**将 Apache httpd 作为应用开发平台**

**Apache 模块开发**

Apache httpd 从 2.0 之后，已经不仅仅局限于一个 http 的服务器，更是一个完善而强大，灵活而健壮，且容易扩展的开发平台。开发人员通过定制 Apache 模块，可以几乎无限制的扩展 Apache httpd，使其更好的与实际应用场景匹配，而又无需考虑底层的网络传输细节。这样既可以提高开发效率，节省开发成本，又能充分利用 Apache 本身的健壮性及可靠性。

2https://dw1.s81c.com/developerworks/i/v17/dw-cmts-arrow.png [评论](https://www.ibm.com/developerworks/cn/opensource/os-cn-apachehttpd/#icomments)

[邱 俊涛](https://www.ibm.com/developerworks/cn/opensource/os-cn-apachehttpd/#authorN10029), 软件工程师, 云电同方研发中心

2012 年 2 月 24 日

* [expand](https://www.ibm.com/developerworks/cn/opensource/os-cn-apachehttpd/#toggle)**内容**



在 IBM Bluemix 云平台上开发并部署您的下一个应用。

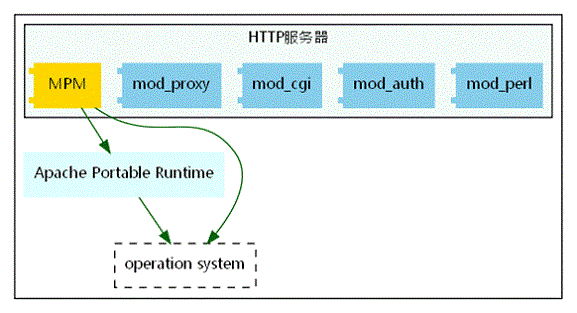
[**开始您的试用**](https://developer.ibm.com/sso/bmregistration?lang=zh_CN&ca=dwchina-_-bluemix-_-os-cn-apachehttpd-_-sidebar)

**基础知识**

**Apache httpd 开发平台简介**

Apache httpd 自 2.0 之后，针对 1.0 做了大量的改进，包括对自身内核的改造，扩展机制的改进，APR(Apache Portable Runtime) 的剥离 ( 使得 Apache 成为一个真正意义的跨平台服务器 )。Apache 2.0 成为一个很容易扩展的开发平台。

**图 1. Apache 服务器结构**



如上图中所示，Apache 中包含了大量的扩展，也就是本文中要详细讨论的模块 (module), 如 mod\_cgi 用以处理 cgi 脚本，mod\_perl 用以处理 perl 脚本，将 perl 的能力与 Apache httpd 结合起来等等。用户可以通过一定的开发标准来开发符合自己业务场景的模块，并动态的加载到 Apache 中，Apache 会根据配置文件中的规则来定位，调用模块，完成客户端请求。

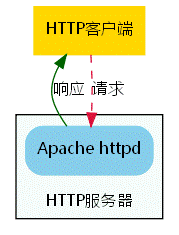
**Apache httpd 模块机制**

Apache httpd 由一个内核和大量的模块组成，包括用以加载其他模块的功能单元自身也是一个模块。一般而言，一个 HTTP 服务器的工作序列是这样的：接受客户端请求 ( 可能是请求一个

部署在 HTTP 服务器程序可访问的文件 )，读取该文件作为响应返回。

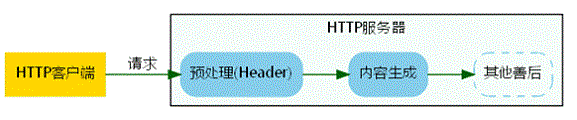
我们在浏览器的地址栏中输入类似这样的 URL：http://host/index.html，浏览器将会尝试与 host 指定的 HTTP 服务器的 80 端口建立连接，如果成功，则发送 HTTP 请求，获取 index.html 页面。如果成功，则在浏览器中解析该 HTML 文件。

**图 2. 简化的 HTTP 服务流程**



这种工作方式在静态页面的场景下没有任何问题。但是实际应用往往会与数据库交互，动态生成页面内容。如服务端较为流行的 cgi/php 脚本等。这就需要更高级，更灵活的内容生成器做支持。

**图 3. Apache 请求处理流程**



在请求处理流程中，包括预处理，内容生成及其他善后操作等。在预处理阶段，可以进行权限校验，HTTP 头信息识别等；内容生成阶段则通过与操作系统其他资源交互 ( 如文件读写，数据库访问等 ) 来完成动态内容的生成；最后在善后操作中，可能会进行日志记录，资源释放等操作。

下面我们就来开始我们的 Apache 模块开发之旅，首先定义一个最简单的模块原型，然后逐步扩展，使其可以完成更多的功能。在最后，这个模块可以读取客户端 POST 的数据，并将该数据加工，最终回显给客户端 ( 可以是浏览器或其他应用 )。

[**回页首**](https://www.ibm.com/developerworks/cn/opensource/os-cn-apachehttpd/#ibm-pcon)

**模块编写**

事实上，Apache 为开发人员提供了一系列的开发工具，使用它们，开发人员可以很方便的定义符合业务场景的 Apache 模块。这些工具中，较为常用的是 apxs。用 apxs 可以生成模块的模板文件，Makefile，目录结构等等。同时也可以用以编译，链接而生成最终的共享库文件 ( 在 Linux 系统下为 so 文件，类似于 windows 平台的动态链接库 dll)，Apache 模块的最终形式即为共享库文件。

**第一个 Apache 模块**

这一小节中，我们先使用 apxs 工具来生成一个模块模板，命名为 sample，命令如下：

**清单 1. 生成基本模板**

$ /usr/sbin/apxs -g -n sample

选项 -g 表示生成 (generate), 选项 -n sample 指定模块名称为 sample, apxs 会生成以下内容：

**清单 2. apxs 程序的输出**

$ /usr/sbin/apxs -g -n sample

Creating [DIR] sample

Creating [FILE] sample/Makefile

Creating [FILE] sample/modules.mk

Creating [FILE] sample/mod\_sample.c

Creating [FILE] sample/.deps

在当前目录创建了一个名为 sample( 模块名 ) 的目录，然后在该目录下生成了 Makefile 及 mod\_sample.c 文件，这个文件中已经包含了完整的模块代码。可以看到，Apache 模块代码与其他 C 工程中的代码并无二致。在不做任何修改的情况下，编译该文件就可以生成我们的第一个 Apache 模块 mod\_sample.so。这个模块的功能非常简单——仅以 text/html 形式向发起请求的客户端返回一个字符串：“The sample page from mod\_sample.c”。

**通用模板**

所有的 Apache 模块都需要遵从一定的规范来编写，这样 Apache 动态加载模块才能识别我们编写的模块并正常的工作。上例中通过 apxs 生成的 mod\_sample.c 即为一个典型的模块模板，该模板包括头文件的引入，模块存根的生成，handler 的示例等，开发人员可以通过 handler 的示例来完成模块的开发。

**清单 3. 通用模板**

/\* The sample content handler \*/

static int sample\_handler(request\_rec \*r)

{

if (strcmp(r->handler, "sample")) {

return DECLINED;

}

r->content\_type = "text/html";

if (!r->header\_only)

ap\_rputs("The sample page from mod\_sample.c\n", r);

return OK;

}

static void sample\_register\_hooks(apr\_pool\_t \*p)

{

ap\_hook\_handler(sample\_handler, NULL, NULL, APR\_HOOK\_MIDDLE);

}

/\* Dispatch list for API hooks \*/

module AP\_MODULE\_DECLARE\_DATA sample\_module = {

STANDARD20\_MODULE\_STUFF,

NULL, /\* create per-dir config structures \*/

NULL, /\* merge per-dir config structures \*/

NULL, /\* create per-server config structures \*/

NULL, /\* merge per-server config structures \*/

NULL, /\* table of config file commands \*/

sample\_register\_hooks /\* register hooks \*/

};

首先需要一个实际处理客户端请求的函数 (handler)，命名方式一般为”模块名 \_handler”，接收一个 request\_rec 类型的指针，并返回一个 int 类型的状态值。如：

**清单 4. handler 原型**

static int sample\_handler(request\_rec \*r);

request\_rec 指针中包括所有的客户端连接信息及 Apache 内部的指针，如连接信息表，内存池等，这个结构类似于 J2EE 开发中 servlet 的 HttpRequest 对象及 HttpResponse 对象。通过 request\_rec，我们可以读取客户端请求数据 / 写入响应数据，获取请求中的信息 ( 如客户端浏览器类型，编码方式等 )。

紧接着是一个注册函数，一般命名为”模块名 \_register\_hooks”，传入参数为 Apache 的内存池指针。这个函数用于通知 Apache 在何时，以何种方式注册响应函数 (handler)。

最后，是模块的定义，Apache 模块加载器通过这个结构体中的定义来在适当的时刻调用适当的函数以处理响应。应该注意的是，第一个成员默认填写为 STANDARD20\_MODULE\_STUFF，其他成员我们将在后续的小节中详细讨论。最后一个成员为上边讨论过的注册函数。

**运行模块**

编写好模块之后，通过 apxs 的编译功能将模块编译为共享库。生成的文件位于模块目录下的 .libs( 以点号开始的文件及文件夹在 Linux 系统中为隐藏文件，如果不指定 ls 命令的参数，ls 不会将此目录列出来，需要指定 -a 选项 ) 目录下。将此文件拷贝至 apache 安装目录下的 modules 目录 ( 为了便于描述，下文中以 apache\_home 表示 apache 的安装目录 )。然后修改 apache\_home/conf/httpd.conf，在该配置文件的最后加入：

**清单 5. httpd.conf 配置修改**

LoadModule sample\_module modules/mod\_sample.so

<Location /sample>

SetHandler sample

</Location>

LoadModule 指令意义为加载模块，setHandler 设置 handler，此处为”sample”，这个字符串与代码中的 r->handler 相同。这样，Apache 在处理对 /sample 的请求时，会调用我们编写的模块。

**清单 6. 重启 apache**

$ apache\_home/bin/httpd – d apache\_home – k stop

$ apache\_home/bin/httpd – d apache\_home – k start

然后，调用 curl 测试 sample 的模块：

**清单 7. 测试**

$ curl http://10.111.43.145:9527/sample -v

\* About to connect() to 10.111.43.145 port 9527 (#0)

\* Trying 10.111.43.145... connected

\* Connected to 10.111.43.145 (10.111.43.145) port 9527 (#0)

> GET /sample HTTP/1.1

> User-Agent: curl/7.21.6 (i686-pc-linux-gnu) \

libcurl/7.21.6 OpenSSL/0.9.7a zlib/1.2.1.2 libidn/0.5.6

> Host: 10.111.43.145:9527

> Accept: \*/\*

>

< HTTP/1.1 200 OK

< Date: Tue, 26 Jul 2011 07:12:19 GMT

< Server: Apache/2.0.63 (Unix)

< Content-Length: 34

< Connection: close

< Content-Type: text/html

<

The sample page from mod\_sample.c

\* Closing connection #0

我们在使用 curl 命令的时候，指定 -v 选项，则会打印出详细的客户端和服务端的交互信息，”>”开头的行表示 curl 发送到服务器的请求信息，而”<”开始的则为服务器端的响应信息。可以看到，模块已经如我们所预期的那样工作了。下面，我们将逐步扩展这个模块，使其功能更加强大。

**接收客户端数据的 echo 模块**

如果 Apache 模块只能产生内容，那么使用普通的 HTML 文件 ( 即使用 httpd 默认的内容生成器 ) 也可以完成。模块存在的意义在于，它可以轻松地处理客户端传递的数据，并将这些数据加工，然后响应客户端请求。我们在这一小节将开发一个可以接收客户端发送的 POST 请求，并将请求原封不动的回显给客户端的模块。

在这个模块中，我们自定义一个函数，用于读取 POST 请求数据。

**读取客户端数据**

**清单 8. 读取数据函数**

/\*\*

\* @brief read\_post\_data 从 request 中获取 POST 数据到缓冲区

\*

\* @param req apache request\_rec 对象

\* @param post 接收缓冲区

\* @param post\_size 接收缓冲区长度

\*

\* @return

\*/

static int read\_post\_data(request\_rec \*req, char \*\*post, size\_t \*post\_size){

char buffer[DFT\_BUF\_SIZE] = {0};

size\_t bytes, count, offset;

bytes = count = offset = 0;

if(ap\_setup\_client\_block(req, REQUEST\_CHUNKED\_DECHUNK) != OK){

return HTTP\_BAD\_REQUEST;

}

if(ap\_should\_client\_block(req)){

for(bytes = ap\_get\_client\_block(req, buffer, DFT\_BUF\_SIZE);

bytes > 0;

bytes = ap\_get\_client\_block(req, buffer, DFT\_BUF\_SIZE)){

count += bytes;

if(count > \*post\_size){

\*post = (char \*)realloc(\*post, count);

if(\*post == NULL){

return HTTP\_INTERNAL\_SERVER\_ERROR;

}

}

\*post\_size = count;

offset = count - bytes;

memcpy((char \*)\*post+offset, buffer, bytes);

}

}else{

\*post\_size = 0;

return OK;

}

return OK;

}

这个函数通过 Apache 提供的 API:ap\_get\_client\_block 将请求中 POST 的数据读入到缓冲区，如果预分配的缓冲区不够，则重新分配内存存放，并同时修改缓冲区的实际长度。然后，我们在 handler 中调用此函数

**清单 9. echo 模块的 handler 实现**

/\* The sample content handler \*/

static int echo\_post\_handler(request\_rec \*req)

{

if (strcmp(req->handler, "echo\_post")) {

return DECLINED;

}

if((req->method\_number != M\_GET) && (req->method\_number != M\_POST)){

return HTTP\_METHOD\_NOT\_ALLOWED;

}

char \*post = (char \*)malloc(sizeof(char)\*DFT\_BUF\_SIZE);

size\_t post\_size = DFT\_BUF\_SIZE;

if(post == NULL){

return HTTP\_INTERNAL\_SERVER\_ERROR;

}

memset(post, '\0', post\_size);

int ret = read\_post\_data(req, &post, &post\_size);

if(ret != OK){

free(post);

post = NULL;

post\_size = 0;

return ret;

}

ap\_set\_content\_type(req, "text/html;charset=utf-8");

ap\_set\_content\_length(req, post\_size);

if(post\_size == 0){

ap\_rputs("no post data found", req);

return OK;

}

ap\_rputs(post, req);

free(post);

post = NULL;

post\_size = 0;

return OK;

}

handler 读到客户端 POST 数据之后，将数据原封不动地回显 (echo) 给客户端。在调用 ap\_rputs 将数据写回客户端之后，释放动态分配的内存。

**运行 echo 模块**

配置信息如第一小节中，拷贝模块到 apache\_home/modules/ 下之后，重启 httpd，以便重新加载 echo\_post 模块。我们仍旧使用 curl 程序来模拟客户端调用，发送一个文件的内容到 echo\_post 模块：

**清单 10. 测试**

$ curl -F "file=@Makefile" http://10.111.43.145:9527/echo\_post

curl 的 -F 选项指定一个本地文件，并将文件内容 POST 到指定的 URL，运行结果如下：

**清单 11. 运行结果**

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

100 2334 100 1167 100 1167 28789 28789 --:--:-- --:--:-- --:--:-- 29175

------------------------------4dba7c85f3d1

Content-Disposition: form-data; name="file"; filename="Makefile"

Content-Type: application/octet-stream

##

## Makefile -- Build procedure for sample echo\_post Apache module

## Autogenerated via ``apxs -n echo\_post -g''.

##

builddir=.

top\_srcdir=/etc/httpd

top\_builddir=/etc/httpd

include /usr/lib/httpd/build/special.mk

# the used tools

APXS=apxs

APACHECTL=apachectl

# additional defines, includes and libraries

#DEFS=-Dmy\_define=my\_value

#INCLUDES=-Imy/include/dir

......

可以看到，echo\_post 会将 Makefile 的内容原封不动的返回。我们需要更进一步的改造这个模块，并引入模块的配置等新的内容。

**可配置的 echo 模块**

在这一小节，我们继续扩展上例中的 echo\_post 模块，我们将 echo\_post 扩展为可配置的模块，通过修改配置文件 httpd.conf 中设置 ConvertType 的值，可以使得模块在运行时的行为发生变化。

**配置信息读取**

首先定义一个配置信息的结构体：

**清单 12. 配置定义结构体**

typedef struct{

int convert\_type; // 转换类型

}cust\_config\_t;

为了便于演示，这个结构体仅有一个成员，convert\_type, 表示转换类型，如果在配置文件中该值被设置为 0，则将客户端 POST 的数据转换为大写，如果为 1，则转换为小写。这样即可通过配置信息修改模块运行时的行为。

加入函数声明：

**清单 13. 函数声明**

static void \*create\_config(apr\_pool\_t \*pool, server\_rec \*server);

static const char \*set\_mod\_config(cmd\_parms \*params,

void \*config, const char \*arg);

create\_config 函数用以创建一个用户自定义的结构体，告诉 Apache 如果创建这个结构体。set\_mod\_config 函数用以设置配置结构体中的成员，这个函数注册在 command\_rec 数组中。而 command\_rec 数组则保存在模块声明结构体中： 定义一个 command\_rec 结构体类型的数组：

**清单 14. command\_rec 结构体**

static const command\_rec cust\_echo\_cmds[] = {

AP\_INIT\_TAKE1("ConvertType",

set\_mod\_config,

NULL,

RSRC\_CONF,

"convert type of post data"),

{0}

};

这个模块的模块声明部分较前面小节中的例子更复杂一些，我们启用了配置信息表 cust\_echo\_cmds，并注册了创建配置结构的函数 create\_config。

**清单 15. 注册模块回调函数**

/\* Dispatch list for API hooks \*/

module AP\_MODULE\_DECLARE\_DATA cust\_echo\_post\_module = {

STANDARD20\_MODULE\_STUFF,

NULL, /\* create per-dir config structures \*/

NULL, /\* merge per-dir config structures \*/

create\_config, /\* create per-server config structures \*/

NULL, /\* merge per-server config structures \*/

cust\_echo\_cmds, /\* table of config file commands \*/

cust\_echo\_post\_register\_hooks /\* register hooks \*/

};

**运行可配置 echo 模块**

在 httpd.conf 配置文件中，加入 LoadModule 指令加载此模块，并设置配置项 ConvertType 为 0

**清单 16. httpd.conf 配置文件**

LoadModule cust\_echo\_post\_module modules/mod\_cust\_echo\_post.so

<Location /cust\_echo\_post>

SetHandler cust\_echo\_post

</Location>

#configure for cust\_echo\_post

ConvertType 0

这样，我们通过 curl 测试可以得到如下结果：

**清单 17. 测试 1**

$ curl -d "hello darkness my old friend" \

http://10.111.43.145:9527/cust\_echo\_post

HELLO DARKNESS MY OLD FRIEND

修改 httpd.conf 中的 ConvertType 为 1，重启 Apache httpd，重新测试：

**清单 18. 测试 2**

$ curl -d "HELLO DARKNESS MY OLD FRIEND" \

http://10.111.43.145:9527/cust\_echo\_post

hello darkness my old friend

**过滤器**

过滤器事实上是另一种形式的模块，其生命周期及调用时机请参看第一小节。Apache 对通用的数据结构都做过一些封装，并以库的方式提供 ( 即 APR(Apache Portable Runtime))。在过滤器中，有两个比较重要的数据结构：apr\_bucket 和 apr\_bucket\_brigade。apr\_bucket\_birgade 相当于一个环状队列，而 apr\_bucket 是队列中的元素。这两个数据结构的名字可能来源于救火队。救火人员站成一个长队，一个队头临近水源，另一头用水灭火，然后每个队列中的人员将水桶从上一个人手中接过，然后传递给下一个人。过滤器的工作方式与此类似，所有的过滤器形成一个长链，数据从上一个过滤器流入，进行过滤，然后将加工过的数据流入下一个过滤器。

我们的过滤器非常简单，从上一个过滤器中读到数据，将数据中的字符串转换为大写，然后将桶 (apr\_bucket) 传递给下一个过滤器。Apache 提供了丰富的 API 来完成这一系列的操作。

**大小写转换过滤器**

**清单 19. 过滤器实现**

static apr\_status\_t case\_filter(ap\_filter\_t \*filter,

apr\_bucket\_brigade \*bbin){

request\_rec \*req = filter->r;

conn\_rec \*con = req->connection;

apr\_bucket \*bucket;

apr\_bucket\_brigade \*bbout;

//create brigade

bbout = apr\_brigade\_create(req->pool, con->bucket\_alloc);

//iterate the full brigade

APR\_BRIGADE\_FOREACH(bucket, bbin){

if(APR\_BUCKET\_IS\_EOS(bucket) || APR\_BUCKET\_IS\_FLUSH(bucket)){

APR\_BUCKET\_REMOVE(bucket);

APR\_BRIGADE\_INSERT\_TAIL(bbout, bucket);

return ap\_pass\_brigade(filter->next, bbout);

}

char \*data, \*buffer;

apr\_size\_t data\_len;

//read content of current bucket in brigade

apr\_bucket\_read(bucket, &data, &data\_len, APR\_NONBLOCK\_READ);

buffer = apr\_bucket\_alloc(data\_len, con->bucket\_alloc);

int i;

for(i = 0; i < data\_len; i++){

//convert

buffer[i] = apr\_toupper(data[i]);

}

apr\_bucket \*temp\_bucket;

temp\_bucket = apr\_bucket\_heap\_create(

buffer, data\_len, apr\_bucket\_free, con->bucket\_alloc);

APR\_BRIGADE\_INSERT\_TAIL(bbout, temp\_bucket);

}

return APR\_SUCCESS;

}

注册这个过滤器：

**清单 20. 注册过滤器**

static void filter\_echo\_post\_register\_hooks(apr\_pool\_t \*p)

{

ap\_register\_output\_filter(filter\_name,

case\_filter, NULL, AP\_FTYPE\_RESOURCE);

}

**运行过滤器模块**

对过滤器的配置与前几个小节略有差异，在 httpd.conf 中，不但要使用 LoadModule 指令加载过滤器模块，还要使用 SetOutputFilter 指令来指定过滤器的应用场景：

**清单 21. httpd.conf 配置文件**

LoadModule filter\_echo\_post\_module modules/mod\_filter\_echo\_post.so

AddOutputFilter CaseFilter .cf

我们在指令中指定，CaseFilter 这个过滤器仅对扩展名为 .cf 的 URL 请求做过滤，其他请求则不过滤。然后，在 DocumentRoot 中创建一个文件，名为 casetest.cf 的文件，内容如下：

**清单 22. 测试文件内容**

$ cat casetest.cf

This line should be upcased.

So does this line.

然后重启 Apache，并使用 curl 测试该过滤器，可以得到以下结果：

**清单 23. 测试**

$ curl http://10.111.43.145:9527/casetest.cf

THIS LINE SHOULD BE UPCASED.

SO DOES THIS LINE.

即，将该文件中的字符串转换为大写字母输出。

[**回页首**](https://www.ibm.com/developerworks/cn/opensource/os-cn-apachehttpd/#ibm-pcon)

**问题处理**

实际开发过程中，代码很难一次性通过所有的应用场景，可能会有各种考虑不到的边界问题发生，因此需要一个方便的调试方式做支持。本文仅讨论在 Linux 操作系统环境下的调试。Apache 的开发者为此预留了一个入口，当启动 Apache 时，加入选项 (-X) 表示进入调试模式。

**模块的调试**

首先启动 gdb(Gnu Debugger)，gdb 是 Linux 下调试的工具，调试程序非常方便。可以通过 gdb 以调试模式启动 httpd，设置断点，进行调试 ( 部分 gdb 冗余的输出已经被删除 )：

**清单 24. 使用 gdb 调试 apache 模块**

$ gdb -q apache/bin/httpd

(gdb) break cust\_echo\_post\_handler

Function "cust\_echo\_post\_handler" not defined.

Make breakpoint pending on future shared library load? (y or [n]) y

Breakpoint 1 (cust\_echo\_post\_handler) pending.

(gdb) run -X -d apache

Starting program: /home/pts/juntao/cprog/idp/apache/bin/httpd -X -d apache

Breakpoint 2 at 0x416a4a: file mod\_cust\_echo\_post.c, line 122.

[Switching to Thread -1208781120 (LWP 23658)]

break 是 gdb 中设置断点的命令，run 是 gdb 程序的命令，表示启动应用程序。run 之后，gdb 停留在控制台上，Apache 进入 main-loop。 这时候我们即可通过客户端发送请求到 Apache httpd：

**清单 25. 使用 curl 模拟浏览器发送请求**

$ curl -F "file=@Makefile" http://10.111.43.145:9527/cust\_echo\_post

httpd 接受到请求之后，会进入我们预设的断点处，这样就可以开始调试：

**清单 26. 调试**

Breakpoint 2, cust\_echo\_post\_handler (req=0x8e1fad8) at mod\_cust\_echo\_post.c:122

(gdb)

这时候我们可以使用 next( 单步 )/step( 步入 )/print( 查看变量 ) 等命令来调试自己的模块。调试结束后，通过 quit 命令退出 gdb。gdb 本身功能非常强大，有兴趣的读者可以参考“参考资料”小节的资料。

[**回页首**](https://www.ibm.com/developerworks/cn/opensource/os-cn-apachehttpd/#ibm-pcon)

**结束语**

本文介绍了 Apache httpd 开发平台的运行方式，请求处理流程。并将一个最简单的模块模板逐步扩展，形成一个可以使用的模块，该模块将根据配置信息，将客户端 POST 的数据进行大 / 小写转化。最后是一个简单的过滤器示例，通过对整个开发流程的熟悉，读者可以开始开发自己的 Apache 模块，以适应不同的实际开发场景。