



下载APP



26 | IAM项目是如何设计和实现访问认证功能的？

2021-07-24 孔令飞

《Go 语言项目开发实战》

课程介绍 >



讲述：孔令飞

时长 21:25 大小 19.63M



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：Basic、Digest、OAuth、Bearer。这一讲，我们再来看下 IAM 项目是如何设计和实现认证功能的。

IAM 项目用到了 Basic 认证和 Bearer 认证。其中，Basic 认证用在前端登陆的场景，Bearer 认证用在调用后端 API 服务的场景下。

接下来，我们先来看下 IAM 项目认证功能的整体设计思路。



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。

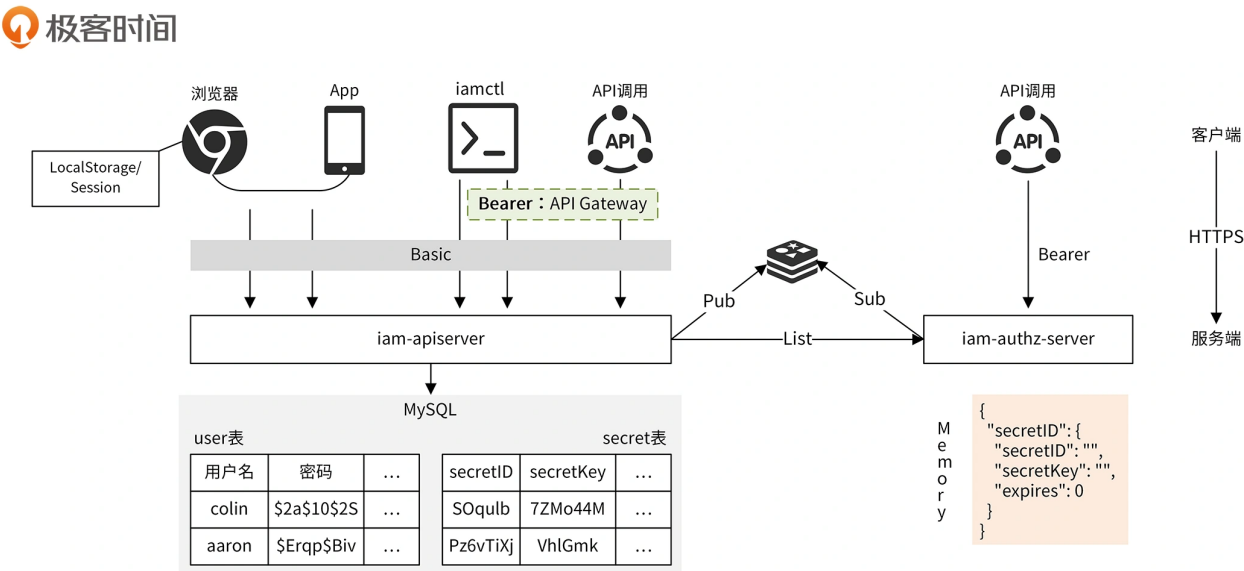
当然，/v1/authz也可以直接注册到 API 网关中。在实际的 Go 项目开发中，也是我推荐的一种方式。但在这里，为了展示实现 Bearer 认证的过程，iam-authz-server 自己实现了 Bearer 认证。讲到 iam-authz-server Bearer 认证实现的时候，我会详细介绍这一点。

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：

[复制代码](#)

```
1 jwtStrategy, _ := newJWTAuth().(auth.JWTStrategy)
2 g.POST("/login", jwtStrategy.LoginHandler)
3 g.POST("/logout", jwtStrategy.LogoutHandler)
4 // Refresh time can be longer than token timeout
5 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。

[复制代码](#)

```
1 func (mw *GinJWTMiddleware) LoginHandler(c *gin.Context) {
2     if mw.Authenticator == nil {
3         mw.unauthorized(c, http.StatusInternalServerError, mw.HTTPStatusMessageFunc(
4             return
5         })
6     }
7     data, err := mw.Authenticator(c)
8
9     if err != nil {
10         mw.unauthorized(c, http.StatusUnauthorized, mw.HTTPStatusMessageFunc(err,
11             return
12         })
13 }
```

```
14 // Create the token
15 token := jwt.New(jwt.GetSigningMethod(mw.SigningAlgorithm))
16 claims := token.Claims.(jwt.MapClaims)
17
18 if mw.PayloadFunc != nil {
19     for key, value := range mw.PayloadFunc(data) {
20         claims[key] = value
21     }
22 }
23
24 expire := mw.TimeFunc().Add(mw.Timeout)
25 claims["exp"] = expire.Unix()
26 claims["orig_iat"] = mw.TimeFunc().Unix()
27 tokenString, err := mw.signedString(token)
28
29 if err != nil {
30     mw.unauthorized(c, http.StatusUnauthorized, mw.HTTPStatusMessageFunc(ErrFa
31     return
32 }
33
34 // set cookie
35 if mw.SendCookie {
36     expireCookie := mw.TimeFunc().Add(mw.CookieMaxAge)
37     maxage := int(expireCookie.Unix() - mw.TimeFunc().Unix())
38
39     if mw.CookieSameSite != 0 {
40         c.SetSameSite(mw.CookieSameSite)
41     }
42
43     c.SetCookie(
44         mw.CookieName,
45         tokenString,
46         maxage,
47         "/",
48         mw.CookieDomain,
49         mw.SecureCookie,
50         mw.CookieHTTPOnly,
51     )
52 }
53
54 mw.LoginResponse(c, http.StatusOK, tokenString, expire)
55 }
```

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。

[复制代码](#)

```
1 func authenticator() func(c *gin.Context) (interface{}, error) {
2     return func(c *gin.Context) (interface{}, error) {
3         var login loginInfo
4         var err error
5
6         // support header and body both
7         if c.Request.Header.Get("Authorization") != "" {
8             login, err = parseWithHeader(c)
9         } else {
10            login, err = parseWithBody(c)
11        }
12        if err != nil {
13            return "", jwt.ErrFailedAuthentication
14        }
15
16        // Get the user information by the login username.
17        user, err := store.Client().Users().Get(c, login.Username, metav1.GetOptio
18        if err != nil {
19            log.Errorf("get user information failed: %s", err.Error())
20
21            return "", jwt.ErrFailedAuthentication
22        }
23
24        // Compare the login password with the user password.
25        if err := user.Compare(login.Password); err != nil {
26            return "", jwt.ErrFailedAuthentication
27        }
28
29        return user, nil
30    }
31 }
```


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：

 复制代码


```
1 $ curl -XPOST -H"Authorization: Basic YWRtaW46QWRtaW5AMjAyMQ==" http://127.0.0
2 $ curl -s -XPOST -H'Content-Type: application/json' -d'{"username":"admin","pa
```

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：

 复制代码

```
1 $ curl -XPOST -H"Authorization: Basic YWRtaW46QWRtaW5AMjAyMQ==" http://127.0.0
```

其中，`YWRtaW46QWRtaW5AMjAyMQ==`值由以下命令生成：

 复制代码

```
1 $ echo -n 'admin:Admin@2021'|base64
2 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。


再来看一下PayloadFunc函数：

 复制代码

```
1 func payloadFunc() func(data interface{}) jwt.MapClaims {
2     return func(data interface{}) jwt.MapClaims {
3         claims := jwt.MapClaims{
4             "iss": APIServerIssuer,
5             "aud": APIServerAudience,
6         }
7         if u, ok := data.(*v1.User); ok {
8             claims[jwt.IdentityKey] = u.Name
9             claims["sub"] = u.Name
10        }
11
12        return claims
13    }
14 }
```

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

再来看一下我们刚才说的第三个函数，LoginResponse 函数：

 复制代码

```
1 func loginResponse() func(c *gin.Context, code int, token string, expire time.
2     return func(c *gin.Context, code int, token string, expire time.Time) {
3         c.JSON(http.StatusOK, gin.H{
4             "token": token,
5             "expire": expire.Format(time.RFC3339),
6         })
7     }
8 }
```

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：

 复制代码

```
1 $ curl -XPOST -H"Authorization: Basic YWRtaW46QWRtaW5AMjAyMQ==" http://127.0.0
2 {"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：

 复制代码

```
1 func (mw *GinJWTMiddleware) LogoutHandler(c *gin.Context) {
2     // delete auth cookie
3     if mw.SendCookie {
4         if mw.CookieSameSite != 0 {
5             c.SetSameSite(mw.CookieSameSite)
6         }
7
8         c.SetCookie(
9             mw.CookieName,
10            "",
11            -1,
12            "/",
13            mw.CookieDomain,
14            mw.SecureCookie,
15            mw.CookieHTTPOnly,
16        )
17    }
18
19    mw.LogoutResponse(c, http.StatusOK)
20 }
```

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：

[复制代码](#)

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

来看下 `newCacheAuth` 函数：

[复制代码](#)

```
1 func newCacheAuth() middleware.AuthStrategy {
2     return auth.NewCacheStrategy(getSecretFunc())
3 }
4
5 func getSecretFunc() func(string) (auth.Secret, error) {
6     return func(kid string) (auth.Secret, error) {
7         cli, err := store.GetStoreInsOr(nil)
8         if err != nil {
9             return auth.Secret{}, errors.Wrap(err, "get store instance failed")
10        }
```

```

10     }
11
12     secret, err := cli.GetSecret(kid)
13     if err != nil {
14         return auth.Secret{}, err
15     }
16
17     return auth.Secret{
18         Username: secret.Username,
19         ID:       secret.SecretId,
20         Key:      secret.SecretKey,
21         Expires:  secret.Expires,
22     }, nil
23 }
24 }

```

`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：

[复制代码](#)

```

1 type Secret struct {
2     Username string
3     ID       string
4     Key      string
5     Expires  int64
6 }

```

再来看下 cache 认证策略实现的 `AuthFunc` 方法：

[复制代码](#)

```

1 func (cache CacheStrategy) AuthFunc() gin.HandlerFunc {
2     return func(c *gin.Context) {
3         header := c.Request.Header.Get("Authorization")
4         if len(header) == 0 {
5             core.WriteResponse(c, errors.WithCode(code.ErrMissingHeader, "Authorizat
6             c.Abort()
7
8             return
9         }
10
11         var rawJWT string
12         // Parse the header to get the token part.
13         fmt.Sscanf(header, "Bearer %s", &rawJWT)
14
15         // Use own validation logic, see below

```

```
16     var secret Secret
17
18     claims := &jwt.MapClaims{}
19     // Verify the token
20     parsedT, err := jwt.ParseWithClaims(rawJWT, claims, func(token *jwt.Token)
21         // Validate the alg is HMAC signature
22         if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
23             return nil, fmt.Errorf("unexpected signing method: %v", token.Header["
24         }
25
26         kid, ok := token.Header["kid"].(string)
27         if !ok {
28             return nil, ErrMissingKID
29         }
30
31         var err error
32         secret, err = cache.get(kid)
33         if err != nil {
34             return nil, ErrMissingSecret
35         }
36
37         return []byte(secret.Key), nil
38     }, jwt.WithAudience(AuthzAudience))
39     if err != nil || !parsedT.Valid {
40         core.WriteResponse(c, errors.WithCode(code.ErrSignatureInvalid, err.Erro
41         c.Abort()
42
43         return
44     }
45
46     if KeyExpired(secret.Expires) {
47         tm := time.Unix(secret.Expires, 0).Format("2006-01-02 15:04:05")
48         core.WriteResponse(c, errors.WithCode(code.ErrExpired, "expired at: %s",
49         c.Abort()
50
51         return
52     }
53
54     c.Set(CtxUsername, secret.Username)
55     c.Next()
56 }
57 }
58
59 // KeyExpired checks if a key has expired, if the value of user.SessionState.E
60 func KeyExpired(expires int64) bool {
61     if expires >= 1 {
62         return time.Now().After(time.Unix(expires, 0))
63     }
64
65     return false
66 }
```

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，下面我们一起来看看。

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" : "a45yPqUnQ8gljH43jAGQdRo0bXzNLjIU0hxa" , "typ" : "JWT" }。

从 Token 中获取第一段 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认证步骤中。

调用传入的 keyFunc 获取密钥，这里来看下 keyFunc 的实现：

 复制代码

```
1 func(token *jwt.Token) (interface{}, error) {
2     // Validate the alg is HMAC signature
3     if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
4         return nil, fmt.Errorf("unexpected signing method: %v", token.Header["alg"])
5     }
6
7     kid, ok := token.Header["kid"].(string)
8     if !ok {
9         return nil, ErrMissingKID
10    }
11 }
```

```
12     var err error
13     secret, err = cache.get(kid)
14     if err != nil {
15         return nil, ErrMissingSecret
16     }
17
18     return []byte(secret.Key), nil
19 }
```

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 Header username: 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-apisaerver 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：

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

[复制代码](#)

我们来看下 [auto.AuthFunc\(\)](#) 的实现：

```
1 func (a AutoStrategy) AuthFunc() gin.HandlerFunc {
2     return func(c *gin.Context) {
3         operator := middleware.AuthOperator{}
4         authHeader := strings.SplitN(c.Request.Header.Get("Authorization"), " ", 2)
5
6         if len(authHeader) != authHeaderCount {
7             core.WriteResponse(
8                 c,
9                 errors.WithCode(code.ErrInvalidAuthHeader, "Authorization header forma
10                 nil,
11             )
12             c.Abort()
13
14             return
15         }
16
17         switch authHeader[0] {
18             case "Basic":
19                 operator.SetStrategy(a.basic)
20             case "Bearer":
21                 operator.SetStrategy(a.jwt)
22                 // a.JWT.MiddlewareFunc()(c)
23             default:
24                 core.WriteResponse(c, errors.WithCode(code.ErrSignatureInvalid, "unrecog
```

[复制代码](#)

```
25         c.Abort()
26
27         return
28     }
29
30     operator.AuthFunc()(c)
31
32     c.Next()
33 }
34 }
```

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：

[复制代码](#)

```
1 func (j JWTStrategy) AuthFunc() gin.HandlerFunc {
2     return j.MiddlewareFunc()
3 }
```

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：

[复制代码](#)

```
1 func (mw *GinJWTMiddleware) middlewareImpl(c *gin.Context) {
2     claims, err := mw.GetClaimsFromJWT(c)
3     if err != nil {
4         mw.unauthorized(c, http.StatusUnauthorized, mw.HTTPStatusMessageFunc(err,
5             return
6         }
7
8     if claims["exp"] == nil {
9         mw.unauthorized(c, http.StatusBadRequest, mw.HTTPStatusMessageFunc(ErrMiss
10         return
11     }
12
13     if _, ok := claims["exp"].(float64); !ok {
14         mw.unauthorized(c, http.StatusBadRequest, mw.HTTPStatusMessageFunc(ErrWron
15         return
16     }
```

```
17
18     if int64(claims["exp"].(float64)) < mw.TimeFunc().Unix() {
19         mw.unauthorized(c, http.StatusUnauthorized, mw.HTTPStatusMessageFunc(ErrEx
20             return
21     }
22
23     c.Set("JWT_PAYLOAD", claims)
24     identity := mw.IdentityHandler(c)
25
26     if identity != nil {
27         c.Set(mw.IdentityKey, identity)
28     }
29
30     if !mw.Authorizator(identity, c) {
31         mw.unauthorized(c, http.StatusForbidden, mw.HTTPStatusMessageFunc(ErrForbi
32             return
33     }
34
35     c.Next()
36 }
```


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。

 复制代码

```
1 identity := mw.IdentityHandler(c)
2
3 if identity != nil {
4     c.Set(mw.IdentityKey, identity)
5 }
```

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

[复制代码](#)

```
1 func(c *gin.Context) interface{} {  
2     claims := jwt.ExtractClaims(c)  
3  
4     return claims[jwt.IdentityKey]  
5 }
```

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：

[复制代码](#)

```
1 func authorizator() func(data interface{}, c *gin.Context) bool {  
2     return func(data interface{}, c *gin.Context) bool {  
3         // add username to header  
4         if v, ok := data.(string); ok {  
5             // c.Request.Header.Add(log.KeyUsername, v)  
6             c.Set(CtxUsername, v)  
7  
8             return true  
9         }  
10  
11         return false  
12     }  
13 }
```

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：

[复制代码](#)

```
1 func (a AutoStrategy) AuthFunc() gin.HandlerFunc {
2     return func(c *gin.Context) {
3         operator := middleware.AuthOperator{}
4         authHeader := strings.SplitN(c.Request.Header.Get("Authorization"), " ", 2
5             ...
6         switch authHeader[0] {
7         case "Basic":
8             operator.SetStrategy(a.basic)
9         case "Bearer":
10            operator.SetStrategy(a.jwt)
11            // a.JWT.MiddlewareFunc()(c)
12        default:
13            core.WriteResponse(c, errors.WithCode(code.ErrSignatureInvalid, "unrecog
14            c.Abort()
15
16        return
17    }
18
19    operator.AuthFunc()(c)
20
21    c.Next()
22 }
23 }
```

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

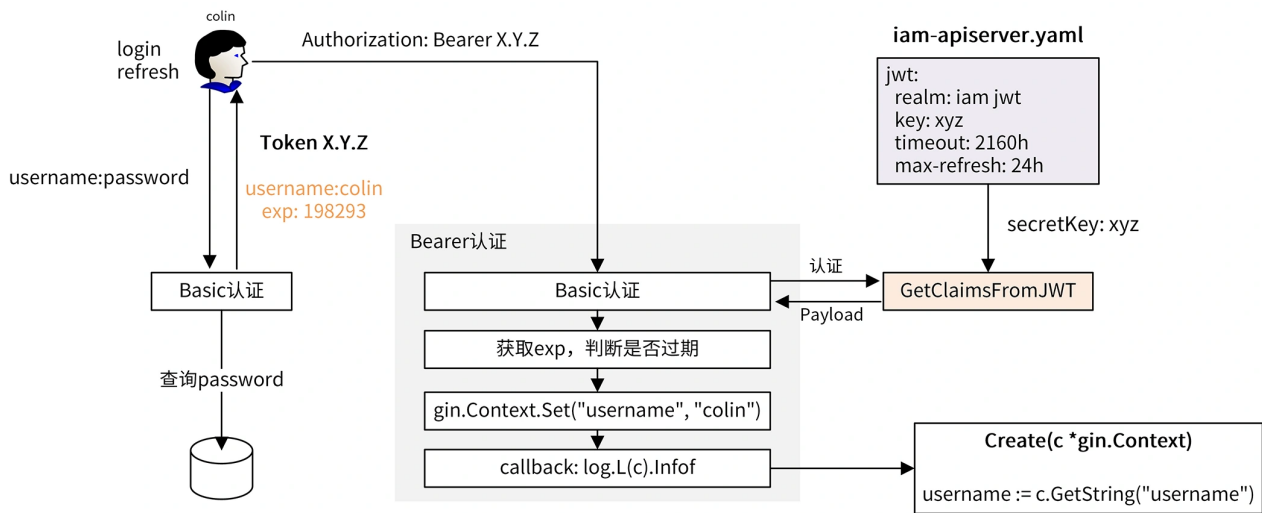
在 IAM 项目中，iam-apiserver 实现了 Basic 认证和 Bearer 认证，iam-authz-server 实现了 Bearer 认证。这一讲重点介绍了 iam-apiserver 的认证实现。

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。

分享给需要的人，Ta订阅后你可得 24 元现金奖励

👍 赞 1 💡 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

- 上一篇 25 | 认证机制：应用程序如何进行访问认证？
- 下一篇 27 | 权限模型：5大权限模型是如何进行资源授权的？

更多课程推荐

说透区块链

拨开迷雾，还原区块链真相

赵铭

区块链服务平台资深架构师



新版升级：点击「👤请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

精选留言 (1)

写留言



helloworld

2021-07-26

本文的意思是说正常的生产环境下，iam-apiserver和iam-authz-server的api的认证功能其实都应该放到网关来实现的，本文之所以由iam项目亲自来实现就是为了方便讲解认证的具体实现方法，我理解的对不对？

作者回复: 老哥理解的没毛病

