频繁的文件IO操作

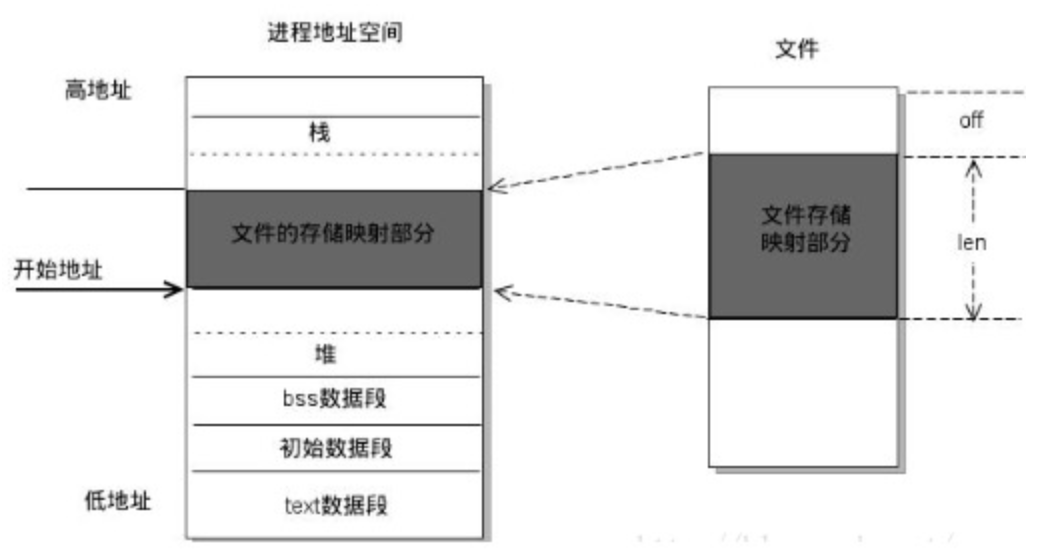
在Android开发中，我们可能需要记录一些文件。 例如，日志文件。 如果使用stream来写文件，频繁的文件IO操作可能会导致性能问题。

为了减少写入文件的频率，我们可能会缓存一定数量的日志，并一次性写入文件。 如果应用程序异常退出，我们可能会丢失内存中的日志信息。

那么有什么更安全的文件写入方式，既能减少IO，又能保证尽可能多的数据写入文件呢？

MMAP是一种在内存中映射文件的方法，即将文件或其他对象映射到进程的地址空间，实现文件的磁盘地址和进程的虚拟地址空间的一对一映射 .

特点：实现这样的映射关系后，进程可以通过指针的方式读写这块内存，系统会自动将脏页写回对应的文件磁盘，即对文件的操作 无需调用read、write等系统调用函数即可完成。 相反，内核空间对这个区域的修改也直接反映了用户空间，从而可以实现不同进程之间的文件共享。 如下图所示：



一般来说，MMAP内存映射的实现可以分为三个阶段：

在进程的虚拟地址空间中，找到一个空闲且连续的虚拟地址作为映射区；

调用系统函数MMAP，将文件的物理地址映射到进程的虚拟地址；

应用进程访问映射区，导致缺页异常，将文件内容复制到物理内存（主存）。

优缺点：

只拷贝一次数据：如果出现缺页异常，直接将数据从磁盘拷贝到进程的用户空间，跳过页缓存。

实现了用户空间和内核空间之间的高效交互：两个空间的修改操作可以直接反映在映射区域中，从而可以被其他空间及时捕获。

它为进程共享内存和相互通信提供了一种方式。

无论是父子进程还是不相关的进程，它都可以将自己的用户空间匿名映射到同一个文件或者同一个区域。 这样就可以通过改变映射区域来达到进程间通信和进程间共享的目的。

同时，如果进程a和进程B都映射了区域C，a第一次读取C时，通过缺页将文件页从磁盘复制到内存中； 但是，当B重读C的同一页时，虽然会出现缺页异常，但不再需要从磁盘复制文件，而是可以直接使用内存中保存的文件数据。

对于大文件，内存映射比普通 IO 流更快，而小文件则不然；

不要调用mappedbytebuffer.force() 方法。 此方法强制操作系统将内存内容写入硬盘。 因此，如果每次写入内存映射文件时都调用 force() 方法，则无法真正从内存映射文件中受益，但它类似于磁盘 IO。

操作系统负责读写内存映射文件。 因此，即使你的Java程序写入内存后挂了，只要操作系统正常工作，数据就会写入磁盘。

如果电源出现故障或者主机出现故障，可能是内存映射文件没有写入磁盘，这意味着一些关键数据可能会丢失。

Android 中的 Binder 也使用 MMAP。 Binder 交付数据时，只需复制一次即可交付给另一个进程。

MMAP可以通过RandomAccessFile和mappedbytebuffer在Android中使用。

通过randomAccessFile.getChannel().map获取到MappedByteBuffer。然后调用ByteBuffer的put方法添加数据。

输入输出流

一般来说关于流的特性有下面几点：

先进先出，最先写入输出流的数据最先被输入流读取到。

顺序存取，可以一个接一个地往流中写入一串字节，读出时也将按写入顺序读取一串字节，不能随机访问中间的数据。

只读或只写，每个流只能是输入流或输出流的一种，不能同时具备两个功能，在一个数据传输通道中，如果既要写入数据，又要读取数据，则要分别提供两个流。

带有缓存的IO操作

与其一个字节一个字节的读取/写入数据，一次性读取/写入一块数据到缓冲中，然后一次读取将会带来性能的提升。

Okio的设计就是基于缓存和内存复用。

数据结构

ByteStrings 和Buffers

ByteString 是不可变的字节序列。Buffer 是一个可变的字节序列。ByteString和Buffer做了一些聪明的事情来节省CPU和内存。

如果将UTF-8字符串编码为ByteString，它会缓存对该字符串的引用，这样，如果稍后对其进行解码，就不需要做任何工作。

Buffer 是作为片段的链表实现的。将数据从一个缓冲区移动到另一个缓冲区时，它会重新分配片段的持有关系，而不是跨片段复制数据。这对多线程特别有用：与网络交互的子线程可以与工作线程交换数据，而无需任何复制或多余的操作。

Sources 和Sinks

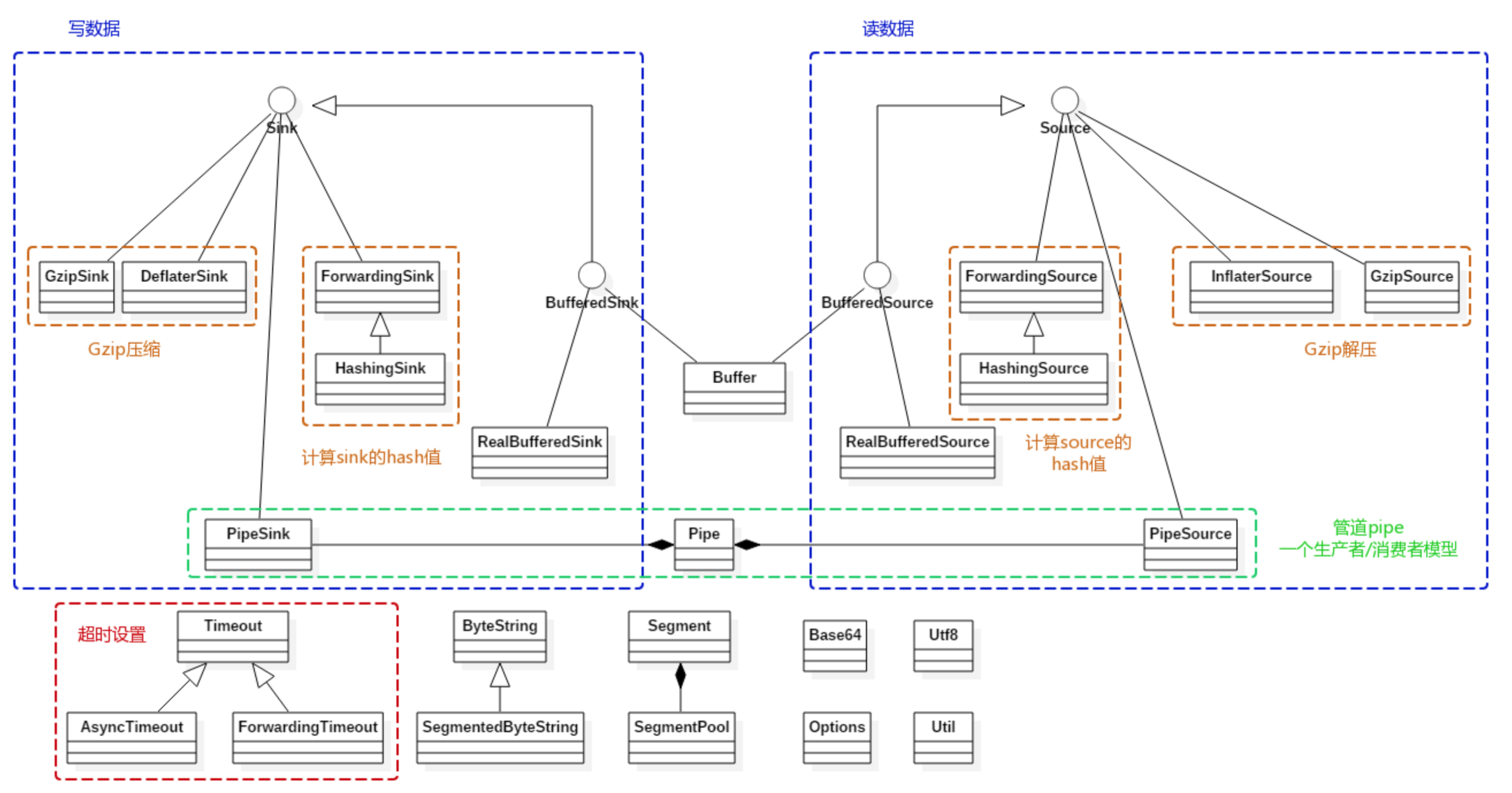
Sources 和 Sinks 分别与 InputStream 和 OutputStream 交互操作。你可以将任何 Source 看做 InputStream ，也可以将任何 InputStream 当做 Source。对于 Sink 和 Outputstream 也是如此。

Segment

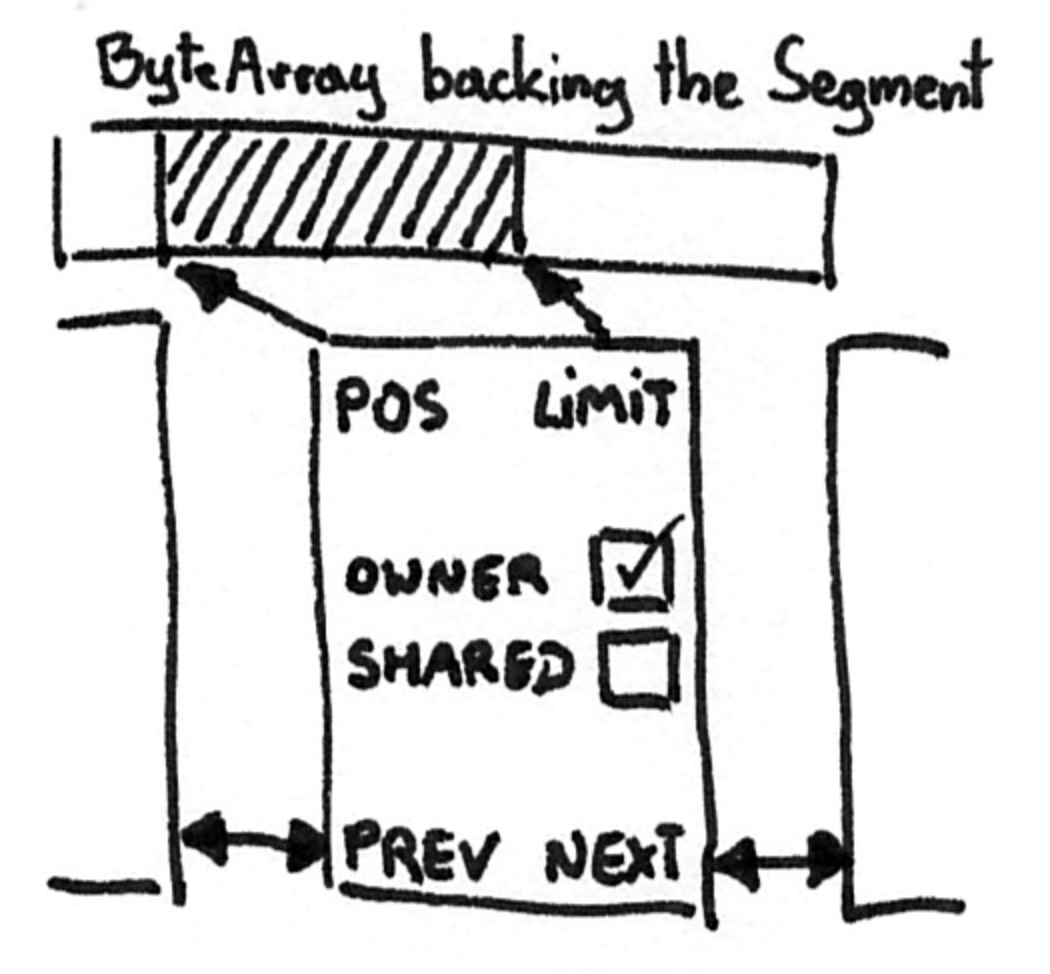
Segment在 Okio 中作为数据缓冲的载体，一个 Segment 的数据缓冲大小为 8192，即 8k。每一个 Segment 都有前驱和后继结点，也就是说 Sement 是一个双向链表链表,准确的来说是一个双向循环链表。读取数据从 Segment 头结点读取，写数据从 Segment 尾结点写。

Okio 中引入池的概念也就是源码中SegmentPool的实现。SegmentPool 负责 Segment 创建和销毁，SegmentPool 最大可以缓存 8 个 Segment。

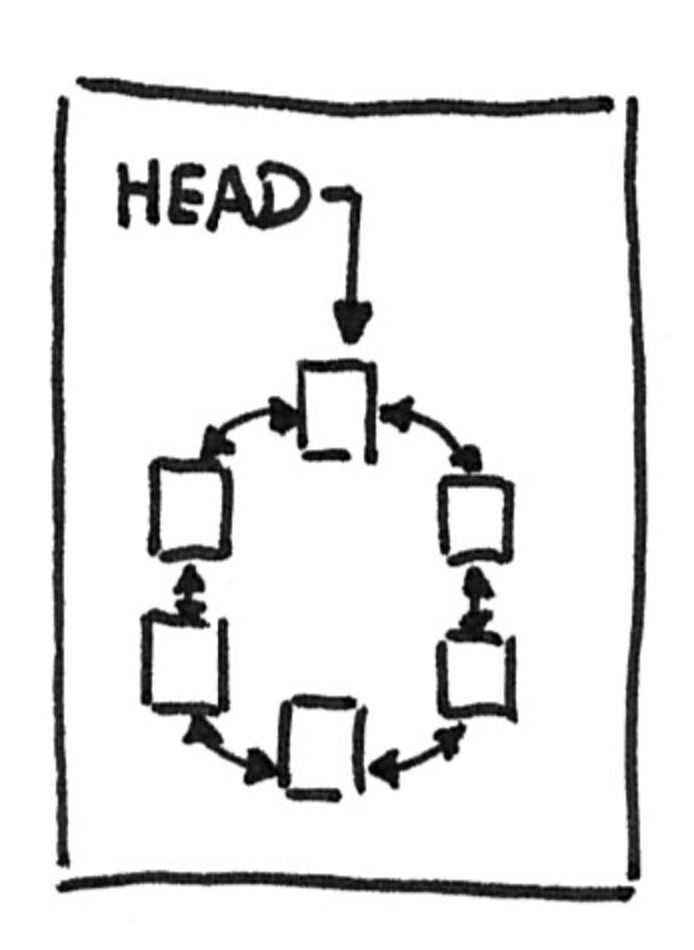
SegmentPool 是一个静态方法，因此也就是全局缓存只有 64 kb。



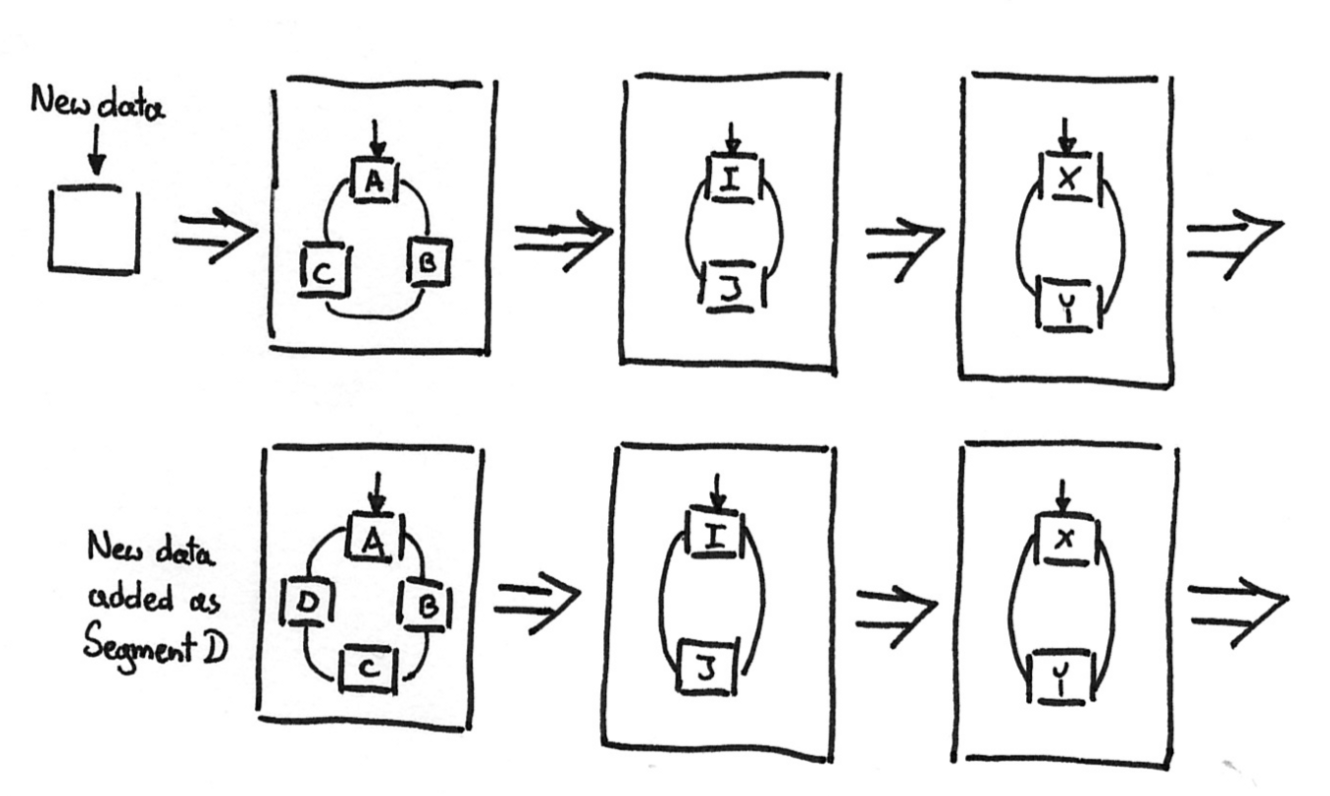
Segment 是双向链表的典型数据结构：它具有对其他 Segment 的 next 和 prev 引用。 它还带有实际的 ByteArray 缓冲区，因为数据必须保存在某个地方。有趣的是两个名为 pos 和 limit 的“指针”以及两个标志：shared 和 owner。



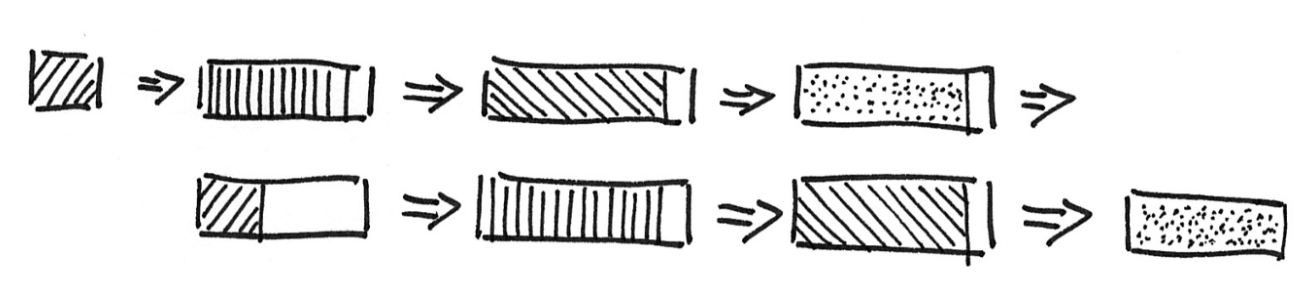
在内部，缓冲区将 Segments 保持为循环链接列表。 这样，只需要维护一个 Segment 引用即可访问 Buffer 的两端，以便分别读取和写入数据。



这意味着缓冲区可以安全地增长，而不会影响通过管道传输到其中的其他 Sinks 和Sources 。



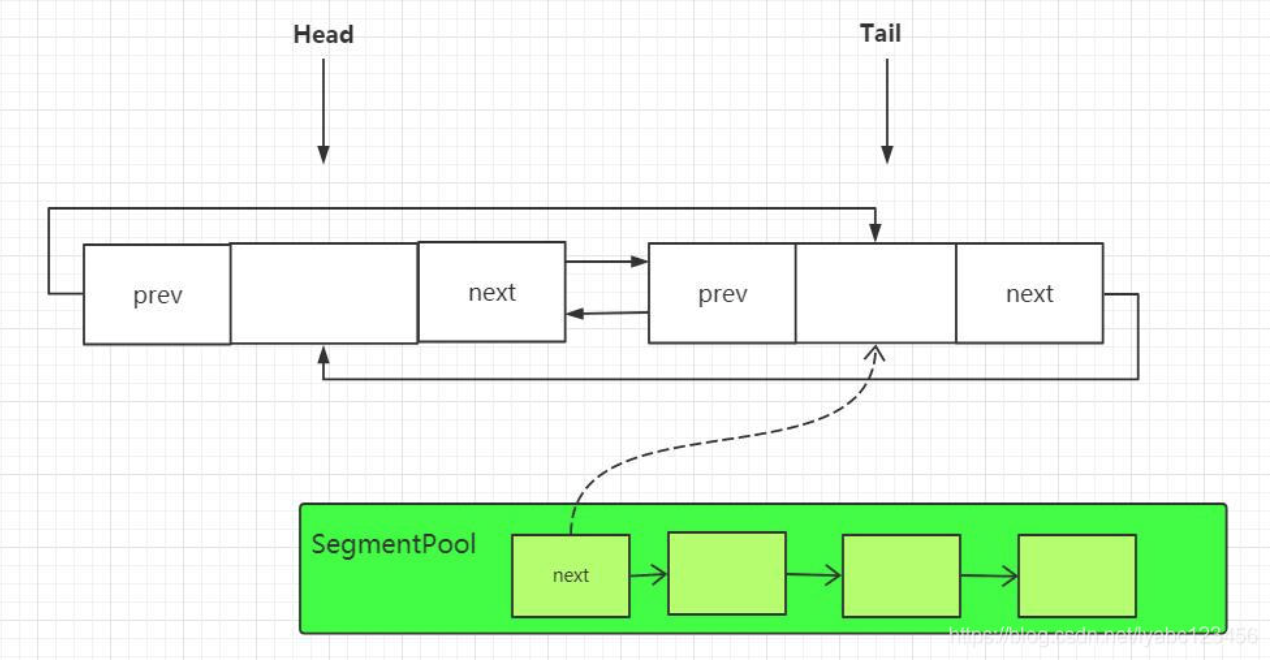
将其与 BufferedInputStream 和 BufferedOutputStream 行为进行比较，其中写入一些数据可能会导致级联刷新。



写操作初始化时的数据结构：



这时头结点和尾节点其实是同一个节点，然后取得 head.prev 也就是 tail 尾节点返回，但是如果此时 tail 能写的字节数限制超过了 8k 或者尾节点不是 data 的拥有者，就会调用tail.push(SegmentPool.take()); 也就是再调用一次 SegmentPool.take() 取到 Segment 池中下一个 Segment. 通过 tail. push() 方法插入到循环链表的尾部。这时 Segment 中的链表会变成下面这样：

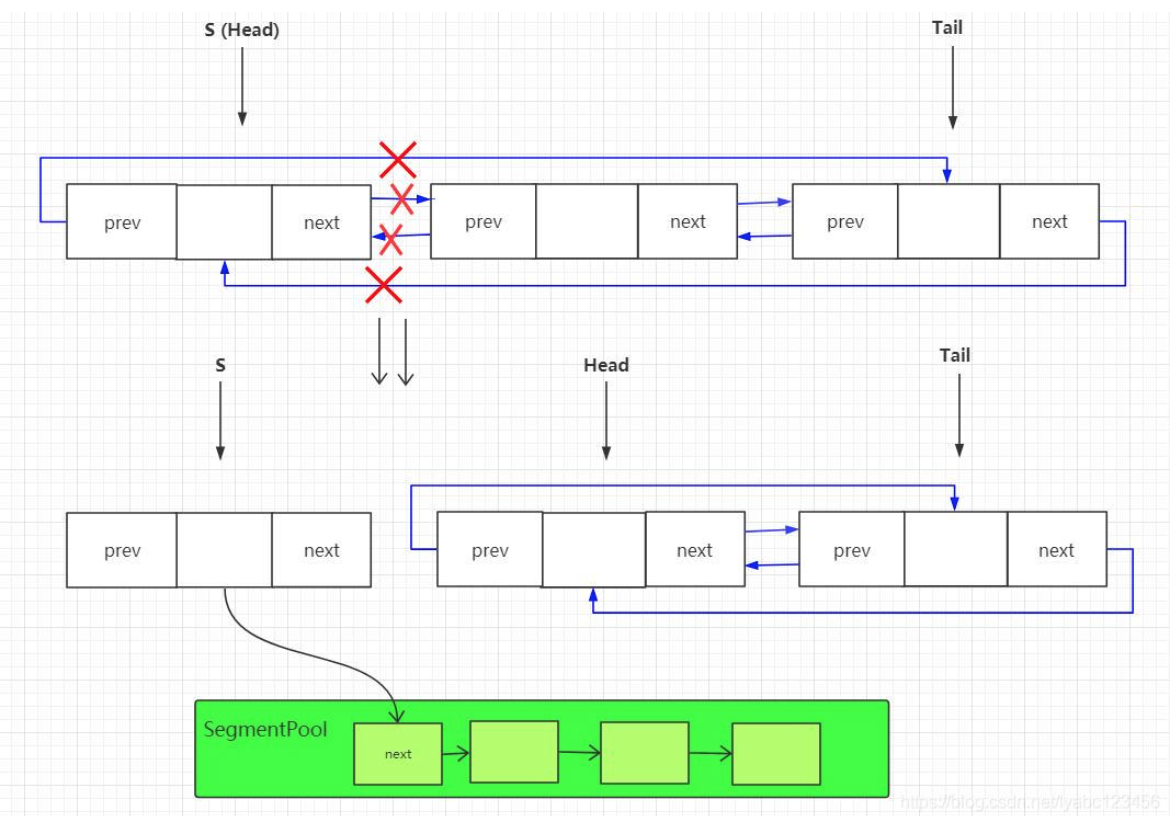


此时插入的节点会作为新的tail节点返回，下一次获取尾节点的时候就会取到它，每当 tail 进行 push 一次，就会将新 push 的节点作为新的尾节点：

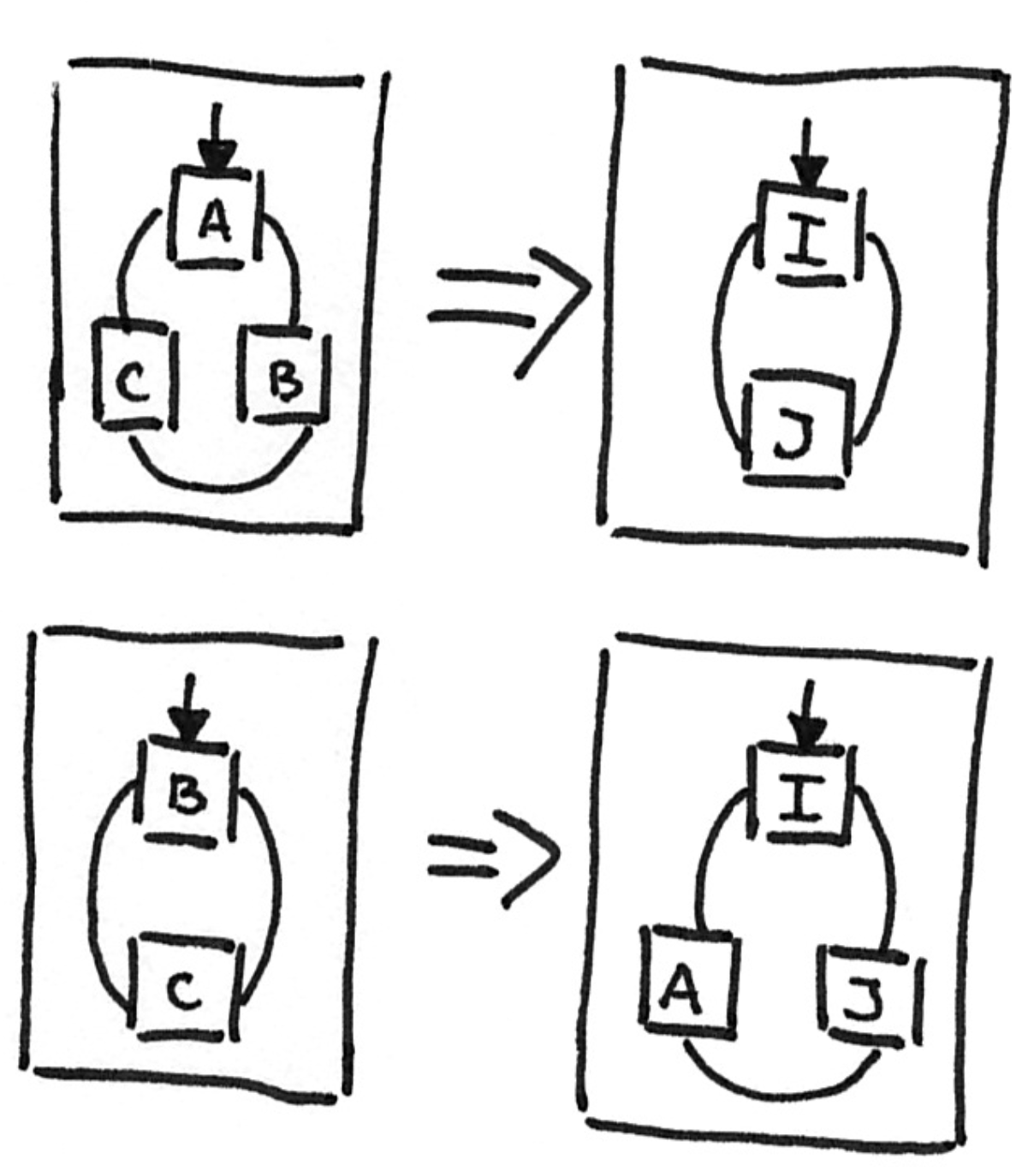


读操作初始化时的数据结构：

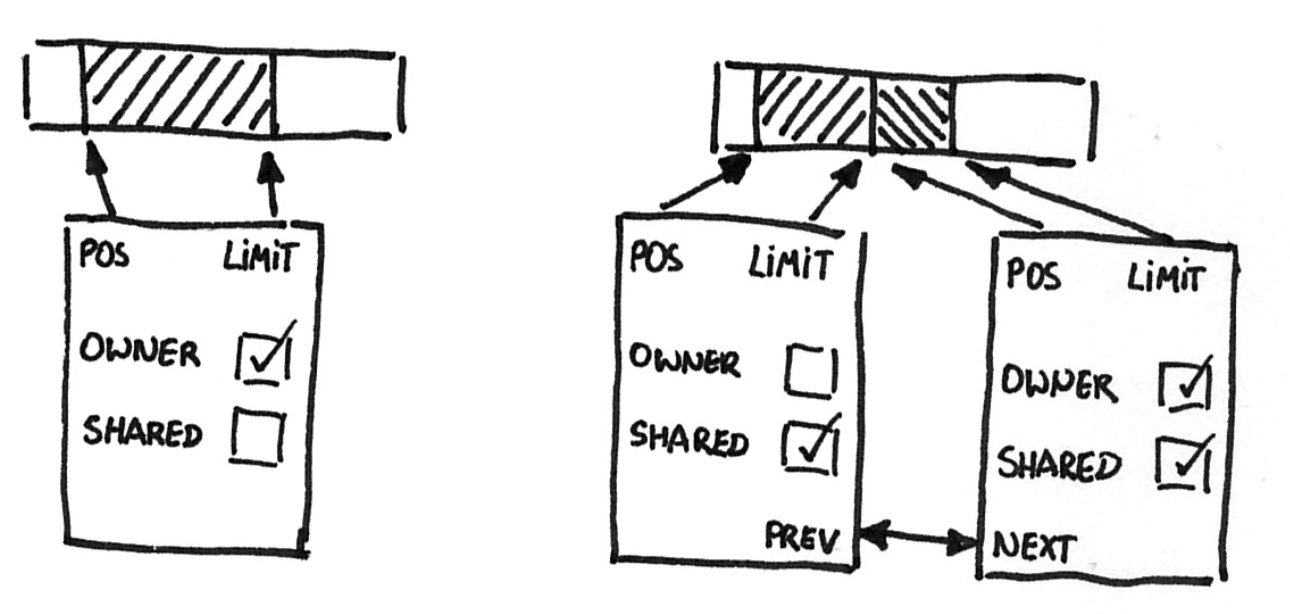
读操作内部也是调用 System.arraycopy 进行字节数组的复制，这里是直接对 head 头结点进行读取，也就是说 Buffer 在每次读数据的时候都是从链表的头部进行读取的，如果读取的头结点的 pos 等于 limit, 这里就会调用 s.pop() 将头节点从链表中删除，并返回下一个节点作为新的头结点引用，然后将删除的节点通过 SegmentPool.recycle(s) 进行回收复用。这时链表中的变化如下：



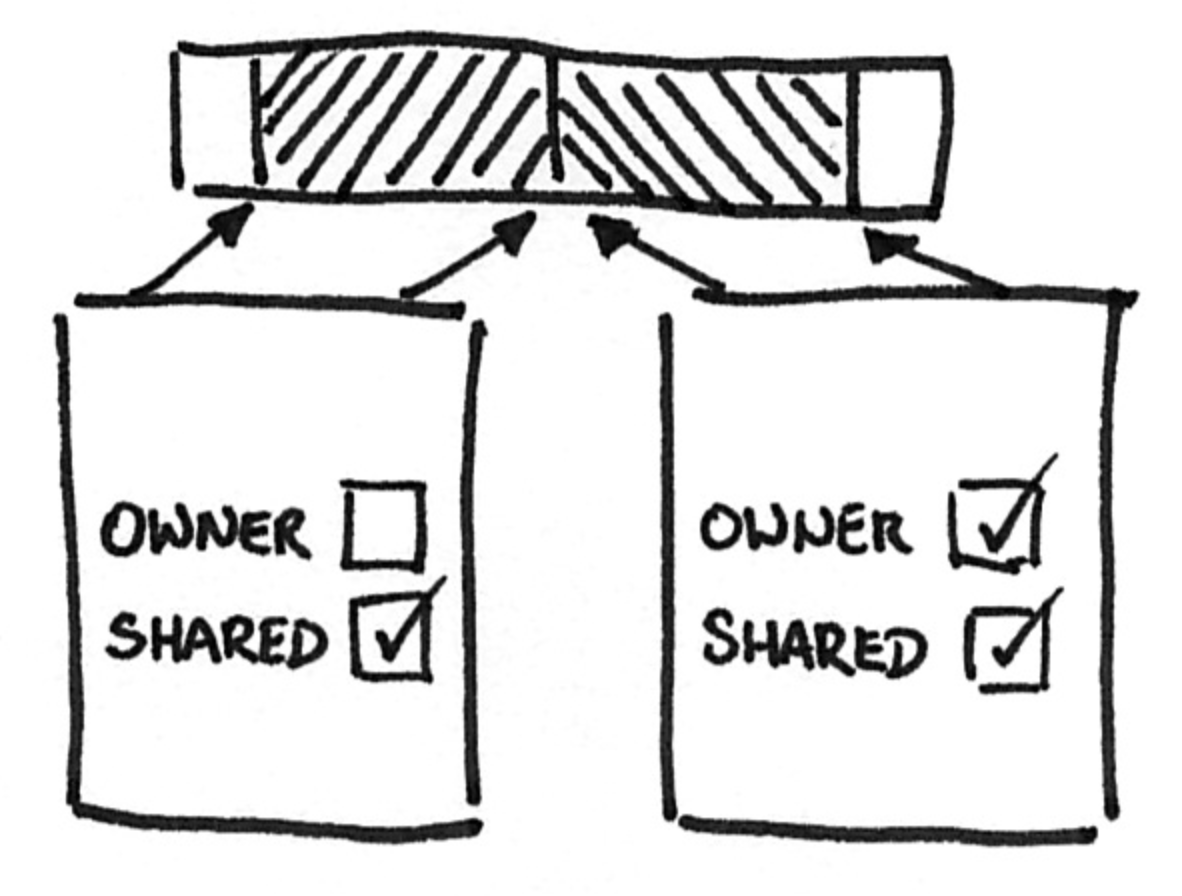
Sinks 和 Sources 通常连接到一个管道中。 我们不需要像 java.io 缓冲流那样在这些管道组件之间复制数据。 所有的 Sources 和 Sink 在幕后都使用缓存，而缓存将数据保存在 Segment 中，所以通常你可以从一个缓存中取出整个 Segment 并将其移动到另一个。



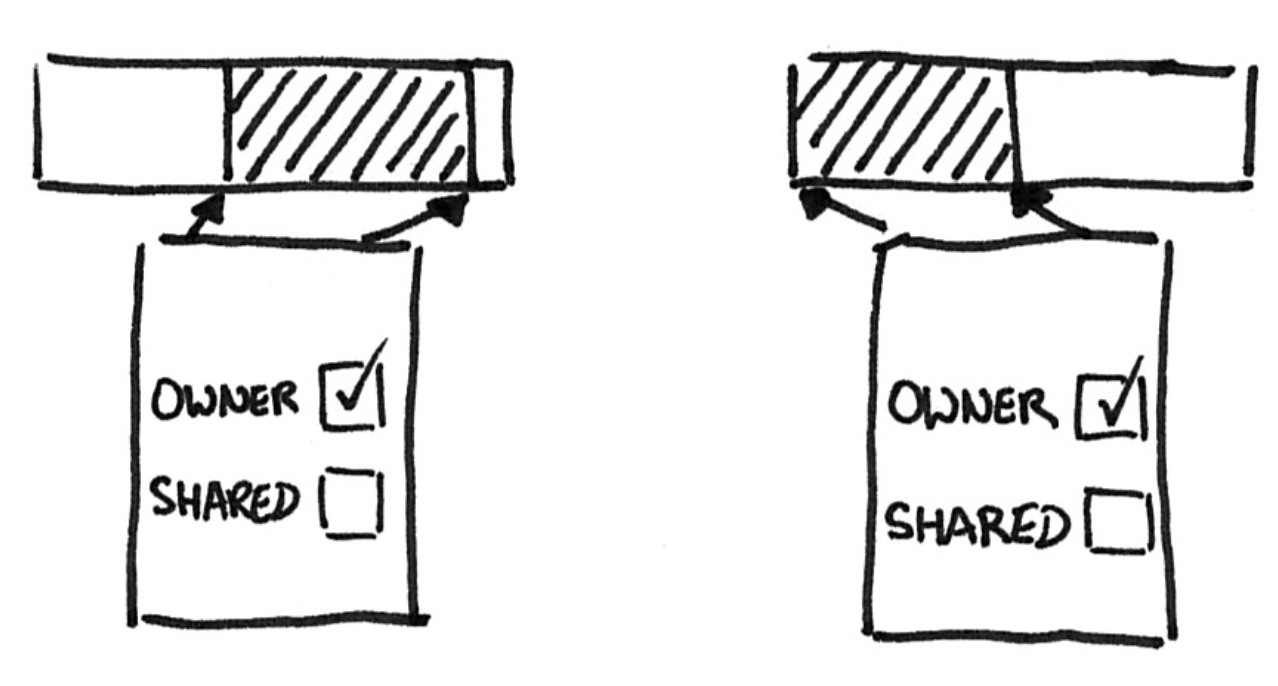
但是如果不想转移整个 Segment 怎么办？ Okio 使用另一个技巧来限制 CPU 密集型 ByteArray 复制：它在多个 Segment 之间共享数据缓冲区。 这就是 pos 和 limit “指针”发挥作用的地方。 它们在每个 Segment 使用的共享 ByteArray 中界定数据范围。 回到我们的用例：Okio 可能会创建另一个由相同 ByteArray 支持的 Segment，而不是将 ByteArray 的一部分复制到另一个 Segment，这比复制数据本身要轻得多。



在上述场景中，两个 Segment 都标有前面提到的shared 标志。 这样 Okio 就知道数据被多个 Segment 使用，并且它不能执行某些操作。 例如，不能写入“header”Segment，因为会覆盖“trailer”Segment 中开头的数据。 另一方面，应该可以写入超过共享 ByteArray 的最后使用的区域，但需要确保只允许一个 Segment 这样做。 这就是owner标志的用途。 Segment 使用的每个 ByteArray 都由一个 Segment 拥有。

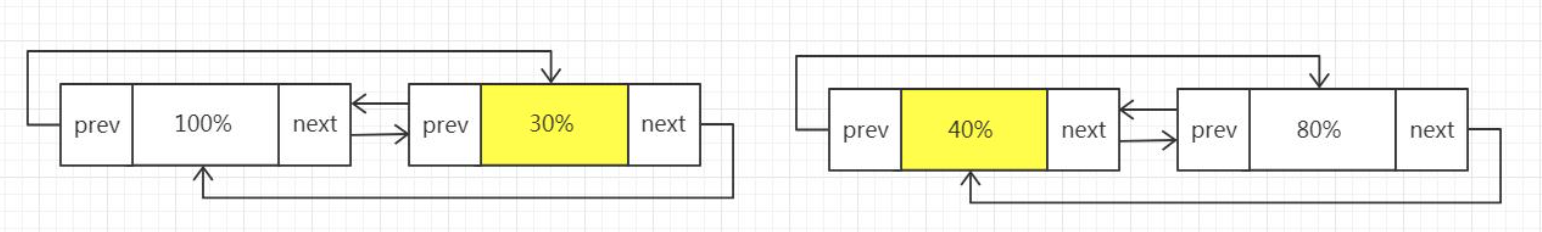


在拥有者段上执行另外一种写操作：数据移位。 想象一下，你正在使用 ByteArray 的后半部分，并且想要将更多数据写入 Segment。 如果 Segment 未共享，则可以将现有数据移动到 ByteArray 的开头，然后追加新数据。 如果 ByteArray 与其他 Segment 共享，则不能执行此操作，因为 ByteArray 的前半部分可能被其他 Segment 使用。

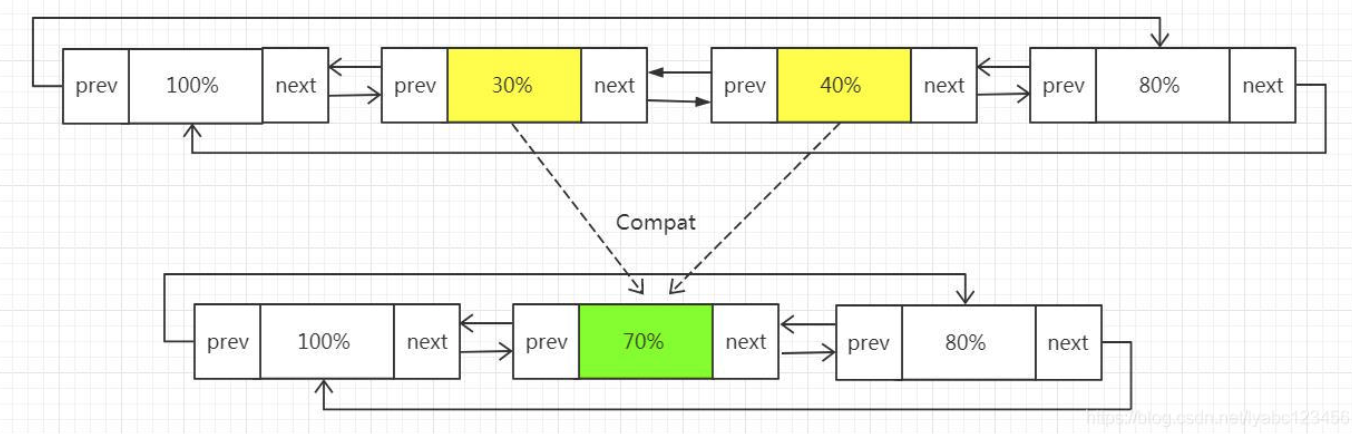


Segment 的合并过程：

假设初始两个 Buffer 中的 Segment 链表如下：



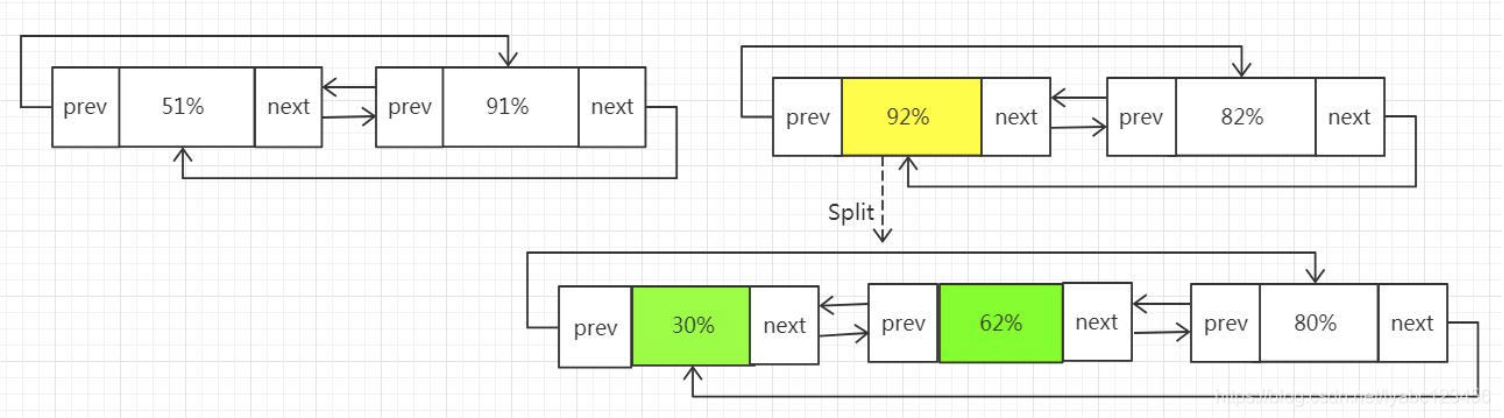
现在将第二个 Buffer 完全写入到第一个 Buffer：



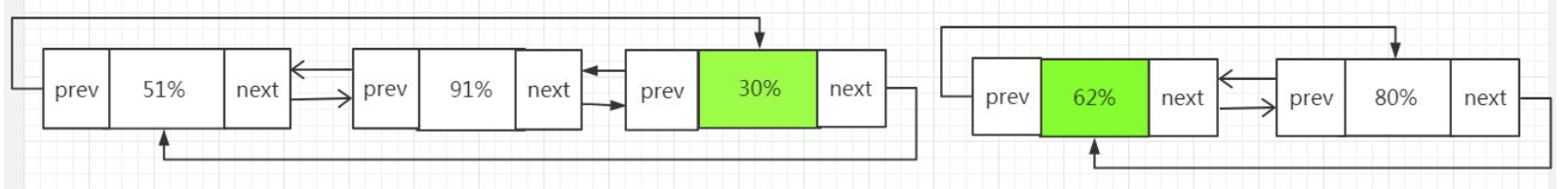
首先，它会直接将第二个 Buffer 的头节点连接到第一个 Buffer 的链尾，然后尝试将链尾的两个 Segment 进行合并，如果合并成功，则在合并之后，图中 40% 的那个 Segment 会被SegmentPool 回收，它的数据完全写入到 30% 的那个 Segment 中，最终生成一个70% 的 Segment，这样就达到了节约内存的目标。

Segment 的拆分过程：

假设初始两个 Buffer 中的 Segment 链表如下：



现在要从第二个Buffer中取前30%的数据写入到第一个Buffer当中，那么首先会将第二个Buffer的头结点Segment进行分割，分割为两个负载为30%和62%的Segment, 接下来移动这个新的30%的Segment节点到第一个Buffer的链表的尾部：



这样就完成了从第二个Buffer取30%的数据写入到第一个Buffer当中的工作。

超时机制：

Okio的亮点之一就是增加了超时机制，防止因为意外导致I/O一直阻塞的问题，默认的超时机制是同步的。AsyncTimeout是Okio中异步超时机制的实现，它是一个单链表，结点按等待时间从小到大排序，head是一个头结点，起占位作用。使用了一个WatchDog的后台线程来不断的遍历所有节点，如果某个节点超时就会将该节点从链表中移除，并关闭Socket。

AsyncTimeout提供了3个方法enter、exit、timeout，分别用于流操作开始、结束、超时三种情况调用。

生产者/消费者模型：

在 Okio 中可以使用 Pipe 来实现一个生产者/消费者模型。Pipe 维护了一个一定大小 Buffer。当该 Buffer 容量达到最大时，线程就会等待直到该 Buffer 有剩余的空间。

pipe 的实现原理，其实就是内部维护了一个 buffer 用来存储数据，可读可写；

当读数据的时候，就会减少 buffer 里面数据的容量，同时通过 buffer.notifyAll() 告诉外界状态发生了变化；

当写数据的时候，只能添加最大容量的数据，写好数据之后，就会通过 buffer.notifyAll() 告诉外界状态发生了变化；