gin框架-JWT验证实践

背景: 在如今前后端分离开发的大环境中，我们需要解决一些登陆，后期身份认证以及鉴权相关的事情，通常的方案就是采用请求头携带token的方式进行实现。本篇文章主要分享下在Golang语言下使用jwt-go来实现后端的token认证逻辑。

`JSON Web Token(JWT)`是一个常用语HTTP的客户端和服务端间进行身份认证和鉴权的标准规范，使用JWT可以允许我们在用户和服务器之间传递安全可靠的信息。

在开始学习JWT之前，我们可以先了解下早期的几种方案。

1 token、cookie、session的区别

Cookie

Cookie总是保存在客户端中，按在客户端中的存储位置，可分为`内存Cookie`和`硬盘Cookie`。

内存Cookie由浏览器维护，保存在内存中，浏览器关闭后就消失了，其存在时间是短暂的。硬盘Cookie保存在硬盘里，有一个过期时间，除非用户手工清理或到了过期时间，硬盘Cookie不会被删除，其存在时间是长期的。所以，按存在时间，可分为`非持久Cookie`和`持久Cookie`。

cookie 是一个非常具体的东西，指的就是浏览器里面能永久存储的一种数据，仅仅是浏览器实现的一种数据存储功能。

`cookie由服务器生成，发送给浏览器`，浏览器把cookie以key-value形式保存到某个目录下的文本文件内，下一次请求同一网站时会把该cookie发送给服务器。由于cookie是存在客户端上的，所以浏览器加入了一些限制确保cookie不会被恶意使用，同时不会占据太多磁盘空间，所以每个域的cookie数量是有限的。

Session

Session字面意思是会话，主要用来标识自己的身份。比如在无状态的api服务在多次请求数据库时，如何知道是同一个用户，这个就可以通过session的机制，服务器要知道当前发请求给自己的是谁

为了区分客户端请求，`服务端会给具体的客户端生成身份标识session`，然后客户端每次向服务器发请求的时候，都带上这个“身份标识”，服务器就知道这个请求来自于谁了。

至于客户端如何保存该标识，可以有很多方式，对于浏览器而言，一般都是使用`cookie`的方式

服务器使用session把用户信息临时保存了服务器上，用户离开网站就会销毁，这种凭证存储方式相对于cookie来说更加安全，但是session会有一个缺陷: 如果web服务器做了负载均衡，那么下一个操作请求到了另一台服务器的时候session会丢失。

因此，通常企业里会使用`redis,memcached`缓存中间件来实现session的共享，此时web服务器就是一个完全无状态的存在，所有的用户凭证可以通过共享session的方式存取，当前session的过期和销毁机制需要用户做控制。

Token

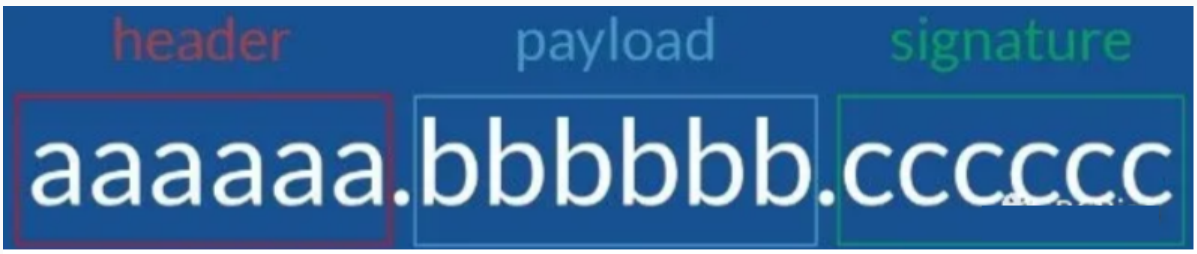
token的意思是“令牌”，是用户身份的验证方式，最简单的token组成: `uid(用户唯一标识)`+`time(当前时间戳)`+`sign(签名,由token的前几位+盐以哈希算法压缩成一定长度的十六进制字符串)`，同时还可以将不变的参数也放进token

这里我们主要想讲的就是`Json Web Token`，也就是本篇的主题:JWT

2 Json-Web-Token(JWT)介绍

一般而言，用户注册登陆后会生成一个jwt token返回给浏览器，浏览器向服务端请求数据时携带`token`，服务器端使用`signature`中定义的方式进行解码，进而对token进行解析和验证。

JWT Token组成部分



JWT-Token组成部分

- header: 用来指定使用的算法(HMAC SHA256 RSA)和token类型(如JWT)
- payload: 包含声明(要求)，声明通常是用户信息或其他数据的声明，比如用户id，名称，邮箱等. 声明可分为三种: registered,public,private
- signature: 用来保证JWT的真实性，可以使用不同的算法

header

```
{
    "alg": "HS256",
    "typ": "JWT"
}
```

对上面的json进行base64编码即可得到JWT的第一个部分

payload

- registered claims: 预定义的声明，通常会放置一些预定义字段，比如过期时间，主题等(iss:issuer,exp:expiration time,sub:subject,aud:audience)
- public claims: 可以设置公开定义的字段
- private claims: 用于统一使用他们的各方之间的共享信息

```
{
    "sub": "xxx-api",
    "name": "bgbiao.top",
    "admin": true
}
```

对payload部分的json进行base64编码后即可得到JWT的第二个部分

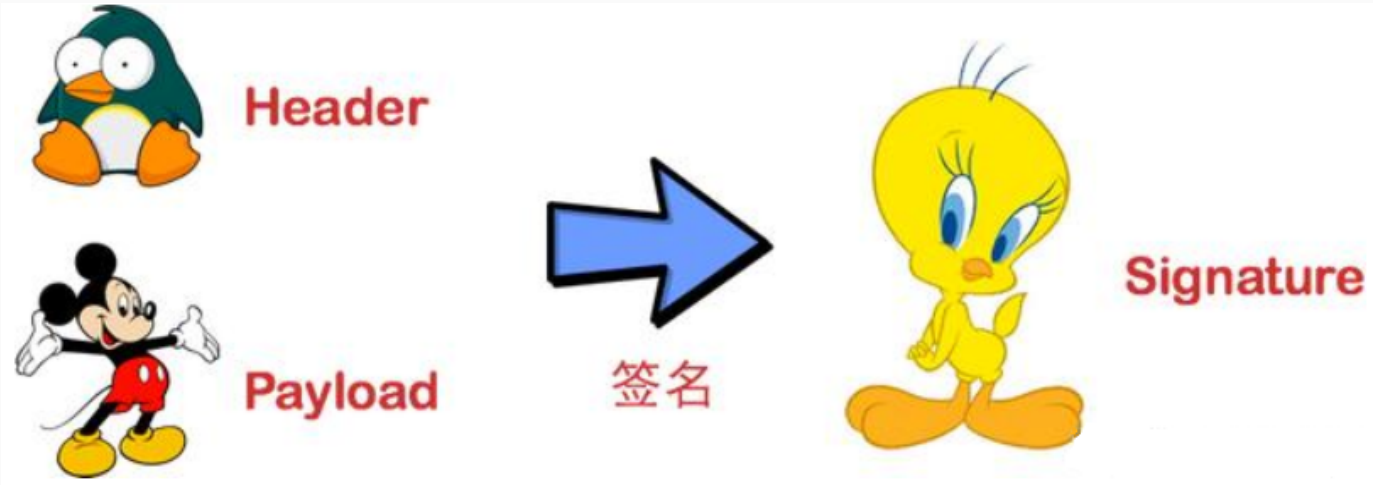
`注意：` 不要在header和payload中放置敏感信息，除非信息本身已经做过脱敏处理

signature

为了得到签名部分，必须有编码过的header和payload，以及一个秘钥，签名算法使用header中指定的那个，然后对其进行签名即可

`HMACSHA256(base64UrlEncode(header)+"."+base64UrlEncode(payload),secret)`

签名是`用于验证消息在传递过程中有没有被更改`，并且，对于使用私钥签名的token，它还可以验证JWT的发送方是否为它所称的发送方。



签名的目的

最后一步签名的过程，实际上是对头部以及载荷内容进行签名。一般而言，加密算法对于不同的输入产生的输出总是不一样的。对于两个不同的输入，产生同样的输出的概率极其地小（有可能比我成世界首富的概率还小）。所以，我们就把“不一样的输入产生不一样的输出”当做必然事件来看待吧。

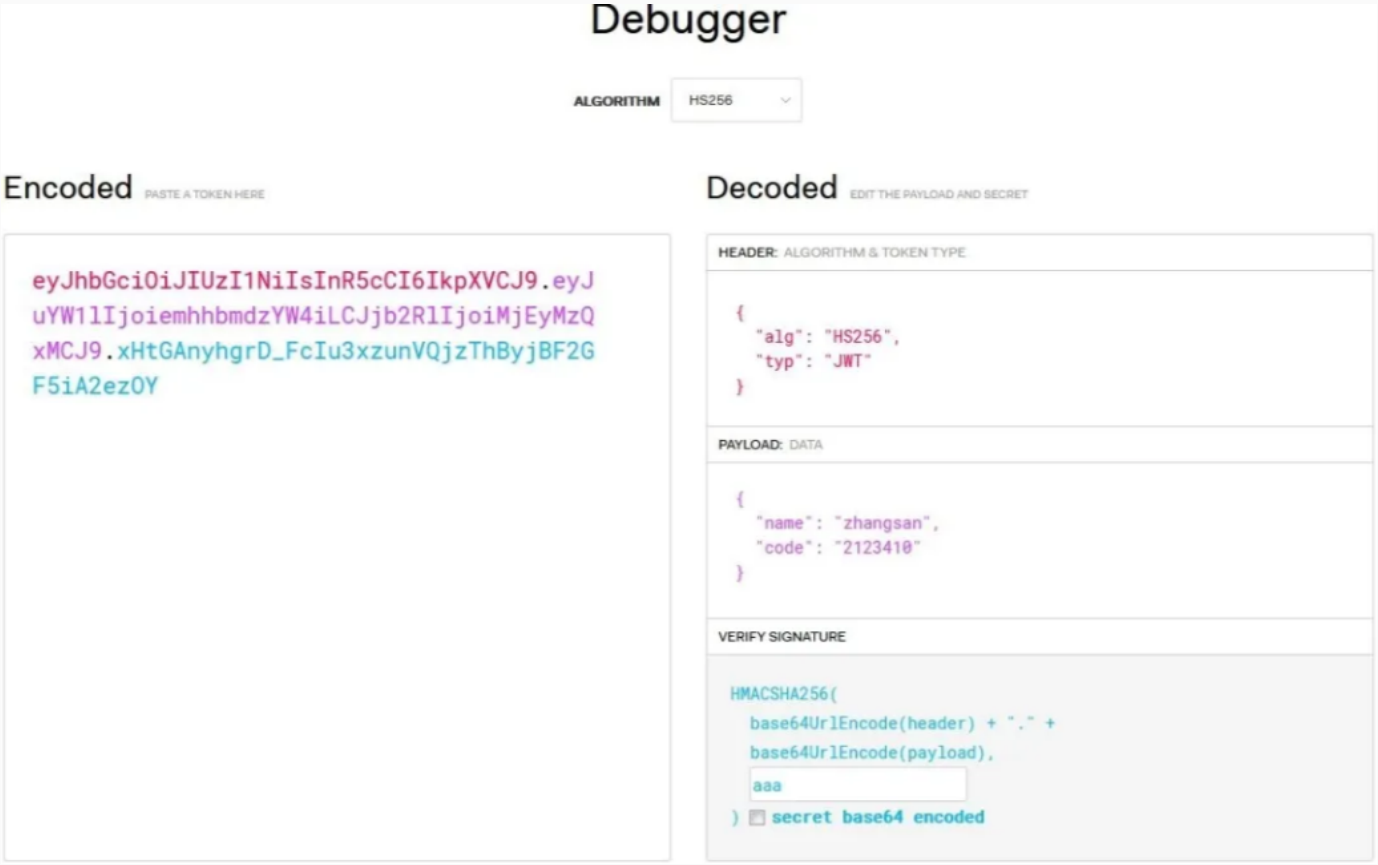
所以，如果有人对头部以及载荷的内容解码之后进行修改，再进行编码的话，那么新的头部和载荷的签名和之前的签名就将是不一样的。而且，如果不知道服务器加密的时候用的密钥的话，得出来的签名也一定会是不一样的。

服务器应用在接受到JWT后，会首先对头部和载荷的内容用同一算法再次签名。那么服务器应用是怎么知道我们用的是哪一种算法呢？别忘了，我们在JWT的头部中已经用alg字段指明了我们的加密算法了。

如果服务器应用对头部和载荷再次以同样方法签名之后发现，自己计算出来的签名和接受到的签名不一样，那么就说明这个Token的内容被别人动过的，我们应该拒绝这个Token，返回一个HTTP 401 Unauthorized响应。

注意：在JWT中，不应该在载荷里面加入任何敏感的数据，比如用户的密码。

在jwt.io网站中，提供了一些JWT token的编码，验证以及生成jwt的工具。
下图就是一个典型的jwt-token的组成部分。

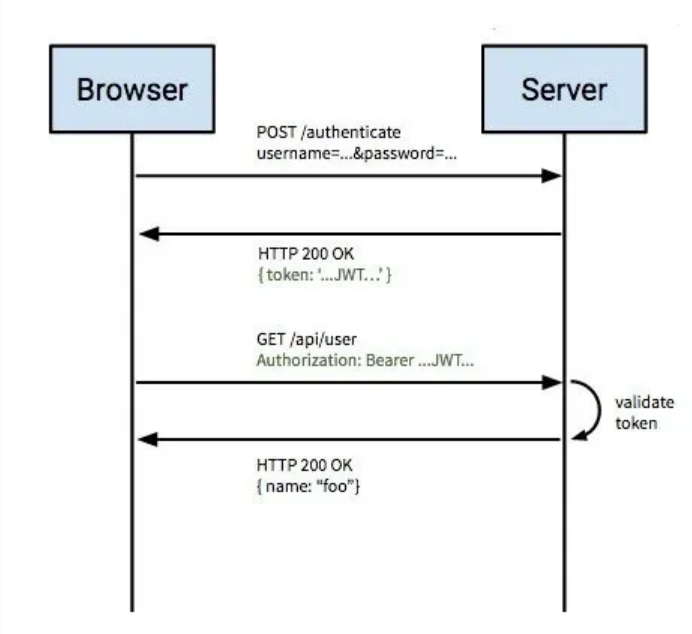


jwt官方签名结构

什么时候用JWT

- Authorization(授权): 典型场景，用户请求的token中包含了该令牌允许的路由，服务和资源。单点登录其实就是现在广泛使用JWT的一个特性
- Information Exchange(信息交换): 对于安全的在各方之间传输信息而言，JSON Web Tokens无疑是一种很好的方式.因为JWTs可以被签名，例如，用公钥/私钥对，你可以确定发送人就是它们所说的那个人。另外，由于签名是使用头和有效负载计算的，您还可以验证内容没有被篡改

JWT(Json Web Tokens)是如何工作的



JWT认证过程

所以，基本上整个过程分为两个阶段，第一个阶段，客户端向服务端获取token，第二阶段，客户端带着该token去请求相关的资源.

通常比较重要的是，服务端如何根据指定的规则进行token的生成。

在认证的时候，当用户用他们的凭证成功登录以后，一个JSON Web Token将会被返回。

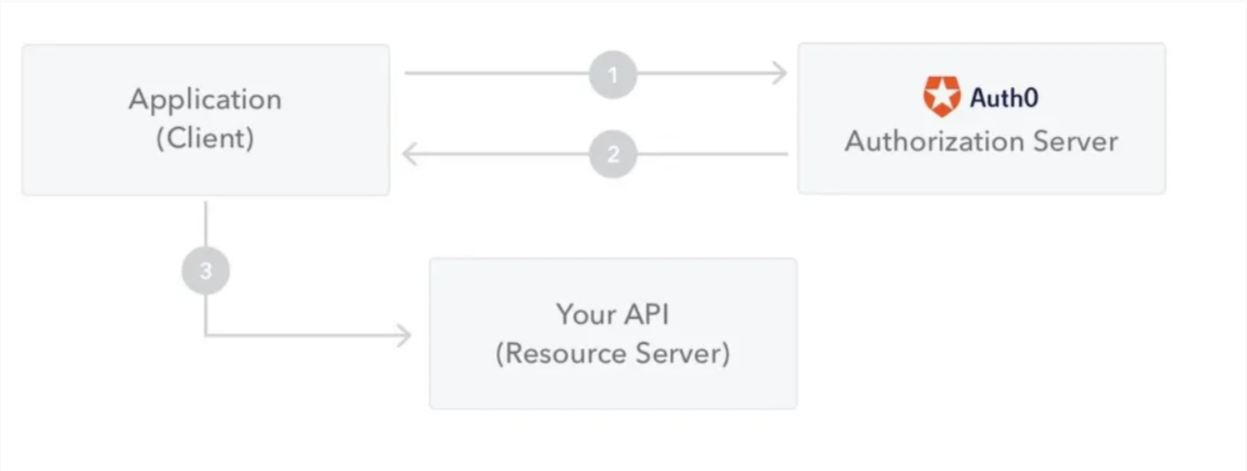
此后，token就是用户凭证了，你必须非常小心以防止出现安全问题。

一般而言，你保存令牌的时候不应该超过你所需要它的时间。

无论何时用户想要访问受保护的路由或者资源的时候，用户代理（通常是浏览器）都应该带上JWT，典型的，通常放在Authorization header中，用Bearer schema:`Authorization: Bearer <token>`

服务器上的受保护的路由将会检查Authorization header中的JWT是否有效，如果有效，则用户可以访问受保护的资源。如果JWT包含足够多的必需的数据，那么就可以减少对某些操作的数据库查询的需要，尽管可能并不总是如此。

如果token是在授权头（Authorization header）中发送的，那么跨源资源共享(CORS)将不会成为问题，因为它不使用cookie.



获取JWT以及访问APIs以及资源

- 客户端向授权接口请求授权
- 服务端授权后返回一个access token给客户端
- 客户端使用access token访问受保护的资源

3 基于Token的身份认证和基于服务器的身份认证

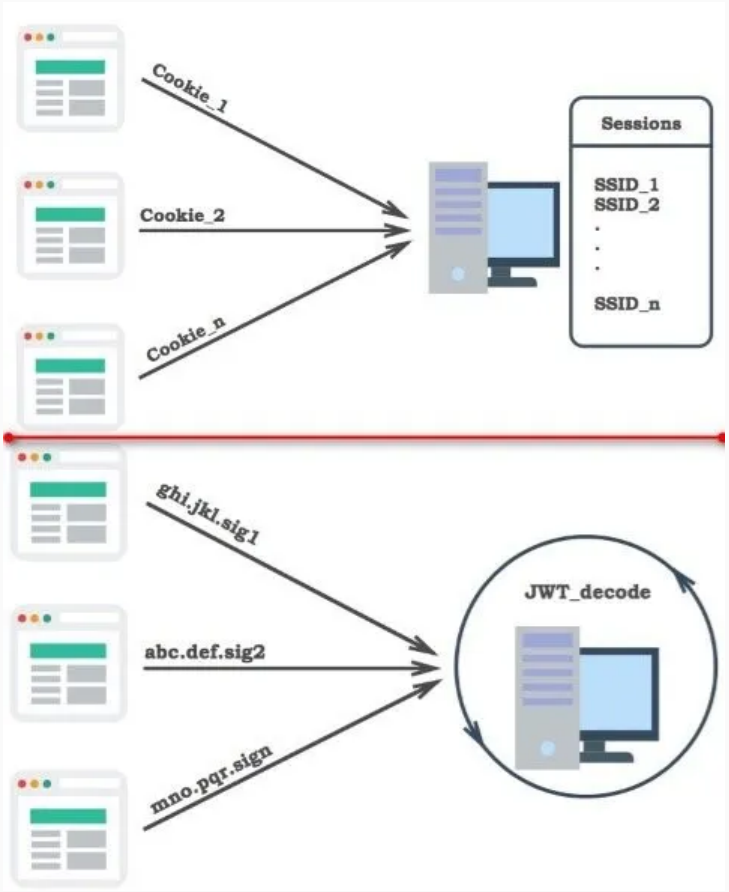
1.基于服务器的认证

前面说到过session，cookie以及token的区别，在之前传统的做法就是基于存储在服务器上的session来做用户的身份认证，但是通常会有如下问题:

- Sessions: 认证通过后需要将用户的session数据保存在内存中，随着认证用户的增加，内存开销会大
- 扩展性: 由于session存储在内存中，扩展性会受限，虽然后期可以使用redis,memcached来缓存数据
- CORS: 当多个终端访问同一份数据时，可能会遇到禁止请求的问题
- CSRF: 用户容易受到CSRF攻击

2.Session和JWT Token的异同

都可以存储用户相关信息，但是session存储在服务端，JWT存储在客户端



session和jwt数据存储位置

3.基于Token的身份认证如何工作

基于Token的身份认证是无状态的，服务器或者session中不会存储任何用户信息.(很好的解决了共享session的问题)

- 用户携带用户名和密码请求获取token(接口数据中可使用appId,appKey)
- 服务端校验用户凭证，并返回用户或客户端一个Token
- 客户端存储token,并在请求头中携带Token
- 服务端校验token并返回数据

`注意：`

- 随后客户端的每次请求都需要使用token
- token应该放在header中
- 需要将服务器设置为接收所有域的请求: `Access-Control-Allow-Origin: *`

4.用Token的好处

- 无状态和可扩展性
- 安全: 防止CSRF攻击;token过期重新认证

5.JWT和OAuth的区别

- 1.OAuth2是一种授权框架 ， JWT是一种认证协议
- 2.无论使用哪种方式切记用HTTPS来保证数据的安全性
- 3.OAuth2用在`使用第三方账号登录的情况`(比如使用weibo, qq, github登录某个app)， 而`JWT是用在前后端分离`, 需要简单的对后台API进行保护时使用

4 Go范例

使用的go jwt库： https://github.com/dgrijalva/jwt-go，这个库有6k多的start，确实是很受欢迎，所以用这个库来写demo

demo例子：

```
1  package main
2
3  import (
4      "errors"
5      "fmt"
6      "net/http"
7      "time"
8
9      jwt "github.com/dgrijalva/jwt-go"
10     "github.com/gin-gonic/gin"
11 )
```

```
12
13 const (
14     ErrorServerBusy = "server is busy"
15     ErrorReLogin    = "relogin"
16 )
17
18 type JWTClaims struct {
19     jwt.StandardClaims
20     UserID   int    `json:"user_id"`
21     Password string `json:"password"`
22     Username string `json:"username"`
23 }
24
25 var (
26     Secret     = "123#111" //salt
27     ExpireTime = 3600      //token expire time
28 )
29
30 func main() {
31     r := gin.Default()
32     r.GET("/login/:username/:password", login)
33     r.GET("/verify/:token", verify)
34     r.GET("/refresh/:token", refresh)
35     r.GET("/sayHello/:token", sayHello)
36     _ = r.Run(":8080")
37 }
38
39 //generate jwt token
40 func genToken(claims *JWTClaims) (string, error) {
41     token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
42     signedToken, err := token.SignedString([]byte(Secret))
43     if err != nil {
44         return "", errors.New(ErrorServerBusy)
45     }
46     return signedToken, nil
47 }
48
49 //登录，获取jwt token
50 func login(c *gin.Context) {
51     username := c.Param("username")
```

```
52      password := c.Param("password")
53      claims := &JWTClaims{
54          UserID:   1,
55          Username: username,
56          Password: password,
57      }
58      claims.IssuedAt = time.Now().Unix()
59      claims.ExpiresAt = time.Now().Add(time.Second * time.Duratio
    n(ExpireTime)).Unix()
60      singedToken, err := genToken(claims)
61      if err != nil {
62          c.String(http.StatusNotFound, err.Error())
63          return
64      }
65      c.String(http.StatusOK, singedToken)
66  }
67
68  //验证jwt token
69  func verifyAction(strToken string) (*JWTClaims, error) {
70      token, err := jwt.ParseWithClaims(strToken, &JWTClaims{}, fu
    nc(token *jwt.Token) (interface{}, error) {
71          return []byte(Secret), nil
72      })
73      if err != nil {
74          return nil, errors.New(ErrorServerBusy)
75      }
76      claims, ok := token.Claims.(*JWTClaims)
77      if !ok {
78          return nil, errors.New(ErrorReLogin)
79      }
80      if err := token.Claims.Valid(); err != nil {
81          return nil, errors.New(ErrorReLogin)
82      }
83      fmt.Println("verify")
84      return claims, nil
85  }
86  func sayHello(c *gin.Context) {
87      strToken := c.Param("token")
88      claim, err := verifyAction(strToken)
89      if err != nil {
```

```
90          c.String(http.StatusNotFound, err.Error())
91      }
92      c.String(http.StatusOK, "hello, ", claim.Username)
93  }
94  func verify(c *gin.Context) {
95      strToken := c.Param("token")
96      claim, err := verifyAction(strToken)
97      if err != nil {
98          c.String(http.StatusNotFound, err.Error())
99          return
100     }
101     c.String(http.StatusOK, "verify: ", claim.Username)
102 }
103 func refresh(c *gin.Context) {
104     strToken := c.Param("token")
105     claims, err := verifyAction(strToken)
106     if err != nil {
107         c.String(http.StatusNotFound, err.Error())
108         return
109     }
110     claims.ExpiresAt = time.Now().Unix() + (claims.ExpiresAt - c
    laims.IssuedAt)
111     signedToken, err := genToken(claims)
112     if err != nil {
113         c.String(http.StatusNotFound, err.Error())
114         return
115     }
116     c.String(http.StatusOK, signedToken, ", ", claims.ExpiresAt)
117 }
```

5 JWT资源

对于jwt的了解： https://jwt.io

jwt介绍：https://jwt.io/introduction/

rfc： https://tools.ietf.org/html/rfc7519

jwt方法参数定义的介绍：https://www.iana.org/assignments/jwt/jwt.xhtml

jwt handbook: https://auth0.com/resources/ebooks/jwt-handbook

JWT各种库：https://jwt.io/#libraries

6 使用Gin框架集成JWT

在Golang语言中，jwt-go库提供了一些jwt编码和验证的工具，因此我们很容易使用该库来实现token认证。
另外，我们也知道gin框架中支持用户自定义middleware，我们可以很好的将jwt相关的逻辑封装在middleware中，然后对具体的接口进行认证。

自定义中间件

在gin框架中，自定义中间件比较容易，只要返回一个`gin.HandlerFunc`即完成一个中间件定义。
接下来，我们先定义一个用于jwt认证的中间件.

```
// 定义一个JWTAuth的中间件
func JWTAuth() gin.HandlerFunc {
    return func(c *gin.Context) {
        // 通过http header中的token解析来认证
        token := c.Request.Header.Get("token")
        if token == "" {
            c.JSON(http.StatusOK, gin.H{
                "status": -1,
                "msg":    "请求未携带token，无权限访问",
                "data":   nil,
            })
            c.Abort()
            return
        }
        log.Print("get token: ", token)
        // 初始化一个JWT对象实例，并根据结构体方法来解析token
        j := NewJWT()
        // 解析token中包含的相关信息(有效载荷)
        claims, err := j.ParserToken(token)
        if err != nil {
            // token过期
            if err == TokenExpired {
                c.JSON(http.StatusOK, gin.H{
                    "status": -1,
                    "msg":    "token授权已过期，请重新申请授权",
                    "data":   nil,
                })
```

```
                c.Abort()
                return
            }
            // 其他错误
            c.JSON(http.StatusOK, gin.H{
                "status": -1,
                "msg":    err.Error(),
                "data":   nil,
            })
            c.Abort()
            return
        }
        // 将解析后的有效载荷claims重新写入gin.Context引用对象中
        c.Set("claims", claims)
    }
}
```

定义jwt编码和解码逻辑

根据前面提到的jwt-token的组成部分，以及`jwt-go`中相关的定义，我们可以使用如下方法进行生成token.

```
// 定义一个jwt对象
type JWT struct {
    // 声明签名信息
    SigningKey []byte
}
// 初始化jwt对象
func NewJWT() *JWT {
    return &JWT{
        []byte("bgbiao.top"),
    }
}
// 自定义有效载荷(这里采用自定义的Name和Email作为有效载荷的一部分)
type CustomClaims struct {
    Name  string `json:"name"`
    Email string `json:"email"`
    // StandardClaims结构体实现了Claims接口(Valid()函数)
    jwt.StandardClaims
}
```

```go
// 调用jwt-go库生成token
// 指定编码的算法为jwt.SigningMethodHS256
func (j *JWT) CreateToken(claims CustomClaims) (string, error) {
    // https://gowalker.org/github.com/dgrijalva/jwt-go#Token
    // 返回一个token的结构体指针
    token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
    return token.SignedString(j.SigningKey)
}
// token解码
func (j *JWT) ParserToken(tokenString string) (*CustomClaims, error) {
    // https://gowalker.org/github.com/dgrijalva/jwt-go#ParseWithClaims
    // 输入用户自定义的Claims结构体对象,token,以及自定义函数来解析token字符串为jwt的Token结构体指针
    // Keyfunc是匿名函数类型: type Keyfunc func(*Token) (interface{}, error)
    // func ParseWithClaims(tokenString string, claims Claims, keyFunc Keyfunc) (*Token, error) {}
    token, err := jwt.ParseWithClaims(tokenString, &CustomClaims{}, func(token *jwt.Token) (interface{}, error) {
        return j.SigningKey, nil
    })
    if err != nil {
        // https://gowalker.org/github.com/dgrijalva/jwt-go#ValidationError
        // jwt.ValidationError 是一个无效token的错误结构
        if ve, ok := err.(*jwt.ValidationError); ok {
            // ValidationErrorMalformed是一个uint常量，表示token不可用
            if ve.Errors&jwt.ValidationErrorMalformed != 0 {
                return nil, fmt.Errorf("token不可用")
                // ValidationErrorExpired表示Token过期
            } else if ve.Errors&jwt.ValidationErrorExpired != 0 {
                return nil, fmt.Errorf("token过期")
                // ValidationErrorNotValidYet表示无效token
            } else if ve.Errors&jwt.ValidationErrorNotValidYet != 0 {
                return nil, fmt.Errorf("无效的token")
            } else {
```

```
                return nil, fmt.Errorf("token不可用")
            }
        }
    }
    // 将token中的claims信息解析出来并断言成用户自定义的有效载荷结构
    if claims, ok := token.Claims.(*CustomClaims); ok && token.Va
lid {
        return claims, nil
    }
    return nil, fmt.Errorf("token无效")
}
```

定义登陆验证逻辑

接下来的部分就是普通api的具体逻辑了，比如可以在登陆时进行用户校验，成功后未该次认证请求生成token。

```
// 定义登陆逻辑
// model.LoginReq中定义了登陆的请求体(name,passwd)
func Login(c *gin.Context) {
    var loginReq model.LoginReq
    if c.BindJSON(&loginReq) == nil {
        // 登陆逻辑校验(查库，验证用户是否存在以及登陆信息是否正确)
        isPass, user, err := model.LoginCheck(loginReq)
        // 验证通过后为该次请求生成token
        if isPass {
            generateToken(c, user)
        } else {
            c.JSON(http.StatusOK, gin.H{
                "status": -1,
                "msg":    "验证失败" + err.Error(),
                "data":   nil,
            })
        }
    } else {
        c.JSON(http.StatusOK, gin.H{
            "status": -1,
            "msg":    "用户数据解析失败",
            "data":   nil,
        })
```

```
    }
}
// token生成器
// md 为上面定义好的middleware中间件
func generateToken(c *gin.Context, user model.User) {
    // 构造SignKey: 签名和解签名需要使用一个值
    j := md.NewJWT()
    // 构造用户claims信息(负荷)
    claims := md.CustomClaims{
        user.Name,
        user.Email,
        jwtgo.StandardClaims{
            NotBefore: int64(time.Now().Unix() - 1000), // 签名生效时间
            ExpiresAt: int64(time.Now().Unix() + 3600), // 签名过期时间
            Issuer:    "bgbiao.top",                    // 签名颁发者
        },
    }
    // 根据claims生成token对象
    token, err := j.CreateToken(claims)
    if err != nil {
        c.JSON(http.StatusOK, gin.H{
            "status": -1,
            "msg":    err.Error(),
            "data":   nil,
        })
    }
    log.Println(token)
    // 封装一个响应数据,返回用户名和token
    data := LoginResult{
        Name:  user.Name,
        Token: token,
    }
    c.JSON(http.StatusOK, gin.H{
        "status": 0,
        "msg":    "登陆成功",
        "data":   data,
    })
```

```
61      return
62  }
```

定义普通待验证接口

```
1  // 定义一个普通controller函数，作为一个验证接口逻辑
2  func GetDataByTime(c *gin.Context) {
3      // 上面我们在JWTAuth()中间中将'claims'写入到gin.Context的指针对象
   中，因此在这里可以将之解析出来
4      claims := c.MustGet("claims").(*md.CustomClaims)
5      if claims != nil {
6          c.JSON(http.StatusOK, gin.H{
7              "status": 0,
8              "msg":    "token有效",
9              "data":   claims,
10         })
11     }
12 }
13 // 在主函数中定义路由规则
14     router := gin.Default()
15     v1 := router.Group("/apis/v1/")
16     {
17         v1.POST("/register", controller.RegisterUser)
18         v1.POST("/login", controller.Login)
19     }
20     // secure v1
21     sv1 := router.Group("/apis/v1/auth/")
22     // 加载自定义的JWTAuth()中间件,在整个sv1的路由组中都生效
23     sv1.Use(md.JWTAuth())
24     {
25         sv1.GET("/time", controller.GetDataByTime)
26     }
27     router.Run(":8081")
```

验证使用JWT后的接口

```
1  # 运行项目
2  $ go run main.go
3  127.0.0.1
```

```
13306
root:bgbiao.top@tcp(127.0.0.1:13306)/test_api?charset=utf8mb4&parseTime=True&loc=Local
[GIN-debug] [WARNING] Creating an Engine instance with the Logger and Recovery middleware already attached.
[GIN-debug] [WARNING] Running in "debug" mode. Switch to "release" mode in production.
 - using env:   export GIN_MODE=release
 - using code:  gin.SetMode(gin.ReleaseMode)
[GIN-debug] POST   /apis/v1/register         --> warnning-trigger/controller.RegisterUser (3 handlers)
[GIN-debug] POST   /apis/v1/login            --> warnning-trigger/controller.Login (3 handlers)
[GIN-debug] GET    /apis/v1/auth/time        --> warnning-trigger/controller.GetDataByTime (4 handlers)
[GIN-debug] Listening and serving HTTP on :8081
# 注册用户
$ curl -i -X POST \
   -H "Content-Type:application/json" \
   -d \
'{
  "name": "hahaha1",
  "password": "hahaha1",
  "email": "hahaha1@bgbiao.top",
  "phone": 10000000000
}' \
 'http://localhost:8081/apis/v1/register'
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Sun, 15 Mar 2020 07:09:28 GMT
Content-Length: 41
{"data":null,"msg":"success ","status":0}%
# 登陆用户以获取token
$ curl -i -X POST \
   -H "Content-Type:application/json" \
   -d \
'{
  "name":"hahaha1",
  "password":"hahaha1"
}' \
```

```
 'http://localhost:8081/apis/v1/login'
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Sun, 15 Mar 2020 07:10:41 GMT
Content-Length: 290
{"data":{"token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyTmFtZSI6ImhhaGFoYTEiLCJlbWFpbCI6ImhhaGFoYTFAYmdiaWFvLnRvcCIsImV4cCI6MTU4NDI1OTg0MSwiaXNzIjoiYmdiaWFvLnRvcCIsIm5iZiI6MTU4NDI1NTI0MX0.HNXSKISZTqzjKd705BOSARmgI8FGGe4Sv-Ma3_iK1Xw","name":"hahaha1"},"msg":"登陆成功","status":0}
# 访问需要认证的接口
# 因为我们对/apis/v1/auth/的分组路由中加载了jwt的middleware，因此该分组下的api都需要使用jwt-token认证
$ curl http://localhost:8081/apis/v1/auth/time
{"data":null,"msg":"请求未携带token，无权限访问","status":-1}%
# 使用token认证
$ curl http://localhost:8081/apis/v1/auth/time -H 'token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyTmFtZSI6ImhhaGFoYTEiLCJlbWFpbCI6ImhhaGFoYTFAYmdiaWFvLnRvcCIsImV4cCI6MTU4NDI1OTg0MSwiaXNzIjoiYmdiaWFvLnRvcCIsIm5iZiI6MTU4NDI1NTI0MX0.HNXSKISZTqzjKd705BOSARmgI8FGGe4Sv-Ma3_iK1Xw'
{"data":{"userName":"hahaha1","email":"hahaha1@bgbiao.top","exp":1584259841,"iss":"bgbiao.top","nbf":1584255241},"msg":"token有效","status":0}%
```

参考源码: gin-jwt-token

7 使用go进行 JWT 验证

对于使用负载均衡的服务器来说,使用 JWT(JSON WEB TOKEN) 是一个更优的选择,session受到单台服务器的限制,一个用户登录过后就只能分配到
这一台服务器上,这和负载均衡的初衷不一致啊,而 jwt 就解决了这类的痛点

使用 JWT 的场景

- 身份验证 用户在登录过后服务器会用 jwt 返回用户可访问的资源,比如权限什么的
- 传递信息 通过 jwt 的`header`和`signature`可以保证`payload`没有被篡改,保证信息的安全

JWT 的结构

JWT 是由`header,payload,signature`三部分组成的,咱们先用例子说话

- header

```
{
  "alg": "HS256",
  "typ": "JWT"
}
// base64编码的字符串`eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9`
```

这里规定了加密算法,hash256

- payload

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
// base64编码的字符串`eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG
9lIiwiYWRtaW4iOnRydWV9`
```

```
这里的内容没有强制要求,因为 paylaod 就是为了承载内容而存在的,不过想用规范的话也
可以参考下面的
* iss: jwt签发者
* sub: jwt所面向的用户
* aud: 接收jwt的一方
* exp: jwt的过期时间，这个过期时间必须要大于签发时间
* nbf: 定义在什么时间之前，该jwt都是不可用的.
* iat: jwt的签发时间
* jti: jwt的唯一身份标识，主要用来作为一次性token,从而回避重放攻击。
```

- signature

是用 `header + payload + secret` 组合起来加密的,公式是:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

这里 `secret` 就是自己定义的一个随机字符串,这一个过程只能发生在 server 端,会随机生成一个 hash 值这样组合起来之后就是一个完整的 jwt 了:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwib
mFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.4c9540f793ab33b13670169bdf
```

```
444c1eb1c37047f18e861981e14e34587b1e04
```

这里有一个用 go 加密和验证 jwt 的 demo

总结

选择 jwt 最大的理由:

1. 内容有公钥私钥,可以保证内容的合法性
2. token 中可以包含很多信息

不过 jwt 不保证的安全问题:

1. 因为`header,paylaod`是 base64编码,相当于明文可见的,因此不能在`payload`中放入敏感信息
2. 并不能保证数据传输时会不会被盗用,这一点和 sessionID 一样,因此不要迷信它有多高的安全性..