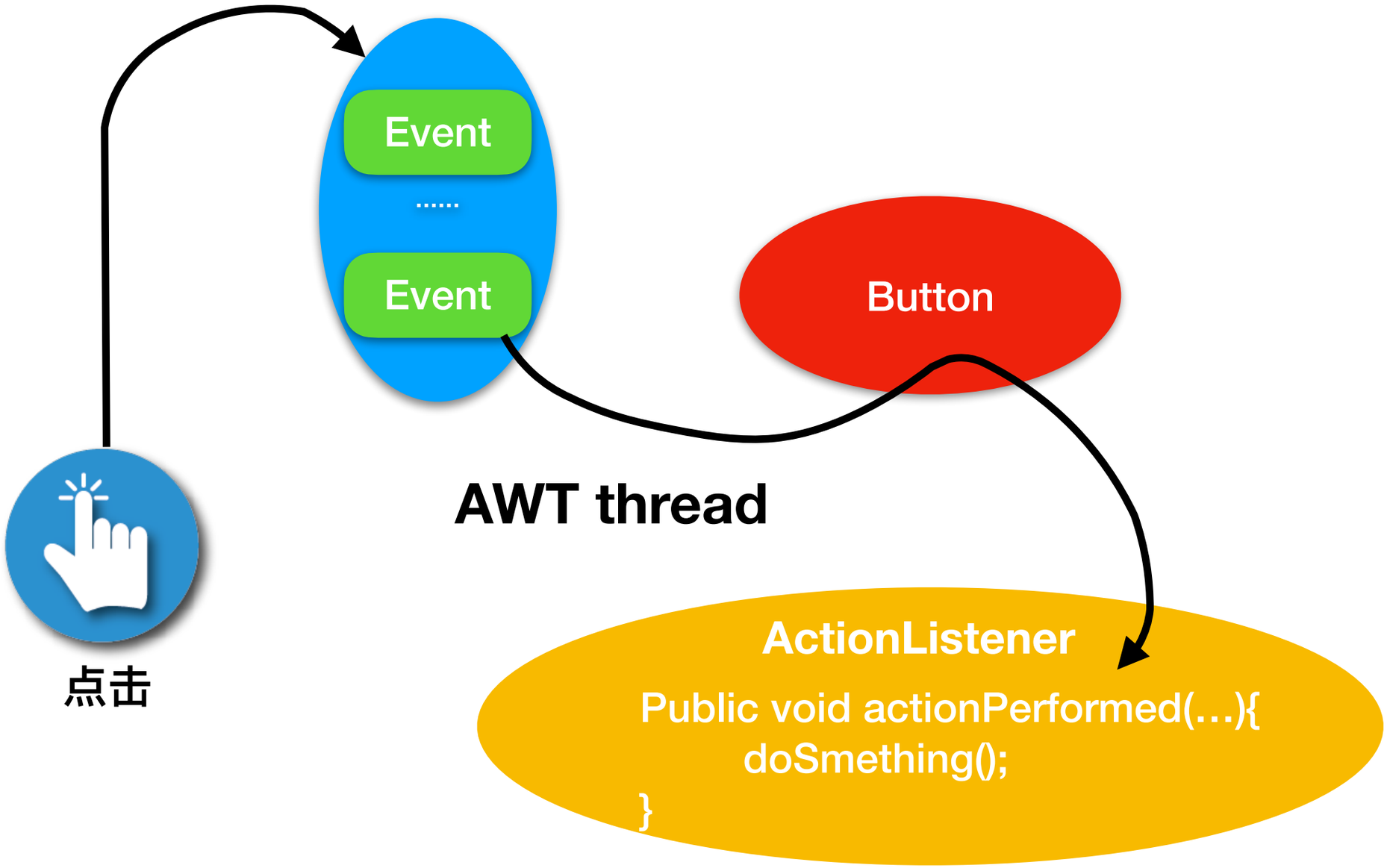
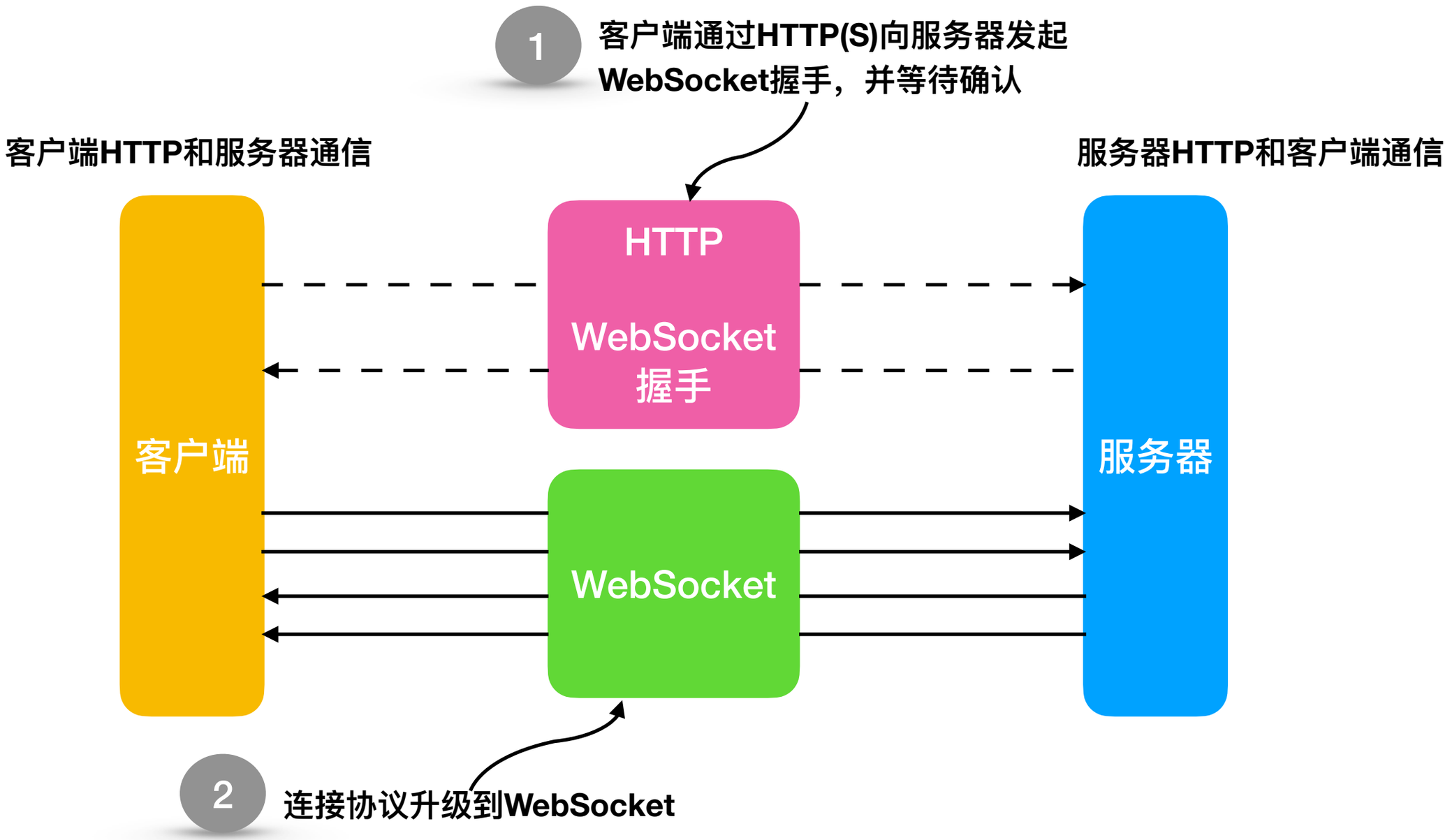
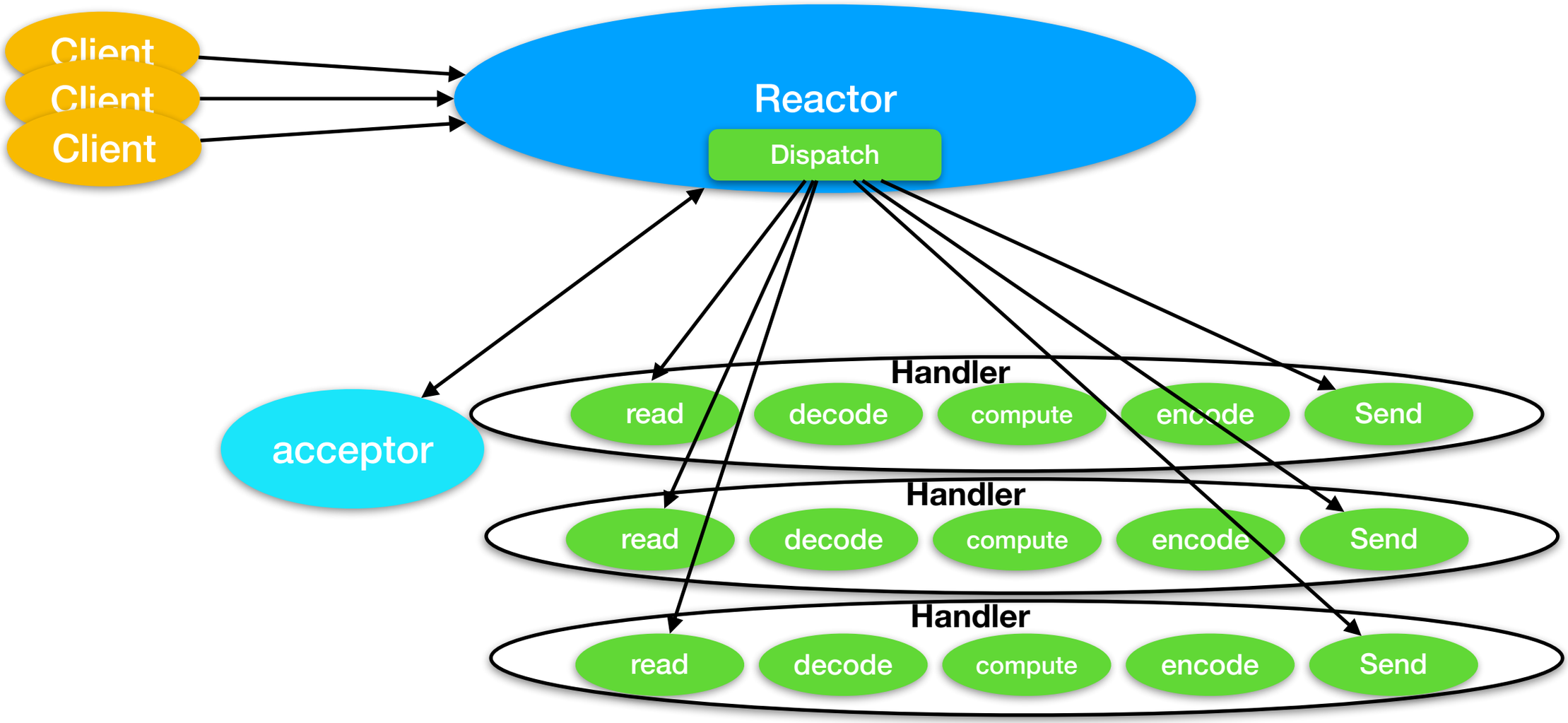

Event
.....
Event
Button
AWT thread
点击
ActionListener
Public void actionPerformed(…){
doSmething();
}

AWT事件驱动



Websocket协议

单Reactor单线程



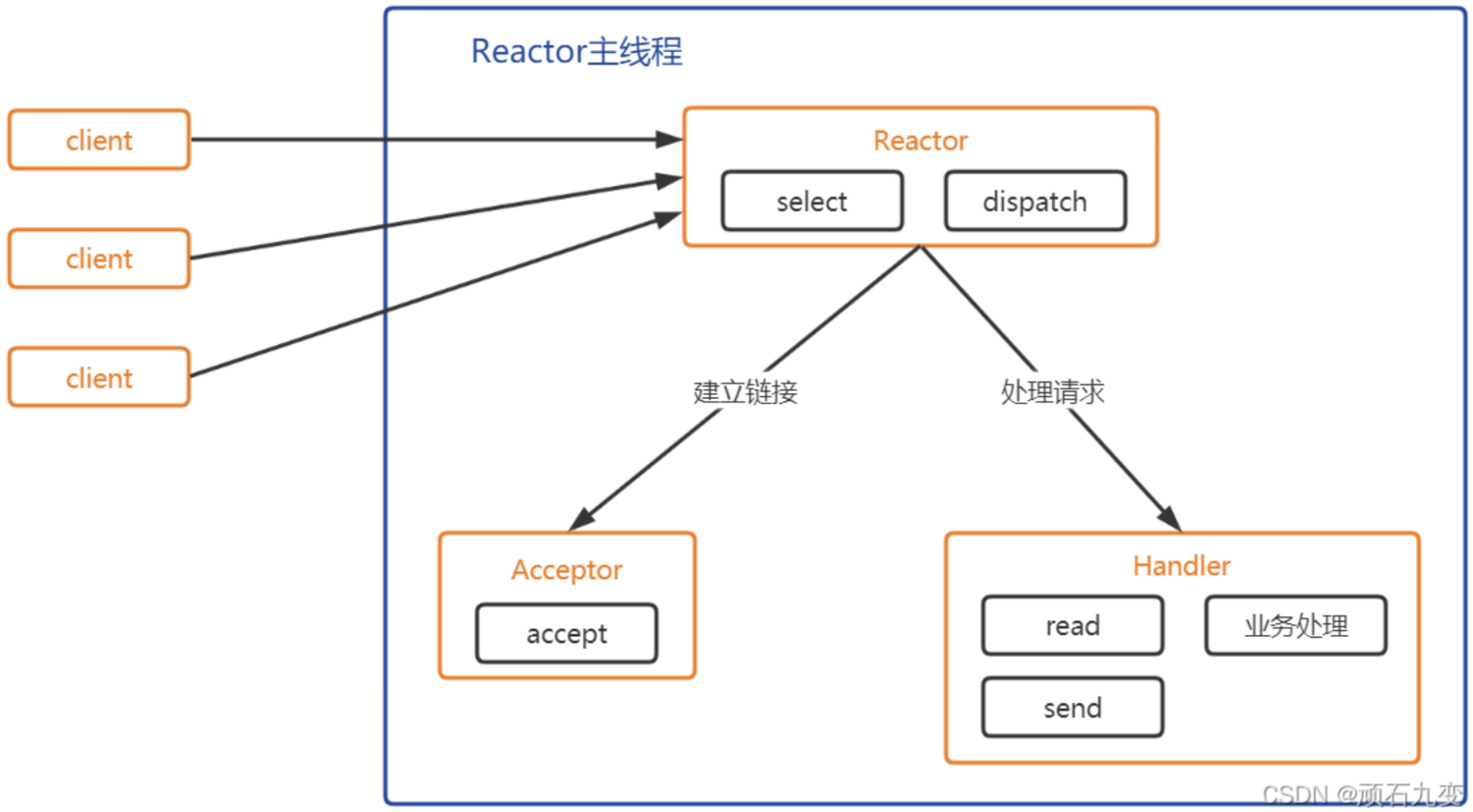
Reactor模型是依赖IO多路复用技术实现监听IO事件，从而源源不断的产生IO就绪事件，在Linux系统下我们使用epoll来进行IO多路复用，我们以Linux系统为例：

单Reactor意味着只有一个epoll对象，用来监听所有的事件，比如连接事件，读写事件。

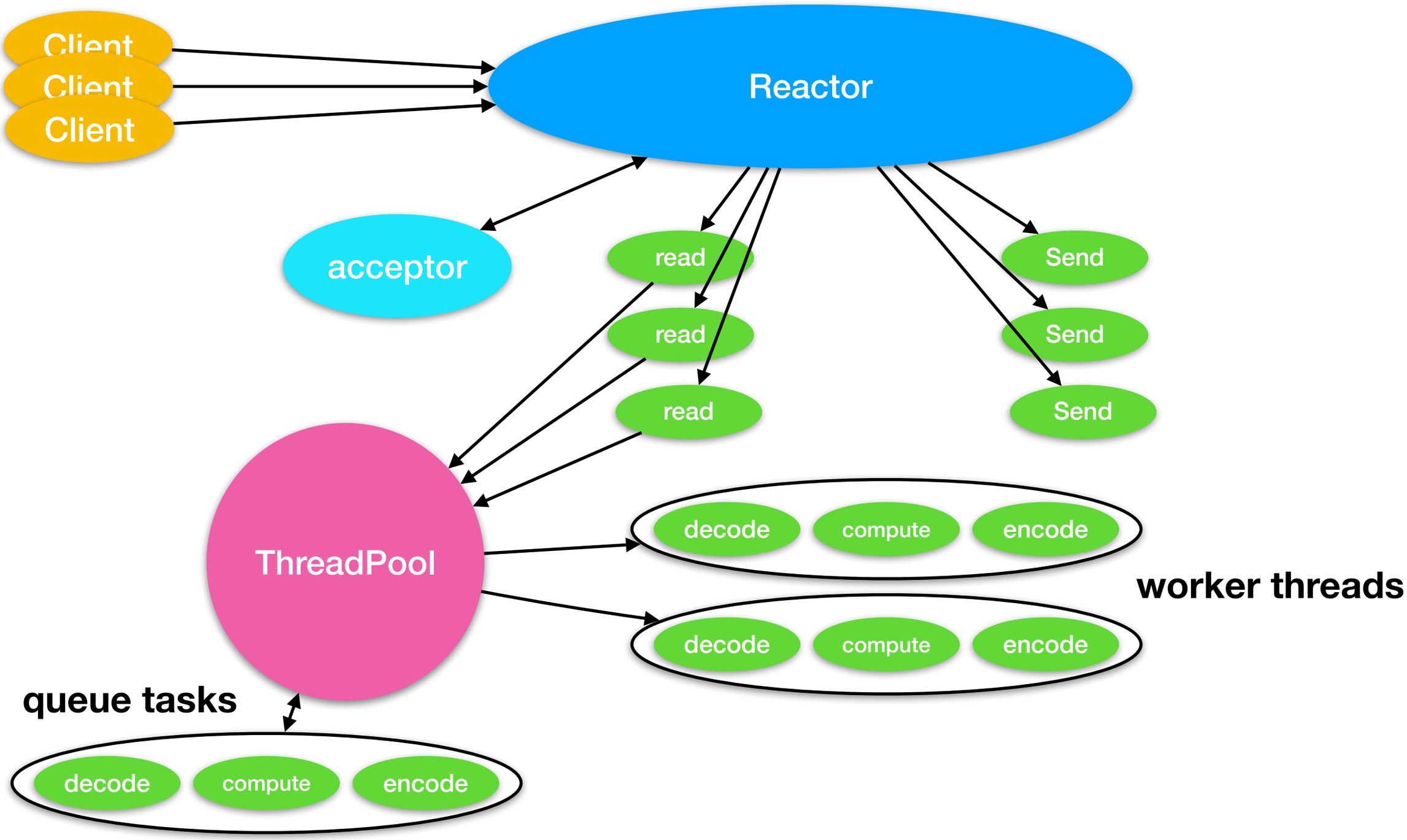
单线程意味着只有一个线程来执行epoll_wait获取IO就绪的Socket，然后对这些就绪的Socket执行读写，以及后边的业务处理也依然是这个线程。

单Reactor单线程模型就好比我们开了一个很小很小的小饭馆，作为老板的我们需要一个人干所有的事情，包括：迎接顾客（accept事件），为顾客介绍菜单等待顾客点菜(IO请求)，做菜（业务处理），上菜（IO响应），送客（断开连接）。

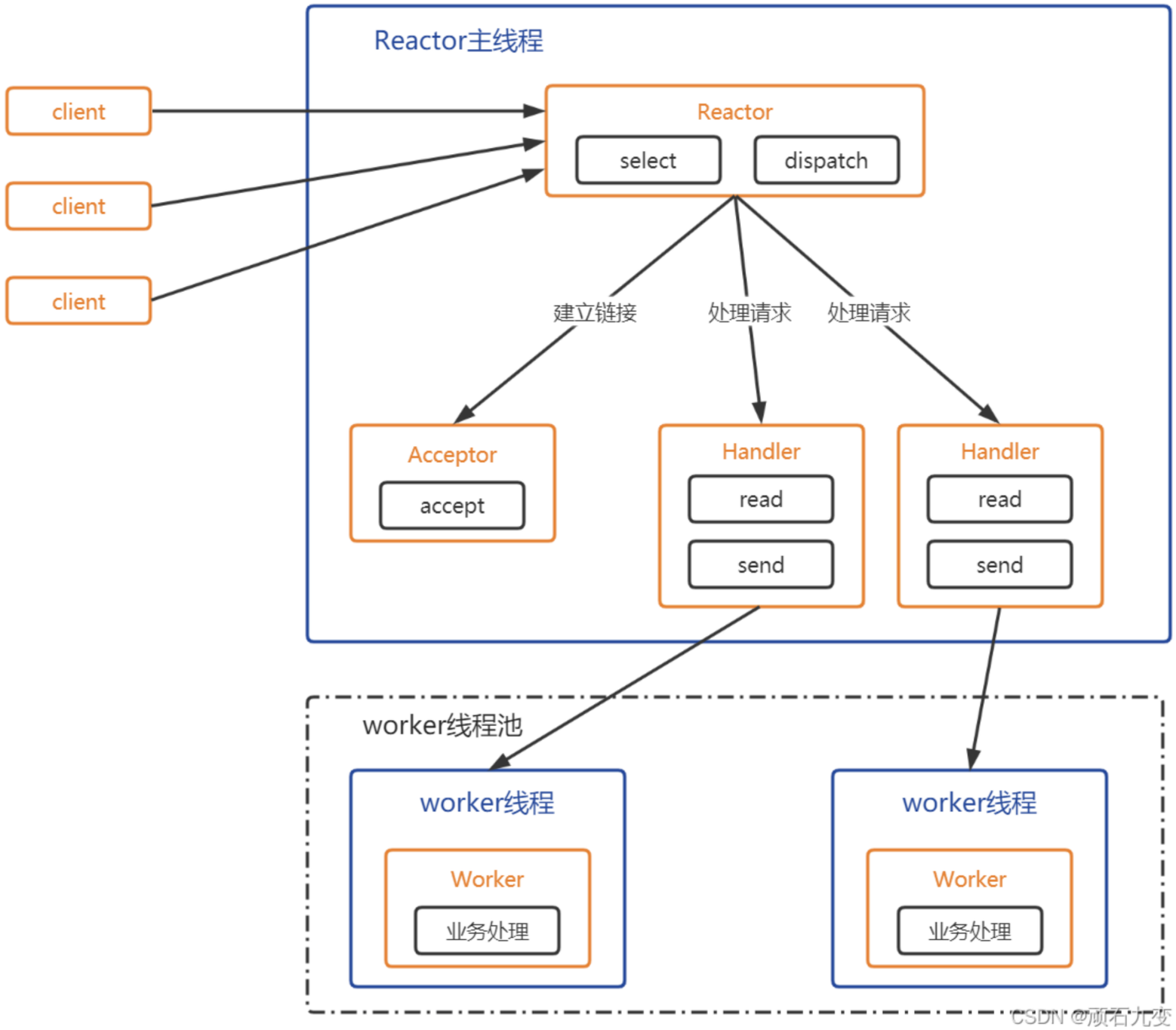
https://blog.csdn.net/wlddhj/article/details/123872275



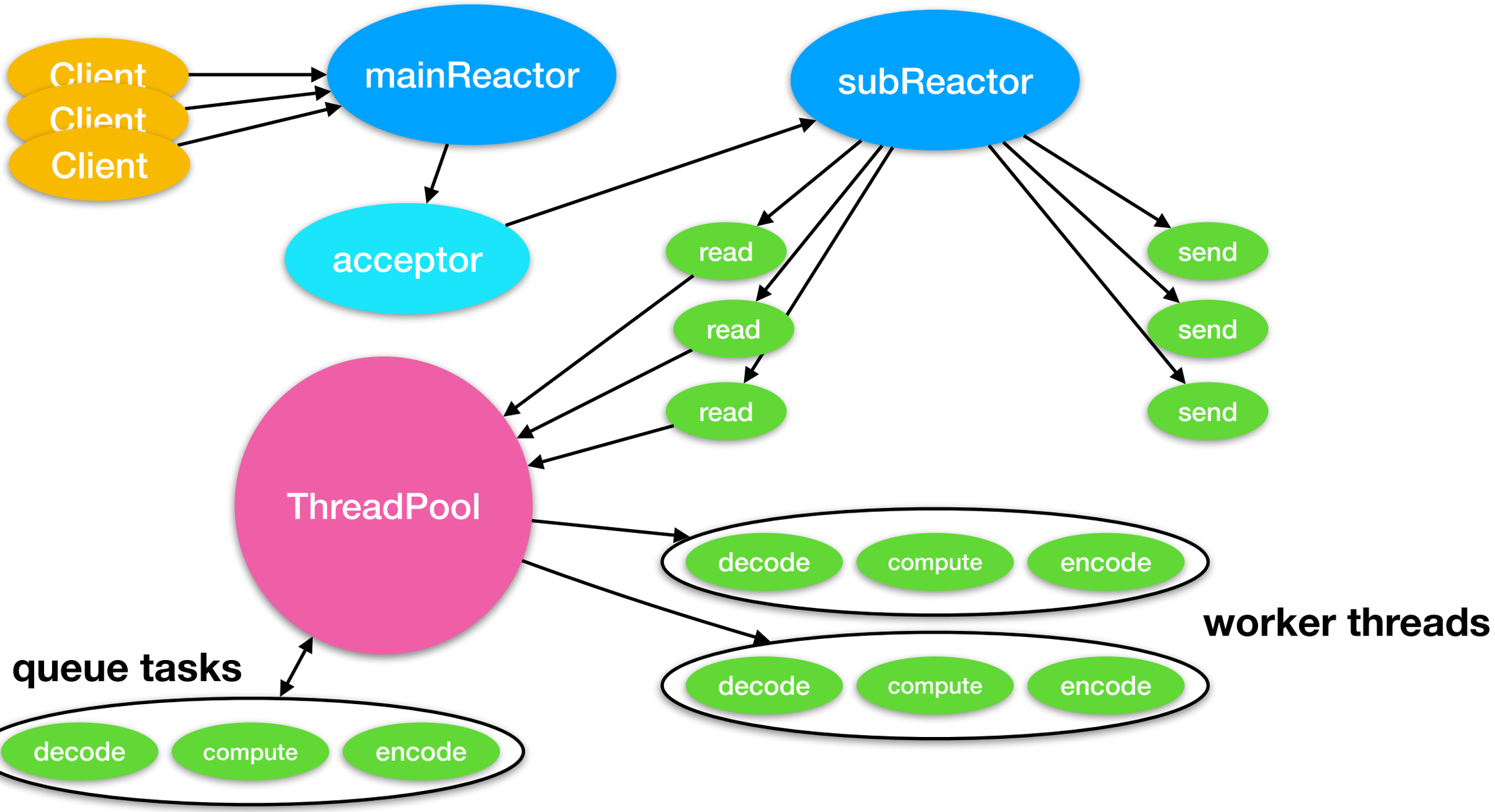
单Reactor多线程



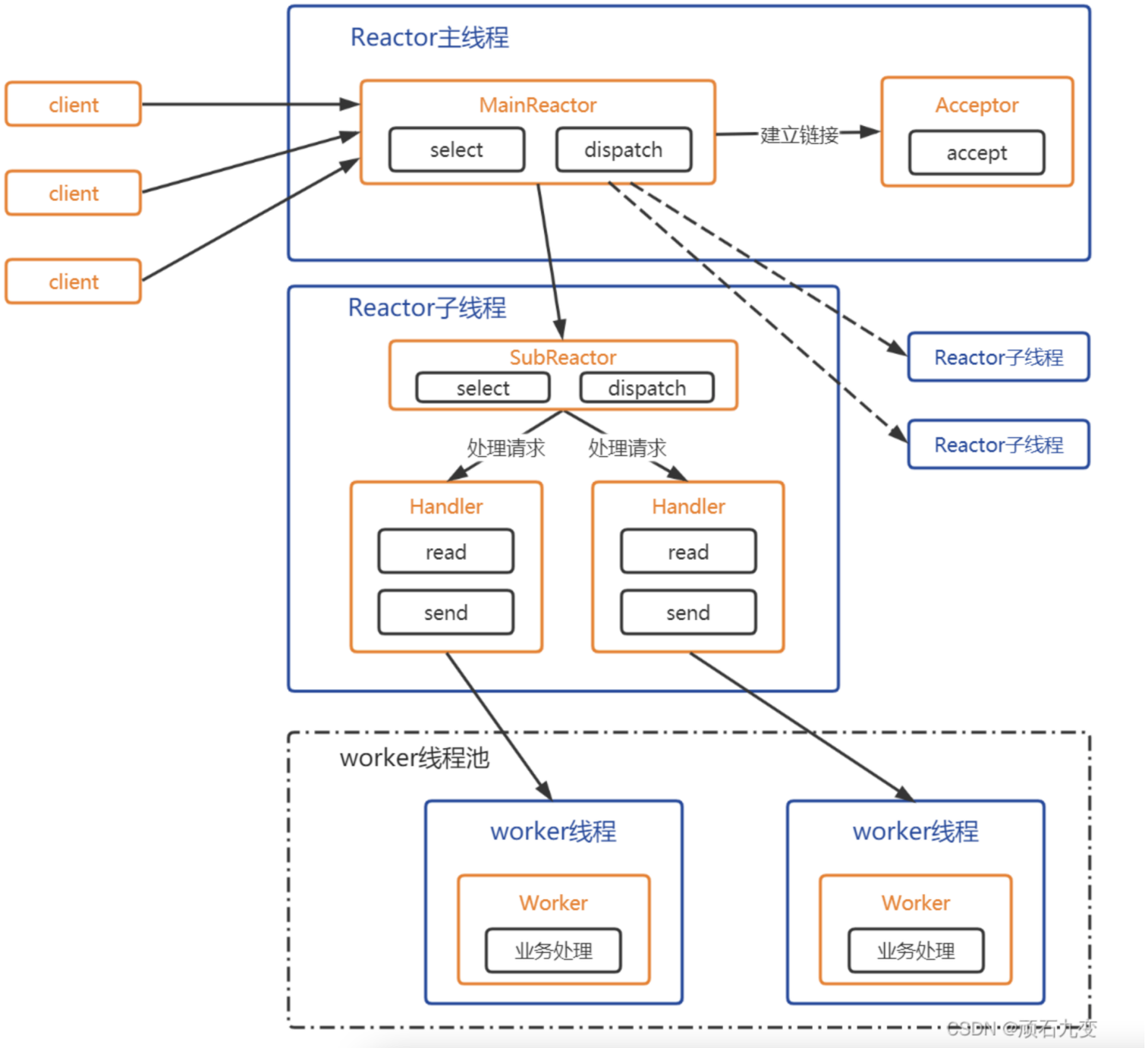
也是只有一个epoll对象来监听所有的IO事件，一个线程来调用epoll_wait获取IO就绪的Socket。
但是当IO就绪事件产生时，这些IO事件对应处理的业务Handler，我们是通过线程池来执行。这样相比单Reactor单线程模型提高了执行效率，充分发挥了多核CPU的优势。

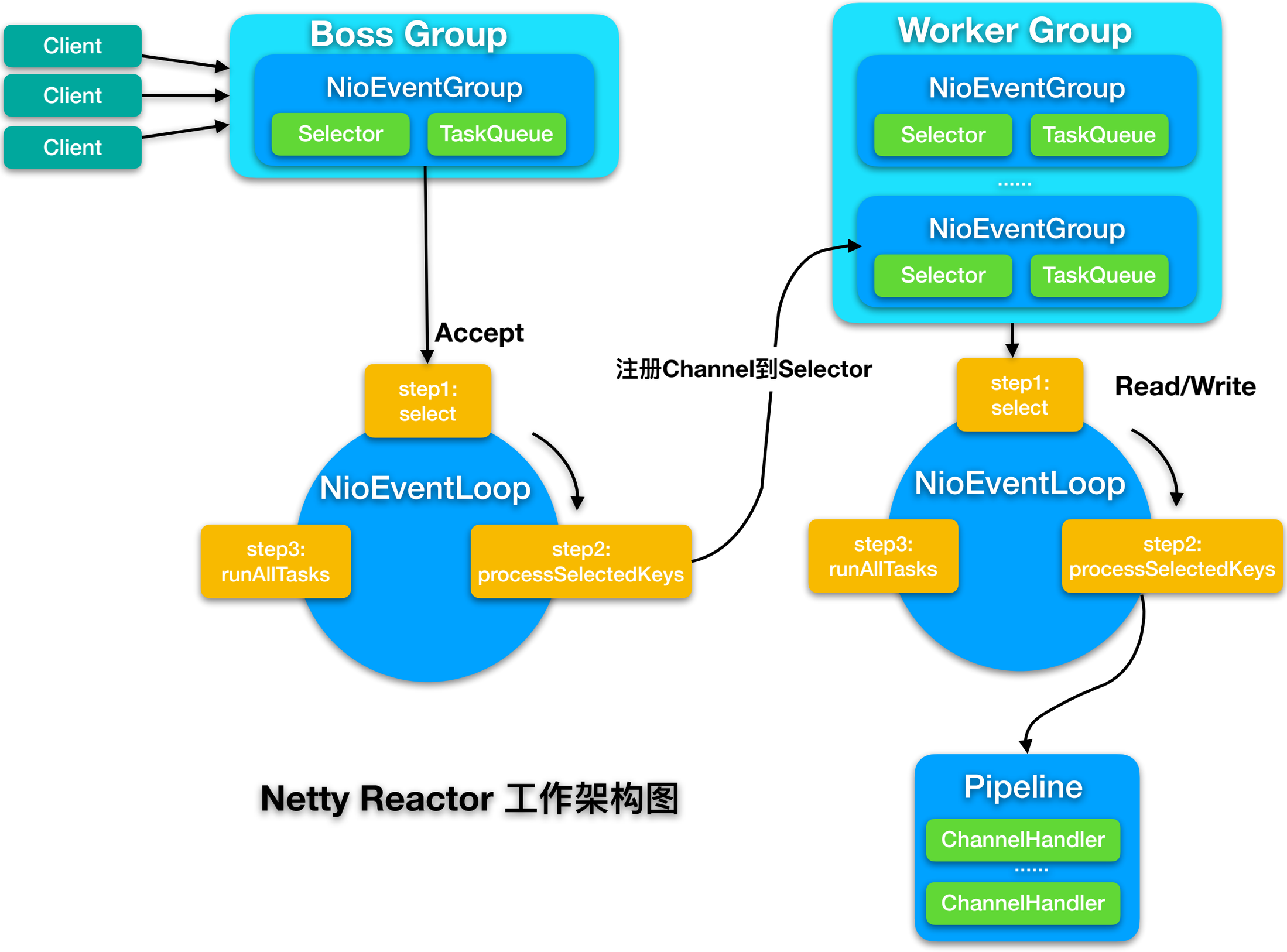

Reactor主线程
client
client
client
Reactor
select
dispatch
建立链接
处理请求
处理请求
Acceptor
accept
Handler
read
send
Handler
read
send
worker线程池
worker线程
Worker
业务处理
worker线程
Worker
业务处理

经典主从Reactor多线程模型

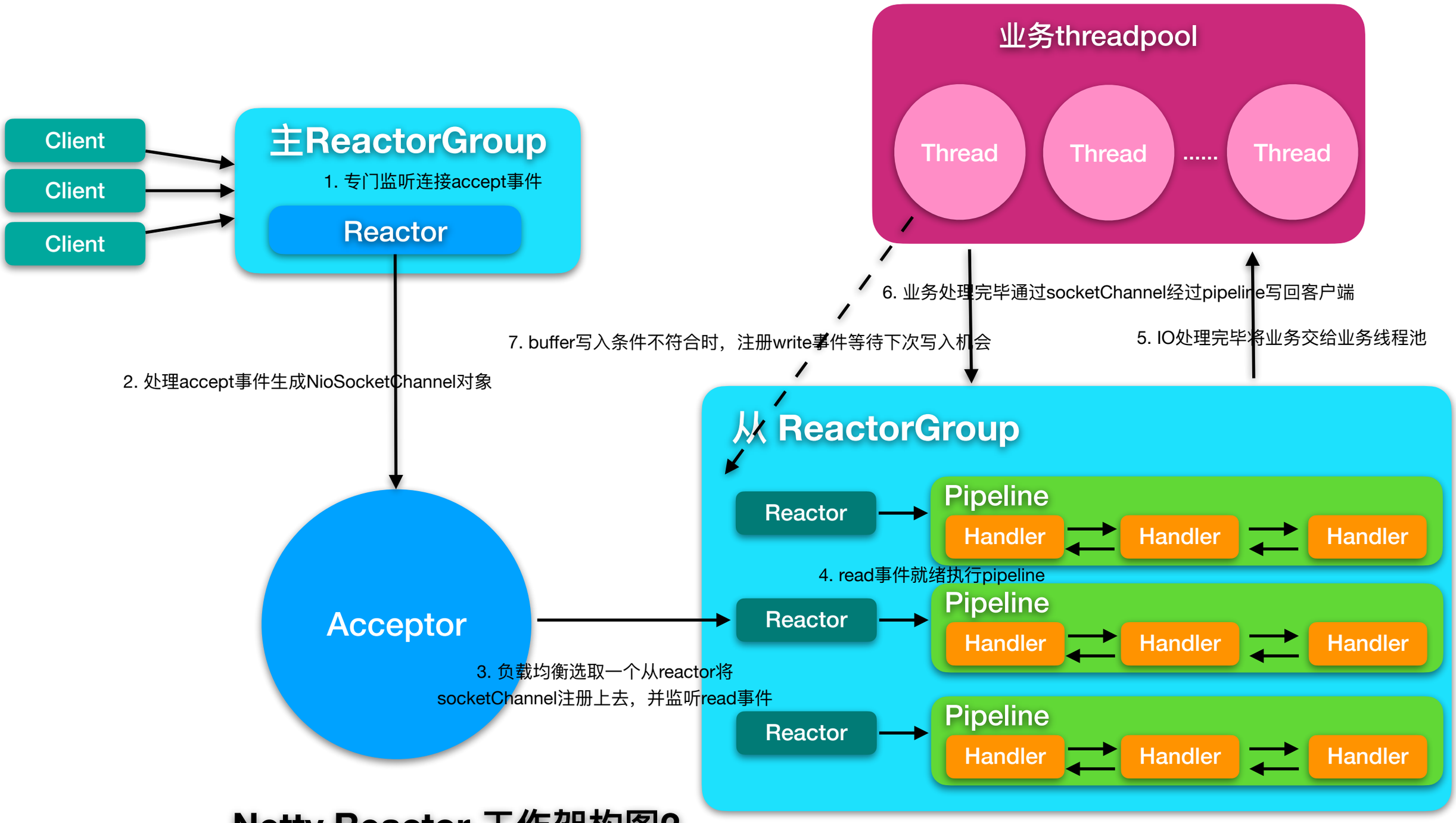


我们由原来的单Reactor变为了多Reactor。主Reactor用来优先专门做优先级最高的事情，也就是迎接客人（处理连接事件），对应的处理Handler就是图中的acceptor。
当创建好连接，建立好对应的socket后，在acceptor中将要监听的read事件注册到从Reactor中，由从Reactor来监听socket上的读写事件。
最终将读写的业务逻辑处理交给线程池处理。
注意：这里向从Reactor注册的只是read事件，并没有注册write事件，因为read事件是由epoll内核触发的，而write事件则是由用户业务线程触发的（什么时候发送数据是由具体业务线程决定的），所以write事件理应是由用户业务线程去注册。
用户线程注册write事件的时机是只有当用户发送的数据无法一次性全部写入buffer时，才会去注册write事件，等待buffer重新可写时，继续写入剩下的发送数据、如果用户线程可以一股脑的将发送数据全部写入buffer，那么也就无需注册write事件到从Reactor中。


Reactor主线程
client
client
client
MainReactor
select
dispatch
建立链接
Acceptor
accept
Reactor子线程
SubReactor
select
dispatch
处理请求
处理请求
Handler
read
send
Handler
read
send
Reactor子线程
Reactor子线程
worker线程池
worker线程
Worker
业务处理
worker线程
Worker
业务处理



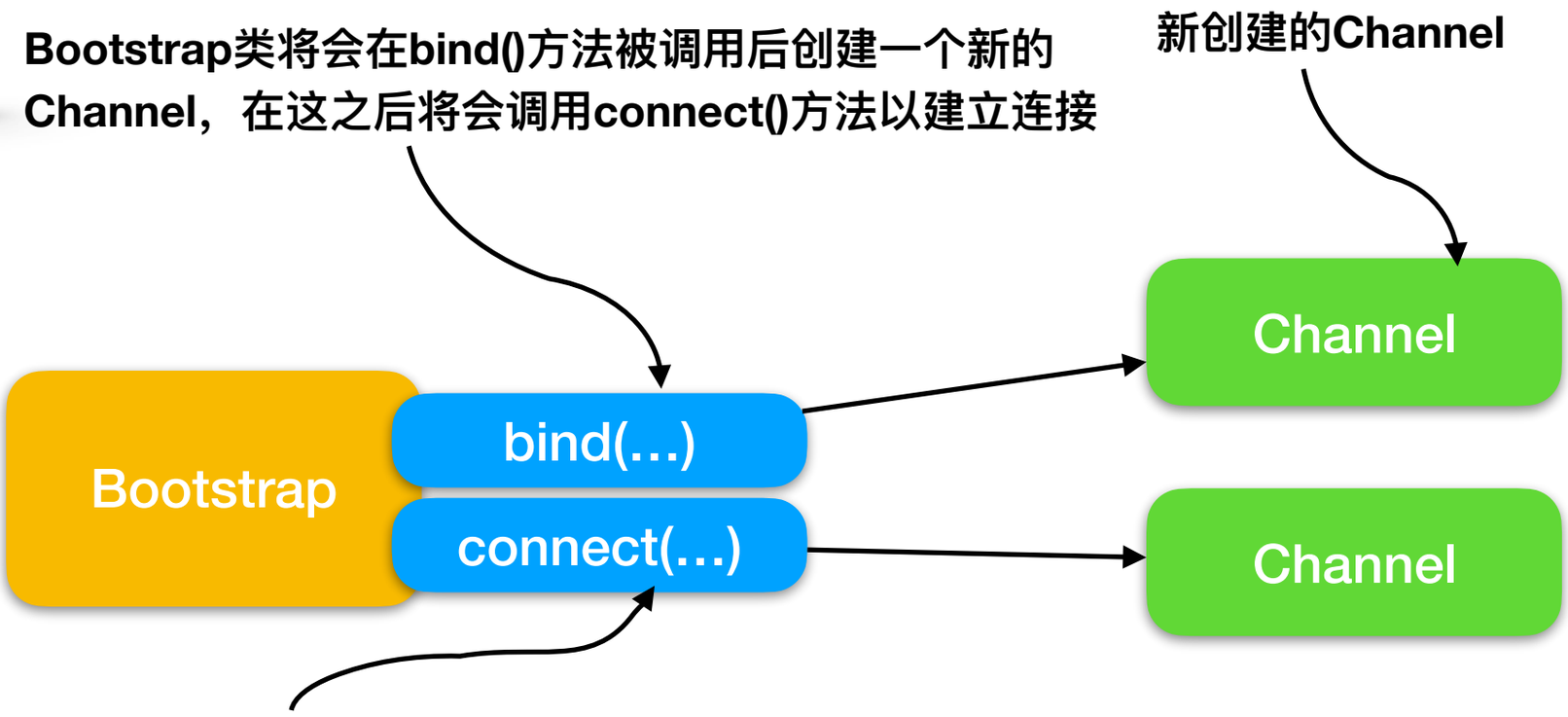
Netty Reactor 工作架构图



Netty Reactor 工作架构图2

https://heapdump.cn/article/4126756

1 Bootstrap类将会在bind()方法被调用后创建一个新的Channel，在这之后将会调用connect()方法以建立连接



2 connect()方法被调用后，Bootstrap类将会创建一个新的Channel

Bootstrap引导过程

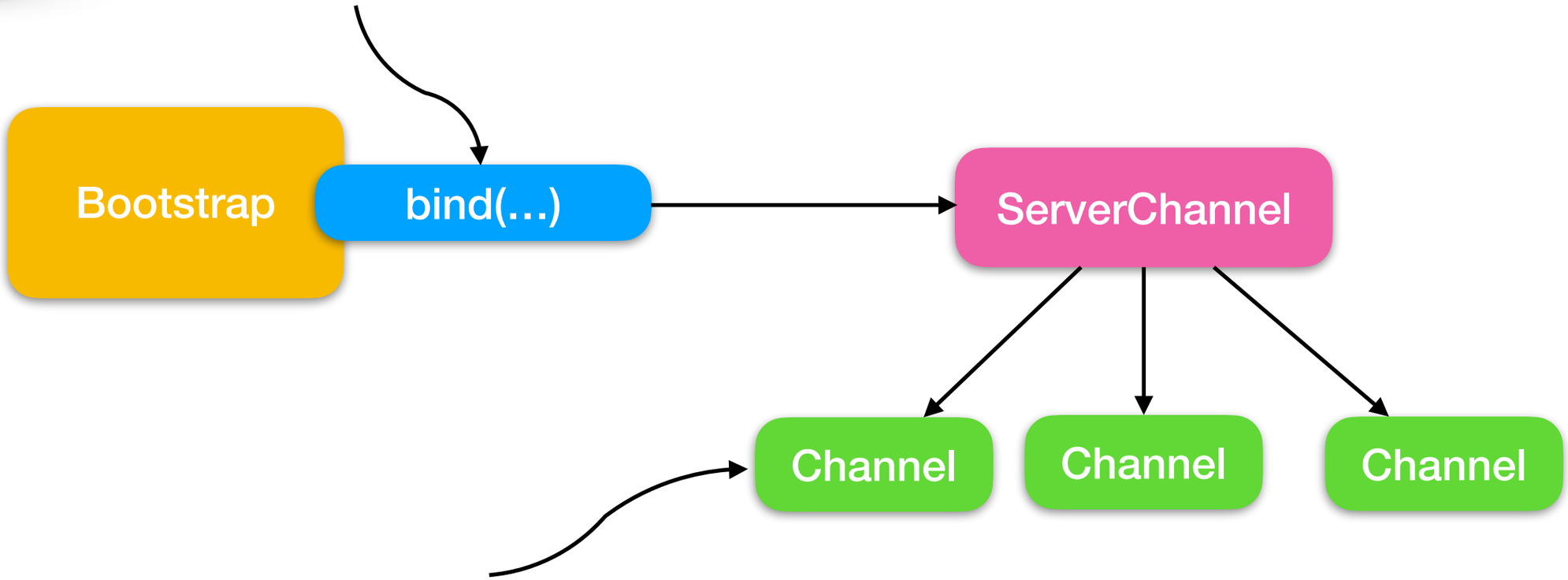
Channel

Channel是Java NIO的基础。它表示一个开放的连接，进行IO操作。基本的 I/O 操作（ bind() 、 connect() 、 read() 和 write() ）依赖于底层网络传输所提供的原语。在基于 Java 的网络编程中，其基本的构造是 class Socket 。Netty 的 Channel 接口所提供的 API，大大地降低了直接使用 Socket 类的复杂性。

EventLoop

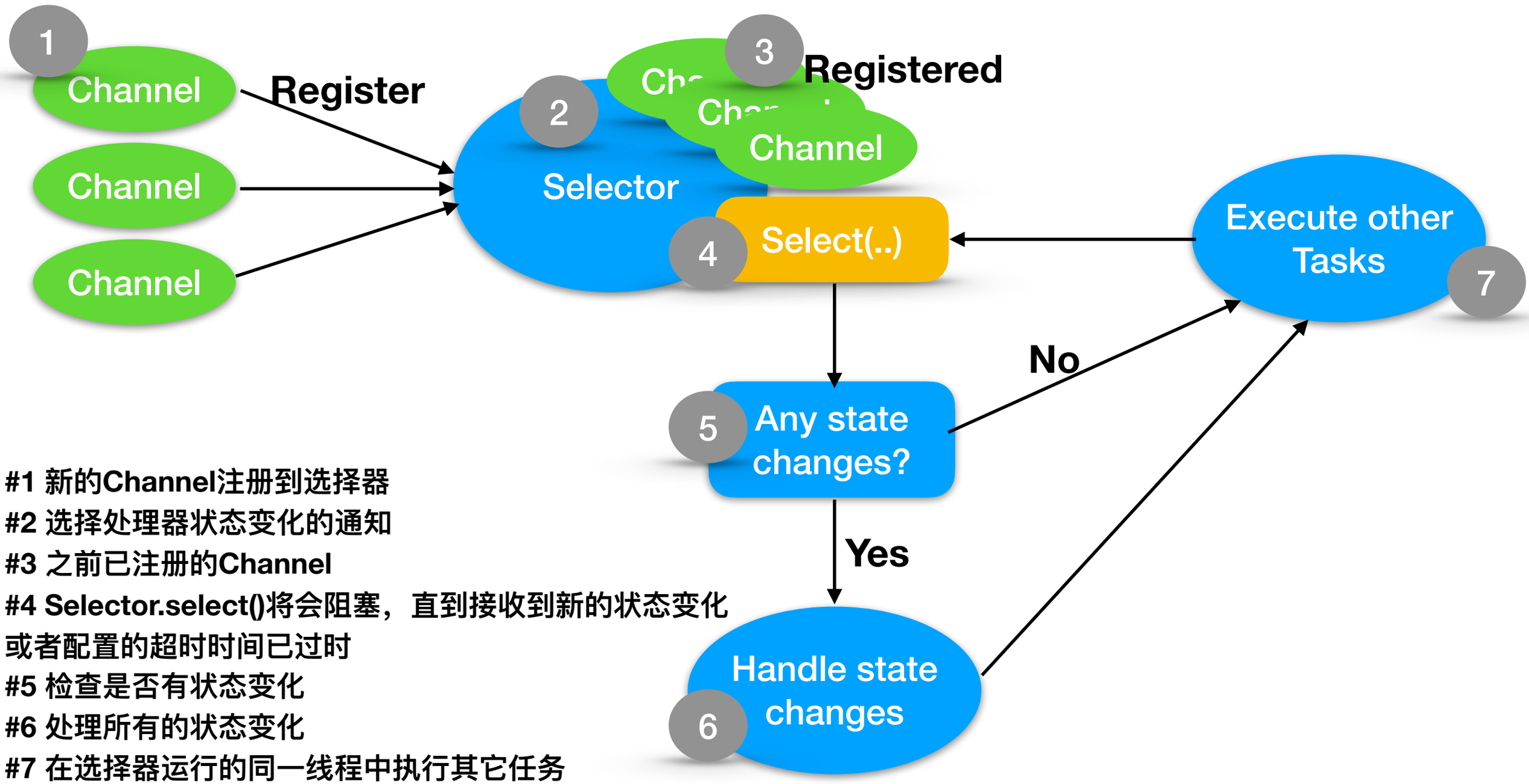
EventLoop 定义了 Netty 的核心抽象，用于处理连接的生命周期中所发生的事件。

1 当bind()方法被调用时，将会创建一个ServerChannel



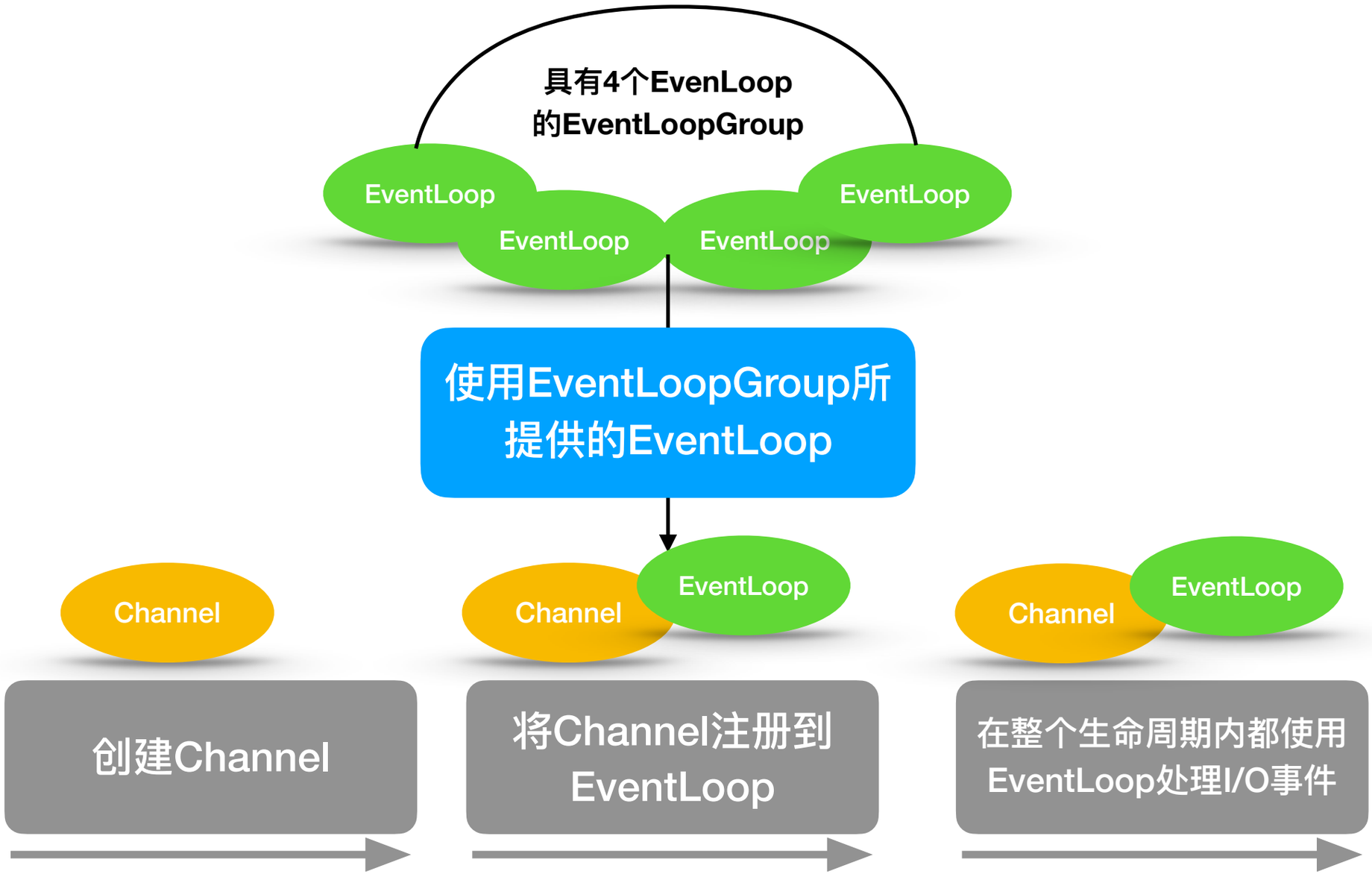
2 当连接被接受时，ServerChannel将会创建一个新的子Channel

ServerBootstrap引导过程



#1 新的Channel注册到选择器
#2 选择处理器状态变化的通知
#3 之前已注册的Channel
#4 Selector.select()将会阻塞，直到接收到新的状态变化或者配置的超时时间已过时
#5 检查是否有状态变化
#6 处理所有的状态变化
#7 在选择器运行的同一线程中执行其它任务

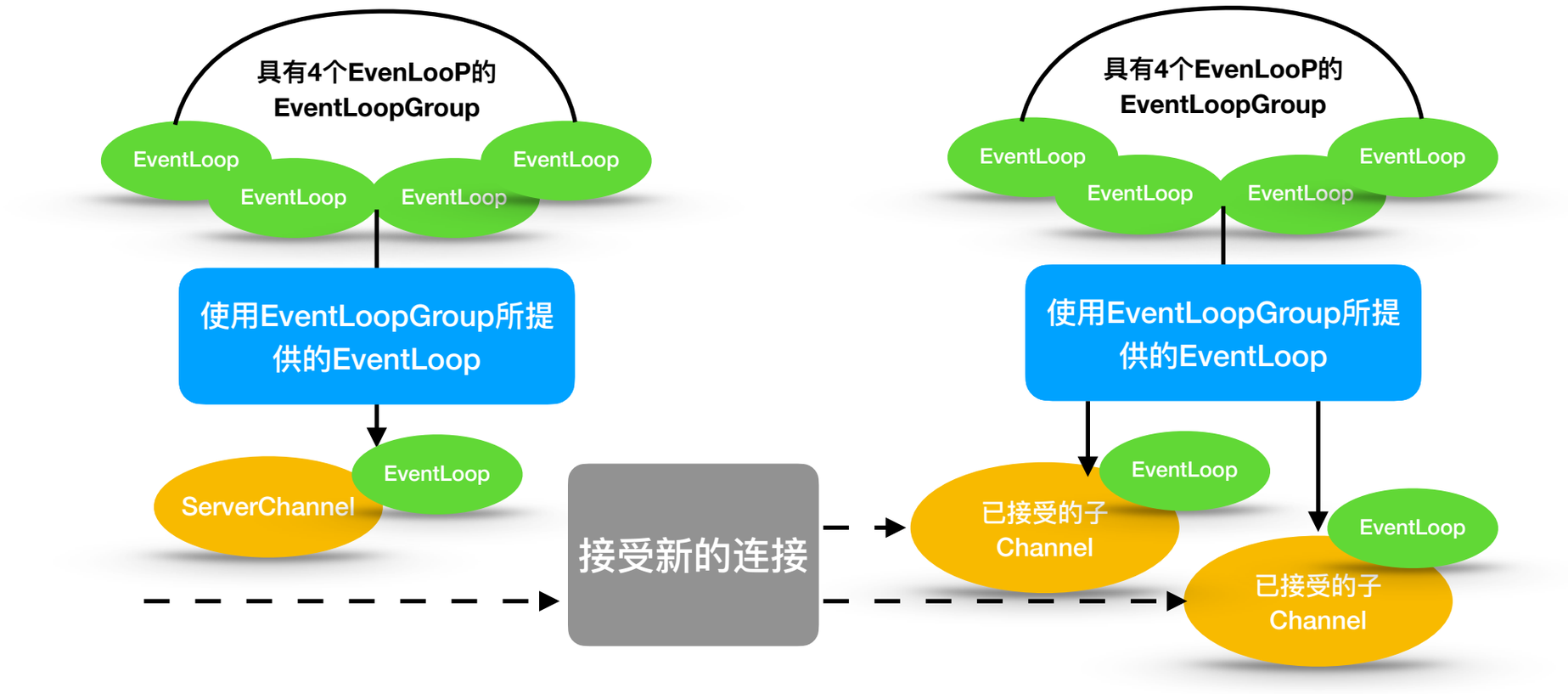
Selector处理逻辑



Channel 、 EventLoop 和 EventLoopGroup关系

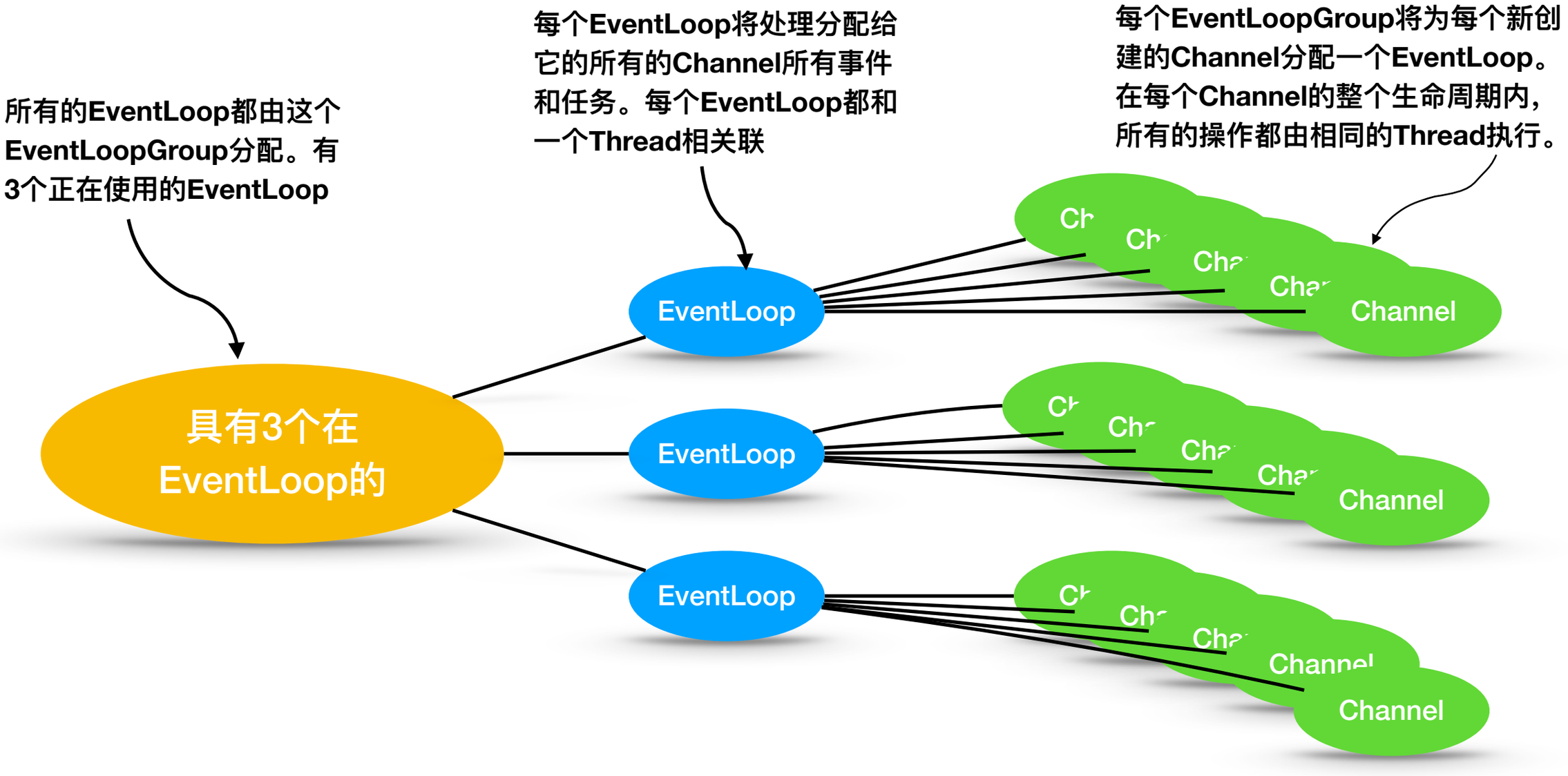
服务器需要两组不同的 Channel 。第一组将只包含一个 ServerChannel ，代表服务器自身的已绑定到某个本地端口的正在监听的套接字。

第二组将包含所有已创建的用来处理传入客户端连接（对于每个服务器已经接受的连接都有一个）的 Channel



具有2个EventLoopGroup的服务器

与 ServerChannel 相关联的 EventLoopGroup 将分配一个负责为传入连接请求创建 Channel 的 EventLoop 。一旦连接被接受，第二个 EventLoopGroup 就会给它的 Channel 分配一个 EventLoop 。



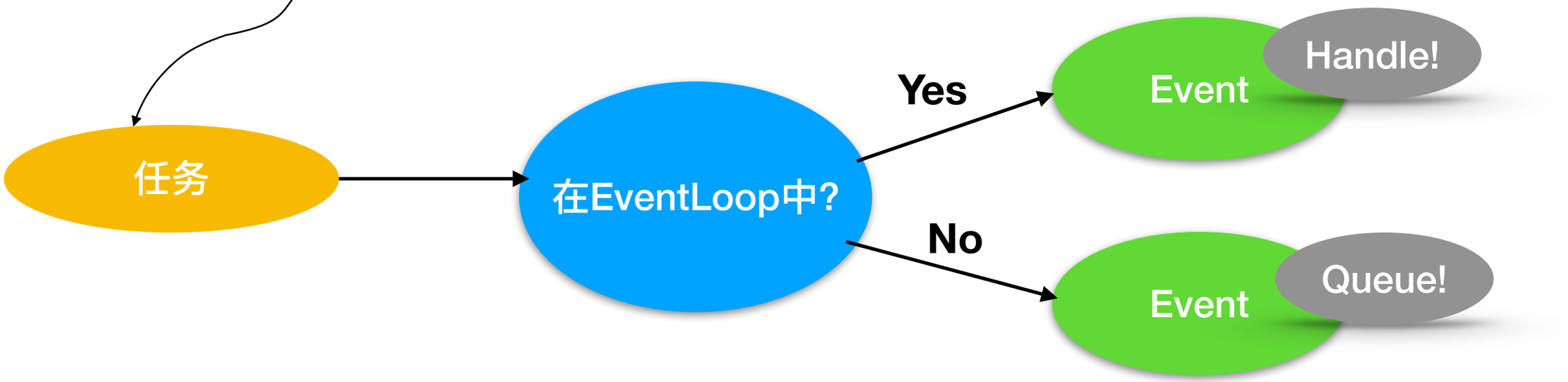
用于非阻塞传输（如 NIO 和 AIO）的EventLoop分配方式

EventLoop 执行逻辑

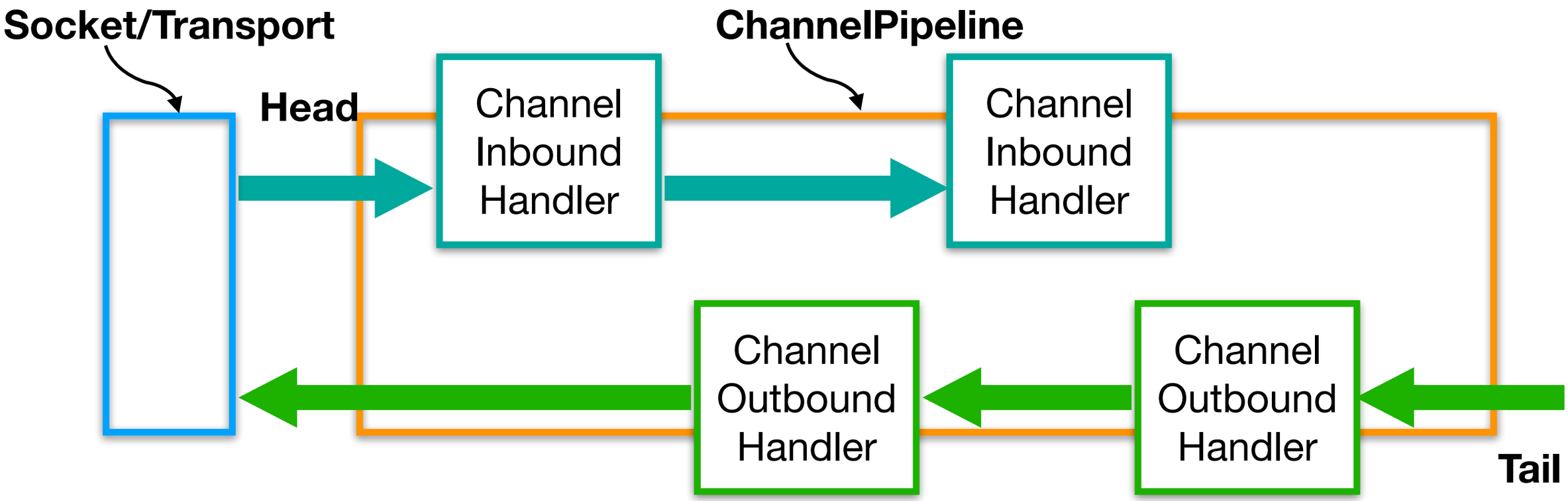
1 将要在EventLoop执行的任务

2 在把任务传递给execute方法之后，执行检查以确定当前调用线程是否就是分配给EventLoop的那个线程

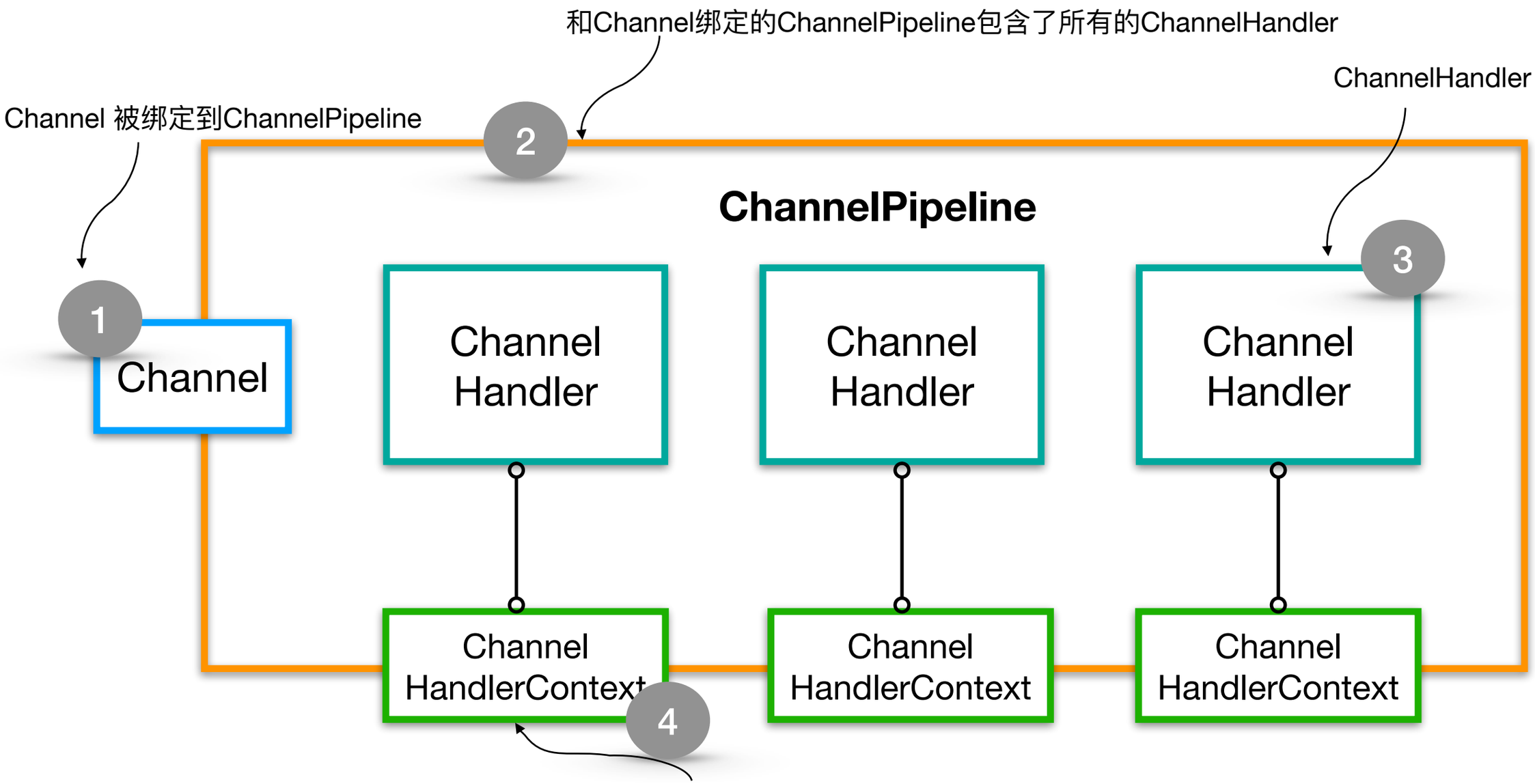
3 如果是相同的线程，则在EventLoop中直接执行



4 如果是不相同的线程，则将任务放入列队中，EventLoop下一次处理它的事件时执行

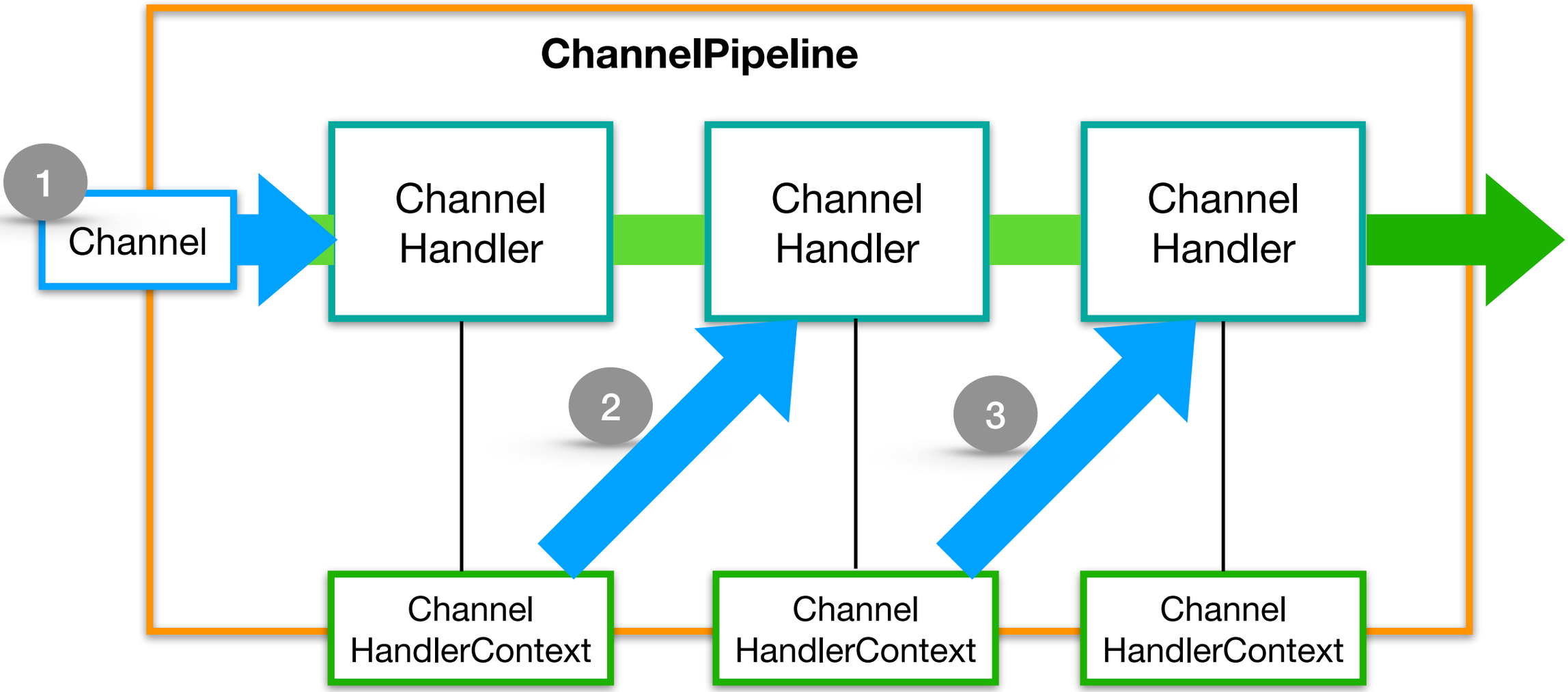


ChannelPipeline

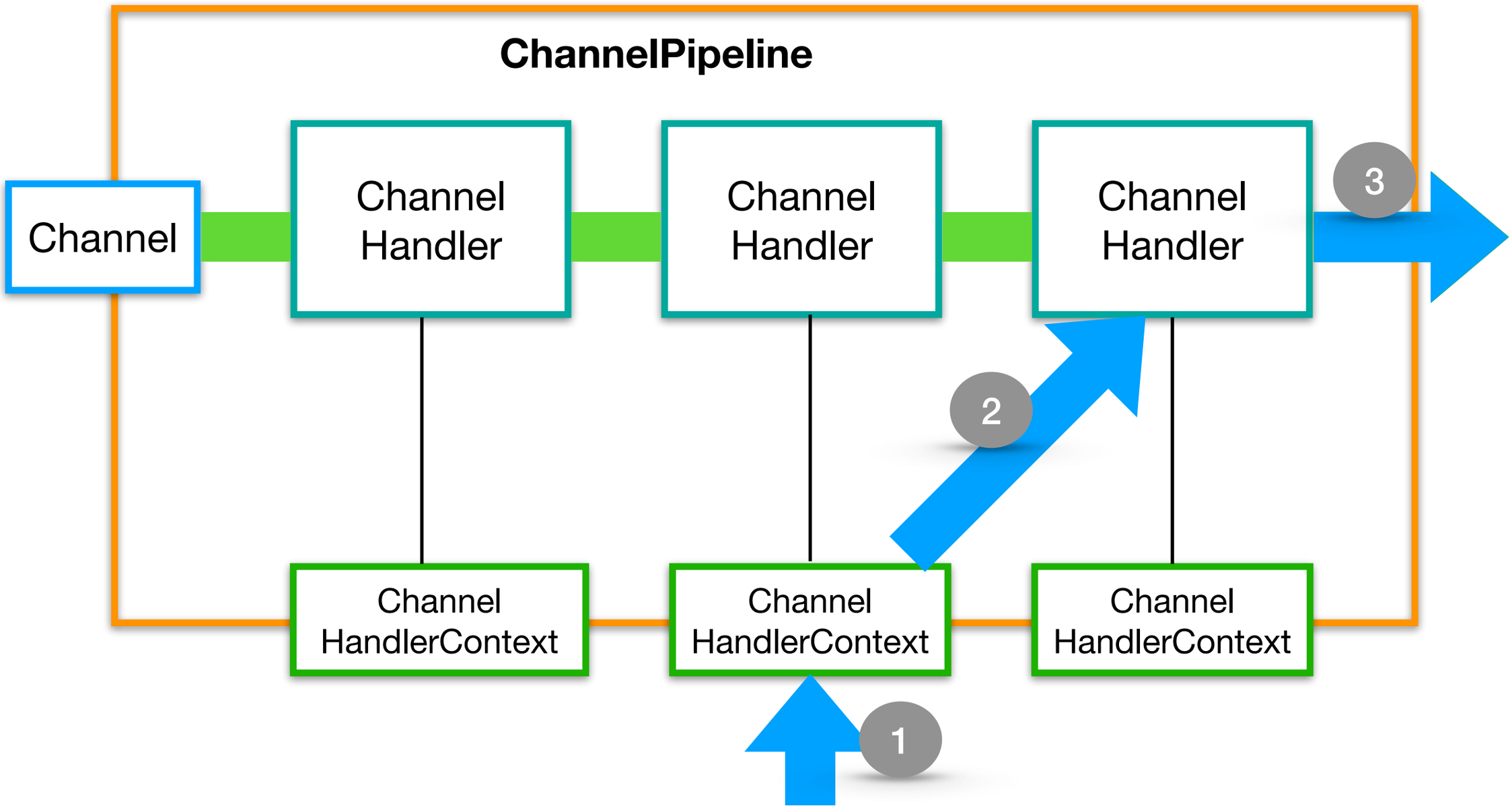


Channel、ChannelPipeline、ChannelHandler 以及 ChannelHandlerContext 之间的关系

#1 事件被传递给了ChannelPipeline中的第一个ChannelHandler
#2 通过使用与之相关联的ChannelHandlerContext，ChannelHandler将事件传递给了ChannelPipleline中的下一个ChannelHandler
#3 通过使用与之相关联的ChannelHandlerContext，ChannelHandler将事件传递给了ChannelPipleline中的下一个ChannelHandler

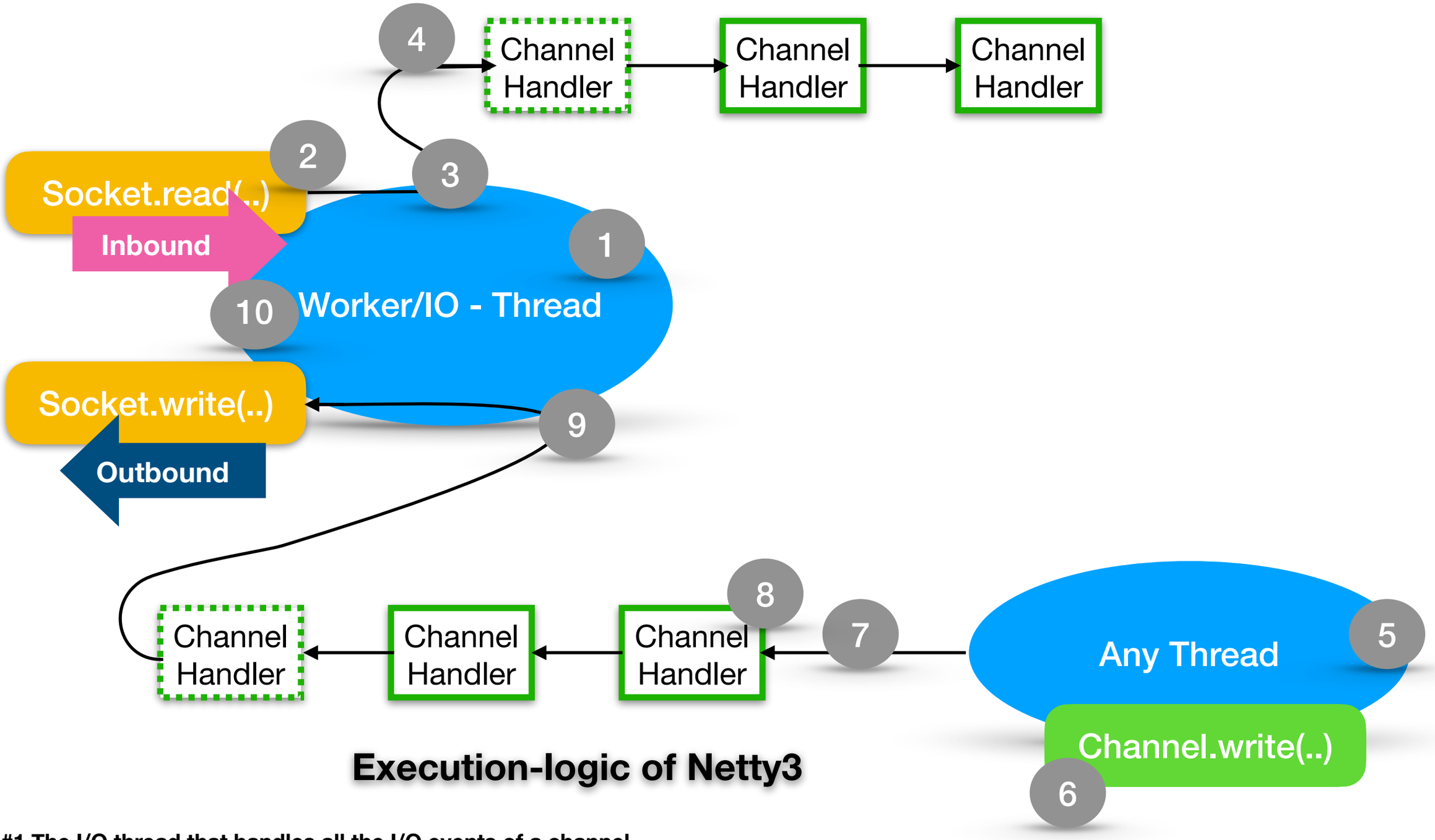


通过Channel或ChannelPipeline进行事件传播



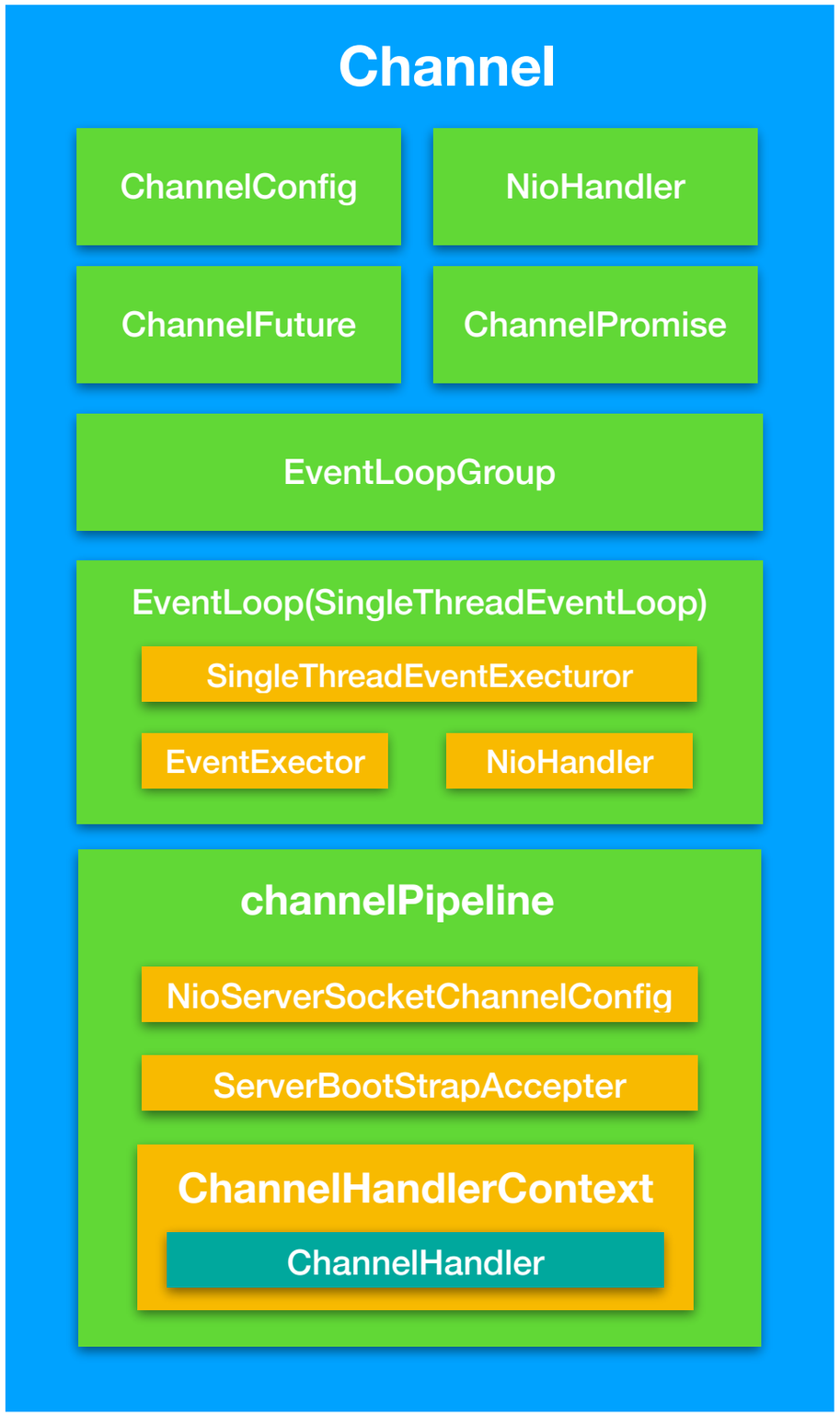
通过ChannelHandlerContext触发的操作的事件流

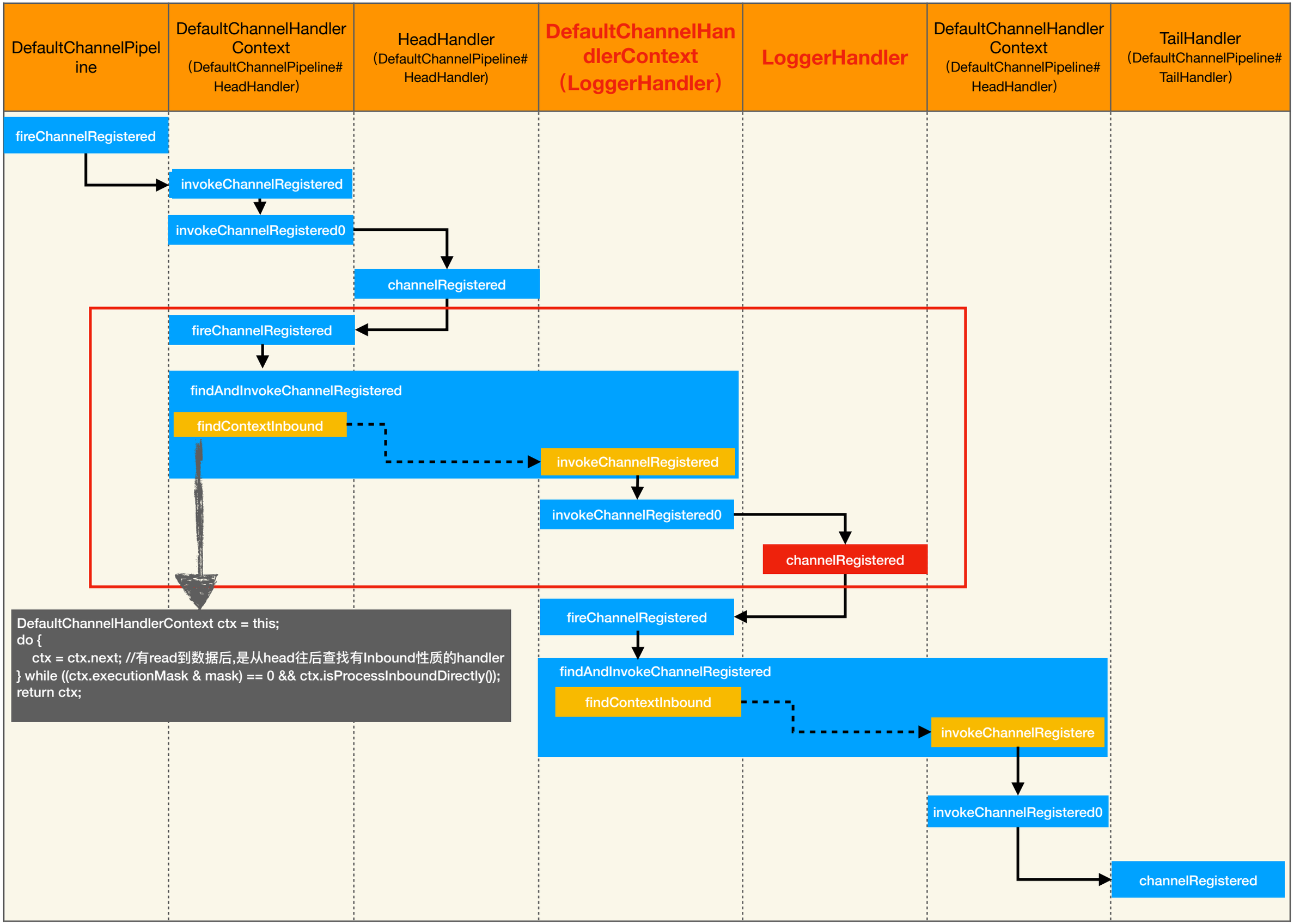
#1 Event passed to specific ChannelHandler usingChannelHandlerContext
#2 Event gets passed
#3 Move out of ChannelPipeline as no ChannelHandlers remain



Execution-logic of Netty3

#1 The I/O thread that handles all the I/O events of a channel
#2 Data gets ready from the socket
#3 Gets processed in the worker, which is bound to the I/O thread
#4 The data / event is passed through all the ChannelHandlers of the Channel . This still happens within the IO thread of the Channel.
#5 Any thread from which an outbound operation is triggered. This may be the IO / thread or any other thread.
#6 Something is passed to the Channel.write(..)
#7 The write operation will generate an event that will get passed to the ChannelHandlers of the channel
#8 The event gets passed through all the ChannelHandlers of the channel This happens in the same thread as from which the write operation was triggerd.
#09 Once processing of the even via the ChannelHandlers is done, it will hand over the event to the worker thread
#10 The data is finally written to the remote peer by the worker thread


Channel
ChannelConfig
NioHandler
ChannelFuture
ChannelPromise
EventLoopGroup
EventLoop(SingleThreadEventLoop)
SingleThreadEventExecturor
EventExector
NioHandler
channelPipeline
NioServerSocketChannelConfig
ServerBootStrapAccepter
ChannelHandlerContext
ChannelHandler



Pipleline中Handler的调用链