北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

计算机应用编程实验

高性能Web服务器

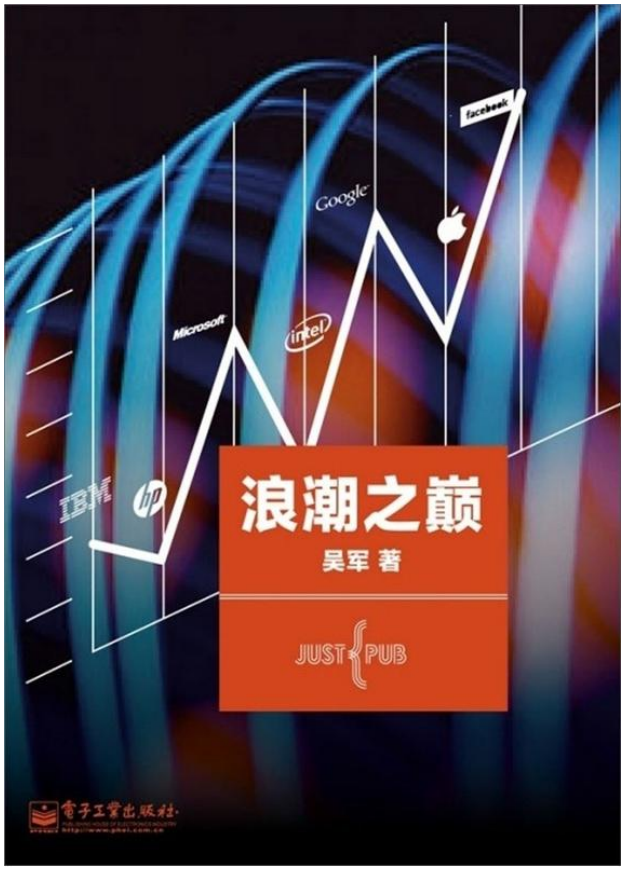
熊永平@网络技术研究院

ypxiong@bupt.edu.cn

周一13:30-15:30@教三328

2012.11.5


淝水古战场
晋



虎嗅 HUXIU.com

精选
有料
干货
有用
细节
内幕

课程进度

~~开始&作业一(9.3)~~ → ~~实验一讨论(9.17)~~ → ~~实验二(10.8)~~

↓

~~实验二讨论(10.22)~~ → 实验三(11.5) → 实验三讨论(11.19)

↓

实验四（12.3） → 实验四讨论（12.17） → 总结(1.7)


四个实验
多线程、网络编程、索引树……
1. 海量字符串
处理大规模字符串
2. 站点分析器
爬取站点所有页面链接并分析
3. Web服务器
处理高并发的服务器
4. 索引助手
对文件建立索引结构
反馈调整

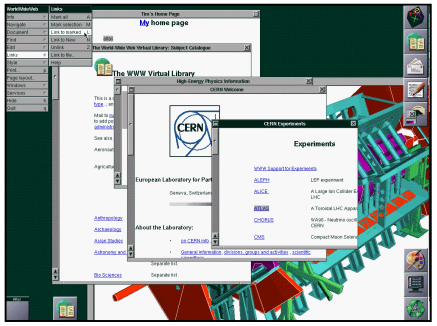
实验三：高性能Web服务器

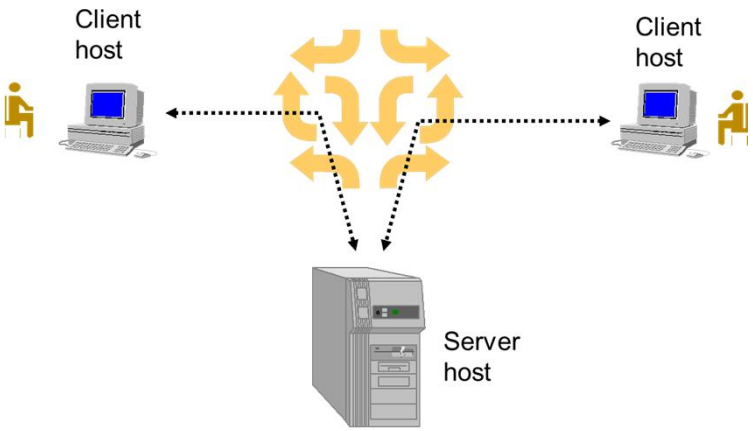
实验目标

- 目标
  - 设计一个web服务器程序，实现
    - 提供静态文件访问
    - 支持高并发
    - 日志管理
- 编程技能
  - C++语言
  - 高并发服务器
  - 多线程与线程池
  - 网络编程框架
  - 复用

Web应用

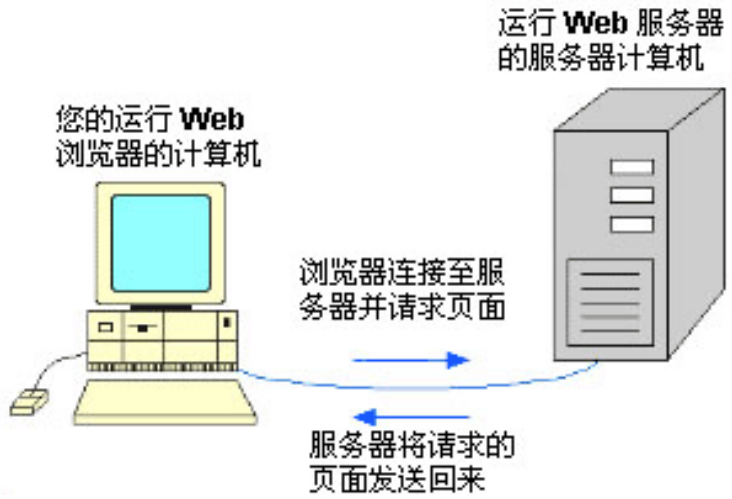
- WWW发明
  - 之前的Email、FTP、BBS应用
  - 可以使用各种协议
- HTTP协议
  - 实现WWW内容传输
- Client/Server计算模式
  - 客户端专注UI
  - 服务器集中在数据管理应用逻辑
  - 服务器被多个客户共享
- B/S架构
  - 以通用的浏览器为客户端(表示层)，企业逻辑代码(功能层)运行于WEB服务器上，程序员主要编写服务器端程序，这种结构开发效率比C/S结构低，但维护方便



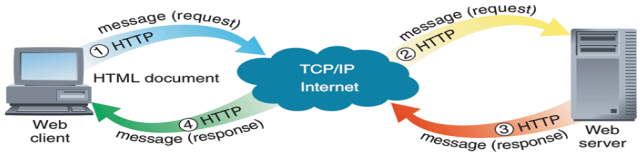


Web服务器

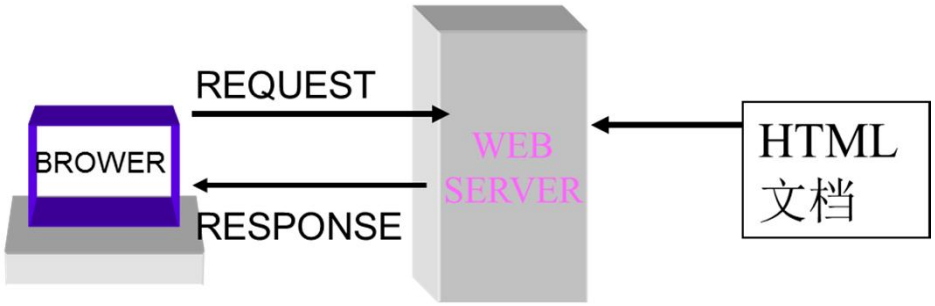
- 语义
  - 存储一个Web站点
    - 包括html/css、js脚本、数据文件等
  - 包括
    - 存储数据的硬件（服务器）
      - 大磁盘、高带宽和高速处理器
    - 响应请求的软件（HTTP服务器）
- 应用
  - 互联网公共网站
  - 企事业单位业务应用系统
  - 交易系统



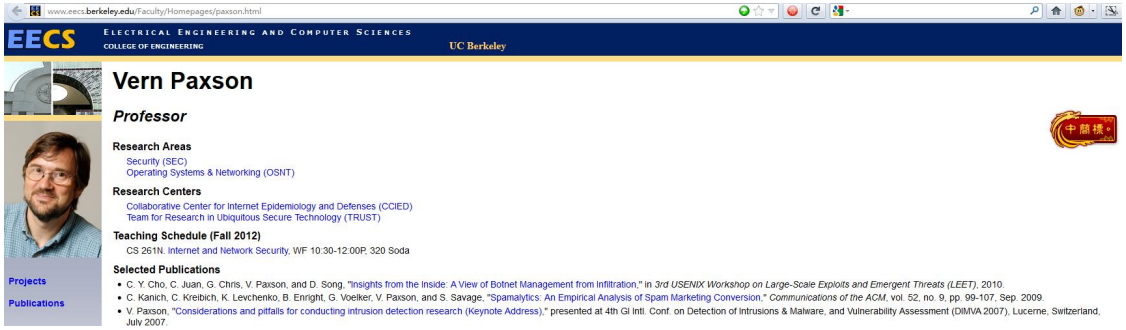
Web应用之静态网站



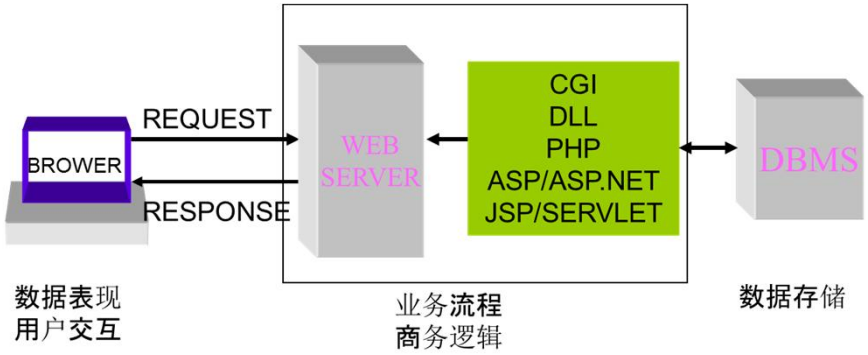
静态网页的优点是设计简单，缺点是：如果要修改内容，必须修改页面文件并重新上传。

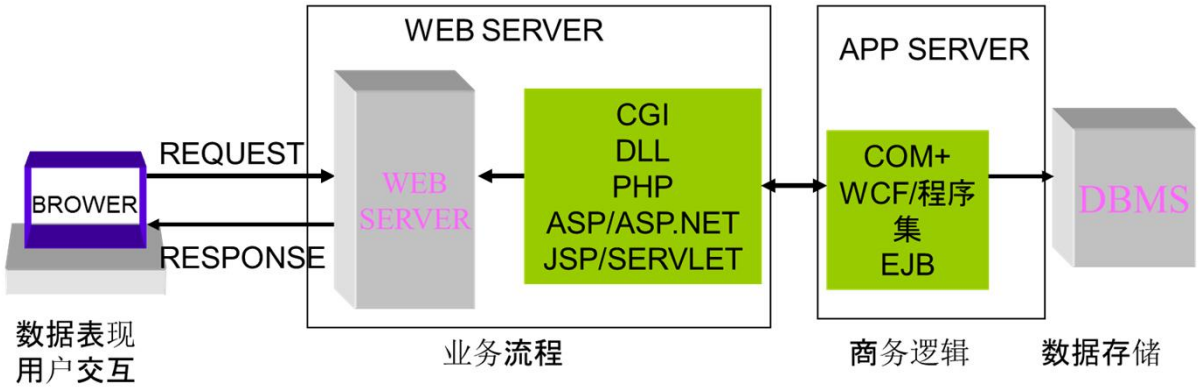


静态B/S结构



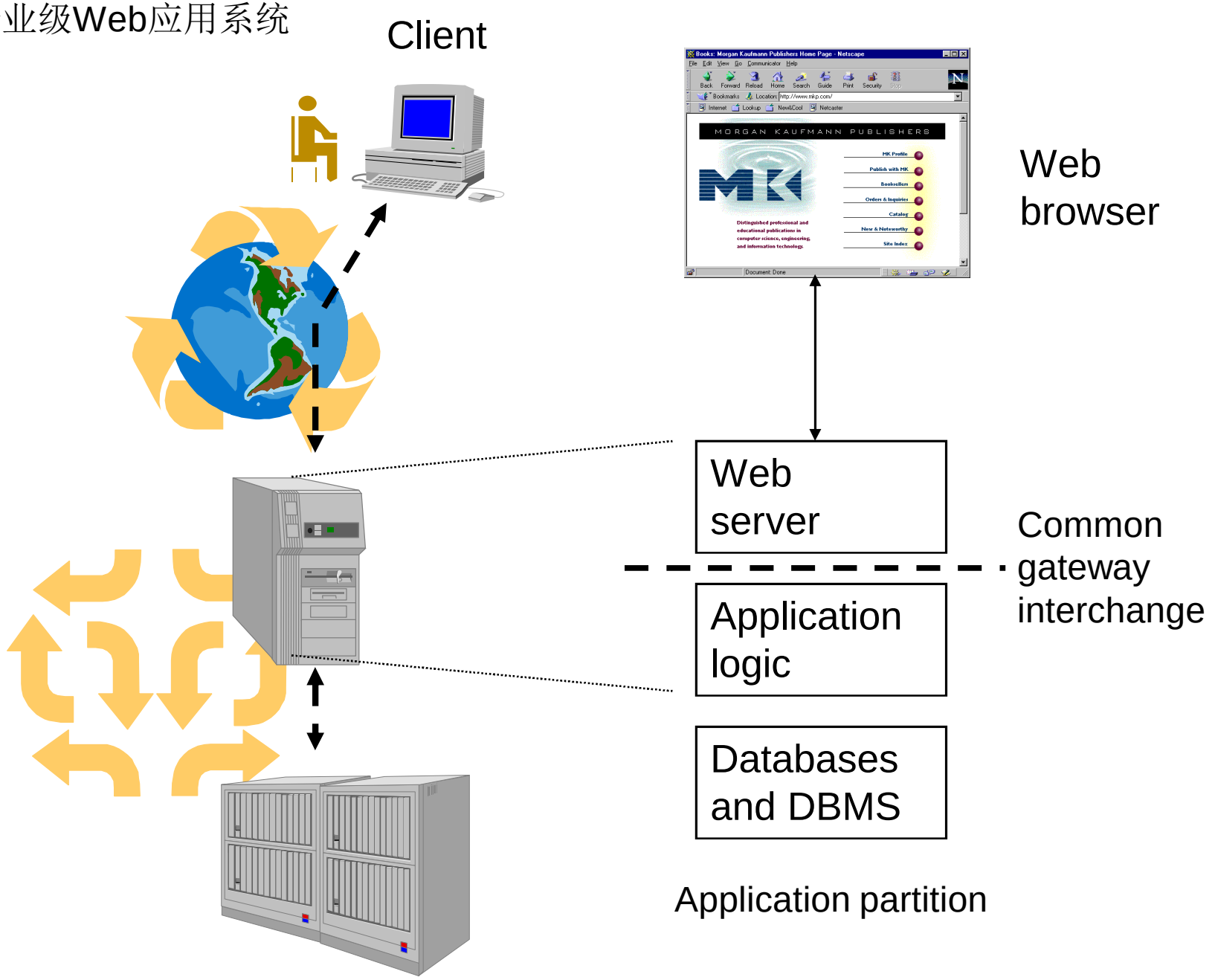
动态Web应用之企业信息系统

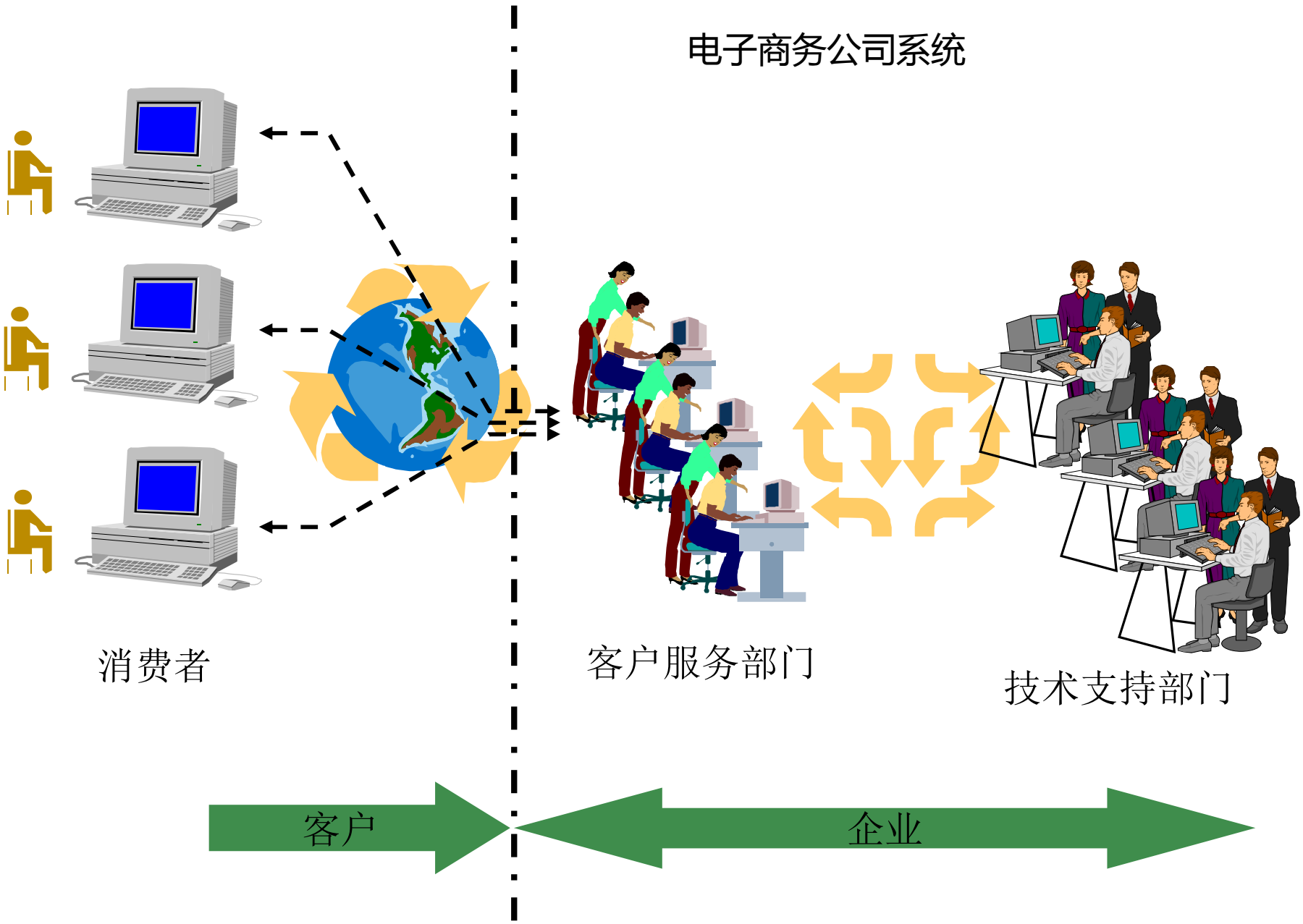


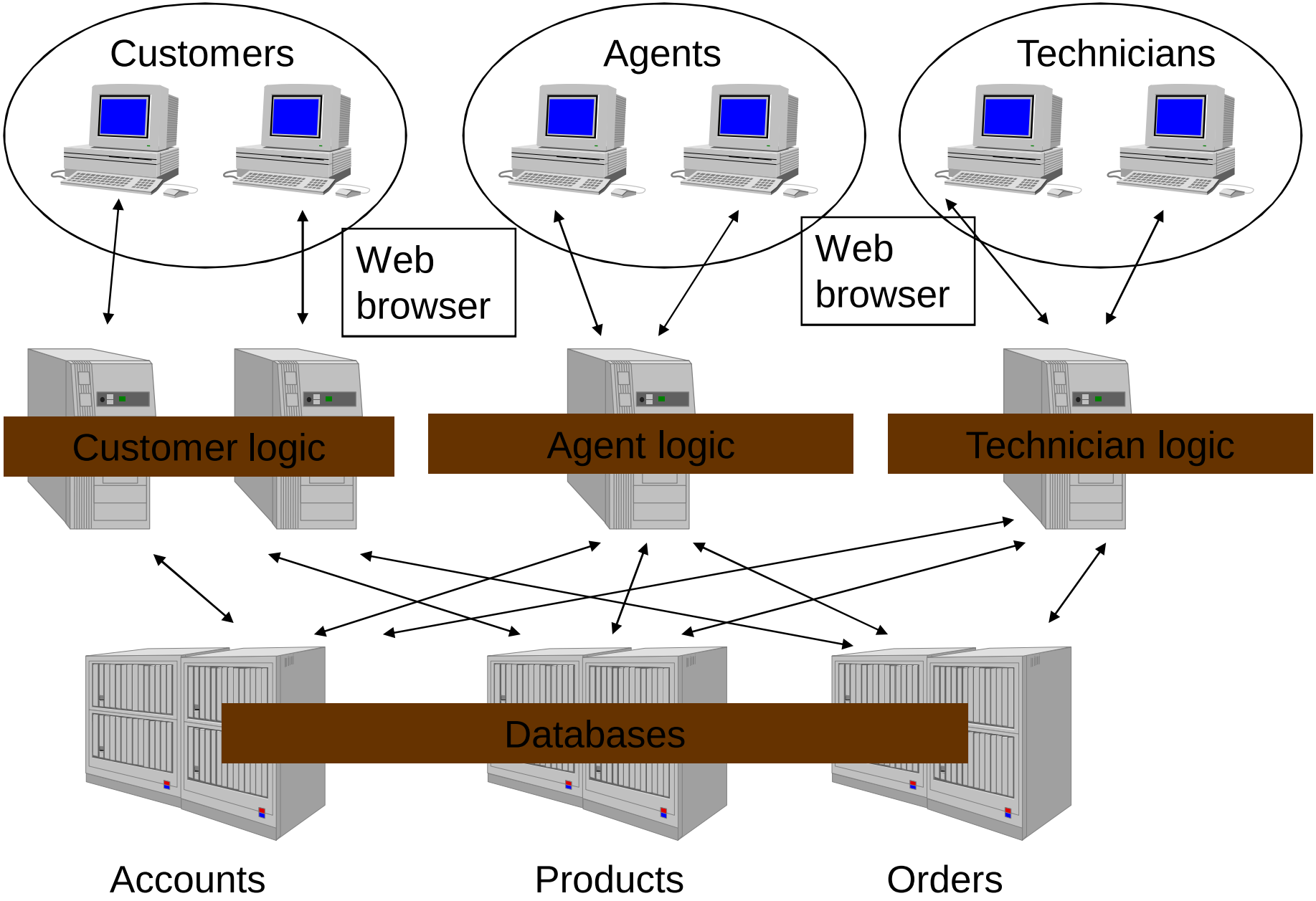


当浏览器向WEB服务器发出资源请求时，服务器加载相应应用程序（动态页面），解释执行后将执行结果传回给浏览器。动态网页还可以与数据库进行交互。这是目前最常用的结构。

普通企业级Web应用系统

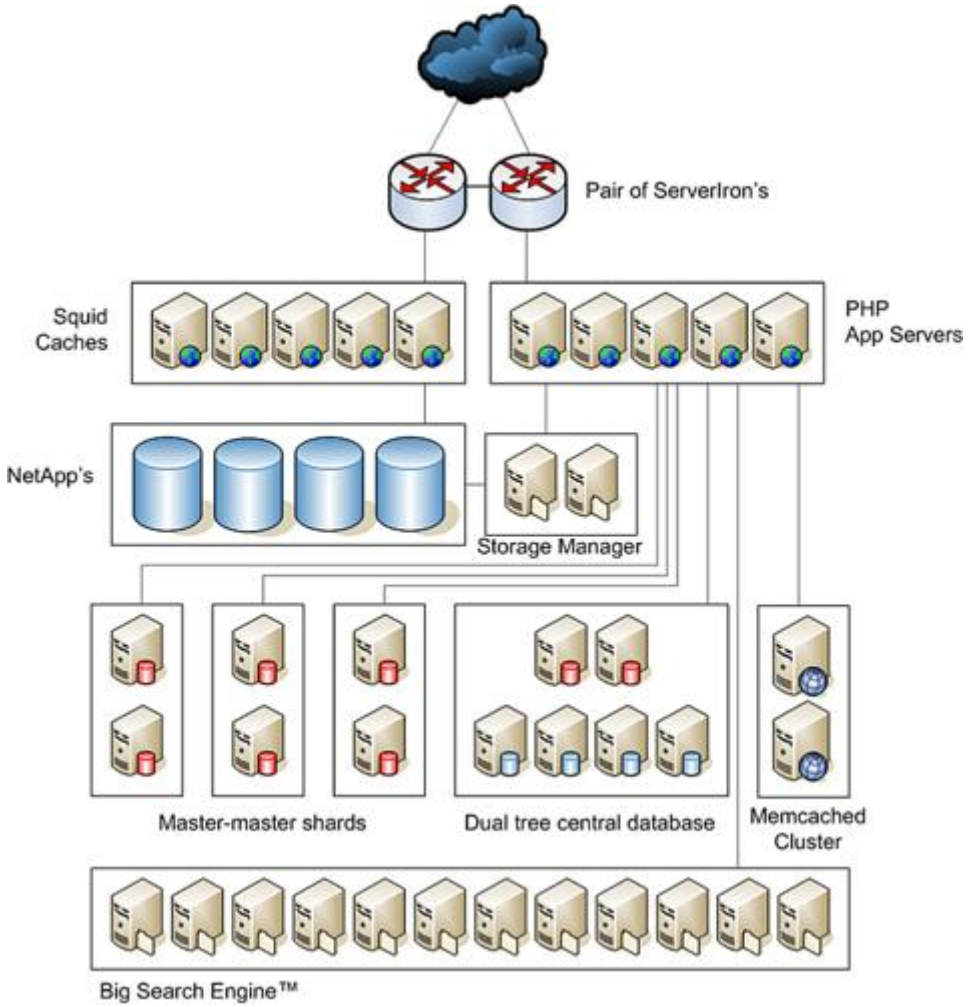



电子商务公司系统
消费者
客户服务部门
技术支持部门
客户
企业


Customers
Agents
Technicians
Web browser
Web browser
Customer logic
Agent logic
Technician logic
Databases
Accounts
Products
Orders

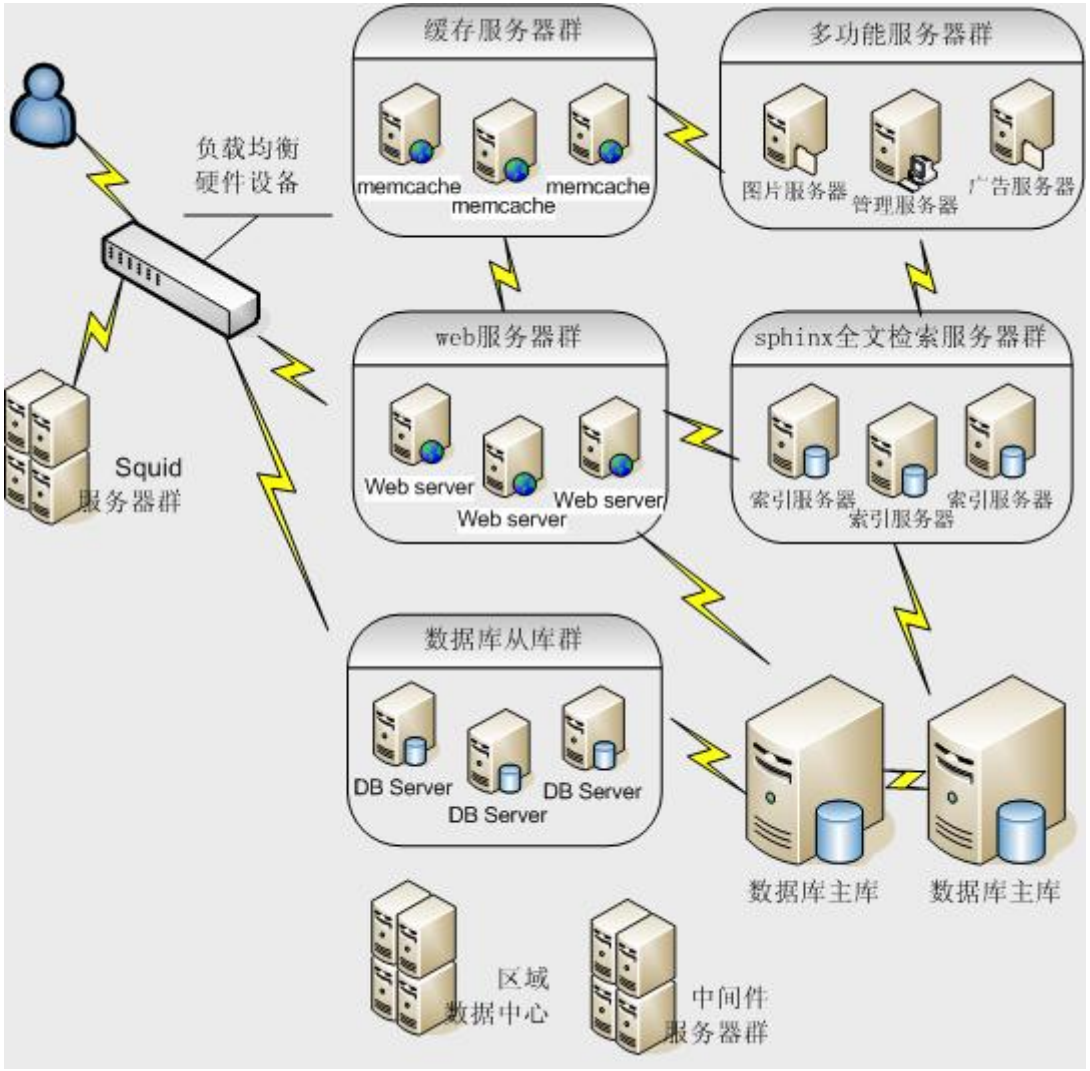
动态Web应用之Flickr

- 每天40亿次查询请求
- 每秒38，000次
- 大约4亿7000万张照片
- 超过2 PB 存储，其中数据库12TB
- 每天新增图片超过 40万（周日峰值超过200万，约1.5TB)
- 响应4万个照片访问请求
- 处理10万个缓存操作
- 运行13万个数据库查询

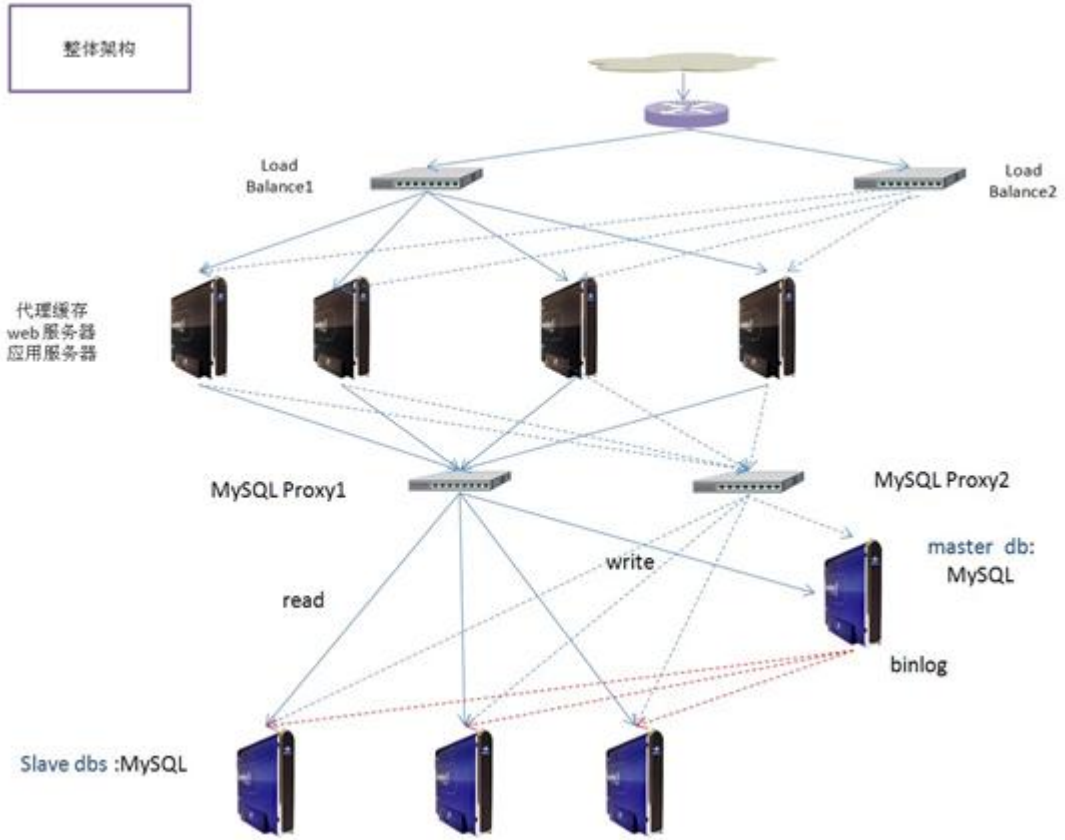


动态Web应用之eBay

2004 eBay
服务器2400台

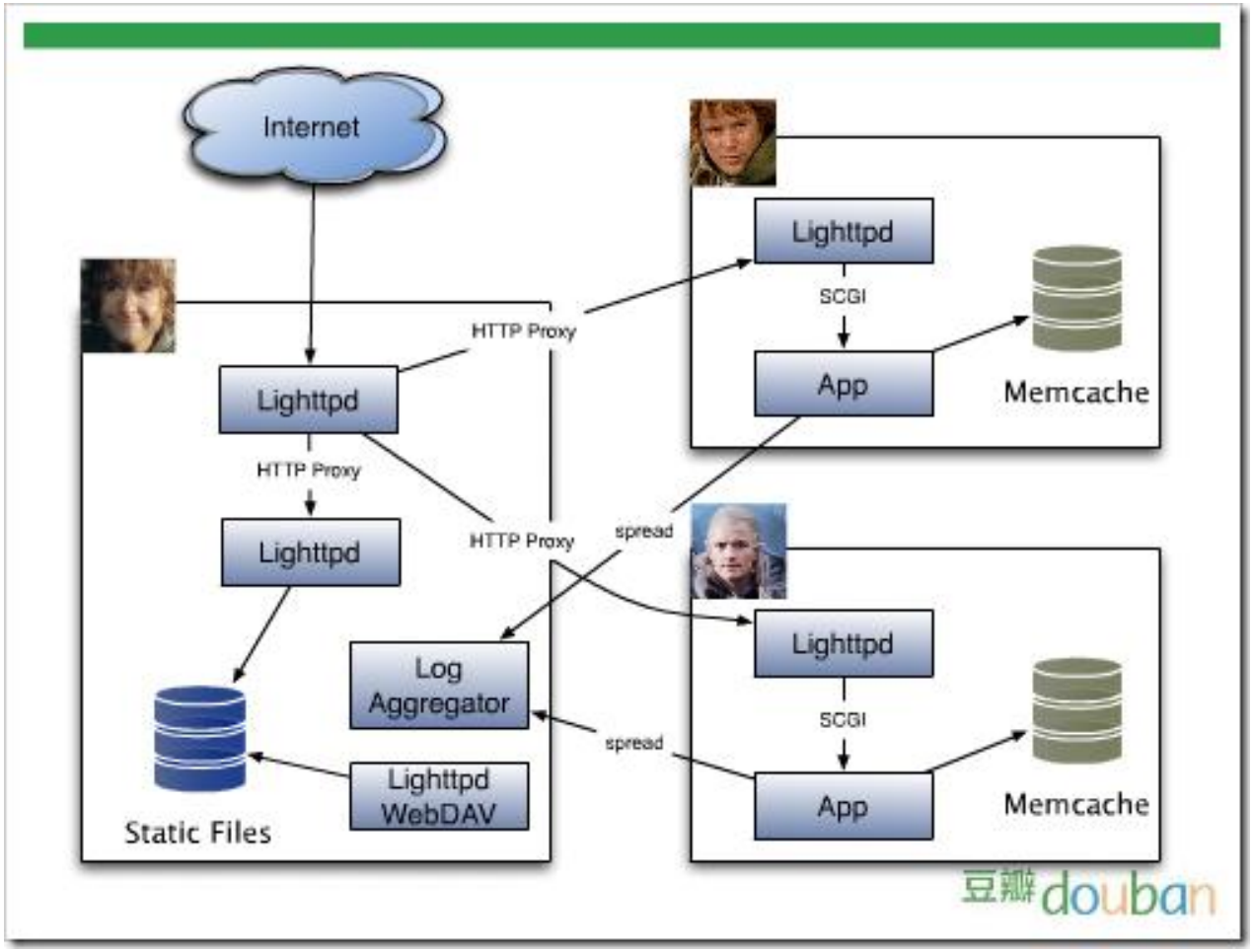


动态Web应用之12306



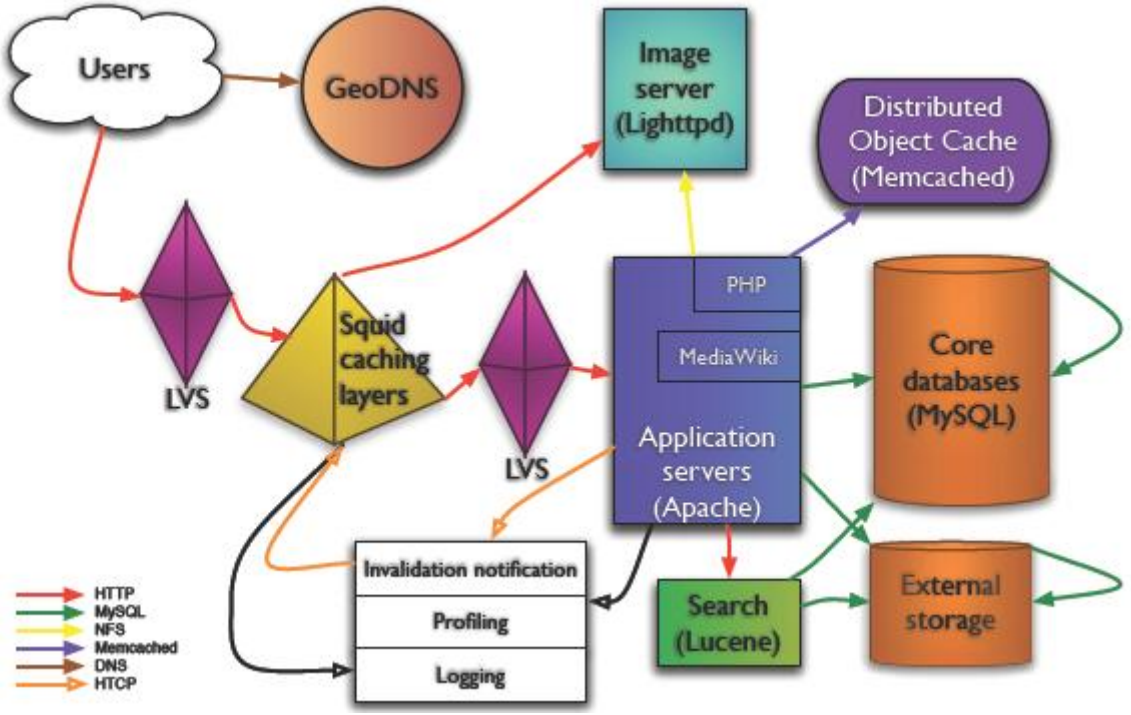
动态Web应用之douban

每天6.4M请求
几百万的图片



动态Web应用之Wikipedia

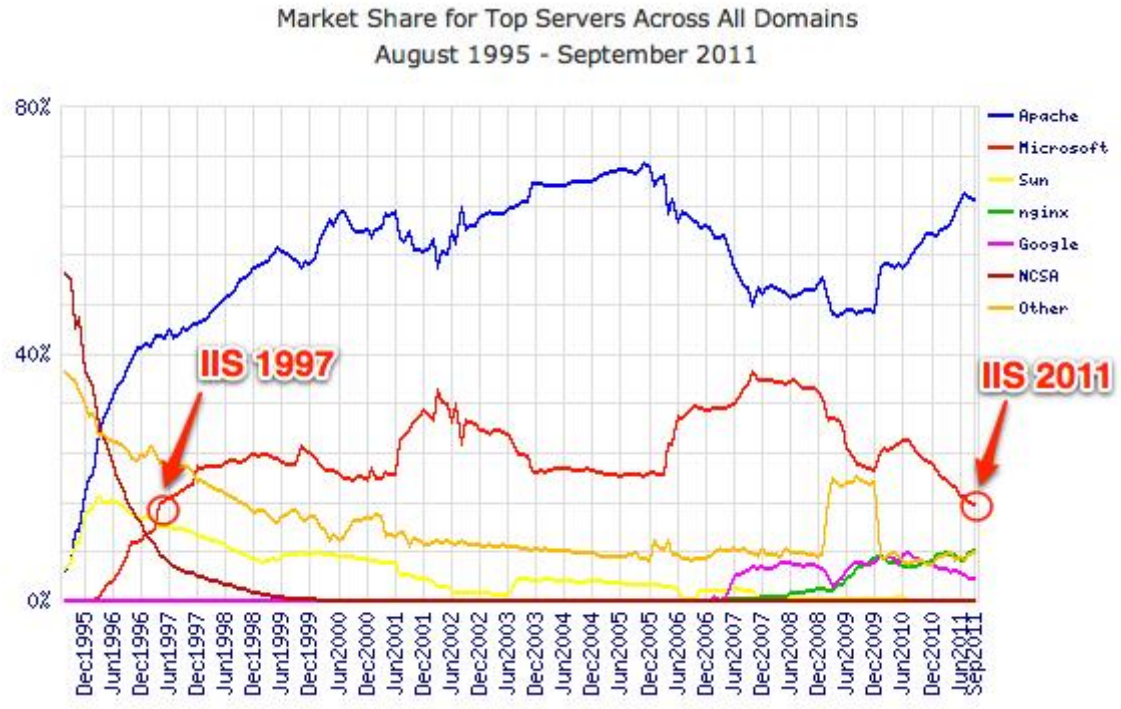
- 维基百科(WikiPedia.org)位列世界第八大网站
- 峰值每秒钟3万个 HTTP 请求
- 每秒钟 3Gbit 流量, 近乎375MB
- 350 台 PC 服务器

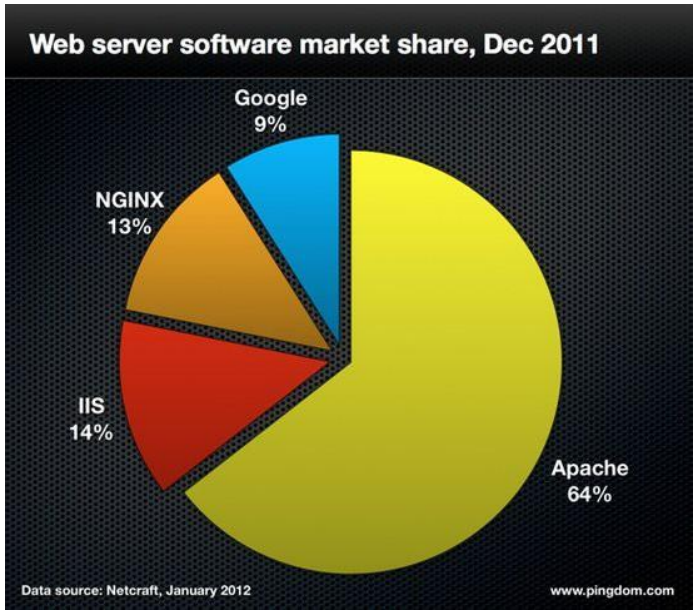


高性能网站架构

- 计算机系统结构
  - 核心问题：性能和成本 tradeoff
  - 技术手段：并行和cache
- 实现方案
  - 负载均衡
    - 网络层：DNS负载均衡、IP层均衡（LVS：NAT,IPTUNNEL,DR）
      - 无法区分应用语义，但是效率高
    - 反向代理负载均衡
      - 应用层均衡（http：nginx），解析和利用应用语义，效率低
  - 数据库
    - 数据的水平切分及垂直切分、数据库读写分离、避免分布式事务
  - 缓存
    - 数据库缓存、服务器缓存/页面缓存/数据缓存/静态化、反向代理缓存
  - 分布式文件系统(MogileFS)、异步(MessageQueue)

工业界Web服务器应用





淘宝网（阿里巴巴）: Linux操作系统 + Web 服务器: Nginx
新浪：FreeBSD + Web 服务器：Apache/Nginx
Yahoo：FreeBSD + Web 服务器：自己的
Google: 部分Linux + Web 服务器：自己的
百度：Linux + Web 服务器: Apache
网易：Linux + Web 服务器: Apache
eBay: Windows Server 2003/8 (大量) + Web 服务器：Microsoft IIS
MySpace: Windows Server 2003/8 + Web 服务器：Microsoft IIS

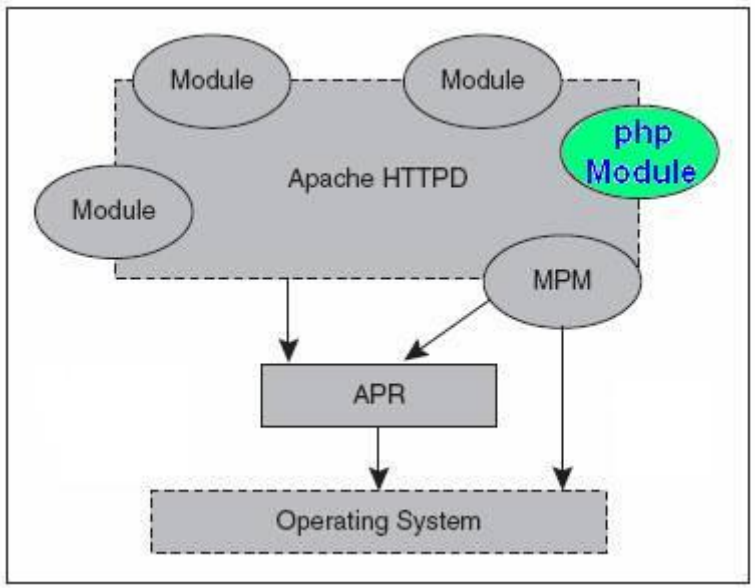
Linux + Apache + PHP + MySQL
Linux + Apache + Java (WebSphere) + Oracle
Windows Server 2003/2008 + IIS + C#/ASP.NET + 数据库

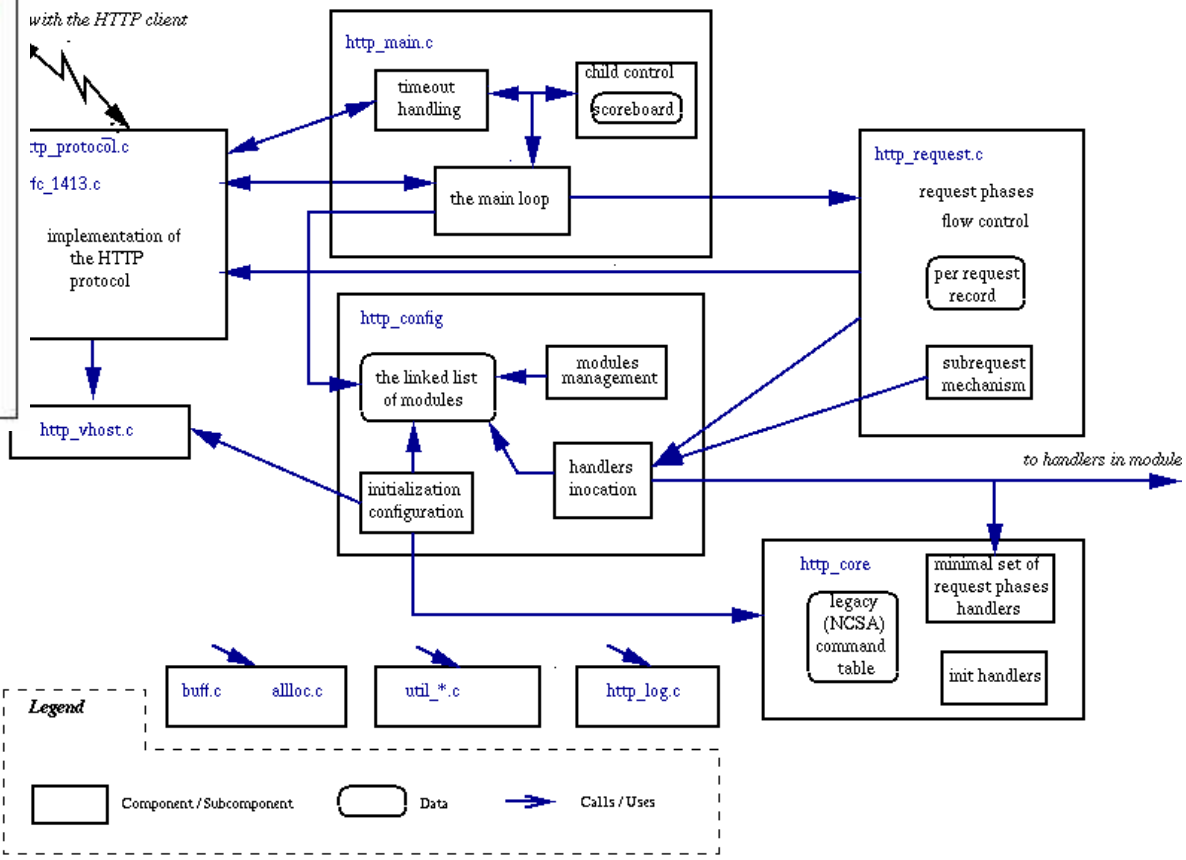
怎么设计高性能Web服务器？

高性能？

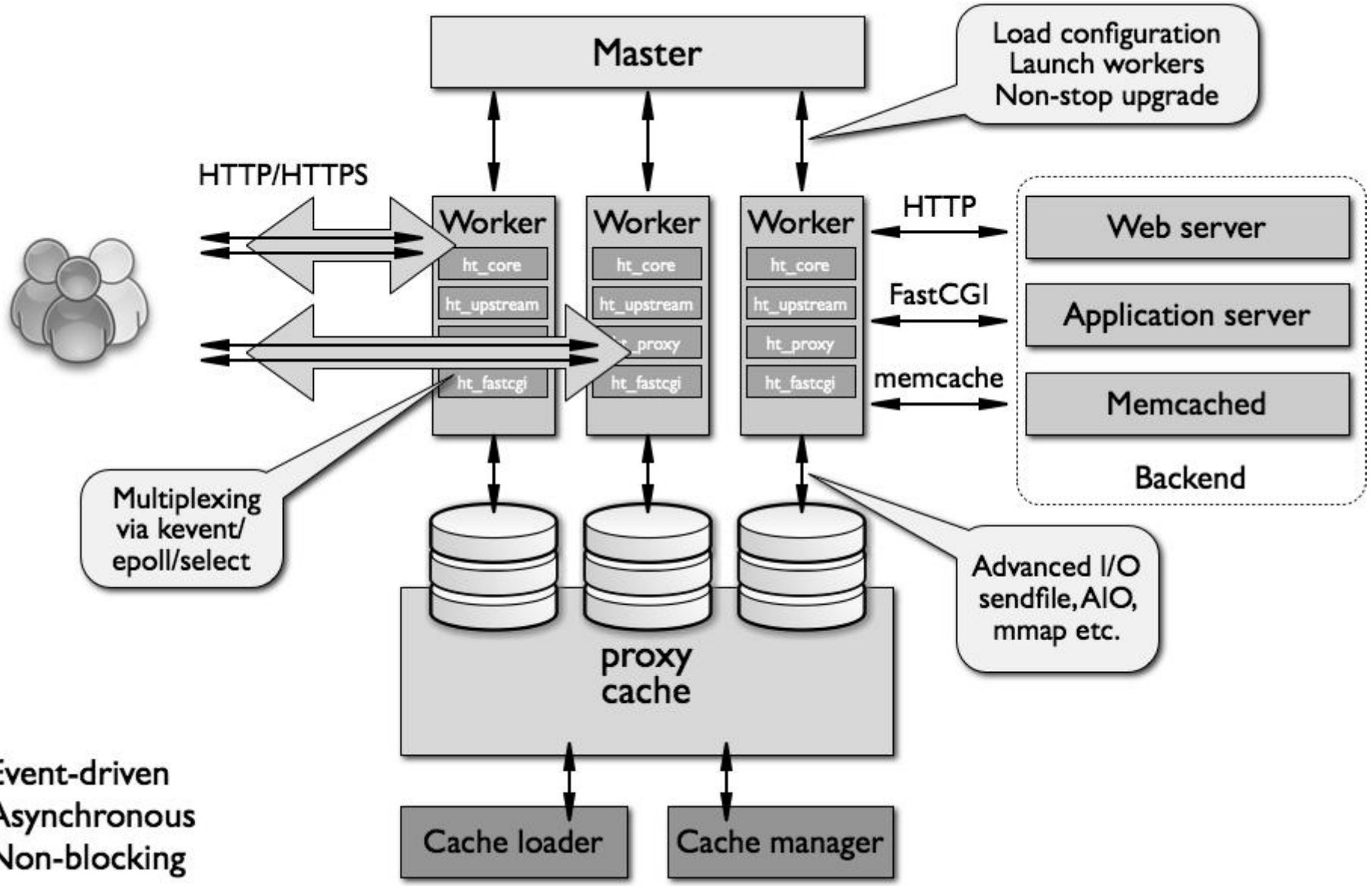
- 并发请求数
  - C10K， C1000K问题，每秒钟处理HTTP请求的数量
  - I/O模型
- 请求响应速度
  - 静态文件传输，可能大文件
  - 压缩传输，gzip压缩
  - 考虑Cache

Apache Overview





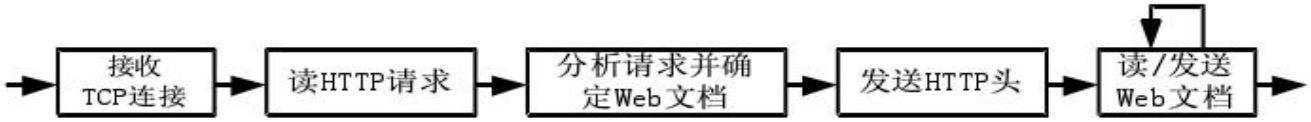
Nginx

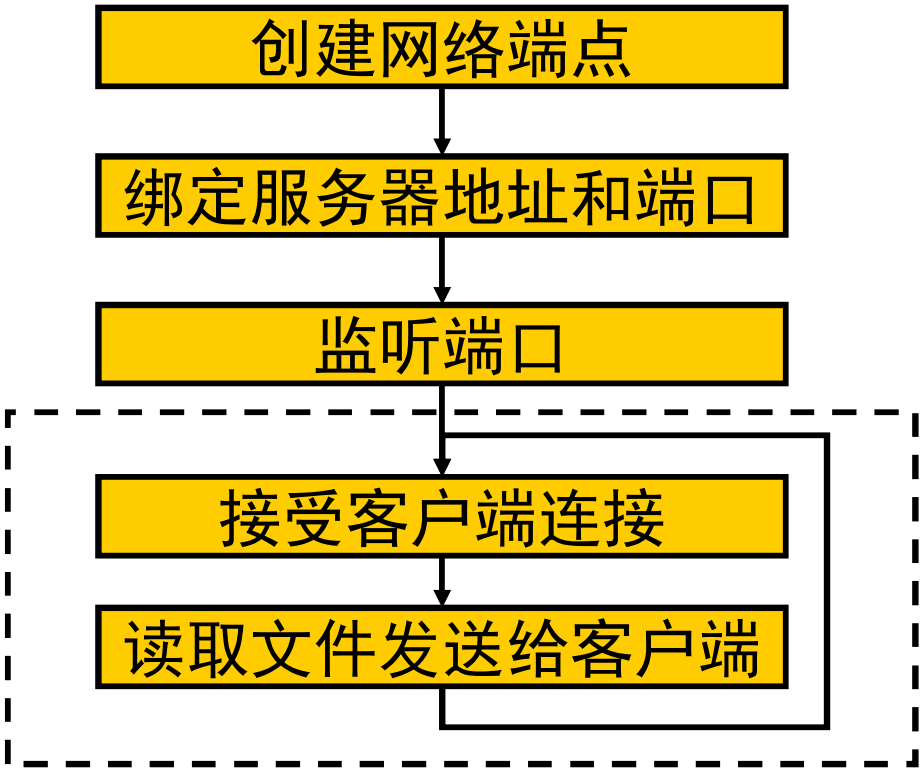


- Event-driven
- Asynchronous
- Non-blocking

V1-循环Web服务器

- 基本的HTTP处理流程



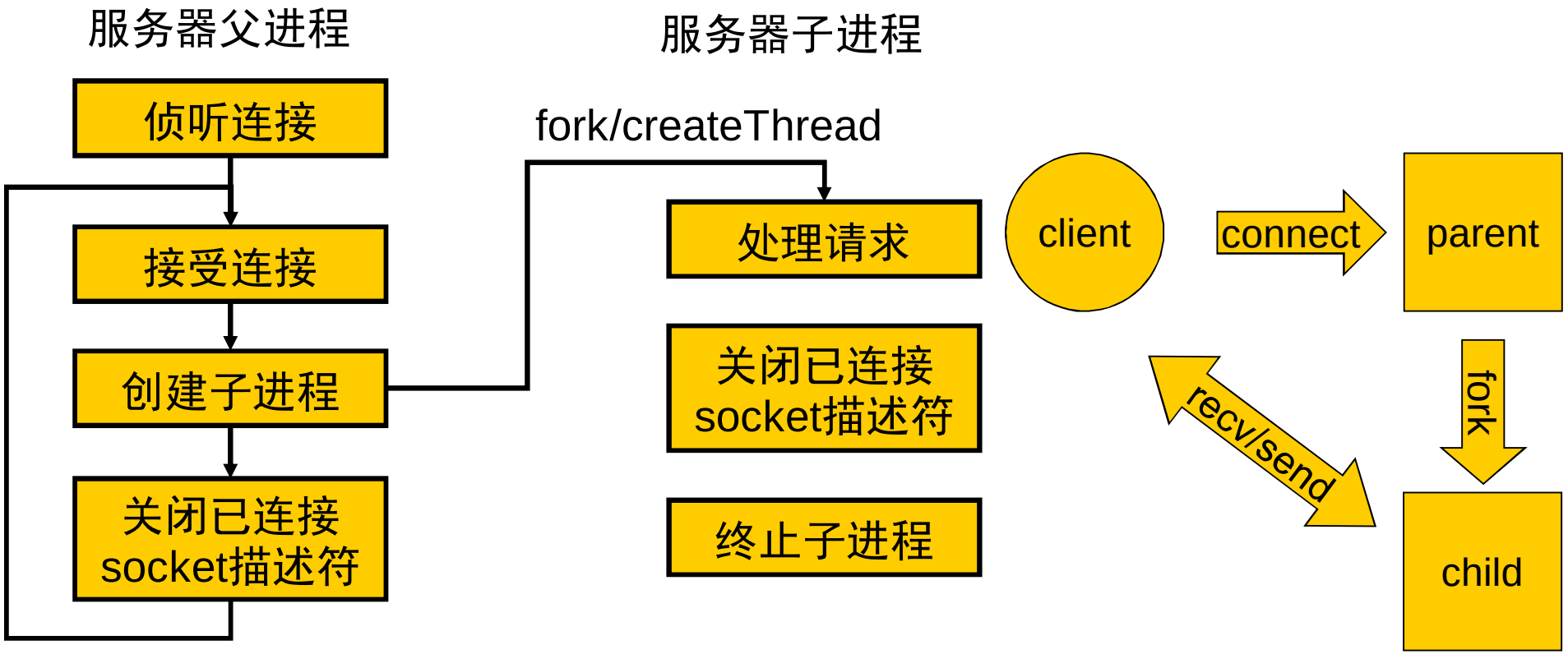


分析

- IO并发程度不够
  - 同一时刻只能处理一个客户端请求
  - 如果服务器需要处理一下任务，不再接收客户
    - Database
    - Local file system
    - Other TCP/UDP servers
    - ...
  - 如何解决？
    - 并发

V2-On Demand并发Web服务器

- 模型
  - 动态为每个客户端创建子进程/子线程
  - 远古时代的Apache



代码框架

```
pthread_t pid;
int   listenfd,  connfd;
listenfd = Socket( ... );
    /* fill in sockaddr_in{} with server's well-known
port */
Bind(listenfd, ... );
Listen(listenfd, LISTENQ);

for ( ; ; ) {
    connfd = Accept (listenfd, ... );    /* probably
blocks */

   if(pthread_create(&pid, NULL,(void*)sub_task,
(void*)&listenfd) != OK)
    perror("pthread_create");
}

    Close(connfd);          /* parent closes connected
socket */
}
void * sub_task(void * listenfd)
{
     doit(listenfd);        /* process the request */
     Close(listenfd);       /* done with this client */

}
```

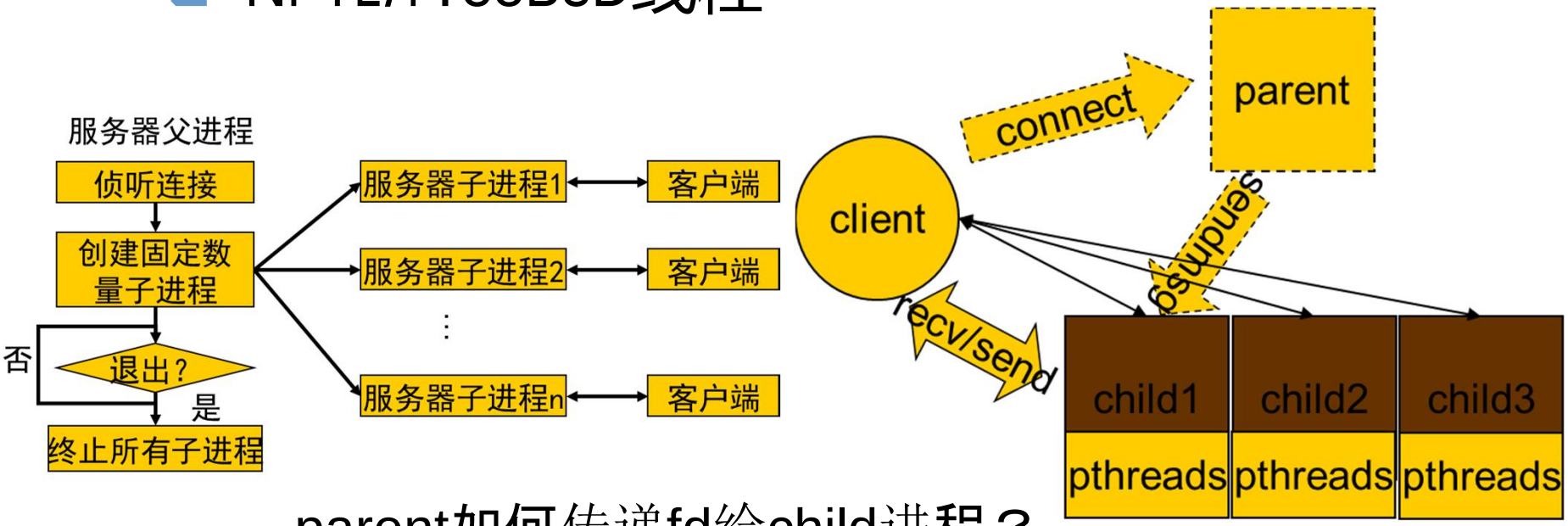
分析

- 特点
  - 一个子进程/线程对应一个客户端
  - 可以同时处理多个客户端
- 优缺点
  - 模型简单、可靠
  - 创建子进程开销较大，进程/线程切换开销大，并发受限
    - 每个进程都有自己的地址空间和上下文信息，大量进程会导致大量的内存消耗，从而影响性能频繁切换进程，也会造成开销
    - 5-10us每次
    - 适合长连接客户，例如企业信息系统

Benchmark	Operating System	Operation	Time per Op
Spawn New Process	NT	spawnl()	12.0 ms
	Linux	fork()/exec()	6.0 ms
Clone Current Process	NT	NONE	N/A
	Linux	fork()	1.0 ms
Spawn New Thread	NT	CreateThread()	0.9 ms
	Linux	pthread_create()	0.3 ms
Switch Current Process (20 runnable processes)	NT	Sleep(0)	0.010 ms
	Linux	sched_yield()	0.019 ms
Switch Current Thread (20 runnable threads)	NT	Sleep(0)	0.006 ms
	Linux	sched_yield()	0.019ms

V3-进程/线程池Web服务器

- 预创建子进程/线程
  - 数量固定/动态调整
  - NPTL/FreeBSD线程



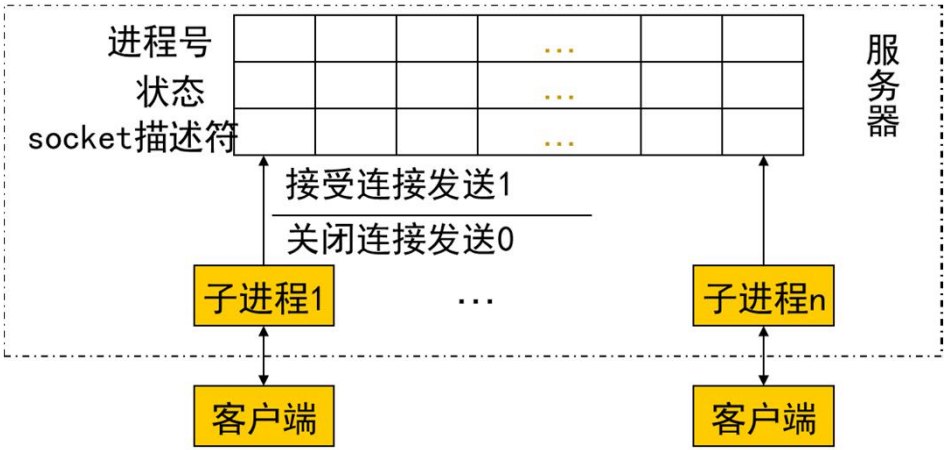
parent如何传递fd给child进程？
sendmsg/recvmsg

分析

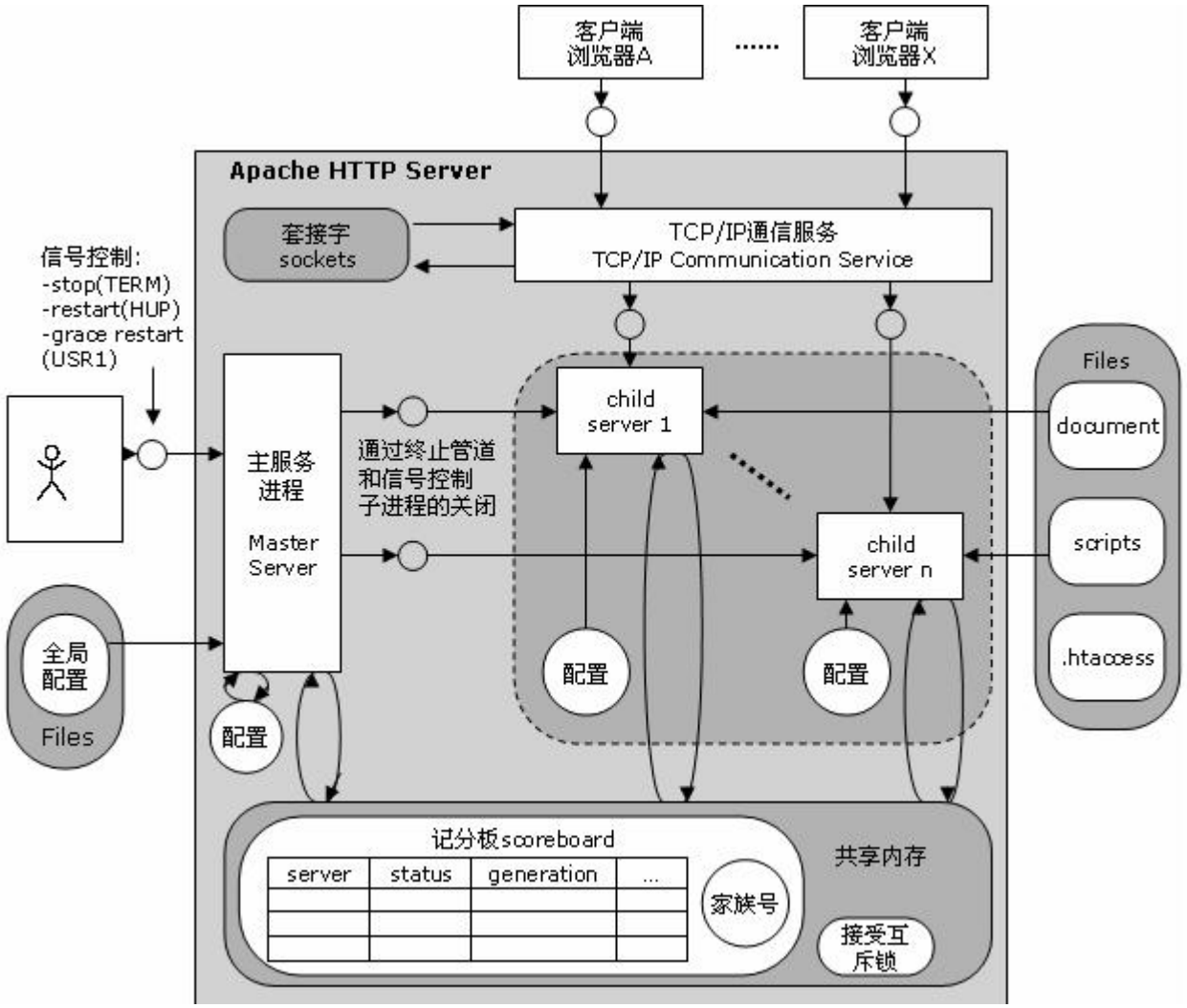
- 实例
  - apache prefork MPM
  - apache worker MPM
- 优点
  - 伸缩性较好
  - 响应速度快，节省创建子进程时间
- 缺点
  - 并发性能差
  - 需要预先估计创建进程的数量
  - 预创建子进程数量少时将导致客户端等待
  - 预创建子进程数量多时将浪费系统资源

实现框架

- 动态调整进程数实现过程
  - 服务器建立socket，并创建一定数量子进程
  - 服务器父进程维护所有子进程的状态表，父进程和子进程通过管道通信
  - 子进程接收连接时给父进程发1，关闭连接时发0
  - 父进程收到1时检查空闲子进程数目是否小于下限，小于下限则创建新的子进程
  - 父进程收到0时检查空闲子进程数目是否大于下限，大于下限则终止一些子进程



Apache实现


客户端
浏览器A
.....
客户端
浏览器X
Apache HTTP Server
套接字
sockets
TCP/IP通信服务
TCP/IP Communication Service
信号控制:
-stop(TERM)
-restart(HUP)
-grace restart
(USR1)
主服务
进程
Master
Server
通过终止管道
和信号控制
子进程的关闭
child
server 1
child
server n
Files
document
scripts
.htaccess
全局
配置
Files
配置
配置
配置
记分板scoreboard
server
status
generation
...
家族号
共享内存
接受互
斥锁

并发Web服务器

- 如有可能 不要尝试多线程
- 用原语高级点的库
- 想清楚了 再做
- 很难调试 做好诊断设施

二维I/O模型

Synchronous IO操作导致请求进程阻塞，直到I/O操作完成

阻塞I/O
非阻塞I/O
I/O多路复用
事件驱动I/O (SIGIO)

阻塞I/O
I/O多路复用（Select）
WSAAsyncSelect（异步选择）
WSAEventSelect(事件选择)
Overlapped（重叠）

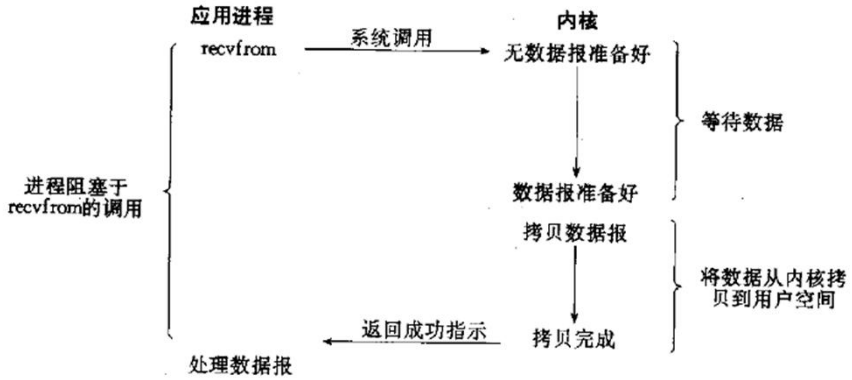
异步I/O (POSIX aio_函数)

IO Completion Port(IOCP，完成端口)

Asynchronous 异步I/O操作不导致请求进程阻塞

阻塞I/O模式

- 模型
  - 创建时默认阻塞模式，当socket不能立即完成I/O操作时，进程或线程进入等待状态，直到操作完成
- 以recv函数为例，
  - 阻塞模式下，程序调用了recv函数后将一直处于等待状态，直到接收完数据
  - 银行柜台存钱



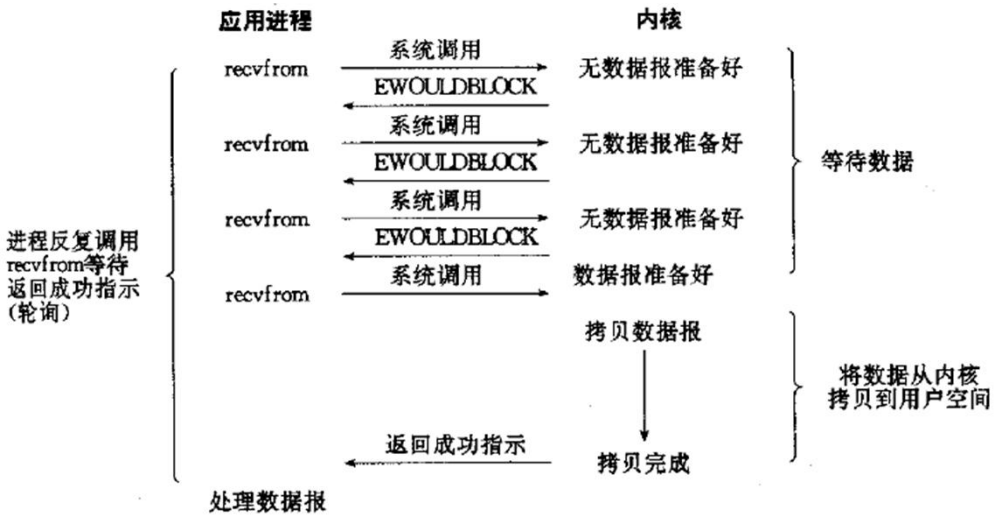
这种模型非常经典，也被广泛使用，优势在于非常简单，等待的过程中占用的系统资源微乎其微，程序调用返回时，必定可以拿到数据；

但简单也带来一些缺点，程序在数据到来并准备好以前，不能进行其他操作，需要有一个线程专门用于等待，这种代价对于需要处理大量连接的服务器而言，是很难接受的

非阻塞I/O模式（Non-Blocking）

- 语义
  - 把socket设置成非阻塞模式，无数据时，也不会进入等待，而是立即返回特定错误
- 实现方法
  - 使用ioctlsocket设置非阻塞模式
  - 示例代码：

```
U_long ul = 1;
SOCKET s =
socket(AF_INET,SOCK_STREAM,0);
ioctlsocket(s,FIONBIO,(u_long *)&ul);
```

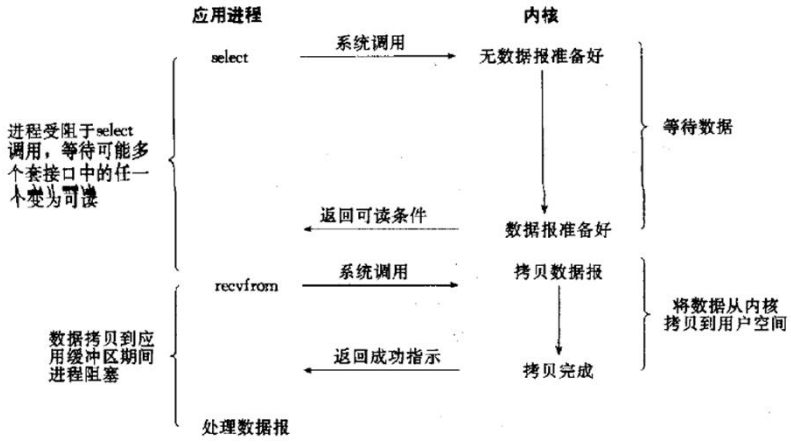


这种模式在没有数据可以接收时，可以进行其他的一些操作，比如有多个socket时，可以去查看其他socket有没有可以接收的数据；

实际应用中，这种I/O模型的直接使用并不常见，因为它需要不停的查询，而这些查询大部分会是无必要的调用，白白浪费了系统资源；非阻塞I/O应该算是一个铺垫，为I/O复用和信号驱动奠定了非阻塞使用的基础

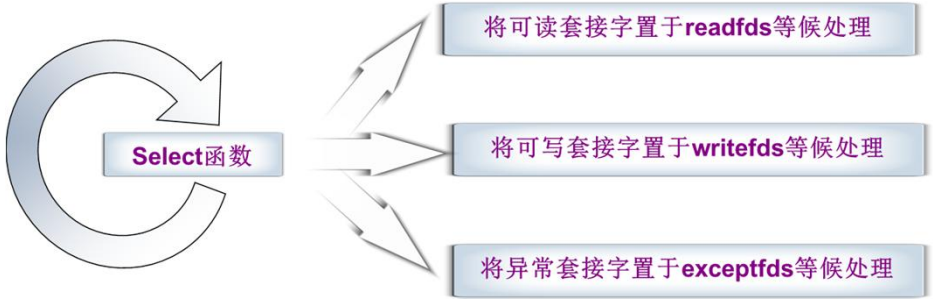
I/O复用（multiplexing）

- 思想
  - 多人存钱，专人查询
  - 查询多个socket可读或可写是否准备好的状态
    - 并发服务器管理多个客户连接IO、本地文件IO等
    - 要求为非阻塞Socket
- 实现方式
  - Select经典方式(*nix，Windows）
  - poll、基于内核通知的epoll（Linux）
  - kqueue（freebsd）
  - Epoll和kequeue多连接时高性能



Select方法

- 功能
  - 确定一个或多个套接字是否有连接到达、已连接socket是否有数据到达、已连接socket是否可以写数据，以便执行同步I/O
- 实现
  - 用户态实现
  - 最大并发数限制为1024
  - 线性扫描全FD 集合效率低




```
#include <sys/select.h>
#include <sys/time.h>

将套接字集合清空  FD_ZERO(*set)
将某个套接字从集合中清除  FD_CLR(s, *set)
检查套接字s，是否在集合set中 FD_ISET(s, *set)
将套接字s放入集合set中 FD_SET(s, *set)
int select(
      int nfds, //忽略，仅为了与berkaley套接字兼容
      fd_set * readfds, //一个套接字集合，用于检查可读性
      fd_set * writefds, //一个套接字集合，用于检查可写性
      fd_set * exceptfds, //一个套接字集合，用于检查错误
      const struct timeval * timeout //指定此函数等待的最
长时间，若为NULL，则最长时间为无限大
)
当有 I/O 事件到来时， select 通知应用程序有事件到了快去处理，
而应用程序必须轮询所有的 FD 集合，测试每个 FD 是否有事件发生，
并处理事件；代码像下面这样：
int res = select(maxfd+1, &readfds, NULL, NULL, 120);
if (res > 0){
    for (int i = 0; i < MAX_CONNECTION; i++){
        if (FD_ISSET(allConnection[i], &readfds)){
            handleEvent(allConnection[i]);
        }
    }
}
// if(res == 0) handle timeout, res < 0 handle error
```

epoll方法

- 功能
  - Select类似，确定哪些IO准
- 实现
  - 内核事件通知方式
  - API
- 优点
  - 不限制FD集合大小
  - 避免了FD轮询
  - 内核事件通知效率高

int epoll_create(int size);

生成一个 Epoll 专用的文件描述符，其实是申请一个内核空间，用来存放你想关注的 socket fd 上是否发生以及发生了什么事件。 size 就是你在这个 Epoll fd 上能关注的最大 socket fd 数，大小自定，只要内存足够。

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);

控制某个 Epoll 文件描述符上的事件：注册、修改、删除。其中参数 epfd 是 epoll_create() 创建 Epoll 专用的文件描述符。相对于 select 模型中的 FD_SET 和 FD_CLR 宏。

int epoll_wait(int epfd,struct epoll_event * events,int maxevents,int timeout);

等待 I/O 事件的发生；参数说明：

epfd: 由 epoll_create() 生成的 Epoll 专用的文件描述符；

epoll_event: 用于回传代处理事件的数组；

maxevents: 每次能处理的事件数；

timeout: 等待 I/O 事件发生的超时值；

返回发生事件数。

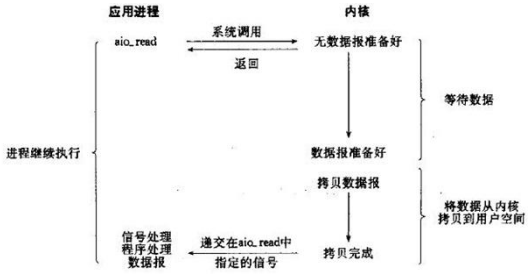
```
#include <sys/select.h>
#include <sys/time.h>
int res = epoll_wait(epfd, events, 20, 120);

for (int i = 0; i < res;i++)
{
    handleEvent(events[n]);
}
```

I/O多路复用分析

- 优点
  - 只需要一个进程来处理所有客户机请求
  - 没有创建和管理进程的开销，系统资源消耗少
  - 没有进程间通信
- 缺点
  - 服务器必须依次处理所有的请求，编程较复杂
  - 服务器循环处理所有就绪客户端，可能会造成延时较长

其它同步I/O模型

- Linux 信号驱动I/O
  - 在socket准备好的时候用信号的方式进行通知，然后应用程序从内核读取数据
  - 区分引起signal的socket操作很困难

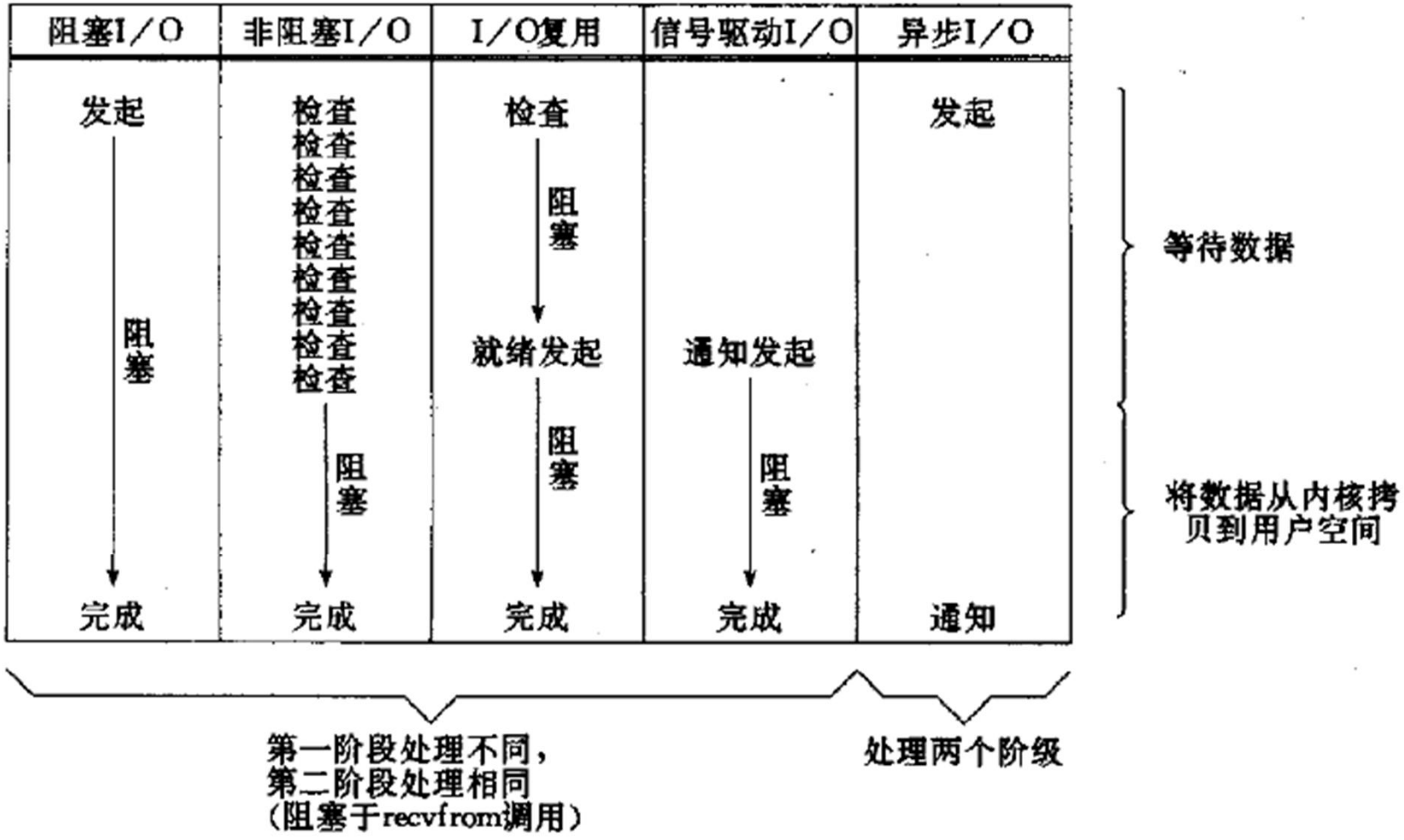


- Windows平台模型
  - WSAAsyncSelect
    - 将socket设为非阻塞，出现网络事件发消息到socket绑定的窗口
  - WSAEventSelect
    - 不用窗口而用Event来传递socket事件，同时用WSAWaitForMultipleEventsEvent等待到Event事件，再通过WSAEnumNetworkEvents获取哪些网络事件发生
- 总结
  - 阻塞I/O模型、非阻塞I/O模型、I/O复用模型和信号驱动I/O模型都是同步I/O模型，因为真正的I/O操作将阻塞进程

Asynchronous I/O模型

- 思想
  - 操作系统**完成IO操作**后，通知应用程序
- Linux aio
  - 把信号值与每个I/O操作关联，信号到达时，I/O操作已经由内核完成，应用只需要继续处理数据
  - 成为了POSIX标准，2.6内核支持，不支持socket
  - 淘宝系统使用
- Windows I/O Completion Port(IOCP)
  - 创建Completion I/O Port，系统附带创建一个请求队列，与IOCP配套产生一个线程池，将每个I/O操作加入IOCP，线程等待I/O操作完成，执行相应的回调函数
  - 迄今为止最为复杂的一种I/O模型，编程复杂
  - 多处理器系统上多个异步I/O的高效多线程处理方法

I/O模型区别（UNIX平台）



V4—I/O复用（事件驱动）服务器

- 单线程/进程事件驱动模型
  - 由多个IO描述符的IO事件驱动
  - 单个线程采用多路复用Select/epoll技术管理所有socket，socket全部设置non-blocking模式，由事件通知触发网络读写
  - 在处理大量连接时，是非常经典的线程模型之一
- 要点
  - 由于所有的socket操作都在一个线程中完成，所以必须保证在线程中，除了网络I/O操作以外，没有其他引发阻塞的调用
- 例子
  - 反向代理服务器Squid
  - 客户端与 WEB服务器数据，全IO操作

基于Epoll的服务器分析

```
#include <iostream>
#include <sys/socket.h>
#include <sys/epoll.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#define MAXLINE 10
#define OPEN_MAX 100
#define LISTENQ 20
#define SERV_PORT 5555
#define INFTIM 1000
void setnonblocking(int sock)
{
    int opts;
    opts = fcntl(sock, F_GETFL);
    if(opts < 0)  {
        perror("fcntl(sock, GETFL)");
        exit(1);
    }
    opts = opts | O_NONBLOCK;
    if(fcntl(sock, F_SETFL, opts) < 0){
```

```
int main()
{
    int i, maxi, listenfd, connfd, sockfd, epfd, nfds;
    ssize_t n;
    char line[MAXLINE];
    socklen_t clilen;
    //声明epoll_event结构体的变量, ev用于注册事件, events数组用于回传要处理的事
    struct epoll_event ev,events[20];
    //生成用于处理accept的epoll专用的文件描述符, 指定生成描述符的最大范围为256
    epfd = epoll_create(256);
    struct sockaddr_in clientaddr;
    struct sockaddr_in serveraddr;
    listenfd = socket(AF_INET, SOCK_STREAM, 0);

    setnonblocking(listenfd);        //把用于监听的socket设置为非阻塞方式
    ev.data.fd = listenfd;           //设置与要处理的事件相关的文件描述符
    ev.events = EPOLLIN | EPOLLET;  //设置要处理的事件类型
    epoll_ctl(epfd, EPOLL_CTL_ADD, listenfd, &ev);     //注册epoll事件
    bzero(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    char *local_addr = "200.200.200.204";
    inet_aton(local_addr, &(serveraddr.sin_addr));
    serveraddr.sin_port = htons(SERV_PORT);  //或者htons(SERV_PORT);
    bind(listenfd,(sockaddr *)&serveraddr, sizeof(serveraddr));
    listen(listenfd, LISTENQ);
```

```
maxi = 0;
  for( ; ; ){
     nfds = epoll_wait(epfd, events, 20, 500); //等待epoll事件的发生
     for(i = 0; i < nfds; ++i){                  //处理所发生的所有事件
        if(events[i].data.fd == listenfd){       //监听事件
           connfd = accept(listenfd, (sockaddr *)&clientaddr, &clilen);
           if(connfd < 0){
              perror("connfd<0");
              exit(1);
           }
           setnonblocking(connfd);           //把客户端的socket设置为非阻塞方式
           char *str = inet_ntoa(clientaddr.sin_addr);
           std::cout << "connect from " << str  <<std::endl;
           ev.data.fd=connfd;                //设置用于读操作的文件描述符
           ev.events=EPOLLIN | EPOLLET;      //设置用于注测的读操作事件
           epoll_ctl(epfd, EPOLL_CTL_ADD, connfd, &ev);
           //注册ev事件
        }
        else if(events[i].events&EPOLLIN) {     //读事件
           if ( (sockfd = events[i].data.fd) < 0){
              continue;
           }
           if ( (n = read(sockfd, line, MAXLINE)) < 0){
              if (errno == ECONNRESET){
                 close(sockfd);
                 events[i].data.fd = -1;
              }
              else{
                 std::cout<<"readline error"<<std::endl;
              }
              //这里解析line中的http请求，然后发回文件
           }
           else if (n == 0){
              close(sockfd);
              events[i].data.fd = -1;
           }
           ev.data.fd=sockfd;              //设置用于写操作的文件描述符
           ev.events=EPOLLOUT | EPOLLET;   //设置用于注测的写操作事件
           //修改sockfd上要处理的事件为EPOLLOUT
           epoll_ctl(epfd, EPOLL_CTL_MOD, sockfd, &ev);
        }
        else if(events[i].events&EPOLLOUT){//写事件
           sockfd = events[i].data.fd;
           write(sockfd, line, n);
           ev.data.fd = sockfd;                //设置用于读操作的文件描述符
           ev.events = EPOLLIN | EPOLLET;      //设置用于注册的读操作事件
           //修改sockfd上要处理的事件为EPOLIN
           epoll_ctl(epfd, EPOLL_CTL_MOD, sockfd, &ev);
        }
     }
  }
}
```

```
#define MAX_EVENTS 500
struct myevent_s
{
    int fd;
    void (*call_back)(int fd, int events, void *arg);
    int events;
    void *arg;
    int status; // 1: in epoll wait list, 0 not in
    char buff[128]; // recv data buffer
    int len;
    long last_active; // last active time
};
// set event
void EventSet(myevent_s *ev, int fd, void (*call_back)(int, int,
void*), void *arg)
{
    ev->fd = fd;
    ev->call_back = call_back;
    ev->events = 0;
    ev->arg = arg;
    ev->status = 0;
    ev->last_active = time(NULL);
}
// add/mod an event to epoll
void EventAdd(int epollFd, int events, myevent_s *ev)
{
    struct epoll_event epv = {0, {0}};
    int op;
    epv.data.ptr = ev;
    epv.events = ev->events = events;
    if(ev->status == 1){
        op = EPOLL_CTL_MOD;
    }
    else{
        op = EPOLL_CTL_ADD;
        ev->status = 1;
    }
    if(epoll_ctl(epollFd, op, ev->fd, &epv) < 0)
        printf("Event Add failed[fd=%d]/n", ev->fd);
    else
        printf("Event Add OK[fd=%d]/n", ev->fd);
}       perror("fcntl(sock,SETFL,opts)");
      exit(1);
   }
}
void EventDel(int epollFd, myevent_s *ev)
{
    struct epoll_event epv = {0, {0}};
    if(ev->status != 1) return;
    epv.data.ptr = ev;
    ev->status = 0;
    epoll_ctl(epollFd, EPOLL_CTL_DEL, ev->fd, &epv);
}
int g_epollFd;
myevent_s g_Events[MAX_EVENTS+1];
```

```
void AcceptConn(int fd, int events, void *arg)
{
    struct sockaddr_in sin;
    socklen_t len = sizeof(struct sockaddr_in);
    int nfd, i;
    // accept
    if((nfd = accept(fd, (struct sockaddr*)&sin, &len)) == -1)
    {
        if(errno != EAGAIN && errno != EINTR)
        {
            printf("%s: bad accept", __func__);
        }
        return;
    }
    do
    {
        for(i = 0; i < MAX_EVENTS; i++)
        {
            if(g_Events[i].status == 0)
            {
                break;
            }
        }
        if(i == MAX_EVENTS)
        {
            printf("%s:max connection limit[%d].", __func__, MAX_EVENTS);
            break;
        }
        // set nonblocking
        if(fcntl(nfd, F_SETFL, O_NONBLOCK) < 0) break;
        // add a read event for receive data
        EventSet(&g_Events[i], nfd, RecvData, &g_Events[i]);
        EventAdd(g_epollFd, EPOLLIN|EPOLLET, &g_Events[i]);
        printf("new conn[%s:%d][time:%d]/n", inet_ntoa(sin.sin_addr),
ntohs(sin.sin_port), g_Events[i].last_active);
    }while(0);
}
void InitListenSocket(int epollFd, short port)
{
    int listenFd = socket(AF_INET, SOCK_STREAM, 0);
    fcntl(listenFd, F_SETFL, O_NONBLOCK); // set non-blocking
    printf("server listen fd=%d/n", listenFd);
    EventSet(&g_Events[MAX_EVENTS], listenFd, AcceptConn,
&g_Events[MAX_EVENTS]);
    // add listen socket
    EventAdd(epollFd, EPOLLIN|EPOLLET, &g_Events[MAX_EVENTS]);
    // bind & listen
    sockaddr_in sin;
    bzero(&sin, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;
    sin.sin_port = htons(port);
    bind(listenFd, (const sockaddr*)&sin, sizeof(sin));
    listen(listenFd, 5);
}
```

```
void RecvData(int fd, int events, void *arg)
{
    struct myevent_s *ev = (struct myevent_s*)arg;
    int len;
    // receive data
    len = recv(fd, ev->buff, sizeof(ev->buff)-1, 0);
    EventDel(g_epollFd, ev);
    if(len > 0)
    {
        ev->len = len;
        ev->buff[len] = '/0';
        printf("C[%d]:%s/n", fd, ev->buff);
        // change to send event
        EventSet(ev, fd, SendData, ev);
        EventAdd(g_epollFd, EPOLLOUT|EPOLLET, ev);
    }
    else if(len == 0)
    {
        close(ev->fd);
        printf("[fd=%d] closed gracefully./n", fd);
    }
    else
    {
        close(ev->fd);
        printf("recv[fd=%d] error[%d]:%s/n", fd, errno, strerror(errno));
    }
}
void SendData(int fd, int events, void *arg)
{
    struct myevent_s *ev = (struct myevent_s*)arg;
    int len;
    // send data
    len = send(fd, ev->buff, ev->len, 0);
    ev->len = 0;
    EventDel(g_epollFd, ev);
    if(len > 0)
    {
        // change to receive event
        EventSet(ev, fd, RecvData, ev);
        EventAdd(g_epollFd, EPOLLIN|EPOLLET, ev);
    }
    else
    {
        close(ev->fd);
        printf("recv[fd=%d] error[%d]/n", fd, errno);
    }
}
```

```
int main(int argc, char **argv)
{
    short port = 12345; // default port
    if(argc == 2){
        port = atoi(argv[1]);
    }
    // create epoll
    g_epollFd = epoll_create(MAX_EVENTS);
    if(g_epollFd <= 0) printf("create epoll failed.%d/n", g_epollFd);
    // create & bind listen socket, and add to epoll, set non-blocking
    InitListenSocket(g_epollFd, port);
    // event loop
    struct epoll_event events[MAX_EVENTS];
    printf("server running:port[%d]/n", port);
    int checkPos = 0;
    while(1){
        long now = time(NULL);
        for(int i = 0; i < 100; i++, checkPos++) // doesn't check listen fd
        {
            if(checkPos == MAX_EVENTS) checkPos = 0; // recycle
            if(g_Events[checkPos].status != 1) continue;
            long duration = now - g_Events[checkPos].last_active;
            if(duration >= 60) // 60s timeout
            {
                close(g_Events[checkPos].fd);
                printf("[fd=%d] timeout[%d--%d]./n", g_Events[checkPos].fd,
g_Events[checkPos].last_active, now);
                EventDel(g_epollFd, &g_Events[checkPos]);
            }
        }
        // wait for events to happen
        int fds = epoll_wait(g_epollFd, events, MAX_EVENTS, 1000);
        if(fds < 0){
            printf("epoll_wait error, exit/n");
            break;
        }
        for(int i = 0; i < fds; i++){
            myevent_s *ev = (struct myevent_s*)events[i].data.ptr;
            if((events[i].events&EPOLLIN)&&(ev->events&EPOLLIN)) // read
event
            {
                ev->call_back(ev->fd, events[i].events, ev->arg);
            }
            if((events[i].events&EPOLLOUT)&&(ev->events&EPOLLOUT)) // write
event
            {
                ev->call_back(ev->fd, events[i].events, ev->arg);
            }
        }
    }
    // free resource
    return 0;
}
```

Events vs threads

Both used in high performance programming
Both excellent for high performance packet actions

- Spawn threads for tasks (read, write, process)
- Any thread can wait until it has input, overall program still moves
- Threads are difficult to debug
- Threads can deadlock against each other
- Not all functions are thread safe, clobbering data
- Onus is on you to choreograph a careful dance, easy to mess up

- Main thread of execution loops over possible actions
- Actions include: read, write, signal, alarm
- Every possible action has an associated “callback”
- Callbacks process data
- Easy to debug, look at active event handler
- Deadlocks don’t happen, data not clobbered by stray thread
- Program is always doing something, or looking for something to do

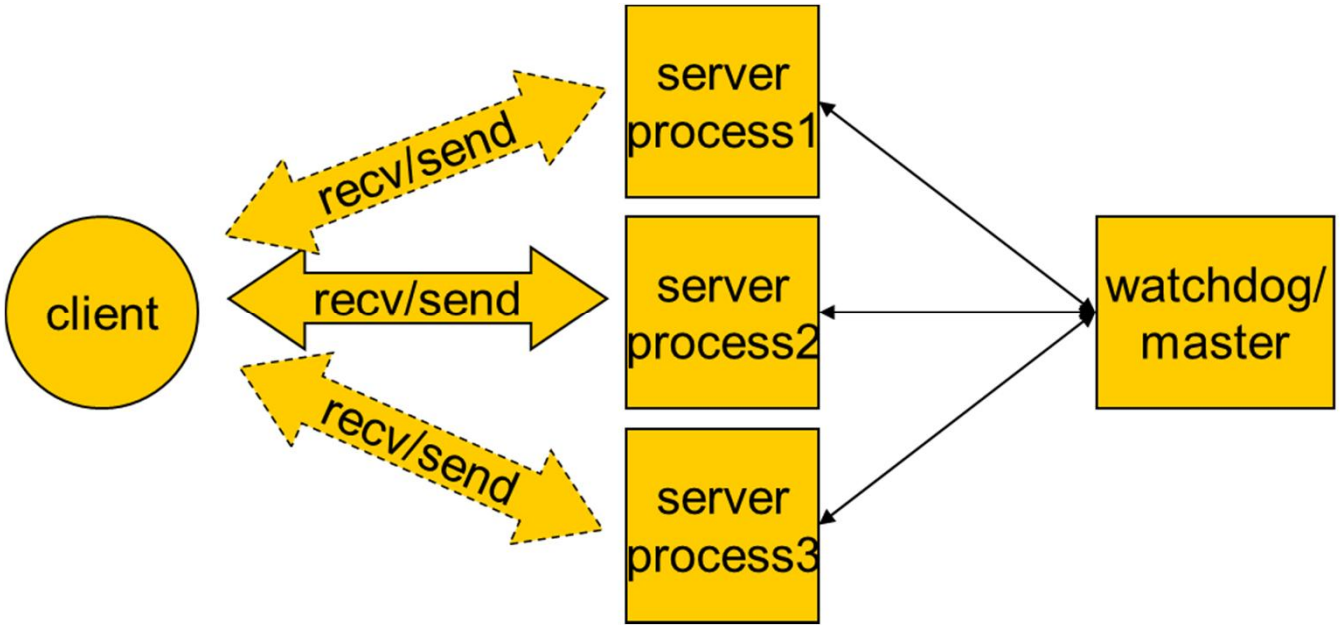
多进程/线程 vs 事件驱动

- 轮询 VS 中断？
  - 高速网卡中的NAPI
- Tradeoff
  - IO密集型
    - 互联网应用
    - 事件驱动更合适
  - CPU密集型
    - Apache称为应用服务器？
    - 具体的业务应用，如业务逻辑、图形图像、数据库读写
    - 一个计算耗时2秒，那么这2秒就是完全阻塞
    - 事件驱动的MySQL？
    - 多进程或线程适合于CPU密集型服务
    - 对于交互式的长连接应用也是常见的选择(比如BBS)，也常用来做内部服务器交互的模型。 这种策略很难满足高性能程序的需求，好处是实现极其简单，容易嵌入复杂的交互逻辑。Apache、ftpd等都是这种工作模式

我们采取什么技术路线？

Web服务器V5--进程/线程池+事件驱动

- 中庸之道
- 例子
  - nginx

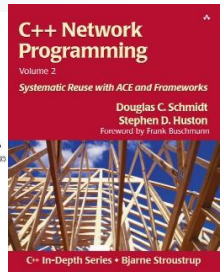


怎么控制client连接的是哪个进程？
进程和线程该怎么选择？

高性能Web服务器HPS

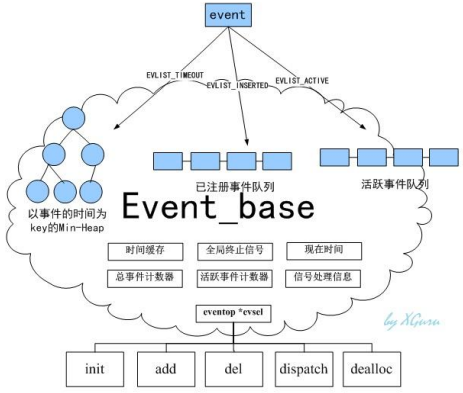
- 目标需求
  - 支持每秒10k连接并发C10K
  - 静态文件读取
  - 支持长连接
  - 实现文件cache
  - 服务器行为log
- 技术路线
  - 多线程+事件驱动方式
  - **复用、复用、复用**
- 实现方法
  - 基于成熟的网络编程框架
  - ACE/Libevent/Boost:Asio
  - 利用现有的源代码
  - 理解其设计模式然后改动部分代码

成熟的网络编程库和框架



Dr. Douglas

- C++
  - ACE（Adaptive Communication Environment）
    - 网络通讯设计模式的集大成者
    - 非常重量级
  - Boost ASIO（ Asynchronous I/O ）
    - www.boost.org
    - Asio 是一个跨平台的C++框架用来处理网络和低级I/O编程
    - 高性能的网络并发设计模式封装
    - 纳入Boost体系，正在申请成为标准网络库
- C
  - Libevent
    - Abstract event framework，IO/Signal/Alarm
    - 没有封装Socket底层操作
  - Apache Portable Runtime Library（APR）



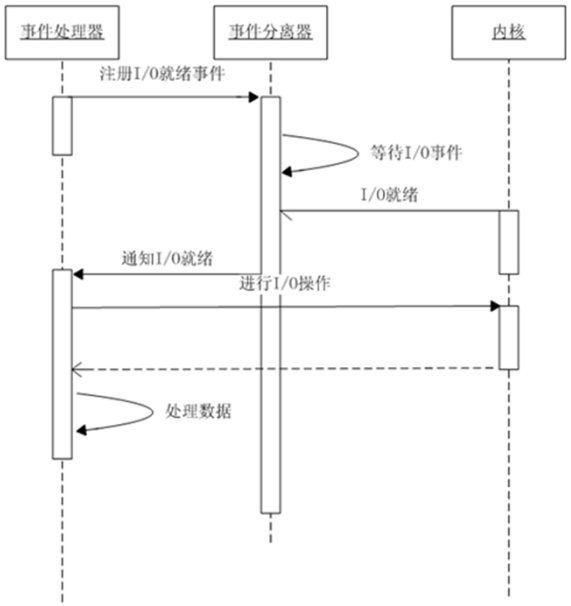
ASIO框架

- 编程框架（Framework）
  - 与库不同（Library）
    - 面向对象的代码组织形式而成的库也叫类库
    - 面向过程的代码组织形式而成的库也叫函数库
    - Glibc，Windows API，MSCRT
  - 框架
    - 为解决一类问题而开发的产品，用户一般只需要使用框架提供的类或函数，即可实现全部功能。
    - 开发者在使用框架的时候，必须使用这个框架的全部代码。
    - MFC、ACE
- ASIO 编程框架
  - 基于状态机的实现高性能服务器编程的最佳模型
  - 利用上面提到的select/epoll/kqueue/iocp等IO机制
  - 屏蔽了繁琐的IO细节
  - 完整的应用程序业务逻辑被强制分拆到很多个handle函数，看起来不够优雅

网络I/O设计模式之Reactor

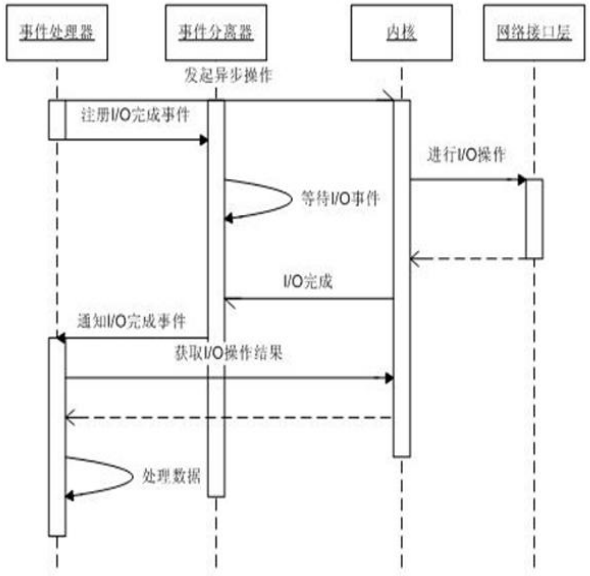
- Reactor
  - 基于同步I/O的
- 工作流程
  - 1.注册读就绪事件和相应的事件处理器；
  - 2.事件分离器等待事件
  - 3.事件到来，激活分离器，分离器调用事件对应的处理器；
  - 4.事件处理器完成实际的读操作，处理读到的数据，注册新的事件，然后返还控制权。

- 某个事件处理者宣称它对某个socket上的读事件很感兴趣；
- 事件分离者等着这个事件的发生；
- 当事件发生了，事件分离器被唤醒，这负责通知先前那个事件处理者；
- 事件处理者收到消息，于是去那个socket上读数据了. 如果需要，它再次宣称对这个socket上的读事件感兴趣，一直重复上面的步骤；



网络I/O设计模式之Proactor

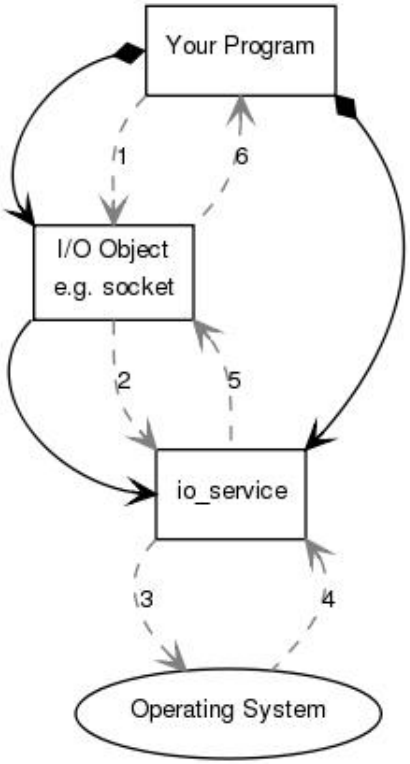
- Proactor
  - 用异步I/O实现
  - 与reactor不同的是，处理器不关心I/O就绪事件，它关注的是完成事件
- Proactor（反应器模式）
  - 1. 处理器发起异步操作，并关注I/O完成事件；
  - 2. 事件分离器等待操作完成事件；
  - 3. 分离器等待过程中，内核并行执行实际的I/O操作，并将结果数据存入用户自定义缓冲区，最后通知事件分离器读操作完成·
  - 4. I/O完成后，通过事件分离器呼唤处理器；
  - 5. 事件处理器处理用户自定义缓冲区中的数据；



- 事件处理者直接投递发一个写操作(当然，操作系统必须支持这个异步操作)．这个时候，事件处理者根本不关心读事件，它只管发这么个请求，它魂牵梦萦的是这个**写操作的完成事件**。这个处理者很拽，发个命令就不管具体的事情了，只等着别人（系统）帮他搞定的时候给他回个话。
- 事件分离者等着这个读事件的完成(比较下与Reactor的不同);
- 当事件分离者默默等待完成事情到来的同时，操作系统已经在一边开始干活了，它从目标读取数据，放入用户提供的缓存区中，最后通知事件分离者，这个事情我搞完了;
- 事件分享者通知之前的事件处理者：你吩咐的事情搞定了;
- 事件处理者这时会发现想要读的数据已经乖乖地放在他提供的缓存区中，想怎么处理都行了。如果有需要，事件处理者还像之前一样发起另外一个写操作，和上面的

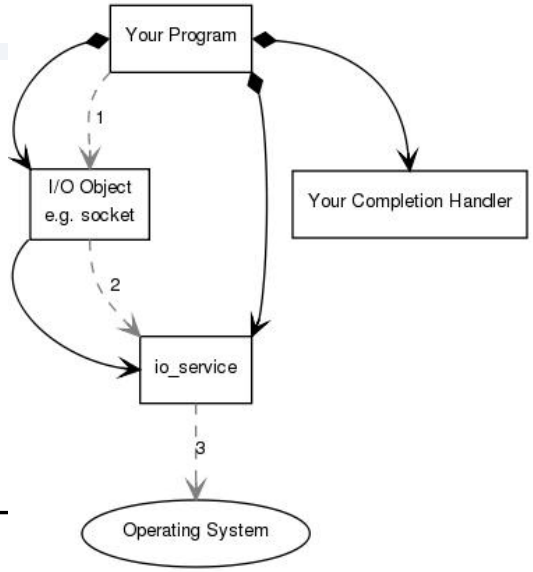
ASIO框架核心概念

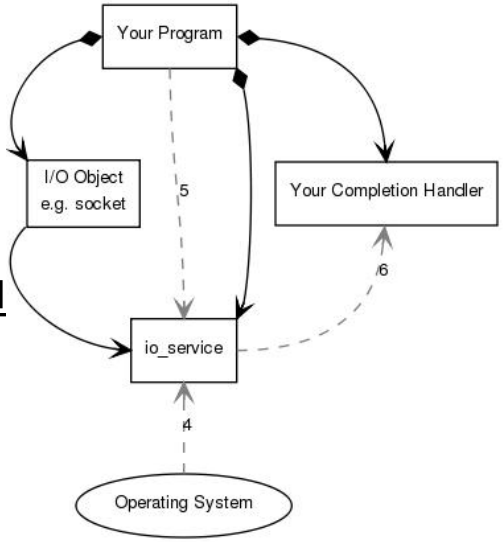
- 同步IO过程
  - Your program会至少有一个io_service对象，表示your program到operating system的I/O服务的纽带
    - asio::io_service io_service;
  - your program需要一个I/O对象如一个TCP socket执行I/O操作 ：
    - asio::ip::tcp::socket socket(io_service);
  - 当一个同步连接操作执行时，事件序列发生如下：
    - 1.Your program通过调用I/O对象初始化连接操作：
      - socket.connect(server_endpoint);
    - 2.I/O对象转发请求给io_service。
    - 3.io_service调用operating system执行连接操作。
    - 4.operating system返回操作的结果给io_service
    - 5.io_service会将操作的任何错误解释成为一个asio::error_code对象。error_code(错误码)可能被与特定值比较，也可能作为一个布尔值被测试(false意味着没有错误发生)。然后结果返回给I/O对象。



ASIO框架核心概念

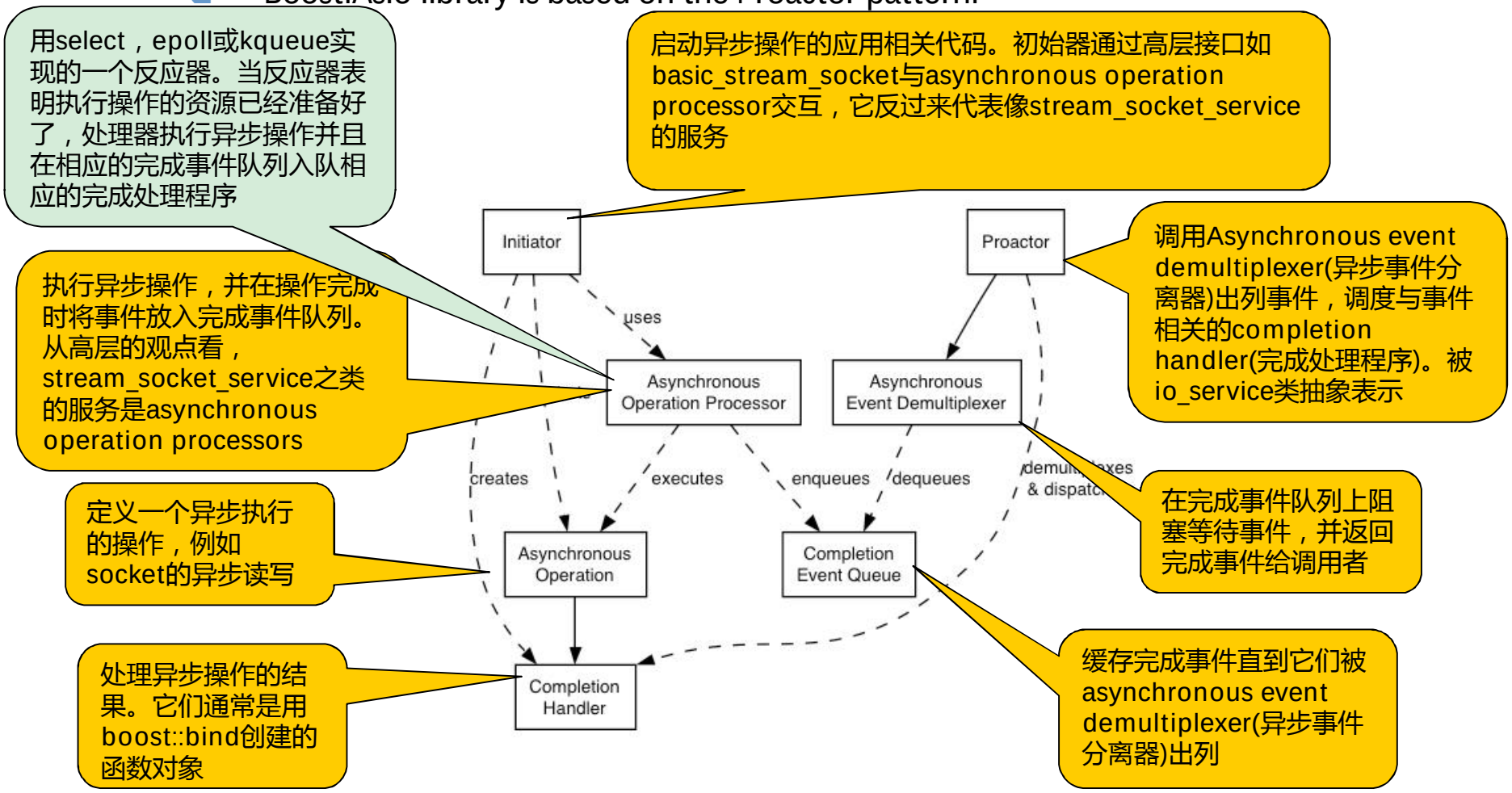
- 异步IO过程
  - 1.Your program通过调用I/O对象初始化连接：
    - socket.async_connect(server_endpoint, your_completion_handler);
  - your_completion_handler是一个函数对象有如下签名
    - void your_completion_handler(const asio::error_code& ec);
  - 2. I/O对象转发请求给io_service
  - 3. io_service给operating system发信号，告诉它应该开始一个异步连接
  - **时间流逝。(在同步的情形中这里会等待整个连接）**
  - 4.operating system把结果放置在准备被io_service对象拾取的队列中表明连接操作已经完成。
  - 5.Your program必须调用io_service::run()(或者一个类似的io_service成员方法)来获取结果。当有未完成的异步操作时，调用io_service::run()会阻塞，所以你通常尽快调用它一旦开始了第一个异步操作。
  - 6.在io_service::run()内部，io_service将运行结果出列，把它翻译为一个error_code，然后把它传给your completion handler。





ASIO基于Proactor设计模式实现

- 模式
  - Boost.Asio library is based on the Proactor pattern.



基于ASIO的Web Server编程

- 目标
  - 静态文件读取
    - 利用asio处理本地静态文件IO
  - 支持长连接，持续保持建立的TCP连接，空闲60秒后关闭
    - 加入定时器
  - 支持文件cache
    - 最近传输过的文件从内存直接读取，不需要再经过文件io，提高性能
- 代码框架
  - http://www.boost.org/doc/libs/1_47_0/doc/html/boost_asio/examples.html

Timers

Examples showing how to customise deadline_timer using different time types.

- boost_asio/example/timers/tick_count_timer.cpp
- boost_asio/example/timers/time_t_timer.cpp

HTTP Server 3

An HTTP server using a single io_service and a thread pool calling `io_service::run()`

- boost_asio/example/http/server3/connection.cpp
- boost_asio/example/http/server3/connection.hpp
- boost_asio/example/http/server3/header.hpp
- boost_asio/example/http/server3/main.cpp
- boost_asio/example/http/server3/mime_types.cpp
- boost_asio/example/http/server3/mime_types.hpp
- boost_asio/example/http/server3/reply.cpp
- boost_asio/example/http/server3/reply.hpp
- boost_asio/example/http/server3/request.hpp
- boost_asio/example/http/server3/request_handler.cpp
- boost_asio/example/http/server3/request_handler.hpp
- boost_asio/example/http/server3/request_parser.cpp
- boost_asio/example/http/server3/request_parser.hpp
- boost_asio/example/http/server3/server.cpp
- boost_asio/example/http/server3/server.hpp

修改文件

boost_asio/example/http/server3/file_cache.hpp
boost_asio/example/http/server3/file_cache.cpp
boost_asio/example/http/server3/log.hpp
boost_asio/example/http/server3/log.cpp

增加文件，参考开源memcache实现

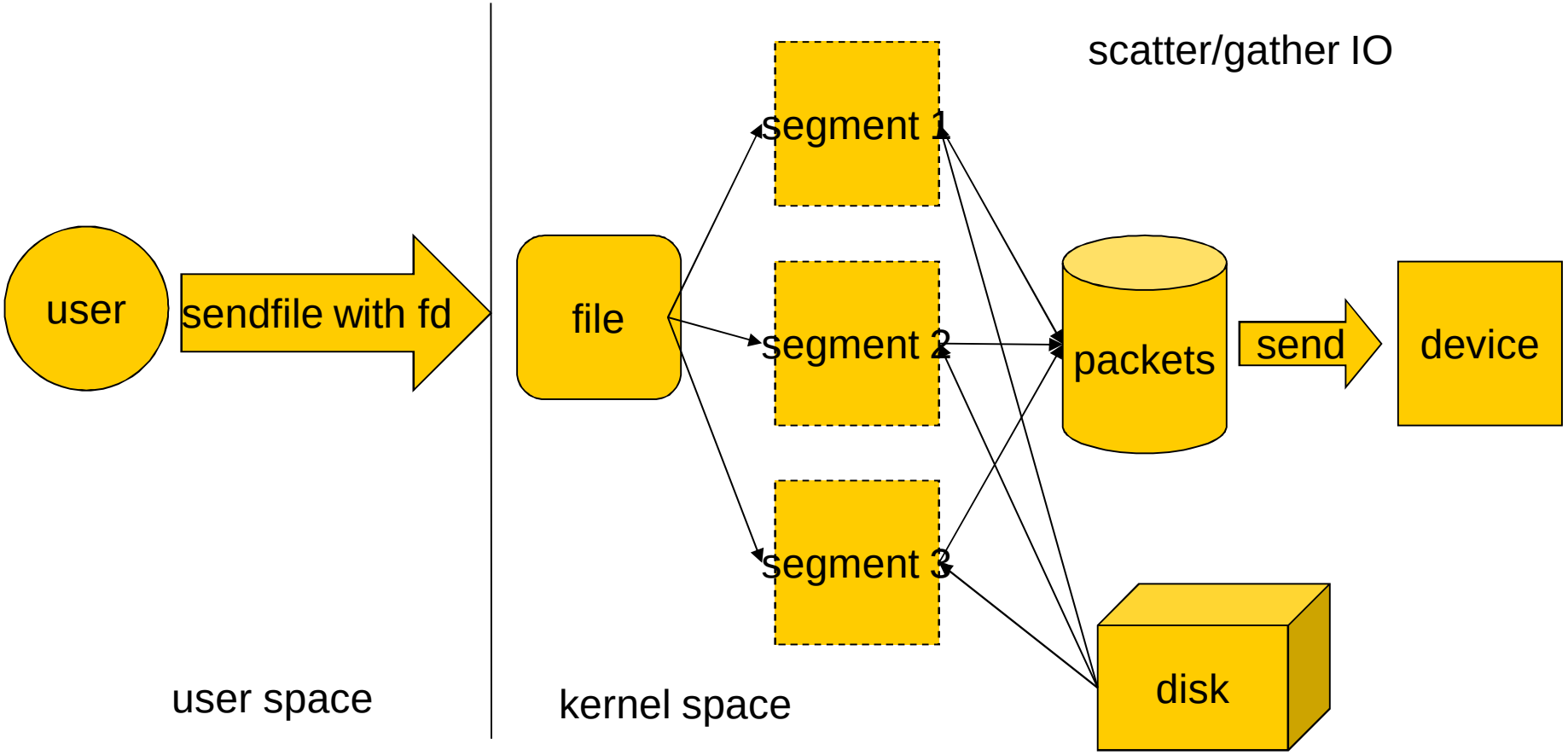
编程说明

- 错误处理
  - 处理信号、IO出错、异常、可恢复
- 调试工具
  - 右表
- 性能杀手
  - 数据拷贝
  - 上下文切换
  - 内存分配
  - 锁竞争
- 日志信息
  - /proc/PID/：查看进程的相关信息
  - /var/log/messages：查看系统及内核日志
  - /proc/net/sockstat
  - /proc/net/dev
  - /proc/net/snmp
- 使用成熟的库
  - Boost里的大量代码关于时间、字符串、定时器等

- strace：查看进程的系统调用
- ltrace：查看进程的库函数调用
- lsof：查看系统已打开的文件句柄
- netstat：查看网络信息
- tcpdump：抓包工具
- top：查看CPU和Load
- uptime：查看系统运行时间和Load
- iostat：磁盘IO统计
- ulimit：查看进程的系统资源限制
- free：查看内存和交换分区
- mpstat：查看中断分布情况
- gprof：性能分析工具

文件传输优化

- sendfile&zero-copy原理简介



实验说明

- 运行要求

./hps <address> <port> <threads> <doc_root> <log_fullname>
第一个参数：IP地址
第二个参数：web服务端口
第三个参数：线程池大小
第四个参数：网站根目录
第五个参数：日志文件名绝对路径，日志格式：

只记录文件请求和发送事件，一个事件一行
2012-10-30 17:38:18.458 Client [%ip:%port] request file %filename
2012-10-30 17:38:18.458 Successful send file %filename with %filesize bytes to client [%ip:%port] after %delay ms from receiving its request.
2012-10-30 17:38:18.458 Fail to send file %filename to client [%ip:%port] due to error %error.

代码与文档格式要求

- 提交格式
  - Code目录
    - connection.cpp connection.hpp header.hpp main.cpp mime_types.cpp mime_types.hpp reply.cpp reply.hpp request.hpp request_handler.cpp request_handler.hpp request_parser.cpp request_parser.hpp server.cpp server.hpp file_cache.cpp file_cache.hpp log.hpp log.cpp（文件可以增加或修改）
    - Makefile编译规则文件，可执行文件名hps
  - Readme
    - 说明程序的编译和使用方法
  - 实验XXXXX设计文档.doc

参考文档和书籍

- 技术文档
  - C10K问题：http://www.kegel.com/c10k.html
- 《C++网络编程》
- 《UNIX网络编程》
- 《内核源代码情景分析》


THE END