43 | bufio包中的数据类型（下）

2018-11-19 郝林

你好，我是郝林，我今天继续分享bufio包中的数据类型。

在上一篇文章中，我提到了bufio包中的数据类型主要有Reader、Scanner、Writer和ReadWriter。并着重讲到了bufio.Reader类型与bufio.Writer类型，今天，我们继续专注bufio.Reader的内容来进行学习。

知识扩展

问题：bufio.Reader类型读取方法有哪些不同？

bufio.Reader类型拥有很多用于读取数据的指针方法，**这里面有4个方法可以作为不同读取流程的代表，它们是：Peek、Read、ReadSlice和ReadBytes。**

Reader值的Peek方法的功能是：读取并返回其缓冲区中的n个未读字节，并且它会从已读计数代表的索引位置开始读。

在缓冲区未被填满，并且其中的未读字节的数量小于n的时候，该方法就会调用fill方法，以启动缓冲区填充流程。但是，如果它发现上次填充缓冲区的时候有错误，那就不会再次填充。

如果调用方给定的n比缓冲区的长度还要大，或者缓冲区中未读字节的数量小于n，那么Peek方法就会把“所有未读字节组成的序列”作为第一个结果值返回。

同时，它通常还会把

> “`bufio.ErrBufferFull`变量的值（以下简称缓冲区已满的错误）”

作为第二个结果值返回，用来表示：虽然缓冲区被压缩和填满了，但是仍然满足不了要求。

只有在上述的情况都没有出现时，Peek方法才能返回：“以已读计数为起始的n个字节”和“表示未发生任何错误的nil”。

bufio.Reader类型的Peek方法有一个鲜明的特点，那就是：即使它读取了缓冲区中的数据，也不会更改已读计数的值。

这个类型的其他读取方法并不是这样。就拿该类型的Read方法来说，它有时会把缓冲区中的未读字节，依次拷贝到其参数p代表的字节切片中，并立即根据实际拷贝的字节数增加已读计数的值。

在缓冲区中还有未读字节的情况下，该方法的做法就是如此。不过，在另一些时候，其所属值的已读计数会等于已写计数，这表明：此时的缓冲区中已经没有任何未读的字节了。

当缓冲区中已无未读字节时，Read方法会先检查参数p的长度是否大于或等于缓冲区的长度。如果是，那么该方法会索性放弃向缓冲区中填充数据，转而直接从其底层读取器中读出数据并拷贝到p中。

这意味着它完全跨过了缓冲区，并直连了数据供需的双方。需要注意的是，Peek方法在遇到类似情况时的做法与这里的区别。

这两种做法孰优孰劣还要看具体的使用场景。Peek方法会在条件满足时填充缓冲区，并在发现参数n的值比缓冲区的长度更大时，直接返回缓冲区中的所有未读字节。

如果我们当初设定的缓冲区长度很大，那么在这种情况下的方法执行耗时，就有可能会比较长。最主要的原因是填充缓冲区需要花费较长的时间。

由fill方法执行的流程可知，它会尽量填满缓冲区中的可写空间。然而，Read方法在大多数的情况下，是不会向缓冲区中写入数据的，尤其是在前面描述的那种情况下，即：缓冲区中已无未读字节，且参数p的长度大于或等于缓冲区的长度。

此时，该方法会直接从底层读取器那里读出数据，所以数据的读出速度就成为了这种情况下方法执行耗时的决定性因素。

当然了，我在这里说的只是耗时操作在某些情况下更可能出现在哪里，一切的结论还是要以性能测试的客观结果为准。

说回Read方法的内部流程。如果缓冲区中已无未读字节，但其长度比参数p的长度更大，那么该方法会先把已读计数和已写计数的值都重置为0，然后再尝试着使用从底层读取器那里获取的数据，对缓冲区进行一次从头至尾的填充。

不过要注意，这里的尝试只会进行一次。无论在这一时刻是否能够获取到数据，也无论获取时是否有错误发生，都会是如此。而fill方法的做法与此不同，只要没有发生错误，它就会进行多次尝试，因此它真正获取到一些数据的可能性更大。

不过，这两个方法有一点是相同，那就是：只要它们把获取到的数据写入缓冲区，就会及时地更新已写计数的值。

再来说ReadSlice方法和ReadBytes方法。这两个方法的功能总体上来说都是持续地读取数据，直至遇到调用方给定的分隔符为止。

ReadSlice方法会先在其缓冲区的未读部分中寻找分隔符。如果未能找到，并且缓冲区未满，那么该方法会先通过调用fill方法对缓冲区进行填充，然后再次寻找，如此往复。

如果在填充的过程中发生了错误，那么它会把缓冲区中的未读部分作为结果返回，同时返回相应的错误值。

注意，在这个过程中有可能会出现虽然缓冲区已被填满，但仍然没能找到分隔符的情况。这时，ReadSlice方法会把整个缓冲区（也就是buf字段代表的字节切片）作为第一个结果值，并把缓冲区已满的错误（即bufio.ErrBufferFull变量的值）作为第二个结果值。

经过fill方法填满的缓冲区肯定从头至尾都只包含了未读的字节，所以这样做是合理的。

当然了，一旦ReadSlice方法找到了分隔符，它就会在缓冲区上切出相应的、包含分隔符的字节切片，并把该切片作为结果值返回。无论分隔符找到与否，该方法都会正确地设置已读计数的值。

比如，在返回缓冲区中的所有未读字节，或者代表全部缓冲区的字节切片之前，它会把已写计数的值赋给已读计数，以表明缓冲区中已无未读字节。

如果说ReadSlice是一个容易半途而废的方法的话，那么可以说ReadBytes方法算得上是相当的执着。ReadBytes方法会通过调用ReadSlice方法一次又一次地从缓冲区中读取数据，直至找到分隔符为止。

在这个过程中，ReadSlice方法可能会因缓冲区已满而返回所有已读到的字节和相应的错误值，但ReadBytes方法总是会忽略掉这样的错误，并再次调用ReadSlice方法，这使得后者会继续填充缓冲区并在其中寻找分隔符。

除非ReadSlice方法返回的错误值并不代表缓冲区已满的错误，或者它找到了分隔符，否则这

一过程永远不会结束。

如果寻找的过程结束了，不管是不是因为找到了分隔符，ReadBytes方法都会把在这个过程中读到的所有字节，按照读取的先后顺序组装成一个字节切片，并把它作为第一个结果值。如果过程结束是因为出现错误，那么它还会把拿到的错误值作为第二个结果值。

在bufio.Reader类型的众多读取方法中，依赖ReadSlice方法的除了ReadBytes方法，还有ReadLine方法。不过后者在读取流程上并没有什么特别之处，我就不在这里赘述了。另外，该类型的ReadString方法完全依赖于ReadBytes方法，前者只是在后者返回的结果值之上做了一个简单的类型转换而已。

最后，我还要提醒你一下，有个安全性方面的问题需要你注意。bufio.Reader类型的Peek方法、ReadSlice方法和ReadLine方法都有可能会造成内容泄露。这主要是因为它们在正常的情况下都会返回直接基于缓冲区的字节切片。我在讲bytes.Buffer类型的时候解释过什么叫内容泄露。你可以返回查看。

调用方可以通过这些方法返回的结果值访问到缓冲区的其他部分，甚至修改缓冲区中的内容。这通常都是很危险的。

总结

我们用比较长的篇幅介绍了bufio包中的数据类型，其中的重点是bufio.Reader类型。

bufio.Reader类型代表的是携带缓冲区的读取器。它的值在被初始化的时候需要接受一个底层的读取器，后者的类型必须是io.Reader接口的实现。

Reader值中的缓冲区其实就是一个数据存储中介，它介于底层读取器与读取方法及其调用方之间。此类值的读取方法一般都会先从该值的缓冲区中读取数据，同时在必要的时候预先从其底层读取器那里读出一部分数据，并填充到缓冲区中以备后用。填充缓冲区的操作通常会由该值的fill方法执行。在填充的过程中，fill方法有时还会对缓冲区进行压缩。

在Reader值拥有的众多读取方法中，有4个方法可以作为不同读取流程的代表，它们是：Peek、Read、ReadSlice和ReadBytes。

Peek方法的特点是即使读取了缓冲区中的数据，也不会更改已读计数的值。而Read方法会在参数值的长度过大，且缓冲区中已无未读字节时，跨过缓冲区并直接向底层读取器索要数据。

ReadSlice方法会在缓冲区的未读部分中寻找给定的分隔符，并在必要时对缓冲区进行填充。

如果在填满缓冲区之后仍然未能找到分隔符，那么该方法就会把整个缓冲区作为第一个结果值返回，同时返回缓冲区已满的错误。

ReadBytes方法会通过调用ReadSlice方法，一次又一次地填充缓冲区，并在其中寻找分隔符。除非发生了未预料到的错误或者找到了分隔符，否则这一过程将会一直进行下去。

Reader值的ReadLine方法会依赖于它的ReadSlice方法，而其ReadString方法则完全依赖于ReadBytes方法。

另外，值得我们特别注意的是，Reader值的Peek方法、ReadSlice方法和ReadLine方法都可能会造成其缓冲区中的内容的泄露。

最后再说一下bufio.Writer类型。把该类值的缓冲区中暂存的数据写进其底层写入器的功能，主要是由它的Flush方法实现的。

此类值的所有数据写入方法都会在必要的时候调用它的Flush方法。一般情况下，这些写入方法都会先把数据写进其所属值的缓冲区，然后再增加该值中的已写计数。但是，在有些时候，Write方法和ReadFrom方法也会跨过缓冲区，并直接把数据写进其底层写入器。

请记住，虽然这些写入方法都会不时地调用Flush方法，但是在写入所有的数据之后再显式地调用一下这个方法总是最稳妥的。

思考题

今天的思考题是：bufio.Scanner类型的主要功用是什么？它有哪些特点？

感谢你的收听，我们下期再见。

戳此查看Go语言专栏文章配套详细代码。