导言

相信那些曾经使用 Go 写过 proxy server 的同学应该对 io.Copy()/io.CopyN()/io.CopyBuffer()/io.ReaderFrom 等接口和方法不陌生，它们是使用 Go 操作各类 I/O 进行数据传输经常需要使用到的 API，其中基于 TCP 协议的 socket 在使用上述接口和方法进行数据传输时利用到了 Linux 的零拷贝技术 sendfile 和 splice。

我前段时间为 Go 语言内部的 Linux splice 零拷贝技术做了一点优化：为 splice 系统调用实现了一个 pipe pool，复用管道，减少频繁创建和销毁 pipe buffers 所带来的系统开销，理论上来说能够大幅提升 Go 的 io 标准库中基于 splice 零拷贝实现的 API 的性能。因此，我想从这个优化工作出发，分享一些我个人对多线程编程中的一些不成熟的优化思路。

**因本人才疏学浅，故行文之间恐有纰漏，望诸君海涵，不吝赐教，若能予以斧正，则感激不尽**。

splice

纵观 Linux 的零拷贝技术，相较于mmap、sendfile和 MSG\_ZEROCOPY 等其他技术，splice 从使用成本、性能和适用范围等维度综合来看更适合在程序中作为一种通用的零拷贝方式。

splice() 系统调用函数定义如下：

**#include <fcntl.h>**

**#include <unistd.h>**

**int** **pipe**(**int** pipefd[2]);

**int** **pipe2**(**int** pipefd[2], **int** flags);

**ssize\_t** **splice**(**int** fd\_in, **loff\_t** \*off\_in, **int** fd\_out, **loff\_t** \*off\_out, **size\_t** len, **unsigned** **int** flags);

fd\_in 和 fd\_out 也是分别代表了输入端和输出端的文件描述符，这两个文件描述符必须有一个是指向管道设备的，这算是一个不太友好的限制。

off\_in 和 off\_out 则分别是 fd\_in 和 fd\_out 的偏移量指针，指示内核从哪里读取和写入数据，len 则指示了此次调用希望传输的字节数，最后的 flags 是系统调用的标记选项位掩码，用来设置系统调用的行为属性的，由以下 0 个或者多个值通过『或』操作组合而成：

* SPLICE\_F\_MOVE：指示 splice() 尝试仅仅是移动内存页面而不是复制，设置了这个值不代表就一定不会复制内存页面，复制还是移动取决于内核能否从管道中移动内存页面，或者管道中的内存页面是否是完整的；这个标记的初始实现有很多 bug，所以从 Linux 2.6.21 版本开始就已经无效了，但还是保留了下来，因为在未来的版本里可能会重新被实现。
* SPLICE\_F\_NONBLOCK：指示 splice() 不要阻塞 I/O，也就是使得 splice() 调用成为一个非阻塞调用，可以用来实现异步数据传输，不过需要注意的是，数据传输的两个文件描述符也最好是预先通过 O\_NONBLOCK 标记成非阻塞 I/O，不然 splice() 调用还是有可能被阻塞。
* SPLICE\_F\_MORE：通知内核下一个 splice() 系统调用将会有更多的数据传输过来，这个标记对于输出端是 socket 的场景非常有用。

splice() 是基于 Linux 的管道缓冲区 (pipe buffer) 机制实现的，所以 splice() 的两个入参文件描述符才要求必须有一个是管道设备，一个典型的 splice() 用法是：

int pfd[2];

pipe(pfd);

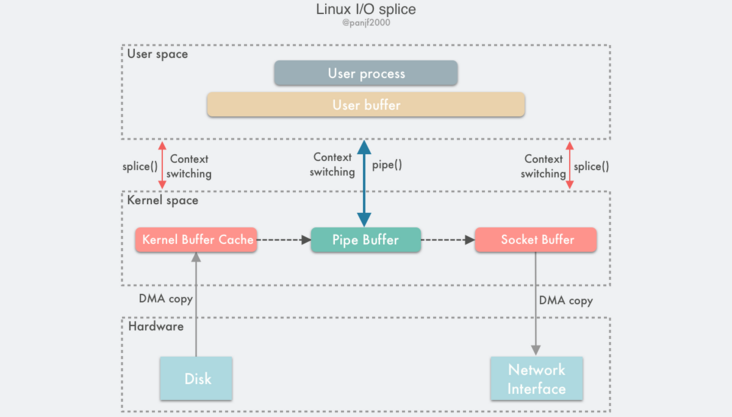
ssize\_t bytes = splice(file\_fd, **NULL**, pfd[1], **NULL**, 4096, **SPLICE\_F\_MOVE**);

**assert**(bytes != -1);

bytes = splice(pfd[0], **NULL**, socket\_fd, **NULL**, bytes, **SPLICE\_F\_MOVE** | **SPLICE\_F\_MORE**);

**assert**(bytes != -1);

数据传输过程图：



使用 splice() 完成一次磁盘文件到网卡的读写过程如下：

1. 用户进程调用 pipe()，从用户态陷入内核态，创建匿名单向管道，pipe() 返回，上下文从内核态切换回用户态；
2. 用户进程调用 splice()，从用户态陷入内核态；
3. DMA 控制器将数据从硬盘拷贝到内核缓冲区，从管道的写入端"拷贝"进管道，splice() 返回，上下文从内核态回到用户态；
4. 用户进程再次调用 splice()，从用户态陷入内核态；
5. 内核把数据从管道的读取端"拷贝"到套接字缓冲区，DMA 控制器将数据从套接字缓冲区拷贝到网卡；
6. splice() 返回，上下文从内核态切换回用户态。

上面是 splice 的基本工作流程和原理，简单来说就是在数据传输过程中传递内存页指针而非实际数据来实现零拷贝，如果有意了解其更底层的实现原理请移步：[《Linux I/O 原理和 Zero-copy 技术全面揭秘》](https://strikefreedom.top/linux-io-and-zero-copy" \t "_blank)。

pipe pool for splice

pipe pool in HAProxy

从上面对 splice 的介绍可知，通过它实现数据零拷贝需要利用到一个媒介 -- pipe 管道（2005 年由 Linus 引入），大概是因为在 Linux 的 IPC 机制中对 pipe 的应用已经比较成熟，于是借助了 pipe 来实现 splice，虽然 Linux Kernel 团队曾在 splice 诞生之初便说过在未来可以移除掉 pipe 这个限制，但十几年过去了也依然没有付诸实施，因此 splice 至今还是和 pipe 死死绑定在一起。

那么问题就来了，如果仅仅是使用 splice 进行单次的大批量数据传输，则创建和销毁 pipe 开销几乎可以忽略不计，但是如果是需要频繁地使用 splice 来进行数据传输，比如需要处理大量网络 sockets 的数据转发的场景，则 pipe 的创建和销毁的频次也会随之水涨船高，每调用一次 splice 都创建一对 pipe 管道描述符，并在随后销毁掉，对一个网络系统来说是一个巨大的消耗。

对于这问题的解决方案，自然而然就会想到 -- 『复用』，比如大名鼎鼎的 HAProxy。

HAProxy 是一个使用 C 语言编写的自由及开放源代码软件，其提供高可用性、负载均衡，以及基于 TCP 和 HTTP 的应用程序代理。它非常适用于那些有着极高网络流量的 Web 站点。GitHub、Bitbucket、Stack Overflow、Reddit、Tumblr、Twitter 和 Tuenti 在内的知名网站，及亚马逊网络服务系统都在使用 HAProxy。

因为需要做流量转发，可想而知，HAProxy 不可避免地要高频地使用 splice，因此对 splice 带来的创建和销毁 pipe buffers 的开销无法忍受，从而需要实现一个 pipe pool，复用 pipe buffers 减少系统调用消耗，下面我们来详细剖析一下 HAProxy 的 pipe pool 的设计思路。

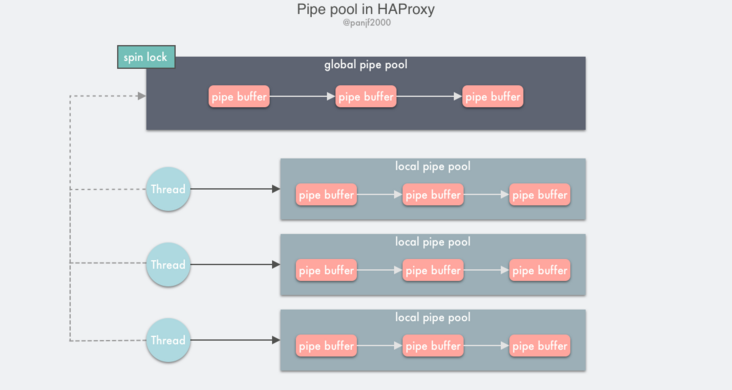
首先我们来自己思考一下，一个最简单的 pipe pool 应该如何实现，最直接且简单的实现无疑就是：一个单链表+一个互斥锁。链表和数组是用来实现 pool 的最简单的数据结构，数组因为数据在内存分配上的连续性，能够更好地利用 CPU 高速缓存加速访问，但是首先，对于运行在某个 CPU 上的线程来说，一次只需要取一个 pipe buffer 使用，所以高速缓存在这里的作用并不十分明显；其次，数组不仅是连续而且是固定大小的内存区，需要预先分配好固定大小的内存，而且还要动态伸缩这个内存区，期间需要对数据进行搬迁等操作，增加额外的管理成本。链表则是更加适合的选择，因为作为 pool 来说其中所有的资源都是等价的，并不需要随机访问去获取其中某个特定的资源，而且链表天然是动态伸缩的，随取随弃。

锁通常使用 mutex，在 Linux 上的早期实现是一种完全基于内核态的 sleep-waiting 也就是休眠等待的锁，kernel 维护一个对所有进程/线程都可见的共享资源对象 mutex，多线程/进程的加锁解锁其实就是对这个对象的竞争。如果现在有 AB 两个进程/线程，A 首先进入 kernel space 检查 mutex，看看有没有别的进程/线程正在占用它，抢占 mutex 成功之后则直接进入临界区，B 尝试进入临界区的时候，检测到 mutex 已被占用，就由运行态切换成睡眠态，等待该共享对象释放，A 出临界区的时候，需要再次进入 kernel space 查看有没有别的进程/线程在等待进入临界区，然后 kernel 会唤醒等待的进程/线程并在合适的时间把 CPU 切换给该进程/线程运行。由于最初的 mutex 是一种完全内核态的互斥量实现，在并发量大的情况下会产生大量的系统调用和上下文切换的开销，在 Linux kernel 2.6.x 之后都是使用 futex (Fast Userspace Mutexes) 实现，也即是一种用户态和内核态混用的实现，通过在用户态共享一段内存，并利用原子操作读取和修改信号量，在没有竞争的时候只需检查这个用户态的信号量而无需陷入内核，信号量存储在进程内的私有内存则是线程锁，存储在通过 mmap 或者 shmat 创建的共享内存中则是进程锁。

即便是基于 futex 的互斥锁，如果是一个全局的锁，这种最简单的 pool + mutex 实现在竞争激烈的场景下会有可预见的性能瓶颈，因此需要进一步的优化，优化手段无非两个：降低锁的粒度或者减少抢(全局)锁的频次。因为 pipe pool 中的资源本来就是全局共享的，也就是无法对锁的粒度进行降级，因此只能是尽量减少多线程抢锁的频次，而这种优化常用方案就是在全局资源池之外引入本地资源池，对多线程访问资源的操作进行错开。

至于锁本身的优化，由于 mutex 是一种休眠等待锁，即便是基于 futex 优化之后在锁竞争时依然需要涉及内核态开销，此时可以考虑使用自旋锁（Spin Lock），也即是用户态的锁，共享资源对象存在用户进程的内存中，避免在锁竞争的时候陷入到内核态等待，自旋锁比较适合临界区极小的场景，而 pipe pool 的临界区里只是对链表的增删操作，非常匹配。

HAProxy 实现的 pipe pool 就是依据上述的思路进行设计的，将单一的全局资源池拆分成全局资源池+本地资源池。



全局资源池利用单链表和自旋锁实现，本地资源池则是基于线程私有存储（Thread Local Storage, TLS）实现，TLS 是一种线程的私有的变量，它的主要作用是在多线程编程中避免锁竞争的开销。TLS 由编译器提供支持，我们知道编译 C 程序得到的 obj 或者链接得到的 exe，其中的 .text 段保存代码文本，.data 段保存已初始化的全局变量和已初始化的静态变量，.bss 段则保存未初始化的全局变量和未初始化的局部静态变量。

而 TLS 私有变量则会存入 TLS 帧，也就是 .tdata 和 .tboss 段，与.data 和 .bss 不同的是，运行时程序不会直接访问这些段，而是在程序启动后，动态链接器会对这两个段进行动态初始化 （如果有声明 TLS 的话），之后这两个段不会再改变，而是作为 TLS 的初始镜像保存起来。每次启动一个新线程的时候都会将 TLS 块作为线程堆栈的一部分进行分配并将初始的 TLS 镜像拷贝过来，也就是说最终每个线程启动时 TLS 块中的内容都是一样的。

HAProxy 的 pipe pool 实现原理：

1. 声明 thread\_local 修饰的一个单链表，节点是 pipe buffer 的两个管道描述符，那么每个需要使用 pipe buffer 的线程都会初始化一个基于 TLS 的单链表，用以存储 pipe buffers；
2. 设置一个全局的 pipe pool，使用自旋锁保护。

每个线程去取 pipe 的时候会先从自己的 TLS 中去尝试获取，获取不到则加锁进入全局 pipe pool 去找；使用 pipe buffer 过后将其放回：先尝试放回 TLS，根据一定的策略计算当前 TLS 的本地 pipe pool 链表中的节点是否已经过多，是的话则放到全局的 pipe pool 中去，否则直接放回本地 pipe pool。

HAProxy 的 pipe pool 实现虽然只有短短的 100 多行代码，但是其中蕴含的设计思想却包含了许多非常经典的多线程优化思路，值得细读。

pipe pool in Go

受到 HAProxy 的 pipe pool 的启发，我尝试为 Golang 的 io 标准库里底层的 splice 实现了一个 pipe pool，不过熟悉 Go 的同学应该知道 Go 有一个 GMP 并发调度器，提供了强大并发调度能力的同时也屏蔽了操作系统层级的线程，所以 Go 没有提供类似 TLS 的机制，倒是有一些开源的第三方库提供了类似的功能，比如 [gls](https://github.com/jtolio/gls)，虽然实现很精巧，但毕竟不是官方标准库而且会直接操作底层堆栈，所以其实也并不推荐在线上使用。

一开始，因为 Go 缺乏 TLS 机制，所以我提交的第一版 go pipe pool 就是一个很简陋的单链表+全局互斥锁的实现，因为这个方案在进程的生命周期中并不会去释放资源池里的 pipe buffers（实际上 HAProxy 的 pipe pool 也会有这个问题），也就是说那些未被释放的 pipe buffers 将一直存在于用户进程的生命周期中，直到进程结束之后才由 kernel 进行释放，这明显不是一个令人信服的解决方案，结果不出意料地被 Go team 的核心大佬 [Ian](https://www.airs.com/ian/) (委婉地)否决了，于是我马上又想了两个新的方案：

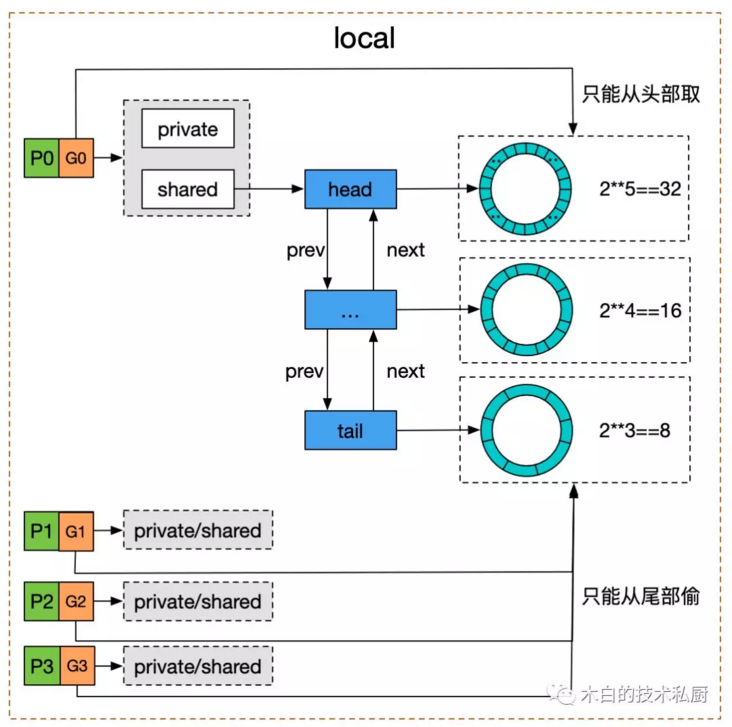
1. 基于这个现有的方案加上一个独立的 goroutine 定时去扫描 pipe pool，关闭并释放 pipe buffers；
2. 基于 sync.Pool 标准库来实现 pipe pool，并利用 runtime.SetFinalizer 来解决定期释放 pipe buffers 的问题。

第一个方案需要引入额外的 goroutine，并且该 goroutine 也为这个设计增加了不确定的因素，而第二个方案则更加优雅，首先因为基于 sync.Pool 实现，其底层也可以说是基于 TLS 的思想，其次利用了 Go 的 runtime 来解决定时释放 pipe buffers 的问题，实现上更加的优雅，所以很快，我和其他的 Go reviewers 就达成一致决定采用第二个方案。

sync.Pool 是 Go 语言提供的临时对象缓存池，一般用来复用资源对象，减轻 GC 压力，合理使用它能对程序的性能有显著的提升。很多顶级的 Go 开源库都会重度使用 sync.Pool 来提升性能，比如 Go 领域最流行的第三方 HTTP 框架 [fasthttp](https://github.com/valyala/fasthttp) 就在源码中大量地使用了 sync.Pool，并且收获了比 Go 标准 HTTP 库高出近 10 倍的性能提升（当然不仅仅靠这一个优化点，还有很多其他的），fasthttp 的作者 Aliaksandr Valialkin 作为 Go 领域的大神（Go contributor，给 Go 贡献过很多代码，也优化过 sync.Pool），在 fasthttp 的 [best practices](https://github.com/valyala/fasthttp#fasthttp-best-practices) 中极力推荐使用 sync.Pool，所以 Go 的 pipe pool 使用 sync.Pool 来实现也算是水到渠成。

sync.Pool 底层原理简单来说就是：私有变量+共享双向链表。

Google 了一张图来展示 sync.Pool 的底层实现：



* 获取对象时：当某个 P 上的 goroutine 从 sync.Pool 尝试获取缓存的对象时，需要先把当前的 goroutine 锁死在 P 上，防止操作期间突然被调度走，然后先尝试去取本地私有变量 private，如果没有则去 shared 双向链表的表头取，该链表可以被其他 P 消费（或者说"偷"），如果当前 P 上的 shared 是空则去"偷"其他 P 上的 shared 双向链表的表尾，最后解除锁定，如果还是没有取到缓存的对象，则直接调用 New 创建一个返回。
* 放回对象时：先把当前的 goroutine 锁死在 P 上，如果本地的 private 为空，则直接将对象存入，否则就存入 shared 双向链表的表头，最后解除锁定。

shared 双向链表的每个节点都是一个环形队列，主要是为了高效复用内存，共享双向链表在 Go 1.13 之前使用互斥锁 sync.Mutex 保护，Go 1.13 之后改用 atomic CAS 实现无锁并发，原子操作无锁并发适用于那些临界区极小的场景，性能会被互斥锁好很多，正好很贴合 sync.Pool 的场景，因为存取临时对象的操作是非常快速的，如果使用 mutex，则在竞争时需要挂起那些抢锁失败的 goroutines 到 wait queue，等后续解锁之后唤醒并放入 run queue，等待调度执行，还不如直接忙轮询等待，反正很快就能抢占到临界区。

sync.Pool 的设计也具有部分的 TLS 思想，所以从某种意义上来说它是就 Go 语言的 TLS 机制。

sync.Pool **基于 victim cache 会保证缓存在其中的资源对象最多不超过两个 GC 周期就会被回收掉**。

因此我使用了 sync.Pool 来实现 Go 的 pipe pool，把 pipe 的管道文件描述符对存储在其中，并发之时进行复用，而且会定期自动回收，但是还有一个问题，当 sync.Pool 中的对象被回收的时候，只是回收了管道的文件描述符对，也就是两个整型的 fd 数，并没有在操作系统层面关闭掉 pipe 管道。

因此，还需要有一个方法来关闭 pipe 管道，这时候可以利用 runtime.SetFinalizer 来实现。这个方法其实就是对一个即将放入 sync.Pool 的资源对象设置一个回调函数，当 Go 的三色标记 GC 算法检测到 sync.Pool 中的对象已经变成白色（unreachable，也就是垃圾）并准备回收时，如果该白色对象已经绑定了一个关联的回调函数，则 GC 会先解绑该回调函数并启动一个独立的 goroutine 去执行该回调函数，因为回调函数使用该对象作为函数入参，也就是会引用到该对象，那么就会导致该对象重新变成一个 reachable 的对象，所以在本轮 GC 中不会被回收，从而使得这个对象的生命得以延续一个 GC 周期。

在每一个 pipe buffer 放回 pipe pool 之前通过 runtime.SetFinalizer 指定一个回调函数，在函数中使用系统调用关闭管道，则可以利用 Go 的 GC 机制定期真正回收掉 pipe buffers，从而实现了一个优雅的 pipe pool in Go，相关的 commits 如下：

* [internal/poll: implement a pipe pool for splice() call](https://github.com/golang/go/commit/643d240a11b2d00e1718b02719707af0708e7519)
* [internal/poll: fix the intermittent build failures with pipe pool](https://github.com/golang/go/commit/6382ec1aba1b1c7380cb525217c1bd645c4fd41b)
* [internal/poll: ensure that newPoolPipe doesn't return a nil pointer](https://github.com/golang/go/commit/8b859be9c3fd1068b659afa1db76dadb210c63de)
* [internal/poll: cast off the last reference of SplicePipe in test](https://github.com/golang/go/commit/832c70e33d8265116f0abce436215b8e9ee4bb08)

为 Go 的 splice 引入 pipe pool 之后，对性能的提升效果如下：

goos: linux

goarch: amd64

pkg: internal/poll

cpu: AMD EPYC 7K62 48-Core Processor

name old time/op new time/op delta

SplicePipe-8 1.36µs ± 1% 0.02µs ± 0% -98.57% (p=0.001 n=7+7)

SplicePipeParallel-8 747ns ± 4% 4ns ± 0% -99.41% (p=0.001 n=7+7)

name old alloc/op new alloc/op delta

SplicePipe-8 24.0B ± 0% 0.0B -100.00% (p=0.001 n=7+7)

SplicePipeParallel-8 24.0B ± 0% 0.0B -100.00% (p=0.001 n=7+7)

name old allocs/op new allocs/op delta

SplicePipe-8 1.00 ± 0% 0.00 -100.00% (p=0.001 n=7+7)

SplicePipeParallel-8 1.00 ± 0% 0.00 -100.00% (p=0.001 n=7+7)

基于 pipe pool 复用和直接创建&销毁 pipe buffers 相比，耗时下降在 99% 以上，内存使用则是下降了 100%。

当然，这个 benchmark 只是一个纯粹的存取操作，并未加入具体的业务逻辑，所以是一个非常理想化的压测，不能完全代表生产环境，但是 pipe pool 的引入对使用 Go 的 io 标准库并基于 splice 进行高频的零拷贝操作的性能必定会有数量级的提升。

这个特性最快应该会在今年下半年的 Go 1.17 版本发布，到时就可以享受到 pipe pool 带来的性能提升了。

小结

通过给 Go 语言实现一个 pipe pool，期间涉及了多种并发、同步的优化思路，我们再来归纳总结一下。

1. 资源复用，提升并发编程性能最有效的手段一定是资源复用，也是最立竿见影的优化手段。
2. 数据结构的选取，数组支持 O(1) 随机访问并且能更好地利用 CPU cache，但是这些优势在 pool 的场景下并不明显，因为 pool 中的资源具有等价性和单个存取(非批量)操作，数组需要预先分配固定内存并且伸缩的时候会有额外的内存管理负担，链表随取随弃，天然支持动态伸缩。
3. 全局锁的优化，两种思路，一种是根据资源的特性尝试对锁的粒度进行降级，一种是通过引入本地缓存，尝试错开多线程对资源的访问，减少竞争全局锁的频次；还有就是根据实际场景适当地选择用户态锁。
4. 利用语言的 runtime，像 Go、Java 这类自带一个庞大的 GC 的编程语言，在性能上一般不是 C/C++/Rust 这种无 GC 语言的对手，但是凡事有利有弊，自带 runtime 的语言也具备独特的优势，比如 HAProxy 的 pipe pool 是 C 实现的，在进程的生命周期中创建的 pipe buffers 会一直存在占用资源（除非主动关闭，但是很难准确把控时机），而 Go 实现的 pipe pool 则可以利用自身的 runtime 进行定期的清理工作，进一步减少资源占用。