QQ频道机器人方案设计

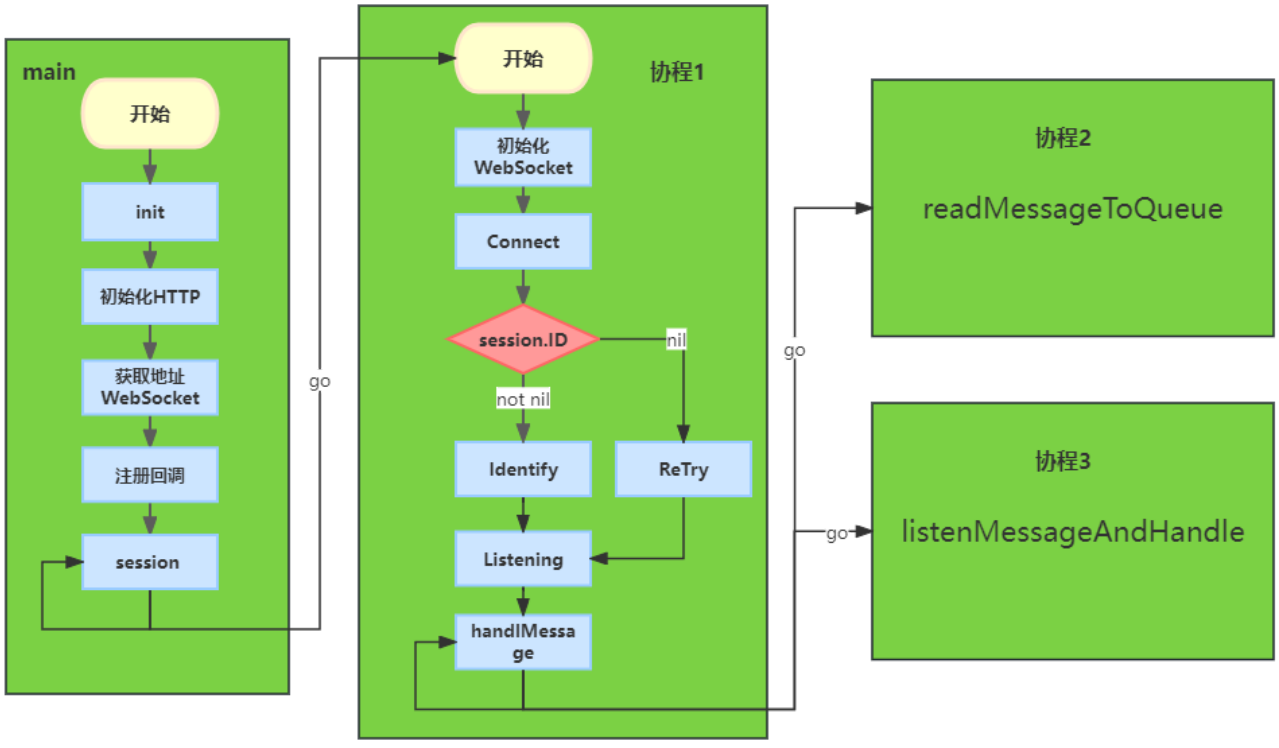
功能：成语接龙、成语查询

1. 构思

由于要实现的是一个对话机器人，因此采用WebSocket来做消息的传递，方便双方互发消息。其次从功能角度来说，经过调查发现，成语的数量大概在两三万条，其中常用的成语数量更是只有几千，因此要实现这个功能，完全可以先进行数据采集，在本地生成一个比较完整的成语库，然后在系统初始化的时候，将数据全部读进内存，并为其建立索引，方便快速访问。
项目的关键点：

1. 网络通信的实现
2. 功能业务逻辑的实现
3. 对成语库数据的处理

2. 整体程序框架设计



程序总体包含四大块，第一块主要完成一些初始化工作和一些相关数据结构的创建；

第二块是代表跟WebSocket操作相关的协程1，在这里主要完成Websocket的创建和连接；第三块的协程2负责将感兴趣的数据读取到队列；第四块协程3负责将管道中的数据读取出来进行处理，根据数据的payload来确定数据类型，然后执行相应的回调函数，本项目中实现成语接龙相关的功能函数在此模块中被调用。

main

init

在init函数中主要完成了两件事：

1. 从yaml配置文件中读取机器人的ID和Token

```
content, err := os.ReadFile("config.yaml")
err = yaml.Unmarshal(content, &config)
```

2. 从json格式的成语数据库中读取数据到内存并初始化相应的数据结构

```
err = json.Unmarshal(jsonData, &idioms)
……
……
```

初始化http

这里会初始化一个http的连接，并将读取到的ID和token等信息以及相关的Header填充到相应的字段。

```
m_client = mpkg.NewClient(config.AppID, config.Token, 3*time.Second)
```

获取WebSocket连接

```
// 通过http访问"https://api.sgroup.qq.com/gateway/bot"获取webSocket连接地址
ws, err := m_client.GetWSS(ctx)
```

根据API文档，通过初始化好的http访问固定的地址，其返回的数据的body里包含了要建立WebSocket相关的数据结构，可以得到WebSocket的接入点信息

```
{
  "wss://api.sgroup.qq.com/websocket/",
```

```
  "shards": 9,
  "session_start_limit": {
    "total": 1000,
    "remaining": 999,
    "reset_after": 14400000,
    "max_concurrency": 1
  }
}
```

注册回调函数

对感兴趣的消息事件编写回调函数，并将其注册到事件列表，同时会返回一个int型变量的表征事件的变量，其会在后续的WebSocket连接过程中参与鉴权

```
intent := mpkg.RegisterHandlers(atMessage)
```

创建Session

将上面得到的一些数据信息初始化到Session的结构里，然后main程序会阻塞在下面的for循环中，每次从管道里读取到一个Session，就会创建一个协程并发起一个WebSocket连接。

```
for session := range l.sessionChan {
        // MaxConcurrency 代表的是每 5s 可以连多少个请求
        time.Sleep(startInterval)
        go l.newConnect(session)
}
```

协程1

这个协程负责WebSocket连接的建立，以及最后会阻塞在Listening()函数的调用中具体步骤大致包括四部分：

```
//创建
wsClient := NewWebsocket(session)
//连接
wsClient.Connect()
//鉴权
wsClient.Identify()
```

```
//监听
wsClient.Listening()
```

在Listening函数中，会启动两个协程分别去处理消息读取和消息处理

```
go c.readMessageToQueue()
go c.listenMessageAndHandle()

for {
        select
        case <-resumeSignal:
        case err := <-c.closeChan:
        case <-c.heartBeatTicker.C:
}
```

最后阻塞在管道上，等待错误信息发生或者心跳请求，执行相应的操作。

协程2（readMessageToQueue）

该协程读取WebSocket收到的消息，然后往队列messageQueue里面写

```
c.messageQueue <- payload
```

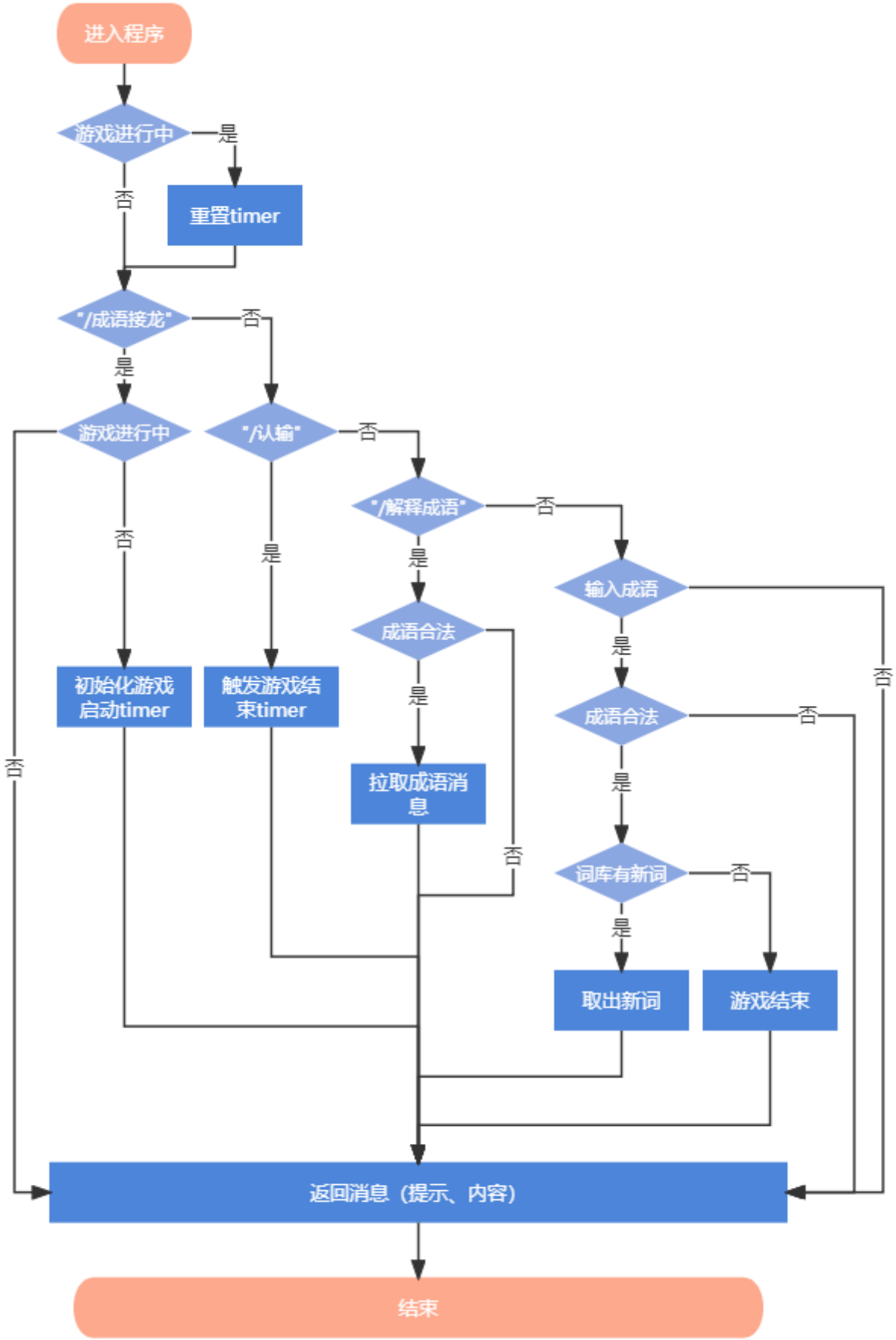
协程3（listenMessageAndHandle）

该协程会不断从messageQueue里面读取消息，然后根据消息中的Type来进行处理，调用相应的回调函数，完成实现的功能。

```
// 解析具体事件，并投递给业务注册的 handler
err := ParseAndHandle(payload)
```

3. 业务逻辑

程序逻辑图


进入程序
游戏进行中
是
否
重置timer
"/成语接龙"
否
是
游戏进行中
"/认输"
否
否
是
初始化游戏
启动timer
触发游戏结
束timer
"/解释成语"
否
是
成语合法
是
否
拉取成语消
息
输入成语
是
否
成语合法
是
否
词库有新词
是
否
取出新词
游戏结束
返回消息（提示、内容）
结束

timer

为了防止启动游戏之后，没有主动释放游戏，设计了一个timer定时器，其设定是超过60s如果没有收到消息反馈，就会调用回调函数自动结束游戏。

```
// timer超时没响应自动结束游戏的回调函数
func timeout_handle(timer *time.Timer, data *mpkg.Message) {
    <-timer.C
    gameIsRunning = false
    lastWord = ""
    m_client.PostMessage(ctx, data.ChannelID, &mpkg.MessageToCreate{MsgID:
data.ID, Content: "欢迎下次继续挑战，本次成语接龙游戏已结束，再见！"})
}
```

因此每次在接收到消息时，实现会判断游戏是否在进行中，如果在进行中，会重置timer，判断是否游戏进行是通过在全局设置了一个bool类型的变量。

```
if gameIsRunning {
        // 收到消息就刷新timer
        tm.Reset(60 * time.Second)
}
```

/成语接龙

接下来程序会进到条件语句中进行判断，如果WebSocket消息体中收到的信息是/成语接龙，那么会启动游戏，并启动timer

```
tm = time.NewTimer(60 * time.Second)
go timeout_handle(tm, data)
```

/认输

如果在游戏过程中，遇到不会了，进行不下去了，除了超时自动结束游戏，也可以主动放弃游戏，输入关键字/认输

/解释成语 xxxx

如果输入为/解释成语 xxxx，则判定为需要对xxxx成语进行解释，会从词库里查找该成语，如果找了，会将成语的相关信息进行输出，如果没找到，则输出提示成语有误。

4. 数据处理

数据结构

```
type IdiomData struct {
    Word        string `json:"word"`        //名称
    Derivation  string `json:"derivation"`  //出处
    Explanation string `json:"explanation"` //解释
    Pinyin      string `json:"pinyin"`      //拼音
}
```

存储成语的结构包含了四部分内容

- 名称
- 解释
- 出处
- 拼音

全局变量

```
// 根据成语首字母建立哈希表，所有首字母相同的成语存储在一个列表中
type idiomsMap map[string]([]*mpkg.IdiomData)
type idiomsSet map[string]bool
type idiomsToPtr map[string]*mpkg.IdiomData
type idiomCount map[string]int

// 接龙游戏的定时timer
var tm *time.Timer
// 存放成语的列表，索引结构中存放的是指针
var idioms []mpkg.IdiomData
// 游戏状态标志位，表示游戏是否在进行中
var gameIsRunning bool = false
// 目前成语的最后一个字，也就是下一个成语应该出现的第一个字
var lastWord string = ""
// 成语索引哈希表
var idiomsIndex idiomsMap
// 记录成语以识别是否出现过
var isSame idiomsSet
// 全量成语的set,用来快速找到指定成语
var idiomDatabase idiomsToPtr
```

```
// 纪录每个列表中还剩多少没出现过的成语，以判断是否应该结束游戏
var idiomCnt idiomCount
```

为了记录游戏过程中的一些状态信息，在全局申明了以上这些变量。

idioms

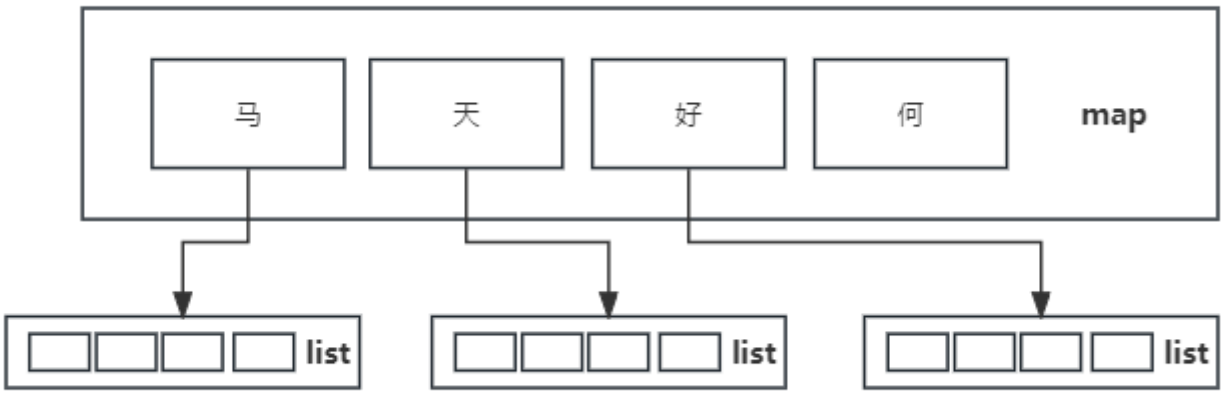
idioms是一个list，用来存放所有的成语，所有实际的数据都存储在这个列表中，其他结构中跟数据先关的存放的都是指针。

idiomDatabase

同时为了能根据成语名称快速拿到其数据结构，idiomDatabase是一个map，来存放成语名称到其数据结构存放地址的映射。

idiomsIndex

为了能够按照成语首文字来进行分类检索，idiomsIndex会对成语做一个分类，将所有首文字相同的成语用一个列表存起来，然后用一个map来纪录首文字到列表的映射关系，当然这存储的也是指针。



idiomCnt

除此之外，还要统计每个列表中，剩余的成语数量，因为成语不能重复使用，随着游戏的进行，可使用的成语会越来越少，如果没有可用的成语了，就要退出，不然可能会出现死循环。

isSame

isSame是一个set，用来纪录从游戏开始到现在，已经说过的成语。

随机查找

每次在某个list找取一个合适的成语，使用了随机函数，这样有利于成语的出现频率相对平衡，如果随机找出来的成语被用过了，那么就会继续随机，直到找到一个合理的成语为止。