题目：epoll浅析

写在前面：分享也是一种讨论交流的形式，本次分享中有一些我个人的理解和观点，越深入，发现自己越无知，欢迎大家讨论交流/批评指正

一. epoll简介：10 min

1. IO多路复用 2 min

IO Multiplexing:

epoll: 是什么

I/O event notification facility，Linux 内核的I/O 事件通知机制，其最大的特点就是性能优异

基于事件的I/O多路复用技术和non-blocking IO网络编程是高性能并发网络服务器程序设计的主流模式

I/O多路复用

I/O：文件描述符fd，尤其指网络连接sock fd的，input/output

多路：多个连接

复用：复用的是线程

一言以蔽之，设计IO多路复用的API目的：优化，内核与监视进程的交互，成本

syscall:

select/poll(Unix, Solaris)/epoll(only Linux)/kqueue(BSD,Macos)/IOCP(win)简介，及其他平台的

见名知意：

select: 从fd中选择ready的进行处理

poll：轮询fd集合

epoll：effective/event poll：高效的poll

epoll等API发布时间线：15 s

1983，socket 发布在 Unix(4.2 BSD)

1983，select 发布在 Unix(4.2 BSD)

1994，Linux 的 1.0，已经支持 socket 和 select，

1997，poll 发布在 Linux 2.1.23

2002，epoll 发布在 Linux 2.5.44 ，内核2.6趋于稳定

C10K/100K/1M问题

服务器编程的基本任务就是处理并发连接，单机每秒处理的并发连接数(10k,100k,1m)

解决方案：1. 应用程序和操作系统配合，感知事件发生 2. 调度事件，分配进程/线程/协程处理事件

5种IO模型:

阻塞I/O+进程(池)/线程(池)

异步I/O+回调

非阻塞I/O：

核心思想是避免阻塞在read()/write()或其他IO syscall上，这样可以最大限度地复用线程，让一个线程可以服务多个socket连接

Non-blocking IO+事件ready通知+多线程

同步模式	阻塞IO	进程挂起	
	非阻塞IO	需要轮询	
	IO多路复用	epoll等	
POSIX定义 同步	信号驱动IO	Unix SIGIO/SIGPOLL	通知IO可读/写发生事件，执行读写和业务逻辑
异步模式	异步IO	POSIX AIO-->朝内核化发展	通知IO可读/写完成事件，执行业务逻辑

IO多路复用/信号驱动IO/异步IO的核心目的：

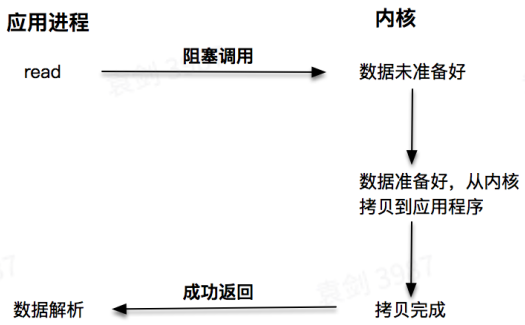
同时检查/监控多个fd

epoll比他们的优点：1，2，3，4，

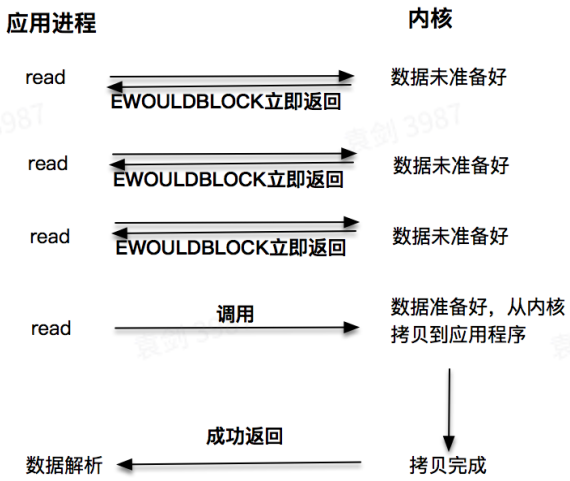
同步，异步，阻塞，非阻塞

同步	阻塞
异步	非阻塞

阻塞 指的是 进程调用接口后 如果接口没有准备好数据，那么这个进程会被挂起 什么也不能做。直到有数据返回时唤醒。



非阻塞 就是进程调用接口后 如果接口没有准备好数据，进程也能处理后续的操作。但是需要不断的去轮询检查 数据是否已经处理完成。



阻塞/非阻塞针对的是： 如果接收不到数据 当前进程的状态。

同步/异步针对的是： 如果接收不到数据当前代码逻辑的状态。

同步/异步是相对的概念：

参考对象：1. 程序执行流。 2. IO：内核/用户态之间拷贝数据的过程。 3.事件发生：包括fd可读/写

同步 指进程 调用接口时 需要等待接口处理完数据并相应进程才能继续执行。这里重点是数据处理完成 并返回

异步 指进程调用接口后，不必等待数据准备完成 可以继续执行。后续数据准备好通过一定的方式获得 例如回调。这里重点是 服务器也必须支持异步操作。不然没法返回数据。

那么获取数据的方式不一样所以编程的复杂度也不一样。

POSIX定义：

同步IO操作指：导致请求进程阻塞，直到IO操作完成

异步IO操作指：不导致请求进程阻塞

解释这些概念，举个生活中的例子，去一餐厅吃饭，人很多得排号，那么有 以下几种结果：

同步阻塞	门口等，一直等到叫自己
同步非阻塞	去附近逛逛，但是得时刻关注是不是排到了(i.e. 轮询)，万一排到人不在门口…
异步阻塞	店家提供短信通知服务，但是自己还是选择门口等（不存在）
异步非阻塞	店家提供短信通知服务，我可以附近逛逛，干干自己的事情，随时等候短信通知

2. select/poll/epoll对比 3-5 min

系统调用	select	poll	epoll
事件集合	内核会修改用户注册的文件描述符集来反馈其中的就绪事件，这使得用户每次调用select都要重置这三个文件描述符集	使用pollfd.events传入事件，使用pollfd.revents反馈就绪事件	使用内核事件表来管理用户事件，epoll_wait的events仅用来保存就绪事件
应用程序索引就绪文件描述符的时间复杂度	O(n)	O(n)	O(1)
最大支持文件描述符个数	有限制，一般是1024	65535	65535
工作模式	LT	LT	支持ET高效模式
内核实现	采用轮询方式	采用轮询方式	采用回调方式

poll与select不同：

A. 通过一个pollfd数组向内核传递需要关注的事件，故没有描述符个数的限制，（增加select监控的fd数：微调内核）

B. pollfd中的events字段和revents分别用于标志关注的事件和发生的事件，故pollfd数组只需要被初始化一次。

C. 但，select和poll是将这个内核fd列表维持在用户态，然后传递到内核中。

epoll是poll的一种优化:

a. 指拷贝ready的事件集合

b. 不需要对所有的fd进行遍历，O(1)回调。

c. 支持ET模式

select/poll可以移植性好，

开销问题：epoll性能优异，为什么

1. 拷贝开销: 减少内核/用户态的信息传递的开销

   每次调用select/poll，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大；而且如果epoll在ET模式下，active的fd会更少一些，拷贝性能损耗会更小。

2. 申请/释放内核内存开销:

   select/poll涉及到内核空间申请内存，释放内存，先尝试申请stack上资源, 但如果需要监听的 fd 数量N 比较大, 就会去申请heap空间的资源，这是非常耗时的。而, epoll 维护了一个红黑树，通过对这棵黑红树进行操作，可以避免大量的内存申请和释放的操作，而且logN查找速度非常快

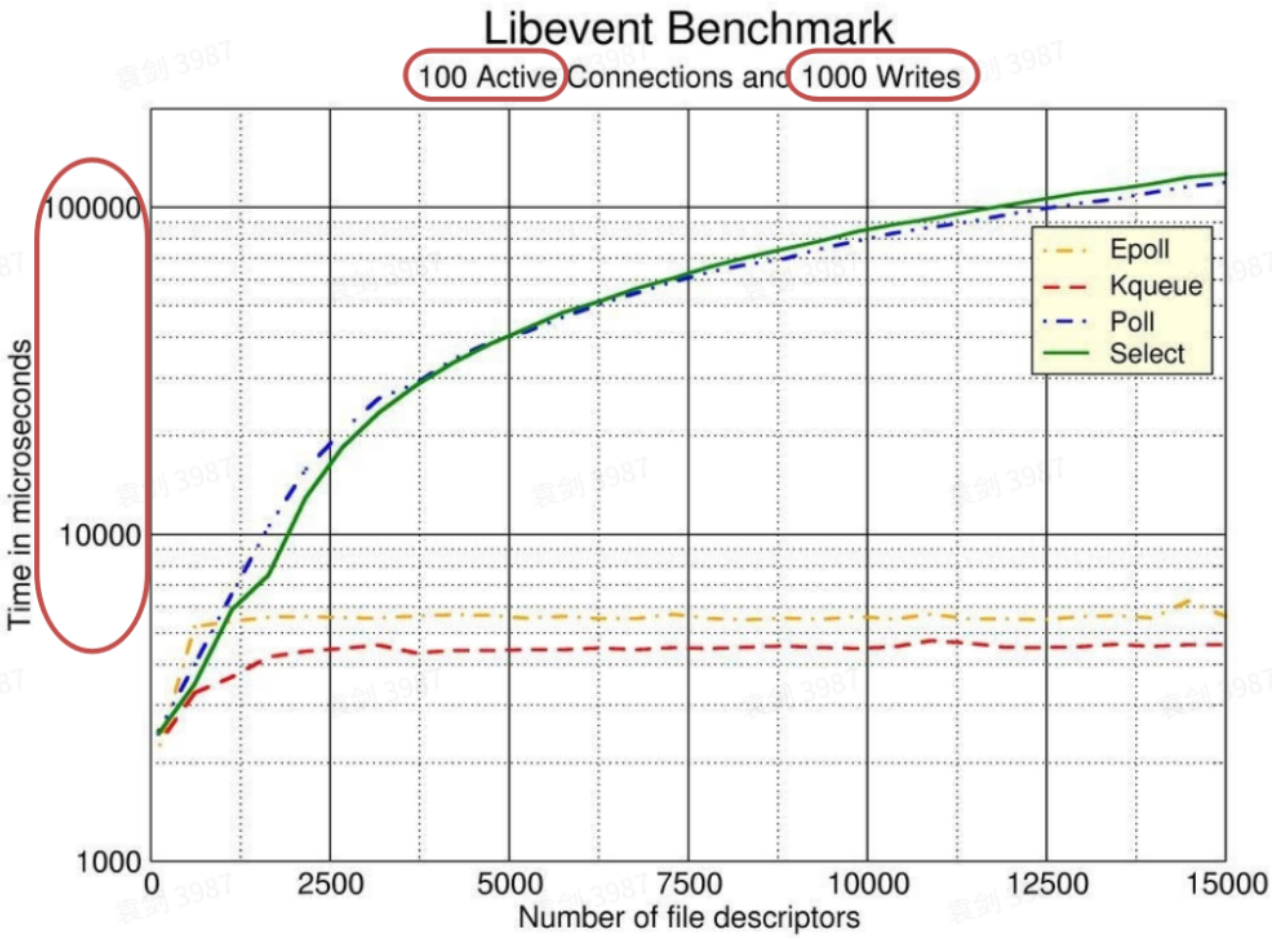
3. 轮询开销：O(N) v.s. O(1), select/poll都需要在内核遍历传递进来的所有fd，O(N)，这个开销在fd很多时也很大; epoll基于回调, O(1)，不随着fd数量增加而线性下降

性能对比：epoll不是银弹，其适用场景：

这是一个限制了 100 个活跃连接的，且每个连接发生 1000 次读写操作为止的基准测试。

纵轴：是请求的响应时间，单位微秒，

横轴：是持有的 socket 句柄数量



这图来自libevent的文档。

随着句柄数量的增加，epoll 和 kqueue 响应时间几乎无变化，而 poll 和 select 的响应时间却增长了非常多。

可以看出来，epoll 性能是很高的，并且随着监听的文件描述符的增加，epoll 的优势更加明显。

不过，这里限制的 100 个连接很重要。epoll 在应对大量网络连接时，只有活跃连接比列较少的情况下才能表现的性能优异。

总结，epoll的适用场景：

epoll 在处理大量非活跃的连接时，性能才会表现的优异。

fd较少，或active fd的比列较高时，epoll不见得比poll更高效，比如假设15000 个 socket 都是活跃的，epoll 和 poll其实差不了太多。

4. 常用C/C++/Golang相关库/框架：2 min

epoll业界的常见应用：基于epoll的网络库：

libevent(c)/netpoll(go)/muduo(c++)/netty(java)，等non-blocking I/O+I/O multiplexing库

基于网络库的应用层应用：

rpc/http/lb/mq/cache/：

kitex/gin/nginx/zeromq/redis

（nginx默认采用ET模式）

这些框架/lib的基石，IO多路复用，

二. epoll的三个接口 20 min

1. epoll_create 2min

用来创建一个 epoll 描述符（ 就是创建了一个 epoll ）

Plain Text

```
1    int epoll_create(int size); //size本来..., 2.6.8之后, size没用处，但需为正，向后兼容，
2    int epoll_create1(int flags); //EPOLL_CLOEXEC，打开的文件描述符在执行exec调用新程序前自动被关闭，比如fork，然后exec替换程序镜像，防止延长fd生命周期
```

epoll_create() 方法创建了一个 epoll 实例， epoll_ctl 和 epoll_wait会用到

参数 size 忽略，但仍需大于 0 整数。

epoll_create1() 的用法和 epoll_create() 基本一致，如果 epoll_create1() 的输入 flags 为 0，则和 epoll_create() 一样，内核自动忽略。

可以增加如 EPOLL_CLOEXEC 的额外选项，如果你有兴趣的话，可以研究一下这个选项有什么意义。

2. epoll_ctl 3 min

操作 epoll 中的 event；

```
Plain Text
1  int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

参数有：

epfd：

fd：

op：

参数	含义
EPOLL_CTL_ADD	添加一个新的epoll事件
EPOLL_CTL_DEL	删除一个epoll事件
EPOLL_CTL_MOD	改变一个监听事件的类型等

event：

而事件类型有七种，常用其中的三种：

宏	事件类型
EPOLLIN	可读
EPOLLOUT	数据
EPOLLERR	发生错误

Ready notification mode:

- LT：EPOLLLE
- ET：EPOLLET 目的：1. 减少事件的触发而导致busy-poll，2 减少事件的解除注册操作
- 适用场景/注意事项： LT/ET的使用场景

Busy loop问题

注销事件

```C++
typedef union epoll_data {
  void  *ptr//它指向用户自定义的数据存储空间，其由用户自定义；当事件触发时，就会有数据
  int   fd;
  uint32_t u32;
  uint64_t u64;
} epoll_data_t;

struct epoll_event {//内核/用户态信息交互的数据结构
  uint32_t   events;  /* Epoll events */
  epoll_data_t data;   /* User data variable */
};
```

3. epoll_wait 2 min

```C++
    int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
    int epoll_pwait(int epfd, struct epoll_event *events, int maxevents, int timeout, const sigset_t *sigmask);
```

在事件循环中工作，调用者进程被挂起，在等待内核 I/O 事件的分发

返回值:

成功返回的是一个大于0的数，表示事件的个数；返回0表示的是超时时间到；若出错返回-1.

epfd：

epoll 实例描述字，也就是 epoll 句柄。

events：

用来从内核得到事件的数组， 数组的每个元素都是一个需要待处理的 I/O 事件，其中有具体的事件类型

maxevents：

告知内核这个events有多大，这个 maxevents的值不能大于创建epoll_create()时的size;

timeout：

是超时时间，大于 0 的整数，表示 epoll_wait 可以返回的最大时间值（毫秒，corner case：0会立即返回，即使没有任何 I/O 事件发生，-1将不确定，一直阻塞到有时间到来，）。

ms，select支持微秒/poll的超时精度也是ms

4. close(epfd):

不用就close，epfd也基于引用计数的， 回收 epoll 实例所分配使用的内核资源

5. 示例： epoll怎么用，经典用法：epoll+非阻塞sockfd+线程池

C++

```
#define MAX_EVENTS 10
struct epoll_event ev, events[MAX_EVENTS];
int listen_sock, conn_sock, nfds, epollfd;

/* Code to set up listening socket, 'listen_sock',
 (socket(), bind(), listen()) omitted
*/

  //step1
epollfd = epoll_create1(0);
if (epollfd == -1) {
  perror("epoll_create1");
  exit(EXIT_FAILURE);
}

//step2
ev.events = EPOLLIN;
ev.data.fd = listen_sock;//
//注册事件
if (epoll_ctl(epollfd, EPOLL_CTL_ADD, listen_sock, &ev) == -1) {
  perror("epoll_ctl: listen_sock");
  exit(EXIT_FAILURE);
}

for (;;) {
//step3
 nfds = epoll_wait(epollfd, events, MAX_EVENTS, -1);
  if (nfds == -1) {
    perror("epoll_wait");
    exit(EXIT_FAILURE);
  }

  for (n = 0; n < nfds; ++n) {
    if (events[n].data.fd == listen_sock) {
      conn_sock = accept(listen_sock,
```

```
                    (struct sockaddr *) &addr, &addrlen);
            if (conn_sock == -1) {
              perror("accept");
              exit(EXIT_FAILURE);
            }
            //note
            setnonblocking(conn_sock);// 设置为non-blocking

            ev.events = EPOLLIN | EPOLLET;
            ev.data.fd = conn_sock;
            if (epoll_ctl(epollfd, EPOLL_CTL_ADD, conn_sock,
                  &ev) == -1) {
              perror("epoll_ctl: conn_sock");
              exit(EXIT_FAILURE);
            }
          } else {
            do_use_fd(events[n].data.fd);// do sth, dispatch,
          }
        }
      }
```

note：

一个非阻塞的socket上调用readn/writen函数, 返回EAGAIN/EWOULDBLOCK。

见名之意: errno

EAGAIN: 再试一次

EWOULDBLOCK: 如果这是一个阻塞socket, 操作将被block

6. 函数实现简介 5 min

了解：

fd是文件描述符,

在内核态, 与之对应的是struct file结构, 可以看作是内核态的文件描述符.

struct file：有成员target file wakeup list等待队列，等在上面的事件

epfd也是一个struct file

核心数据结构：

eventpoll结构体：

单链表ovflist，临时存放，存放传递数据给用户空间时，监听到了事件epitem

双链表readylist： 所有已经ready的epitem都在这个链表里面

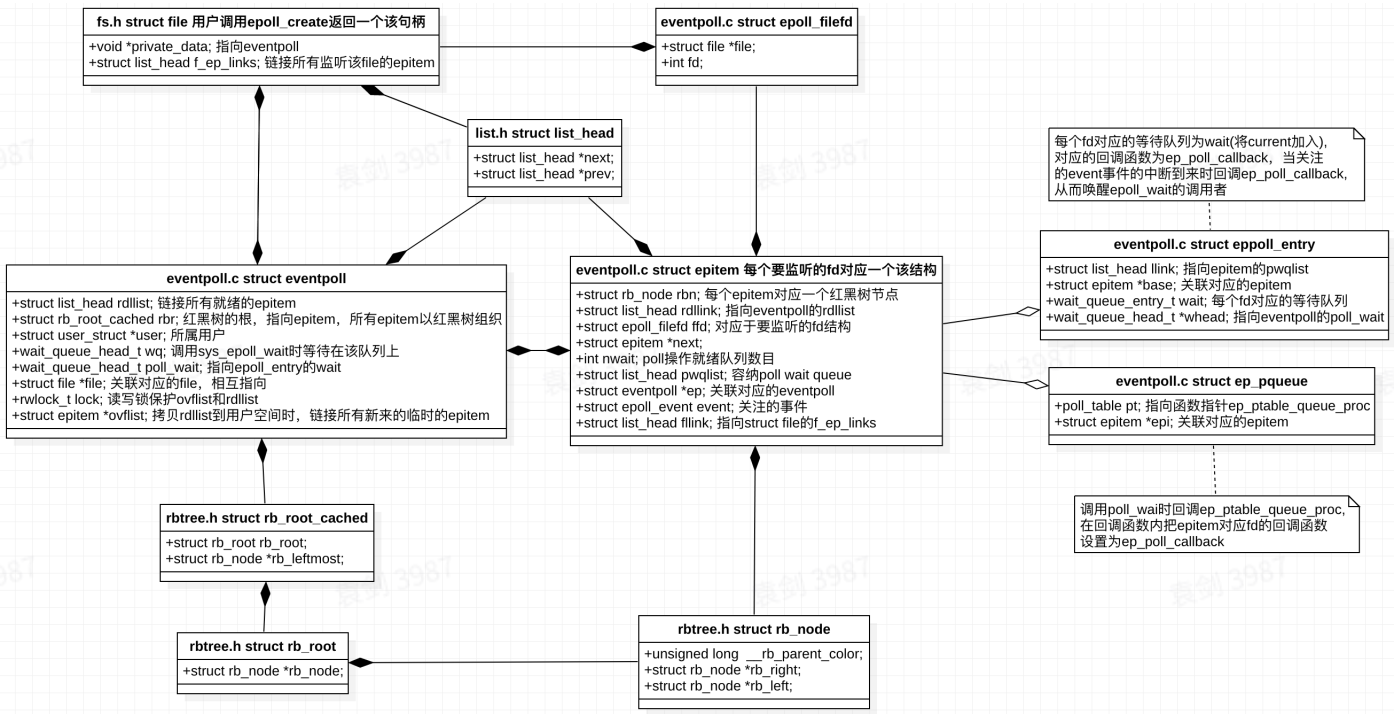
epitem的红黑树: rbt，epitem树的节点，所有要监听的epitem都在这里

C++

```c

/*
 * This structure is stored inside the "private_data" member of the file
 * structure and represents the main data structure for the eventpoll
 * interface.
 */
struct eventpoll {
    /* Protect the access to this structure */
    spinlock_t lock;

    /*
     * This mutex is used to ensure that files are not removed
     * while epoll is using them. This is held during the event
     * collection loop, the file cleanup path, the epoll file exit
     * code and the ctl operations.
     */
    struct mutex mtx;

    /* Wait queue used by sys_epoll_wait() */
    //这个队列里存放的是执行epoll_wait从而等待的进程队列
    wait_queue_head_t wq;

    /* Wait queue used by file->poll() */
    //这个队列里存放的是该eventloop作为poll对象的一个实例，加入到等待的队列
    //这是因为eventpoll本身也是一个file，所以也会有poll操作
    wait_queue_head_t poll_wait;

    /* List of ready file descriptors */
    //这里存放的是事件就绪的fd列表，链表的每个元素是下面的epitem
    struct list_head rdllist;

    /* RB tree root used to store monitored fd structs */
    //这是用来快速查找fd的红黑树
    struct rb_root_cached rbr;

    /*
     * This is a single linked list that chains all the "struct epitem" that
     * happened while transferring ready events to userspace w/out
     * holding ->lock.
     */
    struct epitem *ovflist;

    /* wakeup_source used when ep_scan_ready_list is running */
    struct wakeup_source *ws;

    /* The user that created the eventpoll descriptor */
    struct user_struct *user;

```

```
    //这是eventloop对应的匿名文件，充分体现了Linux下一切皆文件的思想
    struct file *file;

    /* used to optimize loop detection check */
    int visited;
    struct list_head visited_list_link;

#ifdef CONFIG_NET_RX_BUSY_POLL
    /* used to track busy poll napi_id */
    unsigned int napi_id;
#endif
};


/*
 * Each file descriptor added to the eventpoll interface will
 * have an entry of this type linked to the "rbr" RB tree.
 * Avoid increasing the size of this struct, there can be many thousands
 * of these on a server and we do not want this to take another cache line.
 */
struct epitem {
    union {
        /* RB tree node links this structure to the eventpoll RB tree */
        struct rb_node rbn;
        /* Used to free the struct epitem */
        struct rcu_head rcu;
    };

    /* List header used to link this structure to the eventpoll ready list */
    //将这个epitem连接到eventpoll 里面的rdllist的list指针
    struct list_head rdllink;

    /*
     * Works together "struct eventpoll"->ovflist in keeping the
     * single linked chain of items.
     */
    struct epitem *next;

    /* The file descriptor information this item refers to */
    //epoll监听的fd
    struct epoll_filefd ffd;

    /* Number of active wait queue attached to poll operations */
    //一个文件可以被多个epoll实例所监听，这里就记录了当前文件被监听的次数
    int nwait;

    /* List containing poll wait queues */
    struct list_head pwqlist;
```

```
 97
 98      /* The "container" of this item */
 99      //当前epollitem所属的eventpoll
100      struct eventpoll *ep;
101
102      /* List header used to link this item to the "struct file" items list */
103      struct list_head fllink;
104
105      /* wakeup_source used when EPOLLWAKEUP is set */
106      struct wakeup_source __rcu *ws;
107
108      /* The structure that describe the interested events and the source fd */
109      struct epoll_event event;
110  };
```



C++

```
1
2  /* Wait structure used by the poll hooks */
3  struct eppoll_entry {
4      /* List header used to link this structure to the "struct epitem" */
5      struct list_head llink;
6
7      /* The "base" pointer is set to the container "struct epitem" */
8      struct epitem *base;
9
10     /*
11      * Wait queue item that will be linked to the target file wait
12      * queue head.
13      */
14     wait_queue_entry_t wait;
15
16     /* The wait queue head that linked the "wait" wait queue item */
17     wait_queue_head_t *whead;
18 };
19
20 /* Wrapper struct used by poll queueing */
21 struct ep_pqueue {
22     poll_table pt;
23     struct epitem *epi;
24 };
25
26 /*
27  * Do not touch the structure directly, use the access functions
28  * poll_does_not_wait() and poll_requested_events() instead.
29  */
30 typedef struct poll_table_struct {
31     poll_queue_proc _qproc;
32     __poll_t _key;//
33 } poll_table;
```

eppoll_entry:

Wait structure used by the poll hooks

eppoll_entry主要epitem和epitem事件发生时的callback（ep_poll_callback）函数之间的关联，并将上述两个数据结构包装成一个链表节点，挂载到目标文件file的waithead队列中。

ep_pqueue:

主要把epitem和ep_ptable_queue_proc callback函数关联。然后通过目标文件的poll函数调用callback函数ep_ptable_queue_proc。

（poll函数一般由设备驱动提供，以网络设备为例，他的poll函数为sock_poll然后根据sock类型调用不同的poll函数如：packet_poll。packet_poll在通过datagram_poll调用sock_poll_wait，最后在poll_wait实际调用callback函数（ep_ptable_queue_proc））

核心函数：

epoll_create：

创建核心数据结构eventpoll并初始化，把一个未使用的fd作为epfd即index，和一个匿名文件关联，eventpoll地址放到该文件的privata_data指针上，返回epfd

epoll_ctl：

操纵核心数据结构eventpoll的接口，支持对rbt上节点i.e. fd interest set 的增/删/改

1. 准备/校验工作
2. 操纵rbt前需要lock mutex/
3. 调ep_insert()

ep_modify()/ep_remove(): EPOLL_CTL_MOD和EPOLL_CTL_DEL分别对应ep_modify和ep_remove，这两个函数就是从红黑树中去找到对应的节点进行修改和删除操作

ep_insert:

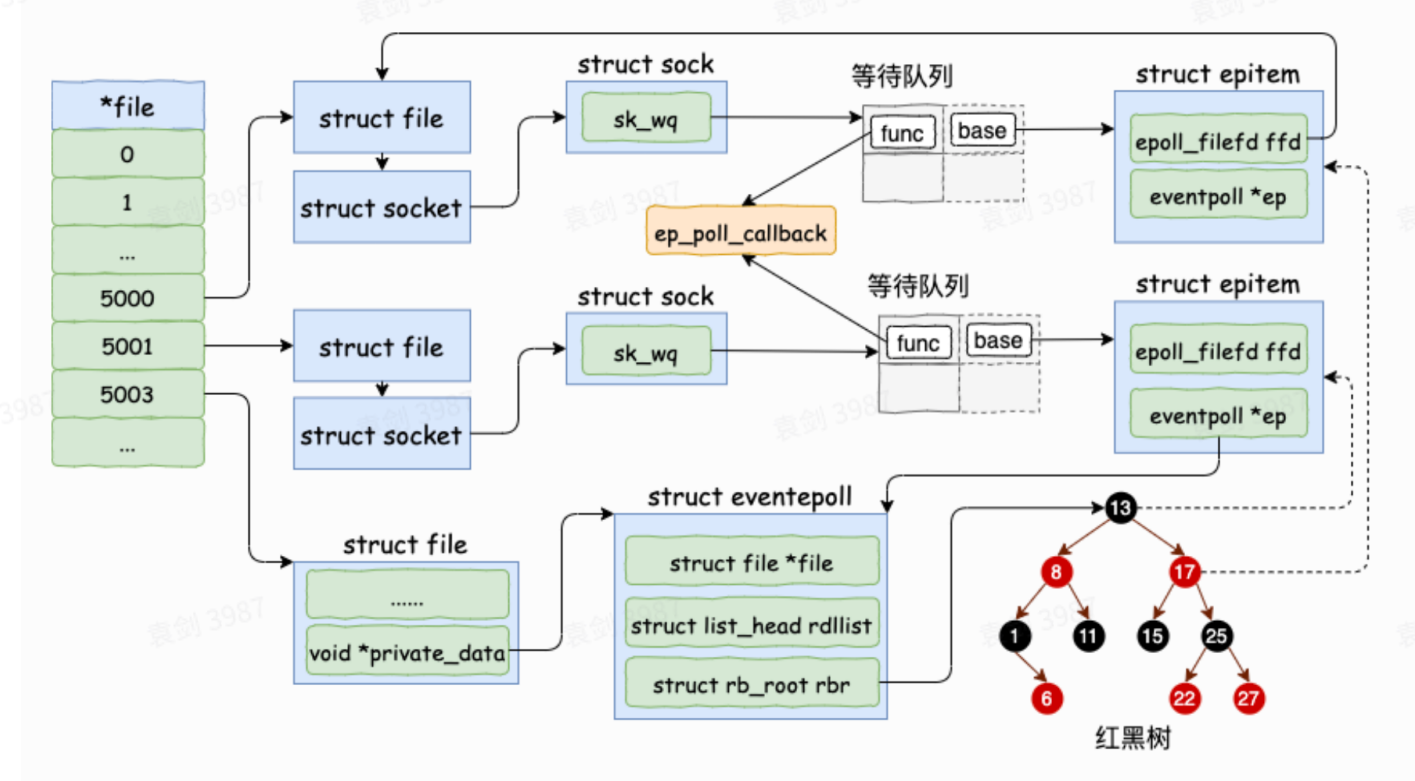
a. 用slab zalloc一个epitem节点，并init
b. 校验工作
c. 把节点插入到rbt
d. 把放到target fd对应的struct file里的 wakeup list链表里
e. 初始化ep_pqueue的poll_table
f. 调用fd对应设备驱动的poll_wait()方法，执行ep_ptable_queue_proc

init_poll_funcptr：设置epq的回调函数为ep_ptable_queue_proc，当调用poll_wait时会调用该回调函数；

ep_ptable_queue_proc: 在poll_wait()中，被同步的调用，设置fd对用关注事件的ep_poll_callback回调函数,当调用poll_wait时会调用该回调函数，

而函数体ep_ptable_queue_proc内部所做的主要工作, 就是把epitem对应fd的事件到来时的回调函数设置为ep_poll_callback

ep_poll_callback： 软中断发生时，被异步回调 , ep_poll_callback所做的主要工作就是把就绪的fd放到就绪链表rdllist上, 然后唤醒epoll_wait的调用者, 被唤醒的进程再把rdllist上就绪的fd的events拷贝给用户进程, 完成一个闭环。



epoll_wait相关

static int ep_poll(struct eventpoll *ep, struct epoll_event __user *events, int maxevents, long timeout): 1. 检查是否有ready的事件 2. 传递到用户空间

苏醒场景

1. 当前进程超时；
2. 当前进程收到一个 signal 信号；
3. 某个描述字上有事件发生；（ep_epoll_callback send a wake_up ）

醒来后调ep_send_events()

ep_send_events():

1. 调用readlist中每个文件描述符的 poll 方法，以便确定确实有事件发生。为什么这样做呢？这是为了确定注册的事件在这个时刻还是有效的，（有可能该fd上的事件，被其他线程的epfd处理）

（这也是为什么，当active的fd的比列比较高的时候，poll和epoll的性能相当。）

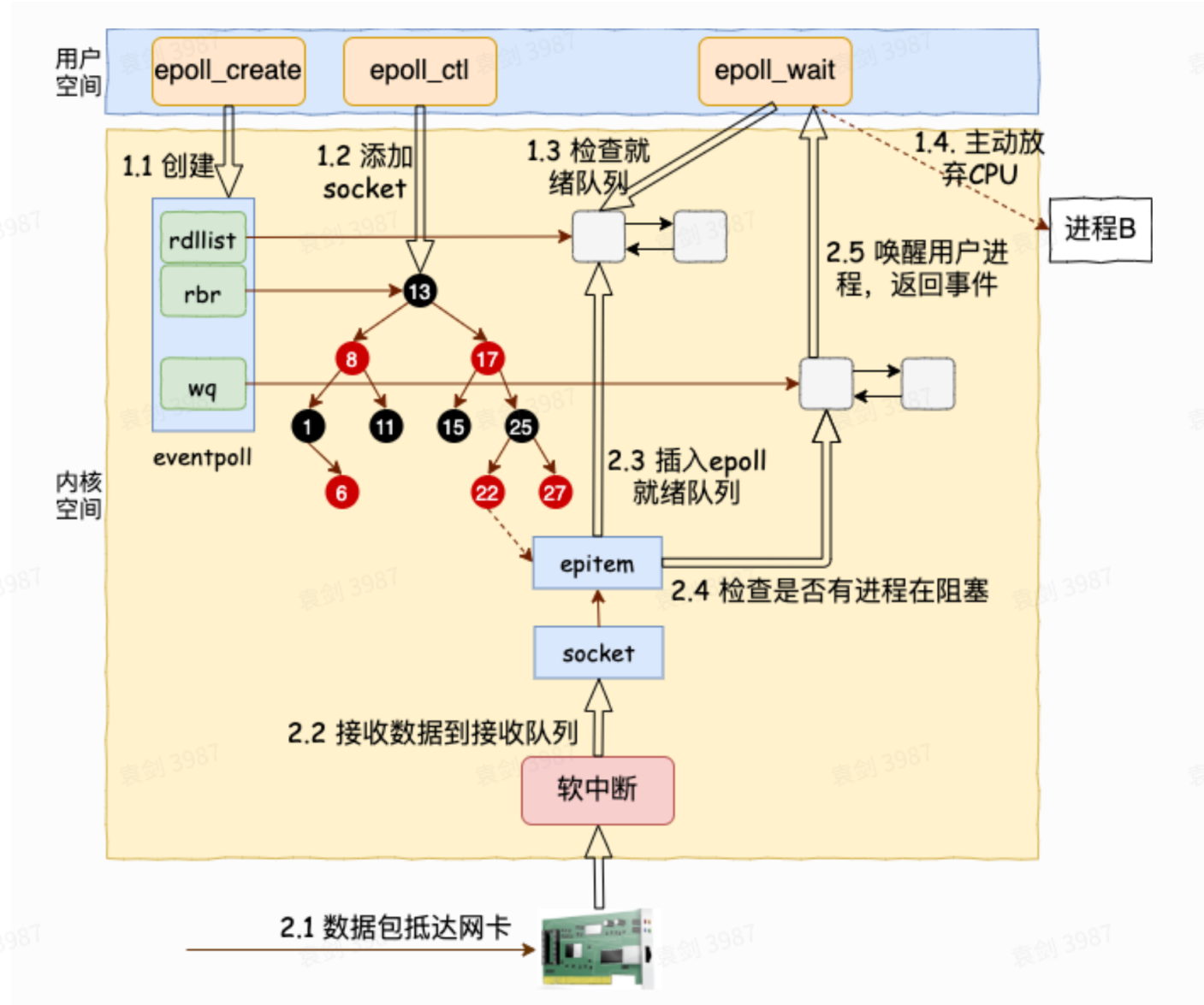
2. 针对 level-triggered 情况，当前的 epoll_item 对象被重新加到 eventpoll 的就绪列表readylist 中，为下一轮epoll_wait做准备；

   如step1所诉，在最终拷贝到用户空间有效事件列表中之前，会调用对应文件的 poll 方法，以确定这个事件是不是依然有效。所以，如果用户空间程序已经处理掉该事件，就不会被再次通知；如果没有处理，意味着该事件依然有效，就会被再次通知

3. 把真正ready事件，从内核空间拷贝到用户空间,

小结：epoll的整个工作流程

时序图(https://icoty.github.io/2019/06/03/epoll-source/)



三. epoll的其他相关事项 15 min

1. non-blocking+IO multiplexing的Context问题：stateful的应用层input/output buffer

2. epoll等事件误报问题：

   该fd事件已被其他线程处理

3. EPOLLONESHOT：

4. 事件处理模式/ one thread per loop/Reactor/sub-Reactor/Proactor:

   IO 多路复用”的事件分发分为两类：Reactor 和 Proactor，

分类标准：谁来做内核/用户态数据搬运，

one thread per loop/Reactor/event loop的核心是：

每个IO线程(事件分发线程）即对应一个事件循环，把IO事件分发到线程中执行注册的回调函数

proactor：IOCP

5. 关于 mmap, epoll 到底用没用到 mmap?

没有。也可以strace追踪一下

6. epollfd本身也是个fd

它本身也可以被epoll，不能epoll自己

7. 协程+同步==异步回调 pass

C++

```
 int32_t procTask()
{
    //do something
    return asyncGetScoreFromXXSvr(); //需要去其他Svr处理一些数据
}

int32_t getScoreFromXXSvrCallBack() //其他Svr处理完毕回调
{
    //continue do something
    return 0;
}
```

比较典型的异步非阻塞策略，前后业务逻辑写在不同地方。对于性能要求不太高的程序可以采用同步阻塞简化逻辑代码，或者使用协程达到“异步非阻塞”

JavaScript

```
int32_t procTask()
{
    //do something
    syncGetScoreFromXXSvr(); //同步处理数据，会造成阻塞
    //coGetScoreFromXXSvr(); //协程处理，假阻塞
    //continue do something
    return 0
}
```

当一个线程正阻塞在epoll_wait()上时，另一个线程修改次epoll_fd的事件关注列表(interest list)会发生什么？

Reference：

1. Linus_Torvalds/linux_kernel/epoll
2. https://mirrors.edge.kernel.org/pub/linux/kernel/
3. The C10K problem
4. https://libevent.org/
5. 深入理解epoll-InfoQ
6. epoll源码分析(基于linux-5.1.4)
7. epoll内核源码分析
8. epoll内核源码详解+自己总结的流程_技术交流_牛客网
9. 图解 | 深入揭秘 epoll 是如何实现 IO 多路复用的! - 文章详情
10. epoll源码深度剖析 - 坚持，每天进步一点点 - 博客园
11. Linux I/O复用中select poll epoll模型的介绍及其优缺点的比较_Chengzi_comm的专栏-CSDN博客
12. 不为人知的网络编程(十)：深入操作系统，从内核理解网络包的接收过程(Linux篇)
13. 云风的 BLOG: IOCP , kqueue , epoll ... 有多重要?