Базовый уровень:

Необходимо реализовать структуру MultiReader, которая объединяет несколько объектов, реализующих интерфейс SizedReadSeekCloser. Структура MultiReader должна сама реализовывать интерфейс SizedReadSeekCloser.

Интерфейс SizedReadSeekCloser определен следующим образом:

**type SizedReadSeekCloser interface {**

**io.ReadSeekCloser**

**Size() int64**

**}**

Функция NewMultiReader должна создавать экземпляр MultiReader:

**type MultiReader struct {**

**// put your code here...**

**}**

**func NewMultiReader(rs ...SizedReadSeekCloser) \*MultiReader {**

**// put your code here...**

**return nil**

**}**

Операции, которые должен поддерживать MultiReader:

Read: должен читать данные последовательно из всех переданных ридеров в том же порядке, что и в NewMultiReader.

Seek: должен позволять перемещать курсор на заданную позицию в объединенной последовательности ридеров.

Close: должен закрыть все ридеры.

Size: должен возвращать суммарный размер данных всех ридеров.

Шаблон для интервьюшницы

**/\*\***

**\* Необходимо реализовать структуру MultiReader, которая объединяет**

**\* несколько объектов, реализующих интерфейс SizedReadSeekCloser.**

**\* Структура MultiReader должна сама реализовывать интерфейс SizedReadSeekCloser.**

**\***

**\* Операции, которые должен поддерживать MultiReader:**

**\***

**\* \* Read: должен читать данные последовательно из всех переданных ридеров в том**

**\* же порядке, в котором они переданы в NewMultiReader**

**\* \* Seek: должен позволять перемещать курсор на заданную позицию в объединенной**

**\* последовательности ридеров.**

**\* \* Close: должен закрыть все ридеры.**

**\* \* Size: должен возвращать суммарный размер данных всех ридеров.**

**\*/**

**/\***

**type io.ReadSeekCloser interface {**

**Read(p []byte) (n int, err error)**

**Seek(offset int64, whence int) (int64, error)**

**Close() error**

**}**

**\*/**

**type SizedReadSeekCloser interface {**

**io.ReadSeekCloser**

**Size() int64**

**}**

**type MultiReader struct {**

**// put your code here...**

**}**

**func NewMultiReader(rs ...SizedReadSeekCloser) \*MultiReader {**

**// put your code here...**

**return nil**

**}**

Продвинутый уровень:

Необходимо в MultiReader добавить функционал для асинхронного префетча данных в буфер ради ускорения последующих операций чтения.

Изменения будут следующими:

**const bufferSize = 1024 \* 1024**

**func NewMultiReader(rs ...SizedReadSeekCloser, buffersNum int) \*MultiReader {**

**// put your code here...**

**return nil**

**}**

Read: должен возвращать данные всегда из своего внутреннего буфера. Буфер, в свою очередь, должен в последовательном порядке асинхронно пополняться данными из ридеров.

Seek: должен в первую очередь перемещать курсор на позицию внутри буфера. В случае, если позиция находится за пределами буфера префетча, тогда переходить на необходимую позицию в объединенной последовательности ридеров.

Close: должен обеспечивать корректное закрытие и освобождение всех ресурсов.

Что хотим услышать

Базовый уровень:

От кандидата не ожидается, что он будет помнить интерфейс io.ReadSeekCloser, поэтому можно напомнить ему в развернутом виде:

**type ReadSeekCloser interface {**

**Read(p []byte) (n int, err error)**

**Seek(offset int64, whence int) (int64, error)**

**Close() error**

**}**

Идея решения:

Сохраняем пришедшие rs к себе в структуру и дальше работаем с ними:

**type MultiReader struct {**

**rs []SizedReadSeekCloser**

**totalSize int64**

**curFileIdx int**

**curOffset int64**

**//...**

**}**

—-—-—-—-—-—-—-—-—-—-—-—-

Size() трививален — пройдемся по всем ридерам rs, посчитаем сумму от всех Size().

На что обратить внимание:

Подсчет суммы должен производится только один раз в методе NewMultiReader, а Size() просто возвращать totalSize. Плохой кандидат пересчитывает сумму каждый раз при вызове Size().

—-—-—-—-—-—-—-—-—-—-—-—-

Дальше можно начать с написания Close. Здесь все просто — пройдемся по всем rs и вызовем для каждого Close().

Close implementation

**func (m \*MultiReader) Close() error {**

**var res error**

**for \_, r := range m.rs {**

**if err := r.Close(); err != nil {**

**res = errors.Join(err, res)**

**}**

**}**

**if res != nil {**

**return fmt.Errorf("error when closing: %w", res)**

**}**

**return nil**

**}**

На что обратить внимание:

Ожидается, что кандидат предложит сджойнить все ошибки. Возвращать первую встречную ошибку — плохо.

—-—-—-—-—-—-—-—-—-—-—-—-

Следующим делом напишем Seek. Именно его написание представляет наибольший интерес в этой задаче.

В первую очередь стоит спросить кандидата, помнит ли он, что означает whence в сигнатуре функции. Будет плохо, если кандидат никогда не слышал про метод Seek и что он делает. Если кандидат не помнит whence, то стоит ему подсказать, что это просто выбор стратегии смещения, всего их существует три:

**// in package "io":**

**const (**

**SeekStart = 0 // смещение относительно начала файла**

**SeekCurrent = 1 // смещение относительно текущей позиции в файле**

**SeekEnd = 2 // смещение относительно конца файла**

**)**

Основная хитрость в написании Seek заключается в том, что после первого Read, каретки в файлах будут опущены, поэтому Seek должен не только выбрать, с какого файла начинать чтение и с какого оффсета, но и поднять каретки во всех файлах, находящиеся после текущего.

Наивное решение: давайте в самом методе Seek после подсчета curFileIdx и curOffset, пробежимся по всем ридерам rs начиная с curFileIdx и поднимем каретку в каждом файле до его начала.

Улучшенное решение: Кандидата должно смутить, что при вызове Seek в конце мы проходимся по файлам и поднимаем для них каретки. Не всегда это будет хорошей стратегией, поэтому можно это делать лениво в самом методе Read.

**func (m \*MultiReader) Read(p []byte) (n int, err error) {**

**for {**

**// ...**

**if m.curFileIdx < len(m.rs) {**

**m.rs[m.curFileIdx].Seek(0, io.SeekStart)**

**}**

**}**

**}**

На что обратить внимание:

Не должно быть никаких багов в вычислениях. Присутствует обработка различных ошибок: неподходящий whence, offset < 0, и так далее.

——-——-——-——-——-——-——-

В самом конце осталось написать Read. Здесь необходимо аккуратно обработать случай, когда дошли до самого конца, и вернуть io.EOF.

Вот и все. Будет плюсом для кандидата, если он напишет в коде проверку на то, что var \_ SizedReadSeekCloser = (\*MultiReader)(nil).

——-——-——-——-——-——-——-

**Продвинутый уровень**

Необходимо обсудить с кандидатом, а где лучше всего запускать горутину для асинхронного префетча данных: при первом вызове Read, в NewMultiReader или вообще создать для этого отдельный метод RunPrefetch. Однако все же стоит остановиться на втором варианте.

Read должен проверить, достаточно ли данных в буфере для удовлетворения запроса. Если данных в буфере недостаточно, он должен ожидать пополнения буфера до требуемого уровня или до тех пор, пока не достигнет конца файла. В случае любой ошибки при чтении (включая EOF) метод должен возвращать соответствующую ошибку после передачи всех доступных на текущий момент данных.

Seek должен сначала просчитать, можно ли переместить курсор на позицию внутри буфера. Если же требуемая позиция выходит за границы буфера, метод должен переводить курсор к соответствующей позиции в общей последовательности ридеров. Это действие приведет к повторному заполнению буфера новыми данными из ридеров, начиная с заданной позиции.

Close должен прерывать любую идущую операцию префетча данных, эффективно останавливая фоновую горутину. А также осуществлять закрытие всех ридеров rs для освобождения всех связанных с ними ресурсов (это могли быть, например, файловые дескрипторы). После вызова Close, попытки чтения через метод Read должны возвращать ошибку, указывающую на то, что поток чтения был закрыт.

На что обратить внимание:

Повторные вызовы Close после первого успешного закрытия не должны приводить к ошибкам.

**Идея решения**

**// Source: https://github.com/rclone/rclone/blob/master/fs/asyncreader/asyncreader.go**

**// Package asyncreader provides an asynchronous reader which reads**

**// independently of write**

**package asyncreader**

**import (**

**"context"**

**"errors"**

**"io"**

**"sync"**

**"time"**

**"github.com/rclone/rclone/fs"**

**"github.com/rclone/rclone/lib/pool"**

**"github.com/rclone/rclone/lib/readers"**

**)**

**const (**

**// BufferSize is the default size of the async buffer**

**BufferSize = 1024 \* 1024**

**softStartInitial = 4 \* 1024**

**bufferCacheSize = 64 // max number of buffers to keep in cache**

**bufferCacheFlushTime = 5 \* time.Second // flush the cached buffers after this long**

**)**

**// ErrorStreamAbandoned is returned when the input is closed before the end of the stream**

**var ErrorStreamAbandoned = errors.New("stream abandoned")**

**// AsyncReader will do async read-ahead from the input reader**

**// and make the data available as an io.Reader.**

**// This should be fully transparent, except that once an error**

**// has been returned from the Reader, it will not recover.**

**type AsyncReader struct {**

**in io.ReadCloser // Input reader**

**ready chan \*buffer // Buffers ready to be handed to the reader**

**token chan struct{} // Tokens which allow a buffer to be taken**

**exit chan struct{} // Closes when finished**

**buffers int // Number of buffers**

**err error // If an error has occurred it is here**

**cur \*buffer // Current buffer being served**

**exited chan struct{} // Channel is closed been the async reader shuts down**

**size int // size of buffer to use**

**closed bool // whether we have closed the underlying stream**

**mu sync.Mutex // lock for Read/WriteTo/Abandon/Close**

**ci \*fs.ConfigInfo // for reading config**

**}**

**// New returns a reader that will asynchronously read from**

**// the supplied Reader into a number of buffers each of size BufferSize**

**// It will start reading from the input at once, maybe even before this**

**// function has returned.**

**// The input can be read from the returned reader.**

**// When done use Close to release the buffers and close the supplied input.**

**func New(ctx context.Context, rd io.ReadCloser, buffers int) (\*AsyncReader, error) {**

**if buffers <= 0 {**

**return nil, errors.New("number of buffers too small")**

**}**

**if rd == nil {**

**return nil, errors.New("nil reader supplied")**

**}**

**a := &AsyncReader{**

**ci: fs.GetConfig(ctx),**

**}**

**a.init(rd, buffers)**

**return a, nil**

**}**

**func (a \*AsyncReader) init(rd io.ReadCloser, buffers int) {**

**a.in = rd**

**a.ready = make(chan \*buffer, buffers)**

**a.token = make(chan struct{}, buffers)**

**a.exit = make(chan struct{})**

**a.exited = make(chan struct{})**

**a.buffers = buffers**

**a.cur = nil**

**a.size = softStartInitial**

**// Create tokens**

**for i := 0; i < buffers; i++ {**

**a.token <- struct{}{}**

**}**

**// Start async reader**

**go func() {**

**// Ensure that when we exit this is signalled.**

**defer close(a.exited)**

**defer close(a.ready)**

**for {**

**select {**

**case <-a.token:**

**b := a.getBuffer()**

**if a.size < BufferSize {**

**b.buf = b.buf[:a.size]**

**a.size <<= 1**

**}**

**err := b.read(a.in)**

**a.ready <- b**

**if err != nil {**

**return**

**}**

**case <-a.exit:**

**return**

**}**

**}**

**}()**

**}**

**// bufferPool is a global pool of buffers**

**var bufferPool \*pool.Pool**

**var bufferPoolOnce sync.Once**

**// return the buffer to the pool (clearing it)**

**func (a \*AsyncReader) putBuffer(b \*buffer) {**

**bufferPool.Put(b.buf)**

**b.buf = nil**

**}**

**// get a buffer from the pool**

**func (a \*AsyncReader) getBuffer() \*buffer {**

**bufferPoolOnce.Do(func() {**

**// Initialise the buffer pool when used**

**bufferPool = pool.New(bufferCacheFlushTime, BufferSize, bufferCacheSize, a.ci.UseMmap)**

**})**

**return &buffer{**

**buf: bufferPool.Get(),**

**}**

**}**

**// Read will return the next available data.**

**func (a \*AsyncReader) fill() (err error) {**

**if a.cur.isEmpty() {**

**if a.cur != nil {**

**a.putBuffer(a.cur)**

**a.token <- struct{}{}**

**a.cur = nil**

**}**

**b, ok := <-a.ready**

**if !ok {**

**// Return an error to show fill failed**

**if a.err == nil {**

**return ErrorStreamAbandoned**

**}**

**return a.err**

**}**

**a.cur = b**

**}**

**return nil**

**}**

**// Read will return the next available data.**

**func (a \*AsyncReader) Read(p []byte) (n int, err error) {**

**a.mu.Lock()**

**defer a.mu.Unlock()**

**// Swap buffer and maybe return error**

**err = a.fill()**

**if err != nil {**

**return 0, err**

**}**

**// Copy what we can**

**n = copy(p, a.cur.buffer())**

**a.cur.increment(n)**

**// If at end of buffer, return any error, if present**

**if a.cur.isEmpty() {**

**a.err = a.cur.err**

**return n, a.err**

**}**

**return n, nil**

**}**

**// WriteTo writes data to w until there's no more data to write or when an error occurs.**

**// The return value n is the number of bytes written.**

**// Any error encountered during the write is also returned.**

**func (a \*AsyncReader) WriteTo(w io.Writer) (n int64, err error) {**

**a.mu.Lock()**

**defer a.mu.Unlock()**

**n = 0**

**for {**

**err = a.fill()**

**if err == io.EOF {**

**return n, nil**

**}**

**if err != nil {**

**return n, err**

**}**

**n2, err := w.Write(a.cur.buffer())**

**a.cur.increment(n2)**

**n += int64(n2)**

**if err != nil {**

**return n, err**

**}**

**if a.cur.err == io.EOF {**

**a.err = a.cur.err**

**return n, err**

**}**

**if a.cur.err != nil {**

**a.err = a.cur.err**

**return n, a.cur.err**

**}**

**}**

**}**

**// SkipBytes will try to seek 'skip' bytes relative to the current position.**

**// On success it returns true. If 'skip' is outside the current buffer data or**

**// an error occurs, Abandon is called and false is returned.**

**func (a \*AsyncReader) SkipBytes(skip int) (ok bool) {**

**a.mu.Lock()**

**defer func() {**

**a.mu.Unlock()**

**if !ok {**

**a.Abandon()**

**}**

**}()**

**if a.err != nil {**

**return false**

**}**

**if skip < 0 {**

**// seek backwards if skip is inside current buffer**

**if a.cur != nil && a.cur.offset+skip >= 0 {**

**a.cur.offset += skip**

**return true**

**}**

**return false**

**}**

**// early return if skip is past the maximum buffer capacity**

**if skip >= (len(a.ready)+1)\*BufferSize {**

**return false**

**}**

**refillTokens := 0**

**for {**

**if a.cur.isEmpty() {**

**if a.cur != nil {**

**a.putBuffer(a.cur)**

**refillTokens++**

**a.cur = nil**

**}**

**select {**

**case b, ok := <-a.ready:**

**if !ok {**

**return false**

**}**

**a.cur = b**

**default:**

**return false**

**}**

**}**

**n := len(a.cur.buffer())**

**if n > skip {**

**n = skip**

**}**

**a.cur.increment(n)**

**skip -= n**

**if skip == 0 {**

**for ; refillTokens > 0; refillTokens-- {**

**a.token <- struct{}{}**

**}**

**// If at end of buffer, store any error, if present**

**if a.cur.isEmpty() && a.cur.err != nil {**

**a.err = a.cur.err**

**}**

**return true**

**}**

**if a.cur.err != nil {**

**a.err = a.cur.err**

**return false**

**}**

**}**

**}**

**// StopBuffering will ensure that the underlying async reader is shut**

**// down so no more is read from the input.**

**//**

**// This does not free the memory so Abandon() or Close() need to be**

**// called on the input.**

**//**

**// This does not wait for Read/WriteTo to complete so can be called**

**// concurrently to those.**

**func (a \*AsyncReader) StopBuffering() {**

**select {**

**case <-a.exit:**

**// Do nothing if reader routine already exited**

**return**

**default:**

**}**

**// Close and wait for go routine**

**close(a.exit)**

**<-a.exited**

**}**

**// Abandon will ensure that the underlying async reader is shut down**

**// and memory is returned. It does everything but close the input.**

**//**

**// It will NOT close the input supplied on New.**

**func (a \*AsyncReader) Abandon() {**

**a.StopBuffering()**

**// take the lock to wait for Read/WriteTo to complete**

**a.mu.Lock()**

**defer a.mu.Unlock()**

**// Return any outstanding buffers to the Pool**

**if a.cur != nil {**

**a.putBuffer(a.cur)**

**a.cur = nil**

**}**

**for b := range a.ready {**

**a.putBuffer(b)**

**}**

**}**

**// Close will ensure that the underlying async reader is shut down.**

**// It will also close the input supplied on New.**

**func (a \*AsyncReader) Close() (err error) {**

**a.Abandon()**

**if a.closed {**

**return nil**

**}**

**a.closed = true**

**return a.in.Close()**

**}**

**// Internal buffer**

**// If an error is present, it must be returned**

**// once all buffer content has been served.**

**type buffer struct {**

**buf []byte**

**err error**

**offset int**

**}**

**// isEmpty returns true is offset is at end of**

**// buffer, or**

**func (b \*buffer) isEmpty() bool {**

**if b == nil {**

**return true**

**}**

**if len(b.buf)-b.offset <= 0 {**

**return true**

**}**

**return false**

**}**

**// read into start of the buffer from the supplied reader,**

**// resets the offset and updates the size of the buffer.**

**// Any error encountered during the read is returned.**

**func (b \*buffer) read(rd io.Reader) error {**

**var n int**

**n, b.err = readers.ReadFill(rd, b.buf)**

**b.buf = b.buf[0:n]**

**b.offset = 0**

**return b.err**

**}**

**// Return the buffer at current offset**

**func (b \*buffer) buffer() []byte {**

**return b.buf[b.offset:]**

**}**

**// increment the offset**

**func (b \*buffer) increment(n int) {**

**b.offset += n**

**}**