Linux的IO多路复用方式

一般数据的访问，都是先等待数据准备，然后把数据拷贝到内核中，再从内核拷贝到进程中。一般阻塞io会从等待数据准备开始阻塞，直到数据准备好仍然阻塞，直到准备好后再完成io才会被唤醒。nio的话在数据准备阶段不会阻塞，而是让线程轮询查看数据是不是准备好了，真正开始io时才会阻塞。

io多路复用使用场景：

- 客户处理多个描述字或多个套接口（比较少）
- tcp服务器又要监听套接口，又要处理连接上的套接口
- 服务器同时处理tcp与udp时
- 服务器处理多个服务或多个协议

select与poll

两者原理一样，只是对描述符的集合描述方式不同

select函数运行进程指示内核等待多个事件的任何一个发生，并在发生后经历一段指定时间后唤醒，int select(描述字个数,读描述符集合,写描述符集合,发生异常描述符集合,超时时间结构体)，读写异常描述符用来指定对什么条件感兴趣，timeout结构体用来检测超时，传入null表示一直等，结构体内时间为0表示轮询（直接返回）。用户进程调用select就会被block直到有描述符就绪。内核会监视所有select负责的socket。

执行流程：连接套接字，绑定端口，监听，情况描述符集合，设置新的描述符集合（并绑定回调函数，把描述符集合从用户态拷到内核），开始select（遍历所有文件描述符，并执行其对应的poll方法，poll方法是多态的，会根据实际情况调不同的tcp_poll，udp_poll等），tcp_poll调用_pollwait，就是调用注册的回调函数，它的工作是把当前线程挂到设备的等待队列上（不代表已经睡眠），在设备收到消息或填完数据（资源准备好）后，唤醒设备等待队列上睡眠的进程，当前线程便被唤醒（并**把描述符从内核态拷到用户态**）。poll方法会返回一个mask掩码表示读写操作是否准备好，若没有任何一个准备好，便会让当前调用select函数的线程睡眠，在超过规定时间没有被唤醒后会被唤醒，重新遍历描述符。

缺点：由于每次执行select时描述符都会从用户态拷贝内核态，并且还需要遍历所有描述符，开销较大，且文件描述符数量较少，只有1024个

epoll

epoll克服了上面的三个缺点，并进行改进，它有三个接口

- epoll_create：创建一个epoll句柄，
- epoll_ctl：注册要监听的事件类型
- epoll_wait：等待事件的产生。

对于poll的三个缺点：epoll_ctl在注册新的事件到epoll句柄中是，就把所有描述符拷贝到内核，不会再执行（select或者说epoll_wait时多次拷贝），只会拷贝一次；每个描述符都有指定回调函数，当资源就位时，就唤醒等待队列上的等待者，然后把就绪的描述符挂到就绪链表上，epoll_wait的工作就只需要检测就绪链表是不是空的；epoll描述符数量这个限制，数目只和内存有关。

且epoll_wait每次醒了以后只需要遍历就绪链表，不需要遍历所有文件描述符，效率更高。

Netty

`Zero-copy`

通常的Zero-copy指在操作数据时, 不需要将数据 buffer 从一个内存区域拷贝到另一个内存区域. 因为少了一次内存的拷贝, 因此 CPU 的效率就得到的提升。

OS 层面上的 `Zero-copy` 通常指避免在 `用户态(User-space)` 与 `内核态(Kernel-space)` 之间来回拷贝数据。

Netty 中的 `Zero-copy` 与上面我们所提到到 OS 层面上的 `Zero-copy` 不太一样, Netty的 `Zero-coyp` 完全是在用户态(Java层面)的, 它的 `Zero-copy` 的更多的是偏向于优化数据操作，比如使用CompositeByteBuf将多个ByteBuf进行逻辑上的拼接，而不需要通过数据拷贝变成物理上的拼接。使用wrapChannelbuffer进行数组或者channelbuffer的包装成为新的channelbuffer（可以用wrap操作拼接多个buf，还可以用slice把buf切片成多段），避免了数据拷贝

buffer

Netty中的buffer和传统NIO的buffer区别来说，NIO的buffer仅仅是传输上的buffer，netty中的buffer是传输buffer和抽象后的逻辑buffer的结合。

netty中的buffer的最上层接口是ChannelBuffer，其并不是封装NIO的ByteBuffer而来，它全部重新实现了一遍，抛弃了NIO的buffer的几大参数与使用filp，clear切换读写，它使用两个索引readerIndex和writerIndex指向读和写的地方，所以ChannelBuffer的读写可以同时进行，不需要切换。

字节序Endianness：创建时可以传入参数指定存储是大端还是小端

buffer实现类：

- HeapChannelBuffer：最常用的实现类，具备前面提到的双索引等基本功能。
- DynamicChannelBuffer：有一个`ensureWritableBytes`方法，类似ArrayList，容量不够时可以自动扩容
- CompositeChannelBuffer：零拷贝的一种实现机制，内部维护了一个ChannelBuffer数组，并使用int数组记录每个buf的使用位置，传入的index会被映射到buf数组中某个buf上，将多个buf变成逻辑上的连接，而不需要拷贝数据变为数据上的连接。
- ByteBufferBackedChannelBuffer：封装了NIO的ByteBuffer，用于实现堆外内存的buf。
- WrappedChannelBuffer：这个接口有多个实现类，但都是对前面的几个ChannelBuffer进行包装装饰，避免数据的拷贝，实现0拷贝机制。比如wrap操作拼接多个buf，还可以用slice把buf切片成多段（SlicedChannelBuffer b=ChannelBuffer.slice()），共用某个ChannelBuffer的存储，但使用不同的index（DuplicatedChannelBuffer b = ChannelBuffer.duplicate()），使得ChannelBuffer只读。

Reactor

在NIO中，线程是不需要因为等待可读可写可连接而阻塞的，这是一种很浪费cpu资源的行为，尽管使用多线程切换来弥补了消耗，但线程的切换又成为了必不可少的消耗，在Reactor中，当线程期望某个事件时，会在中间人出进行注册，并绑定事件达成时的回调函数，这样线程就不需要阻塞等待，可以继续做自己的事了，中间人调用方法获取准备好的事件（如selector调select方法），这时调用者会阻塞直到有事件发生，中间人得到事件的准备情况（对应的通道是否可写可读），当准备好后就执行对应线程绑定的回调函数，这个中间人角色就是Reactor，NIO中的selector。

多线程下的Reactor：采用主从多线程的Reactor，有主线程池与子线程池，主线程池（Reactor线程）中选择一个线程（称为Acceptor线程）绑定要监听的端口，并接收客户端发起的连接，接收成功后，再从主线程池中找其他的线程负责这个连接后续的接入认证等工作。接着与客户端成功建立链路后，把链路摘除然后注册到子线程池（NIO线程）的，让子线程池上的线程来负责后续的IO操作。

Netty的Reactor：Netty服务端使用了“多Reactor线程模式”，

- **当服务器程序启动时，会配置ChannelPipeline**，ChannelPipeline中是一个ChannelHandler链，所有的事件发生时都会触发Channelhandler中的某个方法，这个事件会在ChannelPipeline中的ChannelHandler链里传播。然后，从bossGroup事件循环池中获取一个NioEventLoop来现实服务端程序绑定本地端口的操作，将对应的ServerSocketChannel注册到该NioEventLoop中的Selector上，并注册ACCEPT事件为ServerSocketChannel所感兴趣的事件。
- NioEventLoop事件循环启动，此时**开始监听客户端的连接请求**。
- 当有客户端向服务器端**发起连接请求**时，NioEventLoop的事件循环监听到该ACCEPT事件，Netty底层会接收这个连接，通过accept()方法得到与这个客户端的连接(SocketChannel)，然后**触发ChannelRead事件**(即，ChannelHandler中的channelRead方法会得到回调)，该事件会在ChannelPipeline中的ChannelHandler链中执行、传播。
- ServerBootstrapAcceptor的readChannel方法会该SocketChannel(客户端的连接)**注册到workerGroup(NioEventLoopGroup)中的某个NioEventLoop的Selector上**，并注册READ事件为SocketChannel所感兴趣的事件。启动SocketChannel所在NioEventLoop的事件循环，接下来就可以开始客户端和服务器端的通信了。

分布式理论

基本理论

CAP：

- Consistency：一致性，只分布式系统中多个副本直接保持一致的特性
- Availability：可用性，系统提供的服务是一直可用的状态
- Partition tolerance：分区容错性，分布式系统区分于单机系统的基本特性

分区容错性是分布式系统必不可少的特性，不然就是传统的单机系统了，但在保证P的同时，C与A是不能完全保证的，一般由于主机宕机，网络原因等多方因素，C与A只能保证一个。一般要求强一致性的时候是在涉及到钱财的方面，许多传统的数据库分布式事务都是属于这种模式。而有些要保证高可用性的系统就会放弃强一致性。比如nosql。

BASE理论：Basically Available（基本可用，可用允许时间稍长一点），Soft State（软状态，允许存在中间状态，不是强一致性），Eventially Consistent（最终一致性，数据最终要保证是一致的）

一致性问题

分布式进行事务提交时，往往是多个数据库一起进行修改的，比如买东西的时候，设计到账户，订单，库存等多方面的修改，而这些其实是应该在一个事务中全部进行提交的。但分布式中这些数据的储存一般是在多个库中，所以就需要让这多个库的事务统一的执行后，统一提交或统一回滚。

2PC

将分布式事务的完成分为两个阶段，角色上除了需要进行事务的对象，还需要一个协调者。

- 第一个阶段（准备阶段）：协调者首先会询问所有分布式节点，当前是否准备好了提交事务，这个阶段，所有分布式节点就会完成事务逻辑，并回复给协调者自己是要提交事务还是回滚事务。
- 第二个阶段（提交阶段）：协调者收到所有分布式节点的回复后，如果发现所有节点都表示可用提交事务，协调者就让所有节点执行提交的操作，反之一旦有一个事务声明要进行回滚，那么所有节点都会被要求回滚

2pc的问题：如果在第二阶段，节点执行提交（或回滚）后和协调者一起宕机了（节点还没有把状态传播出去，其他节点不知道当前是什么情况）。这个时候一般会有watchDog临时接替协调者的工作直到协调者重新上线，而这过程中发生的事情和节点状态都会被节点等记录，watchDog会从第一阶段重新询问未下线的节点事务执行情况并重新决定事务的提交还是回滚。但在重新提交（或回滚）时，宕机的那个分布式节点上线了，它也需要执行新的提交（或回滚）。由于这个分布式节点已经执行了提交（或回滚）。此时如果两次执行的是一样的，那么数据是一致的，但执行不一样的话，就导致了数据的不一致了。

为了避免上诉的数据不一致问题，就有了3PC。

3PC

- 第一阶段（canCommit 确认可提交）：协调者提前预约分布式节点是否可以准备执行事务，直到收到所有节点的确认进入下一个阶段
- 第二阶段（preCommit 执行事务）：通知所有节点当前是执行提交还是回滚，让所有节点都可知
- 第三阶段（doCommit 提交或回滚）：执行提交或者回滚，若有宕机出现，watchDog通过询问其中一个未宕机节点当前执行情况就能确定当前事务该做什么

项目搭建流程

- 首先请求全部过**负载均衡**，使用负载均衡算法分流（随机，轮询，ip hash，优先级，任务少的优先），一致性哈希：在0到2的32次方的圆上放置服务器节点，每个请求hash以后去大于它的第一个有服务器的节点上，这样若有服务器宕机或者新加服务器都十分的方便。
- **请求过滤**，比如用布隆过滤器过滤掉无效请求
- **消息队列**来削峰填谷
- **缓存**，如redis，可用于存储热点数据（注意缓存一致性，和集群中的热点数据分片），也可以用来做分布式锁
- **后台**，代码结构设计上面注意静动态数据分开，静态数据可以存缓存，动态数据要直接过数据库，不要过于的追求设计模式，没有最好只有最合适。当查询数据不多时，不放将范围操作都返回所有数据，在内存中进行范围划分，用专门的sql固然更好，但这样可以使得代码中工作量与接口更少，维护成本下降。
- **中间件**
  - DTS定时任务调度
  - RPC
- **数据库**
  - 维度进行划分，如动态静态分表，静态过缓存，最后在后台进行拼装（后台最好各种service继承同一个service，在这个service中统一处理，统一用泛型标识，传参指定类型，最后返回整合好的集合）
  - 分库分表，纵向分表（由于列的热点值进行拆分，分为多个表）；横向分表（横向分表后到多个库，用主键进行hash决定访问那个库）
- **前后分离**，CDN（将数据部署在离客户端最近的地方，加速访问效率）

安全

XSS攻击

- 反射性：返回脚本让前端执行
- 储存性：提交了非法脚本被储存后，再次访问这个数据后脚本在前端被执行，比如留言评论等。

- 基于DOM：前端执行时没有过滤，各种脚本被直接转义执行。

防范：

- 对用户的输入参数进行过滤（ script关键字过滤）
- 对输出进行转义

CSRF攻击

用户正常访问一个网站，这个网站对应转账操作，用户访问后产生了对应的cookie，服务器也对用户有了session，用户在未退出的情况下（ cooike未销毁）访问了骇客的网站。

- 若转账为get操作，骇客直接修改转账目的用户为自己，去访问转账的网站进行转账
- 若转账是post操作，也可以通过隐藏表单直接提交进行转账

防范操作（主要在后端）

- 尽量使用POST
- 转账使用验证码
- 验证referer（查看发送者的域名地址来进行是否可以接收）（但可能有篡改referer）
- 产生随机的token储存在服务器或缓存，访问时带token访问，后端进行验证，token每次都会变化