应用服务器中对JDK的epoll空转bug的处理

2016-02-01 11:00 应用服务器技术讨论圈 135 阅读 4132

前面讲到了epoll的一些机制，与select和poll等传统古老的IO多路复用机制的一些区别，这些区别实质可以总结为一句话，

就是epoll将重要的基于事件的fd集合放在了内核中来完成，因为内核是高效的，所以很多关于fd事件监听集合的操作也是高效的，

不方便的就是，因为在内核中，所以我们需要通过系统调用来调用关于fd操作集合，而不是直接自己攒一个。

如果在linux中，epoll在JDK6中还需要配置，在后续的版本中为JDK的NIO提供了默认的实现，但是epoll在JDK中的实现却是漏洞百出的，

bug非常的多，比较容易复现并且被众多人诟病的就是epoll轮询的处理方法。

sun的bug列表为：

http://bugs.java.com/bugdatabase/view_bug.do?bug_id=6670302

JDK-6670302 : (se) NIO selector wakes up with 0 selected keys infinitely [lnx 2.4]

===》这个bug的描述内容为，在NIO的selector中，即使是关注的select轮询事件的key为0的话，NIO照样不断的从select本应该阻塞的

情况中wake up出来，也就是下图中的红色阻塞的部分：


```
ServerSocketChannel serverChannel = ServerSocketChannel.open();
serverChannel.configureBlocking(false);
ServerSocket server = serverChannel.socket();
server.bind(new InetSocketAddress(port));

Selector selector = Selector.open();
serverChannel.register(selector, SelectionKey.OP_ACCEPT);
SocketChannel clientChannel = null;

System.out.println("0. SERVER STARTED TO LISTEN");
boolean writeNow = false;

while (true) {
    try {
        // wait for selection
        int numKeys = selector.select();

        if (numKeys == 0) {
            System.err.println("select wakes up with zero!!!");
        }

        Iterator it = selector.selectedKeys().iterator();
        while (it.hasNext()) {
            SelectionKey selected = (SelectionKey) it.next();
            int ops = selected.interestOps();

            try {
                // process new connection
                if ((ops & SelectionKey.OP_ACCEPT) != 0) {
                    clientChannel = serverChannel.accept();
                    clientChannel.configureBlocking(false);

                    // register channel to selector
                    clientChannel.register(selector, SelectionKey.OP_READ, null);
                    System.out.println("2. SERVER ACCEPTED AND REGISTER READ OP :
client - " + clientChannel.socket().getInetAddress());
                }

                if ((ops & SelectionKey.OP_READ) != 0) {
                    // read client message
                    System.out.println("3. SERVER READ DATA FROM client - " +
clientChannel.socket().getInetAddress());
                    readClient((SocketChannel) selected.channel(), buffer);

                    // deregister OP_READ
                    System.out.println("PREV SET : " + selected.interestOps());
                    selected.interestOps(selected.interestOps() &
~SelectionKey.OP_READ);
                    System.out.println("NEW SET : " + selected.interestOps());

                    Thread.sleep(SLEEP_P... * 2);
                    new WriterThread(clientChannel).start();
```


然后，因为selector的select方法，返回numKeys是0，所以下面本应该对key值进行遍历的事件处理根本执行不了，

又回到最上面的while(true)循环，循环往复，不断的轮询，直到linux系统出现100%的CPU情况，其它执行任务干不了活，

最终导致程序崩溃。

==》从这个bug上来看，这个绝对是JDK中的问题，select方法就应该是阻塞的，没有key事件过来，那么就不应该返回，

和应用程序的写法没有任何的关系，与之相差不多的一个bug给出了解决的方案：

http://bugs.java.com/bugdatabase/view_bug.do?bug_id=6403933

JDK-6403933 : (se) Selector doesn't block on Selector.select(timeout) (lnx)

这个bug的意思基本上和前面的**JDK-6670302**相差不大，也是Selector不阻塞，前一个bug说明的是最终的现象，

这个**JDK-6403933**的bug说出了实质的原因：

```
EVALUATION

--

This is an issue with poll (and epoll) on Linux. If a file descriptor for a connected socket is
polled with a request event mask of 0, and if the connection is abruptly terminated (RST) then
the poll wakes up with the POLLHUP (and maybe POLLERR) bit set in the returned event set. The
implication of this behaviour is that Selector will wake up and as the interest set for the
SocketChannel is 0 it means there aren't any selected events and the select method returns 0.
```

具体解释为，**在部分Linux的2.6的kernel中，poll和epoll对于突然中断的连接socket会对返回的eventSet事件集合置为POLLHUP，**

也可能是POLLERR，eventSet事件集合发生了变化，这就可能导致Selector会被唤醒。==》这是与操作系统机制有关系的，JDK虽然仅仅

是一个兼容各个操作系统平台的软件，但很遗憾在JDK5和JDK6最初的版本中（严格意义上来将，JDK部分版本都是），这个问题并没有解决，而将这个帽子抛给了操作系统方，这也就是这个bug最终一直到2013年才最终修复的原因，最终影响力太广。

修复的方法，在这个bug中已经提到了：

```
WORK AROUND

--

The workaround to this issue is to cancel the key and flush it from the Selector (by invoking
the selectNow method).
```

上面是第一个建议，首先将SelectKey去除掉，然后“刷新”一下Selector，刷新的方式也就是调用Selector.selectNow方法，

这个示意的代码如下：

```
if (SelectionKey != null)  {  // the key you registered on the temporary selector
   SelectionKey.cancel();   // cancel the SelectionKey that was registered with the temporary
selector
   // flush the cancelled key
   temporarySelector.selectNow();
}
```

这段代码意味着重置，首先将SelectionKey注销掉，然后重新调用非阻塞的selectNow来让Selector换取“新生”。

这种修改方式就是grizzly的commiteer们最先进行修改的，并且通过众多的测试说明这种修改方式大大降低了JDK NIO的问题。

但是，这种修改 … 一共有两点

1.多个线程中的SelectionKey的key的cancel，很可能和下面的Selector.selectNow同时并发，如果是导致key的cancel后运行很可能没有效果

2.与其说第一点使得NIO空转出现的几率大大降低，经过Jetty服务器的测试报告发现，这种重复利用Selector并清空SelectionKey的改法很可能没有任何的效果，

最终的终极办法是创建一个新的Selector：

具体的Jetty服务器的分析地址为：

https://wiki.eclipse.org/Jetty/Feature/JVM_NIO_Bug

Jetty首先定义两了-D参数：

- org.mortbay.io.nio.JVMBUG_THRESHHOLD, defaults to 512 and is the number of zero select returns that must be exceeded in a period.
- org.mortbay.io.nio.MONITOR_PERIOD defaults to 1000 and is the period over which the threshhold applies.

第一个参数是select返回值为0的计数，第二个是多长时间，整体意思就是控制在多长时间内，如果Selector.select不断返回0，说明进入了JVM的bug的模式

那么，Jetty这时候就有所作为了，我们看到Jetty的具体的代码如下：

首先，根据-D参数判断是否进入了JAVA NIO空转的bug模式，一个是判断时间，一个是判断次数，次数通过-jvmBug作为

计数器进行统计；如果一旦确定是bug，可以看到上述代码为了防止并发出现，加了Sychronized锁，接着开启一个新的

Selector，并将原有的SelectionKey的事件全部转移到了新的Selector中，最后将-jvmBug计数器置0；

==》这种处理方法要保险的多，基本上不会有任何的问题了，

Jetty在这个网页中还提供了很多参数，如：

Injecting Delay

In some circumstances, even a newly created select set quickly suffers from the same problems. If Jetty detects that the JVMBUG_THREASHOLD has been exceeded, it reacts by inserting pauses in the selecting thread. The `org.mortbay.io.nio.BUSY_PAUSE` system parameter controls the duration of the pause; it defaults to 50ms.

The pause allows the acceptor thread to stop calling select in a busy loop, and allows the dispatched threads to proceed with handling any selected connections. At 50ms, the worst case is that this delay adds 25ms latency to a request. However, in practise these problems only occur on busy servers with jobs in excess of the available CPUs, so this 25ms is probably not much in excess of an expected scheduling delay.

即使上述的处理方式，对应极少的linux环境和JDK的版本，仍会出现一些问题，这主要是因为网络中断的间隔时间太短造成的，需要给内核一定的时钟周期

进行缓冲，而上述的Jetty的org.mortbay.io.nio.BUSY_PAUSE这个参数就是起到间隔的作用，间隔多少微秒再调用Select，这样基本上能最大程度上避免上述

问题出现了。

从上面Jetty各种处理方法来看，基本能屏蔽低版本JDK和操作系统的epoll的影响，让NIO可以无忧运行。当然，对于NIO框架也是修正了这些错误，

前面提到的Griizzly和Netty都对这个问题采取了响应的策略。

以Netty为例，具体位置在NioSelector的实现类AbsNioSelector中：

```
for (;;) {
    wakenUp.set(false);

    try {
        long beforeSelect = System.nanoTime();
        int selected = select(selector);
        if (selected == 0 && !wakenupFromLoop && !wakenUp.get()) {
            long timeBlocked = System.nanoTime() - beforeSelect;
            if (timeBlocked < minSelectTimeout) {
                boolean notConnected = false;
                // loop over all keys as the selector may was unblocked because of a closed channel
                for (SelectionKey key: selector.keys()) {
                    SelectableChannel ch = key.channel();
                    try {
                        if (ch instanceof DatagramChannel && !ch.isOpen() ||
                                ch instanceof SocketChannel && !((SocketChannel) ch).isConnected() &&
                                        // Only cancel if the connection is not pending
                                        // See https://github.com/netty/netty/issues/2931
                                        !((SocketChannel) ch).isConnectionPending()) {
                            notConnected = true;
                            // cancel the key just to be on the safe side
                            key.cancel();
                        }
                    } catch (CancelledKeyException e) {
                        // ignore
                    }
                }
                if (notConnected) {
                    selectReturnsImmediately = 0;
                } else {
                    if (Thread.interrupted() && !shutdown) {
                        // Thread was interrupted but NioSelector was not shutdown.
                        // As this is most likely a bug in the handler of the user or it's client
                        // library we will log it.
                        //
                        // See https://github.com/netty/netty/issues/2426
                        if (logger.isDebugEnabled()) {
                            logger.debug("Selector.select() returned prematurely because the I/O thread " +
                                    "has been interrupted. Use shutdown() to shut the NioSelector down.");
                        }
                        selectReturnsImmediately = 0;
                    } else {
                        // Returned before the minSelectTimeout elapsed with nothing selected.
                        // This may be because of a bug in JDK NIO Selector provider, so increment the counter
                        // which we will use later to see if it's really the bug in JDK.
                        selectReturnsImmediately++;
```

上述的思路和Jetty的处理方式几乎是一样的，就是netty讲重建Selector的过程抽取成了一个方法，

叫做rebuildSelector，可以看看其方法：

```
public void rebuildSelector() {
    if (!isIoThread()) {
        taskQueue.add(new Runnable() {
            public void run() {
                rebuildSelector();
            }
        });
        return;
    }

    final Selector oldSelector = selector;
    final Selector newSelector;

    if (oldSelector == null) {
        return;
    }

    try {
        newSelector = SelectorUtil.open();
    } catch (Exception e) {
        logger.warn("Failed to create a new Selector.", e);
        return;
    }

    // Register all channels to the new Selector.
    int nChannels = 0;
    for (;;) {
        try {
            for (SelectionKey key: oldSelector.keys()) {
                try {
                    if (key.channel().keyFor(newSelector) != null) {
                        continue;
                    }

                    int interestOps = key.interestOps();
                    key.cancel();
                    key.channel().register(newSelector, interestOps, key.attachment());
                    nChannels ++;
                } catch (Exception e) {
                    logger.warn("Failed to re-register a Channel to the new Selector,", e);
                    close(key);
                }
            }
        } catch (ConcurrentModificationException e) {
            // Probably due to concurrent modification of the key set.
            continue;
        }

        break;
    }

    selector = newSelector;
```

开启新Selector

复制SelectionKey过去

替换成新的

基本上类似，这里就不再缀余。

分析到这里，可以看到为什么NIO框架如Netty，Grizzly，还有最近的炒得很热的Jboss的UnderTow，NIO远远不止这篇文章分析得这一个，还有很多，大可在JDK官网上去查，而这些框架都将NIO的很多不好用的问题，bug隐藏起来了，并加上诸如限流，字符转换，基于设计模式等特性，让开发人员更好的编写高并发的程序，而不用过多的网络的关注与细节。

由此可见，现在JAVA真是越来越危机了，从前几年的SSH把java ee给替换掉，到现在jdk都时不时冒出一个bug来,而且最近JDK8中的一个bug大有超过这个bug之势，jcp社区确实需要好好反省了，要不然java没落了，一干程序员又得下岗再就业了。

总结：

NIO的空转bug历史悠久流传广泛，应用服务器的前端框架一般都采取换一个新Selector的方式对此进行处理，屏蔽掉了JDK5/6的问题，但对于此问题来讲，还是尽量将JDK的版本更新到最新，或者使用NIO框架如Netty，Grizzly等进行研发，以免出更多的问题。

关于链表那点事~~ (/html/371/201510/401001007/1.html)