第7课 Hello World服务器

介绍

- 目标：学习使用C++和socket编程建立一个基础Web服务器。
- 内容概述：
  - 理解Web服务器工作原理。
  - 学习创建和监听socket。
  - 掌握接收和响应HTTP请求的方法。

Web服务器概念详细扩展

定义

- Web服务器：一种服务器软件或硬件，专门用于在互联网或内部网中接收HTTP请求，并根据这些请求提供网页HTML文件或其他数据。
- 核心功能：**处理客户端（如Web浏览器或其他HTTP客户端）的请求，并将结果返回给客户端**。

工作原理（重要）

1. **监听端口**
   - Web服务器在网络中开放一个或多个端口（如HTTP的默认端口80），用于监听客户端的请求。
   - 当服务器绑定到特定端口后，它会持续检测该端口是否有传入的连接请求。
2. **处理请求**
   - 当接收到请求时，服务器首先解析HTTP请求头和请求体的内容。
   - 解析过程包括提取请求的方法（GET、POST等）、目标资源的URI、HTTP版本和请求头信息等。
   - 服务器还可以处理附加数据，例如在POST请求中提交的表单数据或上传的文件。
3. **发送响应**
   - 根据请求的类型和资源，服务器确定响应的内容。这可能是HTML页面、图像、样式表、脚本文件或其他类型的数据。
   - 服务器生成HTTP响应消息，包括状态行（如HTTP/1.1 200 OK）、响应头（如内容类型、内容长度等）和响应体（请求的资源或错误信息）。

小学长的考研Top帮

- 最后，服务器通过与客户端建立的连接发送这个HTTP响应。

重要组成

- 静态内容处理：直接从服务器文件系统中提供文件，如HTML文件、图像、样式表和JavaScript文件。**服务器直接从硬盘读取并发送回客户端。**
- **动态内容生成：运行应用程序或脚本来动态生成网页内容，**
  - 服务器会将请求传递给应用程序服务器或脚本引擎（如PHP解释器、Java Servlet容器、Python WSGI应用服务器等），由它们处理业务逻辑并生成最终的HTTP响应内容。
- 安全性管理：包括加密通信（HTTPS）、身份验证、访问控制等。
  - 支持HTTPS加密通信，通过SSL/TLS协议确保传输数据的安全性。
  - 实现用户身份验证机制，例如基本认证、摘要认证、JWT token验证等。
  - 实施访问控制策略，基于角色的权限管理（RBAC）以保护敏感资源不被未授权访问
- 日志记录：记录服务器活动，如访问记录、错误信息等，用于监控和调试。

服务器类型（了解即可）

1. 专用服务器软件： 这类服务器软件专注于处理HTTP请求，提供Web内容服务，以及进行相关的安全控制、负载均衡和性能优化等功能。其中几个著名例子包括：
   - Apache HTTP Server：这是互联网上最广泛应用的开源Web服务器软件，支持多种操作系统平台。它稳定可靠，高度可配置，并且能够处理静态网页和动态内容（通过与脚本语言解释器如PHP或CGI程序配合工作）。
   - Nginx：也是一款高性能的Web服务器及反向代理服务器、负载均衡器。相较于Apache，Nginx以其非阻塞、事件驱动的架构而闻名，在高并发场景下表现出色，尤其适合做静态文件服务和作为API网关。
   - Microsoft Internet Information Services (IIS)：是微软开发的一款Web服务器产品，主要用于Windows环境。IIS与.NET框架紧密集成，为ASP.NET应用提供了强大的支持。
2. 集成服务器环境： 集成服务器环境（也称为技术栈或开发堆栈）是一组相互协作的技术组件，这些组件共同构成了一个完整的Web应用部署平台。常见的有：
   - LAMP：Linux（操作系统）、Apache（Web服务器）、MySQL（数据库管理系统）和PHP/Perl/Python（脚本语言）的组合。LAMP栈是一个开放源码解决方案，被广泛用于快速搭建和部署动态网站和应用。
   - MEAN：MongoDB（文档型数据库系统）、Express.js（基于Node.js的Web应用框架）、AngularJS（客户端JavaScript MVC框架，现已被Angular取代）和Node.js（JavaScript运行环境，用于服务器端编程）。MEAN栈采用JavaScript作为前后端统一的编程语言，为构建全栈式JavaScript应用提供了便捷途径。

专用服务器软件应用场景：

1. 企业级Web应用部署：
    - Apache和Nginx常用于托管大型企业的静态网站、动态网页应用或API服务。例如，许多电子商务网站、社交媒体平台以及内容管理系统（如WordPress）都基于Apache或Nginx进行部署。
2. **高性能Web服务**：
    - 对于需要处理高并发访问、大流量负载的场景，如视频流媒体服务、实时消息系统等，可以选择Nginx作为前端服务器负责负载均衡和静态资源分发，后端可以搭配其他专用服务器或容器集群。
3. Windows环境下的Web开发：
    - 在Windows操作系统上，IIS由于与微软技术栈的良好兼容性而被广泛使用，尤其适合开发ASP.NET Web应用程序和服务。

集成服务器环境应用场景：

1. **快速开发与原型验证**：
    - LAMP或MEAN堆栈提供了一整套开箱即用的解决方案，开发者可以迅速搭建起一个完整的Web应用环境，并实现从数据库存储到前后端交互的全流程开发。
2. 开源社区和初创公司：
    - 由于LAMP和MEAN都是开源的且成本相对较低，因此深受开源社区和初创公司的喜爱。这些团队可以通过它们构建灵活、可扩展的Web应用，并利用丰富的社区支持和资源快速迭代产品。
3. 全栈JavaScript开发：
    - MEAN栈特别适用于那些希望采用JavaScript统一前后端编程语言的项目，开发者能够更好地复用代码、降低学习曲线，并享受到Node.js带来的高效非阻塞I/O性能。

从学长的考研TOP帮

这节课用到的C++语法知识

类型别名与 `std::function` 模板详解

类型别名（Type Alias）

在C++编程中， `using` 关键字用于创建已存在类型的别名。这一特性有助于：

1. **提升代码可读性**：对于复杂的类型定义，如模板类或模板函数的实例化结果，通过类型别名赋予其更具描述性的名称，使得代码阅读更加直观易懂。

示例：

```
using MyVector = std::vector<int>; // 定义了std::vector<int>类型的别名MyVector
```

2. 简化代码重复：避免在多个地方重复书写相同类型的长而复杂的定义，通过类型别名可以集中管理和引用。

3. 增强抽象与封装：将底层实现细节隐藏起来，用类型别名表示特定的概念或接口，提高代码模块化的程度。

`using namespace std;` 的含义：

在C++标准库中，所有标准组件（如 `vector`、`string`、`cout` 等）都被定义在 `std` 命名空间内。为了使用这些组件，程序员通常需要在代码中明确指定它们来自 `std` 命名空间，例如通过 `std::cout << "Hello, World!";` 来输出字符串。

然而，如果在程序的某个作用域（通常是文件顶部或函数内部）包含 `using namespace std;` 语句，则表明该作用域内的代码可以直接使用 `std` 命名空间内的所有组件，而无需显式地写出 `std::` 前缀。比如，使用了 `using namespace std;` 后，可以简化为 `cout << "Hello, World!";`。

`std::function` 模板（了解）

`std::function` 是C++标准库中的一个模板类，**它能够存储和调用任何可调用对象（Callable Object），包括函数指针、lambda表达式、成员函数指针以及绑定到特定对象的成员函数等**。`std::function` 提供了一种统一的方式来处理不同类型的可调用对象，使得在设计API时更加灵活，无需预先知道具体的调用者类型。

```
// 声明一个 std::function 实例，表示它可以存储接受两个整数参数并返回一个整数的可调用对象
std::function<int(int, int)> add_function;

// 现在可以将符合这个签名的任何可调用对象赋值给它，如下面的普通函数、lambda 或者绑定到成员函数的对象
add_function = [](int a, int b) { return a + b; }; // lambda 函数

```

```cpp
// 如果有一个这样的全局函数:
int global_add(int a, int b) {
    return a + b;
}

// 也可以赋值给上面声明的 std::function 实例
add_function = &global_add;

// 对于类的成员函数，可以通过 std::bind 或 lambda 来创建匹配签名的可调用对象
class MyClass {
public:
    int member_add(int a, int b) {
        return a + b;
    }
};

MyClass obj;
add_function = std::bind(&MyClass::member_add, &obj, std::placeholders::_1, std::placeholders::_2); // 绑定成员函数

// 使用 std::function 调用
int result = add_function(3, 5); // 将会执行加法操作并返回结果8
```

它具有以下核心特征：

1. **存储任意可调用实体**：`std::function` 能够容纳任何符合特定签名的可调用对象，包括全局函数、成员函数指针、lambda表达式、仿函数以及其他Callable对象。

2. 签名指定：在声明一个 `std::function` 实例时，需要明确指出其内部可存储的函数签名，例如：

```cpp
std::function<std::string(const std::string&)> requestHandler; // 可以存储接受一个const std::string&参数并返回一个std::string的函数
```

3. 延迟绑定和运行时多态：`std::function` 允许在运行时动态决定调用哪个具体的函数或对象，实现了类似于虚函数表的机制。

M学长的考研Top帮

4. 类型安全性保证：虽然`std::function`能包含多种类型的可调用实体，但它确保所有存储的对象都能遵循预定义的函数签名进行调用，从而保障类型安全。

举例说明：

```cpp
#include <iostream>
#include <functional>

// 定义一个处理字符串的函数
std::string processString(const std::string& input) {
    return "Processed: " + input;
}

int main() {
    // 使用类型别名定义请求处理器类型
    using RequestHandler = std::function<std::string(const std::string&)>;

    // 创建一个RequestHandler实例，并绑定到processString函数
    RequestHandler handler = processString;

    // 或者使用lambda表达式创建另一个RequestHandler实例
    RequestHandler anotherHandler = [](const std::string& s) -> std::string {
        return "Another way to process: " + s;
    };

    // 现在handler和anotherHandler都可以作为处理字符串请求的对象
    std::string result1 = handler("Hello");
    std::string result2 = anotherHandler("World");

    std::cout << result1 << std::endl;
    std::cout << result2 << std::endl;

    return 0;
}
```

在此例中，`RequestHandler`是一个类型别名，代表了一种可以处理字符串请求的函数签名。实际的处理逻辑可以通过灵活地传递和替换函数指针或者lambda表达式来实现。

计算机通信的完整过程

尽量掌握，面试过程中记得越详细越好，互联网大厂面试属于超高频

下面搬运自《计算机网络自顶向下方法》

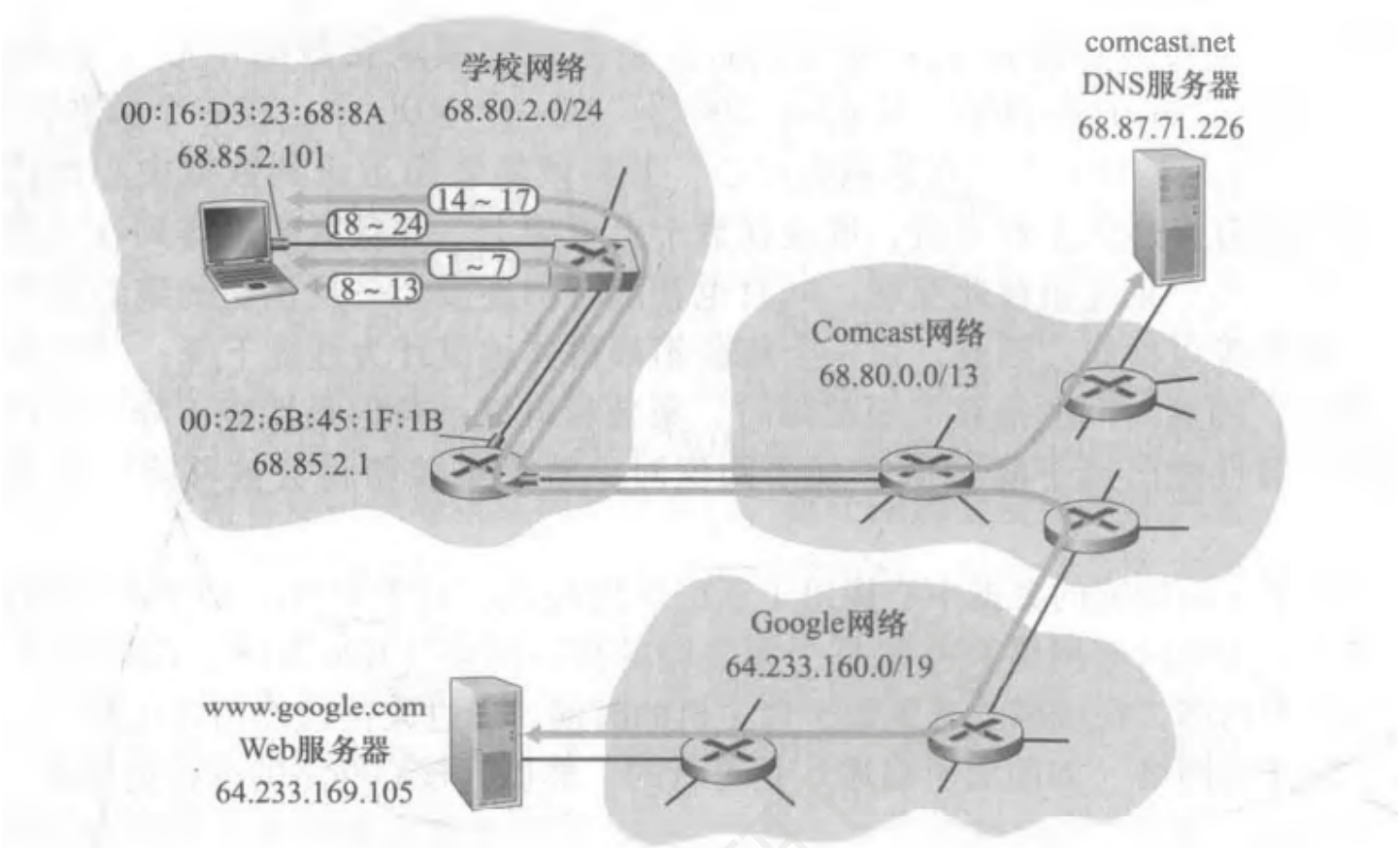

图 6-32　Web 页请求的历程：网络环境和动作

6.7.1　准备：DHCP、UDP、IP 和以太网

我们假定 Bob 启动他的便携机，然后将其用一根以太网电缆连接到学校的以太网交换机，交换机又与学校的路由器相连，如图 6-32 所示。学校的这台路由器与一个 ISP 连接，本例中 ISP 为 comcast. net。在本例中，comcast. net 为学校提供了 DNS 服务；所以，DNS 服务器驻留在 Comcast 网络中而不是学校网络中。我们将假设 DHCP 服务器运行在路由器中，就像常见情况那样。

当 Bob 首先将其便携机与网络连接时，没有 IP 地址他就不能做任何事情（例如下载一个 Web 网页）。所以，Bob 的便携机所采取的一个网络相关的动作是运行 DHCP 协议，以从本地 DHCP 服务器获得一个 IP 地址以及其他信息。

1）Bob 便携机上的操作系统生成一个 **DHCP 请求报文**（4. 3. 3 节），并将这个报文放入具有目的端口 67（DHCP 服务器）和源端口 68（DHCP 客户）的 **UDP 报文段**（3. 3 节）

该 UDP 报文段则被放置在一个具有广播 IP 目的地址（255.255.255.255）和源 IP 地址 0.0.0.0 的 **IP 数据报**中（4.3.1 节），因为 Bob 的便携机还没有一个 IP 地址。

2）包含 DHCP 请求报文的 IP 数据报则被放置在**以太网帧**中（6.4.2 节）。该以太网帧具有目的 MAC 地址 FF:FF:FF:FF:FF:FF，使该帧将广播到与交换机连接的所有设备（如果顺利的话也包括 DHCP 服务器）；该帧的源 MAC 地址是 Bob 便携机的 MAC 地址 00:16:D3:23:68:8A。

3）包含 DHCP 请求的广播以太网帧是第一个由 Bob 便携机发送到以太网交换机的帧。该交换机在所有的出端口广播入帧，包括连接到路由器的端口。

4）路由器在它的具有 MAC 地址 00:22:6B:45:1F 的接口接收到该广播以太网帧，该帧中包含 DHCP 请求，并且从该以太网帧中抽取出 IP 数据报。该数据报的广播 IP 目的地址指示了这个 IP 数据报应当由在该节点的高层协议处理，因此该数据报的载荷（一个 UDP 报文段）**被分解**（3.2 节）向上到达 UDP，DHCP 请求报文从此 UDP 报文段中抽取出来。此时 DHCP 服务器有了 DHCP 请求报文。

5）我们假设运行在路由器中的 DHCP 服务器能够以 CIDR（4.3.3 节）块 68.85.2.0/24 分配 IP 地址。所以本例中，在学校内使用的所有 IP 地址都在 Comcast 的地址块中。我们假设 DHCP 服务器分配地址 68.85.2.101 给 Bob 的便携机。DHCP 服务器生成包含这个 IP 地址以及 DNS 服务器的 IP 地址（68.87.71.226）、默认网关路由器的 IP 地址（68.85.2.1）和子网块（68.85.2.0/24）（等价为“网络掩码”）的一个 DHCP ACK **报文**（4.3.3 节）。该 DHCP 报文被放入一个 UDP 报文段中，UDP 报文段被放入一个 IP 数据报中，IP 数据报再被放入一个以太网帧中。这个以太网帧的源 MAC 地址是路由器连到归属网络时接口的 MAC 地址（00:22:6B:45:1F:1B），目的 MAC 地址是 Bob 便携机的 MAC 地址（00:16:D3:23:68:8A）。

M学长的考研小站

6）包含 DHCP ACK 的以太网帧由路由器发送给交换机。因为交换机是**自学习**的（6.4.3 节），并且先前从 Bob 便携机收到（包含 DHCP 请求的）以太网帧，所以该交换机知道寻址到 00:16:D3:23:68:8A 的帧仅从通向 Bob 便携机的输出端口转发。

7）Bob 便携机接收到包含 DHCP ACK 的以太网帧，从该以太网帧中抽取 IP 数据报，从 IP 数据报中抽取 UDP 报文段，从 UDP 报文段抽取 DHCP ACK 报文。Bob 的 DHCP 客户则记录下它的 IP 地址和它的 DNS 服务器的 IP 地址。它还在其 **IP 转发表**中安装默认网关的地址（4.1 节）。Bob 便携机将向该默认网关发送目的地址为其子网 68.85.2.0/24 以外的所有数据报。此时，Bob 便携机已经初始化好它的网络组件，并准备开始处理 Web 网页获取。（注意到在第 4 章给出的四个步骤中仅有最后两个 DHCP 步骤是实际必要的。）

6.7.2 仍在准备：DNS 和 ARP

当 Bob 将 www.google.com 的 URL 键入其 Web 浏览器时，他开启了一长串事件，这将导致谷歌主页最终显示在其 Web 浏览器上。Bob 的 Web 浏览器通过生成一个 **TCP 套接字**（2.7 节）开始了该过程，套接字用于向 www.google.com 发送 **HTTP 请求**（2.2 节）。为了生成该套接字，Bob 便携机将需要知道 www.google.com 的 IP 地址。我们在 2.4 节中学过，使用 **DNS 协议**提供这种名字到 IP 地址的转换服务。

8）Bob 便携机上的操作系统因此生成一个 **DNS 查询报文**（2.4.3 节），将字符串 www.google.com 放入 DNS 报文的问题段中。该 DNS 报文则放置在一个具有 53 号（DNS 服务器）目的端口的 UDP 报文段中。该 UDP 报文段则被放入具有 IP 目的地址 68.87.71.226（在

M学长的考研小窝

第 5 步中 DHCP ACK 返回的 DNS 服务器地址）和源 IP 地址 68.85.2.101 的 IP 数据报中。

9）Bob 便携机则将包含 DNS 请求报文的数据报放入一个以太网帧中。该帧将发送（在链路层寻址）到 Bob 学校网络中的网关路由器。然而，即使 Bob 便携机经过上述第 5 步中的 DHCP ACK 报文知道了学校网关路由器的 IP 地址（68.85.2.1），但仍不知道该网关路由器的 MAC 地址。为了获得该网关路由器的 MAC 地址，Bob 便携机将需要使用 **ARP 协议**（6.4.1 节）。

10）Bob 便携机生成一个具有目的 IP 地址 68.85.2.1（默认网关）的 **ARP 查询报文**，将该 ARP 报文放置在一个具有广播目的地址（FF:FF:FF:FF:FF:FF）的以太网帧中，并向交换机发送该以太网帧，交换机将该帧交付给所有连接的设备，包括网关路由器。

11）网关路由器在通往学校网络的接口上接收到包含该 ARP 查询报文的帧，发现在 ARP 报文中目标 IP 地址 68.85.2.1 匹配其接口的 IP 地址。网关路由器因此准备一个 **ARP 回答**，指示它的 MAC 地址 00:22:6B:45:1F:1B 对应 IP 地址 68.85.2.1。它将 ARP 回答放在一个以太网帧中，其目的地址为 00:16:D3:23:68:8A（Bob 便携机），并向交换机发送该帧，再由交换机将帧交付给 Bob 便携机。

12）Bob 便携机接收包含 ARP 回答报文的帧，并从 ARP 回答报文中抽取网关路由器的 MAC 地址（00:22:6B:45:1F:1B）。

13）Bob 便携机现在（最终!）能够使包含 DNS 查询的以太网帧寻址到网关路由器的 MAC 地址。注意到在该帧中的 IP 数据报具有 IP 目的地址 68.87.71.226（DNS 服务器），而该帧具有目的地址 00:22:6B:45:1F:1B（网关路由器）。Bob 便携机向交换机发送该帧，交换机将该帧交付给网关路由器。

M学长的考研OOP帮

6.7.3 仍在准备：域内路由选择到 DNS 服务器

14）网关路由器接收该帧并抽取包含 DNS 查询的 IP 数据报。路由器查找该数据报的目的地址（68.87.71.226），并根据其转发表决定该数据报应当发送到图 6-32 的 Comcast 网络中最左边的路由器。IP 数据报放置在链路层帧中，该链路适合将学校路由器连接到最左边 Comcast 路由器，并且该帧经这条链路发送。

15）在 Comcast 网络中最左边的路由器接收到该帧，抽取 IP 数据报，检查该数据报的目的地址（68.87.71.226），并根据其转发表确定出接口，经过该接口朝着 DNS 服务器转发数据报，而转发表已根据 Comcast 的域内协议（如 RIP、OSPF 或 IS-IS，5.3 节）以及**因特网的域间协议 BGP**（5.4 节）所填写。

16）最终包含 DNS 查询的 IP 数据报到达了 DNS 服务器。DNS 服务器抽取出 DNS 查询报文，在它的 DNS 数据库中查找名字 www.google.com（2.4 节），找到包含对应 www.google.com 的 IP 地址（64.233.169.105）的 **DNS 源记录**。（假设它当前缓存在 DNS 服务器中。）前面讲过这种缓存数据源于 google.com 的**权威 DNS 服务器**（2.4.2 节）。该 DNS 服务器形成了一个包含这种主机名到 IP 地址映射的 **DNS 回答报文**，将该 DNS 回答报文放入 UDP 报文段中，该报文段放入寻址到 Bob 便携机（68.85.2.101）的 IP 数据报中。该数据报将通过 Comcast 网络反向转发到学校的路由器，并从这里经过以太网交换机到 Bob 便携机。

17）Bob 便携机从 DNS 报文抽取出服务器 www.google.com 的 IP 地址。最终，在大量工作后，Bob 便携机此时准备接触 www.google.com 服务器！

M学长的考研T0作

6.7.4 Web 客户－服务器交互：TCP 和 HTTP

18）既然 Bob 便携机有了 www. google. com 的 IP 地址，它能够生成 TCP 套接字（2.7 节），该套接字将用于向 www. google. com 发送 HTTP GET 报文（2.2.3 节）。当 Bob 生成 TCP 套接字时，在 Bob 便携机中的 TCP 必须首先与 www. google. com 中的 TCP 执行三次握手（3.5.6 节）。Bob 便携机因此首先生成一个具有目的端口 80（针对 HTTP 的）的 TCP SYN 报文段，将该 TCP 报文段放置在具有目的 IP 地址 64.233.169.105（www. google. com）的 IP 数据报中，将该数据报放置在 MAC 地址为 00:22:6B:45:1F:1B（网关路由器）的帧中，并向交换机发送该帧。

19）在学校网络、Comcast 网络和谷歌网络中的路由器朝着 www. google. com 转发包含 TCP SYN 的数据报，使用每台路由器中的转发表，如前面步骤 14～16 那样。前面讲过支配分组经 Comcast 和谷歌网络之间域间链路转发的路由器转发表项，是由 BGP 协议决定的（第 5 章）。

20）最终，包含 TCP SYN 的数据报到达 www. googole. com。从数据报抽取出 TCP SYN 报文并分解到与端口 80 相联系的欢迎套接字。对于谷歌 HTTP 服务器和 Bob 便携机之间的 TCP 连接生成一个连接套接字（2.7 节）。产生一个 TCP SYNACK（3.5.6 节）报文段，将其放入向 Bob 便携机寻址的一个数据报中，最后放入链路层帧中，该链路适合将 www. google. com 连接到其第一跳路由器。

21）包含 TCP SYNACK 报文段的数据报通过谷歌、Comcast 和学校网络，最终到达 Bob 便携机的以太网卡。数据报在操作系统中分解到步骤 18 生成的 TCP 套接字，从而进入连接状态。

22）借助于 Bob 便携机上的套接字，现在（终于！）准备向 www. google. com 发送字节了，Bob 的浏览器生成包含要获取的 URL 的 HTTP GET 报文（2.2.3 节）。HTTP GET 报文则写入套接字，其中 GET 报文成为一个 TCP 报文段的载荷。该 TCP 报文段放置进一个数据报中，并交付到 www. google. com，如前面步骤 18～20 所述。

23）在 www. google. com 的 HTTP 服务器从 TCP 套接字读取 HTTP GET 报文，生成一个 HTTP 响应报文（2.2 节），将请求的 Web 页内容放入 HTTP 响应体中，并将报文发送进 TCP 套接字中。

24）包含 HTTP 回答报文的数据报通过谷歌、Comcast 和学校网络转发，到达 Bob 便携机。Bob 的 Web 浏览器程序从套接字读取 HTTP 响应，从 HTTP 响应体中抽取 Web 网页的 html，并最终（终于！）显示了 Web 网页。

简化版：

1. DHCP 初始化（获取IP地址）

请求过程：Bob的便携机连接到网络时，因为没有IP地址，所以首先会向网络发送一个DHCP请求报文（目的端口67，源端口68），这是一个广播请求。

报文内容：报文包含了便携机的MAC地址，以0.0.0.0为源IP，目标IP为广播地址255.255.255.255。

DHCP服务器的响应：网络中的DHCP服务器接收到该请求，提供一个IP地址给Bob的便携机，同时提供子网掩码、默认网关、DNS服务器地址等信息。

IP分配确认：便携机从DHCP服务器收到一个包含分配的IP地址的DHCP ACK报文后，确认使用该IP。

2. DNS查询（解析域名）

请求过程：在获取IP地址后，Bob的便携机需要访问www.google.com，但它还不知道这个域名对应的IP地址，因此需要通过DNS进行域名解析。

DNS查询报文：便携机会发送一个DNS查询报文（UDP协议，目的端口53）到DNS服务器的IP地址，该报文包含了需要解析的域名。

DNS服务器的响应：DNS服务器会在收到查询报文后，返回相应的IP地址给Bob的便携机。

3. ARP请求（获取MAC地址）

ARP查询：为了与默认网关进行通信，Bob的便携机需要知道网关的MAC地址。此时便携机会发送一个ARP查询报文（目标MAC地址为广播地址FF:FF:FF:FF:FF:FF）到网络中，以请求网关的MAC地址。

ARP响应：网关收到ARP请求后，返回一个ARP响应报文，包含网关的MAC地址。便携机接收到该报文后，记录下网关的MAC地址。

4. TCP三次握手

TCP连接建立：在获取到www.google.com的IP地址后，Bob的便携机需要与该IP建立TCP连接。为了建立连接，便携机会首先发送一个TCP SYN报文。

三次握手过程：SYN报文到达目标服务器后，服务器会返回一个SYN-ACK报文；便携机收到SYN-ACK后，再发送一个ACK报文确认连接的建立。三次握手完成后，TCP连接正式建立。

5. HTTP请求和响应

HTTP请求：建立TCP连接后，Bob的便携机发送HTTP GET请求来请求网页内容，该请求报文包含了目标URL等信息。

服务器响应：目标服务器接收到HTTP请求后，处理请求并返回一个HTTP响应报文，其中包含所请求的网页内容。

数据传输和关闭连接：便携机接收到HTTP响应后，开始呈现网页内容。在数据传输完成后，TCP连接会通过四次挥手来关闭连接。

Socket编程基础

Socket定义

- socket简介：在网络编程中，socket是一个重要概念，用于描述IP地址和端口，是网络数据传输的端点。通过创建socket，计算机之间可以相互发送和接收数据，实现网络通信。
    - 端口是指网络通信中的一个虚拟通道，用于标识不同的服务或应用程序。每个端口都有一个唯一的号码，范围从0到65535。

    - 服务器监听特定的端口，通常是80端口（HTTP协议的默认端口）或443端口（HTTPS协议的默认端口）
        - 对于客户端和服务器之间的通信，默认会假设HTTP服务运行在80端口，而HTTPS服务运行在443端口。因此，如果浏览器访问 `http://example.com/` 或 `https://example.com/` 时，它会自动尝试连接到该域名对应的80端口或443端口。
- 功能：socket作为网络通信的基础，提供了建立网络连接、数据传输等功能。

Socket类型

- 流式Socket（SOCK_STREAM）
    - 用于创建TCP连接。
    - 保证数据完整性和顺序性。
    - 适用于要求可靠传输的应用，如Web服务器、文件传输等。
- 数据报Socket（SOCK_DGRAM）
    - 用于创建UDP连接。
    - 不保证数据的顺序和可靠性。
    - 适用于实时应用，如在线游戏、实时视频会议等。

TCP连接流程详解：

服务器端（Server）：

1. **创建套接字（create）**：通过系统调用 `socket()` 函数，指定协议类型（如AF_INET或AF_INET6）、传输层协议（如SOCK_STREAM表示TCP）来创建一个用于网络通信的套接字。
2. **绑定端口号（bind）**：使用 `bind()` 函数将套接字与特定的IP地址和端口号关联起来，这样客户端可以通过这个端口号找到并连接到服务器。
3. **监听连接（listen）**：调用 `listen()` 函数使服务器端的套接字进入被动监听状态，等待来自客户端的连接请求。
4. **接受连接请求（accept）**：当有新的连接请求时，`accept()` 函数会阻塞并返回一个新的套接字，该新套接字专门用来与发起连接的客户端进行数据交换。
5. **接收/发送数据（recv/send）**：通过返回的新套接字，服务器使用 `recv()` 和 `send()` 函数与客户端进行全双工的数据传输。
6. **关闭套接字（close）**：完成数据交互后，调用 `close()` 函数关闭已建立连接的套接字。

客户端（Client）：

1. **创建套接字（create）**：与服务器相同，客户端也需要先创建一个套接字。

别人家的考研Top帮

2. **发起连接请求（connect）**：客户端调用 `connect()` 函数主动向服务器发起连接请求，并提供服务器的IP地址和端口号。
3. **发送/接收数据（send/recv）**：一旦连接建立成功，客户端同样可以使用 `send()` 和 `recv()` 函数与服务器交换数据。
4. **关闭套接字（close）**：在数据交换完成后，客户端关闭其使用的套接字以释放资源。

UDP连接流程简介：

由于UDP是无连接的，因此没有“监听”和“接受连接”的概念。但仍有以下基本步骤：

服务器端（Server）：

1. **创建套接字（create）**：与TCP一样，首先创建一个UDP套接字。
2. **绑定端口号（bind）**：通过 `bind()` 函数将套接字绑定到特定的本地IP地址和端口上，以便接收来自客户端的消息。
3. **接收/发送消息（recvfrom/sendto）**：对于UDP，服务器和客户端都使用 `recvfrom()` 和 `sendto()` 函数直接读写数据，它们不仅负责数据传输，还需要处理源和目标地址信息。
4. **关闭套接字（close）**：在不需要继续接收或发送数据时，关闭套接字以释放资源。

客户端（Client）：

1. **创建套接字（create）**：同样创建一个UDP套接字。
2. **发送/接收消息（sendto/recvfrom）**：客户端可以直接使用 `sendto()` 向任意服务器发送数据，并使用 `recvfrom()` 接收从服务器或其他客户端发来的数据。
3. **关闭套接字（close）**：完成数据交互后，客户端关闭其套接字。

总结：

- TCP连接涉及到了三次握手建立连接、数据可靠传输及四次挥手断开连接的过程，而UDP则是无连接的，每个数据报文独立传输，不保证顺序和可靠性。
- UDP服务器和客户端均无需经历监听和接受连接阶段，而是直接通过指定对方地址进行数据报文的收发。

Socket编程流程

1. 创建Socket：使用 `socket()` 函数创建新的socket。

2. 绑定Socket：使用 `bind()` 函数将socket与特定的IP地址和端口号绑定，这样socket才能监听来自该地址和端口的网络请求。
3. 监听连接：对于服务端，使用 `listen()` 函数让socket进入监听状态，等待客户端的连接请求。
4. 接收连接：使用 `accept()` 函数接受客户端的连接请求。
5. 数据交换：使用 `send()` 和 `recv()` （或 `read()` 和 `write()` ）在连接的socket上发送和接收数据。
6. 关闭Socket：通信结束后，使用 `close()` 关闭socket。

创建和监听socket

- 示例代码解析：

```
#include <sys/socket.h>

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

- `domain` ：指定协议族，对于IPv4网络通信，通常设置为 `AF_INET` 。
- `type` ：指定套接字类型，对于面向连接的流式传输服务（如TCP），设置为 `SOCK_STREAM` 。
- `protocol` ：指定特定的协议编号，若为0，则会根据 `domain` 和 `type` 自动选择最合适的协议（对于 `AF_INET` 和 `SOCK_STREAM` ，系统会选择TCP协议）。

函数返回值是一个整数，代表新创建套接字的描述符。如果成功创建套接字，该描述符将用于后续的连接、绑定、接收和发送数据等操作；若失败，则返回-1，并可以通过 `errno` 获取错误代码。

- 定义地址并绑定：

```

    struct sockaddr_in address;         //用于互联网的地址结构。
    address.sin_family = AF_INET;       //设置地址族为IPv4
    address.sin_addr.s_addr = INADDR_ANY;//允许服务器接受发往本机任何IP的请求
    address.sin_port = htons(PORT);     //设置端口号，htons确保端口格式正确。
    //将服务器 socket server_fd 绑定到 address 所代表的地址和端口上，使服务器能够监听来自该地址和端口的连接请求
    bind(server_fd, (struct sockaddr *)&address, sizeof(address));
    listen(server_fd, 3);  //让socket进入被动监听状态。
```

  - `sockaddr_in` ：用于互联网的地址结构。
  - `sin_family` ：设置地址族为IPv4。
  - `sin_addr.s_addr` ：允许服务器接受发往本机任何IP的请求。

- `sin_port`：设置端口号，`htons`确保端口格式正确。

1. bind() 函数：`bind()` 函数的作用是将指定的套接字与给定的本地地址进行关联，也就是为服务器端的套接字分配一个本地IP地址和端口号。这样当客户端发起连接请求时，就知道应该连到哪个IP地址和端口了。

```
1 bind(server_fd, (struct sockaddr *)&address, sizeof(address));
```

   - `server_fd`：是一个已创建好的套接字文件描述符，通过调用 `socket()` 函数得到。
   - `(struct sockaddr *)&address`：是指向一个已初始化的 `sockaddr_in` 结构体（或更通用的 `sockaddr` 结构体）的指针。这个结构体包含了服务器要绑定的本地地址信息，包括IP地址、端口号等。
   - `sizeof(address)`：表示上述结构体的大小。

- 监听网络请求：

```
1 listen(server_fd, 3);
```

   - `listen()`：让socket进入被动监听状态。
   - `3`：等待连接队列的最大长度。

   具体来说，这里的数字表示服务器可以同时等待的连接请求的最大数量。当服务器正在处理一个连接请求时，如果有其他客户端的连接请求到达，它们将被放入等待队列中，直到服务器有空闲的资源来处理它们。

接收连接

- 过程：
   - 使用 `accept()` 函数等待并接受客户端连接请求。
   - `new_socket`：新的socket描述符，专门用于与连接的客户端通信。

   accept() 函数：

```
1 int new_socket = accept(server_fd, (struct sockaddr *)&client_address,
  &client_len);
```

考研Top帮

在服务器端调用 `listen()` 函数进入监听状态后，接下来使用 `accept()` 函数等待并接受客户端的连接请求。`accept()` 会阻塞直到有新的客户端连接到来。

- `server_fd`：之前创建并进入监听状态的服务器套接字描述符。
- `(struct sockaddr *)&client_address`：是一个指向 `sockaddr_in` 或 `sockaddr` 结构体的指针，用于存储发起连接的客户端的地址信息。
- `&client_len`：是指向一个整数的指针，用于存储实际返回的客户端地址结构体的大小。

当一个新的客户端连接成功建立时，`accept()` 函数会返回一个新的套接字描述符 `new_socket`，这个新描述符专用于与该客户端进行

处理HTTP请求

- 读取请求：
  - 使用 `read()` 函数从socket中读取数据。
  - `buffer`：定义为一个足够大的字符数组（缓冲区），用于临时存储从socket中读取的客户端数据。
  - `BUFFER_SIZE`：缓冲区大小，需要确保它足以容纳一个完整的HTTP请求头。
  - `read()` 函数第三个参数通常减去1，以保证在读取到的数据末尾添加结束符 '\0'。

`read()` 函数将从指定的套接字中读取数据，并将其存入缓冲区 `buffer` 中。返回值 `bytes_received` 表示实际读取到的字节数。根据读取到的数据内容，服务器可以解析出HTTP请求方法、URL、HTTP版本等信息，然后执行相应的服务逻辑。

发送HTTP响应

- 响应方法：
  - 使用 `write()` 或 `send()` 函数向客户端发送响应。
  - `response`：响应内容，包括HTTP状态码和响应体。

完整示例代码

- 代码实现：

```
#include <iostream>  // 引入标准输入输出库
#include <map>       // 引入标准映射容器库
#include <functional>// 引入函数对象库，用于定义函数类型
#include <string>    // 引入字符串处理库
#include <sys/socket.h> // 引入socket编程接口
#include <stdlib.h>  // 引入标准库，用于通用工具函数
#include <netinet/in.h> // 引入网络字节序转换函数
#include <string.h>  // 引入字符串处理函数
#include <unistd.h>  // 引入UNIX标准函数库
```

```cpp

// 测试命令 curl http://localhost:8081/register
#define PORT 8080  // 定义监听端口号为8080

using RequestHandler = std::function<std::string(const std::string&)>; // 定义请求处理函数类型

std::map<std::string, RequestHandler> route_table; // 定义路由表，映射路径到对应的处理函数

// 初始化路由表
void setupRoutes() {
    // 根路径处理
    route_table["/"] = [](const std::string& request) {
        return "Hello, World!";
    };

    // 注册处理
    route_table["/register"] = [](const std::string& request) {
        // TODO: 实现用户注册逻辑
        return "Register Success!";
    };

    // 登录处理
    route_table["/login"] = [](const std::string& request) {
        // TODO: 实现用户登录逻辑
        return "Login Success!";
    };

    // TODO: 添加其他路径和处理函数
}

int main() {
    int server_fd, new_socket; // 声明服务器socket和新连接的socket
    struct sockaddr_in address; // 声明网络地址结构
    int addrlen = sizeof(address); // 获取地址结构的长度

    // 创建socket
    server_fd = socket(AF_INET, SOCK_STREAM, 0);

    // 定义服务器地址
    address.sin_family = AF_INET;          // 设置地址族为IPv4
    address.sin_addr.s_addr = INADDR_ANY; // 绑定到所有可用网络接口
    address.sin_port = htons(PORT);        // 设置服务器端口号

    // 将socket绑定到地址
    bind(server_fd, (struct sockaddr *)&address, sizeof(address));
```

M学长的考研TOP帮

```
55
56      // 监听端口上的网络请求
57      listen(server_fd, 3);
58
59      // 初始化路由表
60      setupRoutes();
61
62      while (true) {
63          // 接受来自客户端的连接
64          new_socket = accept(server_fd, (struct sockaddr *)&address,
    (socklen_t*)&addrlen);
65
66          // 读取客户端的请求数据
67          char buffer[1024] = {0};
68          read(new_socket, buffer, 1024);
69          std::string request(buffer);
70
71          // 解析请求的URI
72          std::string uri = request.substr(request.find(" ") + 1);
73          uri = uri.substr(0, uri.find(" "));
74
75          // 根据路由表处理请求并生成响应
76          std::string response_body;
77          if (route_table.count(uri) > 0) {
78              response_body = route_table[uri](request);
79          } else {
80              response_body = "404 Not Found";
81          }
82
83          // 向客户端发送HTTP响应
84          std::string response = "HTTP/1.1 200 OK\nContent-Type: text/plain\n\n"
    + response_body;
85          send(new_socket, response.c_str(), response.size(), 0);
86
87          // TODO: 实现多线程处理来提高性能
88          // TODO: 添加日志系统以记录请求和响应
89          // TODO: 实现更完善的错误处理机制
90
91          // 关闭连接
92          close(new_socket);
93      }
94
95      return 0;
96  }
97
```

学长的考研TOP帮

编译和运行

- 步骤：
  - 使用g++编译：`g++ -o server server.cpp`
  - 运行服务器：`./server`

测试服务器

- 测试方法：
  - 使用浏览器或curl工具测试：`http://localhost:8080` 或 `curl http://localhost:8080`

M学长的考研Top帮