Если вы видите что-то необычное, просто сообщите мне.

Злоупотребление синтаксисом Go для создания DSL(предметно-ориентированного языка)

Go частый выбор для создания внутреностей высокопроизводительных систем, но так он так же разработан с некоторым количеством функций которые хороши для создания высокоуровневых абстракций. Вам не нужно переключаться на динамические языки такие как Ruby или Python, чтобы получить удовольствие от API или объяснительный синтакс.

Довольно часто для выражения API используется DSL(Предметно ориентированный язык). DSL это язык внутри языка который компилируется или интерпритируется внутри языка, в нашем случае Go. Если API хорошо спроектирован, DSL выглядит как его собственная спецификация специально созданная для конкретной задачи. DSL такие как CSS, SQL созданы как отдельный язык с их собственными анализаторами, но на данный момент мы сосредоточимся на одном, который построим с помощью компилятора Go, и используем внутри Go кода.

DSL используется для инфраструктурной автоматизации, модель объявления данных, построения запросов и тонны других приложений. Он может быть приятен для написания DSL для задач типа "подключил и настроил" потому, что они предлагают декларативный синтакс. Чаще чем инперативное описание, вся логика и операции нужны чтобы создать у приложения определенное состояние. DSL позволяет вам объявить желаемое состояние,параметры этого состояния и их реализацию ниже, всего лишь в шаге друг от друга. Конечный код тоже будет старатся выглядеть проще для понимания.

Мы собираемся посмотреть на то, как построить API, которое будет понятно для Go комплиятора, но выглядеть оно будет как отдельный язык. Название этой статьи выбрано потому, что мы собираемся нарушить дух Go только слегка, чтобы поправить букву закона так как нам хочется. Я надеюсь вы собираетесь быть рассудительны при применении сомнительных практик обсуждаемых далее. Я так же надеюсь изучая их, вы будете вдохновлены думать креативно о том, как писать выразительные Go API с которыми будет приятно работать и легко понимать.

Пример использования.

Мы собираемся построить просто DSL для создания HTTP посредника. Эта область отличный кандидат для DSL потому что оно наполнено общими, хорошо понимаемыми и часто переиспользуемыми шаблонами как ограничени доступа, частого ограничения, обработки сессии и так далее. Лучше всего для читалей и писателей кода который реализует эти шаблоны, если код будет читаться как декларативный файл настроек, чем как императивное изобретение колеса.

Личные типы.

Один из тонких но мощных моментов написания Go кода с образным чувством личности типов. Большую часть времени когда мы объявляем свой тип в Go, мы объявляем структуру или интерфейс. Мы так же можем объявить новый личный тип для существующих, ссылаясь на них с именем которое мы выбрали.

Это может быть чем-то простым как создание нового имени простопу типу, такому как `string` для названий хостов:

```
type Host string
```

Мы также можем создать тип для набора:

```
type HostList []Host
type HostSet map[Host]interface{}
```

Теперь где нибудь в нашем коде, переменная типа `HostList` будет `[]Host`, или на самом деле `[]string` под капотом, но с более понятным имененм.

Выгоду от таких типов, кроме как внешнего вида и сохраненных нажатий клавиш, это то что эти типы могут быть расширены с помощью их собственных типов. На пример:

```
func (s HostSet) Add(n Host) {
	s[n] = struct{}{}
}

func (s HostSet) Remove(n Host) {
	delete(s, n)
}

func (s HostSet) Contains(n Host) bool {
	_, found := s[n]
	return found
}
```

Как мы можем использовать `HostSet` как если бы это был более сложная структура контейнера доступная через методы:

```
func main() {
	s := make(HostSet)
	s.Add("golang.org")
	s.Add("google.com")
	s.Add("gopheracademy.org")
	s.Remove("google.com")

	hostnames := HostList{
		"golang.org",
		"google.com",
		"gopheracademy.org",
	}
	for _, n := range hostnames {
		fmt.Printf("%s? %v\n", n, s.Contains(n))
	}
}
```

Вот что мы получим на выходе:

```
golang.org? true
google.com? false
gopheracademy.org? true
```

Что мы тут имеем? Мы созадли абстракцию над простой мапой, мы можем использовать её как набор - у нее есть `Add` и `Remove` операции, и у нее есть проверка `Contains` - созданные нами обороты речи, которые могут быть использованны нами в коде. Он лучше изолирован, чем простая передача `map[string]interface{}` и надеясь что значение "набор имен хостов" соблюдается при доступе к map. Это решение более гибкое и явное, чем скажем:

```
func SetContains(s map[string]interface{}, hostname string) bool {
	_, found := s[hostname]
	return found
}

func main() {
	s := make(map[string]interface{})
	if SetContains(s, hostname) {
		// do stuff
	}
}
```

Поэкспериментируем чуток с созданием новых типов, частично для различных типов срезов и мапов, и даже для каналов. Какие обороты вы можете создать чтобы заставить работать эти типы проще и понятнее?

Высокоуровневые функции.

Go включает некоторые идеи из функционального программирования которое бесценно для создания выразительных и декларативных API. Go предлагает возможность присваивать функции переменных, для передачи функций как аргумент в другую функцию и для

создания анонимных функций и закрытий. Используя высокоуровневые функции которые создают, изменяют, или строят поведение других функций, вы можете легко объединять кусочки логики и функциональности во что-то более сложное целое используя несколько выражений, чаще чем повторение или создание клубка условной логики.

Давайте построим наш пример выше по-новому. Добавим метод в `HostList` который принимает функцию как входной параметр и возвращает новый `HostList`:

```
func (l HostList) Select(f func(Host) bool) HostList {
	result := make(HostList, 0, len(l))
	for _, h := range l {
		if f(h) {
			result = append(result, h)
		}
	}
	return result
}
```

Этот метод `HostList` имеет эффективность создания нового `HostList` для которого предоставленное условие (func f) верно. Создадим простое выражение функции для вставки в `f`:

```
// import “strings”
func IsDotOrg(h Host) bool {
	return strings.HasSuffix(string(h), ".org")
}
```

И используем его в наше новом методе `HostList`

```
myHosts := HostList{"golang.org", "google.com", "gopheracademy.org"}
fmt.Printf("%v\n", myHosts.Select(IsDotOrg))
```

Вывод будет таким:

```
[golang.org gopheracademy.org]
```

`Select` возвращает только те элементы `myHosts` для которых переданная функция `IsDotOrg` будет возвращать `true`, то есть для тех имен у которых есть ".org".

`func(Host) bool` немного кривовата как параметричечкий тип и создает подпись метода `Select` сложно для чтения, поэтому давайте используем наш трюк для типов, чтобы сделать его аккуратным.

```
type HostFilter func(Host) bool
```

Он делает `Select` более читаемым:

```
func (l HostList) Select(f HostFilter) HostList {
        //...
}
```

и добавляет выгоду с которой мы можем объявить некоторые методы `HostFilter`:

```
func (f HostFilter) Or(g HostFilter) HostFilter {
    return func(h Host) bool {
        return f(h) || g(h)
    }
}

func (f HostFilter) And(g HostFilter) HostFilter {
    return func(h Host) bool {
        return f(h) && g(h)
    }
}
```

Если мы хотим объявить функцию которая может использовать `HostFilter` метода, к сожалению нам нжуно пойти другим путём, чтобы это сделать. Чтобы функция имела верный приемни `HostFilter` метода, недостаточно совпадать описанию `HostFilter`, нам нужно объявить функцию как `HostFilter` явно.

```
var IsDotOrg HostFilter = func(h Host) bool {
return strings.HasSuffix(string(h), ".org")
}
```

Но теперь стало ясно что мы начали делать хорошо для нашей угрозы злоупотребления синтаксисом Go. Объявление функции с помощью передачи анонимной функции в переменную дает неясное ощущение. Отметим что это не требование использовать

высокоуровневую функцию или преимущество типа для подписи функции - любоая `func(Host) bool` может быть назначения переменной `HostFilter` или параметру. Это безрассудное объявление нужно только чтобы можно было использовать функции такие как `IsDotOrg` как приемник `HostFilter` метода

Однако выгода, заключается в том, что использование методов в `HostFilter` функции позволяет получить нам интересный синтаксис:

```
var HasGo HostFilter = func (h Host) bool {
    return strings.Contains(string(h), "go")
}

var IsAcademic HostFilter = func(h Host) bool {
    return strings.Contains(string(h), "academy")
}

func main() {
    myHosts := HostList{"golang.org", "google.com", "gopheracademy.org"}
    goHosts := myHosts.Select(IsDotOrg.Or(HasGo))
    academies := myHosts.Select(IsDotOrg.And(IsAcademic))

    fmt.Printf("Go sites: %v\n", goHosts)
    fmt.Printf("Academies: %v\n", academies)
}
```

Запустим:

```
Go sites: [golang.org google.com gopheracademy.org]
Academies: [gopheracademy.org]
```

Мы можем увидеть язык приобретает форму выражений типа `myHosts.Select(IsDotOrg.Or(HasGo))`. Он читается как английский, что-то подобное вы можете услышать на болотах Дагобы. Декларативный синтаксис, начинает появляться - выражения говорять больше о желаемом результате(“select the elements of myHosts that are .orgs or contain ‘Go’) нежели шаги которые требуется чтобы до этого добраться. Мы использовали высокоуровневые функции, `Select`, `And` и `Or`, для построения поведения от этих трех различных кусоков кода в полностью динамичном виде.

Это очень могущественный вид выражения поведения, но все эти методы очереди могут стать запутанными.

```
// etc.
myHosts.Select(IsDotOrg.Or(HasGo).Or(IsAcademic).Or(WelcomesGophers).And(UsesSSL)
```

Поэтому возмож мы должны очитсть вещи используя вариативные метод:

```
var HostFilter Or = func (clauses ...HostFilter) HostFilter {
    var f HostFilter = nil
    for _, c := range clauses {
        f = f.Or(c)
    }
    return f
}
```

И затем переписать очередь вызовов выше таким образом:

```
myHosts.Select(Or(IsDotOrg, HasGo, IsAcademic, WelcomesGophers).And(UsesSSL))
```

```
Предупредение: эти функциональные сущности самые опасные свойства Go - любой Go код который я читал или писал должен злоупотреблять этими возможностями, создание анонимных функци и передача их через слой за слой с косвенным обращением.
```

Одноко, динамика функционального стиля программирования бесценна, когда строишь свой язык внутри Go. Высокоуровневые функции, функции которые управляют другими функциями и возвращают целую новую фунцкию, дают возможно сочинять или параметризовать поведение. Цель создания DSL - упростить решение в классе проблем выставив пользоватлю DSL несколько ползеных идей для решения этих проблем, расширив их и объединив эти идеи в какой-то смысл. Созданное динамическое поведение созднное с помощью высокоуровневых функций это одна из возможностей доставить эту функциональность.

Пример по серьёзнее

Мы построим нашу работу над именами хостов так, чтобы сделать что-то ближе к тому, что мы возможно используем в реальном приложении. Ипмортируем пакет `net/http`, и давайте создадим другой тип:

```
type RequestFilter func(*http.Request) bool
```

Мы можем использовать `RequestFilter` в простом HTTP сервере для вычисления удовлетворяет `http.Request` услови, как мы это делали с `HostFilter` выше. Мы можем использовать эти условия, чтобы определить обработать или отбросить запрос.

Мы перейдем от имен хосто к работе с набором ip адресов. Будем использовать CIDR блоки типа "192.168.0.0/16", который определяет набор ip адресов, в данном случае, от 192.168.0.0 до 192.168.255.255. Создадим `RequestFilter` который фильтрует запросы основанные на ip.

Из `net` пакета, будем использовать `ParseCIDR` функцию, и `PaeseIP` чтобы анализировать входящие запросы. Значения которые возвращает `ParseCIDR` - `IPNet` который имеет удобный метод `Contains`, он будет нам говорить входит ли ip в наш список CIDR блоков.

Давайте импортируем `net` пакет и напишем `RequestFilter` который принимает вариативный набор CIDR блоков в формате `string`:

```
func CIDR(cidrs ...string) RequestFilter {
	nets := make([]*net.IPNet, len(cidrs))
	for i, cidr := range cidrs {
		// TODO: handle err
		_, nets[i], _ = net.ParseCIDR(cidr)
	}
	return func(r *http.Request) bool {
		// TODO: handle err
		host, _, _ := net.SplitHostPort(r.RemoteAddr)
		ip := net.ParseIP(host)
		for _, net := range nets {
			if net.Contains(ip) {
				return true
			}
		}
		return false
	}
```

```
}
```

Заметим, что `net/http` пакет уже содержит тип HTTP обработчика, `HandlerFunc`:

```
type HandlerFunc func(ResponseWriter, *Request)
```

А мы будем использовать высокоуровневую функцию и наш `RequestFilter` для изменения `http.HandlerFuncs`, объявим тип для функции которая обрабатывает `http.HandlerFuncs`:

```
type Middleware func(http.HandlerFunc) http.HandlerFunc
```

И давайте сделаем несколько функций чтобы построить промежуточный слой используя `RequestFilter`:

```
func Allow(f RequestFilter) Middleware {
   return func(h http.HandlerFunc) http.HandlerFunc {
   □return func(w http.ResponseWriter, r *http.Request) {
   □□if f(r) {
   □□□h(w, r)
   □□} else {
   □□□// TODO
   □□□w.WriteHeader(http.StatusForbidden)
   □□}
   □}
   }
}
```

Теперь для примера вы можете изменить http обработчик `MyHandler` так, чтобы он прининмал только запросы от `127.0.0.1`, следующим образом:

```
filteredHandler := Allow(CIDR("127.0.0.1/32"))(MyHandler)
```

Давайте попробуем запустить простой сервер:

```
func hello(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello\n")
}

func main() {
```

```
    http.HandleFunc("/hello", Allow(CIDR("127.0.0.1/32")(hello))
	log.Fatal(http.ListenAndServe(":1217", nil))
}
```

Если мы перейдем по ссылке с локальной машины: `http://0.0.0.0:1217/hello`, то вы должны увидеть "Hello" в ответ, если вы зайдете на эту ссылку с другого ip, то увидите ошибку `403 Forbidden error`.

Для прикола, давайте добавим другой вид `RequestFilter` который реализует реально просто механизм аутентификации

```
func PasswordHeader(password string) RequestFilter {
	return func(r *http.Request) bool {
		return r.Header.Get("X-Password") == password
	}
}
```

И один на основе HTTP метода:

```
func Method(methods ...string) RequestFilter {
	return func(r *http.Request) bool {
		for _, m := range methods {
			if r.Method == m {
				return true
			}
		}
		return false
	}
}
```

Ну и промежуточный слой который что-то просто логирует

```
func Logging(f http.HandlerFunc) http.HandlerFunc {
	return func(w http.ResponseWriter, r *http.Request) {
		fmt.Printf("[%v] - %s %s\n", time.Now(), r.Method, r.RequestURI)
		f(w, r)
	}
}
```

Который мы можем попробовать обновив наш сервер:

```
func main() {
    http.HandleFunc("/hello", Logging(Allow(CIDR("127.0.0.1/32")(hello)))
log.Fatal(http.ListenAndServe(":1217", nil))
}
```

Запустите сервер и посетите страницу `http://localhost:1217/hello` несколько раз в браузере и в консоли сервера вы увидите:

```
[2016-12-14 07:42:12.022266374 -0500 EST] - GET /hello
[2016-12-14 07:42:14.537985456 -0500 EST] - GET /hello
[2016-12-14 07:42:24.220089221 -0500 EST] - GET /hello
```

Синтаксис декларативный как есть, но метод цепочки может быть немного неуклюж. Метод должен быть упорядочем в правильном порядке чтобы вести себя правильно, а результат может быть слолжно читать.

Мы можем использовать структуры для дальнейшего раскрыватя DSL и дать нашим пользвателям даже более ясную возможность объявить их конфигурацию промежуточного слоя.

```
type Filters []RequestFilters
type Stack []Middleware
type Endpoint struct {
Handler    http.HandlerFunc
Allow      Filters
Middleware Stack
}
```

Затем, мы можем выразить эндпоинт выше тем же ограничением как:

```
var MyEndpoint = Endpoint{
Handler: hello,
Allow: Filters{
CIDR("127.0.0.1/32"),
},
Middleware: Stack{
Logging,
```

```
},
}
```

Который в разы проще писать, читать и изменять. Нам просто нужно добавить несколько методов в нашу структуру и типы превращаются из декларативных в удобные `http.HandlerFunc`

```
// Combine creates a RequestFilter that is the conjunction
// of all the RequestFilters in f.
func (f Filters) Combine() RequestFilter {
	return func(r *http.Request) bool {
		for _, filter := range f {
			if !filter(r) {
				return false
			}
		}
		return true
	}
}

// Apply returns an http.Handlerfunc that has had all of the
// Middleware functions in s, if any, to f.
func (s Stack) Apply(f http.HandlerFunc) http.HandlerFunc {
	g := f
	for _, middleware := range s {
		g = middleware(g)
	}
	return g
}

// Builds the endpoint described by e, by applying
// access restrictions and other middleware.
func (e Endpoint) Build() http.HandlerFunc {
	allowFilter := e.Allow.Combine()
	restricted := Allow(allowFilter)(e.Handler)

	return e.Middleware.Apply(restricted)
}
```

И наконец, изменим сервер чтобы использовать новый сервер:

```
func main() {
	http.HandleFunc("/hello", mw.MyEndpoint.Build())
	log.Fatal(http.ListenAndServe(":1217", nil))
}
```

Чтобы увидеть выходу этого миниDSL что мы создали, добавим еще один промежуточный слой:

```
func SetHeader(key, value string) Middleware {
    return func(f http.HandlerFunc) http.HandlerFunc {
        return func(w http.ResponseWriter, r *RequestFilter) {
            w.Header().Set(key, value)
            f(w, r)
        }
    }
}
```

И затем добавим его, вместе с другим `RequestFilter` в наш эндпоинт:

```
var MyEndpoint = Endpoint{
	Handler: hello,
	Allow: Filters{
		CIDR("127.0.0.1/32"),
		PasswordHeader("opensesame"), // added
		Method("GET"), // added
	},
	Middleware: Stack{
		Logging,
		SetHeader("X-Foo", "Bar"), // added
	},
}
```

МЫ добавили существенности в сложность `MyEndpoint` без добавления множества сложности в его объявление.

Этот полезный DLS удобен для построения одного HTTP эндпоинт, но часто мы хотим больше чем просто один сервис. Мы добавим еще один элемент в наше DSL, способ создать несколько маршрутов и их ендпоинты за раз:

```
type Routes map[string]Endpoint

func (r Routes) Serve(addr string) error {
	mux := http.NewServeMux()
	for pattern, endpoint := range r {
		mux.Handle(pattern, endpoint.Build())
	}

	return http.ListenAndServe(addr, mux)
}
```

И наш сервис превращается в:

```
func main() {
	routes := Routes{
		"/hello": {
			Handler: hello,
			Middleware: Stack{
				Logging,
			},
		},
		"/private": {
			Handler: hello,
			Allow: Filters{
				CIDR("127.0.0.1/32"),
				PasswordHeader("opensesame"),
			},
			Middleware: Stack{
				Logging,
			},
		},
		"/test": {
			Handler: hello,
			Middleware: Stack{
				Logging,
				SetHeader("X-Foo", "Bar"),
			},
		},
	}
	log.Fatal(routes.Serve(":1217"))
```

```
}
```

Обратите внимание, что Go автоматически определяет тип структурных литералов конечной точки в карте маршрутов, избавляя нас от лишнего набора текста и беспорядка.

HTTP промежуточный слой DSL показывает как много может удаваться в относительно маленьком наборе Go, но это простой пример. Вот несколько задания для раширения DSL и для того чтобы сделать его более мощным.

- Реализовать дополнительный `ReqeustFilters` как ограничитель частоты, возможно использовать golang.org/x/time/rate или juju/ratelimit, или более сложный механизм аутентификации
- Реазиловать другой промежуточный слой
- Изменить структуру эндпоинта чтобы включить полу `Deny` для типа `Filters`, это будет отвергать запрос если одно из полей `RequestFiltesr` - `true`
- Каждый эндпоинт в конечном примере включает логирование в промежуточном слое, добавьте в DSL средство для применения набора общих ограничений или промежуточный слой для всех энпоинтов.
- Создать способ для стека промежуточного слоя, чтобы создать `context.Context` и работать с обработчиками которые их принимают.

Резюмируем, мы использовать типы для создания абстрактнций поверх наборов простых типов и функций отдельных подписей, и мы берем преимущество Go свойств синтаксиса такие как вариативные функции и приведение типов для написания спокойного и ненагроможденного синтаксиса. Тяжелый подъем в создании DSL был произведен с помощью функций высшего порядка который позволили параметризовать поведения объединеные и настроены во время работы. Мы использовали несколько опасных практик написания кода, но чем больше мы применяем их только когда сокращаем сложность для конечного пользователя, тем крепче мы можем спать ночью.

Go, который вы получаете из коробки ориентирован на детали, минималистичный, и может быть довольно подробным. Go дает инструменты, однако, чтобы построить вашу собственную абстракцию, ваш собственный высокоуровневый язык для написания кода который содержательный, элегантный, и выразительный как и любой другой который вы находите в динамичном или чистом функциональном языке, но это дает нам доступ ко всем свойствам который мы любим в Go.

Revision #8

Created 2021-07-23 07:03:48 UTC by gasick

Updated 2023-04-16 19:30:03 UTC by gasick