco::ExpireCache<std::string, std::string> root_cache;
31     };
32
33
34 }
35
36 #endif /* ROOT_VIEW_HPP */
```

源代码 15: root_view.cpp

```
1  #include "root_view.hpp"
2
3  namespace webcppd {
4     Poco::ExpireCache<std::string, std::string> root_view::root_cache(600000);
5
6     inline void root_view::handleRequest(Poco::Net::HTTPServerRequest& request,
```

```
         Poco::Net::HTTPServerResponse& response) {
        if (request.getMethod() == Poco::Net::HTTPServerRequest::HTTP_GET) {
            this->do_get(request, response);
        } else if (request.getMethod() == Poco::Net::HTTPServerRequest::HTTP_POST) {
            this->do_post(request, response);
        } else if (request.getMethod() == Poco::Net::HTTPServerRequest::HTTP_PUT) {
            this->do_put(request, response);
        } else if (request.getMethod() == Poco::Net::HTTPServerRequest::HTTP_DELETE) {
            this->do_delete(request, response);
        } else {
            this->error(request, response);
        }

    }
}
```

如此，所有 Web 组件只要基于 webcppd::root_view 进行构造，就能轻易共享同一个根缓存器。

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 16: root_view_info.hpp

```
#ifndef ROOT_VIEW_INFO_HPP
#define ROOT_VIEW_INFO_HPP

#include "root_view.hpp"

namespace webcppd {

    class root_view_info : public root_view {
        void do_get(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse& response);

        void do_post(Poco::Net::HTTPServerRequest& request, Poco::Net::HTTPServerResponse& response) {
            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&
                response) {
20              this->error(request, response);
21          }
22
23
24
25      };
26  }
27
28  #endif /* ROOT_VIEW_INFO_HPP */
```

源代码 17: 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
7   #include <mustache.hpp>
8
9   #include "root_view_info.hpp"
10
11  namespace webcppd {
12
13      void root_view_info::do_get(Poco::Net::HTTPServerRequest& request,
            Poco::Net::HTTPServerResponse& response) {
14          std::string cacheKey("cache_mustache_info");
15          if (root_view_info::root_cache.has(cacheKey)) {
16              response.send() << *root_view_info::root_cache.get(cacheKey);
17              return;
18          }
19          Poco::Util::Application& app = Poco::Util::Application::instance();
20          Poco::Util::AbstractConfiguration::Keys rootKeys, configKeys;
21          app.config().keys(rootKeys);
22          Kainjow::Mustache::Data tableList(Kainjow::Mustache::Data::List());
23          for (auto& rootItem : rootKeys) {
```

```
            Kainjow::Mustache::Data table, trList(Kainjow::Mustache::Data::List());
            app.config().keys(rootItem, configKeys);
            for (auto& confItem : configKeys) {
                Kainjow::Mustache::Data kv;
                kv.set("key", confItem);
                kv.set("value", app.config().getString(rootItem + "." + confItem, "none"));
                trList.push_back(kv);
            }
            table.set("caption", rootItem);
            table.set("trList", trList);
            tableList.push_back(table);
            configKeys.clear();
        }
        Poco::FileInputStream IS(app.config().getString("http.tplDirectory", "/var/www/webcppd/tpl")
             + "/demo/info.html");
        std::string tpl;
        Poco::StreamCopier::copyToString(IS, tpl);
        Kainjow::Mustache engine(tpl);

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

}
```

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16

源代码 18: info_css.html

```
<!DOCTYPE html>
<html>
    <head>
        <title>webcppd 配置信息</title>
        <meta charset="UTF-8">
```

```
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <link rel="stylesheet" type="text/css" href="/assets/demo.css"/>
    </head>
    <body>
        {{#tableList}}
        <table class="gridtable">
            <caption>{{caption}}</caption>
            <tr><th>项目</th><th>取值</th></tr>
            {{#trList}}
            <tr>
                <td>{{key}}</td>
                <td>{{value}}</td>
            </tr>
            {{/trList}}
        </table>
        {{/tableList}}
        <script src="//cdn.bootcss.com/jquery/3.1.1/jquery.min.js"></script>
        <script src='/assets/demo.js'></script>
    </body>
</html>
```

源代码 19: demo.css

```
body{
    width:900px;
    margin: 40px auto;
}
table.gridtable {
    font-family: verdana,arial,sans-serif;
    font-size:11px;
    color:#333333;
    border-width: 1px;
    border-color: #666666;
    border-collapse: collapse;
    margin-left:auto;
    margin-right:auto;
    margin-bottom:30px;
}
table.gridtable th {
    border-width: 1px;
    padding: 8px;
    border-style: solid;
    border-color: #666666;
    background-color: #dedede;
```

```
    width:300px;
}
table.gridtable td {
    border-width: 1px;
    padding: 8px;
    border-style: solid;
    border-color: #666666;
    background-color: #ffffff;
    width:300px;
}
table.gridtable caption{
    font-size:19px;
}
```

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 页16 页16

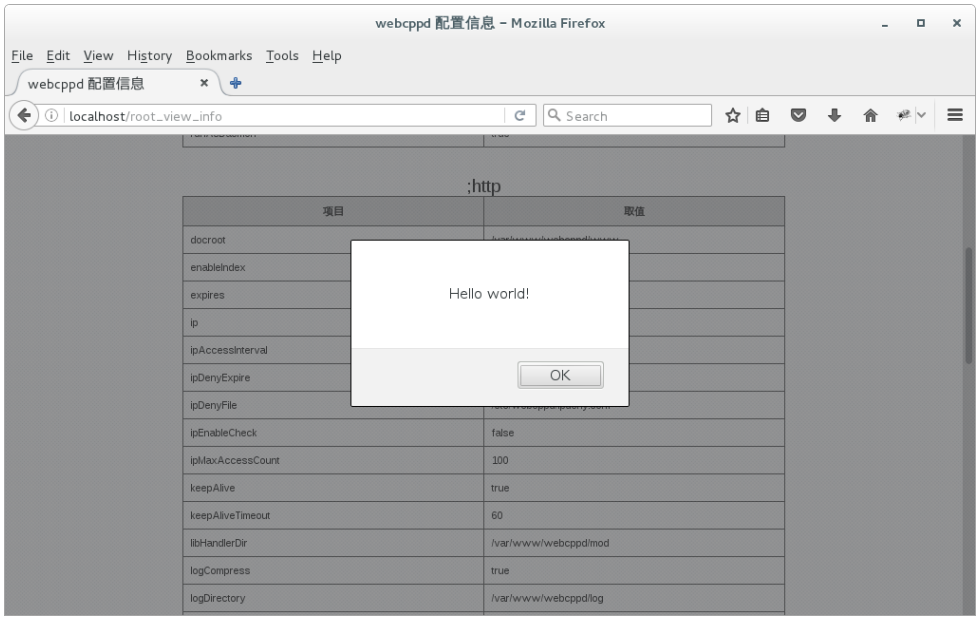


图 4: 分离前端要素演示

标签之前。

源代码 20: JS 要素

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

源代码 21: 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17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20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。