CR1

// Sample program to show how the default error type is implemented.

package main

import "fmt"

// http://golang.org/pkg/builtin/#error

type error interface {

Error() string

}

// http://golang.org/src/pkg/errors/errors.go

type errorString struct {

s string

}

// http://golang.org/src/pkg/errors/errors.go

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

return e.s

}

// http://golang.org/src/pkg/errors/errors.go

// New returns an error that formats as the given text.

func New(text string) error {

return &errorString{text}

}

func main() {

if err := webCall(); err != nil {

fmt.Println(err)

return

}

fmt.Println("Life is good")

}

// webCall performs a web operation.

func webCall() error {

return New("Bad Request")

}

CR2

// Sample program to show how to use error variables to help the

// caller determine the exact error being returned.

package main

import (

"errors"

"fmt"

)

var (

// ErrBadRequest is returned when there are problems with the request.

ErrBadRequest = errors.New("Bad Request")

// ErrPageMoved is returned when a 301/302 is returned.

ErrPageMoved = errors.New("Page Moved")

)

func main() {

if err := webCall(true); err != nil {

switch err {

case ErrBadRequest:

fmt.Println("Bad Request Occurred")

return

case ErrPageMoved:

fmt.Println("The Page moved")

return

default:

fmt.Println(err)

return

}

}

fmt.Println("Life is good")

}

// webCall performs a web operation.

func webCall(b bool) error {

if b {

return ErrBadRequest

}

return ErrPageMoved

}

CR3

// http://golang.org/src/pkg/encoding/json/decode.go

// Sample program to show how to implement a custom error type

// based on the json package in the standard library.

package main

import (

"fmt"

"reflect"

)

// An UnmarshalTypeError describes a JSON value that was

// not appropriate for a value of a specific Go type.

type UnmarshalTypeError struct {

Value string // description of JSON value

Type reflect.Type // type of Go value it could not be assigned to

}

// Error implements the error interface.

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

return "json: cannot unmarshal " + e.Value + " into Go value of type " + e.Type.String()

}

// An InvalidUnmarshalError describes an invalid argument passed to Unmarshal.

// (The argument to Unmarshal must be a non-nil pointer.)

type InvalidUnmarshalError struct {

Type reflect.Type

}

// Error implements the error interface.

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

if e.Type == nil {

return "json: Unmarshal(nil)"

}

if e.Type.Kind() != reflect.Ptr {

return "json: Unmarshal(non-pointer " + e.Type.String() + ")"

}

return "json: Unmarshal(nil " + e.Type.String() + ")"

}

// user is a type for use in the Unmarshal call.

type user struct {

Name int

}

func main() {

var u user

err := Unmarshal([]byte(`{"name":"bill"}`), u) // Run with a value and pointer.

if err != nil {

switch e := err.(type) {

case \*UnmarshalTypeError:

fmt.Printf("UnmarshalTypeError: Value[%s] Type[%v]\n", e.Value, e.Type)

case \*InvalidUnmarshalError:

fmt.Printf("InvalidUnmarshalError: Type[%v]\n", e.Type)

default:

fmt.Println(err)

}

return

}

fmt.Println("Name:", u.Name)

}

// Unmarshal simulates an unmarshal call that always fails.

func Unmarshal(data []byte, v interface{}) error {

rv := reflect.ValueOf(v)

if rv.Kind() != reflect.Ptr || rv.IsNil() {

return &InvalidUnmarshalError{reflect.TypeOf(v)}

}

return &UnmarshalTypeError{"string", reflect.TypeOf(v)}

}

CR4

// Package example4 provides code to show how to implement behavior as context.

package example4

import (

"bufio"

"fmt"

"io"

"log"

"net"

)

// client represents a single connection in the room.

type client struct {

name string

reader \*bufio.Reader

}

// TypeAsContext shows how to check multiple types of possible custom error

// types that can be returned from the net package.

func (c \*client) TypeAsContext() {

for {

line, err := c.reader.ReadString('\n')

if err != nil {

switch e := err.(type) {

case \*net.OpError:

if !e.Temporary() {

log.Println("Temporary: Client leaving chat")

return

}

case \*net.AddrError:

if !e.Temporary() {

log.Println("Temporary: Client leaving chat")

return

}

case \*net.DNSConfigError:

if !e.Temporary() {

log.Println("Temporary: Client leaving chat")

return

}

default:

if err == io.EOF {

log.Println("EOF: Client leaving chat")

return

}

log.Println("read-routine", err)

}

}

fmt.Println(line)

}

}

// temporary is declared to test for the existence of the method coming

// from the net package.

type temporary interface {

Temporary() bool

}

// BehaviorAsContext shows how to check for the behavior of an interface

// that can be returned from the net package.

func (c \*client) BehaviorAsContext() {

for {

line, err := c.reader.ReadString('\n')

if err != nil {

switch e := err.(type) {

case temporary:

if !e.Temporary() {

log.Println("Temporary: Client leaving chat")

return

}

default:

if err == io.EOF {

log.Println("EOF: Client leaving chat")

return

}

log.Println("read-routine", err)

}

}

fmt.Println(line)

}

}

CR5

// Sample program to show see if the class can find the bug.

package main

import (

"fmt"

"log"

)

// customError is just an empty struct.

type customError struct{}

// Error implements the error interface.

func (c \*customError) Error() string {

return "Find the bug."

}

// fail returns nil values for both return types.

func fail() ([]byte, \*customError) {

return nil, nil

}

func main() {

var err error

fmt.Printf("Type of value stored inside the interface: %T\n", err)

if \_, err = fail(); err != nil {

fmt.Printf("Type of value stored inside the interface: %T\n", err)

}

log.Println("No Error")

}

CR6

// Sample program to show how wrapping errors work.

package main

import (

"fmt"

"github.com/pkg/errors"

)

// AppError represents a custom error type.

type AppError struct {

State int

}

// Error implements the error interface.

func (c \*AppError) Error() string {

return fmt.Sprintf("App Error, State: %d", c.State)

}

func main() {

// Make the function call and validate the error.

if err := firstCall(10); err != nil {

// Use type as context to determine cause.

switch v := errors.Cause(err).(type) {

case \*AppError:

// We got our custom error type.

fmt.Println("Custom App Error:", v.State)

default:

// We did not get any specific error type.

fmt.Println("Default Error")

}

// Display the stack trace for the error.

fmt.Println("\nStack Trace\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

fmt.Printf("%+v\n", err)

fmt.Println("\nNo Trace\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

fmt.Printf("%v\n", err)

}

}

// firstCall makes a call to a second function and wraps any error.

func firstCall(i int) error {

if err := secondCall(i); err != nil {

return errors.Wrapf(err, "firstCall->secondCall(%d)", i)

}

return nil

}

// secondCall makes a call to a third function and wraps any error.

func secondCall(i int) error {

if err := thirdCall(); err != nil {

return errors.Wrap(err, "secondCall->thirdCall()")

}

return nil

}

// thirdCall create an error value we will validate.

func thirdCall() error {

return &AppError{99}

}

CR7

// Sample program to show how wrapping errors work with the stdlib.

package main

import (

"errors"

"fmt"

)

// AppError represents a custom error type.

type AppError struct {

State int

}

// Error implements the error interface.

func (c \*AppError) Error() string {

return fmt.Sprintf("App Error, State: %d", c.State)

}

func main() {

// Make the function call and validate the error.

if err := firstCall(10); err != nil {

// Use type as context to determine cause.

switch v := errors.Unwrap(err).(type) {

case \*AppError:

// We got our custom error type.

fmt.Println("Custom App Error:", v.State)

default:

// We did not get any specific error type.

fmt.Println("Default Error")

}

// Display the error.

fmt.Println("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

fmt.Printf("%v\n", err)

}

}

// firstCall makes a call to a second function and wraps any error.

func firstCall(i int) error {

if err := secondCall(i); err != nil {

return fmt.Errorf("firstCall->secondCall(%d) : %w", i, err)

}

return nil

}

// secondCall makes a call to a third function and wraps any error.

func secondCall(i int) error {

if err := thirdCall(); err != nil {

return fmt.Errorf("secondCall->thirdCall() : %w", err)

}

return nil

}

// thirdCall create an error value we will validate.

func thirdCall() error {

return &AppError{99}

}