

26-IAM项目是如何设计和实现访问认证功能的？

你好，我是孔令飞。

上一讲，我们学习了应用认证常用的四种方式，今天我们来看下IAM项目是如何设计和实现认证功能的。

这一讲，我仍然通过HTTP服务来讲解认证功能。Basic、Digest、OAuth、Bearer这四种常用的认证方式里，Go后端服务中用得最多的是 Basic 和 Bearer 两种。其中，Basic主要用在前端登陆的场景下，Bearer主要用在调用后端 API 服务的场景下。

今天，我们通过IAM项目iam-apiserver和iam-authz-server服务认证功能的设计和实现，来看下在实际的Go项目开发中，具体是如何设计和实现认证功能的。

如何设计IAM项目的认证功能？

在认证功能开发之前，我们要根据需求，认真考虑下如何设计认证功能，并在设计阶段通过技术评审。那么我们先来看下，如何设计IAM项目的认证功能。

首先，我们要**梳理清楚认证功能的使用场景和需求**。

- IAM项目的iam-apiserver服务，提供了IAM系统的管理流功能接口，它的客户端可以是前端（这里也叫控制台），也可以是App端。
- 为了方便用户在Linux系统下调用，IAM项目还提供了iamctl命令行工具。
- 为了支持在第三方代码中调用iam-apiserver提供的API接口，还支持了API调用。
- 为了提高用户在代码中调用API接口的效率，IAM项目提供了Go SDK。

可以看到，iam-apiserver有很多客户端，每种客户端适用的认证方式是有区别的。

控制台、App端需要登录系统，所以需要用户名：密码这种认证方式，也即Basic认证。iamctl、API调用、Go SDK因为可以不用登录系统，所以可以采用更安全的认证方式：Bearer认证。同时，Basic认证作为iam-apiserver已经集成的认证方式，仍然可以供iamctl、API调用、Go SDK使用。

这里有个地方需要注意：如果iam-apiserver采用Bearer Token的认证方式，目前最受欢迎的Token格式是JWT Token。而JWT Token需要密钥（后面统一用secretKey来指代），因此需要在iam-apiserver服务中为每个用户维护一个密钥，这样会增加开发和维护成本。

业界有一个更好的实现方式：将iam-apiserver提供的API接口注册到API网关中，通过API网关中的Token认证功能，来实现对iam-apiserver API接口的认证。有很多API网关可供选择，例如腾讯云API网关、Tyk、Kong等。

这里需要你注意：通过iam-apiserver创建的密钥对是提供给iam-authz-server使用的。

另外，我们还需要调用iam-authz-server提供的RESTful API接口：`/v1/authz`，来进行资源授权。API调用比较适合采用的认证方式是Bearer认证。

当然，`/v1/authz`也可以直接注册到API网关中。在实际的Go项目开发中，也是我推荐的一种方式。但在

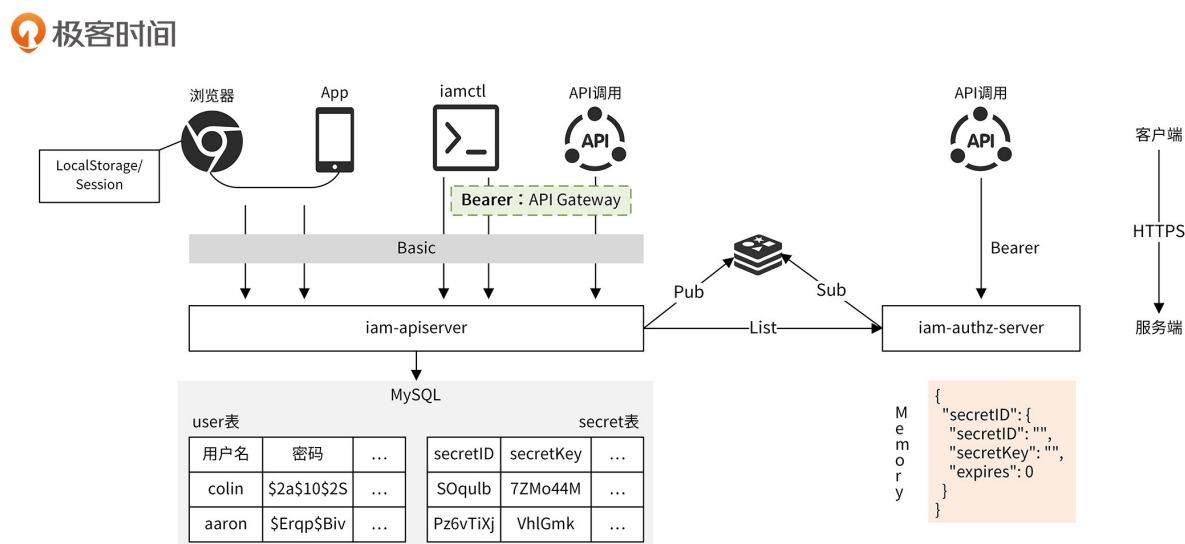
这里，为了展示实现Bearer认证的过程，iam-authz-server自己实现了Bearer认证。讲到iam-authz-server Bearer认证实现的时候，我会详细介绍这一点。

Basic认证需要用户名和密码，Bearer认证则需要密钥，所以iam-apiserver需要将用户名/密码、密钥等信息保存在后端的MySQL中，持久存储起来。

在进行认证的时候，需要获取密码或密钥进行反加密，这就需要查询密码或密钥。查询密码或密钥有两种方式。一种是在请求到达时查询数据库。因为数据库的查询操作延时高，会导致API接口延时较高，所以不太适合用在数据流组件中。另外一种是将密码或密钥缓存在内存中，这样请求到来时，就可以直接从内存中查询，从而提升查询速度，提高接口性能。

但是，将密码或密钥缓存在内存中时，就要考虑内存和数据库的数据一致性，这会增加代码实现的复杂度。因为管控流组件对性能延时要求不那么敏感，而数据流组件则一定要实现非常高的接口性能，所以iam-apiserver在请求到来时查询数据库，而iam-authz-server则将密钥信息缓存在内存中。

那在这里，可以总结出一张IAM项目的认证设计图：



另外，为了将控制流和数据流区分开来，密钥的CURD操作也放在了iam-apiserver中，但是iam-authz-server需要用到这些密钥信息。为了解决这个问题，目前的做法是：

- iam-authz-server通过gRPC API请求iam-apiserver，获取所有的密钥信息；
- 当iam-apiserver有密钥更新时，会Pub一条消息到Redis Channel中。因为iam-authz-server订阅了同一个Redis Channel，iam-authz-server监听到channel有新消息时，会获取、解析消息，并更新它缓存的密钥信息。这样，我们就能确保iam-authz-server内存中缓存的密钥和iam-apiserver中的密钥保持一致。

学到这里，你可能会问：将所有密钥都缓存在iam-authz-server中，那岂不是要占用很大的内存？别担心，这个问题我也想过，并且替你计算好了：8G的内存大概能保存约8千万个密钥信息，完全够用。后期不够用的话，可以加大内存。

不过这里还是有个小缺陷：如果Redis down掉，或者出现网络抖动，可能会造成iam-apiserver中和iam-authz-server内存中保存的密钥数据不一致，但这不妨碍我们学习认证功能的设计和实现。至于如何保证缓

存系统的数据一致性，我会在新一期的特别放送里专门介绍下。

最后注意一点：Basic 认证请求和 Bearer 认证请求都可能被截获并重放。所以，为了确保Basic认证和Bearer认证的安全性，**和服务端通信时都需要配合使用HTTPS协议**。

IAM项目是如何实现Basic认证的？

我们已经知道，IAM项目中主要用了Basic 和 Bearer 这两种认证方式。我们要支持Basic认证和Bearer认证，并根据需要选择不同的认证方式，这很容易让我们想到使用设计模式中的策略模式来实现。所以，在IAM项目中，我将每一种认证方式都视作一个策略，通过选择不同的策略，来使用不同的认证方法。

IAM项目实现了如下策略：

- [auto策略](#)：该策略会根据HTTP头Authorization: Basic XX.YY.ZZ和Authorization: Bearer XX.YY.ZZ自动选择使用Basic认证还是Bearer认证。
- [basic策略](#)：该策略实现了Basic认证。
- [jwt策略](#)：该策略实现了Bearer认证，JWT是Bearer认证的具体实现。
- [cache策略](#)：该策略其实是一个Bearer认证的实现，Token采用了JWT格式，因为Token中的密钥ID是从内存中获取的，所以叫Cache认证。这一点后面会详细介绍。

iam-apiserver通过创建需要的认证策略，并加载到需要认证的API路由上，来实现API认证。具体代码如下：

```
jwtStrategy, _ := newJWTAuth().(auth.JWTStrategy)
g.POST("/login", jwtStrategy.LoginHandler)
g.POST("/logout", jwtStrategy.LogoutHandler)
// Refresh time can be longer than token timeout
g.POST("/refresh", jwtStrategy.RefreshHandler)
```

上述代码中，我们通过[newJWTAuth](#)函数创建了auth.JWTStrategy类型的变量，该变量包含了一些认证相关函数。

- LoginHandler：实现了Basic认证，完成登陆认证。
- RefreshHandler：重新刷新Token的过期时间。
- LogoutHandler：用户注销时调用。登陆成功后，如果在Cookie中设置了认证相关的信息，执行LogoutHandler则会清空这些信息。

下面，我来分别介绍下LoginHandler、RefreshHandler和LogoutHandler。

1. LoginHandler

这里，我们来看下LoginHandler Gin中间件，该函数定义位于github.com/appleboy/gin-jwt包的[auth_jwt.go](#)文件中。

```

func (mw *GinJWTMiddleware) LoginHandler(c *gin.Context) {
    if mw.Authenticator == nil {
        mw.unauthorized(c, http.StatusInternalServerError, mw.HTTPStatusMessageFunc(ErrMissingAuthenticatorFunc,
            return
        }

    data, err := mw.Authenticator(c)

    if err != nil {
        mw.unauthorized(c, http.StatusUnauthorized, mw.HTTPStatusMessageFunc(err, c))
        return
    }

    // Create the token
    token := jwt.New(jwt.GetSigningMethod(mw.SigningAlgorithm))
    claims := token.Claims.(jwt.MapClaims)

    if mw.PayloadFunc != nil {
        for key, value := range mw.PayloadFunc(data) {
            claims[key] = value
        }
    }

    expire := mw.TimeFunc().Add(mw.Timeout)
    claims["exp"] = expire.Unix()
    claims["orig_iat"] = mw.TimeFunc().Unix()
    tokenString, err := mw.signedString(token)

    if err != nil {
        mw.unauthorized(c, http.StatusUnauthorized, mw.HTTPStatusMessageFunc(ErrFailedTokenCreation, c))
        return
    }

    // set cookie
    if mw.SendCookie {
        expireCookie := mw.TimeFunc().Add(mw.CookieMaxAge)
        maxage := int(expireCookie.Unix() - mw.TimeFunc().Unix())

        if mw.CookieSameSite != 0 {
            c.SetSameSite(mw.CookieSameSite)
        }

        c.SetCookie(
            mw.CookieName,
            tokenString,
            maxage,
            "/",
            mw.CookieDomain,
            mw.SecureCookie,
            mw.CookieHTTPOnly,
        )
    }

    mw.LoginResponse(c, http.StatusOK, tokenString, expire)
}

```

从LoginHandler函数的代码实现中，我们可以知道，LoginHandler函数会执行Authenticator函数，来完成Basic认证。如果认证通过，则会签发JWT Token，并执行PayloadFunc函数设置Token Payload。如果我们设置了SendCookie=true，还会在Cookie中添加认证相关的信息，例如Token、Token的生命周

期等，最后执行 LoginResponse 方法返回Token和Token的过期时间。

Authenticator、PayloadFunc、LoginResponse这三个函数，是我们在创建JWT认证策略时指定的。下面我来分别介绍下。

先来看下Authenticator函数。Authenticator函数从HTTP Authorization Header中获取用户名和密码，并校验密码是否合法。

```
func authenticator() func(c *gin.Context) (interface{}, error) {
    return func(c *gin.Context) (interface{}, error) {
        var login loginInfo
        var err error

        // support header and body both
        if c.Request.Header.Get("Authorization") != "" {
            login, err = parseWithHeader(c)
        } else {
            login, err = parseWithBody(c)
        }
        if err != nil {
            return "", jwt.ErrFailedAuthentication
        }

        // Get the user information by the login username.
        user, err := store.Client().Users().Get(c, login.Username, metav1.GetOptions{})
        if err != nil {
            log.Errorf("get user information failed: %s", err.Error())

            return "", jwt.ErrFailedAuthentication
        }

        // Compare the login password with the user password.
        if err := user.Compare(login.Password); err != nil {
            return "", jwt.ErrFailedAuthentication
        }

        return user, nil
    }
}
```

Authenticator函数需要获取用户名和密码。它首先会判断是否有Authorization请求头，如果有，则调用parseWithHeader函数获取用户名和密码，否则调用parseWithBody从Body中获取用户名和密码。如果都获取失败，则返回认证失败错误。

所以，IAM项目的Basic支持以下两种请求方式：

```
$ curl -XPOST -H"Authorization: Basic YWRtaW46QWRtaW5AMjAyMQ==" http://127.0.0.1:8080/login # 用户名:密码通过
$ curl -s -XPOST -H'Content-Type: application/json' -d'{"username":"admin","password":"Admin@2021"}' http://
```

这里，我们来看下 parseWithHeader 是如何获取用户名和密码的。假设我们的请求为：

```
$ curl -XPOST -H"Authorization: Basic YWRtaW46QWRtaW5AMjAyMQ==" http://127.0.0.1:8080/login
```

其中，YWRtaW46QWRtaW5AMjAyMQ==值由以下命令生成：

```
$ echo -n 'admin:Admin@2021'|base64  
YWRtaW46QWRtaW5AMjAyMQ==
```

parseWithHeader实际上执行的是上述命令的逆向步骤：

1. 获取Authorization头的值，并调用strings.SplitN函数，获取一个切片变量auth，其值为 ["Basic", "YWRtaW46QWRtaW5AMjAyMQ=="]。
2. 将YWRtaW46QWRtaW5AMjAyMQ==进行base64解码，得到admin:Admin@2021。
3. 调用strings.SplitN函数获取 admin:Admin@2021，得到用户名为admin，密码为Admin@2021。

parseWithBody则是调用了Gin的ShouldBindJSON函数，来从Body中解析出用户名和密码。

获取到用户名和密码之后，程序会从数据库中查询出该用户对应的加密后的密码，这里我们假设是xxxx。最后authenticator函数调用user.Compare来判断 xxxx 是否和通过user.Compare加密后的字符串相匹配，如果匹配则认证成功，否则返回认证失败。

再来看下PayloadFunc函数：

```
func payloadFunc() func(data interface{}) jwt.MapClaims {  
    return func(data interface{}) jwt.MapClaims {  
        claims := jwt.MapClaims{  
            "iss": APIServerIssuer,  
            "aud": APIServerAudience,  
        }  
        if u, ok := data.(*v1.User); ok {  
            claims[jwt.IdentityKey] = u.Name  
            claims["sub"] = u.Name  
        }  
        return claims  
    }  
}
```

PayloadFunc函数会设置JWT Token中Payload部分的 iss、aud、sub、identity字段，供后面使用。

再来看下我们刚才说的第三个函数，LoginResponse函数：

```
func loginResponse() func(c *gin.Context, code int, token string, expire time.Time) {
```

```

    return func(c *gin.Context, code int, token string, expire time.Time) {
        c.JSON(http.StatusOK, gin.H{
            "token": token,
            "expire": expire.Format(time.RFC3339),
        })
    }
}

```

该函数用来在Basic认证成功之后，返回Token和Token的过期时间给调用者：

```

$ curl -XPOST -H"Authorization: Basic YWRtaW46QWRtaW5AMjAyMQ==" http://127.0.0.1:8080/login
{"expire":"2021-09-29T01:38:49+08:00","token":"XX.YY.ZZ"}

```

登陆成功后，iam-apiserver会返回Token和Token的过期时间，前端可以将这些信息缓存在Cookie中或LocalStorage中，之后的请求都可以使用Token来进行认证。使用Token进行认证，不仅能够提高认证的安全性，还能够避免查询数据库，从而提高认证效率。

2. RefreshHandler

RefreshHandler函数会先执行Bearer认证，如果认证通过，则会重新签发Token。

3. LogoutHandler

最后，来看下LogoutHandler函数：

```

func (mw *GinJWTMiddleware) LogoutHandler(c *gin.Context) {
    // delete auth cookie
    if mw.SendCookie {
        if mw.CookieSameSite != 0 {
            c.SetSameSite(mw.CookieSameSite)
        }

        c.SetCookie(
            mw.CookieName,
            "",
            -1,
            "/",
            mw.CookieDomain,
            mw.SecureCookie,
            mw.CookieHTTPOnly,
        )
    }

    mw.LogoutResponse(c, http.StatusOK)
}

```

可以看到，LogoutHandler其实是用来清空Cookie中Bearer认证相关信息的。

最后，我们来做个总结：Basic认证通过用户名和密码来进行认证，通常用在登陆接口/login中。用户登陆成功后，会返回JWT Token，前端会保存该JWT Token在浏览器的Cookie或LocalStorage中，供后续请求使用。

后续请求时，均会携带该Token，以完成Bearer认证。另外，有了登陆接口，一般还会配套/logout接口和/refresh接口，分别用来进行注销和刷新Token。

这里你可能会问，为什么要刷新Token？因为通过登陆接口签发的Token有过期时间，有了刷新接口，前端就可以根据需要，自行刷新Token的过期时间。过期时间可以通过iam-apiserver配置文件的[jwt.timeout](#)配置项来指定。登陆后签发Token时，使用的密钥（secretKey）由[jwt.key](#)配置项来指定。

IAM项目是如何实现Bearer认证的？

上面我们介绍了Basic认证。这里，我再来介绍下IAM项目中Bearer认证的实现方式。

IAM项目中有两个地方实现了Bearer认证，分别是 iam-apiserver 和 iam-authz-server。下面我来分别介绍下它们是如何实现Bearer认证的。

iam-authz-server Bearer认证实现

先来看下iam-authz-server是如何实现Bearer认证的。

iam-authz-server通过在 /v1 路由分组中加载cache认证中间件来使用cache认证策略：

```
auth := newCacheAuth()
apiv1 := g.Group("/v1", auth.AuthFunc())
```

来看下[newCacheAuth](#)函数：

```
func newCacheAuth() middleware.AuthStrategy {
    return auth.NewCacheStrategy(getSecretFunc())
}

func getSecretFunc() func(string) (auth.Secret, error) {
    return func(kid string) (auth.Secret, error) {
        cli, err := store.GetStoreInsOr(nil)
        if err != nil {
            return auth.Secret{}, errors.Wrap(err, "get store instance failed")
        }

        secret, err := cli.GetSecret(kid)
        if err != nil {
            return auth.Secret{}, err
        }

        return auth.Secret{
            Username: secret.Username,
            ID:       secret.SecretId,
            Key:      secret.SecretKey,
            Expires:  secret.Expires,
        }, nil
    }
}
```



```

    }, nil
}
}

```

newCacheAuth函数调用auth.NewCacheStrategy创建了一个cache认证策略，创建时传入了getSecretFunc函数，该函数会返回密钥的信息。密钥信息包含了以下字段：

```

type Secret struct {
    Username string
    ID        string
    Key       string
    Expires   int64
}

```

再来看下cache认证策略实现的AuthFunc方法：

```

func (cache CacheStrategy) AuthFunc() gin.HandlerFunc {
    return func(c *gin.Context) {
        header := c.Request.Header.Get("Authorization")
        if len(header) == 0 {
            core.WriteResponse(c, errors.WithCode(code.ErrMissingHeader, "Authorization header cannot be empty."), nil)
            c.Abort()

            return
        }

        var rawJWT string
        // Parse the header to get the token part.
        fmt.Sscanf(header, "Bearer %s", &rawJWT)

        // Use own validation logic, see below
        var secret Secret

        claims := &jwt.MapClaims{}
        // Verify the token
        parsedT, err := jwt.ParseWithClaims(rawJWT, claims, func(token *jwt.Token) (interface{}, error) {
            // Validate the alg is HMAC signature
            if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
                return nil, fmt.Errorf("unexpected signing method: %v", token.Header["alg"])
            }

            kid, ok := token.Header["kid"].(string)
            if !ok {
                return nil, ErrMissingKID
            }

            var err error
            secret, err = cache.get(kid)
            if err != nil {
                return nil, ErrMissingSecret
            }

            return []byte(secret.Key), nil
        }, jwt.WithAudience(AuthzAudience))
    }
}

```

```

if err != nil || !parsedT.Valid {
    core.WriteResponse(c, errors.WithCode(code.ErrSignatureInvalid, err.Error()), nil)
    c.Abort()

    return
}

if KeyExpired(secret.Expires) {
    tm := time.Unix(secret.Expires, 0).Format("2006-01-02 15:04:05")
    core.WriteResponse(c, errors.WithCode(code.ErrExpired, "expired at: %s", tm), nil)
    c.Abort()

    return
}

c.Set(CtxUsername, secret.Username)
c.Next()
}
}

// KeyExpired checks if a key has expired, if the value of user.SessionState.Expires is 0, it will be ignor
func KeyExpired(expires int64) bool {
    if expires >= 1 {
        return time.Now().After(time.Unix(expires, 0))
    }

    return false
}

```

AuthFunc函数依次执行了以下四大步来完成JWT认证，每一步中又有一些小步骤，下面我们来一起看看。

第一步，从Authorization: Bearer XX.YY.ZZ请求头中获取XX.YY.ZZ，XX.YY.ZZ即为JWT Token。

第二步，调用github.com/dgrijalva/jwt-go包提供的ParseWithClaims函数，该函数会依次执行下面四步操作。

调用ParseUnverified函数，依次执行以下操作：

从Token中获取第一段XX，base64解码后得到JWT Token的

Header{ “alg” : “HS256” , “kid” : “a45yPqUnQ8gljH43jAGQdRo0bXzNLjlU0hxa” , “typ” : “JWT” }。

从Token中获取第二段YY，base64解码后得到JWT Token的

Payload{ “aud” : “iam.authz.marmotedu.com” , “exp” :1625104314, “iat” :1625097114, “iss” : “iamctl” , “nbf” :1625097114}。

根据Token Header中的alg字段，获取Token加密函数。

最终ParseUnverified函数会返回Token类型的变量，Token类型包含 Method、Header、Claims、Valid这些重要字段，这些字段会用于后续的认证步骤中。

调用传入的keyFunc获取密钥，这里来看下keyFunc的实现：

```

func(token *jwt.Token) (interface{}, error) {
    // Validate the alg is HMAC signature
    if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
        return nil, fmt.Errorf("unexpected signing method: %v", token.Header["alg"])
    }

    kid, ok := token.Header["kid"].(string)
    if !ok {
        return nil, ErrMissingKID
    }

    var err error
    secret, err = cache.get(kid)
    if err != nil {
        return nil, ErrMissingSecret
    }

    return []byte(secret.Key), nil
}

```

可以看到，keyFunc接受 *Token 类型的变量，并获取Token Header中的kid，kid即为密钥ID：secretID。接着，调用cache.get(kid)获取密钥secretKey。cache.get函数即为getSecretFunc，getSecretFunc函数会根据kid，从内存中查找密钥信息，密钥信息中包含了secretKey。

3. 从Token中获取Signature签名字符串ZZZ，也即Token的第三段。
4. 获取到secretKey之后，token.Method.Verify验证Signature签名字符串ZZZ，也即Token的第三段是否合法。token.Method.Verify实际上是使用了相同的加密算法和相同的secretKey加密XX.YY字符串。假设加密之后的字符串为WW，接下来会用WW和ZZ base64解码后的字符串进行比较，如果相等则认证通过，如果不相等则认证失败。

第三步，调用KeyExpired，验证secret是否过期。secret信息中包含过期时间，你只需要拿该过期时间和当前时间对比就行。

第四步，设置HTTP Headerusername: colin。

到这里，iam-authz-server的Bearer认证分析就完成了。

我们来做个总结：iam-authz-server通过加载Gin中间件的方式，在请求/v1/authz接口时进行访问认证。因为Bearer认证具有过期时间，而且可以在认证字符串中携带更多有用信息，还具有不可逆加密等优点，所以/v1/authz采用了Bearer认证，Token格式采用了JWT格式，这也是业界在API认证中最受欢迎的认证方式。

Bearer认证需要secretID和secretKey，这些信息会通过gRPC接口调用，从iam-apiserver中获取，并缓存在iam-authz-server的内存中供认证时查询使用。

当请求来临时，iam-authz-server Bearer认证中间件从JWT Token中解析出Header，并从Header的kid字段中获取到secretID，根据secretID查找到secretKey，最后使用secretKey加密JWT Token的Header和Payload，并与Signature部分进行对比。如果相等，则认证通过；如果不等，则认证失败。

iam-apiserver Bearer认证实现

再来看下 iam-apiserver的Bearer认证。

iam-apiserver的Bearer认证通过以下代码（位于[router.go](#)文件中）指定使用了auto认证策略：

```
v1.Use(auto.AuthFunc())
```

我们来看下[auto.AuthFunc\(\)](#)的实现：

```
func (a AutoStrategy) AuthFunc() gin.HandlerFunc {
    return func(c *gin.Context) {
        operator := middleware.AuthOperator{}
        authHeader := strings.SplitN(c.Request.Header.Get("Authorization"), " ", 2)

        if len(authHeader) != authHeaderCount {
            core.WriteResponse(
                c,
                errors.WithCode(code.ErrInvalidAuthHeader, "Authorization header format is wrong."),
                nil,
            )
            c.Abort()

            return
        }

        switch authHeader[0] {
        case "Basic":
            operator.SetStrategy(a.basic)
        case "Bearer":
            operator.SetStrategy(a.jwt)
            // a.JWT.MiddlewareFunc()(c)
        default:
            core.WriteResponse(c, errors.WithCode(code.ErrSignatureInvalid, "unrecognized Authorization header."), nil)
            c.Abort()

            return
        }

        operator.AuthFunc()(c)

        c.Next()
    }
}
```

从上面代码中可以看到，AuthFunc函数会从Authorization Header中解析出认证方式是Basic还是Bearer。如果是Bearer，就会使用JWT认证策略；如果是Basic，就会使用Basic认证策略。

我们再来看下JWT认证策略的[AuthFunc](#)函数实现：

```
func (j JWTStrategy) AuthFunc() gin.HandlerFunc {  
    return j.MiddlewareFunc()  
}
```

我们跟随代码，可以定位到MiddlewareFunc函数最终调用了github.com/appleboy/gin-jwt包GinJWTMiddleware结构体的[middlewareImpl](#)方法：

```
func (mw *GinJWTMiddleware) middlewareImpl(c *gin.Context) {  
    claims, err := mw.GetClaimsFromJWT(c)  
    if err != nil {  
        mw.unauthorized(c, http.StatusUnauthorized, mw.HTTPStatusMessageFunc(err, c))  
        return  
    }  
  
    if claims["exp"] == nil {  
        mw.unauthorized(c, http.StatusBadRequest, mw.HTTPStatusMessageFunc(ErrMissingExpField, c))  
        return  
    }  
  
    if _, ok := claims["exp"].(float64); !ok {  
        mw.unauthorized(c, http.StatusBadRequest, mw.HTTPStatusMessageFunc(ErrWrongFormatOfExp, c))  
        return  
    }  
  
    if int64(claims["exp"].(float64)) < mw.TimeFunc().Unix() {  
        mw.unauthorized(c, http.StatusUnauthorized, mw.HTTPStatusMessageFunc(ErrExpiredToken, c))  
        return  
    }  
  
    c.Set("JWT_PAYLOAD", claims)  
    identity := mw.IdentityHandler(c)  
  
    if identity != nil {  
        c.Set(mw.IdentityKey, identity)  
    }  
  
    if !mw.Authorizator(identity, c) {  
        mw.unauthorized(c, http.StatusForbidden, mw.HTTPStatusMessageFunc(ErrForbidden, c))  
        return  
    }  
  
    c.Next()  
}
```

分析上面的代码，我们可以知道，middlewareImpl的Bearer认证流程为：

第一步：调用GetClaimsFromJWT函数，从HTTP请求中获取Authorization Header，并解析出Token字符串，进行认证，最后返回Token Payload。

第二步：校验Payload中的exp是否超过当前时间，如果超过就说明Token过期，校验不通过。

第三步：给gin.Context中添加JWT_PAYLOAD键，供后续程序使用（当然也可能用不到）。

第四步：通过以下代码，在gin.Context中添加IdentityKey键，IdentityKey键可以在创建GinJWTMiddleware结构体时指定，这里我们设置为middleware.UsernameKey，也就是username。

```
identity := mw.IdentityHandler(c)

if identity != nil {
    c.Set(mw.IdentityKey, identity)
}
```

IdentityKey键的值由IdentityHandler函数返回，IdentityHandler函数为：

```
func(c *gin.Context) interface{} {
    claims := jwt.ExtractClaims(c)

    return claims[jwt.IdentityKey]
}
```

上述函数会从Token的Payload中获取identity域的值，identity域的值是在签发Token时指定的，它的值其实是用户名，你可以查看[payloadFunc](#)函数了解。

第五步：接下来，会调用Authorizator方法，Authorizator是一个callback函数，成功时必须返回真，失败时必须返回假。Authorizator也是在创建GinJWTMiddleware时指定的，例如：

```
func authorizator() func(data interface{}, c *gin.Context) bool {
    return func(data interface{}, c *gin.Context) bool {
        // add username to header
        if v, ok := data.(string); ok {
            // c.Request.Header.Add(log.KeyUsername, v)
            c.Set(CtxUsername, v)

            return true
        }

        return false
    }
}
```

authorizator函数返回了一个匿名函数，匿名函数在认证成功后，会打印一条认证成功日志。

IAM项目认证功能设计技巧

我在设计IAM项目的认证功能时，也运用了一些技巧，这里分享给你。

技巧1：面向接口编程

在使用[NewAutoStrategy](#)函数创建auto认证策略时，传入了BasicStrategy、JWTStrategy接口类型的参数，这意味着Basic认证和Bearer认证都可以有不同的实现，这样后期可以根据需要扩展新的认证方式。

技巧2：使用抽象工厂模式

[auth.go](#)文件中，通过newBasicAuth、newJWTAuth、newAutoAuth创建认证策略时，返回的都是接口。通过返回接口，可以在不公开内部实现的情况下，让调用者使用你提供的各种认证功能。

技巧3：使用策略模式

在auto认证策略中，我们会根据HTTP 请求头Authorization: XXX X.Y.X中的XXX来选择并设置认证策略（Basic 或 Bearer）。具体可以查看AutoStrategy的[AuthFunc](#)函数：

```
func (a AutoStrategy) AuthFunc() gin.HandlerFunc {
    return func(c *gin.Context) {
        operator := middleware.AuthOperator{}
        authHeader := strings.SplitN(c.Request.Header.Get("Authorization"), " ", 2)
        ...
        switch authHeader[0] {
            case "Basic":
                operator.SetStrategy(a.basic)
            case "Bearer":
                operator.SetStrategy(a.jwt)
                // a.JWT.MiddlewareFunc()(c)
            default:
                core.WriteResponse(c, errors.WithCode(code.ErrSignatureInvalid, "unrecognized Authorization header."), n
                c.Abort()

                return
            }

            operator.AuthFunc()(c)

            c.Next()
        }
    }
}
```

上述代码中，如果是Basic，则设置为Basic认证方法operator.SetStrategy(a.basic)；如果是Bearer，则设置为Bearer认证方法operator.SetStrategy(a.jwt)。SetStrategy方法的入参是AuthStrategy类型的接口，都实现了AuthFunc() gin.HandlerFunc函数，用来进行认证，所以最后我们调用operator.AuthFunc()(c)即可完成认证。

总结

在IAM项目中，iam-apiserver实现了Basic认证和Bearer认证，iam-authz-server实现了Bearer认证。这一讲重点介绍了iam-apiserver的认证实现。

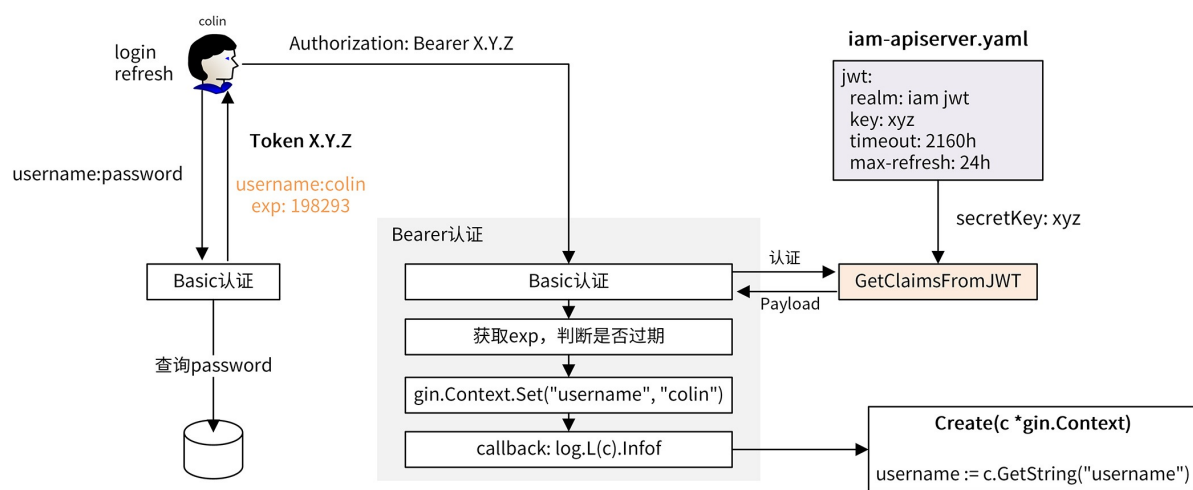
用户要访问iam-apiserver，首先需要通过Basic认证，认证通过之后，会返回JWT Token和JWT Token的过期时间。前端将Token缓存在LocalStorage或Cookie中，后续的请求都通过Token来认证。

执行Basic认证时，iam-apiserver会从HTTP Authorization Header中解析出用户名和密码，将密码再加

密，并和数据库中保存的值进行对比。如果不匹配，则认证失败，否则认证成功。认证成功之后，会返回Token，并在Token的Payload部分设置用户名，Key为 username。

执行Bearer认证时，iam-apiserver会从JWT Token中解析出Header和Payload，并从Header中获取加密算法。接着，用获取到的加密算法和从配置文件中获取到的密钥对Header.Payload进行再加密，得到Signature，并对比两次的Signature是否相等。如果不相等，则返回 HTTP 401 Unauthorized 错误；如果相等，接下来会判断Token是否过期，如果过期则返回认证不通过，否则认证通过。认证通过之后，会将Payload中的username添加到gin.Context类型的变量中，供后面的业务逻辑使用。

我绘制了整个流程的示意图，你可以对照着再回顾一遍。



课后练习

1. 走读github.com/appleboy/gin-jwt包的GinJWTMiddleware结构体的[GetClaimsFromJWT](#)方法，分析一下：GetClaimsFromJWT方法是如何从gin.Context中解析出Token，并进行认证的？
2. 思考下，iam-apiserver和iam-authzserver是否可以使用同一个认证策略？如果可以，又该如何实现？

欢迎你在留言区与我交流讨论，我们下一讲见。