elasticsearch.

Shay Banon, founder of Elasticsearch

一辈子很长，就找个找个有趣的人在一起。（王小波）

计算机应用编程 实验二

并行网络爬虫

熊永平@网络技术研究院

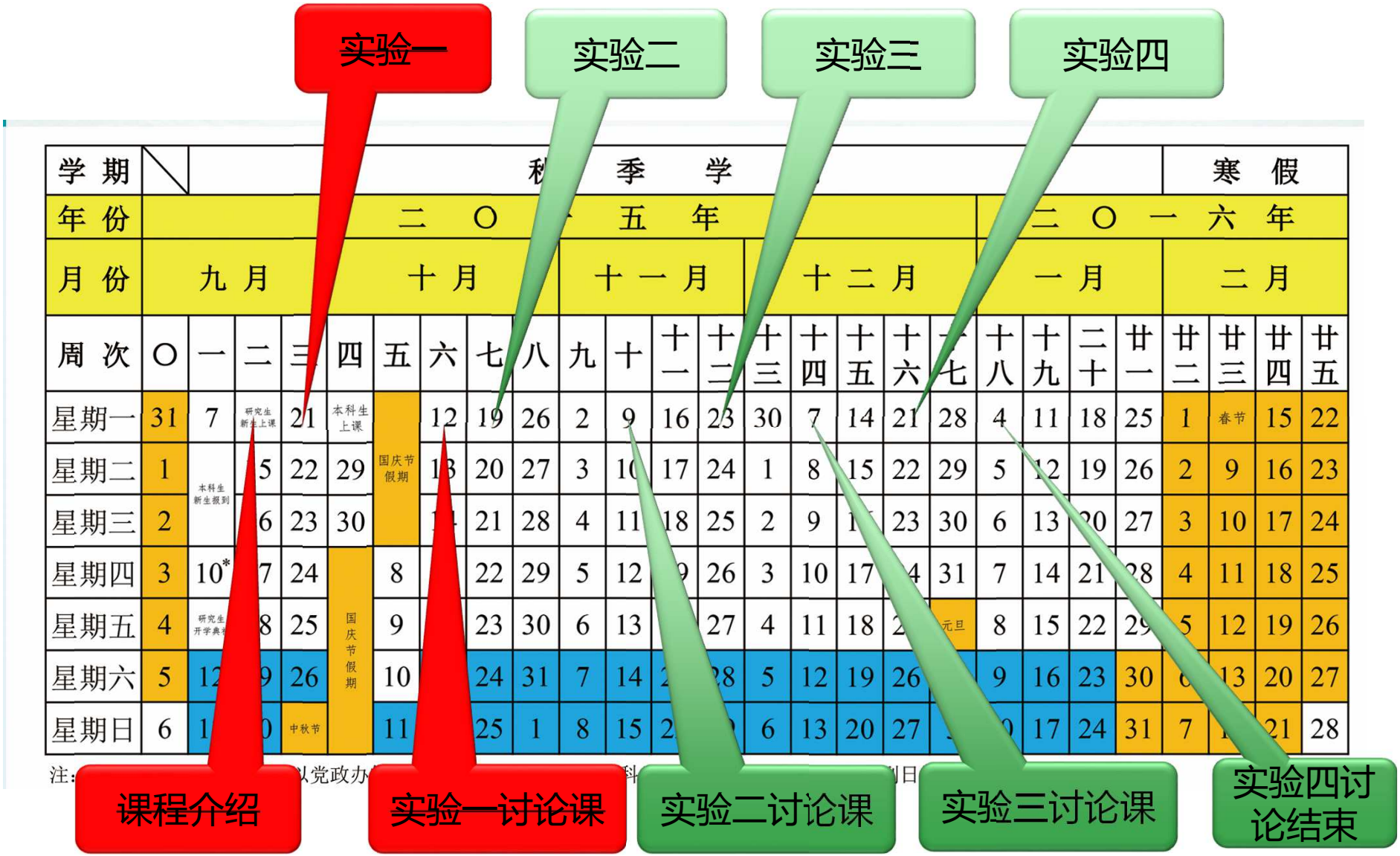
ypxiong@bupt.edu.cn

周一10:10-12:00@3-134

2015.10.19

告诉我，你来了

课程表



学期		秋季学																			寒假			
年份	二〇					五年												二〇一六年						
月份	九月				十月				十一月				十二月					一月			二月			
周次	〇	一	二	三	四	五	六	七	八	九	十	十一	十二	十三	十四	十五	十六	十七	十八	十九	二十	廿一	廿二	廿三	廿四	廿五
星期一	31	7	研究生新生上课	21	本科生上课	国庆节假期	12	19	26	2	9	16	23	30	7	14	21	28	4	11	18	25	1	春节	15	22
星期二	1	本科生新生报到		22	29		13	20	27	3	10	17	24	1	8	15	22	29	5	12	19	26	2	9	16	23
星期三	2		6	23	30			21	28	4	11	18	25	2	9		23	30	6	13	20	27	3	10	17	24
星期四	3	10*	7	24		8		22	29	5	12		26	3	10	17		31	7	14	21	28	4	11	18	25
星期五	4	研究生开学典	8	25	国庆节假期	9		23	30	6	13		27	4	11	18		元旦	8	15	22	29	5	12	19	26
星期六	5	12	9	26		10		24	31	7	14		28	5	12	19	26		9	16	23	30	6	13	20	27
星期日	6	1	0	中秋节		11		25	1	8	15			6	13	20	27		0	17	24	31	7		21	28

注：

实验二：并行网络爬虫


1、实验任务
2、相关基础原理
3、爬虫设计与实现
4、实验说明

实验目标

- 目标
  - 设计一个分布式网络爬虫实现
    - 爬取160000个页面的网站
    - 支持分布式抓取
    - 高性能网络爬虫
- 编程技能
  - C语言练习
  - 多线程与线程池
  - 网络编程与HTTP协议
  - 网页解析
  - 异步网络IO
  - 消息队列
  - Web服务器配置与运行


1、实验任务
2、相关基础原理
3、爬虫设计与实现
4、实验说明

相关基础原理

- WWW与Web
- DNS域名机制
- HTML语言
- 字符集编码
- HTTP协议
- Web服务器
- Socket网络编程
- 网络IO复用
- 多线程
- 消息队列

什么是Web（World Wide Web）

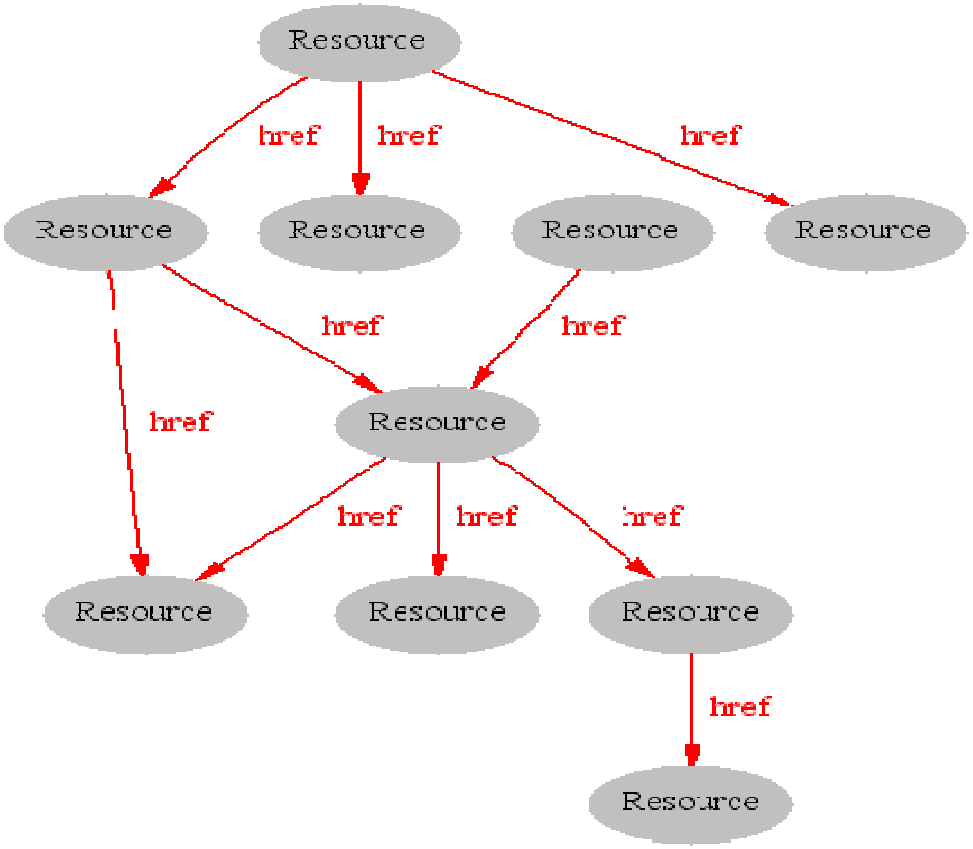
"Web是一个抽象的（假想的）信息空间

The world wide web (web) is a network of information resources.

---- Tim Berners-Lee

Web是...

```
<!-- HEADER -->
<div id="header">

    <h1><a href="index.html" onclick="return link(th

    <a class="header-button toggle">Menu</a>

    <!-- NAVIGATION -->
    <div id="nav">

        <ul>
            <li><a href="index.html" onclick="return
            <li><a href="about.html" onclick="return
            <li><a href="portfolio.html" onclick="re
            <li><a href="gallery.html" onclick="retu
            <li><a href="contact.html" onclick="retu
            <li><a href="styles.html" onclick="retur
        </ul>

    </div> <!-- /nav -->
```

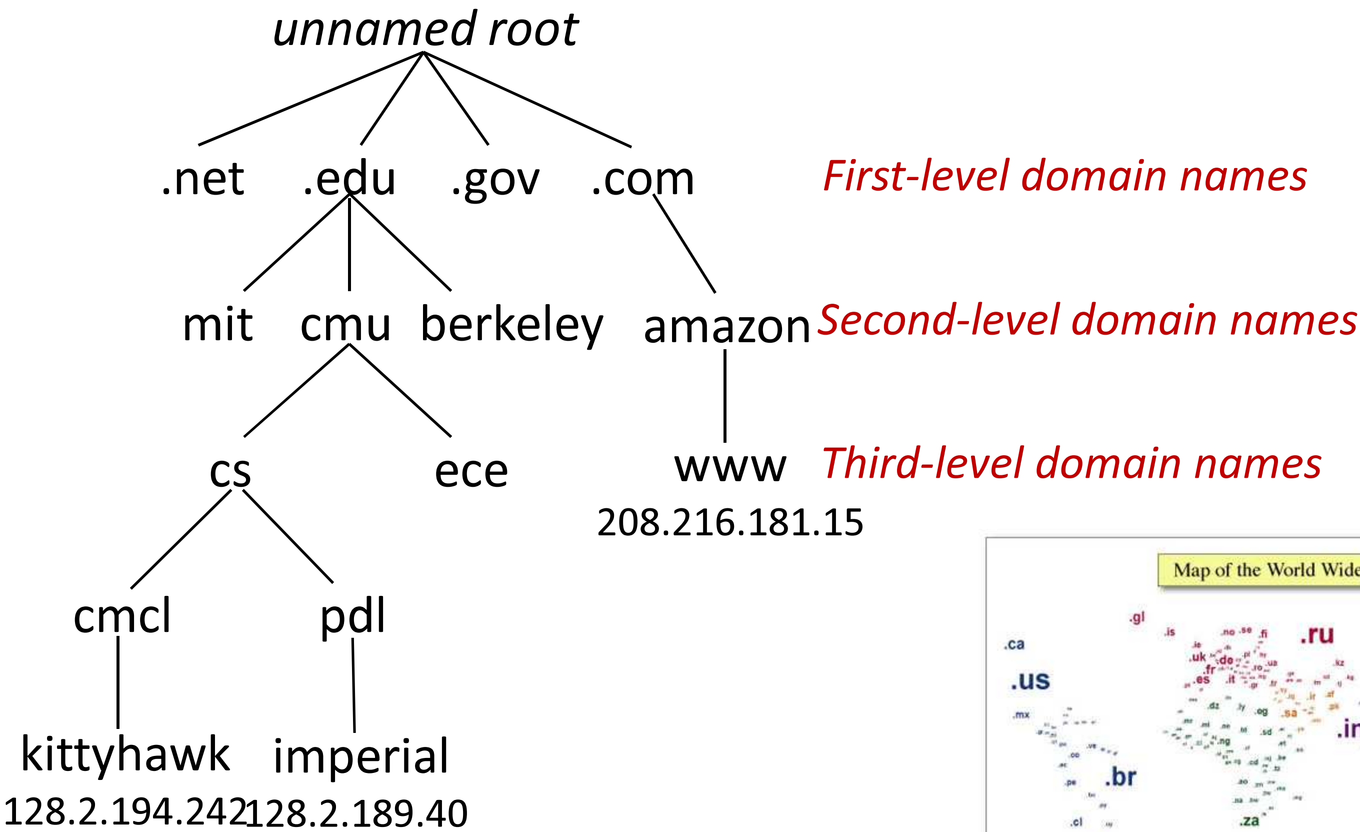


URI

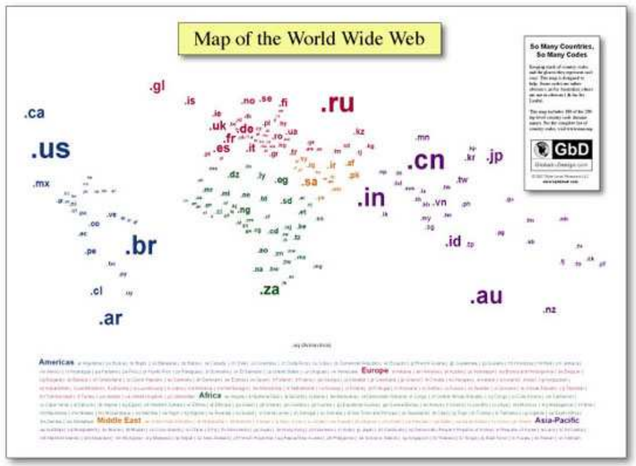
- Web上每个可用资源都有一个由URI编码的地址
- URIs由3部分构成
  - 访问资源的方法：例如HTTP, FTP
  - 存放resource的主机名
  - Resource在主机上的路径
- *Example*
  - *http://www.w3.org/TR*
  - There is a document available via the HTTP protocol
  - Residing on the machines hosting www.w3.org
  - Accessible via the path "/TR"

主机域名机制

➢ 树形结构

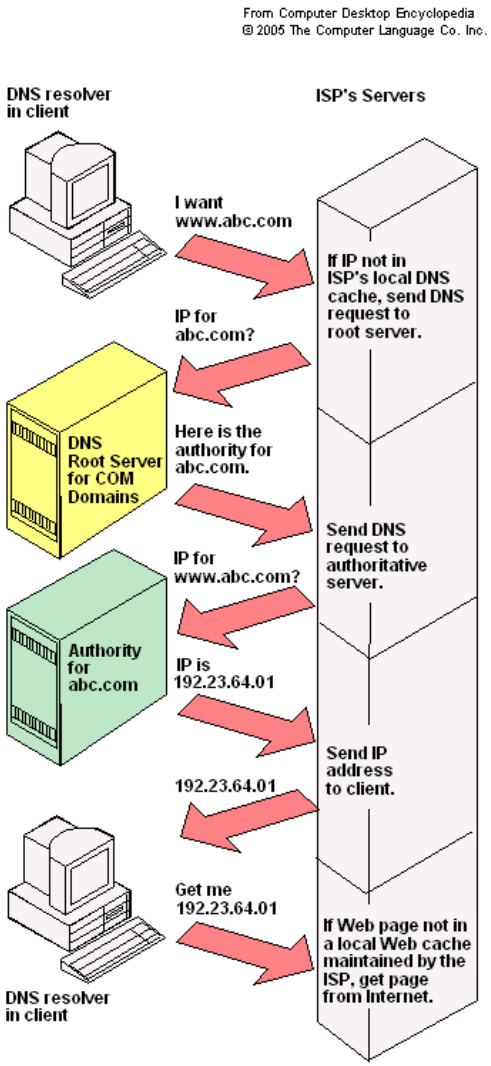


unnamed root
.net
.edu
.gov
.com
First-level domain names
mit
cmu
berkeley
amazon
Second-level domain names
cs
ece
www
Third-level domain names
208.216.181.15
cmcl
pdl
kittyhawk
imperial
128.2.194.242
128.2.189.40



Map of the World Wide Web
.ca
.us
.br
.ar
.ru
.cn
.jp
.in
.au
.za
GbD

DNS 解析过程



- 解析特点
  - DNS解析请求是同步调用的!
  - DNS解析可能会耗费很长时间
  - 主机名可能无法解析
- 问题
  - DNS地址解析性能瓶颈！
- 如何优化性能？
  - 并发DNS client
  - 缓存cache dns results
  - 预取prefetch client

HTML语言

- 超文本文件——ASCII格式
- 例子

```
<html>
        <head>
            <title>网上手机市场</ title>
        </head>
          <body>
          <p>你想要买手机吗?</P>
          <p>你可以访问
              <a href=“http://www.shouji.com”>手机网站</a>了解有关信息。
         </body>
    </html>
```

真实的例子…

HTML文件是什么样子？

字符集编码1

- ASCII
  - ASCII码是7位编码，编码范围是0x00-0x7F。
  - ASCII字符集包括英文字母、阿拉伯数字和标点符号等字符。其中0x00-0x20和0x7F共33个控制字符。
  - 只支持ASCII码的系统会忽略每个字节的最高位，只认为低7位是有效位。
  - 为了传输中文邮件必须使用BASE64或者其他编码方式。
- GB2312（EUC-CN）
  - GB2312基于区位码设计，区位码把编码表分为94个区，每个区对应94个位，每个字符的区号和位号组合起来就是该汉字的区位码。如1601就表示16区1位，对应的字符是“啊”。在区位码的区号和位号上分别加上0xA0就得到了GB2312编码。
  - GB2312字符集中除常用简体汉字字符外还包括希腊字母、日文平假名及片假名字母、俄语西里尔字母等字符。
  - GB2312的编码范围是0xA1A1-0x7E7E，去掉未定义的区域之后可以理解为实际编码范围是0xA1A1-0xF7FE。

字符集编码2

- GBK
  - GBK编码是GB2312编码的超集，向下完全兼容GB2312，同时GBK收录了Unicode基本多文种平面中的所有CJK汉字。
  - GBK也支持希腊字母、日文假名字母、俄语字母等字符，但不支持韩语中的表音字符（非汉字字符）。GBK还收录了GB2312不包含的汉字部首符号、竖排标点符号等字符。
  - GBK的整体编码范围是为0x8140-0xFEFE，不包括低字节是0×7F的组合。高字节范围是0×81-0xFE，低字节范围是0x40-7E和0x80-0xFE。
- GB18030
  - GB18030编码向下兼容GBK和GB2312，兼容的含义是不仅字符兼容，而且相同字符的编码也相同。
  - GB18030收录了所有Unicode3.1中的字符，包括中国少数民族字符，GBK不支持的韩文字符等等，也可以说是世界大多民族的文字符号都被收录在内。

字符集编码3

- UNICODE
  - 每一种语言的不同的编码页，增加了那些需要支持不同语言的软件的复杂度。
  - unicode为每个字符提供了唯一的特定数值，世界上使用的所有字符都列出来，并给每一个字符一个唯一特定数值。
  - Unicode的最初目标，是用1个16位的编码来为超过65000字符提供映射。但这还不够，它不能覆盖全部历史上的文字，也不能解决传输的问题，尤其在那些基于网络的应用中。
  - 因此，Unicode用一些基本的保留字符制定了三套编码方式。它们分别是UTF-8,UTF-16和UTF-32。正如名字所示，在UTF-8中，字符是以8位序列来编码的，用一个或几个字节来表示一个字符。这种方式的最大好处，是UTF-8保留了ASCII字符的编码做为它的一部分，例如，在UTF-8和ASCII中，“A”的编码都是0x41.
  - UTF-16和UTF-32分别是Unicode的16位和32位编码方式。考虑到最初的目的，通常说的Unicode就是指UTF-16。

字符集编码4

- UTF-8
  - Unicode Transformation Format-8bit，是用以解决国际上字符的一种多字节编码，它对英文使用8位（即一个字节），中文使用24为（三个字节）来编码。
  - UTF-8包含全世界所有国家需要用到的字符，是国际编码，通用性强。UTF-8编码的文字可以在各国支持UTF8字符集的浏览器上显示。
  - 如果是UTF8编码，则在外国人的英文IE上也能显示中文，他们无需下载IE的中文语言支持包
- 与GBK比较
  - GBK的文字编码是用双字节来表示的，即不论中英文字符均使用双字节来表示，为了区分中文，将其最高位都设定成1。
  - GBK包含全部中文字符，是国家编码，通用性比UTF8差，不过UTF8占用的数据比GBK大。

字符集编码总结

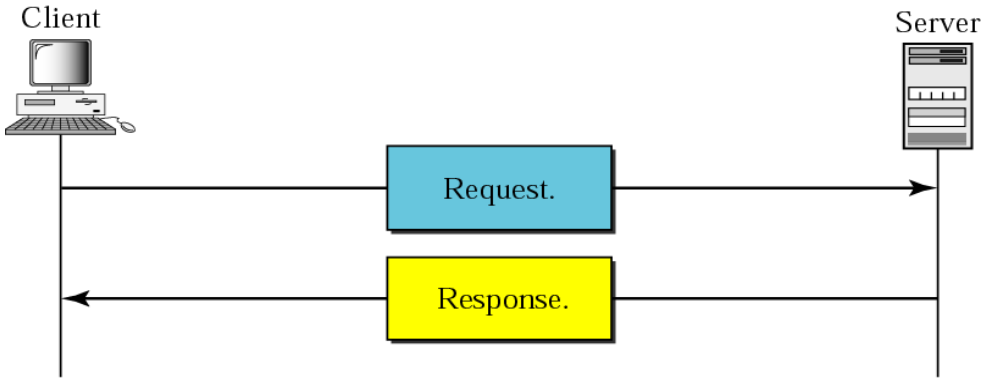
- 总结
  - GBK、GB2312等与UTF8之间都必须通过Unicode编码才能相互转换：

    GBK、GB2312—Unicode--UTF8

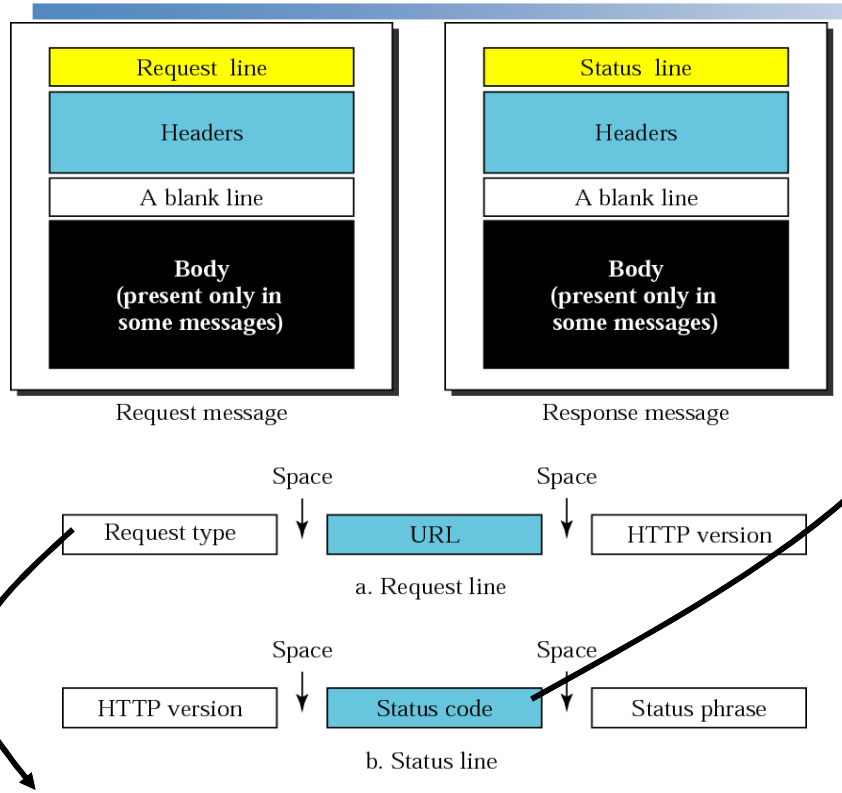
    UTF8--Unicode--GBK、GB2312

HTTP协议

- HTTP layered over bidirectional byte stream
  - TCP port 80
  - **RFC 1945**
- 功能
  - 比之前的FTP简单有效
  - 客户可以从Web服务器上下载几乎所有类型的文件，包括HTML文件，图像/视频/音频等多媒体文件，Java Applet等对象，甚至应用程序等。
- C/S模式



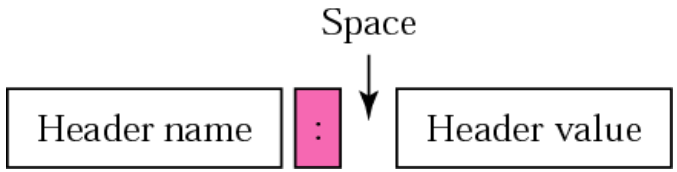
HTTP请求和响应



Method	*Action*
GET	Requests a document from the server
HEAD	Requests information about a document but not the document itself
POST	Sends some information from the client to the server
PUT	Sends a document from the server to the client
TRACE	Echoes the incoming request
CONNECT	Reserved
OPTION	Enquires about available options

Code	*Phrase*	*Description*
	Informational	
100	Continue	The initial part of the request has been received and the client may continue with its request.
101	Switching	The server is complying with a client request to switch protocols defined in the upgrade header.
	Success	
200	OK	The request is successful.
201	Created	A new URL is created.
202	Accepted	The request is accepted, but it is not immediately acted upon.
204	No content	There is no content in the body.
	Redirection	
301	Multiple choices	The requested URL refers to more than one resource.
302	Moved permanently	The requested URL is no longer used by the server.
304	Moved temporarily	The requested URL has moved temporarily.
	Client Error	
400	Bad request	There is a syntax error in the request.
401	Unauthorized	The request lacks proper authorization.
403	Forbidden	Service is denied.
404	Not found	The document is not found.
405	Method not allowed	The method is not supported in this URL.
406	Not acceptable	The format requested is not acceptable.
	Server Error	
500	Internal server error	There is an error, such as a crash, at the server site.
501	Not implemented	The action requested cannot be performed.
503	Service unavailable	The service is temporarily unavailable, but may be requested in the future.

HTTP Header格式



Request headers

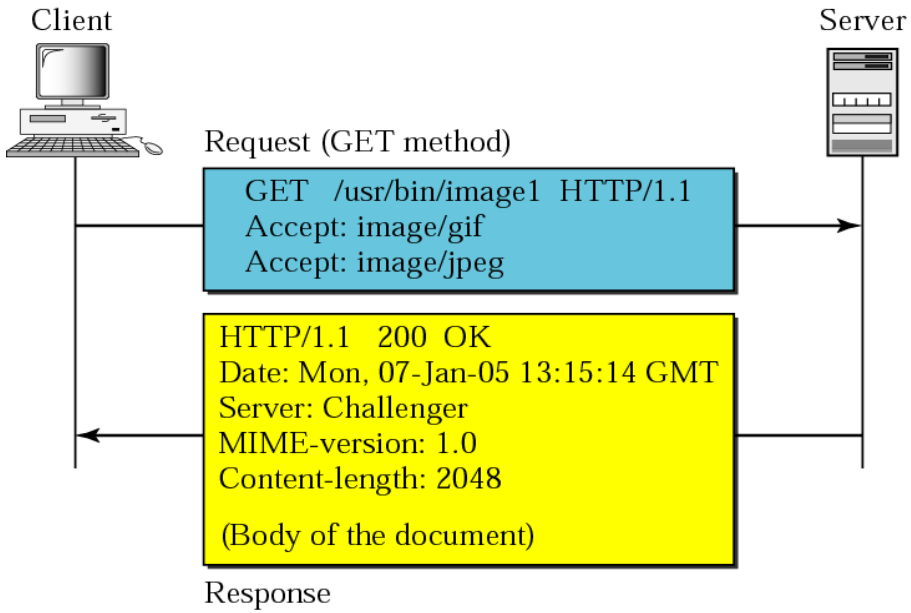
Header	*Description*
Accept	Shows the media format the client can accept
Accept-charset	Shows the character set the client can handle
Accept-encoding	Shows the encoding scheme the client can handle
Accept-language	Shows the language the client can accept
Authorization	Shows what permissions the client has
From	Shows the e-mail address of the user
Host	Shows the host and port number of the client
If-modified-since	Send the document if newer than specified date
If-match	Send the document only if it matches given tag
If-non-match	Send the document only if it does not match given tag
If-range	Send only the portion of the document that is missing
If-unmodified-since	Send the document if not changed since specified date
Referrer	Specifies the URL of the linked document
User-agent	Identifies the client program

Response headers

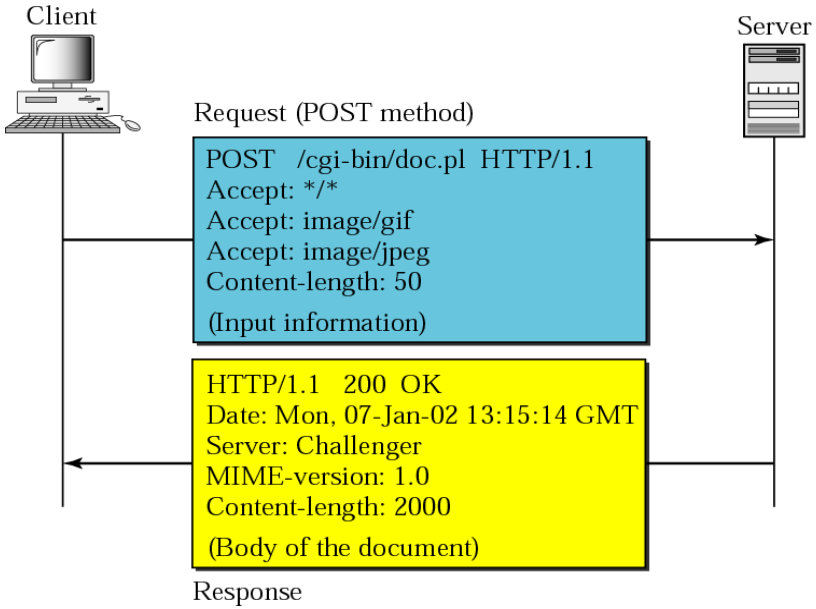
Header	*Description*
Accept-range	Shows if server accepts the range requested by client
Age	Shows the age of the document
Public	Shows the supported list of methods
Retry-after	Specifies the date after which the server is available
Server	Shows the server name and version number

HTTP Example

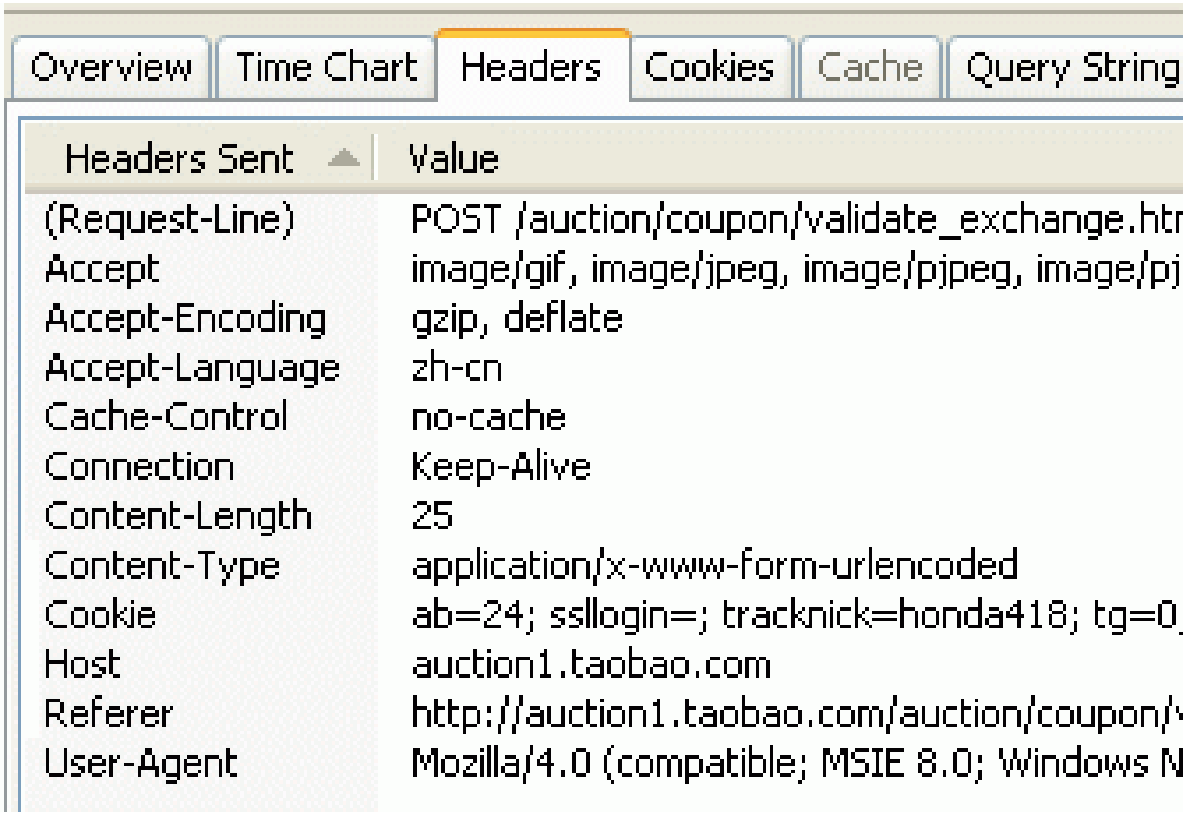
使用GET方法获取图像/usr/bin/image1，客户可以接收GIF 或JPEG格式。



客户机使用POST方法发送数据给服务器，服务器使用perl脚本处理数据。



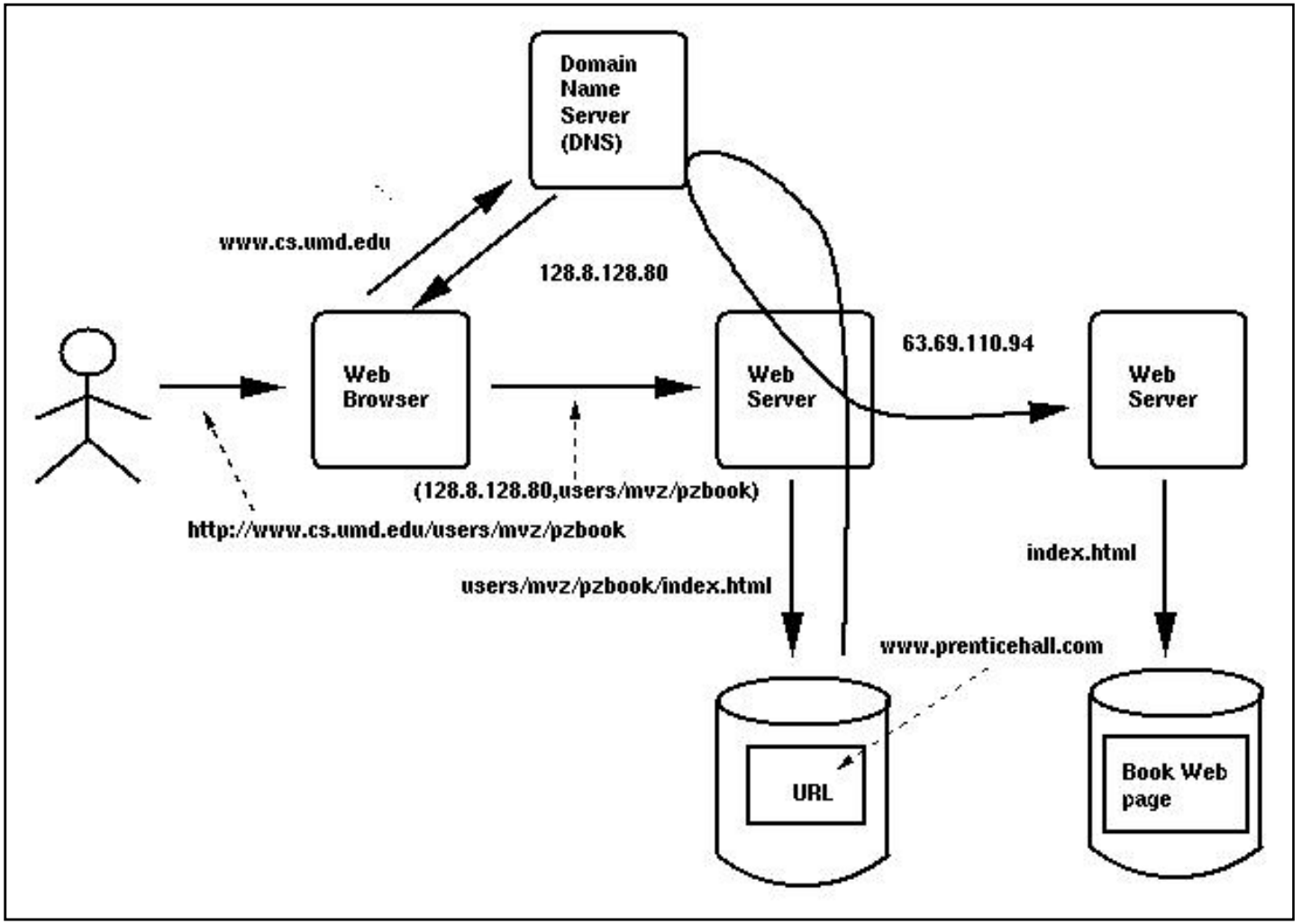
HTTP Example2



Web服务器

- 定义
  - WEB服务器主要功能是提供网上信息浏览服务。
    (1)应用层使用HTTP协议。
    (2)HTML文档格式。
    (3)浏览器统一资源定位器(URL)。
- 功能
  - 为Web文件提供存放空间；
  - 允许Internet用户通过Web服务器访问保存在Web服务器上的Web文件；
  - 提供对Web程序的支持，Web程序即利用CGI、ASP、PHP和JSP等动态网页技术语言编写的Web服务器端运行程序。
- 典型服务器
  - Apache、IIS、Tomcat、Nginx…..

Web运行过程

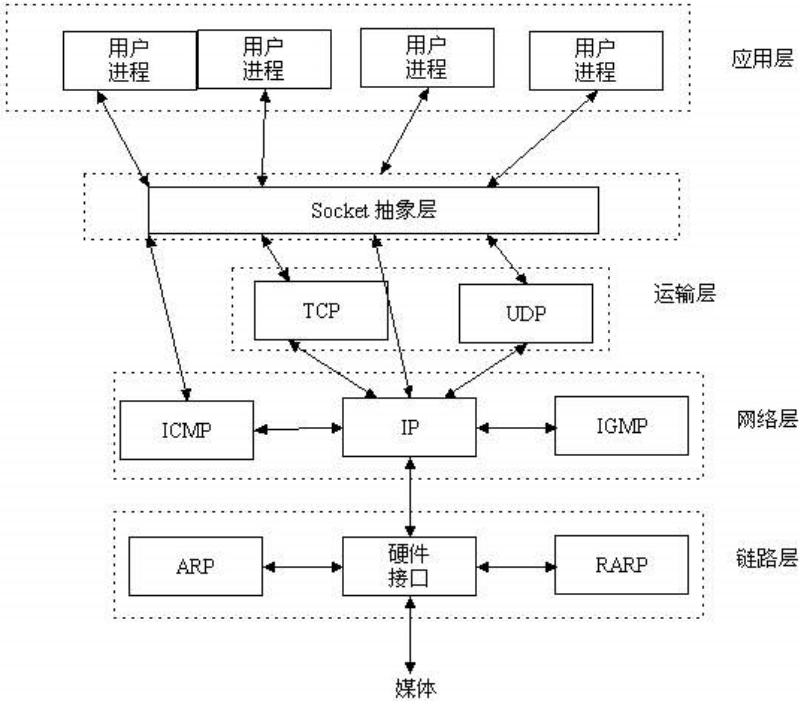

Domain
Name
Server
(DNS)
www.cs.umd.edu
128.8.128.80
Web
Browser
Web
Server
63.69.110.94
Web
Server
(128.8.128.80,users/mvz/pzbook)
http://www.cs.umd.edu/users/mvz/pzbook
index.html
users/mvz/pzbook/index.html
www.prenticehall.com
URL
Book Web
page

Web服务器工作过程

- 步骤
  - 第一步：Web浏览器向特定的服务器发出Web页面请求。
  - 第二步：Web服务器接收到Web页面请求后，寻找所请求的Web页面，并将所请求的Web页面传送Web浏览器。
  - 第三步：Web浏览器接收到所请求的Web页面，并将它显示出来。

Socket网络编程

- Socket是什么?
  - 应用层与TCP/IP协议族通信的中间软件抽象层，是一组接口。
  - Socket是一种设计模式，它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议

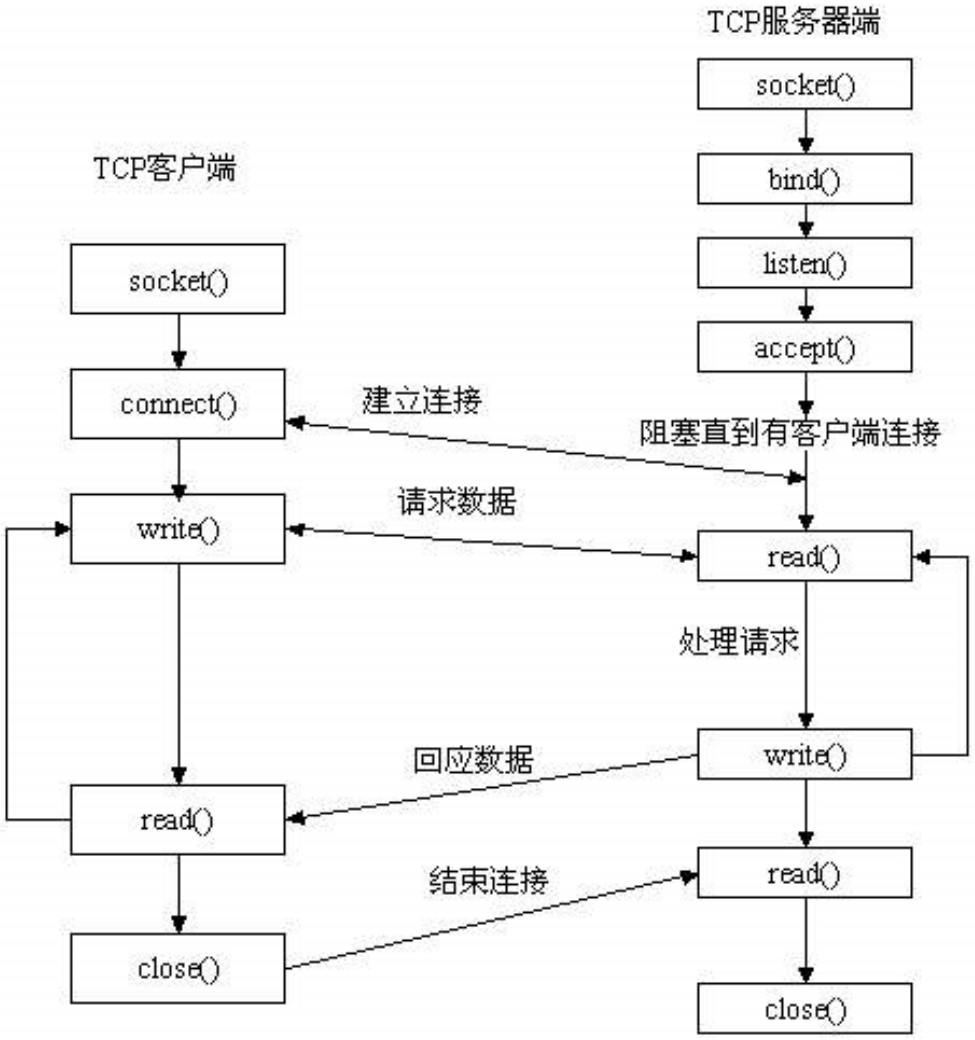


Socket概念

- 编程中的socket

echo	7	验证2台计算机连接有效性
daytime	13	服务器当前时间文本描述
ftp	20/21	21用于命令,20用户数据
telnet	23	远程登录
smtp	25	邮件发送
whois	43	网络管理的目录服务
finger	79	主机用户信息
http	80	HTTP
pop3	110	邮局协议
nntp	119	网络新闻传输协议, 发布Usenet新闻

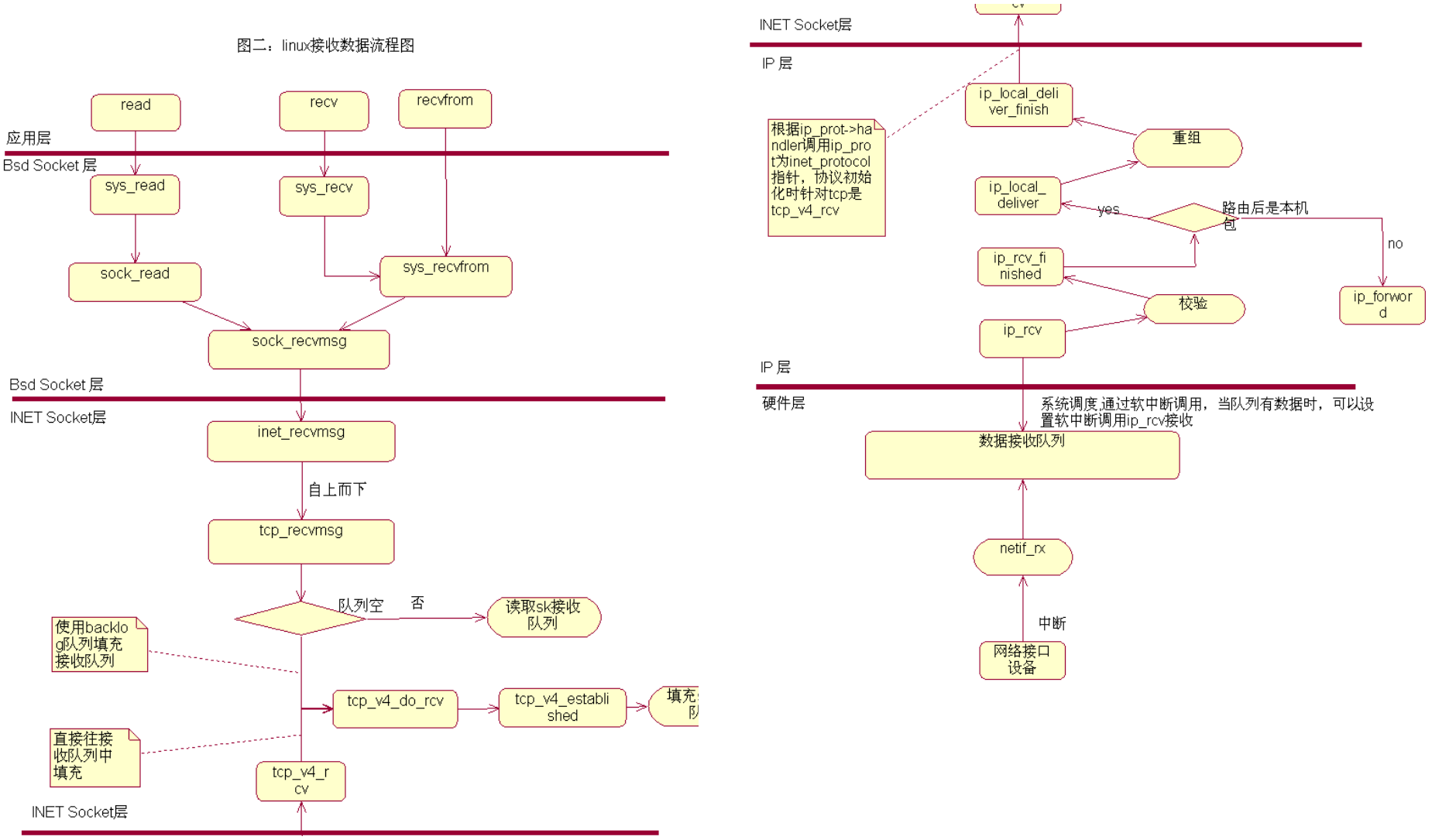
Socket接口框架



服务器端先初始化Socket，然后与端口绑定(bind)，对端口进行监听(listen)，调用accept阻塞，等待客户端连接。

这时如果有个客户端初始化一个Socket，然后连接服务器(connect)，如果连接成功，这时客户端与服务器端的连接就建立了。客户端发送数据请求，服务器端接收请求并处理请求，然后把回应数据发送给客户端，客户端读取数据，最后关闭连接，一次交互结束。

网络接收流程分析


图二：linux接收数据流程图
read
recv
recvfrom
应用层
Bsd Socket 层
sys_read
sys_recv
sock_read
sys_recvfrom
sock_recvmsg
Bsd Socket 层
INET Socket层
inet_recvmsg
自上而下
tcp_recvmsg
队列空
否
读取sk接收队列
使用backlog队列填充接收队列
tcp_v4_do_rcv
tcp_v4_established
直接往接收队列中填充
tcp_v4_rcv
INET Socket层
INET Socket层
IP 层
ip_local_deliver_finish
根据ip_prot->handler调用ip_prot为inet_protocol指针，协议初始化时针对tcp是tcp_v4_rcv
重组
ip_local_deliver
yes
路由后是本机包
no
ip_rcv_finished
校验
ip_forword
ip_rcv
IP 层
硬件层
系统调度,通过软中断调用，当队列有数据时，可以设置软中断调用ip_rcv接收
数据接收队列
netif_rx
中断
网络接口设备

网络IO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）

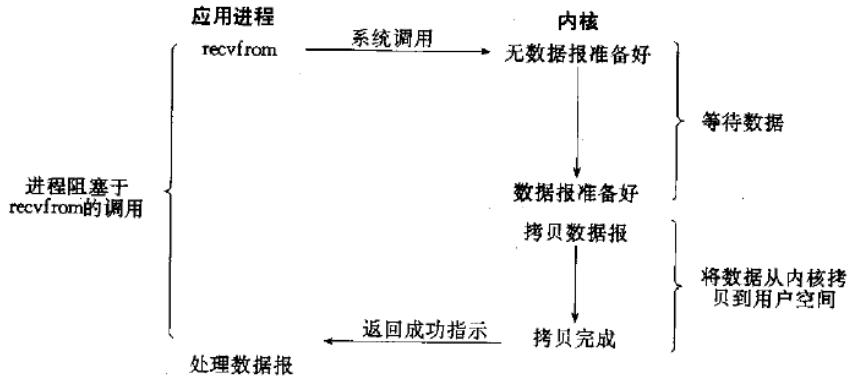
异步I/O (POSIX aio_函数)

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

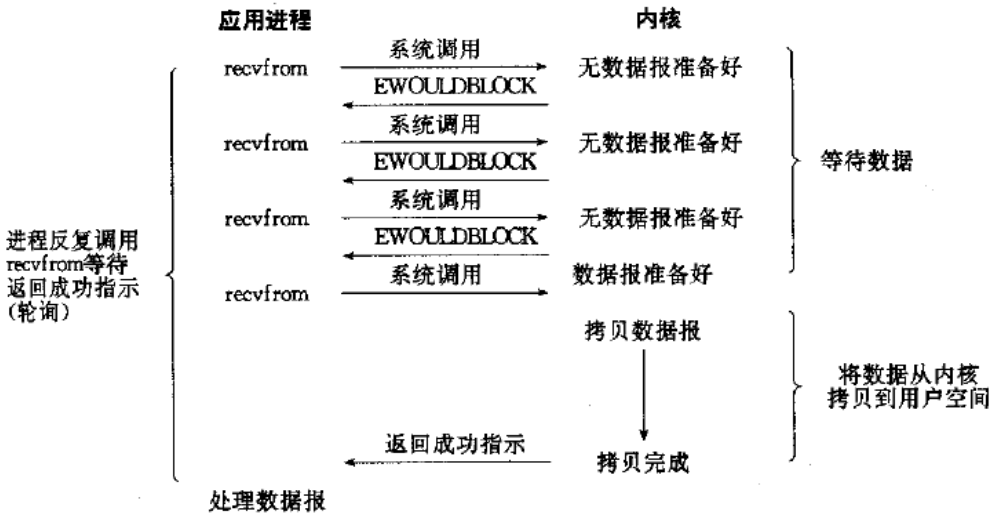


这种模型非常经典，也被广泛使用，优势在于非常简单，等待的过程中占用的系统资源微乎其微，程序调用返回时，必定可以拿到数据；
但简单也带来一些缺点，程序在数据到来并准备好以前，不能进行其他操作，需要有一个线程专门用于等待

非阻塞I/O模式

- 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);
```

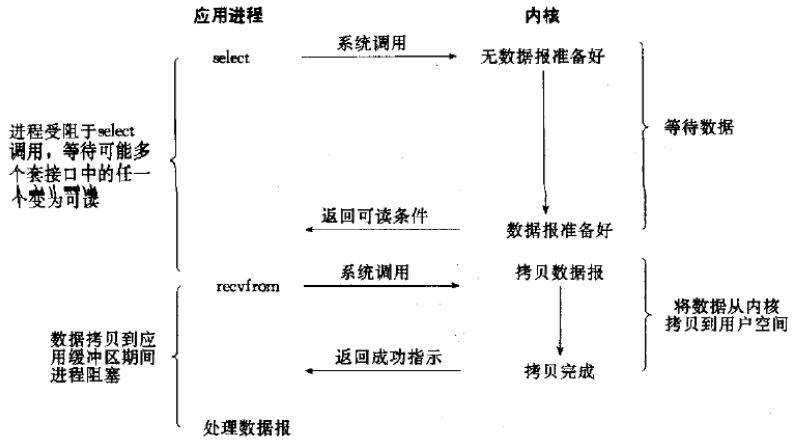


这种模式在没有数据可以接收时，可以进行其他的一些操作，比如有多个socket时，可以去查看其他socket有没有可以接收的数据；

实际应用中，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

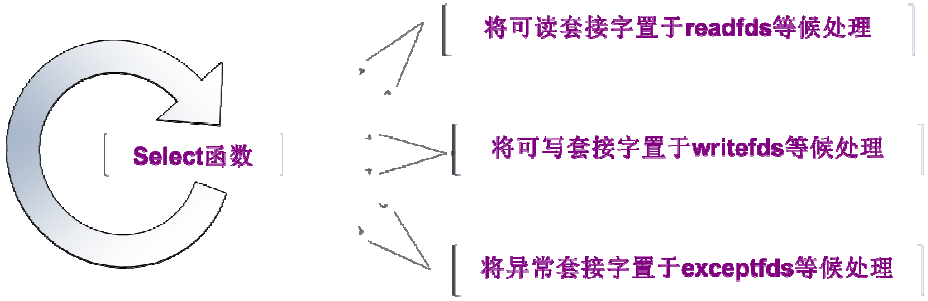
I/O复用（multiplexing）

- 思想
  - 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/poll方法

- 功能
  - Select类似，确定哪些IO准备好
  - 用一个结构保存要监视的每个连接的数组，当在网络套接字上发现数据时，通过回调机制调用处理函数。
- 优缺点
  - 不限制FD集合大小
  - 避免了FD轮询
  - 内核事件通知效率高
  - 结构会非常大，在列表中添加新的网络连接时，修改结构会增加负载并影响性能

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。

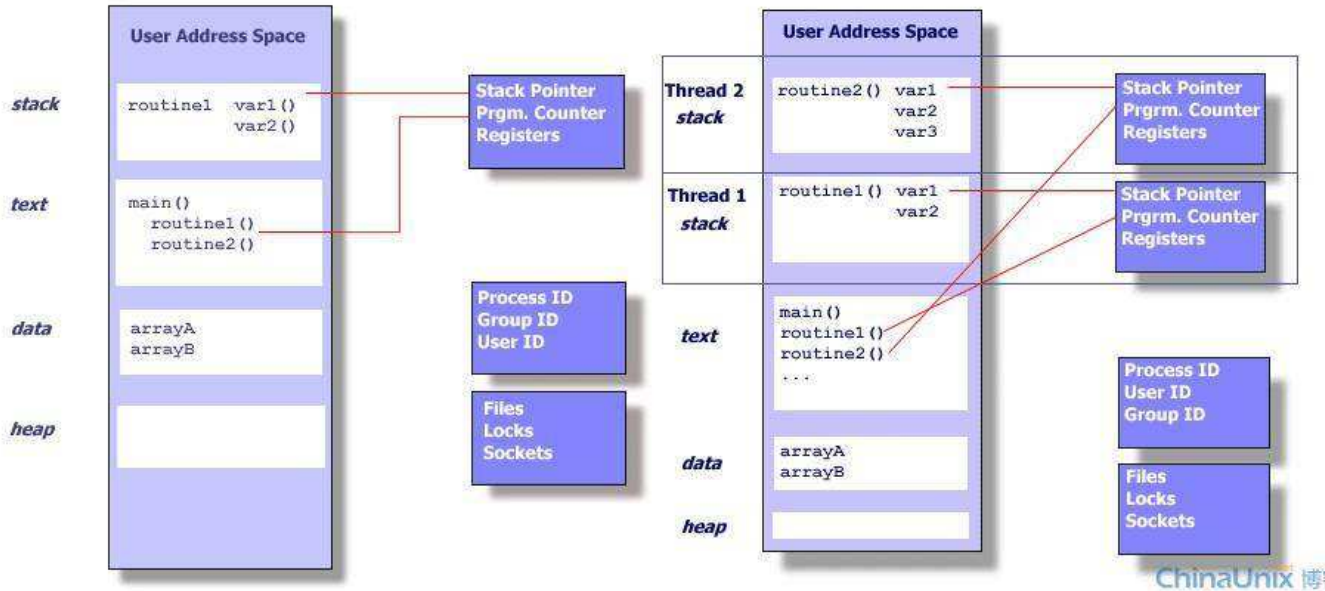
```
#include <sys/select.h>
#include <sys/time.h>
int res = epoll_wait(epfd, events, 20, 120);

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

IO复用（事件驱动）举例

- 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

多线程并发

- 概念
  - 进程（process）：程序在一个数据集合上的运行过程，是系统进行资源分配和调度的一个独立单位
  - 线程（thread）：线程是进程中的一个实体，是系统调度的基本单位。线程自己基本上不占有系统资源，他可以统一进程中的其他线程共享进程资源。



Unix进程　　Unix进程内部的线程

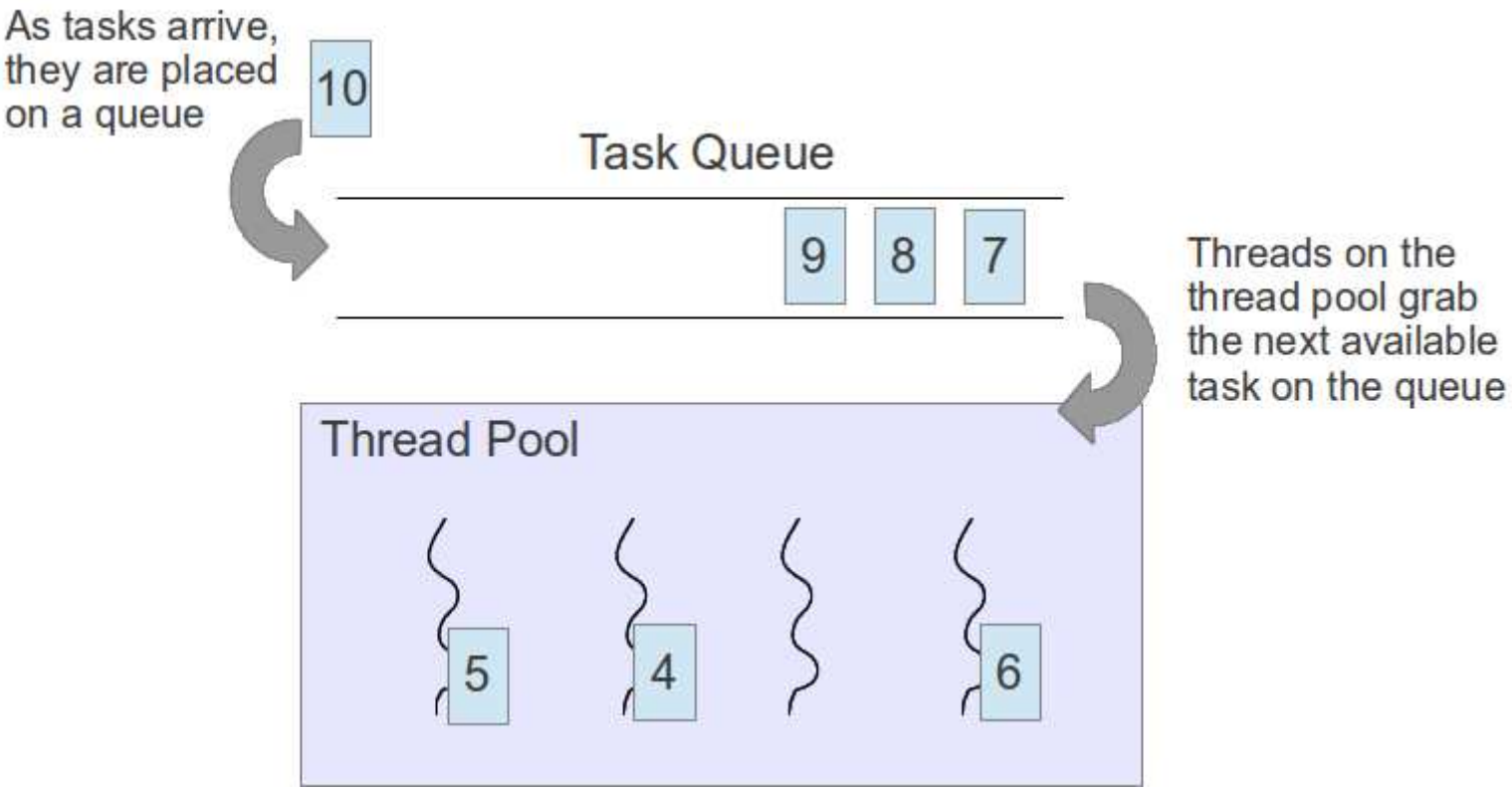
Posix Threads 接口

- *Pthreads:* ~60个控制线程的标准C接口函数
  - 1. 线程创建
    - pthread_create(pthread_t *tid, …, func *f, void *arg)
    - pthread_join(pthread_t tid, void **thread_return)
  - 2. 线程终止
    - pthread_cancel(pthread_t tid)
    - pthread_exit(void *tread_return)
    - return (in primary thread routine terminates the thread)
    - exit() (terminates all threads)
  - 3. 线程同步
    - 互斥锁（Mutex）、条件变量（Condition）、信号量（Semaphore）、自旋锁（Spinlock）、win32 CRITICAL_SECTION（临界区）
    - 抛开效率等问题不谈，信号量能够解决所有的线程间同步问题。

什么是线程池

- 线程池：一种成熟的线程使用模式，以多个线程[循环执行]多个应用逻辑的线程集合
  - 池化技术：线程池、连接池、内存池、对象池
- 技术背景
  - 1、需求（单位之间内必须处理巨大的连接请求）
  - 2、传统的多线程方案（服务器模型不能满足需求：即时创建，即时销毁的策略）
  - 3、线程池的出现正是着眼于线程本身的开销（最大程度的复用对象）
- 效能分析
  - 假设一台服务器完成一项任务的时间为 T ，　　T1：创建线程的时间 ，T2:在线程中执行任务的时间，包括线程间同步所需时间，　T3：线程销毁的时间。T = T1 + T2 + T3
  - 频繁的创建和销毁线程：T1，T3将占有相当大的比例。策略：将T1，T3安排在服务器程序启动和结束时间段或者是一般的空闲时间段
  - 优点：a调整了T1，T3的时间段；b显著的减少了创建的线程数目

线程池


As tasks arrive, they are placed on a queue
10
Task Queue
9
8
7
Threads on the thread pool grab the next available task on the queue
Thread Pool
5
4
6

线程池优点

- 逻辑框架
  - 线程池管理器：用于创建并管理线程
  - 工作线程：线程池中实际执行的线程
  - 任务接口：线程池是与具体任务无关的
  - 任务队列：保存具体需要执行的任务
- 优点
  - 1、缩短应用程序的响应时间。因为在线程池中有线程的线程处于等待分配任务状态（只要没有超过线程池的最大上限），无需创建线程。
  - 2、不必管理和维护生存周期短暂的线程，不用在创建时为其分配资源，在其执行完任务之后释放资源。
  - 3、线程池会根据当前系统特点对池内的线程进行优化处理。

多进程/线程 vs IO复用

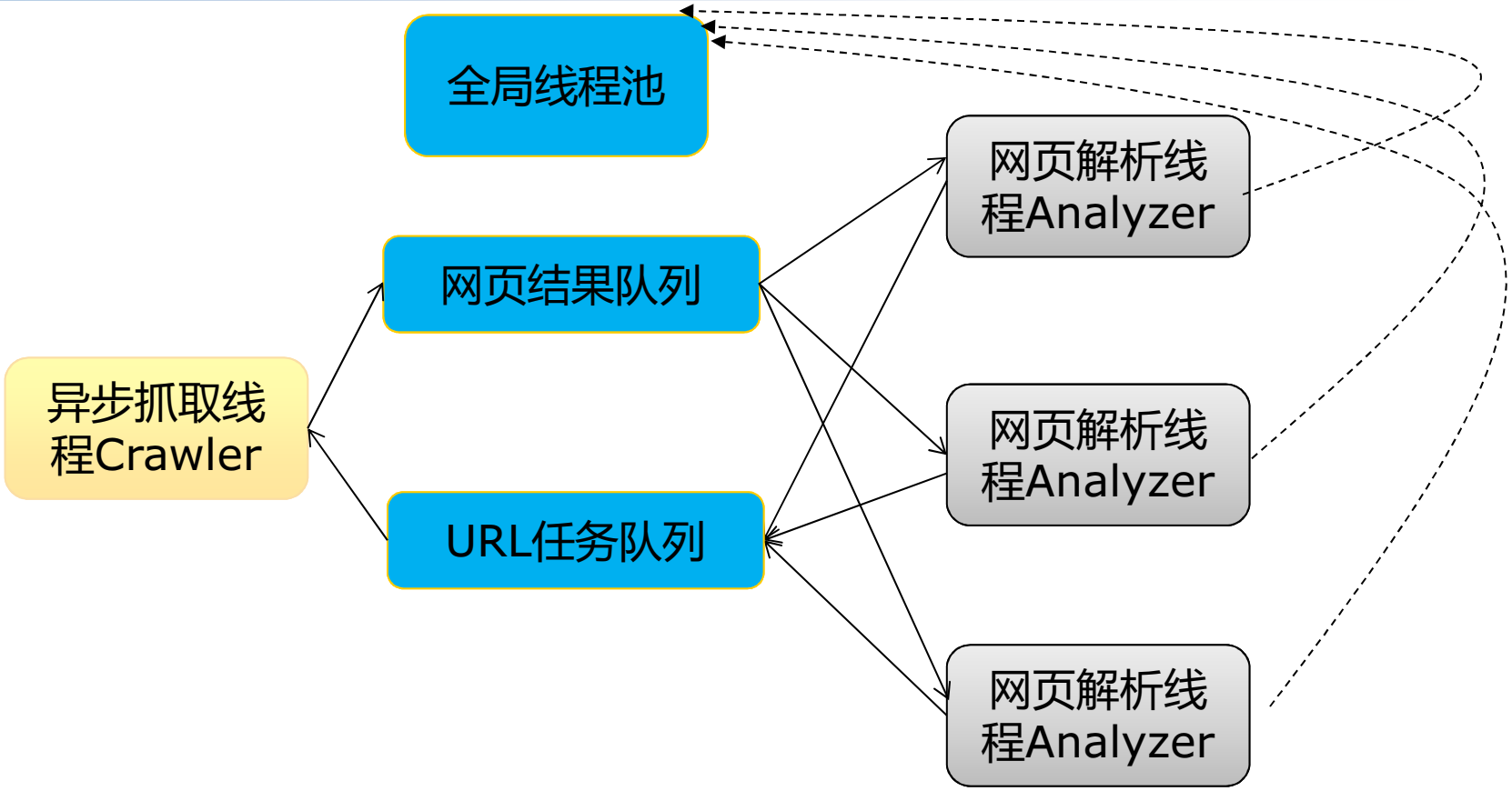
- 多线程缺点
  - 每个连接启动一个新线程。
  - 在 RAM 和 CPU 方面增加相当大的开销，因为每个线程都需要自己的执行空间。
  - 如果每个线程都忙于处理网络连接，线程之间的上下文切换会很频繁。
  - 最后，许多内核并不适 于处理如此大量的活跃线程。
- Tradeoff
  - IO密集型，互联网应用事件驱动更合适
  - CPU密集型
  - 多进程或线程适合于CPU密集型服务，如业务逻辑、图形图像、数据库读写，交互式的长连接应用等。

消息队列

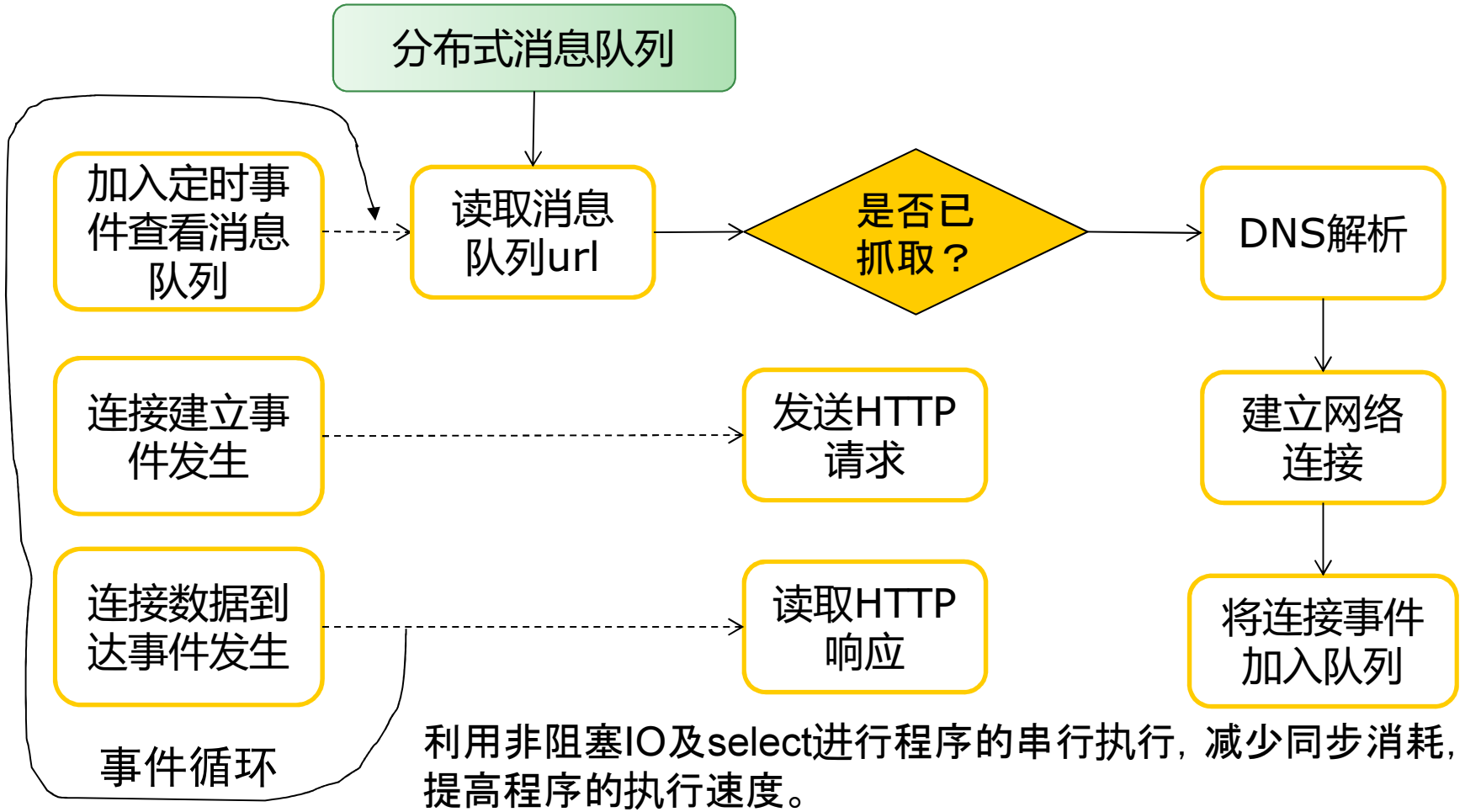
- 定义
  - 消息队列也叫报文队列，就是一个消息的链表。可以把消息看作一个记录，具有特定的格式以及特定的优先级。对消息队列有写权限的进程可以向中按照一定的规则添加新消息；对消息队列有读权限的进程则可以从消息队列中读走消息
  - 消息队列也是进程间通信的主要方法，目前许多中间件技术都是采用消息队列，如ActiveMQ、ZEROMQ、REDIS、Linux自带消息队列等。
- 操作
  - 创建消息队列
  - 往消息队列内写入消息，主要队列满
  - 其他进程不断的读取消息队列内消息
  - 直到队列内没有可读的消息。
  - 删除消息队列
  - 系统关闭


1、实验任务
2、相关基础原理
3、爬虫设计与实现
4、实验说明

并行爬虫架构


全局线程池
网页结果队列
异步抓取线程Crawler
URL任务队列
网页解析线程Analyzer
网页解析线程Analyzer
网页解析线程Analyzer

爬取模块设计



利用非阻塞IO及select进行程序的串行执行，减少同步消耗，提高程序的执行速度。

将爬虫设计为有限状态自动机，每个爬虫都是一个独立的功能实体，减少了非阻塞IO的编码量及出错机率。

DNS解析实现

- The Internet maintains a mapping between IP addresses and domain names in a huge worldwide distributed database called *DNS*
  - Conceptually, programmers can view the DNS database as a collection of millions of *host entry structures*:

```
/* DNS host entry structure */
struct hostent {
   char    *h_name;        /* official domain name of host */
   char    **h_aliases;    /* null-terminated array of domain names */
   int     h_addrtype;     /* host address type (AF_INET) */
   int     h_length;       /* length of an address, in bytes */
   char    **h_addr_list; /* null-terminated array of in_addr structs */
};
```

- Functions for retrieving host entries from DNS:
  - **`gethostbyname`**: query key is a DNS domain name.
  - **`gethostbyaddr`**: query key is an IP address.

DNS解析例子

```
int main(int argc, char **argv) { /* argv[1] is a domain name */
    char **pp;                    /* or dotted decimal IP addr */
    struct in_addr addr;
    struct hostent *hostp;

    if (inet_aton(argv[1], &addr) != 0)
        hostp = Gethostbyaddr((const char *)&addr, sizeof(addr),
                AF_INET);
    else
        hostp = Gethostbyname(argv[1]);
    printf("official hostname: %s\n", hostp->h_name);

    for (pp = hostp->h_aliases; *pp != NULL; pp++)
        printf("alias: %s\n", *pp);

    for (pp = hostp->h_addr_list; *pp != NULL; pp++) {
        addr.s_addr = ((struct in_addr *)*pp)->s_addr;
        printf("address: %s\n", inet_ntoa(addr));
    }
}
```

建立socket连接open_clientfd

```
int open_clientfd(char *hostname, int port) {
  int clientfd;
  struct hostent *hp;
  struct sockaddr_in serveraddr;
  if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1; /* check errno for cause of error */
   /* Fill in the server's IP address and port */
  if ((hp = gethostbyname(hostname)) == NULL)
    return -2; /* check h_errno for cause of error */
  bzero((char *) &serveraddr, sizeof(serveraddr));
  serveraddr.sin_family = AF_INET;
  bcopy((char *)hp->h_addr_list[0],
        (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
  serveraddr.sin_port = htons(port);
  /* Establish a connection with the server */
  if (connect(clientfd, (SA *) &serveraddr,
      sizeof(serveraddr)) < 0)
    return -1;
  return clientfd;
}
```

This function opens a connection from the client to the server at `hostname:port`

Create socket

Create address

Establish connection

http请求构造

```
int Make_httprequest(int socket)
{
    char req[1024];
     sprintf(req, "POST %s HTTP/1.0\r\n", path);
     sprintf(req, "Host: %s\r\n", host);
      sprintf(req, "User-Agent: " USER_AGENT "\r\n");
     sprintf(req, "Content-Type: application/x-www-form-urlencoded\r\n");
      sprintf(req, "Content-Length: %d\r\n", strlen(postdata));
     sprintf(req, "\r\n"); sprintf(req, "%s", postdata);
     Send(socket, req);
}
```

下载网页

Send http request to server

Receive line from web server

```
int main(int argc, char **argv)
{
    int clientfd, port;
    char *host, buf[MAXLINE];
    rio_t rio;
    host = argv[1];  port = atoi(argv[2]);
    clientfd = Open_clientfd(host, port);
    Make_httprequest(clientfd);
    Rio_readinitb(&rio, clientfd);
    printf("Enter message:"); fflush(stdout);
    while (Fgets(buf, MAXLINE, clientfd) != NULL) {
        Fputs(buf, stdout);
        printf("Enter message:"); fflush(stdout);
    }
    Close(clientfd);
    exit(0);
}
```

几个实现要点

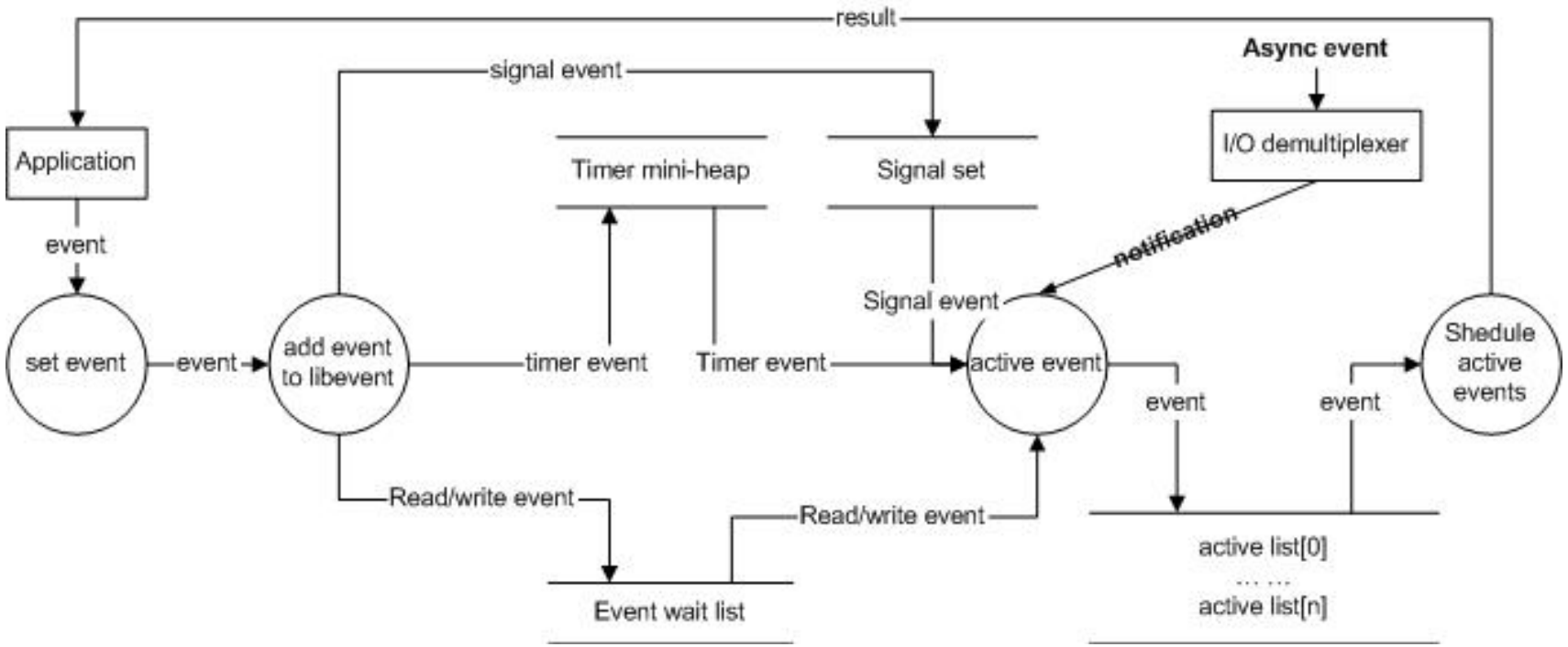
- Connection: keep-alive
  - 在HTTP响应的头部字段中有一项keep-alive选项，服务器不会马上断开连接，需要对数据何时结束进行判断
  - 静态页面：Content-Length：
  - 动态页面：Transfer-Encoding：chunked
  - chunk十六进制包长
- Accept-Encoding: gzip, deflate
  - 传输编码为gzip，需要对gzip编码在内存中进行解压，然后进行分析
  - 利用zlib库进行解压
- HTTP响应处理
  - 只存储200 OK成功响应，其他无视
- URL去重
  - 检查某个URL是否已经被抓过了
  - 基于Bloom filter

基于Libevent异步IO库的并发抓取

- 概述
  - 一个基于事件触发的网络库
- 特点
  - 事件驱动（event-driven），高性能;
  - 轻量级，专注于网络，不如ACE那么臃肿庞大；
  - 跨平台，支持Windows、 Linux、 *BSD和Mac Os；
  - 支持多种I/O多路复用技术， epoll、 poll、 dev/poll、 select和kqueue等；
  - 支持I/O，定时器和信号等事件；
- 应用
  - Libevent已被广泛应用，作为底层的网络库；比如memcached、 Vomit、 Nylon、 Netchat等等。
  - Libevent代码里有很多有用的设计技巧和基础数据结构，比如信息隐藏、函数指针、c语言的多态支持、链表和堆等。

Libevent原理

- 运行原理
  - 使用Libevent也是向Libevent框架注册相应的事件和回调函数
  - 当这些事件发生时，Libevent 会调用这些回调函数处理相应的事件（I/O读写、定时和信号）。

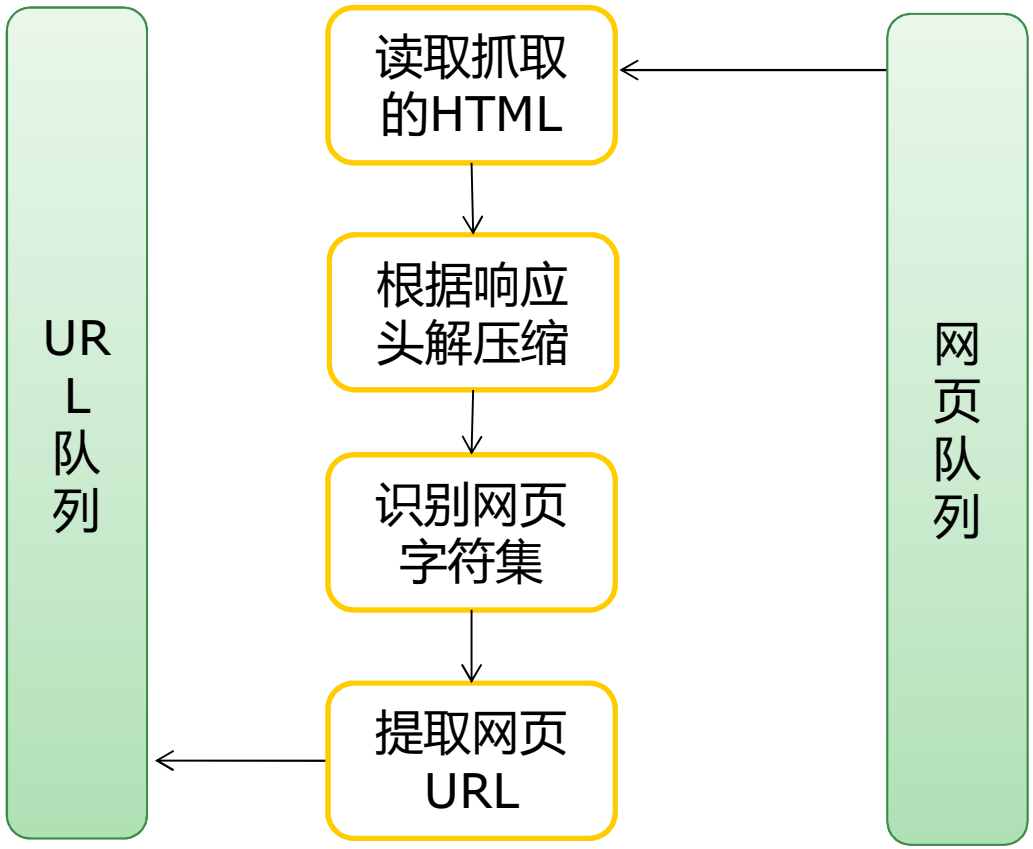


```
struct event {
 TAILQ_ENTRY (event) ev_next;
 TAILQ_ENTRY (event) ev_active_next;
 TAILQ_ENTRY (event) ev_signal_next;
 unsigned int min_heap_idx; /* for managing timeouts */
 struct event_base *ev_base;
 int ev_fd;
 short ev_events;
 short ev_ncalls;
 short *ev_pncalls; /* Allows deletes in callback */
 struct timeval ev_timeout;
 int ev_pri;  /* smaller numbers are higher priority */
 void (*ev_callback)(int, short, void *arg);
 void *ev_arg;
 int ev_res;  /* result passed to event callback */
 int ev_flags;
};
```

向libevent添加事件，需先设置event对象，通过调用libevent提供的函数有：
void event_set(struct event *ev, int fd, short events,void (*callback)(int, short, void *), void *arg)
1.设置事件ev绑定的文件描述符或者信号，对于定时事件，设为-1即可；
2.设置事件类型，比如EV_READ|EV_PERSIST, EV_WRITE, EV_SIGNAL等；
3.设置事件的回调函数以及参数arg；
4.初始化其它字段，比如缺省的event_base和优先级；

```
int main(int argc, char **argv)
{
...
    ev_init();

    /* Setup listening socket */

    event_set(&ev_accept, listen_fd, EV_READ|EV_PERSIST, on_accept, NULL);
    event_add(&ev_accept, NULL);

    /* Start the event loop. */
    event_dispatch();
}
```

网页解析模块设计

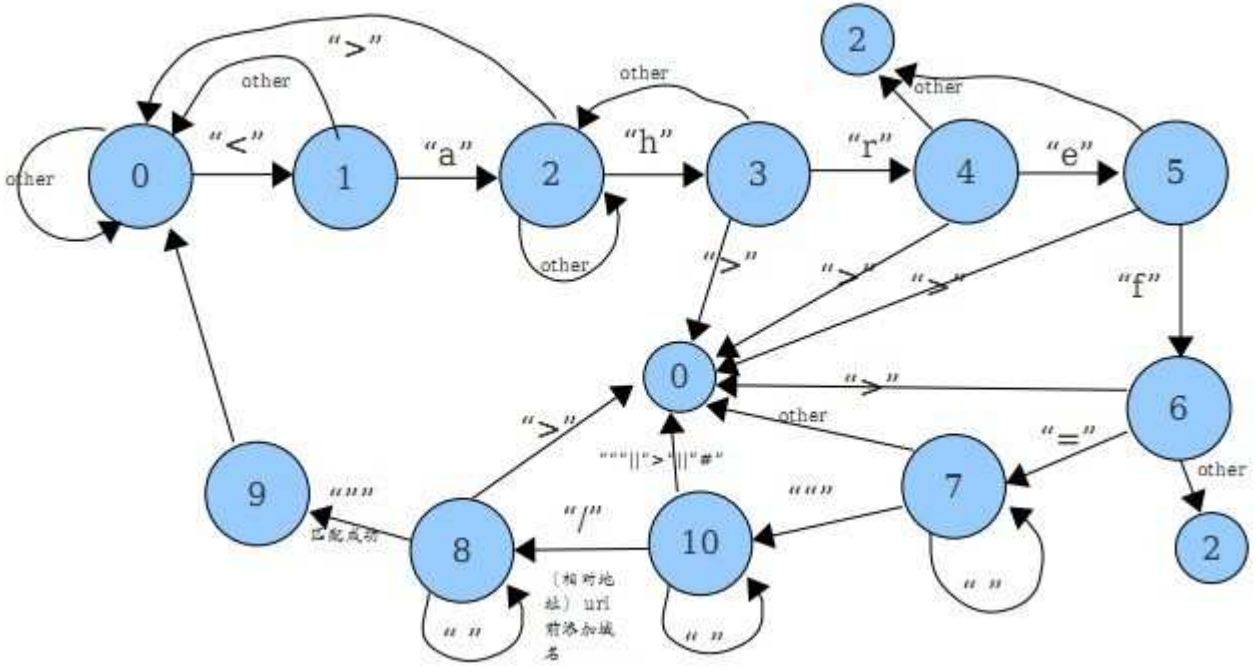


网页解析处理

- HTML页解码
  - 用zlib进行解码
- HTML页面字符集识别
- URL提取
  - 相对路径转换为绝对路径
  - 方法一：原始字符串匹配和暴力分析
    - String match “<a href=”, use strstr() function
    - 可能遇到以下类型的标签
      - <a class=“…” href=“…” …>的标签
      - <a … href=“javascript:void(0)” …>
      - <a … href=“./../a.html” …>
      - <a … href=‘a.html’ …>
    - 不完备容易漏掉
  - 方法二：正则表达式
    - Regex正则表达式库
    - 正则表达式要考虑周全、计算复杂度高

基于DFA的url提取

- linux C regex库只提供贪婪匹配，不支持非贪婪匹配
  - 贪婪匹配 href="m.html",xxxxxxxxsrc="n.jpg"
  - 非贪婪匹配 href="m.html",xxxxxxxxsrc="n.jpg"
- 功能
  - 通过有限状态自动机进行字符串匹配。



线程池模块

- 设计概念

```
typedef struct work_st {
    void (*routine) (void *);
    void *arg;
    struct work_st *next;
} work_t;

typedef struct _threadpool_st {
    // you should fill in this structure with whatever you need
    int num_threads;   //number of active threads
    int qsize;                  //number in the queue
    pthread_t *threads;         //pointer to threads
    work_t *qhead;              //queue head pointer
    work_t *qtail;              //queue tail pointer
    pthread_mutex_t qlock;              //lock on the queue list
    pthread_cond_t q_not_empty;         //non empty and empty
condidtion vairiables
    pthread_cond_t q_empty;
    int shutdown;
    int dont_accept;
} _threadpool;
```

回

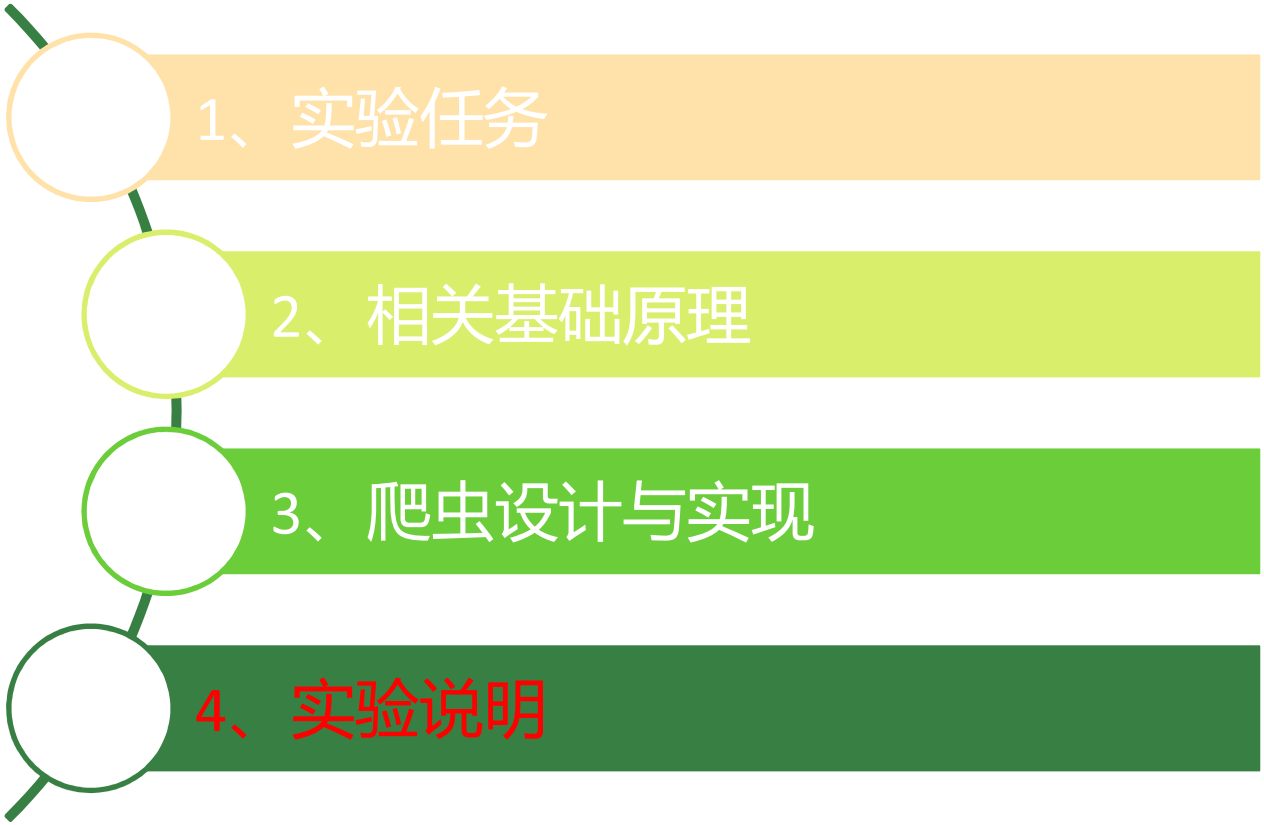
线程池操作

```
void dispatch(threadpool from_me, dispatch_fn dispatch_to_here,
              void *arg)
{
    _threadpool *pool = (_threadpool *) from_me;
    work_t *cur;
    int k;
    k = pool->qsize;
    //make a work queue element.
    cur = (work_t *) malloc(sizeof(work_t));
    if(cur == NULL) {
        fprintf(stderr, "Out of memory creating a work struct!\n");
        return;
    }
    cur->routine = dispatch_to_here;
    cur->arg = arg;
    cur->next = NULL;
    pthread_mutex_lock(&(pool->qlock));
    if(pool->dont_accept) { //Just incase someone is trying to queue more
        free(cur); //work structs.
        return;
    }
    if(pool->qsize == 0) {
        pool->qhead = cur;  //set to only one
        pool->qtail = cur;
        pthread_cond_signal(&(pool->q_not_empty));  //I am not empty.
    } else {
        pool->qtail->next = cur;        //add to end;
        pool->qtail = cur;
    }
    pool->qsize++;
    pthread_mutex_unlock(&(pool->qlock));  //unlock the queue.
}
```

```
void *do_work(threadpool p)
{
    _threadpool *pool = (_threadpool *) p;
    work_t *cur;      //The q element
    int k;
    while(1) {
        pool->qsize = pool->qsize;
        pthread_mutex_lock(&(pool->qlock));  //get the q lock.
        while( pool->qsize == 0) {          //if the size is 0 then wait.
            if(pool->shutdown) {
                pthread_mutex_unlock(&(pool->qlock));
                pthread_exit(NULL);
            }
            //wait until the condition says its no emtpy and give up the lock.
            pthread_mutex_unlock(&(pool->qlock));  //get the qlock.
            pthread_cond_wait(&(pool->q_not_empty), &(pool->qlock));

            //check to see if in shutdown mode.
            if(pool->shutdown) {
                pthread_mutex_unlock(&(pool->qlock));
                pthread_exit(NULL);
            }
        }
        cur = pool->qhead;       //set the cur variable.
        pool->qsize--;                      //decriment the size.
        if(pool->qsize == 0) {
            pool->qhead = NULL;
            pool->qtail = NULL;
        } else {
            pool->qhead = cur->next;
        }
        if(pool->qsize == 0 && ! pool->shutdown) {
            //the q is empty again, now signal that its empty.
            pthread_cond_signal(&(pool->q_empty));
        }
        pthread mutex unlock(&(pool->qlock));
        (cur->routine) (cur->arg);   //actually do work.
        free(cur);                                          //free the work storage.
    }
}
```

消息队列

- 很多实现方式
  - 自己实现
  - *Nix系统自带的Posix message queue
    - The mq_open() function creates a new message queue or opens an existing queue
    - The mq_send() function writes a message to a queue.
    - The mq_receive() function reads a message from a queue.
    - The mq_close() function closes a message queue that the process previously opened.
- 开源消息队列
  - nanomsg
  - ZeroMQ作者用C语言新写的消息队列库
  - http://nanomsg.org/index.html


1、实验任务
2、相关基础原理
3、爬虫设计与实现
4、实验说明

代码说明

- 实现要求
  - Crawler实现IO复用
  - 实现线程池
  - 实现多线程的网页分析和URL提取
  - Crawler和Analyzer之间以消息队列通信

实验说明

- 测试环境在自己机器上搭建Web服务器
  - Web 服务器：apache
  - 将网页放在/var/www/html下
- 目标网站
  - 大约16万个网页
  - 校内服务器
  - 部分链接页面不可用，需要剔除

文件总数	161,033
文件总大小	17,052,578,841
压缩包大小	3,898,532,313

- 运行要求

```
# ./crawler http://192.168.1.2/index.html result.txt
第一个输入参数是入口地址
第二个参数是输出结果文件名
Result.txt格式：
每行一个url信息，每行3个字段：序号 绝对url 文件大小
```


THE END