Linux网络编程

通用地址结构：

```
/* POSIX.1g 规范规定了地址族为 2 字节的值.  */
typedef unsigned short int sa_family_t;
/* 描述通用套接字地址  */
struct sockaddr{
    sa_family_t sa_family;  /* 地址族.  16-bit*/
    char sa_data[14];   /* 具体的地址值 112-bit */
};
```

AF_xxx 初始化 socket 地址
PF_xxx 初始化 socket

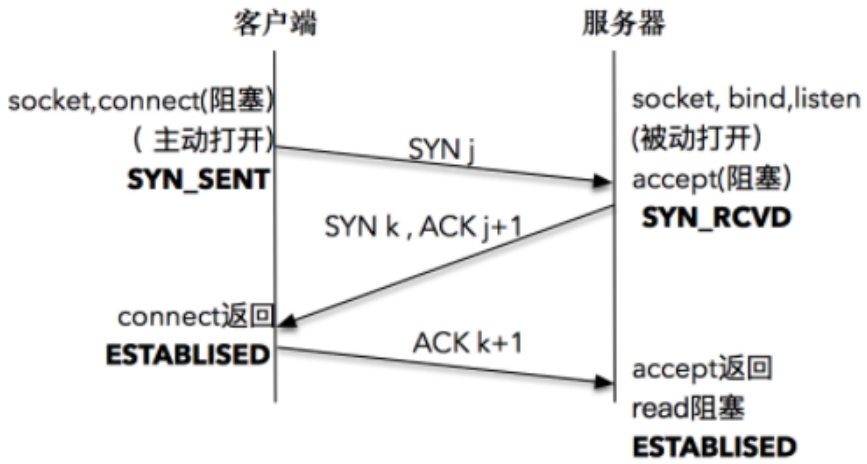
ipv4 ipv6 本地地址

```
/* IPV4 套接字地址，32bit 值.  */
typedef uint32_t in_addr_t;
struct in_addr
{
  in_addr_t s_addr;
};
/* 描述 IPV4 的套接字地址格式  16字节 */
struct sockaddr_in
{
  sa_family_t sin_family; /* 16-bit */
  in_port_t sin_port;     /* 端口口  16-bit*/
  struct in_addr sin_addr;    /* Internet address. 32-bit */
  /* 这里仅仅用作占位符，不做实际用处  */
  unsigned char sin_zero[8];
};
/* 描述 IPV6 28字节 */
struct sockaddr_in6
{
  sa_family_t sin6_family; /* 16-bit */
  in_port_t sin6_port;  /* 传输端口号 # 16-bit */
  uint32_t sin6_flowinfo; /* IPv6流控信息 32-bit*/
  struct in6_addr sin6_addr;  /* IPv6地址128-bit */
  uint32_t sin6_scope_id; /* IPv6域ID 32-bit */
};
/* 本地套接字 */
struct sockaddr_un {
    unsigned short sun_family; /* 固定为 AF_LOCAL */
```

```
    char sun_path[108];   /* 路径名 */
};
```

建立连接

```
int socket(int domain, int type, int protocol);
bind(int fd, sockaddr * addr, socklen_t len);
int listen (int socketfd, int backlog);
int accept(int listensockfd, struct sockaddr *cliaddr, socklen_t
*addrlen);
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t
addrlen);
```



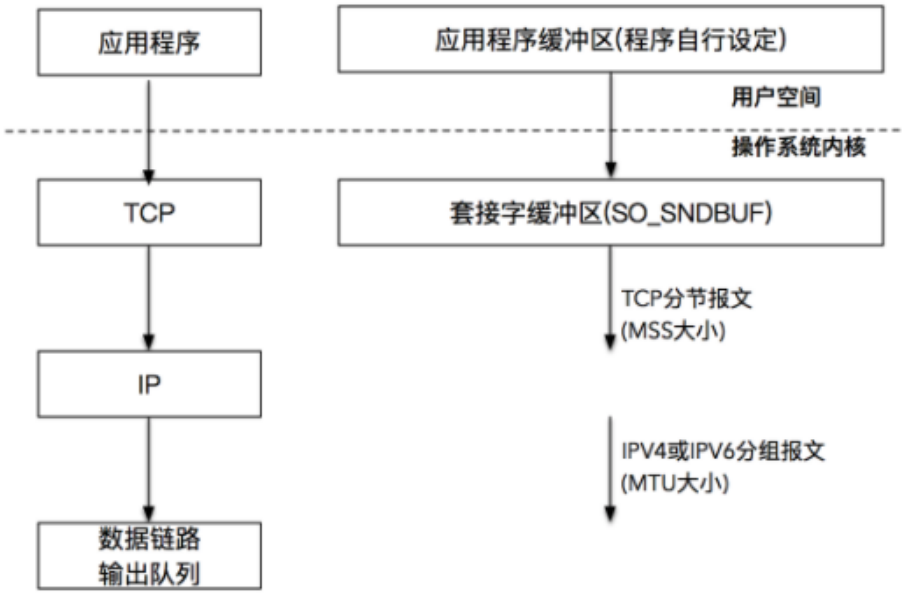
结合epoll ， accpet

发送数据：

```
ssize_t write (int socketfd, const void *buffer, size_t size)
ssize_t send (int socketfd, const void *buffer, size_t size, int flags)
ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags)
```

当我们的应用程序调用 write 函数时，实际所做的事情是把数据从应用程序中拷贝到操作系统内核的发送缓冲区中。
send：可以设置flag ， 带外数据 MSG_NOSIGNAL：还会向系统发送一个异常消息，如果不作处理，系统会出 BrokePipe，程序会退出

缓冲区大小不足，应用程序阻塞。

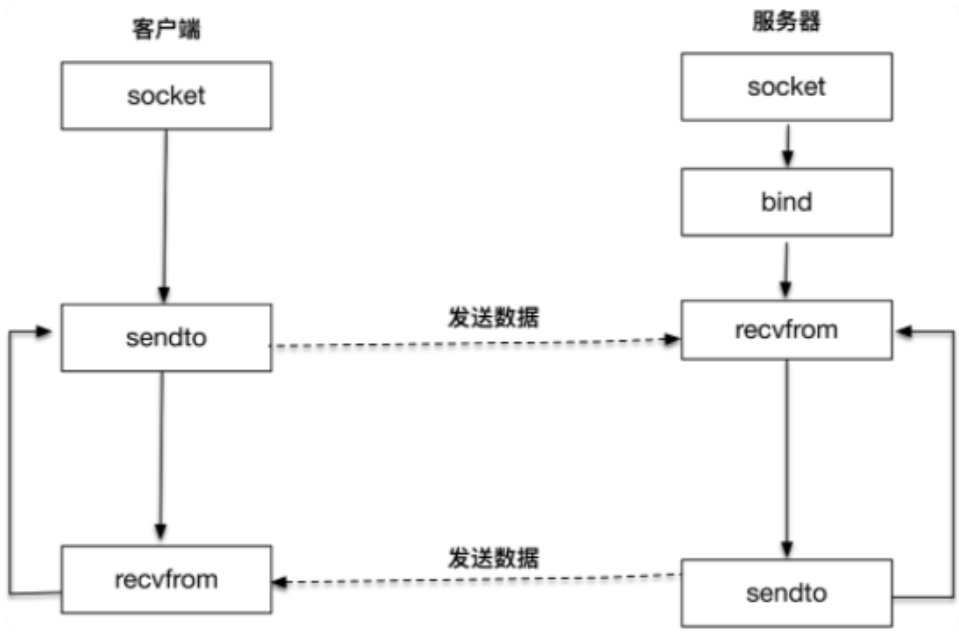


接收数据：

```
ssize_t read (int socketfd, void *buffer, size_t size)
```

返回0，EOF，FIN
返回-1，出错；
发送成功仅仅表示的是数据被拷贝到了发送缓冲区中，并不意味着连接对端已经收到所有的数据。至于什么时候发送到对端的接收缓冲区，或者更进一步说，什么时候被对方应用程序缓冲所接收，对我们而言完全都是透明的。

UDP：

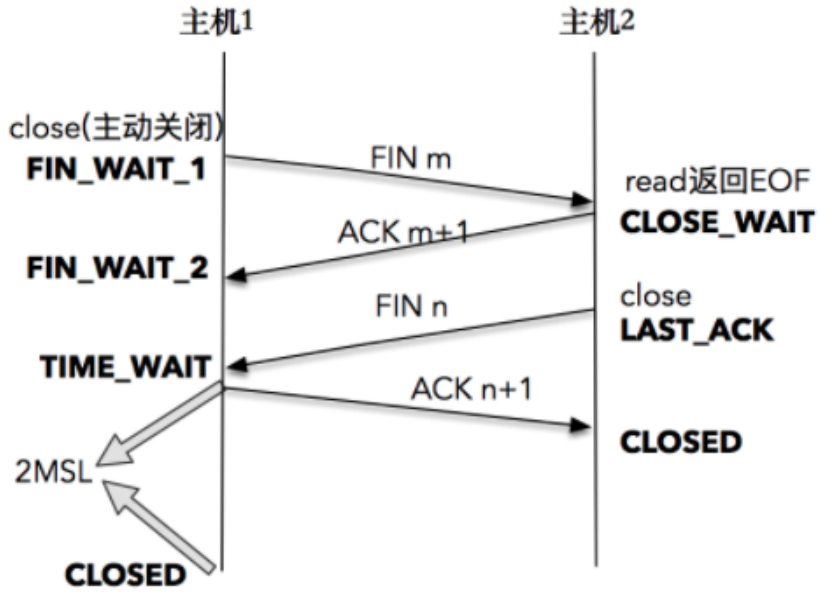



```
#include <sys/socket.h>
ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags,
                 struct sockaddr *from, socklen_t *addrlen);
ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags,
               const struct sockaddr *to, socklen_t addrlen);
```

UDP：服务器端重启后可以继续收到客户端的报文，这在 TCP 里是不可以的，TCP 断联之后必须重新连接才可以发送报文信息

本地套接字 IPC，本地进程间通信 AF_LOCAL。使用文件路径作为目标标识。

四次挥手：



主机 1 先发送 FIN 报文，主机 2 进入 CLOSE_WAIT 状态，并发送一个 ACK 应答，同时，主机 2 通过 read 调用获得 EOF，并将此结果通知应用程序进行主动关闭操作，发送 FIN 报文。
主机 1 在接收到 FIN 报文后发送 ACK 应答，此时主机 1 进入 TIME_WAIT 状态。

TIME_WAIT：
Linux系统停留在TIME_WAIT 的时间为固定的60秒。

1. 容错，防止RST
2. 连接“化身”和报文迷走有关系，为了让旧连接的重复分节在网络中自然消失

TIME_WAIT 的危害：

1. 内存资源占用
2. 端口占用

如何优化：

1. net.ipv4.tcp_max_tw_buckets 系统值调小
2. 调低 TCP_TIMEWAIT_LEN，重新编译系统
3. SO_LINGER

```
struct linger {
  int   l_onoff;       /* 0=off, nonzero=on */
  int   l_linger;       /* linger time, POSIX specifies units as seconds
*/
}
```

l_onoff = 0 , 默认行为 ， 尝试发送缓冲区内容
l_onoff != 0, l_linger = 0， 强行关闭 ， 数据不会发送， 发送RST，被动关闭方阻塞在recv
l_onoff !=0, l_linger != 0，线程阻塞，直到数据发送出去

4. net.ipv4.tcp_tw_reuse 可以复用处于 TIME_WAIT 的套接字为新的连接所用

关闭连接：

```
int close(int sockfd)
```

套接字引用计数减一，关闭 TCP 两个方向的数据流

```
int shutdown(int sockfd, int howto)
```

close 会关闭连接，并释放所有连接对应的资源，而 shutdown 并不会释放掉套接字和所有的资源
close 存在引用计数的概念，并不一定导致该套接字不可用；shutdown 则不管引用计数，直接使得该套接字不可用，如果有别的进程企图使用该套接字，将会受到影响
close 的引用计数导致不一定会发出 FIN 结束报文，而 shutdown 则总是会发出 FIN 结束报文，很重要

TCP Keep-Alive
可以通过select 保活，超时返回0。应用层控制处理。

拥塞控制（拥塞窗口）
TCP 就必须考虑多个连接共享在有限的带宽上，兼顾效率和公平性的控制
慢开始算法 拥塞避免算法
发送端：nagle
接收端：糊涂窗口综合症 延时ACK

服务器快速重新启动 地址已被使用

```
int on = 1;
setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
```

SO_REUSEADDR 套接字选项，允许启动绑定在一个端口，即使之前存在一个和该端口一样的连接，告诉操作系统内核，这样的 TCP 连接完全可以复用 TIME_WAIT 状态的连接。
本机服务器如果有多个地址，可以在不同地址上使用相同的端口提供服务

异常情况：
网络中断造成的对端无FIN包（read阻塞超时）
系统崩溃造成的对端无 FIN 包
崩溃后重启，重传的TCP分组到达重启后的系统，没有该连接的信息，返回RST
有FIN包，相当于接收缓冲区放置一个EOF

解决read阻塞：
1. 给套接字设置超时
   setsockopt(connfd, SOL_SOCKET, SO_RCVTIMEO, (const char *) &tv, sizeof tv);

2. 多路复用

为什么需要多路复用：

1. 如果一个套接字需要等待标准输入，又需要读取来自网络的信息，选择等待标准输入，则不能及时响应网络信息；选择后者，则又不能及时将标准输入的内容发送出去
2. 标准输入、套接字等都看做 I/O 的一路，多路复用的意思，就是在任何一路 I/O 有“事件”发生的情况下，通知应用程序去处理相应的 I/O 事件

select：
使用 select 函数，通知内核挂起进程，当一个或多个 I/O 事件发生后，控制权返还给应用程序，由应用程序进行 I/O 事件的处理

```
int select(int maxfd, fd_set *readset, fd_set *writeset, fd_set
*exceptset, const struct timeval *timeout)
返回：若有就绪描述符则为其数目，若超时则为0，若出错则为-1
struct timeval {
  long   tv_sec; /* seconds */
  long   tv_usec; /* microseconds */
};
```

设置成空 (NULL)，表示如果没有 I/O 事件发生，则 select 一直阻塞等待下去；
设置一个非零的值，这个表示等待固定的一段时间后从 select 阻塞调用中返回；
将 tv_sec 和 tv_usec 都设置成 0，表示根本不等待，检测完毕立即返回；

每次select之前需要保存fd_set，select每次调用时，内核会修改描述符集合，之后应用程序使用FD_ISSET 对每个描述符判断是否有事件发生。

描述符可读：

1. 接收缓冲区有数据可以读
2. 是对方发送了 FIN
3. 监听套接字而言的，有已经完成的连接建立，accept
4. 套接字有错误待处理，使用 read 函数去执行读操作，不阻塞，且返回 −1

描述符可写：

1. 发送缓冲区足够大
2. 写半边已经关闭，如果继续进行写操作将会产生 SIGPIPE 信号
3. 套接字上有错误待处理，使用 write 函数去执行写操作，不阻塞，且返回 −1

poll：
没有文件描述符限制

```
int poll(struct pollfd *fds, unsigned long nfds, int timeout);
返回值：若有就绪描述符则为其数目，若超时则为0，若出错则为-1
struct pollfd {
    int    fd;       /* file descriptor */
    short  events;   /* events to look for */
    short  revents;  /* events returned */
 };
events:
可读事件
```

```
#define POLLIN      0x0001    /* any readable data available */
#define POLLPRI     0x0002    /* OOB/Urgent readable data */
#define POLLRDNORM 0x0040     /* non-OOB/URG data available */
#define POLLRDBAND 0x0080     /* OOB/Urgent readable data */
可写事件
#define POLLOUT     0x0004    /* file descriptor is writeable */
#define POLLWRNORM POLLOUT    /* no write type differentiation */
#define POLLWRBAND 0x0100     /* OOB/Urgent data can be written */
只能通过revents
#define POLLERR     0x0008    /* 一些错误发送 */
#define POLLHUP     0x0010    /* 描述符挂起*/
#define POLLNVAL    0x0020    /* 请求的事件无效*/
```

timeout：如果 <0 的数，表示在有事件发生之前永远等待；如果是 0，表示不阻塞进程，立即返回；如果是一个 >0 的数，表示 poll 调用方等待指定的毫秒数后返回

非阻塞IO：
应用程序阻塞：cpu将时间片给其他进程
非阻塞读： 非阻塞情况下 read 调用会立即返回，一般返回 EWOULDBLOCK 或 EAGAIN 出错信息
非阻塞写：
阻塞write时， 返回字节数和输入的参数总是一样；
非阻塞witre，返回发送缓冲区最大可容纳，立即返回；

操作	系统内核缓冲区状态	阻塞模式	非阻塞模式
read()	接收缓冲区有数据	立即返回	立即返回
	接收缓冲区没有数据	一直等待数据到来	立即返回，带有EWOULDBLOCK或EAGAIN错误
write()	发送缓冲区空闲	全部数据都写入发送缓冲区才返回	能写入多少就写入多少，立即返回
	发送缓冲区不空闲	等待发送缓冲区空闲	立即返回，带有EWOULDBLOCK或EAGAIN错误

非阻塞accept， 规避极端情况，客户端RST情况

epoll：

Number of File Descriptors	poll() CPU time	select() CPU time	epoll() CPU time
10	0.61	0.73	0.41
100	2.9	3	0.42
1000	35	35	0.53
10000	990	930	0.66

epoll
level-triggered 条件触发（水平触发）
edge-triggered 边缘触发 EPOLLET

```
int epoll_create(int size);
int epoll_create1(int flags);
返回值：若成功返回一个大于0的值，表示epoll实例；若返回-1表示出错
```

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
返回值：若成功返回0；若返回-1表示出错
```

```
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
返回值：成功返回的是一个大于0的数，表示事件的个数；返回0表示的是超时时间到；若出错返回-1.
```

epoll_wait 将进程挂起，等待内核IO事件分发

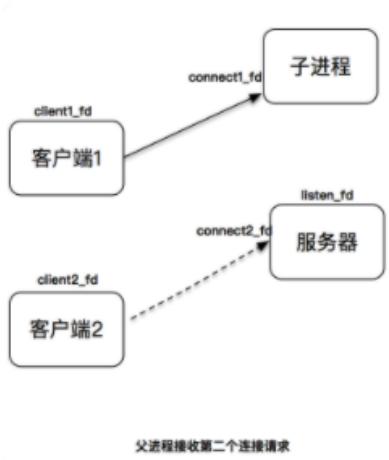
events：返回给用户空间需要处理的 I/O 事件，这是一个数组，数组的大小由 epoll_wait 的返回值决定，这个数组的每个元素都是一个需要待处理的 I/O 事件
maxevents：是一个大于 0 的整数，表示 epoll_wait 可以返回的最大事件值
timeout： epoll_wait 阻塞调用的超时值，如果这个值设置为 -1，表示不超时；如果设置为 0 则立即返回，即使没有任何 I/O 事件发生。

边缘触发：第一次满足时触发
条件触发：只要满足，有数据就触发

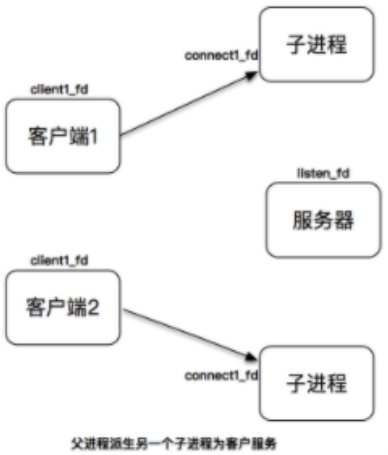
ulimit -n 8192
/etc/sysctl.conf

缓冲区
/proc/sys/net/ipv4/tcp_wmem
/proc/sys/net/ipv4/tcp_rmem

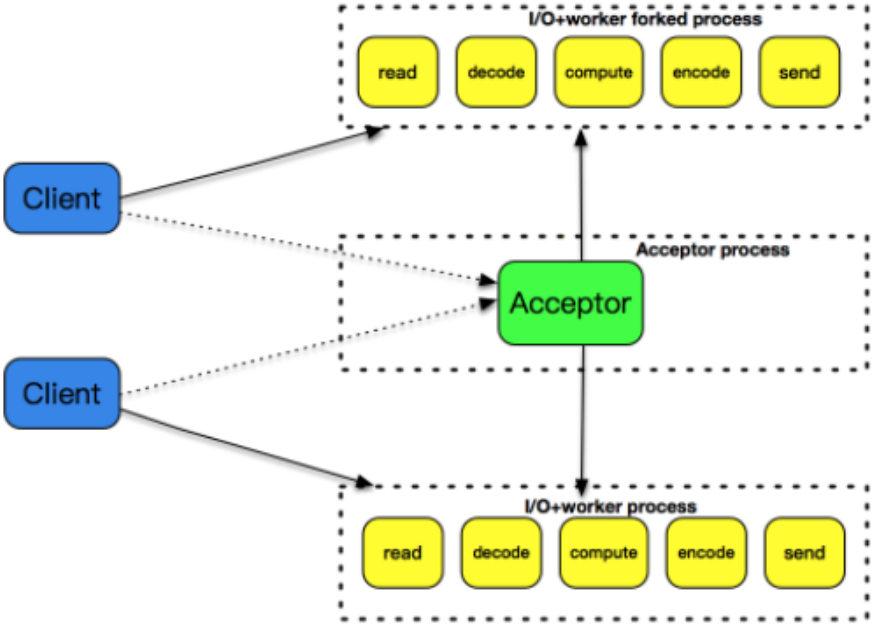
网络模型：
1. 阻塞IO进程模型



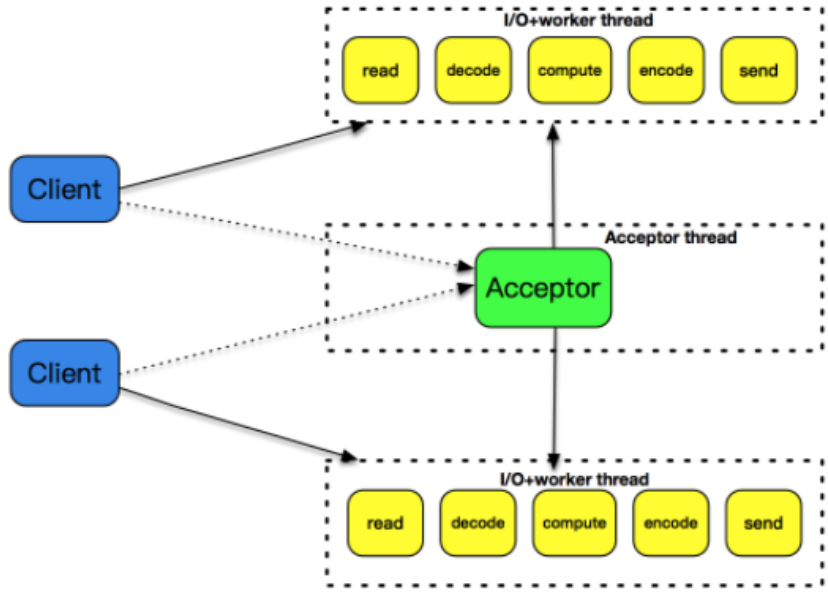
父进程接收第二个连接请求



父进程派生另一个子进程为客户服务



2. 每个连接一个线程处理

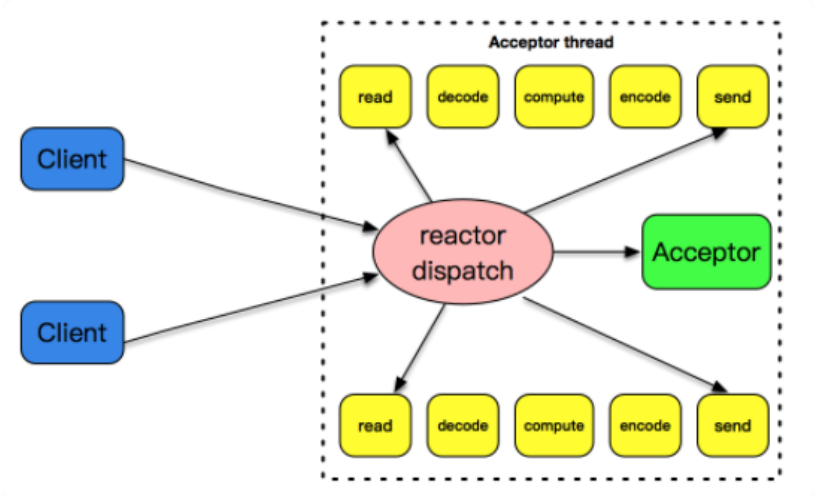


使用线程池，fd队列

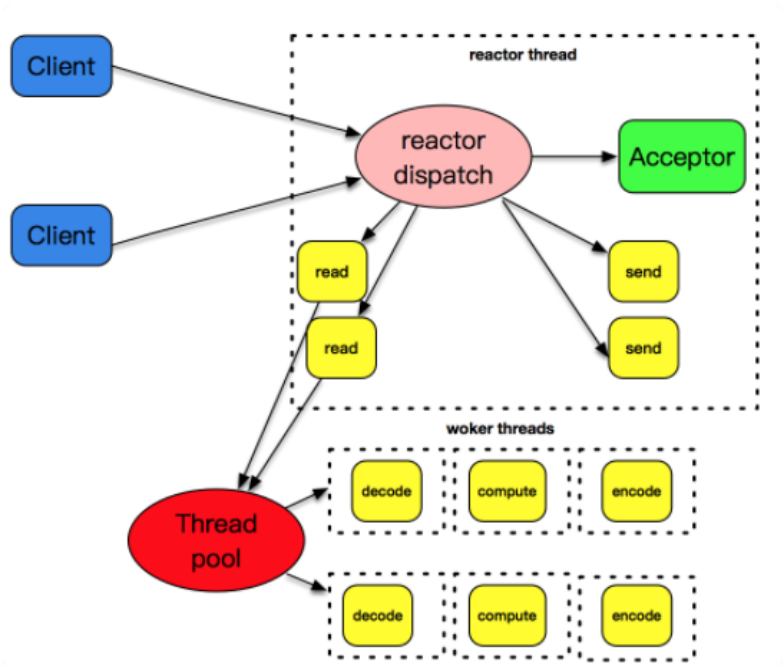
3. 事件驱动（反应堆模型 reactor）EventLoop模型
存在一个无限循环的事件分发线程，reactor线程，EventLoop线程–》epoll
IO操作抽象成事件，每个事件回调函数处理

read：从套接字收取数据；
decode：对收到的数据进行解析；
compute：根据解析之后的内容，进行计算和处理；
encode：将处理之后的结果，按照约定的格式进行编码；
send：最后，通过套接字把结果发送出去。

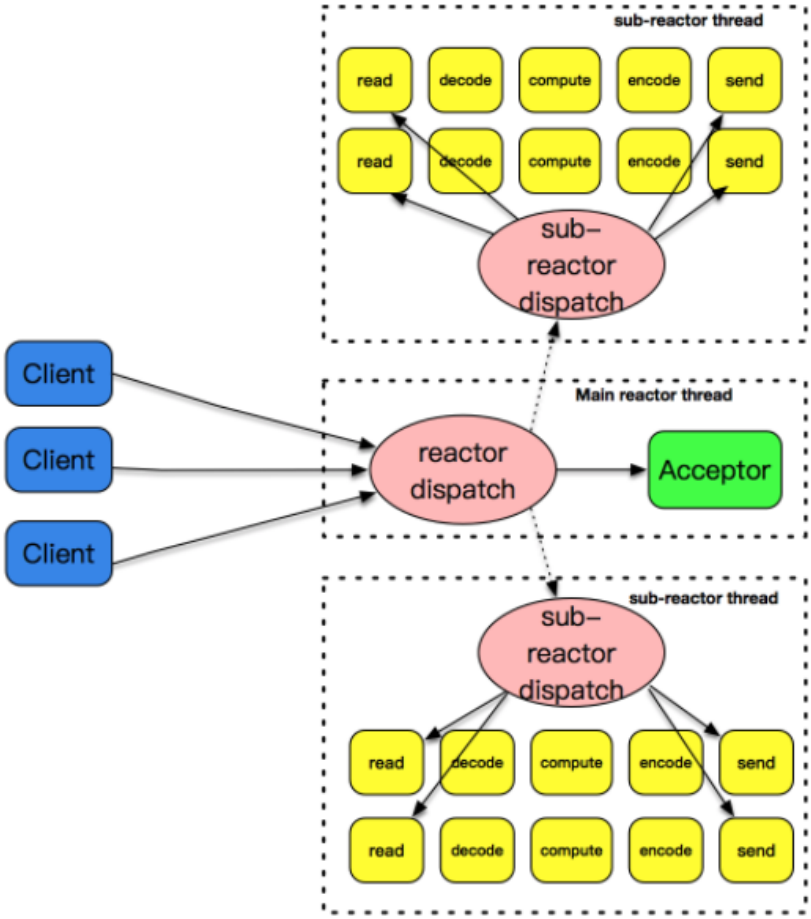
single reactor thread:



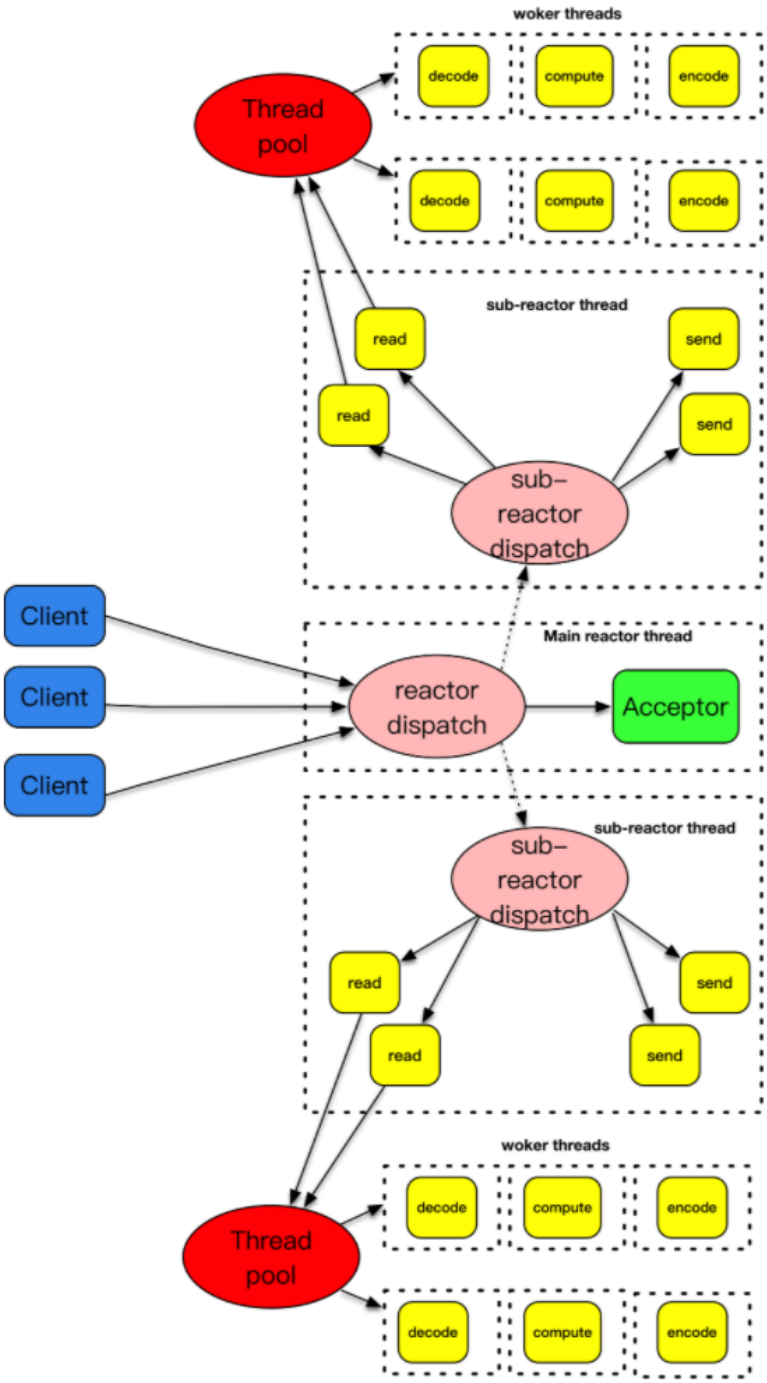
single reactor thread + worker thread:



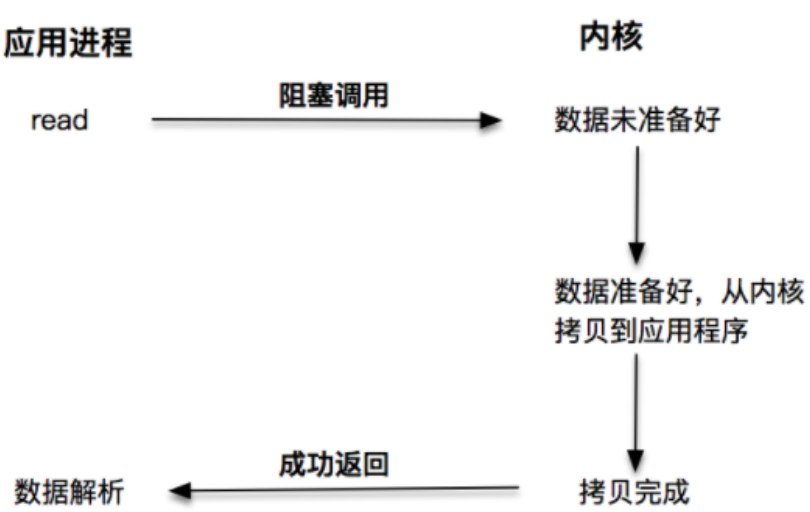
主从reactor：



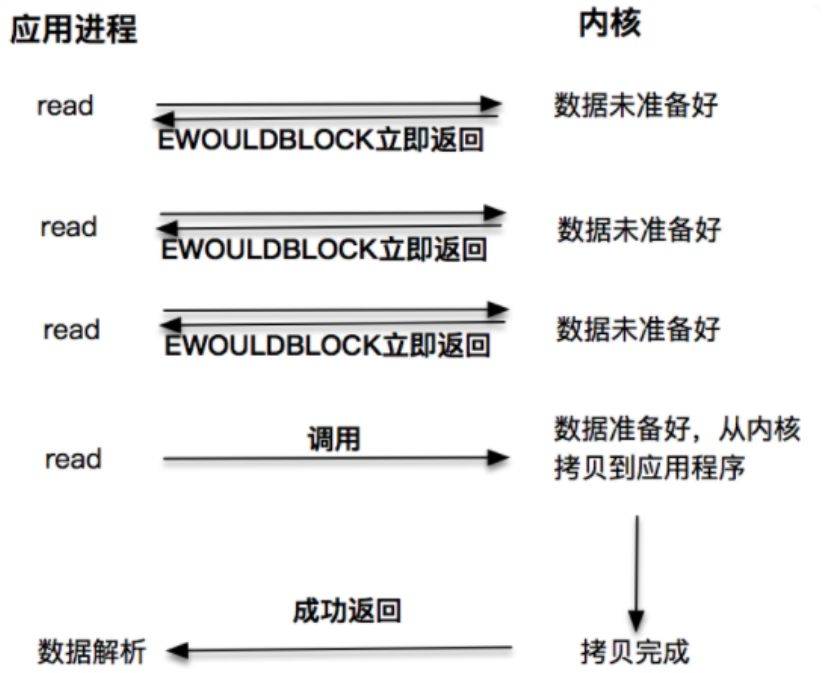
主从reactor–woker threads


woker threads
Thread pool
decode
compute
encode
decode
compute
encode
sub-reactor thread
read
send
read
send
sub–reactor dispatch
Client
Client
Client
Main reactor thread
reactor dispatch
Acceptor
sub-reactor thread
sub–reactor dispatch
read
send
read
send
woker threads
Thread pool
decode
compute
encode
decode
compute
encode

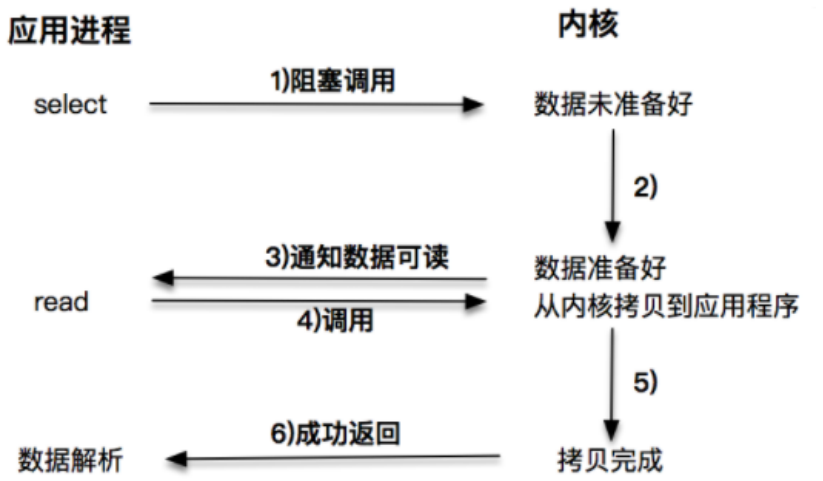
阻塞IO：


应用进程
内核
read
阻塞调用
数据未准备好
数据准备好，从内核
拷贝到应用程序
成功返回
数据解析
拷贝完成

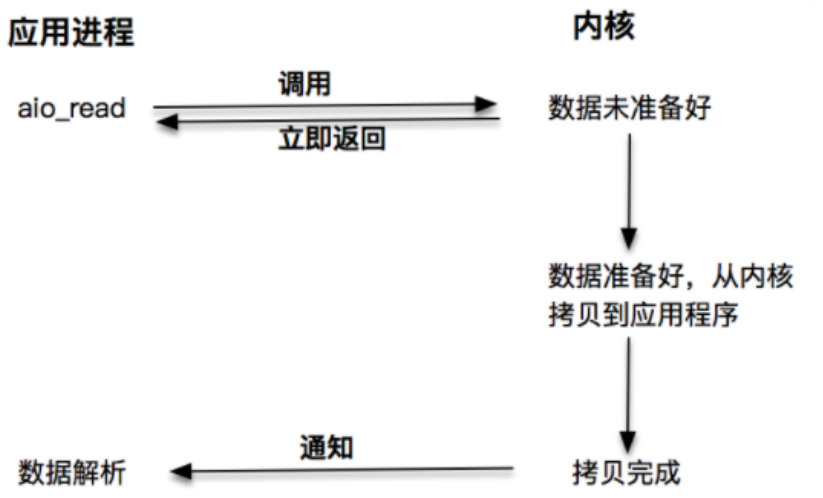
非阻塞IO：



这样每次应用进程轮询，效率较低，于是有了select，poll，epoll，多路复用：



以上最后的read操作都是同步的。



同步I/O	阻塞I/O	非阻塞I/O	select、poll、epoll等多路复用+非阻塞I/O
异步I/O	异步I/O (aio)		

Proactor：
线程并不负责处理 I/O 调用，它只是负责在对应的 read、write 操作完成的情况下，分发完成事件到不同的处理函数。
Reactor 模式是基于待完成的 I/O 事件，而 Proactor 模式则是基于已完成的 I/O 事件

epoll原理：
eventpoll： epoll_create 创建返回的实例，epoll_wait，epoll_ctl对file，private_data操作
epitem：调用 epoll_ctl 增加一个 fd，红黑树的节点
eppoll_entry ： 每个 fd 关联到一个 epoll实例，产生一个entry

EPOLL_CTL_ADD操作：
epoll_ctl 函数通过 epoll 实例句柄来获得对应的匿名文件，epoll_ctl 通过目标文件和对应描述字，在eventpoll 结构体中的红黑树中查找是否存在该套接字
如果发现是一个 ADD 操作，并且在树中没有找到对应的二叉树节点，就会调用 ep_insert 进行二叉树节点的增加，设置回调函数

select poll 将fd从用户拷贝到内核空间，内核空间申请内存，释放内存等过程。
epoll维护红黑树，对红黑树操作，减少内存申请释放过程