Systemdump

About me

A driver pattern in go

Aug 27, 2017

Writing modular programs

Modular programming implies decoupling abstractions from implementations. Quite often, your program is built atop a particular piece of technology and you realize that it could be easily replaced with something else, keeping all functionalities available and working. At this point you are saying to yourself: *“I just need a modular way to pick any of those implementations while writing generic code in the rest of my application”*. In other words you are looking for *drivers*.

Good old drivers

Unlike plugins that leverage a mechanism to extend the set of features offered by a program, drivers focus on offering a strict environment tied to other pieces of code by contract. A contract is the only interaction medium with a driver, putting aside implementations specificities.

The layout

Firsteval we need a driver registry that will reside in the `driver` package. We will then create a `drivers` package containing our drivers structured by group. Each group has a `register` package that eases the import process in the rest of the application and sets build constraints.

```
.
├── driver
│   └── registry.go
└── drivers
    └── group
        ├── group.go
        ├── driver1
        │   └── driver1.go
        ├── driver2
        │   └── driver2.go
        └── register
```

It’s all about contracts

Without surprise, the way to enforce a contract is with an `interface`. Let’s pretend we want to write a sample application that could leverage multiple printing backends.

```go
type Printer interface {
        Open(dest string) error
        Print([]byte) (n int, err error)
        Close() error
}
```

The driver registry

At some point we will basically need to retrieve the driver implementation from a name i.e. a string. This means that we need drivers to declare themselves to the registry under aliases.

Note: The following code is simplified (it’s likely that the reflect package will send panics directly to the app if you are messing up with types).

```go
package driver

import (
        "reflect"
        "sync"
)

var registry struct {
        contracts sync.Map
        drivers   sync.Map
}

// Declare ties a contract to a group name in the registry. It panics if the
// contract is not an Interface.
func Declare(group string, contract interface{}) {
        if reflect.TypeOf(contract).Elem().Kind() != reflect.Interface {
                panic("Contract is not an Interface for driver group " + group)
        }
        registry.contracts.Store(group, contract)
}

// Load fetches a driver.  It panics if the driver can't be retrieved.
func Load(group, name string) interface{} {
        fqn := fullQualifiedName(group, name)
        driver, ok := registry.drivers.Load(fqn)
        if !ok {
                panic("Unknown driver " + fqn)
        }
        return driver
}

// Register pushes a driver into a registry group with the given name. It
// panics whenever the  group is missing from the registry or the driver is
// not implementing the group contract.
func Register(group, name string, driver interface{}) {
        fqn := fullQualifiedName(group, name)
        contract, ok := registry.contracts.Load(group)
        if !ok {
                panic("Unknown driver group " + group)
        }
        if !reflect.TypeOf(driver).Implements(reflect.TypeOf(contract.Elem()) {
                panic("Unsatisfied contract for driver " + fqn)
        }
        registry.drivers.Store(fqn, driver)
}

func fullQualifiedName(group, name string) string {
        return group + ":" + name
}
```

Declaring a driver group

Now let’s declare our driver group refering to the `Printer` interface.

```
package printer

import "repo/user/project/driver"

func init() {
        driver.Declare("printer", (*Printer)(nil))
}

type Printer interface {
        Open(dest string) error
        Print([]byte) (n int, err error)
        Close() error
}
```

Writing drivers

We first implement a driver that writes to the console.

```
package console
```

```
import (
        "fmt"

        "repo/user/project/driver"
)

func init() {
        driver.Register("printer", "console", &Console{})
}

type Console struct{}

func (c *Console) Open(string) error {
        return nil
}

func (c *Console) Print(buf []byte) (int, error) {
        return fmt.Print(buf)
}

func (c *Console) Close() error {
        return nil
}
```

Then we implement a driver that writes to a file.

```
package file

import (
        "os"

        "repo/user/project/driver"
)

func init() {
        driver.Register("printer", "file", &File{})
}

type File struct {
        dest *os.File
}

func (f *File) Open(dest string) (err error) {
        f.dest, err = os.Create(dest)
        return
}

func (f *File) Print(buf []byte) (int, error) {
        return f.dest.Write(buf)
}
```

```go
func (f *File) Close() error {
        return f.dest.Close()
}
```

Drivers at compile time

We want to avoid a build process that includes irrelevant drivers for a target OS, doesn't allow to exclude unstable drivers for production or build minimal binaries. Fortunately, go already provides all necessary material to elaborate fine grained cross-plateform application builds thanks to build constraints.

As an example, if we declare the following in the `register` package of our driver group:

```go
// +build !exclude_driver_console

package register

import _ "repo/user/project/drivers/printer/console"
```

Then this driver will not be built if you pass `-tags=exclude_driver_console` to the build chain.

Using drivers

Using drivers is trivial, we simply import the `register` package and the driver group in the proper order.

```go
package main

import (
        "flag"

        "repo/user/project/driver"

        "repo/user/project/drivers/printer"
        _ "repo/user/project/drivers/printer/register"
)

func main() {
        driverName := flag.String("driver", "console", "Printing driver")
        flag.Parse()

        printer := driver.Load("printer", *driverName).(printer.Printer)
        printer.Open("out")
        printer.Print([]byte("Hello world!"))
        printer.Close()
}
```

Result:

```
$ go run main.go --driver=console
Hello world!

$ go run main.go --driver=file
$ cat out
Hello world!
```

Share me on → Twitter | Facebook | Google+ | Linkedin | Reddit

React to this post by writing a comment below!

RSS
xlucas
systemdump
Xavier Lucas

Technical insights, considerations and stories of the IT sphere told from the inside.