**Reactor 组件详解：5种角色**

1. **Handle （句柄或描述符）：本质上表示一种资源，是由操作系统提供的，该资源用于表示一个个事件，比如文件描述符，或是针对网络编程中的socket描述符，事件既可以来源于外部，也可以来源于内部，外部时间比如说客户端的连接请求，客户端发送过来的数据等，内部事件比如说操作系统产生的定时事件。他本质就是一个文件的描述符。Handle是时间产生的发源地。**
2. **Synchronous event demultiplexer （同步事件分离器）：它本身是一个系统调用，用于等待事件的发生（事件可能是一个也可能是多个），调用放调用它的时候会被阻塞，一直阻塞到同步事件分离器上有事件发生为止，对于linux来说，同步事件分离器指的就是常用的I/O多路复用机制，比如说select poll epoll 等。在java nio 中，同步分离器对应的组件就是selector ，对应的阻塞方法就是select。**
3. **Event handler （事件处理器）：本身由多个回调方法组成，这些方法的调用构成了与应用相关的对于某个事件的反馈机制。Netty相比于java nio 来说，在事件处理器这个角色上进行了一个升级，他为我们提供了大量的回调方法，供我们在特定事件产生的时实现相应的回调方法进行业务逻辑的处理。**
4. **Concrete event handler（具体事件处理器）：是事件处理器的实现，它本身实现了事件处理器所提供的各个回调方法，从而实现了特定于业务逻辑，他本质就是我们所编写的一个个处理器的实现。**
5. **Initiation dispatcher （初始分发器）：实际上就是reactor角色，它本身定义一些规范，这些规范用于控制事件的注册以及调度方式，，同时提供了应用进行事件处理器的注册，删除等配置，它本身就是整个是事件处理器的核心；初始分发器通过同步时间分离器来等待事件，一旦发生事件，初始分发器就会分离出每一个事件，然后调用事件处理器，最后调用相关的回调方法来处理这些事件。**

**Reactor 组件的流程**

1. **当应用向Initiation dispatcher注册具体的时间处理器时，应用会标识出该事件处理器希望Initiation dispatcher 在某个时刻事件发生时向其通知该事件（类似于观察模式），该事件与handle关联**
2. **Initiation dispatcher会要求每个事件处理器都向其传递内部的handle，该handle向操作系统标识了事件处理器**
3. **当所有事件处理器都注册完毕后，应用回到用handle\_events方法来启动Initiation dispatcher的事件循环，这时，Initiation dispatcher会将每个注册的事件处理器的handle合并起来，并使用同步事件分离器等待这些事件发生，比如说，tcp协议会使用select同步事件分离器操作来等待客户端发送的数据到达连接的socket handle 上。**
4. **当某个事件源对应的handle变为ready状态时，（比如说，tcp socket 变为等待读状态时），同步事件分离器就会通知Initiation dispatcher**
5. **Initiation dispatcher 会触发事件处理器的回调方法，从而相应这个处于ready状态的handle，当事件发生时，Initiation dispatcher会将被事件源激活的handle的类型（type）作为key 来寻找并分发给恰当的事件处理器回调方法。**
6. **Initiation dispatcher会回调事件处理的handle\_event回调方法来执行特定的应用功能（开发者所编写的功能），从而响应这个事件，所发生的事件类型可以作为该方法参数并被该方法内部使用来执行额外的特定于服务的分离与分发。**

**EventLoopGroup 与 channel 之间的注册关系。**

**config（）——》ServerBootstrapConfig (对ServerBootstrap一次封装)。**

**group（） ———》拿到的就是EventLoopGroup**

**register() -->channel 与 eventLoopGroup 中的一个 Eventexecutor进行执行所绑定，**

**最终的绑定就是把当前的channel 通过自身的register 方法与singeThreadExecutor(Eventexecutor的抽象类)里的定义的selector进行注册，返回selectionKey。**

1. **一个eventloopgroup当中会包含一个或多个EventLoop**
2. **一个EventLoop 在它的生命周期中只会与一个thread进行绑定**
3. **所有的eventLoop所处理的I/O事件都会在与之关联的Thread上执行**
4. **一个channel在它的声明周期中只会注册一个eventloop**
5. **一个eventloop 会在运行期间，会被分配多个channel**

**重要的结论：在netty中，channel的实现一定是线程安全的；基于此，我们可以存储一个channel的引用或者channelFuture的引用（只能读与等待），并且在需要向远程端点发送数据时，通过引用这个channel相应的方法； 即便当时有很多线程都在使用该channel也不会出现多线程问题。**

**重要结论：我们在业务开发中，不要长时间执行的耗时业务逻辑放入到eventloop的执行队列中，因为它会一直阻塞该线程所对应的所有channel上的其它的执行任务，如果我们需要进行阻塞调用或者是耗时的操作，那么我们需要另起一个线程池去执行（业务线程池）**

**有两种实现方式：**

1. **在channelhandler的回调方法中使用自己定义的业务线程池实现异步**
2. **借助于netty提供的向channelpipeline添加handler时，调用addlast方法来传递eventExecutor**

**默认情况下，调用addlast方法添加handler时，channelhandler中的回调方法都是由I/O线程所执行，如果调用了channelpipeline 中的含有eventGroup 参数的方法，那么channelHandler中的回调方法将由参数中group线程组去执行。**

**JDK所提供的future只能通过手工方式检查执行结果，而这个操作是会阻塞的，netty则对channelfuture进行了增强，通过channelfuturelistener以回调的方式来获取执行结果，去除手工检查阻塞的方式，类似于consumer；值得注意的事：channelFuturelistener的operationCompelete方法是由I/O线程执行的，因此要注意不要在这里执行耗时操作，否则需要使用额外的线程池；请多留意channelFuture 类的注释文档，关于如何判定连接超时的最佳写法，值得参考。**

**重点：simplechannelinboundhandler 这个方法中，调用read0方法后会自动release客户端传入过来值的引用，因此你在read0里写异步方法使用到该值有可能会出现问题be careful。**

**在Netty 中有两种发送消息的方式，可以直接写到channel中，也可以写到与channelhandler 所关联的那个channelHandlerContext中，对于前一种方式来讲，消息会从channelpipeline的末尾开始流动，对于后一种方式来讲，消息将会从该handler的下一个handler开始流动。**

**结论：**

1. **Channelhandlercontext 与channelhandler之间的关联绑定关系是永远不会发生改变，因此对其进行缓存是没有任何问题的。**
2. **对于channel的同名方法来讲，channelhandlerContext的方法会产生更短的事件流，提升应用的性能。**
3. **开发经验：当一个服务器既当服务器又当客户端，起一个代理的作用，这样通过channelhandlerContext ctx 获取eventloop 去设置当前服务器与另一个服务器之间绑定同一个evenloop，使用同一个I/O 线程。bootstrap.group(ctx.channel().eventLoop())去绑定相同的eventLoop进行I/O 操作。**

**数据总是以字节的形式传输的，对于channel来说总是通过buffer来进行读写的，跟stream来对比，它是双向的；既能读也能写；但是stream只能是inputstream 或者是outputstream不可能同时是两个；因此channel更能反应底层操作系统的原理。**

**NIO 进行文件读取所涉及的操作：**

1. **从fileinputstream 对象获取到channel对象。**
2. **创建buffer**
3. **将数据从channel中读取buffer对象中**

**Mark <= position <= limit <= capacity**

1. **flip()方法**
2. **将limit设置为当前的position**
3. **将position设置为0**
4. **将mark废弃-1**
5. **clear()方法**
6. **将limit值设置为capacity**
7. **将position设置为0**
8. **compact()方法**
9. **将所有未读的数据复制到buffer起始位置处**
10. **将position设为最后一个未读元素的后面**
11. **将limit设置为capacity**
12. **现在buffer准备好，但是不会覆盖原来未读的数据**
13. **其中directByteBuffer 自身是一个java对象，在java堆中，而这个对象中有个long类型的字段address，记录着一块调用malloc()申请到的native memory。**
14. **Directbytebuffer自身是java堆内的，他背后真正承载的数据buffer是在堆外，但是还是所属于用户态。**
15. **Hotspot虚拟机里的GC除了CMS之外都是要移动对象的，是所谓的“copacting GC”**
16. **如果要把一个java里的byte[]对象的引用传递给native代码，让native代码直接访问数据的内容的话，这需要保证native方法在访问的时候这个byte[]对象不能被移动，也就是要被“pin钉住”。**
17. **所以虚拟机做了一个中转，它假设把heapbytebuffer 背后的byte数组里的内容拷贝一次是一个时间开销是阔以接受的，听听故事假设真正的I/O操作可能是一个很慢的操作，于是它就把heap中的byte数据拷贝至directbytebuffer背后的native memory，这个拷贝会涉及sum.misc.Unsafe.copyMemory()的调用，背后类似于memcpy()的实现，这个操作本质上是会在整个拷贝过程中不会发生GC （因为中间没有safepoint所以GC无法发生）， 然后数据到native memory中后，就真正去做I/O把directbytebuffer背后的native memory地址传递给真正做I/O的函数，这边就不需要再去访问java对象去读写要的I/O的数据了。**
18. **Heap buffer**
19. **Direct buffer**
20. **Composite buffer**

**Heap buffer（堆缓冲区）**

**最常用的类型，bytebuffer将数据直接存储到jvm的堆空间，并且实际数据放到byte arrary中来实现**

**优点：由于数据是存储在jvm 中，因此可以快速创建与快速释放，并且它提供了直接访问内部字节数组的方法。hasArrary() ;arrary()**

**缺点：每次写数据时，都需要先将数据复制到直接缓冲区中在进行网络传输**

**Direct buffer（直接缓冲区）**

**在之外直接分配内存，直接缓冲区并不会占用堆的容量空间，因为它是由操作系统在本地内存进行的数据分配。**

**优点：在使用socket编程进行数据传递时，性能非常好，因为数据直接位于操作系统的本地内存中，所以不需要从jvm将数据复制到直接缓冲区中，性能很好。**

**缺点：因为direct buffer 是直接在操作系统内存中，所以内存空间的分配与释放要比堆空间复杂，而且速度要慢一些**

**Netty通过提供内存池来解决这个问题，直接缓冲区并不支持通过字节数组的方式来访问数据。**

**重点： 对于后端的业务消息编解码来说，使用heapbytebuffer；对于I/O通信线程在读写缓冲区时，使用directbytebuffer**

**JDK的bytebuffer与netty的bytebuffer之间的差异对比:**

1. **Netty 的bytebuffer采用了读写索引分离的策略（readindex 与 writeindex），一个初始化（里面尚未有数据）的bytebuf 的 读写索引都指向0**
2. **当读写索引处于同一个位置的时候，如果我们继续读取，那么就会抛出indexoutofboundsexception**
3. **对于bytebuf的任何读写操作都会分别单独维护读索引与写索引，maxcapacity 最大的容量默认为integer.max\_value；因此它阔以自动扩容。**

**Netty的bytebuffer的优点：**

1. **存储字节的数据是动态的，其最大值是integer.max\_value 这里的动态性体现在write方法中，write方法在执行时会判断buffer容量，如果不足则会自动的扩充**
2. **Bytebuffer的读写索引是完全分开的，用起来比较方便**

**AtomicIntegerFieldUpdater要点总结：**

1. **更新器跟新的必须是int类型的变量，不能是包装类型；**
2. **更新器跟新的必须是volatile类型的变量，确保线程之间共享变量时的立即可见性**
3. **变量不能是static的，必须是实例的变量，因为Unsafe.objectFieldOffset()方法不支持静态变量，（cas 算法本质上是通过对象实例的偏移量来直接进行赋值）**
4. **更新器只能修改它可见范围内的变量，因为跟新器是通过反射的方式来得到这个变量的，如果变量不可见就会报错**
5. **Netty中为什么会全局是用这个原子类实例对象去操作每个实例的cnf属性呢？而采用定义一个AtomicInteger的静态变量去增加减；因为atomicinteger也是同样对int的封装；而且每一实例都会有这样一个变量的引用，会增加性能的消耗，因此采用全局的跟新器，去跟新一个int的变量，会减少内存消耗同时，性能会更好。**
6. **当一个客户端连接上服务器，继承自ChannelInitializer<SocketChannel>这个类并实现intialChannel(SocketChannel st) 方法的实例，在bootstrap创建的时候会加入childHandler中，因此它也同时是一个handler 只不过执行完该方法就会被移除，因此在客户端建立连接形成新的socketchannel ,会产生与之对应的新的一个pipeline；同理调用该channelinitializer的同一个实例调用initialChannel方法去添加定义好的handler**
7. **总结： 每个socketchannel新建立连接时，会调用一次initialchannel方法创建新的执行器链，这个链包含两种类型的处理器，inboundhandler 与outboundhandler；因此handler的实例是不共享的。同理同一个handler实例可以对应多个channelhandlercontext。**