使用libevent编写高并发HTTP server - keep_simple

libevent库使得高并发响应HTTP Server的编写变得很容易。整个过程包括如下几部：初始化，创建HTTP Server, 指定callback, 进入事件循环。另外在回调函数中，可以获取客户端请求(request的HTTP Header和参数等)，进行响应的处理，再将结果发送给客户端(response的HTTP Header和内容，如html代码)。

libevent除了设置generic的callback，还可以对特定的请求路径设置对应的callback(回调/处理函数)。

示例代码(方便日后参考编写需要的HTTP server)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>     //for getopt, fork
#include <string.h>     //for strcat
//for struct evkeyvalq
#include <sys/queue.h>
#include <event.h>
//for http
#include <evhttp.h>
#include <signal.h>

#define MYHTTPD_SIGNATURE   "myhttpd v 0.0.1"

//处理模块
void httpd_handler(struct evhttp_request *req, void *arg) {
    char output[2048] = "\0";
    char tmp[1024];

    //获取客户端请求的URI(使用evhttp_request_uri或直接req->uri)
    const char *uri;
    uri = evhttp_request_uri(req);
    sprintf(tmp, "uri=%s\n", uri);
    strcat(output, tmp);

    sprintf(tmp, "uri=%s\n", req->uri);
    strcat(output, tmp);
    //decoded uri
    char *decoded_uri;
    decoded_uri = evhttp_decode_uri(uri);
    sprintf(tmp, "decoded_uri=%s\n", decoded_uri);
    strcat(output, tmp);
```

```
        //解析URI的参数(即GET方法的参数)
        struct evkeyvalq params;
        evhttp_parse_query(decoded_uri, &params);
        sprintf(tmp, "q=%s\n", evhttp_find_header(&params, "q"));
        strcat(output, tmp);
        sprintf(tmp, "s=%s\n", evhttp_find_header(&params, "s"));
        strcat(output, tmp);
        free(decoded_uri);

        //获取POST方法的数据
        char *post_data = (char *) EVBUFFER_DATA(req->input_buffer);
        sprintf(tmp, "post_data=%s\n", post_data);
        strcat(output, tmp);

        /*
        具体的：可以根据GET/POST的参数执行相应操作，然后将结果输出
        ...
        */

        /* 输出到客户端 */

        //HTTP header
        evhttp_add_header(req->output_headers, "Server",
MYHTTPD_SIGNATURE);
        evhttp_add_header(req->output_headers, "Content-Type",
"text/plain; charset=UTF-8");
        evhttp_add_header(req->output_headers, "Connection", "close");
        //输出的内容
        struct evbuffer *buf;
        buf = evbuffer_new();
        evbuffer_add_printf(buf, "It works!\n%s\n", output);
        evhttp_send_reply(req, HTTP_OK, "OK", buf);
        evbuffer_free(buf);

    }
    void show_help() {
        char *help = "written by Min (http://54min.com)\n\n"
            "-l <ip_addr> interface to listen on, default is 0.0.0.0\n"
            "-p <num>     port number to listen on, default is 1984\n"
            "-d           run as a deamon\n"
            "-t <second>  timeout for a http request, default is 120
seconds\n"
            "-h           print this help and exit\n"
            "\n";
```

```
        fprintf(stderr, help);
    }
    //当向进程发出SIGTERM/SIGHUP/SIGINT/SIGQUIT的时候，终止event的事件侦听循环
    void signal_handler(int sig) {
        switch (sig) {
            case SIGTERM:
            case SIGHUP:
            case SIGQUIT:
            case SIGINT:
                event_loopbreak();  //终止侦听event_dispatch()的事件侦听循环，
执行之后的代码
                break;
        }
    }

    int main(int argc, char *argv[]) {
        //自定义信号处理函数
        signal(SIGHUP, signal_handler);
        signal(SIGTERM, signal_handler);
        signal(SIGINT, signal_handler);
        signal(SIGQUIT, signal_handler);

        //默认参数
        char *httpd_option_listen = "0.0.0.0";
        int httpd_option_port = 8080;
        int httpd_option_daemon = 0;
        int httpd_option_timeout = 120; //in seconds

        //获取参数
        int c;
        while ((c = getopt(argc, argv, "l:p:dt:h")) != -1) {
            switch (c) {
            case 'l' :
                httpd_option_listen = optarg;
                break;
            case 'p' :
                httpd_option_port = atoi(optarg);
                break;
            case 'd' :
                httpd_option_daemon = 1;
                break;
            case 't' :
                httpd_option_timeout = atoi(optarg);
                 break;
```

```
            case 'h' :
            default :
                    show_help();
                    exit(EXIT_SUCCESS);
            }
        }

        //判断是否设置了-d，以daemon运行
        if (httpd_option_daemon) {
            pid_t pid;
            pid = fork();
            if (pid < 0) {
                perror("fork failed");
                exit(EXIT_FAILURE);
            }
            if (pid > 0) {
                //生成子进程成功，退出父进程
                exit(EXIT_SUCCESS);
            }
        }

        /* 使用libevent创建HTTP Server */

        //初始化event API
        event_init();

        //创建一个http server
        struct evhttp *httpd;
        httpd = evhttp_start(httpd_option_listen, httpd_option_port);
        evhttp_set_timeout(httpd, httpd_option_timeout);

        //指定generic callback
        evhttp_set_gencb(httpd, httpd_handler, NULL);
        //也可以为特定的URI指定callback
        //evhttp_set_cb(httpd, "/", specific_handler, NULL);

        //循环处理events
        event_dispatch();

        evhttp_free(httpd);
        return 0;
    }
```

- 编译：`gcc -o myhttpd -Wall -levent myhttpd.c`

- 运行：`./test`
- 测试：

在浏览器中输入`http://54min.com:8080/index.php?q=test&s=some thing`，显示内容如下：

```
It works!
uri=/index.php?q=test&s=some%20thing
uri=/index.php?q=test&s=some%20thing
decoded_uri=/index.php?q=test&s=some thing
q=test
s=some thing
post_data=(null)
```

并使用Live Http Headers(Firefox addons)查看HTTP headers。

```
HTTP/1.1 200 OK
Server: myhttpd v 0.0.1
Content-Type: text/plain; charset=UTF-8
Connection: close
Date: Tue, 21 Jun 2011 06:30:30 GMT
Content-Length: 72
```

使用libevent库进行HTTP封装方法的 应用

libevent库使得编写高并发高性能的HTTP Server变得很简单。因此实际中，使用libevent可以为任何应用(如数据库)提供一个HTTP based的网络接口，方便多个clients采用任何支持HTTP protocol的语言与server进行交互。例如：

对不支持HTTP协议的数据库(RDBMS/NoSQL)封装HTTP接口

- Memcached Server默认支持的是Memcached Protocol，通过libmemcached库和libevent库，可以为Memcached Server封装一个HTTP接口，从而client端可通过HTTP协议和Memcached Server进行交互；参考。
- 可为Tokyo Cabinet/Tokyo Tyrant数据库封装HTTP接口，方便client端与其交互。参考
- 相同的，也可为其他数据库如MySQL, Redis, MongoDB等封装HTTP接口；

更多的该种应用可以参考：https://github.com/bitly/simplehttp。

也可以为某些应用封装HTTP接口，从而实现以client/server方式使用该应用

- HTTPCWS，以libevent封装中文分词程序，实现client/server方式使用分词功能
- HTTP-SCWS也是类似方式

附：涉及的数据结构和主要函数

数据结构

- struct evhttp_request

表示客户端请求，定义参看：http://monkey.org/~provos/libevent/doxygen-1.4.10/structevhttp__request.html，其中包含的主要域：

```
    struct evkeyvalq *input_headers;    //保存客户端请求的HTTP headers(key-value pairs)
    struct evkeyvalq *output_headers;   //保存将要发送到客户端的HTTP headers(key-value pairs)

    //客户端的ip和port
    char *remote_host;
    u_short remote_port;

    enum evhttp_request_kind kind;  //可以是EVHTTP_REQUEST或EVHTTP_RESPONSE
    enum evhttp_cmd_type type;  //可以是EVHTTP_REQ_GET, EVHTTP_REQ_POST或EVHTTP_REQ_HEAD

    char *uri;          //客户端请求的uri
    char major;         //HTTP major number
    char minor;         //HTTP major number

    int response_code;      //HTTP response code
    char *response_code_line;   //readable response

    struct evbuffer *input_buffer;  //客户端POST的数据
    struct evbuffer *output_buffer; //输出到客户端的数据
```

- struct evkeyvalq

定义参看：http://monkey.org/~provos/libevent/doxygen-1.4.10/event_8h-source.html。`struct evkeyvalq`被定义为`TAILQ_HEAD (evkeyvalq, evkeyval);`，即`struct evkeyval`类型的tail queue。需要在代码之前包含

```
    #include <sys/queue.h>
    #include <event.h>
```

`struct evkeyval`为key-value queue(队列结构)，主要用来保存HTTP headers，也可以被用来保存parse uri参数的结果。

```
    /* Key-Value pairs.  Can be used for HTTP headers but also for query argument parsing. */
    struct evkeyval {
        TAILQ_ENTRY(evkeyval) next; //队列
```

```
    char *key;
    char *value;
};
```

宏`TAILQ_ENTRY(evkeyval)`被定义为：

```
#define TAILQ_ENTRY(type)
struct {
    struct type *tqe_next;      //next element
    struct type **tqe_prev;     //address of previous next element
}
```

- stuct evbuffer

定义参看：http://monkey.org/~provos/libevent/doxygen-1.4.10/event_8h-source.html。该结构体用于input和output的buffer。

```
/* These functions deal with buffering input and output */
struct evbuffer {
    u_char *buffer;
    u_char *orig_buffer;
    size_t misalign;
    size_t totallen;
    size_t off;
    void (*cb)(struct evbuffer *, size_t, size_t, void *);
    void *cbarg;
};
```

另外定义宏方便获取evbuffer中保存的内容和大小：

```
#define EVBUFFER_LENGTH(x)      (x)->off
#define EVBUFFER_DATA(x)        (x)->buffer
```

例如，获取客户端POST数据的内容和大小：

```
EVBUFFER_DATA(res->input_buffer);
EVBUFFER_LENGTH(res->input_buffer);
```

另外`struct evbuffer`用如下函数创建添加和释放：

```
struct evbuffer *buf;
buf = evbuffer_new();
//往buffer中添加内容
```

```
    evbuffer_add_printf(buf, "It works! you just requested: %s\n", req->uri);   //Append a formatted string to the end of an evbuffer.
    //将内容输出到客户端
    evhttp_send_reply(req, HTTP_OK, "OK", buf);
    //释放掉buf
    evbuffer_free(buf);
```

关键函数

- 获取客户端请求的URI

使用`req->uri`或使用函数`const char *evhttp_request_uri(struct evhttp_request *req);`即(`evhttp_request_uri(req);`)。

- 对获取的URI进行解析和其他操作

使用函数`void evhttp_parse_query(const char *uri, struct evkeyvalq *args);`可对uri的参数进行解析，结果保存在`struct evkeyvalq`的key-value pairs中，例如：

```
    char *uri = "http://foo.com/?q=test&s=some+thing";
    struct evkeyvalq args;
    evhttp_parse_query(uri, &args);
    //然后通过evhttp_find_header等函数获取各个参数及对应的值
    evhttp_find_header(&args, "q"); //得到test
    evhttp_find_header(&args, "s"); //得到some thing
```

如下两个函数对URI进行encode和decode：

```
    char *evhttp_encode_uri(const char *uri);
    char *evhttp_decode_uri(const char *uri);
```

URI encode的结果是所有非alphanumeric及`-_`的字符都被类似于`%`和一个2位16进制字符替换(其中空格被`+`号替换)。如上两个函数返回的字符串需要`free`掉。

- 处理HTTP headers相关的函数

HTTP headers保存在`struct evkeyvalq`的结构体中(key-value pairs)，使用如下函数可对其进行修改：

```
    const char *evhttp_find_header(const struct evkeyvalq *, const char *);
    int evhttp_remove_header(struct evkeyvalq *, const char *);
    int evhttp_add_header(struct evkeyvalq *, const char *, const char *);
    void evhttp_clear_headers(struct evkeyvalq *);
```

- Escape特殊的HTML字符

```
char *evhttp_htmlescape(const char *html);
```

特殊字符：`&`被替换为`&`；`"`被替换为`&quot;`；`'`被替换为`&#039;`;`<`被替换为`<`；`>`被替换为`>`。该函数返回的字符串需要`free`掉。

转载自:http://note.sdo.com/u/1730579924/n/D9ETk~jLB38MLX0a8000TB