1. NioEventLoop.run()
2. NioEventLoop.processSelectedKeys()
3. NioEventLoop.processSelectedKeysOptimized(SelectionKey[] selectedKeys)
4. NioEventLoop.processSelectedKey(SelectionKey k, AbstractNioChannel ch)
5. NioMessageUnsafe.read()
6. NioServerSocketChannel.doReadMessages(List<Object> buf)
7. buf.add(**new** NioSocketChannel(**this**, ch));
8. NioSocketChannel(Channel parent, SocketChannel socket)
9. AbstractNioByteChannel (Channel parent, SelectableChannel ch)
10. **super**(parent, ch, SelectionKey.*OP\_READ*)
11. AbstractNioChannel(Channel parent, SelectableChannel ch, **int** readInterestOp)
12. AbstractChannel(Channel parent)
13. unsafe = newUnsafe();pipeline = **new** DefaultChannelPipeline(**this**);
14. AbstractNioMessageChannel.AbstractNioUnsafe.newUnsafe()
15. **new** NioMessageUnsafe();
16. DefaultChannelPipeline.fireChannelRead(Object msg)
17. AbstractChannelHandlerContext.fireChannelRead(**final** Object msg)
18. AbstractChannelHandlerContext.invokeChannelRead(Object msg)
19. SimpleChannelInboundHandler.channelRead(ChannelHandlerContext ctx, Object msg)
20. XXHandler.channelRead0(ChannelHandlerContext ctx, I msg)（具体实现类）

/\*\*

\* Operation-set bit for read operations.

\*

\* <p> Suppose that a selection key's interest set contains

\* <tt>OP\_READ</tt> at the start of a <a

\* href="Selector.html#selop">selection operation</a>. If the selector

\* detects that the corresponding channel is ready for reading, has reached

\* end-of-stream, has been remotely shut down for further reading, or has

\* an error pending, then it will add <tt>OP\_READ</tt> to the key's

\* ready-operation set and add the key to its selected-key&nbsp;set. </p>

\*/

**public** **static** **final** **int** *OP\_READ* = 1 << 0;

/\*\*

\* Operation-set bit for write operations. </p>

\*

\* <p> Suppose that a selection key's interest set contains

\* <tt>OP\_WRITE</tt> at the start of a <a

\* href="Selector.html#selop">selection operation</a>. If the selector

\* detects that the corresponding channel is ready for writing, has been

\* remotely shut down for further writing, or has an error pending, then it

\* will add <tt>OP\_WRITE</tt> to the key's ready set and add the key to its

\* selected-key&nbsp;set. </p>

\*/

**public** **static** **final** **int** *OP\_WRITE* = 1 << 2;

/\*\*

\* Operation-set bit for socket-connect operations. </p>

\*

\* <p> Suppose that a selection key's interest set contains

\* <tt>OP\_CONNECT</tt> at the start of a <a

\* href="Selector.html#selop">selection operation</a>. If the selector

\* detects that the corresponding socket channel is ready to complete its

\* connection sequence, or has an error pending, then it will add

\* <tt>OP\_CONNECT</tt> to the key's ready set and add the key to its

\* selected-key&nbsp;set. </p>

\*/

**public** **static** **final** **int** *OP\_CONNECT* = 1 << 3;

/\*\*

\* Operation-set bit for socket-accept operations. </p>

\*

\* <p> Suppose that a selection key's interest set contains

\* <tt>OP\_ACCEPT</tt> at the start of a <a

\* href="Selector.html#selop">selection operation</a>. If the selector

\* detects that the corresponding server-socket channel is ready to accept

\* another connection, or has an error pending, then it will add

\* <tt>OP\_ACCEPT</tt> to the key's ready set and add the key to its

\* selected-key&nbsp;set. </p>

\*/

**public** **static** **final** **int** *OP\_ACCEPT* = 1 << 4;

**public** NioServerSocketChannel(ServerSocketChannel channel) {

**super**(**null**, channel, SelectionKey.*OP\_ACCEPT*);

config = **new** NioServerSocketChannelConfig(**this**, javaChannel().socket());

}

启动NioEventLoop线程池

eventLoop().execute()启动线程

**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.

channel.eventLoop().execute(**new** Runnable() {

@Override

**public** **void** run() {

**if** (regFuture.isSuccess()) {

channel.bind(localAddress, promise).addListener(ChannelFutureListener.*CLOSE\_ON\_FAILURE*);

} **else** {

promise.setFailure(regFuture.cause());

}

}

});

}

EventLoop选择器

**private** **interface** EventExecutorChooser {

EventExecutor next();

}

**private** **final** **class** PowerOfTwoEventExecutorChooser **implements** EventExecutorChooser {

@Override

**public** EventExecutor next() {

**return** children[childIndex.getAndIncrement() & children.length - 1];

}

}

**private** **final** **class** GenericEventExecutorChooser **implements** EventExecutorChooser {

@Override

**public** EventExecutor next() {

**return** children[Math.*abs*(childIndex.getAndIncrement() % children.length)];

}

}

NioServerSocketChannel创建过程：

1. ServerBootstrap - **public** B channel(Class<? **extends** C> channelClass),bind()中真正初始化NioServerSocketChannel
2. BootstrapChannelFactory - **public** T newChannel()
3. **public** NioServerSocketChannel() {}
4. *DEFAULT\_SELECTOR\_PROVIDER.* openServerSocketChannel()
5. AbstractNioMessageChannel(**null**, channel, SelectionKey.*OP\_ACCEPT*)
6. AbstractChannel (Channel parent)
7. AbstractNioMessageChannel. newUnsafe()
8. **new** NioMessageUnsafe()
9. config = **new** NioServerSocketChannelConfig(**this**, javaChannel().socket());

全局Executor

GlobalEventExecutor ChannelFuture.sync()中使用

AbstractBootstrap

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

regFuture.operationComplete();

ChannelInboundHandlerAdapter.channelRead()实现此类的方法注意，msg内存不会自动释放，继承了SimpleChannelInboundHandler的类，会自动释放Msg。

SimpleChannelInboundHandler

@Override

**public** **void** channelRead(ChannelHandlerContext ctx, Object msg) **throws** Exception {

**boolean** release = **true**;

**try** {

**if** (acceptInboundMessage(msg)) {

@SuppressWarnings("unchecked")

I imsg = (I) msg;

channelRead0(ctx, imsg);

} **else** {

release = **false**;

ctx.fireChannelRead(msg);

}

} **finally** {

**if** (autoRelease && release) {

ReferenceCountUtil.*release*(msg);

}

}

}

ServerBootstrap,bind流程：

1. TCPServer – ServerBootstrap.bind(server,port)
2. AbstractBootstrap.doBind(server,port)
3. AbstractBootstrap.doBind(**final** SocketAddress localAddress)
4. AbstractBootstrap.initAndRegister()异步register
5. AbstractBootstrap.init(channel)
6. MultithreadEventLoopGroup.register(Channel channel)
7. SingleThreadEventLoop.register(Channel channel)
8. SingleThreadEventLoop.register(**final** Channel channel, **final** ChannelPromise promise)
9. AbstractUnsafe.register(EventLoop eventLoop, **final** ChannelPromise promise)
10. SingleThreadEventExecutor.execute(Runnable task)
11. startThread();正式启动接收连接的线程
12. NioeventLoop.run()
13. NioeventLoop.runAllTasks();task.run()
14. AbstractBootstrap.doBind(…); 异步bind

ServerBootstrapAcceptor:

p.addLast(**new** ChannelInitializer<Channel>() {

@Override

**public** **void** initChannel(Channel ch) **throws** Exception {

ch.pipeline().addLast(**new** ServerBootstrapAcceptor(

currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));

}

});

SelectionKey.OP\_READ

1<<0 = 1

SelectionKey.OP\_WRITE

1<<2 = 4

SelectionKey.OP\_CONNECT

1<<3 = 8

SelectionKey.OP\_ACCEPT

1<<4 = 16

NioServerSocketChannel创建时默认InterestKey为*OP\_ACCEPT*

**public** NioServerSocketChannel(ServerSocketChannel channel) {

**super**(**null**, channel, SelectionKey.*OP\_ACCEPT*);

config = **new** NioServerSocketChannelConfig(**this**, javaChannel().socket());

}

NioSocketChannel创建时默认InterestKey为OP\_READ

**protected** AbstractNioByteChannel(Channel parent, SelectableChannel ch) {

**super**(parent, ch, SelectionKey.*OP\_READ*);

}

DefaultChannelPipeLine fireChannelXXX事件触发源头：

pipeline.fireChannelRegistered()

AbstractChannel：

**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;

doRegister();

neverRegistered = **false**;

registered = **true**;

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** (firstRegistration && isActive()) {

pipeline.fireChannelActive();

}

} **catch** (Throwable t) {

// Close the channel directly to avoid FD leak.

closeForcibly();

closeFuture.setClosed();

safeSetFailure(promise, t);

}

}

AbstractNioChannel：

**protected** **void** doRegister() **throws** Exception {

**boolean** selected = **false**;

**for** (;;) {

**try** {

selectionKey = javaChannel().register(eventLoop().selector, 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.fireChannelUnregistered();

AbstractChannel. deregister(**final** ChannelPromise promise)

pipeline.fireChannelActive();

Server - bind：

bind(**final** SocketAddress localAddress, **final** ChannelPromise promise)

Client - connect:

AbstractNioChannel:

fulfillConnectPromise(ChannelPromise promise, **boolean** wasActive)

pipeline.fireChannelInActive();

AbstractChannel.close(**final** ChannelPromise promise)

closeAndDeregister(ChannelOutboundBuffer outboundBuffer, **final** **boolean** wasActive)

AbstractChannel.disconnect(**final** ChannelPromise promise)

pipeline.fireChannelRead();

pipeline.fireChannelReadComplete

server:NioMessageUnsafe.read();

client: NioByteUnsafe.read();