# 准备

### WPS里面如何去掉红色波浪线

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

# 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

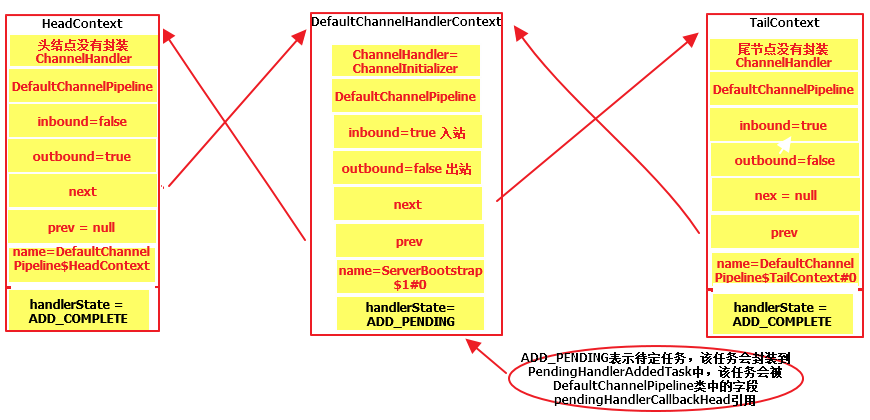
**final** ChannelFuture initAndRegister() {  
 Channel channel = **null**;  
 **try** {  
 channel = **channelFactory**.newChannel();  
 **init(channel);** } **catch** (Throwable t) {  
 **if** (channel != **null**) {  
 *// channel can be null if newChannel crashed (eg SocketException("too many open 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) @Override **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();  
 }  
 });  
}

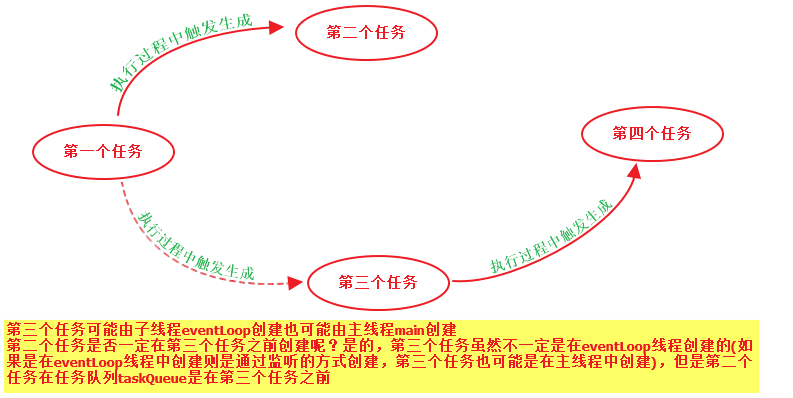
**private void** notifyListenersNow() {  
 Object listeners;  
 **synchronized** (**this**) {  
 *// Only proceed if there are listeners to notify and we are not already notifying listeners.* **if** (**notifyingListeners** || **this**.**listeners** == **null**) {  
 **return**;  
 }  
 **notifyingListeners**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

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NioEventLoopp

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.* **if** (**isActive()**) {  
 **if** (firstRegistration) {  
 **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**

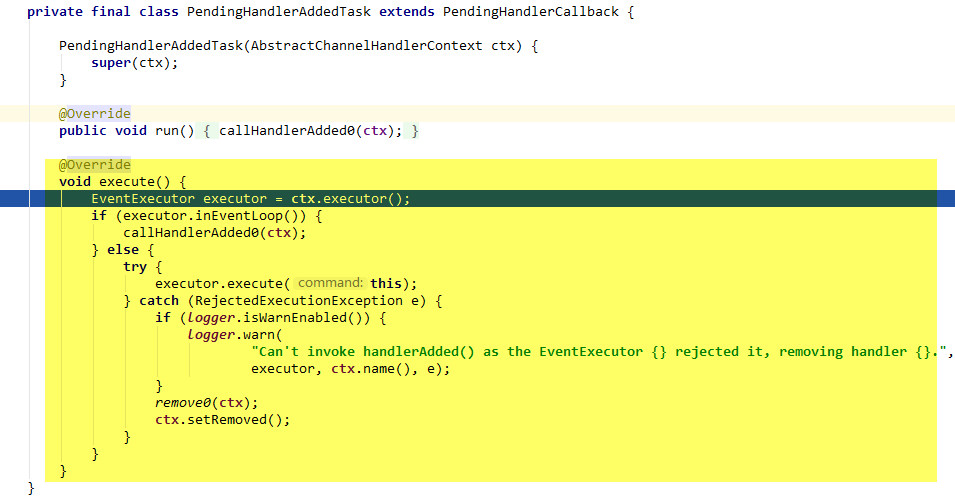
***//回调方法，这些回调是在register完成之前添加的*****callHandlerAddedForAllHandlers**();  
 }  
}

**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

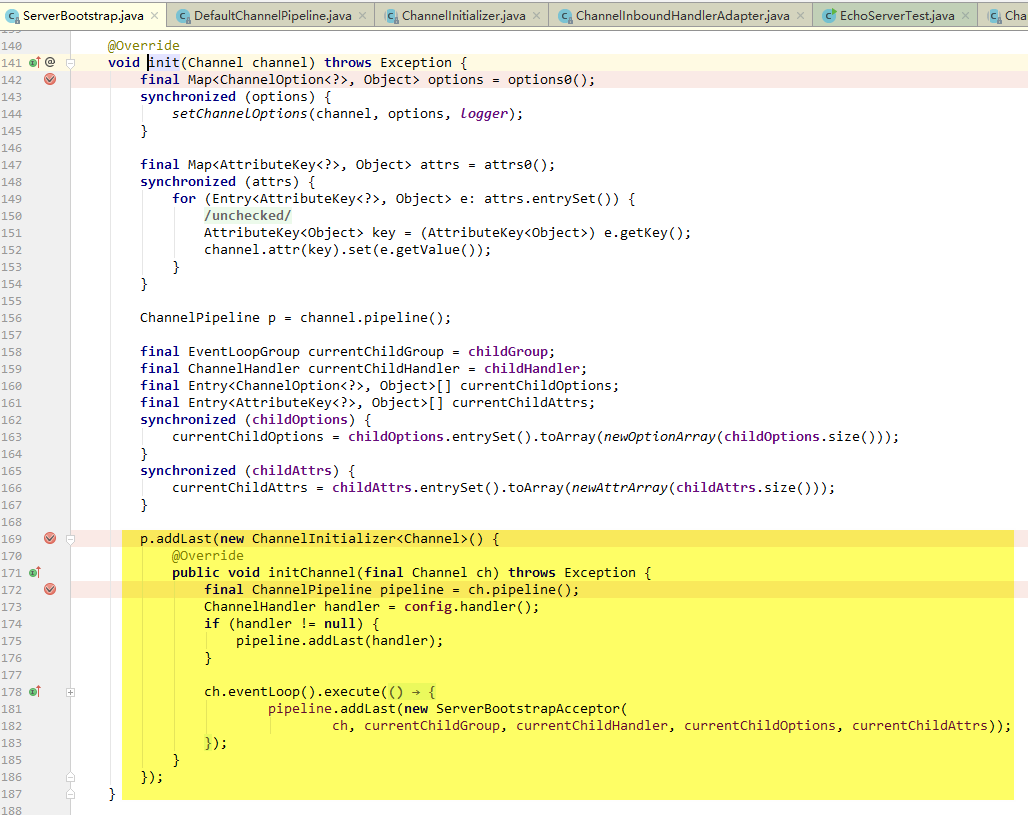
//2.ChannelHandler.handlerAdded()  
 ctx.handler().handlerAdded(ctx);  
 } **catch** (Throwable t) {  
 **boolean** removed = **false**;  
 **try** {  
 *remove0*(ctx);  
 **try** {  
 ctx.handler().handlerRemoved(ctx);  
 } **finally** {  
 ctx.setRemoved();  
 }  
 removed = **true**;  
 } **catch**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

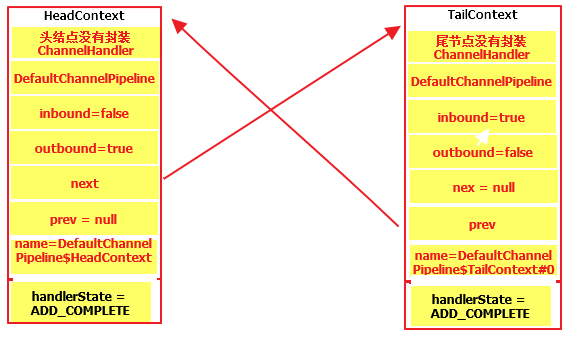
@SuppressWarnings(**"unchecked"**)  
**private boolean** initChannel(ChannelHandlerContext ctx) **throws** Exception {  
 **if** (**initMap**.putIfAbsent(ctx, Boolean.***TRUE***) == **null**) { *// Guard against re-entrance.* **try** {

**//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);  
}

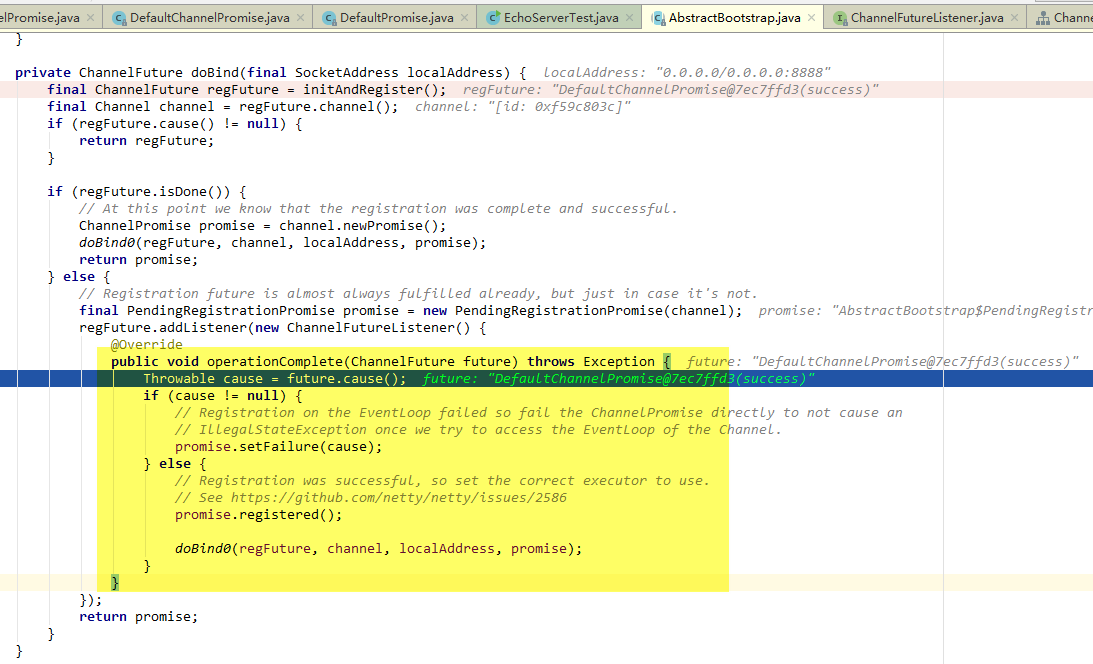
**private boolean** setValue0(Object objResult) {  
 **if** (***RESULT\_UPDATER***.compareAndSet(**this**, **null**, objResult) ||  
 ***RESULT\_UPDATER***.compareAndSet(**this**, ***UNCANCELLABLE***, objResult)) {  
 checkNotifyWaiters();  
 **return true**;  
 }  
 **return false**;  
}

**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**

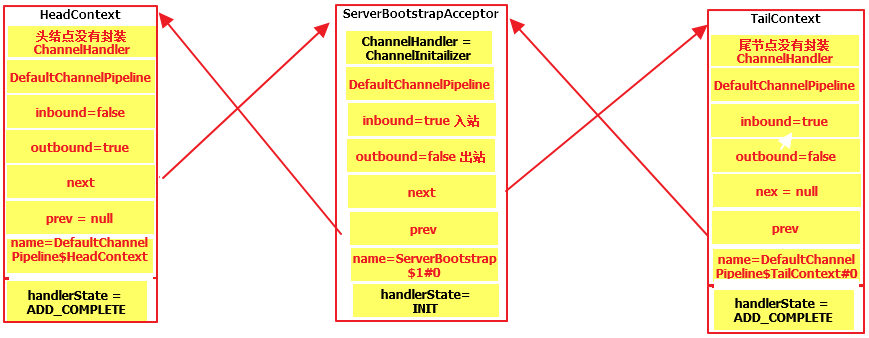
**//例多次添加到一个或多个ChannelPipeline中，而不存在竞争条件。**  
 **if** (!h.isSharable() && h.**added**) {  
 **throw new** ChannelPipelineException(  
 h.getClass().getName() +  
 **" is not a @Sharable handler, so can't be added or removed multiple times."**);  
 }  
 h.**added** = **true**;  
 }  
}

#### 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

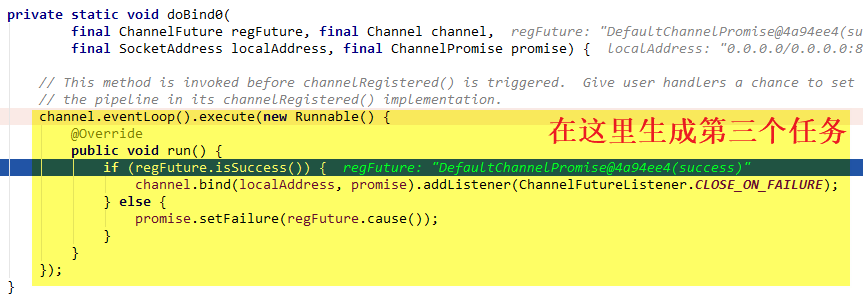
**private void** callHandlerAdded0(**final** AbstractChannelHandlerContext ctx) {  
 **try** {  
 *// We must call setAddComplete before calling handlerAdded. Otherwise if the handlerAdded method generates  
 // any pipeline events ctx.handler() will miss them because the state will not allow it.* **ctx.setAddComplete();**

**//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**;  
}

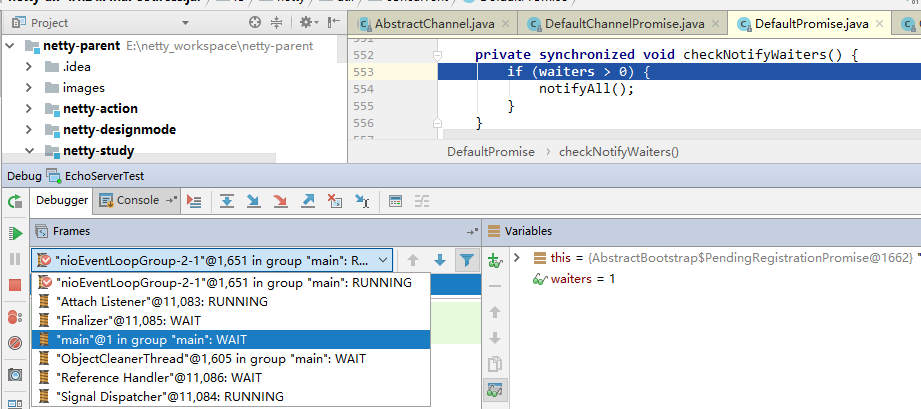
**private synchronized void** checkNotifyWaiters() {

**//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

## 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**);  
 }  
}

# 接受客户端的连接OP\_ACCEPT

## 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** {  
 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);  
 }  
 }  
}

### 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;

//获取系统当前的纳秒值  
 **long** currentTimeNanos = System.*nanoTime*();  
 **long** selectDeadLineNanos = currentTimeNanos + delayNanos(currentTimeNanos);  
 **for** (;;) {  
 **long** timeoutMillis = (selectDeadLineNanos - currentTimeNanos + 500000L) / 1000000L;  
 **if** (timeoutMillis <= 0) {  
 **if** (selectCnt == 0) {  
 selector.selectNow();  
 selectCnt = 1;  
 }  
 **break**;  
 }  
  
 *// If a task was submitted when wakenUp value was true, the task didn't get a chance to call  
 // Selector#wakeup. So we need to check task queue again before executing select operation.  
 // If we don't, the task might be pended until select operation was timed out.  
 // It might be pended until idle timeout if IdleStateHandler existed in pipeline.* **if** (hasTasks() && **wakenUp**.compareAndSet(**false**, **true**)) {  
 selector.selectNow();  
 selectCnt = 1;  
 **break**;  
 }  
  
 **int** selectedKeys = selector.select(timeoutMillis);  
 selectCnt ++;  
  
 **if** (selectedKeys != 0 || oldWakenUp || **wakenUp**.get() || hasTasks() || hasScheduledTasks()) {  
 *// - Selected something,  
 // - waken up by user, or  
 // - the task queue has a pending task.  
 // - a scheduled task is ready for processing* **break**;  
 }  
 **if** (Thread.*interrupted*()) {  
 *// Thread was interrupted so reset selected keys and break so we not run into a busy loop.  
 // As this is most likely a bug in the handler of the user or it's client library we will  
 // also log it.  
 //  
 // See https://github.com/netty/netty/issues/2426* **if** (***logger***.isDebugEnabled()) {  
 ***logger***.debug(**"Selector.select() returned prematurely because "** +  
 **"Thread.currentThread().interrupt() was called. Use "** +  
 **"NioEventLoop.shutdownGracefully() to shutdown the NioEventLoop."**);  
 }  
 selectCnt = 1;  
 **break**;  
 }  
  
 **long** time = System.*nanoTime*();  
 **if** (time - TimeUnit.***MILLISECONDS***.toNanos(timeoutMillis) >= currentTimeNanos) {  
 *// timeoutMillis elapsed without anything selected.* selectCnt = 1;  
 } **else if** (***SELECTOR\_AUTO\_REBUILD\_THRESHOLD*** > 0 &&  
 selectCnt >= ***SELECTOR\_AUTO\_REBUILD\_THRESHOLD***) {  
 *// The selector returned prematurely many times in a row.  
 // Rebuild the selector to work around the problem.* ***logger***.warn(  
 **"Selector.select() returned prematurely {} times in a row; rebuilding Selector {}."**,  
 selectCnt, selector);  
  
 rebuildSelector();  
 selector = **this**.**selector**;  
  
 *// Select again to populate selectedKeys.* selector.selectNow();  
 selectCnt = 1;  
 **break**;  
 }  
  
 currentTimeNanos = time;  
 }  
  
 **if** (selectCnt > ***MIN\_PREMATURE\_SELECTOR\_RETURNS***) {  
 **if** (***logger***.isDebugEnabled()) {  
 ***logger***.debug(**"Selector.select() returned prematurely {} times in a row for Selector {}."**,  
 selectCnt - 1, selector);  
 }  
 }  
 } **catch** (CancelledKeyException e) {  
 **if** (***logger***.isDebugEnabled()) {  
 ***logger***.debug(CancelledKeyException.**class**.getSimpleName() + **" raised by a Selector {} - JDK bug?"**,  
 selector, e);  
 }  
 *// Harmless exception - log anyway* }  
}

#### 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();  
 }  
}

#### 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);  
}

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