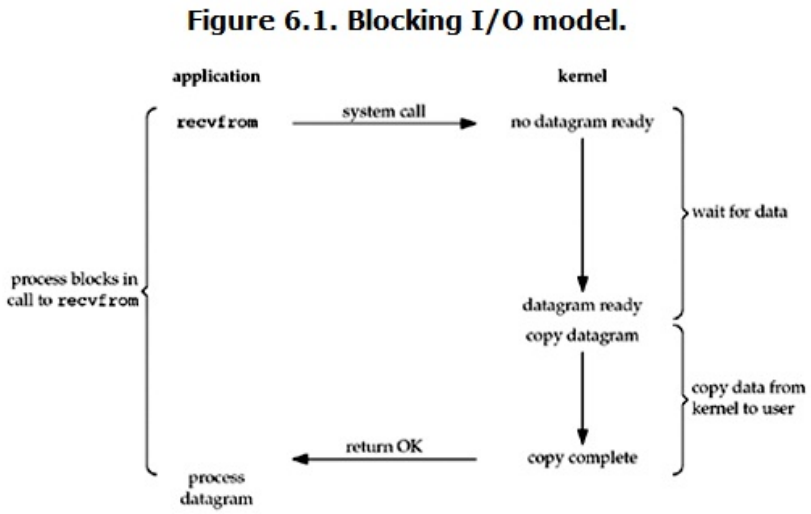
同步、异步、阻塞、非阻塞、IO复用

IO发生时涉及的对象和步骤：以read为例，一个是调用这个IO的process (or thread)，另一个就是系统内核(kernel)。

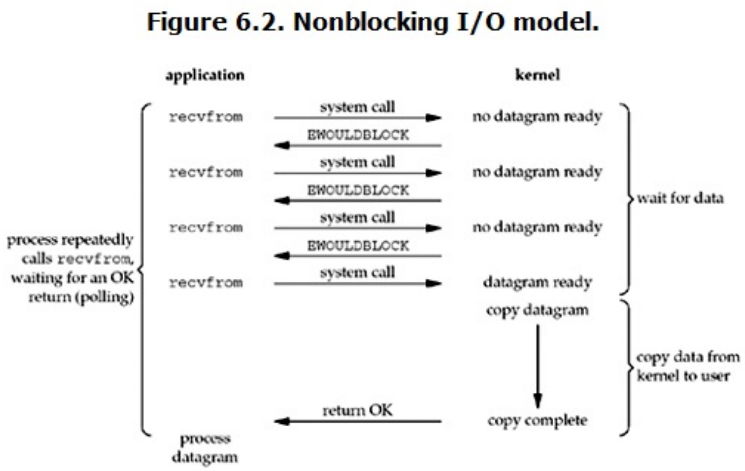
当一个read操作发生时，它会经历两个阶段：

- 等待数据准备
- 将数据从内核拷贝到进程中（同步、异步的关键！如果此步骤由内核完成则为异步；如果为用户完成，如调用recvfrom，则为同步）

阻塞IO：IO执行的两个阶段都是block的

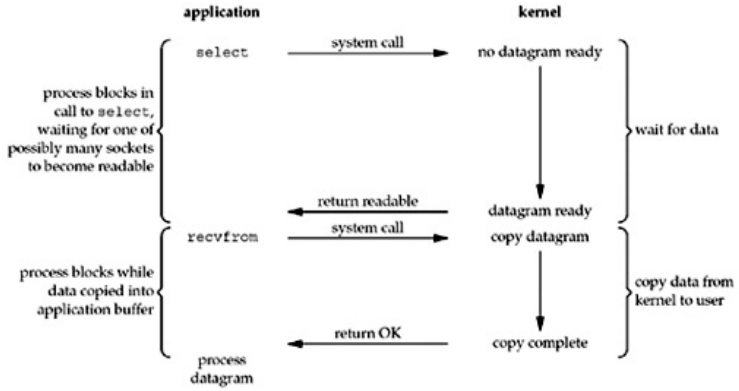


非阻塞IO：进程需要不断地询问内核数据是否准备好，IO执行的第二阶段是block的，属于同步模式



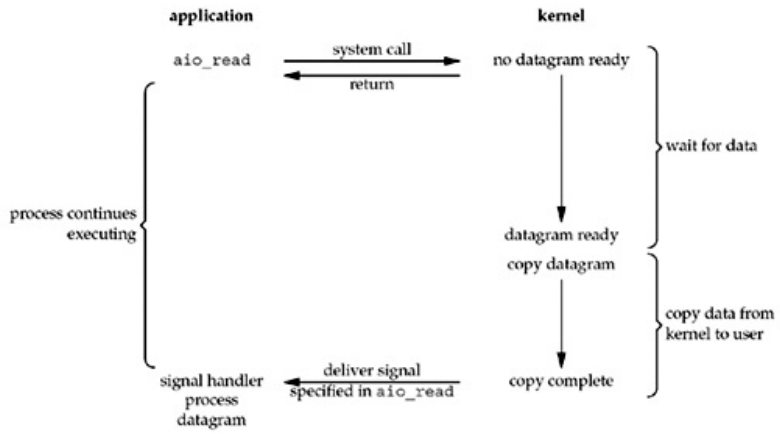
IO复用：单进程可以处理多个IO连接，比如套接字、管道等。IO复用一般和非阻塞一起使用，监听的文件描述符一般设置为非阻塞模式，这是为了避免错误使用阻塞fd，导致阻塞，IO复用只有一个阻塞点，比如说epoll会阻塞在epoll_wait。当有事件发生，通知用户进程处理事件。IO复用中read的第二阶段仍需要用户进程block，故属于同步模式。

Figure 6.3. I/O multiplexing model.



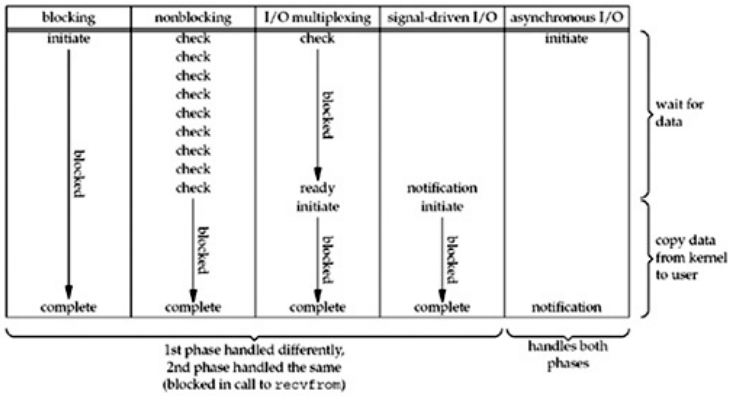
异步IO：应用进程发起异步读取操作后，内核会立即返回，不会阻塞应用进程。等内核的数据准备完毕、内核负责将数据拷贝到用户内存后，通知应用进程操作完毕。异步模式用于实现Proactor模式。

Figure 6.5. Asynchronous I/O model.



各个IO模型的对比：

Figure 6.6. Comparison of the five I/O models.



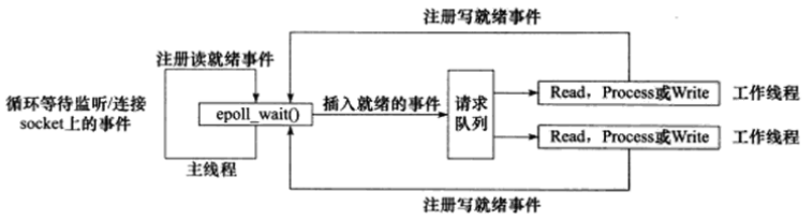
参考文档：IO - 同步，异步，阻塞，非阻塞 （亡羊补牢篇）

Reactor模式和Proactor模式

一般地,I/O多路复用机制都依赖于一个事件多路分离器(Event Demultiplexer)。分离器对象可将来自事件源的I/O事件分离出来，并分发到对应的read/write事件处理器(Event Handler)。开发人员预先注册需要处理的事件及其事件处理器（或回调函数）；事件分离器负责将请求事件传递给事件处理器。两个与事件分离器有关的模式是Reactor和Proactor。

Reactor模式采用同步IO，而Proactor采用异步IO。

Reactor模式：事件分离器负责等待文件描述符或socket为读写操作准备就绪，然后将就绪事件传递给对应的工作线程，最后由工作线程负责完成实际的读写工作。



Proactor模式：工作线程--或者兼任事件分离器，只负责发起异步读写操作。IO操作本身由内核来完成。传递内核的参数需要包括用户定义的数据缓冲区地址和数据大小，内核才能从中得到写出操作所需数据，或写入从socket读到的数据。事件分离器捕获IO操作完成事件，然后将事件传递给对应工作线程。

boost asio实现的proactor，并不是真正意义上的异步IO，底层是用epoll实现的，模拟异步IO的。

TODO：重点说一下Reactor模式

简单理解：

> *reactor：能收了你跟俺说一声。 proactor: 你给我收十个字节，收好了跟俺说一声。*

参考链接：

网络编程：Reactor与Proactor的概念

Proactor和Reactor模型

IO设计模式：Reactor和Proactor对比

问题：

传统的阻塞编程，虽然处理逻辑非常清楚，但是程序常处于阻塞等待模式，CPU资源利用率低。

异步的编程模式，通过设置回调函数的方式进程回调处理。异步编程方式会切割代码逻辑，且过多的回调嵌套会导致代码可读性很差。

能不能想办法“异步编程同步化”，或者说“同步方式编异步的程序”？

参考：

Boost.Asio C++网络编程

C++协程库 libgo

C++11中Async/Await等关键字

TODO

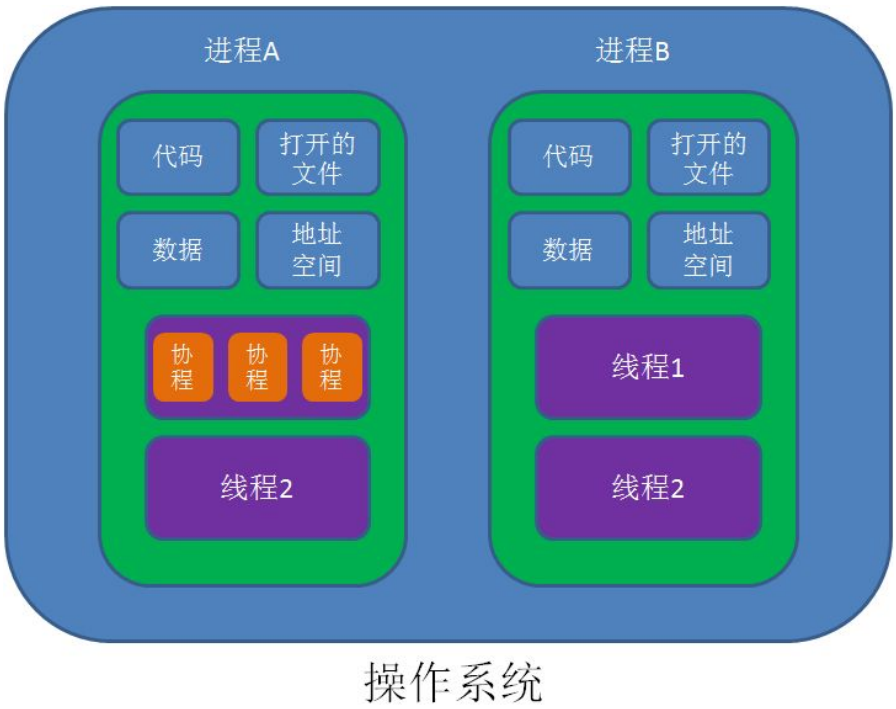
协程

进程和线程存在的痛点：比如说在多线程（多生产者多消费者）中，涉及到条件变量、互斥锁等，会影响性能：

1. 涉及到条件变量、互斥锁，需要陷入内核
2. 涉及到阻塞状态和可运行状态之间的切换
3. 涉及到线程上下文的切换

协程 是一种比线程更加轻量级的存在。

1. 协程在用户态管理、调度，而不是由内核管理，其开销小于线程。
2. 协程切换时机：协程自己让出CPU，协程调度器是不会主动调度的。
3. 1:N模式：即一个线程作为容器，其中防止多个协程。



python举例说明：主协程负责生产、consumer协程负责消费，yield关键字表示协程会在此处“暂停”，直到等待条件满足，才会往下执行。但是yield让协程暂停，和线程的阻塞是有本质区别的。协程的暂停完全由程序控制，线程的阻塞状态是由操作系统内核来进行切换。

```python
def consume():
    while True:
        # consumer协程等待接收数据
        number = yield
        print('开始消费 ', number)


consumer = consume()
# 让初始化状态的consumer协程先执行起来，在yield处停止
next(consumer)
for num in range(0, 100):
    print('开始生产 ', num)
    # 发送数据给consumer协程
    consumer.send(num)
```

协程的适用情景：

适用于IO密集型程序。如果线程中有一个协程是CPU密集型的，他不会主动触发调度器调度，那么就会出现其他协程得不到执行的情况。

参考链接：内核线程与用户线程的一点小总结 《程序员的自我修养》·笔记

Socket选项

level	option name	数据类型	说　明
SOL_SOCKET（通用 socket 选项，协议无关）	SO_DEBUG	int	打开调试信息
	SO_REUSEADDR	int	重用本地地址
	SO_TYPE	int	获取 socket 类型
	SO_ERROR	int	获取并清除 socket 错误状态
	SO_DONTROUTE	int	不查看路由表，直接将数据发送给本地局域网内的主机。含义和 send 系统调用的 MSG_DONTROUTE 标志类似
	SO_RCVBUF	int	TCP 接收缓冲区大小
	SO_SNDBUF	int	TCP 发送缓冲区大小
	SO_KEEPALIVE	int	发送周期性保活报文以维持连接
	SO_OOBINLINE	int	接收到的带外数据将存留在普通数据的输入队列中（在线存留），此时我们不能使用带 MSG_OOB 标志的读操作来读取带外数据（而应该像读取普通数据那样读取带外数据）
	SO_LINGER	linger	若有数据待发送，则延迟关闭
	SO_RCVLOWAT	int	TCP 接收缓存区低水位标记
	SO_SNDLOWAT	int	TCP 发送缓存区低水位标记
	SO_RCVTIMEO	timeval	接收数据超时（见第 11 章）
	SO_SNDTIMEO	timeval	发送数据超时（见第 11 章）
IPPROTO_IP（IPv4 选项）	IP_TOS	int	服务类型
	IP_TTL	int	存活时间
IPPROTO_IPV6（IPv6 选项）	IPV6_NEXTHOP	sockaddr_in6	下一跳 IP 地址
	IPV6_RECVPKTINFO	int	接收分组信息
	IPV6_DONTFRAG	int	禁止分片
	IPV6_RECVTCLASS	int	接收通信类型
IPPROTO_TCP（TCP 选项）	TCP_MAXSEG	int	TCP 最大报文段大小
	TCP_NODELAY	int	禁止 Nagle 算法

https://blog.csdn.net/zs634134578

SO_REUSEADDR和SO_REUSEPORT选项：

1. SO_REUSEADDR允许启动一个监听服务器并捆绑其众所周知端口，即使以前建立的将此端口用做他们的本地端口的连接仍存在。这通常是重启监听服务器时出现，若不设置此选项，则bind时将出错。
2. SO_REUSEADDR允许在同一端口上启动同一服务器的多个实例，只要每个实例捆绑一个不同的本地IP地址即可。对于TCP，我们根本不可能启动捆绑相同IP地址和相同端口号的多个服务器。
3. SO_REUSEADDR允许单个进程捆绑同一端口到多个套接口上，只要每个捆绑指定不同的本地IP地址即可。这一般不用于TCP服务器。
4. SO_REUSEADDR允许完全重复的捆绑：当一个IP地址和端口绑定到某个套接口上时，还允许此IP地址和端口捆绑到另一个套接口上。一般来说，这个特性仅在支持多播的系统上才有，而且只对UDP套接口而言（TCP不支持多播）。

SO_LINGER选项：

SO_LINGER选项用于控制close系统调用在关闭TCP连接时的行为。默认情况下，当我们使用close系统调用来关闭一个socket时，close将立即返回，TCP模块负责把该socket对应的TCP发送缓冲区中残留的数据发送给对方。 设置SO_LINGER选项的值时，我们需要给setsockopt（getsockopt）系统调用传递一个linger类型的结构体，其定义如下：

```
#include <sys/socket.h>

struct linger
{
    int  l_onoff; //开启（非0）还是关闭（0）该选项
    int  l_linger; // 滞留时间
};
```

根据linger结构体中两个成员变量的不同值，close 系统调用可能产生如下3种行为之一：

- l_onoff 等于0。此时SO_LINGER选项不起作用，close用默认行为关闭socket。
- l_onoff 不为0，l_linger等于0. 此时close 系统调用立即返回，TCP模块将丢弃被关闭的socket对应的TCP发送缓冲区中残留的数据，同时给对方一个复位报文段。因此，这种情况给服务器提供了异常终止一个连接的方法。
- l_onoff不为0，l_linger大于0 。此时close的行为取决于两个条件：

  （1）被关闭的socket对应的TCP发送缓冲区中是否还有残留的数据；（2）该socket是阻塞的还是非阻塞的。 对于阻塞的socket，close将等待一段长为l_linger的时间，直到TCP模块发送完所有残留数据并得到对方的确认。如果这段之间内TCP模块没有发送完残留数据并得到对方的确认，那么close系统调用将返回-1并设置errno为EWOULDBLOCK。 如果socket是非阻塞的，close将立即返回，

此时我们需要根据其返回值和errno来判断残留数据是否已经发送完毕。

SO_RCVBUF和SO_SNDBUF选项：

SO_RCVBUF和SO_SNDBUF选项分别表示TCP接收缓冲区和发送缓冲区的大小。不过，当我们用setsockopt来设置TCP的接收缓冲区和发送缓冲区的大小时，系统都会将其值加倍，并且不得小于其个最小值。TCP接收缓冲区的最小值是256字节，而发送缓冲区的最小值是2048字节。

此外，我们可以直接修改内核参数/proc/sys/net/ipv4/tcp_rmem和/proc/sys/net/ipv4/tcp_wmem来强制TCP接收缓冲区和发送缓冲区的大小没有最小值限制。

SO_RCVLOWAT和SO_SNDLOWAT选项

SO_RCVLOWAT和SO_SNDLOWAT选项分别表示TCP接收缓冲区和发送缓冲区的低水位标记。它们一般被I/O复用系统调用，用来判断socket是否可读或可写。

- 当TCP接收缓冲区中可读数据的总数大于其低水位标记时，I/O复用系统调用将通知应用程序可以从对应的socket上读取数据；
- 当TCP发送缓冲区中的空闲空间（可以写入数据的空间）大于其低水位标记时，I/O复用系统调用将通知应用程序可以往对应的socket上写入数据。

默认情况下，TCP接收缓冲区的低水位标记和TCP发送缓冲区的低水位标记均为1字节。

参考链接:Linux 高性能服务器编程——socket选项

网络编程Socket之RST详解

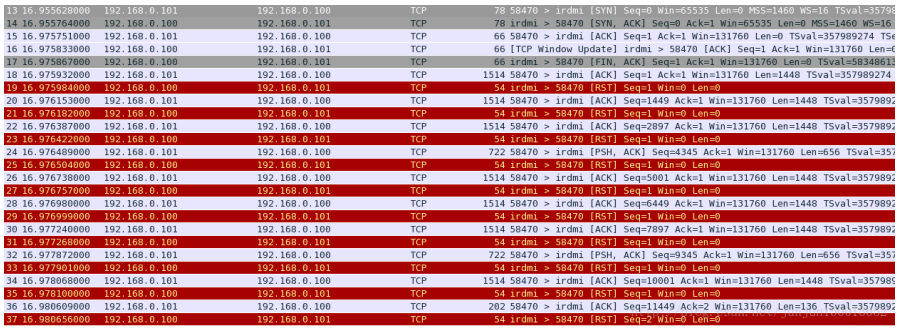
产生RST的三个原因：

1. 三次握手时，SYN达到目标地址的端口，然而在该端口上没有正在监听的服务器
2. TCP一端关闭连接，另一端继续写
3. 服务器重启，收到了重启前的连接的数据

情况1：客户端连接一个不存在的服务器地址

客户端通过connect发起三次握手，发送完第一个SYN分节后，就收到了服务端的RST分节。客户端 `connect() < 0` 。

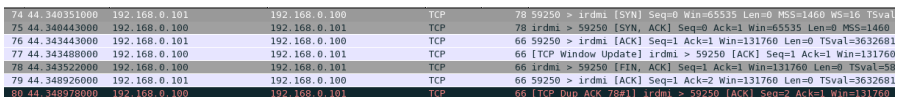
情况2：服务器使用close()关闭连接，客户端继续write()



server和client 已经建立了连接，server调用了close, 发送FIN 段给client，此时server不能再通过socket发送和接收数据，此时client调用read，如果接收到FIN 段会返回0，但client此时还是可以write 给server的，write调用只负责把数据交给TCP发送缓冲区就可以成功返回了，所以不会出错，而server收到数据后应答一个RST段，表示服务器已经不能接收数据，连接重置，client收到RST段后无法立刻通知应用层，只把这个状态保存在TCP协议层。如果client再次调用write发数据给server，由于TCP协议层已经处于RST状态了，因此不会将数据发出，而是发一个SIGPIPE信号给应用层，SIGPIPE信号的缺省处理动作是终止程序。因此进程必须捕获它以免不情愿地被终止；

1. 如果对端TCP发送一个FIN（对端进程终止），那么该套接字变为可读，并且read返回0；
2. 如果对端TCP发送一个RST（对端主机崩溃并重新启动），那么该套接字变为可读，并且read返回-1，而errno中含有确切的错误码（**ECONNRESET**错误）；

情况3：服务器重启，然后接收到重启前的连接的数据，即收到了一个根本不存在的连接上的分节；因为重启后TCP丢失了崩溃前的所有连接信息，因此服务器TCP对于这种情况会响应客户端一个RST。



参考链接：网络编程Socket之RST详解

异常处理

TODO

参考链接：linux编程 异常处理

并发模型：多线程、多进程、Select、Poll、Epoll

多进程/多线程：

这两种模型思想类似，就是让每一个到来的连接都有一个进程/线程来服务。这种模型的代价是它要时间和空间。连接较多时，进程/线程切换的开销比较大。因此这类模型能接受的最大连接数都不会高，一般在几百个左右。

Select的调用过程：

1. 使用copy_from_user从用户空间拷贝fd_set到内核空间
2. 注册回调函数__pollwait
3. 遍历所有fd，调用其对应的poll方法（对于socket，这个poll方法是sock_poll，sock_poll根据情况会调用到tcp_poll,udp_poll或者datagram_poll）
4. 以tcp_poll为例，其核心实现就是__pollwait，也就是上面注册的回调函数。
5. __pollwait的主要工作就是把current（当前进程）挂到设备的等待队列中，不同的设备有不同的等待队列，对于tcp_poll来说，其等待队列是sk->sk_sleep（注意把进程挂到等待队列中并不代表进程已经睡眠了）。在设备收到一条消息（网络设备）或填写完文件数据（磁盘设备）后，会唤醒设备等待队列上睡眠的进程，这时current便被唤醒了。
6. poll方法返回时会返回一个描述读写操作是否就绪的mask掩码，根据这个mask掩码给fd_set赋值。
7. 如果遍历完所有的fd，还没有返回一个可读写的mask掩码，则会调用schedule_timeout是调用select的进程（也就是current）进入睡眠。当设备驱动发生自身资源可读写后，会唤醒其等待队列上睡眠的进程。如果超过一定的超时时间（schedule_timeout指定），还是没人唤醒，则调用select的进程会重新被唤醒获得CPU，进而重新遍历fd，判断有没有就绪的fd。
8. 把fd_set从内核空间拷贝到用户空间。

Select的缺点：

1. 单个进程能够监视的文件描述符的数量存在最大限制（#define __FD_SETSIZE 1024）
2. 每次调用、返回都会有内核与用户空间内存拷贝
3. 每次调用，内核都需要遍历句柄数组，判断是否有事件，是否加入fd回调
4. 返回的是整个句柄数组，应用程序需要遍历整个数组才能判断哪些句柄发生了事件
5. select的触发方式是水平触发，应用程序如果没有完成对一个已经就绪的文件描述符进行IO操作，那么之后每次select调用还是会将这些文件描述符通知进程。

Poll实现：

Poll的实现和Select很相似，但它解决了Select的最大并发数的限制，使用链表保存文件描述符，且其描述fd集合的方式页不同，Poll使用的是pollfd结构体。

Epoll实现：

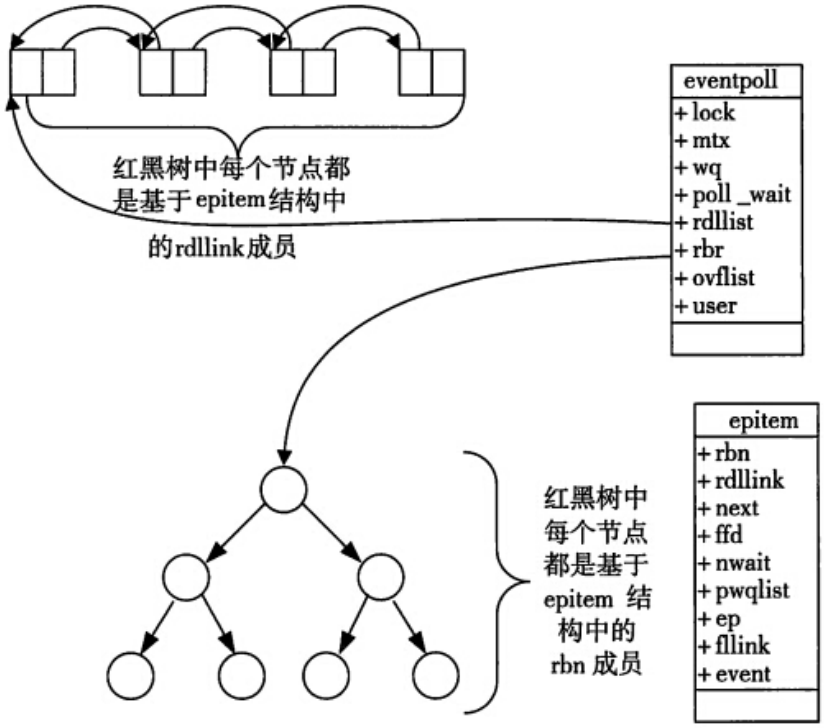
```
struct eventpoll{
    ....
    /*红黑树的根节点，这颗树中存储着所有添加到epoll中的需要监控的事件*/
    struct rb_root  rbr;
    /*双链表中则存放着将要通过epoll_wait返回给用户的满足条件的事件*/
    struct list_head rdlist;
    ....
};

//在epoll中，对于每一个事件，都会建立一个epitem结构体
struct epitem{
    struct rb_node  rbn;//红黑树节点
    struct list_head    rdllink;//双向链表节点
    struct epoll_filefd  ffd;  //事件句柄信息
    struct eventpoll *ep;    //指向其所属的eventpoll对象
    struct epoll_event event; //期待发生的事件类型
};
```

每一个epoll对象都有一个独立的eventpoll结构体，用于存放通过epoll_ctl方法向epoll对象中添加进来的事件。这些事件都会挂载在红黑树中，如此，重复添加的事件就可以通过红黑树而高效的识别出来(红黑树的插入时间效率是lgn，其中n为树的高度)。

而所有添加到epoll中的事件都会与设备(网卡)驱动程序建立回调关系，也就是说，当相应的事件发生时会调用这个回调方法。这个回调方法在内核中叫ep_poll_callback,它会将发生的事件添加到rdlist双链表中。

当调用epoll_wait检查是否有事件发生时，只需要检查eventpoll对象中的rdlist双链表中是否有epitem元素即可。如果rdlist不为空，则把发生的事件复制到用户态，同时将事件数量返回给用户。



Epoll解决了Select/Poll中存在的缺点，具体解决如下：

1. epoll保证了每个fd在整个过程中只会拷贝一次。每次注册新的事件到epoll句柄时，就会把所有的fd拷贝进内核。
2. epoll不需要像select和poll一样，每次都把current轮流加入fd对应的设备等待队列中，而只是在epoll_ctl时将current挂一遍，并为每个fd指定一个回调函数，当设备就绪，唤醒等待队列上的等待者时，就会调用这个回调函数，这个回调函数就会把就绪的fd加入到一个就绪链表中。epoll_wait()的工作实际上就是在这个就绪链表中看看有没有就绪fd。
3. epoll没有fd的限制，所支持fd的上限是最大可以打开文件的数目。

参考链接：

select、poll、epoll之间的区别总结整理

Epoll详解及源码分析

epoll内核源码详解+自己总结的流程

linux 内核poll/select/epoll实现剖析

TODO 在小并发、连接活跃的情况下，为什么Select性能会比Epoll好？

TODO epoll可以监听的文件文件描述符类型：

ET和LT区别：

- LT：水平触发，效率略低于ET触发，但是容易编程。只要有数据没有被获取，内核就不断通知，不需要担心漏包问题。
- ET：边沿触发，效率比较高。但对编程要求比较高，需要避免漏包。

ET漏包问题：缓存区有数据没有读取完，但是不会通知

假如有这样一个例子：

1. 我们已经把一个用来从管道中读取数据的文件句柄(RFD)添加到epoll描述符
2. 这个时候从管道的另一端被写入了2KB的数据
3. 调用epoll_wait(2)，并且它会返回RFD，说明它已经准备好读取操作
4. 然后我们读取了1KB的数据
5. 调用epoll_wait(2)……

ET比LT效率高的原理：减少系统调用来达到提高并行效率 TODO!

注意：ET模式下必须要设置成非阻塞模式，因为ET模式为了避免漏包，需要一直读写，直到出错（EAGAIN）。如果fd为阻塞态，会导致整个进程阻塞。

EAGAIN的含义：

非阻塞模式下，例如，ET模式下连续调用read()，会返回一个错误EAGAIN，提示当前没有数据可读，请稍后重试。这个错误不会破坏socket同步，对于非阻塞socket而言，不算是一种错误。

阻塞模式下recv()返回EAGAIN的可能原因：设置了超时时间

SO_RCVTIMEO和SO_SNDTIMEO会导致read/write函数返回EAGAIN

SO_RCVTIMEO and **SO_SNDTIMEO**Specify the receiving or sending timeouts until reporting an error. The argument is a *struct timeval*. If an input or output function blocks for this period of time, and data has been sent o received, the return value of that function will be the amount of data transferred; if no data has been transferred and the timeout has been reached then -1 is returned with*errno* set to **EAGAIN** or **EWOULDBLOCK**, or **EINPROGRESS** (for **connect**(2)) just as if the socket was specified to be nonblocking. If the timeout is set to zero (the default) then the operation will never timeout. Timeouts only have effect for system calls that perform socket I/O (e.g., **read**(2), **recvmsg**(2), **send**(2), **sendmsg**(2)); timeouts have no effect for **select**(2), **poll**(2), **epoll_wait**(2), and so on.

Muduo使用的是LT：

- ET模式中，它对用户提出了更高的要求，即每次读，必须读到不能再读(出现EAGAIN)，每次写，写到不能再写(出现EAGAIN)。LT模式相对ET在此处可以减少系统调用次数。

- LT模式不会出现漏掉事件的bug。
- LT则简单的多，可以选择也这样做，也可以为编程方便，比如每次只read一次。

Epoll事件：

```
//感兴趣的事件和被触发的事件
struct epoll_event {
    __uint32_t events; /* Epoll events */
    epoll_data_t data; /* User data variable */
};

//保存触发事件的某个文件描述符相关的数据（与具体使用方式有关）
typedef union epoll_data {
    void *ptr;
    int fd;
    __uint32_t u32;
    __uint64_t u64;
} epoll_data_t;

```

events可以是以下几个宏的集合：

> *EPOLLIN ：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）；EPOLLOUT：表示对应的文件描述符可以写，需要时打开，否则会一直触发事件；EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；EPOLLERR：表示对应的文件描述符发生错误；EPOLLHUP：表示对应的文件描述符被挂断；EPOLLET： 将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的。 EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里*

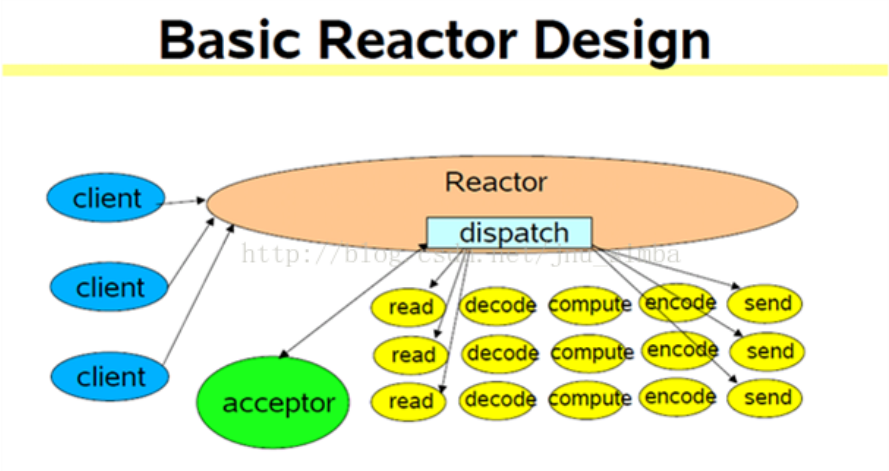
TODO具体解释！！

参考链接：

EPOLLIN , EPOLLOUT , EPOLLPRI, EPOLLERR 和 EPOLLHUP事件

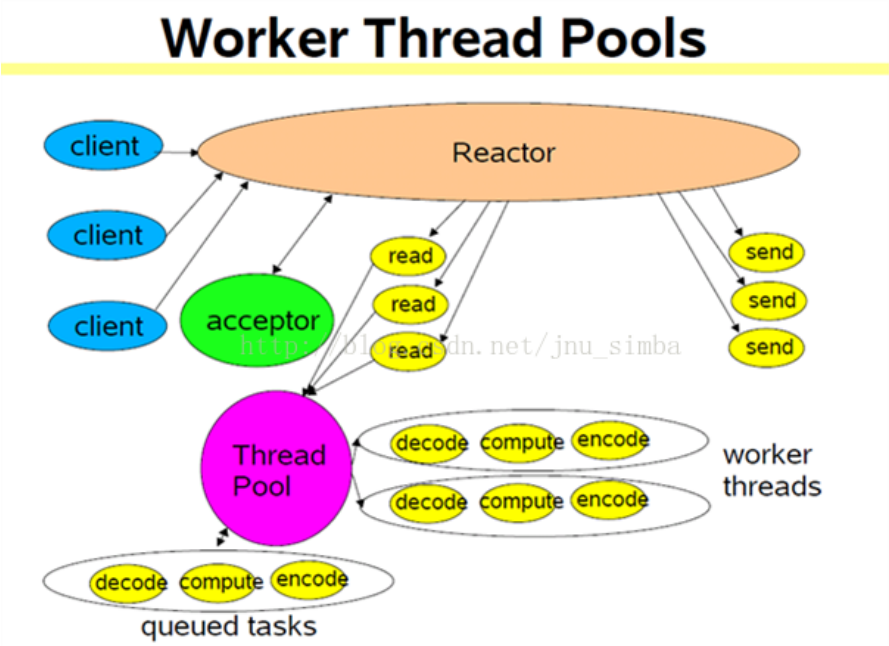
并发服务器模型

one loop



整个程序只有一个线程，触发事件、处理事件。由于事件的处理可能耗费比较多的时间（密集计算、读写日志、读写数据库等），导致时间循环耗时较长、后续事件的处理延迟比较大。

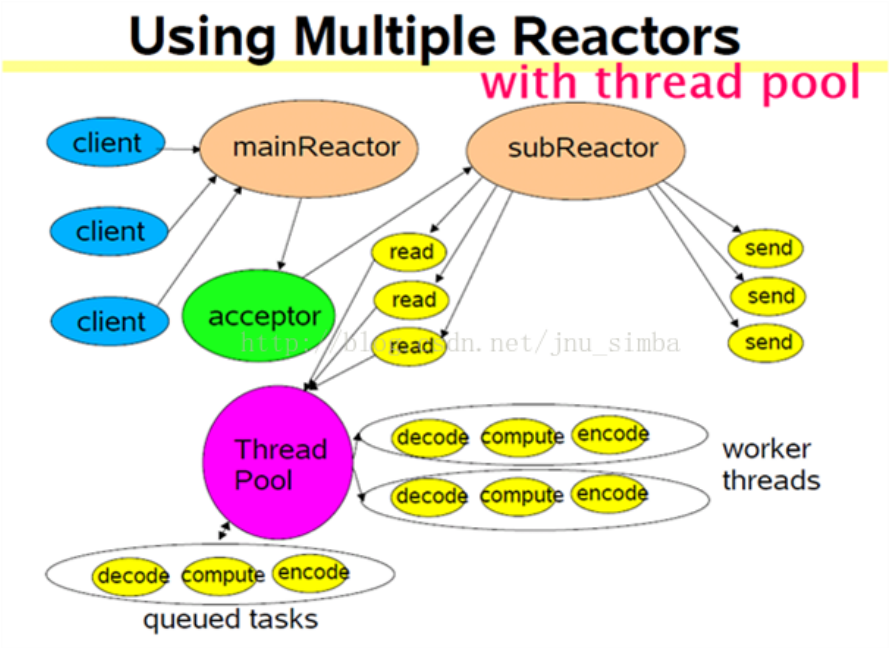
one loop+ thread pool



为了避免处理耗时操作带来的延迟问题，可以将耗时操作（比如写日志、读取数据库等）放置一个线程池中进行。

一般来说在主IO中，做的工作较少：响应事件、read/write、处理一些不耗时的任务。但是如果并发量很大，主IO中轮流处理所有的事件仍会产生较大延迟。

one loop per thread + thread pool



mainReactor只负责处理accept fd的事件（即负责连接的建立），将连接的fd以*Round*-robin的方式加入subReactor中。之后，该fd只有对应的subReactor负责。如何下发fd？如果subReactor正在睡眠呢？见后续eventfd。

常见并发服务器模型比较：

方案	并发模型	UNP 对应	多进程	多线程	阻塞IO	IO复用	长连接	并发性	多核	开销	互通	顺序性	线程数量确定	特点
0	accept+read/write	0	no	no	Y	no	no	无	no	低	no	Y	Y	一次服务一个客户
1	accept+fork	1	Y	no	Y	no	Y	低	Y	高	no	Y	no	process-per-connection
2	accept+thread	6	no	Y	Y	no	Y	中	Y	中	Y	Y	no	thread-per-connection
3	prefork	2/3/4/5	Y	no	Y	no	Y	低	Y	高	no	Y	no	见 UNP
4	pre threaded	7/8	no	Y	Y	no	Y	中	Y	中	Y	Y	no	见 UNP
5	poll (reactor)	sec6.8	no	no	no	Y	Y	高	no	低	Y	Y	Y	单线程 reactor
6	reactor +thread-per-task	无	no	Y	no	Y	Y	中	Y	中	Y	no	no	thread-per-request
7	reactor +worker thread	无	no	Y	no	Y	Y	中	Y	中	Y	Y	no	worker-thread-per-connection
8	reactor +thread poll	无	no	Y	no	Y	Y	高	Y	低	Y	no	Y	主线程 IO，工作线程计算
9	reactors in threads	无	no	Y	no	Y	Y	高	Y	低	Y	Y	Y	one loop per thread
10	reactors in processes	无	Y	no	no	Y	Y	高	Y	低	no	Y	Y	Nginx
11	reactors + thread pool	无	no	Y	no	Y	Y	高	Y	低	Y	no	Y	最灵活的 IO 与 CPU 配置

一般而言，多线程服务器中的线程可分为以下几类：

- IO线程(负责网络IO)
- 计算线程(负责复杂计算)
- 第三方库所用线程

阻抗匹配原则

如果池中线程在执行任务时，密集计算所占的时间比重为P (P是cpu繁忙百分比)，而系统一共有C个CPU，为了让这C个CPU跑满而又不过载，线程池大小为T = C/ P;

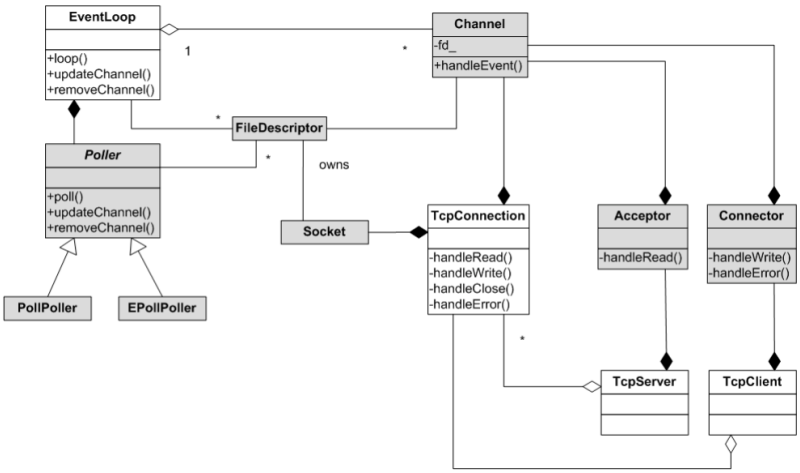
T的大小可以上下浮动0.5.

例如：c = 4, p = 0.5, 则线程池的任务有一半是计算，一半等在IO上。那么T = 8,如果启动更多的线程，并不能提高吞吐量，反而因为增加上下文切换的开销，降低性能。

如果P < 0.2，这个公式就不适用了。T可以取一个固定值，比如说5 * C。

Muduo整体架构

muduo的架构只能说是Reactor的标准实现。



核心结构

程序中的每一个类和结构体当然都必不可少，其中能体现并发模型和整体架构的，我认为是有两个：

· Channel类：Channel是Reactor结构中的“事件”，它自始至终都属于一个EventLoop，负责一个文件描述符的IO事件，在Channel类中保存这IO事件的类型以及对应的回调函数，当IO事件发生时，最终会调用到Channel类中的回调函数。因此，程序中所有带有读写时间的对象都会和一个Channel关联，包括loop中的eventfd，listenfd，HttpData等。

· EventLoop：One loop per thread意味着每个线程只能有一个EventLoop对象，EventLoop即是时间循环，每次从poller里拿活跃事件，并给到Channel里分发处理。EventLoop中的loop函数会在最底层(Thread)中被真正调用，开始无限的循环，直到某一轮的检查到退出状态后从底层一层一层的退出。

TODO

TCP网络编程的本质是处理三个半事件

1. 连接建立：服务器accept接受连接，客户端connect发起连接
2. 连接断开：主动断开（close、shutdown），被动断开（read返回0）
3. 消息到达：文件描述符可写

4. 消息发送完毕：算半个！这里的发送完毕指的是数据写入操作系统缓存区，即write()返回的字节数，将由TCP协议栈负责数据的发送和重传，不代表对方已经接收到数据。

连接维护(针对非阻塞IO)

建立连接

· 建立连接的过程 连接的建立比较简单，server端通过socket()，bind()，listen()，并使用epoll ET模式监听listenfd的读请求，当TCP连接完成3次握手后，会触发listenfd的读事件，应用程序调用accept()，会检查已完成的连接队列，如果队列里有连接，就返回这个连接，出错或连接为空时返回-1。此时，已经可以进行正常的读写操作了。 当然，因为是ET模式，accept()要一直循环到就绪连接为空。

· 分析 之所以说建立连接的过程比较简单，是因为数据的通信已经由操作系统帮我们完成了，这里的通信是指3次握手的过程，这个过程不需要应用程序参与，当应用程序感知到连接时，此时该连接已经完成了3次握手的过程，accept就好了。另一个原因是一般情况下，连接的建立都是client发起的，server端被动建立连接就好了，也不会出现同时建立的情况。

· 限制 假设server只监听一个端口，一个连接就是一个四元组(原ip，原port，对端ip, 对端port)，那么理论上可以建立2^48个连接，可是，fd可没有这么多(操作系统限制、用户进程限制)。当连接满了，如果空等而不连接，那么就绪队列也满了后，会导致新连接无法建立。这里的做法我参考了muduo，准备一个空的文件描述符，accept()后直接close()，这样对端不会收到RST，至少可以知道服务器正在运行。

关闭连接

相对于连接的建立，关闭连接则复杂的多，远不是一个close()那么简单，关闭连接要优雅。

什么时候关闭连接？

通常server和client都可以主动发Fin来关闭连接

对于client(非Keep-Alive)，发送完请求后就可以shutdown()写端，然后收到server发来的应答，最后close掉连接。也可以不shutdown()写，等读完直接close。对于Keep-Alive的情况，就要看client的心情了，收到消息后可以断，也可以不断，server应该保证不主动断开。

对于server端，毫无疑问应该谨慎处理以上所有情况。具体说来:

· 出现各种关于连接的错误时，可以直接close()掉

· 短连接超时的请求，可以close()，也可以不关

· 长连接对方长时间没有请求(如果没有保活机制)，可以close()，也可以不关

· client发出Fin，server会收到0字节，通常不能判断client是close了还是shutdown，这时server应当把消息发完，然后才可以close()，如果对方调用的是close，会收到RST，server能感知到，就可以立即close了

· 短连接正常结束，server可以close，也可以不close，大多数的实现是不close的(对HTTP1.1而言)

EPOLLIN触发但是read()返回0的情况

这种情况通常有两个原因:

· 对端已经关闭了连接，这时再写该fd会出错，此时应该关闭连接

· 对端只是shutdown()了写端，告诉server我已经写完了，但是还可以接收信息。server应该在写完所有的信息后再关闭连接。更优雅的做法是透明的传递这个行为，即server顺着关闭读端，然后发完数据后关闭

Buffer的实现

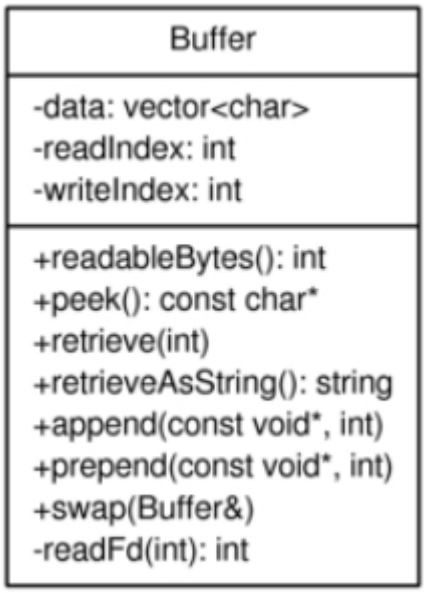
应用层Buff的必要性：

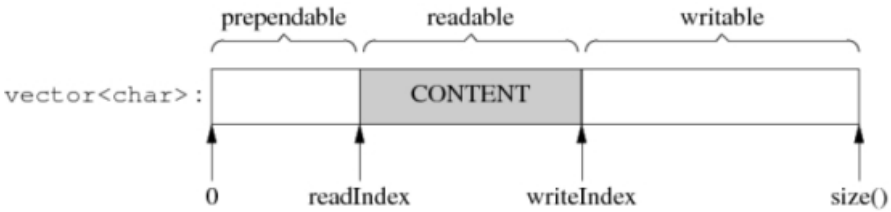
1、发送的数据一次没有发送完毕：一次write发送的数据量有上限，如果一次尚未将数据发送完毕，又来了另外一次write，会导致前一个包发送不完整，顺序不一致，故需要发送缓冲区，保证数据发送顺序、完整

2、收到的数据不构成一条完整的消息。（在读取数据的时候，一次读取的数据有上限，一般来说，应用层自定义包，包头中包含本数据包的大小，一次读取的数据可能不满足一个完整包，故需要放到一个输入缓冲区中，当收到一个完整包之后，再将其取出来，返回给上层。）

线程安全性：

对于buffer的操作都在用一个线程中。muduo中读写操作只发生在IO线程中（即loop中）。





自动增长：

append()添加数据，使用vector自动增长机制，即以2^k增长。

内部腾挪：

经过若干次读写，readIndex移到了比较靠后的位置，留下了巨大的prependable空间，可以将数据移动到前面。如果重新分配内存，反正也是要将数据拷贝到新分配的内存区域，代价会更大。

腾挪时机：

```
void append(const char* /*restrict*/ data, size_t len)
{
  ensureWritableBytes(len);
  std::copy(data, data+len, beginWrite());
  hasWritten(len);
}
```

```
void ensureWritableBytes(size_t len)
{
  if (writableBytes() < len)
  {
    makeSpace(len);
  }
  assert(writableBytes() >= len);
}
```

```
void makeSpace(size_t len)
{
  if (writableBytes() + prependableBytes() < len + kCheapPrepend)
  {
    // FIXME: move readable data
    buffer_.resize(writerIndex_+len);
  }
  else
  {
    // move readable data to the front, make space inside buffer
    assert(kCheapPrepend < readerIndex_);
    size_t readable = readableBytes();
    std::copy(begin()+readerIndex_,
              begin()+writerIndex_,
              begin()+kCheapPrepend);
    readerIndex_ = kCheapPrepend;
    writerIndex_ = readerIndex_ + readable;
    assert(readable == readableBytes());
  }
} « end makeSpace »
```

前方添加：

预留了prependable字段，让程序可以以很低的代价在数据前面添加几个字节。

readFd():读取socket数据

```
ssize_t Buffer::readFd(int fd, int* savedErrno)
{
  // saved an ioctl()/FIONREAD call to tell how much to read
  char extrabuf[65536];
  struct iovec vec[2];
  const size_t writable = writableBytes();
  vec[0].iov_base = begin()+writerIndex_;
  vec[0].iov_len = writable;
  vec[1].iov_base = extrabuf;
  vec[1].iov_len = sizeof extrabuf;
  // when there is enough space in this buffer, don't read into extrabuf.
  // when extrabuf is used, we read 128k-1 bytes at most.
  const int iovcnt = (writable < sizeof extrabuf) ? 2 : 1;
  const ssize_t n = sockets::readv(fd, vec, iovcnt);
  if (n < 0)
  {
    *savedErrno = errno;
  }
  else if (implicit_cast<size_t>(n) <= writable)
  {
    writerIndex_ += n;
  }
  else
  {
    writerIndex_ = buffer_.size();
    append(extrabuf, n - writable);
  }
  // if (n == writable + sizeof extrabuf)
  // {
  //   goto line_30;
  // }
  return n;
} « end readFd »
```

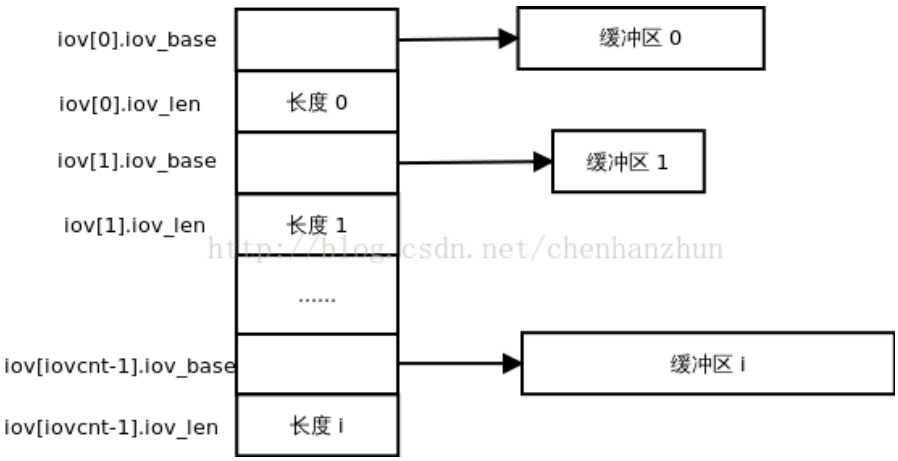
利用了栈上空间，避免了每个连接初始Buffer过大造成的内存浪费，也可以避免反复调用read()的系统开销。

readv/writev

```
/* 读、写多个非连续的缓冲区 */

/*
 * 函数功能：读取数据到多个非连续的缓冲区，或从多个非连续缓冲区写数据到文件；
 * 返回值：若成功则返回已读、写的字节数，若出错则返回-1；
 * 函数原型：
 */
#include <sys/uio.h>
ssize_t readv(int filedes, const struct iovec *iov, int iovcnt);
ssize_t writev(int filedes, const struct iovec *iov, int iovcnt);
/*
 * 说明：
 * iovec的指针结构如下：
 */
struct iovec
{
    void *iov_base;     /* starting address of buffer */
    size_t iov_len;     /* size of buffer */
};
```

下图说明了 readv 和 writev 的参数和 iovec 结构：



http://blog.csdn.net/chenhanzhun

- writev 以顺序 iov[0]，iov[1] 至 iov[iovcnt-1] 从缓冲区中聚集输出数据。writev 返回输出的字节总数。

- readv 则将读入的数据按照上述同样顺序散布到缓冲区中，readv 总是先填满一个缓冲区，然后再填写下一个。readv 返回读到的总字节数。如果遇到文件结尾，已无数据可读，则返回0。

Buffer收缩问题：多线程向一个conn发送大量数据，但conn发送不过来，导致数据会大量堆积，Buffer会不断扩容，但Buffer未提供有效的自动收缩算法。

线程池实现

互斥锁实现的线程池：

（略）见muduo源码，很标注的做法。

本版本线程池中使用条件变量、互斥锁维护任务队列，当多线程访问该任务队列时，如果线程没有得到锁资源，则会陷入内核，执行进程执行状态的切换。代价比较高！

CAS无锁队列实现的线程池：

此处主要介绍CAS、如何实现无锁队列、ABA问题，以及解决措施

> *CAS是英文单词**Compare And Swap**的缩写。*

```
//内存地址dest，旧的预期值accum，要修改的新值newval。
//更新一个变量的时候，只有当变量的预期值A和内存地址V当中的实际值相同时，才会将内存地址V对应的值修改为B。
bool compare_and_swap (int* accum, int* dest, int newval)
{
  if( *accum == *dest ) {
      *dest = newval;
      return true;
  }
  return false;
}
```

CAS队列伪代码：

```
EnQueue(x)//进队列改良版
{
    q = newrecord();
    q->value = x;
    q->next = NULL;
```

```
6
7       p = tail;
8       oldp = p
9       do{
10          //保证拿到尾部节点，因为其他线程可能会修改尾部节点
11          while(p->next != NULL)
12              p = p->next;
13      }while( CAS(p.next, NULL, q) != TRUE); //自旋
14
15      CAS(tail, oldp, q); //置尾结点
16  }
```

```
1   DeQueue()//出队列
2   {
3       do{
4           p = head;
5           if(p->next == NULL){
6               return ERR_EMPTY_QUEUE;
7           }
8       while( CAS(head, p, p->next) != TRUE );
9       return p->next->value;
10  }
```

CAS属于乐观锁，它认为程序中并发情况不那么严重，会让程序不断尝试更新； 互斥锁属于悲观锁，它认为程序后中并发很严重，不断尝试代价会很高，所以陷入内核进行切换。

CAS适用场景：

1. 并发量不太高的情况。如果并发量很高，使用同步锁更合适。
2. Atomic、Lock等底层实现使用的就是CAS

CAS存在的问题：

1. CPU开销比较大，如果更新失败，会反复尝试。
2. 不能保证代码块的原子性，只能保证一个变量的原子性操作。
3. ABA问题

ABA问题：

结合实际应用场景，举例：

举一个提款机的例子。假设有一个遵循CAS原理的提款机，小灰有100元存款，要用这个提款机来提款50元。由于提款机硬件出了点小问题，小灰的提款操作被同时提交两次，开启了两个线程，两个线程都是获取当前值100元，要更新成50元。理想情况下，应该一个线程更新成功，另一个线程更新失败，小灰的存款只被扣一次。

存款余额：100元

线程1(提款机)：　获取当前值100，期望更新为50
线程2(提款机)：　获取当前值100，期望更新为50

线程2获取完当前值后，阻塞了。小灰妈刚好给小灰汇款50元。

存款余额：50元

线程1(提款机)：　获取当前值100，**成功更新为50**
线程2(提款机)：　获取当前值100，期望更新为50，**BLOCK**
线程3(小灰妈)：　**获取当前值50，期望更新为100**

线程2仍然是阻塞状态，线程3执行成功，把余额从50改成100。

存款余额：100元

线程1(提款机)：　获取当前值100，成功更新为50，已返回
线程2(提款机)：　获取当前值100，期望更新为50，BLOCK
线程3(小灰妈)：　获取当前值50，**成功更新为100**

线程2恢复运行，由于阻塞之前已经获得了“当前值”100，并且经过compare检测，此时存款实际值也是100，所以成功把变量值100更新成了50。

线程1(提款机)：　获取当前值100，成功更新为50，已返回
线程2(提款机)：　获取“当前值” 100，**成功更新为50**
线程3(小灰妈)：　获取当前值50，成功更新为100 ，已返回

这就导致错误！！！

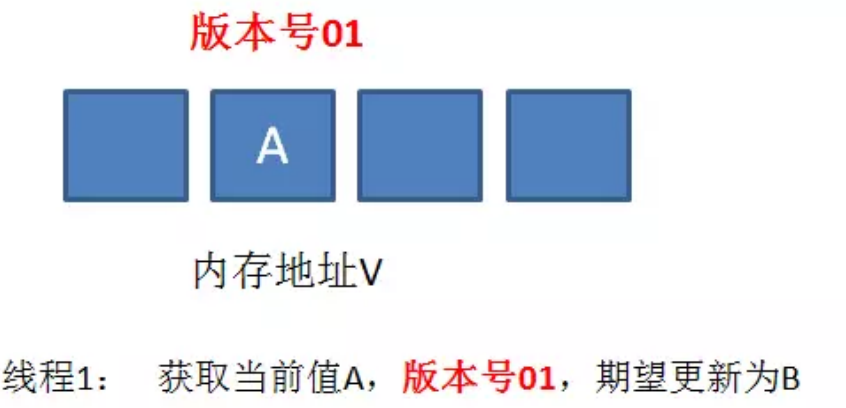
ABA问题的Wiki描述：

1. 进程P1在共享变量中读到值为A
2. P1被抢占了，进程P2执行
3. P2把共享变量里的值从A改成了B，再改回到A，此时被P1抢占。
4. P1回来看到共享变量里的值没有被改变，于是继续执行。

ABA解决措施：

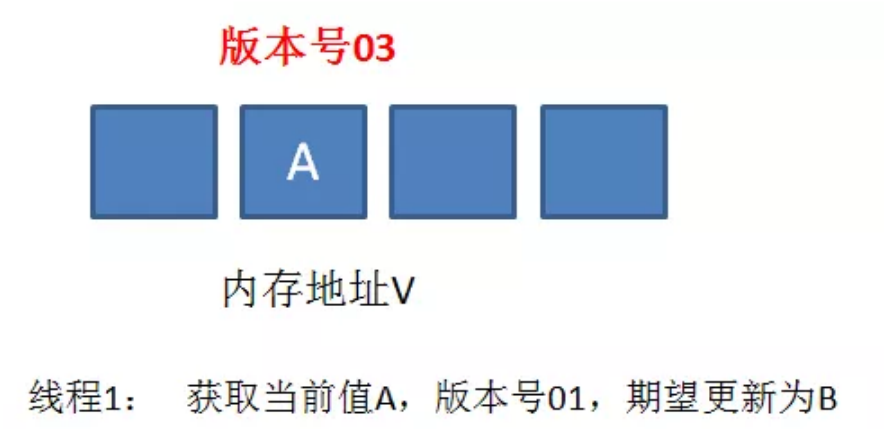
引入版本号，在Compare阶段，不仅要比较期望值A和地址V中的实际值，还要比较变量的版本号是否一致。

假设地址V中存储着变量值A，当前版本号是01。线程1获得了当前值A和版本号01，想要更新为B，但是被阻塞了。

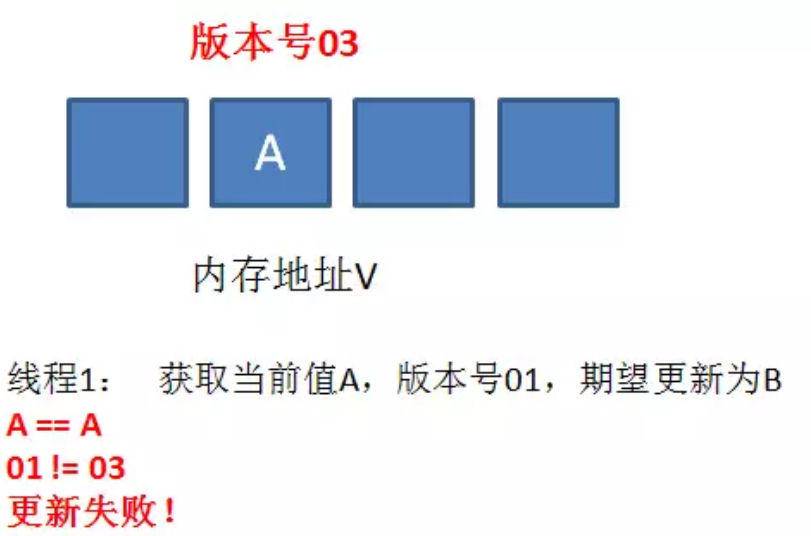


线程1：　获取当前值A，**版本号01**，期望更新为B

这时候，内存地址V中的变量发生了多次改变，版本号提升为03，但是变量值仍然是A。



随后线程1恢复运行，进行Compare操作。经过比较，线程1所获得的值和地址V的实际值都是A，但是版本号不相等，所以这一次更新失败。



参考链接：

利用CAS操作（Compare & Set）实现无锁队列

漫画：什么是CAS机制？（进阶篇）

定时器实现

服务端程序设计中，与时间相关的常见任务：

- 获取当前时间，计算时间间隔
- 时区转换与日期计算
- 定时操作

Linux中的计时函数，用于获得当前时间：

- time(2)/time_t（秒）：精度不够
- ftime(3)/struct timeb(毫秒)：被启用
- gettimeofday(2)/struct timeval(微秒)：推荐！用户态实现的，没有上下文切换和陷入内核的开销
- clock_gettime(2)/struct timespec(纳秒)，内核态实现，开销较大
- clock(时钟滴答数)

PS：遇到过的问题，多线程中使用clock计算时间间隔 发现时间比时间运行时间长，其他的计时函数可能也会有：

```
clock()函数的功能：这个函数返回从“开启这个程序进程”到“程序中调用C++ clock()函数”时之间的CPU时钟计时单元（clock tick）数当程序单线程或者单核心机器运行时，这种时间的统计方法是正确的。但是如果要执行的代码多个线程并发执行时就会出问题，因为最终end-begin将会是多个核心总共执行的时钟嘀嗒数，因此造成时间偏大。
```

定时函数：

- sleep(3)、alarm(2)、usleep(3)：使用了SIGALRM信号，多线程程序中处理信号很麻烦，应该尽量避免；会导致程序睡眠
- nanosleep(2)和clock_nanosleep(2)：会导致睡眠
- getitimer(2)/setitimer(2)
- timer_create(2)/timer_settime(2)/timer_gettime(2)/timer_delete(2)
- timerfd_create(2)/timerfd_gettime(2)/timerfd_settime(2)：推荐！
- epoll_wait、select中超时时间精度为毫秒

Linux中在用户态实现完全精确可控的计时和定时是做不到的，因为当前任务可能会被随时切换出去，这在CPU负载大的时候尤其明显。

3种常见的定时器实现方案：

- 排序链表，插入算法复杂度O(n)
- 最小堆，插入算法复杂度O(lgn)
- 时间轮，插入算法复杂度O(1)

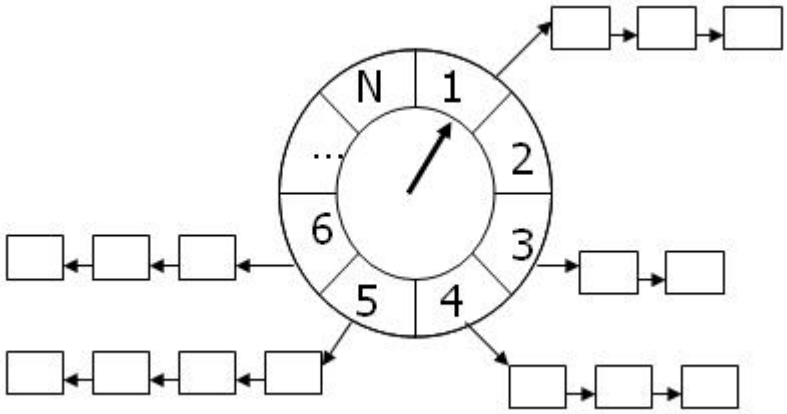
在muduo中使用最小堆+timerfd实现定时器，使用时间轮+心跳包 处理超时请求和长时间不活跃的连接。

最小堆：muduo中介绍了时间轮的实现和用stl里set的实现，这里我的实现直接使用了stl里的priority_queue，底层是小根堆，并采用惰性删除的方式，时间的到来不会唤醒线程，而是每次循环的最后进行检查，如果超时了再删，因为这里对超时的要求并不会很高，如果此时线程忙，那么检查时间队列的间隔也会短，如果不忙，也给了超时请求更长的等待时间。

muduo定时器详细介绍 TODO

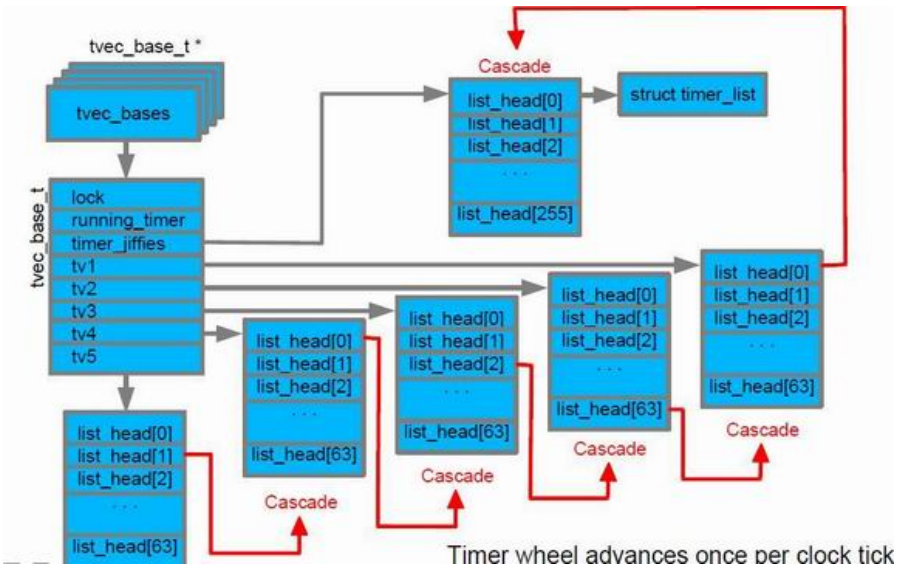
简单时间轮：一个齿轮，每个齿轮保存一个超时的node链表。一个齿轮表示一个时间刻度，比如钟表里面一小格代表一秒，钟表的秒针每次跳一格。假设一个刻度代表10ms，则2^32 个格子可表示1.36年，2^16个格子可表示10.9分钟。当要表示的时间范围较大时，空间复杂度会大幅增加。

PS：muduo使用简单时间轮管理超时链接。



多轮时间轮：类似于水表，当小轮子里的指针转动满一圈后，上一级轮子的指针进一格。 采用五个轮子每个轮子为一个简单时间轮，大小分别为 2^8， 2^6， 2^6， 2^6， 2^6，所需空间：2^8 + 2^6 + 2^6 + 2^6 + 2^6 = 512， 可表示的范围为 0 -- 2^8 * 2^6 * 2^6* 2^6* 2^6 = 2^32 。

Linux底层的定时器实现便是基于此。下图为引用的linux内核定时器的实现。



linux2.6.16之前，采用的就是多轮时间轮的做法。它只支持低精度的时钟：

- 系统启动后，会读取时钟源设备(RTC, HPET，PIT...),初始化当前系统时间；

- 内核会根据HZ(**系统定时器频率，节拍率**)参数值，设置时钟事件设备，启动tick(**节拍**)中断。HZ表示1秒种产生多少个时钟硬件中断，tick就表示连续两个中断的间隔时间。
- 设置时钟事件设备后，时钟事件设备会定时产生一个tick中断，触发时钟中断处理函数，更新系统时钟,并检测timer wheel，进行超时事件的处理。

linux2.6.16 ，内核支持了高精度的时钟，内核采用新的定时器hrtimer，其实现逻辑和Linux 2.6.16 之前定时器逻辑区别：

- hrtimer采用红黑树进行高精度定时器的管理，而不是时间轮；
- 高精度时钟定时器不在依赖系统的tick中断，而是基于事件触发。

新内核定时器框架采用了基于事件触发，而不是以前的周期性触发。新内核实现了hrtimer(high resolution timer)，hrtimer的设计目的，就是为了解决time wheel的缺点：

- timer wheel只能支持ms级别的精度，hrtimer可以支持ns级别；
- Timer wheel与内核其他模块的高耦合性，比如说timerfd作为epoll监听fd

hrtimer的工作原理：通过将高精度时钟硬件的下次中断触发时间设置为红黑树中最早到期的 Timer 的时间，时钟到期后从红黑树中得到下一个 Timer 的到期时间，并设置硬件，如此循环反复。 在高精度时钟模式下，操作系统内核仍然需要周期性的tick中断，以便刷新内核的一些任务。前面可以知道，hrtimer是基于事件的，不会周期性出发tick中断，所以为了实现周期性的tick中断(dynamic tick)：系统创建了一个模拟 tick 时钟的特殊 hrtimer，将其超时时间设置为一个tick时长，在超时回来后，完成对应的工作，然后再次设置下一个tick的超时时间，以此达到周期性tick中断的需求。 引入了dynamic tick,是为了能够在使用高精度时钟的同时节约能源，这样会产生tickless 情况下，会跳过一些 tick。

参考链接：

Linux内核时钟系统和定时器实现

让事件飞 ——Linux eventfd 原理与实践

跨线程的事件通知机制的实现

eventfd用于实现跨线程的事件通知机制。

eventfd的使用：

```
1  int eventfd(unsigned int initval, int flags);
```

创建一个eventfd对象，或者说打开一个eventfd的文件，类似普通文件的open操作。

该对象是一个内核维护的无符号的64位整型计数器。初始化为initval的值。

flags可以以下三个标志位的OR结果：

- EFD_CLOEXEC：FD_CLOEXEC，简单说就是fork子进程时不继承，对于多线程的程序设上这个值不会有错的。
- EFD_NONBLOCK：文件会被设置成O_NONBLOCK，一般要设置。
- EFD_SEMAPHORE：（2.6.30以后支持）支持semophore语义的read，简单说就值递减1。

这个新建的fd的操作很简单：

- read(): 读操作就是将counter值置0，如果是semophore就减1。
- write(): 设置counter的值。

eventfd VS pipefd

eventfd是Linux在后版本中才提出来的一种IPC通讯方式，不同进程可以通过eventfd机制建立起一个共享的计数器，这个计数器由内核负责维护，充当了信息的角色，与它关联的进程可以对其进行读写，从而起到进程间通讯的目的。它更加轻量级，也更加灵活，它避免了pipe必须一端读一端写这个弊端，而且只创建了一个文件描述符，pipe必须创建读写两个文件描述符。eventfd的性能要比pipe要更好。

举例说明：

muduo中 主IO阻塞在Epoll::loop()的epoll_wait()上，但计算线程池中需要将计算的结果通过主IO发送，此时涉及到跨线程调用，如何发送数据？如何唤醒主IO?

```cpp
void EventLoop::loop()
{
  assert(!looping_);
  assertInLoopThread();
  looping_ = true;
  quit_ = false;  // FIXME: what if someone calls quit() before loop() ?
  LOG_TRACE << "EventLoop " << this << " start looping";

  while (!quit_)
  {
    activeChannels_.clear();
    pollReturnTime_ = poller_->poll(kPollTimeMs, &activeChannels_);
    ++iteration_;
    if (Logger::logLevel() <= Logger::TRACE)
    {
      printActiveChannels();
    }
    // TODO sort channel by priority
    eventHandling_ = true;
    for (ChannelList::iterator it = activeChannels_.begin();
        it != activeChannels_.end(); ++it)
    {
      currentActiveChannel_ = *it;
      currentActiveChannel_->handleEvent(pollReturnTime_);
    }
    currentActiveChannel_ = NULL;
    eventHandling_ = false;
    doPendingFunctors();
  } « end while !quit_ »

  LOG_TRACE << "EventLoop " << this << " stop looping";
  looping_ = false;
} « end loop »
```

```cpp
void EventLoop::runInLoop(const Functor& cb)
{
  //判断调用线程是否是loop线程，如果是直接执行；否则加入pendingFunctors
  if (isInLoopThread())
  {
    cb();
  }
  else
  {
    queueInLoop(cb);
  }
}
```

```cpp
void EventLoop::queueInLoop(const Functor& cb)
{
  //加入队列中
  {
  MutexLockGuard lock(mutex_);
  pendingFunctors_.push_back(cb);
  }

  //如果调用线程不是loop线程 或者 正在执行pendingFunctors时，需要唤醒loop
  if (!isInLoopThread() || callingPendingFunctors_)
  {
    wakeup();
  }
}
```

解释一下为什么要判断callingPendingFunctors_条件？因为doPendingFunctors()使用copy-on-write处理pendingFunctors!

```cpp
void EventLoop::doPendingFunctors()
{
  std::vector<Functor> functors;
  callingPendingFunctors_ = true;

  {
  MutexLockGuard lock(mutex_);
  functors.swap(pendingFunctors_);
  }

  for (size_t i = 0; i < functors.size(); ++i)
  {
    functors[i]();
  }
  callingPendingFunctors_ = false;
}
```

eventfd的使用：

```cpp
void EventLoop::wakeup()
{
  uint64_t one = 1;
  ssize_t n = sockets::write(wakeupFd_, &one, sizeof one);
  if (n != sizeof one)
  {
    LOG_ERROR << "EventLoop::wakeup() writes " << n << " bytes instead of 8";
  }
}

void EventLoop::handleRead()
{
  uint64_t one = 1;
  ssize_t n = sockets::read(wakeupFd_, &one, sizeof one);
  if (n != sizeof one)
  {
    LOG_ERROR << "EventLoop::handleRead() reads " << n << " bytes instead of 8";
  }
}
```

Log 的实现

```cpp
#define LOG_TRACE if (muduo::Logger::logLevel() <= muduo::Logger::TRACE) \
  muduo::Logger(__FILE__, __LINE__, muduo::Logger::TRACE, __func__).stream()
#define LOG_DEBUG if (muduo::Logger::logLevel() <= muduo::Logger::DEBUG) \
  muduo::Logger(__FILE__, __LINE__, muduo::Logger::DEBUG, __func__).stream()
#define LOG_INFO if (muduo::Logger::logLevel() <= muduo::Logger::INFO) \
  muduo::Logger(__FILE__, __LINE__).stream()
#define LOG_WARN muduo::Logger(__FILE__, __LINE__, muduo::Logger::WARN).stream()
#define LOG_ERROR muduo::Logger(__FILE__, __LINE__, muduo::Logger::ERROR).stream()
#define LOG_FATAL muduo::Logger(__FILE__, __LINE__, muduo::Logger::FATAL).stream()
#define LOG_SYSERR muduo::Logger(__FILE__, __LINE__, false).stream()
#define LOG_SYSFATAL muduo::Logger(__FILE__, __LINE__, true).stream()
```

`LOG_INFO << “hello world”;` 宏展开后实际为

`Muduo::Logger(__FILE__, __LINE__).stream() << “...”;`

1. 创建一个 Logger(**FILE, LINE**) 的匿名对象；
2. 调用这个匿名对象的 stream() 成员函数；
3. 调用重载后的 « 操作符，输入日志信息；
4. 析构该匿名对象，析构函数内会调用真正的output日志信息函数。

异步日志库：

Log的实现了学习了muduo，Log的实现分为前端和后端，前端往后端写，后端往磁盘写。为什么要这样区分前端和后端呢？因为只要涉及到IO，无论是网络IO还是磁盘IO，肯定是慢的，慢就会影响其它操作，必须让它快才行。

这里的Log前端是前面所述的IO线程，负责产生log，后端是Log线程，设计了多个缓冲区，负责收集前端产生的log，集中往磁盘写。这样，Log写到后端是没有障碍的，把慢的动作交给后端去做好了。

后端主要是由多个缓冲区构成的，集满了或者时间到了就向文件写一次。采用了muduo介绍了“双缓冲区”的思想，实际采用4个多的缓冲区(为什么说多呢？为什么4个可能不够用啊，要有备无患)。4个缓冲区分两组，每组的两个一个主要的，另一个防止第一个写满了没地方写，写满或者时间到了就和另外两个交换**指针**，然后把满的往文件里写。

与Log相关的类包括FileUtil、LogFile、AsyncLogging、LogStream、Logging。 其中前4个类每一个类都含有一个append函数，Log的设计也是主要围绕这个**append**函数展开的。

· FileUtil是最底层的文件类，封装了Log文件的打开、写入并在类析构的时候关闭文件，底层使用了标准IO，该append函数直接向文件写。

· LogFile进一步封装了FileUtil，并设置了一个循环次数，每过这么多次就flush一次。

· AsyncLogging是核心，它负责启动一个log线程，专门用来将log写入LogFile，应用了“双缓冲技术”，其实有4个以上的缓冲区，但思想是一样的。AsyncLogging负责(定时到或被填满时)将缓冲区中的数据写入LogFile中。

· LogStream主要用来格式化输出，重载了<<运算符，同时也有自己的一块缓冲区，这里缓冲区的存在是为了缓存一行，把多个<<的结果连成一块。

· Logging是对外接口，Logging类内涵一个LogStream对象，主要是为了每次打log的时候在log之前和之后加上固定的格式化的信息，比如打log的行、文件名等信息。

双缓冲区异步日志是什么？为什么要这样做？对这个日志系统有没有进行压力测试？

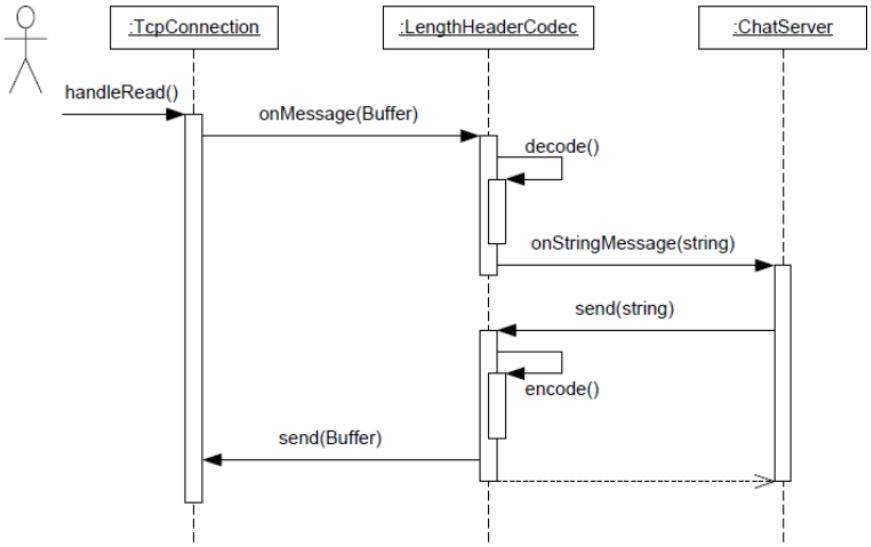
编解码器 Codec

在网络通信时通常是以消息为单位，每条消息有明确的长度与界限。程序每次接收到一个完整的消息时才开始处理，发送时也需要以一个完整的消息交给网络库。

为了实现编解码器，需要在数据包报头中添加相关信息。数据报头部字段主要有len（表明数据包的长度、用于分包）、type（表明数据报的类型，用于实现分发器）、taskId（任务一般是以状态机的形式进行，等待接收消息后推动对应任务的状态机）、result（任务执行状态）等。

- Codec::send(std::string...)：将要发送的数据作为一个包发送，即给数据加上包头。

- TCP分包：根据数据包包头的长度，可以确定每条消息的长度，为消息划分界限。如果只收到了半条消息，那么不会触发消息回调，数据会停留在 Buffer 里，等待收到一个完整的消息再通知处理函数。

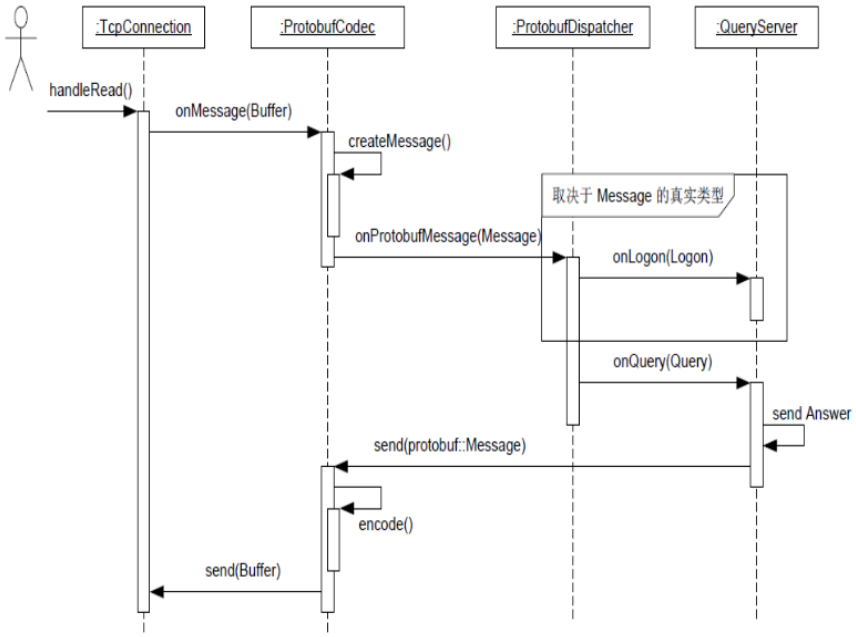


分发器 Dispatcher

在使用 TCP 长连接，且在一个连接上传递不止一种 类型消息的情况下，客户代码需要对收到的消息按类型做分发，不同的消息类型调用不同的回调函数。

比方说，收到 Logon 消息就交给 QueryServer::onLogon() 去处理，收到 Query 消息就交给 QueryServer::onQuery() 去处理。

换句话说，又是一层间接性，Codec 拦截了 TcpConnection 的数据，把它转换为 完整的数据包，Dispatcher 拦截了 Codec 的 callback，按消息具体类型把它分派给对应的 callbacks。



限制并发数

服务器进程会有一个最大连接数，如果达到最大连接数，会报错！

系统最大TCP连接数：

1. 理论上限：

   一个TCP连接使用4元组进行表示{local ip, local port, remote ip, remote port }

   - Client最大tcp连接数

     client每次发起请求时，通常会让系统选取一个空闲的本地端口，client独占该端口。tcp端口的数据类型为unsigned short，因此端口最大的端口数为65536，端口0有特殊含义。故在所有端口都作为client端的情况下，最大tcp连接数为65535。

   - Server最大tcp连接数

     此处不考虑地址复用、不考虑多网卡，即server只有一个ip地址，本地监听端口也是独占的。故理论上server端tcp的最大连接数为：

     1×server可用的端口号(2^16)×remote ip数量(2^32)×remote port数量(2^16) = 2^64

     但在实际环境中，受到机器资源、操作系统等限制，特别是server端，其最大并发tcp连接数远不能达到理论上限。在unix/linux下限制连接数的主要因素是内存和允许的文件描述符个数（每个tcp连接都要占用一定内存，每个socket就是一个文件描述符），另外1024以下的端口通

常为保留端口。在默认2.6内核配置下，每个socket占用内存在15~20k之间。

2. 实际上限：

- 软限制：指Linux在当前系统能够承受的范围内进一步限制用户一个进程中同时打开的文件数。通常软限制小于或等于硬限制。

  文件句柄进程级限制：

  - 进程限制、临时修改：使用命令查看、修改 ulimit -n
  - 重启后失效的修改：/etc/security/limits.conf
  - 永久修改：编辑/etc/rc.local

  文件句柄全局限制：

  执行 cat /proc/sys/fs/file-nr 输出 9344 0 592026 分别为： 1.已经分配的文件句柄数， 2.已经分配但没有使用的文件句柄数， 3.最大文件句柄数。

  用 root 权限修改 /etc/sysctl.conf 文件:

  ```
  fs.file-max = 1000000
  net.ipv4.ip_conntrack_max = 1000000
  net.ipv4.netfilter.ip_conntrack_max = 1000000
  ```

- 硬限制：是根据系统硬件资源状况(主要是系统内存)计算出来的系统最多可同时打开的文件数量。如果没有特殊需要，不应该修改此限制，除非想为用户级打开文件数限制设置超过此限制的值。

参考网页：

Linux下高并发socket最大连接数所受的各种限制

单服务器最大tcp连接数及调优汇总

在服务端开发时，为了避免程序超载，避免描述符耗尽，需要限制并发连接数。当达到最大连接数时，如何处理后续新连接？

推荐做法：准备一个空闲的文件描述符，当达到最大连接数时，先关闭该文件描述符，此时可以获得一个文件描述符名额，再accept拿到新的socket连接的描述符；然后close()，这样便可以优雅地关闭客户端连接。

或者，程序中设置软链接上限，超过上限的链接都会被关闭。

平滑升级

问题：如果要对服务器进行升级，比如说重启服务器加载新的代码或配置，但不想让用户感觉到服务器升级了，参考nginx的平滑升级。

重启流程：

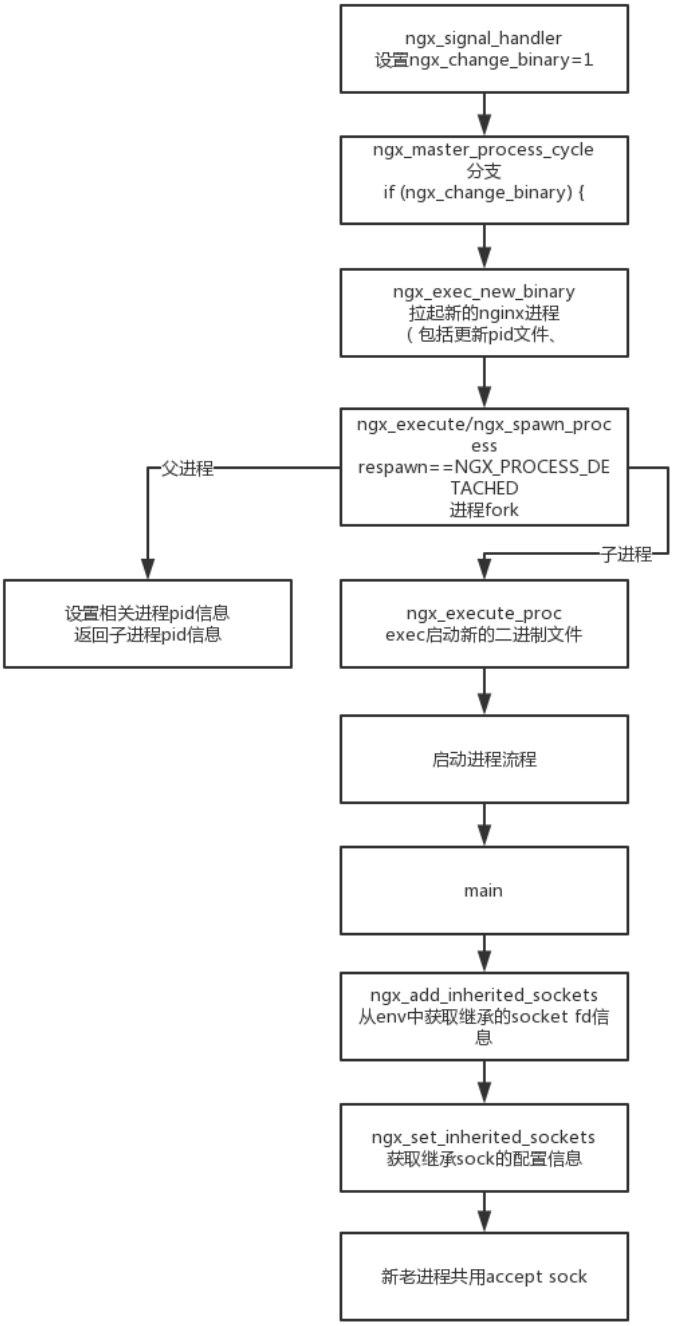
- 启动新的server
- 新旧server并存提供服务，旧server不接受新的请求，新server负责后续请求
- 旧server处理完所有的请求之后退出

第二步中，如何保证新旧server并存，如果重启前后的server端口一致，如何保证两个可以监听同一端口？

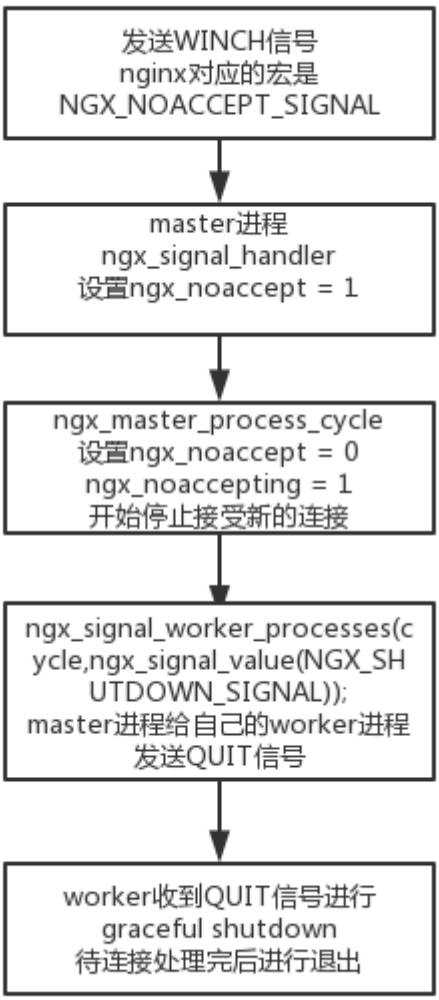
在尚未开启端口复用选项（SO_REUSEPORT）的情况下，直接再起一个会报错：Address already in use.

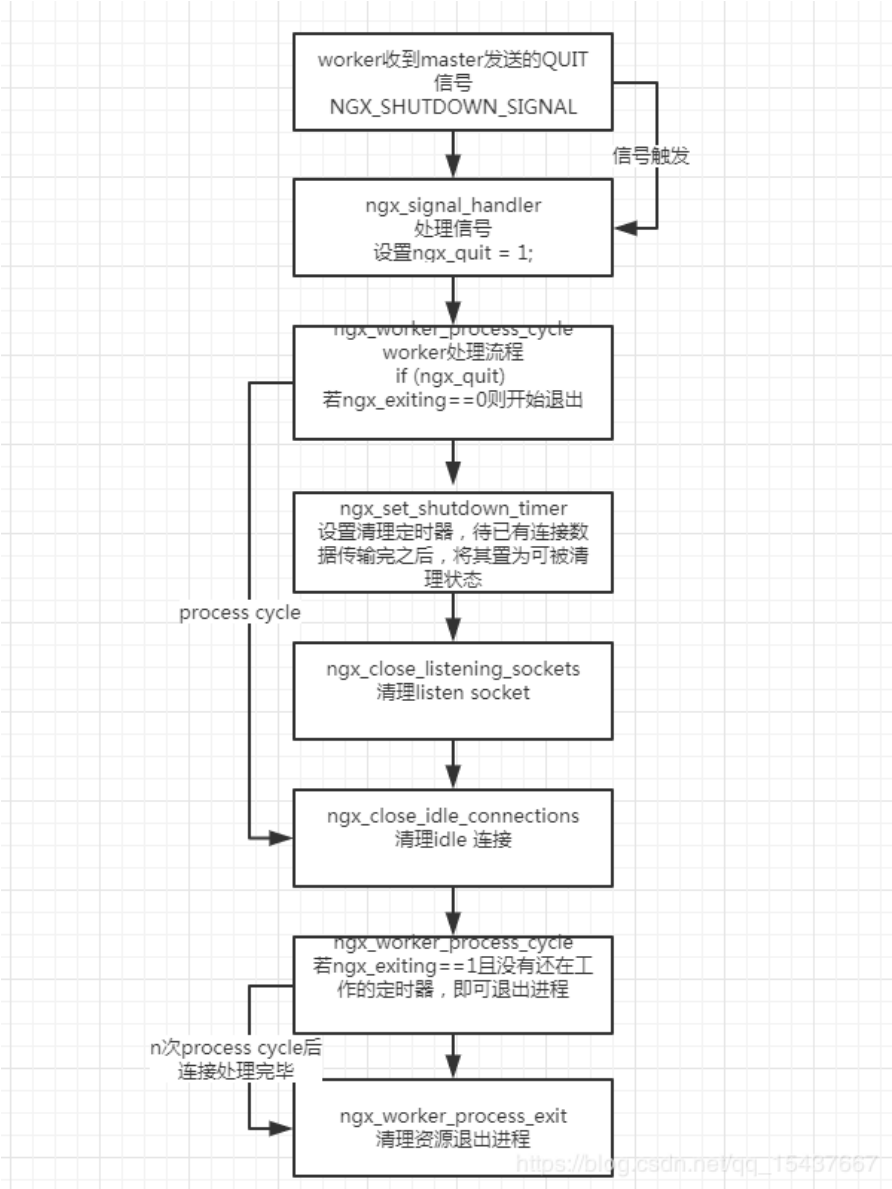
平滑重启的关键点有两个：

- 使用信号USR2、WINCH、QUIT
    - USR2：升级可执行文件



- WINCH：master停止worker进程


发送WINCH信号
nginx对应的宏是
NGX_NOACCEPT_SIGNAL
master进程
ngx_signal_handler
设置ngx_noaccept = 1
ngx_master_process_cycle
设置ngx_noaccept = 0
ngx_noaccepting = 1
开始停止接受新的连接
ngx_signal_worker_processes(c
ycle,ngx_signal_value(NGX_SH
UTDOWN_SIGNAL));
master进程给自己的worker进程
发送QUIT信号
worker收到QUIT信号进行
graceful shutdown
待连接处理完后进行退出



- QUIT：停止master进程
- 通过fork/exec启动新的版本的进程，execv时通过环境变量传递文件描述符

参考网页：

nginx热更新原理分析

TODO：nginx中使用了信号，fork() + exec()，调研多线程环境中，信号和fork()存在的问题？为什么muduo中不推荐在多线程编程中使用信号和fork()？

muduo使用的简单介绍

关注三个半事件

Server端编程：

Client端编程：

IO线程池：

计算线程池：

怎么检查内存泄漏的？

怎么进行压测的？

实现内存池

pipelining模型

火焰图

参考链接：

使用火焰图（FlameGraph）来分析程序性能

宋宝华：火焰图：全局视野的Linux性能剖析

Linux下用火焰图进行性能分析