Java超神之路-IO_NIO_NETTY

> author：编程界的小学生
>
> date：2021/03/03

一、IO/NIO

1、BufferedOutputStream和FileOutputStream谁快?

buffer快，因为FileOutputStream的是每次witre()都会执行syscall调用内核写入pagecache，而BufferedOutputStream的write()则是每次都写入jvm的8kb大小的缓冲区，等8kb满了后再执行syscall调用系统内核写入pagecache，不会每次都进行系统调用。

2、什么是BIO和NIO？有什么区别?

- BIO

  同步阻塞IO。服务端当accept一个客户端请求后，在recv或send调用阻塞时，将无法继续accept其他请求，比如等上一个阻塞的accept的recv或send完才能继续处理下一个，也就是无法处理并发！也无法在阻塞下面处理读写请求，因为还在阻塞中，下面的读写程序走不了，解决方式是服务端可以采取多线程，当accept一个请求后，开启线程进行recv，这样就能完成并发处理，但若并发过大的话会造成多线程的频繁切换，也会造成很大的开销。

- NIO

同步非阻塞IO。服务端当accept一个客户端请求后，不会阻塞住，而是会加入fds集合中，每次轮询一遍fds集合recv（非阻塞）数据，没有数据的话立即返回-1，完全非阻塞，所以可以同时接收多个accept请求，也可以在accept下面继续读写请求，因为accept不会阻塞。不用开辟新线程，自己单线程就能办。但是每次轮询所有fd（包括没有发生读写事件的fd）都会产生一次系统调用，会很浪费性能的。这时候就引入了IO多路复用技术。

3、什么是IO多路复用?

服务端采取单线程通过select/poll/epoll系统调用函数获取fd列表，遍历有事件的fd进行accept/recv/send处理，简单来说IO多路复用就是将NIO的多次系统调用转换为一次系统调用，具体的轮询操作交给内核去处理，而不是每轮询一次就进行一次系统调用。

4、nginx/redis所使用的IO模型是什么?

epoll模型。

5、select、poll、epoll之间的区别是什么?

5.1、select

5.1.1、执行流程

- select模型每次都直接将fds全部拷贝到内核态。因为内核态比用户态快很多。
- select函数会将有数据的那个fd标记一下，代表这个fd有数据。然后遍历fds，找到有数据的fd进行读取等处理，处理完后这个fd不能复用，每次都要重新创建新的fds集合且将用户态的fds拷贝到内核态。

5.1.2、缺点

- 底层采取bitmap存储，默认最大支持1024个fd
- fd不可重用，每次内核态都给原有的fd标记了，导致每次都需要重新创建一个新的fds出来。
- 每次创建完新的fds都需要从用户态拷贝到内核态，拷贝也需要开销。
- 每次都for循环遍历fds，然后逐个对比看哪个fd是被标记过的（代表有数据），所以需要O(n)的时间复杂度。

5.2、poll

5.2.1、执行流程

poll主要引入了pollfd结构体，里面包含了一个revents字段，这个字段用来进行标记是否有数据用的，引入结构体主要解决了select的fd不能复用的问题。

- 有数据的fd会将这个结构体里的revents字段 设置为POLLIN，代表有数据。
- 然后for循环判断fds里这个结构体，看revents字段是不是POLLIN，是的话再将这个结构体的这个字段恢复为默认值，然后取出数据，进行逻辑处理。恢复为默认值的用途是为了这个结构体继续复用，而不是像select那样每次都需要重新创建新的fds。

5.2.2、解决了select的哪些问题?

- 采取的是数组存储，而不是bitmap。解决了1024长度限制问题。
- 采取结构体每次修改revents字段，而不破坏fd本身，所以可重用，不需要每次都创建新的fd。

5.2.3、缺点

- 每次创建完新的fds都需要从用户态拷贝到内核态，拷贝也需要开销。
- 每次都for循环遍历fds，然后逐个对比看哪个结构体的revents字段是被标记过的（代表有数据），所以需要O(n)的时间复杂度。

5.3、epoll

5.3.1、执行流程

- epoll将fd放到了红黑树里，且他采取了mmap共享内存的概念，可以忽略用户态拷贝到内核态这阶段带来的开销。
- epoll标记是否有数据的方式是重排，比如五个fd，1，2，3，4，5，1，3，5这三个fd有数据，那么他会重排序变成1，3，5，2，4
- 每一次标记fd有数据的时候都会回调epoll_wait，epoll_wait执行完后会返回有几个fd有数据，那么之前select、poll采取的遍历fds然后逐个对比的操作就变成了系统直接回调有数据的fd方式，直接从O(n)变成了O(1)时间复杂度。

5.3.2、解决了哪些问题?

解决了poll模型留下的最后两个问题，也就是解决了select模型的全部问题。

5.3.3、缺点

只有Linux支持epoll模型，windows不支持。

5.4、总结

	select	**poll**	**epoll**
操作方式	遍历	遍历	回调
底层实现	bitmap	数组	红黑树
IO效率	每次都遍历fds逐个判断找标记的fd，O(n)时间复杂度	每次都遍历fds逐个判断结构体revents字段标记的fd，O(n)时间复杂度	事件通知的方式，每当fd就绪，系统注册回调函数就会被调用，将就绪fd发那个到list里。O(1)时间复杂度
最大连接数	1024	无上限	无上限
fd拷贝	每次调用select，都需要把fds从用户态拷贝到内核态	每次调用poll。都需要把fds从用户态拷贝到内核态	fd首次调用epoll_ctl拷贝到内核并保存，之后每次调用epoll_wait无需再次拷贝

6、epoll水平触发LT与边缘触发ET的区别是什么?

epoll有EPOLLLT和EPOLLET两种触发模式，LT是默认的模式水平触发，ET是边缘触发。边缘触发比水平触发效率更高。因为边缘触发不会让同一个文件描述符多次被处理,比如有些文件描述符已经不需要再读写了,但是在水平触发下每次都会返回,而边缘触发只会返回一次。

- LT模式下，只要这个fd还有数据可读，每次 epoll_wait都会返回它的事件，提醒用户程序去操作
- ET模式下，它只会提示一次，直到下次再有数据流入之前都不会再提示了，无论fd中是否还有数据可读。所以在ET模式下，read一个fd的时候一定要把它的buffer读完，或者遇到EAGAIN错误

7、Redis若采取select、poll、epoll分别是什么效果?

比如三个redis-cli，假设2个redis-cli写入命令

- select：那么select模型是轮询这三个redis-cli的fd，看哪个fd有消息，有的话读取处理消息。当他下次再写命令的时候还需要重新创建fd，然后复制到内核态然后再遍历全部。
- poll：那么poll模型是轮询这三个redis-cli的fd，看哪个fd有消息，有的话读取处理消息。下次再写入的时候还是遍历全局fd，看哪个fd有消息进行处理。省去了每次都创建新的fd且复制的过程。
- epoll：epoll就不轮询了，有消息进来后你通知我，我去处理你的消息，那些没消息的fd我不管。而且复制到内核态的过程我采取牛逼的技术让开销达到最小的极致。

二、Netty

1、Netty的实现原理是什么?

未完待续~！目前没得时间来研究，工作中目前也用不到，暂且搁置哈~