Protote的用法

Pom 文件中引入pom

<dependency>

<groupId>com.google.protobuf</groupId>

<artifactId>protobuf-java</artifactId>

<version>3.22.2</version>

</dependency>

和

<plugin>

<groupId>org.xolstice.maven.plugins</groupId>

<artifactId>protobuf-maven-plugin</artifactId>

<version>0.6.1</version>

<configuration>

<protoSourceRoot>${project.basedir}/src/main/protoc/</protoSourceRoot>

<protocArtifact>com.google.protobuf:protoc:3.4.0:exe:${os.detected.classifier}

</protocArtifact>

<pluginId>grpc-java</pluginId>

<pluginArtifact>io.grpc:protoc-gen-grpc-java:1.5.0:exe:${os.detected.classifier}</pluginArtifact>

</configuration>

<executions>

<execution>

<goals>

<goal>compile</goal>

<goal>compile-custom</goal>

</goals>

</execution>

</executions>

</plugin>

Idea安装Protocol Buffers插件。

然后就可以添加proto文件了。

Why did you think company is good and not flighting. That reason

Goal

Reason

/usr/local/bin/etcd

etcdctl get --prefix /

context.Background()的用处

'telegram-trading-bot/ user\_service' is a program, not an importable package

os.Getenv(“”)

如何获取值

goroutine 轻量级的协程

通过channel 实现各个协程之间的通信

分段栈增长和goroutine在线程基础上多路复用技术的自动化。

这个特性显然是Go语言最强有力的部分，不仅支持了日益重要的多核与多处理器计算机，也弥补了现存编程语言在这方面所存在的不足。

Go 语言在执行速度上也可以与C/C++相提并论。

Go的开发者不需要考虑内存泄漏问题，因为它运行在某种意义上的虚拟机上，以次来实现高效快速的垃圾回收(使用了一种简单的标记-清除算法)。

Go语言还能够在运行时进行反射相关的操作。

使用 go install 能够很轻松地对第三方包进行部署。（需要研究一下如何部署以及第三方包时如何管理的）。

defer 延迟执行，在一个函数中它的位置不会影响它的功能。

**结构体**

定义 type StructName struct{}

结构体的字段可以是匿名字段，只需提供类型。

结构体的字段可以是函数。

结构体的字段可以是结构体。

结构体可以带标签，例如

type person struct{

filed1 bool “”

}

给结构体添加方法

func(s \*StructName) getName(){

}

小结: 无论是通过结构体实例和结构体指针，在访问结构体指针的字段时，go语言会自动解引用指针，使得访问方式更加方便统一。

**指针**

在golang中，传递指针的效率可能远远高于传递值本身，在大多数情况下，传递值的地址就足够了。

\* 取址符，获取一个对象的指针

& 取值符，获取一个对象的值

**数组和切片**

长度固定，不可变，且需要在申明的时候指定数组的长度，并且长度参数还只能是个常数。使用不怎么灵活，所以在go中一般用另外一个类型来代替数组。 --切片

数组的声明格式

var identifier [len]type。 例如 var arr1 [5]int 声明的时候必须指定长度

切片的声明格式

var identifier[] type (和数组的差异为没有常量的长度)。例如 var slice1[]int={}

还需要深入了解数组和切片的差异。

深入理解一下make和new 在go中的区别。

make 只能用于内建的数据结构(如切片、映射和通道),并且返回的是已经初始化后的实例。

内建的数据结构(切片,map,通道,结构体,函数,接口,数组,指针)

slice1:=make([]int)

new 用于任何类型的内存分配，返回的是指向零值的指针。

**map的使用**

不要使用new(), 永远使用make()来构造map

map1 := make(map[string]string) 定义一个key类型为string,值类型为string的map

**接口的定义**

Go语言提倡面向接口编程。

接口是一个集多个方法签名的集合。

任何类型的方法集中只要拥有该接口”对应的全部方法”签名。

就表示它实现了该接口,无须在该类型上显式声明实现了哪个接口。换句话说，只要一个类型拥有了一个接口全部的方法，就算是实现了该接口，不需要再另外声明实现某某接口。

接口只有方法声明，没有实现，没有数据字段。

接口可以匿名嵌入其他接口，或嵌入到结构中。（如何嵌入）

对象赋值给接口时，会发生拷贝，而接口内部存储的是指向这个复制品的指针，既无法修改复制品的状态，也无法获取指针。（如何将对象赋值给接口，在什么场景下需要这么做）

只有当接口存储的类型和对象都为nil时，接口才等于nil。

接口调用不会做receiver的自动转换。

接口同样支持匿名字段方法。(如何匿名字段)

接口也同样实现类似于OOP中的多态。

空接口可以作为任何类型数据的容器。

一个类型可以实现多个接口。

接口命名习惯以er结尾。

**反射包**

主要有两种

reflect.TypeOf 返回一个对象的类型

reflect.ValueOf 返回一个对象的值

**协程和通道**

协程 通过go func() 启用一个协程,协程之间通信可通过通道channel,

chan定义一个通道

定义一个channel的方式 var identify chan type

实例化channel identify = make(chan type)

也可以通过通过更短的方式 identifu:=make(chan type)

尝试使用func()作为参数使用一个协程和通道。

探索一下select的用法

通道操作符 <-

往通道里面写数据 identify := <- id

往通道里面读数据 result := <- identify

通道阻塞

通道收发数据，通常情况下，通道收发数据需要在不同的线程中进行，因为发数据的时候需要接数据，没有地方接数据发数据就发不出去。这个时候发数据的地方是阻塞的。

通道的方向

只接收通道 var c chan <-int

只发送通道 var c <-chan int

这两者其实是同一个东西，通道在创建的时候是可以发送和接收的，只接收和只发送只是在某个作用域中通道的功能而已。例如在函数中有通道类型的参数，那么在这里可以将这个通道显示的定义为只接收或只发送通道。这个通常用于提高代码的清晰度和安全性。

**gin 框架的使用**

快速:基于Radix树的路由，小内存占用。没有反射

支持中间件: 传入的HTTP请求可以由一系列中间件和最终操作来处理。如Logger,GZIP,最终操作DB.

Crash处理: Gin可以catch一个发生在http请求中的panic并recover它，这样，服务器将始终可用。（自带的，不需要做任何操作）

路由器:更好的组织路由，是否需要授权，不同API版本…，这些组可以无限制地嵌套而不会降低性能。

**proto在go中的使用方法**

protoc –go\_out=./ --go-grpc\_out=./ proto/trading\_bot.proto 可以通过这个将proto文件生成对应的go文件，这里会生成两个go文件。

Etcd只是服务注册中的一个，真正调用时通过grpc调用的

**领域事件设计**