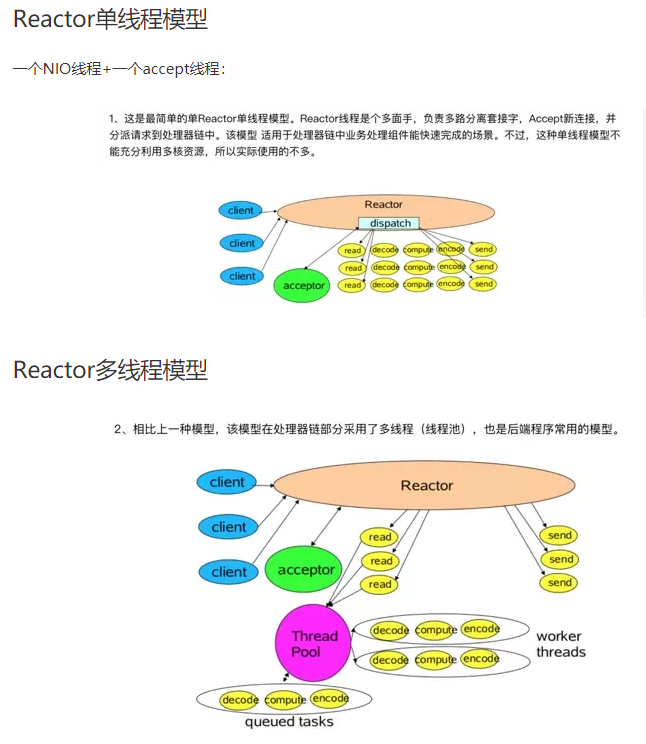
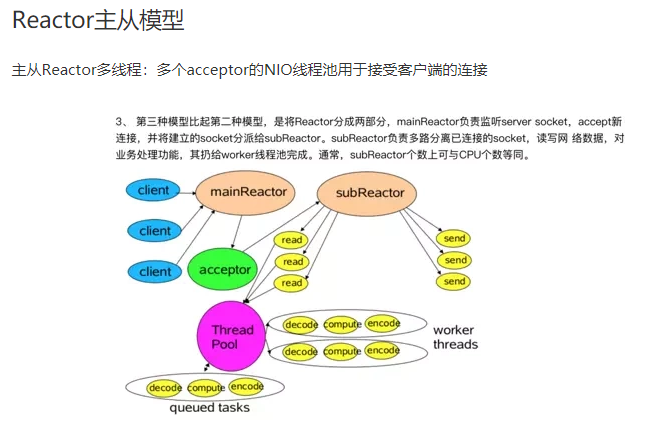
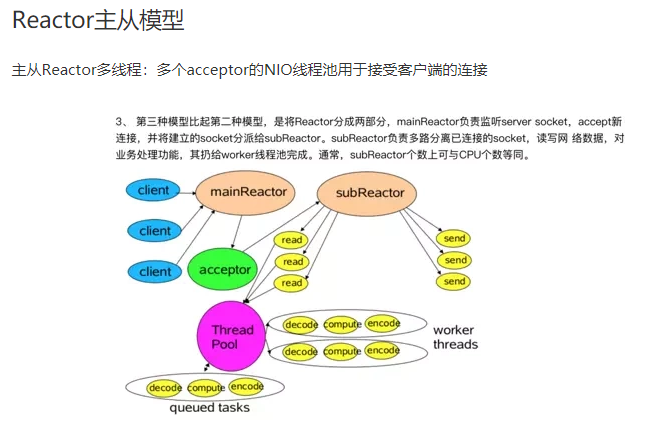
BIO和NIO：

传统的BIO模式下，从头所有进程都是阻塞的，这些进程就干等着，占用系统的资源。

NIO使用事件机制，用一个线程把Accept，读写操作，请求处理的逻辑都干了，如果没有事情做，它也不会死循环，而是将线程休眠起来，直到下一个事件来了再继续干活

Reactor模式





Netty建立在NIO基础之上，Netty在NIO之上又提供了更高层次的抽象

TCP粘包/拆包：

尽管应用层使用了Netty，但是对于操作系统来说，只认TCP协议，尽管我们的应用层是按照ByteBuf为单位来发送数据，server也是按照Bytebuf读取，但是到了底层操作系统仍然是按照字节流发送数据，因此数据到了服务端是按照字节流读入，到了Netty应用后重新组装成ByteBuf，所以可能不对等，所以需要根据自定义协议来组装包。这个过程在服务端称为拆包，在客户端称为粘包。



Netty的零拷贝

在发送数据的时候，传统的实现方式需要四次数据拷贝和四次上下文切换：

1. 数据从磁盘读取到内核的read buffer
2. 数据从内核缓冲区拷贝到用户缓冲区
3. 数据从用户缓冲区拷贝到内核的socket buffer
4. 数据从内核的socket buffer拷贝到网卡接口的缓冲区

零拷贝：避免了上次两次多余拷贝

1. 调用transferTo,数据从文件由DMA引擎拷贝到内核readbuffer
2. 接着调用DMA从内核read buffer将数据拷贝到网卡接口buffer

Netty中的零拷贝：

Bytebuffer bytebuffer使用对外内存（DirectMemory）直接进行Socket读写。原因：如果使用传统的堆内存进行socket读写，JVM会将堆内存buffer拷贝一份到直接内存中然后在写入socket，多了一次缓冲区的内存拷贝。DirectMemory中可以直接通过DMA发送到网卡接口。

Composite Buffers：传统的ByteBuffer如果需要将两个ByteBuffer中的数据组合到一起，我们需要首先创建一个数组，再将两个数组中的数据拷贝到新的数组中，使用CompositeBuffer并没有以组合多个Buffer，而是保存了它们的引用，从而避免了数据的拷贝

Netty封装了NIO JDK 处理了NIO的selector空转轮询bug 通过计数，rebuild selector解决