# 准备

### WPS里面如何去掉红色波浪线

<https://jingyan.baidu.com/article/d5a880ebc1e62b13f047cc49.html>

# ServerBootstrap

# ServerBootstrap.bind().sync()详解

## b.bind()

***//创建一个新的NioServerSocketCahnenl并且与端口绑定*** **public** ChannelFuture bind() {  
 validate();  
 SocketAddress localAddress = **this**.**localAddress**;  
 **if** (localAddress == **null**) {  
 **throw new** IllegalStateException(**"localAddress not set"**);  
 }  
 **return doBind**(localAddress);  
}

**private** ChannelFuture doBind(**final** SocketAddress localAddress) {

**//将Channel注册到EventLoop，并且返回DefaultChannelPromise,**

**//DefaultChannelPromise实现了ChannelFuture接口** **final** ChannelFuture regFuture = **initAndRegister()**;

**//返回NioServerSocketChannel** **final** Channel channel = regFuture.channel();  
 **if** (regFuture.cause() != **null**) {  
 **return** regFuture;  
 }

**//将Channel注册到EventLoop的时使用了ChannelFuture，一旦注册完成，**

**//ChannelFuture将得到通知(即regFuture.isDone()为true表示注册完成)**

**if** (regFuture.isDone()) {  
 ***//至此，我们知道注册已经完成并成功***

ChannelPromise promise = channel.newPromise();  
 ***doBind0***(regFuture, channel, localAddress, promise);  
 **return** promise;  
 } **else** {

**//1.Channel注册eventLoop未完成，即ChannelFuture的result字段**

**//还没被设置成SUCCESS** ***//2.将Channel向EventLoop注册时使用的ChannelFuture几乎总是已经完成***

***//（即regFuture.isDone()为true）,但以防万一,也就说else这步基本上走***

***//不到***

***//3.在这里为什么要有else呢？因为有两个线程(主线程和eventloop线程)***

***//在执行，如果主线程执行比较快则需要执行else里面的代码，以供eventLoop***

***//线程执行的指定部分时再来执行*****final** PendingRegistrationPromise promise = **new** PendingRegistrationPromise(channel);  
 regFuture.addListener(**new** ChannelFutureListener() {  
 @Override  
 **public void** operationComplete(ChannelFuture future) **throws** Exception {  
 Throwable cause = future.cause();  
 **if** (cause != **null**) {  
 **//将Channel向EventLoop注册失败，因此一旦我们尝试访问**

**//通道的EventLoop,ChannelPromise将直接失败，不会导致**

**//llegalStateException @说明1@**

promise.setFailure(cause);  
 } **else** {  
 ***//*注册成功，因此请设置要使用的正确执行器。**

*// See https://github.com/netty/netty/issues/2586* promise.registered();  
  
 *doBind0*(regFuture, channel, localAddress, promise);  
 }  
 }  
 });  
 **return** promise;  
 }  
}

**-------------------------------------------------------------**

**说明1：当Channel已经注册到EventLoop时，一旦我们尝试访问通道的EventLoop，**

**ChannelPromise将直接失败，不会导致llegalStateException**

**public final void** register(EventLoop eventLoop, **final** ChannelPromise promise) {  
 **if** (eventLoop == **null**) {  
 **throw new** NullPointerException(**"eventLoop"**);  
 }  
 **if** (isRegistered()) {  
 promise.setFailure(**new** IllegalStateException(**"registered to an event loop already"**));  
 **return**;  
 }  
 **if** (!isCompatible(eventLoop)) {  
 promise.setFailure(  
 **new** IllegalStateException(**"incompatible event loop type: "** + eventLoop.getClass().getName()));  
 **return**;  
 }

....  
}

**--------------------------------------------------------------**

### initAndRegister()

**final** ChannelFuture initAndRegister() {  
 Channel channel = **null**;  
 **try** {

**//创建Channel(即NioServerSocketChannel)** channel = **channelFactory**.newChannel();

**//初始化Channel** **init(channel);** } **catch** (Throwable t) {  
 **if** (channel != **null**) {

***//1.如果newChannel出现异常(比如SocketException("too many open***

***//files")),那么channel可以为null***

***//2.因为ServerSocketChannel创建成功，需要关闭***channel.unsafe().closeForcibly();

***//由于channel还没有注册，因此我们需要强制使用GlobalEventExecutor***

***//（注意：这里channel已关闭）*****return new** DefaultChannelPromise(channel, GlobalEventExecutor.***INSTANCE***).setFailure(t);  
 }

***//由于Channel还没有注册，因此我们需要强制使用GlobalEventExecutor*****return new** DefaultChannelPromise(**new** FailedChannel(), GlobalEventExecutor.***INSTANCE***).setFailure(t);  
 }  
  
 ChannelFuture regFuture = **config().group().register(channel);** **if** (regFuture.cause() != **null**) {  
 **if** (channel.isRegistered()) {  
 channel.close();  
 } **else** {  
 channel.unsafe().closeForcibly();  
 }  
 }

**----------------------------------------------------------------------**

**说明：1.DefaultChannelPromise的超类DefaultPromise的result字段取值有如下5种情况：null, *SUCCESS,UNCANCELLABLE, CANCELLATION\_CAUSE\_HOLDER,* CauseHolder, 在DefaultChannelPromise初始化的时候result值为null,**

**如果NioServerSocketChannel创建出现异常又或者将ServerSocketChannel注册到Selector时候出现异常,result值为CauseHolder，如下:**  
**final** ChannelFuture initAndRegister() {  
 Channel channel = **null**;  
 **try** {  
 channel = **channelFactory**.newChannel();  
 init(channel);  
 } **catch** (Throwable t) {  
 **if** (channel != **null**) {  
channel.unsafe().closeForcibly();  
**return new** DefaultChannelPromise(channel, GlobalEventExecutor.***INSTANCE***).setFailure(t);  
 }  
**return new** DefaultChannelPromise(**new** FailedChannel(), GlobalEventExecutor.***INSTANCE***).setFailure(t);  
 }

...  
}

**又如：**

**private void** register0(ChannelPromise promise) {  
 **try** {  
 ...  
 } **catch** (Throwable t) {  
 *// Close the channel directly to avoid FD leak.* closeForcibly();  
 **closeFuture**.setClosed();  
  **safeSetFailure(promise, t);**  
 }  
}

**当将ServerSocketChannel注册Selector的任务开始的时候设置result值为*UNCANCELLABLE***

**private void** register0(ChannelPromise promise) {  
 ...**if** (!promise.setUncancellable() || !ensureOpen(promise)) {  
 **return**;  
 }  
 ...  
}

**当将ServerSocketChannel成功注册到Selector之后设置result值为SUCCESS,因为promise是用户线程与eventLoop共享的，用户线程会根据eventLoop线程设置promise的**

**private void** register0(ChannelPromise promise) {  
 ...

safeSetSuccess(promise);

...  
}

**--------------------------------------------------------------**

#### init(channel)

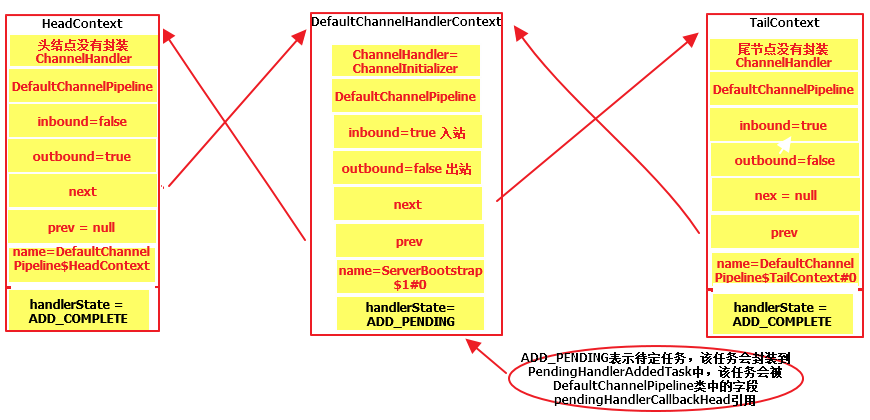
**void** init(Channel channel) **throws** Exception {  
 **final** Map<ChannelOption<?>, Object> options = options0();  
 **synchronized** (options) {  
 *setChannelOptions*(channel, options, ***logger***);  
 }  
  
 **final** Map<AttributeKey<?>, Object> attrs = attrs0();  
 **synchronized** (attrs) {  
 **for** (Entry<AttributeKey<?>, Object> e: attrs.entrySet()) {  
 @SuppressWarnings(**"unchecked"**)  
 AttributeKey<Object> key = (AttributeKey<Object>) e.getKey();  
 channel.attr(key).set(e.getValue());  
 }  
 }  
  **//获取与该Channel关联的pipeline** ChannelPipeline p = channel.pipeline();  
 **//workerGroup** **final** EventLoopGroup currentChildGroup = **childGroup**;

**//封装的EchoServerHandler的ChannelInitializer** **final** ChannelHandler currentChildHandler = **childHandler**;  
 **final** Entry<ChannelOption<?>, Object>[] currentChildOptions;  
 **final** Entry<AttributeKey<?>, Object>[] currentChildAttrs;  
 **synchronized** (**childOptions**) {  
 currentChildOptions = **childOptions**.entrySet().toArray(*newOptionArray*(**childOptions**.size()));  
 }  
 **synchronized** (**childAttrs**) {  
 currentChildAttrs = **childAttrs**.entrySet().toArray(*newAttrArray*(**childAttrs**.size()));  
 }

**//将该ChannelInitializer封装成ChannelHandlerContext并添加到链表，并且**

**//将ChannelHandlerContext封装成pendingHandlerCallbackHead**  
 p.addLast(**new** ChannelInitializer<Channel>() {  
 @Override  
 **public void** initChannel(**final** Channel ch) **throws** Exception {  
 **final** ChannelPipeline pipeline = ch.pipeline();  
 ChannelHandler handler = **config**.handler();  
 **if** (handler != **null**) {  
 pipeline.addLast(handler);  
 }  
  **//EventLoop.execute将该任务添加到任务队列taskQueue** ch.eventLoop().execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 pipeline.addLast(**new** ServerBootstrapAcceptor(  
 ch, currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));  
 }  
 });  
 }  
 });  
}

#### pipeline链表结构



#### config().group().register(channel)

**向eventLoop注册Channel时使用了ChannelFuture。一旦注册完成，传递的ChannelFuture将得到通知，并且返回。**

**public** ChannelFuture register(**final** ChannelPromise promise) {  
 ObjectUtil.*checkNotNull*(promise, **"promise"**);  
 promise.channel().unsafe().register(**this**, promise);  
 **return** promise;  
}

@Override  
**public final void** register(EventLoop eventLoop, **final** ChannelPromise promise) {  
 **if** (eventLoop == **null**) {  
 **throw new** NullPointerException(**"eventLoop"**);  
 }  
 **if** (isRegistered()) {  
 promise.setFailure(**new** IllegalStateException(**"registered to an event loop already"**));  
 **return**;  
 }  
 **if** (!isCompatible(eventLoop)) {  
 promise.setFailure(  
 **new** IllegalStateException(**"incompatible event loop type: "** + eventLoop.getClass().getName()));  
 **return**;  
 }  
  
 AbstractChannel.**this**.**eventLoop** = eventLoop;  
  
 **if** (eventLoop.inEventLoop()) {  
 register0(promise);  
 } **else** {  
 **try** {

**//1.eventLoop.execute将该任务添加到任务队列taskQueue**

**//2.该任务是添加到任务队列taskQueue的第一个任务**

**//3.由于当前线程不是分配EventLoop的线程，所以会为EventLoop分配**

**//一个线程（即reactor线程），该线程执行EventLoop的run方法，**

**//该run方法里面是个死循环。@说明1@** eventLoop.**execute**(**new** Runnable() {  
 @Override  
 **public void** run() {  
 register0(promise);  
 }  
 });  
 } **catch** (Throwable t) {  
 ***logger***.warn(  
 **"Force-closing a channel whose registration task was not accepted by an event loop: {}"**,  
 AbstractChannel.**this**, t);  
 closeForcibly();  
 **closeFuture**.setClosed();  
 safeSetFailure(promise, t);  
 }  
 }  
}

**-------------------------------------------------------------------**

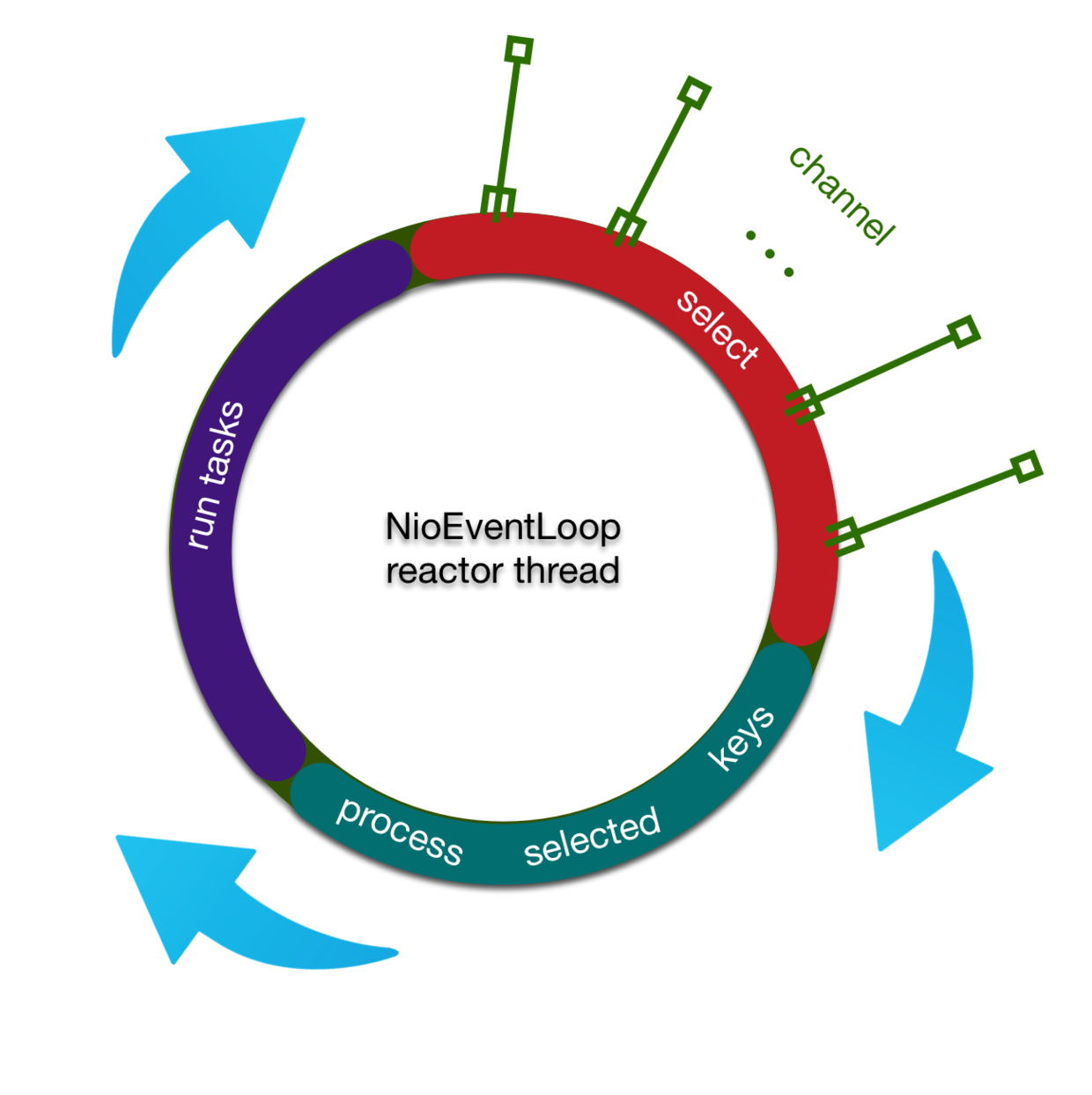
**说明1：reactor线程(即EventLoop.run()方法)主要做了3件事情，用下面一幅图就可以说明。**

**reactor线程大概做的事情分为对三个步骤不断循环：**

**(1)轮询出注册到Selector上的Channel的IO事件(select)**

**(2)处理产生网络IO的Channel**

**(3)处理任务**

[](http://images2017.cnblogs.com/blog/782702/201708/782702-20170830131328390-243400633.png)

**-------------------------------------------------------------**

##### eventLoop.execute(Runnable)

@Override  
**public void** execute(Runnable task) {  
 **if** (task == **null**) {  
 **throw new** NullPointerException(**"task"**);  
 }  
   
 **boolean** inEventLoop = inEventLoop();  
 addTask(task);

**//如果不在eventLoop线程中则尝试为eventLoop分配线程，**

**//如果已经为eventLoop分配过线程则不会再次分配**  
 **if** (!inEventLoop) {  
 startThread();  
 **if** (isShutdown() && removeTask(task)) {  
 *reject*();  
 }  
 }  
 **//1.addTaskWakesUp:**

**// true 如果当且仅当调用*addTask(Runnable)才唤醒执行线程时***

***// false 默认初始化为false***

**//2.判断该任务是不是NonWakeupRunnable类型，比如WriteTask** **if** (!**addTaskWakesUp** && **wakesUpForTask**(task)) {  
 **wakeup**(inEventLoop);  
 }  
}

###### wakesUpForTask

@Override  
**protected boolean** wakesUpForTask(Runnable task) {  
 **return** !(task **instanceof** NonWakeupRunnable);  
}

###### wakeup

@Override  
**protected void** wakeup(**boolean** inEventLoop) {

**//eventLoop在初始化的时候wakenUp为false，设置eventLoop的wakenUp为true**  
 **if** (!inEventLoop && **wakenUp**.compareAndSet(**false**, **true**)) {

**//使尚未返回的第一个选择操作selector.select()立即返回** **selector**.wakeup();  
 }  
}

**//刚创建EventLoop的时候并没有为eventLoop分配线程,初始状态**

**//state=*ST\_NOT\_STARTED;当给eventLoop分配了线程之后，eventLoop的状态state***

***//变成ST\_STARTED***

**private void** startThread() {  
 **if** (**state** == ***ST\_NOT\_STARTED***) {  
 **if** (***STATE\_UPDATER***.compareAndSet(**this**, ***ST\_NOT\_STARTED***, ***ST\_STARTED***)) {  
 **try** {  
 **doStartThread();** } **catch** (Throwable cause) {  
 ***STATE\_UPDATER***.set(**this**, ***ST\_NOT\_STARTED***);  
 PlatformDependent.*throwException*(cause);  
 }  
 }  
 }  
}

**private void** doStartThread() {  
 **assert thread** == **null**;

**//1.executor类型为ThreadPerTaskExecutor，ThreadPerTaskExecutor里面包**

**//一个线程工厂DefaultThreadFactory，为每个任务创建一个线程来执行**

**//2.ThreadPerTaskExecutor在创建EventLoop对象的时候指定** **executor**.execute(**new** Runnable() {  
 @Override  
 **public void** run() {

**//为eventLoop分配一个线程执行eventLoop的run方法** **thread** = Thread.*currentThread*();

**//这里为什么要判断interrupted,因为eventLoop先创建好可以配置是否**

**//中断线程** **if** (**interrupted**) {

//1.对处于慢系统调用(比如sleep,select都是慢系统调用)的

//线程调用interrupt()会抛出InterruptedException异常,sleep

//方法在抛出InterruptedException之前，Java虚拟机会先将该线程

//的中断标识位清除，然后抛出InterruptedException，此时调用

//isInterrupted()方法将会返回false。

//2.对处于非慢系统调用的线程调用interrupt()只会将线程的中断

//标识位设置为true，一旦该线程发生慢系统调用则就会中断该线程

//并抛出InterruptedException异常

//3.中断select()操作会抛出ClosedByInterruptException异常  
 **thread**.interrupt();  
 }  
  
 **boolean** success = **false**;

//更新内部时间戳，该时间戳指示最近一次执行提交的任务的时间。

//(runAllTasks()和runAllTasks(long)自动更新这个时间戳，因此通

//常不需要调用这个方法。但是，如果使用takeTask()或pollTask())

//手动抓取任务，则必须在任务执行循环结束时调用此方法以进行精确的

//静默期检查  
 updateLastExecutionTime();  
 **try** {

**//执行NioEventLoop.run()** SingleThreadEventExecutor.**this**.run();  
 success = **true**;  
 } **catch** (Throwable t) {  
 ***logger***.warn(**"Unexpected exception from an event executor: "**, t);  
 } **finally** {  
 **for** (;;) {  
 **int** oldState = **state**;  
 **if** (oldState >= ***ST\_SHUTTING\_DOWN*** || ***STATE\_UPDATER***.compareAndSet(  
 SingleThreadEventExecutor.**this**, oldState, ***ST\_SHUTTING\_DOWN***)) {  
 **break**;  
 }  
 }  
  
 *// Check if confirmShutdown() was called at the end of the loop.* **if** (success && **gracefulShutdownStartTime** == 0) {  
 ***logger***.error(**"Buggy "** + EventExecutor.**class**.getSimpleName() + **" implementation; "** +  
 SingleThreadEventExecutor.**class**.getSimpleName() + **".confirmShutdown() must be called "** +  
 **"before run() implementation terminates."**);  
 }  
  
 **try** {  
 *// Run all remaining tasks and shutdown hooks.* **for** (;;) {  
 **if** (confirmShutdown()) {  
 **break**;  
 }  
 }  
 } **finally** {  
 **try** {  
 cleanup();  
 } **finally** {  
 ***STATE\_UPDATER***.set(SingleThreadEventExecutor.**this**, ***ST\_TERMINATED***);  
 **threadLock**.release();  
 **if** (!**taskQueue**.isEmpty()) {  
 ***logger***.warn(  
 **"An event executor terminated with "** +  
 **"non-empty task queue ("** + **taskQueue**.size() + **')'**);  
 }  
  
 **terminationFuture**.setSuccess(**null**);  
 }  
 }  
 }  
 }  
 });  
}

**-------------------------------------------------------------------------------------------------------**

说明：NioEventLoop有5个状态，如下：

**private static final int *ST\_NOT\_STARTED*** = 1;  
**private static final int *ST\_STARTED*** = 2;  
**private static final int *ST\_SHUTTING\_DOWN*** = 3;  
**private static final int *ST\_SHUTDOWN*** = 4;  
**private static final int *ST\_TERMINATED*** = 5;

1. 当NioEventLoop初始化的时候值为***ST\_NOT\_STARTED***
2. ***当给NioEventLoop分配了线程之后，值为ST\_STARTED***

### doBind0(regFuture, channel, localAddress, promise)

**private static void** doBind0(  
 **final** ChannelFuture regFuture, **final** Channel channel,  
 **final** SocketAddress localAddress, **final** ChannelPromise promise) {

***//1.该方法在channelRegistered()方法被触发之前被调用。***

**//让用户处理程序有机会在其channelRegistered()实现中设置pipeline。**

**//2.eventLoop.execute()将该任务task添加到taskQueue中**  
**channel.eventLoop().execute**(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **if** (regFuture.isSuccess()) {  
 channel.bind(localAddress, promise).addListener(ChannelFutureListener.***CLOSE\_ON\_FAILURE***);  
 } **else** {  
 promise.setFailure(regFuture.cause());  
 }  
 }  
 });  
}

#### eventLoop.execute(Runnable task)

//该任务task的逻辑如下

@Override  
**public void** execute(Runnable task) {  
 **if** (task == **null**) {  
 **throw new** NullPointerException(**"task"**);  
 }  
  
 **boolean** inEventLoop = inEventLoop();  
 addTask(task);  
 **if** (!inEventLoop) {  
 startThread();  
 **if** (isShutdown() && removeTask(task)) {  
 *reject*();  
 }  
 }

**//1.每添加一个任务都尝试使尚未返回的第一个选择操作select()返回。**  **//2.addTaskWakesUp** :

**// true 表示当且仅当addTask(Runnable)调用将唤醒执行线程,**

**// false 默认值**

**//3.只要有新任务加入，selector.select(timeOut)操作阻塞就会被释放** **if** (!**addTaskWakesUp** && wakesUpForTask(task)) {  
 **wakeup(inEventLoop)**;  
 }  
}

##### wakeup(inEventLoop)

**protected void** wakeup(**boolean** inEventLoop) {  
 **if** (!inEventLoop && **wakenUp**.compareAndSet(**false**, **true**)) {  
 **selector**.wakeup();  
 }  
}

## b.bind().sync()

@Override  
**public** ChannelPromise sync() **throws** InterruptedException {  
 **super**.sync();  
 **return this**;  
}

@Override  
**public** Promise<V> sync() **throws** InterruptedException {  
 **await**();  
 rethrowIfFailed();  
 **return this**;  
}

@Override  
**public** Promise<V> await() **throws** InterruptedException {

**//判断该Future任务是否执行成功** **if** (isDone()) {**//如果执行成功直接返回** **return this**;  
 }  
 **//测试当前线程是否被中断，通过该方法可以清除线程中断的状态。换句话说，如果**

**//这个方法连续被调用两次，那么第二次调用返回false**  
 **if** (Thread.*interrupted*()) {

//如果当前线程被中断过抛出中断异常  
 **throw new** InterruptedException(toString());  
 }  
  
 **checkDeadLock();**  
  
 **synchronized** (**this**) {  
 **while** (!isDone()) {

**//等待者计数加一** incWaiters();  
 **try** {

**//当前线程等待**  
 **wait();**  
 } **finally** {

**//当从当前线程唤醒的时候，计数减一** decWaiters();  
 }  
 }  
 }  
 **return this**;  
}

### checkDeadLock

**protected void** checkDeadLock() {  
 EventExecutor e = executor();  
 **if** (e != **null** && e.inEventLoop()) {  
 **throw new** BlockingOperationException(toString());  
 }  
}

### incWaiters

**private void** incWaiters() {  
 **if** (**waiters** == Short.***MAX\_VALUE***) {  
 **throw new** IllegalStateException(**"too many waiters: "** + **this**);  
 }  
 ++**waiters**;  
}

# eventLoop线程

**private void** doStartThread() {  
 **assert thread** == **null**;  
 **executor**.execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **thread** = Thread.*currentThread*();  
 **if** (**interrupted**) {//判断eventLoop线程是否中断  
 **thread**.interrupt();  
 }  
  
 **boolean** success = **false**;  
 **updateLastExecutionTime();** **try** {  
 SingleThreadEventExecutor.**this**.run();  
 success = **true**;  
 } **catch** (Throwable t) {  
 ***logger***.warn(**"Unexpected exception from an event executor: "**, t);  
 } **finally** {  
 **for** (;;) {  
 **int** oldState = **state**;  
 **if** (oldState >= ***ST\_SHUTTING\_DOWN*** || ***STATE\_UPDATER***.compareAndSet(  
 SingleThreadEventExecutor.**this**, oldState, ***ST\_SHUTTING\_DOWN***)) {  
 **break**;  
 }  
 }  
  
 *// Check if confirmShutdown() was called at the end of the loop.* **if** (success && **gracefulShutdownStartTime** == 0) {  
 ***logger***.error(**"Buggy "** + EventExecutor.**class**.getSimpleName() + **" implementation; "** +  
 SingleThreadEventExecutor.**class**.getSimpleName() + **".confirmShutdown() must be called "** +  
 **"before run() implementation terminates."**);  
 }  
  
 **try** {  
 *// Run all remaining tasks and shutdown hooks.* **for** (;;) {  
 **if** (confirmShutdown()) {  
 **break**;  
 }  
 }  
 } **finally** {  
 **try** {  
 cleanup();  
 } **finally** {  
 ***STATE\_UPDATER***.set(SingleThreadEventExecutor.**this**, ***ST\_TERMINATED***);  
 **threadLock**.release();  
 **if** (!**taskQueue**.isEmpty()) {  
 ***logger***.warn(  
 **"An event executor terminated with "** +  
 **"non-empty task queue ("** + **taskQueue**.size() + **')'**);  
 }  
  
 **terminationFuture**.setSuccess(**null**);  
 }  
 }  
 }  
 }  
 });  
}

updateLastExecutionTime();

**protected void** updateLastExecutionTime() {

//下一次执行时间  
 **lastExecutionTime** = ScheduledFutureTask.***nanoTime***();  
}

**static long** nanoTime() {  
 **return** System.*nanoTime*() - ***START\_TIME***;  
}

# Netty4之Future/Promise异步模型

****在并发编程中，我们通常会用到一组非阻塞的模型：Promise，Future 和 Callback。其中的 Future 表示一个可能还没有实际完成的异步任务的结果，针对这个结果可以添加 Callback 以便在任务执行成功或失败后做出对应的操作，而 Promise 交由任务执行者，任务执行者通过 Promise 可以标记任务完成或者失败。 可以说这一套模型是很多异步非阻塞架构的基础。Netty 4中正提供了这种Future/Promise异步模型。****

****Netty文档说明Netty的网络操作都是异步的， 在源码上大量使用了Future/Promise模型，在Netty里面也是这样定义的：****

* ****Future接口定义了isSuccess(),isCancellable(),cause(),这些判断异步执行状态的方法。（read-only）****
* ****Promise接口在extneds future的基础上增加了setSuccess(), setFailure()这些方法。（writable）****

**java.util.concurrent.Future是Java提供的接口，表示异步执行的状态，Future的get方法会判断任务是否执行完成，如果完成就返回结果，否则阻塞线程，直到任务完成。**

**public** V get() **throws** InterruptedException, ExecutionException {  
 **int** s = **state**;  
 **if** (s <= ***COMPLETING***)  
 s = awaitDone(**false**, 0L);  
 **return** report(s);  
}

**Netty扩展了Java的Future，最主要的改进就是增加了监听器Listener接口，通过监听器可以让异步执行更加有效率，不需要通过get来等待异步执行结束，而是通过监听器回调来精确地控制异步执行结束的时间点。**

**public interface** Future<V> **extends** java.util.concurrent.Future<V> {  
  
 **boolean** isSuccess();  
  
 **boolean** isCancellable();  
  
 Throwable cause();  
Future<V> addListener(GenericFutureListener<? **extends** Future<? **super** V>> listener);  
  
 Future<V> addListeners(GenericFutureListener<? **extends** Future<? **super** V>>... listeners);  
  
 Future<V> removeListener(GenericFutureListener<? **extends** Future<? **super** V>> listener);  
  
 Future<V> removeListeners(GenericFutureListener<? **extends** Future<? **super** V>>... listeners);  
  
 Future<V> sync() **throws** InterruptedException;  
Future<V> syncUninterruptibly();  
Future<V> await() **throws** InterruptedException;  
Future<V> awaitUninterruptibly();  
  
 **boolean** await(**long** timeout, TimeUnit unit) **throws** InterruptedException;  
  
 **boolean** await(**long** timeoutMillis) **throws** InterruptedException;  
  
 **boolean** awaitUninterruptibly(**long** timeout, TimeUnit unit);  
  
 **boolean** awaitUninterruptibly(**long** timeoutMillis);  
  
 V getNow();  
  
 **boolean** cancel(**boolean** mayInterruptIfRunning);  
}

**ChannelFuture接口扩展了Netty的Future接口，表示一种没有返回值的异步调用，同时关联了Channel，跟一个Channel绑定**

**public interface** ChannelFuture **extends** Future<Void> {  
Channel channel();  
  
 @Override  
 ChannelFuture addListener(GenericFutureListener<? **extends** Future<? **super** Void>> listener);  
  
 @Override  
 ChannelFuture addListeners(GenericFutureListener<? **extends** Future<? **super** Void>>... listeners);  
  
 @Override  
 ChannelFuture removeListener(GenericFutureListener<? **extends** Future<? **super** Void>> listener);  
  
 @Override  
 ChannelFuture removeListeners(GenericFutureListener<? **extends** Future<? **super** Void>>... listeners);  
  
 @Override  
 ChannelFuture sync() **throws** InterruptedException;  
  
 @Override  
 ChannelFuture syncUninterruptibly();  
  
 @Override  
 ChannelFuture await() **throws** InterruptedException;  
  
 @Override  
 ChannelFuture awaitUninterruptibly();  
**boolean** isVoid();  
}

**Promise接口也扩展了Future接口，它表示一种可写的Future，就是可以设置异步执行的结果**

**public interface** Promise<V> **extends** Future<V> {  
Promise<V> setSuccess(V result);  
  
 **boolean** trySuccess(V result);  
  
 Promise<V> setFailure(Throwable cause);  
 **boolean** tryFailure(Throwable cause);  
 }

**ChannelPromise接口扩展了Promise和ChannelFuture，绑定了Channel，又可写异步执行结构，又具备了监听者的功能，是Netty实际编程使用的表示异步执行的接口**

**public interface** ChannelPromise **extends** ChannelFuture, Promise<Void> {  
  
 @Override  
 Channel channel();  
  
 @Override  
 ChannelPromise setSuccess(Void result);  
  
 ChannelPromise setSuccess();  
  
 **boolean** trySuccess();  
  
 @Override  
 ChannelPromise setFailure(Throwable cause);  
  
 @Override  
 ChannelPromise addListener(GenericFutureListener<? **extends** Future<? **super** Void>> listener);  
  
 @Override  
 ChannelPromise addListeners(GenericFutureListener<? **extends** Future<? **super** Void>>... listeners);  
  
 @Override  
 ChannelPromise removeListener(GenericFutureListener<? **extends** Future<? **super** Void>> listener);  
  
 @Override  
 ChannelPromise removeListeners(GenericFutureListener<? **extends** Future<? **super** Void>>... listeners);  
  
 @Override  
 ChannelPromise sync() **throws** InterruptedException;  
  
 @Override  
 ChannelPromise syncUninterruptibly();  
  
 @Override  
 ChannelPromise await() **throws** InterruptedException;  
  
 @Override  
 ChannelPromise awaitUninterruptibly();  
ChannelPromise unvoid();  
}

**DefaultChannelPromise是ChannelPromise的实现类，它是实际运行时的Promoise实例。Channel接口提供了newPromise接口，表示Channel要创建一个异步执行的动作**

**public interface** ChannelOutboundInvoker {

ChannelPromise newPromise();

}

**public interface** Channel **extends** AttributeMap, ChannelOutboundInvoker, Comparable<Channel> {

}

**public abstract class** AbstractChannel **extends** DefaultAttributeMap **implements** Channel {

@Override  
**public** ChannelPromise newPromise() {  
 **return pipeline**.newPromise();  
}

}

**看一下DefaultPromise的addListener方法，它判断异步任务执行的状态，如果执行完成，就立即通知监听者，否则加入到监听者队列**

**通知监听者就是找一个线程来执行调用监听的回调函数。**

// DefaultPromise.addListener

@Override  
**public** Promise<V> addListener(GenericFutureListener<? **extends** Future<? **super** V>> listener) {  
 *checkNotNull*(listener, **"listener"**);  
  
 **synchronized** (**this**) {  
 **addListener0**(listener);  
 }  
  
 **if** (isDone()) {//如果执行完成就立即通知监听者  
 **notifyListeners**();  
 }  
  
 **return this**;  
}

**private void** addListener0(GenericFutureListener<? **extends** Future<? **super** V>> listener) {  
 **if** (**listeners** == **null**) {  
 **listeners** = listener;  
 } **else if** (**listeners instanceof** DefaultFutureListeners) {  
 ((DefaultFutureListeners) **listeners**).add(listener);  
 } **else** {  
 **listeners** = **new** DefaultFutureListeners((GenericFutureListener<?>) **listeners**, listener);  
 }  
}

**private void** notifyListeners() {  
 EventExecutor executor = executor();  
 **if** (executor.inEventLoop()) {  
 **final** InternalThreadLocalMap threadLocals = InternalThreadLocalMap.*get*();  
 **final int** stackDepth = threadLocals.futureListenerStackDepth();  
 **if** (stackDepth < ***MAX\_LISTENER\_STACK\_DEPTH***) {  
 threadLocals.setFutureListenerStackDepth(stackDepth + 1);  
 **try** {  
 **notifyListenersNow**();  
 } **finally** {  
 threadLocals.setFutureListenerStackDepth(stackDepth);  
 }  
 **return**;  
 }  
 }  
  
 *safeExecute*(executor, **new** Runnable() {  
 @Override  
 **public void** run() {  
 notifyListenersNow();  
 }  
 });  
}

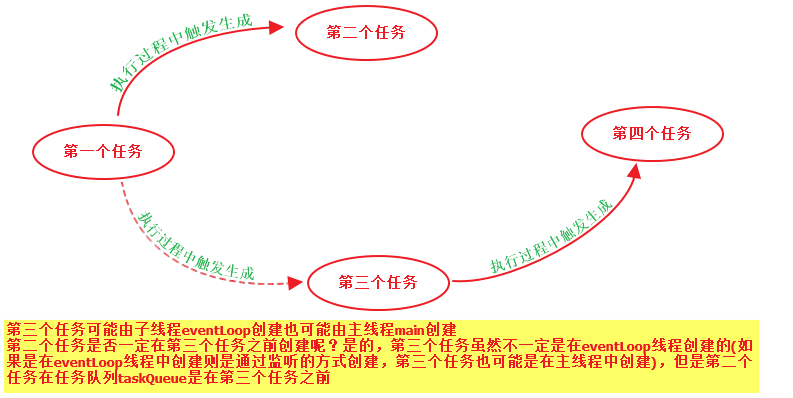
**private void** notifyListenersNow() {  
 Object listeners;  
 **synchronized** (**this**) {  
 *// Only proceed if there are listeners to notify and we are not already notifying listeners.* **if** (**notifyingListeners** || **this**.**listeners** == **null**) {  
 **return**;  
 }  
 **notifyingListeners** = **true**;  
 listeners = **this**.**listeners**;  
 **this**.**listeners** = **null**;  
 }  
 **for** (;;) {  
 **if** (listeners **instanceof** DefaultFutureListeners) {  
 notifyListeners0((DefaultFutureListeners) listeners);  
 } **else** {  
 ***notifyListener0***(**this**, (GenericFutureListener<?>) listeners);  
 }  
 **synchronized** (**this**) {  
 **if** (**this**.**listeners** == **null**) {  
 *// Nothing can throw from within this method, so setting notifyingListeners back to false does not  
 // need to be in a finally block.* **notifyingListeners** = **false**;  
 **return**;  
 }  
 listeners = **this**.**listeners**;  
 **this**.**listeners** = **null**;  
 }  
 }  
}

@SuppressWarnings({ **"unchecked"**, **"rawtypes"** })  
**private static void** notifyListener0(Future future, GenericFutureListener l) {  
 **try** {  
 l.operationComplete(future);  
 } **catch** (Throwable t) {  
 ***logger***.warn(**"An exception was thrown by "** + l.getClass().getName() + **".operationComplete()"**, t);  
 }  
}

**再来看监听者的接口，就一个方法，即等异步任务执行完成后，拿到Future结果，执行回调的逻辑**

**public interface** GenericFutureListener<F **extends** Future<?>> **extends** EventListener {  
**void** operationComplete(F future) **throws** Exception;  
}

# 任务之间的关系(待完善)



# 术语解释

channel : NioServerSocketChannel，NioSocketChannel 每个Channel都会有一个独立的pipeline,

在NioServerSocketChannel创建的时候一起创建ServerSocketChannel

promise : DefaultChannelPromise

指定未来要执行的操作，Promise里面有个result字段，该字段可取值null->

***UNCANCELLABLE ->SUCCESS,当Promise的result字段为SUCCESS的时候意味着某个动作已经完成可以开始下一步操作。如果Promise的result字段不为SUCCESS则***比如一个操作A完成之后通知操作B执行

pipeline: DefaultChannelPipeline

eventLoop： 比如NioEventLoop,在eventLoop创建的时候创建了Selector

ChannelHandlerContext: channelHandler执行的上下文

每创建一个Channel都是创建一个与之关联的pipeline,

# TaskQueue第一个任务详解

## register0(ChannelPromise promise)

**private void** register0(ChannelPromise promise) {  
 **try** {  
 *// check if the channel is still open as it could be closed in the mean time when the register  
 // call was outside of the eventLoop*

***//1.检查channel是否仍然打开，因为channel可以在register()位***

***//于eventLoop之外调用的时候同时关闭*****if** (!**promise.setUncancellable()** || !ensureOpen(promise)) {  
 **return**;  
 }  
 **boolean** firstRegistration = **neverRegistered**;  
 **doRegister();**  
 **neverRegistered** = **false**;

**//ServerSocketChannel已经注册到Selector** **registered** = **true**;

***//确保我们在通知promise之前调用handlerAdded(...)，这是必须的，因为***

***//用户可能已经通过在ChannelFutureListener中的pipeline触发了事件*****pipeline.invokeHandlerAddedIfNeeded();**  
 **safeSetSuccess(promise);** pipeline.fireChannelRegistered();  
 *// Only fire a channelActive if the channel has never been registered. This prevents firing  
 // multiple channel actives if the channel is deregistered and re-registered.*

*//*

***//注意，isActive()在NioServerSocketChannel里面的实现与在***

***//NioSocketChannel里面的实现不同*****if** (**isActive()**) {  
 **if** (firstRegistration) {

**//注册OP\_ACCEPT或OP\_READ事件**

**//1.在给ServerSocketChannel注册OP\_ACCEPT事件的时候可能在此**

**//步骤完成，也可能在task任务中完成的**

**//2.在给SocketChannel注册OP\_READ事件的时候是在此步骤完成的** **pipeline**.fireChannelActive();  
 } **else if** (config().isAutoRead()) {  
 *// This channel was registered before and autoRead() is set. This means we need to begin read  
 // again so that we process inbound data.  
 //  
 // See https://github.com/netty/netty/issues/4805* beginRead();  
 }  
 }  
 } **catch** (Throwable t) {  
 *// Close the channel directly to avoid FD leak.* closeForcibly();  
 **closeFuture**.setClosed();  
 safeSetFailure(promise, t);  
 }  
}

### promise.setUncancellable()

@Override  
**public boolean** setUncancellable() {  
 **if** (***RESULT\_UPDATER***.compareAndSet(**this**, **null**, ***UNCANCELLABLE***)) {  
 **return true**;  
 }  
 Object result = **this**.**result**;  
 **return** !*isDone0*(result) || !*isCancelled0*(result);  
}

### ensureOpen(promise)

**protected final boolean** ensureOpen(ChannelPromise promise) {  
 **if** (isOpen()) {  
 **return true**;  
 }  
  
 safeSetFailure(promise, ***ENSURE\_OPEN\_CLOSED\_CHANNEL\_EXCEPTION***);  
 **return false**;  
}

@Override  
**public boolean** isOpen() {  
 **return ch**.isOpen();  
}



### doRegister();

@Override  
**protected void** doRegister() **throws** Exception {  
 **boolean** selected = **false**;  
 **for** (;;) {  
 **try** {  
 **selectionKey** = javaChannel().register(eventLoop().unwrappedSelector(), 0, **this**);  
 **return**;  
 } **catch** (CancelledKeyException e) {  
 **if** (!selected) {  
 *// Force the Selector to select now as the "canceled" SelectionKey may still be  
 // cached and not removed because no Select.select(..) operation was called yet.* eventLoop().selectNow();  
 selected = **true**;  
 } **else** {  
 *// We forced a select operation on the selector before but the SelectionKey is still cached  
 // for whatever reason. JDK bug ?* **throw** e;  
 }  
 }  
 }  
}

### pipeline.invokeHandlerAddedIfNeeded();触发生成第二个任务

**final void** invokeHandlerAddedIfNeeded() {

**//判断是不是在eventLoop线程中**  
 **assert channel**.eventLoop().inEventLoop();

**//在pipeline初始化的时候firstRegistration值初始化为true，在第一次调用**

**//该方法后设置成false** **if** (**firstRegistration**) {  
 **firstRegistration** = **false**;  
 **//我们现在注册到EventLoop。是时候调用ChannelHandler的callbacks**

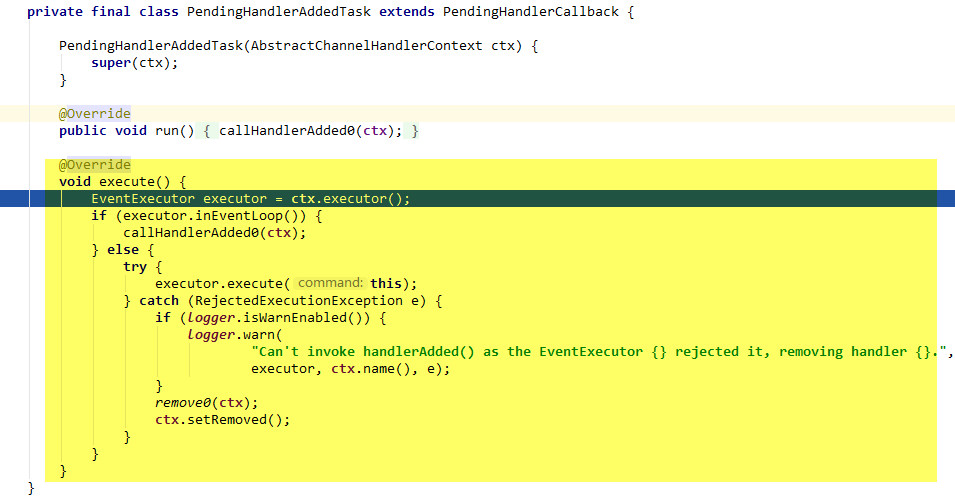
***//回调方法，这些回调是在register完成之前添加的*****callHandlerAddedForAllHandlers**();  
 }  
}

**private void** callHandlerAddedForAllHandlers() {  
 **final** PendingHandlerCallback pendingHandlerCallbackHead;  
 **synchronized** (**this**) {  
 **assert** !**registered**;  
  
 *// This Channel itself was registered.*

***//该ServerSocketChannel已经register了*****registered** = **true**;  
  
 pendingHandlerCallbackHead = **this**.**pendingHandlerCallbackHead**;  
 *// Null out so it can be GC'ed.* **this**.**pendingHandlerCallbackHead** = **null**;  
 }  
  
 *// This must happen outside of the synchronized(...) block as otherwise handlerAdded(...) may be called while  
 // holding the lock and so produce a deadlock if handlerAdded(...) will try to add another handler from outside  
 // the EventLoop.* PendingHandlerCallback task = pendingHandlerCallbackHead;  
 **while** (task != **null**) {  
  **task.execute();** task = task.**next**;  
 }  
}

@Override  
**void execute()** {  
 EventExecutor executor = **ctx**.executor();

//判断是否在eventLoop线程中  
 **if** (executor.inEventLoop()) {//在eventLoop线程中  
  **callHandlerAdded0(ctx);** } **else** {//不在eventLoop线程中  
 **try** {  
 executor.execute(**this**);  
 } **catch** (RejectedExecutionException e) {  
 **if** (***logger***.isWarnEnabled()) {  
 ***logger***.warn(  
 **"Can't invoke handlerAdded() as the EventExecutor {} rejected it, removing handler {}."**,  
 executor, **ctx**.name(), e);  
 }  
 *remove0*(**ctx**);  
 **ctx**.setRemoved();  
 }  
 }  
}



**private void** callHandlerAdded0(**final** AbstractChannelHandlerContext ctx) {  
 **try** {  
 *// We must call setAddComplete before calling handlerAdded. Otherwise if the handlerAdded method generates  
 // any pipeline events ctx.handler() will miss them because the state will not allow it.*

**//设置ChannelHandlerContext的处理状态handlerState=*ADD\_COMPLETE***ctx.setAddComplete();

//1.ctx.handler()得到ChannelHandler

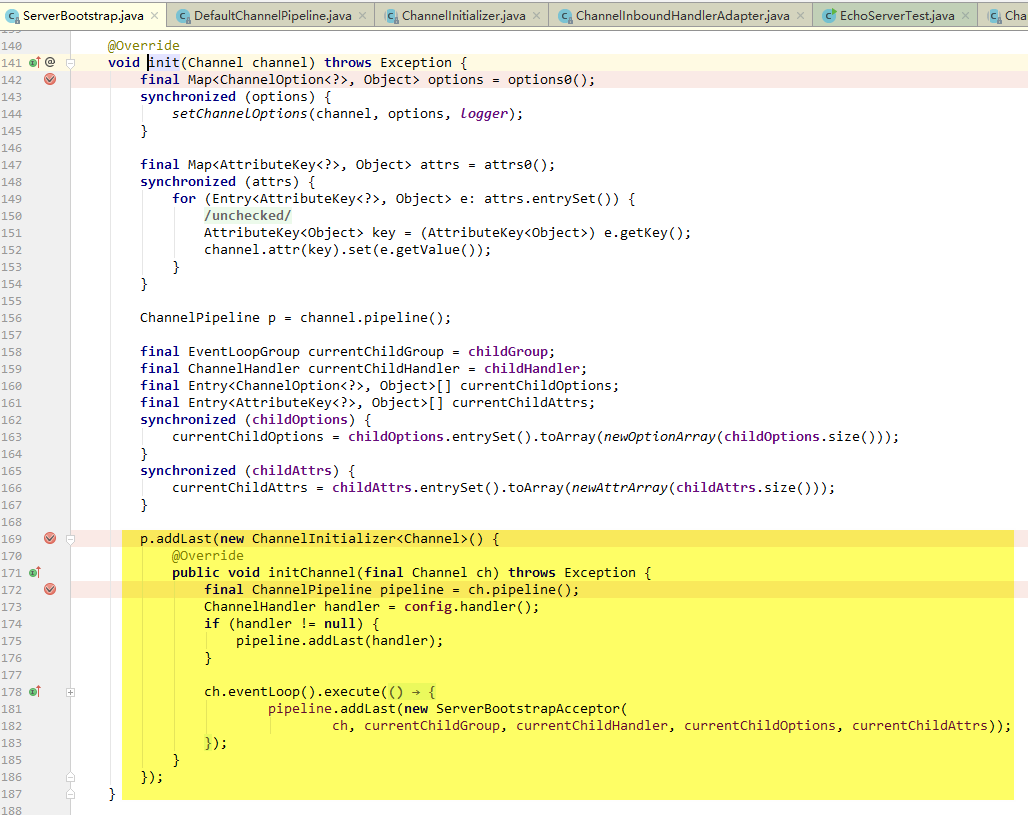
//2.ChannelHandler.handlerAdded()  
 ctx.handler().handlerAdded(ctx);  
 } **catch** (Throwable t) {  
 **boolean** removed = **false**;  
 **try** {  
 *remove0*(ctx);  
 **try** {  
 ctx.handler().handlerRemoved(ctx);  
 } **finally** {  
 ctx.setRemoved();  
 }  
 removed = **true**;  
 } **catch** (Throwable t2) {  
 **if** (***logger***.isWarnEnabled()) {  
 ***logger***.warn(**"Failed to remove a handler: "** + ctx.name(), t2);  
 }  
 }  
  
 **if** (removed) {  
 fireExceptionCaught(**new** ChannelPipelineException(  
 ctx.handler().getClass().getName() +  
 **".handlerAdded() has thrown an exception; removed."**, t));  
 } **else** {  
 fireExceptionCaught(**new** ChannelPipelineException(  
 ctx.handler().getClass().getName() +  
 **".handlerAdded() has thrown an exception; also failed to remove."**, t));  
 }  
 }  
}

#### setAddComplete

**final void** setAddComplete() {  
 **for** (;;) {  
 **int** oldState = **handlerState**;  
 *// Ensure we never update when the handlerState is REMOVE\_COMPLETE already.  
 // oldState is usually ADD\_PENDING but can also be REMOVE\_COMPLETE when an EventExecutor is used that is not  
 // exposing ordering guarantees.* **if** (oldState == ***REMOVE\_COMPLETE*** || ***HANDLER\_STATE\_UPDATER***.compareAndSet(**this**, oldState, ***ADD\_COMPLETE***)) {  
 **return**;  
 }  
 }  
}

#### ctx.handler().handlerAdded(ctx);

说明：ctx.handle()得到的就是对NioServerSocketChannel进行配置时的时候添加的ChannelInitializer



@Override  
**public void** handlerAdded(ChannelHandlerContext ctx) **throws** Exception {  
 **if** (ctx.channel().isRegistered()) {  
 *// This should always be true with our current DefaultChannelPipeline implementation.  
 // The good thing about calling initChannel(...) in handlerAdded(...) is that there will be no ordering  
 // surprises if a ChannelInitializer will add another ChannelInitializer. This is as all handlers  
 // will be added in the expected order.* initChannel(ctx);  
 }  
}

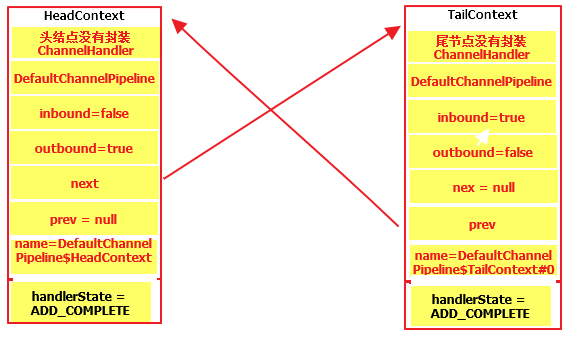
@SuppressWarnings(**"unchecked"**)  
**private boolean** initChannel(ChannelHandlerContext ctx) **throws** Exception {  
 **if** (**initMap**.putIfAbsent(ctx, Boolean.***TRUE***) == **null**) { *// Guard against re-entrance.* **try** {

**//设置NioServerSocketChannel** **initChannel((C) ctx.channel());**  
 } **catch** (Throwable cause) {  
 *// Explicitly call exceptionCaught(...) as we removed the handler before calling initChannel(...).  
 // We do so to prevent multiple calls to initChannel(...).* exceptionCaught(ctx, cause);  
 } **finally** {

**//从链表中删除ChannelHandlerContext** remove(ctx);  
 }  
 **return true**;  
 }  
 **return false**;  
}

**private void** remove(ChannelHandlerContext ctx) {  
 **try** {  
 ChannelPipeline pipeline = ctx.pipeline();  
 **if** (pipeline.context(**this**) != **null**) {  
 pipeline.remove(**this**);  
 }  
 } **finally** {  
 **initMap**.remove(ctx);  
 }  
 }  
}

#### pipeline链表结构



@Override  
**public void** initChannel(**final** Channel ch) **throws** Exception {  
 **final** ChannelPipeline pipeline = ch.pipeline();  
 ChannelHandler handler = **config**.handler();  
 **if** (handler != **null**) {  
 pipeline.addLast(handler);  
 }  
  **//eventLoop.execute()添加一个任务到任务队列taskQueue** ch.eventLoop().execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 pipeline.addLast(**new** ServerBootstrapAcceptor(  
 ch, currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));  
 }  
 });  
}

### safeSetSuccess在这里可能触发生成第三个任务

**protected final void** safeSetSuccess(ChannelPromise promise) {  
 **if** (!(promise **instanceof** VoidChannelPromise) && !**promise.trySuccess()**) {  
 ***logger***.warn(**"Failed to mark a promise as success because it is done already: {}"**, promise);  
 }  
}

@Override  
**public boolean** trySuccess() {  
 **return** trySuccess(**null**);  
}

@Override  
**public boolean** trySuccess(V result) {

//设置promise为SUCCESS意味着这个promise执行完成，

//如果promise有绑定鉴定器则可以通知相关监听器执行了(监听器模式)  
 **if** (**setSuccess0(result)**) {  
  **notifyListeners();//通知监听器**  
 **return true**;  
 }  
 **return false**;  
}

**private boolean** setSuccess0(V result) {  
 **return** setValue0(result == **null** ? ***SUCCESS*** : result);  
}

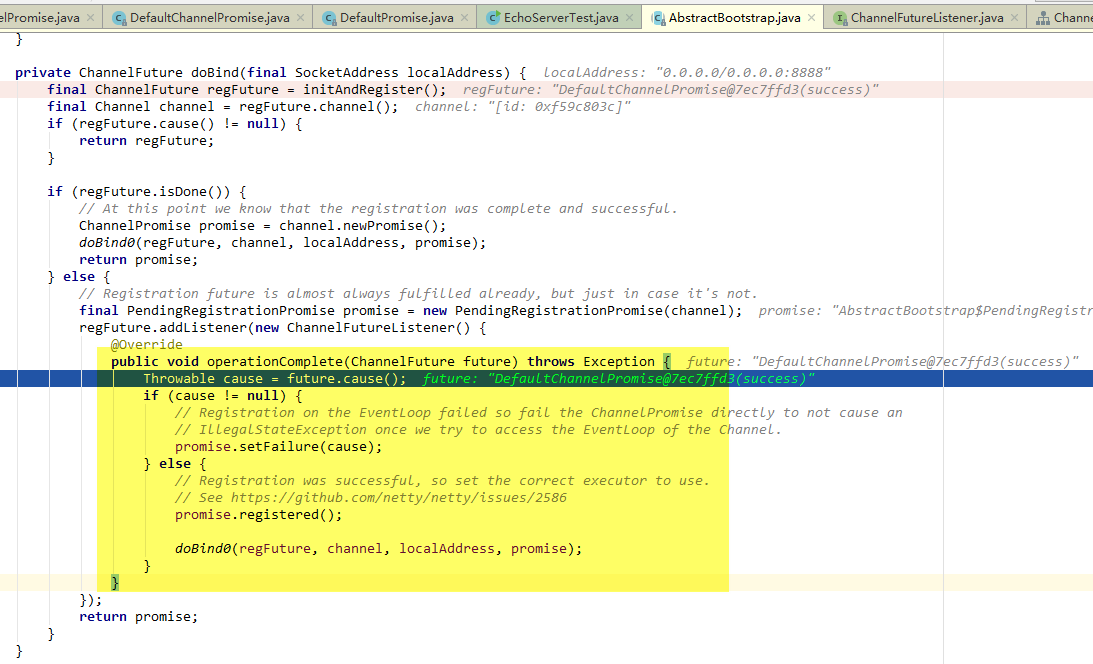
**private boolean** setValue0(Object objResult) {  
 **if** (***RESULT\_UPDATER***.compareAndSet(**this**, **null**, objResult) ||  
 ***RESULT\_UPDATER***.compareAndSet(**this**, ***UNCANCELLABLE***, objResult)) {  
 checkNotifyWaiters();  
 **return true**;  
 }  
 **return false**;  
}

**private void** notifyListeners() {  
 EventExecutor executor = executor();  
 **if** (executor.inEventLoop()) {  
 **final** InternalThreadLocalMap threadLocals = InternalThreadLocalMap.*get*();  
 **final int** stackDepth = threadLocals.futureListenerStackDepth();  
 **if** (stackDepth < ***MAX\_LISTENER\_STACK\_DEPTH***) {  
 threadLocals.setFutureListenerStackDepth(stackDepth + 1);  
 **try** {  
 **notifyListenersNow();** } **finally** {  
 threadLocals.setFutureListenerStackDepth(stackDepth);  
 }  
 **return**;  
 }  
 }  
  
 *safeExecute*(executor, **new** Runnable() {  
 @Override  
 **public void** run() {  
 **notifyListenersNow();** }  
 });  
}

**private void** notifyListenersNow() {  
 Object listeners;  
 **synchronized** (**this**) {  
 *// Only proceed if there are listeners to notify and we are not already notifying listeners.* **if** (**notifyingListeners** || **this**.**listeners** == **null**) {  
 **return**;  
 }  
 **notifyingListeners** = **true**;  
 listeners = **this**.**listeners**;  
 **this**.**listeners** = **null**;  
 }  
 **for** (;;) {  
 **if** (listeners **instanceof** DefaultFutureListeners) {  
 notifyListeners0((DefaultFutureListeners) listeners);  
 } **else** {  
 ***notifyListener0*(this, (GenericFutureListener<?>) listeners);** }  
 **synchronized** (**this**) {  
 **if** (**this**.**listeners** == **null**) {  
 *// Nothing can throw from within this method, so setting notifyingListeners back to false does not  
 // need to be in a finally block.* **notifyingListeners** = **false**;  
 **return**;  
 }  
 listeners = **this**.**listeners**;  
 **this**.**listeners** = **null**;  
 }  
 }  
}

@SuppressWarnings({ **"unchecked"**, **"rawtypes"** })  
**private static void** notifyListener0(Future future, GenericFutureListener l) {  
 **try** {  
 **l.operationComplete(future);**  
 } **catch** (Throwable t) {  
 ***logger***.warn(**"An exception was thrown by "** + l.getClass().getName() + **".operationComplete()"**, t);  
 }  
}

**operationComplete(DefaultChannelPromise)执行的就是之前创建的ChannelFutureListener匿名类对象**



### pipeline.fireChannelRegistered()什么操作都没做

@Override  
**public final** ChannelPipeline fireChannelRegistered() {  
 AbstractChannelHandlerContext.*invokeChannelRegistered*(**head**);  
 **return this**;  
}

**static void** invokeChannelRegistered(**final** AbstractChannelHandlerContext next) {

//如果链表头节点executor为空则返回eventLoop  
 EventExecutor executor = next.executor();  
 **if** (executor.inEventLoop()) {  
 next.**invokeChannelRegistered**();  
 } **else** {  
 executor.execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 next.invokeChannelRegistered();  
 }  
 });  
 }  
}

**private void** invokeChannelRegistered() {  
 **if** (invokeHandler()) {  
 **try** {

//head节点ChannelHandlerContextInbound  
 ((ChannelInboundHandler) handler()).channelRegistered(**this**);  
 } **catch** (Throwable t) {  
 notifyHandlerException(t);  
 }  
 } **else** {  
 fireChannelRegistered();  
 }  
}

@Override  
**public void** channelRegistered(ChannelHandlerContext ctx) **throws** Exception {  
 invokeHandlerAddedIfNeeded();  
 ctx.**fireChannelRegistered**();  
}

#### invokeHandlerAddedIfNeeded

**final void** invokeHandlerAddedIfNeeded() {  
 **assert channel**.eventLoop().inEventLoop();  
 **if** (**firstRegistration**) {  
 **firstRegistration** = **false**;  
 *// We are now registered to the EventLoop. It's time to call the callbacks for the ChannelHandlers,  
 // that were added before the registration was done.* callHandlerAddedForAllHandlers();  
 }  
}

#### fireChannelRegistered

@Override  
**public** ChannelHandlerContext fireChannelRegistered() {

//1.findContextInbound()找到下一个ChannelHandlerContextInbound,在这里

//是链表尾节点tail  
 *invokeChannelRegistered*(findContextInbound());  
 **return this**;  
}

**static void** invokeChannelRegistered(**final** AbstractChannelHandlerContext next) {

//如果tail的executor为空则返回eventLoop  
 EventExecutor executor = next.executor();  
 **if** (executor.inEventLoop()) {  
 next.invokeChannelRegistered();  
 } **else** {  
 executor.execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 next.invokeChannelRegistered();  
 }  
 });  
 }  
}

**private void** invokeChannelRegistered() {  
 **if** (invokeHandler()) {  
 **try** {

//在这个Handler()返回的是tail,this也是tail  
 ((ChannelInboundHandler) handler()).channelRegistered(**this**);  
 } **catch** (Throwable t) {  
 notifyHandlerException(t);  
 }  
 } **else** {  
 fireChannelRegistered();  
 }  
}

//由下列代码可知链表的tail节点空实现

@Override  
**public void** channelRegistered(ChannelHandlerContext ctx) **throws** Exception { }

### isActive()

@Override  
**public boolean** isActive() {

**//isBound()判断ServerSocketChannel是否绑定到端口上，如果已绑定返回true** **return** javaChannel().socket().isBound();  
}

# TaskQueue第二个任务详解

注意：第一个任务执行完了才会执行第二个任务

ch.eventLoop().execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 pipeline.addLast(**new** ServerBootstrapAcceptor(  
 ch, currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));  
 }  
 });

### ServerBootstrapAcceptor初始化

ServerBootstrapAcceptor(  
 **final** Channel channel, EventLoopGroup childGroup, ChannelHandler childHandler,  
 Entry<ChannelOption<?>, Object>[] childOptions, Entry<AttributeKey<?>, Object>[] childAttrs) {  
 **this**.**childGroup** = childGroup;  
 **this**.**childHandler** = childHandler;  
 **this**.**childOptions** = childOptions;  
 **this**.**childAttrs** = childAttrs;  
  
 *// Task which is scheduled to re-enable auto-read.  
 // It's important to create this Runnable before we try to submit it as otherwise the URLClassLoader may  
 // not be able to load the class because of the file limit it already reached.  
 //  
 // See https://github.com/netty/netty/issues/1328* **enableAutoReadTask** = **new** Runnable() {  
 @Override  
 **public void** run() {  
 channel.config().setAutoRead(**true**);  
 }  
 };  
}

@Override  
**public final** ChannelPipeline addLast(ChannelHandler... handlers) {  
 **return** addLast(**null**, handlers);  
}

@Override  
**public final** ChannelPipeline addLast(EventExecutorGroup executor, ChannelHandler... handlers) {  
 **if** (handlers == **null**) {  
 **throw new** NullPointerException(**"handlers"**);  
 }  
  
 **for** (ChannelHandler h: handlers) {  
 **if** (h == **null**) {  
 **break**;  
 }  
 addLast(executor, **null**, h);  
 }  
  
 **return this**;  
}

**将ServerBootstrapAcceptor添加到链表的过程与将ChannelInitializer添加到链表的过程不同，将ServerBootstrapAcceptor添加链表时ServerSocketChannel已经注册到了selector，因此registered标识为true。而将ChannelInitializer添加到链表时ServerSocketChannel并没有注册到selector,此时registered标识为false**

@Override  
**public final** ChannelPipeline addLast(EventExecutorGroup group, String name, ChannelHandler handler) {  
 **final** AbstractChannelHandlerContext newCtx;  
 **synchronized** (**this**) {  
 ***checkMultiplicity*(handler);**  
 newCtx = newContext(group, filterName(name, handler), handler);  
  
 addLast0(newCtx);  
  
 *// If the registered is false it means that the channel was not registered on an eventloop yet.  
 // In this case we add the context to the pipeline and add a task that will call  
 // ChannelHandler.handlerAdded(...) once the channel is registered.* **if** (!**registered**) {  
 newCtx.setAddPending();  
 callHandlerCallbackLater(newCtx, **true**);  
 **return this**;  
 }  
  
 EventExecutor executor = newCtx.executor();//得到NioEventLoop  
 **if** (!executor.inEventLoop()) {//  
 newCtx.setAddPending();  
 executor.execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 callHandlerAdded0(newCtx);  
 }  
 });  
 **return this**;  
 }  
 }  
 callHandlerAdded0(newCtx);  
 **return this**;  
}

#### checkMultiplicity

**private static void** checkMultiplicity(ChannelHandler handler) {  
 **if** (handler **instanceof** ChannelHandlerAdapter) {  
 ChannelHandlerAdapter h = (ChannelHandlerAdapter) handler;

**//1.判断这个ChannelHandler上有没有添加注解@Sharable**

**//2.h.added判断该ChannelHandler之前有没有添加到pipeline，如果之前**

**//已经添加到过pipeline,则h.added为true**

**//3.如果这个ChannnelHandler上有添加则说明这ChannelHandler的相同实**

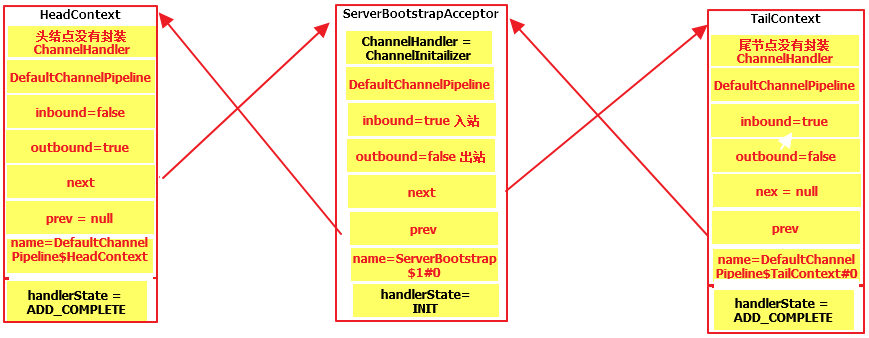
**//例多次添加到一个或多个ChannelPipeline中，而不存在竞争条件。**  
 **if** (!h.isSharable() && h.**added**) {  
 **throw new** ChannelPipelineException(  
 h.getClass().getName() +  
 **" is not a @Sharable handler, so can't be added or removed multiple times."**);  
 }  
 h.**added** = **true**;  
 }  
}

#### pipeline链表结构

***说明：封装ServerBootstrapAcceptor的ChannelHandlerContext的处理状态handerState是直接从INIT变为ADD\_COMPLETE的。***

***而封装ChannelInitializer的ChannelHandlerContext的处理状态handlerState***

***是INIT -> ADD\_PENDING -> ADD\_COMPLETE***



#### callHandlerAdded0

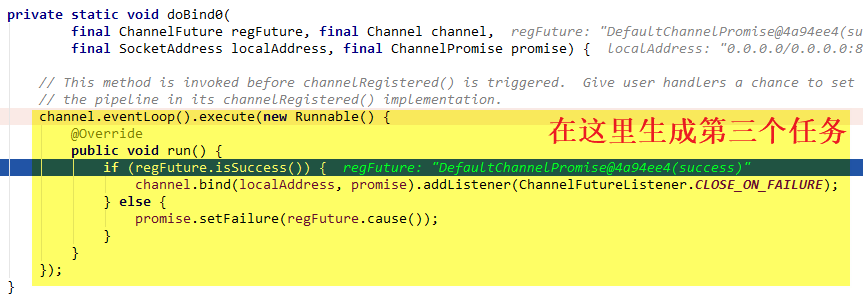
**private void** callHandlerAdded0(**final** AbstractChannelHandlerContext ctx) {  
 **try** {  
 *// We must call setAddComplete before calling handlerAdded. Otherwise if the handlerAdded method generates  
 // any pipeline events ctx.handler() will miss them because the state will not allow it.* **ctx.setAddComplete();**

**//1.ctx.handler()返回*ServerBootstrapAcceptor***

**//2.由于*ServerBootstrapAcceptor的handlerAdded()空实现，所以这里***

**//没有逻辑  
 ctx.handler().handlerAdded(ctx);**  
 } **catch** (Throwable t) {  
 **boolean** removed = **false**;  
 **try** {  
 *remove0*(ctx);  
 **try** {  
 ctx.handler().handlerRemoved(ctx);  
 } **finally** {  
 ctx.setRemoved();  
 }  
 removed = **true**;  
 } **catch** (Throwable t2) {  
 **if** (***logger***.isWarnEnabled()) {  
 ***logger***.warn(**"Failed to remove a handler: "** + ctx.name(), t2);  
 }  
 }  
  
 **if** (removed) {  
 fireExceptionCaught(**new** ChannelPipelineException(  
 ctx.handler().getClass().getName() +  
 **".handlerAdded() has thrown an exception; removed."**, t));  
 } **else** {  
 fireExceptionCaught(**new** ChannelPipelineException(  
 ctx.handler().getClass().getName() +  
 **".handlerAdded() has thrown an exception; also failed to remove."**, t));  
 }  
 }  
}

# TaskQueue第三个任务详解



channel.eventLoop().execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **if** (**regFuture.isSuccess()**) {  
 **channel.bind(localAddress, promise)**.addListener(ChannelFutureListener.***CLOSE\_ON\_FAILURE***);  
 } **else** {  
 promise.setFailure(regFuture.cause());  
 }  
 }  
});

@Override  
**public boolean** isSuccess() {  
 Object result = **this**.**result**;

**//UNCANCELLABLE就是Object对象**  
 **return** result != **null** && result != ***UNCANCELLABLE*** && !(result **instanceof** CauseHolder);  
}

@Override  
**public** ChannelFuture bind(SocketAddress localAddress, ChannelPromise promise) {  
 **return pipeline**.bind(localAddress, promise);  
}

### pipeline.bind

@Override  
**public final** ChannelFuture bind(SocketAddress localAddress, ChannelPromise promise) {  
 **return tail**.bind(localAddress, promise);//链表尾节点tail  
}

@Override  
**public** ChannelFuture bind(**final** SocketAddress localAddress, **final** ChannelPromise promise) {

//ServerSocketChannel要绑定的本地地址  
 **if** (localAddress == **null**) {  
 **throw new** NullPointerException(**"localAddress"**);  
 }

//校验是不是有效的promise  
 **if** (**isNotValidPromise**(promise, **false**)) {  
 *// cancelled* **return** promise;  
 }  
  **//从链表尾节点tail向前寻找ChannelHandlerContextOutbound,**

**//总是能够找到，因为链表的头结点HeadChannelHandlerContextOutbound**  
 **final** AbstractChannelHandlerContext next = findContextOutbound();  
  
 **final** AbstractChannelHandlerContext next = **findContextOutbound();** EventExecutor executor = next.executor();  
 **if** (executor.inEventLoop()) {

//ChannelHandlerContextOutbound调用invokeBind  
  **next.invokeBind(localAddress, promise);** } **else** {  
 *safeExecute*(executor, **new** Runnable() {  
 @Override  
 **public void** run() {  
 next.invokeBind(localAddress, promise);  
 }  
 }, promise, **null**);  
 }  
 **return** promise;  
}

#### isNotValidPromise

**private boolean** isNotValidPromise(ChannelPromise promise, **boolean** allowVoidPromise) {  
 **if** (promise == **null**) {  
 **throw new** NullPointerException(**"promise"**);  
 }  
 //如果promise的result不为null并且不为***UNCANCELLABLE则promise.isDone()***

//返回true  
 **if** (promise.isDone()) {  
 *// Check if the promise was cancelled and if so signal that the processing of the operation  
 // should not be performed.  
 //  
 // See https://github.com/netty/netty/issues/2349* **if** (promise.isCancelled()) {  
 **return true**;  
 }  
 **throw new** IllegalArgumentException(**"promise already done: "** + promise);  
 }  
  
 **if** (promise.channel() != channel()) {  
 **throw new** IllegalArgumentException(String.*format*(  
 **"promise.channel does not match: %s (expected: %s)"**, promise.channel(), channel()));  
 }  
  
 **if** (promise.getClass() == DefaultChannelPromise.**class**) {  
 **return false**;  
 }  
  
 **if** (!allowVoidPromise && promise **instanceof** VoidChannelPromise) {  
 **throw new** IllegalArgumentException(  
 StringUtil.*simpleClassName*(VoidChannelPromise.**class**) + **" not allowed for this operation"**);  
 }  
  
 **if** (promise **instanceof** AbstractChannel.CloseFuture) {  
 **throw new** IllegalArgumentException(  
 StringUtil.*simpleClassName*(AbstractChannel.CloseFuture.**class**) + **" not allowed in a pipeline"**);  
 }  
 **return false**;  
}

#### findContextOutbound

**private** AbstractChannelHandlerContext findContextOutbound() {  
 AbstractChannelHandlerContext ctx = **this**;**//this是尾节点**

**//向前寻找ContextOutbound,链表头节点是ContextOutbound，尾节点是**

**//ContextInbound** **do** {  
 ctx = ctx.**prev**;  
 } **while** (!ctx.**outbound**);  
 **return** ctx;  
}

#### next.executor()

@Override  
**public** EventExecutor executor() {

//头结点head的exector为空，则返回eventLoop  
 **if** (**executor** == **null**) {  
 **return** channel().eventLoop();  
 } **else** {  
 **return executor**;  
 }  
}

#### invokeBind

**private void** invokeBind(SocketAddress localAddress, ChannelPromise promise) {  
 **if** (invokeHandler()) {  
 **try** {

//链表头节点是ChannelOutboundHandler  
 ((ChannelOutboundHandler) handler()).bind(**this**, localAddress, promise);  
 } **catch** (Throwable t) {  
 *notifyOutboundHandlerException*(t, promise);  
 }  
 } **else** {  
 bind(localAddress, promise);  
 }  
}

@Override  
**public void** bind(  
 ChannelHandlerContext ctx, SocketAddress localAddress, ChannelPromise promise)  
 **throws** Exception {  
 ***unsafe.bind***(localAddress, promise);  
}

##### unsafe.bind

@Override  
**public final void** bind(**final** SocketAddress localAddress, **final** ChannelPromise promise) {  
 assertEventLoop();  
  
 **if** (!promise.setUncancellable() || !ensureOpen(promise)) {  
 **return**;  
 }  
  
 *// See: https://github.com/netty/netty/issues/576* **if** (Boolean.***TRUE***.equals(config().getOption(ChannelOption.***SO\_BROADCAST***)) &&  
 localAddress **instanceof** InetSocketAddress &&  
 !((InetSocketAddress) localAddress).getAddress().isAnyLocalAddress() &&  
 !PlatformDependent.*isWindows*() && !PlatformDependent.*maybeSuperUser*()) {  
 *// Warn a user about the fact that a non-root user can't receive a  
 // broadcast packet on \*nix if the socket is bound on non-wildcard address.* ***logger***.warn(  
 **"A non-root user can't receive a broadcast packet if the socket "** +  
 **"is not bound to a wildcard address; binding to a non-wildcard "** +  
 **"address ("** + localAddress + **") anyway as requested."**);  
 }  
 //判断ServerSocketChannel是否已经绑定端口，已绑定返回true  
 **boolean** wasActive = isActive();  
 **try** {  
 **doBind**(localAddress);  
 } **catch** (Throwable t) {  
 safeSetFailure(promise, t);  
 closeIfClosed();  
 **return**;  
 }

//设置ServerSocketChannel感兴趣的事件***OP\_ACCEPT***  
 **if** (!wasActive && isActive()) {

**//触发生成第四个任务**  
 **invokeLater**(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **pipeline**.fireChannelActive();  
 }  
 });  
 }  
  
 **safeSetSuccess**(promise);  
}



###### doBind

@Override  
**protected void** doBind(SocketAddress localAddress) **throws** Exception {

//如果JDK版本大于7  
 **if** (PlatformDependent.*javaVersion*() >= 7) {  
 javaChannel().bind(localAddress, **config**.getBacklog());  
 } **else** {  
 javaChannel().socket().bind(localAddress, **config**.getBacklog());  
 }  
}

###### invokeLater

**private void invokeLater**(Runnable task) {  
 **try** {  
 *// This method is used by outbound operation implementations to trigger an inbound event later.  
 // They do not trigger an inbound event immediately because an outbound operation might have been  
 // triggered by another inbound event handler method. If fired immediately, the call stack  
 // will look like this for example:  
 //  
 // handlerA.inboundBufferUpdated() - (1) an inbound handler method closes a connection.  
 // -> handlerA.ctx.close()  
 // -> channel.unsafe.close()  
 // -> handlerA.channelInactive() - (2) another inbound handler method called while in (1) yet  
 //  
 // which means the execution of two inbound handler methods of the same handler overlap undesirably.* eventLoop().execute(task);**//将该任务添加到taskQueue** } **catch** (RejectedExecutionException e) {  
 ***logger***.warn(**"Can't invoke task later as EventLoop rejected it"**, e);  
 }  
}

###### safeSetSuccess

*/\*\*  
 \* Marks the specified {****@code*** *promise} as success. If the {****@code*** *promise} is done already, log a message.  
 \*/***protected final void** safeSetSuccess(ChannelPromise promise) {  
 **if** (!(promise **instanceof** VoidChannelPromise) && !**promise.trySuccess()**) {  
 ***logger***.warn(**"Failed to mark a promise as success because it is done already: {}"**, promise);  
 }  
}

promise.trySuccess

@Override  
**public boolean** trySuccess() {  
 **return** trySuccess(**null**);  
}

@Override  
**public boolean** trySuccess(V result) {  
 **if** (**setSuccess0**(result)) {  
 notifyListeners();  
 **return true**;  
 }  
 **return false**;  
}

**private boolean** setSuccess0(V result) {  
 **return setValue0**(result == **null** ? ***SUCCESS*** : result);  
}

**private boolean** setValue0(Object objResult) {  
 **if** (***RESULT\_UPDATER***.compareAndSet(**this**, **null**, objResult) ||  
 ***RESULT\_UPDATER***.compareAndSet(**this**, ***UNCANCELLABLE***, objResult)) {  
 checkNotifyWaiters();  
 **return true**;  
 }  
 **return false**;  
}

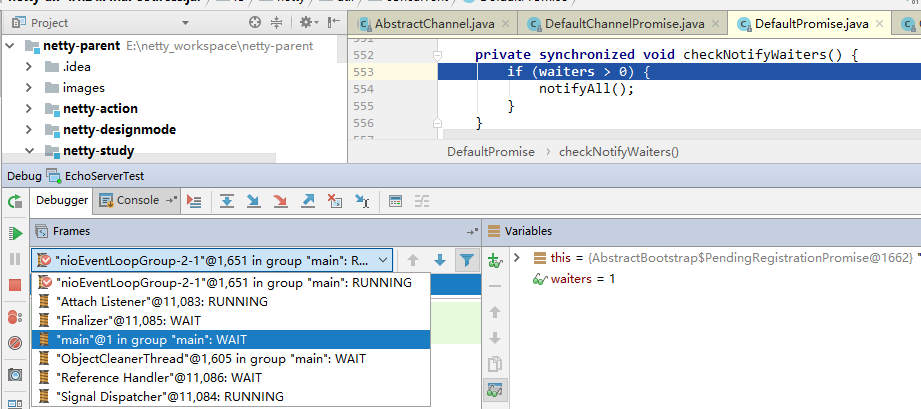
**private synchronized void** checkNotifyWaiters() {

**//waiters大于0，说明线程main或NioEventLoop其中至少有一个线程处于wait**

**//状态**

**if** (**waiters** > 0) {

**//如果线程main或NioEventLoopGroup-2-1处于wait状态，则唤醒** notifyAll();  
 }  
}



# TaskQueue第四个任务详解(OP\_ACCEP)

## invokeLater

**invokeLater**(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **pipeline**.fireChannelActive();  
 }  
 });

**private void** invokeLater(Runnable task) {  
 **try** {  
 *// This method is used by outbound operation implementations to trigger an inbound event later.  
 // They do not trigger an inbound event immediately because an outbound operation might have been  
 // triggered by another inbound event handler method. If fired immediately, the call stack  
 // will look like this for example:  
 //  
 // handlerA.inboundBufferUpdated() - (1) an inbound handler method closes a connection.  
 // -> handlerA.ctx.close()  
 // -> channel.unsafe.close()  
 // -> handlerA.channelInactive() - (2) another inbound handler method called while in (1) yet  
 //  
 // which means the execution of two inbound handler methods of the same handler overlap undesirably.* eventLoop().execute(task);  
 } **catch** (RejectedExecutionException e) {  
 ***logger***.warn(**"Can't invoke task later as EventLoop rejected it"**, e);  
 }  
}

### pipeline.fireChannelActive()

@Override  
**public final** ChannelPipeline fireChannelActive() {  
 AbstractChannelHandlerContext.*invokeChannelActive*(**head**);  
 **return this**;  
}

**static void** invokeChannelActive(**final** AbstractChannelHandlerContext next) {  
 EventExecutor executor = **next.executor()**;  
 **if** (executor.inEventLoop()) {  
 **next.invokeChannelActive();** } **else** {  
 executor.execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 next.invokeChannelActive();  
 }  
 });  
 }  
}

#### next.executor

@Override  
**public** EventExecutor executor() {

//头节点executor如果为空，则返回eventLoop  
 **if** (**executor** == **null**) {  
 **return** channel().eventLoop();  
 } **else** {  
 **return executor**;  
 }  
}

#### next.invokeChannelActive

**private void** invokeChannelActive() {  
 **if** (**invokeHandler()**) {  
 **try** {

**//hander()获取头节点head,头节点既实现了ChannelOutboundHandler**

**//也实现了ChannnelInboundHandler接口**  
 ((ChannelInboundHandler) handler()).**channelActive**(**this**);  
 } **catch** (Throwable t) {  
 notifyHandlerException(t);  
 }  
 } **else** {  
 fireChannelActive();  
 }  
}

##### invokeHandler()

**//1.尽最大努力检测是否调用了//ChannelHandler#handlerAdded(ChannelHandlerContext)。**

**//如果没有调用则返回false，如果调用或无法检测返回true。**

**//2.如果该方法返回false，我们将不会调用ChannelHandler，而只是转发事件。**

**//这是需要的，因为DefaultChannelPipeline可能已经将ChannelHandler放入**

**//链表中，但没有调用 ChannelHandler#handlerAdded(ChannelHandlerContext)。**

**private boolean** invokeHandler() {  
 *// 存储在局部变量中已减少 volatile读.* **int** handlerState = **this**.**handlerState**;  
 **return** handlerState == ***ADD\_COMPLETE*** || (!**ordered** && handlerState == ***ADD\_PENDING***);  
}

##### channelActive(this)

@Override  
**public void** channelActive(ChannelHandlerContext ctx) **throws** Exception {  
  **ctx.fireChannelActive();**  
 **readIfIsAutoRead**();  
}

###### fireChannelActive

@Override  
**public** ChannelHandlerContext fireChannelActive() {  
 *invokeChannelActive*(findContextInbound());  
 **return this**;  
}

findContextInbound()

**private** AbstractChannelHandlerContext findContextInbound() {  
 AbstractChannelHandlerContext ctx = **this**;

//从头节点向后遍历，获取第一个ChannelHandlerContextInbound,即是

//封装了ServerBootstrapAcceptor的ChannelHandlerContext  
 **do** {  
 ctx = ctx.**next**;  
 } **while** (!ctx.**inbound**);  
 **return** ctx;  
}

invokeChannelActive

**static void** invokeChannelActive(**final** AbstractChannelHandlerContext next) {

**//如果ServerBootstrapAcceptor的executor为空则返回eventLoop**  
 EventExecutor executor = next.executor();  
 **if** (executor.inEventLoop()) {

**//netx就是封装了ServerBootstrapAcceptor的ChannelHandlerContext** next.**invokeChannelActive**();  
 } **else** {  
 executor.execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 next.invokeChannelActive();  
 }  
 });  
 }  
}

**private void** invokeChannelActive() {

//判断该ChannelHandlerContext的处理状态handlerState是不是***ADD\_COMPLETE***  
 **if** (invokeHandler()) {  
 **try** {  
 ((ChannelInboundHandler) handler()).**channelActive**(**this**);  
 } **catch** (Throwable t) {  
 notifyHandlerException(t);  
 }  
 } **else** {  
 fireChannelActive();  
 }  
}

//ctx就是封装了ServerBootstrapAcceptor的ChannelHandlerContext

**public void** channelActive(ChannelHandlerContext ctx) **throws** Exception {  
 ctx.fireChannelActive();  
}

@Override  
**public** ChannelHandlerContext fireChannelActive() {  
 ***invokeChannelActive***(**findContextInbound**());  
 **return this**;  
}

findContextInbound

**private** AbstractChannelHandlerContext findContextInbound() {  
 AbstractChannelHandlerContext ctx = **this**;

//获取封装ServerBootsrapAcceptor的ChannelHandlerContext之后的第一个

//ChannelHandlerContextInbound,在这里是tail  
 **do** {  
 ctx = ctx.**next**;  
 } **while** (!ctx.**inbound**);  
 **return** ctx;  
}

*invokeChannelActive*

**static void** invokeChannelActive(**final** AbstractChannelHandlerContext next) {

//如果tail的executor为空则返回eventLoop  
 EventExecutor executor = next.executor();  
 **if** (executor.inEventLoop()) {

//next就是tail  
 next.invokeChannelActive();  
 } **else** {  
 executor.execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 next.invokeChannelActive();  
 }  
 });  
 }  
}

**private void** invokeChannelActive() {  
 **if** (invokeHandler()) {  
 **try** {

//Handler()返回的是tailContext,this就是tailContext  
 ((ChannelInboundHandler) handler()).**channelActive**(**this**);  
 } **catch** (Throwable t) {  
 notifyHandlerException(t);  
 }  
 } **else** {  
 fireChannelActive();  
 }  
}

@Override  
**public void** channelActive(ChannelHandlerContext ctx) **throws** Exception {  
 onUnhandledInboundChannelActive();  
}

*/\*\*  
 \* Called once the {****@link*** *ChannelInboundHandler#channelActive(ChannelHandlerContext)}event hit  
 \* the end of the {****@link*** *ChannelPipeline}.  
 \*/***protected void** onUnhandledInboundChannelActive() {  
}

###### readIfIsAutoRead

**private void** readIfIsAutoRead() {

//config()返回NioServerSocketChannelConfig  
 **if** (**channel**.config().isAutoRead()) {  
 **channel**.read();  
 }  
}

@Override  
**public** Channel read() {  
 **pipeline**.**read**();  
 **return this**;  
}

@Override  
**public final** ChannelPipeline read() {  
 **tail**.read();  
 **return this**;  
}

@Override  
**public** ChannelHandlerContext read() {

**final** AbstractChannelHandlerContext next = findContextOutbound();  
 EventExecutor executor = next.executor();  
 **if** (executor.inEventLoop()) {  
 next.**invokeRead**();  
 } **else** {  
 Runnable task = next.**invokeReadTask**;  
 **if** (task == **null**) {  
 next.**invokeReadTask** = task = **new** Runnable() {  
 @Override  
 **public void** run() {  
 next.invokeRead();  
 }  
 };  
 }  
 executor.execute(task);  
 }  
  
 **return this**;  
}

**private void** invokeRead() {  
 **if** (invokeHandler()) {  
 **try** {  
 ((ChannelOutboundHandler) handler()).**read**(**this**);  
 } **catch** (Throwable t) {  
 notifyHandlerException(t);  
 }  
 } **else** {  
 read();  
 }  
}

@Override  
**public void** read(ChannelHandlerContext ctx) {  
 **unsafe**.**beginRead**();  
}

@Override  
**public final void** beginRead() {  
 assertEventLoop();  
  
 **if** (!isActive()) {  
 **return**;  
 }  
  
 **try** {  
 **doBeginRead**();  
 } **catch** (**final** Exception e) {  
 invokeLater(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **pipeline**.fireExceptionCaught(e);  
 }  
 });  
 close(voidPromise());  
 }  
}

@Override  
**protected void** doBeginRead() **throws** Exception {  
 **if** (**inputShutdown**) {  
 **return**;  
 }  
 **super**.**doBeginRead**();  
}

@Override  
**protected void** doBeginRead() **throws** Exception {  
 *// Channel.read() or ChannelHandlerContext.read() was called* **final** SelectionKey selectionKey = **this**.**selectionKey**;  
 **if** (!selectionKey.isValid()) {  
 **return**;  
 }  
  
 **readPending** = **true**;  
 **//最初只是将ServerSocketChannel注册到selector并没有设置selector要监听**

**//ServerSocketChannel通道任何事件**  
 **final int** interestOps = selectionKey.interestOps();  
 **if** ((interestOps & **readInterestOp**) == 0) {  
 selectionKey.interestOps(interestOps | **readInterestOp**);  
 }  
}

# OP\_ACCEPT注册

**首先，封装ServerSocketChannel的NioServerSocketChannel创建时就指定了感兴趣事件OP\_ACCEPT; 然后，在注册阶段，直接向Selector注册OP\_ACCEPT**

**ServerSocketChannel注册OP\_ACCEPT可能就是第四个任务task,也可能在register0中**

**private void** register0(ChannelPromise promise) {  
 **try** {  
 *// check if the channel is still open as it could be closed in the mean time when the register  
 // call was outside of the eventLoop* **if** (!promise.setUncancellable() || !ensureOpen(promise)) {  
 **return**;  
 }

/**/neverRegistered：标识有没有将Channel注册到selector**

**//Channel有可能是ServerSocketChannel，也有可能是SocketChannel** **boolean** firstRegistration = **neverRegistered**;  
 doRegister();  
 **neverRegistered** = **false**;

//registered：将SocketChannel注册到selector  
 **registered** = **true**;

***//在我们实际通知promise之前，确保我们先调用handlerAdded()。***

***//这是必须的，因为用户可能已经通过ChannelFutureListener中的pipeline***

***//触发了事件*****pipeline**.invokeHandlerAddedIfNeeded();  
  
 safeSetSuccess(promise);  
 **pipeline**.fireChannelRegistered();  
 **if** (isActive()) {  
 **if** (firstRegistration) {

**//注册OP\_ACCEPT或OP\_READ事件**

**//1.在给ServerSocketChannel注册OP\_ACCEPT事件的时候可能在此步骤完成，也**

**//可能在task任务中完成的**

**//2.在给SocketChannel注册OP\_READ事件的时候是在此步骤完成的**  
 **pipeline**.fireChannelActive();  
 } **else if** (config().isAutoRead()) {  
 *// This channel was registered before and autoRead() is set. This means we need to begin read  
 // again so that we process inbound data.  
 //  
 // See https://github.com/netty/netty/issues/4805* beginRead();  
 }  
 }  
 } **catch** (Throwable t) {  
 *// Close the channel directly to avoid FD leak.* closeForcibly();  
 **closeFuture**.setClosed();  
 safeSetFailure(promise, t);  
 }  
}

# OP\_ACCEPT事件

## NioEventLoop.run

@Override  
**protected void** run() {  
 **for** (;;) {  
 **try** {  
 **switch** (**selectStrategy**.calculateStrategy(**selectNowSupplier**, hasTasks())) {

//1.SelectStrategy.***CONTINUE = -2***

//2.SelectStrategy.***SELECT = -1***

//3.由于selector.selectNow()不可能返回负数，因此如果

//taskQueue或tailQueue有任务需要优先执行完  
 **case** SelectStrategy.***CONTINUE***:  
 **continue**;  
 **case** SelectStrategy.***SELECT***:

**//wakenUp.getAndSet(false)得到并设置waKenUp** select(**wakenUp**.getAndSet(**false**));  
  
 *// 'wakenUp.compareAndSet(false, true)' is always evaluated  
 // before calling 'selector.wakeup()' to reduce the wake-up  
 // overhead. (Selector.wakeup() is an expensive operation.)  
 //  
 // However, there is a race condition in this approach.  
 // The race condition is triggered when 'wakenUp' is set to  
 // true too early.  
 //  
 // 'wakenUp' is set to true too early if:  
 // 1) Selector is waken up between 'wakenUp.set(false)' and  
 // 'selector.select(...)'. (BAD)  
 // 2) Selector is waken up between 'selector.select(...)' and  
 // 'if (wakenUp.get()) { ... }'. (OK)  
 //  
 // In the first case, 'wakenUp' is set to true and the  
 // following 'selector.select(...)' will wake up immediately.  
 // Until 'wakenUp' is set to false again in the next round,  
 // 'wakenUp.compareAndSet(false, true)' will fail, and therefore  
 // any attempt to wake up the Selector will fail, too, causing  
 // the following 'selector.select(...)' call to block  
 // unnecessarily.  
 //  
 // To fix this problem, we wake up the selector again if wakenUp  
 // is true immediately after selector.select(...).  
 // It is inefficient in that it wakes up the selector for both  
 // the first case (BAD - wake-up required) and the second case  
 // (OK - no wake-up required).* **if** (**wakenUp**.get()) {  
 **selector**.wakeup();  
 }  
 *// fall through* **default**:  
 }  
  
 **cancelledKeys** = 0;  
 **needsToSelectAgain** = **false**;  
 **final int** ioRatio = **this**.**ioRatio**;  
 **if** (ioRatio == 100) {  
 **try** {  
 processSelectedKeys();  
 } **finally** {  
 *// Ensure we always run tasks.* runAllTasks();  
 }  
 } **else** {  
 **final long** ioStartTime = System.*nanoTime*();  
 **try** {

**//处理SelectionKey** **processSelectedKeys();**  
 } **finally** {  
 ***// 确保我们总是运行任务*****final long** ioTime = System.*nanoTime*() - ioStartTime;  
 runAllTasks(ioTime \* (100 - ioRatio) / ioRatio);  
 }  
 }  
 } **catch** (Throwable t) {  
 *handleLoopException*(t);  
 }  
 *// Always handle shutdown even if the loop processing threw an exception.* **try** {  
 **if** (isShuttingDown()) {  
 closeAll();  
 **if** (confirmShutdown()) {  
 **return**;  
 }  
 }  
 } **catch** (Throwable t) {  
 *handleLoopException*(t);  
 }  
 }  
}

### selectNowSupplier

**private final** IntSupplier **selectNowSupplier** = **new** IntSupplier() {  
 @Override  
 **public int** get() **throws** Exception {  
 **return** selectNow();  
 }  
};

**int** selectNow() **throws** IOException {  
 **try** {

//1.selector.selectNow()选择一组键，其相应的通道已为I/O操作准备就绪。

//此方法执行非阻塞的选择操作。如果自从前一次选择操作后，没有通道变成可

//选择的，则此方法直接返回零。  
 **return selector**.selectNow();  
 } **finally** {  
 ***//如果waKenup为true则使尚未返回的第一个选择操作立即返回*****if** (**wakenUp**.get()) {  
 **selector**.wakeup();  
 }  
 }  
}

### hasTasks

@Override  
**protected boolean** hasTasks() {  
 **return super**.hasTasks() || !**tailTasks**.isEmpty();  
}

*/\*\*  
 \** ***@see*** *Queue#isEmpty()  
 \*/***protected boolean** hasTasks() {  
 **assert** inEventLoop();  
 **return** !**taskQueue**.isEmpty();  
}

### calculateStrategy

@Override  
**public int** calculateStrategy(IntSupplier selectSupplier, **boolean** hasTasks) **throws** Exception {

//1.hasTasks判断**taskQueue**或**tailTasks是否还有未执行的任务**  
 **return** hasTasks ? selectSupplier.get() : SelectStrategy.***SELECT***;  
}

### select(wakenUp.getAndSet(false))

**private void** select(**boolean** oldWakenUp) **throws** IOException {

**//selector是io.netty.channel.nio.SelectedSelectionKeySetSelector**

**//该类是个final类并且该类继承了java.nio.channels.Selector**

**//该类只有两个字段一个字段用于存储已经选择的SelectionKey的集合，另一个**

**//字段用于存储selector与平台相关的实现**  
 Selector selector = **this**.**selector**;  
 **try** {  
 **int** selectCnt = 0;

**//获取JVM从某个固定时间A开始到现在的纳秒值，在同一个JVM中这个**

**//起始计时时间A是固定的。**

**//nanoTime() = 当前时间 - A** **long** currentTimeNanos = System.*nanoTime*();  
 **long** selectDeadLineNanos = currentTimeNanos + delayNanos(currentTimeNanos);  
 **for** (;;) {

**//结果毫秒，除法运算，不保留小数点之后**  
 **long** timeoutMillis = (selectDeadLineNanos - currentTimeNanos + 500000L) / 1000000L;  
 **if** (timeoutMillis <= 0) {  
 **if** (selectCnt == 0) {  
 selector.selectNow();  
 selectCnt = 1;  
 }  
 **break**;  
 }  
  
 ***//当wakenUp的值为true时如果提交了一个任务，则该任务没有机会调用***

***//Selector.wakeup。因此，在执行select操作之前，我们需要再次检查***

***//任务队列。如果不这样做，任务可能会一直挂起，直到select操作超时。***

***//如果IdleStateHandler存在于管道中，则可能将其挂起直到空闲超时。*****if** (hasTasks() && **wakenUp**.compareAndSet(**false**, **true**)) {  
 selector.selectNow();  
 selectCnt = 1;  
 **break**;  
 }  
 **//1.选择一组键，其相应的通道已为I/O操作准备就绪**

**//2.timeoutMillis:选择超时时间，超过此时间之后不管有没有通道的I/O**

**//操作准备就绪都返回，返回值为0** **int** selectedKeys = selector.select(timeoutMillis);  
 selectCnt ++;  
 **//1.如果选择至少一个通道，**

**//2.wakenUp为true,**

**//3.当前的wakenUp为true**

**//4.taskQueue是否为空**

**//5.scheduledTaskQueue是否为空**  
 **if** (selectedKeys != 0 || oldWakenUp || **wakenUp**.get() || hasTasks() || hasScheduledTasks()) {  
 *// - Selected something,  
 // - waken up by user, or  
 // - the task queue has a pending task.  
 // - a scheduled task is ready for processing* **break**;  
 }

//线程被中断，因此重置selectedKeys并跳出for循环，这样我们就不会

//陷入繁忙循环。由于这很可能是用户的处理程序或其客户端库中的错误，

//所以我们也将对其进行日志记录。

**if** (Thread.*interrupted*()) {  
 *// Thread was interrupted so reset selected keys and break so we not run into a busy loop.  
 // As this is most likely a bug in the handler of the user or it's client library we will  
 // also log it.  
 //  
 // See https://github.com/netty/netty/issues/2426* **if** (***logger***.isDebugEnabled()) {  
 ***logger***.debug(**"Selector.select() returned prematurely because "** +  
 **"Thread.currentThread().interrupt() was called. Use "** +  
 **"NioEventLoop.shutdownGracefully() to shutdown the NioEventLoop."**);  
 }  
 selectCnt = 1;  
 **break**;  
 }  
  
 **long** time = System.*nanoTime*();  
 **if** (time - TimeUnit.***MILLISECONDS***.toNanos(timeoutMillis) >= currentTimeNanos) {  
 *// timeoutMillis elapsed without anything selected.*

***//超时时间已过，没有选择到任何键***selectCnt = 1;

//***SELECTOR\_AUTO\_REBUILD\_THRESHOLD ： selector重新被构建的阈值，***

**//当selectCnt超过该阈值时selector就会被重新构建**  
 } **else if** (***SELECTOR\_AUTO\_REBUILD\_THRESHOLD*** > 0 &&  
 selectCnt >= ***SELECTOR\_AUTO\_REBUILD\_THRESHOLD***) {  
 *// The selector returned prematurely many times in a row.  
 // Rebuild the selector to work around the problem.*

***//选择器多次连续地提前返回，重新构建选择器以解决问题。******logger***.warn(  
 **"Selector.select() returned prematurely {} times in a row; rebuilding Selector {}."**,  
 selectCnt, selector);  
  
 rebuildSelector();  
 selector = **this**.**selector**;  
  
 *// Select again to populate selectedKeys.*

***//Select再一次填充selectedKeys***selector.selectNow();  
 selectCnt = 1;  
 **break**;  
 }  
  
 currentTimeNanos = time;  
 }

//***MIN\_PREMATURE\_SELECTOR\_RETURNS = 3最小的提前返回数***  
 **if** (selectCnt > ***MIN\_PREMATURE\_SELECTOR\_RETURNS***) {  
 **if** (***logger***.isDebugEnabled()) {  
 ***logger***.debug(**"Selector.select() returned prematurely {} times in a row for Selector {}."**,  
 selectCnt - 1, selector);  
 }  
 }  
 } **catch** (CancelledKeyException e) {  
 **if** (***logger***.isDebugEnabled()) {  
 ***logger***.debug(CancelledKeyException.**class**.getSimpleName() + **" raised by a Selector {} - JDK bug?"**,  
 selector, e);  
 }  
 *// Harmless exception - log anyway* }  
}

#### SelectedSelectionKeySetSelector的实现

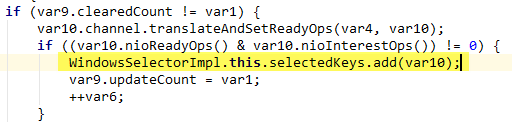
**final class** SelectedSelectionKeySetSelector **extends** Selector {  
 **private final** SelectedSelectionKeySet **selectionKeys**;  
 **private final** Selector **delegate**;  
  
 SelectedSelectionKeySetSelector(Selector delegate, SelectedSelectionKeySet selectionKeys) {  
 **this**.**delegate** = delegate;  
 **this**.**selectionKeys** = selectionKeys;  
 }  
  
 @Override  
 **public boolean** isOpen() {  
 **return delegate**.isOpen();  
 }  
  
 @Override  
 **public** SelectorProvider provider() {  
 **return delegate**.provider();  
 }  
  
 @Override  
 **public** Set<SelectionKey> keys() {  
 **return delegate**.keys();  
 }  
  
 @Override  
 **public** Set<SelectionKey> selectedKeys() {  
 **return delegate**.selectedKeys();  
 }  
  
 @Override  
 **public int** selectNow() **throws** IOException {  
 **selectionKeys**.reset();  
 **return delegate**.selectNow();  
 }  
  
 @Override  
 **public int** select(**long** timeout) **throws** IOException {  
 **selectionKeys**.reset();  
 **return delegate**.select(timeout);  
 }  
  
 @Override  
 **public int** select() **throws** IOException {  
 **selectionKeys**.reset();  
 **return delegate**.select();  
 }  
  
 @Override  
 **public** Selector wakeup() {  
 **return delegate**.wakeup();  
 }  
  
 @Override  
 **public void** close() **throws** IOException {  
 **delegate**.close();  
 }  
}

#### SelectedSelectionKeySet的实现

**在Netty中使用自定义的SelectedSelectionKeySet替换掉了sun.nio.ch.SelectorImpl中的selectedKeys和publicSelectedKeys，这样性能会更好**

**final class** SelectedSelectionKeySet **extends** AbstractSet<SelectionKey> {  
  
 SelectionKey[] **keys**;  
 **int size**;  
  
 SelectedSelectionKeySet() {  
 **keys** = **new** SelectionKey[1024];  
 }  
  
 @Override  
 **public boolean** add(SelectionKey o) {  
 **if** (o == **null**) {  
 **return false**;  
 }  
  
 **keys**[**size**++] = o;  
 **if** (**size** == **keys**.**length**) {  
 increaseCapacity();  
 }  
  
 **return true**;  
 }  
  
 @Override  
 **public int** size() {  
 **return size**;  
 }  
  
 @Override  
 **public boolean** remove(Object o) {  
 **return false**;  
 }  
  
 @Override  
 **public boolean** contains(Object o) {  
 **return false**;  
 }  
  
 @Override  
 **public** Iterator<SelectionKey> iterator() {  
 **throw new** UnsupportedOperationException();  
 }  
  
 **void** reset() {  
 reset(0);  
 }  
  
 **void** reset(**int** start) {  
 Arrays.*fill*(**keys**, start, **size**, **null**);  
 **size** = 0;  
 }  
  
 **private void** increaseCapacity() {  
 SelectionKey[] newKeys = **new** SelectionKey[**keys**.**length** << 1];  
 System.*arraycopy*(**keys**, 0, newKeys, 0, **size**);  
 **keys** = newKeys;  
 }  
}

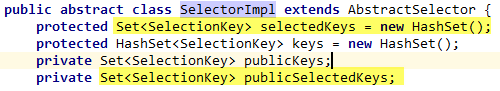
**selectionKeys如何赋值的？**



**在这里WindowSelectorImpl.this.selectedKeys得到的就是SelectedSelectionKeySet**

**为什么呢？在我们看JDK源码的时候WindowSelectorImpl.this.selectedKeys和**

**WindowSelectorImpl.this.publicSelectedKeys的类型是HashSet；如果我们不对SelectorImpl进行修改，则selectedKeys和publicSelectedKeys的类型就是HashSet**



**什么时候对SelectorImpl进行修改的？在NioEventLoop初始化的时候对SelectorImpl进行修改的。**

**NioEventLoop的构造函数如下：**

NioEventLoop(NioEventLoopGroup parent, Executor executor, SelectorProvider selectorProvider,  
 SelectStrategy strategy, RejectedExecutionHandler rejectedExecutionHandler) {  
 **super**(parent, executor, **false**, ***DEFAULT\_MAX\_PENDING\_TASKS***, rejectedExecutionHandler);  
 **if** (selectorProvider == **null**) {  
 **throw new** NullPointerException(**"selectorProvider"**);  
 }  
 **if** (strategy == **null**) {  
 **throw new** NullPointerException(**"selectStrategy"**);  
 }  
 **provider** = selectorProvider;  
 **final** SelectorTuple selectorTuple = **openSelector()**;

**//默认是SelectedSelectionKeySetSelector，**

**//也可能是原始的selector，是可配置的**  
 **selector** = selectorTuple.**selector**;

**//原始的selector** **unwrappedSelector** = selectorTuple.**unwrappedSelector**;  
 **selectStrategy** = strategy;  
}

**//openSelector()方法的实现**

**private** SelectorTuple openSelector() {  
 **final** Selector unwrappedSelector;  
 **try** {

**//打开一个选择器** unwrappedSelector = **provider**.openSelector();  
 } **catch** (IOException e) {  
 **throw new** ChannelException(**"failed to open a new selector"**, e);  
 }  
 **//是否禁止SelectionKeySet优化，如果禁止则不对SelectorImpl进行修改** **if** (***DISABLE\_KEYSET\_OPTIMIZATION***) {  
 **return new** SelectorTuple(unwrappedSelector);  
 }  
 **//优化SelectionKeySet** **final** SelectedSelectionKeySet selectedKeySet = **new** SelectedSelectionKeySet();  
  
 Object maybeSelectorImplClass = AccessController.*doPrivileged*(**new** PrivilegedAction<Object>() {  
 @Override  
 **public** Object run() {  
 **try** {  
 **return** Class.*forName*(  
 **"sun.nio.ch.SelectorImpl"**,  
 **false**,  
 PlatformDependent.*getSystemClassLoader*());  
 } **catch** (Throwable cause) {  
 **return** cause;  
 }  
 }  
 });  
  
 **if** (!(maybeSelectorImplClass **instanceof** Class) ||  
 *// ensure the current selector implementation is what we can instrument.* !((Class<?>) maybeSelectorImplClass).isAssignableFrom(unwrappedSelector.getClass())) {  
 **if** (maybeSelectorImplClass **instanceof** Throwable) {  
 Throwable t = (Throwable) maybeSelectorImplClass;  
 ***logger***.trace(**"failed to instrument a special java.util.Set into: {}"**, unwrappedSelector, t);  
 }  
 **return new** SelectorTuple(unwrappedSelector);  
 }  
  
 **final** Class<?> selectorImplClass = (Class<?>) maybeSelectorImplClass;  
  **//将SelectorImpl的成员变量selectedKeys和publicSelectedKeys类型由HashSet修改成**

**//SelectedSelectionKeySet**  
 Object maybeException = AccessController.*doPrivileged*(**new** PrivilegedAction<Object>() {  
 @Override  
 **public** Object run() {  
 **try** {  
 Field selectedKeysField = selectorImplClass.getDeclaredField(**"selectedKeys"**);  
 Field publicSelectedKeysField = selectorImplClass.getDeclaredField(**"publicSelectedKeys"**);  
  
 Throwable cause = ReflectionUtil.*trySetAccessible*(selectedKeysField, **true**);  
 **if** (cause != **null**) {  
 **return** cause;  
 }  
 cause = ReflectionUtil.*trySetAccessible*(publicSelectedKeysField, **true**);  
 **if** (cause != **null**) {  
 **return** cause;  
 }  
  
 selectedKeysField.set(unwrappedSelector, selectedKeySet);  
 publicSelectedKeysField.set(unwrappedSelector, selectedKeySet);  
 **return null**;  
 } **catch** (NoSuchFieldException e) {  
 **return** e;  
 } **catch** (IllegalAccessException e) {  
 **return** e;  
 }  
 }  
 });  
 **//如果发生异常则禁止SelectionKeySet优化** **if** (maybeException **instanceof** Exception) {  
 **selectedKeys** = **null**;  
 Exception e = (Exception) maybeException;  
 ***logger***.trace(**"failed to instrument a special java.util.Set into: {}"**, unwrappedSelector, e);  
 **return new** SelectorTuple(unwrappedSelector);  
 }  
 **selectedKeys** = selectedKeySet;  
 ***logger***.trace(**"instrumented a special java.util.Set into: {}"**, unwrappedSelector);  
 **return new** SelectorTuple(unwrappedSelector,  
 **new** SelectedSelectionKeySetSelector(unwrappedSelector, selectedKeySet));  
}

#### delayNanos

*/\*\*  
 \* Returns the amount of time left until the scheduled task with the closest dead line is executed.  
 \*/***protected long** delayNanos(**long** currentTimeNanos) {  
 ScheduledFutureTask<?> scheduledTask = **peekScheduledTask();** **if** (scheduledTask == **null**) {

**//*SCHEDULE\_PURGE\_INTERVAL=1秒，即10^9纳秒***  
 **return *SCHEDULE\_PURGE\_INTERVAL***;  
 }  
  
 **return scheduledTask.delayNanos(currentTimeNanos)**;  
}

##### peekScheduledTask

**//如果scheduledTaskQueue中有任务，则返回**

**final** ScheduledFutureTask<?> peekScheduledTask() {  
 Queue<ScheduledFutureTask<?>> scheduledTaskQueue = **this**.**scheduledTaskQueue**;  
 **if** (scheduledTaskQueue == **null**) {  
 **return null**;  
 }  
 **return** scheduledTaskQueue.peek();  
}

##### scheduledTask.delayNanos

**public long** delayNanos(**long** currentTimeNanos) {

**//1.*START\_TIME 任务task创建时初始化的时间***  
 **return** Math.*max*(0, deadlineNanos() - (currentTimeNanos - ***START\_TIME***));  
}

### processSelectedKeys

**private void** processSelectedKeysOptimized() {  
 **for** (**int** i = 0; i < **selectedKeys**.**size**; ++i) {  
 **final** SelectionKey k = **selectedKeys**.**keys**[i];  
 *// null out entry in the array to allow to have it GC'ed once the Channel close  
 // See https://github.com/netty/netty/issues/2363* **selectedKeys**.**keys**[i] = **null**;  
  
 **final** Object a = k.attachment();  
  
 **if** (a **instanceof** AbstractNioChannel) {  
 **processSelectedKey**(k, (AbstractNioChannel) a);  
 } **else** {  
 @SuppressWarnings(**"unchecked"**)  
 NioTask<SelectableChannel> task = (NioTask<SelectableChannel>) a;  
 *processSelectedKey*(k, task);  
 }  
  
 **if** (**needsToSelectAgain**) {  
 *// null out entries in the array to allow to have it GC'ed once the Channel close  
 // See https://github.com/netty/netty/issues/2363* **selectedKeys**.reset(i + 1);  
  
 selectAgain();  
 i = -1;  
 }  
 }  
}

**private void** processSelectedKey(SelectionKey k, AbstractNioChannel ch) {  
 **final** AbstractNioChannel.NioUnsafe unsafe = ch.unsafe();  
 **if** (!k.isValid()) {  
 **final** EventLoop eventLoop;  
 **try** {  
 eventLoop = ch.eventLoop();  
 } **catch** (Throwable ignored) {  
 *// If the channel implementation throws an exception because there is no event loop, we ignore this  
 // because we are only trying to determine if ch is registered to this event loop and thus has authority  
 // to close ch.* **return**;  
 }  
 *// Only close ch if ch is still registered to this EventLoop. ch could have deregistered from the event loop  
 // and thus the SelectionKey could be cancelled as part of the deregistration process, but the channel is  
 // still healthy and should not be closed.  
 // See https://github.com/netty/netty/issues/5125* **if** (eventLoop != **this** || eventLoop == **null**) {  
 **return**;  
 }  
 *// close the channel if the key is not valid anymore* unsafe.close(unsafe.voidPromise());  
 **return**;  
 }  
  
 **try** {  
 **int** readyOps = k.readyOps();  
 *// We first need to call finishConnect() before try to trigger a read(...) or write(...) as otherwise  
 // the NIO JDK channel implementation may throw a NotYetConnectedException.* **if** ((readyOps & SelectionKey.***OP\_CONNECT***) != 0) {  
 *// remove OP\_CONNECT as otherwise Selector.select(..) will always return without blocking  
 // See https://github.com/netty/netty/issues/924* **int** ops = k.interestOps();  
 ops &= ~SelectionKey.***OP\_CONNECT***;  
 k.interestOps(ops);  
  
 unsafe.finishConnect();  
 }  
  
 *// Process OP\_WRITE first as we may be able to write some queued buffers and so free memory.* **if** ((readyOps & SelectionKey.***OP\_WRITE***) != 0) {  
 *// Call forceFlush which will also take care of clear the OP\_WRITE once there is nothing left to write* ch.unsafe().forceFlush();  
 }  
  
 *// Also check for readOps of 0 to workaround possible JDK bug which may otherwise lead  
 // to a spin loop* **if** ((readyOps & (SelectionKey.***OP\_READ*** | SelectionKey.***OP\_ACCEPT***)) != 0 || readyOps == 0) {  
 **unsafe.read();** }  
 } **catch** (CancelledKeyException ignored) {  
 unsafe.close(unsafe.voidPromise());  
 }  
}

@Override  
**public void** read() {  
 **assert** eventLoop().inEventLoop();  
 **final** ChannelConfig config = config();  
 **final** ChannelPipeline pipeline = pipeline();  
 **final** RecvByteBufAllocator.Handle allocHandle = unsafe().recvBufAllocHandle();  
 allocHandle.reset(config);  
  
 **boolean** closed = **false**;  
 Throwable exception = **null**;  
 **try** {  
 **try** {  
 **do** {

//将SocketChannel分装成NioSocketChannel

**//将准备好读的NioSocketChannel存储在集合readBuf中** **int** localRead = **doReadMessages**(**readBuf**);  
 **if** (localRead == 0) {  
 **break**;  
 }  
 **if** (localRead < 0) {  
 closed = **true**;  
 **break**;  
 }  
 **//totalMessages: 已经准备好读通道的数量计数** allocHandle.incMessagesRead(localRead);  
 } **while** (allocHandle.continueReading());  
 } **catch** (Throwable t) {  
 exception = t;  
 }  
  
 **int** size = **readBuf**.size();  
 **for** (**int** i = 0; i < size; i ++) {  
 **readPending** = **false**;  
 **pipeline.fireChannelRead(readBuf.get(i))**;  
 }  
 **readBuf**.clear();  
 allocHandle.readComplete();  
 pipeline.fireChannelReadComplete();  
  
 **if** (exception != **null**) {  
 closed = closeOnReadError(exception);  
  
 pipeline.fireExceptionCaught(exception);  
 }  
  
 **if** (closed) {  
 **inputShutdown** = **true**;  
 **if** (isOpen()) {  
 close(voidPromise());  
 }  
 }  
 } **finally** {  
 *// Check if there is a readPending which was not processed yet.  
 // This could be for two reasons:  
 // \* The user called Channel.read() or ChannelHandlerContext.read() in channelRead(...) method  
 // \* The user called Channel.read() or ChannelHandlerContext.read() in channelReadComplete(...) method  
 //  
 // See https://github.com/netty/netty/issues/2254* **if** (!**readPending** && !config.isAutoRead()) {  
 removeReadOp();  
 }  
 }  
 }  
}

#### doReadMessages

@Override  
**protected int** doReadMessages(List<Object> buf) **throws** Exception {  
 SocketChannel ch = SocketUtils.*accept*(javaChannel());  
  
 **try** {  
 **if** (ch != **null**) {

**//将SocketChannel封装成NioSocketChannel**

**//NioSocketChannel对*OP\_READ事件感兴趣*** buf.add(**new** NioSocketChannel(**this**, ch));  
 **return** 1;  
 }  
 } **catch** (Throwable t) {  
 ***logger***.warn(**"Failed to create a new channel from an accepted socket."**, t);  
  
 **try** {  
 ch.close();  
 } **catch** (Throwable t2) {  
 ***logger***.warn(**"Failed to close a socket."**, t2);  
 }  
 }  
  
 **return** 0;  
}

#### pipeline.fireChannelRead(readBuf.get(i));

对pipeline.fireChannelRead(readBuf.get(i))进行说明，这里pipeline是NioServerSocketChannel的对应的pipeline,因此pipeline链表结构如下：

//参数msg就是NioSocketChannel  
**public final** ChannelPipeline fireChannelRead(Object msg) {  
 AbstractChannelHandlerContext.*invokeChannelRead*(**head**, msg);  
 **return this**;  
}

**static void** invokeChannelRead(**final** AbstractChannelHandlerContext next, Object msg) {  
 **final** Object m = next.**pipeline**.touch(ObjectUtil.*checkNotNull*(msg, **"msg"**), next);  
 EventExecutor executor = next.executor();  
 **if** (executor.inEventLoop()) {  
 **next.invokeChannelRead(m);** } **else** {  
 executor.execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 next.invokeChannelRead(m);  
 }  
 });  
 }  
}

**private void** invokeChannelRead(Object msg) {  
 **if** (invokeHandler()) {  
 **try** {  
 ((ChannelInboundHandler) handler()).channelRead(**this**, msg);  
 } **catch** (Throwable t) {  
 notifyHandlerException(t);  
 }  
 } **else** {  
 fireChannelRead(msg);  
 }  
}

@Override  
**public void** channelRead(ChannelHandlerContext ctx, Object msg) **throws** Exception {  
 ctx.fireChannelRead(msg);  
}

@Override  
**public** ChannelHandlerContext fireChannelRead(**final** Object msg) {

**//findContextInBound找到下一个封装ServerBootstrapAcceptor的**

**//ChannelHandlerContextInBound**  
 *invokeChannelRead*(findContextInbound(), msg);  
 **return this**;  
}

**static void** invokeChannelRead(**final** AbstractChannelHandlerContext next, Object msg) {  
 **final** Object m = next.**pipeline**.touch(ObjectUtil.*checkNotNull*(msg, **"msg"**), next);  
 EventExecutor executor = next.executor();  
 **if** (executor.inEventLoop()) {

**//封装ServerBootstrapAcceptor的ChannelHandlerContextInbound** next.invokeChannelRead(m);  
 } **else** {  
 executor.execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 next.invokeChannelRead(m);  
 }  
 });  
 }  
}

**//ChannelInitailizer有一个抽象方法initChannel**

**//ServerBootstrapAcceptor有个方法channelRead**

@Override  
@SuppressWarnings(**"unchecked"**)  
**public void** channelRead(ChannelHandlerContext ctx, Object msg) {

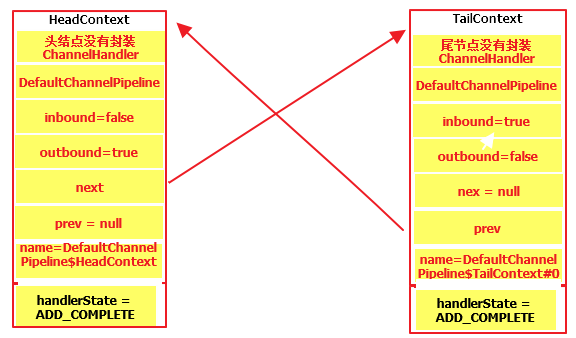
**//msg就是NioSocketChannel**  
 **final** Channel child = (Channel) msg;

**//此childHandler就是我自定义的childHandler,见下图**

**//将childHandler封装成ChannelHandlerContext，然后添加到pipeline的链表**  **child.pipeline().addLast**(**childHandler**);  
  
 *setChannelOptions*(child, **childOptions**, ***logger***);  
  
 **for** (Entry<AttributeKey<?>, Object> e: **childAttrs**) {  
 child.attr((AttributeKey<Object>) e.getKey()).set(e.getValue());  
 }  
  
 **try** {  
 **childGroup**.register(child).addListener(**new** ChannelFutureListener() {  
 @Override  
 **public void** operationComplete(ChannelFuture future) **throws** Exception {  
 **if** (!future.isSuccess()) {  
 *forceClose*(child, future.cause());  
 }  
 }  
 });  
 } **catch** (Throwable t) {  
 *forceClose*(child, t);  
 }  
}

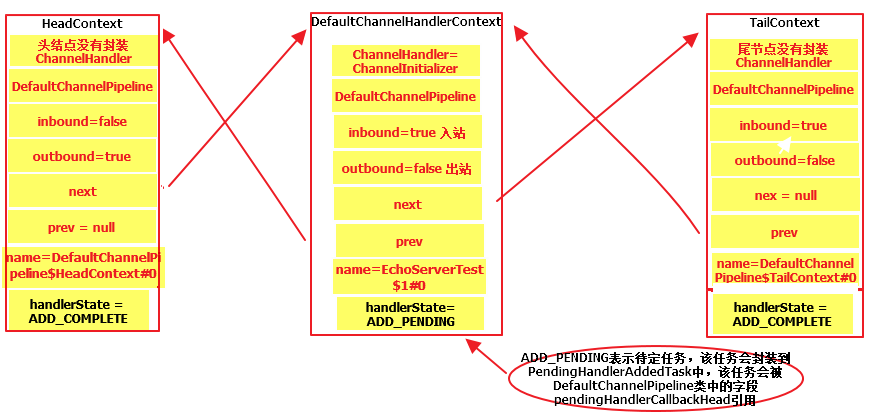


在进行此步骤之前，与NioSocketChannel关联pipeline的链表结构



#### child.pipeline().addLast() **public final** ChannelPipeline addLast(ChannelHandler... handlers) { **return addLast**(**null**, handlers); }

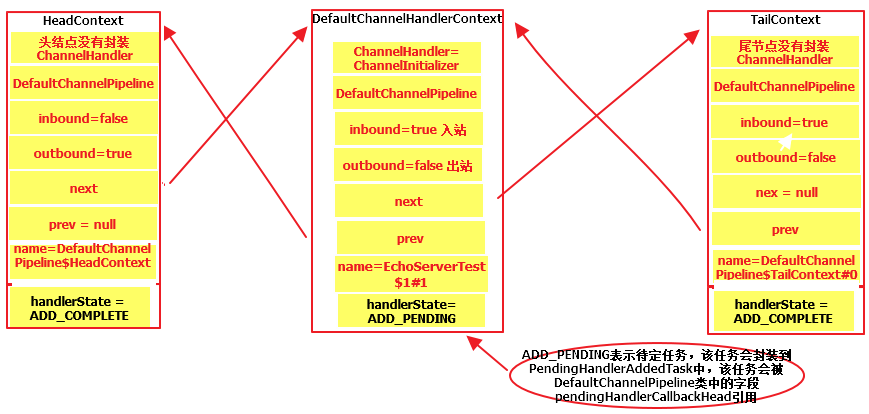
在进行此步骤之后，与NioSocketChannel关联的pipeline的链表结构如下：



**public final** ChannelPipeline **addLast**(EventExecutorGroup executor, ChannelHandler... handlers) {  
 **if** (handlers == **null**) {  
 **throw new** NullPointerException(**"handlers"**);  
 }  
  
 **for** (ChannelHandler h: handlers) {  
 **if** (h == **null**) {  
 **break**;  
 }  
 **addLast(executor, null, h);** }  
  
 **return this**;  
}

每个Channel都有一个Pipeline  
**public final** ChannelPipeline **addLast**(EventExecutorGroup group, String name, ChannelHandler handler) {  
 **final** AbstractChannelHandlerContext newCtx;  
 **synchronized** (**this**) {  
 *checkMultiplicity*(handler);  
  
 newCtx = newContext(group, filterName(name, handler), handler);  
  
 addLast0(newCtx);  
  
 *// If the registered is false it means that the channel was not registered on an eventloop yet.  
 // In this case we add the context to the pipeline and add a task that will call  
 // ChannelHandler.handlerAdded(...) once the channel is registered.* **if** (!**registered**) {  
 newCtx.setAddPending();  
 callHandlerCallbackLater(newCtx, **true**);  
 **return this**;  
 }  
  
 EventExecutor executor = newCtx.executor();  
 **if** (!executor.inEventLoop()) {  
 newCtx.setAddPending();  
 executor.execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 callHandlerAdded0(newCtx);  
 }  
 });  
 **return this**;  
 }  
 }  
 callHandlerAdded0(newCtx);  
 **return this**;  
}

#### pipeline链表结构



#### **childGroup**.register(child)

**public** ChannelFuture register(Channel channel) {

//1.next():从workerGroup中找一个eventLoop

//2.channel就是NioSocketChannel  
 **return** next().**register**(channel);  
}

@Override  
**public** ChannelFuture register(Channel channel) {  
 **return register**(**new** DefaultChannelPromise(channel, **this**));  
}

@Override  
**public** ChannelFuture **register**(**final** ChannelPromise promise) {  
 ObjectUtil.*checkNotNull*(promise, **"promise"**);  
 promise.channel().unsafe().register(**this**, promise);  
 **return** promise;  
}

@Override  
**public final void register**(EventLoop eventLoop, **final** ChannelPromise promise) {  
 **if** (eventLoop == **null**) {  
 **throw new** NullPointerException(**"eventLoop"**);  
 }  
 **if** (isRegistered()) {  
 promise.setFailure(**new** IllegalStateException(**"registered to an event loop already"**));  
 **return**;  
 }  
 **if** (!isCompatible(eventLoop)) {  
 promise.setFailure(  
 **new** IllegalStateException(**"incompatible event loop type: "** + eventLoop.getClass().getName()));  
 **return**;  
 }  
  
 AbstractChannel.**this**.**eventLoop** = eventLoop;  
  
 **if** (eventLoop.inEventLoop()) {  
 register0(promise);  
 } **else** {  
 **try** {

**//1.eventLoop.execute将任务task添加到任务队列taskQueue**

**//2.为eventLoop分配线程**  
 eventLoop.execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 register0(promise);  
 }  
 });  
 } **catch** (Throwable t) {  
 ***logger***.warn(  
 **"Force-closing a channel whose registration task was not accepted by an event loop: {}"**,  
 AbstractChannel.**this**, t);  
 closeForcibly();  
 **closeFuture**.setClosed();  
 safeSetFailure(promise, t);  
 }  
 }  
}

##### **register0**

**private void** register0(ChannelPromise promise) {  
 **try** {  
 *// check if the channel is still open as it could be closed in the mean time when the register  
 // call was outside of the eventLoop* **if** (!promise.setUncancellable() || !ensureOpen(promise)) {  
 **return**;  
 }

/**/neverRegistered：标识有没有将Channel注册到selector**

**//Channel有可能是ServerSocketChannel，也有可能是SocketChannel** **boolean** firstRegistration = **neverRegistered**;  
 doRegister();  
 **neverRegistered** = **false**;

//registered：将SocketChannel注册到selector  
 **registered** = **true**;

***//1.在我们实际通知promise之前，确保我们先调用handlerAdded()。在***

***//handlerAdded()方法里面最终会调用到ChannelInitializer的***

***//initChannel()方法来完成将channelHandler添加到pipeline(注意：***

***//channelHandler是封装成channelHandlerContext添加到pipeline的)，***

***//执行完后会从链表中删除ChannelHandlerContext***

***//2.这是必须的，因为用户可能已经通过ChannelFutureListener中***

***//的pipeline触发了事件*****pipeline**.invokeHandlerAddedIfNeeded();  
  
 safeSetSuccess(promise);

**//这步骤可能什么都不会操作** **pipeline**.fireChannelRegistered();  
 **if** (isActive()) {  
 **if** (firstRegistration) {

**//注册OP\_ACCEPT或OP\_READ事件**

**//1.在给ServerSocketChannel注册OP\_ACCEPT事件的时候可能在此步骤完成，也**

**//可能在task任务中完成的**

**//2.在给SocketChannel注册OP\_READ事件的时候是在此步骤完成的**  
 **pipeline**.fireChannelActive();  
 } **else if** (config().isAutoRead()) {  
 *// This channel was registered before and autoRead() is set. This means we need to begin read  
 // again so that we process inbound data.  
 //  
 // See https://github.com/netty/netty/issues/4805* beginRead();  
 }  
 }  
 } **catch** (Throwable t) {  
 *// Close the channel directly to avoid FD leak.* closeForcibly();  
 **closeFuture**.setClosed();  
 safeSetFailure(promise, t);  
 }  
}

###### pipeline.invokeHandlerAddedIfNeeded()

**final void** invokeHandlerAddedIfNeeded() {  
 **assert channel**.eventLoop().inEventLoop();  
 **if** (**firstRegistration**) {  
 **firstRegistration** = **false**;  
 *// We are now registered to the EventLoop. It's time to call the callbacks for the ChannelHandlers,  
 // that were added before the registration was done.* callHandlerAddedForAllHandlers();  
 }  
}

**private void** callHandlerAddedForAllHandlers() {  
 **final** PendingHandlerCallback pendingHandlerCallbackHead;  
 **synchronized** (**this**) {  
 **assert** !**registered**;  
  
 *// 该通道channnel已经被注册* **registered** = **true**;  
  
 pendingHandlerCallbackHead = **this**.**pendingHandlerCallbackHead**;  
 *// Null out so it can be GC'ed.* **this**.**pendingHandlerCallbackHead** = **null**;  
 }  
  
 *// This must happen outside of the synchronized(...) block as otherwise handlerAdded(...) may be called while  
 // holding the lock and so produce a deadlock if handlerAdded(...) will try to add another handler from outside  
 // the EventLoop.* PendingHandlerCallback task = pendingHandlerCallbackHead;  
 **while** (task != **null**) {  
 task.execute();  
 task = task.**next**;  
 }  
}

**void** execute() {

//ctx返回DefaultChannelHandlerContext  
 EventExecutor executor = **ctx**.executor();  
 **if** (executor.inEventLoop()) {  
 **callHandlerAdded0(ctx);** } **else** {  
 **try** {  
 executor.execute(**this**);  
 } **catch** (RejectedExecutionException e) {  
 **if** (***logger***.isWarnEnabled()) {  
 ***logger***.warn(  
 **"Can't invoke handlerAdded() as the EventExecutor {} rejected it, removing handler {}."**,  
 executor, **ctx**.name(), e);  
 }  
 *remove0*(**ctx**);  
 **ctx**.setRemoved();  
 }  
 }  
 }  
}

**rivate void** callHandlerAdded0(**final** AbstractChannelHandlerContext ctx) {  
 **try** {  
 *// We must call setAddComplete before calling handlerAdded. Otherwise if the handlerAdded method generates  
 // any pipeline events ctx.handler() will miss them because the state will not allow it.*

***//在调用handerAdded()之前，我们必须调用setAddComplete。***

***//否则如果handlerAdded()方法产生任何pipeline事件，ctx.handler()***

***//将错过它们，因为ChannelHandlerContext的状态决定了不允许调用它。***ctx.setAddComplete();

**//1.ctx.handler()返回ChannelInitializer，在这里ChannelInitializer**

**//是用户自定义的**

**//2.ChannelInitializer.handlerAdded(),该方法最终会调用到**

**//initChannel(Channel ch),在这里ch可能是NioServerSocketChannel**

**//也可能是NioSocketChannel** ctx.handler().**handlerAdded(ctx)**;  
 } **catch** (Throwable t) {  
 **boolean** removed = **false**;  
 **try** {  
 *remove0*(ctx);  
 **try** {  
 ctx.handler().handlerRemoved(ctx);  
 } **finally** {  
 ctx.setRemoved();  
 }  
 removed = **true**;  
 } **catch** (Throwable t2) {  
 **if** (***logger***.isWarnEnabled()) {  
 ***logger***.warn(**"Failed to remove a handler: "** + ctx.name(), t2);  
 }  
 }  
  
 **if** (removed) {  
 fireExceptionCaught(**new** ChannelPipelineException(  
 ctx.handler().getClass().getName() +  
 **".handlerAdded() has thrown an exception; removed."**, t));  
 } **else** {  
 fireExceptionCaught(**new** ChannelPipelineException(  
 ctx.handler().getClass().getName() +  
 **".handlerAdded() has thrown an exception; also failed to remove."**, t));  
 }  
 }  
}

@Override  
**public void** handlerAdded(ChannelHandlerContext ctx) **throws** Exception {  
 **if** (ctx.channel().isRegistered()) {  
 *// This should always be true with our current DefaultChannelPipeline implementation.  
 // The good thing about calling initChannel(...) in handlerAdded(...) is that there will be no ordering  
 // surprises if a ChannelInitializer will add another ChannelInitializer. This is as all handlers  
 // will be added in the expected order.* **initChannel(ctx);**  
 }  
}  
  
@SuppressWarnings(**"unchecked"**)  
**private boolean** initChannel(ChannelHandlerContext ctx) **throws** Exception {  
 **if** (**initMap**.putIfAbsent(ctx, Boolean.***TRUE***) == **null**) { *// Guard against re-entrance.* **try** {

**//ctx.channel()返回的可能是NioServerSocketChannel也可能是**

**//NioSocketChannel,在这里是NioSocketChannel** **initChannel((C) ctx.channel());**  
 } **catch** (Throwable cause) {  
 *// Explicitly call exceptionCaught(...) as we removed the handler before calling initChannel(...).  
 // We do so to prevent multiple calls to initChannel(...).* exceptionCaught(ctx, cause);  
 } **finally** {  
 remove(ctx);  
 }  
 **return true**;  
 }  
 **return false**;  
}

###### safeSetSuccess(promise)

***//将指定的promise标记为成功。如果promise已经完成，则记录一条消息*protected final void** safeSetSuccess(ChannelPromise promise) {  
 **if** (!(promise **instanceof** VoidChannelPromise) && !promise.trySuccess()) {  
 ***logger***.warn(**"Failed to mark a promise as success because it is done already: {}"**, promise);  
 }  
}

setSuccess0

@Override  
**public boolean** trySuccess(V result) {  
 **if** (**setSuccess0**(result)) {  
 **notifyListeners**();  
 **return true**;  
 }  
 **return false**;  
}

**private boolean** setSuccess0(V result) {  
 **return** setValue0(result == **null** ? ***SUCCESS*** : result);  
}  
  
**private boolean** setValue0(Object objResult) {  
 **if** (***RESULT\_UPDATER***.compareAndSet(**this**, **null**, objResult) ||  
 ***RESULT\_UPDATER***.compareAndSet(**this**, ***UNCANCELLABLE***, objResult)) {  
 checkNotifyWaiters();  
 **return true**;  
 }  
 **return false**;  
}

**private synchronized void** checkNotifyWaiters() {  
 **if** (**waiters** > 0) {  
 notifyAll();  
 }  
}

notifyListeners

**private void** notifyListeners() {  
 EventExecutor executor = executor();  
 **if** (executor.inEventLoop()) {  
 **final** InternalThreadLocalMap threadLocals = InternalThreadLocalMap.*get*();  
 **final int** stackDepth = threadLocals.futureListenerStackDepth();

//***MAX\_LISTENER\_STACK\_DEPTH ：*** 最大的监听栈深度  
 **if** (stackDepth < ***MAX\_LISTENER\_STACK\_DEPTH***) {

//设置**futureListenerStackDepth**  
 threadLocals.setFutureListenerStackDepth(stackDepth + 1);  
 **try** {  
 **notifyListenersNow();** } **finally** {  
 threadLocals.setFutureListenerStackDepth(stackDepth);  
 }  
 **return**;  
 }  
 }  
  
 *safeExecute*(executor, **new** Runnable() {  
 @Override  
 **public void** run() {  
 notifyListenersNow();  
 }  
 });  
}

**private void** notifyListenersNow() {  
 Object listeners;  
 **synchronized** (**this**) {  
 *// Only proceed if there are listeners to notify and we are not already notifying listeners.*

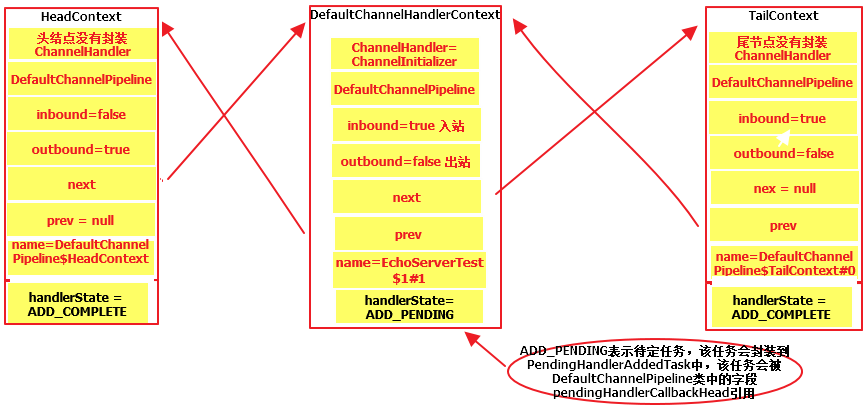
***//listeners是一个匿名类如下图*****if** (**notifyingListeners** || **this**.**listeners** == **null**) {  
 **return**;  
 }  
 **notifyingListeners** = **true**;  
 listeners = **this**.**listeners**;  
 **this**.**listeners** = **null**;  
 }  
 **for** (;;) {  
 **if** (listeners **instanceof** DefaultFutureListeners) {  
 notifyListeners0((DefaultFutureListeners) listeners);  
 } **else** {  
 ***notifyListener0*(this, (GenericFutureListener<?>) listeners);** }  
 **synchronized** (**this**) {  
 **if** (**this**.**listeners** == **null**) {  
 *// Nothing can throw from within this method, so setting notifyingListeners back to false does not  
 // need to be in a finally block.* **notifyingListeners** = **false**;  
 **return**;  
 }  
 listeners = **this**.**listeners**;  
 **this**.**listeners** = **null**;  
 }  
 }  
}



@SuppressWarnings({ **"unchecked"**, **"rawtypes"** })  
**private static void** notifyListener0(Future future, GenericFutureListener l) {  
 **try** {  
 l.operationComplete(future);  
 } **catch** (Throwable t) {  
 ***logger***.warn(**"An exception was thrown by "** + l.getClass().getName() + **".operationComplete()"**, t);  
 }  
}

###### pipeline.fireChannelRegistered()

Pipeline链表结构



**public final** ChannelPipeline fireChannelRegistered() {  
 AbstractChannelHandlerContext.*invokeChannelRegistered*(**head**);  
 **return this**;  
}

**static void** invokeChannelRegistered(**final** AbstractChannelHandlerContext next) {  
 EventExecutor executor = next.executor();  
 **if** (executor.inEventLoop()) {  
 next.invokeChannelRegistered();  
 } **else** {  
 executor.execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 next.invokeChannelRegistered();  
 }  
 });  
 }  
}

**private void** invokeChannelRegistered() {  
 **if** (invokeHandler()) {  
 **try** {  
 ((ChannelInboundHandler) handler()).channelRegistered(**this**);  
 } **catch** (Throwable t) {  
 notifyHandlerException(t);  
 }  
 } **else** {  
 fireChannelRegistered();  
 }  
}

@Override  
**public void** channelRegistered(ChannelHandlerContext ctx) **throws** Exception {  
  **invokeHandlerAddedIfNeeded();  
 ctx.fireChannelRegistered();**}

invokeHandlerAddedIfNeeded

**final void** invokeHandlerAddedIfNeeded() {  
 **assert channel**.eventLoop().inEventLoop();  
 **if** (**firstRegistration**) {//  
 **firstRegistration** = **false**;  
 *// We are now registered to the EventLoop. It's time to call the callbacks for the ChannelHandlers,  
 // that were added before the registration was done.* callHandlerAddedForAllHandlers();  
 }  
}

ctx.fireChannelRegistered

@Override  
**public** ChannelHandlerContext fireChannelRegistered() {

**//从head向后找到下一个ChannnelHandlerContextInbound** *invokeChannelRegistered*(findContextInbound());  
 **return this**;  
}

**static void** invokeChannelRegistered(**final** AbstractChannelHandlerContext next) {  
 EventExecutor executor = next.executor();  
 **if** (executor.inEventLoop()) {  
 next.invokeChannelRegistered();  
 } **else** {  
 executor.execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 next.invokeChannelRegistered();  
 }  
 });  
 }  
}

**private void** invokeChannelRegistered() {  
 **if** (invokeHandler()) {  
 **try** {  
 ((ChannelInboundHandler) handler()).**channelRegistered**(**this**);  
 } **catch** (Throwable t) {  
 notifyHandlerException(t);  
 }  
 } **else** {  
 fireChannelRegistered();  
 }  
}

**public void** channelRegistered(ChannelHandlerContext ctx) **throws** Exception {  
 ctx.fireChannelRegistered();  
}

**/\*\*\*\*\*\*\*\*\*循环 找到尾节点\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***/

@Override  
**public** ChannelHandlerContext fireChannelRegistered() {

**//从head向后找到下一个ChannnelHandlerContextInbound** *invokeChannelRegistered*(findContextInbound());  
 **return this**;  
}

**private** AbstractChannelHandlerContext findContextInbound() {  
 AbstractChannelHandlerContext ctx = **this**;  
 **do** {  
 ctx = ctx.**next**;  
 } **while** (!ctx.**inbound**);  
 **return** ctx;  
}

**static void** invokeChannelRegistered(**final** AbstractChannelHandlerContext next) {  
 EventExecutor executor = next.executor();  
 **if** (executor.inEventLoop()) {  
 next.invokeChannelRegistered();  
 } **else** {  
 executor.execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 next.invokeChannelRegistered();  
 }  
 });  
 }  
}

**private void** invokeChannelRegistered() {  
 **if** (invokeHandler()) {  
 **try** {  
 ((ChannelInboundHandler) handler()).channelRegistered(**this**);  
 } **catch** (Throwable t) {  
 notifyHandlerException(t);  
 }  
 } **else** {  
 fireChannelRegistered();  
 }  
}

**/\*尾节点channelRegistered空实现\*/**

@Override  
**public void** channelRegistered(ChannelHandlerContext ctx) **throws** Exception { }

###### pipeline.fireChannelActive() 就是OP\_READ注册

# OP\_READ注册

**首先，封装SocketChannel的NioSocketChannel创建时就指定感兴趣事件OP\_READ; 然后，在注册阶段，直接向Selector注册OP\_READ**

## register0

**private void** register0(ChannelPromise promise) {  
 **try** {  
 *// check if the channel is still open as it could be closed in the mean time when the register  
 // call was outside of the eventLoop* **if** (!promise.setUncancellable() || !ensureOpen(promise)) {  
 **return**;  
 }  
 **boolean** firstRegistration = **neverRegistered**;

**//把socketChannel注册到Selector**  
 doRegister();  
 **neverRegistered** = **false**;  
 **registered** = **true**;

***//在我们实际通知promise之前，确保我们先调用handlerAdded(),该方法最***

***//终会调用ChannelInitializer的initChannel()来将ChannelHander封装***

***//成ChannelHandlerContext添加到pipeline的链表。***

***//这是必须的，因为用户可能已经通过ChannelFutureListener中的pipeline***

***//触发了事件*****pipeline**.invokeHandlerAddedIfNeeded();  
   
 safeSetSuccess(promise);  
 **pipeline**.fireChannelRegistered();  
 *//只有在channel没有注册的时候才触发channelActive。这可以防止在*

*//取消注册和重新注册通道时触发多个通道活动*

***//注意isActive的实现，在NioServerSocketChannel中的实现与在***

***//NioSocketChannel中的实现不同*****if** (isActive()) {  
 **if** (**firstRegistration**) {

**//注册OP\_ACCEPT或OP\_READ事件**

**//1.在给ServerSocketChannel注册OP\_ACCEPT事件的时候可能在此步骤完成，也**

**//可能在task任务中完成的**

**//2.在给SocketChannel注册OP\_READ事件的时候是在此步骤完成的**  
 **pipeline.fireChannelActive();** } **else if** (config().isAutoRead()) {  
 *// This channel was registered before and autoRead() is set. This means we need to begin read  
 // again so that we process inbound data.  
 //  
 // See https://github.com/netty/netty/issues/4805* beginRead();  
 }  
 }  
 } **catch** (Throwable t) {  
 *// Close the channel directly to avoid FD leak.* closeForcibly();  
 **closeFuture**.setClosed();  
 safeSetFailure(promise, t);  
 }  
}

**在这里要特别注意isActive()的实现**

**在NioServerSocketChannel中的实现如下：**

**public boolean** isActive() {

//判断ServerSocketChannel有没有绑定端口  
 **return** javaChannel().socket().isBound();  
}

**在NioSocKetChannel中的实现如下：**

**public boolean** isActive() {  
 SocketChannel ch = javaChannel();

**//1.isOpen 判断此通道是否处于打开状态**

**//2.isConnected()指示此通道网络套接字是否已连接，当且仅当此通道网络套接字**

**//为打开并且连接的时候返回true,也就是说isConnected为true则isOpen必为**

**//true** **return** ch.isOpen() && ch.isConnected();  
}

### pipeline.fireChannelActive()

@Override  
**public final** ChannelPipeline fireChannelActive() {  
 AbstractChannelHandlerContext.*invokeChannelActive*(**head**);  
 **return this**;  
}

**static void** invokeChannelActive(**final** AbstractChannelHandlerContext next) {  
 EventExecutor executor = next.executor();  
 **if** (executor.inEventLoop()) {  
 next.invokeChannelActive();  
 } **else** {  
 executor.execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 next.invokeChannelActive();  
 }  
 });  
 }  
}

**private void** invokeChannelActive() {  
 **if** (invokeHandler()) {  
 **try** {

**//headler()和this 都是headContext** ((ChannelInboundHandler) handler()).channelActive(**this**);  
 } **catch** (Throwable t) {  
 notifyHandlerException(t);  
 }  
 } **else** {  
 fireChannelActive();  
 }  
}

@Override  
**public void** channelActive(ChannelHandlerContext ctx) **throws** Exception {  
 ctx.fireChannelActive();  
 **//设置通道SocketChannel感兴趣的事情**  
 readIfIsAutoRead();  
}

#### ctx.fireChannelActive() 循环

**head ->封装EchoServerHandler的ChannelHandlerContext -> tail**

**最后遍历到尾节点 空实现**

@Override  
**public** ChannelHandlerContext fireChannelActive() {

*invokeChannelActive*(findContextInbound());  
 **return this**;  
}

**static void** invokeChannelActive(**final** AbstractChannelHandlerContext next) {  
 EventExecutor executor = next.executor();  
 **if** (executor.inEventLoop()) {  
 next.invokeChannelActive();  
 } **else** {  
 executor.execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 next.invokeChannelActive();  
 }  
 });  
 }  
}

**private void** invokeChannelActive() {  
 **if** (invokeHandler()) {  
 **try** {  
 ((ChannelInboundHandler) handler()).**channelActive**(**this**);  
 } **catch** (Throwable t) {  
 notifyHandlerException(t);  
 }  
 } **else** {  
 fireChannelActive();  
 }  
}

**public void** channelActive(ChannelHandlerContext ctx) **throws** Exception {  
 ctx.fireChannelActive();  
}

@Override  
**public void** channelActive(ChannelHandlerContext ctx) **throws** Exception {  
 onUnhandledInboundChannelActive();  
}

**//空实现**

**protected void** onUnhandledInboundChannelActive() {  
}

#### readIfIsAutoRead()

**private void** readIfIsAutoRead() {  
 **if** (**channel**.config().isAutoRead()) {  
 **channel**.read();  
 }  
}

**public** Channel read() {  
 **pipeline**.read();

**//this就是NioSocketChannel** **return this**;  
}

**public final** ChannelPipeline read() {  
 **tail**.**read();**  
 **return this**;  
}

**public** ChannelHandlerContext read() {

//向前找到第一个ChannelHandlerContextOutbound,头节点head是OutBound  
 **final** AbstractChannelHandlerContext next = findContextOutbound();  
 EventExecutor executor = next.executor();  
 **if** (executor.inEventLoop()) {

//next就是headContext  
 next.**invokeRead();**  
 } **else** {  
 Runnable task = next.**invokeReadTask**;  
 **if** (task == **null**) {  
 next.**invokeReadTask** = task = **new** Runnable() {  
 @Override  
 **public void** run() {  
 next.invokeRead();  
 }  
 };  
 }  
 executor.execute(task);  
 }  
   
 **return this**;  
}

**private void** invokeRead() {  
 **if** (invokeHandler()) {  
 **try** {

//handler()和this 是head  
 ((ChannelOutboundHandler) handler()).read(**this**);  
 } **catch** (Throwable t) {  
 notifyHandlerException(t);  
 }  
 } **else** {  
 read();  
 }  
}

@Override  
**public void** read(ChannelHandlerContext ctx) {  
 **unsafe**.beginRead();  
}

@Override  
**public final void** beginRead() {  
 assertEventLoop();  
  
 **if** (!isActive()) {  
 **return**;  
 }  
  
 **try** {  
 **doBeginRead();** } **catch** (**final** Exception e) {  
 invokeLater(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **pipeline**.fireExceptionCaught(e);  
 }  
 });  
 close(voidPromise());  
 }  
}

@Override  
**protected void** doBeginRead() **throws** Exception {  
 *// Channel.read() or ChannelHandlerContext.read() was called* **final** SelectionKey selectionKey = **this**.**selectionKey**;  
 **if** (!selectionKey.isValid()) {  
 **return**;  
 }  
  
 **readPending** = **true**;  
  
 **final int** interestOps = selectionKey.interestOps();  
 **if** ((interestOps & **readInterestOp**) == 0) {

//在NioSocketChannel中readInterestOp是初始化为OP\_READ  
 selectionKey.interestOps(interestOps | **readInterestOp**);  
 }  
}

# OP\_READ事件

## NioEventLoop.run

@Override  
**protected void** run() {  
 **for** (;;) {  
 **try** {  
 **switch** (**selectStrategy**.calculateStrategy(**selectNowSupplier**, hasTasks())) {  
 **case** SelectStrategy.***CONTINUE***:  
 **continue**;  
 **case** SelectStrategy.***SELECT***:

**//1.wakenUp表示是否应该唤醒正在阻塞的select操作，可以**

**//看到netty在进行一次新的loop之前，都会将wakenUp设置成**

**//false,标志新的一轮loop的开始**

**//2.wakenUp.getAndSet(false)返回可能是true，也可能**

**//是false,当是必须将wakenUp设置成false之后才返回。** select(**wakenUp**.getAndSet(**false**));  
  
 **//'wakenUp.compareAndSet(false, true)'总是在调用**

**//'selector.wakeup()'之前执行，以降低wake-up开销。**

**//(Selector.wakeup()是一个开销很大的操作。)**

**//然而，在这种方法中存在一个竞争条件。当wakenUp过早**

**//设置为true时，将触发竞争条件。**

**//如果wakenUp过早被设置为true:**

**//(1) Selector在'wakenUp.set(false)'和//'selector.select(...)'之间被唤醒。(BAD)**

**//(2)Selector在'selector.select(...)'和'if //(wakenUp.get()) { ... }'之间被唤醒。(OK)**

**//在第一种情况下，'wakenUp'被设置为true，下面的//'selector.select(…)'将立即唤醒。**

**//直到'wakenUp'在下一轮中再次设置为//false,'wakenUp.compareAndSet(false, true)'将会失败，//因此，任何唤醒Selector尝试也会失败，导致下面//'selector.select(...)'调用不必要的阻塞**

**//为了解决这个问题，如果在selector.select(…)之后，**

**//wakenUp为true，则我们再次唤醒Selector。**

**//因为第一种情况(BAD - 需要唤醒)和第二种情况**

**//的选择器(OK - 不需要唤醒)都唤醒,效率会很低。**

**//第一种情况 因为马上就要进行selector.select(),所以应该**

**//唤醒。第二情况 因为刚进行过selector.select()操作，所以**

**//不需要被唤醒**

**if** (**wakenUp**.get()) {  
 **selector**.wakeup();  
 }  
 *// fall through* **default**:  
 }  
  
 **cancelledKeys** = 0;  
 **needsToSelectAgain** = **false**;  
 **final int** ioRatio = **this**.**ioRatio**;  
 **if** (ioRatio == 100) {  
 **try** {  
 processSelectedKeys();  
 } **finally** {  
 *// Ensure we always run tasks.* runAllTasks();  
 }  
 } **else** {  
 **final long** ioStartTime = System.*nanoTime*();  
 **try** {  
 **processSelectedKeys();**  
 } **finally** {  
 *// Ensure we always run tasks.* **final long** ioTime = System.*nanoTime*() - ioStartTime;  
 runAllTasks(ioTime \* (100 - ioRatio) / ioRatio);  
 }  
 }  
 } **catch** (Throwable t) {  
 *handleLoopException*(t);  
 }  
 *// Always handle shutdown even if the loop processing threw an exception.* **try** {  
 **if** (isShuttingDown()) {  
 closeAll();  
 **if** (confirmShutdown()) {  
 **return**;  
 }  
 }  
 } **catch** (Throwable t) {  
 *handleLoopException*(t);  
 }  
 }  
}

**private void** processSelectedKeys() {  
 **if** (**selectedKeys** != **null**) {  
 **processSelectedKeysOptimized();** } **else** {  
 processSelectedKeysPlain(**selector**.selectedKeys());  
 }  
}

**private void** processSelectedKeysOptimized() {  
 **for** (**int** i = 0; i < **selectedKeys**.**size**; ++i) {  
 **final** SelectionKey k = **selectedKeys**.**keys**[i];  
 *// null out entry in the array to allow to have it GC'ed once the Channel close  
 // See https://github.com/netty/netty/issues/2363* **selectedKeys**.**keys**[i] = **null**;  
  
 **final** Object a = k.attachment();  
  
 **if** (a **instanceof** AbstractNioChannel) {  
 **processSelectedKey(k, (AbstractNioChannel) a);**  
 } **else** {  
 @SuppressWarnings(**"unchecked"**)  
 NioTask<SelectableChannel> task = (NioTask<SelectableChannel>) a;  
 *processSelectedKey*(k, task);  
 }  
  
 **if** (**needsToSelectAgain**) {  
 *// null out entries in the array to allow to have it GC'ed once the Channel close  
 // See https://github.com/netty/netty/issues/2363* **selectedKeys**.reset(i + 1);  
  
 selectAgain();  
 i = -1;  
 }  
 }  
}

**private void** processSelectedKey(SelectionKey k, AbstractNioChannel ch) {  
 **final** AbstractNioChannel.NioUnsafe unsafe = ch.unsafe();  
 **if** (!k.isValid()) {  
 **final** EventLoop eventLoop;  
 **try** {  
 eventLoop = ch.eventLoop();  
 } **catch** (Throwable ignored) {  
 *// If the channel implementation throws an exception because there is no event loop, we ignore this  
 // because we are only trying to determine if ch is registered to this event loop and thus has authority  
 // to close ch.* **return**;  
 }  
 *// Only close ch if ch is still registered to this EventLoop. ch could have deregistered from the event loop  
 // and thus the SelectionKey could be cancelled as part of the deregistration process, but the channel is  
 // still healthy and should not be closed.  
 // See https://github.com/netty/netty/issues/5125* **if** (eventLoop != **this** || eventLoop == **null**) {  
 **return**;  
 }  
 *// close the channel if the key is not valid anymore* unsafe.close(unsafe.voidPromise());  
 **return**;  
 }  
  
 **try** {  
 **int** readyOps = k.readyOps();  
 *// We first need to call finishConnect() before try to trigger a read(...) or write(...) as otherwise  
 // the NIO JDK channel implementation may throw a NotYetConnectedException.* **if** ((readyOps & SelectionKey.***OP\_CONNECT***) != 0) {  
 *// remove OP\_CONNECT as otherwise Selector.select(..) will always return without blocking  
 // See https://github.com/netty/netty/issues/924* **int** ops = k.interestOps();  
 ops &= ~SelectionKey.***OP\_CONNECT***;  
 k.interestOps(ops);  
  
 unsafe.finishConnect();  
 }  
  
 *// Process OP\_WRITE first as we may be able to write some queued buffers and so free memory.* **if** ((readyOps & SelectionKey.***OP\_WRITE***) != 0) {  
 *// Call forceFlush which will also take care of clear the OP\_WRITE once there is nothing left to write* ch.unsafe().forceFlush();  
 }  
  
 *// Also check for readOps of 0 to workaround possible JDK bug which may otherwise lead  
 // to a spin loop* **if** ((readyOps & (SelectionKey.***OP\_READ*** | SelectionKey.***OP\_ACCEPT***)) != 0 || readyOps == 0) {  
 **unsafe.read();** }  
 } **catch** (CancelledKeyException ignored) {  
 unsafe.close(unsafe.voidPromise());  
 }  
}

**//注意：NioSocketChannel的read()方法实现与NioServerSocketChannel的read()**

**//方法实现不同**  
**public final void** read() {  
 **final** ChannelConfig config = config();  
 **if** (shouldBreakReadReady(config)) {  
 clearReadPending();  
 **return**;  
 }

**//返回NioSocketChannel关联的pipeline** **final** ChannelPipeline pipeline = pipeline();

**//实现负责分配缓冲区。这个接口的实现应该是线程安全的。** **final** ByteBufAllocator allocator = config.getAllocator();  
 **final** RecvByteBufAllocator.Handle allocHandle = recvBufAllocHandle();  
 allocHandle.reset(config);  
  
 ByteBuf byteBuf = **null**;  
 **boolean** close = **false**;  
 **try** {  
 **do** {

**//分配字节缓冲区** byteBuf = allocHandle.allocate(allocator);  
 allocHandle.lastBytesRead(doReadBytes(byteBuf));  
 **if** (allocHandle.lastBytesRead() <= 0) {  
 ***// 没有读任何数据，释放缓冲区.***byteBuf.release();  
 byteBuf = **null**;  
 close = allocHandle.lastBytesRead() < 0;  
 **if** (close) {  
 *//* 当我们收到EOF时，没有什么可读的了*.* **readPending** = **false**;  
 }  
 **break**;  
 }  
  
 allocHandle.incMessagesRead(1);  
 **readPending** = **false**;

**pipeline.fireChannelRead(byteBuf);**  
 byteBuf = **null**;  
 } **while** (allocHandle.continueReading());  
  
 allocHandle.readComplete();  
 **pipeline.fireChannelReadComplete();**  
 **if** (close) {  
 closeOnRead(pipeline);  
 }  
 } **catch** (Throwable t) {  
 handleReadException(pipeline, byteBuf, t, close, allocHandle);  
 } **finally** {  
 *// Check if there is a readPending which was not processed yet.  
 // This could be for two reasons:  
 // \* The user called Channel.read() or ChannelHandlerContext.read() in channelRead(...) method  
 // \* The user called Channel.read() or ChannelHandlerContext.read() in channelReadComplete(...) method  
 //  
 // See https://github.com/netty/netty/issues/2254* **if** (!**readPending** && !config.isAutoRead()) {  
 removeReadOp();  
 }  
 }  
 }  
}

### select(wakenUp.getAndSet(false));

**private void** select(**boolean** oldWakenUp) **throws** IOException {  
 Selector selector = **this**.**selector**;  
 **try** {  
 **int** selectCnt = 0;  
 **long** currentTimeNanos = System.*nanoTime*();

**//delayNanos(currentTimeNanos)方法即取任务队列中第一个定时任务的**

**//剩余延迟时间，该返回值大于等于0** **long** selectDeadLineNanos = currentTimeNanos + delayNanos(currentTimeNanos);  
 **for** (;;) {  
 **long** timeoutMillis = (selectDeadLineNanos - currentTimeNanos + 500000L) / 1000000L;

**//定时任务队列中有任务的截止时间快到了(<=0.5s)，就跳出循环** **if** (timeoutMillis <= 0) {

//跳出之前如果发现目前为止还没有进行过select操作(selectCnt等

//于0，则说明还没有进行select操作)，那么就调用一次selectNow()，

//该方法会立即返回，不会阻塞  
 **if** (selectCnt == 0) {  
 selector.selectNow();  
 selectCnt = 1;  
 }  
 **break**;  
 }

***//1.轮询过程中发现有任务加入，中断本次轮询***

***//2.当wakenUp为true时如果提交了一个任务，那么该任务没有机会调用***

***//Selector.wakeup()。因此在执行selector.selectNow()操作前我们***

***//需要再次检查任务队列。***

***//3.如果我们没有检查任务队列，那么任务可能会被搁置直到selector.***

***//selectNow()操作超时为止。***

***//4.如果在pipeline中存在IdleStateHandler，它可能搁置直到超出***

***//空闲时间。*****if** (hasTasks() && **wakenUp**.compareAndSet(**false**, **true**)) {

//非阻塞式select操作  
 selector.selectNow();  
 selectCnt = 1;  
 **break**;  
 }  
 //阻塞式select操作，执行到这一步，说明netty任务队列里面任务为空，

//并且所有定时任务延迟时间还为到(大于0.5s),于是，在这里进行一次阻

//塞select操作，截止到第一个定时任务的截止时间。  
 **int** selectedKeys = selector.select(timeoutMillis);  
 selectCnt ++;  
 //1.轮询到IO事件(selectedKeys != 0)

//2.oldWakenUp参数为true

//3.任务队列里面有任务(hasTasks)

//4.第一个定时任务延迟时间到了需要被执行(hasScheduledTasks)

//5.用户主动唤醒  
 **if** (selectedKeys != 0 || oldWakenUp || **wakenUp**.get() || hasTasks() || hasScheduledTasks()) {  
 *// - Selected something,  
 // - waken up by user, or  
 // - the task queue has a pending task.  
 // - a scheduled task is ready for processing* **break**;  
 }  
 **if** (Thread.*interrupted*()) {  
 *// Thread was interrupted so reset selected keys and break so we not run into a busy loop.  
 // As this is most likely a bug in the handler of the user or it's client library we will  
 // also log it.  
 //  
 // See https://github.com/netty/netty/issues/2426* **if** (***logger***.isDebugEnabled()) {  
 ***logger***.debug(**"Selector.select() returned prematurely because "** +  
 **"Thread.currentThread().interrupt() was called. Use "** +  
 **"NioEventLoop.shutdownGracefully() to shutdown the NioEventLoop."**);  
 }  
 selectCnt = 1;  
 **break**;  
 }  
  
 **long** time = System.*nanoTime*();  
 **if** (time - TimeUnit.***MILLISECONDS***.toNanos(timeoutMillis) >= currentTimeNanos) {  
 *// timeoutMillis elapsed without anything selected.* selectCnt = 1;  
 } **else if** (***SELECTOR\_AUTO\_REBUILD\_THRESHOLD*** > 0 &&  
 selectCnt >= ***SELECTOR\_AUTO\_REBUILD\_THRESHOLD***) {  
 *// The selector returned prematurely many times in a row.  
 // Rebuild the selector to work around the problem.* ***logger***.warn(  
 **"Selector.select() returned prematurely {} times in a row; rebuilding Selector {}."**,  
 selectCnt, selector);  
  
 rebuildSelector();  
 selector = **this**.**selector**;  
  
 *// Select again to populate selectedKeys.* selector.selectNow();  
 selectCnt = 1;  
 **break**;  
 }  
  
 currentTimeNanos = time;  
 }  
  
 **if** (selectCnt > ***MIN\_PREMATURE\_SELECTOR\_RETURNS***) {  
 **if** (***logger***.isDebugEnabled()) {  
 ***logger***.debug(**"Selector.select() returned prematurely {} times in a row for Selector {}."**,  
 selectCnt - 1, selector);  
 }  
 }  
 } **catch** (CancelledKeyException e) {  
 **if** (***logger***.isDebugEnabled()) {  
 ***logger***.debug(CancelledKeyException.**class**.getSimpleName() + **" raised by a Selector {} - JDK bug?"**,  
 selector, e);  
 }  
 *// Harmless exception - log anyway* }  
}

### pipeline.fireChannelRead(byteBuf);

**headContext -> 封装EchoServerHandler的ChannelHandlerContext**

**从头节点HeadContext开始遍历pipeline链表最终找到封装EchoServerHandler的ChannelHandlerContext**  
**public final** ChannelPipeline fireChannelRead(Object msg) {  
 AbstractChannelHandlerContext.*invokeChannelRead*(**head**, msg);  
 **return this**;  
}

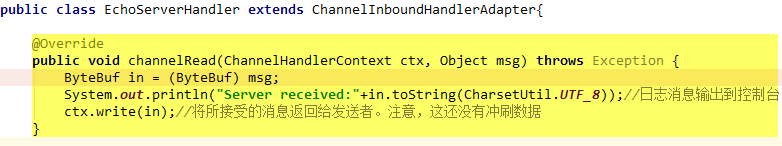
**static void** invokeChannelRead(**final** AbstractChannelHandlerContext next, Object msg) {  
 **final** Object m = next.**pipeline**.touch(ObjectUtil.*checkNotNull*(msg, **"msg"**), next);  
 EventExecutor executor = next.executor();  
 **if** (executor.inEventLoop()) {  
 next.invokeChannelRead(m);  
 } **else** {  
 executor.execute(**new** Runnable() {  
 @Override  
 **public void** run() {  
 next.invokeChannelRead(m);  
 }  
 });  
 }  
}

**private void** invokeChannelRead(Object msg) {  
 **if** (invokeHandler()) {  
 **try** {  
 ((ChannelInboundHandler) handler()).channelRead(**this**, msg);  
 } **catch** (Throwable t) {  
 notifyHandlerException(t);  
 }  
 } **else** {  
 fireChannelRead(msg);  
 }  
}

**public void** channelRead(ChannelHandlerContext ctx, Object msg) **throws** Exception {  
 ctx.fireChannelRead(msg);  
}

**//HeadContext的fireChannelRead的实现是找到下一个pipeline链表节点, 在这里//即是封装了EchoServerHandler的ChannnelHandlerContext,因此会调用到**

**//EchoServerHandler的channnelRead()方法，该方法实现如下：**



**public** ChannelHandlerContext fireChannelRead(**final** Object msg) {  
 *invokeChannelRead*(findContextInbound(), msg);  
 **return this**;  
}

**//最终调用到了EchoServerHandler的channelRead()方法**

@Override  
**public void** channelRead(ChannelHandlerContext ctx, Object msg) **throws** Exception {  
 ByteBuf in = (ByteBuf) msg;

**//控制台输出一条消息**  
 System.***out***.println(**"Server received:"**+in.toString(CharsetUtil.***UTF\_8***));*//日志消息输出到控制台* **ctx.write(in);***//将所接受的消息返回给发送者。注意，这还没有冲刷数据*}

#### ctx.write(in)

**public** ChannelFuture write(Object msg) {  
 **return write**(msg, **newPromise()**);  
}

##### newPromise()

**public** ChannelPromise newPromise() {

//channel()就是 NioSocketChannel  
 **return new** DefaultChannelPromise(channel(), executor());  
}

##### write

**public** ChannelFuture write(**final** Object msg, **final** ChannelPromise promise) {  
 **if** (msg == **null**) {  
 **throw new** NullPointerException(**"msg"**);  
 }  
  
 **try** {  
 **if** (isNotValidPromise(promise, **true**)) {  
 ReferenceCountUtil.*release*(msg);  
 *// cancelled* **return** promise;  
 }  
 } **catch** (RuntimeException e) {  
 ReferenceCountUtil.*release*(msg);  
 **throw** e;  
 }  
 write(msg, **false**, promise);  
  
 **return** promise;  
}

**private void** write(Object msg, **boolean** flush, ChannelPromise promise) {

**//findContextOutbound()找到pipeline链表的头节点HeadContext** AbstractChannelHandlerContext next = findContextOutbound();  
 **final** Object m = **pipeline**.touch(msg, next);  
 EventExecutor executor = next.executor();  
 **if** (executor.inEventLoop()) {  
 **if** (flush) {  
 next.invokeWriteAndFlush(m, promise);  
 } **else** {  
 **next.invokeWrite(m, promise);** }  
 } **else** {  
 AbstractWriteTask task;  
 **if** (flush) {  
 task = WriteAndFlushTask.*newInstance*(next, m, promise);  
 } **else** {  
 task = WriteTask.*newInstance*(next, m, promise);  
 }  
 *safeExecute*(executor, task, promise, m);  
 }  
}

**private void** invokeWrite(Object msg, ChannelPromise promise) {  
 **if** (invokeHandler()) {  
 **invokeWrite0(msg, promise);** } **else** {  
 write(msg, promise);  
 }  
}

#### invokeWrite0

**private void** invokeWrite0(Object msg, ChannelPromise promise) {  
 **try** {  
 ((ChannelOutboundHandler) handler()).write(**this**, msg, promise);  
 } **catch** (Throwable t) {  
 *notifyOutboundHandlerException*(t, promise);  
 }  
}

**public void** write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) **throws** Exception {  
 **unsafe**.**write(msg, promise);**}

@Override  
**public final void** write(Object msg, ChannelPromise promise) {  
 assertEventLoop();  
  
 ChannelOutboundBuffer outboundBuffer = **this**.**outboundBuffer**;  
 **if** (outboundBuffer == **null**) {  
 *// If the outboundBuffer is null we know the channel was closed and so  
 // need to fail the future right away. If it is not null the handling of the rest  
 // will be done in flush0()  
 // See https://github.com/netty/netty/issues/2362* safeSetFailure(promise, ***WRITE\_CLOSED\_CHANNEL\_EXCEPTION***);  
 *// release message now to prevent resource-leak* ReferenceCountUtil.*release*(msg);  
 **return**;  
 }  
  
 **int** size;  
 **try** {  
 msg = filterOutboundMessage(msg);  
 size = **pipeline**.estimatorHandle().size(msg);  
 **if** (size < 0) {  
 size = 0;  
 }  
 } **catch** (Throwable t) {  
 safeSetFailure(promise, t);  
 ReferenceCountUtil.*release*(msg);  
 **return**;  
 }  
  
 outboundBuffer.addMessage(msg, size, promise);  
}

#### invokeFlush0()

**private void** invokeFlush0() {  
 **try** {  
 ((ChannelOutboundHandler) handler()).flush(**this**);  
 } **catch** (Throwable t) {  
 notifyHandlerException(t);  
 }  
}

**public void** flush(ChannelHandlerContext ctx) **throws** Exception {  
 **unsafe**.flush();  
}

**public final void** flush() {  
 assertEventLoop();  
  
 ChannelOutboundBuffer outboundBuffer = **this**.**outboundBuffer**;  
 **if** (outboundBuffer == **null**) {  
 **return**;  
 }  
  
 outboundBuffer.addFlush();  
 flush0();  
}

**protected void** flush0() {  
 **if** (**inFlush0**) {  
 *// Avoid re-entrance* **return**;  
 }  
  
 **final** ChannelOutboundBuffer outboundBuffer = **this**.**outboundBuffer**;  
 **if** (outboundBuffer == **null** || outboundBuffer.isEmpty()) {  
 **return**;  
 }  
  
 **inFlush0** = **true**;  
  
 *// Mark all pending write requests as failure if the channel is inactive.* **if** (!isActive()) {  
 **try** {  
 **if** (isOpen()) {  
 outboundBuffer.failFlushed(***FLUSH0\_NOT\_YET\_CONNECTED\_EXCEPTION***, **true**);  
 } **else** {  
 *// Do not trigger channelWritabilityChanged because the channel is closed already.* outboundBuffer.failFlushed(***FLUSH0\_CLOSED\_CHANNEL\_EXCEPTION***, **false**);  
 }  
 } **finally** {  
 **inFlush0** = **false**;  
 }  
 **return**;  
 }  
  
 **try** {  
 **doWrite(outboundBuffer);** } **catch** (Throwable t) {  
 **if** (t **instanceof** IOException && config().isAutoClose()) {  
 */\*\*  
 \* Just call {****@link*** *#close(ChannelPromise, Throwable, boolean)} here which will take care of  
 \* failing all flushed messages and also ensure the actual close of the underlying transport  
 \* will happen before the promises are notified.  
 \*  
 \* This is needed as otherwise {****@link*** *#isActive()} , {****@link*** *#isOpen()} and {****@link*** *#isWritable()}  
 \* may still return {****@code*** *true} even if the channel should be closed as result of the exception.  
 \*/* close(voidPromise(), t, ***FLUSH0\_CLOSED\_CHANNEL\_EXCEPTION***, **false**);  
 } **else** {  
 **try** {  
 shutdownOutput(voidPromise(), t);  
 } **catch** (Throwable t2) {  
 close(voidPromise(), t2, ***FLUSH0\_CLOSED\_CHANNEL\_EXCEPTION***, **false**);  
 }  
 }  
 } **finally** {  
 **inFlush0** = **false**;  
 }  
}

@Override  
**protected void** doWrite(ChannelOutboundBuffer in) **throws** Exception {  
 **int** writeSpinCount = config().getWriteSpinCount();  
 **do** {  
 Object msg = in.current();  
 **if** (msg == **null**) {  
 *// Wrote all messages.* clearOpWrite();  
 *// Directly return here so incompleteWrite(...) is not called.* **return**;  
 }  
 writeSpinCount -= doWriteInternal(in, msg);  
 } **while** (writeSpinCount > 0);  
  
 incompleteWrite(writeSpinCount < 0);  
}

### pipeline.fireChannelReadComplete()

**HeadContext -> 封装EchoServerHandler的ChannelHandlerContext**

**从头节点HeadContext开始遍历pipeline链表最终找到封装EchoServerHandler**

**的ChannelHandlerContext**

**public final** ChannelPipeline fireChannelReadComplete() {  
 AbstractChannelHandlerContext.*invokeChannelReadComplete*(**head**);  
 **return this**;  
}

**static void** invokeChannelReadComplete(**final** AbstractChannelHandlerContext next) {  
 EventExecutor executor = next.executor();  
 **if** (executor.inEventLoop()) {  
 next.invokeChannelReadComplete();  
 } **else** {  
 Runnable task = next.**invokeChannelReadCompleteTask**;  
 **if** (task == **null**) {  
 next.**invokeChannelReadCompleteTask** = task = **new** Runnable() {  
 @Override  
 **public void** run() {  
 next.invokeChannelReadComplete();  
 }  
 };  
 }  
 executor.execute(task);  
 }  
}

**private void** invokeChannelReadComplete() {  
 **if** (invokeHandler()) {  
 **try** {  
 ((ChannelInboundHandler) handler()).channelReadComplete(**this**);  
 } **catch** (Throwable t) {  
 notifyHandlerException(t);  
 }  
 } **else** {  
 fireChannelReadComplete();  
 }  
}

**//最终找到封装EchoServerHandler的ChannelHandlerContext**

**public void** channelReadComplete(ChannelHandlerContext ctx) **throws** Exception {  
 ctx.**writeAndFlush**(Unpooled.***EMPTY\_BUFFER***) *//冲刷所有待审消息到远程节点。关闭通道后，操作完成。* .addListener(ChannelFutureListener.***CLOSE***);  
}



#### ctx.writeAndFlush(Unpooled.***EMPTY\_BUFFER***)

**public** ChannelFuture writeAndFlush(Object msg) {  
 **return writeAndFlush**(msg, **newPromise()**);  
}

**public** ChannelPromise newPromise() {  
 **return new** DefaultChannelPromise(channel(), executor());  
}

**public** ChannelFuture writeAndFlush(Object msg, ChannelPromise promise) {  
 **if** (msg == **null**) {  
 **throw new** NullPointerException(**"msg"**);  
 }  
  
 **if** (isNotValidPromise(promise, **true**)) {  
 ReferenceCountUtil.*release*(msg);  
 *// cancelled* **return** promise;  
 }  
  
 **write**(msg, **true**, promise);  
  
 **return** promise;  
}

**private void** write(Object msg, **boolean** flush, ChannelPromise promise) {

**//封装EchoServerHandler的ChannelHandlerContext -> HeadContext**

**//找到pipeline链表的头节点HeadContext** AbstractChannelHandlerContext next = findContextOutbound();  
 **final** Object m = **pipeline**.touch(msg, next);  
 EventExecutor executor = next.executor();  
 **if** (executor.inEventLoop()) {  
 **if** (flush) {

//next就是头节点headContext  
 next.invokeWriteAndFlush(m, promise);  
 } **else** {  
 next.invokeWrite(m, promise);  
 }  
 } **else** {  
 AbstractWriteTask task;  
 **if** (flush) {  
 task = WriteAndFlushTask.*newInstance*(next, m, promise);  
 } **else** {  
 task = WriteTask.*newInstance*(next, m, promise);  
 }  
 *safeExecute*(executor, task, promise, m);  
 }  
}

**private void** invokeWriteAndFlush(Object msg, ChannelPromise promise) {  
 **if** (invokeHandler()) {  
  **invokeWrite0(msg, promise);  
 invokeFlush0();** } **else** {  
 writeAndFlush(msg, promise);  
 }  
}

##### invokeWrite0(msg, promise);

**private void** invokeWrite0(Object msg, ChannelPromise promise) {  
 **try** {  
 ((ChannelOutboundHandler) handler()).write(**this**, msg, promise);  
 } **catch** (Throwable t) {  
 *notifyOutboundHandlerException*(t, promise);  
 }  
}

**public void** write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) **throws** Exception {  
 **unsafe**.write(msg, promise);  
}

**public final void** write(Object msg, ChannelPromise promise) {  
 assertEventLoop();  
  
 ChannelOutboundBuffer outboundBuffer = **this**.**outboundBuffer**;  
 **if** (outboundBuffer == **null**) {  
 *// If the outboundBuffer is null we know the channel was closed and so  
 // need to fail the future right away. If it is not null the handling of the rest  
 // will be done in flush0()  
 // See https://github.com/netty/netty/issues/2362* safeSetFailure(promise, ***WRITE\_CLOSED\_CHANNEL\_EXCEPTION***);  
 *// release message now to prevent resource-leak* ReferenceCountUtil.*release*(msg);  
 **return**;  
 }  
  
 **int** size;  
 **try** {  
 msg = filterOutboundMessage(msg);  
 size = **pipeline**.estimatorHandle().size(msg);  
 **if** (size < 0) {  
 size = 0;  
 }  
 } **catch** (Throwable t) {  
 safeSetFailure(promise, t);  
 ReferenceCountUtil.*release*(msg);  
 **return**;  
 }  
  
 **outboundBuffer.addMessage(msg, size, promise);**}

**public void** addMessage(Object msg, **int** size, ChannelPromise promise) {  
 Entry entry = Entry.*newInstance*(msg, size, *total*(msg), promise);  
 **if** (**tailEntry** == **null**) {  
 **flushedEntry** = **null**;  
 } **else** {  
 Entry tail = **tailEntry**;  
 tail.**next** = entry;  
 }  
 **tailEntry** = entry;  
 **if** (**unflushedEntry** == **null**) {  
 **unflushedEntry** = entry;  
 }  
  
 *// increment pending bytes after adding message to the unflushed arrays.  
 // See https://github.com/netty/netty/issues/1619* incrementPendingOutboundBytes(entry.**pendingSize**, **false**);  
}

##### invokeFlush0(); **private void** invokeFlush0() { **try** { ((ChannelOutboundHandler) handler()).flush(**this**); } **catch** (Throwable t) { notifyHandlerException(t); } }

**public void** flush(ChannelHandlerContext ctx) **throws** Exception {  
 **unsafe**.flush();  
}

**public final void** flush() {  
 assertEventLoop();  
  
 ChannelOutboundBuffer outboundBuffer = **this**.**outboundBuffer**;  
 **if** (outboundBuffer == **null**) {  
 **return**;  
 }  
  
 **outboundBuffer.addFlush();** ***flush0();***}

###### outboundBuffer.addFlush();

*/\*\*  
 \* Add a flush to this {****@link*** *ChannelOutboundBuffer}. This means all previous added messages are marked as flushed  
 \* and so you will be able to handle them.  
 \*/*

**public void** addFlush() {  
 *// There is no need to process all entries if there was already a flush before and no new messages  
 // where added in the meantime.  
 //  
 // See https://github.com/netty/netty/issues/2577* Entry entry = **unflushedEntry**;  
 **if** (entry != **null**) {  
 **if** (**flushedEntry** == **null**) {  
 *// there is no flushedEntry yet, so start with the entry* **flushedEntry** = entry;  
 }  
 **do** {  
 **flushed** ++;  
 **if** (!entry.**promise**.setUncancellable()) {  
 *// Was cancelled so make sure we free up memory and notify about the freed bytes* **int** pending = entry.cancel();  
 decrementPendingOutboundBytes(pending, **false**, **true**);  
 }  
 entry = entry.**next**;  
 } **while** (entry != **null**);  
  
 *// All flushed so reset unflushedEntry* **unflushedEntry** = **null**;  
 }  
}

###### flush0()

**protected final void** flush0() {  
 *// Flush immediately only when there's no pending flush.  
 // If there's a pending flush operation, event loop will call forceFlush() later,  
 // and thus there's no need to call it now.* **if** (!**isFlushPending()**) {  
 **super**.**flush0()**;  
 }  
}

**private boolean** isFlushPending() {  
 SelectionKey selectionKey = selectionKey();

*//判断是否有给SocketChannel通道注册OP\_WRITE事件* **return** selectionKey.isValid() && (selectionKey.interestOps() & SelectionKey.***OP\_WRITE***) != 0;  
}

**protected void** flush0() {  
 **if** (**inFlush0**) {  
 *// Avoid re-entrance* **return**;  
 }  
  
 **final** ChannelOutboundBuffer outboundBuffer = **this**.**outboundBuffer**;  
 **if** (outboundBuffer == **null** || outboundBuffer.isEmpty()) {  
 **return**;  
 }  
  
 **inFlush0** = **true**;  
  
 *// Mark all pending write requests as failure if the channel is inactive.* **if** (!isActive()) {  
 **try** {  
 **if** (isOpen()) {  
 outboundBuffer.failFlushed(***FLUSH0\_NOT\_YET\_CONNECTED\_EXCEPTION***, **true**);  
 } **else** {  
 *// Do not trigger channelWritabilityChanged because the channel is closed already.* outboundBuffer.failFlushed(***FLUSH0\_CLOSED\_CHANNEL\_EXCEPTION***, **false**);  
 }  
 } **finally** {  
 **inFlush0** = **false**;  
 }  
 **return**;  
 }  
  
 **try** {  
 **doWrite(outboundBuffer);** } **catch** (Throwable t) {  
 **if** (t **instanceof** IOException && config().isAutoClose()) {  
 */\*\*  
 \* Just call {****@link*** *#close(ChannelPromise, Throwable, boolean)} here which will take care of  
 \* failing all flushed messages and also ensure the actual close of the underlying transport  
 \* will happen before the promises are notified.  
 \*  
 \* This is needed as otherwise {****@link*** *#isActive()} , {****@link*** *#isOpen()} and {****@link*** *#isWritable()}  
 \* may still return {****@code*** *true} even if the channel should be closed as result of the exception.  
 \*/* close(voidPromise(), t, ***FLUSH0\_CLOSED\_CHANNEL\_EXCEPTION***, **false**);  
 } **else** {  
 **try** {  
 shutdownOutput(voidPromise(), t);  
 } **catch** (Throwable t2) {  
 close(voidPromise(), t2, ***FLUSH0\_CLOSED\_CHANNEL\_EXCEPTION***, **false**);  
 }  
 }  
 } **finally** {  
 **inFlush0** = **false**;  
 }  
}

doWrite(outboundBuffer)

**protected void** doWrite(ChannelOutboundBuffer in) **throws** Exception {

//返回SocketChannel  
 SocketChannel ch = javaChannel();  
 **int** writeSpinCount = config().getWriteSpinCount();  
 **do** {  
 **if** (in.isEmpty()) {  
 *// All written so clear OP\_WRITE* clearOpWrite();  
 *// Directly return here so incompleteWrite(...) is not called.* **return**;  
 }  
  
 *// Ensure the pending writes are made of ByteBufs only.* **int** maxBytesPerGatheringWrite = ((NioSocketChannelConfig) **config**).getMaxBytesPerGatheringWrite();  
 ByteBuffer[] nioBuffers = in.nioBuffers(1024, maxBytesPerGatheringWrite);  
 **int** nioBufferCnt = in.nioBufferCount();  
  
 *// Always us nioBuffers() to workaround data-corruption.  
 // See https://github.com/netty/netty/issues/2761* **switch** (nioBufferCnt) {  
 **case** 0:  
 *// We have something else beside ByteBuffers to write so fallback to normal writes.* writeSpinCount -= doWrite0(in);  
 **break**;  
 **case** 1: {  
 *// Only one ByteBuf so use non-gathering write  
 // Zero length buffers are not added to nioBuffers by ChannelOutboundBuffer, so there is no need  
 // to check if the total size of all the buffers is non-zero.* ByteBuffer buffer = nioBuffers[0];  
 **int** attemptedBytes = buffer.remaining();

**//将字节序列从给定的缓冲区写入此通道，返回写入字节数** **final int** localWrittenBytes = ch.write(buffer);  
 **if** (localWrittenBytes <= 0) {  
 incompleteWrite(**true**);  
 **return**;  
 }  
 **adjustMaxBytesPerGatheringWrite**(attemptedBytes, localWrittenBytes, maxBytesPerGatheringWrite);  
 in.removeBytes(localWrittenBytes);  
 --writeSpinCount;  
 **break**;  
 }  
 **default**: {  
 *// Zero length buffers are not added to nioBuffers by ChannelOutboundBuffer, so there is no need  
 // to check if the total size of all the buffers is non-zero.  
 // We limit the max amount to int above so cast is safe* **long** attemptedBytes = in.nioBufferSize();  
 **final long** localWrittenBytes = ch.write(nioBuffers, 0, nioBufferCnt);  
 **if** (localWrittenBytes <= 0) {  
 incompleteWrite(**true**);  
 **return**;  
 }  
 *// Casting to int is safe because we limit the total amount of data in the nioBuffers to int above.* adjustMaxBytesPerGatheringWrite((**int**) attemptedBytes, (**int**) localWrittenBytes,  
 maxBytesPerGatheringWrite);  
 in.removeBytes(localWrittenBytes);  
 --writeSpinCount;  
 **break**;  
 }  
 }  
 } **while** (writeSpinCount > 0);  
  
 incompleteWrite(writeSpinCount < 0);  
}

adjustMaxBytesPerGatheringWrite

**private void** adjustMaxBytesPerGatheringWrite(**int** attempted, **int** written, **int** oldMaxBytesPerGatheringWrite) {  
 *// By default we track the SO\_SNDBUF when ever it is explicitly set. However some OSes may dynamically change  
 // SO\_SNDBUF (and other characteristics that determine how much data can be written at once) so we should try  
 // make a best effort to adjust as OS behavior changes.* **if** (attempted == written) {  
 **if** (attempted << 1 > oldMaxBytesPerGatheringWrite) {  
 ((NioSocketChannelConfig) **config**).setMaxBytesPerGatheringWrite(attempted << 1);  
 }  
 } **else if** (attempted > ***MAX\_BYTES\_PER\_GATHERING\_WRITE\_ATTEMPTED\_LOW\_THRESHOLD*** && written < attempted >>> 1) {  
 ((NioSocketChannelConfig) **config**).setMaxBytesPerGatheringWrite(attempted >>> 1);  
 }  
}

clearOpWrite()从通道删除OP\_WRITE

**protected final void** clearOpWrite() {  
 **final** SelectionKey key = selectionKey();  
 *// Check first if the key is still valid as it may be canceled as part of the deregistration  
 // from the EventLoop  
 // See https://github.com/netty/netty/issues/2104* **if** (!key.isValid()) {  
 **return**;  
 }  
 **final int** interestOps = key.interestOps();  
 **if** ((interestOps & SelectionKey.***OP\_WRITE***) != 0) {  
 key.interestOps(interestOps & ~SelectionKey.***OP\_WRITE***);  
 }  
}

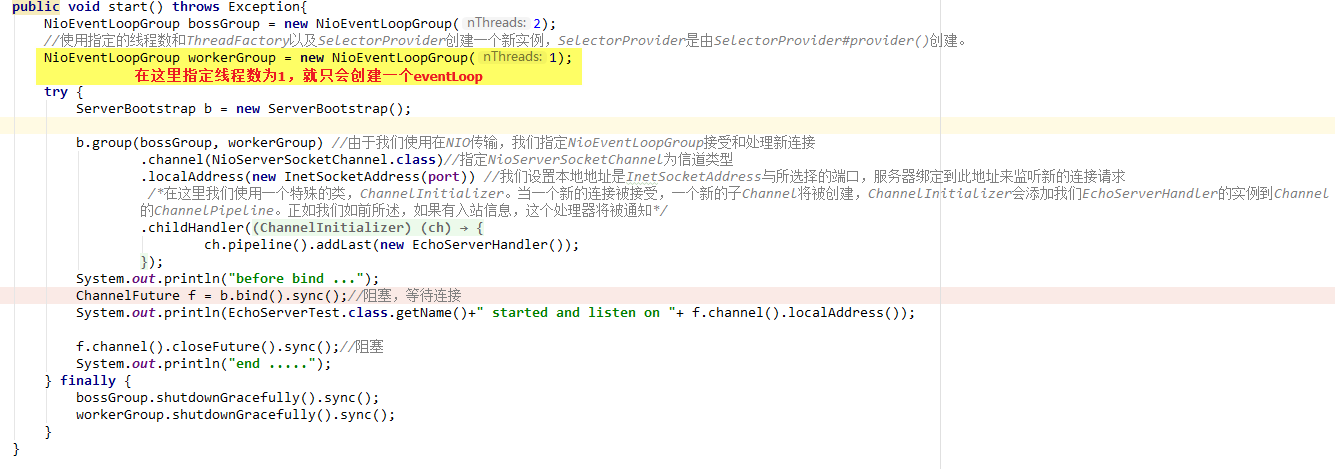
###### ChannelOutboundBuffer类

**public final class** ChannelOutboundBuffer {  
 *// Assuming a 64-bit JVM:  
 // - 16 bytes object header  
 // - 8 reference fields  
 // - 2 long fields  
 // - 2 int fields  
 // - 1 boolean field  
 // - padding* **static final int *CHANNEL\_OUTBOUND\_BUFFER\_ENTRY\_OVERHEAD*** =  
 SystemPropertyUtil.*getInt*(**"io.netty.transport.outboundBufferEntrySizeOverhead"**, 96);  
  
 **private static final** InternalLogger ***logger*** = InternalLoggerFactory.*getInstance*(ChannelOutboundBuffer.**class**);  
  
 **private static final** FastThreadLocal<ByteBuffer[]> ***NIO\_BUFFERS*** = **new** FastThreadLocal<ByteBuffer[]>() {  
 @Override  
 **protected** ByteBuffer[] initialValue() **throws** Exception {  
 **return new** ByteBuffer[1024];  
 }  
 };  
  
 **private final** Channel **channel**;  
  
 *// Entry(flushedEntry) --> ... Entry(unflushedEntry) --> ... Entry(tailEntry)  
 //  
 // The Entry that is the first in the linked-list structure that was flushed* **private** Entry **flushedEntry**;  
 *// The Entry which is the first unflushed in the linked-list structure* **private** Entry **unflushedEntry**;  
 *// The Entry which represents the tail of the buffer* **private** Entry **tailEntry**;  
 *// The number of flushed entries that are not written yet* **private int flushed**;  
  
 **private int nioBufferCount**;  
 **private long nioBufferSize**;  
  
 **private boolean inFail**;  
  
 **private static final** AtomicLongFieldUpdater<ChannelOutboundBuffer> ***TOTAL\_PENDING\_SIZE\_UPDATER*** =  
 AtomicLongFieldUpdater.*newUpdater*(ChannelOutboundBuffer.**class**, **"totalPendingSize"**);  
  
 @SuppressWarnings(**"UnusedDeclaration"**)  
 **private volatile long totalPendingSize**;  
  
 **private static final** AtomicIntegerFieldUpdater<ChannelOutboundBuffer> ***UNWRITABLE\_UPDATER*** =  
 AtomicIntegerFieldUpdater.*newUpdater*(ChannelOutboundBuffer.**class**, **"unwritable"**);  
  
 @SuppressWarnings(**"UnusedDeclaration"**)  
 **private volatile int unwritable**;  
  
 **private volatile** Runnable **fireChannelWritabilityChangedTask**;  
  
 ChannelOutboundBuffer(AbstractChannel channel) {  
 **this**.**channel** = channel;  
 }  
  
 */\*\*  
 \* Add given message to this {****@link*** *ChannelOutboundBuffer}. The given {****@link*** *ChannelPromise} will be notified once  
 \* the message was written.  
 \*/* **public void** addMessage(Object msg, **int** size, ChannelPromise promise) {  
 Entry entry = Entry.*newInstance*(msg, size, *total*(msg), promise);  
 **if** (**tailEntry** == **null**) {  
 **flushedEntry** = **null**;  
 } **else** {  
 Entry tail = **tailEntry**;  
 tail.**next** = entry;  
 }  
 **tailEntry** = entry;  
 **if** (**unflushedEntry** == **null**) {  
 **unflushedEntry** = entry;  
 }  
  
 *// increment pending bytes after adding message to the unflushed arrays.  
 // See https://github.com/netty/netty/issues/1619* incrementPendingOutboundBytes(entry.**pendingSize**, **false**);  
 }

*/\*\*  
 \* Add a flush to this {****@link*** *ChannelOutboundBuffer}. This means all previous added messages are marked as flushed  
 \* and so you will be able to handle them.  
 \*/***public void** addFlush() {  
 *// There is no need to process all entries if there was already a flush before and no new messages  
 // where added in the meantime.  
 //  
 // See https://github.com/netty/netty/issues/2577* Entry entry = **unflushedEntry**;  
 **if** (entry != **null**) {  
 **if** (**flushedEntry** == **null**) {  
 *// there is no flushedEntry yet, so start with the entry* **flushedEntry** = entry;  
 }  
 **do** {  
 **flushed** ++;  
 **if** (!entry.**promise**.setUncancellable()) {  
 *// Was cancelled so make sure we free up memory and notify about the freed bytes* **int** pending = entry.cancel();  
 decrementPendingOutboundBytes(pending, **false**, **true**);  
 }  
 entry = entry.**next**;  
 } **while** (entry != **null**);  
  
 *// All flushed so reset unflushedEntry* **unflushedEntry** = **null**;  
 }  
}

# NioEventLoop中wakenUp详解

# 同一个NioEventLoop能不能接受多个客户端连接

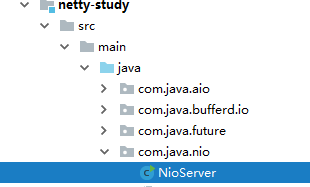
可以，一个NioEventLoop对应一个NioSocketChannel,一个NioSocketChannel对应一个SocketChannel,而所有的SocketChannel都注册到了Selector，注意整个服务端应用只有一个selector，这个selector监听所有通道事件。

每个NioEventLoop都会分配一个线程，该线程负责选择已准备好的通道，然后使用processSelectedKeys()处理该通道事件

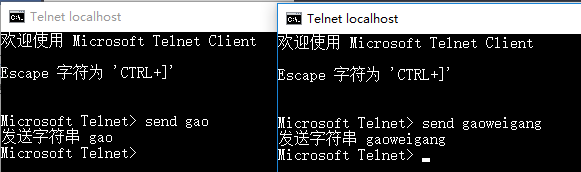
**只是说配置多线程性能会更好而已**

## 观察Java NIO

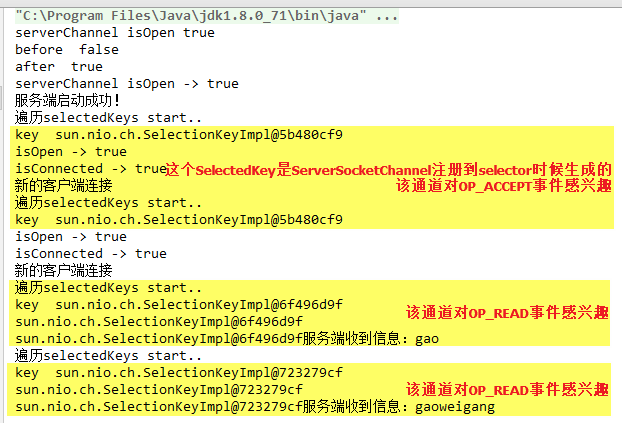
1. 启动服务器动程序



1. 使用cmd命令打开两个终端，然后分别使用telnet localhost 8000命令接连到服务器
2. 给服务器端发送信息

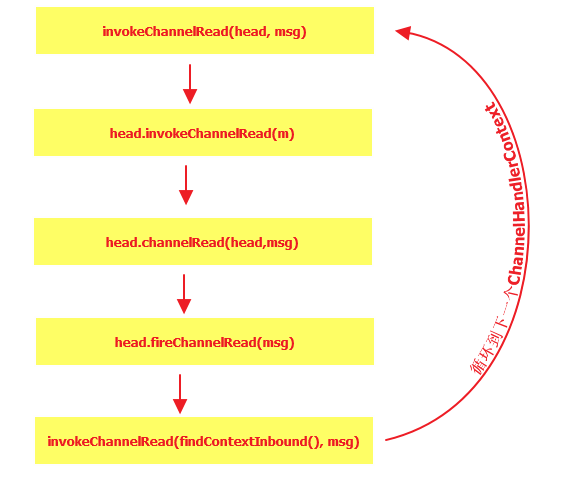


4.看看服务器端的输出



# ChannelHandlerContext详解

**关键点：通过findContextInbound或findContextOutbound循环到下一个ChannelHandlerContext**



# 解决JDK的NIO bug

关于该bug的描述见 [http://bugs.java.com/bugdatabase/view\_bug.do?bug\_id=6595055)](http://bugs.java.com/bugdatabase/view_bug.do?bug_id=6595055" \t "https://www.cnblogs.com/zhangboyu/p/_blank)

# ResourceLeakDetector

资源泄漏探测器实现原理

# Netty中reactor模型

# ClassLoader详解

# AccessController详解

AccessController.*doPrivileged*

# System.nanoTime()详解

参考：https://somefuture.iteye.com/blog/2404685

java有两个获取和时间相关的秒数方法，一个是广泛使用的

System.*currentTimeMillis*();

返回的是从一个长整型结果，表示毫秒。

另一个是

System.*nanoTime*();

返回的是纳秒。

“纳”这个单位 一般不是第一次见。前几年相当火爆的“纳米”和他是同一级别。纳表示的是10的-9次方。在真空中，光一纳秒也只能传播30厘米。

比纳秒大一级别的是微秒，10的-6次方；然后是就是毫秒，10的-3次方。

纳秒下面还有皮秒、飞秒等。

既然纳秒比毫秒高10的6次方精度，那么他们的比值就应该是10的6次方。然而并非如此。

看下面的代码

**long** l = System.*currentTimeMillis*();  
Date date = **new** Date(l);  
System.***out***.println(l);  
System.***out***.println(date);

System.***out***.println(**new** Date());

最后输出的当前时间。



大家可能都知道毫秒方法返回的是自1970年到现在的毫秒数。而Java的日期也是如此，所以他俩是等值的。

但是使用纳秒方法的输出可能让我们摸不着头脑：

**long** l = System.*nanoTime*();  
Date date = **new** Date(l / 1000000);  
System.***out***.println(l);  
System.***out***.println(date);  
System.***out***.println(**new** Date());

输出如下



这个输出在不同的机器上可能不一样，我的输出是Wed Jan 07 22:09:12 CST 1970

为什么会这样？

根据纳秒方法的注释：

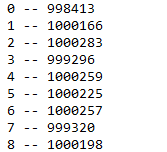
Returns the current value of the running Java Virtual Machine's high-resolution time source, in nanoseconds.  
This method can only be used to measure elapsed time and is not related to any other notion of system or wall-clock time. The value returned represents nanoseconds since some fixed but arbitrary origin time (perhaps in the future, so values may be negative). The same origin is used by all invocations of this method in an instance of a Java virtual machine; other virtual machine instances are likely to use a different origin.

翻译一下就是：返回当前JVM的高精度时间。该方法只能用来测量时段而和系统时间无关。它的返回值是从某个固定但随意的时间点开始的（可能是未来的某个时间）。不同的JVM使用的起点可能不同。

下面写一个程序来反映他和毫秒方法的关系。

**for**(**int** i = 0; i < 9; i++){  
 **long** m = System.*currentTimeMillis*();  
 **long** n = System.*nanoTime*();  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException ignored) {  
 }  
 **long** m1 = System.*currentTimeMillis*();  
 **long** n1 = System.*nanoTime*();  
 **long** m0 = m1 - m;  
 **long** n0 = n1 - n;  
  
 System.***out***.println(i + **" -- "** + (n0 / m0));  
};

输出如下：



不同的测试可能结果不同，不过可以看到，这个比值大约是10的6次方。

# Bootstrap

# OP\_WRITE事件

关于OP\_WRITE事件：

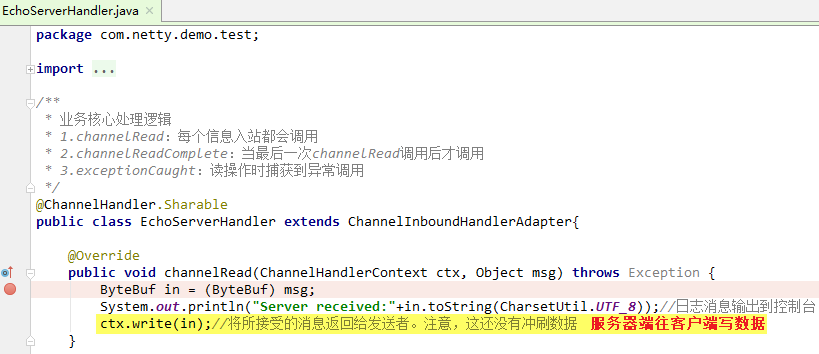
OP\_WRITE事件的就绪条件并不是发生在调用channel的write方法之后，而是在当底层缓冲区有空闲空间的情况下。因为写缓冲区在绝大部分时候都是有空闲空间的，所以如果你注册了写事件，这会使得写事件一直处于就绪，选择处理现场就会一直占用着CPU资源。所以，只有当你确实有数据写时再注册写操作，并在写完以后马上取消注册。

SocketChannel在写数据时，若发现当前buffer还有数据，但写缓冲区已经满的情况下，

socketChannel.write(buffer)会返回已经写出去的字节数，此时为0。那么这个时候我们就需要注册OP\_WRITE事件，这样当写缓冲区又有空闲空间的时候就会触发OP\_WRITE事件，这样我们就可以继续将没写完的数据继续写出了。而且在写完后，一定要记得将OP\_WRITE事件注销。

# OP\_WRITE注册

来看看NioServerChannel的doWrite()操作([ch.unsafe().forceFlush()])方法最终也就是会调用到这里



## ctx.write(in)

**//这里msg是ByteBuf**

**public** ChannelFuture write(Object msg) {  
 **return write(msg, newPromise());**  
}

**public** ChannelFuture write(**final** Object msg, **final** ChannelPromise promise) {  
 **if** (msg == **null**) {  
 **throw new** NullPointerException(**"msg"**);  
 }  
  
 **try** {  
 **if** (isNotValidPromise(promise, **true**)) {  
 ReferenceCountUtil.*release*(msg);  
 *// cancelled* **return** promise;  
 }  
 } **catch** (RuntimeException e) {  
 ReferenceCountUtil.*release*(msg);  
 **throw** e;  
 }  
 **write**(msg, **false**, promise);  
  
 **return** promise;  
}

**private void** write(Object msg, **boolean** flush, ChannelPromise promise) {

**//向前遍历pipeline找到第一个ChannelHandlerContextOutbound，**

**//即头节点headContext**  
 AbstractChannelHandlerContext next = findContextOutbound();  
 **final** Object m = **pipeline**.touch(msg, next);  
 EventExecutor executor = next.executor();  
 **if** (executor.inEventLoop()) {  
 **if** (flush) {

**//next即HeadContext** **next.invokeWriteAndFlush(m, promise);** } **else** {  
 next.invokeWrite(m, promise);  
 }  
 } **else** {  
 AbstractWriteTask task;  
 **if** (flush) {  
 task = WriteAndFlushTask.*newInstance*(next, m, promise);  
 } **else** {  
 task = WriteTask.*newInstance*(next, m, promise);  
 }  
 *safeExecute*(executor, task, promise, m);  
 }  
}

### next.invokeWriteAndFlush(m, promise)

**private void** invokeWriteAndFlush(Object msg, ChannelPromise promise) {  
 **if** (invokeHandler()) {  
 invokeWrite0(msg, promise);  
 invokeFlush0();  
 } **else** {  
 writeAndFlush(msg, promise);  
 }  
}

#### invokeWrite0(msg, promise)

**private void** invokeWrite0(Object msg, ChannelPromise promise) {  
 **try** {

**//handler()和this即HeadContext** ((ChannelOutboundHandler) handler()).write(**this**, msg, promise);  
 } **catch** (Throwable t) {  
 *notifyOutboundHandlerException*(t, promise);  
 }  
}

**public void** write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) **throws** Exception {  
 **unsafe**.write(msg, promise);  
}

**public final void** write(Object msg, ChannelPromise promise) {  
 assertEventLoop();  
  **//1.ChannelOutboundBuffer只是传输的实现者，它被用于AbstractChannel内部**

**//数据结构，用于存储其未决的出站写请求数据。**

**//2.所有的方法必须由IO线程的传输实现调用，以下方法除：size(),isWritable()**

**//,getUserDefinedWritability(),setUserDefinedWritability()**

ChannelOutboundBuffer outboundBuffer = **this**.**outboundBuffer**;  
 **if** (outboundBuffer == **null**) {  
 **//如果outboundBuffer 为null,则说明通道已经关闭，因此需要立即失败。**

**//如果它不是null,其余的处理将在flush0()中完成。** *// See https://github.com/netty/netty/issues/2362* safeSetFailure(promise, ***WRITE\_CLOSED\_CHANNEL\_EXCEPTION***);  
 *// release message now to prevent resource-leak* ReferenceCountUtil.*release*(msg);  
 **return**;  
 }  
  
 **int** size;  
 **try** {

**//在将新消息添加到此AbstractChannel的ChannelOutboundBuffer时调用,**

**//以便Channel实现将消息转换为另一个消息。(例如：堆缓冲区->直接缓冲区)**  
 msg = filterOutboundMessage(msg);**//将消息从堆缓冲区转移到直接缓冲区**

**//计算出消息的大小，根据msg的类型不同计算方式不同** size = **pipeline**.estimatorHandle().size(msg);  
 **if** (size < 0) {  
 size = 0;  
 }  
 } **catch** (Throwable t) {  
 safeSetFailure(promise, t);  
 ReferenceCountUtil.*release*(msg);  
 **return**;  
 }  
   
 outboundBuffer.**addMessage**(msg, size, promise);  
}

##### filterOutboundMessage

**//将消息从堆缓冲区转移到直接缓冲区**

**protected final** Object filterOutboundMessage(Object msg) {  
 **if** (msg **instanceof** ByteBuf) {  
 ByteBuf buf = (ByteBuf) msg;  
 **if** (buf.isDirect()) {  
 **return** msg;  
 }  
  
 **return** newDirectBuffer(buf);  
 }  
  
 **if** (msg **instanceof** FileRegion) {  
 **return** msg;  
 }  
  
 **throw new** UnsupportedOperationException(  
 **"unsupported message type: "** + StringUtil.*simpleClassName*(msg) + ***EXPECTED\_TYPES***);  
}

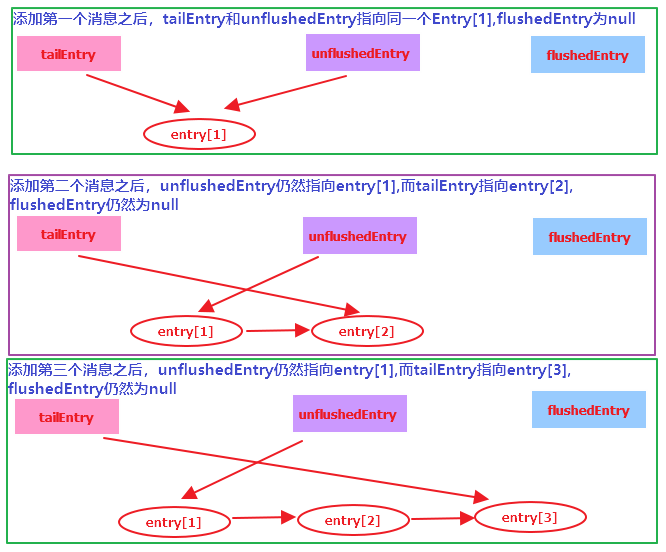
##### addMessage()

**public void** addMessage(Object msg, **int** size, ChannelPromise promise) {

**//total()方法计算出消息msg所占字节数** Entry entry = Entry.*newInstance*(msg, size, *total*(msg), promise);  
 **if** (**tailEntry** == **null**) {  
 **flushedEntry** = **null**;  
 } **else** {  
 Entry tail = **tailEntry**;  
 tail.**next** = entry;  
 }  
 **tailEntry** = entry;  
 **if** (**unflushedEntry** == **null**) {  
 **unflushedEntry** = entry;  
 }  
  
 *// increment pending bytes after adding message to the unflushed arrays.  
 // See https://github.com/netty/netty/issues/1619* **incrementPendingOutboundBytes**(entry.**pendingSize**, **false**);  
}

**----------------------------------------------------------**

说明：调用addMessage时tailEntry,unflushedEntry以及flushedEntry的状态变化



**-----------------------------------------------------------------------------------------------------**

**private void** incrementPendingOutboundBytes(**long** size, **boolean** invokeLater) {  
 **if** (size == 0) {  
 **return**;  
 }  
  
 **long** newWriteBufferSize = ***TOTAL\_PENDING\_SIZE\_UPDATER***.addAndGet(**this**, size);  
 **if** (newWriteBufferSize > **channel**.config().getWriteBufferHighWaterMark()) {  
 setUnwritable(invokeLater);  
 }  
}

**private void** setUnwritable(**boolean** invokeLater) {  
 **for** (;;) {  
 **final int** oldValue = **unwritable**;  
 **final int** newValue = oldValue | 1;  
 **if** (***UNWRITABLE\_UPDATER***.compareAndSet(**this**, oldValue, newValue)) {  
 **if** (oldValue == 0 && newValue != 0) {  
 **fireChannelWritabilityChanged(invokeLater)**;  
 }  
 **break**;  
 }  
 }  
}

**private void** fireChannelWritabilityChanged(**boolean** invokeLater) {  
 **final** ChannelPipeline pipeline = **channel**.pipeline();  
 **if** (invokeLater) {  
 Runnable task = **fireChannelWritabilityChangedTask**;  
 **if** (task == **null**) {  
 **fireChannelWritabilityChangedTask** = task = **new** Runnable() {  
 @Override  
 **public void** run() {  
 pipeline.fireChannelWritabilityChanged();  
 }  
 };  
 }  
 **channel**.eventLoop().execute(task);  
 } **else** {  
 pipeline.fireChannelWritabilityChanged();  
 }  
}

##### io.netty.channel.ChannelOutboundBuffer

//1.(仅仅是传输实现者) 由AbstractChannel使用的内部数据结构，用于存储未决的出站写

//请求。

//2.所有的方法必须由IO线程的传输实现调用，以下方法除外:size(),isWritable(),

//getUserDefinedWritability(), setUserDefinedWritability()

**public final class** ChannelOutboundBuffer {  
 *// Assuming a 64-bit JVM:  
 // - 16 bytes object header  
 // - 8 reference fields  
 // - 2 long fields  
 // - 2 int fields  
 // - 1 boolean field  
 // - padding* **static final int *CHANNEL\_OUTBOUND\_BUFFER\_ENTRY\_OVERHEAD*** =  
 SystemPropertyUtil.*getInt*(**"io.netty.transport.outboundBufferEntrySizeOverhead"**, 96);  
  
 **private static final** InternalLogger ***logger*** = InternalLoggerFactory.*getInstance*(ChannelOutboundBuffer.**class**);  
  
 **private static final** FastThreadLocal<ByteBuffer[]> ***NIO\_BUFFERS*** = **new** FastThreadLocal<ByteBuffer[]>() {  
 @Override  
 **protected** ByteBuffer[] initialValue() **throws** Exception {  
 **return new** ByteBuffer[1024];  
 }  
 };  
  
 **private final** Channel **channel**;  
  
 *// 1.Entry(flushedEntry) --> ... Entry(unflushedEntry) --> ... Entry(tailEntry)  
 // 2.unflushedEntry执行entry.promise.setUncancellable()变成flushedEntry  
 // 3.The Entry that is the first in the linked-list structure that was flushed*

**private** Entry **flushedEntry**;***//刷新的链表结构中第一个条目*** *// The Entry which is the first unflushed in the linked-list structure* **private** Entry **unflushedEntry**;**//链表结构中第一个未刷新的条目**  
 *// The Entry which represents the tail of the buffer* **private** Entry **tailEntry**;**//表示缓冲区尾部的条目**  
 *// The number of flushed entries that are not written yet* **private int flushed**;**//尚未写入的刷新条目数**  
 **private int nioBufferCount**;  
 **private long nioBufferSize**;  
  
 **private boolean inFail**;  
  
 **private static final** AtomicLongFieldUpdater<ChannelOutboundBuffer> ***TOTAL\_PENDING\_SIZE\_UPDATER*** =  
 AtomicLongFieldUpdater.*newUpdater*(ChannelOutboundBuffer.**class**, **"totalPendingSize"**);  
  
 @SuppressWarnings(**"UnusedDeclaration"**)  
 **private volatile long totalPendingSize**;  
  
 **private static final** AtomicIntegerFieldUpdater<ChannelOutboundBuffer> ***UNWRITABLE\_UPDATER*** =  
 AtomicIntegerFieldUpdater.*newUpdater*(ChannelOutboundBuffer.**class**, **"unwritable"**);

...

***//将给定的消息添加到此ChannelOutboundBuffer。***

***//一旦消息被写入，将通知一定的ChannelPromise*****public void** addMessage(Object msg, **int** size, ChannelPromise promise) {  
 Entry entry = Entry.*newInstance*(msg, size, *total*(msg), promise);  
 **if** (**tailEntry** == **null**) {  
 **flushedEntry** = **null**;  
 } **else** {  
 Entry tail = **tailEntry**;  
 tail.**next** = entry;  
 }  
 **tailEntry** = entry;  
 **if** (**unflushedEntry** == **null**) {  
 **unflushedEntry** = entry;  
 }  
  
 *// increment pending bytes after adding message to the unflushed arrays.  
 // See https://github.com/netty/netty/issues/1619* incrementPendingOutboundBytes(entry.**pendingSize**, **false**);  
}

**//为这个ChannelOutboundBuffer添加刷新。这意味着以前添加的所有消息都被标记**

**//刷新，这样你就可以处理它们了****public void** addFlush() {  
 **//如果之前已经有刷新，并且在此期间没有添加新消息，则不需要处理所有条目** *// See https://github.com/netty/netty/issues/2577* Entry entry = **unflushedEntry**;  
 **if** (entry != **null**) {  
 **if** (**flushedEntry** == **null**) {

***//如果还没有flushedEntry,则从entry开始*****flushedEntry** = entry;  
 }  
 **do** {  
 **flushed** ++;  
 **if** (!entry.**promise**.setUncancellable()) {  
 *// Was cancelled so make sure we free up memory and notify about the freed bytes*

***//已取消，请确保释放内存并通知释放字节*****int** pending = entry.cancel();

**//更新未决的出站字节数** decrementPendingOutboundBytes(pending, **false**, **true**);  
 }  
 entry = entry.**next**;  
 } **while** (entry != **null**);  
  
 *// All flushed so reset unflushedEntry* **unflushedEntry** = **null**;  
 }  
}

**private void** decrementPendingOutboundBytes(**long** size, **boolean** invokeLater, **boolean** notifyWritability) {  
 **if** (size == 0) {  
 **return**;  
 }  
  
 **long** newWriteBufferSize = ***TOTAL\_PENDING\_SIZE\_UPDATER***.addAndGet(**this**, -size);  
 **if** (notifyWritability && newWriteBufferSize < **channel**.config().getWriteBufferLowWaterMark()) {  
 setWritable(invokeLater);  
 }  
}

}

##### io.netty.channel.ChannelOutboundBuffer.Entry

**static final class** Entry {  
 **private static final** Recycler<Entry> ***RECYCLER*** = **new** Recycler<Entry>() {  
 @Override  
 **protected** Entry newObject(Handle<Entry> handle) {  
 **return new** Entry(handle);  
 }  
 };  
  
 **private final** Handle<Entry> **handle**;  
 Entry **next**;  
 Object **msg**;  
 ByteBuffer[] **bufs**;  
 ByteBuffer **buf**;  
 ChannelPromise **promise**;  
 **long progress**;  
 **long total**;  
 **int pendingSize**;  
 **int count** = -1;  
 **boolean cancelled**;  
  
 **private** Entry(Handle<Entry> handle) {  
 **this**.**handle** = handle;  
 }  
  
 **static** Entry newInstance(Object msg, **int** size, **long** total, ChannelPromise promise) {  
 Entry entry = ***RECYCLER***.get();  
 entry.**msg** = msg;  
 entry.**pendingSize** = size + ***CHANNEL\_OUTBOUND\_BUFFER\_ENTRY\_OVERHEAD***;  
 entry.**total** = total;  
 entry.**promise** = promise;  
 **return** entry;  
 }  
  
 **int** cancel() {  
 **if** (!**cancelled**) {  
 **cancelled** = **true**;  
 **int** pSize = **pendingSize**;  
  
 *// release message and replace with an empty buffer* ReferenceCountUtil.*safeRelease*(**msg**);  
 **msg** = Unpooled.***EMPTY\_BUFFER***;  
  
 **pendingSize** = 0;  
 **total** = 0;  
 **progress** = 0;  
 **bufs** = **null**;  
 **buf** = **null**;  
 **return** pSize;  
 }  
 **return** 0;  
 }  
  
 **void** recycle() {  
 **next** = **null**;  
 **bufs** = **null**;  
 **buf** = **null**;  
 **msg** = **null**;  
 **promise** = **null**;  
 **progress** = 0;  
 **total** = 0;  
 **pendingSize** = 0;  
 **count** = -1;  
 **cancelled** = **false**;  
 **handle**.recycle(**this**);  
 }  
  
 Entry recycleAndGetNext() {  
 Entry next = **this**.**next**;  
 recycle();  
 **return** next;  
 }  
}

#### invokeFlush0()

**private void** invokeFlush0() {  
 **try** {

**//handler()和this即HeadContext** ((ChannelOutboundHandler) handler()).flush(**this**);  
 } **catch** (Throwable t) {  
 notifyHandlerException(t);  
 }  
}

**public void** flush(ChannelHandlerContext ctx) **throws** Exception {  
 **unsafe**.flush();  
}

**public final void** flush() {  
 assertEventLoop();  
  
 ChannelOutboundBuffer outboundBuffer = **this**.**outboundBuffer**;  
 **if** (outboundBuffer == **null**) {  
 **return**;  
 }  
 **//为这个ChannelOutboundBuffer添加刷新。这意味着以前添加的所有消息都被标记**

**//刷新，这样你就可以处理它们了** outboundBuffer.addFlush();

**//处理所有被标记刷新的消息** flush0();  
}

##### addFlush()

**//向这个ChannelOutboundBuffer添加刷新。这意味着以前添加的所有消息都被标记为**

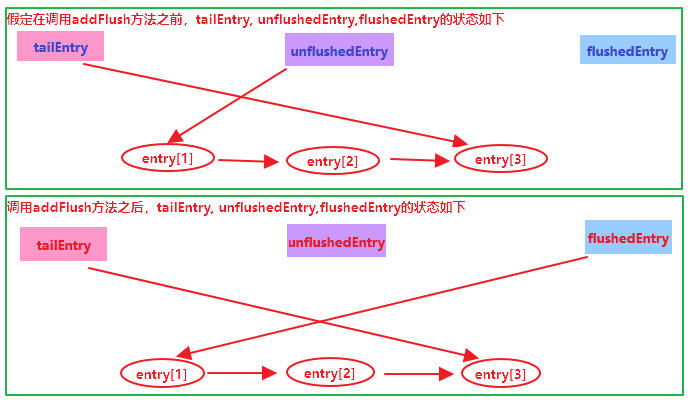
**//刷新,因此您将能够处理他们。****public void** addFlush() {  
  **//如果之前已经有刷新，与此同时没有新的消息添加，则不需要处理所有条目。** *// See https://github.com/netty/netty/issues/2577* Entry entry = **unflushedEntry**;  
 **if** (entry != **null**) {  
 **if** (**flushedEntry** == **null**) {  
 ***//没有了flushedEntry，从entry开始，也就是说待写入的数据***

***//flushedEntry都写入完了，再将unflushedEntry转换成flushedEntry*****flushedEntry** = entry;  
 }  
 **do** {  
 **flushed** ++;  
 **if** (!entry.**promise**.setUncancellable()) {  
 ***//已取消，请确保释放内存并通知释放的字节*****int** pending = entry.cancel();

**//减去未决的出站字节数** **decrementPendingOutboundBytes**(pending, **false**, **true**);  
 }  
 entry = entry.**next**;  
 } **while** (entry != **null**);  
  
 ***// 所有的entry都刷新了，所以重置unflushedEntry*****unflushedEntry** = **null**;  
 }  
}

**--------------------------------------------**

**说明：调用addFlush()方法，tailEntry,unflushedEntry,flushedEntry状态变化**

****

**--------------------------------------------------------**

**//减去未决的出站字节数**

**//size: 待减去的未决的出站字节数**

**//invokeLater:是否延迟调用**

**private void** decrementPendingOutboundBytes(**long** size, **boolean** invokeLater, **boolean** notifyWritability) {  
 **if** (size == 0) {  
 **return**;  
 }  
 **//原子操作，更新未决的出站字节数，返回新的待写入写缓冲区的字节数** **long** newWriteBufferSize = ***TOTAL\_PENDING\_SIZE\_UPDATER***.addAndGet(**this**, -size);

**//当 新未决的出站字节数newWriteBufferSize 小于 写缓冲区低水位标志**  
 **if** (notifyWritability && newWriteBufferSize < **channel**.config().getWriteBufferLowWaterMark()) {  
 **setWritable(invokeLater);** }  
}

**private void** setWritable(**boolean** invokeLater) {  
 **for** (;;) {  
 **final int** oldValue = **unwritable**;  
 **final int** newValue = oldValue & ~1;  
 **if** (***UNWRITABLE\_UPDATER***.compareAndSet(**this**, oldValue, newValue)) {  
 **if** (oldValue != 0 && newValue == 0) {  
 **fireChannelWritabilityChanged(invokeLater);**  
 }  
 **break**;  
 }  
 }  
}

**private void** fireChannelWritabilityChanged(**boolean** invokeLater) {  
 **final** ChannelPipeline pipeline = **channel**.pipeline();

**//是否延迟调用,如果为true,则创建一个任务丢到任务队列taskQueue** **if** (invokeLater) {  
 Runnable task = **fireChannelWritabilityChangedTask**;  
 **if** (task == **null**) {  
 **fireChannelWritabilityChangedTask** = task = **new** Runnable() {  
 @Override  
 **public void** run() {

**//一旦Channel的可写状态(即是否可写)更改后调用。你能使用**

**//Channel.isWritable()来检查Channel的可写状态。** pipeline.**fireChannelWritabilityChanged()**;  
 }  
 };  
 }  
 **channel**.eventLoop().execute(task);  
 } **else** {

**//一旦Channel的可写状态(即是否可写)更改后调用。你能使用**

**//Channel.isWritable()来检查Channel的可写状态。**  
 pipeline.**fireChannelWritabilityChanged()**;  
 }  
}

**public final** ChannelPipeline fireChannelWritabilityChanged() {  
 AbstractChannelHandlerContext.*invokeChannelWritabilityChanged*(**head**);  
 **return this**;  
}

##### flush0()

**protected void** flush0() {

**//*避免重复进入***  
 **if** (**inFlush0**) {**return**;  
 }  
  
 **final** ChannelOutboundBuffer outboundBuffer = **this**.**outboundBuffer**;  
 **if** (outboundBuffer == **null** || outboundBuffer.isEmpty()) {  
 **return**;  
 }  
  
 **inFlush0** = **true**;  
  
 ***//如果通道处于非活动状态，则将所有挂起的写请求标记为失败*****if** (!isActive()) {  
 **try** {  
 **if** (isOpen()) {**//通道是否仍然处于打开状态** outboundBuffer.failFlushed(***FLUSH0\_NOT\_YET\_CONNECTED\_EXCEPTION***, **true**);  
 } **else** {  
 *// Do not trigger channelWritabilityChanged because the channel is closed already.* outboundBuffer.failFlushed(***FLUSH0\_CLOSED\_CHANNEL\_EXCEPTION***, **false**);  
 }  
 } **finally** {  
 **inFlush0** = **false**;  
 }  
 **return**;  
 }  
  
 **try** {  
 **doWrite(outboundBuffer);** } **catch** (Throwable t) {  
 **if** (t **instanceof** IOException && config().isAutoClose()) {  
 */\*\*  
 \* Just call {****@link*** *#close(ChannelPromise, Throwable, boolean)} here which will take care of  
 \* failing all flushed messages and also ensure the actual close of the underlying transport  
 \* will happen before the promises are notified.  
 \*  
 \* This is needed as otherwise {****@link*** *#isActive()} , {****@link*** *#isOpen()} and {****@link*** *#isWritable()}  
 \* may still return {****@code*** *true} even if the channel should be closed as result of the exception.  
 \*/* close(voidPromise(), t, ***FLUSH0\_CLOSED\_CHANNEL\_EXCEPTION***, **false**);  
 } **else** {  
 **try** {  
 shutdownOutput(voidPromise(), t);  
 } **catch** (Throwable t2) {  
 close(voidPromise(), t2, ***FLUSH0\_CLOSED\_CHANNEL\_EXCEPTION***, **false**);  
 }  
 }  
 } **finally** {  
 **inFlush0** = **false**;  
 }  
}

###### outboundBuffer.failFlushed()

**void** failFlushed(Throwable cause, **boolean** notify) { ***//确保此方法不会重入。添加到当前promise的监听器可以在下面循环的***

***//tryFailure()调用中的当前线程通知，并且监听器可以间接触发另一个fail()***

***//调用(通常通过关闭通道)*** *// See https://github.com/netty/netty/issues/1501* **if** (**inFail**) {  
 **return**;  
 }  
  
 **try** {  
 **inFail** = **true**;

***//将所有挂起的写请求标记为失败***  
 **for** (;;) {  
 **if** (!remove0(cause, notify)) {  
 **break**;  
 }  
 }  
 } **finally** {  
 **inFail** = **false**;  
 }  
}

remove0

**private boolean** remove0(Throwable cause, **boolean** notifyWritability) {  
 Entry e = **flushedEntry**;  
 **if** (e == **null**) {  
 clearNioBuffers();  
 **return false**;  
 }  
 Object msg = e.**msg**;  
  
 ChannelPromise promise = e.**promise**;  
 **int** size = e.**pendingSize**;  
  
 removeEntry(e);  
  
 **if** (!e.**cancelled**) {  
 *//仅仅释放消息，如果之前未取消，则失败并减去未决的出站字节数* ReferenceCountUtil.*safeRelease*(msg);  
 **//失败** *safeFail*(promise, cause);

**//减去未决的出站字节数**  
 decrementPendingOutboundBytes(size, **false**, notifyWritability);  
 }  
  
 ***// 回收 entry***e.**recycle()**;  
  
 **return true**;  
}

clearNioBuffers()

*//清除数组中所有的ByteBuffer，以便对其进行垃圾回收  
// See https://github.com/netty/netty/issues/3837***private void** clearNioBuffers() {  
 **int** count = **nioBufferCount**;  
 **if** (count > 0) {  
 **nioBufferCount** = 0;  
 Arrays.*fill*(***NIO\_BUFFERS***.get(), 0, count, **null**);  
 }  
}

removeEntry(e)

**private void** removeEntry(Entry e) {  
 **if** (-- **flushed** == 0) {  
 *// processed everything* **flushedEntry** = **null**;  
 **if** (e == **tailEntry**) {  
 **tailEntry** = **null**;  
 **unflushedEntry** = **null**;  
 }  
 } **else** {  
 **flushedEntry** = e.**next**;  
 }  
}

safeFail(promise, cause)

**private static void** safeFail(ChannelPromise promise, Throwable cause) {  
 *//仅当给定的Promise不是voidChannelPromise类型时才记录日志* PromiseNotificationUtil.*tryFailure*(promise, cause, promise **instanceof** VoidChannelPromise ? **null** : ***logger***);  
}

recycle()

**void** recycle() {  
 **next** = **null**;  
 **bufs** = **null**;  
 **buf** = **null**;  
 **msg** = **null**;  
 **promise** = **null**;  
 **progress** = 0;  
 **total** = 0;  
 **pendingSize** = 0;  
 **count** = -1;  
 **cancelled** = **false**;  
 **handle**.recycle(**this**);  
}

###### doWrite(outboundBuffer)

**protected void** doWrite(ChannelOutboundBuffer in) **throws** Exception {  
 SocketChannel ch = javaChannel();

//获取配置的写自旋次数，默认值为16  
 **int** writeSpinCount = config().getWriteSpinCount();  
 **do** {  
 **if** (in.isEmpty()) {  
 ***// 所有的Entry都已经写入，则清除注册的OP\_WRITE监听***clearOpWrite();  
 ***//此处直接返回，因此incompleteWrite(...)不会调用.*****return**;  
 }  
  
 ***//确保未决的写入仅由ByteBufs构成***

***//每次聚集写入的最大字节，默认为int的最大值*****int** maxBytesPerGatheringWrite = ((NioSocketChannelConfig) **config**).getMaxBytesPerGatheringWrite();

**//Java里面的ByteBuffer** ByteBuffer[] nioBuffers = in.nioBuffers(1024, maxBytesPerGatheringWrite);  
 **int** nioBufferCnt = in.nioBufferCount();  
  
 *// Always us nioBuffers() to workaround data-corruption.  
 // See https://github.com/netty/netty/issues/2761* **switch** (nioBufferCnt) {  
 **case** 0:  
 *// We have something else beside ByteBuffers to write so fallback to normal writes.* writeSpinCount -= doWrite0(in);  
 **break**;  
 **case** 1: {  
 *// Only one ByteBuf so use non-gathering write  
 // Zero length buffers are not added to nioBuffers by ChannelOutboundBuffer, so there is no need  
 // to check if the total size of all the buffers is non-zero.* ByteBuffer buffer = nioBuffers[0];  
 **int** attemptedBytes = buffer.remaining();  
 **final int** localWrittenBytes = ch.write(buffer);  
 **if** (localWrittenBytes <= 0) {  
 incompleteWrite(**true**);  
 **return**;  
 }  
 adjustMaxBytesPerGatheringWrite(attemptedBytes, localWrittenBytes, maxBytesPerGatheringWrite);  
 in.removeBytes(localWrittenBytes);  
 --writeSpinCount;  
 **break**;  
 }  
 **default**: {  
 *// Zero length buffers are not added to nioBuffers by ChannelOutboundBuffer, so there is no need  
 // to check if the total size of all the buffers is non-zero.  
 // We limit the max amount to int above so cast is safe* **long** attemptedBytes = in.nioBufferSize();  
 **final long** localWrittenBytes = ch.write(nioBuffers, 0, nioBufferCnt);  
 **if** (localWrittenBytes <= 0) {  
 incompleteWrite(**true**);  
 **return**;  
 }  
 *// Casting to int is safe because we limit the total amount of data in the nioBuffers to int above.* adjustMaxBytesPerGatheringWrite((**int**) attemptedBytes, (**int**) localWrittenBytes,  
 maxBytesPerGatheringWrite);  
 in.removeBytes(localWrittenBytes);  
 --writeSpinCount;  
 **break**;  
 }  
 }  
 } **while** (writeSpinCount > 0);  
  
 incompleteWrite(writeSpinCount < 0);  
}

clearOpWrite()

**protected final void** clearOpWrite() {  
 **final** SelectionKey key = selectionKey();  
 *// Check first if the key is still valid as it may be canceled as part of the deregistration  
 // from the EventLoop  
 // See https://github.com/netty/netty/issues/2104* **if** (!key.isValid()) {  
 **return**;  
 }  
 **final int** interestOps = key.interestOps();  
 **if** ((interestOps & SelectionKey.***OP\_WRITE***) != 0) {  
 key.interestOps(interestOps & ~SelectionKey.***OP\_WRITE***);  
 }  
}

in.nioBuffers(1024, maxBytesPerGatheringWrite)

**如果当前未决的消息仅由ByteBuf构成，则返回一个直接NIO缓冲区数组。**

**nioBufferCount()和nioBufferSize()将分别返回在NIO缓冲区数组中的NIO缓冲区数量和NIO缓冲区的可读字节总数。**

**注意：返回的数组是重用的，因此不应该转义**

**AbstractChannel.doWrite(ChannelOutboundBuffer)}。**

**例如，请参考NioSocketChannel.doWrite(ChannelOutboundBuffer)。**

**maxCount: 将添加到返回值的缓冲区的最大数量。**

**maxBytes：返回值中包含的最大字节数提示。请注意,可能会超过此值，因为我们尽最大努力在返回值中包含至少一个Bytebuffer,以确保写入进度。**

**public** ByteBuffer[] nioBuffers(**int** maxCount, **long** maxBytes) {  
 **assert** maxCount > 0;  
 **assert** maxBytes > 0;  
 **long** nioBufferSize = 0;  
 **int** nioBufferCount = 0;  
 **final** InternalThreadLocalMap threadLocalMap = InternalThreadLocalMap.*get*();

**//NIO\_BUFFERS是FastThreadLocal<ByteBuffer[]>**  
 ByteBuffer[] nioBuffers = ***NIO\_BUFFERS***.get(threadLocalMap);  
 Entry entry = **flushedEntry**;  
 **while** (isFlushedEntry(entry) && entry.**msg instanceof** ByteBuf) {  
 **if** (!entry.**cancelled**) {  
 ByteBuf buf = (ByteBuf) entry.**msg**;  
 **final int** readerIndex = buf.readerIndex();  
 **final int** readableBytes = buf.writerIndex() - readerIndex;  
  
 **if** (readableBytes > 0) {  
 **if** (maxBytes - readableBytes < nioBufferSize && nioBufferCount != 0) {  
 *// If the nioBufferSize + readableBytes will overflow maxBytes, and there is at least one entry  
 // we stop populate the ByteBuffer array. This is done for 2 reasons:  
 // 1. bsd/osx don't allow to write more bytes then Integer.MAX\_VALUE with one writev(...) call  
 // and so will return 'EINVAL', which will raise an IOException. On Linux it may work depending  
 // on the architecture and kernel but to be safe we also enforce the limit here.  
 // 2. There is no sense in putting more data in the array than is likely to be accepted by the  
 // OS.  
 //  
 // See also:  
 // - https://www.freebsd.org/cgi/man.cgi?query=write&sektion=2  
 // - http://linux.die.net/man/2/writev* **break**;  
 }  
 nioBufferSize += readableBytes;  
 **int** count = entry.**count**;  
 **if** (count == -1) {  
 *//noinspection ConstantValueVariableUse* entry.**count** = count = buf.nioBufferCount();  
 }  
 **int** neededSpace = *min*(maxCount, nioBufferCount + count);  
 **if** (neededSpace > nioBuffers.**length**) {  
 nioBuffers = *expandNioBufferArray*(nioBuffers, neededSpace, nioBufferCount);  
 ***NIO\_BUFFERS***.set(threadLocalMap, nioBuffers);  
 }  
 **if** (count == 1) {  
 ByteBuffer nioBuf = entry.**buf**;  
 **if** (nioBuf == **null**) {  
 *// cache ByteBuffer as it may need to create a new ByteBuffer instance if its a  
 // derived buffer* entry.**buf** = nioBuf = buf.internalNioBuffer(readerIndex, readableBytes);  
 }  
 nioBuffers[nioBufferCount++] = nioBuf;  
 } **else** {  
 ByteBuffer[] nioBufs = entry.**bufs**;  
 **if** (nioBufs == **null**) {  
 *// cached ByteBuffers as they may be expensive to create in terms  
 // of Object allocation* entry.**bufs** = nioBufs = buf.nioBuffers();  
 }  
 **for** (**int** i = 0; i < nioBufs.**length** && nioBufferCount < maxCount; ++i) {  
 ByteBuffer nioBuf = nioBufs[i];  
 **if** (nioBuf == **null**) {  
 **break**;  
 } **else if** (!nioBuf.hasRemaining()) {  
 **continue**;  
 }  
 nioBuffers[nioBufferCount++] = nioBuf;  
 }  
 }  
 **if** (nioBufferCount == maxCount) {  
 **break**;  
 }  
 }  
 }  
 entry = entry.**next**;  
 }  
 **this**.**nioBufferCount** = nioBufferCount;  
 **this**.**nioBufferSize** = nioBufferSize;  
  
 **return** nioBuffers;  
}

##### io.netty.util.internal.InternalThreadLocalMap

**为Netty和所有的FastThreadLocal存储本地线程局部变量的内部数据结构。**

**注意：这个类仅供内部使用，并且随时可能更改。除非你知道你在正在做什么，**

**否则使用FastThreadLocal**

**public final class** InternalThreadLocalMap **extends** UnpaddedInternalThreadLocalMap {  
  
 **private static final** InternalLogger ***logger*** = InternalLoggerFactory.*getInstance*(InternalThreadLocalMap.**class**);  
  
 **private static final int *DEFAULT\_ARRAY\_LIST\_INITIAL\_CAPACITY*** = 8;  
 **private static final int *STRING\_BUILDER\_INITIAL\_SIZE***;  
 **private static final int *STRING\_BUILDER\_MAX\_SIZE***;  
  
 **public static final** Object ***UNSET*** = **new** Object();  
  
 **static** {

**//初始化大小** ***STRING\_BUILDER\_INITIAL\_SIZE*** = SystemPropertyUtil.*getInt*(**"io.netty.threadLocalMap.stringBuilder.initialSize"**, 1024);  
 ***logger***.debug(**"-Dio.netty.threadLocalMap.stringBuilder.initialSize: {}"**, ***STRING\_BUILDER\_INITIAL\_SIZE***);

**//最大大小** ***STRING\_BUILDER\_MAX\_SIZE*** = SystemPropertyUtil.*getInt*(**"io.netty.threadLocalMap.stringBuilder.maxSize"**, 1024 \* 4);  
 ***logger***.debug(**"-Dio.netty.threadLocalMap.stringBuilder.maxSize: {}"**, ***STRING\_BUILDER\_MAX\_SIZE***);  
 }  
  
 **public static** InternalThreadLocalMap getIfSet() {  
 Thread thread = Thread.*currentThread*();  
 **if** (thread **instanceof** FastThreadLocalThread) {  
 **return** ((FastThreadLocalThread) thread).threadLocalMap();  
 }  
 **return *slowThreadLocalMap***.get();  
 }  
  
 **public static** InternalThreadLocalMap get() {  
 Thread thread = Thread.*currentThread*();  
 **if** (thread **instanceof** FastThreadLocalThread) {  
 **return** *fastGet*((FastThreadLocalThread) thread);  
 } **else** {  
 **return** *slowGet*();  
 }  
 }  
  
 **private static** InternalThreadLocalMap fastGet(FastThreadLocalThread thread) {  
 InternalThreadLocalMap threadLocalMap = thread.threadLocalMap();  
 **if** (threadLocalMap == **null**) {  
 thread.setThreadLocalMap(threadLocalMap = **new** InternalThreadLocalMap());  
 }  
 **return** threadLocalMap;  
 }  
  
 **private static** InternalThreadLocalMap slowGet() {  
 ThreadLocal<InternalThreadLocalMap> slowThreadLocalMap = UnpaddedInternalThreadLocalMap.***slowThreadLocalMap***;  
 InternalThreadLocalMap ret = slowThreadLocalMap.get();  
 **if** (ret == **null**) {  
 ret = **new** InternalThreadLocalMap();  
 slowThreadLocalMap.set(ret);  
 }  
 **return** ret;  
 }  
  
 **public static void** remove() {  
 Thread thread = Thread.*currentThread*();  
 **if** (thread **instanceof** FastThreadLocalThread) {  
 ((FastThreadLocalThread) thread).setThreadLocalMap(**null**);  
 } **else** {  
 ***slowThreadLocalMap***.remove();  
 }  
 }  
  
 **public static void** destroy() {  
 ***slowThreadLocalMap***.remove();  
 }  
  
 **public static int** nextVariableIndex() {  
 **int** index = ***nextIndex***.getAndIncrement();  
 **if** (index < 0) {  
 ***nextIndex***.decrementAndGet();  
 **throw new** IllegalStateException(**"too many thread-local indexed variables"**);  
 }  
 **return** index;  
 }  
  
 **public static int** lastVariableIndex() {  
 **return *nextIndex***.get() - 1;  
 }  
  
 *// Cache line padding (must be public)  
 // With CompressedOops enabled, an instance of this class should occupy at least 128 bytes.* **public long rp1**, **rp2**, **rp3**, **rp4**, **rp5**, **rp6**, **rp7**, **rp8**, **rp9**;  
  
 **private** InternalThreadLocalMap() {  
 **super**(*newIndexedVariableTable*());  
 }  
  
 **private static** Object[] newIndexedVariableTable() {  
 Object[] array = **new** Object[32];  
 Arrays.*fill*(array, ***UNSET***);  
 **return** array;  
 }  
  
 **public int** size() {  
 **int** count = 0;  
  
 **if** (**futureListenerStackDepth** != 0) {  
 count ++;  
 }  
 **if** (**localChannelReaderStackDepth** != 0) {  
 count ++;  
 }  
 **if** (**handlerSharableCache** != **null**) {  
 count ++;  
 }  
 **if** (**counterHashCode** != **null**) {  
 count ++;  
 }  
 **if** (**random** != **null**) {  
 count ++;  
 }  
 **if** (**typeParameterMatcherGetCache** != **null**) {  
 count ++;  
 }  
 **if** (**typeParameterMatcherFindCache** != **null**) {  
 count ++;  
 }  
 **if** (**stringBuilder** != **null**) {  
 count ++;  
 }  
 **if** (**charsetEncoderCache** != **null**) {  
 count ++;  
 }  
 **if** (**charsetDecoderCache** != **null**) {  
 count ++;  
 }  
 **if** (**arrayList** != **null**) {  
 count ++;  
 }  
  
 **for** (Object o: **indexedVariables**) {  
 **if** (o != ***UNSET***) {  
 count ++;  
 }  
 }  
  
 *// We should subtract 1 from the count because the first element in 'indexedVariables' is reserved  
 // by 'FastThreadLocal' to keep the list of 'FastThreadLocal's to remove on 'FastThreadLocal.removeAll()'.* **return** count - 1;  
 }  
  
 **public** StringBuilder stringBuilder() {  
 StringBuilder sb = **stringBuilder**;  
 **if** (sb == **null**) {  
 **return stringBuilder** = **new** StringBuilder(***STRING\_BUILDER\_INITIAL\_SIZE***);  
 }  
 **if** (sb.capacity() > ***STRING\_BUILDER\_MAX\_SIZE***) {  
 sb.setLength(***STRING\_BUILDER\_INITIAL\_SIZE***);  
 sb.trimToSize();  
 }  
 sb.setLength(0);  
 **return** sb;  
 }  
  
 **public** Map<Charset, CharsetEncoder> charsetEncoderCache() {  
 Map<Charset, CharsetEncoder> cache = **charsetEncoderCache**;  
 **if** (cache == **null**) {  
 **charsetEncoderCache** = cache = **new** IdentityHashMap<Charset, CharsetEncoder>();  
 }  
 **return** cache;  
 }  
  
 **public** Map<Charset, CharsetDecoder> charsetDecoderCache() {  
 Map<Charset, CharsetDecoder> cache = **charsetDecoderCache**;  
 **if** (cache == **null**) {  
 **charsetDecoderCache** = cache = **new** IdentityHashMap<Charset, CharsetDecoder>();  
 }  
 **return** cache;  
 }  
  
 **public** <E> ArrayList<E> arrayList() {  
 **return** arrayList(***DEFAULT\_ARRAY\_LIST\_INITIAL\_CAPACITY***);  
 }  
  
 @SuppressWarnings(**"unchecked"**)  
 **public** <E> ArrayList<E> arrayList(**int** minCapacity) {  
 ArrayList<E> list = (ArrayList<E>) **arrayList**;  
 **if** (list == **null**) {  
 **arrayList** = **new** ArrayList<Object>(minCapacity);  
 **return** (ArrayList<E>) **arrayList**;  
 }  
 list.clear();  
 list.ensureCapacity(minCapacity);  
 **return** list;  
 }  
  
 **public int** futureListenerStackDepth() {  
 **return futureListenerStackDepth**;  
 }  
  
 **public void** setFutureListenerStackDepth(**int** futureListenerStackDepth) {  
 **this**.**futureListenerStackDepth** = futureListenerStackDepth;  
 }  
  
 **public** ThreadLocalRandom random() {  
 ThreadLocalRandom r = **random**;  
 **if** (r == **null**) {  
 **random** = r = **new** ThreadLocalRandom();  
 }  
 **return** r;  
 }  
  
 **public** Map<Class<?>, TypeParameterMatcher> typeParameterMatcherGetCache() {  
 Map<Class<?>, TypeParameterMatcher> cache = **typeParameterMatcherGetCache**;  
 **if** (cache == **null**) {  
 **typeParameterMatcherGetCache** = cache = **new** IdentityHashMap<Class<?>, TypeParameterMatcher>();  
 }  
 **return** cache;  
 }  
  
 **public** Map<Class<?>, Map<String, TypeParameterMatcher>> typeParameterMatcherFindCache() {  
 Map<Class<?>, Map<String, TypeParameterMatcher>> cache = **typeParameterMatcherFindCache**;  
 **if** (cache == **null**) {  
 **typeParameterMatcherFindCache** = cache = **new** IdentityHashMap<Class<?>, Map<String, TypeParameterMatcher>>();  
 }  
 **return** cache;  
 }  
  
 **public** IntegerHolder counterHashCode() {  
 **return counterHashCode**;  
 }  
  
 **public void** setCounterHashCode(IntegerHolder counterHashCode) {  
 **this**.**counterHashCode** = counterHashCode;  
 }  
  
 **public** Map<Class<?>, Boolean> handlerSharableCache() {  
 Map<Class<?>, Boolean> cache = **handlerSharableCache**;  
 **if** (cache == **null**) {  
 *// Start with small capacity to keep memory overhead as low as possible.* **handlerSharableCache** = cache = **new** WeakHashMap<Class<?>, Boolean>(4);  
 }  
 **return** cache;  
 }  
  
 **public int** localChannelReaderStackDepth() {  
 **return localChannelReaderStackDepth**;  
 }  
  
 **public void** setLocalChannelReaderStackDepth(**int** localChannelReaderStackDepth) {  
 **this**.**localChannelReaderStackDepth** = localChannelReaderStackDepth;  
 }  
  
 **public** Object indexedVariable(**int** index) {  
 Object[] lookup = **indexedVariables**;  
 **return** index < lookup.**length**? lookup[index] : ***UNSET***;  
 }  
  
 */\*\*  
 \** ***@return*** *{****@code*** *true} if and only if a new thread-local variable has been created  
 \*/* **public boolean** setIndexedVariable(**int** index, Object value) {  
 Object[] lookup = **indexedVariables**;  
 **if** (index < lookup.**length**) {  
 Object oldValue = lookup[index];  
 lookup[index] = value;  
 **return** oldValue == ***UNSET***;  
 } **else** {  
 expandIndexedVariableTableAndSet(index, value);  
 **return true**;  
 }  
 }  
  
 **private void** expandIndexedVariableTableAndSet(**int** index, Object value) {  
 Object[] oldArray = **indexedVariables**;  
 **final int** oldCapacity = oldArray.**length**;  
 **int** newCapacity = index;  
 newCapacity |= newCapacity >>> 1;  
 newCapacity |= newCapacity >>> 2;  
 newCapacity |= newCapacity >>> 4;  
 newCapacity |= newCapacity >>> 8;  
 newCapacity |= newCapacity >>> 16;  
 newCapacity ++;  
  
 Object[] newArray = Arrays.*copyOf*(oldArray, newCapacity);  
 Arrays.*fill*(newArray, oldCapacity, newArray.**length**, ***UNSET***);  
 newArray[index] = value;  
 **indexedVariables** = newArray;  
 }  
  
 **public** Object removeIndexedVariable(**int** index) {  
 Object[] lookup = **indexedVariables**;  
 **if** (index < lookup.**length**) {  
 Object v = lookup[index];  
 lookup[index] = ***UNSET***;  
 **return** v;  
 } **else** {  
 **return *UNSET***;  
 }  
 }  
  
 **public boolean** isIndexedVariableSet(**int** index) {  
 Object[] lookup = **indexedVariables**;  
 **return** index < lookup.**length** && lookup[index] != ***UNSET***;  
 }  
}

##### io.netty.util.concurrent.FastThreadLocal

**public class** FastThreadLocal<V> {  
  
 **private static final int *variablesToRemoveIndex*** = InternalThreadLocalMap.*nextVariableIndex*();  
  
 */\*\*  
 \* Removes all {****@link*** *FastThreadLocal} variables bound to the current thread. This operation is useful when you  
 \* are in a container environment, and you don't want to leave the thread local variables in the threads you do not  
 \* manage.  
 \*/* **public static void** removeAll() {  
 InternalThreadLocalMap threadLocalMap = InternalThreadLocalMap.*getIfSet*();  
 **if** (threadLocalMap == **null**) {  
 **return**;  
 }  
  
 **try** {  
 Object v = threadLocalMap.indexedVariable(***variablesToRemoveIndex***);  
 **if** (v != **null** && v != InternalThreadLocalMap.***UNSET***) {  
 @SuppressWarnings(**"unchecked"**)  
 Set<FastThreadLocal<?>> variablesToRemove = (Set<FastThreadLocal<?>>) v;  
 FastThreadLocal<?>[] variablesToRemoveArray =  
 variablesToRemove.toArray(**new** FastThreadLocal[variablesToRemove.size()]);  
 **for** (FastThreadLocal<?> tlv: variablesToRemoveArray) {  
 tlv.remove(threadLocalMap);  
 }  
 }  
 } **finally** {  
 InternalThreadLocalMap.*remove*();  
 }  
 }  
  
 */\*\*  
 \* Returns the number of thread local variables bound to the current thread.  
 \*/* **public static int** size() {  
 InternalThreadLocalMap threadLocalMap = InternalThreadLocalMap.*getIfSet*();  
 **if** (threadLocalMap == **null**) {  
 **return** 0;  
 } **else** {  
 **return** threadLocalMap.size();  
 }  
 }

*/\*\*  
 \* Destroys the data structure that keeps all {****@link*** *FastThreadLocal} variables accessed from  
 \* non-{****@link*** *FastThreadLocalThread}s. This operation is useful when you are in a container environment, and you  
 \* do not want to leave the thread local variables in the threads you do not manage. Call this method when your  
 \* application is being unloaded from the container.  
 \*/***public static void** destroy() {  
 InternalThreadLocalMap.*destroy*();  
}

}

##### io.netty.channel.WriteBufferWaterMark

***1.WriteBufferWaterMark被用于设置写缓冲区低水位标志和高水位标志。***

***2.如果写缓冲区中排队的字节数超过高水位标志，Channel.isWritable()将开始返回false***

***3.如果写缓冲区中排队的字节数超过高水位标志，然后下降到低水位标志以下，Channel.isWritable()将再次开始返回true***

**public final class** WriteBufferWaterMark {  
  
 **private static final int *DEFAULT\_LOW\_WATER\_MARK*** = 32 \* 1024;  
 **private static final int *DEFAULT\_HIGH\_WATER\_MARK*** = 64 \* 1024;  
  
 **public static final** WriteBufferWaterMark ***DEFAULT*** =  
 **new** WriteBufferWaterMark(***DEFAULT\_LOW\_WATER\_MARK***, ***DEFAULT\_HIGH\_WATER\_MARK***, **false**);  
  
 **private final int low**;  
 **private final int high**;  
  
 */\*\*  
 \* Create a new instance.  
 \*  
 \** ***@param low*** *low water mark for write buffer.  
 \** ***@param high*** *high water mark for write buffer  
 \*/* **public** WriteBufferWaterMark(**int** low, **int** high) {  
 **this**(low, high, **true**);  
 }  
  
 */\*\*  
 \* This constructor is needed to keep backward-compatibility.  
 \*/* WriteBufferWaterMark(**int** low, **int** high, **boolean** validate) {  
 **if** (validate) {  
 **if** (low < 0) {  
 **throw new** IllegalArgumentException(**"write buffer's low water mark must be >= 0"**);  
 }  
 **if** (high < low) {  
 **throw new** IllegalArgumentException(  
 **"write buffer's high water mark cannot be less than "** +  
 **" low water mark ("** + low + **"): "** +  
 high);  
 }  
 }  
 **this**.**low** = low;  
 **this**.**high** = high;  
 }  
  
 */\*\*  
 \* Returns the low water mark for the write buffer.  
 \*/* **public int** low() {  
 **return low**;  
 }  
  
 */\*\*  
 \* Returns the high water mark for the write buffer.  
 \*/* **public int** high() {  
 **return high**;  
 }  
  
 @Override  
 **public** String toString() {  
 StringBuilder builder = **new** StringBuilder(55)  
 .append(**"WriteBufferWaterMark(low: "**)  
 .append(**low**)  
 .append(**", high: "**)  
 .append(**high**)  
 .append(**")"**);  
 **return** builder.toString();  
 }  
  
}

# **[NETTY源码分析之揭开REACTOR线程的面纱（一）](https://www.cnblogs.com/zhangboyu/p/7452607.html)**

netty最核心的就是reactor线程，对应项目中使用广泛的NioEventLoop，那么NioEventLoop里面到底在干些什么事？netty是如何保证事件循环的高效轮询和任务的及时执行？又是如何来优雅地fix掉jdk的nio bug？带着这些疑问，本篇文章将庖丁解牛，带你逐步了解netty reactor线程的真相[源码基于4.1.6.Final]

## **reactor 线程的启动**

NioEventLoop的run方法是reactor线程的主体，在第一次添加任务的时候被启动

NioEventLoop 父类 SingleThreadEventExecutor 的execute方法

@Overridepublic void execute(Runnable task) {

...

boolean inEventLoop = inEventLoop();

if (inEventLoop) {

addTask(task);

} else {

startThread();

addTask(task);

...

}

...

}

外部线程在往任务队列里面添加任务的时候执行 startThread() ，netty会判断reactor线程有没有被启动，如果没有被启动，那就启动线程再往任务队列里面添加任务

private void startThread() {

if (STATE\_UPDATER.get(this) == ST\_NOT\_STARTED) {

if (STATE\_UPDATER.compareAndSet(this, ST\_NOT\_STARTED, ST\_STARTED)) {

doStartThread();

}

}

}

SingleThreadEventExecutor 在执行doStartThread的时候，会调用内部执行器executor的execute方法，将调用NioEventLoop的run方法的过程封装成一个runnable塞到一个线程中去执行

private void doStartThread() {

...

executor.execute(new Runnable() {

@Override

public void run() {

thread = Thread.currentThread();

...

SingleThreadEventExecutor.this.run();

...

}

}

}

该线程就是executor创建，对应netty的reactor线程实体。executor 默认是ThreadPerTaskExecutor

默认情况下，ThreadPerTaskExecutor 在每次执行execute 方法的时候都会通过DefaultThreadFactory创建一个FastThreadLocalThread线程，而这个线程就是netty中的reactor线程实体

ThreadPerTaskExecutor

public void execute(Runnable command) {

threadFactory.newThread(command).start();

}

关于为啥是 ThreadPerTaskExecutor 和 DefaultThreadFactory的组合来new一个FastThreadLocalThread，这里就不再详细描述，通过下面几段代码来简单说明

标准的netty程序会调用到NioEventLoopGroup的父类MultithreadEventExecutorGroup的如下代码

protected MultithreadEventExecutorGroup(int nThreads, Executor executor,

EventExecutorChooserFactory chooserFactory, Object... args) {

if (executor == null) {

executor = new ThreadPerTaskExecutor(newDefaultThreadFactory());

}

}

然后通过newChild的方式传递给NioEventLoop

@Overrideprotected EventLoop newChild(Executor executor, Object... args) throws Exception {

return new NioEventLoop(this, executor, (SelectorProvider) args[0],

((SelectStrategyFactory) args[1]).newSelectStrategy(), (RejectedExecutionHandler) args[2]);

}

关于reactor线程的创建和启动就先讲这么多，我们总结一下：netty的reactor线程在添加一个任务的时候被创建，该线程实体为 FastThreadLocalThread(这玩意以后会开篇文章重点讲讲)，最后线程执行主体为NioEventLoop的run方法。

## **reactor 线程的执行**

那么下面我们就重点剖析一下 NioEventLoop 的run方法

@Overrideprotected void run() {

for (;;) {

try {

switch (selectStrategy.calculateStrategy(selectNowSupplier, hasTasks())) {

case SelectStrategy.CONTINUE:

continue;

case SelectStrategy.SELECT:

select(wakenUp.getAndSet(false));

if (wakenUp.get()) {

selector.wakeup();

}

default:

// fallthrough

}

processSelectedKeys();

runAllTasks(...);

}

} catch (Throwable t) {

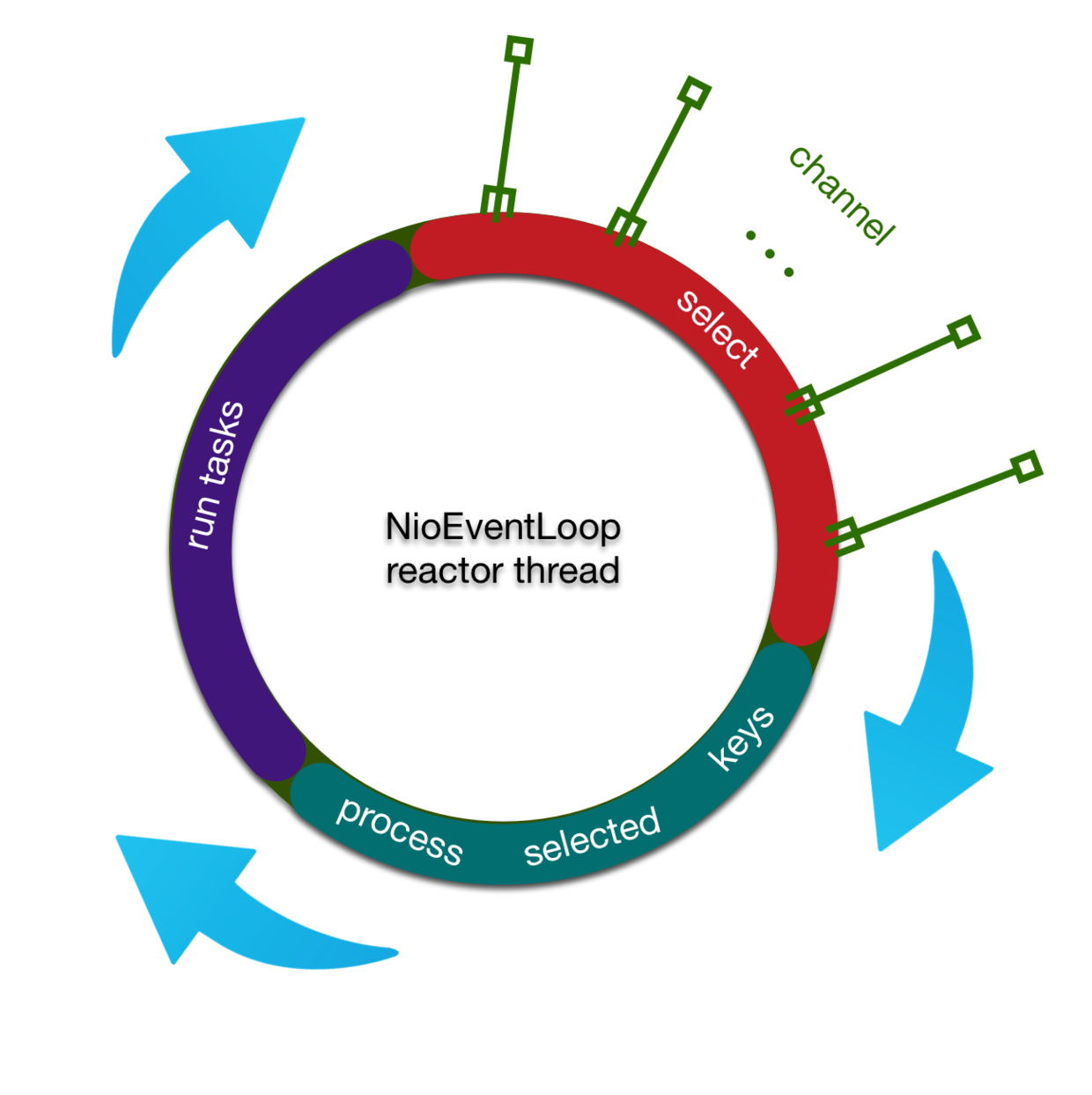
handleLoopException(t);

}

...

}

我们抽取出主干，reactor线程做的事情其实很简单，用下面一幅图就可以说明

[](http://images2017.cnblogs.com/blog/782702/201708/782702-20170830131228890-503330369.png)

reactor action

reactor线程大概做的事情分为对三个步骤不断循环

1.首先轮询注册到reactor线程对用的selector上的所有的channel的IO事件

select(wakenUp.getAndSet(false));if (wakenUp.get()) {

selector.wakeup();

}

2.处理产生网络IO事件的channel

processSelectedKeys();

3.处理任务队列

runAllTasks(...);

下面对每个步骤详细说明

### **select操作**

select(wakenUp.getAndSet(false));if (wakenUp.get()) {

selector.wakeup();

}

wakenUp 表示是否应该唤醒正在阻塞的select操作，可以看到netty在进行一次新的loop之前，都会将wakeUp 被设置成false，标志新的一轮loop的开始，具体的select操作我们也拆分开来看

1.定时任务截止事时间快到了，中断本次轮询

int selectCnt = 0;long currentTimeNanos = System.nanoTime();long selectDeadLineNanos = currentTimeNanos + delayNanos(currentTimeNanos);

for (;;) {

long timeoutMillis = (selectDeadLineNanos - currentTimeNanos + 500000L) / 1000000L;

if (timeoutMillis <= 0) {

if (selectCnt == 0) {

selector.selectNow();

selectCnt = 1;

}

break;

}

....

}

我们可以看到，NioEventLoop中reactor线程的select操作也是一个for循环，在for循环第一步中，如果发现当前的定时任务队列中有任务的截止事件快到了(<=0.5ms)，就跳出循环。此外，跳出之前如果发现目前为止还没有进行过select操作（if (selectCnt == 0)），那么就调用一次selectNow()，该方法会立即返回，不会阻塞

这里说明一点，netty里面定时任务队列是按照延迟时间从小到大进行排序， delayNanos(currentTimeNanos)方法即取出第一个定时任务的延迟时间

protected long delayNanos(long currentTimeNanos) {

ScheduledFutureTask<?> scheduledTask = peekScheduledTask();

if (scheduledTask == null) {

return SCHEDULE\_PURGE\_INTERVAL;

}

return scheduledTask.delayNanos(currentTimeNanos);

}

关于netty的任务队列(包括普通任务，定时任务，tail task)相关的细节后面会另起一片文章，这里不过多展开

2.轮询过程中发现有任务加入，中断本次轮询

for (;;) {

// 1.定时任务截至事时间快到了，中断本次轮询

...

// 2.轮询过程中发现有任务加入，中断本次轮询

if (hasTasks() && wakenUp.compareAndSet(false, true)) {

selector.selectNow();

selectCnt = 1;

break;

}

....

}

netty为了保证任务队列能够及时执行，在进行阻塞select操作的时候会判断任务队列是否为空，如果不为空，就执行一次非阻塞select操作，跳出循环

3.阻塞式select操作

for (;;) {

// 1.定时任务截至事时间快到了，中断本次轮询

...

// 2.轮询过程中发现有任务加入，中断本次轮询

...

// 3.阻塞式select操作

int selectedKeys = selector.select(timeoutMillis);

selectCnt ++;

if (selectedKeys != 0 || oldWakenUp || wakenUp.get() || hasTasks() || hasScheduledTasks()) {

break;

}

....

}

执行到这一步，说明netty任务队列里面队列为空，并且所有定时任务延迟时间还未到(大于0.5ms)，于是，在这里进行一次阻塞select操作，截止到第一个定时任务的截止时间

这里，我们可以问自己一个问题，如果第一个定时任务的延迟非常长，比如一个小时，那么有没有可能线程一直阻塞在select操作，当然有可能！But，只要在这段时间内，有新任务加入，该阻塞就会被释放

外部线程调用execute方法添加任务

@Overridepublic void execute(Runnable task) {

...

wakeup(inEventLoop); // inEventLoop为false

...

}

调用wakeup方法唤醒selector阻塞

protected void wakeup(boolean inEventLoop) {

if (!inEventLoop && wakenUp.compareAndSet(false, true)) {

selector.wakeup();

}

}

可以看到，在外部线程添加任务的时候，会调用wakeup方法来唤醒 selector.select(timeoutMillis)

阻塞select操作结束之后，netty又做了一系列的状态判断来决定是否中断本次轮询，中断本次轮询的条件有

* 轮询到IO事件 （selectedKeys != 0）
* oldWakenUp 参数为true
* 任务队列里面有任务（hasTasks）
* 第一个定时任务即将要被执行 （hasScheduledTasks（））
* 用户主动唤醒（wakenUp.get()）

4.解决jdk的nio bug

关于该bug的描述见 [http://bugs.java.com/bugdatabase/view\_bug.do?bug\_id=6595055)](http://bugs.java.com/bugdatabase/view_bug.do?bug_id=6595055" \t "https://www.cnblogs.com/zhangboyu/p/_blank)

该bug会导致Selector一直空轮询，最终导致cpu 100%，nio server不可用，严格意义上来说，netty没有解决jdk的bug，而是通过一种方式来巧妙地避开了这个bug，具体做法如下

long currentTimeNanos = System.nanoTime();for (;;) {

// 1.定时任务截止事时间快到了，中断本次轮询

...

// 2.轮询过程中发现有任务加入，中断本次轮询

...

// 3.阻塞式select操作

selector.select(timeoutMillis);

// 4.解决jdk的nio bug

long time = System.nanoTime();

if (time - TimeUnit.MILLISECONDS.toNanos(timeoutMillis) >= currentTimeNanos) {

selectCnt = 1;

} else if (SELECTOR\_AUTO\_REBUILD\_THRESHOLD > 0 &&

selectCnt >= SELECTOR\_AUTO\_REBUILD\_THRESHOLD) {

rebuildSelector();

selector = this.selector;

selector.selectNow();

selectCnt = 1;

break;

}

currentTimeNanos = time;

...

}

netty 会在每次进行 selector.select(timeoutMillis) 之前记录一下开始时间currentTimeNanos，在select之后记录一下结束时间，判断select操作是否至少持续了timeoutMillis秒（这里将time - TimeUnit.MILLISECONDS.toNanos(timeoutMillis) >= currentTimeNanos改成time - currentTimeNanos >= TimeUnit.MILLISECONDS.toNanos(timeoutMillis)或许更好理解一些）,  
如果持续的时间大于等于timeoutMillis，说明就是一次有效的轮询，重置selectCnt标志，否则，表明该阻塞方法并没有阻塞这么长时间，可能触发了jdk的空轮询bug，当空轮询的次数超过一个阀值的时候，默认是512，就开始重建selector

空轮询阀值相关的设置代码如下

int selectorAutoRebuildThreshold = SystemPropertyUtil.getInt("io.netty.selectorAutoRebuildThreshold", 512);if (selectorAutoRebuildThreshold < MIN\_PREMATURE\_SELECTOR\_RETURNS) {

selectorAutoRebuildThreshold = 0;

}

SELECTOR\_AUTO\_REBUILD\_THRESHOLD = selectorAutoRebuildThreshold;

下面我们简单描述一下netty 通过rebuildSelector来fix空轮询bug的过程，rebuildSelector的操作其实很简单：new一个新的selector，将之前注册到老的selector上的的channel重新转移到新的selector上。我们抽取完主要代码之后的骨架如下

public void rebuildSelector() {

final Selector oldSelector = selector;

final Selector newSelector;

newSelector = openSelector();

int nChannels = 0;

try {

for (;;) {

for (SelectionKey key: oldSelector.keys()) {

Object a = key.attachment();

if (!key.isValid() || key.channel().keyFor(newSelector) != null) {

continue;

}

int interestOps = key.interestOps();

key.cancel();

SelectionKey newKey = key.channel().register(newSelector, interestOps, a);

if (a instanceof AbstractNioChannel) {

((AbstractNioChannel) a).selectionKey = newKey;

}

nChannels ++;

}

break;

}

} catch (ConcurrentModificationException e) {

// Probably due to concurrent modification of the key set.

continue;

}

selector = newSelector;

oldSelector.close();

}

首先，通过openSelector()方法创建一个新的selector，然后执行一个死循环，只要执行过程中出现过一次并发修改selectionKeys异常，就重新开始转移

具体的转移步骤为

1. 拿到有效的key
2. 取消该key在旧的selector上的事件注册
3. 将该key对应的channel注册到新的selector上
4. 重新绑定channel和新的key的关系

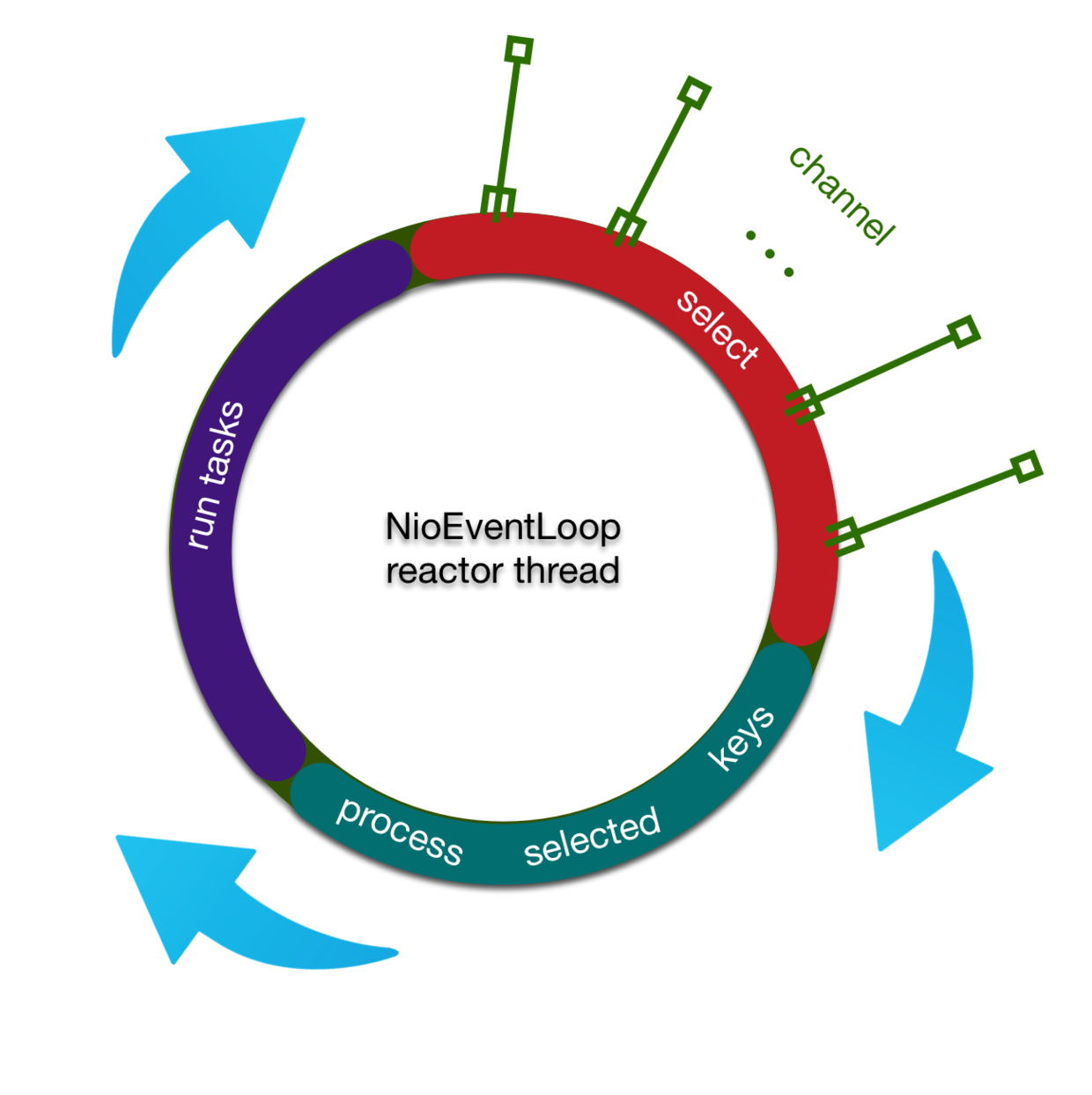
转移完成之后，就可以将原有的selector废弃，后面所有的轮询都是在新的selector进行

最后，我们总结reactor线程select步骤做的事情：不断地轮询是否有IO事件发生，并且在轮询的过程中不断检查是否有定时任务和普通任务，保证了netty的任务队列中的任务得到有效执行，轮询过程顺带用一个计数器避开了了jdk空轮询的bug，过程清晰明了

由于篇幅原因，下面两个过程将分别放到一篇文章中去讲述，尽请期待

# **[NETTY源码分析之揭开REACTOR线程的面纱（二）](https://www.cnblogs.com/zhangboyu/p/7452611.html)**

如果你对netty的reactor线程不了解，建议先看下上一篇文章[netty源码分析之揭开reactor线程的面纱（一）](http://www.jianshu.com/p/0d0eece6d467" \t "https://www.cnblogs.com/zhangboyu/p/_blank)，这里再把reactor中的三个步骤的图贴一下

[](http://images2017.cnblogs.com/blog/782702/201708/782702-20170830131328390-243400633.png)

reactor线程

我们已经了解到netty reactor线程的第一步是轮询出注册在selector上面的IO事件（select），那么接下来就要处理这些IO事件（process selected keys），本篇文章我们将一起来探讨netty处理IO事件的细节

我们进入到reactor线程的 run 方法，找到处理IO事件的代码，如下

processSelectedKeys()；

跟进去

private void processSelectedKeys() {

if (selectedKeys != null) {

processSelectedKeysOptimized(selectedKeys.flip());

} else {

processSelectedKeysPlain(selector.selectedKeys());

}

}

我们发现处理IO事件，netty有两种选择，从名字上看，一种是处理优化过的selectedKeys，一种是正常的处理

我们对优化过的selectedKeys的处理稍微展开一下，看看netty是如何优化的，我们查看 selectedKeys 被引用过的地方，有如下代码

private SelectedSelectionKeySet selectedKeys;

private Selector NioEventLoop.openSelector() {

//...

final SelectedSelectionKeySet selectedKeySet = new SelectedSelectionKeySet();

// selectorImplClass -> sun.nio.ch.SelectorImpl

Field selectedKeysField = selectorImplClass.getDeclaredField("selectedKeys");

Field publicSelectedKeysField = selectorImplClass.getDeclaredField("publicSelectedKeys");

selectedKeysField.setAccessible(true);

publicSelectedKeysField.setAccessible(true);

selectedKeysField.set(selector, selectedKeySet);

publicSelectedKeysField.set(selector, selectedKeySet);

//...

selectedKeys = selectedKeySet;

}

首先，selectedKeys是一个 SelectedSelectionKeySet 类对象，在NioEventLoop 的 openSelector 方法中创建，之后就通过反射将selectedKeys与 sun.nio.ch.SelectorImpl 中的两个field绑定

sun.nio.ch.SelectorImpl 中我们可以看到，这两个field其实是两个HashSet

// Public views of the key setsprivate Set<SelectionKey> publicKeys; // Immutableprivate Set<SelectionKey> publicSelectedKeys; // Removal allowed, but not additionprotected SelectorImpl(SelectorProvider sp) {

super(sp);

keys = new HashSet<SelectionKey>();

selectedKeys = new HashSet<SelectionKey>();

if (Util.atBugLevel("1.4")) {

publicKeys = keys;

publicSelectedKeys = selectedKeys;

} else {

publicKeys = Collections.unmodifiableSet(keys);

publicSelectedKeys = Util.ungrowableSet(selectedKeys);

}

}

selector在调用select()族方法的时候，如果有IO事件发生，就会往里面的两个field中塞相应的selectionKey(具体怎么塞有待研究)，即相当于往一个hashSet中add元素，既然netty通过反射将jdk中的两个field替换掉，那我们就应该意识到是不是netty自定义的SelectedSelectionKeySet在add方法做了某些优化呢？

带着这个疑问，我们进入到 SelectedSelectionKeySet 类中探个究竟

final class SelectedSelectionKeySet extends AbstractSet<SelectionKey> {

private SelectionKey[] keysA;

private int keysASize;

private SelectionKey[] keysB;

private int keysBSize;

private boolean isA = true;

SelectedSelectionKeySet() {

keysA = new SelectionKey[1024];

keysB = keysA.clone();

}

@Override

public boolean add(SelectionKey o) {

if (o == null) {

return false;

}

if (isA) {

int size = keysASize;

keysA[size ++] = o;

keysASize = size;

if (size == keysA.length) {

doubleCapacityA();

}

} else {

int size = keysBSize;

keysB[size ++] = o;

keysBSize = size;

if (size == keysB.length) {

doubleCapacityB();

}

}

return true;

}

private void doubleCapacityA() {

SelectionKey[] newKeysA = new SelectionKey[keysA.length << 1];

System.arraycopy(keysA, 0, newKeysA, 0, keysASize);

keysA = newKeysA;

}

private void doubleCapacityB() {

SelectionKey[] newKeysB = new SelectionKey[keysB.length << 1];

System.arraycopy(keysB, 0, newKeysB, 0, keysBSize);

keysB = newKeysB;

}

SelectionKey[] flip() {

if (isA) {

isA = false;

keysA[keysASize] = null;

keysBSize = 0;

return keysA;

} else {

isA = true;

keysB[keysBSize] = null;

keysASize = 0;

return keysB;

}

}

@Override

public int size() {

if (isA) {

return keysASize;

} else {

return keysBSize;

}

}

@Override

public boolean remove(Object o) {

return false;

}

@Override

public boolean contains(Object o) {

return false;

}

@Override

public Iterator<SelectionKey> iterator() {

throw new UnsupportedOperationException();

}

}

该类其实很简单，继承了 AbstractSet，说明该类可以当作一个set来用，但是底层使用两个数组来交替使用，在add方法中，判断当前使用哪个数组，找到对应的数组，然后经历下面三个步骤  
1.将SelectionKey塞到该数组的逻辑尾部  
2.更新该数组的逻辑长度+1  
3.如果该数组的逻辑长度等于数组的物理长度，就将该数组扩容

我们可以看到，待程序跑过一段时间，等数组的长度足够长，每次在轮询到nio事件的时候，netty只需要O(1)的时间复杂度就能将 SelectionKey 塞到 set中去，而jdk底层使用的hashSet需要O(lgn)的时间复杂度

这里关于为何使用两个数组循环交替使用，其实我也是很费解，思考了很久，查找SelectedSelectionKeySet 所有使用的地方，我觉得使用一个数组就能够达到优化目的，并且不用每次都判断使用哪个数组，所以对于该问题，我提了一个issue给netty官方，官方也给出了答复说会跟进，issue链接：[https://github.com/netty/netty/issues/6058#](https://github.com/netty/netty/issues/6058" \t "https://www.cnblogs.com/zhangboyu/p/_blank)

关于netty对SelectionKeySet的优化我们暂时就跟这么多，下面我们继续跟netty对IO事件的处理，转到processSelectedKeysOptimized

private void processSelectedKeysOptimized(SelectionKey[] selectedKeys) {

for (int i = 0;; i ++) {

// 1.取出IO事件以及对应的channel

final SelectionKey k = selectedKeys[i];

if (k == null) {

break;

}

selectedKeys[i] = null;

final Object a = k.attachment();

// 2.处理该channel

if (a instanceof AbstractNioChannel) {

processSelectedKey(k, (AbstractNioChannel) a);

} else {

NioTask<SelectableChannel> task = (NioTask<SelectableChannel>) a;

processSelectedKey(k, task);

}

// 3.判断是否该再来次轮询

if (needsToSelectAgain) {

for (;;) {

i++;

if (selectedKeys[i] == null) {

break;

}

selectedKeys[i] = null;

}

selectAgain();

selectedKeys = this.selectedKeys.flip();

i = -1;

}

}

}

我们可以将该过程分为以下三个步骤

1.取出IO事件以及对应的netty channel类

这里其实也能体会到优化过的 SelectedSelectionKeySet 的好处，遍历的时候遍历的是数组，相对jdk原生的HashSet效率有所提高

拿到当前SelectionKey之后，将selectedKeys[i]置为null，这里简单解释一下这么做的理由：想象一下这种场景，假设一个NioEventLoop平均每次轮询出N个IO事件，高峰期轮询出3N个事件，那么selectedKeys的物理长度要大于等于3N，如果每次处理这些key，不置selectedKeys[i]为空，那么高峰期一过，这些保存在数组尾部的selectedKeys[i]对应的SelectionKey将一直无法被回收，SelectionKey对应的对象可能不大，但是要知道，它可是有attachment的，这里的attachment具体是什么下面会讲到，但是有一点我们必须清楚，attachment可能很大，这样一来，这些元素是GC root可达的，很容易造成gc不掉，内存泄漏就发生了

这个bug在 4.0.19.Final版本中被修复，建议使用netty的项目升级到最新版本^^

2.处理该channel

拿到对应的attachment之后，netty做了如下判断

if (a instanceof AbstractNioChannel) {

processSelectedKey(k, (AbstractNioChannel) a);

}

源码读到这，我们需要思考为啥会有这么一条判断，凭什么说attachment可能会是 AbstractNioChannel对象？

我们的思路应该是找到底层selector, 然后在selector调用register方法的时候，看一下注册到selector上的对象到底是什么鬼，我们使用intellij的全局搜索引用功能，最终在 AbstractNioChannel中搜索到如下方法

protected void doRegister() throws Exception {

// ...

selectionKey = javaChannel().register(eventLoop().selector, 0, this);

// ...

}

javaChannel() 返回netty类AbstractChannel对应的jdk底层channel对象

protected SelectableChannel javaChannel() {

return ch;

}

我们查看到SelectableChannel方法，结合netty的 doRegister() 方法，我们不难推论出，netty的轮询注册机制其实是将AbstractNioChannel内部的jdk类SelectableChannel对象注册到jdk类Selctor对象上去，并且将AbstractNioChannel作为SelectableChannel对象的一个attachment附属上，这样再jdk轮询出某条SelectableChannel有IO事件发生时，就可以直接取出AbstractNioChannel进行后续操作

下面是jdk中的register方法

//\*

//\* @param sel

//\* The selector with which this channel is to be registered

//\*

//\* @param ops

//\* The interest set for the resulting key

//\*

//\* @param att

//\* The attachment for the resulting key; may be <tt>null</tt>public abstract SelectionKey register(Selector sel, int ops, Object att)

throws ClosedChannelException;

由于篇幅原因，详细的 processSelectedKey(SelectionKey k, AbstractNioChannel ch) 过程我们单独写一篇文章来详细展开，这里就简单说一下  
1.对于boss NioEventLoop来说，轮询到的是基本上就是连接事件，后续的事情就通过他的pipeline将连接扔给一个worker NioEventLoop处理  
2.对于worker NioEventLoop来说，轮询到的基本上都是io读写事件，后续的事情就是通过他的pipeline将读取到的字节流传递给每个channelHandler来处理

上面处理attachment的时候，还有个else分支，我们也来分析一下  
else部分的代码如下

NioTask<SelectableChannel> task = (NioTask<SelectableChannel>) a;

processSelectedKey(k, task);

说明注册到selctor上的attachment还有另外一中类型，就是 NioTask，NioTask主要是用于当一个 SelectableChannel 注册到selector的时候，执行一些任务

NioTask的定义

public interface NioTask<C extends SelectableChannel> {

void channelReady(C ch, SelectionKey key) throws Exception;

void channelUnregistered(C ch, Throwable cause) throws Exception;

}

由于NioTask 在netty内部没有使用的地方，这里不过多展开

3.判断是否该再来次轮询

if (needsToSelectAgain) {

for (;;) {

i++;

if (selectedKeys[i] == null) {

break;

}

selectedKeys[i] = null;

}

selectAgain();

selectedKeys = this.selectedKeys.flip();

i = -1;

}

我们回忆一下netty的reactor线程经历前两个步骤，分别是抓取产生过的IO事件以及处理IO事件，每次在抓到IO事件之后，都会将 needsToSelectAgain 重置为false，那么什么时候needsToSelectAgain会重新被设置成true呢？

还是和前面一样的思路，我们使用intellij来帮助我们查看needsToSelectAgain被使用的地方，在NioEventLoop类中，只有下面一处将needsToSelectAgain设置为true

NioEventLoop.java

void cancel(SelectionKey key) {

key.cancel();

cancelledKeys ++;

if (cancelledKeys >= CLEANUP\_INTERVAL) {

cancelledKeys = 0;

needsToSelectAgain = true;

}

}

继续查看 cancel 函数被调用的地方

AbstractChannel.java

@Overrideprotected void doDeregister() throws Exception {

eventLoop().cancel(selectionKey());

}

不难看出，在channel从selector上移除的时候，调用cancel函数将key取消，并且当被去掉的key到达 CLEANUP\_INTERVAL 的时候，设置needsToSelectAgain为true,CLEANUP\_INTERVAL默认值为256

private static final int CLEANUP\_INTERVAL = 256;

也就是说，对于每个NioEventLoop而言，每隔256个channel从selector上移除的时候，就标记 needsToSelectAgain 为true，我们还是跳回到上面这段代码

if (needsToSelectAgain) {

for (;;) {

i++;

if (selectedKeys[i] == null) {

break;

}

selectedKeys[i] = null;

}

selectAgain();

selectedKeys = this.selectedKeys.flip();

i = -1;

}

每满256次，就会进入到if的代码块，首先，将selectedKeys的内部数组全部清空，方便被jvm垃圾回收，然后重新调用selectAgain重新填装一下 selectionKey

private void selectAgain() {

needsToSelectAgain = false;

try {

selector.selectNow();

} catch (Throwable t) {

logger.warn("Failed to update SelectionKeys.", t);

}

}

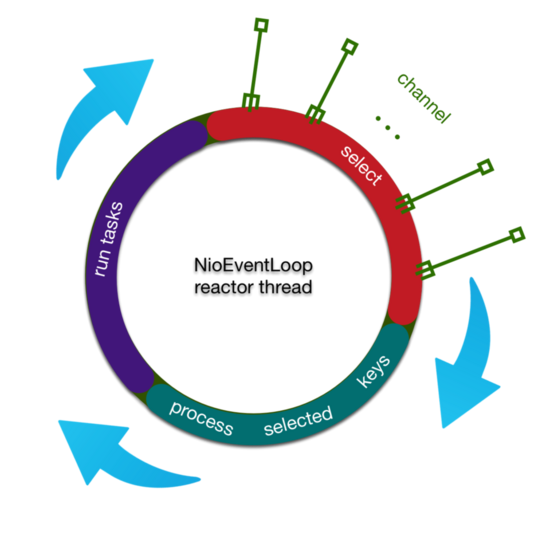
netty这么做的目的我想应该是每隔256次channel断线，重新清理一下selectionKey，保证现存的SelectionKey及时有效

到这里，我们初次阅读源码的时候对reactor的第二个步骤的了解已经足够了。总结一下：netty的reactor线程第二步做的事情为处理IO事件，netty使用数组替换掉jdk原生的HashSet来保证IO事件的高效处理，每个SelectionKey上绑定了netty类AbstractChannel对象作为attachment，在处理每个SelectionKey的时候，就可以找到AbstractChannel，然后通过pipeline的方式将处理串行到ChannelHandler，回调到用户方法

下一篇文章，我们将一起来看下netty中reactor线程中最后一步，runTasks，你将了解到netty中异步执行任务机制的细节，尽请期待

# **NETTY源码分析之揭开REACTOR线程的面纱（三）**

上两篇博文( [netty源码分析之揭开reactor线程的面纱（一）](https://link.juejin.im/?target=http://www.jianshu.com/p/0d0eece6d467" \t "https://www.codercto.com/a/_blank) ， [netty源码分析之揭开reactor线程的面纱（二）](https://link.juejin.im/?target=http://www.jianshu.com/p/467a9b41833e" \t "https://www.codercto.com/a/_blank) )已经描述了netty的reactor线程前两个步骤所处理的工作，在这里，我们用这张图片来回顾一下：



简单总结一下reactor线程三部曲

1. 轮询出IO事件
2. 处理IO事件
3. 处理任务队列

今天，我们要进行的是三部曲中的最后一曲【处理任务队列】，也就是上面图中的紫色部分。

读完本篇文章，你将了解到netty的异步task机制，定时任务的处理逻辑，这些细节可以更好地帮助你写出netty应用

## **Bnetty中的task的常见使用场景**

我们取三种典型的task使用场景来分析

### **一. 用户自定义普通任务**

ctx.channel().eventLoop().execute(new Runnable() {

@Override

public void run() {

//...

}});

我们跟进 execute 方法，看重点

@Overridepublic void execute(Runnable task) {

//...

addTask(task);

//...}

execute 方法调用 addTask 方法

protected void addTask(Runnable task) {

// ...

if (!offerTask(task)) {

reject(task);

}}

然后调用 offerTask 方法，如果offer失败，那就调用 reject 方法，通过默认的 RejectedExecutionHandler 直接抛出异常

final boolean offerTask(Runnable task) {

// ...

return taskQueue.offer(task);}复制代码

跟到 offerTask 方法，基本上task就落地了，netty内部使用一个 taskQueue 将task保存起来，那么这个 taskQueue又是何方神圣？

我们查看 taskQueue 定义的地方和被初始化的地方

private final Queue<Runnable> taskQueue;

taskQueue = newTaskQueue(this.maxPendingTasks);

@Overrideprotected Queue<Runnable> newTaskQueue(int maxPendingTasks) {

// This event loop never calls takeTask()

return PlatformDependent.newMpscQueue(maxPendingTasks);}

复制代码

我们发现 taskQueue 在NioEventLoop中默认是mpsc队列，mpsc队列，即多生产者单消费者队列，netty使用mpsc，方便的将外部线程的task聚集，在reactor线程内部用单线程来串行执行，我们可以借鉴netty的任务执行模式来处理类似多线程数据上报，定时聚合的应用

在本节讨论的任务场景中，所有代码的执行都是在reactor线程中的，所以，所有调用 inEventLoop() 的地方都返回true，既然都是在reactor线程中执行，那么其实这里的mpsc队列其实没有发挥真正的作用，mpsc大显身手的地方其实在第二种场景

### **二. 非当前reactor线程调用channel的各种方法**

// non reactor thread

channel.write(...)

上面一种情况在push系统中比较常见，一般在业务线程里面，根据用户的标识，找到对应的channel引用，然后调用write类方法向该用户推送消息，就会进入到这种场景

关于channel.write()类方法的调用链，后面会单独拉出一篇文章来深入剖析，这里，我们只需要知道，最终write方法串至以下方法

AbstractChannelHandlerContext.java

private void write(Object msg, boolean flush, ChannelPromise promise) {

// ...

EventExecutor executor = next.executor();

if (executor.inEventLoop()) {

if (flush) {

next.invokeWriteAndFlush(m, promise);

} else {

next.invokeWrite(m, promise);

}

} else {

AbstractWriteTask task;

if (flush) {

task = WriteAndFlushTask.newInstance(next, m, promise);

} else {

task = WriteTask.newInstance(next, m, promise);

}

safeExecute(executor, task, promise, m);

}}复制代码

外部线程在调用 write 的时候， executor.inEventLoop() 会返回false，直接进入到else分支，将write封装成一个 WriteTask （这里仅仅是write而没有flush，因此 flush 参数为false）, 然后调用 safeExecute 方法

private static void safeExecute(EventExecutor executor, Runnable runnable, ChannelPromise promise, Object msg) {

// ...

executor.execute(runnable);

// ...}复制代码

接下来的调用链就进入到第一种场景了，但是和第一种场景有个明显的区别就是，第一种场景的调用链的**发起线程**是reactor线程，第二种场景的调用链的发起线程是用户线程，用户线程可能会有很多个，显然多个线程并发写 taskQueue 可能出现线程同步问题，于是，这种场景下，netty的mpsc queue就有了用武之地

### **三. 用户自定义定时任务**

ctx.channel().eventLoop().schedule(new Runnable() {

@Override

public void run() {

}}, 60, TimeUnit.SECONDS);

第三种场景就是定时任务逻辑了，用的最多的便是如上方法：在一定时间之后执行任务

我们跟进 schedule 方法

public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit) {//...

return schedule(new ScheduledFutureTask<Void>(

this, command, null, ScheduledFutureTask.deadlineNanos(unit.toNanos(delay))));} 复制代码

通过 ScheduledFutureTask , 将用户自定义任务再次包装成一个netty内部的任务

<V> ScheduledFuture<V> schedule(final ScheduledFutureTask<V> task) {

// ...

scheduledTaskQueue().add(task);

// ...

return task;

}

到了这里，我们有点似曾相识，在非定时任务的处理中，netty通过一个mpsc队列将任务落地，这里，是否也有一个类似的队列来承载这类定时任务呢？带着这个疑问，我们继续向前

Queue<ScheduledFutureTask<?>> scheduledTaskQueue() {

if (scheduledTaskQueue == null) {

scheduledTaskQueue = new PriorityQueue<ScheduledFutureTask<?>>();

}

return scheduledTaskQueue;}复制代码

果不其然， scheduledTaskQueue() 方法，会返回一个优先级队列，然后调用 add 方法将定时任务加入到队列中去，但是，这里为什么要使用优先级队列，而不需要考虑多线程的并发？

因为我们现在讨论的场景，调用链的发起方是reactor线程，不会存在多线程并发这些问题

但是，万一有的用户在reactor之外执行定时任务呢？虽然这类场景很少见，但是netty作为一个无比健壮的高性能io框架，必须要考虑到这种情况。

对此，netty的处理是，如果是在外部线程调用schedule，netty将添加定时任务的逻辑封装成一个普通的task，这个task的任务是**添加定时任务的任务**，而不是添加定时任务，其实也就是第二种场景，这样，对 PriorityQueue的访问就变成单线程，即只有reactor线程

完整的schedule方法

<V> ScheduledFuture<V> schedule(final ScheduledFutureTask<V> task) {

if (inEventLoop()) {

scheduledTaskQueue().add(task);

} else {

// 进入到场景二，进一步封装任务

execute(new Runnable() {

@Override

public void run() {

scheduledTaskQueue().add(task);

}

});

}

return task;

}

复制代码

在阅读源码细节的过程中，我们应该多问几个为什么？这样会有利于看源码的时候不至于犯困！比如这里，为什么定时任务要保存在优先级队列中，我们可以先不看源码，来思考一下优先级对列的特性

优先级队列按一定的顺序来排列内部元素，内部元素必须是可以比较的，联系到这里每个元素都是定时任务，那就说明定时任务是可以比较的，那么到底有哪些地方可以比较？

每个任务都有一个下一次执行的截止时间，截止时间是可以比较的，截止时间相同的情况下，任务添加的顺序也是可以比较的，就像这样，阅读源码的过程中，一定要多和自己对话，多问几个为什么

带着猜想，我们研究了一下 ScheduledFutureTask ，抽取出关键部分

final class ScheduledFutureTask<V> extends PromiseTask<V> implements ScheduledFuture<V> {

private static final AtomicLong nextTaskId = new AtomicLong();

private static final long START\_TIME = System.nanoTime();

static long nanoTime() {

return System.nanoTime() - START\_TIME;

}

private final long id = nextTaskId.getAndIncrement();

/\* 0 - no repeat, >0 - repeat at fixed rate, <0 - repeat with fixed delay \*/

private final long periodNanos;

@Override

public int compareTo(Delayed o) {

//...

}

// 精简过的代码

@Override

public void run() {

}

这里，我们一眼就找到了 compareTo 方法， cmd+u 跳转到实现的接口，发现就是 Comparable 接口

public int compareTo(Delayed o) {

if (this == o) {

return 0;

}

ScheduledFutureTask<?> that = (ScheduledFutureTask<?>) o;

long d = deadlineNanos() - that.deadlineNanos();

if (d < 0) {

return -1;

} else if (d > 0) {

return 1;

} else if (id < that.id) {

return -1;

} else if (id == that.id) {

throw new Error();

} else {

return 1;

}}

进入到方法体内部，我们发现，两个定时任务的比较，确实是先比较任务的截止时间，截止时间相同的情况下，再比较id，即任务添加的顺序，如果id再相同的话，就抛Error

这样，在执行定时任务的时候，就能保证最近截止时间的任务先执行

下面，我们再来看下netty是如何来保证各种定时任务的执行的，netty里面的定时任务分以下三种

1.若干时间后执行一次 2.每隔一段时间执行一次 3.每次执行结束，隔一定时间再执行一次

netty使用一个 periodNanos 来区分这三种情况，正如netty的注释那样

/\* 0 - no repeat, >0 - repeat at fixed rate, <0 - repeat with fixed delay \*/

private final long periodNanos;

了解这些背景之后，我们来看下netty是如何来处理这三种不同类型的定时任务的

public void run() {

if (periodNanos == 0) {

V result = task.call();

setSuccessInternal(result);

} else {

task.call();

long p = periodNanos;

if (p > 0) {

deadlineNanos += p;

} else {

deadlineNanos = nanoTime() - p;

}

scheduledTaskQueue.add(this);

}

}}

if (periodNanos == 0) 对应 若干时间后执行一次 的定时任务类型，执行完了该任务就结束了。

否则，进入到else代码块，先执行任务，然后再区分是哪种类型的任务， periodNanos 大于0，表示是以固定频率执行某个任务，和任务的持续时间无关，然后，设置该任务的下一次截止时间为本次的截止时间加上间隔时间 periodNanos ，否则，就是每次任务执行完毕之后，间隔多长时间之后再次执行，截止时间为当前时间加上间隔时间， -p 就表示加上一个正的间隔时间，最后，将当前任务对象再次加入到队列，实现任务的定时执行

netty内部的任务添加机制了解地差不多之后，我们就可以查看reactor第三部曲是如何来调度这些任务的

## **reactor线程task的调度**

首先，我们将目光转向最外层的外观代码

runAllTasks(long timeoutNanos);

顾名思义，这行代码表示了尽量在一定的时间内，将所有的任务都取出来run一遍。 timeoutNanos 表示该方法最多执行这么长时间，netty为什么要这么做？我们可以想一想，reactor线程如果在此停留的时间过长，那么将积攒许多的IO事件无法处理(见reactor线程的前面两个步骤)，最终导致大量客户端请求阻塞，因此，默认情况下，netty将控制内部队列的执行时间

好，我们继续跟进

protected boolean runAllTasks(long timeoutNanos) {

fetchFromScheduledTaskQueue();

Runnable task = pollTask();

//...

final long deadline = ScheduledFutureTask.nanoTime() + timeoutNanos;

long runTasks = 0;

long lastExecutionTime;

for (;;) {

safeExecute(task);

runTasks ++;

if ((runTasks & 0x3F) == 0) {

lastExecutionTime = ScheduledFutureTask.nanoTime();

if (lastExecutionTime >= deadline) {

break;

}

}

task = pollTask();

if (task == null) {

lastExecutionTime = ScheduledFutureTask.nanoTime();

break;

}

}

afterRunningAllTasks();

this.lastExecutionTime = lastExecutionTime;

return true;}

这段代码便是reactor执行task的所有逻辑，可以拆解成下面几个步骤

1. 从scheduledTaskQueue转移定时任务到taskQueue(mpsc queue)
2. 计算本次任务循环的截止时间
3. 执行任务
4. 收尾

按照这个步骤，我们一步步来分析下

### **从scheduledTaskQueue转移定时任务到taskQueue(mpsc queue)**

首先调用 fetchFromScheduledTaskQueue() 方法，将到期的定时任务转移到mpsc queue里面

private boolean fetchFromScheduledTaskQueue() {

long nanoTime = AbstractScheduledEventExecutor.nanoTime();

Runnable scheduledTask = pollScheduledTask(nanoTime);

while (scheduledTask != null) {

if (!taskQueue.offer(scheduledTask)) {

// No space left in the task queue add it back to the scheduledTaskQueue so we pick it up again.

scheduledTaskQueue().add((ScheduledFutureTask<?>) scheduledTask);

return false;

}

scheduledTask = pollScheduledTask(nanoTime);

}

return true;}

可以看到，netty在把任务从scheduledTaskQueue转移到taskQueue的时候还是非常小心的，当taskQueue无法offer的时候，需要把从scheduledTaskQueue里面取出来的任务重新添加回去

从scheduledTaskQueue中拉取一个定时任务的逻辑如下，传入的参数 nanoTime 为当前时间(其实是当前纳秒减去 ScheduledFutureTask 类被加载的纳秒个数)

protected final Runnable pollScheduledTask(long nanoTime) {

assert inEventLoop();

Queue<ScheduledFutureTask<?>> scheduledTaskQueue = this.scheduledTaskQueue;

ScheduledFutureTask<?> scheduledTask = scheduledTaskQueue == null ? null : scheduledTaskQueue.peek();

if (scheduledTask == null) {

return null;

}

if (scheduledTask.deadlineNanos() <= nanoTime) {

scheduledTaskQueue.remove();

return scheduledTask;

}

return null;}

可以看到，每次 pollScheduledTask 的时候，只有在当前任务的截止时间已经到了，才会取出来

### **计算本次任务循环的截止时间**

Runnable task = pollTask();

//...

final long deadline = ScheduledFutureTask.nanoTime() + timeoutNanos;

long runTasks = 0;

long lastExecutionTime;

这一步将取出第一个任务，用reactor线程传入的超时时间 timeoutNanos 来计算出当前任务循环的deadline，并且使用了 runTasks ， lastExecutionTime 来时刻记录任务的状态

### **循环执行任务**

for (;;) {

safeExecute(task);

runTasks ++;

**//0x3F即(2^6 -1),二进制表示为111 111**

if ((runTasks & 0x3F) == 0) {

lastExecutionTime = ScheduledFutureTask.nanoTime();

if (lastExecutionTime >= deadline) {

break;

}

}

task = pollTask();

if (task == null) {

lastExecutionTime = ScheduledFutureTask.nanoTime();

break;

}}复制代码

这一步便是netty里面执行所有任务的核心代码了。 首先调用 safeExecute 来确保任务安全执行，忽略任何异常

protected static void safeExecute(Runnable task) {

try {

task.run();

} catch (Throwable t) {

logger.warn("A task raised an exception. Task: {}", task, t);

}}

然后将已运行任务 runTasks 加一，每隔 0x3F 任务，即每执行完64个任务之后，判断当前时间是否超过本次reactor任务循环的截止时间了，如果超过，那就break掉，如果没有超过，那就继续执行。可以看到，netty对性能的优化考虑地相当的周到，假设netty任务队列里面如果有海量小任务，如果每次都要执行完任务都要判断一下是否到截止时间，那么效率是比较低下的

### **收尾**

afterRunningAllTasks();

this.lastExecutionTime = lastExecutionTime;

收尾工作很简单，调用一下 afterRunningAllTasks 方法

protected void afterRunningAllTasks() {

runAllTasksFrom(tailTasks);}

NioEventLoop 可以通过父类 SingleTheadEventLoop 的 executeAfterEventLoopIteration 方法向 tailTasks 中添加收尾任务，比如，你想统计一下一次执行一次任务循环花了多长时间就可以调用此方法

public final void executeAfterEventLoopIteration(Runnable task) {

// ...

if (!tailTasks.offer(task)) {

reject(task);

}

//...}

this.lastExecutionTime = lastExecutionTime; 简单记录一下任务执行的时间，搜了一下该field的引用，发现这个field并没有使用过，只是每次不停地赋值，赋值，赋值...，改天再去向netty官方提个issue...

reactor线程第三曲到了这里基本上就给你讲完了，如果你读到这觉得很轻松，那么恭喜你，你对netty的task机制已经非常比较熟悉了，也恭喜一下我，把这些机制给你将清楚了。我们最后再来一次总结，以tips的方式

* 当前reactor线程调用当前eventLoop执行任务，直接执行，否则，添加到任务队列稍后执行
* netty内部的任务分为普通任务和定时任务，分别落地到MpscQueue和PriorityQueue
* netty每次执行任务循环之前，会将已经到期的定时任务从PriorityQueue转移到MpscQueue
* netty每隔64个任务检查一下是否该退出任务循环

如果你想系统地学Netty，我的小册 [《Netty 入门与实战：仿写微信 IM 即时通讯系统》](https://link.juejin.im/?target=https://juejin.im/book/5b4bc28bf265da0f60130116" \t "https://www.codercto.com/a/_blank) 可以帮助你，如果你想系统学习Netty原理，那么你一定不要错过我的Netty源码分析系列视频： [coding.imooc.com/class/230.h…](https://link.juejin.im/?target=https://coding.imooc.com/class/230.html" \t "https://www.codercto.com/a/_blank)

以上就是本文的全部内容，希望本文的内容对大家的学习或者工作能带来一定的帮助，也希望大家多多支持 [码农网](https://www.codercto.com/)

# NETTY源码分析之Promise

# NETTY源码分析之ByteBuf

# [JCTools简介](https://segmentfault.com/a/1190000011009097)

## 序

JCTools是一款对jdk并发数据结构进行增强的并发工具，主要提供了map以及queue的增强数据结构。原来netty还是自己写的MpscLinkedQueueNode，后来新版本就换成使用JCTools的并发队列了。

## 增强map

* ConcurrentAutoTable(后面几个map/set结构的基础)
* NonBlockingHashMap
* NonBlockingHashMapLong
* NonBlockingHashSet
* NonBlockingIdentityHashMap
* NonBlockingSetInt

## 增强队列

* SPSC - Single Producer Single Consumer (Wait Free, bounded and unbounded)
* MPSC - Multi Producer Single Consumer (Lock less, bounded and unbounded)
* SPMC - Single Producer Multi Consumer (Lock less, bounded)
* MPMC - Multi Producer Multi Consumer (Lock less, bounded)

## maven

<dependency>

<groupId>org.jctools</groupId>

<artifactId>jctools-core</artifactId>

<version>2.1.0</version>

</dependency>

## ConcurrentAutoTable

替代AtomicLong，专门为高性能的counter设计的。只有几个方法

**public** **void** **add**( **long** x );

**public** **void** **decrement**();

**public** **void** **increment**();

**public** **void** **set**( **long** x );

**public** **long** **get**();

**public** **int** **intValue**();

**public** **long** **longValue**();

**public** **long** **estimate\_get**();

对比AtomicLong主要是操作之后没有立即返回

**public** **final** **long** **incrementAndGet**();

**public** **final** **long** **decrementAndGet**()

## NonBlockingHashMap

NonBlockingHashMap是对ConcurrentHashMap的增强，对多CPU的支持以及高并发更新提供更好的性能。  
NonBlockingHashMapLong是key为Long型的NonBlockingHashMap。  
NonBlockingHashSet是对NonBlockingHashMap的简单包装以支持set的接口。  
NonBlockingIdentityHashMap是从NonBlockingHashMap改造来的，使用System.identityHashCode()来计算哈希  
NonBlockingSetInt是一个使用CAS的简单的bit-vector

原来是

// --- hash ----------------------------------------------------------------

// Helper function to spread lousy hashCodes. Throws NPE for null Key, on

// purpose - as the first place to conveniently toss the required NPE for a

// null Key.

**private** **static** **final** **int** **hash**(**final** Object key) {

**int** h = key.hashCode(); // The real hashCode call

h ^= (h>>>20) ^ (h>>>12);

h ^= (h>>> 7) ^ (h>>> 4);

h += h<<7; // smear low bits up high, for hashcodes that only differ by 1

**return** h;

}

改为

// --- hash ----------------------------------------------------------------

// Helper function to spread lousy hashCodes

**private** **static** **final** **int** **hash**(**final** Object key) {

**int** h = System.identityHashCode(key); // The real hashCode call

// I assume that System.identityHashCode is well implemented with a good

// spreader, and a second bit-spreader is redundant.

//h ^= (h>>>20) ^ (h>>>12);

//h ^= (h>>> 7) ^ (h>>> 4);

**return** h;

}

## doc

* [JCTools](https://github.com/JCTools/JCTools" \t "https://segmentfault.com/a/_blank)
* [Java Concurrent Counters By Numbers](http://psy-lob-saw.blogspot.com/2013/06/java-concurrent-counters-by-numbers.html" \t "https://segmentfault.com/a/_blank)
* [Lock Free Queues Index](http://psy-lob-saw.blogspot.com/p/lock-free-queues.html" \t "https://segmentfault.com/a/_blank)
* [135 Million messages a second between processes in pure Java](http://psy-lob-saw.blogspot.com/2013/04/lock-free-ipc-queue.html" \t "https://segmentfault.com/a/_blank)

# 参考：

<https://www.cnblogs.com/zhangboyu/p/7452607.html>

<https://www.cnblogs.com/zhangboyu/p/7452611.html>

<https://www.codercto.com/a/34435.html>

Netty源码解析——基于NIO网络传输模式的OP\_ACCEPT,OP\_CONNECT,OP\_READ,OP\_WRITE

事件处理流程

https://cloud.tencent.com/developer/article/1152647