23 | 实战项目（上）：一个简单的高性能 HTTP Server

2022-02-14 于航

《深入C语言和程序运行原理》 课程介绍

讲述：于航

时长 13:26 大小 12.32M

你好，我是于航。

在“C 核心语法实现篇”中，通过观察 C 代码被编译后的产物，我们了解了 C 基本语法在机器指令层面的多种具体实现细节。进入“C 工程实战篇”后，通过探索 C 标准库，我们发现了 C 语言为我们提供的更多优秀能力，并同时深入分析了它们的内部实现原理。在此基础之上，通过探讨 C 项目编码规范、代码优化技巧、自动化测试与结构化编译等话题，我们对 C 语言在实际工程中的应用方式又有了更深刻的理解。

但“光说不练假把式”，在本模块最后，就让我们通过实现一个完整的 C 语言项目，来整体回顾之前的学习内容，并尝试在实战过程中体会 C 这门语言的独特魅力。

这是一个怎样的实战项目？

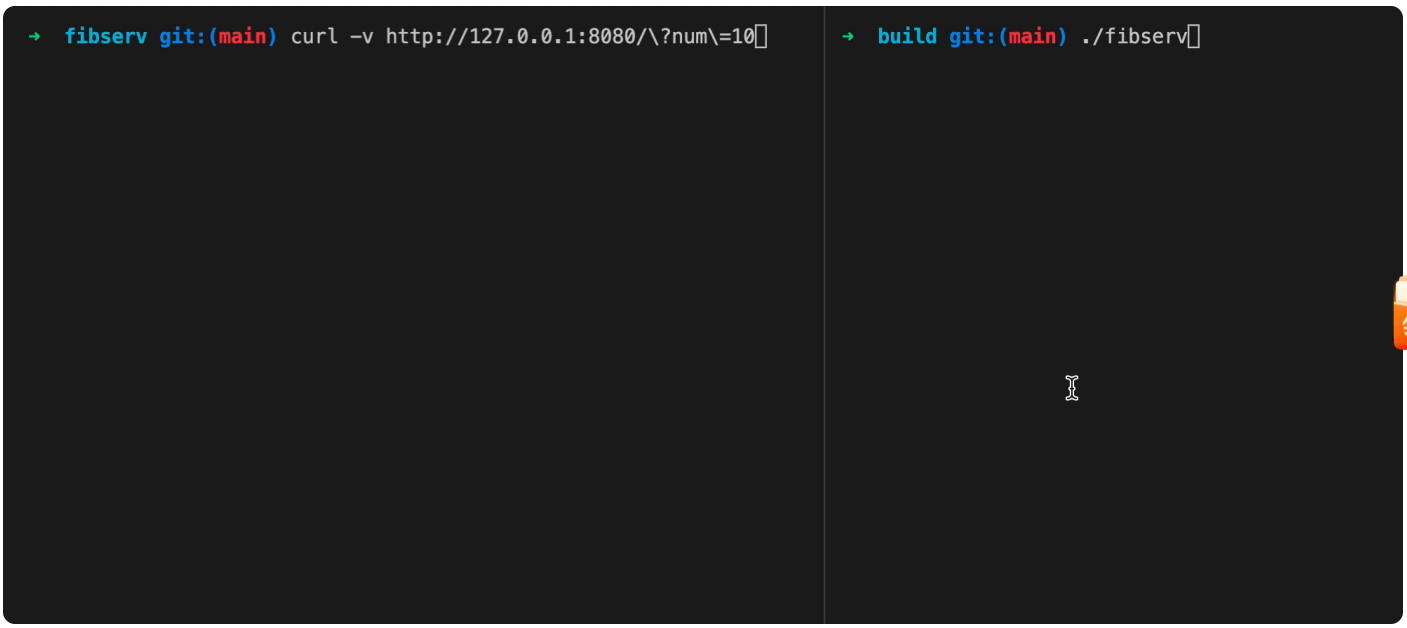
俗话说得好，“有趣是第一生产力”。但似乎是从大学时代第一次接触 C 语言开始，我们就对使用这门语言开发的项目有了刻板印象，感觉它们不是枯燥的用户后台管理系统，就是各类晦涩的、与操作系统或硬件深入“绑定”的底层应用。但现实情况却并非如此。正如我在开篇词中介绍的那样，C 语言可以被广泛使用在应用软件、系统软件、编程语言、嵌入式开发等各类场景中。而今天我们要做的项目，便是应用软件类目下服务器应用中的一种，“HTTP Server”。

Server 翻译过来即“服务器”，它在整个互联网世界中，主要用于接收由客户端发来的请求，并在处理之后返回相应结果。而 HTTP 服务器则将可处理的请求类型限定为了 “HTTP 请求”。这类服务器的稳定运行，支撑了我们日常生活中需要与互联网打交道的大多数事务。比如，每一次打开网页，都伴随着浏览器发出 HTTP 请求，服务器返回 HTTP 响应的过程。而这些返回的内容在经过浏览器渲染后被呈现在了你的面前。

当然，考虑到篇幅和实现难度，在本次实战中，我们不会实现一个支持完整 HTTP 协议的服务器应用。我将会带你实现一个名为 “FibServ” 的程序。**在这一讲接下来的内容中，我将从理论的角度，来为你介绍应该如何使用 C 语言在 Linux 环境下实现它的主要功能。而在下一讲，我们将带着这些理论成果进入到实际的编码环节。**

FibServ 在运行时可以接收形式为 “**/?num={pos}**” 的 GET 请求。其中，参数 num 对应的值 pos 为一个具体的整数。随后，程序会将该值作为一个索引信息，并返回斐波那契（Fibonacci）数列中对应该位置上的值。

你可以参考下面这张动图，来观察程序的实际运行状态。



图片中包含有左右两个命令行窗口。在右侧窗口中，我们首先运行了 FibServ。它会在当前计算机的 8080 端口上监听即将收到的 HTTP 请求。紧接着，在左侧窗口里，我们使用 curl 命令，向当前计算机（127.0.0.1）的 8080 端口发送了一个带有 “num=10” 参数的 GET 请求。经过一段时间，当 FibServ 处理完该请求后，包含有结果值 “55” 的响应被传送回来。

可以看到，FibServ 的功能十分简单。接下来，我们就从方案设计的角度入手，来看应该如何实现它的主要功能。

如何使用 POSIX 接口实现 TCP Server？

实际上，FibServ 的最核心功能便是对 HTTP 请求的接收与应答。在本次实战中，我们将以 HTTP 1.1 标准作为实现要求。如下所示，根据 RFC 7230 对该标准的规定，HTTP 作为一种应用层协议，需要基于以传输层 TCP 协议建立的网络连接来实现。

> Although HTTP is independent of the transport protocol, the “http” scheme is specific to TCP-based services because the name delegation process depends on TCP for establishing authority.

在 Linux 系统中，借助套接字（Socket）接口，我们便能够建立这样的一个连接。这套接口属于 POSIX.1 标准的一部分，因此，它也同时被 Unix 与各种类 Unix 操作系统采用。套接字接口的全称一般为“套接字网络进程间通信接口”。从名称上就可以看出，通过这个接口，多个进程之间便可进行在同一网络下，甚至是跨不同网络的通信过程。对应到上面的动图，FibServ 与 curl 之间的交互过程便是如此。

而**通过配合使用名为 socket、bind、listen、accept 以及 close 的五个接口，我们便能够完成 FibServ 最核心的网络请求接收功能**。

其中，socket 接口用于创建套接字。套接字是一种用于描述通信端点的抽象实体，你可以这样简单理解：无论是客户端还是服务器，只要是能够参与到网络通信，并且可以进行数据传递的实体，它们都可以被抽象为一种特定类型的 socket 对象。

相应地，socket 接口暴露出了三个参数，用于指定这些不同对象在多个重要特征（通信域、套接字类型，及所使用的协议）上的不同。该接口的函数原型如下所示，接口在调用后会返回一个整型的文件描述符，以用于在后续代码中指代该 socket 资源。

```
int socket(
  int domain,
  int type,
  int protocol);
```

接下来，通过名为 bind 的接口，我们可以让套接字与一个具体的地址进行关联。通常来说，bind 被更多地用于为服务器类型端点对应的 socket 对象分配固定地址。这样，客户端便可通过这个地址来连接该服务器程序。

但需要注意的是，bind 接口使用的地址必须是在程序进程运行所在的计算机上有效的。在 FibServ 的实现中，我们将直接使用本机地址 “127.0.0.1”。bind 接口在调用时共接收两部分信息，一部分为某个具体 socket 对象对应的文件描述符；另一部分为与所关联地址相关的数据结构。它的函数原型如下所示：

```
int bind(
  int sockfd,
  const struct sockaddr *addr,
  socklen_t addrlen);
```

此时，通过名为 listen 的接口，我们可以将一个 socket 对象变为“被动 socket”，也就是说，该 socket 会在某个地址上持续被动地等待从外部发来的连接请求，而不会自己主动发起连接。当该接口调用完毕后，所有 socket 接收到的连接请求都会被暂时存放到一个队列中，以等待下一步处理。

如下面的函数原型所示，listen 接口共接收两个参数，第一个参数为某个具体 socket 对象对应的文件描述符；第二个参数用于控制“暂存队列”的大小。当该队列发生溢出时，后续的连接请求将会被直接拒绝（ECONNREFUSED）。


```
int listen(int sockfd, int backlog);
```

最后，通过 accept 接口，我们可以从被动 socket 对应的暂存队列中依次取出已经到来的连接请求。这里，该接口会为每一个已接受的请求建立一个新的、表示已连接的 socket 对象。而在接下来的程序中，通过使用 read 与 write 等 IO 接口，我们便可直接使用该 socket，来读取

出对应请求携带的数据，并同时将适当的结果返回给客户端。而若在调用 accept 接口时，暂存队列中没有待处理的连接请求，则接口调用者将会进入阻塞状态，直到下一个连接请求的到来。

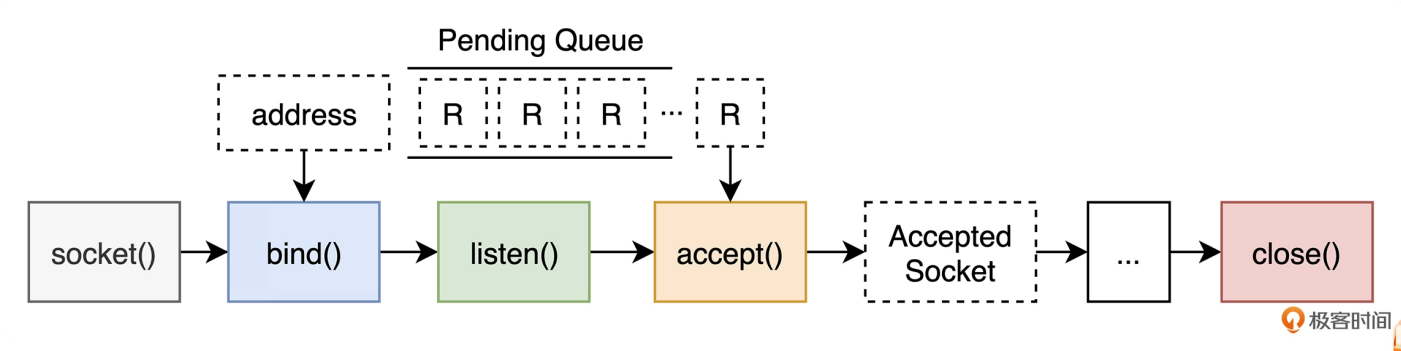
通过下面的函数原型，你可以看到该接口接收的参数与 bind 接口十分类似。

```
int accept(
  int sockfd,
  struct sockaddr *restrict addr,
  socklen_t *restrict addrlen);
```

与 IO 操作类似的是，当这个新创建的、对应于已接受连接的 socket 对象被使用完毕后，我们也需要通过 close 接口来关闭它所对应的文件描述符。在这个过程中，与该 socket 相关的系统资源会被清理，并且，对应的 TCP 连接也会被关闭。

```
int close(int fd);
```

至此，通过上面的几个简单步骤，我们便可成功实现一个基本的 TCP Server。这里你可以暂缓脚步，通过下面这张图片，来回顾一下上述实现流程。



TCP 协议主要规定了应该如何在通信双方之间，提供可靠和有序的数据流传输能力。因此，它并未对基于该连接传送的具体数据格式做任何要求和假设。而对这些传输字节流的解释，则由 TCP 连接双方根据应用层的具体协议来进行。在此基础上，我们便能够进一步来实现 HTTP 协议。

TCP 之上：HTTP 协议有何不同？

与 TCP 协议的复杂性比起来，HTTP 协议就相对简单很多。

在 HTTP 1.1 中，请求与响应的报文均是以纯文本的形式来在客户端与服务器之间传递的。这也就意味着，当 FibServ 在处理一个 HTTP 请求时，实际上就是在处理这个请求对应的，一堆按照特定格式组织的 ASCII 字符。至于这些字符的具体内容，你可以通过 read 接口，从已被接受的连接对应的 socket 对象中读取出来。

对 FibServ 来说，一个正确的 HTTP 请求报文的格式可能如下所示：

```
GET /?num=10 HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: ApacheBench/2.3
Accept: */*
```

整个报文被分为三部分，即**起始行、首部字段，以及主体**。其中，起始行中包含有与该请求相关的方法（GET）、路径（/?num=10），以及协议与版本（HTTP/1.1）信息。而首部字段中则包含所有的请求头信息，这些信息通常用于控制客户端或服务器应该如何处理该请求。对于某些属于特定方法的请求，报文中通常还可能包含有与请求一同发送过来的必要数据，这部分数据则被整理在了报文“最下方”的主体部分中。

如下所示，相应的 HTTP 响应报文也有着同样的三部分结构，只是起始行中包含的信息发生了变化：

```
HTTP/1.1 200 OK
Content-type: text/plain
Content-length: 2

55
```


这里的 “200” 和 “OK” 分别表示了该响应的状态码与可读状态信息。总的来看，请求报文的起始行描述了这个请求“希望要做的事情”；而响应报文的起始行则描述了“这件事做完后的结果状态”。与之前的请求报文不同，这个响应报文中还包含有主体部分的数据 “55”，而这部分数据便是从服务器返回的请求结果。

另外还值得一提的是，标准中规定，起始行和首部的每一行都需要以 CRLF，即“回车符”加“换行符”的形式结尾。而在首部字段与主体之间，则需要以 CRLF 结尾的一行空行进行分割。这是我们在代码中构建 HTTP 报文时需要注意的一点。

HTTP 1.1 除了对报文的具体格式做了详细规定外，它还对 TCP 连接控制、缓存管理、认证机制等其他重要功能的实现要求进行了说明。但考虑到实现成本，在本次实战中，我们仅会对 FibServ 收到的 HTTP 请求报文进行适当解析，并返回相应的响应报文。而在其他部分的实现上，可能并没有遵循 HTTP 协议的相关规定（比如默认情况下应使用长连接）。

我们会应用哪些优化策略？

为了尽可能提高 FibServ 处理请求时的性能，我们将从几个很容易想到的地方入手，来对程序进行适当的优化。

简易线程池

首先来思考下：如何让程序充分利用多核 CPU 的多个处理单元？相信这个问题一定难不倒你。没错，答案就是使用多线程。

在 FibServ 的实现中，我们将为它构建一个拥有固定 N 个处理线程的简易线程池。其中，N 可以由用户在运行 FibServ 时，通过添加额外的参数 “thread_count” 来指定。每一个线程在运行时，都会通过 accept 接口，独立地从 socket 对应的暂存队列中取出下一个待连接请求，并进行相应处理。通过这种方式，我们可以充分利用多个 CPU 核心，以让它们并行地处理多个请求。

尾递归调用

另一方面，对于斐波那契数列的计算函数，我将分别提供它的正常递归版本与尾递归版本。通过这种方式，你能够明显地观察到尾递归优化带来的，可观的性能提升。

避免忙等待

最后一个优化点虽然不会带来直观的性能改变，但对于理解“条件变量”在实际项目中的应用方式，却是十分有帮助的。

为了确保 FibServ 能够在请求处理线程异常退出时，仍然保证线程池中的线程数量为 N，这里我不会使用忙等待的方式持续判断存活的线程数量。相对地，我会使用条件变量，来让处理线

程在退出时，及时通知主线程创建新的处理线程。而主线程也将在处理线程数量满足要求时，再次进入阻塞状态。

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。

今天我主要为你介绍了与本次实战项目相关的一些理论性知识，以便为下一讲的实际编码打下基础。

我们要构建的是一个名为 FibServ 的程序，该程序在运行时会扮演 HTTP 服务器的角色，并持续监听来自本地的 HTTP 请求。相应的请求需要为 GET 方法，并携带名为 “num” 的查询参数。FibServ 在收到该类型请求后，会计算斐波那契数列中在对应位置上的项，并将该值返回给客户端。

在 Unix 与类 Unix 系统中，借助 POSIX.1 标准提供的五个接口，即 socket、bind、listen、accept 与 close，我们可以为程序实现监听并接收 TCP 连接请求的功能。而 HTTP 协议作为一种基于纯文本的应用层协议，我们可以在此基础上，在程序层面完成对请求报文的解析，以及响应报文的构建过程。

为了进一步提升 FibServ 处理请求时的效率，我们还将为它提供了简易的线程池实现，以通过增加工作线程的方式，来进一步利用多核 CPU 的处理单元。同时，通过适当改写用于求取斐波那契数列的计算函数，编译器可以帮助我们将它的实现方式由递归优化为迭代，进而大幅提升运行性能。最后，借助条件变量，我们可以优化线程池在线程异常退出时的处理方式，让整个处理流程变得更加优雅。

思考题

针对我们的实战项目 FibServ，你还能想到哪些可以进一步优化的地方呢？欢迎在评论区告诉我你的想法。

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，欢迎你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，你最高得 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

上一篇　期中测试丨来检验下你的学习成果吧！

下一篇　24丨实战项目（下）：一个简单的高性能 HTTP Server

精选留言 (4)

写留言

zxk
2022-04-05

使用 reactor 模型

Frankey
2022-02-16

使用epoll？

作者回复: 没错，异步 IO 是一个好主意。


I WANN BE THAT G...
2022-02-16

c语言有尾递归优化吗？

作者回复: C 编译器可以对符合要求的代码进行尾递归优化，可以参考 06 讲的内容。


LDxy
2022-02-14

使用缓存，把已经计算过的数据缓存起来

作者回复: 很棒，使用 HTTP 缓存是一个好方案。