

## 41 | io包中的接口和工具（下）

2018-11-14 郝林



上一篇文章中，我主要讲到了`io.Reader`的扩展接口和实现类型。当然，`io`代码包中的核心接口不止`io.Reader`一个。

我们基于它引出的一条主线只是`io`包类型体系中的一部分。我们很有必要再从另一个角度去探索一下，以求对`io`包有更加全面的了解。下面的一个问题就与此有关。

### 知识扩展

**问题：**`io`包中的接口都有哪些？它们之间都有着怎样的关系？

我们可以把没有嵌入其他接口并且只定义了一个方法的接口叫做简单接口。在`io`包中，这样的接口一共有**11**个。

在它们之中，有的接口有着众多的扩展接口和实现类型，我们可以称之为核心接口。**`io`包中的核心接口只有3个，它们是：`io.Reader`、`io.Writer`和`io.Closer`。**

我们还可以把`io`包中的简单接口分为四大类。这四大类接口分别针对于四种操作，即：读取、写入、关闭和读写位置设定。前三种操作属于基本的**I/O**操作。

关于读取操作，我们在前面已经重点讨论过核心接口`io.Reader`。它在`io`包中有**5**个扩展接口，并有**6**个实现类型。除了它，这个包中针对读取操作的接口还有不少。我们下面就来梳理一下。

首先来看`io.ByteReader`和`io.RuneReader`这两个简单接口。它们分别定义了一个读取方法，即：`ReadByte`和`ReadRune`。

但与`io.Reader`接口中`Read`方法不同的是，这两个读取方法分别只能够读取下一个单一的字节和**Unicode**字符。

我们之前讲过的数据类型`strings.Reader`和`bytes.Buffer`都是`io.ByteReader`和`io.RuneReader`的实现类型。

不仅如此，这两个类型还都实现了`io.ByteScanner`接口和`io.RuneScanner`接口。

`io.ByteScanner`接口内嵌了简单接口`io.ByteReader`，并定义了额外的`UnreadByte`方法。如此一来，它就抽象出了一个能够读取和读回退单个字节的功能集。

与之类似，`io.RuneScanner`内嵌了简单接口`io.RuneReader`，并定义了额外的`UnreadRune`方法。它抽象的是可以读取和读回退单个**Unicode**字符的功能集。

再来看`io.ReaderAt`接口。它也是一个简单接口，其中只定义了一个方法`ReadAt`。与我们在前面说过的读取方法都不同，`ReadAt`是一个纯粹的只读方法。

它只去读取其所属值中包含的字节，而不对这个值进行任何的改动，比如，它绝对不能去修改已读计数的值。这也是`io.ReaderAt`接口与其实现类型之间最重要的一个约定。

因此，如果仅仅并发地调用某一个值的`ReadAt`方法，那么安全性应该是可以得到保障的。

另外，还有一个读取操作相关的接口我们没有介绍过，它就是`io.WriterTo`。这个接口定义了一个名为`WriteTo`的方法。

千万不要被它的名字迷惑，这个`WriteTo`方法其实是一个读取方法。它会接受一个`io.Writer`类型的参数值，并会把其所属值中的数据读出并写入到这个参数值中。

与之相对应的是`io.ReaderFrom`接口。它定义了一个名叫`ReadFrom`的写入方法。该方法会接受一个`io.Reader`类型的参数值，并会从该参数值中读出数据,并写入到其所属值中。

值得一提的是，我们在前面用到过的`io.CopyN`函数，在复制数据的时候会先检测其参数`src`的值是否实现了`io.WriterTo`接口。如果是，那么它就直接利用该值的`WriteTo`方法，把其中的数据拷贝给参数`dst`代表的值。

类似的，这个函数还会检测`dst`的值是否实现了`io.ReaderFrom`接口。如果是，那么它就会利用这个值的`ReadFrom`方法，直接从`src`那里把数据拷贝进该值。

实际上，对于`io.Copy`函数和`io.CopyBuffer`函数来说也是如此，因为它们在内部做数据复

制的时候用的都是同一套代码。

你也看到了，`io.ReaderFrom`接口与`io.WriterTo`接口对应得很规整。实际上，在`io`包中，与写入操作有关的接口都与读取操作的相关接口有着一定的对应关系。下面，我们就来说一说前者。

首先当然是核心接口`io.Writer`。基于它的扩展接口除了有我们已知的`io.ReadWriter`、`io.ReadWriteCloser`和`io.ReadWriteSeeker`之外，还有`io.WriteCloser`和`io.WriteSeeker`。

我们之前提及的`*io.pipe`就是`io.ReadWriter`接口的实现类型。然而，在`io`包中并没有`io.ReadWriteCloser`接口的实现，它的实现类型主要集中在`net`包中。

除此之外，写入操作相关的简单接口还有`io.ByteWriter`和`io.WriterAt`。可惜，`io`包中也没有它们的实现类型。不过，有一个数据类型值得在这里提一句，那就是`*os.File`。

这个类型不但是`io.WriterAt`接口的实现类型，还同时实现了`io.ReadWriteCloser`接口和`io.ReadWriteSeeker`接口。也就是说，该类型支持的I/O操作非常的丰富。

`io.Seeker`接口作为一个读写位置设定相关的简单接口，也仅仅定义了一个方法，名叫`Seek`。

我在讲`strings.Reader`类型的时候还专门说过这个`Seek`方法，当时还给出了一个与已读计数估算有关的例子。该方法主要用于寻找并设定下一次读取或写入时的起始索引位置。

`io`包中有几个基于`io.Seeker`的扩展接口，包括前面讲过的`io.ReadSeeker`和`io.ReadWriteSeeker`，以及还未曾提过的`io.WriteSeeker`。后者是基于`io.Writer`和`io.Seeker`的扩展接口。

我们之前多次提到的两个指针类型`strings.Reader`和`io.SectionReader`都实现了`io.Seeker`接口。顺便说一句，这两个类型也都是`io.ReaderAt`接口的实现类型。

最后，关闭操作相关的接口`io.Closer`非常通用，它的扩展接口和实现类型都不少。我们单从名称上就能够一眼看出`io`包中的哪些接口是它的扩展接口。至于它的实现类型，`io`包中只有`io.PipeReader`和`io.PipeWriter`。

## 总结

我们来总结一下这两篇的内容。在Go语言中，对接口的扩展是通过接口类型之间的嵌入来实现的，这也常被叫做接口的组合。而`io`代码包恰恰就可以作为接口扩展的一个标杆，它可以成为我们运用这种技巧时的一个参考标准。

在本文中，我根据接口定义的方法的数量以及是否有接口嵌入，把io包中的接口分为了简单接口和扩展接口。

同时，我又根据这些简单接口的扩展接口和实现类型的数量级，把它们分为了核心接口和非核心接口。

在io包中，称得上核心接口的简单接口只有**3**个，

即：`io.Reader`、`io.Writer`和`io.Closer`。这些核心接口在Go语言标准库中的实现类型都在**200**个以上。

另外，根据针对的I/O操作的不同，我还把简单接口分为了四大类。这四大类接口针对的操作分别是：读取、写入、关闭和读写位置设定。

其中，前三种操作属于基本的I/O操作。基于此，我带你梳理了每个类别的简单接口，并讲解了它们在io包中的扩展接口，以及具有代表性的实现类型。

除此之外，我还从多个维度为你描述了一些重要程序实体的功用和机理，比如：数据段读取器`io.SectionReader`、作为同步内存管道核心实现的`io.pipe`类型，以及用于数据拷贝的`io.CopyN`函数，等等。

我如此详尽且多角度的阐释，正是为了让你能够记牢io代码包中有着网状关系的接口和数据类型。我希望这个目的已经达到了，最起码，本文可以作为你深刻记忆它们的开始。

最后再强调一下，io包中的简单接口共有**11**个。其中，读取操作相关的接口有**5**个，写入操作相关的接口有**4**个，而与关闭操作有关的接口只有**1**个，另外还有一个读写位置设定相关的接口。

此外，io包还包含了**9**个基于这些简单接口的扩展接口。你需要在今后思考和实践的是，你在什么时候应该编写哪些数据类型实现io包中的哪些接口，并以此得到最大的好处。

## 思考题

今天的思考题是：io包中的同步内存管道的运作机制是什么？

[戳此查看Go语言专栏文章配套详细代码。](#)

# GO语言核心36讲

3个月带你通关 GO 语言

郝林

《Go 并发编程实战》作者  
GoHackers 技术社群发起人  
前轻松筹大数据负责人

