12.29.2022 Golang Lesson instruction

**1. Error handling**.

Errors indicate any abnormal condition occurring in the program. Let's say we are trying to open a file and the file does not exist in the file system. This is an abnormal condition and it's represented as an error.

Errors in Go are plain old values. Just like any other built-in type such as int, float64, ... error values can be stored in variables, passed as parameters to functions, returned from functions, and so on.

Errors are represented using the built-in error type. We will learn more about the error type later in this tutorial.

**2. The error type**

The error type is an interface type. An error variable represents any value that can describe itself as a string. Here is the interface’s declaration:

type error interface {

Error() string

}

Note that **fmt.Errorf** will prove extremely useful when used to wrap another error with the **%w** format verb - but I’ll get into more detail on that further down in the article.

There are a few other important things to note in the example above.

* Errors can be returned as **nil**, and in fact, it’s the default, or “zero”, value of on error in Go. This is important since checking **if err != nil** is the idiomatic way to determine if an error was encountered (replacing the **try**/**catch** statements you may be familiar with in other programming languages).
* Errors are typically returned as the last argument in a function. Hence in our example above, we return an **int** and an **error**, in that order.
* When we do return an error, the other arguments returned by the function are typically returned as their default “zero” value. A user of a function may expect that if a non-nil error is returned, then the other arguments returned are not relevant.
* Lastly, error messages are usually written in lower-case and don’t end in punctuation. Exceptions can be made though, for example when including a proper noun, a function name that begins with a capital letter, etc.

**3.Defining Expected Errors**

Another important technique in Go is defining expected Errors so they can be checked for explicitly in other parts of the code. This becomes useful when you need to execute a different branch of code if a certain kind of error is encountered.

**Defining Sentinel Errors**

Building on the **Divide** function from earlier, we can improve the error signaling by pre-defining a “Sentinel” error. Calling functions can explicitly check for this error using **errors.Is**:

package main

import (

    "errors"

    "fmt"

)

var ErrDivideByZero = errors.New("divide by zero")

func Divide(a, b int) (int, error) {

    if b == 0 {

        return 0, ErrDivideByZero

    }

    return a / b, nil

}

func main() {

    a, b := 10, 0

    result, err := Divide(a, b)

    if err != nil {

        switch {

        case errors.Is(err, ErrDivideByZero):

            fmt.Println("divide by zero error")

        default:

            fmt.Printf("unexpected division error: %s\n", err)

        }

        return

    }

    fmt.Printf("%d / %d = %d\n", a, b, result)

}

#### 4. Defining Custom Error Types

Many error-handling use cases can be covered using the strategy above, however, there can be times when you might want a little more functionality. Perhaps you want an error to carry additional data fields, or maybe the error’s message should populate itself with dynamic values when it’s printed.

You can do that in Go by implementing custom errors type.

Below is a slight rework of the previous example. Notice the new type **DivisionError**, which implements the **Error** **interface**. We can make use of **errors.As** to check and convert from a standard error to our more specific **DivisionError**.

package main

import (

    "errors"

    "fmt"

)

type DivisionError struct {

    IntA int

    IntB int

    Msg  string

}

func (e \*DivisionError) Error() string {

    return e.Msg

}

func Divide(a, b int) (int, error) {

    if b == 0 {

        return 0, &DivisionError{

            Msg: fmt.Sprintf("cannot divide '%d' by zero", a),

            IntA: a, IntB: b,

        }

    }

    return a / b, nil

}

func main() {

    a, b := 10, 0

    result, err := Divide(a, b)

    if err != nil {

        var divErr \*DivisionError

        switch {

        case errors.As(err, &divErr):

            fmt.Printf("%d / %d is not mathematically valid: %s\n",

              divErr.IntA, divErr.IntB, divErr.Error())

        default:

            fmt.Printf("unexpected division error: %s\n", err)

        }

        return

    }

    fmt.Printf("%d / %d = %d\n", a, b, result)

}

**5. What is fan-in and fan-out**

In our original problem, we want to calculate one set of metrics for each month, a total of 12 months. An easy way to solve this is to assign one worker and have it work out all 12 metrics. If each month takes about one min to calculate, it will take 12 mins.

We can agree this is way too slow. In Go, we can use concurrent programming to dedicate 12 workers (fan-out) and each worker can calculate one of the 12 month’s metrics.

Once all 12 workers finish their calculation, we could regroup their data (fan-in). Before implementing this pattern, we need to understand channels since all communications between Goroutines is done through Channels.

package main

import (

"fmt"

"time"

)

func generate(data string) <- chan string{

    channel := make(chan string)

    go func(){

        for{

            channel <- data

            time.Sleep(1000)

        }

    }()

    return channel

}

type Processor  struct{

    jobChannel chan string

    done chan \*Worker

    workers []\*Worker

}

type Worker struct{

    name string

}

func (w \* Worker) processJob(data string, done chan \*Worker){

    // Use the data and process the job

    go func(){

        fmt.Println("Working on data ", data, w.name)

        time.Sleep(3000)

        done <- w

    }()

}

func GetProcessor() \* Processor{

    p := &Processor{

        jobChannel: make(chan string),

        workers: make([]\*Worker,5),

        done: make( chan \*Worker),

    }

    for i := 0; i < 5; i++ {

        w := &Worker{name : fmt.Sprintf("<Worker - %d>", i)}

        p.workers[i] = w

    }

    p.startProcess()

    return p

}

func (p \*Processor) startProcess(){

    go func(){

        for{

            select {

            default :

                if len(p.workers) > 0{

                    w := p.workers[0]

                    p.workers = p.workers[1:]

                    w.processJob( <- p.jobChannel,p.done)

                }

            case w := <- p.done:

                p.workers = append(p.workers, w)

            }

        }

    }()

}

func (p \*Processor) postJob(jobs <-chan string){

    p.jobChannel <- <-jobs

}

func main() {

    source := generate("data string")

    process := GetProcessor()

    for i := 0; i < 12; i++ {

        process.postJob(source)

    }

}