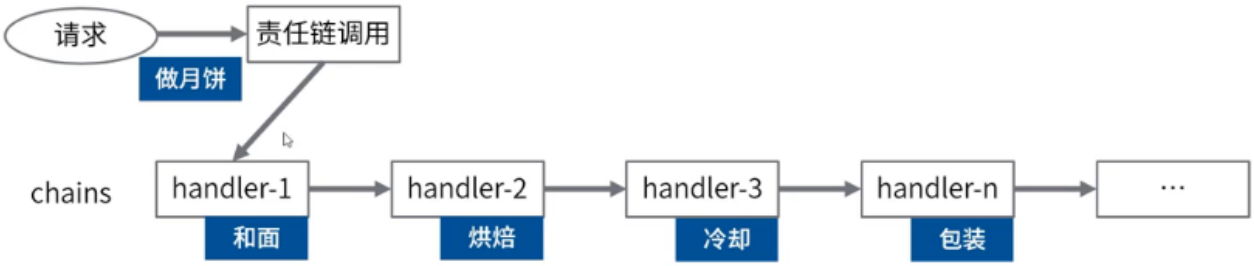
Netty职责链Pipeline详解

设计模式-责任链模式

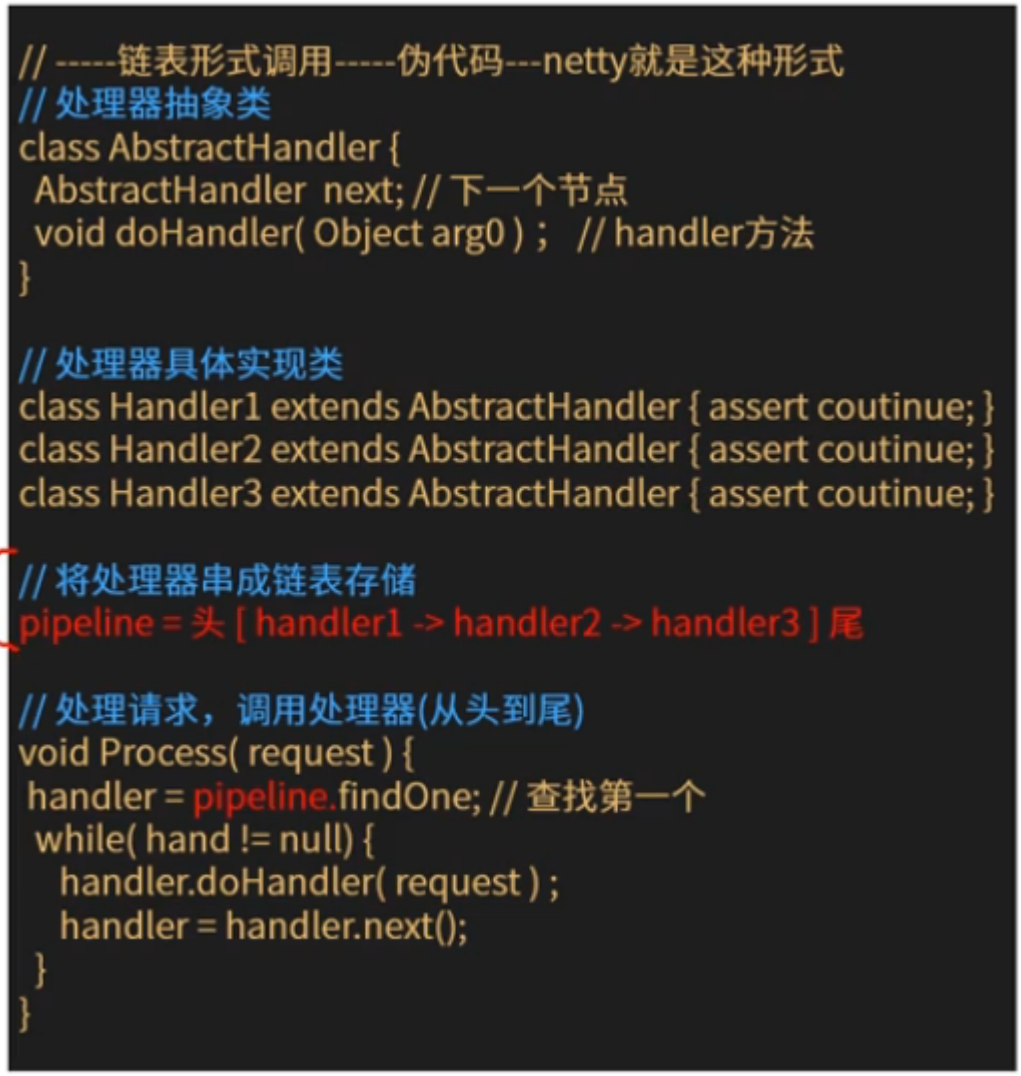
责任链模式为请求创建了一个处理对象的链。发起请求和具体请求的过程进行解耦。职责链上的处理者负责处理请求，客户端只需要将请求发送到职责链上即可，无需关心请求的处理细节和请求的传递。



实现责任链模式

实现责任链模式4个要素：处理器抽象类，具体的处理器实现类，保存处理器信息，处理执行

```
// -----集合形式存储-----伪代码---类似tomcat中filters
// 处理器抽象类
class AbstractHandler {  void doHandler( Object arg0 ) }

// 处理器具体实现类
class Handler1 extends AbstractHandler { assert coutinue; }
class Handler2 extends AbstractHandler { assert coutinue; }
class Handler3 extends AbstractHandler { assert coutinue; }

// 创建集合并存储所有处理器实例信息
List handlers = new List();
handlers.add(handler1, handler2, handler3);

//处理请求，调用处理器()
void Process( request ) {
 for( handler in handlers ) {
   handler.doHandler( request ) ;
 }
}

// 发起请求调用，通过责任链处理请求
call. process(request);
```



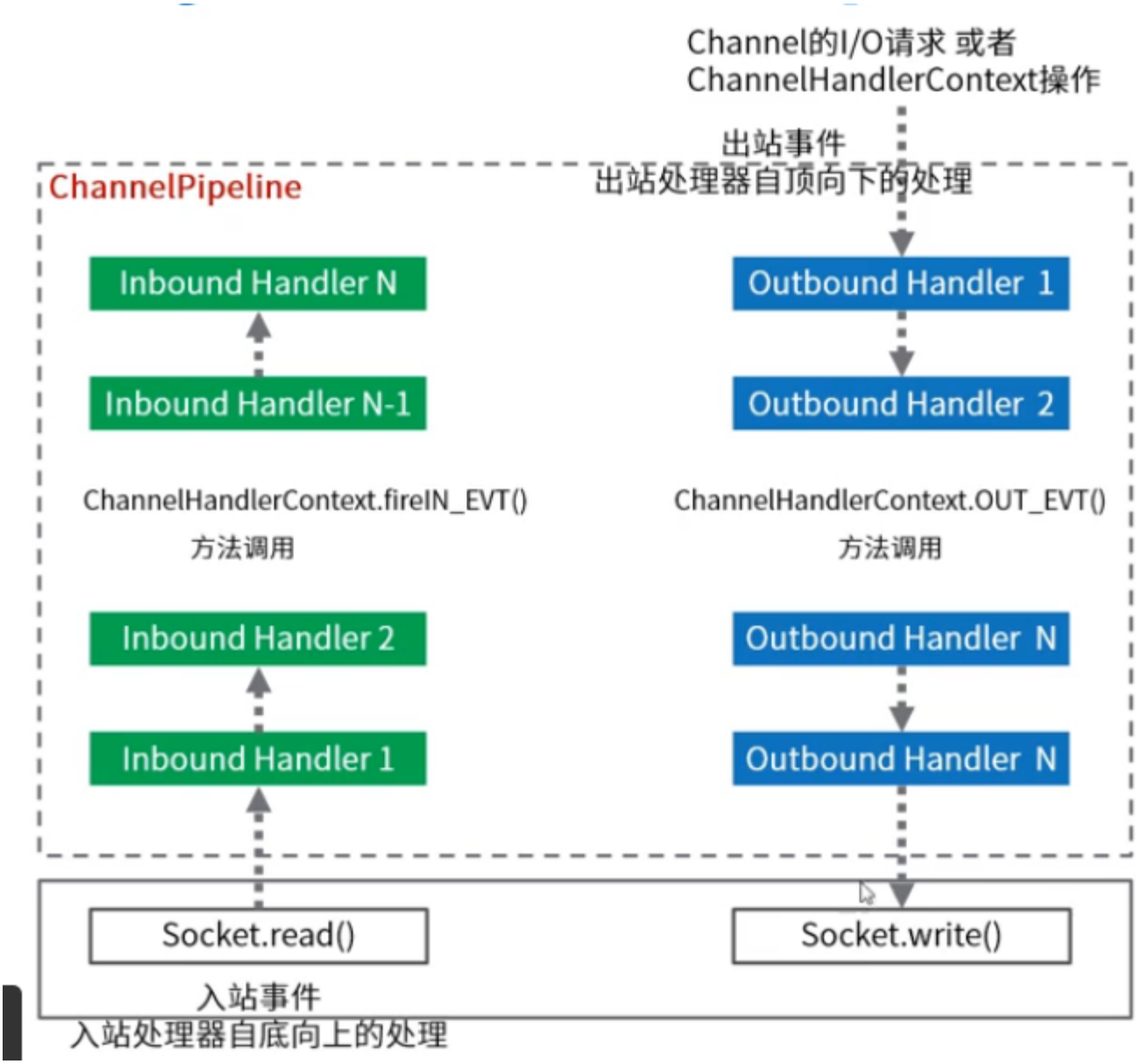

```java
public class PipelineDemo {

    /**
     * 初始化时需要构造一个头作为责任链的开始,没有具体处理过程只是将请求进行传播
     */
    public HandlerChainContext head = new HandlerChainContext(new AbstractHandler() {
        @Override
        void doHandler(HandlerChainContext handlerChainContext, Object arg0) {
            handlerChainContext.runNext(arg0);
        }
    });

    public void requestProcess(Object arg0) {
        this.head.handler(arg0);
    }

    public void addLast(AbstractHandler handler) {
        HandlerChainContext context = head;
        while (context.next != null) {
            context = context.next;
        }
        context.next = new HandlerChainContext(handler);
```

```
    }

    public static void main(String[] args) {
        PipelineDemo pipelineDemo = new PipelineDemo();
        pipelineDemo.addLast(new HandlerOne());
        pipelineDemo.addLast(new HandlerTwo());
        pipelineDemo.addLast(new HandlerOne());
        pipelineDemo.requestProcess("开始做蛋糕");
    }
}

/**
 * 责任链上下文,负责维护链
 */
class HandlerChainContext {
    HandlerChainContext next;
    AbstractHandler handler;

    public HandlerChainContext(AbstractHandler handler) {
        this.handler = handler;
    }

    void handler(Object arg0) {
        this.handler.doHandler(this, arg0);
    }

    void runNext(Object arg0) {
        if (null != this.next) {
            this.next.handler(arg0);
        }
    }
}

/**
 * 抽象处理器
 */
abstract class AbstractHandler {
    abstract void doHandler(HandlerChainContext handlerChainContext, Object arg0);
}

class HandlerOne extends AbstractHandler {
    @Override
    void doHandler(HandlerChainContext handlerChainContext, Object arg0) {
        arg0 = arg0.toString() + " 和面";
        System.out.println("hanlerOne handle: " + arg0);
        handlerChainContext.runNext(arg0);
    }
}

class HandlerTwo extends AbstractHandler {
    @Override
    void doHandler(HandlerChainContext handlerChainContext, Object arg0) {
        arg0 = arg0.toString() + " 加奶油";
```

```
        System.out.println("handlerTwo handle: " + arg0);
        handlerChainContext.runNext(arg0);
    }
}
```

Netty中的ChannelPipeline责任链

Pipeline管道中保存了通道所有的处理器信息。创建新Channel时自动创建一个专有的pipeline。入站事件和出站操作会调用Pipeline上的处理器。



入站事件和出站事件

1. 入站事件：通常指I/O线程生成了入站数据。通俗点说就是从socket底层自己往上往上冒出来的事件都是入站事件。比如EventLoop收到selector的OP_READ事件，入站处理器调用socketChannel.read(ByteBuffer)接收到数据后，这将导致通道的ChannelPipeline中包含下一个钟的channelRead方法被调用。
2. 出站事件：经常指I/O线程执行实际的输出操作。通俗点说就是想主动往socket底层操作的事件都是出站事件。比如bind方法用意是请求server socket绑定到给定的SocketAddress，这将导致通道的ChannelPipeline中包含的下一个出站处理器中的bind方法被调用。

Netty中事件定义

inbound 入站事件

事件	描述
fireChannelRegistered	channel注册事件
fireChannelUnregistered	channel解除注册事件
fireChannelActive	channel活跃事件
fireChannelInactive	channel非活跃事件
fireExceptionCaught	异常事件
fireUserEventTriggered	用户自定义事件
fireChannelRead	channel读事件
fireChannelReadComplete	channel读完成事件
fireChannelWritabilityChanged	channel写状态变化事件

outbound 出站事件

事件	描述
bind	端口绑定事件
connect	连接事件
disconnect	断开连接事件
close	关闭事件
deregister	解除注册事件
flush	刷新数据到网络事件
read	读事件，用于注册OP_READ到selector。
write	写事件
writeAndFlush	写出数据事件

Pipeline中的handler是什么

1. ChannelHandler：用于处理I/O事件或拦截I/O操作，并转发到ChannelPipeline中的下一个处理器。这个顶级接口定义功能很弱，实际使用时会去实现下面两大子接口：处理入站I/O事件的ChannelInboundHandler，处理出站I/O操作的ChannelOutboundHandler。
2. 适配器类：为了开发方便，避免所有handler去实现一遍接口方法，Netty提供了简单的实现类：ChannelInboundHandlerAdapter处理入站I/O事件，ChannelOutboundHandlerAdapter来处理出站I/O事件，ChannelDuplexHandler来支持同时处理入站和出站事件。
3. ChannelHandlerContext：实际存储在Pipeline中的对象并非ChannelHandler，而是上下文对象。将handler包裹在上下文对象中，通过上下文对象与它所属的ChannelPipeline交互，向上或向下传递事件或者修改pipeline都是通过上下文对象进行的。

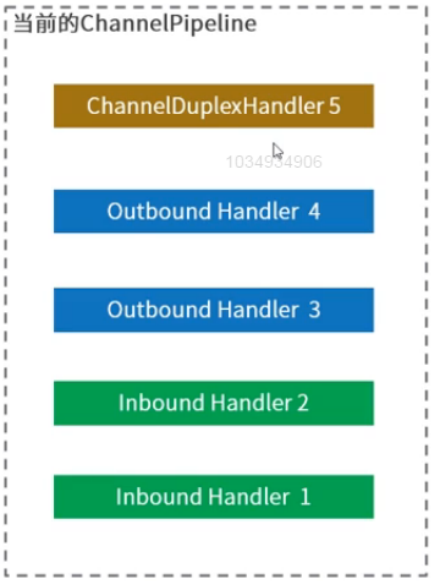
维护Pipeline中的handler

ChannelPipeline是线程安全的，ChannelHandler可以在任何时候添加或删除。例如，你可以在即将交换敏感信息时插入加密处理程序，并在交换完后删除它。

方法名称	描述
addFirst	最前面插入
addLast	最后面插入
addBefore	插入到指定处理器前面
addAfter	插入到指定处理器后面
remove	移除指定处理器
removeFirst	移除第一个处理器
removeLast	移除最后一个处理器
replace	替换指定的处理器

```
// 示例伪代码
ChannelPipeline p = ...;
p.addLast("1", new InboundHandlerA());
p.addLast("2", new InboundHandlerB());
p.addLast("3", new OutboundHandlerA());
p.addLast("4", new OutboundHandlerB());
p.addLast("5", new InboundOutboundHandlerX());
```

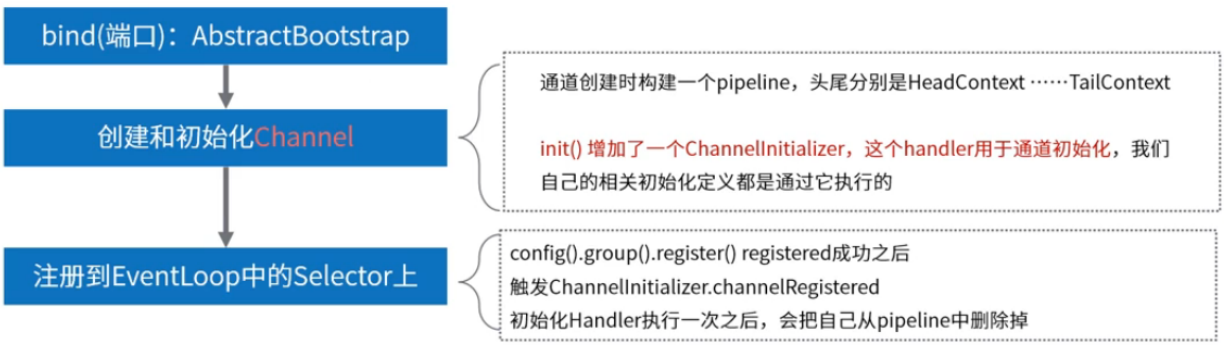
handler的执行分析



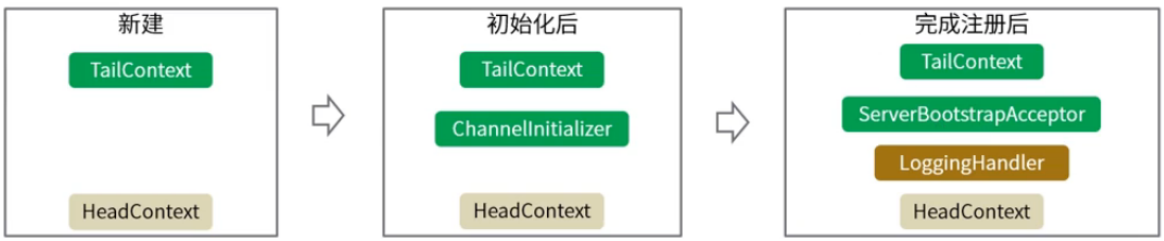
当入站事件时，执行顺序是1、2、3、4、5
当出站事件时，执行顺序是5、4、3、2、1
在这一原则之上，ChannelPipeline在执行时会进行选择
3和4为出站处理器，因此入站事件的实际执行是:1、2、5
1和2为入站处理器，因此出站事件的实际执行是:5、4、3

不同的入站事件会触发handler不同的方法执行：
上下文对象中 fire** 开头的方法，代表入站事件传播和处理
其余的方法代表出站事件的传播和处理。

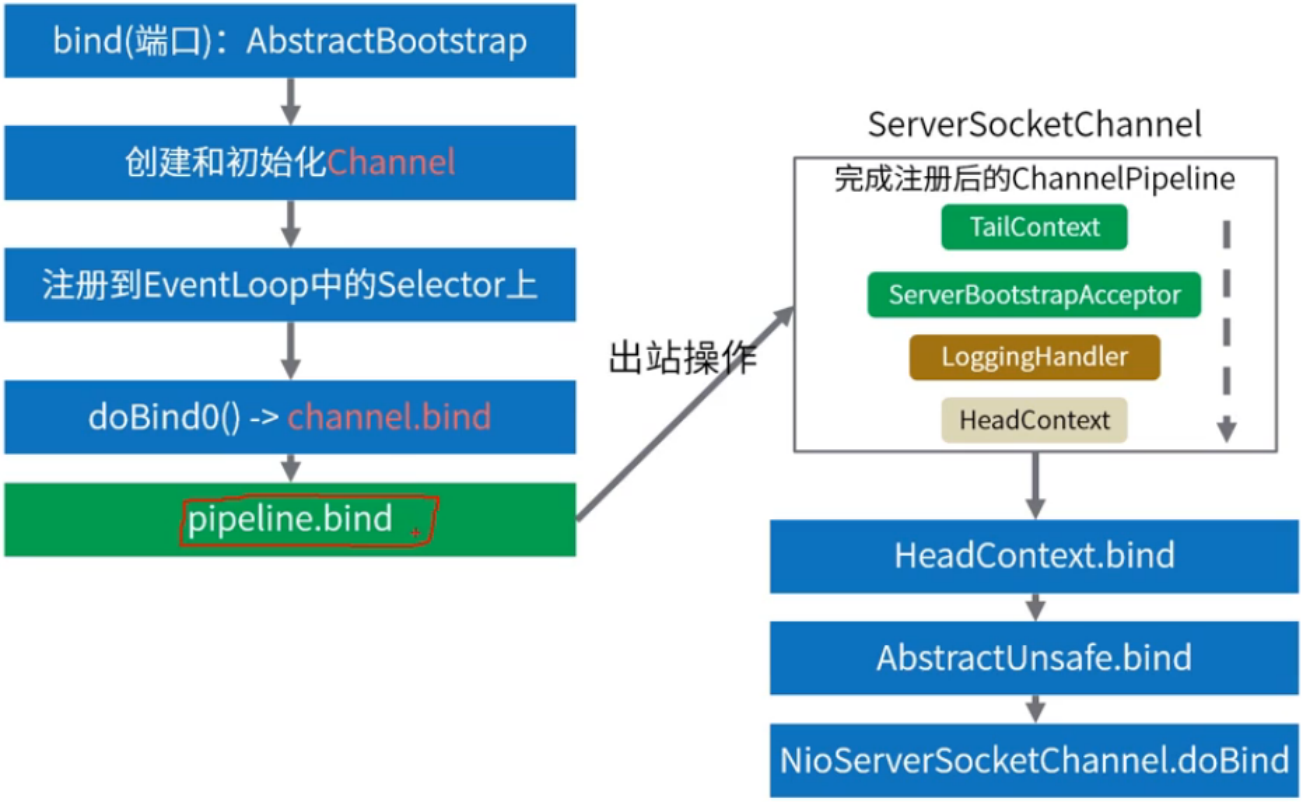
分析registered入站事件的处理



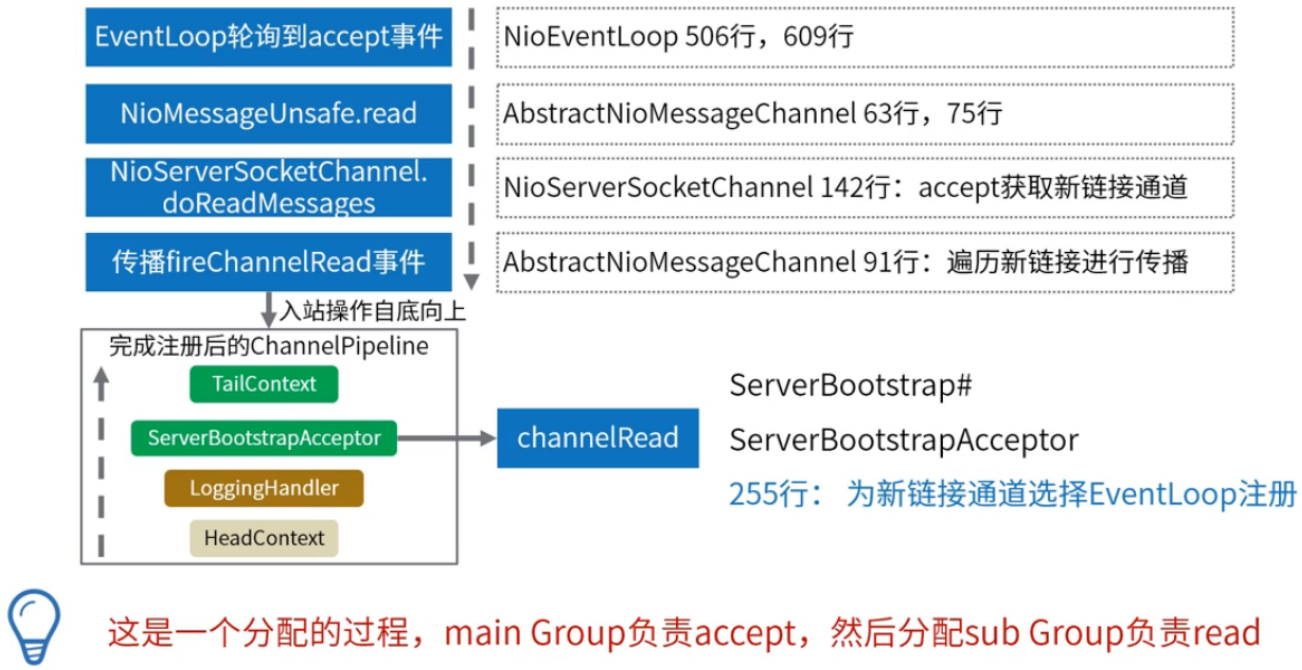
ServerSocketChannel.pipeline的变化



分析bind出站事件的处理



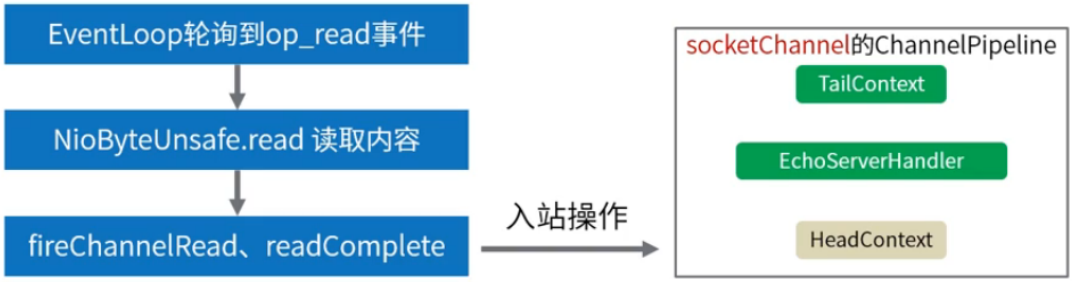
分析accept入站事件的处理



这是一个分配的过程，main Group负责accept，然后分配sub Group负责read

分析read入站事件的处理

pipeline分析的关键4要素：什么事件、有哪些处理器、哪些会被触发、执行顺序



小结

用户在管道中有一个或多个channelhandler来接收I/O事件(例如读取)和请求I/O操作(例如写入和关闭)。

一个典型的服务器在每个通道的管道中都有以下处理程序，但是根据协议和业务逻辑的复杂性和特征，可能会有所不同:

协议解码器——将二进制数据(例如ByteBuf)转换为Java对象。

协议编码器——将Java对象转换为二进制数据。

业务逻辑处理程序——执行实际的业务逻辑(例如数据库访问)。