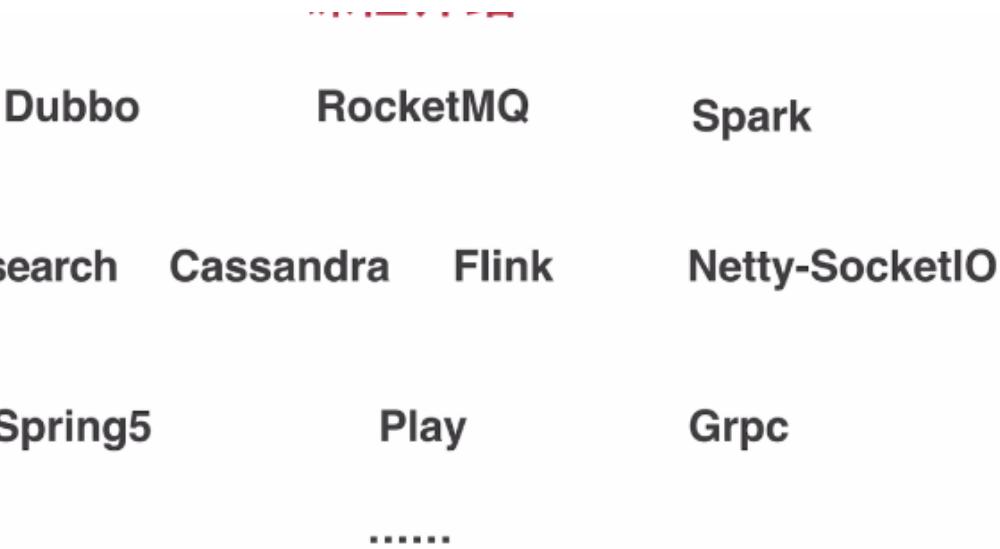


## — netty简介

### 1.1 业界使用netty的开源框架

- dubbo
  - RocketMQ
  - Spark
  - ElasticSearch
  - Cassandra 开源分布式nosql数据库
  - Flink 分布式高性能高可用的流处理框架
  - Netty-SocketIO socketIO协议的java服务端实现
  - Spring5 使用netty作为http协议框架
  - Grpc 谷歌开源的高性能Rpc框架
- 
- ```
graph TD; A[Dubbo] --- B[RocketMQ]; A --- C[Spark]; B --- D[Elasticsearch]; B --- E[Cassandra]; B --- F[Flink]; B --- G[Netty-SocketIO]; C --- H[Spring5]; C --- I[Play]; C --- J[Grpc]
```

### 1.2. Netty是什么？为什么使用netty之后，几乎不用担心性能问题

- 异步事件驱动框架，用于快速开发高性能服务端和客户端
- 封装了JDK底层BIO和NIO模型，提供高度可用的API（提供了非常多的扩展点，使API更加灵活丰富，channelHandler热插拔机制，解放了业务逻辑之外的细节问题，使业务逻辑的热添加和删除变得容易）
- 自带编解码器解决了拆包粘包问题，用户只用关心业务逻辑
- 精心设计的reactor线程模型支持高并发海量连接（为什么netty只使用了少量的线程，就能管理成千上万甚至几十万的连接）
- 自带各种协议栈让你处理任何一种通用协议都几乎不用亲自动手

### 1.3. 为什么学netty

- 各大开源框架选择netty作为底层通信框架
- 更好的使用，少走弯路
- 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 -- 可以作为第一个深入研究的开源框架

#### 1.4. 目标

- 掌握netty底层核心原理，解决各类问题，深度调优
- 给netty官方提issue
- 实现一个简易版的netty
- 开启阅读源码之旅
- 加速掌握基于netty的各类中间件

#### 1.5. 技术储备

- java基础，多线程
- TCP原理， NIO

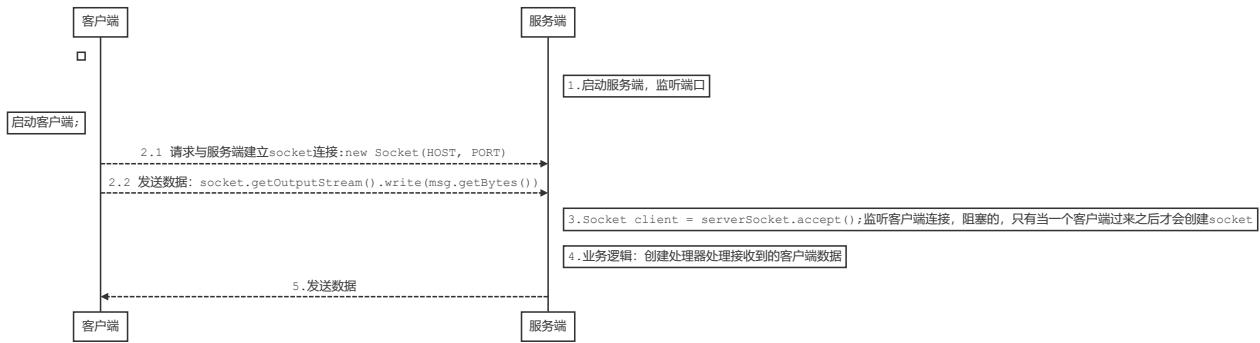
## 二 netty基本组件

包括：NioEventLoop（发动机：起了两种类型的线程），Channel（对连接的封装，数据读写），ByteBuf（数据流），Pipeline（逻辑处理链），ChannelHandler（逻辑）

# Netty基本组件

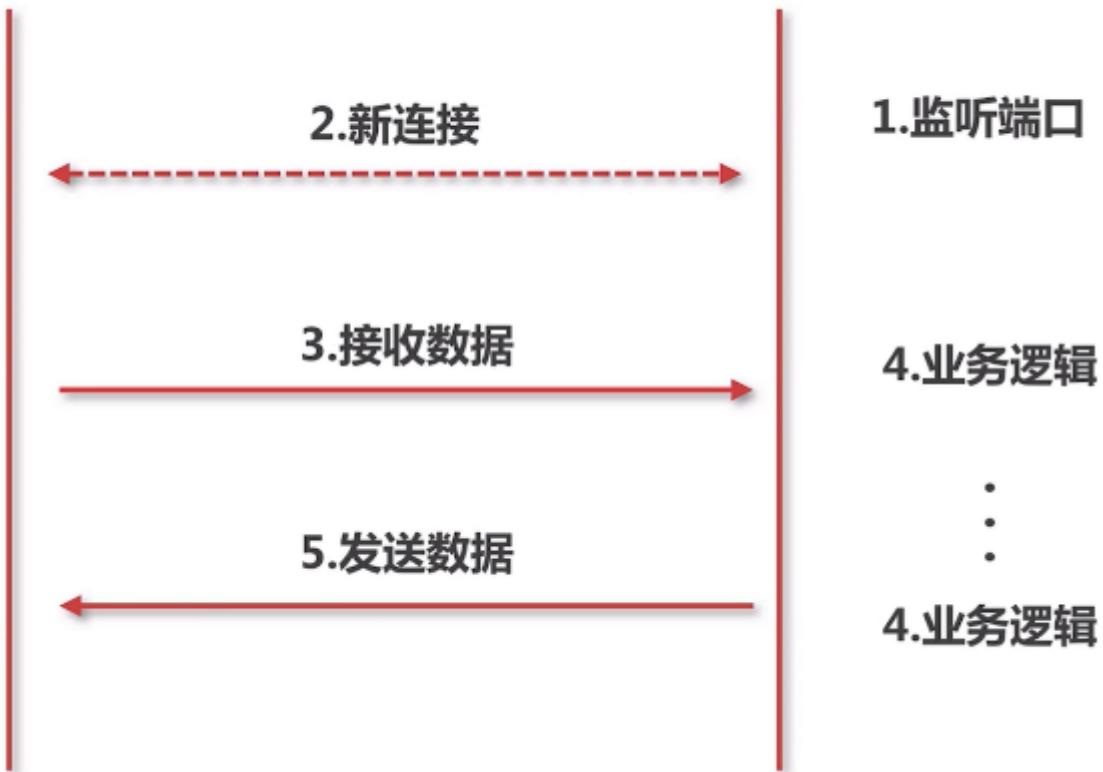


#### 2.1 不使用netty情况下，模拟传统的客户端与服务端通信



## 客户端

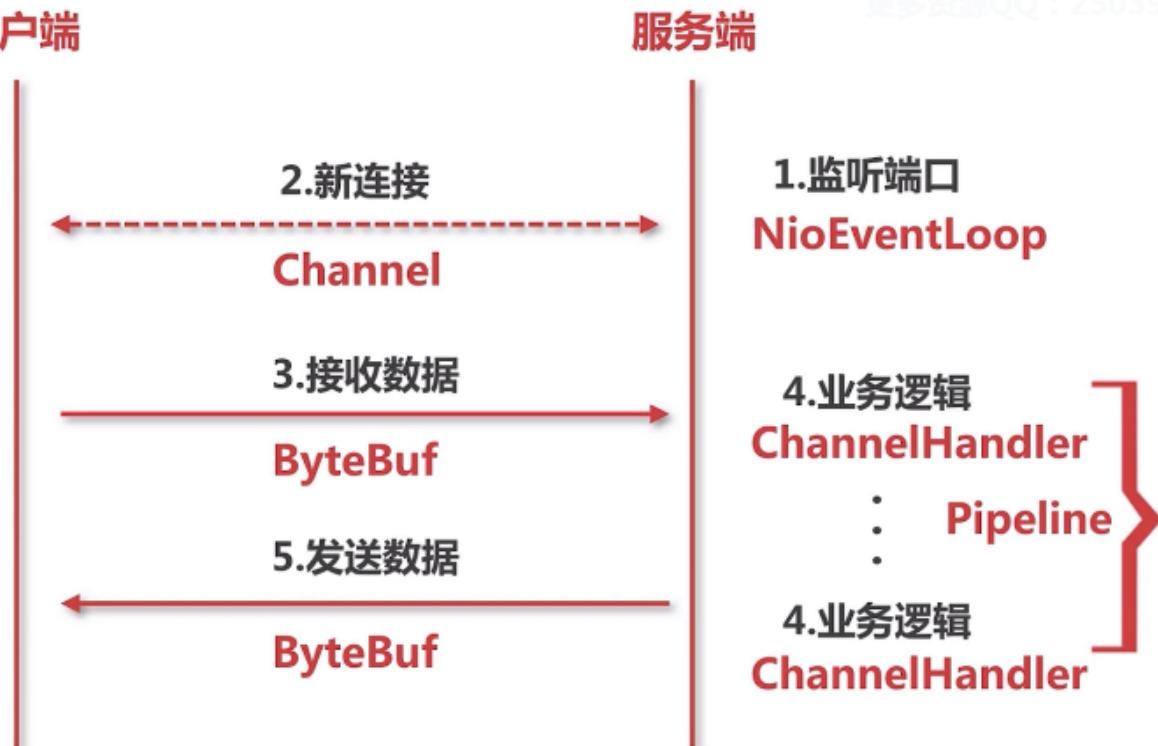
## 服务端



2.1.1 监听端口实际包含两层含义：对应两个while循环

- a) server不断的在某个端口上监听新用户的连接
- b) 新用户的连接建立完成后，在对应的端口上不断的监听新连接的数据

netty实现：



## 2.2 netty - NioEventLoop

### 对应socket编程的线程



## NioEventLoop : nio事件循环

## 2.2.1 新连接的接入

### 2.2.2 当前存在的连接上数据流的读写

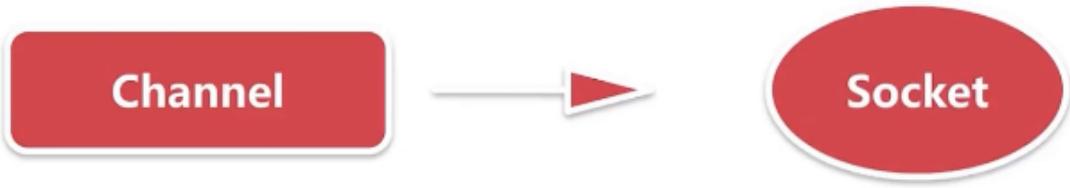
## 2.3 netty - channel

channel 定义:

- \* A nexus to a network socket or a component which is capable of I/O operations such as read, write, connect, and bind.

## 对应socket编程的socket

### 端口上监听到的新用户的连接



io.netty.channel.nio.AbstractNioMessageChannel#doReadMessages

```
socketChannel ch = javaChannel().accept();
```

java IO编程模型 -- 当作socket处理

NIO编程模型 -- socketChannel

netty -- 封装成自定义的channel

基于channel，一系列的读写都可以在这个连接上操作，其实就是对socket的抽象

#### 2.4 netty - ByteBuf



14

服务端接受用户的数据流的载体都是基于ByteBuf，封装了很多api可以与底层的连接的数据流通信

#### 2.5 netty - channelHandler

服务端处理业务逻辑的处理器



io.netty.channel.DefaultChannelPipeline#addFirst(io.netty.channel.ChannelHandler...)

通过ChannelPipeline可以动态的添加channelHandler

```

DefaultChannelPipeline
    m DefaultChannelPipeline(Channel)
    m addAfter(EventExecutorGroup, String, String, ChannelHandler): ChannelPipeline 1
    m addAfter(String, String, ChannelHandler): ChannelPipeline ↑ChannelPipeline
    m addAfter0(AbstractChannelHandlerContext, AbstractChannelHandlerContext): void
    m addBefore(EventExecutorGroup, String, String, ChannelHandler): ChannelPipeline
    m addBefore(String, String, ChannelHandler): ChannelPipeline ↑ChannelPipeline
    m addBefore0(AbstractChannelHandlerContext, AbstractChannelHandlerContext): void
    m addFirst(ChannelHandler...): ChannelPipeline ↑ChannelPipeline
    m addFirst(EventExecutorGroup, ChannelHandler...): ChannelPipeline ↑ChannelPipeline
    m addFirst(EventExecutorGroup, String, ChannelHandler): ChannelPipeline ↑ChannelPipeline
    m addFirst(String, ChannelHandler): ChannelPipeline ↑ChannelPipeline
    m addFirst0(AbstractChannelHandlerContext): void
    m addLast(ChannelHandler...): ChannelPipeline ↑ChannelPipeline
    m addLast(EventExecutorGroup, ChannelHandler...): ChannelPipeline ↑ChannelPipeline
    m addLast(EventExecutorGroup, String, ChannelHandler): ChannelPipeline ↑ChannelPipeline
    m addLast(String, ChannelHandler): ChannelPipeline ↑ChannelPipeline
    m addLast0(AbstractChannelHandlerContext): void
    m bind(SocketAddress): ChannelFuture ↑ChannelOutboundInvoker

```

```

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

```

实际生产环境下，客户端与服务端的通信的时候都很复杂，

一般都需要定义二进制的协议，对二进制协议的数据进行数据包的拆分，对不同类型的协议数据包转换成不同的java对象并作不同的处理。

netty把每一个处理过程都当作ChannelHandler。

将不同的处理过程交给不同的channelHandler处理。

用户可以自定义channelHandler。

例如：数据包分包器

## 2.6 netty - Pipeline - 逻辑链

netty什么时候将Pipeline加入到每一个客户端连接的处理过程的。

```

protected AbstractChannel(Channel parent, ChannelId id) {
    this.parent = parent;
    this.id = id;
    unsafe = newUnsafe();
    pipeline = newChannelPipeline();
}

```

Pipeline

Logic Chain

### 三. netty服务端启动

```
public final class Server {  
    public static void main(String[] args) {  
        // bossGroup 对应 socket编程中 server端 的 线程  
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);  
        // workGroup 对应 socket编程中 client端 的 线程  
        EventLoopGroup workGroup = new NioEventLoopGroup();  
  
        ServerBootstrap bootstrap = new ServerBootstrap();  
  
        bootstrap.group(bossGroup, workGroup)  
            .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();  
                }  
            });  
        try {  
            //服务端创建的入口 bind()  
            ChannelFuture channelFuture = bootstrap.bind(8888).sync();  
            channelFuture.channel().closeFuture().sync();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

3.1 思考：服务端的socket在哪里初始化？在哪里accept连接？

3.2 netty服务端启动的四个过程

3.2.1 创建服务端channel

```
ChannelFuture channelFuture = bootstrap.bind(8888).sync();
```

```
private ChannelFuture doBind(final SocketAddress localAddress) {  
    final ChannelFuture regFuture = initAndRegister();  
    final Channel channel = regFuture.channel();  
    ....  
}
```

```
final ChannelFuture initAndRegister() {  
    Channel channel = null;  
    try {  
        channel = channelFactory.newChannel(); // 创建服务端channel  
        init(channel); // 初始化服务端channel  
    } catch (Throwable t) {  
        ....  
    }  
}
```

```
@Override  
public T newChannel() {  
    try {  
        // 这里的clazz指什么? --> NioServerSocketChannel.class  
        return clazz.newInstance(); // 反射  
    } catch (Throwable t) {  
        throw new ChannelException("Unable to create Channel from class " + clazz, t);  
    }  
}
```

## 创建服务端Channel

**bind() [用户代码入口]**

**initAndRegister() [初始化并注册]**

**newChannel() [创建服务端channel]**

NioServerSocketChannel\*\*如何构造的

# 反射创建服务端Channel

**newSocket()** [通过jdk来创建底层jdk channel]

**NioServerSocketChannelConfig()** [tcp参数配置类]

**AbstractNioChannel()**

**configureBlocking(false)** [阻塞模式]

**AbstractChannel()** [创建id,unsafe,pipeline]

```
io.netty.channel.socket.nio.NioServerSocketChannel#NioServerSocketChannel()
public NioServerSocketChannel() {
    this(newSocket(DEFAULT_SELECTOR_PROVIDER));
}
```

```
io.netty.channel.socket.nio.NioServerSocketChannel#NioServerSocketChannel(java.nio.channels.ServerSocketChannel)

public NioServerSocketChannel(ServerSocketChannel channel) {
    super(null, channel, SelectionKey.OP_ACCEPT);
    config = new NioServerSocketChannelConfig(this, javaChannel().socket());
}
```

```
io.netty.channel.nio.AbstractNioChannel#AbstractNioChannel
protected AbstractNioChannel(Channel parent, SelectableChannel ch, int readInterestOp) {
    super(parent);
    this.ch = ch;
    this.readInterestOp = readInterestOp;
    try {
        ch.configureBlocking(false);
    } catch (IOException e) {
        ...
    }
}
```

```
io.netty.channel.AbstractChannel#AbstractChannel(io.netty.channel.Channel)
protected AbstractChannel(Channel parent) {
    this.parent = parent;
    id = newId();
    unsafe = newUnsafe();
    pipeline = newChannelPipeline();
}
```

### 3.2.2 初始化服务端channel

- io.netty.bootstrap.AbstractBootstrap#init() 初始化入口
  - set ChannelOptions, ChannelAttrs
  - set ChildOptions, ChildAttrs
  - config handler [配置服务端Pipeline]
  - add ServerBootstrapAcceptor[添加连接接入器]

## 初始化服务端Channel

**init() [初始化入口]**

**set ChannelOptions, ChannelAttrs**

**set ChildOptions, ChildAttrs**

**config handler [配置服务端pipeline]**

**add ServerBootstrapAcceptor [添加连接器]**

--> 保存用户自定义的属性

--> ServerBootstrapAcceptor 创建新连接接入器（一个特殊的channelHandler）

--> 将用户自定义的属性传到新连接接入器中，当accept到一个新连接，通过这几个属性对新的连接进行配置，就可以把一个新的连接绑定到一个新的线程上去。

```
@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(new Option[childOptions.size()]);
}
synchronized (childAttrs) {
    currentChildAttrs = childAttrs.entrySet().toArray(new Attr[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);
        }
        ch.eventLoop().execute(new Runnable() {
            @Override
            public void run() {
                pipeline.addLast(new ServerBootstrapAcceptor(
                    currentChildGroup, currentChildHandler, currentChildOptions,
currentChildAttrs));
            }
        });
    }
});
}

```

### 3.2.3 将channel注册到事件轮询器selector

- io.netty.channel.AbstractChannel.AbstractUnsafe#register (channel) [入口]
- this.eventLoop = eventLoop [绑定线程]
- io.netty.channel.AbstractChannel.AbstractUnsafe#register0 [实际注册]
  - io.netty.channel.AbstractChannel#doRegister [调用底层JDK底层注册]
  - io.netty.channel.DefaultChannelPipeline#invokeHandlerAddedIfNeeded [添加 channelHandler的时候触发用户回调]

- o io.netty.channel.DefaultChannelPipeline#fireChannelActive [传播channel注册成功事件到用户代码方法]

```
#ChannelFuture regFuture = 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;
    }
    // 处理所有I/O事件
    AbstractChannel.this.eventLoop = eventLoop;
    if (eventLoop.inEventLoop()) {
        register0(promise);
    } else {
        try {
            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);
        }
    }
}
```

```
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;

    // 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.
    pipeline.invokeHandlerAddedIfNecessary();

    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);
    }
}
}

```

io.netty.channel.nio.AbstractNioChannel#doRegister

```

@Override
protected void doRegister() throws Exception {
    boolean selected = false;
    for (;;) {
        try {
            //调用jdk底层的注册方法 this代表 NioServerSocketChannel.class
            //将NioServerSocketChannel作为一个attachment传到jdk底层的channel
            selectionKey = javaChannel().register(eventLoop().selector, 0, this);
            return;
        } catch (CancelledKeyException e) {
            if (!selected) {
                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;  
    }  
}  
}  
}
```

java.nio.channels.SelectableChannel#register(java.nio.channels.Selector, int, java.lang.Object)

```
* @param sel  
*      The selector with which this channel is to be registered  
* @param ops  
*      The interest set for the resulting key  
* @param att  
*      The attachment for the resulting key; may be <tt>null</tt>  
public abstract SelectionKey register(Selector sel, int ops, Object att)  
throws ClosedChannelException;
```

## 注册selector

**AbstractChannel.register(channel) [入口]**

**this.eventLoop = eventLoop [绑定线程]**

**resgiter0() [实际注册]**

**doRegister() [调用jdk底层注册]**

**invokeHandlerAddedIfNeeded()**

**fireChannelRegistered() [传播事件]**



3.2.4 端口绑定

# 端口绑定

AbstractUnsafe.bind() [入口]

doBind()

javaChannel().bind() [jdk底层绑定]

pipeline.fireChannelActive() [传播事件]

HeadContext.readIfIsAutoRead()

- io.netty.channel.AbstractChannel.AbstractUnsafe#bind [入口]
- io.netty.channel.AbstractChannel#doBind

```
@Override
public final void bind(final SocketAddress localAddress, final ChannelPromise promise) {
    assertEventLoop();

    if (!promise.setUncancellable() || !ensureOpen(promise)) {
        return;
    }
    if (Boolean.TRUE.equals(config().getOption(ChannelOption.SO_BROADCAST)) &&
        localAddress instanceof InetSocketAddress &&
        !((InetSocketAddress) localAddress).getAddress().isAnyLocalAddress() &&
        !PlatformDependent.isWindows() && !PlatformDependent.isRoot()) {
        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.");
    }
    //端口绑定之前是false, doBind之后变为true
    boolean wasActive = isActive();
    try {

        doBind(localAddress);

    } catch (Throwable t) {
        safeSetFailure(promise, t);
        closeIfClosed();
        return;
    }
}
```

```
}

//端口绑定之前不是active, 绑定之后编程active
if (!wasActive && isActive()) {
    invokeLater(new Runnable() {
        @Override
        public void run() {
            //传播事件
            pipeline.fireChannelActive();
        }
    });
}

safeSetSuccess(promise);
}
```

io.netty.channel.socket.nio.NioServerSocketChannel#doBind

```
@Override
protected void doBind(SocketAddress localAddress) throws Exception {
    if (PlatformDependent.javaVersion() >= 7) {
        //调用java底层api
        javaChannel().bind(localAddress, config.getBacklog());
    } else {
        javaChannel().socket().bind(localAddress, config.getBacklog());
    }
}
```

io.netty.channel.DefaultChannelPipelineHandlerContext#channelActive

```
@Override
public void channelActive(ChannelHandlerContext ctx) throws Exception {
    ctx.fireChannelActive();

    readIfIsAutoRead();
}
```

io.netty.channel.DefaultChannelPipelineHandlerContext#readIfIsAutoRead

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

io.netty.channel.AbstractChannel#read

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

io.netty.channel.AbstractChannelHandlerContext#read

```
@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;  
}
```

io.netty.channel.DefaultChannelPipeline.HeadContext#read

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

io.netty.channel.AbstractChannel.AbstractUnsafe#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());
    }
}

```

io.netty.channel.nio.AbstractNioChannel#doBeginRead

```

@Override
protected void doBeginRead() throws Exception {
    // Channel.read() or ChannelHandlerContext.read() was called
    // channel注册到selector上后返回的key, key对应channel
    final SelectionKey selectionKey = this.selectionKey;
    if (!selectionKey.isValid()) {
        return;
    }

    readPending = true;

    //Retrieves this key's interest set
    final int interestOps = selectionKey.interestOps();
    //与运算
    if ((interestOps & readInterestOp) == 0) {
        //将2者进行或运算以后重新注册到selectionKey上 即在之前事件的基础上再增加一个事件
        //readInterestOp 其实是NioServerSocketChannel的构造函数传进来的SelectionKey.OP_ACCEPT
        //是个accept事件
        selectionKey.interestOps(interestOps | readInterestOp);
    }
}

```

小结：

---》 端口绑定bind

---》 触发active事件

---》 服务端channel doBeginRead方法, 向selector注册accept事件, 这样netty就可以接收新的连接

tip:

与(&)运算： 同为 1 才为1

|        |     |
|--------|-----|
| 5 二进制  | 101 |
| 3 二进制  | 011 |
| 结果 001 |     |

或(|)运算：有一个为1, 则为1

5 二进制 101

3 二进制 011

结果 111

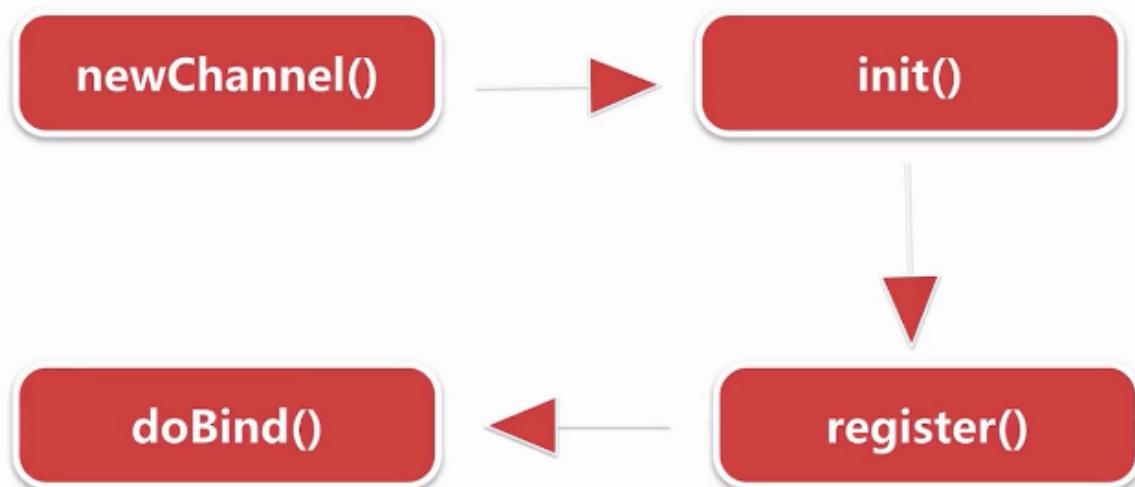
异或(^)运算：不相同则为 1

5 二进制 101

3 二进制 011

结果 110

## 服务端启动核心路径总结



## 四、NioEventLoop

### 4.1 思考三个问题

- 默认情况下，Netty服务端起多少线程？何时启动？
  - 2\*cpu 调用execute方法时，判断当前线程是EventLoop线程，若是说明线程已启动，若是外部线程，则会调用startThread方法，判断当前线程是否启动，没有则启动当前线程。
- Netty如何解决JDK空轮询bug，避免cpu飙高的？  
超过512次，重新创建selector，并把原selector的所有的key移交到新的select or
- Netty如何保证异步串行无锁化？  
netty通过inEventLoop方法判断是外部线程，将所有操作封装成一个task丢到MpscQueue中，挨个执行。
  - 拿到客户端的一个channel，不需要对这个channel同步就可以进行多线程并发读写。
  - channelHandler中的所有操作都是线程安全的，不需要进行同步

## 4.2 NioEventLoop创建

# NioEventLoop创建

**new NioEventLoopGroup() [线程组， 默认2\*cpu]**

**new ThreadPerTaskExecutor() [线程创建器]**

**for(){ newChild() } [构造NioEventLoop]**

**chooserFactory.newChooser() [线程选择器]**

- new NioEventLoopGroup(1) [线程组， 默认 2\*CPU]：若不传构造参数， 默认创建2倍cpu核心数的 NioEventLoopGroup

```
protected MultithreadEventLoopGroup(int nThreads, Executor executor, Object... args) {
    super(nThreads == 0 ? DEFAULT_EVENT_LOOP_THREADS : nThreads, executor, args);
}
```

- new ThreadPerTaskExecutor(newDefaultThreadFactory()) [线程创建器]：负责创建NioEventLoop底层对应的线程
  - 每次执行任务都会创建一个线程：netty自己封装的FastThreadLocalThread，并非原生的线程。
  - NioEventLoop线程命名规则 nioEventLoop-1-xx
- for(){ new Child() } [构造NioEventLoop]
  - 保存线程执行器 ThreadPerTaskExecutor
  - 创建一个MpscQueue

io.netty.channel.nio.NioEventLoopGroup#newChild

```
@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]);  
}
```

```
protected SingleThreadEventLoop(EventLoopGroup parent, Executor executor,  
                                boolean addTaskWakesUp, int maxPendingTasks,  
                                RejectedExecutionHandler rejectedExecutionHandler)  
{  
    super(parent, executor, addTaskWakesUp, maxPendingTasks,  
          rejectedExecutionHandler);  
    tailTasks = newTaskQueue(maxPendingTasks);  
}
```

io.netty.channel.nio.NioEventLoop#newTaskQueue

```
@Override  
protected Queue<Runnable> newTaskQueue(int maxPendingTasks) {  
    // This event loop never calls takeTask()  
    //Mpsc Multiply producer (外部线程) single consumer (NioEventLoop线程)  
    return PlatformDependent.newMpscQueue(maxPendingTasks);  
}
```

```
/**  
 * Create a new {@link Queue} which is safe to use for multiple producers  
 * (different threads) and a single  
 * consumer (one thread!).  
 */  
public static <T> Queue<T> newMpscQueue(final int maxCapacity) {  
    return Mpsc.newMpscQueue(maxCapacity);  
}
```

- 创建一个selector

```
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;
```

```
//一个selector和一个NioEventLoop绑定  
selector = openSelector();  
selectStrategy = strategy;  
}
```

- chooserFactory.newChooser(children) [线程选择器]: 为每一个新连接, 分配NioEventLoop线程

io.netty.util.concurrent.MultithreadEventExecutorGroup#next

第1个连接进来的时候选择第1个nioEventLoop进行绑定

.....

第n个连接进来的时候选择第n个nioEventLoop进行绑定

第n+1个连接进来的时候选择第1个nioEventLoop进行绑定, 循环进行

## NioEventLoopGroup.next()

NioEventLoop[]



**netty经过优化:** 与运算实现循环取数组下标, 要比取模运算高效的多, 因为在计算机底层, 与运算是二进制的运算。

# chooserFactory.newChooser()

isPowerOfTwo() [判断是否是2的幂，如2、4、8、16]

PowerOfTwoEventExecutorChooser [优化]

index++ & (length-1)

GenericEventExecutorChooser [普通]

abs(index++ % length)

```
public EventExecutorChooser newChooser(EventExecutor[] executors) {  
    if (isPowerOfTwo(executors.length)) {  
        return new PowerOfTwoEventExecutorChooser(executors);  
    } else {  
        return new GenericEventExecutorChooser(executors);  
    }  
}
```

io.netty.util.concurrent.DefaultEventExecutorChooserFactory.GenericEventExecutorChooser#next

```
@Override  
public EventExecutor next() {  
    return executors[Math.abs(idx.getAndIncrement() % executors.length)];  
}
```

io.netty.util.concurrent.DefaultEventExecutorChooserFactory.PowerOfTwoEventExecutorChooser#next

```
public EventExecutor next() {  
    return executors[idx.getAndIncrement() & executors.length - 1];  
}
```

# PowerOfTwoEventExecutorChooser

**idx.getAndIncrement() & executors.length - 1**

|                             |                    |
|-----------------------------|--------------------|
| <b>idx</b>                  | <b>1 1 1 0 1 0</b> |
| &                           |                    |
| <b>executors.length - 1</b> | <b>1 1 1 1</b>     |
| <b>result</b>               | <b>1 0 1 0</b>     |

## 4.3 NioEventLoop启动流程

NioEventLoop启动两大触发器：

- 服务端启动绑定端口
  - 服务端将具体绑定端口的操作封装成一个task，调用NioEventLoop的execute方法
  - netty判断调用execute方法的线程是否是Nio线程，若不是，调用startThread()方法尝试创建线程。
  - 通过线程执行器 ThreadPerTaskExecutor创建nio线程 - FastThreadLocalThread
  - NioEventLoop对象会将创建的线程保存，目的是为了：判断后续对NioEventLoop相关的执行线程是否是本身，若不是，就封装成一个task，扔到一个taskQueue中串行执行，保证线程安全
  - 调用驱动NioEventLoop运转的核心方法：run()

# NioEventLoop启动

bind() —> execute(task) [入口]

startThread() -> doStartThread() [创建线程]

ThreadPerTaskExecutor.execute()

thread = Thread.currentThread()

NioEventLoop.run() [启动]

```
io.netty.bootstrap.AbstractBootstrap#doBind0
private static void doBind0(
    .....
    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());
            }
        }
    });
}
```

```
io.netty.util.concurrent.SingleThreadEventExecutor#execute
@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);
        }
    }
```

```
io.netty.util.concurrent.AbstractEventExecutor#inEventLoop
判断线程是否是EventLoop线程
@Override
public boolean inEventLoop() {
    return inEventLoop(Thread.currentThread());
}
```

```
io.netty.util.concurrent.SingleThreadEventExecutor#inEventLoop
io.netty.util.concurrent.SingleThreadEventExecutor#thread

private volatile Thread thread;
@Override
public boolean inEventLoop(Thread thread) {
    return thread == this.thread;
}
```

```
private void startThread() {
    if (STATE_UPDATER.get(this) == ST_NOT_STARTED) {
        if (STATE_UPDATER.compareAndSet(this, ST_NOT_STARTED, ST_STARTED)) {
            doStartThread();
        }
    }
}
```

```
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;
            } catch (Throwable t) {
                logger.warn("Unexpected exception from an event executor: ", t);
            } finally {
                for (;;) {
                    int oldstate =
STATE_UPDATER.get(SingleThreadEventExecutor.this);
```

```
        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);  
        }  
    }  
}  
});  
}
```

## 4.4 NioEventLoop执行逻辑 (底层干了哪些事情, 如何保证高效运转)

## SingleThreadEventExecutor.this.run()

1207365

## NioEventLoop.run()

run() -> for (;;) {

select() [检查是否有io事件]

processSelectedKeys() [处理io事件]

runAllTasks() [处理异步任务队列]

NioEventLoop 执行逻辑：

for 循环体做三件事情：

- 》 调用select方法轮询注册到selector上的连接的i/o事件
- 》 调用processSelectedKeys()处理轮询出来的i/o事件
- 》 调用runAllTasks()方法处理外部线程扔到taskQueue中的任务

```
@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;
//默认50, 处理i/o事件和运行任务时间是1 : 1
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);
}
}
```

- 检测i/o事件，select方法执行逻辑

- deadline及任务穿插逻辑处理
    - select操作进行deadline处理，判断如果当前有任务在taskQueue里面就终止本次select操作
  - 阻塞式select
    - 如果没有到截止时间并且taskQueue没有任务，就进行阻塞式select操作
  - 避免jdk空轮询bug
    - 阻塞式select操作结束之后，判断这次select操作是否真的阻塞这么长时间，如果没有阻塞这么长时间，则表示可能触发了jdk nio空轮询的bug，接下来netty判断触发空轮询次数是否达到一定的阈值（512），如果达到阈值，就通过替换原来select操作的方式，巧妙的避开了空轮询的bug

## io.netty.channel.nio.NioEventLoop#select

```

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) {
                if (selectCnt == 0) {
                    selector.selectNow();
                    selectCnt = 1;
                }
                break;
            }
            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()) {
                break;
            }
            if (Thread.interrupted()) {
                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) {
                logger.warn(
                    "Selector.select() returned prematurely {} times in a row; rebuilding Selector {}.",
                    selectCnt, selector);
            }
        }
    } catch (IOException e) {
        logger.error("Exception in Selector.select()", e);
    }
}

```

```

        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);
    }
}
}
}

```

io.netty.channel.nio.NioEventLoop#rebuildSelector

```

public void rebuildSelector() {
    if (!inEventLoop()) {
        execute(new Runnable() {
            @Override
            public void run() {
                rebuildSelector();
            }
        });
        return;
    }

    final Selector oldSelector = selector;
    final Selector newSelector;

    if (oldSelector == null) {
        return;
    }

    try {
        newSelector = openSelector();
    } catch (Exception e) {
        logger.warn("Failed to create a new Selector.", e);
        return;
    }
}

```

```

        // Register all channels to the new Selector.
        int nChannels = 0;
        for (;;) {
            try {
                for (SelectionKey key: oldSelector.keys()) {
                    Object a = key.attachment();
                    try {
                        if (!key.isValid() || key.channel().keyFor(newSelector)
                            != null) {
                            continue;
                        }

                        int interestOps = key.interestOps();
                        key.cancel();
                        SelectionKey newKey =
                            key.channel().register(newSelector, interestOps, a);
                        if (a instanceof AbstractNioChannel) {
                            // Update SelectionKey
                            ((AbstractNioChannel) a).selectionKey = newKey;
                        }
                        nChannels++;
                    } catch (Exception e) {
                        logger.warn("Failed to re-register a Channel to the new
                        Selector.", e);
                        if (a instanceof AbstractNioChannel) {
                            AbstractNioChannel ch = (AbstractNiochannel) a;
                            ch.unsafe().close(ch.unsafe().voidPromise());
                        } else {
                            @SuppressWarnings("unchecked")
                            NioTask<SelectableChannel> task =
                                (NioTask<SelectableChannel>) a;
                            invokeChannelUnregistered(task, key, e);
                        }
                    }
                }
            } catch (ConcurrentModificationException e) {
                // Probably due to concurrent modification of the key set.
                continue;
            }

            break;
        }

        selector = newSelector;

        try {
            // time to close the old selector as everything else is registered
            // to the new one
            oldSelector.close();
        } catch (Throwable t) {
            if (logger.iswarnEnabled()) {
                logger.warn("Failed to close the old Selector.", t);
            }
        }
    }
}

```

```

        }
    }

    logger.info("Migrated " + nChannels + " channel(s) to the new
selector.");
}

```

io.netty.channel.nio.NioEventLoop#openSelector

```

private Selector openSelector() {
    final Selector selector;
    try {
        //调用jdkApi创建selector
        selector = provider.openSelector();
    } catch (IOException e) {
        throw new ChannelException("failed to open a new selector", e);
    }
    //如果不需要优化，直接返回原生selector
    if (DISABLE_KEYSET_OPTIMIZATION) {
        return selector;
    }
    //SelectedSelectionKeySet 底层是用数组 + keySize的方式实现的
    final SelectedSelectionKeySet selectedKeySet = new
    SelectedSelectionKeySet();

    Object maybeSelectorImplClass = AccessController.doPrivileged(new
PrivilegedAction<Object>() {
        @Override
        public Object run() {
            try {
                return Class.forName(
                    "sun.nio.ch.SelectorImpl",
                    false,
                    PlatformDependent.getSystemClassLoader());
            } catch (ClassNotFoundException e) {
                return e;
            } catch (SecurityException e) {
                return e;
            }
        }
    });
}

if (!(maybeSelectorImplClass instanceof Class) ||
    // ensure the current selector implementation is what we can
instrument.
    !((Class<?>) maybeSelectorImplClass).isAssignableFrom(selector.getClass())) {
    if (maybeSelectorImplClass instanceof Exception) {
        Exception e = (Exception) maybeSelectorImplClass;
        logger.trace("Failed to instrument a special java.util.Set
into: {}", selector, e);
    }
    return selector;
}

```

```

    }

    final Class<?> selectorImplClass = (Class<?>) maybeSelectorImplClass;
    Object maybeException = AccessController.doPrivileged(new
    PrivilegedAction<Object>() {
        @Override
        public Object run() {
            try {
                Field selectedKeysField =
selectorImplClass.getDeclaredField("selectedKeys");
                Field publicSelectedKeysField =
selectorImplClass.getDeclaredField("publicSelectedKeys");

                selectedKeysField.setAccessible(true);
                publicSelectedKeysField.setAccessible(true);

                selectedKeysField.set(selector, selectedKeySet);
                publicSelectedKeysField.set(selector, selectedKeySet);
                return null;
            } catch (NoSuchFieldException e) {
                return e;
            } catch (IllegalAccessException e) {
                return e;
            } catch (RuntimeException e) {
                // JDK 9 can throw an inaccessible object exception here;
                since Netty compiles
                    // against JDK 7 and this exception was only added in JDK
                9, we have to weakly
                        // check the type
                if
                    ("java.lang.reflect.InaccessibleObjectException".equals(e.getClass().getNam
e())))
                {
                    return e;
                } else {
                    throw e;
                }
            }
        }
    });
    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;
}

```

java.lang.Class#isAssignableFrom 判断是否是一个类的实现

## ■ 处理i/o事件，processSelectedKeys执行逻辑

select操作每次都会把已经就绪状态的i/o事件，放到底层一个HashSet的数据结构中。  
netty默认情况下，会通过反射将select底层的HashSet转换成数组的方式进行优化，  
在处理每一个Keyset的时候，都会拿到对应的一个attachment，这个attachment就是  
向selector注册i/o事件的时候绑定的经过netty封装之后的channel。

- selected keySet优化
  - 用数组替换select HashSet的实现，做到add方法时间复杂度为o(1)
- processSelectedKeysOptimized()
  - 真正处理I/O事件

```
private void processSelectedKeysOptimized(SelectionKey[] selectedKeys) {  
    for (int i = 0;; i++) {  
        final SelectionKey k = selectedKeys[i];  
        if (k == null) {  
            break;  
        }  
        // null out entry in the array to allow to have it GC'ed once the  
        Channel close  
        // See https://github.com/netty/netty/issues/2363  
        selectedKeys[i] = null;  
  
        final Object a = k.attachment();  
  
        if (a instanceof AbstractNioChannel) {  
            processSelectedKey(k, (AbstractNioChannel) a);  
        } else {  
            @SuppressWarnings("unchecked")  
            NioTask<SelectableChannel> task = (NioTask<SelectableChannel>)  
a;  
            processSelectedKey(k, task);  
        }  
  
        if (needsToSelectAgain) {  
            // null out entries in the array to allow to have it GC'ed once  
            the Channel close  
            // See https://github.com/netty/netty/issues/2363  
            for (;;) {  
                i++;  
                if (selectedKeys[i] == null) {  
                    break;  
                }  
                selectedKeys[i] = null;  
            }  
  
            selectAgain();  
            // Need to flip the optimized selectedKeys to get the right  
            reference to the array  
            // and reset the index to -1 which will then set to 0 on the  
            for loop  
            // to start over again.
```

```

        //
        // See https://github.com/netty/netty/issues/1523
        selectedKeys = this.selectedKeys.flip();
        i = -1;
    }
}
}

```

io.netty.channel.nio.NioEventLoop#processSelectedKey(java.nio.channels.SelectionKey,  
io.netty.channel.nio.AbstractNioChannel)

```

private void processSelectedKey(SelectionKey k, AbstractNioChannel ch) {
    final AbstractNioChannel.NioUnsafe unsafe = ch.unsafe();
    if (!k.isValid()) {
        final EventLoop eventLoop;
        try {
            eventLoop = ch.eventLoop();
        } catch (Throwable ignored) {
            // If the channel implementation throws an exception because there is
            // no event loop, we ignore this
            // because we are only trying to determine if ch is registered to this
            // event loop and thus has authority
            // to close ch.
            return;
        }
        // only close ch if ch is still registered to this EventLoop. ch could have
        // deregistered from the event loop
        // and thus the SelectionKey could be cancelled as part of the
        // deregistration process, but the channel is
        // still healthy and should not be closed.
        // See https://github.com/netty/netty/issues/5125
        if (eventLoop != this || eventLoop == null) {
            return;
        }
        // close the channel if the key is not valid anymore
        unsafe.close(unsafe_voidPromise());
        return;
    }

    try {
        //读取事件
        int readyops = k.readyOps();
        // We first need to call finishConnect() before try to trigger a read(...) or write(...) as otherwise
        // the NIO JDK channel implementation may throw a NotYetConnectedException.
        if ((readyOps & SelectionKey.OP_CONNECT) != 0) {
            // remove OP_CONNECT as otherwise Selector.select(..) will always
            // return without blocking
            // See https://github.com/netty/netty/issues/924
            int ops = k.interestOps();
            ops &= ~SelectionKey.OP_CONNECT;
            k.interestOps(ops);
        }
    }
}

```

```

        unsafe.finishConnect();
    }
    // Process OP_WRITE first as we may be able to write some queued buffers
    // and so free memory.
    if ((readyOps & SelectionKey.OP_WRITE) != 0) {
        // Call forceFlush which will also take care of clear the OP_WRITE once
        // there is nothing left to write
        ch.unsafe().forceFlush();
    }
    // Also check for readops of 0 to workaround possible JDK bug which may
    // otherwise lead
    // to a spin loop
    if ((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT)) != 0 || readyOps == 0) {
        unsafe.read();
        if (!ch.isOpen()) {
            // Connection already closed - no need to handle write.
            return;
        }
    }
} catch (CancelledKeyException ignored) {
    unsafe.close(unsafe_voidPromise());
}
}

```

- **runAllTask()执行逻辑**

任务分为两种，普通任务和定时任务，netty执行这些任务的时候，首先会将定时任务聚合到普通任务队列中，再挨个执行这些任务，并且每执行64个任务之后，计算当前执行时间是否超过最大允许执行时间，如果超过，就直接中断，中断之后就执行下一次nioEventLoop的循环

- task的分类和添加
- 普通任务队列 MpscQueue
  - 定时任务队列

io.netty.util.concurrent.AbstractScheduledEventExecutor#schedule(java.lang.Runnable, long, java.util.concurrent.TimeUnit)

```

public ScheduledFuture<?> schedule(Runnable command, Long delay, TimeUnit unit) {
    ObjectUtil.checkNotNull(command, "command");
    ObjectUtil.checkNotNull(unit, "unit");
    if (delay < 0) {
        throw new IllegalArgumentException(
            String.format("delay: %d (expected: >= 0)", delay));
    }
    return schedule(new ScheduledFutureTask<Void>(
        this, command, null,
        ScheduledFutureTask.deadlineNanos(unit.toNanos(delay))));
}

.....
<V> ScheduledFuture<V> schedule(final ScheduledFutureTask<V> task) {

```

```

if (inEventLoop()) {
    scheduledTaskQueue().add(task);
} else {
    //scheduledTaskQueue 是一个普通的PriorityQueue，非线程安全的
    //为了保证线程安全，将添加定时任务的操作也当作一个普通的task，来保证所有的
    //定时任务的操作都是在nioEventLoop中实现的
    execute(new Runnable() {
        @Override
        public void run() {
            scheduledTaskQueue().add(task);
        }
    });
}
return task;
}

```

#### ■ 任务的聚合

io.netty.util.concurrent.SingleThreadEventExecutor#fetchFromScheduledTaskQueue

定时任务队列排队机制

io.netty.util.concurrent.ScheduledFutureTask#compareTo

```

public int compareTo(Delayed o) {
    if (this == o) {
        return 0;
    }

    ScheduledFutureTask<?> that = (ScheduledFutureTask<?>) o;
    long d = deadlineNanos() - that.deadlineNanos();
    if (d < 0) {
        return -1;
    } else if (d > 0) {
        return 1;
    } else if (id < that.id) {
        return -1;
    } else if (id == that.id) {
        throw new Error();
    } else {
        return 1;
    }
}

```

#### ■ reactor线程任务的执行

io.netty.util.concurrent.SingleThreadEventExecutor#runAllTasks(long)

```

protected boolean runAllTasks(long timeoutNanos) {
    fetchFromScheduledTaskQueue();
    Runnable task = pollTask();
    if (task == null) {

```

```

        afterRunningAllTasks();
        return false;
    }

    final long deadline = ScheduledFutureTask.nanoTime() +
timeoutNanos;
    long runTasks = 0;
    long lastExecutionTime;
    for (;;) {
        safeExecute(task);

        runTasks++;

        // Check timeout every 64 tasks because nanoTime() is
        // relatively expensive. 耗时的
        // XXX: Hard-coded value - will make it configurable if it
        // is really a problem.
        if ((runTasks & 0x3F) == 0) {
            lastExecutionTime = ScheduledFutureTask.nanoTime();
            if (lastExecutionTime >= deadline) {
                break;
            }
        }

        task = pollTask();
        if (task == null) {
            lastExecutionTime = ScheduledFutureTask.nanoTime();
            break;
        }
    }

    afterRunningAllTasks();
    this.lastExecutionTime = lastExecutionTime;
    return true;
}

```

- 新连接接入通过chooser绑定一个NioEventLoop

## 五. netty-新连接接入

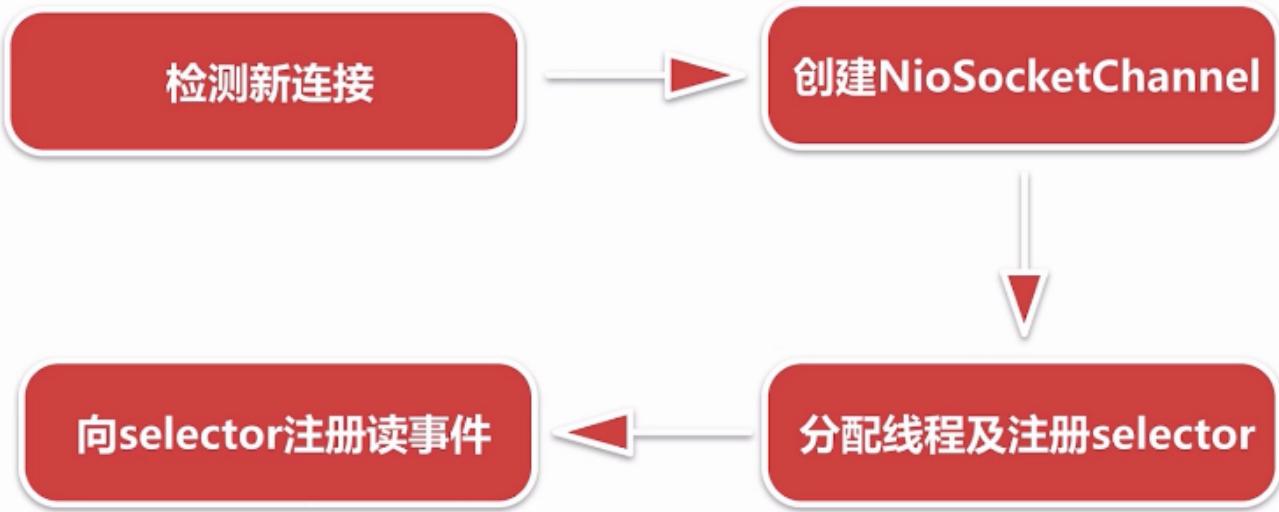
### 5.1 netty 新连接接入概述及思考问题

1) netty是在哪里检测有新连接接入的

2) 新连接是怎样注册到NioEventLoop线程的

NIO模型的多路复用，多个连接复用一个线程，对netty而言，就是NioEventLoop

# Netty新连接接入处理逻辑



## 5.2.1 检测新连接

新连接通过服务端channel绑定的selector轮询出accept事件（即I/O事件）

## 5.2.2 创建NioSocketChannel

基于JDK nio的channel创建一个netty的nioSocketChannel，也就是客户端channel

## 5.2.3 分配线程及注册selector

netty给客户端channel分配NioEventLoop并把这条channel注册到NioEventLoop对应的selector上，至此这条channel后续相关的读写都由此NioEventLoop管理

## 5.2.4 向selector注册读事件

注册的过程和服务端启动注册的accept事件复用同一段逻辑

# 检测新连接

processSelectedKey(key, channel) [入口]

NioMessageUnsafe.read()

doReadMessages() [while循环]

javaChannel().accept()

断点调试：启动服务端 ---> telnet 127.0.0.1 8888方式创建新的连接

小结：

在服务端channel的NioEventLoop run () 的 第二个过程：

- > NioEventLoop#processSelectedKey(SelectionKey, AbstractNioChannel) 检测出accept事件之后，
- > 通过jdk的accept方法创建jdk的channel，
- > 并包装成netty自定义的channel，
- > List readBuf 临时存放channnel，
- > 此过程中通过Handle对象控制连接接入的速率， 默认情况下一次性读取16个连接

io.netty.channel.nio.AbstractNioMessageChannel.NioMessageUnsafe#read

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

int size = readBuf.size();
for (int i = 0; i < size; i++) {
    readPending = false;
    pipeline.fireChannelRead(readBuf.get(i));
}
readBuf.clear();
allocHandle.readComplete();
pipeline.fireChannelReadComplete();

if (exception != null) {
    closed = closeOnReadError(exception);

    pipeline.fireExceptionCaught(exception);
}

if (closed) {
    inputshutdown = true;
    if (isOpen()) {
        close(voidPromise());
    }
}
} finally {
    // See https://github.com/netty/netty/issues/2254
    if (!readPending && !config.isAutoRead()) {
        removeReadOp();
    }
}
}
}
```

# 创建NioSocketChannel

`new NioSocketChannel(parent, ch) [入口]`

`AbstractNioByteChannel(p, ch, op_read)`

`configureBlocking(false) & save op`

`create id, unsafe, pipeline`

`new NioSocketChannelConfig()`

`setTcpNoDelay(true) 禁止Nagle算法`

客户端channel创建完成之后，将服务端channel和客户端channel作为参数传递到NioSocketChannel的构造函数中，接下来进行一系列的创建过程。

```
// this 代表创建客户端channel的服务端nioServerSocketChannel (通过反射创建) ch 代表jdk accept创建的channel
//new NioSocketChannel 构造出来的netty的客户端channel
new NioSocketChannel(this, ch);
...
public NioSocketChannel(Channel parent, SocketChannel socket) {
    super(parent, socket);
    config = new NioSocketChannelConfig(this, socket.socket());
}
```

NioSocketChannel的构造函数是入口。做两件事情：

- 逐层调用父类的构造函数
  - 配置此channel为非阻塞，将感兴趣的读事件OP\_READ,保存到成员变量方便后续注册到selector上

```
protected AbstractNioChannel(Channel parent, SelectableChannel ch, int
readInterestOp) {
    super(parent);
    this.ch = ch;
    this.readInterestOp = readInterestOp;
    try {
        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);
    }
}

```

- 创建和此channel相关的一些组件 id 作为channel的唯一标识 unsafe作为底层数据读写 pipeline 作为业务逻辑的载体。

```

protected AbstractChannel(Channel parent) {
    this.parent = parent;
    id = newId();
    unsafe = newUnsafe();
    pipeline = newChannelPipeline();
}

```

- 创建一个和NioSocketChannel绑定的配置类
  - 设置此channel tcpNoDelay为 true, 禁止negle算法 (使小的数据包集合成大的数据包再发送出去), 保证小的数据包尽可能发出去, 降低延时

```

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 {
            setTcpNoDelay(true);
        } catch (Exception e) {
            // Ignore.
        }
    }
}

```

思考：与创建服务端channel不同的是，服务端channel是利用反射创建，而这里直接使用new 关键词，netty为什么这么设计？

### 5.3 Netty中channel的分类

- NioServerSocketChannel

- Netty服务端channel的创建：用户代码传进来一个class类，netty拿到这个类通过反射方式创建。
- NioSocketChannel
  - 新连接接入过程中，拿到jdk底层创建的channel之后，通过显式的new关键字创建客户端channel
- unsafe
  - 用于实现每一种channel底层具体的协议

### 5.3.1 netty中channel的层级关系

#### io.netty.channel.Channel:

```
A nexus to a network socket or a component which is capable of I/O
* operations such as read, write, connect, and bind.
```

#### io.netty.channel.AbstractChannel

```
A skeletal {@link Channel} implementation.

...
private final Channel parent;
private final ChannelId id;
private final Unsafe unsafe;
private final DefaultChannelPipeline pipeline;
...
private volatile EventLoop eventLoop;
```

#### io.netty.channel.nio.AbstractNioChannel

使用select轮询的方式进行读写事件的监听

抽象出来，只关心I/O事件

```
Abstract base class for {@link Channel} implementations which use a selector based
approach.

...
private final SelectableChannel ch;
protected final int readInterestOp;
volatile SelectionKey selectionKey;

...
protected AbstractNioChannel(Channel parent, SelectableChannel ch, int readInterestOp)
{
    super(parent);
    this.ch = ch;
    this.readInterestOp = readInterestOp;
    try {
        ch.configureBlocking(false);
    } catch (IOException e) {
        ...
    }
}
```

#### io.netty.channel.nio.AbstractNioByteChannel

## io.netty.channel.nio.AbstractNioByteChannel#AbstractNioByteChannel

```
protected AbstractNioByteChannel(Channel parent, SelectableChannel ch) {
    super(parent, ch, SelectionKey.OP_READ);
}
```

客户端channel：创建的时候调用父类AbstractNioChannel 的构造函数，传递 read事件（I/O事件） 读取数据

```
{@link AbstractNioChannel} base class for {@link Channel}s that operate on bytes.
...
public final void read() {
    final ChannelConfig config = config();
    final ChannelPipeline pipeline = pipeline();
    final ByteBufAllocator allocator = config.getAllocator();
    final RecvByteBufAllocator.Handle allocHandle = recvBufAllocHandle();
    allocHandle.reset(config);
    ByteBuf byteBuf = null;
    boolean close = false;
    try {
        do {
            byteBuf = allocHandle.allocate(allocator);
            //读取字节数据
            allocHandle.lastBytesRead(doReadBytes(byteBuf));
            ...
        } while (allocHandle.continueReading());
        ...
    }
}
```

## io.netty.channel.nio.AbstractNioMessageChannel

服务端channel：创建的时候调用父类AbstractNioChannel 的构造函数，传递accept事件（I/O事件），监听连接

```
* {@link AbstractNioChannel} base class for {@link Channel}s that operate on messages.

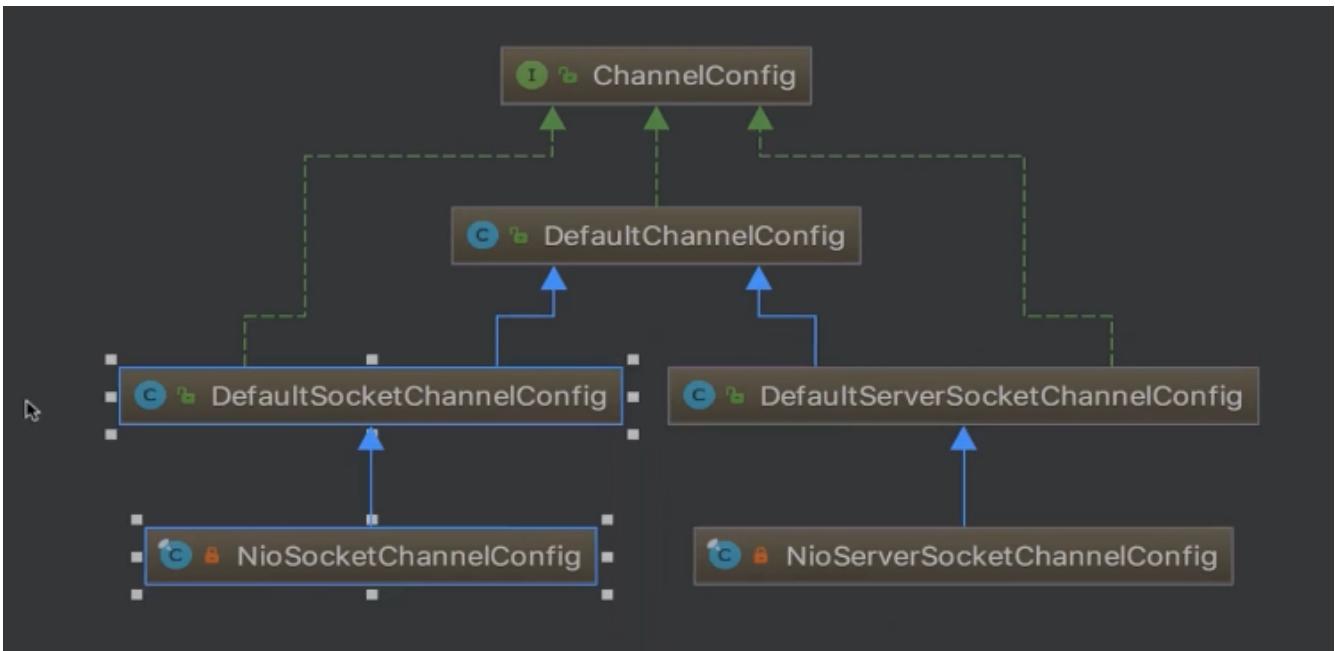
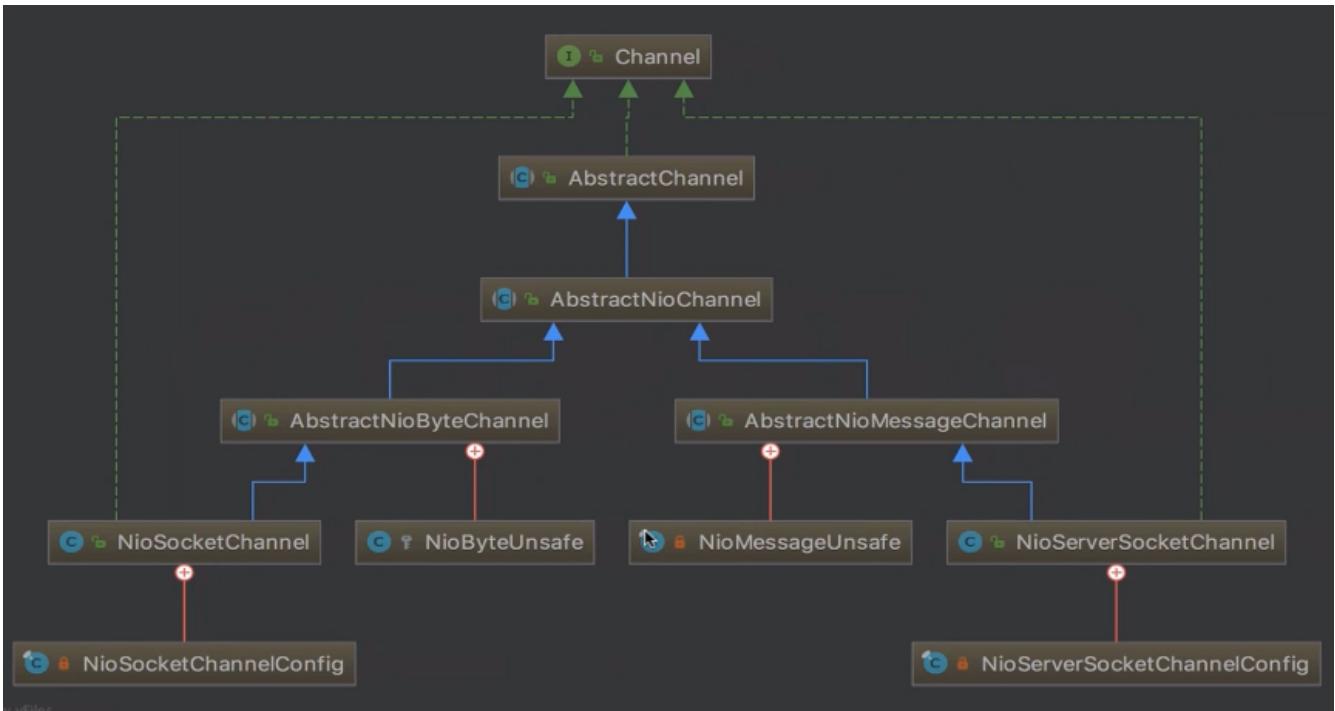
public void read() {
    assert eventLoop().inEventLoop();
    final ChannelConfig config = config();
    final ChannelPipeline pipeline = pipeline();
    final RecvByteBufAllocator.Handle allocHandle = unsafe().recvBufAllocHandle();
    allocHandle.reset(config);
    boolean closed = false;
    Throwable exception = null;
    try {
        try {
            do {
                ...
                int localRead = doReadMessages(readBuf);
                ...
                allocHandle.incMessagesRead(localRead);
            } while (allocHandle.continueReading());
        } catch (Throwable t) {
            exception = t;
        }
    }
}
```

```

    }
}

}

```



## 5.4 新连接 NioEventLoop分配和Selector注册

`io.netty.channel.nio.AbstractNioMessageChannel.NioMessageUnsafe#read`

--> `io.netty.channel.nio.AbstractNioMessageChannel#doReadMessages` 创建客户端channel

```

@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) {
        ...
    }
    return 0;
}

```

- > io.netty.channel.ChannelPipeline#fireChannelRead for 循环遍历每一条客户端连接，调用服务端 channel的PipeLine的fireChannelRead 方法

```

...
for (int i = 0; i < size; i++) {
    readPending = false;
    pipeline.fireChannelRead(readBuf.get(i));
}
...

```

- > 回顾netty服务端启动：io.netty.bootstrap.ServerBootstrap#init
- > 服务端channel PipeLine的构成

服务端channel PipeLine传播channelRead事件会从head开始--ServerBootstrapAcceptor--最后到Tail

即 `pipeline.fireChannelRead(readBuf.get(i))` 会将每一条客户端连接通过fireChannelRead逐层传到 ServerBootstrapAcceptor，即调用io.netty.bootstrap.ServerBootstrap.ServerBootstrapAcceptor#channelRead

三

## 服务端Channel的pipeline构成



- > ServerBootstrapAcceptor#channelRead主要做以下几件事情
  - 添加childHandler
  - 设置options和attrs
  - 选择NioEventLoop并注册selector

```

public void channelRead(ChannelHandlerContext ctx, Object msg) {

```

```

final Channel child = (Channel) msg;
//这里的childHandler是以一个特殊的handler, 即服务端启动时传进来的ChannelInitializer
child.pipeline().addLast(childHandler);
//childOptions 底层jdk读写相关的参数
for (Entry<ChannelOption<?>, Object> e: childOptions) {
    try {
        if (!child.config().setOption((ChannelOption<Object>) e.getKey(),
e.getValue())) {
            logger.warn("Unknown channel option: " + e);
        }
    } catch (Throwable t) {
        logger.warn("Failed to set a channel option: " + child, t);
    }
}
//childAttrs 在客户端channel上绑定一些自定义的属性 如密钥 channel存活时间等
for (Entry<AttributeKey<?>, Object> e: childAttrs) {
    child.attr((AttributeKey<Object>) e.getKey()).set(e.getValue());
}

try {
    // childGroup 是一个workGroup 注册的时候选择一个NioEventLoop进行注册
    childGroup.register(child).addListener(new ChannelFutureListener() {
        @Override
        public void operationComplete(ChannelFuture future) throws Exception {
            if (!future.isSuccess()) {
                forceClose(child, future.cause());
            }
        }
    });
} catch (Throwable t) {
    forceClose(child, t);
}
}
}

```

io.netty.channel.ChannelInitializer#handlerAdded --»

io.netty.channel.ChannelInitializer#initChannel(io.netty.channel.ChannelHandlerContext)

```

private boolean initChannel(ChannelHandlerContext ctx) throws Exception {
    if (initMap.putIfAbsent(ctx, Boolean.TRUE) == null) { // Guard against re-entrance.
        try {
            //这里会回调到用户代码里的方法
            ChannelInitializer#initChannel(io.netty.channel.socket.SocketChannel)
                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 {
            //调用remove将自身删除
        }
    }
}

```

```
        remove(ctx);
    }
    return true;
}
return false;
}
//这就是netty为新连接添加channelHandler的逻辑
```

io.netty.channel.EventLoopGroup#register(io.netty.channel.Channel)

```
io.netty.channel.MultithreadEventLoopGroup#register(io.netty.channel.Channel)
@Override
public ChannelFuture register(Channel channel) {
    return next().register(channel);
}
```

next()返回一个NioEventLoop

```
@Override
public EventLoop next() {
    return (EventLoop) super.next();
}
```

io.netty.util.concurrent.MultithreadExecutorGroup#next

```
@Override
public EventExecutor next() {
    return chooser.next();
}
```

客户端channel选择nioEventLoop并注册selector的过程

```
io.netty.channel.SingleThreadEventLoop#register(io.netty.channel.ChannelPromise)
@Override
public ChannelFuture register(final ChannelPromise promise) {
    ObjectUtil.checkNotNull(promise, "promise");
    promise.channel().unsafe().register(this, promise);
    return promise;
}
```

```
io.netty.channel.nio.AbstractNioChannel#doRegister
protected void doRegister() throws Exception {
    boolean selected = false;
    for (;;) {
        try {
            selectionKey = javaChannel().register(eventLoop().selector, 0, this);
            return;
        } catch (CancelledKeyException e) {
```

```

        if (!selected) {
            eventLoop().selectNow();
            selected = true;
        } else {
            throw e;
        }
    }
}
}

```

小结：服务端channel在检测到新连接并且创建完客户端channel之后，会调用一个连接器做一些处理，

包括给客户端channel填充逻辑处理器channelHanler，配置options和attrs，选定一个NioEventLoop进行绑定，并把channel注册到NioEventLoop的selector上，这时不关心任何事件。

## 5.5 NioSocketChannel读事件的注册

io.netty.channel.AbstractChannel.AbstractUnsafe#register0

通过debug方式了解相应代码

io.netty.channel.DefaultChannelPipeline.HeadContext#channelActive

```

@Override
public void channelActive(ChannelHandlerContext ctx) throws Exception {
    //传播channelActive 事件
    ctx.fireChannelActive();

    readIfIsAutoRead();
}

```

```

private void readIfIsAutoRead() {
    if (channel.config().isAutoRead()) { //默认只要绑定端口就会接收连接，只要当前连接绑定到selector
        上就会自动读，即向selector上注册读事件
        channel.read();
    }
}

```

io.netty.channel.AbstractChannel#read

```

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

```

io.netty.channel.DefaultChannelPipeline#read

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

io.netty.channel.AbstractChannelHandlerContext#invokeRead

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

io.netty.channel.DefaultChannelPipeline.HeadContext#read

```
public void read(ChannelHandlerContext ctx) {
    unsafe.beginRead();
}
```

io.netty.channel.AbstractChannel.AbstractUnsafe#beginRead

```
public final void beginRead() {
    assertEventLoop();
    ...
    try {
        doBeginRead();
    } catch (final Exception e) {
        ...
    }
}
```

io.netty.channel.nio.AbstractNioChannel#doBeginRead

```
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();
    // readInterestOp 即创建nioSocketChannel时传进来的OP_READ
```

```
    if ((interestOps & readInterestOp) == 0) {
        selectionKey.interestOps(interestOps | readInterestOp);
    }
}
```

## 六. netty-PipeLine

PipeLine是netty的大动脉，主要负责读写事件的传播

### 6.1 思考问题

- netty是如何判断ChannelHandler类型的
  - 调用pipeline添加handlerContext节点的时候，根据instanceOf关键词，判断当前节点是 inbound还是outbound类型，并用一个boolean类型的变量标识
- 对于ChannelHandler的添加应遵循什么样的顺序
  - inbound类型事件的传播与添加inboundHanler的顺序正相关
  - outbound类型事件的传播与添加outboundHanler的顺序正相关
- 用户手动触发事件传播，不同的触发方式有什么样的区别？
  - 通过channel触发时，从tail或head节点往下传播
  - 通过当前节点触发时，从当前节点开始往下传播

### 6.2 学习内容

- pipeLine的初始化
- 添加和删除ChannelHandler
- 事件和异常的传播

### 6.3 pipeLine的初始化

#### 6.3.1 pipeline在创建channel的时候被创建

不管服务端channel还是客户端channel，都会调用abstractChannel的构造函数

io.netty.channel.AbstractChannel#AbstractChannel(io.netty.channel.Channel, io.netty.channel.ChannelId)

```
protected AbstractChannel(Channel parent, ChannelId id) {
    this.parent = parent;
    this.id = id;
    unsafe = newUnsafe();
    pipeline = newChannelPipeline();
}
```

io.netty.channel.DefaultChannelPipeline#DefaultChannelPipeline

```

protected DefaultChannelPipeline(channel channel) {
    this.channel = objectUtil.checkNotNull(channel, "channel");
    succeededFuture = new SucceededChannelFuture(channel, null);
    voidPromise = new VoidChannelPromise(channel, true);

    tail = new TailContext(this);
    head = new HeadContext(this);

    //通过next和prev将tail和head变成一个双向链表数据结构
    head.next = tail;
    tail.prev = head;
}

```

### 6.3.2 pipeline节点数据结构: ChannelHandlerContext

pipeline的每一个节点都是ChannelHandlerContext这种数据结构

```

public interface ChannelHandlerContext extends AttributeMap, ChannelInboundInvoker,
ChannelOutboundInvoker {
    /**
     * Return the {@link Channel} which is bound to the {@link ChannelHandlerContext}.
     */
    Channel channel();

    /** 
     * Returns the {@link EventExecutor} which is used to execute an arbitrary task.
     */
    EventExecutor executor();

    /**
     * The unique name of the {@link ChannelHandlerContext}.The name was used when then
     {@link ChannelHandler}
     * was added to the {@link ChannelPipeline}. This name can also be used to access the
     registered
     * {@link ChannelHandler} from the {@link ChannelPipeline}.
     */
    String name();

    /**
     * The {@link ChannelHandler} that is bound this {@link ChannelHandlerContext}.
     */
    ChannelHandler handler();

    /**
     * Return {@code true} if the {@link ChannelHandler} which belongs to this context was
     removed
     * from the {@link ChannelPipeline}. Note that this method is only meant to be called
     from with in the
     * {@link EventLoop}.
     */
    boolean isRemoved();

    @Override

```

```
channelHandlerContext fireChannelRegistered();

@Override
channelHandlerContext fireChannelUnregistered();

@Override
channelHandlerContext fireChannelActive();

@Override
channelHandlerContext fireChannelInactive();

@Override
channelHandlerContext fireExceptionCaught(Throwable cause);

@Override
channelHandlerContext fireUserEventTriggered(Object evt);

@Override
channelHandlerContext fireChannelRead(Object msg);

@Override
channelHandlerContext fireChannelReadComplete();

@Override
channelHandlerContext fireChannelWritabilityChanged();

@Override
channelHandlerContext read();

@Override
channelHandlerContext flush();

/**
 * Return the assigned {@link ChannelPipeline}
 */
ChannelPipeline pipeline();

/**内存分配器：当前节点有数据读写，分配bytebuf的时候用哪个内存分配器分配
 * Return the assigned {@link ByteBufAllocator} which will be used to allocate {@link ByteBuf}s.
 */
ByteBufAllocator alloc();

/**
 * @deprecated Use {@link channel#attr(AttributeKey)}
 */
@Deprecated
@Override
<T> Attribute<T> attr(AttributeKey<T> key);

/**
 * @deprecated Use {@link channel#hasAttr(AttributeKey)}
 */

```

```
@Deprecated  
@Override  
    <T> boolean hasAttr(AttributeKey<T> key);  
}
```

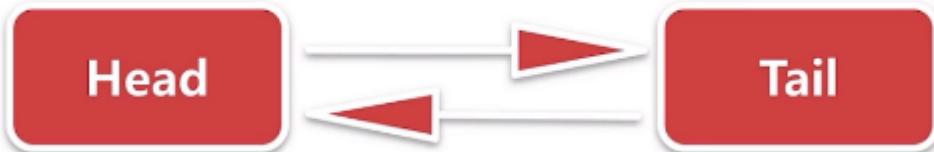
- io.netty.util.AttributeMap 存储自定义属性
- io.netty.channel.ChannelInboundInvoker 传播读事件
- io.netty.channel.ChannelOutboundInvoker 传播写事件
- ChannelHandlerContext 的默认实现 io.netty.channel.AbstractChannelHandlerContext
  - //pipeline的串行结构主要依赖于以下两个属性  
`volatile AbstractChannelHandlerContext next;`  
`volatile AbstractChannelHandlerContext prev;`

### 6.3.3 Pipeline两大哨兵 TailContext & HeadContext

Head节点的unsafe负责实现channel的具体协议

Tail节点起到终止事件和异常传播的作用

## pipeline默认结构



```
protected DefaultChannelPipeline(Channel channel) {  
    ...  
    tail = new TailContext(this);  
    head = new HeadContext(this);  
    ...  
}
```

- TailContext - inbound处理器

```

AbstractChannelHandlerContext(DefaultChannelPipeline pipeline, EventExecutor executor,
String name, boolean inbound, boolean outbound) {
    this.name = ObjectUtil.checkNotNull(name, "name");
    this.pipeline = pipeline;
    this.executor = executor;
    this.inbound = inbound;
    this.outbound = outbound;
    // Its ordered if its driven by the EventLoop or the given Executor is an instanceof
    OrderedEventExecutor.
    ordered = executor == null || executor instanceof OrderedEventExecutor;
}

```

```

// A special catch-all handler that handles both bytes and messages.
final class TailContext extends AbstractChannelHandlerContext implements
ChannelInboundHandler {

    TailContext(DefaultChannelPipeline pipeline) {
        super(pipeline, null, TAIL_NAME, true, false);
        setAddComplete(); //设置已添加标记
    }
    //业务逻辑处理器
    @Override
    public ChannelHandler handler() {
        return this;
    }

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

    @Override
    public void channelUnregistered(ChannelHandlerContext ctx) throws Exception { } //空

    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception { } //空

    @Override
    public void channelInactive(ChannelHandlerContext ctx) throws Exception { } //空

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

    @Override
    public void handlerAdded(ChannelHandlerContext ctx) throws Exception { } //空

    @Override
    public void handlerRemoved(ChannelHandlerContext ctx) throws Exception { } //空

    @Override
    public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception
{
    // This may not be a configuration error and so don't log anything.
    // The event may be superfluous for the current pipeline configuration.
}

```

```

        ReferenceCountUtil.release(evt);
    }
    //发生异常时进行处理
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws
Exception {
        onUnhandledInboundException(cause);
    }
    //传进来的消息未被处理时会触发
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        onUnhandledInboundMessage(msg);
    }

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

```

结论：TailHead主要做一些收尾的事情，如异常未处理好会给予警告，消息没处理会建议你处理

- HeadContext - outbound处理器

```

final class HeadContext extends AbstractChannelHandlerContext
    implements ChannelOutboundHandler, ChannelInboundHandler {
    //处理底层数据读写
    private final Unsafe unsafe;

    HeadContext(DefaultChannelPipeline pipeline) {
        super(pipeline, null, HEAD_NAME, false, true);
        unsafe = pipeline.channel().unsafe();
        setAddComplete();
    }

    @Override
    public ChannelHandler handler() {
        return this;
    }

    @Override
    public void handlerAdded(ChannelHandlerContext ctx) throws Exception {
        // NOOP
    }

    @Override
    public void handlerRemoved(ChannelHandlerContext ctx) throws Exception {
        // NOOP
    }

    @Override
    public void bind(
        ChannelHandlerContext ctx, SocketAddress localAddress, ChannelPromise
promise)
        throws Exception {

```

```
        unsafe.bind(localAddress, promise);
    }

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

    @Override
    public void disconnect(ChannelHandlerContext ctx, ChannelPromise promise)
throws Exception {
        unsafe.disconnect(promise);
    }

    @Override
    public void close(ChannelHandlerContext ctx, ChannelPromise promise) throws
Exception {
        unsafe.close(promise);
    }

    @Override
    public void deregister(ChannelHandlerContext ctx, ChannelPromise promise)
throws Exception {
        unsafe.deregister(promise);
    }

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

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

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

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws
Exception {
        ctx.fireExceptionCaught(cause);
    }

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

```
        invokeHandlerAddedIfNeeded();
        ctx.fireChannelRegistered();
    }

@Override
public void channelUnregistered(ChannelHandlerContext ctx) throws Exception {
    ctx.fireChannelUnregistered();

    // Remove all handlers sequentially if channel is closed and unregistered.
    if (!channel.isOpen()) {
        destroy();
    }
}

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

    readIfIsAutoRead();
}

@Override
public void channelInactive(ChannelHandlerContext ctx) throws Exception {
    ctx.fireChannelInactive();
}

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

@Override
public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {
    ctx.fireChannelReadComplete();

    readIfIsAutoRead();
}

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

@Override
public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws
Exception {
    ctx.fireUserEventTriggered(evt);
}

@Override
```

```

    public void channelWritabilityChanged(ChannelHandlerContext ctx) throws
Exception {
    ctx.fireChannelWritabilityChanged();
}
}

```

结论：HeadContext 主要做的事情：1) 往下传播读写事件：netty每次传播读写事件都会从Head开始

2) 读写事件委托给 unsafe 进行读写

## 6.4 Pipeline 添加ChannelHandler

- 判断是否重复添加ChannelHandler
- 创建节点并添加至链表
  - 创建的节点就是ChannelHandlerContext，将ChannelHandler包装成ChannelHandlerContext 添加到链表
- 回调添加完成事件
  - ChannelInitializer被添加完成之后会回调到用户代码（自己实现的initChannel()方法）

```

// 用户代码 ChannelInitializer 是个抽象方法
bootstrap.group(bossGroup, workGroup)
    .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 ChannelInboundHandlerAdapter());
            ch.pipeline().addLast(new ChannelOutboundHandlerAdapter());
        }
    });
}

```

io.netty.channel.DefaultChannelPipeline#addLast(io.netty.util.concurrent.EventExecutorGroup, java.lang.String, io.netty.channel.ChannelHandler)

```

@Override
public final ChannelPipeline addLast(EventExecutorGroup group, String name, ChannelHandler
handler) {
    final AbstractChannelHandlerContext newCtx;
    synchronized (this) {
        // 判断是否重复添加
        checkMultiplicity(handler);
        // 创建节点
        newCtx = newContext(group, filterName(name, handler), handler);
        // 并添加至链表
        addLast0(newCtx);
    }
}

```

```

    // If the registered is false it means that the channel was not registered on an
    eventloop yet.
    // In this case we add the context to the pipeline and add a task that will call
    // ChannelHandler.handlerAdded(...) once the channel is registered.
    if (!registered) {
        newCtx.setAddPending();
        callHandlerCallbackLater(newCtx, true);
        return this;
    }
    EventExecutor executor = newCtx.executor();

    if (!executor.inEventLoop()) {
        newCtx.setAddPending();
        //io.netty.util.concurrent.SingleThreadEventExecutor#execute 添加到Mpsc队列
        executor.execute(new Runnable() {
            @Override
            public void run() {
                callHandlerAdded0(newCtx);
            }
        });
        return this;
    }
    //callHandlerAdded0(newCtx);
    return this;
}

```

#### io.netty.channel.DefaultChannelPipeline#callHandlerAdded0

```

private void callHandlerAdded0(final AbstractChannelHandlerContext ctx) {
    try {
        //回调到用户代码:io.netty.channel.ChannelInitializer#handlerAdded
        ctx.handler().handlerAdded(ctx);
        //自旋 + CAS 设置已添加标记 ADD_COMPLETE
        ctx.setAddComplete();
    } catch (Throwable t) {
        boolean removed = false;
        try {
            //删除当前节点
            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);
            }
        }
        ...
    }
}

```

```
}
```

io.netty.channel.ChannelInitializer#handlerAdded

```
public void handlerAdded(ChannelHandlerContext ctx) throws Exception {
    if (ctx.channel().isRegistered()) {
        ...
        initChannel(ctx);
    }
}
```

io.netty.channel.ChannelInitializer#initChannel(C)

```
//抽象方法
protected abstract void initChannel(C ch) throws Exception;
```

## 6.5 删除ChannelHandler

### 6.5.1 应用场景之权限校验

```
public class AuthHandler extends SimpleChannelInboundHandler<ByteBuf> {

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws Exception {
        if (pass(msg)) {
            //校验通过将当前节点删除
            ctx.pipeline().remove(this);
        } else {
            //校验不通过直接关闭连接
            ctx.close();
        }
    }

    private boolean pass(ByteBuf password) {
        return false;
    }
}
```

### 6.5.2 删除过程

- 找到节点

io.netty.channel.DefaultChannelPipeline#getContextOrDie(io.netty.channel.ChannelHandler)  
io.netty.channel.DefaultChannelPipeline#context(io.netty.channel.ChannelHandler)

```
//通过遍历链表的方式找到ChannelHandler对应的ChannelHandlerContext
public final ChannelHandlerContext context(ChannelHandler handler) {
    if (handler == null) {
        throw new NullPointerException("handler");
    }
}
```

```

AbstractChannelHandlerContext ctx = head.next;
for (;;) {
    if (ctx == null) {
        return null;
    }
    if (ctx.handler() == handler) {
        return ctx;
    }
    ctx = ctx.next;
}
}

```

- 链表的删除

io.netty.channel.DefaultChannelPipeline#remove(io.netty.channel.AbstractChannelHandlerContext)

```

//标准的链表删除节点方法
private static void remove0(AbstractChannelHandlerContext ctx) {
    AbstractChannelHandlerContext prev = ctx.prev;
    AbstractChannelHandlerContext next = ctx.next;
    prev.next = next;
    next.prev = prev;
}

```

- 回调删除Handler事件

io.netty.channel.DefaultChannelPipeline#callHandlerRemoved0

```

private void callHandlerRemoved0(final AbstractChannelHandlerContext ctx) {
    // Notify the complete removal.
    try {
        try {
            //这里会拿到当前节点的channelHandler，调用handlerRemoved方法，最终调用到回调方法里面
            ctx.handler().handlerRemoved(ctx);
        } finally {
            //用户的回调方法执行结束之后，执行setRemoved方法
            ctx.setRemoved();
        }
    } catch (Throwable t) {
        fireExceptionCaught(new ChannelPipelineException(
            ctx.handler().getClass().getName() + ".handlerRemoved() has thrown
            an exception.", t));
    }
}

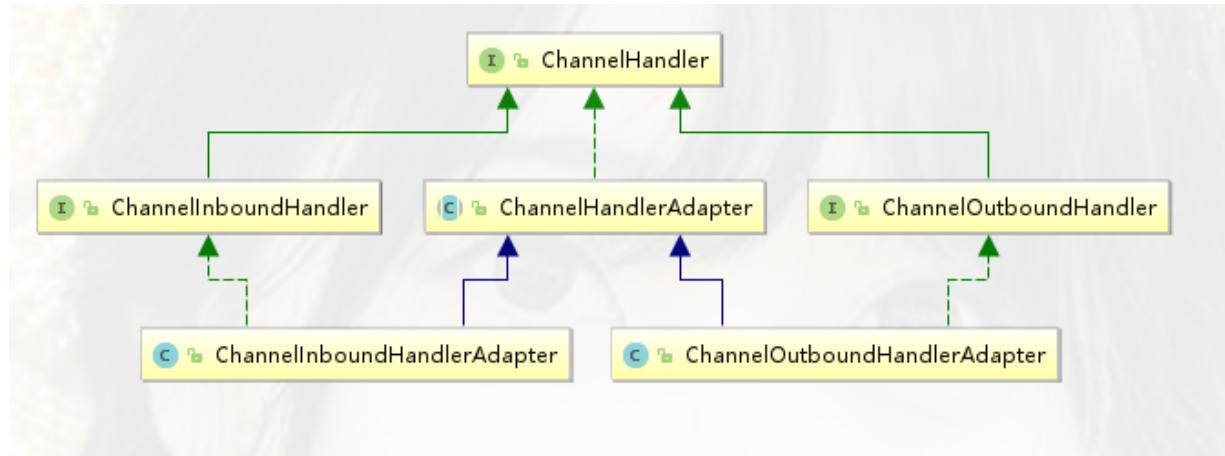
```

## 6.6 inBound事件的传播

- 何为inBound事件以及ChannelInBoundHandler
- ChannelRead事件的传播

- ChannelRead事件是典型的inbound事件
- 按照添加handler的顺序传播
- SimpleInBoundHandler处理器
  - 自动释放bytebuf对象

netty中channelHandler接口的继承关系如图



最顶层接口 ChannelHandler

默认实现 ChannelHandlerAdapter

ChannelInboundHandler和ChannelOutboundHandler:

继承了ChannelHandler接口，分别定制了一些特殊的功能

平时用户代码编写channelHandler较多用到：

直接继承： ChannelInboundHandlerAdapter 和ChannelOutboundHandlerAdapter

ChannelInboundHandler在ChannelHandler基础上做了扩展：

```

public interface ChannelInboundHandler extends ChannelHandler {

    /**
     * channel注册到nioEventLoop对应的selector后会回调到 ChannelHandler
     */
    void channelRegistered(ChannelHandlerContext ctx) throws Exception;

    void channelUnregistered(ChannelHandlerContext ctx) throws Exception;

    /**
     * channel激活后的回调
     */
    void channelActive(ChannelHandlerContext ctx) throws Exception;

    /**
     * channel失效的回调
     */
    void channelInactive(ChannelHandlerContext ctx) throws Exception;

    /**
     * channel读到了一些数据或是接收了一些连接,
     * 对服务端channel而言, msg是连接
     * 对客户端channel而言, msg是bytebuf数据
     */
  
```

```

void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception;

/**
读完之后的回调
*/
void channelReadComplete(ChannelHandlerContext ctx) throws Exception;

void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception;

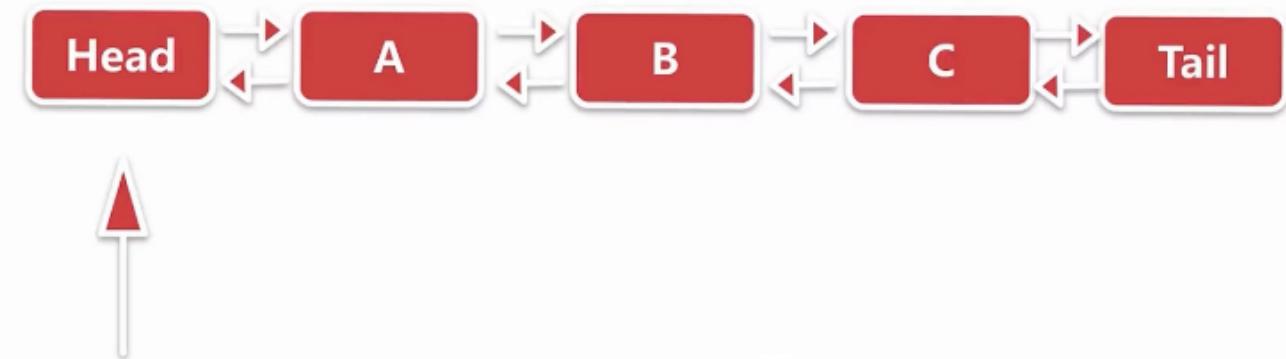
void channelWritabilityChanged(ChannelHandlerContext ctx) throws Exception;

/**
异常捕获的回调
*/
@Override
@SuppressWarnings("deprecation")
void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception;
}

```

## inBound事件的传播

14



**channelRead()**

com.leh.netty.pipeline.InboundHandlerB

```

public class InboundHandlerB extends ChannelInboundHandlerAdapter {
    /**
     * channelHandler收到激活事件后会对收到对象进行打印，并继续传播
     * @param ctx
     * @param msg
     * @throws Exception
     */
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        System.out.println("InboundHandlerB: " + msg);
    }
}

```

```
//通过context调用fireChannelRead 会将事件从当前节点开始传播
ctx.fireChannelRead(msg);
}

/**
 * 通道被激活的时候拿到对应的 pipeline , 激活一个ChannelRead事件
 * @param ctx
 * @throws Exception
 */
@Override
public void channelActive(ChannelHandlerContext ctx) throws Exception {
    //通过channel的Pipeline调用fireChannelRead
    //会将事件从pipeline的headContext节点传播
    ctx.channel().pipeline().fireChannelRead("hello netty");
}
}
```

io.netty.channel.DefaultChannelPipeline#fireChannelRead

```
//从head开始
public final ChannelPipeline fireChannelRead(Object msg) {
    AbstractChannelHandlerContext.invokeChannelRead(head, msg);
    return this;
}
```

```
static void invokeChannelRead(final AbstractChannelHandlerContext next, Object msg) {
    final Object m = next.pipeline.touch(ObjectUtil.checkNotNull(msg, "msg"), next);
    //此时的next即io.netty.channel.DefaultChannelPipeline.HeadContext
    EventExecutor executor = next.executor();
    if (executor.inEventLoop()) {
        //调用HeadContext的invokeChannelRead方法
        next.invokeChannelRead(m);
    } else {
        executor.execute(new Runnable() {
            @Override
            public void run() {
                next.invokeChannelRead(m);
            }
        });
    }
}
```

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

io.netty.channel.DefaultChannelPipelineHandlerContext#channelRead

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

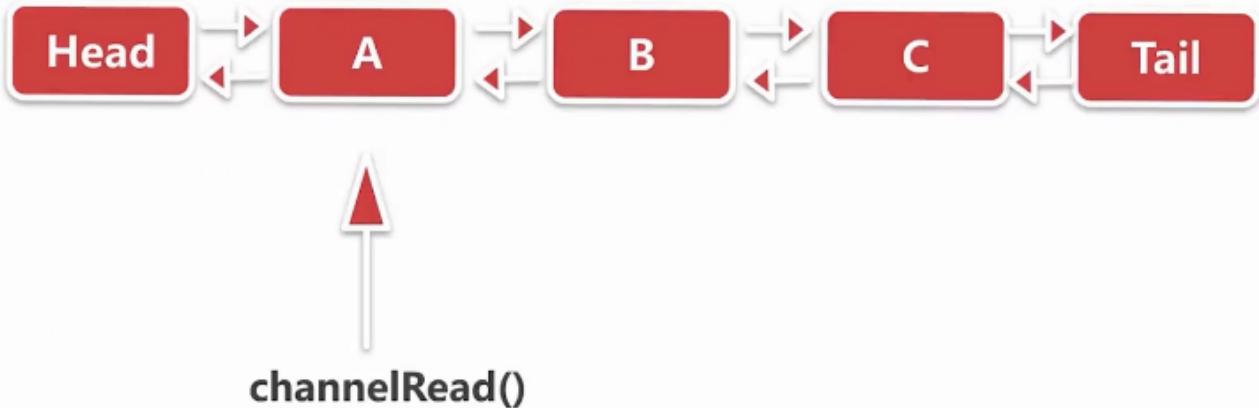
io.netty.channel.AbstractChannelHandlerContext#fireChannelRead

```
public ChannelHandlerContext fireChannelRead(final Object msg) {
    //寻找下一个inboundHandler 找到之后通过invokeChannelRead 继续向下传播
    invokeChannelRead(findContextInbound(), msg);
    return this;
}
```

io.netty.channel.AbstractChannelHandlerContext#findContextInbound

```
private AbstractChannelHandlerContext findContextInbound() {
    AbstractChannelHandlerContext ctx = this;
    do {
        ctx = ctx.next;
    } while (!ctx.inbound);
    //inbound属性在addLast添加handler时候赋值的
    //循环查找，找到inboundhandler即返回
    return ctx;
}
```

# inBound事件的传播



此时找到了下一个inboundHandler是com.leh.netty.pipeline.InboundHandlerA

io.netty.channel.AbstractChannelHandlerContext#invokeChannelRead(io.netty.channel.AbstractChannelHandlerContext, java.lang.Object)

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

com.leh.netty.pipeline.InboundHandlerA#channelRead

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
    System.out.println("InboundHandlerA: " + msg);
    //通过handler直接调用fireChannelRead 会把这个事件从当前节点开始往下传播
    ctx.fireChannelRead(msg);
}
```

同理,接下来继续传播到 B--> C-->Tail

到达TailContext节点, 仍然是一个inboundHandler

io.netty.channel.DefaultChannelPipeline.TailContext#channelRead

```
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
    onUnhandledInboundMessage(msg);
}
```

io.netty.channel.DefaultChannelPipeline#onUnhandledInboundMessage

```
//消息传递直到tail节点都未被处理时进入该方法进行打印并释放
protected void onUnhandledInboundMessage(Object msg) {
    try {
        logger.debug(
            "Discarded inbound message {} that reached at the tail of the pipeline. " +
            "Please check your pipeline configuration.", msg);
    } finally {
        ReferenceCountUtil.release(msg);
    }
}
```

### SimpleChannelInBoundHandler处理器应用场景

当channelRead事件传播的时候，如果msg对象是个byteBuf，并且在handler中对byteBuf做了读写处理但没有继续向下传播，那么就传播不到tail节点，就没办法自动释放，这就需要用户自己释放，如果用户代码没有处理释放bytebuf，最终可能导致内存泄漏。

netty帮助我们封装了SimpleChannelInBoundHandler，能够自动释放byteBuf；

SimpleChannelInBoundHandler是如何做到自动释放byteBuf的？

```
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) {
            //最终由SimpleChannelInBoundHandler自动释放
            ReferenceCountUtil.release(msg);
        }
    }
}

...
//channelRead0是个抽象方法
protected abstract void channelRead0(ChannelHandlerContext ctx, I msg) throws Exception;
```

```

public class AuthHandler extends SimpleChannelInboundHandler<ByteBuf> {

    //用户只需要自己去覆盖SimpleChannelInboundHandler的channelRead0方法进行读写，不需要考虑释放
    bytebuf
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws
Exception {
        if (pass(msg)) {
            //校验通过将当前节点删除
            ctx.pipeline().remove(this);
        } else {
            //校验不通过直接关闭连接
            ctx.close();
        }
    }
    @Override
    public void handlerRemoved(ChannelHandlerContext ctx) throws Exception {
        System.out.println("test 回调删除");
    }
    private boolean pass(ByteBuf password) {
        return true;
    }
}

```

## 6.7 outBound事件的传播

### 6.7.1 思考两个问题

- 何为outBound事件及channelOutBoundHandler
  - write()事件的传播
- io.netty.channel.ChannelInboundHandler 被动触发的事件  
 io.netty.channel.ChannelOutboundHandler 用户主动发起的事件

```

public interface ChannelOutboundHandler extends ChannelHandler {
    /**
     * Called once a bind operation is made.
     */
    void bind(ChannelHandlerContext ctx, SocketAddress localAddress, ChannelPromise
promise) throws Exception;

    /**
     * Called once a connect operation is made.
     */
    void connect(
        ChannelHandlerContext ctx, SocketAddress remoteAddress,
        SocketAddress localAddress, ChannelPromise promise) throws Exception;

    void disconnect(ChannelHandlerContext ctx, ChannelPromise promise) throws
Exception;

```

```

    /**
     * Called once a close operation is made.
     */
    void close(ChannelHandlerContext ctx, ChannelPromise promise) throws Exception;

    void deregister(ChannelHandlerContext ctx, ChannelPromise promise) throws
Exception;

    /**
     * Intercepts {@link ChannelHandlerContext#read()}.
     */
    void read(ChannelHandlerContext ctx) throws Exception;

    /**
     * Called once a write operation is made.
     */
    void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise)
throws Exception;

    /**
     * Called once a flush operation is made.
     */
    void flush(ChannelHandlerContext ctx) throws Exception;
}

```

```

public final class TestChannelOutboundHandler {

    public static void main(String[] args) {
        // bossGroup 对应 socket编程中 server端 的 线程
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        // workGroup 对应 socket编程中 client端 的 线程
        EventLoopGroup workGroup = new NioEventLoopGroup();

        ServerBootstrap bootstrap = new ServerBootstrap();

        bootstrap.group(bossGroup, workGroup)
            .channel(NioServerSocketChannel.class)
            .childOption(ChannelOption.TCP_NODELAY, true)
            .childAttr(AttributeKey.newInstance("childAttr"), "childAttrValue")
            .childHandler(new ChannelInitializer<SocketChannel>() {

                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline().addLast(new outboundHandlerA());
                    ch.pipeline().addLast(new outboundHandlerB());
                    ch.pipeline().addLast(new outboundHandlerC());
                }
            });
        try {
            //服务端创建的入口 bind()

```

```

        channelFuture.channelFuture = bootstrap.bind(8888).sync();
        channelFuture.channel().closeFuture().sync();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

output: 结论: outboundHandler添加的顺序和在pipeline中传播的顺序是相反的
OutboundHandlerC: hello netty
OutboundHandlerB: hello netty
OutboundHandlerA: hello netty

```

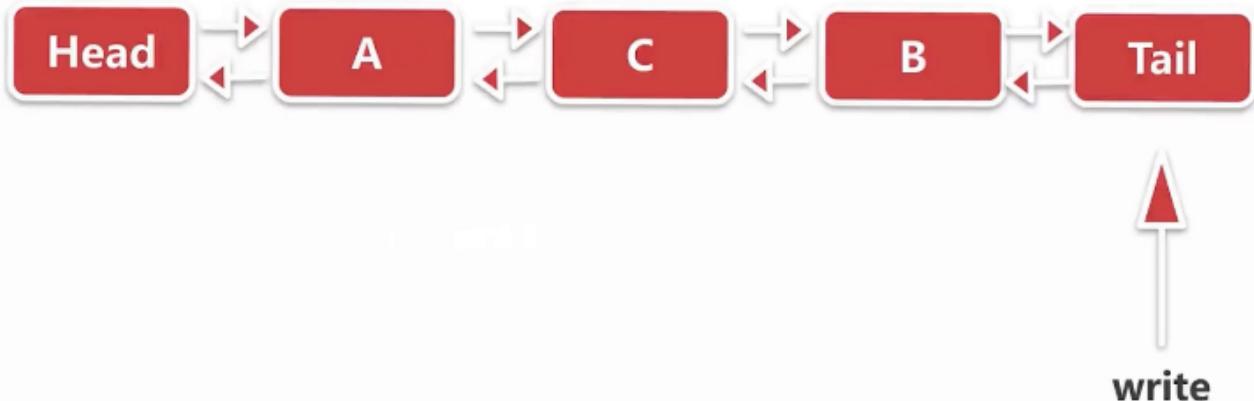
```

public class OutboundHandlerB extends ChannelOutboundHandlerAdapter {
    @Override
    public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {
        System.out.println("OutboundHandlerB: " + msg);
        //继续向下传播
        ctx.write(msg, promise);
    }

    /**
     * 做一个定时器的调用
     * 模拟实际项目中读到数据处理完毕后给客户端一个响应
     * @param ctx
     * @throws Exception
     */
    @Override
    public void handlerAdded(ChannelHandlerContext ctx) throws Exception {
        ctx.executor().schedule(() -> {
            //从Tail节点开始传播
            ctx.channel().write("hello netty");
            //从当前节点开始传播
            //ctx.write("hello netty");
        }, 3, TimeUnit.SECONDS);
    }
}

```

# outBound事件的传播



## 6.7.2 源码分析

io.netty.channel.DefaultChannelPipeline#write(java.lang.Object) 委托给pipeline进行读写

```
//从Tail节点开始往前写
public final ChannelFuture write(Object msg) {
    return tail.write(msg);
}
```

```
public ChannelFuture write(Object msg) {
    //promise是一个回调
    return write(msg, newPromise());
}
```

```
private void write(Object msg, boolean flush, ChannelPromise promise) {
    //寻找outbound节点
    AbstractChannelHandlerContext next = findContextOutbound();
    final Object m = pipeline.touch(msg, next);
    EventExecutor executor = next.executor();
    if (executor.inEventLoop()) {
        if (flush) {
            next.invokeWriteAndFlush(m, promise);
        } else {
            //找到B节点后调用
            next.invokeWrite(m, promise);
        }
    } else {
        AbstractWriteTask task;
        if (flush) {
            task = WriteAndFlushTask.newInstance(next, m, promise);
        } else {
            task = WriteTask.newInstance(next, m, promise);
        }
    }
}
```

```

        }
        safeExecute(executor, task, promise, m);
    }
}

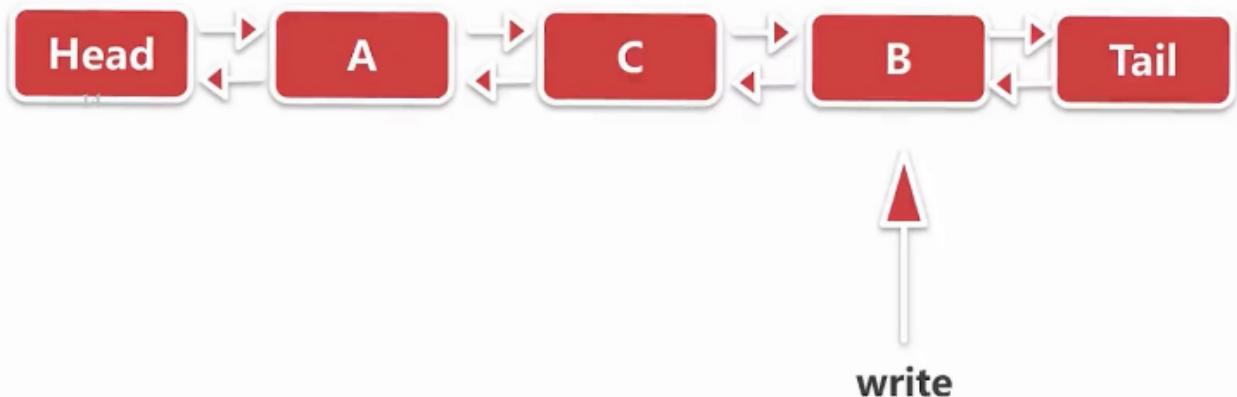
```

io.netty.channel.AbstractChannelHandlerContext#findContextOutbound

```

private AbstractChannelHandlerContext findContextOutbound() {
    AbstractChannelHandlerContext ctx = this;
    do {
        ctx = ctx.prev;
    } while (!ctx.outbound);
    return ctx;
}

```



io.netty.channel.AbstractChannelHandlerContext#invokeWrite0

```

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

```

com.leh.netty.pipeline.OutboundHandlerB#write

```

public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws
Exception {
    System.out.println("OutboundHandlerB: " + msg);
    //继续向下传播
    ctx.write(msg, promise);
}

```

直到到达HeadContext节点

io.netty.channel.DefaultChannelPipeline.HeadContext#write

```
@Override  
public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws  
Exception {  
    //最终调用unsafe的write方法，不再往下传播  
    unsafe.write(msg, promise);  
}
```

## 6.8 异常的传播

- inbound事件传播的顺序和在应用程序中添加inboundHandler的顺序正相关
- outbound事件传播的顺序和在应用程序中添加outboundHandler的顺序逆相关
- 那么异常呢？

模拟异常传播

```
telnet 127.0.0.1 8888 模拟客户端连接
```

```
send ayt 模拟发送消息
```

```
public final class TestExceptionSpread {  
  
    public static void main(String[] args) {  
        // bossGroup 对应 socket编程中 server端 的 线程  
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);  
        // workGroup 对应 socket编程中 client端 的 线程  
        EventLoopGroup workGroup = new NioEventLoopGroup();  
  
        ServerBootstrap bootstrap = new ServerBootstrap();  
  
        bootstrap.group(bossGroup, workGroup)  
            .channel(NioServerSocketChannel.class)  
            .childOption(ChannelOption.TCP_NODELAY, true)  
            .childAttr(AttributeKey.newInstance("childAttr"), "childAttrValue")  
            .childHandler(new ChannelInitializer<SocketChannel>() {  
                @Override  
                protected void initChannel(SocketChannel ch) throws Exception {  
                    ch.pipeline().addLast(new ExceptionInboundHandlerA());  
                    ch.pipeline().addLast(new ExceptionInboundHandlerB());  
                    ch.pipeline().addLast(new ExceptionInboundHandlerC());  
                    ch.pipeline().addLast(new ExceptionOutboundHandlerA());  
                    ch.pipeline().addLast(new ExceptionOutboundHandlerB());  
                    ch.pipeline().addLast(new ExceptionOutboundHandlerC());  
                }  
            });  
        try {  
            //服务端创建的入口 bind()  
            ChannelFuture channelFuture = bootstrap.bind(8888).sync();  
            channelFuture.channel().closeFuture().sync();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```

    }
}

output:
ExceptionInboundHandlerB.exceptionCaught
ExceptionInboundHandlerC.exceptionCaught
ExceptionOutboundHandlerA.exceptionCaught
ExceptionOutboundHandlerB.exceptionCaught
ExceptionOutboundHandlerC.exceptionCaught

```

模拟异常

```

public class ExceptionInboundHandlerB extends ChannelInboundHandlerAdapter {

    //抛出异常
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        throw new BusinessException("exception from ExceptionInboundHandlerB");
    }

    //抛出异常回调
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {
        System.out.println("ExceptionInboundHandlerB.exceptionCaught");
        //异常向下传播
        ctx.fireExceptionCaught(cause);
    }
}

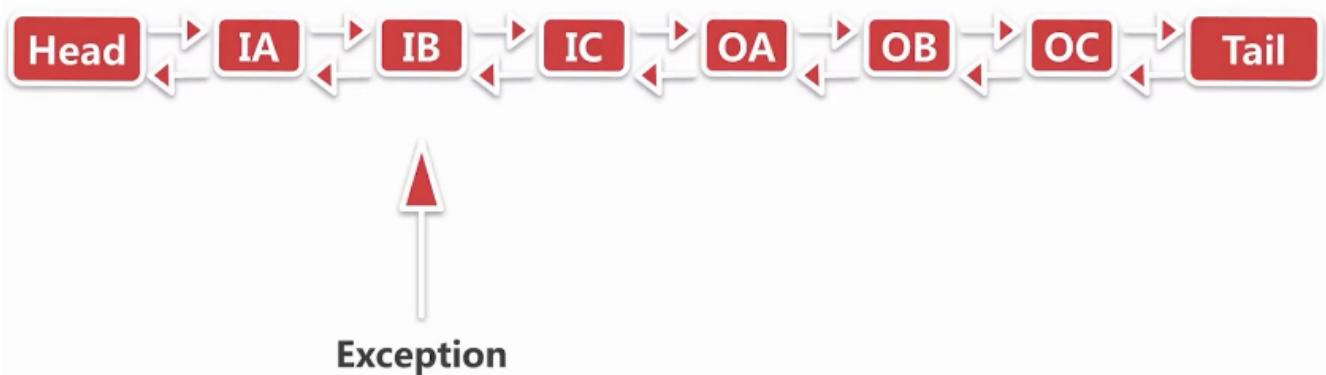
```

### 6.8.1 netty 异常的触发链

- netty异常的传播与channelHandler的添加顺序正相关，当其中某个channelHandler读写数据发生异常，就会把异常从当前节点开始逐个向下传播，如果传播到最后一个节点没有异常处理器的话，最终到Tail节点，tail节点默认会给一个告警，并进行异常信息的打印。

暂停

## 异常事件的传播

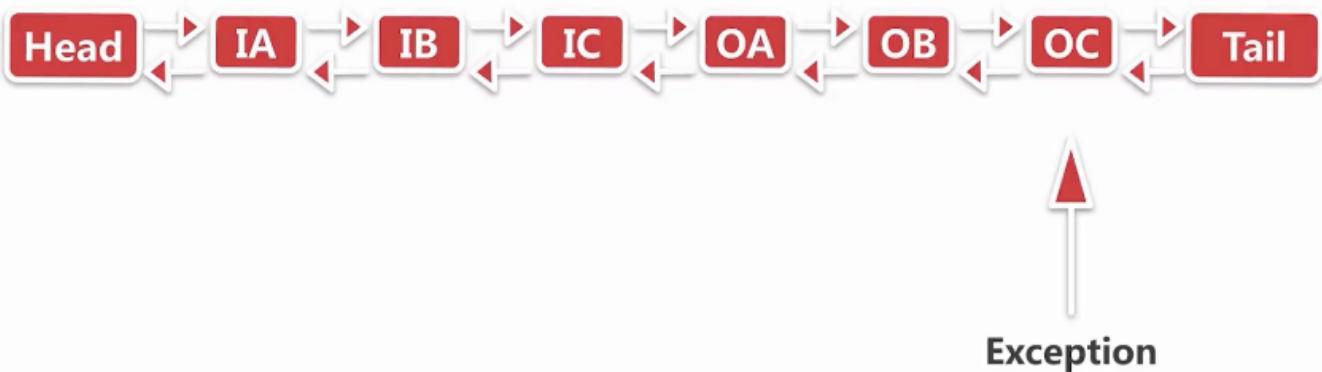


io.netty.channel.AbstractChannelHandlerContext#fireExceptionCaught

```
//向下传播直接拿到当前的节点的next节点
public ChannelHandlerContext fireExceptionCaught(final Throwable cause) {
    invokeExceptionCaught(next, cause);
    return this;
}
```

3.暂停

## 异常事件的传播



直到传播到Tail节点--> 相当于哨兵

io.netty.channel.DefaultChannelPipeline.TailContext#exceptionCaught

```
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {
    onUnhandledInboundException(cause);
}
```

```
io.netty.channel.DefaultChannelPipeline#onUnhandledInboundException
protected void onUnhandledInboundException(Throwable cause) {
    try {
        logger.warn(
            "An exceptionCaught() event was fired, and it reached at the tail of the
pipeline. " +
            "It usually means the last handler in the pipeline did not handle
the exception.", cause);
    } finally {
        ReferenceCountUtil.release(cause);
    }
}
```

### 6.8.2 netty 实际项目中异常处理最佳实践

在每一条channelHandler链的最后添加终极的异常处理器，确保所有的异常若中途没有被处理，都会落到此处理器，并且可以针对不同的异常类型分别处理

```
public class ExceptionCaughtHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception
    {
        //根据异常的类型进行处理
        if (cause instanceof BusinessException) {
            System.out.println("BusinessException");
        }
    }
}
```

```
public final class TestExceptionSpread {

    public static void main(String[] args) {

        EventLoopGroup bossGroup = new NioEventLoopGroup(1);

        EventLoopGroup workGroup = new NioEventLoopGroup();

        ServerBootstrap bootstrap = new ServerBootstrap();

        bootstrap.group(bossGroup, workGroup)
            .channel(NioServerSocketChannel.class)
            .childOption(ChannelOption.TCP_NODELAY, true)
            .childAttr(AttributeKey.newInstance("childAttr"), "childAttrValue")
            .childHandler(new ChannelInitializer<SocketChannel>() {

                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline().addLast(new ExceptionInboundHandlerA());
                    ch.pipeline().addLast(new ExceptionInboundHandlerB());
                    ch.pipeline().addLast(new ExceptionInboundHandlerC());
                    ch.pipeline().addLast(new ExceptionOutboundHandlerA());
                    ch.pipeline().addLast(new ExceptionOutboundHandlerB());
                    ch.pipeline().addLast(new ExceptionOutboundHandlerC());
                    //添加终极异常处理器
                    ch.pipeline().addLast(new ExceptionCaughtHandler());
                }
            });
        try {
            //服务端创建的入口 bind()
            ChannelFuture channelFuture = bootstrap.bind(8888).sync();
            channelFuture.channel().closeFuture().sync();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

//output:
ExceptionInboundHandlerB.exceptionCaught
ExceptionInboundHandlerC.exceptionCaught
ExceptionOutboundHandlerA.exceptionCaught
```

```
ExceptionOutboundHandlerB.exceptionCaught  
ExceptionOutboundHandlerC.exceptionCaught  
BusinessException
```

## 七、netty - byteBuf

本章是netty 内存分配 相关的内容, byteBuf 是直接与底层i/o打交道的一层抽象。

### 7.1 思考问题

- 内存的类别有哪些
  - 堆内内存 和堆外内存
- 如何减少多线程内存分配之间的竞争
- 不同大小的内存是如何进行分配的

### 7.2 本章主要内容

- 内存与内存管理器的抽象
- 不同规格大小和不同类别的内存的分配策略
- 内存的回收过程

### 7.3 byteBuf的结构以及重要API

#### 7.3.1 byteBuf结构

```
* <pre>  
*   +-----+-----+-----+  
*   | discardable bytes | readable bytes | writable bytes |  
*   |                 | (CONTENT)      |                 |  
*   +-----+-----+-----+  
*   |           |           |           |  
*   0       <=     readerIndex    <=     writerIndex    <=     capacity  (<= MaxCapacity)  
* </pre>
```

readerIndex :若读操作，从当前指针开始读数据

writerIndex: 若写操作，从当前指针开始写

0 - readerIndex 无效的数据

readerIndex - writerIndex 可读的数据

writerIndex - capacity 表明这段空闲的，可以进行写

capacity - MaxCapacity 当写的空间不足，netty会提前将 writable bytes 部分会进行扩容，MaxCapacity 表示最大能扩充的空间，若仍超过此空间，则会拒绝

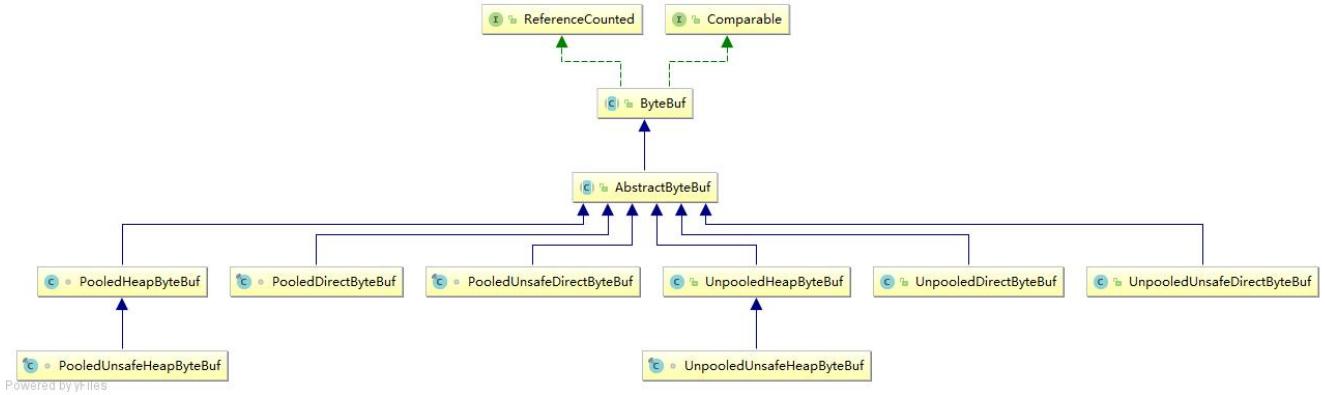
#### 7.3.2 read , write , set方法

#### 7.3.3 mark 和reset方法

### 7.4 byteBuf的分类

- pooled 和 unpooled
  - pooled 从池子中取一段预先分配好的内存

- unpooled 直接调用现有的API去分配一块内存
- unsafe 和 非unsafe
  - unsafe 依赖jdk底层的unsafe对象
  - 非unsafe 不依赖jdk底层的unsafe对象
- Heap 和 Direct

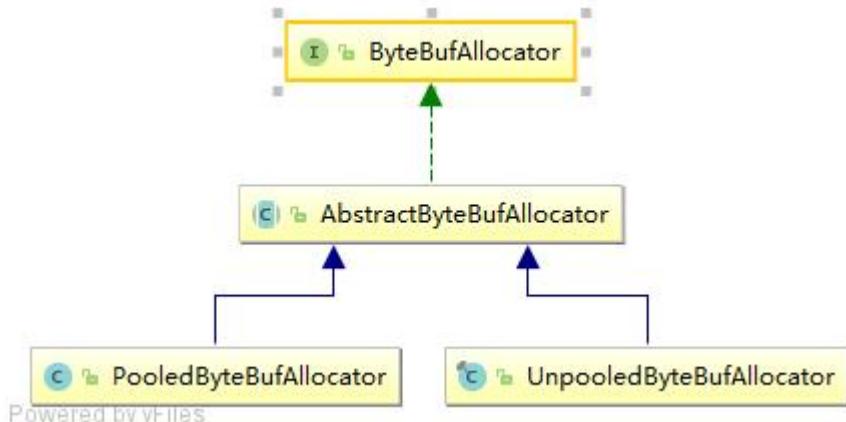


```

/**
A skeletal implementation of a buffer. 实现基本骨架
*/
io.netty.buffer.AbstractByteBuf
public abstract class AbstractByteBuf extends ByteBuf {
    ...
    int readerIndex;
    int writerIndex;
    private int markedReaderIndex;
    private int markedWriterIndex;
    private int maxCapacity;
    ...
}
  
```

## 7.5 内存分配器 ByteBufAllocator分析

思考：ByteBufAllocator是如何对上节中6中类型的byteBuf是如何分配的



最顶层抽象 io.netty.buffer.ByteBufAllocator 负责分配所有类型的内存

- ByteBufAllocator

- AbstractByteBufAllocator
  - 暴露两个方法交给具体子类实现

```
/** 堆上内存
 * Create a heap {@link ByteBuf} with the given initialCapacity and
maxCapacity.
 */
protected abstract ByteBuf newHeapBuffer(int initialCapacity, int
maxCapacity);

/** 堆外内存
 * Create a direct {@link ByteBuf} with the given initialCapacity and
maxCapacity.
 */
protected abstract ByteBuf newDirectBuffer(int initialCapacity, int
maxCapacity);
```

- ByteBufAllocator 两大子类
  - PooledByteBufAllocator
  - UnpooledByteBufAllocator

## 7.6 UnpooledByteBufAllocator 分析

- heap内存的分配

io.netty.buffer.UnpooledByteBufAllocator#newHeapBuffer

```
protected ByteBuf newHeapBuffer(int initialCapacity, int maxCapacity) {
    // netty自己判断是否有unsafe对象
    return PlatformDependent.hasUnsafe() ? new UnpooledUnsafeHeapByteBuf(this,
initialCapacity, maxCapacity)
        : new UnpooledHeapByteBuf(this, initialCapacity, maxCapacity);
}
```

- direct内存的分配

```
protected ByteBuf newDirectBuffer(int initialCapacity, int maxCapacity) {
    ByteBuf buf = PlatformDependent.hasUnsafe() ?
        UnsafeByteBufUtil.newUnsafeDirectByteBuf(this, initialCapacity,
maxCapacity) :
        new UnpooledDirectByteBuf(this, initialCapacity, maxCapacity);

    return disableLeakDetector ? buf : toLeakAwareBuffer(buf);
}
```

```
io.netty.buffer.UnpooledUnsafeDirectByteBuf#setByteBuffer
final void setByteBuffer(ByteBuffer buffer, boolean tryFree) {
    if (tryFree) {
        ByteBuffer oldBuffer = this.buffer;
        if (oldBuffer != null) {
            if (doNotFree) {
```

```
        doNotFree = false;
    } else {
        freeDirect(oldBuffer);
    }
}
}

this.buffer = buffer;
//获取buffer内存地址并保存
memoryAddress = PlatformDependent.directBufferAddress(buffer);
tmpNioBuf = null;
capacity = buffer.remaining();
}
```

```
io.netty.util.internal.PlatformDependent#directBufferAddress
public static long directBufferAddress(ByteBuffer buffer) {
    return PlatformDependent0.directBufferAddress(buffer);
}
```

```
static long directBufferAddress(ByteBuffer buffer) {
    return getLong(buffer, ADDRESS_FIELD_OFFSET);
}
```

io.netty.util.internal.PlatformDependent0#getLong(java.lang.Object, long)

```
private static long getLong(Object object, long fieldoffset) {
    return UNSAFE.getLong(object, fieldoffset);
}
```

## 取数据

io.netty.buffer.UnpooledUnsafeDirectByteBuf#\_getByte

```
protected byte _getByte(int index) {
    return UnsafeByteBufUtil.getByte(addr(index));
}
```

```
io.netty.buffer.UnsafeByteBufUtil#getByte(long)
static byte getByte(long address) {
    return PlatformDependent.getByte(address);
}
```

```
io.netty.util.internal.PlatformDependent0#getByte(long)
static byte getByte(long address) {
    return UNSAFE.getByte(address);
}
```

io.netty.buffer.UnpooledDirectByteBuf#\_getByte

```
protected byte _getByte(int index) {
    //调用jdk的ByteBuffer的API
    return buffer.get(index);
}
```

分析：unsafe和非unsafe的认识：

非unsafe 最终会通过一个内存地址 + 偏移量的方式去拿到对应的数据；

而非unsafe会通过 数组 + 下标 调用jdk底层的ByteBuffer的api去拿数据

一般而言通过unsafe取数据要快一点。

## 7.7 PooledByteBufAllocator 分析

- 拿到线程局部缓存PoolThreadCache（多线程调用newHeapBuffer时拿到当前线程的cache）
- 在线程局部缓存的Area上进行内存分配

以 `io.netty.buffer.PooledByteBufAllocator#newHeapBuffer` 为例：

```
final class PoolThreadLocalCache extends FastThreadLocal<PoolThreadCache> {

    @Override
    protected synchronized PoolThreadCache initialValue() {
        final PoolArena<byte[]> heapArena = leastUsedArena(heapArenas);
        final PoolArena<ByteBuffer> directArena = leastUsedArena(directArenas);

        return new PoolThreadCache(
            heapArena, directArena, tinyCacheSize, smallCacheSize, normalCacheSize,
            DEFAULT_MAX_CACHED_BUFFER_CAPACITY, DEFAULT_CACHE_TRIM_INTERVAL);
    }

    .....
}
```

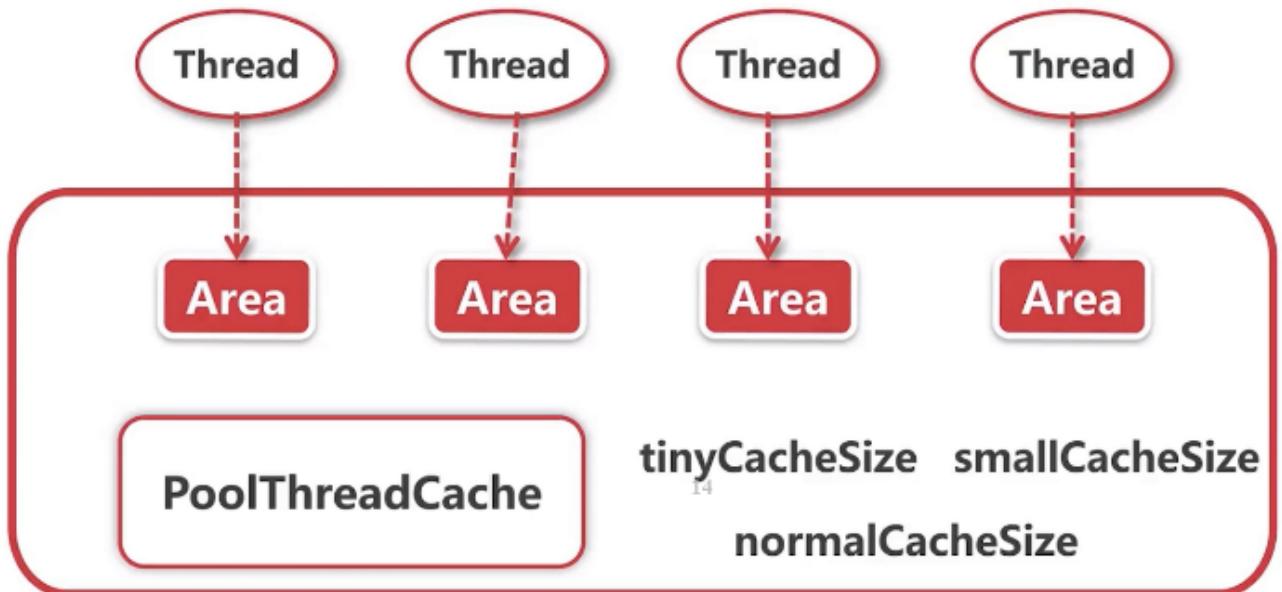
```
private final PoolThreadLocalCache threadCache;

...
protected ByteBuf newHeapBuffer(int initialCapacity, int maxCapacity) {
    PoolThreadCache cache = threadCache.get();
    PoolArena<byte[]> heapArena = cache.heapArena;

    ByteBuf buf;
    if (heapArena != null) {
        buf = heapArena.allocate(cache, initialCapacity, maxCapacity);
    } else {
        buf = new UnpooledHeapByteBuf(this, initialCapacity, maxCapacity);
    }

    return toLeakAwareBuffer(buf);
}
```

# PooledByteBufAllocator结构



## 7.8 directArena分配direct内存的流程

- 从对象池里面拿到PooledByteBuf进行复用
  - PoolThreadCache 是每一个线程都会维护的一个对象
- 从缓存上j进行内存分配
- 从内存堆里面进行内存分配

以 `io.netty.buffer.PooledByteBufAllocator#newDirectBuffer` 为例：

```
protected ByteBuf newDirectBuffer(int initialCapacity, int maxCapacity) {  
    PoolThreadCache cache = threadCache.get();  
    PoolArena<ByteBuffer> directArena = cache.directArena;  
  
    ByteBuf buf;  
    if (directArena != null) {  
        buf = directArena.allocate(cache, initialCapacity, maxCapacity);  
    } else {  
        if (PlatformDependent.hasUnsafe()) {  
            buf = UnsafeByteBufUtil.newUnsafeDirectByteBuf(this, initialCapacity,  
maxCapacity);  
        } else {  
            buf = new UnpooledDirectByteBuf(this, initialCapacity, maxCapacity);  
        }  
    }  
    return toLeakAwareBuffer(buf);  
}
```

`io.netty.buffer.PoolArena#allocate(io.netty.buffer.PoolThreadCache, int, int)`

```
PooledByteBuf<T> allocate(PoolThreadCache cache, int reqCapacity, int maxCapacity) {  
    //第一步 拿到byteBuf  
    PooledByteBuf<T> buf = newByteBuf(maxCapacity);  
    //第二步 进行内存分配 (先在缓存上进行内存分配, 若未命中缓存, 则进行实际的内存分配)  
    allocate(cache, buf, reqCapacity);  
    return buf;  
}
```

io.netty.buffer.PoolArena.DirectArena#newByteBuf

```
protected PooledByteBuf<ByteBuffer> newByteBuf(int maxCapacity) {  
    if (HAS_UNSAFE) {  
        return PooledUnsafeDirectByteBuf.newInstance(maxCapacity);  
    } else {  
        return PooledDirectByteBuf.newInstance(maxCapacity);  
    }  
}
```

io.netty.buffer.PooledUnsafeDirectByteBuf#newInstance

```
private static final Recycler<PooledUnsafeDirectByteBuf> RECYCLER = new  
Recycler<PooledUnsafeDirectByteBuf>() {  
    @Override  
    protected PooledUnsafeDirectByteBuf newObject(Handle<PooledUnsafeDirectByteBuf>  
handle) {  
        //handle负责byteBuf对象的回收  
        return new PooledUnsafeDirectByteBuf(handle, 0);  
    }  
};  
  
static PooledUnsafeDirectByteBuf newInstance(int maxCapacity) {  
    //带有回收特性的对象池  
    PooledUnsafeDirectByteBuf buf = RECYCLER.get();  
    //复用byteBuf对象  
    buf.reuse(maxCapacity);  
    return buf;  
}
```

io.netty.buffer.PooledByteBuf#reuse

```
final void reuse(int maxCapacity) {  
    maxCapacity(maxCapacity);  
    setRefCount(1);  
    setIndex0(0, 0);  
    discardMarks();  
}
```

## 7.9 内存规格的介绍

不同大小内存的分配逻辑是不一样的。

为什么把16M作为分界点？

16M对应Chunk，所有的内存申请都是通过以Chunk为单位向操作系统进行申请的，后续所有的内存分配都是在Chunk里面做对应的操作。

例如要分配一个1M的内存，首先申请一个16M的Chunk --> 在16M里面取出一段内存作为1M --> 把1M对应的连续内存扔到ByteBuf里面。

为什么把8K作为分界点？

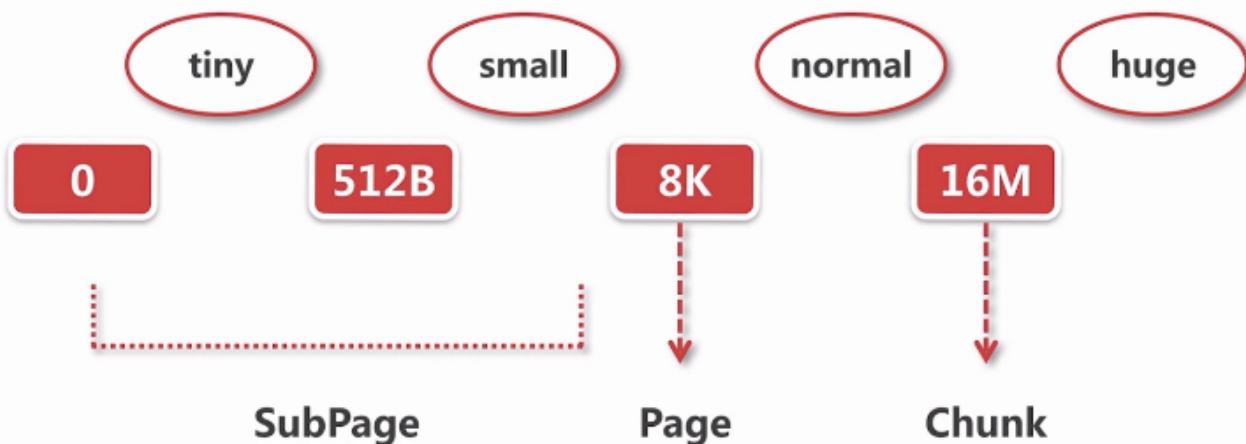
16M可能有点大，把16M的Chunk按page的方式切分， $16M/8k = 2^{11}$ 个page=2048 1page = 8K

这样想要分配16k内存的时候，只需要找到两个连续的page即可，这样的分配更高效

比如要申请16个字节，如果以page进行分配，可能会很浪费，netty按照内存大小的规格把8K继续进行切分。

若要分配512B，将8k分成8k/512个片段，每次找到一个空闲的片段进行分配即可。

## 内存规格介绍



### 7.10 缓存数据结构

netty中与缓存相关的数据结构: **MemoryRegionCache**

MemoryRegionCache由3部分组成：

- queue
  - 每一个元素是个实体entry
    - chunk 代表分配单位
    - handler 指向一段唯一连续的内存
- sizeClass netty中的内存规格
  - tiny
  - small
  - normal
- size

```
io.netty.buffer.PoolThreadCache
```

```
io.netty.buffer.PoolArena.SizeClass
```

```
enum SizeClass {  
    Tiny,  
    Small,  
    Normal  
}
```

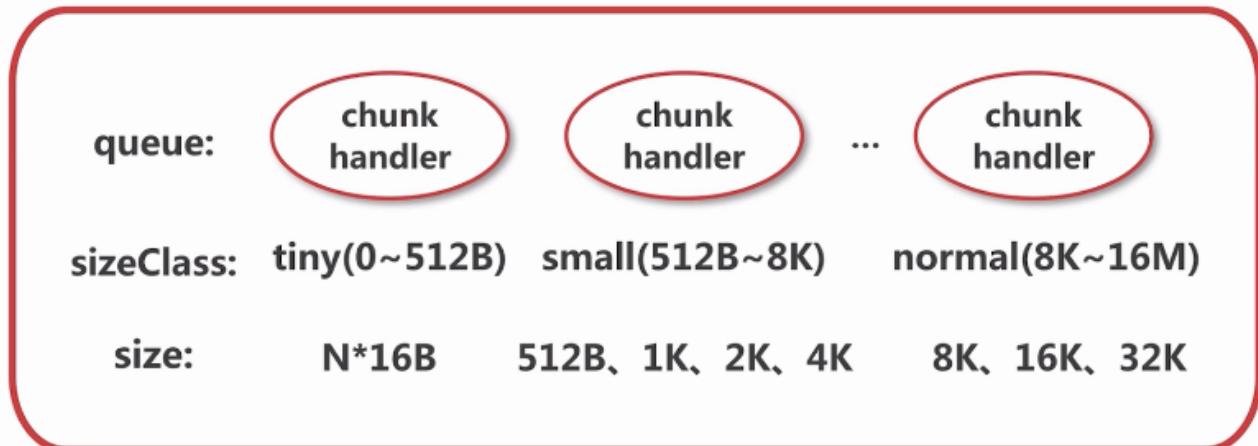
```
io.netty.buffer.PoolThreadCache.MemoryRegionCache.Entry
```

```
static final class Entry<T> {  
    final Handle<Entry<?>> recyclerHandle;  
    PoolChunk<T> chunk;  
    long handle = -1;  
  
    Entry(Handle<Entry<?>> recyclerHandle) {  
        this.recyclerHandle = recyclerHandle;  
    }  
  
    void recycle() {  
        chunk = null;  
        handle = -1;  
        recyclerHandle.recycle(this);  
    }  
}
```

```
io.netty.buffer.PoolThreadCache.MemoryRegionCache
```

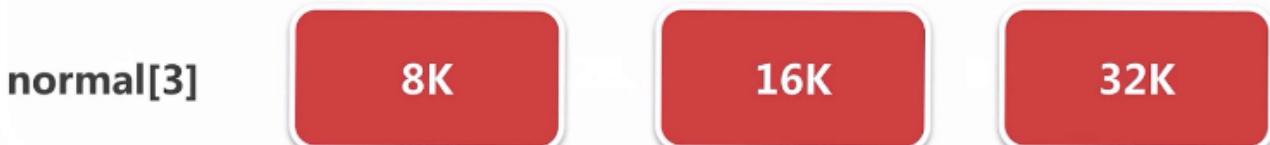
```
private abstract static class MemoryRegionCache<T> {  
    private final int size;  
    private final Queue<Entry<T>> queue;  
    private final SizeClass sizeClass;  
    private int allocations;  
  
    MemoryRegionCache(int size, SizeClass sizeClass) {  
        this.size = MathUtil.safeFindNextPositivePowerOfTwo(size);  
        queue = PlatformDependent.newFixedMpscQueue(this.size);  
        this.sizeClass = sizeClass;  
    }  
    ....  
}
```

# MemoryRegionCache



亭

# MemoryRegionCache



io.netty.buffer.PoolArena#numTinySubpagePools

```
//右移4位 相当于 除以 16  
static final int numTinySubpagePools = 512 >>> 4; //32
```

io.netty.buffer.PoolThreadCache#PoolThreadCache

```
final class PoolThreadCache {  
  
    private static final InternalLogger logger =  
InternalLoggerFactory.getInstance(PoolThreadCache.class);  
  
    //Arena 竞技场 直接开辟一块内存
```

```

final PoolArena<byte[]> heapArena;
final PoolArena<ByteBuffer> directArena;

// Hold the caches for the different size classes, which are tiny, small and normal.
//缓存一块连续内存
private final MemoryRegionCache<byte[][]> tinySubPageHeapCaches;
private final MemoryRegionCache<byte[][]> smallSubPageHeapCaches;
private final MemoryRegionCache<ByteBuffer[]> tinySubPageDirectCaches;
private final MemoryRegionCache<ByteBuffer[]> smallSubPageDirectCaches;
private final MemoryRegionCache<byte[][]> normalHeapCaches;
private final MemoryRegionCache<ByteBuffer[]> normalDirectCaches;
...
}

```

```

tinySubPageDirectCaches = createSubPageCaches(
    tinyCacheSize, PoolArena.numTinySubpagePools, sizeClass.Tiny);

```

小结：每一个线程中维护一个PoolThreadCache，PoolThreadCache会维护三种内存规格大小的cache，每种规格的cache又进行细分，其数组中的每一个元素都维持缓存队列，即MemoryRegionCache。

## 7.11 命中缓存的分配流程

- 找到对应size的MemoryRegionCache
- 从queue中弹出一个entry给ByteBuf初始化
- 将弹出的entry扔到对象池中进行复用
  - 通过Recycle回收，减少GC，减少对象重复的创建与销毁

内存分配的入口：

```
io.netty.buffer.PoolArena#allocate(io.netty.buffer.PoolThreadCache, io.netty.buffer.PooledByteBuf, int)
```

```

// capacity < pageSize
boolean isTinyOrSmall(int normCapacity) {
    return (normCapacity & subpageOverflowMask) == 0;
}

// normCapacity < 512
static boolean isTiny(int normCapacity) {
    return (normCapacity & 0xFFFFE00) == 0;
}

private void allocate(PoolThreadCache cache, PooledByteBuf<T> buf, final int reqCapacity) {
    final int normCapacity = normalizeCapacity(reqCapacity);
    if (isTinyOrSmall(normCapacity)) { // capacity < pageSize
        int tableIdx;
        PoolSubpage<T>[] table;
        boolean tiny = isTiny(normCapacity);
        if (tiny) { // < 512
            //第一种
            if (cache.allocateTiny(this, buf, reqCapacity, normCapacity)) {
                // was able to allocate out of the cache so move on

```

```

        return;
    }
    tableIdx = tinyIdx(normCapacity);
    table = tinySubpagePools;
} else {
    //第二种
    if (cache.allocateSmall(this, buf, reqCapacity, normCapacity)) {
        // was able to allocate out of the cache so move on
        return;
    }
    tableIdx = smallIdx(normCapacity);
    table = smallSubpagePools;
}

final PoolSubpage<T> head = table[tableIdx];

synchronized (head) {
    final PoolSubpage<T> s = head.next;
    if (s != head) {
        assert s.doNotDestroy && s.elemSize == normCapacity;
        long handle = s.allocate();
        assert handle >= 0;
        s.chunk.initBufWithSubpage(buf, handle, reqCapacity);

        if (tiny) {
            allocationsTiny.increment();
        } else {
            allocationsSmall.increment();
        }
        return;
    }
}
allocateNormal(buf, reqCapacity, normCapacity);
return;
}
if (normCapacity <= chunksize) {
    //第三种
    if (cache.allocateNormal(this, buf, reqCapacity, normCapacity)) {
        // was able to allocate out of the cache so move on
        return;
    }
    allocateNormal(buf, reqCapacity, normCapacity);
} else {
    // Huge allocations are never served via the cache so just call allocateHuge
    allocateHuge(buf, reqCapacity);
}
}
}

```

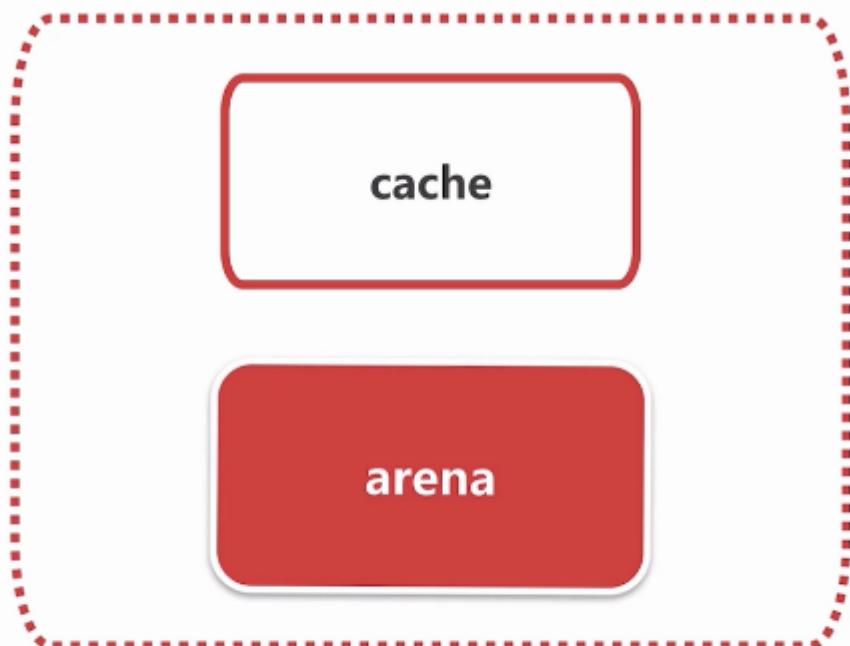
io.netty.buffer.PoolThreadCache.MemoryRegionCache#allocate

```
public final boolean allocate(PooledByteBuf<T> buf, int reqCapacity) {  
    Entry<T> entry = queue.poll();  
    if (entry == null) {  
        return false;  
    }  
    initBuf(entry.chunk, entry.handle, buf, reqCapacity);  
    entry.recycle();  
  
    // allocations is not thread-safe which is fine as this is only called from the same  
    thread all time.  
    ++ allocations;  
    return true;  
}
```

io.netty.util.Recycler.DefaultHandle#recycle

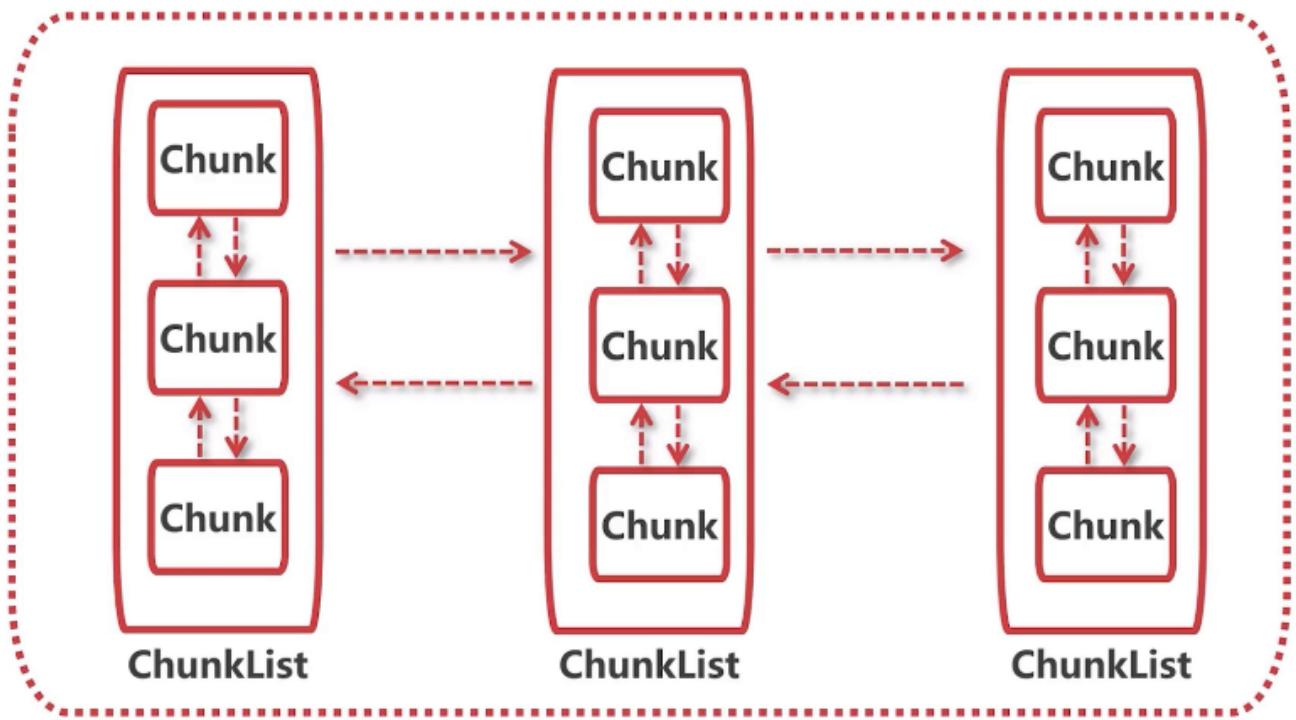
```
public void recycle(Object object) {  
    if (object != value) {  
        throw new IllegalArgumentException("object does not belong to handle");  
    }  
    stack.push(this);  
}
```

## 7.12 Arena、Chunk、Page、Subpage概念



# PoolThreadCache

Arena数据结构：



## chunk

```
abstract class PoolArena<T> implements PoolArenaMetric {
    static final boolean HAS_UNSAFE = PlatformDependent.hasUnsafe();

    enum SizeClass {
        Tiny,
        Small,
        Normal
    }

    static final int numTinySubpagePools = 512 >>> 4;
```

```

final PooledByteBufAllocator parent;

private final int maxOrder;
final int pageSize;
final int pageShifts;
final int chunkSize;
final int subpageOverflowMask;
final int numSmallSubpagePools;
private final PoolSubpage<T>[] tinySubpagePools;
private final PoolSubpage<T>[] smallSubpagePools;

private final PoolChunkList<T> q050;
private final PoolChunkList<T> q025;
private final PoolChunkList<T> q000;
private final PoolChunkList<T> qInit;
private final PoolChunkList<T> q075;
private final PoolChunkList<T> q100;
...
}

}

```

```

//PoolSubpage 数据结构 双向链表
final class PoolSubpage<T> implements PoolSubpageMetric {
    ...
    PoolSubpage<T> prev;
    PoolSubpage<T> next;
    ...
}

```

内存分配的流程：

从一个线程的PoolThreadCache中获取对应的Arena，Arena再通过PoolChunkList里面去一个chunk，

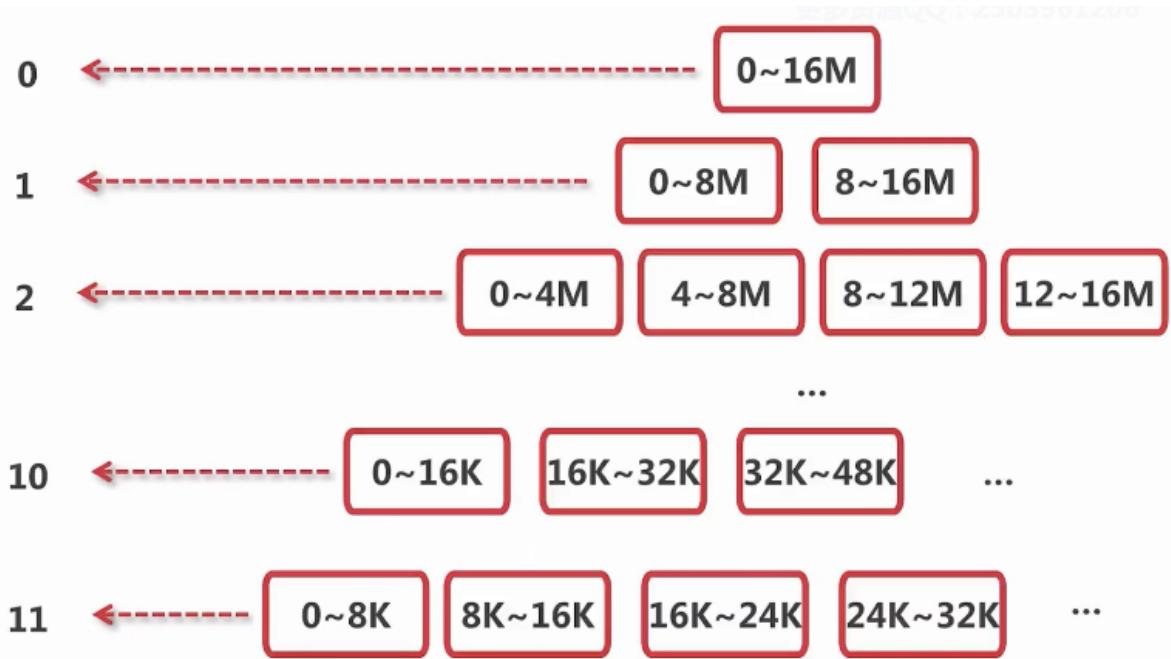
chunk会判断内存分配大小：

超过1个page，则以page为单位；若远小于一个page，则将一个page切分成多个子page进行内存划分。

### 7.13 page 级别内存分配 allocateNormal()

- 尝试在现有的chunk上分配
- 创建一个chunk进行内存分配
- 初始化pooledByteBuf

一个chunk最终是以K为单位组织内存的，chunk中的每一个节点可以标识对应的这块内存是否被使用



io.netty.buffer.PoolArena#allocateNormal

```

private synchronized void allocateNormal(PooledByteBuf<T> buf, int reqCapacity, int
normCapacity) {
    if (q050.allocate(buf, reqCapacity, normCapacity) || q025.allocate(buf, reqCapacity,
normCapacity) ||
        q000.allocate(buf, reqCapacity, normCapacity) || qInit.allocate(buf, reqCapacity,
normCapacity) ||
        q075.allocate(buf, reqCapacity, normCapacity)) {
        ++allocationsNormal;
        return;
    }

    // Add a new chunk.
    PoolChunk<T> c = newChunk(pageSize, maxorder, pageshifts, chunksize);
    long handle = c.allocate(normCapacity);
    ++allocationsNormal;
    assert handle > 0;
    c.initBuf(buf, handle, reqCapacity);
    qInit.add(c);
}

```

## 7.14 subpage 级别的内存分配 allocateTiny()

- 定位一个Subpage对象
- 初始化Subpage
- 初始化PooledByteBuf

io.netty.buffer.PoolChunk#allocateSubpage

```

private long allocateSubpage(int normCapacity) {
    // Obtain the head of the PoolSubPage pool that is owned by the PoolArena and
    // synchronize on it.
    // This is need as we may add it back and so alter the linked-list structure.
    PoolSubpage<T> head = arena.findSubpagePoolHead(normCapacity);
    synchronized (head) {
        int d = maxOrder; // subpages are only be allocated from pages i.e., leaves
        int id = allocateNode(d);
        if (id < 0) {
            return id;
        }

        final PoolSubpage<T>[] subpages = this.subpages;
        final int pageSize = this.pageSize;

        freeBytes -= pageSize;

        int subpageIdx = subpageIdx(id);
        PoolSubpage<T> subpage = subpages[subpageIdx];
        if (subpage == null) {
            subpage = new PoolSubpage<T>(head, this, id, runOffset(id), pageSize,
normCapacity);
            subpages[subpageIdx] = subpage;
        } else {
            subpage.init(head, normCapacity);
        }
        return subpage.allocate();
    }
}

```

## io.netty.buffer.PoolSubpage#init

```

PoolSubpage(PoolSubpage<T> head, PoolChunk<T> chunk, int memoryMapIdx, int runOffset, int
pageSize, int elemSize) {
    this.chunk = chunk;
    this.memoryMapIdx = memoryMapIdx;
    this.runOffset = runOffset;
    this.pageSize = pageSize;
    bitmap = new long[pageSize >>> 10]; // pageSize / 16 / 64
    init(head, elemSize);
}

void init(PoolSubpage<T> head, int elemSize) {
    doNotDestroy = true;
    this.elemSize = elemSize;
    if (elemSize != 0) {
        maxNumElems = numAvail = pageSize / elemSize;
        nextAvail = 0;
        bitmapLength = maxNumElems >>> 6;
        if ((maxNumElems & 63) != 0) {
            bitmapLength++;
        }
    }
}

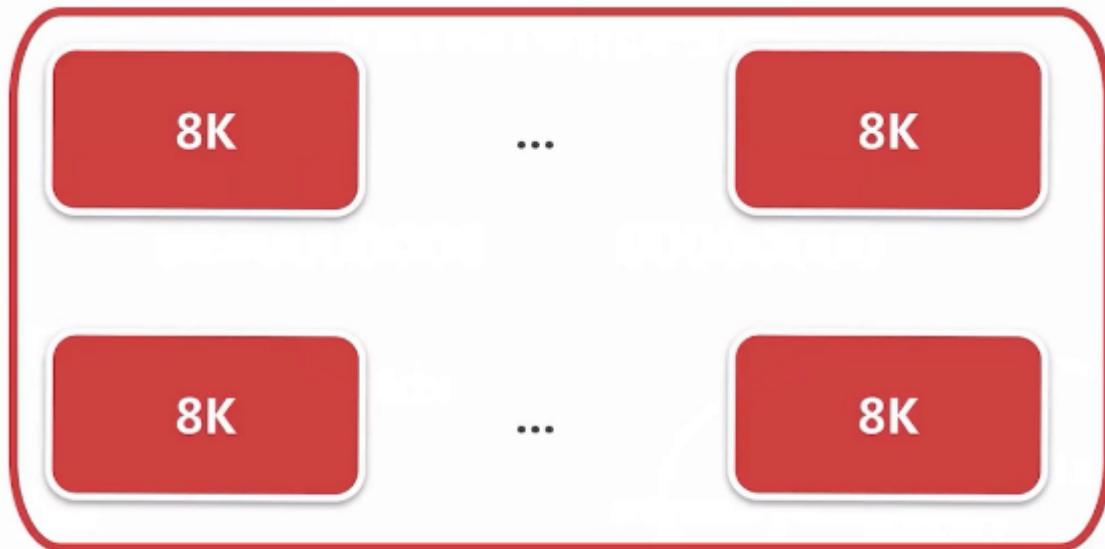
```

```
        for (int i = 0; i < bitmapLength; i++) {  
            bitmap[i] = 0;//标识哪个子page已经被分配  
        }  
    }  
    addToPool(head);  
}
```

## tinySubpagePools



chunk中的subpages



## chunk中的subpages

io.netty.buffer.PoolSubpage#allocate

```
//从位图中找到一个未被使用的subpage
long allocate() {
    if (elemSize == 0) {
        return toHandle(0);
    }

    if (numAvail == 0 || !doNotDestroy) {
        return -1;
    }

    final int bitmapIdx = getNextAvail();
    int q = bitmapIdx >>> 6;
    int r = bitmapIdx & 63;
    assert (bitmap[q] >>> r & 1) == 0;
    bitmap[q] |= 1L << r;

    if (-- numAvail == 0) {
        removeFromPool();
    }
    return toHandle(bitmapIdx);
}
```

io.netty.buffer.PoolSubpage#toHandle

```
private long toHandle(int bitmapIdx) {
    return 0x4000000000000000L | (long) bitmapIdx << 32 | memoryMapIdx;
}
```

Handle的构成\*\*

# handle的构成

0x40000000      00000000

或

**bitmapIdx**

或

**memoryMapIdx**

等价于

**bitmapIdx** **memoryMapIdx** 拼接

14

io.netty.buffer.PoolChunk#initBufWithSubpage(io.netty.buffer.PooledByteBuf, long, int, int)

```
private void initBufwithSubpage(PooledByteBuf<T> buf, long handle, int bitmapIdx, int reqCapacity) {
    assert bitmapIdx != 0;

    int memoryMapIdx = memoryMapIdx(handle);

    PoolSubpage<T> subpage = subpages[subpageIdx(memoryMapIdx)];
    assert subpage.doNotDestroy;
    assert reqCapacity <= subpage.elemSize;

    buf.init(
        this, handle,
        //page的偏移量 + subpage的偏移量 = 整个内存的偏移量 再给到bytebuf,
        //这样bytebuf就可以基于这个偏移量进行读写
        runOffset(memoryMapIdx) + (bitmapIdx & 0x3FFFFFF) * subpage.elemSize, reqCapacity,
        subpage.elemSize,
        arena.parent.threadCache());
}
```

7.15 ByteBuf的回收

```
public class AllocatorDemo {  
    public static void main(String[] args) {  
        PooledByteBufAllocator allocator = PooledByteBufAllocator.DEFAULT;  
        //申请一块内存  
        ByteBuf byteBuf = allocator.directBuffer(16);  
        //释放内存  
        byteBuf.release();  
    }  
}
```

io.netty.buffer.AbstractReferenceCountedByteBuf#release()

```
public boolean release() {  
    return release0(1);  
}
```

io.netty.buffer.PooledByteBuf#deallocate

```
protected final void deallocate() {  
    if (handle >= 0) {  
        final long handle = this.handle;  
        this.handle = -1;  
        memory = null;  
        chunk.arena.free(chunk, handle, maxLength, cache);  
        recycle();  
    }  
}
```

### ByteBuf的释放分析：

- 连续的内存区段加到缓存

io.netty.buffer.PoolThreadCache#cache(io.netty.buffer.PoolArena<?>, int,  
io.netty.buffer.PoolArena.SizeClass)

```
boolean add(PoolArena<?> area, PoolChunk chunk, long handle, int normCapacity,  
SizeClass sizeClass) {  
    MemoryRegionCache<?> cache = cache(area, normCapacity, sizeClass);  
    if (cache == null) {  
        return false;  
    }  
    return cache.add(chunk, handle);  
}  
  
private MemoryRegionCache<?> cache(PoolArena<?> area, int normCapacity, SizeClass  
sizeClass) {  
    switch (sizeClass) {  
    case Normal:  
        return cacheForNormal(area, normCapacity);  
    case Small:  
        return cacheForSmall(area, normCapacity);  
    }
```

```

        case Tiny:
            return cacheForTiny(area, normCapacity);
        default:
            throw new Error();
    }
}

```

io.netty.buffer.PoolThreadCache.MemoryRegionCache#add

```

public final boolean add(PoolChunk<T> chunk, long handle) {
    Entry<T> entry = newEntry(chunk, handle);
    boolean queued = queue.offer(entry);
    if (!queued) {
        // If it was not possible to cache the chunk, immediately recycle the entry
        entry.recycle();
    }
    return queued;
}

```

- 标记连续的内存区段为未使用

- page级别
- subPage级别

io.netty.buffer.PoolChunkList#free

```

boolean free(PoolChunk<T> chunk, long handle) {
    chunk.free(handle);
    if (chunk.usage() < minUsage) {
        remove(chunk);
        // Move the PoolChunk down the PoolChunkList linked-list.
        return move0(chunk);
    }
    return true;
}

```

io.netty.buffer.PoolChunk#free

```

void free(long handle) {
    int memoryMapIdx = memoryMapIdx(handle);
    int bitmapIdx = bitmapIdx(handle);

    if (bitmapIdx != 0) { // free a subpage
        PoolSubpage<T> subpage = subpages[subpageIdx(memoryMapIdx)];
        assert subpage != null && subpage.doNotDestroy;

        // Obtain the head of the PoolSubPage pool that is owned by the PoolArena
        // and synchronize on it.
        // This is need as we may add it back and so alter the linked-list
        // structure.
        PoolSubpage<T> head = arena.findSubpagePoolHead(subpage.elemSize());
        synchronized (head) {

```

```

        if (subpage.free(head, bitmapIdx & 0x3FFFFFF)) {
            return;
        }
    }
    freeBytes += runLength(memoryMapIdx);
    setValue(memoryMapIdx, depth(memoryMapIdx));
    updateParentsFree(memoryMapIdx);
}

```

- ByteBuf加到对象池

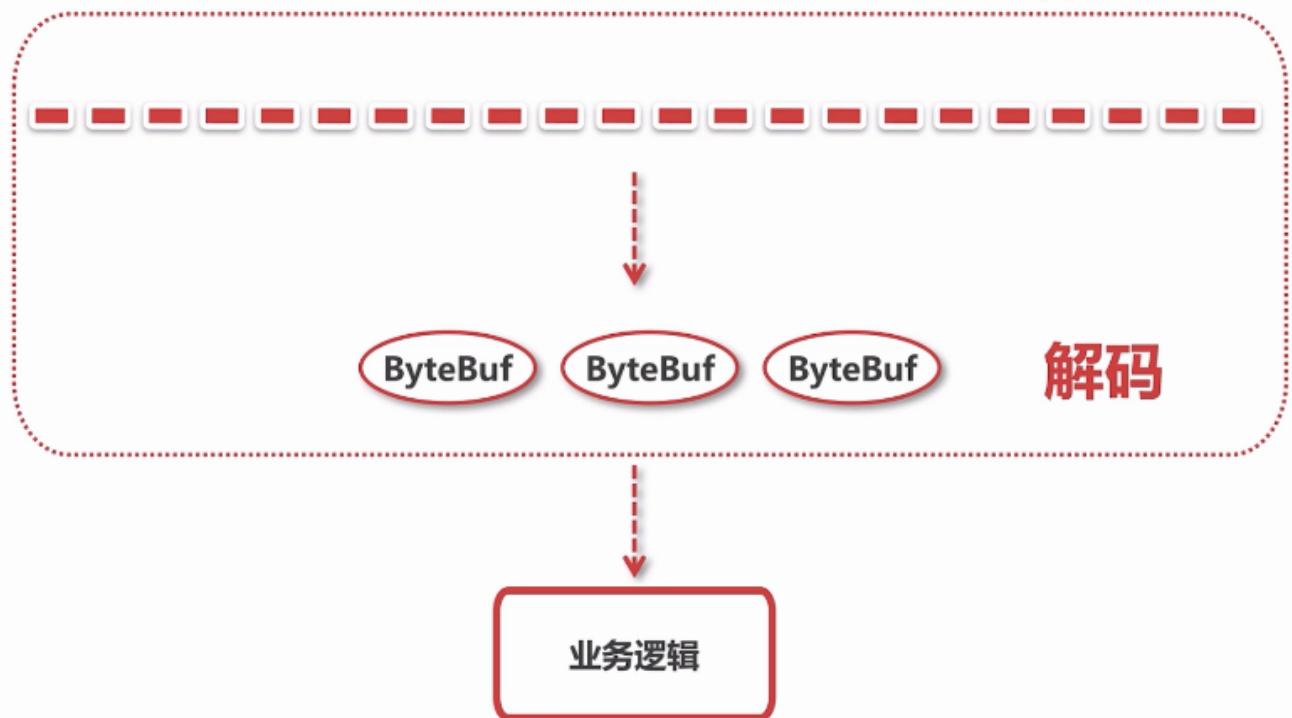
bytebuf可能会经常被申请和释放，如果qps高的时候可能会产生很多的bytebuf对象，为了尽可能的减少GC，netty使用对象池技术，当申请byteBuf的时候尽可能从对象池中取，当释放byteBuf的时候，将其放到对象池中。

## 7.16 ByteBuf 总结

- ByteBuf的api和分类
- 分配pooled内存的总步骤
- 不同规格的pooled内存分配与释放

## 八. Netty 解码

解码：把一串二级制数据流解析成一个个自定义的数据包即Bytebuf，后续业务处理都基于bytebuf。



### 8.1.1思考问题

- 解码器抽象的解码过程
- netty中有哪些拆箱即用的解码器

### 8.1.2 本章内容

- 解码器基类实现抽象解码的过程
- netty中常见的解码器分析

## 8.2 解码器的基类 - ByteToMessageDecoder解码步骤

暴漏decode抽象方法，具体实现交给子类。好处是：不同的子类的协议可以有自定义的decode方法，而最外层解码的框架都是一致的。

- 累加字节流
- 调用子类的decode方法进行解析
- 将解析到的ByteBuf向下传播

io.netty.handler.codec.ByteToMessageDecoder#channelRead

```
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
    if (msg instanceof ByteBuf) {
        CodecOutputList out = CodecOutputList.newInstance();
        try {
            ByteBuf data = (ByteBuf) msg;
            first = cumulation == null;
            if (first) {
                cumulation = data;
            } else {
                //当前累加器里的数据和读进来数据进行累加 - 注意累加器的扩容逻辑
                cumulation = cumulator.cumulate(ctx.alloc(), cumulation, data);
            }
            // out 用来接收从cumulation中解析到的对象
            callDecode(ctx, cumulation, out);
        } catch (DecoderException e) {
            throw e;
        } catch (Throwable t) {
            throw new DecoderException(t);
        } finally {
            if (cumulation != null && !cumulation.isReadable()) {
                numReads = 0;
                cumulation.release();
                cumulation = null;
            } else if (++numReads >= discardAfterReads) {
                // we did enough reads already try to discard some bytes so we not risk to
see a OOME.
                // See https://github.com/netty/netty/issues/4275
                numReads = 0;
                discardSomeReadBytes();
            }

            int size = out.size();
            decodeWasNull = !out.insertSinceRecycled();
            // 讲解析到的对象向下传播到业务解码器
            fireChannelRead(ctx, out, size);
            out.recycle();
        }
    } else {
        ctx.fireChannelRead(msg);
    }
}
```

```
    }
}
```

## io.netty.handler.codec.ByteToMessageDecoder#callDecode

```
protected void callDecode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) {
    try {
        while (in.isReadable()) {
            int outSize = out.size();

            if (outsize > 0) {
                fireChannelRead(ctx, out, outsize);
                out.clear();

                // Check if this handler was removed before continuing with decoding.
                // If it was removed, it is not safe to continue to operate on the buffer.
                //
                // See:
                // - https://github.com/netty/netty/issues/4635
                if (ctx.isRemoved()) {
                    break;
                }
                outsize = 0;
            }

            int oldInputLength = in.readableBytes();
            decode(ctx, in, out);

            // Check if this handler was removed before continuing the loop.
            // If it was removed, it is not safe to continue to operate on the buffer.
            //
            // See https://github.com/netty/netty/issues/1664
            if (ctx.isRemoved()) {
                break;
            }
            //未解析到对象
            if (outSize == out.size()) {
                if (oldInputLength == in.readableBytes()) {
                    break;
                } else {
                    continue;
                }
            }
            if (oldInputLength == in.readableBytes()) {
                throw new DecoderException(
                    StringUtil.simpleClassName(getClass()) +
                    ".decode() did not read anything but decoded a message.");
            }

            if (isSingleDecode()) {
                break;
            }
        }
    }
}
```

```

        } catch (DecoderException e) {
            throw e;
        } catch (Throwable cause) {
            throw new DecoderException(cause);
        }
    }
}

```

### 8.3 基于固定长度解码器分析

io.netty.handler.codec.FixedLengthFrameDecoder

```

/**
 * A decoder that splits the received {@link ByteBuf}s by the fixed number
 * of bytes. For example, if you received the following four fragmented packets:
 * <pre>
 * +---+---+---+---+
 * | A | BC | DEFG | HI |
 * +---+---+---+---+
 * </pre>
 * A {@link FixedLengthFrameDecoder}[@code (3)} will decode them into the
 * following three packets with the fixed length:
 * <pre>
 * +---+---+---+
 * | ABC | DEF | GHI |
 * +---+---+---+
 * </pre>
 */
public class FixedLengthFrameDecoder extends ByteToMessageDecoder {

    //按frameLength的长度为分割进行解析
    private final int frameLength;

    /**
     * Creates a new instance.
     *
     * @param frameLength the length of the frame
     */
    public FixedLengthFrameDecoder(int frameLength) {
        if (frameLength <= 0) {
            throw new IllegalArgumentException(
                    "frameLength must be a positive integer: " + frameLength);
        }
        this.frameLength = frameLength;
    }

    @Override
    protected final void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)
        throws Exception {
        Object decoded = decode(ctx, in);
        if (decoded != null) {
            out.add(decoded);
        }
    }
}

```

```

/**
 * Create a frame out of the {@link ByteBuf} and return it.
 *
 * @param ctx          the {@link ChannelHandlerContext} which this {@link ByteToMessageDecoder} belongs to
 * @param in           the {@link ByteBuf} from which to read data
 * @return frame       the {@link ByteBuf} which represent the frame or {@code null} if no frame could
 *                     be created.
 */
protected Object decode(
    @SuppressWarnings("UnusedParameters") ChannelHandlerContext ctx, ByteBuf in)
throws Exception {
    //当前累加器可读字节 < frameLength
    if (in.readableBytes() < frameLength) {
        return null;
    } else {
        //从当前累加器截取frameLength个字节进行解析
        return in.readRetainedSlice(frameLength);
    }
}
}

```

## 8.4 基于行解码器分析

io.netty.handler.codec.LineBasedFrameDecoder

发过来的字节流是以 \r \n 或直接以 \n 结尾的字节流，该解码器的功能就是以 换行符 为分割，将字节流解析成完整的数据包。

回车、换行的区别

在windows中：

‘\r’ (回车)：即将光标回到当前行的行首(而不会换到下一行)，之后的输出会把之前的输出覆盖

‘\n’ 换行，换到当前位置的下一位置，而不会回到行首；

Unix系统里，每行结尾只有“<换行>”，即"\n"；

Windows系统里面，每行结尾是“<回车><换行>”，即“\r\n”；

Mac系统里，每行结尾是“<回车>”，即"\r"；

也就是：

Linux中遇到换行符("\n")会进行回车+换行的操作，回车符 ("\\r") 反而只会作为控制字符("^\M")显示，不发生回车的操作。

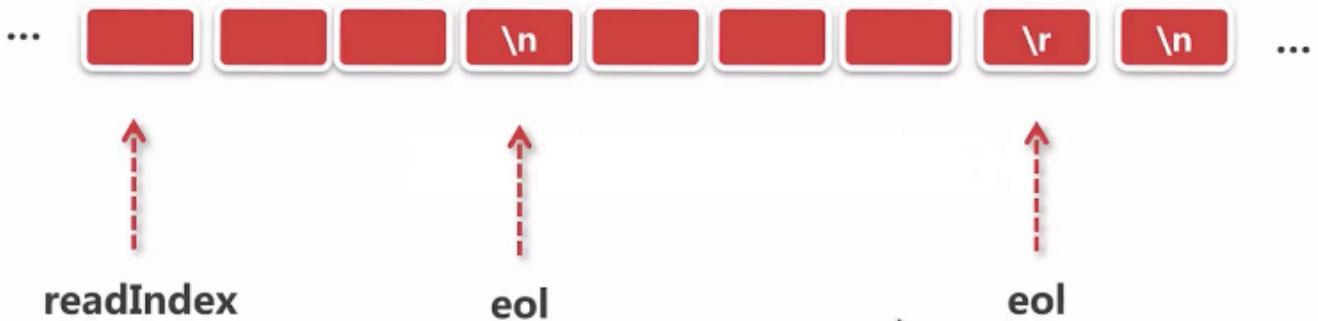
而Windows中要回车符+换行符("\\r\\n")才会回车+换行，缺少一个控制符或者顺序不对都不能正确的另起一行。

LineBasedFrameDecoder 支持两种格式的换行符；

以\n结尾 eol指向\n

若以\r\n结尾，会把\r\n当作整体作为换行符，eol指向\r

## 非丢弃模式处理



```
public class LineBasedFrameDecoder extends ByteToMessageDecoder {  
  
    /** Maximum length of a frame we're willing to decode.  
     * 能接收的最大字节  
     */  
    private final int maxLength;  
    /** Whether or not to throw an exception as soon as we exceed maxLength.  
     * 超过能接收的最大字节是否马上抛出异常  
     */  
    private final boolean failFast;  
    /**  
     * 最终解析出来的数据包是否带换行符，为true表示不带换行符  
     */  
    private final boolean stripDelimiter;  
  
    /** True if we're discarding input because we're already over maxLength. */  
    private boolean discarding;  
    private int discardedBytes;  
}  
  
.....  
  
protected final void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)  
throws Exception {  
    Object decoded = decode(ctx, in);  
    if (decoded != null) {  
        out.add(decoded);  
    }  
}
```

```
}

.....



protected Object decode(ChannelHandlerContext ctx, ByteBuf buffer) throws Exception {
    final int eol = findEndOfLine(buffer);
    if (!discarding) {
        if (eol >= 0) {
            final ByteBuf frame;
            final int length = eol - buffer.readerIndex();
            final int delimLength = buffer.getByte(eol) == '\r'? 2 : 1;

            if (length > maxLength) {
                buffer.readerIndex(eol + delimLength);
                fail(ctx, length);
                return null;
            }

            if (stripDelimiter) {
                frame = buffer.readRetainedSlice(length);
                buffer.skipBytes(delimLength);
            } else {
                frame = buffer.readRetainedSlice(length + delimLength);
            }

            return frame;
        } else {
            final int length = buffer.readableBytes();
            if (length > maxLength) {
                discardedBytes = length;
                buffer.readerIndex(buffer.writerIndex());
                discarding = true;
                if (failFast) {
                    fail(ctx, "over " + discardedBytes);
                }
            }
            return null;
        }
    } else {
        if (eol >= 0) {
            final int length = discardedBytes + eol - buffer.readerIndex();
            final int delimLength = buffer.getByte(eol) == '\r'? 2 : 1;
            buffer.readerIndex(eol + delimLength);
            discardedBytes = 0;
            discarding = false;
            if (!failFast) {
                fail(ctx, length);
            }
        } else {
            discardedBytes += buffer.readableBytes();
            buffer.readerIndex(buffer.writerIndex());
        }
        return null;
    }
}
```

```

    }
}

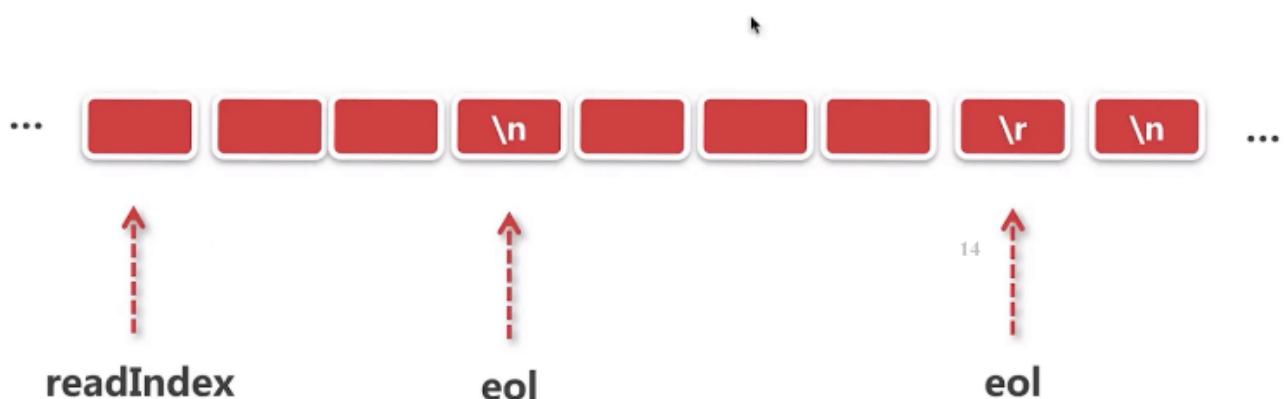
.....
/*
 * Returns the index in the buffer of the end of line found.
 * Returns -1 if no end of line was found in the buffer.
 */
private static int findEndofLine(final ByteBuf buffer) {
    int i = buffer.forEachByte(ByteProcessor.FIND_LF);
    if (i > 0 && buffer.getByte(i - 1) == '\r') {
        i--;
    }
    return i;
}


```

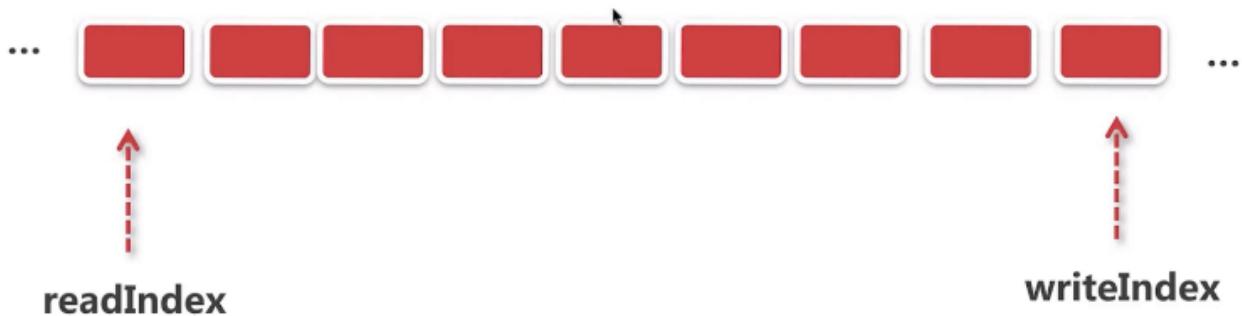
## 非丢弃模式处理



## 丢弃模式处理



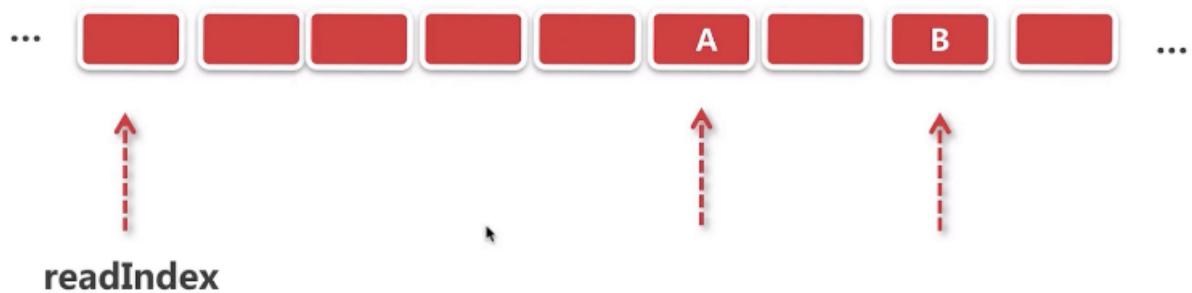
# 丢弃模式处理



## 8.5 基于固定分隔符解码器分析

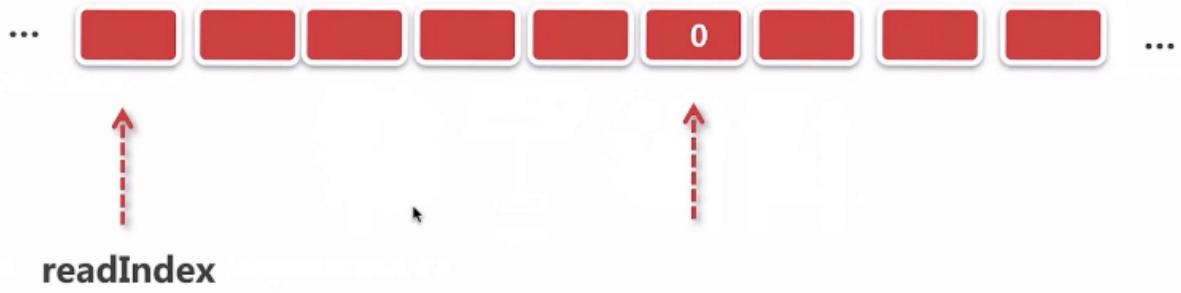
- 行处理器
  - 若分隔符为换行符则直接使用行处理器
- 找到最小分隔符

# 最小分隔符



- 解码

# 找到分隔符



# 未找到分隔符



传进来的二进制字节流带有一系列的分隔符，经过这个解码器处理，可以分拆成一个个完整的数据包。

io.netty.handler.codec.DelimiterBasedFrameDecoder

```
protected Object decode(ChannelHandlerContext ctx, ByteBuf buffer) throws Exception {
    if (lineBasedDecoder != null) {
        return lineBasedDecoder.decode(ctx, buffer);
    }
    // Try all delimiters and choose the delimiter which yields the shortest frame.
    int minFrameLength = Integer.MAX_VALUE;
    ByteBuf minDelim = null;
    for (ByteBuf delim: delimiters) {
        int frameLength = indexof(buffer, delim);
        if (frameLength >= 0 && frameLength < minFrameLength) {
            minFrameLength = frameLength;
            minDelim = delim;
        }
    }
    if (minDelim != null) {
        int minDelimLength = minDelim.capacity();
        ByteBuf frame;

        if (discardingTooLongFrame) {
            // We've just finished discarding a very large frame.
            // Go back to the initial state.
            discardingTooLongFrame = false;
            buffer.skipBytes(minFrameLength + minDelimLength);

            int tooLongFrameLength = this.tooLongFrameLength;
            this.tooLongFrameLength = 0;
            if (!failFast) {
                fail(tooLongFrameLength);
            }
        }
        return null;
    }
}
```

```

        if (minFrameLength > maxFrameLength) {
            // Discard read frame.
            buffer.skipBytes(minFrameLength + minDelimLength);
            fail(minFrameLength);
            return null;
        }

        if (stripDelimiter) {
            frame = buffer.readRetainedSlice(minFrameLength);
            buffer.skipBytes(minDelimLength);
        } else {
            frame = buffer.readRetainedSlice(minFrameLength + minDelimLength);
        }

        return frame;
    } else {
        if (!discardingTooLongFrame) {
            if (buffer.readableBytes() > maxFrameLength) {
                // Discard the content of the buffer until a delimiter is found.
                tooLongFrameLength = buffer.readableBytes();
                buffer.skipBytes(buffer.readableBytes());
                discardingTooLongFrame = true;
                if (failFast) {
                    fail(tooLongFrameLength);
                }
            }
        } else {
            // still discarding the buffer since a delimiter is not found.
            tooLongFrameLength += buffer.readableBytes();
            buffer.skipBytes(buffer.readableBytes());
        }
        return null;
    }
}

```

## 8.6 基于长度域解码器分析 - 最为常用

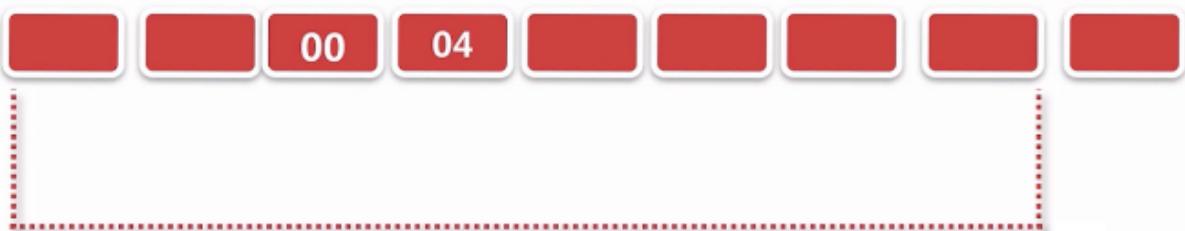
lengthFieldOffset:长度域在二进制数据流中的偏移量是多少

lengthFieldLength:从长度域开始往后的几个字节组合起来表示长度

lengthAdjustment:

initialBytesToStrip:

# 重要参数



**lengthFieldOffset: 2**

**lengthFieldLength: 2**

io.netty.handler.codec.LengthFieldBasedFrameDecoder

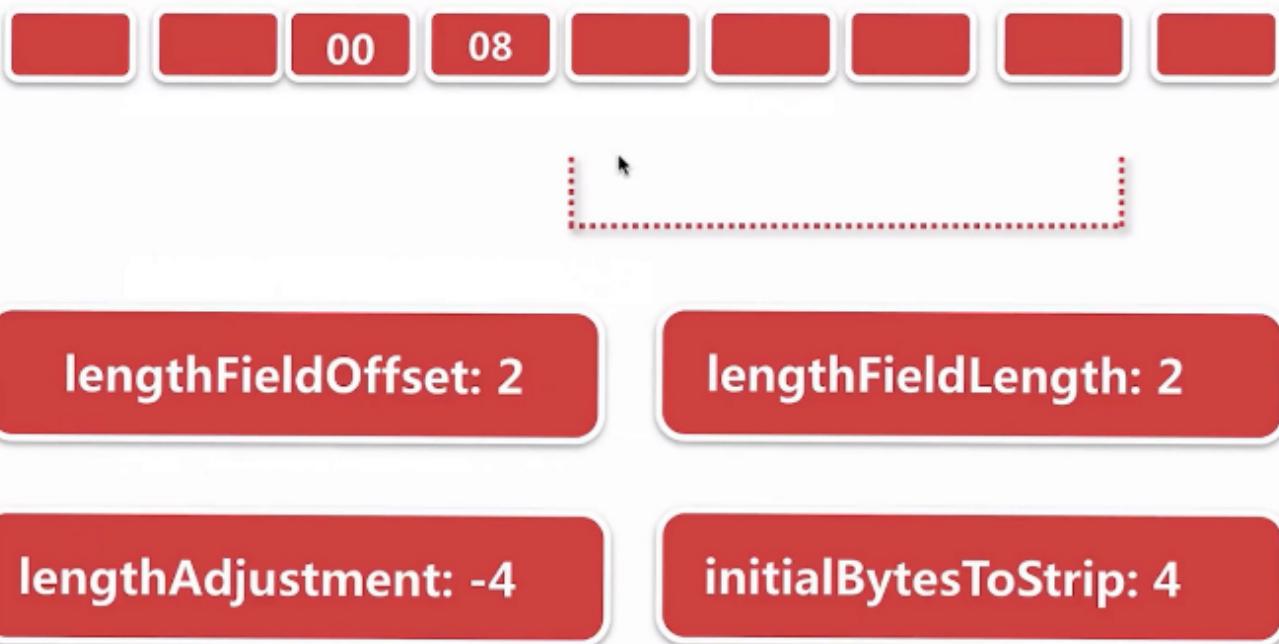
```
/**  
 * <b>lengthFieldOffset</b>    = <b>0</b>  
 * <b>lengthFieldLength</b>    = <b>2</b>  
 * lengthAdjustment        = 0  
 * initialBytesToStrip = 0 (= do not strip header)  
 *  
 * BEFORE DECODE (14 bytes)          AFTER DECODE (14 bytes)  
 * +-----+-----+          +-----+-----+  
 * | Length | Actual Content |----->| Length | Actual Content |  
 * | 0x000C | "HELLO, WORLD" |          | 0x000C | "HELLO, WORLD" |  
 * +-----+-----+          +-----+-----+  
 * ....  
 */  
public class LengthFieldBasedFrameDecoder extends ByteToMessageDecoder {  
  
    private final ByteOrder byteOrder;  
    private final int maxFrameLength;  
    private final int lengthFieldOffset;  
    private final int lengthFieldLength;  
    private final int lengthFieldEndOffset;  
    private final int lengthAdjustment;  
    private final int initialBytesToStrip;  
    private final boolean failFast;  
    private boolean discardingTooLongFrame;  
    private long tooLongFrameLength;  
    private long bytesToDiscard;  
}
```

## 8.7 基于长度域解码器步骤

- 计算要抽取的数据包长度

- 跳过字节逻辑处理
- 丢弃模式下的处理

## 长度域解码示例



### 8.8 解码器总结

## 九. netty 编码

### 9.1 问题：如何把对象变成字节流，最终写到socket底层

- encode() —— 分别ByteBuf对象，encoder调用抽象的encode方法将接收到的数据填充到ByteBuf对象
- write() —— 传递到Head节点，通过底层unsafe将当前的byteBuf塞到Unsafe底层维护的一个缓冲区对象中，同时计算写缓冲区数据是否超过高水位，若写缓冲区中可写的字计数 > 64 K, 就改变channel的状态为不可写
- flush() —— 传递到Head节点，调用unsafe的flush方法，调整指针 -》 通过for循环不断从写缓冲区拿 byteBuf -》 转化成JDK底层可以接收的ByteBuffer-》 通过JDK的channel将byteBuffer写出去-》 将写完一个将缓冲区中当前节点进行删除-》 若写缓冲区中可写的字计数 < 32 K, 就改变channel的状态为可写

# writeAndFlush()



```
/**  
 * @Auther: leh  
 * @Date: 2019/11/15 15:55  
 * @Description: 将user对象编码  
 * -----  
 * | 4 | 4 | ? |  
 * -----  
 * | length | age | name |  
 * -----  
 */  
public class Encoder extends MessageToByteEncoder<User> {  
    @Override  
    protected void encode(ChannelHandlerContext ctx, User user, ByteBuf out) throws  
Exception {  
        byte[] bytes = user.getName().getBytes();  
        out.writeInt(4 + bytes.length);  
        out.writeInt(user.getAge());  
        out.writeBytes(bytes);  
    }  
}
```

## 9.2 writeAndFlush 抽象步骤

- 从tail节点开始往前传播
- 逐个调用channelHanler的write方法
- 逐个调用channelHandler的flush方法 - 真正写到底层去

```
public class BizHandler extends ChannelInboundHandlerAdapter {  
    @Override  
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {  
        User user = new User(18, "zhangsan");  
        ctx.channel().writeAndFlush(user);  
    }  
}
```

```
io.netty.channel.AbstractChannelHandlerContext#writeAndFlush(java.lang.Object,  
io.netty.channel.ChannelPromise)
```

```
public ChannelFuture writeAndFlush(Object msg, ChannelPromise promise) {  
    if (msg == null) {  
        throw new NullPointerException("msg");  
    }  
  
    if (!validatePromise(promise, true)) {  
        ReferenceCountUtil.release(msg);  
        // cancelled  
        return promise;  
    }  
  
    write(msg, true, promise);  
  
    return promise;  
}  
  
...  
  
private void write(Object msg, boolean flush, ChannelPromise promise) {  
    AbstractChannelHandlerContext next = findContextOutbound();  
    final Object m = pipeline.touch(msg, next);  
    EventExecutor executor = next.executor();  
    if (executor.inEventLoop()) {  
        if (flush) {  
            next.invokeWriteAndFlush(m, promise);  
        } else {  
            next.invokeWrite(m, promise);  
        }  
    } else {  
        AbstractWriteTask task;  
        if (flush) {  
            task = WriteAndFlushTask.newInstance(next, m, promise);  
        } else {  
            task = WriteTask.newInstance(next, m, promise);  
        }  
        safeExecute(executor, task, promise, m);  
    }  
}  
  
...  
  
private void invokeWriteAndFlush(Object msg, ChannelPromise promise) {  
    if (invokeHandler()) {  
        invokeWrite0(msg, promise);  
        invokeFlush0();  
    } else {  
        writeAndFlush(msg, promise);  
    }  
}
```

```
...
private void invokewrite(Object msg, ChannelPromise promise) {
    if (invokeHandler()) {
        invokewrite0(msg, promise);
    } else {
        write(msg, promise);
    }
}
```

### 9.3 抽象编码器处理逻辑 MessageToByteEncoder

将一个对象变成字节并写到socket底层。

## 编码器处理逻辑：MessageToByteEncoder



io.netty.handler.codec.MessageToByteEncoder#write

```
public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws
Exception {
    ByteBuf buf = null;
    try {
        //判断当前encoder处理器是否能处理这个对象
        if (acceptOutboundMessage(msg)) {
            @SuppressWarnings("unchecked")
            I cast = (I) msg;
            buf = allocateBuffer(ctx, cast, preferDirect);
            try {
                //最终会将转化后的对象填充到分配到的buf中
                encode(ctx, cast, buf);
            } finally {
                //释放原始对象
                ReferenceCountUtil.release(cast);
            }
        }
        if (buf.isReadable()) {
```

```

        //传播写好的数据直到Head节点
        ctx.write(buf, promise);
    } else {
        buf.release();
        ctx.write(Unpooled.EMPTY_BUFFER, promise);
    }
    buf = null;
} else {
    ctx.write(msg, promise);
}
} catch (EncoderException e) {
    throw e;
} catch (Throwable e) {
    throw new EncoderException(e);
} finally {
    if (buf != null) {
        buf.release();
    }
}
}

...

```

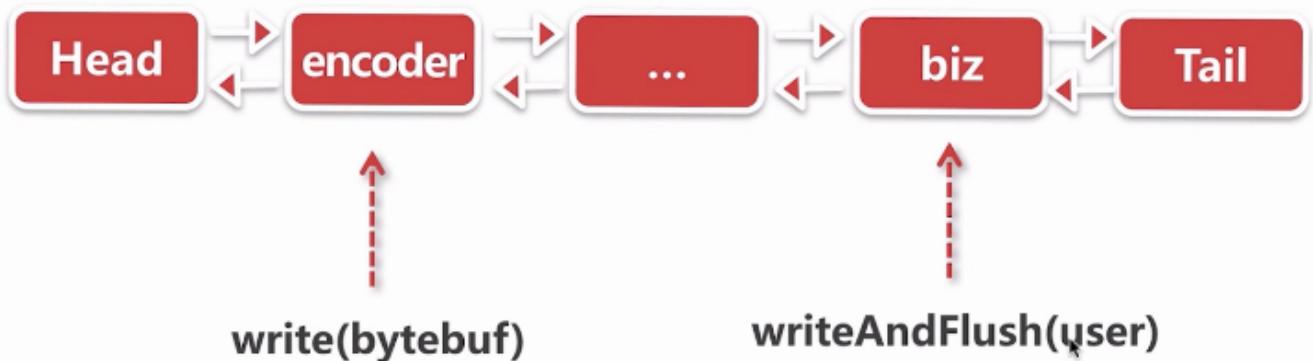
`protected ByteBuf allocateBuffer(ChannelHandlerContext ctx, @SuppressWarnings("unused") I msg, boolean preferDirect) throws Exception {`

```

    if (preferDirect) {
        return ctx.alloc().ioBuffer(); //分配堆外内存
    } else {
        return ctx.alloc().heapBuffer(); //分配堆内内存
    }
}
```

#### 9.4 写buffer队列

## writeAndFlush()



encoder将传进来的user对象转换成bytebuf传递到head节点，最终调用head节点的write方法

io.netty.channel.DefaultChannelPipeline.HeadContext#write

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

io.netty.channel.AbstractChannel.AbstractUnsafe#write

```
public final void write(Object msg, ChannelPromise promise) {
    assertEventLoop();

    ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;
    if (outboundBuffer == null) {
        // If the outboundBuffer is null we know the channel was closed and so
        // need to fail the future right away. If it is not null the handling of the rest
        // will be done in flush0()
        // See https://github.com/netty/netty/issues/2362
        safeSetFailure(promise, WRITE_CLOSED_CHANNEL_EXCEPTION);
        // release message now to prevent resource-leak
        ReferenceCountUtil.release(msg);
        return;
    }

    int size;
    try {
        msg = filterOutboundMessage(msg);
        size = pipeline.estimatorHandle().size(msg);
        if (size < 0) {
            size = 0;
        }
    } catch (Throwable t) {
        safeSetFailure(promise, t);
        ReferenceCountUtil.release(msg);
        return;
    }

    outboundBuffer.addMessage(msg, size, promise);
}
```

## write-写buffer队列

- direct化ByteBuf
  - 将非堆外内存转化为堆外内存 (零拷贝)

io.netty.channel.nio.AbstractNioByteChannel#filterOutboundMessage

```
protected final Object filterOutboundMessage(Object msg) {
    if (msg instanceof ByteBuf) {
        ByteBuf buf = (ByteBuf) msg;
        if (buf.isDirect()) {
            return msg;
        }
    }
}
```

```

    }
    return newDirectBuffer(buf);
}

if (msg instanceof FileRegion) {
    return msg;
}
throw new UnsupportedOperationException(
    "unsupported message type: " + StringUtil.simpleClassName(msg) +
EXPECTED_TYPES);
}

```

- 插入写队列 (写缓冲区 outBoundBuffer)

io.netty.channel.ChannelOutboundBuffer#addMessage

```

//三个指针
// Entry(flushedEntry) --> ... Entry(unflushedEntry) --> ... Entry(tailEntry)
//
// The Entry that is the first in the linked-list structure that was flushed
private Entry flushedEntry;
// The Entry which is the first unflushed in the linked-list structure
private Entry unflushedEntry;
// The Entry which represents the tail of the buffer
private Entry tailEntry;

.....
public void addMessage(Object msg, int size, ChannelPromise promise) {
    Entry entry = Entry.newInstance(msg, size, total(msg), promise);
    if (tailEntry == null) {
        flushedEntry = null;
        tailEntry = entry;
    } else {
        Entry tail = tailEntry;
        tail.next = entry;
        tailEntry = entry;
    }
    if (unflushedEntry == null) {
        unflushedEntry = entry;
    }

    // increment pending bytes after adding message to the unflushed arrays.
    // See https://github.com/netty/netty/issues/1619
    incrementPendingOutboundBytes(size, false);
}

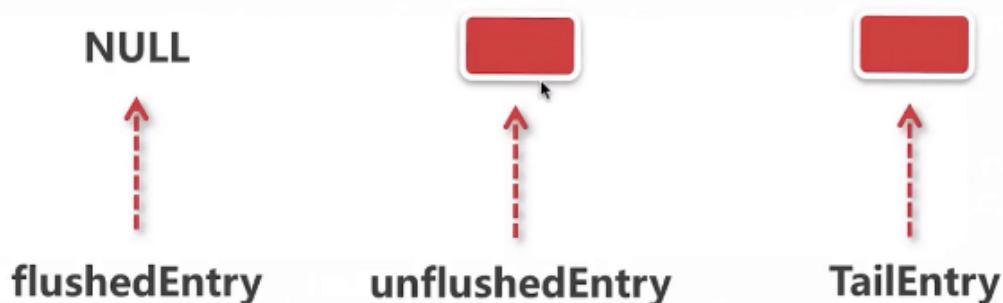
```

## 第一次调用write



## 第二次调用write

14



## 第N次调用write

14



- 设置写状态

- 若写缓冲区中的数据 > 默认的高水位 64k 就设置当前的channel为不可写状态

```
io.netty.channel.ChannelOutboundBuffer#incrementPendingOutboundBytes(long, boolean)
```

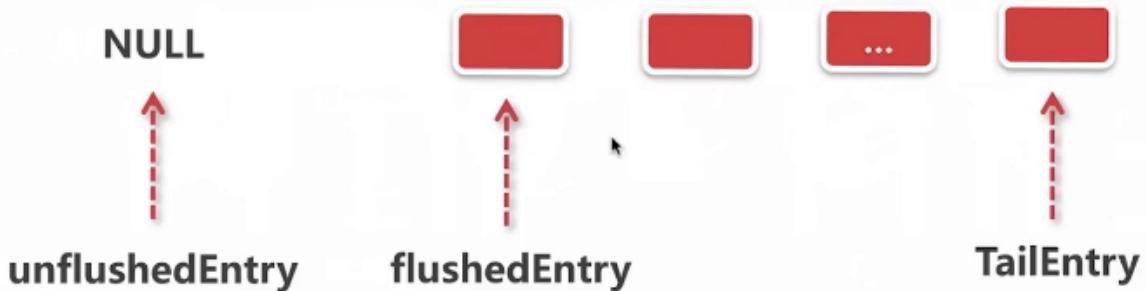
```
private void incrementPendingOutboundBytes(long size, boolean invokeLater) {  
    if (size == 0) {  
        return;  
    }  
  
    long newwriteBufferSize = TOTAL_PENDING_SIZE_UPDATER.addAndGet(this, size);  
    if (newwriteBufferSize > channel.config().getWriteBufferHighwaterMark()) {  
        setUnwritable(invokeLater);  
    }  
}  
  
...  
  
// 自旋 + CAS  
private void setUnwritable(boolean invokeLater) {  
    for (;;) {  
        final int oldValue = unwritable;  
        final int newValue = oldValue | 1;  
        if (UNWRITABLE_UPDATER.compareAndSet(this, oldValue, newValue)) {  
            if (oldValue == 0 && newValue != 0) {  
                fireChannelWritabilityChanged(invokeLater);  
            }  
            break;  
        }  
    }  
}
```

## 9.5 刷新buffer队列

**flush - 刷新buffer队列** 将缓存区中的数据写入socket通道

- 添加刷新标志并设置写状态
  - 移动三个指针
  - 若写缓冲区中的数据 < 默认的低水位 32k 就设置当前的channel为可写状态

**flush**



- 遍历buffer队列，过滤ByteBuf

- 调用jdk底层的API进行自选写

io.netty.channel.DefaultChannelPipelineHandlerContext#flush

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

io.netty.channel.AbstractChannel.AbstractUnsafe#flush

```
public final void flush() {
    assertEventLoop();

    ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;
    if (outboundBuffer == null) {
        return;
    }
    outboundBuffer.addFlush();
    flush0();
}
```

io.netty.channel.ChannelOutboundBuffer#decrementPendingOutboundBytes(long, boolean, boolean)

```
private void decrementPendingOutboundBytes(long size, boolean invokeLater, boolean
notifywritability) {
    if (size == 0) {
        return;
    }
    long newwriteBufferSize = TOTAL_PENDING_SIZE_UPDATER.addAndGet(this, -size);
    if (notifywritability && newwriteBufferSize <
channel.config().getWriteBufferLowWaterMark()) {
        setWritable(invokeLater);
    }
}
```

io.netty.channel.DefaultChannelConfig#writeBufferWaterMark

```
private volatile WriteBufferWaterMark writeBufferWaterMark = WriteBufferWaterMark.DEFAULT;
```

io.netty.channel.WriteBufferWaterMark

```
public final class WriteBufferwaterMark {

    private static final int DEFAULT_LOW_WATER_MARK = 32 * 1024;
    private static final int DEFAULT_HIGH_WATER_MARK = 64 * 1024;

    public static final WriteBufferwaterMark DEFAULT =
        new WriteBufferwaterMark(DEFAULT_LOW_WATER_MARK, DEFAULT_HIGH_WATER_MARK,
false);

    private final int low;
    private final int high;
    ...
}
```

## io.netty.channel.nio.AbstractNioByteChannel#doWrite

```
protected void dowrite(ChannelOutboundBuffer in) throws Exception {
    int writeSpinCount = -1;

    boolean setOpwrite = false;
    for (;;) {
        Object msg = in.current();
        if (msg == null) {
            // wrote all messages.
            clearOpwrite();
            // Directly return here so incompleteWrite(...) is not called.
            return;
        }

        if (msg instanceof ByteBuf) {
            ByteBuf buf = (ByteBuf) msg;
            int readableBytes = buf.readableBytes();
            if (readableBytes == 0) {
                in.remove();
                continue;
            }

            boolean done = false;
            long flushedAmount = 0;
            if (writeSpinCount == -1) {
                // 并发编程中使用自旋可以提高内存使用率和吞吐量
                writeSpinCount = config().getWriteSpinCount();
            }
            for (int i = writeSpinCount - 1; i >= 0; i --) {
                // 将buf写到socket
                int localFlushedAmount = dowriteBytes(buf);
                if (localFlushedAmount == 0) {
                    setOpwrite = true;
                    break;
                }
            }
        }
    }
}
```

```

        flushedAmount += localFlushedAmount;
        if (!buf.isReadable()) {
            done = true;
            break;
        }
    }

    in.progress(flushedAmount);

    if (done) {
        in.remove();
    } else {
        // Break the loop and so incompleteWrite(...) is called.
        break;
    }
} else if (msg instanceof FileRegion) { //认为是从磁盘读到socket
    FileRegion region = (FileRegion) msg;
    boolean done = region.transferred() >= region.count();
    ...
} else {
    // Should not reach here.
    throw new Error();
}
}

incompleteWrite(setOpwrite);
}

```

#### io.netty.channel.ChannelConfig#getWriteSpinCount

```

/**
 * 自旋提高写的吞吐量， 默认自旋16次
 * Returns the maximum loop count for a write operation until
 * {@link WritableByteChannel#write(ByteBuffer)} returns a non-zero value.
 * It is similar to what a spin lock is used for in concurrency programming.
 * It improves memory utilization and write throughput depending on
 * the platform that JVM runs on. The default value is {@code 16}.
 */
int getWriteSpinCount();

```

#### io.netty.channel.socket.nio.NioSocketChannel#doWriteBytes

```

protected int doWriteBytes(ByteBuf buf) throws Exception {
    final int expectedWrittenBytes = buf.readableBytes();
    return buf.readBytes(javaChannel(), expectedWrittenBytes);
}

```

#### io.netty.buffer.PooledDirectByteBuf#readBytes(java.nio.channels.GatheringByteChannel, int)

```
public int readBytes(GatheringByteChannel out, int length) throws IOException {
    checkReadableBytes(length);
    int readBytes = getBytes(readerIndex, out, length, true);
    readerIndex += readBytes;
    return readBytes;
}
```

io.netty.buffer.PooledDirectByteBuf#getBytes(int, java.nio.channels.GatheringByteChannel, int, boolean)

```
// GatheringByteChannel 是 JDK底层的
private int getBytes(int index, GatheringByteChannel out, int length, boolean internal)
throws IOException {
    checkIndex(index, length);
    if (length == 0) {
        return 0;
    }

    //JDK相关的ByteBuffer 将netty自定义的byteBuf塞到JDK的ByteBuffer
    ByteBuffer tmpBuf;
    if (internal) {
        tmpBuf = internalNioBuffer();
    } else {
        tmpBuf = memory.duplicate();
    }
    index = idx(index);
    tmpBuf.clear().position(index).limit(index + length);
    // 调用JDK底层的write方法，将tmpBuf写到socket通道中，并返回写的字节数
    return out.write(tmpBuf);
}
```

## 9.6 netty编码总结

# 十. Netty性能优化工具类解析

## 10.1 性能优化工具类概述

- FastThreadLocal

作用同ThreadLocal，但比ThreadLocal更快。多线程访问同一变量的时候，通过线程本地化的方式(线程隔离，独享变量，并且对变量修改不影响另外的线程)，避免多线程竞争，保证同一个线程里面状态一致性的同时优化程序性能。

- Recycler

实现了轻量级别的对象池机制，对象池的作用：提升性能

- 不需要每次都去new对象，分配内存，减少内存使用率
- 避免反复创建对象，减少JVM GC的压力

## 10.2 FastThreadLocal的使用

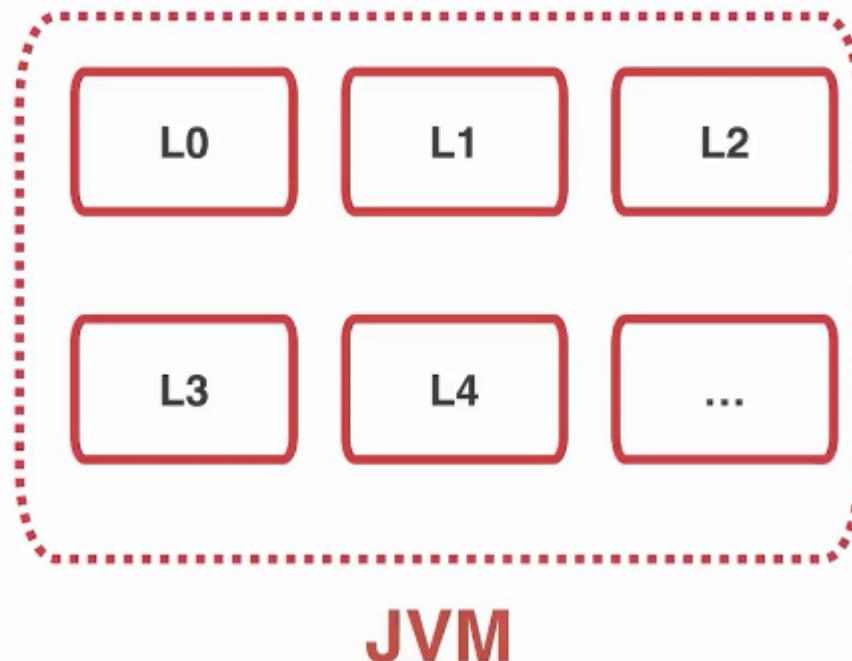
### 10.3 FastThreadLocal的创建 和 get()实现

- FastThreadLocal的创建

在JVM当中，若一个类中通过new去创建一个FastThreadLocal，它就会带来一个索引值，第一个调用new FastThreadLocal()方法，它的索引值就是0，若当前类或其他类中第二个去new FastThreadLocal()，它的索引值就是1，每一个FastThreadLocal在JVM当中都有唯一一个index。

L0, L1, L2.....索引

## FastThreadLocal的实现机制



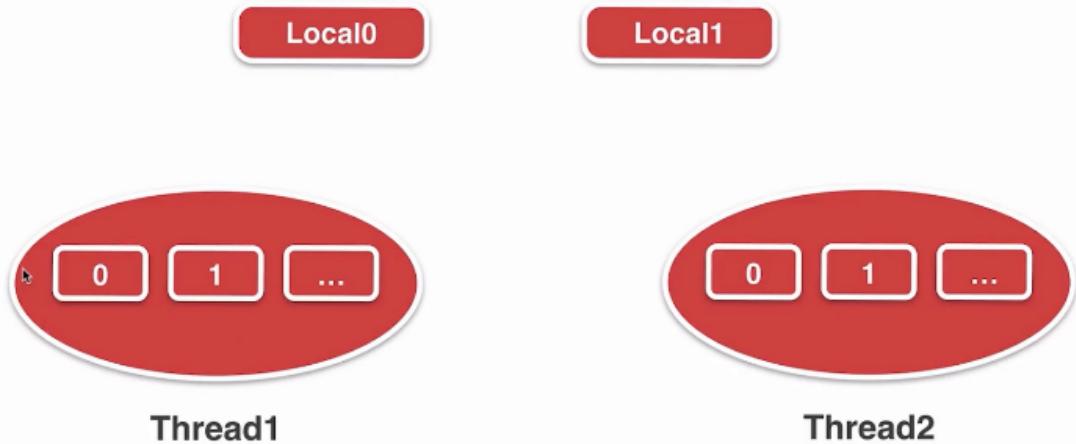
- FastThreadLocal get()实现
  - 获取ThreadLocalMap

## 获取ThreadLocalMap的方式



- 直接通过索引取出对象

# FastThreadLocal的实现机制



- 初始化

## 10.4 FastThreadLocal的set实现

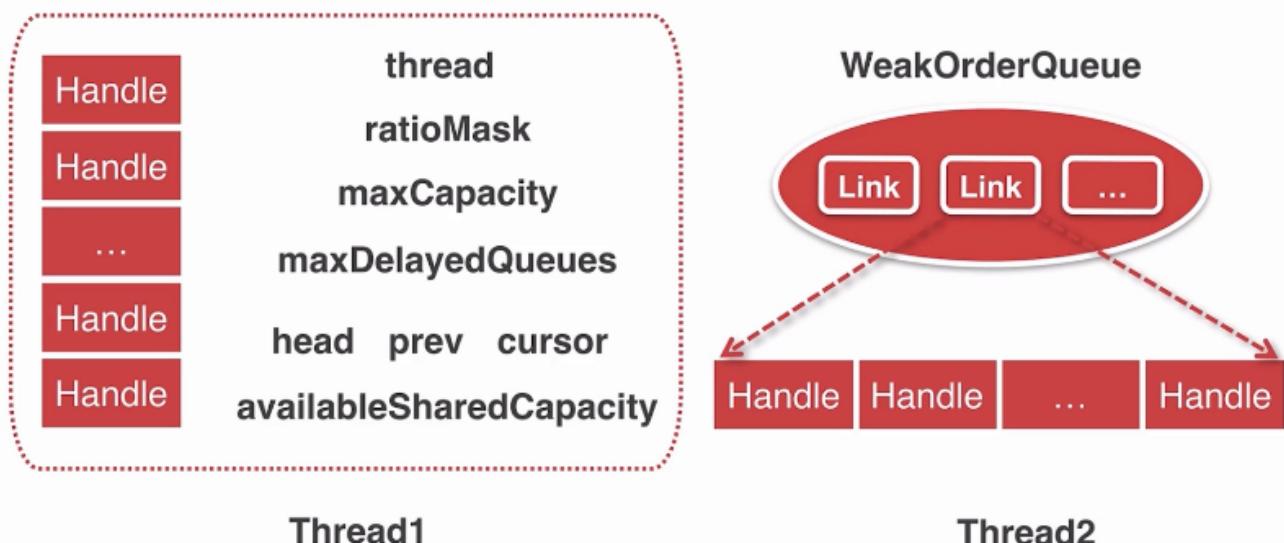
- 获取ThreadLocalMap
- 直接通过索引set对象
- remove对象

## 10.5 轻量级对象池-Recycler的使用

### 10.6 Recycler的创建

虚线代表stack

# Recycler的创建



## 10.7 从Recycler中获取对象

- 获取当前线程的stack

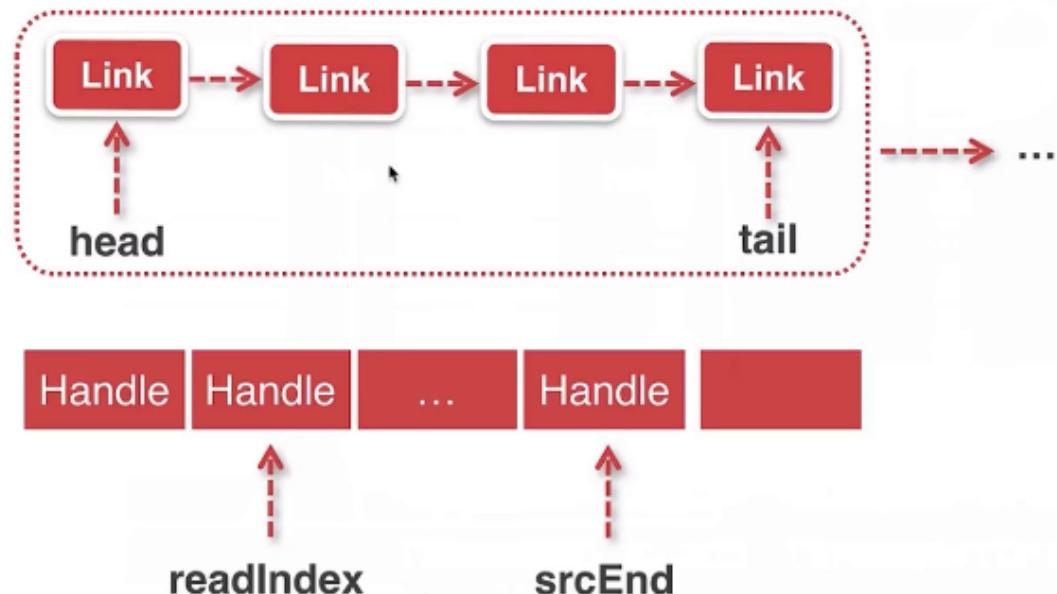
- 从Stack里面弹出对象
- 创建对象并绑定到Stack

## 10.8 同线程回收对象到Recycler

## 10.9 异线程回收对象到Recycler

- 获取WeakOrderQueue
- 创建WeakOrderQueue
- 将对象追加到WeakOrderQueue

# WeakOrderQueue结构

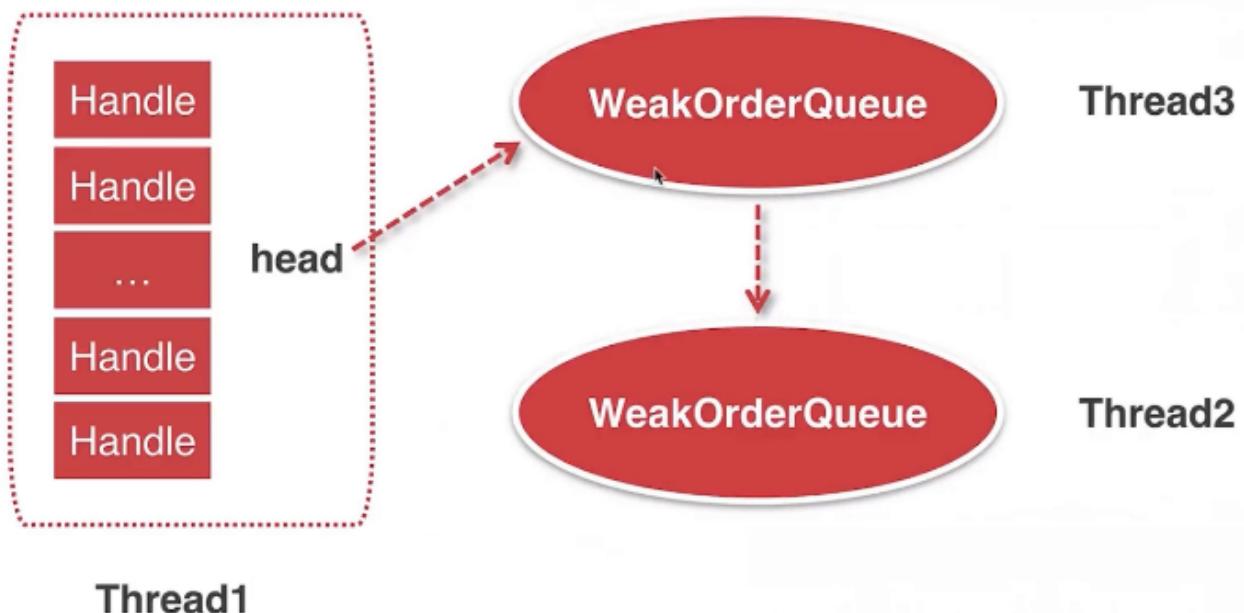


io.netty.util.Recycler.WeakOrderQueue.Link

```
private static final class Link extends AtomicInteger {
    private final DefaultHandle<?>[] elements = new DefaultHandle[LINK_CAPACITY];

    private int readIndex;
    private Link next;
}
```

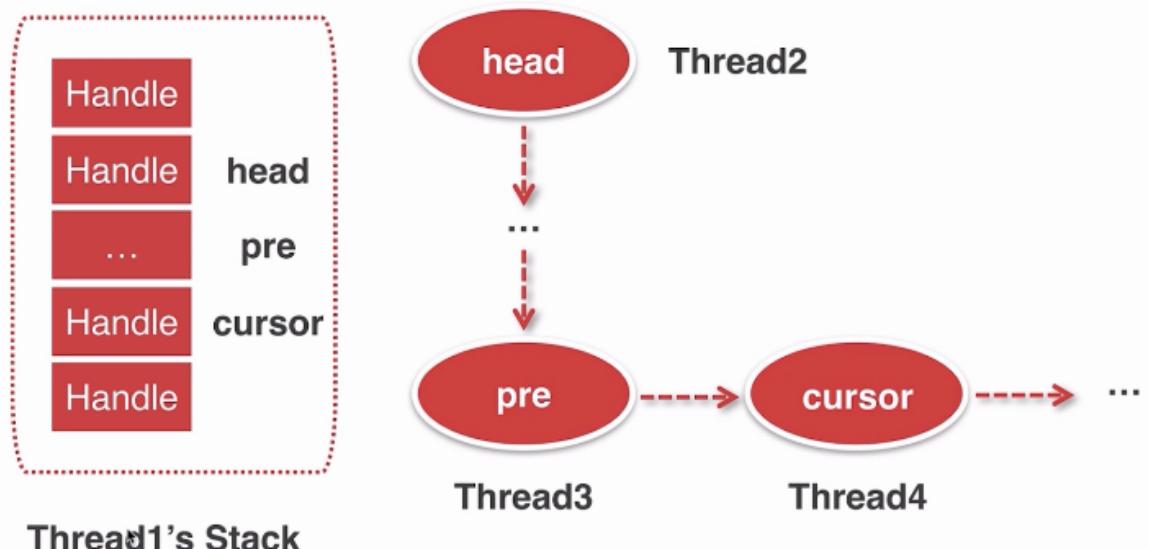
# WeakOrderQueue与stack绑定



## 10.10 异线程收割对象

- 获取当前线程的Stack
- 从Stack里面弹出对象
- 创建对象并绑定到Stack

# Stack与WeakOrderQueue



io.netty.util.Recycler.Stack#scavenge

```

boolean scavenge() {
    // continue an existing scavenge, if any
    if (scavengeSome()) {
        return true;
    }

    // reset our scavenge cursor
    prev = null;
    cursor = head;
    return false;
}

```

## 10.11 性能优化工具类总结

### 十一. Netty设计模式的应用

- 单例模式
  - 一个类全局只有一个对象
  - 延迟创建（懒汉模式）
  - 避免线程安全问题
  - netty中应用 -- 简单优雅
    - io.netty.handler.timeout.ReadTimeoutException

```

public final class ReadTimeoutException extends TimeoutException {

    private static final long serialVersionUID = 169287984113283421L;

    //加载类的时候初始化INSTANCE，JVM在加载ReadTimeoutException类的时候默认会加上同步
    //块，以此来保证线程安全。--饿汉模式
    public static final ReadTimeoutException INSTANCE = new
    ReadTimeoutException();

    //构造私有化，实现全局只有一个对象
    private ReadTimeoutException() { }
}

```

- io.netty.handler.codec.mqtt.MqttEncoder

```

@ChannelHandler.Sharable
public final class MqttEncoder extends
MessageToMessageEncoder<MqttMessage> {

    public static final MqttEncoder INSTANCE = new MqttEncoder();

    private MqttEncoder() { }

    .....
}

```

- 策略模式

- 封装一些列可相互替换的算法家族
- 动态选择某一种策略
- 普通示例

```
/**  
 * @see DefaultEventExecutorChooserFactory#newChooser(EventExecutor[])  
 */  
public class Strategy {  
    private Cache cacheMemory = new CacheMemoryImpl();  
    private Cache cacheRedis = new CacheRedisImpl();  
  
    public interface Cache {  
        boolean add(String key, Object object);  
    }  
  
    public class CacheMemoryImpl implements Cache {  
        @Override  
        public boolean add(String key, Object object) {  
            // 保存到map  
            return false;  
        }  
    }  
  
    public class CacheRedisImpl implements Cache {  
        @Override  
        public boolean add(String key, Object object) {  
            // 保存到redis  
            return false;  
        }  
    }  
  
    public Cache getCache(String key) {  
        if (key.length() < 10) {  
            return cacheRedis;  
        }  
        return cacheMemory;  
    }  
}
```

- netty中应用

- io.netty.util.concurrent.DefaultEventExecutorChooserFactory#newChooser

```
public EventExecutorChooser newChooser(EventExecutor[] executors) {
    if (isPowerOfTwo(executors.length)) {
        return new PowerOfTowEventExecutorChooser(executors);
    } else {
        return new GenericEventExecutorChooser(executors);
    }
}
```

- 装饰者模式

- 装饰者和被装饰者继承同一接口
- 装饰者给被装饰者动态修改行为
- 普通示例

```
/*
 * @see io.netty.buffer.WrappedByteBuf;
 * @see io.netty.buffer.UnreleasableByteBuf
 * @see io.netty.buffer.SimpleLeakAwareByteBuf
 */
public class Decorate {

    // 优惠方案
    public interface OnSalePlan {
        float getPrice(float oldPrice);
    }

    // 无优惠
    public static class NonePlan implements OnSalePlan {
        static final OnSalePlan INSTANCE = new NonePlan();

        private NonePlan() {
        }

        public float getPrice(float oldPrice) {
            return oldPrice;
        }
    }

    // 立减优惠
    public static class KnockPlan implements OnSalePlan {
        // 立减金额
        private float amount;

        public KnockPlan(float amount) {
            this.amount = amount;
        }

        public float getPrice(float oldPrice) {
            return oldPrice - amount;
        }
    }
}
```

```

        }

    }

    // 打折优惠
    public static class DiscountPlan implements OnSalePlan {
        // 折扣
        public int discount;
        private OnSalePlan previousPlan;

        public DiscountPlan(int discount, OnSalePlan previousPlan) {
            this.discount = discount;
            this.previousPlan = previousPlan;
        }

        public DiscountPlan(int discount) {
            this(discount, NonePlan.INSTANCE);
        }

        public float getPrice(float oldPrice) {
            return previousPlan.getPrice(oldPrice) * discount / 10;
        }
    }

    public static void main(String[] args) {
        DiscountPlan simpleDiscountPlan = new DiscountPlan(5);
        System.out.println(simpleDiscountPlan.getPrice(100));

        KnockPlan previousPlan = new KnockPlan(50);
        DiscountPlan complexDiscountPlan = new DiscountPlan(5, previousPlan);
        System.out.println(complexDiscountPlan.getPrice(100));
    }
}

```

- netty中应用

- io.netty.buffer.WrappedByteBuf

```

class WrappedByteBuf extends ByteBuf {

    protected final ByteBuf buf;

    protected WrappedByteBuf(ByteBuf buf) {
        if (buf == null) {
            throw new NullPointerException("buf");
        }
        this.buf = buf;
    }

    @Override
    public final boolean hasMemoryAddress() {
        return buf.hasMemoryAddress();
    }
}

```

```
}

@Override
public final long memoryAddress() {
    return buf.memoryAddress();
}

@Override
public final int capacity() {
    return buf.capacity();
}

...

}
```

- io.netty.buffer.UnreleasableByteBuf
- io.netty.buffer.SimpleLeakAwareByteBuf

```
final class SimpleLeakAwareByteBuf extends WrappedByteBuf {

    private final ResourceLeak leak;

    SimpleLeakAwareByteBuf(ByteBuf buf, ResourceLeak leak) {
        super(buf);
        this.leak = leak;
    }

    @Override
    public ByteBuf touch() {
        return this;
    }

    @Override
    public ByteBuf touch(Object hint) {
        return this;
    }

    @Override
    public boolean release() {
        boolean deallocated = super.release();
        if (deallocated) {
            leak.close();
        }
        return deallocated;
    }

    ...

}
```

- 观察者模式

- 观察者和被观察者
- 观察者订阅消息，被观察者发布消息
- 订阅则能收到，取消订阅则收不到
- 普通示例

```

public class observerTest {
    /**
     * 被观察者
     */
    public interface Observable {
        void registerObserver(Observer o);

        void removeObserver(Observer o);

        void notifyObserver();
    }

    /**
     * 观察者
     */
    public interface Observer {
        void notify(String message);
    }

    public static class Girl implements Observable {
        private String message;

        List<Observer> observerList;

        public Girl() {
            observerList = new ArrayList<>();
        }

        @Override
        public void registerObserver(Observer o) {
            observerList.add(o);
        }

        @Override
        public void removeObserver(Observer o) {
            observerList.remove(o);
        }

        @Override
        public void notifyObserver() {
            for (Observer observer : observerList) {
                observer.notify(message);
            }
        }

        public void hasBoyFriend() {
    }
}

```

```
        message = "女神有男朋友了";
        notifyObserver();
    }

    public void getMarried() {
        message = "女神结婚了，你们都死心吧!";
        notifyObserver();
    }

    public void getsingled() {
        message = "女神单身了，你们有机会了!";
        notifyObserver();
    }
}

/**
 * 男孩
 */
public static class Boy implements Observer {
    public void notify(String message) {
        System.out.println("男孩收到消息: " + message);
    }
}

/**
 * 男人
 */
public static class Man implements Observer {
    public void notify(String message) {
        System.out.println("男人收到消息: " + message);
    }
}

/**
 * 老男人
 */
public static class OldMan implements Observer {
    public void notify(String message) {
        System.out.println("老男人收到消息: " + message);
    }
}

public static void main(String[] args) {
    Girl girl = new Girl();
    Boy boy = new Boy();
    Man man = new Man();
    OldMan oldMan = new OldMan();

    // 通知男孩、男人、老男人，女神有男朋友了
    girl.registerObserver(boy);
    girl.registerObserver(man);
    girl.registerObserver(oldMan);
    girl.hasBoyFriend();
```

```
System.out.println("=====");

// 通知男孩，男人，女神结婚了
girl.removeObserver(oldMan);
girl.getMarried();
System.out.println("=====");

girl.registerObserver(oldMan);
girl.getSingled();
}

//netty中的观察者模式/channel-future模式
public void write(NioSocketChannel channel, Object object) {
    //writeAndFlush 方法 本质是创建被观察者 channelFuture
    ChannelFuture channelFuture = channel.writeAndFlush(object);
    //addListener 本质是向被观察者中添加一系列观察者
    channelFuture.addListener(future -> {
        if (future.isSuccess()) {
            ...
        } else {
            ...
        };
    });
    channelFuture.addListener(future -> {
        if (future.isSuccess()) {
            ...
        } else {
            ...
        };
    });
    channelFuture.addListener(future -> {
        if (future.isSuccess()) {
            ...
        } else {
            ...
        };
    });
}
```

- netty中应用
    - io.netty.channel.AbstractChannelHandlerContext#writeAndFlush(java.lang.Object, io.netty.channel.ChannelPromise)
  - 迭代器模式
    - 迭代器接口 io.netty.buffer.ByteBuf#forEachByte(io.netty.util.ByteProcessor)
      - io.netty.buffer.CompositeByteBuf 实现零拷贝
    - 对容器里面的各个对象进行访问

- 责任链模式

- 责任处理接口 `io.netty.channel.ChannelHandler`
- 创建链，添加删除责任处理器接口 `io.netty.channel.ChannelPipeline`
- 上下文 `io.netty.channel.ChannelHandlerContext`
- 责任终止机制

## 责任链模式



## 十二. Netty高并发性能调优

### 12.1 单机百万连接调优

- 如何模拟百万连接
  - 两台虚拟机：一个客户端一个服务端如何模拟单机百万连接
  - 客户端代码 打成jar包运行在客户端虚拟机

```
public class Client {

    private static final String SERVER_HOST = "192.168.1.42";

    public static void main(String[] args) {
        new Client().start(BEGIN_PORT, N_PORT);
    }

    public void start(final int beginPort, int nPort) {
        System.out.println("client starting....");
        EventLoopGroup eventLoopGroup = new NioEventLoopGroup();
        final Bootstrap bootstrap = new Bootstrap();
        bootstrap.group(eventLoopGroup);
        bootstrap.channel(NioSocketChannel.class);
        bootstrap.option(ChannelOption.SO_REUSEADDR, true);
        bootstrap.handler(new ChannelInitializer<SocketChannel>() {
            @Override
            protected void initChannel(SocketChannel ch) {
            }
        });

        int index = 0;
        int port;
        while (!Thread.interrupted()) {
            port = beginPort + index;
            try {

```

```
        ChannelFuture channelFuture = bootstrap.connect(SERVER_HOST, port);
        channelFuture.addListener((ChannelFutureListener) future -> {
            if (!future.isSuccess()) {
                System.out.println("connect failed, exit!");
                System.exit(0);
            }
        });
        channelFuture.get();
    } catch (Exception e) {
    }

    if (++index == nPort) {
        index = 0;
    }
}
}
```

- 服务端代码 打成jar包运行在服务端虚拟机

```
public final class Server {  
  
    public static void main(String[] args) {  
        new Server().start(BEGIN_PORT, N_PORT);  
    }  
  
    public void start(int beginPort, int nPort) {  
        System.out.println("server starting....");  
  
        EventLoopGroup bossGroup = new NioEventLoopGroup();  
        EventLoopGroup workerGroup = new NioEventLoopGroup();  
  
        ServerBootstrap bootstrap = new ServerBootstrap();  
        bootstrap.group(bossGroup, workerGroup);  
        bootstrap.channel(NioServerSocketChannel.class);  
        bootstrap.childOption(ChannelOption.SO_REUSEADDR, true);  
  
        bootstrap.childHandler(new ConnectionCountHandler());  
  
        for (int i = 0; i < nPort; i++) {  
            int port = beginPort + i;  
            bootstrap.bind(port).addListener((ChannelFutureListener) future -> {  
                System.out.println("bind success in port: " + port);  
            });  
        }  
        System.out.println("server started!");  
    }  
}
```

- #### ○ 连接数统计

```

@Sharable
public class ConnectionCountHandler extends ChannelInboundHandlerAdapter {

    private AtomicInteger nConnection = new AtomicInteger();

    //每隔2s统计连接数
    public ConnectionCountHandler() {
        Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(() -> {
            System.out.println("connections: " + nConnection.get());
        }, 0, 2, TimeUnit.SECONDS);

    }

    @Override
    public void channelActive(ChannelHandlerContext ctx) {
        nConnection.incrementAndGet();
    }

    @Override
    public void channelInactive(ChannelHandlerContext ctx) {
        nConnection.decrementAndGet();
    }

}

```

启动服务端，通常绑定一个端口如8000，然后客户端去连接，默认情况下端口号的大小是有限制的，最大65535，而1024及1024以下的端口只能被root用户使用，在扣除一些常用的端口，实际情况是，客户端可以去连接服务端的端口只有6w左右。最多只能实现单机6w连接。

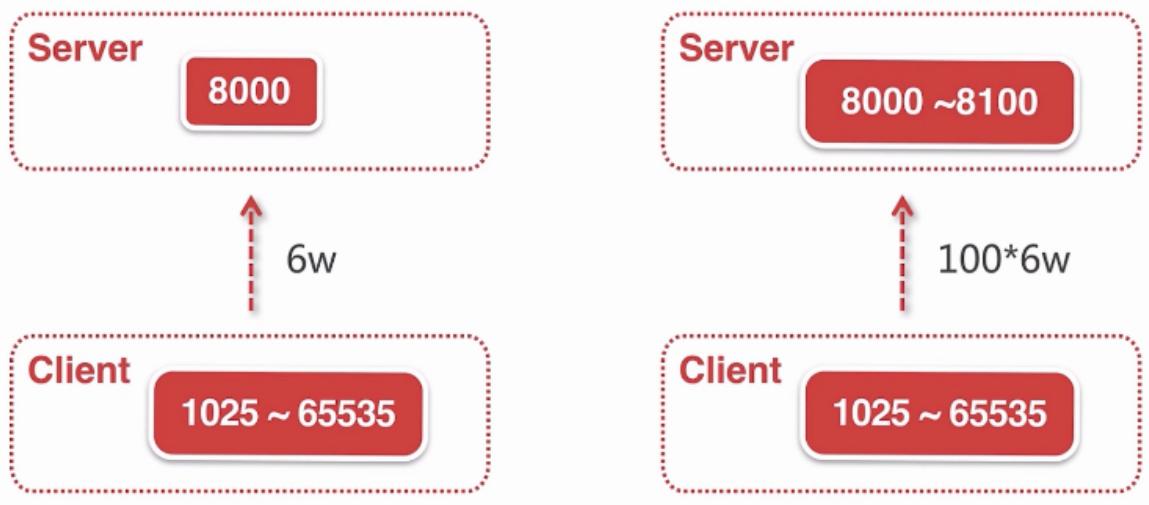
要达到百万，简单的办法：使用十几个客户端，每个客户端去连接相同的服务端端口，但同时也是比较麻烦的，因为要同时启动十几个客户端。

另一个办法：服务端开启100个端口。对客户端而言，同一个端口号可以连接Server不同的端口，6w个端口连接服务端100个端口，累加起来就是600w。

TCP基础知识：**四元组概念**：源IP地址、目的IP地址、源端口、目的端口。、

**TCP会话(TCP\_Session\_IDT)**可以通过**四元组<源IP地址、目的IP地址、源端口号和目的端口号>唯一标识**，源端口相同而目的端口不同，TCP底层认为是不同的TCP连接。

# 如何模拟单机百万连接



- 突破局部文件句柄限制
  - linux系统中，默认情况下，单个进程可以打开的文件句柄数是有限的
    - ulimit -n ==> 1024
  - 一个TCP连接对应一个文件句柄 对于应用程序而言，一个服务端默认建立的连接数是有限制的。

/etc/security/limits.conf

\* 表示当前用户

\* hard nofile 1000000

\* soft nofile 1000000

追加后重启虚拟机，是配置生效

- 突破全局文件句柄限制

cat /proc/sys/fs/file-max

echo 1000000 > /proc/sys/fs/file-max

但是修改后，若重启虚拟机，file-max会还原。

```
sudo -i, 加载用户变量，并跳转到目标用户home目录;  
sudo -s, 不加载用户变量，不跳转目录;
```

如何解决：

/etc/sysctl.conf

fs.file-max=1000000

sudo sysctl -p

## 12.2 Netty应用级别性能调优

- Netty应用级别性能瓶颈

代码实现统计qps, 平均响应时间统计：

```

@ChannelHandler.Sharable
public class ClientBusinessHandler extends SimpleChannelInboundHandler<ByteBuf> {
    public static final ChannelHandler INSTANCE = new ClientBusinessHandler();

    private static AtomicLong beginTime = new AtomicLong(0);
    private static AtomicLong totalResponseTime = new AtomicLong(0);
    private static AtomicInteger totalRequest = new AtomicInteger(0);

    public static final Thread THREAD = new Thread(() -> {
        try {
            while (true) {
                long duration = System.currentTimeMillis() - beginTime.get();
                if (duration != 0) {
                    System.out.println("qps: " + 1000 * totalRequest.get() / duration + ", " + "avg response time: " + ((float) totalResponseTime.get()) / totalRequest.get());
                    Thread.sleep(2000);
                }
            }
        } catch (InterruptedException ignored) {
        }
    });
}

@Override
public void channelActive(ChannelHandlerContext ctx) {
    ctx.executor().scheduleAtFixedRate(() -> {

        ByteBuf byteBuf = ctx.alloc().ioBuffer();
        byteBuf.writeLong(System.currentTimeMillis());
        ctx.channel().writeAndFlush(byteBuf);

    }, 0, 1, TimeUnit.SECONDS);
}

@Override
protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) {
    totalResponseTime.addAndGet(System.currentTimeMillis() - msg.readLong());
    totalRequest.incrementAndGet();

    if (beginTime.compareAndSet(0, System.currentTimeMillis())) {
        THREAD.start();
    }
}

@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
    // ignore
}
}

```

- Netty应用级别性能调优过程

如何在生产环境中，使用netty的应用响应时间从秒级到毫秒级的优化？

思路：1) 将耗时的操作放到线程池里执行

```
@ChannelHandler.Sharable
public class ServerBusinessHandler extends SimpleChannelInboundHandler<ByteBuf> {
    public static final ChannelHandler INSTANCE = new ServerBusinessHandler();

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) {
        ByteBuf data = Unpooled.directBuffer();
        data.writeBytes(msg);
        //getResult相对耗时，可能从数据库读取数据，程序瓶颈的地方
        Object result = getResult(data);
        ctx.channel().writeAndFlush(result);
    }

    protected Object getResult(ByteBuf data) {
        // 90.0% == 1ms
        // 95.0% == 10ms 1000 50 > 10ms
        // 99.0% == 100ms 1000 10 > 100ms
        // 99.9% == 1000ms 1000 1 > 1000ms
        int level = ThreadLocalRandom.current().nextInt(1, 1000);

        int time;
        if (level <= 900) {
            time = 1;
        } else if (level <= 950) {
            time = 10;
        } else if (level <= 990) {
            time = 100;
        } else {
            time = 1000;
        }

        try {
            Thread.sleep(time);
        } catch (InterruptedException e) {
        }

        return data;
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        // ignore
    }
}
```

=====》

```

@ChannelHandler.Sharable
public class ServerBusinessThreadPoolHandler extends ServerBusinessHandler {
    public static final ChannelHandler INSTANCE = new
ServerBusinessThreadPoolHandler();
    private static ExecutorService threadPool = Executors.newFixedThreadPool(1000);

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) {
        ByteBuf data = Unpooled.directBuffer();
        data.writeBytes(msg);
        //将耗时的部分交给线程池，由线程池分配线程执行
        threadPool.submit(() -> {
            Object result = getResult(data);
            ctx.channel().writeAndFlush(result);
        });
    }
}

```

## 2) 调整线程池的中线程的数量

com.leh.netty.performance.thread.ServerBusinessThreadPoolHandler#threadPool

```

private static ExecutorService threadPool = Executors.newFixedThreadPool(1000);

public class Server {
    public static void main(String[] args) {

        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        //创建业务线程组
        EventLoopGroup businessGroup = new NioEventLoopGroup(1000);

        ServerBootstrap bootstrap = new ServerBootstrap();
        bootstrap.group(bossGroup, workerGroup);
        bootstrap.channel(NioServerSocketChannel.class);
        bootstrap.childOption(ChannelOption.SO_REUSEADDR, true);

        bootstrap.childHandler(new ChannelInitializer<SocketChannel>() {
            @Override
            protected void initChannel(SocketChannel ch) {
                ch.pipeline().addLast(new FixedLengthFrameDecoder(Long.BYTES));
                //将业务处理器交给业务线程处理
                ch.pipeline().addLast(businessGroup,
ServerBusinessHandler.INSTANCE);
                // ch.pipeline().addLast(ServerBusinessThreadPoolHandler.INSTANCE);
            }
        });
    }
}

```

```
        bootstrap.bind(PORT).addListener((ChannelFutureListener) future ->
System.out.println("bind success in port: " + PORT));
    }

}
```