



下载APP



24 | 实战项目（下）：一个简单的高性能 HTTP Server

2022-02-16 于航

《深入C语言和程序运行原理》

课程介绍 >



讲述：于航

时长 16:16 大小 14.90M



你好，我是于航。

在 [23 讲](#) 中，我对本次实战项目将要构建的程序 FibServ 的功能做了基本介绍，并从理论的角度，带你对它的基本实现方案有了一个初步认识。而这一讲，我们将通过实际编码，来应用这些理论知识。

领资料

为了便于你理解这一讲的内容，我已经将本项目的完整代码实现放到了 GitHub 上，你可以点击 [这个链接](#)，先大致浏览一下每个源文件的内容。而在后续讲解到相关代码时，也会在整段代码的第一行，通过注释的方式将这些代码的所在源文件标注出来。比如注释“libs/structs.h#L9-L11”，便表示当前所示的代码段对应于项目 libs 目录下，structs.h 文件内的第 9 到 11 行。其他注释的含义你可以此类推。

接下来，我会带你从基本的项目目录创建，到模块功能编写，再到代码编译和程序运行，一步步地完成整个项目的开发过程。

项目基本结构

首先，我们来看应该如何组织整个项目的目录结构。根据预估的项目体量，我使用了如下图所示的目录结构：



这里，整个项目包含有三个目录：`build`、`libs` 以及 `src`。其中，`build` 目录用于存放程序在 CMake 下的临时编译结果。如果你对这个目录还不太熟悉，可以参考我在 [22 讲](#) 中为你介绍的例子。`libs` 目录中主要存放可以模块化的独立功能实现，这些功能会以头文件的形式来提供外部可用接口，以供不同的应用程序使用。而最后的 `src` 目录则存放有与应用程序 FibServ 实现相关的源代码。

在此基础之上，你会发现我们分别在 `libs` 目录与项目根目录下，同时创建了用于控制 CMake 编译流程的 `CMakeLists.txt` 文件。其中，前者主要用于控制 `libs` 目录内 C 源代码的编译流程；而后者则用于控制应用程序 `FibServ` 的编译流程。我在这一讲后面的“编译与运行”一节中，还会再具体介绍这两个文件中的配置项。

处理用户输入参数

`FibServ` 在启动时可以接收一个由用户指定的，名为“`thread_count`”的参数。这个参数被用于控制 `FibServ` 应启用多少线程来处理收到的 HTTP 请求。

这里，我专门封装了一个用于描述服务器整体配置状态的结构类型 `serverSettings`，其中仅有的 `threadCount` 字段便对应于该参数，代码如下：

[复制代码](#)

```
1 // libs/structs.h#L9-L11
2 typedef struct {
3     int threadCount;
4 } serverSettings;
```

而通过对 `main` 函数的两个参数 `argc` 与 `argv` 进行解析，我们能够得到用户在运行程序时传入的所有参数。在名为 `setupServerSettings` 的函数中，我们通过上述这种方式，完成了对用户传入参数的解析与保存过程，解析得到的所有合法选项均被存放在一个 `serverSettings` 类型的对象中。该函数的实现代码如下所示：

[复制代码](#)

```
1 // libs/helpers.h#L67-L86
2 void setupServerSettings(int argc, const char** argv, serverSettings* ss) {
3     while (argc-- > 1) {
4         // process key.
5         const char* keyHead = argv[argc];
6         const char* keyPos = strchr(keyHead, '=');
7         const size_t keyLen = keyPos - keyHead + 1;
8         char key[keyLen];
9         wrapStrFromPTR(key, keyLen, keyHead, keyPos);
10        // process value.
11        const char* valHead = keyHead + keyLen;
12        const char* valPos = strchr(valHead, '\0');
13        const size_t valLen = valPos - valHead + 1;
14        char val[valLen];
15        for (size_t i = 0; valHead <= valPos; valHead++)
16            val[i++] = *valHead;
```

```
17     if (strcmp(key, "thread_count") == 0) {
18         ss->threadCount = atoi(val);
19     }
20 }
21 }
```

可以看到，通过判断 `argc` 的值是否大于 1（略过 `argv` 的第一个“程序文件名”参数），我们便能够对经由 `argv` 传递过来的输入参数进行遍历。每一次的遍历过程都分为两个步骤，即分别获取形如“key=value”的设置项的 key 与 value 两部分内容。

这里，我们使用 C 标准库中的 `strchr` 函数，来得到每个选项中“=”与字符串结尾空字符“\0”的位置，并以此将整个选项分为两段。在此基础之上，通过分别收集这两个区间（首字符至“=”，以及“=”至结尾空字符）内的字符，我们能够将一个选项的“键”与“值”进行拆分。最后，使用 `strcmp` 函数来比对所得到的键是否有效，若有效，则将相应的值存储到指定的 `serverSettings` 对象中。

至此，`FibServ` 便完成了用户配置项的初始化工作。接下来，我们继续为它实现 TCP Server 的核心功能。

实现 TCP Server

根据在上一讲中得到的结论，我们将使用来自 POSIX.1 标准的五个接口，`socket`、`bind`、`listen`、`accept` 与 `close`，来实现基本的 TCP 请求监听与连接创建等功能。

监听请求

为了将 `FibServ` 接受和处理 HTTP 请求的代码抽离出来，以方便后续的多线程改造，这里我将 TCP Server 的实现过程分为两个部分来介绍。

首先来看如何让程序进入“监听”状态，并持续等待 TCP 连接请求的到来。这部分代码将由 `main` 函数所在线程来执行，对应的代码如下所示：

```
1 // src/main.c#L70-95
2 int serverFd;
3 sockaddr_in address;
4 int addrLen = sizeof(address);
5
```

[复制代码](#)

```
6 // establish a socket.
7 if ((serverFd = socket(AF_INET, SOCK_STREAM, 0)) == 0) { ... }
8
9 bzero(&address, addrLen);
10 address.sin_family = AF_INET;
11 address.sin_addr.s_addr = INADDR_ANY; // -> 0.0.0.0.
12 address.sin_port = htons(PORT);
13
14 // assigns specified address to the socket.
15 if (bind(serverFd, (sockaddr*)&address, sizeof(address)) < 0) { ... }
16
17 // mark the socket as a passive socket.
18 if (listen(serverFd, MAX_LISTEN_CONN) < 0) { ... }
```

观察上述代码的第 7、15 与 18 行，你会发现，我们按顺序分别调用了接口 `socket`、`bind` 与 `listen`。其中，`socket` 接口共接收三个参数：`AF_INET` 宏常量表明使用 IPV4 因特网域；`SOCK_STREAM` 宏常量表明使用有序、可靠、双向，且面向连接的字节流；最后的参数 0 表明，让接口根据前两个参数自动选择使用的协议。当然，在这样的配置组合下，将默认使用 TCP 协议。

接下来，借助 `bind` 接口，我们可以为前一步创建的 `socket` 对象绑定一个地址。在 IPV4 域下，对应的地址用 `sockaddr_in` 类型的结构来表示。在代码的第 10~12 行，我们对该类型的一个对象 `address` 进行了初始化。其中，`sin_addr` 结构内的 `s_addr` 字段用于配置相应的地址信息。这里，`INADDR_ANY` 宏常量表示将 `socket` 绑定到地址 “0.0.0.0”。**通过这种方式，程序可以监听其运行所在的机器上发送到所有网卡（网络接口）的网络请求。**

另外需要注意的是，我们在为 `address` 对象设置用于表示网络端口的 `sin_port` 字段时，使用了名为 `htons` 的方法，来将一个 `short` 类型的端口值转换为**网络协议所要求字节序（通常为大端序）下的表示形式**。而之所以这样做，是为了让异构计算机在交换协议信息的过程中，不会被不同平台的不同字节序混淆。

紧接着，通过调用 `listen` 接口，程序开始监听即将到来的 TCP 连接请求。这里，通过它的第二个参数，我们指定了暂存队列最多能够存放的待连接请求个数为 `MAX_LISTEN_CONN` 个。

管理连接

到这里，程序便可以通过 `accept` 接口来不断地接受连接请求。此时，配合使用 `read` 和 `write` 这两个 IO 接口，我们可以获取客户端发来的数据，并在数据处理完毕后，再将特定内容返回给客户端。最后，当连接不再使用时，调用 `close` 接口即可将它关闭。关于这部分实现，你可以参考文件 “`src/main.c`” 中第 36~39 行，以及第 60 行的代码。

处理 HTTP 请求和响应

当 TCP 连接成功建立后，我们便可在此基础上进行与 HTTP 协议相关的操作，整个流程也十分简单，可以分为三个步骤：


1. 通过解析收到的 HTTP 请求报文，我们可以获取由客户端发送过来的参数值 `num`；
2. 调用斐波那契数列计算函数，相应的结果项可以被计算出来；
3. 通过构造一个合法的 HTTP 响应报文，我们可以将这个计算结果返回给客户端。

下面，我们来依次看看这几个步骤的具体实现。

解析请求

先来看第一步，解析请求。为了简化实现，这里我们将 HTTP 请求报文的解析过程分为了简单、直接的两个步骤，即**提取路径（URI）和解析查询参数值**。

当然，现实中的 HTTP 服务器应用通常会首先对报文的格式进行完整性校验，然后再进行类似的后续处理。与这部分逻辑相关的实现被封装在了名为 `retrieveGETQueryIntValByKey` 的函数中，它的实现代码如下所示：

 复制代码

```
1 // libs/helpers.c#L37-L65
2 int retrieveGETQueryIntValByKey(char* req, const char* key) {
3     int result = 0;
4
5     // extract uri;
6     const char* uriHead = strchr(req, ' ') + 1;
7     const char* uriTail = strchr(uriHead, '\n');
8     size_t uriLen = uriTail - uriHead + 1;
9     char strUri[uriLen];
10    wrapStrFromPTR(strUri, uriLen, uriHead, uriTail);
11
12    // parse uri;
13    UriUriA uri;
```



```
14 UriQueryListA* queryList;
15 int itemCount;
16 const char* errorPos;
17 if (uriParseSingleUriA(&uri, strUri, &errorPos) == URI_SUCCESS) {
18     if (uriDissectQueryMallocA(&queryList, &itemCount, uri.query.first, uri.qu
19         while (itemCount--) {
20             if (strcmp(queryList->key, key) == 0) {
21                 result = atoi(queryList->value);
22                 break;
23             }
24             queryList = queryList->next;
25         }
26         uriFreeQueryListA(queryList);
27     }
28 }
29 return result;
30 }
```

我在上一讲中曾为你介绍过 HTTP 请求报文的基本格式。其中，路径信息位于报文的“起始行”部分，而起始行中的每个元素则均以空格进行分割。因此，这里我们的第一步便是要**获取当整个报文从头向后逐字符遍历时，遇到的前两个空格之间的那段文本**。从代码中可以看到，这里我们再次使用了名为 `strchr` 的标准库函数来达到这个目的。

接下来，我们就会遇到“如何从路径文本中解析出给定查询参数的值？”这个棘手的问题。由于路径的形式可能有多种变化，比如未带有给定参数（`/?foo=1`）、带有除给定参数外的其他参数（`/?foo=1&num=10`）、带有子路径的参数（`/child/?num=10`），等等。因此，为了完善地处理这类情况，这里我们选用了——一个开源的第三方 URI 解析库，`uriparser`。

`uriparser` 的使用方式十分简单，通过配合使用 `uriParseSingleUriA` 与 `uriDissectQueryMallocA` 这两个接口，我们可以将所传入 URI 的查询参数与值提取成一个单链表。而通过对它进行遍历（`itemCount` 表示参数个数），我们便能够找到目标参数的值。你可以点击 [这个链接](#)，来了解关于 `uriparser` 的更多信息。

计算斐波那契数列

此时，我们已经得到了由客户端通过 HTTP 请求传送过来的参数 `num`。而下一步要完成的，便是 `FibServ` 的核心功能，即计算斐波那契数列第 `num` 项的值。与此相关的代码实现如下所示：

```
1 // libs/helpers.c#L8-L20
2 int __calcFibTCO(int n, int x, int y) {
3     if (n == 0)
4         return x;
5     if (n == 1)
6         return y;
7     return __calcFibTCO(n - 1, y, x + y);
8 }
9
10 int __calcFibRecursion(int n) {
11     if (n <= 1)
12         return n;
13     return __calcFibRecursion(n - 1) + __calcFibRecursion(n - 2);
14 }
15
16 int calcFibonacci(int n) {
17     // return __calcFibTCO(n, 0, 1); // TCO version.
18     return __calcFibRecursion(n); // recursion version.
19 }
```

这里，我为你提供了两种不同的函数实现。其中，函数 `__calcFibRecursion` 为正常递归版本；而函数 `__calcFibTCO` 为对应的尾递归版本。由于这两个版本函数需要的参数个数不同，因此为了统一对外的调用接口，我们为外部代码又提供了另一个稳定接口 `calcFibonacci`，而该接口在内部则可根据需要动态调用上述的两种函数实现。

返回响应

当计算过程结束后，我们便可以构造 HTTP 响应报文，并将结果返回给客户端。**表示操作成功的响应报文应使用内容为“HTTP/1.1 200 OK”的响应头。**在本例中，我们没有返回任何首部字段，因此，响应头与主体之间可以直接使用以 CRLF 结尾的一行空行进行分割。关于这部分实现，你可以参考文件“`src/main.c`”中第 55~56 行的代码。

整合多线程

经过上面几个步骤，我们已经可以让 `FibServ` 正常地接收与处理 HTTP 请求，并返回包含有正确结果的 HTTP 响应。但为了进一步提升 `FibServ` 的请求处理效率，我们将使用多线程技术来对它进行优化。

分离处理线程

首先，我们把请求处理相关的逻辑全部提取并封装在了名为 `acceptConn` 的函数中。这样，每一个线程便能够完全独立地来处理一个连接。该函数的完整实现代码如下所示：

[复制代码](#)

```
1 noreturn void* acceptConn(void *arg) {
2     acceptParams* ap = (acceptParams*) arg;
3     int acceptedSocket;
4
5     while (1) {
6         pthread_cleanup_push(renewThread, &acceptedSocket);
7         // extracts a request from the queue.
8         if ((acceptedSocket = accept(ap->serverFd, ap->addr, ap->addrLen)) < 0) {
9             perror("In accept");
10            pthread_exit(NULL);
11        }
12
13        // deal with HTTP request.
14        char reqBuf[HTTP_REQ_BUF];
15        bzero(reqBuf, HTTP_REQ_BUF);
16        const size_t receivedBytes = read(acceptedSocket, reqBuf, HTTP_REQ_BUF);
17        if (receivedBytes > 0) {
18            char resBuf[HTTP_RES_BUF];
19
20            // retrieve number from query.
21            pthread_mutex_lock(&mutex);
22            const int num = retrieveGETQueryIntValByKey(reqBuf, "num");
23            pthread_mutex_unlock(&mutex);
24
25            int fibResult = calcFibonacci(num);
26            // follow the format of the http response.
27            sprintf(resBuf, "HTTP/1.1 200 OK\r\n\r\n%d", fibResult);
28            write(acceptedSocket, resBuf, strlen(resBuf));
29        }
30        close(acceptedSocket);
31        pthread_cleanup_pop(0);
32    }
33 }
```

为了能够在线程内部获取与 `socket` 相关的文件描述符、地址，以及地址长度信息，以便于后续 `accept` 等接口的调用，这里我将这三个参数封装在了名为 `acceptParams` 的结构中。该结构的一个对象将在线程创建时以指针的形式传递进来，供所有处理线程使用。

在上述代码的第 5 行，通过一个死循环结构，线程便可持续不断地处理收到的连接请求。接下来，通过 `accept` 接口，我们可以从暂存队列中接受一个连接请求。代码的第 16 行，

借助 `read` 函数，程序获取到了由客户端发送来的数据。这些数据将随着第 22 与 25 行的处理，最终变为我们需要的计算结果。最后，在代码的第 27~28 行，我们将该结果整合在一个 HTTP 响应报文中，并返回给客户端。

优雅地处理异常

除此之外，你可能会发现，我们在代码的第 6 与 31 行，还调用了名为 `pthread_cleanup_push` 与 `pthread_cleanup_pop` 的两个函数。这两个函数的主要目的在于为线程添加退出时会自动执行的回调函数。这样，我们便能够在线程由于 `accept` 调用失败而退出时，及时地通知主线程重新创建新的处理线程，以保证线程池中线程的数量维持在一个稳定状态。关于这两个函数的更详细用法，你可以点击 [这个链接](#) 来了解。

但也正是因为需要对线程的状态进行更细致的管理，在本次实战项目中，我们并没有使用 C 标准中的线程库 `threads.h`，而是选用了 POSIX 标准下的 `pthread` 接口。但对于大多数的线程控制接口来说，两者只是在接口名称和使用方式上稍有差异，而背后的运行原理是完全一致的。

如下面的代码所示，在这个回调函数中，我们首先关闭了退出线程当前正在使用的 TCP 连接。接着，在互斥锁的保护下，我们更新了用于记录活动线程数量的全局变量 `threadCounter`，并通过 `pthread_cond_signal` 来通知 `main` 函数所在线程，重新创建处理线程（如果你对这里条件变量相关接口的使用方式还不太熟悉，可以重新回顾 [14 讲](#) 的相关内容）。


 复制代码

```
1 void renewThread(void *arg) {
2     int* acceptedSocket = (int*) arg;
3     close(*acceptedSocket);
4     pthread_mutex_lock(&mutex);
5     threadCounter--;
6     pthread_cond_signal(&cond); // notify main thread.
7     pthread_mutex_unlock(&mutex);
8 }
```

到这里，一切准备就绪。最后，让我们回到 `main` 函数内部，将“命运的齿轮”转动起来。

创建线程

如下面的代码所示，位于 main 函数内的死循环结构主要用于不断地创建新的处理线程。可以看到，在代码的第 11~12 行，我们使用 pthread_create 接口创建新的线程，并同时递增了工作线程计数器，即全局变量 threadCounter 的值。而直到该值大于等于程序配置状态对象 serverSettings 中 threadCount 的值时，通过调用 pthread_cond_wait 接口，main 线程进入到了阻塞状态。这个状态将会一直维持，直至有工作线程发生异常退出。

 复制代码


```
1 // src/main.c#L97-L109
2 while (1) {
3     pthread_mutex_lock(&mutex);
4     while (threadCounter >= ss.threadCount)
5         pthread_cond_wait(&cond, &mutex);
6     pthread_mutex_unlock(&mutex);
7
8     // create new thread to handle the request.
9     pthread_t threadId;
10    acceptParams ap = { serverFd, (sockaddr*) &address, (socklen_t*) &addrLen };
11    pthread_create(&threadId, NULL, acceptConn, &ap);
12    atomic_fetch_add(&threadCounter, 1);
13    printf("[Info] Thread Created: No.%d\n", threadCounter);
14 }
```

到这里，FibServ 在 C 代码层面的基本实现就结束了。接下来，我们将编写用于该项目的 CMake 配置文件，并尝试使用 cmake 命令对它进行编译。

编译与运行

我曾在这讲的第一个小节“项目基本结构”中提到，在本项目中，我们将使用两个 CMakeLists.txt 文件，来分别控制 libs 目录内的源代码，以及 FibServ 应用程序的编译过程。通过这种方式，CMake 会首先将前者编译为独立的“.a”静态库文件，然后再将该文件与 src 下的源代码一起编译，并生成最终的可执行文件。


libs 目录下的 CMakeLists.txt 文件内包含有以下内容：

 复制代码

```
1 # libs/CMakeLists.txt
2 aux_source_directory(. DIR_LIB_SRCS)
3 add_library(core STATIC ${DIR_LIB_SRCS})
```

其中，指令 `aux_source_directory` 用于收集当前目录下所有源文件的名称，并将这些名称存储到变量 `DIR_LIB_SRCS` 中。随后，通过 `add_library` 命令，从前一步收集而来的源文件将被一同编译，并生成名为 “core” 的静态库。

返回项目根目录，我们再来看用于控制应用程序 FibServ 编译的 `CMakeLists.txt` 文件中的内容，具体如下所示：

 复制代码

```
1 cmake_minimum_required(VERSION 3.21)
2 project(mini-http-server)
3
4 set(TARGET_FILE "http-echo-server")
5 set(CMAKE_BUILD_TYPE Release)
6 set(CMAKE_C_STANDARD 17)
7
8 # a simple way to check non-standard C header files (includes the atomic-relat
9 include(CheckIncludeFiles)
10 check_include_files("pthread.h;stdatomic.h;sys/socket.h;netinet/in.h;unistd.h"
11 if (EPHREAD EQUAL 1)
12     message(FATAL_ERROR "Necessary header files are not found!")
13 endif()
14
15 # for headers in "/libs" and other external installed packages.
16 include_directories(. /usr/local/include)
17
18 # load source files and sub-directories.
19 aux_source_directory(./src DIR_SRCS)
20 add_subdirectory(libs/)
21
22 # load packages.
23 find_package(uriparser 0.9.6 CONFIG REQUIRED char)
24
25 # for executable.
26 add_executable(${TARGET_FILE} ${DIR_SRCS})
27 target_link_libraries(${TARGET_FILE} PUBLIC core m pthread uriparser::uriparse
```

其中，你需要关注的是第 9~13 行的配置代码。这里，通过名为 `check_include_files` 的宏函数，我们能够让 CMake 在编译代码前，检测应用所需的指定头文件是否可以在当前环境下使用。若否，则直接终止项目的编译配置过程。

在接下来的第 16 行代码中，我们指定了程序编译时的头文件查找目录。通过第 19~20 行代码，FibServ 对应实现的源文件，以及所需要的内部依赖项（core 静态库）被引用进

来。在第 23 行，通过 `find_package` 指令，CMake 可以帮助我们查找程序需要使用的外部依赖包（这里即 `uriparser`）。

需要注意的是，`uriparser` 在可以被 CMake 正常使用之前，它需要被正确地编译和安装，你可以点击 [这个链接](#) 来查看更详细的说明。最后，通过 `add_executable` 与 `target_link_libraries` 两个指令，我们便可完成 `FibServ` 二进制文件的编译和链接过程。

当然，同我们在上一讲中介绍的方法类似，使用下面这行命令，我们可以一次性完成项目的编译前配置、代码编译与程序运行：

```
1 mkdir build && cd build && cmake .. && cmake --build . && ./fibserv
```

[复制代码](#)

总结

好了，讲到这里，今天的内容也就基本结束了。最后我来给你总结一下。

在这一讲中，我带你从基本的项目目录创建，到模块功能编写，再到代码编译和程序运行，一步步地完成了整个 `FibServ` 项目的开发过程，并为你详细介绍了这个项目的各个重要组成部分在代码层面的具体实现方式。

在“C 工程实战篇”的最后，我用两讲的篇幅带你实现了一个完整的 C 语言项目。希望通过这些内容，你能够对 C 语言在真实项目中的应用方式有更深刻的理解。同时，也希望你能以此为起点，在实践中持续运用这个模块中介绍到的各种 C 标准库功能与工程化实践技巧，进一步加深对整个 C 语言，乃至相关技术、工具和框架的理解。

思考题

你知道如何通过复用 TCP 连接来进一步优化 `FibServ` 的性能吗？欢迎在评论区告诉我你的实现方案，也欢迎你直接提交 PR！

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎把这节课分享给你的朋友或同事，我们一起交流。

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 3 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 23 | 实战项目（上）：一个简单的高性能 HTTP Server

下一篇 25 | 可执行二进制文件里有什么？

更多学习推荐

2021 最新大厂 Go 工程师面试真题

大厂面试真题 + 金九银十全新整理 + 核心知识全覆盖

限量免费领取



精选留言 (2)

写留言



fee1in 置顶

2022-02-16

花了两个小时 终于将示例跑通 后面的同学doxygen问题 可以参考老师在uriparser的issue
<https://github.com/uriparser/uriparser/issues/137>

共 1 条评论 >

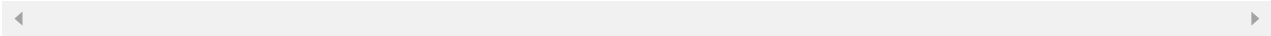
2



叶三
2022-02-16

于老师讲的太好了，受益匪浅

作者回复：感谢支持！



2