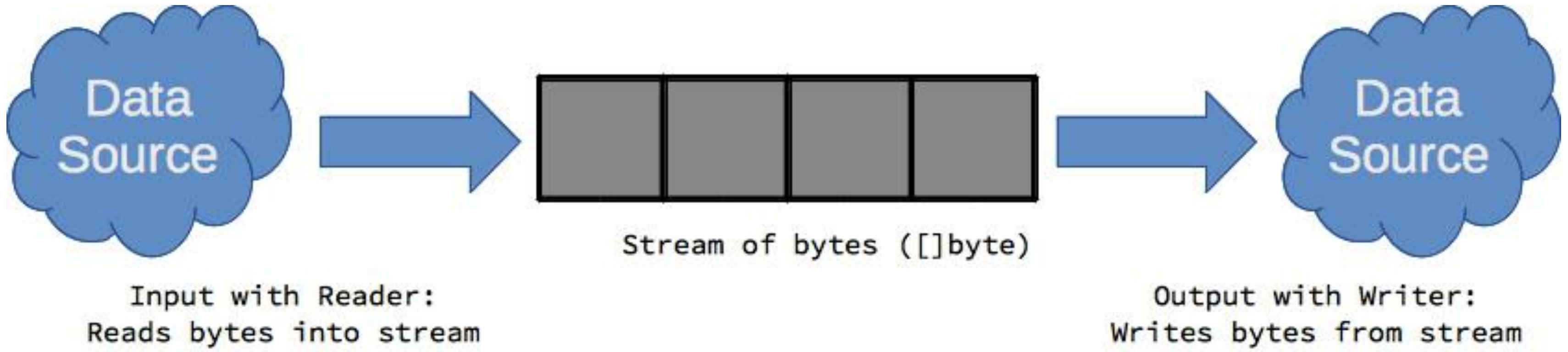
File IO in Go

Miti Bhat

Intro

- Similar to other languages, such as Java, Go models data input and output as a stream that flows from sources to targets. Data resources, such as files, networked connections, or even some in-memory objects, can be modeled as streams of bytes from which data can be *read* or *written* to, as illustrated in the following figure:



Streams

The stream of data is represented as a slice of bytes ([]byte) that can be accessed for reading or writing. As we will explore in this chapter, the io package makes available the io.Reader interface to implement code that reads and transfers data from a source into a stream of bytes. Conversely, the io.Writer interface lets implementers create code that reads data from a provided stream of bytes and writes it as output to a target resource. Both interfaces are used extensively in Go as a standard idiom to express IO operations. This makes it possible to interchange readers and writers of different implementations and contexts with predictable results.

Working with files

The os package (h t t p s : / / g o l a n g . o r g / p k g / o s /) exposes the os.File type which represents a file handle on the system. The os.File type implementsseveral IO primitives, including the io.Reader and io.Writer interfaces, which allows file content to be processed using the standard streaming IO API.

Creating and opening files

The os.Create function creates a new file with the specified path. If the file already exists, os.Create will overwrite it. The os.Open function, on the other hand, opens an existing file for reading.

The following source snippet opens an existing file and creates a copy of its content using the io.Copy function. One common, and recommended practice to notice is the deferred call to the method Close on the file. This ensures a graceful release of OS resources when the function exits:

Demo: file0.go

The os.OpenFile function provides generic low-level functionalities to create a new file or open an existing file with fine-grained control over the file's behavior and its permission. Nevertheless, the os.Open and os.Create functions are usually used instead as they provide a simpler abstraction then the os.OpenFile function.

Creating and opening files

The os.OpenFile function take three parameters. The first one is the path of the file, the second parameter is a masked bit-field value to indicate the behavior of the operation (for example, read-only, read-write, truncate, and so on) and the last parameter is a posixcompliant permission value for the file.

The following abbreviated source snippet re-implements the file copy code, from earlier.

This time, however, it uses the os.FileOpen function to demonstrate how it works:

Demo: file1.go

Files writing and reading

We have already seen how to use the os.Copy function to move data into or out of a file.

Sometimes, however, it will be necessary to have complete control over the logic that writes or reads file data. The following code snippet, for instance, uses the WriteString method from the os.File variable, fout, to create a text file:

Demo: filewrite0.go

- If, however, the source of your data is not text, you can write raw bytes directly to the file as shown in the following source snippet

Files writing and reading

- As an io.Reader, reading from of the io.File type directly can be done using the *Read* method. This gives access to the content of the file as a raw stream of byte slices. The following code snippet reads the content of file ../ch0r/dict.txt as raw bytes assigned to slice p up to 1024-byte chunks at a time:
- Demo: fileread.go

Standard input, output, and error

- The os package includes three pre-declared variables, os.Stdin, os.Stdout, and os.Stderr, that represent file handles for standard input, output, and error of the OS respectively. The following snippet reads the file f1 and writes its content to io.Stdout, standard output, using the os.Copy function (standard input is covered later):
- Demo: osstd.go

Reading from standard input

- Instead of reading from an arbitrary io.Reader, the fmt.Scan, fmt.Scanf, and fmt.Scanln are used to read data from standard input file handle, os.Stdin. The following code snippet shows a simple program that reads text input from the console:
- Demo: fmtscan1.go
- In the previous program, the fmt.Scanf function parses the input using the format specifier "%d" to read an integer value from the standard input. The function will throw an error if the value read does not match exactly the specified format. For instance, the following shows what happens when character D is read instead of an integer:

Encoding and decoding data

- Another common aspect of IO in Go is the encoding of data, from one representation to another, as it is being streamed. The encoders and decoders of the standard library, found in the *encoding* package (h t t p s : / / g o l a n g . o r g / p k g / e n c o d i n g /), use the io.Reader and io.Writer interfaces to leverage IO primitives as a way of streaming data during encoding and decoding.
- Go supports several encoding formats for a variety of purposes including data conversion, data compaction, and data encryption. This chapter will focus on encoding and decoding data using the *Gob* and *JSON* format for data conversion.

Binary encoding with gob

- The gob package (*https://golang.org/pkg/encoding/gob*) provides an encoding format that can be used to convert complex Go data types into binary. Gob is self-describing, meaning each encoded data item is accompanied by a type description. The encoding process involves streaming the gob-encoded data to an io.Writer so it can be written to a resource for future consumption.
- The following snippet shows an example code that encodes variable books, a slice of the Book type with nested values, into the gob format. The encoder writes its generated binary data to an os.Writer instance, in this case the file variable of the *os.File type:
- Demo: gob0.go

Binary encoding with gob

Although the previous example is lengthy, it is mostly made of the definition of the nested data structure assigned to variable books. The last half-dozen or more lines are where the encoding takes place. The gob encoder is created with

enc := gob.NewEncoder(file).

Encoding the data is done by simply calling enc.Encode(books) which streams the encoded data to the provide file.

The decoding process does the reverse by streaming the gob-encoded binary data using an io.Reader and automatically reconstructing it as a strongly-typed Go value. The following code snippet decodes the gob data that was encoded and stored in the books.data file in the previous example. The decoder reads the data from an io.Reader, in this instance the variable file of the *os.File type:

Demo: gob1.go

Binary encoding with gob

- Decoding a previously encoded gob data is done by creating a decoder using dec := gob.NewDecoder(file).
- The next step is to declare the variable that will store the decoded data. In our example, the books variable, of the []Book type, is declared as the destination of the decoded data. The actual decoding is done by invoking dec.Decode(&books). Notice the Decode() method takes the address of its target variable as an argument. Once decoded, the books variable will contain the reconstituteddata structure streamed from the file.

Encoding data as JSON

- The encoding package also comes with a *json* encoder sub-package (https://golang.org/ pkg/encoding/json/) to support JSON-formatted data. This greatly broadens the number of languages with which Go programs can exchange complex data structures. JSON encoding works similarly as the encoder and decoder from the gob package.
- The difference is that the generated data takes the form of a clear text JSON-encoded format instead of a binary. The following code updates the previous example to encode the data as JSON:
- Demo: json0.go

Encoding data as JSON

- The code is exactly the same as before. It uses the same slice of nested structs assigned to the books variable. The only difference is the encoder is created with enc := json.NewEncoder(file) which creates a JSON encoder that will use the file variable as its io.Writer destination. When enc.Encode(books) is executed, the content of the variable books is serialized as JSON to the local file books.dat, shown in the following code (formatted for readability):
- Demo; books.dat

Encoding data as JSON

- The generated JSON-encoded content uses the name of the struct fields as the name for the JSON object keys by default. This behavior can be controlled using struct tags (as in *Controlling JSON mapping with struct tags*).
- Consuming the JSON-encoded data in Go is done using a JSON decoder that streams its source from an io.Reader. The following snippet decodes the JSON-encoded data, generated in the previous example, stored in the file book.dat. Note that the data structure (not shown in the following code) is the same as before:
- Demo: json1.go

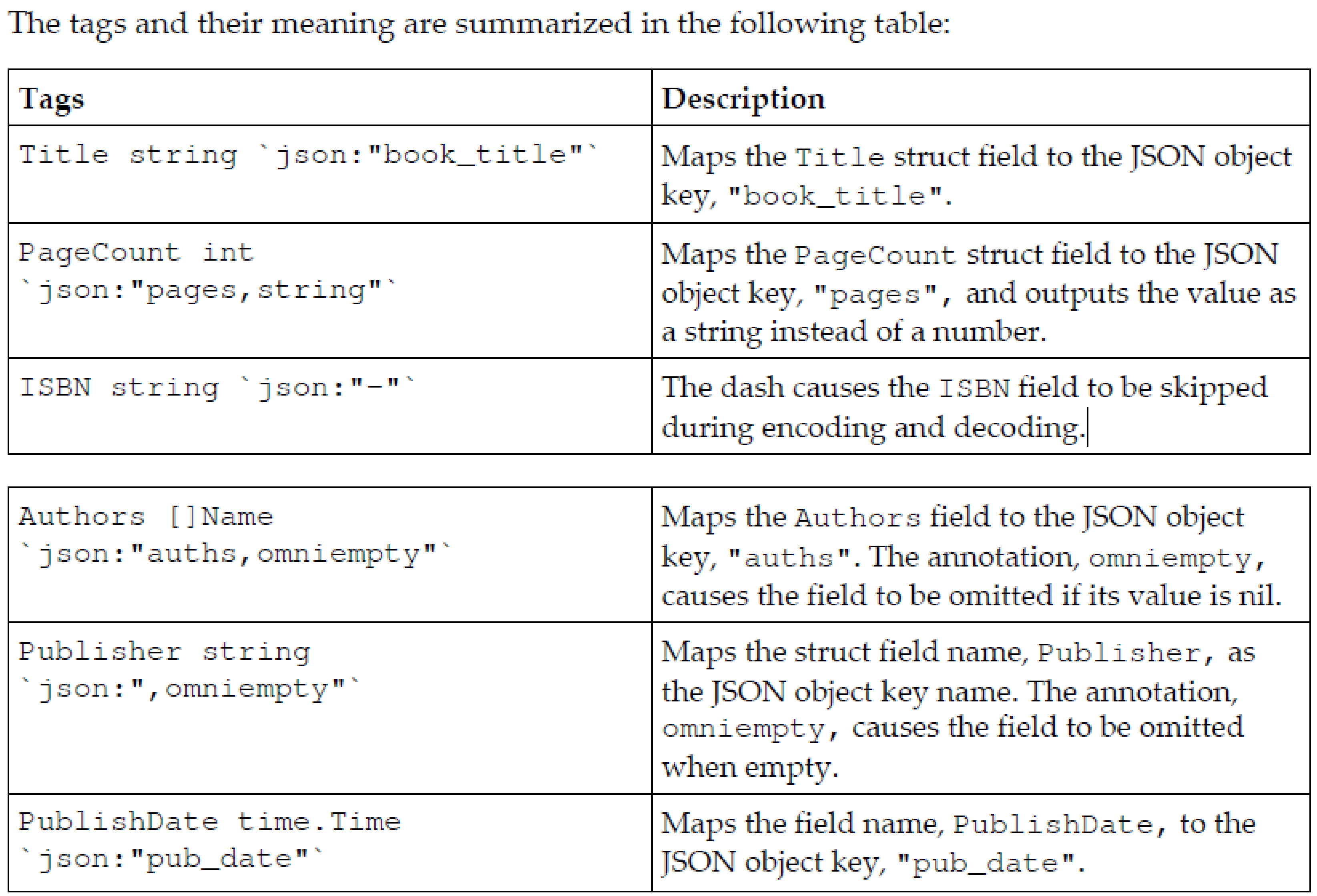
Encoding data as JSON

- The data in the books.dat file is stored as an array of JSON objects. Therefore, the code must declare a variable capable of storing an indexed collection of nested struct values. In the previous example, the books variable, of the type []Book is declared as the destination of the decoded data. The actual decoding is done by invoking dec.Decode(&books). Notice the Decode() method takes the address of its target variable as an argument. Once decoded, the books variable will contain the reconstituted data structure streamed from the file.

Controlling JSON mapping with struct tags

- By default, the name of a struct field is used as the key for the generated JSON object. This can be controlled using struct type tags to specify how JSON object key names are mapped during encoding and decoding of the data. For instance, the following code snippet declares struct fields with the json: tag prefix to specify how object keys are to be encoded and decoded:
- Demo: json2.go

Tags



The tags and their meaning are summarized in the following table:

Tags	Description
`` Title string `json:"book_title"` ``	Maps the `Title` struct field to the JSON object key, `"book_title"`.
`` PageCount int `json:"pages,string"` ``	Maps the `PageCount` struct field to the JSON object key, `"pages"`, and outputs the value as a string instead of a number.
`` ISBN string `json:"-"` ``	The dash causes the `ISBN` field to be skipped during encoding and decoding.
`` Authors []Name `json:"auths,omniempty"` ``	Maps the `Authors` field to the JSON object key, `"auths"`. The annotation, `omniempty`, causes the field to be omitted if its value is nil.
`` Publisher string `json:",omniempty"` ``	Maps the struct field name, `Publisher`, as the JSON object key name. The annotation, `omniempty`, causes the field to be omitted when empty.
`` PublishDate time.Time `json:"pub_date"` ``	Maps the field name, `PublishDate`, to the JSON object key, `"pub_date"`.

Controlling JSON mapping with struct tags

- Demo: books.data
- Notice the JSON object keys are titled as specified in the struct tags. The object key "pages" (mapped to the struct field, PageCount) is encoded as a string. Finally, the struct field, ISBN, is omitted, as annotated in the struct tag.

Custom encoding and decoding

- The JSON package uses two interfaces, *Marshaler* and *Unmarshaler*, to hook into encoding and decoding events respectively. When the encoder encounters a value whose type implements json.Marshaler, it delegates serialization of the value to the method MarshalJSON defined in the Marshaller interface. This is exemplified in the following abbreviated code snippet where the type Name is updated to implement json.Marshaller as shown:
- Demo: json3.go
- In the previous example, values of the Name type are serialized as a JSON string (instead of an object as earlier). The serialization is handled by the method Name.MarshallJSON which returns an array of bytes that contains the last and first name separated by a comma.
- The preceding code generates the following JSON output:
- Demo:
- The Name type is an implementation of json.Unmarshaler. When the decoder encounters a JSON object with the key "Authors", it uses the method Name.Unmarshaler to reconstitute the Go struct Name type from the JSON string.

Thank You