io.Reader и io.Writer в Go

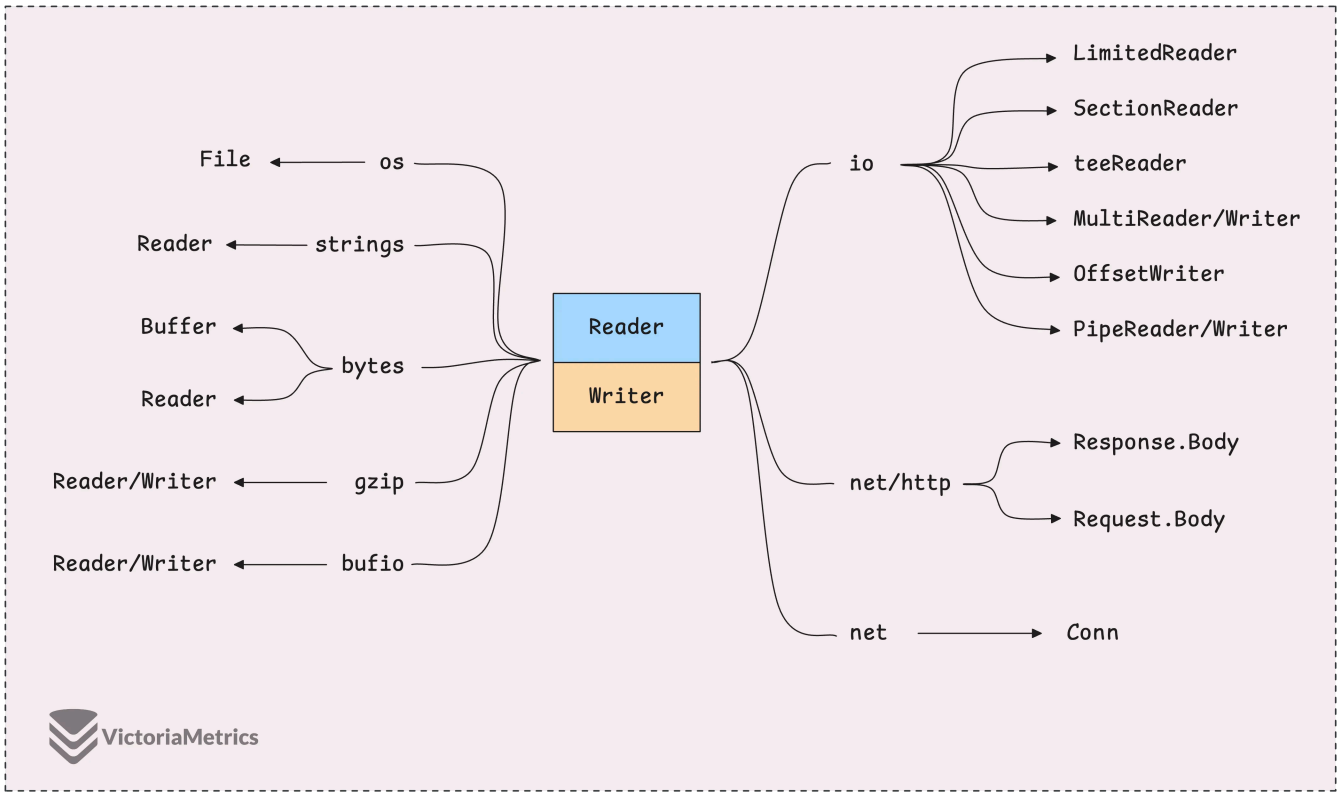
февраля 2, 2025 · 10 минут · German Gorelkin

Перевод/заметки Go I/O Readers, Writers, and Data in Motion

Интерфейсы `io.Reader` и `io.Writer` являются одними из самых часто используемых инструментов в процессе работы с вводом и выводом данных.

Существует множество конкретных реализаций этих интерфейсов, каждая из которых предназначена для выполнения различных задач, таких как чтение из

файлов, сетей, буферов или даже сжатых данных.



Что такое `io.Reader`?

`io.Reader` - это очень простой интерфейс, у него всего один метод:

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

Когда вы вызываете метод `Read`, он начинает заполнять слайс `p` данными из некоторого источника.

Этот источник может быть любым: файлом, сетевым соединением или даже обычной строкой. Однако `io.Reader` не обращает внимания на происхождение данных. Его цель — скопировать эти данные в предоставленный вами слайс.

`Read` не гарантирует, что весь слайс будет заполнен. Он возвращает количество фактически прочитанных данных, которое обозначается как `n`. Если больше нет данных для чтения, метод вернет ошибку `io.EOF`, что означает, что вы достигли конца потока.

> *Может ли он возвращать одновременно `n` (прочитанные байты) и `err` (ошибку)?*

Да, иногда метод `Read` возвращает не нулевое количество байт (то есть `n>0`) и ошибку.

Главный совет в этой ситуации — всегда сначала обрабатывайте прочитанные байты (если `n>0`), даже если есть ошибка.

Ошибка может быть не `io.EOF` — она может быть связана с другой проблемой, возникшей после чтения части данных. Поэтому вы всё ещё можете получить корректные данные, и вам не стоит упускать их, сразу переходя к обработке ошибки.

> *Почему `Reader` не возвращает данные, а заполняет переданный байтовый слайс?*

Передавая предварительно созданный слайс в метод `Read`, Go предоставляет вам больше возможностей для управления.

Вы сами определяете размер слайса и решаете, где будут храниться данные. Если бы `Reader` каждый раз возвращал новый слайс, это привело бы к множеству ненужных выделений памяти, что замедлило бы работу и привело к неоправданным затратам ресурсов.

`os.File` ЭТО `Reader`

Чтобы прочитать содержимое файла, вы можете воспользоваться функцией `os.Open(...)`. Эта функция открывает файл и возвращает объект типа `*os.File`, который реализует интерфейс `io.Reader`.

```
f, err := os.Open("test.txt")
if err != nil {
    panic(err)
}
defer f.Close() // no error handling
```

Как только вы получили `*os.File`, вы можете обращаться с ним как с любым другим `Reader`. Вы считываете его содержимое в буфер и продолжаете читать, пока не получите `io.EOF`.

```
// Make a buffer to store the data
for {
    // Read from file into buffer
    n, err := f.Read(buf)

    // If bytes were read (n > 0), process them
    if n > 0 {
        fmt.Println("Received", n, "bytes:", string(buf[:n]))
    }

    // Check for errors, but handle EOF properly
    if err != nil {
        if err == io.EOF {
            break // End of file, stop reading
        }
        panic(err) // Handle other potential errors
    }
}
```

Вы можете попробовать уменьшить размер буфера (с 1024 байта до 16, 32 и т.д.) и посмотреть, как это повлияет на вывод.

Подводные камни `io.ReadAll` и `io.Copy`

То, как мы только что читали файл, очень похоже на работу популярного метода `io.ReadAll` .

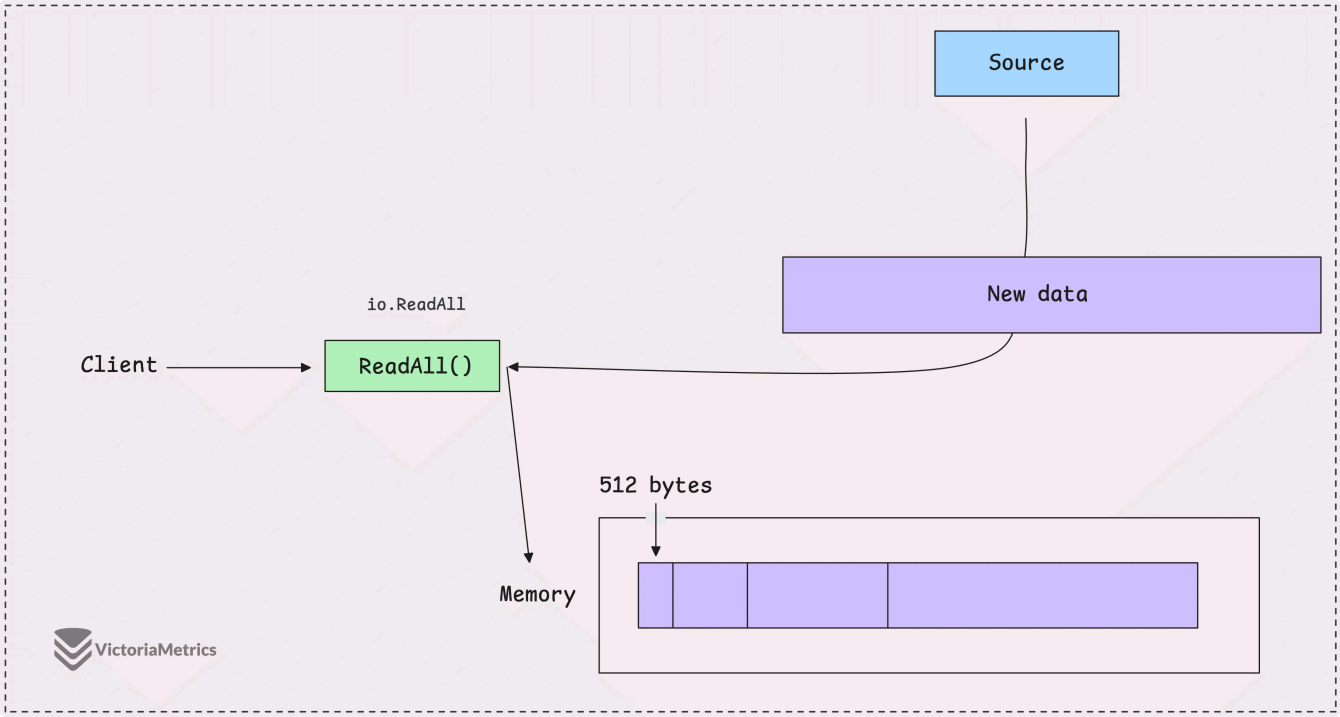
Вы, вероятно, использовали его, когда вам нужно было получить все данные сразу. Например, чтобы прочитать всё тело HTTP-ответа, загрузить весь файл или просто получить все данные из потока.

```
func main() {
    f, err := os.Open("test.txt")
    if err != nil {
        panic(err)
    }
    defer f.Close() // no error handling

    body, err := io.ReadAll(f)
    ...
}
```

`io.ReadAll` — это очень удобный метод, который скрывает от вас все детали, связанные с чтением данных, и автоматически увеличивает байтовый срез.

Он начинает с буфера размером 512 байт, и если данных больше, то буфер автоматически расширяется с помощью функции `append()`.

Несмотря на удобство, есть одна серьёзная проблема: нет ограничений на объём считываемых данных.



Если вы попытаетесь вызвать `io.ReadAll` для очень большого потока данных, например, для огромного файла или HTTP-ответа, который значительно превышает ожидания, функция будет продолжать читать и выделять память до тех пор, пока либо не завершится чтение, либо не закончится память в системе.

Например, если вы хотите подсчитать количество раз, когда в файле встречается буква `a`, использование `io.ReadAll` для предварительного прочтения всего файла, а затем подсчет числа вхождений `a` будет излишним.

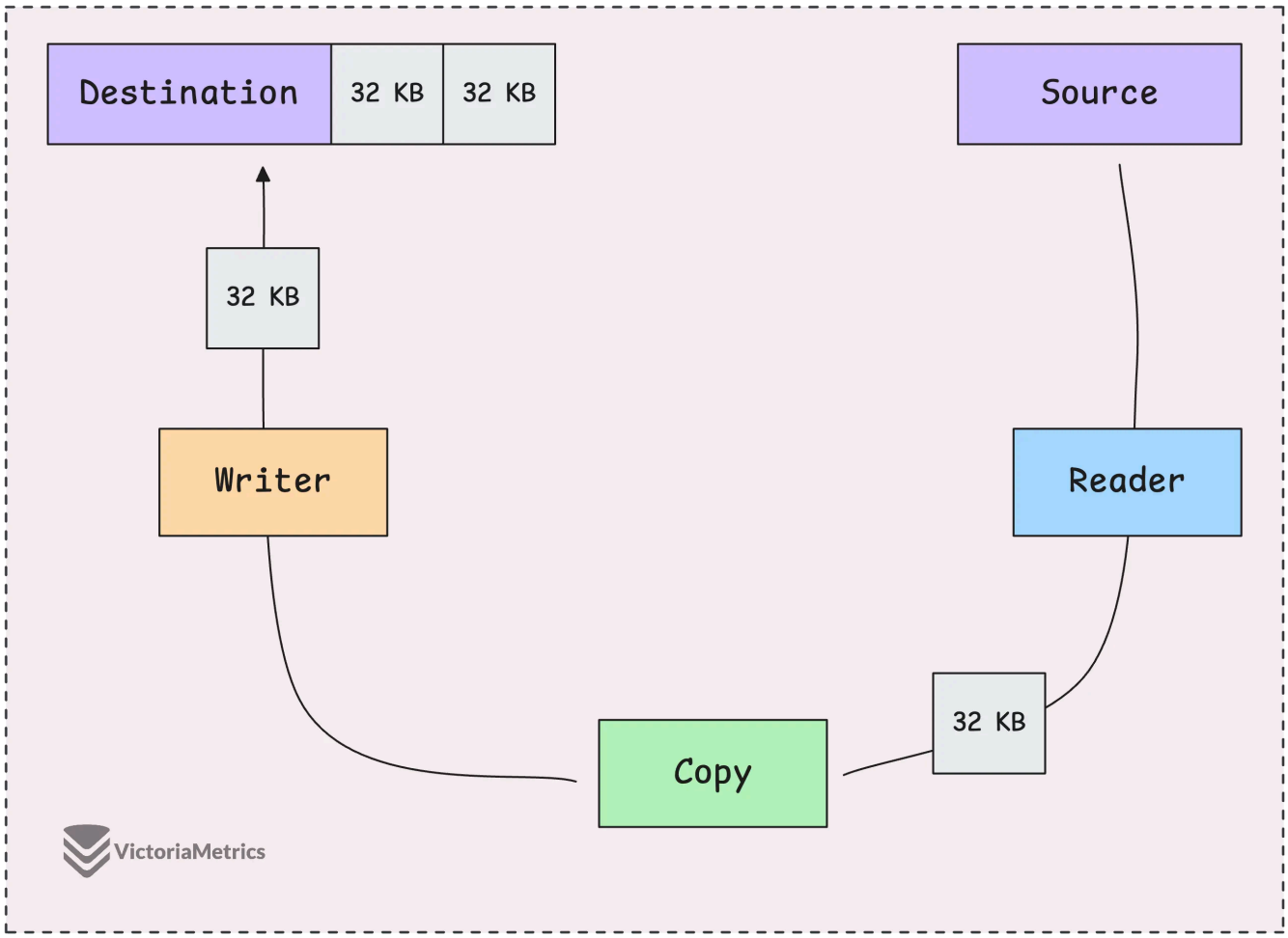
В таких случаях `io.ReadAll` не является оптимальным выбором. Гораздо эффективнее будет обрабатывать данные по мере их поступления, используя потоковую передачу или другие методы.

Если вы сталкиваетесь с задачами, связанными с передачей данных между системами, пересылкой тела HTTP-запроса, чтением файла и его отправкой по сети или скачиванием и сохранением файлов, у вас есть отличный

инструмент — `io.Copy`. Он станет вашим незаменимым помощником в таких ситуациях.

```
func Copy(dst Writer, src Reader) (written int64, err error) { ... }
```

`io.Copy` — это превосходный метод, который использует фиксированный буфер размером 32 килобайта для эффективной передачи данных.



Вместо того чтобы загружать весь файл в память целиком, он считывает данные небольшими фрагментами по 32 КБ и записывает каждый фрагмент сразу в место назначения, не увеличивая буфер. Благодаря этому объём используемой памяти остаётся небольшим, вне зависимости от размера обрабатываемых данных.

Другие реализации `io.Reader`

Существует множество различных реализаций `io.Reader`, но давайте рассмотрим несколько наиболее распространённых. Например, `strings.NewReader` позволяет работать со строкой как с потоком данных, подобно тому, как мы поступаем с файлом или сетью:

```
r := strings.NewReader("Hello, World!")
```

Это отличный вариант, когда вам нужно сымитировать чтение из потока, например для тестирования, но источник при этом остается статичным, таким как строка. Это особенно полезно, когда вы хотите интегрировать его в API или функции, ожидающие `io.Reader`.

Еще один важный элемент - `http.Response.Body`, который представляет собой `io.ReadCloser`.

В нем хранится тело HTTP-ответа, и что важно, это не просто `io.Reader`, но и `Closer`. Это означает, что вам нужно явно закрыть его после завершения чтения, чтобы все ресурсы, связанные с телом ответа, были освобождены.

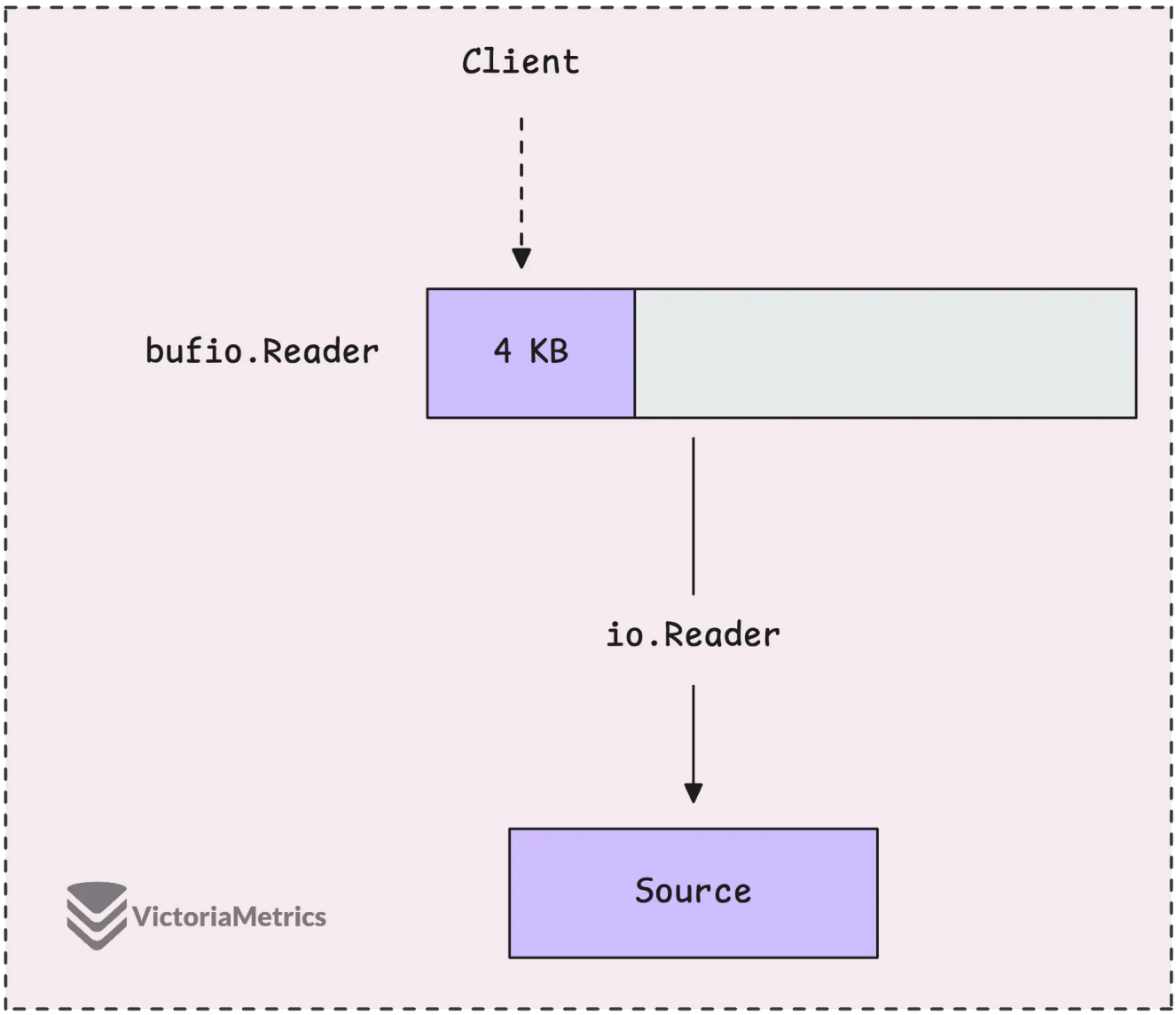
```
resp, err := http.Get("https://example.com")
if err != nil {
    panic(err)
}
defer resp.Body.Close()

r := resp.Body
// Usually, you'd use io.ReadAll to read the full body
// body, err := io.ReadAll(r)
```

`http.Client` в Go использует постоянные соединения (*keep-alive*), то есть пытается повторно использовать одно и то же TCP-соединение для нескольких запросов к одному и тому же серверу. Но если вы не прочитаете и не закроете тело ответа, это соединение не сможет быть использовано повторно. Поэтому крайне важно убедиться, что тело ответа полностью прочитано и закрыто, как только вы закончили с ним работу.

Еще один полезный инструмент — это `bufio.Reader`. Он разработан для того, чтобы обернуть существующий `io.Reader` и сделать работу более эффективной благодаря буферизации входных данных.

```
r := bufio.NewReader(f)
```

Когда вы используете `bufio.Reader`, он не обращается к основному источнику данных каждый раз, когда вы вызываете `reader.Read`.



Вместо этого он заранее считывает большой фрагмент данных и сохраняет его в буфере (по умолчанию размер буфера составляет 4КБ). Затем, каждый раз, когда вы запрашиваете данные, он предоставляет их из буфера. Это значительно сокращает время, затрачиваемое читателем на взаимодействие с исходным источником данных.

Как только буфер исчерпывается, он запрашивает новый фрагмент из источника. Если вы запрашиваете больше данных, чем может вместить буфер, `bufio.Reader` может просто пропустить буфер и читать непосредственно из источника.

Конечно, вы можете изменять размер буфера по своему усмотрению, он не ограничен 4 КБ:

```
r := bufio.NewReaderSize(f, 32 * 1024)
```

Суть `bufio.Reader` заключается в том, чтобы уменьшить частоту обращения к источнику данных за счет кэширования данных в памяти.

После изучения всех вышеперечисленных ридеров, я думаю, вы уже достаточно хорошо поняли, как работает `io.Reader`. Поэтому давайте быстро рассмотрим другие полезные типы ридеров:

- `compress/gzip.Reader`: Считывает и распаковывает данные gzip, а также проверяет целостность данных с помощью контрольных сумм и проверки размера.
- `encoding/base64.NewDecoder`: Декодер base64 также выполняет функции ридера. Он принимает закодированный входной поток и, разделяя его на части, декодирует каждую из них. Каждые четыре байта, представленные в формате base64, преобразуются в три байта необработанных данных, которые затем помещаются в предоставленный слайс.
- `io.SectionReader`: Считайте, что это ридер, который фокусируется на определенном фрагменте данных в большом наборе данных. Вы задаете участок, и он считывает данные только из этого участка.
- `io.LimitedReader`: Накладывает ограничение на общий объём данных, который можно считать с базового устройства. Ограничение действует не только для одного чтения, но и для нескольких.
- `io.MultiReader`: Объединяет несколько экземпляров `io.Reader` в один, читая из них последовательно, как если бы все они были скомпонованы.
- `io.TeeReader`: Аналогично `io.Copy()`, но вместо того, чтобы копировать все данные сразу, позволяет вам решать, когда и сколько читать, копируя данные в другое место в режиме реального времени.
- `io.PipeReader`: Обеспечивает работу канала, в котором `PipeReader` получает данные, записанные `PipeWriter`. Процесс чтения блокируется до тех пор, пока не появятся новые данные для обработки.

Большинство этих ридеров оборачиваются вокруг другого `io.Reader` - будь то базовый `io.Reader` или что-то вроде `bufio.Reader` (который сам оборачивается вокруг `io.Reader`).

Если вы захотите создать свой собственный ридер, то в основном будете следовать той же схеме. Пример с ограничением параллелизма,

используемый в VictoriaMetrics:

```
type Reader struct {
    r                    io.Reader
    increasedConcurrency bool
}

// Read implements io.Reader.
//
// It increases concurrency after the first call or after the next call after
func (r *Reader) Read(p []byte) (int, error) {
    n, err := r.r.Read(p)
    if !r.increasedConcurrency {
        if !incConcurrency() {
            err = &httpserver.ErrorWithStatusCode{
                Err: fmt.Errorf("cannot process insert request for %.3f second
                    "Possible solutions: to reduce workload; to increase compu
                    "to increase -insert.maxQueueDuration; to increase -maxCon
                    maxQueueDuration.Seconds(), *maxConcurrentInserts),
                StatusCode: http.StatusServiceUnavailable,
            }
            return 0, err
        }
        r.increasedConcurrency = true
    }
    return n, err
}
```

Этот ридер оборачивается вокруг любого `io.Reader`, и его основная задача - ограничить количество одновременных операций чтения. Если лимит достигнут, он ставит дальнейшие операции чтения в очередь до тех пор, пока не появятся ресурсы или не наступит таймаут.

Что такое `io.Writer`?

Сигнатура метода `Write` очень похожа на сигнатуру метода `Read`:

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

Метод `Write` пытается записать содержимое байтового среза `p` в какое-либо заранее определенное место назначения, например в файл или сетевое

соединение.

Возвращаемое значение `n` показывает, сколько байт было записано. В идеале `n` должно быть равно `len(p)`, что означает, что весь фрагмент был записан. Однако это не всегда так. Если `n` меньше `len(p)`, то это свидетельствует о том, что была записана только часть данных, и ошибка `err` сообщит вам, что пошло не так.

Например `os.File`:

```
func main() {
    f, err := os.Open("test.txt")
    if err != nil {
        panic(err)
    }
    defer f.Close() // no error handling

    _, err = f.Write([]byte("Hello, World!"))
    if err != nil {
        panic(err)
    }
}

// panic: write test.txt: bad file descriptor
```

В этом случае ошибка `bad file descriptor` возникает из-за того, что `os.Open` открывает файл в режиме только для чтения. Это означает, что вы не можете записать в этот файл. Чтобы исправить проблему, вам нужно открыть файл в режиме, который позволяет производить запись.

Вы можете использовать `os.OpenFile`, который дает вам больше контроля над тем, как открывается файл:

```
f, err := os.OpenFile("test.txt", os.O_RDWR|os.O_CREATE, 0o644)
```

Теперь все будет работать как надо. Go добавит данные в начало файла и перезапишет существующее содержимое. Если же вы хотите добавить данные вместо перезаписи, вы можете указать опцию `os.O_APPEND`.

Ещё один полезный инструмент для работы с данными — `bufio.Writer`. Он функционирует подобно `bufio.Reader`, но предназначен для записи данных,

что может значительно повысить производительность за счёт уменьшения количества операций.

Он оборачивает `io.Writer` и буферизирует данные:

```
type Writer struct {
    err error
    buf []byte
    n   int
    wr  io.Writer
}
```

`bufio.Writer` не записывает данные непосредственно на устройство. Вместо этого он копирует их во внутренний буфер. Если буфер достаточно велик (по умолчанию его размер составляет 4 КБ), данные хранятся в буфере до тех пор, пока он не заполнится полностью, а затем записываются сразу все.

```
func main() {
    f, err := os.OpenFile("test.txt", os.O_RDWR|os.O_CREATE, 0o644)
    if err != nil {
        panic(err)
    }
    defer f.Close() // no error handling

    w := bufio.NewWriter(f)
    _, err = w.Write([]byte("Hello, World!"))
    if err != nil {
        panic(err)
    }
}
```

Обратите внимание, что `bufio.Writer` не будет записывать данные в файл до тех пор, пока буфер не заполнится или вы не очистите его вручную. Если вы выполните приведенный выше код и не вызовете `Flush()`, данные могут не быть записаны даже после завершения программы, что может привести к их потере.

Поэтому всегда вызывайте `Flush()` после записи, чтобы гарантировать, что все буферизованные данные будут записаны.

```
func main() {
    ...
```

```
        if err = w.Flush(); err != nil {
            panic(err)
        }
    }
```

Вы также можете быть знакомы с удобными утилитами для форматированного вывода: `fmt.Fprintf` и `fmt.Fprintln`.

```
fmt.Fprintln(os.Stdout, "Hello VictoriaMetrics")
```

`fmt.Fprintln` записывает форматированные данные в любой `io.Writer`. В данном случае `os.Stdout` - это терминал или консоль, и это очень удобно, когда вам нужно записать форматированные строки непосредственно в файл или любой другой устройство.

Комментарии в Telegram-группе!

go

« ПРЕДЫДУЩАЯ
Graceful Shutdown в Go

СЛЕДУЮЩАЯ »
fmt.Sprintf vs String Concat