浅谈IO的多路复用技术之一(select和epoll实质)

2016-01-29 23:39 feiying 1 阅读 288

JAVA的NIO技术从1.5开始，一直到现在的JDK8，这套JDK自带的API几乎填充了了整个java端服务器的代码实现，人们都是大谈特谈这些接口，但是很少有人深究操作系统实现的底层细节，这篇文章带你简单浏览一下这些底层的细节。

JDK 1.5 中NIO出来后，搞出了几个类，Selector，Channel，Buffer，关心的事件如read/write等这些内容，实质这些类是java部分的再次封装，早在很早之前，基于操作系统的select接口就已经存在.我们可以在linux的命令行的环境中 man select一下，看看select接口的系统调用：

```
/* According to POSIX.1-2001 */
#include <sys/select.h>

/* According to earlier standards */
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

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

基于POSIX-2001的接口，需要引入4个.h文件，select系统调用的参数一共有5个：

参数1：你所监视的系统描述符fd最大的，然后+1，====》比较奇怪

参数2：fd读的状态有通知了，回填到这个读的集合中

参数3：fd写集合传入，也回填到这个参数

参数4：异常集合传入，也回填到这个参数

参数5：超时设置，如果没有这个参数，2,3,4参数啥都没发生，select就一直阻塞，

通过这些参数，我们就可以了解，select的系统调用级别的代码几乎和java的NIO的类库很像（或者说java类库中的就是按照select来实现）

select这个系统调用是很古老的，一般的Unix的衍生操作系统都支持，移植行也是非常的好，并且基于事件进行组织；

但是，通过前面你查看参数就可以总结出来，其缺陷也是多多：

缺陷1：参数1非常的奇怪，最大的fd还要+1，经常有人会没有加1，而导致系 统调用失败，对于此，只能怨Unix设计者的古老了，没有招；

缺陷2：2,3,4参数，每一次是我设置的需要监视的参数，需要传入到这个select中，但是在恰恰这个传入的参数，如果有事件发生的话，也是回填到这个参数。==》这相当于什么？相当于你好不容易传入的东西，都被冲掉了，因此，你每一次select完事之后，这些fd你还得重新设置一遍，==》可以总结接口非常的难用，而java的NIO接口在这里也延续了这一习惯。

缺陷3：void FD_CLR(int fd, fd_set *set);===》对于fd描述符的操作集合的方法很不好用，这个极容易引起混淆，注意这个系统调用不是清空，而是删除，名字容易糊涂.

缺陷4：监视的fd，仅仅就是读，写，异常，监视事件范围太单一，不利于查找；

看到这些缺陷，可以发现，JAVA的NIO和这个非常的类似，说的没错，最早期的NIO的底层实现就是select，至少是JDK1.5，和JDK1.6中都是。

在JDK1.6的后期的版本中，需要打开-D参数：

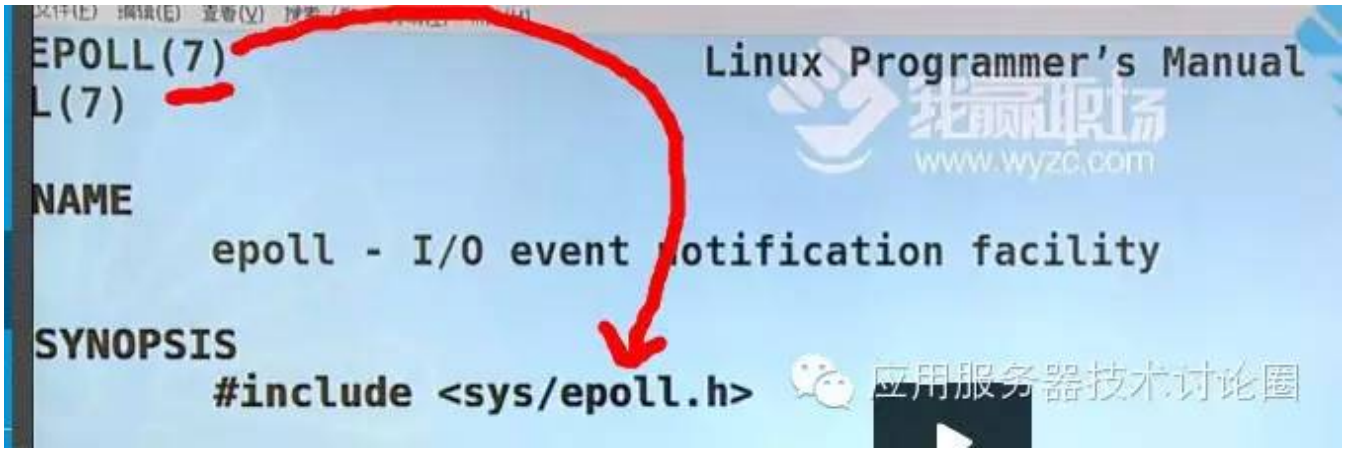
-Djava.nio.channels.spi.SelectorProvider=sun.nio.ch.EPollSelectorProvider

这个参数，就指示JAVA 的NIO框架默认就采用的epoll作为底层支撑。

到了JDK1.7的时候，默认就是epoll，select已经完全退出历史舞台了。

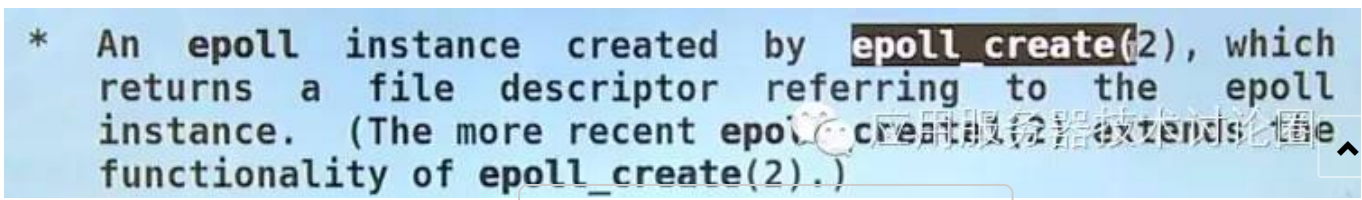
为什么这里提到了epoll？epoll有啥优点呢？

其实epoll也就是一个系统调用，你在linux中man epoll一下，它仍会告诉你epoll是什么东西：



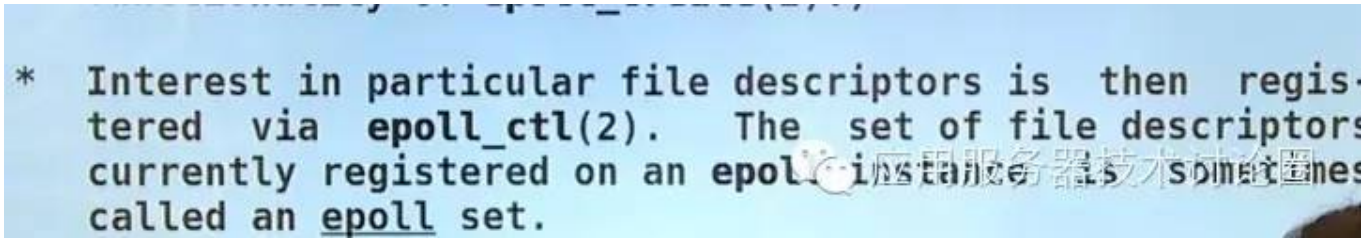
可以看到，根本就不是man 2，因为epoll 函数就是一个方言

第一步：



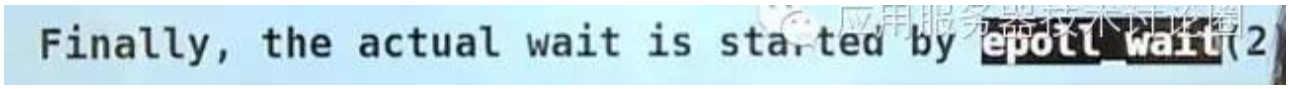
通过epoll_create进行创建epoll 实例

第二步：



通过epoll_ctl对fd进行注册感兴趣的事件

第三步：



通过epoll_wait来等待，来查看监视结果

上面的三个步骤貌似还挺麻烦，但你要仔细分析一下，你就知道为什么epoll好的原因了；

其中一个重要的系统调用就是通过epoll_ctl函数注册感兴趣的时间，而这个epoll_ctl函数：

参数1：刚才epoll_create的系统调用的返回的内容，也就是epoll的实例

参数2：op操作

EPOLL_CTL_ADD

Register the target file descriptor fd on the epoll instance referred to by the file descriptor epfd and associate the event event with the internal file linked to fd.

EPOLL_CTL_MOD

Change the event event associated with the target file descriptor fd.

EPOLL_CTL_DEL

Remove (deregister) the target file descriptor fd from the epoll instance referred to by epfd. The event is ignored and can be NULL (but see BUGS below).

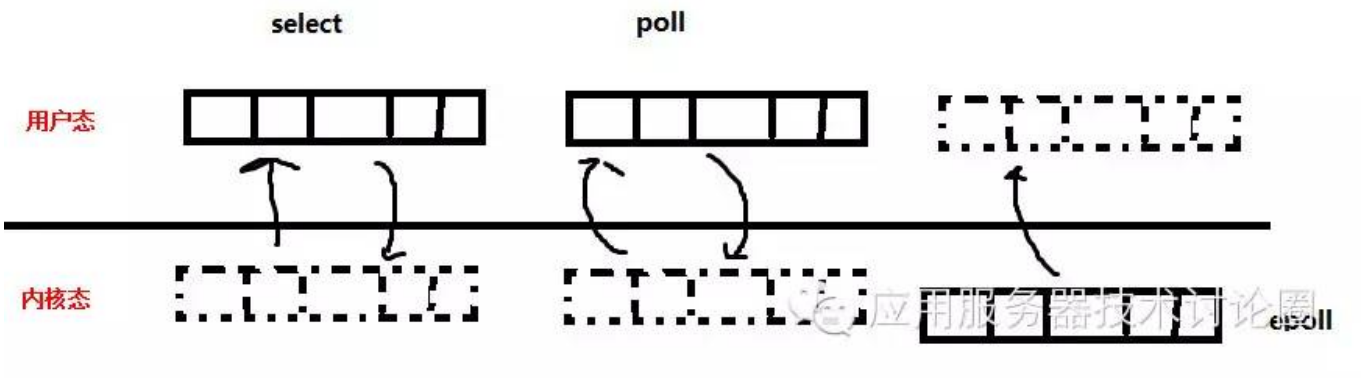
参数3：针对的对象是文件描述符

参数4：针对参数3的fd的哪个事件

=========》分析到这里，我们可以发现，在epoll中貌似操作的fd事件集合是开放一个系统调用供客户端进行调用的，而不是类似select中我们自己可以攒1个fd集合，但是在epoll这里不行，我们只能以调用系统调用函数的方式，操纵这个fd。

而这种架构，就如下图所示，这也就表明了，为啥epoll优异的原因：

1.关于fd事件数组的复制



上图是对比了三个系统调用，你可以理解poll和select差不多，实线上是用户态，实线下是内核态。

可以看到select和poll的fd_set集合，是在用户态进行定义，然后你通过系统调用，将这个参数传入到内核态中，这是一次复制，这个数据结构就在内核态也被复制一份（虚线部分）；

而select和poll的系统调用结束，发现有一些fd有事件来了，再将这个数据结构，从内核态传回用户态，然后用户再进行遍历；

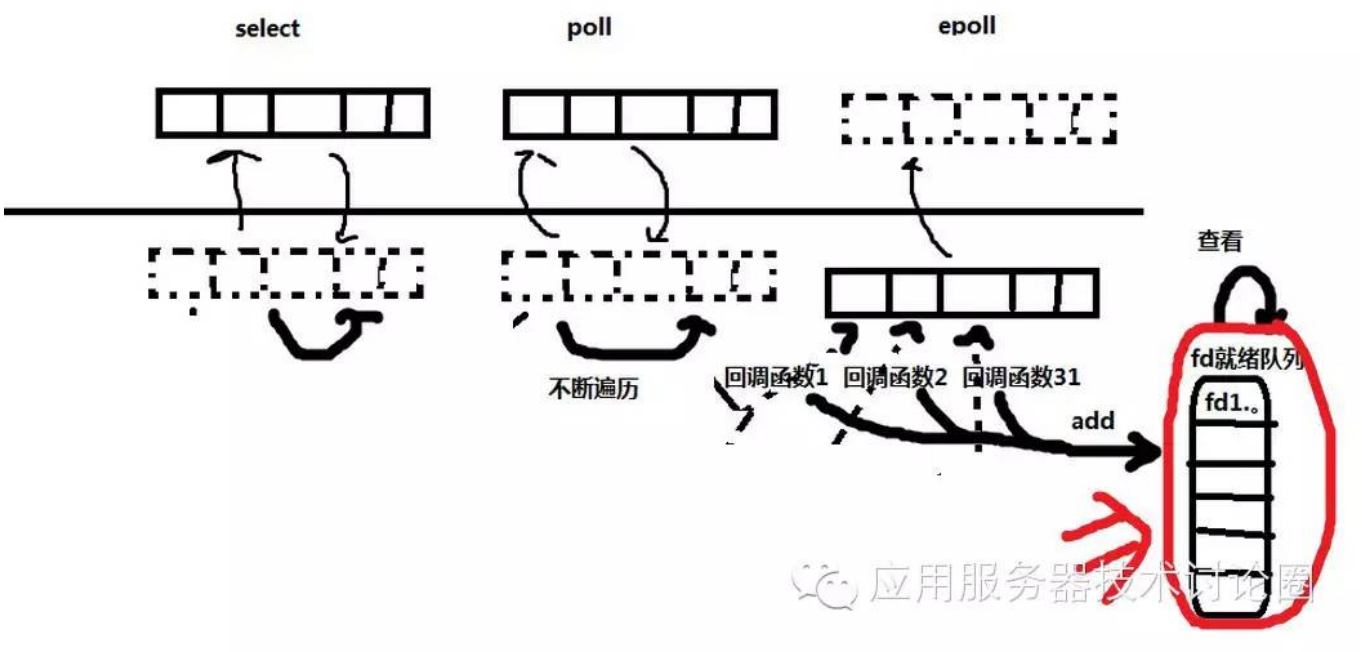
里外里，这就是两次fd数组的复制，我们要是有10000个fd关注，可以看到，每一次select和poll都来回折腾一遍，消耗太大！

===》epoll改进在于通过epoll_create系统调用，直接在内核态创建fd数组，没有复制

epoll系统调用结束，发现有一些fd事件来了，将内核态传入用户态，这有一次复制；

总结，一次系统调用查找到有事件的fd，select，poll两次来回在用户态和内核态复制，而epoll只有1次，这个就是第一个优点。

2.fd事件数组的遍历



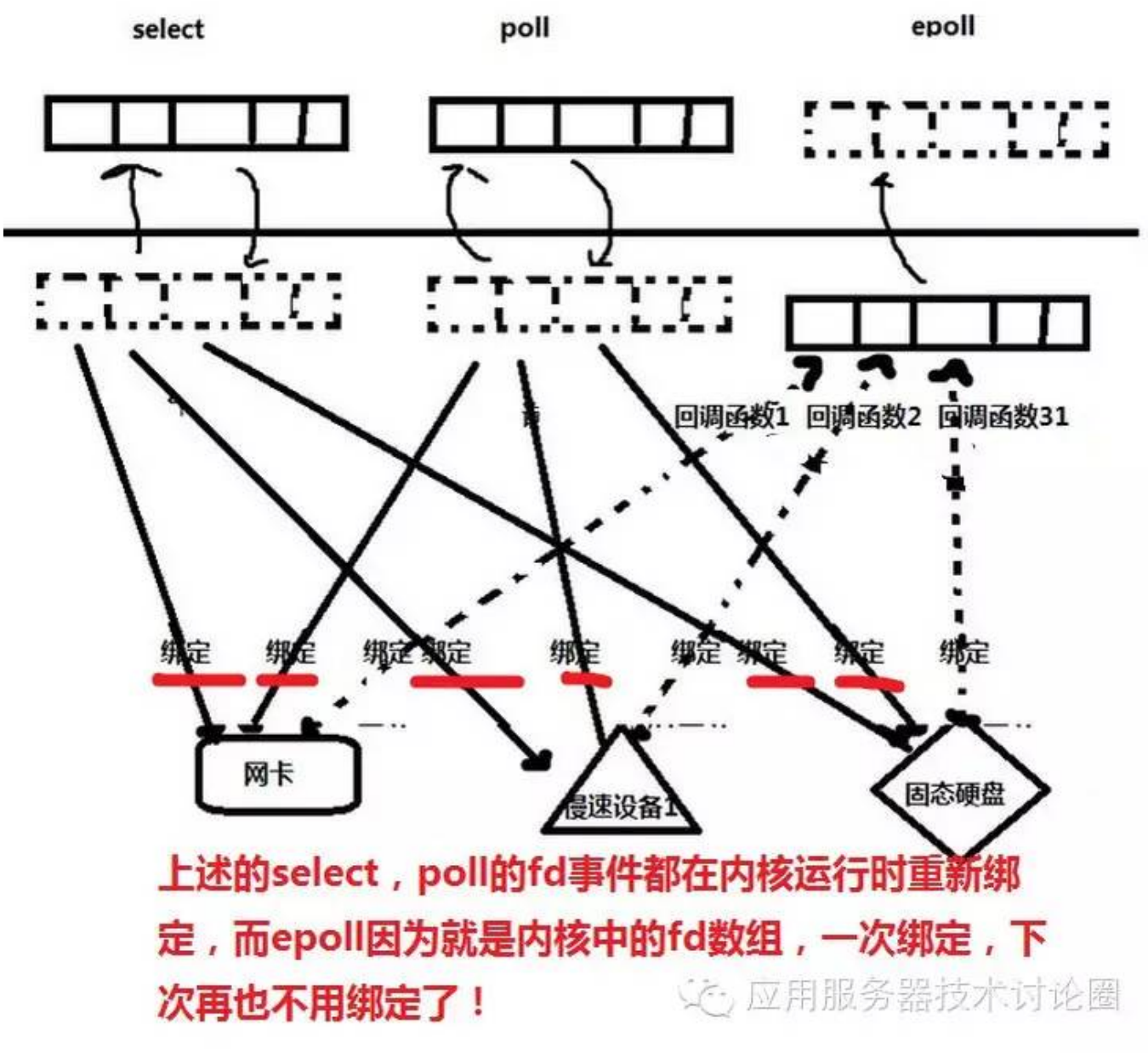
select和poll在内核中也对应fd_set数组，可以看到这是从用户态拷贝到内核中的，而epoll的fd_set数组是内核中的数据结构，这是我们已知的二者的大不同；正因为如此，select和poll的fd_set数组，就是普通的数据，没有任何的附加功能，因此IO多路复用，硬件事件发生后（也称就绪状态），会直接赋值到这个fd_set数组中；**而select和poll在每一次阻塞-唤醒，这一过程中，至少有1到n次的select的轮询工作**；===》select和poll需要遍历，**epoll同样也得遍历，但是epoll的机制在于遍历的内容少的吓人**，epoll中内核所谓的fd_set集合，并不是遍历的对象，他其中每一个fd都对应回调函数，当就绪事件发生后，将这个真正有事件的fd连同事件，一块放到一个**epollfd就绪队列**中。

可以看到，每一次epoll遍历仅仅是这个fd就绪队列，这个队列中的fd全部都是就绪的，甚至可以这么说，epoll就压根没有遍历，只需判断一下fd就绪队列是否为空，不为空就返回，因此效率惊人，同比100w个fd做监视，对于那种网卡类的稀疏网络事件的情况（也就是大部分时间,甚至99%以上的时间都没事干，没流量），select和poll一般至少要遍历100w次或者200w次，甚至设置超时时间的话，在等待超时时间这段cpu就爆满了；

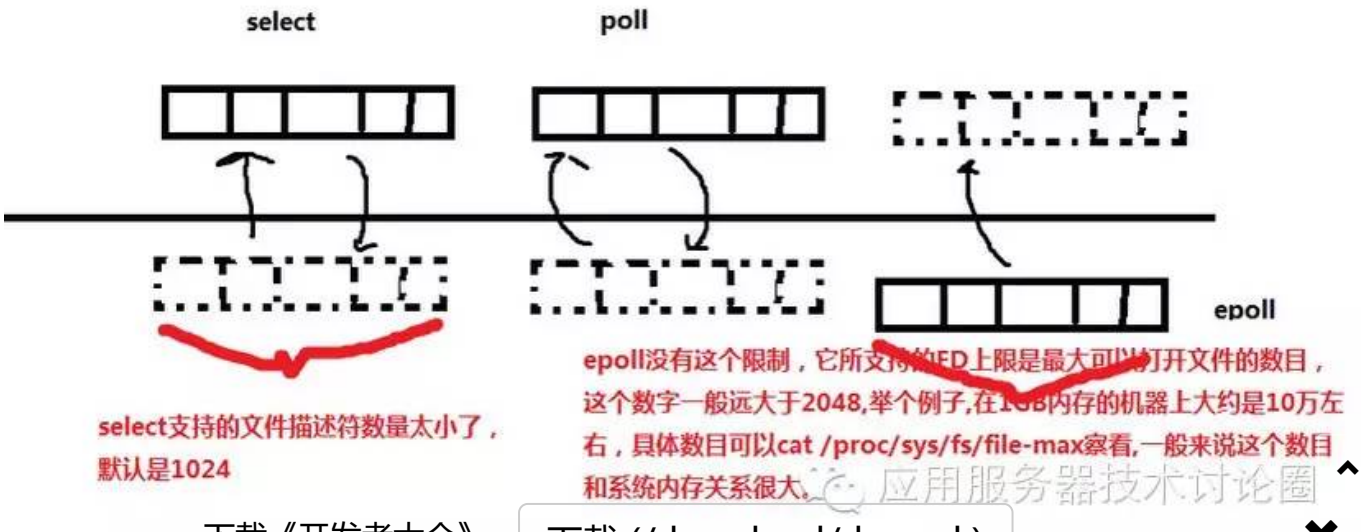
==》但是epoll仅仅遍历1次？2次？最坏的等待超时也仅仅是n次，数量级差太多了，这也就是epoll的优势，总结一下，也就是epoll单独搞了一个fd就绪队列的模式，减少了遍历！

3.**从内核角度来看，基于fd事件数组在内核态都需要与硬件驱动进行绑定，绑定是很耗时的，epoll的fd事件数组，就在内核中，绑定1次就OK，**

而select这些用户态的数组，执行到内核态中，每一次都需要重新绑定一次：



4.**fd事件的限制：**



总结了上面的4条，其实epoll性能优异可以归结于一句话，就是epoll的事件fd放在了内核中，不在用户态折腾了，直接更底层的进行操作，省去了不少的事情，这个是实质的原因！

java中的NIO在JDK后续的版本中，在linux的环境下，基本都是epoll了，当然类似于epoll的机制，Solaris中有eventq，FreeBSD中的kqueue也相当的猛，

这些都是IO多路复用技术，它们的本质并不是AIO，所谓的AIO至少到目前位置，没有什么好的系统调用实现，虽然有AIO的接口，但基于硬件平台的不同，效果差强人意。而IO多路复用技术，是通过一个按照时钟周期轮询的装置，基于事件去你注册的事件集合，有事件的话直接返回，没有事件的话如果没有超时时间的话，就阻塞，从这一点来看，貌似是异步的过程，但是这个过程和纯AIO还不一样，纯的AIO接口根本不需要什么Selector，epoll实例这些装置，还有上述的各种fd集合的扫描，绑定，遍历，和用户态到内核态的赋值和迁移，直接就是事件驱动，一个注册事件对应一个内核级别的绑定，上述的fd集合这些费劲的东西根本都不需要。不过随着时代的发展，基于硬件的AIO接口现在很多项目已经也在用，效率也是惊人的高的。

java的NIO这块目前底层技术还是IO多路复用为主，linux中epoll是主要解决方案。

明天我们会聊聊，NIO的一些bug，比较常见的空转的那个bug，看看一些服务器是咋解决这个问题的。

后天我们会继续这一个话题，看看java的AIO的接口，聊聊最新的进展。

敬请大家期待。

分享：

玩转Netty – 从Netty3升级到Netty4 (/html/308/201504/206233287/1.html)

金蝶中间件2015招聘来吧！Come on！ (/html/308/201505/206307460/1.html)

GlassFish 4.1 发布，J2EE 应用服务器 (/html/308/201505/206323120/1.html)

Tomcat对keep-alive的实现逻辑 (/html/308/201505/206357679/1.html)

猜您喜欢

等价类划分 (/html/196/201606/2247483766/1.html)

Golang在视频直播平台的高性能实践（含PPT下载） (/html/175/201603/404230356/1.html)

如何避免Java中的“!=null”？ (/html/472/201605/2651716271/1.html)

逗逼程序猿养成记（一） (/html/447/201608/2649313574/1.html)

大数据告诉你，什么样的家庭能培养出高考“状元”！ (/html/318/201507/207599285/1.html)