学一门语言之前，我们要弄清楚这门语言的优势是什么。

为什么要学Go，Go的特点和优势是什么？

在《Go语言编程》一书中，作者是这样介绍Go的：“并发与分布式，多核化和集群化是互联网时代的典型特征，那语言需要哪些特性来应对这些特征呢？

第一个话题是并发执行的“执行体”。执行体是个抽象的概念，在操作系统层面有多个概念与之对应，比如操作系统自己掌管的进程（process）、进程内的线程（thread）以及进程内的协程（coroutine，也叫轻量级线程）。多数语言在语法层面并不直接支持协程，而通过库的方式支持的协程的功能也并不完整，比如仅仅提供协程的创建、销毁与切换等能力。如果在这样的协程中调用一个同步IO操作，比如网络通信、本地文件读写，都会阻塞其他的并发执行协程，从而无法真正达到协程本身期望达到的目标。

Go语言在语言级别支持协程，叫goroutine。Go语言标准库提供的所有系统调用（syscall）操作，当然也包括所有同步IO操作，都会出让CPU给其他goroutine，这让事情变得非常简单。”

这点话就充分的体现出 Go 的优势：它从语言层面提供能够并行的goroutine协程。

在学python的coroutine协程的时候，我们已经充分的尝到了协程的甜头，相比于多进程和多线程并发，python可以通过协程+事件循环的方式做到让单线程不断切换协程做到并发。

而且协程的创建远比线程和进程的创建所需的开销小，且协程的切换远比进程和线程的速度快，因此一个单线程可以开启成千上万个协程进行高并发。而多进程和多线程只能开启最多几十个。

但是python的coroutine协程也有它的局限性：

1.coroutine协程只能并发不能并行

单线程内创建的多个coroutine协程只能并发不能并行，所以只用到了单核。如果想利用到多核，就必须开多进程，并在每个进程中创建多个协程（由于GIL锁，python的多线程也只用到了单核，开多进程才能用到多核）。

2.coroutine协程中的IO操作必须是异步非阻塞的IO操作

在python的协程中，由于协程是并发的，其本质是一个单线程在调度这些协程，所以一旦使用同步的IO操作就会阻塞住整个线程，这样所有的协程都会被卡住，无法做到高并发。而如果要使用非阻塞的IO操作，就必须使用一些支持异步操作的库(如aiohttp,aiomysql,aioredis)来代替我们以前所熟悉的同步操作的库(requests,pymysql,redis)。增加了理解和学习成本。

Go的goroutine也是一种协程，不过它是一种比较特殊的协程，它和coroutine协程的共同特点是：

A.两者都是有用户程序创建的并且由用户程序制定算法来进行调度，而进程和线程则是由操作系统进行创建和调度的，用户程序不需要关心内核如何去调度进程和线程。

B.它们相比于进程和线程都是轻量级的，创建和切换的开销远比进程线程的创建切换小。

它们的不同点在于：

A.goroutine可以做到并行，能利用到多核。coroutine只能并发，只用到单核

B.正由于goroutine能并行，所以在goroutine中能够使用同步的IO操作，这样即使同步IO把某一个goroutine执行体阻塞了，也会跳到其他核下的goroutine执行任务。

下面再说说goroutine的通信。作者是这样写道的：“

执行体间的通信包含2个方式：‘执行体之间的互斥与同步’ 和‘执行体之间的消息传递’

先说“执行体之间的互斥与同步”。当执行体之间存在共享资源（一般是共享内存）时，为保证内存数据的安全和一致性，需要对访问该共享资源的相关执行体进行互斥（通过加锁等机制）。当多个执行体之间的逻辑存在时序上的依赖时，也往往需要在执行体之间进行同步（通过条件变量等机制）。互斥与同步是执行体间最基础的交互方式。

多数语言在库层面提供了线程间的互斥与同步支持，那么协程之间的互斥与同步呢？呃，不好意思，没有。事实上多数语言标准库中连协程都是看不到的。

再说“执行体之间的消息传递”。在并发编程模型的选择上，有两个流派，一个是共享内存模型，一个是消息传递模型。多数传统语言选择了前者，少数语言选择后者，其中选择“消息传递模型”的最典型代表是Erlang语言。业界有专门的术语叫“Erlang风格的并发模型”，其主体思想是两点：一是“轻量级的进程（Erlang中‘进程’这个术语就是我们上面说的‘执行体’）”，二是“消息乃进程间通信的唯一方式”。当执行体之间需要相互传递消息时，通常需要基于一个消息队列（message queue）或者进程邮箱（process mail box）这样的设施进行通信。

Go语言推荐采用“Erlang风格的并发模型”的编程范式，尽管传统的“共享内存模型”仍然被保留，允许适度地使用。在Go语言中内置了消息队列的支持，只不过它叫通道（channel）。两个goroutine之间可以通过通道来进行交互。”

上面这段话的重点是：Go语言中的goroutine协程之间的通信很少通过共享变量进行，而主要是通过一个消息队列的结构进行通信，在go语言中这种消息队列叫做通道(channel)。

总结起来一句话：Go语言的优势在于它在语言层面支持能并行的轻量级的goroutine协程使得Go以极高的效率和低开销做到高并发完成任务。

go语言的特性

自动垃圾回收

函数多返回值

匿名函数和闭包

类型（type）和接口（interface）

并发编程（goroutine和channels）

反射

更丰富的内置类型

错误处理

语言交互性

这里稍微先简单的介绍其中几个特性：

**自动垃圾回收**

我们可以先看下不支持垃圾回收的语言的资源管理方式，以下为一小段C语言代码：

void foo()

{

char \* p = new char [128]; //对p指向的内存块进行赋值

func1(p); //使用内存指针

delete [] p; // 释放p指向的内存

}

由于各种非预期的原因，比如由于开发者的疏忽导致最后的delete语句没有被调用，都会引发经典而恼人的内存泄露问题。假如该函数被调用得非常频繁，那么我们观察该进程执行时，会发现该进程所占用的内存会一直疯长，直至占用所有系统内存并导致程序崩溃，而如果泄露的是系统资源的话，那么后果还会更加严重，最终很有可能导致系统崩溃。

手动管理内存的另外一个问题就是由于指针的到处传递而无法确定何时可以释放该指针所指向的内存块。假如代码中某个位置释放了内存，而另一些地方还在使用指向这块内存的指针，那么这些指针就变成了所谓的“野指针”（wild pointer）或者“悬空指针”（dangling pointer），对

这些指针进行的任何读写操作都会导致不可预料的后果。

到目前为止，内存泄露的最佳解决方案是在语言级别引入自动垃圾回收算法（Garbage Collection，简称GC）。所谓垃圾回收，即所有的内存分配动作都会被在运行时记录，同时任何对该内存的使用也都会被记录，然后垃圾回收器会对所有已经分配的内存进行跟踪监测，一旦发现有些内存已经不再被任何人使用，就阶段性地回收这些没人用的内存进行释放。当然，因为需要尽量最小化垃圾回收的性能损耗，以及降低对正常程序执行过程的影响，现实 中的垃圾回收算法要比这个复杂得多。

**匿名函数和闭包**

在Go语言中，所有的函数也**是值类型，可以作为参数传递**。Go语言支持常规的匿名函数和闭包，比如下列代码就定义了一个名为f的匿名函数，开发者可以随意对该匿名函数变量进行传递和调用

f := func (x, y int ) int {

return x + y

}

**并发编程**

Go语言引入了goroutine概念，通过使用goroutine而不是裸用操作系统的并发机制，以及使用**消息传递**来共享内存而不是使用共享内存来通信，Go语言让并发编程变得更加**轻盈和安全（开销小而且数据一致性有保障）**。

通过在函数调用前使用关键字go，我们即可让该函数以goroutine方式执行。goroutine是一种比线程更加轻盈、更省资源的协程。Go语言通过系统的线程来多路派遣这些函数的执行，使得每个用go关键字执行的函数可以运行成为一个单位协程。

当一个协程阻塞的时候，调度器就会自动把其他协程安排到另外的线程中去执行，从而实现了程序无等待并行化运行。而且调度的开销非常小，一颗CPU调度的规模不下于每秒百万次，这使得我们能够创建大量的goroutine。

Go语言实现了CSP（通信顺序进程，Communicating Sequential Process）模型来作为goroutine间的推荐通信方式。在CSP模型中，一个并发系统由**若干并行运行的顺序进程**组成（所以goroutine其实是多进程？），**每个进程不能对其他进程的变量赋值(**不同进程间的协程是互不干扰的独立的？**)**。进程之间只能通过一对通信原语实现协作。

Go语言用channel（通道）这个概念来轻巧地实现了CSP模型。**channel的使用方式比较接近Unix 系统中的管道（pipe）概念**， 可以方便地进行跨goroutine的通信。 另外，由于一个进程内创建的所有goroutine运行在同一个内存地址空间中，因此如果同一进程中不同的 goroutine不得不去访问共享的内存变量，访问前应该先获取相应的读写锁。Go语言标准库中的 sync包提供了完备的读写锁功能。

**反射**

反射（reflection）是在Java语言出现后迅速流行起来的一种概念。**通过反射，你可以获取对象类型的详细信息，并可动态操作对象**。反射是把双刃剑，功能强大但代码可读性并不理想。若非必要，我们并不推荐使用反射。

反射最常见的使用场景是做**对象的序列化**。 例如，Go语言标准库的encoding/json、encoding/xml、encoding/gob、 encoding/binary等包就大量依赖于反射功能来实现。

**语言交互性**

由于Go语言与C语言之间的天生联系，Go语言的设计者们自然不会忽略如何重用现有C模块的这个问题，这个功能直接被命名为Cgo。Cgo既是语言特性，同时也是一个工具的名称。 在Go代码中，可以按Cgo的特定语法混合编写C语言代码，然后Cgo工具可以将这些混合的C 代码提取并生成对于C功能的调用包装代码。开发者基本上可以完全忽略这个Go语言和C语言的边界是如何跨越的。

**安装Go**

在windows直接搜百度，按照指示安装即可。

在Linux中直接yum install golang 即可。Go是我见过的安装最简单的语言。

安装完之后，执行

go version

命令，可以查看go的版本。

编辑器和IDE推荐使用goland，不过需要破解，可以参考

<https://www.cnblogs.com/double12gzh/p/13680180.html>

接下来我们运行第一个go程序。

// hello.go

package main

import "fmt"

func main() {

fmt.Println("hello world")

}

首先要以目录的形式用goland打开hello.go所在的目录，然后设置 goPath 和 goRoot。

按 ctrl + shift + f10即可开始运行。

**代码解读:**

每个Go源代码文件的开头都是一个package声明，表示该Go代码所属的包(一个包就是一个目录，在文件第一行声明“package abc”表示这个文件属于abc这个包，基本上一个abc包下的所有文件的第一行都是package abc)。包是Go语言里最基本的分发单位，也是**工程管理**中依赖关系的体现。

要生成Go**可执行程序**，必须建立一个名字为main的包，并且在该包中包含一个叫main()的函数（该函数是Go 文件的执行起点）。 **Go语言的main()函数不能带参数，也不能定义返回值。**命令行传入的参数在os.Args变量中保存。如果需要支持命令行开关，可使用flag包。对于包不为main的文件，这些文件不能执行，只能被main文件引入，由main文件一起执行。

在包声明之后，是一系列的import语句，用于导入该程序所依赖的包。 由于本示例程序用 到了Println()函数，所以需要导入该函数所属的fmt包。 有一点需要注意，**不得包含在源代码文件中没有用到的包**，否则Go编译器会报编译错误。 这与下面提到的强制左花括号{的放置位置以及之后会提到的函数名的大小写规则，均体现了Go语言在语言层面解决软件工程问题的设计哲学。

所有函数（包括在对象编程中会提到的类型成员函数）以关键字 func开头。一个常规的 函数定义包含以下部分：

func 函数名(参数列表)(返回值列表) {

// 函数体

}

对应的一个实例如下：

func Compute(value1 int , value2 float64)(result float64, err error) {

// 函数体

}

Go支持多个返回值。以上的示例函数Compute()返回了两个值，一个叫 result，另一个是 err。**并不是所有返回值都必须赋值**。在函数返回时没有被明确赋值的返回值都会被设置为默认值，比如result会被设为0.0，err会被设为nil。

Go程序的代码注释与C++保持一致，即同时支持以下两种用法：

/\*

块注释

\*/

// 行注释

相信熟悉C和C++的读者也发现了另外一点，即在这段Go示例代码里没有出现分号。**Go 程序并不要求开发者在每个语句后面加上分号表示语句结束**。有些读者可能会自然地把左花括号{另起一行放置，这样做的结果是Go 编译器报告编译错误，这点需要特别注意。

**Go程序编译**

直接运行一个go代码

go run hello.go

使用这个命令，会将编译、链接和运行3个步骤合并为一步，运行完后

在当前目录下也看不到任何中间文件和最终的可执行文件。

如果我们只编译这个文件而不运行结果，可以使用build命令

go build hello.go

然后他就会生成一个hello的可执行的二进制文件

./hello # 执行这个二进制文件

从根本上说，Go命令行工具只是一个源代码管理工具，或者说是一个前端。真正的Go编译器和链接器是6g（编译器）和6l（编译器）。32位版本工具为8g和8l

工程管理

在实际的开发工作中，直接调用编译器进行编译和链接的场景是少而又少，因为在工程中不会简单到只有一个源代码文件，且源文件之间会有相互的依赖关系。如果这样一个文件一个文件逐步编译肯定不行。

为此出现了Go命令行工具，Go使用目录结构和包名来推导工程结构和构建顺序。

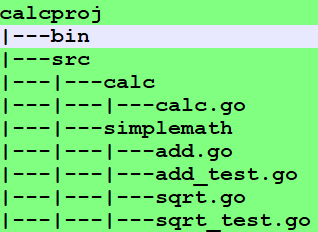
例如有一个工程项目，包含两个部分

1. 可执行程序，名为calc，内部只包含一个calc.go文

2. 算法库，名为simplemath，每个命令对应于一个同名的go文件，

比如calc add 1 2这个命令对应add.go文件

这个工程项目的结构如下，其中bin用来放编译后的二进制可执行文件。项目的go源码必须放在src目录中，否则go build 一个项目的时候会失败。



可执行程序 calc.go 的代码

//calc.go

package main

import "os" // 用于获得命令行参数os.Args

import "fmt"

import "simplemath"

import "strconv"

var Usage = func () {

fmt.Println("USAGE: calc command [arguments] ...")

fmt.Println("\nThe commands are:\n\tadd\tAddition of two values.\n\tsqrt\tSquare root of a non-negative value.")

}

func main() {

args := os.Args

if args == nil || len(args) < 2 {

Usage()

return

}

switch args[1] {

case "add":

if len(args) != 4 {

fmt.Println("USAGE: calc add <integer1><integer2>")

return

}

v1, err1 := strconv.Atoi(args[2])

v2, err2 := strconv.Atoi(args[3])

if err1 != nil || err2 != nil {

fmt.Println("USAGE: calc add <integer1><integer2>")

return

}

ret := simplemath.Add(v1, v2)

fmt.Println("Result: ", ret)

case "sqrt":

if len(args) != 3 {

fmt.Println("USAGE: calc sqrt <integer>")

return

}

v, err := strconv.Atoi(args[2])

if err != nil {

fmt.Println("USAGE: calc sqrt <integer>")

return

}

ret := simplemath.Sqrt(v)

fmt.Println("Result: ", ret)

default :

Usage()

}

}

Simplemath包的代码

// add.go

package simplemath

func Add(a int , b int ) int {

return a + b

}

// sqrt.go

package simplemath

import "math"

func Sqrt(i int ) int {

v := math.Sqrt( float64 (i))

return int (v)

}

// add\_test.go

package simplemath

import "testing"

func TestAdd1(t \*testing.T) {

r := Add(1, 2)

if r != 3 {

t.Errorf("Add(1, 2) failed. Got %d, expected 3.", r)

}

}

// sqrt\_test.go

package simplemath

import "testing"

func TestSqrt1(t \*testing.T) {

v := Sqrt(16)

if v != 4 {

t.Errorf("Sqrt(16) failed. Got %v, expected 4.", v)

}

}

编译前要做的事情：

1. 检查是否有这两个环境变量 GOROOT和GOPATH

其中 GOROOT 是你go的安装目录位置，GOPATH是你项目(calcproj)的目录位置

必须有这两个环境变量才能对项目进行编译。

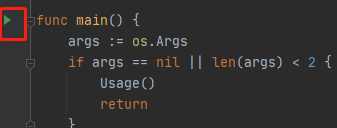
GOPATH也可以接受多个路径，路径和路径之间用冒号分割。

1. 调试代码

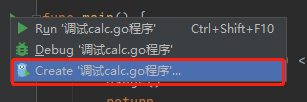
必须把代码调试到没有报错以及没有逻辑错误之后才能编译，不然到头来发现有错误还得重新编译。

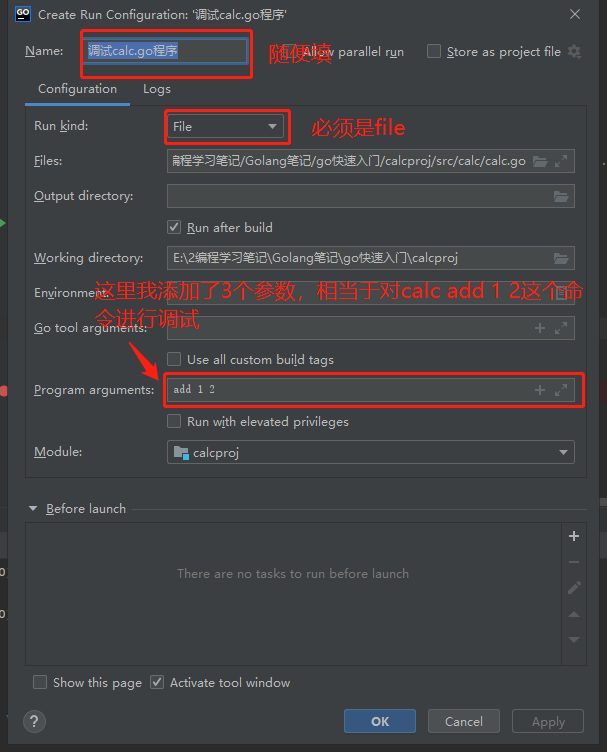
使用Goland的断点调试方法（带参数）如下：

首先，在可执行文件calc.go的main函数旁边，有一个小绿色箭头



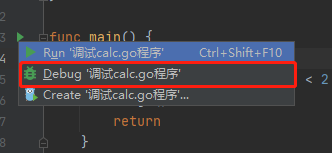
设置调试配置





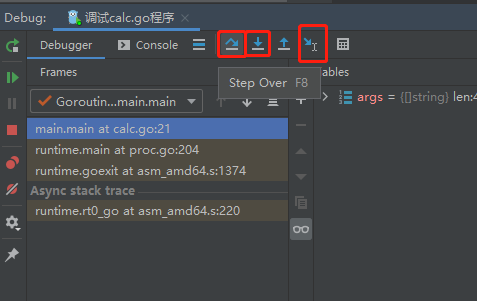
开始调试：

先打好断点，然后



请注意 ctrl + shift + F10是不会走断点的。

主要通过下面三个红框的箭头进行往下调试



第一个箭头是走到下一行

第二个箭头是走到内部的函数

第三个箭头是走到下一个断点

调试没有问题之后，我们开始编译。

编译：

cd bin

go build calc

执行：

./calc add 1 2

得到 3

执行go build calc的时候会先分析calc.go中的import语句以了解包的依赖关系，从而在编译calc.go之前先把依赖的simplemath编译打包好，才开始编译calc.go可执行程序。

接下来介绍怎么运行add\_test.go这些单元测试，可以在任意目录下执行以下命令，他会运行simplemath包下所有的带\_test的测试文件。

go test simplemath

运行结果列出了测试的内容、测试结果和测试时间

**问题追踪和调试**

使用goland的话可以使用断点调试，但是如果使用的普通的IDE和编辑器，我们可以使用打印日志这种调试方法。

Go语言包中包含一个fmt包，其中提供了大量易用的打印函数，我们会 接触到的主要是 Printf()和Println()

下面是几个使用Printf()和Println()的例子：

fval := 110.48

ival := 200

sval := "This is a string. "

fmt.Println("The value of fval is", fval)

fmt.Printf("fval=%f, ival=%d, sval=%s\n", fval, ival, sval)

fmt.Printf("fval=%v, ival=%v, sval=%v\n", fval, ival, sval)

但在正式开始用 Go开发服务器系统 时，我们就不能只依赖fmt包了，而是需要设计严格的日志规范。此时要使用log包。

下一章我们正式介绍如何用go写代码。