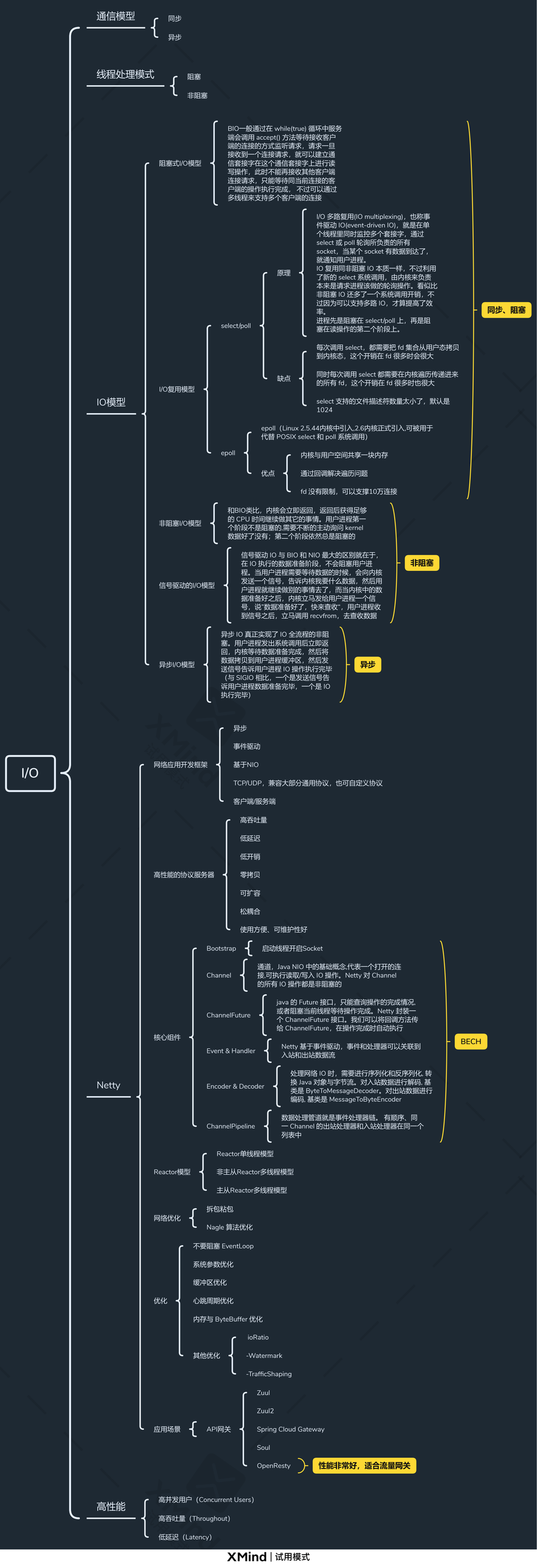
I/O
通信模型
同步
异步
线程处理模式
阻塞
非阻塞
IO模型
阻塞式I/O模型
BIO一般通过在 while(true) 循环中服务端会调用 accept() 方法等待接收客户端的连接的方式监听请求，请求一旦接收到一个连接请求，就可以建立通信套接字在这个通信套接字上进行读写操作，此时不能再接收其他客户端连接请求，只能等待同当前连接的客户端的操作执行完成， 不过可以通过多线程来支持多个客户端的连接
I/O复用模型
select/poll
原理
I/O 多路复用(IO multiplexing)，也称事件驱动 IO(event-driven IO)，就是在单个线程里同时监控多个套接字，通过 select 或 poll 轮询所负责的所有 socket，当某个 socket 有数据到达了，就通知用户进程。
IO 复用同非阻塞 IO 本质一样，不过利用了新的 select 系统调用，由内核来负责本来是请求进程该做的轮询操作。看似比非阻塞 IO 还多了一个系统调用开销，不过因为可以支持多路 IO，才算提高了效率。
进程先是阻塞在 select/poll 上，再是阻塞在读操作的第二个阶段上。
缺点
每次调用 select，都需要把 fd 集合从用户态拷贝到内核态，这个开销在 fd 很多时会很大
同时每次调用 select 都需要在内核遍历传递进来的所有 fd，这个开销在 fd 很多时也很大
select 支持的文件描述符数量太小了，默认是 1024
epoll
epoll（Linux 2.5.44内核中引入.2.6内核正式引入.可被用于代替 POSIX select 和 poll 系统调用）
优点
内核与用户空间共享一块内存
通过回调解决遍历问题
fd 没有限制，可以支撑10万连接
同步、阻塞
非阻塞I/O模型
和BIO类比，内核会立即返回，返回后获得足够的 CPU 时间继续做其它的事情，用户进程第一个阶段不是阻塞的需要不断的主动询问 kernel 数据好了没有；第二个阶段依然总是阻塞的
信号驱动的I/O模型
信号驱动 IO 与 BIO 和 NIO 最大的区别就在于，在 IO 执行的数据准备阶段，不会阻塞用户进程。当用户进程需要等待数据的时候，会向内核发送一个信号，告诉内核我要什么数据，然后用户进程就继续做别的事情去了，而当内核中的数据准备好之后，内核立马发给用户进程一个信号，说"数据准备好了，快来查收"，用户进程收到信号之后，立马调用 recvfrom，去查收数据
非阻塞
异步I/O模型
异步 IO 真正实现了 IO 全流程的非阻塞。用户进程发出系统调用后立即返回，内核等待数据准备完成，然后将数据拷贝到用户进程缓冲区，然后发送信号告诉用户进程 IO 操作执行完毕（与 SIGIO 相比，一个是发送信号告诉用户进程数据准备完毕，一个是 IO 执行完毕）
异步
Netty
网络应用开发框架
异步
事件驱动
基于NIO
TCP/UDP，兼容大部分通用协议，也可自定义协议
客户端/服务端
高性能的协议服务器
高吞吐量
低延迟
低开销
零拷贝
可扩容
松耦合
使用方便、可维护性好
核心组件
Bootstrap
启动线程开启Socket
Channel
通道，Java NIO 中的基础概念.代表一个打开的连接.可执行读取/写入 IO 操作。Netty 对 Channel 的所有 IO 操作都是非阻塞的
ChannelFuture
java 的 Future 接口，只能查询操作的完成情况，或者阻塞当前线程等待操作完成。Netty 封装一个 ChannelFuture 接口。我们可以将回调方法传给 ChannelFuture，在操作完成时自动执行
Event & Handler
Netty 基于事件驱动，事件和处理器可以关联到入站和出站数据流
Encoder & Decoder
处理网络 IO 时，需要进行序列化和反序列化. 转换 Java 对象与字节流。对入站数据进行解码. 基类是 ByteToMessageDecoder。对出站数据进行编码. 基类是 MessageToByteEncoder
ChannelPipeline
数据处理管道就是事件处理器链。 有顺序、同一 Channel 的出站处理器和入站处理器在同一个列表中
BECH
Reactor模型
Reactor单线程模型
非主从Reactor多线程模型
主从Reactor多线程模型
网络优化
拆包粘包
Nagle 算法优化
优化
不要阻塞 EventLoop
系统参数优化
缓冲区优化
心跳周期优化
内存与 ByteBuffer 优化
其他优化
ioRatio
-Watermark
-TrafficShaping
应用场景
API网关
Zuul
Zuul2
Spring Cloud Gateway
Soul
OpenResty
性能非常好，适合流量网关
高性能
高并发用户（Concurrent Users）
高吞吐量（Throughout）
低延迟（Latency）