Chapter 11 网络编程

- **客户端-服务器模型**: 由一个 `服务器进程` 和一个或多个 `客户端进程` 组成.

  > 注意这里说的是 `进程` , 而不是 `主机` . 因为一台主机可以同时运行多个客户端或服务器; 一个事务也可以在不同的主机上.
  > 而无论客户端和服务器如何映射到主机上, 这个模型都是相同的.

  - **事务(Transaction)**: 客户端-服务器模型中的基本操作. 是客户端和服务器执行的一系列步骤.
  - **请求**: 客户端需要服务时, 向服务器发送请求.
  - **响应**: 服务器给客户端发送一个响应, 然后等待下一个请求.
- 对主机而言, 网络只是一个单纯的I/O设备, 作为数据源和数据接收方. 通常采用DMA传输.
- **适配器(Adapter)**插在I/O总线扩展槽中, 为主机提供了到网络的 `物理接口` .
- **局域网(Local Area Network, LAN)**
- **以太网(Ethernet)**: 当今最流行的一种局域网技术.
- **以太网段(Ethernet Segment)**: 由 `线缆` 和**集线器(Hub)**组成. 线缆的一端连接到主机的 `适配器` 上, 另一端连接到集线器的一个 `端口` 上.
  `集线器` 不加分辨地将从一个端口上接收到的所有位复制到所有其它的端口上, 于是每台主机都能看到每个位.
- 每个 `以太网适配器` 都有一个全球唯一的48位的地址. (cpy: 这里的 `以太网适配器` 应该就是上文提到的 `适配器` 的以太网版本)
- 一台主机可以发送 `一段位(a chunk of bits)` (叫做 **`帧(frame)`**)到这个 `以太网段` 内的其它任何主机.
  每个 `帧` 包含固定长度的 `头部位(header bits)` , 用于标识这个帧的 `源` , `目的地址` 以及 `帧长度` .
  `头部位` 后紧随着数据位的 `有效载荷(payload)` .
  每个主机适配器都可以看到这个帧, 不过只有目的主机会实际读取这个帧.
- **网桥(Bridge)**: 使用电缆和网桥, 多个 `以太网段` 可以连接成更大的 `局域网` , 称为**桥接以太网(bridged Ethernet)**.
  其中, 一些电缆负责连接 `网桥` 与 `网桥` , 另一些电缆则连接 `网桥` 和 `集线器` .
  与 `集线器` 不同, `网桥` 不会(无脑地)把帧发送到所有的端口, 而是会通过一些分配算法, **有选择地**将帧从一个端口复制到另一个端口.

  > 为了简化桥接以太网的表示, 我们可以把 `集线器` 和 `网桥` 以及连接它们的 `线缆` 抽象成一根水平线, 所有的主机都连载这根线上.

- *多个不兼容的局域网* 可以通过**路由器(Router)**连接起来, 组成一个**互联网络(internet)**.
  `路由器` 对它连接到的每个网络都有一个适配器(端口).
  `路由器` 可以由各种 `局域网` 和 `广域网` 构建 `互联网络` .

- **广域网(Wide-Area Network, WAN)**
- 运行在 `主机` 和 `路由器` 上的**协议软件(protocol software)**: 让 `源主机` 跨越 `不兼容的网络` 发送数据到另一台 `目的主机` .

  协议必须提供两种基本能力:
  - 命名机制: 每台主机会被分配至少一个 `互联网络地址(internet address)` , 唯一标识这台主机.
  - 传送机制: 数据位被按照 `统一的方式` 封装成**包(packets)**, 消除局域网技术间的差异.

    一个 `包` 由**包头(header)**和**有效载荷(payload)**组成. `包头` 包括包的大小, 以及源主机和目的主机的地址.
- 基于 `互联网络协议` 在 `不兼容的局域网之间` 传送数据的流程:

  假设 `主机A` 想传送数据给另一个局域网的 `主机B` .

  主机A先获得要传输的数据, 然后主机A上的协议软件会给数据先后附加 `互联网络帧头` 和 `LAN1帧头` . 从而创建了一个LAN1的 `帧` .

  > 值得指出, 这里是对数据的两层封装:
  > `LAN1帧头` 被 `适配器` 用于在 `局域网LAN1` 内寻址到 `路由器` , 是适配器要用的;
  > `互联网络帧头` 则被 `路由器` 用于在 `互联网络` 中寻址到 `主机B` , 是路由器要用的.

  当这个 `帧` 到达路由器时, 路由器从它的LAN1适配器读取这个帧, 并传送到自己的协议软件中,
  从中提取目的 `互联网络地址` , 从而确定应该转发到 `局域网LAN2` , 并剥落 `LAN1帧头` , 加上 `LAN2帧头` ,
  然后把新构造的 `帧` 传送到路由器的 `LAN2适配器` , 然后复制该帧到网络上, 最终被 `主机B` 的适配器接收, 传送到主机B的协议软件.
  主机B的协议软件剥落 `包头` 和 `帧头` , 最终得到了数据.
- 上面的流程告诉了我们互联网思想的精髓: **封装**.

- **全球IP因特网(The Global IP Internet)**
- **TCP/IP协议**: 传输控制协议 / 互联网络协议.

  为了简化讨论, 我们把 `TCP/IP` **看做**一个**整体**协议, 讨论它为应用程序员提供的功能.
  包括命名方法, 递送机制, 提供进程间可靠的 `全双工连接` .
- 从程序员角度, 因特网可以被看做一个世界范围的 `主机集合` , 满足:
  - `主机集合` 被映射到一组32位的 `IP地址` ;
  - 这组 `IP地址` 被映射到一组**因特网域名(Internet Domain Name)**;
  - 主机间的 `进程` 可以通过 `连接` 通信.
- **IP地址**: 32位无符号整数.
  - 按照**网络字节顺序**(大端法)存放. 即**无论主机字节顺序**如何, IP地址**在内存中总是以大端法存放**的.

    `ntohl/htonl/ntohs/htons` 函数用于 `主机字节(Host)` 和 `网络字节(Network)` 顺序的转换, 有32位(long)和16位(short)版本.
  - 通常以**点分十进制表示法**表示, 比如地址 `0x8002c2f2` 被表示为 `128.2.194.242`

`inet_pton/inet_ntop` 函数用于 `IP地址` 和 `点分十进制字符串` 之间的转换. p表示presentation, n表示network.

- 在C语言中, IP地址的 `类型名` 为一个叫做 `in_addr` 的结构体. 其中 `s_addr` 在内存中一定是大端法存放.

```
struct in_addr {
    uint32_t s_addr;    // Big-endian
};
```

> 事实上, 为IP地址单独定义一个标量类型其实更为合适, 不过由于历史遗留原因, IP地址被存放在这样一个结构体中.

- **域名(Domain Name)**: `客户端` 和 `服务器` 通信使用的 `IP地址` 对于人类而言难以记忆.
  - **一级域名**: 如 `com` , `edu` , `gov` , `org` , `net`
  - **二级域名**: 如 `cmu.edu`

    一旦一个组织得到了一个 `二级域名` , 就可以在这个**子域**中创建任何新的域名了, 比如 `cs.cmu.edu`
  - 所以域名的层次结构是倒着看的.
- 因特网提供将 `域名` 映射到 `IP地址` 的机制.

  这是通过世界范围分布的数据库**DNS(Domain Name System, 域名系统)**来维护的.

  DNS数据库包含上百万个 `主机条目结构(host entry structure)` ,
  - 每个条目定义了 `一组域名` 和 `一组IP地址` 之间的映射, 如:

    `twitter.com` 和 `www.twitter.com` 映射到 `199.16.156.6` 和 `199.16.156.70` 和 `199.16.156.102` .
  - 某些合法的 `域名` 并没有映射到任何 `IP地址` , 如 `edu` , `ics.cs.cmu.edu` 等等.
- **回送地址(Loopback Address)**: 每台因特网主机都有本地定义的域名 `localhost` , 映射到 `回送地址` : `127.0.0.1` .
- **连接** : 因特网上的 `客户端` 和 `服务器` 通过在 `连接` 上发送和接收 `字节流` 来通信. 具有以下性质:
  - **点对点**(point-to-point): 连接一对**进程**. (~~注意不是 主机~~ ; 而是主机上的一个 `进程`)
- **全双工**(full duplex): 数据可以同时双向流动 (`半双工` 如电视机, 是单向接收的)
  - **可靠的**(reliable): 从 `源进程` 发出的 `字节流` 会被 `目的进程` 以 `发出时的顺序` 收到.

    (与之相对, `不可靠的` 如网络视频, 某一小块数据丢失了, 显示为花屏, 也并不会有太大的影响.)

- **套接字(Socket)**:
  - 在内核看来, `套接字` 是 `连接` 的一个端点. 每个套接字有一个 **`套接字地址`** , 由 `IP地址` 和16位整数 `端口(号)` 组成: `地址:端口` .

    (注意这里的端口是 `软件端口` , 而非 `硬件端口`)
  - 在Linux程序看来, `套接字` 就是一个(有相应描述符的) `打开文件` .

    "Clients and servers communicate with each other by reading from and writing to socket descriptors."

> 来自网络: 套接字这个词令人迷惑. 英文Socket意即插座. 服务器就像一个大插排, 包含很多插座, 客户端就是像一个插头, 每一个线程代表一条电线, 客户端将电线的插头插到服务器插排上对应的插座上, 就可以开始通信了.

- 当 **客户端** 发起一个连接请求时, 客户端套接字地址 中的端口是由内核自动分配的, 称为**临时端口(ephemeral port)**.
  而 **服务器** 的 套接字地址 中的 端口 则**通常**(暗示可以修改)是某个**知名端口(well-known port)**.
  比如: Web服务器 通常使用端口 80 , 对应的**知名名字**为 http ; email 通常使用端口 25 , 对应 smtp (Simple Mail Transfer Protocol).

- **套接字对(Socket Pair)**: 一个 连接 由它两端的 套接字地址 唯一确定. 具有格式: `(cliaddr:cliport, servaddr:servport)`
- **套接字地址(sockaddr_in)** 结构体组织形式:

```
struct sockaddr_in{                             // "in" for "internet", 即<因特网应用>的套接字地址结构
    uint16_t                sin_family;     // 协议族(Protocol Family)(Always AF_INET)
  uint16_t              sin_port;           // 端口号(Port number)
  struct in_addr    sin_addr;           // IP地址
  unsigned char     sin_zero[8];    // 对齐用, 为了和 struct sockaddr 对齐.
};                                                      // (stuct sockaddr 是通用结构体, 下面会讲)
```

```
struct sockaddr{                                // <通用的>套接字地址结构
    uint16_t                sa_family;      // 协议族(Protocol Family)
  char                      sa_data[14];    // Address Data
};
```

  sockaddr 这种 通用结构体 的出现是出于下述原因:
  - 套接字接口函数需要传入一个 套接字地址结构 的指针, 然而有各种 不同类型的 套接字地址结构.
  - 那个年代还没有 void* 指针, 所以只好用 sockaddr* 充当 void* 的功能, 让 套接字接口 可以接收所有类型的套接字地址结构.
  - 所以在 connect / bind / accept 这些 套接字接口 函数中, 套接字地址 参数的类型都是 sockaddr* 的.

- **套接字接口(Socket Interface)**: 一组可以和 UnixI/O 结合起来, 用于创建网络应用的函数.
  - **socket** : **客户端和服务器**用 socket 来创建一个 套接字描述符 .

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

    - socket 返回的描述符还不能用于读/写. 因为内核还不知道你是作为客户端还是服务器.
  - **connect** : **客户端**用 connect 与服务器建立连接.

```
int connect(int clientfd, const struct sockaddr* addr, socklen_t addrlen);
```

- `connect` 函数试图和地址为addr的服务器建立一个 `因特网链接` .
- `connect` 函数会阻塞, 直到连接成功或发生错误.
- 如果连接成功, 那么 `clientfd` 描述符就可以用于读写了.

- **`bind`** : **服务器**用 `bind` 将一个 `服务器套接字地址` 和一个 `套接字描述符` 联系起来.

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

- **`listen`** : **服务器**调用 `listen` 函数, 告诉内核描述符是被服务器使用的.

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

- 我们要显式地 `listen` , 是因为 `socket` 函数返回的描述符被内核**默认**为 `主动套接字` , 即被客户端使用.
- `listen` 将这个套接字转化为 `监听套接字` . 可以接收来自客户端的连接请求.
- `backlog` 参数告诉内核在拒绝连接请求之前, 队列中排队的连接请求的最大数量. 默认为1024.

- **`accept`** : **服务器**用 `accept` 函数等待来自客户端的连接请求. 返回时, 返回一个 `已连接描述符` .

```
int accept(int listenfd, struct sockaddr* addr, int* addrlen); // 返回值: 已连接描述符
```

- `服务器` 可以通过这个描述符, 利用Unixl/O实现与 `客户端` 通信.
- `accept` 函数等待客户端的连接请求到达 `监听描述符listenfd` , 然后在addr中填写客户端的套接字地址.
- 返回值为一个 `已连接描述符` .

- **`getaddrinfo()`** 函数:

  将 `<主机名, 主机地址, 服务名/端口号> 的字符串表示` , 转化为 `套接字地址结构` (的封装结构体(`struct addrinfo`)).

```
int getaddrinfo(const char* host, const char* service,
                const struct addrinfo* hint,
                          struct addrinfo** result);     // 参数 result 用于存放返回的链表.
```

  其中, `addrinfo` 的定义如下:

```
struct addrinfo{
    int                             ai_flags;
  int                           ai_family;      // 用作 socket 的第 1 个参数.
  int                           ai_socktype;    // 用作 socket 的第 2 个参数.
  int                           ai_protocol;    // 用作 socket 的第 3 个参数.
  char*                         ai_canonname;
  size_t                        ai_addrlen;
```

```
  struct sockaddr*  ai_addr;            // 指向 sockaddr 结构的指针
  struct addrinfo*  ai_next;            // 指向<链表>中下一个元素的指针
};
```

- `result` 参数用于存放返回的链表.
  - 当客户端调用 `getaddrinfo` 之后, 客户端可以遍历这个链表, 依次尝试每个套接字地址, 直到调用 `socket` 和 `connect` 成功.
  - 当服务器调用 `getaddrinfo` 之后, 服务器同样会尝试遍历链表, 依次尝试每个套接字地址, 直到调用 `socket` 和 `bind` 成功.
- `host` 参数可以是域名, 也可以是数字地址(如点分十进制IP地址).
- `service` 参数可以是服务名(如http), 也可以是端口号(如80).
- `hints` 参数是可选的, 提供一些定制功能.

  注意, `hints` 的类型也是 `addrinfo` , 不过我们只能设置其中4个字段, 其余必须置0.
  - `ai_family` 字段设置为 `AF_INET` 会把列表限制为 `IPv4` 地址.
- `ai_socktype` 字段设置为 `SOCK_STREAM` 会将列表限制为对每个地址最多一个 `addrinfo` 结构. (本书讨论范围内必须设置)
  - `ai_flags` 字段是一个掩码.

    如果设置 `AI_PASSIVE` , 告诉函数: 返回的 `套接字地址` 会被服务器用作 `监听套接字` .

    此时, 参数 `host` 必须设置为 `NULL` , 且得到的 `套接字地址结构` 中的 `地址字段` 将是 `通配符地址(wildcard address)` .

    这告诉内核: 这个服务器会接受发送到这个主机的所有IP地址的请求.

- **`getnameinfo()`** 函数:

  和 `getaddrinfo` 相反, `getnameinfo` 函数将一个 `套接字地址结构` 转换为相应的 `<主机和服务名>字符串` .

```
int getnameinfo(const struct sockaddr* sa, socklen_t salen,
                char* host,      size_t hostlen,     // host[hostlen]缓冲区用于存放返回值
                char* service, size_t servlen,   // service[servlen]缓冲区用于存放返回值
                int flags);
```

- `flags` 参数同样是一个掩码, 两个有用的设置:
  - `NI_NUMERICHOST` : 设置该函数返回的 `host` 为 `数字地址字符串` . (默认情况下, 会返回 `域名`)
  - `NI_NUMERICSERV` : 设置返回的 `service` 为 `端口号字符串` . (默认情况下, 会检查 `/etc/services` , 并尽可能返回 `服务名`)

- **套接字接口的包装函数**: `open_clientfd` 和 `open_listenfd` , 用于客户端和服务器相互通信.

```
int open_clientfd(char* hostname, char* port);  // 客户端用
```

- **功能**: 客户端用于建立与服务器的连接, 这个服务器运行在主机 `hostname` 上, 并在端口号 `port` 上监听连接请求.
  它返回一个打开的套接字描述符, 并且这个描述符已经可以直接被UnixI/O用于输入和输出了.
- **实现**: 调用 `getaddrinfo` , 我们就得到了一个链表, 里面存放的是一组潜在的可用于连接的服务器地址.
  然后我们依次尝试进行 `socket()` 和 `connect()` , 直到成功返回 `clientfd` , 或者全部失败返回 `-1` .

```
1  int open_listenfd(char* port);  // 服务器用
```

- **功能**: 服务器用于创建一个监听描述符. 这个描述符准备好在端口port上接受连接请求.
- **实现**: 以 `host` 为 `NULL` , 并设置 `flags` 为 `AI_PASSIVE` 来调用 `getaddrinfo` ,
  然后对得到的链表元素依次尝试 `socket` , `bind` , `listen` , 返回 `listenfd` 或者 `-1` .

- **HTTP(Hypertext Transfer Protocol, 超文本传输协议)**.
- **Web内容**: 可以用**HTML(Hyper Markup Language, 超文本标记语言)**来编写.
- **内容**: 与**MIME类型**相关的字节序列.
  常用的 `MIME类型` 包括: HTML页面, 无格式文本, PNG/JPG/GIP格式二进制图像.
- 服务**静态内容(Static Content)**: 取一个磁盘文件, 并把它的内容返回给客户端.
- 服务**动态内容(Dynamic Content)**: 运行一个可执行文件, 并将它的输出返回给客户端.
- **URL(Universal Resource Locator, 通用资源定位符)**: 与Web服务器返回的 `内容` 相关联的 `文件` 的唯一的名字.
  比如 `http://www.google.com:80/index.html` 表示因特网主机 `www.google.com` 上一个称为 `/index.html` 的 `HTML` 文件,
  这个文件由监听端口为80的Web服务器管理.

  > `URI` 和 `URL` 的关系: `URL` 是 `URI` 的一种实现方式, 特别地, URL是用 `文件的地址` 来唯一地区分某个文件.

- 可执行文件的URL可以在文件名后包括 `程序参数` . `?` 字符用于分隔 `文件名` 和 `参数` , `参数` 之间用 `&` 字符分隔.
  比如 `http://bluefish.ics.cs.cmu.edu:8000/cgi-bin/adder?15000&213` 的参数为 `15000` 和 `213` .

- **HTTP请求(Requests)**: (我们只讨论GET方法): 由一个 `请求行` , 零个或多个 `请求报头` , 一个 `空的文本行` 组成.
  - **请求行(Request Line)**: 格式为 `method URI version` . 例如: `GET / HTTP/1.1` .
  - **请求报头(Request Header)**: 格式为 `header-name: header-data` . 例如: `Host: www.aol.com` .
    `Host报头` 中的数据指示了原始服务器的域名, `代理链` 中的 `代理` 可以借此判断被请求的内容的副本是不是已经被缓存了.
- **HTTP响应(Responses)**: 由一个 `响应行` , 零个或多个 `响应报头` , 一个 `空的文本行` , 一个 `响应主体` 组成.
  - **响应行(Response Line)**: 格式为 `version status-code status-message` . 比如: `HTTP/1.0 200 OK` .
  - **响应报头(Response Header)**: 我们主要关注 `content-Type` 和 `content-Length` . 即 `MIME类型` 和 `响应主体的大小(字节)` .
  - **响应主体(Response Body)**

- **CGI(Common Gateway Interface, 通用网关接口)**.
- **Q_1: 客户端如何将程序的参数传递给服务器?**
  - `GET` 请求的参数在URI中传递: `?` 分隔文件名和参数, 参数之间用 `&` 分隔. 参数中不允许出现空格, 必须用 `%20` 编码. 特殊字符同理.
  - `POST` 请求的参数是在 `请求主体` 中传递, 不过注意在 `GET` 中, 是没有 `请求主体` 这个东西的.
- **Q_2: 服务器如何将参数传递给子进程?**
  - 当服务器接收到比如请求 `GET /cgi-bin/adder?15000&213 HTTP/1.1` 时,
    首先调用 `fork` 来创建一个子进程, 然后在子进程中调用 `execve` 执行 `/cgi-bin/adder` 程序.
    这种程序被称为**CGI程序**. 而这个CGI程序的参数由**环境变量** `QUERY_STRING` 传递.
    比如在本例中, `QUERY_STRING` 会被设置为 `"15000&213"` .
    `adder` 程序在运行时, 可以通过 `getenv` 函数来引用这个参数.
- **Q_3: 服务器如何把其他信息传递给子进程?**
  - 还是通过环境变量!
- **Q_4: 子进程把它的输出发送到哪里?**
  - CGI程序本身只是单纯把内容输出到标准输出
  - 这就要求服务器在 `execve` 这个 `CGI程序` 之前, 先得调用 `dup2` 函数把 `标准输出` 重定向到和客户端相关联的 `已连接描述符` .
  - 父进程不知道子进程生成的内容的 `类型` 和 `大小` ,
    所以子进程负责输出 `Content-type` 和 `Content-length` 响应报头, 以及终止报头的 `空行` .
  - 在 `POST` 请求中, 子进程的 `标准输入` 也需要重定向到已连接描述符. 因为 `CGI程序` 会从 `标准输入` 中读取 `请求主体` 中的参数.

- **代理(Proxy)**
-