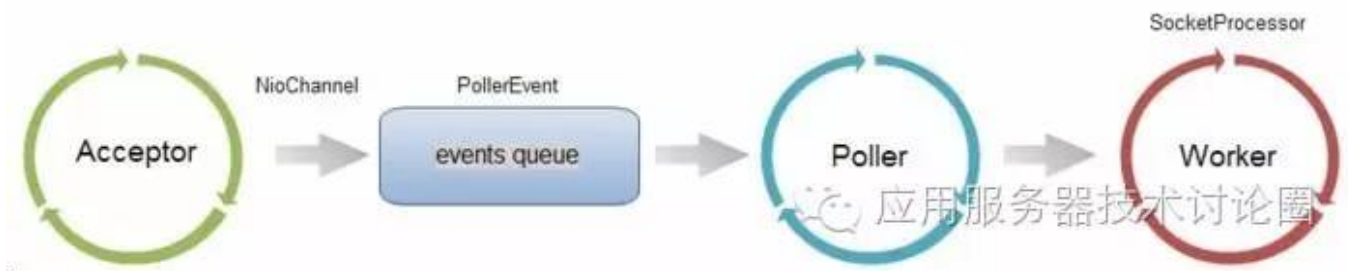


NioSelectorPool究竟有什么用

2016-06-25 22:04 feiying 0 阅读 117

我们回顾一下NIO通道中的主要流程：



上图中的Poller线程尤为关键，其主要的角色是使用java nio的API对网络事件Selection Key进行遍历，然后调用工作线程SocketProcessor进行http协议/请求头分析，这块我们比较清楚了，对于其反向的过程，http响应其实也是一样。

1.NIO通道下Servlet响应的回写过程

我们来看一个回写堆栈，一目了然：

```
owns: NioChannel (id=1/4)
NioSelectorPool.write(ByteBuffer, NioChannel, Selector, long, boolean) line: 171
InternalNioOutputBuffer.writeToSocket(ByteBuffer, boolean, boolean) line: 139
InternalNioOutputBuffer.addToBB(byte[], int, int) line: 197
InternalNioOutputBuffer.commit() line: 177
Http11NioProcessor(AbstractHttp11Processor<S>).action(ActionCode, Object) line: 740
Response.action(ActionCode, Object) line: 177
InternalNioOutputBuffer(AbstractOutputBuffer<S>).endRequest() line: 369
Http11NioProcessor(AbstractHttp11Processor<S>).endRequest() line: 1808
Http11NioProcessor(AbstractHttp11Processor<S>).process(SocketWrapper<S>) line: 1147
Http11NioProtocol$Http11ConnectionHandler(AbstractProtocol$AbstractConnectionHandler<S,P>).process(SocketWrapper<S>) line: 1500
NioEndpoint$SocketProcessor.doRun() line: 1500
NioEndpoint$SocketProcessor.run() line: 1456
ThreadPoolExecutor(ThreadPoolExecutor).runWorker(ThreadPoolExecutor$Worker) line: 1145
ThreadPoolExecutor$Worker.run() line: 615
TaskThread$WrappingRunnable.run() line: 61
TaskThread(Thread).run() line: 745
```

a.该堆栈在工作线程池中，也就是SocketProcessor，可以分析出来上述的堆栈是在NioEndpoint中，这个肯定是NIO通道

b.Http11NioProcessor是http11的协议处理器，目前在process方法中，说明正在处理，看到Response了，说明此时正在回写

下载《开发者大全》

下载 (/download/dev.apk)



c.Response是前端抽象出来的回写对象（和后端容器那个HttpResponse不一样），action方法是基于Reponse的不同的事件

d.commit方法，说明这个时候Servlet处理已经完成，Response正在进行commit，也就是将输出流中的数据通过socket进行发送

e.InternalNioOutputBuffer是一个写出缓冲，其内部主要有转码，缓存等操作

f.最后调用的是NioSelectorPool进行写出

我们就来分析一下NioSelectorPool的作用。

2.NIOSelectorPool的非阻塞写入模式

NioSelectorPool类是一个比较重要的类，它承担了NIO通道下的非阻塞或者阻塞模式下写入，也是一个Tomcat写入写出最底层的类。

先看看NioSelectorPool类的非阻塞写入，看一下NioSelectorPool 类的Write方法：

```

    public int write(ByteBuffer buf, NioChannel socket, Selector selector,
        long writeTimeout, boolean block) throws IOException {
        if ( SHARED && block ) {
            return blockingSelector.write(buf,socket,writeTimeout);
        }
        SelectionKey key = null;
        int written = 0;
        boolean timedout = false;
        int keycount = 1; //assume we can write
        long time = System.currentTimeMillis(); //start the timeout timer
        try {
            while ( (!timedout) && buf.hasRemaining() ) {
                int cnt = 0;
                if ( keycount > 0 ) { //only write if we were registered for a write
                    cnt = socket.write(buf); //write the data
                    if (cnt == -1) throw new EOFException();

                    written += cnt;
                    if (cnt > 0) {
                        time = System.currentTimeMillis(); //reset our timeout timer
                        continue; //we successfully wrote, try again without a selector
                    }
                }
                if (cnt==0 && (!block)) break; //don't block
            }
            if ( selector != null ) {
                //register OP_WRITE to the selector
                if (key==null) key = socket.getIOChannel().register(selector, SelectionKey.OP_WRITE);
                else key.interestOps(SelectionKey.OP_WRITE);
                if (writeTimeout==0) {
                    timedout = buf.hasRemaining();
                } else if (writeTimeout<0) {
                    keycount = selector.select();
                }
            }
        } catch (IOException e) {
            //...
        }
    }

```

写出成功

写入不成功

应用服务器技术讨论圈

通过代码分析，上面的NioChannel socket实际就是Poller现成的socketChannel通道，这个通道是注册在Poller线程中的Selector中，很多资料管这个Poller线程中的Selector叫做 主Selector，管NioSelectorPool叫做辅 Selector。

主Selector的作用就是接受Acceptor发现的网络包，然后进一步处理，解析网络包中的事件，并开启socketchannel通道注册到主Selector，或者继续接收上一次socketchannel通道中没有接受完的东西，这块逻辑我们已经在Poller线程的讲解中讲过了。

在NioSelectorPool的write方法中，调用主Selector中注册的socketchannel，如果cnt返回值大于0，说明网络目前发送正常，没有任何问题，可以看到代码中，直接就continue了；

如果cnt返回值为0，说明没有包被写出，那么这个时候的处理按道理来讲，应该是轮询，一直到发出去为止。

但是这样的处理，非常耗性能，特别是在Tomcat前端这种，寸时必争的地方，那么就想一个方法：



如果写入不成功的话，新启动一个Selector，这相当于使用Poller线程中注册过的socketchannel，没有占用Poller线程的主selector，新开启一个selector进行写入：

```

1.
if ( selector != null ) {
    //register OP_WRITE to the selector
    if (key==null) key = socket.getIOChannel().register(selector, SelectionKey.OP_WRITE);
    else key.interestOps(SelectionKey.OP_WRITE);
    if (writeTimeout==0) {
        timeout = buf.hasRemaining();
    } else if (writeTimeout<0) {
        keycount = selector.select();
    } else {
        keycount = selector.select(writeTimeout);
    }
}
2.
if (writeTimeout > 0 && (selector == null || keycount == 0) ) timeout = (System.currentTimeMillis() - writeTimeout);
} //while
if ( timeout ) throw new SocketTimeoutException();
} finally {
    if (key != null) {
        key.cancel();
        3.
        if (selector != null) selector.selectNow(); //remove the key from the selector
    }
}

```

看到其首先是将当前的socketchannel重新注册到新开启的selector中，然后进行select，一直到select返回成功写出，然后再将key注销掉。

总结一下，基于低并发来讲，一次socketchannel写入就会成功了，如果是高并发，非阻塞模式写入，需要新开启一个NIO通道，将socketchannel注册到新开启的Selector上，这样的好处就是不占用Poller线程。

这就是非阻塞的方式，也是NIO模式下的NioSelectorPool的最主要的作用。

3.Tomcat的NioSelectorPool配置

对于NioSelectorPool，在Tomcat中可以在NIO通道进行配置：

selectorPool.maxSelectors	(int) 以减少选择器的争用，在池中使用的选择器最大个数。命令行 <code>org.apache.tomcat.util.net.NioSelectorShared</code> 值设置为false时，使用此选项。默认值是200。
selectorPool.maxSpareSelectors	(int) 以减少选择器的争用，在池中使用的最大备用选择器个数。当选择器返回到池中时，系统可以决定保留它或者让他垃圾回收。当 <code>org.apache.tomcat.util.net.NioSelectorShared</code> 值设置为false时，使用此选项。默认值是-1（无限制）。
命令行选项	下面的命令行选项可用于NIO连接器： - <code>Dorg.apache.tomcat.util.net.NioSelectorShared=true false</code> 默认情况下是true。如果你想每个线程只有一个选择器，将此值设置为false。当你将它设置为false，你可以通过使用selectorPool.maxSelectors属性控制选择器池的大小。

命令行 -D 参数 `org.apache.tomcat.util.net.NioSelectorShared`，这个是标识新开启的selector是一个还是多个，从代码就可以看出来：

```

    public int write(ByteBuffer buf, NioChannel socket, Selector selector,
        long writeTimeout, boolean block) throws IOException {
        if ( SHARED && block ) {
            return blockingSelector.write(buf,socket,writeTimeout);
        }

        protected NioBlockingSelector blockingSelector;

```

如果开启了这个-D参数(默认其实也是true),那么新开启的selector仅仅只有1个，而当前的这次写入操作在Tomcat中的调用是block的话，也就是block属性为true，直接调用就是NioBlockingSelector来写入。

其它的两个参数，只有当-D参数配置不是true才有效，主要的作用就是控制新创建的selector在NioSelectorPool中的数目，selectorPool.maxSelectors是NioSelectorPool池中最大的selector个数，selectorPool.maxSpareSelectors是即使空闲下来，在NioSelectorPool池中还要保留的selector的个数；

可以在get，set方法中看到他们的身影：

```

    public void put(Selector s) throws IOException {
        if ( SHARED ) return;
        if ( enabled ) active.decrementAndGet();
        if ( enabled && (maxSpareSelectors== -1 || spare.get() < Math.min(maxSpareSelectors,maxSelectors)) ) {
            spare.incrementAndGet();
            selectors.offer(s);
        }
        else s.close();
    }

```

在每一次新增selector的时候，需要考虑看看当前池中目前有多少，如果没有超出的话，可以新建selector，否则只能使用池中的现有的selector；

```

public Selector get() throws IOException{
    if ( SHARED ) {
        return getSharedSelector();
    }
    if ( (!enabled) || active.incrementAndGet() >= maxSelectors ) {
        if ( enabled ) active.decrementAndGet();
        return null;
    }
    Selector s = null;
    try {
        s = selectors.size()>0?selectors.poll():null;
        if ( s == null ) {
            synchronized (Selector.class) {
                // Selector.open() isn't thread safe
                // http://bugs.sun.com/view_bug.do?bug_id=6427854
                // Affects 1.6.0_29, fixed in 1.7.0_01
                s = Selector.open();
            }
        }
        else spare.decrementAndGet();
    } catch (NoSuchElementException x) {
        try {
            synchronized (Selector.class) {
                // Selector.open() isn't thread safe
                // http://bugs.sun.com/view_bug.do?bug_id=6427854
                // Affects 1.6.0_29, fixed in 1.7.0_01
                s = Selector.open();
            }
        } catch (IOException iox) {
        }
    } finally {
        if ( s == null ) active.decrementAndGet(); // we were unable to find a selector
    }
    return s;
}

protected Selector getSharedSelector() throws IOException {
    if ( SHARED && SHARED_SELECTOR == null ) {
        synchronized ( NioSelectorPool.class ) {
            if ( SHARED_SELECTOR == null ) {
                synchronized (Selector.class) {
                    // Selector.open() isn't thread safe
                    // http://bugs.sun.com/view_bug.do?bug_id=6427854
                    // Affects 1.6.0_29, fixed in 1.7.0_01
                    SHARED_SELECTOR = Selector.open();
                }
                log.info("Using a shared selector for servlet write/read");
            }
        }
    }
    return SHARED_SELECTOR;
}

```

单例

对于每一次获得selector，都需要调用get方法，通过池中的状态来判断到底是否是新创建selector，还是用池中现在就有的；

如果设置了-D参数，那么可以看到getShaerdSelector通过单例模式，只能开启一个Selector；

对于上述的两个变量spare，active，这两个变量因为自增和自减，为了减少同步范围，使用的是原子变量

```

protected AtomicInteger active = new AtomicInteger(0);
protected AtomicInteger spare = new AtomicInteger(0);

```

这个是需要向Tomcat学习一下的；

最后，值得一说的就是，为什么Selector要开启多个呢，难道我们经常做NIO的程序，不是一个selector带一大堆通道吗？

实际上我们看javadoc：

SelectableChannel的register(Selector selector, ...)和Selector的select()方法都会操作Selector对象的共享资源all-keys集合。

SelectableChannel及Selector的实现对操作共享资源的代码块进行了同步,从而避免了对共享资源的竞争。

同步机制使得一个线程执行SelectableChannel的register(Selector selector, ...)时,

不允许另一个线程同时执行Selector的select()方法,反之亦然。

这里面已经讲述了，当通道的register，和selector的select都会产生keys的遍历，这种事会有同步竞争的，而selector的设置应该保证keys的数目可控在一定的范围内才行，因此，只要在高并发的情况下，只要资源允许的话，selector是多创建几个的，而上述的代码是一selector—socketchannel的模式，会减少上述的同步问题。

在Tomcat中，如果什么都不配置，默认就是：

```
1 2010-2-1 13:01:01 org.apache.tomcat.util.net.NioSelectorPool getSharedSelector
2 信息: Using a shared selector for servlet write/read
3 2010-2-1 13:01:01 org.apache.coyote.http11.Http11NioProtocol init
4 信息: Initializing Coyote HTTP/1.1 on http-8080
```

4.NIOSelectorPool的阻塞写入模式

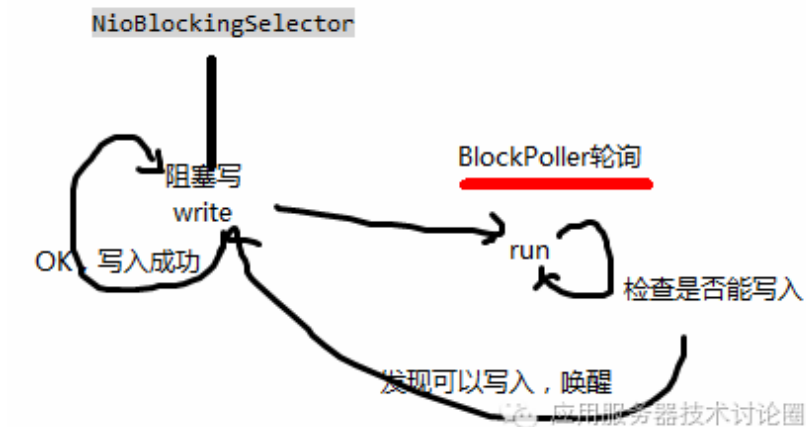
对于上一节的程序，如果write传入的参数是block，直接调用的就是NioBlockingSelector：

```
public int write(ByteBuffer buf, NioChannel socket, Selector selector,
                long writeTimeout, boolean block) throws IOException {
    if ( SHARED && block ) {
        return blockingSelector.write(buf,socket,writeTimeout);
    }
}

protected NioBlockingSelector blockingSelector;
```

对于传入的参数是否是block，这是根据不同的Tomcat场景选择，在很多场景下，如Servlet对流的阻塞读取，在规范中定义就是阻塞式的，而对于JSP页面的访问，block就为false；

再来看看NioBlockingSelector这个类，其整体时序框图为：



该Selector之所以叫做阻塞，是因为一旦通过socketchannel写入不成功，就会阻塞NioBlockingSelector的write线程，这个时候相当于直接阻塞了Tomcat的工作线程SocketProcessor了，所以这个就是阻塞；


```

public int write(ByteBuffer buf, NioChannel socket, long writeTimeout)
    throws IOException {
    SelectionKey key = socket.getIOChannel().keyFor(socket.getPoller().getSelector());
    if ( key == null ) throw new IOException("Key no longer registered");
    KeyReference reference = keyReferenceStack.pop();
    if (reference == null) {
        reference = new KeyReference();
    }
    KeyAttachment att = (KeyAttachment) key.attachment();
    int written = 0;
    boolean timedout = false;
    int keycount = 1; //assume we can write
    long time = System.currentTimeMillis(); //start the timeout timer
    try {
        while ( (!timedout) && buf.hasRemaining()) {
            if (keycount > 0) { //only write if we were registered for a write
                int cnt = socket.write(buf); //write the data
                if (cnt == -1)
                    throw new EOFException();
                written += cnt;
                if (cnt > 0) {
                    time = System.currentTimeMillis(); //reset our timeout timer
                    continue; //we successfully wrote, try again without a selector
                }
            }
            // 写入成功
            try {
                if ( att.getWriteLatch()==null || att.getWriteLatch().getCount()==0) att.startWriteLatch(1);
                poller.add(att,SelectionKey.OP_WRITE,reference);
                if (writeTimeout < 0) {
                    att.awaitWriteLatch(Long.MAX_VALUE,TimeUnit.MILLISECONDS);
                } else {
                    // 直接wait
                    att.awaitWriteLatch(writeTimeout,TimeUnit.MILLISECONDS);
                }
            } catch (InterruptedException ignore) {
                // Ignore
            }
            // 写入不成功
            if ( att.getWriteLatch()!=null && att.getWriteLatch().getCount()> 0) {
                //we got interrupted, but we haven't received notification from the poller.
                keycount = 0;
            } else {
                //latch countdown has happened
                keycount = 1;
                att.resetWriteLatch();
            }
        }

        if (writeTimeout > 0 && (keycount == 0))
            timedout = (System.currentTimeMillis() - time) >= writeTimeout;
    } //while
    if (timedout)

```

上述的write方法和NioSelectorPool中的write方法差不多，一次socketchannel写入成功，那最好，如果不成功的话，一共做两件事：

第一，这里通过一个Latch在主线程进行wait：

```

protected void awaitLatch(CountDownLatch latch, long timeout, TimeUnit unit) throws InterruptedException {
    if ( latch == null ) throw new IllegalStateException("Latch cannot be null");
    // Note: While the return value is ignored if the latch does time
    // out, logic further up the call stack will trigger a
    // SocketTimeoutException
    latch.await(timeout,unit);
}
public void awaitReadLatch(long timeout, TimeUnit unit) throws InterruptedException { awaitLatch(readLatch,timeout,unit);}
public void awaitWriteLatch(long timeout, TimeUnit unit) throws InterruptedException { awaitLatch(writeLatch,timeout,unit);}

```

我们可以看到，这个同步机制就是使用的java并发包中的CountDownLatch，也就是倒计时门栓，这个同步门栓的好处就在于是通过倒计时来进行控制同步。

第二，就是使用了一个BlockPoller的子线程，将这个socketchannel关注的key加入到BlockPoller子线程中去，由子线程进行轮询：

```

public void add(final KeyAttachment key, final int ops, final KeyReference ref) {
    if ( key == null ) return;
    NioChannel nch = key.getSocket();
    if ( nch == null ) return;
    final SocketChannel ch = nch.getIOChannel();
    if ( ch == null ) return;

    Runnable r = new Runnable() {
        @Override
        public void run() {
            SelectionKey sk = ch.keyFor(selector);
            try {
                if (sk == null) {
                    sk = ch.register(selector, ops, key);
                    ref.key = sk;
                } else if (!sk.isValid()) {
                    cancel(sk, key, ops);
                } else {
                    sk.interestOps(sk.interestOps() | ops);
                }
            } catch (CancelledKeyException cx) {
                cancel(sk, key, ops);
            } catch (ClosedChannelException cx) {
                cancel(sk, key, ops);
            }
        }
    };
    events.offer(r);
    wakeup();
}

```

 应用服务器技术讨论圈

加入的子线程最后被放到同步队列中：

```

protected final SynchronizedQueue<Runnable> events =
    new SynchronizedQueue<>();

```

在run方法中被执行：


```

@Override
public void run() {
    while (run) {
        try {
            events();
            int keyCount = 0;
            try {
                int i = wakeupCounter.get();
                if (i > 0)
                    keyCount = selector.selectNow();
                else {
                    wakeupCounter.set(-1);
                    keyCount = selector.select(1000);
                }
                wakeupCounter.set(0);
                if (!run) break;
            }
        }

        Iterator<SelectionKey> iterator = keyCount > 0 ? selector.selectedKeys().iterator() : null;

        // Walk through the collection of ready keys and dispatch
        // any active event.
        while (run && iterator != null && iterator.hasNext()) {
            SelectionKey sk = iterator.next();
            KeyAttachment attachment = (KeyAttachment)sk.attachment();
            try {
                attachment.access();
                iterator.remove();
                sk.interestOps(sk.interestOps() & (~sk.readyOps()));
                if (sk.isReadable()) {
                    countDown(attachment.getReadLatch());
                }
                if (sk.isWritable()) {
                    countDown(attachment.getWriteLatch());
                }
            } catch (CancelledKeyException ckx) {
                sk.cancel();
                countDown(attachment.getReadLatch());
                countDown(attachment.getWriteLatch());
            }
        }
    }
}

```

唤醒主线程中的await

应用服务器技术讨论圈

当run线程中发现网络可通了，实际上就是selector.selectNow返回了，并且关注的keys得到了，说明这个时候通过socket

channel发送数据包了，这个时候主线程await被唤醒，while循环再进行执行，下一次循环中就可以通过socketchannel.write发送数据了；

试想一下，如果没有这段代码，那么主线程中的while循环就得进行空转，而主线程就是SocketProcessor，这个是多么损害Tomcat前端性能呢！

总结：

NioSelectorPool是Tomcat对Servlet的写入和写出的实现类，根据不同的场景分为阻塞和非阻塞两种方式，在Tomcat中可以配置是否selector共享（1个）还是通过池管理来控制多个，默认就是1个。


分享：

下载《开发者大全》

下载 (/download/dev.apk)



阅读 117

 0

应用服务器技术讨论圈 更多文章

东方通加码大数据业务 拟募资8亿收购微智信业 (/html/308/201504/206211355/1.html)

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

坏人都是赢家！ (/html/300/201604/2651257019/1.html)

产品发布后，一个QA的总结与思考(续) (/html/49/201510/400017216/1.html)

Cortex-A73和Mali-G71发布，重新定义2017年旗舰移动设备 (/html/431/201605/2650236680/1.html)

据说程序员写完代码是这个样子，99%的人都中枪了 (/html/186/201512/401242792/1.html)

大数据时代面临的 8 大趋势 (/html/318/201503/203463823/1.html)