# 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) {

**//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;  
 }

**//DefaultChannelPromise注册eventLoop完成,可能在eventLoop中完成** **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** {**//DefaultChannelPromise注册eventLoop未完成** ***//注册future几乎总是已实现的.但是以防止它没有实现*****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**) {  
 **//在EventLoop上注册失败，因此ChannelPromise在我们**

**//尝试访问通道的EventLoop时不会直接导**

**//致IllegalStateException。**

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;  
 }  
}

### initAndRegister

**final** ChannelFuture initAndRegister() {  
 Channel channel = **null**;  
 **try** {  
 channel = **channelFactory**.newChannel();  
 **init(channel);** } **catch** (Throwable t) {  
 **if** (channel != **null**) {  
 *// channel can be null if newChannel crashed (eg SocketException("too many open files"))* channel.unsafe().closeForcibly();  
 *// as the Channel is not registered yet we need to force the usage of the GlobalEventExecutor* **return new** DefaultChannelPromise(channel, GlobalEventExecutor.***INSTANCE***).setFailure(t);  
 }  
 *// as the Channel is not registered yet we need to force the usage of the 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();  
 }  
 }

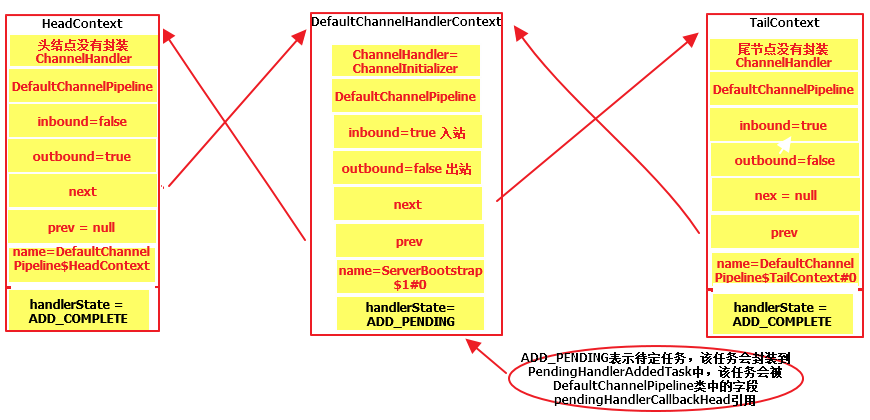
#### init(channel)

**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());  
 }  
 }  
  
 ChannelPipeline p = channel.pipeline();  
  
 **final** EventLoopGroup currentChildGroup = **childGroup**;  
 **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)

@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，该线程执行EventLoop的run方法，该run方法里面是个死**

**//循环。** 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);  
 }  
 }  
}

@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*();  
 }  
 }  
 //判断该任务是不是不唤醒任务，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**;

**//为eventLoop分配一个线程执行eventLoop的run方法** **executor**.execute(**new** Runnable() {  
 @Override  
 **public void** run() {

**//为eventLoop分配线程** **thread** = Thread.*currentThread*();  
 **if** (**interrupted**) {//  
 **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**);  
 }  
 }  
 }  
 }  
 });  
}

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

**private static void** doBind0(  
 **final** ChannelFuture regFuture, **final** Channel channel,  
 **final** SocketAddress localAddress, **final** ChannelPromise promise) {  
  
 *// This method is invoked before channelRegistered() is triggered. Give user handlers a chance to set up  
 // the pipeline in its channelRegistered() implementation.*

***//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channelregister()实现中设置管道。**

**//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());  
 }  
 }  
 });  
}

//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*();  
 }  
 }  
  
 **if** (!**addTaskWakesUp** && wakesUpForTask(task)) {  
 wakeup(inEventLoop);  
 }  
}

## 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();  
 }  
 });  
}

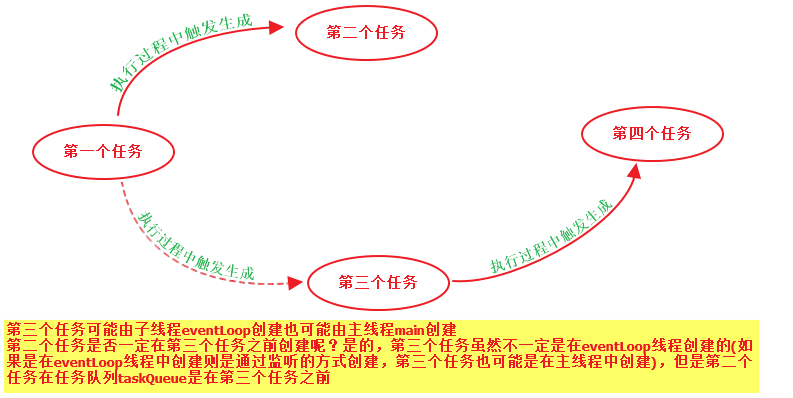
**private void** notifyListenersNow() {  
 Object listeners;  
 **synchronized** (**this**) {  
 *// Only proceed if there are listeners to notify and we are not already notifying listeners.* **if** (**notifyingListeners** || **this**.**listeners** == **null**) {  
 **return**;  
 }  
 **notifyingListeners** = **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**;  
 **registered** = **true**;  
  
 *// Ensure we call handlerAdded(...) before we actually notify the promise. This is needed as the  
 // user may already fire events through the pipeline in the ChannelFutureListener.*

***//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**

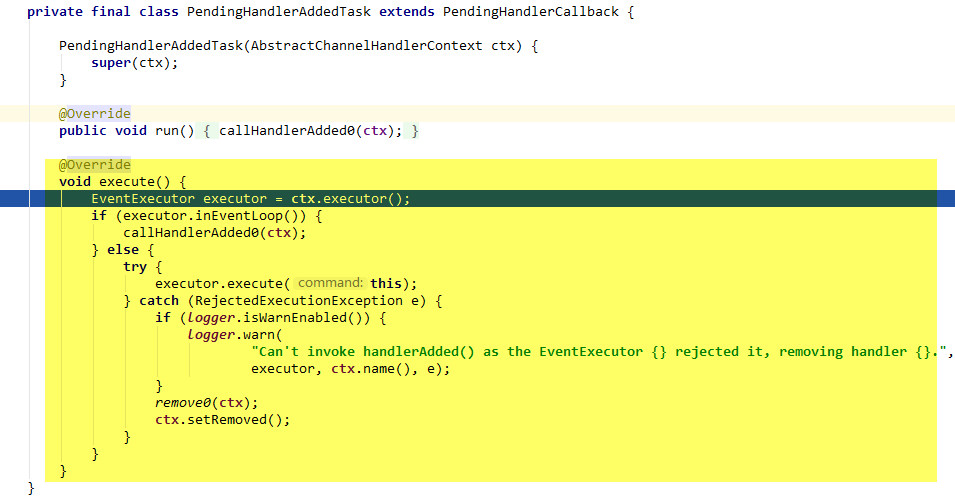
***//回调方法，这些回调是在register完成之前添加的*****callHandlerAddedForAllHandlers**();  
 }  
}

**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

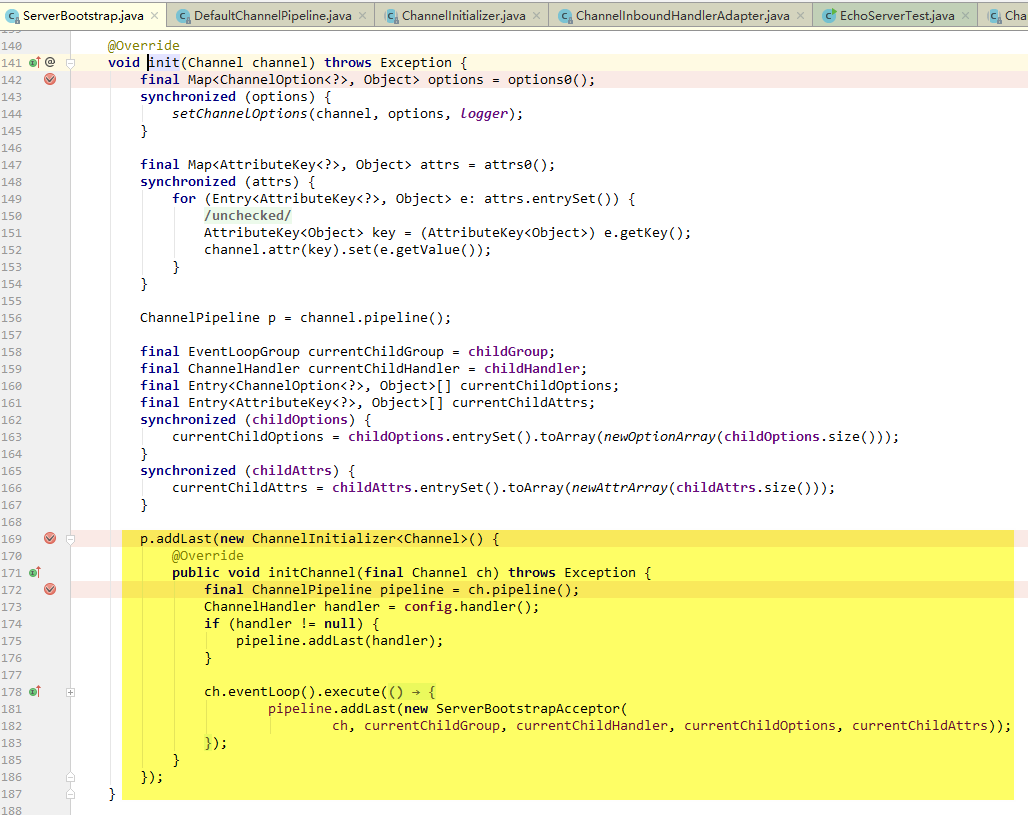
//2.ChannelHandler.handlerAdded()  
 ctx.handler().handlerAdded(ctx);  
 } **catch** (Throwable t) {  
 **boolean** removed = **false**;  
 **try** {  
 *remove0*(ctx);  
 **try** {  
 ctx.handler().handlerRemoved(ctx);  
 } **finally** {  
 ctx.setRemoved();  
 }  
 removed = **true**;  
 } **catch** (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);  
 }  
}

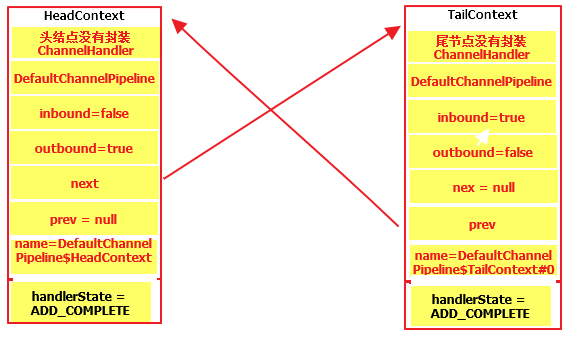
@SuppressWarnings(**"unchecked"**)  
**private boolean** initChannel(ChannelHandlerContext ctx) **throws** Exception {  
 **if** (**initMap**.putIfAbsent(ctx, Boolean.***TRUE***) == **null**) { *// Guard against re-entrance.* **try** {

**//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);  
}

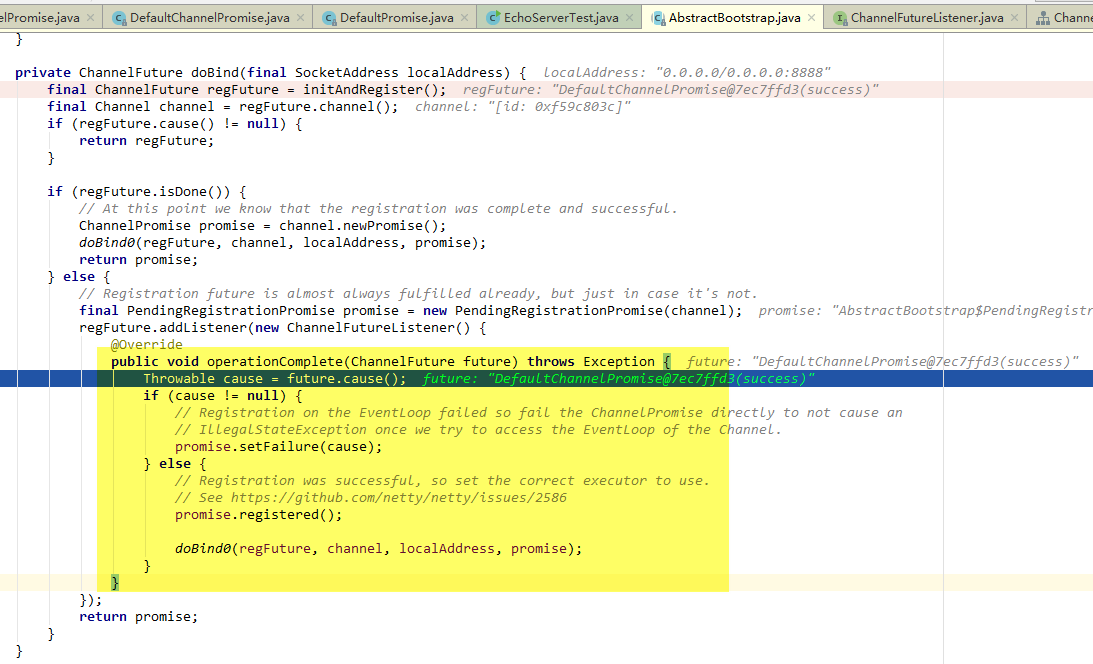
**private boolean** setValue0(Object objResult) {  
 **if** (***RESULT\_UPDATER***.compareAndSet(**this**, **null**, objResult) ||  
 ***RESULT\_UPDATER***.compareAndSet(**this**, ***UNCANCELLABLE***, objResult)) {  
 checkNotifyWaiters();  
 **return true**;  
 }  
 **return false**;  
}

**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**

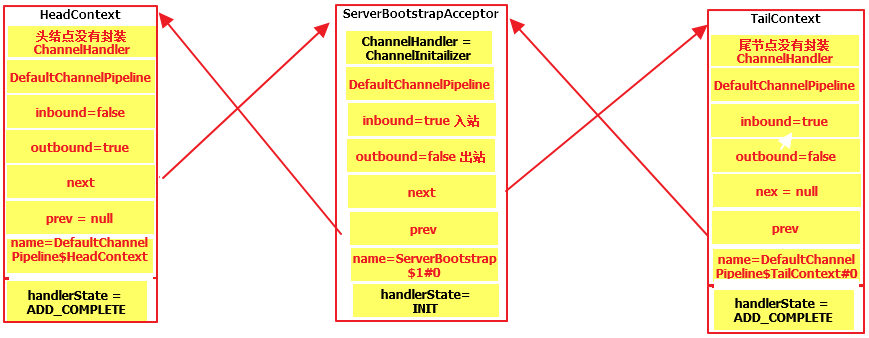
**//例多次添加到一个或多个ChannelPipeline中，而不存在竞争条件。**  
 **if** (!h.isSharable() && h.**added**) {  
 **throw new** ChannelPipelineException(  
 h.getClass().getName() +  
 **" is not a @Sharable handler, so can't be added or removed multiple times."**);  
 }  
 h.**added** = **true**;  
 }  
}

#### 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

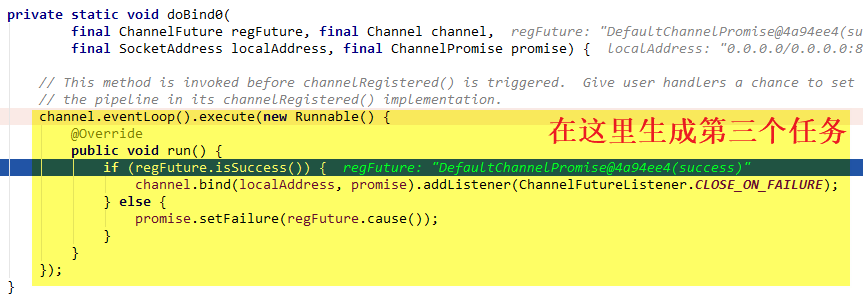
**private void** callHandlerAdded0(**final** AbstractChannelHandlerContext ctx) {  
 **try** {  
 *// We must call setAddComplete before calling handlerAdded. Otherwise if the handlerAdded method generates  
 // any pipeline events ctx.handler() will miss them because the state will not allow it.* **ctx.setAddComplete();**

**//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**;  
}

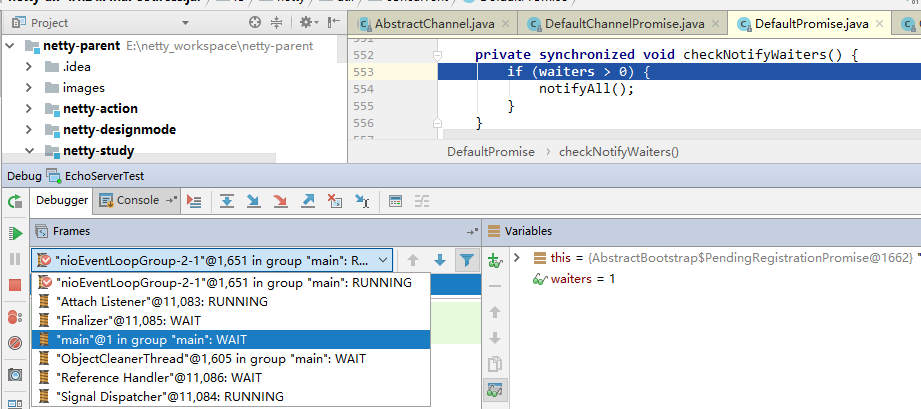
**private synchronized void** checkNotifyWaiters() {

**//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

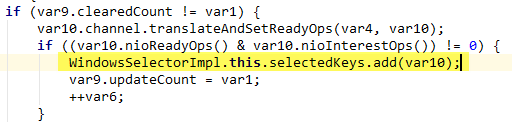
**final class** SelectedSelectionKeySetSelector **extends** Selector {  
 **private final** SelectedSelectionKeySet **selectionKeys**;  
 **private final** Selector **delegate**;  
  
 SelectedSelectionKeySetSelector(Selector delegate, SelectedSelectionKeySet selectionKeys) {  
 **this**.**delegate** = 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;  
 }  
}

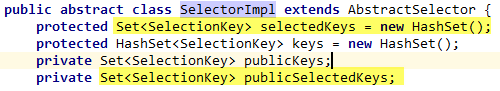
**selectionKeys如何赋值的？**



**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) {

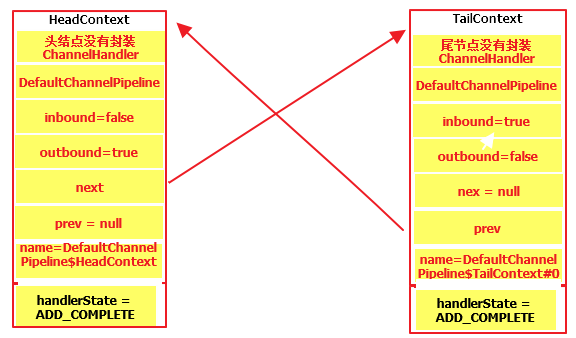
**//msg就是NioSocketChannel**  
 **final** Channel child = (Channel) msg;

**//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);  
 }  
}

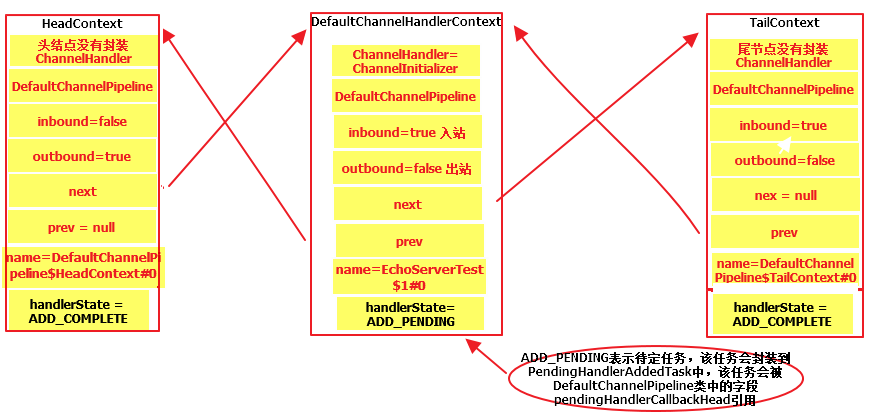


在进行此步骤之前，与NioSocketChannel关联pipeline的链表结构



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

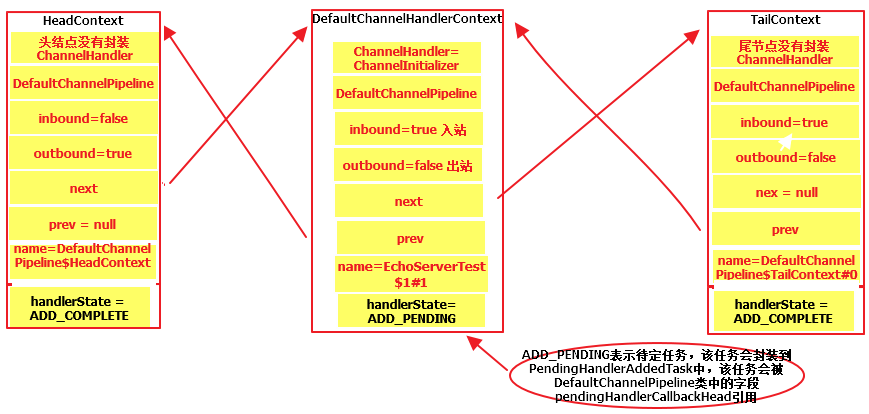
在进行此步骤之后，与NioSocketChannel关联的pipeline的链表结构如下：



**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.*

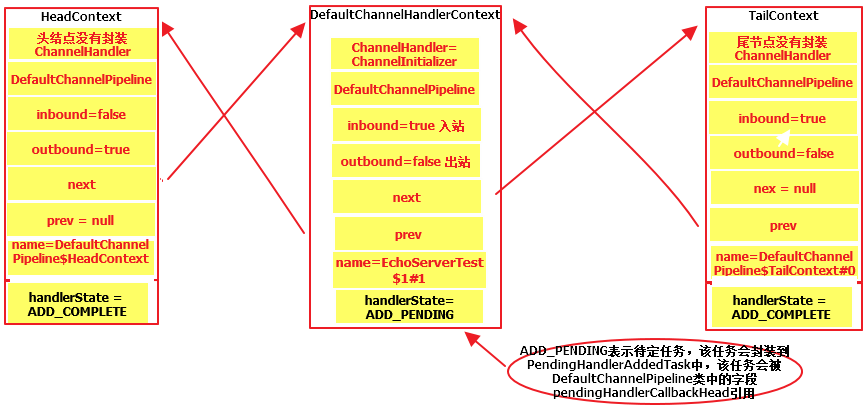
***//listeners是一个匿名类如下图*****if** (**notifyingListeners** || **this**.**listeners** == **null**) {  
 **return**;  
 }  
 **notifyingListeners** = **true**;  
 listeners = **this**.**listeners**;  
 **this**.**listeners** = **null**;  
 }  
 **for** (;;) {  
 **if** (listeners **instanceof** DefaultFutureListeners) {  
 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***:  
 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();  
 }  
 }  
 }  
}

### 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**;  
}

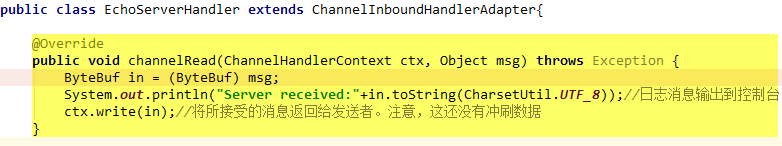
**static void** invokeChannelRead(**final** AbstractChannelHandlerContext next, Object msg) {  
 **final** Object m = next.**pipeline**.touch(ObjectUtil.*checkNotNull*(msg, **"msg"**), next);  
 EventExecutor executor = next.executor();  
 **if** (executor.inEventLoop()) {  
 next.invokeChannelRead(m);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

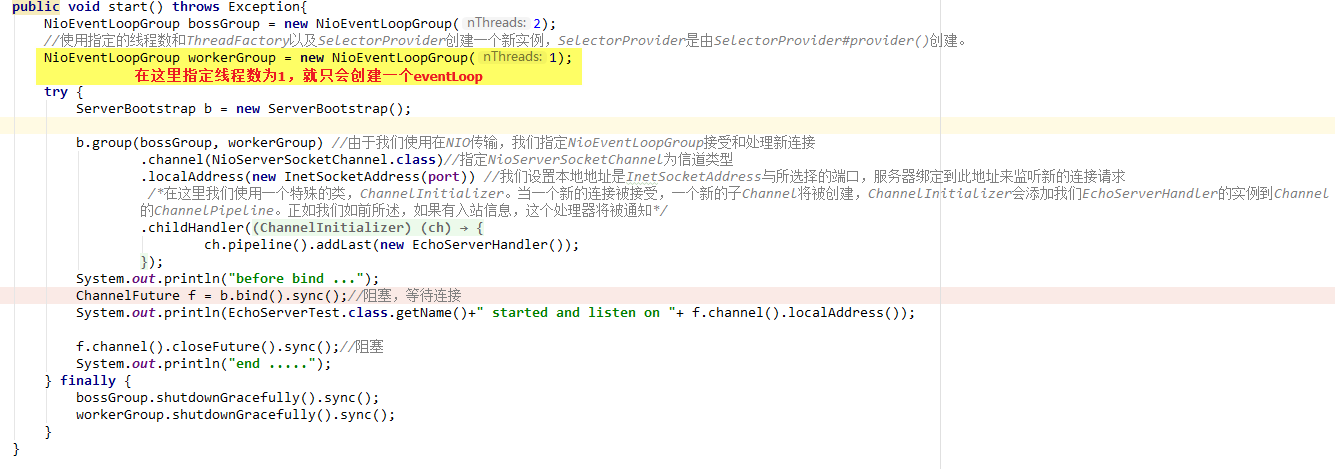
**protected final void** clearOpWrite() {  
 **final** SelectionKey key = selectionKey();  
 *// Check first if the key is still valid as it may be canceled as part of the deregistration  
 // from the EventLoop  
 // See https://github.com/netty/netty/issues/2104* **if** (!key.isValid()) {  
 **return**;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

# 同一个NioEventLoop能不能接受多个客户端连接

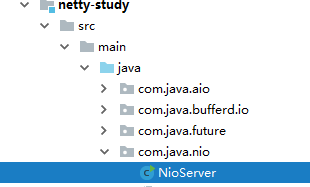
可以，一个NioEventLoop对应一个NioSocketChannel,一个NioSocketChannel对应一个SocketChannel,而所有的SocketChannel都注册到了Selector，注意整个服务端应用只有一个selector，这个selector监听所有通道事件。

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

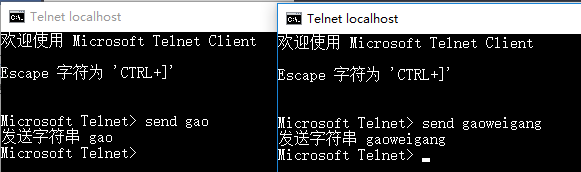
**只是说配置多线程性能会更好而已**

## 观察Java NIO

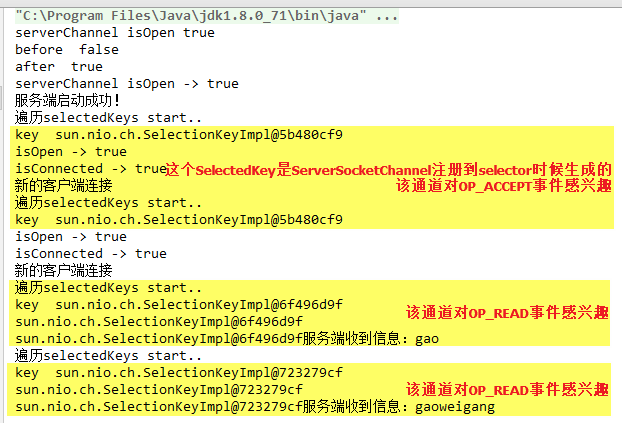
1. 启动服务器动程序



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

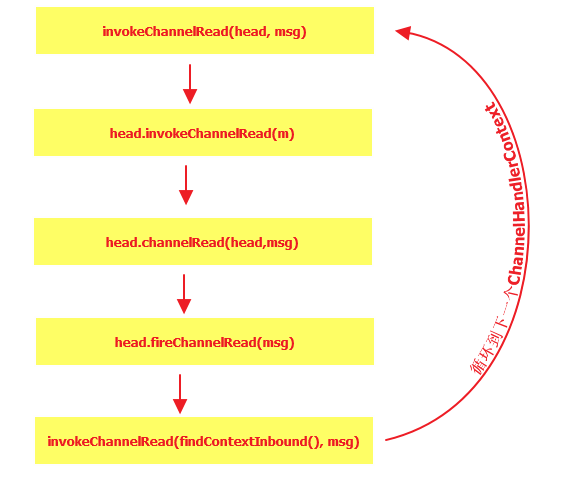


4.看看服务器端的输出



# ChannelHandlerContext详解

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



# 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：

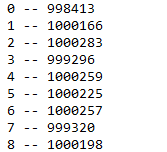
Returns the current value of the running Java Virtual Machine's high-resolution time source, in nanoseconds.  
This method can only be used to measure elapsed time and is not related to any other notion of system or wall-clock time. The value returned represents nanoseconds since some fixed but arbitrary 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

# OP\_WRITE注册