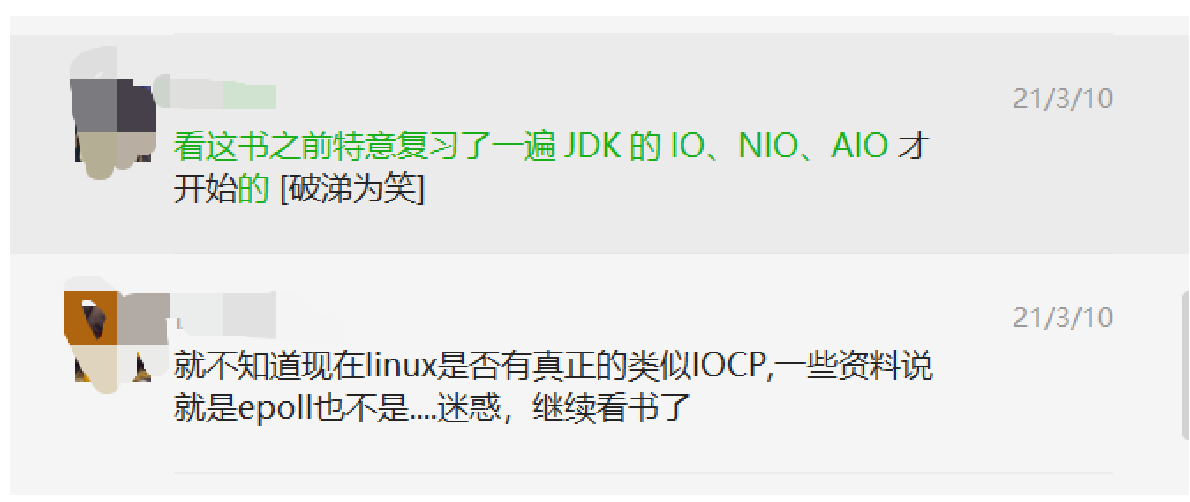
彻底明白：操作系统 select、poll 核心原理

背景



学习高并发，epoll是个基础

从菜鸟到大神视频的宗旨：

与《高并发三部曲》想配合，为大家打通任督二脉

epoll的重要性

epoll作为linux下高性能网络服务器的必备技术至关重要，

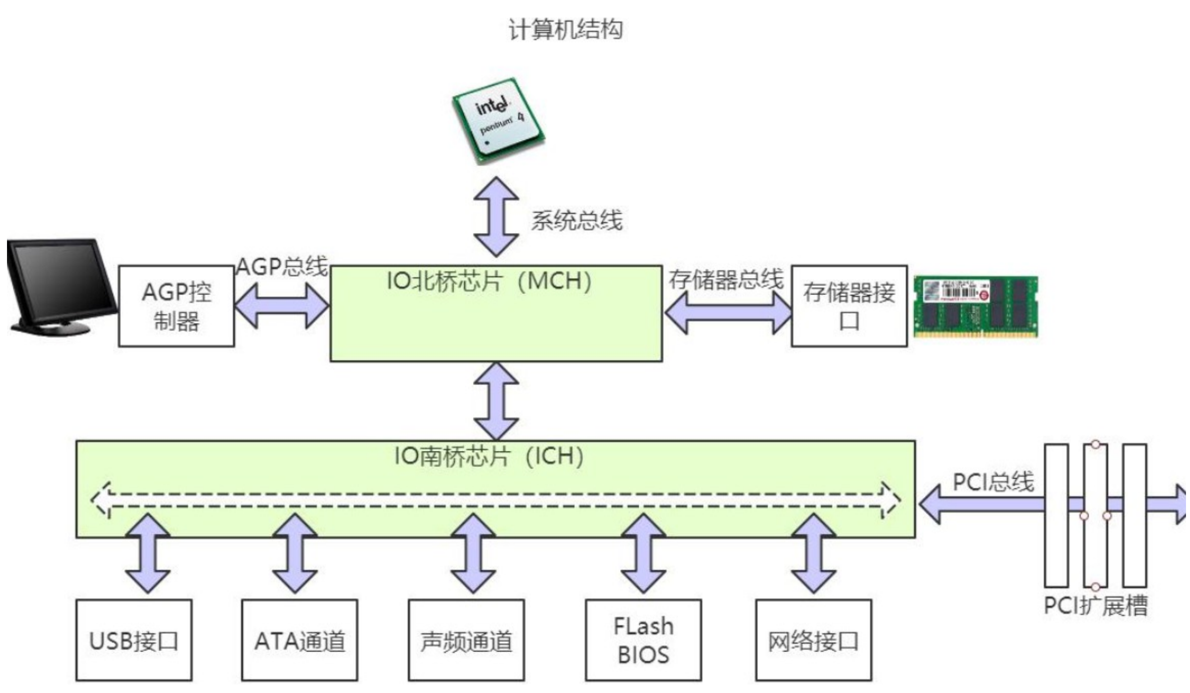
Java NIO、nginx、redis、skynet和大部分游戏服务器都使用到这一多路复用技术。

epoll的重要性，大厂面试必备

不少大厂在招聘服务端同学时，可能会问及epoll相关的问题。

比如epoll和select的区别是什么？

epoll高效率的原因是什么？



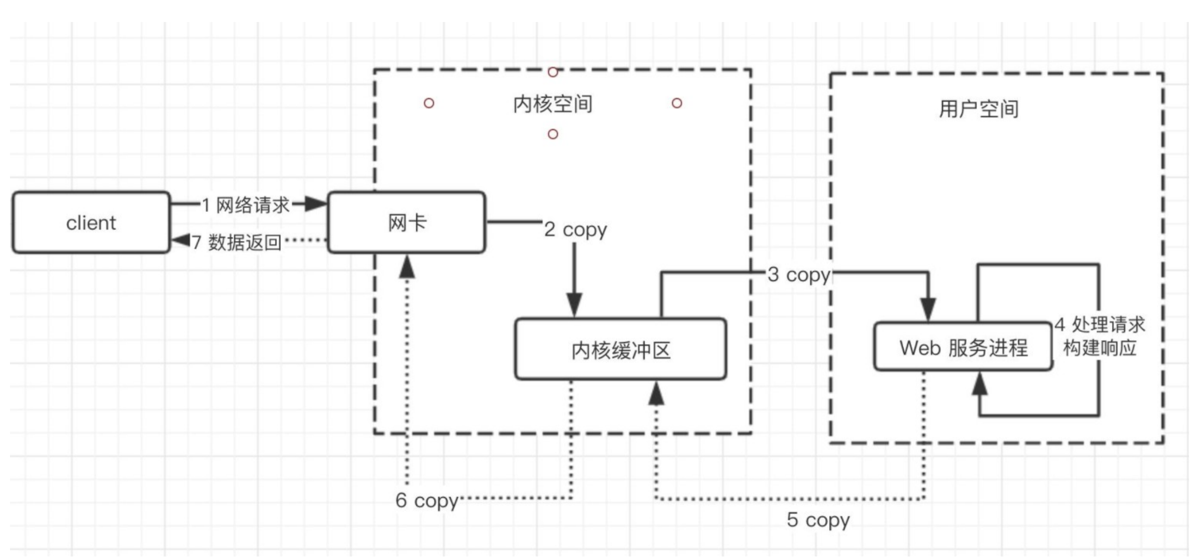
数据的读取过程：网卡会把接收到的数据写入Socket内核缓冲

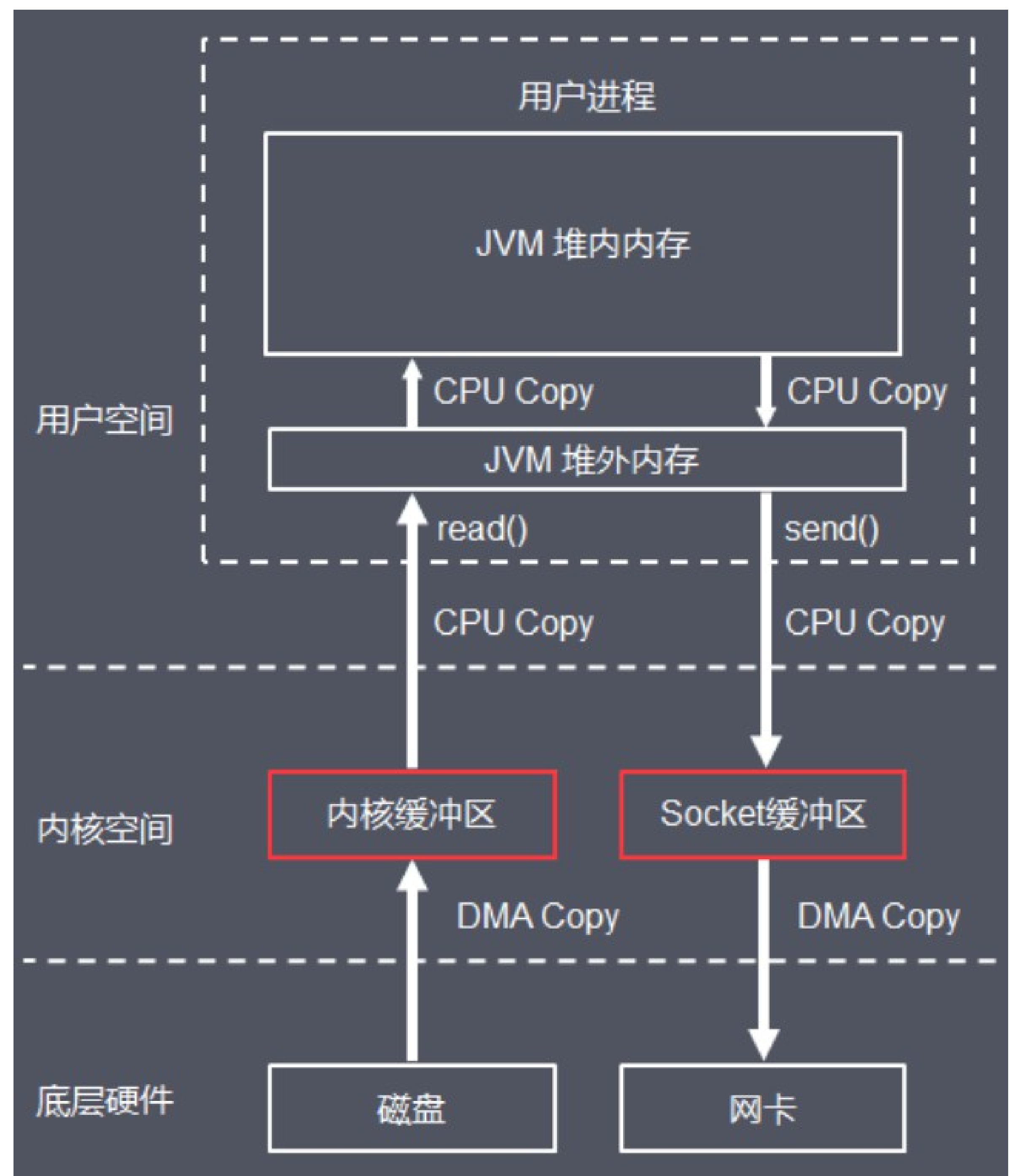
①阶段 ： 网卡收到网线传来的数据

②阶段： 硬件电路的传输；

③阶段 最终将数据写入到内存中的某个地址上

这个过程涉及到DMA传输、IO通路选择等硬件有关的知识

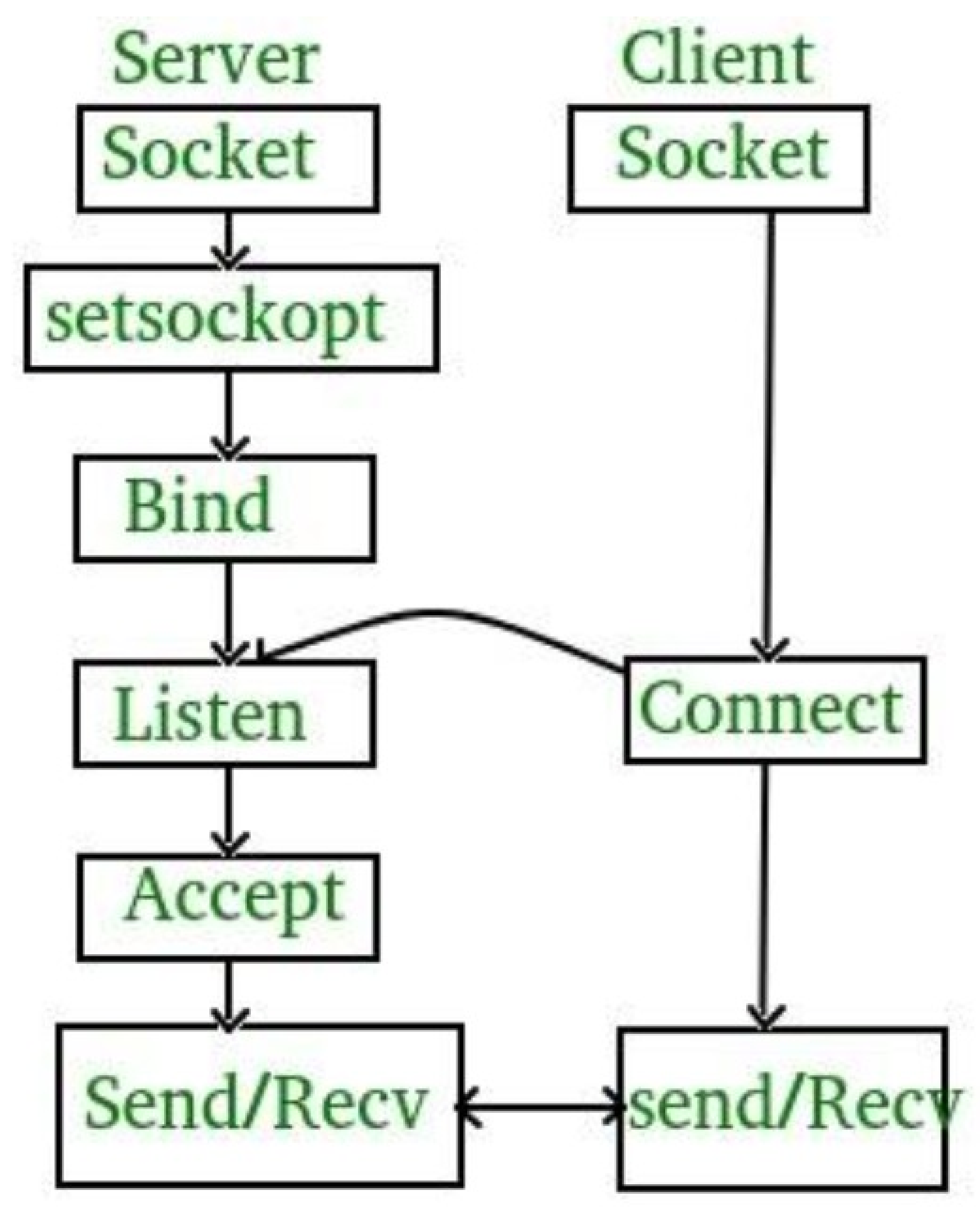




网络编程

这是一段最基础的网络编程代码，先新建 socket 对象，依次调用 bind 、 listen 、 accept ， 最后调用 recv 接收数据。

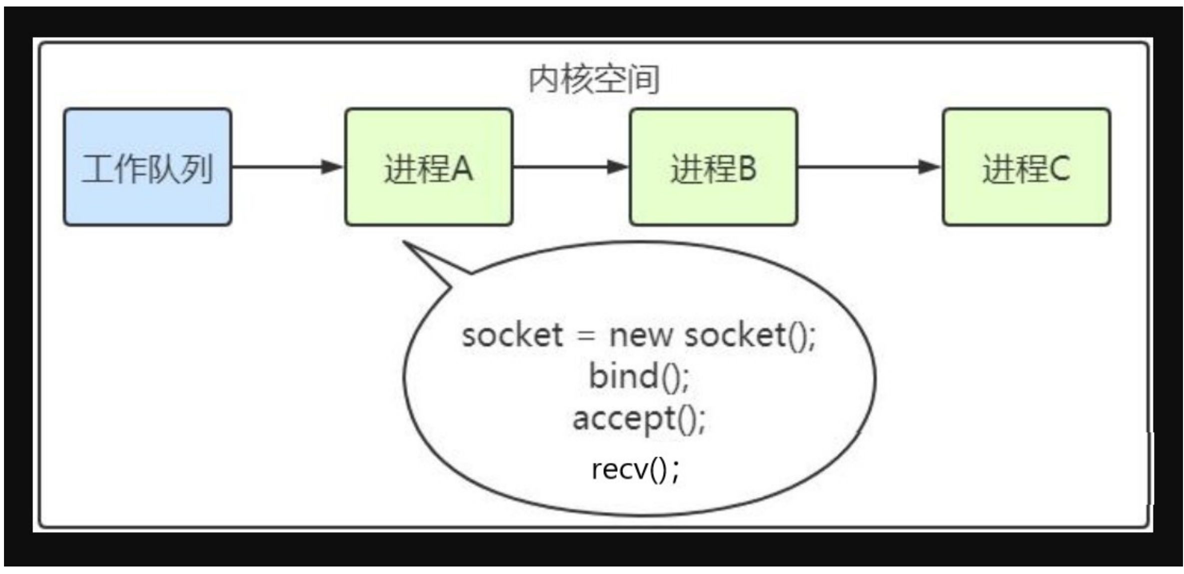
```
//创建socket
int s = socket(AF_INET, SOCK_STREAM, 0);
//绑定
bind(s, ...)
//监听
listen(s, ...)
//接受客户端连接
int c = accept(s, ...)
//接收客户端数据
recv(c, ...);
//将数据打印出来
printf(...)
```



操作系统工作队列

操作系统为了支持多任务，实现了进程调度的功能，会把进程分为“运行”和“等待”等几种状态。运行状态是进程获得cpu使用权，正在执行代码的状态；等待状态是阻塞状态，比如上述程序运行到recv时，程序会从运行状态变为等待状态，接收到数据后又变回运行状态。操作系统会分时执行各个运行状态的进程，由于速度很快，看上去就像是同时执行多个任务。

下图中的计算机中运行着A、B、C三个进程，其中进程A执行着上述基础网络程序，一开始，这3个进程都被操作系统的工作队列所引用，处于运行状态，会分时执行。



socket 对象的结构

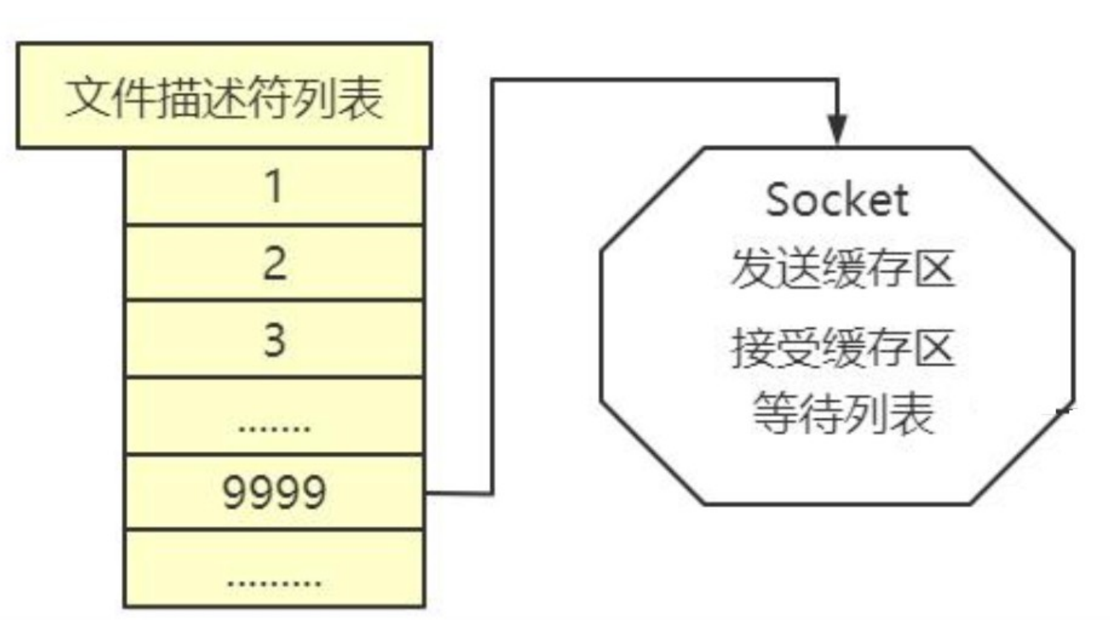
当进程 A 执行到创建 socket 的语句时，操作系统会创建一个由文件系统管理的 socket 对象。

这个 socket 对象包含了

- 发送缓冲区
- 接收缓冲区
- 等待队列等成员。

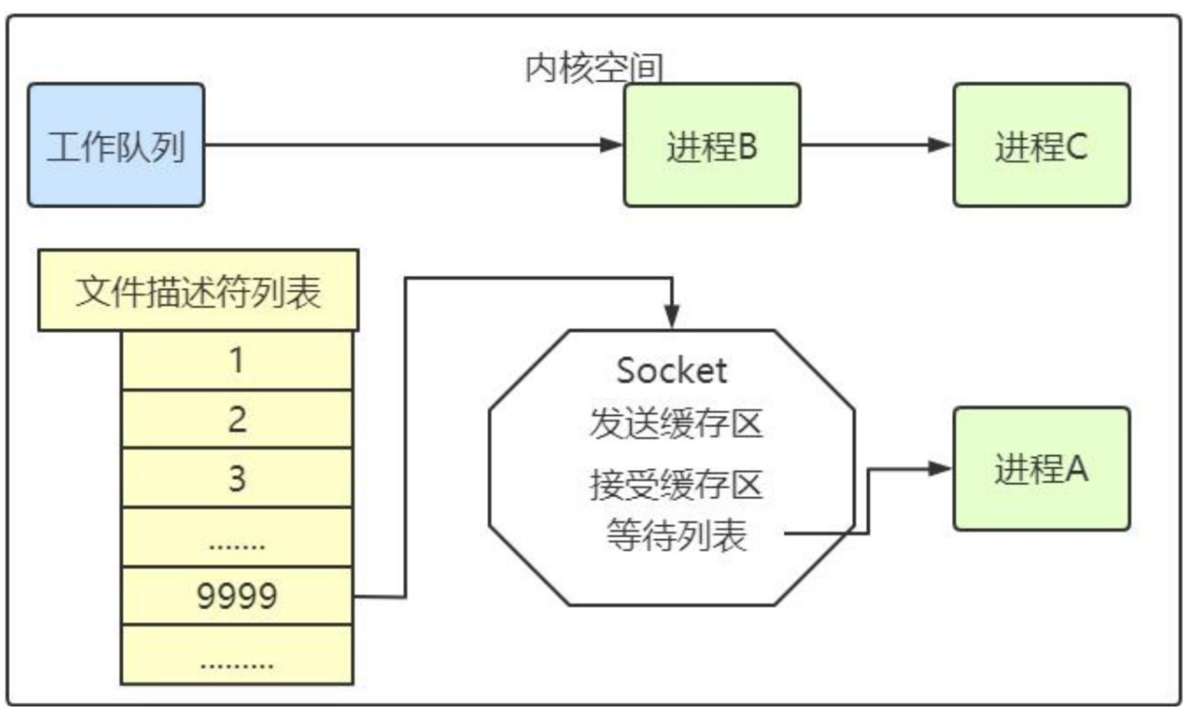
socket 的等待队列是个非常重要的结构，它指向所有需要等待该 socket 事件的进程。

socket 对象图解



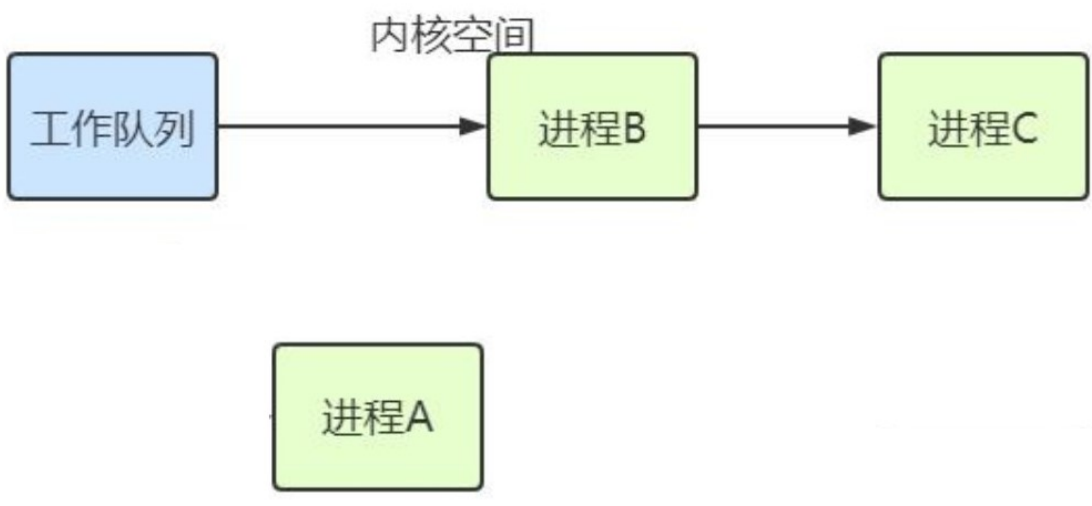
进程 A 从工作队列移动到该 socket 的等待队列中

当程序执行到 recv 时，操作系统会将进程 A 从工作队列移动到该 socket 的等待队列中 。线程阻塞。

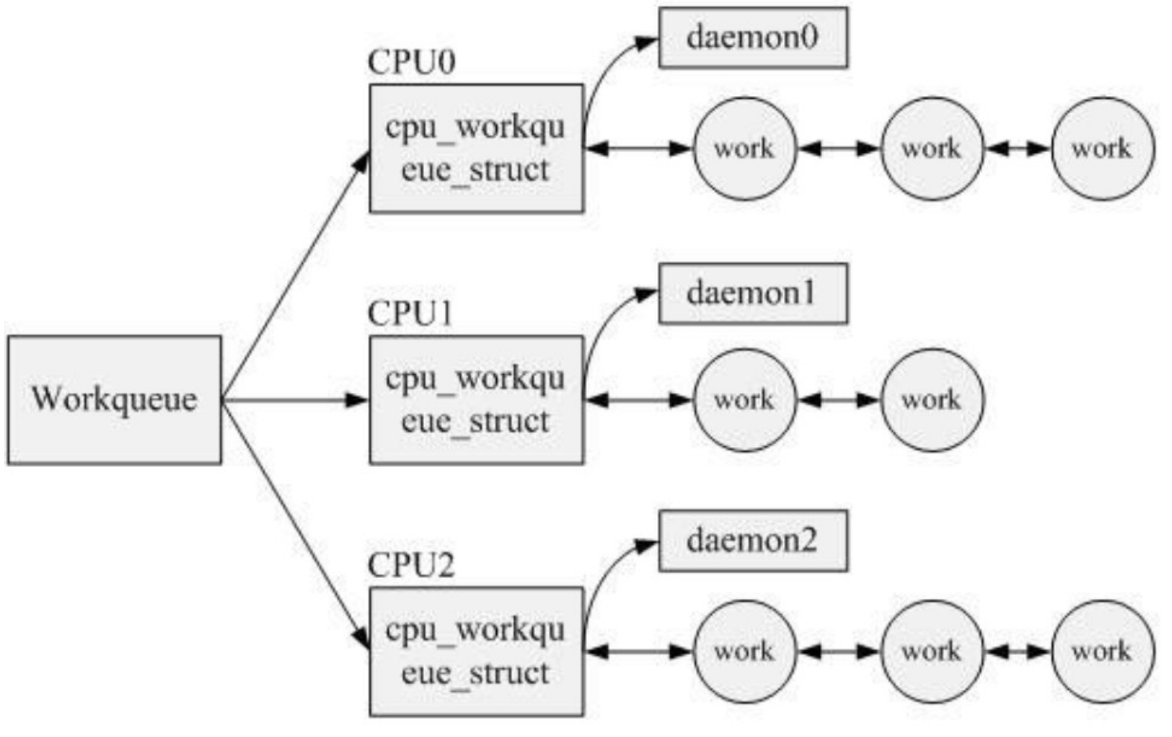


题外知识：阻塞线程不会占用CPU资源

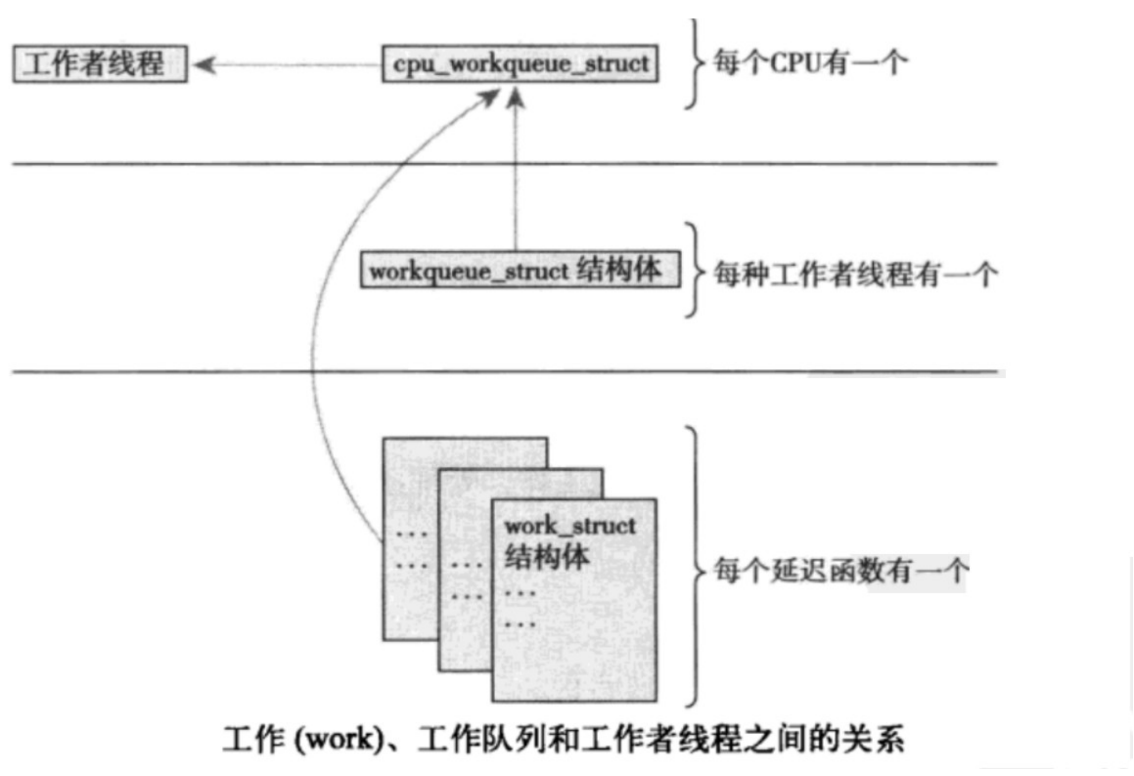
由于工作队列只剩下了进程 B 和 C ，依据进程调度， cpu 会轮流执行这两个进程的程序，不会执行进程 A 的程序。所以进程 A 被阻塞，不会往下执行代码，也不会占用 cpu 资源。



CPU的工作队列



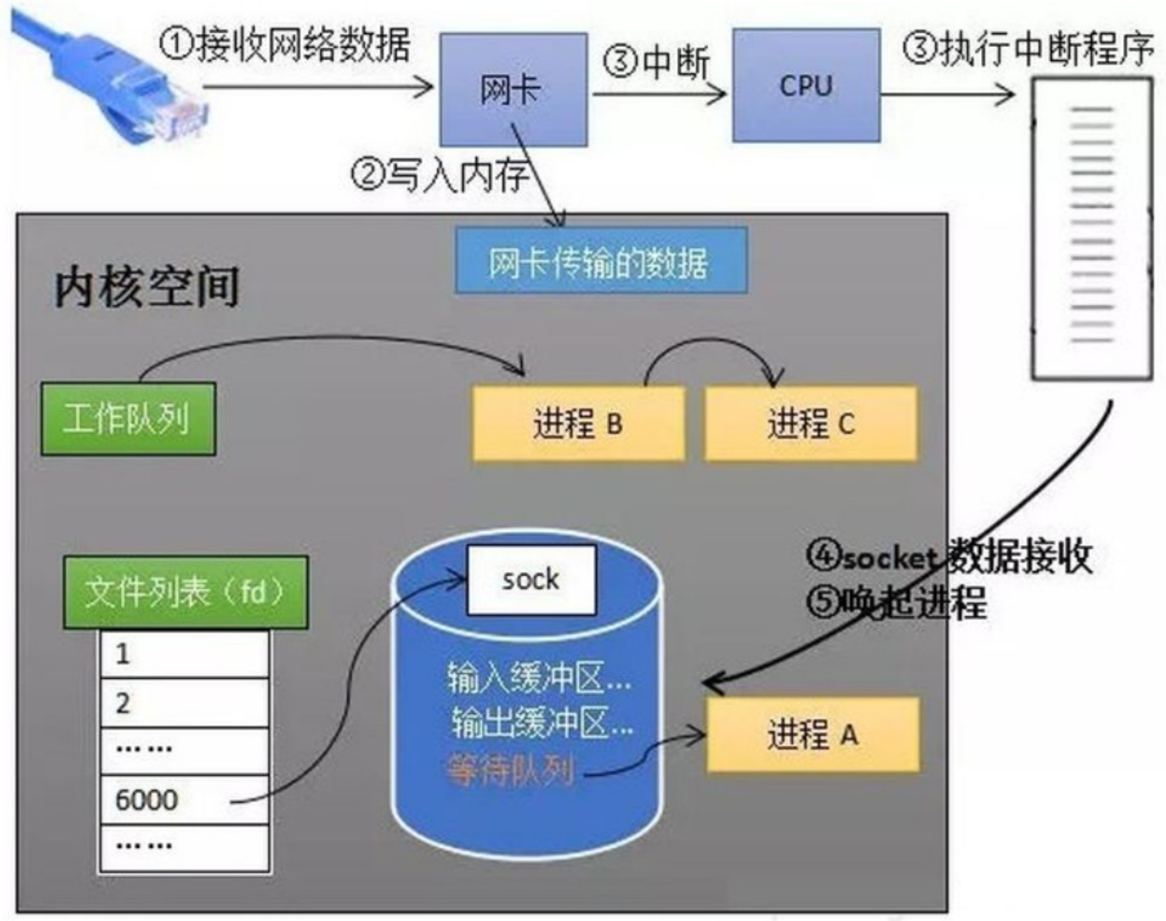
线程、CPU与工作队列之间的关系



工作 (work)、工作队列和工作者线程之间的关系

内核接收数据全过程

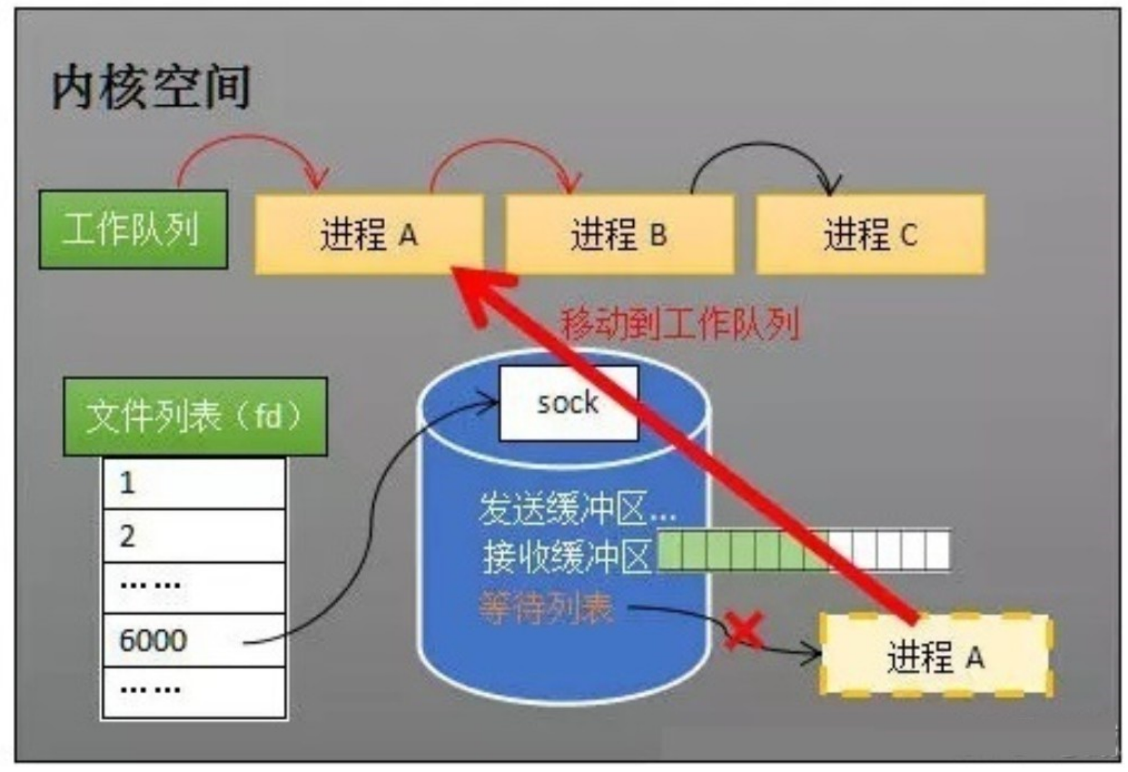
当 socket 接收到数据后，操作系统将该 socket 等待队列上的进程重新放回到工作队列，该进程变成运行状态，继续执行代码。也由于 socket 的接收缓冲区已经有了数据， recv 可以返回接收到的数据。



内核接收数据全过程：

- 计算机收到了对端传送的数据(步骤 ①)
- 数据经由网卡传送到内存(步骤 ②)
- 然后网卡通过中断信号通知 CPU 有数据到达，CPU 执行中断程序(步骤 ③)
- 中断程序先将网络数据写入到对应 Socket 的接收缓冲区里面(步骤 ④)
- 再唤醒进程 A(步骤 ⑤)，重新将进程 A 放入工作队列中。

唤醒进程的过程如下图所示：



问题1：内核如何知道接收的网络数据时属于哪个socket?

socket数据包格式（源ip，源端口，协议，目的ip，目的端口）

一般通过目的ip，目的端口 就可以识别出来接收到的网络数据属于哪个socket。

如果目的ip，目的端口相同呢?

其实多个客户端与同一个服务端建立了连接，这个时候内核就会有多个socket。

并且为它们分配多个fd文件描述符。它们收到网络数据后无法通过目的端口来直接匹配socket，还需要再通过源ip和端口来确定属于哪个socket。

问题2：内核如何同时监控多个socket?

内核负责轮询所有socket，当某个socket有数据到达了，就通知用户进程。

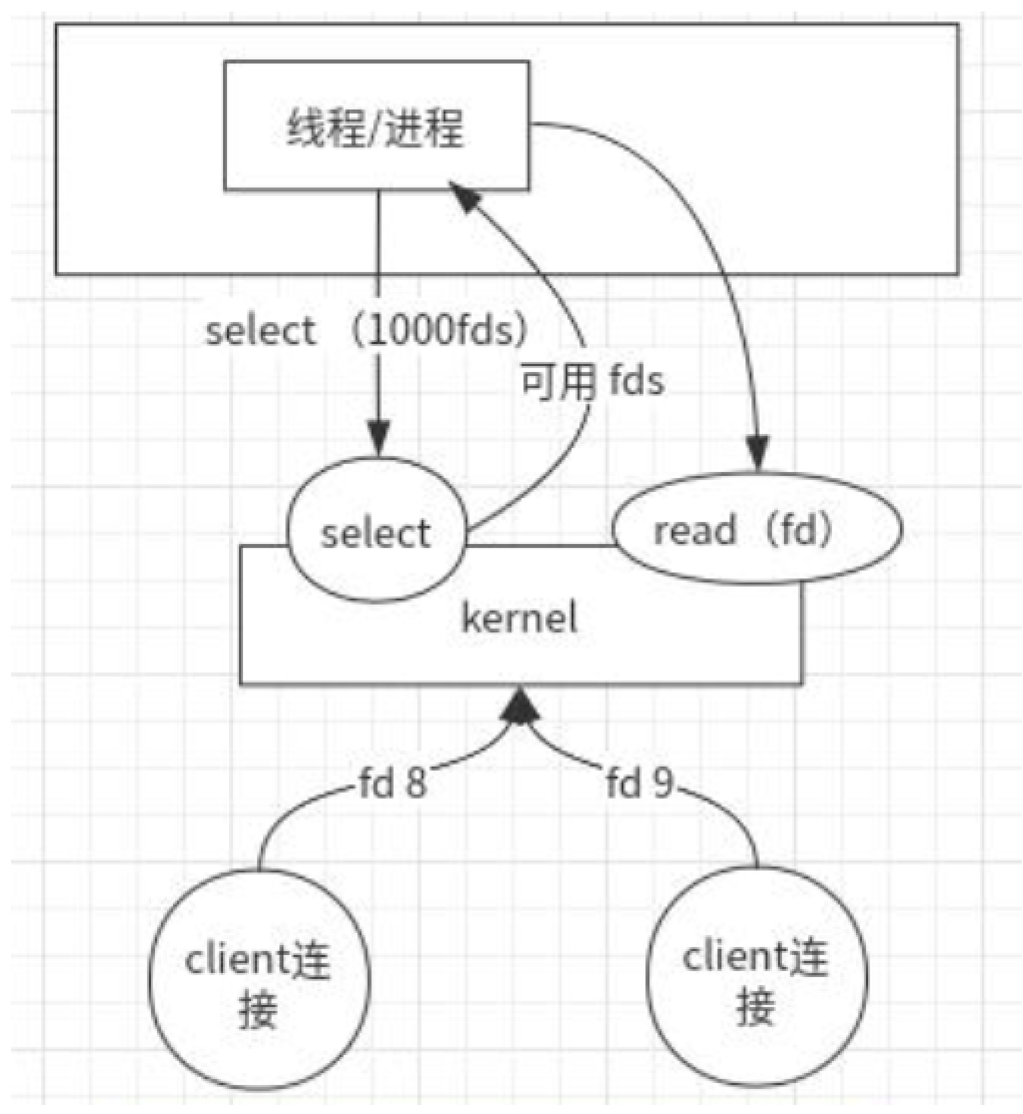
> 目前的经典解决办法：I/O 多路复用 - IO multiplexing

多路复用在Linux内核代码迭代过程中依次支持了三种调用:

- SELECT
- POLL
- EPOLL

select 操作

Linux 内核、Windows内核都提供了 select 操作，可以把1024个文件描述符的IO事件轮询，简化为一次轮询，轮询发生在内核空间。



在如下的代码中，先准备一个数组 fds 存放着所有需要监视的 socket 。然后调用 select ， 如果 fds 中的所有 socket 都没有数据，select 会阻塞，直到有一个 socket 接收到数据， select 返回，唤醒进程。

```
int fds[] = 存放需要监听的 socket
while(1){
int n = select(..., fds, ...)  //fds 加入到socket 阻塞队列，select 返回后，唤醒进程
for(int i=0; i < fds.count; i++){
   if(FD_ISSET(fds[i], ...)){
       //fds[i]的数据处理}
   }
}
```

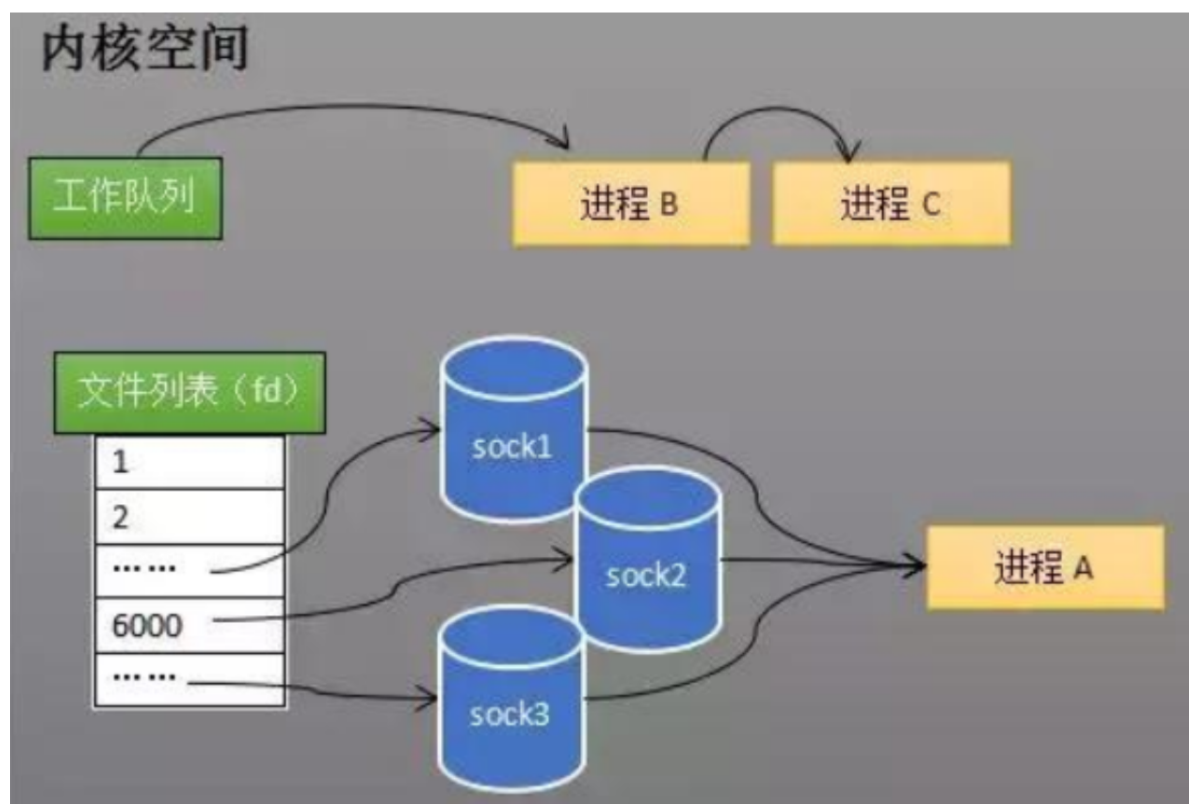
用户可以遍历 fds数组 ，通过 FD_ISSET 判断具体哪个 socket 收到数据，然后做出处理。

使用select的核心步骤：

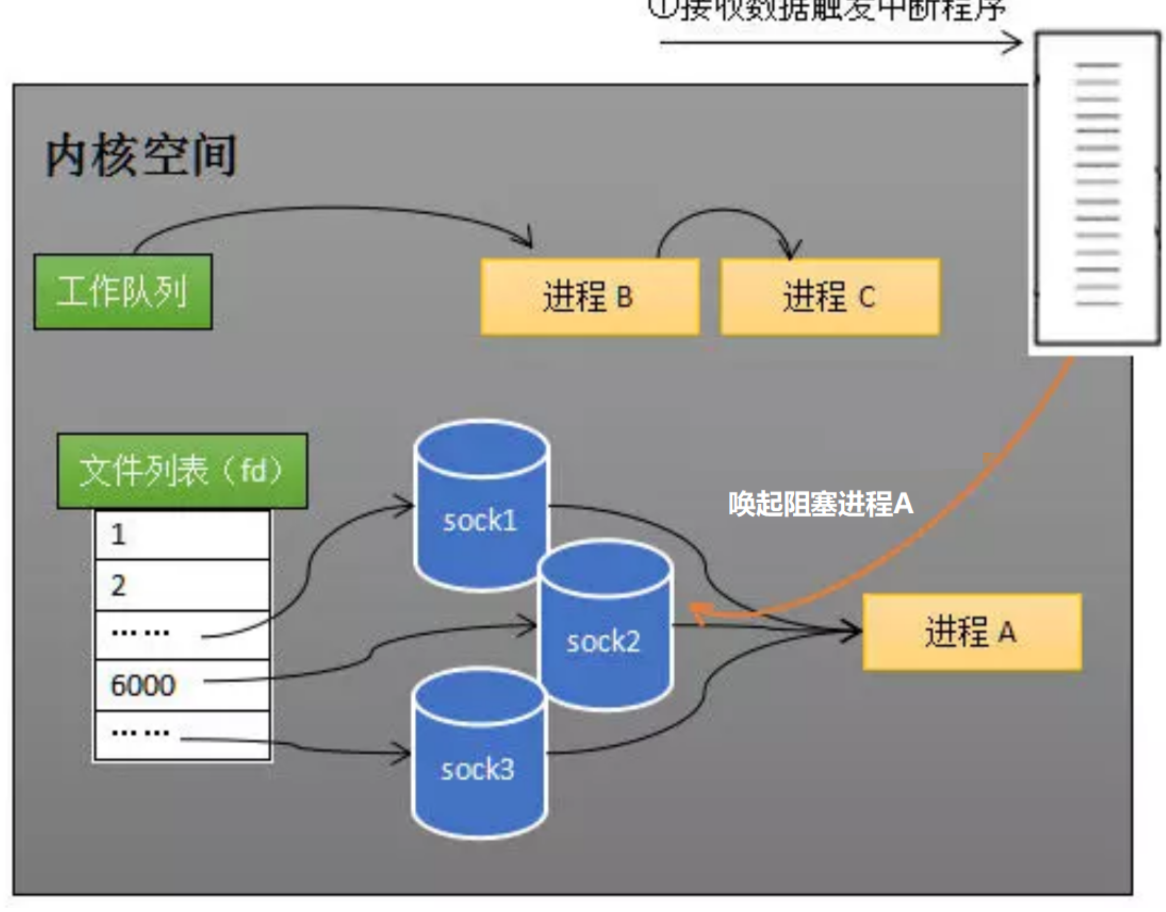
- 先准备了一个数组 fds，让 fds 存放着所有需要监视的 socket。
- 然后调用 select，如果 fds 中的所有 socket 都没有数据，select 会阻塞，直到有一个 socket 接收到数据，select 返回，唤醒进程。
- 用户可以遍历 fds，通过 FD_ISSET 判断具体哪个 socket 收到数据，然后做出处理。

select例子

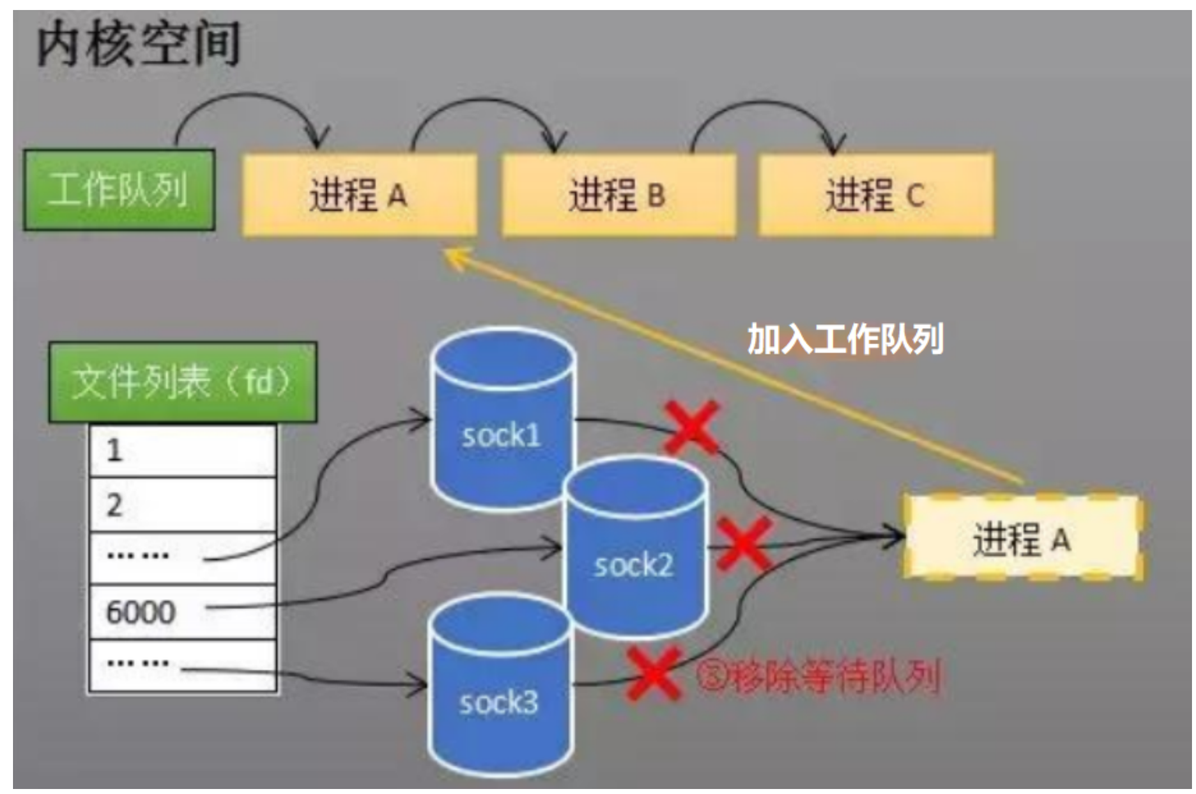
假如程序同时监视如下图的 Sock1、Sock2 和 Sock3 三个 Socket，那么在调用 Select 之后，操作系统把进程 A 分别加入这三个 Socket 的等待队列中。



当任何一个socket收到数据后，中断程序将唤起进程。下图展示了sock2接收到了数据的处理流程。



所谓唤起进程，就是将进程从所有的等待队列中移除，加入到工作队列里面。如下图所示。



对于调用了select的进程A而言：

1. A存在于多个socket的等待队列中
2. 当某个socket被写入数据时，A也被唤醒并从多个socket的等待队列中移除后加入内核的工作队列
3. 但是此时A并不知道是哪个socket被写入了数据，所以只能遍历所有socket
4. 在A处理完任务后移出内核的工作队列，但是此时却需要遍历所有socket并加入它们的等待队列中

select的不足：

两次socket列表遍历：

- 第1次：每次调用select都需要将fds列表传递给内核，有一定的开销。进程加入socket的等待队列时，需要遍历所有socket。
- 第2次：当进程A被唤醒后，进程只知道至少有一个socket接收了数据，不知道是谁。程序需遍历一遍socket列表，才可以得到就绪的socket。唤醒后需要从等待队列中移除。

> 正是因为遍历操作开销大，出于效率的考量，才会规定select的最大监视数量，默认只能监视1024个socket。

poll的出现

1997 年，出现了 poll 作为 select 的替代者，最大的区别就是，poll 不再限制 socket 数量。

poll系统调用

poll其实内部实现基本跟select一样，区别在于它们底层组织fd[]的数据结构不太一样，从而实现了poll的最大文件句柄数量限制去除了

> poll的描述fd集合的方式不同，poll使用**pollfd结构**而不是select的**fd_set结构**，其他的都差不多，管理多个描述符也是进行轮询，根据描述符的状态进行处理，**但是poll没有最大文件描述符数量的限制**。

```
SYNOPSIS
       #include <poll.h>

       int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

---

pollfd结构

成员变量说明：
（1）fd：每一个 pollfd 结构体指定了一个被监视的文件描述符，可以传递多个结构体，指示 poll() 监视多个文件描述符。
（2）events：表示要告诉操作系统需要监测fd的事件（输入、输出、错误），每一个事件有多个取值
（3）revents：revents 域是文件描述符的操作结果事件，内核在调用返回时设置这个域。events 域中请求的任何事件都可能在 revents 域中返回。

```
struct pollfd {
    int   fd;         /* file descriptor */
    short events;     /* requested events */
    short revents;    /* returned events */
};
```

---

events&revents的取值如下：

事件	描述	是否可作为输入（events）	是否可作为输出（revents）
POLLIN	数据可读（包括普通数据&优先数据）	是	是
POLLOUT	数据可写（普通数据&优先数据）	是	是
POLLRDNORM	普通数据可读	是	是
POLLRDBAND	优先级带数据可读（linux不支持）	是	是
POLLPRI	高优先级数据可读，比如TCP带外数据	是	是
POLLWRNORM	普通数据可写	是	是

事件	描述	是否可作为输入（events）	是否可作为输出（revents）
POLLWRBAND	优先级带数据可写	是	是
POLLRDHUP	TCP连接被对端关闭，或者关闭了写操作，由GNU引入	是	是
POPPHUP	挂起	否	是
POLLERR	错误	否	是
POLLNVAL	文件描述符没有打开	否	是

---

socket读就绪条件(读事件)：

1. 该套接字接收缓冲区中的数据字节数大于等于套接字接收缓存区**低水位标记SO_RCVLOWAT。**

   对于TCP和UDP套接字而言，**缓冲区低水位的值默认为1**。那就意味着，默认情况下，只要缓冲区中有数据，那就是可读的。我们可以通过使用SO_RCVLOWAT套接字选项(参见setsockopt函数)来设置该套接字的低水位大小。此种描述符就绪(可读)的情况下，当我们使用read/recv等对该套接字执行读操作的时候，套接字不会阻塞，而是成功返回一个大于0的值（即可读数据的大小）。
2. **该连接的读半部关闭**（也就是接收了FIN的TCP连接）。对这样的套接字的读操作，将不会阻塞，而是返回0（也就是EOF）。
3. 该套接字是一个listen的**监听套接字**，并且目前已经完成的连接数不为0。对这样的套接字进行accept操作通常不会阻塞。
4. 有一个**错误套接字待处理**。对这样的套接字的读操作将不阻塞并返回-1（也就是返回了一个错误），同时把errno设置成确切的错误条件。这些待处理错误(pending error)也可以通过指定SO_ERROR套接字选项调用getsockopt获取并清除。

---

socket写就绪条件(写事件)：

1. socket内核中, 发送缓冲区中的可用字节数(发送缓冲区的空闲位置大小), **大于等于低水位标记SO_SNDLOWAT**, 此时可以无阻塞的写, 并且返回值大于0。

   对于TCP和UDP而言，这个**低水位SO_SNDLOWAT的值默认为2048**，而**套接字默认的发送缓冲区大小是8k**，这就意味着一般一个套接字连接成功后，就是处于可写状态的。我们可以通过SO_SNDLOWAT套接字选项（参见setsockopt函数）来设置这个低水位。此种情况下，我们设置该套接字为非阻塞，对该套接字进行写操作(如write,send等)，将不阻塞，并返回一个正值（例如由传输层接受的字节数，即发送的数据大小）。
2. **该连接的写半部关闭**(主动发送FIN包的TCP连接)。对这样的套接字的写操作将会产生SIGPIPE信号。所以我们的网络程序基本都要自定义处理SIGPIPE信号。因为SIGPIPE信号的默认处理方式是程序退出。
3. 使用非阻塞的connect套接字已建立连接，或者connect已经以失败告终。即connect有结果了。
4. 有一个错误的套接字待处理。对这样的套接字的写操作将不阻塞并返回-1（也就是返回了一个错误），同时把errno设置成确切的错误条件。这些待处理的错误也可以通过指定SO_ERROR套接字选项调用getsockopt获取并清除。

条件	可读吗?	可写吗?	异常吗?
有数据可读	•		
关闭连接的读一半	•		
给监听套接口准备好新连接	•		
有可用于写的空间		•	
关闭连接的写一半		•	
待处理错误	•	•	
TCP带外数据			•

poll() 系统调用的本质

select() 和 poll() 系统调用的本质一样

poll() 的机制与 select() 在本质上没有多大差别，每次调用时，都需要把 fd 集合从用户态拷贝到内核态，

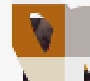
二者管理多个描述符也是进行轮询，根据描述符的状态进行处理。

彻底明白：epoll 底层原理

背景

21/3/10

看这书之前特意复习了一遍 JDK 的 IO、NIO、AIO 才开始的 [破涕为笑]



21/3/10

就不知道现在linux是否有真正的类似IOCP,一些资料说就是epoll也不是....迷惑，继续看书了

学习高并发，epoll是个基础

从菜鸟到大神视频的宗旨：

与《高并发三部曲》想配合，为大家打通任督二脉

---

epoll的重要性

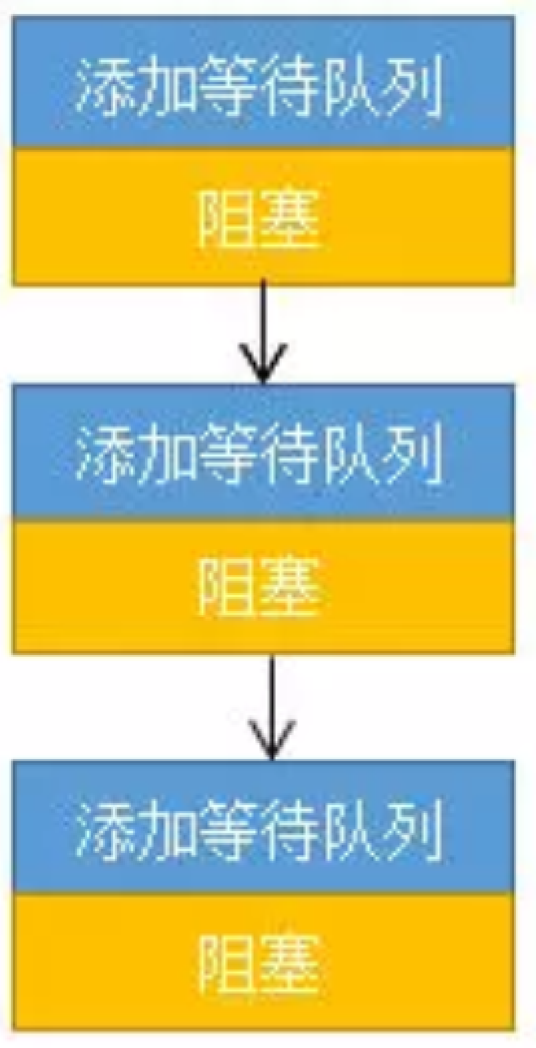
epoll作为linux下高性能网络服务器的必备技术至关重要，

Java NIO、nginx、redis、skynet和大部分游戏服务器都使用到这一多路复用技术。

---

select /poll低效的原因

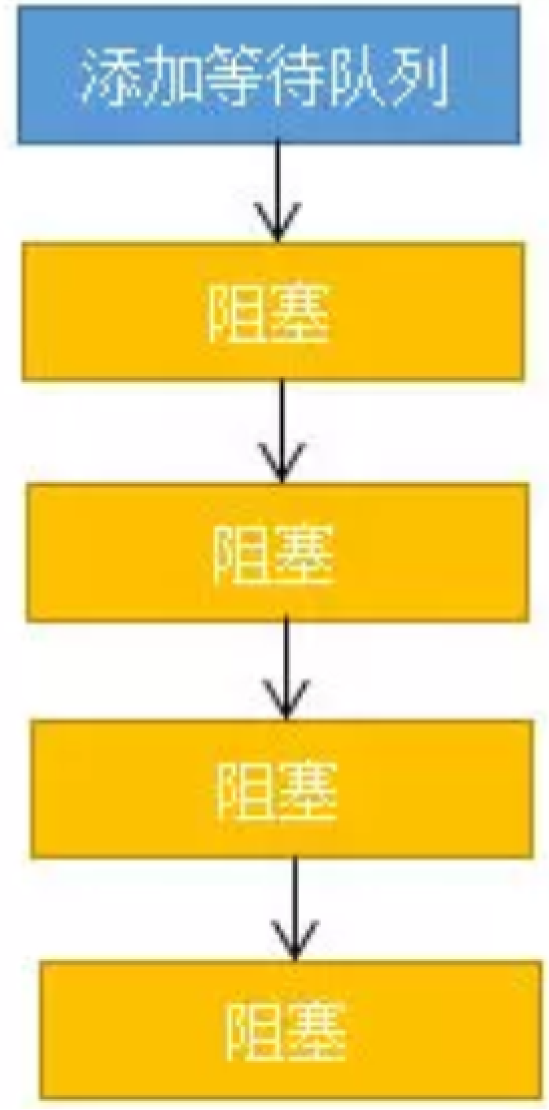
一次select调用，**将"维护等待队列"和"阻塞等待"两个步骤合二为一，紧密耦合**



每次调用 select 都需要这两步操作：添加进程到socket的等待队列，阻塞进程。

然而大多数应用场景中，需要监视的 socket 相对固定，并不需要每次都修改。

如何优化：**epoll 将这两个操作分开，先用 epoll_ctl 维护等待队列，再调用 epoll_wait 阻塞进程。**



显而易见地，效率就能得到提升。

---

epoll 的出现

epoll 是在 select 、poll出现 N 多年后才被发明的，是 select 和 poll 的增强版本。 epoll 通过以下一些措施来改进效率。

---

epoll 的优化措施：

- 功能分离：进程到等待队列，进程阻塞
- 引入了就绪列表rdlist

---

epoll 的三个方法

- epoll_create：内核会创建一个 eventpoll 对象（专用的文件描述符，也就是程序中 epfd 所代表的对象）

  eventpoll 对象也是文件系统中的一员，和 socket 一样，它也会有等待队列。

- epoll_ctl：添加待监控的socket

如果通过 epoll_ctl 添加 sock1、sock2 和 sock3 的监视，内核会将三个 socket 添加到 eventpoll 监听队列。

- epoll_wait：阻塞等待

  进程 A 运行到了 epoll_wait 语句之后，进程A会等待eventpoll 的等待队列。

epoll 的优化措施一：功能分离

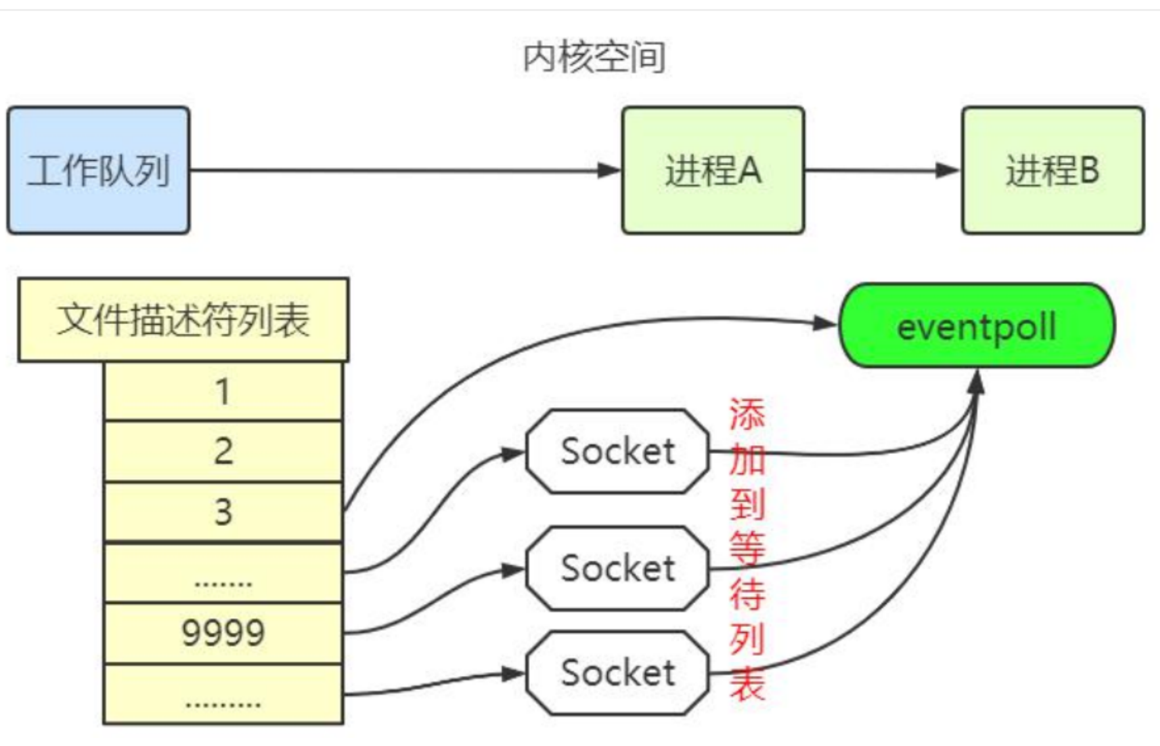
select 低效的原因之一是将"维护等待队列"和"阻塞进程"两个步骤合二为一。

> 大多数应用场景中，需要监视的 socket 相对固定，并不需要每次都修改。

epoll 将这两个操作分开，先用 epoll_ctl 维护等待队列，再调用 epoll_wait 阻塞进程。

显而易见的，效率就能得到提升。

epoll 的等待列表



epoll 的优化措施二：就绪列表rdlist

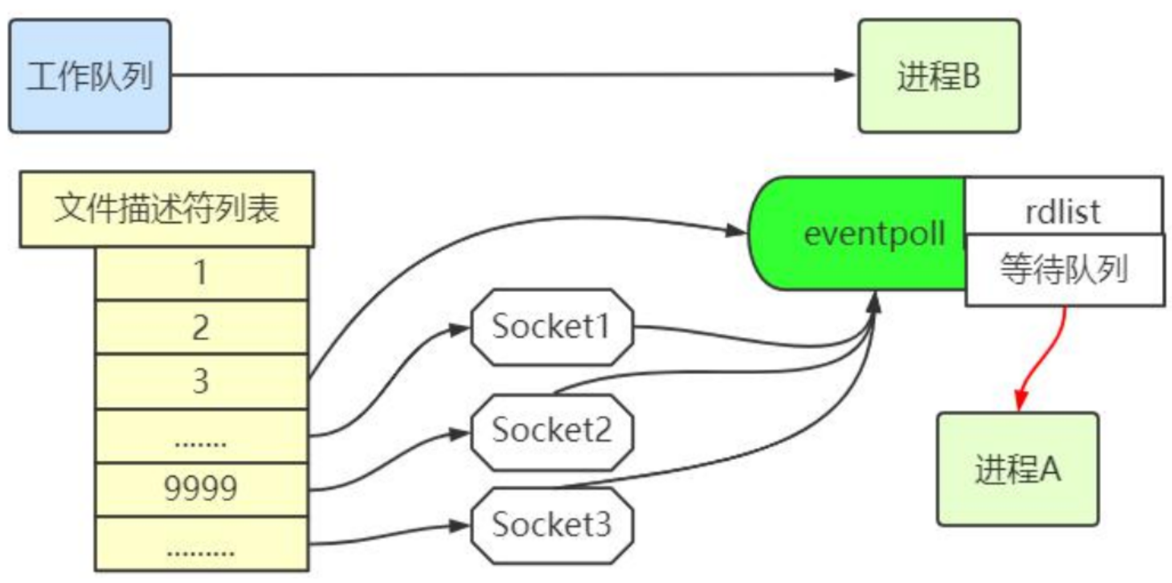
select 低效的另一个原因在于程序不知道哪些 socket 收到数据，只能一个个遍历。如果内核维护一个"就绪列表"rdlist，引用收到数据的 socket ，就能避免遍历。

如下的代码中，先用 epoll_create 创建一个 epoll 对象 epfd， 再通过 epoll_ctl 将需要监视的 socket 添加到 epfd 的专用等待列表中，最后调用epoll_wait 等待数据，返回rdlist列表中的就绪socket。

```
int epfd = epoll_create(...);
epoll_ctl(epfd, ...); //第一步：将所有需要监听的 socket 添加到 epfd 中等待队列

while(1){
   int n = epoll_wait(...)  //第二步：阻塞进程，等待事件
   for(接收到数据的 socket){
     //处理
   }
}
```

假设计算机中正在运行进程 A 和进程 B， 在某时刻进程 A 运行到了 epoll_wait 语句。

内核会将进程 A 放入 eventpoll 的等待队列中，阻塞进程。

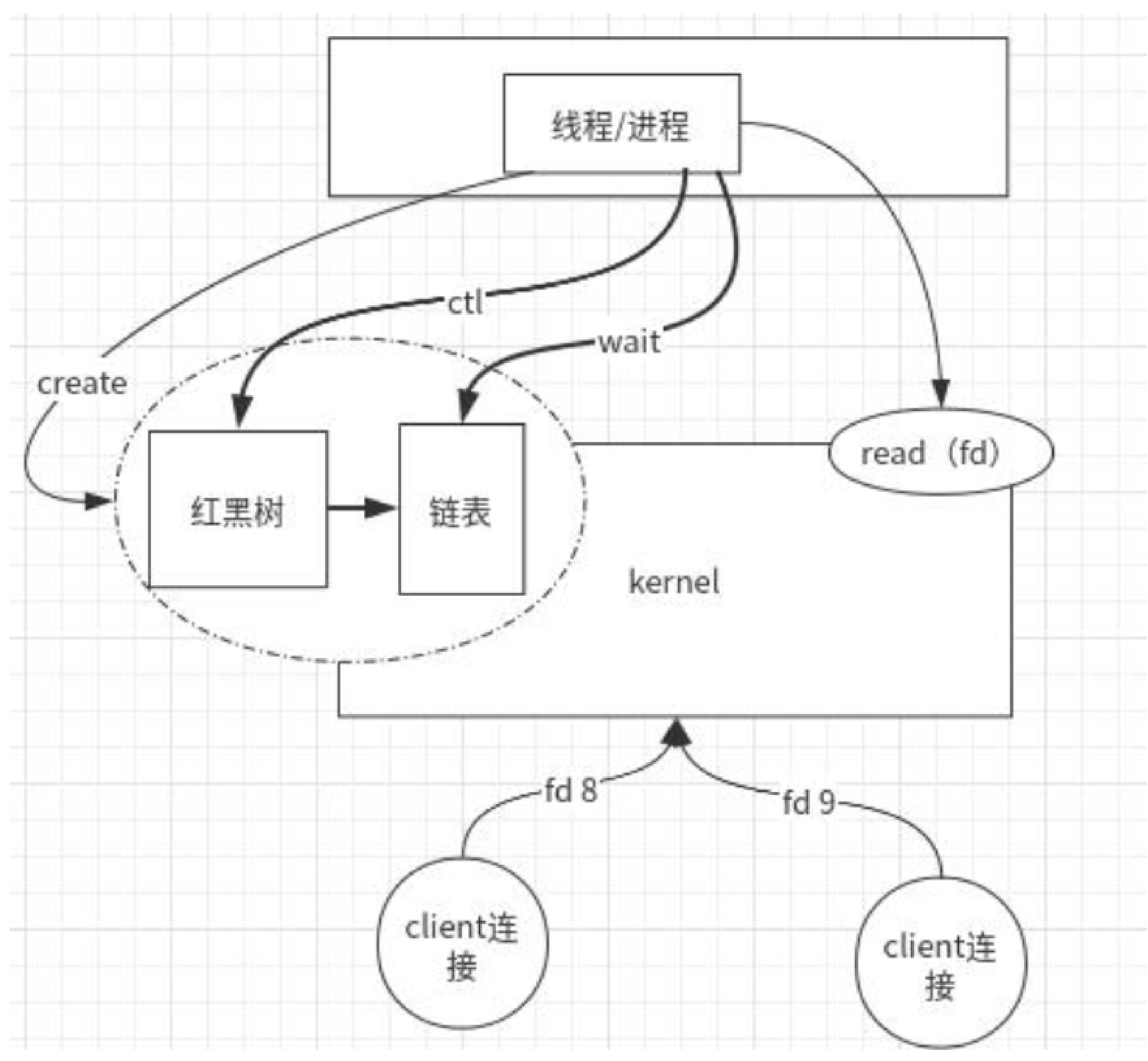


当 socket 接收到数据，中断程序做两个工作：

- 一方面修改 rdlist
- 另一方面唤醒 eventpoll 等待队列中的进程，进程 A 再次进入运行状态。

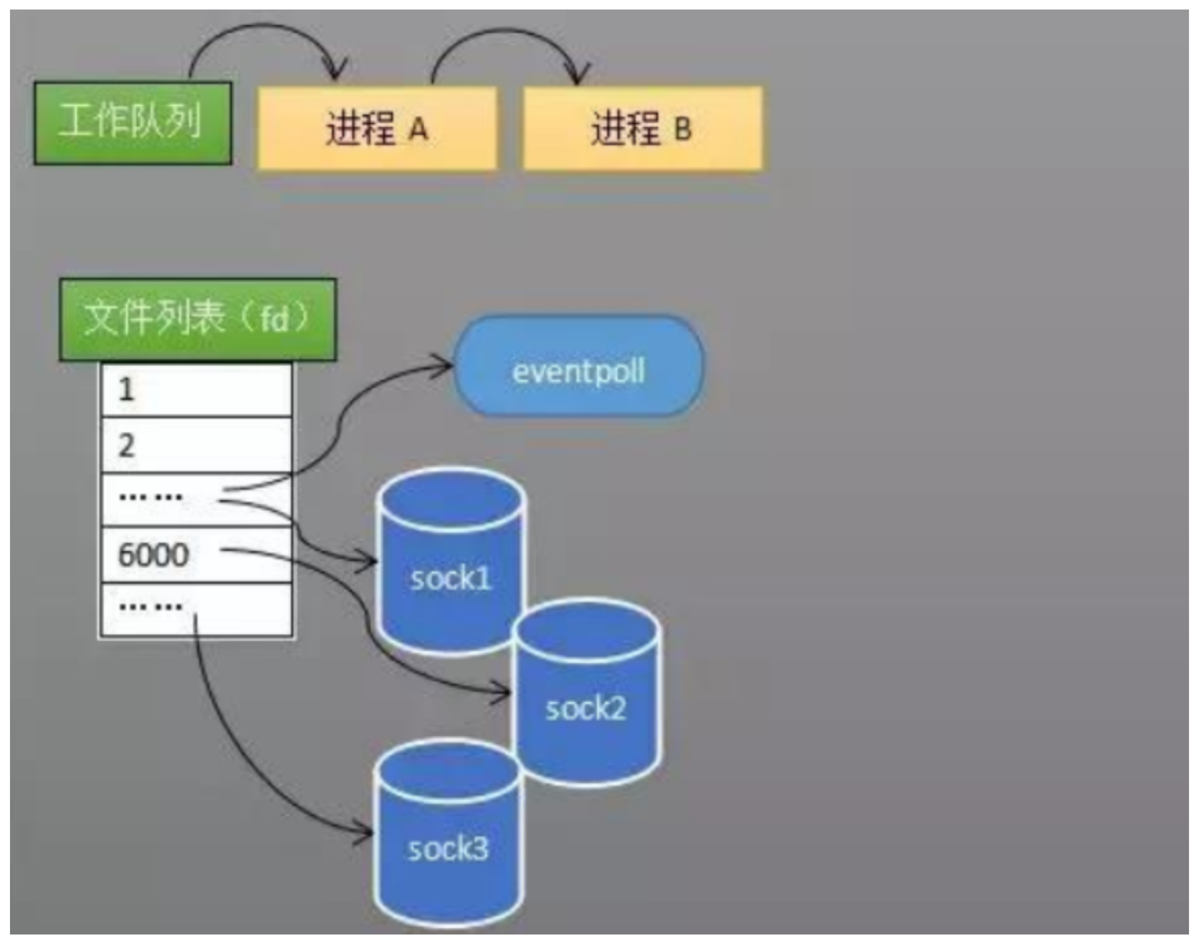
也因为 rdlist 的存在，进程 A 可以知道哪些 socket 发生了变化。

Epoll三大步骤



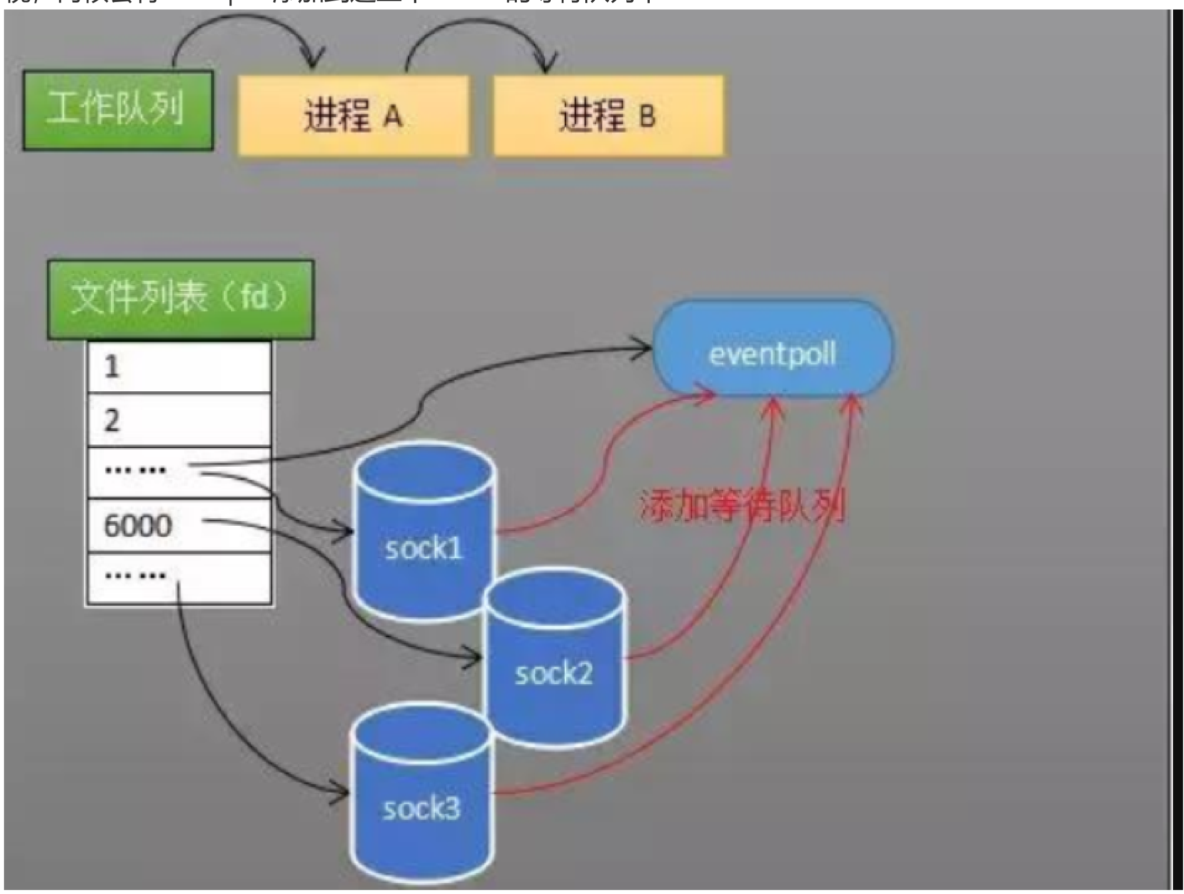
Epoll第一步epoll_create：

当某个进程调用epoll_create方法时，内核会创建一个eventpoll对象（epfd文件描述符），和socket一样，它也会有等待队列。

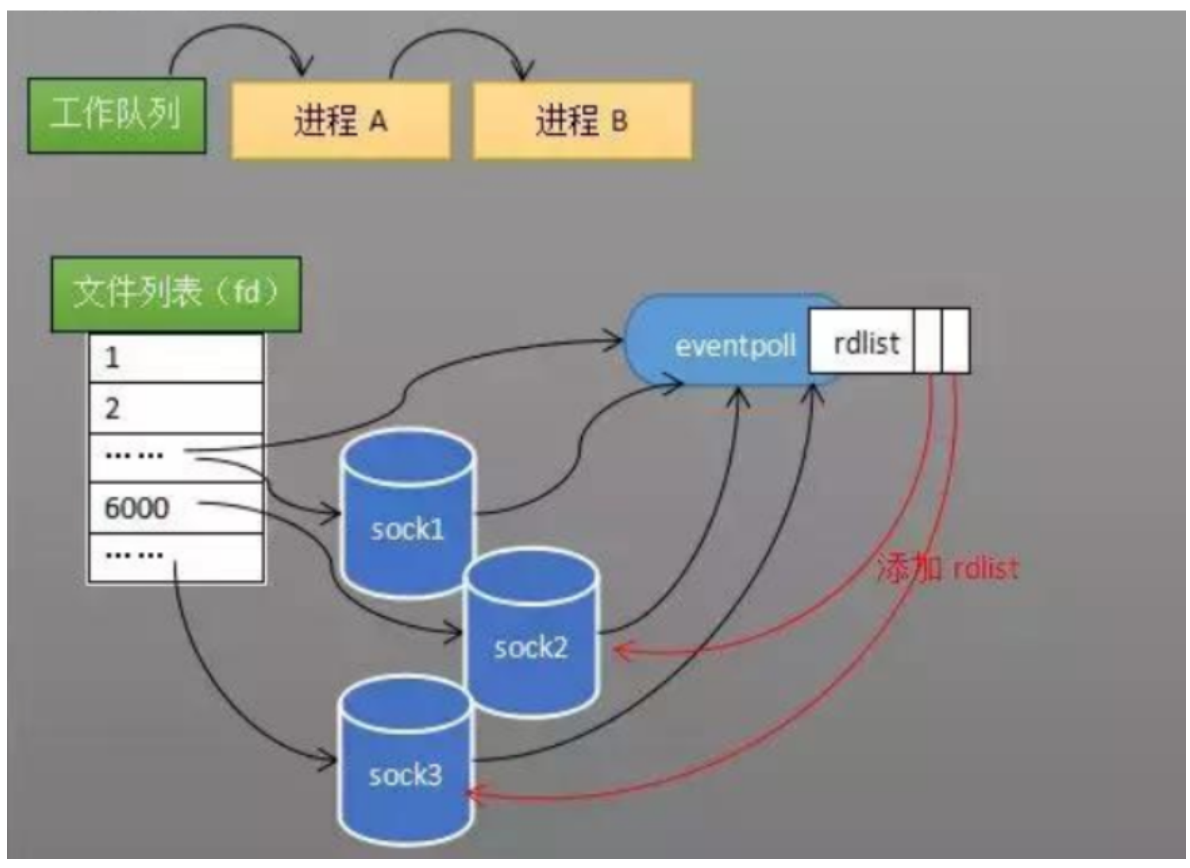


Epoll第2步epoll_ctl：

可以用epoll_ctl添加或删除所要监听的socket。eg：如果通过epoll_ctl添加sock1、sock2和sock3的监视，内核会将eventpoll添加到这三个socket的等待队列中

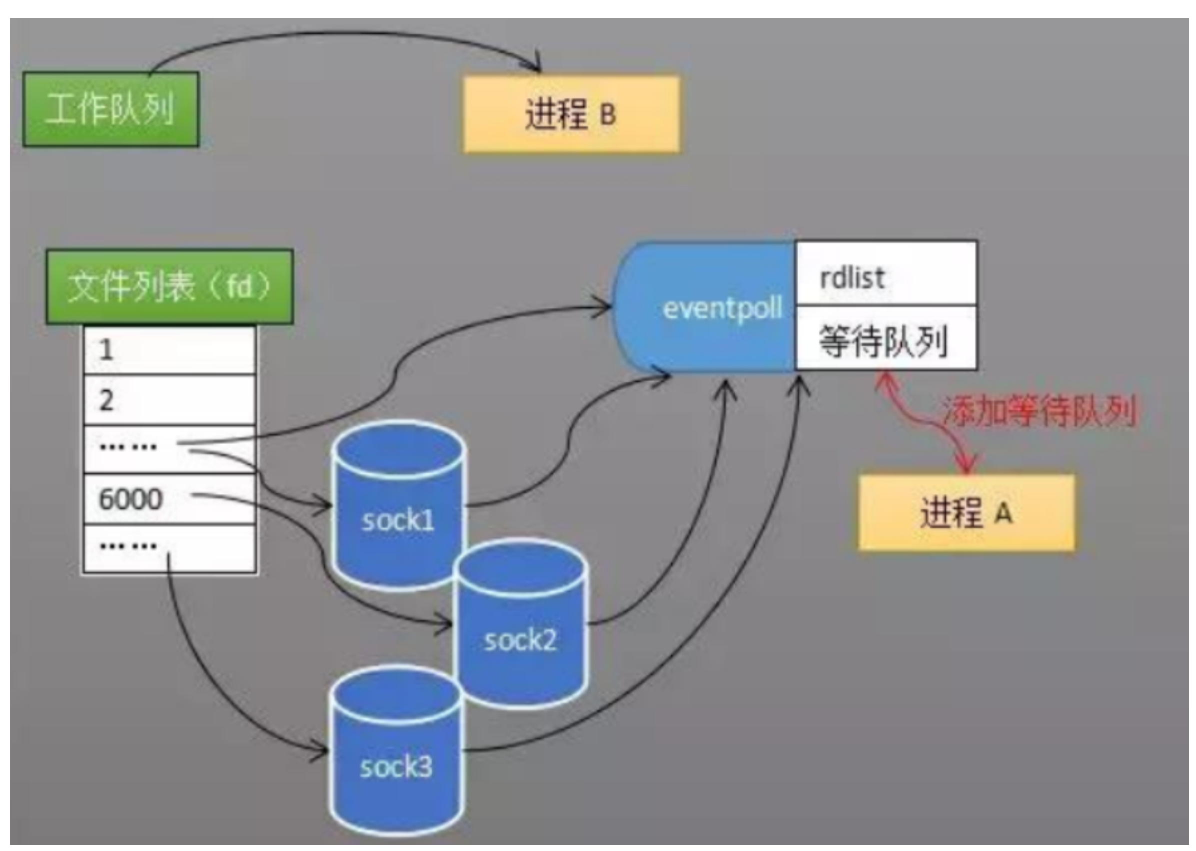


当socket收到数据后，中断程序会操作eventpoll的就行队列rdlist，而不是直接操作读取数据的进程（如果进行A）。当sock2和sock3收到数据后，中断程序让这两个socket进入rdlist。

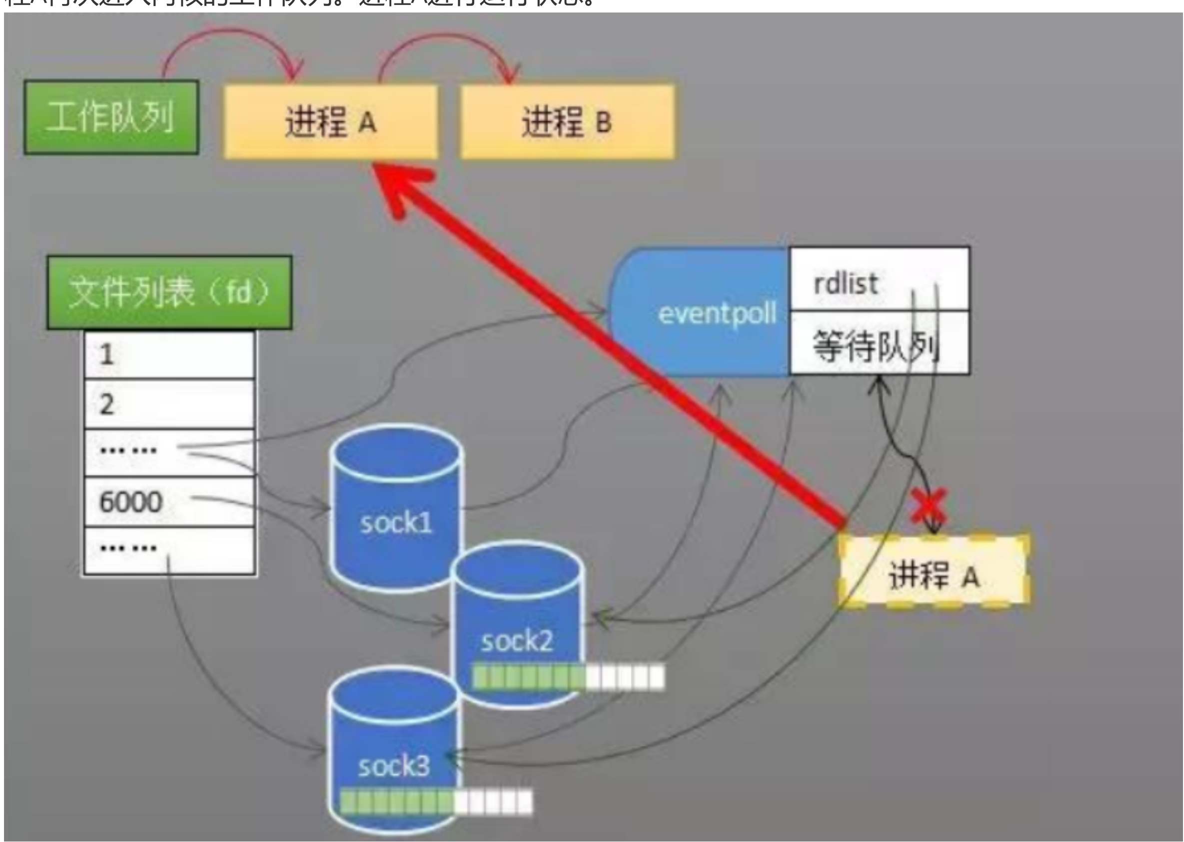


Epoll第3步epoll_wait：

当程序执行到epoll_wait时，如果rdlist非空则返回，如果rdlist为空，阻塞进程.



当socket接收到数据，中断程序一方面将其插入rdlist，另一方面唤醒eventpoll等待队列中的进程，进程A再次进入内核的工作队列。进程A进行运行状态。



Linux epoll API:epoll_create

```
#include

int epoll_create(int size) //创建一个epoll的句柄。自从linux2.6.8之后，size参数是被忽略的。
```

需要注意的是，当创建好epoll句柄后，它就是会占用一个fd值，如果查看/proc/进程id/fd/, 能够看到这个fd的，所以在使用完epoll后，必须调用close()关闭，否则可能导致fd被耗尽。

```
cd /proc/进程id/fd/
 ls -l
lrwx--- 1 root root 64 Nov 21 09:44 133 -> /dev/sda1
lrwx--- 1 root root 64 Nov 21 09:44 134 -> /dev/sdb1
lrwx--- 1 root root 64 Nov 21 09:44 136 -> /dev/sdb1
lrwx--- 1 root root 64 Nov 21 09:44 137 -> socket:[22460]
lrwx--- 1 root root 64 Nov 21 09:44 138 -> socket:[7326842]
lrwx--- 1 root root 64 Nov 21 09:44 139 -> socket:[7341066]
```

socket:后面的一串数字是什么呢？其实是该socket的inode号

Linux epoll API: epoll_ctl

epoll的事件注册函数，先注册要监听的事件类型。

```
#include

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

- 第一个参数是epoll_create()的返回值 epollfd句柄epfd。
- 第二个参数表示动作，用三个宏来表示：

EPOLL_CTL_ADD：注册新的fd到epfd中；

EPOLL_CTL_MOD：修改已经注册的fd的监听事件；

EPOLL_CTL_DEL：从epfd中删除一个fd；

- 第三个参数是需要监听的fd，比如说socket A、socket B、socket C。
- 第四个参数是告诉内核需要监听什么事 ,使用epoll_event结构

---

struct epoll_event结构如下：

```
struct epoll_event {
    __uint32_t events; /* Epoll events */
    epoll_data_t data; /* User data variable */
};
```

---

epoll的events事件类型

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；
- 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

---

epoll_event结构data成员变量：

是一个union类型的变量，类型定义如下

```
typedef union epoll_data {
    void *ptr;
    int fd;
    __uint32_t u32;
    __uint64_t u64;
} epoll_data_t;
```

union如果该结构对象属于动态存储类型，其成员具有不确定、灵活的初始值

Linux epoll API: epoll_wait

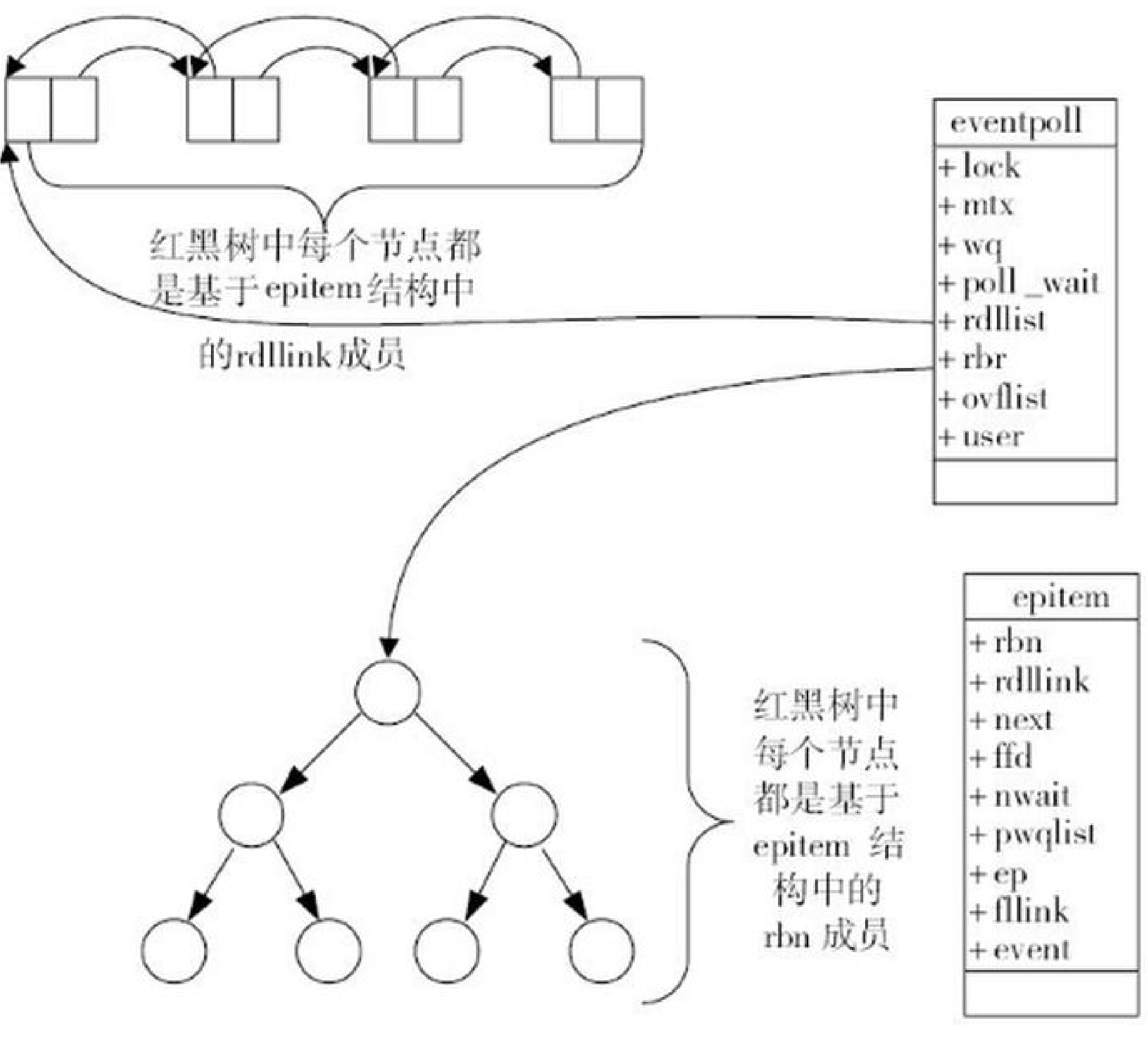
等待epoll监控的事件中已经发送的事件，类似于select()调用。

```
#include

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

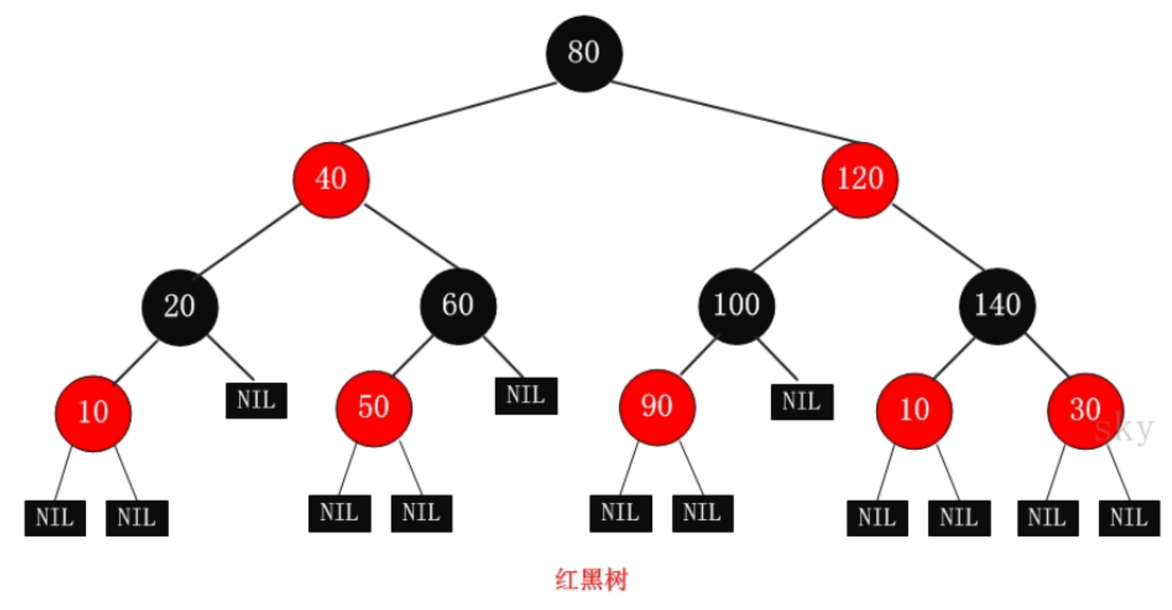
- 第二个参数events用来从内核得到事件的集合。epoll将会把发生的事件赋值到events数组中（events不可以是空指针，内核只负责把数据复制到这个events数组中，不会去帮助我们在用户态中分配内存）。
- 第三个参数maxevents告之内核这个events有多大，这个 maxevents的值不能大于创建epoll_create()时的size，
- 第四个参数参数timeout是超时时间（毫秒，0会立即返回，-1将不确定，也有说法说是永久阻塞）。该函数返回需要处理的事件数目，如返回0表示已超时。

eventpoll数据结构

eventpoll的rbr是一颗红黑树，存放所有的socket;rdlist是一个双向链表，存放已经发生IO事件的socket; 等待队列为poll_wait， 进程A放在poll_wait里;



红黑树是一种自平衡二叉查找树，搜索、插入和删除时间复杂度都是O(log(N))，效率较好。eventpoll的rbr等待队列是一颗红黑树，监听所有socket的IO事件。

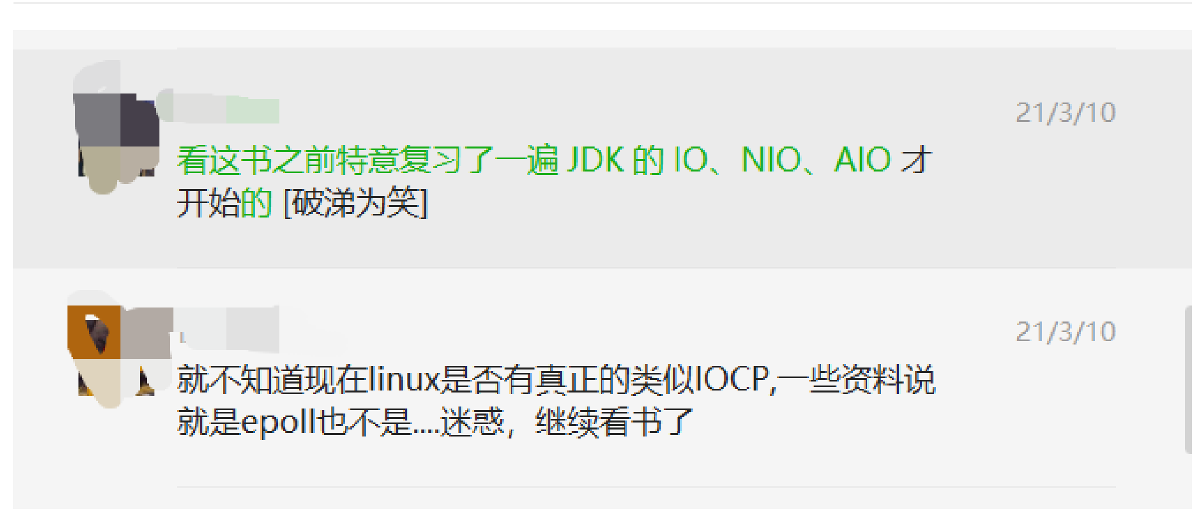


红黑树

正是因为epoll采用了红黑树的结构，所以能够监听的连接数，可以远远大于1024个

彻底明白：高速ET模式与Netty的高速Selector

背景



学习高并发，epoll是个基础

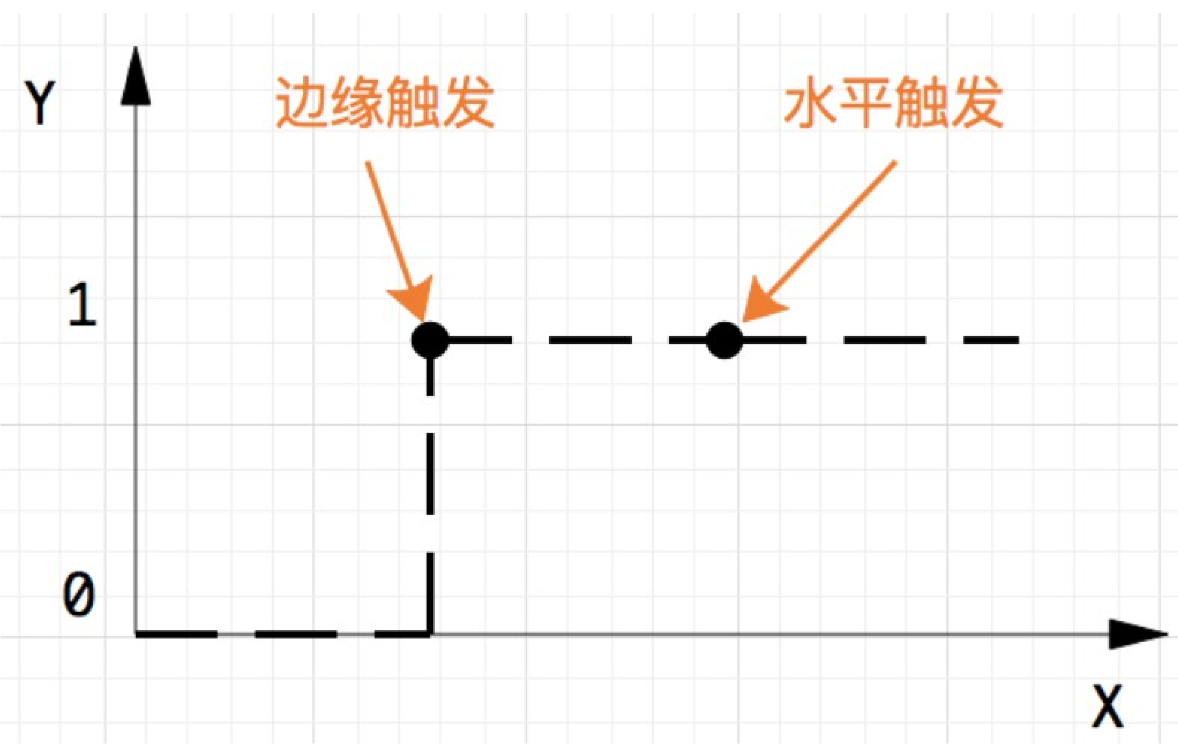
epoll事件的两种触发模式

epoll有EPOLLLT和EPOLLET两种触发模式

- LT是默认的模式
- ET是“高速”模式。

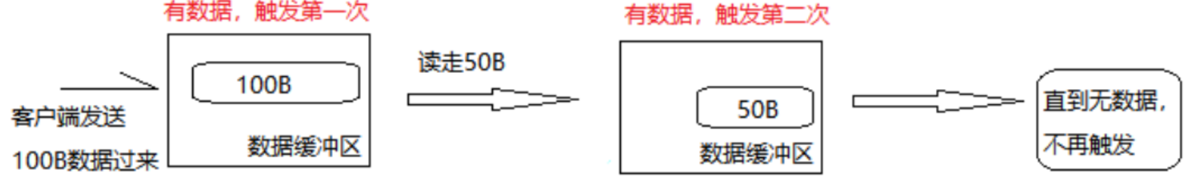
ET和LT，来自电子的概念

- ET 边沿触发：当电平出现变化的时候才触发事件。
- LT 水平触发：只要存在高电平就一直触发事件。



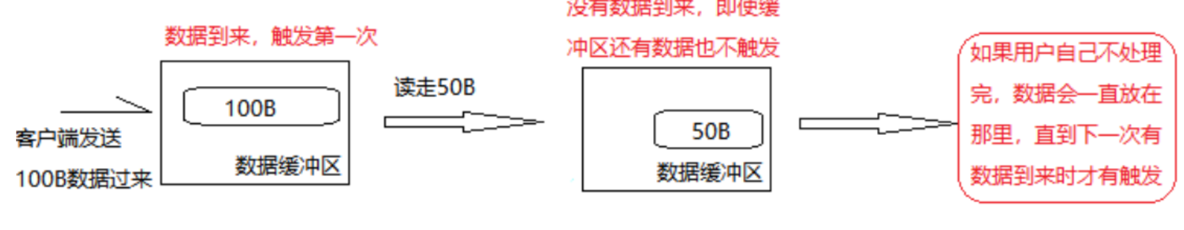
执行epoll_wait时LT（水平触发）

LT（水平触发）模式下，只要这个文件描述符还有数据可读，每次 **epoll_wait**都会返回它的事件，提醒用户程序去操作；



执行epoll_wait时**ET（边缘触发）模式**

ET（边缘触发）模式下，在它检测到有 I/O 事件时，通过**epoll_wait**调用会得到有事件通知的文件描述符，对于每一个被通知的文件描述符，如可读，则必须将该文件描述符一直读到空，让 errno 返回 EAGAIN 为止，否则下次的 epoll_wait 不会返回余下的数据，会丢掉事件。



如果ET模式**不是非阻塞**（OIO）的，那这个一直读或一直写势必会在最后一次阻塞。

epoll_wait时ET和LT对比

- 边缘触发（ET）：无论epoll事件是否处理完毕，仅触发一次

  缘触发（ET）的意思是执行epoll_wait时，内核检测到数据到达后立马返回到应用层。但是这仅仅只返回这一次，如果缓冲区中的数据没有读取完，再次执行epoll_wait时**不会继续触发**，需要下一次来数据了才能触发。

  > 也就是说，一次数据不会重复发送到应用层，不管你是否读完了。

- 水平触发（LT）：水平触发（LT）的意思是只要存在高电平就一直触发事件，执行epoll_wait时，只要检测到有数据就返回。

  > 无论epoll事件是否处理完毕，只要事件没有处理完毕，每一次epoll_wait都触发该事件。也就是说，如果没有读完，一次数据会重复发送到应用层，每一次epoll_wait发送一次。

ET和LT的区别

ET和LT的区别在于触发事件的条件不同，LT比较符合编程思维（有满足条件的就触发），ET触发的条件更苛刻一些（仅在发生变化时才触发），对使用者的要求也更高，理论效率更高。

总之，ET的效率高于LT模式，因为产生的事件数更少，可以减少内核往应用层空间复制数据的次数。在进行高性能网络编程的时候，往往都是选择**非阻塞IO+ET触发模式**，这种模式下可以做到想读数据的时候就读，不想读就不读。

> epoll中的触发模式有两种，边缘触发和水平触发，**默认情况下使用的是水平触发**。

Java中的selector触发模式

值得一提的是java nio的selector会根据操作系统不同采用不同的实现

- 在linux 2.6以后的NIO的selector版本，则是`EPollSelectorProvider`，`EPollSelectorProvider`底层使用的是epoll，采用的是水平触发模式；
- 在macos下是`KQueueSelectorProvider`，`KQueueSelectorProvider`底层使用了kqueue来进行IO多路复用；
- 而netty中提供的额外的`EpollEventLoop`则采用了边缘触发。

Netty的Selector

虽然jdk自身提供了selector的epoll实现，netty仍实现了自己的epoll版本，根据netty开发者在StackOverflow的回答，主要原因有两个：

1. 支持更多socket option，比如TCP_CORK和SO_REUSEPORT
2. 使用了边缘触发（ET）模式

如何使用netty自己的Linux epoll实现

首先需要在项目中添加netty-transport-native-epoll依赖，因为这个不在netty核心包中

```
<dependencies>
  <dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-transport-native-epoll</artifactId>
    <version>${project.version}</version>
    <classifier>linux-x86_64</classifier>
  </dependency>
  ...
</dependencies>
```

如何使用netty自己的Linux epoll实现 step2：

然后将代码中的`NioEvnetLoop`、`NioSocketChannel`、`NioServerSocketChannel`等替换为Epoll开头的类即可

- NioEventLoopGroup → EpollEventLoopGroup
- NioEventLoop → EpollEventLoop
- NioServerSocketChannel → EpollServerSocketChannel
- NioSocketChannel → EpollSocketChannel

如何使用netty自己的Linux epoll实现,详细地址：

https://netty.io/wiki/native-transports.html

netty.io/wiki/native-transports.html

Netty project News Downloads Documentation Get Involved

Native transports

Did you know this page is automatically generated from a Github Wiki page? You can improve it by yourself here!

Netty provides the following platform specific JNI transports:

- Linux (since 4.0.16)
- MacOS/BSD (since 4.1.11)

These JNI transports add features specific to a particular platform, generate less garbage, and generally improve performance when compared to the NIO based transport.

Using the native transports

Note that you must specify the proper classifier for the dependency to ensure that the corresponding native libraries are included.

Using the Linux native transport

Because the native transport is compatible with the NIO transport, you can just do the following search-and-replace:

- `NioEventLoopGroup` → `EpollEventLoopGroup`
- `NioEventLoop` → `EpollEventLoop`
- `NioServerSocketChannel` → `EpollServerSocketChannel`
- `NioSocketChannel` → `EpollSocketChannel`

Because the native transport is not part of the Netty core, you need to pull the `netty-transport-native-epoll` as a dependency in your Maven `pom.xml`:

彻底解密：IO事件在Java、Native之间的翻译

背景：

不同语言，不同的IO事件的表达

> 这是一个非常基础的背景知识

Java的IO事件在类型

SelectionKey中的事件常量。可以监听四种不同类型的事件（这四种事件用SelectionKey的四个常量来表示）：

- SelectionKey.OP_CONNECT “ **连接就绪** ”（连接事件）
- SelectionKey.OP_ACCEPT “ **接收就绪** ”（新连接接收事件）
- SelectionKey.OP_READ “ **读就绪** ” （通道读取缓冲区可读）
- SelectionKey.OP_WRITE “ **写就绪** ” （通道写入缓冲区可读）

SelectionKey中的事件常量

```
//读操作符，左移位后的整型值为1
public static final int OP_READ = 1 << 0;
//写操作符，左移位后的整型值为4
public static final int OP_WRITE = 1 << 2;
//连接操作符，左移位后的整型值为8
public static final int OP_CONNECT = 1 << 3;
//接收操作符，左移位后的整型值为16
public static final int OP_ACCEPT = 1 << 4;
```

OP_CONNECT表示某个Channel成功连接到服务器。

OP_ACCEPT表示一个Server Socket Channel准备好接收新进入的连接称为。

OP_READ一个有数据可读的通道可以说是“ **读就绪** ”。

OP_WRITE 表示channel等待写数据，或者说可以写入数据，就是通道的写入缓冲区大于**低水位SO_SNDLOWAT的值（默认为2048）**，而**套接字默认的发送缓冲区大小是8k**，所以总是可写的。

---

poll() 系统调用的事件

---

pollfd结构

成员变量说明：
（1）fd：每一个 pollfd 结构体指定了一个被监视的文件描述符，可以传递多个结构体，指示 poll() 监视多个文件描述符。
（2）events：表示要告诉操作系统需要监测fd的事件（输入、输出、错误），每一个事件有多个取值
（3）revents：revents 域是文件描述符的操作结果事件，内核在调用返回时设置这个域。events 域中请求的任何事件都可能在 revents 域中返回。

```
struct pollfd {
    int   fd;           /* file descriptor */
    short events;       /* requested events */
    short revents;      /* returned events */
};
```

---

poll的events事件类型

事件	描述	是否可作为输入（events）	是否可作为输出（revents）
POLLIN	数据可读（包括普通数据&优先数据）	是	是
POLLOUT	数据可写（普通数据&优先数据）	是	是
POLLRDNORM	普通数据可读	是	是
POLLRDBAND	优先级带数据可读（linux不支持）	是	是
POLLPRI	高优先级数据可读，比如TCP带外数据	是	是
POLLWRNORM	普通数据可写	是	是
POLLWRBAND	优先级带数据可写	是	是
POLLRDHUP	TCP连接被对端关闭，或者关闭了写操作，由GNU引入	是	是
POPPHUP	挂起	否	是
POLLERR	错误	否	是
POLLNVAL	文件描述符没有打开	否	是

---

socket读就绪条件(读事件)：

1. 该套接字接收缓冲区中的数据字节数大于等于套接字接收缓存区**低水位标记SO_RCVLOWAT。**

   对于TCP和UDP套接字而言，**缓冲区低水位的值默认为1。**那就意味着，默认情况下，只要缓冲区中有数据，那就是可读的。我们可以通过使用SO_RCVLOWAT套接字选项(参见setsockopt函数)来设置该套接字的低水位大小。此种描述符就绪(可读)的情况下，当我们使用read/recv等对该套接字执行读操作的时候，套接字不会阻塞，而是成功返回一个大于0的值（即可读数据的大小）。
2. **该连接的读半部关闭**（也就是接收了FIN的TCP连接）。对这样的套接字的读操作，将不会阻塞，而是返回0（也就是EOF）。
3. 该套接字是一个listen的**监听套接字**，并且目前已经完成的连接数不为0。对这样的套接字进行accept操作通常不会阻塞。
4. 有一个**错误套接字待处理。**对这样的套接字的读操作将不阻塞并返回-1（也就是返回了一个错误），同时把errno设置成确切的错误条件。这些待处理错误(pending error)也可以通过指定SO_ERROR套接字选项调用getsockopt获取并清除。

---

socket写就绪条件(写事件):

1. socket内核中, 发送缓冲区中的可用字节数(发送缓冲区的空闲位置大小), **大于等于低水位标记 SO_SNDLOWAT**, 此时可以无阻塞的写, 并且返回值大于0。

   对于TCP和UDP而言，这个**低水位SO_SNDLOWAT的值默认为2048**，而**套接字默认的发送缓冲区大小是8k**，这就意味着一般一个套接字连接成功后，就是处于可写状态的。我们可以通过SO_SNDLOWAT套接字选项（参见setsockopt函数）来设置这个低水位。此种情况下，我们设置该套接字为非阻塞，对该套接字进行写操作(如write,send等)，将不阻塞，并返回一个正值（例如由传输层接受的字节数，即发送的数据大小）。

2. **该连接的写半部关闭**(主动发送FIN包的TCP连接)。对这样的套接字的写操作将会产生SIGPIPE信号。所以我们的网络程序基本都要自定义处理SIGPIPE信号。因为SIGPIPE信号的默认处理方式是程序退出。
3. 使用非阻塞的connect套接字已建立连接，或者connect已经以失败告终。即connect有结果了。
4. 有一个错误的套接字待处理。对这样的套接字的写操作将不阻塞并返回-1（也就是返回了一个错误），同时把errno设置成确切的错误条件。这些待处理的错误也可以通过指定SO_ERROR套接字选项调用getsockopt获取并清除。

条　件	可读吗？	可写吗？	异常吗？
有数据可读	•		
关闭连接的读一半	•		
给监听套接口准备好新连接	•		
有可用于写的空间		•	
关闭连接的写一半		•	
待处理错误	•	•	
TCP带外数据			•

---

poll() 系统调用的本质

select() 和 poll() 系统调用的本质一样

poll() 的机制与 select() 在本质上没有多大差别，每次调用时，都需要把 fd 集合从用户态拷贝到内核态，

二者管理多个描述符也是进行轮询，根据描述符的状态进行处理。

Net.java通过JNI加载不同平台的poll事件的定义值

```
    public static final short POLLIN;
    public static final short POLLOUT;
    public static final short POLLERR;
    public static final short POLLHUP;
    public static final short POLLNVAL;
    public static final short POLLCONN;

    static native short pollinValue();
    static native short polloutValue();
```

```
    static native short pollerrValue();
    static native short pollhupValue();
    static native short pollnvalValue();
    static native short pollconnValue();

    static {
        IOUtil.load();
        initIDs();

        POLLIN     = pollinValue();
        POLLOUT    = polloutValue();
        POLLERR    = pollerrValue();
        POLLHUP    = pollhupValue();
        POLLNVAL   = pollnvalValue();
        POLLCONN   = pollconnValue();
    }
```

Net.c中的事件的值

这些也对应c中的select轮询事件,windows的Net.c中值如下

```
JNIEXPORT jshort JNICALL
Java_sun_nio_ch_Net_pollinValue(JNIEnv *env, jclass this)
{
    return (jshort)POLLIN;
}

JNIEXPORT jshort JNICALL
Java_sun_nio_ch_Net_polloutValue(JNIEnv *env, jclass this)
{
    return (jshort)POLLOUT;
}

JNIEXPORT jshort JNICALL
Java_sun_nio_ch_Net_pollerrValue(JNIEnv *env, jclass this)
{
    return (jshort)POLLERR;
}

JNIEXPORT jshort JNICALL
Java_sun_nio_ch_Net_pollhupValue(JNIEnv *env, jclass this)
{
    return (jshort)POLLHUP;
}

JNIEXPORT jshort JNICALL
Java_sun_nio_ch_Net_pollnvalValue(JNIEnv *env, jclass this)
{
    return (jshort)POLLNVAL;
}

JNIEXPORT jshort JNICALL
Java_sun_nio_ch_Net_pollconnValue(JNIEnv *env, jclass this)
{
    return (jshort)POLLCONN;
```

```
}
```

NIO事件到JNI事件的翻译

- 注册时的翻译：从NIO事件到JNI事件

SocketChannelImpl#translateInterestOps

- 查询时的翻译：从JNI事件到NIO事件

SocketChannelImpl#translateReadyOps

SocketChannelImpl#translateInterestOps

```
/**
 * Translates an interest operation set into a native poll event set
 */
public int translateInterestOps(int ops) {
    int newOps = 0;
    if ((ops & SelectionKey.OP_READ) != 0)
        newOps |= Net.POLLIN;
    if ((ops & SelectionKey.OP_WRITE) != 0)
        newOps |= Net.POLLOUT;
    if ((ops & SelectionKey.OP_CONNECT) != 0)
        newOps |= Net.POLLCONN;
    return newOps;
}
```

SocketChannelImpl#translateReadyOps

```
/**
 * Translates native poll revent ops into a ready operation ops
 */
public boolean translateReadyOps(int ops, int initialOps, SelectionKeyImpl ski) {
    int intOps = ski.nioInterestOps();
    int oldOps = ski.nioReadyOps();
    int newOps = initialOps;
    ....
    boolean connected = isConnected();
    if (((ops & Net.POLLIN) != 0) &&
        ((intOps & SelectionKey.OP_READ) != 0) && connected)
        newOps |= SelectionKey.OP_READ;
    ....

    if (((ops & Net.POLLOUT) != 0) &&
        ((intOps & SelectionKey.OP_WRITE) != 0) && connected)
        newOps |= SelectionKey.OP_WRITE;

    ski.nioReadyOps(newOps);
    return (newOps & ~oldOps) != 0;
}
```

epoll() 系统调用的事件

struct epoll_event结构如下：

```
struct epoll_event {
    __uint32_t events; /* Epoll events */
    epoll_data_t data; /* User data variable */
};
```

---

epoll的events事件类型

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；
- 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

---

epoll_event结构data成员变量：

是一个union类型的变量，类型定义如下

```
typedef union epoll_data {
    void *ptr;
    int fd;
    __uint32_t u32;
    __uint64_t u64;
} epoll_data_t;
```

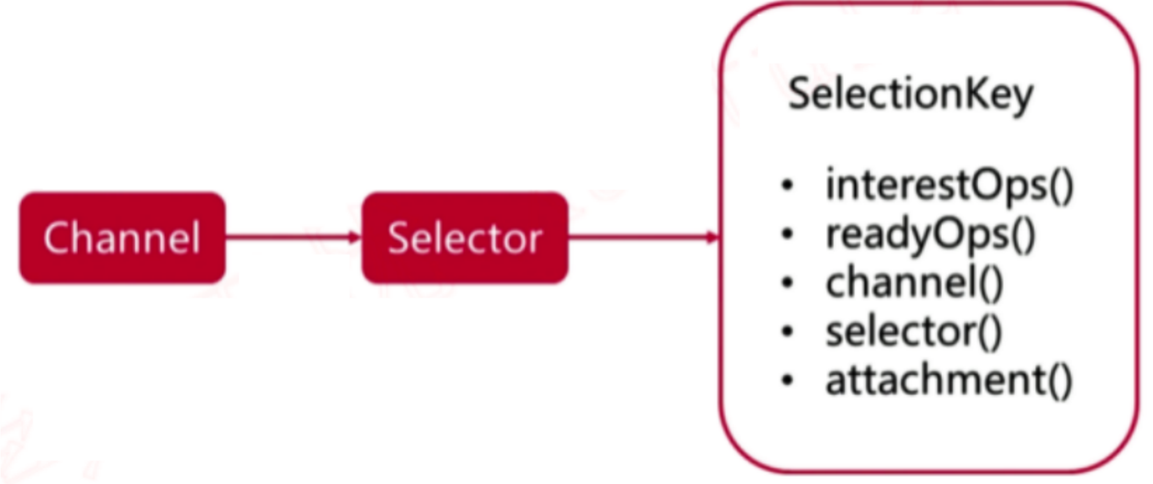
union如果该结构对象属于动态存储类型，其成员具有不确定、灵活的初始值

Netty使用了epoll的ET模式实现高速IO事件的查询

彻底解密：SelectionKey (选择键) 核心原理

SelectionKey介绍

一个SelectionKey键表示了一个**channel通道对象**和一个**selector选择器**对象之间的注册关系。



SelectionKey对象代表着Channel和Selector的关联关系

大家都习惯讲关联关系。SelectionKey对象代表着一个Channel和它注册的Selector间的关联关系。

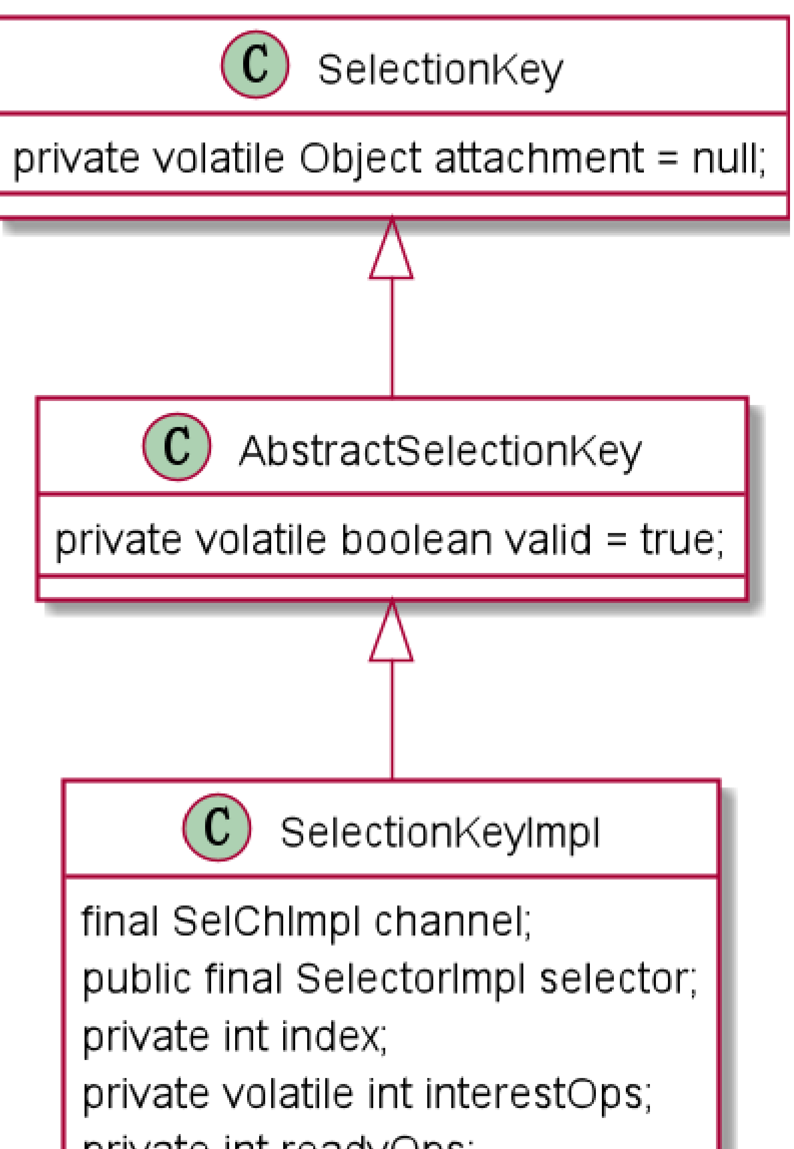
SelectionKey#channel( )方法可返回与该键相关的SelectableChannel对象；而SelectionKey#selector( )则返回相关的Selector对象。

此外，SelectionKey中包含两个重要属性，两个以整数形式进行编码的比特掩码：

- interestOps：代表对注册Channel所感兴趣的事件集合。interest集合是使用注册通道时给定的值初始化的，可以通过调用键对象的interestOps(int ops)方法修改。同时，可以调用键对象的interestOps()方法获取当前interest集合。当相关的Selector上的select( )操作正在进行时改变键的interest集合，不会影响那个正在进行的选择操作。所有更改将会在select( )的下一个调用中体现出来；
- readyOps：代表interest集合中从上次调用select( )以来已经就绪的事件集合，它是interestOps的子集。注册通道时，初始化为0，只有在选择器选择操作期间可能被更新。可以调用键对象的readyOps()方法获取当前ready集合。需注意的是ready集合返回的就绪状态只是一个提示，不是保证。底层的通道在任何时候都会不断改变。其他线程可能在通道上执行操作并影响它的就绪状态。

SelectionKeyImpl继承关系

SelectionKey<|-- AbstractSelectionKey<|-- SelectionKeyImpl



向通道注册事件

注册通道时，如果我们不止对一种操作感兴趣，可以用“位或”操作符将多个常量连接起来。如下：

```
socketChannel.register(selector,
SelectionKey.OP_CONNECT|SelectionKey.OP_READ|SelectionKey.OP_WRITE);
```

SelectionKey中使用了四个常量来代表事件类型：

SelectionKey.OP_READ：通道已经准备好进行读取；

SelectionKey.OP_WRITE：通道已经准备好写入；

SelectionKey.OP_CONNECT：通道对应的socket已经准备好连接；

SelectionKey.OP_ACCEPT：通道对应的server socket已经准备好接受一个新连接。

SelectionKey维护两个set集合成员：

- interestOps

希望Selector监听Channel的哪些事件。

- readyOps

已经发生了的事件

```
public class SelectionKeyImpl extends AbstractSelectionKey {
    final SelChImpl channel;
    public final SelectorImpl selector;
    private int index;
    private volatile int interestOps;
    private int readyOps;
```

每个Selector中还维护了publicKeys和publicSelectedKeys两个视图，供客户端使用。

- publicKeys
  publicKeys是keys的视图，

调用Selector的keys()方法返回的就是publicKeys，publicKeys不支持添加和删除操作；

- publicSelectedKeys

调用Selector的selectedKeys()方法访问publicSelectedKeys。

publicSelectedKeys是selectedKeys的视图，它是是个不可增长的集合，即不支持add操作，但支持remove操作

> 调用publicSelectedKeys集合的remove操作实际是从selectedKeys中删除一个SelectionKey对象。

Collections.unmodifiableSet & ungrowableSet

```java
public abstract class SelectorImpl extends AbstractSelector {
    protected Set<SelectionKey> selectedKeys = new HashSet();
    protected HashSet<SelectionKey> keys = new HashSet();
    private Set<SelectionKey> publicKeys;
    private Set<SelectionKey> publicSelectedKeys;

    protected SelectorImpl(SelectorProvider var1) {
        super(var1);
        if (Util.atBugLevel( s: "1.4")) {
            this.publicKeys = this.keys;
            this.publicSelectedKeys = this.selectedKeys;
        } else {
            this.publicKeys = Collections.unmodifiableSet(this.keys);
            this.publicSelectedKeys = Util.ungrowableSet(this.selectedKeys);
        }

    }
```

SelectionKey#interestOps()方法

我们可以通过以下方法来判断Selector是否对Channel的某种事件感兴趣

```
int interestSet = selectionKey.interestOps();
boolean isInterestedInAccept = (interestSet & SelectionKey.OP_ACCEPT) ==
SelectionKey.OP_ACCEPT；
boolean isInterestedInConnect = interestSet & SelectionKey.OP_CONNECT;
boolean isInterestedInRead = interestSet & SelectionKey.OP_READ;
boolean isInterestedInWrite = interestSet & SelectionKey.OP_WRITE;
```

SelectionKey#.readyOps()

ready 集合是通道已经准备就绪的事件的集合。JAVA中定义以下几个方法用来检查这些操作是否就绪.

```
//创建ready集合的方法
int readySet = selectionKey.readyOps();
//检查这些操作是否就绪的方法
key.isAcceptable();//是否可接收，是返回 true
boolean isWritable()：//是否可写，是返回 true
boolean isConnectable()：//是否可连接，是返回 true
boolean isAcceptable()：//是否可接收，是返回 true
```

从SelectionKey导航到Channel和Selector很简单。如下：

```
Channel channel = key.channel();
Selector selector = key.selector();
key.attachment();
```

为SelectionKey添加附件

可以将一个对象或者更多信息附着到SelectionKey上，这样就能方便的进行事件回调操作。例如，可以附加 与通道一起使用的Buffer，或是包含聚集数据的某个对象。使用方法如下：

```
key.attach(attachedObj); // 附着到SelectionKey上
...
Object attachedObj = key.attachment();
```

注册Channel的时候添加附件

可以在用register()方法向Selector注册Channel的时候附加对象

```
SelectionKey key = channel.register(selector, SelectionKey.OP_READ,
attachedObj);
```

SelectionKey除开构造器方法，有13个方法：

```
public abstract SelectableChannel channel();//返回该SelectionKey对应通道
public abstract Selector selector();//返回该SelectionKey注册的选择器
public abstract boolean isValid();//判断该SelectionKey是否有效
public abstract void cancel();//撤销该SelectionKey
public abstract int interestOps();//返回SelectionKey的关注操作符
//设置该SelectionKey的关注键，返回更改后新的SelectionKey
public abstract SelectionKey interestOps(int ops);
public abstract int readyOps();//返回SelectionKey的预备操作符

//这里readyOps()方法返回的是该SelectionKey的预备操作符，至于什么是预备操作符在最终实现类
SelectionKeyImpl中会讲解。
//判断该SelectionKey的预备操作符是否是OP_READ
public final boolean isReadable() {
    return (readyOps() & OP_READ) != 0;
}
//判断该SelectionKey的预备操作符是否是OP_WRITE
public final boolean isWritable() {
    return (readyOps() & OP_WRITE) != 0;
}
//判断该SelectionKey的预备操作符是否是OP_CONNECT
public final boolean isConnectable() {
    return (readyOps() & OP_CONNECT) != 0;
}
//判断该SelectionKey的预备操作符是否是OP_ACCEPT
public final boolean isAcceptable() {
    return (readyOps() & OP_ACCEPT) != 0;
}

//设置SelectionKey的附件
public final Object attach(Object ob) {
```

```
    return attachmentUpdater.getAndSet(this, ob);
}
//返回SelectionKey的附件
public final Object attachment() {
    return attachment;
}
```

AbstractSelectionKey的属性

AbstractSelectionKey继承了SelectionKey类，它也是一个抽象类，相比其他俩个类，它的代码就简洁多了，它只有一个属性：

```
//用于判断该SelectionKey是否有效，true为有效，false为无效，默认有效
private volatile boolean valid = true;
```

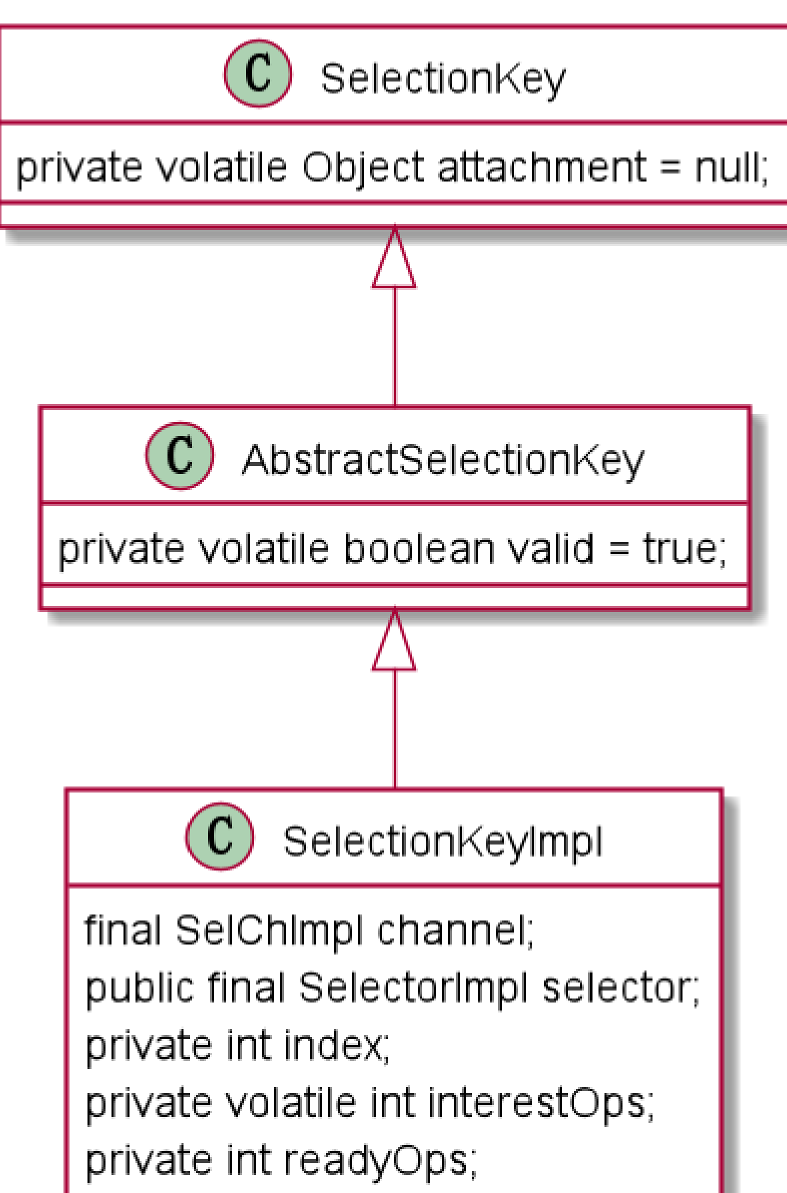
AbstractSelectionKey的方法

AbstractSelectionKey除开构造器方法，还有三个实现方法：

```
//判断该SelectionKey是否有效
public final boolean isValid() {
    return valid;
}
//将该SelectionKey设为无效
void invalidate() {
    valid = false;
}
//将该SelectionKey从选择器中删除
//注意，删除的SelectionKey并不会马上从选择器上删除，而是会加入一个需删除键的集合中，等到下一次调用选择方法才会将它从选择器中删除，至于具体实现会在选择器源码分析章节中讲
public final void cancel() {
    synchronized (this) {
    if (valid) {valid = false;((AbstractSelector)selector()).cancel(this);
        }
    }
}
```

SelectionKeyImpl

SelectionKeyImpl是SelectionKey的最终实现类，它继承了AbstractSelectionKey类，在该类中不仅实现了SelectionKey和抽象类的方法，而且扩展了其他方法。



SelectionKeyImpl的属性

SelectionKeyImpl中有5个新的属性:

```
final SelChImpl channel;  //该SelectionKey对应的通道

public final SelectorImpl selector;  //该SelectionKey注册的选择器

private int index;  //  SelectionKey集合中的下标索引,该SelectionKey在注册选择器中储存SelectionKey集合中的下标索引，当该SelectionKey被撤销时，index为-1

//SelectionKey的关注操作符
private volatile int interestOps; //感兴趣的事件interestOps
//SelectionKey的预备操作符
private int readyOps; //就绪的事件readyOps
```

从上面属性中可以看到，SelectionKeyImpl有俩个操作符属性：**感兴趣的事件interestOps**和**就绪的事件readyOps。**

感兴趣的事件interestOps可以通过channel通道的注册方法register(Selector sel, int ops)注册。

```
public final SelectionKey register(Selector sel, int ops, Object att) throws
ClosedChannelException
{
    synchronized (regLock) {
        if (!isOpen())
            throw new ClosedChannelException();
        if ((ops & ~validOps()) != 0)
            throw new IllegalArgumentException();
        if (blocking)
            throw new IllegalBlockingModeException();
        SelectionKey k = findKey(sel);
        if (k != null) {
            //将输入的参数ops储存在SelectionKey的interestOps属性中
            k.interestOps(ops);
            k.attach(att);
        }
        if (k == null) {
            synchronized (keyLock) {
                if (!isOpen())
                    throw new ClosedChannelException();
                k = ((AbstractSelector)sel).register(this, ops, att);
                addKey(k);
            }
        }
        return k;
    }
}
```

感兴趣的事件interestOps还可直接通过interestOps()方法设置

```
selectionKey.interestOps(SelectionKey.OP_READ | SelectionKey.OP_WRITE);
```

就绪的事件readyOps是通道实际发生的操作事件

当我们对选择器Selector.select()方法层层追溯，到达该方法的最终实现doSelect(long var1)方法，会发现doSelect方法调用了一个updateSelectedKeys()方法来更新选择器的SelectionKey集合，而updateSelectedKeys方法又调用了updateSelectedKeys(this.updateCount)方法来进行实际的更新操作，而updateSelectedKeys方法最终通过processFDSet方法来实现更新。在processFDSet方法里有两个方法调用实现了SelectionKey里readyOps属性的更新：**translateAndSetReadyOps(用于设置readyOps)**、**translateAndUpdateReadyOps(用于更新readyOps)**

```
public boolean translateAndUpdateReadyOps(int var1, SelectionKeyImpl var2) {
    return this.translateReadyOps(var1, var2.nioReadyOps(), var2);
}
public boolean translateAndSetReadyOps(int var1, SelectionKeyImpl var2) {
    return this.translateReadyOps(var1, 0, var2);
}
```

判断事件是否注册

int interestSet = selectionKey.interestOps();

boolean isInterestedInAccept = (interestSet & SelectionKey.OP_ACCEPT) == SelectionKey.OP_ACCEPT；

boolean isInterestedInConnect = interestSet & SelectionKey.OP_CONNECT;

boolean isInterestedInRead = interestSet & SelectionKey.OP_READ;

boolean isInterestedInWrite = interestSet & SelectionKey.OP_WRITE;

访问这个ready set就绪的事件

ready 集合是通道已经准备就绪的操作的集合。在一次选择(Selection)之后，你会首先访问这个ready set。Selection将在下一小节进行解释。可以这样访问ready集合：

```
int readySet = selectionKey.readyOps();
```

可以用像检测interest集合那样的方法，来检测channel中什么事件或操作已经就绪。

检测channel中什么事件已经就绪

在一次selection之后，我们可以使用以下几个方法来检测channel中什么事件已经就绪：

selectionKey.isAcceptable()：是否已准备好接受新连接；
selectionKey.isConnectable()：是否已准备好连接；
selectionKey.isReadable()：是否已准备好读取；
selectionKey.isWritable()：是否已准备好写入。

使用的比特掩码来检测就绪事件

我们也可以使用相关的比特掩码来检测就绪状态，与调用上面的方法是一致的。如：

```
if ((selectionkey.readyOps( ) & SelectionKey.OP_READ) != 0) {
    readBuffer.clear( );
    key.channel( ).read (readBuffer);
    ...
}
```

SelectionKey#isAcceptable()

```
public abstract class SelectionKey {

    public abstract int readyOps();

    public final boolean isAcceptable() {

        return (readyOps() & OP_ACCEPT) != 0;

    }
```

获取已经就绪的事件集合SelectionKeyImpl#readyOps()

```
public class SelectionKeyImpl
         extends AbstractSelectionKey {

    private int readyOps;

    public int readyOps() {
        this.ensureValid();
        return this.readyOps;
    }
```

代码中获取已经就绪的事件集合

```
// 7、获取选择键集合
Iterator<SelectionKey> selectedKeys = selector.selectedKeys().iterator();
```

新连接就绪事件的简单处理

```java
while (selectedKeys.hasNext()) {
    // 8、获取单个的选择键，并处理
    SelectionKey selectedKey = selectedKeys.next();

    // 9、判断key是具体的什么事件
    if (selectedKey.isAcceptable()) {
        // 10、若选择键的IO事件是"连接就绪"事件,就获取客户端连接
        SocketChannel socketChannel = serverSocketChannel.accept();
        // 11、切换为非阻塞模式
        socketChannel.configureBlocking(false);
        // 12、将该通道注册到selector选择器上
        socketChannel.register(selector, SelectionKey.OP_READ);
    } else if (selectedKey.isReadable()) {
```

SelectionKey#isAcceptable（）的源码

```java
public static final int OP_ACCEPT = 1 << 4;

public final boolean isAcceptable() {
return (readyOps() & OP_ACCEPT) != 0;
}
```

读就绪事件的简单处理

```java
    } else if (selectedKey.isReadable()) {
        // 13、若选择键的IO事件是"可读"事件,读取数据
        SocketChannel socketChannel = (SocketChannel) selectedKey.channel();

        // 14、读取数据
        ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
        int length = 0;
        while ((length = socketChannel.read(byteBuffer)) > 0) {
            byteBuffer.flip();
            Logger.info(new String(byteBuffer.array(), offset: 0, length));
            byteBuffer.clear();
        }
        socketChannel.close();
    }
```

SelectionKey#isReadable（）的源码

```
public static final int OP_READ = 1 << 0;
// 测试此键的通道是否可以读取。
//如果此键的通道不支持读操作,则此方法总是返回<tt> false </tt>
public final boolean isReadable() {
    return (readyOps() & OP_READ) != 0;
}
```

---

selectionKey如何失效

一个selectionKey被创建后将保持有效，调用selectionKey的cancel()方法或关闭其通道或关闭其选择器将导致其失效。

当我们调用selectionKey的cancel()方法后，它将被放在相关的选择器的cancelledKeys集合中。注册关系不会立即被取消，但是selectionKey会立即失效。当再次调用select( )方法时（或者一个正在进行的select()调用结束时），cancelledKeys中的被取消的键将被清理掉。

检查selectionKey是否仍然有效

我们可以调用isValid( )方法来检查selectionKey是否仍然有效。

关闭其通道或关闭其选择器将导致其失效

---

selectionKey的附件

selectionKey除了维护Channel和Selector的注册关系外，还提供了保存“附件”的功能，并提供方法访问它。这是一种允许我们将任意对象与键关联的便捷方法。

这个对象可以引用任何对象，例如业务对象、会话句柄、其他通道等等。

当我们在遍历与选择器相关的键时，可以使用附加在selectionKey上的对象句柄来获取相关的上下文。

attach( )方法将在selectionKey中保存所提供的对象的引用，attachment( )方法则用来获取与selectionKey关联的附件句柄。

---

selectionKey的附件操作

- public final Object attach(Object ob):

将给定的对象作为附件添加到此key上.在key有效期间,附件可以在多个ops事件中传递.

- public final Object attachment():

获取附件.一个channel的附件,可以再当前Channel(或者说是SelectionKey)生命周期中共享

> attachment数据不会作为socket数据在网络中传输.

EchoServerReactor 中的处理器附件

```
//分步处理,第一步,接收accept事件
SelectionKey sk =
        serverSocket.register(selector, SelectionKey.OP_ACCEPT);
//attach callback object, AcceptorHandler
sk.attach(new AcceptorHandler());
```

EchoServerReactor 中的处理器附件

```
void dispatch( @NotNull SelectionKey sk) {
    Runnable handler = (Runnable) sk.attachment();
    //调用之前attach绑定到选择键的handler处理器对象
    if (handler != null) {
        handler.run();
    }
}
```

Netty中的AbstractNioChannel.doRegister

这是流程的最后一步，代码如下：

```java
//倒数第一步，执行注册

@Override
protected void doRegister() throws Exception {
    boolean selected = false;
    for (;;) {
        try {
 selectionKey = javaChannel().register(eventLoop().selector, 0, this);
            return;
        } catch (CancelledKeyException e) {
            if (!selected) {
                // Force the Selector to select now as the "canceled" SelectionKey may still be
                // cached and not removed because no Select.select(..) operation was called yet.
                eventLoop().selectNow();
                selected = true;
            } else {
                // We forced a select operation on the selector before but the SelectionKey is still cached
                // for whatever reason. JDK bug ?
                throw e;
            }
        }
    }
}
```

迭代 selectedKeys 获取就绪的 IO 事件，为每个事件都调用 processSelectedKey 来处理它.

在前面的channel注册时，将 NioSocketChannel 以附加字段的方式添加到了selectionKey 中。在这里，**通过k.attachment()取得这个通道对象**，然后就调用 processSelectedKey 来处理这个 IO 事件和通道。

```java
entExecutor.java × | SingleThreadEventLoop.java × | NioEventLoop.java × | NioEventLoopGroup.java × | Multi
    }

    private void processSelectedKeysPlain( @NotNull Set<SelectionKey> selectedKeys) {
        // check if the set is empty and if so just return to not create garbage by
        // creating a new Iterator every time even if there is nothing to process.
        // See https://github.com/netty/netty/issues/597
        if (selectedKeys.empty) {
            return
        }

        Iterator<SelectionKey> i = selectedKeys.iterator()   事件迭代
        for () {
            val k = i.next()
            final Object a = k.attachment()
            i.remove()

            if (a instanceof AbstractNioChannel) {
                processSelectedKey(k, (AbstractNioChannel) a)   处理事件
            } else {
                @SuppressWarnings("unchecked")
                NioTask<SelectableChannel> task = (NioTask<SelectableChannel>) a
                processSelectedKey(k, task);
```

简介　目录　评价

Netty核心原理与底层知识 学习盛宴

视频 1. NioEventloop（反应器）核心原理

视频 2. NioEventLoop 任务队列 核心原理

视频 3. ChannelConfig 通道配置类

视频 4. 彻底明白：内核态、用户态、内核空间、用户空间

selectionKey的使用例子

```
selector=Selector.open();
serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
while (true){
    selector.select();
    Set<SelectionKey> selectionKeys=selector.selectedKeys();
    if(null==selectionKeys) continue;
    Iterator<SelectionKey> iterator=selectionKeys.iterator();
    while(iterator.hasNext()){
        SelectionKey selectionKey=iterator.next();
        if(selectionKey.isAcceptable()){
            // a connection was accepted by a ServerSocketChannel.
        }else if(selectionKey.isReadable()){
            // a channel is ready for reading
        }else if(selectionKey.isWritable()){
           // a channel is ready for writing
        }
        iterator.remove();
    }
}
```

强制停止Selector#select()后的selectKey结果

```
selector=Selector.open();
serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
while (true){
    selector.select();
    Set<SelectionKey> selectionKeys=selector.selectedKeys();
    if(null==selectionKeys) continue;
    Iterator<SelectionKey> iterator=selectionKeys.iterator();
    while(iterator.hasNext()){
       ...
    }
}
```

强制停止Selector#select()操作的方法

选择器执行选择的过程，系统底层会依次询问每个通道是否已经就绪，这个过程可能会造成调用线程进入阻塞状态,那么我们有以下三种方式可以唤醒在select（）方法中阻塞的线程。

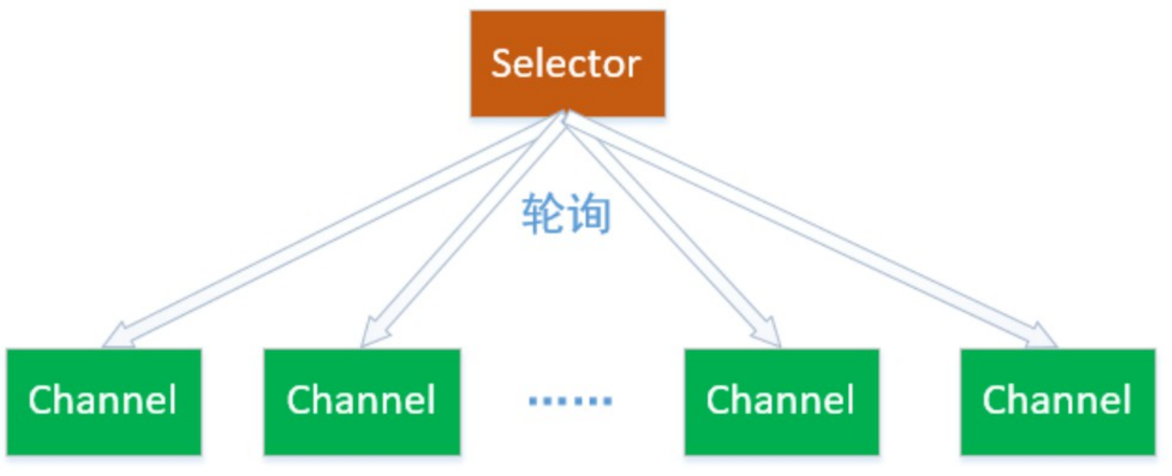
- **wakeup()方法**： 通过调用Selector对象的wakeup（）方法让处在阻塞状态的select()方法立刻返回
  该方法使得选择器上的第一个还没有返回的选择操作立即返回。如果当前没有进行中的选择操作，那么下一次对select()方法的一次调用将立即返回。
- **close()方法**： 通过close（）方法关闭Selector，
  该方法使得任何一个在选择操作中阻塞的线程都被唤醒（类似wakeup（）），同时使得注册到该Selector的所有Channel被注销，所有的键将被取消，但是Channel本身并不会关闭。
- **调用interrupt()**
  调用该方法会使睡眠的线程抛出InterruptException异常，捕获该异常并在调用wakeup()

彻底明白：Selector(选择器) 核心原理

Selector简介

简单来说，Selector就是SelectableChannel对象的多路复用器。通常调用Selector类的静态方法open来创建一个选择器对象，该方法使用系统默认SelectorProvider对象的openSelector方法来创建新的选择器。当然，还可以自定义实现SelectorProvider并重写openSelector方法来创建自定义选择器。

一个Channel对象注册到选择器之后，会返回一个SelectionKey对象，这个SelectionKey对象代表这个Channel和它注册的Selector间的关系。SelectionKey中维护着两个很重要的属性：interestOps、readyOps，并通过这两个属性管理通道上注册的事件。interestOps中保存了我们希望Selector监听Channel的哪些事件，在Selector每次做select操作时，若发现该Channel有我们所监听的事件发生时，就会将感兴趣的监听事件设置到readyOps中，这样我们可以根据事件的发生执行相应的I/O操作。



Selector的创建

通过调用Selector.open()方法创建一个Selector对象，如下：

```java
public class NioDiscardServer {

    public static void startServer() throws IOException {

        // 1、创建一个 Selector选择器
        Selector selector = Selector.open();
```

注册Channel到Selector

```java
// 1、创建一个 Selector选择器
Selector selector = Selector.open();

// 2、获取通道
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
// 3.设置为非阻塞
serverSocketChannel.configureBlocking(false);
// 4、绑定连接
serverSocketChannel.bind(new InetSocketAddress(NioDemoConfig.SOCKET_SERVER_PORT));
Logger.info( s: "服务器启动成功");

// 5、将通道注册到选择器上,并注册的IO事件为: "接收新连接"
SelectionKey sk = serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
```

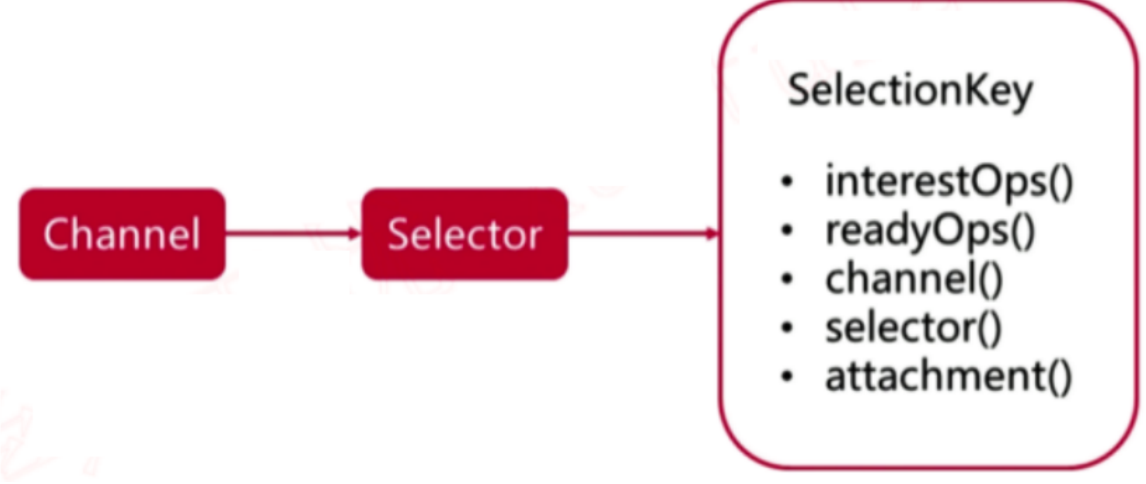
Channel必须是非阻塞的。
所以FileChannel不适用Selector，因为FileChannel不能切换为非阻塞模式，更准确的来说是因为FileChannel没有继承SelectableChannel。Socket channel可以正常使用。

SelectableChannel抽象类 有一个 **configureBlocking（）** 方法用于使通道处于阻塞模式或非阻塞模式。

```
abstract SelectableChannel configureBlocking(boolean block)
```

SelectionKey介绍

一个SelectionKey键表示了一个**channel通道对象**和一个**selector选择器**对象之间的注册关系。



SelectionKey的核心方法

```
key.attachment(); //返回SelectionKey的attachment，attachment可以在注册channel的时候指定。
key.channel(); // 返回该SelectionKey对应的channel。
key.selector(); // 返回该SelectionKey对应的Selector。
key.interestOps(); //返回代表需要Selector监控的IO操作的bit mask
key.readyOps(); // 返回一个bit mask，代表在相应channel上可以进行的IO操作。
```

Selector#select()方法介绍：

在刚初始化的Selector对象中，这三个集合都是空的。

通过Selector的select（）方法可以选择已经准备就绪的通道 （这些通道包含你感兴趣的的事件）。

比如你对读就绪的通道感兴趣，那么select（）方法就会返回读事件已经就绪的那些通道。

下面是Selector几个重载的select()方法：

- int select()：阻塞到至少有一个通道在你注册的事件上就绪了。
- int select(long timeout)：和select()一样，但最长阻塞时间为timeout毫秒。
- int selectNow()：非阻塞，只要有通道就绪就立刻返回。

---

selector维护三个set集合：

- Keys：键集

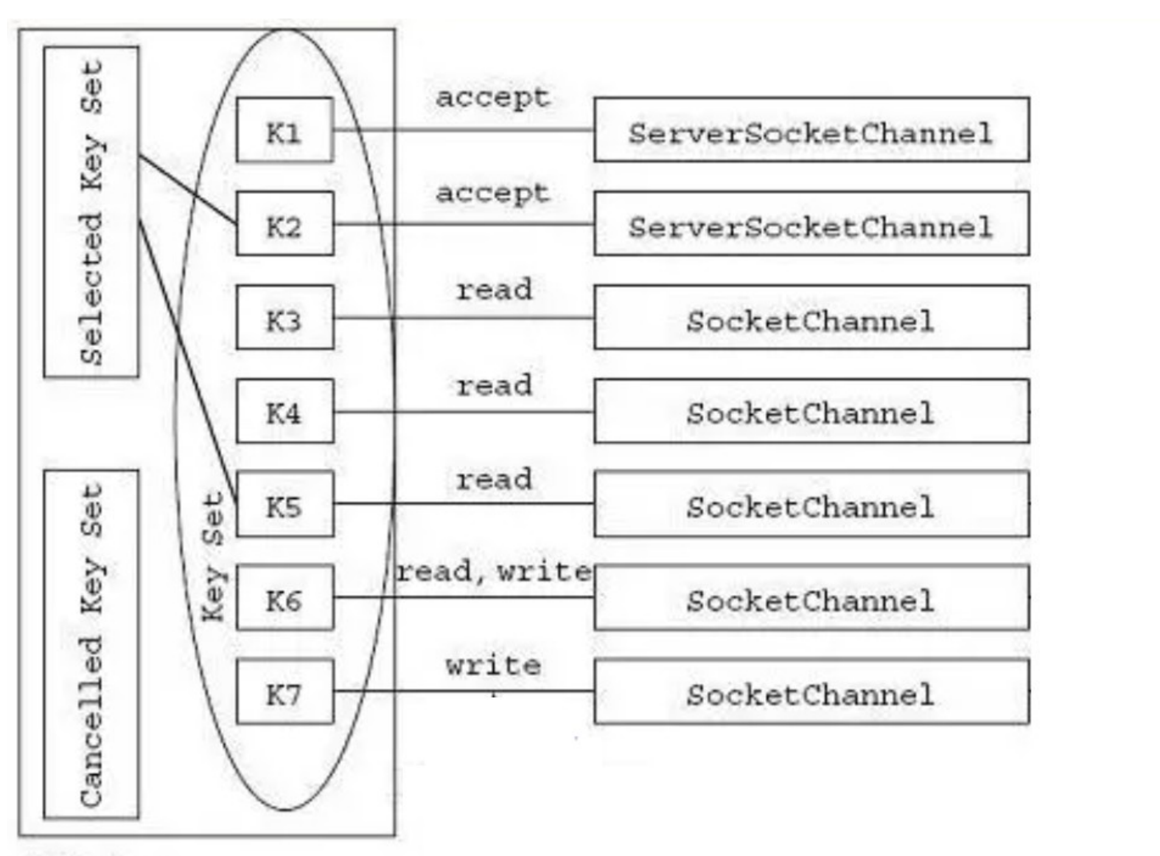
存放注册到Selector的所有key

- selectedKeys：已选择键集

被选中的key，它是检测到registeredKeys中key感兴趣的事件发生后存放key的地方，

- cancelledKeys：已取消键集

其cancel()方法被调用过的，待反注册的key



调用Selector的selectedKeys()方法来访问已选择键集合

一旦调用select()方法返回，并且返回值不为0时，则 可以通过调用Selector的selectedKeys()方法来访问已选择键集合 。如下：

```java
// 6、轮询感兴趣的I/O就绪事件（选择键集合）
while (selector.select() > 0) {
    // 7、获取选择键集合
    Iterator<SelectionKey> selectedKeys = selector.selectedKeys().iterator();
    while (selectedKeys.hasNext()) {
        // 8、获取单个的选择键，并处理
        SelectionKey selectedKey = selectedKeys.next();

        // 9、判断key是具体的什么事件
        if (selectedKey.isAcceptable()) {
            // 10、若选择键的IO事件是"连接就绪"事件,就获取客户端连接
            SocketChannel socketChannel = serverSocketChannel.accept();
            // 11、切换为非阻塞模式
            socketChannel.configureBlocking(false);
            // 12、将该通道注册到selector选择器上
            socketChannel.register(selector, SelectionKey.OP_READ);
        } else if (selectedKey.isReadable()) {
            // 13、若选择键的IO事件是"可读"事件,读取数据
```

停止Selector#select()选择操作的方法

选择器执行选择的过程，系统底层会依次询问每个通道是否已经就绪，这个过程可能会造成调用线程进入阻塞状态,那么我们有以下三种方式可以唤醒在select（）方法中阻塞的线程。

- **wakeup()方法**： 通过调用Selector对象的wakeup（）方法让处在阻塞状态的select()方法立刻返回
  该方法使得选择器上的第一个还没有返回的选择操作立即返回。如果当前没有进行中的选择操作，那么下一次对select()方法的一次调用将立即返回。
- **close()方法**： 通过close（）方法关闭Selector，
  该方法使得任何一个在选择操作中阻塞的线程都被唤醒（类似wakeup（）），同时使得注册到该Selector的所有Channel被注销，所有的键将被取消，但是Channel本身并不会关闭。
- **调用interrupt()**
  调用该方法会使睡眠的线程抛出InterruptException异常，捕获该异常并在调用wakeup()

不同Selector实现类

目前Selector目前有如下实现

- Object - java.lang
  - Selector - java.nio.channels
    - AbstractSelector - java.nio.channels.spi
      - SelectorImpl - sun.nio.ch
        - AbstractPollSelectorImpl - sun.nio.ch
          - PollSelectorImpl - sun.nio.ch
        - DevPollSelectorImpl - sun.nio.ch
        - EPollSelectorImpl - sun.nio.ch
        - EventPortSelectorImpl - sun.nio.ch
        - WindowsSelectorImpl - sun.nio.ch

不同操作系统平台的不同Selector实现类

针对linux平台的实现：

- PollSelectorImpl：基于poll来实现
- EPollSelectorImpl：基于epoll来实现

针对windows平台的实现：

- WindowsSelectorImpl

Netty使用的实现类

- Netty的NioEventLoop则是使用上述linux平台的实现PollSelectorImpl。
- 额外，Netty自己提供了另外一种epoll实现，没有直接采用上述jdk自带的EPollSelectorImpl。

最强揭秘：Selector.open() 选择器打开的底层原理

Selector.open()方法的使用

通过调用Selector.open()方法创建一个Selector对象，如下：

```
public class NioDiscardServer {

    public static void startServer() throws IOException {

        // 1、创建一个 Selector选择器
        Selector selector = Selector.open();
```

Selector.open();

```
public static Selector open() throws IOException {
    return SelectorProvider.provider().openSelector();
}
```

这里会创建一个provider

创建一个provider

```
public static SelectorProvider provider() {
        synchronized (lock) {
            if (provider != null)
                return provider;
            return AccessController.doPrivileged(
                new PrivilegedAction<>() {
                    public SelectorProvider run() {
                        if(loadProviderFromProperty())
                                return provider;
                            if(loadProviderAsService())
                                return provider;

provider=sun.nio.ch.DefaultSelectorProvider.create();
                            return provider;
                        }
                    });
        }
    }
```

这里调用了DefaultSelectorProvider创建SelectorProvider

在windows环境中这里创建的是WindowsSelectorProvider

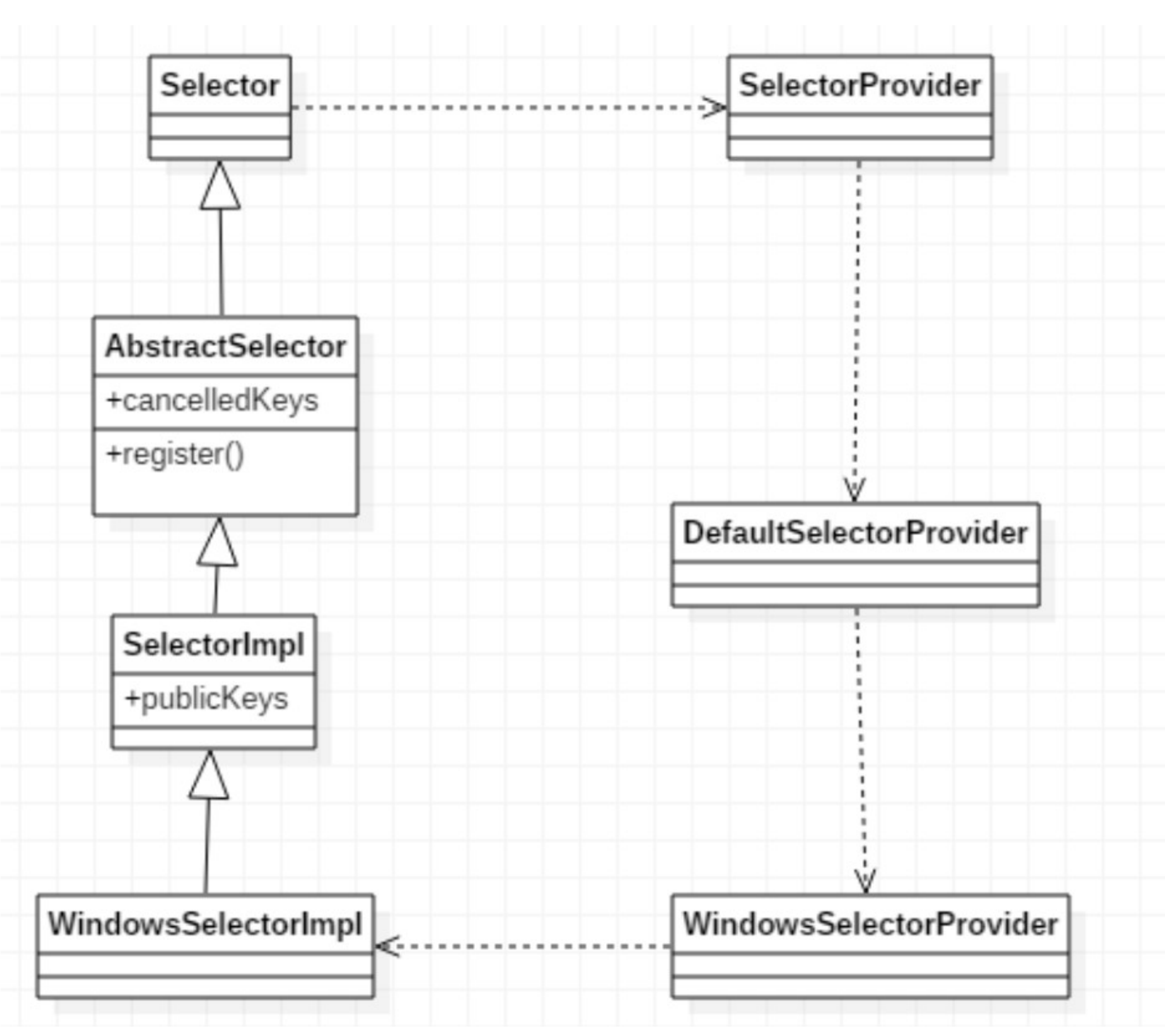
JDK操作系统版本有关，这里是Windows。

```
public static SelectorProvider create() {
   return new sun.nio.ch.WindowsSelectorProvider();
}
```

随后调用WindowsSelectorProvider.openSelector

```
public class WindowsSelectorProvider extends SelectorProviderImpl {

    public AbstractSelector openSelector() throws IOException {
        return new WindowsSelectorImpl(this);
    }
}
```

JDK操作系统版本有关，这里是Windows。



SelectorProvider的实现与操作系统相关

Java NIO根据操作系统不同， 针对nio中的Selector有不同的实现：

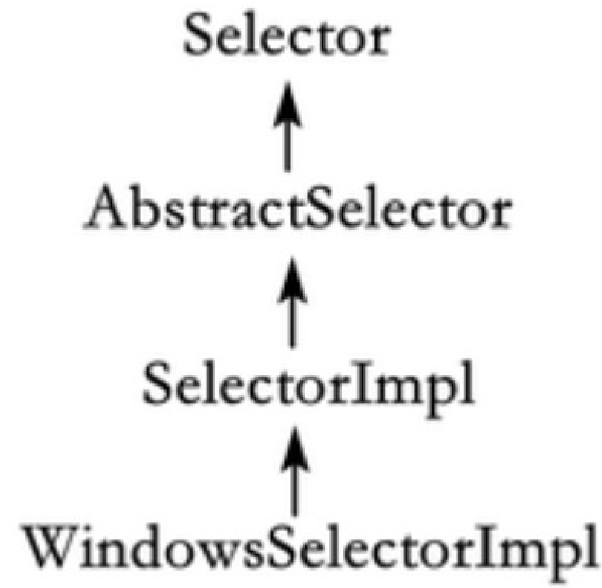
macosx:KQueueSelectorProvider

solaris:DevPollSelectorProvider

Linux:EPollSelectorProvider (Linux kernels >= 2.6)或PollSelectorProvider

windows:WindowsSelectorProvider

WindowsSelectorProvider的继承关系图



SelectorImpl会初始化selectedKeys和keys

```
public abstract class SelectorImpl extends AbstractSelector {

    /** 已选择集合，被 select()查出的IO事件的，添加到该键集并返回 */
        protected Set<SelectionKey> selectedKeys = new HashSet();

     /** 键集，register()时添加到该键级 */
    protected HashSet<SelectionKey> keys = new HashSet();

    protected SelectorImpl(SelectorProvider selectorProvider) {
        super(selectorProvider);
        ...
    }
}
```

AbstractSelector会初始化provider

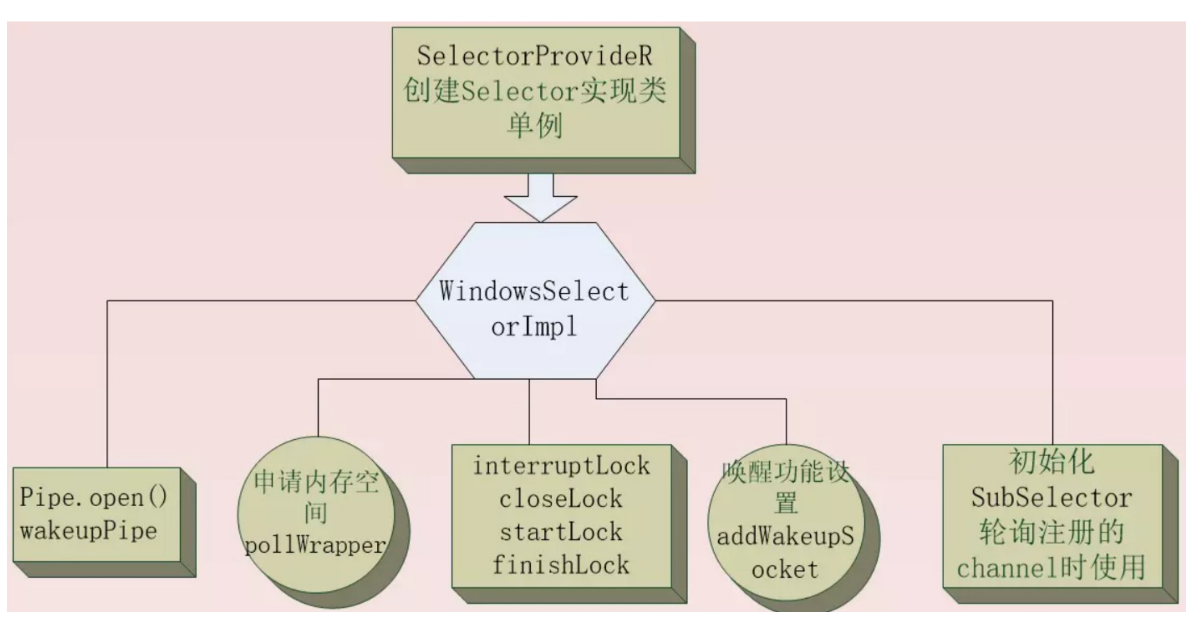
```
public abstract class AbstractSelector extends Selector {
    private final SelectorProvider provider;

    /** 已取消键集，已被取消但其通道尚未注销的键的集合 */
    private final Set<SelectionKey> cancelledKeys = new HashSet<SelectionKey>();

     protected AbstractSelector(SelectorProvider provider) {
        this.provider = provider;
    }
}
```

这里是WindowsSelectorProvider

WindowsSelectorImpl构造器

```
WindowsSelectorImpl(SelectorProvider selectorprovider)
        throws IOException
   {
        super(selectorprovider);
        channelArray = new SelectionKeyImpl[8];
        totalChannels = 1;
        threadsCount = 0;
        closeLock = new Object();
        interruptTriggered = false;
        updateCount = 0L;
        pollWrapper = new PollArrayWrapper(8);  //申请直接内存
        wakeupSourceFd = ((SelChImpl)wakeupPipe.source()).getFDVal();  //唤醒通道
        SinkChannelImpl sinkchannelimpl = (SinkChannelImpl)wakeupPipe.sink();
        sinkchannelimpl.sc.socket().setTcpNoDelay(true);
        wakeupSinkFd = sinkchannelimpl.getFDVal();
        pollWrapper.addWakeupSocket(wakeupSourceFd, 0); //唤醒通道加入到 事件轮询队列
   }
```

WindowsSelectorProvider.open()



WindowsSelectorImpl结构

名称	作用
SelectionKeyImpl[] channelArray	存放注册的SelectionKey
PollArrayWrapper pollWrapper	底层的本机轮询数组包装对象，用于存放注册的Socket文件描述符和事件掩码
List threads	辅助线程，多个线程有助于提高高并发时的性能
Pipe wakeupPipe	用于唤醒被阻塞的查询线程
FdMap fdMap	保存文件描述符和SelectionKey的对应关系
SubSelector subSelector	调用JNI的poll和处理就绪的SelectionKey
StartLock startLock	辅助线程使用该锁等待主线程的开始信号
FinishLock finishLock	主线程用该锁等待所有辅助线程执行完毕

FdMap结构

保存socket文件描述符句柄和SelectionKey的关系，socket的事件发生时，可以通过socket文件描述符句柄从FdMap中获取到对应的SelectionKey。

```
private final static class FdMap extends HashMap<Integer, MapEntry> {
    static final long serialVersionUID = 0L;
    private MapEntry get(int desc) {
        return get(new Integer(desc));
    }
    private MapEntry put(SelectionKeyImpl ski) {
        return put(new Integer(ski.channel.getFDVal()), new MapEntry(ski));
    }
    private MapEntry remove(SelectionKeyImpl ski) {
        Integer fd = new Integer(ski.channel.getFDVal());
        MapEntry x = get(fd);
        if ((x != null) && (x.ski.channel == ski.channel))
            return remove(fd);
        return null;
    }
}
```

SubSelector

SubSelector封装了调用JNI poll的逻辑，以及获取就绪SelectionKey的方法。

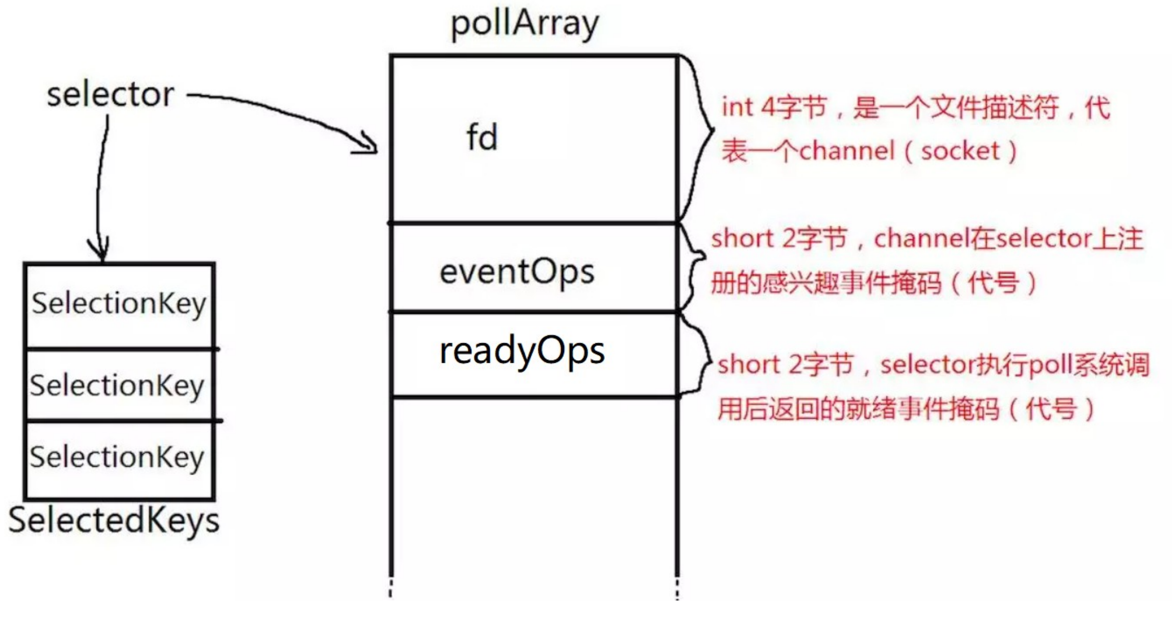
每一个Selector都有一个SubSelector，其内存保存了select/poll/epoll获取到的可读文件描述符，可写文件描述符以及异常的文件描述符。这样Selector就有自己单独的就绪文件描述符数组。

```
private final int pollArrayIndex;
private final int[] readFds = new int [MAX_SELECTABLE_FDS + 1];
private final int[] writeFds = new int [MAX_SELECTABLE_FDS + 1];
private final int[] exceptFds = new int [MAX_SELECTABLE_FDS + 1];
```

PollArrayWrapper 中 pollArray事件轮询队列

pollArray用Unsafe类申请一块物理内存，存放注册时的socket句柄fdVal和event的数据结构，其中pollfd共8字节，0~3字节保存socket句柄，4~7字节保存event。

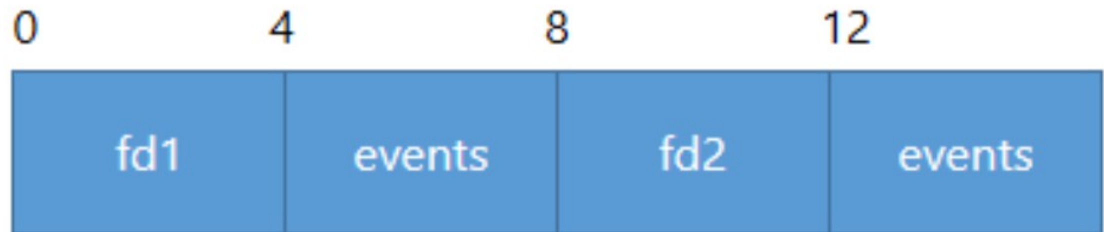
这两个信息在8字节数据单元中的偏移量分别由FD_OFFSET和EVENT_OFFSET标识，也就是0和4。



---

pollArray文件描述符poll事件轮询队列的元素

pollArray中存放的内容是一个个8字节的数据单元。这个8字节数据有两部分组成，每部分占用4个字节，分别是fd信息和这个fd关注的事件events，pollArray中的数据结构如下：



PollArrayWrapper的成员

```
class PollArrayWrapper
{
    private AllocatedNativeObject pollArray;//底层内存空间
    long pollArrayAddress;//内存空间起始位置
```

```
    private static final short FD_OFFSET = 0;文件描述id开始位置
    private static final short EVENT_OFFSET = 4;//兴趣事件开始位置
    static short SIZE_POLLFD = 8;//文件描述id的长度int（4）+操作事件长度4

    static final short POLLIN = 1;
    static final short POLLOUT = 4;
    static final short POLLERR = 8;
    static final short POLLHUP = 16;
    static final short POLLNVAL = 32;
    static final short POLLREMOVE = 2048;
    static final short POLLCONN = 2;//
```

events&revents的取值如下：

事件	描述	是否可作为输入（events）	是否可作为输出（revents）
POLLIN	数据可读（包括普通数据&优先数据）	是	是
POLLOUT	数据可写（普通数据&优先数据）	是	是
POLLRDNORM	普通数据可读	是	是
POLLRDBAND	优先级带数据可读（linux不支持）	是	是
POLLPRI	高优先级数据可读，比如TCP带外数据	是	是

事件	描述	是否可作为输入（events）	是否可作为输出（revents）
POLLWRNORM	普通数据可写	是	是
POLLWRBAND	优先级带数据可写	是	是
POLLRDHUP	TCP连接被对端关闭，或者关闭了写操作，由GNU引入	是	是
POPPHUP	挂起	否	是
POLLERR	错误	否	是
POLLNVAL	文件描述符没有打开	否	是

PollArrayWrapper的构造器

```
    //创建第i个subselector的poll轮询队列
     PollArrayWrapper(int i)
     {
         int j = i * SIZE_POLLFD;
     //分配内存空间
         pollArray = new AllocatedNativeObject(j, true);
     //初始化空间起始地址
         pollArrayAddress = pollArray.address();
         size = i;//初始化容量
     }
```

NativeObject分配直接内存

```
  //分配i大小的内存空间，flag为是否分配内存页
     protected NativeObject(int i, boolean flag)
     {
         if(!flag)
         {
             allocationAddress = unsafe.allocateMemory(i);
             address = allocationAddress;
         } else
         {
             int j = pageSize();
             long l = unsafe.allocateMemory(i + j);
             allocationAddress = l;//已分配内存空间
             address = (l + (long)j) - (l & (long)(j - 1));//空间起始位置
         }
     }
```

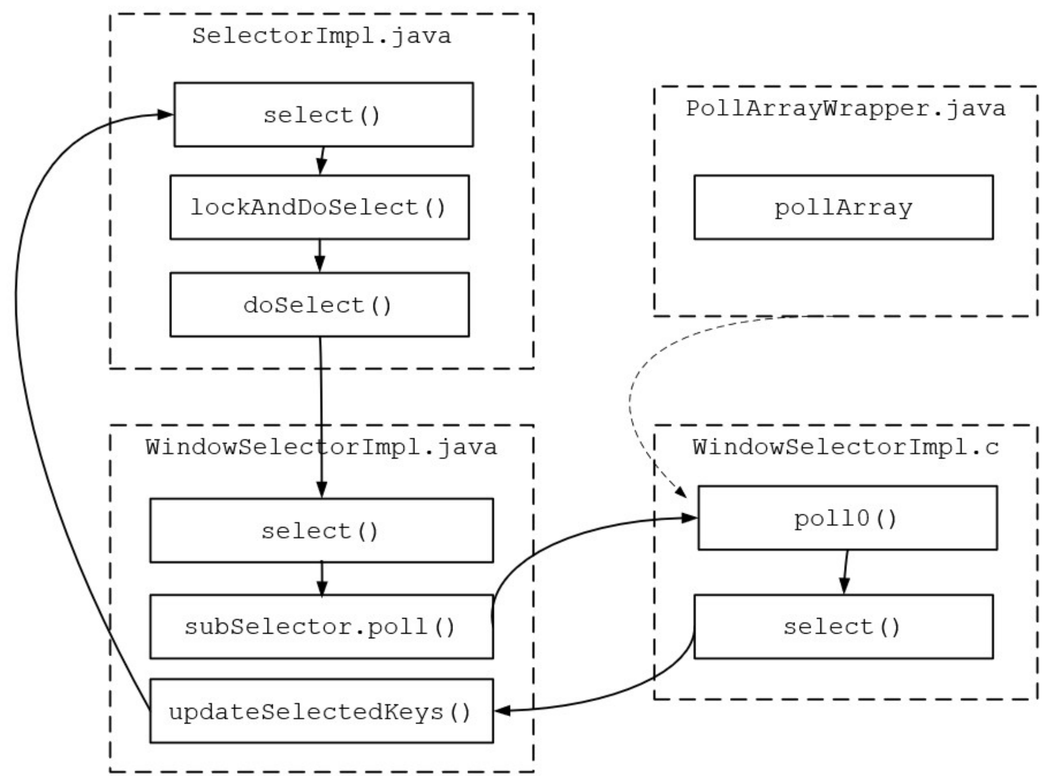
SubSelector的属性

```
class WindowsSelectorImpl extends SelectorImpl {
   private static final int MAX_SELECTABLE_FDS = 1024;
   private final class SubSelector {
        private final int pollArrayIndex; //pollArray轮询队列的开始坐标
        // 下面的数组，将持有 native select() 的结果.
        // 下面的数组，第一个元素，为被查询到的socket的数量
        // 后面的元素，为被查询到的socket的文件描述符
        private final int[] readFds = new int [MAX_SELECTABLE_FDS + 1];
        private final int[] writeFds = new int [MAX_SELECTABLE_FDS + 1];
        private final int[] exceptFds = new int [MAX_SELECTABLE_FDS + 1];
```

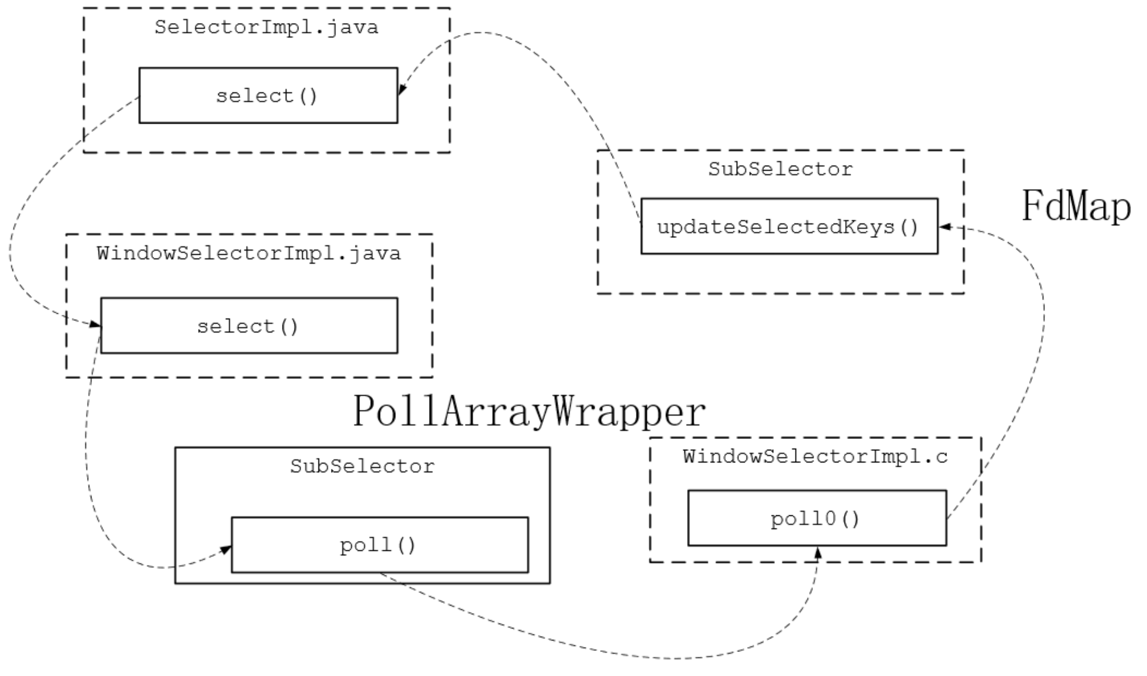
SubSelector的方法

```
private SubSelector() {
    this.pollArrayIndex = 0; // 主线程
}
private SubSelector(int threadIndex) { // helper threads 辅助线程
    this.pollArrayIndex = (threadIndex + 1) * MAX_SELECTABLE_FDS;
}
private int poll() throws IOException{ // poll for the main thread
    return poll0(pollWrapper.pollArrayAddress,
                 Math.min(totalChannels, MAX_SELECTABLE_FDS),
                 readFds, writeFds, exceptFds, timeout);
}
private int poll(int index) throws IOException {
    // poll for helper threads  辅助线程轮询
    return  poll0(pollWrapper.pollArrayAddress +
             (pollArrayIndex * PollArrayWrapper.SIZE_POLLFD),
             Math.min(MAX_SELECTABLE_FDS,
                     totalChannels - (index + 1) * MAX_SELECTABLE_FDS),
             readFds, writeFds, exceptFds, timeout);
}
private native int poll0(long pollAddress, int numfds,
     int[] readFds, int[] writeFds, int[] exceptFds, long timeout);
```

调用JNI本地poll0方法查询事件



selector的大致查询过程

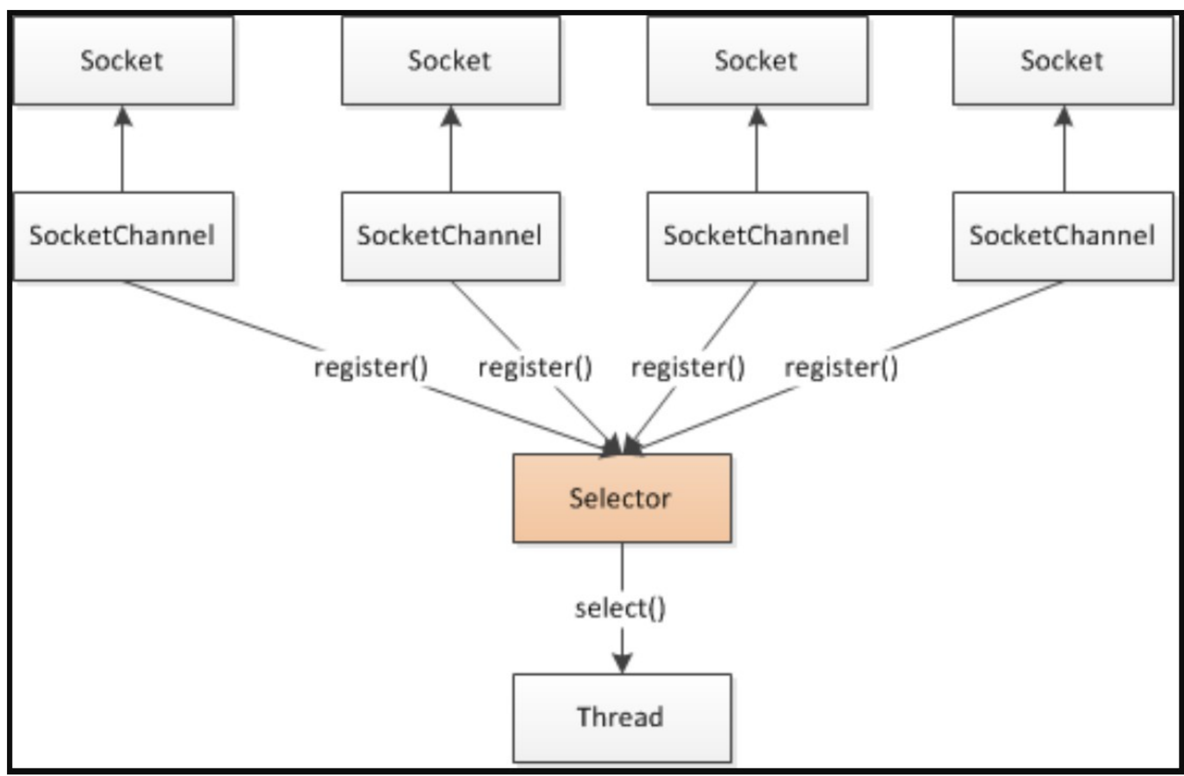


最强揭秘：channel到selector注册的底层原理

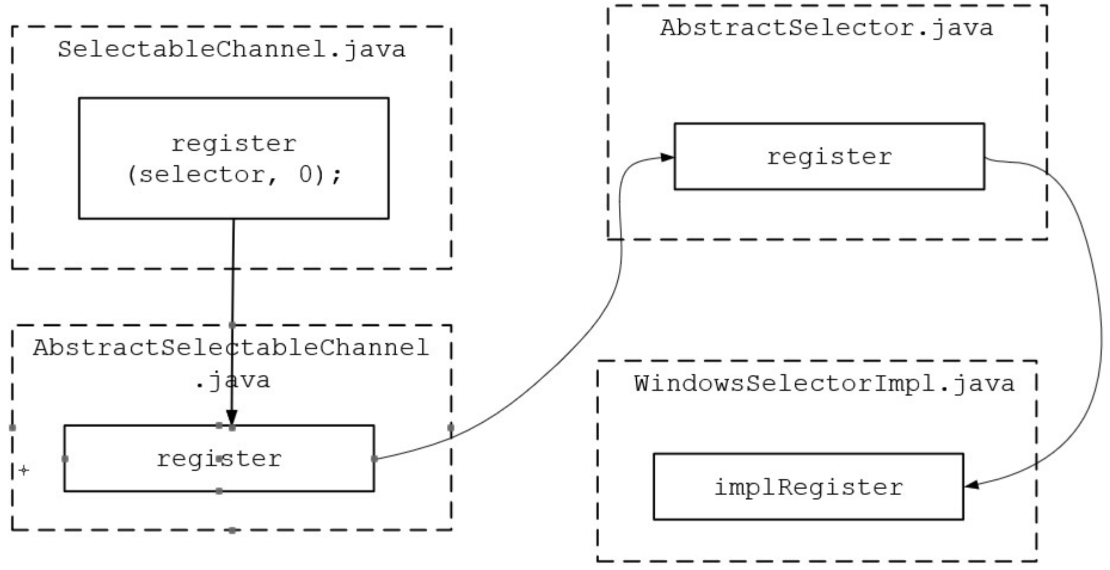
channel到selector注册的代码

```java
// 9、判断key是具体的什么事件
if (selectedKey.isAcceptable()) {
    // 10、若选择键的IO事件是"连接就绪"事件,就获取客户端连接
    SocketChannel socketChannel = serverSocketChannel.accept();
    // 11、切换为非阻塞模式
    socketChannel.configureBlocking(false);
    // 12、将该通道注册到selector选择器上
    socketChannel.register(selector, SelectionKey.OP_READ);
} else if (selectedKey.isReadable()) {
    // 13、若选择键的IO事件是"可读"事件,读取数据
    SocketChannel socketChannel = (SocketChannel) selectedKey.channel();
```

通道注册示意图



注册大致流程



SelectableChannel#register();

sk = channel.register(selector, 0);

```
public final SelectionKey register(Selector sel, int ops)
    throws ClosedChannelException
{
    return register(sel, ops, null);
}

public abstract SelectionKey register(Selector sel, int ops, Object att)
throws ClosedChannelException;
```

channel.register() 方法的第二个参数。

这是一个“ **interest集合** ”，意思是在**Selector对Channel的那些事件感兴趣**。可以监听四种不同类型的事件（这四种事件用SelectionKey的四个常量来表示）：

- SelectionKey.OP_CONNECT “ **连接就绪** ”（连接事件）
- SelectionKey.OP_ACCEPT “ **接收就绪** ”（新连接接收事件）
- SelectionKey.OP_READ “ **读就绪** ” （通道读取缓冲区可读）
- SelectionKey.OP_WRITE “ **写就绪** ” （通道写入缓冲区可读）

OP_CONNECT表示某个Channel成功连接到服务器。

OP_ACCEPT表示一个Server Socket Channel准备好接收新进入的连接称为。

OP_READ一个有数据可读的通道可以说是“ **读就绪** ”。

OP_WRITE表示channel等待写数据，或者说可以写入数据，就是通道的写入缓冲区大于**低水位SO_SNDLOWAT的值（默认为2048）**，而**套接字默认的发送缓冲区大小是8k**，所以总是可写的。

用“位或”操作符将多个常量连接起来

注册通道时，如果我们不止对一种操作感兴趣，可以用“位或”操作符将多个常量连接起来。如下：

```
socketChannel.register(selector,
  SelectionKey.OP_CONNECT|
                      SelectionKey.OP_READ|
                                SelectionKey.OP_WRITE);
```

AbstractSelectableChannel.register(selector, ops, att)

```
public final SelectionKey register(Selector sel, int ops, Object att)
    throws ClosedChannelException {
    synchronized (regLock) {

        //如果该channel和selector已经注册过
        SelectionKey k = findKey(sel);
        if (k != null) {
            k.interestOps(ops);
            k.attach(att); //则直接添加事件和附件。
        }
        if (k == null) {
            // New registration
            synchronized (keyLock) {
                if (!isOpen())
                    throw new ClosedChannelException();
                k = ((AbstractSelector)sel).register(this, ops, att);
                addKey(k);
            }
        }
        return k;
    }
}
```

AbstractSelector.register

```
protected final SelectionKey register(AbstractSelectableChannel ch,
        int ops,  Object attachment) {
    if (!(ch instanceof SelChImpl))
        throw new IllegalSelectorException();
        //创建key
    SelectionKeyImpl k = new SelectionKeyImpl((SelChImpl) , this);
    k.attach(attachment);  //添加附件
    synchronized (publicKeys) {
        implRegister(k);  //实施注册
    }
    k.interestOps(ops);
    return k;
}
```

WindowsSelectorImpl#implRegister实现注册

```
protected void implRegister(SelectionKeyImpl ski) {
    synchronized (closeLock) {
        if (pollWrapper == null)
            throw new ClosedSelectorException();

        //如果当前channel的数量totalChannels等于SelectionKeyImpl数组大小
         // 对SelectionKeyImpl数组和pollWrapper进行扩容操作。
        growIfNeeded();

        //选择键加入数组
        channelArray[totalChannels] = ski;
        ski.setIndex(totalChannels);
        fdMap.put(ski);
        keys.add(ski);  //选择键注册
        pollWrapper.addEntry(totalChannels, ski); //socket句柄添加到对应的pollfd
        totalChannels++;  //通道数增加
    }
}
```

pollWrapper的addEntry

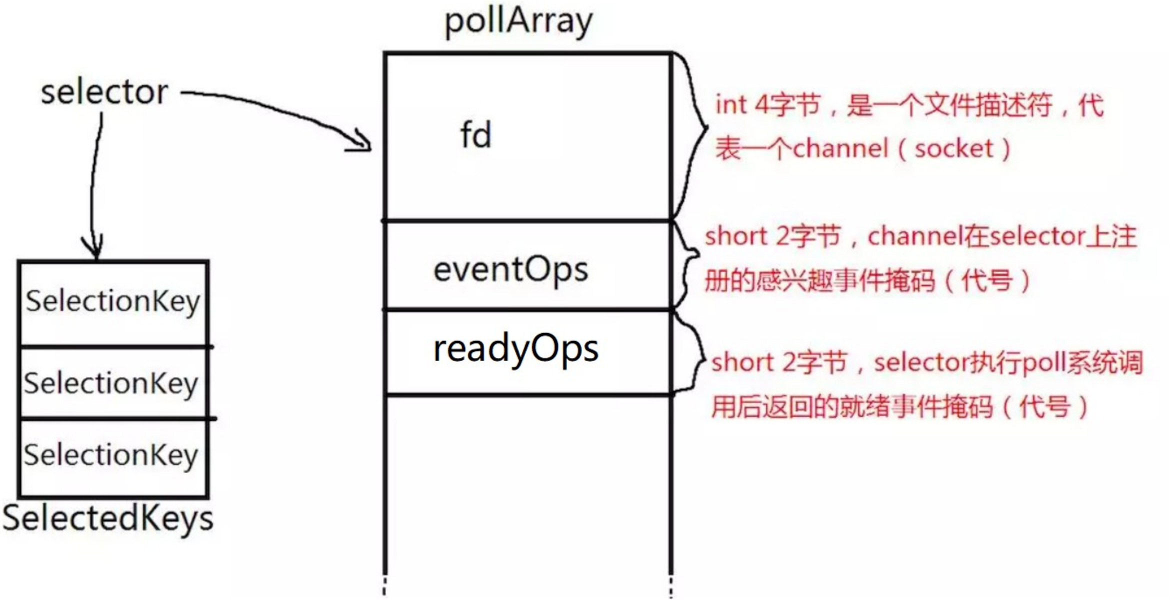
ski作为pollWrapper的addEntry()的参数，最终通过putDescriptor放到上图的pollArray的fd域中：

```
 void addEntry(int index, SelectionKeyImpl ski) {
        putDescriptor(index, ski.channel.getFDVal());
    }

void putDescriptor(int index, int fd) {
        pollArray.putInt(SIZE_POLLFD * index + FD_OFFSET, fd);
    }
```

> FD_OFFSET和EVENT_OFFSET标识，也就是0和4； static short SIZE_POLLFD = 8;

pollArray的元素内容

pollArray的确包含了所有fd和fd关心的事件信息，而且这些信息的确从register中传递过来的



回顾AbstractSelector.register

```
protected final SelectionKey register(AbstractSelectableChannel ch,
                                          int ops,
                                          Object attachment)
    {
        if (!(ch instanceof SelChImpl))
            throw new IllegalSelectorException();
        SelectionKeyImpl k = new SelectionKeyImpl((SelChImpl)ch, this);
        k.attach(attachment);
        synchronized (publicKeys) {
            implRegister(k);
        }
        k.interestOps(ops);
        return k;
    }
```

selectkey#interestOps(ops)代码跟进，最终来到WindowsSelectorImpl的putEventOps中。

SelectionKeyImpl#interestOps(ops)

```
public SelectionKey interestOps(int var1) {
        this.ensureValid();
        return this.nioInterestOps(var1);
}
```

SelectionKeyImpl.nioInterestOps

```
    public SelectionKey nioInterestOps(int ops) {
        if ((var1 & ~this.channel().validOps()) != 0) {
            throw new IllegalArgumentException();
        } else {
            this.channel.translateAndSetInterestOps(ops, this);
            this.interestOps = var1;
            return this;
        }
    }
```

channel#translateAndSetInterestOps

调用通道的translateAndSetInterestOps方法将该关注类型对应的POLL类型写入写入到选择键的pollWrapper属性中

```
//SocketChannel
public void translateAndSetInterestOps(int ops, SelectionKeyImpl sk) {
    int pollOps = 0;
    //OP_READ
    if ((ops & 1) != 0) {
        pollOps |= Net.POLLIN;
    }
    //OP_WRITE
    if ((ops & 4) != 0) {
        pollOps |= Net.POLLOUT;
    }
    //OP_CONNECT
    if ((ops & 8) != 0) {
        pollOps |= Net.POLLCONN;
    }
    sk.selector.putEventOps(sk, pollOps);
}
```

WindowsSelectorImpl的putEventOps中

```
public void putEventOps(SelectionKeyImpl sk, int pollOps) {
        synchronized (closeLock) {
            if (pollWrapper == null)
                throw new ClosedSelectorException();
            // make sure this sk has not been removed yet
            int index = sk.getIndex();
            if (index == -1)
                throw new CancelledKeyException();
            pollWrapper.putEventOps(index, pollOps);
        }
    }
```

重点：将感兴趣的事件放到pollArray中由i指定的位置。

调用pollWrapper的putEventOps将事件信息放到pollArray中由i指定的位置。

```
void putEventOps(int index, int event) {
        pollArray.putShort(SIZE_POLLFD * index + EVENT_OFFSET, (short)event);
    }
```

> D_OFFSET和EVENT_OFFSET标识，也就是0和4； static short SIZE_POLLFD = 8;

WindowsSelectorImpl#implRegister实现注册

pollArray中index索引有意思，与sk的index属性相对应，与sk在channelArray中的索引对应

```
protected void implRegister(SelectionKeyImpl ski) {
    synchronized (closeLock) {
        if (pollWrapper == null)
            throw new ClosedSelectorException();

        //如果当前channel的数量totalChannels等于SelectionKeyImpl数组大小
         // 对SelectionKeyImpl数组和pollWrapper进行扩容操作。
        growIfNeeded();

        //选择键加入数组
        channelArray[totalChannels] = ski;
        ski.setIndex(totalChannels);
        fdMap.put(ski);
        keys.add(ski);  //选择键注册
        pollWrapper.addEntry(totalChannels, ski); //socket句柄添加到对应的pollfd
        totalChannels++;  //通道数增加
    }
}
```

扩容channelArray

在注册之前会先会判断当前注册的Channel数量 是否达到需要启动辅助线程的阈值。如果达到阈值则需要扩容pollWrapper数组，同时还要 将wakeupSourceFd加入到扩容后的第一个位置 （具体作用下面会讲解）。

```
private void growIfNeeded() {
    if (channelArray.length == totalChannels) {
        //channel数组已满,扩容两倍
        int newSize = totalChannels * 2; // Make a larger array
        SelectionKeyImpl temp[] = new SelectionKeyImpl[newSize];
        System.arraycopy(channelArray, 1, temp, 1, totalChannels - 1);
        channelArray = temp;
        //文件描述符数组扩容
        pollWrapper.grow(newSize);
    }
    //达到最大文件描述符数量时添加辅助线程
    if (totalChannels % MAX_SELECTABLE_FDS == 0) { // more threads needed
        //将唤醒的文件描述符加入到扩容后的第一个位置。
        pollWrapper.addWakeupSocket(wakeupSourceFd, totalChannels);
        totalChannels++;
        //添加线程数
        threadsCount++;
    }
}
```

---

pollWrapper.grow(newSize)扩容PollArrayWrapper

```
void grow(int newSize) {
    //创建新的数组
    PollArrayWrapper temp = new PollArrayWrapper(newSize);
    for (int i = 0; i < size; i++)
    //将原来的数组的内容存放到新的数组中
        replaceEntry(this, i, temp, i);
    //释放原来的数组
    pollArray.free();
    //更新引用
    pollArray = temp.pollArray;
    //更新大小
    this.size = temp.size;
    //更新地址
    pollArrayAddress = pollArray.address();
}
```

辅助线程的管理

扩容完成时，当通道数量>1024时，需要添加一个辅助线程以并行的处理所有文件描述符。

其中常量MAX_SELECTABLE_FDS=1024，即每增加1024个channel，就会增加一个辅助线程；每减少1024个channel，就会减少一个辅助线程。

```
private void growIfNeeded() {
  ...
        //达到最大文件描述符数量时添加辅助线程
    if (totalChannels % MAX_SELECTABLE_FDS == 0) { // more threads needed
        //将唤醒的文件描述符加入到扩容后的第一个位置。
        pollWrapper.addWakeupSocket(wakeupSourceFd, totalChannels);
        totalChannels++;
        //添加线程数
        threadsCount++;
}
}
```

WindowsSelectorImpl下的多线程提高poll的性能

主线程处理前1024个文件描述符，第二个辅助线程处理1025到2048的文件描述符，以此类推。

这样使得主线程调用poll的时候，通过多线程并行执行一次性获取到所有的已就绪的文件描述符，从而提高在高并发时的poll的性能。

每1024个PollFD的第一个句柄都要设置为wakeupSourceFd，因此在扩容的时候也需要将新的位置的第一个设置为wakeupSourceFd，该线程的目的是为了唤醒辅助线程 。

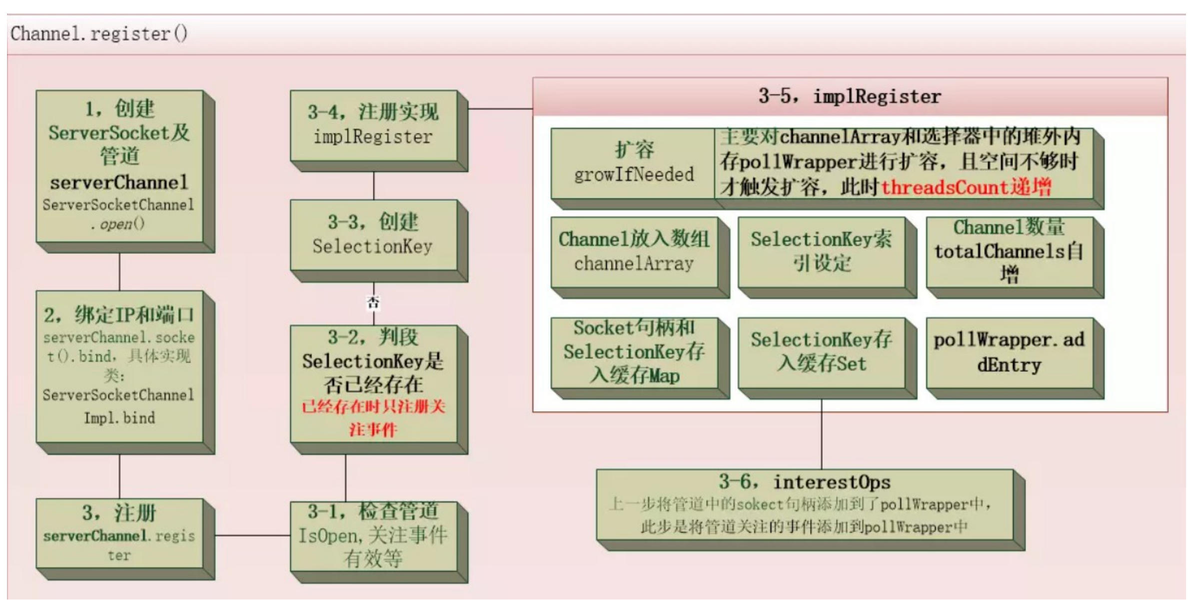
当多个线程阻塞在Poll，若此时主线程已经处理完成，则需要等待所有辅助线程完成，通过向wakeupSourceFd发送信号以激活Poll不在阻塞。

现在我们知道了windows下poll多线程的使用方法，因为多线程poll还需要其他的数据结构支持同步，具体的多线程执行逻辑我们下面再讨论。

WindowsSelectorImpl的性能问题

在项目性能测试过程中发现，同样的代码，连接同样数量（10万）的设备（设备和代码之间通过NIO有大量的数据交互），在Linux下CPU利用率只有**20%~30%**，而在windows下却一直**高于80%。**

回顾：整体的注册流程



最强揭秘：Selector.select() 事件查询的底层原理

```java
// 5、将通道注册到选择器上,并注册的IO事件为："接收新连接"
serverChannel.register(selector, SelectionKey.OP_ACCEPT);
Logger.tcfo( s: "serverChannel is linstening...");
// 6、轮询感兴趣的I/O就绪事件（选择键集合）
while (selector.select() > 0) {
    if (null == selector.selectedKeys()) continue;
    // 7、获取选择键集合
    Iterator<SelectionKey> it = selector.selectedKeys().iterator();
    while (it.hasNext()) {
        // 8、获取单个的选择键，并处理
        SelectionKey key = it.next();
        if (null == key) continue;
```

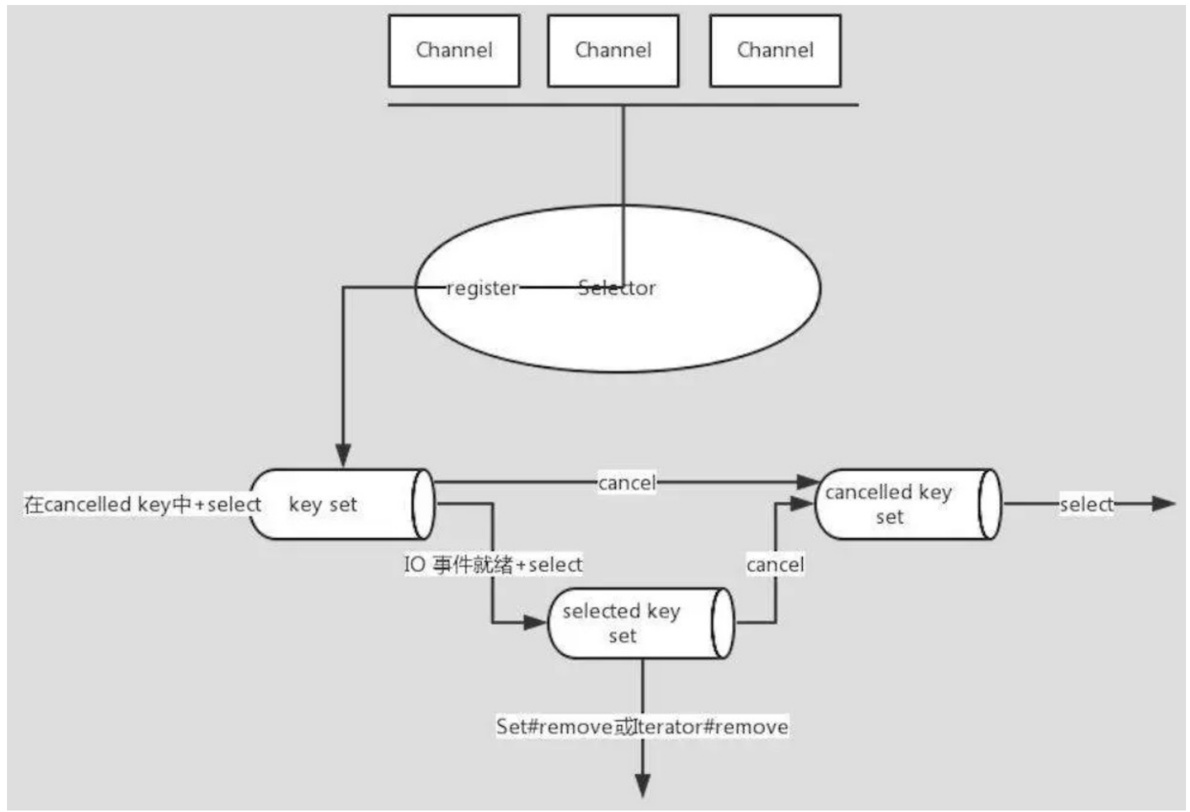
select方法的核心流程：

当应用层中Selector不断调用select()方法时

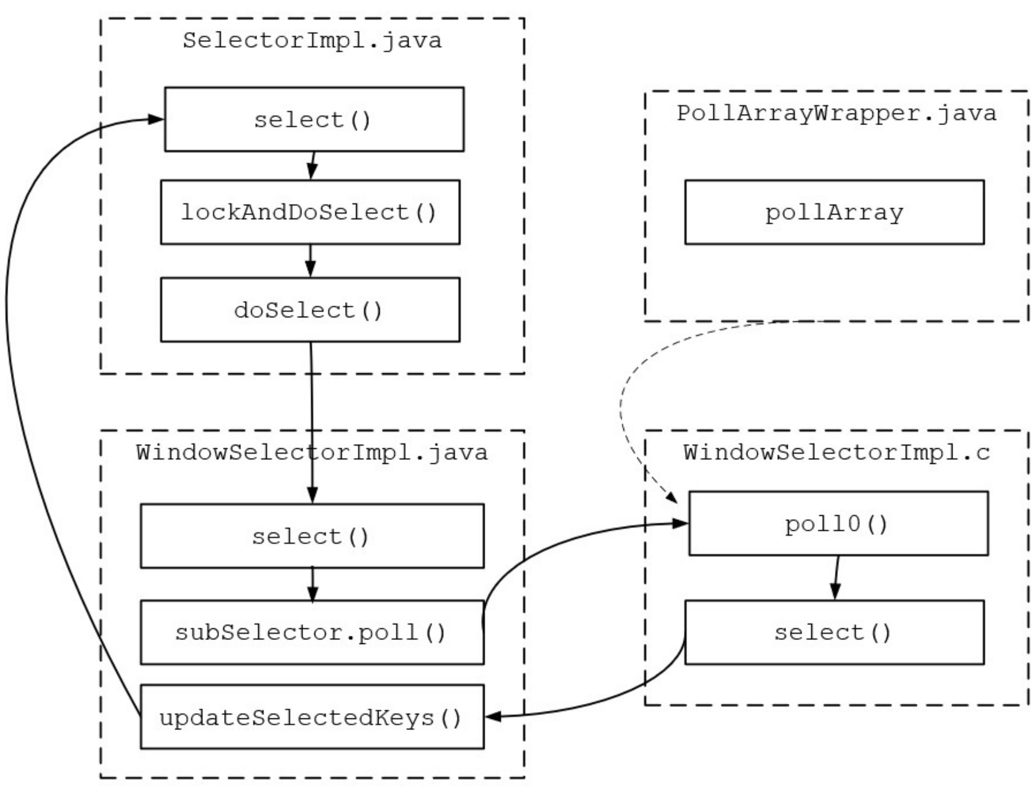
- 首先根据cancelledKeys去删除registeredKeys和selectedKeys对应的key以至取消对应的key
- 然后调用操作系统去做操作系统级别的select，一旦有registeredKeys感兴趣的事件，则将对应事件的key添加到selectedKeys中；
- 如selectedKeys已存在key了则将事件添加到key中readyOps（已准备好的事件集）中。

经过此番操作，当应用层调用Selector的selectedKeys()则取到被选中的key集，进而可以获取到感兴趣事件对应的channel，根据事件对channel进行操作。

事件的大致流转过程



select方法的调用流程



SelectorImpl#select

```
@Override
public final int select() throws IOException {
```

```
        return lockAndDoSelect(null, -1);
}

@Override
public final int selectNow() throws IOException {
        return lockAndDoSelect(null, 0);
}

@Override
public final int select(Consumer<SelectionKey> action, long timeout)
        throws IOException
{
        Objects.requireNonNull(action);
        if (timeout < 0)
                        throw new IllegalArgumentException("Negative timeout");
        return lockAndDoSelect(action, (timeout == 0) ? -1 : timeout);
}
```

SelectorImpl#lockAndDoSelect

```
private int lockAndDoSelect(Consumer<SelectionKey> action, long timeout)
throws IOException
{
    synchronized (this) {
            ensureOpen();
            if (inSelect)
                 throw new IllegalStateException("select in progress");
            inSelect = true;
            try {
                synchronized (publicSelectedKeys) {
                    return doSelect(action, timeout);
                }
            } finally {
                            inSelect = false;
            }
    }
}
```

WindowsSelectorImpl#doSelect

```
@Override
protected int doSelect(Consumer<SelectionKey> action, long timeout)
                throws IOException
{
    assert Thread.holdsLock(this);
    this.timeout = timeout; // set selector timeout
    processUpdateQueue();
    processDeregisterQueue();
    if (interruptTriggered) {
        resetWakeupSocket();
        return 0;
    }
```

```
    // Calculate number of helper threads needed for poll. If necessary
    // threads are created here and start waiting on startLock
    adjustThreadsCount();
    finishLock.reset(); // reset finishLock
    // Wakeup helper threads, waiting on startLock, so they start polling.
    // Redundant threads will exit here after wakeup.
    startLock.startThreads();
    // do polling in the main thread. Main thread is responsible for
    // first MAX_SELECTABLE_FDS entries in pollArray.
    try {
        begin();
        try {
            subSelector.poll();
        } catch (IOException e) {
            finishLock.setException(e); // Save this exception
        }
        // Main thread is out of poll(). Wakeup others and wait for them
        if (threads.size() > 0)
            finishLock.waitForHelperThreads();
    } finally {
            end();
    }
    // Done with poll(). Set wakeupSocket to nonsignaled  for the next run.
    finishLock.checkForException();
    processDeregisterQueue();
    int updated = updateSelectedKeys(action);
    // Done with poll(). Set wakeupSocket to nonsignaled  for the next run.
    resetWakeupSocket();
    return updated;
}
```

内部类subSelector#poll()

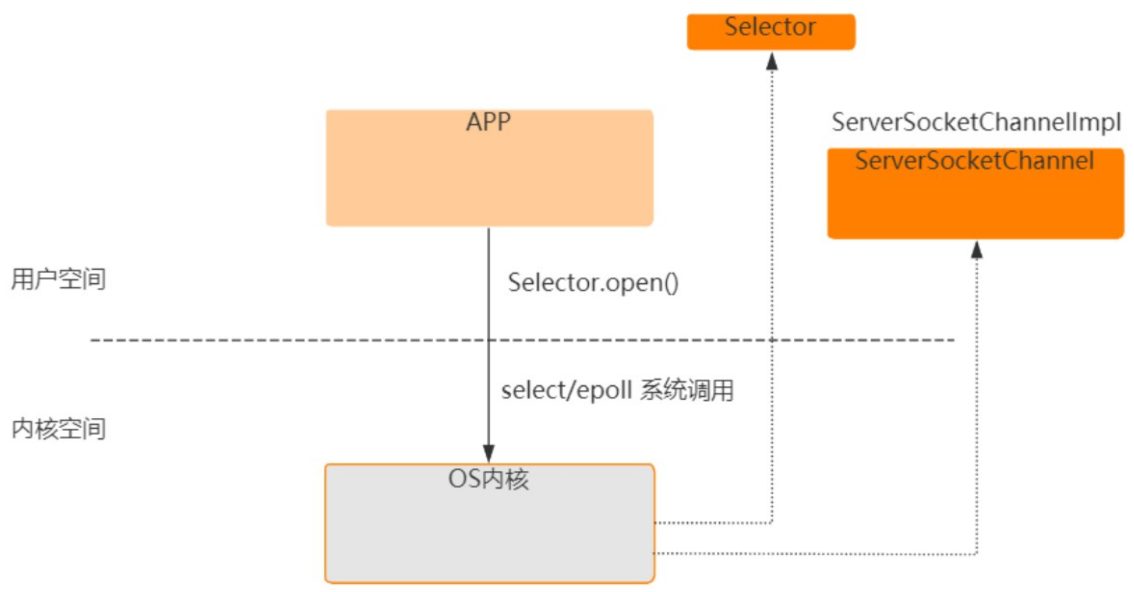
```
private int poll() throws IOException{ // poll for the main thread
        return poll0(pollWrapper.pollArrayAddress,
                Math.min(totalChannels, MAX_SELECTABLE_FDS),
                        readFds, writeFds, exceptFds, timeout);
}

private native int poll0(long pollAddress, int numfds,
                    int[] readFds, int[] writeFds, int[] exceptFds, long
timeout);
```

C语言中的WindowsSelectorImpl#poll0()

```
JNIEXPORT jint JNICALL
Java_sun_nio_ch_WindowsSelectorImpl_00024SubSelector_poll0(JNIEnv *env, jobject this,
                                   jlong pollAddress, jint numfds,
                                   jintArray returnReadFds, jintArray returnWriteFds,
                                   jintArray returnExceptFds, jlong timeout)
{
    /* Set FD_SET structures required for select */
    for (i = 0; i < numfds; i++) {
        if (fds[i].events & POLLIN) {
           readfds.fd_array[read_count] = fds[i].fd;
           read_count++;
        }
        if (fds[i].events & (POLLOUT | POLLCONN))
        {
           writefds.fd_array[write_count] = fds[i].fd;
           write_count++;
        }
       ...
    }
    /* Call select */
    if ((result = select(0 , &readfds, &writefds, &exceptfds, tv))
                                             == SOCKET_ERROR) {
       ....
    }
}
}
```

将fds指定的fd，解析到readfds，writefds和exceptfds中。等select返回后，结果信息再存放在readfds，writefds和exceptfds中。

描　述	select监控的操作		
	读	写	例外
有数据可读	•		
连接的读通道被关闭	•		
listen插口已经将连接排队	•		
插口差错未处理	•		
缓存可供写操作用，且一个连接存在或还没有连接请求		•	
连接的写通道被关闭		•	
插口差错未处理		•	
OOB同步标记未处理			•



c的select方法的主要功能

这里的c的select方法对应于内核中的sys_select调用。

sys_select首先将第二三四个参数指向的fd_set拷贝到内核，然后对每个被SET的描述符调用进行poll，并记录在第二三四个参数指向的fd_set中，如果有事件发生，select会将临时结果写到用户空间并返回；

当轮询一遍后没有任何事件发生时，如果指定了超时时间，则select会睡眠到超时，睡眠结束后再进行一次轮询，并将临时结果写到用户空间，然后返回。

在windows上select方法原型：

在windows上select方法原型：

```
int select(int nfds, FD_SET* readfds, FD_SET* writefds, FD_SET* exceptfds, const struct timeval* timeout);
```

- 第1个参数 :nfds起到兼容的作用，通常设置为0.
- 第2到第4个参数：Readfds，writefds，exceptfds分别用来保存对读，写，异常感兴趣的文件描述符。可以将FD_SET理解成一个存放fd的集合。同时在select返回时，这三个fd集合中的数据表示对应事件已经就绪的fd。这三个参数即作为入参又作为出参。
- 第5个参数：Timeout为超时参数，如果在指定时间内没有fd感兴趣的事件发生的话，select也会返回，只不过readfds，writefds和exceptfds中是没有结果数据的。

SubSelector.processSelectedKeys

> 分别对可读fd、可写fd、异常fd、分别调用 processFdSet

```java
private int processSelectedKeys(long updateCount, Consumer<SelectionKey> action) {
    int numKeysUpdated = 0;
    numKeysUpdated += processFDSet(updateCount, action, readFds,
                                   Net.POLLIN,
                                   false);
    numKeysUpdated += processFDSet(updateCount, action, writeFds,
                                   Net.POLLCONN |
                                   Net.POLLOUT,
                                   false);
    numKeysUpdated += processFDSet(updateCount, action, exceptFds,
                                   Net.POLLIN |
                                   Net.POLLCONN |
                                   Net.POLLOUT,
                                   true);
    return numKeysUpdated;
}
```

SubSelector.processFDSet

```java
    private int processFDSet(long updateCount,
                             int[] fds, int rOps,
                             boolean isExceptFds)
    {
        int numKeysUpdated = 0;
        for (int i = 1; i <= fds[0]; i++) {
            int desc = fds[i];
          ...
            MapEntry me = fdMap.get(desc);  //导航, fd 到 sk
            // If me is null, the key was deregistered in the previous
            // processDeregisterQueue.
            if (me == null)
                continue;
            SelectionKeyImpl sk = me.ski;

            if (WindowsSelectorImpl.this.selectedKeys.contains(sk)) {
                if (sk.channel.translateAndUpdateReadyOps(var4, sk)...) {
                    sk.updateCount = updateCount;
                   ....
                }
            }
        }
        return numKeysUpdated;
    }
}
```

ServerSocketChannelImpl.translateReadyOps

```
public boolean translateReadyOps(int ops, int initialOps,
                                 SelectionKeyImpl sk) {
    int intOps = sk.nioInterestOps(); // Do this just once, it synchronizes
    int oldOps = sk.nioReadyOps();
    int newOps = initialOps;
    ...

    if (((ops & PollArrayWrapper.POLLIN) != 0) &&
        ((intOps & SelectionKey.OP_READ) != 0) &&
        (state == ST_CONNECTED))
        newOps |= SelectionKey.OP_READ;

    ...

    sk.nioReadyOps(newOps);
    return (newOps & ~oldOps) != 0;
}
```

Selector的select()只用单线程

因为对这几个集合的操作不是线程安全的,所以一般使用Selector的select()只用单线程

而对于select得到的channel和对应的IO操作,可以新开线程或者使用线程池来处理。

这也正是IO复用的意义所在。

内部多个SelectThread的原理

由于select最大一次性获取1024个文件描述符。因此为了提高poll的性能WindowsSelectorImpl底层 通过引入多个辅助线程的方式实现多线程poll以提高高并发时的性能问题。我们先看一下注册的逻辑

```
protected void implRegister(SelectionKeyImpl ski) {
    synchronized (closeLock) {
        if (pollWrapper == null)
            throw new ClosedSelectorException();
        //判断是否需要扩容队列以及添加辅助线程
        growIfNeeded();
        //保存到缓存中
        channelArray[totalChannels] = ski;
        //保存在数组中的位置
        ski.setIndex(totalChannels);
        //保存文件描述符和SelectionKeyImpl的映射关系到FDMap
        fdMap.put(ski);
        //保存到keys中
        keys.add(ski);
        //保存文件描述符和事件到native数组中
        pollWrapper.addEntry(totalChannels, ski);
        totalChannels++;
    }
}
```

最强揭秘：Selector.wakeup() 唤醒的底层原理

wakeup核心原理

Selector对外提供了wakeup方法

```
public abstract Selector wakeup();
```

Selector中提供了使线程从被阻塞的select( )方法中优雅地退出的能力：

```
public abstract Selector wakeup();
```

wakeup()实现的功能：

- 如果一个线程在调用select()或select(long)方法时被阻塞，调用wakeup()会使线程立即从阻塞中唤醒；如果调用wakeup()期间没有select操作，下次调用select相关操作会立即返回，不会执行poll()，包括调用selectNow()。
- 在Select期间，多次调用wakeup()与调用一次效果是一样的。

注意：如果调用wakeup()期间没有select操作，后续若先调用一次selectNow()，再次调用select()则会导致阻塞。

如果一个线程在调用select()或select(long)方法时被阻塞，调用wakeup()会使线程立即从阻塞中唤醒；如果调用wakeup()期间没有select操作，下次调用select相关操作会立即返回。在Select期间，多次调用wakeup()与调用一次效果是一样的。

Selector关闭

当我们调用Selector的close()方法时，会首先执行wakeup操作，任何一个在选择操作中阻塞的线程都将被唤醒。同时会注销绑定在选择器上的所有通道，释放与此选择器相关联的任何其他资源。

如果选择已经处于关闭状态，再次调用close()方法不会由任何作用。若调用该选择器除close()和wakeup()之外的操作都会导致ClosedSelectorException异常。

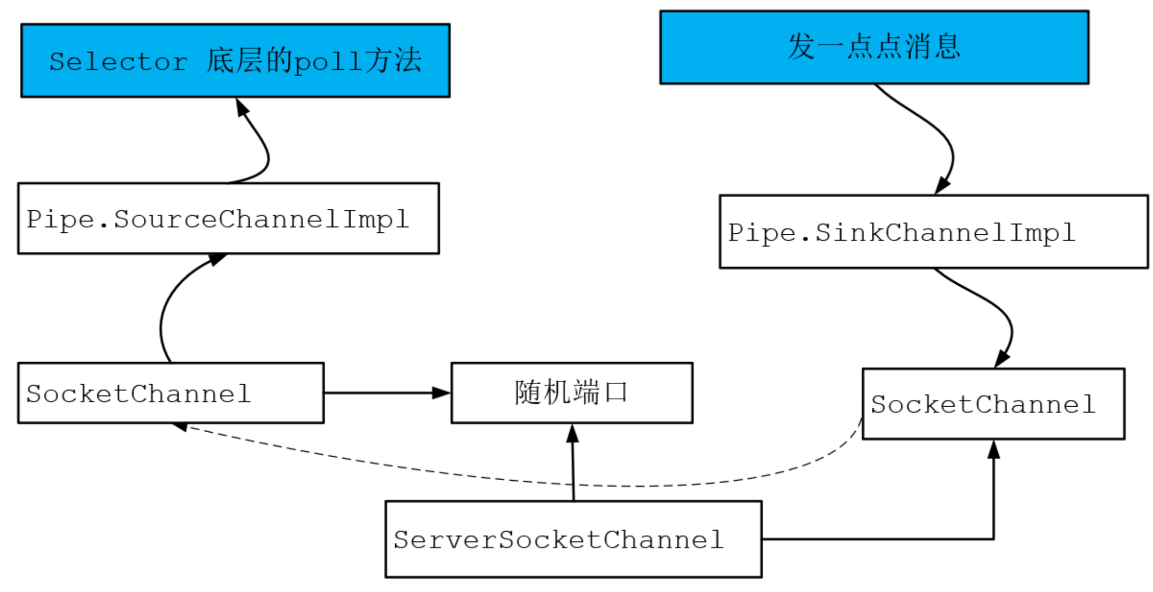
SelectorImpl#implCloseSelector

```
@Override
public final void implCloseSelector() throws IOException {
    wakeup();
    synchronized (this) {
        implClose();
        synchronized (publicSelectedKeys) {
            // Deregister channels
            Iterator<SelectionKey> i = keys.iterator();
            while (i.hasNext()) {
                SelectionKeyImpl ski = (SelectionKeyImpl)i.next();
                deregister(ski);
                SelectableChannel selch = ski.channel();
                if (!selch.isOpen() && !selch.isRegistered())
                            ((SelChImpl)selch).kill();
                            selectedKeys.remove(ski);
                            i.remove();
                }
            assert selectedKeys.isEmpty() && keys.isEmpty();
        }
    }
}
```

在Windows会建立一对自己和自己的loopback的TCP连接；在Linux上会开一对pipe（pipe在Linux下一般都是成对打开），那就是如果想要唤醒select，只需要朝着自己的这个loopback连接发点数据过去，于是，就可以唤醒阻塞在select上的线程了。

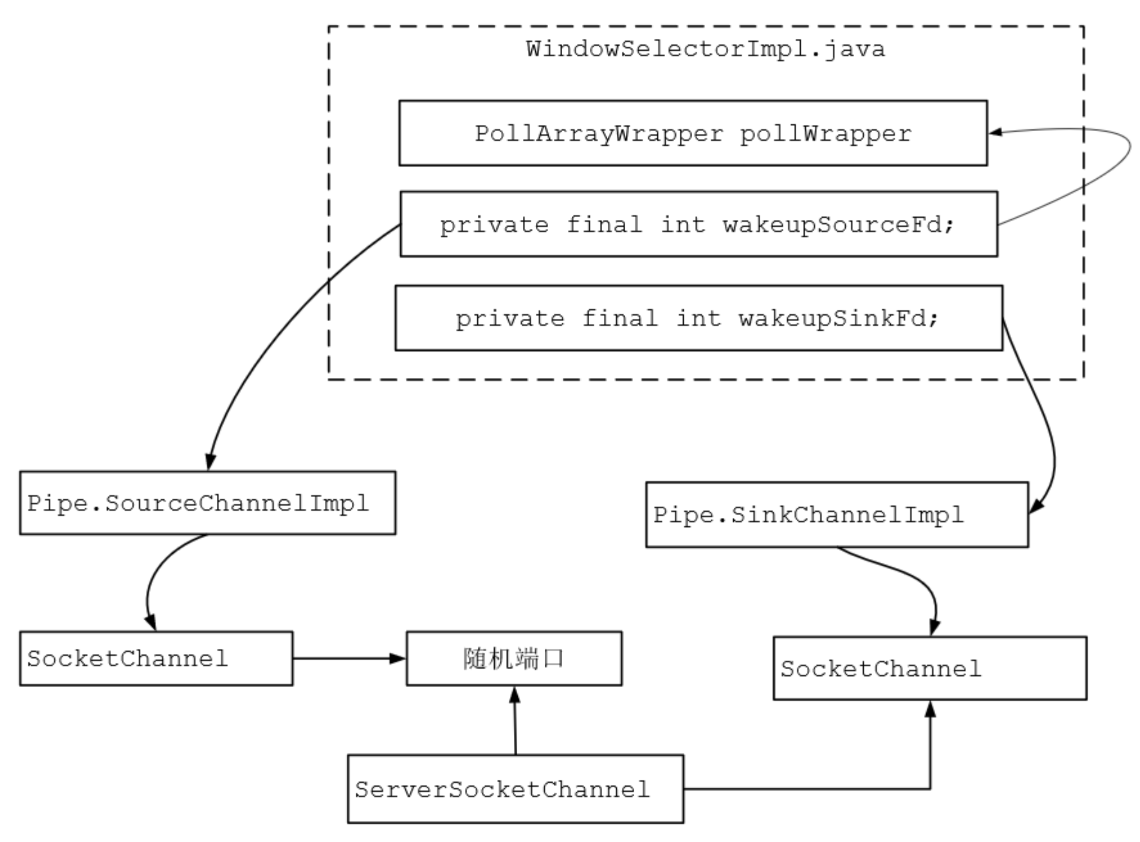
WindowsSelectorImpl的loopback连接

对于windows，每当调用一次Selector的open方法就建立了两个TCP的链接，一个Server绑定了一个随机的端口号，一个client连接，server和client相连获得，client作为**接受端source**，server的侧的传输通道channel作为**发送端sink**，如果要是实现wakeup，**发送端sink**就给这个**接受端source**发送一点儿数据就OK 了。



唤醒消息的发送端和接收端

```
WindowsSelectorImpl(SelectorProvider sp) throws IOException {
        super(sp);
        pollWrapper = new PollArrayWrapper(INIT_CAP);
        //打开管道
        wakeupPipe = Pipe.open();
        //获取接收端source的fd
        wakeupSourceFd = ((SelChImpl)wakeupPipe.source()).getFDVal();
        //获取唤醒消息的发送端sink
        SinkChannelImpl sink = (SinkChannelImpl)wakeupPipe.sink();
        // 禁用发送端 Nagle algorithm 使得 the wakeup is more immediate
        (sink.sc).socket().setTcpNoDelay(true);

        //持有发送端的fd，用于后续发送唤醒消息1
        wakeupSinkFd = ((SelChImpl)sink).getFDVal();

     //接收端的fd，加入poll队列
        pollWrapper.addWakeupSocket(wakeupSourceFd, 0);
}
```



PipeImpl的成员

```
class PipeImpl  extends Pipe
{

    // Number of bytes in the secret handshake.
```

```
    private static final int NUM_SECRET_BYTES = 16;

    // Random object for handshake values
    private static final Random RANDOM_NUMBER_GENERATOR = new SecureRandom();

    // Source and sink channels
    private SourceChannel source; //接收端通道
    private SinkChannel sink;  //发送端通道
    //获取接收端通道
    public SourceChannel source() {return source;}
    //获取发送端通道
    public SinkChannel sink() {return sink;}
}
```

pipe的简单实例

```
public class PipeDemo {
    @Test
    public void testPipe() throws Exception {
        String msg = "疯狂创客圈 高并发 研习社群!";
        Pipe pipe = Pipe.open();
        // 用于发送数据的SinkChannel
        Pipe.SinkChannel sinkChannel = pipe.sink();
        sinkChannel.write(ByteBuffer.wrap((msg).getBytes())); // 发送数据

        // 用于接收数据的SourceChannel
        Pipe.SourceChannel sourceChannel = pipe.source();
        ByteBuffer byteBuffer = ByteBuffer.allocate(msg.length());
        sourceChannel.read(byteBuffer);                    // 读取数据
        Logger.tcfo(new String(byteBuffer.array()));    // 打印数据
    }
}
```

PipeImpl#LoopbackConnector

```
private class LoopbackConnector implements Runnable {
@Override
public void run() {
ServerSocketChannel ssc = null;
SocketChannel sc1 = null;
SocketChannel sc2 = null;

try {
    // Create secret with a backing array.
    ByteBuffer secret = ByteBuffer.allocate(NUM_SECRET_BYTES);
    ByteBuffer bb = ByteBuffer.allocate(NUM_SECRET_BYTES);

    // Loopback address
```

```
    InetAddress lb = InetAddress.getLoopbackAddress();
    InetSocketAddress sa = null;
    for(;;) {
        // 在loopback address 上绑定一个监听通道
        if (ssc == null || !ssc.isOpen()) {
            ssc = ServerSocketChannel.open();
            ssc.socket().bind(new InetSocketAddress(lb, 0));
            //保持一下操作系统分配的端口
            sa = new InetSocketAddress(lb, ssc.socket().getLocalPort());
        }

        // 发起建立连接
        sc1 = SocketChannel.open(sa);
        RANDOM_NUMBER_GENERATOR.nextBytes(secret.array());
        do {
            sc1.write(secret); //写入校验密码
        } while (secret.hasRemaining());
        secret.rewind();

        // 接收连接，然后进行校验
        sc2 = ssc.accept();
        do {
            sc2.read(bb);
        } while (bb.hasRemaining());
        bb.rewind();

        if (bb.equals(secret)) //校验结果正确
            break;
        sc2.close();//校验结果不正确
        sc1.close();//校验结果不正确
    }

    // Create source and sink channels
    //接收端通道
    source = new SourceChannelImpl(sp, sc1);
    //发送端通道
    sink = new SinkChannelImpl(sp, sc2);
} catch (IOException e) {
...
}
}
}
```

创建pipe的过程，pipe.open()，打开本机通道

```
//获取本地地址
  InetAddress inetaddress = InetAddress.getByName("127.0.0.1");
  if (!$assertionsDisabled && !inetaddress.isLoopbackAddress())
      throw new AssertionError();
  //打开一个ServerSocket通道
  serversocketchannel = ServerSocketChannel.open();

  //ServerSocket通道绑定地址
  serversocketchannel.socket().bind(new InetSocketAddress(inetaddress, 0));
  InetSocketAddress inetsocketaddress =
      new InetSocketAddress(inetaddress, serversocketchannel.socket().getLocalPort());


  //打开一个SocketChannel通道
  socketchannel = SocketChannel.open(inetsocketaddress);
```

创建pipe的过程，两个socket的链接校验

这里即为上图中最下面那部分创建pipe的过程，windows下的实现是创建两个本地的socketChannel,然后连接（链接的过程通过写一个随机long做两个socket的链接校验），两个socketChannel分别实现了管道的source与sink端。

```
try {
    // Create secret with a backing array.
    ByteBuffer secret = ByteBuffer.allocate(NUM_SECRET_BYTES);
    ByteBuffer bb = ByteBuffer.allocate(NUM_SECRET_BYTES);

    // Loopback address
    InetAddress lb = InetAddress.getLoopbackAddress();
    InetSocketAddress sa = null;
    for(;;) {
        // 在loopback address 上绑定一个监听通道
        if (ssc == null || !ssc.isOpen()) {
            ssc = ServerSocketChannel.open();
            ssc.socket().bind(new InetSocketAddress(lb, 0));
            //保持一下操作系统分配的端口
            sa = new InetSocketAddress(lb, ssc.socket().getLocalPort());
        }

        // 发起建立连接
        sc1 = SocketChannel.open(sa);
        RANDOM_NUMBER_GENERATOR.nextBytes(secret.array());
        do {
            sc1.write(secret); //写入校验密码
        } while (secret.hasRemaining());
        secret.rewind();

        // 接收连接，然后进行校验
        sc2 = ssc.accept();
        do {
            sc2.read(bb);
        } while (bb.hasRemaining());
        bb.rewind();

        if (bb.equals(secret)) //校验结果正确
            break;
        sc2.close();//校验结果不正确
        sc1.close();//校验结果不正确
    }
```

回到WindowsSelectorImpl构造方法

```
wakeupSourceFd = ((SelChImpl)wakeupPipe.source()).getFDVal();

        // Disable the Nagle algorithm so that the wakeup is more immediate
        SinkChannelImpl sink = (SinkChannelImpl)wakeupPipe.sink();
        (sink.sc).socket().setTcpNoDelay(true);
        wakeupSinkFd = ((SelChImpl)sink).getFDVal();
        pollWrapper.addWakeupSocket(wakeupSourceFd, 0);
```

可以看到这里SinkChannel禁用了TCP中的Nagle算法，并且把SourceChannel的fd(文件描述符)放入到了pollWrapper中

Nagel算法

TCP/IP协议中，无论发送多少数据，总是要在数据前面加上协议头，同时，对方接收到数据，也需要发送ACK表示确认。为了尽可能的利用网络带宽，TCP总是希望尽可能的发送足够大的数据。（一个连接会设置MSS参数，因此，TCP/IP希望每次都能够以MSS尺寸的数据块来发送数据）。Nagle算法就是为了尽可能发送大块数据，避免网络中充斥着许多小数据块。

Nagle算法的基本定义是任意时刻，最多只能有一个未被确认的小段。 所谓"小段"，指的是小于MSS尺寸的数据块，所谓"未被确认"，是指一个数据块发送出去后，没有收到对方发送的ACK确认该数据已收到。

Nagle算法的规则（可参考tcp_output.c文件里tcp_nagle_check函数注释）：
（1）如果包长度达到MSS，则允许发送；
（2）如果该包含有FIN，则允许发送；
（3）设置了TCP_NODELAY选项，则允许发送；
（4）未设置TCP_CORK选项时，若所有发出去的小数据包（包长度小于MSS）均被确认，则允许发送；
（5）上述条件都未满足，但发生了超时（一般为200ms），则立即发送。

为啥要禁用了TCP中的Nagle算法?

为了及时得把一个很短的消息传递到对方

短到什么程度呢？ 一个字节1

回到WindowsSelectorImpl构造方法

```
wakeupSourceFd = ((SelChImpl)wakeupPipe.source()).getFDVal();

        // Disable the Nagle algorithm so that the wakeup is more immediate
        SinkChannelImpl sink = (SinkChannelImpl)wakeupPipe.sink();
        (sink.sc).socket().setTcpNoDelay(true);
        wakeupSinkFd = ((SelChImpl)sink).getFDVal();
        pollWrapper.addWakeupSocket(wakeupSourceFd, 0);
```

把source端放到了pollWrapper中第一个轮询通道的位置

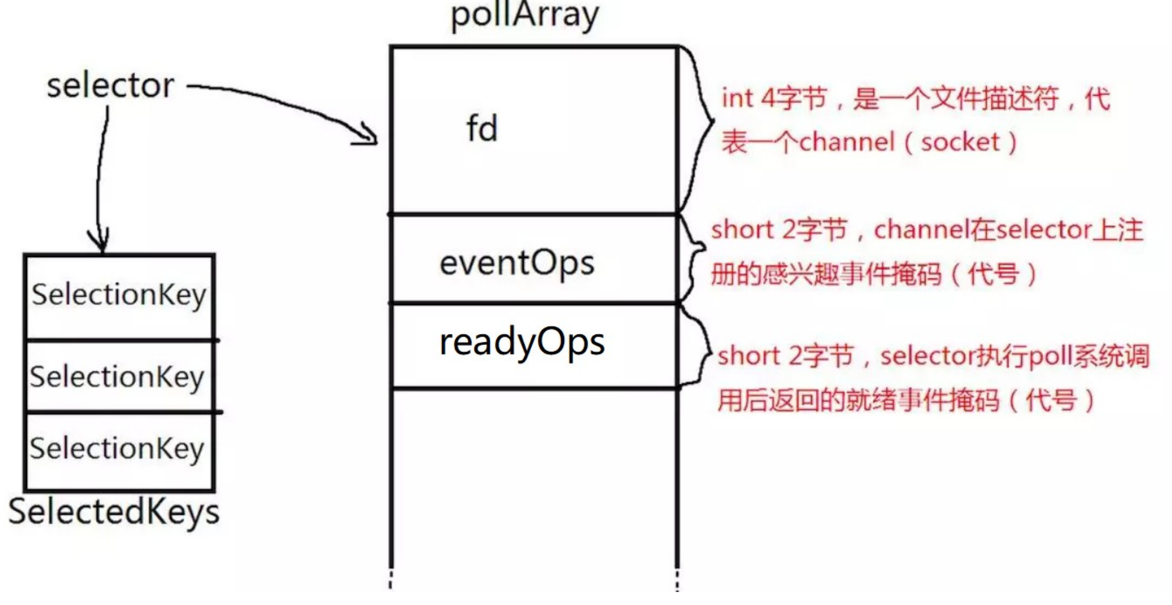
---

source端放到了pollWrapper中

```
    private AllocatedNativeObject pollArray; // The fd array
// Adds Windows wakeup socket at a given index.

    void addWakeupSocket(int fdVal, int index) {
        putDescriptor(index, fdVal);
        putEventOps(index, POLLIN);
    }
// Access methods for fd structures
    void putDescriptor(int i, int fd) {
        pollArray.putInt(SIZE_POLLFD * i + FD_OFFSET, fd);
    }
    void putEventOps(int i, int event) {
        pollArray.putShort(SIZE_POLLFD * i + EVENT_OFFSET, (short) event);
    }
```

回顾：pollArray的结构

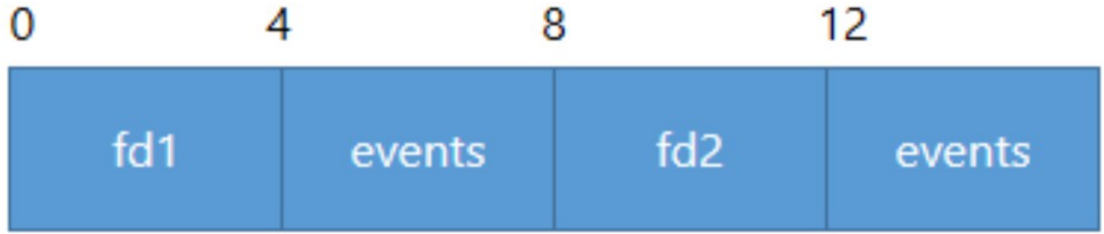
pollArray用Unsafe类申请一块物理内存，存放注册时的socket句柄fdVal和event的数据结构，其中pollfd共8字节，0~3字节保存socket句柄，4~7字节保存event。



---

pollArray的元素

pollArray中存放的内容是一个个8字节的数据单元。这个8字节数据有两部分组成，每部分占用4个字节，分别是fd信息和这个fd关注的事件events，这两个信息在8字节数据单元中的偏移量分别由FD_OFFSET和EVENT_OFFSET标识，也就是0和4。pollArray中的数据结构如下：




```
private AllocatedNativeObject pollArray; //一段连续的内存空间
long pollArrayAddress;   // PollArrayAddress是pollArray的起始地址
@Native private static final short FD_OFFSET     = 0; // fd offset in pollfd
@Native private static final short EVENT_OFFSET  = 4; // events offset in pollfd
static short SIZE_POLLFD = 8; // sizeof pollfd struct
private int size; // Size of the pollArray
```

---

wakeup的消息发送

- WindowsSelectorImpl的wakeup实现

回顾WindowsSelectorImpl的成员

名称	作用
SelectionKeyImpl[] channelArray	存放注册的SelectionKey
PollArrayWrapper pollWrapper	底层的本机轮询数组包装对象，用于存放Socket文件描述符和事件掩码
List threads	辅助线程，多个线程有助于提高高并发时的性能
Pipe wakeupPipe	用于唤醒poll线程，包括主线程和辅助线程
FdMap fdMap	保存文件描述符和SelectionKey的映射关系
SubSelector subSelector	调用JNI的poll和处理就绪的SelectionKey
StartLock startLock	新增的辅助线程使用该锁等待主线程的开始信号
FinishLock finishLock	主线程用该锁等待所有辅助线程执行完毕

WindowsSelectorImpl的wakeup实现

```
public Selector wakeup() {
        synchronized(this.interruptLock) {
            if (!this.interruptTriggered) {
                this.setWakeupSocket();
                this.interruptTriggered = true;
            }

            return this;
        }
    }

    private void setWakeupSocket() {
        this.setWakeupSocket0(this.wakeupSinkFd);
    }

   private native void setWakeupSocket0(int var1);
```

可见wakeup()是通过pipe的write 端send(scoutFd, &byte, 1, 0)，发生一个字节1，来唤醒poll（）。所以在需要的时候就可以调用selector.wakeup()来唤醒selector。

WindowsSelectorImpl#setWakeupSocket0

通过c语言的send方法，发生一个字节1，来唤醒底层的 poll（）

```
JNIEXPORT void JNICALL
Java_sun_nio_ch_WindowsSelectorImpl_setWakeupSocket0(JNIEnv *env, jclass this,
                                                 jint scoutFd)
{
    /* Write one byte into the pipe */
    const char byte = 1;
    send(scoutFd, &byte, 1, 0);
}
```

FdMap实际上是一个HashMap，key为选择key的文件描述id，value为MapEntry，选择key的包装

```
private static final class FdMap extends HashMap
{
    static final long serialVersionUID = 0L;
    private FdMap()
    {
    }
    //根据key文件描述id获取key
    private MapEntry get(int i)
    {
        return (MapEntry)get(new Integer(i));
    }
    //添加key
    private MapEntry put(SelectionKeyImpl selectionkeyimpl)
    {
        return (MapEntry)put(
            new Integer(selectionkeyimpl.channel.getFDVal()),
                            new MapEntry(selectionkeyimpl));
    }
    ...
}
```

```
class PollArrayWrapper {
    private AllocatedNativeObject pollArray;
    long pollArrayAddress;
    private static final short FD_OFFSET = 0;
    private static final short EVENT_OFFSET = 4;
    static short SIZE_POLLFD = 8;
    private int size;

    PollArrayWrapper(int var1) {
        int var2 = var1 * SIZE_POLLFD;
        this.pollArray = new AllocatedNativeObject(var2, b: true);
        this.pollArrayAddress = this.pollArray.address();
        this.size = var1;
    }
```

PollArrayWrapper pollWrapper：内部创建了一个上述介绍的AllocatedNativeObject对象（用于存放注册的Channel），而pollWrapper则更像是一个工具类，来方便的用户操作AllocatedNativeObject对象，pollWrapper把普通的操作都转化成对内存的操作，如下图所示

```
// Access methods for fd structures
int getEventOps(int i) {
    int offset = SIZE_POLLFD * i + EVENT_OFFSET;
    return pollArray.getShort(offset);
}

int getReventOps(int i) {
    int offset = SIZE_POLLFD * i + REVENT_OFFSET;
    return pollArray.getShort(offset);
}

int getDescriptor(int i) {
    int offset = SIZE_POLLFD * i + FD_OFFSET;
    return pollArray.getInt(offset);
}

void putEventOps(int i, int event) {
    int offset = SIZE_POLLFD * i + EVENT_OFFSET;
    pollArray.putShort(offset, (short)event);
}

void putReventOps(int i, int revent) {
    int offset = SIZE_POLLFD * i + REVENT_OFFSET;
    pollArray.putShort(offset, (short)revent);
}

void putDescriptor(int i, int fd) {
    int offset = SIZE_POLLFD * i + FD_OFFSET;
    pollArray.putInt(offset, fd);
}
```

第三步：一旦第二步返回就说明有事件或者超时了，一旦有事件，则linux的poll调用会把产生的事件遍历的赋值到poll调用指定的地址上，即我们指定的一个个pollfd结构体，映射到java对象就是PollArrayWrapper的AllocatedNativeObject，这时候我们获取事件就是遍历底层的每一个地址，拿到pollfd结构体中的revents，如果revents不为0代表发生了事件，还要与Channel关注的事件进行相&操作，不为0代表发生了Channel关注的事件了，并清空pollfd结构体中的revents数据供下次使用，代码如下

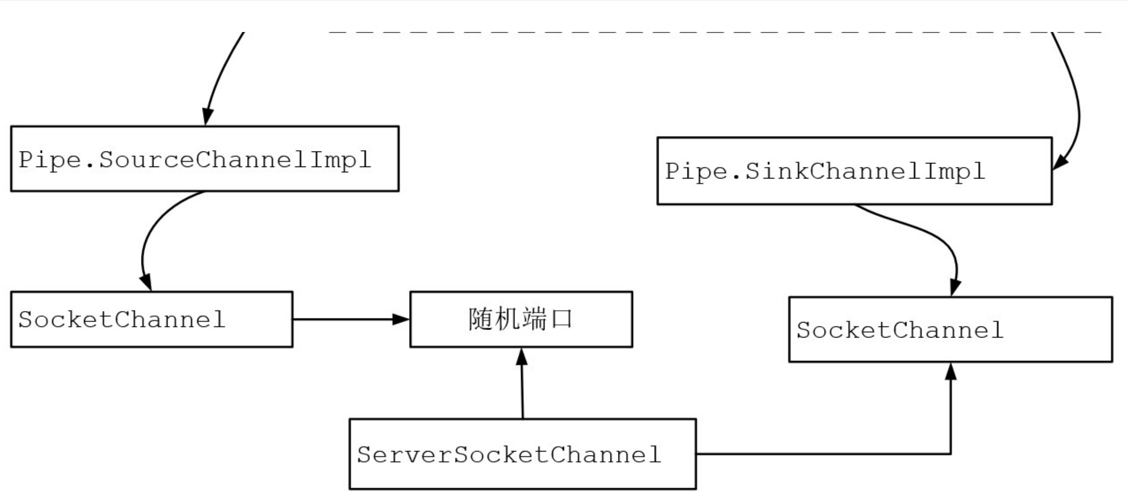
```java
protected int updateSelectedKeys() {
    int numKeysUpdated = 0;
    // Skip zeroth entry; it is for interrupts only
    for (int i=channelOffset; i<totalChannels; i++) {
        int rOps = pollWrapper.getReventOps(i);
        if (rOps != 0) {
            SelectionKeyImpl sk = channelArray[i];
            pollWrapper.putReventOps(i, 0);
            if (selectedKeys.contains(sk)) {
                if (sk.channel.translateAndSetReadyOps(rOps, sk)) {
                    numKeysUpdated++;
                }
            } else {
                sk.channel.translateAndSetReadyOps(rOps, sk);
                if ((sk.nioReadyOps() & sk.nioInterestOps()) != 0) {
                    selectedKeys.add(sk);
                    numKeysUpdated++;
                }
            }
        }
    }
    return numKeysUpdated;
}
```

这里就是通过指针操作直接获取对应底层结构体的revents数据。

我们知道PollSelectorImpl在select过程的阻塞时间受控于所注册的Channel的事件，一旦有事件才会进行返回，没有事件的话就一直阻塞，为了可以允许手动控制这种局面的话，就额外增加了一个监控，即对pipe的读监控。对pipe的读文件描述符即fd0注册到PollArrayWrapper中的第一个位置，如果我们对pipe的写文件描述符fd1进行写数据操作，则pipe的读文件描述符必然会收到读事件，即可以使PollSelectorImpl不再阻塞，立即返回。

```java
 ByteBuffer bytebuffer = ByteBuffer.allocate(8);
//获取通道的随机long值
  long l = PipeImpl.rnd.nextLong();
  bytebuffer.putLong(l).flip();
  //向serverSocket通道发送一个long值，即8个字节
  socketchannel.write(bytebuffer);
  do {
      //serverSocket接受连接
      socketchannel1 = serversocketchannel.accept();
      bytebuffer.clear();
      //接受client通道端发送过来的数据
      socketchannel1.read(bytebuffer);
      bytebuffer.rewind();
      if (bytebuffer.getLong() == l)   break;
      socketchannel1.close();
  } while (true);
```

```
//根据client通道，构造SourceChannelImpl
//作为管道的接收端
source = new SourceChannelImpl(sp, socketchannel);
//根据ServerChannel接受连接产生的SocketChannel通道
//构造SinkChannelImpl
 //作为管道的发送端
sink = new SinkChannelImpl(sp, socketchannel1);
```



this.pollWrapper.addWakeupSocket(this.wakeupSourceFd, 0);

```
void addWakeupSocket(int var1, int var2) {
    this.putDescriptor(var2, var1);
    this.putEventOps(var2, Net.POLLIN);
}
// 保存文件描述符fd
void putDescriptor(int var1, int var2) {
    this.pollArray.putInt(SIZE_POLLFD * var1 + 0, var2);
}
// 保存兴趣事件ops
void putEventOps(int var1, int var2) {
    this.pollArray.putShort(SIZE_POLLFD * var1 + 4, (short)var2);
}
```

这里将source的POLLIN事件标识为感兴趣的，当sink端有数据写入时，source对应的文件描述符wakeupSourceFd就会处于就绪状态

poll函数的事件标志符值

常量	说明
POLLIN	普通或优先级带数据可读
POLLRDNORM	普通数据可读
POLLRDBAND	优先级带数据可读
POLLPRI	高优先级数据可读
POLLOUT	普通数据可写
POLLWRNORM	普通数据可写
POLLWRBAND	优先级带数据可写
POLLERR	发生错误
POLLHUP	发生挂起
POLLNVAL	描述字不是一个打开的文件

服务端连接过程

1、创建ServerSocketChannel实例serverSocketChannel，并bind到指定端口。
2、创建Selector实例selector；
3、将serverSocketChannel注册到selector，并指定事件OP_ACCEPT。
4、while循环执行：
4.1、调用select方法，该方法会阻塞等待，直到有一个或多个通道准备好了I/O操作或等待超时。
4.2、获取选取的键列表；
4.3、循环键集中的每个键：
4.3.a、获取通道，并从键中获取附件（如果添加了附件）；
4.3.b、确定准备就绪的操纵并执行，如果是accept操作，将接收的信道设置为非阻塞模式，并注册到选择器；
4.3.c、如果需要，修改键的兴趣操作集；
4.3.d、从已选键集中移除键

在步骤3中，selector只注册了serverSocketChannel的OP_ACCEPT事件

- 如果有客户端A连接服务，执行select方法时，可以通过serverSocketChannel获取客户端A的socketChannel，并在selector上注册socketChannel的OP_READ事件。
- 如果客户端A发送数据，会触发read事件，这样下次轮询调用select方法时，就能通过socketChannel读取数据，同时在selector上注册该socketChannel的OP_WRITE事件，实现服务器往客户端写数据。