链滴

字节跳动在 Go 网络库上的实践

作者：panjf2000

原文链接：https://ld246.com/article/1589257501102

来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="bytedance.png" data-src="https://b3logfile.com/file/2020/05/bytedance-23ec38aa.png?imageView2/2/interlace/1/format/pg">
<blockquote>
本文选自“字节跳动基础架构实践”系列文章。
“字节跳动基础架构实践”系列文章是由字节跳动基础架构部门各技术团队及专家倾力打造的技干货内容，和大家分享团队在基础架构发展和演进过程中的实践经验与教训，与各位技术同学一起交成长。
RPC 框架作为研发体系中重要的一环，承载了几乎所有的服务流量。本文将简单介绍字节跳动自网络库 netpoll 的设计及实践；以及我们实际遇到的问题和解决思路，希望能为大家提供一些参考。
</blockquote>
<h2 id="前言">前言</h2>
字节跳动框架组主要负责公司内 RPC 框架的开发与维护。RPC 框架作为研发体系中重要的一环承载了几乎所有的服务流量。随着公司内 Go 语言使用越来越广，业务对框架的要求越来越高，而 Go 原生 net 网络库却无法提供足够的性能和控制力，如无法感知连接状态、连接数量多导致利用率低、法控制协程数量等。为了能够获取对于网络层的完全控制权，同时先于业务做一些探索并最终赋能业，框架组推出了全新的基于 epoll 的自研网络库 —— netpoll，并基于其之上开发了字节内新一代 Goang 框架 KiteX。
由于 epoll 原理已有较多文章描述，本文将仅简单介绍 netpoll 的设计；随后，我们会尝试梳理下我们基于 netpoll 所做的一些实践；最后，我们将分享一个我们遇到的问题，以及我们解决的思路同时，欢迎对于 Go 语言以及框架感兴趣的同学加入我们！
<h2 id="新型网络库设计">新型网络库设计</h2>
<h3 id="Reactor---事件监听和调度核心">Reactor - 事件监听和调度核心</h3>
netpoll 核心是 Reactor 事件监听调度器，主要功能为使用 epoll 监听连接的文件描述符（fd）通过回调机制触发连接上的 读、写、关闭 三种事件。
<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfilecom/file/2020/05/640-8b7603dd.png?imageView2/2/interlace/1/format/jpg">
<h3 id="Server---主从-Reactor-实现">Server - 主从 Reactor 实现</h3>
netpoll 将 Reactor 以 1:N 的形式组合成主从模式。

MainReactor 主要管理 Listener，负责监听端口，建立新连接；
SubReactor 负责管理 Connection，监听分配到的所有连接，并将所有触发的事件提交到协程里进行处理。
netpoll 在 I/O Task 中引入了主动的内存管理，向上层提供 NoCopy 的调用接口，由此支持 Nopy RPC。
使用协程池集中处理 I/O Task，减少 goroutine 数量和调度开销。

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfilecom/file/2020/05/6401-63750c5f.png?imageView2/2/interlace/1/format/jpg">
<h3 id="Client---共享-Reactor-能力">Client - 共享 Reactor 能力</h3>
client 端和 server 端共享 SubReactor，netpoll 同样实现了 dialer，提供创建连接的能力。client 端使用上和 net.Conn 相似，netpoll 提供了 write -> wait read callback 的底层支持。
<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfilecom/file/2020/05/6402-e135986a.png?imageView2/2/interlace/1/format/jpg">
<h2 id="NoCopy-Buffer">NoCopy Buffer</h2>
<h3 id="为什么需要-NoCopy-Buffer--">为什么需要 NoCopy Buffer ?</h3>
在上述提及的 Reactor 和 I/O Task 设计中，epoll 的触发方式会影响 I/O 和 buffer 的设计，大来说分为两种方式：

采用水平触发(LT)，则需要同步的在事件触发后主动完成 I/O，并向上层代码直接提供 buffer。</i>
采用边沿触发(ET)，可选择只管理事件通知(如 go net 设计)，由上层代码完成 I/O 并管理 buffe。

两种方式各有优缺，netpoll 采用前者策略，水平触发时效性更好，容错率高，主动 I/O 可以集内存使用和管理，提供 nocopy 操作并减少 GC。事实上一些热门开源网络库也是采用方式一的设计如 easygo、evio、gnet 等。
但使用 LT 也带来另一个问题，即底层主动 I/O 和上层代码并发操作 buffer，引入额外的并发开。比如：I/O 读数据写 buffer 和上层代码读 buffer 存在并发读写，反之亦然。为了保证数据正确性同时不引入锁竞争，现有的开源网络库通常采取 同步处理 buffer(easygo, evio) 或者将 buffer 再 co y 一份提供给上层代码(gnet) 等方式，均不适合业务处理或存在 copy 开销。
另一方面，常见的 bytes、bufio、ringbuffer 等 buffer 库，均存在 growth 需要 copy 原数组据，以及只能扩容无法缩容，占用大量内存等问题。因此我们希望引入一种新的 Buffer 形式，一举决上述两方面的问题。
<h3 id="NoCopy-Buffer-设计和优势">NoCopy Buffer 设计和优势</h3>
NoCopy Buffer 基于链表数组实现，如下图所示，我们将 []byte 数组抽象为 block，并以链表接的形式将 block 组合为 NoCopy Buffer，同时引入了引用计数、nocopy API 和对象池。

NoCopy Buffer 相比常见的 bytes、bufio、ringbuffer 等有以下优势：

读写并行无锁，支持 nocopy 地流式读写

读写分别操作头尾指针，相互不干扰。

高效扩缩容

扩容阶段，直接在尾指针后添加新的 block 即可，无需 copy 原数组。
缩容阶段，头指针会直接释放使用完毕的 block 节点，完成缩容。每个 block 都有独立的引用计，当释放的 block 不再有引用时，主动回收 block 节点。

灵活切片和拼接 buffer (链表特性)

支持任意读取分段(nocopy)，上层代码可以 nocopy 地并行处理数据流分段，无需关心生命周，通过引用计数 GC。
支持任意拼接(nocopy)，写 buffer 支持通过 block 拼接到尾指针后的形式，无需 copy，保证据只写一次。

NoCopy Buffer 池化，减少 GC

将每个 []byte 数组视为 block 节点，构建对象池维护空闲 block，由此复用 block，减少内存占和 GC。

基于该 NoCopy Buffer，我们实现了 NoCopy Thrift，使得编解码过程内存零分配零拷贝。</p

<h2 id="连接多路复用">连接多路复用</h2>
RPC 调用通常采用短连接或者长连接池的形式，一次调用绑定一个连接，那么当上下游规模很大情况下，网络中存在的连接数以 MxN 的速度扩张，带来巨大的调度压力和计算开销，给服务治理造困难。因此，我们希望引入一种 "在单一长连接上并行处理调用" 的形式，来减少网络中的连接数，种方案即称为 "连接多路复用"。
当前业界也存在一些开源的连接多路复用方案，掣肘于代码层面的束缚，这些方案均需要 copy b ffer 来实现数据分包和合并，导致实际性能并不理想。而上述 NoCopy Buffer 基于其灵活切片和拼

的特性，很好的支持了 nocopy 的数据分包和合并，使得实现高性能连接多路复用方案成为可能。</
>
基于 netpoll 的连接多路复用设计如下图所示，我们将 NoCopy Buffer(及其分片) 抽象为虚拟
接，使得上层代码保持同 net.Conn 相同的调用体验。与此同时，在底层代码上通过协议分包将真实
接上的数据灵活的分配到虚拟连接上；或通过协议编码合并发送虚拟连接数据。
<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/file/2020/05/6403-cf2751ee.png?imageView2/2/interlace/1/format/jpg">
连接多路复用方案包含以下核心要素：

虚拟连接

实质上是 NoCopy Buffer，目的是替换真正的连接，规避内存 copy。
上层的业务逻辑/编解码 均在虚拟连接上完成，上层逻辑可以异步独立并行执行。

Shared map

引入分片锁来减少锁力度。
在调用端使用 sequence id 来标记请求，并使用分片锁存储 id 对应的回调。
在接收响应数据后，根据 sequence id 来找到对应回调并执行。

协议分包和编码

如何识别完整的请求响应数据包是连接多路复用方案可行的关键，因此需要引入协议。
这里采用 thrift header protocol 协议，通过消息头判断数据包完整性，通过 sequence id 标记
求和响应的对应关系。

<h2 id="ZeroCopy">ZeroCopy</h2>
这里所说的 ZeroCopy，指的是 Linux 所提供的 ZeroCopy 的能力。上一章中我们说了业务层
零拷贝，而众所周知，当我们调用 sendmsg 系统调用发包的时候，实际上仍然是会产生一次数据的
贝的，并且在大包场景下这个拷贝的消耗非常明显。以 100M 为例，perf 可以看到如下结果：
<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/file/2020/05/6404-4bcbcd0f.png?imageView2/2/interlace/1/format/jpg">
这还仅仅是普通 tcp 发包的占用，在我们的场景下，大部分服务都会接入 Service Mesh，所以
一次发包中，一共会有 3 次拷贝：业务进程到内核、内核到 sidecar、sidecar 再到内核。这使得有
包需求的业务，拷贝所导致的 CPU 占用会特别明显，如下图：
<img src="https://ld246.com/images/img-loading.svg" alt="6405.png" data-src="https:/
b3logfile.com/file/2020/05/6405-677975b7.png?imageView2/2/interlace/1/format/jpg"></p

为了解决这个问题，我们选择了使用 Linux 提供的 ZeroCopy API（在 4.14 以后支持 send；5.4
以后支持 receive）。但是这引入了一个额外的工程问题：ZeroCopy send API 和原先调用方式不兼
，无法很好地共存。这里简单介绍一下 ZeroCopy send 的工作方式：业务进程调用 sendmsg 之后，
endmsg 会记录下 iovec 的地址并立即返回，这时候业务进程不能释放这段内存，需要通过 epoll 等
内核回调一个信号表明某段 iovec 已经发送成功之后才能释放。由于我们并不希望更改业务方的使用
法，需要对上层提供同步收发的接口，所以很难基于现有的 API 同时提供 ZeroCopy 和非 ZeroCopy
的抽象；而由于 ZeroCopy 在小包场景下是有性能损耗的，所以也不能将这个作为默认的选项。
于是，字节跳动框架组和字节跳动内核组合作，由内核组提供了同步的接口：当调用 sendmsg
时候，内核会监听并拦截内核原先给业务的回调，并且在回调完成后才会让 sendmsg 返回。这使得
们无需更改原有模型，可以很方便地接入 ZeroCopy send。同时，字节跳动内核组还实现了基于 unix
domain socket 的 ZeroCopy，可以使得业务进程与 Mesh sidecar 之间的通信也达到零拷贝。
在使用了 ZeroCopy send 后，perf 可以看到内核不再有 copy 的占用：

<img src="https://ld246.com/images/img-loading.svg" alt="6406.png" data-src="https://b3logfile.com/file/2020/05/6406-949075d6.png?imageView2/2/interlace/1/format/jpg">

从 CPU 占用数值上看，大包场景下 ZeroCopy 能够比非 ZeroCopy 节省一半的 CPU。
<h2 id="Go-调度导致的延迟问题分享">Go 调度导致的延迟问题分享</h2>
在我们实践过程中，发现我们新写的 netpoll 虽然在 avg 延迟上表现胜于 Go 原生的 net 库，是在 p99 和 max 延迟上要普遍略高于 Go 原生的 net 库，并且尖刺也会更加明显，如下图（Go 1.1，蓝色为 netpoll + 多路复用，绿色为 netpoll + 长连接，黄色为 net 库 + 长连接）：
<img src="https://ld246.com/images/img-loading.svg" alt="6407.png" data-src="https://b3logfile.com/file/2020/05/6407-e1dbdacd.png?imageView2/2/interlace/1/format/jpg">

我们尝试了很多种办法去优化，但是收效甚微。最终，我们定位出这个延迟并非是由于 netpoll 身的开销导致的，而是由于 go 的调度导致的，比如说：

由于在 netpoll 中，SubReactor 本身也是一个 goroutine，受调度影响，不能保证 EpollWait 调之后马上执行，所以这一块会有延迟；
同时，由于用来处理 I/O 事件的 SubReactor 和用来处理连接监听的 MainReactor 本身也是 goutine，所以实际上很难保证在多核情况之下，这些 Reactor 能并行执行；甚至在最极端情况之下，能这些 Reactor 会挂在同一个 P 下，最终变成了串行执行，无法充分利用多核优势；
由于 EpollWait 回调之后，SubReactor 内是串行处理 I/O 事件的，导致排在最后的事件可能会长尾问题；
在连接多路复用场景下，由于每个连接绑定了一个 SubReactor，故延迟完全取决于这个 SubReator 的调度，导致尖刺会更加明显。

由于 Go 在 runtime 中对于 net 库有做特殊优化，所以 net 库不会有以上情况；同时 net 库是oroutine-per-connection 的模型，所以能确保请求能并行执行而不会相互影响。
对于以上这个问题，我们目前解决的思路有两个：

修改 Go runtime 源码，在 Go runtime 中注册一个回调，每次调度时调用 EpollWait，把获取的 fd 传递给回调执行；
与字节跳动内核组合作，支持同时批量读/写多个连接，解决串行问题。另外，经过我们的测试，o 1.14 能够使得延迟略有降低同时更加平稳，但是所能达到的极限 QPS 更低。希望我们的思路能够业界同样遇到此问题的同学提供一些参考。

<h2 id="后记">后记</h2>
希望以上的分享能够对社区有所帮助。同时，我们也在加速建设 netpoll 以及基于 netpoll 的新架 KiteX。欢迎各位感兴趣的同学加入我们，共同建设 Go 语言生态！
<h2 id="参考资料">参考资料</h2>

http://man7.org/linu/man-pages/man7/epoll.7.html
https://golang.org/src/runtime/proc.o
https://github.com/panjf2000/gnet
https://github.com/tidwall/evio

<h2 id="原文">原文</h2>
字节跳动在 Go 网络库的实践

</p>

原文链接：字节跳动在 Go 网络库上的实践