Talk about the confusion about Golang IO reading and writing

2020-11-03 22:29:35 OfStack

preface

There are many ways to read and write IO of Golang. At present, I know io library, os library, ioutil library, bufio library, bytes/strings library and so on.

While having more libraries is a good thing, meaning more choice, one thing that confuses me is: Which library to use in what scenarios? Why is that?

Before drawing a conclusion, I would like to give the project structure of the Golang built-in IO library, mainly for the convenience of understanding and reference:

```
#  Only the core directories and files are listed
src:
 - bufio
  - bufio.go
 - bytes
  - buffer.go
  - reader.go
 - io
  - ioutil
   - ioutil.go
  - io.go
 - os
  - file.go
 - strings
  - reader.go
```

1.io library belongs to the bottom interface definition library. Its function is to define 1 some basic interfaces and 1 some basic constants, and to explain the function of these interfaces. This library is generally used only to call its 1 constants, such as io.EOF.

2. The ioutil library is included in the io directory. It is mainly used as a toolkit with a number of useful functions, such as ReadAll(read data from a source), ReadFile (read file content), WriteFile (write data to a file), ReadDir (get directory).

3.os library mainly deals with operating system, so the file operation is basically linked to os library, such as creating a file, opening a file, etc.. This library is often used in conjunction with ioutil, bufio, etc

4. The bufio library can be understood as encapsulating another layer on top of the io library with caching capabilities. It may be confused with the ioutil library and ES45en.Buffer.
4.1 bufio VS ioutil library: Both bufio VS and ioutil libraries provide the ability to read and write files. The only difference between them is that bufio has an extra layer of caching. This advantage mainly reflects when reading large files (ES52en.ReadFile is to load the contents into memory once, if the contents are too large, the memory will burst easily).

4.2 bufio VS ES58en. Buffer: Both provide a layer of caching. The main difference is that bufio is for file to memory caching, while ES61en. Buffer is for memory to memory caching (personally it feels a bit like channel, but you can also see that ES64en. Buffer does not provide an interface to write data to files).

5.bytes and strings libraries: These two libraries are a bit of a puzzle because they both implement the Reader interface, so they differ mainly in what they target, bytes for bytes and strings for strings (they implement their methods in a similar way). Another difference is that bytes also has the functionality of Buffer, but strings does not.

Note: About Reader and Writer interface, it can be simply understood as read source and write source. That is, as long as Read method in Reader is implemented, this thing can be used as a read source, which can contain data and be read by us. So is Writer.

The above is some personal conclusions, the following will be based on the above conclusions to do 1 step, if there is a mistake, please leave a message to correct, than heart ❤ The & # 65039; !

Spy io library

There are three commonly used interfaces in the io library, namely, Reader, Writer and Close.

```
1
2  // Read The method will receive 1 Byte array p , and stores the read data
3  into the array, and finally returns the number of bytes read n .
4  //  Pay attention to n Don't 1 Must equal the length of data to be read,
5  such as a byte array p Is too small, n It's going to be equal to the length
6  of the array
7  type Reader interface {
8    Read(p []byte) (n int, err error)
9  }
10
11 // Write  Method also receives 1 Byte array p , and save the received data
12 to a file or standard output, etc., and return n Represents the length of
13 the data to be written.
14 //  when n Is not equal to len(p) When to return to 1 A mistake.
15 type Writer interface {
16   Write(p []byte) (n int, err error)
17 }

     //  closing
     type Closer interface {
       Close() error
     }
```

The concrete implementation of the Read method can be seen in the strings library:

```
1
2  //  define 1 a Reader The interface body
3  type Reader struct {
4    s    string
5    i    int64 // current reading index
6    prevRune int  // index of previous rune; or < 0
7  }
8
9  //  through NewReader method  reader  Object, a key point here is that the
10 incoming string is assigned to  s  variable
11 func NewReader(s string) *Reader {
12  return &Reader{s, 0, -1}
13 }
14
15 // Read Methods:   The core is  copy  Methods, parameters b It's a slice,
16 but copy Method affects its underlying array
17 func (r *Reader) Read(b []byte) (n int, err error) {
18   if r.i >= int64(len(r.s)) {
19     return 0, io.EOF
20   }
21  r.prevRune = -1
22  //  Core method
23   n = copy(b, r.s[r.i:])
24   r.i += int64(n)
25   return
     }
```

Spy ioutil library

As mentioned above, the ioutil library is a toolkit, which is mainly composed of more practical functions, such as ReadFile, WriteFile, etc. The only thing to note is that they are all one-time reads and one-time writes, so please pay attention to the file size when reading.

Reading data from a file:

```go

func readByFile() {
  data, err := ioutil.ReadFile( "./lab8_io/file/test.txt")
  if err != nil {
    log.Fatal("err:", err)
    return
  }
  fmt.Println("data", string(data)) // hello world !
}

```

To write data to a file:

```go

func writeFile() {
  err := ioutil.WriteFile("./lab8_io/file/write_test.txt", []byte("hello
world!"), 0644)
  if err != nil {
    panic(err)
    return
  }
}

```

Traversal directory: One thing to note about traversing a directory is that it is not sorted in a natural way.

Spy bufio library

The bufio library is also mentioned above. It adds a layer of caching to the io library. Here is an example of bufio reading large files:

```go

func readBigFile(filePath string) error {
  f, err := os.Open(filePath)
  defer f.Close()

  if err != nil {
    log.Fatal(err)
    return err
  }

  buf := bufio.NewReader(f)
  count := 0
  for {
    count += 1
    line, err := buf.ReadString('\n')
    line = strings.TrimSpace(line)
    if err != nil {
      return err
    }
  fmt.Println("line", line)
  //  Here is to avoid printing all
    if count > 100 {
      break
    }
  }
  return nil
}

```

Note:

1. bufio ReadLine/ReadBytes/ReadString/ReadSlice: ReadString and ReadBytes equivalent ReadBytes and ReadLine call ReadSlice

Spy bytes/strings library

As mentioned above, the implementation of Reader interface alone, bytes and strings underlying functions are similar, you can check the source code to prove:

```
1
2  // bytes/reader.go
3  // Read implements the io.Reader interface.
4  func (r *Reader) Read(b []byte) (n int, err error) {
5    if r.i >= int64(len(r.s)) {
6      return 0, io.EOF
7    }
8    r.prevRune = -1
9    n = copy(b, r.s[r.i:])
10   r.i += int64(n)
11   return
12 }
13
14 // strings/reader.go
15 func (r *Reader) Read(b []byte) (n int, err error) {
16   if r.i >= int64(len(r.s)) {
17     return 0, io.EOF
18   }
19   r.prevRune = -1
20   n = copy(b, r.s[r.i:])
21   r.i += int64(n)
22   return
23 }
```

Reference/Recommendation

Explain the implementation principle of bufio package in detail
Golang large file read two schemes
https://gist.github.com/suntong/032173e96247c0411140

Related articles:

- The basic method of reading and writing files in Go language programming
- A Brief discussion on the comparison of SEVERAL ways of GoLang reading documents
- Summary of several methods for reading command parameters in Go language
- go implementation file creation deletion and reading sample code
- golang flag