链滴

I/O

作者：AshShawn
原文链接：https://ld246.com/article/1649339054232
来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20220117.jpg?imageView2/1/w/960/h/540/interlace/1/q/100">
<h2 id="1-网络包接收过程">1.网络包接收过程</h2>
<img src="https://ld246.com/images/img-loading.svg" alt="2bb5f351c7880793bae1657
d4c30835.png" data-src="https://b3logfile.com/file/2022/04/2bb5f351c7880793bae16576d4
30835-8b7fdb0a.png?imageView2/2/interlace/1/format/jpg">
<h3 id="1-内核">1.内核</h3>

数据到达网卡,网卡通过 DMA 搬运数据到内核 DMA 缓冲区
网卡向 cpu 发起硬中断,cpu 回调中断程序,创建 sk_buffer
网卡中断程序向内核发起软中断,通知内核网络数据到达
内核网络协议栈处理数据头,找到对应 socket,拷贝数据到对应的 socket 缓冲区

<h3 id="2-应用">2.应用</h3>

程序调用内核的 read 函数读取 socket 缓冲区,
如果 socket 缓冲区没有数据,则程序阻塞(BIO)
当 socket 接受缓冲区有数据时,将内核空间的数据拷贝至用户空间,系统调用 read 返回,

<h3 id="3-性能开销">3.性能开销</h3>

应用程序通过 <code>系统调用</code> 从 <code>用户态</code> 转为 <code>内核态</c
de> 的开销以及系统调用 <code>返回</code> 时从 <code>内核态</code> 转为 <code>用户
</code> 的开销。
网络数据从 <code>内核空间</code> 通过 <code>CPU拷贝</code> 到 <code>用户空间</
ode> 的开销。
内核线程 <code>ksoftirqd</code> 响应 <code>软中断</code> 的开销,主要为数据包解析
及数据拷贝到 socket 缓冲区。
<code>CPU</code> 响应 <code>硬中断</code> 的开销,主要为 DMA 缓冲区数据拷贝到 sk
buffer。
<code>DMA拷贝</code> 网络数据包到 <code>内存</code> 中的开销。

<h2 id="2-网络包发送流程">2.网络包发送流程</h2>
<img src="https://ld246.com/images/img-loading.svg" alt="b807da4b27460d516ee585
91f8a71ad.png" data-src="https://b3logfile.com/file/2022/04/b807da4b27460d516ee585391
8a71ad-42f62026.png?imageView2/2/interlace/1/format/jpg">
<h3 id="1-内核-">1.内核</h3>

程序调用系统 send 函数,用户态切换成内核态,根据 fd 找到 socket,发送数据封装到 <code>msg
dr</code> 结构体
创建内核数据结构 sk_buffer,拷贝 msghdr 数据到 sk_buffer,添加到 socket 发送队列队尾
发送时拷贝 sk_buffer(tcp 防丢包),进行数据封装填充
循环从发送队列中去除 sk_buffer,依次调用网卡发送数据
发送完毕向 cpu 触发硬中断,硬中断中触发软中断释放 sk_buffer 副本以及网卡发送队列

<h3 id="2-性能开销">2.性能开销</h3>

send 函数用户态切换成内核态, 发送完毕从内核态切换成用户态开销
网卡发送完毕后出发 cpu 硬中断,以及在硬中断中出发软中断
内存拷贝开销

发送数据拷贝至 sk_buffer
sk_buffer 的副本拷贝
发送数据过大时,会进行分片拷贝成多个小的 sk_buffer

<h2 id="3--阻塞-非阻塞-同步-异步">3. 阻塞、非阻塞、同步、异步</h2>
同步与异步关注的是通信机制, 简单来说就是调用方与服务方之间的协同机制
<blockquote>
同步: 调用方 A 发起调用后,服务方 B 在处理完调用方法之前不会返回,即 AB 是串行执行的
</blockquote>
<blockquote>
异步: 调用方 A 发起调用后,服务方 B 会立刻返回,并在处理完调用方法后通过回调等方式通知 A，即 AB 步调可以不一致
</blockquote>
不同于同步与异步, 阻塞与非阻塞关注的是一方的状态
<blockquote>
阻塞: 阻塞当前线程(进入循环或者挂起线程),使线程不能处理其他事
</blockquote>
<blockquote>
非阻塞: 不阻塞当前线程,线程仍然可以处理其他事
</blockquote>

整个流程分为两个阶段:

数据准备阶段: 数据从网卡拷贝到内存再到 socket 数据缓冲区
数据拷贝阶段: 内核空间(socket 数据缓冲区)拷贝到用户空间

<h3 id="1-阻塞">1.阻塞</h3>

第一阶段和第二阶段都会 <code>阻塞等待</code>
<h3 id="2-非阻塞">2.非阻塞</h3>

第一阶段 <code>不会等待</code>，但是在第二阶段还是会 <code>等待</code>。
<h3 id="3-同步">3.同步</h3>

<code>同步模式</code> 在数据准备好后，是由 <code>用户线程</code> 的 <code>内核</code> 来执行 <code>第二阶段</code>。所以应用程序会在第二阶段发生 <code>阻塞</code>，直到数据从 <code>内核空间</code> 拷贝到 <code>用户空间</code>，系统调用才会返回。
Linux 下的 <code>epoll</code> 和 Mac 下的 <code>kqueue</code> 都属于 <code>同步IO</code>。
<h3 id="4-异步">4.异步</h3>

<code>异步模式</code> 下是由 <code>内核</code> 来执行第二阶段的数据拷贝操作，当 code>内核</code> 执行完第二阶段，会通知用户线程 IO 操作已经完成，并将数据回调给用户线程所以在 <code>异步模式</code> 下 <code>数据准备阶段</code> 和 <code>数据拷贝阶段</c

de> 均是由 <code>内核</code> 来完成，不会对应用程序造成任何阻塞。
<h2 id="4-五种IO模型">4.五种 IO 模型</h2>
<h3 id="4-1-阻塞IO-BIO-">4.1 阻塞 IO(BIO)</h3>
<img src="https://ld246.com/images/img-loading.svg" alt="1b89fc3f60514395c6b89b8b0093e0f.png" data-src="https://b3logfile.com/file/2022/04/1b89fc3f60514395c6b89b84b003e0f-c11126f4.png?imageView2/2/interlace/1/format/jpg">
<h5 id="阻塞读">阻塞读</h5>
调用系统 <code>read</code> 函数,用户线程从用户态切换为内核态,查看 socket 缓冲区是否数据

socket 缓冲区有数据, 将数据从内核空间拷贝到用户空间,系统 IO 调用返回
socket 缓冲区无数据,用户线程让出 CPU,进入 <code>阻塞状态</code>。当数据到达 <code>ocket</code> 接收缓冲区后，内核唤醒 <code>阻塞状态</code> 中的用户线程进入 <code>就状态</code>，随后经过 CPU 的调度获取到 <code>CPU quota</code> 进入 <code>运行状态<code>，将内核空间的数据拷贝到用户空间，随后系统调用返回。

<h5 id="阻塞写">阻塞写</h5>
调用系统 <code>send</code> 函数,用户线程从用户态切换为内核态,将发送数据从用户空间贝到内核空间中的 <code>Socket</code> 发送缓冲区中。

<code>Socket</code> 发送缓冲区能够容纳下发送数据时，用户线程会将全部的发送数据写入 <code>Socket</code> 缓冲区,执行发送流程后返回
<code>Socket</code> 发送缓冲区空间不够，无法容纳下全部发送数据时，用户线程让出 CPU 进入 <code>阻塞状态</code>，直到 <code>Socket</code> 发送缓冲区能够容纳下全部发送数时，内核唤醒用户线程，执行后续发送流程。

<h5 id="阻塞IO模型">阻塞 IO 模型</h5>

每个请求单独线程处理
并发量增大则服务端瞬间会创建大量线程,造成资源浪费
创建好线程后,若长期处于空闲时间,则线程们一直阻塞
线程切换开销大

<h5 id="适用场景">适用场景</h5>

链接少
并发低
重型操作

<h3 id="4-2-非阻塞IO-NIO-">4.2 非阻塞 IO(NIO)</h3>
<img src="https://ld246.com/images/img-loading.svg" alt="4fe3ca17ee37ca2b23840032cf04d0f.png" data-src="https://b3logfile.com/file/2022/04/4fe3ca17ee37ca2b238400325cf040f-c8859f78.png?imageView2/2/interlace/1/format/jpg">
用尽量少的线程去处理更多的连接
<h5 id="非阻塞读">非阻塞读</h5>
调用系统 <code>read</code> 函数,用户线程从用户态切换为内核态,查看 socket 缓冲区是否数据

socket 缓冲区有数据, 将数据从内核空间拷贝到用户空间,系统 IO 调用返回
socket 缓冲区无数据,系统调用立马返回,线程 <code>不会阻塞</code>,也 <code>不让出cpu/code>,而是会继续 <code>轮训</code> 直到 <code>Socket</code> 接收缓冲区中有数据为止

<h5 id="非阻塞写">非阻塞写</h5>

调用系统 <code>send</code> 函数,用户线程从用户态切换为内核态,将发送数据从用户空间贝到内核空间中的 <code>Socket</code> 发送缓冲区中。

<code>Socket</code> 发送缓冲区能够容纳下发送数据时，用户线程会将全部的发送数据写入 <code>Socket</code> 缓冲区,执行发送流程后返回
<code>Socket</code> 发送缓冲区空间不够，无法容纳下全部发送数据时，<code>能写多写多少</code>，写不下了，就立即返回。并将写入到发送缓冲区的字节数返回给应用程序，方便用线程不断的 <code>轮训</code> 尝试将 <code>剩下的数据</code> 写入发送缓冲区中。

<h5 id="非阻塞IO模型">非阻塞 IO 模型</h5>
<img src="https://ld246.com/images/img-loading.svg" alt="a1f0be37b1e7dcb677a851d784e0ae0.png" data-src="https://b3logfile.com/file/2022/04/a1f0be37b1e7dcb677a851db78e0ae0-a59682d1.png?imageView2/2/interlace/1/format/jpg">
利用一个线程或者很少的线程，去 <code>不断地轮询</code> 每个 <code>Socket</code> 的接收缓冲区是否有数据到达，如果没有数据，<code>不必阻塞</code> 线，而是接着去 <code>轮询</code> 下一个 <code>Socket</code> 接收缓冲区，直到轮询到数后，处理连接上的读写，或者交给业务线程池去处理，轮询线程则 <code>继续轮询</code> 其他的 <code>Socket</code> 接收缓冲区。
<h5 id="适用场景-">适用场景</h5>
在 <code>非阻塞IO模型</code> 下，需要用户线程去 <code>不断地</code> 发起 <code系统调用</code> 去轮训 <code>Socket</code> 接收缓冲区，这就需要用户线程不断地从 <cod>用户态</code> 切换到 <code>内核态</code>，<code>内核态</code> 切换到 <code>用户</code>。
单纯的 <code>非阻塞IO</code> 模型还是无法适用于高并发的场景
<h3 id="4-3-IO多路复用">4.3 IO 多路复用</h3>
<img src="https://ld246.com/images/img-loading.svg" alt="8c51b0c8322d3026fb7bc8040179ec4.png" data-src="https://b3logfile.com/file/2022/04/8c51b0c8322d3026fb7bc8074079ec4-4345b8ba.png?imageView2/2/interlace/1/format/jpg">

多路：我们的核心需求是要用尽可能少的线程来处理尽可能多的连接，这的 <code>多路</code> 指的就是我们需要处理的众多连接。
复用：核心需求要求我们使用 <code>尽可能少的线程</code>，<code尽可能少的系统开销</code> 去处理 <code>尽可能多</code> 的连接（<code>多路</code>）那么这里的 <code>复用</code> 指的就是用 <code>有限的资源</code>，比如用一个线程或者定数量的线程去处理众多连接上的读写事件。换句话说，在 <code>阻塞IO模型</code> 中一个连就需要分配一个独立的线程去专门处理这个连接上的读写，到了 <code>IO多路复用模型</code>，多个连接可以 <code>复用</code> 这一个独立的线程去处理这多个连接上的读写。

非阻塞 IO 中其实实现了多路复用,只不过是用户态去不断轮询,会导致用户态与内核态频繁切换,费资源
我们可以把频繁的轮询操作交给操作系统内核来替我们完成，这样就避免了在 <code>用户空间/code> 频繁的去使用系统调用来轮询所带来的性能开销。
<h5 id="select">select</h5>
<img src="https://ld246.com/images/img-loading.svg" alt="8f1f3c7b23a88b7758b9ebb0854ec05.png" data-src="https://b3logfile.com/file/2022/04/8f1f3c7b23a88b7758b9ebb7085ec05-23be2c9a.png?imageView2/2/interlace/1/format/jpg">
<code>select</code> 系统调用将 <code>轮询</code> 的操作交给了 <code>内核</code> 来帮助我们完成，从而避免了在 <code>用户空间</code> 不断的发起轮询所带来的的系统性能开。

首先用户线程在发起 <code>select</code> 系统调用的时候会 <code>阻塞</code> 在 <coe>select</code> 系统调用上。此时，用户线程从 <code>用户态</code> 切换到了 <code>内态</code> 完成了一次 <code>上下文切换</code>
用户线程将需要监听的 <code>Socket</code> 对应的文件描述符 <code>fd</code> 数组

过 <code>select</code> 系统调用传递给内核。此时，用户线程将 <code>用户空间</code> 中文件描述符 <code>fd</code> 数组 <code>拷贝</code> 到 <code>内核空间</code>。
当用户线程调用完 <code>select</code> 后开始进入 <code>阻塞状态</code>，<code>核</code> 开始轮询遍历 <code>fd</code> 数组，查看 <code>fd</code> 对应的 <code>Socet</code> 接收缓冲区中是否有数据到来。如果有数据到来，则将 <code>fd</code> 对应 <code BitMap</code> 的值设置为 <code>1</code>。如果没有数据到来，则保持值为 <code>0</cod >。
内核遍历一遍 <code>fd</code> 数组后，如果发现有些 <code>fd</code> 上有 IO 数据到，则将修改后的 <code>fd</code> 数组返回给用户线程。此时，会将 <code>fd</code> 数组从 code>内核空间</code> 拷贝到 <code>用户空间</code>。
当内核将修改后的 <code>fd</code> 数组返回给用户线程后，用户线程解除 <code>阻塞</c de>，由用户线程开始遍历 <code>fd</code> 数组然后找出 <code>fd</code> 数组中值为 <co e>1</code> 的 <code>Socket</code> 文件描述符。最后对这些 <code>Socket</code> 发起 统调用读取数据。
由于内核在遍历的过程中已经修改了 <code>fd</code> 数组，所以在用户线程遍历完 <code> d</code> 数组后获取到 <code>IO就绪</code> 的 <code>Socket</code> 后，就需要 <code 重置</code> fd 数组，并重新调用 <code>select</code> 传入重置后的 <code>fd</code> 数 ，让内核发起新的一轮遍历轮询

这里的文件描述符数组 其实是一个 <code>BitMap</code>，<code>Bi Map</code> 下标为 <code>文件描述符fd</code>，下标对应的值为：<code>1</code> 表示该 <code>fd</code> 上有读写事件，<code>0</code> 表示该 <code>fd</code> 上没有读写事 。
性能开销

发生 2 次上下文 <code>切换</code>
发生 2 次文件描述符集合的 <code>拷贝</code>
虽然由原来在 <code>用户空间</code> 发起轮询 <code>优化成了</code> 在 <code>内 空间</code> 发起轮询但 <code>select</code> 不会告诉用户线程到底是哪些 <code>Socket</ ode> 上发生了 <code>IO就绪</code> 事件，只是对 <code>IO就绪</code> 的 <code>Socke </code> 作了标记，用户线程依然要 <code>遍历</code> 文件描述符集合去查找具体 <code>I 就绪</code> 的 <code>Socket</code>。时间复杂度依然为 <code>O(n)</code>。
<code>内核</code> 会对原始的 <code>文件描述符集合</code> 进行修改。导致每次在用 空间重新发起 <code>select</code> 调用时，都需要对 <code>文件描述符集合</code> 进行 <c de>重置</code>。
<code>BitMap</code> 结构的文件描述符集合，长度为固定的 <code>1024</code>,所以 能监听 <code>0~1023</code> 的文件描述符。
<code>select</code> 系统调用 不是线程安全的。
并发量增大,开销线性增长,只适合 1000 个左右的并发

<h5 id="poll">poll</h5>
poll 与 select 原理之一,主要变化点为修改了 1024 个文件描述符的限制,这样就没有了最大描述 数量的限制（当然还会受到系统文件描述符限制）
<h5 id="epoll">epoll</h5>
select 和 poll 现有问题:

每次都需要全量传入 fd 集合,导致大量 fd 在用户空间与内核空间频繁复制
内核不会通知具体 <code>IO就绪</code> 的 <code>socket</code>，只是在这些 <code> O就绪</code> 的 socket 上打好标记，所以当 <code>select</code> 系统调用返回时，在 <cod >用户空间</code> 还是需要 <code>完整遍历</code> 一遍 <code>socket</code> 文件描述 集合来获取具体 <code>IO就绪</code> 的 <code>socket</code>。
在 <code>内核空间</code> 中也是通过遍历的方式来得到 <code>IO就绪</code> 的 <cod >socket</code>。
红黑树

<img src="https://ld246.com/images/img-loading.svg" alt="da90cb043b59a9d4bf06424d19c9b78.png" data-src="https://b3logfile.com/file/2022/04/da90cb043b59a9d4bf06424bd1c9b78-a80329b1.png?imageView2/2/interlace/1/format/jpg">
epoll 内部使用红黑树来保存所有监听的 socket,添加和查找复杂度为 O(logn) ,节点中每个数字为 socket 的句柄
<ol start="2">
就绪队列

<img src="https://ld246.com/images/img-loading.svg" alt="e413f9afdab692b69c8ec601e49d1a0.png" data-src="https://b3logfile.com/file/2022/04/e413f9afdab692b69c8ec601ce491a0-5e2521de.png?imageView2/2/interlace/1/format/jpg">
当 socket 从网络中获取到数据后，会发生通知给 epoll，epoll 会将当前 socket 添加到就绪队中，并且唤醒等待中的进程（也就是调用 <code>epoll_wait</code> 的进程）。
当进程被唤醒后，就会从就绪队列中，把就绪的 socket 复制到用户提供的数组中。
当 <code>epoll_wait</code> 返回后，用户就可以从 <code>events</code> 数组中获取到绪的 socket，并可对其进行读写操作。
<ol start="3">
常用方法

1）调用 epoll_create()建立一个 epoll 对象(在 epoll 文件系统中为这个句柄对象分配资源)
2）调用 epoll_ctl 向 epoll 对象中添加这 100 万个连接的套接字
3）调用 epoll_wait 收集发生的事件的连接
epoll 增加优化:

<code>epoll</code> 在内核中通过 <code>红黑树</code> 管理海量的连接，所以在调用 <ode>epoll_wait</code> 获取 <code>IO就绪</code> 的 socket 时，不需要传入监听的 socket 件描述符。从而避免了海量的文件描述符集合在 <code>用户空间</code> 和 <code>内核空间</cde> 中来回复制。

<blockquote>
<code>select，poll</code> 每次调用时都需要传递全量的文件描述符集合，导致大量频繁的贝操作。
</blockquote>

<code>epoll</code> 仅会通知 <code>IO就绪</code> 的 socket。避免了在用户空间遍历开销。

<blockquote>
<code>select，poll</code> 只会在 <code>IO就绪</code> 的 socket 上打好标记，依然全量返回，所以在用户空间还需要用户程序在一次遍历全量集合找出具体 <code>IO就绪</code> 的 socket。
</blockquote>

<code>epoll</code> 通过在 <code>socket</code> 的等待队列上注册回调函数 <code>epoll_callback</code> 通知用户程序 <code>IO就绪</code> 的 socket。避免了在内核中轮询的销。

<blockquote>
大部分情况下 <code>socket</code> 上并不总是 <code>IO活跃</code> 的，在面对海量接的情况下，<code>select，poll</code> 采用内核轮询的方式获取 <code>IO活跃</code> 的 scket，无疑是性能低下的核心原因。
</blockquote>
<h3 id="4-4-信号驱动IO">4.4 信号驱动 IO</h3>

<img src="https://ld246.com/images/img-loading.svg" alt="59005f3cc3ce8c2282b979ce0f7cc2c.png" data-src="https://b3logfile.com/file/2022/04/59005f3cc3ce8c2282b979ce30f7c2c-1170d632.png?imageView2/2/interlace/1/format/jpg">
<h3 id="4-5-异步IO-AIO-">4.5 异步 IO(AIO)</h3>
<img src="https://ld246.com/images/img-loading.svg" alt="a488022a0a9abed4d6cb472e229b654.png" data-src="https://b3logfile.com/file/2022/04/a488022a0a9abed4d6cb472ce29b654-cc42efb7.png?imageView2/2/interlace/1/format/jpg">
<h3 id="4-6-总结">4.6 总结</h3>
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https://b3logfile.com/file/2022/04/image-6ccfe1bb.png?imageView2/2/interlace/1/format/jpg"></>
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https://b3logfile.com/file/2022/04/image-35bf70ba.png?imageView2/2/interlace/1/format/jpg"></>
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https://b3logfile.com/file/2022/04/image-befe6de8.png?imageView2/2/interlace/1/format/jpg"></>
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https://b3logfile.com/file/2022/04/image-368facb0.png?imageView2/2/interlace/1/format/jpg"></>
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https://b3logfile.com/file/2022/04/image-aa0aec5f.png?imageView2/2/interlace/1/format/jpg"></>
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https://b3logfile.com/file/2022/04/image-54e319e1.png?imageView2/2/interlace/1/format/jpg"></>
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https://b3logfile.com/file/2022/04/image-39efd4e5.png?imageView2/2/interlace/1/format/jpg"></>
java 中的 NIO 其实是多路复用 IO 模型,与 linux 中的 NIO(非阻塞 IO)存在异同点
Blocking IO 主要是里面的方法操作步骤都是同步的，而 Non-Blocking IO 则不是。它是通过册事件来触发对应的 handler 来执行。比起 thread per connection 的方式，它能更好的利用资源因为回调的机制对于异步的通信方式来说减少了轮询或者其它强制同步机制的开销，效率算是比较理的
NIO 中的非阻塞主要体现在

read 和 write 都是立刻返回的,不像 bio 中可能会阻塞,因为 nio 中 read 和 write 针对的都是 buf er, bio 中则是针对流,效率较低
事件线程是阻塞的,即调用了 epoll_wait, 但是事件处理线程是非阻塞的

<h2 id="5-用户空间的IO线程模型">5.用户空间的 IO 线程模型</h2>
<h3 id="5-1-Reactor">5.1 Reactor</h3>
<code>Reactor</code> 是利用 <code>NIO</code> 对 <code>IO线程</code> 进行不的分工：

使用前边我们提到的 <code>IO多路复用模型</code> 比如 <code>select,poll,epoll,kqueue<code>,进行 IO 事件的注册和监听。
将监听到 <code>就绪的IO事件</code> 分发 <code>dispatch</code> 到各个具体的处理 <ode>Handler</code> 中进行相应的 <code>IO事件处理</code>。

通过 <code>IO多路复用技术</code> 就可以不断的监听 <code>IO事件</code>，不断的发 <code>dispatch</code>，就像一个 <code>反应堆</code> 一样，看起来像不断的产生 <coe>IO事件</code>，因此我们称这种模式为 <code>Reactor</code> 模型。
<h5 id="单Reactor单线程">单 Reactor 单线程</h5>

<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:/b3logfile.com/file/2022/04/image-c3f7ad4f.png?imageView2/2/interlace/1/format/jpg"></>

单 <code>Reactor</code> 意味着只有一个 <code>epoll</code> 对象，用来监听所有的件，比如 <code>连接事件</code>，<code>读写事件</code>。
<code>单线程</code> 意味着只有一个线程来执行 <code>epoll_wait</code> 获取 <code>O就绪</code> 的 <code>Socket</code>，然后对这些就绪的 <code>Socket</code> 执行读，以及后边的业务处理也依然是这个线程。

<h5 id="单Reactor多线程">单 Reactor 多线程</h5>
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:/b3logfile.com/file/2022/04/image-3b78d26e.png?imageView2/2/interlace/1/format/jpg">

<code>单reactor</code> 是只有一个 <code>epoll</code> 对象来监听所有的 <code>IO件</code>，一个线程来调用 <code>epoll_wait</code> 获取 <code>IO就绪</code> 的 <cod>Socket</code>
但是当 <code>IO就绪事件</code> 产生时，这些 <code>IO事件</code> 对应处理的业务 <ode>Handler</code>，我们是通过线程池来执行。这样相比 <code>单Reactor单线程</code> 型提高了执行效率，充分发挥了多核 CPU 的优势。

<h6 id="主从Reactor多线程">主从 Reactor 多线程</h6>
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:/b3logfile.com/file/2022/04/image-049cd219.png?imageView2/2/interlace/1/format/jpg">></>

我们由原来的 <code>单Reactor</code> 变为了 <code>多Reactor</code>。<code>主Rector</code> 用来优先 <code>专门</code> 做优先级最高的事情，也就是处理 <code>连接事件/code>，对应的处理 <code>Handler</code> 就是图中的 <code>acceptor</code>。
当创建好连接，建立好对应的 <code>socket</code> 后，在 <code>acceptor</code> 中要监听的 <code>read事件</code> 注册到 <code>从Reactor</code> 中，由 <code>从Reacto</code> 来监听 <code>socket</code> 上的 <code>读写</code> 事件。

<h3 id="5-2-Proactor">5.2 Proactor</h3>
等待后续更新.....
<h3 id="5-3-Reactor与Proactor对比">5.3 Reactor 与 Proactor 对比</h3>
等待后续更新....
<h3 id="5-4Netty的IO模型">5.4Netty 的 IO 模型</h3>
netty 中 IO 模型为 Reactor 模型,三种类型都涉及,单主要使用的是 <code>主从Reactor多线程型</code>
<img src="https://ld246.com/images/img-loading.svg" alt="67229e4487d82c220fa28f2af6d3d62.png" data-src="https://b3logfile.com/file/2022/04/67229e4487d82c220fa28f2a3f6dd62-5685b84e.png?imageView2/2/interlace/1/format/jpg">

<code>Reactor</code> 在 <code>netty</code> 中是以 <code>group</code> 的形式现的，<code>netty</code> 中将 <code>Reactor</code> 分为两组，一组是 <code>MainReacorGroup</code> 也就是我们在编码中常常看到的 <code>EventLoopGroup bossGroup</code> 另一组是 <code>SubReactorGroup</code> 也就是我们在编码中常常看到的 <code>EventLoop roup workerGroup</code>。
<code>MainReactorGroup</code> 中通常只有一个 <code>Reactor</code>，专门负责最重要的事情，也就是监听连接 <code>accept</code> 事件。当有连接事件产生时，在对应的处理 <code>handler acceptor</code> 中创建初始化相应的 <code>NioSocketChannel</code>（表一个 <code>Socket连接</code>）。然后以 <code>负载均衡</code> 的方式在 <code>Sub

eactorGroup</code> 中选取一个 <code>Reactor</code>，注册上去，监听 <code>Read事件<
code>。

<blockquote>
<code>MainReactorGroup</code> 中只有一个 <code>Reactor</code> 的原因是，通常
们服务端程序只会 <code>绑定监听</code> 一个端口，如果要 <code>绑定监听</code> 多个端
，就会配置多个 <code>Reactor</code>。
</blockquote>

<code>SubReactorGroup</code> 中有多个 <code>Reactor</code>，具体 <code>React
r</code> 的个数可以由系统参数 <code>-D io.netty.eventLoopThreads</code> 指定。默认的 <
ode>Reactor</code> 的个数为 <code>CPU核数 * 2</code>。<code>SubReactorGroup</co
e> 中的 <code>Reactor</code> 主要负责监听 <code>读写事件</code>，每一个 <code>Reac
or</code> 负责监听一组 <code>socket连接</code>。将全量的连接 <code>分摊</code> 在
个 <code>Reactor</code> 中。
一个 <code>Reactor</code> 分配一个 <code>IO线程</code>，这个 <code>IO线程</co
e> 负责从 <code>Reactor</code> 中获取 <code>IO就绪事件</code>，执行 <code>IO调用
取IO数据</code>，执行 <code>PipeLine</code>。

<blockquote>
<code>Socket连接</code> 在创建后就被 <code>固定的分配</code> 给一个 <code>Reac
or</code>，所以一个 <code>Socket连接</code> 也只会被一个固定的 <code>IO线程</code>
执行，每个 <code>Socket连接</code> 分配一个独立的 <code>PipeLine</code> 实例，用来
排这个 <code>Socket连接</code> 上的 <code>IO处理逻辑</code>。这种 <code>无锁串行化
/code> 的设计的目的是为了防止多线程并发执行同一个 socket 连接上的 <code>IO逻辑处理</cod
>，防止出现 <code>线程安全问题</code>。同时使系统吞吐量达到最大化
</blockquote>
<blockquote>
由于每个 <code>Reactor</code> 中只有一个 <code>IO线程</code>，这个 <code>IO线
</code> 既要执行 <code>IO活跃Socket连接</code> 对应的 <code>PipeLine</code> 中的 <
ode>ChannelHandler</code>，又要从 <code>Reactor</code> 中获取 <code>IO就绪事件</
ode>，执行 <code>IO调用</code>。所以 <code>PipeLine</code> 中 <code>ChannelHandl
r</code> 中执行的逻辑不能耗时太长，尽量将耗时的业务逻辑处理放入单独的业务线程池中处理，
则会影响其他连接的 <code>IO读写</code>，从而近一步影响整个服务程序的 <code>IO吞吐</c
de>。
</blockquote>

当 <code>IO请求</code> 在业务线程中完成相应的业务逻辑处理后，在业务线程中利用持有的
<code>ChannelHandlerContext</code> 引用将响应数据在 <code>PipeLine</code> 中反向
播，最终写回给客户端。

<blockquote>
文章参考: <a href="https://ld246.com/forward?goto=https%3A%2F%2Fmp.weixin.qq.co
%2Fs%2FYlf_ClsjIWJf8nTUQuwMoA" target="_blank" rel="nofollow ugc">https://mp.weixin.q
.com/s/Ylf_ClsjIWJf8nTUQuwMoA
</blockquote>