Interfaces in Go.

4 December 2019

Evgeny Khabarov
Senior Golang developer, Bold Commerce

Agenda

- 54 slides
- 2 pictures
- a lot of examples

What is an interface?

Quotes

Interface_(computing) (https://en.wikipedia.org/wiki/Interface_%28computing%29)

```
"In computing, an interface is a shared *boundary* across which two or more
separate components of a computer system exchange information." (c)
```

Duck_typing (https://en.wikipedia.org/wiki/Duck_typing)

```
"If it walks like a duck and it quacks like a duck, then it must be a duck." (c)
```

Effective Go (https://golang.org/doc/effective_go.html#interfaces_and_types)

```
"Interfaces in Go provide a way to specify the *behavior* of an object: if
something can do *this*, then it can be used *here*." (c)
```

In other words...

- Interface is a fixed sets of methods.
- When I use an interface in Go I expect certain **behavior**.

Interfaces in other languages

Java

interface:

```
public interface MyInterface {
    public String hello = "Hello";
    public void sayHello();
}
```

implementation:

```
public class MyInterfaceImpl implements MyInterface {
    public void sayHello() {
        System.out.println(MyInterface.hello);
    }
}
```

- Interface can contain constants
- Interface should be implemented **explicitly**

PHP

interface:

```
interface iTemplate {
    const b = 'Interface constant';

    public function setVariable($name, $var);
    public function getHtml($template);
}
```

implementation:

```
class Template implements iTemplate  {
    public function setVariable($name, $var) { ... }

    public function getHtml($template) {
      echo iTemplate::b;
    }
}
```

- Interface can contain constants
- Interface should be implemented **explicitly**

Interfaces in Go

interface:

```
type Doer interface {
    Do() errror
}
```

implementation:

```
type TheThing struct { ... }

func (t *TheThing) Do() error { ... }
```

- Interface cannot contain constants
- Interface can be implemented **implicitly**, i.e. there is no link between the two
- Interface and its implementation could be defined in different packages

How to use interfaces?

Interface

```
type Starter interface {
    Start()
}
```

Function

```
type Starter interface {
    Start()
}

func StartSomething(s Starter) {
    s.Start()
}
```

Implementation

```
type Job struct{}

func (Job) Start() {
    fmt.Println("job started")
}

type Consumer struct{}

func (Consumer) Start() {
    fmt.Println("consumer started")
}
```

All together

```
type Starter interface {
    Start()
}

func StartSomething(s Starter) {
    s.Start()
}

type Job struct{}

func (Job) Start() {
    fmt.Println("job started")
}

type Consumer struct{}

func (Consumer) Start() {
    fmt.Println("consumer started")
}

func main() {
    StartSomething(Job{})
    StartSomething(Consumer{})
}
```

Run

Pretty simple, isn't it?

Data types in Go

Go is statically typed

Each variable has its own type

```
type MyInt int

var i int
var j MyInt
```

The variables **i** and **j** have distinct static types and, although they have the same underlying type, they cannot be assigned to one another without a conversion. (c)

```
i = j // error: cannot use j (type MyInt) as type int in assignment
i = int(j)
```

Interface type

- Interface type is a set of methods , it can store any concrete (non-interface) type, which implements interface methods.

```
type Starter interface {
    Start()
}

type Job struct{}

func (Job) Start() {}

type Consumer struct{}

var s Starter
var j Job
var c Consumer

func main() {
    j = Job{}
    s = j
    s = c // error
}
```

Run

interface{}

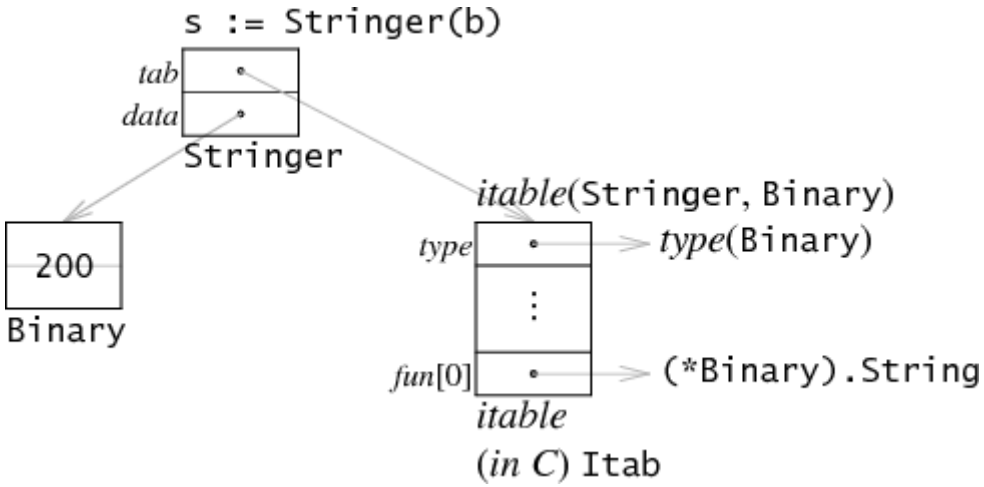
- is an empty interface with empty set of methods
- is satisfied by any value, with **zero** or more methods
- is statically typed

```
var (
    i interface{}
    a int
)

func main() {
    a = 10
    i = a

    fmt.Printf("a: %#v\n", a)
    fmt.Printf("i: %#v\n", i)
    fmt.Printf("type of a: %T\n", a)
    fmt.Printf("type of i: %T\n", i)
}
```

Run

interface{} under the covers

- is implemented by pair type **T** and value **V**
- where **T** is concrete type like *int*, *struct* etc.
- **V** is a concrete value such as an *int*, *struct* or *pointer*, never an interface itself, and has type **T**.



research.swtch.com/interfaces (https://research.swtch.com/interfaces)

How to know what's inside?

```
func main() {
    var i interface{}

    i = struct {
        Name string
    }{"Go"}

    fmt.Printf("type of i: %T\n", i)
    fmt.Printf("values with default format: %v\n", i)
    fmt.Printf("+fields: %+v\n", i)
    fmt.Printf("+types: %#v\n", i)
}
```

Run

- See fmt (https://golang.org/pkg/fmt/) package for details.

nil interface{} value

```
type MyError struct{}

func (MyError) Error() string { return "MyError" }

func Do(i int) error {
    var e *MyError = nil
    if i < 0 {
        e = &MyError{}
    }
    return e
}

func main() {
    if err := Do(1); err != nil { // Is err == nil?
        fmt.Printf("err is not nil: %#v\n", err)
        return
    }
    fmt.Println("err is nil")
}
```

Run

nil interface{} value (cont.)

- An interface value is nil only if the V and T **are both unset**
- T=nil, V is not set

One more note about interface{}

Try to avoid using it.

"interface{} says nothing" (c) Rob Pike

go-proverbs.github.io/ (https://go-proverbs.github.io/)

Runtime checking for interface implementation

Make an assertion for interface{} type

```
package main

type (
    Starter interface{ Start() }
    Stopper interface{ Stop() }
    Job     struct{}
)

func (Job) Start() { print("job started") }

func main() {
    var j interface{} = Job{}

    if s, ok := j.(Starter); ok {
        print("use Starter interface\n")
        s.Start()
    }

    if s, ok := j.(Stopper); ok {
        print("use Stopper interface\n")
        s.Stop()
    }
}
```

Run

Let's try to use wrong method

```
package main

type (
    Starter interface{ Start() }
    Stopper interface{ Stop() }
    Job     struct{}
)

func (Job) Start() { print("job started") }

func main() {
    var j interface{} = Job{}

    if s, ok := j.(Stopper); ok {
        s.Start() // no such method: build time error
    }
}
```

Run

How about assertion for non-interface type?

```
package main

type (
    Starter interface{ Start() }
    Job     struct{}
)

func (Job) Start() { print("job started") }

func main() {
    var j Job = Job{}

    if s, ok := j.(Starter); ok {
        print("use Starter interface\n")
        s.Start()
    }
}
```

Run

It works for interface types only.

```
package main

type (
    Starter interface{ Start() }
    Stopper interface{ Stop() }
    Job     struct{}
)

func (Job) Start() { print("job started") }

func main() {
    var j Starter = Job{}

    if s, ok := j.(Stopper); ok {
        print("use Stopper interface\n")
        s.Stop()
    }
    print("Stopper implementation is not found.")
}
```

Run

Type switch

```
type (
    Starter interface{ Start() }
    Job     struct{}
)

func (Job) Start() { print("job started\n") }

func Do(value interface{}) {
    switch v := value.(type) {
    case int:
        print(5+v, "\n")
    case Starter:
        v.Start()
    case string:
        print("This is a ", v)
    }
}

func main() {
    Do(1)
    Do(Job{})
    Do("string")
}
```

Run

Interface implementation

Private implementation

```
package customer

import "database/sql"

type CustomerRepo interface { // exported interface
    Get(int) *Customer
}

type repo struct { // unexported implementation
    db *sql.DB
}

func NewRepo(db *sql.DB) CustomerRepo {
    return repo{db: db}
}

func (repo) Get(id int) *Customer { return db.Select() }
```

```
package main

func main() {
    repo := customer.NewRepo(db)
    c := repo.Get(1)
}
```

Independent implementation

Job struct is defined in **job** package

```
package job

type Job struct{}

func (Job) Start() { print("job started") }
```

```
package main

// import ".../job"

func main() {
    var j interface{} = /*job.*/ Job{}

    if s, ok := j.(interface{ Start() }); ok {
        s.Start()
    }
}
```

Run

Interface extending in PHP

```
<?
interface a {
    public function foo();
}

interface b {
    public function bar();
}


class ab implements a, b
{
    public function foo() { }
    public function bar() { }
}
?>
```

Embedded interfaces

```
package main

type (
    Starter        interface{ Start() }
    Stopper        interface{ Stop() }
    StarterStopper interface {
        Starter
        Stopper
    }
    Job struct{}
)

func (Job) Start() { print("job started\n") }
func (Job) Stop()  { print("job stopped\n") }

func main() {
    var j interface{} = Job{}

    if s, ok := j.(StarterStopper); ok {
        print("use StarterStopper interface\n")
        s.Start()
        s.Stop()
    }
}
```

Run

Examples from standard library

fmt.Stringer

```
type Stringer interface {
    String() string
}
```

- Used by **Print*** functions of fmt (https://golang.org/pkg/fmt/) package, for converting objects to string.

```
package main

import "fmt"

type a struct{}
type b struct{}

func (b) String() string {
    return "String was executed"
}

func main() {
    fmt.Printf("a: %s\n", a{})
    fmt.Printf("b: %s\n", b{})
}
```

Run

error

```
type error interface {
    Error() string
}
```

- Used by fmt package.
- Used for creating custom error.

sort.Interface

```
type Interface interface {
    Len() int
    Less(i, j int) bool
    Swap(i, j int)
}
```

- A type, typically a collection, that satisfies **sort.Interface** can be sorted by the routines in sort (https://golang.org/pkg/sort/) package.

io.Reader / io.Writer

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Writer interface {
    Write(p []byte) (n int, err error)
}
```

- This two are simple interfaces which are widely used.

```
func Fprint(w io.Writer, a ...interface{}) (n int, err error) // fmt package
```

io.Writer is implemented by
- os.File
- http.ResponseWriter
- bytes.Buffer
- log.Logger

Is integer type able to serve HTTP requests?

Why not?

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

type Counter int

func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    *ctr++
    fmt.Fprintf(w, "counter = %d\n", *ctr)
}

func main() {
    var c Counter
    log.Fatal(http.ListenAndServe(":8044", &c))
}
```

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

gRPC example

Goal

- Our goal is to apply params **Limit** and **Page** come from incoming gRPC request to DB query.

```
// service.proto
service Orders {
  rpc List(Request) returns (Response);
}
message Request {
  int32 limit = 1;
  int32 page = 2;
}
```

```
// service.pb.go (autogenerated)
type Request struct {
    Limit int32
    Page  int32
}

func (m *Request) GetLimit() int32 { /*...*/ }
func (m *Request) GetPage() int32  { /*...*/ }
```

Interfaces

We define several interfaces:

```
type PageLimiter interface {
    GetLimit() int32
    GetPage() int32
}

type Applier interface {
    Apply(squirrel.SelectBuilder) squirrel.SelectBuilder
}

type PageLimiterApplier interface {
    PageLimiter
    Applier
}
```

Helper

```
type pageLimiter struct {
    limit, page int32
}

func New(limit, page int32) PageLimitApplier {
    return &pageLimiter{limit: limit, page: page}
}

func (pl *pageLimiter) GetLimit() int32 {
    return pl.limit
}

func (pl *pageLimiter) GetPage() int32 {
    return pl.page
}

func (pl *pageLimiter) Apply(q sq.SelectBuilder) sq.SelectBuilder {
    if pl.GetLimit() < 1 {
        return q
    }
    return q.
        Limit(uint64(pl.GetLimit())).
        Offset((uint64(pl.GetPage()) - 1) * uint64(pl.GetLimit()))
}
```

Helper (cont.)

```
func MustApply(a Applier, q *sq.SelectBuilder) error {
    if a == nil {
        return ErrApplierIsNil
    }
    if q == nil {
        return ErrSelectBuilderIsNil
    }

    *q = a.Apply(*q)

    return nil
}

func FromParams(pl PageLimiter) PageLimitApplier {
    page := pl.GetPage()
    if page < 1 {
        page = 1
    }
    limit := pl.GetLimit()
    if limit < 1 {
        limit = 50
    }
    return New(limit, page)
}
```

Repo, service & gRPC server

```
func (s *grpcServer) List(req *Request) (*Response, error) {
    orders, err := s.service.List(helpers.FromParams(req))
}
```

```
func (s *service) List(pla PageLimitApplier) ([]Order, error) {
    log("limit", pla.GetLimit(), "page", pla.GetPage())

    orders, err := s.repo.List(pla)
    // do something with categories
}
```

```
func (r *repo) List(a Applier) ([]Order, error) {
    q := squirrel.Select("...").From("orders").Where( /*...*/ )

    if err := pagination.MustApply(a, &q); err != nil {
        return nil, err
    }

    // do Select
}
```

Conclusion

Questions?

Links

- en.wikipedia.org/wiki/Interface_(computing) (https://en.wikipedia.org/wiki/Interface_(computing))
- en.wikipedia.org/wiki/Duck_typing (https://en.wikipedia.org/wiki/Duck_typing)
- go-proverbs.github.io/ (https://go-proverbs.github.io/)
- golang.org/doc/effective_go.html#interfaces_and_types (https://golang.org/doc/effective_go.html#interfaces_and_types)
- golang.org/doc/effective_go.html#interface_conversions (https://golang.org/doc/effective_go.html#interface_conversions)
- golang.org/doc/effective_go.html#embedding (https://golang.org/doc/effective_go.html#embedding)
- golang.org/doc/effective_go.html#interface_methods (https://golang.org/doc/effective_go.html#interface_methods)
- golang.org/doc/faq#nil_error (https://golang.org/doc/faq#nil_error)
- research.swtch.com/interfaces (https://research.swtch.com/interfaces)

Thank you

Evgeny Khabarov
Senior Golang developer, Bold Commerce
@eekhabarov (http://twitter.com/eekhabarov)