# Netty源码解析

## 第1节 Netty深入剖析

### ·使用Netty作为底层框架的计数



### ·Netty是什么

·异步事件驱动框架，用于快速开发高性能服务器和客户端

·封装了JDK底层BIO和NIO模型，提供高度可用的API

·自带编解码器解决拆包粘包问题，用户只关心业务逻辑

·精心设计的reactor线程模型支持高并发海量连接

·自带各种协议栈

### ·学习的必要性

·各大开源项目选择Netty作为底层通信框架

·更好的使用，少走弯路

·遇到Bug？单机连接数上不去？性能遇到瓶颈？如何调优

·详解reactor线程模型，实践中举一反三

·庞大的项目是如何组织的，设计模式，体验优秀的设计

·阅读源码其实没那么困难

### ·怎么学

·自己摸索不如前人之路

·对于Socket编程，逐个切入

·踩过的坑，积累的经验总结

·调试，分析技巧

·各章节独立模块进行拆分

·总分总的方式渐进讲解

·实例演示，试图演示，调试分析

### ·学习目标

·掌握Netty底层原理，轻松解决各类疑难杂症，深度调优

·给官方提issue

·实现一个简易版的Netty

·开启阅读源码之旅

### ·适合人群

·系统掌握Netty底层核心原理

·加速掌握基于Nett的各种中间件框架

·对技术有追求，对优秀代码有执念的开发者

·从未阅读过源码，有勇气迈出第一步的开发者

### ·技术储备

·熟悉java基础，熟悉多线程

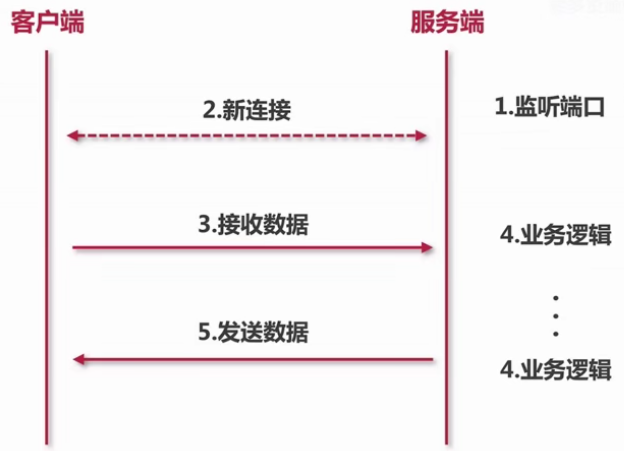
·熟悉TCP原理，NIO

·使用过Netty

·maven，Intellij IDEA

## Netty基本组件

### 2.1 TCP示例

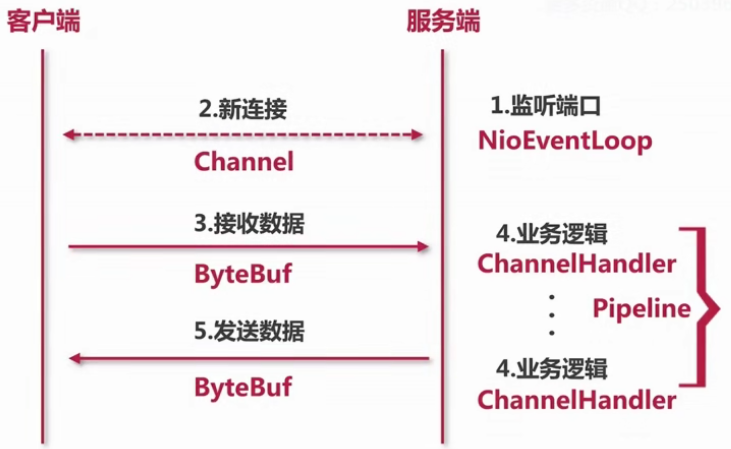


|  |
| --- |
| **import** java.io.IOException; **import** java.net.ServerSocket; **import** java.net.Socket;  **public class** Server {   **private** ServerSocket **serverSocket**;   **public** Server(**int** port) {  **try** {  **this**.**serverSocket** = **new** ServerSocket(port);  System.***out***.println(**"服务端启动成功，端口:"** + port);  } **catch** (IOException exception) {  System.***out***.println(**"服务端启动失败"**);  }  }   **public void** start() {  **new** Thread(**new** Runnable() {  @Override  **public void** run() {  doStart();  }  }).start();  }   **private void** doStart() {  **while** (**true**) {  **try** {  Socket client = **serverSocket**.accept();  **new** ClientHandler(client).start();  } **catch** (IOException e) {  System.***out***.println(**"服务端异常"**);  }  }  } } |

|  |
| --- |
| **import** java.io.IOException; **import** java.io.InputStream; **import** java.net.Socket;  **public class** ClientHandler {   **public static final int *MAX\_DATA\_LEN*** = 1024;  **private final** Socket **socket**;   **public** ClientHandler(Socket socket) {  **this**.**socket** = socket;  }   **public void** start() {  System.***out***.println(**"新客户端接入"**);  **new** Thread(**new** Runnable() {  @Override  **public void** run() {  doStart();  }  }).start();  }   **private void** doStart() {  **try** {  InputStream inputStream = **socket**.getInputStream();  **while** (**true**) {  **byte**[] data = **new byte**[***MAX\_DATA\_LEN***];  **int** len;  **while** ((len = inputStream.read(data)) != -1) {  String message = **new** String(data, 0, len);  System.***out***.println(**"客户端传来消息: "** + message);  **socket**.getOutputStream().write(data);  }   }    } **catch** (IOException e) {  e.printStackTrace();  }  } } |

|  |
| --- |
| **public class** ServerBoot {   **private static final int *PORT*** = 8000;   **public static void** main(String[] args) {  Server server = **new** Server(***PORT***);  server.start();  }  } |
| **import** java.io.IOException; **import** java.net.Socket;  **public class** Client {  **private static final** String ***HOST*** = **"127.0.0.1"**;  **private static final int *PORT*** = 8000;  **private static final int *SLEEP\_TIME*** = 5000;   **public static void** main(String[] args) **throws** IOException {  **final** Socket socket = **new** Socket(***HOST***, ***PORT***);   **new** Thread(**new** Runnable() {  @Override  **public void** run() {  System.out.println(**"客户端启动成功!"**);  **while** (**true**) {  **try** {  String message = **"hello world"**;  System.out.println(**"客户端发送数据: "** + message);  socket.getOutputStream().write(message.getBytes());  } **catch** (Exception e) {  System.out.println(**"写数据出错!"**);  }  sleep();  }    }  }).start();   }   **private static void** sleep() {  **try** {  Thread.*sleep*(***SLEEP\_TIME***);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  } } |
| **服务端启动成功，端口:8000**  **新客户端接入**  **客户端传来消息: hello world**  **客户端传来消息: hello world**  **客户端传来消息: hello world**  **客户端传来消息: hello world** |
| **客户端启动成功!**  **客户端发送数据: hello world**  **客户端发送数据: hello world**  **客户端发送数据: hello world**  **客户端发送数据: hello world**  **客户端发送数据: hello world** |

### 2.2Netty对于socket的抽象

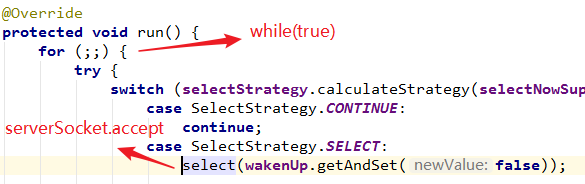


### 2.3Netty组件介绍

#### ·NioEventLoop == Socket示例中的Thread

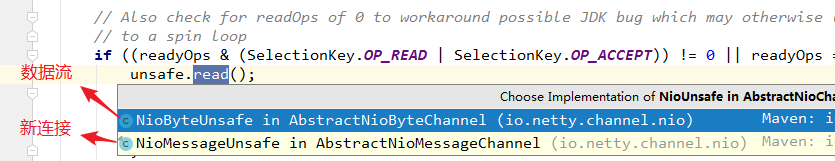
1. 服务端接收客户端连接线程；
2. 处理每个连接的读写线程；
3. 启动两种类型的线程。

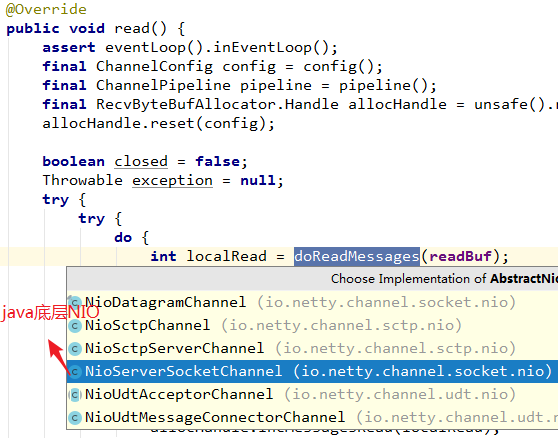
|  |
| --- |
| Tips：intelij iead快捷键  Ctrl + n 模糊搜索类名  Ctrl + f12 类中搜索方法  Ctrl + e 弹出最近使用的文件，可快速切换  Alt + Shift + a 最近修改过的class  Ctrl + alt + b 查看方法的实现类 |

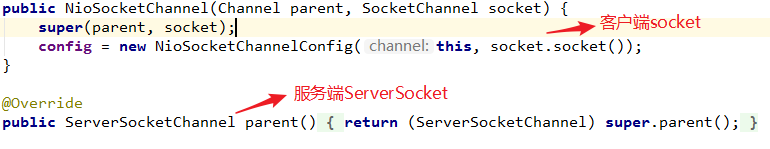


#### ·Channel == socket

1. NioSocketChannel对应客户端socket；
2. ServerSocketChannel对应服务端ServerSocket；
3. Channel对socket封装；

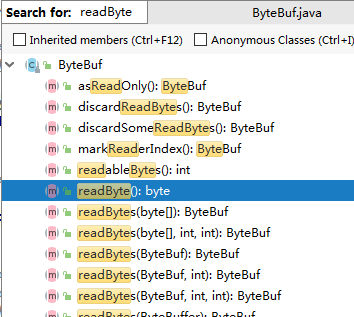


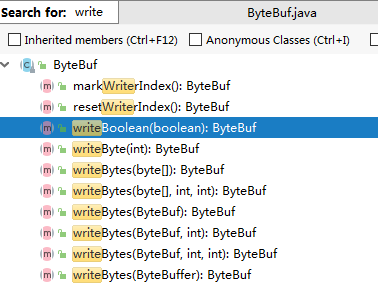




#### ·ByteBuf == IO bytes

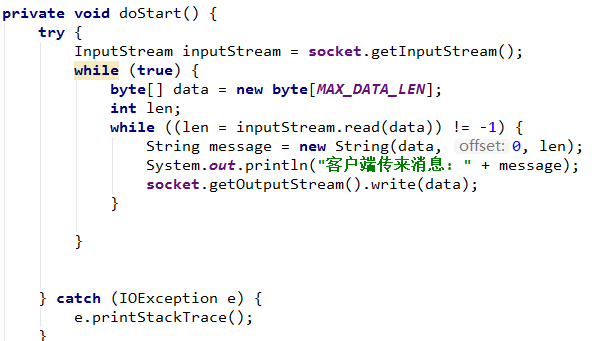
1. readByte对应输入流
2. writeByte对应输出流

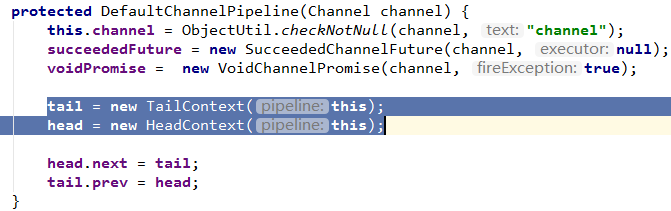




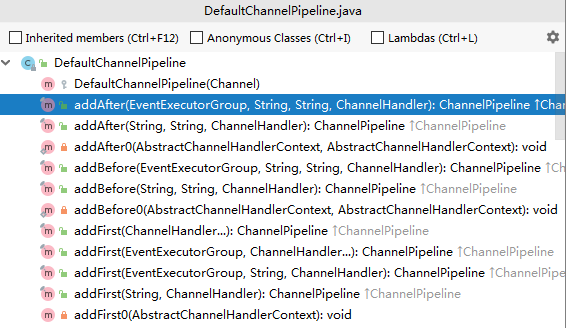
#### ·Pipeline == Logic Chain

（1）对应处理FilterChain，本质是一个链表





·ChannelHandler == Logic



## 第3节 Netty服务端启动

### 3.1Netty demo

|  |
| --- |
| **public class** ServerHandler **extends** ChannelInboundHandlerAdapter {  @Override  **public void** channelActive(ChannelHandlerContext ctx) {  System.***out***.println(**"channelActive"**);  }   @Override  **public void** channelRegistered(ChannelHandlerContext ctx) {  System.***out***.println(**"channelRegistered"**);  }   @Override  **public void** handlerAdded(ChannelHandlerContext ctx) {  System.***out***.println(**"handlerAdded"**);  }   @Override  **public void** channelRead(**final** ChannelHandlerContext ctx, Object msg) **throws** Exception {  **super**.channelRead(ctx, msg);   **new** Thread(**new** Runnable() {  @Override  **public void** run() {  *// 耗时的操作* String result = loadFromDB();   ctx.channel().writeAndFlush(result);  ctx.executor().schedule(**new** Runnable() {  @Override  **public void** run() {  *// ...* }  }, 1, TimeUnit.SECONDS);   }  }).start();  }   **private** String loadFromDB() {  **return "hello world!"**;  } } |

|  |
| --- |
| **public class** AuthHandler **extends** SimpleChannelInboundHandler<ByteBuf> {  @Override  **public void** channelRead(ChannelHandlerContext ctx, Object msg) **throws** Exception {  *//* }   @Override  **protected void** channelRead0(ChannelHandlerContext ctx, ByteBuf password) **throws** Exception {  **if** (paas(password)) {  ctx.pipeline().remove(**this**);  } **else** {  ctx.close();  }  }   **private boolean** paas(ByteBuf password) {  **return false**;  } } |

|  |
| --- |
| **import** io.netty.bootstrap.ServerBootstrap; **import** io.netty.channel.ChannelFuture; **import** io.netty.channel.ChannelInitializer; **import** io.netty.channel.ChannelOption; **import** io.netty.channel.EventLoopGroup; **import** io.netty.channel.nio.NioEventLoopGroup; **import** io.netty.channel.socket.SocketChannel; **import** io.netty.channel.socket.nio.NioServerSocketChannel; **import** io.netty.util.AttributeKey;  **public class** Server {  **public static void** main(String[] args) {  EventLoopGroup bossGroup = **new** NioEventLoopGroup(1);  EventLoopGroup workerGroup = **new** NioEventLoopGroup();   ServerBootstrap b = **new** ServerBootstrap();  b.group(bossGroup, workerGroup)  .channel(NioServerSocketChannel.**class**)  .childOption(ChannelOption.***TCP\_NODELAY***, **true**)  .childAttr(AttributeKey.*newInstance*(**"childAttr"**), **"childAttrValue"**)  .handler(**new** ServerHandler())  .childHandler(**new** ChannelInitializer<SocketChannel>() {  @Override  **protected void** initChannel(SocketChannel ch) **throws** Exception {  ch.pipeline().addLast(**new** AuthHandler());  }  });  **try** {  ChannelFuture f = b.bind(8888).sync();  f.channel().closeFuture().sync();  } **catch** (InterruptedException e) {  e.printStackTrace();  } **finally** {  bossGroup.shutdownGracefully();  workerGroup.shutdownGracefully();  }  } } |

### 3.2服务端Channel创建

两个问题：

1. 服务端的socket在哪里初始化；
2. 在哪里accept连接；

Netty服务端启动4个过程：

1. 创建服务端Channel；调用JDK底层API创建JDK的channel，然后Netty将其包装为自己的channel，同时创建一些基本组件，绑定在此channel上。
2. 初始化服务端Channel；初始化基本属性，添加逻辑处理器。
3. 注册selector；Netty将channel注册到事件轮训器selector，并把服务端channel作为Attachment，绑定在JDK底层的服务端Channel。这样后续有事件轮训出来，就可以直接拿到Attachment，那么这个Attachment就是Netty封装的一个Channel。
4. 端口绑定。

|  |
| --- |
| ChannelFuture f = b.bind(8888).sync(); |
| **public** ChannelFuture bind(**int** inetPort) {  **return** bind(**new** InetSocketAddress(inetPort)); } |
| **public** ChannelFuture bind(SocketAddress localAddress) {  validate();  **if** (localAddress == **null**) {  **throw new** NullPointerException(**"localAddress"**);  }  **return** doBind(localAddress); } |
| **private** ChannelFuture doBind(**final** SocketAddress localAddress) {  **final** ChannelFuture regFuture = initAndRegister(); |
| **final** ChannelFuture initAndRegister() {  Channel channel = **null**;  **try** {  channel = **channelFactory**.newChannel(); |

（1）服务端的socket在哪里初始化？

NioServerSocketChannel



|  |
| --- |
| **private static** ServerSocketChannel newSocket(SelectorProvider provider) {  **try** {  */\*\*  \* Use the {****@link*** *SelectorProvider} to open {****@link*** *SocketChannel} and so remove condition in  \* {****@link*** *SelectorProvider#provider()} which is called by each ServerSocketChannel.open() otherwise.  \*  \* See <a href="https://github.com/netty/netty/issues/2308">#2308</a>.  \*/* **return** provider.openServerSocketChannel();  } **catch** (IOException e) {  **throw new** ChannelException(  **"Failed to open a server socket."**, e);  } } |

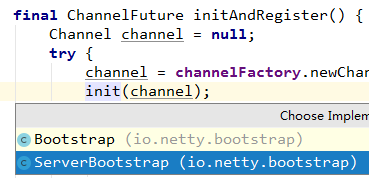
反射创建服务端Channel

|  |
| --- |
| .channel(NioServerSocketChannel.**class**) |
| **public** NioServerSocketChannel(SelectorProvider provider) {  **this**(*newSocket*(provider)); // 通过JDK底层来创建jdk channel } |
| **public** NioServerSocketChannel(ServerSocketChannel channel) {  **super**(**null**, channel, SelectionKey.***OP\_ACCEPT***);  **config** = **new** NioServerSocketChannelConfig(**this**, javaChannel().socket()); //TCP参数类配置 } |
| **protected** AbstractNioChannel(Channel parent, SelectableChannel ch, **int** readInterestOp) {  **super**(parent);  **this**.**ch** = ch;  **this**.**readInterestOp** = readInterestOp;  **try** {  ch.configureBlocking(**false**); // 设置非阻塞 |
| **protected** AbstractChannel(Channel parent) {  **this**.**parent** = parent;  // 创建id，unsafe，pipeline  **id** = newId();  **unsafe** = newUnsafe();  **pipeline** = newChannelPipeline(); } |

### 3.3服务端Channel初始化







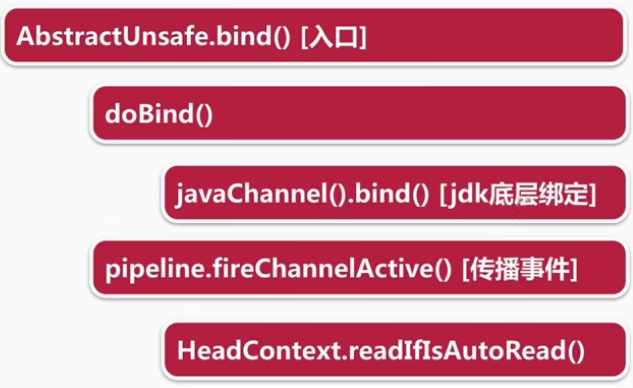
|  |
| --- |
| @Override **void** init(Channel channel) **throws** Exception {  **final** Map<ChannelOption<?>, Object> options = options0();  **synchronized** (options) {  channel.config().setOptions(options);  }   **final** Map<AttributeKey<?>, Object> attrs = attrs0();  **synchronized** (attrs) {  **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()));  }   p.addLast(**new** ChannelInitializer<Channel>() {  @Override  **public void** initChannel(Channel ch) **throws** Exception {  **final** ChannelPipeline pipeline = ch.pipeline();  ChannelHandler handler = **config**.handler();  **if** (handler != **null**) {  pipeline.addLast(handler);  }   *// We add this handler via the EventLoop as the user may have used a ChannelInitializer as handler.  // In this case the initChannel(...) method will only be called after this method returns. Because  // of this we need to ensure we add our handler in a delayed fashion so all the users handler are  // placed in front of the ServerBootstrapAcceptor.* ch.eventLoop().execute(**new** Runnable() {  @Override  **public void** run() {  pipeline.addLast(**new** ServerBootstrapAcceptor(  currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));  }  });  }  }); } |

### 3.4注册selector



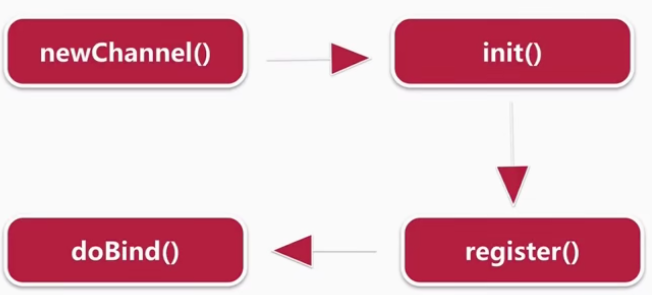
|  |
| --- |
| **private** ChannelFuture doBind(**final** SocketAddress localAddress) {  **final** ChannelFuture regFuture = initAndRegister(); |
| ChannelFuture regFuture = config().group().register(channel); |
|  |
|  |
|  |
|  |

### 3.5端口绑定



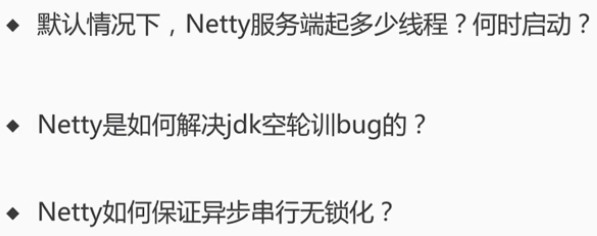
|  |
| --- |
| ChannelFuture f = b.bind(8888).sync(); |
|  |
|  |
|  |
| @Override **protected void** doBind(SocketAddress localAddress) **throws** Exception {  **if** (PlatformDependent.*javaVersion*() >= 7) {  javaChannel().bind(localAddress, **config**.getBacklog());  } **else** {  javaChannel().socket().bind(localAddress, **config**.getBacklog());  } } |
|  |
| @Override **public void** channelActive(ChannelHandlerContext ctx) **throws** Exception {  ctx.fireChannelActive();   readIfIsAutoRead(); } |
|  |
| @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) {  selectionKey.interestOps(interestOps | **readInterestOp**);  } } |
|  |
| **protected** AbstractNioMessageChannel(Channel parent, SelectableChannel ch, **int** readInterestOp) {  **super**(parent, ch, readInterestOp); } |

### 3.6服务端启动核心路径总结



## 第4节 NioEventLoop

### 4.1NioEventLoop三个问题



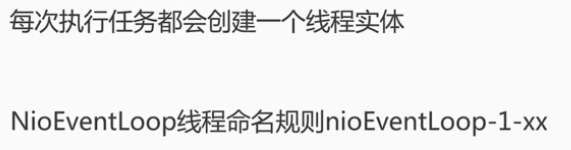
### 4.2 NioEventLoop创建



具体代码：

|  |
| --- |
| EventLoopGroup bossGroup = **new** NioEventLoopGroup(1); EventLoopGroup workerGroup = **new** NioEventLoopGroup(); |
| **public** NioEventLoopGroup(**int** nThreads) {  **this**(nThreads, (Executor) **null**); } |
| **public** NioEventLoopGroup(**int** nThreads, Executor executor) {  **this**(nThreads, executor, SelectorProvider.*provider*()); } |
| **public** NioEventLoopGroup(  **int** nThreads, Executor executor, **final** SelectorProvider selectorProvider) {  **this**(nThreads, executor, selectorProvider, DefaultSelectStrategyFactory.***INSTANCE***); } |
| **public** NioEventLoopGroup(**int** nThreads, Executor executor, **final** SelectorProvider selectorProvider,  **final** SelectStrategyFactory selectStrategyFactory) {  **super**(nThreads, executor, selectorProvider, selectStrategyFactory, RejectedExecutionHandlers.*reject*()); } |
| **protected** MultithreadEventLoopGroup(**int** nThreads, Executor executor, Object... args) {  **super**(nThreads == 0 ? ***DEFAULT\_EVENT\_LOOP\_THREADS*** : nThreads, executor, args); } |
| **private static final int *DEFAULT\_EVENT\_LOOP\_THREADS***;  **static** {  ***DEFAULT\_EVENT\_LOOP\_THREADS*** = Math.*max*(1, SystemPropertyUtil.*getInt*(  **"io.netty.eventLoopThreads"**, Runtime.*getRuntime*().availableProcessors() \* 2));   **if** (***logger***.isDebugEnabled()) {  ***logger***.debug(**"-Dio.netty.eventLoopThreads: {}"**, ***DEFAULT\_EVENT\_LOOP\_THREADS***);  } } |
| **protected** MultithreadEventExecutorGroup(**int** nThreads, Executor executor, Object... args) {  **this**(nThreads, executor, DefaultEventExecutorChooserFactory.***INSTANCE***, args); } |
| **public static final** DefaultEventExecutorChooserFactory ***INSTANCE*** = **new** DefaultEventExecutorChooserFactory(); |
|  |

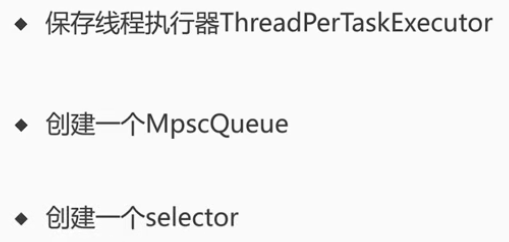
### 4.3 ThreadPerTaskExecutor



nioEventLoop线程名首字符小写的原因：

|  |
| --- |
| **public static** String toPoolName(Class<?> poolType) {  **if** (poolType == **null**) {  **throw new** NullPointerException(**"poolType"**);  }   String poolName = StringUtil.*simpleClassName*(poolType);  **switch** (poolName.length()) {  **case** 0:  **return "unknown"**;  **case** 1:  **return** poolName.toLowerCase(Locale.***US***);  **default**:  **if** (Character.*isUpperCase*(poolName.charAt(0)) && Character.*isLowerCase*(poolName.charAt(1))) {  **return** Character.*toLowerCase*(poolName.charAt(0)) + poolName.substring(1);  } **else** {  **return** poolName;  }  } } |
| **public** DefaultThreadFactory(String poolName, **boolean** daemon, **int** priority, ThreadGroup threadGroup) {  **if** (poolName == **null**) {  **throw new** NullPointerException(**"poolName"**);  }  **if** (priority < Thread.***MIN\_PRIORITY*** || priority > Thread.***MAX\_PRIORITY***) {  **throw new** IllegalArgumentException(  **"priority: "** + priority + **" (expected: Thread.MIN\_PRIORITY <= priority <= Thread.MAX\_PRIORITY)"**);  }   **prefix** = poolName + **'-'** + ***poolId***.incrementAndGet() + **'-'**;  **this**.**daemon** = daemon;  **this**.**priority** = priority;  **this**.**threadGroup** = threadGroup; } |
| @Override **public** Thread newThread(Runnable r) {  Thread t = newThread(**new** DefaultRunnableDecorator(r), **prefix** + **nextId**.incrementAndGet());  **try** {  **if** (t.isDaemon()) {  **if** (!**daemon**) {  t.setDaemon(**false**);  }  } **else** {  **if** (**daemon**) {  t.setDaemon(**true**);  }  }   **if** (t.getPriority() != **priority**) {  t.setPriority(**priority**);  }  } **catch** (Exception ignored) {  *// Doesn't matter even if failed to set.* }  **return** t; } |
| **public** FastThreadLocalThread(ThreadGroup group, Runnable target, String name) {  **super**(group, target, name); } |

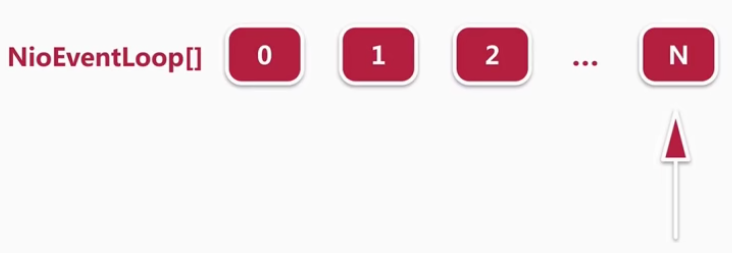
### 4.4创建NioEventLoop线程



|  |
| --- |
|  |
| @Override **protected** EventLoop newChild(Executor executor, Object... args) **throws** Exception {  **return new** NioEventLoop(**this**, executor, (SelectorProvider) args[0],  ((SelectStrategyFactory) args[1]).newSelectStrategy(), (RejectedExecutionHandler) args[2]); } |
| 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;  **selector** = openSelector();  **selectStrategy** = strategy; } |
| **protected** SingleThreadEventExecutor(EventExecutorGroup parent, Executor executor,  **boolean** addTaskWakesUp, **int** maxPendingTasks,  RejectedExecutionHandler rejectedHandler) {  **super**(parent);  **this**.**addTaskWakesUp** = addTaskWakesUp;  **this**.**maxPendingTasks** = Math.*max*(16, maxPendingTasks);  **this**.**executor** = ObjectUtil.*checkNotNull*(executor, **"executor"**);  **taskQueue** = newTaskQueue(**this**.**maxPendingTasks**);  **rejectedExecutionHandler** = ObjectUtil.*checkNotNull*(rejectedHandler, **"rejectedHandler"**); } |
|  |
| @Override **protected** Queue<Runnable> newTaskQueue(**int** maxPendingTasks) {  *// This event loop never calls takeTask()* **return** PlatformDependent.*newMpscQueue*(maxPendingTasks); } |

### 4.5创建线程选择器

·NioEventLoopGroup.next()

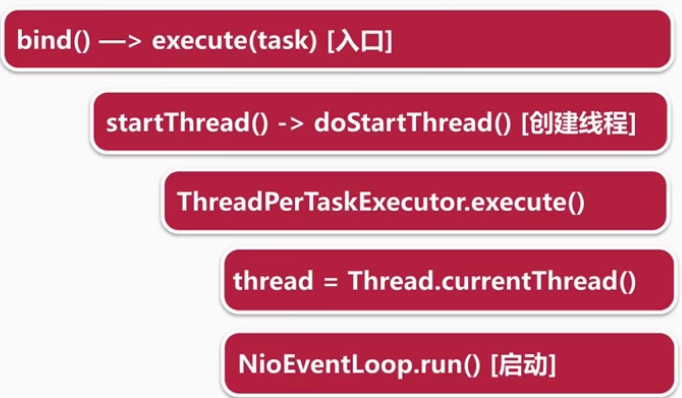


·chooserFactory.newChooser()



|  |
| --- |
| **chooser** = chooserFactory.newChooser(**children**); |
| @Override **public** EventExecutorChooser newChooser(EventExecutor[] executors) {  **if** (*isPowerOfTwo*(executors.**length**)) {  **return new** PowerOfTowEventExecutorChooser(executors);  } **else** {  **return new** GenericEventExecutorChooser(executors);  } } |
| **private static final class** GenericEventExecutorChooser **implements** EventExecutorChooser {  **private final** AtomicInteger **idx** = **new** AtomicInteger();  **private final** EventExecutor[] **executors**;   GenericEventExecutorChooser(EventExecutor[] executors) {  **this**.**executors** = executors;  }   @Override  **public** EventExecutor next() {  **return executors**[Math.*abs*(**idx**.getAndIncrement() % **executors**.**length**)];  } } |
| **private static final class** PowerOfTowEventExecutorChooser **implements** EventExecutorChooser {  **private final** AtomicInteger **idx** = **new** AtomicInteger();  **private final** EventExecutor[] **executors**;   PowerOfTowEventExecutorChooser(EventExecutor[] executors) {  **this**.**executors** = executors;  }   @Override  **public** EventExecutor next() {  **return executors**[**idx**.getAndIncrement() & **executors**.**length** - 1];  } } |
|  |
| %取模效率低下，&取余更贴合计算机底层位运算 |

### 4.6 NioEventLoop启动



|  |
| --- |
| ChannelFuture f = b.bind(8888).sync(); |
| **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());  }  }  }); } |
|  |
| @Override **public void** execute(Runnable task) {  **if** (task == **null**) {  **throw new** NullPointerException(**"task"**);  }   **boolean** inEventLoop = inEventLoop();  **if** (inEventLoop) {  addTask(task);  } **else** {  startThread();  addTask(task);  **if** (isShutdown() && removeTask(task)) {  *reject*();  }  }   **if** (!**addTaskWakesUp** && wakesUpForTask(task)) {  wakeup(inEventLoop);  } } |
| @Override **public boolean** inEventLoop(Thread thread) {  **return** thread == **this**.**thread**; } |
|  |
|  |
| **protected** Thread newThread(Runnable r, String name) {  **return new** FastThreadLocalThread(**threadGroup**, r, name); } |

### 4.7 NioEventLoop执行逻辑；

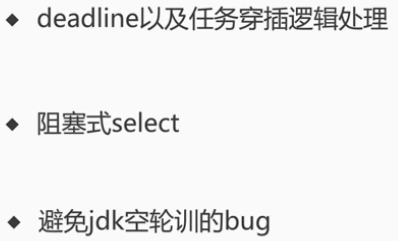


具体代码：

|  |
| --- |
| **private void** doStartThread() {  **assert thread** == **null**;  **executor**.execute(**new** Runnable() {  @Override  **public void** run() {  **thread** = Thread.*currentThread*();  **if** (**interrupted**) {  **thread**.interrupt();  }   **boolean** success = **false**;  updateLastExecutionTime();  **try** {  SingleThreadEventExecutor.**this**.run();  success = **true**; |
|  |
| @Override **protected void** run() {  **for** (;;) {  **try** {  **switch** (**selectStrategy**.calculateStrategy(**selectNowSupplier**, hasTasks())) {  **case** SelectStrategy.***CONTINUE***:  **continue**;  **case** SelectStrategy.***SELECT***:  select(**wakenUp**.getAndSet(**false**));**if** (**wakenUp**.get()) {  **selector**.wakeup();  }  **default**:  *// fallthrough* } |
| **cancelledKeys** = 0; **needsToSelectAgain** = **false**; **final int** ioRatio = **this**.**ioRatio**; // 默认50 **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);  } } |

### 4.8 Step1检测IO事件Select

·select方法执行逻辑



|  |
| --- |
| **private void** select(**boolean** oldWakenUp) **throws** IOException {  Selector selector = **this**.**selector**;  **try** {  **int** selectCnt = 0;  **long** currentTimeNanos = System.*nanoTime*();  **long** selectDeadLineNanos = currentTimeNanos + delayNanos(currentTimeNanos); |
| **for** (;;) {  **long** timeoutMillis = (selectDeadLineNanos - currentTimeNanos + 500000L) / 1000000L;  **if** (timeoutMillis <= 0) {  // 超期 终止select  **if** (selectCnt == 0) {  selector.selectNow();  selectCnt = 1;  }  **break**;  } |
| **if** (hasTasks() && **wakenUp**.compareAndSet(**false**, **true**)) {  // 任务队列不空 终止select  selector.selectNow();  selectCnt = 1;  **break**; } |

·截止时间没到，并且任务队列为空，进行阻塞式select

|  |
| --- |
| **int** selectedKeys = selector.select(timeoutMillis); selectCnt ++;  **if** (selectedKeys != 0 || oldWakenUp || **wakenUp**.get() || hasTasks() || hasScheduledTasks()) {  *// 空轮（512次）训时，以下四种事件发生*  *// - Selected something,  // - waken up by user, or  // - the task queue has a pending task.  // - a scheduled task is ready for processing* **break**; } |

·避免jdk空轮训的bug

|  |
| --- |
| **int** selectedKeys = selector.select(timeoutMillis); selectCnt ++; |
| **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(512) 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; } |
| */\*\*  \* Replaces the current {****@link*** *Selector} of this event loop with newly created {****@link*** *Selector}s to work  \* around the infamous epoll 100% CPU bug.  \*/* **public void** rebuildSelector() { |
| *// Register all channels to the new Selector.* **int** nChannels = 0; **for** (;;) {  **try** {  **for** (SelectionKey key: oldSelector.keys()) { |
| **int** interestOps = key.interestOps(); key.cancel(); SelectionKey newKey = key.channel().register(newSelector, interestOps, a); |

### 4.9 Step2处理IO事件 ProcessSelectKeys

【总结】Netty默认情况下会通过反射将Selector底层的HashSet转换成数组的方式进行优化，然后在处理每个KeySet时，都会拿到对应的Attachment，而这个Attachment就是在向Selector注册IO事件时，经过Netty封装后的Channel。

·processSelectedKey()执行逻辑

·selected keySet优化，Netty将hashSet改进为封装的数组，提高select效率O(1)

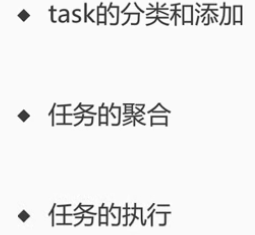
|  |
| --- |
| NioEventLoop(NioEventLoopGroup parent, Executor executor, SelectorProvider selectorProvider,  SelectStrategy strategy, RejectedExecutionHandler rejectedExecutionHandler) {  **super**(parent, executor, **false**, ***DEFAULT\_MAX\_PENDING\_TASKS***, rejectedExecutionHandler);  **if** (selectorProvider == **null**) {  **throw new** NullPointerException(**"selectorProvider"**);  }  **if** (strategy == **null**) {  **throw new** NullPointerException(**"selectStrategy"**);  }  **provider** = selectorProvider;  **selector** = openSelector();  **selectStrategy** = strategy; } |
| **if** (***DISABLE\_KEYSET\_OPTIMIZATION***) {  // 默认优化Set到Array优化  **return** selector; } |
| **final** SelectedSelectionKeySet selectedKeySet = **new** SelectedSelectionKeySet(); |
| **final class** SelectedSelectionKeySet **extends** AbstractSet<SelectionKey> {   **private** SelectionKey[] **keysA**;  **private int keysASize**; |
| @Override **public boolean** add(SelectionKey o) {  **if** (o == **null**) {  **return false**;  }   **if** (**isA**) {  **int** size = **keysASize**;  **keysA**[size ++] = o;  **keysASize** = size;  **if** (size == **keysA**.**length**) {  doubleCapacityA();  }  } |
| **private void** doubleCapacityA() {  SelectionKey[] newKeysA = **new** SelectionKey[**keysA**.**length** << 1];  System.*arraycopy*(**keysA**, 0, newKeysA, 0, **keysASize**);  **keysA** = newKeysA; } |
| **// 废弃Netty不用的Set接口**  @Override **public boolean** remove(Object o) {  **return false**; }  @Override **public boolean** contains(Object o) {  **return false**; }  @Override **public** Iterator<SelectionKey> iterator() {  **throw new** UnsupportedOperationException(); } |
| Field selectedKeysField = selectorImplClass.getDeclaredField(**"selectedKeys"**); Field publicSelectedKeysField = selectorImplClass.getDeclaredField(**"publicSelectedKeys"**);  selectedKeysField.setAccessible(**true**); publicSelectedKeysField.setAccessible(**true**);  selectedKeysField.set(selector, selectedKeySet); publicSelectedKeysField.set(selector, selectedKeySet); |
| **if** (maybeException **instanceof** Exception) {  **selectedKeys** = **null**;  Exception e = (Exception) maybeException;  ***logger***.trace(**"failed to instrument a special java.util.Set into: {}"**, selector, e); } **else** {  **selectedKeys** = selectedKeySet;  ***logger***.trace(**"instrumented a special java.util.Set into: {}"**, selector); }  **return** selector; |

·processSelectedKeysOptimized()，真正处理IO事件

|  |
| --- |
| **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);  } } |
| **private void** processSelectedKeys() {  **if** (**selectedKeys** != **null**) {  processSelectedKeysOptimized(**selectedKeys**.flip());  } **else** {  processSelectedKeysPlain(**selector**.selectedKeys());  } } |
| SelectionKey[] flip() {  **if** (**isA**) {  **isA** = **false**;  **keysA**[**keysASize**] = **null**;  **keysBSize** = 0;  **return keysA**;  } |
| **final** Object a = k.attachment(); |
| **if** (!k.isValid()) {  unsafe.close(unsafe.voidPromise());  **return**; } |
| **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();  **if** (!ch.isOpen()) {  *// Connection already closed - no need to handle write.* **return**;  } } |

### 4.10 Step3 reactor线程任务的执行

【总结】任务的执行分两种：定时任务和普通任务，Netty执行时就将定时任务聚合到普通任务里，然后挨个执行这些任务；并且在默认执行64个任务之后，计算下当前时间，是否超过最大允许执行时间，如果超过则直接中断，中断后执行下一次的NioEventLoop循环。



（1）task的分类和添加

·普通task；

·定时任务task；

分别存放不同队列。

|  |
| --- |
| **protected** Queue<Runnable> newTaskQueue(**int** maxPendingTasks) {  **return new** LinkedBlockingQueue<Runnable>(maxPendingTasks); } |

普通任务的添加：

|  |
| --- |
| **boolean** inEventLoop = inEventLoop(); **if** (inEventLoop) {  addTask(task); } |
| **protected void** addTask(Runnable task) {  **if** (task == **null**) {  **throw new** NullPointerException(**"task"**);  }  **if** (!offerTask(task)) {  reject(task);  } } |
| **final boolean** offerTask(Runnable task) {  **if** (isShutdown()) {  *reject*();  }  **return taskQueue**.offer(task); } |

定时任务队列添加：

|  |
| --- |
| @Override **public** <V> ScheduledFuture<V> schedule(Callable<V> callable, **long** delay, TimeUnit unit) {  ObjectUtil.*checkNotNull*(callable, **"callable"**);  ObjectUtil.*checkNotNull*(unit, **"unit"**);  **if** (delay < 0) {  **throw new** IllegalArgumentException(  String.*format*(**"delay: %d (expected: >= 0)"**, delay));  }  **return** schedule(**new** ScheduledFutureTask<V>(  **this**, callable, ScheduledFutureTask.*deadlineNanos*(unit.toNanos(delay)))); } |
| <V> ScheduledFuture<V> schedule(**final** ScheduledFutureTask<V> task) {  **if** (inEventLoop()) {  scheduledTaskQueue().add(task);  } **else** {  // 不是当前EventLoop线程，使用SingleThreadEvent单线程提交  execute(**new** Runnable() {  @Override  **public void** run() {  scheduledTaskQueue().add(task);  }  });  }   **return** task; } |
|  |

（2）任何的聚合

在执行task时，首先会将定时任务中的task聚合到普通的taskQueue里，这个taskQueue就是一个MPSCQueue（多生产单消费Queue）

|  |
| --- |
| @Override **protected void** run() {  **for** (;;) {  **try** {  **switch** (**selectStrategy**.calculateStrategy(**selectNowSupplier**, hasTasks())) {  **case** SelectStrategy.***CONTINUE***:  **continue**;  **case** SelectStrategy.***SELECT***:  select(**wakenUp**.getAndSet(**false**));**if** (**wakenUp**.get()) {  **selector**.wakeup();  }  **default**:  *// fallthrough* }   **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);  }  } } |
| **protected boolean** runAllTasks(**long** timeoutNanos) {  fetchFromScheduledTaskQueue(); |
| **private boolean** fetchFromScheduledTaskQueue() {  **long** nanoTime = AbstractScheduledEventExecutor.*nanoTime*();  Runnable scheduledTask = pollScheduledTask(nanoTime);  **while** (scheduledTask != **null**) {  // 超时任务添加到普通队列失败后，重新放回超时队列  **if** (!**taskQueue**.offer(scheduledTask)) {  *// No space left in the task queue add it back to the scheduledTaskQueue so we pick it up again.* scheduledTaskQueue().add((ScheduledFutureTask<?>) scheduledTask);  **return false**;  }  scheduledTask = pollScheduledTask(nanoTime);  }  **return true**; } |
| @Override **public int** compareTo(Delayed o) {  **if** (**this** == o) {  **return** 0;  }   ScheduledFutureTask<?> that = (ScheduledFutureTask<?>) o;  **long** d = deadlineNanos() - that.deadlineNanos();  // 先按照截止时间排序，截止时间早优先  **if** (d < 0) {  **return** -1;  } **else if** (d > 0) {  **return** 1;  // 截止时间一样，使用id排序，id小优先  } **else if** (**id** < that.**id**) {  **return** -1;  } **else if** (**id** == that.**id**) {  **throw new** Error();  } **else** {  **return** 1;  } } |
| **protected final** Runnable pollScheduledTask(**long** nanoTime) {  **assert** inEventLoop();   Queue<ScheduledFutureTask<?>> scheduledTaskQueue = **this**.**scheduledTaskQueue**;  ScheduledFutureTask<?> scheduledTask = scheduledTaskQueue == **null** ? **null** : scheduledTaskQueue.peek();  **if** (scheduledTask == **null**) {  **return null**;  }   **if** (scheduledTask.deadlineNanos() <= nanoTime) {  scheduledTaskQueue.remove();  **return** scheduledTask;  }  **return null**; } |

（3）任务的执行

接下来挨个执行任务。

|  |
| --- |
| **protected boolean** runAllTasks(**long** timeoutNanos) {  // 1 聚合超时任务队列至普通任务队列  fetchFromScheduledTaskQueue();  // 2 从普通任务队列拿一个任务  Runnable task = pollTask();  **if** (task == **null**) {  // 每次任务执行完的后续操作  afterRunningAllTasks();  **return false**;  }   **final long** deadline = ScheduledFutureTask.*nanoTime*() + timeoutNanos;  **long** runTasks = 0;  **long** lastExecutionTime;  **for** (;;) {  // 当前定时任务抛出异常后，继续执行后续的任务，而非停止  *safeExecute*(task);   runTasks ++;   *// Check timeout every 64 tasks because nanoTime() is relatively expensive.  // XXX: Hard-coded value - will make it configurable if it is really a problem.*  *// 每64个任务判断一下耗时的nanoTime()是否超过截止时间，而不是每次任务执行都判断* **if** ((runTasks & 0x3F) == 0) {  lastExecutionTime = ScheduledFutureTask.*nanoTime*();  **if** (lastExecutionTime >= deadline) {  **break**;  }  }  // 不停的出队执行任务，直到没有可消费的任务  task = pollTask();  **if** (task == **null**) {  lastExecutionTime = ScheduledFutureTask.*nanoTime*();  **break**;  }  }   afterRunningAllTasks();  **this**.**lastExecutionTime** = lastExecutionTime;  **return true**; } |
| **protected static void** safeExecute(Runnable task) {  **try** {  task.run();  } **catch** (Throwable t) {  ***logger***.warn(**"A task raised an exception. Task: {}"**, task, t);  } } |

### 4.11NioEventLoop总结

（1）NioEventLoop创建

NioEventLoop构造函数

·用户代码在创建BossGroup和WorkerGroup时，NioEventLoop被创建，默认不传参数时，会创建默认2 \* CPU 核数个NioEventLoop，每个NioEventLoop都会有一个Chooser，进行线程逻辑分配。

·而这个Chooser也会针对NioEventLoop的个数做一定的优化，NioEventLoop在创建的时候会创建一个Selector，和一个定时任务队列。

·创建Selector时Netty会通过反射的方式，用数组形式替换掉Selector中的两个HashSet数据结构。

（2）NioEventLoop启动

·execute方法：NioEventLoop在调用execute方法时启动线程，此线程时FastThreadLocal线程，启动线程后，Netty会将创建的线程，保存到成员变量，这样就能判断执行NioEventLoop的逻辑是否是本线程。

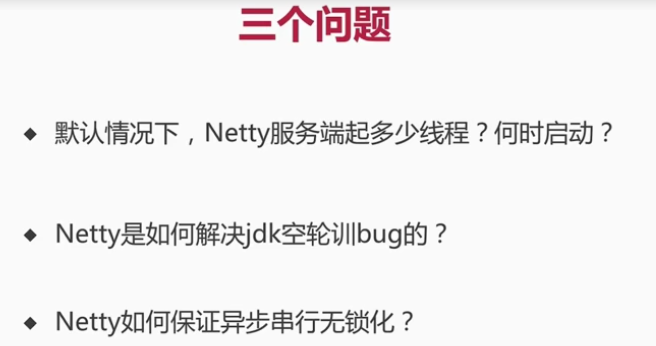
（3）NioEventLoop执行逻辑

·run方法：主要包括三个过程，

第一个过程是检查IO事件，

第二个过程处理IO事件，

第三个过程执行任务队列。

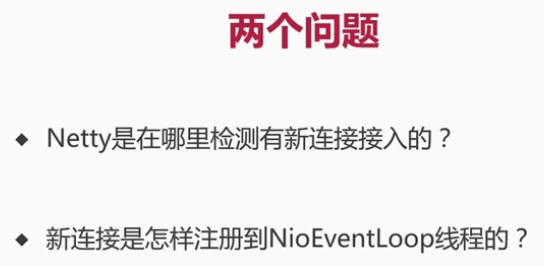


（1）2 \* CPU；doBind时注册Channel，调用Channel所属eventLoop的execute方法，添加任务队列，判断inEventLoop是否当前线程，否则startThread启动一个FastThreadLocal线程。

（2）select时截止时间没到，且任务队列为空，进行阻塞式select，即512次空轮训后，重新rebuildSelector，把key重新注册到新打开的selector上。

（3）所有操作抽象成task，存放至MPSC队列，按照截止时间顺序的执行。

## 第5节 Netty新连接接入



Step1 检测新连接；

Step2 创建NioSocketChannel；

Step3 分配线程及注册Selector；

Step4 向selector注册读事件。

### 5.1 检测新连接



|  |
| --- |
| **if** ((readyOps & (SelectionKey.***OP\_READ*** | SelectionKey.***OP\_ACCEPT***)) != 0 || readyOps == 0) {  unsafe.read();  **if** (!ch.isOpen()) {  *// Connection already closed - no need to handle write.* **return**;  } } |
|  |
| Telnet 127.0.0.1 8888 |
| @Override **public void** read() {  **assert** eventLoop().inEventLoop();  **final** ChannelConfig config = config();  **final** ChannelPipeline pipeline = pipeline();  // 通过handle控制连接接入速率，默认情况下只允许16个连接  **final** RecvByteBufAllocator.Handle allocHandle = unsafe().recvBufAllocHandle();  allocHandle.reset(config);   **boolean** closed = **false**;  Throwable exception = **null**;  **try** {  **try** {  **do** {  **int** localRead = doReadMessages(**readBuf**);  **if** (localRead == 0) {  **break**;  }  **if** (localRead < 0) {  closed = **true**;  **break**;  }   allocHandle.incMessagesRead(localRead);  } **while** (allocHandle.continueReading());  } **catch** (Throwable t) {  exception = t;  } |
|  |
| @Override **protected int** doReadMessages(List<Object> buf) **throws** Exception {  SocketChannel ch = javaChannel().accept();   **try** {  **if** (ch != **null**) {  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; } |

### 5.2 NioSocketChannel的创建



NioSocketChannel构造函数

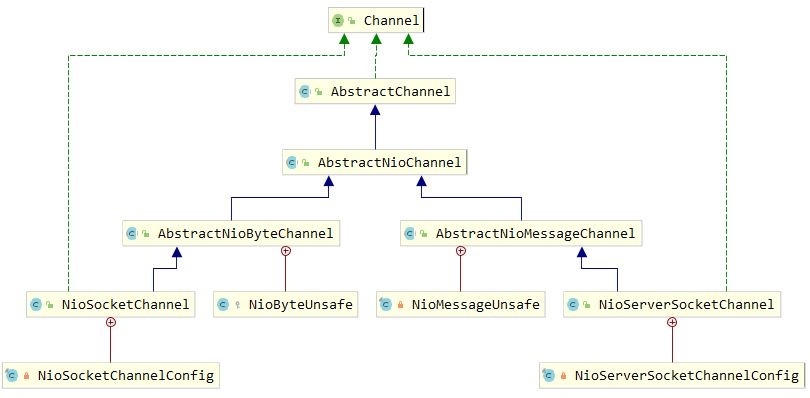
|  |
| --- |
| @Override **protected int** doReadMessages(List<Object> buf) **throws** Exception {  SocketChannel ch = javaChannel().accept();   **try** {  **if** (ch != **null**) {  // this:Netty服务端channel NioServerSocketChannel，ch：jdk accept到客户端channel  buf.add(**new** NioSocketChannel(**this**, ch));  **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; } |
| **public** NioSocketChannel(Channel parent, SocketChannel socket) {  **super**(parent, socket);  **config** = **new** NioSocketChannelConfig(**this**, socket.socket()); } |
| **protected** AbstractNioByteChannel(Channel parent, SelectableChannel ch) {  **super**(parent, ch, SelectionKey.***OP\_READ***); } |
| **protected** AbstractNioChannel(Channel parent, SelectableChannel ch, **int** readInterestOp) {  **super**(parent);  **this**.**ch** = ch;  **this**.**readInterestOp** = readInterestOp;  **try** {  // 设置非阻塞模式  ch.configureBlocking(**false**);  } **catch** (IOException e) {  **try** {  ch.close();  } **catch** (IOException e2) {  **if** (***logger***.isWarnEnabled()) {  ***logger***.warn(  **"Failed to close a partially initialized socket."**, e2);  }  }   **throw new** ChannelException(**"Failed to enter non-blocking mode."**, e);  } } |
| **protected** AbstractChannel(Channel parent) {  **this**.**parent** = parent;  // 创建关键组件：id，unsafe，pipeline  **id** = newId();  **unsafe** = newUnsafe();  **pipeline** = newChannelPipeline(); } |
| **public** DefaultSocketChannelConfig(SocketChannel channel, Socket javaSocket) {  **super**(channel);  **if** (javaSocket == **null**) {  **throw new** NullPointerException(**"javaSocket"**);  }  **this**.**javaSocket** = javaSocket;   *// Enable TCP\_NODELAY by default if possible.* **if** (PlatformDependent.*canEnableTcpNoDelayByDefault*()) {  **try** {  // 设置TCP\_NODELAY，尽量将小的数据包封装成大数据包发送  setTcpNoDelay(**true**);  } **catch** (Exception e) {  *// Ignore.* }  } } |

### 5.3 Channel的分类

（1）NioServerSocketChannel

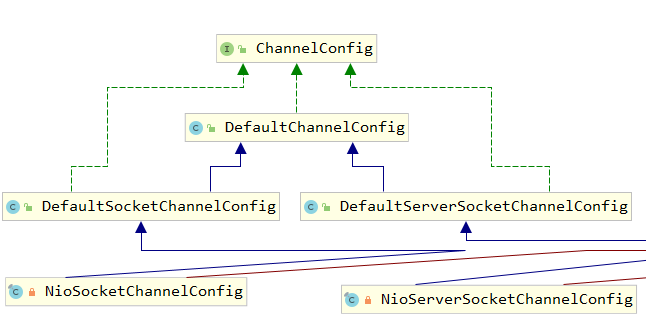
（2）NioSocketChannel

（3）unsafe



【总结】

1. 顶层接口Channel；
2. AbstractChannel实现Channel大部分功能；
3. AbstractNioChannel主要通过Selector处理NIO事件；
4. 客户端Channel和服务端Channel，区别：客户端注册read事件，服务端注册accept事件；
5. 客户端Channel和服务端Channel，底层依赖Unsafe对象，实现两种Channel底层的读写；
6. 读操作：服务端读一条连接，客户端读取数据；
7. 每种Channel都会有ChannelConfig进行绑定，每种Config实现Channel相关配置



客户端NioSocketChannelConfig

服务端NioServerSocketChannelConfig

### 5.4新连接NioEventLoop分配和Selector注册

Channel创建后，需要分配NioEventLoop线程，并注册Selector。

服务端Channel的pipeline构成



·添加childHandler；

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

·设置options和attrs；

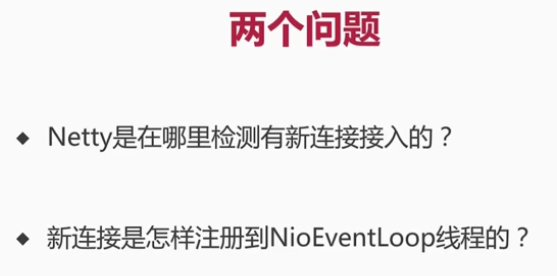
### 5.5NioSocketChannel读事件注册

（1）启动Server，并注册新连接

|  |
| --- |
|  |
|  |
|  |
|  |
| Shift + f7 ： Method to Setp Into |
|  |
| 服务端只要绑定端口，就会默认接受连接，这里就是只要连接绑定到Selector上去，都会自动读。自动读就是向Selector注册一个读事件。 |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| 服务端启动时，在初始化，并且注册完服务端Channel之后，最终也会调用这里，那这里的selectionKey也就是之前在向selector上注册时返回的一个key，然后之前注册的时候，只是绑一个0，就是不关心任何事件，注册到当前Channel对应的selector上。 |
| 当读事件进来就把它绑定到selector上，这样在下次轮训的时候，有数据进来，就可以进行数据的读写。 |
|  |
|  |
|  |
|  |

### 5.6总结





1. Boss线程的第一个过程，轮询出Accept事件；Boss线程第二个过程，通过JDK底层的channel的accept方法去创建这条连接；
2. Boss线程调用chooser的next方法，拿到一个NioEventLoop，然后将这条连接注册到这个NioEventLoop的selector上。

## 第6节Pipeline

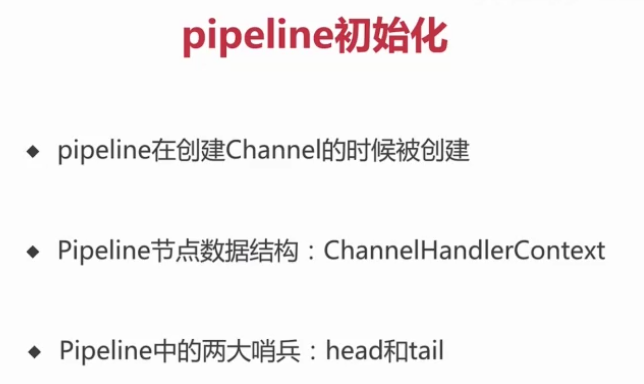
大动脉，负责Netty事件的传播



ChannelHandler类型：分为InBound、OnBound类型

1. pipeline的初始化；
2. 添加删除ChannelHandler
3. 事件个异常的传播；

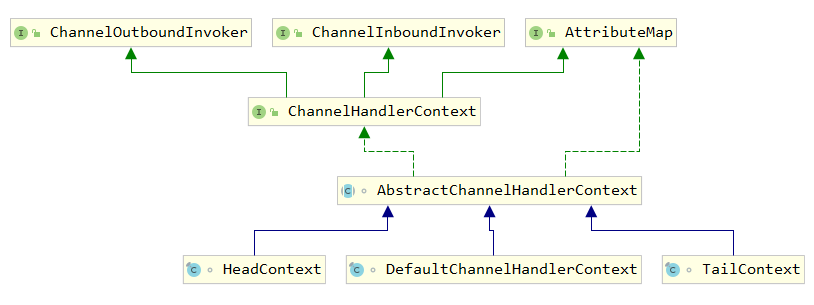
### 6.1Pipeline初始化



（1）Pipeline在创建Channel的时候被创建

|  |
| --- |
|  |
|  |
|  |
|  |

（2）Pipeline节点数据结构：ChannelHandlerContext



|  |
| --- |
|  |

（3）Pipeline两大哨兵TailContext、HeadContext

|  |
| --- |
|  |
|  |