

Golang

简介

本电子书全部内容整理与公众号TechFlow

本书涵盖了golang的基本入门知识，包括goroutine、channel、CSP结构等内容。本书内容可以随意转载，但请注明出处。



GoLang——Hello World，打开新世界的大门

今天是**Go语言系列**的第一篇文章，我们来聊聊这门新的语言和它的基础语法。

浅谈Golang

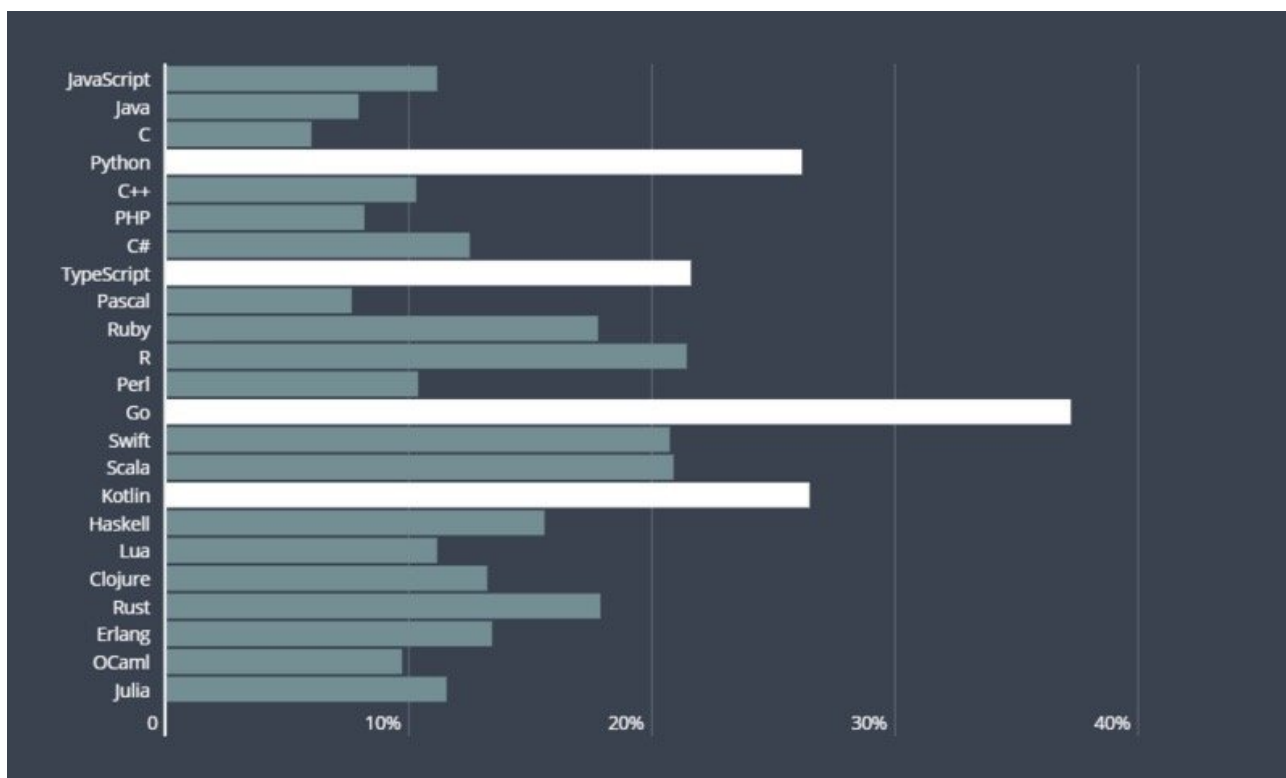
作为程序员而言，**往往对于学习新的语言都是有抗拒的**。如果你用惯了Java，那么你可能不太愿意去学Python，如果你刚学C或者C++可能你也会看不上Java。因为这个原因还会引发很多口水仗，这很正常。我当时写C++的时候也看不上Java，写了Java又不想学Python，现在学会了Python，有时候也会不想看其他语言写的代码。

但是随着我们的成长和实力的提升，我逐渐发现学习一门语言的**成本在飞快地下降**。毕竟天下语言都是程序，就好像武侠小说里各门各派的武功眼花缭乱，但是归结起来无非是拳脚功夫、刀剑或者是内功这么几种。语言也是一样，虽然我会的语言也不够多，远远没达到可以指点江山的地步。但也发现了很多语言之间的相关性实在是很强，有些理念一脉相承，有些更像是一个模子里刻出来的。既然如此，我们为什么需要学习Go这样一门语言呢？

网上相关的信息很多，很多大牛架构师高谈阔论。很多观点和看法我并没有很深的体会，所以我就不做搬运工了。简单说下我个人的一点**浅薄的看法**。

当初学习Golang这门语言的原因很简单，纯粹是因为工作需要。当前的公司几乎所有的系统都是以Golang写的，所以学习Golang是必须的。但是**学会语言基本的使用和精通一门语言这是两回事**，学会基本的用法是因为工作需要，而让我想要花时间把这门语言精通是因为一门课程。这门课程非常著名，它是MIT（麻省理工）的分布式系统的公开课，当中的内容和课后作业都非常硬核，更关键的是课后作业是用Golang写的。

在学习这门课程的过程当中，经过了一些思考和一些观察，Golang的确**在分布式处理的场景和问题当中有一定的优势**，许多优秀的解决方案都是基于Golang写的。并且这两年各类语言的流行变迁情况也印证了这一点，在最新的全世界程序员最想要学习的语言当中，Golang排名前3。在全球语言流行排行当中，Golang也冲进了前十，要知道这门语言才诞生十年。



并且在国内Golang工程师的需求量也与日俱增，我纯属好奇去拉勾网看了一下，全国Golang的岗位是341个。



想去互联网好公司，就上拉勾

868691家公司 | 6953406个职位，在拉勾等你

职位 (341)

公司 (0)

相关搜索: node.js php ruby c python 后端 javascript golang后端 全栈

看起来似乎不多，但是我们再来看下就业大户Java，也才500+



想去互联网好公司，就上拉勾

868691家公司 | 6953406个职位，在拉勾等你

职位 (500+) 公司 (0)

Java

相关搜索: java后端 java web java实习 java后端实习

要知道，Java几乎是CS必学的语言，所有CS毕业以及相关专业毕业的学生都可以号称自己会Java。但是Go不一样，它太新了，诞生不过十年，以目前高校的反应速度以及筛选课程的能力，它入选中国的教材目测最少还需要5-10年。并且和Java相比，Golang要好学的多。所以如果你还没有毕业，想要成为一名工程师，想要找一份工作，那可以考虑学习一下Go，说不定竞争压力会比Java小得多。

我个人觉得Golang是一门非常有个性的语言，长处与短处都非常的明显。关于它的长处很多，相比于一一列举出来，在实际学习和运用的过程当中领会到的感受会深得多。

Hello World

学语言第一件事就是敲Hello World，我想所有语言都不例外，我们也来看下Golang的Hello World，来看下它的基本结构。

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     fmt.Println("Hello World")
9 }
```

我们来简单看下这段代码，可以把这段代码分成三个部分。

```

code_for_article > go lang > go main.go > ...
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8      fmt.Println("Hello World")
9  }

```

当前包

引入包

main 函数

最上面的是模块名，也可以说成是包名，然后是引入包的语句。这一块其实没什么好说的，很多语言都是这样的结果，比如Java和Python。在Golang当中**main package**表示一个**独立的程序**，而不是一个包。在main package下的main函数代表这个独立程序的执行入口，和C++以及Java当中的main函数比较类似。我们可以在main函数里调用其他包的各种函数。

fmt是Golang当中的**标准输出包**，我们调用它来输出我们想打印的东西。

写完了Hello World之后就要执行了，Golang提供了两种执行方式，一种是直接go run + 要执行的文件名。还有一种是和C++一样先进行编译，再通过./调用编译之后的二进制包。

当然执行之前我们需要在电脑上配置Golang的环境，这方面网上的资料很多，并且Golang的安装也比较简单，基本上没有什么坑，所以照着网上的博客安装就好了。

我们来分别看下这两种调用方式，第一种我们直接使用go run执行代码：

```

mac ~/Documents go lang go run main.go
Hello World

```

还有一种方式是我们先通过go build对Golang的代码**先进行编译**，会生成一个二进制文件，之后我们直接./运行这个文件。

```

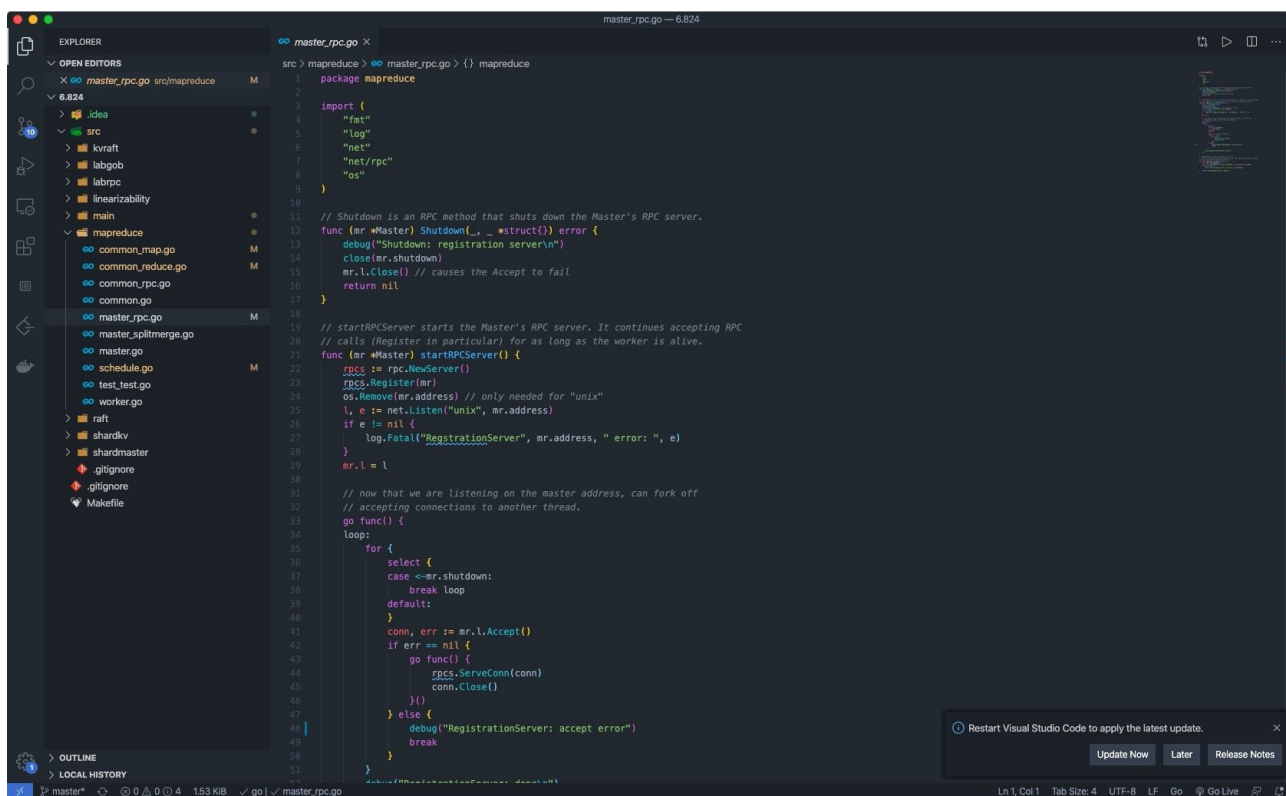
mac ~/Documents go lang go build main.go
mac ~/Documents go lang ./main
Hello World

```

所以从这里我们可以看出来，**Golang是编译型语言而不是解释性语言**，因此它的效率会非常高，实际上由于Golang的一些底层设计和特性，Golang的运行效率非常高，在绝大多数场景比Java更快，仅仅次于C++。

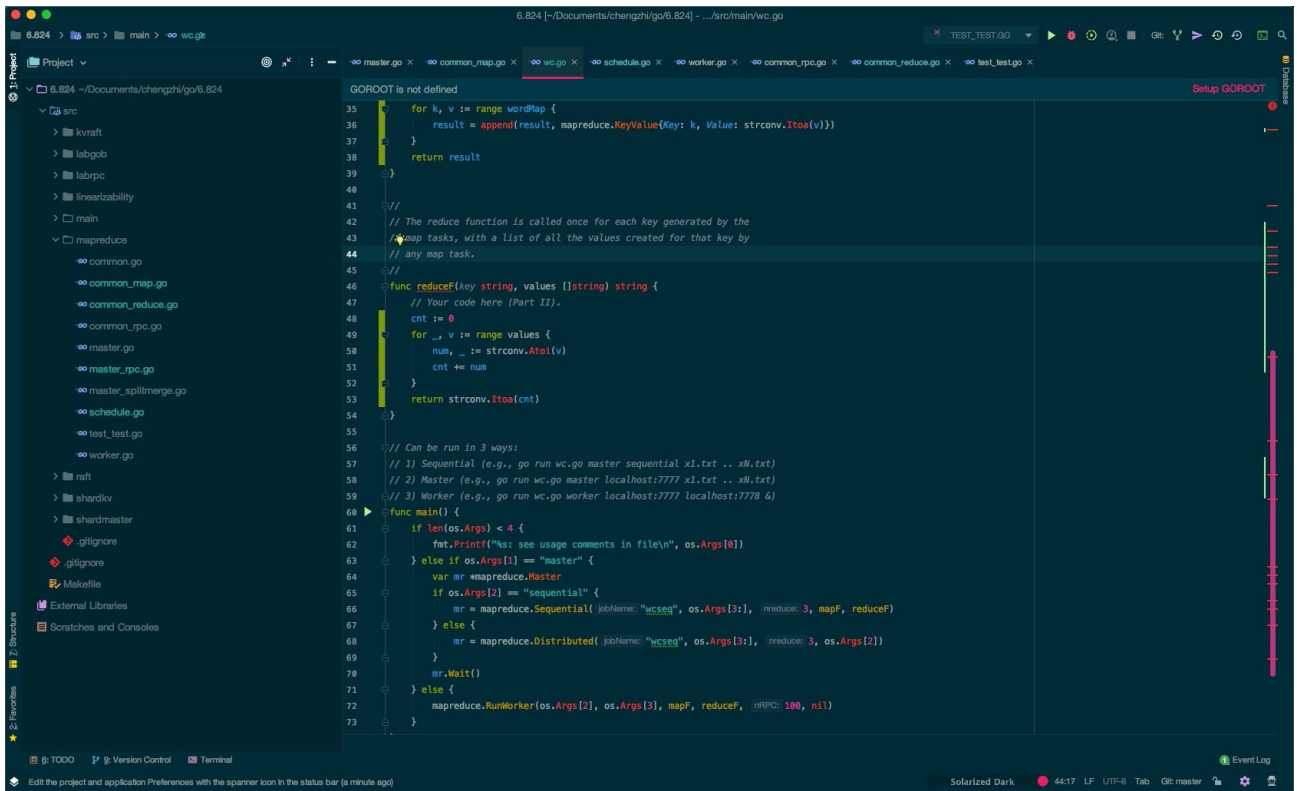
IDE

最后简单聊聊Golang的开发环境，其实现在开发环境已经普世化了，很多代码编辑器可以用来写各种语言。比如业内比较流行的**vscode**，**Atom**，**Sublime**等等，这几种当中我个人最喜欢vscode，功能非常强大，拥有海量的插件支持，并且页面风格和使用体验也不错，并且还是免费的。

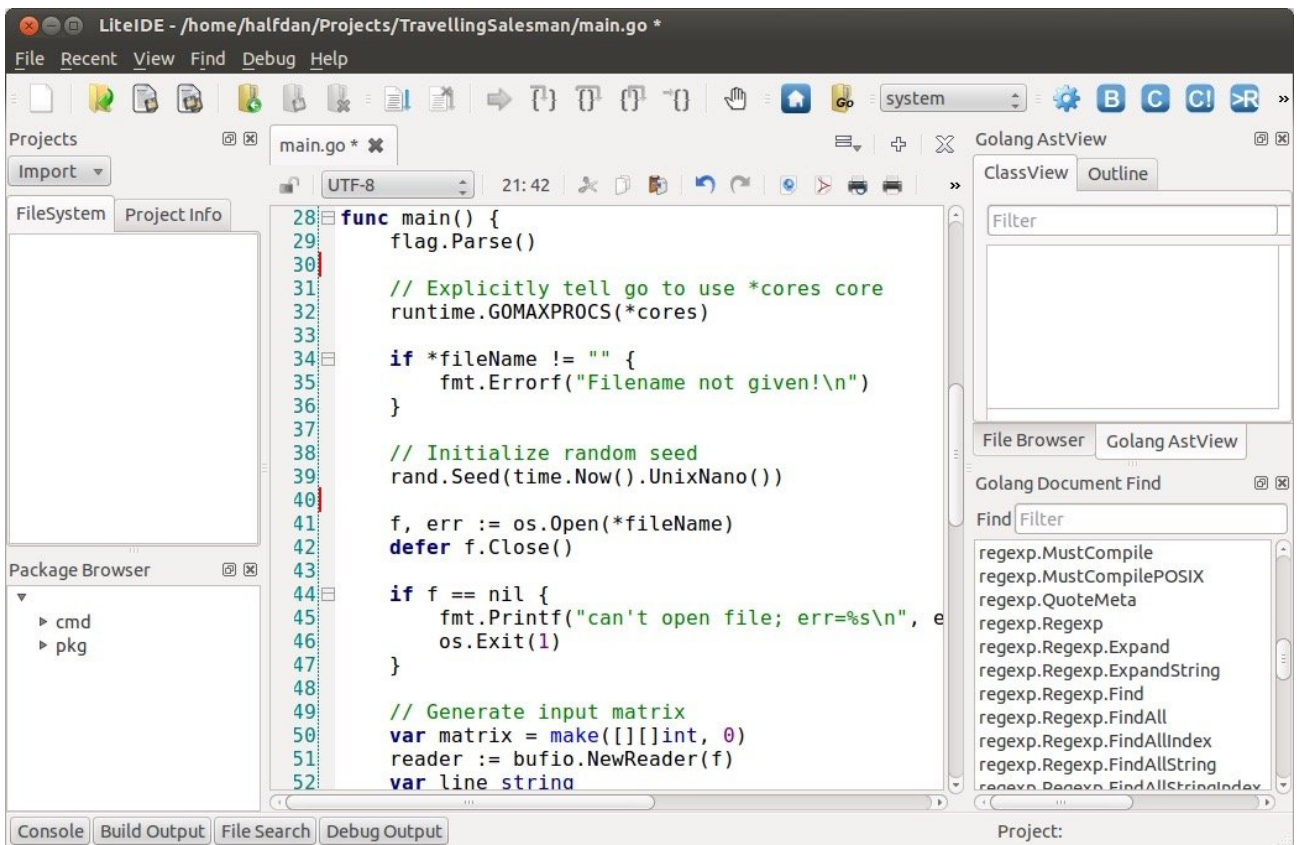


另外一个比较常用的是Goland，它是**Jet Brains**公司专门为Golang开发的IDE。Jet Brains公司专门开发各种语言的IDE，几乎每一种语言的IDE都广受欢迎。写过Java的同学应该对IDEA都不陌生，它的体验比eclipse要好用得多。同样，Goland对于Golang的支持也非常好，使用体验非常棒，而且如果之前用过它家的其他产品会非常适应。

Goland页面风格以及各方面体验都非常棒，但缺点也很明显，一个是**基本上只支持Golang**，另一个缺点就是**贵**。免费的社区版要阉割掉一些功能，而专业版则要好几千人民币。不过如果是学生的话可以免费申请，不得不说还是非常人性化。



除了这些之外，还有七牛团队开源的liteide，基于C++ QT开发，因此执行效率很高，但据说调试功能不太好用。我也没有用过，感兴趣的小伙伴可以试试。



扫码关注我，获取更多文章。



Golang——富有个性化的语言设计

今天是**Golang专题的第二篇**，我们来看看Go的语言规范。

在我们继续今天的内容之前，先来回答一个问题。

有同学在后台问我，**为什么说Golang更适合分布式系统的开发**？它和Java相比有什么优势吗？

其实回答这个问题需要涉及很多概念，比如操作系统当中关于进程、线程、协程等很多概念。我们将这些内容进行简化，举一个最简单的线程的例子。我们来写一段在**java当中实现多线程**的例子：

```
1  public class MyThread implements Runnable {
2      public void run() {
3          System.out.println("I am a thread")
4      }
5  }
6
7  public class Test {
8      public static void main(String args[]) {
9          MyThread thread1 = new MyThread();
10         thread1.start();
11     }
12 }
```

我们再来看看Golang：

```
1  func run() {
2      fmt.Println("I am a Thread")
3  }
4
5  func main() {
6      go run()
7  }
```

这么一对比是不是简单很多？

Golang的语言规范

大家都知道程序员最大的分歧之一就是**花括号到底应该写在哪一行**，有另写一行的，也有跟在循环体后面的。这两拨人分成了两个流派，彼此征战不休，也衍生出了许多段子。

为了统一风格，很多语言对代码风格做了规范。比如Python就去掉了花括号，而使用空格来进行代码缩进。然而不幸的是，有些人缩进用四个空格，也有些人用tab，这双方又形成了阵营，彼此争吵不停.....

也许Golang的开发曾经饱受代码风格争吵的苦恼，所以Golang做了一个划时代的事情，它**严格限制了代码风格**，强行统一大家都必须使用同一套风格，否则就分分钟报错给你看。所以在我们进行具体的语法学习之前，先从语言规范开始，否则等我们后面养成了不好的习惯再想要改正就会成本很高。其实改正代码风格是一件很难的事情，老实说我的代码风格不是很好，总是使用一些cur、pnt、node、u、v这种简单的变量，这也是当年打acm留下来的习惯，想改一时半会蛮难的。所以大家一定要在初期就养成好习惯，坏习惯就留给我一个人吧（大雾）。

package规范

Golang的语言规范很多，涉及的面很广，有些我们暂时用不到，我们先挑基础的说。首先是**package规范**，对于package来说它的名字应该和目录保持一致，采取有意义的包名，不要起一些别人看不懂的名字。比如test、unit这种，并且不能和标准库冲突。

其次是在我们引包的时候，需要注意**不要使用相对路径**，而应该使用绝对路径。

```
1 // wrong
2 import "../../repo"
3
4 // correct
5 import "github.com/repo/package"
```

当然我们可以装一个**goimport工具**，帮助我们自动引包。但是自动引包也会有坑，尤其是当目录下存在两个包名称一样的时候，有可能会引入错误，需要我们自己留意。

代码风格规范

Go语言当中规定了我们应该使用**驼峰标准**来命名变量，不能使用_。在Go当中首字母大写表示结构体中的变量或者是包中的函数public，如果是**小写则表示是private**，这一点尤其需要注意。刚开始写go的时候都会很不习惯，因此踩坑是常有的事。

golang当中是有常量的，golang当中的常量一样用**驼峰标准，首字母大写**。比如我们起一个常量叫做app_env，表示当前app运行的环境，我们必须这样定义：

```
const AppEnv = "env"
```

另一点是Golang的设计者**认为行尾加上分号毫无必要**，所以在编译器当中添加了会在行尾自动加上分号的功能。所以我们可以加也可以不加，但是一般认为没有必要这么做。所以普遍来说，除了在循环体或者是判断条件当中，我们一般是不写分号的。当然也有特殊情况，比如你想要把多条语句写在一行的时候：

```
1 var a int; var b float;
2 a = 3; b = 3.2;
```

当然还是一般不推荐这么干，建议分成多行，更加美观。

另外一点是golang当中**所有的变量和包都必须用上**，不允许定义没有使用的东西，否则也会报错。也就是说严格限制了我们写代码时候的谨慎。不能随意申请用不到的变量，大多数语言当中没有这样的限制，但是golang当中做了限制，所以我们写代码的时候要小心。

另外一点是关于花括号，在golang当中**严格限制了花括号写在当前行**，而不是另起一行。

```
1 // wrong
2 if expression
3 {
4     ...
5 }
6
7 // correct
8 if expression {
9     ...
10 }
```

从上面这个例子我们还可以注意到一点，就是在golang当中**if后面的条件不加括号**，这点和Python一样。但是如果你写惯了java或者是C++刚开始可能会不太适应。

最后一点是golang的代码规范检测工具golint当中规定了**所有的函数以及结构体头部必须要写注释**，并且对注释的规范也进行了限制。注释的规范是**名称加上说明**，如果不写或者是不规范的话，代码虽然可以运行，但是无法通过golint的规范检测。一般来说公司的开发环境都会做限制，只有通过golint规范检测的代码才可以提交发布。

```
1 // HelloWorld print hello world
2 func HelloWorld() {
3     fmt.Println("Hello World")
4 }
```

另外一点是golang**不支持隐式类型转换**，比如int和int32以及int64，会被视作是不同的类型。如果我们将一个int32的变量赋值给int类型，则会引起报错，必须要我们手动转换。这当然增加了编码时候的工作，但是也避免了很多由精度不一样产生的问题。

除了这些之外，golang当中还定义了对结构体定义以及错误处理等内容的规范。但是对于我们初学者而言，目前这些是必须要了解的，其他的内容可以等我们后续遇见了再熟悉。

一门语言对于代码风格做了严格的规范限制**对于初学者而言可能是一件比较蛋疼的事情**，因为要记的东西变多了，我们不仅要学会语法，还要搞清楚这些规范。但是当我们熟悉了或者是工作了之后，会发现这其实是一件好事。对于多人协作的场景而言，大家都遵守一样的规范会大大**提升代码交流以及协作的效率**。如果你们看过其他代码风格和自己完全不同的人的代码之后，相信你们对于这点一定会有更深的认识。

总结

从规范的严格程度以及对**面向对象的阉割**程度看起来，golang简直不像是一门新生的语言，倒有些上世纪老派语言的风格。但是偏偏golang又有很多新鲜特性，比如允许**函数值返回多个结果**，支持匿名函数以及部分函数式编程的功能等等。在初学的阶段，我也非常抗拒它，可能是因为Python写得太多了，习惯了动态语言。但是随着对这门语言了解的深入，我越来越多地发现了它这些设计理念背后的思考和智慧，慢慢对它改观，时至今日，我已经不再怀疑这是一门优秀的语言，这几年的流行并不是没有道理的。

另外很重要的一点是，因为golang太特立独行了，所以经常会让我思考它这么做背后的用意是什么？这么一思考，加上查阅一些资料，能够发现很多之前思维当中的盲点。在之前学习语言的时候，我是绝对不会去思考语言的设计者为什么要这么设计的，只会依葫芦画瓢，照着把相关的内容学会仅此而已。这样的思考除了能够提升对于语言本身的理解之外，也能够提升对问题场景的思考和理解，对于工程师而言，后者其实是更为重要的。

当然这些内容我光说是没有用的，也需要屏幕前的你用心去体会。

希望大家都能感受到golang的魅力，都能在此过程当中收货成长，加油！

扫码关注我，获取更多文章



golang——为什么有的语言要把变量类型写在后面？

今天是**Golang**的**第三篇**，我们一起来看看Golang当中的变量。

变量声明

Golang当中的**变量类型和C/C++比较接近**，一般用的比较多的也就是int，float和字符串。Golang当中不一样的地方主要有几点，第一点是严格区分了int，int16，int32和int64，同样区分了float，float32和float64。

前文当中说过，Golang是不支持隐式转换的，哪怕是int和int32也一样。

```
1 var v1 int
2 var v2 int32 = 10
3 v1 = v2
```

上面的代码是会报错的，因为我们用一个**int类型的变量去接收了int32类型**的。虽然Golang当中int一般也是32位的整数，但是这依然会被认为是两个不同的类型。

第二点是Golang当中**自带了map类型**，像是java以及C++虽然也有map，但是都不是原生支持的，而是必须要通过引入包才可以使用的。所以Golang的设计者就觉得这很二，没必要啊，既然所有程序员都要用到map，为什么还非要引入包才能使用呢，直接嵌入在默认类型里好了。于是Golang的基本类型当中就有map。

另外一点是Golang当中是有指针的，但是和C语言当中的指针不太一样，我们先记住这一点，具体的内容我们在后面介绍。

最后一点是Golang当中多了**复数类型**，也就是complex64和complex128，用来支持复数的运算。一般情况下我们也不太用得到，所以暂时略过。

Golang的声明方法很简单，使用var关键字进行。和平常的语言不太一样的是Golang当中的**变量类型写在变量名的后面**，而不是前面。

比如：

```
1 var v1 int
2 var v2 float32
```

刚开始的时候会觉得有些不太适应，但是这样设计是有它的道理的。尤其在**涉及指针**的时候，把变量类型写在后面的方法可以增加可读性。比如我们来看一个例子，在C语言当中支持**指向函数的指针**。

```
double (*funcPtr)(double a, double b);
```

这是一个指向函数的指针，如果我们增加一下难度，比如我们把这个**函数指针作为参数**传入其中也变成一个函数指针，整个定义写出来就会非常复杂：

```
double (*funcPtr)(double a, void(*funcPtr2)(int b, double c));
```

这个**可读性非常差**，估计要看很久才能看懂，如果是忘了函数指针这茬，估计就彻底看不懂了。

我们再来看Golang的定义：

```
var v1 func(a float32, funcA func(b int, c double)) double
```

相比之下，golang的定义要比C看起来**可读性强很多**。如果你看不太明白上面的例子也没关系，我们只需要记住这个结论即可。

初始化

Golang当中有三种**初始化**的方法，我们直接来看代码：

```
1 var v2 int = 10
2 var v3 = 10
3 v4 := 10
```

第一种方式最复杂，我们不但写出了变量类型还写出了初始化之后的值。第二种精简许多，我们只写了值，编译器会自动根据我们写的值去匹配对应的类型。最后一种我们连var和类型都不写了，但是需要加上冒号，和赋值操作做区分。

刚开始可能会有一些不太适应，尤其是Python选手，一向无所谓类型的。但是写习惯了之后还可以，并没有体验很差，而且写这种语法很严谨的语言有助于**提升我们的代码风格和严谨**。

唯一一点要注意的就是同一个变量不能被连续申明两次，下面两种写法都是错误的。

```
1 var s string = "hello"
2 var s string = "hello"
3
4 s := "hello"
5 s := "world"
```

变量赋值

变量赋值其实没什么好说的，就是等于号赋值，直接左边等于右边即可。

如果只使用赋值的话，所有的变量必须已经经过初始化才可以。毕竟不是动态语言，不像Python不能随便定义。不过有一点需要注意，Golang当中非常务实地提供了多变量的赋值操作。比如我们要交换两个变量的值，我们可以用一行语句完成，不再需要引入额外变量了。

```
a, b = b, a
```

另外，Golang当中也支持**匿名变量**，也就是说对于我们不需要的返回值，我们可以不用额外定义一个变量去接收。否则没有用处，还会报错。

比如，假设我们一个函数返回两个变量，但是我们只需要用到其中的一个，我们可以这样写：

```
ret, _ = sample()
```

如果这里的变量之前没有定义过，我们还可以这么写：

```
ret, _ := sample()
```

直接用函数返回值来声明并赋值变量。

关于变量这一块Golang和C++等语言变化不大，如果有C++基础的话，学习起来应该非常快速。并且相比于C++繁多的语法和众多的应用方法，Golang的**学习曲线要平缓很多**，入门也更简单。我相信对你们来说一定都不是问题。

勘误

最后做一个小小的勘误，由于我学习资料过于陈旧，导致上周关于golang中常量定义的阐述发生了错误，在此进行勘误。在最新的golang版本当中，规定const变量也通过驼峰命名法命名，并且首字母必须大写。

这里感谢Taosama大神的勘误。

```
const HelloWorld = "hello world"
```

扫码关注，获取更多文章~



Golang入门教程——基本操作篇

今天是**Golang专题的第四篇**，这一篇文章将会介绍golang当中的函数、循环以及选择判断的具体用法。

函数

在之前的文章当中其实我们已经接触过函数了，因为我们写的main函数本质上也是一个函数。只不过由于main函数没有返回值，也没有传参，所以省略了很多信息。

```
1 func main() {  
2     fmt.Println("Hello World")  
3 }
```

下面，我们来看看一个完整的函数是怎样的，这是golang官网上的例子。

```
1 func add(x int, y int) int {  
2     return x + y  
3 }
```

这是一个非常简单的a+b的函数，我想大家应该都能看懂。我们来重点关注一下函数的格式。首先是func关键字，我们使用这个关键字定义一个函数，之后跟着的是函数名，然后是函数的传参，最后是函数的返回值。

这个顺序可能和我们之前普遍接触的语法不太一样，例如C++当中是把函数返回类型写在最前面，然后是函数名和传参。再比如Python当中则是没有返回值的任何信息，只有def关键字和函数名以及传入的参数。

golang有些像是Python和C++的综合体，总体来说我觉得内涵上更接近C++，但是写法上和Python更接近一些。

我们理解了函数的定义之后，下面来看看golang当中支持的一些特性。

变量简写

在变量声明的时候，我们如果定义两个相同类型的变量是可以把它们进行缩写的。比如我们定义两个int类型的变量，分别叫做a和b。那么可以简写成这样：

```
var a, b int
```

同样，在函数当中，如果传入的参数类型相同，也一样是可以简写的。我们可以把**x和y两个参数缩写在一起**，用逗号分开，共享变量类型。

```
1 func add(x, y int) int {  
2     return x + y  
3 }
```

多值返回

在前面介绍golang特性的时候曾经提到过，golang作为一个看起来很守旧的语言，但是却支持很多新鲜的特性。其中最知名的一个特性就是函数支持**多值返回**，即使是现在，也只有少量的语言支持这一特性。

在许多语言当中，如果需要返回多个值，往往需要用一个结构体或者是tuple、list等数据结构将它们包装起来。但是在golang当中支持同时返回多个结果，这将会极大地方便我们的编码。

```
1 func sample() (string, string) {  
2     return "sample1", "sample2"  
3 }
```

多值返回也会有一个小小的问题，就是如果我们要返回的值过多，会导致这个return会写得很长，或者是组装的逻辑变得很复杂。或者是很容易产生遗漏、搞混顺序之类的问题，golang当中针对这个问题也进行优化，支持我们**对返回值进行命名**。当命名的变量赋值完成之后，我们就可以直接用return关键字返回所有数据。

这个操作很难用语言描述很清楚，我们来看下面的例子：

```
1 func sample(x, y, z int) (xPrime, yPrime, zPrime int) {
2     xPrime, yPrime, zPrime = x-1, y+1, z-2
3     return
4 }
```

在上面的代码当中，在返回之前，我们先给要返回的值起好了名字，我们在函数体当中对这些值进行赋值完成之后，我们就可以直接return了，golang会自动将它们的值填充进行返回。这样不但可以简化一定的编码过程，也可以增加可读性。

defer

golang的函数当中有一个特殊的用法，就是defer。这个用法据说其他语言也有，但是我暂时没有见到过。defer是一个关键字，用它修饰的语句会被存入栈中，**直到函数退出的时候执行**。

比如：

```
1 func main() {
2     defer fmt.Println("world")
3
4     fmt.Println("hello")
5 }
```

上面这两行代码虽然defer的那一行在先，但是并不会被先执行，而是等main函数执行退出之前才会执行。

看起来这个用法有一点点怪，但是它的用处很大，经常用到。比如当我们打开一个文件的时候，不管文件有没有打开成功，我们**都需要记得关闭文件**。但如果文件打开不成功可能会有异常或者是报错，如果我们把这些情况全部都考虑到，会变得非常复杂。所以这个时候我们通常都会用defer来执行文件的关闭。

要注意的是，defer修饰的代码会被放入栈中。所以最后会按照先进后出的原则进行执行。比如：


```

1 func main() {
2     for i := 0; i < 10; i++ {
3         defer fmt.Println(i)
4     }
5
6     fmt.Println("done")
7 }

```

最后执行的结果是9876543210，而不是相反。这一点蛮重要的，有的时候如果搞混了，很容易出现问题。

循环

和其他语言不同，Golang当中**只有一种循环**，就是for循环。没有while，更没有do while循环。在golang的设计中设想当中，只需要一种循环，就可以实现所有的功能。从某种程度上来说，也的确如此，golang中的循环有点像是C++和Python循环的结合体，集合两种所长。

首先，我们先来看下for循环的语法，在for循环当中，我们使用分号分开循环条件。循环条件分为三个部分，第一个部分是初始化部分，我们对循环体进行初始化，第二个部分是判断部分，判断循环结束的终止条件，第三个部分是循环变量的改变部分。

写出来大概是这样的：

```

1 for i := 0; i < 10; i++ {
2     fmt.Println(i)
3 }

```

这个语法是不是和C++中的循环很像呢？可以说除了没有括号之外，基本上就是一样的。golang当中同样支持++的自增操作，不过golang中只支持i++，而不支持++i。

和C++一样，这三段当中的任何一段都是可以省略的，比如我们可以省略判断条件：

```

1 for i := 0; ; i++ {
2     fmt.Println(i)
3     if i > 10 {

```

```
4         break
5     }
6 }
```

我们也可以省略循环变量的自增条件：

```
1  for i := 0; i < 10; {
2      i += 2
3      fmt.Println(i)
4  }
```

甚至可以全部省略，如果全部省略的话，等价于C++中的while(true)循环，也就是死循环。

range的用法

如果我们用循环遍历一个数组或者是map，它的这个用法和Python中的用法非常类似。我们来看下，假如我们有一个数组是：

```
1  nums := []int{2, 3, 4}
2  sum := 0
3  for i, v := range nums {
4      sum += v
5      fmt.Println(i)
6  }
```

这个用法等价于Python中的for i, v in enumerate(nums)。也就是通过range会同时返回数组和map中的下标与对应的值，我们再来看下map，其实也是一样的。

```
1  kvs := map[string]string{"a": "apple", "b": "banana"}
2  for k, v := range kvs {
3      fmt.Printf("%s -> %s\n", k, v)
4  }
```

如果你看不懂map和数组的定义没有关系，我们会在之后的文章当中再来详细讲解，这篇的主要内容是循环。我们只需要看得懂for循环的range操作即可。

判断

golang当中支持if与switch进行条件判断。我们先来看if，在golang当中的if和Python比较接近，在if的判断条件外面不需要加上小括号()，但是if的执行条件当中必须要大括号{}，即使只有一行代码。

比如刚才我们写的循环中的那个break。

```
1  for i := 0; ; i++ {
2      fmt.Println(i)
3      if i > 10 {
4          break
5      }
6  }
```

在判断中初始化

上面的逻辑在各个语言中都大同小异，很多语言都是这么写的。但是golang对于if还有特殊的支持，golang支持在if条件当中加上初始化信息。

比如：

```
1  if v := sample(); v < 10 {
2      fmt.Println(v)
3  }
```

上面当中的v是在if执行的时候才进行的初始化，也就是说我们将变量的初始化和if判断结合在了一起。这个用法非常重要，在golang当中也大规模使用，所以我们一定要学会这个用法。

switch

golang当中也支持switch用法，它的基本套路和C++一样，但是在细微的地方又做了优化。

比如和if一样，switch也支持在执行的时候初始化。比如：

```
1  switch flag := sample(); flag {
2  case "a":
3      fmt.Println(flag)
4  case "b":
5      fmt.Println(flag)
6  default:
7      fmt.Println(flag)
8  }
```

看明白了吗，代码当中的flag是我们执行switch的时候才创建出来的。分号之前的都是初始化的代码，分号之后的表达式才是switch进行判断的内容。

还有一个小细节需要注意，在golang当中使用switch的时候，每个case的判断条件后面**不需要再加上break**。我们在写其他语言的时候，如果用到switch要么就是忘记了case的执行条件后面要加上break，要么就是写很多break非常麻烦。golang的设计者觉得每个case都加上break太二了，因为大家基本上都只用switch执行一个case，所以就去掉了必须要加上break这个设定。

switch执行顺序

在golang当中，switch的判断条件按照顺序执行。

为什么要强调这个呢？因为你很有可能会看到有些人的代码里的**switch没有判断条件**，比如：

```
1  switch a := sample();{
2  case a < 5:
3      fmt.Println(a)
4  case a > 5:
5      fmt.Println(a)
6  default:
7      fmt.Println("end")
8  }
```

在上面这段代码当中，我们根本没有为switch设置判断的根据，这段逻辑完全等同于若干个if-else条件的罗列，它在golang当中同样是允许的。

题外话

今天本来是分布式专题，但实在是没有想到什么很好的题目，我也不喜欢强求，干脆就换个主题吧。以后分布式专题还会更新，不过可能要改成**间歇式**的了，后面想少写点理论，能够分享一点可以实际用上的东西（所以需要的时间比较久）。

不知道大家从今天的内容当中有没有感受到golang这门语言的个性，很多地方看起来中规中矩，却又能创造出新的用法来，至少我是很佩服设计者的想法的。golang当中这些新特性初见的时候往往会觉得不喜欢和排斥，怎么看怎么怪异，但是写多了之后还是蛮香的。

今天的文章就到这里，原创不易，恳请点个**关注**，你的举手之劳对我来说很重要。



手把手golang基础教程——数组与切片

今天是golang专题的第五篇，这一篇我们将会了解golang中的数组和切片的使用。

数组与切片

golang当中数组和C++中的定义类似，除了变量类型写在后面。

比如我们要声明一个长度为10的int型的数组，会写成这样：

```
var a [10]int
```

数组的长度定义了之后不能改变，这点和C++以及Java是一样的。但是在我们日常使用的过程当中，除非我们非常确定数组长度不会发生变化，否则我们一般不会使用数组，而是使用切片(slice)。

切片有些像是数组的引用，它的大小可以是动态的，因此更加灵活。所以在我们日常的使用当中，比数组应用更广。

切片的声明源于数组，和Python中的list切片类似，我们通过指定左右区间的范围来声明一个切片。这里的范围和Python一样，左闭右开。我们来看个例子：

```
1 var a [10]int
2 var s []int = a[0:4]
```

这是标准的声明写法，我们也可以不用var来声明，而是直接利用数组给切片赋值，比如上面的语句可以写成这样：

```
s := a[:4]
```

在Python当中，当我们使用切片的时候，解释器会为我们将切片对应的数据复制一份。所以切片之后和之前的结果是不同的，但是golang当中则不同。切片和数据对应的是**同一份数据**，切片只是数组的一个引用，如果原数组的数据发生变化，那么会连带着切片中的数据一起变化。

还是刚才那个例子：

```
1 var a [10]int
2 var s []int = a[0:4]
3 fmt.Println(s)
```

这样我们输出得到的结果是[0 0 0 0]，因为数组初始化默认值为0。而假如我们修改一个a中的元素，我们再来打印s，得到的结果就不同了：

```
1 var a [10]int
2 var s []int = a[0:4]
3 a[0] = 4
4 fmt.Println(s)
```

这样得到的结果就是[4 0 0 0]，虽然我们并没有修改s当中的数据，由于s本质是a的引用，所以a中发生变化会连带着s一起变化。

进阶用法

前面说了，因为切片比数组更加方便，所以我们日常使用当中都倾向于使用切片，而不是数组。但是根据目前的语法，切片都是从数组当中产生的，这岂不是意味着，我们如果想要使用切片，**必须先要创建出一个对应的数组来吗？**

golang的设计者考虑到了这个问题，为了方便我们的使用，golang设计了直接定义切片的方法。

这是一个数组的声明，我们固定了数组的长度，并且用指定的元素对它进行了初始化。


```
var a = [3]int{0, 1, 2}
```

如果我们去掉长度的声明，那么它就成了一个切片的声明：

```
var a = []int{0, 1, 2}
```

这样是同样可以运行的，在golang的内部下面的语句同样创建了数组，我们获取的a是这个数组的一个切片。但是这个数组对我们是**不可见**的，golang编译器替我们省略了这个逻辑。

长度和容量

理解了切片和数组之间的关系之后，我们就可以来看它的**长度**和**容量**这两个概念了。

这个单词的英文分别是length和capability，长度指的是**切片本身包含的元素个数**，而容量则是**切片对应的数组从开始到末尾包含的元素个数**。我们可以用len操作来获取切片的长度，用cap操作来获取它的容量。

我们来看一个例子，首先我们创建一个切片，然后写一个函数来打印出一个切片的长度和容量：

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     s := []int{1, 2, 3, 4, 5, 6}
7     printSlice(s)
8
9 }
10
11 func printSlice(s []int) {
12     fmt.Printf("len=%d cap=%d %v\n", len(s), cap(s), s)
13 }
```

当我们运行之后得到的结果是这样的：

```
len=6 cap=6 [1 2 3 4 5 6]
```

```
Program exited.
```

这个和我的预期应该是一致的，我们创建出了6个元素的切片，自然它的容量和长度应该都是6，但接下来的操作可能就会有点出入了。

我们对这个切片再进行切片，继续输出切片之后的容量和长度：

```
1 s = s[:2]
2 printSlice(s)
```

运行之后会得到下面这个结果：

```
len=2 cap=6 [1 2]
```

```
Program exited.
```

我们发现它的长度变成了2，但是容量还是6，这个也不是特别难理解。因为虽然当前的切片长度变小了，但是它对应的数组并没有任何变化，所以它的容量应该还是6。

我们继续，我们继续切片：

```
1 s := []int{1, 2, 3, 4, 5, 6}
2 s = s[:2]
3 s = s[:4]
```

```
4 printSlice(s)
```

得到这样的结果：

```
len=4 cap=6 [1 2 3 4]
```

```
Program exited.
```

事情开始有点不一样了，比较令人关注的点有两个。一个是s在之前切片结束之后的结果长度是2，但是我们居然可以对它切片到下标4的位置。这说明我们在执行切片的时候，**执行的对象并不是切片本身**，而是切片背后对应的数组。这一点非常重要，如果不能理解这点，那么切片的很多操作看起来都会觉得匪夷所思难以理解。

第二个点是**切片的容量依然没有发生变化**，这样不会发生变化，那么我们再换一种切片的方法试试，看看会不会有什么不同。

```
1 s = s[2:]
2 printSlice(s)
```

这一次得到的结果就不同了，它是这样的：

```
len=0 cap=4 []
```

```
Program exited.
```

这一次发生了变化了，切片的容量变成了4，也就是说变小了，这是为什么呢？

原因很简单，因为数组的**头指针的位置移动了**。数组原本的长度是6，往右移动了两位，剩下的长度自然就是4了。但是剩下的问题是，为什么数组的头指针会移动呢？

因为数组的头指针和切片的位置是挂钩的，我们前面的切片操作虽然会改变切片中的元素和它的长度，但是都没有改变切片指针的位置。而这一次我们进行的切片是[2:]，当我们执行这个操作的时候，本质上是指针的位置向右移动到了2。

这也是为什么切片的容量定义是它对应的数组从开始到末尾元素的个数，而不是对应的数组元素的个数。因为指针向右移动会改变容量的大小，但是**数组本身的长度是没有变化的**。

我们来看个例子就明白了：

```
1  var a = [6]int{1, 2, 3, 4, 5, 6}
2      s := a[:]
3      //printSlice(s)
4      s = s[:2]
5      printSlice(s)
6      s = s[2:]
7      printSlice(s)
8      //s[0] = 4
9      fmt.Println(a)
```

我们这一次使用显性的切片，我们对s进行一系列切片之后，它的容量变成了4，但是a当中的元素个数还是6，并没有变化。所以**不能简单将容量理解成数组的长度**，而是切片位置到数组末尾的长度。因为切片操作会改变切片指针的位置，从而改变容量，但是数组的大小是没有变化的。

make操作

一般在我们使用切片的时候，我们都是把它当做动态数组用的，也就是Python中的list。所以我们一方面不希望关心切片背后数组，另一方面希望能够有一个区分度较大的构造方法，和创建数组做一个鲜明的区分。

所以基于以上考虑，golang当中为我们提供了一个**make方法**，可以用来创建切片。由于make还可以用来创建其他的类型，比如map，所以我们在使用make的时候，需要传入我们想要创建的变量类型。这里我们想要创建的是切片，所以我们要传入切片的类型，也就是[]int，或者是[]float等等。之后，我们需要传入切片的长度和容量。

比如：

```
s := make([]int, 0, 5)
```

我们就得到了一个长度为0，容量是5的切片。我们也可以只传入一个参数，如果只传入一个参数的话，表示切片的长度和容量相等。

像是这样：

```
s := make([]int, 5)
```

我们如果打印这个s的话，会得到[0 0 0 0 0]，也就是说golang会为我们给切片填充零值。

append方法

前面说了和数组比起来切片的使用更加灵活，意味着**切片的长度是可变的**，我们可以通过使用append方法向切片当中追加元素。

golang中的append方法和Python已经其他语言不同，golang中的append方法需要传入**两个参数**，一个是切片本身，另一个是需要添加的元素，最后会返回一个切片。

所以我们应该写成这样：

```
1 s := make([]int, 4)
2 s = append(s, 4)
```

这么做的目的也很简单，因为切片的**长度是动态的**，也就意味着切片对应的数组的长度也是可变的，至少是可能增大的。如果当前的数组容量不足以存储切片的时候，golang会分配一个更大的数组，这时候会返回一个指向新数组的切片。也就是说由于切片底层实现机制的关系，导致了append方法不能做成inplace的，所以必须要进行返回。我猜，这也是由于性能

二维切片

最后我们来看看**二维切片**在golang当中应该怎么实现，只要能理解二维，拓展到多维也是一样。

golang创造二维切片的方式和C++创建二维的vector有些类似，我们一开始先直接定义一个二维的切片，然后用循环往里面填充。我们定义二维切片的方法和一维的切片类似，只是多了一个方括号而已，之后我们用循环往其中填充若干个一维切片：

```
1 mat := make([][]int, 10)
2 for i := 0; i < 10; i++ {
3     mat[i] = make([]int, 10)
4 }
```

结尾

到这里，golang中关于数组和切片的常见的用法就介绍完了。不仅如此，关于切片底层的实现原理，我们也有了一点浅薄的理解。刚开始接触切片这个概念的时候可能会觉得有点怪，总觉得好像和我们之前学习的语言对不上号，关于容量的概念也不太容易理解，这个是非常正常的，本质上来说，这一切看起来不太正常或者是不太舒服的地方，背后都有创作者的思考，以及为了**性能的权衡**。所以，如果你觉得想不通的话，可以多往这个方面思考，也许会有不一样的收获。

今天的文章就到这里，原创不易，**扫码关注**我，获取更多精彩文章。



人人都能懂的go语言教程——字符串篇

今天是golang专题的第6篇文章，这篇主要和大家聊聊golang当中的字符串的使用。

字符串定义

golang当中的字符串本质是只读的字符型数组，和C语言当中的char[]类似，但是golang为它封装了一个变量类型，叫做string。知道了string这个类型之后，我们就可以很方便地来初始化：

```
1 var str string
2 str1 := "hello world"
3 var str2 = "hello world too"
```

这里应该没什么难度，很好理解。由于这个数组是只读的，所以我们可以通过下标获取某一位的字符，但是不允许修改。

```
1 // 允许
2 fmt.Println(str1[3])
3
4 // 错误
5 str1[3] = 'l'
```

这个也不是golang的独创，很多语言当中都有这个限制，因为会将字符串作为const类型存储在专门的区域。所以不允许字符串进行修改，比如Python也是如此。

除了像是数组一样，支持下标的访问之外，go中的字符串还支持拼接以及求长度的操作。我们可以用len函数获取一个字符串的长度，用+来表示字符串的拼接：

```
1 len("hello")
2 // 5
3
4 c := "hello" + "world"
5 // c="helloworld"
```


这些本来也属于常规操作，并不值得一提，但是关于len函数，值得仔细说说。这里有一个坑，关于utf-8编码。我们来看下面这个例子：

```
1 str := "hello 世界"
2 fmt.Println(len(str))
```

按照我们的设想，它返回的应该是8，但是实际上我们这么操作会得到12。原因很简单，因为在utf-8编码当中，一个汉字需要3个字节编码。那如果我们想要得到字符串本身的长度，而不是字符串占据的字节数，应该怎么办呢？这个时候，我们需要用到一个新的结构叫做rune，它表示单个Unicode字符。

所以我们可以将string转化成rune数组，之后再来计算长度，得到的结果就准确了。

```
1 str := "hello 世界"
2 fmt.Println(len([]rune(str)))
```

这样我们得到的结果就是8了，和我们预期一致了。如果你在使用golang的时候，需要用到utf-8编码，一定要小心。

类型转换

golang当中的字符串不像Java或者其他语言一样封装地非常完善，当我们想要将整形或者是浮点型转成字符串，或者是想要将字符串转成整形和浮点型的时候并没有方法可以直接调用，而必须要通过库函数。golang当中提供了strconv库，用来实现字符串的一些操作。

字符串转整数、浮点数

字符串转整数的方法有两个，一个是ParseInt还有一个是ParseUint，这两个方法本质上都是将字符串转成整数。区别在于前者会保留符号，后者是无符号的，用于无符号整数。

这两个函数都接受三个参数，第一个参数是要转类型的字符串，第二个参数是字符串的进制，比如二进制、八进制还是16进制、32进制。第三个参数表示返回bit的大小，有效值为0，8，16，32，64，如果传入0就返回int或者是uint类型，如果是32，则会返回int32类型。

函数的返回值有两个，第一个是类型转换之后的结果，第二个是一个error，也就是异常类型，表示在转换的过程当中是否有出现异常。如果没有异常，那么这个值会是一个nil。我们判断异常是否是nil就知道有无错误产生，这也是golang当中判断操作有没有异常的常规做法。

所以，代码写出来会是这样的：

```
1 value, err := strconv.ParseInt("33225", 10, 32)
2 if err != nil {
3     fmt.Println("error happens")
4 }
```

如果你不想要这么多功能，就想简单一点将字符串转成int来使用，也可以通过Atoi函数。相比于ParseInt它要简单一些，只需要传入字符串即可，它默认按照10进制进行转换，并且转换之后会返回int类型的整数。

```
1 value, err := strconv.Atoi("33234")
2 if err != nil {
3     fmt.Println("error happens")
4 }
```

字符串转浮点数只有一个函数，就是ParseFloat，由于浮点数没有进制一说，所以它只有两个参数。第一个参数是待转的字符串，第二个参数是bit的大小。和ParseInt一样，它会返回两个结果，一个是转换之后的结果，一个是error异常。

```
1 value, err := strconv.ParseFloat("33.33", 32)
2 if err != nil {
3     fmt.Println("error happens")
4 }
```

整数、浮点数转字符串

将整数和浮点数转字符串都是用Format方法，根据我们要转的类型不同，分为FormatInt和FormatFloat。FormatInt可以认为是ParseInt的逆向操作，我们固定传入一个int64的类型，和整数的进制。golang会根据我们的数字和进制，将它转成我们需要的字符串。

如果指定的进制超过10进制，那么会使用a-z字母来表示大于10的数字。

比如我们把180转成16进制，会得到b4

```
1 num := 180
2 fmt.Println(strconv.FormatInt(int64(num), 16))
```

如果我们固定要按照10进制的整数进行转换，golang还为我们提供了简化的函数Itoa，默认按照10进制转化，它等价于FormatInt(i, 10)，这样我们只需要传入一个值即可。

```
1 num := 180
2 fmt.Println(strconv.Itoa(num))
```

浮点数转字符串逻辑大同小异，但是传参稍有变化。因为浮点数可以用多种方式来表示，比如科学记数法或者是十进制指数法等等。golang当中支持了这些格式，所以允许我们通过传入参数来指定我们希望得到的字符串的格式。

FormatFloat接受4个参数，第一个参数就是待转换的浮点数，第二个参数表示我们希望转换之后得到的格式。一共有'f', 'b', 'e', 'E', 'g', 'G'这几种格式。

看起来有些眼花缭乱，我们仔细说说。

'f' 表示普通模式：(-ddd.dddd)

'b' 表示指数为二进制：(-dddp±ddd)

'e' 表示十进制指数，也就是科学记数法的模式：(-d.dddde±dd)

'E' 和'e'一样，都是科学记数法的模式，只不过字母e大写：(-d.ddddE±dd)

'g' 表示指数很大时用'e'模式，否则用'f'模式

'G' 表示指数很大时用'E'模式，否则用'f'模式

我们来看个例子：

```
1 num := 23423134.323422
2 fmt.Println(strconv.FormatFloat(float64(num), 'f', -1, 64))
3 fmt.Println(strconv.FormatFloat(float64(num), 'b', -1, 64))
4 fmt.Println(strconv.FormatFloat(float64(num), 'e', -1, 64))
5 fmt.Println(strconv.FormatFloat(float64(num), 'E', -1, 64))
6 fmt.Println(strconv.FormatFloat(float64(num), 'g', -1, 64))
7 fmt.Println(strconv.FormatFloat(float64(num), 'G', -1, 64))
```

得到的结果如下：

```
23423134.323422
6287599743057036p-28
2.3423134323422e+07
2.3423134323422E+07
2.3423134323422e+07
2.3423134323422E+07
```

image-20200522114728835

字符串和bool型转换

除了常用的整数和浮点数之外，strconv还支持与bool类型进行转换。

其中将字符串转成bool类型用的是ParseBool，它只有一个参数，只接受0, 1, t, f, T, F, true, false, True, False, TRUE, FALSE这几种取值，否则会返回错误。

```
1 flag, err := strconv.ParseBool('t')
2 if err != nil {
3     fmt.Println("error happens")
4 }
```

将bool转字符串调用FormatBool方法，它也只有一个参数，就是一个bool类型的变量，返回值也是确定的，如果是True就返回"true"，如果是False就返回"false"。

```
fmt.Println(strconv.FormatBool(true))
```

字符串运算包

前面介绍的strconv包是golang当中字符串的一个转换操作包，可以用来将字符串转成其他类型，将其他类型转化成字符串。关于字符串本身的一些操作，还有一个专门的包叫做strings。

字符串比较

我们可以通过strings.Compare来比较两个字符串的大小，这个函数类似于C语言当中的strcmp，会返回一个int。

```
cmp := strings.Compare(str1, str2)
```

cmp等于-1表示str1字典序小于str2，如果str1和str2相等，cmp等于0。如果cmp=1，表示str1字典序大于str2。

其他函数

我们可以用Index函数查找一个字符串中子串的位置，它会返回第一次出现的位置，如果不存在返回-1.

```
var theInd = strings.Index(str, "sub")
```

类似的方法是LastIndex，它返回的是出现的最后一个位置，同样，如果不存在返回-1.

```
var theLastIdx = strings.LastIndex(str, "last")
```

我们可以用Count来统计子串在整体当中出现的次数。

```
strings.Count("abcabcabababc", "abc")
```

第一个参数是母串，第二个参数是子串。如果子串为空，则返回母串的长度+1.

有count自然就有重复，我们可以用Repeat方法来讲字符串重复指定的次数：

```
repeat := strings.Repeat("abc", 10)
```

还有Replace函数，可以替换字符串中的部分。这个函数接收四个参数，分别是字符串，匹配串和目标串，还有替换的次数。如果小于0，表示全部替换。

```
1 str := "aaaddc"
2
3 strings.Replace(str, "a", "b", 1) // baaddc
4 strings.Replace(str, "a", "b", -1)
```

我们还可以通过Split方法来分割字符串，它的使用方法和Python当中的split一样，我们传入字符串与分隔符，会返回根据分隔符分割之后的字符串数组：

```
1 str := "abc,bbc,bbd"
2
3 slice := strings.Split(str, ",")
```

除了Split之外，我们也经常使用它的逆操作也就是Join。通过我们指定的分隔符，将一个字符串数组拼接在一起。

```
1 slice := []string{"aab", "aba", "baa"}
2 str := strings.Join(slice, ",")
```

strings当中的函数除了刚才列举的之外还有很多，比如用来去除字符串首尾多余字符的Trim和TrimLeft，判断是否包含前缀的HasPrefix和判断是否包含后缀的HasSuffix等等，由于篇幅限制，不一一列举了，大家用到的时候可以查阅strings的api文档。

总结

到这里，关于golang当中string的一些基本用法就介绍完了。一般来说，我们日常需要用到的功能，strings和strconv这两个库就足够使用了。初学者可能经常会把这两个库搞混淆，其实很容易分清，strings当中封装的是操作字符串的一些函数。比如字符串判断、join、split等各种处理，而strconv是专门用来字符串和其他类型进行转换的，除此之外基本上没有其他的功能。牢记这两点之后，很容易区分开。

今天介绍的api有些多，如果记不过来也没有关系，我们只需要大概有一个印象即可，具体可以使用到的时候再去查阅相关的资料。

如果觉得有所收获，请给我一个关注。



Golang入门教程——map篇

今天是golang专题的第7篇文章，我们来聊聊golang当中map的用法。

map这个数据结构我们经常使用，存储的是key-value的键值对。在C++/java当中叫做map，在Python中叫做dict。这些数据结构的名称虽然不尽相同，背后的技术支撑也不一定一样，比如说C++的map是红黑树实现的，Java中的hashmap则是通过hash表。但是使用起来的方法都差不多，除了Java是通过get方法获取键值之外，C++、Python和golang都是通过方括号获取的。

声明与初始化

golang中的map声明非常简单，我们用map关键字表示声明一个map，然后在方括号内填上key的类型，方括号外填上value的类型。

```
var m map[string] int
```

这样我们就声明好了一个map。

但是要注意，这样声明得到的是一个空的map，map的零值是nil，可以理解成空指针。所以我们不能直接去操作这个m，否则会得到一个panic。

```
panic: assignment to entry in nil map
```

panic在golang当中表示非常严重不可恢复的错误，可以恢复的错误有些类似于Java或者是其他语言当中的异常，当异常出现的时候，我们可以选择handle住它们，让程序不崩溃继续运行。而那些非常严重，无法handle的异常在golang当中称为panic。

golang当中的异常处理机制和其他语言相差很大，整体的逻辑和内核都不太一样。当然这个是一个比较大的话题，我们这里可以简单将它理解成error就行了。

回到map上来，我们声明了一个map之后，想要使用它还需要对它进行初始化。使用它的方法也很简单，就是使用make方法创建一个实例来。它的用法和之前通过make创建元组非

常类似：

```
1 m = make(map[string] int)
2
3 // 我们还可以指定创建出来的map的存储能力的大小
4 m = make(map[string] int, 100)
```

我们也可以在声明的时候把初始化也写上：

```
var m = map[string] int {"abc": 3, "ccd": 4}
```

当然也可以通过赋值运算符，直接make出一个空的map来：

```
m := make(map[string] int)
```

增删改查

map创建好了当然是要用的，整体使用起来和Python当中的dict比较像，比较简单直观，没有太多弯弯绕的东西。我们一个一个来看，首先是map的添加元素。map的添加元素直接用方括号赋值即可：

```
m["abc"] = 4
```

同样，我们需要保证这里的m经过初始化，否则也会包nil的panic。如果key值在map当中已经存在，那么会自动替换掉原本的key。也就是说map的更新和添加元素都是一样的，都是通过这种方式。如果不存在就是添加，否则则是更新。

删除元素也很简单，和Python当中类似，通过delete关键字删除。

```
delete(m, "abc")
```

当我们删除key的时候，如果是其他的语言，我们需要判断这个key值是否存在，否则的话不能删除，或者是会引起异常。在golang当中并不会，对这点做了优化。如果要删除的key值原本就不在map当中，那么当我们调用了delete之后，什么也不会发生。但是有一点，必须要保证传入的map不为nil，否则也会引起panic。

最后，我们看下元素的查找。对于Java和Python来说我们都是通过一些判断语句来进行判断的，比如java的话是containsKey，Python的话用in操作符。在golang当中我们则是直接通过方括号进行查询，那么这就有了一个问题，如果key不在其中怎么办？

如果是其他语言，我们直接访问一个不存在的key是会抛出异常的，但是在golang当中不会触发panic，只会返回一个error。所以我们可以用两个变量去接收map访问的结果，如果没有error，那么会得到一个nil。我们只需要判断error是不是为nil，就知道元素存不存在了。

进一步，我们还可以将这个逻辑和if的初始化操作合在一起：

```
1  if val, ok := m["1234"]; ok {  
2      fmt.Println(val)  
3  }
```

最后，我们看一个实际运用map的例子，通过map来生成统计字符串当中单词数量的wordCount：

```
1  package main  
2  
3  import (  
4      "golang.org/x/tour/wc"  
5      "strings"  
6  )  
7  
8  func WordCount(s string) map[string]int {  
9      cnt := make(map[string]int)  
10     // 通过Split方法拆分字符串  
11     for _, str := range strings.Split(s){  
12         // 直接++即可，golang会自动填充
```

```
13     cnt[str]++
14 }
15 return cnt
16 }
17
18 func main() {
19     wc.Test(WordCount)
20 }
```

总结

到这里，关于golang当中map的使用就算是介绍完了。我们可以发现，map一如既往地体现了golang语法精简的特点。比如通过返回error的操作省略了判断元素是否存在map当中的操作，刚开始的时候会觉得有些不太适应，但是接触多了之后，会发现这些都是有套路的。golang的套路就是精简，能省就省，能简单绝不复杂。

这一点不仅在map上体现，在其他特性上也是一样。在后续的内容当中，我们还会继续感知这一点。

如果喜欢本文，可以的话，请点个关注，给我一点鼓励，也方便获取更多文章。



Golang简单入门教程——函数进阶篇

今天是golang专题的第八篇，我们来聊聊golang当中的函数。

我们在之前的时候已经介绍过了函数的基本用法，知道了怎么样设计或者是定义一个函数，以及怎么样调用一个函数，还了解了defer的用法。今天这篇文章我们来继续深入这个话题，来看看golang当中关于函数的一些进阶的用法。

返回error

前文当中我们曾经提到过，在golang当中并没有try catch捕获异常的机制。在其他语言当中异常只有一种，可以通过try catch语句进行捕获，而golang当中做了区分，将异常分为两种，一种是在函数当中返回的error，另外一种严重的会引起程序崩溃的panic。

在golang中，error也是一个数据类型，由于golang支持函数的多值返回，所以我们可以设置一个返回值是error。我们通过对这个error的判断来获取运行函数的情况。

举个例子，比如说，假设我们实现一个Divide函数实现两个int相除。那么显然我们需要除数不能为0，当除数为0的时候我们需要返回一个异常。这个时候我们可以把代码写成这样：

```
1 // Divide test
2 func Divide(a, b int) (ret int, err error) {
3     if b == 0 {
4         err = errors.New("divisor is zero")
5         return
6     }
7     return a / b, nil
8 }
```

当我们调用函数的时候，我们用两个变量去接收这个函数返回的结果，第二个变量的类型是error。当这个函数成功执行的时候第二个变量的结果为nil，我们只需要判断它是否等于nil，就可以知道函数执行是否成功。如果不成功，我们还可以记录失败的原因。

```
1 func main() {
2     ret, err := Divide(5, 2)
3     if err == nil {
```

```
4         fmt.Println(ret)
5     } else {
6         fmt.Println(err)
7     }
8 }
```

这种用法在golang当中非常常见，我们之前在介绍字符串相关操作的时候也介绍过返回error的用法。我们在设计函数的时候如果需要判断输入的合法性可以使用error，这样就可以保证handle住非法的情况，并且也能让下游感知到。

不定参数

不定参数的用法在很多语言当中都有，比如在Python当中，不定参数是*args。通过*args我们可以接受任何数量的参数，由于Python是弱变量类型的语言，所以args这些参数的类型可以互不相同。但是golang不行，golang严格限制类型，**不定参数必须要保证类型一样**。除此之外，其他的用法和Python一样，不定参数会以数组的形式传入函数内部，我们可以使用数组的api进行访问。

我们来看一个例子，我们通过...来定义不定参数。比如我们可以实现一个sum函数，可以将任意个int进行累加。

```
1 func Sum(nums ... int) int{
2     ret := 0
3     for _, num := range nums {
4         ret += num
5     }
6     return ret
7 }
```

我们来仔细研究一下上面这个例子，在这个例子当中，我们通过...传入了一个不定参数，我们不定参数的类型只写一次，写在...的后面。从底层实现的机制上来说，不定参数本质上是**将传入的参数转化成数组的切片**。但是这就有了一个问题，既然传入的是一个数组的切片，我们为什么要专门设置一个关键字，而不是规定传入一个切片呢？

比如上面的代码我们完全可以写成这样：

```

1 func Sum(nums []int) int{
2     ret := 0
3     for _, num := range nums {
4         ret += num
5     }
6     return ret
7 }

```

无论从代码的阅读还是编写上来看相差并不大，好像这样做完全没有意义，其实不是这样的。这个关键字简化的并不是函数的设计方，而是函数的使用方。如果我们规定了函数的输入是一个切片，那么当我们在传入数据的时候，**必须要使用强制转化**，将我们的数据转化成切片，比如这样：

```
Sum([]int(3, 4, 6, 8))
```

而使用...关键字我们则可以省略掉强制转化的过程，上面的代码我们写成这样就可以了：

```
Sum(3, 4, 6, 8)
```

很明显可以看出差异，使用不定参数的话调用方会轻松很多，不需要再进行额外的转换。如果我们要传入的也是一个数组，那么**在传递的时候也需要用...符号将它展开**。

```

1 a := make([]int)
2 a = append(a, 3)
3 a = append(a, 4)
4 Sum(a...)
5 Sum(a[1:]...)

```

既然聊到不定参数的传递，那么又涉及到了一个问题，当我们想要像Python那样传递多个类型不同的参数的时候，应该怎么办呢？按照道理golang是静态类型的语言，限制死了参数的类型，是不能随便转换的才对。但是偏偏这样操作是可以的，因为golang当中有一个特殊的类型，叫做**interface**。

interface的用法很多，一个很重要的用法是用在面向对象当中充当结构体的接口。这里我们不做过多深入，我们只需要知道，interface的一个用法是**可以用来代替所有类型的变量**。我们来看一个例子：

```
1 func testInterface(args ...interface{}) {
2     for _, arg := range args {
3         switch arg.(type) {
4             case int:
5                 fmt.Println("it's a int")
6             case string:
7                 fmt.Println("it's a string")
8             case float32:
9                 fmt.Println("it's a float")
10            default:
11                fmt.Println("it's an unknown type")
12        }
13    }
14 }
15
16
17 func main() {
18     testInterface(3, 4.5, "abc")
19 }
```

我们可以用.type获取一个interface变量实际的类型，这样我们就实现了任意类型任意数量参数的传入。

匿名函数和闭包

匿名函数我们在Python当中经常使用到，其实这个概念出现已久，最早可以追溯到1958年Lisp语言。所以这并不是一个新鲜的概念，只是传统的C、C++等语言没有支持匿名函数的功能，所以显得好像是一个新出现的概念一样。golang当中也支持匿名函数，但是golang当中匿名函数的使用方式和Python等语言稍稍有些不同。

在Python当中我们是通过lambda关键字来定义匿名函数，它可以被传入另一个函数当中，也可以赋值给一个变量。golang当中匿名函数的定义方式和普通函数基本是一样的，**只是没有函数名而已**，不过它也可以被传入函数或者是赋值给另一个变量。

比如：


```

1 s := func(a, b int) int {
2     return a + b
3 }
4
5 c := s(3, 4)

```

除了匿名函数之外，golang还支持**闭包**。闭包的概念我们在之前Python闭包的介绍当中曾经提到过，我们之前也用过好几次，闭包的本质不是一个包，而是一个函数，**是一个持有外部环境变量的函数**。比如在Python当中，我们经常可以看到这样的写法：

```

1 def outside(x):
2     def inside(y):
3         print(x, y)
4     return inside
5
6
7 ins = outside(3)
8 ins(5) #3, 5

```

我们可以看到outside这个函数返回了inside这个函数，对于inside这个函数而言，它持有了x这个变量。x这个变量并不是属于它的，而是定义在它的外部域的。并且我们在调用inside的时候是无法干涉这个变量的，这就是一个闭包的典型例子。根据轮子哥的说法，**闭包的闭的意思并不是封闭内部，而是封闭外部**。当外部scope失效的时候，函数仍然持有一份外部的环境的值。

golang当中闭包的使用方法大同小异，我们来看一个类似的例子：

```

1 func main() {
2     a := func(x int) (func(int)) {
3         return func(y int){
4             fmt.Println(x, y)
5         }
6     }
7     b := a(4)
8     b(5)
9 }

```

这个闭包的例子和刚才上面Python那个例子是一样的，唯一不同的是由于golang是强类型的语言，所以我们需要在定义闭包的时候将输入和输出的类型定义清楚。

总结

关于golang当中函数的高级用法就差不多介绍完了，这些**都是实际编程当中经常使用的方法**，如果想要学好golang这门语言的话，这些是基本功。如果你之前有其他语言的基础，来写go的话，整体上手的难度还是不大的，很多设计都可以在其他的语言当中找到影子，有了参照来学会简单得多。

我很难描述实际工作当中写golang的体验，和我写任何一门其他的语言都不一样，有一种一开始期望很低，慢慢慢慢总能发现惊喜的感觉。我强烈建议大家去实际感受一下。

、

如果喜欢本文，可以的话，请**点个关注**，给我一点鼓励，也方便获取更多文章。



Golang入门教程——面向对象篇

今天是**golang专题**的第9篇文章，我们一起来看看golang当中的面向对象的部分。

在现在高级语言当中，面向对象几乎是不可或缺也是一门语言最重要的部分之一。golang作为一门刚刚诞生十年的新兴语言自然是支持面向对象的，但是golang当中面向对象的概念和特性与我们之前熟悉的大部分语言都不尽相同。比如Java、Python等，相比之下，golang这个部分的设计**非常得简洁和优雅**（仁者见仁），所以即使你之前没有系统地了解过面向对象，也没有关系，也一定能够看懂。

常见的面向对象的部分，比如继承、构造函数、析构函数，这些内容在golang当中**统统没有**，因此整体的学习成本和其他的语言比起来会更低一些。

struct

在golang当中没有类的概念，代替的是**结构体**（struct）这个概念。我们可以给结构体类型定义方法，为了表明该方法的适用对象是当前结构体，我们需要在方法当中定义接收者，位于func关键字和方法名之间。

我们一起来看看一个例子：

```
1  type Point struct {  
2      x int  
3      y int  
4  }  
5  
6  func (p Point) Dis() float64 {  
7      return math.Sqrt(float64(p.x*p.x + p.y*p.y))  
8  }
```

在上面这段代码当中我们定义了一个叫做Point的结构体，以及一个面向这个结构体的方法Dis。我们一个一个来看它们的语法。

对于结构体来说，我们**通过type关键字定义**。在golang当中type关键字的含义是定义一个新的类型。比如我们也可以这样使用type：

```
type Integer int
```

它的含义是**从int类型定义了一个新的类型Integer**，从此之后我们可以在后序的代码当中使用Integer来代替int。它有些类似于C++当中的typedef，结合这个含义，我们再来看结构体的定义就很好理解了。其实是我们通过struct关键字构造了一个结构体，然后使用type关键字定义成了一个类型。

之后我们创建了一个面向结构体Point的函数Dis，这个函数和我们之前使用的函数看起来并没有太多的不同，唯一的区别在于我们在func和函数名之间多了一个(p Point)的定义。这其实是**定义这个函数的接收者**，也就是说它接受一个结构体的调用。

不仅如此，我们可以给golang当中的任何类型添加方法，比如：

```
1  type Integer int
2
3  func (a Integer) Less(b Integer) bool {
4      return a < b
5  }
```

在这个例子当中，我们给原生的int类型添加了Less这个方法，用来比较大小。我们在添加方法之前使用type给int起了一个别名，这是因为**golang不允许给简单的内置类型添加方法，并且接收者的类型定义和方法声明必须在同一个包里**，我们必须使用type关键字临时定义一个新的类型。这里要注意的是，虽然我们定义出来的Integer和int的功能完全一样，但是它们属于不同的类型，不能互相赋值。

和别的语言比较起来，这样的定义的一个好处就是清晰。举个例子，比如在Java当中，同样的功能会写成不同的样子：

```
1  class Integer {
2      private int val;
3      public boolean less(Integer b) {
4          return this.val < b.val;
5      }
6  }
```

对于初学者而言，可能会觉得困惑，less函数当中的这个this究竟是哪里来的？其实这是因为Java的成员方法当中隐藏了this这个参数，这一点在Python当中要稍稍清晰一些，因为它将self参数明确地写了出来：

```
1 class Integer:
2     def __init__(self, val):
3         self.val = val
4     def less(self, val):
5         return self.val < val.val
```

而golang明确了结构体函数的接收者以及参数，显得更加清晰。

指针接收者

golang当中，我们也可以将函数的接收者**定义成指针类型**。

比如我们可以将刚才的函数写成这样：

```
1 type Point struct {
2     x int
3     y int
4 }
5
6 func (p *Point) Dis() float64 {
7     return math.Sqrt(float64(p.x*p.x + p.y*p.y))
8 }
```

指针接收者和类型接收者在使用上是一样的，我们并不需要将结构体转化成指针类型，可以直接进行调用。golang内部会自己完成这个转化：

```
1 func main() {
2     p := Point{3, 4}
3     fmt.Print(p.Dis())
4 }
```

那么这两者的区别是什么呢？我们既然可以定义成普通的结构体对象，为什么还要有一个指针对象的接收者呢？

其实很好理解，两者的区别有些类似于C++当中的值传递和引用传递。在值传递当中，我们传递的是值的一个拷贝，我们在函数当中修改参数并不会影响函数外的结果。而引用传递则不然，传递的是参数的引用，我们在函数内部修改它的话，会影响函数外的值。

也就是说在golang当中，如果我们函数接收的是一个指针类型，我们可以在函数内部修改这个结构体的值。否则的话，传入的是一个拷贝，我们在其中修改值并不会影响它本身。我们来看个例子：

```
1 func (p *Point) Modify() {
2     p.x += 5
3     p.y -= 3
4 }
5
6 func main() {
7     p := Point{3, 4}
8     p.Modify()
9     fmt.Print(p)
10 }
```

上面这段代码当中函数的接收者是一个指针，所以我们得到的结果会是{8, 1}，如果我们把指针去掉，改成普通的值接收的话，那么最后的结果仍然是{3, 4}。

总结

我们今天学的内容有些多，我们来简单梳理一下。首先，我们了解了通过type和struct关键字来定义一个结构体，**结构体是golang当中面向对象的载体**，golang抛弃了传统的面向对象的实现方式和特性，拥有自己的面向对象的理念。

对于结构体来说，我们可以把它当做是接受者传递给一个函数，使得我们可以以类似调用类当中方法的形式来调用一个函数。并且对于函数而言，接受者除了值以外还可以是一个指针。如果是指针的话，当我们对结构体值进行修改的时候，会影响到原值。即使我们定义的接收者类型是指针，我们在调用的时候也不必显示将它转化成结构体指针，golang当中会自动替我们完成这样的转化。

面向对象部分可以说是golang这一门语言当中最大的创新之一，也正是因为抛弃了传统的类以及继承、派生的概念，使得golang当中的面向对象语法糖相对简洁。也因此有人将golang称为**升级版的C语言**。虽然我们啰啰嗦嗦写了很多，但是实际谈到的内容并不多，我想理解起来也不会特别困难。

今天的文章到这里就结束了，如果喜欢本文，可以的话，请**点个关注**，给我一点鼓励，也方便获取更多文章。



golang | Go语言入门教程——结构体初始化与继承

今天是golang专题第10篇文章，我们继续来看golang当中的面向对象部分。

在上一篇文章当中我们一起学习了怎么创建一个结构体，以及怎么给结构体定义函数，还有函数接收者的使用。今天我们来学习一下结构体本身的一些使用方法。

初始化

在golang当中结构体初始化的方法有四种。

new关键字

我们可以通过new关键字来创建一个结构体的实例，这种方法和其他语言比较类似，这样会得到一个空结构体指针，当中所有的字段全部填充它类型对应的零值。比如int就对应0，float对应0.0，如果是其他结构体则对应nil。

```
1  type Point struct {  
2      x int  
3      y int  
4  }  
5  
6  func main() {  
7      var p *Point = new(Point)  
8      fmt.Print(p)  
9  }
```

从这段代码当中我们可以看到，new函数返回的是一个结构体指针，而不是结构体的值。一般我们很少用new关键字，而是直接通过结构体加花括号的方式来初始化。

结构体名称

相比于使用new关键字，我们更常用的是通过结构体名称加上花括号的方式来进行初始化。

如果我们不再花括号当中填写参数的话，那么同样会得到一个填充了零值的结构体。结构体当中的所有属性都会被赋予这个类型对应的零值。

```
1  type Point struct {
2      x int
3      y int
4  }
5
6  func main() {
7      p := Point{}
8      fmt.Print(p)
9  }
```

如果我们想要初始化一个结构体的指针，我们只需要在结构体名称之前加上取地址符&即可。所以创建一个结构体指针可以这样：

```
1  func main() {
2      p := &Point{}
3      fmt.Print(p)
4  }
```

golang当中取地址符和声明指针的关键字和C语言是一样的，对于熟悉C语言的同学来说，这应该并不困难。

我们在花括号当中填充参数，这些参数会按照顺序填充到结构体的属性当中。为了防止混淆，我们也可以在值之前加上它对应的属性名称。

```
1  func main() {
2      p := &Point{0, 0}
3      k := &Point{x: 0, y: 10}
4      fmt.Print(p)
5  }
```

继承

很多人不喜欢golang的主要原因就是觉得golang阉割了面向对象的很多功能之后，导致开发的时候束手束脚，总觉得不太方便。其中为人诟病得比较厉害的就是继承，觉得golang当中没有继承，写有依赖的结构体的时候非常蛋疼。

我之前一度也这么觉得，最近仔细研究了其中的道道之后，发现我错了，golang当中也是有继承的，不过它实现的方式和我们一般理解上的不太一样，有一些出其不意。所以我们拿正统的眼光去看它总会觉得它不伦不类，哪里不太对劲。这种感觉有点像是武侠小说里名门正派看旁门左派的感觉，但旁门左派并不代表就不行，也有能打的。

在我们正常的映像当中，我们实现继承就应该是标明当前这个类的父类是哪个类，这样底层编译器自动将父类的属性和方法都拷贝一份到子类当中来。加上private、public等关键词束缚，来控制一下什么方法和属性可以被继承什么不可以就完美了。

我们用Python举个例子，Python当中对于继承的定义已经非常简洁了，实现起来大概是这样的：

```
1 class A:
2     pass
3
4 class B(A):
5     pass
```

直接在类名的后面就加上继承的信息，实际上绝大多数主流语言也都是这么干的。但golang不是，它做了一件什么事呢？它将父类作为变量定义在了子类的里面，严格说起来这已经不是继承了，算是一种奇怪的组合，但它起到的功能类似于继承。

我光说理解起来很累，我们来看个例子，比如我们当下有一个父类（结构体），它有两个结构体方法：

```
1 type Father struct {
2     Name string
3 }
4
5 func(entity Father) Hello() {...}
6 func(entity Father) World() {...}
```

现在我们要创建一个它的子类，需要把Father这个结构体填进去，变成其中一个成员变量。

```
1 type Child struct {  
2     Father  
3     ...  
4 }
```

那有了这么一个看起来很奇怪的子类之后，我们怎么调用父类的方法呢？

答案是直接调用。

```
1 child := Child{}  
2 child.Hello()
```

按照我们的理解，由于父类是子类其中的一个成员，所以我们想要调用父类的方法，应该写成child.Father.Hello()才对。但实际上golang替我们做了相关的优化，我们直接调用方法，也可以找到父类当中的方法。

如果我们要改写父类的方法也不困难，我们可以这样操作：

```
1 func (entity Child) World() {  
2     entity.Father.World()  
3     ...  
4 }
```

如此，父类当中的World方法就被Child改写了，这样就完成了继承当中对父类函数的改写。

总结

到这里，关于golang当中结构体初始化与继承的介绍就结束了。不知道大家看完这篇有什么样的感受，我最大的感觉是好像没有第一次看到它的时候那么难以接受了XD。

据说这个设计和C++当中的虚基类的概念非常接近，但是虚基类非常难以理解（比如我就没能理解），以至于许多C++工程师会自动忽略它的存在。相比之下，golang的这种设计要容易理解得多。虽然看起来麻烦，但是理解起来也并不困难。



Golang interface的使用与面向对象多态

今天是golang专题的第11篇文章，我们一起来聊聊golang当中多态的这个话题。

如果大家系统的学过C++、Java等语言以及面向对象的话，相信应该对**多态**不会陌生。

多态是**面向对象范畴**当中经常使用并且非常好用的一个功能，如果你之前没有学过的话也没有关系，我们用一个简单的例子来说明一下。多态主要是用在强类型语言当中，像是Python这样的弱类型语言，变量的类型可以随意变化，没有任何限制，其实区别不是很大。

多态的含义

对于Java或者是C++而言，我们在使用变量的时候，**变量的类型是明确的**。但是如果我们希望它可以宽松一点，比如说我们用父类指针或引用去调用方法，但是在执行的时候，能够**根据子类的类型去执行子类当中的方法**。也就是说实现我们用相同的调用方式调出不同结果或者是功能的情况，这种情况就叫做多态。

举个非常经典的例子，比如说猫、狗和人都是哺乳动物。这三个类都有一个say方法，大家都知道猫、狗以及人类的say是不一样的，猫可能是喵喵叫，狗是汪汪叫，人类则是说话。

```
1  class Mammal {
2      public void say() {
3          System.out.println("do nothing")
4      }
5  }
6
7
8  class Cat extends Mammal{
9      public void say() {
10         System.out.println("meow");
11     }
12 }
13
14
15 class Dog extends Mammal{
16     public void say() {
17         System.out.println("woof");
18     }
19 }
20
21 class Human extends Mammal{
```

```
22     public void say() {
23         System.out.println("speak");
24     }
25 }
```

这段代码大家应该都不难看懂，这三个类都是Mammal的子类，假设这个时候我们有一系列实例，它们都是Mammal的子类的实例，但是这三种类型都有，我们希望用一个循环来一起全都调用了。虽然我们接收变量的时候是用的Mammal的父类类型去接收的，但是我们调用的时候却会获得各个子类的运行结果。

比如这样：

```
1  class Main {
2      public static void main(String[] args) {
3          List<Mammal> mammals = new ArrayList<>();
4          mammals.add(new Human());
5          mammals.add(new Dog());
6          mammals.add(new Cat());
7
8          for (Mammal mammal : mammals) {
9              mammal.say();
10         }
11     }
12 }
```

不知道大家有没有get到精髓，我们创建了一个父类的List，将它各个子类的实例放入了其中。然后通过了一个循环用父类对象来接收，并且调用了say方法。我们希望虽然我们用的是父类的引用来调用的方法，但是它可以自动根据子类的类型调用对应不同子类当中的方法。

也就是说我们得到的结果应该是：

```
1  speak
2  woof
3  meow
```

这种功能就是多态，说白了我们可以在父类当中定义方法，在子类当中创建不同的实现。但是在调用的时候依然还是用父类的引用去调用，编译器会自动替我们做好内部的映射和转化。

抽象类与接口

这样实现当然是可行的，但其实有一个小小的问题，就是Mammal类当中的say方法多余了。因为我们使用的只会是它的子类，并不会用到Mammal这个父类。所以我们没必要实现父类Mammal中的say方法，做一个标记，表示有这么一个方法，子类实现的时候需要实现它就可以了。

这就是抽象类和抽象方法的来源，我们可以把Mammal做成一个抽象类，声明say是一个抽象方法。抽象类是不能直接创建实例的，只能创建子类的实例，并且抽象方法也不用实现，只需要标记好参数和返回就行了。具体的实现都在子类当中进行。说白了**抽象方法就是一个标记**，告诉编译器凡是继承了这个类的子类必须要实现抽象方法，父类当中的方法不能调用。那**抽象类就是含有抽象方法的类**。

我们写出Mammal变成抽象类之后的代码：

```
1  abstract class Mammal {  
2      abstract void say();  
3  }
```

很简单，因为我们只需要定义方法的参数就可以了，**不需要实现方法的功能**，方法的功能在子类当中实现。由于我们标记了say这个方法是一个抽象方法，凡是继承了Mammal的子类都必须要实现这个方法，否则一定会报错。

抽象类其实是一个擦边球，我们可以在抽象类中定义抽象的方法也就是只声明不实现，**也可以在抽象类中实现具体的方法**。在抽象类当中非抽象的方法子类的实例是可以直接调用的，和子类调用父类的普通方法一样。但假如我们不需要父类实现方法，我们提出提取出来的父类中的所有方法都是抽象的呢？针对这种情况，Java当中还有一个概念叫做接口，也就是interface，**本质上来说interface就是抽象类**，只不过是只有抽象方法的抽象类。

所以刚才的Mammal也可以写成：

```
1 interface Mammal {  
2     void say();  
3 }
```

把Mammal变成了interface之后，子类的实现没什么太大的差别，只不过将extends关键字换成了implements。另外，子类只能继承一个抽象类，但是可以实现多个接口。早先的Java版本当中，interface只能够定义方法和常量，在Java8以后的版本当中，我们也可以在接口当中实现一些默认方法和静态方法。

接口的好处是很明显的，我们可以用**接口的实例来调用所有实现了这个接口的类**。也就是说接口和它的实现是一种要宽泛许多的继承关系，大大增加了灵活性。

以上虽然全是Java的内容，但是讲的其实是面向对象的内容，如果没有学过Java的小伙伴可能看起来稍稍有一点点吃力，但总体来说问题不大，没必要细扣当中的语法细节，get到核心精髓就可以了。

讲这么一大段的目的是为了厘清面向对象当中的一些概念，以及接口的使用方法和理念，后面才是本文的重头戏，也就是Go语言当中接口的使用以及理念。

Go语言中的接口

Go语言当中也有接口，但是它的理念和使用方法和Java稍稍有所不同，它们的使用场景以及实现的目的是类似的，本质上都是为了抽象。通过接口提取出了一些方法，所有继承了这个接口的类都必然带有这些方法，那么我们通过接口获取这些类的实例就可以使用了，大大增加了灵活性。

但是Java当中的接口有一个很大的问题就是**侵入性**，说白了就是会颠倒供需关系。举个简单的例子，假设你写了一个爬虫从各个网页上爬取内容。爬虫爬到的内容的类别是很多的，有图片、有文本还有视频。假设你想要抽象出一个接口来，在这个接口当中定义你规定的一些提取数据的方法。这样不论获取到的数据的格式是什么，你都可以用这个接口来调用。这本身也是接口的使用场景，但问题是处理图片、文本以及视频的**组件可能是开源或者是第三方的**，并不是你开发的。你定义接口并没有什么卵用，别人的代码可不会继承这个接口。

当然这也是可以解决的，比如你可以在这些第三方工具库外面自己封装一层，实现你定义的接口。这样当然是OK的，但是显然比较麻烦。

Golang当中的接口解决了这个问题，也就是说它**完全拿掉了原本弱化的继承关系**，只要接口中定义的方法能对应的上，那么就可以认为这个类实现了这个接口。

我们先来创建一个interface，当然也是通过type关键字：

```
1 type Mammal interface {  
2     Say()  
3 }
```

我们定义了一个Mammal的接口，当中声明了一个Say函数。也就是说**只要是拥有这个函数的结构体就可以用这个接口来接收**，我们和刚才一样，定义Cat、Dog和Human三个结构体，分别实现各自的Say方法：

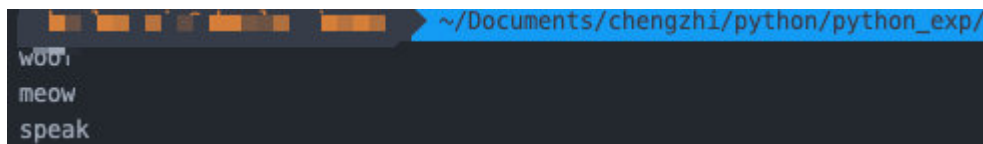
```
1 type Dog struct{}  
2  
3 type Cat struct{}  
4  
5 type Human struct{}  
6  
7 func (d Dog) Say() {  
8     fmt.Println("woof")  
9 }  
10  
11 func (c Cat) Say() {  
12     fmt.Println("meow")  
13 }  
14  
15 func (h Human) Say() {  
16     fmt.Println("speak")  
17 }
```

之后，我们尝试使用这个接口来接收各种结构体的对象，然后调用它们的Say方法：

```
1 func main() {  
2     var m Mammal  
3     m = Dog{}  
4     m.Say()  
5     m = Cat{}  
6 }
```

```
6     m.Say()  
7     m = Human{}  
8     m.Say()  
9 }
```

出来的结果当然和我们预想的一样：



```
~/Documents/chengzhi/python/python_exp/  
w00t  
meow  
speak
```

总结

今天我们一起聊了面向对象中多态以及接口的概念，借此进一步了解了为什么golang中的接口设计非常出色，因为它解耦了接口和实现类之间的联系，使得进一步增加了我们编码的灵活度，解决了供需关系颠倒的问题。但是世上没有绝对的好坏，golang中的接口在方便了我们编码的同时也带来了一些问题，比如说由于没了接口和实现类的强绑定，其实也一定程度上增加了开发和维护的成本。

总体来说这是一个仁者见仁的改动，有些写惯了Java的同学可能会觉得没有必要，这是过度解绑，有些人之前深受其害的同学可能觉得这个进步非常关键。但不论你怎么看，这都不影响我们学习它，毕竟学习本身是不带立场的。今天的内容当中包含一些Java和面向对象的概念，只是用来引出后面golang的内容，如果存在部分不理解的地方，希望大家抓大放小，理解核心关键就好了，不需要细扣每一个细节。

今天的文章到这里就结束了，如果喜欢本文的话，请来一波**素质三连**，给我一点支持吧（**关注、在看、点赞**）。



Golang | 既是接口又是类型，interface是什么神仙用法？

今天是golang专题的第12篇文章，我们来继续聊聊interface的使用。

在上一篇文章当中我们介绍了面向对象的一些基本概念，以及golang当中interface和多态的实现方法。今天我们继续来介绍interface当中其他的一些方法。

万能类型interface

在Java以及其他语言当中接口是一种写法规范，而在golang当中，interface其实也是一种值，它可以像是值一样传递。并且在它的底层，它其实是一个值和类型的元组。

这里我们来看下golang官方文档其中的一个例子：

```
1  package main
2
3  import (
4      "fmt"
5      "math"
6  )
7
8  type I interface {
9      M()
10 }
11
12 type T struct {
13     S string
14 }
15
16 func (t *T) M() {
17     fmt.Println(t.S)
18 }
19
20 type F float64
21
22 func (f F) M() {
23     fmt.Println(f)
24 }
25
26 func main() {
```

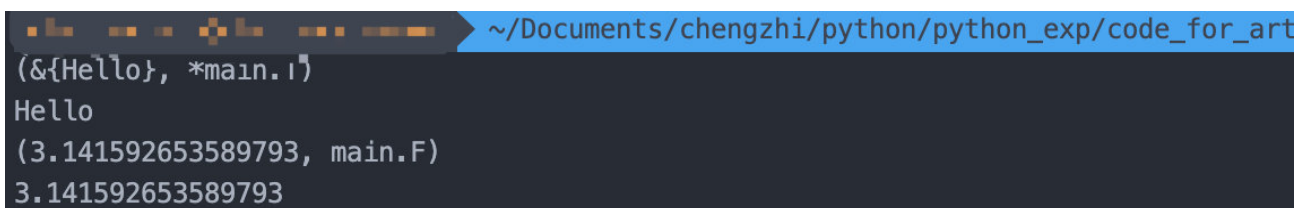
```

27     var i I
28
29     i = &T{"Hello"}
30     describe(i)
31     i.M()
32
33     i = F(math.Pi)
34     describe(i)
35     i.M()
36 }
37
38 func describe(i I) {
39     fmt.Printf("(%v, %T)\n", i, i)
40 }

```

在上面的代码当中定义了一个叫做describe的方法，在这个方法当中我们输出了两个值，一个是接口i对应的值，另一个是**接口i的类型**。

我们输出的结果如下：



```

~/Documents/chengzhi/python/python_exp/code_for_art
(&{Hello}, *main.I)
Hello
(3.141592653589793, main.F)
3.141592653589793

```

image-20200724084346988

可以看到接口当中既存储了对应的结构体的实例的信息，也存储了**结构体的类型**。因此interface可以理解成一种特殊的类型。

实际上也的确如此，我们可以把interface理解成一种**万能数据类型**，它可以接收任何类型的值。我们看下下面这种用法：

```

1  var a1 interface{} = 1
2  var a2 interface{} = "abc"
3  list := make([]interface{}, 0)
4  list = append(list, a1)
5  list = append(list, a2)
6  fmt.Println(list)

```

在代码当中我们创建了一个interface{}类型的slice，它可以接收任何类型的值和实例。另外我们用interface{}这个类型也可以接收任何结构体的值。这里可能会有些迷惑，其实很容易想明白。interface表示一种类型，可以接收任何实现了interface当中规定的方法的类型的值。当我们定义interface{}的时候，其实是定义了**空的interface**，相当于不需要实现任何方法的空interface，所以任何类型都可以接收，这也就是它成为万能类型的原因。

我们接收当然没有问题，问题是我们怎么使用这些interface类型的值呢？

一种方法是我们可以**判断一个interface的变量类型**。判断的方法非常简单，我们在interface的变量后面用.(type)的方法来判断。它和map的key值判断一样，会返回一个值和bool类型的标记。我们可以通过这个标记判断这个类型是否正确。

```
1  if v, ok := a1.(int); ok {
2      fmt.Println(v)
3  }
```

如果类型比较多的话使用switch也是可以的：

```
1  switch v := i.(type) {
2  case int:
3      fmt.Println("int")
4  case string:
5      fmt.Println("string")
6  }
```

空值nil

interface类型的空值是nil，和Python当中的None是一个意思，表示一个指针指向空。如果我们在Java或者是其他语言当中对一个空指针调用方法，那么会触发NullPointerException，也就是空指针报错。这也是我们初学者在编程当中最容易遇到的错误，往往原因是忘记了对声明进行初始化导致的。

但是在golang当中不会，即使是**nil也可以调用interface的方法**。举个例子：

```

1  type T struct {
2      S string
3  }
4
5  func (t *T) M() {
6      fmt.Println(t.S)
7  }
8
9  func main() {
10     var i I
11     var t *T
12     i = t
13     i.M()
14 }

```

我们将t赋值给了i，问题是t并没有进行初始化，所以它是一个nil，那么我们的i也会是一个nil。我们对nil调用M方法，在M方法当中我们打印了t的局部变量S。由于t此刻是一个nil，它并没有这个变量，所以会引发一个invalid memory address or nil pointer dereference的错误，也就是对空指针进行寻址的错误。

要解决这个错误，其实很简单，我们可以**在M方法当中对t进行判断**，如果发现t是一个nil，那么我们则跳过执行的逻辑。当我们把M函数改成这样之后，就不会触发空指针的问题了。

```

1  func (t *T) M() {
2      if t == nil {
3          fmt.Println("nil")
4          return
5      }
6      fmt.Println(t.S)
7  }

```

nil触发异常的问题也是初学者经常遇到的问题之一，这也要求我们在实现结构体内方法的时候一定要记得判断调用的对象是否为nil，避免不必要的问题。

赋值的类型选择

我们都知道golang当中通过interface来实现多态，只要是实现了interface当中定义的函数，那么我们就可以将对应的实例赋值给这个interface类型。

这看起来没有问题，但是在实际执行的时候仍然会有一点点小小的问题。比如说我们有这样一段代码：

```
1  type Integer int
2
3  type Operation interface {
4      Less(b Integer) bool
5      Add(b Integer)
6  }
7
8
9  func (a Integer) Less(b Integer) bool {
10     return a < b
11 }
12
13 func (a *Integer) Add(b Integer) {
14     *a += b
15 }
```

这段代码非常简单，我们定义了一个Operation的interface，并且实现了Integer类型的两个方法。表面上看一切正常，但是有一个细节。**Less和Add这两个方法针对的类型是不同的**，Less方法我们不需要修改原值，所以我们传入的是Integer的值，而Add方法，我们需要修改原值，所以我们传入的类型是Integer的指针。

那么问题来了，这两个方法的类型不同，我们还可以将它的值赋值给Operation这个interface吗？如果可以的话，我们应该传递的是值还是指针呢？下面代码当中的第二行和第三行究竟哪个是正确的呢？

```
1  var a Integer = 1
2  var b Operation = &a
3  var b Operation = a
```

答案是第二行的是正确的，原因也很简单，因为我们传入指针之后，golang的编译器会自动生成一个新的Less方法。在这个转换了类型的方法当中去调用了原本的方法，相当于做了一层中转。


```
1 func (a *Integer) Less(b Integer) bool{
2     return (*a).Less(b)
3 }
```

那反过来行不行呢？我们也写出代码：

```
1 func (a Integer) Add (b Integer) {
2     (&a).Add(b)
3 }
```

显然这样是不行的，因为函数执行之后修改的只能是Add这个方法当中a这个参数的值，而没办法修改原值。这和我们想要的不符合，所以golang没有选择这种策略。

总结

在今天的文章当中我们介绍了golang当中interface的一些高级用法，比如将它作为万能类型来接收各种格式的值。比如interface的空指针调用问题，以及interface中的两个函数接收类型不一致的问题。

也就是说在go语言当中，interface既是一种多态实现的规范，又有全能类型这样衍生的功能，这个设计的确是很惊艳的。对interface的熟练使用可以在一些问题当中大大降低我们编码的复杂度，以及运行的效率。这也是golang的原生优势之一。

相关阅读

面向对象回顾，golang中多态的实现方法

今天的文章到这里就结束了，如果喜欢本文的话，请来一波**素质三连**，给我一点支持吧（关注、在看、点赞）。



Go语言 | CSP并发模型与Goroutine的基本使用

今天是golang专题的第13篇文章，我们一起来聊聊golang当中的并发与Goroutine。

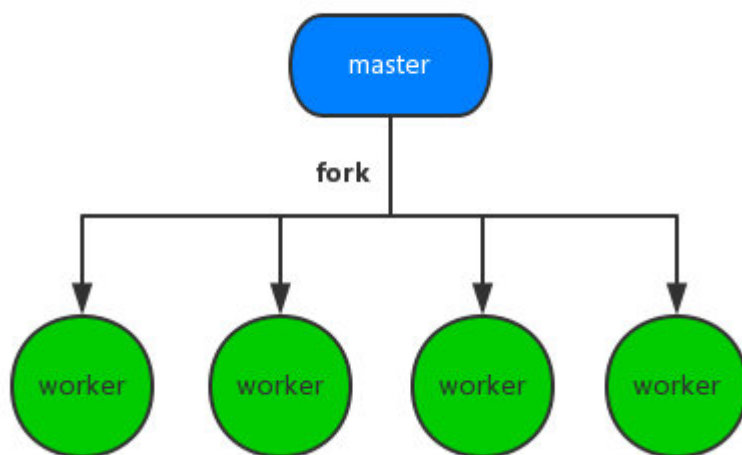
在之前的文章当中我们介绍完了golang当中常用的使用方法和规范，在接下来的文章当中和大家聊聊golang的核心竞争力之一，**并发模型与Goroutine**。

我们都知道并发是提升资源利用率最基础的手段，尤其是当今大数据时代，流量对于一家互联网企业的重要性不言而喻。串流显然是不行的，尤其是对于web后端这种流量的直接载体。并发是一定的，问题在于怎么执行并发。常见的并发方式有三种，分别是多进程、多线程和协程。

并发实现模型

多进程

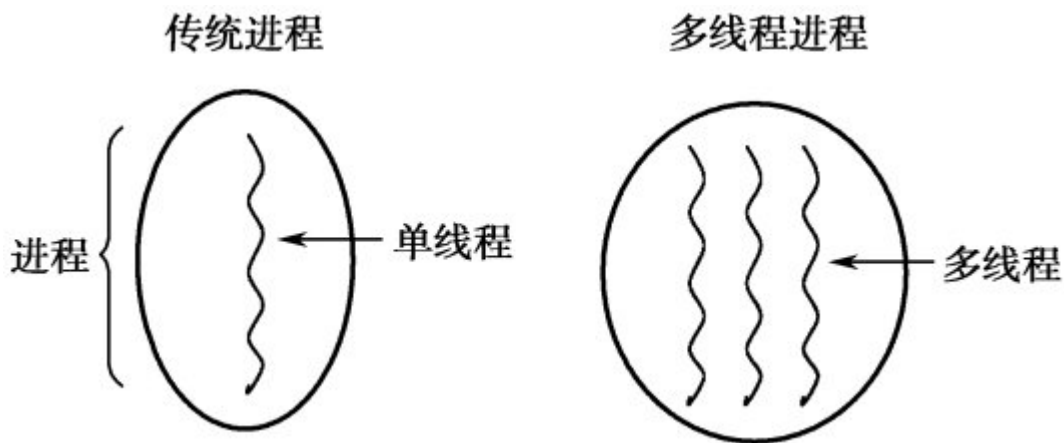
在之前的文章当中我们曾经介绍过，进程是操作系统资源分配的最小单元。所以多进程是在操作系统层面的并发模型，因为所有的进程都是有操作系统的内核管理的。所以每个进程之间是独立的，每一个进程都会有自己单独的内存空间以及上下文信息，一个进程挂了不会影响其他进程的运行。这个也是多进程最大的优点，但是它的缺点也很明显。



最大的缺点就是开销很大，创建、销毁进程的开销是最高的，远远高于创建、销毁线程。并且由于进程之间互相独立，导致进程之间通信也是一个比较棘手的问题，进程之间共享内存也非常不方便。因为这些弊端使得在大多数场景当中使用多进程都不是一个很好的做法。

多线程

多线程是目前最流行的并发场景的解决方案，由于线程更加轻量级，创建和销毁的成本都很低。并且线程之间通信以及共享内存非常方便，和多进程相比开销要小得多。

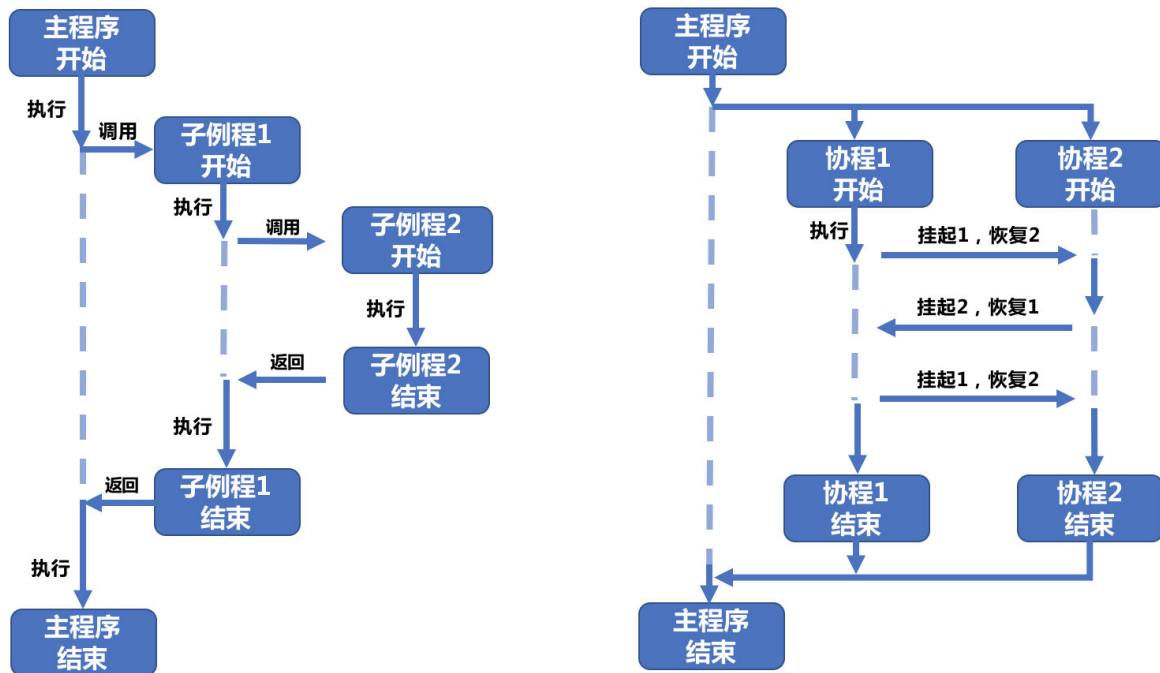


但是多线程也有缺点，一个缺点也是**开销**。虽然线程的开销要比进程小得多，但是如果创建和销毁频繁的话仍然是不小的负担。针对这个问题诞生了线程池这种设计。创建一大批线程放入线程池当中，需要用的时候拿出来使用，用完了再放回，回收和领用代替了创建和销毁两个操作，大大提升了性能。另外一个是**资源的共享**，由于线程之间资源共享更加频繁，所以在一些场景当中我们需要加上锁等设计，避免并发带来的数据紊乱。以及需要避免死锁等问题。

协程

也叫做**轻量级线程**，本质上仍然是线程。相比于多线程和多进程来说，协程要小众得多，相信很多同学可能都没有听说过。和多线程最大的区别在于，协程的调度**不是基于操作系统的而是基于程序的**。

也就是说协程更像是程序里的函数，但是在执行的过程当中可以随时挂起、随时继续。



我们举个例子，比如这里有两个函数：

```
1 def A():
2     print '1'
3     print '2'
4     print '3'
5
6 def B():
7     print 'x'
8     print 'y'
9     print 'z'
```

如果我们在一个线程内执行A和B这两个函数，要么先执行A再执行B要么先执行B再执行A。输出的结果是确定的，但如果我们用协程来执行A和B，有可能A函数执行了一半刚输出了一条语句的时候就转而去执行B，B输出了一条又再回到A继续执行。不管执行的过程当中发生了几次中断和继续，**在操作系统当中执行的线程都没有发生变化**。也就是说这是程序级的调度。

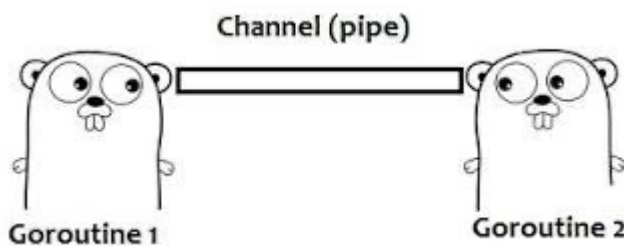
那么和多线程相比，我们创建、销毁线程的开销就完全没有了，整个过程变得非常灵活。但是缺点是由于是程序级别的调度，所以**需要编程语言自身的支持**，如果语言本身不支持，就很难使用了。目前原生就支持协程的语言并不多，显然golang就是其中一个。

共享内存与CSP

我们常见的多线程模型一般是通过共享内存实现的，但是共享内存就会有很多问题。比如**资源抢占的问题、一致性问题**等等。为了解决这些问题，我们需要引入多线程锁、原子操作等等限制来保证程序执行结果的正确性。

除了共享内存模型之外，还有一个经典模型就是CSP模型。CSP模型其实并不新，发表已经好几十年了。CSP的英文全称是Communicating Sequential Processes，翻译过来的意思是通信顺序进程。CSP描述了并发系统中的互动模式，是一种面向并发的语言的源头。

Golang只使用了CSP当中关于Process/Channel的部分。简单来说Process映射Goroutine，Channel映射Channel。Goroutine即Golang当中的协程，Goroutine之间没有任何耦合，可以完全并发执行。Channel用于给Goroutine传递消息，保持数据同步。虽然Goroutine之间没有耦合，但是它们与Channel依然存在耦合。



整个Goroutine和Channel的结构有些**类似于生产者消费者模式**，多个线程之间通过队列共享数据，从而保持线程之间独立。这里不过多深入，我们大概有一个印象即可。

Goroutine

Goroutine即**golang当中的协程**，这也是golang这门语言的核心精髓所在。正是因为Goroutine，所以golang才叫做golang，所以人们才选择golang。

相比于Java、Python等多线程的复杂的使用体验而言，golang当中的Goroutine的使用非常简单，简单到爆表。只需要一个关键字就够了，那就是go。所以你们应该明白为什么golang叫做Go语言不叫别的名字了吧？

比如我们有一个函数：

```
1 func Add(x, y int) int{
2     z := x + y
3     fmt.Println(z)
4 }
```

我们希望启动一个goroutine去执行它，应该怎么办？很简单，只需要一行代码：

```
go Add(3, 4)
```

我们还可以用go关键字来使用goroutine来执行一个匿名函数：

```
1 go func(x, y int) {
2     fmt.Println(x + y)
3 }(3, 4)
```

需要注意的是，当我们使用go关键字的时候，是**不能获取返回值的**。也就是说`z := go Add(3, 4)`是违法的。乍看起来似乎不合理，但是道理其实是很简单的。如果我们希望一个变量承接一个函数的返回值，说明这里的逻辑是串行的，那么我们使用goroutine的意义是什么？所以这里看似不合理，其实是设计者下了心思的。

总结

关于并发模型与goroutine的基本原理就介绍到这里了，goroutine的使用方法我们已经了解了，但是还有很多问题没有解决。比如**多个goroutine之间怎么通信**，我们**如何知道goroutine的执行状态**，当我们创建多个goroutine的时候，我们怎么知道goroutine都结束没有？

关于这些问题，我们将会在今后的文章当中给大家分享，敬请期待。

今天的文章到这里就结束了，如果喜欢本文的话，请来一波**素质三连**，给我一点支持吧（**关注、在看、点赞**）。



Golang | 简介channel常见用法，完成goroutine通信

今天是golang专题的第14篇文章。

今天来看看golang当中另一个很重要的概念——**信道**。我们之前介绍goroutine的时候曾经提过一个问题，当我们启动了多个goroutine之后，我们怎么样让goroutine之间保持通信呢？

要回答这个问题就需要用到信道。

channel

信道的英文是channel，在golang当中的关键字是chan。它的用途是用来**在goroutine之间传输数据**，这里你可能要问了，为什么一定得是goroutine之间传输数据呢，函数之间传递不行吗？

因为正常的传输数据直接以参数的形式传递就可以了，只有在并发场景当中，多个线程彼此隔离的情况下，才需要一个特殊的结构传输数据。

Chan看起来比较怪，在其他语言当中基本没有出现过，但是它的原理和使用都非常简单。

我们先来看它的使用，首先是定义一个chan，还是老规矩，通过**make关键字**创建。我们之前也提过，golang当中的一个设计原则就是能省则省，能简单则简单。从这个make关键字就看得出来，它可以创建的东西太多了，既可以创建一个切片，也可以创建map，还可以创建信道。

所以当我们创建一个chan的时候，可以通过make实现。

```
Ch := make(chan int)
```

我们在chan后面跟上一个类型，表示这个信道**传输的数据类型**。如果你想要传输任何类型呢，那可以用我们之前说过的interface{}。

Chan创建了之后，我们想要从其中获取数据或者是把数据放入其中也非常简单，简单到都没有api，直接用形象的传输语句就可以了。

比如我们现在有一个chan是ch，我们想要放入数据，我们可以这样ch <- a。我们想要从ch当中获取数据，我们可以v := <- ch。

我们用箭头表示数据的流动，是不是很形象很直观呢？

阻塞

但是还没完，chan有一个很关键的点在于，chan的使用是**阻塞**的。也就是说下游从chan当中拿走一个数据我们才可以传入一个数据。否则的话，传输数据的代码就会一直等待chan清空。

同样，如果我们定义了一个从chan当中读取数据的语句，假如当前的chan是空的话，那么它也会一直阻塞等待，直到chan当中有数据了为止。

所以我们就知道了，chan的使用场景当中**需要一个生产方，也需要一个消费方**。我们来看一个golang官方的一个例子：

```
1 package main
2
3 import "fmt"
4
5 func sum(s []int, c chan int) {
6     sum := 0
7     for _, v := range s {
8         sum += v
9     }
10    c <- sum // 将和送入 c
11 }
12
13 func main() {
14     s := []int{7, 2, 8, -9, 4, 0}
15
16     c := make(chan int)
17     go sum(s[:len(s)/2], c)
18     go sum(s[len(s)/2:], c)
19     x, y := <-c, <-c // 从 c 中接收
20
21     fmt.Println(x, y, x+y)
22 }
```

我们启动了两个goroutine去对数组进行求和并进行返回，goroutine生产的数据是没办法直接return的，所以只能通过chan的形式传输出来。chan传输出来需要下游消费，所以上面两个goroutine的数据会传输到x, y: <-c, <-c 这一句语句当中。

前面说过了，chan的传输是阻塞的，所以这一句语句会一直等待，直到上面两个goroutine都计算完成了为止。

如果你看的有些发蒙，觉得好似有些理解了又好似没有的话，那么很简单的一个办法是在理解的时候把这个使用场景做一个变幻。把chan的使用场景想象成我们之前介绍过的**生产者消费者设计模式**，chan在其中扮演的角色其实就是队列。

生产者往队列当中传输数据，消费者进行消费，唯一不同的是这个队列的容量是1，必须要生产和消费端都准备就绪了才会进行数据传输。

chan的缓冲

前文说了，chan的容量只有1，只有消费端和生产端都就绪的时候才可以传输数据。我们也可以给chan**加上缓冲**，如果消费端来不及把所有的数据都消费完，允许生产端先把数据暂时存在chan当中，先不发生阻塞，只有在chan满了之后才会阻塞。

用法也很简单，我们在通过make创建chan的时候多加上一个参数表示容量即可，和我们之前创建切片的道理很类似。

```
Ch := make(chan int, 100)
```

比如这样，我们就创建了一个缓冲区为100的信道。

但多说一句，其实这种情况不太常用，原因也很简单。因为**上下游的消费情况是统一的**，如果生产者生产的速度过快，而消费端跟不上的话，即使把它先暂存在缓冲区当中也没什么用，早晚还是会要阻塞的。

close

当我们对信道使用结束之后，可以通过close语句将它关闭。

Close这个操作**只能在生产端进行**，消费端如果close信道会引发一个panic。我们在从chan接收数据的时候，可以加上一个参数判断信道是否关闭。

```
1 v, ok := <- ch
2 if !ok {
3     return
4 }
```

这样我们就可以判断chan关闭的时间了。

今天的文章到这里就结束了，如果喜欢本文的话，请来一波**素质三连**，给我一点支持吧（**关注、在看、点赞**）。



Go语言 | goroutine不只有基础的用法，还有这些你不知道的操作

今天是golang专题第15篇文章，我们来继续聊聊channel的使用。

在我们的上篇文章当中我们简单介绍了golang当中channel的使用方法，channel是golang当中一个非常重要的设计，可以理解为生产者消费者模式当中的队列。但channel和队列不一样的是，golang当中集成了一些其他的用法，使得我们的使用更加灵活，开发并发相关的功能更加简单。

select机制

我们来思考一个问题，假设我们的数据源有多个，也就是说我们可能会从多个入口获取数据，但是我们并不知道这些数据源当中哪个先把数据准备好。我们希望实现轮询这些channel，哪个数据准备好了就读取哪个，否则就阻塞等待，这个功能应该怎么办呢？

我们当然可以自己用循环来实现，但是这显然是不合理的，golang当中针对这个问题提供了专门的解决方法，它就是select关键字。

select机制并不是golang这门语言独创的，早在Unix时代就有了select机制。通过select函数监控一系列文件的句柄，一旦其中一个发生了改动，select函数就会返回。而golang当中的select则用来在channel当中进行选择，有点像是switch，写出来的代码大概是这个样子：

```
1 select {
2     case <- chan1:
3     case chan2 <- 1:
4     default:
5 }
```

select后面跟多个case以及default，其中default并不是必须的。case后面必须要接一个chan的操作，可以从一个chan当中读入数据，也可以是向一个chan当中写入数据。如果所有的case都不成功，则会进入default语句当中，如果没有default语句，那么select会陷入阻塞。

一般情况下我们使用select是为了从多个数据源中获取数据，当多个chan同时有数据的时候，使用select可以让我们避免判断哪个数据源数据ready的问题。

range机制

我们之前在介绍slice遍历的时候曾经介绍过range机制，我们可以通过range来遍历一个数组或者是map。就像是这样：

```
1 arr := make([]int, 0)
2 for i := range arr {
3     // do something
4 }
5
6 mp := make(map[string]int)
7
8 for k, v := range mp {
9     // do something
10 }
```

很多时候我们会把这个用法当做是迭代器的迭代，就像是Java和Python中的那样。但实际上range机制的**底层原理是chan**，当我们使用range的时候，它表示会不断地从chan当中接受值，直到它关闭。

所以我们可以这样遍历一个chan当中的数据：

```
1 ch := make(chan int)
2
3 for c := range ch {
4     // do something
5 }
```

超时机制

有没有想过一个问题，channel的写入和写出都是阻塞的，也就是说如果是从chan当中读取数据，必须要上游已经传输了才可以读取到。同样，如果往没有缓冲区的chan写入数据也需

要下游消费了才能写入成功。**阻塞往往是有很大隐患的**，如果处理不好很容易导致整个程序锁死。

我们需要设计机制来解决这个问题，比较好的方案就是设置定时器，如果超过一定的时间chan还没有响应成功的话，那么就人工停止程序。这一点说起来还有点麻烦的，比如我们要启动一个定时器，要手动终止goroutine，但是结合select机制其实并不难实现，我们来看代码。

```
1  timeout := make(chan bool)
2  go func() {
3      time.Sleep(1e9)
4      timeout <- true
5  }()
6
7
8  select {
9      case <- ch:
10         // do something
11      case <- timeout:
12         // break
13  }
```

说白了很简单，也就是我们**额外启动一个goroutine做休眠操作**，当休眠结束之后也通过chan发送消息，这样如果我们select先接受到了timeout的信号就说明程序已经超时了。当然这只是一个很简单的demo，实际使用的话需要考虑的情况可能还会更多。

channel传递

有没有想过一个问题，既然chan可以传输任何类型的数据，那么我们能不能用一个chan传输一个chan呢？

这样的操作是可以的，因为在有些场景当中相比于直接把数据传输给下游，我们**传输读取数据的chan可能更加方便**。有点授人以渔的意思，更加厉害的是我们可以**结合函数式编程**，把处理数据的函数一并传输给下游。这样下游读取到数据，并且用读取到的处理函数来处理，这样可以更加定制化，如果以后数据和处理方式都发生改动，也只需要在上游修改，可以更加解耦。

我们同样来看一个demo：

```

1  type MetaData struct {
2      value interface{}
3      handler func(interface{}) int
4      downstream chan interface{}
5  }
6
7
8  func handle(queue chan *MetaData) {
9      for data := range queue {
10         data.downstream <- data.handler(data.value)
11     }
12 }

```

这只是一个简单的案例，想要在实际应用当中真的使用上还需要定义大量的接口以及做很多设计。我们只需要知道有这么一种设计模式和用法就可以了。

单向channel

最后，我们来说说单向channel，也就是说我们**指定channel是只读的或者是只写的**。但其实这是一个伪命题，原因也很简单，如果只写数据没人读，或者是只读但是不能写，那么这个channel有什么用呢？只有有人读有人写才可以完成数据流通不是吗？

的确如此，所以这里所说的单向channel其实**并不是真正意义上的单向**，只是说我们为了规范，对使用方进行限制。比如说我们限定在消费函数当中不能写入，在生产函数当中不能消费。我们在通过函数传递chan的时候，可以通过加上限定让chan在函数当中变成单向的。

```

1  var ch chan <- float32 // 只写chan
2  var ch <- chan float32 // 只读chan

```

除此以外我们还可以把一个正常的chan转化成单向的chan：

```

1  var ch chan int
2  ch1 := <- chan int(ch)
3  ch2 := chan <- int(ch)

```


我们一般不在程序当中做这样的转化，而是用在函数参数当中，这也主要是为了起到规范的作用。

```
1 func Test(ch <- chan int) {  
2     for val := range ch {  
3         // do something  
4     }  
5 }
```

今天的文章到这里就结束了，如果喜欢本文的话，请来一波**素质三连**，给我一点支持吧（**关注、在看、点赞**）。



Go语言 | 并发设计中的同步锁与waitgroup用法

今天是golang专题的第16篇文章，我们一起来聊聊golang当中的并发相关的一些使用。

虽然关于goroutine以及channel我们都已经介绍完了，但是关于并发的机制仍然没有介绍结束。只有goroutine以及channel有时候还是不足以完成我们的问题，比如多个goroutine**同时访问一个变量**的时候，我们怎么保证这些goroutine之间不会互相冲突或者是影响呢？这可能需要我们对资源进行加锁或者是采取其他的操作了。

同步锁

golang当中提供了两种常用的锁，一种是sync.Mutex另外一种sync.RWMutex。我们先说说Mutex，它就是**最简单最基础的同步锁**，当一个goroutine持有锁的时候，其他的goroutine只能等待到锁释放之后才可以尝试持有。而RWMutex是读写锁的意思，它**支持一写多读**，也就是说允许支持多个goroutine同时持有读锁，而只允许一个goroutine持有写锁。当有goroutine持有读锁的时候，会阻止写操作。当有goroutine持有写锁的时候，无论读写都会被堵塞。

我们使用的时候需要根据我们场景的特性来决定，如果我们的场景是读操作多过写操作的场景，那么我们可以使用RWMutex。如果是写操作为主，那么使用哪个都差不多。

我们来看下使用的案例，假设我们当前有多个goroutine，但是我们只希望持有锁的goroutine执行，我们可以这么写：

```
1  var lock sync.Mutex
2
3  for i := 0; i < 10; i++ {
4      go func() {
5          lock.Lock()
6          defer lock.Unlock()
7          // do something
8      }()
9  }
```

虽然我们用for循环启动了10个goroutine，但是由于互斥锁的存在，**同一时刻只能有一个goroutine在执行**。

RWMutex区分了读写锁，所以我们一共会有4个api，分别是Lock, Unlock, RLock, RUnlock。Lock和Unlock是写锁的加锁以及解锁，而RLock和RUnlock自然就是读锁的加锁和解锁了。具体的用法和上面的代码一样，我就不多赘述了。

全局操作一次

在一些场景以及一些设计模式当中，会要求我们某一段代码只能执行一次。比如很著名的**单例模式**，就是将我们经常使用的工具设计成单例，无论运行的过程当中初始化多少次，得到的都是同一个实例。这样做的目的是减去创建实例的时间，尤其是像是数据库连接、hbase连接等这些实例创建的过程非常的耗时。

那我们怎么在golang当中实现单例呢？

有些同学可能会觉得这个很简单啊，我们只需要用一个bool型变量判断一下初始化是否有完成不就可以了么？比如这样：

```
1  type Test struct {}
2  var test Test
3  var flag = false
4
5  func init() Test{
6      if !flag {
7          test = Test{}
8          flag = true
9      }
10     return test
11 }
```

看起来好像没有问题，但是仔细琢磨就会发现不对的地方。因为**if判断当中的语句并不是原子的**，也就是说有可能同时被很多goroutine同时访问。这样的话有可能test这个变量会被多次初始化并且被多次覆盖，直到其中一个goroutine将flag置为true为止。这可能会导致一开始访问的goroutine获得的test都各不相同，而产生未知的风险。

要实现单例其实很简单，sync库当中为我们提供了现成的工具once。它可以传入一个函数，**只允许全局执行这个函数一次**。在执行结束之前，其他goroutine执行到once语句的时

候会被阻塞，保证只有一个goroutine在执行once。当once执行结束之后，再次执行到这里的时候，once语句的内容将会被跳过，我们来结合一下代码来理解一下，其实也非常简单。

```
1  type Test struct {}
2  var test Test
3
4  func create() {
5      test = Test{}
6  }
7
8  func init() Test{
9      once.Do(create)
10     return test
11 }
```

waitgroup

最后给大家介绍一下waitgroup的用法，我们在使用goroutine的时候有一个问题是我们在**主程序当中并不知道goroutine执行结束的时间**。如果我们只是要依赖goroutine执行的结果，当然可以通过channel来实现。但假如我们明确地希望等到goroutine执行结束之后再执行下面的逻辑，这个时候我们又该怎么办呢？

有人说可以用sleep，但问题是我们并不知道goroutine执行到底需要多少时间，怎么能事先知道需要sleep多久呢？

为了解决这个问题，我们可以使用sync当中的另外一个工具，也就是waitgroup。

waitgroup的用法非常简单，只有三个方法，一个是**Add**，一个是**Done**，最后一个是**Wait**。其实waitgroup内部存储了当前有多少个goroutine在执行，当调用一次Add x的时候，表示当下同时产生了x个新的goroutine。当这些goroutine执行完的时候，我们让它调用一下Done，表示执行结束了一个goroutine。这样当所有goroutine都执行完Done之后，wait的阻塞会结束。

我们来看一个例子：

```
1  sample := Sample{}
2
```

```
3  wg := sync.WaitGroup{}
4
5  go func() {
6      // 增加一个正在执行的goroutine
7      wg.Add(1)
8      // 执行完成之后Done一下
9      defer wg.Done()
10     sample.JoinUserFeature()
11 }()
12
13 go func() {
14     wg.Add(1)
15     defer wg.Done()
16     sample.JoinItemFeature()
17 }()
18
19 wg.Wait()
20 // do something
```

总结

上面介绍的这些工具和库都是我们日常在并发场景当中经常使用的，也是一个golang工程师**必会的技能之一**。到这里为止，关于golang这门语言的基本功能介绍就差不多了，后面将会介绍一些实际应用的内容，敬请期待吧。

今天的文章到这里就结束了，如果喜欢本文的话，请来一波**素质三连**，给我一点支持吧（**关注、在看、点赞**）。

