ماشین حساب

در این تمرین می‌خواهیم یک ماشین حساب ساده را به شکل یک *api* پیاده سازی کنیم.

این ماشین حساب قرار است که فقط قابلیت جمع و تفریق داشته باشد! به این صورت که بعد از اجرای برنامه یک *tcp server* بر روی آدرس و پورت مورد نظر فعال شده تا با درخواست هایی مشخص اعداد را جمع یا تفریق کنیم.

ساختار پروژه

پروژه‌ی اولیه را از این لینک دانلود کنید. ساختار فایل‌های پروژه به صورت زیر است:

```
.
├── go.mod
├── go.sum
├── main.go
└── main_sample_test.go
```

همانطور که می‌بینید این پروژه دارای یک فایل *main.go* بوده که بخش‌هایی از آن از قبل نوشته شده است، شما می‌بایست با استفاده از دانش خود این بخش‌ها را پر کنید و همچنین برنامه را کامل کنید.

ساختار سرور

main.go

```go
type Server struct {}
```

همانطور که می‌بینید شما باید یک سرور پیاده سازی کنید، این سرور دارای متدهایی می‌باشد که آنها را مورد بررسی قرار می‌دهیم:

تابع NewServer

main.go

```go
func NewServer(port string) *Server {}
```

همانطور که از امضاء این تابع مشخص می‌باشد وظیفه این تابع ساختن یک سرور جدید و برگرداندن اشاره‌گری به آن می‌باشد، همچنین این تابع یک ورودی از جنس رشته دریافت می‌کند که پورتی می‌باشد که سرور باید بر روی آن اجرا شود (بعد از استفاده از متد Start)

متد Start

main.go

```go
func (s *Server) Start() {}
```

این متد وظیفه شروع فعالیت سرور را دارد و بعد از اجرا آن باید بر روی پورت تعیین شده سرور ما اجرا شده و آماده دریاف درخواست‌ها باشد.

اطلاعات بیشتر و تضمین‌ها

- فرمت درخواست‌ها همیشه به این صورت خواهد بود:

```
{
  "action":  "add",
  "numbers": "2,1"
}
```

فیلد `action` دو مقدار بیشتر نمی‌تواند داشته باشد `add` برای جمع و `sub` برای تفریق.

- هر تعدادی عدد ممکن است در درخواست وجود داشته باشد که شما باید تمام آن‌ها را بسته به درخواست ارسال شده جمع و یا تفریق کنید. مثلاً:

```
{
 "action":  "sub",
 "numbers": "2,1,55,100,1,-3"
}
```

- زمانی که درخواست درستی ارسال می‌شود سرور می‌بایست جوابی با فرمت `The result of your query is: %d` ارسال کند که این مقدار `%d` برابر با نتیجه محاسبات است.
- زمانی که یک درخواست بدون پارامتر `numbers` ارسال شود باید خطای `'numbers' parameter missing` برگردانده شود.
- چنانچه درخواست شامل موارد غیر عددی باشد باید خطای `Invalid number format` بازگردانده شود.
- اگر محاسبات باعث `overflow` شدن شود باید یک ارور با متن `Overflow` برگردانده شود.
- ارسال جواب باید با استفاده از JSON صورت بگیرد و به فرمت زیر فرستاده شود
- توجه کنید که رشته برگردانده شده باید مطابق نمونه باشد و حروف بزرگ و کوچک آن رعایت شود، همچنین تضمین می‌شود که فقط به متن‌های زیر به عنوان خروجی نیاز دارید و فقط بخش اعداد متغیر می‌باشد.

```
"result": "The result of your query is: 150, "error": ""
"result": "", "error": "'numbers' parameter missing"
"result": "", "error": "Overflow"
```

اورفلو شدن *overflow*

زمانی که محاسباتی بر روی یک عدد انجام شود اگر مقدار نتیجه بیشتر و یا کمتر از حد مجاز نوع متغیر مربوط باشد با سرریز شدن *Overflow* مواجه می‌شوید، برای مثال یک int64 می‌تواند حداکثر عدد 9223372036854775807 را در خود جای دهد.

اکر درخواستی به سمت سرور فرستاده شود که درست بوده اما نتیجه آن باعث *overflow* شدن یک int64 می‌شود باید ارور *Overflow* را برگردانید. خیلی از زبان‌ها به صورت خودکار زمانی که همچین اتفاقی رخ دهد ارور می‌دهند اما گولنگ برای هندل کردن این موضوع تدبیری اندیشیده و آن این است که بعد از رد کردن حداکثر یا حداقل یک نوع عدد آن را wrap کرده و از انتها ادامه می‌دهد، برای مثال اگر به عدد `9223372036854775807` یک را اضافه کنیم بجای ارور به عدد `-9223372036854775808` بر می‌خوریم، شما باید این موضوع را در نظر گرفته و در برنامه خود اگر چنین اتفاقی رخ داد ارور برگردانید.

آن‌چه باید آپلود کنید

فایل `main.go` خود را آپلود کرده و توجه کنید که فایل یا پوشه دیگری را آپلود نکنید.

مدرسه قدیمی

در این تمرین، شما باید یک سیستم ساده برای مدیریت مدارس در یک شهر را پیاده‌سازی کنید. این سیستم شامل دو نقش اصلی است: **معلم** و **دانش‌آموز**. هر دو این نقش‌ها به عنوان کاربران سیستم در نظر گرفته می‌شوند.

سناریو

در این سیستم دو نقش وجود دارد. استاد و دانشجو. همگی کاربران سیستم هستند. در این سیستم میتوان مدرسه و سپس به مدرسه درس اضافه کرد. سپس دانش‌آموزها مجاز به انتخاب کلاس از دانشگاه خودشان هستند. اساتید مجاز به بعهده گرفتن چند درس از مدرسه‌های متفاوت هستند. دانش آموزان تنها مجاز به انتخاب درس از یک مدرسه مشخص هستند. اما میتوانند از همان مدرسه چند درس انتخاب کنند.

مدارس و کلاس‌ها:

- امکان ایجاد مدارس جدید در سیستم وجود دارد.
- هر مدرسه می‌تواند شامل چندین کلاس باشد.
- کلاس‌ها به مدارس مشخصی تعلق دارند.

```go
type School struct {
    Id      uint    `json:"id,omitempty"`
    Name    string  `json:"name,omitempty"`
    Classes []Class `json:"classes,omitempty"`
}

type Class struct {
    Id       uint     `json:"id,omitempty"`
    Name     string   `json:"name,omitempty"`
    SchoolId uint     `json:"school_id,omitempty"`
    Teacher  Person   `json:"teacher,omitempty"`
    Students []Person `json:"students,omitempty"`
}
```

کاربران (معلمان و دانش‌آموزان):

- امکان ایجاد کاربران جدید (معلم یا دانش‌آموز) در سیستم وجود دارد.
- معلمان می‌توانند کلاس‌هایی را در مدارس مختلف به عهده گرفته و تدریس کنند.
- دانش‌آموزان فقط می‌توانند در کلاس‌های یک مدرسه مشخص ثبت‌نام کنند، اما می‌توانند در چندین کلاس از همان مدرسه شرکت کنند.
- یک فرد نمی‌تواند هم‌زمان هم معلم و هم دانش‌آموز باشد.

```go
type Person struct {
    Id      uint   `json:"id,omitempty"`
    Name    string `json:"name,omitempty"`
    Classes []uint `json:"calasses,omitempty"`
}
```

ساختار پروژه

پروژه‌ی اولیه را از این لینک دانلود کنید. ساختار فایل‌های پروژه به صورت زیر است:

```
.
├── entity.go
├── go.mod
├── go.sum
├── main.go
└── sample_test.go
```

همانطور که می‌بینید این پروژه دارای یک فایل *main.go* بوده که بخش‌هایی از آن از قبل نوشته شده است، شما می‌بایست با استفاده از دانش خود این بخش‌ها را پر کنید و همچنین برنامه را کامل کنید.

آن‌چه باید پیاده‌سازی کنید

شما باید یک **Socket Server** پیاده‌سازی کنید که بدون استفاده از هیچ پروتکل پیچیده‌ای (مثل HTTP)، به درخواست‌های کلاینت‌ها پاسخ دهد. ارتباطات بین کلاینت و سرور با استفاده از **فرمت JSON** انجام می‌شود. کلاینت در واقع همان تست ها هستند که در فایل پروژه میتوانید نحوه کار کردن آن را دریابید.

متدها و درخواست‌ها سرور شما باید پنج متد زیر را پشتیبانی کند:

1- **افزودن یک مدرسه جدید** (CreateSchoolMethod): در این درخواست یک استراکت `School` (البته در قالب JSON) فرستاده می‌شود و یک پاسخ در این فرمت از شما دریافت می‌کند:

```go
type Response struct {
    Status  bool        `json:"status,omitempty"`
    Message string      `json:"message,omitempty"`
    Data    interface{} `json:"data,omitempty"`
}
```

2- **افزودن فردی جدید** (CreatePersonMethod): در این درخواست یک استراکت `Person` فرستاده می‌شود و پاسخ سرور مانند مورد قبل است.

3- **افزودن کلاسی جدید** (CreateClassMethod): در این درخواست هم یک استراکت `Class` فرستاده می‌شود و پاسخ سرور مانند مورد قبل است.

4- **اضافه کردن دانش‌آموز به کلاس** (AddStudentToClassMethod): در این درخواست یک استراکت `AddStudentToClassReq` فرستاده می‌شود و پاسخ سرور مانند مورد قبل است:

```go
type AddStudentToClassReq struct {
    StudentId uint `json:"student_id,omitempty"`
    ClassId   uint `json:"class_id,omitempty"`
}
```

5- **دریافت اطلاعات فردی** (WhoAmIMethod): در این درخواست یک استراکت `Person` شامل `ID` فرستاده می‌شود، و در پاسخ باید اطلاعات فردی آن شخص را دریافت کند. مثلا اگر فردی استاد است، یک لیست از `id` کلاس‌هایی که او به عنوان معلم در آن‌ها حضور دارد داده میشود.

نکات مهم پیاده‌سازی

- برنامه نباید باگ منطقی داشته باشد. مثلا یک درس را دو استاد نمی‌توانند به عهده بگیرند. کیفیت منطق برنامه به شما واگذار شده است. مثلاً بررسی کنید که معلم و مدرسه مورد نظر در هنگام ایجاد کلاس وجود داشته باشند.
- برای هر مدرسه، کلاس و فرد جدید، یک id یکتا اختصاص دهید.
- کیفیت و منطق بسیاری از موارد (به غیر از فانکشنالیتی‌های خواسته شده) به شما واگذار شده تا خودتان راه حل مورد نظر خود را پیاده‌سازی کنید و این مورد از اساس‌ترین بخش‌های این تمرین است.

آن‌چه باید آپلود کنید

پس از اتمام پروژه، فایل `main.go` را آپلود کنید.

ترموستات هوشمند

- محدودیت زمان: ۴ ثانیه
- محدودیت حافظه: ۱۰۲۴ مگابایت

قلی که با آمدن فصل بهار انتظار داشت دیگر هوا گرم شود، از اینکه هوا مدام گرم و سرد می‌شود بسیار دلخور است. او می‌خواهد برای اتاق‌های خانه یک سیستم ترموستات هوشمند طراحی کند تا بتواند با خیال راحت به کارهایش برسد و مجبور نباشد مدام شوفاژ و کولر را خاموش و روشن کند.

ترموستاتی که قلی مد نظر دارد دارای یک سیستم کنترل مرکزی است که لیست همه‌ی اتاق‌ها و اطلاعات ترموستات هر اتاق را نگه‌داری می‌کند؛ قادر است با دماسنج، دمای اتاق‌ها را ثبت کند و بر اساس اینکه آیا کسی در اتاق حضور دارد یا نه، فن اتاق را روشن کند تا دمای آن به دمای مطلوب برسد.

آن‌چه باید پیاده‌سازی کنید

در این قسمت ساختارهایی را که باید پیاده‌سازی کنید مشاهده می‌کنید:

```go
type Thermostat struct {
}

type Room struct {
}

type SystemController struct {
}
```

استراکت `Thermostat` : این استراکت نمایانگر ساختار یک ترموستات است که دارای خصوصیات مربوط به دمای فعلی و دمای هدف است.

استراکت `Room` : این استراکت نمایانگر ساختار یک اتاق است. هر اتاق دارای شناسه، یک ترموستات و سنسور تشخیص حضور افراد در اتاق است.

استراکت `SystemController` : این استراکت مسئول مدیریت مجموعه‌ای از اتاق‌ها است و اطلاعات همه‌ی آن‌ها را نگه‌داری می‌کند.

نکته: با نگاه کردن به فایل تست نمونه می‌توانید بفهمید ک چه فیلدهایی در استراکت‌ها اجباری هستند و باید حتما وجود داشته باشند. به جز این موارد می‌توانید فیلدهای دیگری بر اساس نیاز مسئله ایجاد کنید.

امضای توابع و مشخصات مورد نیاز

در این قسمت توابعی را که باید پیاده‌سازی کنید مشاهده می‌کنید که در ادامه به توضیح آن می‌پردازیم:

```go
func NewSystemController() *SystemController

func (s *SystemController) AddRoom(room *Room) error

func (s *SystemController) UpdateRoomTemperature(roomID string, newTemp int) error

func (s *SystemController) GenerateReports()

func (r *Room) SetOccupancy(occupied bool)

func (r *Room) StartFan() error

func (r *Room) StopFan() error

func (r *Room) GetCurrentTemperature()

func (r *Room) GetTargetTemperature()

func (r *Room) GetIsRoomOccupied()
```

متد `NewSystemController`

این تابع یک نمونه از `SystemController` را ایجاد می‌کند.

متد `AddRoom`

این تابع یک اتاق جدید را به سیستم اضافه می‌کند. این عملیات ممکن است به صورت همروند انجام شود.

خطاها:

شناسه اتاق باید منحصر به فرد باشد. اضافه کردن اتاق با شناسه تکراری باعث ایجاد خطا با پیام زیر می‌شود:

```
room already exists
```

اگر شناسه اتاق خالی باشد باعث ایجاد خطا با پیام زیر می‌شود:

```
room ID is required
```

متد `GetCurrentTemperature` : این تابع دمای فعلی اتاق را برمی‌گرداند.

متد `GetTargetTemperature` : این تابع دمای هدف اتاق را برمی‌گرداند.

متد `GetIsRoomOccupied` : این تابع وضعیت اِشغال بودن اتاق را برمی‌گرداند.

متد `GetIsFanRunning` : این تابع وضعیت خاموش یا روشن بودن فن اتاق را نشان می‌دهد.

متد `UpdateRoomTemperature` :

این تابع دمای فعلی یک اتاق مشخص را به‌روزرسانی می‌کند. در واقع دماسنج اتاق از این تابع استفاده می‌کند تا دمای فعلی اتاق را تعیین کند. اگر دمای فعلی با دمای هدف مطابقت داشته باشد، فن متوقف می‌شود، در غیر این صورت فن شروع به کار می‌کند تا دما را به دمای هدف برساند.

خطاها:

اگر دمای وارد شده منفی باشد باعث ایجاد خطا با پیام زیر می‌شود:

```
invalid target temperature
```

همچنین اگر شناسه اتاق اشتباه باشد باعث ایجاد خطا با پیام زیر می‌شود:

```
room does not exist
```

متد `GenerateReports` :

این تابع گزارش‌هایی از وضعیت دما و حالت کارکرد فن در سیستم تولید می‌کند. این گزارش‌ها شامل شناسه‌ی اتاق‌ها و حالت کاری ترموستات هرکدام از آن‌ها است.

محاسبه حالت کاری فن به این صورت است: اگر فن در حال خنک کردن اتاق باشد `cooling` ، اگر در حال گرم کردن اتاق باشد `heating` و اگر خاموش باشد `off` .

برای مثال اگر در کل سیستم ۲ اتاق داشته باشیم با شناسه‌های `101` و `102` و دمای اولی در حال کاهش و دمای دومی در حال افزایش باشد در map خروجی این دو جفت key-value را خواهیم داشت.

```
"101" -> "cooling"
"102" -> "heating"
```

متد `SetOccupancy` :

این تابع وضعیت اِشغال بودن اتاق را تنظیم می‌کند. اگر اشخاصی در اتاق حضور داشته باشند، فن به منظور رساندن دما به دمای هدف فعال می‌شود و اگر اتاق خالی شود، فن متوقف می‌گردد. البته که دمای ترموستات هم در خاموش یا روشن شدن فن تاثیر دارد. اما به طور کلی اگر اتاق خالی باشد فن روشن نخواهد شد.

متد `StartFan` :

این تابع فن اتاق را فعال می‌کند. فن‌ها قادر هستند که به تدریج دمای اتاق را تغییر دهند تا آن را به دمای هدف خود برسانند. سرعت تغییر دمای آن‌ها ۱ درجه سانتیگراد بر ثانیه است. دقت کنید که فن هر اتاقی می‌تواند خاموش یا روشن باشد و همه‌ی آن‌ها باید بتوانند به طور همزمان در اتاق خود کار کنند.

خطاها:

اگر شخصی در اتاق حضور نداشته باشد باعث ایجاد خطا با پیام زیر می‌شود:

```
room is not occupied
```

اگر سعی کنیم یک فن که روشن است را دوباره روشن کنیم، باعث ایجاد خطا با پیام زیر می‌شود:

```
fan already running
```

اگر دمای اتاق نیازی به تنظیم نداشته باشد فن نباید روشن شود و باعث ایجاد خطا با پیام زیر می‌شود:

```
no adjustment needed
```

متد `StopFan` : این تابع فن اتاق را متوقف می‌کند.

خطاها:

اگر فن در حال کار نباشد، باعث ایجاد خطا با پیام زیر می‌شود:

```
fan not running
```

نکات بیشتر

- دقت کنید که همه‌ی متدها ممکن است به صورت concurrent تست شوند.
- در توابعی که خطا برمی‌گردانند اگر خطایی رخ نداد، مقدار nil برمی‌گردد.
- با خواندن تست‌های نمونه متوجه خواهید شد که در هر استراکت چه فیلدهایی باید حتما وجود داشته باشند.
- در فایل `main_sample_test.go` چند تست نمونه قرار داده شده که می‌توانید آن‌ها را با دستور `go test` اجرا کنید. همچنین می‌توانید برای خود تست‌های بیشتری بنویسید.

جزئیات پروژه

پروژه‌ی اولیه را از این لینک دانلود کنید. ساختار فایل‌های پروژه به صورت زیر است:

```
.
├── go.mod
├── go.sum
├── main.go
└── main_sample_test.go
```

آن‌چه باید آپلود کنید

پس از پیاده‌سازی توابع خواسته شده، فایل `main.go` را آپلود کنید.

لطفاً تو صف وایسید

در این سوال می‌خواهیم یک سیستم صف پیام یا *Message broker* مانند *RabbitMQ* درست کنیم تا مهارت‌هایمان را در برنامه‌نویسی شبکه و هم‌روندی بهبود ببخشیم.

سیستم صف پیام ما باید قادر باشد پیام‌ها را از `producer` ها دریافت کرده و به `consumer` ها تحویل دهد. می‌خواهیم این سیستم ویژگی‌های زیر را داشته باشد: سرور *TCP* شما باید قادر باشد به‌طور همزمان با چندین تولیدکننده و مصرف‌کننده ارتباط برقرار کند، از اولویت‌بندی پیام‌ها پشتیبانی کند، به‌طوری که پیام‌های با اولویت بالاتر زودتر تحویل داده شوند. مصرف‌کنندگان همچنین می‌توانند به یک یا چند **topic** مشترک شوند و تولیدکنندگان می‌توانند پیام‌ها را به موضوعات مشخصی ارسال کنند. پیام‌ها باید بلافاصله به تمام مصرف‌کنندگانی که به موضوع مربوطه **subscribe** کرده‌اند، ارسال شوند.

آن‌چه باید پیاده‌سازی کنید

پروژه‌ی اولیه را از این لینک دانلود کنید. ساختار فایل‌های پروژه به صورت زیر است:

```
├ initial_queue
  ├ queue
  │ └ queue.go
  ├ server
  │ ├ server.go
  │ └ topic.go
  ├ tests
  │ └ main_sample_test.go
  ├ go.mod
  ├ go.sum
  └ main.go
```

امضای توابع و مشخصات مورد نیاز

```go
// queue.go

type Message struct{
    ID       uuid.UUID
    Content  string
    Priority int
    Index    int
    // other fields
}

type MessageQueue []*Message

func NewMessageQueue() IMessageQueue {
    mq := &MessageQueue{}
    heap.Init(mq)
    return mq
}
```

استراکت Message: این استراکت نمایانگر یک پیام است که دارای فیلدهای مربوط به شناسه پیام، محتوا و اولویت است. `MessageQueue` هم نمایانگر صف پیام‌هاست که باید اینترفیس `heap.Interface` را پیاده‌سازی کند که در قسمت درسنامه‌ها بیشتر توضیح داده خواهد شد. برای شناسه نیز باید از uuid استفاده کنید.

```go
// topic.go

type Topic struct {
    Name string
    MQ   queue.IMessageQueue
    // other fields
}

func (t *Topic) GetMessageQueue() *queue.MessageQueue
```

استراکت Topic: نمایانگر یک موضوع است که دارای صف پیام‌های مخصوص به خود و عملکردهای مربوط به مدیریت پیام‌ها است.

متد GetMessageQueue: صفِ پیام‌های مربوط به آن موضوع را برمی‌گرداند.

```go
// server.go

type Server struct {
    Addr string
    // other fields
}

func NewServer(address string) *Server {
    return &Server{
        Addr: address,
    }
}

func (s *Server) Run() error

func (s *Server) Stop()

func (s *Server) GetTopic(topicName string) (*Topic, bool)


```

```
func (s *Server) GetClientConnections() []net.Conn
```

استراکت Server: این استراکت سرور *TCP* را پیاده‌سازی می‌کند که با تولیدکنندگان و مصرف‌کنندگان ارتباط برقرار می‌کند و مدیریت انتشار و اشتراک پیام‌ها را بر عهده دارد.

متد Run: سرور را راه اندازی کرده و اگر مشکلی به وجود بیاید ارور برمی‌گرداند.

متد Stop: سرور را خاموش می‌کند. این متد باید همه‌ی پروسه‌های موجود را قطع کند.

نکته: با نگاه کردن به فایل تست نمونه می‌توانید بفهمید که چه فیلدهایی در استراکت‌ها اجباری هستند و باید حتماً وجود داشته باشند. به جز این موارد می‌توانید فیلدهای دیگری بر اساس نیاز مسئله ایجاد کنید.

متد GetTopic: بر اساس نام موضوع، شیء `Topic` مربوطه را برمی‌گرداند. همچنین یک بولین برمی‌گرداند که اگر `true` باشد به معنای این است که تاپیک از قبل وجود داشته و الان ساخته نشده. دقت کنید که اگر نام تاپیک وجود نداشته باشد باید ایجاد شود.

متد GetClientConnections: لیست کانکشن‌های همه‌ی کاربران متصل به سرور را برمی‌گرداند.

نکته مهم: دقت کنید پیاده‌سازی ساختارها، متدها و اینترفیس‌های خواسته شده الزامی است اما برای اینکه سرور خود را تکمیل کنید می‌توانید هرطور که خواستید کد بزنید. در ادامه به توصیف رفتار سرور می‌پردازیم.

ویژگی‌های سرور

در این قسمت رفتار سرور و در قسمت بعد فرمت پیام‌ها را بررسی می‌کنیم.

- با صدا زدن تابع `NewServer` یک نمونه از سرور را دریافت می‌کنیم و با صدا زدن متد `Run` سرور شروع می‌کند به گوش دادن روی آدرس داده شده. یک کلاینت می‌تواند به این *endpoint* درخواست بفرستد و از طریق سوکت به سرور متصل بماند.
- ارتباط بین کاربران (تولیدکنندگان و مصرف‌کنندگان) و سرور باید از طریق **TCP Socket** و با استفاده از فرمت **JSON** انجام شود.
- کاربران باید اتصال خود را با سرور نگه دارند و پیام‌ها را از طریق همین اتصال ارسال و دریافت کنند.
- هر پیام بعد از اینکه توسط همه‌ی مصرف‌کننده‌ها مصرف شد از هیپ حذف می‌شود.
- هر بار یک کاربر در یک موضوع `subscribe` می‌کند، پیام‌های قبلی را دریافت نمی‌کند و فقط پیام‌های بعد از عضویتش را می‌بیند.
- همه‌ی کلاینت‌هایی که در یک موضوع مشترک هستند، باید پیام‌های منتشر شده در آن موضوع را در هنگام انتشار بلافاصله ببینند (مثل یک چت روم).
- هر موضوع باید دارای صف پیام‌های خود باشد که به صورت صف اولویت‌دار با استفاده از `container/heap` پیاده‌سازی شده است. طبیعتاً لزوم استفاده از صفِ با اولویت و هیپ در load کم قابل درک نیست.
- سرور باید بتواند به‌صورت همروند با چندین تولیدکننده و مصرف‌کننده ارتباط برقرار کند.
- اگر موضوعی وجود نداشته باشد، سرور باید آن را هنگام اشتراک یا انتشار پیام ایجاد کند.
- مصرف‌کنندگان می‌توانند از یک موضوع لغو اشتراک کنند و در این صورت دیگر پیام‌های آن موضوع را دریافت نمی‌کنند.
- کلاینت‌ها می‌توانند با ارسال درخواست `close_connection` اتصال خود را با سرور قطع کنند. در این صورت این کاربر باید از همه‌ی موضوعات حذف شده و سپس ارتباطش با سرور قطع شود.
- با ارسال درخواست `shutdown` سرور خاموش شده و همه‌ی ارتباطات قطع می‌شوند. (به اصطلاح graceful shutdown)

پروتکل ارتباطی

پیام‌های ارسالی از تولیدکنندگان:

- اقدام برای انتشار پیام:

```
{
  "action": "publish",
  "message": {
    "topic": "topic_name",
    "content": "message content",
    "priority": 3
  }
}
```

- اگر content یا topic یا priority خالی باشد، خطای مناسب برگردانده می‌شود:

```
message content is required
topic is required
priority is required
```

دقت کنید که فرمت JSON ارسال ارور به این شکل است:

```
{
    "error": "error message"
}
```

پیام‌های ارسالی از مصرف‌کنندگان:

- اقدام برای اشتراک:

```
{
  "action": "subscribe",
  "topic": "topic_name"
}
```

- اقدام برای لغو اشتراک:

```
{
  "action": "unsubscribe",
  "topic": "topic_name"
}
```

- در این دو مورد اگر فیلد `topic` خالی باشد، خطای زیر برگردانده می‌شود:

```
topic is required
```

- فرمت پیام دریافتی توسط مصرف‌کننده:

```
{
  "action": "deliver",
  "message": {
    "message_id": "unique_id",
    "topic": "topic_name",
    "content": "message content",
    "priority": 1
  }
}
```

پاسخ‌های سرور:

- در صورت موفقیت:

```
{
  "status": "ok"
}
```

- در صورت خطا:

```
{
  "error": "error message"
}
```

قطع ارتباط یک کلاینت با سرور:

```
{
   "action": "close_connection"
}
```

خاموش کردن سرور:

```
{
   "action": "shutdown"
}
```

نکات بیشتر

- سرور پس از اقدام‌های `publish` ، `subscribe` ، `unsubscribe` در صورت موفقیت باید پیام `ok` را دریافت کند.
- در صورتی که `action` اشتباه باشد باید ارور `unknown action` را دریافت کنیم.
- اگر در درخواست‌ها مقادیر یکی از فیلدهای `message` ، `topic` ، `content` ، `priority` خالی باشد باید ارور متناسب را در فرمت مناسب که پیشتر ذکر شد دریافت کنیم:

```
message is required
message content is required
topic is required
priority is required
```

- برای شناسایی کلاینت‌ها می‌توانید از `remote address` آن‌ها استفاده کنید که از `net.Conn` قابل دریافت است.

درسنامه‌ها و توضیحات

▼ توضیحات در مورد پیاده‌سازی صف

صف اولویت‌دار (Priority Queue) یک نوع ساختار داده است که در آن هر عنصر دارای یک اولویت است و عنصری که دارای بالاترین اولویت است، قبل از عناصر دیگر پردازش می‌شود. برخلاف صف معمولی که از قانون *FIFO* (اولین ورودی، اولین خروجی) پیروی می‌کند، صف اولویت‌دار ترتیب عناصر را بر اساس اولویت آن‌ها تعیین می‌کند. برای مصورسازی این ساختار داده می‌توانید از این لینک استفاده کنید.

پکیج `container/heap` در Go ابزارهایی را برای پیاده‌سازی ساختار داده‌ی هیپ *Heap* فراهم می‌کند. هیپ یک ساختار داده است که می‌تواند به صورت کارآمد عنصر با کمترین (یا بیشترین) مقدار را در زمان $O(1)$ پیدا کند و درج و حذف عناصر در آن در زمان $O(log(n))$ انجام می‌شود.

برای استفاده از این پکیج، باید یک نوع داده (استراکت) را تعریف کنیم که اینترفیس `heap.Interface` را پیاده‌سازی کند. این اینترفیس شامل ۵ متد است که در مثال زیر آن‌ها را توضیح می‌دهیم.

در ابتدا یک ساختار برای عناصر صف و سپس یک تایپ برای خود صف تعریف می‌کنیم:

```
package main

import (
    "container/heap"
)

type Item struct {
    value    string
    priority int
    index    int
```

```
}

type PriorityQueue []*Item
```

متد Len: تعداد عناصر موجود در هیپ را برمی‌گرداند.

```
func (pq PriorityQueue) Len() int { return len(pq) }
```

متد Less: تعیین می‌کند که آیا عنصر در اندیس `i` باید قبل از عنصر در اندیس `j` قرار گیرد یا نه.

```
func (pq PriorityQueue) Less(i, j int) bool {
    return pq[i].priority < pq[j].priority
}
```

متد Swap: عناصر در اندیس‌های `i` و `j` را با هم جابجا می‌کند.

```
func (pq PriorityQueue) Swap(i, j int) {
    pq[i], pq[j] = pq[j], pq[i]
    pq[i].index = i
    pq[j].index = j
}
```

متد Push: یک عنصر را به هیپ اضافه می‌کند.

```
func (pq *PriorityQueue) Push(x interface{}) {
    n := len(*pq)
    item := x.(*Item)
    item.index = n
    *pq = append(*pq, item)
}
```

متد Pop: عنصر با اولویت بالا را از هیپ حذف کرده و برمی‌گرداند.

```
func (pq *PriorityQueue) Pop() interface{} {
    old := *pq
    n := len(old)
    item := old[n-1]
    item.index = -1
    *pq = old[0 : n-1]
    return item
}
```

حالا می‌توانیم با استفاده از این ساختارها صف خود را بسازیم.

```
items := map[string]int{
    "task1": 3,
    "task2": 1,
    "task3": 2,
}

pq := make(PriorityQueue, 0)

for value, priority := range items {
    item := &Item{
        value:    value,
        priority: priority,
    }
    heap.Push(&pq, item)
}
```

خارج کردن عناصر از صف بر اساس اولویت:

```
for pq.Len() > 0 {
    item := heap.Pop(&pq).(*Item)
    fmt.Printf("Processing %s with priority %d\n", item.value, item.priority)
}
```

خروجی:

```
Processing task2 with priority 1
Processing task3 with priority 2
Processing task1 with priority 3
```

نکته: در پکیج `heap` اولویتِ بالاتر، مقدار عددی کمتری دارد. مثلاً اولویت ۱ از ۳ بالاتر است.

▼ تولید شناسه یکتا با استفاده از UUID

برای تولید شناسه‌های یکتا (UUID)، می‌توانید از این پکیج استفاده کنید.

ابتدا پکیج را نصب کنید:

```
go get github.com/google/uuid
```

سپس می‌توانید به صورت زیر از آن استفاده کنید:

```
package main

import (
    "fmt"
    "github.com/google/uuid"
)

func main() {
    v4, err:= uuid.NewRandom()
    if err != nil {
        log.Fatal("cannot generate v4 uuid")
    }
    fmt.Printf("v4 uuid: %v\n", v4)
}
```

حتماً از ورژن ۴ استفاده و در مورد مزایای آن نیز مطالعه کنید.

▼ کار با JSON در Go

برای کار با JSON در Go، می‌توانید از پکیج `encoding/json` استفاده کنید.

مثال: تبدیل یک ساختار به JSON و برعکس.

```
package main

import (
    "encoding/json"
    "fmt"
)

type Message struct {
    Action  string `json:"action"`
    Content string `json:"content"`
}

func main() {
    msg := Message{
        Action:  "publish",
        Content: "Hello, World!",
    }

    jsonData, err := json.Marshal(msg)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(jsonData))

    var msg2 Message
    err = json.Unmarshal(jsonData, &msg2)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("%+v\n", msg2)
}
```

برای ارسال JSON در یک کانکشن راه‌های مختلفی وجود دارد که به یکی از آن‌ها اشاره می‌کنیم.

```
package main

import (
    "encoding/json"
    "log"
    "net"
)

func main() {
    address := "127.0.0.1:8080"
    conn, err := net.Dial("tcp", address)
    if err != nil {
        log.Fatal(err)
    }

    encoder := json.NewEncoder(conn)
    message := map[string]interface{}{
        "content":  "hello world",
        "priority": 1,
    }
```

```
    err = encoder.Encode(message)
    if err != nil {
        log.Fatal(err)
    }
}
```

همچنین با `NewDecoder` می‌توان دیتا دریافت کرد.

▼ ایجاد و راه‌اندازی یک سرور TCP Socket در Go

برای این کار می‌توانید از پکیج net استفاده کنید. در ادامه یک مثال ساده می‌آوریم.

```
package main

import (
    "fmt"
    "log"
    "net"
)

func main() {
    ln, err := net.Listen("tcp", ":8080")
    if err != nil {
        log.Fatal(err)
    }
    defer ln.Close()
    log.Println("server started...")

    for {
        // new connection
        conn, err := ln.Accept()
        if err != nil {
            log.Println(err)
            continue
        }
        // handle connection concurrently
        go handleConnection(conn)
    }
}
```

اینکه چطور کاربر را در کانکشن نگه دارید با شما 😊 .

آن‌چه باید آپلود کنید

پس از پیاده‌سازی برنامه خواسته شده، پوشه‌های queue و server را در یک فایل zip قرار داده و ارسال کنید. می‌توانید فایل و پوشه‌های دیگری هم قرار دهید. محتویات زیپ شما در کنار فولدر تست قرار خواهد گرفت و تست خواهد شد.

فایل `main.go` برای خودتان است می‌توانید از آن برای تست کردن برنامه استفاده کنید. البته بهتر است که برای خود تست بنویسید.