

## 31-数据流：通过iam-authz-server设计，看数据流服务的设计

你好，我是孔令飞。

在 [28讲](#) 和 [29讲](#)，我介绍了IAM的控制流服务iam-apiserver的设计和实现。这一讲，我们再来看下IAM数据流服务iam-authz-server的设计和实现。

因为iam-authz-server是数据流服务，对性能要求较高，所以采用了一些机制来最大化API接口的性能。另外，为了提高开发效率，避免重复造轮子，iam-authz-server和iam-apiserver共享了大部分的功能代码。接下来，我们就来看下，iam-authz-server是如何跟iam-apiserver共享代码的，以及iam-authz-server是如何保证API接口性能的。

### iam-authz-server的功能介绍

iam-authz-server目前的唯一功能，是通过提供 /v1/authz RESTful API接口完成资源授权。 /v1/authz 接口是通过[github.com/ory/ladon](https://github.com/ory/ladon)来完成资源授权的。

因为iam-authz-server承载了数据流的请求，需要确保API接口具有较高的性能。为了保证API接口的性能，iam-authz-server在设计上使用了大量的缓存技术。

### github.com/ory/ladon包介绍

因为iam-authz-server资源授权是通过 `github.com/ory/ladon` 来完成的，为了让你更好地理解iam-authz-server的授权策略，在这里我先介绍下 `github.com/ory/ladon` 包。

Ladon是用Go语言编写的用于实现访问控制策略的库，类似于RBAC（基于角色的访问控制系统，Role Based Access Control）和ACL（访问控制列表，Access Control Lists）。但是与RBAC和ACL相比，Ladon可以实现更细粒度的访问控制，并且能够在更为复杂的环境中（例如多租户、分布式应用程序和大型组织）工作。

Ladon解决了这个问题：在特定的条件下，谁能够/不能够对哪些资源做哪些操作。为了解决这个问题，Ladon引入了授权策略。授权策略是一个有语法规则的文档，这个文档描述了谁在什么条件下能够对哪些资源做哪些操作。Ladon可以用请求的上下文，去匹配设置的授权策略，最终判断出当前授权请求是否通过。下面是一个Ladon的授权策略样例：

```
{
  "description": "One policy to rule them all.",
  "subjects": ["users:<peter|ken>", "users:maria", "groups:admins"],
  "actions" : ["delete", "<create|update>"],
  "effect": "allow",
  "resources": [
    "resources:articles:<.*>",
    "resources:printer"
  ],
  "conditions": {
    "remoteIP": {
      "type": "CIDRCondition",
      "options": {
        "cidr": "192.168.0.1/16"
      }
    }
  }
}
```

```
}  
}
```

策略（Policy）由若干元素构成，用来描述授权的具体信息，你可以把它们看成一组规则。核心元素包括主题（Subject）、操作（Action）、效力（Effect）、资源（Resource）以及生效条件（Condition）。元素保留字仅支持小写，它们在描述上没有顺序要求。对于没有特定约束条件的策略，Condition元素是可选项。一条策略包含下面6个元素：

- 主题（Subject），主题名是唯一的，代表一个授权主题。例如，“ken” or “printer-service.mydomain.com”。
- 操作（Action），描述允许或拒绝的操作。
- 效力（Effect），描述策略产生的结果是“允许”还是“拒绝”，包括 allow（允许）和 deny（拒绝）。
- 资源（Resource），描述授权的具体数据。
- 生效条件（Condition），描述策略生效的约束条件。
- 描述（Description），策略的描述。

有了授权策略，我们就可以传入请求上下文，由Ladon来决定请求是否能通过授权。下面是一个请求示例：

```
{  
  "subject": "users:peter",  
  "action" : "delete",  
  "resource": "resources:articles:ladon-introduction",  
  "context": {  
    "remoteIP": "192.168.0.5"  
  }  
}
```

可以看到，在 remoteIP="192.168.0.5" 生效条件（Condition）下，针对主题（Subject） users:peter 对资源（Resource） resources:articles:ladon-introduction 的 delete 操作（Action），授权策略的效力（Effect）是 allow 的。所以Ladon会返回如下结果：

```
{  
  "allowed": true  
}
```

Ladon支持很多Condition，具体见下表：

Condition	描述
CIDRCondition	检查传入的key值是否匹配condition所设定的CIDR, key值是一个remote IP
StringEqualCondition	检查传入的key值是否是字符串类型, 并且等于condition所设定的值
BooleanCondition	检查传入的key值是否是bool类型, 并且等于condition所设定的值
StringMatchCondition	检查传入的key值是否匹配condition指定的正则规则
EqualsSubjectCondition	检查传入的key值是否匹配subject
StringPairsEqualCondition	检查传入的key值是否是包含两个元素的数组, 并且数组中的两个元素是否相等
ResourceContainsCondition	检查传入的key值是否出现在resource字符串中

至于如何使用这些Condition, 你可以参考 [Ladon Condition使用示例](#)。此外, Ladon还支持自定义Condition。

另外, Ladon还支持授权审计, 用来记录授权历史。我们可以通过在ladon.Ladon中附加一个ladon.AuditLogger来实现:

```
import "github.com/ory/ladon"
import manager "github.com/ory/ladon/manager/memory"

func main() {

    warden := ladon.Ladon{
        Manager: manager.NewMemoryManager(),
        AuditLogger: &ladon.AuditLoggerInfo{}
    }

    // ...
}
```

在上面的示例中, 我们提供了ladon.AuditLoggerInfo, 该AuditLogger会在授权时打印调用的策略到标准错误。AuditLogger是一个interface:

```
// AuditLogger tracks denied and granted authorizations.
type AuditLogger interface {
```

```
LogRejectedAccessRequest(request *Request, pool Policies, deciders Policies)
LogGrantedAccessRequest(request *Request, pool Policies, deciders Policies)
}
```

要实现一个新的AuditLogger，你只需要实现AuditLogger接口就可以了。比如，我们可以实现一个AuditLogger，将授权日志保存到Redis或者MySQL中。

Ladon支持跟踪一些授权指标，比如 deny、allow、not match、error。你可以通过实现ladon.Metric接口，来对这些指标进行处理。ladon.Metric接口定义如下：

```
// Metric is used to expose metrics about authz
type Metric interface {
    // RequestDeniedBy is called when we get explicit deny by policy
    RequestDeniedBy(Request, Policy)
    // RequestAllowedBy is called when a matching policy has been found.
    RequestAllowedBy(Request, Policies)
    // RequestNoMatch is called when no policy has matched our request
    RequestNoMatch(Request)
    // RequestProcessingError is called when unexpected error occurred
    RequestProcessingError(Request, Policy, error)
}
```

例如，你可以通过下面的示例，将这些指标暴露给prometheus：

```
type prometheusMetrics struct{}

func (mtr *prometheusMetrics) RequestDeniedBy(r ladon.Request, p ladon.Policy) {}
func (mtr *prometheusMetrics) RequestAllowedBy(r ladon.Request, policies ladon.Policies) {}
func (mtr *prometheusMetrics) RequestNoMatch(r ladon.Request) {}
func (mtr *prometheusMetrics) RequestProcessingError(r ladon.Request, err error) {}

func main() {

    warden := ladon.Ladon{
        Manager: manager.NewMemoryManager(),
        Metric:  &prometheusMetrics{},
    }

    // ...
}
```

在使用Ladon的过程中，有两个地方需要你注意：

- 所有检查都区分大小写，因为主题值可能是区分大小写的ID。
- 如果ladon.Ladon无法将策略与请求匹配，会默认授权结果为拒绝，并返回错误。

## iam-authz-server使用方法介绍

上面，我介绍了iam-authz-server的资源授权功能，这里介绍下如何使用iam-authz-server，也就是如何调用 /v1/authz 接口完成资源授权。你可以通过下面的3大步骤，来完成资源授权请求。

## 第一步，登陆iam-apiserver，创建授权策略和密钥。

这一步又分为3个小步骤。

1. 登陆iam-apiserver系统，获取访问令牌：

```
$ token=`curl -s -XPOST -H'Content-Type: application/json' -d'{"username":"admin","password":"Admin@2021"}'
```

2. 创建授权策略：

```
$ curl -s -XPOST -H"Content-Type: application/json" -H"Authorization: Bearer $token" -d'{"metadata":{"name"
```

3. 创建密钥，并从请求结果中提取secretID 和 secretKey：

```
$ curl -s -XPOST -H"Content-Type: application/json" -H"Authorization: Bearer $token" -d'{"metadata":{"name"
{"metadata":{"id":23,"name":"authztest","createdAt":"2021-04-08T07:24:50.071671422+08:00","updatedAt":"2021
```

## 第二步，生成访问 iam-authz-server的 token。

iamctl 提供了 jwt signin 子命令，可以根据 secretID 和 secretKey 签发 Token，方便使用。

```
$ iamctl jwt sign ZuxvXNfG08BdEMqkTaP41L2DLAr1E6Jpqoox 7Sfa5EfAPIwcTLGCfSvqLf0zZGCjF3l8 # iamctl jwt sign $
eyJhbGciOiJIUzI1NiIsImtpZCI6Ilp1eHZYTmZHMdHCZEVNcWtUYVA0MUwyRExBcmxFNkpwW9veCI6InR5cCI6IkpXVCJ9.eyJhdWQiOi
```

你可以通过 iamctl jwt show <token> 来查看Token的内容：

```
$ iamctl jwt show eyJhbGciOiJIUzI1NiIsImtpZCI6Ilp1eHZYTmZHMdHCZEVNcWtUYVA0MUwyRExBcmxFNkpwW9veCI6InR5cCI6I
Header:
{
  "alg": "HS256",
  "kid": "ZuxvXNfG08BdEMqkTaP41L2DLAr1E6Jpqoox",
  "typ": "JWT"
}
Claims:
{
  "aud": "iam.authz.marmotedu.com",
```

```
"exp": 1617845195,
"iat": 1617837995,
"iss": "iamctl",
"nbf": 1617837995
}
```

我们生成的Token包含了下面这些信息。

## Header

- alg: 生成签名的算法。
- kid: 密钥ID。
- typ: Token的类型，这里是JWT。

## Claims

- aud: JWT Token的接受者。
- exp: JWT Token的过期时间（UNIX时间格式）。
- iat: JWT Token的签发时间（UNIX时间格式）。
- iss: 签发者，因为我们是使用 iamctl 工具签发的，所以这里的签发者是 iamctl。
- nbf: JWT Token的生效时间（UNIX时间格式），默认是签发时间。

## 第三步，调用/v1/authz接口，完成资源授权请求。

请求方法如下：

```
$ curl -s -XPOST -H'Content-Type: application/json' -H'Authorization: Bearer eyJhbGciOiJIUzI1NiIsImtpZCI6Ii1  
{"allowed":true}
```

如果授权通过，会返回：{"allowed":true}。如果授权失败，则返回：

```
{"allowed":false,"denied":true,"reason":"Request was denied by default"}
```

## iam-authz-server的代码实现

接下来，我们来看下iam-authz-server的具体实现，我会从配置处理、启动流程、请求处理流程和代码架构4个方面来讲解。

## iam-authz-server的配置处理

iam-authz-server服务的main函数位于[authzserver.go](#)文件中，你可以跟读代码，了解iam-authz-server的代码实现。iam-authz-server的服务框架设计跟iam-apiserver的服务框架设计保持一致，也是有3种配置：Options配置、组件配置和HTTP服务配置。

Options配置见[options.go](#)文件：

```
type Options struct {
    RPCServer      string
    ClientCA       string
    GenericServerRunOptions *genericoptions.ServerRunOptions
    InsecureServing *genericoptions.InsecureServingOptions
    SecureServing  *genericoptions.SecureServingOptions
    RedisOptions   *genericoptions.RedisOptions
    FeatureOptions *genericoptions.FeatureOptions
    Log            *log.Options
    AnalyticsOptions *analytics.AnalyticsOptions
}
```

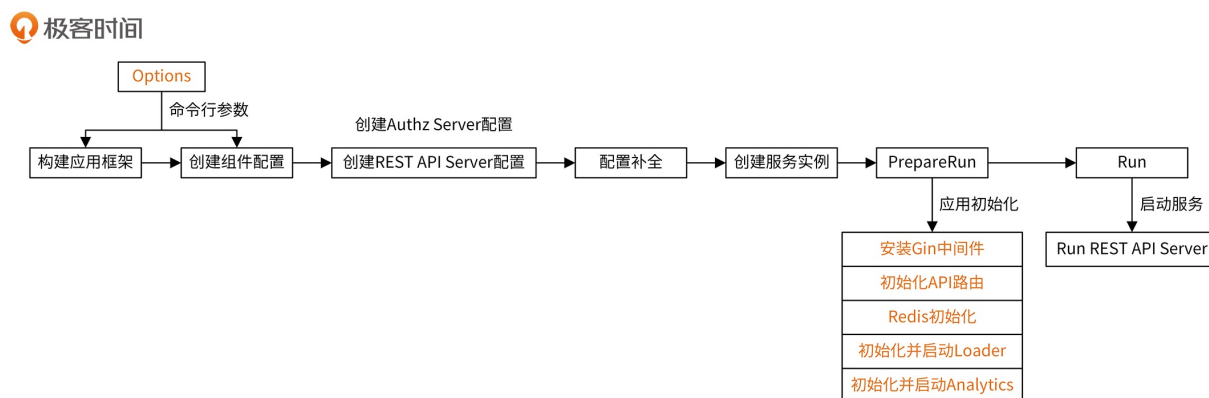
和iam-apiserver相比，iam-authz-server多了AnalyticsOptions，用来配置iam-authz-server内的Analytics服务，Analytics服务会将授权日志异步写入到Redis中。

iam-apiserver和iam-authz-server共用了GenericServerRunOptions、InsecureServing、SecureServing、FeatureOptions、RedisOptions、Log这些配置。所以，我们只需要用简单的几行代码，就可以将很多配置项都引入到iam-authz-server的命令行参数中，这也是命令行参数分组带来的好处：批量共享。

## iam-authz-server启动流程设计

接下来，我们来详细看下iam-authz-server的启动流程。

iam-authz-server的启动流程也和iam-apiserver基本保持一致。二者比较大的不同在于Options参数配置和应用初始化内容。另外，和iam-apiserver相比，iam-authz-server只提供了REST API服务。启动流程如下图所示：



## iam-authz-server 的 RESTful API请求处理流程

iam-authz-server的请求处理流程也是清晰、规范的，具体流程如下图所示：



**首先**，我们通过API调用（<HTTP Method> + <HTTP Request Path>）请求iam-authz-server提供的RESTful API接口 POST /v1/authz。

**接着**，Gin Web框架接收到HTTP请求之后，会通过认证中间件完成请求的认证，iam-authz-server采用了Bearer认证方式。

**然后**，请求会被我们加载的一系列中间件所处理，例如跨域、RequestID、Dump等中间件。

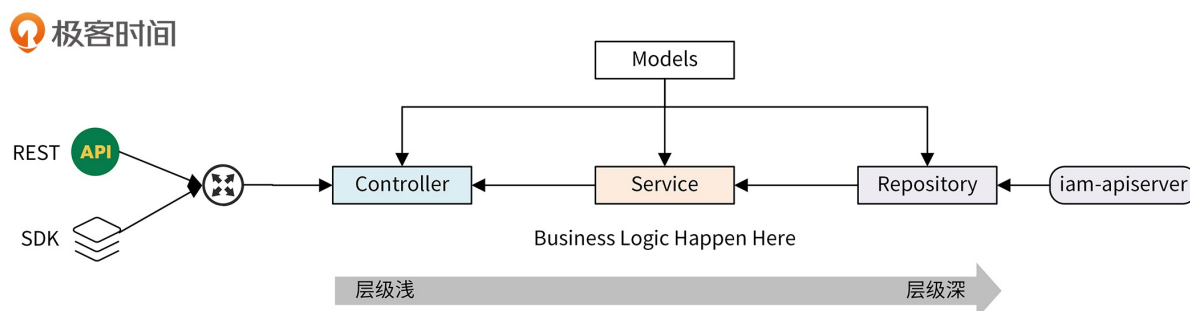
**最后**，根据<HTTP Method> + <HTTP Request Path>进行路由匹配。

比如，我们请求的RESTful API是POST /v1/authz，Gin Web框架会根据 HTTP Method 和 HTTP Request Path，查找注册的Controllers，最终匹配到 [authzController.Authorize](#) Controller。在 Authorize Controller中，会先解析请求参数，接着校验请求参数、调用业务层的方法进行资源授权，最后处理业务层的返回结果，返回最终的 HTTP 请求结果。

## iam-authz-server的代码架构

iam-authz-server的代码设计和iam-apiserver一样，遵循简洁架构设计。

iam-authz-server的代码架构也分为4层，分别是模型层（Models）、控制层（Controller）、业务层（Service）和仓库层（Repository）。从控制层、业务层到仓库层，从左到右层级依次加深。模型层独立于其他层，可供其他层引用。如下图所示：



iam-authz-server 和 iam-apiserver 的代码架构有这三点不同：

- iam-authz-server客户端不支持前端和命令行。



- iam-authz-server仓库层对接的是iam-apiserver微服务，而非数据库。
- iam-authz-server业务层的代码存放在目录[authorization](#)中。

## iam-authz-server关键代码分析

和 iam-apiserver 一样，iam-authz-server也包含了一些优秀的设计思路 and 关键代码，这里我来一一介绍下。

### 资源授权

先来看下，iam-authz-server是如何实现资源授权的。

我们可以调用iam-authz-server的 /v1/authz API接口，实现资源的访问授权。 /v1/authz 对应的 controller方法是[Authorize](#)：

```
func (a *AuthzController) Authorize(c *gin.Context) {
    var r ladon.Request
    if err := c.ShouldBind(&r); err != nil {
        core.WriteResponse(c, errors.WithCode(code.ErrBind, err.Error()), nil)

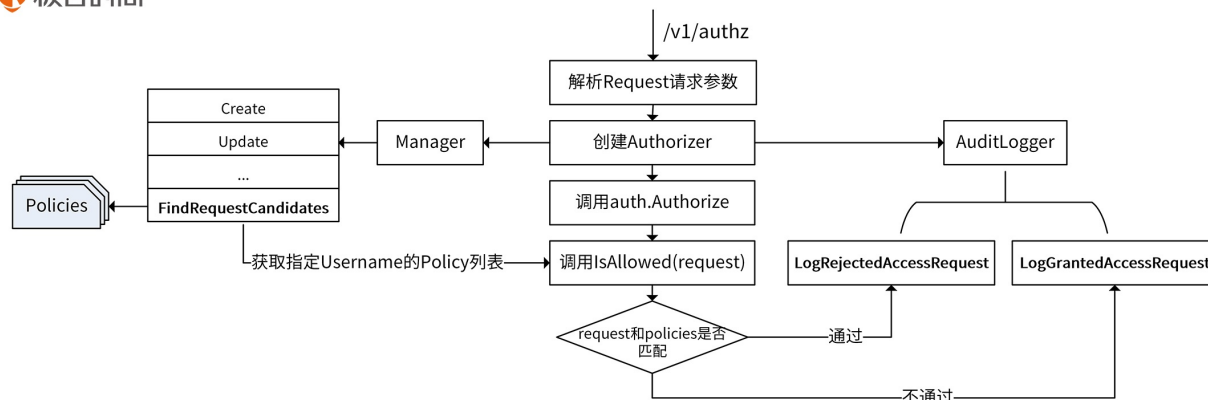
        return
    }

    auth := authorization.NewAuthorizer(authorizer.NewAuthorization(a.store))
    if r.Context == nil {
        r.Context = ladon.Context{}
    }

    r.Context["username"] = c.GetString("username")
    rsp := auth.Authorize(&r)

    core.WriteResponse(c, nil, rsp)
}
```

该函数使用 [github.com/ory/ladon](https://github.com/ory/ladon) 包进行资源访问授权，授权流程如下图所示：



具体分为以下几个步骤：

第一步，在Authorize方法中调用 `c.ShouldBind(&r)`，将API请求参数解析到 `ladon.Request` 类型的结构体变量中。

第二步，调用[authorization.NewAuthorizer](#)函数，该函数会创建并返回包含Manager和AuditLogger字段的[Authorizer](#)类型的变量。

Manager包含一些函数，比如 Create、Update和FindRequestCandidates等，用来对授权策略进行增删改查。AuditLogger包含 LogRejectedAccessRequest 和 LogGrantedAccessRequest 函数，分别用来记录被拒绝的授权请求和被允许的授权请求，将其作为审计数据使用。

第三步，调用[auth.Authorize](#)函数，对请求进行访问授权。auth.Authorize函数内容如下：

```
func (a *Authorizer) Authorize(request *ladon.Request) *authzv1.Response {
    log.Debug("authorize request", log.Any("request", request))

    if err := a.warden.IsAllowed(request); err != nil {
        return &authzv1.Response{
            Denied: true,
            Reason: err.Error(),
        }
    }

    return &authzv1.Response{
        Allowed: true,
    }
}
```

该函数会调用 `a.warden.IsAllowed(request)` 完成资源访问授权。IsAllowed函数会调用 `FindRequestCandidates(r)` 查询所有的策略列表，这里要注意，我们只需要查询请求用户的policy列表。在Authorize函数中，我们将username存入ladon Request的context中：

```
r.Context["username"] = c.GetHeader("username")
```

在[FindRequestCandidates](#)函数中，我们可以从Request中取出username，并根据username查询缓存中的policy列表，FindRequestCandidates实现如下：

```
func (m *PolicyManager) FindRequestCandidates(r *ladon.Request) (ladon.Policies, error) {
    username := ""

    if user, ok := r.Context["username"].(string); ok {
        username = user
    }

    policies, err := m.client.List(username)
    if err != nil {
        return nil, errors.Wrap(err, "list policies failed")
    }
}
```

```

ret := make([]ladon.Policy, 0, len(policies))
for _, policy := range policies {
    ret = append(ret, policy)
}

return ret, nil
}

```

IsAllowed函数代码如下：

```

func (l *Ladon) IsAllowed(r *Request) (err error) {
    policies, err := l.Manager.FindRequestCandidates(r)
    if err != nil {
        go l.metric().RequestProcessingError(*r, nil, err)
        return err
    }

    return l.DoPoliciesAllow(r, policies)
}

```

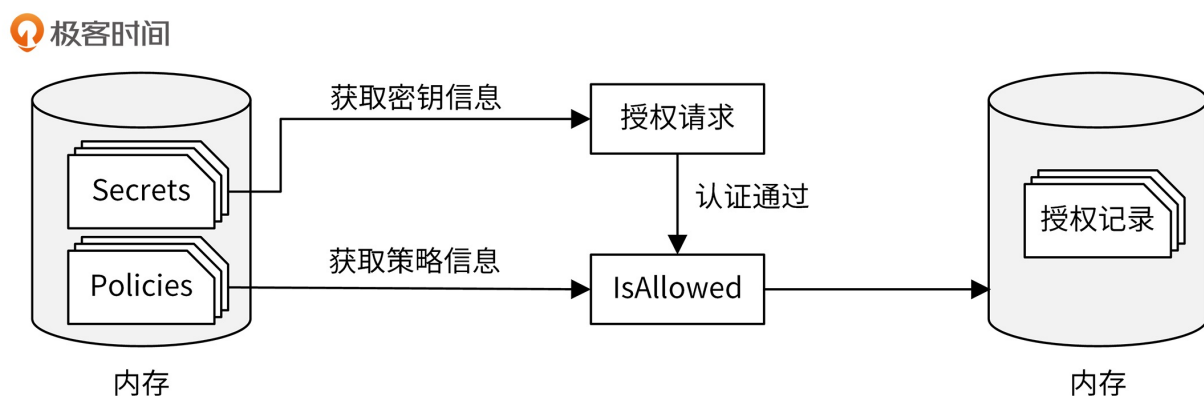
IsAllowed会调用 DoPoliciesAllow(r, policies) 函数，来匹配policy列表。如果policy列表中有一条记录跟request不匹配，就调用LogRejectedAccessRequest函数记录拒绝的请求，并返回值为非nil的error，error中记录了授权失败的错误信息。

如果所有的policy都匹配request，则调用LogGrantedAccessRequest函数记录允许的请求，并返回值为nil的error。

为了降低请求延时，LogRejectedAccessRequest和LogGrantedAccessRequest会将授权记录存储在Redis中，之后由iam-pump进程读取Redis，并将授权记录持久化存储在MongoDB中。

## 缓存设计

iam-authz-server主要用来做资源访问授权，属于数据流的组件，对接口访问性能有比较高的要求，所以该组件采用了缓存的机制。如下图所示：



iam-authz-server组件通过**缓存密钥和授权策略信息**到内存中，加快密钥和授权策略的查询速度。通过**缓存授权记录**到内存中，提高了授权数据的写入速度，从而大大降低了授权请求接口的延时。

上面的缓存机制用到了Redis key-value存储，所以在iam-authz-server初始化阶段，需要先建立Redis连接（位于[initialize](#)函数中）：

```
go storage.ConnectToRedis(ctx, s.buildStorageConfig())
```

这个代码会维护一个Redis连接，如果Redis连接断掉，会尝试重连。这种方式可以使我们在调用Redis接口进行数据读写时，不用考虑连接断开的问题。

接下来，我们就来详细看看，iam-authz-server是如何实现缓存机制的。

### 先来看下密钥和策略缓存。

iam-authz-server通过[load](#)包来完成密钥和策略的缓存。

在iam-authz-server进程启动时，会创建并启动一个Load服务（位于[initialize](#)函数中）：

```
load.NewLoader(ctx, cacheIns).Start()
```

**先来看创建Load服务。**创建Load服务时，传入了cacheIns参数，cacheIns是一个实现了[Loader](#)接口的实例：

```
type Loader interface {  
    Reload() error  
}
```

**然后看启动Load服务。**通过Load实例的[Start](#)方法来启动Load服务：

```
func (l *Load) Start() {  
    go startPubSubLoop()  
    go l.reloadQueueLoop()  
    go l.reloadLoop()  
  
    l.DoReload()  
}
```

Start函数先启动了3个协程，再调用 `l.DoReload()` 完成一次密钥和策略的同步：

```
func (l *Load) DoReload() {
    l.lock.Lock()
    defer l.lock.Unlock()

    if err := l.loader.Reload(); err != nil {
        log.Errorf("failed to refresh target storage: %s", err.Error())
    }

    log.Debug("refresh target storage succ")
}
```

上面我们说了，创建Load服务时，传入的cacheIns实例是一个实现了Loader接口的实例，所以在[DoReload](#)方法中，可以直接调用Reload方法。cacheIns的Reload方法会从iam-apiserver中同步密钥和策略信息到iam-authz-server缓存中。

我们再来看下，startPubSubLoop、reloadQueueLoop、reloadLoop 这3个Go协程分别完成了什么功能。

## 1. startPubSubLoop协程

[startPubSubLoop](#)函数通过[StartPubSubHandler](#)函数，订阅Redis的 iam.cluster.notifications channel，并注册一个回调函数：

```
func(v interface{}) {
    handleRedisEvent(v, nil, nil)
}
```

[handleRedisEvent](#)函数中，会将消息解析为[Notification](#)类型的消息，并判断Command的值。如果是NoticePolicyChanged或NoticeSecretChanged，就会向 reloadQueue channel中写入一个回调函数。因为我们不需要用回调函数做任何事情，所以这里回调函数是nil。 reloadQueue 主要用来告诉程序，需要完成一次密钥和策略的同步。

## 2. reloadQueueLoop协程

reloadQueueLoop函数会监听 reloadQueue ，当发现有新的消息（这里是回调函数）写入时，会实时将消息缓存到 requeue 切片中，代码如下：

```
func (l *Load) reloadQueueLoop(cb ...func()) {
    for {
        select {
        case <-l.ctx.Done():
            return
        case fn := <-reloadQueue:
            requeueLock.Lock()
            requeue = append(requeue, fn)
            requeueLock.Unlock()
            log.Info("Reload queued")
        }
```

```

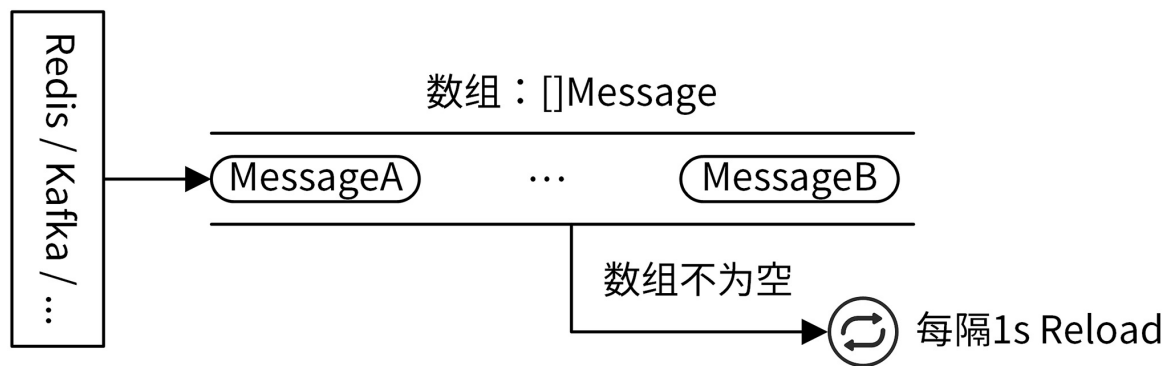
    if len(cb) != 0 {
        cb[0]()
    }
}
}
}
}

```

### 3. reloadLoop协程

通过`reloadLoop`函数启动一个timer定时器，每隔1秒会检查 `requeue` 切片是否为空，如果不为空，则调用 `l.DoReload` 方法，从iam-apiserver中拉取密钥和策略，并缓存在内存中。

密钥和策略的缓存模型如下图所示：



#### 密钥和策略缓存的具体流程如下：

接收上游消息（这里是从Redis中接收），将消息缓存到切片或者带缓冲的channel中，并启动一个消费协程去消费这些消息。这里的消费协程是`reloadLoop`，`reloadLoop`会每隔1s判断 `requeue` 切片是否长度为0，如果不为0，则执行 `l.DoReload()` 缓存密钥和策略。

讲完了密钥和策略缓存，**再来看下授权日志缓存。**

在启动iam-authz-server时，还会启动一个Analytics服务，代码如下（位于 [internal/authzserver/authzserver.go](#) 文件中）：

```

if s.analyticsOptions.Enable {
    analyticsStore := storage.RedisCluster{KeyPrefix: RedisKeyPrefix}
    analyticsIns := analytics.NewAnalytics(s.analyticsOptions, &analyticsStore)
    analyticsIns.Start()
    s.gs.AddShutdownCallback(shutdown.ShutdownFunc(func(string) error {
        analyticsIns.Stop()

        return nil
    })))
}

```

[NewAnalytics](#)函数会根据配置，创建一个Analytics实例：

```
func NewAnalytics(options *AnalyticsOptions, store storage.AnalyticsHandler) *Analytics {
    ps := options.PoolSize
    recordsBufferSize := options.RecordsBufferSize
    workerBufferSize := recordsBufferSize / uint64(ps)
    log.Debug("Analytics pool worker buffer size", log.Uint64("workerBufferSize", workerBufferSize))

    recordsChan := make(chan *AnalyticsRecord, recordsBufferSize)

    return &Analytics{
        store:          store,
        poolSize:        ps,
        recordsChan:     recordsChan,
        workerBufferSize: workerBufferSize,
        recordsBufferFlushInterval: options.FlushInterval,
    }
}
```

上面的代码创建了一个带缓冲的 recordsChan：

```
recordsChan := make(chan *AnalyticsRecord, recordsBufferSize)
```

recordsChan 存放的数据类型为[AnalyticsRecord](#)，缓冲区的大小为 recordsBufferSize（通过 --analytics.records-buffer-size 选项指定）。可以通过[RecordHit](#)函数，向recordsChan 中写入 AnalyticsRecord 类型的数据：

```
func (r *Analytics) RecordHit(record *AnalyticsRecord) error {
    // check if we should stop sending records 1st
    if atomic.LoadUint32(&r.shouldStop) > 0 {
        return nil
    }

    // just send record to channel consumed by pool of workers
    // leave all data crunching and Redis I/O work for pool workers
    r.recordsChan <- record

    return nil
}
```

iam-authz-server是通过调用 LogGrantedAccessRequest 和 LogRejectedAccessRequest 函数来记录授权日志的。在记录授权日志时，会将授权日志写入 recordsChan channel中。[LogGrantedAccessRequest](#)函数代码如下：

```
func (auth *Authorization) LogGrantedAccessRequest(r *ladon.Request, p ladon.Policies, d ladon.Policies) {
    conclusion := fmt.Sprintf("policies %s allow access", joinPoliciesNames(d))
```

```

rstring, pstring, dstring := convertToString(r, p, d)
record := analytics.AnalyticsRecord{
    Timestamp:  time.Now().Unix(),
    Username:   r.Context["username"].(string),
    Effect:     ladon.AllowAccess,
    Conclusion: conclusion,
    Request:    rstring,
    Policies:   pstring,
    Deciders:   dstring,
}

record.SetExpiry(0)
_ = analytics.GetAnalytics().RecordHit(&record)
}

```

上面的代码，会创建AnalyticsRecord类型的结构体变量，并调用RecordHit将变量的值写入 recordsChan channel中。将授权日志写入 recordsChan channel中，而不是直接写入Redis中，这可以大大减少写入延时，减少接口的响应延时。

还有一个worker进程从recordsChan中读取数据，并在数据达到一定阈值之后，批量写入Redis中。在[Start](#)函数中，我们创建了一批worker，worker个数可以通过 --analytics.pool-size 来指定。Start函数内容如下：

```

func (r *Analytics) Start() {
    analytics = r
    r.store.Connect()

    // start worker pool
    atomic.SwapUint32(&r.shouldStop, 0)
    for i := 0; i < r.poolSize; i++ {
        r.poolWg.Add(1)
        go r.recordWorker()
    }

    // stop analytics workers
    go r.Stop()
}

```

上面的代码通过 go r.recordWorker() 创建了 由poolSize 指定个数的[recordWorker](#)（worker），recordWorker函数会从 recordsChan 中读取授权日志并存入recordsBuffer中，recordsBuffer的大小为workerBufferSize，workerBufferSize计算公式为：

```

ps := options.PoolSize
recordsBufferSize := options.RecordsBufferSize
workerBufferSize := recordsBufferSize / uint64(ps)

```

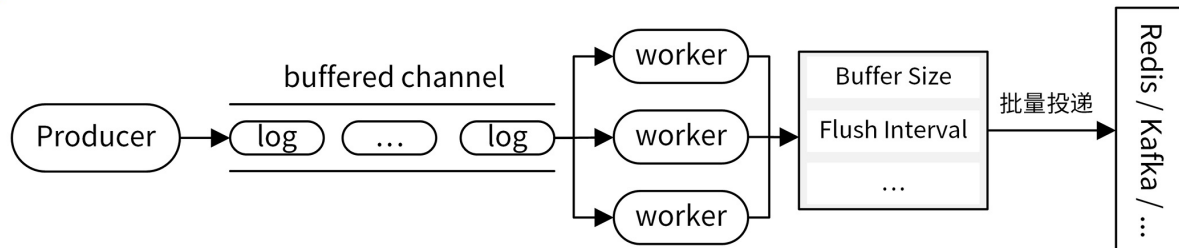
其中，options.PoolSize由命令行参数 --analytics.pool-size 指定，代表worker的个数，默认 50；options.RecordsBufferSize由命令行参数 --analytics.records-buffer-size 指定，代表缓存的授



权日志消息数。也就是说，我们把缓存的记录平均分配给所有的worker。

当recordsBuffer存满或者达到投递最大时间后，调用 `r.Store.AppendToSetPipelined(analyticsKeyName, recordsBuffer)` 将记录批量发送给 Redis，为了提高传输速率，这里将日志内容编码为msgpack格式后再传输。

上面的缓存方法可以抽象成一个缓存模型，满足实际开发中的大部分需要异步转存的场景，如下图所示：

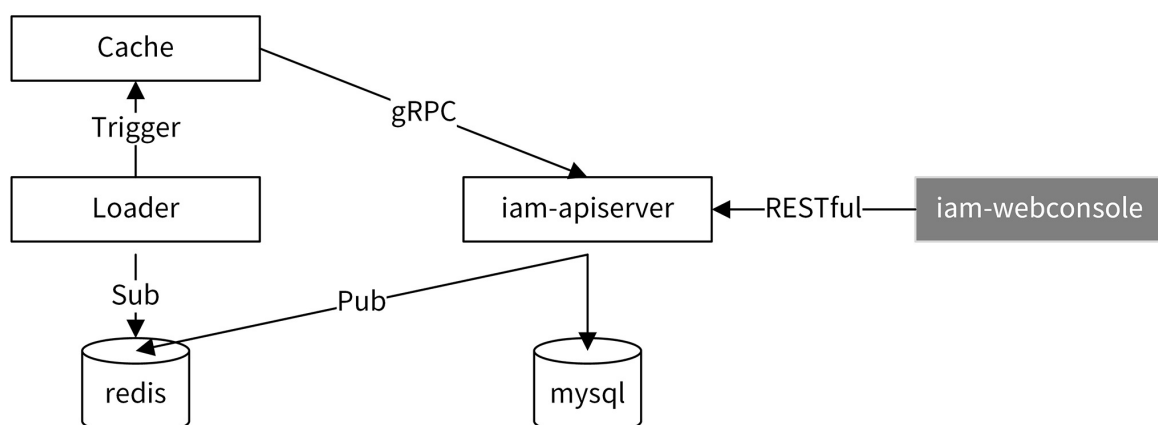


Producer将数据投递到带缓冲的channel中，后端有多个worker消费channel中的数据，并进行批量投递。你可以设置批量投递的条件，一般至少包含**最大投递日志数**和**最大投递时间间隔**这两个。

通过以上缓冲模型，你可以将日志转存的时延降到最低。

## 数据一致性

上面介绍了 iam-authz-server的 /v1/authz 接口，为了最大化地提高性能，采用了大量的缓存设计。因为数据会分别在持久化存储和内存中都存储一份，就可能会出现数据不一致的情况。所以，我们也要确保缓存中的数据 and 数据库中的数据是一致的。数据一致性架构如下图所示：



密钥和策略同步流程如下：

1. 通过iam-webconsole请求iam-apiserver创建（或更新、删除）密钥（或策略）。
2. iam-apiserver收到“写”请求后，会向Redis `iam.cluster.notifications` channel发送 `PolicyChanged`或`SecretChanged`消息。

3. Loader收到消息后，会触发cache loader实例执行 Reload 方法，重新从iam-apiserver中同步密钥和策略信息。

Loader不会关心 Reload 方法的具体实现，只会在收到指定消息时，执行 Reload 方法。通过这种方式，我们可以实现不同的缓存策略。

在cache实例的 Reload 方法中，我们其实是调用仓库层Secret和Policy的List方法来获取密钥和策略列表。仓库层又是通过执行gRPC请求，从iam-apiserver中获取密钥和策略列表。

cache的Reload方法，会将获取到的密钥和策略列表缓存在ristretto类型的Cache中，供业务层调用。业务层代码位于[internal/authzserver/authorization](#)目录下。

## 总结

这一讲中，我介绍了IAM数据流服务iam-authz-server的设计和实现。iam-authz-server提供了 /v1/authz RESTful API接口，供第三方用户完成资源授权功能，具体是使用Ladon包来完成资源授权的。Ladon包解决了“在特定的条件下，谁能够/不能够对哪些资源做哪些操作”的问题。

iam-authz-server的配置处理、启动流程和请求处理流程跟iam-apiserver保持一致。此外，iam-authz-server也实现了简洁架构。

iam-authz-server通过缓存密钥和策略信息、缓存授权日志来提高 /v1/authz 接口的性能。

在缓存密钥和策略信息时，为了和iam-apiserver中的密钥和策略信息保持一致，使用了Redis Pub/Sub机制。当iam-apiserver有密钥/策略变更时，会往指定的Redis channel Pub一条消息。iam-authz-server订阅相同的channel，在收到新消息时，会解析消息，并重新从iam-apiserver中获取密钥和策略信息，缓存在内存中。

iam-authz-server执行完资源授权之后，会将授权日志存放在一个带缓冲的channel中。后端有多个worker消费channel中的数据，并进行批量投递。可以设置批量投递的条件，例如最大投递日志数和最大投递时间间隔。

## 课后练习

1. iam-authz-server和iam-apiserver共用了应用框架（包括一些配置项）和HTTP服务框架层的代码，请阅读iam-authz-server代码，看下IAM项目是如何实现代码复用的。
2. iam-authz-server使用了[ristretto](#)来缓存密钥和策略信息，请调研下业界还有哪些优秀的缓存包可供使用，欢迎在留言区分享。

欢迎你在留言区与我交流讨论，我们下一讲见。

## 精选留言：

- daz2yy 2021-08-06 07:33:02  
老师，请问下，数据流和控制流这个怎么来理解呢，是从什么角度来定义的服务类型的？
- hello 2021-08-05 16:54:50  
老师，咨询您一个问题，使用GiN做WEB服务，微服务间通过gRPC通讯，如何选择配置注册中心，老师

能否推荐几款比较流行开源的配置注册中心。