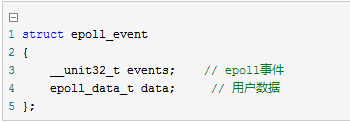
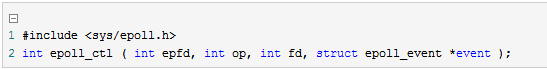
**文件描述符创建:**

****

**注册监控事件：**

****

函数说明：

     fd：要操作的文件描述符

     op：指定操作类型

操作类型：

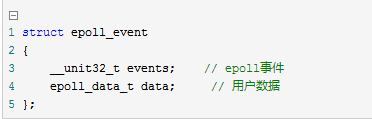
     EPOLL\_CTL\_ADD：往事件表中注册fd上的事件

     EPOLL\_CTL\_MOD：修改fd上的注册事件

     EPOLL\_CTL\_DEL：删除fd上的注册事件

     event：指定事件，它是epoll\_event结构指针类型

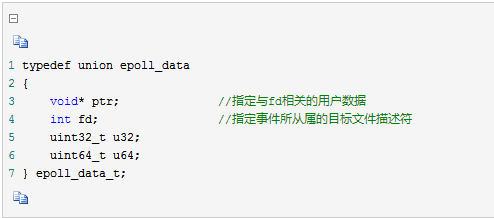
     epoll\_event定义：

****

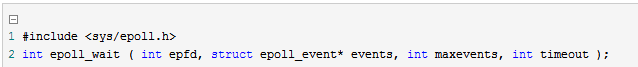
结构体说明：

     events：描述事件类型，和poll支持的事件类型基本相同（两个额外的事件：EPOLLET和EPOLLONESHOT，高效运作的关键）

     data成员：存储用户数据

****

**wait函数：**

****

函数说明：

     返回：成功时返回就绪的文件描述符的个数，失败时返回-1并设置errno

     timeout：指定epoll的超时时间，单位是毫秒。当timeout为-1是，epoll\_wait调用将永远阻塞，直到某个时间发生。当timeout为0时，epoll\_wait调用将立即返回。

     maxevents：指定最多监听多少个事件

     events：检测到事件，将所有就绪的事件从内核事件表中复制到它的第二个参数events指向的数组中。

**1.select的缺陷（select适用于活跃度较多的场景，epoll适用于活跃度较少的场景）**

高并发的核心解决方案是1个线程处理所有连接的“等待消息准备好”，这一点上epoll和select是无争议的。但select预估错误了一件事，当数十万并发连接存在时，可能每一毫秒只有数百个活跃的连接，同时其余数十万连接在这一毫秒是非活跃的。select的使用方法是这样的：

      返回的活跃连接 ==select（全部待监控的连接）。

      什么时候会调用select方法呢？在你认为需要找出有报文到达的活跃连接时，就应该调用。所以，调用select在高并发时是会被频繁调用的。这样，这个频繁调用的方法就很有必要看看它是否有效率，因为，它的轻微效率损失都会被“频繁”二字所放大。它有效率损失吗？显而易见，全部待监控连接是数以十万计的，返回的只是数百个活跃连接，这本身就是无效率的表现。被放大后就会发现，处理并发上万个连接时，select就完全力不从心了。

      此外，在Linux内核中，select所用到的FD\_SET是有限的，即内核中有个参数\_\_FD\_SETSIZE定义了每个FD\_SET的句柄个数。

其次，内核中实现 select是用**轮询**方法，即每次检测都会遍历所有FD\_SET中的句柄，显然，select函数执行时间与FD\_SET中的句柄个数有一个比例关系，即 select要检测的句柄数越多就会越费时。看到这里，您可能要要问了，你为什么不提poll？笔者认为select与poll在内部机制方面并没有太大的差异。相比于select机制，poll只是取消了最大监控文件描述符数限制，并没有从根本上解决select存在的问题。

**2.epoll的原理**

epoll精巧的使用了3个方法来实现select方法要做的事：

1.新建epoll描述符==epoll\_create()

2.epoll\_ctrl(epoll描述符，添加或者删除所有待监控的连接)

3.返回的活跃连接 ==epoll\_wait（ epoll描述符 ）

与select相比，epoll分清了频繁调用和不频繁调用的操作。例如，epoll\_ctrl是不太频繁调用的，而epoll\_wait是非常频繁调用的。

epoll的三大关键要素：**mmap、红黑树、链表**。epoll是通过内核与用户空间mmap同一块内存实现的。减少用户态和内核态之间的数据交换。内核可以直接看到epoll监听的句柄，效率高。

epoll\_wait的工作流程：

1.epoll\_wait调用ep\_poll，当rdlist为空（无就绪fd）时挂起当前进程，直到rdlist不空时进程才被唤醒。

2.文件fd状态改变（buffer由不可读变为可读或由不可写变为可写），导致相应fd上的回调函数ep\_poll\_callback()被调用。

3.ep\_poll\_callback将相应fd对应epitem加入rdlist，导致rdlist不空，进程被唤醒，epoll\_wait得以继续执行。

4.ep\_events\_transfer函数将rdlist中的epitem拷贝到txlist中，并将rdlist清空。

5.ep\_send\_events函数（很关键），它扫描txlist中的每个epitem，调用其关联fd对用的poll方法。此时对poll的调用仅仅是取得fd上较新的events（防止之前events被更新），之后将取得的events和相应的fd发送到用户空间（封装在struct epoll\_event，从epoll\_wait返回）。

epoll关于LT和ET小结：

       LT：水平触发，效率会低于ET触发，尤其在大并发，大流量的情况下。但是LT对代码编写要求比较低，不容易出现问题。LT模式服务编写上的表现是：只要有数据没有被获取，内核就不断通知你，因此不用担心事件丢失的情况。

       ET：边缘触发，效率非常高，在并发，大流量的情况下，会比LT少很多epoll的系统调用，因此效率高。但是对编程要求高，需要细致的处理每个请求，否则容易发生丢失事件的情况。

      从本质上讲：与LT相比，ET模型是通过减少系统调用来达到提高并行效率的。