Netty

> Netty是Java领域有名的开源网络库，特点是高性能和高扩展性，因此很多流行的框架都是基于它来构建的，比如我们熟知的Dubbo、Rocketmq、Hadoop等，针对高性能RPC，一般都是基于Netty来构建，比如sock-bolt。

1 Netty核心模块组件

1.1 Bootstrap、ServerBootstrap

Bootstrap意思是引导，一个Netty应用通常由一个Bootstrap开始，主要作用是配置整个Netty程序，串联各个组件，

1、Bootstrap类是客户端程序的启动引导类

Bootstrap 用于启动一个 Netty TCP 客户端，或者 UDP 的一端。

通常使用 #connet(...) 方法连接到远程的主机和端口，作为一个 Netty TCP 客户端。

也可以通过 #bind(...) 方法绑定本地的一个端口，作为 UDP 的一端。

仅仅需要使用一个 EventLoopGroup 。

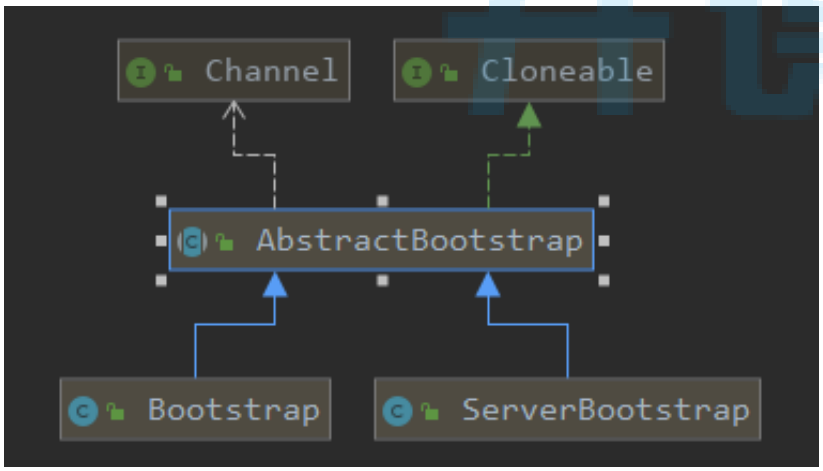
2、ServerBootstrap是服务端启动引导类

ServerBootstrap 往往是用于启动一个 Netty 服务端。

通常使用 #bind(...) 方法绑定本地的端口上，然后等待客户端的连接。

使用两个 EventLoopGroup 对象(当然这个对象可以引用同一个对象):

bossGroup 只是处理连接请求 , 真正的和客户端业务处理，会交给 workerGroup完成



> 这两个类都继承了AbstractBootstrap，因此它们有很多相同的方法和职责。它们都是**启动器**，能够帮助 Netty 使用者更加方便地组装和配置 Netty ，也可以更方便地启动 Netty 应用程序。相比使用者自己从头去将 Netty 的各部分组装起来要方便得多，降低了使用者的学习和使用成本。它们是我们使用 Netty 的入口和最重要的 API ，可以通过它来连接到一个主机和端口上，也可以通过它来绑定到一个本地的端口上。总的来说，它们两者之间相同之处要大于不同。

> Bootstrap & ServerBootstrap 对于 Netty ，就相当于 Spring Boot 是 Spring 的启动器。
>
> 它们和其它组件之间的关系是它们将 Netty 的其它组件进行组装和配置，所以它们会组合和直接或间接依赖其它的类。

常见的方法有

```
public ServerBootstrap group(EventLoopGroup parentGroup, EventLoopGroup childGroup)，该方法用于服务器端，用来设置两个 EventLoop
public B group(EventLoopGroup group) ，该方法用于客户端，用来设置一个 EventLoop

public B channel(Class<? extends C> channelClass)，该方法用来设置一个服务器端的通道实现

public <T> B option(ChannelOption<T> option, T value)，用来给 ServerChannel 添加配置

public <T> ServerBootstrap childOption(ChannelOption<T> childOption, T value)，用来给接收到的通道添加配置

xxx.handler(null) // 该 handler对应 bossGroup , childHandler 对应 workerGroup

public ServerBootstrap childHandler(ChannelHandler childHandler)，该方法用来设置业务处理类（自定义的 handler），childHandler 对应 workerGroup

public ChannelFuture bind(int inetPort) ，该方法用于服务器端，用来设置占用的端口号

public ChannelFuture connect(String inetHost, int inetPort) ，该方法用于客户端，用来连接服务器端
```

1.2 ChannelFuture

```
//b为ServerBootstrap实例
 ChannelFuture f = b.bind().sync();
```

Netty 中所有的 IO 操作都是异步的，不能立刻得知消息是否被正确处理。但是可以过一会等它执行完成或者直接注册一个监听，具体的实现就是通过 Future 和 ChannelFutures，他们可以注册一个监听，当操作执行成功或失败时监听会自动触发注册的监听事件

常见的方法有

```
Channel channel(), 返回当前正在进行 IO 操作的通道
ChannelFuture sync(), 等待异步操作执行完毕
```

1.3 Selector

Netty基于java.nio.channels.Selector对象实现IO多路复用，通过Selector一个线程可以监听多个连接的Channel事件。当向一个Selector中注册Channel后，Selector内部的机制就可以自动不断的Select这些注册的Channel是否有就绪的IO事件（可读、可写、网络连接完成等）。

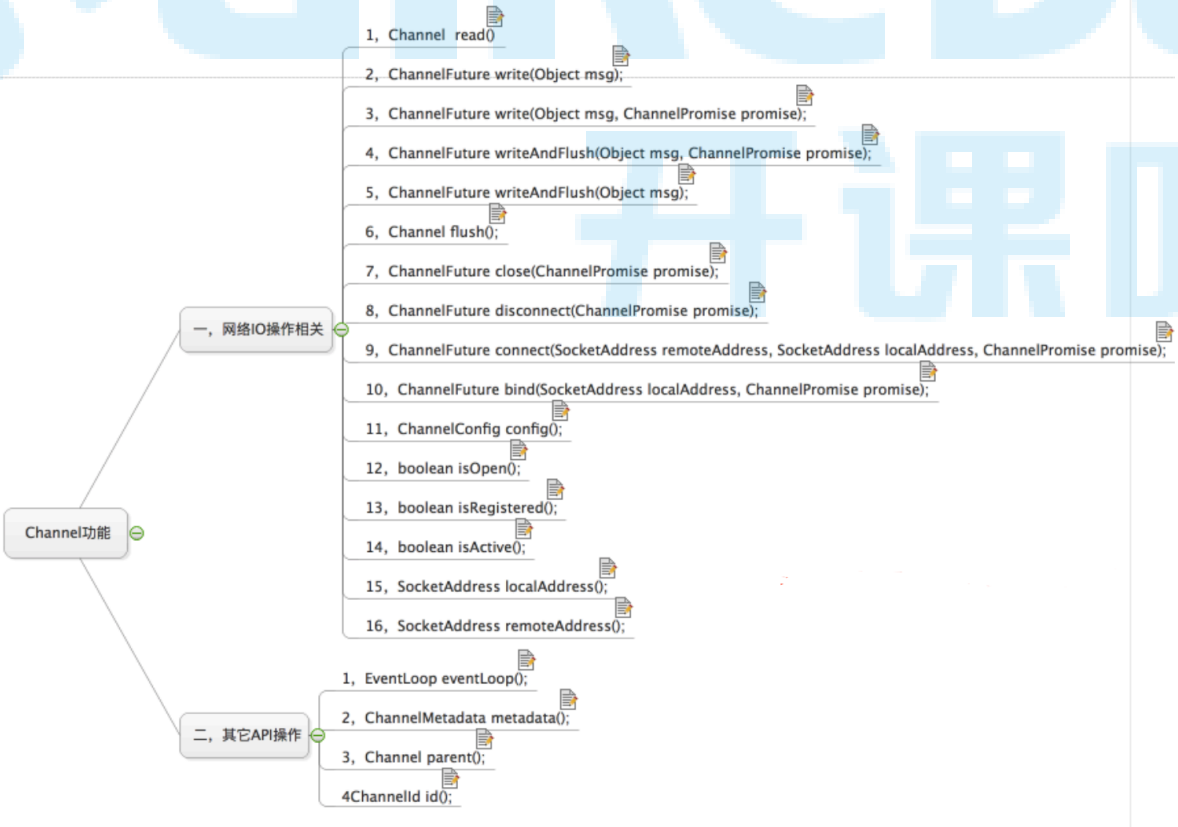
一个NioEventLoop中会有一个线程以及一个Selector, 这个线程就是我们所说的I/O线程

1.4 Channel

Channel 是 Netty 网络操作抽象类，使用了**Facade 模式**聚合了一组功能，除了包括基本的 I/O 操作，如 bind、connect、read、write 之外，还包括了 Netty 框架相关的一些功能，如获取该 Channel 的 EventLoop 。

1.4.1 Channel主要功能

1. 网络的读写
2. 客户端发起连接、主动关闭连接
3. 链路关闭
4. 获取通信双方的网络地址



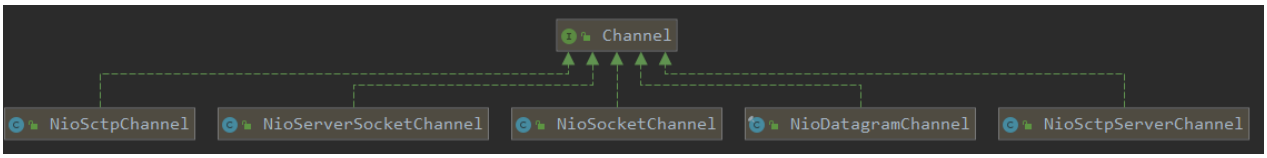
1.4.2 Channel设计理念

1.在Channel 接口层，采用 Facade 模式进行统一封装，讲网络I/O 操作、网络I/O 相关联的其他操作封装起来，统一对外提供。

2.Channel 接口的定义：**大而全**，为SocketChannel 和ServerSocketChannel 提供统一试图，由不同子类现实不同的功能，公共功能在抽象父类中实现，最大程度地实现功能和接口的重用。

3.具体实现采用聚合模式而非组合模式，将相关的功能类聚合在Channel中，由Channel 统一负责分配和调度，功能实现更加灵活。

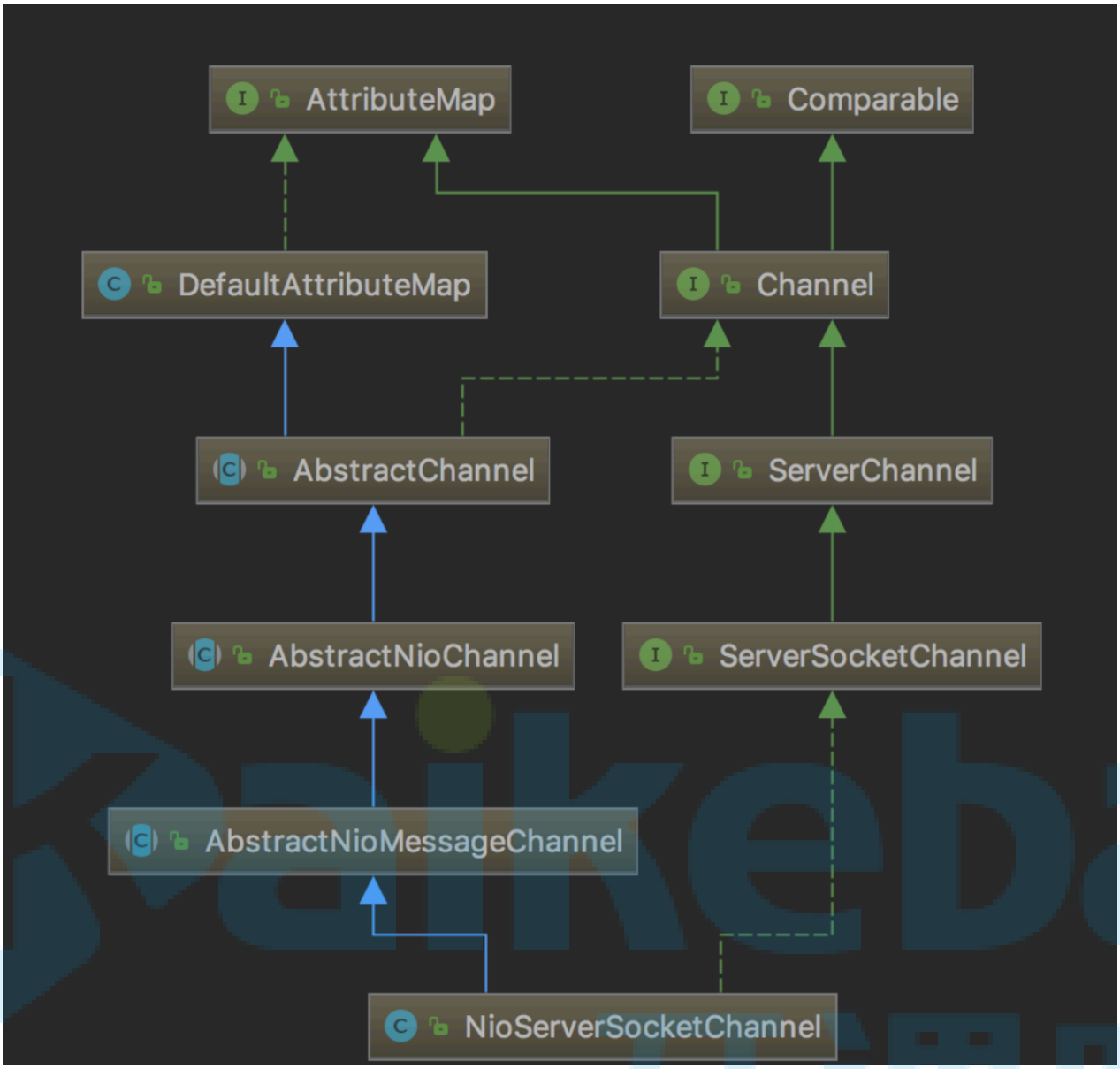
不同协议、不同的阻塞类型的连接都有不同的 Channel 类型与之对应，常用的 Channel 类型:



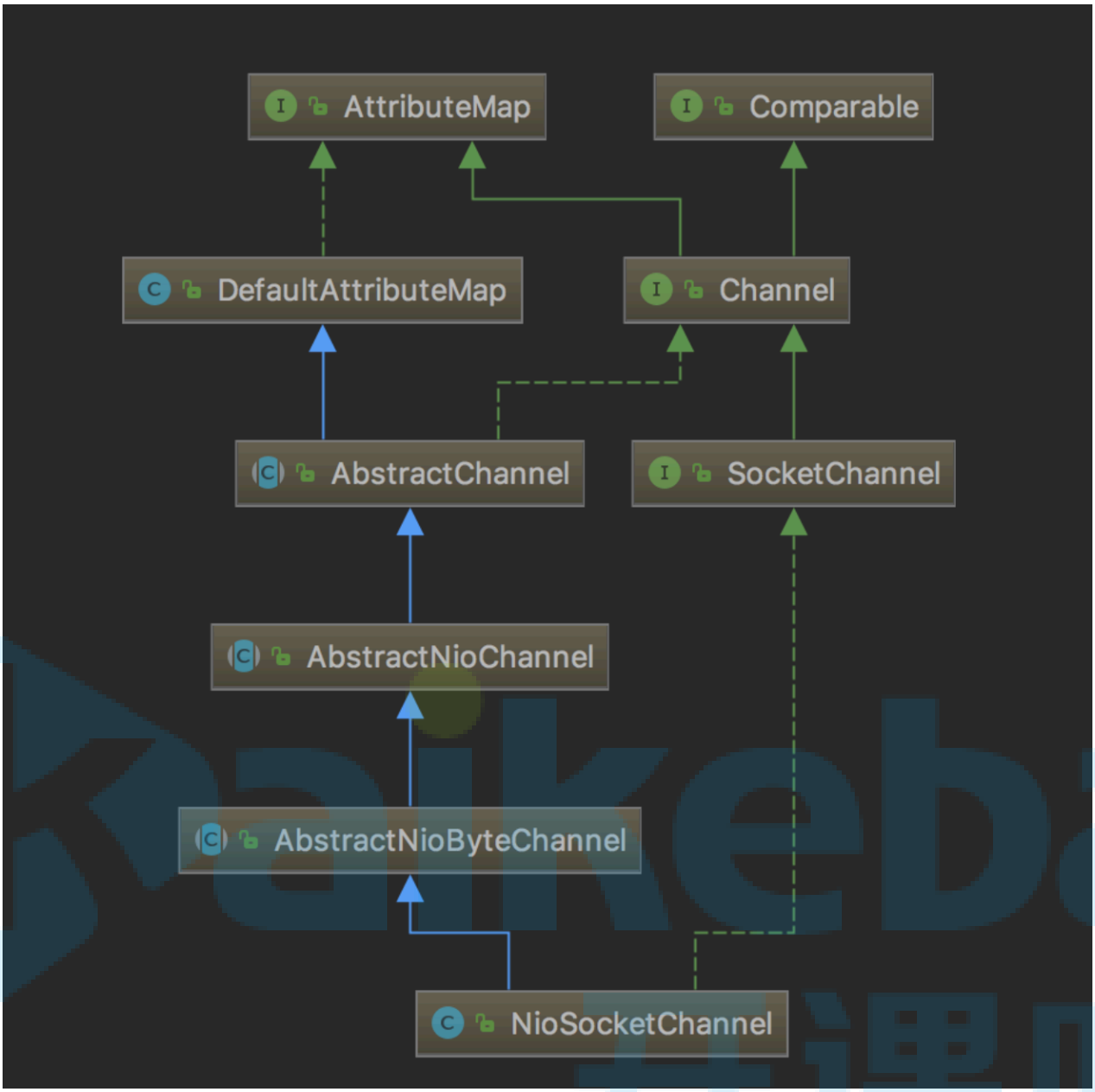

```
NioSocketChannel，异步的客户端 TCP Socket 连接。
NioServerSocketChannel，异步的服务器端 TCP Socket 连接。
NioDatagramChannel，异步的 UDP 连接。
NioSctpChannel，异步的客户端 Sctp 连接。
NioSctpServerChannel，异步的 Sctp 服务器端连接，这些通道涵盖了 UDP 和 TCP 网络 IO 以及文件 IO。
```

1.4.3 Channel继承关系

server端


AttributeMap
Comparable
DefaultAttributeMap
Channel
AbstractChannel
ServerChannel
AbstractNioChannel
ServerSocketChannel
AbstractNioMessageChannel
NioServerSocketChannel

client端：



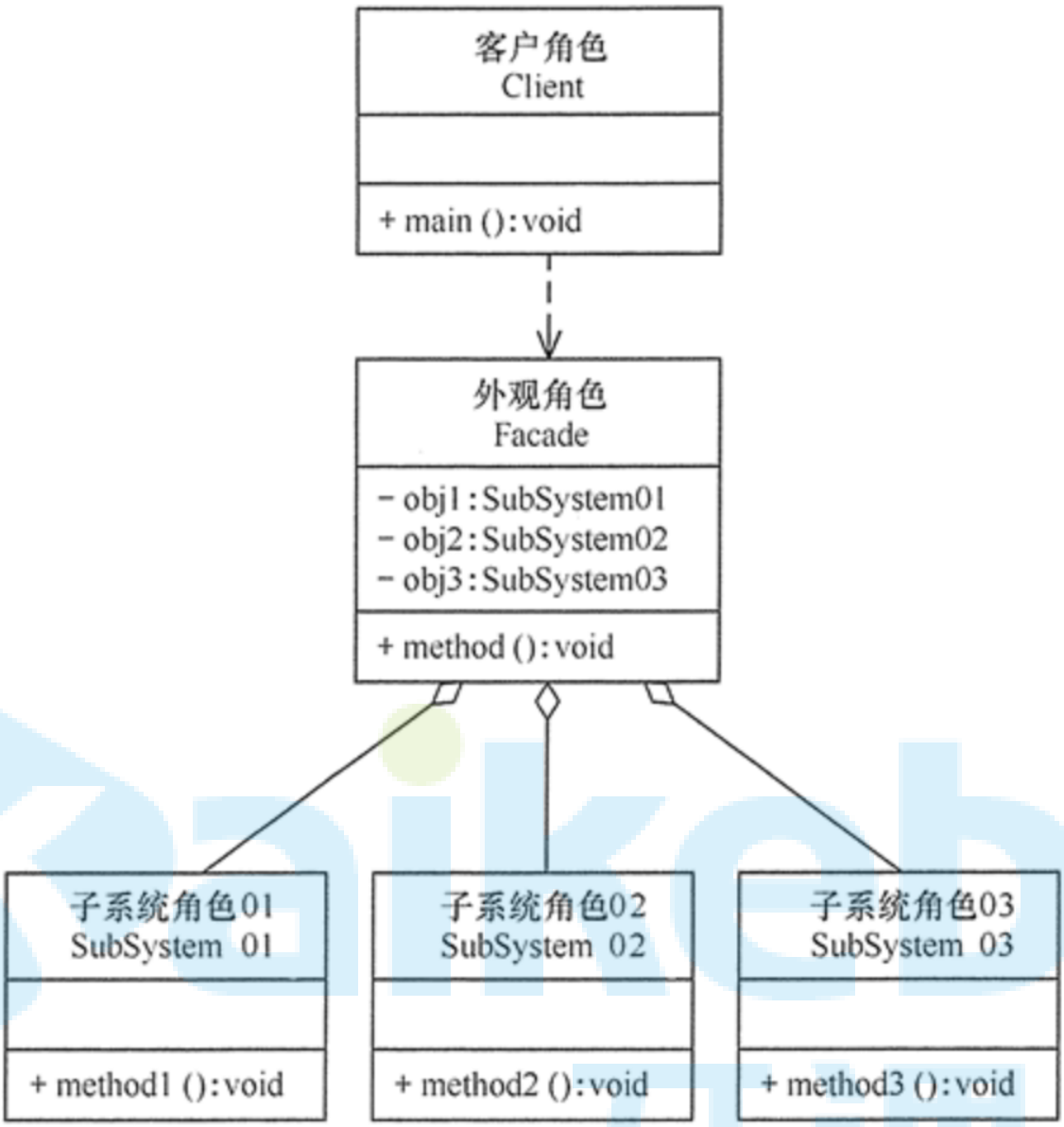
1.4.4 外观模式

外观（Facade）模式的定义：又叫门面模式，是一种通过**为多个复杂的子系统提供一个一致的接口**，而使这些子系统更加容易被访问的模式。该模式对外有一个统一接口，外部应用程序不用关心内部子系统的具体的细节，这样会大大降低应用程序的复杂度，提高了程序的可维护性。

外观（Facade）模式的结构比较简单，主要是定义了一个高层接口。它包含了对各个子系统的引用，客户端可以通过它访问各个子系统的功能。现在来分析其基本结构和实现方法。

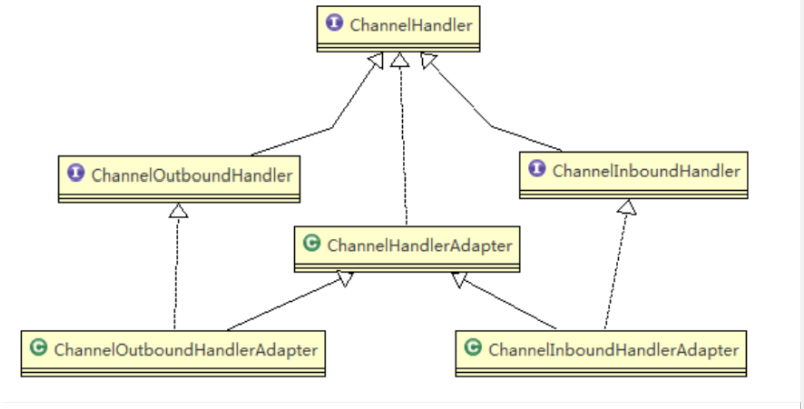
外观（Facade）模式包含以下主要角色。

- 外观（Facade）角色：为多个子系统对外提供一个共同的接口。
- 子系统（Sub System）角色：实现系统的部分功能，客户可以通过外观角色访问它。
- 客户（Client）角色：通过一个外观角色访问各个子系统的功能。



1.5 ChannelHandler

1. ChannelHandler属于业务的核心接口，处理 I/O 事件或拦截 I/O 操作，并将其转发到其 ChannelPipeline（业务处理链）。
2. ChannelHandler 本身并没有提供很多方法，因为这个接口有许多的方法需要实现，方便使用期间，可以继承它的子类
3. ChannelHandler 及其实现类一览图（后）



- ChannelInboundHandler 用于处理入站 I/O 事件。
- ChannelOutboundHandler 用于处理出站 I/O 操作。

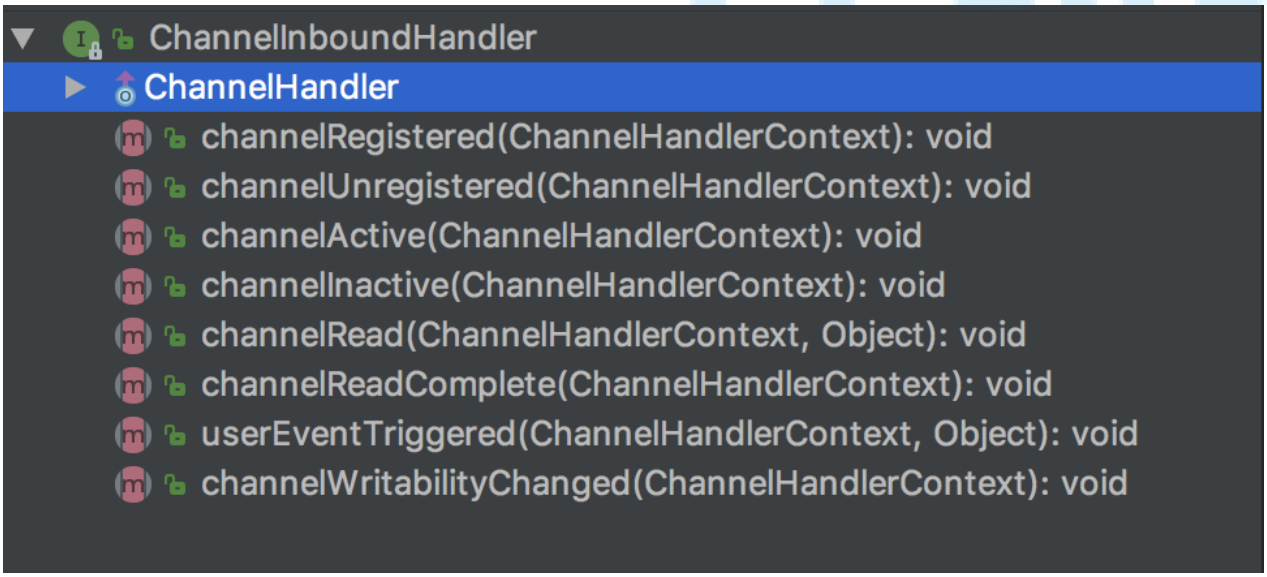
//适配器

- ChannelInboundHandlerAdapter 用于处理入站 I/O 事件。
- ChannelOutboundHandlerAdapter 用于处理出站 I/O 操作。
- ChannelDuplexHandler 用于处理入站和出站事件。

ChannelHandler里面定义三个生命周期方法，分别会在当前ChannelHander加入ChannelHandlerContext中，从ChannelHandlerContext中移除，以及ChannelHandler回调方法出现异常时被回调。

主要实现它的子接口ChannelInboundHandler和ChannelOutboundHandler，为了便利，框架提供了ChannelInboundHandlerAdapter，ChannelOutboundHandlerAdapter和ChannelDuplexHandler这三个适配类提供一些默认实现，在使用的时候只需要实现你关注的方法即可

ChannelInboundHandler

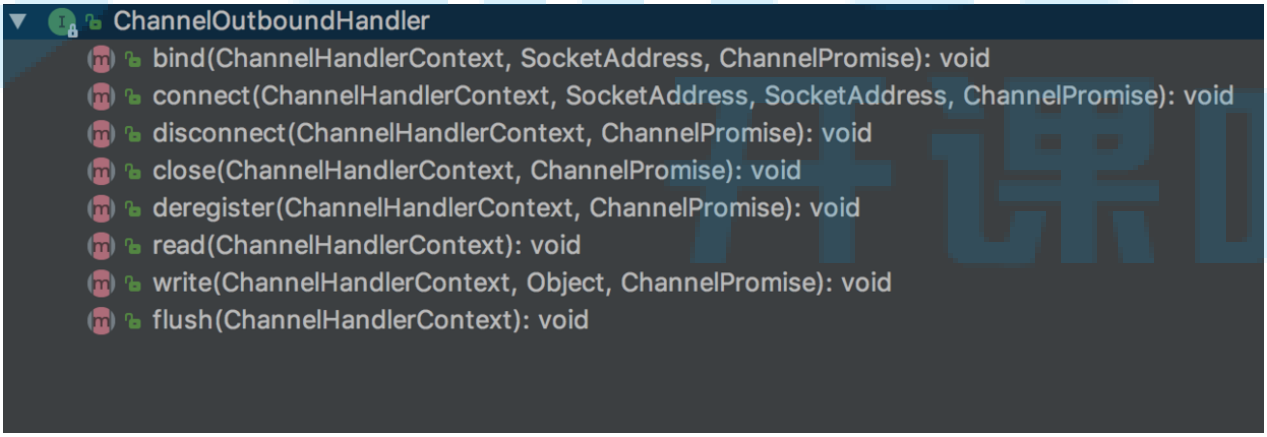


介绍一下这些回调方法被触发的时机

回调方法	触发时机
channelRegistered	当前channel注册到EventLoop
channelUnregistered	当前channel从EventLoop取消注册
channelActive	当前channel活跃的时候
channelInactive	当前channel不活跃的时候，也就是当前channel到了它生命周期末
channelRead	当前channel从远端读取到数据
channelReadComplete	channel read消费完读取的数据的时候被触发
userEventTriggered	用户事件触发的时候
channelWritabilityChanged	channel的写状态变化的时候触发

可以注意到每个方法都带了ChannelHandlerContext作为参数，具体作用是，在每个回调事件里面，处理完成之后，使用ChannelHandlerContext的fireChannelXXX方法来传递给下个ChannelHandler，netty的codec模块和业务处理代码分离就用到了这个链路处理

ChannelOutboundHandler



回调方法	触发时机
bind	bind操作执行前触发
connect	connect 操作执行前触发
disconnect	disconnect 操作执行前触发
close	close操作执行前触发
deregister	deregister操作执行前触发
read	read操作执行前触发
write	write操作执行前触发
flush	flush操作执行前触发

注意到一些回调方法有ChannelPromise这个参数，我们可以调用它的addListener注册监听，当回调方法所对应的操作完成后，会触发这个监听 下面这个代码，会在写操作完成后触发，完成操作包括成功和失败

```
public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise
promise) throws Exception {
    ctx.write(msg,promise);
    System.out.println("out write");
    promise.addListener(new GenericFutureListener<Future<? super Void>>() {
        @Override
        public void operationComplete(Future<? super Void> future) throws
Exception {
            if(future.isSuccess()){
                System.out.println("OK");
            }
        }
    });
}
```

ChannelInboundHandler和ChannelOutboundHandler的区别

ChannelInboundHandler的channelRead和channelReadComplete回调

ChannelInboundHandler的channelRead回调负责执行入栈数据的decode逻辑

ChannelOutboundHandler的write和flush回调上

ChannelOutboundHandler的write负责执行出站数据的encode工作

其他回调方法和具体触发逻辑有关，和in与out无关。

经常需要自定义一个 Handler 类去继承 ChannelInboundHandlerAdapter，然后通过重写相应方法实现业务逻辑，我们接下来看看一般都需要重写哪些方法

```
package com.kkb.demo.netty.example.simple;

public class NettyServerHandler extends ChannelInboundHandlerAdapter {

    .....
}
```

1.6 ChannelHandlerContext

保存 Channel 相关的所有上下文信息，同时关联一个 ChannelHandler 对象

即 ChannelHandlerContext 中包含一个具体的事件处理器 ChannelHandler，同时 ChannelHandlerContext 中也绑定了对应的 pipeline 和 Channel 的信息，方便对 ChannelHandler 进行调用。

每个ChannelHandler通过add方法加入到ChannelPipeline中去的时候，会创建一个对应的ChannelHandlerContext，并且绑定，ChannelPipeline实际维护的是ChannelHandlerContext 的关系。

每个ChannelHandlerContext之间形成**双向链表**

在DefaultChannelPipeline源码中可以看到会保存第一个ChannelHandlerContext以及最后一个ChannelHandlerContext的引用

```
public class DefaultChannelPipeline implements ChannelPipeline {
  .....

    final AbstractChannelHandlerContext head;
    final AbstractChannelHandlerContext tail;
  ....
}
```

而在AbstractChannelHandlerContext源码中可以看到

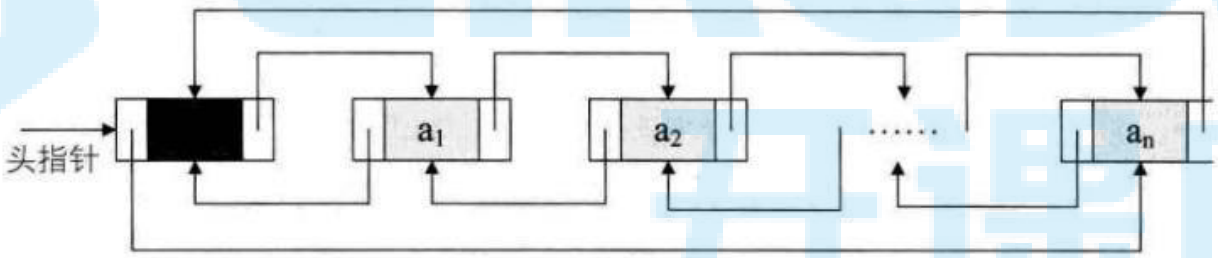
```
abstract class AbstractChannelHandlerContext implements ChannelHandlerContext,
ResourceLeakHint {
...........
volatile AbstractChannelHandlerContext next;
volatile AbstractChannelHandlerContext prev;

 ....
}
```

常用方法

- ChannelFuture close()，关闭通道
- ChannelOutboundInvoker flush()，刷新
- ChannelFuture writeAndFlush(Object msg)，将数据写到 ChannelPipeline 中当前
- ChannelHandler 的下一个 ChannelHandler 开始处理（出站）

双向链表(double linked list)

双向链表是在单链表的每个结点中，再设置一个指向其前驱结点的指针。所以在双向链表中的结点都有两个指针域：**一个指向直接后继，一个指向直接前驱**。



对于链表中的某个结点`p`，它的后继的前驱是它自己，同样，它的前驱的后继也是它自己

`p->next->prior = p;` `p->prior->next = p;`

1.7 ChannelPipeline

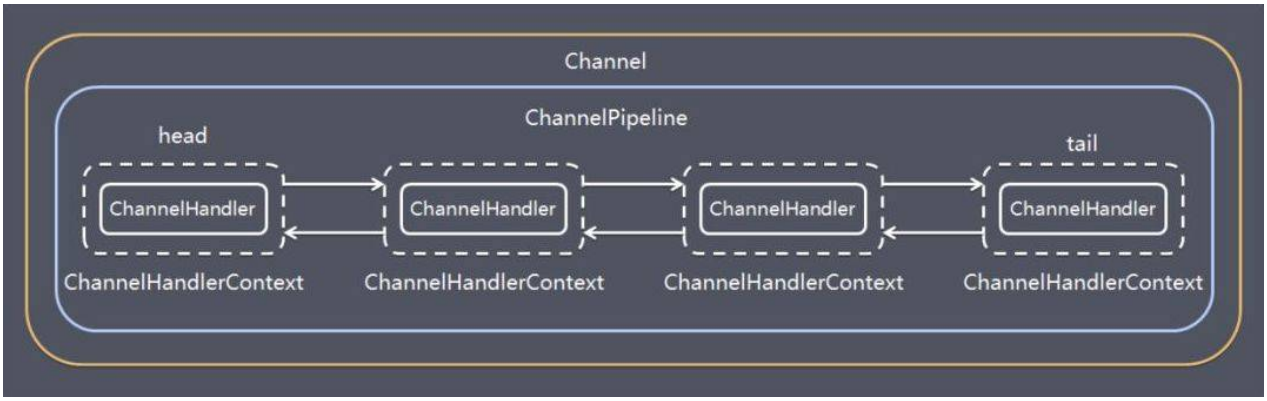
1.7.1 ChannelPipeline主要功能

ChannelPipeline 是一个重点：

1. ChannelPipeline 是一个 Handler 的集合，它负责处理和拦截 inbound 或者 outbound 的事件和操作，相当于一个贯穿 Netty 的链。（也可以这样理解：ChannelPipeline 是保存 ChannelHandler 的 List，用于处理或拦截 Channel 的入站事件和出站操作）
2. ChannelPipeline 实现了一种高级形式的拦截过滤器模式，使用户可以完全控制事件的处理方式，以及 Channel 中各个的 ChannelHandler 如何相互交互
3. **每个 Channel 都有且仅有一个 ChannelPipeline 与之对应，一个 Channel 包含了一个 ChannelPipeline，而 ChannelPipeline 中又维护了一个由 ChannelHandlerContext 组成的**

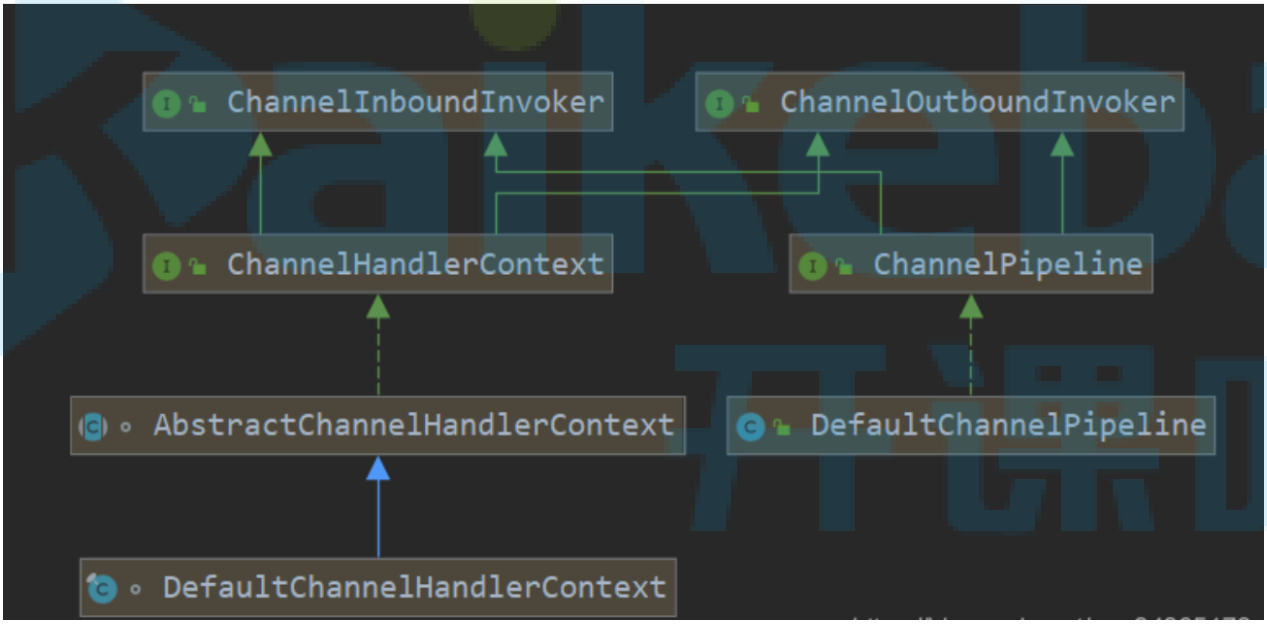
双向链表，并且每个 ChannelHandlerContext 中又关联着一个 ChannelHandler。

它们的组成关系如下



入站事件和出站事件在一个双向链表中，入站事件会从链表head往后传递到最后一个入站的handler，出站事件会从链表tail往前传递到最前一个出站的handler，两种类型的handler互不干扰。

1.7.2 ChannelPipeline初始化



在Channel创建的时候，会同时创建ChannelPipeline

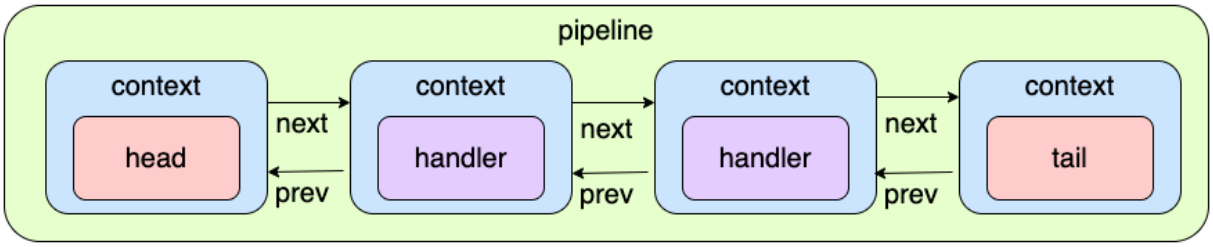
```
public abstract class AbstractChannel extends DefaultAttributeMap implements
Channel {
  .....
    private final DefaultChannelPipeline pipeline;
  .....
    protected AbstractChannel(Channel parent) {
        this.parent = parent;
        id = newId();
        unsafe = newUnsafe();
        pipeline = newChannelPipeline();
    }
  ....
}
```

在ChannelPipeline中也会持有Channel的引用

```
public class DefaultChannelPipeline implements ChannelPipeline {
....
private final Channel channel;
....
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);

        head.next = tail;
        tail.prev = head;
    }
    ....
    }
```

ChannelPipeline会维护一个ChannelHandlerContext的双向链表

```
public class DefaultChannelPipeline implements ChannelPipeline {
.....
final AbstractChannelHandlerContext head;
final AbstractChannelHandlerContext tail;
...
}
```

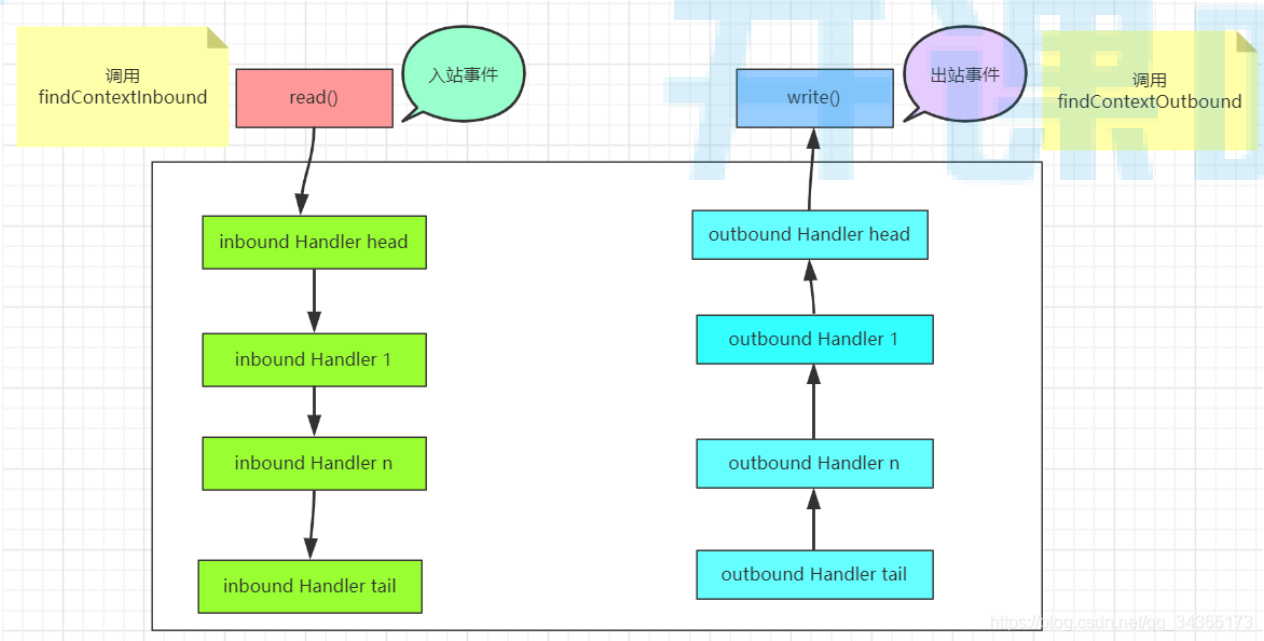


链表的头尾有默认实现

```
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);

        head.next = tail;
        tail.prev = head;
    }
```

1.7.3 入站事件和出站事件

pipeline保存了通道所有的处理器信息，在创建一个channel的时候，会创建一个这个channel专有的pipeline，入站事件和出站事件都会调用这个pipeline上面的处理器。

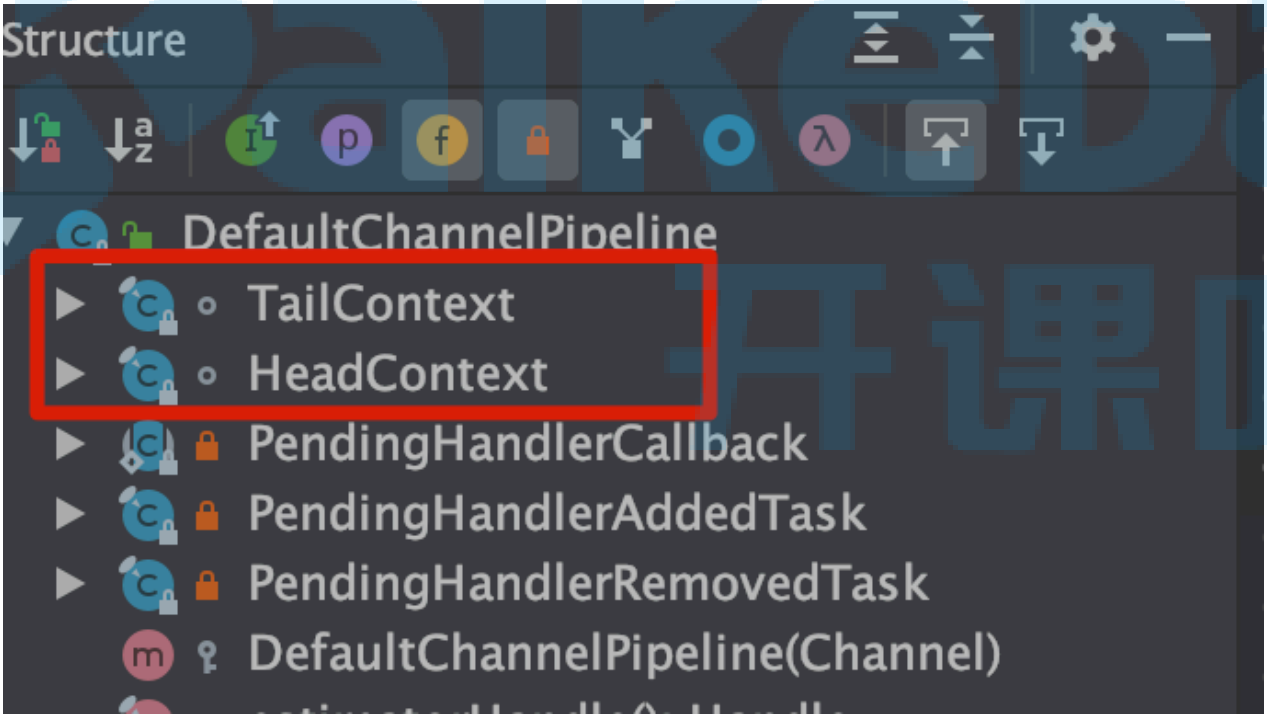



```
abstract class AbstractChannelHandlerContext implements ChannelHandlerContext,
ResourceLeakHint {
..........
 private AbstractChannelHandlerContext findContextInbound(int mask) {
```

```
        AbstractChannelHandlerContext ctx = this;
        EventExecutor currentExecutor = executor();
        do {
            ctx = ctx.next;
        } while (skipContext(ctx, currentExecutor, mask, MASK_ONLY_INBOUND));
        return ctx;
    }

    private AbstractChannelHandlerContext findContextOutbound(int mask) {
        AbstractChannelHandlerContext ctx = this;
        EventExecutor currentExecutor = executor();
        do {
            ctx = ctx.prev;
        } while (skipContext(ctx, currentExecutor, mask, MASK_ONLY_OUTBOUND));
        return ctx;
    }
    ..........
```

上面两个方法的作用，是判断下一个context是不是入站或者出站事件，是的话才往下传递数据。

1.7.4 HeadContext



HeadContext实现了ChannelOutboundHandler，ChannelInboundHandler这两个接口

```
final class HeadContext extends AbstractChannelHandlerContext
            implements ChannelOutboundHandler, ChannelInboundHandler {
```

因为在头部，所以说HeadContext中关于in和out的回调方法都会触发关于ChannelInboundHandler，HeadContext的作用是进行一些前置操作，以及把事件传递到下一个ChannelHandlerContext的ChannelInboundHandler中去 看下其中channelRegistered的实现

```
@Override
        public void channelRegistered(ChannelHandlerContext ctx) {
            invokeHandlerAddedIfNeeded();
            ctx.fireChannelRegistered();
        }
```

从语义上可以看出来在把这个事件传递给下一个ChannelHandler之前会回调ChannelHandler的handlerAdded方法 而有关ChannelOutboundHandler接口的实现，会在链路的最后执行，看下write方法的实现

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

这边的unsafe接口封装了底层Channel的调用，之所以取名为unsafe，是不需要用户手动去调用这些方法。

> 这个和阻塞原语的unsafe不是同一个

1.7.5 TailContext

TailContext实现了ChannelInboundHandler接口，会在ChannelInboundHandler调用链最后执行，只要是对调用链完成处理的情况进行处理，看下channelRead实现

```
    final class TailContext extends AbstractChannelHandlerContext implements
ChannelInboundHandler {
public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
    onUnhandledInboundMessage(msg);
}
}
```

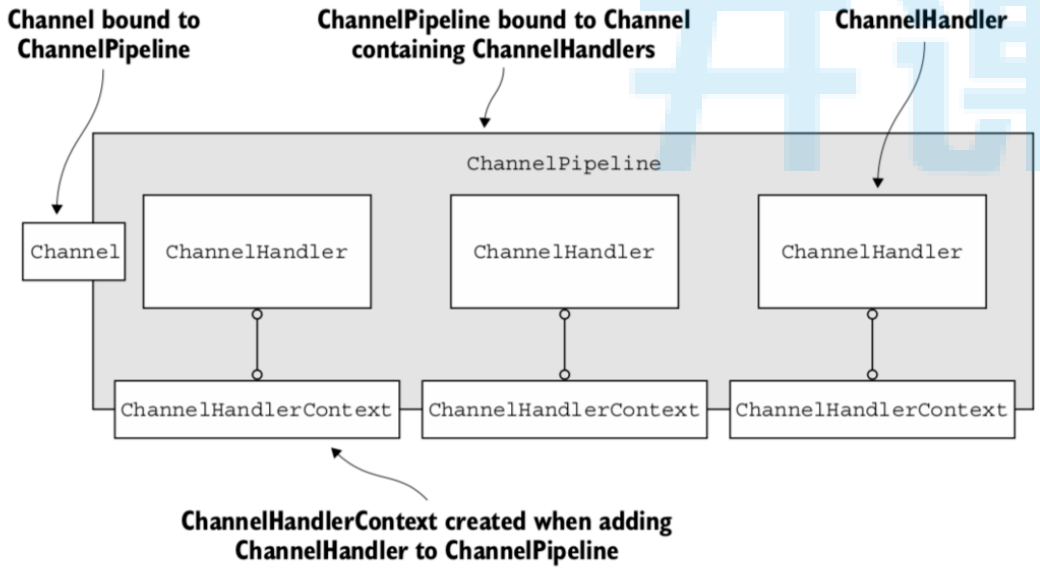
如果我们自定义的最后一个ChannelInboundHandler，也把处理操作交给下一个ChannelHandler，那么就会到TailContext，在TailContext会提供一些默认处理

```
protected void onUnhandledInboundMessage(ChannelHandlerContext ctx, Object
msg) {
        onUnhandledInboundMessage(msg);
        if (logger.isDebugEnabled()) {
            logger.debug("Discarded message pipeline : {}. Channel : {}.",
                         ctx.pipeline().names(), ctx.channel());
        }
    }
```

比如channelRead中的onUnhandledInboundMessage方法，会把msg资源回收，防止内存泄露

```
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);
        }
    }
```

1.7.6 组件之间的关系



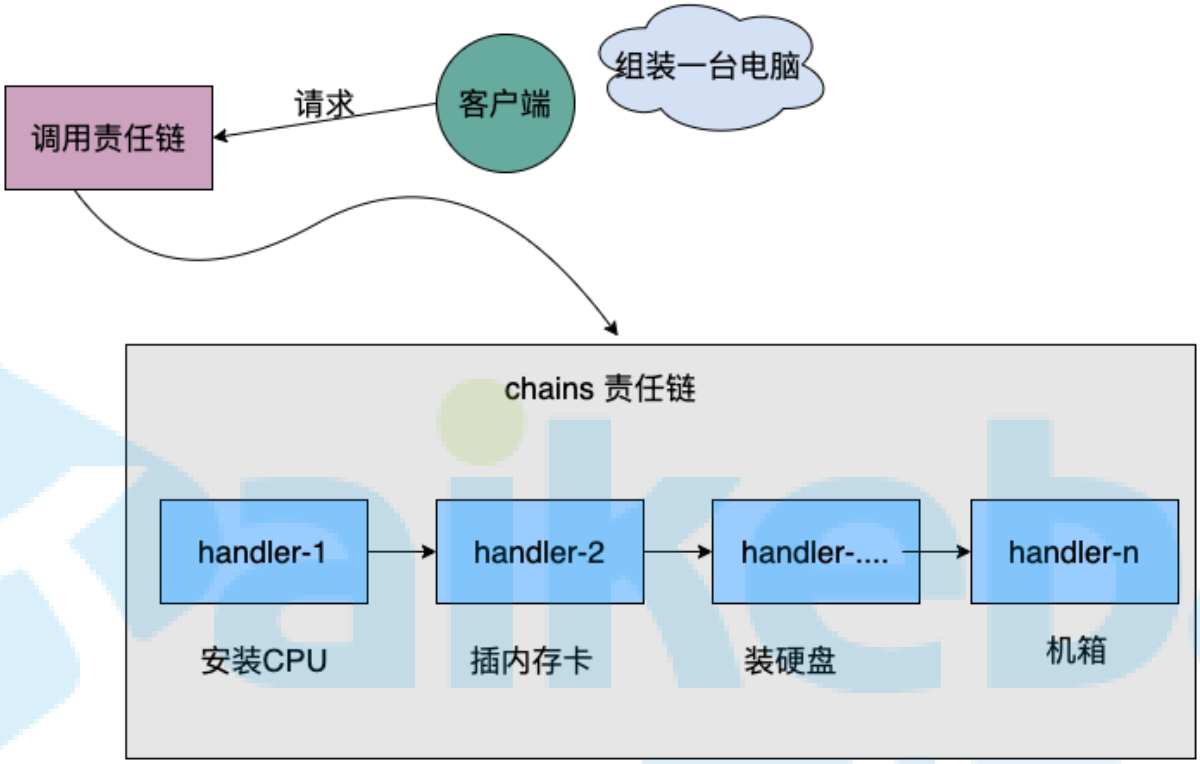
1.每个Channel会绑定一个ChannelPipeline，每个ChannelPipeline会持有一个Channel

2.每个ChannelHandler对应一个ChannelHandlerContext，ChannelPipeline持有ChannelHandlerContext链表，也就相当于持有ChannelHandler链表

3.ChannelHandlerContext作为上下文，持有ChannelPipeline和它对应ChannelHandler的引用，持有ChannelPipeline相当于间接持有Channel，同时持有它上/下一个ChannelHandlerContext的引用

1.7.7 责任链模式

责任链模式为请求创建一个处理数据的链。 客户端发起的请求和具体处理请求的过程进行了解耦，责任链上的处理者负责处理请求，客户端只需要把请求发送到责任链就行了，不需要去关心具体的处理逻辑和处理请求在责任链中是怎样传递的。



责任链模式的简单实现

责任链模式的实现，需要4个关键要素：

1、处理器抽象类

2、处理器抽象类的具体实现类

3、保存和维护处理器信息的类

4、处理器执行的类

下面看一个简单的demo，基于责任链模式的思想：

```
public class PipelineDemo {
    //初始化链的头部
    public HandlerContext head = new HandlerContext(new AbstractHandler() {
        @Override
        void doHandler(HandlerContext context, Object arg) {
            context.runNext(arg);
        }
    });
```

```
    //开始执行
    public void request(Object arg) {
        this.head.handler(arg);
    }

    //添加节点到尾部
    public void addLast(AbstractHandler handler) {
        HandlerContext context = head;
        while (context.next != null) {
            context = context.next;
        }
        context.next = new HandlerContext(handler);
    }


    public static void main(String[] args) {
        PipelineDemo pipelineChainDemo = new PipelineDemo();
        pipelineChainDemo.addLast(new Handler2());
        pipelineChainDemo.addLast(new Handler1());
        pipelineChainDemo.addLast(new Handler1());
        pipelineChainDemo.addLast(new Handler2());

        // 发起请求
        pipelineChainDemo.request("火车呜呜呜~~");
    }

}

//处理器的信息，维护处理器
class HandlerContext {
    //下一个节点
    HandlerContext next;
    AbstractHandler handler;

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

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

    //执行下一个
    void runNext(Object arg) {
        if (this.next != null) {
            this.next.handler(arg);
        }
    }
```

```
}

//处理器抽象类
abstract class AbstractHandler {
    abstract void doHandler(HandlerContext context, Object arg);
}

//处理器的具体实现类
class Handler1 extends AbstractHandler {

    @Override
    void doHandler(HandlerContext context, Object arg) {
        arg = arg.toString() + "Handler1的小尾巴～～";
        System.out.println("Handler1的实例正在处理: " + arg);
        //执行下一个
        context.runNext(arg);
    }
}

//处理器的具体实现类
class Handler2 extends AbstractHandler {

    @Override
    void doHandler(HandlerContext context, Object arg) {
        arg = arg.toString() + "Handler2的小尾巴～～";
        System.out.println("Handler2的实例正在处理: " + arg);
        //执行下一个
        context.runNext(arg);
    }
}
```

1.8 ChannelOption

ChannelConfig是Channel的配置类， 而ChannelConfig内部各种配置选项依赖于ChannelOption类的实现，可以认为ChannelConfig中用了一个Map来保存参数，Map的key是ChannelOption，ChannelConfig 定义了相关方法来获取和修改Map中的值。

```
  public interface ChannelConfig {
...
    Map<ChannelOption<?>, Object> getOptions();//获取所有参数
   boolean setOptions(Map<ChannelOption<?>, ?> options);//替换所有参数
   <T> T getOption(ChannelOption<T> option);//获取以某个ChannelOption为key的参数值
   <T> boolean setOption(ChannelOption<T> option, T value);//替换某个
ChannelOption为key的参数值
....
}
```

ChannelOption定义了对一个Channel的各种属性配置选项，包括了各种底层连接的详细信息，如keep-alive或者超时属性以及缓冲区的设置等。

ChannelOption 参数如下：

ChannelOption.SO_BACKLOG
对应 TCP/IP 协议 listen 函数中的 backlog 参数，用来初始化服务器可连接队列大小。服务端处理客户端连接请求是顺序处理的，所以同一时间只能处理一个客户端连接。多个客户端来的时候，服务端将不能处理的客户端连接请求放在队列中等待处理，backlog 参数指定了队列的大小。

ChannelOption.SO_KEEPALIVE
一直保持连接活动状态

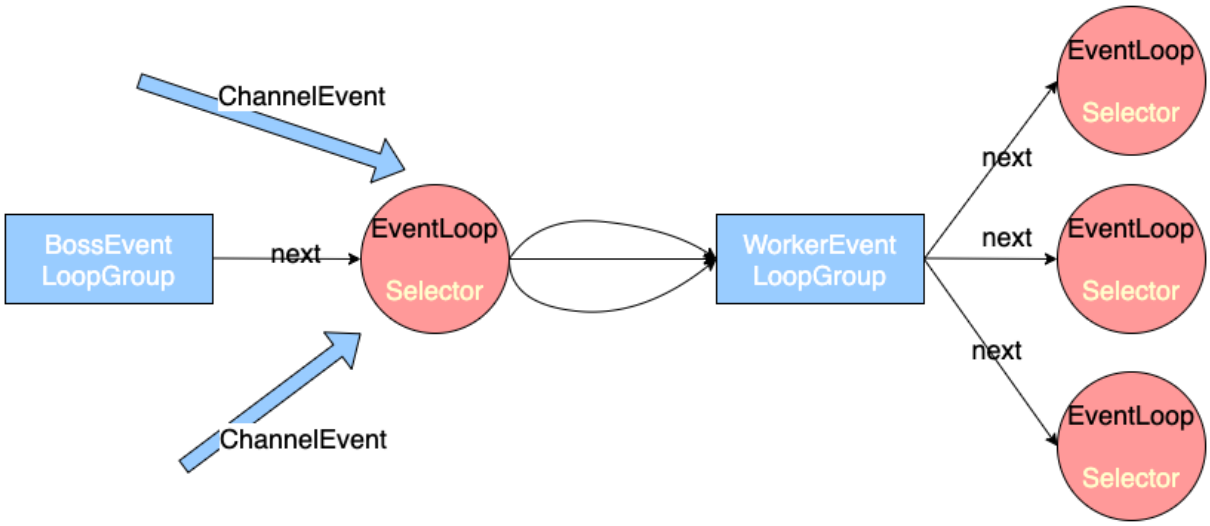
1.9 EventLoopGroup 和 NioEventLoopGroup

1.9.1 ServerSocketChannel 与 SocketChannel

EventLoopGroup 是一组 EventLoop 的抽象，Netty 为了更好的利用多核 CPU 资源，一般会有多个 EventLoop 同时工作，每个 EventLoop 维护着一个 Selector 实例。

EventLoopGroup 提供 next 接口，可以从组里面按照一定规则获取其中一个 EventLoop 来处理任务。在 Netty 服务器端编程中，我们一般都需要提供两个 EventLoopGroup，例如：BossEventLoopGroup 和 WorkerEventLoopGroup。

通常一个服务端口即一个 ServerSocketChannel 对应一个 Selector 和一个 EventLoop 线程。BossEventLoop 负责接收客户端的连接并将 SocketChannel 交给 WorkerEventLoopGroup 来进行 IO 处理，如下图所示



BossEventLoopGroup通常是一个单线程的EventLoop，EventLoop维护着一个注册了ServerSocketChannel的Selector实例BossEventLoop不断轮询Selector将连接事件分离出来。

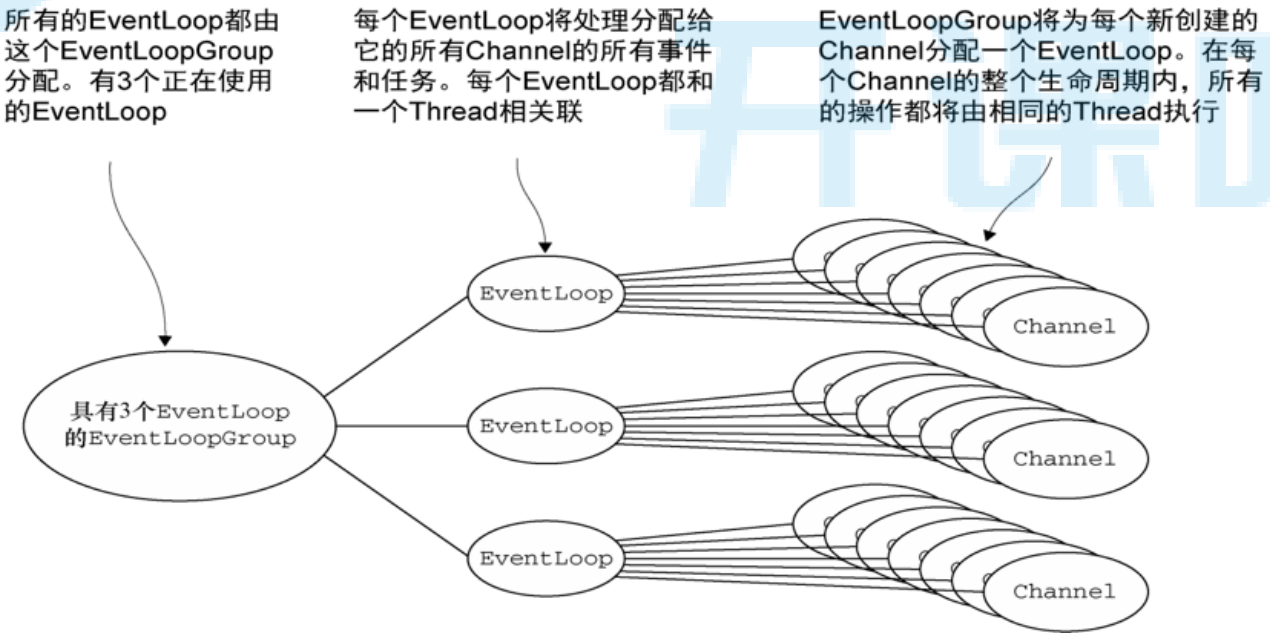
通常是OP_ACCEPT事件，然后将受到SocketChannel交给WokerEventLoopGroup

WokerEventLoopGroup会有next选择其中一个EventLoop来将这个SocketChannel注册到其维护的Selector并对其后续的IO事件进行处理。

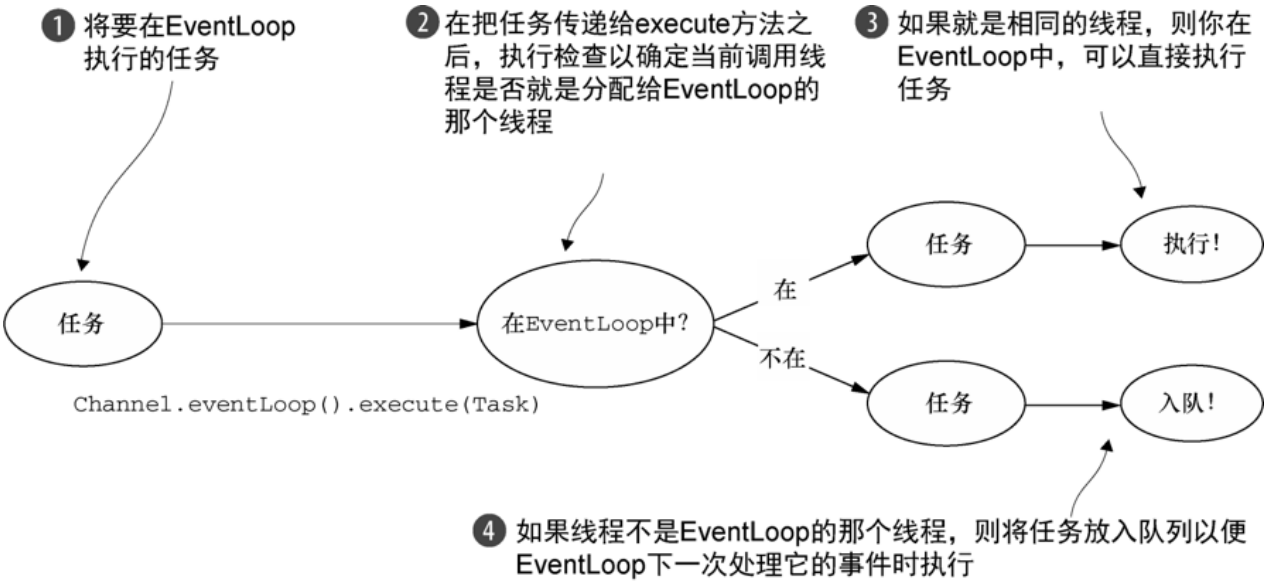
1.9.2 EventLoop与Channel

EventLoop进行的是Selector的维护。

EventLoopGroup用于线程组维护，并发控制，任务处理。



1.9.3 任务执行



关于EventLoop以及EventLoopGroup的映射关系为：

- 一个EventLoopGroup 包含一个或者多个EventLoop；
- 一个EventLoop 在它的生命周期内只和一个Thread 绑定；
- 所有由EventLoop 处理的I/O 事件都将在它专有的Thread 上被处理；
- 一个Channel 在它的生命周期内只注册于一个EventLoop；
- 一个EventLoop 可能会被分配给一个或多个Channel。

Channel 为Netty 网络操作抽象类，EventLoop 主要是为Channel 处理 I/O 操作，两者配合参与 I/O 操作。当一个连接到达时，Netty 就会注册一个 Channel，然后从 EventLoopGroup 中分配一个 EventLoop 绑定到这个Channel上，在该Channel的整个生命周期中都是有这个绑定的 EventLoop 来服务的。

1.9.4 常用方法

```
public NioEventLoopGroup(), 构造方法

public Future<?> shutdownGracefully(), 断开连接，关闭线程
```

1.10 ByteBuf

1.10.1 Unpooled

Netty 提供一个专门用来操作缓冲区（即 Netty 的数据容器）的工具类Unpooled

常用方法

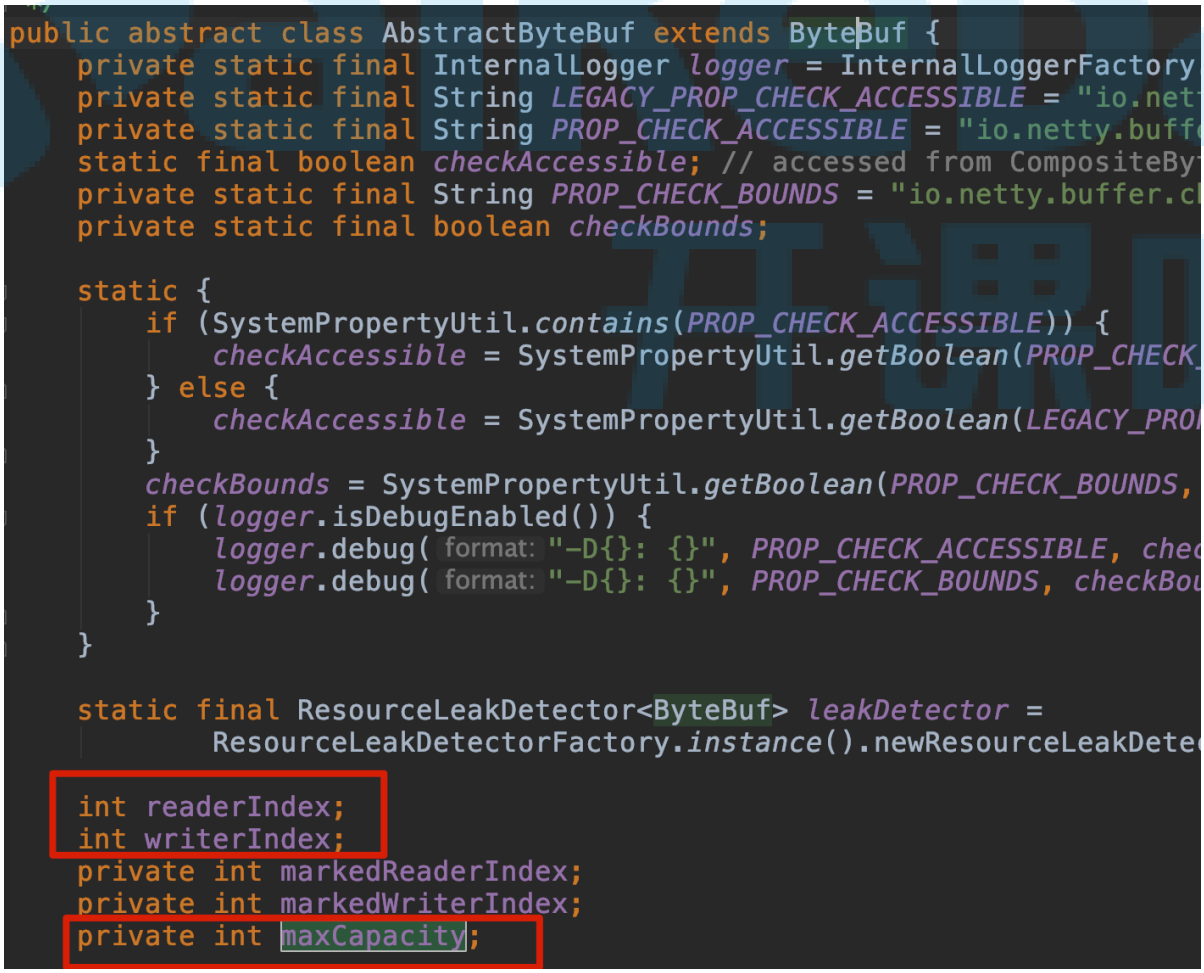
```
/**
通过给定的数据和字符编码返回一个ByteBuf 对象(类似NIO中的ByteBuffer，但是有区别)
**/
```

```
public static ByteBuf copiedBuffer(CharSequence string, Charset charset) {
        ObjectUtil.checkNotNull(string, "string");
        if (CharsetUtil.UTF_8.equals(charset)) {
            return copiedBufferUtf8(string);
        }
        if (CharsetUtil.US_ASCII.equals(charset)) {
            return copiedBufferAscii(string);
        }
        if (string instanceof CharBuffer) {
            return copiedBuffer((CharBuffer) string, charset);
        }

        return copiedBuffer(CharBuffer.wrap(string), charset);
    }
```

1.10.2 ByteBuf的三个指针

分别是：

- readerIndex（读指针）
- writerIndex（写指针）
- maxCapacity（最大容量）



1.10.3 示例

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

```
import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;

public class NettyByteBuf01 {
    public static void main(String[] args) {

        //创建一个ByteBuf
        //说明
        //1. 创建 对象，该对象包含一个数组arr , 是一个byte[10]
        //2. 在netty 的buffer中，不需要使用flip 进行反转
        //   底层维护了 readerindex 和 writerIndex
        //3. 通过 readerindex 和  writerIndex 和  capacity， 将buffer分成三个区域
        // 0---readerindex 已经读取的区域
        // readerindex---writerIndex ，  可读的区域
        // writerIndex -- capacity, 可写的区域
        ByteBuf buffer = Unpooled.buffer(10);

        for (int i = 0; i < 10; i++) {
            System.out.println("writerIndex=" + buffer.writerIndex());//10
            buffer.writeByte(i);

        }

        System.out.println("capacity=" + buffer.capacity());//10


        while (buffer.isReadable()){

 System.out.println(buffer.readByte()+",readerIndex="+buffer.readerIndex());

        }
        System.out.println("=======");
        //输出
```

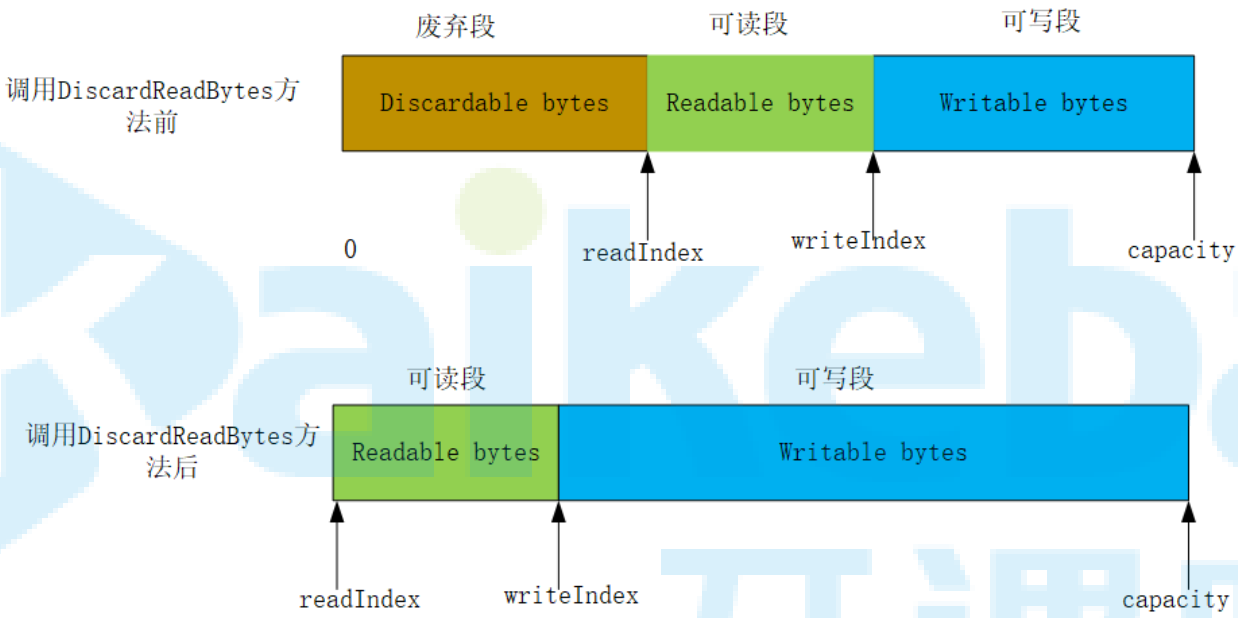
```
        for(int i = 0; i<buffer.capacity(); i++) {

 System.out.println(buffer.getByte(i)+",readerIndex="+buffer.readerIndex());
        }

        System.out.println("done");
    }
}
```

示例 2

```
import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;

import java.nio.charset.Charset;

public class NettyByteBuf02 {
    public static void main(String[] args) {

        //创建ByteBuf
        ByteBuf byteBuf = Unpooled.copiedBuffer("hello,world!",
Charset.forName("utf-8"));

        //使用相关的方法
        if (byteBuf.hasArray()) { // true

            byte[] content = byteBuf.array();

            //将 content 转成字符串
            System.out.println(new String(content, Charset.forName("utf-8")));

            System.out.println("byteBuf=" + byteBuf);

            System.out.println(byteBuf.arrayOffset()); // 0
            System.out.println(byteBuf.readerIndex()); // 0
            System.out.println(byteBuf.writerIndex()); // 12
            System.out.println(byteBuf.capacity()); // 36

            //System.out.println(byteBuf.readByte()); //
            System.out.println(byteBuf.getByte(0)); // 104
            System.out.println((char)byteBuf.getByte(0)); // h


            int len = byteBuf.readableBytes(); //可读的字节数  12
            System.out.println("len=" + len);

            //使用for取出各个字节
            for (int i = 0; i < len; i++) {
                System.out.println((char) byteBuf.getByte(i));
```

```
            }

            //按照某个范围读取
            System.out.println(byteBuf.getCharSequence(0, 4,
Charset.forName("utf-8")));
            System.out.println(byteBuf.getCharSequence(4, 6,
Charset.forName("utf-8")));
        }
    }
}
```

1.10.4 discardReadBytes



从上面的图中可以观察到，调用discardReadBytes方法后，readIndex置为0，writeIndex也往前移动了Discardable bytes长度的距离，扩大了可写区域。但是这种做法会**严重影响效率**，它进行了大量的拷贝工作。如果要进行数据的清除操作，建议使用**clear**方法。调用**clear()**方法将会将readIndex和writeIndex同时置为0，不会进行内存的拷贝工作，同时要注意，clear方法不会清除内存中的内容,只是改变了索引位置而已。

1.11 群聊系统

实例要求：

1. 编写一个 Netty 群聊系统，实现服务器端和客户端之间的数据简单通讯（非阻塞）
2. 实现多人群聊
3. 服务器端：可以监测用户上线，离线，并实现消息转发功能
4. 客户端：通过 `channel` 可以无阻塞发送消息给其它所有用户，同时可以接受其它用户发送的消息（有服务器转发得到）
5. 目的：进一步理解 Netty 非阻塞网络编程机制

代码如下：

```
import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.*;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.handler.codec.string.StringDecoder;
import io.netty.handler.codec.string.StringEncoder;

public class GroupChatServer {
    private int port; //监听端口

    public GroupChatServer(int port) {
        this.port = port;
    }

    //编写run方法，处理客户端的请求
    public void run() throws Exception {

        //创建两个线程组
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workerGroup = new NioEventLoopGroup(); //16个
NioEventLoop

        try {
            ServerBootstrap b = new ServerBootstrap();

            b.group(bossGroup, workerGroup)
                    .channel(NioServerSocketChannel.class)
                    .option(ChannelOption.SO_BACKLOG, 128)
                    .childOption(ChannelOption.SO_KEEPALIVE, true)
                    .childHandler(new ChannelInitializer<SocketChannel>() {

                        @Override
                        protected void initChannel(SocketChannel ch) throws
Exception {

                            //获取到pipeline
                            ChannelPipeline pipeline = ch.pipeline();
                            //向pipeline加入解码器
                            pipeline.addLast("decoder", new StringDecoder());
                            //向pipeline加入编码器
                            pipeline.addLast("encoder", new StringEncoder());
                            //加入自己的业务处理handler
                            pipeline.addLast(new GroupChatServerHandler());
```

```
                    }
                });

            System.out.println("netty 服务器启动");
            ChannelFuture channelFuture = b.bind(port).sync();

            //监听关闭
            channelFuture.channel().closeFuture().sync();
        } finally {
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }

    public static void main(String[] args) throws Exception {

        new GroupChatServer(7000).run();
    }
}


import io.netty.channel.Channel;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.channel.group.ChannelGroup;
import io.netty.channel.group.DefaultChannelGroup;
import io.netty.util.concurrent.GlobalEventExecutor;

import java.text.SimpleDateFormat;

public class GroupChatServerHandler extends
SimpleChannelInboundHandler<String> {
    //public static List<Channel> channels = new ArrayList<Channel>();

    //使用一个hashmap 管理
    //public static Map<String, Channel> channels = new
HashMap<String,Channel>();

    //定义一个channle 组，管理所有的channel
    //GlobalEventExecutor.INSTANCE) 是全局的事件执行器，是一个单例
    private static ChannelGroup channelGroup = new
DefaultChannelGroup(GlobalEventExecutor.INSTANCE);
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");


    //handlerAdded 表示连接建立，一旦连接，第一个被执行
    //将当前channel 加入到  channelGroup
    @Override
```

```
    public void handlerAdded(ChannelHandlerContext ctx) throws Exception {
        Channel channel = ctx.channel();
        //将该客户加入聊天的信息推送给其它在线的客户端
        /*
        该方法会将 channelGroup 中所有的channel 遍历，并发送 消息，
        我们不需要自己遍历
         */
        channelGroup.writeAndFlush("[客户端]" + channel.remoteAddress() + " 加入
聊天" + sdf.format(new java.util.Date()) + " \n");
        channelGroup.add(channel);


    }

    //断开连接，将xx客户离开信息推送给当前在线的客户
    @Override
    public void handlerRemoved(ChannelHandlerContext ctx) throws Exception {

        Channel channel = ctx.channel();
        channelGroup.writeAndFlush("[客户端]" + channel.remoteAddress() + " 离开
了\n");
        System.out.println("channelGroup size" + channelGroup.size());

    }

    //表示channel 处于活动状态，提示 xx上线
    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {

        System.out.println(ctx.channel().remoteAddress() + " 上线了~");
    }

    //表示channel 处于不活动状态，提示 xx离线了
    @Override
    public void channelInactive(ChannelHandlerContext ctx) throws Exception {

        System.out.println(ctx.channel().remoteAddress() + " 离线了~");
    }

    //读取数据
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, String msg) throws
Exception {

        //获取到当前channel
        Channel channel = ctx.channel();
        //这时我们遍历channelGroup，根据不同的情况，回送不同的消息

        channelGroup.forEach(ch -> {
```

```
            if (channel != ch) { //不是当前的channel,转发消息
                ch.writeAndFlush("[客户]" + channel.remoteAddress() + " 发送了消息" + msg + "\n");
            } else {//回显自己发送的消息给自己
                ch.writeAndFlush("[自己]发送了消息" + msg + "\n");
            }
        });
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {
        //关闭通道
        ctx.close();
    }
}
```

客户端

```
import io.netty.bootstrap.Bootstrap;
import io.netty.channel.*;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.string.StringDecoder;
import io.netty.handler.codec.string.StringEncoder;

import java.util.Scanner;

public class GroupChatClient {
    //属性
    private final String host;
    private final int port;

    public GroupChatClient(String host, int port) {
        this.host = host;
        this.port = port;
    }

    public void run() throws Exception {
        EventLoopGroup group = new NioEventLoopGroup();

        try {

            Bootstrap bootstrap = new Bootstrap()
                    .group(group)
                    .channel(NioSocketChannel.class)
```

```
                    .handler(new ChannelInitializer<SocketChannel>() {

                        @Override
                        protected void initChannel(SocketChannel ch) throws
Exception {

                            //得到pipeline
                            ChannelPipeline pipeline = ch.pipeline();
                            //加入相关handler
                            pipeline.addLast("decoder", new StringDecoder());
                            pipeline.addLast("encoder", new StringEncoder());
                            //加入自定义的handler
                            pipeline.addLast(new GroupChatClientHandler());
                        }
                    });

            ChannelFuture channelFuture = bootstrap.connect(host,
port).sync();
            //得到channel
            Channel channel = channelFuture.channel();
            System.out.println("-------" + channel.localAddress() + "--------
");
            //客户端需要输入信息，创建一个扫描器
            Scanner scanner = new Scanner(System.in);
            while (scanner.hasNextLine()) {
                String msg = scanner.nextLine();
                //通过channel 发送到服务器端
                channel.writeAndFlush(msg + "\r\n");
            }
        } finally {
            group.shutdownGracefully();
        }
    }

    public static void main(String[] args) throws Exception {
        new GroupChatClient("127.0.0.1", 7000).run();
    }
}

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;

public class GroupChatClientHandler  extends
SimpleChannelInboundHandler<String> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, String msg) throws
Exception {
        System.out.println(msg.trim());
    }
```

```
}
```

1.12 心跳检测机制案例

实例要求：

1. 编写一个 Netty 心跳检测机制案例,当服务器超过 3 秒没有读时，就提示读空闲
2. 当服务器超过 `5` 秒没有写操作时，就提示写空闲
3. 实现当服务器超过 7 秒没有读或者写操作时，就提示读写空闲

```
import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelPipeline;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.handler.logging.LogLevel;
import io.netty.handler.logging.LoggingHandler;
import io.netty.handler.timeout.IdleStateHandler;

import java.util.concurrent.TimeUnit;

public class MyHeartBeatServer {
    public static void main(String[] args) throws Exception {

        //创建两个线程组
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workerGroup = new NioEventLoopGroup(); //16个
NioEventLoop
        try {

            ServerBootstrap serverBootstrap = new ServerBootstrap();

            serverBootstrap.group(bossGroup, workerGroup);
            serverBootstrap.channel(NioServerSocketChannel.class);
            serverBootstrap.handler(new LoggingHandler(LogLevel.INFO));
            serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>
() {

                @Override
```

```
                protected void initChannel(SocketChannel ch) throws Exception
{
                    ChannelPipeline pipeline = ch.pipeline();
                    //加入一个netty 提供 IdleStateHandler
                    /*
                    说明
                    1. IdleStateHandler 是netty 提供的处理空闲状态的处理器
                    2. long readerIdleTime : 表示多长时间没有读, 就会发送一个心跳检测
包检测是否连接
                    3. long writerIdleTime : 表示多长时间没有写, 就会发送一个心跳检测
包检测是否连接
                    4. long allIdleTime : 表示多长时间没有读写, 就会发送一个心跳检测
包检测是否连接

                    5. 文档说明
                    triggers an {@link IdleStateEvent} when a {@link Channel}
has not performed
 * read, write, or both operation for a while.
 *                  6. 当 IdleStateEvent 触发后 , 就会传递给管道 的下一个handler去处
理
 *                  通过调用(触发)下一个handler 的 userEventTiggered , 在该方法中去
处理 IdleStateEvent(读空闲, 写空闲, 读写空闲)
                     */
                    pipeline.addLast(new IdleStateHandler(7000, 7000, 10,
TimeUnit.SECONDS));
                    //加入一个对空闲检测进一步处理的handler(自定义)
                    pipeline.addLast(new MyServerHandler());
                }
            });

            //启动服务器
            ChannelFuture channelFuture = serverBootstrap.bind(7000).sync();
            channelFuture.channel().closeFuture().sync();

        } finally {
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }
}


import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import io.netty.handler.timeout.IdleStateEvent;

public class MyServerHandler extends ChannelInboundHandlerAdapter {
    @Override
```

```
    public void userEventTriggered(ChannelHandlerContext ctx, Object evt)
throws Exception {

        if (evt instanceof IdleStateEvent) {

            //将  evt 向下转型 IdleStateEvent
            IdleStateEvent event = (IdleStateEvent) evt;
            String eventType = null;
            switch (event.state()) {
                case READER_IDLE:
                    eventType = "读空闲";
                    break;
                case WRITER_IDLE:
                    eventType = "写空闲";
                    break;
                case ALL_IDLE:
                    eventType = "读写空闲";
                    break;
            }
            System.out.println(ctx.channel().remoteAddress() + "--超时类型--" +
eventType);
            System.out.println("服务器做相应处理..");

            //如果发生空闲，我们关闭通道
            // ctx.channel().close();
        }
    }
}
```

1.13 Netty 通过 WebSocket 编程实现服务器和客户端长连接

实例要求：

1. Http 协议是无状态的，浏览器和服务器间的请求响应一次，下一次会重新创建连接。
2. 要求：实现基于 `WebSocket` 的长连接的全双工的交互
3. 改变 Http 协议多次请求的约束，实现长连接了，服务器可以发送消息给浏览器
4. 客户端浏览器和服务器端会相互感知，比如服务器关闭了，浏览器会感知，同样浏览器关闭了，服务器会感知

```
import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelPipeline;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.handler.codec.http.HttpObjectAggregator;
import io.netty.handler.codec.http.HttpServerCodec;
```

```
import io.netty.handler.codec.http.websocketx.WebSocketServerProtocolHandler;
import io.netty.handler.logging.LogLevel;
import io.netty.handler.logging.LoggingHandler;
import io.netty.handler.stream.ChunkedWriteHandler;

public class MyWebSocketServer {
    public static void main(String[] args) throws Exception {

        //创建两个线程组
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workerGroup = new NioEventLoopGroup(); //16个
NioEventLoop
        try {

            ServerBootstrap serverBootstrap = new ServerBootstrap();

            serverBootstrap.group(bossGroup, workerGroup);
            serverBootstrap.channel(NioServerSocketChannel.class);
            serverBootstrap.handler(new LoggingHandler(LogLevel.INFO));
            serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>
() {

                @Override
                protected void initChannel(SocketChannel ch) throws Exception
{
                    ChannelPipeline pipeline = ch.pipeline();

                    //因为基于http协议，使用http的编码和解码器
                    pipeline.addLast(new HttpServerCodec());
                    //是以块方式写，添加ChunkedWriteHandler处理器
                    pipeline.addLast(new ChunkedWriteHandler());

                    /*
                    说明
                    1. http数据在传输过程中是分段, HttpObjectAggregator ，就是可以将
多个段聚合

                    2. 这就就是为什么，当浏览器发送大量数据时，就会发出多次http请求
                     */
                    pipeline.addLast(new HttpObjectAggregator(8192));
                    /*
                    说明
                    1. 对应websocket ，它的数据是以 帧(frame) 形式传递
                    2. 可以看到WebSocketFrame 下面有六个子类
                    3. 浏览器请求时 ws://localhost:7000/hello 表示请求的uri
                    4. WebSocketServerProtocolHandler 核心功能是将 http协议升级为
ws协议 ，保持长连接

                    5. 是通过一个 状态码 101
                     */
```

```
                    pipeline.addLast(new
WebSocketServerProtocolHandler("/hello2"));

                    //自定义的handler ， 处理业务逻辑
                    pipeline.addLast(new MyTextWebSocketFrameHandler());
                }
            });

            //启动服务器
            ChannelFuture channelFuture = serverBootstrap.bind(7000).sync();
            channelFuture.channel().closeFuture().sync();

        } finally {
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }
}
```

```
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.handler.codec.http.websocketx.TextWebSocketFrame;

import java.time.LocalDateTime;

public class MyTextWebSocketFrameHandler extends
SimpleChannelInboundHandler<TextWebSocketFrame> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, TextWebSocketFrame
msg) throws Exception {

        System.out.println("服务器收到消息 " + msg.text());

        //回复消息
        ctx.channel().writeAndFlush(new TextWebSocketFrame("服务器时间" +
LocalDateTime.now() + " " + msg.text()));
    }

    //当web客户端连接后， 触发方法
    @Override
    public void handlerAdded(ChannelHandlerContext ctx) throws Exception {
        //id 表示唯一的值，LongText 是唯一的 ShortText 不是唯一
        System.out.println("handlerAdded 被调用" +
ctx.channel().id().asLongText());
```

```
        System.out.println("handlerAdded 被调用" +
ctx.channel().id().asShortText());
    }


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

        System.out.println("handlerRemoved 被调用" +
ctx.channel().id().asLongText());
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
        System.out.println("异常发生 " + cause.getMessage());
        ctx.close(); //关闭连接
    }
}
```

hello.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<script>
    var socket;
    //判断当前浏览器是否支持websocket
    if(window.WebSocket) {
        //go on
        socket = new WebSocket("ws://localhost:7000/hello2");
        //相当于channelReado, ev 收到服务器端回送的消息
        socket.onmessage = function (ev) {
            var rt = document.getElementById("responseText");
            rt.value = rt.value + "\n" + ev.data;
        }

        //相当于连接开启(感知到连接开启)
        socket.onopen = function (ev) {
            var rt = document.getElementById("responseText");
            rt.value = "连接开启了.."
        }
```

```
        //相当于连接关闭(感知到连接关闭)
        socket.onclose = function (ev) {

            var rt = document.getElementById("responseText");
            rt.value = rt.value + "\n" + "连接关闭了.."
        }
    } else {
        alert("当前浏览器不支持websocket")
    }

    //发送消息到服务器
    function send(message) {
        if(!window.socket) { //先判断socket是否创建好
            return;
        }
        if(socket.readyState == WebSocket.OPEN) {
            //通过socket 发送消息
            socket.send(message)
        } else {
            alert("连接没有开启");
        }
    }
</script>
    <form onsubmit="return false">
        <textarea name="message" style="height: 300px; width: 300px">
</textarea>
        <input type="button" value="发生消息"
onclick="send(this.form.message.value)">
        <textarea id="responseText" style="height: 300px; width: 300px">
</textarea>
        <input type="button" value="清空内容"
onclick="document.getElementById('responseText').value=''">
    </form>
</body>
</html>
```

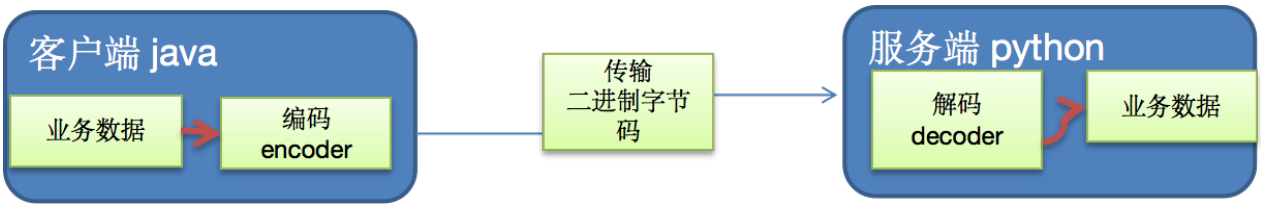
2 Google Protobuf

2.1 编码和解码的基本介绍

编写网络应用程序时，因为数据在网络中传输的都是二进制字节码数据，在发送数据时就需要编码，接收数据时就需要解码[示意图]

codec（编解码器）的组成部分有两个：decoder（解码器）和 encoder（编码器）；

encoder 负责把业务数据转换成字节码数据，decoder 负责把字节码数据转换成业务数据



2.2 Netty 本身的编码解码的机制和问题分析

1. Netty 自身提供了一些 codec(编解码器)
2. Netty 提供的编码器 StringEncoder，对字符串数据进行编码 ObjectEncoder，对java对象进行编码。
3. Netty 提供的解码器 StringDecoder，对字符串数据进行解码 ObjectDecoder，对 java 对象进行解码。
4. Netty 本身自带的 ObjectDecoder 和 ObjectEncoder 可以用来实现 POJO 对象或各种业务对象的编码和解码，底层使用的仍是Java序列化技术,而Java序列化技术本身效率就不高，存在如下问题
    - 无法跨语言
    - 序列化后的体积太大，是二进制编码的5倍多。
    - 序列化性能太低

引出新的解决方案[Google 的 Protobuf]

2.3 Protobuf

Protobuf是由谷歌开源而来，在谷歌内部久经考验。它将数据结构以.proto文件进行描述，通过代码生成工具可以生成对应数据结构的POJO对象和Protobuf相关的方法和属性。

特点如下:

- 结构化数据存储格式(XML,JSON等)
- 高效的编解码性能
- 语言无关、平台无关、扩展性好

数据交互xml、json、protobuf格式比较

1. json: 一般的web项目中，最流行的主要还是json。因为浏览器对于json数据支持非常好，有很多内建的函数支持。
2. xml: 在webservice中应用最为广泛，但是相比于json，它的数据更加冗余，因为需要成对的闭合标签。json使用了键值对的方式，不仅压缩了一定的数据空间，同时也具有可读性。
3. protobuf:是后起之秀，是谷歌开源的一种数据格式，适合高性能，对响应速度有要求的数据传输场景。因为profobuf是二进制数据格式，需要编码和解码。数据本身不具有可读性。因此只能反序列化之后得到真正可读的数据。

相对于其它protobuf更具有优势

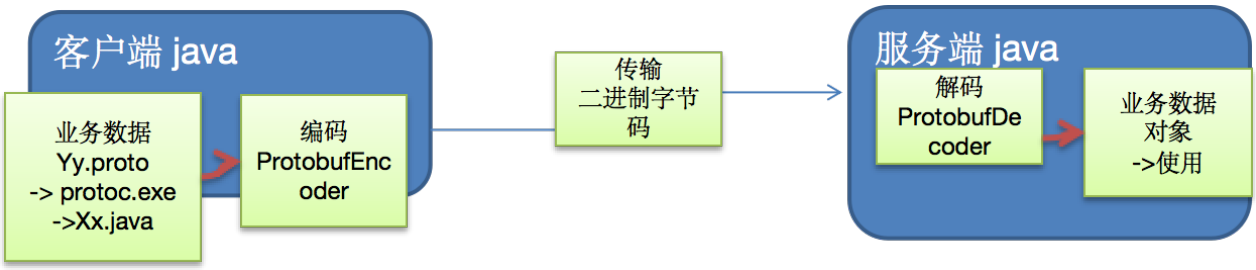
1. 序列化后体积相比Json和XML很小，适合网络传输
2. 支持跨平台多语言
3. 消息格式升级和兼容性还不错
4. 序列化反序列化速度很快，快于Json的处理速速

结论： 在一个需要大量的数据传输的场景中，如果数据量很大，那么选择protobuf可以明显的减少数据量，减少网络IO，从而减少网络传输所消耗的时间。

因而，对于打造一款高性能的通讯服务器来说，protobuf 传输格式，是最佳的解决方案。

参考文档：https://developers.google.com/protocol-buffers/docs/proto 语言指南

protobuf 使用示意图



2.4 示例操作步骤

1、idea安装插件Protobuf Support

2、安装protobuf编译器

https://github.com/protocolbuffers/protobuf/releases

3、添加依赖

```
<!-- https://mvnrepository.com/artifact/com.google.protobuf/protobuf-java -->
    <dependency>
      <groupId>com.google.protobuf</groupId>
      <artifactId>protobuf-java</artifactId>
      <version>3.14.0</version>
    </dependency>
```

4、编写文件Student.proto

```
syntax = "proto3";
option optimize_for = SPEED; // 加快解析
option java_package="com.kkb.demo.netty.example.codec";    //指定生成到哪个包下
option java_outer_classname="MyDataInfo"; // 外部类名, 文件名

//protobuf 可以使用message 管理其他的message
message MyMessage {

    //定义一个枚举类型
    enum DataType {
        StudentType = 0; //在proto3 要求enum的编号从0开始
        WorkerType = 1;
    }

    //用data_type 来标识传的是哪一个枚举类型
```

```
    DataType data_type = 1;

    //表示每次枚举类型最多只能出现其中的一个, 节省空间
    oneof dataBody {
        Student student = 2;
        Worker worker = 3;
    }

}


message Student {
    int32 id = 1;//Student类的属性
    string name = 2; //
}
message Worker {
    string name=1;
    int32 age=2;
}
```

5、编写 .proto 文件，生成java文件

```
~/protobuf/bin/protoc --java_out=.
com/kkb/demo/netty/example/codec/Student.proto
```

6、Netty客户端 Handler发送

```
public class NettyClientHandler extends ChannelInboundHandlerAdapter {
    //当通道就绪就会触发该方法
    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {

//        MyDataInfo.MyMessage message =
//                MyDataInfo.MyMessage.newBuilder().setDataType(
MyDataInfo.MyMessage.DataType.StudentType)
//
.setStudent(MyDataInfo.Student.newBuilder().setId(6).setName("北
京").build()).build();
//        ctx.writeAndFlush(message);
        MyDataInfo.MyMessage message =
                MyDataInfo.MyMessage.newBuilder().setDataType(
MyDataInfo.MyMessage.DataType.WorkerType)

.setWorker(MyDataInfo.Worker.newBuilder().setAge(20).setName("杭
州").build()).build();
        ctx.writeAndFlush(message);
    }
```

```
  .....
}
```

7、Netty客户端添加编码器

```
 //设置相关参数
            bootstrap.group(group) //设置线程组
                    .channel(NioSocketChannel.class) // 设置客户端通道的实现类(反射)
                    .handler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void initChannel(SocketChannel ch) throws Exception {
                            ChannelPipeline channelPipeline = ch.pipeline();
                            channelPipeline.addLast("encoder",new ProtobufEncoder());
                            channelPipeline.addLast(new NettyClientHandler()); //加入自己的处理器
                        }
                    });
```

8、Netty服务端添加解码器

```
.childHandler(new ChannelInitializer<SocketChannel>() {//创建一个通道初始化对象(匿名对象)
                        //给pipeline 设置处理器
                        @Override
                        protected void initChannel(SocketChannel ch) throws Exception {
                            System.out.println("客户socketchannel hashcode=" + ch.hashCode()); //可以使用一个集合管理 SocketChannel， 再推送消息时，可以将业务加入到各个channel 对应的 NIOEventLoop 的 taskQueue 或者 scheduleTaskQueue
                            ChannelPipeline pipeline = ch.pipeline();
                            pipeline.addLast("decoder",new ProtobufDecoder(MyDataInfo.MyMessage.getDefaultInstance()));
                            pipeline.addLast(new NettyServerHandler());
                        }
                    });
```

9、Netty服务端Handler接收消息

```
public class NettyServerHandler extends SimpleChannelInboundHandler<MyDataInfo.MyMessage> {


    //读取数据实际(这里我们可以读取客户端发送的消息)
```

```
    @Override
    public void channelRead0(ChannelHandlerContext ctx, MyDataInfo.MyMessage
msg) throws Exception {

        MyDataInfo.MyMessage.DataType dataType = msg.getDataType();
        if(dataType==MyDataInfo.MyMessage.DataType.StudentType){
            MyDataInfo.Student student = msg.getStudent();
            System.out.println("student
id="+student.getId()+",name="+student.getName());

        }else if(dataType==MyDataInfo.MyMessage.DataType.WorkerType){
            MyDataInfo.Worker worker = msg.getWorker();
            System.out.println("woker
age="+worker.getAge()+",name="+worker.getName());
        }else {
            System.out.println("类型不存在");
        }
    }

  ...
}
```