**NettyReactor模式**[](" \l "nettyreactor" \o "Permanent link)

* 57、Reactor模式透彻理解及其在Netty中的应用
* 58、Reactor模式与Netty之间的关系详解
* 59、Acceptor与Dispatcher角色分析
* 61、Reactor模式5大角色分析
* 62、Reactor模式组件调用关系全景分析
* 63、Reactor模式与Netty组件对比及Acceptor组件的作用

**Reactor模式透彻理解及其在Netty中的应用**[](" \l "reactornetty" \o "Permanent link)

Netty整体架构是Reactor模式的完整体现。

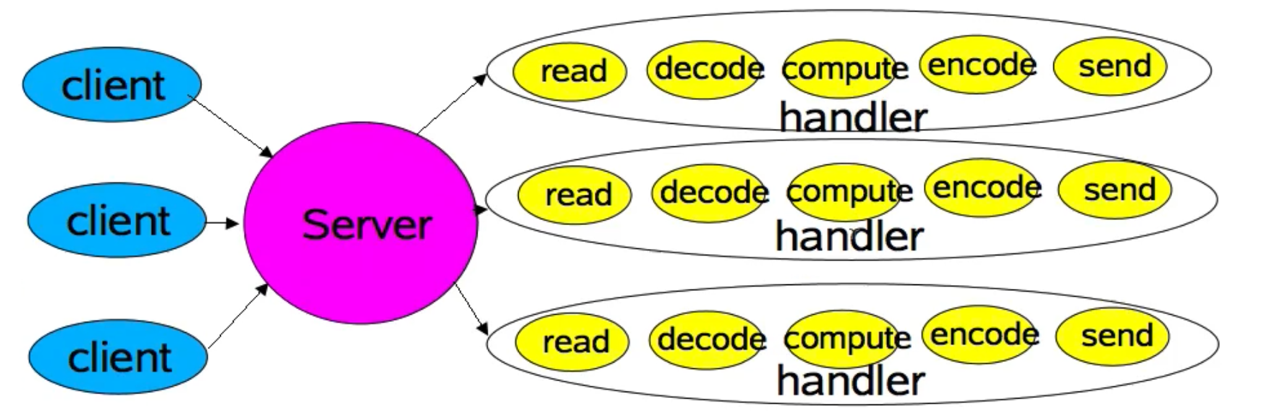
《Scaleble IO in Java 》— Doug Lea,从阻塞式IO开始讲起，一直讲到Netty Reactor模式

* Scalable network Services
* Event-driven processing
* Reactor pattern
  + Basic version
  + Multithreaded versions
  + Other variants
* Walthrough of java.nio.nonblocking IO APIs

**Network Services**[](#network-services)

* Web services, Distributed Objects,etc
* Most have same basic structure:
  + Read Request
  + Decode request
  + Process Service
  + Encode reply
  + Send reply
* But differ in nature and cost of each Step
  + XML parsing,File transfer, Web page generation, computatingal services。....

**Classic Service Designs**[](#classic-service-designs)

* Each handler may be started in its own thread
* 图中每个handler都需要启动一个线程来处理，也就是serverSocket 每连接一个socket都会启动一个线程. 

**Classic ServerSocket Loop**[](#classic-serversocket-loop)

**class** **Server** **implements** Runnable**{**

**public** **void** **run(){**

**try{**

ServerSocket ss **=** **new** ServerSocket**(**PORT**);**

**while(!**Thread**.**interrupted**()){**

**new** Thread**(new** Handler**(**ss**.**acept**())).**start**();**

*//or,single-threaded,or a thread pool*

**}**

**}catch(**IOException ex**){***/\*\*/***}**

**}**

**}**

**static** **class** **Handler** **implements** Runnable**{**

**final** Socket socket**;**

Handler**(**Socket s**){**

Socket **=** s**;**

**}**

**public** **void** **run(){**

**try{**

**byte[]** input **=** **new** **byte[**MAX\_INPUT**];**

socket**.**getInputStream**.**read**(**input**);**

**byte[]** ouput **=** process**(**input**);**

socket**.**getOutputStream**.**write**(**output**);**

**}catch(**IOException ex**){***/\*\*/***}**

**}**

**}**

**Scalability Goals 可伸缩性的目标**[](" \l "scalability-goals" \o "Permanent link)

* Graceful degradation under increasing load(more client) 当有更多的客户端的时候能够优雅的降低了负载（优雅的降级）
* Continuous inprovement with increasing resources(CPU ,memory,disk,bandwidth) d当资源在不断增加的时候能够持续改善性能
* Also meet avaliability and performance goals
  + Short latencies 更短的延迟
  + meeting peak demand 应对峰值
  + Tunable quality of service 高可用
* **Divide-and-conquer is usually the best approach for achieving any scalability goal， Divide-and-conquer就是分而治之**

**Divide and Conquer**[](#divide-and-conquer)

* Divide processing into small tasks
  + Each task performs an action without blocking
  + 在上面的那种模式中 read、decode、compute、encode、send都在一个handler线程中
  + read、decode、compute、encode、send整个都是按照顺序执行,然而这里面会涉及到很多IO操作，IO我们知道和CPU的处理速度式完全不匹配的，如果让CPU等待IO操作完成，是对性能的一种极大的损耗。
  + 所以这里说将处理过程分解为很多小的任务，每个任务都是非阻塞的
* Execute each task when it is enabled 当每个任务可用的时候去执行每个任务
  + Here,an IO event usually serves as trigger
  + 通常IO事件都是作为一个触发器的
  + 即当一个IO操作完成之后就会产生一个事件
* Basic mechanisms supported in java.io
  + Non-blocking reads and writes 非阻塞的读和写
  + Dispatch tasks associated with sensed IO events 分发能感知到的IO事件的任务

**Event-driven Designs**[](#event-driven-designs)

* Usually more effcient than alternative 通常比其他方式更高效
  + Fewer resources 更少资源消耗
    - dont usully need a thread pre client, 并不需要对每个客户端产生一个线程
  + Less overhead
    - less context switching ,often less locking
  + but dispatching can be slower 但是分发能变得更慢的
    - 必须手动的将action绑定到事件上面
* Usually harder to program
  + Must break up into simple non-blocking action 必须分解为一些简单的非阻塞的动作
  + must keep track of logic state of service 异步的必须track

**Reactor 模式**[](" \l "reactor" \o "Permanent link)

* Reactor : response to IO events by dispatching the appropriate handler, 通过分发合适的handler，针对IO事件的响应，即：怎么对IO事件进行响应的，通过分发合适的处理器。
* Handlers: perform non-blocking actions
* Manage by bind handlers to event 通过将handlers绑定到事件进行管理

**Basic Reactor Design**[](#basic-reactor-design)

**Single threaded version**[](#single-threaded-version)

图中Reactor其实是一个线程对象，它做了什么事情：

* 他去检测和监听客户端向服务器发起的连接
* 当一个连接一旦建立好之后，或者事件被传进来之后，reactor会通过一种 **派发的方式**，将客户端发过来的数据，发给特定的线程对象（特定的处理器）。由特定的处理器对客户端发过来的数据进行处理

**Java.nio.Support**[](" \l "javaniosupport" \o "Permanent link)

* Channels
  + Connections to files ,sockets etc that support non-blocking reads
* Buffers
  + Array-like objects that can be directly read or write by Channels
* Selectors
  + Tell which of a set of Channels have IO events
* SelectionKeys
  + Maintain IO event status and bindings,通过selectionKeys就能知道是那个Channel绑定了哪些事件

**Reactor 1 : Setup实现**[](" \l "reactor-1-setup" \o "Permanent link)

Reactor就是一个线程对象，Netty中NIOEventLoop就是一个Reactor

**class** **Reactor** **implements** Runnable**{**

**final** Selector selector**;**

**final** ServerSocketChannel serverSocketChannel**;**

Reactor**(int** port**){**

selector **=** Selector**.**open**();**

serverSocketChannel **=** ServerSocketChannel**.**open**();**

serverSocketChannel**.**socket**().**bind**(new** InetSocketAddress**(**port**));**

serverSocketChannel**.**configureBlocking**(false);**

SelectionKey sk **=** serverSocketChannel**.**register**(**selector**,**SelectionKey**.**OP\_ACCEPT**);**

sk**.**attach**(new** Acceptor**());**

**}**

**}**

**Reactor 2:Dispatch Loop**[](#reactor-2dispatch-loop)

*//class Reactor continued*

**public** **void** **run(){***//normally in a new thread*

**try{**

**while(!**Thread**.**interrrupted**()){**

selector**.**select**();***//这时一个阻塞的方法*

Set selected **=** selector**.**selectedKeys**();**

Iterator it **=** selected**.**iterator**();**

**while(**it**.**hasNext**()){**

dispatch**((**SelectionKey**)(**it**.**next**()));**

selected**.**clear**();**

**}**

**}**

**}catch(**IOException ex**){***/\*\*/***}**

**}**

**void** **dispatch(**SelectionKey k**){**

Runnable r **=** **(**Runnable**)(**k**.**attachment**());**

**if(**r**!=null){**

r**.**run**();**

**}**

**}**

**Reactor3: Acceptor**[](#reactor3-acceptor)

*//class Reactor continued*

**class** **Accptor** **implements** Runnable**{**

*//inner class*

**public** **void** **run(){**

**try{**

SocketChannel c **=** serverSocketChannel**.**accept**();**

**if(**c**!=null){**

**new** Handler**(**selector**,**c**);**

**}**

**}catch(**IOException ex**){***/\*\*/***}**

**}**

**}**

**Reactor4: Handler setup**[](#reactor4-handler-setup)

**final** **class** **Handler** **implements** Runnable**{**

**final** SocketChannel socketChannel**;**

**final** SelectionKey sk**;**

Handler**(**Selector sel**,** SocketChannel c**){**

scoket **=** c**;**

c**.**configureBlocking**(false);**

sk **=** socket**.**register**(**sel**,**0**);**

sk**.**attch**(this);**

sk**.**interestOps**(**SelectionKey**.**OP\_READ**);**

sel**.**wakeup**();**

**}**

**}**

**58 Reactor模式与Netty之间关系详解**[](" \l "58-reactornetty" \o "Permanent link)

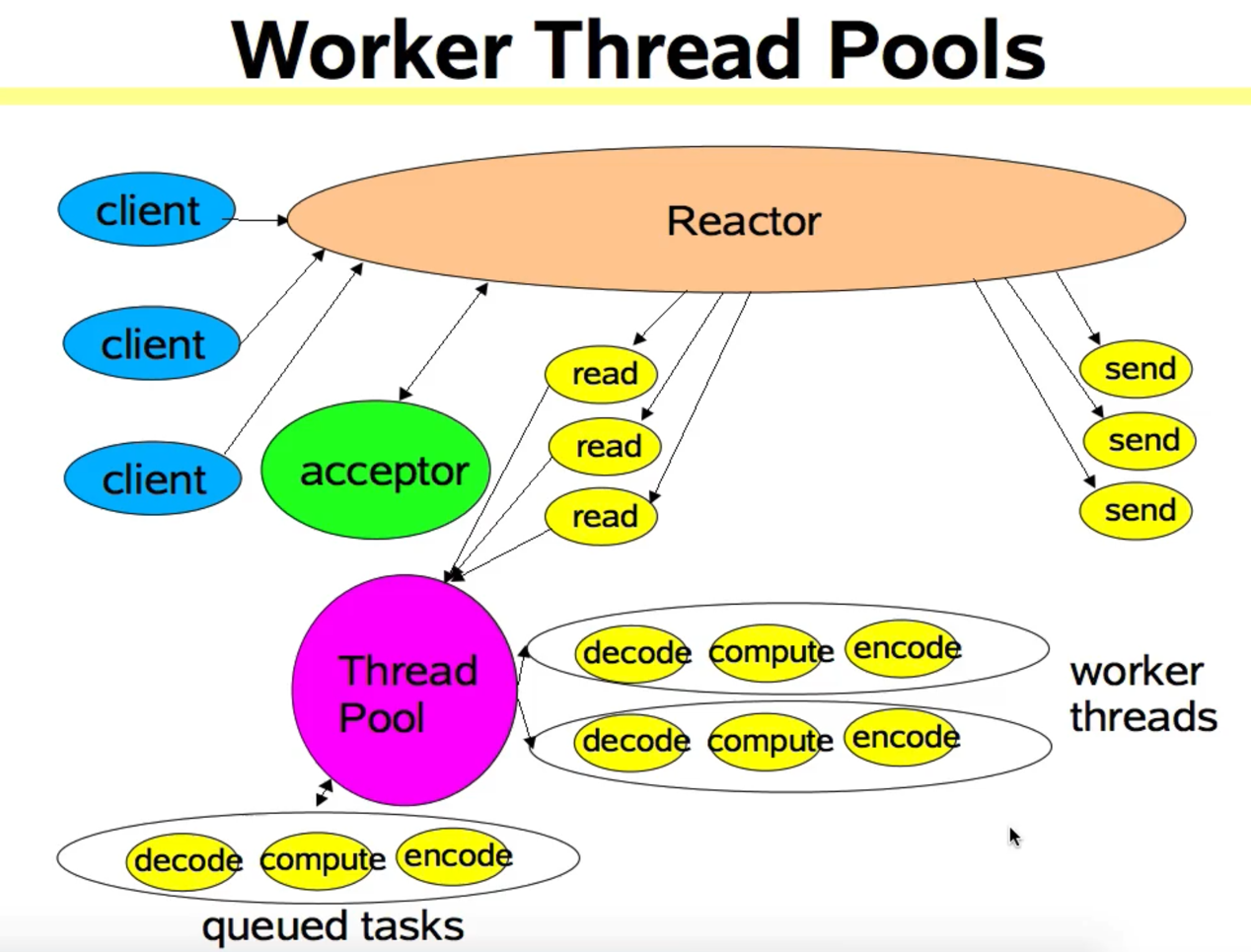
**Mutithreaded Designs**[](#mutithreaded-designs)

* Worker Threads
  + Reactor should quickly trigger handlers
    - because Handler processing slows down Reactor Thread
  + 建议将非IO处理的过程交给其他线程去处理 Offload non-IO processing to other threads
* Mutiple Reactor Threads
  + Reactor threads can saturate doing IO
  + Distribute load to other reactors
    - load-balance to match CPU and IO rates

**Worker Threads**[](#worker-threads)

* offload non-IO processing to speed up Reacotr thread, 将非IO处理移交出去提高Reactor的速度

**Worker Thread Pools**[](#worker-thread-pools)



**Handler with Thread Pool**[](#handler-with-thread-pool)

**class** **Handler** **implements** Runnable**{**

*//uses util.concurrent thread pool*

**static** PooledExecutor pool **=** **new** PooledExecutor**(...);**

**static** **final** **int** PROCESSING **=** 3**;**

**synchronized** **void** **read(){**

socke**.**read**(**input**);**

**if(**inputISComplete**()){**

state **=** POCESSING**;**

pool**.**execute**(new** Processer**());**

**}**

**}**

**class** **Processer** **implements** Runnable**{**

**public** **void** **run(){**

processAndHandOff**();**

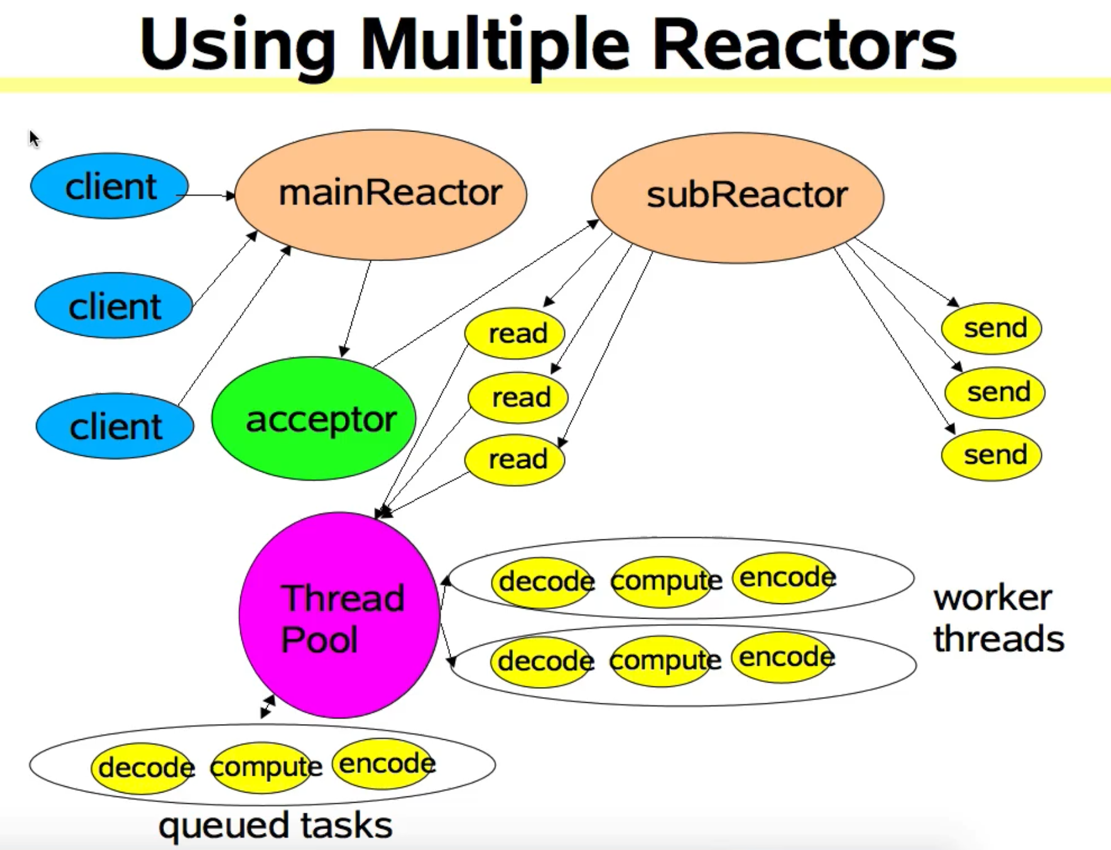
**}**

**}**

**}**

**Mutiple Reactor Threads**[](#mutiple-reactor-threads)

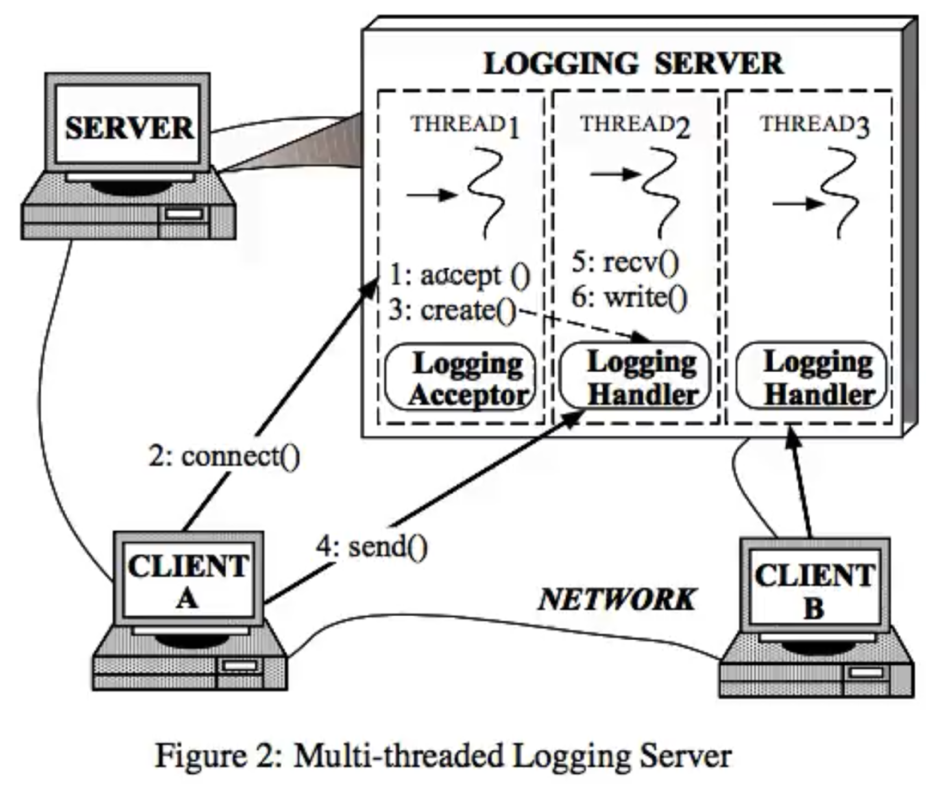
* Using Reactor Pools
* Selector**[]** selectors**;***//also create threads*
* **int** next **=** 0**;**
* **class** **Acceptor** **implements** Runnable**{**
* **public** **synchronized** **void** **run(){**
* **...**
* SocketChannel socketChannel **=** serverSocketChannel**.**accept**();**
* **if(**socketChannel**!=null){**
* **new** Handler**(**selectors**[**next**],**socketChannel**);**
* **if(++**next **==** selectors**.**length**){**
* next **=** 0**;**
* **}**
* **}**
* **}**
* **}**



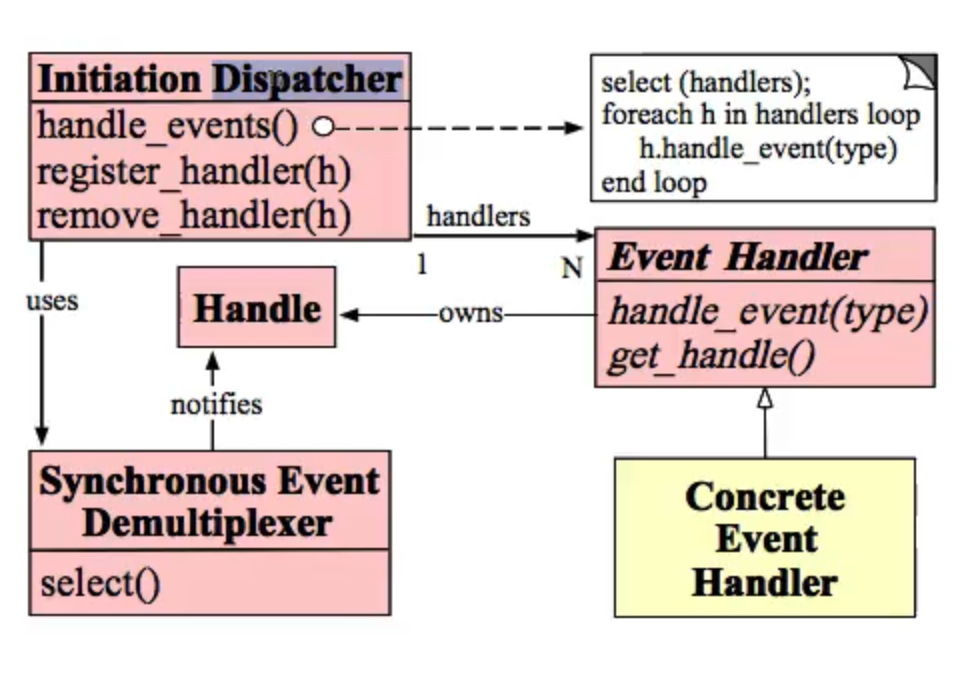
**论文：An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events 多路处理和同步事件分发处理的一种对象行为模式**[](" \l "an-object-behavioral-pattern-for-demult" \o "Permanent link)

**Reactor Also Known as Dispatcher,Notifier。**

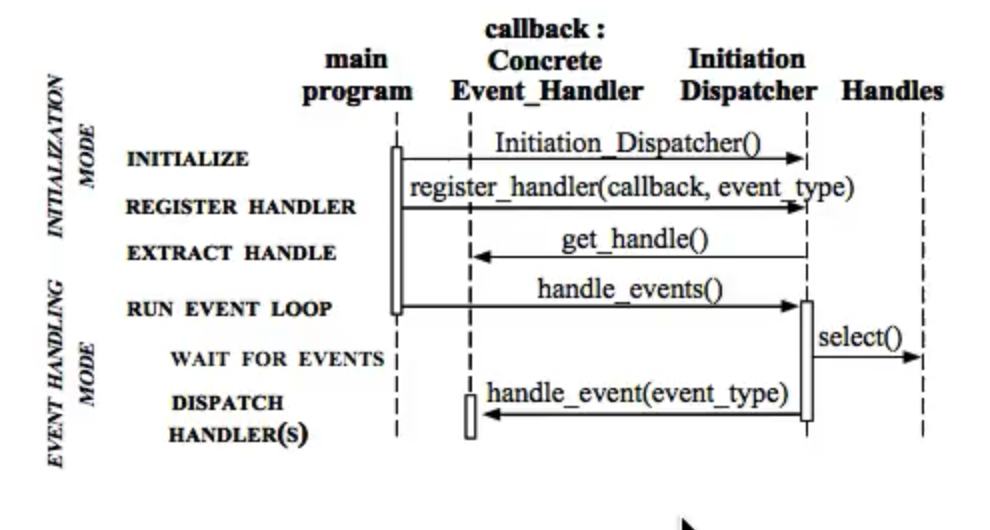
**Muti-threaded Logging Server**[](#muti-threaded-logging-server)

* 1、第一步是Acceptor调用 accept()阻塞等待
* 2、client connect()
* 3、Acceptor Create() Logging Handler
* 4、Client send() data to Server
* 5、recv()+write()
* 

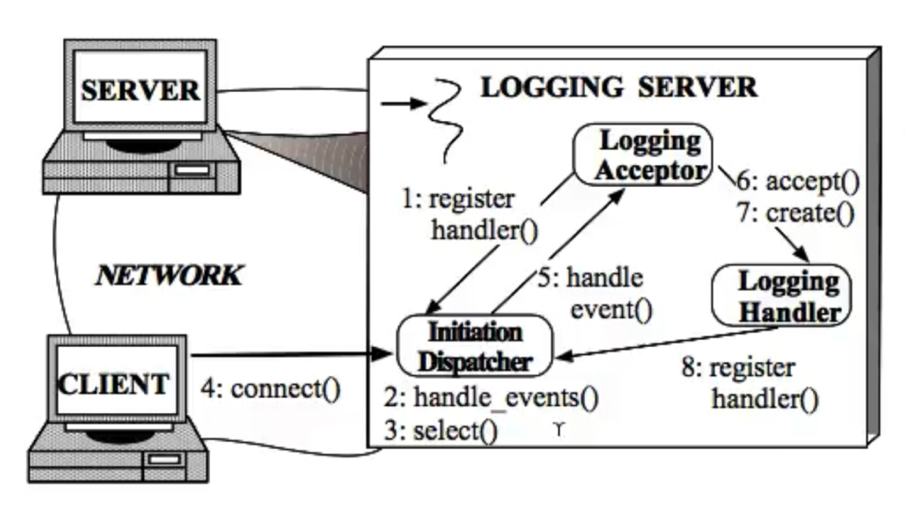
**The structure of the participants of the Reactor pattern is illustrate in the following OMT class diagram:**[](#the-structure-of-the-participants-of-th)



- Snchronous Event Demultipexer 同步事件分离器 + 同步事件分离器 都是通过底层操作系统来实现 + select() 就是通过操作系统实现的阻塞 + Snchronous Event Demultipexer 中的select会在某个事件发生时候 notifies **Handle** - Handle + 句柄/文件描述 + **Event Handler** owns **Handle** - Event Handler + 事件处理器 Concrete Event Handler实现了Event Handler + handle\_event(type) 这个方法是属于某个事件或者说是属于某个Handle + 我们写的handler都是由Dispatcher来负责调用 - Initiation Dispatcher + Initiation Dispatcher就是一个Dispatcher,首先 Event Handler会向Initiation Dispatcher进行注册。 + 接收注册之后的Initiation Dispatcher中会有: \* handle\_events() \* register\_handler(h) \* remove\_handler(h) + 最后根据事件调用handlers - d



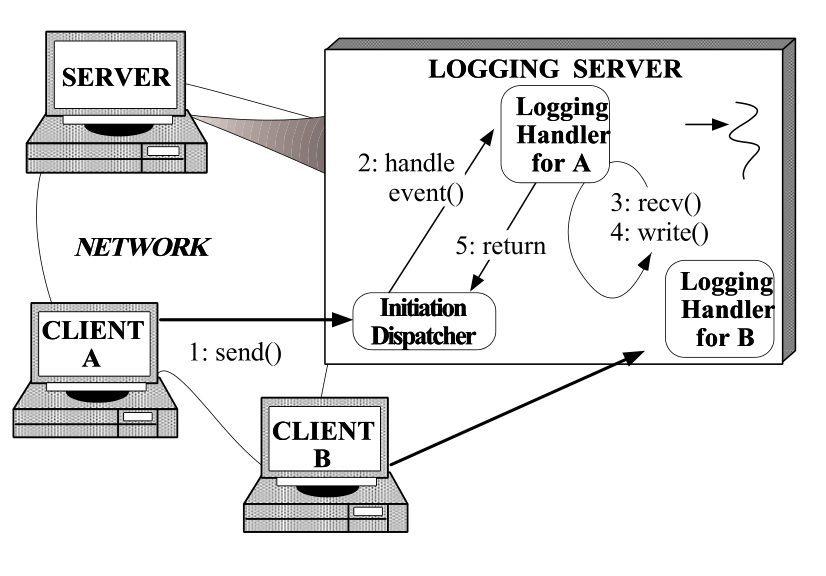
**Client Connects to a Reactive Logging Server**[](#client-connects-to-a-reactive-logging-s)



**This Sequence of steps can be summarized as follows:**[](#this-sequence-of-steps-can-be-summarize)

1. The logging server(1) registers the **Logging Acceptor** with the **Initiation Dispatcher** to handle connection requests;
2. The logging Server invokes the **handle events** method(2) of the **Initiation Dispatcher**。绑定OP\_READ事件
3. The **Initiation Dispatcher** invokes the synchronous event demutiplexing **elect**(3) operation to wait for connection requests or logging data to arrive;
4. A client connects(4) to the logging server;
5. The **Logging Acceptor** is notified by the **Initiation Dispatcher**(5) of the new connection request 实际上就是通知Logging Acceptor新的连接已经到来
6. The **Logging Acceptor** accepts(6) the new connection; 通过调用accept()方法调用
7. The **Logging Acceptor** creates(7) a **Logging Handler** to service the new client
8. **Logging Handler** registers(8) its socket handle(socket描述符) with the **Initation Dispatcher** and instructs the dispatcher to notify it when the socket becomes “ready for reading “

**Client Sends Logging Record to a Reactive Logging Server**[](#client-sends-logging-record-to-a-reacti)



**This Sequence of steps can be summarized as follows:**[](#this-sequence-of-steps-can-be-summarize)

1. The client sends(1) a logging record;
2. **Initiation Dispatcher** notifies(2) the associated **logging handler** when a client logging record is queued on its socket handle by OS;
3. The record is received(3) in a non-blocking manner (steps 2 and 3 repeat until the logging record has been received completely);
4. The **Logging Handler** process the logging record and writes(4)it to the standard output
5. The \*\*Logging Handler returns (5) control to the Initiation Dispatcher/s event loop;

**Netty的服务器启动流程是尤其重要的，需要形成一种宏观的影响，至少netty启动后，它里面的内存、线程啊、调度是个什么样子的。客户端的连接建立之后这个连接是怎么建立的。数据是怎么从客户端流转到服务端的这个是比较重要的。**

**传统模式和Reactor的模式对别有哪些劣势？**[](" \l "reactor_1" \o "Permanent link)

**传统的模式是什么样子的？**[](" \l "_1" \o "Permanent link)

服务器端比如通过serverSocket调用它的accept()方法来去进行阻塞进行客户端的连接，客户端的连接之后出现一种什么样的结果呢？显然就会出现一个socket处理就会对应服务端的一个线程，多个Socket就会对应服务器端的多个线程。这种方式的问题就是，服务器端会存在大量的线程跟客户端进行通信。

* 但是服务器端的线程是有限的；传统的一个socket一个线程的这种模式
* 线程在进行上下文切换的时候一定是有开销的；随着线程数量的增长，这个上下文切换的成本也是越来越高
* 可能这个连接建立好之后，客户端向服务端发送了一次数据，紧接着这个连接在一定时间内就没有数据进行传输了。虽然没有数据传输了，但是这个socket还是得保持着，既然socket要保持着，那么线程肯定是要存活着一直在那运行着，然而线程其实什么都没有做，因为对应着客户端本身是没有向服务端发送数据的。他处于一种空置的状态，这种情况就会导致服务器端有大量的线程存在。但是每个线程都没有什么事情可做，因为这个时候数据交互的并不是很频繁。这样会导致线程资源极大浪费
* 对于高并发一定会产生以上的种种问题。

**这也是reactor模式出现的一种历史原因**，因为有以上三种情况才导致reacot模式的出现，显然socket的出现就能解决传统的一个socket一个线程所面临的这些问题。

**Reactor模式的角色构成**[](" \l "reactor_2" \o "Permanent link)

Reactor模式一共有5总角色构成

1. Handle: 就是句柄或是文件描述符；对于window就是句柄，对于linux就是文件描述符；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的事件，比如文件描述符，或者是针对网络编程中的Socket描述符。也可以是来自于内部；外部事件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等。它本质上就是一个文件描述符。Handle就是产生事件的地方。我们要监听事件其实就是要监听Handle。
2. Synchronous Event Demultiplexer（同步事件分离器）:同步首先表明是阻塞的。他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）。调用方在调用它的时候会被阻塞，一直阻塞到同步事件分离器上事件产生为止。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JavaNIO中，同步事件分离器对应的组件就是Selector，对应的阻塞方法就是select()方法。
3. Event Handler(事件处理器)：本身由多个回调方法构成与引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，他为开发者提供了大量的回调方法，供我们在特定事件产生的时候实现相应的对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。他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的逻辑。他本质上就是我们所编写的一个个的处理器实现
5. 最为核心的一个角色–Initiation Dispatcher(初始分发器)：其实就是douglea的reactor模式图中的Reacotr。即Initiation Dispatcher == Reactor(MainReactor+SubReactor); 他本身定义了一些规范，这些规范用于控制事件的调度方式，同时又提供了应用进行事件处理器的注册、删除等操作。他本身是整个事件处理器的核心所在，Initiation Dispatcher 会通过同步事件分离器来等待事件的发生。一旦事件发生，Initiation Dispatcher首先会分离出每一个事件（其实就是遍历SelectionKeys 这个set集合取出来里面的每一个SelectionKey,因为每个selectionKey就是一个事件），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。
6. select（handlers）;
7. foreach h in handlers loop
8. h.handle\_events(type)
9. end loop

**channelRead0()方法是谁来调用的？**[](" \l "channelread0" \o "Permanent link)

**public** **class** **MyServerHandler** **extends** SimpleChannelInboundHandle**<**String**>{**

@Override

**protected** **void** **channelRead0(**ChannelHandlerContext ctx**,** String msg**){**

**}**

**}**

答案：**MyServerHandler.channelRead0()方法是由IO线程池中某个线程池调用，也就是WorkGroup（subReactor=Initiation Dispatcher）调用**，因此得出另一个结论，当channelRead0()方法中的逻辑很耗时的话，是不是会阻塞IO线程池里的线程呢？ 答案：是的，当然是的了，这个时候就要采用业务线程池用异步的方式进行处理。

**以上五个组件在一个请求或者事件产生之后一直到对这个事件进行响应的业务处理整个流程是什么样子的？**[](" \l "_2" \o "Permanent link)

1. 首先执行的是Initiation Dispatcher，也就是Reactor这个角色，首先将若干个Event Handler（具体的就是Concrete Event Handler）注册到Initiation 。
2. 注册的同时指定感兴趣的事件，这个事件是通过Handle来标识的，并且Handle是被Event Handler所owns的，这个事件对应于NIO中就是InterestKey（OP\_Read）
3. 注册之后，紧接着Initiation Dispatcher就开启了他自己的事件循环(死循环)，类似与Netty NIOEventLoop一样是一个死循环。并在这个事件循环中它会通过Synchronous Event Demultiplexer等待事件的产生（通过select()方法，poll\epoll等等）
4. 当有感兴趣的事件（新的连接或者新的数据进来）在与Event\_Handler关联的Handle上发生时，然后同步事件分离器就会获取产生的事件的集合，然后产生的事件的集合就会返回给Initiation Dispatcher(它会知道这个事件是什么),然后选择与这个事件绑定的事件处理器，然后遍历这个事件处理器，根据这个事件类型找出注册其上的Concrete event handler。并由Initiation Dispatcher来通知注册在其上的Concrete Event Handler

**Reactor模式的流程**[](" \l "reactor_3" \o "Permanent link)

1. 当应用向Initiation Dispatcher注册具体的事件处理器时，会标识出该事件处理器希望Initiation Dispatcher在某个事件发生时向其通知的该事件，该事件与Handle关联。
2. Initiation Dispatcher会要求每个时间处理器向其传递内部的Handle，该Handle向操作系统标识了事件处理器。
3. 当所有的事件处理器注册完毕后，引用会调用handle\_events方法来启动Initiation Dispatcher事件循环。这时，Initiation Dispatcher会将注册的事件管理器的Handle合并起来，并使用同步事件分离器等待这些事件的发生。比如TCP协议的会使用select同步事件分离器来等待客户端发送数据到达连接的socket handle上。
4. 当与某个事件源对应的Handle变为ready状态时候(比如，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。当事件发生的时，Initiation Dispatcher会将被事件源激活的Handle作为key来寻找恰当的时间处理器回调方法。
6. Initiation Dispatcher会回调事件处理器的handle\_events回调方法来执行特定于应用的功能(开发这自己所编写的功能)，从而相应这个事件。所发生的事件类型可以作为该方法参数冰杯该方法内部使用来及执行额外的特定玉服务的分离与分发

**Acceptor组件的作用**[](" \l "acceptor" \o "Permanent link)

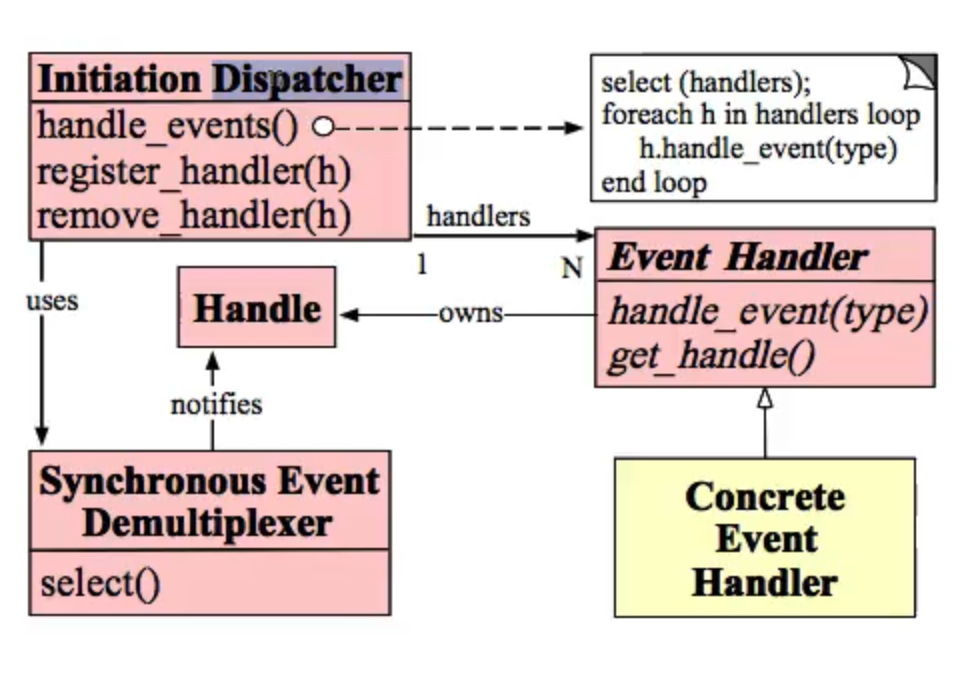
在dougeLea的文章中，MainRactor(BossGroup)的作用：接收连接，也是它唯一的作用。接收客户端发过来的连接。那么连接接收到之后，它就会把连接扔给后面的subReactor(workGroup)

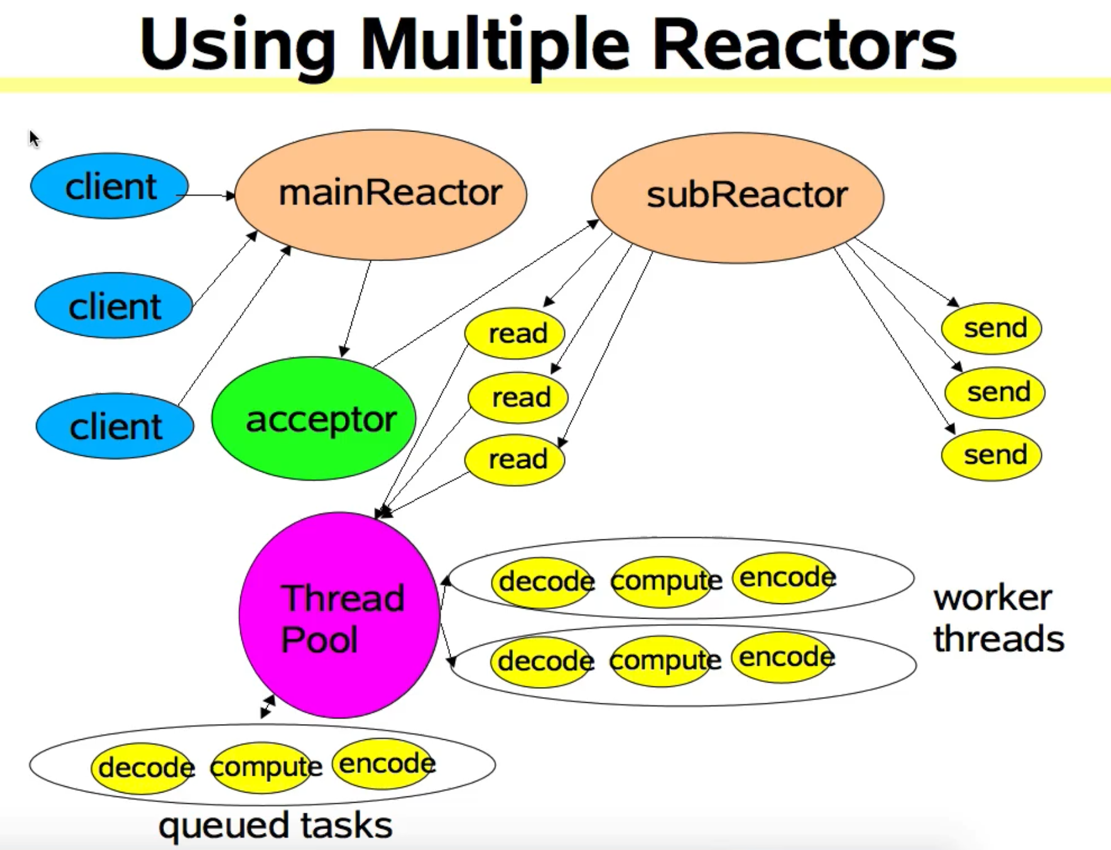
subReacotr(workGroup)的作用：就是IO线程组，他里面就是进行IO操作和读取和写入等等。

**那么mainReactor接收到连接之后是怎么把这个连接丢给subReacotor的呢？**[](" \l "mainreactorsubreacotor" \o "Permanent link)

其实就是Accptor的作用，他来干了这个事件。

netty在服务器绑定到特定的端口号的时候，首先创建一系列的组件及组件之间的关系设定，再去绑定到特定的端口号，在这个过程当中，会往mainReactor换句话说就是会往BossGroup里面会增加一个Handler,那么这个handler在netty里就是ChannelInitializerInboundHandler,在其里面会new出来一个ServerBoostrapAccetor这样的一个对象。这个对象是我们每次绑定到一个端口号上的时候，netty都会帮着去做的事件。换句话说这个Acceptor是netty内置的一个组件，它是在服务器绑定到特定的地址和端口的时候自动就帮我们创建好的实例。并且把它加入到了pipeline当中（通道当中），**Acceptor本身就是一个处理器**，而且在加入到pipeline的时候会加入到最后的位置。当前面的一系列设置都完成后，进入Acceptor，紧接着Acceptor收到了客户端发过来的数据之后，就在Acceptor的channelRead0()方法中将原来的客户端的Channel绑定到workGroup当中。 就是因为有两个Reactor（mainReactor和subReactor）所以才需要acceptor将mainReactor接收到连接之后把这个连接丢给subReacotor。如果只有一个reactor则不需要。





**public** **class** **demo** **{**

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

EventLoopGroup bossGrouup **=** **new** NioEventLoopGroup**();**

EventLoopGroup workGrouup **=** **new** NioEventLoopGroup**();**

ServerBootstrap serverBootStrap **=** **new** ServerBootstrap**();**

ChannelFuture channelFuture **=** serverBootStrap**.**bind**(**8899**).**sync**();**

**}**

**}**

**public** ChannelFuture **bind(int** inetPort**)** **{**

**return** bind**(new** InetSocketAddress**(**inetPort**));**

**}**

**public** ChannelFuture **bind(**SocketAddress localAddress**)** **{**

validate**();**

**if** **(**localAddress **==** **null)** **{**

**throw** **new** NullPointerException**(**"localAddress"**);**

**}**

**return** doBind**(**localAddress**);**

**}**

**private** ChannelFuture **doBind(final** SocketAddress localAddress**)** **{**

**final** ChannelFuture regFuture **=** initAndRegister**();**

**......**

**}**

**final** ChannelFuture **initAndRegister()** **{**

Channel channel **=** **null;**

**try** **{**

channel **=** channelFactory**.**newChannel**();**

init**(**channel**);**

**}**

ServerBootstrap**.**java **----->**ServerBootstrapAcceptor

@Override

**void** **init(**Channel channel**)** **throws** Exception **{**

**final** Map**<**ChannelOption**<?>,** Object**>** options **=** options0**();**

**synchronized** **(**options**)** **{**

setChannelOptions**(**channel**,** options**,** logger**);**

**}**

**final** Map**<**AttributeKey**<?>,** Object**>** attrs **=** attrs0**();**

**synchronized** **(**attrs**)** **{**

**for** **(**Entry**<**AttributeKey**<?>,** Object**>** e**:** attrs**.**entrySet**())** **{**

@SuppressWarnings**(**"unchecked"**)**

AttributeKey**<**Object**>** key **=** **(**AttributeKey**<**Object**>)** e**.**getKey**();**

channel**.**attr**(**key**).**set**(**e**.**getValue**());**

**}**

**}**

ChannelPipeline p **=** channel**.**pipeline**();**

**final** EventLoopGroup currentChildGroup **=** childGroup**;**

**final** ChannelHandler currentChildHandler **=** childHandler**;**

**final** Entry**<**ChannelOption**<?>,** Object**>[]** currentChildOptions**;**

**final** Entry**<**AttributeKey**<?>,** Object**>[]** currentChildAttrs**;**

**synchronized** **(**childOptions**)** **{**

currentChildOptions **=** childOptions**.**entrySet**().**toArray**(**newOptionArray**(**0**));**

**}**

**synchronized** **(**childAttrs**)** **{**

currentChildAttrs **=** childAttrs**.**entrySet**().**toArray**(**newAttrArray**(**0**));**

**}**

p**.**addLast**(new** ChannelInitializer**<**Channel**>()** **{**

@Override

**public** **void** **initChannel(final** 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**(**

ch**,** currentChildGroup**,** currentChildHandler**,** currentChildOptions**,** currentChildAttrs**));**

**}**

**});**

**}**

**});**

**}**

ServerBootstrapAcceptor**.**java

**private** **static** **class** **ServerBootstrapAcceptor** **extends** ChannelInboundHandlerAdapter **{**

**private** **final** EventLoopGroup childGroup**;**

**private** **final** ChannelHandler childHandler**;**

**private** **final** Entry**<**ChannelOption**<?>,** Object**>[]** childOptions**;**

**private** **final** Entry**<**AttributeKey**<?>,** Object**>[]** childAttrs**;**

**private** **final** Runnable enableAutoReadTask**;**

ServerBootstrapAcceptor**(**

**final** Channel channel**,** EventLoopGroup childGroup**,** ChannelHandler childHandler**,**

Entry**<**ChannelOption**<?>,** Object**>[]** childOptions**,** Entry**<**AttributeKey**<?>,** Object**>[]** childAttrs**)** **{**

**this.**childGroup **=** childGroup**;**

**this.**childHandler **=** childHandler**;**

**this.**childOptions **=** childOptions**;**

**this.**childAttrs **=** childAttrs**;**

*// Task which is scheduled to re-enable auto-read.*

*// It's important to create this Runnable before we try to submit it as otherwise the URLClassLoader may*

*// not be able to load the class because of the file limit it already reached.*

*//*

*// See https://github.com/netty/netty/issues/1328*

enableAutoReadTask **=** **new** Runnable**()** **{**

@Override

**public** **void** **run()** **{**

channel**.**config**().**setAutoRead**(true);**

**}**

**};**

**}**

**\*\*\*\*\*\*\*\*\*\***在这里完成连接由MainReactor向subReactor转移的过程

@Override

@SuppressWarnings**(**"unchecked"**)**

**public** **void** **channelRead(**ChannelHandlerContext ctx**,** Object msg**)** **{**

**final** Channel child **=** **(**Channel**)** msg**;**

child**.**pipeline**().**addLast**(**childHandler**);**

setChannelOptions**(**child**,** childOptions**,** logger**);**

**for** **(**Entry**<**AttributeKey**<?>,** Object**>** e**:** childAttrs**)** **{**

child**.**attr**((**AttributeKey**<**Object**>)** e**.**getKey**()).**set**(**e**.**getValue**());**

**}**

**\*\*\*\*\*\*\*\*\*\*\***就是下面childGroup**.**register**(**child**).**addList**{}** 完成具体转移

**\*\*\*\*\*\*\*\*\*\*\***其中childGroup 就是 **private** **final** EventLoopGroup childGroup**;** 其实就是workGroup

**\*\*\*\*\*\*\*\*\*\*\***child就是channel

**try** **{**

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**);**

**}**

**}**