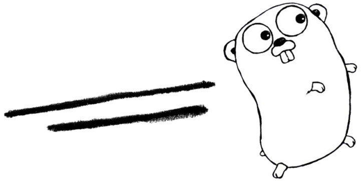
Dependency Injection in Go

@brownylin



Outline

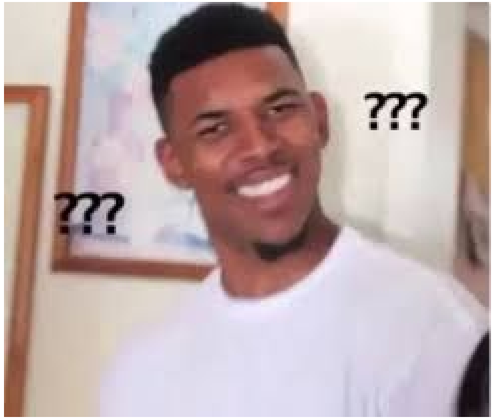
- Introduction
- DI framework
- Conclusion

Outline

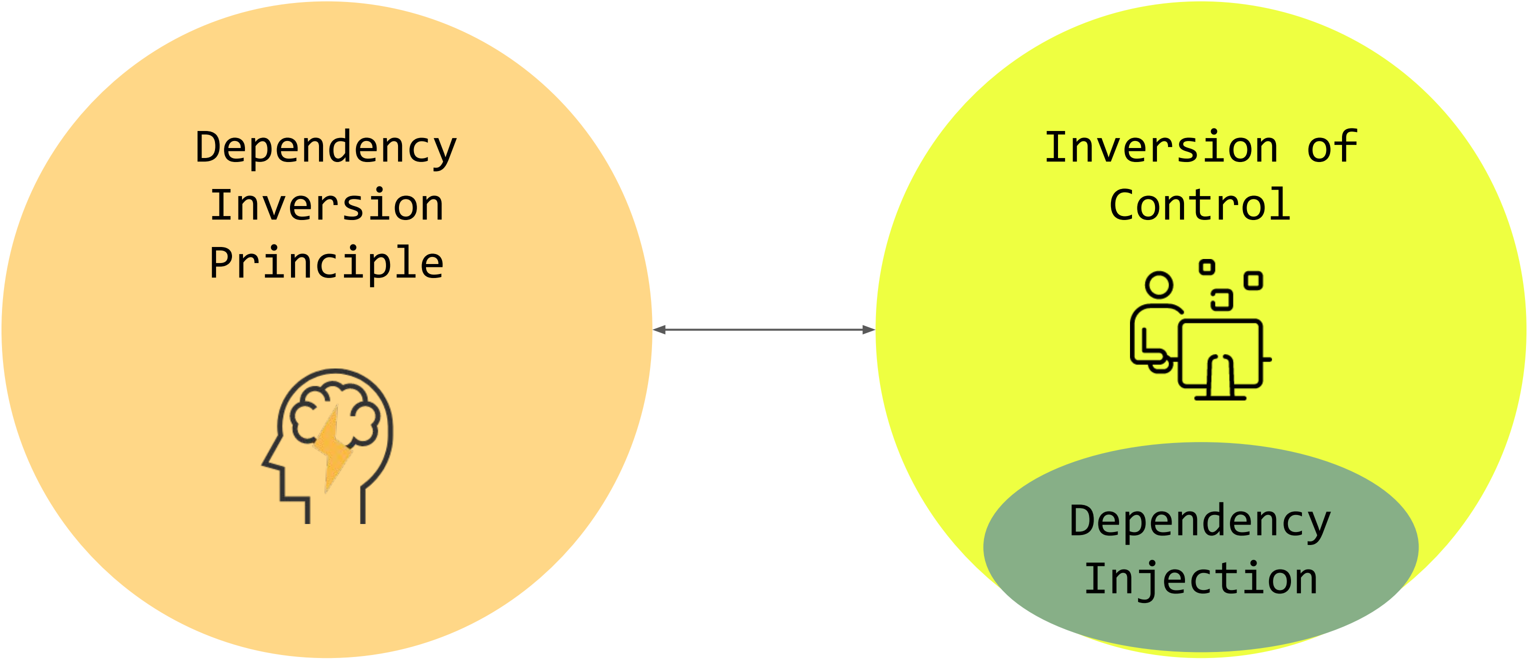
- Introduction
- DI framework
- Conclusion

Terms

- Dependency Inversion Principle (DIP)
- Inversion of Control (IoC)
- Dependency Injection (DI)



Terms



Dependency Inversion Principle

Inversion of Control

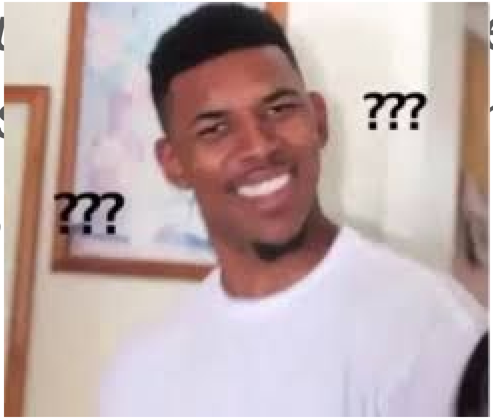
Dependency Injection

Dependency Inversion Principle (DIP)

- S.O.L.I.**D.**
- A guiding principle to loosely coupled system
    - *High-level modules should not depend on low-level modules. Both should depend on abstractions*
    - *Abstractions should not depend on details. Details should depend on abstractions*

Dependency Inversion Principle (DIP)

- S.O.L.I.**D.**
- A guiding principle to loosely coupled system
  - *High-level modu epend on low-level modules. Both s n abstractions*
  - *Abstractions sh on details. Details should depend*



Dependency Inversion Principle (DIP)

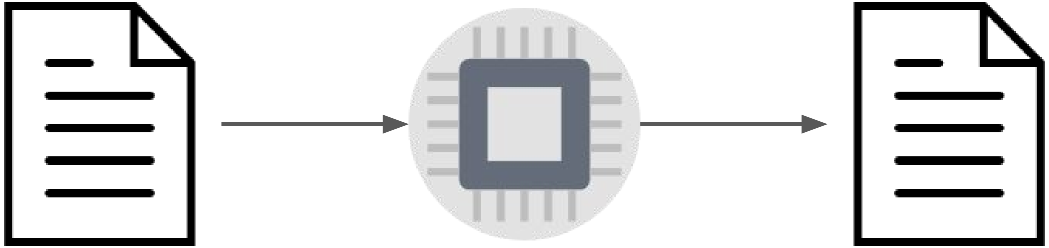
- S.O.L.I.**D.**
- A guiding principle to loosely coupled system
  - *Abstraction & Inversion*

Problems (coupled system)

- Changes are risky
- Testing is difficult
- Semantics is complex

Example

Let's say you implement a encryption algorithm



encryptor

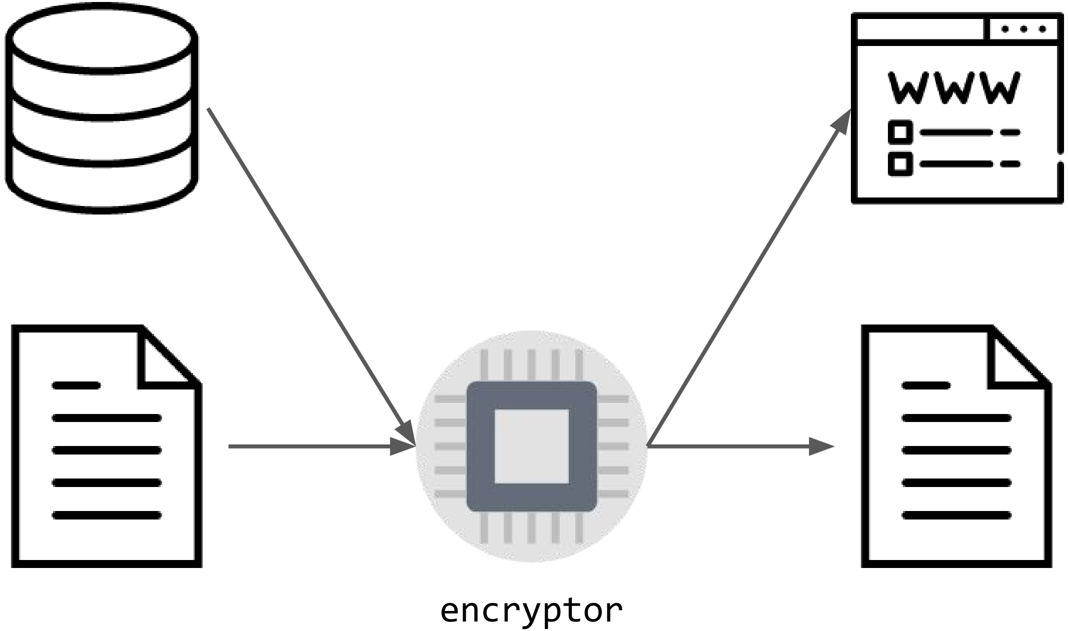
```go
func (e *Encryptor) Run(src, dst string) error {

    dat, err := ioutil.ReadFile(src)
    if err != nil {
        return nil
    }

    result := e.encrypt(dat)

    return ioutil.WriteFile(dst, result, 0644)
}

func (e *Encryptor) encrypt(dat []byte) []byte {
    return []byte("awesome encrypt")
}
```

read src from file

encrypt

write result to dst

Requirements are ~~always~~ changed



```
func (e *Encryptor) Run(srcType, dstType string) error {

    var src []byte
    switch srcType {       →  read src according to srcType
    case "file":
        src = e.readFromFile()
    case "database":
        src = e.readFromDatabase()
    }

    // encrypt
    r := e.encrypt(dat)

    switch dstType {       →  write result according to dstType
    case "file":
        src = e.writeToFile(r)
    case "webservice":
        src = e.writeToWebservice(r)
    }
}
```

```go
func (e *Encryptor) Run(srcType, dstType string) error {

    var src []byte
    switch srcType {
    case "file":
        src = e.readFromFile(x, y, z)
    case "database":
        src = e.readFromDatabase(i, j)
    }

    // encrypt
    ...
}
```

Depends on low level module interface

```go
func (e *Encryptor) Run(
    srcType, dstType, x, y, z, i, j string) error {

    var src []byte
    switch srcType {
    case "file":
        src = e.readFromFile(x, y, z)
    case "database":
        src = e.readFromDatabase(i, j)
    }

    // encrypt
    ...
}
```

1. Changes are risky
2. Testing is difficult
3. Semantics is complex

Abstraction

```go
func (e *Encryptor) Run(r IReader, w IWriter) error {
    // read file
    dat, err := r.Read()
    if err != nil {
        return nil
    }

    // encrypt
    result := e.encrypt(dat)

    // output encrypted content
    return w.Write(result)
}
```

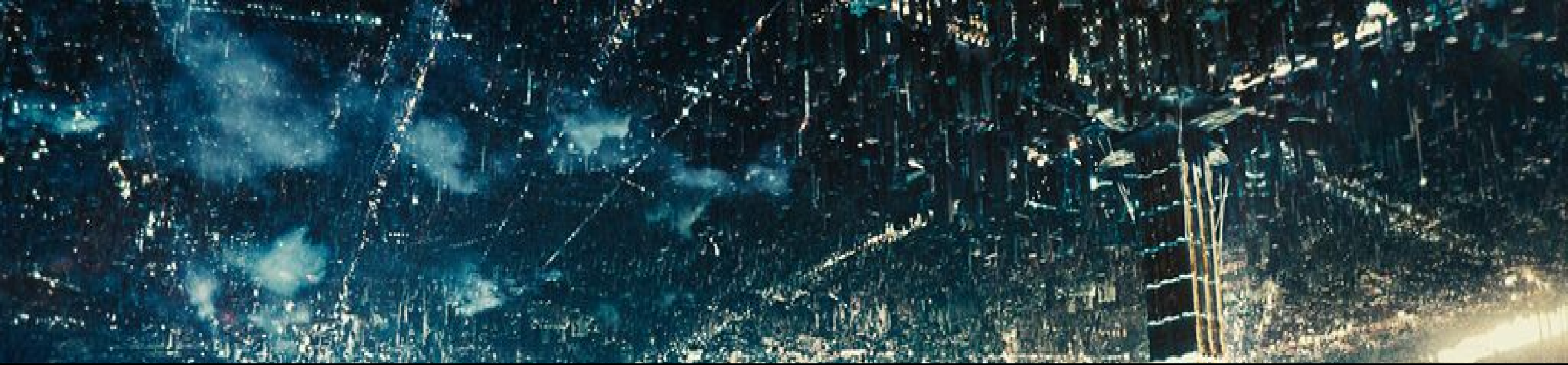
High level defines the abstraction

```go
type IReader interface {
    Read() ([]byte, error)
}

type IWriter interface {
    Write(dat []byte) error
}
```

```
type IReader interface {
    Read() ([]byte, error)
}
```

```
type fileReader struct {
    src string
}

func (f *fileReader) Read() ([]byte, error) {
    return ioutil.ReadFile(f.src)
}

type dbReader struct {
    host  string
    query string
}

func (d *dbReader) Read() ([]byte, error) {
    return []byte("query db: host[%s], query[%s]", d.host, d.query)
}
```

Low level implements the abstraction



Inversion

```
fr := &fileReader{
    src: "/a/b/c",
}

dbr := &dbReader{
    host:  "127.0.0.1",
    query: "q",
}

e.Run(fr, ...)
or
e.Run(dbr, ...)
```

1. Changes are NOT risky
2. Testing is NOT difficult
3. Semantics is NOT complex

Inversion of Control

- DIP in different scopes
  - *The control of the interface*
  - *The control of dependency creation and binding*
  - *The control of the flow (procedural to event-driven)*

Dependency Injection ↓

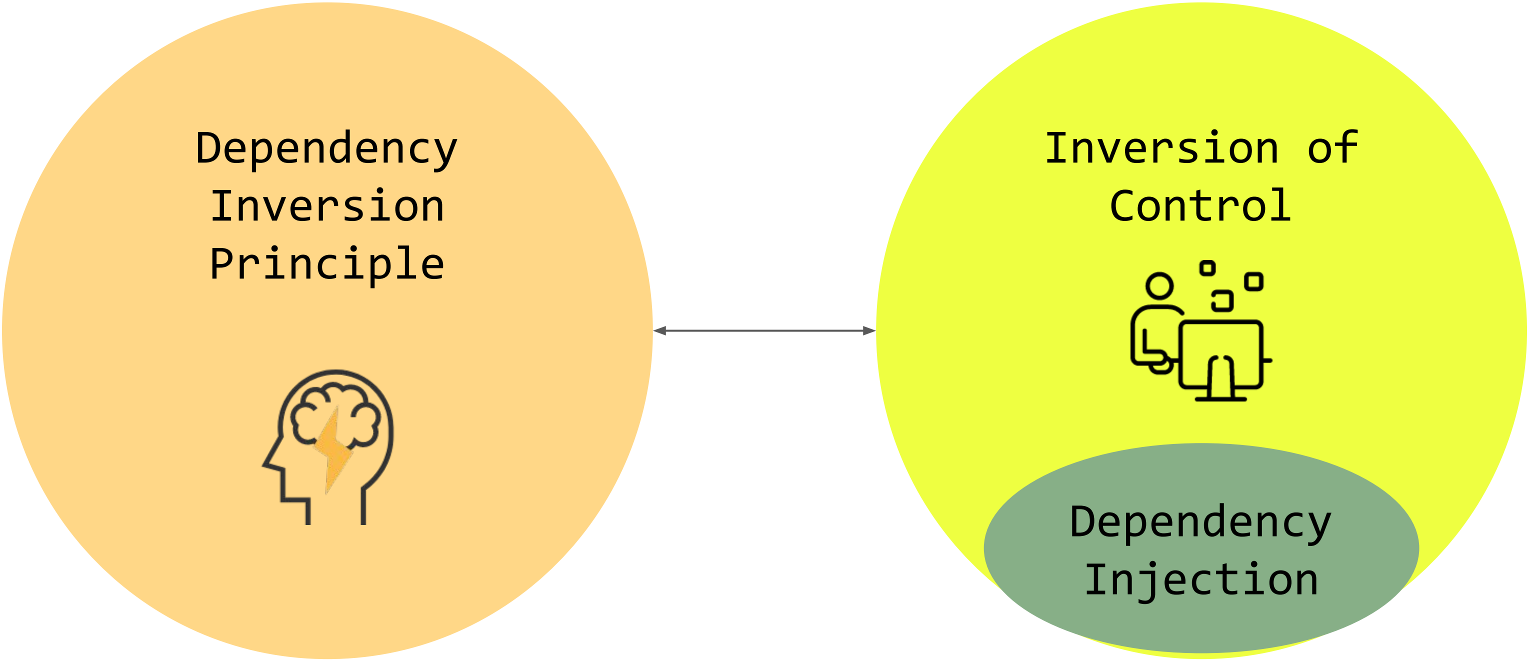
Dependency Injection

A dependency is passed to an object as an argument rather than the object creating or finding it

↑

inversion

```
fr := &fileReader{
    src: "/a/b/c",
}

dbr := &dbReader{
    host:  "127.0.0.1",
    query: "q",
}

e.Run(fr, ...) ← Some kinds of injection
```



Dependency Inversion Principle

Inversion of Control

Dependency Injection

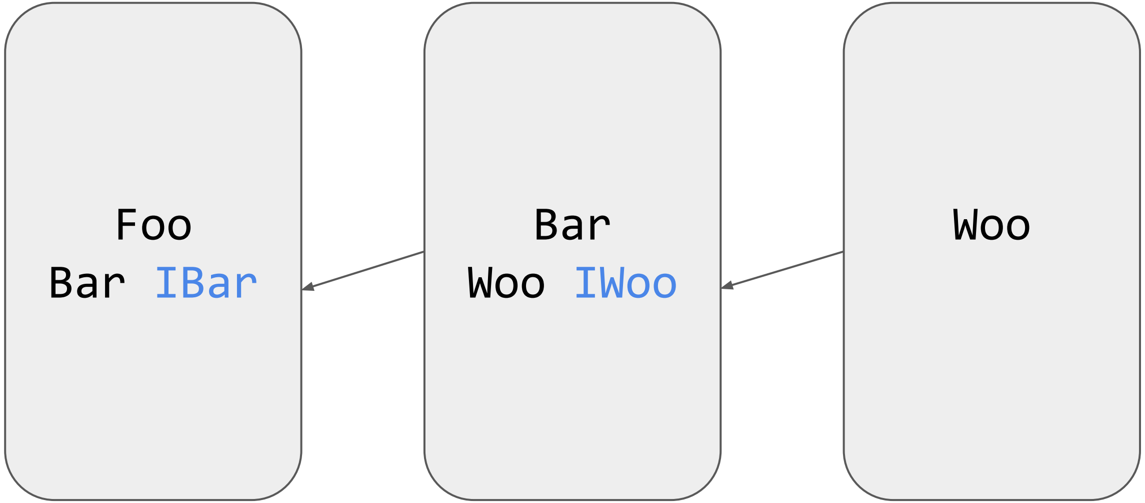
Abstraction & **Inversion** applied on **different design scopes** to address **the problems of coupled system**

Outline

- Introduction
- DI framework
- Conclusion

Problems (nested/meshed dependencies)

```
woo := &Woo{}
bar := &Bar{Woo: woo}
foo := &Foo{Bar: bar}
```

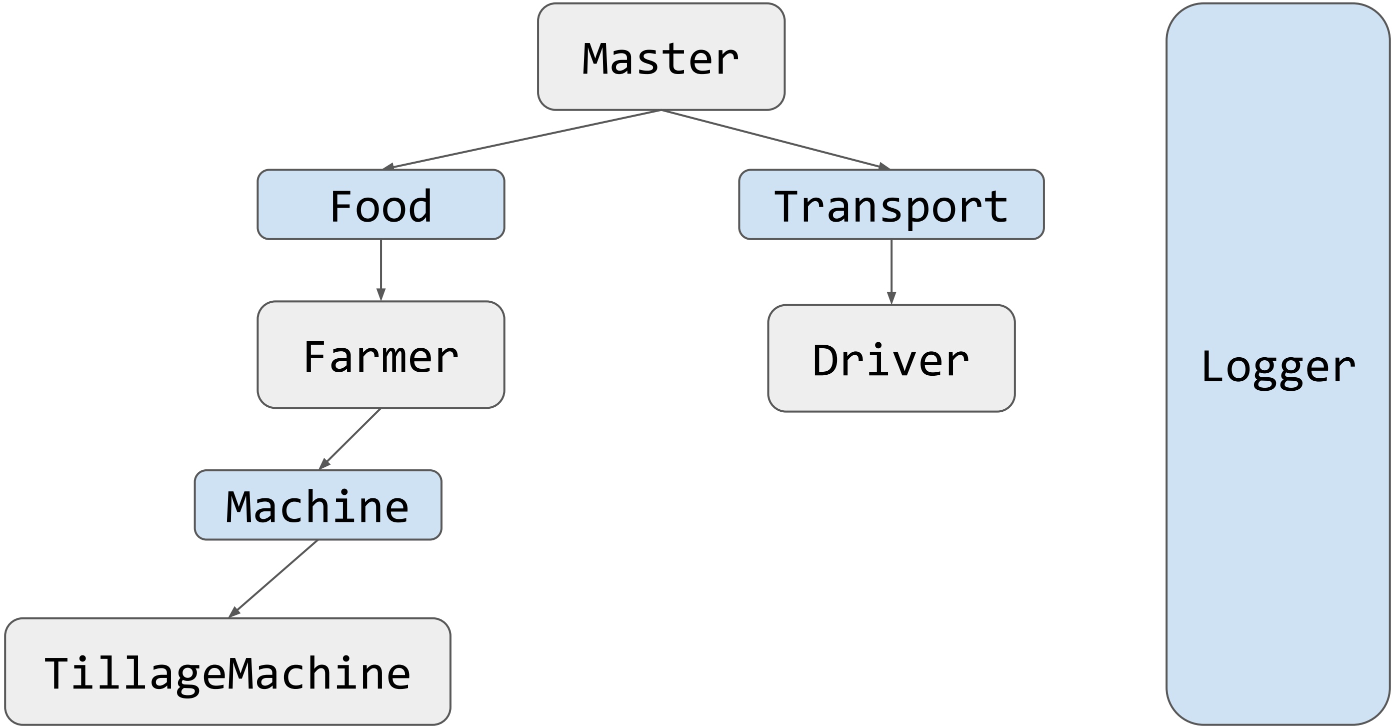


```
go get github.com/browny/inject
```

The improvements

1. More convenient config format
2. Support constructor
3. Return constructed dependency graph

Example


Master
Food
Transport
Farmer
Driver
Logger
Machine
TillageMachine

```go
package example

type Logger interface {
    Log(format string, a ...interface{})
}

type Food interface {
    GetRice()
}

type Machine interface {
    Run(n int) error
}

type Transport interface {
    Fly(src, dst string)
}
```

```go
type MyLogger struct{}

func (m *MyLogger) Log(format string, v ...interface{}) {
    log.Printf(format, v...)
}

type Master struct {
    Logger    `inject:"logger"`
    Food      `inject:"example.Master.Food"`
    Transport `inject:"example.Master.Transport"`
}
```

Mailbox address of dependencies

```go
type Farmer struct {
    Logger  `inject:"logger"`
    Machine `inject:"example.TillageMachine.Machine"`
}

func (f *Farmer) GetRice() {
    err := f.Machine.Run(3)
    if err != nil {
        f.Log("Machine breaks, no rice")
    }
    f.Log("Got rice")
}

type TillageMachine struct {
    Logger `inject:"logger"`
}

func (tm *TillageMachine) Run(n int) error {
    tm.Log("Tillage %d hours", n)
    return nil
}
```

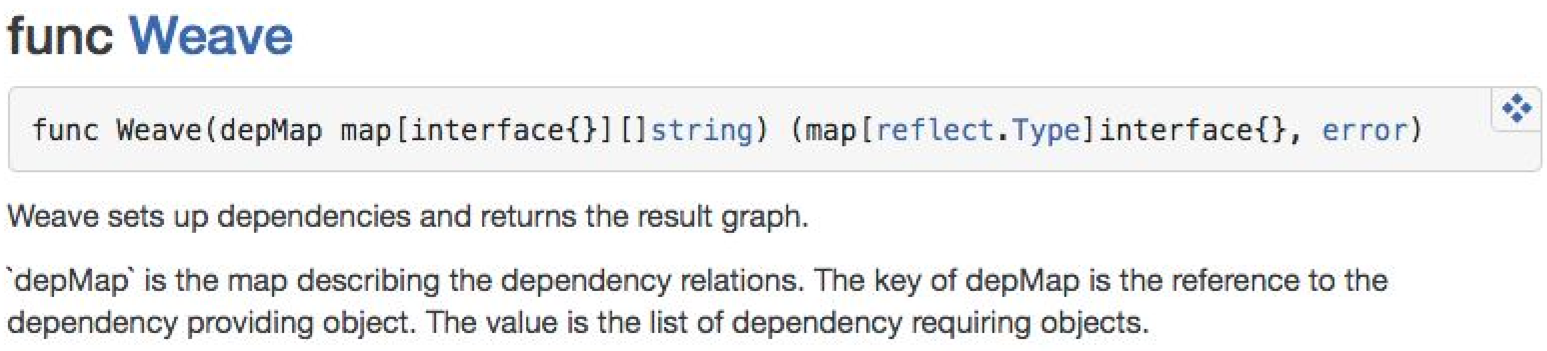
```go
type Driver struct {
    Logger `inject:"logger"`
    plane  string
}

func (d *Driver) Setup() error {
    d.plane = "Boeing787"
    return nil
}

func (d *Driver) Fly(src, dst string) {
    d.Log("%s Fly from %s to %s", d.plane, src, dst)
}
```

} → Constructor

The caller configures the dependencies

```
depMap := map[interface{}][]string{
    &myLogger: []string{
        "logger",
    },
    &driver: []string{
        "example.Master.Transport",
    },
    &farmer: []string{
        "example.Master.Food",
    },
    &tillMachine: []string{
        "example.TillageMachine.Machine",
    },
    &master: []string{},
}
```

```
driver := example.Driver{}
farmer := example.Farmer{}
master := example.Master{}
myLogger := example.MyLogger{}
tillMachine := example.TillageMachine{}
```



func Weave

```
func Weave(depMap map[interface{}][]string) (map[reflect.Type]interface{}, error)
```

Weave sets up dependencies and returns the result graph.

`depMap` is the map describing the dependency relations. The key of depMap is the reference to the dependency providing object. The value is the list of dependency requiring objects.

```
graph, err := Weave(depMap)
s.NoError(err)

master.Food.GetRice()
master.Transport.Fly("C++", "Go")

f := graph[reflect.TypeOf(&example.Farmer{})].(*example.Farmer)
f.Machine.Run(5)
```

Outline

- Introduction
- DI framework
- Conclusion

Disadvantages

- DI framework dependent
- Code is difficult to trace
- Errors are pushed to run-time (circular reference, bad binding, ...)

Caveats

- DI framework dependent -> 凡事總有代價
- Code is difficult to trace -> 好的風格
- Errors are pushed to run-time -> 想辦法測試

Interface{} everything?

Dependency Injection is EVIL

https://www.tonymarston.net/php-mysql/dependency-injection-is-evil.html

Recap

- Clarify terms (DIP, IoC, DI)
- Go through a DI framework
- Review the disadvantages