# کتاب فارسی گولنگ به طور خلاصه

آشنایی با ساختار گولنگ در کمترین زمان ممکن


**نویسندگان**

*سید امیر ایرانی*
Junior Full Stack Developer
Mobile, Desktop Developer

*مهندس مهدی موسوی*
Junior Full Stack Developer

**آدرس اینترنتی کتاب برای دانلود کدها**

*https://github.com/Gommunity/gosuccinctly*

**محفل گفتگو برنامه نویسان گولنگ**

*https://goforask.com*

**« این کتاب را به پدر و مادرم یعنی بزرگترین و پیچیده ترین واژه هستی که هنوز معنی روشنی و واضحی از این ۲ معجزه زندگی ارائه نشده و هنوز مشخص نیست چرا تمام حیات یک فرزند مانند یک بند ناف به تعداد نفس هایی که آن ها میکشند وابسته است ... تقدیم می کنم .»**

**امیر ایرانی**

## مباحث کتاب

## به گولنگ خوش آمدید !

نمی دونم برنامه نویس باتجربه ای هستید یا نه ! اما مطمنم قرار هست عاشق گولنگ بشید .
در سال ۲۰۰۷ گولنگ در گوگل متولد شد ، زمانی که سه مهندس باتجربه با نام های Robert Griesemer, Rob Pike, and Ken Thompson با هم تصمیم بر ساخت یک زبان برنامه نویسی سیستمی گرفتند . گوگل یک کمپانی خاص هست در نتیجه مشکلاتی هم که برایش پیش میاد نیز خاص هست . با تعداد بی شماری کامپیوتر که بر روی یک شبکه قرار گرفتند به کمک پلتفورم های مختلفی که با زبان های گوناگونی نوشته و توسعه داده شده اند هیچ Taskای بر روی زبان های برنامه نویسی سیستمی وجود ندارد که برای گوگل بی اهمیت باشد .
منطقی که پشت توسعه زبان گولنگ وجود داشت این بود که آن ها دنبال زبان برنامه نویسی بودند که بتوان به کمک آن نرم افزارهای قدرتمند و مقیاس پذیری نوشت و در عین حال خیلی از پیچیدگی های زبان های برنامه نویسی سنتی را هم کنار گذاشت . خیلی از برنامه نویسا به خصوص خودم واقعا از نوشتن برنامه روی گولنگ لذت می بریم چون گولنگ به هیچ وجه دنبال ساخت یک الگو جدید برای برنامه نویسی نبود بلکه هدف اصلی آن ساخت زبانی جدید برای بر طرف کردن چالش هایی است که امروزه برنامه نویسان با آن رو به رو می شوند .
خوب حالا سوال اینجاست که چه چالش هایی وجود داشته که مهندسان گولنگ را خلق کردند !؟
(Ken Thompson خالق سیستم عامل Unix است)

## گولنگ کامپایل شونده است

برای اجرای برنامه‌های نوشته شده در گولنگ ، باید آن‌ها را کامپایل نمایید. خروجی عملیات کامپایل ، کد ماشین است. بدون نیاز به VM ، بدون نیاز به JIT و بدون نیاز به تفسیر . همانند C حاصل برنامه شما مستقیما کدهای ماشین خواهد بود. در حال حاضر کامپایلر گولنگ که خودش در زبان C نوشته شده ، قادر است برای پلتفرم‌های x64 ، x86 و arm کد ماشین تولید کند. سرعت کامپایل شدن برنامه‌ها در گولنگ بسیار بسیار بالاست و در این زمینه جای هیچ رقابتی را برای دیگر زبان‌های کامپایلری مانند C++، C# ، C ، Java و ... باقی نگذاشته است .

خود کامپایلر نیز به صورت رسمی برای سیستم عامل‌های FreeBSD ، Mac ، Linux و Windows منتشر می‌شود . اما به شکل غیر رسمی ، کاربران گزارش کرده اند که روی سیستم‌های دیگری مثل Plane9 ، Android و بقیه BSD ها هم موفق به اجرای آن شده اند.

در گولنگ نیاز به چیزهایی شبیه makefile ها و یا برنامه‌های مدیریت پروژه مثل Maven نیست . کامپایلر گولنگ از فایلی که تابع main در آن قرار دارد شروع کرده و خودش بقیه کدهای مورد نیاز را شناسایی و لینک می‌نماید ! حتی اگر یک برنامه چند صد هزار خطی با تعداد زیادی سورس فایل داشته باشید ، فقط یک خط دستور ساده در ترمینال کافیست تا کل برنامه شما کامپایل شود.

لازم به ذکر است که کامپایلر گولنگ کدهای شما را به شکل Static لینک می‌نماید . به این معنی که حاصل کامپایل برنامه شما در نهایت یک فایل اجرایی یک تکه خواهد بود؛ مهم نیست که برنامه شما از تعداد زیادی فایل و یا Package های جانبی تشکیل شده ، چیزی که در نهایت به شما تحویل داده خواهد شد یک فایل اجرایی ساده است که برای اجرا شدن هیچ پیش نیازی لازم ندارد و به راحتی قابل انتقال به دستگاه‌های دیگر است .

کامپایلر گولنگ با کسی شوخی ندارد ! چیزهایی که در زبان‌های دیگر باعث Warning می‌شوند ، همگی در گولنگ به عنوان Error در نظر گرفته شده اند . مثلا اگر یک متغیر تعریف کرده باشید اما از آن استفاده نکرده باشید ، کامپایلر به جای یک Warning سطحی با یک پیغام Error کل عملیات کامپایل را متوقف می‌کند ! در گولنگ چنین اشتباهاتی پذیرفتنی نیست !

## گولنگ و سینتکس آشنا

گولنگ از خانواده C است و به همین دلیل برنامه نویسانی که با JavaScript ، PHP ، C# ، Java ، C++ ، C و ... آشنایی دارند ، بسیار راحت سینتکس این زبان را یاد خواهند گرفت . برای مثال با کمی دقت در کد زیر ، براحتی متوجه منظور آن خواهید شد :

```
package main

import "fmt"

func main(){

     a := [] int{11, 22, 33, 44, 55}
     s := 0

     for i := range a {
          s += a[i]
     }

     fmt.Printf("%d \n", s)
}
```

سینتکس یک زبان تاثیر زیادی در پیشرفت آن دارد . وقتی سینتکس زبان آشنا باشد برنامه نویسان راحت تر آن را یاد خواهند گرفت و به همین نسبت محبوبیت زبان بالاتر خواهد رفت . با بلا رفتن محبوبیت ، سازندگان زبان با اشتیاق و سرعت بیشتری به کار توسعه مشغول می‌شوند . در این حالت ضریب اعتماد برنامه نویسان به زبان بالاتر می‌رود و کتابخانه‌ها و ابزارهای بیشتری برای زبان تولید می‌شود . برنامه نویسان بیشتر از اینکه کد نویسی کنند به کد خوانی مشغول اند. تا زمانی که سینتکس یک زبان توسط برنامه نویسان پذیرفته نشود نمی‌تواند محبوبیت چندایی پیدا کند . برای مثال زبان‌های Functional با آن همه قابلیت‌های منحصر به فردی که دارند ، در اکثر اوقات فقط به دلیل داشتن سینتکس نا آشنا در رابطه با جذب برنامه نویسان با مشکل مواجه می‌شوند.

سینتکس گولنگ را می‌توان ترکیبی از سینتکس زبان‌های Python ، C و Pascal به حساب آورد . سعی شده تا سینتکس زبان کوچک ، تمیز و قابل فهم باشد . برای مثال ساختار حلقه در گولنگ فقط با for پیاده سازی شده و حلقه‌های while یا do از آن حذف شده اند . نکته جالب دیگری در مورد سینتکس گولنگ این است که گرامر آن Regular است ! و به همین دلیل پردازش کدهای گولنگ برای ابزارهای جانبی مثل IDE ها بسیار آسان خواهد بود . همچنین Coding پیش فرض تمام فایل‌ها UTF-8 است ! (Ken Thompson و Rob Pike خودشان خالق UTF-8 هستند)

## گولنگ یک زبان رویه‌ای است (Procedural)

سوالی که برای تعداد زیادی از برنامه نویسان مشتاق پیش می‌آید این است که آیا گولنگ شیءگراست؟ جواب این است که خوشبختانه خیر ! حداقل نه به شکلی که در زبان‌های شیءگرا با آن آشنا هستید.
باید توجه داشته باشید که شیءگرایی یک مفهوم است نه یک قابلیت . اینطور نیست که در زبانی باشد و در زبان دیگری وجود نداشته باشد . برای مثال در زبانی مثل C که شیءگرا نیست هم می‌توان از مفاهیم شیءگرایی استفاده کرد . حتی در یک زبان Functional مانند Lisp هم می‌توان از شیءگرایی بهره برد . البته مسلم است که اگر یک زبان دارای گرامر خاصی برای این منظور باشد پیاده سازی کدهای شیءگرا در آن آسان تر خواهد بود. مانند همان چیزی که در PHP ، Java و ... موجود است.

اولین نکته‌ای که باید درک شود این است که شیءگرا بودن یک زبان ، به هیچ عنوان تضمینی بر کیفیت ساخت آن زبان نیست ! حتی تضمینی بر کیفیت کدهایی که در آن زبان نوشته می‌شوند هم نیست . دومین نکته این است که شیءگرایی ، برعکس چیزی که در کتاب‌ها درباره اش می‌خوانید و تبلیغاتی که حول و حوش آن می‌شود ، دارای مخالفان زیادی است!

صحبت در مورد نظرات مثبت و منفی نسبت به شیءگرایی در این نوشته نمی‌گنجد . شما می‌توانید خودتان در این باره تحقیق کنید .

## ساختار رویه‌ای گولنگ در برابر شیءگرایی

طراحان گولنگ بر این باورند که مدل شیءگرایی در زبان‌هایی مثل C# ، Java و C++ پیچیدگی‌های زیادی دارد و این پیچیدگی در زبان ، باعث تولید کدهای پیچیده خواهد شد. گولنگ رویه‌ای است (Procedural) ، اما نه یک زبان رویه‌ای کلاسیک مانند C.
طراحان گولنگ نوآوری‌های جالبی در ساختار کلاسیک زبان‌های رویه‌ای ایجاد کرده اند تا گولنگ را به یک زبان رویه‌ای مدرن تبدیل کنند !
برنامه نویسان با کمک این ساختار رویه‌ای مدرن ، نیاز چندانی به آن شیءگرایی مرسوم در زبان‌های دیگر حس نخواهند کرد . در ادامه با تعدادی از ایده‌های جدید گولنگ در این زمینه آشنا می‌شوید .

## ابتدا یک توضیح ساده درباره کلاس‌ها و اشیا

در بیشتر زبان‌های برنامه نویسی روشی وجود دارد که برنامه نویس به کمک آن می‌تواند یک Data Type جدید ایجاد کند . Data Type که به اختصار Type خوانده می‌شود ، الگویی است که تعیین می‌کند یک داده چه ساختاری در حافظه خواهد داشت و چه عملیاتی می‌تواند روی آن انجام داد .
زبان‌های مختلف ، روش‌های مختلفی برای ساخت یک Type ارایه کرده اند . مثلا در زبان C از Struct برای این منظور استفاده می‌شود . (با همراهی typedef) در اکثر زبان‌های شیءگرا هم ساختاری وجود دارد به نام Class که به برنامه نویس امکان ساخت یک Type جدید را می‌دهد . در زبان‌های شیءگرا ، یک Type می‌تواند از تعدادی فیلد و متد تشکیل شود که به ترتیب تعیین کننده خصوصیات و رفتار آن Type هستند . برای استفاده از یک Type ، باید یک نمونه از آن Type را در حافظه ایجاد کنید . در زبان‌های شیءگرا این نمونه‌ها را اصطلاحا Object یا شی می‌نامند .

## گولنگ به جای Class از Struct استفاده می‌کند

گولنگ هم مثل C از Structها برای ساخت یک Type جدید استفاده می‌کند . با این تفاوت که Structهای گولنگ نسبت به C پیشرفته ترند . یک Struct در گولنگ می‌تواند علاوه بر داشتن فیلد ، دارای متد هم باشد . متدها در گولنگ همان توابع معمولی هستند . حتی داخل structها هم نوشته نمی‌شوند . فقط لازم است با یک تغییر کوچک به هنگام تعریفشان ، آن‌ها را به structها نسبت دهیم .

در واقع با وجود داشتن چنین Structهایی در زبان گولنگ ، شما نیازی به داشتن چیزی مثل Class ندارید! همان کارهایی که با Classها قابل انجام است ، با این Structهای جدید خیلی راحت تر و سبک تر قابل انجام خواهد بود.

زبان برنامه نیسی Rust هم که در حال توسعه از طرف Mozilla است ، با اینکه در نسخه‌های اولیه خود دارای ساختار Class بود ، اما در نسخه ۰.۴ ساختار Class را از زبان حذف کرد و آن را با Structهایی مشابه چیزی که در گولنگ وجود دارد جایگزین نمود .

## گولنگ از Composition به جای وراثت استفاده می‌کند

Java سعی کرد با حذف قابلیت وراثت چندگانه که در C++ وجود داشت ، باعث ساده شدن مکانیزم وراثت در زبان شود . گولنگ یک قدم جلوتر رفت و کلا با حذف وراثت ، Compositon را به جای آن جایگزین کرد .

Composition چیست؟ فرض کنید دو Struct به نام‌های A و B تعریف کرده ایم . B می‌تواند A را مانند یک فیلد معمولی در Struct خود قرار دهد تا به اعضای موجود در A دسترسی داشته باشد . همانند Structهای تو در تو در زبان C . به این ترتیب بدون درگیر شدن با پیچیدگی‌های مبحث وراثت ، می‌توانیم مکانیزمی شبیه آن را در کدهایمان داشته باشیم.
حتما می‌دانیم که خیلی از زبان‌های شیءگرا از یک سیستم سلسله مرتبه‌ای برای کار با اشیا بهره می‌برند . مثلا در بیشتر آن ها، یک

شی Object وجود دارد و بقیه اشیاء همگی به طریقی از آن ارث می‌برند . در گولنگ چنین چیزی وجود ندارد ، هر Type برای خودش مستقل است . نیازی نیست کامپایلر در هر عمل کامپایل رابطه وراثت بین Typeها را چک کند . یکی از دلایل اصلی سرعت کامپایلر گولنگ نیز همین مساله است . اطمینان داشته باشید که خودتان هم با استفاده از قابلیت ترکیب سازی در گولنگ ، متوجه مزیت آن نسبت به وراثت خواهید شد .

حتی در کتاب Design Patterns: Elements of Reusable Object-Oriented Software که یکی از معروف ترین کتب مرجع در زمینه شیءگرایی می‌باشد ، عنوان شده است که :
ترکیب سازی اشیاء را به وراثت ترجیح دهید .
Favor Object Composition over class inheritance

## گولنگ می‌تواند برای اعضا حق دسترسی تعیین کند

اگر نام یک عضو با حرف کوچک شروع شود (مانند hello) ، آن عضو فقط برای اعضای داخل Package در دسترس است و اگر نام یک عضو با حرف بزرگ شروع شود (مانند Hello) ، آن عضو می‌تواند در محیط خارج از Package نیز در دسترس قرار گیرد ، در این حالت فقط با یک نگاه به نام آن عضو ، می‌توان به سطح دسترسی آن پی برد .
دقت کنید که این روش فقط یک "استایل نام گذاری" نیست . این یک "قانون" است ، به این معنی که کامپایلر واقعا در زمان کامپایل این سطوح دسترسی را چک می‌کند .

## گولنگ دارای ساختار interface است

interface به عنوان یکی از بهترین قابلیت‌های معرفی شده توسط زبان‌های شیءگرا در گولنگ حضور دارد . برنامه نویسان Java و C# با interfaceها کاملا آشنایی دارند .
به زبان ساده ، یک interface مشابه یک (سند قرارداد) است . تمام Typeهایی که به یک interface وابسته هستند موظف اند از قراردادهایی که توسط آن interface تعریف شده تبعیت کنند . بدین صورت آن interface می‌تواند در موقعیت‌های مختلف ، به وکالت از تمام Typeهای وابسته به آن مورد استفاده قرار بگیرد (به جای این که تک تک آن Typeها را جداگانه احضار کنید) .
در گولنگ قرارداد بین یک interface و Typeهای وابسته به آن ، فقط شامل تعاریف متدها می‌شود .
Interfaceها نگرش اصلی زبان گولنگ به مبحث Polymorphism می‌باشند؛ آن‌ها به عنوان یکی از قابلیت‌های مهم زبان تلقی شده و به هنگام ساخت APIها ، بسیار مورد استفاده قرار می‌گیرند . مطالعه و یادگیری آن‌ها برای افراد علاقه مند به گولنگ توصیه می‌شود .
درست است که گولنگ چیزی تحت عنوان Class ندارد ، اما اجباری نیست که برای نوشتن کدهای شیءگرا حتما از Classها استفاده کنید . اجباری هم نیست که حتما برنامه‌های خود را به صورت شیءگرا طراحی کنید . این توهم‌ای است که امثال زبان‌هایی مثل C# ، Java و C++ به شما تلقین کرده اند.
گولنگ تمام قابلیت‌های لازم برای برنامه نویسی شیءگرا را در اختیار شما قرار داده است . حتی می‌توان گفت که شیءگرایی در گولنگ به نسبت خیلی از زبان‌های دیگر ساده تر و راحت تر است . مساله این است که دیدگاه گولنگ نسبت به ساخت برنامه ها ، با دیدگاهی که زبان‌هایی مثل Java یا C# دارند متفاوت است . هدف آن‌ها یکی است، اما روش کارشان با یکدیگر فرق دارد . برنامه نویسی در گولنگ بر مبنای Typeها و توابع صورت می‌گیرد ، نه Classها و متدها .

سازندگان گولنگ فکر نمی‌کنند که برای نوشتن برنامه‌های ساخت یافته در ابعاد وسیع ، حتما باید به شیءگرایی متوسل شد؛ شاید راه‌های ساده تر و مناسب تری هم وجود داشته باشد . این دیدگاه شبیه همان نگرشی است که زبان‌های Functional به عنوان قطب مخالف زبان‌های شیءگرا مدت هاست که آن را عنوان می‌کنند .

## گولنگ Static-Type است

**زبان‌های Static نسبت به زبان‌های Dynamic از سه مزیت عمده برخوردارند :**

***سرعت :*** چون در زبان‌های Static نوع تمام داده‌ها از قبل مشخص می‌شود ، سرعت اجرای برنامه به مراتب بالاتر از زبان‌های Dynamic خواهد بود . در زبان‌های Dynamic نوع داده‌ها به هنگام اجرا مشخص خواهد شد .
***امنیت :*** در زبان‌های Static کامپایلر قادر است تمام داده‌ها و پارامترها را چک کند تا اگر برنامه نویس به صورت سهوی متغیری را در جای اشتباهی به کار برده بود ، قبل از کامپایل برنامه به او هشدار داده شود.
***مستندات :*** مستندسازی کدها در زبان‌های Dynamic نیاز به دقت بالایی دارد. برای مثال باید نوع پارامترهای یک تابع را در مستندات ذکر کنیم تا برنامه نویسان دیگر بدانند که قرار است چه نوع داده‌ای را به تابع ارسال کنند. اما در زبان‌های Static نوع هر پارامتر جزیی از خود کد است و برنامه نویس با یک نگاه ساده به نحوه تعریف تابع می‌تواند اطلاعات زیادی درباره آن بدست آورد. جدا از مزایایی که زبان‌های Static ارائه می‌کنند ، یک عیب بزرگ نیز دارند : اینکه Static هستند. درست است، Static بودن یک زبان شبیه چاقوی دولبه است . مزیت اصلی آن، همان عیب آن است.

در این زبان‌ها باید مدام با Typeها سرو کله بزنید. برنامه نویسان زبان‌های Dynamic به خوبی می‌دانند که Dynamic بودن زبان دلخواهشان ، تا چه میزانی در سرعت کدنویسی شان تاثیر دارد.
خوشبختانه گولنگ می‌تواند Type یک متغیر را از روی مقداری که به آن نسبت می‌دهیم تشخیص دهد. مثلا اگر عدد ۱۲ را در متغیر A بریزیم، گولنگ متغیر A را از نوع int فرض خواهد کرد. این قابلیت شبیه سیستم Type Inference در زبان Haskell است. وقتی چنین سیستم تشخیص Typeای را با مدل ساده و سریع کامپایل برنامه‌ها ادغام کنید، متوجه می‌شوید که سرعت کدنویسی شما قابل رقابت با سرعت کدنویسی در زبان‌های Dynamic خواهد بود.

***جناب آقای Joe Armstrong خالق زبان برنامه نویسی Erlang و پلتفرم OTP در مورد گولنگ می‌گوید :***
من فکر می‌کنم این زبان به سنت Unix و C برگشته و کمبودهای زبان C را جبران کرده است. من فکر نمی‌کنم که C++ یک پیشرفت در این زمینه بوده باشد. اما معتقدم که گولنگ قطعا یک پیشرفت برای زبان C به حساب می‌آید و از طرفی هم این افراد در گذشته با آدم‌هایی مثل Kernighan و امثال اون کار می‌کردن و اطمینان دارم که تجربه بسیار بالایی در ساخت زبان‌ها برنامه نویسی دارند . این زبان خیلی ظریفانه مهندسی شده و از اول خیلی از ابزارهایی که احتیاج دارید در اون وجود داره . حتی اولین نسخه‌ای هم که از این زبان منتشر شد در سطحی از تکامل قرار داشت که انگار سال‌ها در حال توسعه بوده و در کل نظر من در مورد این زبان بسیار مثبت است.

## گولنگ و Concurrency :

قبل از هر چیز ، نیاز داریم این مفاهیم را تعریف کنیم . متاسفانه خیلی از سایت‌ها و منابع ، تعریف درست و قابل فهمی از Concurrency و Parallelism ارایه نکرده اند و درک این مفاهیم را برای خیلی از برنامه نوسان مشکل ساخت اند .
در ادامه با تعریف Rob Pike از این مفاهیم آشنا می‌شویم :
***Concurrency :*** برنامه نویسی بر مبنای مجموعه از واحدهای اجرایی مستقل ، که هدف مشترکی دارند.
***Parallelism :*** توانایی اجرای چندین پردازش به صورت موازی با هدف دستیابی به سرعتی بالتر .

Concurrency مدلی برای ساخت یک برنامه است ، اما Parallelism مدلی برای اجرای برنامه هاست .
Concurrency در فاز ساخت برنامه اعمال می‌شود ، اما Parallelism در زمان اجرای برنامه اتفاق می‌افتد .
در Concurrency واحدهای اجرای مستقل از یکدیگرند ، ولی هدف مشترکی دارند .
در Parallelism پردازش‌ها ممکن است هیچ ربطی به هم نداشته باشند . (مثل پردازش دو برنامه جداگانه)

Concurrency اجرای Parallel واحدهای اجرایی را تضمین نمی‌کند ! ممکن است برنامه شما Concurrent باشد اما اجزایش به صورت Parallel اجرا نشود (مثل Concurrency در Python) . مهم این است که ساختار برنامه به صورت Concurrent نوشته شده باشد.
Concurrency ساختاری را برای برنامه محیا می‌کند که در صورت وجود بستر سخت افزاری و نرم افزاری مناسب ، اجزای مختلف برنامه بتوانند به شکل Parallel پردازش شوند.
به عبارت دیگر، اگر برنامه‌ای به صورت Concurrent ساخته نشود، به صورت Parallel اجرا نخواهد شد ! (البته Load کردن چندین

نمونه از یک برنامه در حافظه برای انجام پردازش Parallel مبحث جداگانه ایست که ربطی به بحث فعلی ندارد)
هر برنامه نویس هم ممکن است به شیوه متفاوتی Concurrency را در ساختار برنامه اش اعمال کند . قانون ثابتی برای طراحی برنامه‌های Concurrent وجود ندارد .
لازم به ذکر است که برای دستیابی به پردازش Parallel ، حتما باید بیش از یک هسته CPU در دسترس باشد تا پردزش Parallel به شکل واقعی اتفاق بیفتد . در سال‌های اخیر با گسترش استفاده از پردازنده‌های چند هسته ای ، سیستم‌های توزیع شده و نیاز وب سرویس‌ها برای پاسخ گو بودن به تعداد بالایی از درخواست ها وجود بستری مناسب برای برنامه نویسی همروند و یا Concurrent کاملا قابل احساس است .
در این بین ، زبان‌های تابع گرا و یا Functional که به طور ذاتی برنامه نویسی Concurrent را به شما تلقین می‌کنند در این زمینه پیش افتاده اند و هر روز بر محبویت شان افزوده می‌شود . زبان‌هایی مثل Clojure ، Haskell ، Erlang و ... .
زبان‌های دستوری و یا Imperative هم هر کدام در این راستا تلاش‌هایی کرده اند ، اما اکثر آن‌ها هنوز هم به طور مستقیم از Thread و Processها استفاده می‌کنند .
درست است که مبنای کار همه پردازش‌ها در نهایت بر پایه Threadها و Processها خواهد بود ، اما این‌ها جزو مفاهیم سطح پایین یک سیستم عامل محسوب می‌شوند و استفاده مستقیم از آن‌ها برای پیاده سازی Concurrency بسیار بسیار دشوار است و در عمل بهینگی لازم را ندارد .

به نظر می‌رسد استفاده از تکنیک عملیات نا همگام و یا Asynchronous تا حدودی به روند ساخت برنامه‌های Concurrent در این زبان‌ها کمک کرده و بهینگی لازم را برای آنان فراهم نموده است.
کتابخانه‌هایی مثل gevent در Python یا محیط‌های همچون node.js برای JavaScript از نمونه‌های موفق در بکارگیری تکنیک عملیات Asynchronous می‌باشند . با اینکه چنین فریم ورک‌هایی از استقبال خوبی برخوردار شده اند، اما راه حلی برای برنامه نویسی Concurrent به حساب نمی‌آیند .
حقیقت این است که عملیات Asynchronous برای گونه خاصی از برنامه‌ها که رخدادهای I/O در مقیاس بالا در آن‌ها اتفاق می‌افتند بسیار خوب عمل می‌کنند (مثل وب سرورها) اما وقتی صحبت از پردازش‌ای سنگین می‌شود ، مدل Asynchronous راه حل مناسبی ارایه نمی‌کند .
در مدل Asynchronous اگر قسمتی از برنامه نیاز به پردازش طولانی مدت داشته باشد ، بقیه اجرا باید منتظر بمانند تا کار آن قسمت تمام شود؛ چرا که تمام اجزای برنامه فقط در یک Thread پردازش می‌شود . از همین رو پردازش‌ها باید بسیار کوچک و گذرا تعریف شوند .

***با توضیحات بالا به نظر می‌رسد یک پیاده سازی مناسب از قابلیت Concurrency باید دارای خصوصیات زیر باشد :***

- پیاده سازی Concurrency باید ساده و آسان باشد .
- پیاده سازی Concurrency باید بهینه و سبک باشد .
- پیاده سازی Concurrency باید تا جایی که ممکن است همه منظوره باشد .

خوشبختانه یکی از دلایل اصلی ساخت زبان گولنگ پشتیبانی قدرتمند از برنامه نویسی Concurrent بوده است . این زبان نه به صورت یک کتابخانه و نه به صورت یک قابلیت جانبی ، بلکه به صورت درونی از Concurrency پشتیبانی می‌کند . حتی دارای یک سینتکس مخصوص برای این کار است .
گولنگ چنین بستری را مدیون تجربه سی ساله Rob Pike در زمینه طراحی سیستم عامل‌ها و زبان‌های Concurrent است . هر چه باشد ، کار این افراد در گذشته و حال ساخت و طراحی سیستم عامل بوده است .
از آنجایی که برنامه نویسی Concurrent در گولنگ اهمیت زیادی دارد ، طراحان زبان یک قابلیت منحصر به فرد را برای این منظور در زبان جاسازی کرده اند و آن Goroutines است .
اگر یادتان باشد گفتیم که Concurrency یعنی برنامه نویسی بر مبنای مجموعه‌ای از واحدهای اجرایی مستقل ، اما نگفتیم منظورمان از واحد اجرایی مستقل چیست . در واقع ، هر زبانی دیدگاه خاص خودش را به این مفهموم دارد .
در زبان گولنگ ساختاری به اسم Goroutine بیانگر این واحد اجرایی مستقل است .
یک Goroutine در واقع همانند یک Coroutine است و تشابه اسمی آن‌ها بی دلیل نیست . اما قبل از هر چیز توضیح کوتاهی داشته باشیم برای افرادی که با Coroutineها آشنایی ندارند .
در زبان‌های برنامه نویسی وقتی یک تابع اجرای می‌شود ، اجرای آن تابع تا زمای که کارش به طور کامل به پایان نرسیده و یا مقداری از آن برگشت داده نشده ادامه خواهد داشت . Coroutine تابعی است که می‌تواند اجرایش را در میانه راه متوقف کند و به حالت Standby برود .

بعدها اگر دوباره به این تابع بازگشتیم ، اجرای تابع از جایی که قبلا متوقف شده بود ادامه پیدا می‌کند. این قابلیت برنامه نویس را قادر می‌سازد تا Coroutineها را زمانبندی نماید و یا بین آن‌ها سوییچ کند .
رمز کار در این است که علاوه بر داشتن یک Stack سراسری برای نگه داری وضعیت کلی برنامه ، برای هر Coroutine نیز یک Stack جداگانه ساخته می‌شود تا Coroutine بتواند وضعیت فعلی خود را به هنگام سوییچ شدن در آن ذخیره کند .
یک Coroutine بسیار شبیه یک Thread است . وقتی سیستم عامل از یک Thread به Thread دیگری سوییچ می‌کند ، Thread قبلی را به حالت Standby فرو می‌برد تا وقتی دوباره به آن برگشت آن Thread قادر به ادامه اجرایش باشد . برای همین است که گاهی Coroutineها را با نام Green Thread هم صدا می‌زنند چرا که رفتار آن‌ها بسیار شبیه Threadها است .

Coroutineها در زبان‌هایی مثل Python ، Ruby ، Lua ، Perl ، Scheme ، Haskell ، Erlang و خیلی زبان‌های دیگر وجود دارند ، هرچند که ممکن است با اسم متفاوتی ظاهر شوند .

برای مثال در Python با نام Greenlet ، در Ruby با نام Fiber و یا در Erlang با نام Lightweight Process شناخته می‌شوند . البته هر کدام از این زبان‌ها سلیقه خاص خودشان را در پیاده سازی Coroutineها اعمال کرده اند .

***Coroutineها چندین فرق اساسی با Threadها دارند :***

- عمل زمانبندی و سوییچ کردن بین Threadها به صورت اتوماتیک و توسط سیستم عامل انجام می‌شود . اما در Coroutineها خود برنامه نویس باید به صورت دستی زمانبندی و کنترل اجرای Coroutineها را مدیریت کند .
- از آن جایی که Coroutineها در واقع نوعی از توابع هستند ، قالبا در داخل یک Thread اجرا می‌شوند و به همین خاطر قادر نیستند از چندین هسته پردازنده استفاده کنند . اما Threadها می‌توانند براحتی روی هسته‌های مختلف پخش شوند .
- Coroutineها در لایه کاری خود زبان برنامه نویسی مثل سیستم Runtime یا VM آن زبان اجرا و مدیریت می‌شوند . اما Threadها در لایه کاری سیستم عامل اجرا و مدیریت می‌شود .
- چون Coroutineها نوعی از توابع معمولی هستند و در لایه کاری زبان برنامه نویسی اجرا می‌شوند ، پس اجرای آن‌ها در حافظه و یا سوییچ کردن بین آن‌ها ده‌ها و شاید هم صدها برابر سریعتر و بهینه تر از Thread هاست .

یک Groutine در واقع پیاده سازی منحصر به فرد گولنگ از Coroutine هاست که به عنوان واحد اصلی Concurrency در این زبان جاسازی شده است . در ادامه با خصوصیات Goroutineها آشنا می‌شوید :

- اجرای Goroutineها در لایه سیستم عامل صورت نمی‌گیرد و در لایه خود زبان (سیستم Runtime) مدیریت می‌شود . (البته وظیفه اجرای Grotoutineها در لایه سیستم عامل به عهده Thread هاست)
- زمانبندی Goroutineها به طور اتوماتیک توسط سیستم Runtime زبان انجام می‌شود و این مسئولیت از دوش برنامه نویس برداشته شده است .
- سیستم Runtime می‌تواند Goroutineها را روی چندین Thread پخش کند و چون Threadها نیز می‌توانند روی هسته‌های مختلف CPU پخش شوند ، پس اجرای واقعی Parallel اتفاق می‌افتد .
- سیستم Runtime در Go فقط مسئول کنترل Goroutineها در یک ماشین است . یعنی یک ماشین با چند CPU ، یا یک CPU چند هسته ای . پردازش‌های Distributed باید توسط خود برنامه نویس طراحی شود .
- Goroutineها بسیار سبک و بهینه هستند . در کامپیوتری که ممکن است با ایجاد ۱.۰۰۰ عدد Thread کرش کند ، می‌توان ۱.۰۰۰.۰۰۰ عدد از Goroutine را اجرا کرد ! سایز پیش فرض Stack برای هر Goroutine فقط ۴kB است .
- Goroutineها بلاک نمی‌شوند . اگر در یک Goroutine عملیات بلاک شونده I/O صورت بگیرد ، بقیه Goroutineها در Thread دیگری به اجرای خودشان ادامه می‌دهند .
- در جایی هم که ممکن باشد ، سیستم Runtime خود به خود از عملیات Asynchronous برای رخدادهای I/O استفاده می کند . شما نیاز نیست با مدل برنامه نویسی Asynchronous درگیر شوید .
- Goroutineها بر مبنای سیستم انتقال پیام (Message Passing) کار می‌کنند و به این شیوه قادرند با یکدیگر در ارتباط باشند . در گولنگ پیام‌ها توسط Channelها که در واقع همان کانال‌های ارتباطی بین Goroutineها هستند رد و بدل می‌شوند .
- به صورت پیش فرض ، عمل انتقال پیام در گولنگ به شکل Synchronous اتفاق می‌افتد . یعنی پیام فقط زمانی فرستاده می‌شود . که فرستنده و گیرنده هر دو آماده باشند. این قضیه باعث ساده تر شدن برنامه نویسی می‌شود . البته در صورت لزوم می‌توانید عمل انتقال پیام را به شکل Asynchronous نیز انجام دهید.
- Channelهای گولنگ مانند خود زبان Static Type هستند . اگر یک channel تعیین کند که قرار است فقط داده‌های int را ردو بدل کند، داده دیگری از آن نخواهد گذشت.

احتمالا متوجه شباهت Goroutineهای گولنگ و Lightweightهای Erlang شده اید . البته این دو جدای از شباهت ظاهری ، تفاوت‌های پایه‌ای بسیاری با یکدیگر دارند . Goroutineها بر مبنای مدل CSP پیاده سازی شده اند در حالی که Lightweight Processها بر مبنای مدل Actor توسعه پیدا کرده اند .

***مهم ترین تفاوت مدل CSP و مدل Actor به شرح زیر است :***

- در مدل CSP واحدهای اجرایی بی نام هستند در حالی که در مدل Actor دارای شناسه می‌باشند .
- ارسال پیام در مدل CSP به شکل Synchronous انجام می‌گیرد در حالی که در مدل Actor به شکل Asynchronous اتفاق می‌افتد .
- ارسال پیام در مدل CSP به کمک Channelها انجام می‌گیرد ولی در مدل Actor مستقیم و بدون واسطه است .

هر کدام از این مدل‌ها مزایا و معایب خودشان را دارند . همچنین باید دقت داشت که گولنگ مانند Erlang یک زبان Functional نیست و از ساختار Immutable استفاده نمی‌کند ، پس لازم است برنامه نویس کمی بیشتر در ساخت برنامه‌های Concurrent محتاط باشد .

Rob Pike در یک ویدیو آموزشی در YouTube به نام Go Concurrency Patterns مثالی جالب از توانایی Goroutineها را به همگان نشان داد . او کدی نوشته بود که صدهزار Goroutine را در حافظه ایجاد می‌کرد . سپس یک عدد int بین این Goroutineها دست به دست می‌چرخید و هر Goroutine هم یک واحد به آن اضافه می‌کرد .
دقت کنید که برنامه کامپایل نشده بود . بنابراین وقتی Pike دکمه Run را فشار می‌داد ، برنامه باید کامپایل می‌شد ، لینک می‌شد ، در حافظه اجرا می‌شد و جواب اجرا برگشت داده می‌شد . کل این پروسه فقط یک ثانیه به طول انجامید .
احتمالاً اجرای یک برنامه Hello World در Java یا #C که از قبل هم کامپایل شده باشد ، ممکن است بیشتر از یک ثانیه به طول انجامد ! این حرف جنبه شوخی داشت و از نظر علمی چیزی را ثابت نمی‌کند .

## گولنگ و قابلیت‌های Functional

در سال‌ها اخیر زبان‌های Functional از سایه بیرون آمده اند و محبوبیت خوبی برای خود دست و پا کرده اند . مخصوصا بعد از معرفی مدل MapReduce از طرف گوگل که با الهام گیری از زبان‌های Functional شکل گرفته بود ، و همچنین رایج شدن CPUهای چند هسته‌ای و نیاز به بستر مناسب برای Concurrency و پیشرو بودن زبان‌های Functional در این زمینه ، توجه همه به آن‌ها معطوف شده است.
توابع ()map و ()reduce از توابع اساسی زبان‌های Functional می‌باشند و تقریبا در تمام زبان‌های Functional حضور دارند.
حتی زبان‌های غیر Functional مثل JavaScript ، Ruby ، Python و ... هم با اینکه جزو زبان‌های Functional به حساب نمی‌آیند اما دلشان نیامده تا بعضی از ویژگی‌های زبان‌های Functional را ارایه نکنند .

گولنگ یک زبان سیستمی است؛ یعنی هر چقدر هم که ساده باشد ، باید همانند C جنبه‌های سطح پایین خود را حفظ کند . نباید انتظار داشت که چنین زبانی گرایش Functional داشته باشد. اما طراحان گولنگ ترجیح داده اند که کمی هم چاشنی Functional به زبان اضافه کنند .

***در گولنگ ، توابع جزو اعضای درجه اول زبان به حساب می‌آیند (First-Class) یعنی می‌توان :***

- یک تابع را همانند مقادیر معمولی به عنوان آرگومان به توابع دیگر ارسال کرد .
- یک تابع را به عنوان جواب خروجی از تابع دیگر برگشت داد .
- یک تابع را به یک متغیر نسبت داد؛ به همان صورتی که یک عدد را به یک متغیر int نسبت می‌دهیم .

گولنگ همچنین دارای قابلیت استفاده از توابعی به نام (Anonymous Functions) است . توابع Anonymous توابعی هستند که می‌توانند به صورت لحظه‌ای تولید شوند و مورد استفاده قرار بگیرند؛ یعنی نیازی نیست که مانند توابع معمولی از قبل آن‌ها را در جایی از کدهایتان تعریف کرده باشید . این توابع خصوصا هنگام کار با Goroutineها بسیار مفید واقع می‌شوند .

وقتی زبانی دارای توابع First-Class باشد و امکان تعریف توابع Anonymous را هم داشته باشد ، یعنی Closureها نیز در آن زبان حضور دارند . کار Closureها بر بنای توابع تو در تو استوار است . ممکن است برنامه نویسان که کمتر با زبان‌های Functional آشنایی

دارند با Closureها آشنا نباشند . پس بهتر است کمی در باره Closureها توضیح دهیم .

## توضیح درباره Closure ها

به حالت معمول فیلدهایی که داخل یک تابع تعریف می‌شوند پس از اتمام اجرای آن تابع از بین می‌روند . در زبان‌هایی که توابع در آن‌ها First-Class هستند ، خروجی یک تابع می‌تواند یک تابع دیگر باشد؛ به همان صورتی که ممکن است یک عدد int به عنوان خروی در نظر گرفته شود .
فرض کنیم تابعی داریم به نام Outer که یک زیر تابع به اسم Inner را به عنوان خروجی برگشت می‌دهد . مسلما وقتی که تابع Outer تابع Inner را برگشت می‌دهد ، اجرای تابع Outer تمام شده قلمداد می‌گردد . در چنین حالتی باید تمام فیلدها و اطلاعات تابع Outer از بین برود .
اما اگر تابع Inner که به عنوان خروجی برگشت داده شده ، از فیلدهای تابع Outer استفاده کرده باشد ، آن فیلدها تا زمان زنده بودن تابع Inner از بین نخواهند رفت ! بنابراین تابع Inner می‌تواند حتی بعد از اتمام کار تابع Outer هم از فیلدهای آن استفاده کند . تابعی مانند Inner که می‌تواند برای استفاده داخلی خودش ، داده‌هایی از خارج را به خود وابسته سازد ، Closure نامیده می‌شود .
درک بهتر Closureها به کمی زمان و تمرین نیاز دارد؛ اما بدانید که اگر با کتابخانه jQuery کار می‌کنید ، احتمالا ده‌ها بار بدن اینکه خودتان متوجه شوید از Closureها استفاده کرده اید .

## بررسی کتابخانه های گولنگ

کتابخانه استاندارد گولنگ یکی از جامع ترین کتابخانه‌های موجود در بین تمام زبان‌های برنامه نویسی است . از طرفی چون گولنگ یک زبان نو ظهور است ، طراحی کتابخانه در حالت بسیار تمیز و یکپارچه‌ای قرار دارد .

- Packageهایی برای آرشیو و فشرده سازی : tar, zip, bzip2, flate, gzip, lzw, zlib
- Packageهایی برای رمزنگاری و عملیات هش : aes, cipher, des, dsa, rc4, rsa, md5, sha1, sha256, sha512, tls, x509
- Packageهایی برای کار با فایل‌های مختلف : base32 ,base64, binary, csv, gob, hex, pem, json, xml
- Packageهایی برای کار با گرافیک دو بعدی : color, draw, gif, jpeg, png
- Packageهایی برای کار با مباحث شبکه : html, cgi, fcgi, mail, url, jsonrpc

ذات سیستمی گولنگ در اینجا مشخص می‌شود . تمام Packageهای بالا در گولنگ نوشته شده اند . برای داشتن خیلی از این Packageها در زبان‌های دیگر، باید متوسل به کتابخانه‌های نوشته شده در زبان C می‌شدیم.
خوبی گولنگ در این است که برای اعمال سطح پایین دیگر نیازی به C نیست . گولنگ در واقع همان C مدرن است ، از طرف همان کسانی که روزی C و Unix را به شما معرفی کرده بودند !
همه Packageهای بالا را همراه کنید با Packageهایی برای عملیات ریاضی ، زمان و تاریخ ، محیط سیستم عامل ، پایگاه داده ، ورودی و خروجی ، رشته ها ، پردازش و Parse کردن متن و ...
در ضمن کتابخانه استاندارد با یک HTTP Server داخلی همراه است که به راحتی می‌توان آن را با Nginx یا Node.js مقایسه کرد . در واقع ، اکثر سایت‌هایی که در ابتدای این نوشته به آن‌ها اشاره شد هم از همین Server داخلی برای خدمات رسانی به کاربرانشان استفاده می‌کنند .
همچنین در کتابخانه استاندارد گولنگ یک سیستم Template Engine ارائه شده تا برنامه نویسی وب را برای شما آسان تر کند . با این حساب ، احتمالا متوجه شده اید که برنامه نویسی وب در گولنگ ، نیاز چندانی به فریم‌ورک‌های مرسوم در زبان‌های دیگر نخواهید داشت . کتابخانه استاندارد گولنگ همه چیز را از قبل برایتان مهیا کرده است .
از نظر مستندات ، گولنگ در جایگاه بسیار خوبی قرار دارد و برای تمام Packageها و تک تک توابع و پارامترهای شان به طور کامل مستندات وجود دارد .

اگر زبان انگلیسی شما در حد مطلوبی قرار دارد (که به عنوان یک برنامه نویس باید هم این چنین باشد) در زمینه یادگیری گولنگ با هیچ مشکلی مواجه نخواهید شد و نیاز به هیچ کتاب و منبع خاصی نخواهید داشت؛ چرا که مستندات موجود در سایت گولنگ به اندازه کافی کامل و مناسب است .
با این که زمان زیادی از انتشار نسخه پایدار نمی‌گذرد ، اما گولنگ ابزارهای جانبی نستباً کاملی در اختیار دارد :

- ابزار Go که کار کامپایل و نصب Package را آسان کرده است . این ابزار حتی قابلیت این را دارد که Packageها را به صورت اتوماتیک از سایت‌هایی مثل Github دریافت و نصب نماید !
- ابزار Godoc قادر است سایت golang.org را به صورت محلی در کامپیوتر شما اجرا کند . سایت اصلی golang.org هم به کمک همین ابزار در حال اجراست . همچنین godoc می‌تواند مستندات مربوط به Packageها و توابع آن‌ها را مستقیما در ترمینال نمایش دهد .
- ابزار Gofmt استایل کد نویسی شما را مدیریت می‌کند ، برای مثال هرکدام ازا عضای تیم می‌توانند استایل خودشان را داشته باشند، اما در نهایت از Gofmt برای یک پارچه کردن استایل کدهای پروژه استفاده کنند .
- ابزار Gocode هم برای کمک به ادیتورها و IDEها طراحی شده . این ابزار بررسی ، کدهای شما و اطلاعات موجود در آن‌ها را استخراج کرده و در اختیار ادیتورها و IDEهای می‌گذارد. آن‌ها هم می‌توانند از این اطلاعات برای پیاده سازی قابلیت Auto completion استفاده نمایند .
- مهم ترین ابزار کار هر برنامه نویس ویرایشگر متن است . خوشبختانه گولنگ از پشتیبانی خوبی در این زمینه برخوردار است . در رابطه با ویرایشگرهای ساده و سبک ، گولنگ از Gedit ، BBEdit ، ++Notepad ، Kate و ... به صورت رسمی پشتیبانی می‌کند.

برای کسانی که با ویرایشگرهای حرفه‌ای کدنویسی می‌کنند ، گولنگ پشتیبانی رسمی و کاملی را برای ویرایشگرهای بی رقیب Vim و Emacs ارائه کرده است . در واقع از آنجایی که تیم سازندگان گولنگ و اکثر جامعه کابران آن نیز با همین ابزارها کدنویسی می‌کنند ، پشتیبانی از این دو از اولویت بسیار بالایی برخوردار است .
در گولنگ همانند دیگر زبان‌های کامپایلری نیاز خاصی به وجود IDE حس نخواهید کرد؛ اما برای کسانی که کار با IDEها را ترجیح می‌دهند ، گولنگ از پشتیبانی مناسبی برای IDEهای IntelliJ ، Eclipse و VSCode برخوردار است .

## اجرا اولین برنامه در زبان برنامه نویسی گو

ابتدا فایلی با فرمت go مانند hello.go بسازید و کد زیر را در آن قرار دهید :

کد شماره ۱ http://play.golang.org/p/2C7wwJ6nxG

```
// Our first program will print the classic "hello world"
// message. Here's the full source code.
package main

import "fmt"

func main() {
    fmt.Println("hello world")
}
```

حالا برای اجرای کد دو حالت وجود داره اگه از دستور `go build` استفاده کنید ابتدا کد شما به فایل باینری تبدیل و خروجی آن ساخته می شود که در محیط ترمینال می توانید اجرا کنید و اگر از دستور `go run` استفاده کنید کد شما ابتدا در temporary سیستم ساخته و سپس اجرا می شود .

```
$ go run listing1.go
hello world
Sometimes we'll want to build our programs into binaries. We can do this using go build.
$ go build hello-world.go
$ ls
hello-world     hello-world.go
We can then execute the built binary directly.
$ ./hello-world
hello world
```

***نکات قابل توجه :***

- همیشه برای آغاز یک برنامه باید نامی به عنوان نام پکیج در ابتدای فایل مانند `package main` قرار گیرد . شما می توانید نام های مختلفی را برای `namespace` خود انتخاب کنید اما فایل اصلی که دستور اجرا برنامه گو روی آن صادر می شود حتما باید نام پکیج آن `main` باشد .
- دقت داشته باشیم وقتی برنامه آغاز به کار می کند فقط کدهایی که درون تابع `main` قرار می دهیم اجرا می شود پس بهتر است کدهایمان را به صورت فانکشنال خارج آن بنویسیم و داخل تابع `main` صدا بزنیم .
- تابعی با نام `init` در گو وجود دارد که در صورت تعریف آن با اجرای برنامه ابتدا محتویات داخل آن اجرا شده و بعد تکمیل به سراغ تابع `mian` می رود همانند کد زیر :

کد شماره ۲ https://play.golang.org/p/vyoCoYrzTpo

```
package main

import "fmt"

func init() {
    fmt.Println("first")
```

```
}

func main() {
    fmt.Println("second")
}
```

## Values در گولنگ

گولنگ همانند زبان های دیگر انواع مختلفی از مقادیر را شامل می شود برای مثال : رشته ها ، اعداد صحیح ، اعداد شناور (اعشاری) ، بولین ها و غیره .

کد شماره ۳ http://play.golang.org/p/fgGVOyuZdu

```
// Go has various value types including strings,
// integers, floats, booleans, etc. Here are a few
// basic examples.

package main

import "fmt"

func main() {

    // Strings, which can be added together with `+`.
    fmt.Println("go" + "lang")

    // Integers and floats.
    fmt.Println("1+1 =", 1+1)
    fmt.Println("7.0/3.0 =", 7.0/3.0)

    // Booleans, with boolean operators as you'd expect.
    fmt.Println(true && false)
    fmt.Println(true || false)
    fmt.Println(!true)
}
```

خروجی :

```
$ go run listing3.go
golang
1+1 = 2
7.0/3.0 = 2.3333333333333335
false
true
false
```

- رشته ها را با علامت + می توان کنار هم یا به اصطلاح جمع کرد .

## Variables در گولنگ

همان طور که در گولنگ نمی توان پکیجی را لود کرد و استفاده نکرد در اینجا نیز در صورتی که متغیری میسازید باید از آن در برنامه خودتان استفاده کنید به کد زیر دقت کنید و ببینید متغیر ها باید به صراحت ساخته و در صورت نیاز مقدار دهی شوند .

کد شماره ۴ https://play.golang.org/p/1FnG0dJfxs8

```
// In Go, _variables_ are explicitly declared and used by
// the compiler to e.g. check type-correctness of function
// calls.

package main

import "fmt"

func main() {

    // `var` declares 1 or more variables.
    var a = "initial"
    fmt.Println(a)

    // You can declare multiple variables at once.
    var b, c int = 1, 2
    fmt.Println(b, c)

    // Go will infer the type of initialized variables.
    var d = true
    fmt.Println(d)

    // Variables declared without a corresponding
    // initialization are _zero-valued_. For example, the
    // zero value for an `int` is `0`.
    var e int
    fmt.Println(e)

    // The `:=` syntax is shorthand for declaring and
    // initializing a variable, e.g. for
    // `var f string = "short"` in this case.
    f := "short"
    fmt.Println(f)
}
```

خروجی :

```
$ go run listing4.go
initial
1 2
true
0
short
```

### نکات :

- متغیر a با کلمه کلیدی var بدون تعیین نوع متغیر ساخته و مقدار دهی شده است . در صورت تعیین نکردن نوع متغیر این کار به صورت خودکار انجام می شود .
- در تعریف متغیر f مشاهده می کنیم متغیری ساخته و مقدار دهی شده بدون تعیین نوع متغیر در واقع نوع تعریف کوتاه متغیر به این صورت است که نیازی نیست کلمه کلیدی var نوشته شود و فقط کافیست به جای = از := استفاده شود . برای تعریف این متغیر به شکل ساده و با تعیین نوع متغیر باید از کد "var f string = "short استفاده کرد .
- در صورت عدم مقداری دهی با توجه به نوع متغیر مقداری ثابت در نظر گرفته می شود برای مثال مقدار متغیر e برابر 0 خواهد بود ، پس در صورتی که متغیری بدون مقدار تعیین می کنید باید حتما نوع آن را نیز تعیین کنید .
- متغیرهای b و c نیز به سینتکس تعریف چند متغیر با یک بار استفاده از کلمه کلیدی var اشاره دارد و می توانید چندین متغیر در یک خط بسازید . الگوی دیگر :

```
var (
        b int = 1
        c int = 2
)
```

## Constants در گولنگ

ثابت ها در گولنگ از مقادیر کارکتر ، رشته ، بولین و اعداد پشتیبانی می کنند .

کد شماره ۵ https://play.golang.org/p/T5sj0elNnp

```
// Go supports _constants_ of character, string, boolean,
// and numeric values.

package main

import "fmt"
import "math"

// `const` declares a constant value.
const s string = "constant"

func main() {
    fmt.Println(s)

    // A `const` statement can appear anywhere a `var`
    // statement can.
    const n = 500000000

    // Constant expressions perform arithmetic with
    // arbitrary precision.
    const d = 3e20 / n
    fmt.Println(d)

    // A numeric constant has no type until it's given
    // one, such as by an explicit cast.
    fmt.Println(int64(d))
```

```
    // A number can be given a type by using it in a
    // context that requires one, such as a variable
    // assignment or function call. For example, here
    // `math.Sin` expects a `float64`.
    fmt.Println(math.Sin(n))
}
```

خروجی :

```
$ go run listing5.go
constant
6e+11
600000000000
-0.28470407323754404
```

**نکات :**

- ثابت ها هر جایی که می توان متغیر ساخت قابل ساخت هستند .
- تفاوت کلی ثابت ها با متغیر در این است که مقدار ثابت ها قابل تغییر نیست .
- اگه شما هم با `Javascript` یا `Typescript` که از استانداردهای `Ecmascript` بهره می برد کار می کنید توجه داشته باشید در گولنگ نمی توان داخل یک ثابت تابعی قرار داد و فقط دیتا تایپ های مشخصی که ذکر شد قابل استفاده هستند .

## For در گولنگ

تنها متد ایجاد حلقه در گولنگ استفاده از کلمه کلیدی For است .

کد شماره ۶ https://play.golang.org/p/KNLLSX4Io_

```
// `for` is Go's only looping construct. Here are
// three basic types of `for` loops.

package main

import "fmt"

func main() {

    // The most basic type, with a single condition.
    i := 1
    for i <= 3 {
        fmt.Println(i)
        i = i + 1
    }

    // A classic initial/condition/after `for` loop.
    for j := 7; j <= 9; j++ {
        fmt.Println(j)
    }

    // `for` without a condition will loop repeatedly
    // until you `break` out of the loop or `return` from
    // the enclosing function.
    for {
        fmt.Println("loop")
        break
    }

    // You can also `continue` to the next iteration of
    // the loop.
    for n := 0; n <= 5; n++ {
        if n%2 == 0 {
            continue
        }
        fmt.Println(n)
    }
}
```

خروجی :

```
$ go run listing6.go
1
2
3
7
8
9
loop
1
3
5
```

**نکات :**

- ساده ترین متد استفاده از حلقه مثال اول است .
- در مثال دوم متغیر `j` داخل حلقه ساخته و مقداردهی شده و شرط و تغییرات روی آن تعریف شده و در اصطلاح به آن `classic initial/condition/after` در حلقه گویند .
- مثال سوم حلقه ای بی نهایت است که می توان با کلمه کلیدی `break` یا `return` آن را متوقف کرد .
- در مثال آخر برای نمایش هر `iteration` حلقه با استفاده از شرط ، از کلمه کلیدی `continue` استفاده می شود .
- کلمات کلیدی `break` و `continue` در اکثر زبان های برنامه نویسی مشترک است .

## If/Else در گولنگ

تعریف شرط `If/Else` یکی از ساده ترین سینتکس ها در گولنگ می باشد .

کد شماره ۷ https://play.golang.org/p/g-aqMz0Ivf

```
// Branching with `if` and `else` in Go is
// straight-forward.

package main

import "fmt"

func main() {

    // Here's a basic example.
    if 7%2 == 0 {
        fmt.Println("7 is even")
    } else {
        fmt.Println("7 is odd")
    }

    // You can have an `if` statement without an else.
    if 8%4 == 0 {
        fmt.Println("8 is divisible by 4")
    }

    // A statement can precede conditionals; any variables
    // declared in this statement are available in all
    // branches.
    if num := 9; num < 0 {
        fmt.Println(num, "is negative")
    } else if num < 10 {
        fmt.Println(num, "has 1 digit")
    } else {
        fmt.Println(num, "has multiple digits")
    }
}

// Note that you don't need parentheses around conditions
// in Go, but that the braces are required.
```

خروجی :

```
$ go run listing7.go
7 is odd
8 is divisible by 4
9 has 1 digit
```

### نکات :

- مثال اول به شکل ساده یک شرط و خلاف آن را سنجیده ایم .
- در مثال دوم می بینیم که می توان شرطی را بدون `else` به کار برد .
- در مثال سوم نحوه استفاده از چند شرط با استفاده از `else if` نشان داده شده و متغیری که در شرط تعریف می شود در داخل همه فضاهای شرطی قابل دسترسی است .
- در گولنگ نیاز نیست اطراف شرط از پرانتز استفاده کنید اما محتوای داخل آن باید داخل یک فضای `{ }` قرار بگیرد .

## Switch در گولنگ

برای همه ما پیش میاد که نیاز داریم شرطهای زیادی را بسنجیم و باید چندین بار عبارت `else if` را تایپ کنیم اما اکثر زبان های برنامه نویسی از ویژگی با عنوان `Switch` بهره می برند که سنجش شرط های زیاد را ساده تر می کند .

کد شماره ۸ https://play.golang.org/p/TJ4Az0KuLfL

```
// _Switch statements_ express conditionals across many
// branches.

package main

import "fmt"
import "time"

func main() {

    // Here's a basic `switch`.
    i := 2
    fmt.Print("Write ", i, " as ")
    switch i {
    case 1:
        fmt.Println("one")
    case 2:
        fmt.Println("two")
    case 3:
        fmt.Println("three")
    }

    // You can use commas to separate multiple expressions
    // in the same `case` statement. We use the optional
    // `default` case in this example as well.
    switch time.Now().Weekday() {
    case time.Saturday, time.Sunday:
        fmt.Println("It's the weekend")
    default:
        fmt.Println("It's a weekday")
    }

    // `switch` without an expression is an alternate way
    // to express if/else logic. Here we also show how the
    // `case` expressions can be non-constants.
```

```
    t := time.Now()
    switch {
    case t.Hour() < 12:
        fmt.Println("It's before noon")
    default:
        fmt.Println("It's after noon")
    }

    // A type `switch` compares types instead of values.  You
    // can use this to discover the type of an interface
    // value.  In this example, the variable `t` will have the
    // type corresponding to its clause.
    whatAmI := func(i interface{}) {
        switch t := i.(type) {
        case bool:
            fmt.Println("I'm a bool")
        case int:
            fmt.Println("I'm an int")
        default:
            fmt.Printf("Don't know type %T\n", t)
        }
    }
    whatAmI(true)
    whatAmI(1)
    whatAmI("hey")
}
```

خروجی :

```
$ go run listing8.go
Write 2 as two
It's a weekday
It's after noon
I'm a bool
I'm an int
Don't know type string
```

**نکات :**

- در اولین مثال مشاهده می کنیم سینتکس تعریف سوییچ را به ساده ترین شکل ممکن.
- در مثال دوم مشاهده می کنیم که می توان به کمک کاما `,` چند مقدار برای یک گزینه یا `case` انتخاب کنیم و با استفاده از کلمه کلیدی `default` می توانیم در صورتی که مقدار ورودی با هیچ گزینه ای مطابقت نداشت آن را نمایش دهیم .
- در مثال سوم مشاهده می کنیم که می توان به سوییچ مقداری نداد تا نقش `If/Else` را ایفا کند و داخل `case` شرطی را قرار داد که وابستگی به ورودی ندارد .
- در مثال آخر کلید واژه `Interface` رو می بینیم که در قسمت های بعد کامل توضیح داده می شود و در این مثال ما نوع مقادیر را می سنجیم و یکی از سینتکس های تعریف یک تابع را مشاهده می کنیم که در ادامه آموزش ها به نحوه تعریف توابع خواهیم پرداخت .

## Arrays در گولنگ

در گولنگ `Data Type` دیگری با عنوان `Collection` وجود دارد که همان آرایه محسوب می شود .

کد شماره ۹ https://play.golang.org/p/l-A8eBnwio

```
// In Go, an _array_ is a numbered sequence of elements of a
// specific length.

package main

import "fmt"

func main() {

    // Here we create an array `a` that will hold exactly
    // 5 `int`s. The type of elements and length are both
    // part of the array's type. By default an array is
    // zero-valued, which for `int`s means `0`s.
    var a [5]int
    fmt.Println("emp:", a)

    // We can set a value at an index using the
    // `array[index] = value` syntax, and get a value with
    // `array[index]`.
    a[4] = 100
    fmt.Println("set:", a)
    fmt.Println("get:", a[4])

    // The builtin `len` returns the length of an array.
    fmt.Println("len:", len(a))

    // Use this syntax to declare and initialize an array
    // in one line.
    b := [5]int{1, 2, 3, 4, 5}
    fmt.Println("dcl:", b)

    // Array types are one-dimensional, but you can
    // compose types to build multi-dimensional data
    // structures.
    var twoD [2][3]int
    for i := 0; i < 2; i++ {
        for j := 0; j < 3; j++ {
            twoD[i][j] = i + j
        }
    }
    fmt.Println("2d: ", twoD)
}
```

خروجی :

```
$ go run listing9.go
emp: [0 0 0 0 0]
set: [0 0 0 0 100]
get: 100
len: 5
dcl: [1 2 3 4 5]
2d:  [[0 1 2] [1 2 3]]
```

## نکات :

- هر آرایه ای دو خصوصیت `Data Type` و `Length` را دارا می باشد . ساده ترین متد تعریف یک آرایه در گولنگ مثال اول است که داخل متغیر `a` آرایه ای شامل ۵ خانه که همان صفت `Length` است و با `Data Type` عدد صحیح `int` تعریف شده است .
- اگر آرایه رو ساختید ولی مقدار دهی نکردید مقادیر ثابتی در خانه ها میشیند برای مثال اگر نوع داده ها `int` باشد در هر خانه `0` مینشیند .
- بر خلاف زبان هایی که از استاندارد `Ecmascript` استفاده می کنند نوع داده های داخل یک آرایه باید یکسان باشد .
- برای مقدار دادن به یک خانه خاص در آرایه می توان از متد `array[index] = value` استفاده کرد .
- در گولنگ تابعی با عنوان `len` وجود دارد که طول آرایه یا همان `Length` را نمایش می دهد .
- برای مقدار دادن به یک آرایه هنگام تعریف آن می توان از { } استفاده کرد برای مثال نحوه تعریف آرایه در متغیر `b` را مشاهده نمایید .
- تا اینجا یاد گرفتیم چطور آرایه یک بعدی بسازیم اما برای تعریف آرایه چند بعدی می توان از مثال آخر استفاده کرد که آرایه ای ۲ بعدی تعریف شده است .
- می توان به جای تعداد خانه ها از `...` به صورت `[...]string{"Penn", "Teller"}` استفاده نمود و در این حالت موقع کامپایل تعداد خانه ها شمرده می شود و جایگزین .
- نکته بسیار مهمی که در آرایه ها وجود دارد این است که اگر آرایه رو به عنوان یک پارامتر درون یک تابع ارسال کنیم اون مقدار داخل تابع یک کپی از آرایه محسوب می شود در صورتی که در Slice ها چنین نیست و برای رفع این محدودیت می توان از پوینتر ها برای رفرنس دادن موقعیت آرایه بر روی memory استفاده کرد .

کد شماره ۱۰ https://play.golang.org/p/Lu01QP5AOri

```
package main

import (
        "fmt"
)

func main() {
        myArray := [...]string{"Apples", "Oranges", "Bananas"}
        fmt.Printf("Initial array values: %v\n", myArray)
        myFunction(myArray)
        fmt.Printf("Final array values: %v\n", myArray)
}

func myFunction(arr [3]string) {
        // Change Oranges to Strawberries
        arr[1] = "Strawberries"
```

```
        fmt.Printf("Array values in myFunction(): %v\n", arr)
}

// Output
// Initial array values: [Apples Oranges Bananas]
// Array values in myFunction(): [Apples Strawberries Bananas]
// Final array values: [Apples Oranges Bananas]
```

- با چاپ یک آرایه مقداری همانند `[v1 v2 v3 ...]` نمایش داده می شود .
- نکته مهمی که در آرایه ها وجود دارد اینکه شما نمی توانید تعداد خانه های آن را بیشتر از مقدار تعیین شده قرار بدید و چندان قابل انعطاف نیست .

## Slices در گولنگ

Slice ها یک نوع داده کلیدی در گولنگ محسوب می شوند که یک رابط قوی برای مدیریت آرایه ها فراهم می کنند . در واقع اگر می خواهید آرایه ها امکانات بیشتری و قابل انعطاف تر باشند باید از Slice ها استفاده کرد .

کد شماره ۱۱ https://play.golang.org/p/Z3_U32sn8RF

```
// _Slices_ are a key data type in Go, giving a more
// powerful interface to sequences than arrays.

package main

import "fmt"

func main() {

    // Unlike arrays, slices are typed only by the
    // elements they contain (not the number of elements).
    // To create an empty slice with non-zero length, use
    // the builtin `make`. Here we make a slice of
    // `string`s of length `3` (initially zero-valued).
    s := make([]string, 3)
    fmt.Println("emp:", s)

    // We can set and get just like with arrays.
    s[0] = "a"
    s[1] = "b"
    s[2] = "c"
    fmt.Println("set:", s)
    fmt.Println("get:", s[2])

    // `len` returns the length of the slice as expected.
    fmt.Println("len:", len(s))

    // In addition to these basic operations, slices
    // support several more that make them richer than
    // arrays. One is the builtin `append`, which
    // returns a slice containing one or more new values.
```

```
    // Note that we need to accept a return value from
    // `append` as we may get a new slice value.
    s = append(s, "d")
    s = append(s, "e", "f")
    fmt.Println("apd:", s)

    // Slices can also be `copy`'d. Here we create an
    // empty slice `c` of the same length as `s` and copy
    // into `c` from `s`.
    c := make([]string, len(s))
    copy(c, s)
    fmt.Println("cpy:", c)

    // Slices support a "slice" operator with the syntax
    // `slice[low:high]`. For example, this gets a slice
    // of the elements `s[2]`, `s[3]`, and `s[4]`.
    l := s[2:5]
    fmt.Println("sl1:", l)

    // This slices up to (but excluding) `s[5]`.
    l = s[:5]
    fmt.Println("sl2:", l)

    // And this slices up from (and including) `s[2]`.
    l = s[2:]
    fmt.Println("sl3:", l)

    // We can declare and initialize a variable for slice
    // in a single line as well.
    t := []string{"g", "h", "i"}
    fmt.Println("dcl:", t)

    // Slices can be composed into multi-dimensional data
    // structures. The length of the inner slices can
    // vary, unlike with multi-dimensional arrays.
    twoD := make([][]int, 3)
    for i := 0; i < 3; i++ {
        innerLen := i + 1
        twoD[i] = make([]int, innerLen)
        for j := 0; j < innerLen; j++ {
            twoD[i][j] = i + j
        }
    }
    fmt.Println("2d: ", twoD)
}
```

خروجی :

```
$ go run listing11.go
emp: [  ]
set: [a b c]
get: c
```

```
len: 3
apd: [a b c d e f]
cpy: [a b c d e f]
sl1: [c d e]
sl2: [a b c d e]
sl3: [c d e f]
dcl: [g h i]
2d:  [[0] [1 2] [2 3 4]]
```

### نکات :

- برخلاف آرایه ها ، برش ها (Slices) با Length همیشه ثابتی تعریف نمی شوند و به مقادیری که داخل آن ها ریخته می شود بستگی دارند.
- برای ساخت یک برش خالی و بدون مقدار می توان از تابع make کمک گرفت ، در مثال اول ما یک برش با سایز ۳ و بدون هیچ مقداری ساخته ایم .
- در برش ها نیز همانند آرایه ها می توان مقادیری برای هر خانه انتخاب کرد و آن ها را فراخواند .
- تابع len نیز طول برش را همانند آرایه ها بر می گرداند .
- برش ها قابلیت های بیشتری نسبت به آرایه ها دارند برای مثال به کمک تابع append می توان یک یا چند مقدار را به برش اضافه نمود .
- برای کپی کردن یک برش می توان از دستور copy در تعریف برش جدید بر روی متغیر c یاد کرد که ابتدا برشی با سایز برش مد نظر ساخته می شود و سپس مقادیر داخل آن کپی می شوند .
- برش ها از سینتکس slice[low:high] برای دریافت قسمت مورد نظر شما از داخل برش استفاده می کنند . ( به متغیر l توجه کنید )
- در متغیر t نحوه تعریف یک برش با مقادیر در یک خط نمایش داده شده است .
- برش ها را می شود به ساختار دیتاهای چندبعدی پیاده سازی کرد اما طول مقادیر داخل برش ها می تواند متفاوت باشد و شبیه آرایه های چند بعدی نیست .
- برش ها بر خلاف آرایه ها موقعی که به عنوان یک پارامتر به تابع ارسال می شود به صورت رفرنس عمل کرده و تغییرات روی آن باعث تغییر روی برش اصلی می شود .

کد شماره ۱۲ https://play.golang.org/p/kHqSobJC2Yv

```
package main

import (
        "fmt"
)

func main() {
        mySlice := []string{"Apples", "Oranges", "Bananas"}
        fmt.Printf("Initial slice values: %v\n", mySlice)
        myFunction(mySlice)
        fmt.Printf("Final slice values: %v\n", mySlice)
}

func myFunction(fruits []string) {
        // Change Oranges to Strawberries
        fruits[1] = "Strawberries"
        fmt.Printf("Slice values in myFunction(): %v\n", fruits)
}
```

```
// Output :
// Initial slice values: [Apples Oranges Bananas]
// Slice values in myFunction(): [Apples Strawberries Bananas]
// Final slice values: [Apples Strawberries Bananas]
```

- با این که برش ها کاملتر از آرایه ها هستند اما به صورت مشابه در `fmt.Println` چاپ می شوند .

> نتیجه : در واقع اگر آرایه را با مقدار تعریف کردیم ولی تعداد خانه ها رو مشخص نکردیم یا از `...` نیز استفاده نکردیم یعنی ما از Slice استفاده کردیم و نه Array و روش ساخت بدون مقدار به کمک تابع درونی `make` صورت می گیرد .

### ظرفیت نهایی برش ها

خوب اگه دقت کرده باشید موقع استفاده از `make` ما باید تعداد خونه ها یا ظرفیت `Capacity` را مشخص کنیم برای این کار ، اما این تابع درون ساخته شده متغیر سومی هم دارد که به معنا حداکثر ظرفیت است . در برش ها تابع درونی به نام `cap` وجود داره که ظرفیت برش را نشون میده !
در مثال زیر برشی با ظرفیت اولیه ۴ و حداکثری ۸ ساخته می شود .

کد شماره ۱۳ https://play.golang.org/p/p3ewucCUFer

```
package main

import (
        "fmt"
)

func main() {
        mySlice := make([]int, 4, 8)
        fmt.Printf("Initial Length: %d\n", len(mySlice))
        fmt.Printf("Capacity: %d\n", cap(mySlice))
        fmt.Printf("Contents: %v\n", mySlice)
}

// Output :
// Initial Length: 4
// Capacity: 8
// Contents: [0 0 0 0]
```

خوب چون حداکثر ظرفیت ۸ بود اگر بخوایم خانه ۹ را پر کنیم با خطا مواجه میشیم :

کد شماره ۱۴ https://play.golang.org/p/BtiZKrqyImq

```
package main

import (
        "fmt"
)

func main() {

        mySlice := make([]int, 0, 8)
```

```
        mySlice = append(mySlice, 1, 3, 5, 7, 9, 11, 13, 17)

        fmt.Printf("Contents: %v\n", mySlice)
        fmt.Printf("Number of Items: %d\n", len(mySlice))
        fmt.Printf("Capacity: %d\n", cap(mySlice))

        mySlice[8] = 19

        fmt.Printf("Contents: %v\n", mySlice)
        fmt.Printf("Number of Items: %d\n", len(mySlice))
        fmt.Printf("Capacity: %d\n", cap(mySlice))

}

/*
Output :

Contents: [1 3 5 7 9 11 13 17]
Number of Items: 8
Capacity: 8
panic: runtime error: index out of range

goroutine 1 [running]:
main.main()
        /home/amir/Desktop/Code/go/Go-Succinctly/listing22.go:18 +0x464
exit status 2
*/
```

اما همان طور که بیان شد یکی دیگر راه های افزودن ، تابع درونی append است که یکی از کارهای جادویی این تابع اینکه اگر خانه ای بیشتر از ظرفیت بسازید خطا نمیده و خودکار ظرفیت رو دو برابر قرار می دهد .

https://play.golang.org/p/yE-1Rhccebv

کد شماره ۱۵

```
package main

import (
        "fmt"
)

func main() {

        mySlice := make([]int, 0, 8)
        mySlice = append(mySlice, 1, 3, 5, 7, 9, 11, 13, 17)

        fmt.Printf("Contents: %v\n", mySlice)
        fmt.Printf("Number of Items: %d\n", len(mySlice))
        fmt.Printf("Capacity: %d\n", cap(mySlice))

        mySlice = append(mySlice, 19)

        fmt.Printf("Contents: %v\n", mySlice)
        fmt.Printf("Number of Items: %d\n", len(mySlice))
```

```
        fmt.Printf("Capacity: %d\n", cap(mySlice))

}

/*
Output :

Contents: [1 3 5 7 9 11 13 17]
Number of Items: 8
Capacity: 8
Contents: [1 3 5 7 9 11 13 17 19]
Number of Items: 9
Capacity: 16
*/
```

دیگر نگران ظرفیت برش ها نباشید !

## منابع بیشتر

Go Slices: usage and internals https://blog.golang.org/go-slices-usage-and-internals

## Maps در گولنگ

Map در واقع مجموعه ای از دیتا ها به صورت key-value هستش با ساختار `associative data type` همانند دیکشنری در پایتون .

https://play.golang.org/p/U67R66Oab8r کد شماره ۱۶

```
// _Maps_ are Go's built-in [associative data type]
(http://en.wikipedia.org/wiki/Associative_array)
// (sometimes called _hashes_ or _dicts_ in other languages).

package main

import "fmt"

func main() {

    // To create an empty map, use the builtin `make`:
    // `make(map[key-type]val-type)`.
    m := make(map[string]int)

    // Set key/value pairs using typical `name[key] = val`
    // syntax.
    m["k1"] = 7
    m["k2"] = 13

    // Printing a map with e.g. `fmt.Println` will show all of
    // its key/value pairs.
    fmt.Println("map:", m)

    // Get a value for a key with `name[key]`.
    v1 := m["k1"]
    fmt.Println("v1: ", v1)

    // The builtin `len` returns the number of key/value
    // pairs when called on a map.
    fmt.Println("len:", len(m))

    // The builtin `delete` removes key/value pairs from
    // a map.
    delete(m, "k2")
    fmt.Println("map:", m)

    // The optional second return value when getting a
    // value from a map indicates if the key was present
    // in the map. This can be used to disambiguate
    // between missing keys and keys with zero values
    // like `0` or `""`. Here we didn't need the value
    // itself, so we ignored it with the _blank identifier_
    // `_`.
    _, prs := m["k2"]
    fmt.Println("prs:", prs)
```

```
    // You can also declare and initialize a new map in
    // the same line with this syntax.
    n := map[string]int{"foo": 1, "bar": 2}
    fmt.Println("map:", n)
}
```

خروجی :

```
$ go run listing16.go
map: map[k1:7 k2:13]
v1:  7
len: 2
map: map[k1:7]
prs: false
map: map[foo:1 bar:2]
```

## نکات :

- همان طور که در نگاه اول مشخص میشه نحوه تعریف Map مشابه Slice هستش با این تفاوت که سینتکس آن به صورت `make(map[key-type]val-type)` تعریف می شود و در کد بالا داخل متغیر `m` پیاده سازی شده است .
- برای مقداردهی به یک کلید می توان از سینتکس `name[key] = val` و برای دریافت مقدار یک کلید می توان از `name[key]` استفاده نمود .
- برای چاپ می توان از `fmt.Println` استفاده کرد و خروجی همانند `map[k1:7 k2:13]` دریافت کرد همانند کد بالا .
- همانند `array` و `slice` تابع `len` برای `map` نیز قابل استفاده است تا بتوانید تعداد کلید ها رو بشمارید .
- تابع داخلی `delete` که شامل دو پارامتر هستش ( پارامتر اول دریافت `map` و پارامتر دوم کلید ) به ما این دسترسی را می دهد تا کلید خاصی را حذف کنیم .
- قبلا بیان شد که اگر نوع مقدار مثلا `int` قرار داده شد و مقداری مشخص نشد ( یا وجود نداشت ) برای ما `0` بر می گرده ! اما اگه ما واقعا `0` را خودمان به عنوان متغیر قرار داده باشیم و بخوایم بسنجیم ، از کجا میشه فهمید این `0` را ما گذاشتیم یا در صورت نبودن مقدار برگشت داده شده . بعضی جاها عبارتی به نام `comma OK` ذکر کردن که به این معناس وقتی با سینتکس `name[key]` شروع به دریافت مقدار می کنید در واقع دو خروجی وجود دارد و اون خروجی دوم به صورت `boolean` به ما میگه که این کلید مقدار دهی شده یا خیر . در مثال بالا نمونه ای ذکر شده است اما به جای `ok` از نام `prs` استفاده شده .
- در آخرین مثال کد بالا نحوه تعریف `map` و مقدار دهیش در یک خط مشخص شده است .
- شما می توانید با کلمه کلیدی `range` روی آرایه ها ، برش ها و `map` ها به کمک `for` پروسه `iterate` را انجام بدید اما نکته حایز اهمیتی که وجود دارد اینکه بر اساس طراحی زبان Go این خروجی بر روی `map` نظم نداره و هر دفعه به صورت راندوم نمایش داده می شود .

> This is by design: the Go team didn't want programmers to rely upon an ordering that was essentially unreliable, so they randomized the iteration order to make that impossible.

## مثال :

کد شماره ۱۷ https://play.golang.org/p/jYhjajBm7pQ

```
package main

import "fmt"
```

```
func main() {
        actor := map[string]int{
                "Paltrow": 43,
                "Cruise":  53,
                "Redford": 79,
                "Diaz":    43,
                "Kilmer":  56,
                "Pacino":  75,
                "Ryder":   44,
        }

        for i := 1; i < 4; i++ {
                fmt.Printf("\nRUN NUMBER %d\n", i)
                for key, value := range actor {
                        fmt.Printf("%s : %d years old\n", key, value)
                }
        }
}

/*
Output :

RUN NUMBER 1
Redford : 79 years old
Diaz : 43 years old
Kilmer : 56 years old
Pacino : 75 years old
Ryder : 44 years old
Paltrow : 43 years old
Cruise : 53 years old

RUN NUMBER 2
Paltrow : 43 years old
Cruise : 53 years old
Redford : 79 years old
Diaz : 43 years old
Kilmer : 56 years old
Pacino : 75 years old
Ryder : 44 years old

RUN NUMBER 3
Cruise : 53 years old
Redford : 79 years old
Diaz : 43 years old
Kilmer : 56 years old
Pacino : 75 years old
Ryder : 44 years old
Paltrow : 43 years old
*/
```

راه حل : بهترین راه حل تغییر ساختار هستش به این صورت که ابتدا اطلاعات را به ساختار برش ها انتقال بدیم و سپس با پکیج `sort` آن ها رو مرتبط کنیم .

https://play.golang.org/p/GwFoxWxhuaF

کد شماره ۱۸

```
package main

import (
        "fmt"
        "sort"
)

func main() {
        actor := map[string]int{
                "Paltrow": 43,
                "Cruise":  53,
                "Redford": 79,
                "Diaz":    43,
                "Kilmer":  56,
                "Pacino":  75,
                "Ryder":   44,
        }

        // Store the keys in a slice
        var sortedActor []string
        for key := range actor {
                sortedActor = append(sortedActor, key)
        }
        // Sort the slice alphabetically
        sort.Strings(sortedActor)

        /* Retrieve the keys from the slice and use
           them to look up the map values */
        for _, name := range sortedActor {
                fmt.Printf("%s : %d years old\n", name, actor[name])
        }
}

/*
Output :

Cruise : 53 years old
Diaz : 43 years old
Kilmer : 56 years old
Pacino : 75 years old
Paltrow : 43 years old
Redford : 79 years old
Ryder : 44 years old
*/
```

### ترفند :

داخل `if` ما می توانیم متغیری تعریف کنیم و سپس با `;` اون را جدا کنیم و به ارزشیابی بپردازیم . صرفا برای آشنایی با چنین سینتکس هایی بر روی
`if statement` ها مطرح شد .

https://play.golang.org/p/ueJmA28U5EF کد شماره ۱۹

```
package main

import "fmt"

func main() {

        m := make(map[string]int)

        m["k1"] = 7
        m["k2"] = 13

        delete(m, "k2")

        if _, ok := m["k2"]; ok {
                fmt.Println("Ok")
        } else {
                fmt.Println("NotOk")
        }
}
```

## Range در گولنگ

کلمه کلیدی `range` به ما اجازه `iterate` بر روی ساختار داده هایی که تاکنون آموختیم را می دهد .

https://play.golang.org/p/SkL_AS-1Jd

کد شماره ۲۰

```
// _range_ iterates over elements in a variety of data
// structures. Let's see how to use `range` with some
// of the data structures we've already learned.

package main

import "fmt"

func main() {

    // Here we use `range` to sum the numbers in a slice.
    // Arrays work like this too.
    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum)

    // `range` on arrays and slices provides both the
    // index and value for each entry. Above we didn't
    // need the index, so we ignored it with the
    // blank identifier `_`. Sometimes we actually want
    // the indexes though.
    for i, num := range nums {
        if num == 3 {
            fmt.Println("index:", i)
        }
    }

    // `range` on map iterates over key/value pairs.
    kvs := map[string]string{"a": "apple", "b": "banana"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }

    // `range` can also iterate over just the keys of a map.
    for k := range kvs {
        fmt.Println("key:", k)
    }

    // `range` on strings iterates over Unicode code
    // points. The first value is the starting byte index
    // of the `rune` and the second the `rune` itself.
    for i, c := range "go" {
        fmt.Println(i, c)
```

```
    }
}
```

خروجی :

```
$ go run listing20.go
sum: 9
index: 1
a -> apple
b -> banana
key: a
key: b
0 103
1 111
```

**نکات :**

- در مثال اول یعنی متغیر `nums` ، ما بر روی یک برش یا `slice` عمل `iterate` را انجام دادیم .
- در واقع کلمه کلیدی `range` دو خروجی دارد که در `array` و `slice` ها اولی `index` و دومی مقدار خانه است . در مثال اول ما متغیر `index` را با قرار دادن `blank identifier` یا همان _ گفتیم که استخراج نکن چون اگر استخراج کنیم و استفاده نکنیم کامپایلر گو خطا می دهد .
- در متغیر `kvs` ما یک `map` با مقدار اولیه ساختیم و آن را `iterate` کردیم اما تفاوت خروجی `range` در `map` این است که اولین پارامتر کلید و دومین پارامتر مقدار است .
- برای `map` ها می توانیم فقط یک خروجی تحت عنوان کلید از `range` بگیریم .
- به کمک `range` همچنین می توانیم بر روی رشته ها `iterate` کنیم ، در اینجا پارامتر اول ایندکس و پارامتر دوم Unicode code points را بر می گردونه !

## Functions در گولنگ

توابع یکی از قابلیت های مهم هر زبانی محسوب می شود و در گولنگ دارای سینتکس بسیار ساده ای هستند به مثال های زیر توجه کنید .

https://play.golang.org/p/JpNfVtAjrc4 کد شماره ۲۱

```
// _Functions_ are central in Go. We'll learn about
// functions with a few different examples.

package main

import "fmt"

// Here's a function that takes two `int`s and returns
// their sum as an `int`.
func plus(a int, b int) int {

    // Go requires explicit returns, i.e. it won't
    // automatically return the value of the last
    // expression.
    return a + b
}

// When you have multiple consecutive parameters of
// the same type, you may omit the type name for the
// like-typed parameters up to the final parameter that
// declares the type.
func plusPlus(a, b, c int) int {
    return a + b + c
}

func main() {

    // Call a function just as you'd expect, with
    // `name(args)`.
    res := plus(1, 2)
    fmt.Println("1+2 =", res)

    res = plusPlus(1, 2, 3)
    fmt.Println("1+2+3 =", res)
}
```

خروجی :

```
$ go run listing21.go
1+2 = 3
1+2+3 = 6
```

### نکات :

- برای تعریف یک تابع ساده به الگو پیاده سازی تابع `plus` توجه کنید ، همیشه باید دیتا تایپ پارامترهای ورودی مشخص شوند در این مثال ما دو پارامتر `a` و `b` را با تایپ `int` به تابع منتقل می کنیم .
- بعد از تعیین پارامتر های ورودی و قبل از قرار دادن کد داخل براکت برای تابع کلید واژه `int` قرار گرفته به این معنا که مقدار خروجی تابع باید دیتا تایپ `int` باشد .
- در صورتی که دیتا تایپ تمام پارامترهای ورودی یکسان باشد می توان همانند تابع `plusPlus` کلید واژه دیتا تایپ را در آخر همه پارامترها ذکر کرد .
- برای صدا زدن یک تابع کافیست از سینتکس `name(args)` استفاده کرد .
- توابع در گولنگ ویژگی های زیادی دارند که در آموزش های بعدی معرفی خواهند شد .

### الگو دیگر تعریف تابع :

کد شماره ۲۲ https://play.golang.org/p/bOG8oBahGnm

```
package main

import "fmt"

func main() {

        plus := func(a int, b int) int {

                return a + b
        }

        res := plus(1, 2)
        fmt.Println("1+2 =", res)
}
```

خروجی :

```
$ go run listing22.go
1+2 = 3
```

### نکات :

- به هیچ عنوان نمی شود داخل یک تابع ، تابع دیگری ساخت مگر با استفاده از این سینتکس .
- در استفاده از این سینتکس توجه داشته باشید برای `Recursion` نمی توانید از این قابلیت استفاده کنید ، همانند زبان های `Javascript` و `Typescript` از نام دوم نیز نمی توان استفاده کرد .

## Multiple Return Values در گولنگ

زبان برنامه نویسی گو به ما اجازه می دهد تا هر تابع چند خروجی داشته باشد برای مثال ما می توانیم خروجی تابع و خطا را همزمان از تابع بگیریم و مثل بعضی زبان ها نیاز نیست یک خروجی وجود داشته باشد و ما بسنجیم آیا خروجی خطا هست یا نه !

کد شماره ۲۳ https://play.golang.org/p/chwFmr5dG1

```
// Go has built-in support for _multiple return values_.
// This feature is used often in idiomatic Go, for example
// to return both result and error values from a function.

package main

import "fmt"

// The `(int, int)` in this function signature shows that
// the function returns 2 `int`s.
func vals() (int, int) {
    return 3, 7
}

func main() {

    // Here we use the 2 different return values from the
    // call with _multiple assignment_.
    a, b := vals()
    fmt.Println(a)
    fmt.Println(b)

    // If you only want a subset of the returned values,
    // use the blank identifier `_`.
    _, c := vals()
    fmt.Println(c)
}
```

خروجی :

```
$ go run listing23.go
3
7
7
```

### نکات :

- در کد بالا تابعی با عنوان `vals` ساختیم اما این تابع دو خروجی دارد بنابراین نحوه تعریف نوع داده خروجی متفاوت است نسبت به یک تابع معمولی و با سینتکس `(data-type, data-type, ...)` تعریف می شود ، همچنین خروجی آن را با `,` باید از هم جدا کرد .
- در مثال اول ما به کمک دو متغیر `a` و `b` دو خروجی تابع `vals` را دریافت کردیم .

- در مثال بعدی ما فقط خروجی دوم را با متغیر `c` استخراج کردیم و بجای خروجی اول از blank identifier `_` استفاده کردیم تا مقدار خروجی اول دریافت یا استخراج نشود .

## Variadic Functions در گولنگ

یکی از جذاب ترین قابلیت های گو وجود Variadic Function (توابع متنوع) هاست که به شما اجازه می دهد هر چقدر پارامتر نیاز دارید بدون تعریف وارد تابع کنید همانند Rest-parameter ها در es6 . معمول ترین Variadic Function در گو تابع `fmt.Println` می باشد .

کد شماره ۲۴ https://play.golang.org/p/7f0JlVhToDD

```
// [_Variadic functions_](http://en.wikipedia.org/wiki/Variadic_function)
// can be called with any number of trailing arguments.
// For example, `fmt.Println` is a common variadic
// function.

package main

import "fmt"

// Here's a function that will take an arbitrary number
// of `int`s as arguments.
func sum(nums ...int) {
    fmt.Print(nums, " ")
    total := 0
    for _, num := range nums {
        total += num
    }
    fmt.Println(total)
}

func main() {

    // Variadic functions can be called in the usual way
    // with individual arguments.
    sum(1, 2)
    sum(1, 2, 3)

    // If you already have multiple args in a slice,
    // apply them to a variadic function using
    // `func(slice...)` like this.
    nums := []int{1, 2, 3, 4}
    sum(nums...)
}
```

خروجی :

```
$ go run listing24.go
[1 2] 3
[1 2 3] 6
[1 2 3 4] 10
```

## نکات

- در مثال بالا تابع `sum` به کمک `...` که قبل نوع داده قرار گرفته به ما اجازه می دهد چندین پارامتر را به داخل تابع انتقال دهیم .
- برای صدا زدن این توابع می تواند به صورت `funcName(param1, param2, ...)` عمل کرد .
- اگر شما داده ای با ساختار `slice` دارید به راحتی به کمک `operator ...` می توانید با سینتکس `funcName(mySlice...)` آن را به تابع انتقال دهید .

## Closures در گولنگ

به دلیل اینکه Closures در همه زبان های برنامه نویسی وجود داره قبل از شروع مطالعه بخشی از ویکی پدیا خالی از لطف نیست .

> **Closure (computer programming)**
>
> The following program fragment defines a higher-order function (function returning a function) add with a parameter x and a nested function addX. The nested function addX has access to x, because x is in the lexical scope of addX. The function add returns a closure; this closure contains (1) a reference to the function addX, and (2) a copy of the environment around addX in which x has the value given in that specific invocation of add.
>
> ```
> function add(x)
>    function addX(y)
>        return y + x
>    return addX
>
> variable add1 = add(1)
> variable add5 = add(5)
>
> assert add1(3) = 4
> assert add5(3) = 8
> ```
>
> https://en.wikipedia.org/wiki/Closure_(computer_programming)

زبان گو از توابع ناشناس anonymous functions در فرم closures بهره می برد . توابع ناشناس این امکان را می دهند تا تابعی بدون نام به صورت inline بسازیم .

https://play.golang.org/p/zb93qzV6iN3 کد شماره ۲۵

```
// Go supports [_anonymous functions_](http://en.wikipedia.org/wiki/Anonymous_function),
// which can form <a href="http://en.wikipedia.org/wiki/Closure_(computer_science)">
<em>closures</em></a>.
// Anonymous functions are useful when you want to define
// a function inline without having to name it.

package main

import "fmt"

// This function `intSeq` returns another function, which
// we define anonymously in the body of `intSeq`. The
// returned function _closes over_ the variable `i` to
// form a closure.
func intSeq() func() int {
    i := 0
    return func() int {
        i++
        return i
    }
```

```
}

func main() {

    // We call `intSeq`, assigning the result (a function)
    // to `nextInt`. This function value captures its
    // own `i` value, which will be updated each time
    // we call `nextInt`.
    nextInt := intSeq()

    // See the effect of the closure by calling `nextInt`
    // a few times.
    fmt.Println(nextInt())
    fmt.Println(nextInt())
    fmt.Println(nextInt())

    // To confirm that the state is unique to that
    // particular function, create and test a new one.
    newInts := intSeq()
    fmt.Println(newInts())
}
```

خروجی :

```
$ go run listing25.go
1
2
3
1
```

## نکات

- ما با تعریف یک تابع و نوع داده خروجیش آشنا شدیم اما اگر خروجی تابع یک تابع دیگر بود در این حالت باید از سینتکس `func() data-type` استفاده کنیم ، همانند تابع `intSeq` . در اینجا `data-type` خروجی ، خروجی تابع درونیست .
- در تابع `intSeq` خروجی ما یک تابع دیگر است و چون تابع درونی به متغیر `i` که در تابع بالایی خودش وجود داره مرتبط است میشه بیان داشت خروجی تابع `intSeq` یک فرم از `closure` است .
- داخل تابع `main` متغیری تحت عنوان `nextInt` تعریف کردیم و داخل آن تابع `intSeq` را قرار دادیم حالا هر چقدر ما `nectInt` را صدا یا `invoke` کنیم `state` ما یا همون متغیر `i` درون تابع `intSeq` بروز می شود .
- برای اینکه بفهمیم این `state` مخصوص تابع اصلیس دوباره آن را تحت عنوان `newInts` ساختیم و `invoke` کردیم .

## Recursion در گولنگ

همه برنامه نویسان به اصطلاحاتی چون `Recursion` در علوم کامپیوتر آشنا هستند و گولنگ نیز از این قابلیت پشتیبانی می کند و در زیر مثال کلاسیک فاکتوریل را داریم .

کد شماره ۲۶ https://play.golang.org/p/RFn-rf42ap

```
// Go supports
// <a href="http://en.wikipedia.org/wiki/Recursion_(computer_science)"><em>recursive
functions</em></a>.
// Here's a classic factorial example.

package main

import "fmt"

// This `fact` function calls itself until it reaches the
// base case of `fact(0)`.
func fact(n int) int {
    if n == 0 {
        return 1
    }
    return n * fact(n-1)
}

func main() {
    fmt.Println(fact(7))
}
```

خروجی :

```
$ go run listing26.go
5040
```

### نکات :

- وقتی توابع از درون ، خودشون را صدا می زنن به اصطلاح مبحث `recursion` (بازگشتی) به وجود می آید .
- در مثال بالا تا زمانی که `n-1` مساوی صفر نشود این تابع خودش را صدا می زند .

## Pointers در گولنگ

> In computing, a memory address is a reference to a specific memory location used at various levels by software and hardware. Memory addresses are fixed-length sequences of digits conventionally displayed and manipulated as unsigned integers. Such numerical semantic bases itself upon features of CPU (such as the instruction pointer and incremental address registers), as well upon use of the memory like an array endorsed by various programming languages.
> https://en.wikipedia.org/wiki/Memory_address

اگر تا حالا با زبان های c و c++ کار نکردید احتمالا با پوینترها آشنا نیستید . پوینتر برای انتقال آدرس حافظه memory location استفاده می شود .

کد شماره ۲۷ https://play.golang.org/p/KdE4TBbUL2

```
// Go supports <em><a
href="http://en.wikipedia.org/wiki/Pointer_(computer_programming)">pointers</a></em>,
// allowing you to pass references to values and records
// within your program.

package main

import "fmt"

// We'll show how pointers work in contrast to values with
// 2 functions: `zeroval` and `zeroptr`. `zeroval` has an
// `int` parameter, so arguments will be passed to it by
// value. `zeroval` will get a copy of `ival` distinct
// from the one in the calling function.
func zeroval(ival int) {
    ival = 0
}

// `zeroptr` in contrast has an `*int` parameter, meaning
// that it takes an `int` pointer. The `*iptr` code in the
// function body then _dereferences_ the pointer from its
// memory address to the current value at that address.
// Assigning a value to a dereferenced pointer changes the
// value at the referenced address.
func zeroptr(iptr *int) {
    *iptr = 0
}

func main() {
    i := 1
    fmt.Println("initial:", i)

    zeroval(i)
    fmt.Println("zeroval:", i)

    // The `&i` syntax gives the memory address of `i`,
    // i.e. a pointer to `i`.
```

```
    zeroptr(&i)
    fmt.Println("zeroptr:", i)

    // Pointers can be printed too.
    fmt.Println("pointer:", &i)
}
```

خروجی :

```
$ go run listing27.go
initial: 1
zeroval: 1
zeroptr: 0
pointer: 0x42131100
```

**نکات :**

- برای ایجاد پوینتر از کارکتر `&` استفاده می شود و در مثال بالا دو تابع داریم اولی `zeroval` که یک تابع معمولی است و تابع `zeroptr` که کنار نوع داده ورودی علامت `*` مشاهده می کنید به این معنا که ورودی به کمک پوینتر رفرنس داده شده است و در ادامه هنگام مقدار دهی نیز از `*` استفاده شده تا مقدار در آدرس حافظه ای که رفرنس داده شده ذخیره شود .
- در تابع `main` با مثال پوینتر پیاده سازی شده و قابل درک است و در آخرین خط مشاهده می کنیم که آدرس حافظه پوینتر قابل چاپ است .

## Structs در گولنگ

یکی از ساختارهای کاربردی در گولنگ `Struct` هاست ، این ساختار اجازه می دهد تا مجموعه از داده ها را به صورت کالکشن داشته باشیم چیزی شبیه به `object` در زبان های دیگر .

کد شماره ۲۸ https://play.golang.org/p/OMCP5KFC10

```
// Go's _structs_ are typed collections of fields.
// They're useful for grouping data together to form
// records.

package main

import "fmt"

// This `person` struct type has `name` and `age` fields.
type person struct {
    name string
    age  int
}

func main() {

    // This syntax creates a new struct.
    fmt.Println(person{"Bob", 20})

    // You can name the fields when initializing a struct.
    fmt.Println(person{name: "Alice", age: 30})

    // Omitted fields will be zero-valued.
    fmt.Println(person{name: "Fred"})

    // An `&` prefix yields a pointer to the struct.
    fmt.Println(&person{name: "Ann", age: 40})

    // Access struct fields with a dot.
    s := person{name: "Sean", age: 50}
    fmt.Println(s.name)

    // You can also use dots with struct pointers - the
    // pointers are automatically dereferenced.
    sp := &s
    fmt.Println(sp.age)

    // Structs are mutable.
    sp.age = 51
    fmt.Println(sp.age)
}
```

خروجی :

```
$ go run listing28.go
{Bob 20}
{Alice 30}
{Fred 0}
&{Ann 40}
Sean
50
51
```

## نکات

- برای ساخت یک `struct` از کلمه کلیدی `type` به صورت `type StructName struct {}` استفاده می کنیم .
- برای ساخت یک instance از ساختار `struct` دو الگو بلند و کوتاه وجود دارد که الگو بلند مقدار دهی با کلید است و الگو کوتاه آن مقداردهی با توجه به کلید در خود ساختار است .

```
    // Short way
    fmt.Println(person{"Bob", 20})

    // Long way
    fmt.Println(person{name: "Alice", age: 30})
```

- اگر در ساخت یک instance برای کلید مقداری قرار داده نشود با توجه به نوع داده مقدار ثابتی می گیرد .
- برای قرار دادن پوینتر کافیست از اپراتور `&` قبل شروع تعریف آن استفاده کنیم .
- برای خواندن و مقدار دهی می توانید از `.` به صورت `yourStruct.paramOne = sth` استفاده کنید .
- `struct` ساختاری قابل تغییر است .
- اگر نیاز شد `instance` ی از این ساختار بسازیم و پوینتر بهش وصل کنیم سینتکس کوتاهی برای آن وجود دارد :

```
var MyPerson *person
MyPerson = new(person)

// Short way
MyPerson := new(person)
```

## Methods در گولنگ

methods ها توابعی هستند که بر روی ساختار struct ساخته می شوند برای مثال فکر کنید ساختار struct یک object در جاوا اسکریپت هستش و ما می خوایم یک prototype که تابع هست به اون اضافه کنیم همین ! این قابلیت بوجود اومد زیرا همیشه لازم میشه داده های داخل ساختار struct را با توابعی تغییر داد .

کد شماره ۲۹ https://play.golang.org/p/254m_9Yjwa

```
// Go supports _methods_ defined on struct types.

package main

import "fmt"

type rect struct {
    width, height int
}

// This `area` method has a _receiver type_ of `*rect`.
func (r *rect) area() int {
    return r.width * r.height
}

// Methods can be defined for either pointer or value
// receiver types. Here's an example of a value receiver.
func (r rect) perim() int {
    return 2*r.width + 2*r.height
}

func main() {
    r := rect{width: 10, height: 5}

    // Here we call the 2 methods defined for our struct.
    fmt.Println("area: ", r.area())
    fmt.Println("perim:", r.perim())

    // Go automatically handles conversion between values
    // and pointers for method calls. You may want to use
    // a pointer receiver type to avoid copying on method
    // calls or to allow the method to mutate the
    // receiving struct.
    rp := &r
    fmt.Println("area: ", rp.area())
    fmt.Println("perim:", rp.perim())
}
```

خروجی :

```
$ go run listing29.go
area:  50
perim: 30
area:  50
perim: 30
```

**نکات :**

- در کد بالا دو `method` با نام های `area` و `perim` برای ساختار `rect` با سینتکس `func (AnyVarible StructName) MethodName() Data-Type` ساخته شده است .

```
// Go automatically handles conversion between values
// and pointers for method calls. You may want to use
// a pointer receiver type to avoid copying on method
// calls or to allow the method to mutate the
// receiving struct.
```

## Embedded types در گولنگ

حالا که با ساختار `struct` و `method` های آن آشنا شدید باید بدانیم هر `type` از داده می تواند `type` داده دیگری را بسازد اما اون `type` ناشناخته خواهد بود ، به این معنا که نمی توانید اسمی به آن بدهید بجاش شما می توانید اونو با نام `type` آخر صدا کنید .
برای مثال در زیر `type` داده ای برای کد تخفیف پیاده کردیم :

```
type Discount struct {
  percent     float32
  promotionId string
}
```

حالا در زیر `type` ی برای یک تخفیف ویژه + تخفیف اولیه میسازیم :

```
type ManagersSpecial struct {
  Discount // The embedded type
  extraoff float32
}
```

برای استفاده از نوع داده بالا می توانیم به شکل زیر عمل کنیم .

```
januarySale := Discount{15.00, "January"}
managerSpecial := ManagersSpecials{januarySale, 10.00}
```

همچنین برای دسترسی به نوع داده `embed` شده میشه از روش زیر استفاده کرد :

```
managerSpecial.Discount.percent // 15.00
managerSpecial.Discount.promotionId // "January"
```

اگر ما `method`ی به `Discount` متصل کنیم برای صدا زدن آن باید از سینتکس زیر استفاده کرد :

```
managerSpecial.Discount.someMethod(someParameter)
```

## مثال عملی

کد شماره ۳۰ https://play.golang.org/p/ufYS79Dx4BO

```
package main

import (
        "fmt"
)

type Discount struct {
```

```
        percent     float32
        promotionId string
}

type ManagersSpecial struct {
        Discount
        extraoff float32
}

func main() {

        normalPrice := float32(99.99)

        januarySale := Discount{15.00, "January"}
        managerSpecial := ManagersSpecial{januarySale, 10.00}

        discountedPrice := januarySale.Calculate(normalPrice)
        managerDiscount := managerSpecial.Calculate(normalPrice)

        fmt.Printf("Original price: $%4.2f\n", normalPrice)
        fmt.Printf("Discount percentage: %2.2f\n",
                januarySale.percent)
        fmt.Printf("Discounted price: $%4.2f\n", discountedPrice)
        fmt.Printf("Manager's special: $%4.2f\n", managerDiscount)
}

func (d Discount) Calculate(originalPrice float32) float32 {
        return originalPrice - (originalPrice / 100 * d.percent)
}

func (ms ManagersSpecial) Calculate(originalPrice float32) float32 {
        return ms.Discount.Calculate(originalPrice) - ms.extraoff
}
```

## Interfaces در گولنگ

`interface` همان طوری که از اسمش پیداست کارش اینکه مطمن بشه یکسری دیتا ها وجود داشته باشه و نحوه تعریف آن مشابه `struct` است . در مثال پایین روی دو ساختار `rect` و `circle` ما دو متد `area` و `perim` ساختیم و داخل تابع `measure` ما نوع داده ورودی برای پارامتر `g` را اینترفیس `geometry` قرار دادیم و با این کار به تابع میگیم اگر ساختار `g` از این اینترفیس پیروی نکرد یعنی دو تابع `area` و `perim` نداشت اون موقع خطا بده !

کد شماره ۳۱ https://play.golang.org/p/313UebA3rD

```
// _Interfaces_ are named collections of method
// signatures.

package main

import "fmt"
import "math"

// Here's a basic interface for geometric shapes.
type geometry interface {
    area() float64
    perim() float64
}

// For our example we'll implement this interface on
// `rect` and `circle` types.
type rect struct {
    width, height float64
}
type circle struct {
    radius float64
}

// To implement an interface in Go, we just need to
// implement all the methods in the interface. Here we
// implement `geometry` on `rect`s.
func (r rect) area() float64 {
    return r.width * r.height
}
func (r rect) perim() float64 {
    return 2*r.width + 2*r.height
}

// The implementation for `circle`s.
func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}
func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}

// If a variable has an interface type, then we can call
// methods that are in the named interface. Here's a
```

```
// generic `measure` function taking advantage of this
// to work on any `geometry`.
func measure(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}

func main() {
    r := rect{width: 3, height: 4}
    c := circle{radius: 5}

    // The `circle` and `rect` struct types both
    // implement the `geometry` interface so we can use
    // instances of
    // these structs as arguments to `measure`.
    measure(r)
    measure(c)
}
```

خروجی :

```
$ go run listing31.go
{3 4}
12
14
{5}
78.53981633974483
31.41592653589793
```

## Empty interface در گولنگ

کد زیر را میگن `empty interface` :

```
interface {}
```

برخی زبان های دیگر مثل .Net و Java بهش `marker interface` می گن . در نگاه اول احمقانه به نظر میاد ولی خیلی کاربردیست . همان طور که می دونیم کامپایلر گولنگ به خاطر اینکه زبان `static-type` هست خیلی حساس عمل می کند و اگر نوع داده مشخص نشده باشد ما با خطا مواجه می شویم اما گاهی هنگام اجرا برنامه ما نمی دونیم چه نوع داده ای دریافت می کنیم پس اینترفیس خالی به ما کمک می کند این مشکل رو حل کنیم .

### مثال :

کد شماره ۳۲ https://play.golang.org/p/7uw26q10mbY

```
package main

import (
        "fmt"
)

func main() {
        displayType(42)
        displayType(3.14)
        displayType("ここでは文字列です")
}

func displayType(i interface{}) {
        switch theType := i.(type) {
        case int:
                fmt.Printf("%d is an §§§§§§§§§§integer\n", theType)
        case float64:
                fmt.Printf("%f is a 64-bit float\n", theType)
        case string:
                fmt.Printf("%s is a string\n", theType)
        default:
                fmt.Printf("I don't know what %v is\n", theType)
        }
}
```

## Type assertion در گولنگ

اگر از `empty interface` برای دسترسی به مقداری که نوع آن نامعلوم هست استفاده کردیم حتما برای تعیین نوع داده باید از این قابلیت استفاده کنیم . در واقع شما نمی توانید داده ای که نوع آن معلوم نیست را به راحتی استفاده کنید و باید به جاش نوع داده رو برای آن با این روش مشخص کنید .

```
var anything interface{} = "something"
aString := anything.(string)
```

کد شماره ۳۳ https://play.golang.org/p/njiry2LJ_qk

```
package main

import (
        "fmt"
)

func main() {
        var anything interface{} = "something"
        aString := anything.(string)
        fmt.Println(aString)
}

// Output :
// something
```

اما اگر نوع داده را اشتباه وارد کنیم با `panic` رو به رو میشویم !

کد شماره ۳۴ https://play.golang.org/p/9KT-MkAf3UJ

```
package main

import (
        "fmt"
)

func main() {
        var anything interface{} = "something"
        aInt := anything.(int)
        fmt.Println(aInt)
}

/*
Output :

panic: interface conversion: interface is string, not int
goroutine 1 [running]:
panic(0xda640, 0xc820010180)
    /usr/local/go/src/runtime/panic.go:464 +0x3e6
main.main()
    .../src/hello/main.go:9 +0xa8
```

```
exit status 2
*/
```

خوب اما حالا برای مدیریت بهتر خطایی که ممکنه بوجود بیاد باید توجه داشته باشیم این تابع دو خروجی دارد . خروجی اول ، داده تبدیل شده است و خروجی دوم متغیری که با نوع `bool` به ما نشان می دهد که آیا نوع داده درست انتخاب شده است یا نه .

کد شماره ۳۵ https://play.golang.org/p/IcR_JPZvsSM

```
package main

import (
        "fmt"
)

func main() {
        var anything interface{} = "something"
        aInt, ok := anything.(int)
        if !ok {
                fmt.Println("Cannot turn input into an integer")
        } else {
                fmt.Println(aInt)
        }
}

// Output :
// Cannot turn input into an integer
```

## منابع کتاب


- Book introduction by Mahdi Musavi https://hitos.ir/
- Go by Example https://gobyexample.com
- Go Succinctly https://www.syncfusion.com/ebooks/go_succinctly
- Go in Action https://www.manning.com/books/go-in-action


نگارش ۱.۰.۲
تیر ۱۳۹۷