版本号：v1.0 阶段：初稿

# WebServer项目文档

MRL Liu

（未经许可，不得传播）

2022年06月01日

# 第三方工具篇

## 一、Linux的命令解释器

**命令解释器**比图形界面执行要快。Linux系统提供了多种**命令解释器**，如**shell**（/bin/sh），如bash（/bin/bash）。其中**Bash解释器**就是Linux中**终端**默认的**命令解释器**。

### Bash解释器常用快捷键

|  |
| --- |
| **Tab键**：补齐命令、补齐路径、显示当前目录下的所有目录（按两下） **Ctrl+L**：清屏，等价于输入“clear”。 **Ctrl+C**：中断进程，将当前正在运行的命令中断。 遍历历史命令：ctrl+p（向上），ctrl+n（向下） 移动光标命令：  光标左移：ctrl+b（向左） 光标右移：ctrl+f（向右）  移动到头部：ctrl+a（home） 移动到尾部：ctrl+e（end） |

### 2、Linux命令格式

command [-option][pararmeter]

查看**内建命令（Linux内置命令）**和**外部命令（第三方程序命令）**的区别：type -a 命令

## 二、Linux的程序编译器

Linux的**GCC编译器**（GNU Cimpiler Collection，GNU编译器套件）是由GNU开发的编程语言编译器，现已被大多数类UNIX操作系统（如Linux、BSD、Max OS X等）采纳为标准编译器，同样存在Windows版本的GCC编译器。GCC最初用于编译C语言，现在已经能够编译C、C++、Java、Go等主流语言。

查看GCC版本：gcc -V / --version

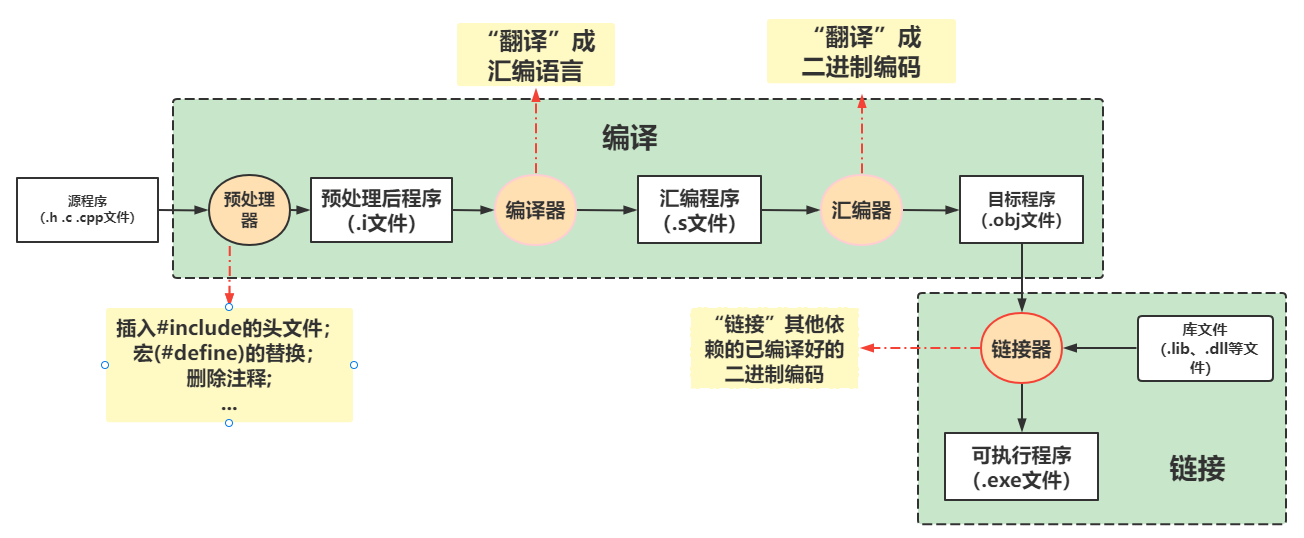
### 1、GCC的工作流程

GCC编译器的编译命令：

|  |
| --- |
| 常见编译格式： **gcc hello.c -o hello** //将hello.c编译为为hello.out gcc **-g** hello.c -o hello //将hello.c编译为为hello.out,包含调试信息 等价于四步编译指令： gcc **-E** hello.c -o hello.i //将hello.c**预处理**为hello.i gcc **-S** hello.i -o hello.s //将hello.i**编译**为hello.s(汇编文件) gcc **-c** hello.s -o hello.o //将hello.s**汇编**为hello.o(二进制文件) gcc hello.o -o hello //将hello.o**链接**为hello.out(linux的可执行文件) 其他编译指令： gcc -On gcc -Wall hello.c//显示所有警告信息 gcc -Wall -Werror hello.c //将所有警告信息当作错误处理 gcc hello.c -DDEBUG //编译时定义宏（-D后跟着DEBUG） |

### 2、C++的编译流程

在开发过程中，程序员写的C++源程序文件（.cpp）都要被IDE中集成的**C++编译器**（g++）编译生成.exe文件，C++编译器就完成了C++的编译流程。C++的编译流程总体可以分为两个步骤：**编译**和**链接**，其过程如下：



## Linux的项目编译工具

**MakeFile**是**Linux系统**专用的**项目编译工具**。它提供了一种**Makefile文件**，在该文件中可以定义整个项目的编译规则。**Makefile**文件的命名规则必须是**无后缀**的MakeFile，**大小写均可**，但是推荐使用MakeFile。

在含有MakeFile的目录下，执行命令**make**即可按照MakeFile的规则编译当前项目。

Make的工具的安装：sudo apt install make.

MakeFile的作用是**定义项目编译规则**，**实现项目的自动化编译**。一个项目目录中一般存在多个源文件（.c/.cpp）和相关目录，不同文件之间的编译顺序也不相同，如果手动编译太过繁琐，所以可以直接将编译规则写入MakeFile文件。

## Linux的压力测试工具

**Web bench**（[网页工作台](http://home.tiscali.cz/~cz210552/webbench.html)）是一款**Linux系统**上用**C语言编写**的专用的**轻量级**的**网站压力测试工具**，最多可以对一个网站模拟3w左右的并发请求，对**中小型网站**有明显的效果。

### 1、Web bench的安装

**web bench**是一个**压缩包**（webbench-1.5.tar.gz），可以手动从[网页工作台](http://home.tiscali.cz/~cz210552/webbench.html)下载，也可以使用**命令行**下载：

|  |
| --- |
| wget http://www.ha97.com/code/webbench-1.5.tar.gz # 用wget下载webbench-1.5.tar.gz tar xf webbench-1.5.tar.gz # 解压webbench-1.5.tar.gz中的所有文件 yum install gcc\* ctags\* -y # 安装gcc编译环境 make && make install # 编译并且安装编译程序 |

### 2、Web bench的使用

在**终端**使用如下**命令行**启动测试：

|  |
| --- |
| webbench **-c** 100 **-t** 30 http://127.0.0.1:1316/ |

其中-c表示**设置客户端访问的数量**，可省略，默认为1；-t表示设置**客户端访问的时间**，可省略，默认为30s；

得到立即反馈：

|  |
| --- |
| Benchmarking:GET http://127.0.0.1:1316/ 100 clients,running 30 sec. |

执行完毕后得到类似反馈：

|  |
| --- |
| Speed=513172 pages/min,27651526 bytes/sec. Requests: 256586 susceed,0 failed. |

使用的注意事项：

1）webbench做压力测试时，**自身也会消耗CPU和内存资源**，为了测试准确，请将webbench安装在别的服务器上。

2）**测试时并发应当由小逐渐加大**，比如并发100时观察一下网站负载是多少、打开页面是否流畅，并发200时又是多少、网站打开缓慢时并发是多少、网站打不开时并发又是多少。

## 五、Linux的HTTP分析工具

# Web bench篇

## 一、Web bench的工作流程

webbench-1.5.tar.gz中的文件大致分为如下三类：安装好后，就会得到如下3个

* **源码文件**：socket.c、webbench.c
* **脚本文件**：MakeFile
* **编译文件**：webbench、webbench.o

其中**脚本文件**MakeFile表明如何编译源码，**编译文件**是通过**脚本文件**MakeFile编译后生成的，即在**终端**切换到该文件夹所在路径，执行**make**，就会生成webbench、webbench.o文件。可以看到，该工具十分简单，源码只有两个文件：socket.c、webbench.c。其程序执行的**工作流程**如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| int **main**(int **argc**,char\* **argv**[]){  1、处理**终端**输入的**命令行参数**，检查参数是否合法  2、调用**build\_request()**，输入**URL路径**，构造一个**HTTP请求**，存储在全局字符数组**request**中  3、打印创建**HTTP请求**信息,例如"Benchmarking:GET http://127.0.0.1:1316/"等  4、调用**bench()**:   |  |  |  |  | | --- | --- | --- | --- | | 1、尝试创建**Socket对象**连接服务端程序 2、尝试创建**匿名管道**pipe。 3、通过**fork()**创建client个**子进程**，以下是每个**子进程**执行流程：   |  |  |  | | --- | --- | --- | | 1、在每个子进程中，调用**benchcore()**：   |  |  | | --- | --- | | 1、设置闹钟**alarm(benchtime)**，达到规定时间后产生**SIGALRM信号**，执行timerexpired=1 2、进入死循环while(1)：   |  | | --- | | 1、检查timerexpired是否为1，若为1，退出循环 2、创建一个**Socket对象**连接**服务端程序**，更新访问结果 3、借助**Socket对象**向服务端发送**HTTP请求**，更新访问结果 4、接收**服务端**返回的**HTTP相应数据**，更新访问结果 | |   2、将**访问结果**通过**fdopen(,"w")**写入**匿名管道** |   4、通过**fdopen(,"r")**从**匿名管道**中读取**子进程**写入的数据，累计全部数据 5、打印结果数据 |   } |

**socket.c**中只有一个方法Socket()，负责建立与目标的TCP连接，并返回客户端连接使用的套接字；**webbench.c**完成了压力测试的主要功能**：**对命令行参数的处理，构建**HTTP请求**，创建**子进程**进行**压力测试**等。

## 创建N个相同子进程

Web bench是创建了N个**子进程**，每个子进程执行**相同的代码**，我们分析下创建**子进程**的实现。

### 1、fork()的使用

Linux中通过**fork()**来创建子进程，**在执行fpid=fork()之前，只有当前的父进程在执行这段代码；在执行fpid=fork()之后，当前的父进程和新复制出的子进程都开始并发执行之后的代码**。所以为了父进程和子进程执行不同的代码，需要根据fpid的值分别处理，否则就会执行相同的代码。子进程的fork返回值是0，父进程的fork返回值>0，这也是区分**父进程**和**子进程**的方法，至于其他的内容，在fork之前的东西两个进程的一样的。

### 2、fork()的底层

在调用**fork()**时，Linux的内核程序实际上只会复制**父进程**的页表以及给子**进程**创建一个进程描述符，并不会立即开辟新的内存空间来复制一份父进程的数据，此时**父进程**和**子进程**共享同一份数据资源，只有**父进程**或者**子进程**发生**写**的操作时，**子进程**才会复制**父进程**的**数据**到一个新的内存空间，这种技术就是**写时拷贝技术**。

但是在程序员看来，**fork()**就是立即克隆出了一个和**父进程**完全相同的子进程，只不过其底层使用了**写时拷贝技术**来**优化**。

### 3、父子进程的关闭

假如**父进程**先于**子进程**关闭，那么子进程就成为**孤儿进程**，被操作系统接管

假如**子进程**先于**父进程**终止，而父进程又没有调用**wait**或**waitpid函数**，**父进程**结束后，**子进程**成为**僵尸进程**，此时**子进程**虽然执行完毕但是**始终占有内核资源**，同时也减少了系统可以创建的最大进程数。

假如**子进程**先于**父进程**终止，而父进程调用了**wait**或**waitpid函数**，那么**父进程**会等待**子进程**结束后再结束，子进程不会成为**僵尸进程**。

|  |  |  |
| --- | --- | --- |
| 父进程创建一个子进程 | 父进程创建2个不同的子进程 | 父进程创建N个相同的子进程 |
| int fd = **fork()**; if(fd < 0){  exit(1); } else if(fd == 0){  **// child code** } else {  **// father code** } | int fd1,fd2; fd1 = **fork()**; if(fd < 0){  exit(1); } else if(fd == 0){  **// child 1 code** } else {  **fd2 = fork()**;  if(fd2 < 0){  exit(1);  } else if(fd2 == 0){  **// child 2 code**  } else {  **// father code**  } } | // 父进程创建n个子进程 pid\_t pid; for(int i = 0; i < n; i++){  **pid=fork()**;  if(pid<=0){  sleep(1);  **break**;// 必不可少  }  } // 不同的进程执行不同的代码 if(pid < 0){  exit(1); }  else if(pid == 0){  **// child code** }  else {   **// father code** } |

## 三、父子进程通信：匿名管道

这里使用的是Linux的**匿名管道**（因为其创建管道使用了**pipe()**,命名管道使用**mkfifo()**），其基本用法是

**匿名管道**的特点如下：

1、**匿名管道**是一种**半双工通信**，数据只能从写端流入读端。

2、**匿名管道**读写的是**内核程序**的**缓冲区**的两端fd[0]和fd[1]。

3、**匿名管道**只能用于**父子进程通信**。这是因为匿名管道没有标识符，其他无亲缘关系的进程无法访问到该管道，只有创建它的父进程和使用它的子进程可以访问。在进行父子进程间通信时，**一定要父进程先创建管道，再创建子进程**，这样子进程才能访问匿名管道。

4、**匿名管道**默认是**阻塞**的，父子进程抢占式运行，**如果父进程或者子进程先读**，发现是空管道时就会阻塞，等待写入；某个进程写入一次后，读进程抢占到才能读入。

5、**匿名管道**读取数据是**将数据剪**切，所以**多个进程读匿名管道，无法拿到相同的数据**

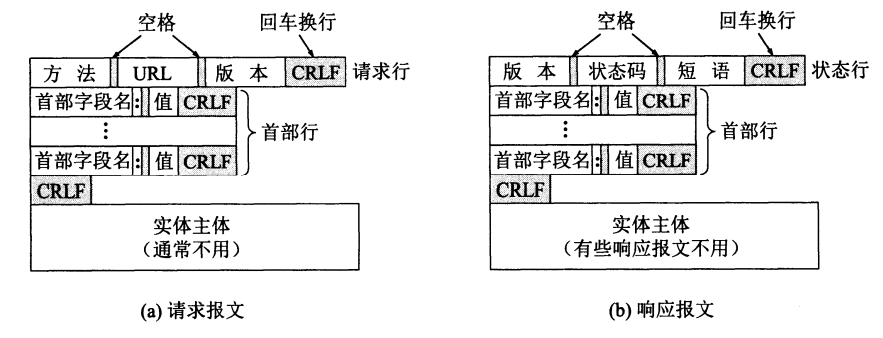
6、**匿名管道**的读写具有**原子性**，即读操作或写操作要么完成，要么没有开始，不存在边读边写，或者同时写

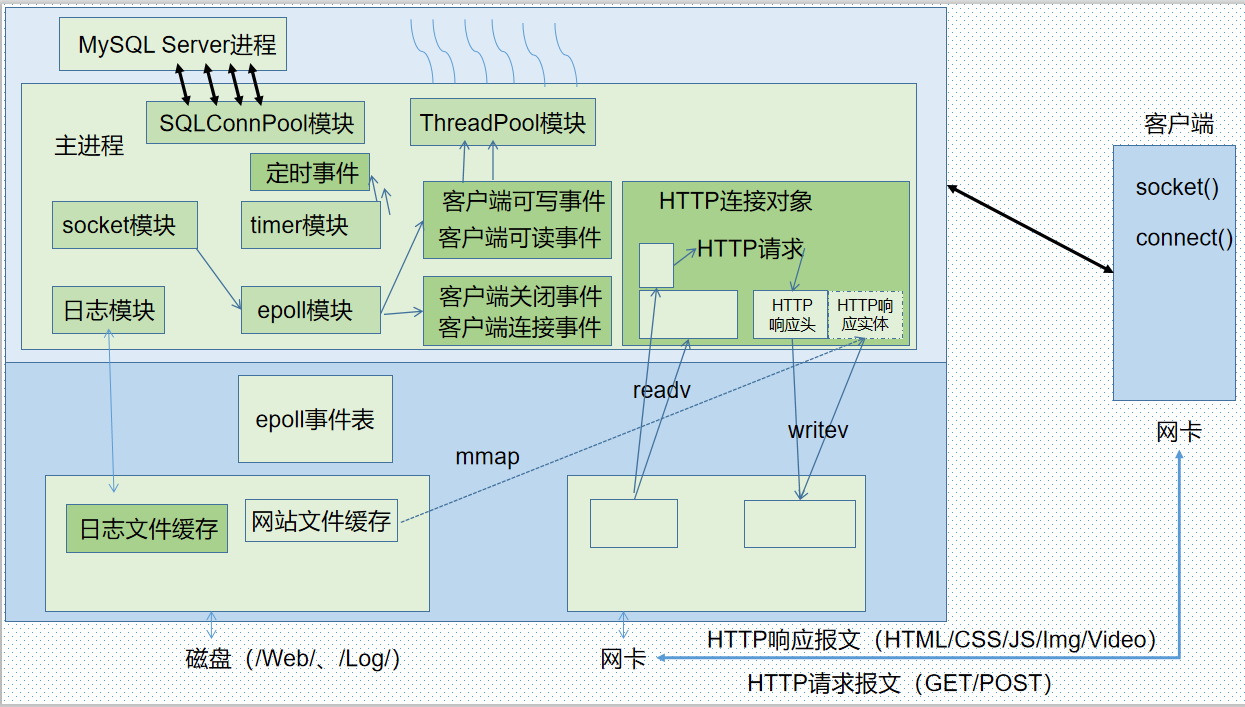
7、**匿名管道**的读写是**字节流**，用户write两次写入字符串，对于管道来说两次输入的数据没有明显边界，read可以读任意字节大小，在读取时无法区分两次输入，前后两次输入粘连在一起，read的返回值是读取到数据的字节数。

|  |  |
| --- | --- |
| // 父进程创建管道 int mypipe[2]; int rt=pipe(mypipe); FILE \*f; // 父进程创建client个子进程 pid\_t pid; for(int i = 0; i < n; i++){  pid=fork();  if(pid<=0){  sleep(1);  break;// 必不可少  }  } if(pid<0){  sys\_err("fork err"); } | // 子进程向管道写数据 else if (pid == 0){  fdopen(mypipe[1],"w");//打开管道的写端  fprintf(f,"%d %d %d\n",speed,failed,bytes);// 子进程每次写入一行格式化的数据  fclose(f); } // 父进程从管道读数据 else{     fdopen(mypipe[0],"r");//打开管道的读端     while(client--){         fscanf(f,"%d %d %d",&i,&j,&k);// 父进程每次读入一行格式化的数据    }     fclose(f); } |

# WebServer篇

**一、项目架构逻辑图**

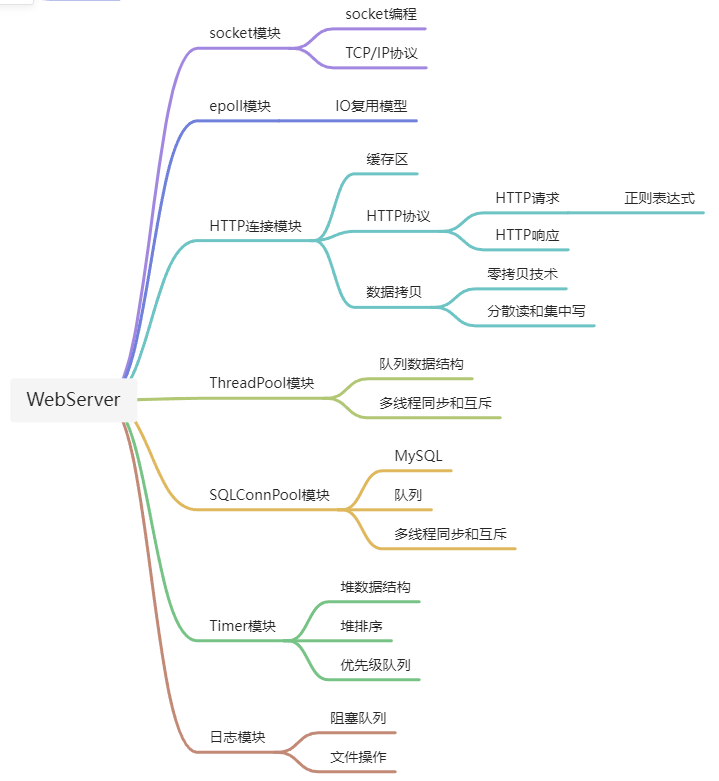




**二、工作流程**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| int **main**(int **argc**,char\* **argv**[]){  1、创建**WebServer对象**：WebServer\* server=new WebServer();   |  |  |  |  | | --- | --- | --- | --- | | 1、创建**心跳计时器**：HeapTimer **m\_pTimer**=new HeapTimer();  2、创建**监听套接字**：ListenSocket **m\_pSocket**=new ListenSocket();  3、创建**Epoll模型**： Epoller **m\_pEpoller**=new Epoller();  4、创建**线程池**：ThreadPool **m\_pThreadPool**=new ThreadPool(threadNum);  5、初始化**HTTP连接**：this->InitHttpConn();   |  | | --- | | srcDir\_ = **getcwd**(nullptr, 256);// 获取当前工作目录  strncat(srcDir\_, "/web/", 16);  **HttpConn**::userCount = 0;// 设置HTTP连接的用户数量  **HttpConn**::srcDir = srcDir\_;// 设置当前工作目录 |  1. 初始化**监听套接**字：this->InitListenSocket(port,OptLinger);  |  | | --- | | **m\_pSocket**->Init(port,OptLinger) |  1. 初始化**Epoll模型**：this->InitEpoller(trigMode);  |  | | --- | | this->\_InitEventMode(trigMode);  int \_listenFd=**m\_pSocket**->GetListenFd();  int ret = **m\_pEpoller**->AddFd(\_listenFd, listenEvent\_ | EPOLLIN);// 注册epoll事件  **m\_pSocket**->SetFdNonblock(\_listenFd); |   8、初始化**数据库连接池**：**SqlConnPool**::Instance()->InitPool();  9、初始化**日志模块**：**Log**::Instance()->init(logLevel, "./log", ".log", logQueSize); |   2、启动**WebServer对象**: server->Run();   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 1、获取**监听套接字**：int **\_listenFd**=m\_pSocket->GetListenFd(); 2、循环监听**epoll事件**：while(!isClose\_)：   |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 1、获取跳动间隔并且执行定时事件：if(timeoutMS\_ > 0) timeMS = **m\_pTimer**->GetNextTick();  2、阻塞等待**epoll事件**：int eventCnt = **m\_pEpoller**->Wait(timeMS);// 最长阻塞timeMS毫秒  3、分类处理**epoll事件**：for(int i = 0; i < eventCnt; i++)：   |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 1、获取**epoll事件**的**套接**字fd：int fd = **m\_pEpoller**->GetEventFd(i);  2、uint32\_t events = m\_pEpoller->GetEvents(i);  处理**客户端**请求的**连接事件**：if(fd==**\_listenFd**) DealListen\_();   |  |  | | --- | --- | | 1、尝试连接客户端：struct sockaddr\_in addr; int fd=m\_pSocket->Accept(&addr);  2、如果连接失败或者客户端较多，不允许连接  3、连接成功时，将fd和addr存储起来：   |  | | --- | | 1、初始化一个**HTTP连接**：m\_UsersDict[fd].init(fd, addr);//<连接套接字,HttpConn对象>  2、为fd添加一个**定时关闭事件**：m\_pTimer->add(fd, timeoutMS\_, bind(&WebServer::CloseConn\_, this, &m\_UsersDict[fd]));  3、添加监听fd的**epoll事件**：m\_pEpoller->AddFd(fd, EPOLLIN | connEvent\_);  4、设置fd为**非阻塞状态**：m\_pSocket->SetFdNonblock(fd); | |   处理**客户端**请求的**关闭**/**出错**：else if(events&(EPOLLRDHUP | EPOLLHUP | EPOLLERR))：   |  |  | | --- | --- | | assert(m\_UsersDict.count(fd) > 0);// 如果没有找到该用户  CloseConn\_(&m\_UsersDict[fd]);   |  | | --- | | 1、删除监听fd的**epoll事件**：m\_pEpoller->DelFd(client->GetFd());  2、关闭**HTTP连接**：client->Close(); | |   处理客户端的**可读事件**：else if(events & EPOLLIN)：   |  |  |  |  | | --- | --- | --- | --- | | assert(m\_UsersDict.count(fd) > 0);  DealRead\_(&m\_UsersDict[fd]);   |  |  |  | | --- | --- | --- | | 1、为fd添加**定时事件**：m\_pTimer->adjust(client->GetFd(), timeoutMS\_);  2、开启一个**线程**：m\_pThreadPool->AddTask(std::bind(&WebServer::OnRead\_, this, client));   |  |  | | --- | --- | | 1、**HTTP连接**读取：int ret = client->read(&readErrno);  2、读取出错就关闭**HTTP连接**：CloseConn\_(client);  3、读取成功就处理读取的数据   |  | | --- | | 调用**client->process()**：  1、如果返回为真，激活fd的**可写epoll事件**：  **m\_pEpoller**->ModFd(client->GetFd(), connEvent\_ | EPOLLOUT);  2、如果返回为假，激活fd的**可读epoll事件**：  **m\_pEpoller**->ModFd(client->GetFd(), connEvent\_ | EPOLLIN); | | | |   处理客户端的**可写事件**：else if(events & EPOLLOUT)：   |  |  |  |  | | --- | --- | --- | --- | | assert(m\_UsersDict.count(fd) > 0);  DealWrite\_(&m\_UsersDict[fd]);   |  |  |  | | --- | --- | --- | | 1、为fd添加**定时事件**：**m\_pTimer**->adjust(client->GetFd(), timeoutMS\_);  2、开启一个**线程**：**m\_pThreadPool**->AddTask(std::bind(&WebServer::OnWrite\_, this, client));   |  |  | | --- | --- | | 1、**HTTP连接**读取：int ret = client->write(&writeErrno);  2、如果传输完成： if(client->ToWriteBytes() == 0&&client->IsKeepAlive())：   |  | | --- | | 调用**client->process()**：  1、如果返回为真，激活fd的**可写epoll事件**：  **m\_pEpoller**->ModFd(client->GetFd(), connEvent\_ | EPOLLOUT);  2、如果返回为假，激活fd的**可读epoll事件**：  **m\_pEpoller**->ModFd(client->GetFd(), connEvent\_ | EPOLLIN); |   3、如果传输没有完成： else if(ret < 0&&writeErrno == EAGAIN)：  **m\_pEpoller**->ModFd(client->GetFd(), connEvent\_ | EPOLLOUT);//**可写epoll事件**  4、如果传输出错：CloseConn\_(client); | | |   处理未知事件：else LOG\_ERROR("Unexpected event"); | | |   } |

**三、主要知识点分布**



第4章 定时器

服务器程序通常管理着众多定时事件，比如定期检测一个客户连接的活动状态，因此有效地组织这些定时事件，使之能在预期的时间点被触发且不影响服务器的主要逻辑，对于服务器的性能有着至关重要的影响。

为此，程序员通常将每个定时事件分别封装成定时器，并使用某种容器类数据结构（比如链表、 排序链表和时间轮），将所有定时器串联起来，以实现对定时事件的统一管理。

设计定时器的一种思路是将所有定时事件放入一个排序容器，主循环以固定的频率调用一个心搏函数tick()，在其中依次检测到期的定时器，然后执行到期定时器上的回调函数。

设计定时器的另一种思路是：使用时间堆，将所有定时事件放在一个最小堆中，将最小堆中的堆顶元素的到期时间最近的时间间隔作为心搏间隔。这样，一旦心搏函数tick()被调用，超时时间最小的定时器必然到期，我们就可以在tick函数中处理该定时器。然后，再次从剩余的定时器中找出超时时间最小的一个，并将这段最小时间设置为下一次心搏间隔。如此反复，就实现了较为精确的定时。

## 一、定时器的数据结构

|  |  |
| --- | --- |
| 定时事件的设计 | 定时事件容器的设计 |
| typedef std::function<void()> TimeoutCallBack;//返回值为空的对象，定时回调  typedef std::chrono::high\_resolution\_clock Clock;  typedef std::chrono::milliseconds MS;// 毫秒级别  typedef Clock::time\_point TimeStamp;// 时间点，单位秒数  struct TimerNode {  int id;// 标识符，该项目中是socket描述符  TimeStamp expires;// 到期时间点，使用的自1970年开始的绝对时间，单位：秒  TimeoutCallBack cb;// 定时回调函数，到期时应该执行的操作  // < 运算符重载函数，用来比较不同的时间点，方便排序  bool operator<(const TimerNode& t) {  return expires < t.expires;  }  }; | std::vector<TimerNode> heap\_;// 时间堆（最小堆，到期时间最小的在第1个）  std::unordered\_map<int, size\_t> ref\_;// 哈希表<TimerNode.id,> |

## 二、执行定时事件

|  |  |
| --- | --- |
| int HeapTimer::GetNextTick() {  tick();// 跳动函数，执行所有到期的定时事件  size\_t res = -1;  // 如果堆不为空  if(!heap\_.empty()) {  // 计算堆顶元素的到期间隔  res = std::chrono::duration\_cast<MS>(heap\_.front().expires - Clock::now()).count();  if(res<0) {res=0;}  }  return res;  } | // 跳动函数（执行定时事件）  void HeapTimer::tick() {  if(heap\_.empty()) {  return;  }  // 遍历堆，将所有到期的定时事件全部执行  while(!heap\_.empty()) {  TimerNode node = heap\_.front();//取第一个节点  // 如果还没有到期，跳出循环  if(std::chrono::duration\_cast<MS>(node.expires - Clock::now()).count() > 0) {  break;  }  node.cb();// 调用回调函数  pop();// 去除该节点  }  } |

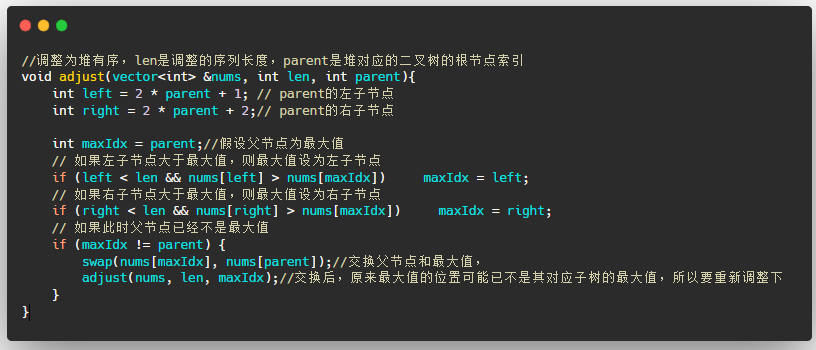
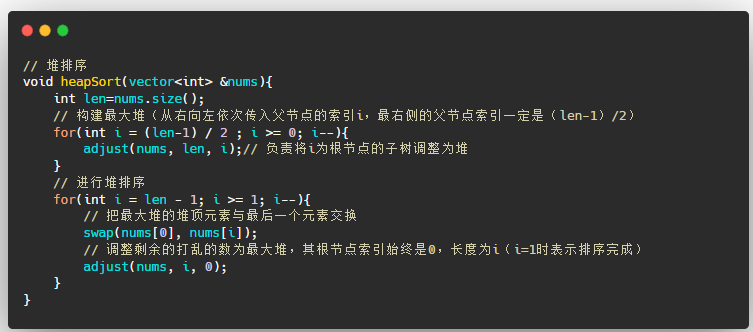
## 三、添加定时事件

|  |  |
| --- | --- |
| // 添加一个定时事件，参数分别是socket描述符、时间间隔、回调函数  void HeapTimer::add(int id, int timeout, const TimeoutCallBack& cb) {  assert(id >= 0);  size\_t i;  // 如果ref\_中不存在id，队尾添加新节点，调整堆  if(ref\_.count(id) == 0) {  i = heap\_.size();// 大小  ref\_[id] = i;  heap\_.push\_back({id, Clock::now() + MS(timeout), cb});// 添加一个TimerNode  siftup\_(i);  }  // 如果ref\_中不存在id，覆盖原有节点，调整堆  else {  i = ref\_[id];  heap\_[i].expires = Clock::now() + MS(timeout);  heap\_[i].cb = cb;  if(!siftdown\_(i, heap\_.size())) {  siftup\_(i);  }  }  } | void HeapTimer::adjust(int id, int timeout) {  /\* 调整指定id的结点 \*/  assert(!heap\_.empty() && ref\_.count(id) > 0);  heap\_[ref\_[id]].expires = Clock::now() + MS(timeout);;  siftdown\_(ref\_[id], heap\_.size());  } |

## 四、调整堆结构

|  |  |
| --- | --- |
| void HeapTimer::SwapNode\_(size\_t i, size\_t j) {  assert(i >= 0 && i < heap\_.size());  assert(j >= 0 && j < heap\_.size());  std::swap(heap\_[i], heap\_[j]);  ref\_[heap\_[i].id] = i;  ref\_[heap\_[j].id] = j;  }  void HeapTimer::siftup\_(size\_t i) {  assert(i >= 0 && i < heap\_.size());// 检查参数是否合法  size\_t j = (i-1) / 2;  while(j >= 0) {  if(heap\_[j] < heap\_[i]) { break; }  SwapNode\_(i, j);  i=j;// 更新i  j=(i-1)/2;// 更新j  }  } | bool HeapTimer::siftdown\_(size\_t index, size\_t n) {  assert(index >= 0 && index < heap\_.size());// 检查参数是否合法  assert(n >= 0 && n <= heap\_.size());// 检查参数是否合法  size\_t i = index;  size\_t j = i \* 2 + 1;  while(j < n) {  if(j + 1 < n && heap\_[j + 1] < heap\_[j]) j++;  if(heap\_[i] < heap\_[j]) break;  SwapNode\_(i, j);// 交换两个节点  i = j;  j = i \* 2 + 1;  }  return i > index;  } |

## 五、堆排序



# 线程池

线程池（thread pool）是一种在Linux并发场景中经常用到的工具。线程池的实现思路基本如下：

* 提前创建若干个线程，将其设置为休眠
* 当需要线程执行任务时，向线程池中添加一个任务，该线程池自动挑选一个空闲线程来执行任务。

## C++的多线程

C++11开始在语言层面支持多线程，提供了标准的线程库std::thread，其基础操作如下：

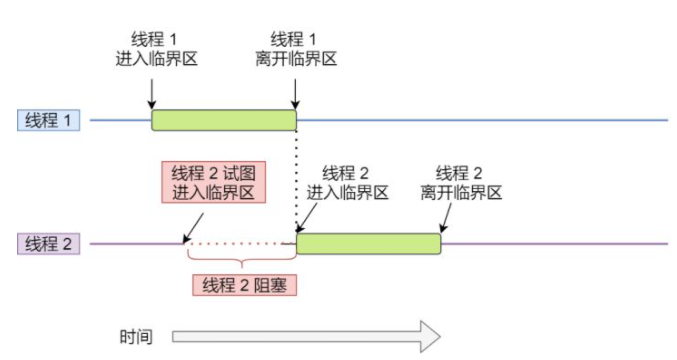
|  |
| --- |
| #include<thread>  using namespace std;  **// 创建并且执行线程对象**  thread t1;// 创建一个空的线程对象；  thread t2(f1,n+1);// 创建一个线程对象，传入返回值为void的f1函数，f1参数为n+1；  thread t3([]{});//创建一个线程对象，传入lambda表达式  **// 线程的阻塞设置**  t1.join;// 阻塞主线程，执行完后，主线程继续执行，一般用在主线程最后，防止子线程没有执行完毕就退出  t1.detach();// 不阻塞主线程，同时主线程继续执行，子线程变成守护线程，被C++运行时库接管。  //一个线程只能调用一次join()或detach(),调用detach()后不能再调用join()  **// 线程的操作**  joinable()// 检查线程是否可被join或detach()  std::this\_thread::getid()// 获取当前线程的ID  std::thie\_thread::yield()// 当前线程放弃执行  sleep\_until()// 线程休眠至某个指定时刻才被重新唤醒  sleep\_for()// 线程休眠某个指定的时间段才被重新唤醒 |

## C++的多线程同步

**CPU在进程之间的调度可能导致进程在读写一些共享数据时出现问题。**（例如进程A在读写某个共享变量a时恰好被CPU打断，然后共享变量a被另一个进程B进行了修改，这时进程A可能会产生错误的结果。）

### 1、同步和互斥的区别

两个或多个进程读写某些**共享数据**，而最后的结果取决于运行的精确时序，称为**竞争条件**（race condition），包含**竞争条件**的程序大部分运行良好，但是极少数情况会出现错误。想要避免竞争条件，就需要确保一个进程在使用一个共享资源时其他资源不可以做同样的操作，这就是**互斥**（mutual exclusion）。一个进程中会对共享内存进行访问的程序片段叫做**临界区域**（critical region）或**临界区**（critical section）。



同一个进程中的多个线程共享进程内的资源，所以多个线程之间的工作也需要互斥，同时多线程也需要同步。所谓同步就是并发进程/线程在一些关键点上可能需要互相等待与互通消息，这种相互制约的等待与互通信息称为进程/线程同步。

同步和互斥的区别：

|  |
| --- |
| 同步就好比：「操作A应在操作B之前执行」，「操作C必须在操作A和操作B都完成之后才能执行」等；  互斥就好比：「操作A和操作B不能在同一时刻执行」； |

### 2、同步和互斥的实现

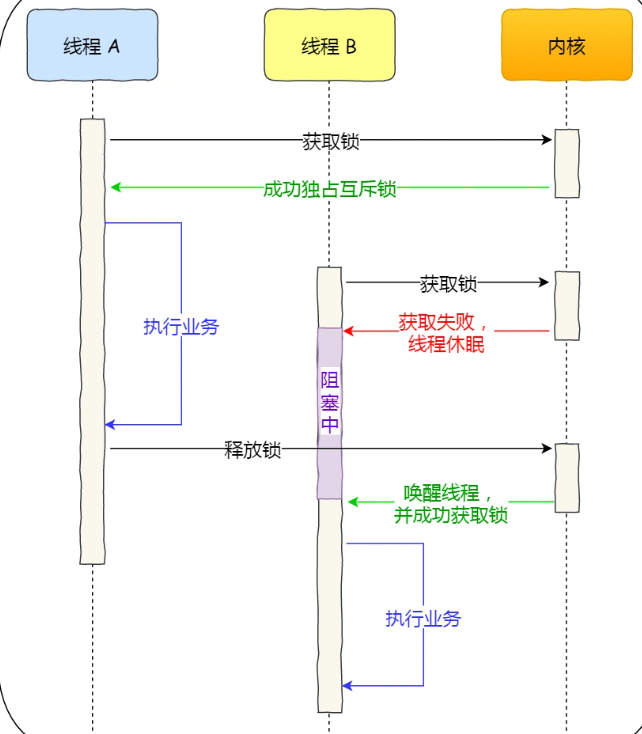
#### 互斥锁

使用加锁(lock)和解锁(unlock)操作来控制对共享资源的访问，就可以解决不同进程或线程之间的互斥问题，这种锁也叫做**互斥锁**或**互斥量**。互斥锁的数据类型是：pthread\_mutex\_t。

**互斥锁的工作原理/流程：**

|  |
| --- |
| 互斥锁是一种互斥机制，本质上是一把锁，在访问共享资源前对加锁，在访问完成后释放锁。  在当前线程加锁后，解锁前，其他任何线程都不能访问该互斥锁锁住的临界区，进入休眠或者忙等待，这体现了互斥性和唯一性。  在当前线程解锁后，内核程序会在等待队列中取出一个等待线程，将其唤醒并让其获得该互斥锁进行执行。 |

根据回斥锁的实现方式，也可分为**忙等待锁**（**自旋锁**（spin lock））和**无等待锁**。忙等待锁中获取不到锁的线程就会一直进行while循环而不做任何事，一直占据CPU资源，直到锁可用。无等待锁则是线程没有锁的时候就放入锁的等待队列，然后将CPU资源让给其他线程。



互斥锁可以保证任何时刻只能有一个线程访问临界区，请注意把握临界区的选择，如果将线程中的大部分甚至全部代码都设置为临界区，那么线程的执行将趋近于串行执行，失去了并行执行的优势，降低并发效率。

自旋锁加锁和解锁的性能开销比互斥锁要小一点，但是长时间执行会占据CPU资源，所以**当临界区的代码执行时间非常短时，应该选用自旋锁，否则使用互斥锁（无等待锁）**。

C++11通过#include <mutex>提供了互斥锁，该文件提供4种锁变量：

|  |
| --- |
| std::mutex，最基本的 Mutex 类。 std::recursive\_mutex，递归 Mutex 类。 std::time\_mutex，定时 Mutex 类。 std::recursive\_timed\_mutex，定时递归 Mutex类。 |

上述这4种锁变量，其中基本都有lock()、unlock()和try\_lock()等操作，这些操作都需要程序员自行调用。为了更加方便地对互斥量进行上锁和解锁，该文件又提供了2种更加方便地上锁的类：

|  |
| --- |
| std::lock\_guard //与 Mutex RAII 相关，上锁后，不能再手动解锁/上锁，只能在声明区结束时自动解锁。 std::unique\_lock//与 Mutex RAII 相关，上锁后，可以再次手动解锁或者上锁，声明区结束时自动解锁。 |

unique\_lock效率不如lock\_guard，但是unique\_lock的颗粒度更细，可以在声明区中手动解锁和上锁。

使用unique\_lock的用法如下：

|  |
| --- |
| #include <mutex>  std::mutex mtx;// 互斥锁  {  std::unique\_lock<std::mutex> locker(mtx); // 对声明区上锁  // 执行临界区的代码（保证同一时刻只能有一个线程在执行这段代码）  locker.unlock();// 临时解锁  // 执行非临界区的代码（同一时刻可能有多个线程在执行这段代码）  locker.lock();// 再次加锁  } |

#### （2）条件变量

条件变量是利用线程间共享的全局变量进行同步的线程同步机制；在条件变量上等待的线程以睡眠的方式等待条件变量的满足；一个线程等待"条件变量的条件成立"挂起，另一个线程使"条件成立"；条件变量的使用总是和一个互斥锁结合在一起。

C++11中条件变量的使用：

|  |
| --- |
| #include <condition\_variable>// 条件变量  std::mutex mtx;// 创建一个互斥锁  std::condition\_variable cond;//创建一个条件变量  {  std::unique\_lock<std::mutex> locker(mtx);// 对声明区上锁  cond.wait(locker);// 阻塞当前线程，释放锁，等待被唤醒  cnd.wait(locker, []{ return ready;}); //如果条件满足，继续执行;否则阻塞当前线程并释放锁，等待被唤醒  cnd.wait\_for(locker, time\_duration, []{ return ready;}); // 如果条件满足或者超时则执行，否则当前线程并释放锁  cnd.notify\_one();//唤醒等待队列中的第一个阻塞线程；不存在锁争用，能够立即获得锁,其余的线程不会被唤醒  cnd.notify\_all();//唤醒等待队列中的所有阻塞线程，存在锁争用，只有一个线程能够获得锁，其余未获取锁的线程继续尝试获得锁  }// 自动解锁 |

## C++的线程池

线程池（thread pool）是一种在Linux并发场景中经常用到的工具，它的作用是提前创建好若干个线程，避免在业务繁忙的高并发场景中临时频繁创建和销毁线程带来的性能损耗。线程池的基本需求是：

* 提前创建若干个线程，将其设置为休眠
* 当需要线程执行任务时，向线程池中添加一个任务，该线程池自动挑选一个空闲线程来执行任务。

实现线程池的难点在于解决并发问题：将多个任务同时分配给多个线程，需要避免多个线程同时竞争同一个任务。解决思路是使用一个任务队列，向队列添加任务时需要加锁，确保同一时刻只能被一个线程添加；从队列取出任务时需要加锁，确保线程池中的多个线程在同一时刻只能有一个线程取出任务，并且该线程在执行任务时需要解锁，最大程度保证并发性，这中间需要涉及多线程、互斥锁、条件变量。

这里只介绍其中的核心函数：

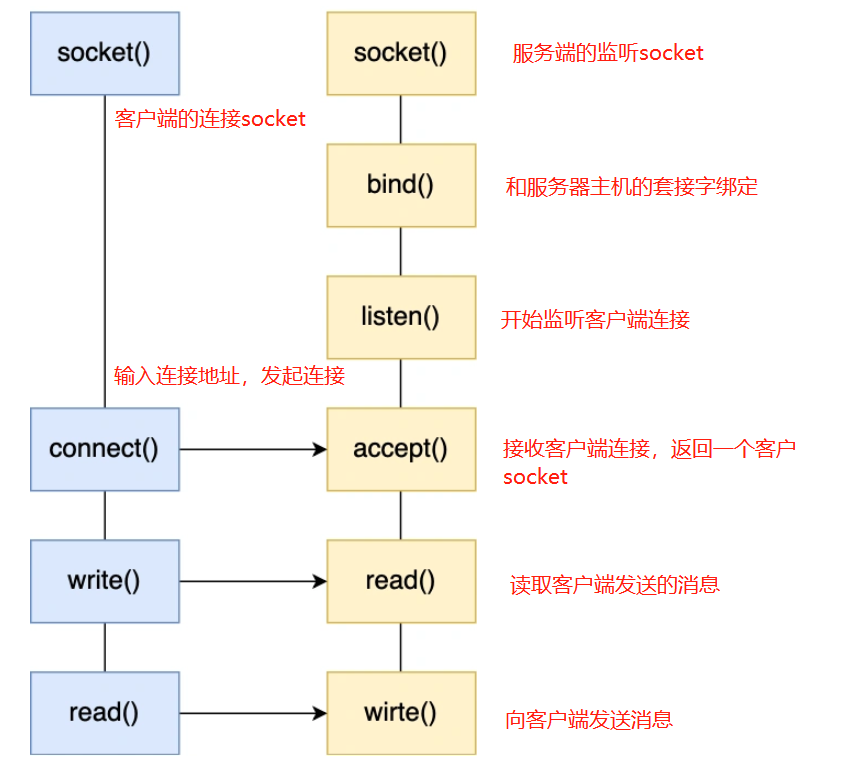
|  |  |
| --- | --- |
| 线程池的数据结构 |  |
| 线程池的主要接口 |  |
| 创建若干个空闲的线程 |  |
| 销毁空闲的线程 |  |

## C++的智能指针

# Socket编程

## 一、socket的主要API

想要在不同主机间的进程间通信就必须使用socket编程。



socket()创建一个listened\_socket，需要指定网络协议为IPv4还是IPv6

bind()给listened\_socket绑定一个套接字（IP地址和端口号）:绑定IP地址就是和对应网卡的IP地址绑定，一台主机可能有多个网卡；绑定端口号就是选用一个空闲端口号作为当前程序的唯一标识。

listen()启动当前网络程序来监听绑定的端口号，这个可以作为判定一个网络程序是否启动的条件

accept()从内核程序中来获取客户端的连接，默认采用阻塞I/O，即没有客户端连接返回就会一直阻塞。

当客户端程序调用connect()函数来连接指定地址的主机时，客户端和服务端的TCP三次握手就开始了，

当内核程序接收到一个客户端连接时，返回一个已经完成连接的Socket。

上述模型是最基本的网络通信模型，它默认采用了同步阻塞I/O，基本只能用来一对一通信。

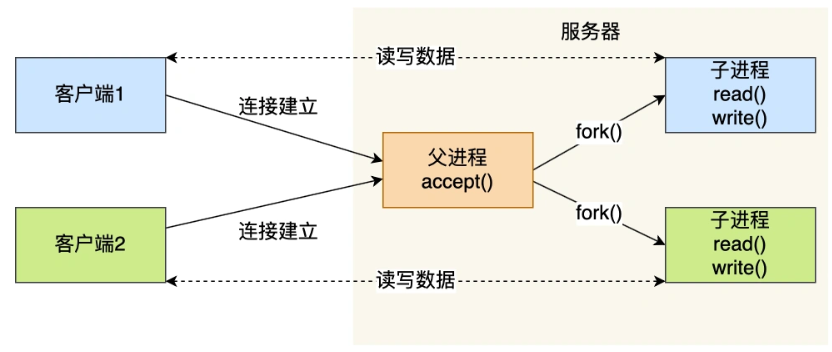
在Linux编程中：

|  |  |
| --- | --- |
| 创建一个监听套接字 | int m\_listenfd = socket(PF\_INET, SOCK\_STREAM, 0); |
| 创建一个IPv4的Socket地址 | struct sockaddr\_in address;//socket地址  bzero(&address, sizeof(address));//将该地址清空为0  address.sin\_family = AF\_INET;//设置IPv4的地址族  //inet\_pton(AF\_INET,ip,&address.sin\_addr);//手动设置IP地址  address.sin\_addr.s\_addr = htonl(INADDR\_ANY);// 自动获取本机的IP地址并且设置  address.sin\_port = htons(port);//端口号 |
| 绑定Socket地址 | setsockopt(m\_listenfd, SOL\_SOCKET, SO\_REUSEADDR, &flag, sizeof(flag));//允许重用本地地址和端口  int ret = bind(m\_listenfd, (struct sockaddr \*)&address, sizeof(address));//绑定socket地址 |
| 监听Socket | ret = listen(m\_listenfd, 5); |
| 阻塞等待接受客户端的连接 | struct sockaddr\_in client;  socklen\_t client\_addrlen=sizeof(client);  connfd=accept(m\_listenfd,(struct sockaddr\*)&client,&client\_addrlen); |
| 服务端接收数据 | char buff[4096];  n = recv(connfd,buff,4096,0);//从connfd中接收最大长度为ΪMAXLINE的数据到buff中，返回接收到的长度 |
| 服务端发送数据 | send(connfd,"你好，服务端欢迎你\n",26,0) |

# IO复用

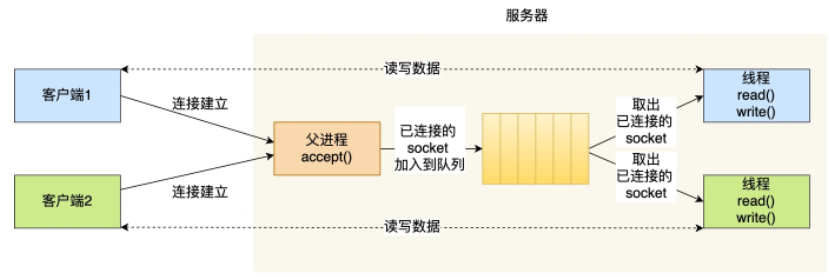
## 一、多进程模型

多进程模型就是为每个客户端分配一个进程来处理请求。服务器的主进程负责监听客户的连接，一旦与客户端连接完成，accept()函数就会返回一个「已连接 Socket」，这时就通过fork()函数创建一个子进程，子进程专心处理和「已连接Socket」的通信。



该模型的不足：创建进程的数量是有上限的（一般不大于100个）；频繁创建进程和销毁进程会严重增加系统开销；如果子进程没有做好回收就会成为僵尸进程，最终耗尽系统资源。

## 二、多线程模型



主进程将接收到的「已连接Socket」放入一个全局队列，然后使用pthread\_create()来创建线程处理和每个「已连接Socket」的通信，为了避免频繁地创建和销毁线程，可以使用线程池技术（提前创建若干个线程），同时为了避免多线程竞争，需要互斥锁机制。

该模型的不足：创建线程的数量是有上限的（一般不大于100个）。

## 三、I/O复用

I/O复用模型可以在一个进程中维护多个「已连接Socket」的通信，原理就是多路复用技术：

|  |
| --- |
| 一个进程虽然任一时刻只能处理一个请求，但是处理每个请求的事件时，耗时控制在1毫秒以内，这样1秒内就可以处理上千个请求，把时间拉长来看，多个请求复用了一个进程，这就是多路复用，这种思想很类似一个CPU并发多个进程，所以也叫做时分多路复用。 |

阻塞I/O需要多进程/线程才能用于并发场景，普通的非阻塞I/O需要在应用程序中不断检查才能用于并发场景，两者在并发场景中都不高效。I/O复用是并发场景中最常使用的I/O通知机制。I/O复用的原理是：

* 应用程序通过I/O复用函数向内核程序注册一组事件集合；
* 内核程序不断检查事件集合，通过I/O复用函数把其中就绪的事件通知给应用程序。

本质上IO复用将普通的非阻塞I/O在应用程序中的轮询通过I/O复用函数交给了内核程序，所以应用程序可以在一个进程/线程中同时处理多个I/O操作。Linux上常用的I/O复用函数是select、poll和epoll\_wait。

需要指出的是，I/O复用函数本身是阻塞的，它们能提高程序效率的原因在于它们可以在同一个线程/进程中具有同时监听多个I/O事件的能力，所以IO复用可以解决阻塞I/O不能用于大量并发的缺陷。

【常见优点】非常适合用于并发场景 【常见缺点】实现复杂。 【应用场景】网络socket。

### select和poll

select和poll实现多路复用的方式是：

|  |
| --- |
| 1. 将已连接socket都放到一个文件描述符集合， 2. 然后调用select/poll函数，进入阻塞等待。（该函数将文件描述符集合拷贝到内核程序里，让内核通过遍历来检查是否有网络事件产生，当检查到有事件产生后，将此已连接socket标记为可读或可写，接着再把整个文件描述符集合拷贝回用户程序里） 3. 用户程序需要遍历select/poll函数返回的集合，分类型处理各种情况。 |

select和poll的区别：

|  |
| --- |
| poll和select并没有太大的本质区别，都是使用「线性结构」存储进程关注的socket集合，select使用固定长度的BitsMap，表示文件描述符集合，该集合默认最大值是1024；poll使用动态数组以链表形式来表示文件描述符集合，突破了select的文件描述符个数限制，当然还会受到系统文件描述符限制。 |

select和poll的缺点：

|  |
| --- |
| select和poll的程序需要进行2次「遍历」文件描述符集合（一次是在内核态里，一次是在用户态里）；还会2次「拷贝」文件描述符集合（从用户空间传入内核空间，由内核修改后，再传出到用户空间），其时间复杂度为O(n)。当客户端数量上升时，性能的损耗也会快速增长。 |

### （2）epoll

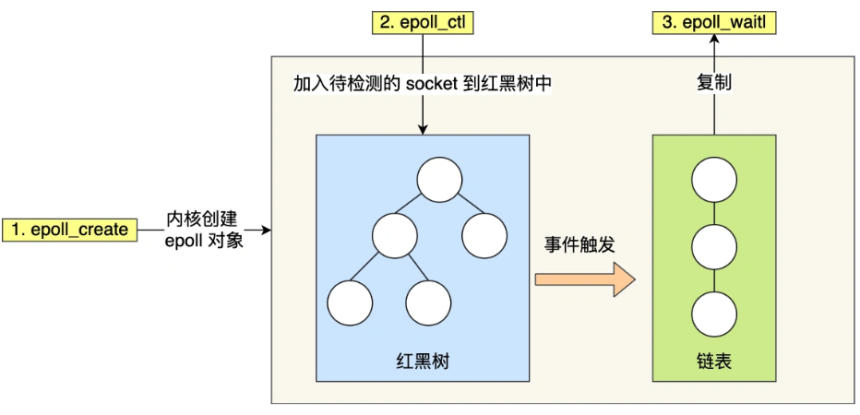
epoll是Linux特有的IO复用函数，其实现和使用与select或者poll有较大差异,其性能也更加高效。epoll机制需要多个系统调用。

|  |
| --- |
| 1. 使用epoll\_create()创建epoll模型，返回一个epoll文件描述符，用来唯一标识要访问的内核事件表 2. 使用epoll\_ctl()向epoll模型中注册一个关联socket的epoll事件（socket需要设置为非阻塞I/O） 3. 调用epoll\_wait()来阻塞监听。epoll\_wait函数如果检测到事件，就将所有就绪的事件从内核程序的内核事件表中复制用户程序。 4. 遍历返回的就绪事件，分情况处理不同的epoll事件。 |

epoll通过如下两个方面很好解决了select/poll的问题：

第一点，epoll在内核程序中使用红黑树来跟踪进程所有待检测的socket，红黑树是个高效的数据结构，增删查一般时间复杂度是O(logn)，而且其每次操作都只需要传入一个待检测的socket，减少了内核和用户空间大量的数据拷贝和内存分配。

第二点，epoll在内核程序中使用事件驱动机制来返回就绪的socket，内核维护了一个链表来记录就绪事件，当用户调用epoll\_wait()函数时，只会返回所有的就绪事件。



epoll支持两种事件触发模式：LT模式（Level Trigger，水平触发）和ET模式（Edge Trigger，边缘触发）。

|  |
| --- |
| LT：epoll的默认工作模式。epoll\_wait每个事件会通知多次，应用程序可以不立即处理，只要满足事件条件，比如内核中有数据需要读，就一直不断地把这个事件传递给用户  ET：epoll的高效工作模式。epoll\_wait每个事件只会通知一次，应用程序必须要立即处理，只有第一次满足条件时才触发，之后就不会再传递同样的事件了。 |

LT模式是默认的工作模式，这种模式下epoll相当于一个效率较高的poll。ET模式在很大程度上降低了同一个epoll事件被重复触发的次数，因此效率要比LT模式高。select/poll只有水平触发模式，epoll默认的触发模式是水平触发，但是可以根据应用场景设置为边缘触发模式。

总结：epoll是几乎是大规模并行网络程序设计的代名词，一个线程里可以处理大量的tcp连接，cpu消耗也比较低。很多框架模型，nginx, node.js底层均使用epoll实现。

## epoll的主要API

### （1）创建epoll模型

主要代码如下：

int epollfd=epoll\_create(5);   
assert(epollfd!=-1);

其中的关键API如下：

#include＜sys/epoll.h＞  
int epoll\_create(int size);

size参数现在并不起作用，只是给内核一个提示，告诉它事件表需要多大。

该函数返回一个epoll文件描述符，用来唯一指定要访问的内核事件表。

### **（2）操作epoll模型**

主要代码如下：

|  |
| --- |
| addfd(epollfd,listenfd,true);//监控listenfd /\*将文件描述符fd上的EPOLLIN注册到epollfd指示的epoll内核事件表中\*/  void addfd(int epollfd,int fd,bool enable\_et) {  epoll\_event event;   event.data.fd=fd;//要监控的listenfd  event.events=EPOLLIN;// 可读事件  if(enable\_et) {event.events|=EPOLLET; }//指定是否对fd启用ET模式  epoll\_ctl(epollfd,EPOLL\_CTL\_ADD,fd,&event); //注册epoll事件  setnonblocking(fd); //设为非阻塞模式 } /\*将文件描述符设置成非阻塞的\*/ int setnonblocking(int fd) {  int old\_option=fcntl(fd,F\_GETFL);   int new\_option=old\_option|O\_NONBLOCK;   fcntl(fd,F\_SETFL,new\_option);   return old\_option;  } |

其中的关键API如下：

#include＜sys/epoll.h＞  
int epoll\_ctl(int epfd,int op,int fd,struct epoll\_event\*event)

fd参数是要操作的文件描述符，op参数则指定操作类型。操作类型有如下3种：

* EPOLL\_CTL\_ADD，往事件表中注册fd上的事件。
* EPOLL\_CTL\_MOD，修改fd上的注册事件。
* EPOLL\_CTL\_DEL，删除fd上的注册事件。

event参数指定事件，它是epoll\_event结构指针类型。epoll\_event的定义如下：

|  |
| --- |
| 结构体 struct epoll\_event{  \_\_uint32\_t events;/\*epoll事件\*/  epoll\_data\_t data;/\*用户数据\*/ }; // 联合体 typedef union epoll\_data {  void\*ptr;  int fd;//目标文件描述符  uint32\_t u32;  uint64\_t u64; }epoll\_data\_t; |

epoll\_ctl成功时返回0，失败则返回-1并设置errno。

### **（3）检测epoll事件**

主要代码如下：

|  |
| --- |
| epoll\_event events[MAX\_EVENT\_NUMBER];//存储监听到的事件 while (1) {  int ret=epoll\_wait(epollfd,events,MAX\_EVENT\_NUMBER,-1); // 阻塞监听事件  if(ret<0) {  printf("epoll\_wait failure\n");   break;   }  lt(events,ret,epollfd,listenfd);/\*使用LT模式\*/   //et(events,ret,epollfd,listenfd);/\*使用ET模式\*/  } |

其中的关键API如下：

#include＜sys/epoll.h＞  
int epoll\_wait(int epfd,struct epoll\_event\*events,int maxevents,int timeout);

epoll\_wait函数如果检测到事件，就将所有就绪的事件从内核事件表（由epfd参数指定）中复制到它的第二个参数events指向的数组中。这个数组只用于输出epoll\_wait检测到的就绪事件，而不像select和poll的数组参数那样既用于传入用户注册的事件，又用于输出内核检测到的就绪事件。这就极大地提高了应用程序索引就绪文件描述符的效率。

epoll获取事件的时候，无须遍历整个被侦听的描述符集，只要遍历那些被内核I/O事件异步唤醒而加入就绪队列的描述符集合。

# 数据拷贝

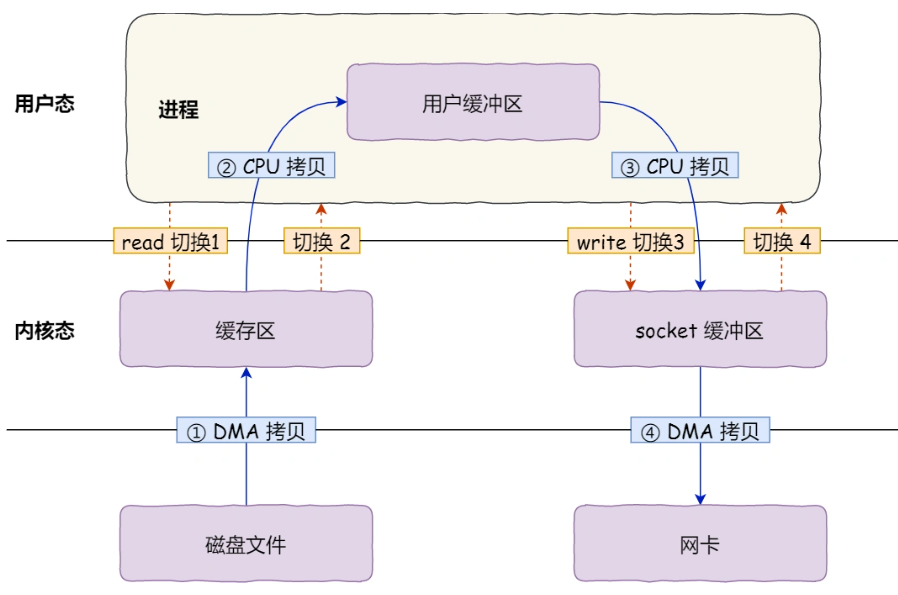
## 一、read和write的数据拷贝

在I/O设备控制方式中，我们简单介绍了DMA技术，它用于磁盘等块设备，自动将数据从磁盘中拷贝到内存的缓存区，或者将数据从内存拷贝到磁盘中，我们暂且将这种拷贝称为DMA拷贝。

假设现在在服务端要提供文件传输功能：读取磁盘上的文件，然后通过网络协议发送给客户端。两个文件I/O的系统调用如下：



代码很简单，但是背后发生了很多事情：



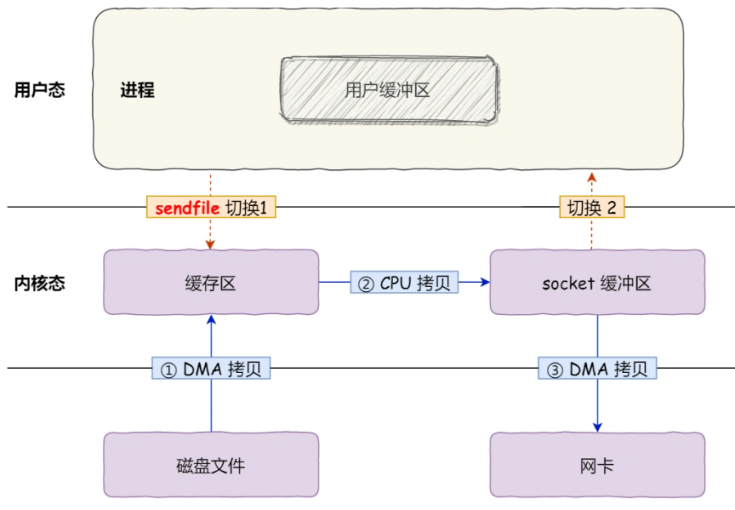
可以看到，两次系统调用共发生了4次「**用户态和内核态的上下文切换**」和4次「**数据拷贝**」。可以发现，我们发送一份数据，内部却拷贝了4次，多余的数据拷贝的时间消耗，在高并发场景下容易被放大和累计，影响性能。

## 二、零拷贝技术

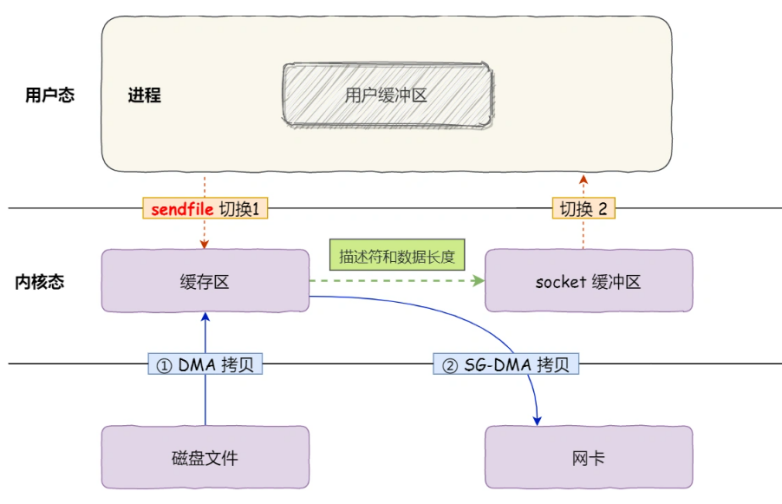
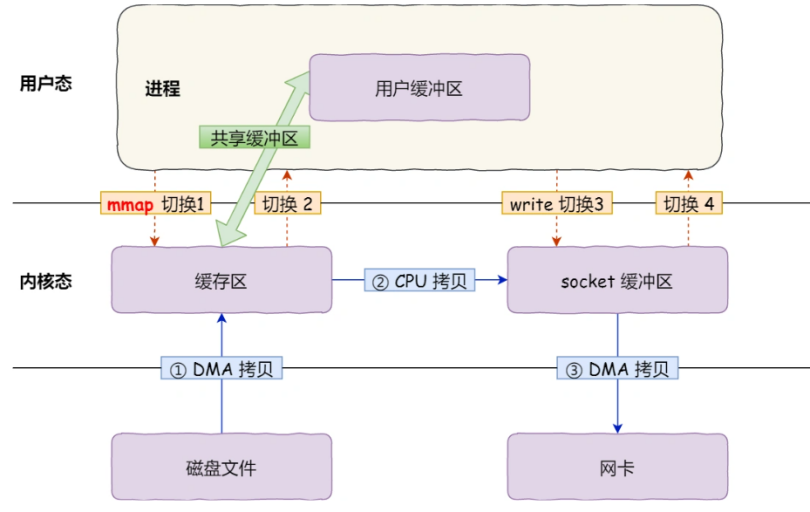
首先在文件传输情景下，用户缓冲区是没必要存在的，因为用户程序并不会修改文件；其次如果可以减少内存中的缓存区，则也会减少「**数据拷贝**」的次数

这种优化思路的实现就是零拷贝技术，零拷贝技术的实现通常存在如下2种方式：

（1）使用系统调用函数mmap+write。用mmap()替换read()系统调用函数，mmap()系统调用函数会直接把内核缓冲区里的数据「映射」到用户空间，这样，就减少了用户缓存区，减少了一次拷贝。



（2）使用系统调用函数sendfile。减少一次系统调用，其次如果网卡支持SG-DMA技术（The Scatter-Gather Direct Memory Access），可以进一步减少一个缓冲区。



这就是所谓的零拷贝（Zero-copy）技术，因为我们没有在内存层面去拷贝数据，也就是说全程没有通过CPU来搬运数据，所有的数据都是通过DMA来进行传输的。

总结下，零拷贝技术用在文件传输场景下，可以全程不需要CPU参与，只通过DMA控制，传统文件传输方式，需要4次「**用户态和内核态的上下文切换**」和4次「**数据拷贝**」，零拷贝技术只需要2次「**用户态和内核态的上下文切换**」和2次「**数据拷贝**」，文件传输的性能提高至少一倍以上。很多著名的开源项目例如Kafka和Nginx都使用了零拷贝技术。

接下来介绍磁盘中的**页缓存/磁盘高速缓存**：

CPU和磁盘之间的读写就是之前说的CPU和I/O设备的读写，其中间需要添加一个**缓冲区**，对于磁盘来讲，这个缓存区就是**页缓存**（PageCache），也就是**磁盘高速缓存**。请注意，**页缓存**是在逻辑上属于磁盘，在物理上属于内存。

**页缓存用来缓存最近被访问的数据，当空间不足时淘汰最久未被访问的缓存**。所以，读磁盘数据的时候，优先**在页**缓存找，如果数据存在则可以直接返回；如果没有，则从磁盘中读取，然后缓存 页缓存中。同时，对于机械磁盘来讲，**页缓存也会采取预读功能**，加入I/O请求读取32KB的数据，但是CPU会在页缓存中读取64KB的数据，将前32KB的数据返回。

当**传输大文件**（GB 级别的文件）**的**时候，**页缓存会不起作用**，这是因为每当用户访问大文件时，如果内核程序就把它们载入页缓存中，页缓存空间将会很快被这些大文件占满，针对大文件的传输，不应该使用 PageCache，也就是说不应该使用零拷贝技术，因为可能由于 PageCache 被大文件占据，而导致「热点」小文件无法利用到 PageCache，这样在高并发的环境下，会带来严重的性能问题。

所以，零拷贝技术使用了页缓存，是一种非直接I/O，它适合传输热点小文件,不适合传输大文件。传输大文件不使用页缓存，而且采用异步模式，综上，高并发场景下的文件传输：

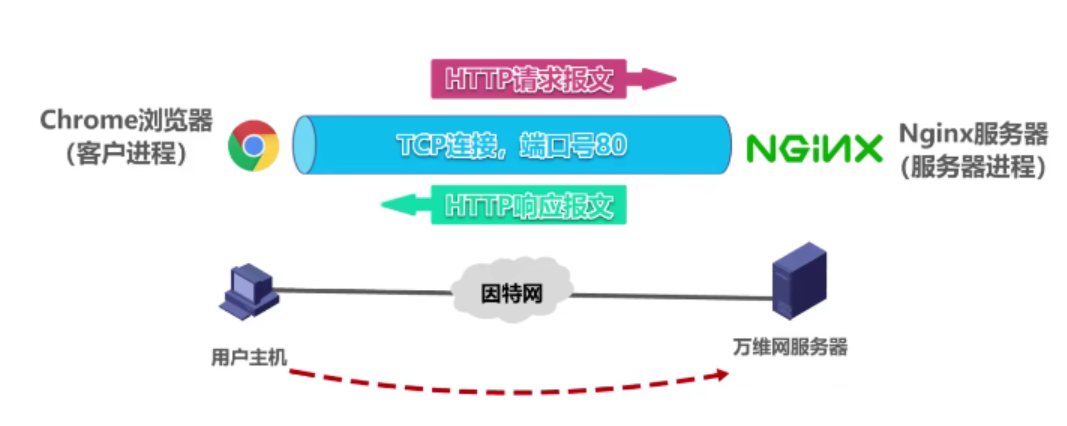
传输小文件：非直接I/O+零拷贝技术

传输大文件：直接I/O+异步I/O

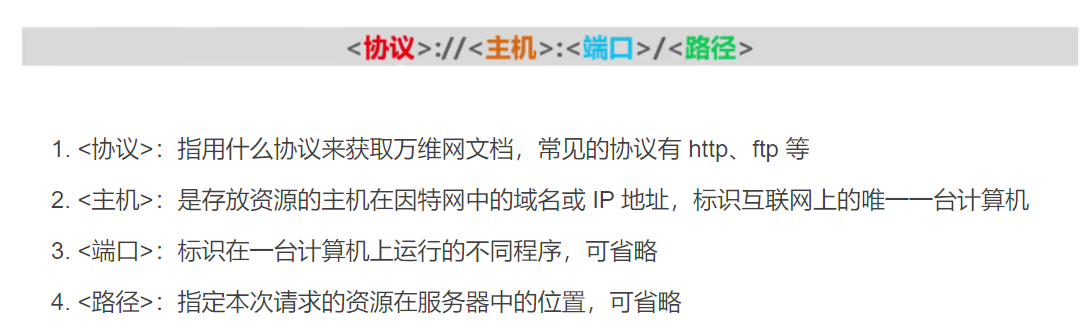
## 三、分散读和集中写

# HTTP篇

超文本传输协议HTTP（Hyper Text Transfer Protocol）是Web服务的应用层协议，超本文指的是含有超链接的文本，HTTP的作用就是在客户端和服务端之间传输文档、图片、视频等文件信息，特别是网页相关文档（HTML、CSS、JS）。



Web浏览器必须通过URL（统一资源定位符）来访问Web服务器上的特定Web页面，URL一般由以下四个部分组成:



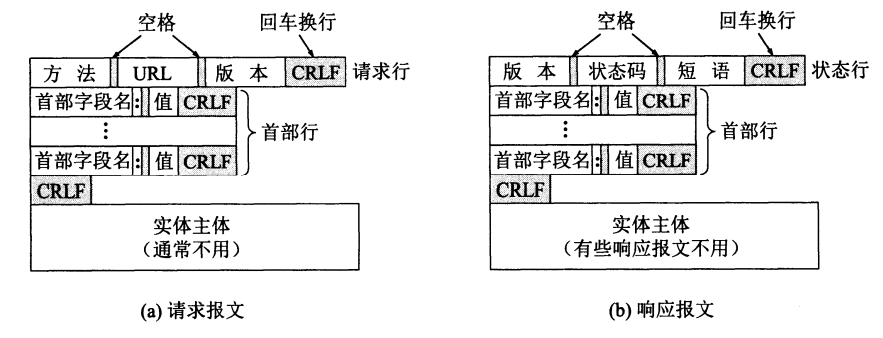
## 一、HTTP协议的特点

HTTP协议有多个版本，HTTP1.0，现在使用最多的是HTTP1.1，特点如下：

|  |
| --- |
| （1）HTTP协议采用C/S的体系结构，由两部分程序实现：一个客户机程序（一般是Web浏览器）和一个服务器程序（一般是Web服务器），客户端发送一个HTTP请求，服务端就要回应要给HTTP响应。  （2）HTTP协议本身是无连接的，但是其运输层采用面向连接的TCP协议。HTTP/1.0采用非持续连接（也叫短连接）方式，即浏览器每次请求一个文件都要与服务器建立一次TCP连接，这样会产生双倍的RTT开销。HTTP/1.1默认采用持续连接（也叫长连接），也可以切换为非持续连接，浏览器与服务器建立一次TCP连接即可传输多次文件，  （3）HTTP协议是无状态（stateless）的，即服务器不会通过HTTP存储任何关于客户机的状态信息，假定用户短时间内发送了相同请求，服务器都会依次做出反应。  （4）HTTP协议的传输是明文传输，报文是可读性很高的ASCII文本，可读性高但是不安全，而且HTTP不验证通信双方的身份。  （5）HTTP协议灵活易扩展，报文中的很多字段都允许开发人员自定义，而且由于HTTP协议在应用层，也可以在其和运输层之间添加额外的层来完善功能，例如HTTPS也就是在HTTP与TCP层之间增加了SSL/TLS安全传输层，HTTP/3甚至把TCP层换成了基于UDP的QUIC。 |

## 二、HTTP协议的请求和响应

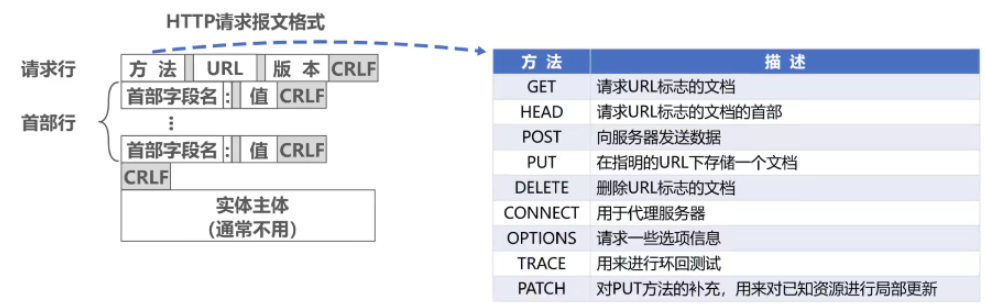
客户机向服务器发送HTTP请求报文，服务器向客户机发送HTTP响应报文。HTTP报文由三部分组成：开始行（请求行或状态行）、首部行和实体主体。HTTP1.0和HTTP1.1都采用ASCII文本，可读性高；HTTP报文的请求格式如下：



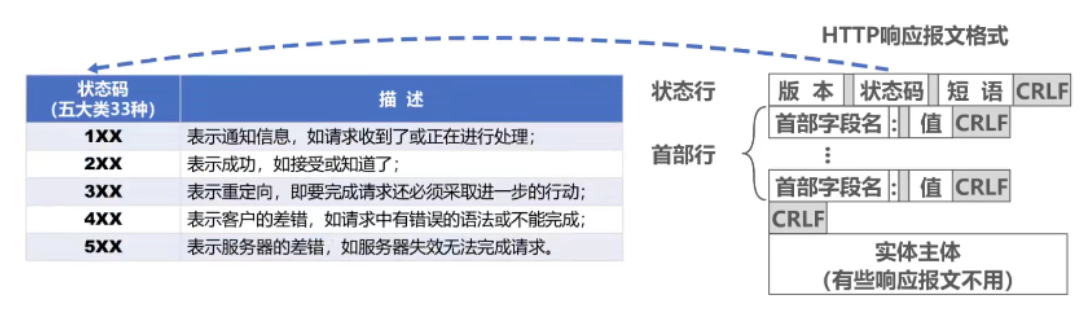
### 1、HTTP报文中的常见字段

|  |  |
| --- | --- |
| **HTTP报文实例** | **HTTP报文字段** |
| GET / HTTP/1.1  Host: 127.0.0.1:1316  User-Agent: Mozilla/5.0  Accept: text/html,application/xhtml+xml,  Accept-Language: zh-CN,zh;q=0.9  Accept-Encoding: gzip, deflate  Connection: keep-alive  Cookie: PSTM=1654071693; | GET指定从服务器获取一个资源  HTTP/1.1指定HTTP协议的版本号  Host指定目标服务器的域名（可能是同一台服务器上的不同网站）  Connection指定本次连接使用长连接还是短连接  Accept指定本次请求可以接收的数据类型，\*/\*表示任何类型  Accept-Encoding指定本次请求可以接收的数据压缩格式  Accept-Language指定本次请求可以接收的语言 |
| HTTP/1.1 200 OK  Connection: keep-alive  Content-Encoding: gzip  Connect-Type: text/html; charset=utf-8  Connect-Length: 3385  <!DOCTYPE html>  <html lang="en">  <head>  </head>  ... | HTTP/1.1指定HTTP协议的版本号  200是状态码，不同的状态码表示本次响应的不同处理结果，200说明请求成功  OK是短语，用来和状态码配合说明处理结果  Connection指定本次连接使用长连接还是短连接  Connect-Encoding指定本次响应的数据压缩格式，之后的字节就属于下一个回应了  Connect-Type指定本次响应的数据类型  Connect-Length指定本次响应的消息实体的数据长度，之后的字节就属于下一个响应 |

### HTTP协议中的不同请求方法



### HTTP协议中的五类状态码



### 4、HTTP协议中的长连接和短连接

在浏览器上输入一次网址或者切换新的网页，并不会只发送一个HTTP请求，这是因为一个网页中有文本、图片、视频等多种信息，可能需要分类型传输。想要完整地获取一个网页，会发送多个HTTP请求：请求HTML文档的GET类型的HTTP请求；请求CSS文件的GET类型的HTTP请求；请求JS文件的GET类型的HTTP请求；请求图片的GET类型的HTTP请求；请求视频的GET类型的HTTP请求等。那么如何高效地发送和接收这些HTTP报文呢？答案是HTTP连接并发和HTTP长连接设置。

HTTP连接并发就是现在的浏览器在有多个HTTP请求时会同时建立多个HTTP连接来同时传输，提升效率。

HTTP长连接设置就是每个HTTP连接建立后，会有一个时间限制，在规定时间内如果还有HTTP请求就继续发送，没有了再关闭HTTP连接。这种长连接设置是因为HTTP连接是建立在TCP连接上的，如果每次发送HTTP请求都需要建立一次TCP连接，那么网页的相应速度将会较慢；在早期的HTTP1.0版本中，每次建立HTTP连接只能发送一个HTTP请求和HTTP相应。

### 5、HTTP协议中的cookie

# MySQL数据库连接池

首先导入依赖文件#include<mysql/mysql.h>，该文件包含MYSQL类的相关API。一个数据库连接池SqlConnPool，池中有多个数据库连接。

## 一、初始化连接池和关闭连接池

首先我们要维护三个变量：最大连接数MAX\_CONN\_，当前可用连接数FREE\_CONN\_和当前已用连接数USED\_CONN\_。然后所有的数据库连接要保存在一个容器中，这里使用队列容器，也可以使用链表等其他容器。

|  |
| --- |
| int MAX\_CONN\_;// 最大连接数 int USED\_CONN\_;//当前已用连接数 int FREE\_CONN\_;//当前可用连接数 queue<MYSQL \*> connQue\_;//存储数据库连接的队列容器 |

### 1、相关API介绍

初始化和关闭单个数据库连接的相关API：

首先是初始化和关闭MYSQL C API库的操作：mysql\_library\_init()和mysql\_library\_end()。该对API一般放在所有的数据库操作之前喝结束后使用。

（1）初始化一个MYSQL连接的实例对象：MYSQL \* mysql\_init(MYSQL \*mysql)：

该API的含义：若输入NULL，该函数将分配、初始化、并返回新对象，否则直接初始化对象并返回对象的地址；分配内存不足时，返回NULL。

（2）接着建立一个和MySQL-server的连接：MYSQL\* mysql\_real\_connect(MYSQL \*mysql,const char \*host,const char \*user, const char \*passwd, const char \*db, unsigned int port,const char \*unix\_socket,unsigned long client\_flag)：

MYSQL \*为mysql\_init函数返回的指针；host为NULL或 “localhost”，后者链接本地计算机；user是账户名，一般为“root”；passwd为密码，一般为NULL；db是数据库名称，为NULL时使用默认的test数据库；port为端口号；unix\_socket为unix连接方式，NULL时表示不使用socket或者管道机制；client\_flag一般为0。如果连接成功，返回第一个参数，否则返回NULL。

关闭一个数据库连接的操作：void mysql\_close(MYSQL \*sock)

### 2、初始化连接池

在初始化的时候我们首先建立connSize个数据库连接：

|  |
| --- |
| void InitPool(const char\* host, int port,  const char\* user,const char\* pwd,   const char\* dbName,  int connSize = 10) {  for (int i = 0; i < connSize; i++) {  MYSQL \*sql = nullptr;  // 初始化一个数据库连接  sql = mysql\_init(sql);  if (!sql) cout<<"MySql init error!"<<endl;  // 建立一个数据库连接  sql = mysql\_real\_connect(sql, host,user, pwd,dbName, port, nullptr, 0);  if (!sql) cout<<"MySql Connect error!")<<endl;  // 保存数据库连接  connQue\_.push(sql);  }  MAX\_CONN = connSize; } |

### 3、关闭连接池

同时，关闭数据库连接池中所有连接的代码如下：

void ClosePool() {  
    // 遍历队列  
    while(!connQue\_.empty()) {  
        // 元素出队  
        auto item = connQue\_.front();  
        connQue\_.pop();  
        // 关闭连接  
        mysql\_close(item);  
   }  
    mysql\_library\_end();//关闭MySQL的使用      
}

## 二、获取连接和释放连接

接下来介绍对连接池的2种操作（获取和释放），它们本质上就是队列容器的出队和入队。但是由于该方法可能同时被多个线程操作，所以需要用到锁机制。

这里我们使用信号量来控制并发线程的数量不超过连接池的数量，通过互斥锁来控制同一时刻只有一个线程可以进行连接池的获取或释放操作。

初始化信号量和互斥锁：

#include <semaphore.h>//信号量  
#include <mutex>//互斥锁  
using namespace std;  
​  
sem\_t semId\_;//定义一个信号量变量  
sem\_init(&semId\_, 0, MAX\_CONN\_);//初始化信号量  
mutex mtx\_;//互斥量

|  |  |
| --- | --- |
| 获取连接 | 释放连接 |
| MYSQL\* SqlConnPool::GetConn() {  MYSQL \*conn = nullptr;  if(connQue\_.empty()){  cout<<"SqlConnPool busy!"<<endl;  return conn;  }  // 信号量-1  sem\_wait(&semId\_);  {  lock\_guard<mutex> locker(mtx\_);//在该声明周期内上锁  // 一个连接出队  conn = connQue\_.front();  connQue\_.pop();  }  return conn; } | void FreeConn(MYSQL\* sql) {     assert(sql);     lock\_guard<mutex> locker(mtx\_);//在该声明周期内上锁     connQue\_.push(sql);//入队     sem\_post(&semId\_);//信号量+1 } |