مقدمه

فرض کن ما داریم یه سری **پکیج** می‌نویسیم که یه هدف خاص رو دنبال می‌کنن.
مثلاً یه مجموعه پکیج برای کار با **درگاه پرداخت به‌پرداخت**:

- پکیج client برای ارسال request
- پکیج response برای تجزیه جواب‌ها
- پکیج validator برای اعتبارسنجی داده‌ها
- و...

همه‌ی این‌ها با هم یه مجموعه منسجم هستن که قراره به بقیه هم کمک کنن.

حالا می‌خوایم این پکیج‌هارو در اختیار بقیه بذاریم

یه راه خیلی ساده:

بریم کل پوشه‌ی پکیج‌هامونو zip کنیم، آپلود کنیم تو یه سایت و هر کسی خواست استفاده کنه، بیاد دانلودش کنه و بندازه تو پوشه‌ی پروژه خودش

خب، ساده‌ست، ولی بریم ببینیم ایراداش چیه...

ایرادات روش دستی

1-مشکل وابستگی‌های چندگانه

اگه پکیج خودمون به یه پکیج دیگه‌ای وابسته باشه ما باید اون رو هم **دانلود کنیم، نگه داریم، و کنار پروژه‌مون بذاریم** ،یعنی نه تنها کد خودتو باید توزیع کنی، بلکه همه‌ی وابستگی‌هات رو هم باید باهاش بچسبونی

2-تکرار وابستگی‌ها

وقتی چند پکیج مختلف به یک وابستگی مشترک نیاز دارن، ولی هر کدوم نسخه‌ی متفاوتی از اون وابستگی رو داخل خودشون نگه‌داری می‌کنن، ممکنه پروژه‌ی نهایی دچار تداخل و ناسازگاری بشه.

برای مثال:

Marry یه پکیج توسعه داده که به یه پکیجی به اسم tools نیاز داره. این پکیج tools رو **Mark** در سال 2023 منتشر کرده بوده، و Marry اون نسخه رو دانلود کرده و داخل پروژه‌ی خودش قرار داده.

Ryan هم یه پکیج دیگه توسعه داده که اونم به tools نیاز داره. اما این بار نسخه‌ی جدیدتری از tools رو که **Mark** در سال 2024 منتشر کرده (با یه‌سری تغییرات)، دانلود کرده و داخل پروژه‌ی خودش قرار داده.

حالا تو به هر دو پکیجی که توسط Marry و Ryan توسعه داده شدن نیاز داری. با اضافه کردن این دو پکیج به پروژه‌ت، در واقع داری **دو نسخه‌ی متفاوت از** tools رو به‌صورت ضمنی وارد ساختار پروژه می‌کنی.

این وضعیت می‌تونه باعث بشه:

- فایل‌های مشابه با رفتار متفاوت در پروژه وجود داشته باشن
- توابع یا ساختارهایی که انتظار داری، در یکی از نسخه‌ها حذف یا تغییر کرده باشن
- ناسازگاری در نوع داده‌ها یا امضاهای توابع به وجود بیاد
- در زمان کامپایل یا اجرا خطاهای غیرمنتظره دریافت کنی
- و در نهایت، توی importها به مشکل بخوری و مجبور بشی دستی alias تعریف کنی یا ساختار پروژه رو تغییر بدی تا Go بفهمه منظورت کدوم نسخه‌ست

3-پیچیدگی در نگهداری

هروقت بخوایم یه خط کد رو تغییر بدیم یا یه باگ کوچیک رو درست کنیم:

- باید دوباره zip کنیم
- آپلود کنیم
- به همه خبر بدیم که "هی بچه‌ها! نسخه‌ی جدید اومد، خودتون دستی دانلود کنید"

اینجاست که Go Modules وارد میشه

Go اومد گفت:

من یه سیستم رسمی می‌سازم به اسم Module
که بتونی:

- پروژه‌تو استاندارد بسته‌بندی کنی
- وابستگی‌هاتو ثبت کنی
- با یه دستور (go get) همه چیزو خودم هندل کنم!

امکانات مهم ماژول

ویژگی	توضیح
go.mod	توش مشخص می‌کنی اسم ماژولت چیه و به چی وابسته‌ست
go.sum	Go به صورت خودکار چک‌سام و نسخه دقیق وابستگی‌ها رو نگه می‌داره
go get	آدرس ماژول رو بده، Go خودش می‌گیره و در مسیر مناسب قرار می‌ده
Versioning	می‌تونی نسخه‌گذاری (v1.0.0, v2.3.1, ...) انجام بدی
Reproducibility	پروژه‌ت در هر جا و هر زمانی با همون وابستگی‌ها build میشه

جمع بندی

ماژول‌ها تو Go اومدن که یه بار برای همیشه بساط نسخه‌بندی دستی، کپی‌پیست، تداخل پکیج‌ها و جهنم dependency ها رو جمع کنن.

اگه پکیج‌هاتو با module منتشر کنی، دیگه نه فقط خودت راحتی، بلکه همه‌ی اونایی که از کدت استفاده می‌کنن هم نفس راحت می‌کشن

بررسی یک پروژه بدون استفاده از ماژول

فرض کن یه اپلکیشن به زبان GO برای یه سیستم فروش اینترنتی داریم که از دو درگاه بانکی استفاده می‌کنه: یکی از طرف **Behpardakht** که توسط توسعه‌دهنده‌ای به نام **John** نوشته شده، و یکی دیگه از طرف **Saman** که کار **Liam** بوده. هر کدوم پروژه‌شون رو به صورت فایل زیپ منتشر کردن، و ساختار پروژه‌هاشون هم به شکل زیره:

ساختار پروژه behpardakht:

```
behpardakht
├── client
│   └── client.go
└── packages
    └── xml
        └── store.go
```

ساختار پروژه saman:

```
saman
├── client
│   └── client.go
└── packages
    └── xml
        └── store.go
```

طبق یه قرارداد نانوشته بین توسعه‌دهنده‌ها، هر کسی که پکیجش به یه سری کد وابسته‌ست، اون وابستگی‌ها رو داخل فولدر packages در مسیر اصلی پروژه‌ش قرار می‌ده .ما هم که می‌خوایم از این دو درگاه توی پروژه‌ی خودمون استفاده کنیم، میایم و هر دو رو داخل فولدر packages پروژه‌ی خودمون کپی می‌کنیم.

ساختار پروژه Ecommerce

```
ecommerce
├── main.go
└── packages
    ├── behpardakht
    │   ├── client
    │   │   └── client.go
    │   └── packages
    │       └── xml
    │           └── store.go
    └── saman
        ├── client
        │   └── client.go
        └── packages
            └── xml
                └── store.go
```

حالا می‌ریم سراغ main.go تا از این دو درگاه استفاده کنیم:

```go
package main


import (
    "ecommerce/packages/behpardakht/packages/xml"
    "ecommerce/packages/saman/packages/xml"
)


func main() {
    xml.BuildXMLRequest("Behpardakht")
    xml.BuildXMLRequest("Saman")
}
```

و اینجاست که Go بهمون اخطار می‌ده:

```
.\main.go:5:2: xml redeclared in this block
        .\main.go:4:2: other declaration of xml
.\main.go:5:2: "ecommerce/packages/saman/packages/xml" imported and not used
```

مشکل چیه؟

خیلی ساده‌ست: هر دو درگاه به پکیج xml خودشون وابسته‌ن .و چون هر کدوم این پکیج رو جداگانه توسعه دادن، ممکنه یکی‌شون از نسخه‌ی 2023 استفاده کنه و اون یکی از نسخه‌ی 2024 که یه سری تغییرات داشته. ما نمی‌تونیم فقط یکی رو ایمپورت کنیم و انتظار داشته باشیم اون یکی هم با همون کار کنه—این یه تداخل کلاسیکه.

راه‌حل: استفاده از alias در import

برای اینکه بتونیم هر دو نسخه‌ی xml رو همزمان استفاده کنیم، باید به هر کدوم یه اسم مستعار بدیم:

```
package main


import (
    behXML "ecommerce/packages/behpardakht/packages/xml"
    samXML "ecommerce/packages/saman/packages/xml"
)


func main() {
    behXML.BuildXMLRequest("Behpardakht")
    samXML.BuildXMLRequest("Saman")
}
```

و اینطوری مشکل حل می‌شه. هر کدوم از درگاه‌ها با نسخه‌ی خودش از xml کار می‌کنه، بدون اینکه تداخلی پیش بیاد.

نکته پایانی

این فقط یکی از دردسرهاییه که وقتی بدون استفاده از **Go Modules** پکیج‌ها رو دستی مدیریت می‌کنی، باهاش مواجه می‌شی. ماژول‌ها نه‌تنها مسیرها رو واضح‌تر می‌کنن، بلکه نسخه‌بندی، وابستگی‌ها، و ایمپورت‌ها رو هم خیلی تمیزتر هندل می‌کنن.

استفاده از قابلیت های Go Module

حالا بریم ببینیم چطوری می‌تونیم قابلیت‌های مختلف **Go Module** رو روی پروژه‌ای که قبلاً داشتیم (یعنی همون phonebook) عملی کنیم و پروژه رو به‌روز کنیم.

1-تغییر اسم پروژه

اول از همه، فولدر اصلی پروژه رو که قبلاً اسمش exercise بود، می‌ذاریم phonebook یعنی فولدر رو از exercise به phonebook تغییر نام می‌دیم.

2-تغییر نام ماژول در go.mod

حالا می‌ریم توی ریشه پروژه (phonebook) و فایل go.mod رو باز می‌کنیم.

تو این فایل، اسم ماژول رو که قبلاً exercise بود، تغییر می‌دیم به phonebook.

این خط :

```
module exercise
```

میشه:

```
module phonebook
```

3-به‌روزرسانی import ها

حالا باید بگردیم توی کل پروژه و هر جایی که توی دستورهای import اسم exercise بود، اون رو با phonebook جایگزین کنیم.

phonebook/phone_book/report.go

```
import "phonebook/table"
```

phonebook/phone_book/store.go

```
import "phonebook/validator"
```

phonebook/application/main.go

```
import "phonebook/phone_book"
```

استفاده از پکیج

برای اینکه از پکیج‌هایی که دیگران توسعه دادن داخل پروژه‌مون استفاده کنیم (مثلاً پکیج‌های معروفی مثل uuid، gin، validator و ...) می‌تونیم از دستور زیر توی ترمینال استفاده کنیم:

```
go get [package_address]@[version]
```

مثال

```
go get github.com/fatih/color@v1.16.0
```

در این مثال، ما نسخه‌ی v1.16.0 از پکیج color رو از GitHub می‌گیریم.

اگه نسخه رو ننویسی، Go خودش می‌ره آخرین نسخه‌ی پایدار (latest stable) رو می‌گیره:

```
go get github.com/fatih/color
```

وقتی موقع go get نسخه‌ی پکیج رو مشخص نمی‌کنی Go می‌ره سراغ **آخرین نسخه‌ی stable** اون پکیج، ولی سعی می‌کنه یه کاری کنه که با نسخه‌ی Go پروژه‌ات (که تو go.mod نوشتی) **سازگار باشه**.

اما نکته اینجاست:
Go تضمین صددرصدی برای سازگاری نسخه‌ی پکیج با نسخه‌ی Go پروژه‌ات نداره. یعنی ممکنه نسخه‌ای از پکیج که به‌عنوان آخرین نسخه پایدار انتخاب می‌کنه، از قابلیت‌ها یا سینتکس‌هایی استفاده کنه که توی نسخه‌ی Go پروژه‌ات پشتیبانی نمی‌شه یا باعث بروز خطا و ناسازگاری بشه.

بهترین کار چیه؟

همیشه موقع go get، نسخه‌ی پکیج رو مشخص کن:

اینجوری:

- کنترل کامل داری رو اینکه چه نسخه‌ای نصب می‌شه
- با نسخه‌ی Go پروژه‌ت چک کردی که سازگاره
- پروژه‌ت قابل پیش‌بینی می‌مونه و راحت‌تر تو تیم کار می‌کنی

پشت پرده دستور go get

1. پکیج موردنظر از اینترنت دانلود می‌شه و در کش Go ذخیره می‌شه (نه داخل پروژه).
2. توی فایل go.mod یک خط require اضافه می‌شه که مشخص می‌کنه پروژه‌ت به اون پکیج نیاز داره.
3. توی فایل go.sum چک‌سام اون پکیج ذخیره می‌شه تا Go بتونه در آینده مطمئن بشه نسخه تغییر نکرده یا خراب نشده.
4. حالا می‌تونی خیلی راحت اون پکیج رو با import توی کدت بیاری و ازش استفاده کنی.

فایل go.mod چیه؟

یه جور **نقشه‌ی اصلی** برای مدیریت وابستگی‌های پروژه‌ته. توش Go نگه می‌داره:

- این پروژه با کدوم نسخه‌ی Go نوشته شده
- به کدوم پکیج‌ها (و کدوم نسخه‌هاشون) نیاز داره

وقتی go get می‌زنی:

مثلاً:

```
go get github.com/fatih/color@v1.16.0
```

خط زیر به go.mod اضافه می‌شه (یا آپدیت می‌شه):

```
require github.com/fatih/color v1.16.0
```

یعنی: این پروژه برای اجرا شدن، به این نسخه از این پکیج نیاز داره.

ممکنه پکیج‌های **غیراصلی** که این پکیج بهشون وابسته‌ست (dependencies transitive) هم بیان تو فایل.

محتوای فایل go.mod پس از نصب پکیج faith/color در سناریوی واقعی و تست شده:

```
module phonebook

go 1.24.4

require (
	github.com/fatih/color v1.16.0 // indirect
	github.com/mattn/go-colorable v0.1.13 // indirect
	github.com/mattn/go-isatty v0.0.20 // indirect
	golang.org/x/sys v0.14.0 // indirect
)
```

معنای indirect در فایل go.mod

وقتی توی بخش require فایل go.mod کنار یه پکیج این عبارت می‌بینی:

```
github.com/fatih/color v1.16.0 // indirect
```

یعنی این پکیج **مستقیماً توی کد پروژه‌ی خودت import نشده**، ولی:

- **یکی از پکیج‌هایی که تو خودت مستقیم وارد کردی، به این پکیج وابسته است**، غیر مستقیم یا (transitive dependency)
- Go این پکیج رو برای کامل بودن و سازگاری پروژه می‌ذاره، ولی چون تو مستقیماً ازش استفاده نکردی، نشون می‌ده که indirect هست

ولی به محض اینکه خودت بری و اون پکیج رو توی یه فایل import کنی، دیگه از حالت indirect در میاد و تبدیل می‌شه به یه وابستگی مستقیم.

استفاده از پکیج faith/color

هدفمون اینه که با کمک توابع کتابخونه‌ی fatih/color، رنگ کادر دور جدول رو (که با پکیج table ساخته بودیم) از سفید بی‌رنگ دربیاریم و سبزش کنیم!

برای این کار، یه سری تغییر کوچیک تو فایل phonebook/table/draw.go می‌دیم:

phonebook/table/darw.go

```
// PrintRow prints a row of values centered in columns with a fixed width.
func PrintRow(values []string, width int) {
	color.New(color.FgGreen).Print("|")
	for _, val := range values {
		fmt.Print(padCenter(val, width))
		color.New(color.FgGreen).Print("|")
	}
	fmt.Println()
}

func PrintLine(columns int, width int) {
	line := "+"
	for i := 0; i < columns; i++ {
		line += strings.Repeat("-", width) + "+"
	}
	color.Green(line)
}
```

حالا اگه این دستور رو تو ترمینال بزنیم:

```
go run ./application
```

می‌بینیم که جدولمون با رنگ سبز جذاب نمایش داده می‌شه. یعنی دقیقاً همون چیزی که می‌خواستیم.

ولی بریم یه نگاهی هم به فایل go.mod بندازیم:

```
module phonebook

go 1.24.4

require (
	github.com/fatih/color v1.16.0 // indirect
	github.com/mattn/go-colorable v0.1.13 // indirect
	github.com/mattn/go-isatty v0.0.20 // indirect
	golang.org/x/sys v0.14.0 // indirect
)
```

با اینکه ما از fatih/color مستقیم استفاده کردیم، ولی هنوز کنارش نوشته // indirect

اینجا یه نکته‌ی فنی وجود داره: احتمال زیاد Go هنوز تغییراتی که ما تو کدمون دادیم رو کامل نفهمیده! یعنی هنوز نفهمیده که داریم مستقیم از color استفاده می‌کنیم.

توی همچین مواقعی خودمون باید به صورت دستی با فراخوانی دستور go mod tidy این فایل رو به‌روز کنیم.

دستور go mod tidy

وقتی داری با Go و ماژول‌ها (go.mod) کار می‌کنی، با گذشت زمان ممکنه بعضی از پکیج‌ها رو اضافه کنی و بعداً پاکشون کنی، یا یه سری وابستگی‌های غیرمستقیم (indirect dependencies) وارد پروژه‌ت بشن که دیگه به کار نیان.

اینجاست که go mod tidy مثل یه رفتگر حرفه‌ای میاد وسط و همه‌چی رو تمیز و مرتب می‌کنه.

خلاصه کارهایی که go mod tidy انجام می‌ده:

1. **اضافه کردن وابستگی‌های فراموش‌شده**
   اگه تو کدت از یه پکیجی استفاده کرده باشی، ولی توی go.mod اسمش نیومده باشه (مثلاً چون دستی واردش نکردی)، این دستور می‌ره بررسی می‌کنه و اون پکیج رو به require اضافه می‌کنه.
2. **حذف وابستگی‌های اضافی**
   اگه یه پکیج دیگه تو کدت استفاده نمی‌شه ولی هنوز تو go.mod مونده (مثلاً قبلاً استفاده می‌کردی ولی دیگه حذفش کردی)، go mod tidy اون رو از require می‌ندازه بیرون.
3. **آپدیت فایل go.sum**
   فایل go.sum برای امنیت و اطمینان از صحت نسخه‌ی پکیج‌ها استفاده می‌شه go mod tidy بررسی می‌کنه که hash ها و امضاهای cryptographic موجود توی این فایل با واقعیت همخونی دارن یا نه. اگه نیاز باشه، چیزهایی بهش اضافه یا ازش حذف می‌کنه.

مثال کاربردی

همونطور که دیدیم، وقتی تو پکیج table از ماژول fatih/color استفاده کردیم، فایل go.mod بلافاصله به‌روزرسانی نشد. هنوز هم توش می‌بینیم که fatih/color به صورت indirect لیست شده، یعنی Go فکر می‌کنه که ما مستقیم ازش استفاده نکردیم.

```
require github.com/fatih/color v1.16.0 // indirect
```

وقتی بزنی:

```
go mod tidy
```

Go پروژه رو اسکن می‌کنه، می‌بینه عه! تو داری color.New(...).Print(...) استفاده می‌کنی. پس این دیگه indirect نیست، باید مستقیم بشه!

یک بار دیگه برگردیم سراغ فایل **go.mod** تا ببینیم چه تغییری توش ایجاد شده. بیا ببینیم دقیقاً چی شده!

```
module phonebook

go 1.24.4

require github.com/fatih/color v1.16.0

require (
	github.com/mattn/go-colorable v0.1.13 // indirect
	github.com/mattn/go-isatty v0.0.20 // indirect
	golang.org/x/sys v0.14.0 // indirect
)
```

همون‌طور که معلومه، پکیج **fatih/color** دیگه **indirect** نیست، چون مستقیم خودمون تو کدمون ایمپورتش کردیم و داریم ازش استفاده می‌کنیم. پس طبیعیه که دیگه به‌عنوان وابستگی غیرمستقیم شناخته نشه.

کی باید از go mod tidy استفاده کنیم؟

- بعد از اینکه پکیجی رو **اضافه یا حذف** کردی
- وقتی go.mod یا go.sum به نظر به‌روز نمیاد
- قبل از **کامیت نهایی** پروژه (تا فایل‌ها تمیز و مرتب باشن)
- وقتی که یه پکیج رو استفاده کردی ولی هنوز تو go.mod نرفته یا indirect مونده

فایل go.sum چیه و چرا وجود داره؟

وقتی پروژه‌ای با **Go Modules** می‌سازی، چند تا فایل مهم کنار هم کار می‌کنن تا پروژه درست و امن اجرا بشه. یکی از اون‌ها همین فایل **go.sum** هست.

نقش اصلی — go.sum مثل یه قفل امنیتی

go.sum مثل یه دفترچه ثبت هست که **برای هر نسخه از هر پکیجی که پروژه‌ات بهش وابسته‌ست، یه امضای دیجیتال (hash) دقیق نگه می‌داره**.

چرا این «امضا» یا hash مهمه؟

فرض کن داری یه پکیج محبوب رو از اینترنت می‌گیری، مثلاً github.com/fatih/color نسخه v1.16.0
این پکیج فایل‌های زیادی داره، وقتی دانلود می‌شه ممکنه یکی از حالت های زیر اتفاق بیوفته:

- فایل‌ها دقیقاً مثل همون نسخه اصلی باشن
- کسی نسخه رو خراب کرده باشه (مثلاً یه کد مخرب بهش اضافه کرده باشه)
- به دلایل اتفاقی بعضی فایل‌ها ناقص یا خراب دانلود بشن.

حالا چطوری مطمئن بشیم که همون نسخه‌ای رو داریم که نویسنده پکیج منتشر کرده و به دست ما رسیده؟

اینجا نقش go.sum پررنگ می‌شه:

- برای هر نسخه پکیجی که با دستور go get (یا حتی با روش های دیگه) نصب می‌کنی، Go یک هش (hash) دقیق از محتوای اون نسخه و همچنین از فایل go.mod مربوط به اون پکیج محاسبه می‌کنه.
- این هش‌ها به go.sum اضافه می‌شن.

هر بار که کسی (مثلاً تو یا همکارات) پروژه رو کلون می‌کنه و می‌خواد پکیج‌ها رو دوباره دانلود کنه، Go با مقایسه هش‌های دریافتی با اون چیزی که توی go.sum ثبت شده، چک می‌کنه:

- آیا نسخه‌ی پکیج دقیقاً همونیه که قبلاً بوده؟
- یا دستکاری شده یا خراب شده؟

بررسی تغییرات فایل go.sum بعد از نصب پکیج جدید

فرض کن این دستور رو زدی:

```
go get github.com/faith/color@v1.16.0
```

1. Go این نسخه پکیج رو از اینترنت دانلود می‌کنه.
2. یه هش دقیق از محتوای اون نسخه و فایل go.modش می‌سازه.
3. این هش‌ها رو به انتهای فایل go.sum پروژه اضافه می‌کنه.
4. توی فایل go.mod هم اون نسخه به عنوان dependency ثبت می‌شه.

محتوای فایل go.sum پس از نصب پکیج faith/color در سناریوی واقعی و تست شده:

```
github.com/fatih/color v1.16.0 h1:zmkK9Ngbjj+K0yRhTVONQh1p/HknKYSlNT+vZCzyokM=

github.com/fatih/color v1.16.0/go.mod
h1:fL2Sau1YI5c0pdGEVCbKQbLXB6edEj1ZgiY4NijnWvE=

github.com/mattn/go-colorable v0.1.13
h1:fFA4WZxdEF4tXPZVKMLwD8oUnCTTo08duU7wxecdEvA=

github.com/mattn/go-colorable v0.1.13/go.mod
h1:7S9/ev0klgBDR4GtXTXX8a3vIGJpMovkB8vQcUbaXHg=

github.com/mattn/go-isatty v0.0.16/go.mod
h1:kYGgaQfpe5nmfYZH+SKPsOc2e4SrIfOl2e/yFXSvRLM=

github.com/mattn/go-isatty v0.0.20 h1:xfD0iDuEKnDkl03q4limB+vH+GxLEtL/jb4xVJSWWEY=

github.com/mattn/go-isatty v0.0.20/go.mod
h1:W+V8PltTTMOvKvAeJH7IuucS94S2C6jfK/D7dTCTo3Y=

golang.org/x/sys v0.0.0-20220811171246-fbc7d0a398ab/go.mod
h1:oPkhp1MJrh7nUepCBck5+mAzfO9JrbApNNgaTdGDITg=

golang.org/x/sys v0.6.0/go.mod h1:oPkhp1MJrh7nUepCBck5+mAzfO9JrbApNNgaTdGDITg=

golang.org/x/sys v0.14.0 h1:Vz7Qs629MkJkGyHxUlRHizWJRG2j8fbQKjELVSNhy7Q=

golang.org/x/sys v0.14.0/go.mod h1:/VUhepiaJMQUp4+oa/7Zr1D23ma6VTLIYjOOTFZPUcA=
```

نتیجه نهایی

- وقتی تو یا همکارانت بعداً پروژه رو کلون می‌کنید و می‌خواید بسازید، Go چک می‌کنه که هش‌های پکیج‌هایی که دانلود می‌کنه با هش‌های ذخیره شده توی go.sum یکی باشن.
- اگه یکی از هش‌ها تغییر کرده باشه، Go هشدار می‌ده یا خطا می‌ده و جلوی استفاده از نسخه‌های تغییر یافته یا خراب رو می‌گیره.

چرا این برای امنیت و ثبات پروژه حیاتی‌ست؟

- جلوی ورود کد مخرب یا نسخه‌های تغییر یافته رو می‌گیره.
- باعث می‌شه همه اعضای تیم دقیقا با همون نسخه‌ها کار کنن و پروژه توی همه محیط‌ها یکسان اجرا بشه.
- کمک می‌کنه که پروژه‌ها پایدار و قابل تکرار باشند (reproducible builds)

روش‌های دیگه برای نصب پکیج‌ها

تا اینجا دیدیم که چطوری با دستور go get یه پکیج رو نصب می‌کنیم. ولی خب، این تنها راهش نیست! دوتا روش دیگه هم داریم که خیلی وقتا به درد می‌خورن، مخصوصاً اگه بخوایم دقیق‌تر و دستی کار کنیم یا یه ورژن خاص از یه پکیج رو مدنظر داشته باشیم.

روش اول: اضافه‌کردن مستقیم به فایل go.mod

اگه دقیقاً می‌دونی دنبال چه پکیجی هستی و می‌خوای یه ورژن خاص ازش رو استفاده کنی، می‌تونی مستقیم بری داخل فایل go.mod و خودت این خط رو اضافه کنی:

```
require github.com/fatih/color v1.16.0
```

حالا برای اینکه این وابستگی واقعاً بره نصب شه از دستور زیر استفاده میکنی:

```
go mod download
```

این دستور فقط و فقط وظیفه‌ش **دانلود کردن** پکیج‌هاست. نه چیزی پاک می‌کنه، نه چیزی آپدیت می‌کنه! فقط می‌ره اون پکیج‌هایی که توی فایل go.mod لیست شدن رو از اینترنت می‌گیره و توی کش لوکال Go ذخیره می‌کنه.

روش دوم: ایمپورت مستقیم توی کد

اگه حوصله نداری دستی بری تو go.mod چیزی بنویسی، یه راه آسون‌تر هم هست:

همون‌جایی که نیاز داری، پکیج رو مستقیم ایمپورت کن:

```
import "github.com/fatih/color"
```

حالا فقط کافیه این دستور رو بزنی:

```
go mod tidy
```

Go خودش می‌فهمه یه پکیج جدید اضافه کردی، میره پیداش می‌کنه، توی go.mod و go.sum ثبتش می‌کنه و دانلودش هم می‌کنه. خیلی شیک و تمیز!

آپگرید پکیج ها

وقتی پروژه‌مون با چندتا پکیج و ماژول کار می‌کنه، گاهی لازم میشه نسخه‌ی یکی یا چندتا از اون‌ها رو به‌روز کنیم. این کار دو روش اصلی داره که هر کدوم مزایا و محدودیت خودش رو داره. بریم با هم بررسی‌ش کنیم.

روش اول: آپگرید فقط یه پکیج خاص

این کار دو تا حالت داره:

روش کورکورانه

یعنی بدون اینکه چک کنیم قراره به چه نسخه‌ای آپدیت بشه، فقط با یه دستور ساده می‌زنیم که Go خودش بره آخرین نسخه‌ی stable و سازگار رو برامون بیاره:

```
go get -u github.com/fatih/color
```

Go خودش میره آخرین نسخه‌ی پایدار رو پیدا می‌کنه و جایگزین نسخه قبلی می‌کنه. مثلاً تو پروژه‌ی phonebook قبلاً این پکیج اینطوری تو go.mod بوده:

```
require github.com/fatih/color v1.16.0
```

بعد از اجرای اون دستور، می‌بینیم تبدیل می‌شه به:

```
require github.com/fatih/color v1.18.0
```

روش هوشیارانه

اگه می‌خوای قبل از آپدیت کردن بدونی کدوم پکیج نسخه‌ی جدیدتری داره، می‌تونی از این دستور استفاده کنی:

```
go list -m -u all
```

این دستور نسخه‌ی فعلی هر ماژول رو با آخرین نسخه‌ی موجود مقایسه می‌کنه و اگه آپدیتی باشه، توی براکت برات نشون می‌ده.

مثلاً برای پروژه phonebook خروجی اینطوری شد:

```
phonebook
github.com/cpuguy83/go-md2man/v2 v2.0.6 [v2.0.7]
github.com/fatih/color v1.16.0 [v1.18.0]
github.com/inconshreveable/mousetrap v1.1.0
github.com/mattn/go-colorable v0.1.13 [v0.1.14]
github.com/mattn/go-isatty v0.0.20
github.com/russross/blackfriday/v2 v2.1.0
github.com/spf13/cobra v1.9.1
github.com/spf13/pflag v1.0.6 [v1.0.7]
golang.org/x/sys v0.14.0 [v0.34.0]
gopkg.in/check.v1 v0.0.0-20161208181325-20d25e280405 [v1.0.0-20201130134442-10cb98267c6c]
gopkg.in/yaml.v3 v3.0.1
```

اینجا کاملاً مشخصه که نسخه‌ی جدیدتری برای پکیج faith/color منتشر شده ولی ما هنوز از نسخه‌ی قدیمی استفاده می‌کنیم.

حالا که فهمیدیم آپدیت جدید هست، دو راه برای ارتقاش داریم:

1-روش مستقیم:

```
go get github.com/fatih/color@v1.18.0
```

2- روش دستی:

خودت برو داخل فایل go.mod، اون خط مربوط به fatih/color رو از:

```
require github.com/fatih/color v1.16.0
```

تغییر بده به:

```
require github.com/fatih/color v1.18.0
```

و بعد یکی از این دستوراتو بزن:

```
go mod download
```

یا

```
go mod tidy
```

هر دوشون کارتو راه می‌ندازن download فقط پکیج رو می‌گیره، ولی tidy علاوه‌بر اون، پکیج‌های اضافی رو هم پاک‌سازی می‌کنه و dependency هات رو مرتب می‌کنه.

روش دوم: آپگرید همه پکیج‌ها با هم

اگر می‌خوای همه ماژول‌هایی که در پروژه استفاده می‌کنی رو یکجا به آخرین نسخه سازگار آپدیت کنی، از این دستور استفاده می‌کنی:

```
go get -u ./...
```

- u- یعنی آپدیت
- ./... یعنی همه پکیج‌های داخل پروژه از روت تا زیرپوشه‌ها

این دستور همه ماژول‌های پروژه رو به نسخه‌های جدیدتر ارتقا میده (البته تا جایی که با محدودیت‌های دیگه پروژت تداخل نداشته باشن).

جمع بندی و مقایسه دو روش

مورد	روش اول (آپگرید تک پکیج)	روش دوم (آپگرید همه پکیج‌ها)
کنترل نسخه	امکان انتخاب ورژن دلخواه داری	همه به آخرین ورژن‌های سازگار
راحتی استفاده	کمی زمان‌بر، مخصوصاً برای چند پکیج	سریع و ساده برای همه ماژول‌ها
خطر ناسازگاری	کمتر، چون فقط یه پکیج رو تغییر می‌دی	ممکنه باعث ناسازگاری بین ماژول‌ها بشه
مناسب برای	وقتی فقط یه باگ یا ویژگی خاص می‌خوای	وقتی می‌خوای کل پروژه رو به‌روز کنی

نکته

- قبل از آپدیت بهتره حتما تست‌ها رو اجرا کنی و مطمئن بشی تغییر ورژن باعث شکستن کد نشه.
- به خاطر داشته باش که گاهی آپدیت‌های بزرگ (major) ممکنه تغییرات ناسازگار داشته باشن و نیاز به بازنویسی بخشی از کد داشته باشی.

دانگرید کردن پکیج ها

گاهی وقتا ممکنه یه پکیج رو آپگرید کنی و بعد چند دقیقه تست و بررسی، متوجه بشی که اون آپگرید باعث بروز مشکلاتی تو برنامه‌ت شده. مثلاً یه بخشی از کدت دیگه درست کار نمی‌کنه، یا یه باگی ظاهر شده که تو نسخه قبلی اصلاً نبود. تو همچین شرایطی، بهترین کار اینه که برگردی به نسخه قبلی — یعنی **دانگریدش** کنی.

یا حتی ممکنه یه پکیج جدید بخوای نصب کنی که اون پکیج جدید، به یه نسخه قدیمی‌تر از یه پکیج دیگه نیاز داشته باشه. اگه نسخه فعلی‌ت با اون سازگار نباشه، بازم مجبوری پکیج مورد نظر رو به عقب برگردونی تا همه‌چی با هم مچ بشه.

در کل، دانگرید کردن یعنی با دست خودت مشخص کنی که می‌خوای یه نسخه قدیمی‌تر از یه پکیج خاص رو استفاده کنی — نه آخرین نسخه‌ای که Go خودش تشخیص می‌ده.

برای اینکه یه پکیج رو دانگرید کنیم، فقط کافیه نسخه مورد نظرت رو مشخص کنی. مثلا فرض کن داریم از نسخه v1.18.0 پکیج github.com/fatih/color استفاده می‌کنیم، ولی به هر دلیلی (مثل ناسازگاری یا باگ) می‌خوایم برگردیم به نسخه v1.16.0 تو این صورت از دستور زیر استفاده می‌کنیم:

```
go get github.com/fatih/color@v1.16.0
```

بعد از اجرا، Go نسخه‌ی مورد نظر رو می‌گیره، فایل‌های go.mod و go.sum رو آپدیت می‌کنه، و شما رسماً به نسخه قبلی برگشتید.

نکته: تو فایل go.mod هم می‌تونی دستی نسخه رو عوض کنی و بعدش با دستور go mod tidy یا go mod download تغییر نسخه پکیج رو اعمال کنی.

دانگرید گروهی؟

Go دستور خاصی برای دانگرید همه‌ی پکیج‌ها با هم نداره. چون دانگرید همیشه باید با آگاهی و مشخص بودن نسخه انجام بشه، چون ممکنه کلی dependency بشکنه. برخلاف آپگرید که go get -u all داریم، برای دانگرید فقط باید تک‌به‌تک و نسخه‌به‌نسخه عمل کنیم.

انواع پکیج ها

1-پکیج‌های کتابخانه‌ای (Library Packages)

این نوع پکیج‌ها خودشون به تنهایی اجرا نمی‌شن، بلکه **یه سری قابلیت‌ها یا توابع و ساختارها رو در اختیار برنامه‌ی ما می‌ذارن** که بتونیم ازشون استفاده کنیم.

مثال معروف:

```
import "github.com/fatih/color"
```

پکیج color یه کتابخونه‌ست که بهت اجازه می‌ده تو خروجی ترمینال از متن‌های رنگی استفاده کنی. مثلاً می‌تونی یه پیام خطا رو قرمز کنی یا یه پیام موفق رو سبز.

ویژگی‌ها:

- فایل‌هاش با package someName شروع می‌شن (معمولاً همون اسم پکیجه)
- هیچ تابع ()main ندارن
- هدفشون اجرا نیست، بلکه استفاده تو برنامه‌ست
- با دستور go get نصب می‌شن

2-پکیج‌های اجرایی (Executable Packages)

این پکیج‌ها دقیقاً برای **اجرا شدن** طراحی شدن، نه صرفاً استفاده تو یه پروژه دیگه. بهشون می‌گیم **CLI tools** یا ابزار خط فرمان. وقتی نصبشون می‌کنی، یه فایل اجرایی (exe. تو ویندوز یا یه فایل باینری تو لینوکس) تو پوشه‌ی GOBIN ساخته می‌شه و می‌تونی مستقیم از ترمینال اجراشون کنی.

مثال معروف:

```
go install golang.org/x/tools/cmd/godoc@latest
```

پکیج godoc یکی از ابزارهای رسمی Go هست که بهت اجازه می‌ده یه سرور محلی راه بندازی و داکیومنت همه پکیج‌های Go رو با فرمت HTML ببینی.

بعد از نصب تو کنسول دستور زیر رو اجرا کن:

```
godoc -http=:6060
```

و تو مرورگر برو به:

```
http://localhost:6060
```

تو این صفحه، می‌تونی مستندات مربوط به تمام توابع، پکیج‌ها و کتابخونه‌هایی که خودمون نوشتیم یا استفاده کردیم رو ببینی.

یادت میاد تو درسنامه‌ی "پکیج"، برای تابع PrintRow از پکیج table مستندات نوشتیم؟ همون کامنت‌های بالا سر تابع که توضیح می‌داد چیکار می‌کنه.

حالا با کمک این ابزار اجرایی (CLI tool) که نصب کردیم، می‌تونی همه اون توضیحات رو توی یه رابط کاربری خوشگل و مرتب ببینی، دقیقاً مثل سایت golang.org ولی برای پروژه‌ی خودت!

func **PrintRow** ¶

```
func PrintRow(values []string, width int)
```

PrintRow prints a row of values centered in columns with a fixed width.

ویژگی‌ها:

- همیشه یه فایل main.go دارن که با package main شروع می‌شه
- حتماً یه تابع main() دارن
- با go install نصب می‌شن، نه go get
- فایل اجرایی می‌سازن که می‌تونی از ترمینال صداش بزنی

پکیج های اجرایی در کجا ساخته می‌شوند؟

وقتی یه **executable package** رو با دستور go install نصب می‌کنی، خروجی اون که یه فایل اجراییه (مثلاً exe. تو ویندوز یا یه فایل بدون پسوند تو لینوکس/مک)، می‌ره توی یه مسیر خاص به اسم:

```
$GOBIN
```

اگه متغیر محیطی GOBIN رو خودت تعریف کرده باشی، فایل اجرایی دقیقا می‌ره اونجا.

ولی اگه GOBIN رو تنظیم نکرده باشی، Go به‌طور پیش‌فرض می‌فرستدش به:

```
$GOPATH/bin
```

و اگه GOPATH هم تنظیم نشده باشه (که تو نسخه های جدید Go معمولا همینطوره) مسیر پیش‌فرضش می‌شه:

```
~/go/bin
```

یعنی یه پوشه‌ی bin داخل پوشه‌ی go توی خونه‌ی کاربر.

جدول مقایسه پکیج های کتابخانه ای با اجرایی

ویژگی	پکیج کتابخانه‌ای (Library Package)	پکیج اجرایی (Executable Package)
هدف اصلی	ارائه توابع، متدها و ابزارهایی برای استفاده در پروژه‌ها	ساخت یک برنامه مستقل و قابل اجرا
شامل تابع main	ندارد	دارد (حتماً باید داشته باشد)
قابل import شدن	بله، میشه آن را در پکیج‌های دیگر import کرد	نه، هدفش اجراست، نه استفاده‌ی مجدد
قابلیت reuse	طراحی شده برای استفاده مجدد در پروژه‌های دیگر	معمولاً برای اجرای یک کار خاص درCLI
نحوه اجرا	با استفاده از آن در کد دیگر و اجرای برنامه‌ی اصلی	به صورت مستقیم با اجرای فایل یا دستور go run/go install
مثال مسیر پکیج	github.com/fatih/color	golang.org/x/tools/cmd/godoc
نحوه نصب/استفاده	go get فقط دریافت و اضافه به ماژول	go install کامپایل و نصب در $GOBIN
خروجی build	فایل a. (در کش Go برای لینک کردن به پروژه‌ها)	فایل اجرایی (binary) در مسیر $GOBIN
مکان قرارگیری پس از نصب	در کش Go Module ($GOPATH/pkg/mod/...)	در $GOBIN
مثال واقعی کاربردی	استفاده از color.Red("Hello") در یک CLI tool رنگی	اجرای godoc -http=:6060 برای دیدن مستندات پروژه

برنامه خودمون = یه پکیج اجرایی!

اگه یه نگاه به جدول بندازی، راحت می‌تونی بفهمی که برنامه‌هایی که خودمون تو Go می‌نویسیم، در واقع یه جور **پکیج اجرایی (executable package)** هستن.
چون:

- همیشه با package main شروع می‌کنیم
- یه تابع ()main داریم که نقطه‌ی شروع برنامه‌ست
- می‌تونیم با go run اجراش کنیم یا با go build یه فایل اجرایی بسازیم

یعنی این برنامه‌ی کوچیک و ساده‌ای که خودمون نوشتیم، همون ساختار و رفتاری رو داره که پکیج‌های اجرایی معروف مثل godoc دارن.

پس میتونیم ادعا کنیم: **برنامه‌ای که خودمون می‌نویسیم، یه نمونه واقعی از یه پکیج اجراییه!**

تفاوت دستور go get با go install

دستور go get برای اضافه کردن یا آپدیت کردن **پکیج‌هایی** استفاده می‌شه که قراره داخل پروژه‌ات ازشون استفاده کنی. یعنی اونا معمولاً **کتابخونه (library)** هستن، و بعد از نصب، توی go.mod ثبت می‌شن و می‌تونی توی کدت import شون کنی. مثلا:

```
go get github.com/fatih/color
```

اینجوری یه کتابخونه رنگ‌دهی به ترمینال رو به پروژه‌ات اضافه می‌کنی.

ولی دستور go install مخصوص پکیج‌هایی هست که خروجی‌شون یه برنامه اجراییه. معمولا برای نصب ابزارهایی مثل linters ، documentation generators یا ابزارهای CLI استفاده می‌شه. اینا قرار نیست داخل کدت import بشن، بلکه بعد از نصب، به صورت یه ابزار CLI مستقل در اختیارته:

اینجوری ابزار godoc رو نصب می‌کنی که برای ساخت و دیدن مستندات Go به کار می‌ره.

```
go install golang.org/x/tools/cmd/godoc@latest
```

چرا دو تا دستور جدا داریم؟

چون هدف این دو نوع پکیج فرق داره:

- go get می‌خواد یه پکیج رو وارد دنیای پروژه‌ات کنه.
- go install می‌خواد یه ابزار اجرایی بسازه که تو ترمینال باهاش کار کنی.

نام‌گذاری پکیج و آدرس ماژول

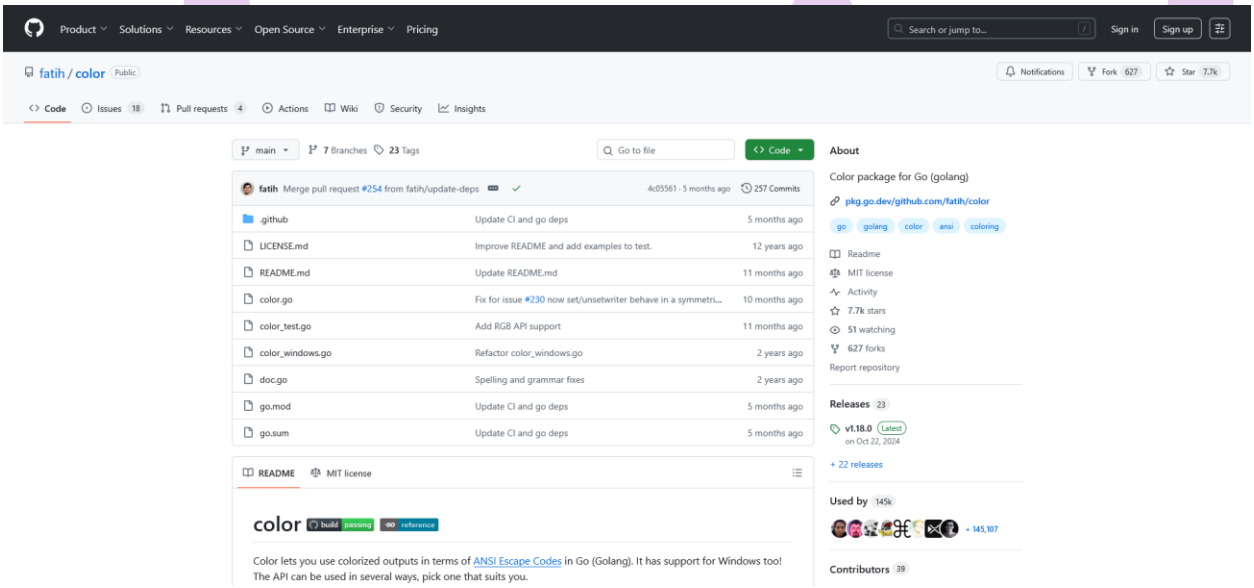
تا حالا توجه کردی وقتی می‌خوای یه پکیج رو نصب کنی، باید آدرس خاصی براش بنویسی؟ مثلاً برای نصب پکیج color از دستور زیر استفاده کردیم:

```
go get github.com/fatih/color
```

اینجا آدرس پکیج هست:

```
github.com/fatih/color
```

اگه این لینک رو تو مرورگر باز کنی، صفحه‌ای از سایت GitHub باز میشه که سورس‌کد پکیج اونجاست.



ولی واقعاً Go چجوری این پکیج رو پیدا می‌کنه؟

وقتی یه پکیج رو import یا با go get نصب می‌کنی، Go این‌طوری جلو میره:

1. **اول از همه** می‌ره تو کش لوکال داخل پوشه‌ای به اسم (pkg/mod) می‌گرده ببینه قبلاً این پکیج دانلود شده یا نه.
2. اگه اونجا نبود، می‌ره سراغ سرور رسمی Go به آدرس https://proxy.golang.org و تلاش می‌کنه از اونجا نسخه‌ی مناسب رو بگیره.
3. اگه اونجا هم پیدا نشد یا پروکسی خاموش باشه (مثلاً تو پروژه‌های خصوصی)، می‌ره مستقیماً سراغ آدرس ریپو (repo) مثل GitHub یا GitLab و از اونجا دانلودش می‌کنه.

اما نکته‌ی خیلی مهم اینجاست:

Go هیچ وقت خودش آدرس ریپو رو حدس نمی‌زنه!
بلکه فقط و فقط به اسم ماژولی که توی go.mod نوشته شده اعتماد می‌کنه.
یعنی اگه تو توی go.mod نوشتی:

```
module github.com/fatih/color
```

Go انتظار داره که دقیقاً توی همین آدرس (یعنی GitHub با مسیر fatih/color) سورس پکیج باشه.

مزایای این روش چیه؟

1. **جلوگیری از تداخل اسامی**
فرض کن یه نفر دیگه هم یه پکیج نوشته به اسم color اگه ماژول‌ها اسم خاص خودشونو نداشتن (یعنی فقط اسم ساده می‌داشتن)، موقع import کردن، Go نمی‌فهمید کدوم رو منظورت بوده!
2. **امنیت بیشتر و پیش‌بینی‌پذیری**
وقتی اسم ماژول دقیقاً همون آدرس repo باشه، Go خیالش راحته که داره پکیج درستی رو از منبع درستی می‌گیره.

پس خلاصه‌ی ماجرا:

وقتی می‌خوای یه پکیج نصب کنی، باید آدرس دقیق repo رو بدونی چون اسم ماژول توی go.mod دقیقاً باید همون باشه Go از روی این اسم می‌فهمه که باید کجا دنبال سورس بگرده، و اصلاً دنبال اسم‌های متفاوت یا مسیرهای حدسی نمی‌گرده.

چرا اسم ماژول در Go باید دقیقاً آدرس مخزن باشد؟

فرض کن می‌خوای یه پکیج در Go بسازی و به بقیه هم بدی استفاده کنن. خب، اولین چیزی که باید براش تعیین کنی، **اسم ماژوله**. اما این اسم ماژول چرا باید دقیقاً مثل آدرس مخزن (repository) واقعی باشه؟ بیاید با هم بررسی کنیم.

1-نقش اسم ماژول چیه؟

اسم ماژول تو Go مثل یه آدرس اینترنتی برای پکیج توئه.

مثلاً:

```
github.com/fatih/color
```

2- حالا فرض کنیم هر کسی هر اسمی می‌تونست بذاره

مثلاً تو می‌خوای یه ماژول بسازی و اسمش رو بذاری:

```
github.com/fatih/color
```

ولی این مال خودت نیست! یه کپی جعلی از پکیج fatih/color هست.

اگر این کار ممکن بود، چی می‌شد؟

3-مشکل اینجا کجاست؟

- وقتی یکی با go get github.com/fatih/color می‌خواست این پکیج رو بگیره،
- به جای اینکه کد اصلی fatih/color رو بگیره، نسخه جعلی و تقلبی تو رو دریافت می‌کرد!
- این باعث می‌شد که خیلی‌ها به اشتباه از کد خراب یا تغییر داده شده استفاده کنن.

4-نقش proxy.golang.org چیست؟

proxy.golang.org مثل یه کتابخونه بزرگ واسطه هست که:

- نسخه‌های اصلی پکیج‌ها رو از مخازن واقعی می‌گیره و کش می‌کنه،
- وقتی کسی درخواست میده، اون نسخه اصلی و تایید شده رو بهش می‌ده،
- و نمی‌ذاره کسی نسخه جعلی رو توش ذخیره کنه یا به بقیه بده.

5-نتیجه‌گیری

پس به همین دلیل:

- Go اجازه نمی‌ده اسم ماژول‌ها هر چی که دوست داری باشه.
- اسم ماژول **باید دقیقاً آدرس مخزن اصلی** باشه،
- تا وقتی شما یا بقیه می‌خواید اون ماژول رو نصب کنید، Go و proxy.golang.org مطمئن باشن که دارین نسخه اصلی و سالم کد رو می‌گیرید، نه نسخه تقلبی یا دستکاری شده.

نکته آخر

این قانون کمک می‌کنه که:

- امنیت پروژه‌ها حفظ بشه،
- کسی نتونه با جعل نام ماژول‌ها کد خراب یا مخرب پخش کنه،
- و همه از یک منبع معتبر و درست استفاده کنند.

نسخه بندی پکیج و ماژول

وقتی یه ماژول می‌سازیم (مثلاً یه پکیج که قراره بقیه هم ازش استفاده کنن)، ممکنه تو آینده تغییرش بدیم، باگ‌هاشو برطرف کنیم یا امکانات جدید بهش اضافه کنیم. برای اینکه Go و بقیه بدونن **کدِ کدوم نسخه رو دارن استفاده می‌کنن**، باید بهش **ورژن (version)** بدیم.

فرمت ورژن‌ها تو Go چی شکلیه؟

ورژن‌ها تو Go از قانون Semantic Versioning (SemVer) پیروی می‌کنن و به این صورت هستن:

```
v<MAJOR>.<MINOR>.<PATCH>
```

مثلاً:

- v1.2.3

حالا بیایم تیکه‌تیکه بگیم هر قسمت چی میگه:

MAJOR-1 مثلاً عدد 1 در (v1.2.3)

عدد **اصلی** هست. اگه تغییرات بزرگی دادی که پکیج دیگه با برنامه هایی که با نسخه‌های قبلی نوشته شده ناسازگاره (breaking changes) تو این حالت این عدد باید زیاد بشه.

مثال فرضی از **fatih/color**:

نسخه v1.16.0 تابعی داره به نام color.New() ،فرض کن توسعه‌دهنده در آینده در نسخه v2.0.0 تصمیم میگیره اسم این تابع رو به Make تغییر بده. در این صورت از اونجایی که کدهایی که قبلاً color.New() را استفاده می‌کردند دیگر کار نمی‌کنند، ورژن باید از v1.16.0 به v2.0.0 افزایش پیدا کند و در توضیحات گفته بشه که تابع New حذف یا تغییر نام داده شده.

به طور کلی آپگرید پکیج از نسخه (x).0.0 به (x+1).0.0 باید با دقت بالایی انجام بشه چون ممکنه تغییراتی اتفاق افتاده باشه که باعث بشه کد هایی که با نسخه قبل از پکیج نوشته شدن دیگه کار نکنن.

فرمت تگ:

```
git tag v2.0.0
```

تغییرات در go.mod (اگر v2 یا بالاتر):

```
module github.com/fatih/color/v2
```

MINOR-2 (مثلاً عدد 2 در v1.2.3)

وقتی **قابلیت جدیدی** به ماژولت اضافه کردی ولی با نسخه قبلی هنوز سازگاره (یعنی کدهای قبلی هنوز کار می‌کنن)، این عدد زیاد میشه.

مثال فرضی از **fatih/color**:

نسخه v1.3.0 ویژگی جدیدی اضافه کرده به نام color.RGB(r, g, b string) که به‌جای رنگ‌های پیش‌فرض، اجازه تعریف رنگ سفارشی RGB می‌دهد. این اضافه شدن جدید است ولی توابع قبلی همچنان قابل استفاده‌اند.

فرمت تگ:

```
git tag v1.3.0
```

PATCH-3 (مثلاً عدد 3 در v1.2.3)

اگه **باگ کوچیک** یا اصلاحات جزئی کردی (بدون تغییر عملکرد کلی)، این عدد زیاد میشه.

مثال واقعی از faith/color:

توسعه‌دهنده در نسخه 1.16.0 یک باگ کوچیک در رفتار زیرخط (underline) و رنگ پیش‌زمینه (fg) مشاهده کرد و اونو برطرف کرد، بنابراین برای رفع این مشکل نسخه 1.16.1 رو منتشر کرد — بدون تغییر API برای کاربران.

ورژن‌های پیش‌نشر (پیش‌نمایش)

ممکنه موقع نصب پکیج های مختلف ورژن‌هایی مثل این ببینی:

- v1.5.0-beta.1
- v2.0.0-rc.2

این‌ها نسخه‌های آزمایشی هستن. پسوندهایی مثل -beta, -alpha یا -rc یعنی:

- beta: نسخه آزمایشی برای تست عمومی
- rc: مخفف واژه "Release Candidate" و یعنی نزدیک نسخه نهایی

حالا چجوری ورژن ماژول خودمون رو مشخص کنیم؟

مرحله 1: تگ زدن در Git

ورژن ماژول‌ها با **تگ Git** مشخص میشه.

فرض کن کد ماژولت تو یه ریپو گیت هست، مثل:

```
github.com/john/utils
```

وقتی می‌خوای یه نسخه رسمی ازش منتشر کنی، باید بری تو گیت و یه تگ بزنی:

```
git tag v1.0.0
git push origin v1.0.0
```

دقت کن که تگ **حتماً** باید با v شروع بشه، مثل v1.0.0، تا go module اونو بشناسه.

مرحله 2: تغییر فایل go.mod (در صورت نیاز)

اگر قراره **ورژن 2 یا بالاتر** بدی (v2.0.0, v3.0.0, ...) باید داخل فایل go.mod هم اسم ماژول رو آپدیت کنی:

```
module github.com/radin/mycoolpkg/v2
```

Go از نسخه 2 به بعد توقع داره مسیر import هم نسخه داشته باشه! یعنی کسی که استفاده می‌کنه باید بنویسه:

```
import "github.com/radin/mycoolpkg/v2/utils"
```

ولی تا v1.x.x نیازی به این کار نیست.

چک کردن ورژن‌های منتشرشده

با این دستور می‌تونی لیست نسخه‌های منتشر شده یه ماژول رو ببینی:

```
go list -m -versions github.com/fatih/color
```

نکته امنیتی: تگ اشتباه نزن!

اگر یه تگ اشتباهی زدی و پوش کردی، دیگه خیلی راحت نمی‌تونی پاکش کنی چون Go proxy ممکنه اونو کش کنه. پس قبل از git tag زدن حتماً مطمئن شو که کدت آماده‌ی انتشار هست.

منتشر کردن ماژول جدید

حالا که با مفاهیم اصلی مثل ساخت ماژول، ورژن‌گذاری، فایل go.mod و نحوه‌ی استفاده‌ی بقیه از پکیج‌ها آشنا شدی...

می‌تونی با خیال راحت دست به کار بشی و **ماژول خودت رو منتشر کنی**!

فرقی نمی‌کنه پکیجت یه کتابخونه باشه یا یه ابزار مستقل، الان دیگه اون‌قدر مسلط شدی که بدون نگرانی بتونی یه پروژه‌ی استاندارد راه بندازی و اون رو به دست بقیه برسونی.

فقط کافیه یه آشنایی کوچیک با ابزار Git پیدا کنی (که یاد گرفتنش هم خیلی سخت نیست) و بعدش همه‌چی آماده‌ست برای رفتن به مرحله‌ی انتشار!

دستور go mod init _ چیزی که تا الان جا مونده بود!

یه دستور مهم بود که تا الان تو این درسنامه بهش اشاره نکردیم و عمداً بهش نپرداختیم:

```
go mod init <module-name>
```

چرا تا الان نگفتیم؟

چون خیلی از ما وقتی با IDE هایی مثل **GoLand** یا حتی VS Code کار می‌کنیم،
وقتی یه پروژه جدید می‌سازیم، **IDE خودش خودکار فایل go.mod رو برامون می‌سازه**.
اسم ماژول هم معمولاً دقیقاً مثل اسم پروژه‌ایه که ساختیم.

پس شاید حتی بدون اینکه بفهمیم go mod init از پشت صحنه اجرا شده!

اما این دستور چیکار می‌کنه؟

- یه فایل go.mod تو ریشه پروژه ایجاد می‌کنه.
- اسم ماژول رو می‌ذاره همونی که شما تو <module-name> مشخص کردی.
- از این به بعد، کل ماژول و وابستگی‌هاش رو **بر اساس این اسم و مسیر** مدیریت می‌کنه.

مثلاً:

```
go mod init github.com/john/utils
```

این باعث میشه که تو کدهای دیگه بتونی ماژولت رو اینطوری import کنی:

```
import "github.com/john/utils"
```

حتی اگه IDE برات خودکار go.mod رو ساخته، خوبه که بدونی **پشت پرده چه اتفاقی افتاده** و
اگه یه روز خودت خواستی پروژه رو دستی راه بندازی، بلد باشی این دستور رو بزنی.
این دستور درواقع اولین قدم رسمی برای تبدیل یه پوشه خالی به یه **ماژول واقعی Go** هست.

تمرین: بررسی مفاهیم در قالب استفاده از پکیج tablewriter

تو این درسنامه یاد گرفتیم ماژول چیه، چطور یه پکیج رو نصب می‌کنیم go.mod و go.sum چه‌کار می‌کنن، و با دستورهایی مثل go get و go mod tidy چطور می‌تونیم پروژه‌مون رو مدیریت کنیم.

حالا وقتشه این مفاهیمو تو یه سناریوی واقعی تمرین کنیم.

قراره با یه پکیج پرکاربرد به اسم tablewriter کار کنی و جدول‌های خوشگل توی ترمینال رسم کنی. هم تجربه کار با نسخه‌های مختلف پکیج رو پیدا می‌کنی، هم دستت تو نصب و پاک کردن پکیج‌ها راه می‌افته، هم یه پروژه واقعی‌تر می‌سازی.

1-با دستور زیر تمام نسخه‌های این پکیج رو ببین:

```
go list -m -versions github.com/olekukonko/tablewriter
```

2-حالا نسخه‌ی v0.0.5 رو نصب کن:

```
go get github.com/olekukonko/tablewriter@v0.0.5
```

3-فایل‌های go.mod و go.sum رو باز کن و ببین چه تغییری کردن

سوال: چرا کنار پکیج tablewriter نوشته شده indirect ؟ یعنی چی؟

4-با دستور زیر پکیج رو به آخرین نسخه آپگرید کن:

```
go get -u github.com/olekukonko/tablewriter
```

5-حالا دوباره دانگرید کن به 0.0.5

```
go get github.com/olekukonko/tablewriter@v0.0.5
```

6-دستور go mod tidy رو اجرا کن و دوباره تغییرات go.mod و go.sum رو ببین و تحلیل کن.

7-وارد پروژه‌ی phonebook شو و فایل table/draw.go رو باز کن. این خط رو به ابتدای فایل اضافه کن:

```
import "github.com/olekukonko/tablewriter"
```

8-با کمک مستندات پکیج به آدرس github.com/olekukonko/tablewriter سعی کن ازش برای رسم یه جدول شیک استفاده کنی، اطلاعات دفترچه تلفن رو تو جدول نشون بده.

9-حالا go.mod رو بررسی کن. نباید پکیج tablewriter توش ببینی، دستور زیر رو بزن:

```
go mod tidy
```

10-دوباره بررسی کن go.mod و go.sum چه تغییری کردن

11-یه بار دیگه پکیج رو دانگرید کن به v0.0.5 و برنامه رو اجرا کن، آیا هنوز درست اجرا میشه؟ اگه نه، دلیلش چیه؟

12-پکیج tablewriter به نظرت **کتابخانه‌ایه یا اجرایی**؟ چرا؟